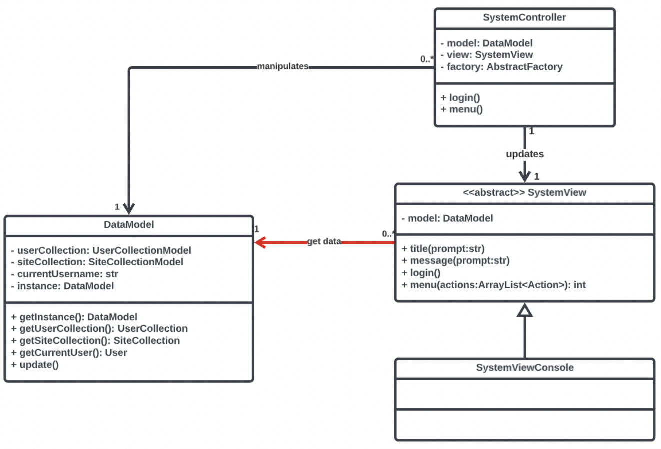
Assignment 3: Design Rationale

<u>**Architectural Patterns**</u>

MVC (NEW)

Scenario	The scenario called for using an architectural pattern. By using an architecture, we can simplify understanding of the codebase for future development, testing and extensibility. Separating business logic from the user interface is important for future extensibility. Otherwise, changing the view will impact the business logic and introduce bugs into the system. A division of responsibility makes bug isolation easier.
Pros	By separating the structure into models, controllers and views, it organises the classes such that maintainability is easier due to separation of responsibilities (Single Responsibility Principle). This software architecture is well-suited to database driven applications. This program relies heavily on the database for data access/writing purposes. MVC supports multiple views (UI) which is very important for systems like booking because multiple devices with different view logic can be implemented with the same controller. This improves extensibility of the system.
Cons	Without a facade, controllers may depend heavily on each other (tight coupling). For example, a booking modification may require changes across multiple controllers. Since we are using passive MVC for our implementation, changes in the model do not automatically trigger the view to update. As such, code complexity increases since we need the intermediary step of the controller informing the view to update. MVC structure increases the complexity of our system, especially now that we have multiple sets of MVCs - impacts on readability. Methods have strict controls. For example, view methods should not contain business logic. This division of labour improves code understanding but implementation may be harder.
Other	In our initial design for A2, we implemented the facade pattern for Menu. However, the new requirements for A3 required moving to a MVC architecture. As such we completely changed the design, and in the process removed the need for the Menu facade.

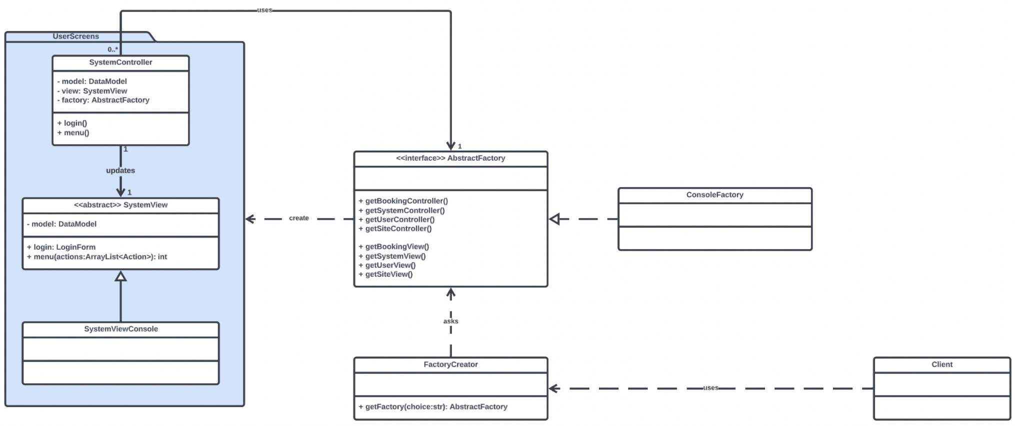


Creational Design Patterns

Abstract Factory (NEW)

We used the abstract factory pattern to enhance the MVC architecture.

Scenario	The software is designed around the MVC architecture. We believe the implementation should be extensible to new view layouts in the future (i.e. CLI vs GUI). For example, if the software is currently using CLI, the controller should be able to create new CLI views without knowledge of it being a CLI. The same applies if we switch to a GUI at runtime. This is more extensible due to loose coupling. Each action should not need to know the entire process of creating a controller (such as supplying the correct view in the correct format) since this leads to tight coupling and lower maintainability.
Pros	The abstract factory pattern implemented avoids tight coupling between controllers, actions and views. The creation code for controllers and views is extracted into a single place, which makes it easier to maintain (Single Responsibility Principle). We can introduce new view types (i.e. GUI) without breaking existing code (Open/Closed Principle).
Cons	This design increases the complexity of the code since we introduce many new abstractions and classes to implement the pattern.



