Engineering testable and maintainable software with Spring Boot and React

Conor Muldoon, Levent Görgü, John J. O'Sullivan, Wim G. Meijer, Bartholomew Masterson, and Gregory M. P. O'Hare

Abstract—Single-page applications with RESTful APIs have become a dominant architecture in software development over the past decade. The success of this architecture is driven, in part, by libraries and frameworks, such as React and Spring Boot, that aid in the development of complex front-end JavaScript components and back-end Java classes and controllers.

Applying the 'Law of Demeter' (LoD), or principle of least knowledge, is key to the development of object-oriented code that is testable and maintainable. Dependency injection eases the burden of developing code that complies with the LoD. With JavaScript, adopting a functional style, but with managed side effects, as with React hooks, reduces the complexity of components and enables optimisations to be performed.

The LoD does not provide guidance for code with primitive variables or functional aspects. This paper modifies and extends the LoD for non-pure object-oriented languages, first-class JavaScript functions, and the properties of React components. Component closure factories are introduced to aid in the development of declarative React components. This extends the application of the principle of least knowledge beyond pure object-oriented systems and validation is provided with a case study on Unison, an application that enables the tracking and graphing of numerical weather forecast data.

1 Introduction

Over the past decade, a new architecture for web applications has emerged through the uptake of JavaScript frameworks, such as Angular [1], React [2], and Vue [3], that support the development of Single Page Applications (SPAs). With SPAs, as the user interacts with the browser, the current page is rewritten locally using JavaScript, rather than requesting completely new pages from the server. SPAs have several advantages over traditional approaches to web development. They reduce the latency and avoid interruptions due to successive page loads, making the application user experience more responsive and feel more akin to desktop or native applications than would otherwise be possible.

This paper discusses the principles of designing testable and maintainable code specifically in the context of using React for front-end component development and Spring Boot for developing back-end RESTful APIs. With SPAs, all of the content required for the application is either completely obtained when the page is loaded or dynamically obtained and added to the page as the user interacts with the system. In the case of React, when the application obtains data from the server, it is typically in the form of data in a JSON format. Once the data has been obtained, the state of the front-end components is updated and they are then re-rendered using a virtual DOM[1].

Spring is a web development framework that was designed to enable developers to enjoy the benefits of Java Enterprise Edition (EE) whist minimising the complexity encountered in application code [4]. Spring Boot (see Appendix A) is the Spring framework's convention over configuration solution that takes an opinionated view[2] of the Spring functionality and third-party libraries to create stand-alone production-grade applications that are automatically configured where possible.

The focus of this paper is on the design principles for the development of React and Spring-Boot applications such that they are maintainable and easily amenable to testing. We advocate adopting a modification of the 'Law of Demeter' (LoD) [5], referred to as the 'Law of Persephone' (LoP) [3], for object-oriented development and adopting a functional but, nonetheless stateful, programming style, in the development of React components through the use of hooks. The LoP, introduced in Section 4, modifies the LoD and extends the application of the principle of least knowledge to non-pure object-oriented languages. Although the LoD has been applied to non-pure object-oriented languages for some time, it makes no reference, or provides no guidance, with regard to the non-pure parts of the codebase or the use of primitive variables.

The LoD and LoP do not account for first-class functions and, as such, the LoP is extended in Section 5 to cater for

- *C. Muldoon and L. Görgü are with the School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland. E-mail: {conor.muldoon, levent.gorgu}@ucd.ie*
- *J. J. O'Sullivan is with the School of Civil Engineering, University College Dublin, Belfield, Dublin 4, Ireland. E-mail: jj.osullivan@ucd.ie*
- *W. G. Meijer and B. Masterson are with the School of Biomolecular and Biomedical Science, University College Dublin, Belfield, Dublin 4, Ireland. E-mail: {wim.meijer, b.masterson}@ucd.ie*
- *G. M. P. O'Hare is with the School of Computer Science and Statistics, Trinity College Dublin and the School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland. E-mail: gregory.ohare@ucd.ie*

Manuscript received August, 2021; revised September, 2021.

1. Virtual DOM is something of a misnomer and may not represent a HTML DOM, as is the case with React Native.
2. With opinionated frameworks, software is configurated by default in accordance with best practices or how it is most frequently used.
3. In Greek mythology, Persephone was the daughter of Demeter, the queen of the underworld, and the personification of vegetation.

the use of JavaScript first-class functions in general. It is then further extended for React to enable the use of the JSX syntax and React component properties (see Section 5.1). Although, object-oriented and functional programming are often viewed as alternative technologies, in this paper they are viewed as complimentary [6]. Indeed, functional concepts are increasingly being incorporated into traditional imperative object-oriented programming languages, such as with the introduction of lambda expressions in Java 8.

In addition to the principle of least knowledge, the paper discusses other development guidelines that increase testability, such as avoiding static code in Java. The paper introduces component closure factories (see Section 5.2), in the context of React, to address the prop drilling problem.

In developing testable code, one of the key requirements is that the tests should enable bugs to be easily localised to aid in the debugging process. Consider the analogy of how an aeroplane is tested. The engineers do not build the entire artefact, without testing anything, and then see if it flies, rather the components of the aeroplane are tested at an individual level, at an integration level, at a systems level, and at an end-to-end level. Providing different levels of abstraction for testing, makes it easier to identify where in the system a fault has occurred, particularly in the case where it is a low-level fault. Testing in this way is necessary where there are many interacting components. To illustrate how the principle of least knowledge, and other development guidelines, are adopted in practice, this paper provides a case study on Unison - a web application and front-end, developed by the authors, for tracking, obtaining, and graphical numerical weather forecast data from meteorological services. The case study discusses the architecture of Unison and the manner in which hypermedia is adopted to increase decoupling between the client and server. .

2 Related Research

There are a large number of textbooks that have been written on both React and Spring Boot, but critical analyses and comprehensive research studies of the frameworks are somewhat lacking in the literature.

Madsen et al. [7] perform a formal analysis that captures the essence of React with the goal of aiding in program understanding and of facilitating the development of bug finding static analysis tools. The small-step operational semantics models three key operations of React - the mounting and unmounting of components, the reconciliation of component descriptors and mounted components, and state transitions. The notion of a *well-behaved* React component is introduced whereby a ranking function is imposed on components such that the render methods of components only return components of a lower rank. Well-behavedness is then proven for the aforementioned operations. This work is presented in the context of React components that use ECMAScript 6 (ES6) classes. Recently developed React components, however, use hooks for managing state and side effects (see Section 5) and many pre-existing components have been re-engineered to avoid using ES6 classes.

MSstack [8] is a full-stack framework that is used to model both business processes and domain knowledge through a unified specification that is used in the production of boilerplate code for applications that adopt a microservices architecture. The system has been benchmarked against Spring Boot and the Web Services Oxygenated 2 (WSO2) Microservices Framework for Java [9] (MSF4J). The results show that MSstack performs better with regard to throughput and the framework reduces the memory usage in addition to latency. MSstack is less general than Spring Boot and takes an *opinionated* approach in focusing solely on microservice related components. Although the system is full stack in that it enables the development of user interface components, it does not incorporate modern front-end JavaScript technologies, such as React, and does not support the development of single page applications. The following test types were identified to be architecturally supported by MSstack - unit testing, contract testing, integration testing, load testing, and end-to-end testing. Mutation testing (see Appendix F.3) was not identified and no reference was made with regard to the general testability of the code. MSstack uses event sourcing for persistence and operations on data are not atomic. Thus, consistency is eventual and not immediate/strong.

Guice [10], [11] is a framework that was developed by Google to enable dependency injection and avoid the use of the new keyword in Java (see Section 4.4). It was the first Java framework to use Java annotations for dependency injection and was the winner of the Jolt award in 2008. Guice facilitates the development of code that conforms to the 'Law of Demeter' (LoD) [5]. In the Google Clean Code and Design Technology talks [12], [13], Hevery[4] discusses the advantages of the LoD and constructor injection (see Section 4.10). The perceived issue with the LoD of wide class interfaces and the need for forwarding methods is not due to the LoD *per se*, but rather to not using it correctly. If constructor injection is used, the new keyword is avoided, and the class is only provided with the dependencies it requires, this problem does not occur (see sections 4.1 and 4.11).

One of the problems of developing code in this way, however, is that, if implemented without using a dependency injection framework, it requires the development of a large number of factories to create the instances of the objects required. Consider Agent Factory Micro Edition (AFME) [14], [15], [16] , which is an intelligent agent platform for resource constrained mobile devices, the design of which was strongly influence by the LoD. AFME facilitated agent migration and was designed for devices that support the Java Micro Edition (JME) Connected Limited Device Configuration (CLDC) [17]. To support migration, the platform required decoupled actuator and perceptor (sensor) Java classes to be dynamically loaded at runtime using reflection. CLDC had limited support for reflection, however, and could not support constructor injection. This required the creation of factories for each class in question. Java Standard and Enterprise editions, however, have greater capabilities in terms of reflection and dependency injection frameworks, such as Guice and Spring, remove or reduce the need to invoke the factory pattern.

4. Hevery was the original developer of AngularJS.

3 Nomenclature

The manner in which terms, such as object-oriented programming, message passing, beans, and late binding are used within the literature varies. In this section, definitions are provided that illustrate how the terms are used in the context of this paper. The definitions are intended to be stipulative rather than lexical, save the object-oriented programming definition. Lexical definitions can only be considered correct or incorrect with regard to how well they reflect usage. Stipulative definitions cannot be considered correct or incorrect, rather they are adopted to avoid confusion and unnecessary arguments over the signification terms.

The concept of user defined types or classes originated in the Simula programming language. The term object-oriented, however, was first coined by Alan Kay in the late 1960s. Simula was a major influence on Smalltalk, the language that Kay developed. Kay was trained as a microbiologist, prior to moving into computer science, and developed object-oriented programming as a metaphor to cell communication. This is in contrast to how the term is currently used with regard to languages, such as Java. The two main contributions Kay made were in relation to message passing and late binding.

One of the problems with current object-oriented languages is that objects are dependent on a lot of environmental information and are not really modular. Consider the case where you wish to use a Java class previously developed for a different application or package. Not only do you have to include the class, but all of the classes it inherits from and those interfaces it implements. Additionally, all of the classes and interfaces it has dependencies on at both a class and method level must also be included. Not only that, but for each of those classes and interfaces, their dependencies and ancestors must also be included, and so forth. Maven can be used to address this problem, but it requires importing the entire software package rather than a subset of it.

According to Kay [18], the concepts behind what are now referred to as objects in languages, such as Java, date back to the work of Barton [19]. Barton designed a computer architecture that was never realised in hardware but was the forerunner to the JVM. Kay's objects used extreme late binding and were non-deterministic. With extreme late binding, upon receiving a message, an object may reject a request or respond that it does not understand the message. What is referred to as message passing in Java programming is technically equivalent to method invocation. This is discussed in greater detail below. Developing code capable of dealing with non-deterministic message passing leads to the creation of software that is more adaptive to change and unexpected behaviour. Kay referred to this as scalability, but scalability in the sense of developing a system of many interacting components, rather than in the algorithmic sense or with regard to data structures.

