Configuration Management

Versione 4.1

Elisa Antolli

Alice Culaon

Diego Pillon

Data Creazione: 04/12/2015

Data ultima modifica: 28/04/2015

INDICE

1. TABELLA DELLE REVISIONI 3

2. SCOPO DEL DOCUMENTO 3

3. AUTORI DEL DOCUMENTO 3

4. GLOSSARIO 4

5. MEMBRI DEL TEAM 4

6. RESPONSABILITA' DI PROGETTO 5

7. COMUNICAZIONI 6

8. L' AMBIENTE, I TOOLS E L' INFRA-STRUTTURA 6

9. GESTIONE DOCUMENTI 7

10. MODELLO DI SVILUPPO 14

11. CICLO DI VITA 15

12. LA GESTIONE DEI CAMBIAMENTI 17

13. STILE DEL CODICE 19

APPENDICE A:
CODE CONVENTIONS FOR THE JAVA™ PROGRAMMING LANGUAGE 20

1. TABELLA DELLE REVISIONI

Rev./Ver.	Data	Descrizione	Autore
1/1.0	04-12-2014	Creazione del Documento; definizione di alcune parti	Elisa Antolli
2/2.0	23-12-2014	Integrazione del documento	Alice Culaon
3/3.0	07-03-2015	Applicazione del Modello stilistico per i documenti	Diego Pillon
4/3.1	24-03-2015	Aggiustamenti dei contenuti	Elisa Antolli Alice Culaon
5/3.2	24-03-2015	Aggiustamenti dei contenuti e del layout	Diego Pillon
6/3.3	30-03-2015	Integrazione dei paragrafi sul modello di sviluppo e sul ciclo di vita e gestione dei cambiamenti	Diego Pillon
7/4.0	22-04-2015	Integrazione di nuovi contenuti per il CM; definizione di una nuova regola di namig, più corretta; correzione di alcuni dati e raffinamento del Layout; Indipendenza del CM dal singolo progetto; eliminazione paragrafo sulla Metodologia Applicata;	Diego Pillon Alice Culaon
8/4.1	28-04-2015	Inclusione dell'Appendice sullo stile della codifica Java	Diego Pillon

Tot. Rev. 8	**Versione corrente 4.1**

2. SCOPO DEL DOCUMENTO

Lo scopo del CM è di raggiungere un livello di efficienza all'interno del gruppo di lavoro tale che permetta di affrontare l'attività di produzione del software in maniera ingegneristica.

Il documento descrive tecniche e metodologie applicate allo sviluppo e perseguite dal team. Si divide in tre fasi:

1. identificare gli elementi;
2. controllare le modifiche;
3. conoscere lo stato di un sistema;
4. In esso si definiscono anche:
   - 4.1. gli strumenti;
   - 4.2. i ruoli del team;
   - 4.3. le attività;
   - 4.4. le responsabilità;
   - 4.5. le caratteristiche del prodotto;

3. AUTORI DEL DOCUMENTO

1. *Elisa Antolli;*
2. *Alice Culaon;*
3. *Diego Pillon;*

4. GLOSSARIO

Termine	Descrizione	Pagine
CM	Configuration Management	3, 5, 8, 9
Sistema	l'insieme delle persone, dei servizi, e delle infrastrutture (fisiche, digitali) che fanno parte del progetto e della gestione della sua attività	6, 12, 13, 14, 15, 17
MVC	Model view controller, metodologia di sviluppo utilizzata	14
Metodologia di sviluppo	consiste in una serie di metodi e procedure di modellazione, usate per sviluppare le caratteristiche di un progetto, inclusi in un processo: elenco delle indicazioni riguardanti i passi da intraprendere	4
Ciclo di vita	si riferisce al modo in cui una metodologia di sviluppo o un modello di processo scompongono l'attività di realizzazione di prodotti software in sottoattività fra loro coordinate, il cui risultato finale è il prodotto stesso e tutta la documentazione ad esso associata	3, 6, 7, 14, 15
Naming	Sistema di assegnazione dei nomi ai file e documenti	5, 8, 9, 34, 35
Versioning	gestione di versioni multiple di un insieme di informazioni	9

Termini del glossario: 7

5. MEMBRI DEL TEAM

Nome	Cognome	Matricola	E-mail
Elisa	Antolli	N.D.	elisa.antolli@gmail.com
Alice	Culaon	109785	culaon.alice@spes.uniud.it
Diego	Pillon	57812	pillon.diego@spes.uniud.it

Totale dei componenti: 3

6. RESPONSABILITA' DI PROGETTO

Oggetto	Tool utilizzato
Configuration Manager (CM)	
Elisa Antolli	Stabilire le politiche; Scrivere il piano di Configuration Management; Decisioni sulla configurazione dell'Ambiente di Sviluppo; Stabilire le Baselines; Integrazioni delle parti strutturali (Database, Controller, Interfacce,..);
Diego Pillon	Interazione con il Committente;
Controllore del Configuration Management (CCM)	
Diego Pillon	Stabilire il controllo del processo; Come effettuare modifiche al processo di sviluppo; Come e quando devono esserci nuove richieste;
Sviluppatore/programmatore	
Elisa Antolli	Implementazione del codice;
Alice Culaon	Creazione e compilazione del Database; Implementazione del Controller;
Diego Pillon	Implementazione e integrazione delle interfacce;
Analista	
Elisa Antolli	Modifiche all'Analisi dei Requisiti; Interazione (esecutiva) con il committente;
Controllore di Qualità	
Alice Culaon	Testing; Verifica della coerenza con gli standard di qualità; Verifica della coerenza con le linee guida dei modelli ufficiali; Test di usabilità ed accessibilità; Test automatizzato; Test di rischio;
Designer Grafico	
Diego Pillon	Personalizzazione grafica dei documenti e stile del layout; Schemi DFD; Modelli grafici dei prototipi;
Elisa Antolli	Schema DER;
Version Manager (Sistemista)	
Elisa Antolli	Controllo dei Backups; Gestione dei Server; Gestione degli accessi; Realizzazione della configurazione dell'ambiente di sviluppo e tools;
Diego Pillon	Storia del Progetto: cronologia delle modifiche; Gestione della documentazione; Gestione dei files di Progetto; Versionamento e Naming;
Totali Ruoli: 7	**Totale Attività: 32**

7. COMUNICAZIONI

7.1. *Comunicazioni Impresa:*
vengono organizzate riunioni calendarizzate per discutere delle attività da svolgere e dei compiti di ognuno.
Le comunicazioni all'interno del team sono effettuate mediante skype e whatsapp per le comunicazioni a distanza.
Per le comunicazioni vengono ufficializzate usando le email sopra riportate;

7.2. *Comunicazioni con i Clienti:*
le comunicazioni con i clienti sono effettuate di persona, via telefono, skype o email.