Factory Method (NEW)

We used the factory method pattern for the creation of covid tests.

Scenario	The scenario demanded that the software be able to support many different types of covid tests. We also believe there should be a framework to implement new types in the future for better extensibility. For example, scientific innovations lead to new COVID-19 tests.
Pros	This pattern avoids tight coupling between client and concrete covid tests. We can introduce new test types without breaking existing code (Open/Closed Principle). Since product creation code is centralised, the code is easier to interpret (Single Responsibility Principle).
Cons	This pattern increases code complexity since we need new classes and interfaces to implement the pattern. If new types of covid tests are not introduced, this pattern is likely unneeded.
Other	We switched to the factory method pattern (we represented covid test types as enum in A2). This design improves extensibility since different covid tests can have unique implementations.

Singleton (CHANGED)

We used the singleton design pattern for DataModel.

Scenario	The scenario demanded that the data (such as sites, booking and users) be synced across the entire program after every API refresh. The data also needs to be accessed globally due to every screen (view and controller) relying on its contents.
Pros	We believe that the singleton design pattern is appropriate given its properties of only instance and its global accessibility via static methods.
Cons	Singletons violate the Single Responsibility Principle since they control their own creation and lifecycle. Singletons face issues when moving to a multi-threaded environment which may occur as the software evolves in the future. In the current software, this is not an issue.
Other	A similar idea was implemented in A2 but the idea has been consolidated into a singular singleton that encapsulates business data to simplify the dependency graph in MVC.

Structural Design Patterns

Adapter Pattern (EXTENDED)

We used the adapter pattern for API and the classes that implement its interface.

Scenario	The scenario demanded that we access data from the https://fit3077.com database. However, the raw data is in a format that is incompatible with the software classes (such as Site or Booking). We also thought that it was important for the software to be extensible to new APIs without much difficulty. For example, if a new hospital wishes to use the software but have their own personal database of covid tests.
Pros	Data transformation into a usable format is encapsulated in the ApiAdapter class (Single Responsibility Principle). For example, the Booking class should not be responsible for transforming raw JSON data from the API. We can introduce a new API without breaking existing code by implementing the interface (Open/Closed Principle).
Cons	This design increases the complexity of the code since we introduce new interfaces and classes. If we never use another API, a simple ApiService class would suffice.
Other	The initial design supported new extensions. In A3, we easily implemented new requirements that involved the database using this adapter pattern (such as notifications and booking modifications). Existing code outside of the ApiPackage did not break due to these extensions.

Behavioural Design Patterns

Memento Pattern (NEW)

Scenario	The scenario demanded that bookings be revertible to a previous state by the user. However, we believe the implementation of booking should not be exposed when snapshots are created. The controller should only interact with the public interface of booking.
Pros	Since this pattern allows booking to create its own snapshot, the booking object's state can be saved without violating its encapsulation (Single Responsibility Principle). State recovery can be executed in an understandable manner. This design is more extensible since less implementation knowledge is needed.
Cons	While not currently the case, other classes may change the state of a memento in the future. As such, the state of the memento becomes unreliable.
Other	The pattern implementation had to be adapted to support server-side saving

	of the memento. As such, its traditional design was changed.

Command Pattern (CHANGED)

We used the command pattern for Action and its implementations.

Scenario	The scenario demanded a method to navigate between different screens. We also believe that navigation should be executed in a clean and extensible fashion to improve code understandability. Controllers should not need to manage navigation since that is not their purpose. Their primary purpose should be to interact with the model and view.
Pros	Navigation between screens/controllers to execute functionality is encapsulated in each action (Single Responsibility Principle). New actions to navigate between screens can be introduced without breaking existing code (Open/Closed Principle).
Cons	If actions become more complex and rely on multiple controllers in the future, this pattern can lead to high coupling with other classes.
Other	We continued and extended the use of the command pattern from A2. However, the purpose of Action has been refactored to encapsulate navigation. This design change accommodates MVC and reduced coupling in each action.

Package Cohesion Design Principles

Stable Dependencies Principle (EXTENDED)

We adhered to the Stable Dependencies Principle when designing the Models package.

Scenario	The scenario requirements demanded many functions that depend on business data. As such it is important that business data is stored in a stable format in the Models package.
Pros	The Models package has low efferent coupling so it does not have many reasons to change. Due to this high stability, other classes can trust this package.
Cons	The Models package high afferent coupling since many classes depend on its classes. If a model needs to change due to requirements, it can be difficult to implement without hurting dependent classes (such as controllers).
Other	We adapted and extended this principle from its implementation in A2.