Restructuring optimisations for object-oriented mobile applications

Pietu Pohjalainen
University of Helsinki,
Department of Computer Science
ppohja@iki.fi

1 Abstract

The Java programming language has made its way to a number of mobile devices. In theory, the language endorses the write-once run-anywhere scheme, but in practice applications written for mobile devices are limited by hardware and software constraints such as small memory and limited bandwidth in deployment.

Writing applications in object-oriented style usually brings additional static and dynamic overhead to programs. This leads to developers forgetting their good manners, as the constrained environment is used as an excuse for writing unmaintainable code.

This paper describes class-hierarchy restructuring methods for removing overhead introduced by object-oriented framework constructs. In the deployed package, every included class consumes certain amount of space. By removing unnecessary hot spot interface classes at linking time, the total number of classes is reduced, which leads to smaller total space consumption.

We present two algorithms for class inlining and initial results on applying them to an application framework based product-line.

2 Introduction

Object-oriented application frameworks have long been seen as a good way to achieve reuse in software engineering. When developing such a framework with several derived applications, the reused parts of the software will receive more throughout testing, thus increasing the overall quality of the common code and allowing faster time-to-market for the developing organization.

Developing multiple software products with shared core components and some application specific functionality using application frameworks is called a product-line approach [Bos00, CN01]. A usual implementation of the approach consists of first implementing the desired core components and then defining a reuse interface, also known as hot spots [Pre99] for application specific code. While the approach gives flexibility in the development process, it comes with a runtime cost: every method defined in the framework specialization interface needs to be lately bound; moreover, the usual implementation of these hot-spots also includes additional interface classes to be introduced to the framework.

While all of the overhead introduced in framework abstraction cannot be avoided, some of these bindings can be bound at program deployment time or at link time. This is especially the case when the framework interfaces are used to encapsulate differences that exists only during development and are not anymore present in the runtime execution package.

This paper discusses methods of automatically optimising away these unnecessary framework hot spot classes, with the intention to not only to improve the execution times of the program, but to decrease program static deployment size as well. The rest of the paper is organized as follows. Section 3 defines the problem area and discusses its standard solutions. Section 4 shows how certain unnecessary overhead usually intro-
duced in the framework hot spot interfaces can be reduced. Section 5 discusses the software engineering principles that are cause of the attacked overhead. Section 6 shows our experimental results after algorithms presented in the previous section being applied to several Java 2 Mobile Edition programs. Section 7 summarizes some of the related work in this problem area. Finally, section 8 gives a conclusion of this paper.

3 Problem overview

In application frameworks, it is a usual habit to encapsulate variations in derived applications behind an interface that defines the abstract service interface of the encapsulated objects. In this paper context, examples of such behaviour could be playing of sounds, variations being MIDI capability versus simple sound generators; or different screen sizes, variations for small, medium and big screen sizes. Implementing this kind of functionality in the Java programming languages [GJSB00] is commonly performed using interface classes.

While in normal software products, being targeted to be used in resource rich desktop computers, the runtime price of one or two extra interface classes per framework hot spot is certainly feasible in exchange for achieved additional flexibility. However, this is not always the case in software products being targeted to mobile computation platforms, where the available resources are often very scarce. In practise, when developing software to the mobile phones, every kilobyte in the software size does count, as almost every aspect in development, deployment and execution of the programs is somehow constrained.

However, despite the initial belief of unnecessary bloat, the product-line approach can be efficiently implemented using this kind of interface constructs for handling the deviations in the common computation platform, as the specialization interface can be seen as a development-time temporary construct. An intuitive implementation of hiding differences between two hardware device models or vendor APIs is to introduce a common interface and implementing classes. Figure

1 shows an example of such a situation: a generic sound playing capability is encapsulated behind an abstract interface SoundInterface.

This organization of the source classes allows the encapsulated implementation to be changed without affecting any of the client code, thus allowing each derived application to follow the common structure of the product-line architecture in question. Now the framework only knows the abstract usage interface of the application code, and the question of building final deployment package can be delayed to be done on a separate product-line configuration phase.

Figure 1: Class hierarchy for two different sound capabilities MIDI and Beep

Figure 2: Distinct class hierarchies in two deployment packages Adv(anced) and Basic

In this situation, the interface with multiple implementations is introduced with the intention to model a concept that varies in the target plat-
forms. During the development, benefits of strongly typed language can be gained, but at deployment time, there is only one implementing class per each modelled concept.

The product-line configuration phase selects the classes that will be contained in the final deployment package and configures the object creational code to follow the chosen configuration. In this example, the two implementations of the common interface are meant to model the differences in two different vendor APIs or hardware platforms - and thus they are parts of different derivations of the product-line and will not reside in the same deployment package. For this reason, two deployment packages will be created, as shown in Figure 2.