8. L' AMBIENTE, I TOOLS E L' INFRA-STRUTTURA

Di seguito vengono presentati gli strumenti usati dai vari membri del team per lo sviluppo del sistema:

Oggetto	Tool utilizzato
Documentazione	
Requisiti, Specifica	Microsoft Office 2013;
Diagrammi	Visual Paradigm 12.0 (v.CE_12_0_20150304); Paint.NET v3.5.10; Diagram Designer v.1.27.3; Carta e penna;
Cronologia	MS Project;
Sviluppo	
Linguaggio	Java Virtual Machine 1.7; Java + Spring framework; Spring tool suite 2007-14;
Database	MySQL 5 + Hibernate 4 with annotation;
Archiviazione e gestione documentale (Gestione di Versione + Backup)	Git - http://git-scm.com/downloads; Git Bash 1.9.4; Smart Git 6.5.1 version non-commercial; Maven Repository;
Ciclo di vita	Modello a Cascata
Preventivo	
Calcoli	MS Office Excel 2007 (12.0.4518.1014) MSO (12.0.4518.1000);
Design	
Wireframes	Paint.NET v3.5.10; Photoshop Cs5; Carta e penna;
Prototyping	Pencil v.2.0.5;
Mockup	Paint.NET v3.5.10; Photoshop Cs5;
Tecnologie Web e Front endTest	
Browsers	Firefox v.33.1; Firefox v.36.0.4; Chrome v.39.0.2171.71 m; Chrome v.41.0.2272.101 m; IE Tester (6, 7, 8, 9);

	IE 11.0; Safari (Versione 7.0.6);
Resolutions	1024 x 768; 1280 x 720; 1280 x 768; 1360 x 768; 1366 x 786
Front-end Mobile	Bootstrap
Tester	Firebug 2.0.6; JavaScript- jQuery 1.8.3 /jQuery u.i 1.9.2
Livello Sistema Operativo	
Sistemi Operativi	Windows 8.1 Windows Vista Home Basic SP2 Mac OS X Maverick
Antivirus	Avira v.15.0.8.656
Comunicazioni	
Comunicazioni vocali/ riunioni	Skype 7.1.0.105; Incontri calendarizzati di persona;
Messaggistica	WhatsApp; Servizi email (google: gmail);
Livello Hardware	
Calcolatore usato da Elisa Antolli	Processore intel core i5 Mem. Ram 4G HD 500G
Calcolatore usato da Alice Culaon	Macbook Air Processore 1,3 GHz Intel Core i5 Memoria 4 GB 1600 MHz DDR3 Grafica Intel HD Graphics 5000 1536 MB
Calcolatore usato da Diego Pillon	Intel Pentium Dual CPU T3400 (2,16 Ghz, 2,17Ghz) 32-bit Mem. Ram 4Gb NVIDIA GeForce 8200M G HD 200G
Testing	
Metodologia di test	Test dinamico: Glass-box (JUnit versione 4)

9. GESTIONE DOCUMENTI

I sistemi software sono in costante evoluzione. La manutenzione del software, cioè, l'attività di modifica ai dispositivi esistenti, può consumare il 75% del costo totale del loro ciclo di vita. Circa il 20% di tutti gli sforzi di manutenzione viene utilizzato per correggere i difetti di progettazione e il restante 80% sono utilizzati nell'adattamento software, attività che tende al cambiamento dei requisiti funzionali, regole aziendali e l'applicazione di tecniche di reingegnerizzazione.
La gestione dei documenti nel Configuration Management Software nasce dalla necessità di controllare questi cambiamenti mediante metodi e strumenti, al fine di massimizzare la produttività e ridurre al minimo gli errori durante l'evoluzione.

9.1. *Repository:*
Attraverso l'uso della piattaforma BitBucket, uno strumento online illimitato per

l'archiviazione, condivisione e il controllo delle versioni dei file, viene creato un repository per ogni progetto, che accompagna e raccoglie tutta la documentazione e il codice. Ogni Progetto avrà una sua radice e sarà collocato nello spazio on line reso disponibile da BitBucket.

9.2. *Naming:*

Per quanto riguarda i documenti di lavoro, essi dovranno rispettare le regole di naming sotto elencate:

1. Iniziali (i) nome e cognome committente o ragione sociale numero progressivo del cliente.
2. Iniziali (i) Nome progetto + numero progressivo progetti (relativi al cliente):
3. Iniziali (i) Nome Documento.
   Le iniziali vengono usate per non allungare troppo i nomi di ogni singolo documento
4. Quindi i nomi di ogni documento si presenteranno nella seguente forma, dove "i" indica l'uso delle iniziali di ciò che segue, "n." indica un numero progressivo che sarà globale per i committenti, mentre sarà relativo al singolo committente nel caso dei progetti:

`iNomeCommittente[n.Progressivo]_iNomeProgetto[n.Progressivo]_iNomeDocumento`

5. L'uso dei numeri progressivi sarebbe sufficiente di per sé a identificare i committenti e i progetti, senza l'ausilio delle iniziali, ma, si sceglie di usare anch'esse per dare maggior forza di identificazione, anche visiva, per un utente, in caso di manipolazione diretta dei file.

   Esempio : Partendo dai seguenti dati ipotetici:

   Nome Comittente: Maurizio Pighin , n. progr. committente = 12 (essendo il 12° committente dell'azienda);
   Nome Progetto: Fumetteria n. progr. progetto = 3 (essendo il 3° suo progetto);

   Otteniamo la seguente tabella di nomi per i documenti relativi a quel progetto:

Tipo Documento	Nome Documento in Progetto
Configuration Management (o Configuration Plan)	CM.docx
Analisi dei Requisiti	MP[1]_F[1]_AR.docx
Specifica dei Requisiti	MP[1]_F[1]_SR.docx
Metriche	MP[1]_F[1]_MT.docx
Analisi dei Costi	MP[1]_F[1]_AC.docx
Qualità	MP[1]_F[1]_QU.docx
Pianificazione della tempistica	MP[1]_F[1]_PT.docx
Preventivo	MP[1]_F[1]_PR.docx
Documento di progetto	MP[1]_F[1]_DP.docx
Testing	MP[1]_F[1]_TE.docx
Consuntivo	MP[1]_F[1]_CO.docx
Manuale di installazione	MP[1]_F[1]_MI.docx
Manuale utente	MP[1]_F[1]_MU.docx
Verbale 1	MP[1]_F[1]_V1.docx
Manutenzione (o Contratto Manutentivo)	MP[1]_F[1]_MA.docx

Totale dei docuementi = 15

Strutturando il naming in questa maniera, si ritiene che sia più semplice identificare un determinato documento senza particolari sforzi. La ricerca di un singolo documento sia semplificata, in quanto prima viene identificato il committente, poi il progetto e infine il documento; nel decidere l'adozione di tale tecnica, si sono fatte le seguenti assunzioni: si presume che:

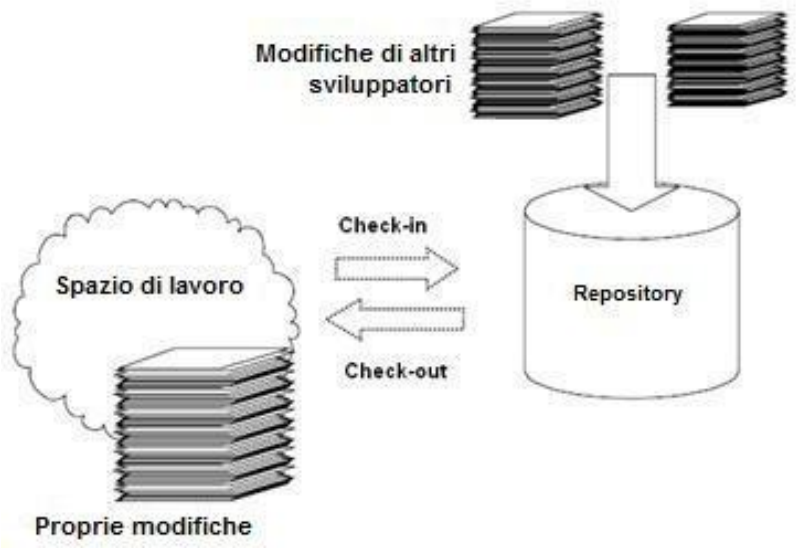
5.1. un committente possa incaricarci di più progetti;
5.2. vi possano essere più committenti con le medesime iniziali;
5.3. vi possano essere più progetti con la stessa iniziale per lo stesso committente;
5.4. che non vi possono essere essere più documenti con le stesse iniziali all'interno del progetto, essendo queste fissate;
5.5. ogni verbale avrà un suo documento con un ulteriore numero progressivo assegnato (relativo al progetto).

