Parcels: a Fast and Feature-Rich Binary Deployment Technology

Eliot Miranda [a] David Leibs [b] Roel Wuyts [c]

[a] *Cincom Systems*

[b] *Neometron, Inc.*

[c] *Decomp, Université Libre de Bruxelles, Belgium*

Abstract

While development of a software system is important, it is also very important to have suitable mechanisms for actually deploying code. Current state-of-the-art deployment approaches force the developer to structure the code in such a way that deployment is possible, thereby severely inhibiting reuse and comprehensibility of the system. This paper presents *parcels*, an atomic deployment mechanism for objects and source code that supports *shape changing of classes*, *method addition*, *method replacement*, and *partial loading*. The key to making this deployment mechanism feasible and fast is a *pickling* algorithm that allows the unpickling to be done iteratively instead of with a recursive descent parser. Parcels were developed for VisualWorks Smalltalk, and have been the default deployment mechanism the past years for thousands of customers.

Key words: Code Deployment, Packaging, Pickling, Smalltalk

1 Introduction

This paper considers deployment technologies that are vehicles for storing objects and their behaviour that permit their transportation *between* and importation *into* systems. From this perspective deployment technologies are an essential part of current programming practice. They provide a medium and a mechanism for upgrading, distributing or selling software, a means of physical sharing to reduce disc and memory footprint, and of logical sharing to simplify updating of multiple programs, or incremental updating of a single program.

Such deployment technologies take a number of forms such as source files, binary programs, binary shared and unshared libraries, and, in the OO world, many *pickling* formats [1,2,3,4,5]. A *pickling format* is a recursive grammar

for defining graphs of objects. An object graph is traversed and a stream of tokens in the grammar is produced that describes the graph. This is the *pickled* representation of the graph and is typically stored in a file (but can for example also be used to transfer objects across the network). A parser for the grammar can be used to reconstruct an equivalent graph, *unpickling* the objects therein. A first problem that often occurs is that the grammars encode a straightforward flattening of the elements in the graph, and consequently show a significant parsing overhead when they are read back.

A second problem shared by pickling formats is that at time of use all prerequisites of an object must be present for the object to be successfully restored. For example, a Java class file can only be loaded if its super class is already present and has a definition that matches that expected by the class file. As another example, a shared library linked against other shared libraries requires that those other libraries are available. Because all prerequisites have to be present, the system needs to be decomposable in a tree, whereas it really is a –possibly cyclic– graph. This poses problems during development, where it is sometimes needed to structure the code in such a way that it can be deployed. For example, a visitor in Java cannot be deployed independently from the classes it visits.

Parcels address the problems described above in two ways. First of all parcels use a pickling format that eliminates the need for the recursive descent parser that is normally used when unpickling. As a result the loading times are much faster. Secondly parcels support references to objects outside of the parcel, and, moreover, can be loaded even if not all prerequisites are present.

On top of that parcels can be unloaded (restoring the situation before they were loaded), support partial loading and have sophisticated mechanisms for solving loading and unloading problems, such as support to shape-change classes, method additions, and method replacements.

In short, parcels are a very fast and flexible deployment mechanism that has been enjoyed since the release of VisualWorks 3 by thousands of developers.

The rest of this paper is structured as follows. Section 2 starts by giving an overview of parcels and the features they offer, and introduces the `Color-Editing parcel` example used throughout the paper. Section 3 discusses pickling formats, and the particular one used in parcels to speed-up the loading process. Section 4 shows how parcels support the advanced loading and unloading features such as redefinition of entities, method replacements and partial loading. Section 6 validates the approach. Section 7 discusses problems with the parcels and how we plan to address them in our future work. Section 8 shows how related work handles binary code deployment. Section 9 concludes this paper.

2 Overview of Parcels

Parcels are a binary code deployment technology for VisualWorks Smalltalk that improves upon older technologies by being faster and much more flexible regarding loading and unloading. Parcels store arbitrary Smalltalk objects but are oriented towards the storage of objects that represent the initialized state of Smalltalk code modules. They allow the definition of namespaces, namespace-global variables, classes and methods. Note that methods can be defined on classes defined in that parcel, but can also be defined on classes that are part of the system the parcel is loaded in. When a method is added to an existing class in the system, we call it a *method addition*. When a method from a parcel replaces a method that already exists in the system, we talk about *method replacement*.

Loading a parcel adds the definitions from the parcel to the system, and unloading a parcel removes them. This is not without complications, because of method additions, method replacements and class redefinitions. The system has a sophisticated mechanism to ensure that code that was redefined by the parcel gets restored when the parcel is unloaded.

The parcel system manages class initialization and "uninitialization" automatically. Parcels provide a set of optional configurable actions for performing arbitrary tasks at various times in the load/unload cycle of a parcel. They include a set of prerequisite parcel names which are used to auto-load prerequisite parcels, and information such as a measure of foot-print and several kinds of optional meta-information that can be browsed without loading the parcel (such as version information, developer information or notes).

Parcels exist either as a byte stream, that may persist outside the system (typically in a disc file), or as an object in the system. In the system they may be under development, having never been written out, or loaded, in which case they are also subject to further development. The in-system form of a Parcel is an object that notes which code entities it defines, and various other properties (like the parcel's name, its prerequisites, version, comment, the methods replaced by a parcel on load, etc).

