Design and Implementation of Smalltalk Mixin Classes

Bruno Schäffer

Ubilab

Ubilab Technical Report 98.11.1

Ubilab

Ubilab is the information technology laboratory of UBS AG. It pursues a small number of attractive, highly competitive research projects with the aim of maintaining the status of a recognized research institution.

It is the task of Ubilab to actively assist UBS with its goal of becoming a leader in the mastery of modern IT equipment, techniques, and methods. This task is pursued by means of intimate involvement in application-oriented research and advanced development projects. Furthermore, UBS contributes to fostering the interaction between IT research and application, that is between academia in its search for new methods, and business in its application of them. To this purpose Ubilab cooperates with universities and other research institutions on a worldwide basis. The goal is to expand the scope of the Laboratory by carrying out common projects, thus guaranteeing—provided top-notch partners are found—the quality of the research. The impact of Ubilab is therefore focussed on both the IT departments of UBS AG and the IT research community at large.

For further information about Ubilab, staff, projects, or publications see our World Wide Web (WWW) pages. Feel free to contact the staff personally via e-mail or to write to the mailing address given below.

Location of Ubilab

Universitätsstrasse 84
CH-8033 Zurich
Switzerland

Mailing address

UBS AG, Ubilab
Postfach
CH-8098 Zurich

Electronic access

Phone: ++41 1 236 57 14
Fax: ++41 1 236 46 71
E-mail: firstname.lastname@ubs.com
WWW: http://www.ubs.com/ubilab

Table of Contents

1 Introduction

Inheritance allows to define new classes based on already existing ones. A derived class usually spezializes state and behavior of a class. Common state and behavior can be generalized in a common base class. If a class can be derived from a single class only we call it single inheritance. In contrast, multiple inheritance combines the state and behavior of two or more classes.

Smalltalk as a language supports only single inheritance. The first implementations of Smalltalk-80 [Gold89] supported multiple inheritance too. However, the class library never made use of this feature and it was removed there after.

Multiple inheritance gained popularity in the second half of the eighties through languages like C++ [Str91] or Eiffel [Mey88]. Subsequently, the design and implementation of class libraries and frameworks like CommonPoint [Cot95] or PowerPlant [Met96] made use of it.

It was also realized that multiple inheritance incorporates a lot of semantics and its implementation is tricky. Moreover, its application has serious consequences to the structure of a class library. With single inheritance the inheritance graph is a tree, whereas with multiple inheritance it is an arbitrary graph. In most cases the design of a class hierarchy based on multiple inheritance is more difficult rather than easier. It also makes a class library much harder to understand. Furthermore, designers tend to mix together classes where no semantic heritage is present. Multiple inheritance also seduces to neglect or postpone redesign of the class hierarchy.

Inexperienced designers often use multiple inheritance when they should use composition. Composition is much cheaper in terms of semantics and more flexible, because it can be configured at run-time. Inheritance is mostly static and effects all instances at once.

In most cases designers just want to provide additional functionality in a class. Since most classes define behavior as well as state, this state is inherited too. This leads to problems where a class is multiply inherited from along the class hierarchy. The state of this class is now repeatedly realized. This might be intended in certain cases, whereas usually the state of the base class is to be shared. Therefore in C++ the designer of a class has to decide, whether the state of a base class may be repeated in a derived class or not. However, it is always questionable if a designer has to foresee all possible uses a class. This contradicts to the principle of object-oriented programming to postpone decisions as long as possible, preferably until run-time.

2 Smalltalk as a reflective environment

Smalltalk is based on a fully reflective architecture. A reflective architecture is one in which a process can access and manipulate a full, explicit, and causally connected representation of its own state. “Causally connected” means that any changes made to a process’ self-representation are immediately reflected in its actual state and behavior (see [Mae87], [Foo89]).

In Smalltalk, all information about the system is available at run-time:

- Metaclasses describe all classes, their state, behavior, inheritance relationship etc.
- Code is available in its source form and in its compiled form (byte code)

This meta information can be manipulated at run-time and in fact is done while developing with Smalltalk.

3 Mixin classes

3.1 Definition

It was soon realized that unlimited multiple inheritance introduces more problems than it tries to solves. Since in most cases it was intended to inherit behavior only, a restricted form of multiple inheritance, namely mixin-classes was coined [Bra90]. Mixin-classes are usually abstract and do not define any state. They only serve to declare (and sometimes define) a certain behavior through a set of methods. Classes that need to offer this behavior have to inherit from this mixin-class. This leads to two different kinds of classes:

- Base classes: they can define state and behavior
- Mixin classes: they can only define behavior

A class can only inherit from one primary base class and mix in any number of mixin-classes.

3.2 Application of mixin classes

3.2.1 Class libraries

A single rooted class library defines common behavior for all its classes in its root class, usually called Object. Examples of this behavior are printing, comparing, testing, copying, dependencies etc.

In statically typed languages it is necessary to declare this behavior already in the root class for compatibility reasons. However, class library designers are now in a dilemma. On the one hand they try to specify as much common behavior as possible in the root class. This increases the reusability of the root class and of classes that make use of this behavior. On the other hand the root class becomes too “heavy” and inheritors of the root class cannot choose the behavior on their own. Moreover, derived classes have to override methods they do not want to provide.

Mixin classes can provide a smaller granularity inheritors can choose from. For example not every class needs a printing or dependency behavior, so this can be mixed in in classes which need it.

Classes or methods requiring a certain behavior can be more specific in the interface by stating the necessary behavior through mixin classes. For example, a class SortedCollection demands a comparing behavior from the objects added to it. Requiring a comparing protocol, which is already defined as a mixin class, is much more descriptive than demanding a set methods like #=, #< etc.

3.2.2 The Tools and Materials Metaphor

A typical application of mixin-classes is the tools-materials metaphor (see [Bür95], [Rie95]). This metaphor presents a guideline, how to view and analyze the world and also how to derive a design. The domain of the tools-materials metaphor is the workplace of a skilled worker. His/her world is separated into tools and materials. A task is carried out by applying a tool to a material. A typical example from the banking domain is a form sheet as a material and a specialized calculator as a tool. In order to be worked on with a tool, a material has to provide a certain behavior. This behavior is called an aspect and is usually being defined using a mixin-class.

3.3 Mixin classes using state

It is not always convenient to provide an implementation of a mixin-class without any state. A mixin-class that tries to postpone any state to its implementor is sometimes too abstract and requires a lot of implementation effort by its client. This can be circumvented by declaring abstract methods for accessing state. A class inheriting from this mixin class has to realize the state and has to override those abstract accessor methods.

The designer of a mixin class always has to make a compromise between abstraction and implementation. If a mixin class is too abstract, it requires too much effort by its users in terms of implementation. On the other hand, an implementation usually restricts the usability of a mixin class, although an implementation can be adapted by overriding certain methods.

3.4 Shortcomings

Inheritance is mostly a static relationship, even in dynamic environments like Smalltalk. Therefore mixin classes are not suited when a combination of properties has to be dynamic and on a per-instance basis. In this case composition is clearly prefered to inheritance.

With regard to design patterns E. Gamma et al. [Gam95] describe two applications of multiple inheritance, namely Bridge and Adapter. In both cases the implementations lose flexibility if they are based on multiple inheritance rather than composition.

3.5 Implementations of mixin classes

3.5.1 C++

Mixin classes can be easily defined in C++ [Str91]. They are just classes without any instance variables. Such a class need not be virtual and therefore avoids a sequence of problems with virtual base classes (e. g. ambiguities, casting, breaking encapsulation, hardly understandable etc.)

Taligent published a comprehensive guide [Tal94] for developing large object-oriented software using C++. In this guide they distinguish between two categories of classes, base classes and mixin classes. Base classes represent fundamental functional objects, and mixin classes represent optional functionality. A class may inherit from zero or one base classes, plus zero or more mixin classes. A class that inherits from a base class is itself a base class. Adhering to these guidelines helps to avoid virtual base class problems and makes a class hierarchy much more comprehensible.

3.5.2 Java

Java [Arn96] supports interfaces which are basically mixin classes. However, Java interfaces declare just an interface, but cannot provide an implementation, be it abstract or not. The implementation has to be completely realized in the class supporting an interface.

3.5.3 CLOS

CLOS [Bob88] provides rich mechanisms for multiple inheritance. Any number of classes can be combined to build a new class. Hence, mixin classes can be simulated as well.

CLOS linearizes the class hierarchy in order to resolve ambiguities, which makes complex class hierarchies even less comprehensible. Moreover, slight changes to the class hierarchy can result in completely different behavior.

4 Mixin classes in Smalltalk

4.1 Rationale

Smalltalk has true dynamic binding. At compile time the compiler does not (and cannot) check whether an object will understand a message. This is done exclusively at runtime. Therefore two objects need not be derived from a common base class in order to be compatible. They just have to be able to respond to the same set of messages. Smalltalk has the convention of grouping together related methods in a protocol or method category. This is not a language feature but just a means of organizing the behavior of a

class. Tools like the system browser make use of method categories to provide a better overview of the functionality of a class.

As a consequence people tend to copy methods in Smalltalk rather than deriving classes from a common (abstract) base class. One can often find implementation inheritance rather than interface inheritance. This leads to confusing class hierarchies since inheritance does not determine compatibility alone. Moreover, copying methods or pieces of it contradicts factorization and results in unmaintainable code. For example, the comparing protocol is declared in the class Magnitude. If a class needs a comparing behavior one has either to derive it from Magnitude and override #<, #= and #hash or copy all methods from Magnitude into the new class and implement #<, #= and #hash. Both alternatives are undesirable, the first one leads to strange class hierarchies, the second one to hardly maintainable code.

