VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5341
Fax: +64 4 495 5045
Internet: Tech.Reports@mcs.vuw.ac.nz

Explaining Issues in Software Reusability via an Audio System Analogy

Robert Biddle and Ewan Tempero

Technical Report CS-TR-97/3
July 1997

Abstract

This paper presents a novel way to introduce and explain several key software design concepts: using an analogy between software systems and audio systems. We stress issues involving software reusability, and we find that the combination of such a practical focus, and an analogy based on such a familiar domain, work together well. Our research work involves a conceptual model for better understanding support for reusability, and we also find the audio analogy works well with our model. The paper explains the elements of the analogy in terms of both our model and standard concepts in software design, and shows how the analogy supports reasonable and motivating explanations of these important issues.

Publishing Information

This was originally published as: Robert Biddle and Ewan Tempero, "Explaining Issues in Software Reusability via an Audio System Analogy" In *Proceedings of the 2nd Australasian Conference on Computer Science Education*, ACM SIGCSE, 1997.

Explaining Issues in Software Reusability via an Audio System Analogy

Robert Biddle and Ewan Tempero

Computer Science
Victoria University of Wellington
Wellington, New Zealand

{*Robert.Biddle,Ewan.Tempero*}*@comp.vuw.ac.nz*

Abstract

This paper presents a novel way to introduce and explain several key software design concepts: using an analogy between software systems and audio systems. We stress issues involving software reusability, and we find that the combination of such a practical focus, and an analogy based on such a familiar domain, work together well. Our research work involves a conceptual model for better understanding support for reusability, and we also find the audio analogy works well with our model. The paper explains the elements of the analogy in terms of both our model and standard concepts in software design, and shows how the analogy supports reasonable and motivating explanations of these important issues.

1 Introduction

For several years we have used used music to introduce our course on object-oriented programming. The music provides a memorable beginning to the course, but it is the technology involved in playing the music that we discuss. We begin by showing an audio cassette, and explain where it was manufactured and recorded, and we then show a portable audio system and explain where it was manufactured. The cassette and the player are manufactured by different companies in different countries, and we explain they have never been used together until this day.

We insert the cassette into the player, press a button, and the music begins.

The point of this explanation is to emphasise to the students to tremendous advantage of systems consisting of components with standardised interfaces. The demonstration continues, and involves not only cassettes, but compact discs, headphones, speakers, power supplies, and so on. As we demonstrate, we explain the relevance to software technology and design: there are many aspects to discuss, and many details to illuminate.

In our course, we explain many aspects of object-oriented programming, but what we stress most is its role as a supporting paradigm and technology for creating reusable software components. Our musical introduction involves a domain and supporting paradigm and technology that most students are already familiar with, shows how the components have been carefully designed to work together, and demonstrates the various benefits.

The goals of this paper are to document this pedagogical approach, to explain in detail the analogies we make, and to show how and why we believe the analogies to be reasonable. We believe that reusability is an important part of software engineering, and feel this approach has been useful in improving our teaching of it.

Our research work also concerns various aspects of software reusability, and in particular involves a conceptual model for better understanding how to support reusability [2]. The model has influenced our teaching already, leading especially to novel explanations of encapsulation and of inheritance [1]. However, our audio analogy pre-dates our model, and in this paper we will examine how well the audio analogy supports the model. We therefore show how the analogy fits with our model, as well as fitting with well-known design concepts, which of course are also consistent with our model.

The next section presents an outline of our conceptual model for understanding reusability, and briefly explains how we use the model to explain key principles of software design. Section three describes the aspects of component audio systems that seem most relevant to software design, and shows how these might be explained both in terms of our model for software reusability and in terms of several key standard concepts. In section four we outline the apparent discrepancies between the audio analogy and our model. Finally, we present our conclusions.

2 A Model for Understanding Reusability

In our approach, we explicitly distinguish the concepts of "reuse" and "reusability". While there is clearly a relationship between the two — the proof of the reusability is in the reuse — there is also very definitely a difference. Code reuse is an activity that takes place after code has been initially created, whereas reusable code is the result of activity that takes place when code is created [3].

