****

**Scuola di**

**Ingegneria**

*Corso di Laurea Triennale in Ingegneria Informatica*

**Correzione Esame Ingegneria del Software**

**10 luglio 2019**

**Parte**  Observer + Proxy, Unified Process.

**Parte**  Documentazione delle attività dei laboratori di ricerca di un dipartimento.

Edoardo Re

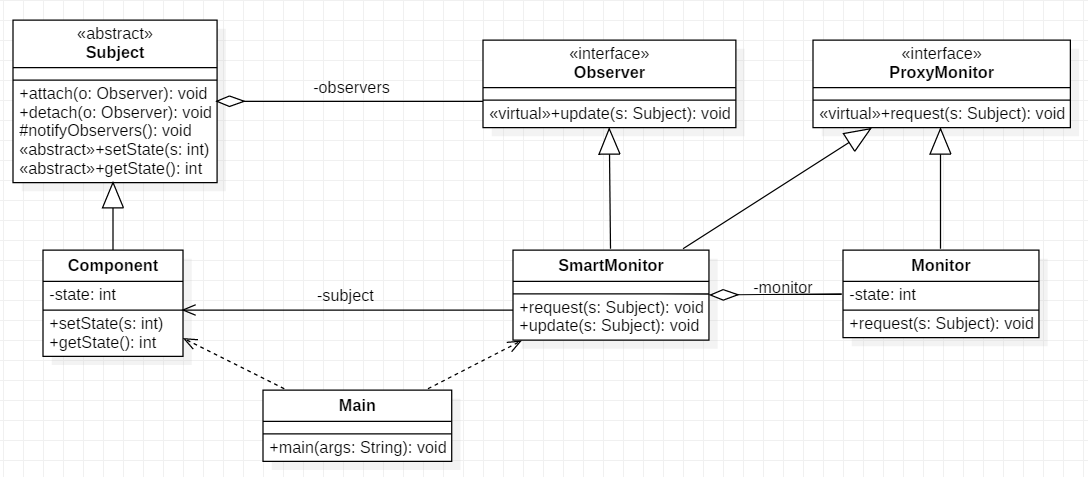
**596294**

**Domanda 1:** Si consideri il seguente scenario nel quale vengono combinati i design patterns Observer e Proxy, di cui per convenienza di terminologia viene riportata la struttura come documentata in [GOF]:

* Un oggetto di tipo Monitor deve tenere traccia delle variazioni di stato di un oggetto di tipo Component secondo un meccanismo implementato attraverso il design pattern Observer;
* la creazione del Monitor è onerosa, e viene quindi rimandata fino al momento della prima notifica, secondo un meccanismo di lazy load implementato attraverso il design pattern Proxy;
* specificamente, nello schema risultante un oggetto di tipo SmartMonitor si registra come Observer del Component, e al momento in cui riceve la prima notifica provvede a mettere in vita il Monitor e a passargli le notifiche in forwarding.

Si disegni lo schema di progetto delle classi, si dettaglino frammenti di codice che caratterizzano le implementazioni risultanti, e si definisca il codice di un metodo main che testa l’implementazione in uno scenario caratterizzante.

**Si descriva la struttura con cui vengono composti i due patterns, il Monitor e un Client che opera nel ruolo di test driver (il main) attraverso l’uso di un class diagram:**

****

Lo stato di Component osservato è implementato per semplicità come un intero, Component agisce come ConcreteSubject e SmartMonitor agisce come Observer e Proxy. Il Main istanzia un oggetto di Component e un oggetto di SmartMonitor che viene aggiunto alla lista degli osservatori dello stato dell’oggetto di tipo Component. Quando varia lo stato si invoca notifyObservers() che invoca update(s) che chiamando request(s) se non è presente un istanza di Monitor la crea ed esegue la request su tale istanza (nel codice una semplice print, nella realtà potrebbe essere una qualche operazione onerosa).

**Si discutano le conseguenze della implementazione dell’Observer in modo push o pull, evidenziando nei due casi la diversa struttura della signature dei metodi di notifica e update.**

Il design pattern Observer può essere implementato secondo due approcci, uno push e uno pull. Nel primo caso sarà necessario inserire nella signature di notify() e in seguito di update() lo stato che si desidera passare all'Observer, in questo caso l'intero state. Se fossero presenti più stati di interesse sarebbe però necessario distinguere tra i vari Observer o incapsulare tutti i dati in un oggetto. Secondo l'approccio pull invece non vengono passati parametri in quanto è compito del concreteObserver, venuto a conoscenza di una modiﬁca, recuperare lo stato (se gli interessa, cosa che permette maggiore ﬂessibilità della tecnica push nell'aggiornare il prorio stato), prevedendo però che l'Observer conosca il Subject da interrogare. Più semplicemente possiamo dire che nel modello Push si ha il passaggio dei parametri con la chiamata *update(parameters)* mentre in quello Pull si ha la chiamata ad *update()* senza parametri.

