Supervenience as a Design Pattern

JUAN R. REZA, Rezar, LLC.

Single-inheritance object-oriented languages have been extended with features for adding methods to a class from other classes without disturbing the source code of the receiving unit. Well known examples include traits, mix-ins, default methods in interfaces, and method references. The purpose of these offerings is explained in terms of convenient re-use of individual instance methods. These are most applicable when inheritance of a needed method is not possible without altering the application's class hierarchy. The intended objectives include: (a) avoid duplicating a method in another class, (b) modify a class or interface for another version of the system without forcing the first version to change, (c) allow dynamic composition of methods into a "class", (d) accomplish each of these objectives without forcing the creation of a new class with a slightly different name or position in a namespace or hierarchy. These are certainly useful objectives in making practical coding choices. However, point solutions based on methods may lead class designs away from other principles supporting quality and maintainability of the design such as separation of concerns.

The concept of supervenience is an approach to software design for adding *responsibilities* at the classifier level. As such, it supports thinking in terms of the specific concerns of classes and interfaces when considering adding behaviors. The supervenience language feature was created specifically for the combining of responsibilities while not disturbing the classes that host the implemented methods. A range of procedural programming languages offer coding features that support this concept in whole or in part, whether or not they were designed with supervenience in mind.

This paper proposes that the supervenience relationship can be viewed as a Design Pattern. It also introduces a graphical notation for the supervenience relationship. The definition of supervenience as a language feature is reviewed. The concept as realized in several programming languages and frameworks are compared.

Categories and Subject Descriptors: **D.1.5 [Programming Techniques]:** Object-oriented programming; **D.2.10 [Software engineering]**: Design; **D.3.3 [Programming Languages]** : Language Constructs and Features

General Terms: Design Patterns

Additional Key Words and Phrases: Supervenience, Inheritance, Java 8, default method, Objective-C.

ACM Reference Format:

Reza, J. 2013. Supervenience as a Design Pattern. 20th Conference on Pattern Languages of Programs (PLoP), Allerton Park in Monticello, IL (October 2013), 14 pages.

1. INTRODUCTION

Single-inheritance object-oriented languages have been extended with a variety of features that break out of the constraints of a strict class hierarchy. The motivation for certain of these extensions comes from a set of problems that emerge when software systems evolve. Adding new responsibilities to classes for new releases of an application should not be accomplished by changing the existing code base. At the same time, it is desirable to introduce new implementations that do improve the old generation of the system, or keep it compatible with new infrastructure. Even within the same development effort, aside from revisions, we wish to combine behaviors from different classes without duplicating or splitting code and without rejiggering a class's place in a hierarchy. Supervenience has been defined as a feature to support these intentions.

The language feature that was created specifically to realize supervenience in a single-inheritance language is described briefly here. A supervenience relationship is the relationship between a class that declares itself to be considered the "super" and one or more of its supervened-on classes. The supervened-on classes' place in the hierarchy is not changed. Simply stated, this feature introduces the keyword super into the syntax for class (or interface) declaration. The super keyword tells the compiler to generate a class that will act *as though* it is the superclass of one or more supervened-on "subclasses". The full semantics and syntactic rules have been studied in previous work to ensure consistency, type-safety, and intuitively useful disambiguation. These issues are not repeated here.

This research is in partial fulfillment of the recertification requirements of the author's CSDP credential.
Author's address: J. R. Reza, 2001 S Magnolia Dr., Tallahassee, FL 32301; email: Juan.Reza@ieee.org
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 20th Conference on Pattern Languages of Programs (PLoP). PLoP'13, October 23-26, Monticello, Illinois, USA. Copyright 2013 Is held by the author(s). HILLSIDE 978-1-941652-00-8

1.1 The Concept of Supervenience in single-inheritance

We will start with a recap of the supervenience language feature. The word supervenience means a "coming over upon" another in some way that leaves the original unchanged in itself.

The language feature is introduced here with an example that has intuitive appeal. Using Java as the basis for studying supervenience, the following class declaration includes the keyword "super" in the role of signifying the application of "supervenience". The declaration of a supervening class in "java supervenience" has a super clause which appears before the extends clause and body, as shown in statement (1).

```
class SuperClaz  super  ProtoClaz, RelatedApp  { <<member-selections>> ... }      (1)
```

Here, the `super` keyword tells the compiler to compile the class SuperClaz such that it will *act as though* it is the superclass of both *supervened-on* "subclasses" ProtoClaz and RelatedApp. These "subclasses" are unaltered. This supervening class declares a list of methods defined in one or more supervened-on classes, the *member-selections,* which will be visible to code using SuperClaz. Only methods declared in the *member-selections* will be visible to using code through SuperClaz. In the using code the SuperClaz class can act as the type or superclass of either of the supervened-on classes. A variable of type SuperClaz, in this example, can hold an instance of only one of the "subclasses" at any one time. It can be instantiated as shown in (2) which instantiates ProtoClaz and then RelatedApp.

```
SuperClaz  app = new ProtoClaz(); app = new RelatedApp();                (2)
```

Upon execution of (2), app refers to an instance of SuperClaz which holds an internal (hidden) reference to RelatedApp. The ProtoClaz instance is then available for garbage collection.

In earlier work, potential ambiguities and special cases have been explored and resolved. The definition of supervenience as an extension of Java also considers the Java `interface` with the `super` keyword. The "what if" questions and their resolution are not repeated in this paper. The various implications of this artificial super construct are not pertinent to the present consideration of supervenience as a design pattern.

