JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, No. 1, May-June 2002

Adding Dynamic Interfaces to Smalltalk

Benny Sadeh, Independent Consultant, Mindsmiths
Stéphane Ducasse Software Composition Group University of Berne, Switzerland

The concept of interfaces is central to object-oriented methodologies and is one of the most attractive features of Java and COM. Although Smalltalk always had interfaces implicitly, in Smalltalk interfaces are not first-class objects: they cannot be conversed with, referred to, or reflected upon. Consequently, Smalltalkers have been deprived of such an important and useful tool.

Since a fundamental feature of Smalltalk is that just about everything in the language is an implementation feature, explicit, static interfaces can be added to Smalltalk using Smalltalk itself with ease. However, such an addition would short-change the powerful dynamic aspects of Smalltalk.

In this article we present SmallInterfaces; a new ontology of dynamic interfaces which makes a powerful use of the dynamic nature of Smalltalk. SmallInterfaces adds interfaces as honorary members to Smalltalk's extensive reflection mechanism, in a manner portable across the many Smalltalk variants

.

1 INTRODUCTION

The term Interface is central to object-oriented methodologies [Reen96] and object foundation [Abad96], and has been recently popularised by COM and Java. Smalltalk had interfaces implicitly from its beginning. However, since Smalltalk does not have interfaces as first-class objects, they cannot be conversed with, referred to, or reflected upon. Because interfaces has proven to be extremely useful in supporting program understanding and facilitating the transition from a conceptual design to a concrete implementation, the lack of explicit interfaces in Smalltalk deprives Smalltalk developers of such an important and useful tool.

Since a fundamental feature of Smalltalk is that just about everything in the language is an implementation feature, explicit interfaces can be added to Smalltalk using Smalltalk itself with relative ease. However, since Smalltalk is not merely a language but a live, dynamic environment, adding static interfaces would lead to an *ad-hoc* solution. Moreover, because Smalltalk is also an environment, every solution which extends Smalltalk with interfaces has to integrate them into the Smalltalk IDE as well. On top of this challenge, since there exist multiple Smalltalk dialects, it is preferable that such a solution is portable across Smalltalk dialects and facilitates the exchange of interfaces in and out of the Smalltalk environment.

This paper describes how one solution, SmallInterfaces, has addressed the challenge and extended Smalltalk with explicit *dynamic* interfaces in a portable fashion. Moreover, the solution presented here does not limit itself to reproducing static interfaces in Smalltalk but *defines a new ontology* of interfaces. This ontology empowers interfaces to become dynamic and adaptive objects completely causally connected with their environment [Mae87], a definition which adapts better to Smalltalk's dynamic nature. SmallInterfaces is a freeware add-on to Smalltalk developed by Benny Sadeh, and so far has been ported to VisualWorks, VisualAge, Squeak, Smalltalk X, and GemStone variants of Smalltalk. For downloads and further details see: http://brain.cs.uiuc.edu/VisualWorks/SmallInterfaces.

The structure of this paper is as follows. We briefly describe what are interfaces, then we present a new ontology of dynamic interfaces for Smalltalk. We then illustrate how such interfaces solve different practical problems encountered by software developers, then follow with a description of how dynamic interfaces are integrated within the Smalltalk IDE. Finally, we compare other approaches before concluding.

2 INTERFACES

Interfaces are fundamental aspects of object-oriented programming [Cann89], [Abad96]. The term *Interface* is central to object-oriented methodologies and is one manifestation of what commonly referred to as *Type*. Unlike a *Class*, which is a concrete type, an interface is an abstract type. An interface specifies messages an object will understand but has no method implementations for those messages, where a class specifies how those messages will be executed by having concrete method implementations for those messages.

As a language level construct an interface contains a set of method declarations, each being a method signature. The make of a signature varies substantially across languages. In its widest form a signature consists of a name, an ordered list of qualified incoming arguments (parameters), a qualified return argument (result), a set of possibly thrown exceptions, and a set of possibly triggered events. There are many opinions about what is the correct composition of a method signature; the question is open for debate and is outside the scope of this paper. SmallInterfaces takes the position that for Smalltalk it is sufficient that a signature is of the simplest form: it is solely the method selector (In Smalltalk the method selector implicitly includes the number of parameters). It does not include the incoming parameters types or the returned result's type as Smalltalk is dynamically typed.

Object-Oriented languages differ on how they facilitate interfaces. Some, like C++, fold interfaces implicitly into the Class construct (abstract virtual class). Some, like Objective-C and Java, provide Interface and Class explicitly as two distinct constructs.

And yet others, like Smalltalk, provide interfaces implicitly via polymorphic message sends.

