|  |  |
| --- | --- |
| Specifications: Learner applies the specifications to the application, including the use of custom classes, using multiple objects effectively, and dependency injection. | Should be evident within code supplied. |
| MVC: The application uses the MVC pattern appropriately. | Should be evident within code supplied. |
| I/O Operations: The application can perform I/O operations to a file to store and retrieve data. | Should be evident within code supplied.  Specifically, within the FileIO class. |
| List/Map: The application uses a List or Map to hold data in memory. | Should be evident within code supplied.  Specifically, the ArrayList class is used. |
| Java Syntax: The application uses proper Java syntax and constructs. | Should be evident within code supplied. |
| Dependency Injection: Learner can explain the relationship between dependency injection and loosely-coupled code. | Dependency injection is the act of providing objects with their dependencies directly, rather than requiring them to construct them. This means that the objects need not be privy to the specific implementation of the dependency they are using, which means that changes to said dependency are less prone to requiring the objects dependent on them to change to accommodate.  Especially when used with interfaces, this facilitates loosely-coupled code, as it helps to ensure no specific coupling between an object and any specific implementation of its dependencies. |
| OOP Concepts: Learner can explain object-oriented programming, including classes and objects. | Object oriented programming is the paradigm that centres around the use of objects as the core units around which programs function. Rather than methods or functions existing in some bank or snippet of code and executing in a vacuum when called, they exist exclusively (theoretically, at least) within classes which are used as the blueprint for objects. When created, it is these objects that can be used to run the methods within them. This allows for greater encapsulation, as all parameters and methods of a class are contained within objects of that class, and (when following proper practice) are not modified or accessed directly (in the case of parameters) from outside that class. This in turn allows for more modular, self-reliant code that is easier to expand and maintain. |
| Interfaces: Learner can explain what an interface is using examples from the code. | An interface is essentially a guarantee or requirement as to the functionality of a class that implements it. They contain only the declarations of methods that classes implementing them are expected to implement. This essentially places minimum standards on the functionality of the code – methods it must possess, with parameters it must accept, and return values it must provide. While this does not in any way guarantee compatibility with classes who utilise the implementors of a given interface, it ensures that the minimum standards of functionality are met, at least to the extent that exceptions are unlikely to be thrown regarding unimplemented methods being called or unexpected parameter types being provided.  When properly commented, an interface can also serve as a useful blueprint for anyone attempting to implement its functionality, which can greatly assist in the creation of implementing classes.  Within the code, examples of interfaces can be found in the DataIO and UserIO classes. These exist so that other implementations of the dao and ui elements of the program may be created for different purposes, such as I/O from an SQL database or a GUI respectively. |
| Inheritance: Learner can explain inheritance using examples from the code. | Inheritance is the act of extending the functionality of a single class into another, known as a subclass or child class, via use of the ‘extends’ keyword. In doing so, the child class inherits all non-private methods of the parent class, allowing them near identical functionality, although these methods can be overridden by alternative implementations withing the child class, if the required functionality is not identical.  This allows for many powerful capabilities within the code. It facilitates polymorphism, as child objects may functionally be considered objects of the parent class. It allows for effective reuse of code for different, related purposes, and reduces the need for repetition.  Unfortunately, few examples of inheritance exist within the code, but there is one, at least. As all classes inherently extend the Object class in java, methods of that class may be overridden without the use of any explicit extension declaration. The DVD class overrides the .toString() method of the Object class, changing its implementation from returning a string containing the memory location of the object, to returning a string containing information about the object itself. |
| Composition: Learner can describe the use of composition, using examples from the code. | Composition is the concept of objects possessing other objects. More specifically, it involves objects requiring objects of a different class to function properly. This may be seen as objects containing variables that are themselves objects of a different type, or methods that require objects as a parameter.  To use objects from the code as an example, the constructor for DVDController creates objects of type UserIOCommandLine and FileIO, which are stored as variables within the object. This relationship, of an object possessing, storing within itself, and utilising objects of another type is the essence of composition.  Composition allows objects to reliably utilise the functionality of objects of other classes without fear that their access to said objects will suddenly dematerialise, change, or otherwise become unreliable. |
| Agile: Learner can explain the use of Agile as an approach to software development. | Agile software development is a counter-culture movement opposing the traditional ‘waterfall’ method of software design derived from other industries with less abstract and malleable production goals. Rather than approaching a programming project as one might a woodworking project (as in waterfall), where design requirements are conceived of and finalised before production is commenced, and remain static until completion, with testing and bug fixing regarded as a process only to be attempted once the software is already written and complete, agile focuses on iterative design and development.  While agile does not represent one unified approach to all aspects of software design, there are some generally accepted elements which constitute agile development:  Firstly, instead of generating a strict design specification, agile discusses the scope and objectives of a project, and what might reasonably be achieved within given timeframes, as well as risks, costs, and challenges. This is ostensibly to be agreed upon by all stakeholders before the project moves forward.  Once these aspects have been considered, work begins on creating a minimum viable product, i.e. an implementation of the most simple elements required for the project to be viably conceived. This often consists of nothing more than the core architecture upon which the project is to be built, with no core functionality implemented.  Once this basic starting point has been reached, the standard practice of agile development is to organise the project in iterative cycles, usually lasting no more than a month, in which specific goals and deliverables are achieved (or at least attempted) by members of the team.  Often these are broken down into smaller increments, with the goal to achieve less expansive goals within a measurable timeframe.  Once the project reaches a state where all key functionality is in place and the standard of the project meets acceptable requirements established by the shareholders, the project transitions into its final state, in which bugfixes and small patches are iteratively worked on until the now-feature complete project. At the commencement of this phase, the project is generally considered complete, and is delivered to the customer.  Many other elements of an agile working lifestyle have been established by different groups within the grassroots movement, such as daily standing meetings to encourage brevity, use of Kanban or similar task assignment frameworks, and even concepts such as pair programming, in which two programmers share a workstation and work collaboratively on a single task. All of these things can be considered examples of agile, but none of them are required for a project to follow agile principles. This is due to the inherently nebulous nature of the movement due to its lack of standardisation built upon a grassroots origin. |
| Data Marshalling: Learner can explain data marshalling and unmarshalling. | Data marshalling is the process of converting data stored in memory into a functional format that may be passed into a file, database, or other long-term storage that will persist after the program terminates. This generally involves converting all information within a data object requiring storage into string format, so that it may be stored in a database, text file, or other format that may be read and successfully reinterpreted at a later date. Other formats may however be used based on the storage method chosen. In addition to the requirements of storing data, the marshalling process also involves the inclusion of additional data within the written file used to ensure that the data is parseable when read. This includes the use of special characters such as ‘,’, ‘::’, or line breaks as delimiters to identify where certain data entries end.  Data unmarshalling is the reverse of this process. Taking whatever data was written into storage via the marshalling process, undoing all modifications and encryption performed on the data in order to facilitate storage, and parsing the data into a format that can be interpreted, understood and utilised by the program.  The most obvious litmus test for a successful data marshalling/unmarshalling process is if the program may be run multiple times and perform correctly and reliably after multiple cycles of the marshalling and unmarshalling process have taken place. |
| |  | | --- | | Code Style: Code is written with appropriate indents, naming conventions, and comments so that other developers can read the code easily. | | Should be evident within the code, as well as within the attached JavaDoc documentation. |