TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS – 2012

END OF YEAR

SWEN425

DESIGN PATTERNS

Time Allowed: 3 Hours

Permitted materials:

Instructions:

- This examination will be marked out of **180** marks.
- Read each question carefully before attempting it.
- Answer all six questions. Each question has the same value, and should take approximately 30 minutes to answer.
- You may answer the questions in any order. Make sure you clearly identify the question you are answering.
- Many of the questions require you to discuss an issue, or to express and justify an opinion. For such questions, the assessment will take into account the *evidence* you present and any *insight* you demonstrate.
- Some of the questions ask for examples from object-oriented languages or of design patterns. Your answers need only refer to object-oriented languages or patterns discussed in the course, but you may refer to other programming languages and patterns if you wish.
- You may write code samples and/or draw diagrams to illustrate your answers to any of the questions.
- This exam is open book. Non-electronic reference books and handwritten notes are permitted.
- No calculators permitted

1. Creational Patterns **(30 marks)**

Josh Bloch has described how a Builder lets you create objects like this:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).
   calories(100).sodium(35).carbohydrate(27).build();
```

instead of using a constructor with many arguments:

```
NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);
```

or multiple setter calls:

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

(from *Creating and Destroying Java Objects*, Joshua Bloch, informit.com, May 16 2008)

(a) **(5 marks)** Explain in detail how the Builder works in this example. (Drawing an interaction diagram may help your explanation).

(b) **(5 marks)** Explain the advantages and disadvantages of a Builder in this design, compared with constructor- or setter- based designs.

(c) **(4 marks)** Explain how you would (or would not) expand this design to create multiple related products like an Abstract Factory.

(d) **(4 marks)** Explain how you could specialise the product of a Builder in the same way you can specialise the product of a Factory Method?

(e) **(4 marks)** Can a Builder be based on Prototypes? Explain why or why not?

(f) **(4 marks)** Should a Builder be a Singleton? Explain why or why not?

(g) **(4 marks)** Can a Builder build Flyweights? Explain why or why not?

2. Structural Patterns **(30 marks)**

Several patterns, including Chain of Responsability and Proxy, can use Smaltalk's dynamic reflexive "`doesNotUnderstand`" messages (or their equivalent in other languages, such as Ruby's "`method_missing`" or Python's "`__getattr__`".

(a) **(5 marks)** Sketch the implementation of a Chain of Responsibility using a dynamic reflexive mechanism like "`doesNotUnderstand`". (You don't have to use Smalltalk syntax!)

(b) **(5 marks)** What are the advantages and disadvantages of a "doesNotUnderstand" implementation, compared with a more straightforward implementation?

(c) **(5 marks)** Can you extend your implementation to work as an Adapter? Explain why or why not?

(d) **(5 marks)** Languages like Self, Groovy, and JavaScript support some kind of *delegation*, where an object additionally can "inherit" from another preëxisting object — not just from a class. Could delegation support these kinds of patterns? Explain why or why not?

(e) **(10 marks)** Patterns like Facade, Bridge, and Mediator are very "heavyweight" — they have a big impact on your design, almost working like large-scale components rather than small reusable objects. The actual Facade, Bridge, and Mediator classes introduced by these patterns are often very special purpose, ad-hoc, and customized. Do you agree with this characterisation? Explain why or why not. Should these patterns really be considered "anti-patterns" — that is patterns that you should **not** use in your programs? Or, if we must have Facades, Bridges, and Mediators, explain why. Are these patterns needed to handle legacy code, because of bad designs elsewhere, or for some other reasons?

3. Behavioural Patterns **(30 marks)**

Many Behavioural patterns make small objects that act almost like anonymous functions, or change other methods to accept those objects as arguments. Many object-oriented langauges now support lambda expressions (also called "blocks", anonymous functions, first class functions, delegates) — including Smalltalk, C♯, Java 8, C++11, Python, JavaScript, Ruby, and Dart.

For each of the patterns below, explain, giving examples, how they can be implemented using lambda expressions, and explain the advantages and disadvantages of that implementation. If the pattern cannot be implemented sensibly in this way, explain why.

(a) **(5 marks)** Internal Iterators

(b) **(5 marks)** External Iterators

(c) **(5 marks)** Command

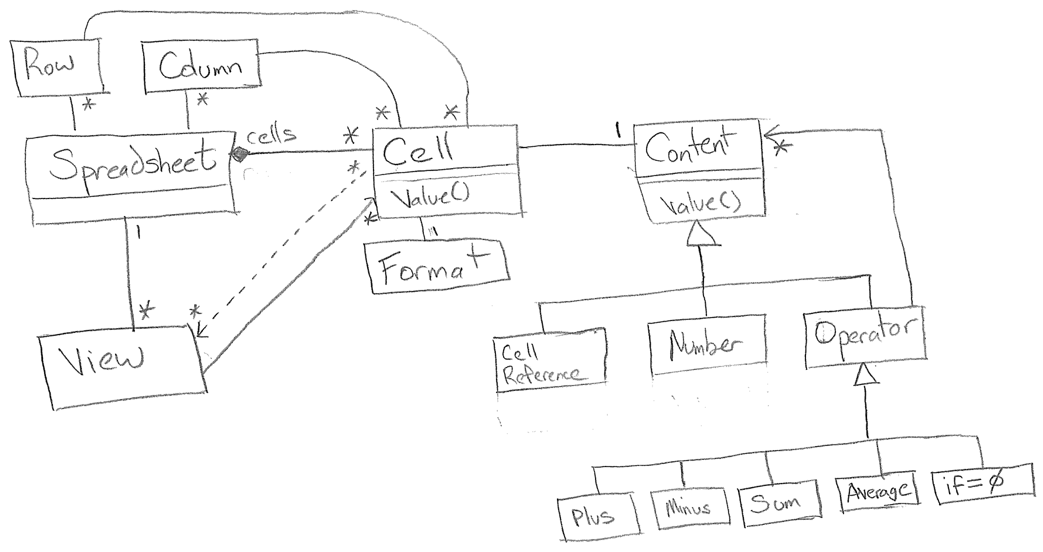
(d) **(5 marks)** Strategy

(e) **(5 marks)** Template Method

(f) **(5 marks)** Observer

4. Patterns in System Design **(30 marks)**

The following class diagram shows the design of a simple spreadsheet framework.



This design includes a number of design patterns.

Identify **three** patterns used in this design.

For each of these three patterns:

(a) **(1 mark)** Name the pattern.

(b) **(2 marks)** List each of the **Participants** of the pattern, and name the corresponding concrete class(es) in the design.

(c) **(7 marks)** Describe the design problem *in the spreadsheet framework* that the pattern solves, and explain why the pattern solves that problem.

5. Null Object **(30 marks)**

Design Patterns explains how the Strategy pattern can refactor switch-statements so that:

```
void Composition::Repair () {
    switch (_breakingStrategy) {
        case SimpleStrategy:
            ComposeWithSimpleCompositor();
            break;
        case TeXStrategy:
            ComposeWithTeXCompositor();
            break;
```

can be repalced by:

```
void Composition::Repair () {
   _compositor->Compose();
```

The **Null Object** pattern is a new pattern that refactors if-statements that test against "null" — or perhaps some more abstract "empty" or "missing" value — in exactly the same way, so that:

```
void processLateAdditions(Collection collection)
  if (collection == null) {
     System.out.println("Got no late additions");
  } else {
     System.out.println("Got "+collection.size()+" late additions");
  }
```

can be replaced by:

```
void processLateAdditions(Collection collection)
     System.out.println("Got "+collection.size()+" late additions");
```

Draft a *Design Patterns* style description of the **Null Object** pattern. You should cover the main sections of a pattern description: Intent, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, and Related Patterns; and should draw diagrams where appropriate.

6. Problems and Solutions **(30 marks)**

*The line about "**A Solution to a Problem in a Context**" is nothing more than a fairy story to tell to second year students. Patterns are just solutions — nothing more, nothing less.*

Attributed to Thomas J. "Tad" Peckish.

Is there more to a pattern than its solution? If you see a particular "solution design" in a program, can you always tell which pattern is being used? Is Java-style serialization always the Memento pattern, or Java-style cloning always the Prototype pattern? Is there any real difference between the Proxy and Decorator patterns? Is it important to keep these two patterns separated, or would it be easier if they were combined into just one pattern based on the common features of the two patterns' class diagrams? Discuss this question generally, giving examples of other patterns with closely related solutions, or that are often tied to particular language features.

* * * * * * * * * * * * * * *