PART IIA 2005
3F6: Software engineering and design
Principal Assessor: Dr T W Drummond

Datasheet: None

Solutions to 3F6 Software Engineering and Design questions, 2005

1. *Specification, software testing*

   (a) *Defect testing* largely checks that the software meets the *functional* requirements in the specification—it checks that the software functions in the agreed manner.
   Statistical testing is not concerned with finding defects, but in checking and quantifying the reliability of the software (e.g rate of failure). Reliability is a *non-functional* requirement—it is not concerned with the response of the system to inputs, but instead with its availability. [15%]

   (b) i. *Functional testing* checks that the software behaves according to its specifications by providing various inputs and checking the outputs. The structure of the program is not considered—the software is treated as a black box which simply has to respond correctly. *Equivalence partitioning* is a means of selecting test data for functional testing (since it is not usually feasible to try every possible input). Inputs are divided into valid and invalid ranges (partitions), and test cases are then chosen which are either typical values (well within the valid range) or possible error cases (on or near the boundary).
   In this case, invalid inputs would be when `a=0` and `b` is negative (since $0^{-n} = 1/0^n = \infty$ for positive n), and when `a` or `b` are such that the answer would over- or under-flow the `float` datatype. Appropriate test cases would make sure to include these problems, as well as typical examples. For example, five test cases would be:

   ```
   a=10        b=2
   a=-10       b=-2
   a=0         b=-1
   a=1000      b=10
   a=0.00001   b=-10
   ```
   [20%]

   ii. *Structural testing* makes use of access to the source code for the software to design carefully-targeted test cases and to ensure that all of the paths through the code is tested. It compares test program behaviour with the apparent intention of the source code.
   The program flow graph for this function is shown in Figure 1. Calculating the Cyclomatic Compexity:

   $$\begin{aligned} CC(G) &= \text{edges} - \text{nodes} + 2 \\ &= 8 - 7 + 2 \\ &= 3 \end{aligned}$$

   or simply by inspection of the program flow graph, there are three independent paths, corresponding to the cases when `b` is positive, negative or zero.

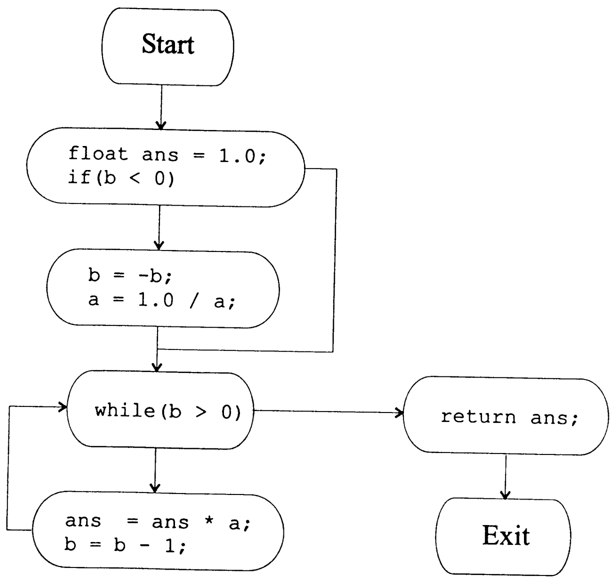


Figure 1: Program flow graph for `exp()` function

If positive, the `while` loop will be entered; if negative, the `if` statement is entered, and then the `while` loop; and if b=0 then neither of the conditional parts of the function are entered. Example test cases would therefore be:

```
a=10  b=-1
a=10  b=0
a=10  b=1
```

[30%]

iii. *Software inspection* examines the source code, without executing the software. Software inspection checks for common programming errors, and compares the source code with the requirements document and design diagrams to verify that the source code meets the specifications. Typically, over 60% of all program errors can be detected by software inspection, so this is a very useful stage in software verification. [15%]

(c) There are in fact several input cases which will cause problems in this function. The main error is when a=0 and b is negative, resulting in an untrapped divide by zero error. This error should be detected by functional testing (since the divide-by-zero case is one likely to be included by equivalence partitioning), and by software inspection (which would check that divide by zero errors were correctly handled). Structural testing is not guaranteed to find this error since there is no path to deal with divide by zero that needs testing.

