

Indice

Introduzione all'Ingegneria del Software

Cos'è il software?

Cos'è l'Ingegneria del Software?

Qual è la differenza tra l'ingegneria del Software e l'Informatica?

Qual è la differenza tral'Ingegneria del Software e Ingegneria dei Sistemi?

Cos'è un processo software?

Cos'è un modello di processo software?

Quali sono i costi dell'Ingegneria del Software?

Distribuzione dei costi delle attività

Cosa sono i metodi dell'Ingegneria del Software?

Descrizione dei modelli

Regole

Raccomandazioni

Orientamenti sui processi

Cos'è Computer-Aided Software Engineering (CASE)?

Quali sono gli attributi di un buon software?

Quali sono le sfide principali dell'Ingegneria del Software?

Responsabilità professionale ed etica

Questioni di responsabilità professionale

Codice etico ACM/IEEE

Dilemmi etici

Processi software

Il processo software

Modello di processo software

Modelli generici di processo software

Modello a cascata

Fasi del modello a cascata

Problemi del modello a cascata

Sviluppo evolutivo

Problemi dello sviluppo evolutivo

Applicabilità dello sviluppo evolutivo

Ingegneria del software basata su componenti

Sviluppo orientato al riutilizzo

Iterazione del processo

Consegna incrementale

Vantaggi dello sviluppo incrementale

Sviluppo a spirale

Settori del modello a spirale

Attività del processo

Specifiche del software

Il processo di ingegneria dei requisiti
Progettazione e implementazione del software
Attività del processo di progettazione
Il processo di progettazione del software
Metodi strutturati
Programmazione e debug
Validazione del software
Fasi di testing
Evoluzione del software
Ingegneria del software assistita dal computer
 Tecnologia CASE
 Integrazione CASE
 Strumenti, workbench, ambienti

Requisiti Software

Ingegneria dei Requisiti
Cosa sono i Requisiti?
Astrazione dei Requisiti (Davis)
Tipi di Requisiti
Definizioni e specificazioni
 Definizione dei requisiti dell'utente
 Specifiche dei requisiti di sistema
Requisiti lettori
Requisiti funzionali e non funzionali
Esempi di requisiti funzionali
 Requisiti Funzionali del Sistema LIBSYS
 Problemi di imprecisione dei requisiti
 Completezza e coerenza dei requisiti
Requisiti non funzionali
 Classificazione dei requisiti non funzionali
 Tipi di requisiti non funzionali
 Esempi di requisiti non funzionali
Obiettivi e requisiti
Misure dei requisiti
Integrazione dei requisiti
Requisiti di dominio
Sistema di Protezione Ferroviaria
Problemi dei requisiti di dominio
 Comprensibilità
 Implicito
Requisiti e progettazione
Requisiti utente
Problemi con il linguaggio naturale
Linee guida per la scrittura dei requisiti
 Inventare un formato standard e usarlo per tutti i requisiti

Utilizzare il linguaggio in modo coerente

Usare il testo evidenziato per identificare le parti chiave dei requisiti

Evitare l'uso del gergo informatico

Requisiti di sistema

Specifiche di linguaggio strutturato

Specifiche basate su moduli

Specifiche del nodo basata su moduli

Specifiche tabulari

Specifiche tabulare

Modelli grafici

Diagramma di sequenza del prelievo ATM

Processi di ingegneria dei requisiti

Elicitazione dei requisiti

Analisi dei requisiti

Gestione dei requisiti

Validazione e verifica dei requisiti

Il processo di ingegneria dei requisiti

Studi di fattibilità

Implementazione degli studi di fattibilità

Elicitazione ed analisi

Problemi dell'analisi dei requisiti

Attività del processo di analisi dei requisiti

La spirale dei requisiti

Punti di vista

Tipi di punti di vista

Colloqui

Fattori sociali e organizzativi

Ambito dell'etnografia

Scenari

Casi d'uso

Modellazione del sistema

Tipi di modelli

Modelli comportamentali

Diagrammi di flusso di dati

Elaborazione degli ordini DFD

Modelli a macchina a stati

Modelli di dati semantici

Dizionari dei dati

Voci del dizionario dei dati

Modelli di oggetti

Modelli di oggetti e l'UML

Metodi strutturati

Debolezze dei metodi

Convalida dei requisiti

Verifica dei requisiti

Tecniche di convalida dei requisiti
Controlli delle revisioni
Gestione dei requisiti e la loro incompletezza
Tracciabilità
 Matrice di tracciabilità
Supporto agli strumenti CASE (Computer-Aided Software Engineering)
 Il documento dei requisiti
Lo standard dei requisiti IEEE
Struttura del documento dei requisiti
Metodi formali
 Uso dei metodi formali
 Costi di sviluppo con specificazione formale
 Tecniche di specifica

Design

Progettazione e realizzazione del software
 Progettazione del software
 Implementazione
 Attività del processo di progettazione
Il processo di progettazione del software
Architettura software
 Progettazione architetturale
Strutturazione del sistema
Diagrammi a blocchi e linee
 Sistema di controllo del robot di imballaggio
Vantaggi di un'architettura esplicita
Architettura e caratteristiche del sistema
Decisioni di progettazione architetturale
Modelli architetturali
Organizzazione del sistema
 Modello del repository
 Vantaggi del modello repository
 Svantaggi del modello repository
 Modello Client-Server
 Caratteristiche del modello client-server
 Vantaggi del modello client-server
 Svantaggi del modello client-server
 Modello di macchina astratta o a strati
Sistema di gestione delle versioni
Sotto-sistemi e moduli
 Sotto-sistemi
 Moduli
 Decomposizione modulare
 Stili di decomposizione modulare
Modello ad oggetti

Vantaggi del modello ad oggetti
Svantaggi del modello ad oggetti
Pipelining orientato alle funzioni
 Sistema di elaborazione delle fatture
 Caratteristiche del modello a pipeline
Stili di controllo
 Controllo centralizzato
 Sistemi Basati su Eventi
 Modelli di diffusione
 Trasmissione selettiva
 Modelli basati su interruzioni
 Controllo guidato dagli interrupt

Architetture dei sistemi distribuiti
Tipi di sistema
Sistemi distribuiti
 Vantaggi dei sistemi distribuiti
 Svantaggi dei sistemi distribuiti
 Architetture dei sistemi distribuiti
 Architetture multiprocessore
Architetture client-server
Architettura dell'applicazione a strati
 Strati dell'applicazione
Modelli di client leggero e client pesante
 Modello client leggero
 Modello client pesante
Architettura a tre livelli
Architettura degli oggetti distribuiti
 Vantaggi dell'architettura degli oggetti distribuiti
CORBA
Architetture peer-to-peer
 Architettura peer-to-peer decentralizzata
 Architettura peer-to-peer semi-centralizzata
Architetture orientate ai servizi
Standard dei servizi

Progettazione orientata agli oggetti
Sviluppo orientato agli oggetti
Analisi, progettazione, e programmazione orientate agli oggetti
 Caratteristiche dell'OOD
 Vantaggi dell'OOD
Linguaggio di modellazione unificato (UML - Unified Model Language)
Processo di progettazione orientato agli oggetti
 Fasi del processo
Contesto del sistema e modello di utilizzo

Contesto del sistema
Modello di utilizzo
Architettura a strati
 Sottosistemi in un sistema di mappatura meteorologico
 Casi d'uso per una stazione meteorologica
 Descrizione del caso d'uso
 Progettazione architetturale
 Architettura stazione meteorologica
Identificazione degli oggetti
 Approcci all'identificazione
Modelli di progettazione
 Modelli statici
 Modelli dinamici
 Esempi di modelli di progettazione
 Modelli di sequenza
 Modelli di macchine a stati
 Altri modelli
Modelli di sottosistemi
 Sottosistemi di una stazione metereologica
Modelli di sequenza
 Sequenza di raccolta dei dati
Diagrammi di stato
 Diagramma di stato di una stazione metereologica
Specifiche delle interfacce degli oggetti
 Interfaccia di una stazione meterologica

Sviluppo e mantenimento
Sviluppo del software
 Panorama dello sviluppo software
 Sviluppo software rapido (RAD)
 Requisiti
 Caratteristiche dei processi RAD
 Un processo di sviluppo iterativo
 Processo di sviluppo incrementale
 Vantaggi dello sviluppo incrementale
 Problemi dello sviluppo incrementale
 Sviluppo incrementale e prototipazione
 Obiettivi conflittuali
 Prototipazione del software
 Prototipi usa e getta
Ambienti RAD
 Un ambiente RAD
 Riutilizzo COTS
 Metodi agili
 Principi dei metodi agili

Problemi con i metodi agili
-Extreme programming
Pratiche di extreme programming
Il ciclo di rilascio dell'XP
Scenari di requisiti
Story card per il download dei documenti
Testing in XP
Schede di attività per il download dei documenti
Descrizione del caso di test
Programmazione in coppia

Riutilizzo del software

Ingegneria basata sul riuso
Il panorama del riuso
Approcci di riutilizzo
Riutilizzo dei concetti
Design patterns
Tipi di generatori di programmi
Framework applicativi
Riutilizzo del sistema applicativo
Riutilizzo dei prodotti COTS
Problema di integrazione del sistema COTS
Linee di prodotti software
Sistemi ERP
Sviluppo basati su componenti
CBSE essenziale
CBSE e principi di progettazione
Componenti
Interfacce dei componenti
Componenti e oggetti
Modelli di componenti
Il processo CBSE

Evoluzione del software

Modifica del software
Modello a spirale dell'evoluzione
Dinamiche di evoluzione del programma
Leggi di Lehman
Manutenzione del software
Distribuzione degli sforzi di manutenzione
Fattori dei costi di manutenzione
Processi di evoluzione
Il processo di evoluzione del sistema
Reingegnerizzazione del sistema
Vantaggi della reingegnerizzazione

Inoltro e reingegnerizzazione
Attività del processo di reingegnerizzazione
Approcci di reingegnerizzazione
Evoluzione del sistema legacy
Categorie del sistema legacy

Verifica e validazione
-**Verifica vs validazione**
Il processo V&V
Obiettivi del V&V
Fiducia nella V&V
Verifica statica e dinamica
Test del programma
 Tipi di test
Test e debug
Il piano di test del software
Ispezioni del software
 Successo dell'ispezione
 Ispezioni e test
 Ispezioni del programma
 Presupposti per l'ispezione
 Procedura di ispezione
 Utilizzo dell'analisi statica automatica
 Verifica e metodi formali
 Sviluppo software per camere bianche

Maggiori dettagli sul test del software

Il processo di test
Testare gli obiettivi del processo
Il processo di test del software
Testare le politiche
Test del sistema
Test d'integrazione
Test di rilascio
Test della scatola nera
Linee guida per i test
Casi d'uso
Test delle prestazioni
Prove di stress
Test dei componenti
Test delle classi di oggetti
Test dell'interfaccia
 Errori dell'interfaccia
 Linee guida per i test dell'interfaccia
Progettazione del caso di prova

Test basati sui requisiti
Partizionamento per equivalenze
Linee guida per i test (sequenze)
Test del percorso
Percorsi indipendenti
Testare l'automazione

Gestione della configurazione

Famiglie di sistema
Pianificazione della gestione della configurazione
Il piano CM
Identificazione dell'elemento di configurazione
Gerarchia di configurazione
Gestione dei cambiamenti
Revisione dei cambiamenti
Registro delle modifiche
Informazioni sull'intestazione del componente
Versioni/varianti/rilasci
Identificazione della versione
Gestione dei rilasci
Rilasci di sistema
Problemi di rilascio
Strategia di rilascio del sistema
Costruzione del sistema
Strumenti CASE per la gestione della configurazione
Strumenti CASE
 Strumenti di gestione del cambiamento
 Strumenti di gestione delle versioni

Introduzione all'Ingegneria del Software

Cos'è il software?

Il software è costituito da programmi informatici e dalla documentazione associata, che può includere requisiti, modelli di progettazione e manuali utente. I prodotti software possono essere suddivisi in due categorie principali:

- **Generici:** sviluppati per essere venduti a una vasta gamma di clienti diversi. Un esempio di software generico è Microsoft Excel o Microsoft Word.
- **Personalizzati (su misura):** sviluppati specificamente per un singolo cliente, basati sulle loro esigenze e specifiche.

La creazione di nuovo software può avvenire attraverso vari metodi, tra cui lo sviluppo di nuovi programmi, la configurazione di sistemi software generici e il riutilizzo di software esistente.

Cos'è l'Ingegneria del Software?

L'Ingegneria del Software è una disciplina ingegneristica che si occupa di tutti gli aspetti della produzione del software. Gli ingegneri del software devono seguire un approccio sistematico e organizzato al loro lavoro e utilizzare strumenti e tecniche appropriate in base al problema da risolvere, ai vincoli di sviluppo e alle risorse disponibili.

Qual è la differenza tra l'ingegneria del Software e l'Informatica?

- **Informatica:** si concentra sulla teoria e sui fondamenti dell'informatica.
- **Ingegneria del Software:** si focalizza sugli aspetti pratici dello sviluppo e della fornitura di software utile ed economico.

Qual è la differenza tra l'Ingegneria del Software e Ingegneria dei Sistemi?

- L'**Ingegneria dei Sistemi** si occupa di tutti gli aspetti dello sviluppo di sistemi basati su computer, compresi hardware, software ed ingegneria dei processi.
- L'**Ingegneria del Software** fa parte dell'Ingegneria dei Sistemi e riguarda l'infrastruttura software, il controllo, le applicazioni e i database all'interno del sistema.
- Gli ingegneri dei sistemi sono coinvolti nella specifica del sistema, nella progettazione architettonica, nell'integrazione e nella distribuzione.

Cos'è un processo software?

Un processo software è un insieme di attività il cui obiettivo è lo sviluppo o l'evoluzione del software. Le attività generiche comuni a tutti i processi software includono:

- **Specifico**: definire cosa il sistema deve fare e stabilire i vincoli di sviluppo.
- **Sviluppo**: creare il sistema software.
- **Validazione**: verificare che il software soddisfi le aspettative del cliente.
- **Evoluzione**: apportare modifiche al software in risposta alle esigenze che cambiano.

Cos'è un modello di processo software?

Un modello di processo software è una rappresentazione semplificata di un processo software da una specifica prospettiva.

