OOSE

Dr Graham McDonald

Joao Almeida-Domingues[*]

University of Glasgow

January 13th, 2020 – March 25th, 2020

Contents

1 Modelling 2
- 1.1 3 Principles of OOP . . . 2
- 1.2 OO Design . . . 2
- 1.3 UML . . . 3

2 OO Software Metrics 6
- 2.1 How can quality be measured? . . . 6
- 2.2 CK Metrics . . . 7

3 Software Quality 11
- 3.1 Introduction . . . 11
- 3.2 Debugging Techniques . . . 12

These lecture notes were collated by me from a mixture of sources , the two main sources being the lecture notes provided by the lecturer and the content presented in-lecture. All other referenced material (if used) can be found in the *Bibliography* and *References* sections.

The primary goal of these notes is to function as a succinct but comprehensive revision aid, hence if you came by them via a search engine , please note that they're not intended to be a reflection of the quality of the materials referenced or the content lectured.

Lastly, with regards to formatting, the pdf doc was typeset in LaTeX, using a modified version of Stefano Maggiolo's class

[*]2334590D@student.gla.ac.uk

1 Modelling

Lecture 1
January 13th, 20

Object , Encapsulation , Polymorphism , Inheritance , Abstraction , Class , Methods , Attributes , End-User

This course will focus on the basic concepts of OOP. The main idea being that one can model real world objects as abstractions in software. We take the object's attributes and functions and convert them into a single entity consisting of data and methods which operate on that data

1.1 3 Principles of OOP

1.1 definition. Encapsulation when data and operations exist within the same entity

1.2 definition. Inheritance classes can inherit attributes and methods from other classes

1.3 definition. Polymorphism ability of an entity to take many forms

As seen in JP2, the data and methods are defined within the class, and are often protected from outside access unless via getter and setters. If a class is a subset of another class then it inherits all (or most) of its behaviours and attributes and so it can be *subclassed*. This ability of the superclass to take many forms depending on which child is called is one of the crucial aspects of OOP. When methods are *overridden* by child class a method of an object reference to a superclass is invoked at compile time, and it is later dispatched to the overridden of the specific class instance at run time

1.2 OO Design

Software design relies on a symbiotic relation between the end-user and the designer. Often one is given a spec/problem statement by a client, or given a certain user story (recall HCI:1F) and from there the designer will use its modelling knowledge to interpret it in the light of OOP paradigms

1. Identify real world objects (look at the nouns in the spec)
2. Identify relationships between objects

   Generalization : Abstract common features (e.g *move*)

   Containment : Object A $\subseteq$ B (e.g *Dog* $\subset$ *Animal*)

   Multiplicity : Quantity relation (e.g Dog (1) $\leftrightarrow$ (Many) Paws)
3. Identify operations and associate them with objects (this is usually done by looking at the verbs in the spec)
4. Create an Interface

   it is essentially a contract which guaranteed that each object represented by a given class will behave in a specified manner

it must include a *return type* , *purpose/description* , *pre-conditions* , i.e what must be true prior to the method being called , *post-conditions* , what must be true when returning

5. Object Encapsulation , which describes how objects communicate via operations and how this affects the end-user

1.3 *UML*

1.4 definition. Design specify the structure of a system and its behaviour

1.5 definition. Domain Model conceptual model of the domain that includes both data and behaviour

When designing software we go from *"what"* to *"how"*, i.e. we worry about how we can represent real world systems in software. In particular, how we can use classes, fields etc. Out of this process a *domain model* emerges which represents, at the required level of formality, the system with concepts, roles and their interaction

UML is an open standard language created to represent diagrammatically an OO system. It has a *descriptive* side which provides a formal syntax as well as a more flexible *prescriptive* one, where usage shapes conventions

Given this flexibility UML has several uses, from providing a sketch of the software, which can be used to give the client an idea of the current stage of the design, and improve upon it given her feedback (e.g *use cases* modeling) or more like a blueprint, where the complete design is given to engineers for implementation.

Class Diagrams

1.6 definition. Class Diagrams represent the classes in an OO system, their fields, methods and their interactions

Pros & Cons

In this course we'll focus on class diagrams, which focus on the representation of classes. These are particularly good for providing an overview of the data and attributes, as well as the main entities in play in the design along with the complexity of their interaction. They are however not adequate for prying into the logic of the program or its control flow.