This paper considers supervenience when it is viewed as the core idea of a Design Pattern. Like other design patterns, the focus is on the class and architectural relationship among classes, rather than method level coding tactics and work-arounds. We look at several object-oriented languages that have features supporting some or all of the coding possibilities implied by the supervenience construct. .

1.2 Archetypal pattern arising from the supervenience relationship

The pattern is illustrated in Figure 1 using an example based on the now-popular notion that quantum particles have a wave and point-like nature: the wave-particle duality. We can imagine a simulation application that models objects as waves and particles. Other real-world subjects also have dual or multiple views comparable to this example.

The relationship between a supervening class and one or more of its "subclasses" calls for a graphical notation that is not present in standard UML class diagrams. In the interest of keeping with the tradition of GoF design patterns (Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.), the new graphical notation is introduced here. In Figure 1, the red double-ended arrow represents the "super" keyword. It is a combination of the UML class notation for inheritance and the notation for reference. The arrow head pointing to the "subclasses" this represents how the compiler must implement the relationship: by hidden composition.

The class `WaveLike` defines method `getVelocity()`, `getFrequency()`, `getOther()` , `getName()`. `ParticleLike` defines method `getVelocity()`, `getPosition()`, and `getMass()`, and `getName(`.

`Electromagnetic` introduces and defines method `getPathProbability()`. `Electromagnetic` also includes the member-selections: all of the above (`getVelocity`, `getFrequence`, `getPosition`, `getMass`). Notice that member-selections does not include `getOther()` and `getName()`. These will not be visible to through reference variables declared as `Electromagnetic`. Finally note that the supervening class can reconcile different signatures of methods with a wrapper method if appropriate.

The example suggests that some behaviors are common to a wave and particle while others are defined in one or the other. The superclass combines these behaviors, supplying a definition for any member-selection that is not supported by one of the supervened-on classes.

As a beneficial consequence, the experimental apparatus, or simulation, can exercise behaviors of the object while modeling the object as a single thing.

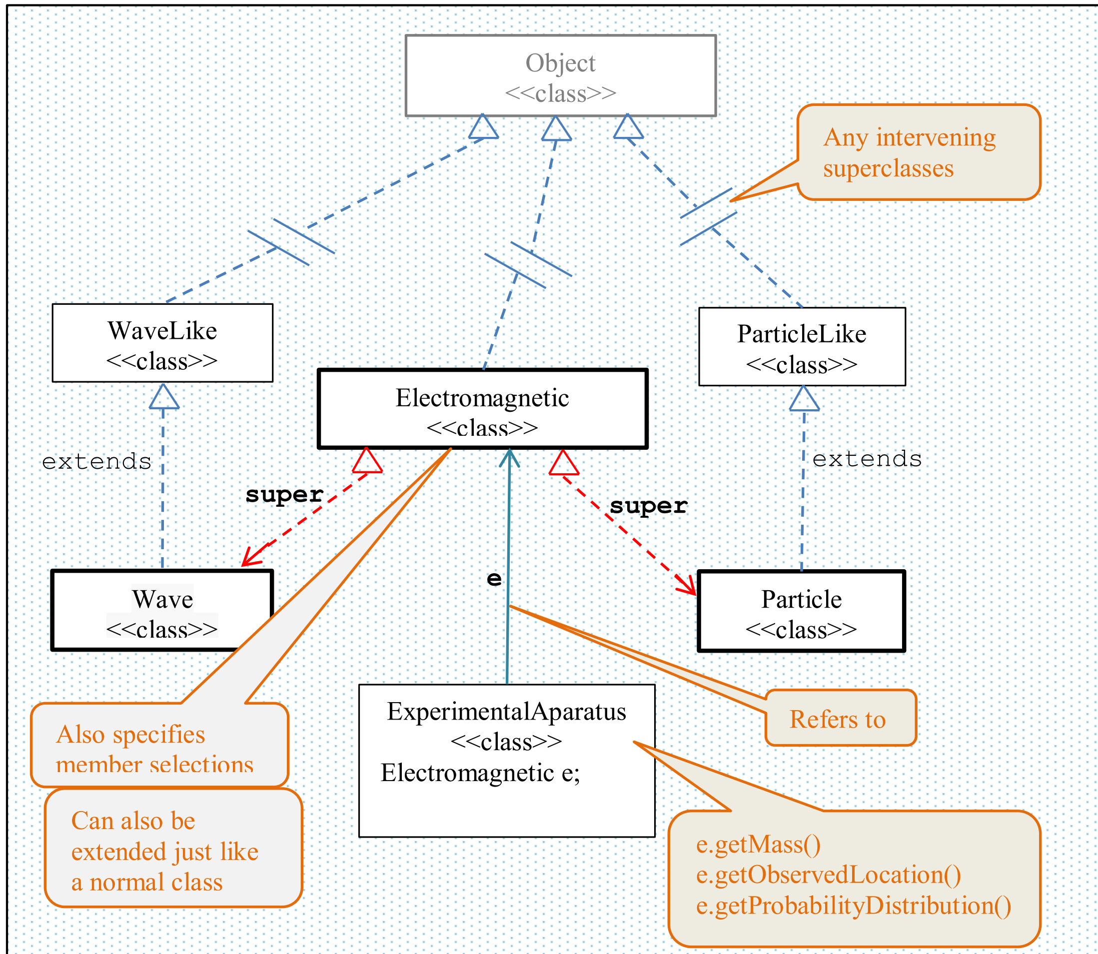

Fig. 1. Example class-like diagram of the basic supervenience design pattern.