**Si dettaglino frammenti di codice della realizzazione che illustrano gli aspetti salienti dello schema, e si definisca uno scenario di test, realizzato per semplicità nella forma di un main(), che esercita lo schema in uno scenario che ne caratterizza l’intento.**

1. **import** java.util.ArrayList;
2. **import** java.util.List;
4. **public** **class** Main {
5. **public** **static** **void** main(String[] args) {
6. Subject component = **new** Component();
7. SmartMonitor sm = **new** SmartMonitor();
8. component.attach(sm);
9. component.setState(10);
10. component.setState(11);
11. //...
12. component.setState(12);
13. component.setState(13);
14. component.detach(sm);
15. component.setState(14);
16. }
17. }
19. **abstract** **class** Subject {
20. **private** List<Observer> observers = **new** ArrayList<Observer>();
22. **public** **void** attach(Observer o) {
23. observers.add(o);
24. }
26. **public** **void** detach(Observer o) {
27. observers.remove(o);
28. }
30. **protected** **void** notifyObservers() {
31. **for** (Observer o : observers)
32. o.update(**this**);
33. }
35. **public** **abstract** **void** setState(**int** s);
37. **public** **abstract** **int** getState();
38. }
40. **interface** Observer {
41. **void** update(Subject s);
42. }
44. **class** Component **extends** Subject {
45. **private** **int** state;
47. @Override
48. **public** **void** setState(**int** state) {
49. **this**.state = state;
50. notifyObservers();
51. }
53. @Override
54. **public** **int** getState() {
55. **return** state;
56. }
57. }
59. **interface** ProxyMonitor {
60. **void** request(Subject s);
61. }
63. **class** Monitor **implements** ProxyMonitor {
64. **private** **int** state;
66. @Override
67. **public** **void** request(Subject s) {
68. state = s.getState();
69. System.out.println("Stato aggiornato in: " + state);
70. }
71. }
73. **class** SmartMonitor **implements** Observer, ProxyMonitor {
74. **private** Monitor monitor;
75. **private** Subject subject;
77. @Override
78. **public** **void** update(Subject s) {
79. subject = s;
80. **this**.request(subject);
81. }
83. @Override
84. **public** **void** request(Subject s) {
85. **if** (monitor == **null**){
86. monitor = **new** Monitor();
87. System.out.println("Creato Monitor");
88. }
89. monitor.request(subject);
90. }
91. }

(Il seguente codice deve essere inserito all’interno di un unico file *Main.java)*

**Output:**

*Creato Monitor*

*Stato aggiornato in: 10*

*Stato aggiornato in: 11*

*Stato aggiornato in: 12*

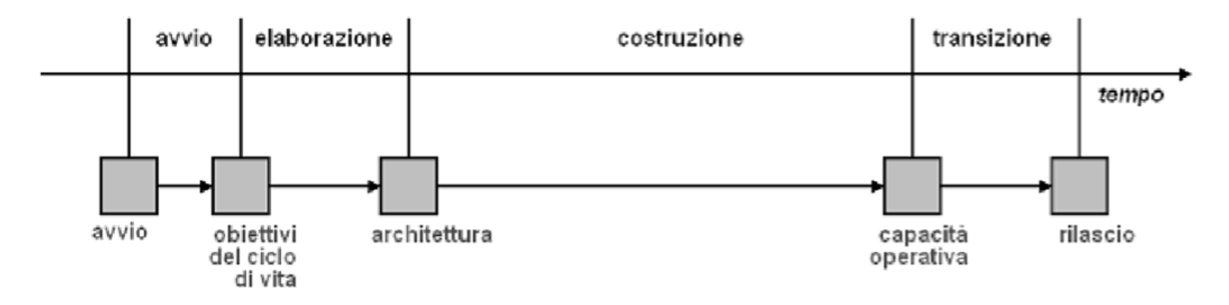
*Stato aggiornato in: 13*

***Process finished with exit code 0***

**Domanda 2:**

**Si illustrino le caratteristiche del modello di sviluppo Unified Process, evidenziando in che senso il modello è incrementale, iterativo e incentrato sull’architettura, e dettagliandone le fasi, i prodotti attesi al termine di ciascuna fase e i flussi di attività nel tempo.**

