CSC236 L3: ADT Beetle Game Implementation

This is a lab designed for individual or pair programming work.

Note that this lab was created by Dr. Jan Pearce of Berea College.

- **This lab may be done individually or in a team of 2 students (i.e with no more than one partner)** and you may discuss the work with the professor, the TAs, and even your classmates. **However,** making any copies of your work or allowing anyone else to copy your work is and will be treated as **academic dishonesty**.
- You will be committing your source code on Github Classroom and submitting this document to Moodle.

Enter your name(s):

Setup

If you are working with a partner, make sure to fill out the following table as to who is doing what.

Team Roles	**Member Name**
First Driver/ Second Navigator	
First Navigator/ Second Driver	

The Beetle Game Explained

Beetle is a game of luck involving a die and a play style similar to Hangman in which each player tries to finish making a full beetle before the other players. This game is turn based, and for each turn, the rules are:

1. A single 6-sided die is rolled. Depending on the number, a body part to a beetle can be added as shown:
    a. A roll of 1 is for a body and only 1 body is allowed.
    b. A roll of 2 is for a head, only 1 head is allowed, and a body is required.
    c. A roll of 3 is for two legs, only 6 legs are allowed, and a body is required.
    d. A roll of 4 is for an eye, only 2 eyes are allowed, and a head is required.

e. A roll of 5 is for a feeler, only 2 feelers are allowed, and a head is required.
f. A roll of 6 is for a tail, only 1 tail is allowed, and a body is required.

2. If the beetle already has the maximum number of a certain body part, the player's turn is over without adding anything and we go to step 5.
3. If the beetle does not have another part that is required (i.e. a head for an eye), the player's turn is over without adding anything and we go to step 5.
4. If a part can be added, it is added to the beetle, and we go to step 1 .
5. Switch to the other player's turn.

In terms of gameplay in general, your program is to do the following:

1. Display whose turn it is, and when the current player's turn is over, it displays the message that the player's turn is over and switches to the other one.
2. Display what the Beetle looks like for a particular player after each successful roll of the die and body parts are added. **As specified later, make sure you have the logic of adding parts BEFORE you attempt this part**. While checking the body adding functionality, it will be enough to have a message about what part was added for each successful die roll. For example, you can start with a display like the following. **This is simply an example. Do something different!**

```
Player 1's turn
A 1 is rolled.
A body was created and added. POOF!
A 5 is rolled.
The beetle needs a head, so nothing was added. Switch.
Player 2's turn
A 3 is rolled.
The beetle is missing a body, so nothing was added. Switch.
Player 1's turn
A 2 is rolled.
A head was created and attached to the body. POOF!
A 1 is rolled.
A body already exists, so nothing was added. Switch.
```

3. Allow for a human to play against the computer. If you want to enable multiple players to play against each other, feel free **once you have completed the minimal requirements**.