Mixin classes help to factorize common behavior on a finer granularity. This way behavior is explicitly stated in a class. However, classes do not have to be derived from this class but need only to mixin this behavior.

Subsequently, we use both the terms “mixin class” and “protocol class” for such a class.

4.2 Design

Enhancing Smalltalk with mixin-classes was done on three levels:

- Two new classes, namely BaseObject and Protocol, are provided. Mixin classes have to be derived directly or indirectly from Protocol. Classes that make use of mixin classes must have BaseObject as a direct or indirect base class.
- A new metaclass, MetaBaseclass, was introduced. The metaclasses of classes derived from BaseObject are instances of MetaBaseclass. Creating or changing a class in Smalltalk is a relatively complicated process. In order to encapsulate this process and to make it as atomic as possible Smalltalk provides a class ClassBuilder. Accordingly, for BaseObject classes there is now a class BaseclassBuilder.
- Small changes and a few additions to the existing Smalltalk infrastructure were necessary.

The key to changing the message lookup in Smalltalk is the method #doesNotUnderstand. However, it only allows to adapt the method lookup after a requested method could not be found. BaseObject overrides #doesNotUnderstand such that it tries to find a corresponding method in the mixin classes. This search is performed in a depth first manner:

- The search is first performed along the primary base hierarchy (through the normal method lookup process).
- The search continues in the protocol classes of the receiver. Each protocol class is searched up to Protocol (actually up to Object, but any method in Object would have been found along the primary base hierarchy anyway).
- The search in the protocol classes is repeated along the primary base hierarchy up to BaseObject.

If an appropriate method can be found it is executed, otherwise Object>>#doesNotUnderstand is called.

Introducing a new metaclass was not the only possible solution. One could have implemented the required state and behavior as class (instance) variables resp. methods. The metaclass solution is cleaner since it keeps metaclass responsibilities off BaseObject.

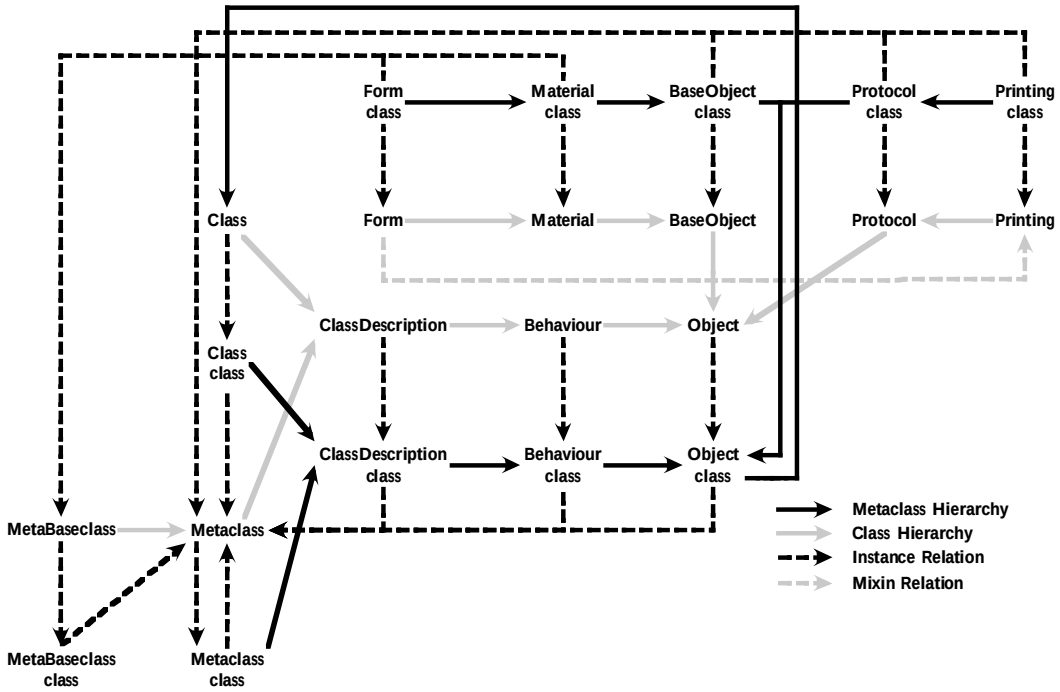

Fig. 1 Relationships between classes, metaclasses, base, and protocol classes ([Gold89])

One of the design goals was to keep the changes to the VisualWorks system minimal. This eases portability and adaption to new versions of VisualWorks. The ideal solution would have been to derive a class BaseClass from Class. All changes to Behavior, ClassDescription, and Class could have been implemented in BaseClass. However, this was not possible since the method lookup of a class method in Object continues in the instance methods of Class, but should have continued in BaseClass. All methods in BaseClass would have to determine whether the receiver is derived from BaseObject or from Object in order to provide the appropriate behavior. Hence it was decided to change a few (3 methods) in Behavior, ClassDescription, and Class. Moreover, two methods were added to Class, namely #subclass:protocol:instanceVariableNames: classVariableNames: poolDictionaries:category: and #subprotocol:category:. These methods are required in order to create new classes derived from BaseObject resp. Protocol.

A new class using mixin classes is defined as follows:

```
BaseObject subclass: #Customer
    protocol: 'Comparing Printing'
    instanceVariableNames: 'id firstName lastname address'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Example'
```

A new class using mixin classes is defined as follows:

```
Protocol subprotocol: #Comparing
    category: 'Common Protocols'
```

Apart from the metaclasses the programming tools had to be adapted as well. The user now has additional menu entries in the class list to select different class definition templates. The editor is state based and therefore it was also necessary to introduce additional states (and methods) for the new class definitions.

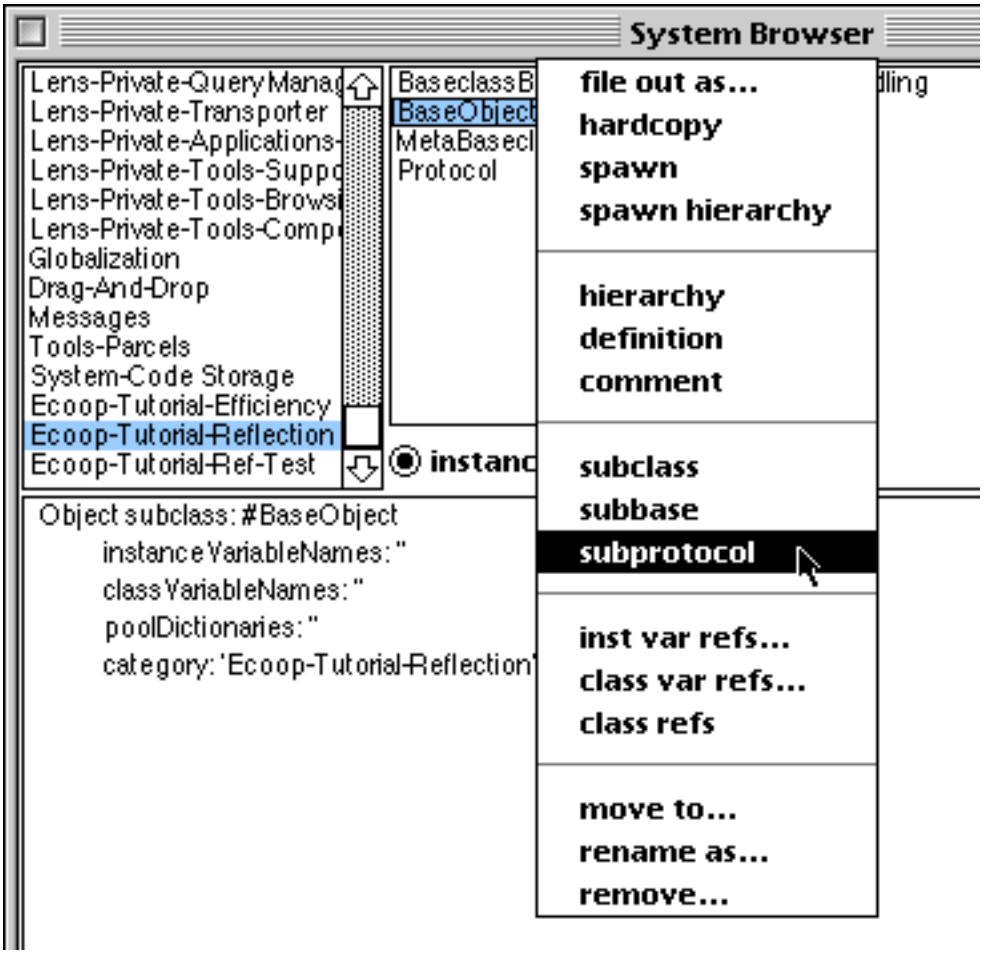

Fig. 2 Additional entries in class list menu

The class BaseclassBuilder is derived from ClassBuilder and implements additional plausibility checking if the class to be built or changed is derived from BaseObject. BaseclassBuilder checks whether:

- A class using mixin classes is derived directly or indirectly from BaseObject.
- The direct or indirect base class of a mixin class is Protocol.
- Protocol classes do not occur multiply in a class definition.
- All protocols in a class definition are indeed classes.
- Protocol classes do not have instance variables.

