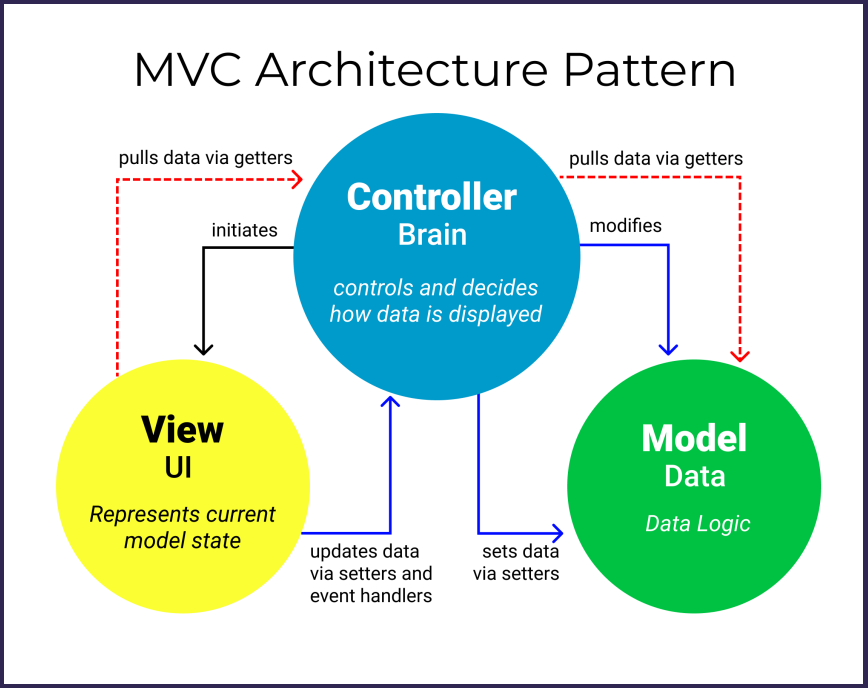
