Method Mixins

Erik Ernst

University of Aarhus, Denmark
eernst@daimi.au.dk

Abstract: The world of programming has been conquered by the procedure call mechanism, including object-oriented method invocation which is a procedure call in context of an object. This paper presents an alternative, *method mixin* invocations, that is optimized for flexible creation of composite behavior, where traditional invocation is optimized for as-is reuse of existing behavior. Tight coupling reduces flexibility, and traditional invocation tightly couples transfer of information and transfer of control. Method mixins decouple these two kinds of transfer, thereby opening the doors for new kinds of abstraction and reuse. Method mixins use shared name spaces to transfer information between caller and callee, as opposed to traditional invocation which uses parameters and returned results. This relieves the caller from dependencies on the callee, and it allows direct transfer of information further down the call stack, e.g., to a callee's callee. The mechanism has been implemented in the programming language gbeta. Variants of the mechanism could be added to almost any imperative programming language.

1 Introduction

Mixins are well-known as a device that allows for decoupling of subclasses from their superclasses [BC90, FKF98, BPS99]. This paper presents a related concept, *method mixins*, allowing for a novel and in some ways more profound decoupling of behavior specifications than traditional procedure or method calls.

The basic idea is that a method need not be a monolithic entity, it may be constructed from smaller parts, method mixins, which may be reused to construct many different methods. A procedure call or method invocation may also be used to create a more complex behavior from existing, simpler behaviors, but method mixins represent a very different set of tradeoffs.

There are several reasons for having method mixins in a language. With a traditional invocation, the caller depends on multiple characteristics of the callee, including the number and order of arguments. With a method mixin, the caller is completely independent of the callee. With a traditional invocation, the caller can provide extra information to a callee by adding an extra argument to the call; this means that *all* callees (e.g., a given method in many classes) must be modified to accept that extra argument. With method mixins, the caller may provide extra information freely; if there are a hundred callees and only one of them need the extra information, then the other 99 callees need not be changed. Traditional methods communicate with each other using arguments and shared state in objects or globally. Mixin methods support a novel scope for shared state which improves encapsulation and makes programs more robust in context of recursion or multiple threads. Finally, a novel and useful feature of method mixins is that they are able to create customized methods at run-time.

The contributions of this paper are: The basic concept of method mixins and their properties, a detailed comparison with traditional invocation, and a presentation of mixin methods as they have been implemented in gbeta [Ern99a, Ern01].

The rest of this paper is organized as follows: Section 2 introduces method mixins and their informal semantics, and Sec. 3 gives some examples of how they can be used. Section 4 compares method mixins with traditional method invocation. Related work is discussed in Sec. 5. Finally, Sec. 6 describes the implementation status, and Sec. 7 concludes.

2 Method Mixins

Method mixins are building blocks for methods, enabling a similar kind of incremental specification for behavior as that which is well-known from the world of classes. Similarly to class mixins, method mixins are more flexible than traditional (single or multiple) inheritance mechanisms. To explain this, we need to discuss the relation between invocations and objects, and to take a short detour into the language BETA [MMPN93] which supports an important predecessor of method mixins, as well as the language gbeta in which we have implemented full-blown method mixins. We end this section with a language independent description of the informal semantics of method mixins.

A procedure call or method invocation is associated with an activation record on a stack, which holds state required by the invocation such as actual arguments and local variables. In some languages, such as Smalltalk [GR89] and BETA, activation records are full-fledged objects, and activation records are similar to objects also in other languages. Activation records and objects differ in that an activation record has a behavior (the code or body of the method) whereas an object in many languages does not *per se* have a behavior—it has methods, and each method has a behavior. We can bridge this gap by considering objects that have a "default" method. A message send is then (1) dispatch, i.e., selecting the method, (2) creation of an object that plays the role as an activation record, (3) initialization of the activation object (method arguments are initialized according to the actual argument list, a `this` pointer is initialized to refer to the receiver, etc.), and (4) execution of the activation object. Finally, if the method returns a result it is delivered to the calling context.

Activation records are traditionally specified monolithically, as an implicit aspect of the treatment of the given method declaration by the compiler and run-time system. Commonalities between methods remain implicit and cannot be exploited for reuse purposes. When working with objects we are used to abstraction and reuse mechanisms such as inheritance, but these mechanisms are absent for methods.

