Manual

ALI MAHJUR, Malek Ashtar University of Technology, Iran

This paper introduces a new computation model whose purpose is to facilitate the design of algorithms.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Ali Mahjur. 2018. Manual. *ACM Trans. Comput. Theory* 37, 4, Article 111 (August 2018), 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

This document is the specification of the Parham programming language.

2 COMPONENT AND CLASS

In Parham, a program is a set of components. A component itself is a set of classes.

Parham has two types of classes: internal and interface. An internal class is a class which is internal to its container component. Therefore, it is not possible to extend it.

An interface class is a class whose behavior is partially defined by its container component. Such classes are extended when the component is instantiated.

Every component except the main component has at least one interface class. The main component has no explicit interface classes. It has an implicit interface class whose name is the same as the name of the component.

3 DECLARATIONS

The body of a component is a set of declarations. They define the behavior of the component itself or its classes. Parham supports the following declaration types.

(1) Field declaration
(2) Method declaration
(3) Constructor declaration
(4) Association declaration
(5) Instance declaration
(6) Inclusion declaration

Author's address: Ali Mahjur, mahjur@mut.ac.ir, Malek Ashtar University of Technology, P.O. Box 1212, Tehran, Tehran, Iran.

1942-3454/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

3.1 Field declaration

A field declaration defines a field for a class. A field declaration specifies the following properties.

(1) Container class
(2) Type
(3) Name
(4) Modifiers

This is an example of a field declaration.

```
Input[Data]
Data
{
   int value;
}
```

The type of a field can be a primitive type or an internal class. The possible modifiers of a field are the following.

(1) *provides*
(2) *requires*
(3) *const*

3.2 Method declaration

3.3 Constructor declaration

A constructor initializes an object of a class when it is created. It is possible to define any number of constructors for a class of a component. If the programmer does not define any constructor for a class, the compiler defines a default constructor for it whose arguments and body are void.

A constructor has a body. In addition, it has to call the constructor of the following items to initialize them too.

(1) Map
(2) Super
(3) Fields whose types are internal classes.

A call to an upper constructor specifies the field that it constructs using one of the following order.

(1) The name of the field. This case is used to initialize a field declaration or a named map declaration.
(2) The name of an instance.
(3) The name of an instance and an interface class of its component.
(4) The name of a class. It is used to initialize a super class.
(5) The name of a component
(6) The name of a component and an interface class of its component.

If the programmer does not provide a call to an upper constructor and the class has a constructor whose arguments are void, the compiler adds a call to this constructor. Otherwise, the compiler generates an error.

3.4 Association declaration

3.5 Instance declaration

The reusability model of Parham is based on the component concept. To reuse a component an instance of it should be introduced. Assume that component **LinkedList** is declared.

```
LinkedList[Root, Node]
   Root->Node head;
```

```
  Node->Node next;
  Root ()
  {
    head = null;
  }
  void Root.insert (Node node)
  {
    node.next = head;
    head = null;
  }
```

Now, component `Observer` can declare an instance of `LinkedList`.

```
Observer[Subject, Listener]
  LinkedList[Subject, Listener]
```

In this example, an instance of `LinkedList` is declared. Therefore, `Observer.Subject` inherits from `LinkedList.Root` and `Observer.Listener` inherits from `LinkedList.Node`. Of course, it is possible to assign a name to a role.

```
Observer[Subject, Listener]
  LinkedList[Subject subject, Listener]
```

A component can instantiate another component any number of times.

When a component is instantiated, the compiler tries to resolve its *requires* methods and fields. The resolve process is done in two steps. In the first step, explicit resolving is done. In this step, the *requires* methods that the programmer explicitly has specified are resolved. For this purpose, every component instance can have a delegate section where *requires* methods are explicitly resolved. This is an example of this model.

```
Observer[Sensor, Light]
{
  Listener.notify = x.y.update;
}
```

In the above example, the left side specifies the method which is resolved: `Listener.notify`. Recall that the signature of `Observer` is as follows.

```
Observer[Subject, Listener]
```

Therefore, the above resolve section selects class `Listener`. It then selects method `notify` of it.

The right side is as follows. It specifies an instance, a class and a method of it to resolve. If there is no ambiguity instance and class names can be omitted.

The implicit resolving is done as follows. When a component is instantiated its `requires` methods that have not resolved explicitly or explicitly marked to be resolved again are looked up and if a match was found it is resolved. Note that if more than one match is found, resolve is not done.

3.6 Inclusion declaration

4 MODIFIERS

A declaration may have some modifiers. Parham supports the following modifiers.

(1) *provides*
(2) *requires*
(3) *local*
(4) *extern*
(5) *const*

4.1 *provides* Modifiers

In general, everything declared within a component is private to that component. If something should be accessible from the outside of the component it should be marked by *provides* modifier. Now, when an instance of the component is declared its declarations that have the *provides* modifier can be accessed to the instantiating component.

The following declarations may have *provides* modifier.

(1) Enumeration types
(2) Typedef types
(3) Const values
(4) Association declarations
(5) Component Instances
(6) Methods
(7) Fields

Note that a constructor has *provides* modifier implicitly. Therefore, it is not required to mark them by *provides*.

4.2 *requires* Modifiers

A component may declare some of its attributes (methods and fields) as *requires*. It means that when such a component is instantiated the *requires* attributes must be provided. Of course, if a method is marked as *requires* the component may provide a default behavior for it. In this case, providing the method is optional.

The first step in providing a *requires* method is to specify such a method. The interface classes of a component can have *requires* attributes (methods and fields). The *requires* methods of an interface class of a component are specified using the following rules.

(1) A direct *requires* attribute (method or field)
(2) An indirect *requires* attribute (method or field) which has not resolved
(3) An indirect *requires* attribute (method or field) which is marked as *requires* again.

Providing an implementation for a *requires* method can be done in two ways. When a component is instantiated its *requires* methods are specified. Then, there are two ways to provide implementation for them.

(1) Delegate clause
(2) Name matching

This is the description of delegate clause. Every instance declaration has an optional clause. This clause may specify implementation for *requires* methods of the instance.

In name matching, when a component is instantiated, its *requires* methods are lookup in the container component and other instance declarations of the component. If a match is found it is assumed as the implementation of the *requires* method.

Fields are handled in the same way too.

Finally, all *requires* attributes of an internal class must be resolved.