<table>
<thead>
<tr>
<th>SoundInterface</th>
<th>MidiDevice.class</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sTime = 10</td>
<td>int sTime = 10</td>
</tr>
<tr>
<td>void playSound()</td>
<td>void playSound()</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>midi.play(sTime);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Before inlining  =>  After inlining

**Figure 3:** Merging an interface to a class

In addition to changing the interface implementing class, it is also necessary to change every call site in the framework and application code that accesses the concrete implementation through the interface that will be removed. While the product-line approach states that general framework code shouldn’t be affected by application specific changes, this optimisation doesn’t violate this rule, as it turns out that this merging of interface and implementation classes can be performed automatically using whole-program level class hierarchy analysis (CHA) at program build time.

For this reason, source code of a derivation of framework can reside unaffected in the process. Instead, the restructuring is seen as yet another optimisation phase during compilation of the code: it is integrated to the building process of applications and the restructuring accesses only object code of the compilation.

In the next section, we will present algorithms for merging interface and abstract classes to the implementing concrete (derived) classes in Java. Although this implementation is specific to Java, the underlying idea can be applied to any object-oriented language that employs the notion of classes as the inheritance mechanism.

### 4 Algorithms

In the following algorithms, the whole program is analysed and modified in the Java bytecode form [TF99]. The actual implementation uses Bytecode Engineering Library [Dah01] as the bytecode-level handling machinery.

Our solution to the specialization interface overhead relies on analysing class relationships at the whole-program level. In short, an interface and the implementing class can be merged together if there is only one class that implements a given interface; an abstract class can be merged to a concrete class if there is only one class that extends the abstract class. This analysis is a rather simple traversal of the program class hierarchy tree.

A bigger problem is to the changing of all usage
sites. When merging an interface or an abstract class into a concrete class, all call sites in the program need to be updated to reflect the optimised situation. Changing all call sites can be done with the algorithm presented in Algorithm 1.

**Data:** interFace is an interface, or abstract class

**Data:** cls is the only implementing class

**Data:** list is the list of all classes in the program

foreach MemberVariable var in interFace do
  newName ← interFace.name_var.name;
  cls.{newName} ← var;

foreach Class curCls in list do
  foreach Field F in curCls do
    if TypeOf(F) is interFace then
      TypeOf(F) ← cls;
  endforeach
  foreach Method M in curCls do
    if ReturnType(M) is interFace then
      ReturnType(M) ← cls;
    endif
    foreach Argument A in M do
      if ArgumentType(A) is interFace then
        ArgumentType(A) ← cls;
      endif
    endforeach
    foreach Instruction instr in M do
      switch instr I type is do
        case INVOKEINTERFACE
          if instr.target is interFace then
            instr.opcode ← INVOKEVIRTUAL;
            instr.target ← cls;
          endif
        case FieldInstruction
          if instr.target is interFace then
            instr.target ← cls;
          endif
      endswitch
    endforeach
  endforeach
endfor

Algorithm 1: Merging an interface into a class

The algorithm works as follows. First, it lifts all fields defined in the specialisation interface to the only implementing class. Then, every field in every class that are of the type interFace are changed to be of type implementation class. Then, the argument and return types of each of the methods in every class are changed to be of the implementation class type, whenever they previously were of the type interFace. Finally, every instruction calling any method via interface definition (Java’s invokeinterface instruction) are redirected to the implementing class and each interface field access is rerouted to the implementing class.

After this algorithm has been executed, there is no more notion of the type interFace present in the program; all references to that type are replaced with the type of the implementing class.

![Abstract class in class hierarchy](image)

**Figure 4:** Abstract class in class hierarchy

The same idea can be extended to also handle the case of an abstract class and one implementing concrete class. In this case, the algorithm needs to analyse whether the implementing class overrides methods in the abstract class and whether it calls the overridden method. In the latter case, the merging algorithm needs to concatenate the method body of the overridden algorithm to the calling site; otherwise the overridden method can be removed, as it cannot get called. This situation is depicted in Figure 4.

The merging algorithm of an abstract class and concrete class can reuse the functionality in Algorithm 1 by calling it as the first step. The rest of the algorithm is shown in Algorithm 2.
Data : Abstract class: abstCls
Data : Class: cls
call(Algorithm 1);
cls.extends ← abstCls.superclass;
foreach Method M in abstCls do
    if cls.M overrides abstCls.M then
        if cls.M invokes abstCls.M then
            cls.M ← concatenate(abstCls.M, cls.M);
        else
            cls.M ← abstCls.M;
    else
        cls.M ← abstCls.M;
removeClass(abstCls);

Algorithm 2: Merging an Abstract class into a Class

5 Discussion

As briefly discussed before, our goal is to allow developers of mobile applications to gain the benefits of product-line architectures in their constrained execution environment. This is done by removing unnecessary intermediate classes that are used only as a development-time flexibility adding vehicles, i.e. that have no execution-time function of late binding.