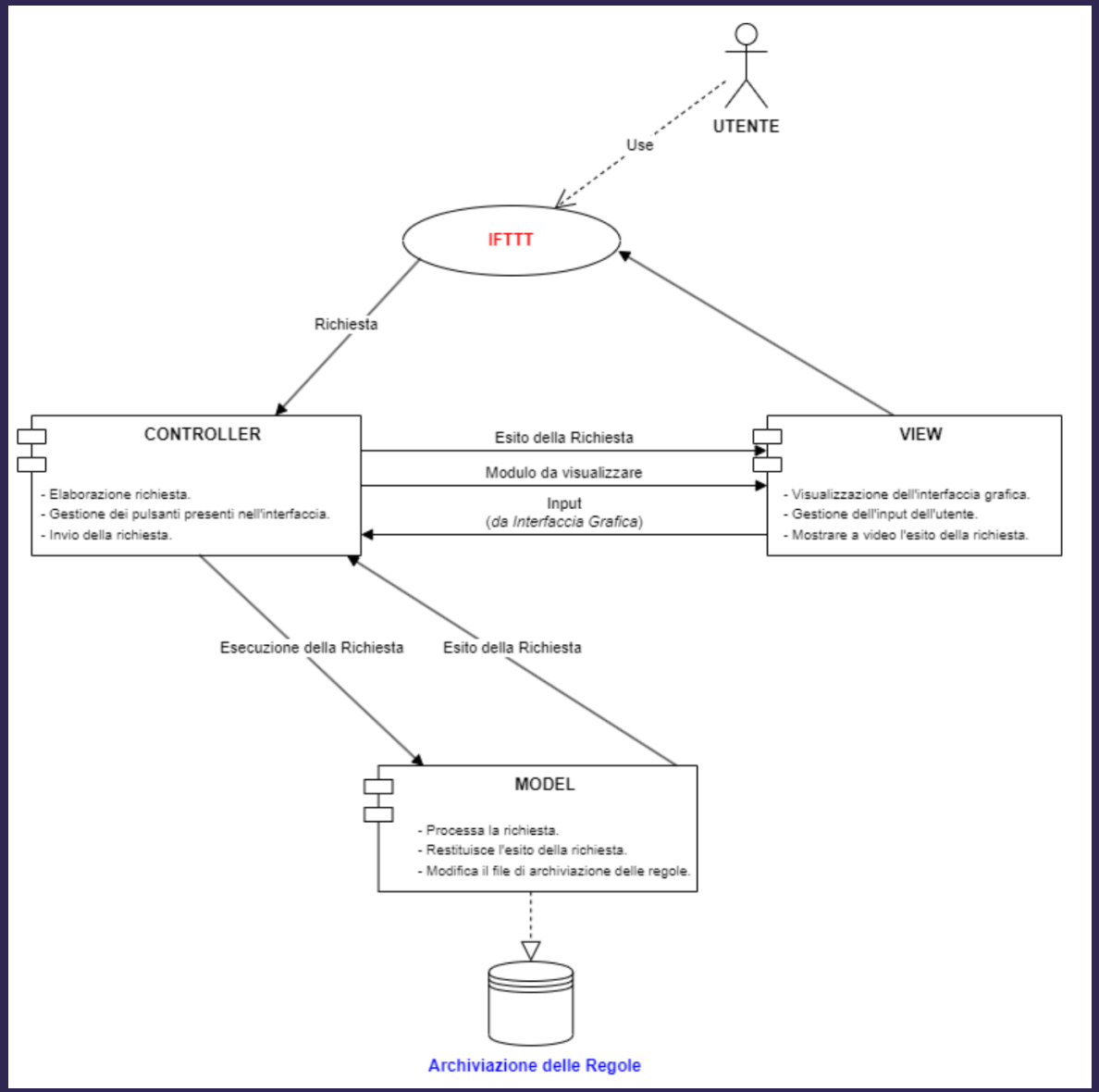
Applicazione IFTTT