Other inputs which could cause problems are when a and b are sufficiently large or small that ans overflows or underflows, and the particular case of the maximum negative number (b=0x10000....0) since with two's complement integers this number has no positive equivalent. The case of 0^0 is also potentially a problem, since the result is undefined (it could plausibly be either zero or one), but it is generally accepted that the most appropriate answer is $0^0 = 1$, which is what this function returns. [20%]

Examiner's remarks: This question was about software specification and software testing. This was a reasonably popular question, and there were some reasonable answers. Candidates clearly knew the theory behind the various types of software testing, but they found it more difficult to apply the theory to a concrete answer. Likewise, whilst most of them correctly identified the error in the function when asked to do so in the final part of the question, many did not appreciate when answering the earlier part of the question, partitioning, that the process they went through in finding the error by eye were similar to the process that was required in that form of testing, and so they should also have identified that particular error case in that part of the question as well.

2. *Object-oriented software design, concurrent systems, monitors, CORBA*

(a) For easily-maintainable software, it is desirable that it has low coupling and high coherency. Low coupling means that one software module does not depend on the specific implementation of another module—i.e. that replacement parts can be plugged in without breaking other parts of the software. High coherency means that all the parts in a module are related. The task of a module can then be easily identified, and modifications to a particular functionality only need to be made in one place. Errors can also be easily tracked down.

Object-oriented programming groups together program elements into objects, each of which represents a concept. An object has its own state (data) and functions, as opposed to function-oriented programming where data and functions are owned and accessible globally. Objects thus naturally provide high coherency. Objects have two parts: a declared public interface and a hidden private implementation. Any object which provides the correct interface can be used in a piece of software, i.e. the implementation should not be important, and objects thus have loose coupling. Much modern code is written using object-oriented design, and the code is in general easier to maintain and the objects easier to reuse. [20%]

(b) Monitors provide mutual exclusion automatically. Only one process thread is allowed within a monitor object at any one time, so if a member function on an instance of the monitor has been called, any other call to any function on that object will be blocked until the previous function call is finished.

Synchronisation is provided using *condition variables.* If a monitor object wishes to wait until some other thread signals that a condition is true, it executes the cwait(SomeCondition) command, which blocks the current thread and passes control to the next waiting process. When the event that the first thread was waiting for occurs, some other thread calls csignal(SomeCondition), which passes control immediately back to the first thread to continue processing. [15%]

(c) Figure 2 shows the monitor object. [25%]

(d) A CORBA IDL file simply defines the interface to the object, so needs only be based on the public interface in the monitor. There is no int data type in CORBA—instead long (or short) is used. Parameters must also be identified as either in, out or inout, depending on whether they are read, written to, or both. Figure 3 shows the IDL file. [10%]

(e) To extend a CORBA interface without breaking existing clients, one derives a new interface from the old one. This new interface will include a function to return the current number of values on the stack, as shown in Figure 4 [10%]

(f) To block when the stack is in an invalid state, the throw commands in the monitor need replacing with cwait commands on relevant condition variables, here called not_empty and not_full. csignal on not_empty is called every time an item is added to the stack, and csignal on not_full is called every time an item is popped from the stack. These can safely be called every time, since csignal is ignored if no process is waiting. They must be called as the last thing in the function, since the function immediately exits once csignal is called. Figure 5 shows the modified monitor. [20%]

Examiner's remarks: This question was about concurrent systems (particularly monitors) and CORBA. This was the least popular question on the paper, by some way, but was generally very well answered by those who attempted it. Most candidates showed a good understanding of object-oriented programming and the syntax of C++, and the use of monitors and CORBA. The 'sting in the tail', asking them to use condition variables for synchronisation, caused some difficulties, with a number of candidates remembering the wrong example from lectures (remembering mutual exclusion using semaphores instead). There were several candidates who failed to finish this question, presumably running out of time at the end of the exam.

```
monitor Stack
{
private:
   const int max_size = 100;
   int stack[max_size];
   int stackpos;

public:
   void push(int a);
   void pop(int& a);
};

Stack::Stack() : stackpos(0)
{
}

void Stack::push(int a)
{
   if(stackpos == max_size)
      throw StackOverflowError;

   stack[stackpos] = a;
   stackpos = stackpos + 1;
}

void Stack::pop(int& a)
{
   if(stackpos == 0)
      throw StackEmptyError;

   stackpos = stackpos - 1;
   a = stack[stackpos];
}
```

Figure 2: The Stack monitor object

```
interface Stack
{
   void push(in long a);
   void pop(out long a);
};
```

Figure 3: CORBA IDL file for the Stack interface

```
interface Stack
{
   void push(in long a);
   void pop(out long a);
};

interface ExtendedStack :Stack
{
    long size();
};
```

Figure 4: CORBA IDL file for the Stack interface, and the ExtendedStack interface, which adds a function to find the size of the stack.

