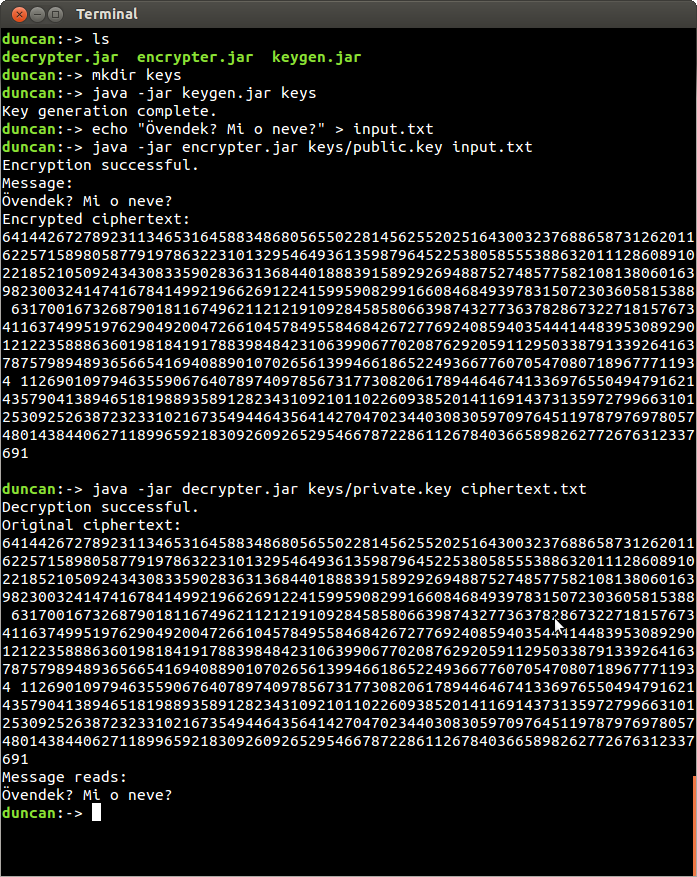
RSA Encryption Implementation

My solution consists of three executable .jar files:

* keygen.jar: generates a private and public key pair. It takes a single command line argument that specifies a path to the directory into which to write the public and private keys. This path is relative to the location of the jar. If no argument is supplied it writes the files to the same location as the jar. The files written are ‘public.key’ and ‘private.key’.
* encrypter.jar: takes an input file and encrypts it using a given public key. It writes the result as ‘ciphertext.txt’ to the same location as the jar file. It takes two arguments specifying where the key is and where the input file is. If no arguments are supplied the program looks for files called ‘public.key’ and ‘input.txt’ in the same location as the jar.
* decrypter.jar: takes a cipher text file and a private key file and prints the decrypted message and ciphertext to the screen. It also writes the decrypted message to a file called ‘output.txt’. It takes two arguments specifying where the private key is and where the ciphertext is. These default to looking for files called ‘ciphertext.txt’ and ‘private.key’ in the same location as the jar.

The operation of the three files from a Linux terminal is documented below:



The .jar files, keys, input and output can be accessed in the ‘final’ directory of this submission.

The classes that make up the system model reflect five responsibilities:

* key generation
* decryption
* encryption
* string parsing
* file handling

My first attempt at the problem was to understand the key generation algorithm I had been asked to implement. Work began with a group discussion with several fellow students in which we worked out the steps the key generator would have to complete to produce the keys:

Set e to 65337

Are e and phi coprime?

Find phi

phi = (p-1)(q-1)

Yes

No

Generate p and q

Multiply p and q to get n

Find d using Euclidean Algorithm

This inventory of the steps involved suggested I create a set of functions, each individually testable, which would combine to produce the keys.

# Tests

I completed this function-writing portion of my work by writing tests first, then writing code to pass those tests. I began by using the examples we had been given to extract values for p and q as input, along with what values were expected for phi, n, d and e as output. I decided that all my functions would handle the integer primitive type rather than BigInteger objects, and that I would translate the algorithms to use BigIntegers later. This helped me keep my code more readable and intuitive while I was trying to get the algorithm to pass the tests.

