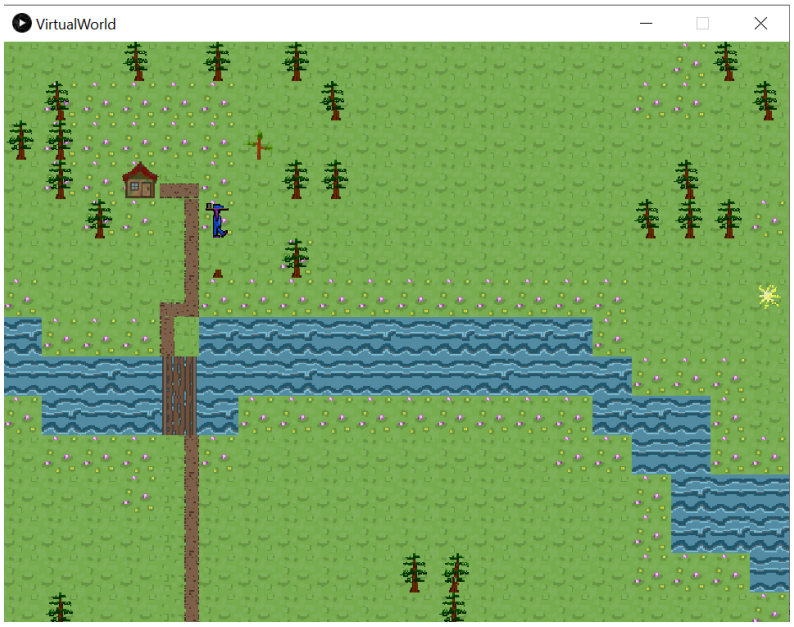
# Forest Project cont.(Project 3)




In the previous project you identified methods and moved them into the most appropriately associated classes. In doing so, you may have noticed that some classes support functionality (methods) that are not appropriate for all instances of the class. Moreover, these classes support data attributes that are not used by all instances of the class.

This is an issue of **cohesion**. Specifically, these classes exhibit low (poor) cohesion by representing multiple concepts, combining all attributes and methods used by each concept in a single class.

This project asks that you improve the code base by splitting each class exhibiting low cohesion into multiple, highly cohesive classes. Doing so in the context of Java will require identifying common methods for each subset of these new classes and then introducing a new **parent type** for each logical grouping of these methods (more on this below).

## Objectives

- Deepen your understanding of the specific functionality of the large project design
- To be able to read and understand java code and be able to evaluate the cohesion of the class structure in existing code
- To be able to implement abstract classes and/or interfaces and use them to improve cohesion in a project design

- Specifically, for this assignment, in the actual java code, introduce appropriate classes in order to remove the need for enumerated types and for any other classes that contain methods that do not support the primary role of instances of that class
- To be able to make design changes to a large code base and have the code still work

## Task Overview

You must identify those classes with low cohesion and then split these classes into separate classes exhibiting high cohesion. Since each of these new classes will introduce a separate type, you may need to "root" them at a single type (as defined by an **interface** or an **abstract class**) to satisfy Java's type checking rules.

Based on the original source code, there are likely two categories of classes with low cohesion. The first category consists of those classes that depend on **ActionKind** or **EntityKind**. The second category depends on your final distribution of the methods in the original **Functions** class.

- Kind:
  The original source code uses **ActionKind** and **EntityKind** to allow each **Action** instance and each **Entity** instance to play one of potentially many roles (this mimics a tagged union). You are to eliminate these **Kind** classes (enums) by splitting **Action** and **Entity** into multiple new classes.

- Other:
  Review all of the classes with an focus on cohesion. Does a class contain data that is not used by all instances of the class (i.e., each "kind" uses only subsets of the data)? Does a class contain methods that do not support the primary role of instances of the class (e.g., static methods that are used to create instances or parse files, but that are not actually part of the functionality provided by the instances)?

As before, you are encouraged to develop both the UML design document and the code refactoring at the same time. You are further encouraged to implement the refactoring incrementally so that your refactored program executes properly at each step.

## Introducing Parent Types

Consider this example.

Above we've discussed the `EntityKind` enums, which are used to differentiate between different kinds of `Entities`.
In this project, you'll need to split those classes into multiple new classes, each of which represents a _specific_ kind of `Entity`. However, to satisfy Java's typechecking rules, we need to "root" those new classes at a single type.

You have several strategies in your arsenal that will help you address this. Namely, you can introduce an `interface` or an `abstract class`. Carefully consider the pros and cons of either approach.

### *Strategy 1*

An `interface` can define a number of abstract methods which are then implemented by each of the implementing subclasses. This solves our problem of rooting our new subclasses at a single parent type.However, it will introduce a fair amount of code duplication.
This is because each implementing subclass will need to implement all of the abstract methods that are listed in the `interface`, _even if the implementations are identical for multiple subclasses_.

How to address this?

### *Strategy 2*

This can be addressed by using `default` methods in your interfaces. Default methods let you provide implementations for certain methods (which will be used by the implementing subclasses unless they have their own implementations).

This solves the problem of duplicated method implementations but does still cause difficulties because interfaces cannot have instance variables.

### *Strategy 3*

You can address this by instead using an `abstract class`. `Abstract classes`, as you recall, can have a mix of abstract and fully implemented methods (in a manner very similar to an interface having abstract and default methods).

A key difference is that abstract classes can also have _instance variables_---this means you can avoid duplication of _data_, not just methods.

**So why not just use abstract classes if they solve so many problems?**
Remember that a class can extend _no more than one_ abstract class. As you design your solution, you will find that this introduces a number of constraints, not all of which are desirable.