Ci sono diverse prospettive sul processo, sono:

- **Prospettiva del flusso di lavoro** (sequenza di attività)
- **Prospettiva del flusso di dati** (flusso di informazioni)
- **Prospettiva dei ruoli/azioni** (chi fa cosa)

Esempi di modelli di processo generici sono:

- **Modello a cascata**
- **Sviluppo iterativo**
- **Ingegneria del Software basata su componenti**

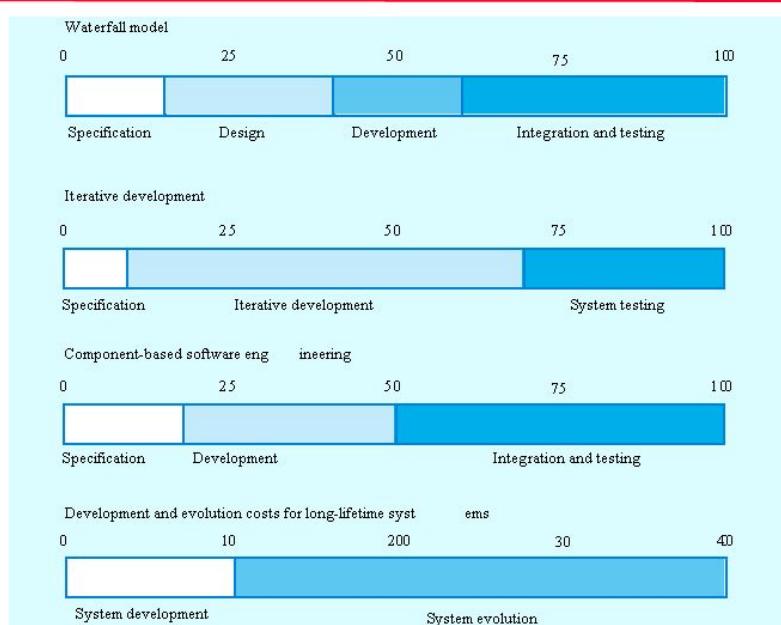
Quali sono i costi dell'Ingegneria del Software?

I costi dell'Ingegneria del Software sono approssimativamente distribuiti nel seguente modo:

- Circa il 60% dei costi sono attribuibili al processo di sviluppo.
- Il restante 40% riguarda il testing e la validazione del software.
- Per il software personalizzato, i costi di evoluzione spesso superano i costi di sviluppo iniziale.
- I costi variano a seconda del tipo di sistema in sviluppo, dalle prestazioni richieste e dalla sua affidabilità.
- La distribuzione dei costi può variare notevolmente a seconda del modello di sviluppo utilizzato.

Distribuzione dei costi delle attività

Activity cost distribution



15

Cosa sono i metodi dell'Ingegneria del Software?

I metodi nell'Ingegneria del Software rappresentano approcci strutturati e disciplinati per lo sviluppo del software. Questi metodi includono descrizioni dei modelli, regole, raccomandazioni e orientamenti sui processi, tutti finalizzati a guidare il processo di sviluppo software in modo efficiente e organizzato.

Descrizione dei modelli

La descrizione dei modelli fornisce una rappresentazione visiva e concettuale del sistema software, aiutando gli sviluppatori a comprendere l'architettura, le componenti e le relazioni del software.

Regole

Le regole impongono vincoli sui modelli dei sistemi, stabilendo limiti e requisiti necessari per il software, come ad esempio requisiti di sicurezza o prestazioni.

Raccomandazioni

Le raccomandazioni offrono consigli sulle migliori pratiche per il processo di sviluppo, coprendo aspetti come la progettazione, il testing e la gestione dei requisiti.

Orientamenti sui processi

Gli orientamenti sui processi definiscono le attività e le fasi da seguire durante il ciclo di vita del software, garantendo che il processo di sviluppo sia ben definito e seguito in modo coerente.

Cos'è Computer-Aided Software Engineering (CASE)?

Il CASE (Computer-Aided Software Engineering) è un insieme di sistemi software progettati per fornire supporto automatizzato alle attività dei processi software. Questi sistemi sono spesso utilizzati per supportare l'applicazione di metodi di sviluppo software.

I **CASE superiori** sono strumenti che sostengono le prime fasi del processo, come la definizione dei requisiti e la progettazione, mentre i **CASE inferiori** sono orientati alle fasi successive, come la programmazione e il testing.

Quali sono gli attributi di un buon software?

Gli attributi di un buon software includono la capacità di fornire funzionalità e prestazioni richieste, oltre alla manutenibilità, l'affidabilità, l'efficienza e l'accettabilità.

- La **manutenibilità** implica che il software può essere modificato per adattarsi alle esigenze in evoluzione.
- L'**affidabilità** richiede che il software sia robusto e senza errori.
- L'**efficienza** significa che il software non deve sprecare risorse del sistema.
- L'**accettabilità** implica che il software deve essere comprensibile, utilizzabile e compatibile con altri sistemi.

Quali sono le sfide principali dell'Ingegneria del Software?

- **Eterogeneità:** sviluppare tecniche per costruire software che possano funzionare su piattaforme e ambienti di esecuzione diversi.
- **Consegna rapida:** sviluppare tecniche per accelerare il processo di sviluppo e consegna del software.
- **Affidabilità:** sviluppare tecniche per dimostrare che il software è affidabile e sicuro per gli utenti.

Responsabilità professionale ed etica

Nel campo dell'Ingegneria del Software, esiste una responsabilità professionale ed etica. Gli ingegneri devono comportarsi in modo onesto ed eticamente responsabile. Il comportamento etico va oltre il semplice rispetto della legge.

Questioni di responsabilità professionale

- **Riservatezza:** Gli ingegneri dovrebbero rispettare la riservatezza delle informazioni dei loro datori di lavoro o clienti, anche senza un accordo formale di riservatezza.
- **Competenza:** Gli ingegneri non dovrebbero rappresentare in modo scorretto il loro livello di competenza e non dovrebbero accettare incarichi al di fuori delle loro competenze.
- **Diritti di proprietà intellettuale:** Gli ingegneri devono rispettare le leggi locali in materia di proprietà intellettuale, come brevetti e copyright, e garantire la protezione della proprietà intellettuale dei datori di lavoro e dei clienti.
- **Abuso del computer:** Gli ingegneri non dovrebbero utilizzare le proprie competenze tecniche per abusare dei computer altrui.

Codice etico ACM/IEEE

L'ACM (Association for Computing Machinery) e l'IEEE (Institute of Electrical and Electronics Engineers) hanno sviluppato un Codice etico che contiene otto Principi legati al comportamento e alle decisioni dei professionisti e studenti del settore dell'informatica e dell'Ingegneria del Software.

Dilemmi etici

Inoltre, alcune situazioni etiche possono presentare dilemmi, come il disaccordo con le politiche aziendali, l'abbandono di un sistema critico per la sicurezza senza il completamento del testing, o la partecipazione allo sviluppo di sistemi di armi militari o sistemi nucleari.

Processi software

Il processo software

Il processo software rappresenta un insieme strutturato di attività necessarie per sviluppare un sistema software.

Queste attività comprendono:

- **Specific**
- **Progettazione**
- **Validazione**
- **Evoluzione del software**

Modello di processo software

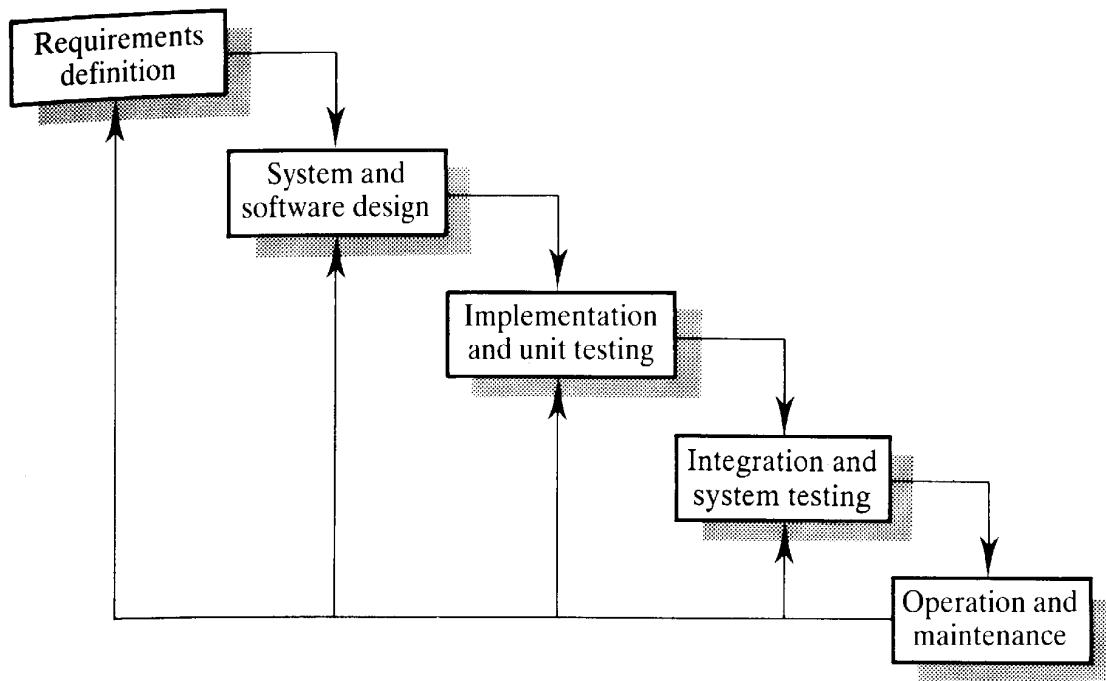
Un modello di processo software è una rappresentazione astratta di come avviene un processo di sviluppo software. Questo modello offre una descrizione dettagliata di come un sistema software viene creato da una prospettiva specifica. Un modello di processo può variare da un'organizzazione all'altra e può essere personalizzato per adattarsi alle esigenze specifiche del progetto in corso.

Modelli generici di processo software

- **Il modello a cascata:** caratterizzato da fasi separate e distinte di specifica e sviluppo.
- **Sviluppo evolutivo:** specifica, sviluppo e validazione sono intercalati.
- **Ingegneria del software basata su componenti:** il sistema è assemblato da componenti esistenti o sistemi COTS (Commercial-off-the-shelf).

Esistono numerose varianti di questi modelli, tra cui lo sviluppo formale in cui viene utilizzato un processo simile a quello a cascata, ma viene utilizzata una specifica formale affinata attraverso diverse fasi per una progettazione implementabile.

Modello a cascata



Fasi del modello a cascata

- Analisi e definizione dei requisiti
- Progettazione del sistema e del software
- Implementazione e test unitari
- Integrazione e test di sistema
- Operazione e manutenzione

Lo svantaggio principale di questo modello è la difficoltà di adattarsi ai cambiamenti una volta avviato il processo, in quanto una fase deve essere completata prima di passare alla fase successiva.

Problemi del modello a cascata

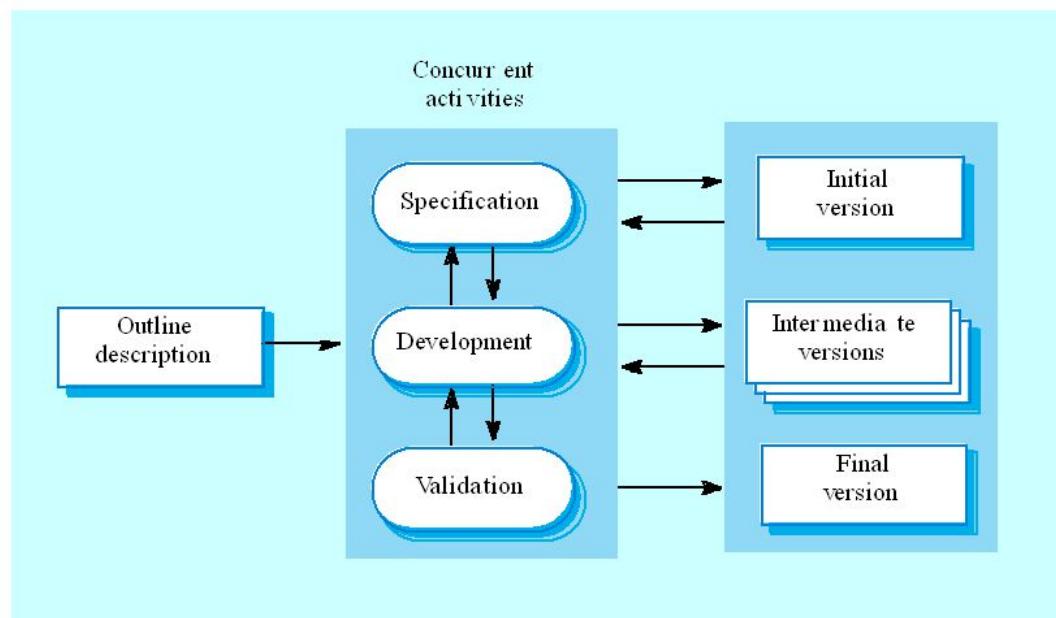
- La suddivisione inflessibile del progetto in fasi distinte rende difficile rispondere ai cambiamenti nei requisiti del cliente.
- Questo modello è appropriato quando i requisiti sono ben compresi e i cambiamenti saranno limitati durante il processo di progettazione.
- Tuttavia, pochi sistemi aziendali hanno requisiti stabili, e il modello a cascata è principalmente utilizzato per progetti di ingegneria dei sistemi di grandi dimensioni che coinvolgono sviluppo in diversi siti.

Sviluppo evolutivo

Suddiviso in due approcci:

- **Sviluppo esplorativo:** l'obiettivo è lavorare con i clienti ed evolvere un sistema finale da una specifica iniziale. Si inizia con requisiti ben compresi e si aggiungono nuove funzionalità proposte dal cliente.
- **Prototipazione usa e getta:** l'obiettivo è comprendere i requisiti del sistema e dovrebbe iniziare con requisiti poco compresi per chiarire ciò che è veramente necessario.

Evolutionary development



Problemi dello sviluppo evolutivo

Problemi associati a questo modello includono la mancanza di visibilità del processo, strutture insoddisfacenti nei sistemi e la necessità di abilità speciali, ad esempio, in linguaggi per la prototipazione rapida.

Applicabilità dello sviluppo evolutivo

Questo modello è applicabile a sistemi interattivi di piccole o medie dimensioni, a parti di sistemi di grandi dimensioni come l'interfaccia utente e a sistemi a vita breve.

Ingegneria del software basata su componenti

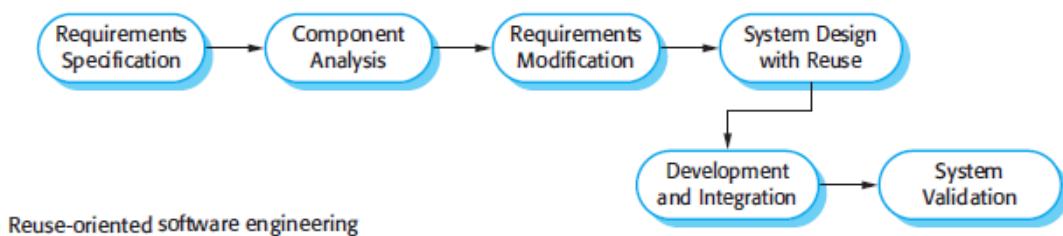
Basata sul riuso sistematico, in cui i sistemi sono integrati da componenti esistenti o sistemi COTS (Commercial-off-the-shelf).

Le fasi del processo includono:

- Analisi delle componenti
- Modifica dei requisiti
- Progettazione del sistema con riuso
- Sviluppo e integrazione.

Questo approccio è in aumento grazie all'emergere di standard delle componenti.

Sviluppo orientato al riutilizzo



Iterazione del processo

I requisiti del sistema evolvono costantemente durante un progetto.