The dictionary definition of object-oriented programming [20] is adopted in this article, which does not empathise message passing and late binding, but aligns with how the phrase is used by practitioners of the Java programming language:

> *"using a methodology which enables a system to be modelled as a set of objects which can be controlled and manipulated in a modular manner."*

Nevertheless, it is acknowledged the important role concepts, such as late binding and reflection, play in the Java programming language, albeit to a lesser extent than Smalltalk. Without late binding, for instance, it would not be possible to create mock instances of classes for testing using frameworks, such as Mockito. With Java, it is not possible to support extreme late binding.

Accessor methods are frequently used in Java APIs. The term accessor is often used to refer to either a getter method or a setter method, but sometimes usage of accessor only refers to getter methods and mutator refers to setter methods. In this article, getter and setter will be used to refer to methods that only return objects' attributes and only assign objects' attributes respectively. The usage of setter here differs from React hook set functions for updating state discussed in Section 5. In developing Smalltalk, Kay's goal was to create universally isolated cells that could mimic any desired behaviour and only communicated using messages. The use of accessor methods would not be considered message passing in this setting.

Kay noted that many Java APIs have a large number of setter methods, which convert an object to a data structure and cause problems due non-local state mutations [18]. APIs written in Java do not have to be implemented in this way.

In a somewhat controversial article [21], Holub discusses maintainability problems caused by the overuse of getters and setters in the context of Java. Sections 4.13 and 4.14 discuss how getters and setters can be avoided without exposing the states of objects and provide the rationales for doing so. There are, of course, edge cases at the "procedural boundary" of the software, as acknowledged by Holub, where getters and setters should be used.

Holub refers to message passing, but does not elaborate on what is meant by this in a Java setting. Message passing can refer to several things, such as communication between threads, communication between objects, or distributed communication. Message passing, in terms of object communication in Java is technically equivalent to a method invocation - a cached lookup up of code to be executed for a named operation. The code is implicitly passed the object on which it was called and can access all methods, variables, and objects within the class it is associated with and instances of the class it is associated with that were passed as arguments (including, for example, private attributes). This is different from method invocation in other languages, such as the Actor language. Static method invocations in Java are different to method invocations on objects. A static method is almost equivalent to a procedure call in non-object-oriented languages, such as C. It is independent of an object instance, but it is related to a class with regard to visibility and access to static methods, variables, and objects within the class.

JavaScript supports first-class functions, which are implemented as Function objects that can be passed to other functions as arguments and returned by other functions. Function objects can have properties, but differ from JSON objects in that they are callable. In the context of JavaScript, when the paper refers to methods, it is referring to

JavaScript functions that are also properties of JSON objects or other JavaScript Function objects.

In this paper, HTTP methods are referred to as methods rather than verbs as that is what they are referred to in the HTTP specifications, which are standardized.

Support for dependency injection is one of the key features of the Spring framework. In [12], [13], dependency injection is taken to simply mean that only assignment is performed in the constructor. More generally, however, particularly in the context of the Spring framework, dependency injection has a broader scope. In Spring, for instance, support is provided for field and method injection, although this is ill advised (see Section 4.10). In this paper, dependency injection is taken to be a design pattern that separates the creation location of dependencies from the client classes or objects that require them; the injection of dependencies to clients is achieved through the use of either a reflection-based injection framework, such as Spring, or code generation framework, such as Google Guice.

Bean is a term that is used inconsistently within the Java development community. In this paper, beans refer to Spring beans, rather than Enterprise Java Beans (EJBs). They are POJO objects that are instantiated, managed, and injected into the IoC container by the Spring Framework. Within Spring, beans are created using a recipe, referred to as a bean definition, for instantiating object instances and determining the scope of the instances. The scopes supported are singleton, prototype, request, session, and global session. The singleton scope is the default scope and creates a single object instance. The prototype scope creates an object instance for every Spring component that has a dependency on the bean class. The request scope creates a bean for the lifecycle of a HTTP request and the session scope and global session scopes for of a HTTP session and a global HTTP session respectively.

4 Principles of Maintainability and Testability for Spring

A number of take-home lessons have been learnt together with coding guidelines regarding the development of Spring and Java code that is *maintainable* and *testable*. This section discusses adopting the principle of least knowledge along with other development guidelines that serve to increase the modularity of the software and enable objects to be decoupled for unit testing. One of the key advantages of using Spring is that it facilitates dependency injection (see Appendix A for an overview of Spring). Dependency injection aids in avoiding static code, which is difficult to test. Some of the principles, discussed in this section, are applicable to Java, in general, but there will be additional difficulties or trade-offs if developing a Java application without the support of a dependency injection framework.

4.1 Least Knowledge

The Law of Demeter (LoD) or principle of least knowledge [5] is an object-oriented programming language style the encodes the ideas of encapsulation and modularity and can be summarized as follows:

- Each unit should have only limited knowledge about other units: specifically, only units "closely" related to the current unit.
- Each unit should only talk to its friends; don't talk to strangers.
- Only talk to your immediate friends.

The LoD for functions requires that for a method *m* of an Object *O*, *m* may only invoke methods of the following types of object:

1) *O* itself.
2) *O*'s attributes.
3) *m*'s parameters.
4) Objects instantiated within *m* or by methods *m* calls.
5) Global objects (static in Java).

It is instructive to consider the types of method invocations the LoD prohibits rather than the types of method invocation it permits. Although the getter methods of objects passed as method arguments or attributes are not explicitly prohibited by the LoD, none of the methods on the objects returned by the getter methods could be invoked if following the law. This renders getter methods *ineffectual* in such cases. In object-oriented languages that are not pure, variables of primitive types such as ints or doubles could be returned by getter methods and these variables could be used by the calling object. The LoD is only concerned with objects and method invocation and, as such, does not prohibit this. In this paper, however, we argue for the use of delegation in protecting objects' states and for the avoidance of this type of getter method in vast majority circumstances (see Section 4.13). Section 4.2 introduces the LoP, which modifies the LoD and extends the application of the principle of least knowledge to non-pure object-oriented languages. It should be noted that the LoD has implications beyond the use methods of objects accessed via getters; it is more general than that.

Although the LoD, from a purely object-oriented perspective, effectively renders the accessor methods of attributes and objects passed as arguments ineffectual, it does not prohibit the invocation of methods on objects instantiated within the methods called. The prohibition only relates to objects that existed when the call began. As such, *fluent* APIs, as used within the Spring Security configuration classes, that return new objects on method calls are permitted. When not using fluent APIs, a common rule of thumb, referred to as "use only one dot", can be used to identify many LoD violations. For example, it would catch the a.getB().doC() violation. It would not catch the violation if getB() was assigned to a variable and the doC() method was thereafter invoked.

Strictly speaking, the LoD only relates to invoking methods on objects returned from the methods of attributes or arguments. Invoking methods in a method, M, on objects returned from getters of a newly created object, O, not passed as arguments to M or not attributes of M's class is not a violation provided the getters don't return attributes passed to the constructor of O that were instantiated prior to M's invocation. This goes against the spirit of the LoD though and should be avoided in that there is no guarantee that the O's class won't be used incorrectly by developers not familiar with the LoD. Moreover, using delegation with newly created objects constructed after M's invocation will tend to lead to greater reuse, particularly if the objects' classes are used elsewhere.

A common misconception with developing code that conforms to the LoD is that it can lead to the development of redundant forwarding wrapper methods that increase the footprint of the software and only propagate method invocations. As noted by Hevery [12], this problem will not occur if objects are only passed dependencies that they directly need (see Section 4.11), which is in the spirit of the LoD. If a wrapper is being used, it means that the object whose method is being called through the wrapper should have been a dependency or attribute of the calling class. Wrapper methods are not a consequence of the LoD. One of the true disadvantages of the LoD is that it can lead to the creation of a large number of factories, but this is alleviated through the use of dependency injection.

4.2 Law of Persephone

In this paper, we introduce a modification to the LoD, referred to as the *Law of Persephone* (LoP), which extends the application of the principle of least knowledge to non-pure object-oriented languages, such as Java, that make use of primitive types. Although the LoD has been applied to non-pure languages for some time, it provides no guidance with regard to the non-pure functionality.

With regard to objects, the LoP is more restrictive than the LoD in that it does not allow the invocation of methods on static global objects by the non-static methods of objects[5]. A variety of issues related to using static code are given in sections 4.3 to 4.7. Additionally, the LoP is more restrictive on the type of attributes methods can be invoked on. This additional restriction prevents the invocation of methods for attributes assigned using setter methods (see 4.14), but it is more general than that in that it also applies to methods that do more than just assignment.

The LoP is defined in two parts that relate to permissible actions that may be performed on objects and primitive variables. The LoP for methods requires that for a non-static method m of an Object O, m may only invoke methods of the following types of object:

1) O itself.
2) O's attributes where the attributes were assigned within the constructor or instantiated within O.
3) m's parameters.
4) Objects instantiated within m or by methods m calls.

The LoP for primitive variables requires that for a non-static method m of an Object O, m may only use variables of the following types:

1) O's attributes where the attributes were assigned within the constructor.
2) O's attributes where the attributes were assigned within a method p of O and were not a function of p's parameters.
3) m's parameters.
4) Variables created within m or by methods m calls.

In a similar manner to the LoD, the LoP for primitive variables prohibits the use of variables that were obtained using the getter methods of attributes or parameters, but it is more general in that it also applies to methods that do more than only return attributes. In a similar manner to the LoP for methods, attributes that were assigned as a function of the method's arguments are also prohibited.

5. Static methods may invoke other static methods in the creation of static constants used by non-static methods.

4.3 Avoid using static methods

One of the key advantages of object-oriented programming over procedural programming is that object-oriented systems introduce a seam that enables functionality to be modularized. This significantly increases the testability of the codebase. As noted in Section 3, static method calls in Java are almost equivalent to function calls in procedural languages, such as C. This makes them difficult to test. The reason for this is that static method invocations cannot be decoupled from the class in which they are invoked. This inhibits unit testing and causes problems with regard to mocking and using mocking frameworks (see Appendix F.1), such as Mockito. In contrast, if static methods are avoided and the functionality is provided by objects passed as dependencies to the constructor, it is easy to mock the dependent functionality and localise the scope of tests.

4.4 Avoid using the new keyword except within bean definitions and factories

Two of the objectives in the development of the Google Guice framework were to avoid writing code that creates objects using the new keyword and to minimise the need for writing factories [11]. The reason it is advantageous to avoid the new keyword is that it is static and difficult to test [12]. In a similar manner to the tight coupling of static methods to the calling class (see Section 4.3), an object created using new cannot be decoupled from the class that creates it. Ideally, rather than creating a new instance, an object instance would be passed as a dependency to the constructor. The dependency, in this case, could be then mocked, which would simplify the writing of unit tests. Within the Spring framework, bean definitions are used for creating object instances that are injected into components that have dependencies on the created objects. As an example of the problem opaque dependencies, consider the equals method for the URL class within Java. The equals method is a blocking operation, it performs a DNS lookup, and requires a network connection. If the DNS lookup functionality had been captured as a separate class and made an explicit dependency in the constructor, it would be easy to mock. This would reduce delays in testing.

The approach of using beans for dependency injection cannot be used in cases where methods need to create new instances. In such cases, factories can be used to create the instances. This was the approach adopted in the creation of actuators and perceptors to support migration in AFME [15] for instance. The advantage of using factories, in a testing context with Spring, is that the factories can be mocked and passed to the constructor or nulled out.