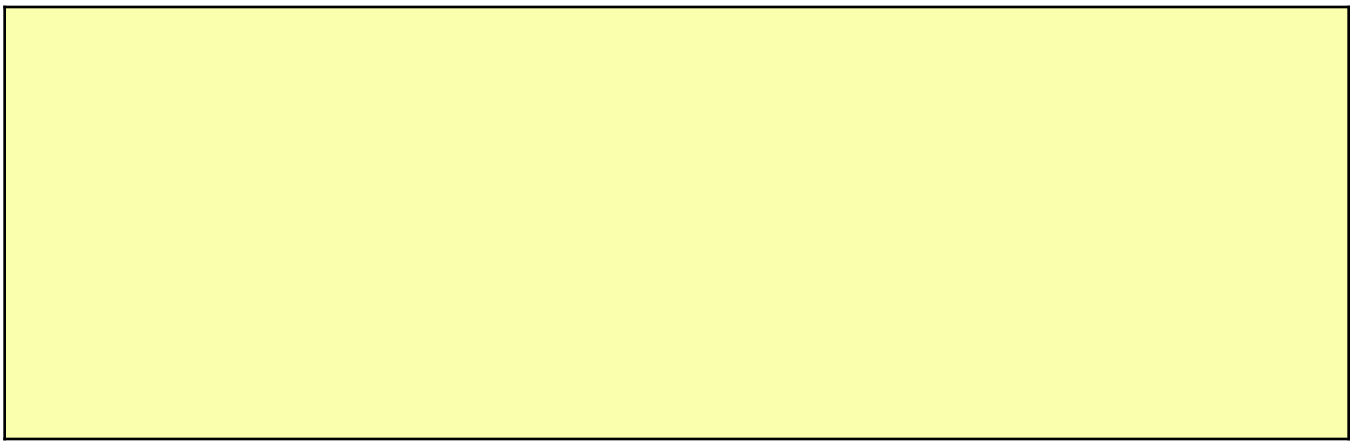
Milestone 1: Initial Design Document

The design portion of this document (until Implementation Steps) is due on Wednesday

High Level Design:

1. Write a **HIGH LEVEL** design flow chart to describe what your program is going to do. You should include high level processes such as “how to roll the die and give a result”,“how to add the correct body parts”, and so forth. **Be sure you refrain from going down to a low level that resembles lines of code of any kind. Instead, stick to descriptors which might refer to larger coherent blocks of code**. Some ideas you may want to carefully consider:
    a) How can your program have one player and then the other take turns and stop when the game is over?
    b) How can your Beetle keep track of its state of how many and what kind of body parts it has?
    c) How can your program incorporate the idea of rolling the die and adding body parts to the beetle? One option is to have the player roll the die and then explicitly add parts to the beetle. We will use a different approach in this lab; the beetle rolls its own die and based on the outcome, add parts to itself (or not). The user would simply call a method of the **`Beetle`** class called something appropriate like `play()`.

    Note that a flow chart style that makes your program visual like https://www.gliffy.com/sites/gliffy/files/image/2020-07/image-blog-ten-tips-flowchart.jpg can really help at this point.



Developing the Beetle Class - Attributes

Given the flowchart above, you are to start designing the **`Beetle`** class as an ADT. Remember that in OOP, classes have attributes (variables) and behaviors (methods), and these should be **directly related** to the class. The attributes are given by **nouns**, like feeler, head, name, and so forth. Each player has their own Beetle object so that they could have different body parts, and each Beetle has its own **`Dice`** object that it rolls to determine what body parts to attach.

2. Discuss the **instance and class variables** in your **`Beetle`** class and what data type they should be. These variables should all be **private**. Please include a variable for the Beetle object's name and an object instance of the **`Dice`** class. If you plan to use an array to store information about its body parts, discuss the advantages and disadvantages of using them.

Developing the Beetle Class - Methods

The **`Beetle`** class should have at least the following methods:

- A constructor that initializes all the attributes so that the Beetle object has no body parts.
- A friend function that overloads the "<<" operator to output the Beetle object to the `cout` stream.
- Getter functions that return information about the state of the Beetle object, such as whether it has a body and so forth. For example, a function like `get_body()` can return true if it has a body. Be careful about what the return type of these functions are.
- A method that causes the Beetle to roll its die and then adds body parts to itself based on the outcome and whether it can.

Other methods depend on how you want the Beetle to behave. For example, you may find having a method like `add_head()` that adds a head may make sense, or you may want a more general purpose method like `add_part(partNum)` that takes as input the appropriate body part that the Beetle adds to itself. Another thing to consider is WHERE you want to place the logic to decide whether a body part can be added. Inside the individual `add_<bodyPart>()` or some other method? There are many ways to design this program, so make sure you think carefully about what behaviors to add!

3. List the methods and a brief description of what they will do:

CRC Cards

Now you are ready to put everything together into a CRC card.

4. Write the specifications for each attribute and method of the **`Beetle`** class based on functional abstraction. Note that even if you are only using one class, we are still using the CRC cards because they are standard practice in design. Here, you are using the **`Beetle`** and **`Dice`** classes, but because the **`Dice`** class is given to you, you can outline just the **`Beetle`** class.

