**Learning Outcomes**

|  |
| --- |
|  |

|  |
| --- |
| * **MLO1** - Specify constraints and operations for a class diagram using OCL * **MLO1** - Model state-related behaviour using state machines |

**Summary**

|  |
| --- |
| Accurate requirement models are needed before we move onto design.  OCL formally specifies the constraints and operations that cannot be expressed clearly in a UML model. It is considered side-effect free as it cannot change anything in the model.  For heavy messaging classes, UML state diagrams can outline their behaviour. This could be a behaviour, lifecycle, or hybrid approach. |

**Lesson 1: Operation Specification**

|  |  |  |
| --- | --- | --- |
| **Operation Specification** |  | Before proceeding to design, a stabilised class diagram needs the operations to be specified by:   * Completing the signature of the operations * Identifying what the operation intends to do using operation specifications   **Operation specifications** describe the detailed behaviour of an operation. They:   * Allow a user to confirm the correctness of the proposed behaviour * Guide a programmer in the appropriate implementation of the operation   Operations are defined primarily in terms of the services they deliver and inputs they receive. This is known as **specification by contract** and is like real world contractual agreements.   * Focus is on what the operation should achieve: black box, ie irrelevant details hidden * This is different to focusing on how the operation should work: white box * Typically, they are written as pre and post condition pairs   Logic of an operations can be described using algorithmic or non-algorithmic approaches.   * **Non-algorithmic** approaches concentrate on describing the logic as a black box, hiding the details of the implementation * **Algorithmic** approaches describe the internal logic of a process, specifying the sequence of steps which are performed   In **object-oriented programming**, non-algorithmic approaches are preferred because:   * Only programmers responsible for a class need to focus on its internal implementation * Even distribution of effort across classes should result in small, simple classes   Non-algorithmic approaches may use methods such as decision tables and pre and post conditions.  A **decision table** is a matrix that shows the conditions under which actions are taken. They work best when different combinations of inputs can produce multiple alternate outcomes.    **Pre and post conditions** answer the following questions:   * What conditions must be satisfied for an operation to execute? * What is the result of the operation completing?   Algorithmic approaches use activity diagrams, pseudocode, or structured English to describe internal logic. |

**Online**:Section 4.3, Software Engineering, University of York

**Print**:Chapter 10.1-10.4, Object-Oriented Systems Analysis and Design, Bennett et al

**Lesson 2: Object Constraint Language**

|  |  |  |
| --- | --- | --- |
| **OCL Keywords**     * **Operations** * **Collections** |  | Natural language is expressive but suffers from ambiguities due to interpretation.  Object Constraint Language (OCL) provides a language where all constructs have precise and defined meaning. This avoids issues with natural language expression and constraints difficult to diagram.  **Mutation** operations are not used in OCL as it is side effect free and cannot change anything in the model.  The specification for OCL 2.4 can be found [here](https://www.omg.org/spec/OCL/2.4/PDF).  **Context** (context) defines the domain within which the expression is valid.   * context Customer   **Invariants** (inv) are properties which are always true. The context is usually a class.   * context Customer   inv: age >=18  **Pre and post conditions** (pre, post) . The context is always an operation.   * context Account::withdraw(amount:Integer)   pre: amount > 0 and  balance-amount >= odLimit and  accessor = holder  post: balance = balance@pre-amount  **Self** (self) is used to refer the context object if there is the possibility of confusion across classes.   * context Customer   inv: self.age >= 18  **Result** (result) is like the return statement in Java: it indicates the value returned from an operation.   * context: Account::balanceEnquiry():Integer   pre: accessor = holder  post: result = balance  OCL supports four basic data types each with valid operations:   * Real: + - \* / >= <= > < * Integer: + - \* / >= <= > < * String: size() this returns the number of characters in the string * Boolean: and or not implies if then else endif   OCL also offers **collection** data types (set, orderedset, bag, sequence) and valid operations:   * size(): integer * isEmpty(), notEmpty * sum() * count(object): integer * includes(object): Boolean * excludes(object): Boolean * including(object): collection * excluding(objects): collection * select(condition): collection * exists(condition), forAll(condition): Boolean   Operations on collections are indicated using **arrow notation** (->).   * Context Customer   Inv: self.heldAccount->size() <= 5  The type of an object can be checked through the **oclIsTypeOf** operation.   * context Customer   inv: self.oclIsTypeOf(Customer) |

**Online**:Section 4.5, Software Engineering, University of York

**Print**:Chapter 10.5-10.6, Object-Oriented Systems Analysis and Design, Bennett et al

**Lesson 3: State Machines**

|  |  |  |
| --- | --- | --- |
| **State Machine Notation**     * **Transitions** * **Internal States** |  | All objects have a state. For example, a library book may have both a checked out and available state.  State related behaviour results in the same object behaving differently depending on its state.  A state machine therefore is a depiction of all the possible lifecycles that single class might follow:     * An object must immediately move from the **initial pseudostate** to a named state * An object cannot leave the **final state** once it has been entered * All other **states** are shown as rounded rectangular boxes with meaningful labels   **Transitions** between states are triggered by one of four types:     * Boolean **change triggers** * A **call trigger** (synchronus) or **signal trigger** (asynchronous) * Time needed to pass for a **relative time trigger** should be in brackets * **Guard conditions** can be added to annotations and are indicated by square brackets * **Actions** caused by the trigger such as sending signals are indicated by /   **Internal state activities** are not associated with transitions, but it can be useful to model internal activities associated with the state. The state rectangle is divided into two or three components:     * **Entry**, **Exit** and **Do** are reserved keywords * Note that Entry and Exit cannot have guard conditions as they are always invoked * Internal transitions do not invoke exit and entry actions |
| **Advanced Notations** |  | In a **composite state** the rectangle becomes its own state diagram, to allow modelling of substates.    **Concurrent states** contain multiple independent substates that may run and complete at different times:    **Synchronised concurrent states** use forks and joins to create parallel nested submachines:     * A **fork** will create a pseudostate, splitting progression into a concurrent substate * This state will not be exited until both complete and merge at the **join** pseudostate |
| **Preparing State Machines** |  | In a **behavioural approach**, events are collected from all interaction diagrams. A first draft will show actions designed to respond to the events, which will correspond to the operations needed to be coded.  A **lifecycle approach** is less formal and does not consider interaction diagrams but instead works directly from use cases. It considers the lifecycles of objects of each class.  A combination of both approaches is usually recommended, as each informs the other. |

**Online**:Section 4.7, Software Engineering, University of York

**Print**:Chapter 11.1-11.4.2, 11.5, Object-Oriented Systems Analysis and Design, Bennett et al