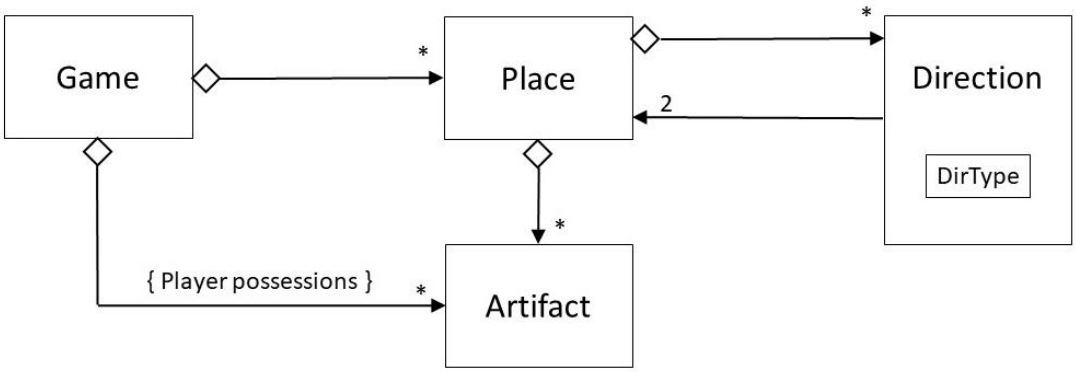
CS 342 – Software Design – Fall 2018
Term Project Part II
Addition of Artifacts and Data File Processing

Due: **Wednesday 3 Oct. Electronic copy due at 12:30 P.M. Optional paper copy may be handed in during class.**

Overall Assignment

For the next release two major features will be added: Support for artifacts and the processing of input data files. In addition the Direction class will be modified to employ an enumerated type. The revised diagram of the system is as follows:



The changes that will need to be made are the following:

1. Modify the Direction class to use an internal private enum class called dirType to keep track of what "kind" of direction each Direction item is.

    a. Allowable values of the enumerated type are NONE, N, S, E, W, U, D, NE, NW, SE, SW, NNE, NNW, ENE, WNW, ESE, WSW, SSE, and SSW.

    b. Each type should have two fields defined: an abbreviation such as N or NNW, and a text such as North or North-northwest.

2. Make modifications to support the Artifact class

    a. Create the Artifact class. Each Artifact should contain a name, a description, a value, a mobility, and a keyPattern.

        i. Mobility is a measure of size or weight. A negative value indicates an immovable object. Used only for optional enhancements at this time.

        ii. KeyPattern is zero for any artifact that cannot act as a key. Negative keyPatterns may be applied as master keys as an optional enhancement.

    b. Add storage in both Place and Game classes to hold a collection of Artifacts. The Artifacts held by the Game class are the possessions of the player. Methods will be needed to add and remove Artifacts to and from the collections.

c. When displaying a Place, the artifacts present should be listed with their properties.

d. The Direction class will now need a lockPattern field and a useKey(Artifact) method, that toggles the state of its lock if the keyPattern of the Artifact matches the lockPattern of the Direction. (lockPatterns and keyPatterns are ints.)

e. The play( ) method of the Game class will need to support 4 new commands:

i. GET artifact

ii. DROP artifact

iii. USE artifact – At this point the only usable Artifacts have a non-zero keyPattern, which will attempt useKey( ) on all Directions in the current place.

iv. INVE or INVENTORY – List all artifacts currently in possession of the player, along with the total value and total mobility.

3. Write constructors for the Game, Place, Directionand Artifact classes that take an open file as their argument. Use the constructors to load in Mystic City 3.0, according to the GDF version 3.1 file format.