Maybe the most important application area for these class-hierarchy reorganising optimisations are different object creation patterns. Let us consider the Abstract Factory pattern [GHJV93], being implemented as shown in Figure 5.

In this setting, an abstract factory interface is introduced to client code to instantiate products. The abstract product interfaces define operations for the instantiated products. A usual application of this pattern in mobile applications is to hide the differences in actual hardware and software platforms, such as screen size and capabilities offered by different vendor APIs.

This structure is clearly flexibility targeted to development-time easiness. Once the product-line configuration phase is done, only one concrete factory and one concrete product is placed to the final program. In this situation, the intermediate interfaces AbstractFactory and AbstractProductA no longer serve any purpose and can be removed by applying the Algorithm 1.

Similar benefits can be achieved when using the Factory Method-pattern [GHJV95] as an alternative solution to the product-line configuration problem. Many behavioral patterns can also gain from the optimisations presented here. When considering the Template Method-pattern [GHJV95], as shown in Figure 6 as an instance, an abstract class implements a skeleton operation which calls concrete implementations of primitive operations, that are defined in concrete subclasses.

In mobile applications, this is again another good way to hide differences in execution platforms: for instance, there might be different kinds of handsets, of which other model allows music be played in MIDI-format while the other model permits only usage of simple beeps as the sound effect, as shown in 1. These kind of differences can also be hided in the concrete class implementations and let the abstract sound effect and music playing logic be implemented in the abstract class.

In this situation, there again is just one concrete
class per instance of a product-line family program, depending on the capabilities of the handset. Using Algorithm 2 it is now possible to automatically merge the abstract logic with the concrete implementation into one class. Now we are inlining not only method definitions, but their implementations as well. Because the notion of the inlined class disappears after the operation, we could call this concept class inlining as opposed to method and object inlining.

It should be noted that these algorithms are only applicable to products where the closed world assumption holds. This means that when performing the class hierarchy analysis, it is assumed that all possible classes are known at analysis time. Generally in Java, this isn’t true, as the dynamic class loading facilities [LB98] allows the applications to load classes during execution time. For this reason, a program may use classes that do not even exist at the program starting time — and optimisations presented here shouldn’t be applied to programs that perform that kind of tricks unless there is a deoptimisation possibility for reverting back to the original structure.

However, in our application area the closed world assumption holds. A widely used mobile execution platform, Sun’s Connected, Limited Device Configuration prohibits the usage of user-defined class loaders due to security reasons [RTV01]. As a consequence to this, the optimisation algorithm can reason on the workings of system provided class loader.

While our algorithms are expected to be used in forward-engineering manner, it could be used to detect possible instances of software anti patterns [BMIM98]. If the analysed software doesn’t contain inlinable interfaces or abstract classes, it could be a smell of insufficient abstraction boundaries in the produced software.

6 Experimental results

We have implemented a experimental optimiser that performs whole-program analysis for Java 2 Mobile Edition, Mobile Information Device Profile (J2ME/MIDP) programs. The optimiser relies on the fact that dynamic class loading is restricted to be always performed through the system provided class loader.

In the optimisation, the expected savings in static space consumption come from three sources: the size of the removable interfaces and abstract classes; reduction in call site constant pool sizes; and finally in size of actual calling instruction sizes. Although the final size of the deployed application is reduced by compression provided by the Java archive format, there still is a clear correlation with uncompressed class file sizes and compressed deployment archive size.

Static size of an interface class depends on the number of defined methods and number of referenced types. This information is stored in the class constant pool; when compiling with Sun’s Java compiler, an empty interface that doesn’t define any methods takes $81 + (2 \times \text{len})$ bytes where \text{len} is the length of the interface name. Each defined method and referenced type then adds content to the constant pool, size depending on the length of the method names and referenced type names. A diagram showing interface size in bytes on the y-axis and method count on the x-axis is shown in Figure 7. The final class size in the diagram depends on the average method name length: the first bar represents average length of four characters, the second bar shows the size for
Figure 7: Final size of an interface class file with different method name lengths

length of eleven characters and the third bar gives the size for thirty characters.

For instance, if an interface defines ten methods with average method name length of four characters, the final class size would be about 250 bytes. On the other hand, if the interface defines forty methods with average name length of thirty characters, the class file size would be near to 1,750 bytes. However, these calculations don’t account the possible declarations of foreign types and method parameters, but are presented here to give a mileage of the discussed field.

Reduction in call site constant pool sizes happens as references to removed class can make the constant pool more compact, as the total number of classes in the software is reduced, and there is no more need to have any references to the removed class at the call site. However, if all access to the service provided by the interface is performed purely via this interface (i.e. there are no references to the concrete class at all), these references are just swapped and no static size reduction occurs.

