**Progettazione Agile**

Il software è molto sensibile ai cambiamenti, tramite la metodologia Agile si può pianificare il lavoro non a compartimenti stagni, ma coinvolgendo il committente per adattare il progetto e dare delle priorità.

Si struttura di piccole iterazioni, aggiungendo poca funzionalità alla volta. Il testing automatico DEVE esserci in ogni software. Comunicazione importante tra sviluppatori e clienti, integrazione continua comunica la presenza di problematiche. Feedback (su tutto il sistema, sull’utilizzabilità del sistema, sull’andamento del progetto).

Product Backlog🡪Spring Backlog🡪Sprint (più di uno) 🡪Deliverable

**Debito tecnico**: Il debito tecnico software si riferisce all'insieme delle attività necessarie per mantenere e migliorare un sistema software esistente. Comprende il refactoring del codice, la risoluzione dei bug, l'aggiornamento delle dipendenze e la manutenzione delle attività di configurazione.

**Valori XP**: Comunicazione, Semplicità, Feedback, Coraggio.

Il testing abbassa i costi di produzione, i test si scrivono bottom-up, vanno accumulati e la verifica della correttezza deve essere automatica. Sono facili da scrivere. JUnit: i test seguono lo stesso directory path dei sorgenti

@Test public void test(){assertEquals(1,foo());} . @Before e @BeforeEach i metodi con queste annotation sono eseguiti prima dei test(set up), ci sono anche @After e @AfterEach.

@RunWith e @Suite: sono delle annotation che consentono di eseguire più classi di test insieme @RunWith(Suite.class) @Suite.SuiteClasses({class1.class, class2.class})class TestClass{…}

**In Java il double dispatching** (dispatch sui parametri oltre all’object dispatcher) non è possibile, si può risolvere usando un dispatcher manuale (poco OOP) oppure usare Visitor pattern, che simula il double dispatching.

AssertThrows(ExeceptionClass.class, ()->{…}) consente di verificare se viene lanciata l’eccezione specificata

@ParameterizedTest @ValueSource (ints={…}) public void test(int n){…} consente di parametrizzare il test e chiamarlo con valori diversi, specificati nel value source

@Ignore ignore il test, @Test(timeout=1000) consente di uscire dal test dopo 1000ms(1sec) con un errore (utile se rischio loop infinito)

“Un design pattern descrive soluzioni semplici e eleganti che descrivono problemi specifici in ambito OOP”

Dynamic binding = risoluzione runtime dei tipi.

Type Check: Static Check:compilatore verifica validità chiamata in fase di compilazione, Dynamic Check: il controllo viene fatto runtime**. Il pattern è una mini-architettura software**.

**Immagine che contiene testo

Descrizione generata automaticamentePattern Proxy**: Fornisce un placeholder per un altro oggetto per controllarne degli altri. Consente di gestire l’accesso ad un oggetto.

**Iterator:** un oggetto può avere più iteratori che fanno cose in modo diverso, ma usando sempre la stessa interfaccia. Per ottenere uno specifico iteratore si può usare un factory method.

**CRC (Class-Responsability-Collaboration):** le classi devono avere poche responsabilità e collaborano per delegare alcuni task.

**Template Pattern:** permette di stabilire lo scheletro di un algoritmo delegando alcune funzioni ad altre classi, avendo più versioni dello stesso algoritmo.

**Decorator Pattern:** Si occupa dell’estensione di un oggetto a runtime dando la possibilità di combinare vari oggetti.

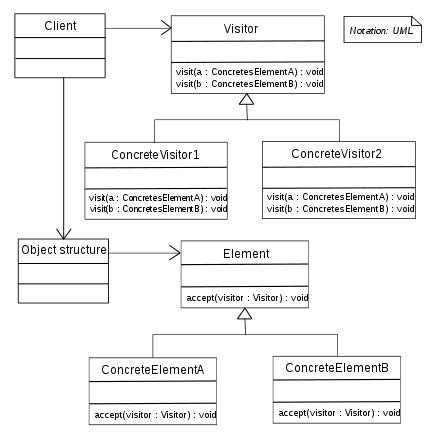
Le responsabilità sono passate in modo dinamico, consente combinazioni ricorsive ed evita di avere un alto numero di sottoclassi.

**Dependency Injection:**

è una modalità di configurazione degli oggetti, le dipendenze sono passate dall’esterno. Non vi è più una dipendenza dalla classe concreta, ma da un’interfaccia comune con le API e nelle classi concrete si implementano.

Da costruttore si può capire se vi sono altre dipendenze con altri oggetti, difficile gestire da soli, si usa framework.

**Guice** è un framework di DI. Si specifica la configurazione in un modulo di configurazione che estende AbstractModule. Nel metodo configure() si specificano le classi che dipendono dal modulo. Tramite il metodo **bind(Interfaccia.class).to(ClasseConcreta.class)**. Inserendo **@Inject** nel costruttore si inizializzano le dipendenze specificate nel modulo, il costruttore non verrà mai chiamato esplicitamente, ma lo costruisce il framework.