BETA has supported activation records as objects with a default method [KMMPN87] since the late 1970'ies. This was a consequence of unifying the concepts 'class' and 'method' into the more general concept *pattern*. Henceforth, the terms class and method are used to mean pattern in connection with BETA—a class is just a pattern that is used in a class-like manner (instances are bound to named references and their state is significant), and a method is a pattern that is used in a method-like manner (instances are anonymous and the side-effects or return value of their default method is significant). Since inheritance is needed with classes, and classes are patterns, BETA supports inheritance for patterns. Hence, it also supports inheritance for methods. This makes it possible to reuse and extend behaviors in a way that is similar to reuse of classes by inheritance. Arguments and local variables are inherited in a similar way as instance variables are inherited by a subclass, and behavior is refined by means of the `INNER` mechanism, as described below. For an example, see Box 1.

BETA style method inheritance

```
void operation(Account a) {
  int tmp; a.lock(); INNER; a.unlock();
}
void withdraw(int amount) extends operation {
  tmp = a.getBalance()-amount; a.setBalance(tmp);
}
```

Box 1

We use a Java-like syntax, which is quite different from BETA [pwimaBu] and superficially distinguishes between classes and methods; however, we have made this choice because BETA syntax is not widely known. The only new concept here is that a method `extends` another method, which gives rise to inheritance hierarchies of methods. For a given method in such a hierarchy, there is a *chain* from that method to its immediate supermethod etc. up to the top of the hierarchy (here: `withdraw` inherits from `operation`). The local variable `tmp` illustrates inheritance of instance variables—`withdraw` inherits `tmp` from `operation`. For behavior, we have to consider the inheritance structure explicitly. Execution starts at the *least* specific level (here: `operation`), and `INNER` invokes the next more specific level (here: `withdraw`). If there is no such more specific level, then `INNER` has no effect. Thus, the method `operation` contains the basic behavior of a whole category of methods, namely the ones that need to lock the account `a`, do *something*, and unlock it again; `withdraw` is but one of the possible methods using this behavioral scheme. For illustration, here is the same method expressed without method inheritance:

Equivalent method

```
void withdraw(Account a, int amount) {
  int tmp;
  a.lock();
  tmp = a.getBalance()-amount; a.setBalance(tmp);
  a.unlock();
}
```

Box 2

Note that method inheritance creates a refined method based on given methods; when (in standard OO) a class inherits a method from a superclass no refinement takes place, so this is an entirely different concept.

Even though BETA style method inheritance and the INNER mechanism support a kind of invocation, it is not sufficiently flexible to justify considering INNER in BETA as an alternative to the traditional method invocation mechanism. The problem is that each submethod in BETA is called by INNER in one and only one method, namely the immediate supermethod (e.g., in Box 1, the direct call site for withdraw is INNER in the body of operation, and it could not be anywhere else). To make it clear how restricting this is, consider a rudimentary dialect of BASIC that supports GOTO, but does not support CALL or RETURN. We can write a subroutine and call it (using two GOTO statements for CALL and RETURN), but we can only call this subroutine from one particular call site because the second GOTO will "return" to the same location every time. Similarly, a submethod in BETA can only be called from the body of one particular method, namely the immediate supermethod—the *caller* is hardwired into the callee at compile-time. Note, however, that the *callee* may vary for a given INNER in BETA (e.g., INNER in operation may call any submethod of operation, e.g., withdraw).

In gbeta, the restriction on the caller of a submethod has been removed. The language gbeta is a generalization of the language BETA. The gbeta concepts are generalized versions of corresponding BETA concepts, so we use the words 'method' and 'class' about patterns in the same way as in BETA, and for the same reasons. The most important difference between BETA and gbeta in this context is that a pattern [Ern99a, Ch.3] in gbeta is an ordered composition of mixins, whereas a pattern [MMPN93, Ch.3&6] in BETA is constructed by single inheritance from a statically known superpattern. This means that every BETA submethod has exactly one supermethod from which it can be called, whereas in gbeta every mixin may be called from an unbounded set of mixins. Hence, the caller of a gbeta mixin may vary, as well as the callee. This makes INNER in gbeta similarly powerful as ordinary object-oriented method invocation with late binding, where a given method may be called from multiple places, and a given call site may invoke multiple method implementations. However, these two mechanisms represent very different trade-offs in many other respects, so a detailed comparison is required. We will present such a comparison in Sec. 4.

Method mixins
```
mixin void m_operation(Account a)
  { int tmp; a.lock(); INNER; a.unlock(); }
mixin void m_withdraw(int amount) △
  { tmp = a.getBalance()-amount; a.setBalance(tmp); }
method withdraw = m_operation ⊕ m_withdraw;
```
Box 3

Box 3 shows the usage of gbeta mixins to express an equivalent withdraw method as the one in Box 1. It is again given in a Java-like syntax for readability; '$\oplus$' is the mixin composition operator. The symbol '$\triangle$' stands for a specification of the requirements by the mixin on other mixins, i.e., what it expects to inherit. If there is no such specification, m_withdraw cannot be type checked, because the names a and tmp are not declared anywhere. It would still be possible to type check compositions such as `m_operation` $\oplus$ `m_withdraw`, but it would be very hard to type check dynamic mixin compositions. One solution is to let $\triangle$ be a set of requirements on attributes in super-mixins; each mixin would then have a composition type according to its requirements, and the types of mixins would be used to check that all requirements are satisfied when mixins are composed. In gbeta another approach is taken—the gbeta type system [Ern01, Ern99a] is based on structural typing at the level of mixins, not at the level of individual declarations, and $\triangle$ denotes a list of mixins. Any composition that includes *at least* these mixins will make the body of this mixin type safe. We may then type check and generate code for each mixin separately.