6. Caso a parte sarà il CM che, essendo unico, verrà chiamato solo CM.

9.3. Versioning:

All'interno dell'impresa usiamo Git attraverso Sourcetree e Smart-Git per controllare le operazioni di "check-in" e "check-out".
Il sistema di controllo di versione consente ai file di essere ottenuti attraverso un'operazione conosciuta come "check-out"; essi potranno essere modificati nell'ambiente di lavoro dello sviluppatore e poi essere restituiti al repository attraverso un'operazione conosciuta come "check-in", come esemplificato nelle Figure seguenti.

Figura 1



Il repository è uno spazio di memorizzazione dei files che è sotto il controllo del CM. Questi files sono chiamati elementi di configurazione. Quando il file viene aggiunto la prima volta al repository, l'oggetto comincia dalla versione 1 e così via. Ad ogni operazione di check-in eseguita, la versione dell'elemento di configurazione viene incrementata in uno.

Per ciascun file memorizzato, sono allegate informazioni quali, data di creazione o di modifica, i commenti e le versioni.

Figura 2

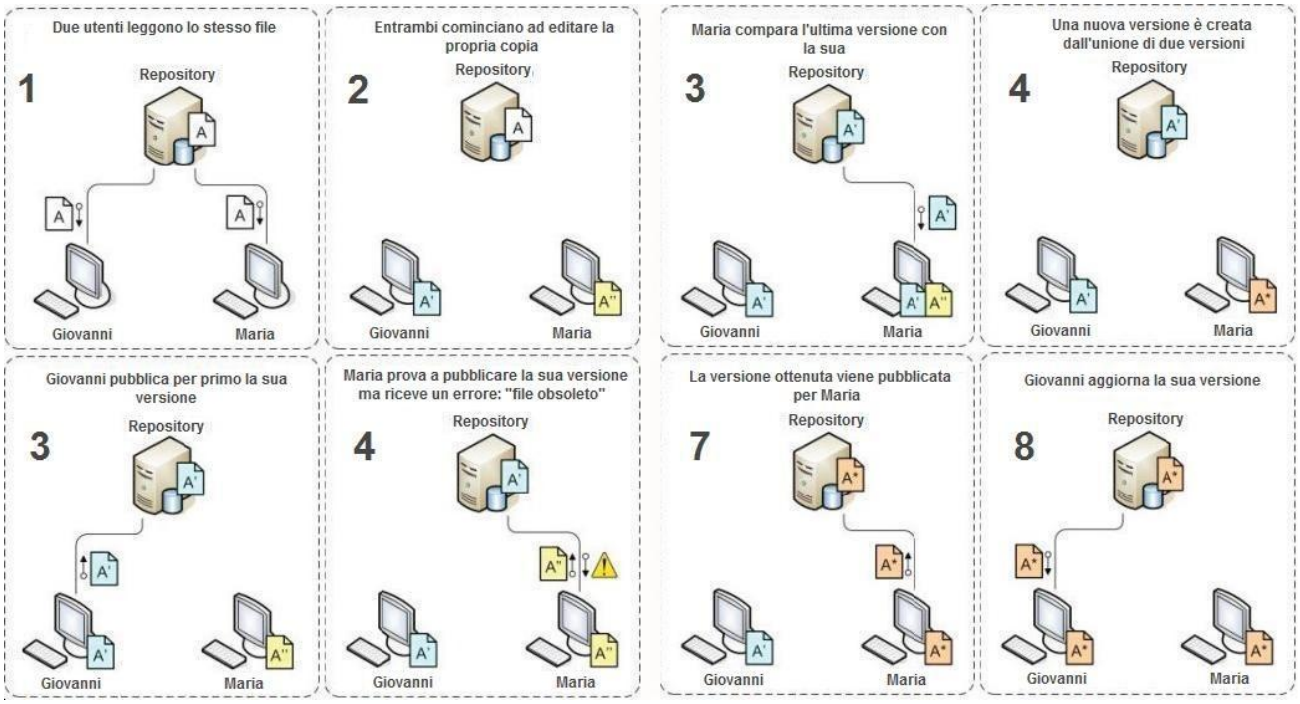


Figura 3

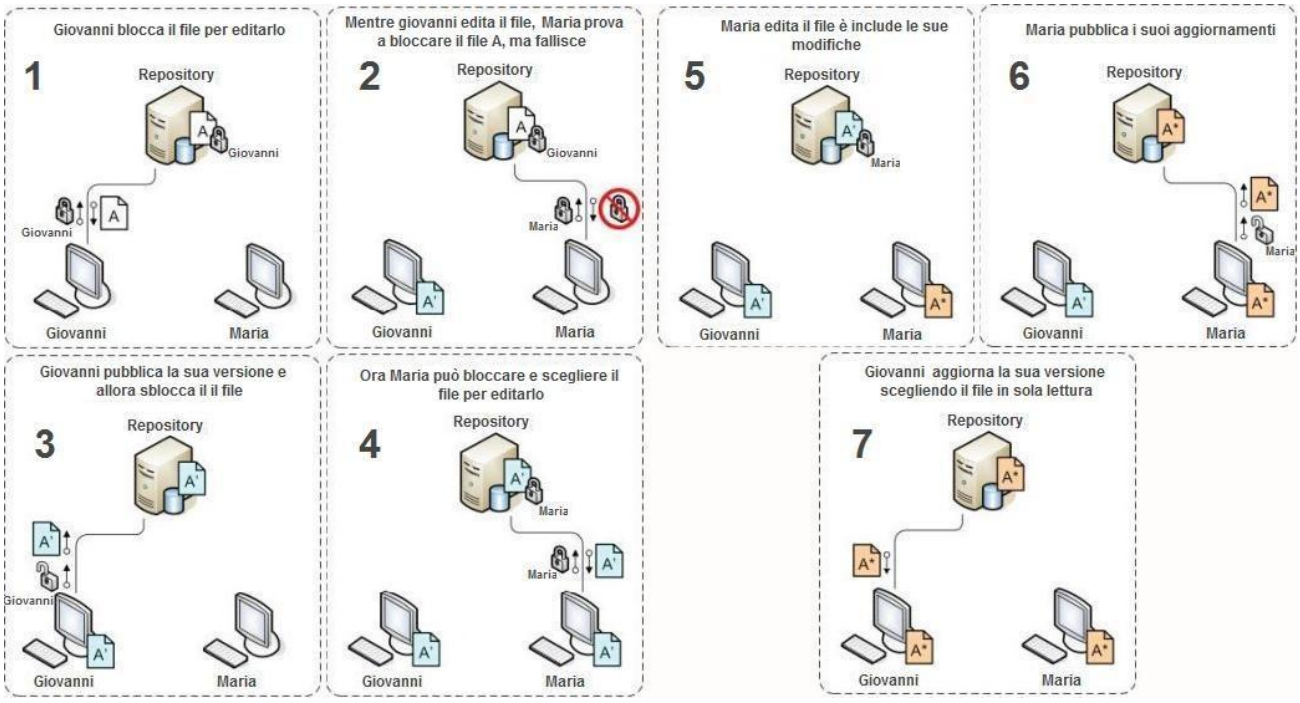
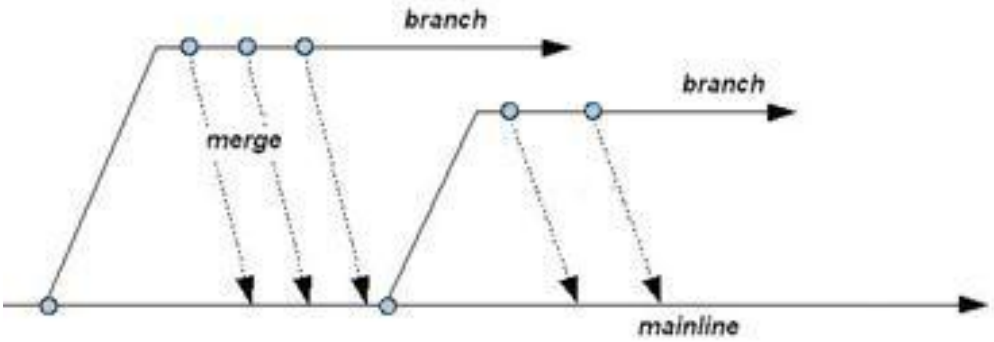


