COSC427: Advanced OO Design

Exam 2006

Time allowed:	**2 hours**.
Number of pages:	**5**.
Number of questions:	**7**.
Total marks:	**100**.
Resources:	**Open book**: You may use any printed or electronic resources. You may not communicate, directly or electronically, with other people. If you want to print anything, ask a supervisor to collect it from the printer for you.
Answers:	Write all answers in your **answer book**. There is no spoon.

1. **[4 marks for whole question]** Provide a name (and if possible a wiki or web page reference) for each idea (a)-(d):

   (a) [1 mark] An approach for executing parse trees.

   (b) [1 mark] It is better to use abstract classes than concrete ones.

   (c) [1 mark] Design documentation is essential.

   (d) [1 mark] A pattern containing a class called `Quantity`.

2. **[28 marks for whole question]** For each of the following descriptions (a) – (g):

   i. **[1 mark each]** Name a *maxim* (or *pattern*, *code smell*, etc) that captures the idea.
   ii. **[1 mark each]** Explain the rationale for this *maxim* (i.e. why the *maxim* exists).
   iii. **[2 marks each]** Comment on the validity and value of this *maxim*.

   (a) [4 marks] Domain model objects should not access user interface objects.

   (b) [4 marks] A class should not access its subclasses.

   (c) [4 marks] If a class is unlikely to change, it should be abstract.

   (d) [4 marks] Don't initialise attributes in constructors; instead, initialise them the first time a getter is called.

   (e) [4 marks] Structure code so that a `foo()` method in some object calls `foo()` methods in other objects.

   (f) [4 marks] A maxim violated by the Observer pattern.

   (g) [4 marks] Don't write getters that return fields that can be changed.

3. **[5 marks]** If you could nominate one person to receive a lifetime achievement award for services to OO design, who would you choose, and why?

4. **[5 marks]** Some designers suggest that it is a good idea to always declare separate interfaces from implementations, so that no class ever uses an implementation class directly; instead it uses only abstract interfaces. Is this a good idea? Justify your answer.

5. [23 marks] Following criticism of the Frogs Design, the designer has studied design patterns and tried again. Figure 1 is the class diagram, and Figure 2 provides notes. Find as many Gang of Four patterns as you can in this new design. Name each pattern and identify where its major features (classes, attributes, methods and relationships) occur in this design. Provide just enough information to convince the marker you really know how the pattern is realised in this design. Briefly comment on the value of each pattern in this context.

