

Software Architecture

Hand-in Assignment

Week 4

Instructions:

- Submit your solutions through GitHub Classroom and **remember** to submit a link to your repository in BrightSpace. Otherwise, I will have a hard time distinguishing between the different groups.
 - When submitting, you should push your solutions to the existing repository. In this assignment you only need to add code where "**Implement this**" is stated.
 - In this assignment there is a possibility to test your solutions with **autograde.py**. I hope it works on your machine. If not, you should spend some time setting up Python and CMake. Again, it's optional. Regardless, it's a huge advantage to read and understand the two test files in order to implement.
 - You **must** add comments in your programs to explain your code. Any solutions submitted without any comments will not be graded.
-

Description

In the lecture several design patterns were introduced. You have already used one these in the previous exercise when injecting the log object into the calculator. This is in fact a strategy pattern which is a very commonly used pattern.

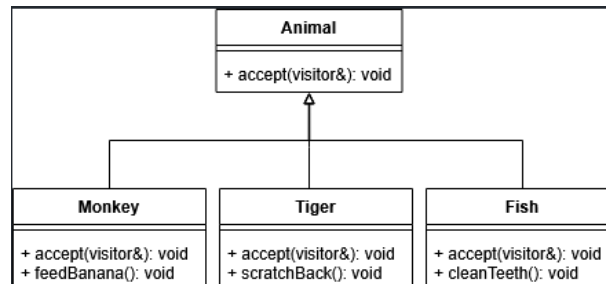
Today's exercise covers two new patterns; the **Visitor Pattern** and the **Observer Pattern**.



Exercises

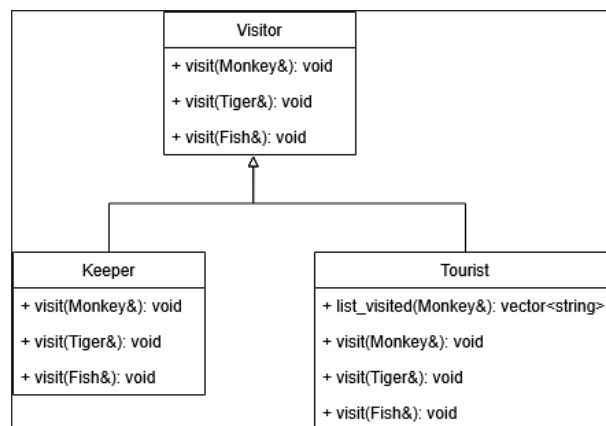
Visitor Pattern:

The intent of the visitor pattern is to decouple an algorithm from the data structure it operates on. We use the example of a zoo to elaborate this. A zoo houses several animals see each with concrete operations that are relevant only for the specific animals. In the context of the pattern these animals are the data structure which is acted on by the algorithm.



Keeping the animals fit and healthy requires that a zoo keeper visits these animals to take care of needs on a daily basis. The keeper is in this case the algorithm which acts upon the data structure by feeding, scratching backs, etc.

In the following parts we will implement the classes just described.



(0) **Read** the instructions from Page 1 please :)

(1) **Examine the** `test_animals.cpp` **test case**

Make sure that you understand the relation between the accept and visit methods.

(2) **Implement the** `accept` **method of the** `Monkey`, `Tiger` **and** `Fish` **class.**

This should call the visit method of the visitor with an concrete instance of an animal. Hint, use `this` keyword.

```
1 // do double-dispatch like this
2 visitor.visit(*this);
```

(3) **Re-run the test to verify that the first section of the test is passing (If you got the tests to work)**

(4) **Implement the `Tourist` class**

Implement the `visit` method(s) of the `Tourist` class. Whenever visiting an animal this should add store the name of the lower-case name of the animal in the vector `visited`

(5) **Re-run the test to verify the second section of the test is passing (If you got the tests to work)**

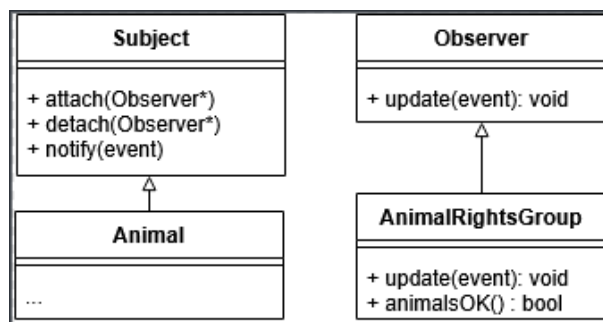
(6) **Implement the `Keeper` class**

- Implement the `visit` method(s) of the `Keeper` class. Unlike the tourist who just observes, the keeper must interact with the animals to fulfill their needs.
- When visiting a `Monkey`, call `feed_banana()`.
- When visiting a `Tiger`, call `scratch_back()`.
- When visiting a `Fish`, call `clean_teeth()`.

NOTE: that the `Keeper`'s interactions will be put to use and tested in the upcoming Observer section!

Observer Pattern

Our zoo has been attracting unwanted media attention; tourist have fed the monkeys, keepers have forgotten to scratch the tigers backs. To avoid further media attention we must keep track of who visits the animals and what actions they perform.



This is a perfect situation to use an observer pattern. In this case we consider the animals subjects and implement an observer `AnimalRightsGroup`. The animal rights group monitors the animals to see that the zoo treats all animals according to their needs.

(0) **Read** the instructions from Page 1 please :)

(1) **Examine** the `test_observers.cpp` test case

(2) **Implement** the `attach` and `detach` method of the `Subject` class

- `attach` should add a pointer to an `Observer` to the `subscribers` vector
- `detach` should remove the pointer

(3) **Implement** the `notify` method of the `Subject` class.

This should call the `update` method of every subscriber, like in the pseudo code below:

```
1   for s in subscribers:
2       s.update(event)
```

- (4) Implement the `feed_banana`, `scratch_back` and `clean_teeth` method of the `Monkey`, `Tiger` and `Fish` class.

```
1 void Monkey::feedBanana(visitor &v)
2 {
3     notify(AnimalEvent::monkeyFed);
4 }
```

- (5) Implement the `update` method of the `AnimalRightsGroup`.
Depending on the type of event recieved this should update the number of times the animals have been fed, scratched, etc. Seek inspiration from the `animals_ok` method.
- (6) Run the tests again to verify that they all pass (If you got the tests to work)