L'iterazione può essere applicata a qualsiasi dei modelli di processo generici, e ci sono due approcci correlati:

- **Consegna incrementale**
- **Sviluppo a spirale**

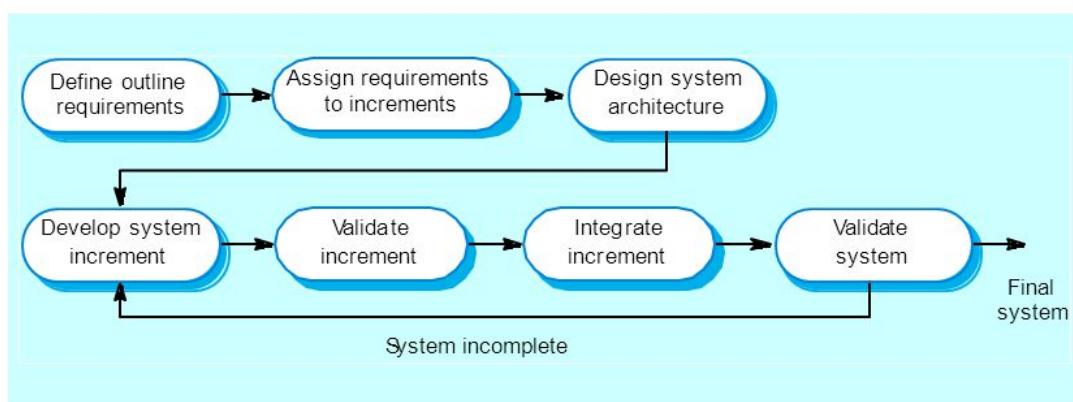
Consegna incrementale

Il processo di sviluppo e consegna è suddiviso in incrementi, ciascuno dei quali fornisce una parte della funzionalità richiesta.

I requisiti degli utenti sono prioritari, e i requisiti di massima priorità sono inclusi nei primi incrementi.

Una volta avviato lo sviluppo di un incremento, i requisiti sono congelati, anche se i requisiti per incrementi successivi possono continuare a evolvere.

Incremental development



Modified from Sommerville's originals Software Engineering, 7th edition, Chapter 4

Slide 17

Vantaggi dello sviluppo incrementale

- Fornisce valore al cliente con ciascun incremento, consentendo di avere funzionalità del sistema disponibili in anticipo.
 - Gli incrementi iniziali fungono da prototipo per aiutare a elencare i requisiti per gli incrementi successivi.
 - Riduce il rischio complessivo del progetto.
 - Gli aspetti di sistema di massima priorità ricevono più test.

Sviluppo a spirale

Rappresenta il processo come una spirale con fasi che rappresentano obiettivi specifici. Le fasi non sono fisse, ma vengono scelte in base a ciò che è richiesto, con valutazioni e riduzioni dei rischi esplicite durante il processo.

Settori del modello a spirale

- **Definizione degli obiettivi:** vengono identificati obiettivi specifici per la fase.
- **Valutazione e riduzione dei rischi:** i rischi vengono valutati e vengono messe in atto attività per ridurre i rischi chiave
- **Sviluppo e convalida:** viene scelto un modello di sviluppo per il sistema, che può essere uno qualsiasi dei modelli generici
- **Pianificazione:** il progetto viene esaminato, e viene pianificata la fase successiva della spirale

Attività del processo

Le attività principali del processo software includono:

- **Specifiche del software**
- **Progettazione e implementazione del software**
- **Validazione del software**
- **Evoluzione del software**

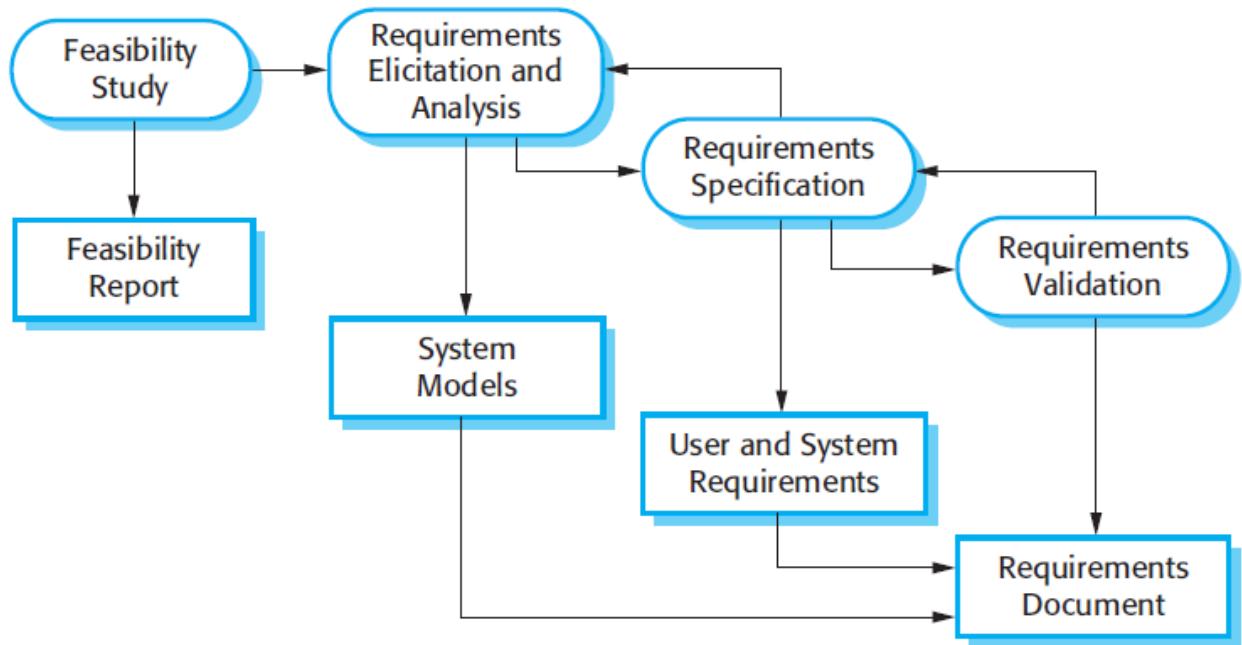
Specifiche del software

Questo processo implica stabilire quali servizi sono richiesti e i vincoli sul funzionamento e lo sviluppo del sistema.

Le attività di ingegneria dei requisiti includono:

- **Studio di fattibilità**
- **Elicitazione e analisi dei requisiti**
- **Specifiche dei requisiti**
- **Validazione dei requisiti**

Il processo di ingegneria dei requisiti



Progettazione e implementazione del software

Questo processo riguarda la conversione della specifica del sistema in un sistema eseguibile.

Include:

- La **progettazione del software**, che mira a creare una struttura che realizzi la specifica
- L'**implementazione**, che traduce questa struttura in un programma eseguibile.

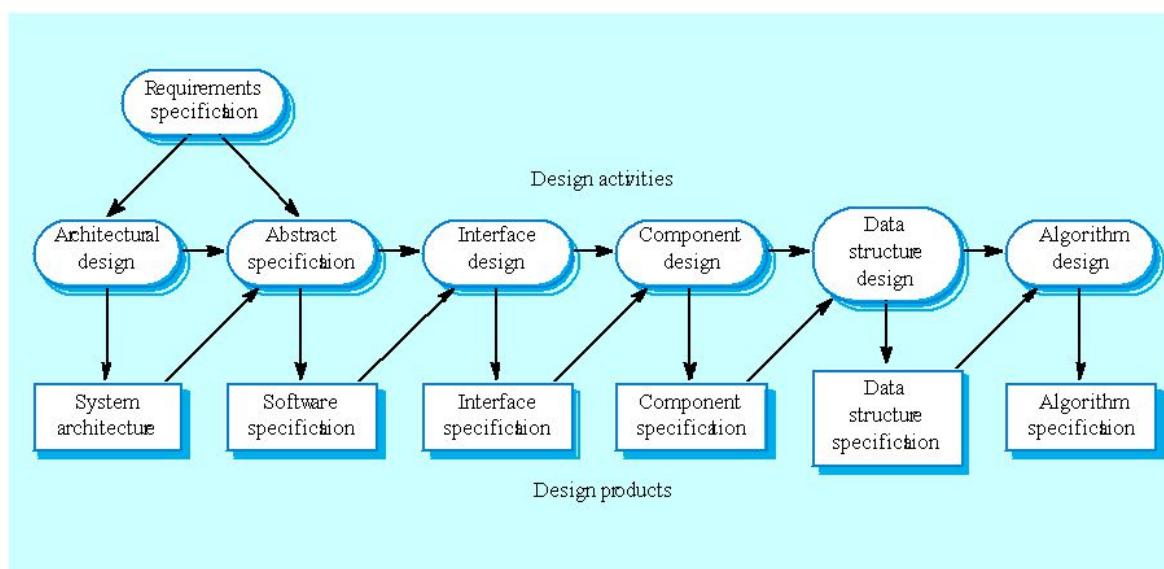
Le attività di progettazione e implementazione sono strettamente correlate e possono essere intercalate.

Attività del processo di progettazione

- **Progettazione architetturale**
- **Specificazione astratta**
- **Progettazione delle interfacce**
- **Progettazione dei componenti**
- **Progettazione delle strutture dati**
- **Progettazione degli algoritmi**

Il processo di progettazione del software

The software design process



Modified from Sommerville's originals

Software Engineering, 7th edition. Chapter 11

Slide 4

Metodi strutturati

Si riferiscono a un approccio sistematico allo sviluppo di progettazioni software, solitamente documentate come un insieme di modelli grafici.

Modelli possibili includono:

- **Modello di oggetti**
- **Modello di sequenza**
- **Modello di transizione di stato**
- **Modello strutturale**
- **Modello di flusso dei dati**

Programmazione e debug

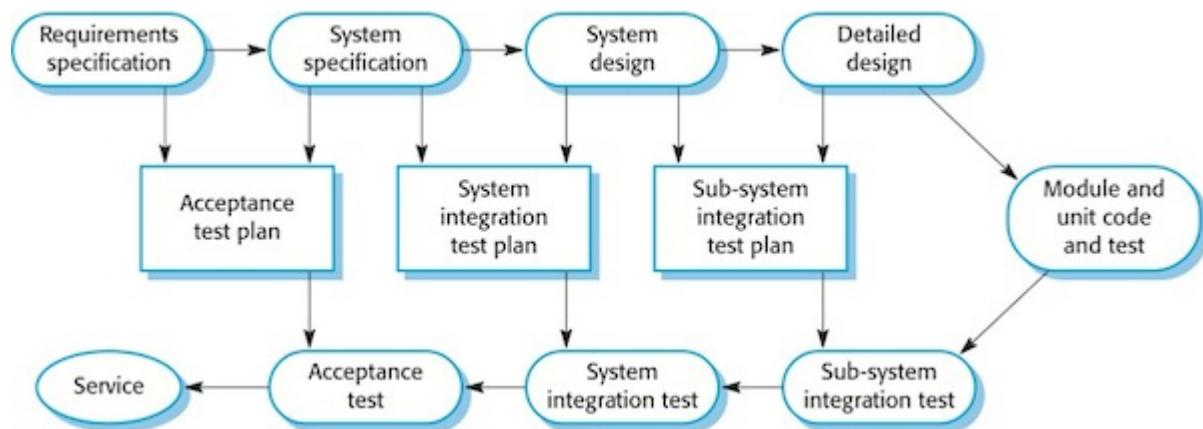
Queste attività coinvolgono la traduzione di una progettazione in un programma e la rimozione degli errori dal programma. La programmazione è un'attività personale (non esiste un processo di programmazione generico), e i programmatore eseguono test per scoprire i difetti e li rimuovono durante il debug.

Validazione del software

La verifica e la convalida (V & V) mirano a dimostrare che un sistema è conforme alla sua specifica e soddisfa i requisiti del cliente. Involge processi di verifica e revisione e test di sistema. Il test di sistema comporta l'esecuzione del sistema con casi di test derivati dalla specifica dei dati reali da elaborare da parte del sistema.

Fasi di testing

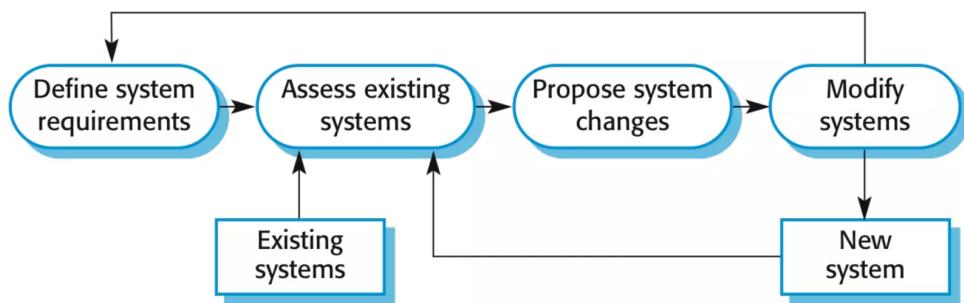
- **Testing dei componenti o unità:** singoli componenti vengono testati separatamente.
 - **Testing di sistema:** il sistema nel suo complesso viene testato, concentrandosi sulle proprietà emergenti.
 - **Testing di accettazione:** il sistema è testato con dati del cliente per verificare la soddisfazione dei requisiti.



Evoluzione del software

Il software è intrinsecamente flessibile e può cambiare, specialmente quando i requisiti cambiano. Anche se c'è stata una demarcazione tra sviluppo ed evoluzione (manutenzione), questo è sempre più irrilevante poiché sempre meno sistemi sono completamente nuovi.

System evolution



Ingegneria del software assistita dal computer

L'ingegneria del software assistita dal computer (**Computer aided software engineering** o CASE) supporta i processi di sviluppo e evoluzione del software, con automazione di attività e strumenti.

Esempi di automazione delle attività:

- Editor grafici per lo sviluppo del modello di sistema
- Dizionario dei dati per gestire le entità di progettazione
- Costruttore grafico dell'interfaccia utente per la costruzione dell'interfaccia utente
- Debugger per supportare la ricerca dei difetti del programma
- Traduttori automatici per generare nuove versioni di un programma.

