International Journal of Software Engineering and Its Applications
Vol.8, No.1 (2014), pp.95-106
http://dx.doi.org/10.14257/ijseia.2014.8.1.09

# Input and Output Application Programming Interface for Plaid Standard Library

Nazri Kama

*Advanced Informatics Schoool*
*Universiti Teknologi Malaysia*
*nazrikama@ic.utm.my*

***Abstract***

*Object oriented (OO) is the most accepted object modeling languages nowadays. Although OO languages are successful in modeling real world entities, nevertheless, there is little to no support for modeling state of object precisely in code. To overcome this problem, a new general purpose programming language is proposed that is built on two paradigms which are typestate-oriented and permission-based. One of the most vital factor for every language to be accepted and effective for users is having an enhanced standard library and clearly one of the most important parts is Input and Output (I/O) Application Programming Interface (API). One of a new programming language is Plaid language. Since the Plaid is at its early stage of development, I/O features provided by the Plaid standard library are very basic and limited. This paper presents I/O API that we have developed for the Plaid based on the features provided by Java I/O API. For the evaluation we used two metrics which are the number of features provided by I/O API, and the number of exceptions eliminated by using Plaid and typestate-oriented programming language.*

***Keywords:*** *Plaid Programming, Application Programming Interface*

## 1. Introduction

Plaid is a new general purpose object-oriented (OO) programming language under development. It is built on two paradigms: typestate-oriented (TO) and permission-based [1,2]. In typestate-oriented paradigm, programmers can extent object-oriented programming with typestates. They treat typestate as a class, which has its own representation, interface, and behavior. However, an instance of a typestate can change its typestate during the runtime. Basically, programmers can encode the abstract states and change them using state change operator which results in different representation, interface, and behavior of objects at runtime [3]. In a permission-based programming language, programmers indicate how an object might be aliased and how the aliases mutate the object. That means for each reference to an object, the type declaration compose of two parts: (1) a type which specify object's typestate and an access permission which verbalizes how the reference can be consumed [4] and; (2) specify the permissions to other aliased of the same object [5].

The last version of the Plaid standard library consists of three packages: plaid.collections for working with collections, plaid.io for reading and writing data, and plaid.lang containing the language's core elements such as primitive types. Both packages for collections and language are in suitable conditions, whereas the package for I/O is not. It only supports some basic features for working with file system and reading from and writing to files. Within this research, we have developed an I/O API for Plaid which provides almost all features provided by java.io package in Java class library. Those features of java.io package which are to be

ISSN: 1738-9984 IJSEIA

included in the Plaid I/O API are listed in the project's Plaid Specification Request (PSR for short) documents. In this research, we have designed and implemented an I/O API for the Plaid standard library (version 0.4.0) based on java.io package's features.

This paper is laid out as: Section 2 presents related work. Then, Section 3 describes the design and implementation of I/O. Later, Section 4 and Section 5 explain the testing and results of the testing. Finally, conclusion and future work are presented.

## 2. Related Work

In this section, a concept of typestate-oriented programming is discussed whereas later a concise look is taken on previous work has been done in this area.

### 2.1. Typestate-Oriented Programming

In object-oriented programming [6], to work with an object, the developer needs to know two type of information which are : (1) what methods are ever permitted on the object (a file object, for example provides only open(), close(), read(), and write() methods), which is expressed by the interface of object; and (2) how they can be used in terms of sequence of calls (again for a file object, it is not possible to call read() or write()after calling close() method). In other words, which methods of all provided by the type can be called in the current context or in technical terms, in the current state. These two expressions for the second type of information are equivalent, since every single statement in codes, which normally is calling a method will cause a change in the state of machine (context), even though we do not care about most of them. For the second type of information, if such exists, it has been said that the type (object) has usage protocol (protocol for short) that programmers must follow in order for the type (object) to work properly.

Fortunately, in the languages which are strongly-typed (that can statically check type-correctness), violating the object interface and calling a method which is not provided by the object, is not a problem, because it leads to compile-time error and the programmer simply becomes aware of this violation. In this way a type-checker detects the violations independent of the context of an operation relative to other operations. That is, no one needs to be worry about their defined interface violation, even though programmers who use their type have not read its documentation. It is made possible by the feature of object-oriented programming which let programmers express explicitly the interface of object in code rather than in a separate documentation. In other words, protocols essentially define legal sequences of method calls.

