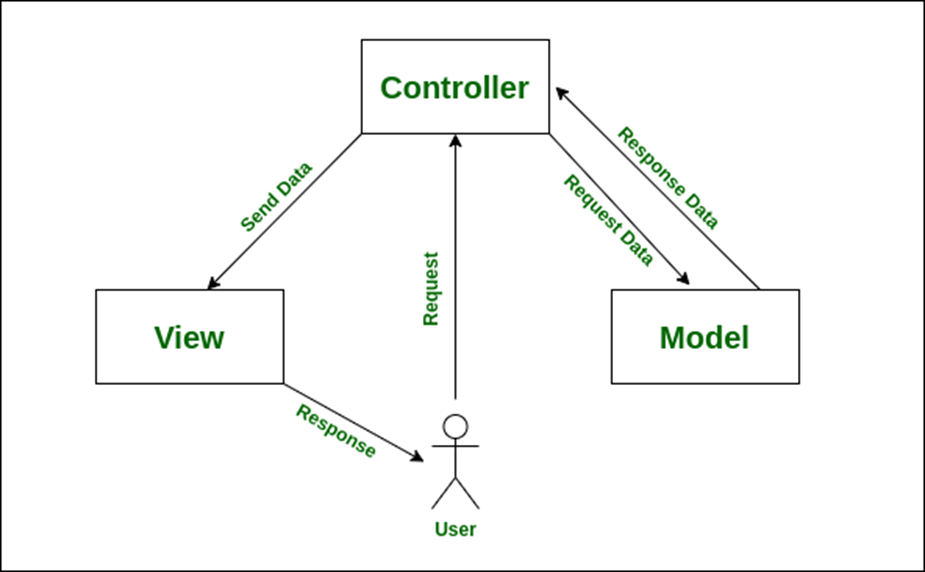
**Software Design**

Strutturare il design del software è un procedimento che richiede tempo e attenzione ai dettagli: è necessaria una interazione tra i requisiti definiti e l’architettura del sistema, inoltre è saggio effettuare più tentativi proprio perché non è semplice trovare una soluzione perfetta.

Siccome i dettagli non sono importanti a livello di design, è necessario effettuare un’operazione di *Astrazione*, in particolare *Astrazione dei Dati*: si mira a trovare una gerarchia nei dati a disposizione, per evitare di complicare la struttura di design con le varie specifiche.

Al fine di ottenere un buon sistema software, bisogna fare in modo che il progetto abbia un elevato livello di *coesione* (misura del grado in cui gli elementi del modulo sono funzionalmente correlati, ovvero che mirano all’esecuzione di un unico aspetto) e un basso livello di *accoppiamento* (misura del grado di interdipendenza tra i moduli, ovvero che gli elementi non dipendono fortemente tra di loro); ciò permette una comunicazione più veloce e semplice, le modifiche non influenzano troppo altri moduli e la comprensibilità dell’intero sistema diventa più facile. Nel nostro progetto si parla di *coesione procedurale*, perché gli elementi devono seguire un determinato ordine per essere eseguiti (per esempio, il passaggio di un paziente dalla Lista Graduatoria alla Lista Operatoria), e *accoppiamento di dati* (i dati vengono passati tra i vari moduli).

Si fa riferimento al paradigma MVC (Model-View-Controller).



***Design Pattern***

I pattern sono definiti come “soluzioni a problemi ricorrenti” e permettono di ottenere una definizione più semplice del design applicato (andando a descrivere la situazione di applicazione, fornendo soluzioni e presentando risultati e vincoli derivati).

Per implementare il framework del progetto sono stati utilizzati vari tipi di design pattern:

* **General Hierarchy Pattern** (permette di trattare un gruppo di oggetti come se fossero l’istanza di un singolo oggetto, andando a definire una gerarchia tra le classi, come, per esempio, tra i medici e gli infermieri operanti nell’intervento)
* **Player Role Pattern** (sfruttando le classi astratte, si può far “giocare” ad un attore diversi ruoli, per esempio il medico che registra i pazienti, li inserisce nella Lista Operatoria e compila il verbale dell’intervento)
* **Delegation Pattern** (per permettere ad una classe di compiere lo stesso metodo già usato da un’altra classe: la cabina di regia e i medici quando operano sulle liste)
* **Facade Pattern** (permette, attraverso un’interfaccia semplice, l’accesso a sottoinsiemi che espongono interfacce complesse e diverse tra loro, come i pazienti e il Sistema di gestione dell’Ospedale per accedere alle informazioni richieste)
* **Factory Pattern** (utile per creare un framework che possa gestire diverse classi e oggetti, indipendentemente dall’applicazione: le sottoclassi sono libere di scegliere quale oggetto istanziare)
* **Builder Pattern** (per costruire un’applicazione complessa, a partire da molti componenti oggetti: sfrutta un oggetto costruzione per tenere traccia di ciò che viene creato, ma non espone i dettagli della composizione alle singole parti)

Nel nostro progetto sono stati sfruttati maggiormente il *Factory Pattern*, per la generazione e gestione dati, e il *Facade Pattern,* per creare un’interfaccia chiara e semplice rispetto al sistema complesso.