Il modello di sviluppo Unified Process (UP) è un modello OO fortemente basato su UML per lo sviluppo ed il ciclo di vita di un software. Si parla di modello incrementale perché ogni parte è divisa in incrementi; iterativo perché dopo un certo lasso di tempo ci ritroviamo ad eseguire lo stesso compito (in maniera iterativa); incentrato sull’architettura perché si parte da un’architettura iniziale definita nei primi incrementi e successivamente viene estesa(UP è basato su un’architettura). Tale modello fa un grande utilizzo di casi d’uso e fattori di rischio. È suddiviso in 4 fasi principali:



**Avvio:** si delineano gli obiettivi di progettazione e si studia come raggiungerli, vengono definiti anche i requisiti e i rischi maggiori. Si ha uno studio di fattibilità e il consenso manageriale.

**Elaborazione:** viene progettato il sistema hardware e software, rivedendo se necessario i requisiti. Si crea la documentazione fino a raggiungere l’80% dei casi d’uso, gli attributi di qualità, si pianifica la costruzione e si valutano le risorse necessarie.

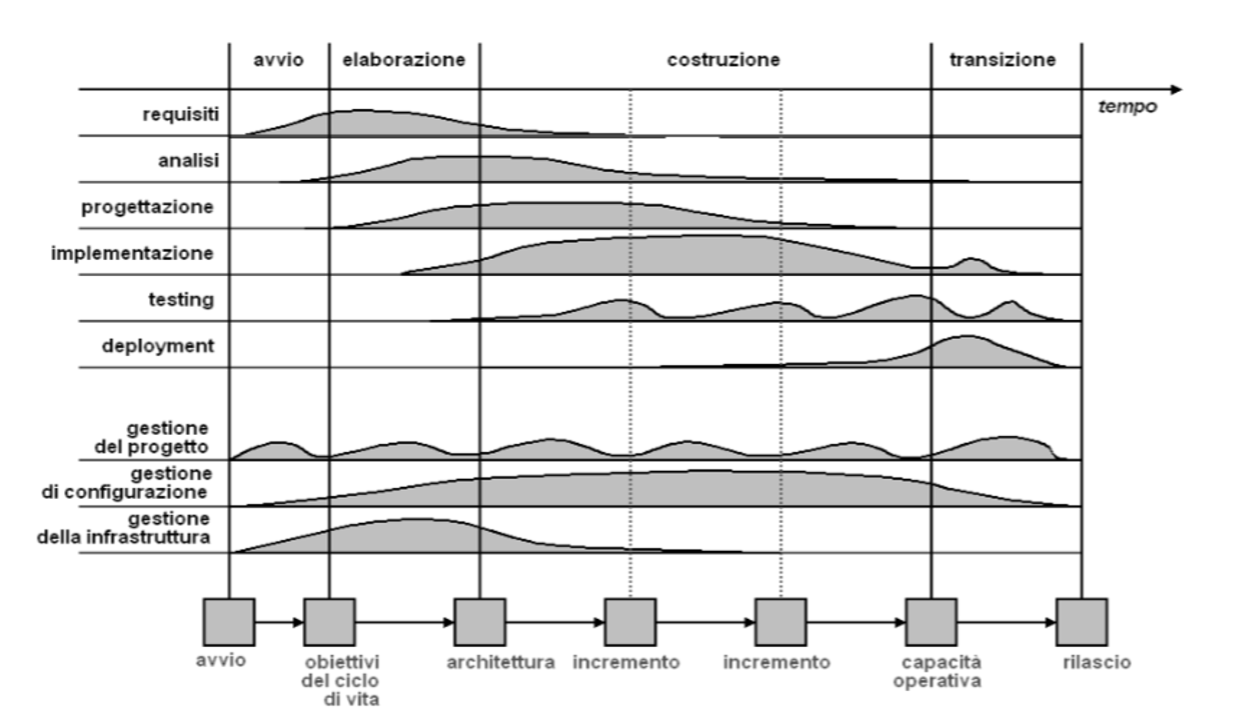
**Costruzione:** si realizza il sistema del progetto e si estende in maniera incrementale. Si popola il database co valori di prova (beta-test).

**Transizione:** vengono fatti i test su sistema e applicazioni fino a perfezionare il progetto. È l’ultima fase dove si esegue manutenzione e si verifica il corretto funzionamento.