Parcels are accompanied by an optional source file that holds the source code for the binary code in the parcel. This source file is added to a registry of current source files on load, and removed from the registry on unload. This differs markedly from Smalltalk source file-ins where the filed-in source is appended to the system's changed source file, leaving a permanent side effect if the code is removed.

This paper discusses how parcels achieve this set of features. The following section introduces an example that will be used throughout the paper.

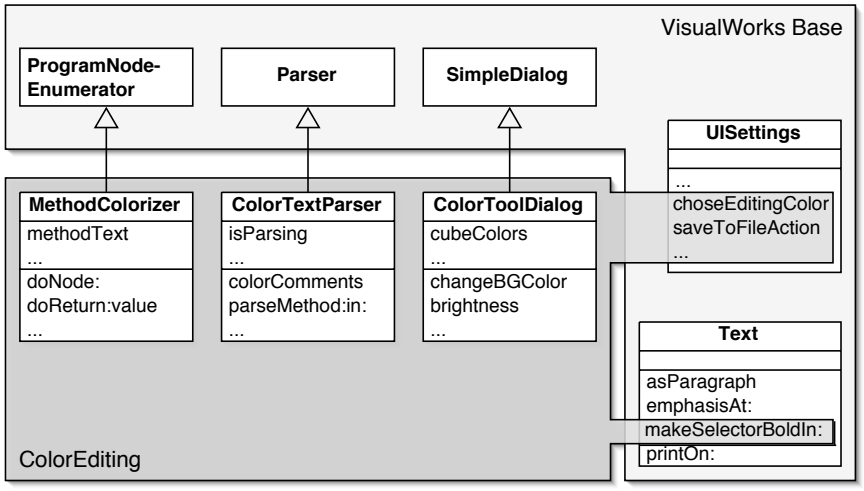

Fig. 1. Figure showing the ColorEditing parcel, that defines three classes (`MethodColorizer`, `ColorTextParser` and `ColorToolDialog`), some method additions on the base class `UISettings` and one method replacement on class `Text`

2.1 Example: The ColorEditing Parcel

Visualworks Smalltalk is distributed as a base image, containing the most commonly used features, in which parcels are loaded to extend this base image. One of these parcels allows code to be displayed in colour, highlighting syntax elements with colours. This changes the default behaviour implemented in the base image, where only the selector of a method is displayed in bold, and the method body is plain.

The ColorEditing feature is implemented as a parcel that provides a number of classes that implement its behaviour, a number of extension methods that integrate the settings a user can make in the existing Settings tool of VisualWorks (allowing colour settings to be customized from a GUI by an end-user) and one method replacement. The method replacement is needed to change the existing method that displays the selector of a method in bold, to now display it in the style the user desires. Figure 1 shows the ColorEditing parcel in a graphical format. This example is used throughout the paper to show concrete examples of parcel features.

3 Parcel Pickling Format

Parcels use a particular format to store an object graph in a file (called *pickling*). This section discusses pickling of objects in general, and then shows the format used by parcels that permits much faster loading.

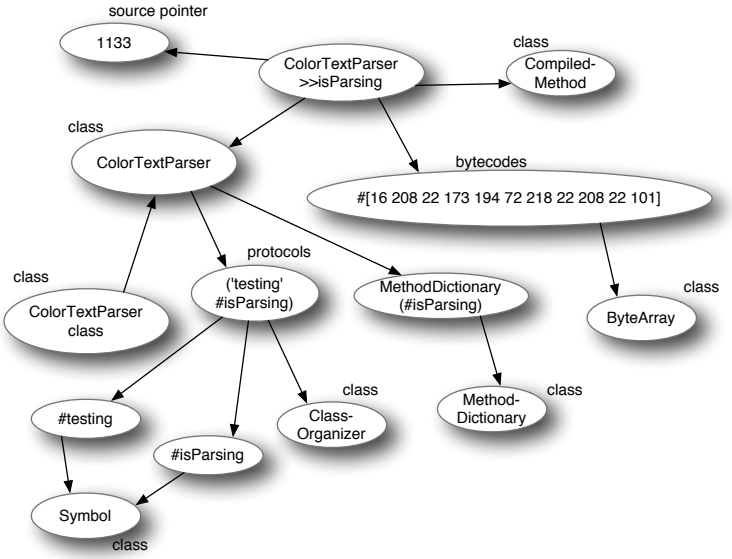

Fig. 2. Part of the object graph for class *ColorTextParser*, with a method *isParsing:*.

3.1 Current Formats for Pickling Objects

The goal of pickling is to store a graph where the nodes are objects, and the arcs are references between the objects. The nodes are typed. Possible node types depend on the language but typically include things like *primitive object types* (integers, floats, characters, strings, etc) and *arbitrary objects* (vectors of slots holding references to other objects in the graph, including their class, etc.). Figure 2 shows a small part of the object graph for the ColorEditing parcel from Figure 1, containing just some of the classes involved, and the method *isParsing:* from class ColorTextParser.

Traditional pickling formats use a stream of bytes organized as a header followed by a byte sequence that encodes an object graph according to a grammar. The grammar is designed to allow unpickling to be done as a form of bytecode interpretation. At each stage the interpreter decodes a portion of the byte sequence to yield a node in the graph. Each type of node has a particular bytecode analogous to a reserved word that introduces a phase in a programming language grammar. At each state in the parse the interpreter dispatches on the bytecode to determine how to interpret the phrase that follows and creates the object encoded by the phrase. The result of the unpickling is the object graph that was stored.

