Exercise 5

Inheritance and Multiple Inheritance

1)

a)

i. In the `Matrix` class:

```
Matrix add(Matrix m) {
  if(m instanceof SparseMatrix) {
    // semi-efficient implementation
  } else {
    // old implementation
  }
}
```

In the `SparseMatrix` class:

```
Matrix add(Matrix m) {
  if(m instanceof SparseMatrix) {
    // efficient implementation
  } else {
    // semi-efficient implementation
  }
}
```

ii. In the `Matrix` class:

```
Matrix add(Matrix m) {
  return m.addMatrix(this);
}
Matrix addMatrix(Matrix m) {
  // old implementation
}
Matrix addSparseMatrix(SparseMatrix m) {
  // semi-efficient implementation
}
```

In the `SparseMatrix` class:

```
Matrix add(Matrix m) {
  return m.addSparseMatrix(this);
}
Matrix addMatrix(Matrix m) {
  // semi-efficient implementation
}
SparseMatrix addSparseMatrix(SparseMatrix m) {
  // efficient implementation
}
```

iii. In the `Matrix` class:

```
Matrix add(Matrix m) {
  // old implementation
}
Matrix add(SparseMatrix m) {
  // semi-efficient implementation
}
```

In the `SparseMatrix` class:

```
SparseMatrix add(Matrix m) {
  // semi-efficient implementation
}
SparseMatrix add(SparseMatrix m) {
  // efficient implementation
}
```

Note that, in all approaches, the implementer does not have to write the semi-efficient implementation twice. For example, the method `SparseMatrix.addMatrix` in the Visitor pattern can be implemented as follows:

```
Matrix addMatrix(Matrix m) {
  m.addSparseMatrix(this);
}
```

This solution applies to commutative operations, such as the addition of matrices. We cannot do that for the multiply operation.

b) The last approach is probably the simplest and most intuitive.

c) For the first and last approaches, all that would be lost is the potential extra efficiency when adding a `SparseMatrix` to a `Matrix`. However, for the second approach (Visitor pattern) it's essential to be able to add the extra methods to the superclass, in order to make the second dispatch possible. Whatever the approach to binary methods, if the `add` method in `Matrix` had been written using direct field accesses on its argument (rather than calls to `get()`) then it will need to be rewritten anyway when the subclass is added.

2)

a) The receiver can be immediately returned from such a call. We could overload:

```
Matrix add(ZeroMatrix m) {
  return this;
}
```

However, in a language like Java, which does static dispatch re: argument types, this will not have the desired effect when a `ZeroMatrix` instance has a less specific static type.

b)

i. In the `Matrix` class:

```
Matrix add(Matrix m) {
  if(m instanceof ZeroMatrix) {
    return this;
  } else if(m instanceof SparseMatrix) {
    // semi-efficient implementation
  } else {
    // old implementation
  }
}
```

In the `SparseMatrix` class:

```
Matrix add(Matrix m) {
  if(m instanceof ZeroMatrix) {
    return this;
  } else if(m instanceof SparseMatrix) {
    // efficient implementation
  } else {
    // semi-efficient implementation
  }
}
```

In the `ZeroMatrix` class:

```
Matrix add(Matrix m) {
  return m;
}
```

ii. In the `Matrix` class:

```
Matrix add(Matrix m) {
  return m.addMatrix(this);
}
Matrix addMatrix(Matrix m) {
  // old implementation
}
Matrix addSparseMatrix(SparseMatrix m) {
  // semi-efficient implementation
}
```

In the `SparseMatrix` class:

```
Matrix add(Matrix m) {
  return m.addSparseMatrix(this);
}
Matrix addMatrix(Matrix m) {
  // semi-efficient implementation
}
SparseMatrix addSparseMatrix(SparseMatrix m) {
  // efficient implementation
}
```

In the `ZeroMatrix` class:

```
Matrix add(Matrix m) {
  return m;
```

```
}
Matrix addMatrix(Matrix m) {
  return m;
}
SparseMatrix addSparseMatrix(SparseMatrix m) {
  return m;
}
```

iii. In the `Matrix` class:

```
Matrix add(Matrix m) {
  // old implementation
}
Matrix add(SparseMatrix m) {
  // semi-efficient implementation
}
Matrix add(ZeroMatrix m) {
  return this;
}
```

In the `SparseMatrix` class:

```
SparseMatrix add(Matrix m) {
  // semi-efficient implementation
}
SparseMatrix add(SparseMatrix m) {
  // efficient implementation
}
SparseMatrix add(ZeroMatrix m) {
  return this;
}
```

In the `ZeroMatrix` class:

```
Matrix add(Matrix m) {
  return m;
}
SparseMatrix add(SparseMatrix m) {
  return m;
}
ZeroMatrix add(ZeroMatrix m) {
  return this;
}
```

c) We are forced to require specific implementations for many more cases than we originally thought of, in order to ensure that there is always a most-specific fit for any pair of receiver and argument type. The definitions in **bold** above are the extra ones added for this reason.