*Ogni fase è articolata in iterazioni e ciascuna di esse è organizzata come un sotto progetto ed hanno la durata normalmente da 1 a 3 mesi. La fase con più interazioni è quella di costruzione, alla fine di ogni interazione si ha una milestone ovvero delle procedure che portano la giusta progressione del progetto.*

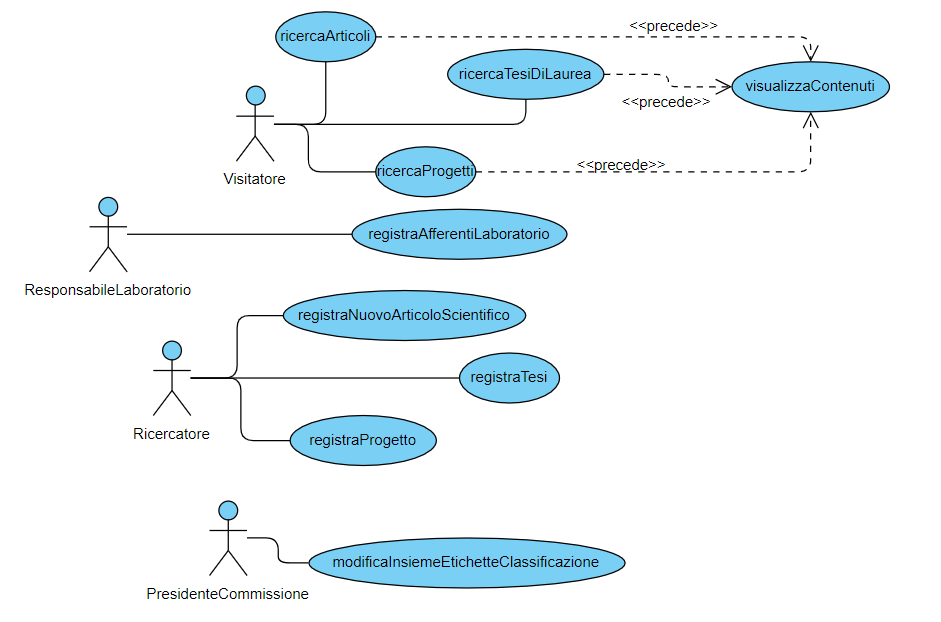
*Si consegnano i prodotti intermedi e viene fatta una overview del lavoro svolto; ogni interazione è composta dalle stesse fasi del modello waterfall. Nella fase di requisiti si modellano i requisiti e si mappano i casi d’uso. Nella fase di analisi si individuano le classi di analisi e le loro relazioni. Nella fase di progettazione si individua la relazione tra documenti di analisi e progetto cercando di rendere sempre meno ampia la separazione tra queste due. Nella fase di implementazione si scelgono e individuano i modelli di implementazione (come per esempio i design pattern). Nella fase di test vengono effettuate le prove finali che precedono il rilascio.*

Il carico di flusso di lavoro è mostrato nel seguente grafico; si nota che il lavoro viene distribuito in maniera da non “sovraccaricare” i progettisti.