A *method mixin* denotes a gbeta mixin, i.e., one of the building blocks in a gbeta pattern. The word 'method' is included to emphasize that this pattern is intended to be used as a method. This concept is novel and useful because it provides an alternative to known invocation mechanisms, and this is not the case with other kinds of mixins. The concept can easily be broadened in order to be introduced into various different programming languages, which we will illustrate in the following by giving the informal semantics of method mixins without reference to BETA or gbeta:

A method m is a list of building blocks called method mixins, m_i, i.e., $m = [m_1 \dots m_k]$. Each method mixin m_i has an associated structure, $S(m_i)$, such as arguments and local variables, and it has an associated behavior, $B(m_i)$, which is a list of instructions (using $S(m_i)$). A method invocation is an instance of a method m. Its structure $S(m)$ is the complete activation record, which combines the structure of all the mixins $S(m_1) \dots S(m_k)$. The behavior of the method invocation, $B(m)$, is created by composing the behaviors $B(m_1) \dots B(m_k)$. Each method mixin behavior is parameterized with 'the meaning of `INNER`'. Let $\emptyset$ denote the null behavior (no operation), and let $B'(m_i)$ denote the composed behavior of all mixins with index i or greater. Then $B'(m_k) = B(m_k)(\emptyset)$, $B'(m_{k-1}) = B(m_{k-1})(B'(m_k))$, etc. up to $B'(m_1) = B(m_1)(B'(m_2))$, and finally $B(m) = B'(m_1)$.

Execution of an `INNER` statement may also be described as a jump-subroutine instruction from the current mixin m_i to the next mixin m_{i+1}. When the behavior of m_{i+1} terminates, the behavior of m_i is resumed just after the `INNER` statement.

3 Examples

Separation of concerns is the most profound reason why method mixins are useful in addition to ordinary method invocation. The concerns are transfer of information and transfer of control. Ordinary method invocation tightly couples the two, whereas method mixin invocation separates them. The example in Box 4 demonstrates that this makes a difference in practice.

Wrapper example

```
class Account {
  void withdraw(int amount)
    { setBalance(getBalance()-amount); }
  void lock() { ... }
  void unlock() { ... }
}
class StdSafeAccount extends Account {
  void wrapper(Closure cl) { lock(); cl.doit(); unlock(); }
  void withdraw(int amnt)
    { Closure cl = new Closure(amnt); wrapper(cl); }
  class Closure {
    int amount;
    Closure(int amount) { this.amount=amount; }
    void doit() { StdSafeAccount.super.withdraw(amount); }
  }
}
class SafeAccount extends Account {
  mixin m_wrapper() { lock(); INNER; unlock();}
  method withdraw = m_wrapper ⊕ super.withdraw;
}
```

Box 4

Class `Account` provides a simple `withdraw` method. Assume that we need to wrap a pair of `lock` and `unlock` operations around the body of `withdraw`. We could duplicate all the method bodies and add `lock`/`unlock`, but we want to avoid such error-prone practices. We want the wrapper as a reusable entity that does not depend on the wrapped method. `StdSafeAccount` shows how we can do this in standard Java, and `SafeAccount` shows how to do it with method mixins.

In `StdSafeAccount` we need to call the original method (`Account.withdraw`) in the body of `wrapper`, and we do not want to depend on exactly what method to call, to give arguments to it, nor to transfer returned results from it. The 'closure' object `cl` makes this possible. The method `withdraw` is overridden to create an instance of `Closure`, initialize it to hold arguments, and give it as an argument to `wrapper`, which will then call `doit` on the closure which will in turn call `super.withdraw`, bracketed by `lock` and `unlock`. If the wrapped method had a return type different from `void`, the returned value could be stored in the closure object and extracted after the call to `wrapper`. So, it is possible but hard to create a `wrapper` that is independent of the signature of the wrapped method.

The signature of a method is a specification of the transfer of information from the caller to the callee and back. The problem with the standard Java approach is that the transfer of information is tightly coupled with the transfer of control: `wrapper` cannot directly call `super.withdraw` without depending on the signature of `super.withdraw`. Method mixin invocation differs from traditional method invocation because `INNER` does not depend on the signature of the callee.

