- Assignments must be completed individually.
- No late assignments will be accepted.
- Provide **concise** answers to the following questions. Use **point form** whenever possible.
- Submit your completed solutions to **Crowdmark**.

[2] 1. The Lecture Notes state that the OO paradigm is the best software development methodology invented so far. However, it is still not perfect. Briefly describe one problem associated with the use of the OO paradigm.

[4] 2. Give an example of a module which has **information hiding** but not **encapsulation**. Briefly explain why your example is correct.

[2] 3. (a) Briefly explain the difference between
   i. a module having encapsulation, and
   ii. an **abstract data type**.

[4] (b) Give an example of an abstract data type which is **not** one given in the text, the Lecture Notes or the Lecture Slides. Briefly explain why your example is correct.

[6] 4. Draw a class diagram which displays the following classes and the relationships among them.

- `Vehicle`
  - Properties:
    - `modelName`
    - `manufactureDate`
  - Methods:
    - `drive`
    - `park`
- `Car` (a derived class of `Vehicle`)
  - Methods:
    - `replaceBrakeFluid`
- `Bicycle` (a derived class of `Vehicle`)
  - Methods:
    - `replaceChain`

[3] 5. (a) Provide an example of **dynamic binding** which is **different** from the `Open` method of the `File` class given in the text and in the Lecture Notes and Lecture Slides.

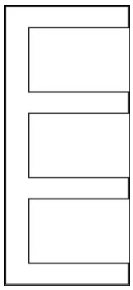
[2] (b) Dynamic binding provides many advantages for programming. However, it also has its shortcomings. State two problems associated with the use of dynamic binding.

[2] (c) Briefly explain the difference between dynamic binding and polymorphism.

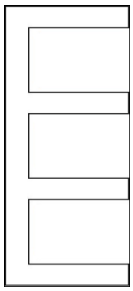
[4] 6. At the start of the discussion of Chapter 8 in the Lecture Notes, we identified two broad categories of software re-use. Give the name of each category, together with one example of a situation in which it occurs which is **not** one given in the text, in the Lecture Notes or in the Lecture Slides.

7. This problem is about **Library Re-use** (part a) and **Application Framework Re-use** (part b). In each part,
   i. shade in the area in the given diagram that represents the part that **gets re-used** under the given scheme,
   ii. give the name of the re-used part (represented by the shaded area), and
   iii. give an example of this type of re-use.

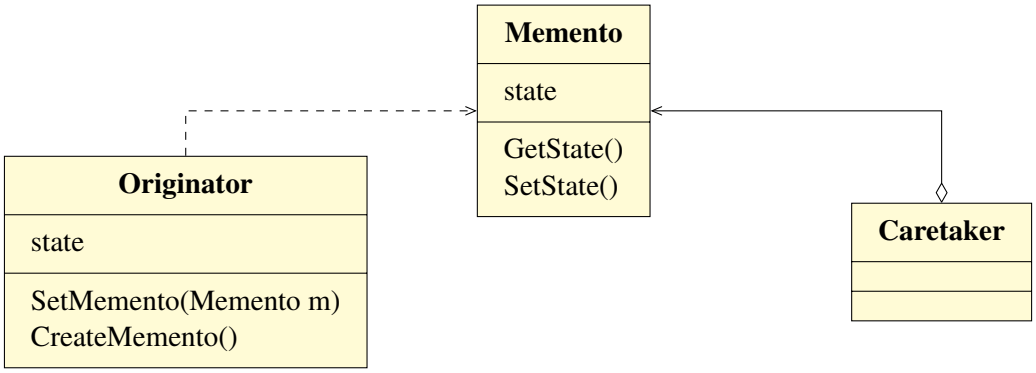
[3] (a) Library Re-use



[3] (b) Application Framework Re-use



8. This question is about the **Memento design pattern**, which is on the list of the 23 patterns from which the text examples are chosen, but is not one of the patterns included in the text. The pattern is described here.
Motivation: Sometimes it is necessary to record the internal state of an object. This is required when implementing checkpoints and "undo" mechanisms, that let the user back out of tentative operations, or recover from errors. You must save state information somewhere, so that you can restore the object to its previous state. But an object typically hides some or all of its state, making it inaccessible to other objects, and thus impossible to save externally. Exposing the state would violate the principle of information hiding.
Class Diagram:

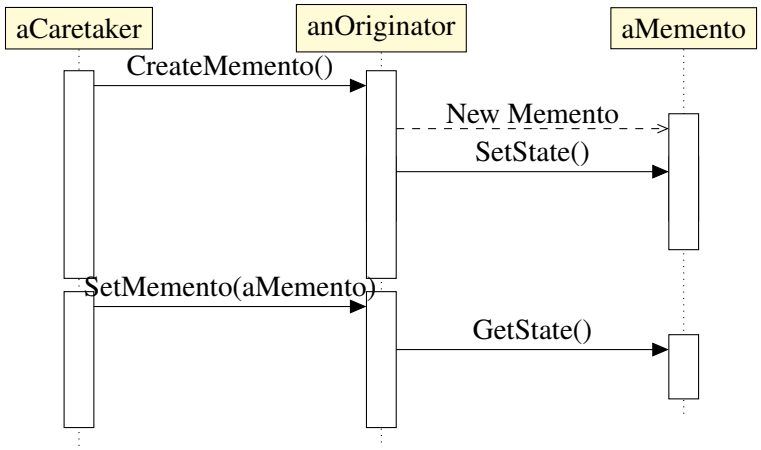


Notation: - - -> for "Creates" and ——> for "References".

Method Pseudocode

(a) `SetMemento(Memento m)`
```
state = m -> GetState()
```
(b) `CreateMemento()`
```
return new Memento(state)
```

Interaction Diagram



- A `Caretaker` requests a `Memento` from an `Originator`, holds it for a time, and passes it back to the `Originator`, as the diagram indicates.
- Sometimes the `Caretaker` will not pass the `Memento` back to the `Originator`, because the `Originator` might never need to revert to an earlier state.

- `Memento`s are passive. Only the `Originator` that created a `Memento` will assign or retrieve its state.

In this question, we will work with the Memento pattern applied to a Visio-like application, for creating diagrams via drag-and-drop. To undo a move made by dragging a `Box`, we will use the `Memento` pattern to remember the original state of the `Box`, so that we can return the `Box` plus any incoming/outgoing arrows to their original positions.

[5] (a) Draw the class diagram for our application, in which

- `Originator` is replaced by `Box`,
  - `SetMemento(Memento m)` is replaced by `SetBoxState(BoxState s)`,
  - `CreateMemento()` is replaced by `CreateBoxState()`,
- `Memento` is replaced by `BoxState`, and
- `Caretaker` is replaced by `BoxUndo`.

[5] (b) Draw the interaction diagram for our application, in which

- `anOriginator` is replaced by `aBox`,
  - `New Memento` is replaced by `New BoxState`,
  - `SetMemento(aMemento)` is replaced by `SetBoxState(aBoxState)`,
  - `CreateMemento()` is replaced by `CreateBoxState()`,
- `aMemento` is replaced by `aBoxState`, and
- `aCaretaker` is replaced by `aBoxUndo`.

[3] (c) The primary motivation to use the Memento design pattern in this question is to preserve the **information hiding** of the `Box` object. Give three reasons why information hiding is a desirable ingredient in effective object-oriented programming.