Developing all code without using the new keyword except within beans in Spring, however, would be somewhat cumbersome. In situations where the object being created is a leaf object, such as a hash map or array list data structure, an object that takes no arguments to the constructor, or the object is part of an external API that the developer does not wish to mock, the new keyword should be used to prevent the codebase becoming over-engineered.

4.5 Avoid the service locator pattern

The service locator pattern (getContext()) enables services or global state to be accessed through the use of a static method. It violates the LoP and hides class dependencies, which makes it more difficult to test using mocks and less obvious to the test developers or other users of the class of what the dependencies are. It is better to make the dependencies to a class explicit.

Using the service locator pattern, reduces the possibility for parallelisation of code to run on threads. It makes classes dependent on a large graph of objects rather than only the objects that they require.

Consider the problem of classes and objects being dependent on a lot of environmental information discussed in Section 3. When using the service locator pattern, not only does the class have a dependency on the classes it uses, but on all of the classes of the service locator and all of their dependencies, and so forth.

4.6 Avoid the singleton pattern

The singleton pattern uses static code to create a single instance of an object and ensures that no additional instances are created. Using the singleton pattern is akin to using global variables. It does not introduce a seam for mocking or enable dependencies to be nulled out when testing. In a similar manner to the service locator, it hides the dependencies of classes and should be avoided in that it increases the difficulty of testing and does not make it obvious to users of the class what the dependencies of that class are. The users of the class would need to look at the class source code to determine that.

This paper does not argue that singletons or single instances of classes should not be used, rather that the static singleton pattern should not be used. Dependency injection should be used to achieve the same ends. The default bean scope in Spring Boot is singleton (see Section 3), which ensures only a single instance of the object is created for the bean definition.

4.7 Inject enumeration arrays

Enumerations in Java are intended to be used in situations where a fixed set of static constant objects are required. The static values enumeration method is used to obtain an array of the enumeration objects. Enumerations in Java are implemented as static final classes. Injecting enumeration arrays rather than using the values method in dependant classes[6] is more declarative with regard to dependencies and increases flexibility with regard to testing. For instance, it is possible to null out an enumeration when unit testing or pass a zero-argument array.

6. The values method may be used in bean definitions.

Using dependency injection in this way also increases flexibility with regard to application developed. This flexibility, for example, enables Unison (see Section 6) to use different sets of weather variables.

Although, at run-time, the set of values an enumeration represents will not change, this article argues that removing as much static code as possible from application code, in general, is a boon for testing.

4.8 Avoid work in the constructor

In creating objects, work in terms of limiting the scope of unconstrained variables or object dependencies should be performed in the constructor. For instance, determining whether a number falls within a range. Typically, such work would consist of checking an if statement and throwing an exception if a constraint is violated.

Situations where constraint checking is required occur rarely, however, and in writing testable code, the only work that should be performed in the constructor, beyond constraint checking, is assignment. The reason for this is that to test code that is not static, the constructor needs to be invoked and work performed within cannot be isolated and must be invoked. Thus, none of the functionality of the rest of the class can be tested independently. If the code in the constructor performs a blocking operation or is slow, this will slow down all tests related to the class.

A further reason for avoiding work in the constructor is that it is cleaner from an API design perspective in that the constructor only addresses a single concern and does not invoke opaque functions. If a dependency needs to be in a particular state prior to the object being used, the work should be done on the dependency prior to it being passed to the constructor. This could be done within a factory or a bean definition if it needs to be repeated in different parts of the codebase.

4.9 Only use a single constructor

The constructor should contain all of the dependences for the class and there should not be another constructor with fewer dependencies or, indeed, different dependencies. When developing application code, none of the dependencies should be null. This ensures robustness in that classes cannot be partially instantiated. When developing test code, it is permissible to null out some of the dependencies. This enables the test developer to signal their intention with regard to parts of the class the test targets. Although better than passing null, optional dependencies should not be passed using the Optional class (see Section 4.15), as this would lead to LoD violations; rather the class should be refactored or redesigned to avoid the optional dependencies.

Functionality that has different dependencies should be represented using different classes. To avoid the need for optional dependencies, a class that has optional dependencies should be divided up into a number of classes. One class that captures the common (non-optional) functionality and a number of additional classes that require the common functionality in addition to other (optional of the original class) dependencies. The class that contains the common functionality should be passed as a dependency to the other classes. In this way, the need to check for the presence of a dependency is avoided. If a class has a large number of dependencies, it is sign that it is addressing multiple concerns and optional dependencies are indicative of a lack of cohesion. Such classes should be redesigned rather using constructors that only receive a subset of the dependencies.

4.10 Use constructor injection

With Spring Boot, it is possible to inject dependencies, using the autowired annotation, at an attribute level or using setter methods. Both of these approaches should be avoided in favour of constructor injection. There are a number of reasons for this. In the case of attribute injection, using constructor injection as an alternative transforms the object from having a dependency on the injection framework to being a POJO. POJOs can simply be created using the new keyword and tested independently of the injection framework, which often makes testing easier. If setter injection is used, POJOs can be created, but if a new dependency is added to the class, the test developer may neglect to perform the set and the code will still compile, but a null pointer exception will occur when the tests are run. Avoiding setter injection and only using a single constructor for injection (see Section 4.9) avoids this issue.

In testing, it is permissible to use autowired attribute injection. The reason that this is not a problem is that the test code itself does not need to be tested (see Appendix F). JUnit requires that test classes to only have a single zero-argument constructor. This prevents constructor injection. Although following the LoP and avoiding setter injection for application code makes it easier to develop tests that do not require the Spring testing support, attribute injection in test code using the Spring testing framework is useful to avoid creating factories or creating large number of new objects.

4.11 Only ask for dependencies you directly need

Creating new objects in the constructor should generally be avoided, but in some cases should be undertaken in order to prevent the code becoming overly complicated (see Section 4.4). If an argument to the constructor is only used in the construction of an attribute, which strictly speaking is not a LoD violation, the attribute should have been created by the calling class and passed as an argument to the constructor rather than the attribute's dependency being passed as an argument.

Situations in which a class is only propagating method calls to attributes indicate that the structure of the codebase is incorrect and should be redesigned. Consider, for example, the case of where the argument, A, of a method is only being propagated by an object, B, to the method of an attribute C. The object in which the method call originated, D, should have had a reference to C and invoked the method on C directly passing A as an argument. That is, C should be a dependency in D's constructor or should have been instantiated by D, perhaps using a factory (see Section 4.4). In short, D should have a reference C rather than B.

Similarly, if attributes are only passed by methods to the methods of their arguments, it means that the dependencies are in the wrong class and should be moved to the calling class of the method. This situation, again, is not a LoD violation. Furthermore, if an argument to a method is only being used to create a new object, the object should have been created prior to calling the method and passed as the argument.

4.12 Avoid the extends keyword

In certain situations, it is not poor design to extend abstract base classes provided the base class is used in a similar manner as an interface, the protected keyword is not used, all member variables are private, and fragile base class issues are avoided. Concrete classes, on the other hand, should not be extended.

This paper advocates the use of interface inheritance (implements) over implementation inheritance (extends)[7]. One of the main issues with implementation inheritance is the *fragile base class problem*. With the fragile base class problem, seemingly safe modifications to a base class can lead to problems in derived classes. Consider, for example, the use of the protected keyword for instance variables in a base class. The protected keyword provides package level access in addition to sub class access. It violates the principle of encapsulation in that the instance variable will be visible to all derived classes. With this approach, the developers of derived classes need to be aware of the internal implementation details in the class ancestral hierarchy. There are additional issues due to the fragile base class problem, but they shall not be discussed in greater detail here. The default scope of the attributes of classes in Java also makes them available to other classes in the package[8] and, thus, if such variables are accessed outside of the class, the principle of encapsulation is violated. Attributes should be explicitly made private.

Although implementation inheritance should not be used in the development of application code, this paper does not advocate using the final keyword to prevent implementation inheritance in all cases. The reason for this is again testing. Classes cannot be mocked using the Mockito framework if they are final.

One argument against the use of composition is that it can lead to the development of a large number of forwarding methods. This is for a different reason than the forwarding methods discussed in Section 4.1, which are avoided if classes are designed to only have dependencies that they directly require (see Section 4.11). In this case, it is due to the fact that in some cases, the developer may wish to use a pre-existing implementation for most of the functionality, but provide new or different functionality for a subset of methods. Splitting large interfaces in the design is one approach to mitigating this issue.

4.13 Use delegation rather than getter methods

When developing software at the functional or procedural boundary or when creating an open, rather than closed, system or API, getter methods that return objects and primitive types are required. It is difficult to imagine how the

7. Default methods should be avoided where possible as discussed in Appendix D

8. The fields of interfaces are implicitly public static final and the methods of interfaces are public by default.

optional pattern (see Section 4.15) could be implemented, for instance, without using some type of getter method. In these cases, getter methods are preferable to making variables public. Notwithstanding this, in the vast majority of cases, getters should not be used.

In Section 4.1, it was noted that although getter methods do not violate the LoD, the invocation of methods on the returned objects does if the getters are invoked on attributes or objects passed as parameters. This renders getter methods, in these cases, useless. No methods can be invoked on the returned objects. Following the LoD requires that, rather than asking for an object through the use of a getter on an attribute or parameter, the dependency on the object should be made explicit by requiring it to be directly passed to the constructor or the methods that require it. The LoD only relates to objects and object-oriented programming. The Java programming language, however, is not a pure object-oriented language and contains primitive types, such as integers and floating-point numbers, for performance reasons. The LoP (see Section 4.2) modifies the LoD and extends the application of the principle of least knowledge to non-pure object-oriented languages. It prohibits methods from using primitive variables instantiated prior to the invocation of the methods and obtained using getters.

One of the core tenets of object-oriented programming and encapsulation is that the implementation details of objects should not be exposed to users of the objects. Getter methods violate this and effectively make primitive variables global. If a change is made, such as changing an integer to a long integer, with getter methods, this change will propagate throughout the codebase of the project and any project or system that uses the class. With object-oriented development, delegation should be used rather than getter methods. With delegation, an object does not request primitive variables or objects from another to perform a task, rather it asks the other object to perform the task on its behalf. This represents a more declarative approach and increases information hiding along with code reuse.

4.14 Avoid mutators

The LoP prohibits the use of attributes assigned using setter methods. In a similar manner to getter methods (see Section 4.13), when developing an open system or API, there are situations where setter methods or mutators should, indeed, be used. It could be argued that setter methods should be used for performance reasons in cases where mutation is required on a subset of the object's state. That is, as an alternative to requesting the object to create a new object instance of its class with a subset of its state passed to the constructor and new a value or values for the subset that is not passed. In the vast majority of cases, however, setter methods should not be used. Class dependencies should be made explicit through the constructor.

Setter methods turn objects into data structures and in doing so damage the maintainability of the code. They cause problems due to non-local or global state transitions and prevent code from being executed in parallel. If setter methods are used for initialising objects, rather than using a single constructor (see Section 4.9), it can lead to partially initialized objects. This makes the code brittle and difficult to test (see Section 4.10). The code will be brittle in that there will be no guarantee that the users of the class will have invoked all of the setters. The code will still compile.

Mutable state is often required due to human users. With object-oriented development, objects should be responsible for updating and modifying their own state. If setters are being used to avoid the new instantiation of very large objects, it's a signal that there is a problem with the design and the class should be broken up into smaller classes or functional units.