In `SafeAccount`, we can create a wrapped version of `withdraw` by simply composing `super.withdraw` and `m_wrapper`. The signature of the composition is the signature of `super.withdraw`, because `m_wrapper` does not accept any arguments nor return results. The composition operator in gbeta is based on a linearization algorithm that is described in [Ern99b], and the space does not permit a detailed explanation here. However, a similar approach could be used to add method mixins to other languages. The actual composition operation is as follows:

$$[\texttt{m\_wrapper}] \oplus [\texttt{m\_237}] = [\texttt{m\_wrapper}, \texttt{m\_237}]$$

The method `Account.withdraw` is denoted by $[\texttt{m\_237}]$ here. The name `m_237` is machine-generated because that method mixin is anonymous in the program.

To summarize, the signature independent wrapper requires an elaborate machinery in standard Java, because transfer of information and transfer of control are inseparable in ordinary method invocation. Using method mixins, transfer of information happens when the composite method is invoked, and transfer of control happens when `INNER` is executed in the wrapper. Thus, the wrapper is easily kept independent of the wrappee.

We now outline a number of useful kinds of method mixins, leaving pieces of code unspecified by using symbols $\mathcal{A}$ and $\mathcal{A}'$ to stand for sequences of statements that do not contain `INNER`, and $\mathcal{B}$ to stand for boolean expressions that do not contain `INNER`. See Box 5.

Five archetypes

```
mixin m_before() = { 𝒜; INNER; }
mixin m_after()  = { INNER; 𝒜′; }
mixin m_around() = { 𝒜; INNER; 𝒜′; }
mixin m_maybe()  = { if (ℬ) INNER; }
mixin m_repeat() = { while (ℬ) INNER; }
```

Box 5

The first archetype, `m_before`, is very common. When composing `m_before` with a method `foo` as in `m_before` $\oplus$ `foo` the code $\mathcal{A}$ runs before the body of `foo`, like a `before:` method in CLOS [Kee89]. This provides simple, sequential composition of behavior, using shared state: If $m_1 \ldots m_n$ are of this kind then the method $[m_1 \ldots m_n]$ runs the actions in m_1, let us call them $\mathcal{A}_1$, followed by the actions $\mathcal{A}_2$ in m_2, etc.

For the second archetype, a similar method $[m_1 \ldots m_n]$ will execute the actions in the order $\mathcal{A}'_n, \mathcal{A}'_{n-1} \ldots \mathcal{A}'_1$, i.e., in *reverse* order, similar in effect to an `after:` method in CLOS.

The mixin composition operation does not depend on whether it is composing method mixins of type 1, type 2, a mixture, or something entirely different, so a method mixin writer can specify composition order rules separately from the composition itself.

The third archetype combines the two first types by having actions both before and after the `INNER` statement, and the result is that the actions are executed in a "parenthesized" order: $\mathcal{A}_1$, $\mathcal{A}_2$,… $\mathcal{A}_n$, $\mathcal{A}'_n$, $\mathcal{A}'_{n-1}$,… $\mathcal{A}'_1$. Of course, `m_wrapper` in Box 4 is an example of this.

The fourth and fifth type of method mixins can be used to control the execution of other method mixins; `m_maybe` is able to determine whether or not the remaining (more specific) method mixins should be executed or not, and `m_repeat` is able to repeat the execution of the remaining method mixins until some criterion ($\mathcal{B}$) is satisfied. These method mixins are reusable building blocks for specific aspects of behavior, which is a novel reuse domain.

4 Comparison

A method mixin (MM) has behavior and local state, just like a method, and it has an enclosing object, just like a method. However, it is invoked by `INNER` rather than by name, and it does not accept arguments or return results at the call site. It communicates with other method mixin instances (MMIs) via shared (inherited) state in other MMIs. It is also possible for method invocations to communicate with each other by means of shared state, e.g., state in a common receiver object; but the shared state of a list of MMIs is not located in an enclosing object, it is located directly in the MMIs.

We have described MM composition as mixin composition extended with behavior composition. This means that the semantics of MM composition depends on the underlying semantics of mixin composition. In particular, it makes a difference whether or not mixins can be composed dynamically, and whether or not an existing object (possibly an activation object) can be transformed by, e.g., adding part objects corresponding to additional (method) mixins. Both of these facilities are available in gbeta, subject to some constraints that are required for type safety [Ern99a, Ch.7]. However, it is easier to ensure good performance when such dynamic features are known to be unused.