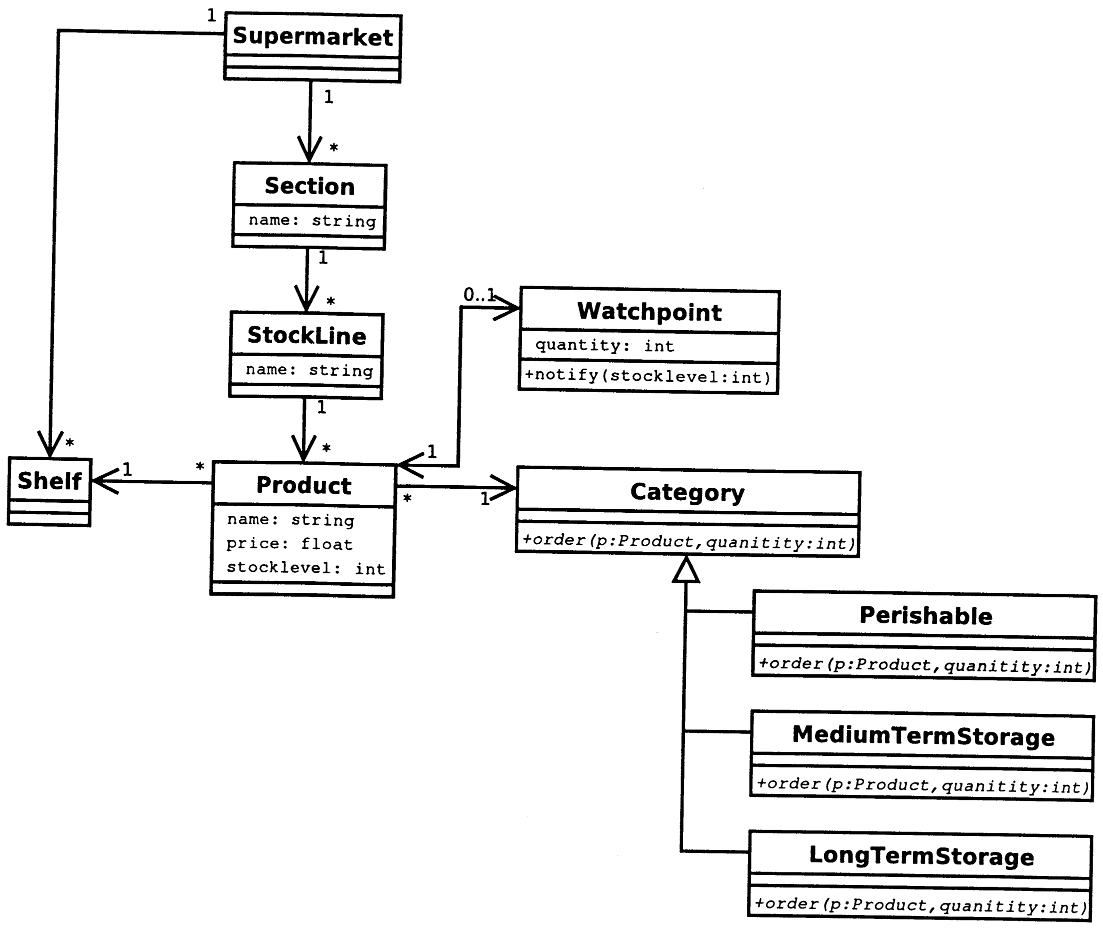
```
monitor Stack
{
private:
   const int max_size = 100;
   int stack[max_size];
   int stackpos;

   condition not_full;
   condition not_empty;

public:
   void push(int a);
   void pop(int& a);
};

Stack::Stack() : stackpos(0)
{
}

void Stack::push(int a)
{
   if(stackpos == max_size)
      cwait(not_full);

   stack[stackpos] = a;
   stackpos = stackpos + 1;
   csignal(not_empty);
}

void Stack::pop(int& a)
{
   if(stackpos == 0)
      cwait(not_empty);

   stackpos = stackpos - 1;
   a = stack[stackpos];
   csignal(not_full);
}
```

Figure 5: The `Stack` monitor object, modified to block on errors rather than throw exceptions.