If the class is solving a purely functional problem, there no need for mutable state and objects can be made immutable, which is a more stringent requirement than avoiding setter methods. Indeed, purely functional languages often forbid mutable state and favour referential transparency. Using immutable objects enables optimisations to be performed and increases the prospects of running parts of the code in parallel or on different CPU cores.

4.15 Use the optional pattern

Tony Hoare has called the introduction of null to programming languages his "billion-dollar mistake" [22]. One of the issues with using null, in the context of Java, is that developers sometimes return null from methods that have an object return type. All method signatures in Java that return objects have an indeterminacy in that either an object or null could be returned. This can lead to defensive programming in that users of such methods cannot tell if null will ever be returned without looking at the method's code[9]. The optional pattern addresses this issue by enabling developers to signal their intention that the returned object is optional. That is, that it may or may not be present. The users of such a method know to add a check for the presence of the object that they wish to use. The users of methods, in general, need only look at the method signatures provided the use of the optional pattern is adopted as practice.

In Java, the optional pattern is realised in the Optional class[10], which is used to create immutable optional objects that can be either empty or contain an object that can retrieved from the optional object. Although, the Optional class contains a get method, the manner in which the Optional class is most frequently used is not a violation of the LoD. That is, when the optional is of a newly created object, which was created within the method that returns the optional. It is a violation of the LoD if a method, A, is called on an object returned from an optional that was passed as an argument to a method, B, that is calling A. It would also be a violation of the LoD if the optional was an attribute of B's class.

The LoD prohibits the use of objects returned from accessors on objects that were passed as parameters or are attributes. That is, objects that were instantiated prior to the invocation of the method. This paper advocates avoiding passing optional objects to methods or to constructors as invoking methods on the objects returned by such optionals would be a LoD violation. The indeterminacy between objects and null is not an issue when constructing objects

9. The documentation could also be consulted provided the code is well/correctly documented, but there is no guarantee of this.

10. The Optional class was introduced in Java 8.

or invoking methods. The problem only arises when the developer needs to look beyond the method signature to determine if a method returns null.

It's good practice never to pass null as an argument save testing, but this paper advocates designing the code in a different way rather than passing an optional instead of null. So, for example, if there is an optional argument for a method, the method should be replaced by two methods, one that has the additional argument and one that doesn't. The method with the additional argument may call the other method or common concerns between the methods could be captured within an additional private method. Similarly, classes should be refactored if they have optional attributes (see Section 4.9).

Optionals are sometimes passed to methods in Spring Boot when the method is being used for an endpoint mapping and the endpoint mapping has optional request or query parameters. In HTTP, however, different URIs represent different resources. There is no concept of an optional request parameter with regard to URIs. Ideally, an optional request parameter would be replaced by two mappings with the same path being used; one of them mapping the additional parameter and the other not. The former may then call the latter or common functionality could be captured in an additional private method as discussed previously. Spring Boot doesn't support this, however, but this could be achieved in Spring Boot with path variables. The path variables would not be URI decoded by the framework though, so this approach is unlikely to be used in practice with the current version. Thus, using Optional for query parameters should be considered an exception.

5 Principles of Maintainability and Testability for React

As is the case with the development of Java code, the principle of least knowledge is key to the development of maintainable and testable code in JavaScript. This section discusses two extensions of the LoP, namely the LoP for JavaScript (LoPJS) and the LoP for React (LoPR) (see Appendix B for an overview of React). The LoPR is an extension of LoPJS and is React specific. Additionally, component closure factories are introduced, which address the React prop drilling problem (see Section 5.2).

React is discussed in the context of React Hooks, rather than the original React implementation, which relied on ES6 classes. ES6 classes, when used in the context of the React lifecycle, are more difficult than hooks to maintain as they increase the complexity of the code. They also reduce the potential for optimizations, such as ahead-of-time compilation. When using hooks, components are written as functions that enable the use of React features, such as state, without using classes.

Two of the most widely used hooks in React are useState and useEffect, which enable state to be incorporated into otherwise purely functional components and side effects to be managed respectively. The convention for updating stateful variables in React is to name the function returned by useState as set variable[11] name where the variable name is the reference to the hook state. So, for example, if the variable name was myVar, the function for updating it would be called setMyVar. This convention is somewhat unfortunate in that setting state in React is not the same as using setter methods. Setting state in React does not simply assign the attributes of objects. Indeed, much of React was designed with avoiding mutable state in mind. For instance, when arguments are passed to components, they are attributes of an immutable object referred to as props. The hook set functions are defined within the hook function's scope and cannot be arbitrarily called by other components, only child components.

Hook set functions are asynchronous and their use can be thought of as scheduling an update to state. When the functions are called, React Fibre, which is React's core algorithm, generates a tree that represents a description of the GUI. If component types for the new state are not different from that of the current state, the tree is diffed with a previous tree in determining what operations need to be performed to update the rendered application. If component types are different, React assumes that the trees of the components will be quite different and does not perform a diff of such components, but creates a new tree for them. Lists are diffed using unique keys.

5.1 Least Knowledge

The LoP, introduced in Section 4.2 enables the use of object methods. It does not consider the use of functions directly. Functions in JavaScript are first-class. Languages with first-class functions enable functions to be passed as arguments to functions and functions to be returned by functions. The LoPJS adheres to the LoP and enables a function, f, to call functions of the following type:

- Functions declared within f's file that are within f's scope.
- Functions passed as arguments to f.
- Functions instantiated within and returned by a function, g, called by f.

To illustrate the types of function calling prohibited by the LoPJS, consider a "use at most one dot" analogue to the "use only one dot" rule of thumb, discussed in Section 4.1, and a function g passed as an argument to f. f can call g, but cannot call methods of g's attributes, only functions instantiated by g or methods of g (LoP adherence). Additionally, methods of imported objects can be called, but not methods of their attributes.

Components in React are, for the most part, used declaratively, although the useRef hook enables child components to be accessed imperatively. Components are typically declared using JSX, which is a syntactic extension to JavaScript that enables code that generates a description of the GUI, which is returned from components, to be written in HTML-like style. Components declared using JSX are compiled into JavaScript code. The attributes passed to a JSX component are passed to the JavaScript component functions as immutable JSON objects, referred to as *props*. Adherence to the LoPJS, would prohibit the use of the methods of objects passed as prop attributes or variables passed as prop attributes. Only functions could be passed as props. One

11. State could also be a JSON object or a function.

way to get around this problem is to avoid using JSX to create components and instead use JavaScript functions that accept variable, object, or function arguments directly rather than via props, but that return the same GUI descriptions. Ideally, however, we would still like to be able to use JSX when creating components.

Although React is responsive to user interaction and events, it is not reactive in the sense of functional reactive programming. React uses a scheduler. There are a number of reasons for using JSX when declaring components that go beyond being able to use a HTML-like syntax, such as with regard to React scheduling or with regard to optimisations such as hoisting constants. When components are declared using JSX the functions that define components are not called directly by the application code, rather React schedules them to be called at an appropriate time.

The LoPR addresses the issue of the prohibition of prop variables and the methods of prop objects and is slightly more permissive than the LoPJS. The LoPR adheres to the LoPJS and enables object methods and variables to be used by a component, C, of the following type:

- Methods of an object, o, where o is an attribute of C's props object.
- Variables that are attributes of C's props object.

It is recommended that the prop attributes should be accessed using JSON destructuring assignment[12] rather than referencing the attributes via the opaque props object. This makes the component's dependencies more explicit and also means that the rule of thumb of "use at most one dot" aids in identifying LoPR violations.

5.2 Component closure factories

This paper introduces the *component closure factory* pattern, in the context of React, to enable creation of components that have dependencies on the states of more than one component at different levels of the component hierarchy while addressing the React prop drilling problem and ensuring components have explicit component dependencies. The prop drilling problem is similar to the issue discussed in Section 4.11 whereby constructors or methods do not directly use object references passed to them, but only pass them on. Prop drilling occurs when components do not use the props directly, but only pass them on to child components. This can be addressed using composition where possible, but in cases where components have dependencies on the states of two or more other components, the situation becomes somewhat more complicated.

In JavaScript, and functional programming in general, a closure is a technique for implementing lexically scope name binding for languages that have first-class functions. Closures can be created in Java using lambda expressions, but not fully fledged closures. Lambda expressions in Java can only access final, or effectively final, variables or object references of the enclosing scope. That is, variables or object references cannot or do not change. In contrast, in JavaScript, closures can access variables, object references, and function references, in the enclosing scope, that change value.

Component closure factories work by creating a closure that accesses the state of a component, A, from the enclosing scope and returns a factory function that receives arguments form another component, B. The factory either creates a component, C, or returns another closure factory to be passed to another component (see Appendix E). The following is an example of a closure component factory for creating a component, MyComponent, based on the state of two components.

```
function createMyComponentFactory(a){

return function (b) {

return <MyComponent a={a} b={b} />

}
}
```

Rather than a component, A, passing a reference to one of its state hooks as a prop to the component, B, that creates MyComponent. Component A creates a factory closure that makes a reference to one of its state hooks from the enclosing scope and passes the factory closure to B. Component B then invokes the factory function for creating MyComponent passing a reference to one of its state hooks.

Using component closure factories in this way addresses the React prop drilling problem for creating components that rely on the state of components at different levels of the component tree. It reduces the number of props that are passed between components, increasing encapsulation and the modularity of the codebase along with reuse. It also makes the code more accurate with regard to the true nature of component dependencies. With this pattern, references to component state hooks are only passed to the components that require them. One of the drawbacks of creating components in this way though is that the developer must invoke factories in JavaScript rather than embedding JSX directly.

Alternative approaches to addressing the prop drilling problem include using the Context API or Redux [23]. The Context API makes state global and component reuse difficult and is only intended for use with state that changes infrequently, such as with the authenticated user or the theme. The Context API is used when data needs to be accessed by *many* components at different nesting levels. Redux is an external library and state container that centralises state, aids with state persistence, and provides management features such as the capacity to undo/redo state transitions. It represents an entire state management system. Component closure factories represent a lighter-weight solution, which does not make state global or centralised or introduce a dependency on a third party library.

5.3 Avoid passing hook set functions to children

As noted previously, state is updated in React using functions, typically referred to as set functions, which is an unfortunate convention in that they are not setter methods in the traditional object-oriented sense (discussed in Section 4.14). Although hook set functions can be passed as props to

12. With JSON destructuring assignment, properties of JSON objects are unpacked and accessed directly.

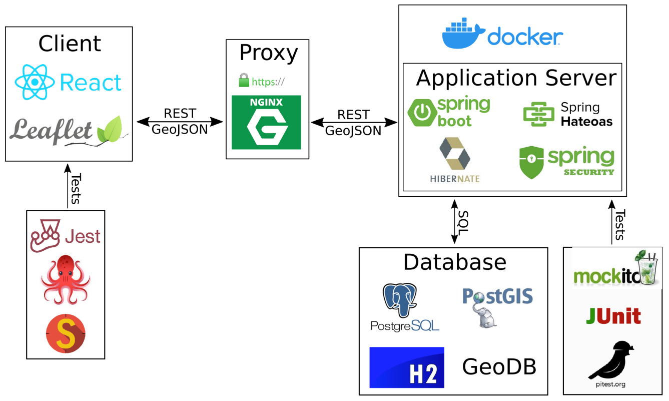

Fig. 1. The four tier Unison Architecture.

child components, this paper advocates avoiding this, where possible, by either merging the child component with the parent component or passing a call-back to the child component that enables the set function to be called indirectly, but limits what the state can be set to. This prevents the state from being updated arbitrarily by child components and increases decoupling. Furthermore, moving functionality from the child to the parent leads to increased reuse if the call-back is passed to other child components. An exception to this would be call-back functions that only propagate calls to the set functions. Such call-backs should be avoided.