A crucial difference between method invocation and MM invocation is the transfer of information. For methods, this happens as part of the invocation, but for MMs there is greater flexibility. Let us consider the possibilities, leaving the mixin composition mechanism very open by just saying that MMIs on the call stack may access attributes in other MMIs further up the stack. One possible choice is to let access be unlimited, such that the entire call stack may be searched for any name that is not defined locally. This is effectively the same as dynamic scope, as in some old versions of LISP. This would however be very hard to make type safe, and it has obscure and surprising semantic properties. For this reason, MMIs are grouped, and every lookup across MMIs is limited to be inside the group. Hence, the MMIs in a group together are an entity—an object, or an activation record if you wish.

Usage of attributes across MMI boundaries may also vary. With a per-declaration structural typing of mixin requirements (cf. $\triangle$ and Box 3), the same name application in a given MM may resolve to many different declarations. This ensures flexibility, but we suspect that it gets harder to understand the effect of code in an MM. As mentioned, mixins in gbeta require specific other mixins, thereby ensuring that the applied names refer to statically known declarations. There is still late binding, but we know exactly which declaration we are using the late-bound value of.

Another design decision is concerned with the life-time of MMIs. An activation record lives from the invocation to the termination of its call, but MMIs may not have to live exactly for the same period as the method invocation that they are part of. Obviously, if an

MM m_j depends on state in an MM m_i then m_i must exist at least at every moment where m_j runs. It would be sound to allow any kind of life-times that respect this criterion. In gbeta we have chosen the following approach: an MMI group can be extended, but no members can be taken out once they are in. This is because other properties of the language—in particular virtual patterns—would make the type analysis unsound if it were possible to remove an MMI from a group.

At this point we can compare MMs with ordinary methods from several different points of view, and that is what we will do in the next section.

4.1 Method Mixins vs. Ordinary Methods

MMs are a supplement to—not a replacement for—ordinary methods. They are so different that programmers should make the trade-offs explicitly. To characterize the purposes for which each is most suitable, consider two different points of view. External: Once we have written an entity that solves a particular problem, it may be reused many times by being accessed from many different contexts. This point of view emphasizes the use of a *preexisting* entity; we call it *black-box* reuse. Internal: An entity may have a general structure that allows it to solve many different problems, if only we are able to express variants. This point of view emphasizes *creation* of variants of a flexible entity; we call it *white-box* reuse.[1] To apply this to behaviors, consider a method or MM 'm'. Black-box reuse focuses on being able to call m flexibly; white-box reuse focuses on flexibly controlling what methods/MMs m calls.

The two kinds of reuse complement each other. Black-box reuse is governed by an explicit interface and is often easy. However, it does not help much if a suitable entity is not available. White-box reuse may then allow us to *build* the kind of entity that is needed. White-box reuse can be more difficult than black-box reuse, in the sense that it usually requires more knowledge about the internal structure of an entity to create a specialized variant of it. In return for this investment, white-box reuse gives us flexibility. It is also reasonable to say that black-box reuse is a large scale mechanism, because a reusable entity could be reused from anywhere in a large scope, e.g., an entire program. On the other hand, white-box reuse is focused on the internals of the entity that is being specialized. We believe that ordinary methods are the most suitable choice for black-box reuse, and MMs are more powerful for white-box reuse. In the following we will compare MMs and ordinary methods from various different points of view, thereby also exposing their weak and strong points.

4.1.1 Communication.

Ordinary methods may communicate with each other by means of arguments and returned results, or by means of shared access to mutable state in some reachable entity, e.g., global

[1]The terms black-box reuse and white-box reuse are used in many contexts, including [GHJV95]; it is not quite clear who coined these terms.

variables or instance variables in an object. Arguments can transfer information from a caller to an immediate callee, and return values can transfer information in the opposite direction. Shared state can be used for communication across multiple activation records, but it is error-prone if there are resource conflicts, e.g., when recursion or multiple threads cause more than one invocation of those methods to exist simultaneously. If an argument is of reference or pointer type then it may be used to establish aliased access to mutable state. Aliases may be used in a similar way as other kinds of shared state. They may have fewer resource conflicts than state in an enclosing object or global state, but they require declaration of a corresponding argument in all methods on a contiguous part of the stack, even if some methods do not use it.

MMIs communicate with each other by means of shared state in the MMIs themselves; they may of course also use shared state elsewhere. Communication via local shared state is considerably more flexible than communication via arguments. In particular, communication may happen across multiple, possibly oblivious MMIs. For instance, `m_repeat` in Box 5 can be used to repeat a part of an method until some condition in the enclosing object is satisfied, and the other MMIs may depend on each other across the repeater without having any knowledge about it. Conversely, the repeater need not depend on the name space inside that method at all. Similarly, the mixin `m_wrapper` in Box 4 is independent of the information passed to the body of `Account.withdraw`. The class `stdSafeAccount` illustrates how much machinery is needed to obliviously pass information through a method such as `wrapper`, and that is not even entirely obliviously because `wrapper` handles the closure object.