Si può evitare di scrivere il costruttore, mettendo @inject su tutti i campi da iniettare. Nel main si usa

**Injector injector=Guice.createInjector(new ClasseModuloConfigurazione());**

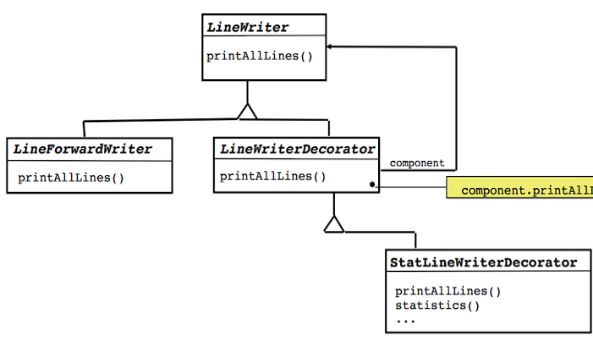
**Interfaccia obj=injector.getInstance(Interfaccia.class)**

Mettendo @Inject su un metodo esso verrà chiamato subito dopo aver creato l’oggetto.

I moduli vanno creati anche per il testing con una configurazione dedicata. Se bisogna fare un costruttore con delle operazioni, non si usa più configure(), ma si crea un factory Method con l’annotation @Provides, se il metodo è complesso si può creare una classe provider che implementa l’interfaccia Provider<T>, in questo caso non serve l’annotazione. Per legare una provider class si usa toProvider() invece di to() nella classe di configurazione del modulo. Si può forzare l’inizializzazione delle dipendenze in un oggetto con requestInjection(Oggetto)

È possibile catalogare il modulo come @Singleton, @SessionScoped, @RequestScoped (annotazione della classe di configurazione) di default si crea una nuova istanza (asEagerSingleton() singleton del oggetto nel configure).

**Mocking:** consente di creare oggetti finti che simulano un comportamento, usato nel testing. Un mock si usa: quando l’oggetto reale non ha valori prevedibili, quando c’è uno stato complicato da riprodurre, l’uso dell’oggetto rallenta l’applicazione, l’oggetto non esiste. Mockito framework di mocking. Tramite **Mockito.mock(classeOInterfaccia)** si crea un oggetto mock con **when(chiama metodo).thenReturn(somthing)** si programma il comportamento quando si chiama un determinato metodo. Non si ha una vera istanza del oggetto, ma un elemento finto (mock)

Tramite il metodo **verify(mock).metodo()** si verifica se un metodo viene chiamato, si può controllare anche il numero di chiamate con **times(), atLeastOnce(), atLeast(n), never()**. ***verify(shopMock, times(3)).doSomething()***

doThrow(new Exception()).when(mock).somthing(): verifica che quando chiamo il metodo mi lancia l l’eccezione specificata. Nei metodi i parametri possono essere passati con anyInt(), anyString(),… per inserire un valore qualsiasi. I mock si possono essere creati con @Mock e i test devono essere eseguiti con RunWith(MockitoJUnitRunner.class) Junit4, JUnit5 @ExtendWith(MockitoExtension.class). se si deve iniettare un mock in un oggetto si può annotare con @InjectMock, l’oggetto deve essere una classa concreata. @Spy usa una classe esistente con il comportamento specificato in essa, ma può essere anche istruita per tracciare il comportamento (verify di un metodo specifico), mentre il mock è una classe finta che eseguo ciò che gli istruiamo con when(). Spy lo si usa per verificare l’interazione con le dipendenze Mock per verificare casi particolari, errori. Stub forniscono risposte alle chiamate durante il test senza rispondere all’esterno, sono classi ad-hoc(MailService ad esempio non invia email, ma simula come se le avesse inviate).

Figura decorator

**AOP:** L'Aspect-Oriented Programming (AOP) è un paradigma di programmazione che consente di gestire la logica **cross-cutting**, ovvero quella logica che si ripete in diverse parti del codice, in modo separato dal flusso di esecuzione principale del programma. Ciò consente di migliorare la modularità del codice, rendendolo più leggibile e manutenibile. Gli aspetti sono implementati come classi separate che contengono la logica cross-cutting. Queste classi possono essere combinate con altre classi tramite la sovrapposizione di metodi, nota come "weaving", al fine di integrare la logica cross-cutting nel flusso di esecuzione principale. **Advice** codice aggiuntivo applicato al modello attuale (unique logging code). **Pointcut** punto di estensione interno all’applicazione, in cui il cross-cutting deve essere applicato. **Aspect** combinazione dei pointcut della descrizione degli advice body. AOP viene usato spesso per il Logging. AOP può essere implementato in 2 modi: Class-waving: inietta le implmentazioni di un singolo aspects direttamente nella classe dove deve essere eseguito. Proxy: attraverso un proxy intercetta le chiamate. I framework solitamente usano proxy poiché più semplice. **Matcher:** elemento che eseguo i match (pointcut), **Method Interceptor:** parte di codice da eseguire (advice). Creando una classe che implementa MethodInterceptor -> implementa il metodo invoke(MethodInvocation invocation) si esegue il codice ogni volta che si trova un match. Tramite invocation.proceed() si esegue il metodo che chiama con getArguments() si ottengono i parametri del metodo con getMethod() si ottengono info sul method. Nel modulo di configurazione Guice della classe si usa **bindInterceptor(Matchers.any(), Matchers.annotatedWith(annotaion.class) bindInterceptor(Che classe è da matchare, quale metodo è da mathcare, che oggetto contiene l’interceptor)**, new InterceptorObject()); questo metodo lega l’interceptor se trova l’annotation specificata in annotatedWith. Altri metodi any(), identicalTo(Object v), inPackage(Package target), inSubPackage(String target), only(Object v), subClassesOf(Class<?> supercls).