The benefits of distinguishing interfaces from classes are not new; interfaces have proven to be extremely useful in facilitating the transition from a conceptual design to a concrete implementation using a role based approach [Cive93a], [Reen96a], [Rieh98], [Kend99], [Rieh00]. Interfaces are used mainly in the following contexts. Firstly, within OO methodologies where roles in the analysis level can be mapped naturally to interfaces in the implementation level. Secondly, in statically typed languages like Java, interfaces allow different objects implementing a common interface to be manipulated via this interface's point of view, therefore facilitating polymorphism. Thirdly, in component-ware interfaces establish which components can talk to which and about what, to facilitate pluggability [Com], [Yell94]. Fourthly, interfaces facilitate better code comprehension.

3 THE OPPORTUNITY: A NEW ONTOLOGY OF INTERFACES FOR SMALLTALK

Smalltalk environments traditionally have lacked support for explicit interfaces. However, since a fundamental feature of Smalltalk is that just about everything in the language is an implementation feature, explicit interfaces can be added to Smalltalk using Smalltalk and its reflective capabilities itself with relative ease [Foot89]. Hence adding interfaces to Smalltalk does not require to modify the language grammar.

Porting a concept from one domain to another presents an opportunity for redefinition; constraints that existed in the source domain might annul themselves in the target domain. The mere transformation might raise possibilities that did not exist in the source domain. When trying to incorporate the explicit interface concept into Smalltalk we have two alternatives: either translate it as is based on the static typed semantics, or redefine it for a dynamically typed environment.

Looking at a static typed definition of an interface such as the one defined for Java, we find a few deficiencies:

- There is no derivation of the relationships between classes and interfaces; if a class implements all of an interface's messages but does not declare it, it is not considered to conform to it.
- There is no derivation of the relationships among interfaces; if one interface implicitly contains another, it is not exchangeable with the other. This is true even if the two interfaces are equivalent (they implement exactly the same messages).
- The interface concept does not apply to the meta level; one cannot define interfaces for class-side methods (static methods).
- Interfaces are not "pure" behavior specification. Some implementations fold other concepts within it; therefore confuse the purpose of the construct. For example, in

COM, interfaces serve as delegates (a real behavior), and in Java an interface can also be used as a tag (e.g., *Serializable*), and as a common pool of shared variables (e.g., *ObjectStreamConstants*).

SmallInterfaces chose to adapt the interface concept to Smalltalk's dynamic nature. Interfaces in Smalltalk are now tangible and adaptive objects, just like any other object in Smalltalk. The most important difference is that the relationship between interfaces and classes is *dynamically inferred by* the environment instead of being hardwired by the developer.

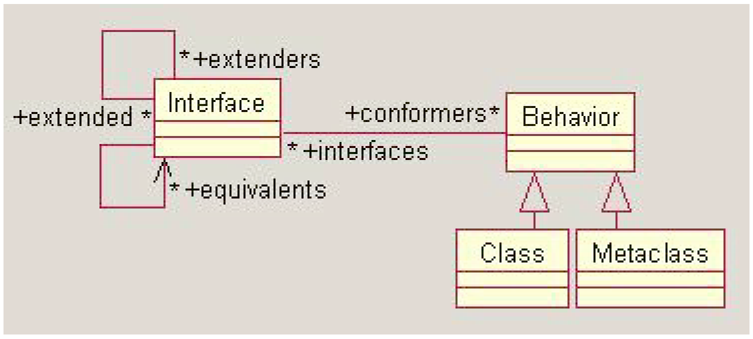


Figure 1: Relationships between Interfaces and Classes.

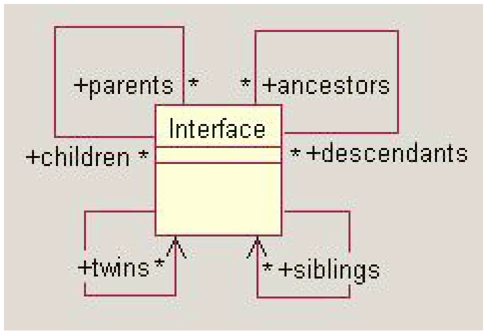


Figure 2: Relationships among Interfaces.

In addition to adding interfaces as honorary members to Smalltalk's extensive reflection mechanism, SmallInterfaces defines a new ontology of interfaces for Smalltalk, which is quite different from the one defined by other languages as shown by the Figures 1 and 2.

This redefinition of interfaces has the following properties:

a) Each interface specifies a set of messages, which together constitutes its *repertoire*.

b) Repertoires are not mutually exclusive; a message can be part of many repertoires.

c) A behavior (class or metaclass) is a *conformer* of an interface if it understands its entire repertoire. This is a many-to-many relationship: a behavior can conform to many interfaces, and an interface can be understood by many behaviors.

d) An `anObject isTypeOf: type` is true if `type` is its class (or one of its superclasses), or one of the interfaces its class conforms to.

e) An interface can be composed of other interfaces. A composite interface is an interface that extends other interfaces; it contains messages from the interfaces it extends, and may add additional messages of its own. A composite interface also referred to as an *extending* interface. These classifications are not mutually exclusive; an interface can be *extended* and *extending* at the same time.