4.1.2 Callee dependencies.

Consider the situation where an ordinary method `foo` is called. This is achieved by writing the name `foo` at the call site, along with an explicit argument list. The call may also be used as an expression if `foo` returns a result. This means that the caller depends on the number, order, and types of arguments; on the name of the callee; and on the result type of the callee. Methods are not first class entities in many object-oriented languages, including Java [AG98] and Eiffel [Mey97], so the only way in which the implementation of `foo` may vary is by late binding. In other languages such as C++ [Str97] we may also use function pointers or similar devices, but the dependency on the signature of the callee is common to all cases.

MMs are very different in this respect. An invocation of an MM is simply the keyword `INNER`, and the caller is entirely independent of the callee. There is no mention of a name for the callee; there are no arguments; and there is no returned result. It is even possible that there *is* no callee, in which case `INNER` will do nothing. This means that an MM is in a sense maximally flexible with respect to the callees that can be attached to it.

4.1.3 Caller dependencies.

Consider the task of writing an ordinary method `bar`; part of the job is to specify the exact arguments and their types, as well as a return type. This works well as an interface to the callers because it is explicitly defined and it allows the body of the method to use information received from callers without knowing anything about them. Hence, an ordinary method does not depend on its callers.

On the other hand, if `bar` needs more information in order to carry out the prescribed task then there is no way to get more information out of the callers except for changing the interface of `bar` and then changing all call sites, e.g., to add some new arguments. This is especially laborious if the `bar` we are working on is one of many implementations of `bar` in many different classes—all those method implementations must then have the new arguments added to their declarations, even if few of them use it.

An MM is again very different. It may depend on declared names in one or more of the MMs in the same method. This means that an MM can only be called when caller MMs provide a suitable name space, and that it is necessary to type check MM composition in order to ensure that all required names are indeed defined, with appropriate types. On the other hand, the MM uses names in its callers one by one, and it is not affected by changes applied to names that it does not use. For instance, we could add a new declaration to a caller and then use that name in one or more callees; all the other callees could remain unchanged.

In summary, an MM may depend on an inferred (partial) interface of some of its fellow MMs. Different MMs may depend on a given MM via different inferred interfaces. In contrast, an ordinary method uses information exclusively from the immediate caller, and via an explicitly defined interface. Different methods called from the same call sites must agree precisely on the interface that they present to those call sites.

4.1.4 Encapsulation.

Ordinary methods are encapsulated in relation to each other, in the sense that a caller knows nothing about the callee except for its name and signature. However, the encapsulation of a group of methods that have a need to communicate with each other may be broken by the reliance on external shared state. When a method changes the state of its enclosing object it may be conceptually well-motivated, but when object state or global state is used mainly for the transfer of information between method invocations then it becomes error-prone.

For communication, MMIs support an exact match between the involved MMIs and the life-time and scope of the state that is used for this communication, hence they enable an improved encapsulation towards the external environment. On the other hand, MMIs are able to use the internal structure of each other, hence they are mutually unencapsulated. In other words, they are optimized for intense collaboration and white-box reuse of each other. Of course, access declarations (`private`, etc.) may be used to make the white box turn gray, thus imposing some discipline upon the collaboration. Considering that methods are generally simpler entities than classes, this style of encapsulation support is arguably sufficient for methods because it is well-known and works for classes.

Note that it is possible for MMIs to contain state that is used throughout the life-time of the MMI group, but not depended upon by all members. A method could for instance include an m_maybe MM (see Box 5) that declares and uses its own state to determine whether or not to invoke the remaining MMs. This would affect the semantics of the method as a whole, but the other MMs in the method need not depend on that local state.

4.1.5 Life-time.

Ordinary method invocations have nested life-times: A caller lives at least as long as any of its callees, and possibly much longer. The number of invocations on the run-time stack is (practically) unbounded. This makes it possible to organize an entire program execution in terms of traditional invocations.

MMs are composed into groups and then invoked as a unit. Different design decisions could be made with respect to the ability to add or remove members from this group during the execution; gbeta supports addition of new members, but execution must then be restarted, so it is recommended to finish building the MMI group before executing it. When the topmost member of the group terminates, the whole MMI group is considered terminated.

Hence, MMs can not be the behavior structuring principle for an entire program execution. Also, MMI groups are expected to consist of relatively few members because they are optimized for intense collaboration and openness.

