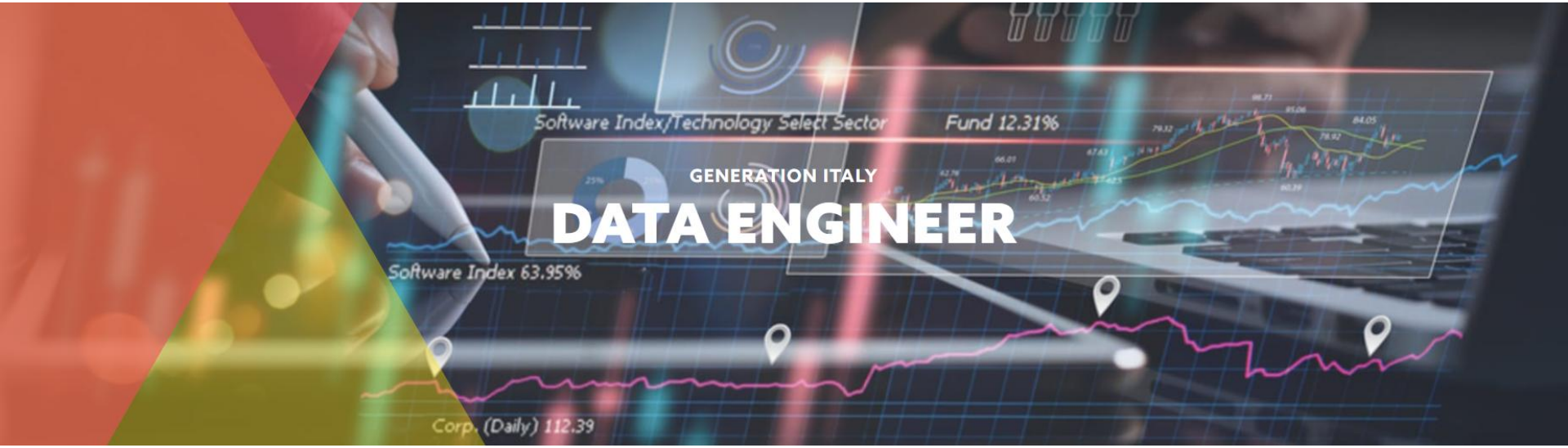
Data Engineer

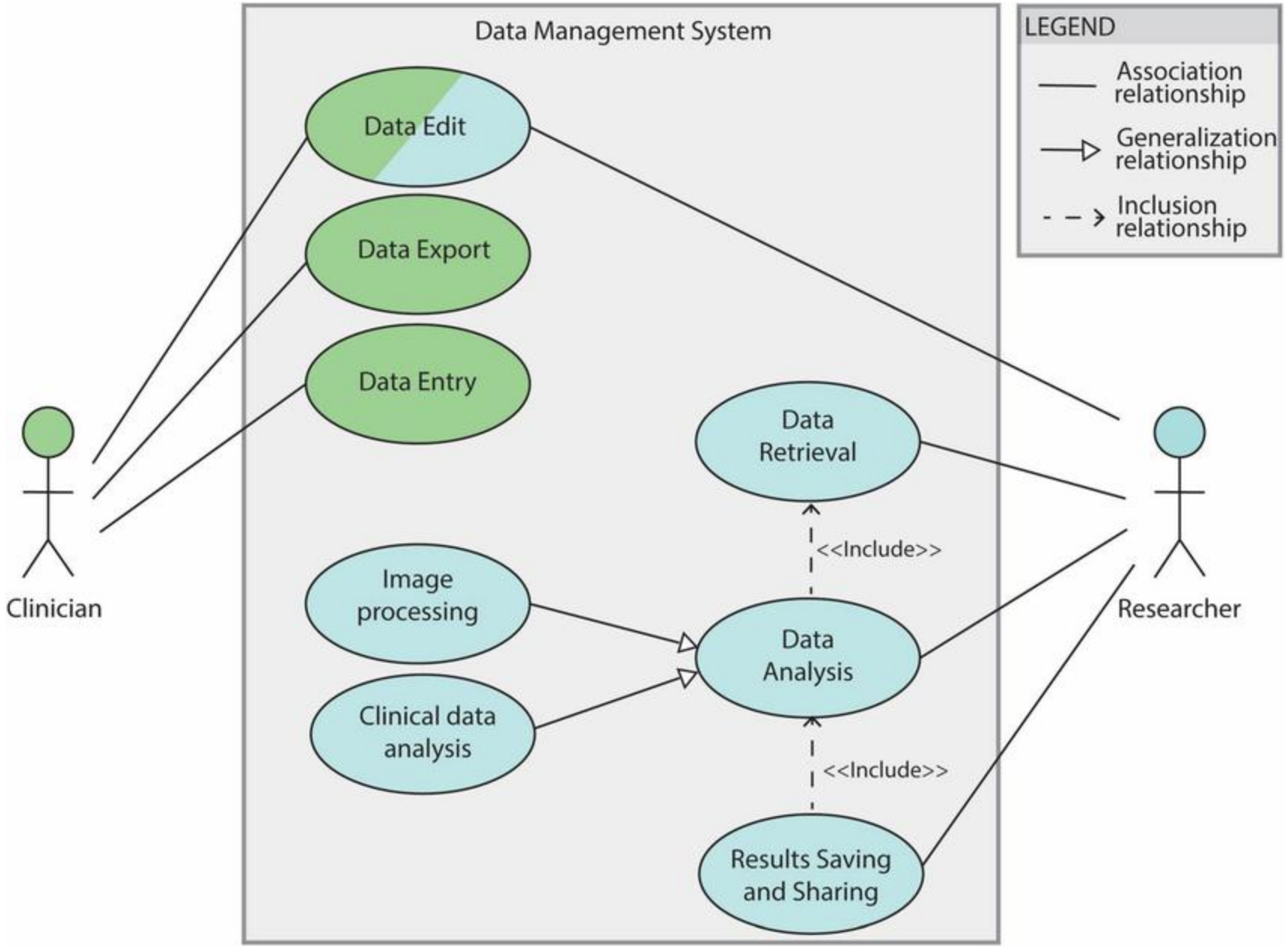


Test Driving Development

Franchini Roberto

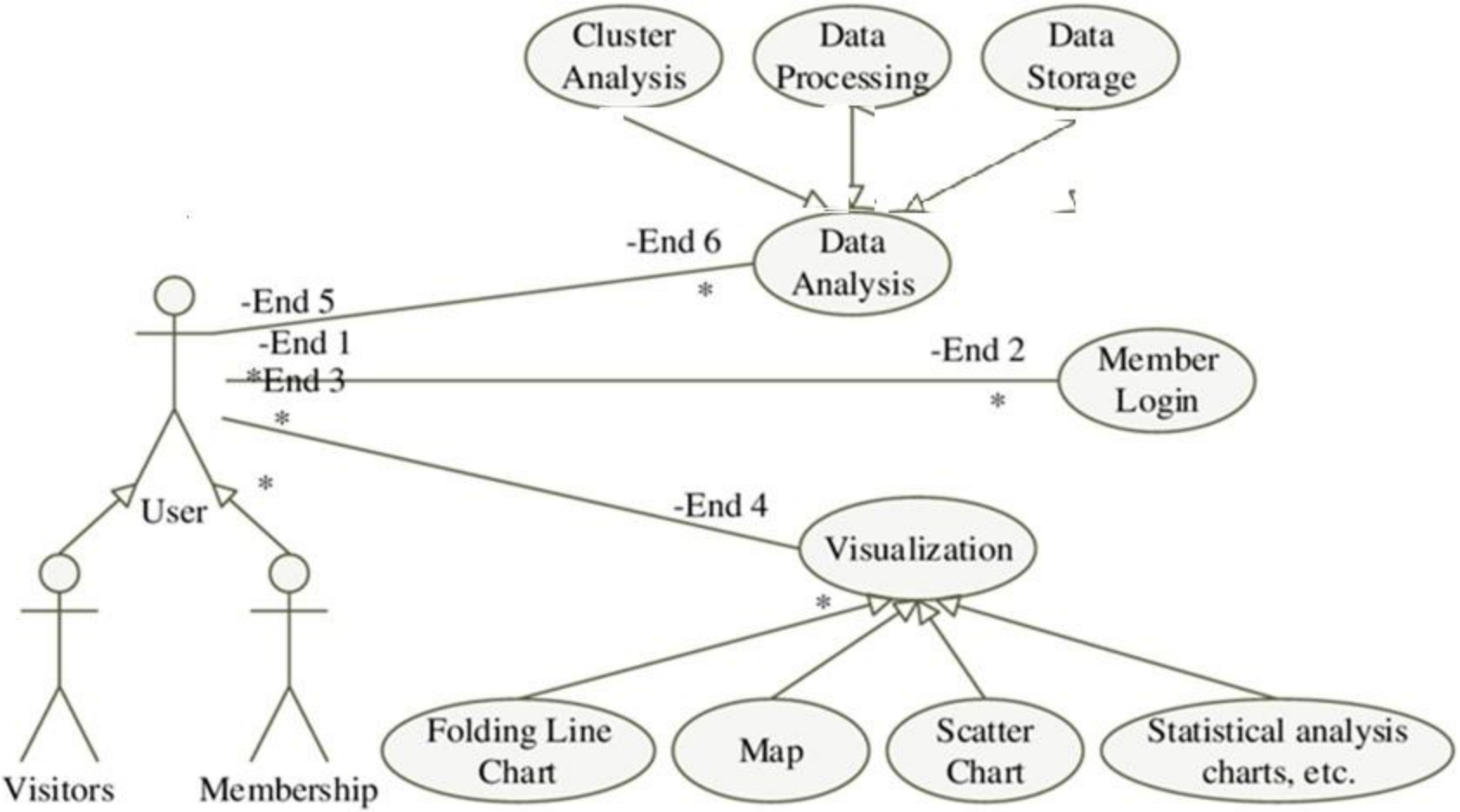
Use case 1

Il sistema vede **clinici** (medici e chirurghi) e **ricercatori** come **attori**. Le linee piene senza freccia indicano l'associazione tra l'attore e il caso d'uso. Le linee piene con la punta della freccia riempita di bianco indicano che un caso d'uso è la generalizzazione di un altro. In questo diagramma "*Elaborazione delle immagini*" e "*Analisi dei