Given the wide adoption of the language in industry, the following conventions were adopted:

1. Name
   - Name : top of the box
   - Interfaces : in between << >>
   - Abstract : italics
2. Methods
   - Methods : `mName(arg:type)`

- Getter/Setter : may be ommited
- Interface : do not omit
- Inherited : omit
- Return : omit if constructor or void

3. Attributes
   - Signature : `visibility name : type`
   - Derived : `/`

not stored, but can be computed from other attributes (e.g area, given width and height attributes)

4. Visibility
   - Public/Private : `+ / -`
   - Protected/Package : `# / ~`
   - Static : underlined
5. Generalization Relationships
   - Top-Down i.e , Parent-Child
   - Class : Solid line , black head
   - Abstract : Solid line, white head
   - Interface : dashed
6. Usage Relationships
   - Multiplicity : * = 0+ , m..n = [m,n]
   - Navigability : Yes -> , No X
   - Aggregation : White diamond
   - Composition : Black Diamond
   - Dependency : Dotted

1.7 definition. Generalization Relationships represent inheritance and interfaces

1.8 definition. Association Relationships used to show that instances of classifiers could be either linked to each other or combined logically or physically into some aggregation

There are 3 major types of associational/usage relationships classified by the degrees to which the related entities are linked. So, dependent types represent a *"uses temporarily"* relation ; aggregation represent *"is part of"* and composition represents an *"is entirely made of"*.

To illustrate the differences between them take the "engine-car" relation. We say that the engine is part of the car, and the car has an engine. A stronger relation would be that of a book and its pages. Note that it is conceivable to image a car without a motor, a car would still be a car, even if it lost its motor. However, what is a book without pages? Lastly, the dependency relation are

the weakest connectors, and simply state a sort of relation where one object *might* use another. It is often an implementation detail, not an intrinsic part of the object itself (e.g. *hasRead* method in *Person* and a *Book* object)

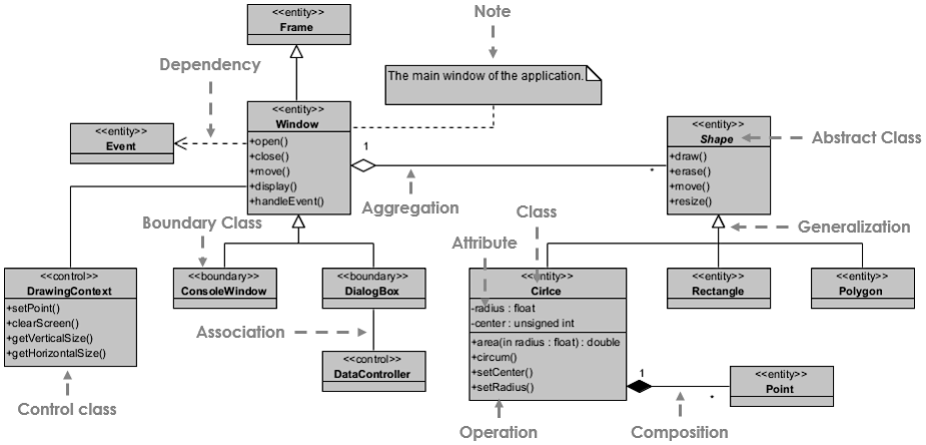


Figure 1: GUI Class Diagram [10]

2 OO Software Metrics

Control Flow Graphs , Cyclomatic Complexity , CK Metrics

2.1 definition. Metric quantitative measure of degree to which a system, component or process possesses a given attribute

2.2 definition. Indicator combination of metrics that provide insight into the product/process

2.3 remark. metrics differs from measures in the sense that metrics are functions which output measures [8]

Motivation

- Budgeting
- To create a baseline, in order to compare the impact of new tools
- Establish productivity trends and estimate staffing needs
- Improve software quality

2.1 *How can quality be measured?*

In 1992 , Basili [2] suggested a *top-down* goal oriented framework, to define what could me used as a measurement