Figura 4



Durante l'utilizzo di un repository si possono riconoscere i seguenti termini:

1. *Mainline*: Ramo di sviluppo principale del progetto.
2. *Branch*: Ramo di sviluppo: ad opera di uno o più sviluppatori, può essere la mainline oppure un ramo di sviluppo secondario;
3. *Commit*: E' l'azione con cui si attua una revisione particolare o anche solo la pubblicazione di un nuovo documento;
4. *Merge*: E' la fusione che tipicamente avviene tra più versioni di uno stesso documento;

9.4. *Backup:*

Il backup viene effettuato attraverso il repository in maniera automatica attraverso il cloud.

9.5. Struttura generale dei documenti e loro contenuto:

1. *Configuration Plan:*
   descrive le politiche organizzative del team e le metodologie applicate per le gestione e lo sviluppo del progetto. Questo documento è unico e rappresenta una linua guida per tutti i progetti. Paragrafi:
   - *1.1. MEMBRI DEL TEAM;*
   - *1.2. RESPONSABILITA' DI PROGETTO;*
   - *1.3. METODOLOGIA APPLICATA;*
   - *1.4. COMUNICAZIONI;*
   - *1.5. L'AMBIENTE, I TOOL E L' INFRA-STRUTTURA;*
   - *1.6. GESTIONE DOCUMENTI;*
   - *1.7. MODELLO DI SVILUPPO;*

2. *Analisi dei Requisiti:*
   il cui scopo è quello di definire le funzionalità che il nuovo prodotto (o il prodotto modificato) deve offrire, ovvero i *requisiti che devono essere soddisfatti* dal software sviluppato. Paragrafi:
   - *2.1. BACKEND;*
   - *2.2. FRONTEND;*

3. *Specifica dei requisiti:*
   formalizza i requisiti presentati nel documento dell'Analisi dei Requisiti, per di disambiguare le richieste dell'utente, utilizzando un linguaggio più tecnico e strutturato del linguaggio naturale. Paragrafi:
   3.1. REQUISITI FUNZIONALI;
   3.2. MATRICE DI TRACCIABILITÀ;

4. *Metriche:*
   si definiscono le metriche di valutazione del prodotto software. Paragrafi:
   4.1. IL MODELLO GOAL QUESTION METRIC;

5. *Analisi dei costi:*
   contiene una stima iniziale del costo del progetto complessivo. L'utilità di questa analisi è quella di stabilire un quadro di riferimento che permetta di stimare realisticamente i costi che l'intero progetto richiederà di affrontare. Paragrafi:
   5.1. IL MODELLO COCOMO;
   5.2. IL MODELLO COCOMO II;
   5.3. STIMA DEL COSTO COCOMO;
   5.4. STIMA DEL COSTO COCOMO II;
   5.5. COSTI ESTERNI;
   5.6. COSTO FINALE;
   5.7. APPROFONDIMENTO PARTE SVILUPPATA;

6. *Qualità:*
   è un documento in cui si definisce il processo di controllo della qualità, il concetto di qualità e i processi attuati per il raggiungimento di un buon progetto. Paragrafi:
   6.1. DEFINIZIONE DI QUALITÀ;
   6.2. METRICHE;
   6.3. STANDARD PER LA DOCUMENTAZIONE E PER IL CODICE SORGENTE;
   6.4. APPENDICE 1 – TABELLA DI RIFERIMENTO ACD;
   6.5. VERIFICA OBIETTIVI DI QUALITÀ;
   6.6. VERIFICA OBIETTIVI DI QUALITÀ ACCESSORI;
   6.7. STANDARD PER LA DOCUMENTAZIONE E PER IL CODICE SORGENTE;

7. *Pianificazione della tempistica:*
   il documento contiene il cronogramma secondo cui il progetto verrà realizzato. Paragrafi:
   7.1. CRONOGRAMMA;

8. *Preventivo:*
   il documento contiene il prezzo preventivato offerto al cliente tenendo conto dei costi sostenuti dal team per la progettazione e sviluppo del sistema. Paragrafi:
   8.1. DESCRIZIONE DEL PRODOTTO;
   8.2. PROPOSTA DI OFFERTA SISTEMA FUMETTERIA;
   8.3. TERMINI DI CONSEGNA;
   8.4. PRESTAZIONI ACCESSORIE;
   8.5. CANONE DI ASSISTENZA E MANUTENZIONE;

9. Documento di progetto:
   si articola ad un livello ancora più approfondito e dettagliato dell'analisi dei requisiti e delle specifiche: documento stabilisce i passi e le tecniche per la progettazione del sistema. Paragrafi:
   9.1. ARCHITETTURA DEL SISTEMA;
   9.2. DESCRIZIONE DELLE CLASSI;
   9.3. DATABASE CENTRALE;
   9.4. INTERFACCE;

10. *Testing:*
    definisce le modalità di test da eseguire sul software prodotto, con lo scopo di verificare e validare il sistema. Paragrafi:
    10.1. METODOLOGIA DI TESTING;
    10.2. CATENE DI TEST;

11. *Consuntivo:*
    confronta il preventivo e l'analisi dei costi, con i risultati concreti ottenuti alla fine del progetto tenendo conto del numero di ore lavorate da ciascun membro del team
    Paragrafi:
    11.1. ORE DI LAVORO;
    11.2. PREVENTIVATO;
    11.3. COSTI EFFETTIVI;
    11.4. CONSIDERAZIONI FINALI;

12. *Manuale d'installazione:*
    il documento descrive come il sistema verrà installato al committente. Paragrafi:
    12.1. PROCEDURA DI INSTALLAZIONE;

13. Manuale utente: il documento ha come obiettivo l'agevolazione della comprensione dei servizi offerti dal sistema. Paragrafi:
    13.1. DESCRIZIONE SISTEMA;
    13.2. ELENCO FUNZIONI DEL SISTEMA;
    13.3. USO DEL SISTEMA;
    13.4. ELENCO ERRORI E LORO GESTIONE;
    13.5. CONTATTI E ASSISTENZA;