Typical optimizations for pickling formats encode common sub-ranges of the integers with their own bytecodes, etc. These techniques have a lot in common with those used for bytecoded language designs like Smalltalk itself. Although significant in reducing the size of a pickled byte stream they don't help to reduce the significant overhead of interpreting the stream.

The big deficiency with respect to speed of the traditional pickling arrangement is that it uses a recursive grammar. During the parse one may encounter an object whose class has not yet been defined, so the reference to the class will be a class definition phrase. The interpreter is faced with the difficulty of reading ahead to instantiate the class before it can instantiate the original object, and must be written as a recursive descent parser.

3.2 Pickling Objects the Parcel Way

As explained in Section 3.1, the object graph that needs to be stored uses typed nodes. In the case of VisualWorks, the nodes can be of the following type:

- *primitive object:* integer, float, character, string, etc.
- *arbitrary object:* vectors of slots holding references to other objects in the graph, including their class)
- *class definition*
- *method definition*
- *symbolic reference* to objects imported from the loading environment. Note that a symbolic reference to a class not only has the name of that class, but also the complete format of the class, such as the names of its named instance variables.

The pickling format of parcels relies on separating and ordering the descriptions of each node from the description of the references that form the arcs of the graph. Therefore the resulting file is divided into two big sections, preceded by a header: the *objects section* followed by the *references section*. Figure 3 shows how the graph from Figure 2 is stored in a parcel. It omits the header and full content of the pickled format for reasons of space.

Objects Section: The first section comprises a sequence of object descriptions clustered by class. Within the first section objects are present in a specific order. First come the symbolic references used to import the classes of all objects in the parcel whose class is not also defined by the parcel. Then come those literals represented as byte vectors, i.e. byte and two-byte strings, byte and two-byte symbols, byte arrays, floats, doubles, large positive and negative integers, and fixed-point scaled fractions. These are typically the only kinds of byte literals that occur in compiled methods. This is followed by information describing the classes defined by the parcel.

Note that classes are sorted according to their loading prerequisites. By default the loading prerequisite of a class returns the class' super class, but a user can extend this by overriding a method on the class' metaclass to return other prerequisites. A tool computes the transitive closure of the prerequisites

"refs to dependent classes"	**"Classes defined in parcel"**	**"arcs"**
'Kernel.Parser'	Kernel	1
16406	Parser	2
22	'ColorTextParser'	Kernel.ColorTextParser class
'source'	16393	Kernel
'mark'	0	#UISettings
'prevEnd'	9	...
'hereChar'	'superclass'	**"Instances of indexed objects"**
'token'	'methodDict'	OrderedCollection
'tokenType'	'format'	1
...	'subclasses'	5
"strings, bytearrays, floats, ..."	'instanceVariables'	CompiledMethod
'methodDict'	'organization'	141
'Kernel'	'name'	2
'testing'	'classPool'	1
'isParsing:'	'environment'	1
...	16413	...
#[16 208 28 173 ... 208 28 101]	7	**"compiled methods"**
#[16 208 23 173 ... 208 23 101]	29	ColorTextParser>>isParsing:
...	'source'	...
0.86	'mark'	
0.8	'prevEnd'	
...	'hereChar'	
1291792260	'token'	
1292108356	'tokenType'	
...	...	

Fig. 3. Part of the result from pickling the object graph of Figure 2. The first two columns form the objects section, while the third one is the references section. We added comments in bold to split the different parts of the sections.

relationship and does a topological sort on the set of classes to be written. This class ordering can also be done at load time, but it can take as much as one third of the complete load time for a parcel. Therefore we chose to do the computation once and store this information in the parcel itself.

References Section: The second section contains the reference information that encodes the arcs that connect the nodes. The reference information is in the same order as the objects in the object table. The first few references are those for the slots of the first object in the object table, followed by those for the slots of the second object, and so on. The VisualWorks virtual machine uses tagged pointers and encodes a 16-bit character set and a 30-bit signed 2's complement sub-range of the integers directly in object pointers. The reference information is also organized as four-byte tagged pointers that similarly encode either immediate integers, or immediate characters or the index of an object in the object table.

The parcel system achieves the above ordering by making the parcel writing process two-phase. The first phase determines the set of objects to be written. In the case of the example this results in a set containing the classes defined in the parcel (`MethodColorizer`, `ColorTextParser`, `ColorToolDialog`), objects for each of their methods, objects referenced in the methods, etc. The second phase clusters the objects collected in the first phase by class and writes them out in the order described above.

Both phases are implemented as an extension of an existing VisualWorks framework to trace object graphs (the `ObjectTracer`). The `ObjectTracer` uses the Visitor pattern [6], and so the parcel format is open to arbitrary extension. This is typically used for special purposes, for example by methods to collect their source code that should accompany the parcel file, or by `ExternalInterface` classes to include extra information such as the names of the external DLLs they provide interfaces to.

The advantage of this pickling format is that the unpickling can be done in an iterative instead of a recursive way, while still not imposing a hierarchical structure from the outset. At the end of reading the file, the objects are recreated. The following section explains this unpickling process in detail.