Class Name:	`Beetle`
Class Attributes	**Class Collaborations**
`1.`	
Class Methods	**Class Collaborations**
`1.`	
Friend functions	**Collaborations**
`1.`	

Design Collaborations

5. List all of the people who helped you think through the design, briefly describing their input.

Reflection

6. Describe how creating a design plan such as we did here before implementing might save a team time in the programming and debugging process

---

Implementation Steps

Update and submit this document when the full lab is due.

It is very tempting to jump right into writing your program, but take your time. A common and effective practice is to make small and incremental improvements. That way, you can check your program as you develop it and ensure that at each point, you have something that works.

We have some starter code for you in this repo **L04-Beetle Game**. Let us first focus on how the game will flow in the `main()` function.

There will be two `Beetle` objects, one for the player and the computer, but how the game proceeds and ends is up to you. Any changes to the Beetle objects must be through their methods; **you cannot**

change their attributes directly except through methods. How complicated this function looks depends on what Beetle behaviors you designed.

7. Detail the logic of this main function and what it will do.

Method Prototypes in the Class Definition

A **method prototype** (aka function or method interface) is a declaration of a function that specifies the function's name and type signature (data types of parameters and return type), but omits the function or method body. The prototype typicialy also contains a docstring that specifies both pre- and post-conditions when they are not obvious.

For example, a prototype from the **`Dice`** class can read like:

```
Dice
public:
int get_sides() const  // how many sides this die has
                    // postcondition: return # of sides of die instance
```

Do you see how it details the name, parameters (in this case, there are none), and the return type? The comment at the end states what the function **does** but NOT **how** it works.

8. Describe specifically how creating prototypes might help a pair of programmers equitably share implementation work.

Displaying Your Beetle using ASCII

Once you feel that your Beetle game is working correctly (adding body parts as it rolls its die), it is time to tackle how to display the Beetle object itself. The starter code contains ASCII art of the complete Beetle in the `complete_beetle` array that the `show()` function simply outputs into the `cout` stream. Use `complete_beetle` for reference for how the final image should look. Your program should change `partial_beetle`, which is initially blank.

9. Discuss how you can use appropriate indices from `complete_beetle` to modify `partial_beetle` in a targeted way as the game progresses.

10. Suppose you want to change the ASCII image of an arbitrarily large Beetle using n rows in `partial_array`. Discuss in the space below the complexity difference between indexing a specific position in the array directly to change the image versus copying the entire array while modifying the string at a single position.

Implementation Plan

Next, list who will implement each of your task components by what date. Be careful to keep this equitable if you are working in a pair.

Team Member Name	Milestone date	Subtask needed

Additional Requirements

DESIGN:
Please keep the above design plan components up to date which meets the lab requirements as you make the following required changes to your code.

REQUIREMENTS

Your program must have good structure and style and also do the following:

- Implement the Beetle game correctly and designed in a modular fashion, correctly using Beetle as an ADT with correct parameter passing and appropriate use of return values.
- Include a descriptive header as a comment at the top of your source code which include the names of all authors, the assignment name and purpose, and a statement that this is the work of the authors except where specifically documented.
- Use only meaningful variable and method names.

- Include a descriptive docstring for each method when the purpose is not clear, and, it should have appropriate pre- and post-conditions whenever these are not clear.
- Display the partially built Beetle as play continues

Make sure to include a fully updated `README.md` with all assistance acknowledged.

To submit

- **Milestone 1**: Complete the design section of this document as your initial design document, download it, and upload it to Moodle by the Milestone 1 due date and time. Pull your repo, make at least one commit.
- **Milestone 2**: Make a sincere attempt to complete your program so that you can ask questions. Ensure you sync your repo to GitHub. Some parts may not be working.
- **Total Completion**: Complete the implementation section of this document, download it, and upload it to Moodle. Complete the README.md file by replacing all the FIXMEs with your responses, then commit and push your repository to Github when this project is due..