dati clinici*" sono due specifiche del caso d'uso generale "*Analisi dei dati*". Le linee tratteggiate indicano le relazioni "*Includi*" tra i casi d'uso: per poter salvare e condividere i risultati è necessario eseguire un'analisi dei dati ("*Analisi dei dati*" è quindi inclusa nel caso d'uso "*Salvataggio e condivisione dei risultati*").



Use case 2

Secondo il modello di analisi dati stabilito, il modulo estrae i dati esistenti per trovare informazioni utili per gli utenti della piattaforma e, dopo l'analisi e l'elaborazione, viene visualizzato nella piattaforma il risultato applicando la tecnologia di visualizzazione. Il case use illustra l'applicazione del modulo di analisi dei dati. L'analisi dei dati divide gli utenti in visitatori e iscritti.



Obiettivi del TDD

- Rendere lo sviluppo migliore e più rapido.
- Mantenere il codice sempre documentato.
- Ottenere codice flessibile e facilmente estendibile.
- Diminuire la presenza di bug nel codice di produzione.

Unit Test

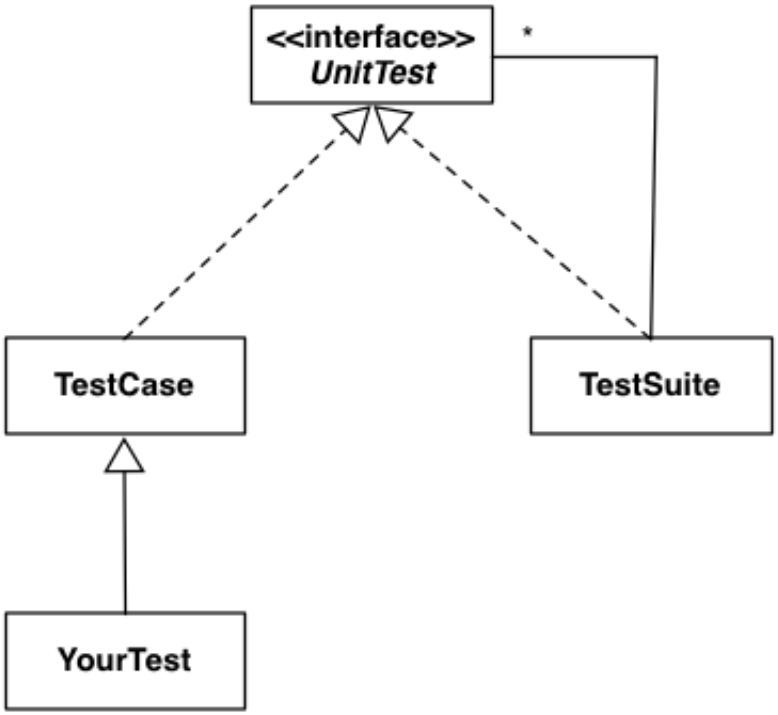
"Unit tests run fast. If they don't run fast, they aren't unit tests."