6 Case Study

Unison is an application, developed by the authors, that enables the user to track and visualise numerical weather forecast data and developers to access the data via a RESTful API. Specifically, Unison has been designed for use with HARMONIE-AROME [24] weather forecast data. HARMONIE-AROME is used by the meteorological services of 10 countries within Europe. Unison has been released as an open-source project [25] and is available for use under an MIT license. It was developed as part of the Acclimatize Project [26] and the weather forecast data is used in machine learning models for determining causal factors in relation to poor water quality. The purpose of this case study is to serve to illustrate how the principles covered in sections 4 and 5 have been applied in practice and the solutions adopted to adhere to them.

Tables 1 and 2 provide a desiderata of the practices and patterns to avoid and adopt respectively, advocated by the paper and adhered to in Unison, along with the primary benefits and caveats of doing so. For a summary of these recommendations of the paper, see Appendix H.

6.1 Unison Architecture

Figure 1 illustrates the four tier Unison Architecture. The functionality delivered by Unison can be viewed in terms of functional and non-functional requirements. The client-side code that addresses the functional requirements were developed on top of React and Leaflet. On the server side, the functional requirements are delivered using Spring Boot, Spring HATEOAS (see Appendix A), and Hibernate (see Appendix C). The other components of the architecture relate to the non-functional requirements of standardization, scalability, discoverability, reliability, GIS support, REST compliance (see Appendix G), security, and operations.

6.2 Spring Boot Application Implementation

Unison stores data for weather variables, user credentials, tracked locations, and the spatial reference system for PostGIS. It contains a scheduler and periodically connects to a HARMONIE-AROME endpoint in order to obtain the latest forecast data. The HARMONIE-AROME schema separates precipitation data from the other weather variables and the data are stored in two separate tables every 6 hours. Unison contains controllers to enable data for individual weather variables to be individually queried and to be returned in either a JSON or CSV format via content negotiation (see Appendix G).

The JSON controllers are injected with finders to obtain data for the given query parameters. The finders are beans that are injected with query string beans, a bean that supports HTTP caching, and Spring's entity manager. There are separate beans and controllers for each variable and the finders use the entity manager to invoke queries and the caching support bean to add the HTTP headers for responses. If the end date for the range of the data requested is in the past, the data is tagged as immutable as it will not change and given a large maximum age in the cache control HTTP header. This enables aggressive HTTP caching to be performed. In creating queries, the JPA entity class (see Appendix C) for the data is used to determine the table names, which ensures that if there is a refactoring, the table names will be updated and that potential typos in the JPQL query string are avoided[13].

Unison is fully compliant with the LoP. In terms of serializing Java objects to JSON, Unison uses Jackson to annotate attributes rather than getter methods. It could be argued that this exposes an object's state, but not in terms of the Java language. A developer using the Java API could not access the private attributes without using reflection. This is at the procedural boundary of the system in any case. Attributes of JSON objects in JavaScript are public. This does not mean they have to be accessed directly. Future work will investigate returning JSON objects that contain JavaScript functions. The client could then use the functions to perform operations. This would make the system more dynamic with regard to delivering code on demand, which could aid in the development of caching strategies.

The CSV controllers are injected with responders to obtain data. Unison favours composition over inheritance and the responders are beans that are injected with the finders, discussed previously, for each variable. The CSV controllers do not use Jackson to serialise the results into JSON, rather they use the responders to write the results in a CSV format and to add a CSV content type HTTP header along with the caching header. Unison provides annotations for CSV headers, which enable the recursive creation of headers from annotated classes, rather than using static header methods.

13. Annotations in Java require that the values of annotation attributes be constant expressions. This prevents the use of entity classes for determining table names when using Spring query annotations.

Practice	Technology	Benefit	Caveat
Static method usage	Spring	Decoupling	Framework dependencies
New keyword usage	Spring	Decoupling	Framework dependencies
Service locator pattern	Spring	Reduces object dependencies	Framework dependencies
Singleton pattern	Spring	Mocking	Framework dependencies
Constructor work	Java	Isolation	Constraint checking
Extends keyword usage	Java	Encapsulation	Abstract classes
Getter method usage	Java	Encapsulation	Open systems
Mutator usage	Java	Robustness and parallelism	Large entities
Child set function invocation	React	Decoupling	Call-back propagation

TABLE 1
Practices and patterns to avoid.

Practice	Technology	Benefit	Caveat
LoP	Spring	Modularity	Factories
Enumeration injection	Spring	Decoupling	Leaf classes
Single constructor usage	Java	Cohesion	JPA
Constructor injection	Spring	Robustness	JUnit
Direct dependency usage	Java	Reduces object dependencies	Factories
Optional pattern	Java	Reduces indeterminacy	Optional arguments
LoPJS	JavaScript	Modularity	Factories
LoPR	React	JSX	Factories
Component closure factories	React	Encapsulation	Direct JSX
Hypermedia	HATEOAS	Decoupling	Indirection
Caching	HTTP	Performance	Stale content
Content negotiation	HTTP	URI consistency	Anchor tags

TABLE 2
Practices and patterns to adopt.

The headers are injected into the responders and are written to the response when requests are received.

Controllers are provided that enable geospatial and user data to be managed. Content negotiation is used to enable the returned geo-spatial data to be obtained in either a GeoJSON or HAL format (see Appendix G) and a deserialiser enables data to be received in a GeoJSON format[14].

Weather links for HATEOAS are implemented as an enumeration and the array is injected, rather than accessed statically. One reason for this is that Met Éireann and Norwegian Meteorological Institute HARMONIE-AROME endpoints support different weather variables[15]. It also makes it easier to isolate functionality for unit testing. Unison avoids the use of static code in the application[16] and the new keyword is only used in bean definitions for the most part.

In terms of debugging, loggers are injected using the prototype bean scope rather than having static boilerplate logging code strewn throughout the codebase.

Although developers often add accessor methods to JPA entity classes, accessor methods are not a requirement of JPA or Hibernate and Unison eschews them in adhering to the LoP. In Section 4.9, it was stated that classes should only have a single constructor. With JPA, a requirement is that entity classes have a zero-argument constructor. It is possible to create a zero-argument constructor that does nothing (see Section 4.8) in order to get around this issue and such constructors can be considered an exception to the argument given in Section 4.9. Even though the single argument constructor is used by JPA in creating object instances, there is no need to create mutators (see Section 4.14) for setting the attributes in that the ORM will assign the attributes automatically.

There is an impedance mismatch problem between JPA and constructor injection. JPA creates previously stored instances of entity classes using data stored in the database. If the entity classes have dependencies on injected components and it is not desirable to store the injected component, such as a repository component, the attribute will be made transient using the transient annotation. Given that the attribute is transient, it will not be created by JPA from stored data. JPA instantiates the instance using the zero-argument constructor. It does not inject dependencies using constructor injection. It is possible to inject dependencies using field injection, but field injection should be avoided (see Section 4.10). An alternative to having the injected component as an attribute of the entity, would be to pass the injected component as an argument to the methods of the entity that have dependencies on it. The problem with this, however, is that if the entity is embedded or an attribute of another entity, this leads to the situation whereby classes have methods that only propagate method calls, as discussed in Section 4.11. The latter alternative, which is adopted in Unison, is preferable in that it is the lesser of two evils in that it does not lead to the testing problems discussed in Section 4.10 or introduce the need for an injection framework to be used in testing the entity.

6.3 React Application Implementation

The Unison React code is compliant with the LoPR (see Section 5.1) and uses component closure factories to avoid the prop drilling problem (see Section 5.2). The initial version of the code was developed using React class-based components, but it has since been redeveloped to use React hooks. Hooks substantially simplify the development of

14. The deserialiser enables a GeoJSON object to be PUT to the server and the controller will receive a Java object representing the data.

15. Met Éireann support global radiation, but the not fog and vice version for the Norwegian Meteorological Institute.

16. Static code is used in the test suite, but test suite is procedural (see Appendix F).

React components. One of the reasons for this is that the this keyword in JavaScript is quite a bit more complicated and difficult to understand than in other languages. The use of classes can make it difficult to perform ahead-of-time compiler optimisations. Additionally, class-based components make it difficult to reuse stateful logic between components. This can lead to the need to restructure the codebase and adopt patterns that change the component hierarchy. With hooks, components can be structured better and functionality does not need to be split based on lifecycle methods.

When the Unison React application begins operating, it initially requests a HAL model of the API (see Appendix G) from the server. Using links from the HAL model, Unison requests location data in a HAL representation and then in a GeoJSON representation. The HAL representation is used to determine what weather variables are associated with the locations and includes links for querying the variables. The GeoJSON data is used for displaying icons for the locations on a map.

React manages the state of the query parameters (from date, to date, and weather variable). When the user clicks on icon on the map, a request is asynchronously made to the server using a HAL link and the resultant data is displayed to the user on a graph. If any of the state variables change, a new request is made automatically, and the user interface is updated. HAL links are also used when adding and deleting locations. The use of hypermedia makes the application dynamic and adaptive to additions to the API in that the client discovers the API functionality.

Adhering to the LoPR and using props object destructuring makes components more declarative with regard to their dependencies. In Unison, component closure factories are declared in the same file as the component. This increases cohesion in that if the dependencies of a component change, the developer does not need make changes to a different file for the factory.

7 Conclusion

In an analogous manner to the principle of least or stationary action in physics, the principle of least knowledge is pivotal to the development of software that is both maintainable and testable. In its original formulation, as the LoD, the principle of least knowledge was applied to the development of object-oriented systems, but the LoD does not consider non-pure object-oriented systems that contain primitive variables. This paper modifies and extends the LoD to the 'Law of Persephone' (LoP), which addresses this. Moreover, it proposes adopting the principle of least knowledge for JavaScript and React development and introduces two extensions to the LoP, namely the LoP for JavaScript (LoPJS) and the LoPJS extension - the LoP for React (LoPR).

Extending the principle of least knowledge to non-pure object-oriented systems, JavaScript, and React is important in that these technologies are widely used in industry. The LoP differs from the LoD in a number of ways. It prohibits the use of static methods, primitive attributes set with mutators, and primitive variables returned by the getters of objects pre-existing to the invocation of the calling methods. This increases encapsulation and decoupling and enables different parts of the codebase to be isolated, making it more amenable to testing. The benefits of testing in the development of large systems are well-known and adhering to the practices and patterns discussed in this article will increase the testability of the software.

In addition to the principle of least knowledge, a number of other practices that should be adopted were discussed. In developing Java application code, the use of the new keyword and work in the constructor should be avoided, the arguments of constructors and methods should only be what is directly required, and the optional pattern should be used to reduce indeterminacy. Additionally, the paper discussed practices specific to dependency injection that should be adopted. In terms of React development, the paper introduced the use of component closure factories, which address the prop drilling problem without making state global.