Unfortunately, such feature is not provided by the conventional object-oriented languages that let programmers know about the usage protocol violations before runtime. Developers have three ways of finding out about protocols: reading informal documentation (which results in too much effort needed for writing and reading documents), receiving runtime exceptions that indicate protocol violations, or observing incorrect (nonsensical) program behavior as a result of protocol violations that broke internal invariants [7]. Considering that "nonsensical" executions can result in arbitrary amount of damage and also they are the most difficult to debug, the third situation is the worst scenario [8].

An empirical study on almost two million lines of code revealed that 7.2% of all types in the code defined protocols, while 13% of classes in the code were clients of the types defining protocols and concludes that protocol checking tools are widely applicable [9]. Based on the definition for protocol in that study, a protocol violation requires an exception be thrown in instance methods as a result of reading an instance field. More protocols could be counted, if one extends this definition to include those cases, in which calling a method does not

 

necessarily throw an exception, but returns a value that using it might leads the program to an exception in the future.

It is clear that the notion of state plays an essential role in modeling the world and reasoning about object behavior. The lack of a feature in the conventional object-oriented programming languages to support for explicitly expressing state machines in actual code is the motivation behind making typestate [10]. Typestate captures the notion of an object's being in a proper (or improper) state for calling a particular method [8]. Files as our example may be open or closed. In the open state, one may read or write. Or one may close it, which causes a state transition to the closed state, in which file can only be (re-)open.

Using typestate concept, it makes possible to develop a program analysis technique which enhances program reliability by detecting at compile-time syntactically legal but semantically undefined execution sequence (type-correct application operations which are nonsensical in their current context) using typestate tracking (just like type-checking). These errors, such as reading a variable before it has been initialized, cannot be detected by type checking or by conventional static scope rules. Type checking detects that sort of nonsense which is independent of the context of an operation. With typestate is becomes practical to implement a secure language, in which all successfully compiled program have fully defined runtime effects, and program errors only depend on incorrect output values. It is enormously precious because in addition to detecting errors, it guarantees that the effect of any program error is limited to the erroneous module [8].

The other benefit of using typestate is that the typestate tracking helps compilers to insert appropriate finalization of data at exception points and on program termination, eliminating the need to support finalization by means of either garbage collection or unsafe de-allocation operations [8].

### 2.2. Plaid Programming Language

The state oriented approach of the Plaid draws inspiration from actors introduced by the Actor model [11] which was the first programming model to treat states in a first class way, but the concurrency approach [12] stays within a call-return function model rather than using messages.

From object modeling point of view, the closest work to the Plaid is Taivalsaari's proposal to extend class-based languages with explicit definitions of logical states (modes), each with its own set of operations and corresponding implementations. The proposed object model in plaid differs in providing explicit state transitions and in allowing different fields in different states [3].

In Plaid, codes reflect the design. That is, designs with stateful abstractions are clearly reflected source code. The implemented source codes match their designs represented in the statechart diagrams. Arbitrarily complex statechart can be encoded in Plaid with the simple rules described in [10]. Each abstract state maps to its own state in code, so the design of the abstraction and its protocol as a whole is highly salient in the code [13].

Plaid prevents common errors. Plaid's explicit state models make error checking more consistent, because the programmer cannot forget to check state constraints when a method is called. The level of abstraction of error messages is also thereby raised: when an inappropriate method is called, instead of triggering an internal run-time exception such as a null pointer, or silently corrupting data (which is the worst scenario), the runtime can signal an error that a particular method is unavailable in the current state. Also, we have shown how state members can be used to enforce consistency of multiple dimensions of state at once [13].

Using Plaid increases reusability. Plaid provides new reuse opportunities. Some state machines are used in many objects. For instance, the Position dimension was reused in both read and write streams, and it could also be reused in many IO and Collection libraries. Open and closed resources like the File and ResultSet are also very common [13].

## 3. Design and Implementation

During the process model adaption activity, we had to examine the efficiency of the proposed model by applying it into a real problem. We tried to do tasks for overall modeling and preliminary design for working with file system. The following are the outputs of these efforts including package diagram, statechart diagram, and typestates diagram.

### 3.1. Package Diagram

There are two packages containing all features of I/O API which are plaid.io.filesystem for working with filesystem and plaid.io.streams for working with streams. To overcome qualified naming in typestate-oriented programming, plaid.io. filesystem includes several sub-packages. Each sub-package is used for different state-space of different type, for example one sub-package for state-space of a valid path, one for an existent path, and so on. In this way, packaging typestates does not result any name conflict.