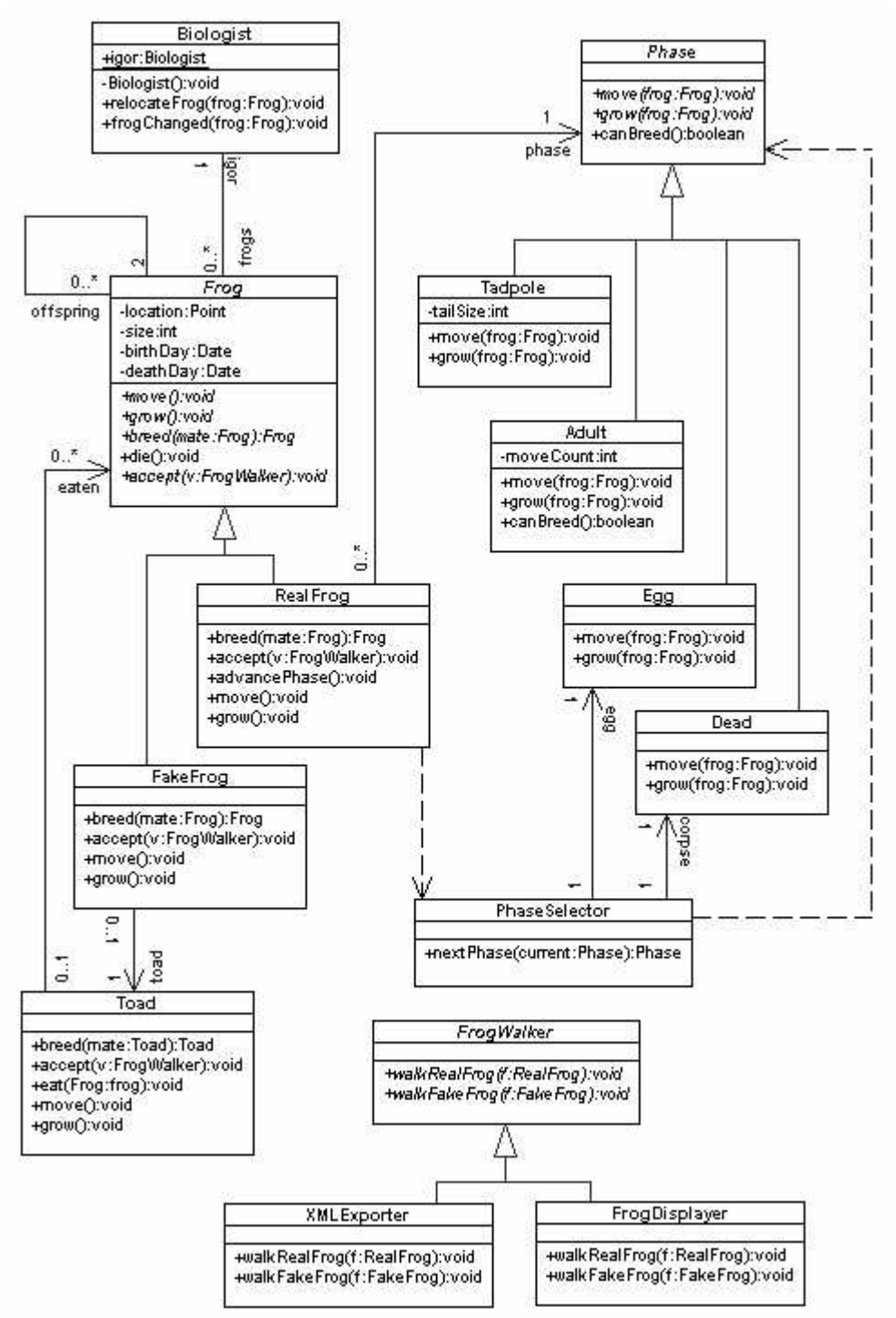

Figure 1: Frog class diagram

- As before, this design models the lifecycles of `Frogs`.
- Getters and setters are omitted from the diagram, but may be assumed where necessary.
- There is one `Biologist`, Igor, who manages the `Frogs`. Igor moves `Frogs` around as required for experiments.
- There are many `Frogs`. The design minimises `Frogs`' knowledge of each other, instead using Igor to coordinate `Frog` behaviour. Whenever a `Frog` changes state, it tells Igor by calling `frogChanged()`, and Igor may choose to `move()` `Frogs`, or tell them to `grow()`, `breed()`, etc.
- A `Toad` can masquerade as a `Frog`, because Igor can't tell the difference. `FakeFrog` makes a `Toad` conform to the type of `Frog`.
- Igor doesn't eat `Frogs` any more, ever since he accidentally ate a `Toad`.
- A `RealFrog` can `advancePhase()` from an `Egg` to a `Tadpole` to an `Adult` to `Dead` by changing its `Phase` object.
- `Phase` objects are provided by a `PhaseSelector`. For every tadpole or adult `Frog`, `PhaseSelector` makes one `Tadpole` or `Adult` phase object. However, it uses a single `Egg` instance and a single `Dead` instance for all `Frogs`, because these two `Phases` don't need any data except `birthDay` and `deathDay`, which can be retrieved from `Frog`.
- A `Frog` can `breed()` to make `offspring` of the appropriate type.
- Tasks such as exporting XML and displaying `Frog` information are separated into `FrogWalker` subclasses. A `Frog` object can `accept()` a `FrogWalker` and call the `walkRealFrog()` or `walkFakeFrog()` method as appropriate for that `Frog`. It then tells `offspring` `Frogs` to `accept()` the `FrogWalker`.

Figure 2: `Frog` notes

6. [10 marks] What conflicts or differences of opinion can be detected between the ideas of ArthurRiel1996 and KenAuer1995? Do the underlying philosophies of the authors differ? Justify your answers, and where possible, support them with evidence from the papers.

7. [25 marks] On your first day in a new software engineering job, you're shown the `Account` class (Figure 3), which represents foreign exchange accounts that hold money in a particular currency. It records all exchanges of money from one `Account` to another, using the current exchange rate to calculate the result. Your manager says:

> "It is pretty much complete – it was designed by a top consultant – but we need you to extend it to cope with exchange rates that vary over time. We need to retain all info, so we should be able to find out what exchange rate was used for any exchange, even if it happened a long time ago. We also need it to handle more than 100 `Accounts`."

Criticise the design (and implementation) and suggest how it should be improved in order to support the requested features and to fix problems the manager was unaware of. Where possible, support your arguments and explain your design with maxims, patterns, refactorings, etc, and provide UML diagrams. If you need to make any assumptions about requirements, make a note of what you assumed and why.

```
public class Account {

    public final int AUSTRALS = 0;
    public final int BOLIVIANOS = 1;
    public final int PESOS = 2;
    public final int SUCRES = 3;

    private int id; // Unique identifier of account

    private float amount; // Number of units of currency

    private int currency; // AUSTRALS, etc

    // Record all exchanges, indexed by src account & dst account.
    private static float[][] exchanges = new float[100][100];

    // Exchange rates, indexed by src currency & dst currency.
    // Rates should be symmetrical, so src & dst can be reversed.
    private static float[][] exchangeRates = new float[3][3];

    public Account(int n, Float a, int c) {
        id = n; // Set id to n
        amount = a; // Set amount to a
        currency = c; // Set currency to c
    }

    public void setRate(int c1, int c2, float rate) {
        exchangeRates[c1][c2] = rate;
        exchangeRates[c2][c1] = rate;
    }

    public boolean exchange(ForExAccount dst, float xAmount) {
        if (xAmount > amount)
            return false;
        amount -= xAmount;
        exchanges[id][dst.id] = xAmount;
        dst.amount += xAmount * exchangeRates[currency][dst.currency];
    }
}
```

Figure 3: `Account` **code**

END OF PAPER