Code has been reused since the dawn of programming through the use of some form of cut-and-paste operation. While this is reusing code, it is ad hoc in nature. This means it cannot be systematically applied in new situations and so cannot provide many benefits. The benefits we expect from *reusable* code include:

- new applications can be built faster, because code does not have to be reinvented (and rewritten and redebugged),
- applications are cheaper to maintain, because components can be improved and the benefits recouped by all applications that use them, and
- applications are easier to understand, because they are built using well-known components.

To get these benefits, we need the kind of code arrangements depicted in figure 2. On the left of the figure, two segments are related in that at some place or places one uses or *invokes* the other. The effect is as if the second segment of code has somehow been inserted into the first segment. In such situations, we call the invoking code the *context* and the invoked code the *component*. Each different context that invokes a component represents a use or reuse of the component. This structure reflects the programming language mechanisms that have supported reusability for many years: macro definition and expansion, and procedure definition and call. However, it also applies to more recent concepts, such as the definition and instantiation of classes in object-oriented programming.

An important aspect of this structure is that it also works in reverse. The common case is *component reuse* as described on the left of figure 2. Sometimes however, we work the other way around: we have a context, and use it with different components, as shown on the right of the figure. For example, we might have context code that calls a procedure, and in a different setting we may need the same context code, but want a different procedure to be called. We call this *context reuse*.

Context and component are really *roles* played by segments of code. As the figure shows, a segment can play both roles: the component role because it is invoked by other contexts, and the context role because it invokes other components. For example, a procedure may be called from elsewhere, but also

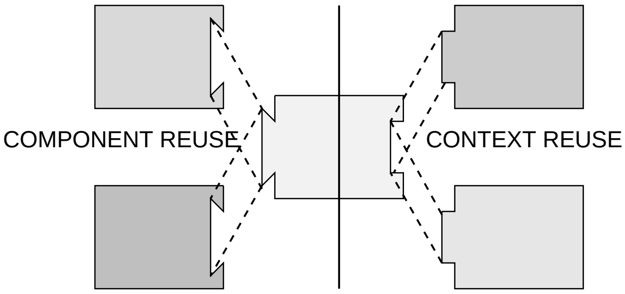

Figure 1: Component Reuse, as shown on the left of this jigsaw diagram involves several different contexts invoking the same component. Context Reuse, as shown on the right of the diagram, involves one context invoking different components using the same interface.

itself call other procedures. This observation, together with the notion of context reuse, shows that any discussion of reusability must encompass both component and context. We believe this is a common failing in previous discussions of this subject.

When a context invokes a component, it has expectations as to what the component does. Similarly, the component has expectations as to how it will be invoked. We call these expectations *dependencies*, and they are the fundamental low-level concept of our model. Dependencies can adversely affect the reusability of a segment of code, because they limit the number of contexts than can invoke it, or the number of segments it can invoke. However, there are dependencies that are beneficial to reusability, by allowing generality and checkability.

generality: The generality of a component is the amount of control a context has over how the context dependencies are met. Similarly for the generality of a context.

For example, parameters of different kinds (values, types, functions) provide different ways for the caller of a function to dictate how the function behaves to meet the caller's requirements.

checkability: The checkability of a dependency is the degree to which the fact that the dependency has been met can be checked.

The requirement that the parameters to a procedure be a certain type is a dependency. Some languages do not check that the types of the actual parameters are correct at all, some check only at run-time, while others check at compile time.

Since some dependencies are beneficial to reusability, it follows that programmers will deliberately introduce such dependencies into the code they write. We therefore find it useful to classify dependencies according to *intention*.

contract: Contract dependencies are those that are deliberately introduced to the segment.

dependency between	context component
beneficial effect on reusability	checkability generality usability
intention of dependency	contract non-contract
language support for dependency	explicit implicit informal

Table 1: Model summary.

non-contract: Non-contract dependencies are those that exist but are not intentional, either by error or because the language provides no way to exclude them.

We are especially interested in how language features affect software reusability, and we have also found it useful to classify dependencies according to the support provided by the software development environment.

explicit: An explicit dependency is one that can be described by constructs in the programming language.

implicit: An implicit dependency is one that cannot be described by constructs in the language, but whether it has been met can be automatically checked by the environment (such as compiler, linker, loader).

informal: An informal dependency is one that cannot be described by constructs in the language, and nor can it be easily checked.

