# ModLang
# Good Practices

Zachary Kieda (zkieda@andrew.cmu.edu)

October 20, 2015

## 1 Semantics

Here, we will go over the semantics of module initialization that we use for the CrowdSimulation project. As we know, ModLang will generate a tree structure of modules. But what about the order and selection of dependency injection, and the initialization techniques that are used. Figure 1 displays an example tree that we will use to display the semantics.

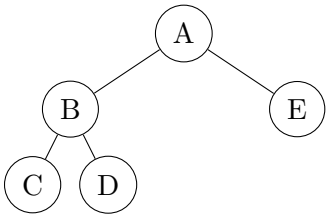


Figure 1 – An example ModLang class tree

### 1.1 Phase 1 : Linking

In the first phase, after the class tree is generated, we link modules together – we insert resources that have the `@AutoWired` annotation. We find resources in the specified manner – we first run a BFS from the current node to child nodes find the module. Then, we perform a BFS from the current node to the parent node if we did not find the module in the children.

Suppose we wanted to inject module D into module A. The priority of modules we use is B-E-C-D. In the case the module is not found in the children, like if we were injecting module E into module B, the priority we would use

is C-D-A-E.

**Note:** during this period we allow a single module to accept an arbitrary number of modules by using the syntax
`@AutoWired java.util.Collection<T extends SubModule>` where `T` is the type of the submodule the client wants to capture. This is processed in this stage.

### 1.2 Phase 2 : Initialization

This is the stage that initializes all of the objects in the hierarchy by running its internal `init()` method. This process must be manually started by the client by calling `init(null)` at the root of the tree. We this because there may be external initialization parameters that the client may want to insert before initialization. A typical example is if the module tree uses `String[] args` as a parameter. In the example below we inject `String[] args` into `ArgsListenerModule`.

```
MyModule root =
    ConstructModule.build(new File("mymodule.mod"));

Optional<ArgsListenerModule> arg =
    root.getModuleByClass(ArgsListenerModule.class);

if(arg.isPresent()){
    arg.get().init(args);
}

root.init(null);
```

Intuitively, initialization is performed by initializing dependencies before initializing the parent. For example, the initialization strategy we would use from Figure 1 is C-D-B-E-A. Note that the root is initialized last, and children are initialized from left to right. This resembles the composition you would see in plain java code, seen below

```
//Same initialization order
//Represents Fig. 1
new A(new B(new C(), new D()), new E());
```

After this phase is complete, we consider all of the modules in this tree to be "complete" in its initialization, and ready to run as a system. This initialization of a tree is akin to the initialization of a java class.

### 1.3 Phase 3 : Listeners

Listener processing occurs immediately after initialization has completed. A module has the option of listening for new modules that are added to the module tree. Initially, we process the modules that are inside of the same linking unit.

In the same unit, we follow a two step process. First, we collect the list of listeners is in the same order used in Section 1.2 (C-D-B-E-A). Then, we process the applicable listeners for each module in the same manner. This is done for two reasons, 1) we want to apply the dependent listeners before applying the parent listeners; the parent may expect some invariant of the child, 2) we process the modules bottom-up since we want the dependent modules processed before the parent.

Finally, after all of the local listeners have been resolved, we process modules that are directly added to this tree. We only check the root of the module added to this tree, and not children of the root.

## 2 Stateless Modules

In stateless modules, we want to treat each module as if it were a function. By stateless, we mean that the module should not carry any state *after* its external initialization.

For example, we may want to inject the `String[] args` to a module that processes the command line arguments. This is done after linking and before initialization as described in Section 1.2. Then, the module should only be a supplier for a representation of the command line arguments and not change its state. Stateless modules should be used to initialize state-based modules. Alternatively, stateless modules should be used by state holding modules during the runtime as supporting functions.

# 3 State-Based Modules