1. Develop a set of goals
2. Develop a set of questions which characterize those goals
3. Specify the metrics required to answer 2
4. Develop appropriate mechanisms for data collection and analysis
5. Collect, validate and analyse the data
6. Analyse in a *post-mortem* fashion
7. Provide feedback

Control Flow Graphs

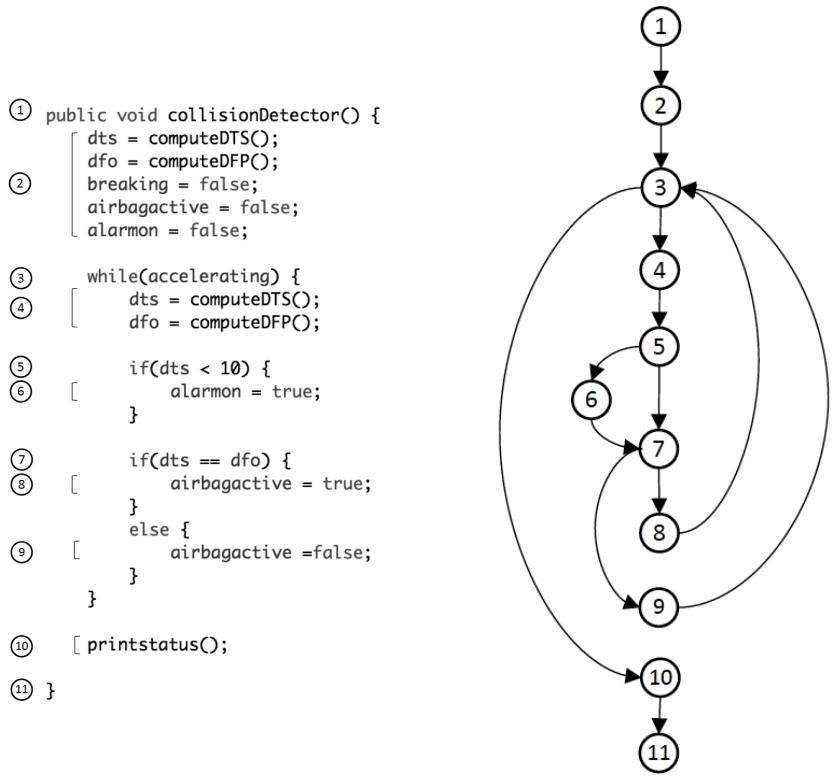
2.4 definition. CFG a graph representing all possible paths that might be transverses during the execution of a program

We represent blocks of code as nodes in a graph, and draw edges between two nodes *iff* the execution of one of a node could transfer control to the one being connected to it

In order to draw a CFG, we start by breaking the code into its major blocks by identifying (1) Methods ; (2) Method Blocks ; (3) Decision Points . We then

translate this into nodes and connect them according to all possible execution states

2.5 example.



2.6 definition. Cyclomatic Number $V(G)$ of a graph G on n vertices and e edges with p connected component. $V(G) = e - n + 2p$

Given the CFG, McCabe [7] introduced a graphic-theoretic complexity measure and showed how it could be used to manage and control program complexity. The goal was *"to find a mathematical technique that will provide a quantitative basis for modularization and allow us to identify software modules that will be difficult to test or maintain."* As a rule of thumb, when restructuring code one aims to start with those with higher $V(G)$, in order to lower their complexity

Pros & Cons

This is a fairly easy to compute metric, and empirical studies have shown good correlation between cyclomatic complexity and understandability. However, it is the hard to grasp the flow of data, and its use might not be appropriate for OO programs.

2.2 *CK Metrics*

More recently, Chidamber and Kemerer [3], introduced 6 other metrics, which

are still in use today, despite the existence of 300+ new ones

1. **WMC** : Weighted Methods per Class
2. **DIT** : Depth of Inheritance Tree
3. **NOC** : Number of Children
4. **CBO** : Coupling Between Objects
5. **RFC** : Response For a Class
6. **LCOM** : Lack of Cohesion of Methods

A summary of the 6 metrics and how they affect complexity , re-usability and modularity are presented below