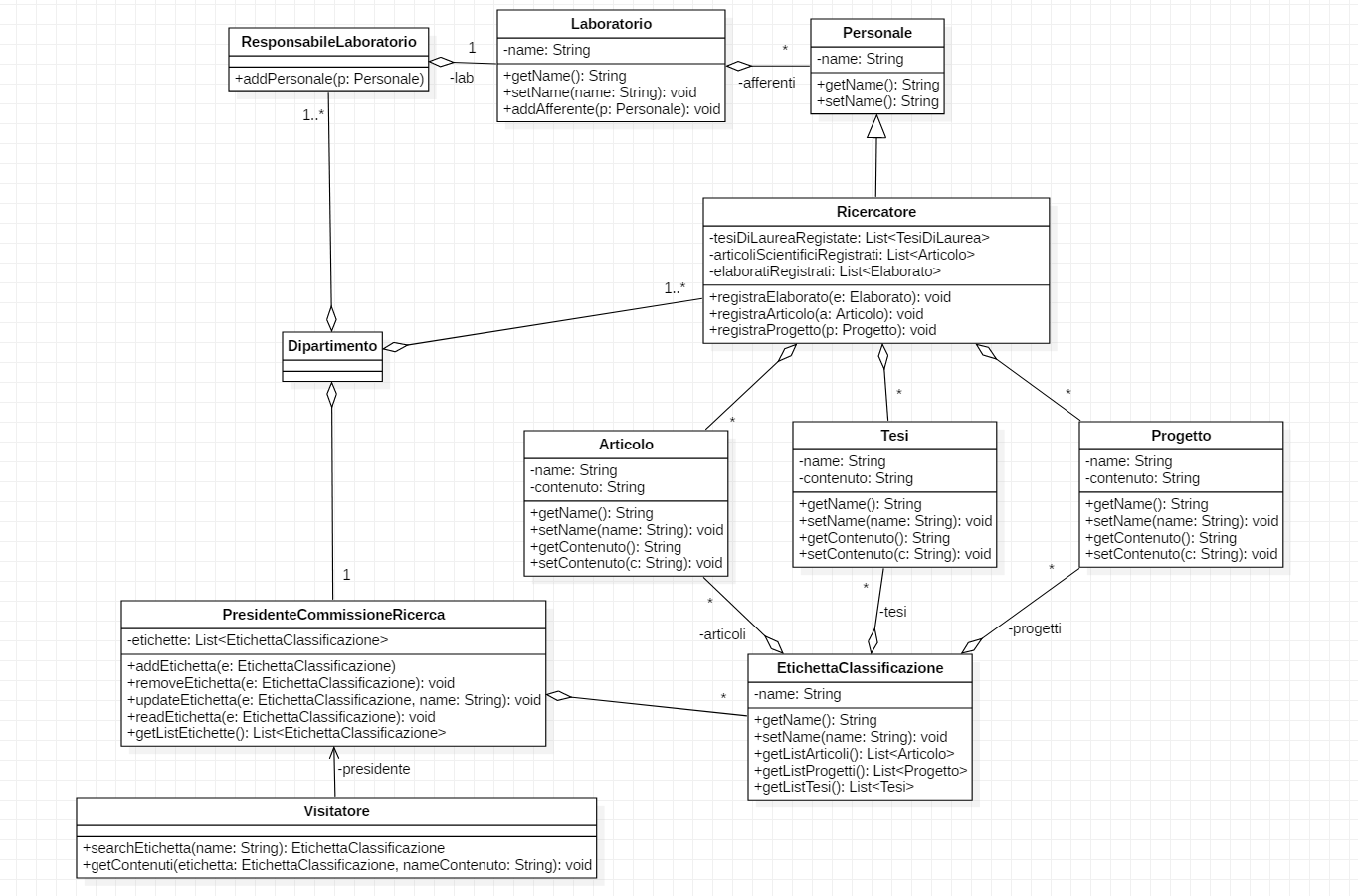


**Parte 2:**

**Si descrivano i casi d`uso delineati nella descrizione attraverso uno use case diagram.**

****

**Si definisca una logica di dominio in prospettiva di specifica/implementazione capace di abilitare le funzioni identificate nei casi d’uso.**

****

**Facendo riferimento alle classi delineate nel modello si discuta l’implementazione dei metodi fondamentali nella realizzazione di alcuni casi d’uso selezionati, specificando per ciascuno di tali metodi in quale classe è allocato e quali sono le collaborazioni attraverso cui è realizzato.**

*Classe Ricercatore:*

Collaborazioni: Tesi, Progetto, Articolo.

I metodi registra…() aggiungono nelle tre rispettive liste di contenuti un riferimento ad un oggetto tra Articolo, Progetto e Tesi.

*Classe ResponsabileLaboratorio:*

Collaborazioni: Laboratorio, Personale.

Il metodo addAfferente() chiamato per forwarding aggiunge alla List<Personale> afferenti = new ArrayList<Personale>(); un riferimento ad un istanza di personale.

*Classe PresidenteCommissioneRicerca:*

Collaborazioni: EtichettaClassificazione, Articolo, Tesi, Progetto.

I metodi permettono di aggiungere, rimuovere, aggiornare, leggere e ricevere la lista delle Etichette gestite dalla classe.

*Classe Visitatore:*

Collaborazioni: PresidenteCommissioneRicerca, EtichettaClassificazione, Articolo, Tesi, Progetto.

Il metodo searchEtichetta(name) ritorna il riferimento all’oggetto con l’etichetta con nome == name.

Il metodo getContenuto() riceve il riferimento sopra trovato e seleziona il contenuto con il nome passato, esegue successivamente la stampa del contenuto.

**Si descriva la logica di dominio attraverso un class diagram in prospettiva concettuale.**