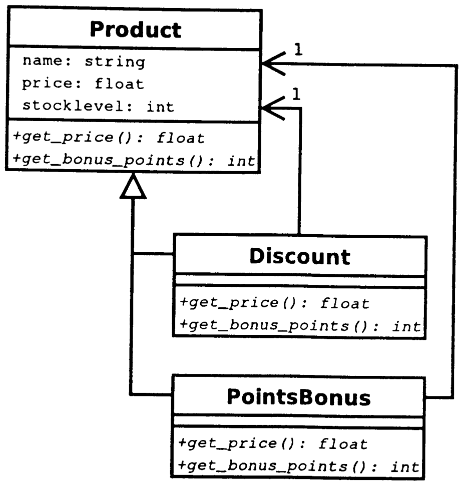
3. *Software design and design patterns*

(a) *Class Derivation* represents an is-a relationship between to classes. An object instance of the derived class can be treated as an instance of the base class and can be used wherever one of these is required. The derived class my also add state and functionality to the class. *Polymorphism* extends this by permitting the derived class to have its own implementation of a function that is declared in the base class.

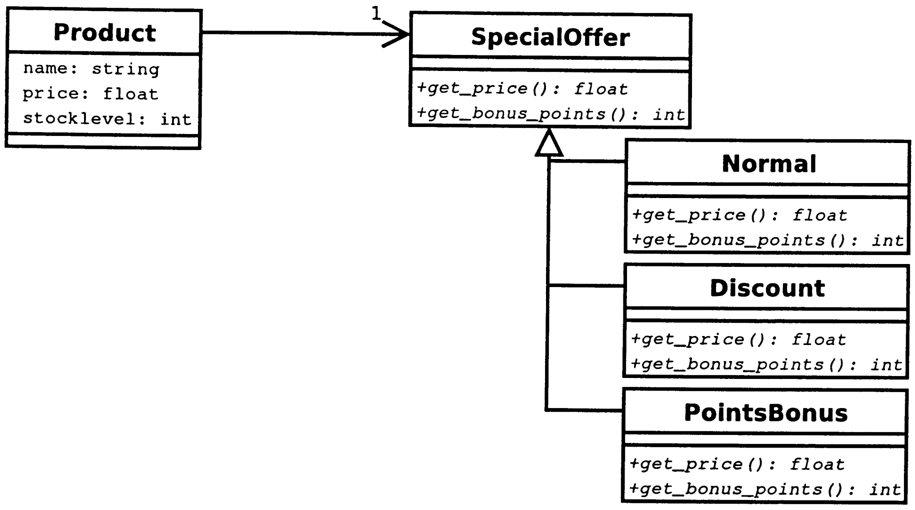
(b) The UML class diagram should contain at least the following components. Candidates may also have classes for Checkout and Customer. The order method in Category represents and example of polymorphism.



(c) There are two methods of doing this. It can be achieved either by use of a decorator pattern:



or by use of a state pattern.