METRIC	GOAL	LEVEL	COMPLEXITY (To develop, to test and to maintain)	RE-USABILITY	ENCAPSULATION, MODULARITY
WMC	Low	▼	▼	▲	
DIT	Trade-off	▼	▼	▼	
		▲	▲	▲	
NOC	Trade-off	▼	▼	▼	
		▲	▲	▲	
CBO	Low	▼	▼		▲
RFC	Low	▼	▼		
LCOM	Low	▼	▼		▲

Figure 2: CK Metrics : Summary

Note that there exists a direct relation between the metric level and complexity level, but sometimes one trades simplicity in favour of reusability

WMC

$$\sum_{i=1}^{n} c_i$$

WMC is the sum of the complexities of the methods in a given class, where c_i represents the McCabe complexity of each method. It acts as a predictor of how much time and effort is required to develop and maintain that class

DIT

We define DIT as the length of any given node to the root of the tree. Adding dependent methods, will increase complexity but improve reusability. The goal is to achieve the appropriate trade-off for a given project

NOC

Similar to DIT, but we look at the number of *direct* children only

CBO

For a given class C , the CBO is the number of other classes to which C is coupled to. Where we define classes to be ”coupled” as ”operating upon” or ”being operated on”

RFC

The number of methods of a class as well as any other methods being called by methods within it

LCOM

Counts the sets of methods in a class that are not related through the sharing of some of the class's fields. It measures the relative *”tightness”* of a class. The aim is to have as cohesive a class as possible, otherwise we should wonder if two methods who share little to no fields should belong to that same class. In practice, the lack of cohesion of a class is calculated by subtracting the number of method pairs that don't share access to the same fields by those who do

2.7 example.

```
public class NumberManipulator
{
    private int _number;

    public int NumberValue => _number;

    public void AddOne() => _number++;

    public void SubtractOne() => _number--;
}

public class NonCohesiveNumberManipulator
{
    private int _firstNumber;
    private int _secondNumber;
    private int _thirdNumber;

    public void IncrementFirst() => _firstNumber++;
```

```
    public void IncrementSecond() => _secondNumber++;
    public void IncrementThird() => _thirdNumber++;
}
```

3 Software Quality

Static Analysis , Bug Patterns , Soundness , Precision , Debugger , Bug , Debugging Techniques , Bytecode , Source Code , Bug Triage , Software Reliability

3.1 *Introduction*

3.1 definition. Software Reliability denotes a product's trustworthiness or dependability

3.2 remark. software reliability varies *directly* with the number of bugs

3.3 definition. Bug a failure in a computer system that produces an incorrect or unexpected result

We can define a bug in terms of which outcome it produces given its spec. In particular, we classify something as a bug, if one of the following occurs

S is spec , O is outcome, and X is a behaviour

1. $X \in S \land X \notin O$
2. $\neg X \in S \land X \in O$
3. $X \notin S \land X \in O$
4. broken spec and broken outcome ; i.e. doesn't do X and it isn't mentioned *but it should*
5. clunky software, broken from an untrained user's perspective

Bug Causes

The major cause of bugs occur at the specification level, followed by the design stage and actual code

At the spec level is often due to a breakdown in communication, or a lack of formality in its implementation (e.g. not written, incomplete, sketch only). The design stage is often overlooked or rushed, leading to inappropriate modelling, or a lack of modeling tools. Coding errors can occur due to a complex nature of the software, or simpler things like poor documentation or limited time for project completion leading to rushed work

3.4 remark. some coding codes examples are wrong messages, overflow errors, override errors, misuse local variable, syntax ...

3.5 remark. the cost of fixing a bug grows linearly with the life of a project, from spec design to release

3.2 *Debugging Techniques*

3.6 definition. Bug Triaging is the assignment of a bug to the most appropriate/capable developer who will fix it

The triager needs to be aware of both the project, the bug reports and the team involved in the project. A general workflow for bug is as follows:

- a bug is reported
- bug is assigned to manager for triage
- if fixed END , else assign to developer to fix
- re-test new code ; if fixed END else goto 2

3.7 definition. Static Analysis a method of a computer program debugging that is done by examining the code without executing the program. It provides an understanding of the code structure

3.8 definition. Threat Modelling inspect at design stage, and draw out possible error paths

3.9 definition. Manual Code Reviews read source code, and reason based on your knowledge whether bugs can occur

3.10 remark. Note that this process can be quite subjective, depending on the inspector

3.11 definition. Automated Tools tools which parse program code, and automatically detect common bugs

3.12 remark. AT is particularly efficient for finding common bugs/bug patterns

3.13 definition. Bug Pattern recurring correlations between signalled errors and underlying bugs

1. **Predictable random number generator**

   given the *pseudo* nature of random generators, it is easy for a malicious user to get hold of the next random number in a series. In Java, there's a better way to generate random numbers by using the `SecureRandom` library instead of the `Random`

2. **Object Deserialization**

   object deserialization of untrusted data can lead to remote code execution (e.g open attachments in untrusted email sources).