When the `Electromagnetic` class is defined as shown, the `Wave` class does not need to be modified, if it existed before. `Wave` does not require re-compilation to support the relationship of `Electromagnetic` class to it. During execution, the `ExperimentalAparatus` may instantiate `Electromagnetic` is in several ways. It can be assigned a subclass instance, if it has no other state of its own, as in (3).

```
Electromagnetic electron = new Particle();                    (3)
```

Since the member-selections are effectively abstract, we cannot instantiate Electromagnetic alone (as in this invalid statement: Electromagnetic em = new Electromagnetic();). Either a default or passing an instance of a subclass is valid, as in (4).

```
Electromagnetic electron = new Electromagnetic(particle);     (4)
```

1.3 Motivation for mechanisms supporting supervenience

Several commonly known cases demonstrate the call for language features that help to view a hierarchy of classes differently in different contexts, without disturbing the basis design.

a) Evolution: Simulation of a car, where a new kind of car (racing or flying) is desired. This is a typical artificial example of an evolving application that motivates us to call for supporting language features.

b) Non-hierarchical nature: Geometric figures library, where a `Polygon` has subclasses `Circle` and `Rectangle`, Liskov (Liskov) pointed out the design decision we face in implementing Square which could drive us to modify Rectangle in ways that cause ambiguity about the length of a side.

c) Common concerns, but incompatible implementations: `String` and `StringBuilder`, where we may desire to use a version of replace() from `String` in `StringBuilder`, as well as other methods from either. We may simply want an interface to hold either String or StringBuilder, exposing just the common methods hosted by these two core Java classes. We may want to simply "add" a method to `String` so that other existing code is not disturbed by the local change. Existing common superclasses such as `CharSequence` don't always anticipate our design requirements.

d) Anticipated growth: We anticipate that new fields will be added to an underlying database where it is then desired to enhance the corresponding Entity bean without disturbing the pre-existing system that uses the bean and without having to complicate the algorithms using the bean in the new system.

e) Duality (*multiality*): We want to model the artifacts and relationships in some real-world phenomena where the nature of the objects depends on the context in which they are operating (see Figure 1).

f) Independence (non-coupling): The addition of behaviors must not force changes to the referenced class or its methods and should not require special code in anticipation of being supervened-on.

g) Type-safety: The language feature intended to support all of the above intentions should not introduce type-safety holes.

h) Limited Access: The language feature should provide a way to decide which methods will be exposed, and it should follow the narrow-to-wider pattern supported by the natural hierarchy.

Counter-intentions:

i) We want an arbitrary set of methods to be combined and added to a base class at runtime. This is an example of a technique that supervenience does not support. Supervenience seeks to maintain compile-time type-safety while composing classes.

j) We want to define several *aspects* of a class so that different code segments may be selected for invocation at runtime according to special code for this purpose. Supervenience deals at the level of classes and interfaces, and supports selection of methods. It does not deal with code sections within methods.

1.4 Distinguishing supervenience from multiple inheritance of classes

The supervening class appears as a superclass, hierarchically above the classes it declares as its "subclasses". These "subclasses" retain their defined position in the hierarchy. It is only the supervening class that creates a view of them in which it is their super type. An instance of the supervening type holds only one instance of one of the "subclasses" at any one time. In contrast, multiple inheritance has the inheriting class below its "superclasses" in the hierarchy. All initializers of the superclasses are executed upon instantiation of the inheriting class.

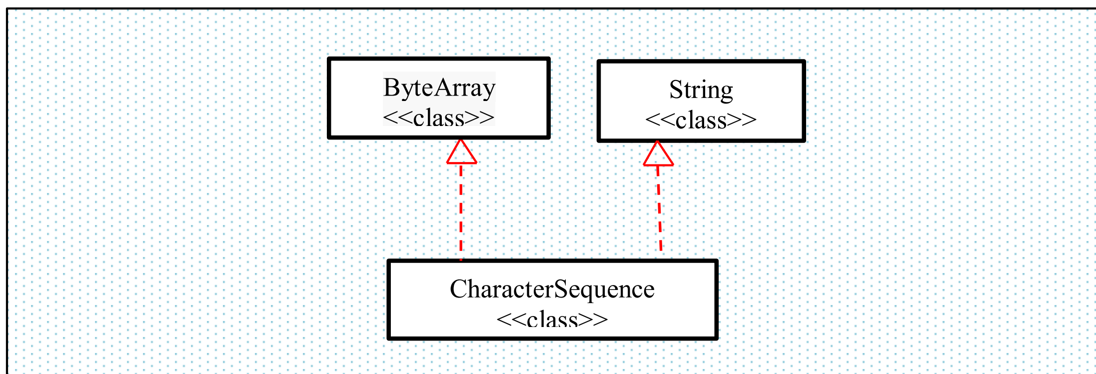

Fig. 2. Multiple inheritance pattern in a C++ -like language.

During execution, the `CharacterSequence` class runs the initializers of both of the super classes in the example, Figure 2.

This distinction is not immediately apparent. On first glance at supervenience, readers have tended to pronounce this construct as the same as multiple inheritance. As a design pattern, supervenience should not be confused with patterns based on multiple inheritance of classes.

1.5 Motivation of supervenience as design pattern concluded

We have presented a series of examples motivating the call for a feature like supervenience. We have given a recap of the meaning of supervenience and how it may be implemented as an explicit language feature. The next section covers features of several programming languages and frameworks that support some or all of the intended capabilities of supervenience.

2. IMPLEMENTATIONS AND SUPPORTING FEATURES

Each of the languages described in this section implement some form of supervenience. Diagrams and code samples for each language are included where needed to illustrate key points.