It was agreed during our group discussion that the easiest function to start with seemed to be Euclid’s algorithm, since we were given a recursive implementation in the UCS lecture notes. I translated Euclid’s algorithm gcd() function into Java, and ran my tests until my code was producing the output given in the examples.

# Extended Euclid and Multiple Return Values

The Extended Euclid function is another recursive function, which given e and phi, produces, amongst other values, the multiplicative inverse of e. This becomes d, one part of the private key. I found a wikibooks article that described the algorithm and provided some example code in Python. I translated this implementation, along with the algorithm supplied in the UCS lecture notes, into Java. I also added the section which translates the multiplicative inverse into d.

When writing this function I hit the problem of returning multiple values from a function in Java. The Extended Euclid function returns three values, which must be passed to the next stage of the recursion. There are two options available: return the values in a collection object, such an Array or HashMap, or create a dedicated class to express a value object made up of those values. Packaging in an array is a less readable approach since the variables get put into anonymous positions in an array, rather than being given human-readable labels. In this case I wrote each value into an array and returned the array, then unpacked the values into separate variables because this seemed less cumbersome than creating a class just for returning Extended Euclid results, since all the values were primitive integers.

Once I had completed these functions I immediately started adding Javadoc comments to the top of my functions. These supplemented my function names, so I would remember later what they did, and I also kept a record of the resources I used to arrive at these functions.

I wanted to be sure I had the correct key generation process in place. To do this I needed an integration test, but before I could write this I needed to write some string manipulation functions that would translate characters into integers. This stage was the first time I started using the BigInteger object, since the large size of the integers produced would necessitate using this data type. Again I wrote the tests first, looking up a Unicode character table to determine what numbers should be produced given short input strings:

I now had two functions:

BigInteger stringToUnicodeNumbers(String string)

In Java, casting a char primitive to an int produces the Unicode character code as an integer. stringToUnicodeNumbers()uses this, and adds extra zeros to each integer so that each character is represented by a five digit integer.

String numberToString(BigInteger b)

The only tricky part of numberToString() was remembering to pad with enough leading zeros so that the integer splits separately into five digit chunks. These integer chunks can then be cast back into chars.

At this point I refactored my key generation functions to use the BigInteger type.

I wrote an integration test that used all my functions to generate a key, encrypt a short message and decrypt it back. The actual encryption and decryption was done using the built-in BigInteger .modPow() function. The test passes if the decrypted message matched the original message.

As well as confirming my code worked as expected, this test highlighted a limitation with the RSA algorithm. The algorithm will only map a finite field of numbers to another finite ciphertext field, because it uses modular arithmetic. Numbers higher than n will wrap around and be mistaken for another number in the field. I mitigated this problem by increasing the bit length of p and q, which increases n and allows longer integers, and therefore characters, to be represented. This also increases the computation time, so I added a splitMessage() function which split the message into seven character chunks. Each chunk was encrypted separately, allowing me to use smaller values for p and q, yet still encrypt longer messages.

<splitMessage>

Up to this point I had approached the problem in a procedural style, with each step given its own function. I had a single Encrypter class with a lot of functions on it, plus some higher level functions that called these step functions to complete the process:

encrypt()

pickPQandPhi()

getD() (Extended Euclid)

splitMessage()

stringToUnicodeNumbers()

decrypt()

partsToBigInts()

decryptArray()

numberToString()

The only higher-level structure that had emerged from my work so far was a value class called encryptedMessagePair. This stored the message, plus d and n, and would be passed from the encrypt process to the decrypt process so that the private key was preserved along with the message. This was far from ideal as in practice the private key was to be stored separately to the message, and I had no provision for storing the public key. Now I had the algorithm implemented and that implementation verified, a refactoring stage was needed.