f) Interfaces form heterarchies. They relate to each other using the family tree metaphor. An interface can have *parents* - the immediate interfaces it extends, and *ancestors* - the progenitor of its family lineage. An interface can have *children* - the immediate interfaces extending it, and *descendants* - family lineage emanating from it. An interface can have *twins* - the interfaces equivalent to it, *siblings* - the interfaces who share all of its parents, and *stepsiblings* - the interfaces that share some of its parents.

g) At the top of the heterarchy[1] are root interfaces, which are parentless interfaces; they extend no other interfaces. At the bottom of the heterarchy are the leaf interfaces which are childless interfaces; no other interfaces extend them. These classifications are not mutually exclusive (Consider the case where an environment contains a single interface - that interface is both a root and a leaf at the same time.)

h) An interface with no repertoire is considered empty, and from the system point it is transparent; both meaningless and harmless.

i) Of equivalence and containment: an interface is equivalent to another if their repertoires are equal. An interface may be defined via an aggregation of other interfaces.

j) Within a universe of objects (such as the Smalltalk image), interfaces form acyclical directed graphs that are not necessarily connected. This universe is dynamic: The inter-relationships among interfaces may change whenever an interface is added to or removed from this universe, or whenever a message is added to or removed from an interface. Likewise are the relationships between classes and interfaces. So the relationships between classes and interfaces as well as among interfaces *are always inferred* based on the actual make of classes and interfaces in the current time in a particular universe.

Now that we have introduced the notion of dynamic interfaces, we show how they can be used to bridge the transition from design to implementation and support program understanding.

[1] Heterarchy: a form of organization resembling a network or fishnet. [Web Dictionary of Cybernetics and Systems]

4 BRIDGING THE GAP BETWEEN DESIGN AND IMPLEMENTATION

In the realm of domain analysis, the use of roles has emerged as an important technique for classifying and describing collaborations among groups of objects [Reen96a], [Rieh98]. Roles reflect the various parts an object may play within a scenario it participates in and are used extensively in Aspect-Oriented Programming design [Kend99]. Roles also serve as a metaphor for communicating object-oriented software designs and recognition of their importance has grown in recent years. For example, the codification of object-oriented software design knowledge using Design Patterns is founded in part on using the metaphor of roles, which describe reusable collaborations between design elements [Gamm95]. Each design element plays an identifiable role with well-defined responsibilities. For example, the Observer pattern describes a collaboration involving two roles: Subject (or Observable) and Observer. An object may be a Subject in one scenario, and an Observer in another. Or both within one scenario.

A design process and a programming language work well together when there is support for a clear translation from the design's conceptual units to those of the language. Interface is the programming language level mechanism which maps well to Role in the design domain. So introducing interfaces in Smalltalk would ease the translation from a Role-based design to implementation.

Declaring an Interface

An implementation of the Observer Pattern in Smalltalk requires that Observer is responsible to understand the message `update:`, while Subject should propagate change events via `changed:` and manage its dependents (observers) using `addDependent:` and `removeDependent:`. How can we use interfaces to implement this pattern?

We create an Observable interface by first evaluating the definition:

```
Interface subclass: #Observable
     instanceVariableNames: ''
     classVariableNames: ''
     poolDictionaries: ''
     interfaces: ''
```

in the Smalltalk IDE and later adding interactively method definitions (with or without comments) for each message in the Observable's repertoire. Alternatively we can create the interface and its repertoire by evaluating:

```
Interface
     newNamed: #Observable
     withSelectors: #(addDependent: removeDependent:
     changed:)
```

Inferring Conformers Dynamically

We can now query for all classes which conform to the Observable interface either by executing: `Observable conformers` or by using the Interface Browser. Note that every time an interface repertoire is changed its conformers are dynamically inferred. Hence if we now execute: `Object interfaces` we get a collection containing the Observable interface.

And after defining the Observer interface as follow:

```
Interface
     newNamed: #Observer
     withSelectors: #(update:)
```

the definition of the class `Object` would now reflect the fact that in Smalltalk any `Object` can play both the Observer and Observable roles:

```
nil subclass: #Object
     instanceVariableNames: ''
     classVariableNames: 'DependentsFields EventHandlers '
     poolDictionaries: ''
     category: 'Kernel'
     interfaces: 'Observable Observer '
```

First Class Interfaces and Stub Generation

As Smalltalk is dynamically typed, type information is not used to validate the correctness of the arguments at compile time. However, using interface names when naming method arguments is a good way to convey type information. For example, the following code shows that the method `update:with`: takes as first argument an object that should understand the interface Observer.

```
Object>>update: anObserver with: aSubject
     ….
```

Documenting code as shown by the preceding example was already possible without explicit support for interfaces. However having tangible interfaces is important because it allows the programmer to browse and identify correct definition of an interface, and validate if a given class implements an interface. Using the *Interface Browser* or methods such as conformers and types, the reader can further investigate the intention of those interfaces as shown in subsequent sections. Moreover, with explicit interfaces we can support the automatic generation of stub methods. Hence executing: `SomeClass implements: Observer` generates the appropriate stub methods; the methods contained in the Observer that are not already implemented by its class hierarchy.