   It is best to avoid untrusted sources, but when not possible, then input validation is robustly applied for sanitation. Thus mitigating any malicious input strings being converted to executable binary

3. **Trust Boundary Violation**

   when the trust boundary is blurred, and data is passed over the boundary without proper validation

4. **Null Pointer Exception**

   when an object which does not exist is accessed

5. **Infinite Recursion**a call to a recursive function that never reaches the base case

3.14 definition. Deserialization taking structured data (e.g JSON) and rebuild it into an object

3.15 definition. Trust Boundary an imaginary line drawn through logical pieces of a program where on one side we assume that the data is trustworthy, and on the other untrustworthy

Manually examine source code can be labour intensive and subjective depending on the inspector

3.16 definition. Debugger special program used to analyse other programs in order to find bugs

The debugger will analyse program code (e.g source, bytecode , etc) and will analyse it in terms of the correctness of its statements, control flow, method class , etc. without actually executing the code

The Limitis of Static Analysis

We can't really tell whether a program P has some property Q , instead we *approximate* Q in analysis P

3.17 remark. Recall *Church-Turing's* introduction of the *Halting Problem* [9] [4]

3.18 definition. Soundness if sound then an alert is raised; unsound means that the detection system can generate false negatives

3.19 definition. Precision measures accuracy of bug detection. If precise then every bug alert is an actual bug ; imprecise implies possible false positives

Most analysis tools involve a trade-off between soundness, precision and execution time. Though, the majority of tools are *conservative* in nature, i.e soundness is often preferred

So, depending on the project, one can approximate towards:

- **Completeness**

  where the detection tool is designed in a such a way so as to *overestimate* possible program behaviours ; with the drawback that the false positives might overshadow the real bugs

- **Soundness**

  where the detection tool *underestimates* the possible program behaviours ; with the drawback that the analysis is incomplete

3.20 remark. in practice a *balanced approximation*, which is neither sound nor complete, allows for enough flexibility so that the program behaviour is more easily estimated

References

openedge

URL: https://documentation.progress.com/output/ua/OpenEdge_latest/index.html#page/dvoop/using-polymorphism-with-classes.html.

Basili et al.: Software Modeling and Measurement: The Goal/Question/Metric Paradigm **basili92**

Basili and Victor R. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*. Sept. 1992. URL: http://hdl.handle.net/1903/7538.

Chidamber et al.: A metrics suite for object oriented design **chidamber˙kemerer94**

S.r. Chidamber and C.f. Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. DOI: 10.1109/32.295895. URL: https://www.pitt.edu/~ckemerer/CK%20research%20papers/MetricForOOD_ChidamberKemerer94.pdf.

Church: An Unsolvable Problem of Elementary Number Theory **church1936**

Alonzo Church. "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58.2 (1936), p. 345. DOI: 10.2307/2371045.

Introduction **umlUtsa**

Introduction. URL: http://www.cs.utsa.edu/~cs3443/uml/uml.html.

Java - Polymorphism **tutorialspoint**

Java - Polymorphism. URL: https://www.tutorialspoint.com/java/java_polymorphism.htm.

Mccabe: A Complexity Measure **mccabe76**

T.j. Mccabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/tse.1976.233837. URL: http://literateprogramming.com/mccabe.pdf.

Software metric **metricsWiki**

Software metric. Dec. 2019. URL: https://en.wikipedia.org/wiki/Software_metric.

Turing: On Computable Numbers, with an Application to the Entscheidungsproblem **turing1937**

A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230.

UML Class Diagram Tutorial **umlVisual**

UML Class Diagram Tutorial. URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/.