Classes can be freely moved from the Object hierarchy to the BaseObject hierarchy and back. If a class is moved off the BaseObject hierarchy its metaclass (and all metaclasses of its direct or indirect subclasses) must be changed to instances of Metaclass. Accordingly metaclasses have to become instances of MetaBaseclass if their classes are moved to the BaseObject hierarchy.

Therefore it was necessary to add this functionality to ClassBuilder, BaseclassBuilder, Metaclass, and MetaBaseclass. If a class is modified the class builder checks whether the metaclass has to be changed as well and performs it.

The lookup of a protocol method is expensive in terms of run-time. Therefore, once a protocol method is found, it is cached in the method dictionary of the receiver. The compiled method is copied, its receiver class is changed to that of the receiver and it is

added to the method dictionary of the receiver. Consequently, with a number of changes to the system these method caches must be flushed:

- The implementation of a protocol method is changed (either in a mixin class or in a primary base class).
- The inheritance hierarchy below BaseObject or Protocol is changed.
- The number or sequence of mixin classes is changed.

Flushing the method caches is done selectively. In most cases only methods that might be invalid are removed.

If a method is only added to the method dictionary, but not entered into the class organization it remains invisible in all development tools. This is exactly the intended behavior, since a protocol method should be excusively visible in the protocol class.

An IBM/VisualAge prototype of the mixin classes proved that this optimization can be ported with little effort.

4.3 Implementation

This chapter describes the four new classes BaseObject, Protocol, MetaBaseclass, and BaseclassBuilder and their most important methods.

4.3.1 BaseObject

The class BaseObject is the base class of all classes that use mixin classes. BaseObject itself is not considered as a primary base class and therefore its metaclass is an instance of Metaclass rather than MetaBaseclass.

```
Object subclass: #BaseObject
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Mixins'
```

BaseObject only overrides #doesNotUnderstand. First it tries to figure out whether the send was caused by a super send or not. This is done by inspecting the byte code of the calling method. In order to get the context of the caller, a full block is created and this blocks builds the starting point for searching the context chain backwards. Based on these findings #doesNotUnderstand searches for the requested method in the protocol classes starting with this class or its superclass. If the method is found a copy of it is installed in the method dictionary of the first protocol supporting class up the hierarchy and the message send is restarted. Otherwise Object>>#doesNotUnderstand is called.

```
doesNotUnderstand: aMessage
    "Method could not be found. Try to find method in protocol classes, copy it,
    install it in the first supporter's method dictionary and send the message again.
    Treat super send differently by starting search in superclass"

    | method block t1 t2 byte sendContext metaclass firstSupporter |
    block := [t1 := t2].
    sendContext := block outerContext sender.
    byte := sendContext method byteAt: sendContext pc - 2.
    metaclass := self class class.
    byte == 242 ifTrue: [metaclass := metaclass superclass].
    MethodDictionary keyNotFoundSignal
        handle: [:ex | method := nil]
        do: [method := metaclass compiledMethodAt: aMessage selector].
    method == nil
        ifTrue: [super doesNotUnderstand: aMessage]
        ifFalse: [method := method copy].
    firstSupporter := metaclass firstProtocolSupporter: method mclass.
    method mclass: firstSupporter.
    firstSupporter addSelector: aMessage selector withMethod: method.
    ^method valueWithReceiver: self arguments: aMessage arguments
```

4.3.2 MetaBaseclass

All metaclasses of classes derived from BaseObject are instances of MetaBaseclass. This class enhances the metaclass system of Smalltalk with mixin classes. MetaBaseclass comprises:

- Creation of classes derived from BaseObject.
- Management of meta information and providing access to it.
- Converting an instance of MetaBaseclass into one of Metaclass.
- Infrastructure support for browsing tools.

```
Metaclass subclass: #MetaBaseclass
    instanceVariableNames: 'protocol '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Mixins'
```

The instance variable protocol contains an OrderedCollection with the names of all supported mixin classes.

The method #canUnderstand: is called by Behavior>>#canUnderstand:. If a method is not found in the primary base hierarchy, the search is continued along the protocol hierarchies.

```
canUnderstand: selector
    "Answer true if one of the receivers protocol classes can respond to the message
    whose selector is the argument"

    protocol do: [:each | ((Smalltalk at:each) canUnderstand: selector) ifTrue: [^true]].
    ^false
```

If a method cannot be found, #compiledMethodAt: tries to find the method in the protocol classes and returns it. This method is called by BaseObject>>#doesNotUnderstand.

```
compiledMethodAt: aSelector
    "return compiled method; search depth first"

    | protocolClass |
    protocol do: [:each | (protocolClass := (Smalltalk at: each)
                whichClassIncludesSelector: aSelector) ~~ nil ifTrue: [^protocolClass compiledMethodAt:
                aSelector]].
    superclass ~~ BaseObject class ifTrue: [^superclass compiledMethodAt: aSelector].
    ^nil
```

The method #flushMethodCache removes all protocol methods from primary base class method dictionaries. A method can be flushed if it is in the method dictionary, but not in the class organization.

```
flushMethodCache
    "flush method caches of this class and its subclasses"

    ((thisClass allSubclasses) add: thisClass; yourself) do: [:each | each selectors do: [:eachSelector | (each
    organization includesElement: eachSelector)
        ifFalse: [each removeSelector: eachSelector]]]
```

The method #supportsProtocol returns true, if aProtocol is supported by a class. A protocol is also supported by mixing in a protocol derived from it.

```
supportsProtocol: aProtocol
    "Check whether receiver´s instance supports a protocol:
    - aProtocol is listed in protocol
    - aProtocol is a baseprotocol of one of the protocols in protocol"

    (protocol includes: aProtocol printString)
        ifTrue: [^true].
    protocol do: [:each | ((Smalltalk at: each asSymbol)
            inheritsFrom: (aProtocol))
```

```
            ifTrue: [^true]].
    superclass ~~BaseObject class ifTrue: [^superclass supportsProtocol: aProtocol].
```

Starting with thisClass the method #firstProtocolSupporter returns the first class up the hierarchy that supports aProtocol. This method is needed to find the appropriate class for installing a protocol method into the method dictionary of a base class.

```
firstProtocolSupporter: aProtocol
    "starting with thisClass return the first class up the hierarchy supporting
    aProtocol"

    | classProtocol |
    thisClass withAllSuperclasses do: [:eachClass | eachClass ~~ BaseObject
            ifTrue:
                [classProtocol := eachClass class protocol.
                (classProtocol includes: aProtocol printString)
                    ifTrue: [^eachClass].
                classProtocol do: [:each | ((Smalltalk at: each asSymbol)
                        inheritsFrom: aProtocol)
                        ifTrue: [^eachClass]]]
            ifFalse: [^nil]]
```

With the deletion of a mixin class it must be removed from all primary base classes supporting it. This is done by calling the method #removeProtocol:.

```
removeProtocol: aProtocol
    "remove a Protocol
    - flush relevant method caches
    - remove protocol from supporting classes"

    | aProtocolName |
    aProtocolName := aProtocol printString.
    (aProtocol inheritsFrom: Protocol)
        ifTrue: [BaseObject allSubclassesDo: [:each | (each class protocol includes: aProtocolName)
                ifTrue:
                    [each class flushMethodCache.
                    each class protocol remove: aProtocolName]]]
```

Four methods provide the infrastructure for converting instances of Metaclass to instances of MetaBaseclass and back.

- The method #converter returns a symbol (#asMetaclass or #asMetaBaseclass) defining the method that has to be called in order to get a copy of a metaclass. This copy is now an instance of the receiver of #converter.
- The methods #asMetaclass and #asMetaBaseclass return a copy of a metaclass whose class is the specified metaclass.
- All instance variables of a metaclass are copied to the new metaclass by using #copyFrom:.

```
converter
    "Return message that has to be executed if self has to be converted"

    ^#asMetaBaseclass

copyFrom: aMetaclass
    "copy all instance variables from aMetaclass into self"

    super copyFrom: aMetaclass.
    aMetaclass class == self class
        ifTrue: [protocol := aMetaclass protocol]
        ifFalse: [protocol := OrderedCollection new]

asMetaclass
    "return an copy of self but as an instance of Metaclass "

    | newMeta |
    newMeta := Metaclass new.
    newMeta copyFrom: self.
```

```
    ^newMeta

asMetaBaseclass
    "return self since it is already an instance of MetaBaseclass "

    ^self
```

4.3.3 Protocol

The class Protocol is just a convention. In order to identify a mixin class it must be derived from Protocol. Protocol and its subclasses must not have instances and instance variables.

```
Object subclass: #Protocol
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Mixins'
```

Protocol overrides #new and #new:. Both methods raise an error if one tries to create an instance of Protocol.

4.3.4 BaseclassBuilder

The class BaseclassBuilder is derived from ClassBuilder. Since ClassBuilder was not designed for reuse, some methods had to be copied to BaseclassBuilder and adapted there. Examples are #createNewSubclass or #modifyExistingClass.

An important part of functionality of BaseclassBuilder is the validation of a new or changed class prior to changing the system. For primary base classe these checks are performed in #validateProtocol and #validateBaseObject.

```
ClassBuilder subclass: #BaseclassBuilder
    instanceVariableNames: 'protocol '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Mixins'
```

The instance variable protocol holds an array of protocol strings.