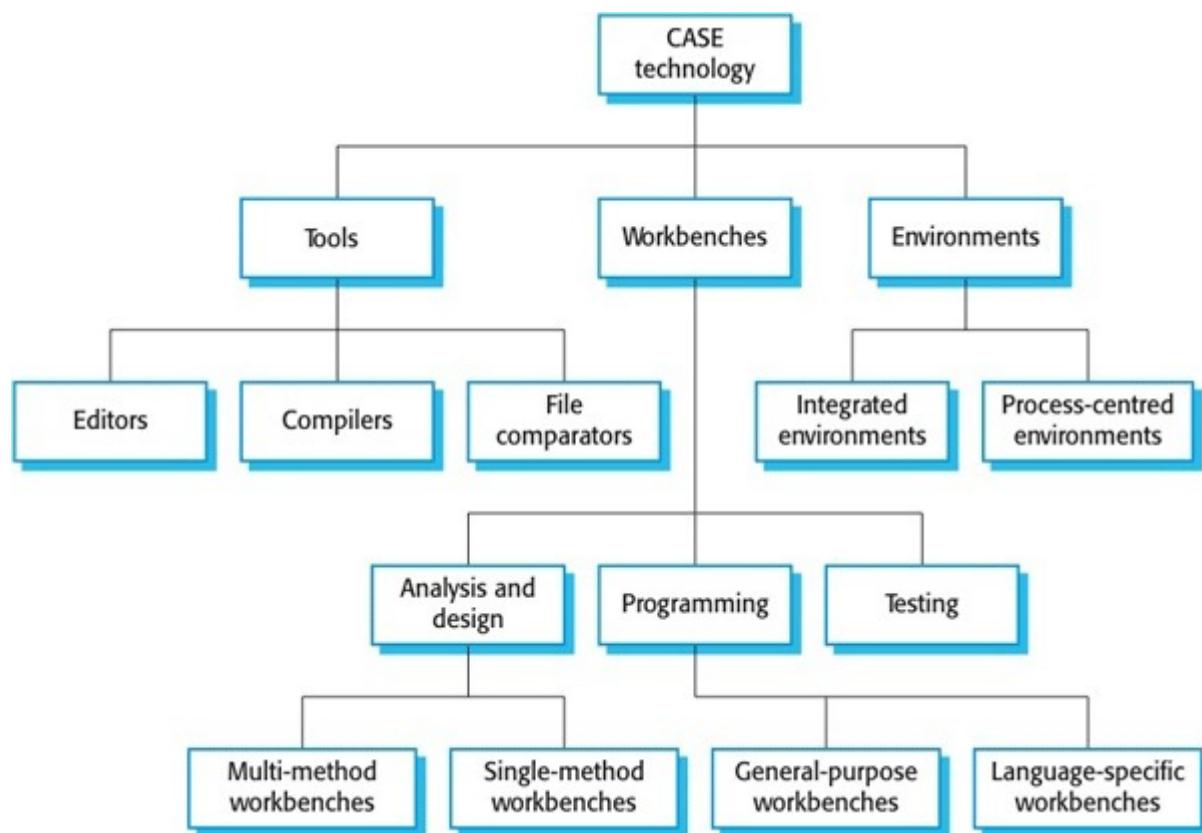
Tecnologia CASE

Nonostante i miglioramenti significativi nel processo software grazie alla tecnologia CASE, l'ingegneria del software richiede pensiero creativo che non può essere completamente automatizzato. Inoltre, gran parte delle attività di ingegneria del software coinvolge interazioni di squadra, che non sono pienamente supportate dalla tecnologia CASE.

Integrazione CASE

- Gli strumenti CASE supportano attività specifiche del processo.
- I workbench includono un insieme di strumenti integrati.
- Gli ambienti supportano tutto o parte del processo software, compresi diversi workbench integrati.

Strumenti, workbench, ambienti



Requisiti Software

Ingegneria dei Requisiti

L'ingegneria dei requisiti è il processo di stabilire i servizi richiesti dal cliente per un sistema e i vincoli che lo definiscono e lo guidano durante lo sviluppo. Questi requisiti rappresentano le descrizioni dei servizi e dei vincoli del sistema e vengono generati nel corso del processo di ingegneria dei requisiti.

Cosa sono i Requisiti?

I requisiti possono essere considerati come affermazioni riguardanti i servizi o i vincoli di un sistema. Queste affermazioni possono essere espresse in modo astratto o specifico, ad esempio mediante specifiche funzionali matematiche. Questa versatilità è importante, in quanto i requisiti possono svolgere una doppia funzione: possono essere la base per un'offerta per un contratto o costituire il contratto stesso tra le parti coinvolte.

Astrazione dei Requisiti (Davis)

"Se un'azienda desidera assegnare un contratto per un grande progetto di sviluppo software, deve definire le proprie esigenze in modo sufficientemente astratto da non predefinire una soluzione. I requisiti devono essere scritti in modo che diversi appaltatori possano fare un'offerta per il contratto, offrendo, ad esempio, modi diversi per soddisfare le esigenze dell'organizzazione cliente. Una volta assegnato un contratto, l'appaltatore deve scrivere una definizione di sistema per il cliente in modo più dettagliato in modo che il cliente capisca e possa convalidare ciò che farà il software. Entrambi questi documenti possono essere chiamati documento dei requisiti del sistema."

Tipi di Requisiti

Esistono due tipi principali di requisiti:

- **Requisiti utente:** sono affermazioni che descrivono i servizi che il sistema deve fornire e i vincoli operativi ad esso associati. Questi requisiti sono scritti in linguaggio naturale e in forma di diagrammi e sono destinati a essere comprensibili per i clienti.
- **Requisiti di sistema:** costituiscono una specifica più dettagliata delle funzioni, dei servizi e dei vincoli del sistema. Sono utilizzati come base per la progettazione del sistema e possono far parte di un contratto tra il cliente e l'appaltatore.

Definizioni e specificazioni

Definizione dei requisiti dell'utente

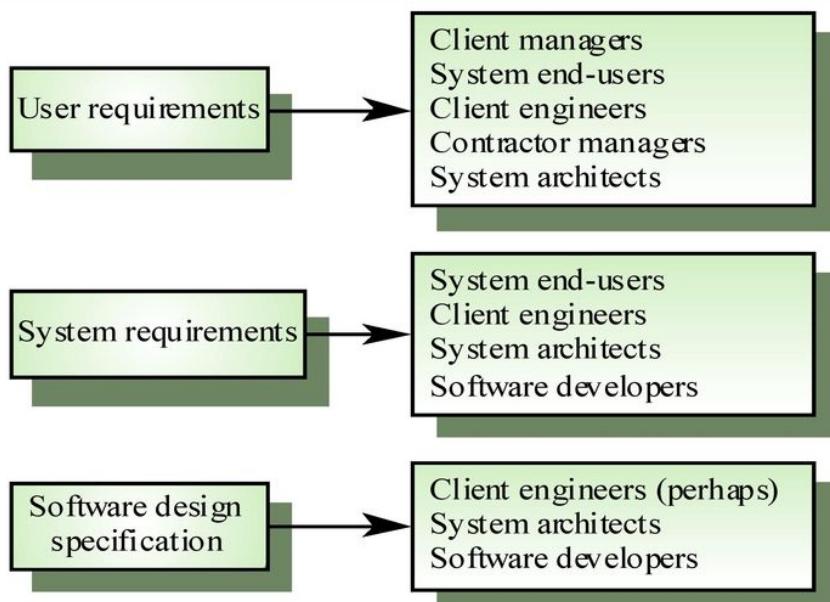
1. Il software deve fornire un mezzo per rappresentare e accedere a file esterni creati da altri strumenti.

Specifica dei requisiti di sistema

1. All'utente dovrebbero essere fornite funzionalità per definire il tipo di file esterni.
2. Ciascun tipo di file esterno può avere uno strumento associato che può essere applicato al file.
3. Ogni tipo di file esterno può essere rappresentato come un'icona specifica su display dell'utente.
4. Dovrebbero essere previste strutture per l'icona che rappresenta un tipo di file esterno definito dall'utente.
5. Quando un utente seleziona un'icona che rappresenta un file esterno, l'effetto di tale selezione è quello di applicare lo strumento associato al tipo di file esterno al file rappresentato dall'icona selezionata.

Requisiti lettori

Requirements readers



Requisiti funzionali e non funzionali

I **requisiti funzionali** descrivono la funzionalità o i servizi che il sistema deve fornire. Indicano cosa il sistema dovrebbe fare, come deve rispondere a input specifici e come dovrebbe comportarsi in situazioni particolari.

I **requisiti non funzionali** stabiliscono i vincoli sui servizi o le funzioni del sistema. Questi possono includere requisiti di tempo, vincoli sul processo di sviluppo, standard di settore e così via.

I **requisiti di dominio** possono derivare dal dominio di applicazione del sistema e rifletterne le caratteristiche.

Esempi di requisiti funzionali

Requisiti Funzionali del Sistema LIBSYS

Il sistema LIBSYS deve soddisfare diversi requisiti funzionali al fine di offrire una corretta ed efficiente esperienza all'utente:

1. **Ricerca Estesa dei Documenti:** Il sistema dovrà consentire all'utente di effettuare ricerche all'interno del vasto database. Questa funzionalità include la possibilità di cercare in tutto il set iniziale di database o di selezionare un sottoinsieme specifico per una ricerca mirata.
2. **Visualizzazione dei Documenti:** Il sistema fornirà un visualizzatore integrato per garantire che l'utente possa leggere agevolmente i documenti archiviati all'interno del sistema. Ciò garantirà una visualizzazione ottimale e la fruibilità dei documenti.
3. **Assegnazione di Identificatori Unici per gli Ordini:** Ad ogni ordine effettuato tramite il sistema, sarà automaticamente assegnato un identificatore univoco, noto come "ID ORDINE." Questo identificatore sarà reso disponibile all'utente, che potrà copiarlo nell'area di archiviazione permanente del proprio account. Questa caratteristica agevola il tracciamento e la gestione degli ordini.

Questi requisiti funzionali sono essenziali per il funzionamento del sistema LIBSYS, garantendo un'esperienza utente efficiente e consentendo agli utenti di sfruttare appieno le risorse e i servizi forniti dal sistema.

Problemi di imprecisione dei requisiti

La mancanza di precisione nei requisiti può causare problemi significativi. I requisiti ambigui possono essere interpretati in modi diversi da sviluppatori e utenti, portando a fraintendimenti.

Ad esempio, l'espressione "visualizzatori appropriati" potrebbe essere interpretata in modi diversi.

È essenziale minimizzare l'ambiguità e garantire requisiti chiari e inequivocabili.

Completezza e coerenza dei requisiti

I requisiti dovrebbero essere:

- **Completi:** Dovrebbero includere descrizioni di tutte le funzionalità richieste dal sistema senza lasciare spazio a omissioni.
- **Coerenti:** Non dovrebbero esserci conflitti o contraddizioni tra le descrizioni delle funzionalità.

Tuttavia, nella pratica, è difficile produrre un documento di requisiti che sia contemporaneamente completo e privo di conflitti.

Requisiti non funzionali

I requisiti non funzionali definiscono proprietà e vincoli del sistema al di fuori delle sue funzionalità specifiche. Questi possono includere aspetti come l'affidabilità, il tempo di risposta, i requisiti di archiviazione.

I requisiti di processo possono essere specificati imponendo un particolare sistema CASE, linguaggio di programmazione o metodo di sviluppo.

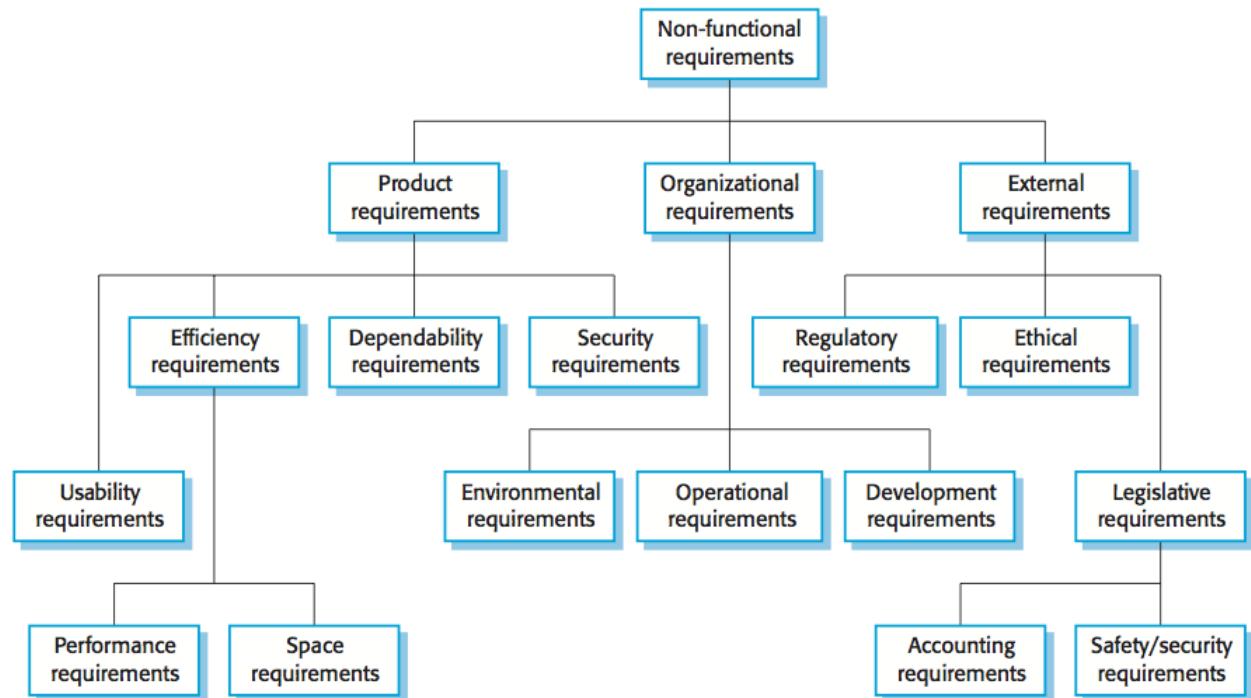
I requisiti non funzionali possono essere più critici dei requisiti funzionali stessi.

Classificazione dei requisiti non funzionali

I requisiti non funzionali possono essere suddivisi in tre categorie principali:

- **Requisiti del prodotto:** riguardano come il sistema deve comportarsi.
- **Requisiti organizzativi:** sono influenzati dalle politiche e dalle procedure dell'organizzazione
- **Requisiti esterni:** derivano da fattori esterni al sistema e al processo di sviluppo.

Tipi di requisiti non funzionali



Esempi di requisiti non funzionali

Alcuni esempi di requisiti non funzionali includono specifiche sulle prestazioni dell'interfaccia utente, conformità alle procedure organizzative, limiti di privacy dei dati e molto altro.

- **Requisito del Prodotto:** L'interfaccia utente per LIBSYS deve essere implementata come semplice HTML senza frame o applet Java.
- **Requisito Organizzativo:** Il processo di sviluppo del sistema e i documenti consegnabili devono conformarsi al processo e ai documenti consegnabili definiti in XYZCo-SP-STAN-95.
- **Requisito Esterno:** Il sistema non deve divulgare informazioni personali sui clienti oltre al loro nome e numero di riferimento agli operatori del sistema

Obiettivi e Requisiti

Gli obiettivi rappresentano le intenzioni generali degli utenti rispetto a ciò che desiderano ottenere da un sistema, come facilità d'uso o efficienza. Questi obiettivi forniscono una visione globale delle aspettative degli utenti nei confronti del sistema.

In contrasto, i requisiti sono dichiarazioni specifiche che delineano dettagliatamente cosa il sistema deve fare e come deve farlo. Nei requisiti, è cruciale definire in modo preciso le funzionalità e le prestazioni del sistema in modo verificabile, ovvero misurabile in modo oggettivo attraverso test o altre metodologie di verifica.

I requisiti non funzionali, che possono riguardare aspetti come la sicurezza o l'usabilità, sono spesso difficili da esprimere in modo preciso, ma è essenziale formulare requisiti non funzionali in modo chiaro e verificabile per garantire la corretta realizzazione del sistema.

Gli obiettivi sono utili per comunicare le aspettative degli utenti, ma devono essere tradotti in requisiti specifici e misurabili per guidare il processo di sviluppo del sistema in modo coerente. La traduzione degli obiettivi in requisiti verificabili aiuta a definire criteri oggettivi per valutare il successo e la conformità del sistema rispetto alle aspettative degli utenti.