14. *Verbali:*
    Paragrafi:
    14.1. CONVOCAZIONE;
    14.2. CONCLUSIONI;
    14.3. CONVOCAZIONE PROSSIMA RIUNIONE;
    14.4. PUNTI DELL'ORDINE DEL GIORNO RIMANDATI;
    14.5. NUOVI PUNTI SU CUI DISCUTERE;

15. *Contratto Manutentivo:*
    Paragrafi:
    15.1. CICLO DI VITA;
    15.2. TRAINING SUL SULL'USO DEL SISTEMA;
    15.3. COPERTURE;
    15.4. ESCLUSIONI;

10. MODELLO DI SVILUPPO

10.1. Il modello MVC

Viene seguito il modello MVC, che è un pattern architetturale adatto per lo sviluppo di software con linguaggi orientati ad oggetti.
La peculiarità che contraddistingue il modello MVC è la separazione tra la logica di presentazione dei dati (l'interfaccia), dalla logica del business (la logica applicativa del software).
Si struttura come segue:

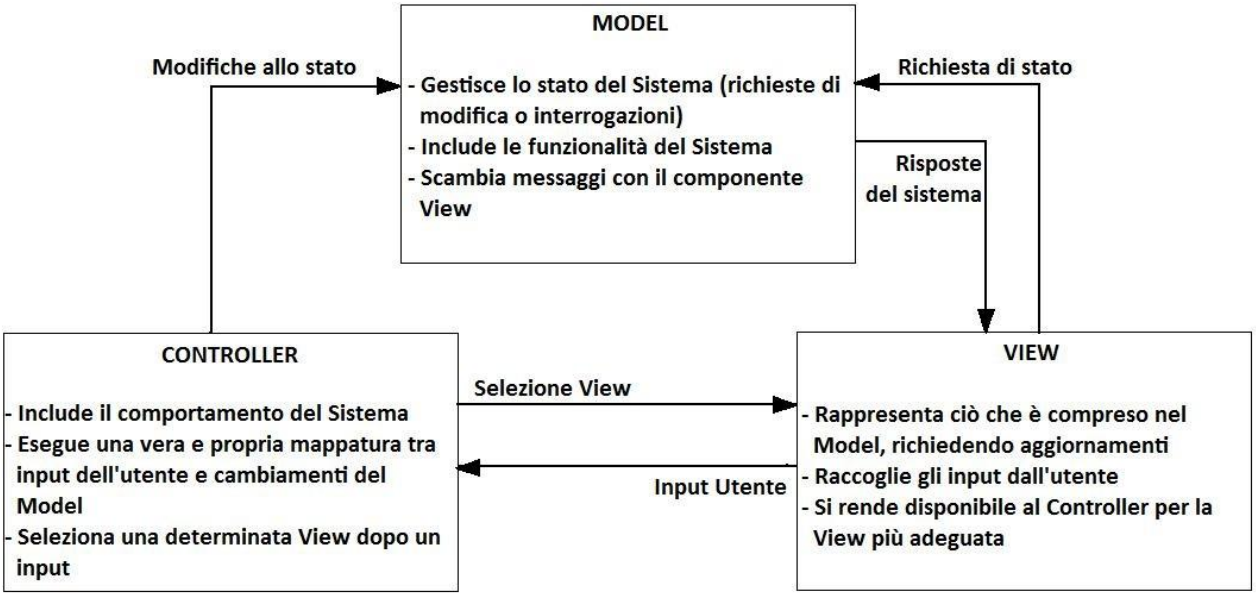
10.1.1. *Model*: livello di sistema costituito da dati, regole di business, la logica e le funzioni del sistema stesso; fornisce i metodi per accedere ai dati utili all'applicazione;
10.1.2. *View*: livello visivo che comprende logica del frontend, logica di presentazione; visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti;
10.1.3. *Control*: livello che fa da intermediario tra l'input; realizza la conversione dei comandi o per il livello Model o per il livello View; riceve i comandi dell'utente, solitamente per mezzo del view, e li attua modificando lo stato degli altri due componenti (del model e di se stesso).

E' necessario quindi che il sistema sia modulare, con un'interfaccia grafica costituita da più schermate, ognuna dedicata alla presentazione dei dati aggiornati.

10.2. *Considerazioni*

Questo modello è più complesso da progettare, ma permette una più più facile gestione di tutte le fasi di sviluppo e di successiva manutenzione, proprio in virtù della succitata separazione secondo lo schema proposto.
Inoltre incrementa e favorisce il riuso dei componenti.
Quindi decidiamo di investire maggiormente in progettazione per cercare di ridurre i costi successivi.

Figura 5



11. CICLO DI VITA

Da quando viene commissionato, ovvero da quando si manifesta la volontà di un "attore", che diverrà il committente, di voler disporre di un sistema a supporto della sua attività, un sistema software comincia il suo ciclo di vita, che si suddivide sostanzialmente in cinque fasi:

11.1. *Specifica:*
raccolta e definizione dei requisiti dell'utente

11.2. *Progettazione:*
è la fase di sviluppo: definizione delle funzioni e del comportamento del sistema;

11.3. *Implementazione:*
scrittura del codice;

11.4. *Validazione e test:*
serie di test sui singoli componenti e anche sulla loro integrazione, comprendente anche la validazione e test dei dati;

11.5. *Evoluzione e Manutenzione:*
gestione degli errori (sia di progettazione che di implementazione) o dei guasti e gestione di nuove richieste da parte dell'utente.
Questa fase sarà espressamente regolamentata da un contratto di manutenzione.

Ma in assenza di precedenti applicazioni di modelli e di consolidate esperienze, soprattutto nei passaggi tra specifica, progettazione e implementazione, abbiamo convenuto fosse meglio "personalizzare" il modello secondo quanto mostrato in Figura 7, per una rapparesentazione dello stesso quantomeno più fedele alla realtà.

Esso è una sorta di ibrido tra il modello a cascata e il modello e il modello iterativo.

Figura 6: Siamo partiti dal classico modello a cascata proposto da Ian Sommerville nel suo Libro..