These approaches to adding or changing functionality have been studied and compared before. This presentation asks the reader to see how they are manifestations of supervenience. As such, the underlying pattern should take shape in the mind. The diagrams and code samples here, with all their imperfections, attempt to show that the different techniques can be thought of as different forms of a single concept for the structure-preserving, superimposing design called supervenience.

2.1 Objective-C

The construct known as a Category (Apple, Inc. [Customizing]) is an example of a feature that can support a pattern exhibiting supervenience. Objective-C also has an anonymous category feature called class extension (Apple, Inc). A “final” class can be modified by creating a compilation unit that masquerades as the original. It references the original class, takes its name, and then the coder is free to declare and implement new methods. These supervenient methods can be referenced in another class and then appear to the calling code as if they were part of the original class. The compiler and dispatcher “know” how to resolve the references.

The sample listing shows how portions of the example of Electromagnetic in Figure 1 might be implemented. The WV prefix represent the WaveLike framework or namespace (prefix for “Wave”). The PT prefix represents the ParticleLike framework. The EA prefix signifies the experimental apparatus. The ProbRef is assumed.

Listing 1. Objective-C sketch referring to the wave-particle duality as an example of supervenience

```
    // in File: WVWave.h
1.  @interface WVWave
2.  -(id) getProbabilityDistribution: ProbRef *prob;
3.  -(id) getFrequency;
4.  @end
    // in File: WVWave.m
5.  @implementation WVWave
6.  #import MyWave.h
7.  -(id) getProbabilityDistribution: ProbRef *prob { . . . }
8.  -(id) getFrequency { . . . }
9.  @end
    // in File: WVWave+MyWave.h
10. #import “WVWave.h”
11. @interface WVWave (MyWave)
12. -(id)getMass: (ProbRef) x;
13. @end
    // in File: WVWave+MyWave.m
```

```
14.   @implementation WVWave
15.   #import WVWave+MyWave.h
16.   WVWave (MyWave) {
17.          // MyWave is the Category name of this compilation unit
18.       -(id)getMass: (ProbRef) x {
19.           // may use the data defined in the original class from Particle
20.           . . . self.data . . .
end
```

Ultimately, the file **WVWave+MyWave**.m in Listing 1 is the compilation unit for MyWave which effectively creates a class that has supervened on the original WVWave with a method from the related PTParticle class. In the same vein we could bring a method from Particle into WVWave. MyWave is implicitly declared at line 11 as a renaming of the original WVWave. It is used at line 18 with the effect of adding method getMass to WVWave. The ExperimentalAparatus can now use WVWave and invoke getMass through it. This code is not meant to compile but only to illustrate how the pattern may take shape.

There are two ways to add a property. First, you can add a property to an existing class by *class extension* (Apple, Inc. [Customizing]). The more dynamic way is by Associate Reference (Apple, Inc. (Runtime)).

The "protocol" feature of Objective-C could have been used in the example to show another way it supports intention #**Error! Reference source not found.**. Briefly the key statements might look like the code segments in (5).

```
@protocol  <PTParticle> APonlyWhatsNeededInterface;                    (5)
id < PTParticle > myParticle = [myParticle init];
```

Objective-C provides a concise mechanism for introducing a property into a class without changing the original source: Associative References (Begemann). This feature deserves further discussion, separate from this paper, with respect to how it relates to supervenience as a design pattern. This feature supports intentions #**Error! Reference source not found.** and #**Error! Reference source not found.**.

Objective-C assigning to the self variable (Apple, Inc.) is illustrated by code segment in (6). This use is during initialization in support of class extension. As was mentioned previously, the supervening class may (usually does) have state of its own in addition to its hidden reference to the subclass instance it carries. This is a pattern that Objective-C supports with self.

```
- (id)initSomething:(NSString *)identifier {                          (6)
  self = [super init];
  if ( self == nil ) { . . .
```

Objective-C type consistency resolved.

The capability to insert a class into a hierarchy creates a dilemma for the type system of the language. In Java, what should be the meaning of keyword `instanceof` with a class superimposed into a given inheritance hierarchy? In Objective-C the same question arises as to the semantics of these intrinsic methods: `isKindOfClass`: and `isMemberOfClass` (Apple, Inc. (Runtime) 8. NSObject Methods).

In such cases, *these should continue to refer to the original (host class) of the unmodified hierarchy*. Separate language constructs should be available to answer the same question with respect to the supervening class, as is possible in both Objective-C and Java supervenience.

This design decision should prevail in the use of supervenience as a design pattern.

The concept of "class cluster" (Apple, Inc. [Clusters]) is also achieved in Java supervenience as a natural implication of the rules around "super" in the class declaration. In both language specifications: (1) Several sub-classes can be specified as valid potential object types in a supervening class. (2) There can be only one of the possible objects referenced through a variable of the type of the parent class at any one time. (3)

The choice of object type that can be assigned to a variable of the parent-type is a runtime action, but the set of possible sub-class types is fixed at compile time.

2.2 Java

The type system of the Java programming language, as of release 7, is strictly hierarchical and single (class) inheritance. As such it has no syntax for supervenience. It does not provide a syntax for adding a method to a class without changing the compilation unit itself or through inheriting a modified super class. Apart from that, the bytecode can be edited to inject methods or change the body of existing methods using tools such as BECL (Apache). The reflection system provides limited means of manipulating methods as Method objects. However, reflection is viewed as circumventing the defined language and will not be considered as a form of supervenience.

2.3 Java 8 with the default method feature

Java 8 introduced the "default method" into the interface classifier (Oracle, Inc.). The stated objective is to allow a new version of the system to add methods to an existing interface E which are defined in another (new) class C without causing the existing code that references E to be forced to change.