3.3 Unpickling a Parcel

The format described above is carefully organized to optimize load times. First of all, the organization ensures that a recursive descent parser is not needed since the class definitions are put before the instantiations of those classes. So the interpreter can batch-up instantiations of all objects of a particular class. As it instantiates objects they are placed in successive elements of an array called the *object table*. This batching provides most benefit for common object types (such as Strings or Symbols) that are the common literals of compiled methods, and for compiled methods themselves.

Once the first section has been parsed the object table is completely populated with the objects that form the nodes of the graph. The graph is knitted together by enumerating over the slots of each object in the object table, resolving its reference information into characters, integers or other objects in the table. The routine that does this operates on a single object. The routine is so simple that it is amenable to implementation as a virtual machine primitive, but performance is so good that we have yet to deploy the primitive! Performance comparisons are given in Section 6.

4 Applying a Parcel

Applying a parcel happens in several consecutive phases: the *preload phase*, the *load phase*, the *install phase* and the *postload phase*:

- *Preload Phase.* The preload phase is where the system is set up to apply a parcel. Prerequisite parcels are loaded, and preload behaviour defined in the parcel is executed.

- *Load Phase.* During the load phase, the objects defined in the parcel are restored. The load phase is done in its own context, so as not to disturb the existing system. Hence at any time prior to installation the load can be aborted and the system will not have been modified in any way.
- *Install Phase* When the load phase finishes successfully, the install phase actually installs the objects defined by the parcel in the system, performs class and global variable initialization and evaluates the parcel's post-load behaviour, if any. The installation phase is guaranteed not to fail, and is effectively atomic.
- *Postload Phase.* This phase allows all loaded parcels to install previously uninstalled method additions and replacements, and classes.

The next sections discuss the phases in detail, and especially the way load problems are handled automatically by the loader.

4.1 Parcel Preload Phase

In the preload phase, it is checked whether the prerequisite parcels of the parcel are already loaded. For each one that is not loaded, the system asks whether it is ok to load the prerequisite, with an option to do so automatically.

The system also has two hooks that allow parcels to execute code before the reading of the parcel starts (the *preread* hook) and before the actual loading starts (the *preload* hook). This fine-grained mechanism allows to test various conditions and to execute code at very specific intervals for those parcels that need it.

4.2 Parcel Load Phase

Loading the parcel means unpickling the contents of the parcel, and adding the resulting objects to a temporary context in the system. During the load phase, problems can occur. For example, the ColorEditing parcel defines the class `ColorTextParser` as a subclass of class `Parser`. When this class is not present, there is a load error. Parcels provide the following advanced features to automatically recover from certain load problems. The features are discussed in more detail afterwards.

(1) The parcel attempts to define a namespace, global, class or static that is already present in the system. This is handled by overrides.
(2) The parcel attempts to load a subclass of a class that has a different shape then that it had when the parcel was written. This is solved by

shape changing classes to effectively update the code defined in the parcel to take the new shape of the class into account.

(3) The parcel attempts to replace a method already present in the system. For this situation parcels support method replacement;

(4) The parcel attempts to use a class not present in the system, either for a method addition or to create a subclass. This is handled by *partial loading*, which boils down to remembering the code that cannot immediately be added.

We now discuss each of the solutions offered by parcels to handle these problems.

Redefinition Support

When a parcel defines a class, namespace or global that already exists, the definition in the system is replaced by the definition from the parcel. Moreover in the case of a class redefinition, existing instances are shape-changed to conform to the parcel's class.

Shape-Changing Classes The shape of a class is the named instance variables it defines. The shape of a class is important in Smalltalk, since Smalltalk methods refer to named instance variables by integer offsets. Hence changing the shape means that compiled methods have to be adjusted to make sure that they still use the correct offsets.

For example, suppose that a new release of the VisualWorks base adds an instance variable to the class `Parser`, and that a user loads the ColorEditing parcel in this new version. The `Parser` class in the system then has a different shape than the `Parser` class against which the methods were compiled. Hence the methods in the parcel need to be updated to make sure they use the correct offsets.

Such a change in definition can be detected because the shape of a class is stored in the parcel file (see the top of the left column in Figure 3 that lists the instance variables of class `Parser`). This information is used to compare the shape for the class import in the parcel against the shape of the class present in the system. When differences are detected the instances of classes that have changed shape are simply remapped, dropping the values of lost instance variables, and setting new variables to the value nil.

Method Replacement The example shows a method that is defined in the ColorEditing parcel that replaces an existing definition of that method in the system: method `makeSelectorBoldIn:` on class `Text`. Method replacement simply replaces the method with the one defined in the parcel. However, the original version of the overridden method is remembered by the system. When a parcel is unloaded it can therefore put the original methods back in place,

as will be discussed in Section 5.

Partial Loading

When deploying code it might happen that a class needed to load a parcel is absent. For example, the class `ColorEditing` adds methods to the class `UISettings`, but a user that does not need a tool for editing system settings could have removed this class. Another example is the class `ColorToolDialog`, that subclasses a class `SimpleDialog`, which again might not exist. Without support for partial loading, the ColorEditing parcel could not be loaded in such a scenario. The only thing that the user could do is to try to figure out the dependencies and create stubs for the needed classes, a tedious and difficult process.