4.1.6 Kinds of entities communicated.

A method argument may only be an entity of certain kinds in some languages, usually known as “first-class” entities. For instance, a Java method cannot receive a method as an argument, and a class can only be given as an argument if it is accessed via the reflection system, which means that type safety is lost and special syntax must be used to create instances of that class, etc. With MMs, the communication takes place by means of shared name spaces. Hence, anything that can be declared can also be communicated.

4.1.7 Name spaces.

An ordinary method provides a name space by means of the formal argument list, and it is up to the caller to initialize this name space. It is possible in some languages, e.g. C++, to define default values for some of the arguments (argument number N and upwards for some N). This makes it possible to call a method with fewer arguments than the argument list contains, corresponding to the situation where the caller “does not say enough” about the arguments. Note that this only provides information to the callee that the callee already had in its declaration.

An MMI cannot be called from a group of callers that “do not say enough”, that would violate the ($\triangle$) requirements check at mixin composition time. This restriction might be lifted by introducing a notion of “default declarations”, but we have not investigated this possibility in details. However, the fact that an MM may choose to use only some of the declared names in statically known callers allows another phenomenon, namely that the callers "say too much". This enables us to change a callee to use more of the provided information at the call site without having to synchronize with all the other callees. As usual, MMs emphasize the ability to flexibly enhance the effect of its callers, whereas ordinary methods emphasize the ability to be called from arbitrary call sites.

Common Lisp [GLS90] supports the notion of a `&rest` parameter, i.e., a name that is bound to the list of arguments given at the current call site in addition to the explicitly declared parameters. Similar mechanisms are provided in Java 1.5 and in C++ by means of an ellipsis, '...', representing any number of additional arguments. This seems to enable callers to "say too much", but since it is just a list of nameless values, it is essentially just syntactic sugar for an ordinary argument of type `List`. To illustrate the difference: if two extra names are provided by an MM caller, then callees can use one of them without being affected by the existence of the other new name; if a Common Lisp call site provides two arguments to the `&rest` parameter then the callees must know about both in order to be able to use the second one—the callee would *not* be unaffected by the provision of the unused argument.

4.1.8 Invocation.

An ordinary method is invoked by calling it. It does not involve other methods, except if it takes one or more arguments of type 'function pointer' (or whatever it is called in the given language) and calls it.

MMs must be composed into an MM group, a method, which is then invoked. One of the MMIs is positioned as the topmost one, and that one is called when the group as a whole is called; the others may or may not be called using `INNER`.

Because it makes a given MM call another one, the operation of composing MMs may be compared to a higher-order function in a functional language. However, such a higher-order function would have to rely on a traditional call chain as described earlier: transfer of information in a pure functional language is even more strictly confined to arguments and returned results than it is in imperative and object-oriented languages.

5 Related Work

Since the comparison between MMs and ordinary methods was the main topic of Sec. 4, a kind of related work has already played an important role. However, some other mechanisms still need to be considered.

The language BETA [MMPN93] is important in relation to everything in gbeta, but the similarities and differences have already been described in earlier sections. Briefly, gbeta generalizes the BETA concepts and mechanisms in such a way that `INNER` becomes comparable in power to ordinary method invocation (though different in nature).

A language mechanism that seems to be usable for similar purposes as MMs is the *macro* mechanism in Common Lisp [GLS90]. Such macros are capable of manipulating source code (not individual characters, but syntax trees) in an extremely flexible manner, because it is based on program transformations by means of arbitrary user-defined functions. This very general transformation capability makes it hard to ensure that the output from a macro has known properties. E.g., in a host language that otherwise supports static type checking, such as gbeta, there would hardly be any other way to check typing properties of a macro call than to actually expand the macro. This indicates that macros are unlikely to be able to coexist with static typing, and at the same time be able to be expanded dynamically. MMs in gbeta can be composed dynamically, and they do not conflict with static type checking.

Dylan [Sha97] macros are similar, but *hygienic*, which means that they do not allow names used in macro calls to be captured in a new binding environment associated with the macro definition, or vice versa. It is possible to 'intentionally violate' the hygiene, e.g., to let a binding of a name in a macro definition be used by expressions in macro calls, but this is considered to be a dangerous practice. In other words, macros are not intended to combine behaviors having shared access to a name space, which is a core idea for MMs.

The *composition filters* [AWBB94, Ber94] approach allows for very flexible adaptation of the effect of sending messages. Composition filters can be used to redirect a message to another receiver, they can be used to change the selector (method name), and they can be used to modify arguments in a message. There are also some special filters for other purposes, e.g., for delaying a message until some condition is satisfied. This approach is largely orthogonal to MMs, because method mixins are concerned with the *construction* of methods, whereas composition filters are concerned with *selection* of which method to call, and how.