The extra requirement seems somewhat annoying for this example, particularly since in all cases where an ambiguity would otherwise arise, the choice of implementation does not intuitively affect the actual result. For example, if we erased the **bold** definitions, then for a `ZeroMatrix` receiver and `ZeroMatrix` argument we would have to choose between the `ZeroMatrix-Matrix` implementation, and the `Matrix-ZeroMatrix` implementation. However, both of these return the non-zero matrix. On the

other hand, consider the case when we have a `SparseMatrix` receiver and a `ZeroMatrix` argument. In this case, we have to choose between the `Matrix-ZeroMatrix` implementation and the `SparseMatrix-SparseMatrix` implementation. But it is not completely obvious that the latter would work correctly for a `ZeroMatrix` argument, depending on its implementation (how much it depended on the appropriate fields from `SparseMatrix` being used/initialised as expected).

d) In the light of this, there seems to be less to choose between the last two approaches. One further observation though is that in the case of multiple dispatch, although the superclass has been modified, it is only for an improvement in efficiency – if it were essential that the superclass were unchanged then the `Matrix-ZeroMatrix` implementation could be omitted from the code above, and everything would work out fine. The other approaches depend upon being able to modify the superclass, which may not always be acceptable in practice.

e) The second approach (Visitor pattern) doesn't require any changes to the existing classes. The other two approaches would have to relinquish the extra efficiency possible when the argument is a zero matrix (but could still be efficient when the receiver was a zero matrix).

3) The following C++ code breaks the invariant:

```
class B : public Person
{ public: B (string n) : Person (n) {} };
class C : public Person
{ public: C (string n) : Person (n) {} };
class D : public B, public C
{ public: D (string n) : B(n), C(n) {} };


void marryMyself ()
{
     D me ("Me");
     B *b = &me;
     C *c = &me;
     b->marry (c);
     if (b->getSpouse ()) cout << b->getSpouse ()->getName ();
}
```

The object `me` contains an object of class `B` and an object of class `C`. The addresses of these objects are different and they are obtained using the assignments to `b` and `c` respectively. During the call `b->marry (c)`, the condition `p == this` compares these two addresses and finds them not equal.

4) The following code breaks the invariant:

```
class Girl : virtual public Person
{
public: Girl () : Person (true) {}
     // diamonds are a girl's best friend
};

class GirlProgrammer : public Girl, public Programmer
{
public: GirlProgrammer () :
     Person (true), Girl (), Programmer () {}
};

void oops ()
{
       GirlProgrammer gp;
}
```

Following the rules of C++ virtual inheritance, the call of the constructor `Person (true)` in class `GirlProgrammer` bypasses the corresponding call `Person (false)` in class `Programmer`, breaking the invariant.

5) Here are the three requested classes:

```
class Queue
{
    int[] contents;
    int size;

public:
    Queue() { contents = new int[100]; size = 0; }
    void enqueue(int x) {…}
    int dequeue() {…}
    int getSize() { return size; }
};

class SumQueue : virtual public Queue
{
    int sum;

public:
    SumQueue() : Queue() { sum = 0; }

    void enqueue(int x)
    {
           sum+=x;
           Queue::enqueue(x);
```

```
    }

    int dequeue()
    {
        int r = Queue::dequeue();
        sum-=r;
        return r;
    }

    int getSum() { return sum; }
};

class ProductQueue : virtual public Queue {…};

class SuperQueue : public ProductQueue, SumQueue
{
public:
    SuperQueue()
        : public Queue(), ProductQueue(), SumQueue() {}

    void enqueue(int x)
    {
        ProductQueue::enqueue(x);
        SumQueue::enqueue(x);
    }

    int dequeue()
    {
        int r = ProductQueue::dequeue();
        SumQueue::dequeue();
        return r;
    }
};
```

One obvious problem is that the `enqueue` and `dequeue` methods of the superclass are called twice. An item is enqueued and dequeued twice. Interestingly, this behaves exactly like a queue, but the capacity is half of the capacity of the original and the `getSize` method reports the correct size multiplied by 2.

We can use traits and linearization to ensure that the enqueue/dequeue methods are called only once. Here is the relevant Scala code:

```
class Queue
{
    …
    def enqueue(x:int) = {…}
    def dequeue():int = {…}
}

trait Sum extends Queue
{
    var sum:int = 0
```

```
    override def enqueue(x:int) =
          { sum+=x; super.enqueue(x) }
    override def dequeue():int =
          { var x = super.dequeue; sum=sum-x; x }
}

trait Prod extends Queue
{
    var count:int = 1
    override def enqueue(x:int) =
          { prod*=x; super.enqueue(x) }
    override def dequeue():int =
          { var x = super.dequeue; prod=prod/x; x }
          // side remark: this assumes no zeros in the queue!
}
```

Now, an object of `Queue with Sum with Prod` has both functionalities, but calls each underlying enqueue/dequeue method only once. The problems of the multiple inheritance solution do not appear here.