Misure dei Requisiti

I requisiti possono essere misurati in termini di diverse proprietà. Alcuni esempi includono:

- Velocità: Transazioni elaborate al secondo, Tempo di risposta utente/evento, Tempo di aggiornamento dello schermo
- Dimensione: Megabyte, Numero di chip ROM
- Facilità d'uso: Tempo di formazione, Numero di riquadri di aiuto
- Affidabilità: Tempo medio tra guasti, Probabilità di non disponibilità, Tasso di insorgenza dei guasti, Disponibilità
- Robustezza: Tempo di riavvio dopo un guasto, percentuale di eventi che causano un guasto, Probabilità di corruzione dei dati in caso di guasto
- Portabilità: Percentuale di dichiarazioni dipendenti dal bersaglio, Numero di sistemi di destinazione

Integrazione dei requisiti

L'interazione dei requisiti è un aspetto comune nell'ingegneria dei requisiti, soprattutto in sistemi complessi. Un esempio che illustra questo concetto può essere considerato in relazione a un sistema spaziale.

Nel contesto di un sistema spaziale, possono emergere conflitti tra diversi requisiti non funzionali. Due di questi requisiti riguardano la minimizzazione del peso e del consumo energetico del sistema. Un requisito specifico potrebbe essere il seguente: per minimizzare il peso, è necessario ridurre il numero di chip separati nel sistema. Allo stesso tempo, per ridurre il consumo energetico, sarebbe vantaggioso utilizzare chip a basso consumo energetico.

Tuttavia, c'è un'interazione tra questi due requisiti. L'uso di chip a basso consumo energetico, sebbene sia benefico per il risparmio energetico, potrebbe richiedere l'uso di un numero maggiore di tali chip. Questo può aumentare il peso complessivo del sistema, che è contrario al primo requisito di minimizzare il peso.

In questa situazione, sorge una domanda importante: quale dei due requisiti è più critico o prioritario? È necessario valutare attentamente questa interazione tra i requisiti e prendere decisioni informate sulla loro implementazione, tenendo conto di vari fattori, tra cui le esigenze del progetto e le restrizioni di design.

L'identificazione, la gestione e la risoluzione dei conflitti tra i requisiti sono aspetti cruciali dell'ingegneria dei requisiti, soprattutto in contesti in cui i sistemi devono soddisfare diverse esigenze e vincoli, come nel caso dei sistemi spaziali.

Requisiti di dominio

I requisiti di dominio, derivano dal dominio di applicazione specifico in cui il sistema verrà utilizzato. Questi requisiti svolgono un ruolo fondamentale nel delineare le caratteristiche e le funzionalità necessarie affinché il sistema possa riflettere in modo accurato le esigenze e le dinamiche del dominio in cui opererà.

Possono includere nuovi requisiti funzionali che si rendono necessari per adattarsi al contesto di applicazione. Inoltre, i requisiti di dominio possono comportare la specifica di vincoli o condizioni che i requisiti esistenti devono rispettare in questo particolare contesto.

La conformità ai requisiti di dominio è di vitale importanza poiché, se questi requisiti non vengono adeguatamente soddisfatti, il sistema rischia di diventare inutilizzabile. Pertanto, è essenziale che gli sviluppatori e gli ingegneri del sistema comprendano in modo completo e preciso i requisiti di dominio al fine di garantire che il sistema finale si integri perfettamente e funzioni in modo appropriato nel contesto in cui verrà utilizzato.

Sistema di protezione ferroviaria

La decelerazione del treno deve essere calcolata come:

- DistanzaTreno = DistanzaControllo + DistanzaPendenza

Dove $DistanzaPendenza$ è $9,81 \text{ m/s}^2 * \text{pendenza compensata}/\alpha$ e dove i valori di $9,81 \text{ m/s}^2$ / α sono noti per diversi tipi di treno

Problemi dei requisiti di dominio

Comprensibilità

Un problema comune nell'ambito dell'analisi dei requisiti è la comprensibilità. Spesso, i requisiti vengono espressi utilizzando il linguaggio del dominio di applicazione, il che può causare difficoltà di comprensione da parte degli ingegneri del software che sviluppano il sistema. Questo divario tra il linguaggio dei requisiti e la comprensione dell'ingegneria del software può portare a fraintendimenti o errori nell'interpretazione dei requisiti stessi.

Implicito

Un'altra sfida legata ai requisiti di dominio è l'implicito. Gli specialisti del dominio, essendo esperti nel loro campo, potrebbero dare per scontato che certi aspetti siano chiari e ovvi, senza rendere esplicativi i requisiti associati. Questo può comportare ambiguità o fraintendimenti nei requisiti, poiché ciò che è implicito per gli specialisti potrebbe non esserlo per gli altri membri del team di sviluppo o gli stakeholder del progetto. Pertanto, è importante sforzarsi di rendere esplicativi tutti i requisiti necessari per evitare confusione e garantire una comprensione comune tra tutte le parti coinvolte nel processo di sviluppo.

Requisiti e progettazione

In teoria, i requisiti dovrebbero stabilire cosa il sistema dovrebbe fare, mentre il design dovrebbe descrivere come il sistema lo fa. Tuttavia, nella pratica, requisiti e progettazione sono spesso inseparabili.

Nel processo di sviluppo di un sistema, l'architettura del sistema può essere progettata in modo specifico per soddisfare i requisiti. Questo significa che il design dell'architettura stessa può essere guidato dai requisiti stabiliti.

Inoltre, il sistema può interagire con altri sistemi, e questi interazioni possono generare nuovi requisiti di progettazione. Ad esempio, se un sistema deve comunicare con un altro sistema tramite un protocollo specifico, il design del sistema deve incorporare questo protocollo come parte della progettazione.

Infine, in alcuni casi, l'uso di una progettazione specifica può essere un requisito di dominio. Ad esempio, un sistema di controllo industriale potrebbe richiedere l'uso di hardware o software specifici come parte dei requisiti di dominio.

Quindi, sebbene requisiti e progettazione siano distinti concettualmente, nella pratica sono strettamente collegati e spesso influenzano l'uno con l'altro durante il processo di sviluppo del sistema.

Requisiti Utente

I requisiti utente devono essere chiari e comprensibili per chiunque utilizzi il sistema, anche se non hanno una conoscenza tecnica dettagliata.

Per garantire questa chiarezza, vengono definiti principalmente attraverso il linguaggio naturale, tabelle e diagrammi. Questi approcci favoriscono una comunicazione efficace tra sviluppatori e utenti, garantendo una visione condivisa delle funzionalità e delle prestazioni del sistema.

Problemi con il linguaggio naturale

I requisiti scritti in linguaggio naturale possono presentare diverse sfide:

- **Mancanza di Chiarezza:** Spesso, la chiarezza è difficile da ottenere nei requisiti scritti in linguaggio naturale. Questo perché cercare di essere estremamente precisi può rendere il documento poco leggibile e comprensibile. Trovare un equilibrio tra la precisione e la leggibilità è una sfida.
- **Confusione dei Requisiti:** I requisiti funzionali e non funzionali spesso si mescolano nei documenti. Questa confusione può complicare il processo di analisi e di sviluppo del sistema. È importante mantenere una chiara distinzione tra questi due tipi di requisiti.
- **Unificazione dei Requisiti:** In alcuni casi, diversi requisiti possono essere espressi insieme in una stessa affermazione. Questo può portare a complicazioni nella gestione e nella tracciabilità dei requisiti. È importante suddividere e specificare chiaramente i requisiti in modo separato.

- **Ambiguità:** L'ambiguità è un problema comune nei requisiti scritti in linguaggio naturale. È essenziale che i lettori e gli scrittori dei requisiti condividano una comprensione comune delle parole utilizzate nei documenti. Tuttavia, il linguaggio naturale è intrinsecamente ambiguo, rendendo difficile evitare completamente questo problema.
- **Eccessiva Flessibilità:** Nel linguaggio naturale, la stessa idea o concetto può essere espressa in modi diversi. Questa flessibilità può portare a interpretazioni varie dei requisiti. La standardizzazione del linguaggio può contribuire a mitigare questo problema.
- **Mancanza di Modularizzazione:** Spesso, le strutture del linguaggio naturale non sono adeguate per suddividere e strutturare i requisiti del sistema in modo logico. La mancanza di modularizzazione può rendere difficile l'organizzazione e la gestione dei requisiti. L'uso di approcci strutturati o modulari può contribuire a risolvere questa problematica.

Questi problemi sottolineano la necessità di applicare buone pratiche e standard nella scrittura dei requisiti, al fine di renderli chiari, coerenti e facilmente comprensibili da tutte le parti coinvolte nello sviluppo di un sistema.

Linee guida per la scrittura dei requisiti

Inventare un formato standard e usarlo per tutti i requisiti

Una pratica fondamentale nella scrittura dei requisiti è quella di stabilire un formato standard e aderirvi in modo coerente. Questo formato standard dovrebbe essere applicato a tutti i requisiti del sistema. Questa uniformità nella presentazione rende i requisiti più facili da leggere e comprendere per tutti gli stakeholder coinvolti nel progetto.

Utilizzare il linguaggio in modo coerente

La coerenza nell'uso del linguaggio è cruciale per evitare ambiguità e frantendimenti nei requisiti. È importante utilizzare un linguaggio chiaro e coerente che rispecchi con precisione il significato desiderato. A questo scopo, è consigliabile adottare alcune regole chiave:

- Utilizzare "deve" per i requisiti obbligatori: Il termine "deve" dovrebbe essere riservato per descrivere i requisiti che sono assolutamente necessari per il funzionamento del sistema. Questi sono gli elementi non negoziabili che devono essere implementati.
- Usare "dovrebbe" per i requisiti desiderabili: Il termine "dovrebbe" può essere utilizzato per requisiti che sono auspicabili, ma non essenziali. Questi requisiti rappresentano funzionalità o caratteristiche che migliorerebbero il sistema ma potrebbero essere negoziabili o implementati successivamente.

Usare il testo evidenziato per identificare le parti chiave dei requisiti

Per rendere i requisiti più chiari e immediati, è utile utilizzare il testo evidenziato o la formattazione speciale per identificare le parti chiave di ciascun requisito. Questo può includere termini chiave, parametri cruciali o elementi di importanza particolare. L'evidenziazione contribuisce a catturare l'attenzione degli stakeholder e semplifica la comprensione dei dettagli rilevanti.

Evitare l'uso del gergo informatico

L'uso del gergo informatico o terminologia tecnica complessa nei requisiti può rendere difficile la comprensione, in particolare per gli stakeholder non tecnici. È consigliabile evitare il linguaggio e i termini troppo tecnici, a meno che siano necessari e inevitabili. Nel caso in cui sia richiesto l'uso di terminologia specifica, è utile fornire spiegazioni o definizioni per garantire la chiarezza.

Requisiti di sistema

I requisiti di sistema forniscono specifiche più dettagliate delle funzioni, dei servizi e dei vincoli del sistema rispetto ai requisiti utente.

Questi requisiti costituiscono una base essenziale per la progettazione del sistema. Inoltre, possono essere inclusi in modo formale nel contratto del sistema, stabilendo in modo vincolante cosa ci si aspetta che il sistema consegna.

È importante notare che i requisiti di sistema possono essere rappresentati o illustrati mediante l'uso di modelli di sistema, che consentono una visione strutturata e coerente delle funzionalità e delle restrizioni del sistema in sviluppo.

Specifiche di linguaggio strutturato

Le specifiche di linguaggio strutturato impongono una struttura predefinita per le specifiche dei requisiti, limitando la libertà dello scrittore dei requisiti. Tutti i requisiti sono redatti in modo standardizzato, e la terminologia utilizzata nella descrizione può essere limitata. Questo approccio ha vantaggi e svantaggi.

Vantaggi:

- Mantiene la maggior parte dell'espressività nella specifica.

Svantaggi:

- Impone un grado di uniformità sulla specifica.

Specifiche basate su moduli

Le specifiche basate su moduli seguono una struttura formale che include:

- Definizione della funzione o dell'entità.
- Descrizione degli input e della loro origine.
- Descrizione degli output e della loro destinazione.
- Indicazione di altre entità necessarie.
- Condizioni pre e post (se appropriate).
- Gli effetti collaterali (se presenti) della funzione.

Form-based node specification

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: Safe sugar level

Description Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r2), the previous two readings (r0 and r1)

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose § the dose in insulin to be delivered

Destination Main control loop

Action: CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

Requires Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition The insulin reservoir contains at least the maximum allowed single dose of insulin..

Post-condition r0 is replaced by r1 then r1 is replaced by r2

Side-effects None

Specifiche tabulari

Le specifiche tabulari vengono utilizzate per integrare il linguaggio naturale e sono particolarmente utili quando è necessario definire una serie di possibili percorsi alternativi.

- Vengono utilizzate per completare il linguaggio naturale.
- Sono particolarmente utili quando è necessario definire una serie di possibili percorsi alternativi.

Specifiche tabulari

Tabular specification

| Condition | Action |
|---|--|
| Sugar level falling ($r_2 < r_1$) | CompDose = 0 |
| Sugar level stable ($r_2 = r_1$) | CompDose = 0 |
| Sugar level increasing and rate of increase decreasing ($((r_2-r_1) < (r_1-r_0))$) | CompDose = 0 |
| Sugar level increasing and rate of increase stable or increasing. ($((r_2-r_1) \square (r_1-r_0))$) | CompDose = round $((r_2-r_1)/4)$ If rounded result = 0 then CompDose = MinimumDose |

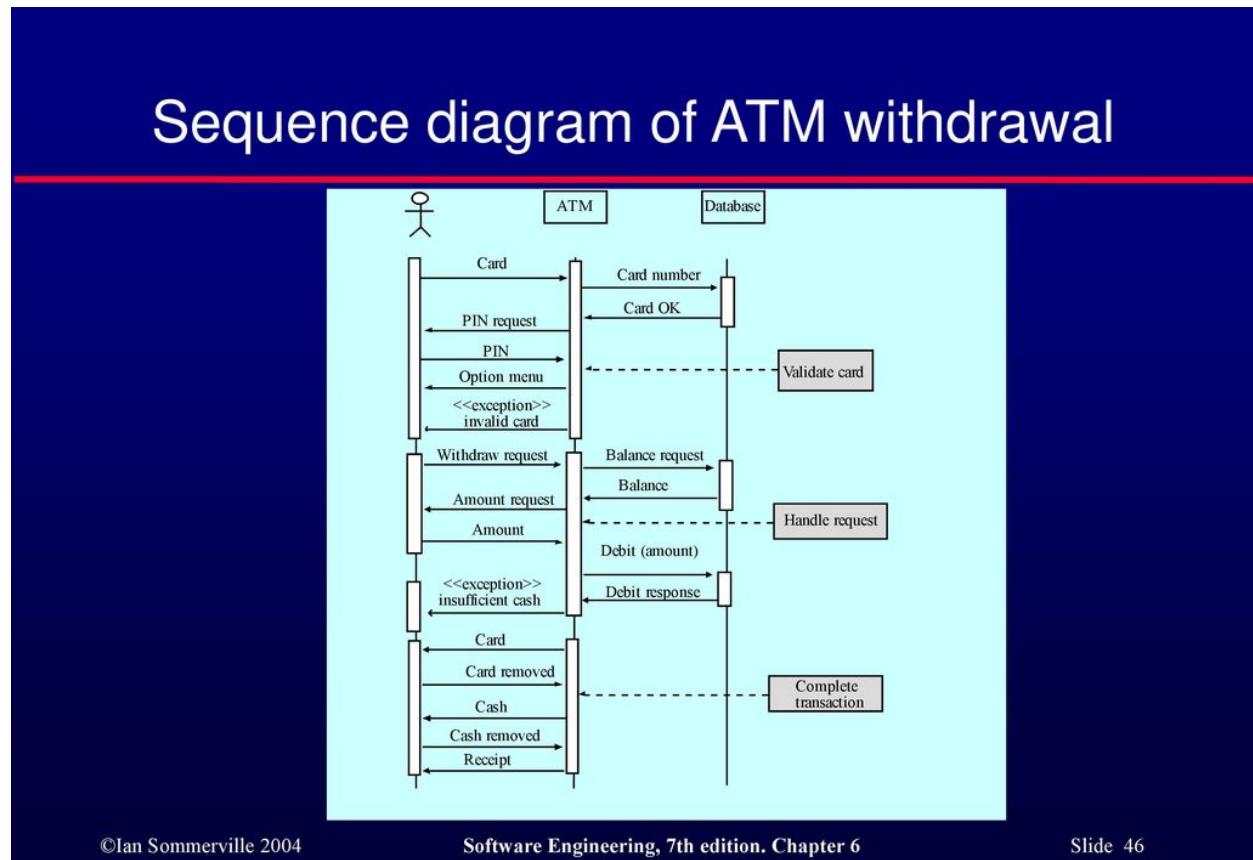
Modelli grafici

I modelli grafici mostrano come avvengono i cambiamenti di stato e descrivono una sequenza di azioni.