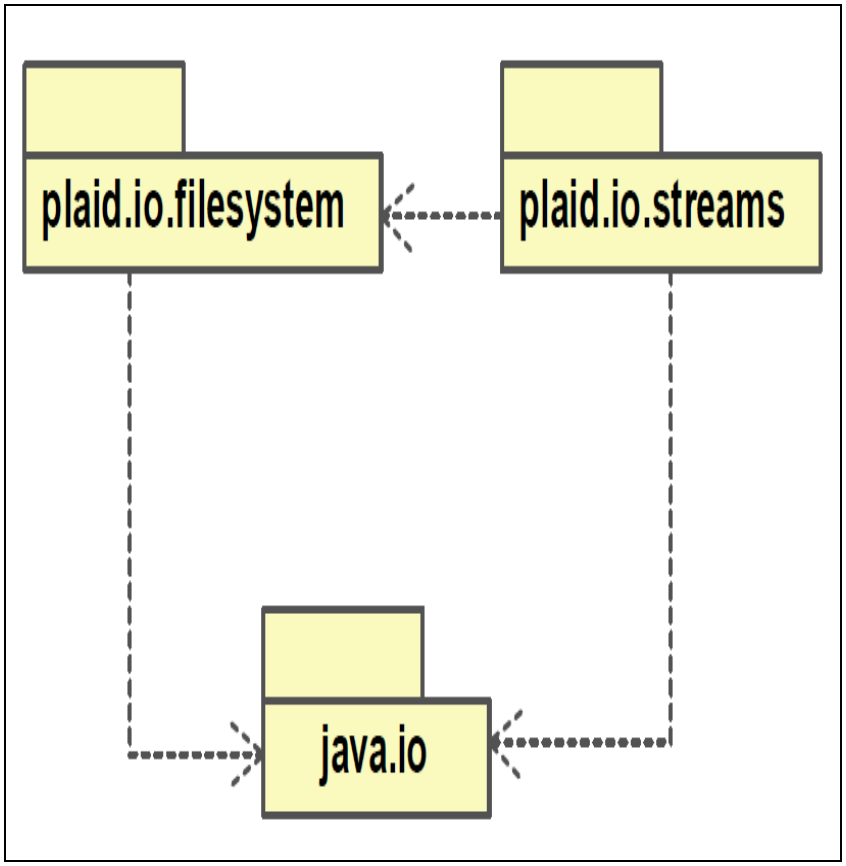


**Figure 1. The Package Diagram for Plaid I/O API**

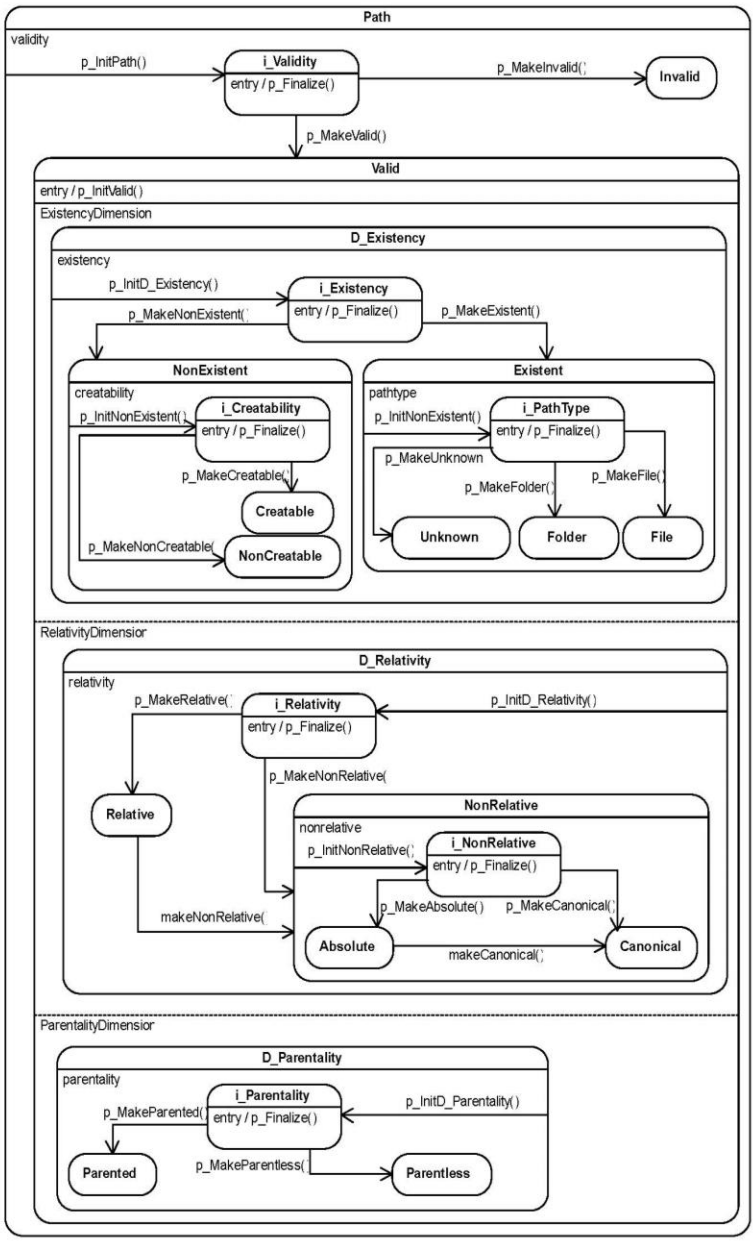


**Figure 2. The Statechart Diagram for Plaid.io.Filesystem Package**

### 3.2. Statechart Diagram

In typestate-oriented programming, statechart is the most important diagram for representing the system. Below the statechart of filesystem package are shown and by using them one can simply describe the features and behaviors of the system.

Working with filesystems starts with a path. A path can be valid or invalid. For example, in windows-based systems c:\windows is a valid path, but c:\con is invalid. None of the operations applicable to a valid path is allowed for an invalid one. For example one may check whether the path exists or not. By contrast performing any checking on an invalid path

throws an exception at runtime. This makes the designer to identify two distinct typestates for a path.

Since the validity of a path depends only on the entered pathname, which is a string value (*i.e.,* c:\windows), the process of construction of a path object should continue to determine its final state (Valid or Invalid). This situation is very common in practice, that is, the object construction from a specific state should result a set of state transitions which ends in one or several stable states (in this case Valid or Invalid is considered as stable state). This can be expressed by using initial transition in Harel's statechart. However, considering the lack of feature to implement initial transition in current version of Plaid, it is implemented as a simple internal transition and titled with a name starting with p_Init (*i.e.,* p_InitPath()).

The initial transition changes the state of receiver into an initial state (*i.e.,* i_Validity). The entry action (p_Finalize() method) of initial state is checking the validity and deciding whether the transition should proceed to Valid (by using p_MakeValid() method) or Invalid (by using p_MakeInvalid() method). This process of initialization can continue and go deep in a cascading way as much as necessary.