The method changeMicroState was copied from ClassBuilder. Prior to recompiling all methods the class's protocol is updated. (Those parts of a method which were changed are emphasized by a dotted underline.)

```
changeMicroState
    "Change those aspects of the class that are relatively independent
    of any other changes to the class"

    self changeCategory.
    self changeClassVariables.
    self changePools.
    self sanityCheck: class.
    self currentClass class protocol: self protocol.
    recompile ifTrue:
        [class withAllSubclasses
            do: [:cls |
                Transcript show: cls name, ' recompiling...'.
                cls rebindAllMethods.
                Transcript show: 'done'; cr]]
```

The method #createNewSubclass first creates a new meta class (= instance of MetaBaseclass). Additionally to ClassBuilder>>#createNewSubclass it sets the protocol of the meta class according to the class definition.

```
createNewSubclass
    "The class does not exist--create a new class-metaclass pair"

    | newMeta |
    self runValidationChecksForNewClass.
```

```
newMeta := self metaclassClass new.
newMeta assignSuperclass: (self classOf: superclass).
newMeta methodDictionary: MethodDictionary new.
newMeta setInstanceFormat: (self classOf: superclass) format.
newMeta protocol: self protocol.
class := newMeta new.
class assignSuperclass: superclass.
class methodDictionary: MethodDictionary new.
class setInstanceFormat: self computeFormat.
self setStructureOf: class.
class setName: className.
self register: class inPlaceOf: nil.
self changeMicroState.
^self logNew: class
```

The method #modifyExistingClass mutates the metaclass if necessary. Additionally, it has to flush all method caches

- of primary base classes that support this protocol or a derived protocol if the class is a protocol class;
- of this class and all its derived classes if the class is derived from BaseObject;

```
modifyExistingClass
    "Enumerate the legal types of changes and perform them.  If a new
    class was created, the final check for which kind of mutation is
    skipped, since a new class needs no mutation"

    self runValidationChecks.
    self adaptMetaclass.
    self flushMethodCache: class.
    self needsMutation
        ifTrue: [self doMutationInPlace
            ifTrue: [self mutateCurrentClass]
            ifFalse: [self mutateToNewClass]].
    self changeMicroState.
    ^self logChanged: self currentClass
```

4.3.5 Infrastructure

The meta-level system of Smalltalk and its infrastructure (e. g. Browsers) are hardly designed for reuse. No wonder, they have never been reused so far apart from prototypes or experiments. Reusing a system not designed for reuse usually results in changing existing code (e. g. introducing case-like structures) or copying code. Nevertheless, one of the design goals was to make as few changes as possible to the system.

Some of the changes were realized in MetaBaseclass and BaseclassBuilder. Overriding was done by copying the appropriate code and inserting the changes. Despite of that, a few methods had to be changed in place (10 methods in Class, ClassDescription, Behavior, Browser, ClassBuilder and SystemDictionary) and some were added (10 methods in Class, Metaclass and Browser).

One alternative would have been to redesign and reimplement the meta-level system of Smalltalk. This certainly is a hairy and daunting task. Moreover, every new release of VisualWorks would have required a major effort to reintroduce these changes. It also would be much more difficult to port the mixin classes to other Smalltalk implementations like VisualAge, where the meta-level system is similar to VisualWorks.

4.3.5.1 Class

The class Class is responsible for class creation. Two new methods had to be added:

- subclass:protocol:instanceVariableNames:classVariableNames:poolDictionaries:category:
  Create an new class with mixing in protocol classes.
- subprotocol:category:
  Create a new subprotocol.

The method subclass:instanceVariableNames:classVariableNames:poolDictionaries:category: was changed in order to create the appropriate class if the base class is derived from BaseObject.

```
subclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat
    "This is the standard initialization message for creating a new class as a subclass
    of an existing class (the receiver)."

    | approved |
    approved := SystemUtils
                validateClassName: t
                confirm: [:msg :nm | Dialog confirm: msg]
                warn: [:msg | Dialog warn: msg].
    approved == nil ifTrue: [^nil].
    (self inheritsFrom: BaseObject)
        ifTrue: [^self
                subclass: t
                protocol: ''
                instanceVariableNames: f
                classVariableNames: d
                poolDictionaries: s
                category: cat].
    ^(self classBuilder) superclass: self; environment: self environment; className: approved; instVarString:
    f; classVarString: d; poolString: (self computeFullPoolString: s); category: cat; beFixed; reviseSystem
```

4.3.5.2 ClassDescription

The class ClassDescription is, among others, responsible for the class organisation and for the definition message of a class. The two new definition messages, as implemented in Class, were added.

```
definitionMessage
    "Answer a MessageSend that defines the receiver."

    | selector isProtocol isBaseObject args |
    isBaseObject := self inheritsFrom: BaseObject.
    isProtocol := self inheritsFrom: Protocol.
    isBaseObject ifTrue: [selector :=
        #subclass:protocol:instanceVariableNames:classVariableNames:poolDictionaries:category:].
    isProtocol ifTrue: [selector := #subprotocol:category:].
    isProtocol | isBaseObject ifFalse: [selector := self isVariable
        ifTrue: [self isBits
            ifTrue:
                [#variableByteSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category
            :]
            ifFalse: [#variableSub-
class:instanceVariableNames:classVariableNames:poolDictionaries:category:]]
        ifFalse: [#subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:]].
    isBaseObject ifTrue: [args := (Array new: 5)
                    at: 1 put: self name;
                    at: 2 put: self class protocolString;
                    at: 3 put: self instanceVariablesString;
                    at: 4 put: self classVariablesString;
                    at: 5 put: self sharedPoolsString; yourself].
    isProtocol ifTrue: [args := Array with: self name].
    isProtocol | isBaseObject ifFalse: [args := Array
                    with: self name
                    with: self instanceVariablesString
                    with: self classVariablesString
                    with: self sharedPoolsString].
    ^MessageSend
        receiver: superclass
        selector: selector
        arguments: (args copyWith: self category asString)
```

4.3.5.3 Behavior

The class Behavior provides the minimum state and behavior for objects that can create instances. A crucial method is #canUnderstand, which returns true if instances of this class can respond to a given selector. This behavior had to be changed for mixin classes, since a method could also be defined in a protocol.

```
canUnderstand: selector
    "Answer true if the receiver can respond to the message whose selector
    is the argument, false otherwise.  The selector can be in the method dictionary
    of the receiver's class, any of its superclasses, or in its protocol classes"

    (self includesSelector: selector) ifTrue: [^true].
    superclass notNil ifTrue: [(superclass canUnderstand: selector) ifTrue: [^true]].
    (self class isMemberOf: MetaBaseclass) ifTrue: [^self class canUnderstand: selector].
    ^false
```

4.3.5.4 ClassBuilder

Prior to modifying a class ClassBuilder has to check whether the class is moved off the BaseObject hierarchy or into it. The method #adaptMetaclass compares the metaclasses of the class to be changed and its potentially new superclass and triggers the mutation of the metaclass if they are not equal.

```
modifyExistingClass
    "Enumerate the legal types of changes and perform them.  If a new
    class was created, the final check for which kind of mutation is
    skipped, since a new class needs no mutation"

    self runValidationChecks.
    self adaptMetaclass.
    self needsMutation
        ifTrue: [self doMutationInPlace
            ifTrue:  [self mutateCurrentClass]
            ifFalse: [self mutateToNewClass]].
    self changeMicroState.
    ^self logChanged: self currentClass

adaptMetaclass
    "if class changes from the Object hierarchy to the BaseObject hierarchy or vice versa, mutate its
    metaclass"

    |  converter selfBase superBase |
    selfBase := self currentClass inheritsFrom: BaseObject.
    superBase := (self superclass inheritsFrom: BaseObject) or: [self superclass == BaseObject].
    converter := self superclass class converter.
    (selfBase xor: superBase) ifTrue:
        [self currentClass withAllSubclasses do:
            [:eachClass | eachClass class become: (eachClass class perform: converter)]]
```

4.3.5.5 Metaclass

Mutating a metaclass necessitates four additional methods in Metaclass. All four methods are overridden in MetaBaseclass.

- The method #converter returns a symbol (#asMetaclass or #asMetaBaseclass) defining the method that has to be called in order to get a copy of a metaclass. This copy is now an instance of the receiver of #converter.
- The methods #asMetaclass and #asMetaBaseclass return a copy of a metaclass whose class is the specified metaclass.
- All instance variables of a metaclass are copied to the new metaclass by using #copyFrom:.