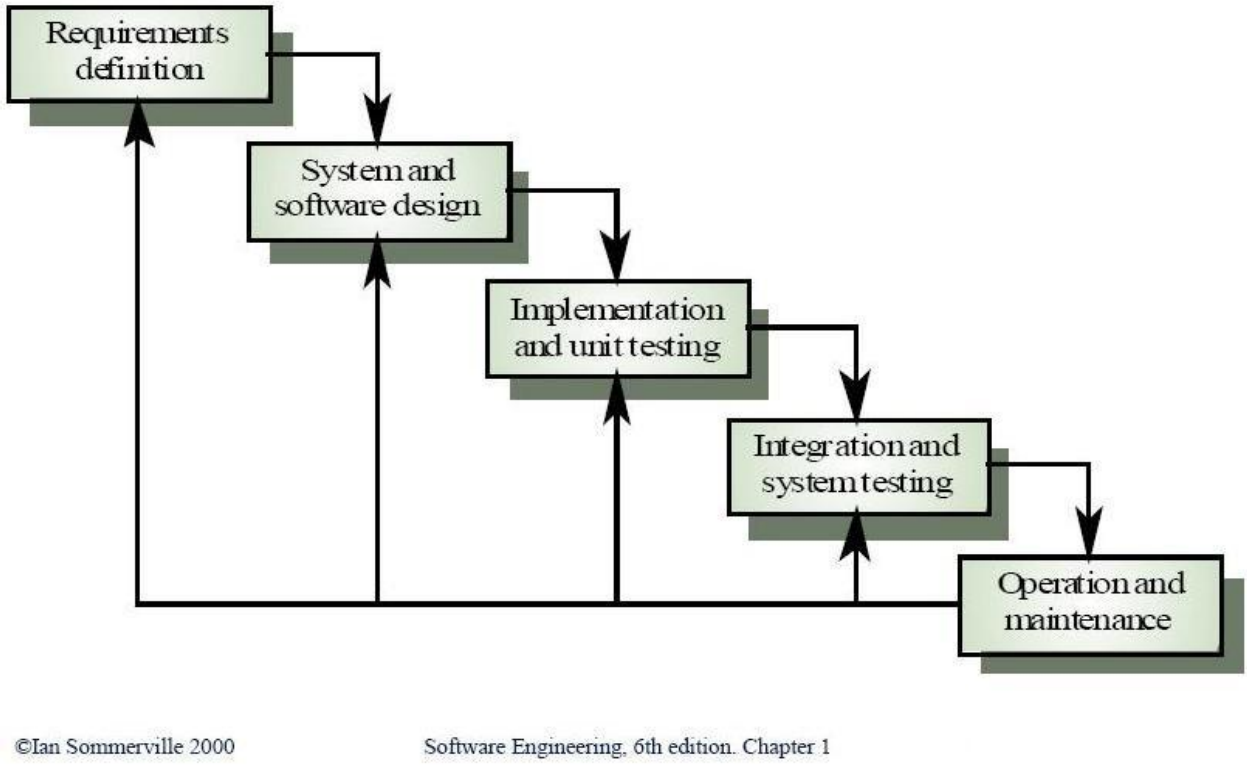
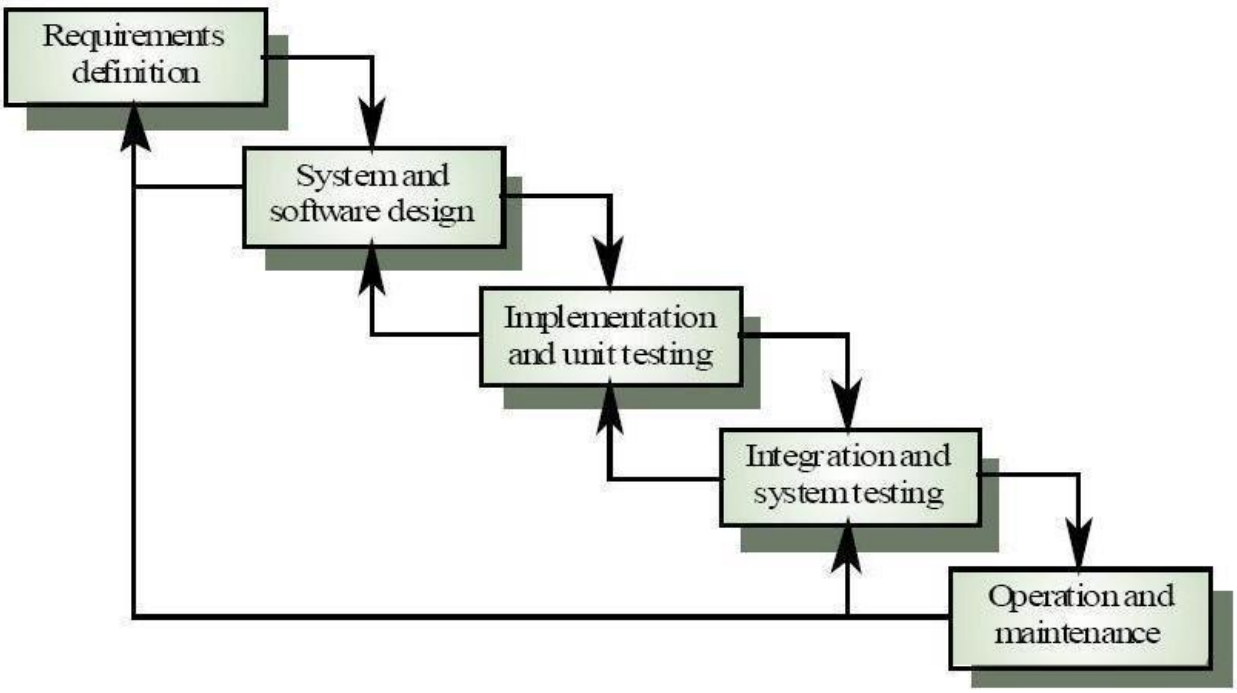


Figura 7 ...e l'abbiamo personalizzata a nostro uso.



In questo caso le attività di una fase successiva possono essere sospese per eventuali modifiche "in corso d'opera" che vadano a toccare anche la fase precedente in alcune sue caratteristiche o per l'integrazione di requisiti non richiesti, o per moduli a cui non si ha pensato in fase di progettazione, ecc...

La freccia finale, che si diparte dalla manutenzione e che fa capo ai requisti sta a rappresentare il caso in cui vi siano nuove richieste da parte del committente, cioè nuovi requisiti non espressi o non riconosciuti precedentemente.

12. LA GESTIONE DEI CAMBIAMENTI

Durante la progettazione di un software o, a maggior ragione, di un sistema software, è cosa normale che lo staff, che se ne occupa, si veda arrivare una certa quantità di richieste di cambiamenti o modifiche: sia di cosa debba essere compreso nel sofware stesso, sia di come questo software debba essere implementato o di come debba presentarsi, in varie fasi del processo di sviluppo e dipenedentemente da chi è promotore della singola proposta/richiesta in questione.
Le proposte/richieste di modifica posso provenire dai seguenti soggetti:

12.1. *Utenti:*
ovviamente da loro partono tutte quelle modifiche che andranno a influire la qualità del loro lavoro e l'efficienza operativa del sistema;