Supporting Documentation Coherence

With the following example we show how the use of explicit and dynamic interfaces can help with program understanding and having a coherent documentation.

The Problem with Implicit Interfaces

In most current Smalltalk dialects interfaces are not a language. Instead, they are implicit within class implementations. So Smalltalk developers usually refer to an interface indirectly by saying that classes A, B, and C are “polymorphically compatible”, meaning those classes understand a certain group of messages. One of the common ways to indicate the existence of such an implicit interface is to implement an `isX` method to return true in all classes implementing an X interface. The following example taken from VisualWorks Smalltalk illustrates such a practice.

```
GenericException class>>isExceptionHandler
     "Answer if the receiver responds to the #handles:
     message as required by the exception-handling
     machinery."

          ^true

GenericException class>>isExceptionCreator
     "Answer whether the receiver understands the behavior
     of an ExceptionCreator. This includes #raiseSignal,
     #new, as well as all the behavior of an
     ExceptionHandler, such as #handles: and #accepts. An
     ExceptionCreator can create objects (via #new) that
     conform to the behavior of SignalledExceptions."

          ^true
```

The example exhibits the following issues:

- `ExceptionCreator` and `ExceptionHandler` are implicit; there are no such entities in the system.
- The relationship between `ExceptionCreator` and `ExceptionHandler`; that `ExceptionCreator` extends `ExceptionHandler`, is not evident anywhere in the system.
- The documentation of `ExceptionHandler` is inaccurate: the comment in `isExceptionHandler` says it consists of: `{handles:},` where the comment in `isExceptionCreator` says it consists of: `{handles:, accepts:, …}`.

Using Explicit Interfaces

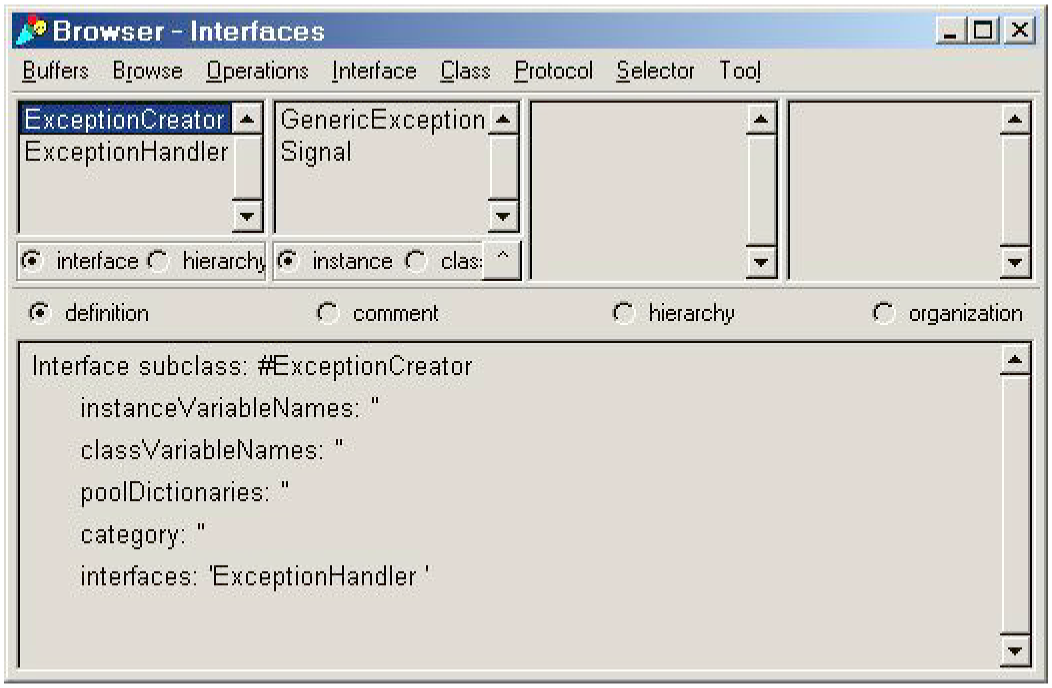
With the benefit of explicit interfaces we can now define and link `ExceptionHandler` and `ExceptionCreator` as one interface extending the other.

```
Interface
     newNamed: #ExceptionHandler
     withSelectors: #(handles: accepts:).

Interface
     newNamed: #ExceptionCreator
```

```
        extending: 'ExceptionHandler'
        additionalSelectors: #(raiseSignal new).
```

Consequently, all relationships among behaviors (classes and metaclasses) and interfaces are derived dynamically as shown by figures 3 & 4.