Game Class Details

Additions and changes to the Game class are as follows:

- Game(Scanner) – A constructor for creating the Game object. The Scanner should be connected to an open file conforming to file format GDF (Game Data Format) 3.1. This method will pass the Scanner on to Place, Direction, and Artifact constructors as needed.
- addPlace(Place) : void – This method will no longer be used.
- print( ) : void – This method will now include the Player inventory in its printout.
- static getCurrentPlace(void) : Place – This needs to be called by the Artifact class so that it can interact with the Place within which it is used.
- play(void) : void – This method will now need to support 4 new commands:
  - GET artifact – Will check to see if the named artifact is present and attainable, and if so, will transfer it from the Place to the player's inventory.
  - DROP artifact – Inverse of GET, if the artifact is in the player's inventory.
  - USE artifact – Call the use( ) method of the artifact. At this point the only usable artifacts are those that have a non-zero keyPattern value.
  - INVENTORY or INVE – List the player's inventory of artifacts, providing the value and "mobility" of each, but not keyValues. You may express mobility in any desired units, such as pounds, kilograms, cubic inches, or a made-up unit of your own. Also report totals.
- The Game class will now need to keep track of a collection of Artifacts, corresponding to the player's possessions.
- ~~There needs to be a way to find a Place given its ID number. One option is to add a method to the Game class that would loop through all Places and return either the desired Place or null. (This could also be used for determining if a given Place ID number has been used before or not.) Another option would be for the Place class to have a static method that would perform this function.~~

Place Class Details

Additions and changes to the Place class are as follows:

- Place(Scanner) – A constructor for creating the Place object. The constructor must also add the newly created Place to the static collection of known Places. See example code online.
- addArtifact(Artifact) : void – adds an Artifact object to this Place's collection of Artifacts.
- static getPlaceByID(int) : Place – Returns the Place associated with the given ID number, or null. This will require that a static collection of all known Places be kept within the Place class. (A Hashmap or Treemap may be a useful collection type.) This method can also be used to detect if an ID has been used before or not, so that duplicates can be avoided.
- useKey(Artifact) : void – Passes the artifact to the useKey( ) method of all Directions present in this Place.
- print( ) : void – Prints out all of the Place information, including all Directions and Artifacts present. For debugging and testing purposes only, and will not be used in the play of the game.
- Internally the Place class will now have to keep track of a collection of Artifact objects, as well as the static collection of all known Places discussed above.

Direction Class Details

Additions and changes to the Place class are as follows:

- Direction(Scanner) – A constructor for creating the Direction object. Note that all new Directions are by default unlocked.
  - The constructor must now add the newly created Direction object to the Place corresponding to its source ("from" field), using Place.getPlaceByID( ).
- A new private nested enumerated type class DirType. See below for details.
  - Also store a variable of this type, instead of storing a string.
- match(String s) : boolean – Returns true if the String passed in matches that of the stored direction, dir. This method merely calls the match(String) method of the DirType variable.
- A new int field lockPattern, which must be non-negative. A zero value indicates the lock status may not be changed
  - useKey(Artifact) - If the keyPattern of the Artifact is positive and equal to the lockPattern, toggle the state of the Direction lock.
- print( ) : void – will now print out the lockPattern.

DirType Enum Details

The enumerated type DirType will need the following fields and methods:

- DirType(String text, String abbreviation) – A constructor for creating the DirType object.
- Defined constants for the 19 defined direction types: NONE, N, S, E, W, U, D, NE, NW, SE, SW, NNE, NNW, ENE, WNW, ESE, WSW, SSE, and SSW. Example data for these types include ("North", "N"), ("North-Northwest", "NNW"), and ("None", "None");
- Two private String data fields, text and abbreviation.
- String toString( ) returns the text field.
- boolean match(String) returns true if the given string matches either the text or the abbreviation, ignoring case.

Artifact Class Details

The Artifact class will need the following:

- Artifact(Scanner) – A constructor for creating the Artifact object.
- value(void) : int – Returns the value of the artifact.
- size(void) : int, or weight(void) : int – Returns the movability value.
- name(void) : String – returns the name of the artifact.
- description(void) : String – returns the description of the artifact.
- use(void) : void – "Uses" the artifact. In the case of keys, this will involve getting the current place from the Game class, and then passing the artifact to the useKey( ) method of the current Place.
- print(void) : void – Used for debugging, prints out full details.