The application of these development principles is considered in the context of a case study on Unison, an open-source application for the tracking and graphing of (HARMONIE-AROME) numerical weather forecast data. Unison uses an ORM at the interface between the object-oriented components and the geospatial relational database. Although the ORM introduced an impedance mismatch problem, it was not found to be a significant issue. The Unison UI is delivered through React and the frontend discovers the API via hypermedia. It could be argued that the indirection introduced through the use of hypermedia reduces the performance of the system in terms of latency and increases the complexity of the client when compared to the approach of using hard-coding URIs. Unison, however, favours the adaptivity that hypermedia delivers and the avoidance of integration problems for third party applications that use the API when changes are made, or new functionality is added, to the API. The use of hypermedia can often offset the performance costs by enabling more aggressive caching for various resources at the front-end. Future work will investigate returning JSON objects that have methods from the API to enable the delivery of more adaptive code on demand.

Acknowledgments

This research (Acclimatize) was part funded by the European Regional Development Fund through the Ireland Wales Cooperation programme.

References

[1] N. Jain, A. Bhansali, and D. Mehta, "Angularjs: A modern mvc framework in javascript," *Journal of Global Research in Computer Science*, vol. 5, no. 12, pp. 17–23, 2015.
[2] C. Staff, "React: Facebook's functional turn on writing javascript," *Communications of the ACM*, vol. 59, no. 12, pp. 56–62, 2016.
[3] B. Nelson, *Getting to Know Vue. js*. Springer, 2018.
[4] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, and D. Kopylenko, "Professional java development with the spring framework," 2005.
[5] K. Lieberherr, I. Holland, and A. Riel, "Object-oriented programming: An objective sense of style," *ACM Sigplan Notices*, vol. 23, no. 11, pp. 323–334, 1988.
[6] A. Kay, "Why is functional programming seen as the opposite of OOP rather than an addition to it?" https://www.quora.com/Why-is-functional-programming-seen-as-the-opposite-of-OOP-rather-than-an-addition-to-it/answer/Alan-Kay-11, 2018 (retrieved March 24, 2020).

[7] M. Madsen, O. Lhoták, and F. Tip, "A semantics for the essence of react," in *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
[8] Y. Jayawardana, R. Fernando, G. Jayawardena, D. Weerasooriya, and I. Perera, "A full stack microservices framework with business modelling," in *2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer)*. IEEE, 2018, pp. 78–85.
[9] WSO2, "Microservices framework for java," https://github.com/wso2/msf4j, retrieved January 15, 2021.
[10] R. Vanbrabant, *Google Guice: agile lightweight dependency injection framework*. APress, 2008.
[11] Google, "Guice," https://github.com/google/guice, retrieved February 23, 2020.
[12] M. Hevery, "The clean code talks - don't look for things!" https://www.youtube.com/watch?v=RlfLCWKxHJ0, 2008.
[13] ——, "Oo design for testability," https://www.youtube.com/watch?v=acjvKJiOvXw, 2009.
[14] C. Muldoon, G. M. O'Hare, R. Collier, and M. J. O'Grady, "Agent factory micro edition: A framework for ambient applications," in *International Conference on Computational Science*. Springer, 2006, pp. 727–734.
[15] C. Muldoon, "An agent framework for ubiquitous services," Ph.D. dissertation, Computer Science, University College Dublin, 2007.
[16] C. Muldoon, G. P. O'Hare, R. W. Collier, and M. O'Grady, "Towards pervasive intelligence: Reflections on the evolution of the agent factory framework," in *Multi-Agent Programming*. Springer, 2009, pp. 187–212.
[17] C. Muldoon, G. M. O'Hare, M. J. O'Grady, and R. Tynan, "Agent migration and communication in wsns," in *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 2008, pp. 425–430.
[18] A. Kay, "Programming and scaling," https://www.youtube.com/watch?v=YyIQKBzIuBY, 2009 (retrieved February 23, 2020).
[19] R. S. Barton, "A new approach to the functional design of a digital computer," *Annals of the History of Computing*, vol. 9, no. 1, pp. 11–15, 1987.
[20] Oxford English Dictionary, "Object-oriented," https://en.oxforddictionaries.com/definition/object-oriented , retrieved February 23, 2020.
[21] A. Holub, "Why getter and setter methods are evil," *JavaWorld*, 2003.
[22] T. Hoare, "Null references: The billion dollar mistake," *Presentation at QCon London*, vol. 298, p. 88, 2009.
[23] D. Abramov, "The history of react and flux with dan abramov," https://threedevsandamaybe.com/the-history-of-react-and-flux-with-dan-abramov/, 2015.
[24] L. Bengtsson, U. Andrae, T. Aspelien, Y. Batrak, J. Calvo, W. de Rooy, E. Gleeson, B. Hansen-Sass, M. Homleid, M. Hortal *et al.*, "The harmonie–arome model configuration in the aladin–hirlam nwp system," *Monthly Weather Review*, vol. 145, no. 5, pp. 1919–1935, 2017.
[25] C. Muldoon, "Unison," https://github.com/conormuldoon /unison, 2021.
[26] Acclimatize, "Acclimatize project," https://www.acclimatize.eu/, 2021.
[27] D. Thomas and A. Hunt, *The Pragmatic Programmer: your journey to mastery*. Addison-Wesley Professional, 2019.
[28] A. Bruns, A. Kornstadt, and D. Wichmann, "Web application tests with selenium," *IEEE Software*, vol. 26, no. 5, pp. 88–91, 2009.
[29] Airbnb, "Enzyme," https://enzymejs.github.io/enzyme/, 2021 (retrieved August 6, 2021).
[30] K. Dodds, "Why i never use shallow rendering," https://kentcdodds.com/blog/why-i-never-use-shallow-rendering, 2018.
[31] C. I. Higgins, J. Williams, D. G. Leibovici, I. Simonis, M. J. Davis, C. Muldoon, P. van Genuchten, G. O'Hare, and S. Wiemann, "Citizen observatory web (cobweb): A generic infrastructure platform to facilitate the collection of citizen science data for environmental monitoring," *International Journal of Spatial Data Infrastructures Research*, vol. 11, 2016.
[32] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.
[33] I. NginX, "If is evil... when used in location context," https://www.nginx.com/resources/wiki/start/topics/depth /ifisevil/, 2021 (retrieved February 23, 2020).

Conor Muldoon Dr. Conor Muldoon is a research scientist in the School of Computer Science at UCD with research interests in the areas of Sensor Networks, Multi-Agent Systems, Distributed Artificial Intelligence, and Ubiquitous Computing. He has held two prestigious Irish Research Council for Science, Engineering, and Technology (IRCSET) fellowships, namely the INSPIRE Marie Curie International Mobility Fellowship and the Embark Fellowship. He holds a Ph.D. in Computer Science and a B.Sc. (Honours) degree in Computer and Software Engineering.

Levent Görgü Dr. Levent Görgü is a research scientist in the School of Computer Science at UCD. He holds a Ph.D. in Computer Science, an M.Sc. in Ubiquitous and Multimedia Systems, and a B.Sc. in Computer Engineering. He research interests include Wireless Sensor Networks and Ubiquitous Computing. He was previously an assistant professor at the Cyprus International University where he taught courses on advanced database management systems, advanced programming, and visual programming.

John J. O'Sullivan Born and brought up in Dublin, I studied at Trinity College Dublin from where I graduated with a Civil Engineering degree in 1993. I completed an M.Sc. at the Queen's University of Belfast in 1994 and following this started a seven-year period that would involve working in a research capacity at the University of Ulster on a number of externally funded projects. I completed my Ph.D. in 1999 after spending time at the UK Flood Channel Facility in HR Wallingford and have also spent time at the University of the Witwatersrand, South Africa and at the University of Adelaide, Australia. Present research interests are concerned with water resources.

Wim G. Meijer Prof. Wim Meijer is Professor of Microbiology and Head of the School of Biomolecular and Biomedical Science at UCD. His research team works in two thematic areas: water quality and human/animal health. His water themed research focuses on water quality of bathing waters and rivers, in particular in relation to faecal contamination, pathogens and antimicrobial resistance, working closely with colleagues in other disciplines, with local authorities and national regulatory bodies. Animal health research is mainly focused on the Rhodococcus equi, a multi-host pathogen infecting phagocytic cells, and the role of microbiota in uterine health in relation to the development of endometritis.

Bartholomew Masterson Dr. Bat Masterson was formerly a Senior Lecturer in Biochemistry at UCD. He is a member of the International Water Association and has had a long-standing involvement in research on the microbial pollution of recreational waters, including the SMART, iCREW and Smart Coasts = Sustainable Communities INTEREG funded projects and projects funded by the Irish EPA STRIVE programme. He is currently active in two further Interreg funded projects, Acclimatize and SWIM.

Gregory M. P. O'Hare Prof. Gregory O'Hare is Professor of Artificial Intelligence and Head of School of Computer Science and Statistics at Trinity College Dublin and a visiting Professor at UCD. Prof. O'Hare has over 500 refereed publications of which over 100 are in high impact journals. He has edited 10 books and has a cumulative career research grant income of circa €82 Million. His research interests are in the areas of Artificial Intelligence and Multi-Agent Systems (MAS), Ubiquitous Computing and Wireless Sensor Networks. He is an established Principal Investigator with Science Foundation Ireland, having been one of the founders of the CLARITY centre (now INSIGHT) and the very successful CONSUS collaboration with Origin Enterprises in the area of Digital Agriculture.

Appendix A
Spring

Spring was originally developed to make it easier to develop enterprise Java applications without the need to use the Enterprise JavaBeans (EJB) architecture, which was originally a poorly designed technology that was difficult to work with and had performance issues. For instance, in the original specification, only remote method invocation was supported, even though a large amount of back-end functionality does not require distributed computing functionality. In EJB 3.0, which was influenced by Spring, a number of shortcomings were addressed; EJB 3.0 uses Plain Old Java Objects (POJOs) and supports dependency injection.

Spring comprises an IoC container that enables dependency injection. This facilitates the effective unit testing of application code. The AOP framework, when used with dependency injection, enables a POJO programming model whereby application code has minimal dependencies on the Spring APIs, but can avail of Spring services declaratively using annotations. This approach is adopted, for instance, for transaction management. AOP increases modularity by enabling the separation of cross-cutting concerns. This is achieved by adding additional functionality without modifying the application code itself.

As noted in the introduction, Spring Boot is Spring's convention over configuration solution that takes an opinionated view of how the Spring framework and third-party libraries should be used. It makes it easy to get started using the framework and create stand-along Spring applications with embedded servers. With Spring Boot, Spring is automatically configured where possible. In addition to Spring Boot, Unison, the case study application discussed in Section 6, makes use of the Spring Security and Spring HATEOAS (Hypermedia as the Engine of Application State) projects to provide support for (1) HTTP security and fine-grained method level security and (2) RESTful hypermedia and API discovery.

Appendix B
React

Traditionally, one of the most difficult aspects of developing JavaScript front-end browser applications has been dealing with the DOM, which is poorly designed from an API perspective and is subject to cross-browser inconsistencies. React was developed to address these issues and to improve the developer experience through the use of a virtual Document Object Model (DOM), which enables developers to avoid directly interacting with the browser DOM whilst, also, somewhat surprisingly, improving the performance [2].