Gruppo 11 (I-Z)

PATTERN ARCHITETTURALE

- Scelta quasi scontata visto l'uso di JavaFX…





PATTERN «Mock Object»

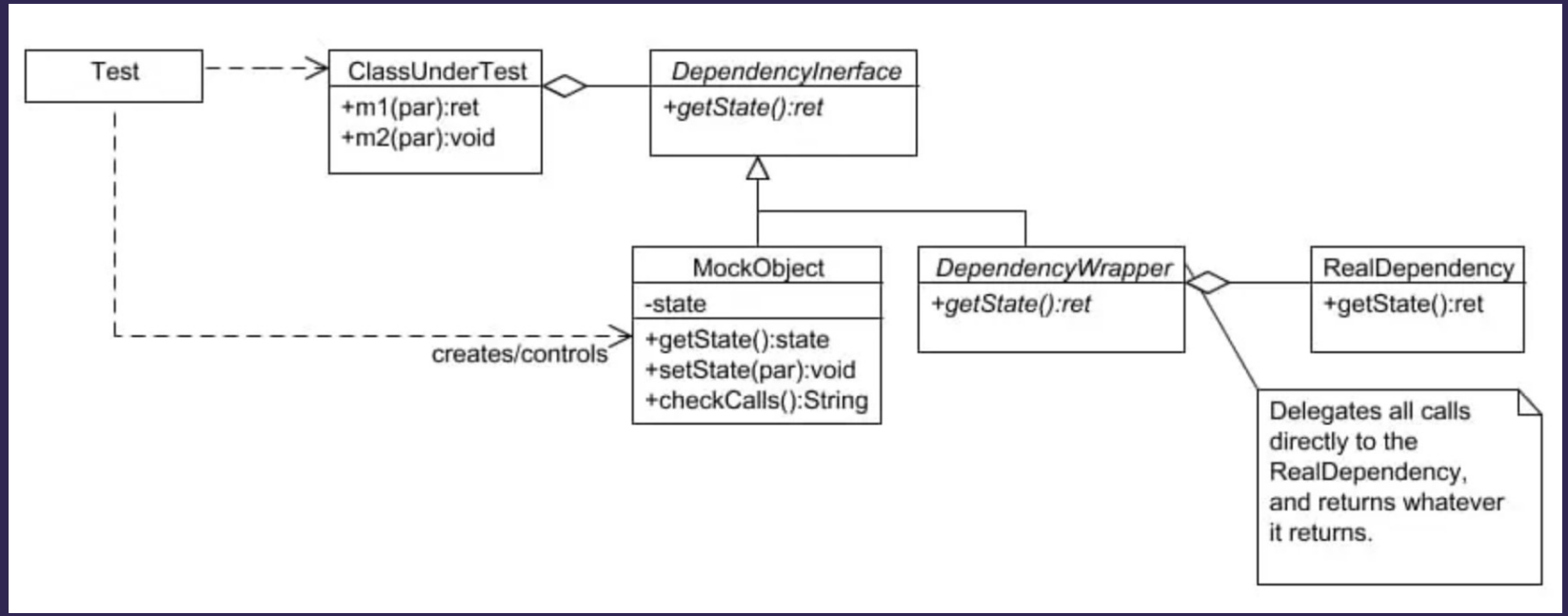
- Nella OOP, gli oggetti simulati (*Mock Object*) sono **oggetti simulati che imitano il comportamento di oggetti reali** consentendo di eseguire agevolmente i Test. Un programmatore in genere crea un oggetto finto per testare il comportamento di qualche altro oggetto, più o meno allo stesso modo in cui un progettista di automobili utilizza un manichino per crash test per simulare il comportamento dinamico di un essere umano negli impatti di un veicolo.
- Sono utili quando un oggetto reale è poco pratico o impossibile da incorporare in uno *Unit Test*.

Si usa quando l'oggetto:

1. fornisce **risultati non deterministici** (es. l'ora corrente o la temperatura corrente);
2. presenta stati difficili da creare o riprodurre (ad esempio un errore di rete);
3. non esiste ancora o può modificare il comportamento;
4. dovrebbe includere informazioni e metodi esclusivamente a scopo di test (e non per il suo compito vero e proprio).

Ad esempio, ***un programma di sveglia che fa suonare un campanello ad una certa ora*** potrebbe ottenere l'ora corrente da un servizio orario. Per verificarlo, il test deve attendere fino all'ora della sveglia per sapere se ha suonato correttamente il campanello. Se viene utilizzato un servizio di tempo simulato al posto del servizio di tempo reale, è possibile programmarlo per fornire l'orario in cui suona il campanello (o qualsiasi altro orario) indipendentemente dal tempo reale, in modo che il programma della sveglia possa essere testato isolatamente.

PATTERN «Mock Object»: **Mockito + JUnit**



Perché non abbiamo usato altri «*Design Pattern*»?

- **Pattern Strutturali ->** L'uso di **Interfacce** e **Classi Astratte**, ci ha permesso di non ricorrere a tali pattern (ad esempio, il *Facade*).

- **Pattern Creazionali ->** Sarebbe stato ragionevole utilizzare il *Singleton* per gestire le due l'istanziazione dei vari oggetti, ma la compartimentazione in **Interfacce** e **Classi Astratte** hanno risolto la problematica.

- **Pattern Comportamentali ->** L'uso esclusivo di oggetti e metodi esistenti nelle **Librerie Java**, ci ha evitato l'uso diretto dell' *Iterator* e dell' *Observer*.

- **Pattern Concorrenziali ->** L'uso della **Classe Thread** ci ha permesso di gestire agevolmente l'esecuzione di operazioni Multithread, senza dover ricorrere all'uso diretto di pattern, quali: il *Reactor* o lo *Scheduler*.

ARCHITETTURA (*Notazione UML delle Classi*)