This objective can be viewed as a solution to an underlying requirement that could be stated another way:

Allow a class in the system to have behaviors declared in an interface S which are not defined in another a class C that also implements the same interface, and do so without causing the class C to implement the behaviors.

This later statement, admittedly convoluted, reads on the Java 8 statement of objective. It elevates the objective into terms that look much like the pattern of supervenience.

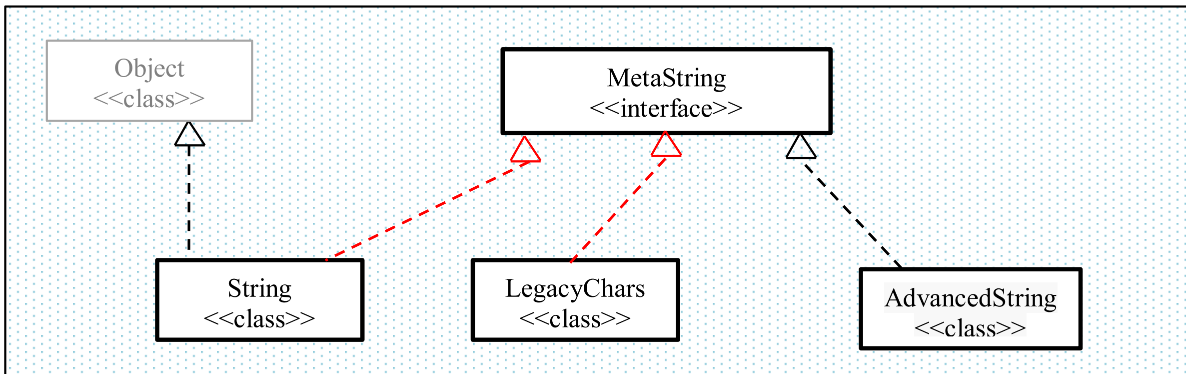

Fig. 3. Hypothetical example of the Java 8 default method feature.

The default method feature supports some of the intentions of supervenience, but not some important uses.

Suppose that we have a library class, `String`, that is often referenced through an interface in the same library, `MetaString`. The desire to add a method to `MetaString` could arise when a new system introduces a class with a very similar set of concerns, `AdvancedString`. It is desired to also reference `AdvancedString` through the interface `MetaString`. In addition to `String`, there is a class, `LegacyChars`, that is in production in the previous version of the application. The problem comes when adding the new method would force changes to `LegacyChars` and `String`. The `String` class might be the responsibility of the maintainer of the library, but changing it could force recompilation of applications using it. The `LegacyChars` class would disturb the legacy application by forcing implementation of the new method, at least as a do-nothing method.

The solution offered by the default method feature solves this narrow problem. A default method is coded in the interface, `MetaString`, thus relieving `LegacyChars` and `String` from having to implement it. However, these still require re-compilation. Also, this approach only works if `String` and `LegacyChars` happen to have declared `MetaString` as (one of) their interfaces. That requires the foresight that a common interface for their common concerns would be useful. Finally, as subsequent versions of the system use `AdvancedString`, the need to carry the default method in their interface may accumulate additional default methods, while wanting to deprecate some no-longer needed. Gradually, this interface could look more like a class than interface.

This solution is distinct from that of supervenience in several ways.

The classes that are supervened-on do not declare the supervening superclass. They are undisturbed and designers do not need to foresee the need to declare a common interface among various classes with similar concerns.

A supervening class declares a set of member-selections which are hosted in (defined in) the "subclasses". If one or more of these methods is not defined in one of the supervened-on classes the supervening class must defined one. This looks very much like the "default method" concept. Unlike the default method, however, the method provided by the supervening class can access and modify state in the host (supervening) class where it is defined. It can also be defined in terms of (reference in its body) any of the methods visible in the supervening class (default methods only access methods declared in the same interface or its superinterfaces). Finally, unlike default method, there cannot be a name conflict between the method defined in the supervening class and any method found among the interfaces of the subclasses since the very purpose of member-selection is to identify methods with the same signature of which only one is present at any one time during execution.

2.4. Java Supervenience

The language extension called Java Supervenience defines a feature set that specifies syntax and semantics built on core Java that supports the concept of supervenience.

The foundation of this feature was mentioned in the introduction. It defines the use of keyword super so that a class can declare itself to act as the (artificial) superclass of one or more other classes which it then has an artificial (hidden) reference to at runtime, under constraint of the method-selections.

The use of "super" in the class declaration is defined in syntax rule js2 (Reza), syntax rule js2, Fig 4.

There are special cases where a qualifier can be used with the "this" variable to select the intended method.

```
Electromagnetic.this.method(...)                    (7)
```

is made possible by the syntax rule "SubalternExpressionName", js4, Fig.6. It permits the statement:

```
Wave.this.getMass( super.getMass());                (8)
```

The syntax for assigning to the **this** variable directly is defined in section 3.3, p84.

```
ParticleLike.this = new Particle()   ...            (9)
```

Note that all of these qualifiers operate at the instance level. The qualifier is not the same as the "::" context resolution operator from C++ and now introduced in Java 8 for static method referencing.

Java supervenience accomplishes much of the same capabilities as Objective-C for the purpose of enhancing a class in-pace within its hierarchy. The syntax rules of the two languages have virtually no similarity. Objective-C has some syntax that is simpler than the corresponding Java supervenience techniques. It supports the list of intentions a bit better as well. In all, the fact that these very different languages can accomplish the stated intentions with similar coding concepts supports the view that they support a common underlying pattern.

2.5 AspectJ

An extension of Java that provides a means of supervening on a class is the AspectJ extension (Eclipse Foundation). AspectJ defines a language-level extension of Java.