React was created by Facebook. It is declarative and component-based and was initially deployed in production for the "liking and commenting" interface on the Facebook News Feed; it has since become one of the most popular SPA frameworks. React is functional in nature and has been described as a "referentially transparent UI" [2]. Typically, the UI is a pure function of some set of inputs that produces the same type of DOM structure for a given set of data. When the data changes, the current rendering is preserved, a rendering is performed using the changed input, and the difference between the new rendering and old rendering is used to determine the parts of the page that get updated. With React, the user interface code is viewed as a black box, which accepts arbitrary data or parameters known as props. This makes it flexible with regard to the structure of the data supported. Components in React are quite similar to JavaScript functions, the difference being that they need to be aware of their own lifecycle events, such as when they are added or removed from the DOM.

Appendix C
Object Relational Mapping

Object-Relational Management (ORM) tools enable the mapping of data from object-oriented to Relational Database Management System (RDBMS) representations and vice versa in order to persist and retrieve object state. There are several reasons for storing data in RDBMSs, such as reducing redundancy, improving query performance, and enforcing data integrity constraints. There are impedance mismatch problems, however, with regard to object-oriented representations and relational representations, which make it difficult, or increase the development effort, in making the systems work well together. This is, partly, due to object-oriented systems being interconnected graphs of objects and relational databases being normalized disjoint tabular systems.

The Java Persistence API (JPA) is a specification that enables developers to specify which objects should persist and how those objects should persist. It is expressly *not* a tool or framework, rather it is a set of concepts which relate to the API, the Java Persistence Query Language (JPQL), and the object-relational model metadata or annotations. The specification can be viewed as a standard that can be implemented by a variety of frameworks. It was originally based on the Hibernate framework. Hibernate is the most popular ORM implementation used within the Java community and, in particular, with Spring developers. It addresses five distinct object-relational impedance mismatch issues:

1) Granularity: In object-oriented systems, it is typical to have coarse grain aggregate classes that have several layers of granularity in terms of the extent to which the class is broken down into smaller parts. A person class, for instance, could contain an address class, a medical history class, and a set class of friends that are of class person. A relational model has only two levels of granularity, namely tables and columns.
2) Implementation Inheritance: Java supports implementation inheritance (see Section 4.12). RDBMSs do not provide a similar capability. Some RDBMSs support subtypes, but it is not standardised across systems.
3) Identity: The Java language has two forms of identity, namely object identity a == b and object equality a.equals(b). In contrast, RDBMSs have only a single form of identity, which is defined using a primary key.
4) Associations: Object associations in Java are unidirectional. To create a bidirectional association between two objects, both objects must reference each

other. Within RDBMSs, associations are defined in terms of foreign keys. Additionally, with object-oriented development, it can be difficult to determine, by looking at the object model, the multiplicity relationships between objects. That is, whether they have a one-to-one relationship, a one-to-many relationship, a many-to-one relationship, or a many-to-many relationship.

5) Navigation: Within object-oriented development, navigation from one object to another occurs by traversing references through the object network. Attempting to mimic this type of traversal would not be an efficient way of retrieving data from a RDBMS. When using RDBMS within object-oriented systems, it is more efficient to minimise the number of SQL statements by combining tables through joins and specifying the targeted entities a priori.

There are a number of disadvantages to using ORMs and these will be more or less pronounced depending on whether the developer wishes to threat the object model as primary and the persistence layer incidental or vice versa.

One of the key objections to ORMs is in relation to data access performance. With ORMs, there is no way to be certain that the generated queries will be optimal. To optimise database access, it is necessary to structure the database carefully and develop hand-crafted queries. Another objection is that the generated queries can be difficult to fine tune, understand, and debug. A third objection is that ORMs have slower start-up times in that they require metadata preparation.

On the positive side, ORMs provide functionality for cache management whereby frequently accessed entities are cached in memory, thereby reducing the load on the database. Another advantage is that they reduce the amount of code required to access the database and reduce development time if the types of queries required by the system are relatively simple. A third advantage is that they hide the vagaries of vendor specific code from the application. A corollary of this is that the application is decoupled from the RDBMS and the impact of switching vendors is reduced.

Appendix D
Only use default methods for backward compatibility

In Java 8, default methods on interfaces were introduced to enable new functionality to be added to interfaces and maintain binary compatibility with software written for older versions of the interfaces. The reason they were introduced is likely to have been to support the evolution of the Collection interface by providing methods that support lambda expressions. In the original development of the Java language, a design decision was made not to support multiple inheritance for classes. The reason for this was due to ambiguities caused by the diamond inheritance problem.

Suppose you have four classes A, B, C, and D. Classes B and C extend A and class D extends both B and C. The diamond problem of inheritance occurs when B and C override the same method of A. D, in this case, is unable to determine whether to inherit the method from B or C.

Multiple inheritance is possible with Java interfaces and the introduction of default methods on interfaces introduced the diamond problem to the language, but the code will not compile when it occurs. As such, default methods should be used sparingly and only for backward compatibility when no other solution is possible.

Appendix E
Returning closure factories

As discussed in Section 5.2, the function returned by a function closure factory can itself also be a closure factory. This enables the pattern to be used in situations when a component has dependencies on the state of 3 or more other components. The following is an example of such a component closure factory and whereby more than one argument is passed to some of the functions.

```
function createMyComponentFactory(a, b){

return function (c, d, e) {

return function (f){

return <MyComponent a={a} b={b} c={c}
d={d} e={e} f={f} />

}
}
}
```

To give an example of the use of the code above, consider a component, A that creates a closure using arguments a and b and then passes the closure to another component, B. B then creates a closure by calling the closure it received using arguments c, d, e and passes the created closure to another component, C. C calls the closure it received using argument f to create MyComponent.

Appendix F
Testing

Test code is in effect procedural. When writing test code, thus, many of the principles discussed in sections 4 and 5 do not apply. Although good design practice should be adopted, such as the Don’t Repeat Yourself (DRY) principle [27], when writing test code, there is a difference to writing test code and the code for the system under test. The test code itself does not need to be testable. The need to test the tests would lead to infinite regress. Indeed, it is not possible to adopt many coding practices for testability, such as using constructor injection (see Section 4.10), when using testing frameworks, such as JUnit, but this is not of consequence. When using JUnit, classes can only be created using a zero-argument constructor. This is necessary as it is intended to ensure separate tests are isolated from each other.

With React, if a component is rendered using the same props and same state, the same user interface will be rendered. Excluding cases, such as when using random number generators or state related to the date or time, if components are pure functions of state and props, the testing process with React is quite fast and easy. That is, tests can be easily developed to ensure that the user interface rendered in a specific way when a component receives a given set of data. This is in contrast to the WebDriver approach whereby the correct sequence of user interface interaction events must be triggered in parent nodes, such as pressing a series of buttons in a particular order or entering information in a form, to ensure that the interface display correctly.

Appendices F.1 to F.4 discuss general testing concerns. This is followed in appendices F.5 and F.6 with a dicussion of snapshot testing and shallow rendering, which are mainly related to React testing.

F.1 Mocking

In this article, we advocate using the Mockito Library for testing Java code. There are several benefits to using Mockito, such as avoiding the need to handwrite mocks. In React, JavaScript Testing framework (Jest) mocks, mock functions, and mock spies should be used in addition to the fetch-mock package, which mocks network requests. Mocking libraries enable more than creating stubs to facilitate unit tests. They enable the behaviour of the code to be verified. For example, the test developer may wish to determine whether a method or function on the mock was called, the order in which methods are called, and so forth.

Mocks are particularly useful in developing tests for the front-end. The React Testing Library can be used to fire events in the interface, which enables user interaction with the system to be mimicked by the test code. Jest mock functions or spies can be used to mock what would be user interaction with the system, such as clicking on a popup. This is more robust and less sensitive to changes in the user interface and runs faster than simulating user interaction with a tool, such as Selenium [28].

F.2 Don't repeat yourself

Although tests are written in a procedural manner, this does not mean no software engineering practices should be adopted when writing them. For instance, the don't repeat yourself principle should be adopted with regard to functional reuse. Given that test code itself is not tested, in Java, we advocate developing static utility classes that capture common functionality between tests in addition to having class specific methods that are used by JUnit test methods. In React, similarly, functions should be developed that capture common test functionality both within and among Jest test modules.

F.3 Mutation testing

Coverage is often used as a metric in establishing how well software has been tested, but it is a somewhat blunt instrument. 100% coverage simply means that all lines of code have been executed at least once. It can provide false confidence that the tests will catch all bugs. Complete testing would require that all possible inputs to the code would be tested through all possible branches or paths. Clearly, this is an intractable problem.

Coverage testing says nothing about the quality of the tests being conducted. One way to determine if a test will detect a bug is to change the code. Mutation testing is a form of white-box testing and is used to evaluate the quality of existing tests and to design new tests. With mutation testing, the code is modified in small ways. The modified version of the code is referred to as a mutant. The mutants are determined by mutation operations that are based on frequent programming errors. If the test suite detects errors introduced by the mutant, this is referred to as killing the mutant. The test suite performance is evaluated with regard to the number of mutants killed. The goals of mutation testing are as follows:

- Determine poorly tested pieces of code.
- Determine the quality of existing tests.
- Calculate the percentage of mutants killed.
- Elucidate error propagation within the codebase.

F.4 The cost of testing

100% test coverage can provide a false sense of security in that it does not imply that code will work for all possible inputs and through all possible paths. It implies simply that all lines of code will be covered by at least one test. Truly verifying that code will work under all circumstances is an intractable problem. As mentioned in Section F.3, coverage testing does not indicate how good the tests are or how good they will be at detecting bugs in the code. It takes time to develop tests and that effort should be directed at where is makes the most impact.

One of the drawbacks of testing is that it increases that amount of work required to be undertaken when changes are made to the system. Ideally, tests should test behaviour rather than internal implementation details, so this should be minimal, but when APIs change, the tests must be rewritten. It is for this reason this paper does not advocate for 100% test coverage, particularly not at early stages of a project where much of the code is experimental. It is not just that the law of diminishing returns comes into play when too much testing is done, but that the tests can result in negative returns in that they affect the structure of the software and increase development time if the coverage level is to be maintained. A more elegant and modular architecture and design will be possible if testing is done where it is required, but eschewed in places where it is not critical. *Over-testing* can increase the complexity and reduce the modularity or the degree of information hiding of the code.

As in many situations where there is a trade-off between costs and benefits, in developing test code there will often be an approximate Pareto distribution whereby 80% of the benefits are accrued from 20% of the tests developed. When test coverage is low, the most important areas of the code for testing should be identified, prioritized, and covered first. A common rule of thumb is to aim for 70% to 80% test coverage. This will vary with regard to the mission critical aspects of the codebase and the risk with regard to getting things wrong.

The costs and benefits of testing should also be considered with regard to the ease to which errors can be corrected once the software has been deployed. In developing mobile phone applications, for example, applications need to be verified by application stores prior to being released, which reduces the speed at which errors can be corrected. Additionally, the users will only receive patches provided they update their applications. This increases the need or utility for applications and application upgrades to be comprehensively tested prior to shipping. A similar situation occurs with web development depending on the caching strategy adopted. If it is possible for users to have old versions of React applications cached in their browsers for a long period of time, the utility of testing increases in that they won't receive updated code rapidly. The benefits of caching of course are that it reduces the load on the server and latency.