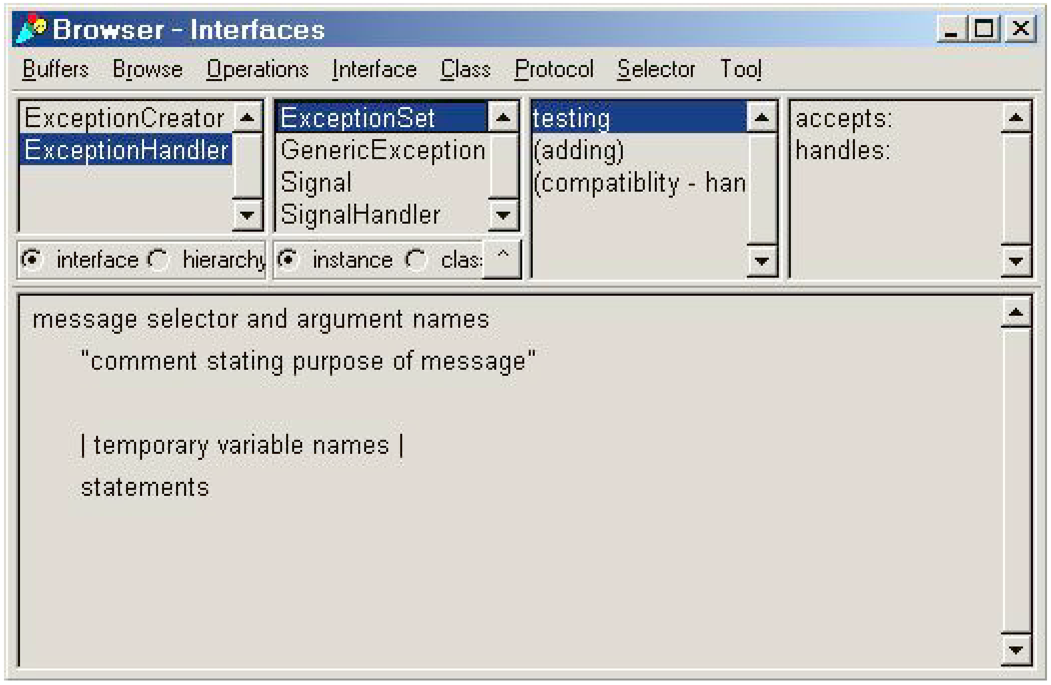


```
Interface subclass: #ExceptionCreator
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: ''
    interfaces: 'ExceptionHandler '
```

Figure 3: ExceptionCreator definition and classes implementing it.

Now given an interface, various information can be obtained, such as:

- Which interfaces extend it?
  `ExceptionHandler extenders => (ExceptionCreator)`
- Which interfaces it extends?
  `ExceptionCreator extended => (ExceptionHandler)`
- Which behaviors understand it?
  `ExceptionCreator conformers => (Signal GenericException class)`
- What messages it consists of?
  `ExceptionCreator repertoire => (#raiseSignal #new #handles: #accepts:)`



```
message selector and argument names
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

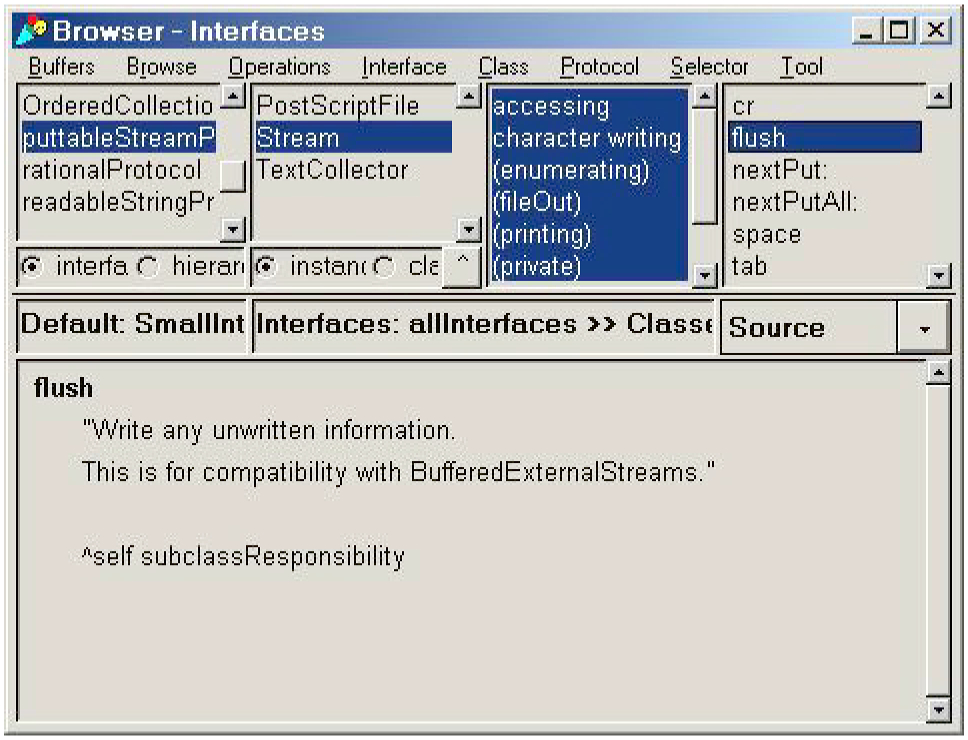
Figure 4: Interface views for classes implementing ExceptionHandler.

Supporting Software Exploration

Smalltalk is a software development environment which comes with a rich and robust library of classes. Alas, this is one of the most common complaints of newcomers to Smalltalk: there is too much out there to know (in the IDE). Where does one start? What is important to know first? This situation is not unique to Smalltalk; it is a common response which many experience when introduced to a system with average complexity.

Interfaces give us another mental navigation tool while browsing. Explicit interfaces are useful in facilitating faster learning of the Smalltalk environment. One way to do so is to supply a specific collection of interfaces to the developers to direct their exploration of a system.

Bundled with SmallInterfaces are all interface specifications described by the ANSI Smalltalk standard [Ansi]. Using those, the developer can now concentrate on exploring the classes conforming to key interfaces while viewing only the aspects of a class which pertain to a specific interface. For example (as shown in the figure below), viewing only the `puttableStreamProtocol` portion of `Stream`.



