COMP 303 Study guide

Francis Piche

September 25, 2018

Contents

Part I
Introduction

1 Disclaimer

These notes are curated from Professor Joseph Vybihal's COMP303 lectures at McGill University. They are for study purposes only. They are not to be used for monetary gain.

2 About This Guide

I make my notes freely available for other students as a way to stay accountable for taking good notes. If you spot anything incorrect or unclear, don't hesitate to contact me via Facebook or e-mail at `http://francispiche.ca/contact/`

Part II
Writing Good Code

Good code is:

- Optimal (Time complexity and memory)
- Simple
- Correct
- Robust (Few crashes, good error handling etc.)
- Easy to read
- Is well documented
- Uses accepted engineering techniques.

3 Strategies for Writing Good Code

3.1 Optimality

To minimize memory usage we could:

- Encode long strings of data (compressing)
- Re-calulate instead of store values (trade-off with speed)
- Check library overhead!

- Data structure overhead (graph has a lot more pointers than a linked list!)

To minimize time complexity:

- Come up with a better algorithm (Improve big O)
- Store more things in RAM (rather than in a DB or over a network)
- Avoid nesting of pointers
- Avoid deep or unnecessary recursion

3.2 Simplicity

Strategies for simplicity:

- Good variable names (avoid single letters except for array index's)
- Don't use lots of variables when an array is more appropriate
- Use simpler data structures if possible
- Reduce the volume of code
- Limit line length
- Modularize the program
- Use algorithms that are well known if possible

3.3 Correctness

The only way to be somewhat sure that your program is correct is through rigorous testing. Some things are easier to test than others, and theres entire QA departments that specialize in testing. But with the rise of Agile, more and more standard Devs need to be good at testing.

Some general ideas on testing:

- 1 test per 1 function
- Test valid inputs
- Test invalid inputs
- Test edge cases
- Validate that the output is correct

There's a ton of material online about testing and I definitely recommend taking some time to learn about more in-depth testing methods than what is covered in this course. It's required in almost all dev jobs and it's a very valuable skill to have.

3.4 Readability

This comes from good indentation, spacing and general style of code. Most people know this implicitly so I won't go into detail. For examples see the lecture slides from Lecture 2.

3.5 Comments as documentation

The first place documentation happens is on the code level. This is where you can remove any ambiguity or sources of confusion from your code for other developers (or even future you). While real projects require real documentation, and excessive comments can be a detriment to readability, more comments are generally better than not enough. Break down your complex algorithms into comment separated steps, or add side-notes on anything that could be confusing to another developer.

4 Well Designed Objects

A well designed object should be one that does not cause any "wut?" moments amongst a team of developers. Specifically:

- Single class per file
- Single purpose
- Expose only essential information
- Support an API structure
- Follow appropriate inheritance methods

4.1 Single Purpose

A class should be used to represent an idea, concept or object (pun-intended), and nothing more. A student should only contain ideas directly relevant to a student. A `Car` class should not have information about busses or trucks.

This ensures that someone using your class can quickly find out everything there is to know about your object in an intuitive way. If your Student class has information about apples buried inside somewhere, someone on your team would have a hard time finding that out.

4.2 Restriction of Information

This is done to ensure that the class will be used correctly. There should be a decent amount of thought put into which parts of the code are "internal" and "external" to the class, ie: what parts of the class does the program need easy access to?

Any time you use the `public` keyword, you should be thinking twice about if it's truly needed.

4.3 Encapsulation

When you take the concepts of single purpose and restriction of information and put them together, you get *encapsulation*. Essentially what this means is that your class should be it's own complete bubble. No code outside the class should do the same things as inside your class, nor should things outside affect the class in an uncontrolled way.

4.4 Inheritance

To avoid duplication of code, we use inheritance to relate objects. This way classes can share common elements and simplify our lives. For details on inheritance see my COMP250 guide.

Note that private information from a parent is not visible t the child.

To design inheritance well:

- A parent is more general
- A child is more specific

4.5 Pre & Post Conditions

To keep things coherent, if a parent object imposes a condition on data, the child should maintain this condition. For example if a parent object has the condition that `salary` $>$ `0` then the child should not violate this by overriding the condition with say, negative values. It could, however, override it with `salary` $>$ `10000`.

This is known as a **rich** starting point. Another example of a rich starting point is using a library, or some sample code.

THis might be an issue since you may be inheriting, importing or implementing more features than are really needed.

4.6 Extending vs Wrapping vs Interfaces

Extending is when a new class contains the parent but adds extra methods and variables. Allows for polymorphism (see 250 guide for more on that).

Interfaces are used when the classes implementing are not necessarily related, but share common methods. For example the `Iterable` interface can be implemented by say, a `Degree` class, which contains a list of courses. But also a `Queue` class which has nothing to do with degrees or courses conceptually. So this allows for polymorphism across multiple inheritance trees.

Wrapping is not a formal construct in a language, but it is the idea of placing objects inside a "wrapper" class, to put objects together in a more abstracted way.

A special case is `abstract base classes`. These cannot be instantiated and contain both implemented and non-implemented methods. They provide the inheritance properties of extending, with the templating of interfaces.

