**Homework 6**

**COM S 362**

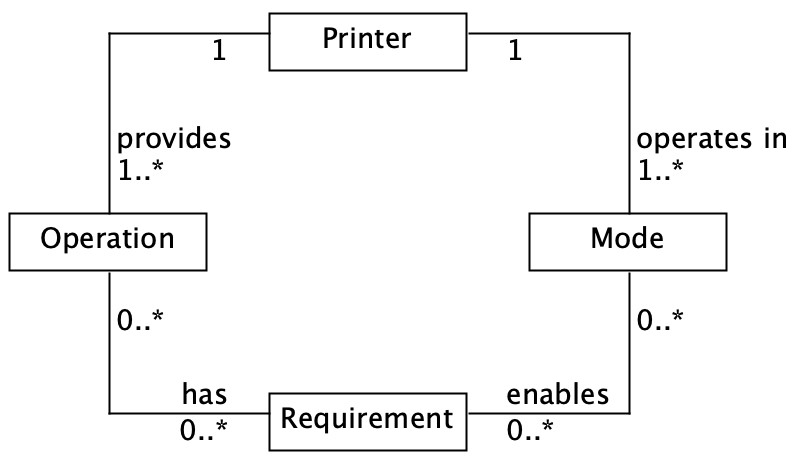
**Fall 2021**

The fine software engineers at Epsom Salty Inc., are developing a driver for their new line of Epsom LP0 printers.

The firmware team has given them the following description of the printer.

At any time, the printer is in one of three modes: initializing, operational or diagnostic. Commands to perform different operations can be sent to the printer. The operations are restricted based on what mode the printer is in. For example, printing a document requires the ability to upload the document and start the print job.

The software analysis team has put together a rough domain model.



The software engineers jump into action and create the following code. It is clearly incomplete but is meant as an experiment to test their ideas about how to structure the code.

|  |
| --- |
| **public** **class** PrintDriver { **private** Mode mode;    **public** PrintDriver() { mode = **new** Mode(); mode.setSate(State.**INIT**);  }    /\*\*   * Uploads a document to the printer and sends the command to print.   \*   * **@param** doc the document * **@return** true if print command successful, false otherwise   \*/  **public** **boolean** printDocument(Document doc) {  // The team realizes operation parameters should eventually // be loaded from a config file, but hard-coded for now.  Operation printDoc = **new** Operation("print doc", **true**, **true**, **false**);    // block the operation if it is not allowed **if** (!mode.isOperationAllowed(printDoc)) {  **return** **false**;  } |

|  |  |
| --- | --- |
| // execute operation...  **return** **true**;  }  }  **public** **class** Operation { **private** **boolean** upload; **private** **boolean** printing; **private** **boolean** cleaning;    **public** Operation(**boolean** upload, **boolean** printing, **boolean** cleaning) { **this**.upload = upload; **this**.printing = printing; **this**.cleaning = cleaning;  }    **public** **boolean** requiresUpload() { **return** upload;  }    **public** **boolean** requiresPrinting() {  **return** printing;  }    **public** **boolean** requiresNozzelCleaning() {  **return** cleaning;  }  }  **public** **class** Mode {  **public** **enum** State {**INIT**, **OPERATIONAL**, **DIAGNOSTIC**};  State state;    /\*\*   * Determines if an operation is allowed.   \*   * **@param** op the operation * **@return** true if allowed, otherwise false   \*/  **public** **boolean** isOperationAllowed(Operation op) { **if** (op.requiresUpload() && state == State.**INIT**) { **return** **false**;  }    **if** (op.requiresPrinting() && state != State.**OPERATIONAL**) {  **return** **false**;  }    **if** (op.requiresNozzelCleaning() && state != State.**DIAGNOSTIC**) { **return** **false**;  }    **return** **true**;  }    **public** **void** setState(State state) { **this**.state = state;  } | |
| } | **public** State getState() {  **return** state;  } |

1. **(20 points)** Describe the application with respect to the Open-Closed Principle. Be specific about what aspects of the application follow or violate the principle.
   1. **The print driver class violates the principal because more code would need to be added if the capabilities of the printer expanded**
   2. **It doesn’t follow the open closed principal since the code doesn’t follow/use dependency inversion.**
   3. It does follow the idea of keep “classes” or objects to only preform their specific task

1. **(20 points)** The team has decided that the behavior of different operations could be better described using multiple classes that all implement a common interface. Here is the interface that operations will implement.

|  |
| --- |
| **public** **interface** IOperation {  **public** **void** setDocuement(Document doc); **public** **void** setNumCopies(**int** copies); **public** **double** getInkLevel();  **public** **void** setPrinterSharingOptions(Sharing sharing); **public** **void** setDiagnosticLevel(DiagnosticLevel level); **public** **boolean** requiresUpload(); **public** **boolean** requiresPrinting(); **public** **boolean** requiresNozzelCleaning(); } |

Is the new design consistent with the Interface Segregation Principle? Describe the positive/negative consequences of the code being in/out of compliance with the Segregation Principle. Be specific about any impacts on future possible changes to the code.

* No, this new segment of code isn’t consistent with the Interface Segregation Principle. The IOperation does have a positive impact on the code since it utilizes dependency inversion which flips what it depends on to the interface and not the concrete class. This allows for more modularity and easier changes. This has a negative impact on the program because if something needed to be changed in the interface that only affected one of the operations classes, regardless all of the classes would at least have to recompile since they all depend on it. This would also cause future issues when changing code because it could be lost on what components of the interface align with each specific “sub-class” of operations.

1. **(20 points)** Consider the following addition to the code.

|  |
| --- |
| **public** **class** MockMode **extends** Mode {  State state;    /\*\*  \* Attempts to update the state so that the operation will be allowed.  \*/  @Override  **public** **boolean** isOperationAllowed(Operation op) {  **if** (op.requiresUpload() || op.requiresPrinting() &&  !op.requiresNozzelCleaning()) { setState(State.**OPERATIONAL**); **return** **true**;  }    **if** (op.requiresNozzelCleaning() && !op.requiresPrinting()) { setState(State.**DIAGNOSTIC**); **return** **true**;  }    **return** **false**;  }  } |

Is MockMode a subtype of Mode according to the Liskov Substitution Principle? What is the consequence to the design of it being or not being a subtype?

* MockMode is not a subtype according to Liskov Substitution principle. This consequence of assumption about the behaviors would get violated when trying to call MockMode in use of its parent. The isOperationAllowed() function are similar but different. If MockMode were to be called under assumptions on how Mode should behave the program would fail or enter fault states because of this. (Ex using MockMode and then seeing if the operation of printing a document is allowed while the state is still in INIT)

1. **(40 points)**
2. List all the dependencies of the application. For each dependency state whether it is in the same or opposite direction of the flow of control.
   1. PrintDocument Control Dependency on Operation
   2. Mode Source Dependency on Operation

1. Draw the class diagram of an improved design. The design should apply a dependency inversion in at least one place. The class diagram must be drawn with software and be “fully dresses” (i.e. include all details).

A picture containing diagram

Description automatically generated

1. Describe any benefits of the improved design. Be specific about the impact of the design on all relevant SOLID design principles (not just dependency inversion).
   1. The improved design will utilize an abstract class for operations which will align with the Open-Closed principal so new features can be added without having to modify existing code. It will also utilizes dependency inversion to the operations class by adding an interface so the PrintDriver and operations class depend on the Interface instead. This also helps with Interface segregation since Operations is an abstract class each implementation of a different operation would implement the interface which will be adjusted to its certain requirements.