The rationale behind the initial state (or any other interim state) is that first, any checking, validation, and assertion can be performed in it. Second they can be used for a clean transition from one state to others. The implementation of any interim state should guarantee that the object never resides in the state after the completion of any transition.

For a valid path one may list several distinct states. For example a valid path might be existent path. The other might be non-existent. Regardless of existency, a valid path can be relative or absolute, as well as parentality. So, a valid path is a composite typestate, consists of three dimensions:

- Existency (ExistencyDimension region) which determines whether the path is exist or not.
- Relativity (RelativityDimension region) which specifies whether the given pathname is relative or absolute.
- Parentality (ParentalityDimension region) which identifies whether the given pathname has parent or not.

Composite typestates can be modeled simply by using orthogonal states in Harel's statechart; in which any transition in one region have no effects on the other regions. The initialization process at this level in each dimension is the same as described before.

It is worthy to note again, that designation of states does not relay only on throwing exceptions. But in some situation, states can be identified based on special values which a method or an operation produces. For example, considering File class in Java, calling getParent() or getParentFile() method results a null value if the receiver does not have any parent. Although calling them on a parentless path does not generate any error, nevertheless, the null value returned by the called method might cause a null exception error later, wherever the developer does not check being null of that value. It indicates that there are two distinct states which should be captured: Parented and Parentless.