GameTester Class Details

The GameTester class is the test driver for the other classes, and contains main( ). It should first print your name and netID. Initially main( ) should create instances of the different classes and test them to make sure they work correctly. Then that code can be commented out, and code put in to create a Game object containing an interconnected network of Places and Directions, and the Game class tested with the play( ) method.

GameTester should take a command-line argument of the name of a file, containing data in GDF 3.0 format. (See separate documentation.) This file should be opened and connected to a Scanner, which should then be passed to the Game constructor. If the file cannot be opened for some reason, then the program should ask the user for an alternate file name, and continue doing so until a file can be opened successfully or the user enters "quit" as the file name.

Note that your classes will be tested both by running your ExamTester main( ) routine and by a separate test driver written by the graders, so make sure your classes work with any reasonable inputs, not just with certain special inputs.

Additonal Methods May Be Needed

In the spirit of information hiding, the internal data and workings of the class have not been specified. **All data should be kept private**, and you may need/want to add additional methods both public and private to those listed here. In particular, constructors, destructors, setters, and getters are often needed but not specified. Be careful, however, as too many setters and getters can destroy information hiding.

If you feel a need to add additional public methods, make sure to document them well in your readme file. It is probably good to discuss them with an instructor as well, in case that method will be needed by all students.

Simplifying Assumptions That May Be Relaxed Later

- TBD

Required Output

- All programs should print your name and netID as a minimum when they first start.
- Beyond that, the system should function as described above.

Evolutionary Development

It is recommended that you first refactor the Direction class to use the enumerated type, and then add support for Artifacts, and finally to develop the constructors that take Scanners as their arguments. You may hard-code the creation and addition of Artifacts until the constructors are written.

It is also recommended that you set up a (git) repository to hold different working versions of your program as they develop, starting with what you turned in for HW1, and saving a new version as you complete the addition of each new feature. If you use a repository that stores the files in a location other than your personal computer, (e.g. GitHub), please ensure that it is kept private.

Other Details:

- The TA must be able to build your programs by typing "make". If that does not work using the built-in capabilities of make, then you need to submit a properly configured makefile along with your source code. As always, you are free to develop wherever you wish, but the TA will be grading the programs based on their performance on the CS department machines.

What to Hand In:

1. Your code, **including a makefile if necessary and a readme file,** should be handed in electronically using Blackboard.
2. The purpose of the readme file is to make it as easy as possible for the grader to understand your program. If the readme file is too terse, then (s)he can't understand your code; If it is overly verbose, then it is extra work to read the readme file. It is up to you to provide the most effective level of documentation.
3. The readme file must specifically provide direction on how to run the program, i.e. what command line arguments are required and whether or not input needs to be redirected. It must also specifically document any optional information or command line arguments your program takes, as well as any differences between the printed and electronic version of your program.
4. If there are problems that you know your program cannot handle, it is best to document them in the readme file, rather than have the TA wonder what is wrong with your program.
5. If you have added any optional enhancements, make sure that they are well documented in the readme file, particularly if they require running the program differently in any way.
6. A printed copy of your program, along with any supporting documents you wish to provide, (such as hand-drawn sketches or diagrams) may be handed in **to the TA** on or shortly after the date specified above. Any hard copy must match the electronic submission exactly.
7. Make sure that your **name and your ACCC account name** appear at the beginning of each of your files. Your program should also print this information when it runs.

Optional Enhancements:

It is course policy that students may go above and beyond what is called for in the base assignment if they wish. These optional enhancements will not raise any student's score above 100 for any given assignment, but they may make up for points lost due to other reasons. Note that all optional enhancements need to be clearly documented in your readme files. The following are some ideas, or you may come up with some of your own:

- Throw Exceptions when problems arise, such as trying to follow a Direction that doesn't exist in the current Place, or trying to traverse a locked Direction.
- Use a negative keyPattern as a master key, by treating it as a bitmask. Use the absolute value of the keyPattern to mask out bits of the lockPattern, and if the result is zero, the key fits.

- Employ the mobility field of Artifacts to limit a players options. For example limiting the number, size, or weight of objects that can be carried (without a bag of holding), or having the player get tired if carrying a load.
- Implement health and/or hunger. Allow some artifacts to have nutritional or medicinal value.