Finally, every call site invoking its target via invokeinterface instruction is replaced with a invokevirtual instruction. The format of the interface invoking instruction has a historical overhead of two bytes that is not present in the regular method invocation instruction. For this reason, every interface call resolved at link time to method call saves two bytes in the final class size.

For empirical results, we have experimented on running the algorithms on a real framework-derived application program targeted to the J2ME/MIDP platform. Because the exact results for these optimisations are depending on the development style of the framework, the results are varying: if the application to be optimised doesn’t employ constructs discussed in section 5, the algorithms are of no use. However, when applying these optimisations to programs that are written in product-line style, implementing specialisation hot spots via interface classes or abstract classes, there are observable savings in package static deployment size.

In our case framework, which is targeted to game development, there are four concepts hidden behind an abstract interface: application behaviour mode, execution platform screen size, execution platform sound capabilities and high score behaviour. Of these concepts, all interfaces except the generic behaviour mode can be inlined away. In the derived example application, this gives a saving of 2,200 bytes in a 64 kilobyte deployment package. While the savings are small in terms of percents, the optimisation is still worthwhile: as the most constrained execution platforms are limited to this 64 kilobyte package size, every byte saved by automatic optimisation allows the developers to add more content to their application.

On the other hand, the current structure of the framework has been crafted at a time when there weren’t these kinds of optimisation algorithms available in development. This has led to ‘fear’ of interface usage: while the framework developers know that using the product-line approach gives benefits in the long run, they haven’t been able to afford the overhead introduced in explicitly defined specialisation hot spots.

7 Related work

Among the early work of link-time optimisation is Fernandez’ work on Modula-3 programs [Fer95]. Her work includes the idea of resolv-
ing targets of interface calls at compile time. Analysing side-effects of statements and exploiting that information in optimisations for Java are discussed at [Cla97]. They are also early to consider the effect of closed world assumption on Java programs with reuse; however, they don’t explicitly handle the effect of dynamic class loading on optimisation.

Close relatives of algorithms presented here are object inlining algorithms presented by Dolby et al. [Dol97, DC00] for the C++ language. Their approach is to combine classes that reside in aggregation relationship: i.e. when there is exactly one Item instance per ItemList instance, they can be merged into a combined form of ItemAnddItemList. This idea was further developed and evaluated by various authors, see for example [VJHB02, LH02, Lau01, BK97].

Algorithms for making lately bound method calls resolved more early than at execution time have long been under research in the field of optimising object-oriented programs. Maybe the earliest report of such effort using class hierarchy analysis is from Apple’s Object Pascal language [Com88]. The same approach has then been applied to the Cecil [DGC95], C++ [AH96], and Java [DA99, SHR+00] programming languages.

Hölzle et al. discuss the possibility of optimisations and deoptimisations in the case of dynamic languages [HCU92]. It is a commonly used technique in Java virtual machines; e.g. Sun’s HotSpot compiler\(^2\) performs inlining of interface methods based on class hierarchy analysis at runtime during dynamic compilation [PVC01].

8 Conclusion and future work

We have presented and discussed a method based on class hierarchy analysis for optimising Java framework based programs at link-time. A novel contribution of the paper is to show the possibility to inline an abstract class to the only implementing concrete class, thus reducing total number of classes in the application.

As far as we know, nobody else has proposed the full inlining of abstract class to the only implementing concrete class. This is probably a consequence of novelty of product-line approach to software engineering: without product-lines, there has not been enough demand for development-time flexibility which will not be needed during run-time anymore. Yet another reason might be the closed world assumption not being true in the general case: with dynamic class loaders being available for use in frameworks, performing any reasoning based on class hierarchy cannot be performed until execution time.

Introducing a link-time optimisation phase to software engineering process based on product-lines allows the developers of the product-line framework to define purely development-time constructs to their code. This supposedly leads to better designs of the frameworks, and thus enables better reuse in framework engineering.

Our future work will contain more throughout evaluation of experiences with the presented optimiser: how its availability will affect structures of the developed product-line frameworks. On the other hand, new link-time optimisations can be developed. A fruitful direction could be to explore concrete type inferencing systems that would employ program control flow to find out the concrete class of polymorphic variables. Instances of known algorithms are presented at least by [Age95, PR94, PC94, GHM00].

Using a more precise than analysis technique then class hierarchy analysis would allow more interface invocations to be converted to regular virtual invocations, helping to statical size as well as to execution time. Moreover, if every invocation via a certain interface can be rerouted, then this interface can be removed, again saving in statical size. However, implementing and experimenting with these algorithms is currently out of this project’s scope.

\(^2\)Not to be confused with hot spots in framework specialisation interface
References