 

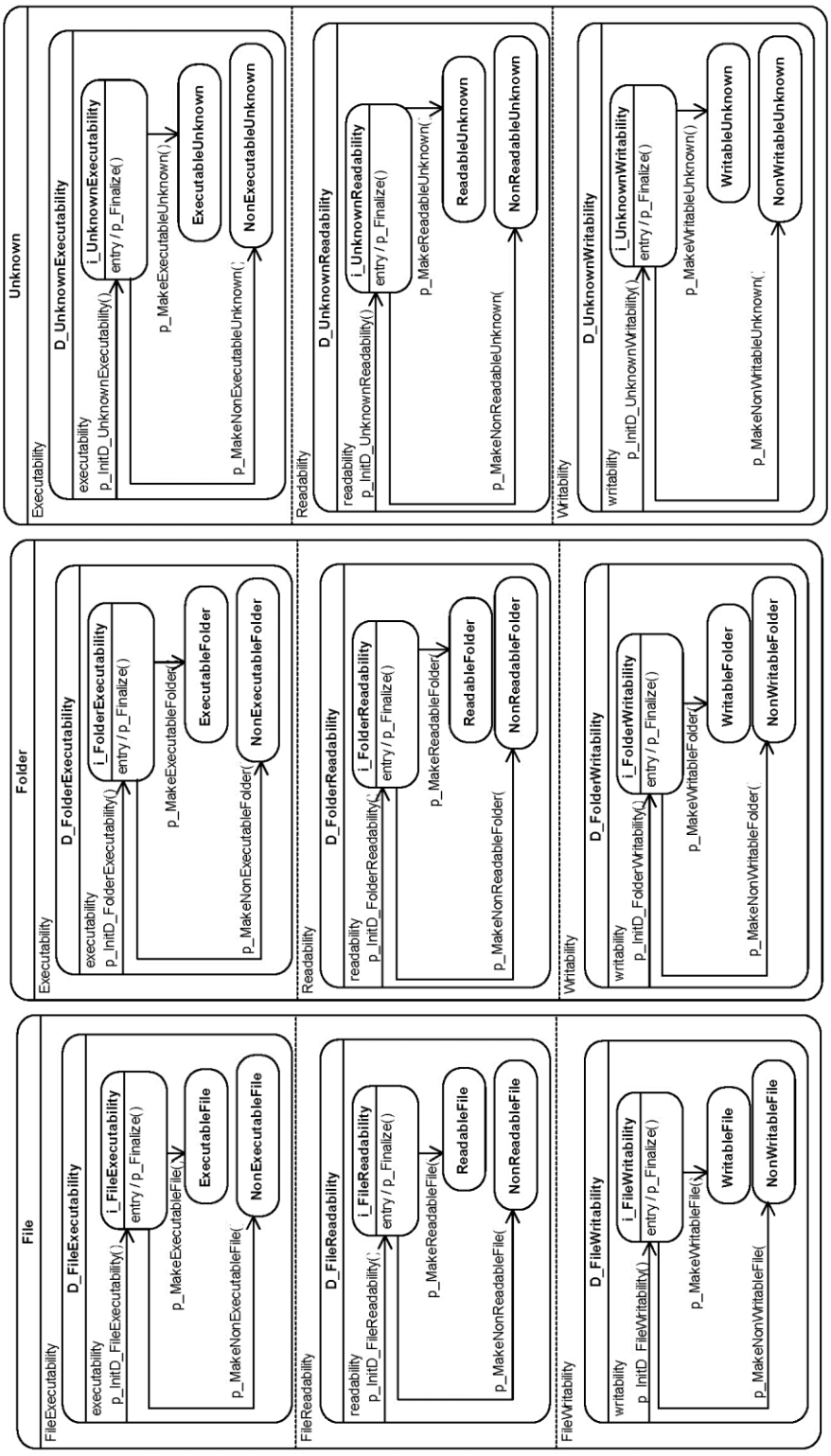


**Figure 3. The Statechart Diagrams for Permission Packages**

For simplicity, the statechart diagram has been divided into two separate diagrams. In the following, Figure 3 shows the three typestates used in the filesystem package: File, Folder, and Unknown. These three typestates representing the type of filesystem element to which the path points contain permissions of the path. The permission of a path in the (Linux-based) filesystem consists of three dimensions: Executability, Readability, and Writability. Each dimension is specialized into one interim state and two stable states. In one stable state the specified permission is allowed, but in the other not. For example an ExecutableFile represent a path which points to an existent file provided with read permission on it. While a NonExecutableFile represent one without read permission. All operations which are allowed for a readable file should be provided by ExecutableFile typestate.