Parcels solve this problem by *partial loading*: the parcel loader constructs a foster home for the code to live in until suitable parentage can be obtained. On encountering an import for a class that is not present the loader raises a warning which the user can respond-to either by aborting the load or by continuing. If the loader continues it creates an instance of a special class (`AbsentClassImport`), stores it at the relevant index in the object table, and initializes the object with all the format information available in the parcel (again using the class import information available in the parcel). Later on in the load attempts may be made to add methods to the `AbsentClassImport` or to subclass it:

Adding methods to a non-existent class. The case of adding methods is handled by the loader collecting these extensions and adding them to the transient properties of the loaded parcel as its uninstalled methods.

Subclassing a non-existent class. When a subclass needs to be made from an absent class, `AbsentClassImport` constructs an impostor super class, an instance of `AbsentClassImporter`, that has all the instance variables defined by the absent import and a few extra to hold information like the binding used to reference the class, and the real name of the absent super class. This impostor then continues to construct the class as for any ordinary class. Class-side code will be shape-changed to account for the extra bookkeeping information (the absent class's binding and name, etc) stored in the impostor. During the install phase of a parcel, this information is used to remember the code in the parcel that could not yet be loaded.

4.3 Parcel Install Phase

When the load phase finishes successfully, the install phase actually installs the objects defined by the parcel in the system: classes are added to their

super class's subclasses sets, globals are added to namespaces, and method additions and replacements are added to existing classes. At the same time uninstalled methods and uninstalled classes are segregated and stored in the parcel's transient properties so that they can be added should these classes become available later on. Once this is complete the system of classes and methods is in a valid state, one that could have been achieved through normal use of the programming environment.

At this point it is safe to perform class and global variable initialization. Although this initialization may cause run-time errors these will now be examples of buggy code, since all installable code has been properly installed, and the system is in a valid state. Hence the programmer is in a position to debug the problems as they would if they were developing unparceled Smalltalk code.

Lastly, the parcel's post-load action, an arbitrary block, is evaluated. Note that messages sent to instances of `AbsentClassImporter` (created during a partial load) are not invoked, since they merely wait to be installed when their prerequisites become available at a later time. The installation phase is guaranteed not to fail, and is effectively atomic.

4.4 Parcel Postload Phase

If a class that was previously absent was made available by a parcel load, then uninstalled methods and classes may become installable. During the post-load phase, all parcels in the system are checked to see whether they can now install previously uninstalled code. Method additions on absent classes are simply added to the now-present versions, making sure to check for any method replacements and adding these to the replaced method set. Uninstalled subclasses of the now-present class are asked to install themselves. The classes re-parent themselves, leaving the `AbsentClassImporters` to be garbage-collected, and their code is once again shape-changed to adjust to the loss of the extra bookkeeping information stored in the `AbsentClassImporter`.

The checking is done by sending a `#postLoad:` message to the parcel, which answers with a boolean indicating whether any uninstalled code was added. The loader continues to send `#postLoad:` to all parcels until all parcels answer false, which tells the loader that no further code was installed, and the system of parcels has reached a fixed-point. In this way the system can handle mutually recursive parcels. Figure 4 illustrates the solution. Four classes (`A`, `B`, `C` and `D`) are involved, that all inherit from eachother. Classes `A` and `C` belong to parcel `P1`, while parcel P2 contains classes `B` and `D`. Loading parcel `P1` installs only class `A` (providing its super class is present). A subsequent load of parcel `P2` will then install classes `B`, after which classes `C` and `D` get installed.

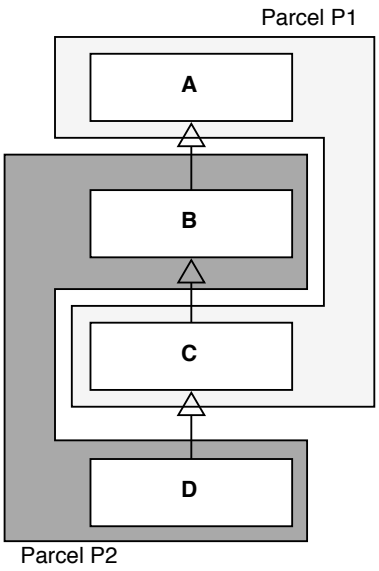

Fig. 4. Two mutually recursive parcels.

5 Unloading

The unloading of a parcel removes the definitions added by the parcel from the system, and removes the source file associated with the parcel (if there was one) from the registry of source files. Parcels tackle several unloading problems:

- When instances exist of classes that are unloaded, *obsolete classes* are created. An obsolete class is the remnant of a removed class. It continues to exist in the image for as long as either at least one compiled method references the removed class or at least one instance of the removed class is reachable. Through the usage of obsolete classes the instances can continue to function. There is one caveat when classes are removed that have subclasses. In this situation the system will prompt for confirmation, since removal of such classes causes problems. For example, removing parcel P2 from the example shown in Figure 4 results in a warning that asks for permission before continuing. If the parcel is then unloaded, problems arise for class `C`, since its super class is removed.
- When a parcel has method replacements (so-called *overrides* in parcel terminology), the code of the original methods is restored. This is possible because the original methods are remembered by the system in a central repository (the class *Override* and its subclasses), and the order in which they were added.

Application	classes/methods/extensions	format	load	write
VW-XML	48 / 543 / 0	chunk	2,615	244
		envy	2,000	900
		parcel	85	211
Soul (v. 2.3)	117 / 1,923 / 18	chunk	7,906	800
		envy	4,000	2,000
		parcel	393	676
RB	187 / 3,327 / 53	chunk	11,554	1,352
		envy	4,000	5,000
		parcel	630	1,920
Jun (v. 4.99.08)	757 / 25,212 / 0	chunk	110,729	8,694
		envy	11,000	14,000
		parcel	2,008	21,979

Table 1
First series of experiments that compare loading and writing times in VisualWorks 3 for the chunk, ENVY/Developer and parcel format (all times in milliseconds - thousands separated by a comma).