Earlier we defined reusability in terms of the number of contexts or components that could *usefully* invoke other components or contexts. This definition reflects the fact that it doesn't matter how reusable a segment of code is, it is not useful if it doesn't do what is required. Thus, determining whether a segment of code is useful is important. Research in human-computer interaction has explored the concept of "usability" and has resulted in improved understanding of many issues in user interface design [4].

Table 1 summarises our model.

3 Audio Components and Reusability

An audio system is essentially a device that plays recorded music. The fundamental technology that makes this possible at all is still developing, and involves acoustics, electronics, mechanics, and so on. However, we focus only on the organisational aspects of the engineering of common audio systems, as they affect the user. In this way we are comparing a programmer who assembles software units together with a listener who connects audio units together.

Category	Examples
recorded media	cassette, compact disc
media player	tape player, disc player
power source	battery, mains
acoustic output	speakers, headphones
amplifier	

Table 2: Audio system functionality segmentation.

Audio systems could be designed and constructed as monolithic devices. In some circumstances, this may be a sensible design: the requirements for a musical greeting card may be best met with a monolithic design. However, in most other situations this design is unrealistic, because it would force users to make unacceptable commitments. Most importantly, users typically want to listen to different music as they wish, while at the same time wanting high-fidelity audio quality.

It would be possible to package all this up as a monolith: recorded music, high-fidelity reproduction electronics, and so on. But people wanting musical choice would need many such monoliths, and the approach would be too expensive or wasteful. Accordingly, most audio systems are designed to make use of separate recorded music. Similar reasoning leads to audio systems being even further segmented into parts, as described in table 2.

With this array of choices, it is clear that monolithic alternatives seldom make sense. In fact, there are many more choices yet, because there are also many qualitative alternatives. The solution is to construct functional components, and let users assemble the components together to achieve the desired functionality in an audio system.

This design allows users to attain the functionality they desire without paying for functionality they do not desire. Moreover, should they later desire additional functionality, they need only obtain the additional functionality, and reuse the functionality they already have.

In order to allow users to assemble the parts together, the parts must be designed to work together. There are a large number and variety of parts that may be of interest, and the number is potentially unlimited, because new parts are being created all the time. Interconnection can only be made feasible by using a much smaller number of ways to connect parts, and the most typical design involves having a kind of connection for each functionality.

3.1 Component Reusability vs. Context Reusability

The component audio system design allows audio systems to be assembled from parts with standard connections, and allows systems to be changed by introducing new parts while reusing others. In our software reusability model, we stress that reusability

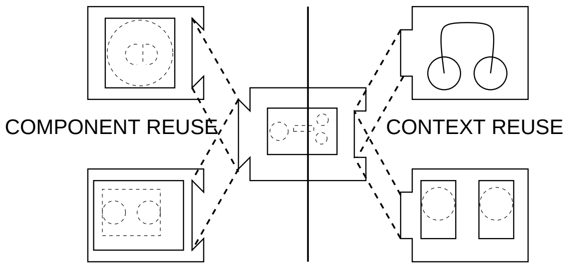

Figure 2: Both Component Reuse and Context Reuse are familiar in the audio system domain. On the right hand side, headphones or speakers can be connected as output from an amplifier; on the left hand side, a disc player and a cassette player can both act as input to an amplifier. In the middle, the amplifier plays both roles.

works two ways: component reusability and context reusability. In the audio model, it is clear that both kinds of reusability are important. For example, it is important that a pair of headphones can be used with many different amplifiers. On the other hand, it is also true an amplifier should be able to work with many different headphones. And it works both ways for several kinds of audio unit (see figure 2).

It seems clear that the symmetry of component and context reusability is important for audio systems, but the utility of the nomenclature is less certain. It might be argued that an amplifier is like a context, and the headphones like a component, because the amplifier is larger, costs more, and provides the audio signal to the headphones. However, the relative costs and sizes are far from certain, and the order may often be reversed (especially if we substitute speakers for headphones). The audio signal seems more significant, but the connection between an amplifier and a power supply does not involve an audio signal, so the issue seems unresolved.