### 3.3. Typestate (Class) Diagram

The following typestate (class) diagram represents the filesystem. These diagrams show the relation between different typestates in terms of generalization, specialization, and composition. Since, no neutral method is not to be shown in statecharts, this diagram is the suitable place to represent them.

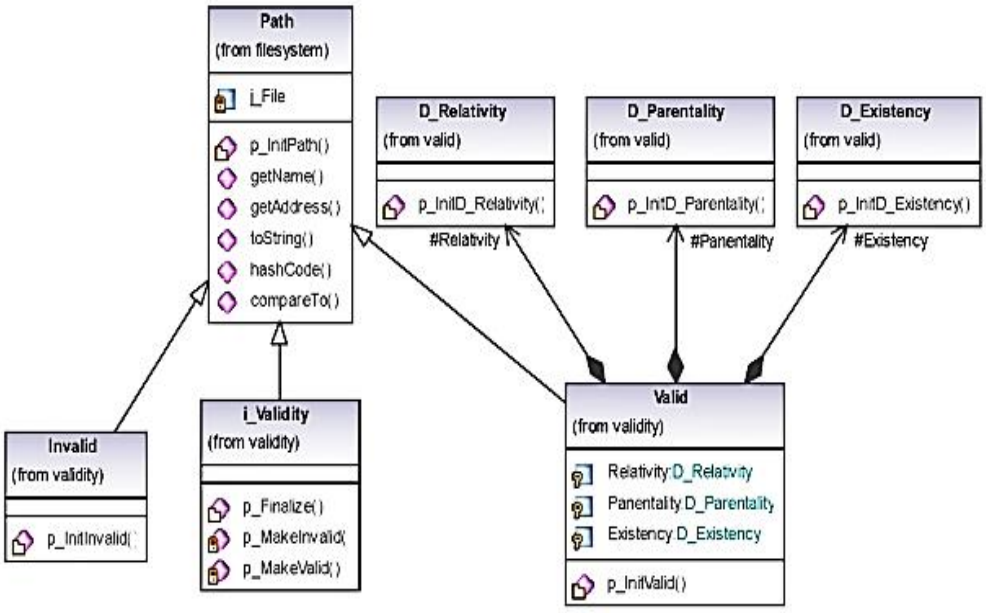


**Figure 3. The Package Diagram for Plaid I/O API**

Figure 3 shows the typestate (class) diagram for the six types which are used for modeling a path (valid or invalid). The typestate diagram is very similar to class diagram and both visualize the same information. The only difference applicable to this project is using the composite notation for different meaning rather than the one in conventional class diagram. In typestate-oriented programming, a type containing several dimensions is implemented using typestate composition feature provided by Plaid that is a novel way to represent different aspects of an object at the same time. The specialization relationships between types are shown in this diagram as well.

In Plaid, there are two different ways for type specialization which are inheritance and composition types. The concept comes from this fact that every object may have more than

 

one dimension. For example one may talk about a human in terms of sexuality, the other in terms of social personality and so on. This should not be confused with polymorphism provided by type specialization.

Note that any object of Permission has two dimensions. That means one may look at it as a Readability object, the other as a Writability one, both at the same time. The interesting point in typestate composition feature in Plaid is that, if the object changes its state within one dimension, this transition does not have any effect on the state of object within the other dimensions.

## 4. Testing

Considering the first-class function characteristic of Plaid programming language, we implemented a special package for testing rather than using an existing tool such as JUnit. The rationale behind implementing a new one is to be able to use benefits which are provided by Plaid such as flexibility and readability of test cases, as well as being easy to define, do, and control the testing process. The testing package contains two types: Testbed and TestCase.

TestCase is an immutable typestate which contains three attributes including name which is a string as test case name, body which is a function to be run whenever run(immutable Boolean verbose) is called. verbose is a boolean flag tells the testing system whether it should log the debugging information during the testing process or not. The other useful method which is provided on a test case object is assert(() -> immutable Boolean assertion, verbose) which gets a function and verbose flag and tries to assert the given function on the test case object.

Testbed is an immutable typestate which has a list containing test cases and provides functionality to create test cases and add them to the list. It has a string attribute which contains the test bed name. The main method in this type is runTestCases (immutable Boolean verbose) which runs all test cases stored in the list of the test bed with the given flag for verbose log of debugging information.