6 Validation

Over the past ten years almost all Smalltalk environments have been extended with various code storage mechanisms that have improved upon source files. This section first shows two series of benchmarks to compare the speed for reading and writing of parcels to other approaches. In a first series of experiments with VisualWorks 3, the ENVY/Developer ormat is compared to the chunk and parcel formats. A second series of experiments compares VisualWorks 7.2 parcels with the XML-chunk, VisualWorks Store, Squeak's Monticello and Dolphin's PAC format. Both series of experiments were performed on the same machine (AMD 3000+, 768 Mb of memory, Windows XP SP1). The section then finishes with an enumeration of some user benefits that show how code deployment is facilitated by the advanced loading features offered by parcels.

6.1 VisualWorks 3 Experiments

Table 1 shows the results of the first series of experiments that compares parcels with the following two formats:

- *envy* The binary format of ENVY/Developer *dat* files), the multi-user development add-on for VisualWorks [7]. We worked on a locally installed repository to eliminate network access, and took the times loading from and saving to the repository (not the importing of the code in the repository).
- *chunk format* The traditional Smalltalk textual chunk format [8].

Note that we only had access to ENVY/Developer, and that this version requires the virtual machine of VisualWorks 3. Therefore we used VisualWorks 3 for this whole first series of experiments, so that we can make a direct comparison between the numbers. We chose the following 4 applications of different sizes to do the comparisons:

- *VM-XML* [1] was a project to come to a dialect independent, XML-based file-out format. It is the smallest application we tested, having only 48 classes.
- *Soul* [9] [2], a logic programming language that lives in symbiosis with Smalltalk. We used the older version 2.3 for these tests, since it was the last version for VisualWorks 3.
- *Refactoring Browser* [10] [3] the browser that pioneered the integration of refactoring operations in a development environment.
- *Jun* [4] is a 3-D graphics framework that maps to OpenGL. It is the largest application we experimented with (consisting of 757 classes).

As can be seen in Table 1, parcels have the fastest loading speed of the formats tested. The chunk format is by far the slowest, since it is an ASCII format that is parsed on reading, adding the code to the system as the file is read. It is not a mechanism to store objects. The ENVY/Developer ormat is a bit faster than the chunk format, especially for larger applications. Regarding writing speed it can be noticed that for bigger applications the creation of parcels is slower than using "simple" formats such as the chink format. This is to be expected, since the writing process is two-phase and has to do more work.

6.2 VisualWorks 7.2 Experiments

In the following series of experiments we compared the reading and writing speed of parcels with more recent techniques from VisualWorks and other Smalltalk environments.

---

[1] version for *VW5i*, http://wiki.cs.uiuc.edu/VisualWorks/VisualWorks+XML+Framework

[2] version 2.3, http://prog.vub.ac.be/research/DMP/soul/soul2.html

[3] version for *VW30*, http://st-www.cs.uiuc.edu/users/brant/RefactoringBrowser/

[4] version 4.99.08, http://www.sra.co.jp/people/aoki/Jun/htmls/Download_e.html

Application	**classes/methods/extensions**	**format**	**load**	**write**
Classifications2	6 / 116 / 2	Store	2,194	1,095
		mcz	376	600
		PAC	238	787
		parcel	189	139
Soul (v. 3.2)	146 / 2,081 / 61	Store	17,586	16,690
	(131 / 1,834 / 51; no init)	mzc	6,799	5,000
	(129 / 1812 / 51; no init)	PAC	4,684	455
		parcel	1,514	1,677
Swazoo	101 / 6,646 / 4	Store	61,721	43,501
	(no initialization)	mcz	3,420	2,100
	(no initialization)	PAC	1,667	734
		parcel	1,764	49,479
SmallWiki	119 / 1,613 / 13	Store	12,959	8,664
	(no initialization)	mcz	5,776	4,000
	(117 / 1,539 / 8; no init)	PAC	3,016	1,095
		parcel	4,229	1,015
	(117 / 1,539 / 8; no init)	parcel	1,903	983
SIXX	37 / 271 / 100	Store	3,352	2,822
	(no initialization)	mcz	1,741	900
	(55 / 573 / 112)	PAC	1,485	4,031
		parcel	2,331	353

Table 2
Comparing loading and writing times in VisualWorks 7.2 for the chunk, Store, Monticello, PAC and parcel format (all times in milliseconds - thousands separated by a comma).

- *VisualWorks Store* Store is the multi-user development add-on for VisualWorks. Since it did not exist for VisualWorks 3, we compared it here with the other non-VisualWorks mechanisms. We installed a local *PostgreSQL* database, and used that for the experiments so that network latency was avoided. We experimented using the commonly used textual format.
- *Squeak Monticello mcz* Monticello is a distributed concurrent versioning system based on a declarative representation of Squeak source code. The *mcz* files are basically zip files that contain a manifest file giving meta-

information and a textual chunk file.
- *Dolphin PAC files* PAC files are the default mechanism for packaging code in the Dolphin environment. They are a chunk-like textual format.