- Mostrano come avvengono i cambiamenti di stato.
- Descrivono una sequenza di azioni.

Questi diagrammi forniscono una rappresentazione chiara e intuitiva del flusso di dati o delle interazioni tra componenti del sistema, aiutando a comprendere il comportamento del sistema in modi visivi.

Diagramma di sequenza del prelievo ATM



Processi di ingegneria dei requisiti

I processi di ingegneria dei requisiti variano notevolmente a seconda di diversi fattori, tra cui il dominio di applicazione, le persone coinvolte e l'organizzazione responsabile dello sviluppo dei requisiti. Tuttavia, ci sono alcune attività comuni che si verificano in tutti i processi di ingegneria dei requisiti. Queste attività, se svolte in modo efficace, contribuiscono alla definizione e al successo del progetto.

Elicitazione dei Requisiti

Il processo di elicitatione dei requisiti coinvolge l'identificazione, la raccolta e la definizione dei requisiti di un sistema. Può comprendere attività come interviste, osservazioni, indagini, analisi documentale e altre tecniche per comprendere le esigenze degli stakeholder. L'obiettivo è catturare i requisiti in modo completo e accurato.

Analisi dei Requisiti

Dopo l'elicitatione, segue l'analisi dei requisiti, durante la quale i requisiti vengono esaminati, organizzati e documentati in modo coerente. Questa fase implica la rimozione di requisiti ambigui o contraddittori e la creazione di documenti di requisiti chiari e comprensibili.

Gestione dei requisiti

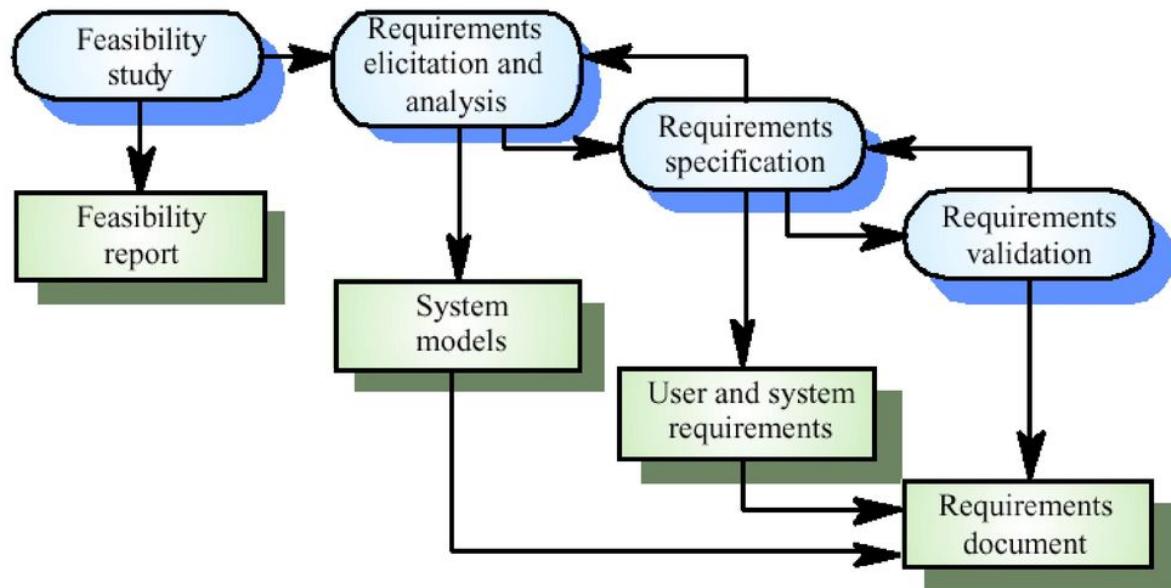
La gestione dei requisiti è un'attività continua durante tutto il ciclo di vita dello sviluppo del software. Questa fase coinvolge la tracciabilità dei requisiti, cioè il collegamento dei requisiti alle funzionalità o ai componenti corrispondenti nel sistema. Inoltre, la gestione dei requisiti gestisce le modifiche ai requisiti esistenti e la gestione delle versioni dei documenti dei requisiti.

Validazione e verifica dei requisiti

La validazione dei requisiti è il processo di conferma che i requisiti stessi siano in linea con le esigenze e le aspettative degli stakeholder. D'altra parte, la verifica dei requisiti riguarda la conferma che i requisiti siano corretti, completi e coerenti. Entrambe queste attività sono essenziali per assicurare che il sistema sia in grado di soddisfare pienamente le aspettative del cliente.

Il processo di ingegneria dei requisiti

The requirements engineering process



©Ian Sommerville 2000

Software Engineering, 6th edition. Chapter 6

Slide 6

Studi di fattibilità

Gli studi di fattibilità sono una fase cruciale nella valutazione di un sistema proposto, poiché servono a stabilire se il sistema è conveniente.

La prima valutazione in uno studio di fattibilità riguarda se il sistema contribuirà agli obiettivi dell'organizzazione. È essenziale assicurarsi che il sistema risolva i problemi o soddisfi le necessità aziendali in modo efficace ed efficiente.

Un altro aspetto importante riguarda la valutazione della fattibilità tecnologica e finanziaria. Lo studio verifica se il sistema può essere realizzato con le tecnologie attuali e senza superare il budget stabilito. Questo è fondamentale per evitare costi eccessivi o rischi tecnologici.

L'ultimo aspetto esaminato è l'integrazione del sistema con gli altri sistemi già in uso nell'organizzazione. È essenziale garantire che il nuovo sistema possa operare senza problemi insieme ai sistemi esistenti e che possa scambiare dati in modo efficace. La mancanza di compatibilità o problemi di integrazione potrebbe renderlo inadatto.

Implementazione degli studi di fattibilità

L'implementazione degli studi di fattibilità coinvolge l'analisi delle informazioni disponibili e necessarie per comprendere appieno il contesto e le esigenze del progetto. Inoltre, richiede la raccolta di dati e informazioni rilevanti, oltre alla compilazione di un rapporto dettagliato che riassume le conclusioni raggiunte. Durante questo processo, è fondamentale porre alcune domande chiave alle persone coinvolte nell'organizzazione:

- Cosa accadrebbe se il sistema in questione non fosse implementato?
- Quali sono i problemi attuali all'interno dei processi che il sistema proposto dovrebbe risolvere?
- In che modo il sistema proposto contribuirà a migliorare la situazione o raggiungere gli obiettivi dell'organizzazione?
- Quali sono i potenziali problemi di integrazione con i sistemi attualmente utilizzati?
- Sarà necessaria l'adozione di nuove tecnologie per la realizzazione del sistema? In tal caso, quali competenze e risorse saranno necessarie?
- Quali strutture e processi all'interno dell'organizzazione devono essere supportati e adattati per accogliere il sistema proposto?

Elicitazione ed analisi

L'elicitazione o la scoperta dei requisiti è un processo cruciale nell'ingegneria dei requisiti, che coinvolge uno staff tecnico collaborativo con i clienti allo scopo di comprendere appieno il dominio applicativo, i servizi che il sistema dovrebbe offrire e i vincoli operativi che devono essere rispettati.

Questo processo può coinvolgere una vasta gamma di parti interessate, ciascuna con una prospettiva unica. Tali parti interessate, comunemente chiamate **stakeholders**, includono gli utenti finali, i manager, gli ingegneri della manutenzione e gli esperti del dominio.

Problemi dell'analisi dei requisiti

L'analisi dei requisiti è una fase critica nello sviluppo di qualsiasi sistema software o prodotto.

Durante questo processo, possono sorgere vari problemi che richiedono attenzione e risoluzione:

1. **Indeterminatezza dei requisiti:** Spesso, gli stakeholder coinvolti potrebbero non avere una chiara comprensione di ciò che vogliono. Questa indeterminatezza può derivare da una mancanza di consapevolezza dei propri bisogni o da difficoltà a esprimere tali bisogni in modo chiaro.
2. **Linguaggio dei requisiti:** Gli stakeholder provengono da diverse sfere professionali e possono utilizzare il loro linguaggio tecnico per esprimere i requisiti. Questo può portare a una possibile incomprensione tra le parti coinvolte. È importante tradurre questi requisiti in un linguaggio comune e comprensibile.
3. **Conflitti tra stakeholder:** Diversi stakeholder possono avere requisiti contrastanti. Ad esempio, i reparti di marketing e sviluppo potrebbero avere visioni diverse del prodotto. La gestione di questi conflitti richiede negoziazione e una chiara definizione dei criteri di priorità.
4. **Influenza di fattori organizzativi e politici:** I requisiti possono anche essere influenzati da fattori organizzativi o politici all'interno dell'azienda. Ad esempio, un dirigente potrebbe insistere su determinati requisiti per ragioni politiche o di prestigio.
5. **Requisiti in evoluzione:** Durante il processo di analisi, i requisiti possono subire cambiamenti dovuti all'emergere di nuovi stakeholder o a modifiche nell'ambiente aziendale. Questi cambiamenti richiedono flessibilità nell'adattare i requisiti esistenti o nell'aggiungere nuovi requisiti.

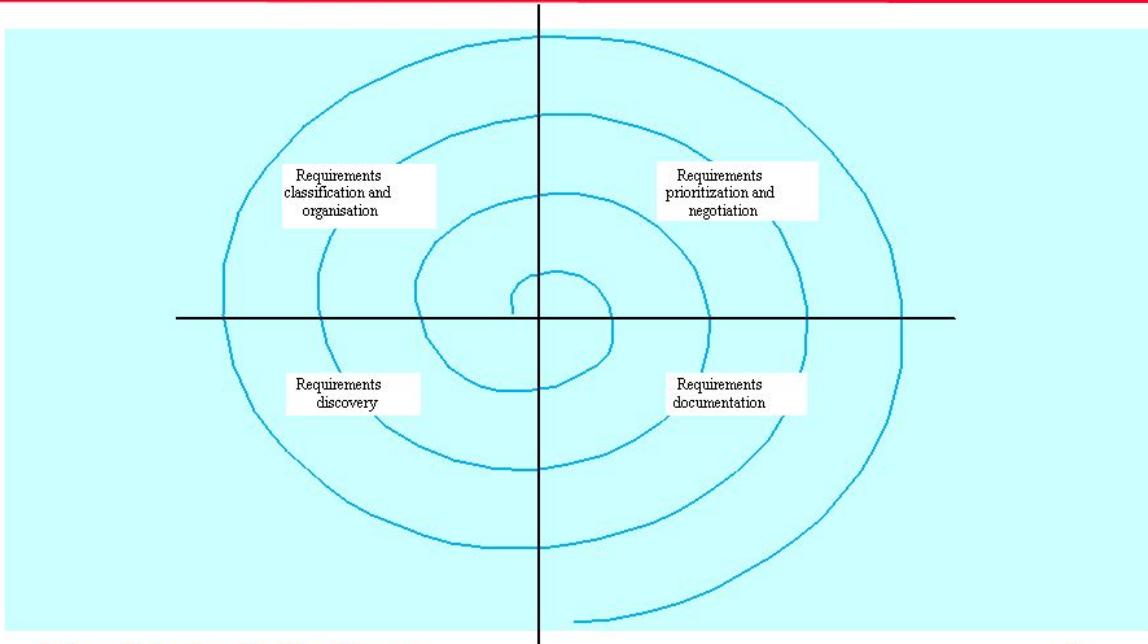
Attività del processo di analisi dei requisiti

L'analisi dei requisiti coinvolge diverse attività che svolgono un ruolo fondamentale nel raccogliere, comprendere e documentare i requisiti del sistema:

1. **Scoperta dei requisiti:** Questa fase richiede un'interazione diretta con gli stakeholder per raccogliere i loro requisiti. In questa fase, vengono scoperti sia i requisiti esplicativi che quelli impliciti, e spesso si includono i requisiti di dominio che riflettono le caratteristiche e le funzionalità del settore in cui il sistema opererà.
2. **Classificazione e organizzazione dei requisiti:** Dopo aver raccolto i requisiti, è fondamentale organizzarli in modo logico e comprensibile. Questo processo comporta il raggruppamento dei requisiti correlati in cluster coerenti per semplificare la gestione e l'analisi.
3. **Priorità e negoziazione:** Non tutti i requisiti possono essere soddisfatti immediatamente o incondizionatamente. È necessario stabilire priorità tra i requisiti e negoziare in caso di conflitti tra di essi. Questa fase comporta anche il coinvolgimento degli stakeholder per definire quali requisiti siano essenziali e quali siano opzionali.
4. **Documentazione dei requisiti:** La documentazione dei requisiti è cruciale per garantire che tutti gli stakeholder abbiano una visione chiara e condivisa di ciò che il sistema deve fare. Questa documentazione fornisce un riferimento stabile per tutto il ciclo di vita dello sviluppo del sistema e può essere utilizzata come base per la progettazione, lo sviluppo e la verifica del sistema.

La spirale dei requisiti

The requirements spiral



Software Engineering, 8th edition. Chapter 7
Courtesy: ©Ian Sommerville 2006

March 20th, 2008

9
Lecture # 12

Punti di vista

I punti di vista rappresentano un modo strutturato per analizzare e organizzare i requisiti al fine di tener conto delle diverse prospettive degli stakeholder. Questa analisi multi-prospettica è cruciale poiché non esiste un unico modo "corretto" per esaminare i requisiti di un sistema.

Tipi di punti di vista

Esistono diversi tipi di punti di vista utili nell'analisi dei requisiti:

1. **Punti di Vista degli Interattori:** Questi punti di vista considerano le persone o altri sistemi che interagiscono direttamente con il sistema in questione. Questi stakeholder hanno esigenze e aspettative specifiche che devono essere catturate nei requisiti.
2. **Punti di Vista Indiretti:** Alcuni stakeholder non utilizzano direttamente il sistema ma possono influenzarne i requisiti. Ad esempio, i regolatori o le parti interessate che definiscono normative e standard devono essere considerati.
3. **Punti di Vista di Dominio:** Questi punti di vista tengono conto delle caratteristiche e dei vincoli specifici del dominio di applicazione che influenzano i requisiti. Ad esempio, le leggi, le regole o le considerazioni etiche rientrano in questa categoria.