Aspect Oriented Programming [FECA05] is also an approach that aims to provide new abstraction mechanisms in programming languages. An AspectJ [KHH$^+$01] aspect could be used to transfer information from one method activation to another, across oblivious invocations. In particular, a `cflow` pointcut in a `percflow` aspect would provide a location to store such information at a particular point on the call stack, and a way to retrieve it from invocations further down that call stack. This is rather costly in terms of performance and also less well integrated into the method call mechanism than method mixins, but it may prove useful as a small design pattern in cases where method mixins would have been an obvious solution.

There has been much work on (class) *mixins* since the seminal paper [BC90] where mixins were established as a separate concept. We do not know of any other language than gbeta where mixins are used as composable method building blocks, so the connection between gbeta mixins and other mixins is not very close in context of this paper. However, we should mention a few points in this area.

In [SCD$^+$93] the notion of a *mixin method* is introduced. Execution of a mixin method makes the receiver object change class and, e.g., obtain some new instance variables. In other words, mixin method invocation is the mechanism that is otherwise known as mixin application (we used the term mixin composition), and mixin methods are therefore unrelated to our concept of MMs (method mixins). On another related point, [FKF98] presents an extension of a subset of Java with mixins, called MIXEDJAVA. In this paper they introduce the notion of an *inheritance interface*, specifying the requirements of a mixin on potential superclasses. In gbeta, the inheritance interface is expressed via the set of required mixins that each mixin must have available. In both cases the mechanism ensures statically that name lookup will indeed succeed at run-time, and the looked up entity will have an appropriate type.

6 Implementation Status

The language gbeta implements method mixins, and the source code is available[2] under the GPL license. The syntax in this paper has been adjusted to be Java like and we omitted some details like module import statements, but apart from that the examples correspond to actual running code in gbeta.

7 Conclusion

We have presented the notion of method mixins; compared their characteristics extensively with those of traditional method invocations; and argued that method mixins represent a substantially different trade-off. Method mixins are optimized for white-box reuse, i.e., the creation of variants of behaviors based on flexible building blocks. Traditional invocation is optimized for black-box reuse, i.e., invoking existing functionality as-is, from many different places. We believe that these two mechanisms supplement each other well. Some special characteristics that make method mixins attractive are as follows: With method mixin invocation, the caller is completely independent of the callee; with traditional invocation, the caller depends at least on the signature of the callee. With method mixins, communication is based on an inferred signature that allows callees to use different subsets of the available information; with traditional invocation the communication via arguments and returned results must be identical on all callees. With method mixins, communication is free to occur across oblivious intermediate method mixin instances; traditional arguments and returned results only support communication between a caller and its immediate callee. Method mixins have been implemented in the language gbeta, where they are tightly integrated with other features of the language, but the basic ideas could be used in many other languages.

References

[AG98] Ken Arnold and James Gosling. *The Java™ Programming Language*. The Java™ Series. Addison-Wesley, Reading, MA, USA, 1998.

[AWBB94] M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting Object Interactions Using Composition Filters. *Lecture Notes in Computer Science*, 791:152++, 1994.

[2]`http://www.daimi.au.dk/~eernst/gbeta/`

[BC90] Gilad Bracha and William Cook. Mixin-based Inheritance. *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, 25(10):303–311, October 1990.

[Ber94] Lodewijk Bergmans. *Composing Concurrent Objects*. PhD thesis, Faculty of Computer Science, University of Twente, 1994.

[BPS99] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A Core Calculus of Classes and Mixins. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, New York, NY, June 1999.

[Ern99a] Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.

[Ern99b] Erik Ernst. Propagating Class and Method Combination. In Rachid Guerraoui, editor, *Proceedings ECOOP'99*, LNCS 1628, pages 67–91, Lisboa, Portugal, June 1999. Springer-Verlag.

[Ern01] Erik Ernst. Family Polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP'01*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2005.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 January 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[GLS90] Jr. Guy L. Steele. *Common Lisp – The Language*. Digital Press, Digital Equipment Corporation, 2nd edition, 1990.

[GR89] Adele Goldberg and David Robson. *Smalltalk–80: The Language*. Addison-Wesley, Reading, MA, USA, 1989.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, USA, 1989.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings ECOOP'01*, LNCS 2072, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Classification of Actions, or Inheritance also for Methods. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP'87*, LNCS 276, pages 98–107, Paris, France, June 15-17 1987. Springer-Verlag.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.

[pwimaBu] Web page with introductory material about BETA: `http://www.mjolner.dk/mjolner-system/documentation/tutorials.ht%ml`.

[SCD$^+$93] Patrick Steyaert, Wim Codenie, Theo D'Hondt, Koen De Hondt, Carine Lucas, and Marc Van Limberghen. Nested Mixin-Methods in Agora. In Oscar M. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 197–219. Springer-Verlag, 1993.

[Sha97] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.