For performing the comparison we took a number of applications that we could run in these different environments. Note that we had to use other applications (or other versions of applications) due to the differences between VisualWorks 3 and VisualWorks 7, such as the addition of namespaces in the latter. Note that we did not port the applications: we merely made sure we could the source code and perform measurements.

- *Classifications2* [11] [5]: The classifications model is the domain model for the StarBrowser. We used it because it is small and runs identical between Squeak and VisualWorks.
- *Soul*: A more recent version (3.2) of the language also used in the other series of experiments.
- *Swazoo* [6]: An open source Smalltalk HTTP server with resource and web request resolution framework.
- *SmallWiki* [7]: An implementation of a wiki-wiki server in Smalltalk.
- *SIXX* [8]: A XML framework for Squeak, Dolphin and VisualWorks.

The results of the experiments are shown in Table 2. Parcels and Dolphin's PAC format are the fastest mechanism for reading. The very good performance of PAC files is quite surprising, since it is basically a chunk format. Upon investigating the issue we found out that the reason is that we removed not only methods and classes that gave compatibility problems for loading (such as DLL/CC classes, that are used by VisualWorks to link with external C and C++ libraries), but also removed the initialization code that would normally be launched after filing in. The Store, chunk and parcel formats therefore do much more work: the code is loaded and then executed. To give an example about the difference this makes, we created a (defunct) SmallWiki parcel with exactly the same classes and methods as used by the Dolphin version, and removed the initialization code. The loading time then falls from 4,229 milliseconds down to only 1,903 milliseconds (versus 3,016 milliseconds for PAC files). Note that this does not explain the difference in loading speed for the SIXX parcel, but for SIXX we had completely different implementations for parcels and PAC.

Surprising is that the parcel writing is also quite fast, about on par with the writing of PAC and chunk files. We are investigating the issue of why the writing of the Swazoo parcel takes such a very long time.

[5] version 0.63, Cincom Public Store
[6] version 0.9.76-bb10, Cincom Public Store
[7] version 0.9.51, Cincom Public Store
[8] version 0.1h, http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/

Our experience has shown that the method replacement and partial loading facilities contribute significantly to improved programmer-productivity when componentising, and to the flexible configuration of deployed systems. Users of the Smalltalk environment *Visual Studio* that use SLL files and that switched to parcels reported that Parcels are much more convenient to use, because they allow for flexible packaging of the code. VisualWorks 3.0 was the first release of VisualWorks to support parcel source, method replacements and partial loading. It was also the first release to deliver all add-on modules in the form of parcels. Method replacements and partial loading were instrumental in being able to decompose the entire system and deliver it on schedule. Later released added minor enhancements.

The method replacement facility enables parcels to perform necessary base modifications on load and enables parcels that perform base modifications to be written such that they unload. Unloading makes it much easier for a programmer to try-out a parcel, and to use an extraneous tool in their development context temporarily, e.g. temporarily loading the mail tool into a development image.

The partial loading facility reduces the number of parcels that must be developed and maintained. They allow the programmer to maintain a single logical entity instead of a proliferation of physical parcels. In certain cases it may be more convenient to deploy a single parcel that in some circumstances only partially loads. The warning suppression block can be used to suppress absent class import warnings in a deployment context, allowing the parcel to load cleanly without alarming the user.

What makes the parcel system so usable is the combination of these features with the added speed benefits over other formats. This explains the popularity of parcels over other VisualWorks mechanisms as soon as they were released.

7 Future Work

The parcel system in its current state has been the code deployment mechanisms for VisualWorks since VisualWorks 3. It also served as the basis for the multi-user development system of VisualWorks (called *Store*). However, parcels have some drawbacks that we are planning to tackle in new releases.

Extending Shape-changing. The Parcel system currently has very rudimentary support for shape-change of instances. We plan to add an extensible

framework by which objects can intervene to augment the simple re-mapping shape-change done at present. Furthermore we plan to add a scheme for adding code to the current system that is used to shape-change instances in old parcels. Ted Kaehler has done a powerful scheme for Squeak's loadable component system [9]. But this scheme requires the user to provide shape-changing code at load time. While this is appropriate and useful in the hands of skilled Smalltalk developers it seems inappropriate for most deployment contexts.

Parcel Security Framework. Parcels have been used to implement a Smalltalk applet framework where code is loaded across the internet a la Java applets. However, no suitable security framework has yet been developed. We find the Java sandbox approach limiting, in that it works by restricting the full power of the system. Instead, we are investigating higher-level mechanisms, in particular proof-carrying code. Proof-carrying code approaches would require a type system such as [12]. Type inference systems are also useful when attempting to determine the boundaries of components [13]. We expect the combination of the Parcel System and a suitable type system to be very synergistic.

No lazy Loading. The standard parcel system does not support lazy loading, *à la* Java class files [14]. Currently the performance of the parcel system is so good that we have not needed to use lazy loading to improve load times, but more experience, for example in a web setting, may cause us to reconsider.

Building the underlying mechanism for a lazy-loading scheme is relatively straightforward in VisualWorks since all references to classes are through variable bindings. Therefore we can simply add and use subclasses of class `VariableBinding` that search for and load the class before returning it. However, a major issue for lazy loading is the design of the registry that maps requests for classes to the parcel to load. The current loader provides a user-specifiable set of directories in which to search for parcels when finding prerequisite parcels. But the simple set of parcel directories, while convenient in many contexts is inadequate when parcels are used to represent applets loaded over the net. Were parcel headers extended with a list of the classes they define then a policy of searching the path for the first parcel that defines the required class might suffice.