```
flush
    "Write any unwritten information.
    This is for compatibility with BufferedExternalStreams."

    ^self subclassResponsibility
```

Figure 5: puttableStreamProtocol view of Stream.

One possible path of exploration is starting from the root interfaces (execute: `Interface rootsOfTheWorld`), and then exploring the constrained interface-view of conforming classes. We can repeatedly drill down through the children of each root or maybe sidetrack exploring other interfaces of an interesting class we meet along the way.

5 IMPLEMENTATION

We now discuss the implementation choices the first author has made while designing SmallInterfaces. These choices where driven by focusing on the developer as the target audience (not the compiler), and by the desire for portability and ease of integration across all Smalltalk environments.

Interfaces as Classes

In SmallInterfaces an interface is implemented as a class, and all interfaces are direct subclasses of Interface. Consequently, an interface can be created and browsed just like any other class. That aspect makes interfaces very visible to the developer.

By choosing to implement interfaces as classes and tailoring the incremental compilation for interfaces, a developer can browse and manipulate interfaces in a familiar fashion. Since all interfaces are classes, they can be filed in and out of the image (the

Smalltalk way of exporting and importing source code). This feature solves the issue of exchanging interfaces among images and across dialects.

When an interface is defined, a class representing it is created with specific methods representing its repertoire. Hence, the following definition creates an interface with methods as shown below:

```
Interface
     newNamed: #MyInterface
     withSelectors: #(methodName …)
```

As a consequence, the body of each interface method definition is of the form:

```
MyInterface>>methodName
     "possible comment"
     ^self implementorsResponsibility
```

An Implementation Note. There is a difference between the inheritance relationship between the classes that represent interfaces and the inheritance relation (extension) between interfaces via their repertoire extension. Hence, the Observer interface inherits from the class Interface at the implementation level but it does not inherit from the Interface class repertoire: the two relationships, inheritance and repertoire extension are disconnected.

Interfaces behavior – a sampler

We briefly describe the public interface defined for interfaces.

Interface creation

An interface can be created explicitly by sending: `newNamed:` and `newNamed:withSelectors:` to `Interface`. Alternatively, an interface can be created by extending existing interfaces using: `newNamed:extending:additionalSelectors:`. An existing interface can be extended via: `,` `extend:` and `extendAll:`.

Heterarchy navigation

The interfaces related to the receiver can be queried via: `extended`, `extenders`, `parents`, `children`, `ancestors`, `descendants`, and other methods using the family tree metaphor.

Interfaces as templates

A class can be created using an interface as a template by sending one of: `asClass`, …, `asClassNamed:super:namespace:` to the interface. The same facility exists for metaclasses by using: `asMetaclass`, …. An interface can be created using a class as a template by sending one of: `asInterface` or `asInterfaceNamed:` to the class.

Conforming behaviors
A class can choose to implement an interface either by using one of the various `implement:` methods, or by including the interface's name within its class definition, as in:

```
NameOfSuperclass subclass: #NameOfClass
        instanceVariableNames: 'instVarName1 instVarName2'
        classVariableNames: 'ClassVarName1 ClassVarName2'
        poolDictionaries: ''
        interfaces: 'NameOfInterface1 NameOfInterface2'
```

Stub methods generation
When a class chooses to implements an interface, the interface generates stub methods for all messages for which there is no existing method implementation via: `createStubedMethodFor:from:`. After which, the developer fills in those stubs. One can query for stubbed methods via: `methodsStubedIn:`.

Conformance querying
The classes and metaclasses conforming to an interface can be queried via: `conformers` and `allConformers` (including subclasses). One can ask for the class' interfaces via: `interfaces`, and whether a class conforms to an interface via: `conformsTo:`.

Object typing
Object was extended with: `types`, which returns all the classes it is a member of and all the interfaces its classes implements. Similar to `isKindOf:` an object can now be queried for its type membership via: `isTypeOf:`.

Repertoire querying
Both interfaces and classes can be asked for the set of methods they implement via: `repertoire`. There are various set operations which can be performed on repertoires, such as finding the difference between the two.

On Integrating Interfaces with the Smalltalk Environment

Since Smalltalk is not merely a language but also a dynamic object environment which facilitates an incremental development process based on incremental compilation, every solution which extends Smalltalk with interfaces has to deal with the challenge of integrating it into the Smalltalk environment as well.

In such a live environment the relationships between classes and interfaces as well as the relationships among interfaces are dynamic. In particular, the causality of connections between classes and interfaces has to be preserved at any moment; when a class, an interface, or a method is added/changed/removed to/from the environment, all relevant interfaces and classes should be immediately affected by the event. This is paramount because of Smalltalk's dynamic nature. The major implication is that the web of relationships needs to be inferred from the actual composition of classes and interfaces in a given universe, for a given moment. For example, the addition of a message to a given interface can change the conformance of classes to that interface. Therefore, it needs to be synchronized with all currently extending interfaces and conforming behaviors.

The causal connection between interfaces and classes and between interfaces themselves is achieved by hooking to the change notification mechanism within the Smalltalk environment. Upon notification of any pertinent change, the relationships between affected classes and/or interfaces are re-synched.

Portability across Smalltalk Dialects

On top of this challenge, since there are multiple Smalltalk dialects in existence, it is desirable that such a solution is portable across Smalltalk dialects and facilitates the transport of interfaces in and out of the Smalltalk image (intra), and across Smalltalk environments (inter). To make things harder still, it is desirable that the solution does not cause any base changes; none of the base classes shape or existing methods should be touched.

Portability is achieved by concentrating on making no base changes, and by isolating the points of differences among the various Smalltalk environments, then rewriting the bridge code for each environment. Most of the points of contention were in the reflection layer which is not yet standardized across Smalltalk dialects. Since one of the major differences among Smalltalk dialects is their GUI implementation, the Interfaces Browser (the GUI portion) was written on top of the Refactory Browser, thus bypassing the GUI portability issue. As a result, environments that did not have the Refactory Browser available do not have the GUI portion available as well.

6 RELATED WORK

[Gott96] introduces the notion of roles in Smalltalk. However, the roles are not connected to the classes in a dynamic way. [Yell94] introduces the notion of interfaces to describe and reason about components. Object interfaces are described as a state automata and the approach is not an integration of interfaces into an object model but a proposal for a new way of describing objects. [Hail90] introduces multiple interfaces that drive the views given by an object and possible access to it. These proposed interfaces are not simply an explicit representation of the class behavior but have an extra semantics and change radically the object model. [Lamp93] introduces the idea of specialization interfaces, i.e., interfaces that describes how methods call each other so that subclass designer can reason about these calling dependencies. The very same idea was extended by Reuse Contracts [Stey96].

Dolphin Smalltalk Protocols

Dolphin Smalltalk incorporates interfaces as well, referred to as Protocols [Dolp01]. Like with SmallInterfaces, all behaviors in Dolphin can specify conformance to an interface. However, Dolphin substantially differs in its choice of interface implementation. In Dolphin, the relationship between classes and interfaces are hard coded by the developer; there is no dynamic inference. Also, every interface is an island; interfaces do not relate to each other.

Dolphin interfaces are instance-based. Because interfaces are instances, they can bare any name, therefore avoiding name collisions with classes such as Object and keywords such as nil. This is a good thing. On the other hand, they are not as visible as classes; the developer has to edit an interface using a Protocol Browser, and cannot specify argument names and comments for each of the messages. Also, interfaces cannot be referenced directly in code.

Dolphin's current implementation of interfaces does not facilitate intra- and inter-exchange of interfaces; they exist only within the context of one image. Furthermore, when a class is filed out, the knowledge of which interfaces it conforms to is lost.