Colloqui

I colloqui sono uno strumento importante nella raccolta dei requisiti e possono essere condotti in modalità formale o informale. Durante questi incontri, il team di ingegneria dei requisiti pone domande agli stakeholder riguardo al sistema che utilizzano o intendono sviluppare. Esistono due tipi principali di colloqui:

1. **Colloqui Chiusi:** Questi colloqui seguono una struttura prefissata con domande specifiche. Possono essere utilizzati per ottenere informazioni dettagliate su aspetti specifici dei requisiti.
2. **Colloqui Aperti:** In questo caso, non vi è una struttura rigida, e le conversazioni sono più flessibili e adattabili alle risposte degli stakeholder. Possono rivelare informazioni inaspettate o dettagli più ampi sulla visione del sistema.

Fattori Sociali e Organizzativi

I sistemi software sono utilizzati in contesti sociali e organizzativi complessi. Questi fattori possono influenzare o addirittura dominare i requisiti del sistema. Non costituiscono un punto di vista specifico ma influenzano tutti gli altri punti di vista. Considerazioni come la cultura aziendale, le dinamiche organizzative e le aspettative degli utenti possono avere un impatto significativo sui requisiti.

Ambito dell'Etnografia

Nell'ingegneria dei requisiti, è importante riconoscere che i requisiti derivano spesso dal modo effettivo in cui le persone lavorano, anziché dal modo in cui i processi sono teoricamente definiti. L'etnografia, che coinvolge l'osservazione diretta delle attività umane, può essere utilizzata per catturare i requisiti basati sulla realtà. La cooperazione tra gli stakeholder e la comprensione delle attività di altre persone svolgono un ruolo essenziale nella definizione dei requisiti.

Scenari

Gli scenari sono rappresentazioni di situazioni reali che illustrano come un sistema può essere utilizzato nella pratica. Ogni scenario dovrebbe includere vari elementi, come:

- **Descrizione della Situazione Iniziale:** Cosa sta accadendo prima dell'utilizzo del sistema.
- **Flusso Normale degli Eventi:** Come si sviluppa l'interazione con il sistema in una situazione tipica.
- **Gestione di Eventi Anomali:** Cosa può andare storto durante l'interazione e come il sistema reagisce a queste situazioni.
- **Informazioni su Attività Simultanee:** Se ci sono altre azioni o processi che si svolgono contemporaneamente all'utilizzo del sistema.
- **Descrizione dello Stato Finale:** Cosa si ottiene o cosa accade alla fine dell'interazione.

Gli scenari offrono una prospettiva chiara su come il sistema si integra nella vita reale e possono essere utilizzati per catturare requisiti essenziali.

Casi d'uso

I casi d'uso sono una tecnica basata sugli scenari nell'UML che identifica gli attori in un'interazione e descrive l'interazione stessa. In un sistema, gli attori rappresentano i ruoli o le entità esterne che interagiscono con il sistema. Questi casi d'uso aiutano a catturare le principali funzionalità del sistema dal punto di vista degli utenti o degli attori coinvolti.

Un insieme di casi d'uso dovrebbe descrivere tutte le interazioni possibili con il sistema. Ciò significa che l'analista o il progettista del sistema deve identificare e documentare tutti i possibili modi in cui gli attori interagiscono con il sistema, compresi gli scenari principali e le eccezioni.

I diagrammi di sequenza possono essere utilizzati per aggiungere dettagli ai casi d'uso mostrando la sequenza di elaborazione degli eventi nel sistema. Questi diagrammi aiutano a visualizzare come i vari oggetti del sistema interagiscono tra loro e con gli attori esterni in risposta agli eventi.

Modellazione del sistema

La modellazione del sistema è un processo chiave nell'analisi dei requisiti. Questa pratica aiuta l'analista a comprendere la funzionalità del sistema e i modelli risultanti vengono utilizzati per comunicare con i clienti, i progettisti e gli sviluppatori. L'uso di diversi modelli consente di esaminare il sistema da diverse prospettive:

- **Prospettiva Esterna:** Questa prospettiva si concentra sugli attori esterni e su come essi interagiscono con il sistema. I casi d'uso sono uno strumento comune per rappresentare questa prospettiva.
- **Prospettiva Comportamentale:** Questa prospettiva esamina il comportamento dinamico del sistema, inclusi i flussi di lavoro, gli eventi e le risposte alle interazioni. I diagrammi di sequenza e di attività sono spesso utilizzati per modellare questa prospettiva.
- **Prospettiva Strutturale:** Questa prospettiva riguarda la struttura interna del sistema, comprese le componenti, i sottosistemi e le relazioni tra di essi. I diagrammi delle classi e gli schemi di architettura sono strumenti utilizzati per rappresentare questa prospettiva.

Tipi di modelli

Nel processo di modellazione del sistema, sono utilizzati vari tipi di modelli per esplorare e rappresentare le diverse dimensioni del sistema:

- **Modello di Elaborazione Dati:** Questo modello illustra come i dati vengono elaborati nelle diverse fasi del sistema. Mostra come i dati entrano nel sistema, subiscono elaborazioni e generano dati in uscita.
- **Modello di Composizione:** Questo tipo di modello visualizza come le entità sono composte da altre entità più piccole. Ad esempio, può mostrare come un oggetto complesso è costituito da oggetti più semplici o componenti.
- **Modello Architetturale:** Questo modello rappresenta l'architettura generale del sistema, inclusi i principali sottosistemi e le loro relazioni. Aiuta a definire la struttura generale del sistema.
- **Modello di Classificazione:** Questo tipo di modello evidenzia come le entità del sistema sono classificate in categorie o classi. È utile per comprendere le relazioni e le similitudini tra le diverse entità.
- **Modello Stimolo/Risposta:** Questo modello mette in evidenza la reazione del sistema agli eventi o agli stimoli esterni. Mostra come il sistema risponde a input specifici e quali azioni vengono scatenate.

Modelli comportamentali

I modelli comportamentali sono fondamentali per descrivere il comportamento di un sistema e ne esistono due principali tipi:

1. **Modelli di Processamento Dati:** Questi modelli mostrano come i dati fluiscono attraverso il sistema, come vengono elaborati e trasformati. Offrono una visione del percorso dei dati all'interno del sistema.
2. **Modelli State Machine:** Questi modelli mettono in evidenza come il sistema risponde agli eventi o alle interazioni con l'ambiente, rappresentando il sistema come uno stato variabile. Forniscono una comprensione del comportamento dinamico del sistema.

Entrambi i tipi di modelli forniscono prospettive diverse sul comportamento del sistema e spesso sono necessari entrambi per ottenere una rappresentazione completa del funzionamento del sistema. Questo approccio integrato è cruciale per la comprensione e la definizione dei requisiti del sistema.

Diagrammi di Flusso Dati

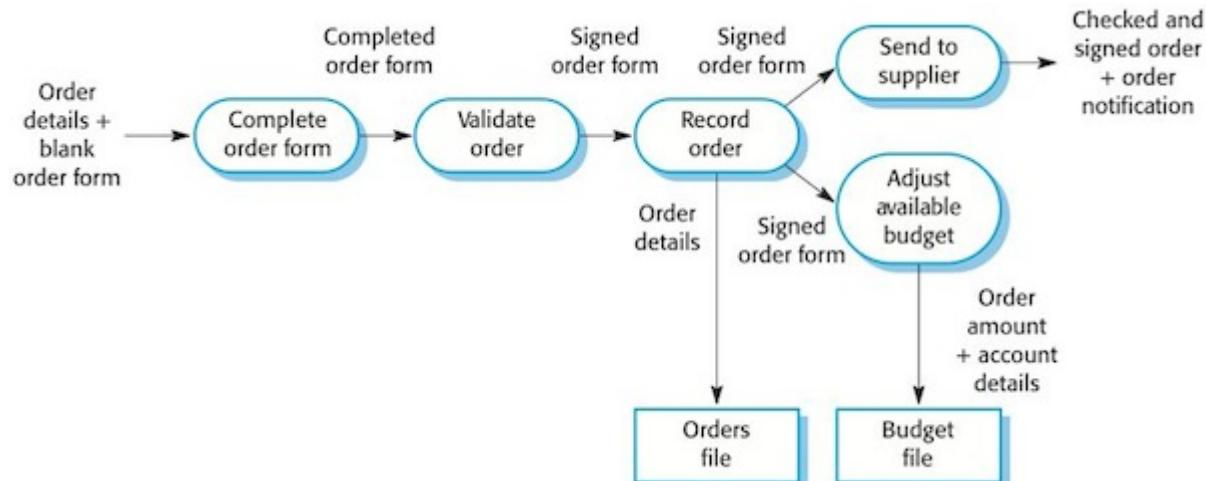
I diagrammi di flusso dati, noti come DFD, sono utilizzati per rappresentare un sistema dal punto di vista funzionale.

Sono essenziali per tracciare come i dati si spostano attraverso il sistema e vengono elaborati dai vari processi. Questa rappresentazione aiuta a comprendere il funzionamento del sistema e le interazioni tra le sue parti.

I DFD possono essere usati per mostrare come avviene lo scambio di dati tra il sistema e altri sistemi o attori nell'ambiente circostante, rendendo chiare le interfacce e le comunicazioni tra le entità coinvolte.

Sono uno strumento prezioso per l'analisi e la progettazione dei sistemi, contribuendo a migliorare la comprensione e la progettazione di sistemi efficienti.

Elaborazione degli ordini DFD



Modelli a Macchina a Stati

I modelli a macchina a stati sono utilizzati per descrivere il comportamento dinamico di un sistema in risposta a eventi esterni e interni. Questi modelli rappresentano il sistema come uno stato iniziale, uno o più stati intermedi e uno stato finale. Gli eventi sono rappresentati come transizioni tra questi stati. Questo tipo di modello è particolarmente utile per rappresentare il comportamento reattivo dei sistemi, come sistemi embedded o software di controllo.

Modelli di Dati Semantici

I modelli di dati semanticci sono utilizzati per definire la struttura logica dei dati elaborati da un sistema. Un modello entità-relazione-attributo è un esempio comune di questo tipo di modello. Questo tipo di modello aiuta a definire le entità (oggetti o concetti nel sistema), le relazioni tra queste entità e gli attributi delle entità stesse. È fondamentale per la progettazione dei database e per comprendere come i dati sono organizzati e correlati all'interno del sistema.

Dizionari dei Dati

I dizionari dei dati sono documenti o elenchi di tutte le definizioni dei nomi utilizzati nei modelli del sistema insieme alle relative descrizioni. Questi dizionari servono a garantire una comprensione e una coerenza comuni tra i membri del team di sviluppo. Possono includere definizioni di entità, relazioni, attributi e altri termini utilizzati nei modelli, consentendo una comunicazione chiara e una comprensione condivisa tra gli stakeholder.

Voci del dizionario dati

Data dictionary entries

| Name | Description | Type | Date |
|-----------------|---|-----------|------------|
| Article | Details of the published article that may be ordered by people using LIBSYS. | Entity | 30.12.2002 |
| authors | The names of the authors of the article who may be due a share of the fee. | Attribute | 30.12.2002 |
| Buyer | The person or organisation that orders a copy of the article. | Entity | 30.12.2002 |
| fee-payable-to | A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee. | Relation | 29.12.2002 |
| Address (Buyer) | The address of the buyer. This is used to any paper billing information that is required. | Attribute | 31.12.2002 |

Modelli di Oggetti

I modelli di oggetti nell'ingegneria dei requisiti descrivono il sistema in termini di classi di oggetti e delle loro associazioni.

Queste classi rappresentano gli insiemi di oggetti con attributi comuni e operazioni che supportano.

I modelli di oggetti includono diverse tipologie, tra cui:

- Modelli basati sull'ereditarietà.
- Modelli basati sull'aggregazione.
- Modelli basati sull'interazione tra le classi di oggetti.

Modelli di oggetti e l'UML

L'UML, o Unified Modeling Language, è uno standard ampiamente utilizzato per la modellazione orientata agli oggetti nel campo dello sviluppo software.

Questa notazione fornisce una lingua visuale per rappresentare in modo chiaro e comprensibile il design di sistemi software, compresi oggetti, classi, relazioni e comportamenti.

L'UML è un'importante risorsa che aiuta gli sviluppatori e gli stakeholder a comunicare in modo efficace, riducendo l'ambiguità nei progetti e offrendo una base solida per l'implementazione pratica dei sistemi.

Metodi strutturati

I metodi strutturati sono un'importante componente dell'ingegneria del software. Essi integrano la modellazione del sistema come parte fondamentale del processo di sviluppo.

Questi metodi definiscono modelli, un processo di creazione di tali modelli e forniscono regole e linee guida.

Gli strumenti CASE supportano questa modellazione, contribuendo all'efficienza e alla qualità del software.

Debolezze dei metodi

I metodi formali per la specifica dei requisiti presentano alcune debolezze da considerare:

- **Assenza di Requisiti Non Funzionali:** In molti casi, i metodi formali si concentrano principalmente sui requisiti funzionali e potrebbero non affrontare in modo esaustivo i requisiti non funzionali, come le prestazioni, la sicurezza o l'usabilità.
- **Mancanza di Indicazioni sull'Appropriazione del Metodo:** Non forniscono spesso una guida chiara su quando un metodo formale sia appropriato per affrontare un determinato problema. Questo richiede una valutazione delle caratteristiche del progetto e dei costi associati all'applicazione dei metodi formali.
- **Eccessiva Documentazione:** L'uso eccessivo di metodi formali può portare a una quantità eccessiva di documentazione, che può risultare onerosa da gestire. È importante bilanciare la quantità di documentazione con le esigenze effettive del progetto.
- **Dettagli e Complessità dei Modelli di Sistema:** Talvolta, i modelli del sistema creati con metodi formali possono essere troppo dettagliati e complessi, rendendoli difficili da comprendere da parte degli stakeholder o degli utenti del sistema.

Convalida dei requisiti

La convalida dei requisiti è un processo cruciale che mira a dimostrare che i requisiti definiti nel documento effettivamente rappresentano ciò che il cliente desidera. Gli errori nei requisiti possono portare a costi elevati e problemi durante lo sviluppo del sistema, quindi la convalida è di importanza critica.

Verifica dei requisiti

La verifica dei requisiti si concentra su vari aspetti chiave per garantire che i requisiti siano di alta qualità e che il sistema li soddisfi adeguatamente. Alcuni aspetti chiave della verifica includono:

- **Validità:** Verificare se il sistema fornisce le funzioni necessarie per soddisfare le esigenze del cliente.
- **Coerenza:** Rilevare e risolvere eventuali conflitti tra i requisiti.
- **Completezza:** Accertarsi che tutti i servizi richiesti dal cliente siano inclusi nei requisiti.
- **Realismo:** Valutare se i requisiti possono essere implementati in base al budget disponibile e alla tecnologia disponibile.
- **Verificabilità:** Accertarsi che i requisiti siano formulati in modo tale che possano essere verificati e testati.

