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

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)

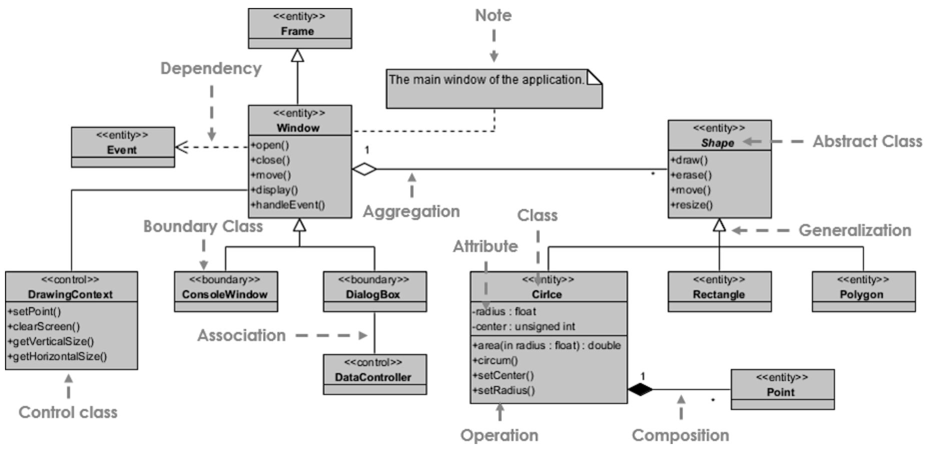

Figure 1: GUI Class Diagram *UML Class Diagram Tutorial*

References

openedge

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

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.

UML Class Diagram Tutorial **umlVisual**

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