The class BaseObject is an exception. Its metaclass is an instance of Metaclass, nevertheless it is treated like a protocol class.

```
converter
    "Return message that has to be executed if self has to be converted"

    thisClass == BaseObject
        ifTrue: [^#asMetaBaseclass]
        ifFalse: [^#asMetaclass]

copyFrom: aMetaclass
    "copy all instance variables from aMetaclass into self"

    superclass := aMetaclass superclass.
    methodDict := aMetaclass getMethodDictionary.
    format := aMetaclass format.
    subclasses := aMetaclass getSubclasses.
    instanceVariables := aMetaclass instVarNames.
    organization := aMetaclass organization.
    thisClass := aMetaclass soleInstance.

asMetaclass
    "return self since it is already an instance of MetaBaseclass "

    ^self

asMetaBaseclass
    "return an copy of self but as an instance of MetaBaseclass "

    | newMeta |
    newMeta := MetaBaseclass new.
    newMeta copyFrom: self.
    ^newMeta
```

4.3.5.6 SystemDictionary

The class SystemDictionary has only one instance, named Smalltalk and holds most information about the structure of the system. Removing a class from the system is processed by SystemDictionary. Removing a protocol class causes flushing all corresponding method caches and removing the protocol from the metaclasses.

```
removeClassNamed: className
    "Remove the class with the name className, and all of its subclasses,
    from the system, and note the removal in the system ChangeSet."

    | class |
    class := self at: className asSymbol ifAbsent: [^self].
    "remove subclasses first"
    class subclasses do: [:subclass | self removeClassNamed: subclass name].
    MetaBaseclass  removeProtocol: class.
    ChangeSet current removeClass: class.
    self organization removeElement: className.
    self at: className put: nil.
    Undeclared declare: className asSymbol from: self.
    self flushClassNameCache.
    SourceFileManager default logChange: class name,' removeFromSystem'.
    class obsolete
```

4.3.5.7 Browser

The class Browser implements the system browser. A lot of information is hard coded and hardly adaptable. Hence most changes to the system had to be done in this class. Newly added methods are:

- #flushMethodCaches:aSelector
  A method was changed. If current class is a protocol then remove selector from all

method dicts using this protocol. If class is a primary base then its method cache and those of its subclasses are flushed.

- #baseObjectDef
  Display base object definition template.
- #subclassDef
  Display subclass definition template.
- #protocolDef
  display subprotocol definition template

The method #classMenu was changed such that the user gets a menu with subclass creation templates (subclass, subbase, and subprotocol) to choose from.

```
classMenu
    "Answer a Menu of operations on classes that is to be displayed
    when the operate menu button is pressed."
    "Browser flushMenus"

    className == nil ifTrue: [^Menu labels: 'subclass\subbase\subprotocol' withCRs values: #(#subclassDef
                #baseObjectDef #protocolDef)].
    ClassMenu == nil ifTrue: [ClassMenu := Menu
                labels: 'file out as...\hardcopy\spawn\spawn hierarchy
    hierarchy\definition\comment
    subclass\subbase\subprotocol
    inst var refs...\class var refs...\class refs
    move to...\rename as...\remove...' withCRs
                lines: #(4 7 10 13)
                values: #(#fileOutClass #printOutClass #spawnClass #spawnHierarchy #showHierarchy
    #editClass #editComment #subclassDef #baseObjectDef #protocolDef #browseFieldReferences
    #browseClassVariables #browseClassReferences #changeClassCategory #renameClass #removeClass )].
    ^ClassMenu
```

In #acceptText:from: we now take into account that there are new browser states (in terms of class definition).

```
acceptText: aText from: aController
    "Text has been changed.  Store or compile the text, depending on
    the current mode of the receiver."

    textMode == #classDefinition ifTrue:
        [| accepted |
        accepted := self acceptClass: aText from: aController.
        aController textHasChanged: accepted not.
        ^accepted].
    textMode == #baseObjectDefinition ifTrue:
        [| accepted |
        accepted := self acceptClass: aText from: aController.
        aController textHasChanged: accepted not.
        ^accepted].
    textMode == #protocolDefinition ifTrue:
        [| accepted |
        accepted := self acceptClass: aText from: aController.
        aController textHasChanged: accepted not.
        ^accepted].
    textMode == #methodDefinition ifTrue:
        [| accepted |
        accepted := self acceptMethod: aText from: aController.
        aController textHasChanged: accepted not.
        ^accepted].
    textMode == #categories ifTrue:
        [Cursor wait showWhile:
            [organization changeFromString: aText string.
            self newCategoryList: category].
        ^true].
    textMode == #protocols ifTrue:
        [self classForSelectedProtocol organization changeFromString: aText string.
        self classForSelectedProtocol reorganize.
        self classForSelectedProtocol logOrganizationChange.
```

```
        self textMode: #protocol; newProtocolList: nil.
        ^ true].
    textMode == #comment ifTrue:
        [self nonMetaClass comment: aText string.
        self textMode: #comment; newProtocolList: nil.
        ^ true].
    ^ false
```

In #text we now have to call the appropriate subclass creation templates.

```
text
    | text |
    textMode == #classDefinition ifTrue:
        [className == nil
            ifTrue: [^ (Class template: category) asText]
            ifFalse: [^ self selectedClass definition asText]].
    textMode == #baseObjectDefinition ifTrue:
        [^ (MetaBaseclass template: category) asText].
    textMode == #protocolDefinition ifTrue:
        [^ ('NameOfSuperProtocol subprotocol: #NameOfProtocol
    category: ', category asString storeString) asText].
    textMode == #methodDefinition ifTrue:
        [selector == nil
            ifTrue: [^ self classForSelectedMethod sourceCodeTemplate asText]
            ifFalse: [^ (self classForSelectedMethod sourceCodeAt: selector) asText
                    makeSelectorBoldIn: self classForSelectedMethod]].
    textMode == #category ifTrue:
        [^ 'category to add' asText].
    textMode == #categories ifTrue:
        [^ organization printString asText].
    textMode == #protocol ifTrue:
        [^ 'protocol to add' asText].
    textMode == #protocols ifTrue:
        [^ self selectedClass organization printString asText].
    textMode == #comment ifTrue:
        [text := self selectedClass comment asText.
        text isEmpty ifFalse: [^ text].
        self selectedClass isMeta ifTrue: [^'Select the browser switch "instance" to see the comment'
asText].
        ^ self selectedClass commentTemplateString asText].
    textMode == #hierarchy ifTrue:
        [^ self selectedClass printHierarchy asText].
    ^ Text new
```

The method #removeClass warns the user if the class to be deleted is a protocol class and it is used by one or more BaseObject classes.

```
removeClass
    | class |
    self changeRequest ifFalse: [^self].
    Dictionary keyNotFoundSignal
        handle:
            [:ex |
            Dialog warn: ('Can''t remove the class. Class <1s> no longer exists.' expandMacrosWith:
                className)
                for: self interfaceWindow.
            ex return]
        do:
            ["KeyNotFoundSignal is raised when the class name to be removed
            in a browser is already removed in another browser."
            class := self nonMetaClass.
            (Dialog confirm: ('Are you certain that you<n>want to remove the class <1p>?'
                expandMacrosWith: class)
                for: self interfaceWindow)
                ifTrue:
                    [class subclasses size > 0 ifTrue: [(Dialog confirm: ('<1p> has subclasses. Remove it
                        anyway?' expandMacrosWith: class)
                            for: self interfaceWindow)
                            ifFalse: [^self]].
                    (MetaBaseclass classesSupportingProtocol: class) size > 0 ifTrue: [(Dialog confirm: ('There
                        are classes supporting Protocol <1p>. Remove it anyway?' expandMacrosWith: class)
```

```
                for: self interfaceWindow)
                ifFalse: [^self]].
        class removeFromSystem.
        self newClassList: nil]]
```

5 Conclusion

Extending Smalltalk with mixin classes can be accomplished with four new classes and a few changes to the existing system. This clearly shows the flexibility of Smalltalk's reflective architecture. Nevertheless, the implementation is very efficient and as soon as the method caches are up-to-date, there is no difference between calling primary base class methods and protocol methods. Moreover, this approach is also portable to at least IBM/VisualAge Smalltalk, which was proofed by a prototypical implementation.

Terry Montlick describes in [Mon96] the design and implementation of mixin classes in Smalltalk. However, his approach uses instance based mixins. For every mixin class an instance is created for each primary base instance. Each primary base instance holds a collection of mixin instances and each mixin instance holds a reference to its primary base instance.

This implementation has some advantages:

- No new metaclasses and changes on the meta classs level.
- Only Object>>#doesNotUnderstand had to be adapted.
- Mixin classes can have instance variables (which is usually not desired).
- Mixins are instance based and can be changed at run-time on an instance basis.

However, this approach also holds several serious disadvantages:

- Conceptionally mixin classes are not supposed to have instances.
- References to mixin instances have to be maintained manually.
- The implementation cannot be optimized (every call to a mixin method has to be redirected in #doesNotUnderstand).
- A primary base instance cannot be referenced using self in a mixin method.

Taking these disadvantages into account a widespread usage of this instance based implementation is quite unlikely.