SmallScript Interfaces

SmallScript is a superset of Smalltalk [SS01]. SmallScript incorporates interfaces at the language construct level, and as such have support for them within the virtual machine. Interfaces are a part of the SmallScript optional type declaration and they can be used for multi-method discrimination.

A SmallScript interface is more than an interface in the sense we define early in this paper; a SmallScript interface is actually a dynamic mixin. It is a full-fledged class that can be instantiated, unlike in SmallInterfaces where an interface is merely a behavior specification. SmallScript interfaces provide multiple inheritance via true delegation (unlike forwarding). A behavior can conform to an interface by dynamically acquiring via aggregation a concrete interface instance. Also, interfaces and their methods are generic; classes supporting a given interface can specialize interface methods at will.

This language construct is very attractive since it allows one to compose objects via aggregation; mix and match behaviors at will while avoiding the confinement which comes with composition via inheritance. Alas, since support for it has to happen in the virtual machine level, such characteristics cannot be added to all Smalltalk in a portable fashion.

7 CONCLUSION AND FUTURE WORK

Currently, there is no support for explicit and dynamically inferred interfaces within Smalltalk which is portable across all Smalltalk variants. In this article we presented

SmallInterfaces that integrate dynamic interfaces to Smalltalk's reflection layer in a portable fashion and without any base changes. The interfaces are not statically declared but inferred dynamically to fit better the dynamic nature of Smalltalk. Using SmallInterfaces, the developer has now the ability to define interfaces and to indicate which classes conform to them. SmallInterfaces integrates interfaces into the existing development environment while providing support for them on the tools level. Furthermore, it facilitates a whole slew of interactions between classes, interfaces, and objects, all in order to support program understanding, documentation, and smooth transition from analysis and design into implementation.

For a dynamically typed language like Smalltalk it is sufficient for a message declaration to specify the number of arguments. That is what SmallInterfaces does. However, there is another meaningful scheme for Smalltalk, where the type (interface or class) of each input and output arguments would be specified as well [John86]. Such a scheme would be very hard to implement across Smalltalks, and would demand the addition of a type-inference engine [Gara01], and optional typing (as done in SmallScript and CLOS) across all Smalltalks. Since SmallInterfaces focuses on helping the design and development process itself, it deemed the first and simpler approach sufficient.

REFERENCES

[Abad96] Martin Abadi and Luca Cardelli, *A Theory of Objects*, Springer, 1996, 0-387-94775-2.

[Cann89] Peter S. Canning and William Cook and Walter L. Hill and Walter G. Olthoff, *Interfaces for Strongly-Typed Object-Oriented Programming,* Proceedings of OOPSLA '89, 457-468, 1989.

[Cive93a] Franco Civello, *Roles for composite objects in object-oriented analysis and design,* Proceedings of OOPSLA '93, 376-393, 1993.

[Dolp01] Dolphin Smalltalk Document, http://www.object-arts.com/

[Foot89] Brian Foote and Ralph E. Johnson, *Reflective Facilities in Smalltalk-80,* Proceedings OOPSLA '89, 327-336, 1989.

[Gamm95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, 1995.

[Gara01] Franscico Garau, *Concrete Type Inference for Squeak*, University of Buenos Aires, 2001.

[Gott96] Georg Gottlob, Michael Schrefl and Brigitte Rock, *Extending Object-Oriented Systems with Roles*, ACM Transactions on Information Systems, 268-296, vol 14, n 3, July, 1996.

[Hail90a] Brent Hailpern and Harold Ossher, *Extending Objects to Support Multiple Interfaces and Access Control*, IEEE Transactions on Software Engineering, vol 16, n 11, 1247-1257, November, 1990.

[John86] Ralph E. Johnson, *Type-Checking Smalltalk*, Proceedings of OOPSLA '8, 315-321, 1986.

[Kend99] Elizabeth, *Role Model Design and Implementations with Aspect-Oriented Programming*, Proceedings of OOPSLA'99, 353-369,1999.

[Lamp93] John Lamping, *Typing the Specialization Interface*, Proceedings of OOPSLA '93, 201-214, 1993.

[Maes87] Pattie Maes, *Concepts and Experiments in Computational Reflection*, Proceedings of OOPSLA '87, 147-155, dec, 1987.

[Reen96] Trygve Reenskaug, *Working with Objects: The OOram Software Engineering Method*, Manning Publications, 1996, 1-884777-10-4.

[Rieh00] Dirk Riehle, *Framework Design: a Role Modelling Approach*, Swiss Federal Institute of Technology, Zurich, 2000.

[Rieh98] Dirk Riehle and Thomas Gross, *Role Model Based Framework Design and Integration*, Proceedings of OOPSLA '98, 117-133, 1998.

[SS01] SmallScript, http://www.qks.com/, http://www.smallscript.net/

[Stey96a] Patrick Steyaert, Carine Lucas, Kim Mens and Theo D'Hondt, *Reuse Contracts: Managing the Evolution of Reusable Assets*, Proceedings of OOPSLA '96, 268-285, 1996.

[Yell94a] Daniel M. Yellin and Robert E. Strom, *Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors*, 176--190, Proceedings of OOPSLA'94, 1994.

About the authors

Dr. Stéphane Ducasse is postdoctorant researcher at University of Bern. His interests are: reflective systems, components, design and implementation of languages and applications, and reengineering of object-oriented applications. He is co-author of the book Object-Oriented Reengineering Patterns. He is the main organizer of the European Smalltalk User Group conference. Email: ducasse@iam.unibe.ch.

Benny Sadeh is an independent consultant who lives in the Bay Area, California. His interests are: reflective systems, metaprogramming, frameworks, Extreme Programming (XP), and doing the right thing. He can be reached at: benny@mindsmiths.com