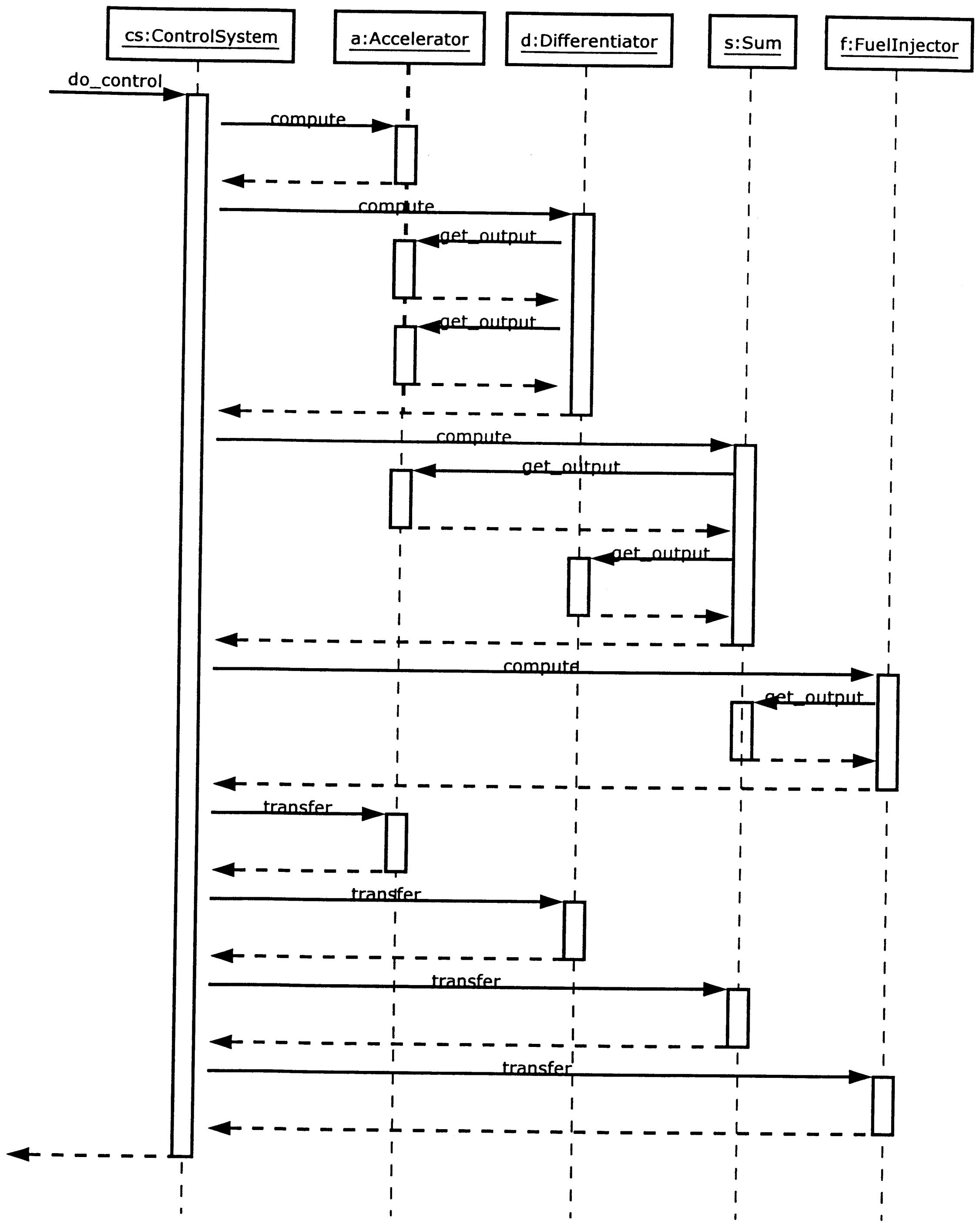
Candidates should also identify that Customer needs an attribute for loyalty points.

Examiner's remarks: This question was about converting a software specification into a design. This was (just) the most popular question on the paper and was generally well answered. Most candidates showed a good understanding of software design principles and were able to generate a good UML diagram showing their design. Better candidates avoided the temptation to introduce classes for each section of the supermarket (e.g. having a separate class for the Bakery). Many candidates also identified a good way of answering the final part although the completeness of the answers varied.