Considering the testing platform and its features the testing process is performed along the coding process, that is while the developer implement every typestate and its transitions and features, the developer should define all test cases needed to test the typestate and its functionality. All test cases are located in package.plaid file which is a general file provided for each package.

## 5. Evaluation

Table below show all protocol violations (or potential exceptions) which are absorbed and transformed into compile-time errors within the API.

**Table 1. Number of Absorbed Protocol Violations (or Potential Exceptions)**

| Feature Number | No. of Absorbed Protocol Violation | No of Absorbed Potential Exception | Total |
|---|---|---|---|
| SR-11-101 | 0 | 0 | 0 |
| SR-11-102 | 0 | 0 | 0 |
| SR-11-103 | 0 | 0 | 0 |
| SR-11-104 | 0 | 0 | 0 |
| SR-11-105 | 0 of 1 | 0 | 0 |
| SR-11-106 | 0 of 1 | 0 | 0 |
| SR-11-107 | 0 of 1 | 0 | 0 |
| SR-11-108 | 0 | 0 | 0 |

| Feature Number | No. of Absorbed Protocol Violation | No of Absorbed Potential Exception | Total |
|---|---|---|---|
| SR-11-109 | 1 of 1 | 1 of 1 | 2 |
| SR-11-110 | 0 of 1 | 0 | 0 |
| SR-11-111 | 0 of 1 | 0 | 0 |
| SR-11-112 | 1 of 1 | 0 | 1 |
| SR-11-113 | 1 of 1 | 0 | 1 |
| SR-11-114 | 0 | 0 | 0 |
| SR-11-115 | 0 of 1 | 1 of 1 | 1 |
| SR-11-116 | 0 of 1 | 1 of 1 | 1 |
| SR-11-117 | 0 of 1 | 1 of 1 | 1 |
| SR-11-118 | 0 of 1 | 1 of 1 | 1 |
| SR-11-119 | 0 of 1 | 1 of 1 | 1 |
| SR-11-120 | 0 of 1 | 1 of 1 | 1 |
| SR-11-121 | 0 | 0 | 0 |
| SR-11-122 | 0 | 1 of 1 | 1 |
| SR-11-123 | 0 | 1 of 1 | 1 |
| SR-11-124 | 0 | 0 | 0 |
| SR-11-125 | 0 of 1 | 1 of 1 | 1 |
| SR-11-126 | 0 of 1 | 1 of 1 | 1 |
| SR-11-127 | 0 | 0 | 0 |
| SR-11-128 | 0 | 1 of 1 | 1 |
| SR-11-129 | 0 of 1 | 1 of 1 | 1 |
| SR-11-130 | 0 of 1 | 1 of 1 | 1 |
| SR-11-131 | 0 of 1 | 0 | 0 |
| SR-11-132 | 0 of 1 | 1 of 1 | 1 |
| SR-11-133 | 0 of 1 | 1 of 1 | 1 |
| SR-11-134 | 0 of 1 | 3 of 3 | 3 |
| SR-11-135 | #N/A | #N/A | Dropped |
| SR-11-136 | 0 of 1 | 3 of 3 | 3 |
| SR-11-137 | #N/A | #N/A | Dropped |
| SR-11-138 | #N/A | #N/A | Dropped |
| SR-11-139 | 0 of 1 | 0 | 0 |
| SR-11-140 | 1 of 1 | 0 | 1 |
| SR-11-141 | #N/A | #N/A | Dropped |
| SR-11-142 | #N/A | #N/A | Dropped |
| SR-11-143 | 1 of 1 | 0 | 1 |
| SR-11-144 | 1 of 1 | 0 | 1 |
| SR-11-145 | 2 of 2 | 1 of 1 | 3 |
| SR-11-146 | 1 of 1 | 0 | 1 |
| SR-11-147 | 1 of 1 | 0 | 1 |
| SR-11-148 | #N/A | #N/A | Dropped |
| SR-11-149 | 1 of 1 | 0 | 1 |
| SR-11-150 | 1 of 1 | 0 | 1 |
| SR-11-151 | 0 | 0 | 0 |
| SR-11-152 | #N/A | #N/A | Dropped |
| SR-11-153 | #N/A | #N/A | Dropped |
| Total | 12 of 33 | 23 of 23 | 35 of 56 |

## 5. Conclusion

In this research the aim was to design and implement an API for I/O to be used in Plaid standard library. The features considered for this API were listed based on Java I/O API.