It specifically allows a program to declare "super type" of a class without directly changing host class ("host" meaning the class where the methods reside). It allows methods in an otherwise completed class to be inserted, and in both cases it respects the Java type system constraints when these supervening objects are accessed from another class. More importantly, The approach of AspectJ encourages the designer to keep the principles of separation of concerns, single concern, Liskov substitution, and other object-oriented principles. AspectJ does impose some overhead, restrictions, and less-elegant code.

AspectJ programs must abide by a "clear crosscutting structure." This creates a coupling from the supervened-upon design object to the aspect that supervenes on it. Thus a two-way coupling is created.

The fact that the modifiable code must be explicitly designed for modification of certain kinds fails to meet the important intention of not having to change the supervened-on class. The capability to retroactively modify behavior without modifying the source code is limited.

Finally, AspectJ constrains the Java programming language such that an Aspect cannot be instantiated with a standard constructor invocation by new. The static cross-cutting functionality yields singletons. These limitations most likely exclude AspectJ from consideration as a language or extension that supports supervenience as conceived here.

AspectJ shares much of the stated intentions of supervenience but also seeks to provide other capabilities. It was not designed with the underlying pattern in mind. As such it does not support the present concept of supervenience very well.

2.6 Scala

Much the same can be said for the Scala programming language as AspectJ.

Scala provides mixins (admin). These are methods that are treated as objects. This approach accomplishes supervenience to some extent but at the individual method level. It is clean and simple. There is a difficulty, however, in that the methods are effectively restricted to being static and they only have access to the class properties rather than the object instances. We are only interested in the OO features of Scala, not its blend of OO and functional programming. See Scala specification (Odersky 3.2.7 Compound Types).

When you want to insert a behavior from one class into another, there is always the issue of how will the method have access to the members of the state in the host class. There is no problem if the method only depends on the arguments. There is no problem if the method is defined in the supervening class to see its “self” members through an interface which is also supported by the host class. But generally that is unlikely and it is unlikely that the host class happens to have the members named the same.

Scala allows the mixin to create a risk of runtime failure. Supervenience should not create holes in type safety.

Supervenience strictly requires that the original design is not changed to support the supervening behavior or class. More strictly speaking the class being supervened-upon does not contain elements that are necessary for the supervening design to define behavior.

The failure to comply with these restrictions makes Scala’s “supervenience” language construct allow risks to the functionality of the application code.

Nevertheless, it is not the job of this design pattern to pass judgment on specific languages. The Scala programming language supports a form of supervenience. Note that the Scala code is very concise.

The intention of supervenience is to never impose something on either the supervening or supervened-on classes. The complications shown in the diagram suggest that Scala does not support that intention as completely as we would expect.

The trait and mixin strategy of Scala technically provides an alternative to composition, not supervenience.

2.7 Context-Oriented Programming

Supervenience can be compared to the language feature known as Context-Oriented Programming (COP) (Hirschfeld, Costanza and Nierstrasz). While COP is not a language in its own right it addresses concerns similar to supervenience. COP is described as a language extension for adding behavior variations depending on runtime conditions. It is intended to activate available behavior according to information in various application contexts. The mechanism for accomplishing this involves layers and scopes of behaviors that may be activated.

Supervenience is distinct from the COP approach. COP does not provide a mechanism to interpose a class or interface over two or more classes. COP is not designed to change effective behavior without changing class's actual hierarchical parent. An application using COP must be coded with the context-sensitivity explicitly part of its structure and algorithms.

In a sense, COP provides *syntactic sugar* that eliminates explicit method calls everywhere that a code segment might be turned on or off after testing context conditions. So Context-Oriented and supervenience are not in competition. One could imagine that the "behavior variants" of COP could be implemented by supervenience.

We can image ways in which the COP mechanism itself could be implemented by using supervenience in an application that was not designed in a language supporting COP.

2.8 Perl, JavaScript, and interpretive languages in general

Interpretive languages that allow runtime declaration and implementation of methods might be seen as achieving support for supervenience in the sense that any members of classes can be defined and changed at runtime. Since the dynamic modification feature is intrinsic to these languages it cannot be blamed for creating runtime type-safety holes. It is an open question whether these languages truly support the intentions of the single-inheritance object-oriented paradigm.

2.9 C-Sharp (Microsoft dot NET C#)

The C# language provides a unique mechanism for adding methods to a class, other than subclassing. The “partial class” construct (Microsoft) apparently compensates for the lack of an inner class construct. The original source of the class is unmodified, in accordance with the concept of supervenience. However, the source code of the compilation unit as seen by the compiler actually is modified by the intervening code change, just not in the same source file. As such, it is unclear whether C# should be considered can implement the supervenience pattern, at least explicitly.

Prior to this paper, there was an attempt to port the Java supervenience syntax into C#. This effort failed because C# has features that would work at cross-purposes with the Java-based construct.

The partial class construct might seem to be similar to the Objective-C capability described in section 0. Difference include:

- C# partial classes must be marked differently from a normal class in order for the separate parts to be compiled as parts of a single class. This partial class (Microsoft) construct prevents c# from supporting intent #8.
- In C#, all methods defined in any part are accessible to all the other parts. This feature alone would be sufficient to show that C# does not support supervenience as described.

The intended purpose of C# partial classes is largely to compensate for the inability of Visual Studio to support round-trip engineering (informatique) of generated code. The IDE generates code, the developer manually edits that code, the IDE integrates the manually generated code seamlessly, and the same process can be repeated.

3. CONSEQUENCES

The quality of software design may be improved by applying the design pattern stemming from supervenience.