Ordering Method Additions When the loader installs method additions and method replacements, it installs all additions before any replacements, since replacements may attempt to invoke additions. One can construct artificial cases where this ordering is inadequate. While so far this policy has proven adequate in practice, it needs to be made full proof.

[9] Private communication. The results can be seen in the Squeak Smalltalk environment found at `http://www.squeak.org/`

8 Related Work

There is some older, undocumented related work. First of all, OTI extended *ENVY/Developer* (used in our benchmarks) with method loading. The resulting system is called *ENVY/App*. There were also some companies that implemented binary application loading themselves (such as Quallcomm). Since we were unable to find information on these systems, let alone obtain running systems, we were unable to compare them or benchmark them with parcels.

Java has support for pickling objects [5]. It uses a recursive approach to accomplish this. As noted in [5], "the current recursive traversal is suitable for only modest size graphs and will need to be extended to accommodate very large graphs." A similar approach is used in Python. It could be interesting to implement the pickling scheme proposed in this paper in Java or Python, and then compare the recursive approach with the non-recursive approach.

Regardless of the speed claims of the pickling format used, we do not know of other packaging mechanisms that have the loading and unloading features that parcels offer. The mechanism that comes closest is probably the *Classbox* system [15], a module system for object-oriented languages that allows *local rebinding*. Local rebinding means that a classbox (module) can make method additions and method replacements that are visible only to the classbox that defines them, without impacting the rest of the system. Such a feature is not available with classes. However, classboxes lack the meta information found in parcels (such as developer name, timestamps, etc.), storing arbitrary objects and last but not least, partial loading.

9 Conclusions

We have described a deployment technology for storing objects and their behaviour that permit their transportation between and importation into systems. The system is novel in that its binary format supports extremely fast loading and its provision of method replacements and partial loading frees the programmer from maintenance tasks required by less flexible technologies. This technology has proven itself in industrial use and underpins the product architecture of VisualWorks 3.0 and 4.0.

We have described some deficiencies of the parcel system and a number of avenues for further work to resolve these issues.

Acknowledgments

We would like to thank Alexandre Bergel, Gilad Bracha, Steve Dahl, Stéphane Ducasse, Brian Foote, and Ralph Johnson for reviewing drafts of this paper. Thanks are also due to the members of the VisualWorks development team who were and continue to be instrumental in the design and implementation of the Parcel system, and to many brave customers who have uncovered the bugs.

References

[1] S. R. Vegdahl, Moving structures between Smalltalk images, in: Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, 1986, pp. 466–471.

[2] Parcplace systems, objectworks reference guide, smalltalk-80, version 2.5, chapter 36, parcPlace Systems (1989).

[3] G. Nelson, Systems Programming With Modula-3, Prentice Hall Series in Innovative Technology, 1991.

[4] D. Ungar, Annotating objects for transport to other worlds, in: Proceedings OOPSLA '95, 1985, pp. 73–87.

[5] R. Riggs, J. Waldo, A. Wollrath, K. Bharat, Pickling state in the Java system, Computing Systems 9 (4) (1996) 291–312.
URL `http://citeseer.nj.nec.com/riggs96pickling.html`

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, Mass., 1995.

[7] J. Pelrine, A. Knight, Mastering ENVY/Developer, Cambridge University Press, 2001.

[8] G. Krasner, Smalltalk-80: Bits of History, Words of Advice, Addison Wesley, Reading, Mass., 1983.

[9] R. Wuyts, A logic meta-programming approach to support the co-evolution of object-oriented design and implementation, Ph.D. thesis, Vrije Universiteit Brussel (2001).
URL `http://www.iam.unibe.ch/~scg/Archive/PhD/Wuyts-phd.pdf`

[10] D. Roberts, J. Brant, R. E. Johnson, A refactoring tool for Smalltalk, Theory and Practice of Object Systems (TAPOS) 3 (4) (1997) 253–263.

[11] R. Wuyts, S. Ducasse, Unanticipated integration of development tools using the classification model, Journal of Computer Languages, Systems and Structures 30 (1-2) (2004) 63–77, special issue: Smalltalk Language.
URL `http://www.iam.unibe.ch/~scg/Archive/Papers/Wuyt03cClassifications.pdf`

[12] G. Bracha, D. Griswold, Strongtalk: Typechecking Smalltalk in a production environment, in: Proceedings OOPSLA '93, ACM SIGPLAN Notices, Vol. 28, 1993, pp. 215–230.

[13] O. Agesen, D. Ungar, Sifting out the gold — delivering compact applications from an exploratory object-oriented programming environment, in: Proceedings OOPSLA '94, LNCS, Springer-Verlag, 1994.

[14] S. Liang, G. Bracha, Dynamic class loading in the Java virtual machine, in: Proceedings of OOPSLA '98, 1998.

[15] A. Bergel, S. Ducasse, R. Wuyts, Classboxes: A minimal module model supporting local rebinding, in: Proceedings of JMLC 2003 (Joint Modular Languages Conference), Vol. 2789 of LNCS, Springer-Verlag, 2003, pp. 122–131, best paper award.
URL
`http://www.iam.unibe.ch/~scg/Archive/Papers/Berg03aClassboxes.pdf`