F.5 Snapshot testing

Snapshot testing is another approach that reduces the need to simulate user interaction with the GUI. Snapshot testing is useful in situations where you wish to ensure that no unexpected changes in the interface occur. This requires decoupling persistent code from code that changes, one such example might be an interface component that displays the date. With snapshot testing a snapshot is taken of the components of the interface that would be rendered in the browser and compares the current snapshot to a previous snapshot. If they don't match, the test fails. Snapshot testing should not be overused, however, as if the tests are constantly failing when any change is made, the developer will ignore the failures and automatically update the snapshots instead.

F.6 Don't use shallow rendering

Enzyme [29] is a popular testing library developed by Airbnb that is frequently used in conjunction with Jest in testing React components. Enzyme supports a feature for testing components that is referred to as shallow rendering. Shallow rendering is intended to enable unit testing on components and to ensure that test assertions are not testing the behaviour of child components. When performing shallow rendering, Enzyme calls React lifecycle hooks related to component mounting and updating along with the component's render method. With shallow rendering, Enzyme uses the result of the render method in creating a wrapper object with associate methods for traversal. It is not possible to interact with DOM elements as nothing is actually rendered. The behaviour of the component in a real browser is not fully tested.

The reason shallow rendering is used include the fact that it enables methods in React components to be called, it is more efficient to avoid rendering all child components for every unit test, and testing composed components might trigger an error from a child component whereas the unit may have worked correctly. The problem with this, however, is that the test code becomes coupled to the internal implementation details of the component [30]. There are alternative approaches to performing unit testing components, such as mocking the props. Those approaches avoid the pitfalls of shallow rendering.

Appendix G
Unison API

The Unison API is an instance of the REpresentational State Transfer (REST) architectural style. To be considered a RESTful architecture, Internet standards should be adopted, and the system should provide a uniform interface and have the properties of being stateless and cacheable. Layering must be possible and a client-server model must be adopted. With layering a client cannot determine if they are interacting directly with the end server or with an intermediary, such as a proxy or load balancer. One of the distinguishing aspects of REST is the use of hypermedia. Unison uses HATEOAS and delivers hypermedia in the Hypertext Application Language (HAL) format. HATEOAS and HAL facilitate the discovery of APIs by clients. This enables the evolution of server code independently of the client. The delay for changes to reach the client-side code can be significant if aggressive caching strategies are adopted or mobile applications are being used that have not been updated or are awaiting approval by application stores (see Appendix F.4). HATEOAS alleviates some of these issues through the use of indirection. Rather than hard-coding URIs into application code, REST clients discover the API and available URIs/resources.

In addition to being RESTful, the Unison API supports the GeoJSON standard format for representing the collection of locations for which weather data is being recorded. There is an issue, however, in that HAL and GeoJSON are incompatible. To get around this problem, an early version of Unison took the approach of embedding links in the properties of the GeoJSON objects. This was similar to the approach adopted within the Citizen OBservatory WEB (COBWEB) infrastructure [31]. The problem with this approach, however, is that although the GeoJSON is correctly formatted, the resource representation is not consistent with the HAL convention. To address this problem, Unison makes use of content negotiation.

Content negotiation enables resources, identified using URIs, to be represented in conflicting formats. There are a number of approaches to delivering content negotiation. Unison uses server-driven content negotiation whereby the server determines the resource representation to deliver based upon the headers of the HTTP requests received. In addition to serving data in either a GeoJSON or HAL format, Unison makes use of content negotiation for representing weather data resources in either a JSON or CSV format for JavaScript library and human consumption respectively. It is also used at the API root (see Appendix G.1). One of the advantages of using content negotiation is that if, in the future, a new format is to be supported, the same URI can be used for the resource while also still supporting the pre-existing formats.

Although not explicitly part of REST, it's considered good practice in API design to have URIs that don't change or contain implementation details with regard to the resources they represent. The use of HAL and HATEOAS help in alleviating issues related to changing URIs. They resolve the issue for applications that discover the API in the correct manner, but nevertheless, links can still be broken if they are being used directly by users or clients who do not follow links through hypermedia to discover resources.

G.1 Avoiding the API prefix with Content Negotiation

In this article, we argue that including an API prefix, such as /api, in a URI is an unnecessary implementation detail.

API prefixes are often used on proxy servers to redirect requests to application servers. Its use for this purpose arguably violates the REST layering principle [32]. Often, the application server generates dynamic content and a proxy server serves static resources. Whether a resource is static or dynamic, changed from static to dynamic, or vice versa should be hidden from the client. The typical user will not be concerned with this detail.

To remove the requirement for using an API prefix on the proxy server, we advocate using content negotiation based on the accept header of the request. So, for instance, if the accept header is for application/json, application/geo+json, text/csv, or application/hal+json, the request is forwarded to the application server, otherwise the resource is served locally. It is relatively easy to implement this for testing using an if statement and the React proxy middleware, which contains an embedded Express server. For a production setting, when using NginX, it becomes more difficult in that the NginX if directive is dangerous when used in blocks, known as location contexts, which define how to handle requests for URIs [33]. In some cases, the directive does not do as one might expect, but rather does something completely different. It can even cause a segmentation fault and, as such, should be avoided where possible. The reason for these issues is that NginX configuration is mostly declarative, but the rewrite module, which the if directive is part of, evaluates instructions imperatively and NginX allows some non-rewrite directives inside if.

To get around these issues and enable conditional forwarding in a safe manner, we make use of the NginX map directive and introduce a mapped variable that will be used within location contexts. The URI of a resource is the default value for the mapped variable, but if we wish to forward the request due to its accept header, the mapped variable receives a null value. In the location context, an attempt is made to serve the resource locally using the try_files directive. If the mapped variable is not null and the resource is available locally, the resource is served. Otherwise, a named location is used to add forwarded headers for HATEOAS and forwards the requests to the application server. This process completely avoids using if and is safe.

Unison enables content negotiation for the root or index resource, so, for instance, a HTML index or a HAL representation of the root can be served. If a HAL representation is not requested, the NginX index directive is used in the usual way to determine the order of preference of local index files of different types, such as whether a HTML index file is given preference over a PHP index file, etc. An additional mapped variable was introduced to handle the root URI. This was necessary to prevent a loop in the configuration. It works in much the same way as discussed for URIs in general, but the difference being that the default value for mapped variable is / rather than the URI.

G.2 REST API Implementation

A requirement of REST is that the HTTP methods be used in the correct manner. In frameworks, such as Ruby on Rails and Spring Data REST, a one-to-one correspondence is often made between the database semantics of Create, Read, Update, and Delete (CRUD) operations to the HTTP PUT/POST, GET, PATCH, and DELETE methods, but this is not the correct use of the HTTP methods. Consider, for example, the HTTP PUT and POST methods, which both can be used for creating resources, the difference being that PUT must be idempotent and ought not be used for operations that cause side effects on the server, whereas POST need not be idempotent. Given that PUT is idempotent, the same PUT operation will always have the same result if repeated. This enables operations to be cached in the network. The CRUD database semantics do not capture this.

The semantics of transferring information over a network and the document model of the Internet are not the same as the domain model and its representation in a database. This causes an impedance mismatch problem, but that is the way it should be. Much of the focus of developing server code should be around addressing not just the impedance mismatch problem from the document model of the Internet to the domain model of the application, but also the impedance mismatch problem from the domain model to the entity-relationship model of the database.

The requirements for storing information in a database, in terms of consistency, redundancy, and normalisation do not directly match the requirements for accessing the data, in terms of bandwidth, convenience, and so forth over a network via an API. For example, in Unison, data from all weather variables, save precipitation[17], are stored in a single table to match the HARMONIE-AROME schema. The API, however, has been designed to enable weather variables to be queried individually, which is a requirement from the front-end and users of the API. To implement this using a technology, such as Spring Data REST, each variable would need to be stored in its own table. This would not be the best design from a database perspective and a transactional layer would need to be developed to ensure consistency, which would be less efficient than using a single table. Alternatively, such technologies could enable data from all columns to be received, but this would be less efficient in terms of bandwidth and more cumbersome for the client. In short, HTTP resources do not always have a one-to-one correspondence with database tables and queries should be invoked on the server to retrieve *only* the data explicitly required.

Appendix H Summary of Recommendations

Generally speaking, avoiding static code increases decoupling, making it easier to isolate different parts of codebase for testing. For this to be practical, however, a dependency injection framework is required. Although ES6 classes enable static code in JavaScript, React hooks avoid ES6 classes and are preferred.

For Java in general, avoiding work in the constructor enables object creation to be separated from the work being performed, which facilitates independent testing. Avoiding the extends keyword and getter methods increases encapsulation. Abstract classes can sometimes be useful if the protected keyword and fragile base class problem

17. The number of temporal points for forecast precipitation from HARMONIE-AROME endpoints may be different than the other variables.

are avoided and getter methods are sometimes required in open systems or at the procedural boundary, but generally delegation should be used rather than getters. Mutators reduce the robustness of the software and prevent parallelism. Although we argue that large classes should be redesigned, an impedance mismatch problem between objects and relational databases means that this might not be best from a database perspective. In such cases, the use of a setter method rather than creating a new instance might be preferred when updating a large entity and using JPA. Passing React set functions to child components should be avoided to increase decoupling except for the case where a call-back function is only propagating calls to a set function.

In terms of practices to adopt, the LoP greatly improves the modularity and testability of the software. Following the LoP will lead to the creation of a large number of factories though if a dependency injection framework is not used. The LoPJS and LoPR extend and modify the LoP for JavaScript and React. Component closure factories increase reuse in terms of creating components. One of the drawbacks, however, is that creating components in this way requires calling JavaScript functions rather than embedding the JSX directly. Component closure factories need only be used for components that rely on the state of components at different levels of the component tree.

Enumeration injection increases decoupling, but, again, requires a dependency injection framework for it to be pragmatic. Using a single constructor with no optional arguments increases cohesion and reduces the likelihood that classes are addressing multiple concerns. JPA requires a single zero-argument constructor, but an empty constructor that does nothing can be provided and considered an exception. Constructor injection improves the robustness of the software when compared field or setter method injection. As is the case with many of the guidelines discussed here, an exception can be made for writing test code and the use of JUnit (see Appendix F). In a similar manner to avoiding the service locator pattern, only passing direct dependencies to the constructor or methods reduces the extent of the object-dependency graph of the class. The problem with this again though is that it can lead to the development of a lot of factories. We argue for returning optionals to avoid the indeterminacy between null and objects in situations where new objects may be created. We do not recommend passing optionals as arguments to constructors or methods.

Unison adopts a RESTful architecture and makes use of hypermedia to enable the client to discover links on the server. This makes the client more adaptive to a changing API. The disadvantage though is that it introduces indirection and increases the complexity of the client. HTTP caching can greatly improve the performance of applications in terms of reducing latency and bandwidth requirements. With caching, content can become stale, however, and there is no guarantee that a cache will be hit other than the browser cache. This paper advocates using content negotiation, which ensures URI consistency and that, as new formats become available, the URI for the resource does not need to be changed. One downside to content negotiation is that the HTML anchor tag does not enable the accept header to be specified, which means a link can only be made to one format unless created programmatically using JavaScript.