Adherence to a syntactically-governed means of combining the methods helps to avoid runtime type exceptions.

Supervenience has been formulated as a single design pattern employing a variety of coding techniques. This provides us with a way of understanding those techniques as participants in a higher level of design abstraction. Doing so can elevate our thought processes for problem solving and design. We can think in terms of that coherent abstraction rather than ad-hoc coding tricks or work-arounds based on individual methods alone. This thought process can broaden our insights even if the language we use does not explicitly provide features supporting supervenience.

Designs that deviate from hierarchy, in response to the necessities of the problem domain, can be understood to be using a solid concept.

Designers of languages, systems and other structures can explore the more-general construct, and make it explicit rather than including what would otherwise be ad-hoc work-around features addressing only part of the list of intentions.

4. APPLICABILITY

Use the Supervenience design pattern when . . .

- The problem domain exhibits multiple "world views".
- We expect an application or system to evolve in a way that more than one generation of the design must be maintained.
- The application must run in different environments where library functions are similar enough to design classes that should operate the same on all platforms, but with relatively small differences that could be accommodated with small changes.
- We are designing with strong attention to separation of concerns and prefer not to introduce (what some may call) kludge code.

5. RELATED PATTERNS

The Template Method supports the design of algorithms that can operate on a variety of data types. Generics may be used as well as simply passing objects via type interface. One of the limitations of Template Method is experienced when two or more parameters must correspond in some way that the method signature has no way to enforce. This is a candidate case where parameters declared with supervenient class types may provide the correspondence that is otherwise not explicitly represented within the algorithm.

This use of supervenience is mentioned here to invite further exploration and challenge. For now, consider a simple example.

```
<S> S compute( S a, S b) { ...}                    (10)
```

Assume the application has supervenient class `Stringy` and another supervenient class `Numbery`. Each declares a range of classes such as `StringBuilder` and `Integer`, respectively. Both sets of classes have their own concept of order and other characteristics that may be pertinent to the template algorithm.

- The template method in TODO (10) accepts anything for the two parameters, provided they have a common type. The objects passed to a and b can be subclasses of Stringy, or they can be subclasses of Numbery. The caller is constrained to passing mutually compatible objects. This only approximates compile-time enforcement of compatibility of a and b because the burden of using supervenience is on the caller. A more ideal solution would have a language feature at the level of the method signature which enforces the intended co-variance of the parameters.

6. DISCUSSION

The simplicity of an object-oriented hierarchy is appealing as a way of organizing design in correspondence with the perceived nesting of objects in nature and man-made things. Yet it gets in the way when there is more than one view of the problem domain. It can impede implementation of simple fixes to a design that involve breaking out of the hierarchy. The temptation to just "give up" and let programmers throw around methods and mix and match pieces of objects has made applications hard to maintain and iterate. This unconstrained approach makes it seem unnecessary to develop a theory as robust as hierarchical object-oriented design that explains how to deviate from it coherently.

Perhaps the present treatment of supervenience as a design pattern is a step toward that theory.

The several languages known to support a form of supervenience, as defined here, have been compared. An informal review of the coding technique seems to suggest that Objective-C and an extension of Java strongly support the intentions of supervenience.

7. FUTURE WORK

Readers have asked for a reference implementation of Java supervenience in order to experiment with it and develop some more evidence for study. The lambda expression feature of Java 8 should also be examined as to its possible support for supervenience.

The concept of supervenience as a design pattern should be re-described more formally. The result of a unifying theory of supervenience should provide a neutral and uniform basis for predicting and measuring the effectiveness of a given approach.

Supervenience as a design pattern requires a second level of depiction. The traditional structure diagram, or other UML diagrams used in design patterns literature, depicts a design at a single point in time where the design would be completed. In that traditional approach, if the application design is changed a different diagram would take the place of the old. Design patterns have not ordinarily spoken to the change-of-design itself as a pattern. The diagrams in this paper attempt that. This technique follows a similar design pattern in a previous paper that is temporally enhanced to address the change of design concept. Since the concept of design pattern with concern across points in time is now no longer a "one off" unique and ad-hoc case, this paper designates the technique as a "diachronic design pattern." The term diachronic (Doric Loon, et al.) is borrowed from the concept in linguistics in which point of interest in examining a language across two points in its evolution is the *changing* of the language; what, how, and why it changes.

8. CONTRIBUTION

The concept of supervenience is formulated as a design pattern applicable to a variety of languages. The particular way of that the concept is applied to each of several languages is discussed in some detail. Each application approach reveals a different subset of the advantages, restrictions, and consequences that a full realization of supervenience would provide.

Supervenience is discussed informally to be a possible "theory" within which each of the approaches to ex-hierarchy functionality change can be seen as a special case. This idea is an example of what E. O. Wilson calls consilience (Wilson).

A notation, double-headed arrow with inheritance triangle has been introduce to represent the supervenience relationship in a UML-like class diagram.

APPENDIX

The technique of supervenience in Objective-C deserves a closer look at the code level.

Suppose we have two list-like classes and we are designing a new class with a method that takes an array and performs traversals of it in various orders by choosing different index values.

The new class, Z, wants to declare the method's parameter strictly in order to minimize testability problems; so declaration as (id) is out of the question.

There are two kinds of array that might be passed to the algorithm:

```
// in one class A ...
NSArray liveSprites = [NSArray arrayWithContentsOfURL:(id) aURL]
// in another class, B, that uses class A
NSMutableArray liveSprites = [NSMutableArray arrayWithContentsOfURL:(id) aURL]
```

So, the new class Z requires that the array supports typical protocols such as NSCopy.