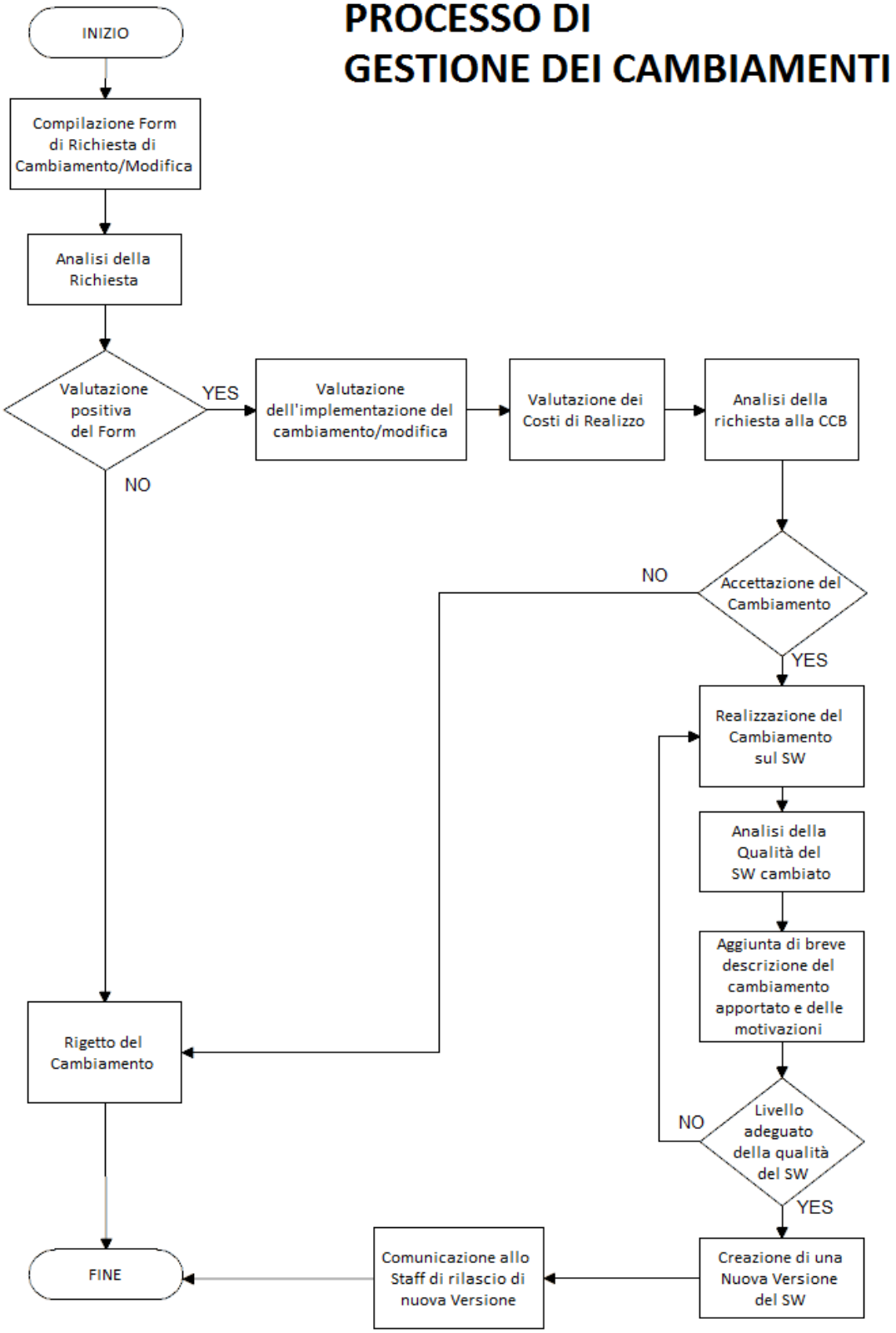
12.2. *Sviluppatori:*
le modifiche che verosimilmente queste figure andranno a proporre, avranno una natura più tecnica rispetto agli utenti;

12.3. *Dalle Forze di Mercato:*
sono modifiche tese a rispondere meglio a esigenze di tipo sia commerciale, sia di livello tecnologico laddove vi sia obsolescenza dei moduli/sistemi attuali.

IL PROCESSO DI GESTIONE DEI CAMBIAMENTI

```
01      Si parte dalla compilazione del form di richiesta;
02      Si analizza il form di richiesta
03      Se il form è valutato positivamente allora;

04            Si valuta come il cambiamento può essere implementato;
05            Si valutano i costi di realizzo;
06            Si sottopone la richiesta alla CCB;

07            Se il cambiamento è accettato
08            Allora si realizza il cambiamento;
09            Si allega breve descrizione del cambiamento apportato e delle
              motivazioni;
10            Si sottopone il SW cambiato per l'approvazione della qualità;

11            Si ripetono i tre passi precedenti (si torna al passo 8)
              finché la qualità non è ad un livello accettabile;

12            Si crea una nuova versione di sistema;
13            Comunicazione ai membri dello staff dell'esistenza di una
              nuova versione;
14      Se il cabiamento non è accettato
15            Allora si rifiuta la richiesta/proposta;
16      Se il form non è valutato positivamente
17            Allora si rifiuta la richiesta/proposta;
```

Figura 8


PROCESSO DI
GESTIONE DEI CAMBIAMENTI
INIZIO
Compilazione Form di Richiesta di Cambiamento/Modifica
Analisi della Richiesta
Valutazione positiva del Form
YES
NO
Valutazione dell'implementazione del cambiamento/modifica
Valutazione dei Costi di Realizzo
Analisi della richiesta alla CCB
Accettazione del Cambiamento
NO
YES
Realizzazione del Cambiamento sul SW
Analisi della Qualità del SW cambiato
Aggiunta di breve descrizione del cambiamento apportato e delle motivazioni
Livello adeguato della qualità del SW
NO
YES
Rigetto del Cambiamento
Creazione di una Nuova Versione del SW
Comunicazione allo Staff di rilascio di nuova Versione
FINE

13. STILE DEL CODICE

Nello scrivere il codice, vengono seguite regole e direttive aderenti ad una convenzione di codifica stabilita, anche per ragioni di coerenza e abba.

Nel nostro caso useremo la convezione della Sun Microsystems del 1999, adottata e messa a disposizione dalla ORACLE, che si può trovare in appendice a questo documento.

APPENDICE A
CODE CONVENTIONS FOR THE JAVA™ PROGRAMMING LANGUAGE

INTRODUCTION

Why Have Code Conventions

Code conventions are important to programmers for a number of reasons:

1. 80% of the lifetime cost of a piece of software goes to maintenance.
2. Hardly any software is maintained for its whole life by the original author.
3. Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
4. If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

For the conventions to work, every person writing software must conform to the code conventions. Everyone.

Acknowledgments

This document reflects the Java language coding standards presented in the *Java Language Specification*, from Sun Microsystems, Inc. Major contributions are from Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, and Scott Hommel.

This document is maintained by Scott Hommel. Comments should be sent to shommel@eng.sun.com

FILE NAMES

This section lists commonly used file suffixes and names.

File Suffixes

Java Software uses the following file suffixes:

File Type	Suffix
Java source	.java
Java bytecode	.class

Number of Suffixes = 2

Common File Names

Frequently used file names include:

File Name	Use
GNUmakefile	The preferred name for makefiles. We use gnumake to build our software.
README	The preferred name for the file that summarizes the contents of a particular directory

Used Names: 2

FILE ORGANIZATION

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

Files longer than 2000 lines are cumbersome and should be avoided.

For an example of a Java program properly formatted, see "Java Source File Example" on page 36.

Java Source Files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files have the following ordering:

1. Beginning comments (see "Beginning Comments" on page 21)
2. Package and Import statements
3. Class and interface declarations (see "Class and Interface Declarations" on page 21)

Beginning Comments

All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

Package and Import Statements

The first non-comment line of most Java source files is a package statement. After that, import statements can follow. For example:

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

Note: The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.

Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear. See "Java Source File Example" on page 36 for an example that includes comments.