4. *Software design and design patterns*

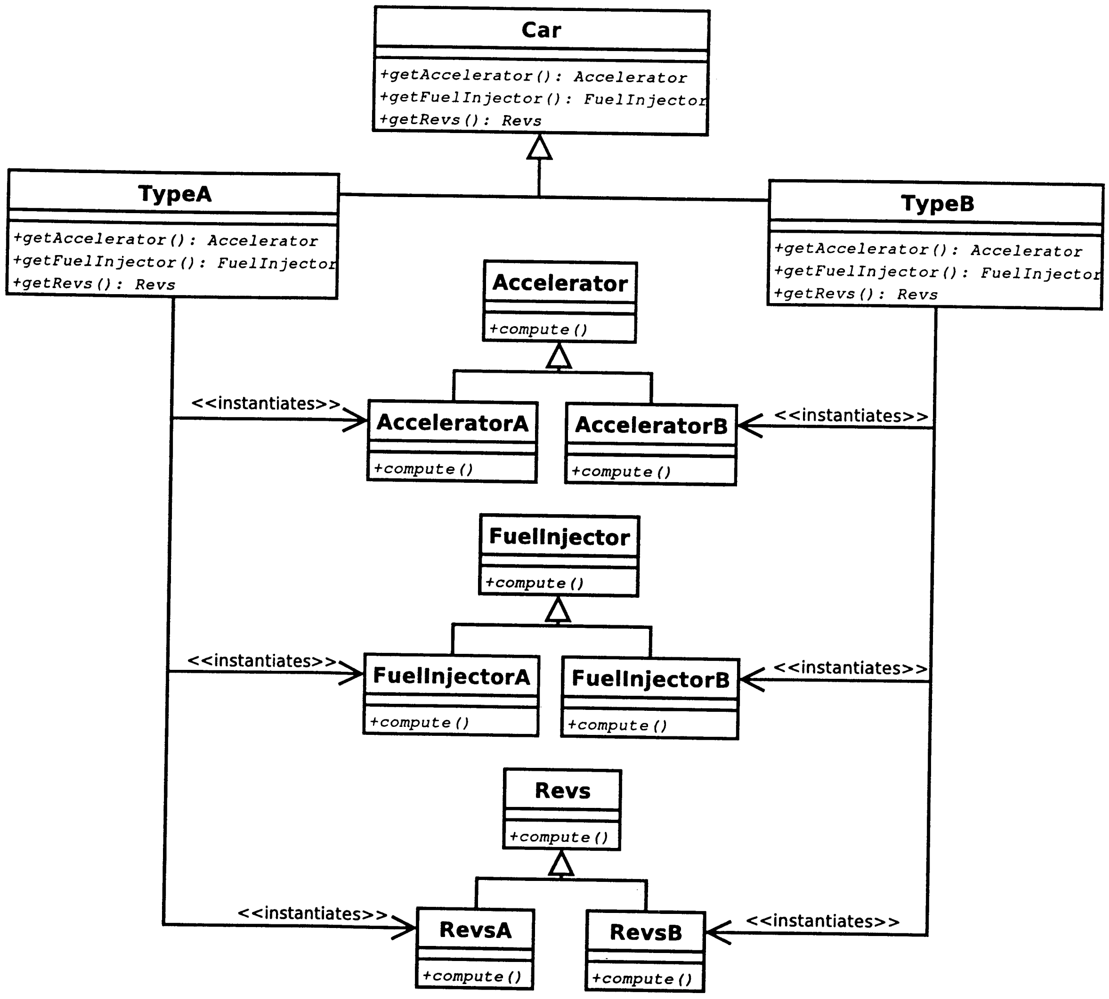
   (a) There are nine classes present: ControlSystem, Component, Integrator, Differentiator, Gain, Sum, FuelInjector, Acclerator and Revs. Each ControlSystem has many Components. Integrator, Differentiator, Gain, Sum, FuelInjector, Accelerator and Revs are derived from Component. Integrator, Differentiator, Gain and FuelInjector each have one Component as their input. Sum has two Components as its inputs.

   (b) Sequence diagram shown below


cs:ControlSystem
a:Accelerator
d:Differentiator
s:Sum
f:FuelInjector
do_control
compute
compute
get_output
get_output
compute
get_output
get_output
compute
get_output
transfer
transfer
transfer
transfer

(c) The order in which the components are added to the control system does *not* matter because they operate by reading outputs cached in the `stored_output` attribute. The newly computed outputs only become visible to other components after the `transfer` method is called. This means that the `do_control` method must be called three times before any effect from the accelerator pedal is seen at the FuelInjector. On the first time, the Accelerator object obtains the position of the pedal and stores the value in its output attribute. The `transfer` method makes this visible at the end of the `do control` method. On the next call, this value is seen by the Sum object and only on the subsequent (third) call to |`do_control`|is the value seen by the FuelInjector.

(d) The best way to do this is to use the *Abstract Factory* pattern. This involves creating an abstract factory class (called Car in the diagram below). This has two subclasses, one for each type of car. The job of the factory classes is to create the correct sort of Accelerator, FuelInjector and Revs objects for each type of car.



Examiner's remarks: This question was about understanding and manipulating an existing software design. The first two parts of the question were well answered by most candidates, with the majority producing a (mostly) correct sequence diagram. The third part of the question caused some difficulties, with may candidates assuming that the system being considered was identical to an examples sheet question, despite many of these being able to correctly deduce things from the differences. The fourth part of the question was well answered by the better candidates who correctly identified a good design pattern to be applied to the problem.