Like many problems in software design, there is no "silver bullet" that solves all your problems.

You will consider design tradeoffs and make your own decisions about how to approach this project, likely using a mix of the above strategies.

No matter what you do, your main guiding principles throughout will be:

- **Improve cohesion**. Classes should only include functionality that relevant to _all_ instances of the class. There should NOT be functionality in a class that only relevant to _some_ instances of the class.

- -**Remove code duplication**. There should be little-to-no code duplication in the project once you're done. Where classes have similar or identical code, abstract out that functionality into a parent type.

The following are some tips on approaching the introduction of interfaces or abstract classes to support splitting classes.

**Note:**

A class should _not_ implement an interface (or extend an abstract parent class) only to then define a method required by the interface (or abstract class) to do nothing at all.

A class should _not_ implement an interface (or extend an abstract parent class) and then define a method required by the interface (or abstract class) to raise an exception indicating that the method is not supported.

Your introduction of parent types for this project must be meaningful. It is insufficient to define a single interface / abstract class with all methods that are then only partially implemented by each of the classes.

- First, copy the original class to each of the new classes (each defining a single role). This can be easily done in the UML editor.

- In each new class, eliminate each data attribute not used by this class and each method not supported by this class. (For this project, you can examine how instances playing this role are created as a hint about which data attributes are actually used.)

- Change the original class into an interface declaring only those methods shared by every new class.

- Group the new classes into sets with similar functionality. Introduce additional interfaces as appropriate (see below).

- Examine the original uses of the objects (before this change) to determine which methods are used by client code. Can the client code still access that method based on the reference type? Will it be able to do so if you change the type to one of the interfaces that you have already introduced? Do any interfaces have to extend a more general interface for it to compile?

At this point, if you only added interfaces, you will have lots of duplicate code. Next, consider if your interfaces could use default methods or if there is any common data / implementation you can pull up if your interface was instead an abstract parent (or if your interface was implemented by an abstract parent).

## Design Document

Your "design document" will consist only of an updated UML diagram.
Copy the UML diagram from the prior design assignment to `**Cohesion.graphml**`. Then update this copy to reflect the new classes and the interfaces.

You can add an **interface** to the UML document by adding a class and then, under the _UML_ tab in properties, setting the _Stereotype_ field to "interface". For each interface that a class implements, be sure to draw an [appropriate arrow] (https://en.wikipedia.org/wiki/Class_diagram#Relationships) (dashed line with open triangle head) from the class to the interface.

If an interface extends another interface in your design, then be sure to connect them with an appropriate arrow (solid line with open triangle head).

## Source Code Refactoring

Your refactoring should mirror the work done for your design document (UML diagram) augmented with feedback from your instructor.

Your refactoring must not add or remove any program functionality. The resulting program must work as before.

### *Tips on Refactoring Methods*

You can use the compiler (on the command-line or in the IDE) to help you with your refactoring. In particular, as you introduce interfaces and abstract classes, the compiler will report attempts to use methods not supported by the specified type. The existence of such errors may indicate missing methods for an interface or, more likely, attempts to treat a group of objects more generally than should be supported (i.e., not all of them implement the desired operation).

As part of your refactoring, you will be eliminating the `*Kind` classes. This is desired to allow each new class to directly implement a single role, but has the unfortunate side-effect of eliminating a simple check of an object's "kind". This check is used, for instance, when searching for the nearest `Tree` to a `Dude`.

Consider the following tips.

- For a class that is being split into multiple class, change the original class into an interface declaring no methods. Compile the program to determine all uses of this interface (the method invocations will trigger compiler errors). Now determine which of these methods must be supported by all instances of an interface or abstract class and which should be supported via additional interfaces.
  You can copy the original class to, and change all references to, `NameTmp` and declare it to _implement_ the new interface (or _extend_ the new abstract class) so that most of the code will continue to compile.

- For those methods that are not logically part of the primary interface defined in the prior step, introduce new interfaces and change the necessary variable declarations to use the new types.

- A check for the \"kind\" of a referenced object can, for now (though we will address this later), be replaced by a use of `instanceof`. Use this sparingly; certainly `instanceof` is not needed to check the type of `this`.

- In the case that a `*Kind` value was passed as a parameter to another method (and then compared within), you can do the following.

- Change the parameter type from the specific `*Kind` to `Class` (this is a type where each instance represents properties of a specific Java class).

- Instead of passing a `*Kind` value, use `.class` to get the object associated with the desired Java class (e.g., `String.class` gives the `Class` object describing the `String` class).

- Change the comparison to use the `isInstance()` method on the `Class` object, passing to this method the object to be checked.

- For two methods that appear to be doing roughly the same thing, but that differ slightly in their implementation: examine the code to determine if the code can be rewritten to match. This does require careful consideration for what each method does (and does _not_) to avoid introducing bugs.

- Some methods may have the same general structure (and match identically in significant portions), but differ in some segments. For such methods, the general structure and identical portions can be refactored into a parent class. This parent class will declare new `protected abstract` method(s) that each subclass then implements to define the unique behavior (as done in the calculator lab).

## Deliverables

Your project submission will contain the following deliverables:

* The refactored source code

* The 'Cohesion.drawio` UML diagram

All of this must be submitted through Canvas. Put all the deliverables in a compressed folder and upload it before the due date.

Your grader should be able to build your project based on the files submitted. An explicit list of files is notgiven because you are creating new files for this assignment, so verify that you have submitted everything properly. Remove files from your submission that are no longer needed for your project (e.g.EntityKind.java).