Alternative solutions attempting single-inheritance and strong typing: (these don't work)

Problem 1. The algorithm could use either an NSArray or NSMutableArray but there is no common parent to them.

Problem 2. We cannot subclass both of these because Objective-C is single-inheritance.

Problem 3. We prefer not to create a new class solely to wrap these two in a (id)array, or declare the parameter as (id) because then we have to cast to one anyway.

So, these approaches only present problems.

Alternative solution by changing the existing classes indirectly with a *supervenient* parent classifier that acts as though both classes extend it.

Solution. Create a common parent classifier and treat both classes as though they inherit from it. Implement as follows.

This solution satisfies the intent. It defies the usual constraint of single inheritance class hierarchy. And it does so without changing the classes directly. Now look at the implementation details.

Define an Objective-C Protocol with the desired methods and restrict the non-common protocols of NSArray and NSMutableArray.

```
@protocol LiveSprite : NSCopy  // declare a pseudo parent type
(id<LiveSprite>) getCell: NSInteger i;
@end
```

For class Z, define an interface in a file named NSArray…

```
@interface NSArray (LiveSpritesCategory)
-  (id<LiveSprite>) getCell: NSInteger i;
@end
```

And do the same for NSMutableArray.

These interfaces (actually "*an* interface" in two compilation units) are *supervening* on the NSArray and NSMutableArray classes which *host*[1] the functionality of protocol NSCopy. They also declare a method that a class which contributes a method must implement. In this case, it is the supervening class that also is contributing the new behavior.

```
@implementation XYZPerson (XYZPersonNameDisplayAdditions)
(id<LiveSprite>) getCell: NSInteger I { . . . }
…
result = [ methodInZ: (id<LiveSpritesCategory>) arrayOfSprites];
```

So, the complex algorithm implemented by class Z can define a method with a parameter allowing either (and only either) `NSArray` or `NSMutableArray`. This accomplishes the intent.

REFERENCES

admin. "A Tour of Scala." 06 March 2009. *The Scala Programming Language.* École Polytechnique Fédérale de Lausanne (EPFL). <http://www.scala-lang.org/node/117>.

Apache. "Byte Code Engineering Library." 17 Oct 2011. *Apache Commons.* 02 June 2013. <http://commons .apache.org/proper/commons-bcel/index.html>.

Apple, Inc. *Programming With Objective-C*. 2012. <http://developer.apple.com/library/ios/documentation/cocoa /conceptual/ProgrammingWithObjectiveC/ProgrammingWithObjectiveC.pdf>.

Apple, Inc. (Runtime). *Objective-C Runtime Programming Guide*. 2012. <http://developer.apple.com/library/ios /#documentation/cocoa/conceptual/ObjCRuntimeGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40 008048>.

Apple, Inc. [Clusters]. "Class Clusters." 09 January 2012. *Concepts in Objective-C Programming, in iOS Developer Library.* <http://developer.apple.com/library/ios/#documentation/general/conceptual/CocoaEncyclopedia /ClassClusters/ClassClusters.html>.

[1] host class: a class that defines, or "hosts" behavior (method), as distinct from a class that presents the method by declaration or by a reference mechanism.

Apple, Inc. [Customizing]. "Customizing Existing Classes." 13 December 2012. *iOS Developer Library, in Programming with Objective-C.* <http://developer.apple.com/library/ios/#documentation/cocoa/conceptual/ProgrammingWithObjectiveC/CustomizingExistingClasses/CustomizingExistingClasses.html>.
Apple, Inc. "Object Initialization." *Concepts in Objective-C Programming*-. <http://developer.apple.com/library/ios/#documentation/general/conceptual/CocoaEncyclopedia/Initialization/Initialization.html>.
Begemann, Ole. "Faking instance variables in Objective-C categories with Associative References." 12 2011. *iOS Development.* May 2013. <http://oleb.net/blog/2011/05/faking-ivars-in-objc-categories-with-associative-references/>.
Doric Loon, et al. "Synchronic Analysis." 15 May 2013. *Wikipedia.* 02 June 2013. <http://en.wikipedia.org/wiki/Synchronic_analysis>.
Eclipse Foundation. "Aspectj crosscutting objects for better modularity." June 2013. <http://www.eclipse.org/aspectj/>.
Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
Hirschfeld, Robert, Pascal Costanza and Oscar Nierstrasz. "Context-oriented Programming." *Journal of Object Technology* March-April 2008: 125-151. <http://www.jot.fm/issues/issue_2008_03/article4/>.
informatique. "Software Engineering Terminology." *Umons Université de Mons.* 2013. <http://informatique.umons.ac.be/genlog/SE/SE-contents.html>.
Liskov, Barbara. "Data Abstraction and Hierarchy." *SIGPLAN Notices* May 1988.
Microsoft. "Partial Classes and Methods ." 2013. *Visual Studio 2012 or maybe C# Programming Guide, it says both.* June 2013. <http://msdn.microsoft.com/en-us/library/vstudio/wa80x488.aspx>.
Odersky, Martin. *The Scala Language Specification, Version 2.9*. Switzerland: PROGRAMMING METHODS LABORATORY (EPFL), 2011. <http://www.scala-lang.org/docu/files/ScalaReference.pdf>.
Oracle, Inc. "Default Methods." 2014. *Java Tutorials.* <http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html>.
Reza, J. R. "Java Supervenience." *Computer Languages, Systems & Structures* 38 (2012) 73–97 (2011).
Wilson, Eduard Osborne. *Consilience: the Unity of Knowledge*. Vintage Books, imprint of Random House, 1998. <http://wtf.tw/ref/wilson.pdf>.

Received September 2013; revised October 2013; accepted October 2013