The worst form of refactoring is one without any stated purpose. I learnt this as my first attempt was to create a Message class, which stored all the key generation values, plus the private key and the public key all in one object. After another look at the problem brief, and a high-level design discussion with a work colleague, I realised that this did not really solve the problem, and that I should split my functions into classes based on the principle of single responsibility.

I looked again at the problem brief and split the functions across the three main responsibilities of the system: key generation, decryption and encryption. I noticed that supporting these domain responsibilities were two other responsibilities: string parsing and file handling. Once these responsibilities had been separated from the domain problem of encryption and decryption, the code became more modular.

# Refactoring: Classes for each responsibility

## RSA Encryption

I have modelled the encryption and decryption process as follows:

BigInteger C

BigInteger M

PublicKey.decrypt()

BigInteger M

BigInteger C

PrivateKey.encrypt()

I decided to model the actual encryption/decryption process purely in terms of BigIntegers. Public and Private Keys can be constructed directly by the KeyGenerator object, or from a String.

## Encoding

Encoding refers to the process of translating a String into a BigInteger, and vice versa. The main challenge was handling the fact that there are in fact four processes: both directions for both encryption and decryption. To model this I created a CipherText class and a PlainText class, which have toBigInteger() and toString() methods. Internally, these classes store the string and convert to BigInteger when needed. These classes can be constructed from either a BigInteger or a String, and they handle their conversion internally.

## Unicode Encoding

The encoding process should be kept as flexible as possible, so I created a Unicode factory class that produces ciphertexts and plaintexts that are used to store the results of encryption and decryption. For example, a different ciphertext or plaintext factory could be used, which produces classes that handle ASCII encoding. The encode-encryption process is handled by the Encrypter and Decrypter classes.

Unicode (implements Encoding)

cipherText(String)

cipherText(BigInteger)

plainText(String)

plainText(BigInteger)

BigInteg

CipherText

toString()

toBigInteger()

PlainText

toString()

toBigInteger()

**Applying the Process for a Long String**

To handle long strings, my program splits the string into separate sections of a fixed width, and then encrypts/decrypts each section separately. To support this, I created two classes: an iterator called OperationOverString and an interface called StringOperation. OperationOverString is an abstract class with two concrete implementations: one for splitting a string by a separator (OperationOverDelimitedString), and one for splitting into sections of a maximum length (OperationOverModularString). Once split, both classes use makeString() to apply a StringOperation over a string. StringOperation is implemented by the Encrypter and Decrypter, so their encrypt/decrypt operations can be applied by OperationOverString.

**File Input/Output**

File IO is complicated in Java, and I wanted to hide this complexity as best to reflect the file requirements of this system. I created a SimpleFile class, which handles writing and reading a given string to a file. It defaults to using UTF-8 encoding to create new files.

# Refactored Tests and Bug Detection

I also had to refactor my integration tests, which uncovered some implementation bugs, particularly in the StringParser class. This was because the code for string manipulation could now be tested separately, where previously it had been an extra process hanging on to the encrypt() and decrypt() functions.

For example, StringOperations.splitAtInterval() went through several implementations, as my first attempt was very buggy. It had problems handling zero-length strings and strings that only took up one chunk, producing String arrays with extra null elements, which would then trip up subsequent functions. Because the functions had been refactored out into separate concerns, the problem was easily isolated and unit tests then helped me solve the issue quickly.

Finally, I created the three command line tools that form my complete RSA implementation. This proved to be fairly easy after the refactor, since each process could be expressed in terms of the functions and classes I had created.

The group discussion helped me to understand the stages of the algorithm, and discussion with a more experience developer helped add purpose to the refactoring process. Test Driven Development helped immensely when implementing the algorithm, and helped me uncover bugs in my implementation later in the development process. Each stage could be built up with confidence as the previous stage had proven to be producing the right output.

Wikibooks article: http://en.wikibooks.org/wiki/Algorithm\_Implementation/Mathematics/Extended\_Euclidean\_algorithm

Article concerning finding e part of public key: https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture12.pdf