6 Literature

[Arn96] Ken Arnold and James Gosling, *The Java Programming Language*, Addison-Wesley, Reading, Mass. (1996)

[Cot95] Sean Cotter and Mike Potel, *Inside Taligent Technology*, Addison-Wesley, Reading, Mass. (1994)

[Bob88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, David A. Moon, *Common Lisp Object System Specification*, X3J13 Document 88-002R, June 1988

[Bra90] Gilad Bracha and William Cook, "Mixin-based Inheritance", in *OOPSLA '90 Proceedings*, Ed.: N. Meyrowitz, Sigplan Notices 10(25), ACM, NewYork (1990)

[Bür95] Ute Bürkle, Guido Gryczan and Heinz Züllighoven, "Object-Oriented System Development in a Banking Project: Methodology, Experiences, and Conclusions", *Human Computer Interaction 10*, 2&3 (1995)

[Foo89] Brain Foote and Ralph E. Johnson, "Reflective Facilities in Smalltalk-80", in *OOPSLA '89 Proceedings*, Ed.: N. Meyrowitz, Sigplan Notices 10(24), ACM, NewYork (1989)

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, Reading, Mass. (1995)

[Gold89] Adele Goldberg and David Robson, *Smalltalk-80, The Language*, Addison Wesley, Reading, Mass. (1989)

[Met96] Metrowerks Inc., *Inside PowerPlant ™ for CW9*, St. Laurent, Canada (1996)

[Mey88] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, Hemel Hempstead, United Kingdom (1988)

[Mae87] Pattie Maes, “Concepts and Experiments in Computational Reflection”, in *OOPSLA '87 Proceedings*, Ed.: N. Meyrowitz, Sigplan Notices 12(22), ACM, NewYork (1987)

[Mon96] Terry Montlick, “Implementing mixins in Smalltalk”, *The Smalltalk Report*, 14-15, 5(9), SIGS Publications Inc., New York (1996)

[Par95] ParcPlace-Digitalk, Inc., *VisualWorks® User’s Guide*, Sunnyvale, CA (1994)

[Rie95] Dirk Riehle and Heinz Züllighoven, "A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor", *Pattern Languages of Program Design*, Ed.: James O. Coplien and Douglas C. Schmidt, Addison-Wesley, Reading, MA (1995)

[Str91] Bjarne Stroustrup, *The C++ Programming Language*, Second Edition, Addison-Wesley, Reading, Mass. (1991)

[Tal94] Taligent Inc., *Taligent´s Guide to Designing Programs*, Addison-Wesley, Reading, Mass. (1995)

7 Source Code

7.1 BaseObject

```
Object subclass: #BaseObject
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Mixins'

doesNotUnderstand: aMessage
    "Method could not be found. Try to find method in protocol classes, copy it,
    install it in the first supporter's method dictionary and send the message again.
    Treat super send differently by starting search in superclass"

    | method block t1 t2 byte sendContext metaclass firstSupporter |
    block := [t1 := t2].
    sendContext := block outerContext sender.
    byte := sendContext method byteAt: sendContext pc - 2.
    metaclass := self class class.
    byte == 242 ifTrue: [metaclass := metaclass superclass].
    MethodDictionary keyNotFoundSignal
        handle: [:ex | method := nil]
        do: [method := metaclass compiledMethodAt: aMessage selector].
    method == nil
        ifTrue: [super doesNotUnderstand: aMessage]
        ifFalse: [method := method copy].
    firstSupporter := metaclass firstProtocolSupporter: method mclass.
    method mclass: firstSupporter.
    firstSupporter addSelector: aMessage selector withMethod: method.
    ^method valueWithReceiver: self arguments: aMessage arguments
```

7.2 Protocol

```
Object subclass: #Protocol
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ecoop-Tutorial-Reflection'

new
    "Raise an error since Protocol must not have any instances"

    self error: 'Protocol must not have instances'

new: anInteger
    "Raise an error since Protocol must not have any instances"

    self error: 'Protocol must not have instances'
```

7.3 MetaBaseclass

```
Metaclass subclass: #MetaBaseclass
    instanceVariableNames: 'protocol '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ecoop-Tutorial-Reflection'

protocol
    ^protocol

protocol: aCollection
    protocol := aCollection asOrderedCollection
```

```
new
    "The receiver can only have one instance.  Create it or complain that
    one already exists."

    thisClass == nil
        ifTrue: [^thisClass := super new]
        ifFalse: [self error: 'A MetaBaseClass should only have one instance!!']

protocolString
    "Answer a string of my protocol names separated by spaces."

    | aStream names |
    aStream := WriteStream on: (String new: 100).
    names := self protocol.
    1 to: names size do: [:i | aStream nextPutAll: (names at: i); space].
    ^ aStream contents

canUnderstand: selector
    "Answer true if one of the receivers protocol classes can respond to the message
    whose selector is the argument"

    protocol do: [:each | ((Smalltalk at:each) canUnderstand: selector) ifTrue: [^true]].
    ^false

compiledMethodAt: aSelector
    "return compiled method; search depth first"

    | protocolClass |
    protocol do: [:each | (protocolClass := (Smalltalk at: each)
                    whichClassIncludesSelector: aSelector) ~~ nil ifTrue: [^protocolClass compiledMethodAt:
                        aSelector]].
    superclass ~~ BaseObject class ifTrue: [^superclass compiledMethodAt: aSelector].
    ^nil

flushMethodCache
    "flush method caches of this class and its subclasses"

    ((thisClass allSubclasses) add: thisClass; yourself) do: [:each | each selectors do: [:eachSelector | (each
        organization includesElement: eachSelector)
                ifFalse: [each removeSelector: eachSelector]]]

supportsProtocol: aProtocol
    "Check whether receiver's instance supports a protocol:
    - aProtocol is listed in protocol
    - aProtocol is a baseprotocol of one of the protocols in protocol"

    (protocol includes: aProtocol printString)
        ifTrue: [^true].
    protocol do: [:each | ((Smalltalk at: each asSymbol)
            inheritsFrom: (aProtocol))
            ifTrue: [^true]].
    superclass ~~BaseObject class ifTrue: [^superclass supportsProtocol: aProtocol].
    ^false

firstProtocolSupporter: aProtocol
    "starting with thisClass return the first class up the hierarchy supporting
    aProtocol"

    | classProtocol |
    thisClass withAllSuperclasses do: [:eachClass | eachClass ~~ BaseObject
            ifTrue:
                [classProtocol := eachClass class protocol.
                (classProtocol includes: aProtocol printString)
                    ifTrue: [^eachClass].
                classProtocol do: [:each | ((Smalltalk at: each asSymbol)
                        inheritsFrom: aProtocol)
                        ifTrue: [^eachClass]]]
            ifFalse: [^nil]]
```

```
template: category
    "Answer an expression that can be edited and evaluated in order to define
     a new class with mixin protocols."

    ^'NameOfSuperclass subclass: #NameOfClass
    protocol: ''protocol1 protocol2''
    instanceVariableNames: ''instVarName1 instVarName2''
    classVariableNames: ''ClassVarName1 ClassVarName2''
    poolDictionaries: ''''
    category: ', category asString storeString

classesSupportingProtocol: aProtocol
    "return a collection of all classes supporting aProtocol"

    | protocolsToTest |
    protocolsToTest := (aProtocol allSubclasses) add: aProtocol; yourself.
    ^BaseObject allSubclasses select: [:eachClass | (protocolsToTest detect: [:eachProtocol | eachClass class s
        upportsProtocol: eachProtocol]
            ifNone: [nil]) notNil]

removeProtocol: aProtocol
    "remove a Protocol
    - flush relevant method caches
    - remove protocol from supporting classes"

    | aProtocolName |
    aProtocolName := aProtocol printString.
    (aProtocol inheritsFrom: Protocol)
        ifTrue: [BaseObject allSubclassesDo: [:each | (each class protocol includes: aProtocolName)
                    ifTrue:
                        [each class flushMethodCache.
                        each class protocol remove: aProtocolName]]]

converter
    "Return message that has to be executed if self has to be converted"

    ^#asMetaBaseclass

copyFrom: aMetaclass
    "copy all instance variables from aMetaclass into self"

    super copyFrom: aMetaclass.
    aMetaclass class == self class
        ifTrue: [protocol := aMetaclass protocol]
        ifFalse: [protocol := OrderedCollection new]

asMetaclass
    "return an copy of self but as an instance of Metaclass "

    | newMeta |
    newMeta := Metaclass new.
    newMeta copyFrom: self.
    ^newMeta

asMetaBaseclass
    "return self since it is already an instance of MetaBaseclass "

    ^self
```