In fact, the same is also really true in software: code components may indeed be larger and more important than code contexts. The important thing is to realise that both kinds of segment are important, but that we often see component reuse and context reuse work in different ways. With typical software components, we often focus on the direction of control flow. Even though audio components are typically oriented around signal flow (see Section 4), we often focus on control, and see a unit with controlling knobs and switches as central, and other units as secondary components. In audio as in software, this can obscure issues in reusability.

3.2 Dependencies, Generality, Checkability

The fundamental level of our software reusability model involves dependencies between segments of code. According to the model, dependencies reduce reusability unless they foster generality, checkability, or usability.

The connections between separate audio components typically involve electro-magnetic changes and shaped physical connectors. Between audio components, such as an amplifier and a pair of headphones, the electro-magnetic dependencies can be seen as similar to implicit dependences between software segments.

These dependencies *do* limit reusability. For example, if we connect audio components with mismatched electro-magnetic requirements we will likely not get the music we want, and may indeed damage the components instead. Eliminating these dependencies does make the components more reusable, and may be a good idea. However, in audio and in software, components that do not have *any* dependencies are not very useful.

For useful generality, some dependencies are indeed necessary. Most importantly, there is a need to pass an audio signal from unit to unit. For example, an amplifier varies an electronic signal, and the headphones vary the acoustic output accordingly. A similar situation applies for the connection between media players and amplifiers. Between media and media player, the dependency is also similar: involving a magnetic connection between cassette and tape player, and an optical connection between compact discs and disc players.

There are also forms of generality that do not directly involve an audio signal itself. For example, audio cassettes have small tabs at the back to pass specifications to cassette players. Some tabs indicate that cassette may be used for recording, and other tabs indicate settings to be used in processing the magnetic signals on the tape.

But the potential problems of dependencies remains, and especially where the danger of a mismatch has significant implications, these problems can inhibit reusability. The standard solution in both software and audio components is very similar, and involves the introduction of *new* dependencies especially to support checkability.

In software, dependencies between segments can be difficult and costly to check: they require either formal verification before execution, or time-consuming run-time checks. Various forms of name-checking and type-checking ease this problem by providing new explicit dependencies that can be easily checked before execution to stand in for the informal dependencies. The names and the types are not essential dependencies, but stand in their place, and can be much more easily checked.

In audio systems, there are similar issues that can be resolved in similar ways. Between components the dependencies are typically electrical, magnetic, or optical. While at some fine level of engineering specification the details of these dependencies are explicit,

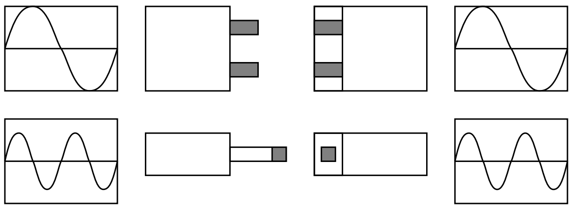

Figure 3: The essential dependencies between audio units typically involve an audio signal represented by an electronic signal. However, these electronic dependencies are usually represented by the physical shape of connectors because they are more easily checked.

it is not usually possible for the user of the components to check them. However, to address these problems, audio equipment typically uses physical connectors involving shape and size that represent the true dependencies, whether electrical, magnetic, or optical. The shapes and sizes of the connectors are not essential dependencies, but stand in their place, and can be much more easily checked (see figure 3). These explicit dependencies constitute the connection interface to the audio unit.

3.3 Encapsulation, Composition and Inheritance

One of the main goals of our model is to help better understand programming language issues that relate to reusability. In particular, we use the model to explain programming language features in terms of their impact on reusability and reuse. Some programming language concepts do not directly relate to audio systems, and so we will not examine details of macros, procedures, or classes. However, some important concepts are more general in their nature, and warrant a closer look. For example, encapsulation, composition, and inheritance have all played significant roles in programming language support for reusability, and appear general enough to consider how they relate to audio systems.