5 Object Identity & Lifecycle

All objects have an idenity,(a reference or anonymous) and a lifecycle. Their lifecycle depends on the language and how they are handled (in C/C++ you have to manually free them from memory, vs garbage collection)

5.1 Referenced Obects

This is when an object is created by assigning its instance to a variable. `Object o = new Object()`. In this way, when no variables remain that point to this object, it is garbage collected (in Java) or causes a memory leak (C/C++).

5.2 Anonymous Objects

These are objects without references. For example `fn(new Object())`. It is then assigned a reference in the scope of the function, or in the scope of the Object to which it was passed into. For example if you pass an object into the constructor of another object, if it is then assigned to a variable in the second object, it will die with the object it is inside of.

5.3 Static vs Instance

A static object (or variable) can exist only once. That is, it cannot be instantiated. There can be no direct communication between static and instance structures.

The `C` version of this would be in the two different ways of creating structs. One where you give your struct a name at the end, vs if you instantiate it afterwards.

6 Object Class and Class Class

All objects in Java extend the Object class. I went in-depth with this in my 250 guide, but I'll summarize here.

There are a few important methods that the Object class has.

- `toString()`
- `equals(Object o)`
- `hashCode()`
- `clone()`

6.1 Object.toString()

Automatically called by many functions such as print statements. Overwrite this to change the string representation of your object. By default, it stores the name of the object, and a unique hexedecimal identifier.

6.2 Object.equals()

Since the `==` operator only checks if the addresses are the same (for reference types), using it with reference types often has strange results. It is therefore necessary to override the `Object.equals()` method to specify what it means for two objects to be equal. Are their attributes equal?(This is the default) Etc. Note that the default will not look at nested Objects, so you must specify an equals method for complicated data structures.

6.3 Object.hashCode()

Hash data structures are only as good as their hash function. So if the default for your data type is really bad, so you'd have to implement your own hash function (override the hashCode() method.)

6.4 Object.clone()

The default clone makes a shallow copy by only cloning the references. (The contents of the original can be changed by the clone!!). A deep copy would be making a clone of all references and make them point to actual independent locations. This can be done with the clone() method.

6.5 Type Inquiry

This is the idea of testing what type a variable is. This is done using the `instanceof` operator. For example: `x isinstance Shape`.

However, it does not check if it is exactly that type, it instead checks if `x` is part of the inheritance tree of Shape. We can use this to check if a cast will be successful.

We can check for exact class matching with the `Object.getClass()` method. It returns an Object containing the type information for a class. This object contains the name, and a pointer to the superclass object. (The class object of it's parent).

The Class class has some other useful methods such as `isArray()` and `getComponentType()` which 1) tests if the class is array, and 2) if it is, get the type of the components of the array.

7 Reflection

This is the practice of a program analyzing itself.

The Class class we looked at is useful for reflection.

We also have:

- Class
- Package
- Field
- Method
- Constructor
- Array

Using these, we can get all the features of a class.

```
Class super = Rectangle.class.getSuper();

Class[] interfaces = Rectangle.class.getInterfaces();

Package p = String.class.getPackage();

Field[] fields = Math.class.getDeclaredFields();
for (Field f: fields){
   if(Modifier.isStatic(f.getModifiers()))
}

Contstructor[] constructors = Rectangle.class.getDeclaredConstructors();

for(Constructor c: constructors){
   Class[] params = c.getParameterTypes(); // can be used on methods too!
Method m = PrintStream.class.getDeclaredMethod("println", String.class);
m.invoke(System.out, "Hello, world!"); //invoke is only for public methods
}
```

Note that we might get some errors with `getDeclaredMethod()` namely, `NoSuchMethodException`, `IllegalAccessException` , and `InvocationTargetException`

We can then use the `setAccessible(bool tf)` method on Fields to set the field to be able to modified. We can then use `set()` and `get()` on the fields.

In this example, we'll resize an array:

```
Object newA = Array.newInstance(a.getClass().getComponantType(),
    2*Array.getLength(a) + 1);

System.arraycopy(a, 0, newA, 0, Array.getLength(a));

a = newA;
```

8 Generic Types

A generic type is instantiated when an actual type is substituted for the type-variable.

```
public class ArrayList<E>{
   public E get(int i){...}
}
```

If a type is used, it must always be the same for the same type variable. So if you give `E` a String, all references to `E` must be with a String.

We can take this further, with extending. For example:

```
public static <E, F extends E> void append(ArrayList<E> a, ArrayList<F> c){...}
```

So this would only allow types E and F if F extends E.

Going *even further* we can use wildcards like this:

```
public static <E> void append(ArrayList<? super F> a, ArrayList<F> b, int
    count){...}
```

To say that anything which is a parent of F is valid.

It's worth noting that all this stuff with generics is really just for the compiler and is an abstraction. The hardware doesn't care, and actually once the code is compiled, everything is an Object. Due to this, all Generics are just Objects when you try to do type inspection on them at runtime.

Generics can't:

- Throw or catch generic types.
- Use primitives
- Be static

- Be instantiated
- Mess up inheritance

Part III
Tools That Aid Good Design

Part IV
Design Patterns