N.	Part of Class/Interface Declaration	Notes
1	Class/interface documentation comment (/**...*/)	See "Documentation Comments" on page 26 for information on what should be in this comment.
2	class or interface statement	
3	Class/interface implementation comment (/*...*/), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
4	Class (static) variables	First the public class variables, then the protected, then package level (no access modifier), and then the private.
5	Instance variables	First public, then protected, then package level (no access modifier), and then private.
6	Constructors	
7	Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

Number of parts/declaration: 7

INDENTATION

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

Line Length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

***Note*:** Examples for use in documentation should have a shorter line length-generally no more than 70 characters.

Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

1. Break after a comma.
2. Break before an operator.
3. Prefer higher-level breaks to lower-level breaks.
4. Align the new line with the beginning of the expression at the same level on the previous line.
5. If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);

var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                          longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
           + 4 * longname6; // PREFER


longName1 = longName2 * (longName3 + longName4
                       - longName5) + 4 * longname6; // AVOID
```

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

Line wrapping for `if` statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();           //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
```

```
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

COMMENTS

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /*...*/, and //. Documentation comments (known as "doc comments") are Java-only, and are delimited by /**...*/. Doc comments can be extracted to HTML files using the javadoc tool.

Implementation comments are mean for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective. to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or nonobvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

***Note*:**The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters such as form-feed and backspace.

Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

```
/*
 * Here is a block comment.
 */
```

Block comments can start with /*-, which is recognized by **indent**(1) as the beginning of a block comment that should not be reformatted. Example:

```
/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 *    one
 *        two
 *            three
 */
```

***Note*:** If you don't use **indent**(1), you don't have to use /*- in your code or make any other concessions to the possibility that someone else might run **indent**(1) on your code.

See also "Documentation Comments" on page 26.

Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format (see section 5.1.1). A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java code (also see "Documentation Comments" on page 26):

```
if (condition) {
    /* Handle the condition. */
    ...
}
```

Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here's an example of a trailing comment in Java code:

```
if (a == 2) {
    return TRUE;            /* special case */
} else {
    return isPrime(a);      /* works only for odd a */
}
```

End-Of-Line Comments

The // comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else {
    return false;          // Explain why here.
}
//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

Documentation Comments

***Note*:** See "Java Source File Example" on page 36 for examples of the comment formats described here.

For further details, see "How to Write Doc Comments for Javadoc" which includes information on the doc comment tags (@return, @param, @see):

http://java.sun.com/products/jdk/javadoc/writingdoccomments.html

For further details about doc comments and javadoc, see the javadoc home page at:

http://java.sun.com/products/jdk/javadoc/

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters /**...*/, with one comment per class, interface, or

member. This comment should appear just before the declaration:

```
/**
 * The Example class provides ...
 */
public class Example { ...
```

Notice that top-level classes and interfaces are not indented, while their members are. The first line of doc comment (/**) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter.

If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment (see section 5.1.1) or single-line (see section 5.1.2) comment immediately *after* the declaration. For example, details about the implementation of a class should go in in such an implementation block comment *following* the class statement, not in the class doc comment.

Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration *after* the comment.

DECLARATIONS

Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level
int size;  // size of table
```

is preferred over

```
int level, size;
```

Do not put different types on the same line. Example:

```
int foo,  fooarray[]; //WRONG!
```

***Note*:** The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

```
int    level;              // indentation level
int    size;          // size of table
Object currentEntry;      // currently selected table entry
```

Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void myMethod() {
    int int1 = 0;         // beginning of method block

    if (condition) {
        int int2 = 0;     // beginning of "if" block
        ...
    }
}
```

The one exception to the rule is indexes of `for` loops, which in Java can be declared in the `for` statement:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // AVOID!
        ...
    }
    ...
}
```

Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

1. No space between a method name and the parenthesis "(" starting its parameter list
2. Open brace "{" appears at the end of the same line as the declaration statement
3. Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample extends Object {
    int ivar1;
    int ivar2;
```

```
        Sample(int i, int j) {
            ivar1 = i;
            ivar2 = j;
        }

        int emptyMethod() {}

        ...
    }
```

4. Methods are separated by a blank line

STATEMENTS

Simple Statements

Each line should contain at most one statement. Example:

```
argv++;       // Correct
argc--;       // Correct
argv++; argc--;       // AVOID!
```

13.1.1. Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`". See the following sections for examples.

1. The enclosed statements should be indented one more level than the compound statement.
2. The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
3. Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

return Statements

A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

if, if-else, if else-if else Statements

The `if-else` class of statements should have the following form:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else{
    statements;
}
```

Note: `if` statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

for Statements

A `for` statement should have the following form:

```
for (initialization; condition; update) {
    statements;
}
```

An empty `for` statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a `for` statement, avoid the complexity of using more than three variables. If needed, use separate statements before the `for` loop (for the initialization clause) or at the end of the loop (for the update clause).

while Statements

A `while` statement should have the following form:

```
while (condition) {
    statements;
}
```

An empty `while` statement should have the following form:

```
while (condition);
```

do-while Statements

A `do-while` statement should have the following form:

```
do {
    statements;
} while (condition);
```

switch Statements

A `switch` statement should have the following form:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */

case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;

default:
    statements;
    break;
}
```

Every time a case falls through (doesn't include a `break` statement), add a comment where the `break` statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every `switch` statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

try-catch Statements

A `try-catch` statement should have the following format:

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

WHITE SPACE

Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

1. Between sections of a source file
2. Between class and interface definitions

One blank line should always be used in the following circumstances:

1. Between methods
2. Between the local variables in a method and its first statement
3. Before a block (see section 5.1.1) or single-line (see section 5.1.2) comment
4. Between logical sections inside a method to improve readability

Blank Spaces

Blank spaces should be used in the following circumstances:

1. A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true) {
    ...
}
```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

2. A blank space should appear after commas in argument lists.
3. All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

4. The expressions in a for statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

5. Casts should be followed by a blank space. Examples:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3))
                              + 1);
```

NAMING CONVENTIONS

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples
Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	class Raster; class ImageSprite;
Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate; interface Storing;

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	run();
		runFast();
		getBackground();

Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.	int i; char c; float myWidth;

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int GET_THE_CPU = 1;

Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese

	organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.	

Number of Identifiers: 7

PROGRAMMING PRACTICES

Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten-often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a struct instead of a class (if Java supported struct), then it's appropriate to make the class's instance variables public.

Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead. For example:

```
classMethod();              //OK
AClass.classMethod();       //OK
anObject.classMethod();     //AVOID!
```

Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

```
if (c++ = d++) {        // AVOID! (Java disallows)
    ...
}
```

should be written as

```
if ((c++ = d++) != 0) {
    ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

```
d = (a = b + c) + r;        // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

Miscellaneous Practices

Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d)     // AVOID!
if ((a == b) && (c == d)) // RIGHT
```

Returning Values

Try to make the structure of your program match the intent. Example:

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

should instead be written as

```
return booleanExpression;
```

Similarly,

```
if (condition) {
    return x;
}
return y;
should be written as
return (condition ? x : y);
```

Expressions before `?' in the Conditional Operator

If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example:

```
(x >= 0) ? x : -x;
```

Special Comments

Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken.

CODE EXAMPLES

Java Source File Example

The following example shows how to format a Java source file containing a single public class. Interfaces are formatted similarly. For more information, see "Class and Interface Declarations" on page 21 and "Documentation Comments" on page 26

```
/*
 * @(#)Blah.java        1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */

package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
```

```
 *
 * @version  1.82 18 Mar 1999
 * @author   Firstname Lastname
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */

    /** classVar1 documentation comment */
    public static int classVar1;

    /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;

    /** instanceVar1 documentation comment */
    public Object instanceVar1;

    /** instanceVar2 documentation comment */
    protected int instanceVar2;

    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;

    /**
     * ...constructor Blah documentation comment...
     */
    public Blah() {
        // ...implementation goes here...
    }

    /**
     * ...method doSomething documentation comment...
     */
    public void doSomething() {
        // ...implementation goes here...
    }

    /**
     * ...method doSomethingElse documentation comment...
     * @param someParam description
     */
    public void doSomethingElse(Object someParam) {
        // ...implementation goes here...
    }
}
```

JAVA CODE CONVENTIONS: COPYRIGHT NOTICE

If you modify the document content in a way that alters its meaning, for example, to conform with your own company's coding conventions, please use this attribution line:

Either way, please include a hypertext link or other reference to the Java Code Conventions Web site at: http://java.sun.com/docs/codeconv/