7.4 BaseclassBuilder

```
ClassBuilder subclass: #BaseclassBuilder
    instanceVariableNames: 'protocol '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ecoop-Tutorial-Reflection'
```

```
protocol
    ^protocol

protocol: anArray
    protocol := anArray

protocolString: aString
    self protocol: (self scan: aString)

createNewSubclass
    "The class does not exist--create a new class-metaclass pair"

    | newMeta |
    self runValidationChecksForNewClass.
    newMeta := self metaclassClass new.
    newMeta assignSuperclass: (self classOf: superclass).
    newMeta methodDictionary: MethodDictionary new.
    newMeta setInstanceFormat: (self classOf: superclass) format.
    newMeta protocol: self protocol.
    class := newMeta new.
    class assignSuperclass: superclass.
    class methodDictionary: MethodDictionary new.
    class setInstanceFormat: self computeFormat.
    self setStructureOf: class.
    class setName: className.
    self register: class inPlaceOf: nil.
    self changeMicroState.
    ^self logNew: class

modifyExistingClass
    "Enumerate the legal types of changes and perform them.  If a new
    class was created, the final check for which kind of mutation is
    skipped, since a new class needs no mutation"

    self runValidationChecks.
    self adaptMetaclass.
    self flushMethodCache: class.
    self needsMutation
        ifTrue: [self doMutationInPlace
            ifTrue: [self mutateCurrentClass]
            ifFalse: [self mutateToNewClass]].
    self changeMicroState.
    ^self logChanged: self currentClass

changeMicroState
    "Change those aspects of the class that are relatively independent
    of any other changes to the class"

    self changeCategory.
    self changeClassVariables.
    self changePools.
    self sanityCheck: class.
    self currentClass class protocol: self protocol.
    recompile ifTrue:
        [class withAllSubclasses
            do: [:cls |
                Transcript show: cls name, ' recompiling...'.
                cls rebindAllMethods.
                Transcript show: 'done'; cr]]

mutateToNewClass
    "Modify class to be a subclass of superclass"

    self flushMethodCaches: class.
    ^super mutateToNewClass

metaclassClass
    "The class used to create new metaclasses when they are needed"

    ^MetaBaseclass
```

```
runValidationChecks
    "'Prove' that the protocols are existing and class is derived from BaseObject"

    super runValidationChecks.
    self validateBaseObject.
    self validateProtocol

runValidationChecksForNewClass
    "'Prove' that the protocols are existing and class is derived from BaseObject"

    super runValidationChecksForNewClass.
    self validateBaseObject.
    self validateProtocol

validateBaseObject
    "'Prove' that class is derived from BaseObject"

    (self superclass == BaseObject or: [self superclass inheritsFrom: BaseObject])
        ifFalse: [self failureSignal raiseErrorString: self className , ' must be derived from BaseObject.']

validateProtocol
    "Check:
    - duplicate protocols
    - protocol is a class
    - protocol is derived from Protocol
    - protocol has no instance variables"
    | classes |
    classes := Smalltalk classNames.
    protocol
        do:
            [:each |
            (protocol occurrencesOf: each) > 1 ifTrue: [self failureSignal raiseErrorString: 'Protocol ' , each , ' occurs
                more than once.'].
            (classes includes: each asSymbol)
                ifFalse: [self failureSignal raiseErrorString: 'Protocol ' , each , ' not available.']
                ifTrue: [((Smalltalk at: each asSymbol)
                        inheritsFrom: Protocol)
                        ifFalse: [self failureSignal raiseErrorString: 'Class ' , each , ' is not derived from Protocol.']].
            (Smalltalk at: each asSymbol) instVarNames isEmpty ifFalse: [self failureSignal raiseErrorString: 'Protocol
' ,             each , ' has instance variables,\which is not valid for protocol classes.' withCRs]]

flushMethodCache: aClass
    "Flush method caches. aClass is BaseClass -> flush class and all classes
    down the hierarchy.
    aClass is Protocol -> flush all classes mixing in this protocol"

    (aClass inheritsFrom: BaseObject)
        ifTrue: [aClass class flushMethodCache].
    (aClass inheritsFrom: Protocol)
        ifTrue: [BaseObject allSubclasses do: [:each | (each class supportsProtocol: aClass)
                    ifTrue: [each selectors do: [:eachSelector | (each organization includesElement: eachSelector)
                                ifFalse: [each removeSelector: eachSelector]]]]]
```

7.5 Class

```
subclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat
    "This is the standard initialization message for creating a new class as a
    subclass
    of an existing class (the receiver)."

    | approved |
    approved := SystemUtils
                validateClassName: t
                confirm: [:msg :nm | Dialog confirm: msg]
                warn: [:msg | Dialog warn: msg].
    approved == nil ifTrue: [^nil].
    (self inheritsFrom: BaseObject)
        ifTrue: [^self
                subclass: t
                protocol: ''
```

```
                instanceVariableNames: f
                classVariableNames: d
                poolDictionaries: s
                category: cat].
    ^(self classBuilder) superclass: self; environment: self environment; className: approved; instVarString: f;
        classVarString: d; poolString: (self computeFullPoolString: s); category: cat; beFixed; reviseSystem

subclass: t protocol: p instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat
    "This is the standard initialization message for creating a new class as a subclass
    of an existing class (the receiver)."

    | approved |
    approved := SystemUtils
            validateClassName: t
            confirm: [:msg :nm | Dialog confirm: msg]
            warn: [:msg | Dialog warn: msg].
    approved == nil ifTrue: [^nil].
    ^BaseclassBuilder new
        superclass: self;
        environment: self environment;
        className: approved;
        protocolString: p;
        instVarString: f;
        classVarString: d;
        poolString: (self computeFullPoolString: s);
        category: cat;
        beFixed;
        reviseSystem

subprotocol: t category: cat
    "This is the standard initialization message for creating a new class as a subclass
    of an existing class (the receiver)."

    | approved |
    approved := SystemUtils
            validateClassName: t
            confirm: [:msg :nm | Dialog confirm: msg]
            warn: [:msg | Dialog warn: msg].
    approved == nil ifTrue: [^nil].
    (self == Protocol or: [self inheritsFrom: Protocol]) ifFalse: [^nil].
    ^ClassBuilder new
        superclass: self;
        environment: self environment;
        className: approved;
        instVarString: '';
        classVarString: '';
        poolString: (self computeFullPoolString: '');
        category: cat;
        beFixed;
        reviseSystem
```

7.6 ClassDescription

```
definitionMessage
    "Answer a MessageSend that defines the receiver."

    | selector isProtocol isBaseObject args |
    isBaseObject := self inheritsFrom: BaseObject.
    isProtocol := self inheritsFrom: Protocol.
    isBaseObject ifTrue: [selector :=
        #subclass:protocol:instanceVariableNames:classVariableNames:poolDictionaries:category:].
    isProtocol ifTrue: [selector := #subprotocol:category:].
    isProtocol | isBaseObject ifFalse: [selector := self isVariable
                    ifTrue: [self isBits
                            ifTrue:
            [#variableByteSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:]
                            ifFalse:
            [#variableSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:]]
                    ifFalse: [#subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:]].
    isBaseObject ifTrue: [args := (Array new: 5)
```

```
                    at: 1 put: self name;
                    at: 2 put: self class protocolString;
                    at: 3 put: self instanceVariablesString;
                    at: 4 put: self classVariablesString;
                    at: 5 put: self sharedPoolsString; yourself].
    isProtocol ifTrue: [args := Array with: self name].
    isProtocol | isBaseObject ifFalse: [args := Array
                    with: self name
                    with: self instanceVariablesString
                    with: self classVariablesString
                    with: self sharedPoolsString].
    ^MessageSend
        receiver: superclass
        selector: selector
        arguments: (args copyWith: self category asString)
```

7.7 Behavior

canUnderstand: selector

```
    "Answer true if the receiver can respond to the message whose selector
    is the argument, false otherwise.  The selector can be in the method dictionary
    of the receiver's class, any of its superclasses, or in its protocol classes"

    (self includesSelector: selector) ifTrue: [^true].
    superclass notNil ifTrue: [(superclass canUnderstand: selector) ifTrue: [^true]].
    (self class isMemberOf: MetaBaseclass) ifTrue: [^self class canUnderstand: selector].
    ^false
```

7.8 Metaclass

converter

```
    "Return message that has to be executed if self has to be converted"

    thisClass == BaseObject
        ifTrue: [^#asMetaBaseclass]
        ifFalse: [^#asMetaclass]
```

copyFrom: aMetaclass

```
    "copy all instance variables from aMetaclass into self"

    superclass := aMetaclass superclass.
    methodDict := aMetaclass getMethodDictionary.
    format := aMetaclass format.
    subclasses := aMetaclass getSubclasses.
    instanceVariables := aMetaclass instVarNames.
    organization := aMetaclass organization.
    thisClass := aMetaclass soleInstance.
```

asMetaclass

```
    "return self since it is already an instance of MetaBaseclass "

    ^self
```

asMetaBaseclass

```
    "return an copy of self but as an instance of MetaBaseclass "

    | newMeta |
    newMeta := MetaBaseclass new.
    newMeta copyFrom: self.
    ^newMeta
```

7.9 ClassBuilder

modifyExistingClass

```
"Enumerate the legal types of changes and perform them.  If a new
class was created, the final check for which kind of mutation is
skipped, since a new class needs no mutation"

self runValidationChecks.
self adaptMetaclass.
self needsMutation
    ifTrue: [self doMutationInPlace
        ifTrue: [self mutateCurrentClass]
        ifFalse: [self mutateToNewClass]].
self changeMicroState.
^self logChanged: self currentClass
```

adaptMetaclass

```
"if class changes from the Object hierarchy to the BaseObject hierarchy or vice versa, mutate       its metaclass"

|  converter selfBase superBase |
selfBase := self currentClass inheritsFrom: BaseObject.
superBase := (self superclass inheritsFrom: BaseObject) or: [self superclass == BaseObject].
converter := self superclass class converter.
(selfBase xor: superBase) ifTrue:
    [self currentClass withAllSubclasses do:
        [:eachClass | eachClass class become: (eachClass class perform: converter)]]
```

7.10 SystemDictionary

removeClassNamed: className

```
"Remove the class with the name className, and all of its subclasses,
from the system, and note the removal in the system ChangeSet."

| class |
class := self at: className asSymbol ifAbsent: [^self].
"remove subclasses first"
class subclasses do: [:subclass | self removeClassNamed: subclass name].
MetaBaseclass  removeProtocol: class.
ChangeSet current removeClass: class.
self organization removeElement: className.
self at: className put: nil.
Undeclared declare: className asSymbol from: self.
self flushClassNameCache.
SourceFileManager default logChange: class name,' removeFromSystem'.
class obsolete
```