- Un test non è di unità se:
  - comunica con il database
  - comunica in rete
  - modifica il database
  - non può essere lanciato in parallelo ad altri test
  - bisogna effettuare diverse operazioni per lanciare il test

Unit Test ed il TDD

- Sono rilasciati con il codice di produzione.
- Nel TDD una funzionalità è rilasciata soltanto se vi è associata almeno uno Unit Test.
- Consentono di effettuare il refactoring di una funzionalità senza “paura”.
- Meno debugging.

L’architettura



Assert

"Gli Assert sono metodi che consentono di asserire che una certa condizione è vera o falsa."

Assert (2)

- assertTrue([message], condition) : Il test è superato se la condizione è vera
- assertFalse([message], condition) : Il test è superato se la condizione è falsa
- assertNull([message], Object) : Il test è superato se il riferimento all'oggetto è NULL.
- assertNotNull([message], Object) : il test è superato se il riferimento all'oggetto non è NULL.
- assertEquals([message], value1, value2) : il test è superato se value1 == value2.
- assertNotEquals([message], value1, value2) : il test è superato se non vale che value1 == value2
- fail() : wrapper di assertTrue(false)

Assert (3)

- (Teoria) Una regola generale è che ogni test deve contenere soltanto un assert.
- (Pratica) Spesso i metodi di test contengono più assert oppure assert singoli ma composti da più condizioni legate da operatori logici.

Testing degli errori

E' fondamentale testare la gestione degli errori. Viene utilizzato il metodo *fail*.

```
public void testXXX()
{
    try
    {
        //an operation
        fail("eccezione non avvenuta")
    }
    catch (Exception e)
    {}
    ...
}
```

- Questo test genera un'eccezione e controlla che sia gestito come ci aspettiamo.
- Nell'esempio ci aspettiamo che sia lanciata l'eccezione nel momento i cui è chiamata *anOperation*.
- Il test fallisce se l'eccezione non avviene, mentre va a buon fine se l'eccezione viene lanciata.
- *fail* è equivalente ad *assertTrue(false*) ma è più leggibile.

Testing degli errori (2)

E' fondamentale testare la gestione degli errori. Viene utilizzato il metodo *fail*.

```
public void testXXX()
{
    try
    {
        //an operation
    }
    catch (Exception e)
    {
        fail(e.getMessage());
    }
    ...
}
```

- Questo test genera un'eccezione e controlla che sia gestito come ci aspettiamo.
- In questo esempio ci aspettiamo che non sia lanciata l'eccezione nel momento in cui è chiamata *anOperation*.
- Il test fallisce se l'eccezione avviene, mentre va a buon fine se l'eccezione non viene lanciata.
- *fail* è equivalente ad *assertTrue(false*) ma è più leggibile.

Le regole d'oro del TDD

- Si scrive nuovo codice solo se un test automatico è fallito.
- Eliminare i duplicati.

Test-First Programming + Refactoring

Implicazioni tecniche

- Ogni programmatore scrive i propri test.
- Fornire risposte rapide ai piccoli cambiamenti.
- Codice con alta coesione e basso accoppiamento, altrimenti risulta difficile fare i test.

Quando fermarsi?

- Quando il sistema effettivamente funziona.
- Il sistema ha superato il 100% dei test.
- Non c’è codice duplicato.
- Il sistema dovrebbe avere meno classi e metodi possibile.

La velocità

"Unit tests run fast. If they don't run fast, they aren't unit tests."

- Nel TDD è importante la velocità con la quale i test sono eseguiti.
- Se i test non fossero veloci allora sarebbero una distrazione.
- Se i test non fossero veloci allora non sarebbero lanciati con alta frequenza: che vantaggio avrebbe il TDD?

Focused integration testing

Gli unit test da soli non bastano, il codice deve comunicare con l'esterno.

- Un *focused integration test* è focalizzato a testare:
  - la comunicazione con il database
  - la comunicazione in rete
  - la comunicazione con il filesystem
  - la comunicazione con oggetti esterni

La velocità di esecuzione dei test

- Unit tests : un centinaio al secondo.
- Focused integration tests : una decina al secondo.
- End-to-end tests : diversi secondi per ogni singol test.