Ideally in typestate-oriented programming, the target is to absorb all exceptions related to violation usage protocols. That is, all runtime errors happening to the end user of system should be transformed to compile-time error, and the developer should be aware of them.

 

However the exceptions related to hardware failure and invalid data entered by user cannot be absorbed in this way. In this project we tried to capture all possible protocol violations for I/O operations and absorb them by using distinct states as different contexts for objects. Above and beyond that, we tried to observe all potential exceptions such as null values returned by methods and eliminate the possibility of generating runtime error caused by them.

For future work we would like to extend the Plaid standard library features by implementing the types in other packages such as java.nio, which are not included in java.io, but related to its concept and have the same nature. Also, re-implement the API to capture more violations and increase the reliability of the system.

## Acknowledgements

The authors would like to thank the Lab of Advanced Informatics School for their offered helps, and all the members of the Lab for their useful discussions that guided us through this research. Also, to all academic staff and students of Advanced Informatics School who have been participating directly and indirectly in this study. The financial of this project is supported by Ministry of Higher Education Malaysia and Universiti Teknologi Malaysia under Vot No: 07J87.

## References

[1] G. S. Fowler and D. G. Korn, "Principles for Writing Reusable Libraries", ACM SIGSOFT Software Engineering Notes, vol. 20, no. 1, **(1995)**.
[2] J. Aldrich, N. E. Beckman, R. Bocchino, K. Naden, D. Saini and S. Stork, "The Plaid Language: Typed Core Specification", Version 0.4.0, April 2012 (No. CMU-ISR-12-103). USA, Carnegie Mellon University.
[3] J. Aldrich, J. Sunshine, D. Saini and Z. Sparks, "Typestate-Oriented Programming", Proceedings of the 24th ACM SIGPLAN Conference Companion on Object-Oriented Programming Systems Languages and Applications, Orlando, Florida, **(2009)** October 25-29.
[4] J. Aldrich, R. Bocchino, R. Garcia, M. Hahnenberg, M. Mohr and K Naden, "Plaid: A Permission-Based Programming Language", Proceedings of the ACM International Conference Companion on Object-Oriented Programming Systems Languages and Applications Companion, London, UK, **(2011)** August 16.
[5] J.-Y. Girard, "Linear Logic", Theoretical Computer Science, vol. 50, no. 1, **(1987)**
[6] J. McCarthy, P. W. Abrahams, D. J. Edwards, S. D. Hart and M. I. Levin, "LISP 1.5 Programmer's Manual", (1st ed.), MIT Press, **(1962)**.
[7] K. Bierhoff and J. Aldrich, "PLURAL: Checking Protocol Compliance under Aliasing", Companion of the 30th International Conference on Software Engineering, Liepzig, Germany, **(2008)**, pp. 10-18.
[8] R. E. Strom and S. Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability", IEEE Transactions on Software Engineering, vol. 12, no. 1, **(1986)**.
[9] N. E. Beckman, D. Kim and J. Aldrich, "An Empirical Study of Object Protocols in the Wild", Proceedings of the 25th European Conference on Object-Oriented Programming, Lancaster, UK, **(2011)** July 25-29.
[10] J. Aldrich, "The Plaid Programming Language", Overview, Portland, **(2011)** July 22-27.
[11] C. Hewitt, P. Bishop and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence", Proceedings of the 3rd International joint Conference on Artificial Intelligence, Barcelona, Spain, **(2011)** July 16-22.
[12] S. Stork, P. Marques and J. Aldrich, "Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs", Proceedings of the 24th ACM SIGPLAN Conference Companion on Object-Oriented Programming Systems Languages and Applications, Portland, **(2011)** July 22-27.
[13] J. Sunshine, K. Naden, S. Stork, J. Aldrich and E. Tanter, "First-Class State Change in Plaid", Proceedings of the 2011 ACM International Conference on Object-Oriented Programming Systems Languages and Applications, Portland, **(2011)** July 22-27.

 

## Author

**Nazri Kama** obtained his first degree at Universiti Teknologi Malaysia (UTM) in Management Information System in 2000, second degree in Real Time Software Engineering at the same university in 2002 and his PhD at The University of Western Australia (UWA) in Software Engineering in 2010. Nazri has a considerable experience in a wide range on Software Engineering area. His major involvement is in software development. Currently, positioned as a Chief Executive Officer of a Spin-off company under his University, Nazri has moved his focus on managing a software development project and providing consultation services to both government and commercial sectors.