7.11 Browser

acceptClass: aText from: aController

```
| oldClass class name |
oldClass := className == nil
            ifTrue: [Object]
            ifFalse: [self selectedClass].
class := Object errorSignal
            handle:
                [:ex |
                ex willProceed
                    ifTrue: [(Dialog confirm: ('<1s><n>Do you want to continue?' expandMacrosWith: ex
                        errorString)
                            for: self interfaceWindow)
                            ifTrue: [ex proceed]]
                    ifFalse: [Dialog warn: ex errorString for: self interfaceWindow].
                ex returnWith: nil]
            do:
                [| theClass |
                Cursor execute
                    showWhile:
```

```
                    [theClass := oldClass subclassDefinerClass new
                            evaluate: aText string
                            in: nil
                            receiver: nil
                            notifying: aController
                            ifFail: [^false]].
                SourceFileManager default logChange: aText string.
                theClass].
        class isBehavior
            ifTrue:
                [class isMeta
                    ifTrue: [name := class soleInstance name]
                    ifFalse: [name := class name].
                self newClassList: name.
                ^true]
            ifFalse: [^false]

acceptMethod: aText from: aController
    | newSelector |
    newSelector := self classForSelectedProtocol
                compile: aText
                classified: protocol
                notifying: aController.
    newSelector == nil ifTrue: [^false].
    self flushMethodCaches: newSelector.
    newSelector == selector
        ifFalse: [self newSelectorList: newSelector].
    ^true

acceptText: aText from: aController
    "Text has been changed.  Store or compile the text, depending on
    the current mode of the receiver."

    textMode == #classDefinition ifTrue:
        [| accepted |
        accepted := self acceptClass: aText from: aController.
        aController textHasChanged: accepted not.
        ^accepted].
    textMode == #baseObjectDefinition ifTrue:
        [| accepted |
        accepted := self acceptClass: aText from: aController.
        aController textHasChanged: accepted not.
        ^accepted].
    textMode == #protocolDefinition ifTrue:
        [| accepted |
        accepted := self acceptClass: aText from: aController.
        aController textHasChanged: accepted not.
        ^accepted].
    textMode == #methodDefinition ifTrue:
        [| accepted |
        accepted := self acceptMethod: aText from: aController.
        aController textHasChanged: accepted not.
        ^accepted].
    textMode == #categories ifTrue:
        [Cursor wait showWhile:
            [organization changeFromString: aText string.
            self newCategoryList: category].
        ^true].
    textMode == #protocols ifTrue:
        [self classForSelectedProtocol organization changeFromString: aText string.
        self classForSelectedProtocol reorganize.
        self classForSelectedProtocol logOrganizationChange.
        self textMode: #protocol; newProtocolList: nil.
        ^ true].
    textMode == #comment ifTrue:
        [self nonMetaClass comment: aText string.
        self textMode: #comment; newProtocolList: nil.
        ^ true].
    ^ false
```

```
baseObjectDef
    "display base object definition template"

    self className: nil.
    self changeRequest ifFalse: [^self].
    self textMode: #baseObjectDefinition.
    self newProtocolList: nil

classMenu
    "Answer a Menu of operations on classes that is to be displayed
    when the operate menu button is pressed."
    "Browser flushMenus"

    className == nil ifTrue: [^Menu labels: 'subclass\subbase\subprotocol' withCRs values: #(#subclassDef
    #baseObjectDef #protocolDef)].
    ClassMenu == nil ifTrue: [ClassMenu := Menu
                labels: 'file out as...\hardcopy\spawn\spawn hierarchy
    hierarchy\definition\comment
    subclass\subbase\subprotocol
    inst var refs...\class var refs...\class refs
    move to...\rename as...\remove...' withCRs
                lines: #(4 7 10 13)
                values: #(#fileOutClass #printOutClass #spawnClass #spawnHierarchy #showHierarchy #edit-
Class    #editComment #subclassDef #baseObjectDef #protocolDef #browseFieldReferences #browseClassVariables
    #browseClassReferences #changeClassCategory #renameClass #removeClass )].
    ^ClassMenu

flushMethodCaches: aSelector
    "- protocol class ->flush method caches of classes using this protocol
    - primary base ->flush method cache down the hierarchy"

    | thisClass |
    thisClass := Smalltalk at: className ifAbsent: [^self].
    (thisClass inheritsFrom: Protocol)
        ifTrue: [BaseObject allSubclasses do: [:each | (each organization includesElement: aSelector)
                ifFalse: [each removeSelector: aSelector]]].
    (thisClass inheritsFrom: BaseObject)
        ifTrue: [thisClass class flushMethodCache]

protocolDef
    "display protocol definition template"

    self className: nil.
    self changeRequest ifFalse: [^self].
    self textMode: #protocolDefinition.
    self newProtocolList: nil

removeClass
    | class |
    self changeRequest ifFalse: [^self].
    Dictionary keyNotFoundSignal
        handle:
            [:ex |
            Dialog warn: ('Can''t remove the class. Class <1s> no longer exists.' expandMacrosWith: className)
                for: self interfaceWindow.
            ex return]
        do:
            ["KeyNotFoundSignal is raised when the class name to be removed
            in a browser is already removed in another browser."
            class := self nonMetaClass.
            (Dialog confirm: ('Are you certain that you<n>want to remove the class <1p>?' expandMacrosWith:
class)
                for: self interfaceWindow)
                ifTrue:
                    [class subclasses size > 0 ifTrue: [(Dialog confirm: ('<1p> has subclasses. Remove it anyway?'
                        expandMacrosWith: class)
                            for: self interfaceWindow)
                            ifFalse: [^self]].
                    (MetaBaseclass classesSupportingProtocol: class) size > 0 ifTrue: [(Dialog confirm: ('There are
                        classes supporting Protocol <1p>. Remove it anyway?' expandMacrosWith: class)
                            for: self interfaceWindow)
```

```
                    ifFalse: [^self]].
                class removeFromSystem.
                self newClassList: nil]]

subclassDef
    "display subclass definition template"

    self className: nil.
    self changeRequest ifFalse: [^self].
    self textMode: #classDefinition.
    self newProtocolList: nil

text
    | text |
    textMode == #classDefinition ifTrue:
        [className == nil
            ifTrue: [^ (Class template: category) asText]
            ifFalse: [^ self selectedClass definition asText]].
    textMode == #baseObjectDefinition ifTrue:
        [^ (MetaBaseclass template: category) asText].
    textMode == #protocolDefinition ifTrue:
        [^ ('NameOfSuperProtocol subprotocol: #NameOfProtocol
    category: ', category asString storeString) asText].
    textMode == #methodDefinition ifTrue:
        [selector == nil
            ifTrue: [^ self classForSelectedMethod sourceCodeTemplate asText]
            ifFalse: [^ (self classForSelectedMethod sourceCodeAt: selector) asText
                    makeSelectorBoldIn: self classForSelectedMethod]].
    textMode == #category ifTrue:
        [^ 'category to add' asText].
    textMode == #categories ifTrue:
        [^ organization printString asText].
    textMode == #protocol ifTrue:
        [^ 'protocol to add' asText].
    textMode == #protocols ifTrue:
        [^ self selectedClass organization printString asText].
    textMode == #comment ifTrue:
        [text := self selectedClass comment asText.
        text isEmpty ifFalse: [^ text].
        self selectedClass isMeta ifTrue: [^'Select the browser switch "instance" to see the comment' asText].
        ^ self selectedClass commentTemplateString asText].
    textMode == #hierarchy ifTrue:
        [^ self selectedClass printHierarchy asText].
    ^ Text new
```

Ubilab Technical Reports

94.6.1 Maffeis S, Bischofberger WR, Mätzel K-U: GTS: *A Generic Multicast Transport Service*

94.9.1 Bischofberger WR, Kofler T, Mätzel K-U, Schäffer B: *Computer Supported Cooperative Software Engineering with Beyond-Sniff*

94.9.2 Bäumer D, Bischofberger WR, Lichter H, Schneider-Hufschmitdt M, Sedlmeier-Scholz V, Züllighoven H: *Prototyping von Benutzungsoberflächen*

94.10.1 Steiger P, Ansel Suter B: *Minnelli Schlussbericht*

94.10.2 Levy N, Hornstein T: *Text-to-Speech Technology: A Survey of German Speech Synthesis Systems*

95.6.1 Riehle D: *Muster am Beispiel der Werkzeug und Material Metapher*

95.7.1 Riehle D, Schäffer B, Schnyder M: *Design and Implementation of a Smalltalk Framework based on the Tools and Materials Metaphor*

97.1.1 Riehle D: *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose*

97.3.1 Brudermann R: *GeoTransporter—Entwurf und Implementierung eines Objekttransportes für das Geo-System*

97.6.1 Mätzel K L, Schnorf P: *Dynamic Component Adaptation*

97.7.1 Barja M L: *A Comparative Evaluation of OODBMSs*

98.5.1 Marsura P, Riehle D: *Design and Implementation of the Java Any Framework*

98.10.1 Bäumer D, Riehle D, Siberski W, Lilienthal C, Megert D, Sylla K H, Züllighoven H: *Values in Object Systems*

Paper copies of Ubilab technical reports can be ordered from the mailing address on the first page or by e-mail from its author using the scheme firstname.lastname@ubs.com. Most reports can also be obtained as PostScript files via WWW (http://www.ubs.com/ubilab).

Abstract

Multiple inheritance has become fashionable with the widespread use of C++. However, its unreflected use introduces more complexity than it tries to solve. In most cases mixin classes are a cleaner solution and better suited to design an object-oriented system. Java, for example, does not offer multiple inheritance but provides interfaces which are mixin classes. Smalltalk did provide multiple inheritance but it was never used in the class library and therefore removed after some time. Smalltalk is a reflective environment and allows the implementation of mixin classes at almost no cost at runtime. We present the design and implementation of Smalltalk mixin classes in VisualWorks\Smalltalk and compare it to an alternative.