In this assignment, you will combine the token and graphics classes to create a basic Holy Grail simulation. In this process, you will gain experience with parsing.

Abstract Classes
Make appropriate classes in your type hierarchy abstract.

Scene Methods
Using the methods of the previous project, add the following methods to your scene:

1. Move Approaching Knights in the X direction: A method that takes an int argument and makes the approaching knights move in the X direction by the amount of the argument.
2. Make the leading approaching knight or guard say: A method that takes a string argument and makes either the leading approaching knight or guard speak the string. The command should toggle between the two. The first time it is called, it should make the guard speak, the next time the leading knight, then the guard, and so on.
3. Pass the leading knight: A parameter-less command that makes the leading knight cross the bridge.
4. Fail the leading knight or the guard: If it is the turn of the leading knight (guard) to speak, then this command make the guard (knight) fall down.

For now you don’t have to worry whether the leading knight is in the standing area or not and whether 3 questions have been answered before the leading knight passes, though you are free to add these constraints.

Token List (Optional part from last assignment)
Define a class (and interface) to represent a collection of token-objects (which were defined in the token assignment). For now, make the collection a history. Later you might need to modify the collection to implement some extra credit features.
Input String with Token History (Optional part from last assignment)

Create another version of the stateful scanner class of assignment 7 (which built on the versions in assignment 4 and 3). Like this class, the new class has two properties. One of these properties is the independent editable property storing the scanned string. The second property, however, is no longer a string property. Instead, it is a dependent readonly property storing an instance of the token history class that contains all the tokens in the scanned string (in the order these tokens were scanned). You cannot subclass the stateful class of assignment 7, as a subclass cannot hide a property defined by the superclass. Simply create a new version of this class by copying and then editing the original code.

For extra credit, reuse the code in the previous version to write the new version (without using inheritance).

Parser

Use inheritance to extend the scanner class above to create a processor/parser of user commands. Thus, your extending subclass transforms the superclass scanner into a parser. Your extension does not have to add any new properties. Instead you should change (a) the constructor of the class to take a reference to the scene object, and (b) override the setter of the input string so that it not only scans the string and fills the token collection (as before) but also interprets the input string as a user command.

The parser class should support the following commands for invoking appropriate methods of the Simulation object. The syntax for invoking these commands is given below:

- `<Command>` → `<Move Command>` | `<Say Command>` | `<Passed Command>` | `<Failed Command>`
- `<Passed Command>` → `passed-token`
- `<Failed Command>` → `failed-token`
- `<Say Command>` → `say-token quoted-string-token`
- `<Move Command>` → `move-token number-token`

Thus, a command is a move, say, passed, or failed command. A passed (failed) command consists of the single passed (failed) token. The move command consists of move token followed by a number token. The say command consists of the say token followed by the quoted string token. The syntax of the tokens was defined in the previous assignment.

If you support the + and - tokens, then you have to use a slightly modified grammar:

- `<Move Command>` → `move-token <Number>`
- `<Number>` → `number -token`
- `<Number>` → `+token number-token`
- `<Number>` → `-token number-token`

Each of the commands in the grammar corresponds to a simulation method above. When your parser matches a user command, it should call the corresponding simulation method.
For instance, if the input string is set to: “move 50”, the setter should call the move method in the simulation object (passed in the constructor) with the argument 50 (after setting the other two dependent properties of the object). Naturally, you should not rescan the input string to add this functionality, but instead process the token collection filled by the super class setter. Do not make the simulation a property of the parser. You are free to change the scanner if you need to but do not put any parsing in that class. If your scanner constructor calls the setter of the input string, this will result in the invocation of the overriding setter in the parser, which may cause problems. It can call a scan method that is not overridden by the parser and is called also by the scanner setter of input string.

Extra Credit
1. Parse optional sign tokens in move distances (e.g. +50 -50 50)
2. Allow the user to enter a sequence of commands in the input string (e.g. move 300 say “What is your name” say “Lancelot”)
3. Do recursive descent parsing. The amount of credit you get for this part will depend on whether you did 1 or 2 also or not.

Refresh Command
To demo the simulation, write a main method that creates and displays (using ObjectEditor) an instance of the scene object and an instance of the parser object, where you have passed the simulation object to the constructor used to instantiate the parser object.

Thus, your display will include multiple ObjectEditor windows – a parser window and a simulation window. Actions made in one window do not automatically update the display of other windows. However, you can invoke View→Refresh command to manually refresh a window. Later we will learn how to fix this problem.

Constraints
Your parser can and should be using instanceof to process tokens created by the scanner.