In programming language design, encapsulation means that an object consists of data and procedures that are encapsulated, bound together, such that the data is hidden, and the only access is via the procedures. In terms of our model, this means that no context can depend on the data or code within the object, but only on the procedural interface. This supports *context* reusability, because it means that a context that may be used with one implementation of an object may also be used with another.

The components in an audio system are typically physically separate, and it seems the connections between them are directly related to the way they work together. In terms of our model, the dependencies seem both contractual and explicit. However, it is possible for the electrical fields of components to interfere. Accordingly, most audio units have a quality similar to encapsulation: access from one unit to another is possible only via the explicit interface. Again, this ensures that a component will not work simply in one context, but in any context with the correct explicit interface.

The analogy between software and audio components works very well in explaining the essential idea of encapsulation in another way. For an audio unit that has human interface, we can point out how the unit is encapsulated so that the user may only control it via the explicit interface of knobs and switches. With audio as with software, it *is* often possible to break the encapsulation with enough effort, but with the consequence that the manufacturer can then no longer guarantee the functionality.

As with units of software, audio units are not monolithic within, but are composed of other internal units. The implementation of a software object may involve many internal objects all working together to provide the state and behaviour of the outer object. The implementation of an audio unit may involve many internal units, mechanical and electrical, all working together to create the functionality of the outer unit. And because of encapsulation, in both software and audio such composition should not influence how the unit may be connected with other units. Students old enough to remember reel-to-reel audio tapes should appreciate the usability implications of encapsulating those reels and tape within modern cassettes.

In the world of software, the idea of inheritance has been seen has been much celebrated, and is often regarded as a key feature in support for reusability. There are many different precise interpretations of how inheritance should work, but we believe the correct starting place in explaining inheritance is to focus on the general principles of how it supports reusability.

Firstly we always explain composition: how objects can be built from other objects used internally. Many approaches to inheritance start by allowing a similar facility. However, we regard the distinguishing feature of inheritance to be support for *interface conformance*. That is, the ability to create a new object that conforms to the interface of another object. For example, in a software system with an `employee` object, inheritance allows us to create a `manager` object that conforms to the interface of `employee` (see figure 4).

We explain that the advantage of this is that it allows context code used with the existing object to be re-used with the new object. For example, a mailing-label program written to use the `employee` object can now also be used with the `manager` object. Thus, inheritance provides interface conformance, which supports *context* reuse. (We explain our approach more fully elsewhere [1].)

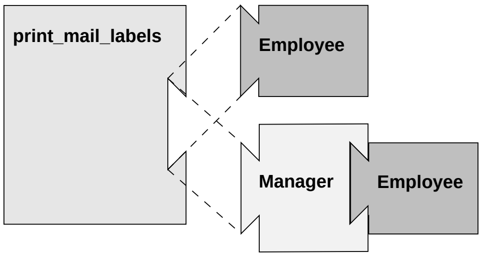

Figure 4: Because the interface of the manager class conforms to the employee class, it can be used in place of the employee class in the context code.

In the world of audio, the idea of interface conformance is very common. For example, many units take an electronically encoded audio signal as their input, with the precise dependencies conformant to a standard, and use the same kind of connector representing this conformance. Accordingly, an audio signal can be passed to many kinds of headphones, speakers, and recording equipment all via the same interface. This is similar to inheritance, especially because what these units do with the signal may be very different.

In software, inheritance allows conformance to an interface, but permits extensions so creating a different and optional interface. (So for example the `manager` may have some functions an `employee` does not have.) In audio this is also possible, and works in a similar way. Many sophisticated units are designed to work usefully according to a standard interface originally created for more primitive units. However, they also have a different and optional interface that concerns their more sophisticated features.

This approach is often typified as "backward compatibility", but the principles are the same as with inheritance. For example, stereo headphones work sensibly with monophonic amplifiers. Sometimes the interface conformance is more strained and more obvious, but is still worthwhile. For example, some portable compact disc players come with a connector that is the shape of a tape cassette, and the disc player emits magnetic signals in the physical location where the tape is usually placed to transit magnetic signals to a tape player. The effect is that compact discs can thus be used as a signal source with amplifiers designed to work only with cassette tapes — such as in car or portable audio systems.

Further on, there is also a parallel between software "wrappers" to create a new interface, and electronic "adaptors" that serve a similar purpose. In all these cases, interface conformance yields the same result: a greater ability to use units in a new situations.

3.4 Usability

A part of our model that we have not yet discussed here is usability. We include this in the model because it is humans who create software units and humans who assemble units together, and there are human factors that affect this. The term usability was coined thinking of the user interface of programs, but it also applies to programmers. When connecting software units together, there are always issues involving human memory and mental dexterity.

When we introduce object-oriented programming using the audio analogy, we actually demonstrate many of the points using actual audio equipment: cassette decks, amplifiers, speakers, etc. As we demonstrate, we must continually pause to connect and control the audio units. Because we hope for some dramatic effect, especially with music involved, our unavoidable fumbling in making connections seemed bothersomely distracting. However, we have come to realise that this too is an important lesson. Assembling software units is *like* that — usability *is* an issue, and it's not always perfect. But the reward is still worthwhile when the music plays — and when the software works.

4 Limits of the Analogy?

As teachers, we are aiming to harness an understood domain in aid of explaining a new one. However, beyond a certain point the analogy becomes stretched too far, problematic, and at this point we abandon it, hoping it was useful in the initial stages.

Some of the ways the analogy breaks concern small details. For example, the ways in which one audio unit customises another are relatively few, whereas this is common in connecting software components. Similarly, the issues of interface conformance become clouded when two units can both conform to each others interface, as is possible in the audio domain. Moreover, interface standardisation is much more widespread in the audio domain, although a look at the shelves of any electronics shop makes it clear this is not a simple issue. And of course, there are important aspects of audio components that we do not dwell on, such as detail issues to do with electrical power.

More significantly, we are introducing the object-oriented paradigm with imperative programming. However, the audio domain is typically concerned with the continuous processing of signals, and so resembles more the data flow paradigm of programming. Accordingly, it is perhaps a poor idea to dwell on how audio units pass information along, because of its poor resemblance to object-oriented practice.

Another important concern involves the fact that connections made between audio components are essentially static. We connect them together, and they stay that way until we disconnect them. This happens with software units as well, and the static connections are important because the code units must be assembled together. However, software components also have dynamic connections that are far from typical in the audio domain. For example, while the interface conformance exhibited in audio systems resembles the static side of inheritance, it is more difficult to find a convincing analogy for the dynamic effects of polymorphism. For example, it might suggest an audio system that selected different speakers on the basis of different kinds of music. Of course, even this is not inconceivable, and so the analogy may still be helpful at its limits.

5 Conclusion

Our principle goals in writing this paper were to document the analogy that we use to introduce our course in object-oriented programming, and to show that the analogy is reasonable. The analogy is between software systems using components, and audio systems using components, and we use the analogy to explain such issues as encapsulation, composition, and inheritance.

However, our particular approach to teaching object-oriented programming is based on our model for support for reusable software. The audio analogy also works with our model, and so enables us to discuss and explain issues such as context and component reuse, dependencies between units, generality and checkability.

As we have disussed above, the analogy is not perfect, and eventually breaks down, and we do not pursue beyond the limits too zealously. Moreover, like any analogy, this is only really useful to the extent that the students already have some experience of audio components. Fortunately this is very widespread. Accordingly, we believe the analogy assists our students more quickly and vividly understand several key issues in software design.

Our style in introducing object-oriented programming is to take a “no-nonsense” approach, and focus on solid engineering advantages involving reuse and reusability. Using an analogy firmly fixed in the real world experience of the students works well with this style, and moreover bolsters the motivational impact.

References

[1] Robert Biddle and Ewan Tempero. Explaining inheritance: A code reusability perspective. In *Proceedings of the Twenty-Seventh ACM SIGCSE Technical Symposium*, February 1996. Also available as Technical Report CS-TR-95/18.

[2] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*. IEEE Computer Society Press, April 1996. Also available as Technical Report CS-TR-95/17.

[3] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[4] Jakob Nielsen. *Usability Engineering*. Academic Press, New York, 1992.