Tecniche di convalida dei requisiti

Esistono diverse tecniche per convalidare i requisiti, tra cui:

- **Revisioni dei Requisiti:** Questo metodo coinvolge un'analisi manuale e sistematica dei requisiti, in cui un gruppo di revisori esamina i requisiti e li confronta con criteri di qualità specifici.
- **Prototipazione:** La creazione di un modello eseguibile del sistema può aiutare a convalidare i requisiti. I prototipi possono fornire un'anteprima del sistema e consentire agli stakeholder di vedere se soddisfa le loro aspettative.
- **Generazione di Casi di Test:** La generazione di casi di test mira a sviluppare test specifici per i requisiti al fine di verificarne la testabilità.

Controlli delle revisioni

Durante le revisioni dei requisiti, è fondamentale eseguire controlli per garantire che i requisiti siano di alta qualità e integrità. Alcuni controlli chiave includono:

- **Verificabilità:** Accertarsi che i requisiti siano formulati in modo realistico e verificabile.
- **Comprensibilità:** Assicurarsi che i requisiti siano scritti in modo chiaro e possano essere compresi da tutti gli stakeholder.
- **Tracciabilità:** Garantire che l'origine di ciascun requisito sia chiaramente indicata e che sia possibile risalire alla sua fonte.
- **Adattabilità:** Valutare se i requisiti possono essere modificati senza causare impatti significativi su altri requisiti o componenti del sistema.

Questi controlli contribuiscono a garantire la qualità e l'integrità dei requisiti definiti.

Gestione dei requisiti e la loro incompletezza

La gestione consiste nella gestione delle modifiche e delle evoluzioni dei requisiti lungo tutto il ciclo di vita di un progetto.

I requisiti sono spesso soggetti a cambiamenti durante il processo di sviluppo. Questo può derivare da una migliore comprensione del sistema, dall'emergere di nuove esigenze o dalla scoperta di incongruenze tra i diversi punti di vista degli stakeholder.

La gestione dei requisiti ha lo scopo di affrontare questa dinamica in modo strutturato. Si preoccupa di catturare, documentare e valutare i cambiamenti nei requisiti, garantendo la tracciabilità tra le diverse versioni.

Questo processo è fondamentale per mantenere il progetto allineato con gli obiettivi iniziali e per garantire che il sistema soddisfi le aspettative degli utenti.

Tracciabilità

La tracciabilità gestisce le relazioni tra requisiti, le loro fonti e la progettazione del sistema, contribuendo a mantenere chiarezza e coerenza nel ciclo di vita dello sviluppo del software.

Un aspetto importante è la **tracciabilità delle fonti**, che collega i requisiti agli stakeholder che li hanno proposti. Questo collegamento fornisce contesto e assicura che le esigenze degli stakeholder siano prese in considerazione nel processo di sviluppo.

Un altro aspetto chiave è la **tracciabilità dei requisiti**, che collega i requisiti tra loro quando esistono dipendenze. Questo garantisce la coerenza e l'interconnessione dei requisiti all'interno del sistema.

Infine, la **tracciabilità della progettazione** collega i requisiti alla progettazione del sistema, garantendo che la progettazione sia allineata con i requisiti identificati dagli stakeholder. La tracciabilità è cruciale per mantenere chiarezza, integrità e consapevolezza nei progetti di ingegneria dei requisiti.

Matrice di tracciabilità

A traceability matrix

- A traceability matrix may be maintained to keep traceability information which **relate requirements to sources, each other and design module**

| Req. id | 1.1 | 1.2 | 1.3 | 2.1 | 2.2 | 2.3 | 3.1 | 3.2 |
|------------|-----|-----|-----|-----|-----|-----|-----|-----|
| 1.1 | D | R | | | | | | |
| 1.2 | | D | | | D | | D | |
| 1.3 | R | | R | | | | | |
| 2.1 | | R | | D | | | D | |
| 2.2 | | | | | | | D | |
| 2.3 | R | | D | | | | | |
| 3.1 | | | | | | | R | |
| 3.2 | | | | | | R | | |

Supporto agli Strumenti CASE (Computer-Aided Software Engineering)

Nel contesto della gestione dei requisiti, gli strumenti CASE svolgono un ruolo significativo. Questi strumenti offrono diverse funzionalità, tra cui:

- **Archiviazione dei Requisiti:** I requisiti dovrebbero essere conservati in un archivio dati sicuro e ben gestito, consentendo un accesso facile e un controllo sulla loro evoluzione nel tempo.
- **Gestione dei Cambiamenti:** Il processo di gestione dei cambiamenti è strutturato come un flusso di lavoro, le cui fasi possono essere definite in anticipo. L'obiettivo è automatizzare parzialmente il flusso di informazioni tra queste fasi per consentire una gestione efficiente delle modifiche ai requisiti nel corso del tempo.
- **Gestione della Tracciabilità:** Gli strumenti CASE spesso supportano la tracciabilità dei requisiti, consentendo il recupero automatico dei collegamenti tra i requisiti, le fonti e la progettazione.

Il Documento dei requisiti

Il documento dei requisiti è l'affermazione ufficiale di ciò che è richiesto dagli sviluppatori del sistema. Un documento dei requisiti ben strutturato dovrebbe includere due parti principali:

- una definizione dei requisiti degli utenti
- una specifica dei requisiti del sistema

Queste due parti consentono di distinguere tra ciò che gli utenti finali richiedono e ciò che il sistema deve effettivamente fare.

Lo standard dei requisiti IEEE

Lo standard dei requisiti dell'IEEE fornisce una struttura generica per un documento dei requisiti che deve essere istanziata e personalizzata per ogni sistema specifico. Questo standard aiuta a garantire che i requisiti siano documentati in modo chiaro e coerente, consentendo una migliore comprensione e gestione dei requisiti.

Struttura del documento dei requisiti

Il documento dei requisiti è un componente essenziale nell'ingegneria dei requisiti, fornendo un quadro completo e organizzato dei requisiti di un sistema. La sua struttura tipica può includere le seguenti sezioni:

- **Prefazione:** Questa sezione introduttiva offre una panoramica del documento e del sistema per cui sono definiti i requisiti.
- **Introduzione:** Qui vengono fornite informazioni di base sul progetto, inclusa la sua motivazione e il contesto.
- **Glossario:** Il glossario elenca e definisce i termini tecnici o specifici del dominio utilizzati nel documento, garantendo chiarezza e coerenza nell'interpretazione dei requisiti.
- **Definizione dei Requisiti degli Utenti:** Questa sezione delinea i requisiti fondamentali del sistema, espressi in modo comprensibile per gli utenti finali o i principali stakeholder.
- **Architettura del Sistema:** Qui si fornisce un'illustrazione della struttura generale del sistema, evidenziando i principali componenti e le loro relazioni. Questa sezione può aiutare a visualizzare come i requisiti si tradurranno in una progettazione architettonica.
- **Specifiche dei Requisiti del Sistema:** In questa parte del documento, vengono elencati tutti i requisiti specifici, suddivisi in requisiti funzionali e non funzionali. Ogni requisito è definito in modo chiaro e deve essere verificabile.
- **Modelli di Sistema:** I modelli di sistema possono essere utilizzati per rappresentare visivamente il comportamento previsto del sistema. Questi modelli possono includere diagrammi di flusso, diagrammi di sequenza o altre rappresentazioni grafiche.
- **Evoluzione del Sistema:** Questa sezione contempla eventuali piani futuri per il sistema, come aggiornamenti o miglioramenti previsti. Potrebbe anche includere una cronologia degli sviluppi previsti.
- **Appendici:** Le appendici possono contenere informazioni aggiuntive che potrebbero essere rilevanti per la comprensione dei requisiti, ma che non sono strettamente parte dei requisiti stessi.
- **Indice:** Un indice facilita la ricerca di requisiti specifici o concetti all'interno del documento.

Metodi formali

I metodi formali sono un approccio all'ingegneria del software che si basa sulla rappresentazione matematica e sull'analisi rigorosa del software. Questi metodi hanno dimostrato di offrire numerosi vantaggi, tra cui la riduzione del numero di difetti nei sistemi. La loro principale area di applicazione si trova nell'ingegneria di sistemi critici, dove gli errori possono avere conseguenze gravi.

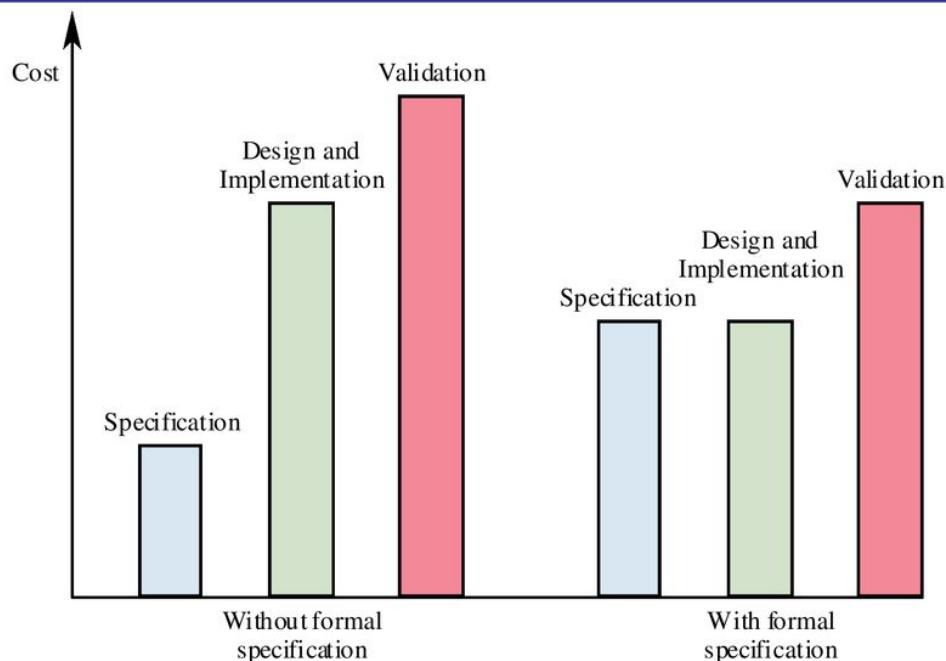
Uso dei metodi formali

I metodi formali vengono impiegati in vari contesti per migliorare la qualità del software. I principali vantaggi derivanti dall'uso di metodi formali includono:

- **Riduzione del numero di difetti:** L'approccio rigoroso e matematico alla specifica e alla verifica dei requisiti aiuta a identificare e correggere gli errori in modo proattivo.
- **Applicazione nell'ingegneria di sistemi critici:** I sistemi critici, come quelli utilizzati in ambiti come l'aviazione, la medicina o la sicurezza, traggono particolare beneficio dai metodi formali, poiché la loro affidabilità è di primaria importanza.
- **Economicità:** Sebbene l'adozione iniziale dei metodi formali possa richiedere investimenti significativi, tali costi sono spesso inferiori rispetto agli alti costi di fallimento di un sistema critico.

Costi di sviluppo con specificazione formale

Development Costs with Formal Specification



Tecniche di specifica

I metodi formali includono diverse tecniche di specifica, tra cui:

- **Specifiche Algebrica:** Questo approccio basato su operazioni e relazioni matematiche è utile per descrivere i requisiti in modo rigoroso.
- **Specifiche Basate su Modelli:** Questa tecnica impiega modelli di stato costruiti utilizzando costrutti matematici come insiemi e sequenze. I modelli di stato aiutano a rappresentare in modo chiaro il comportamento previsto del sistema.

List specification

```
LIST ( Elem )  
sort List  
imports INTEGER  
  
Defines a list where elements are added at the end and removed from the front. The operations are Create, which brings an empty list into existence, Cons, which creates a new list with an added member, Length, which evaluates the list size, Head, which evaluates the front element of the list, and Tail, which creates a list by removing the head from its input list. Undefined represents an undefined value of type Elem.  
  
Create → List  
Cons(List,Elem) → List  
Head (List) → Elem  
Length (List) → Integer  
Tail (List) → List  
  
Head (Create) = Undefined exception (empty list)  
Head (Cons (L, v)) = if L = Create then v else Head (L)  
Length (Create) = 0  
Length (Cons (L, v)) = Length (L) + 1  
Tail (Create) = Create  
Tail (Cons (L, v)) = if L = Create then Create else Cons (Tail (L), v)
```

Design

Progettazione e realizzazione del software

La progettazione e realizzazione del software rappresentano il processo di trasformare una specifica del sistema in un sistema eseguibile. Questo processo coinvolge due fasi principali: progettazione del software e implementazione. È importante notare che queste attività sono strettamente collegate e spesso si intrecciano tra loro.

Progettazione del Software

La progettazione del software è la fase in cui si definisce la struttura software necessaria per realizzare la specifica del sistema. Durante questa fase, gli ingegneri del software creano un piano dettagliato che descrive come il sistema sarà organizzato, quali componenti software saranno necessari, come interagiranno tra loro e come verranno gestite le diverse funzionalità del software. La progettazione del software mira a tradurre i requisiti del sistema in un modello che possa essere implementato in codice.

Implementazione

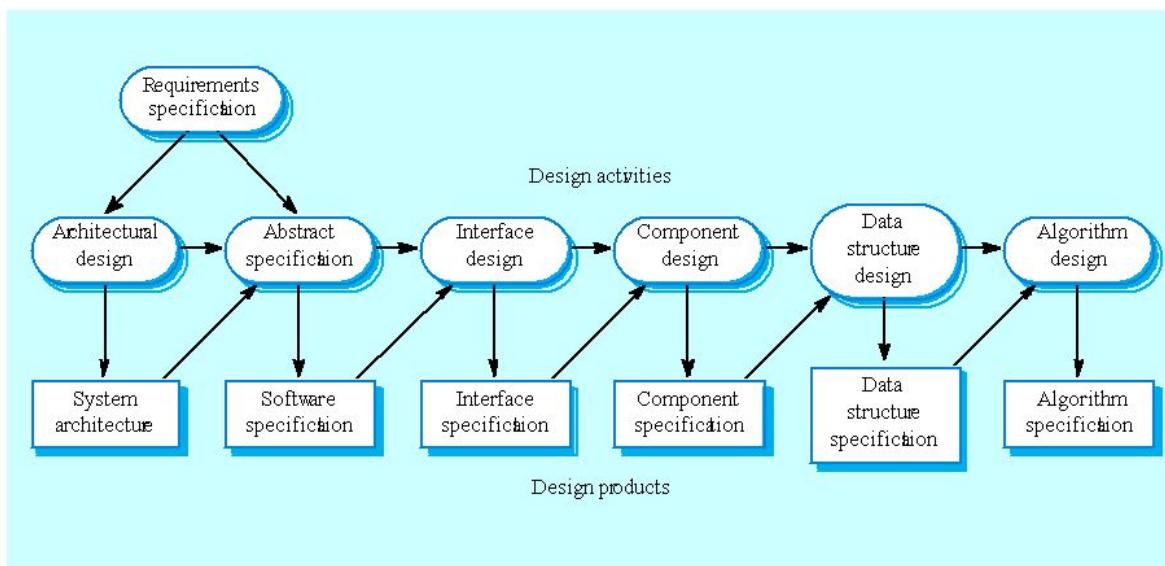
Una volta completata la progettazione del software, inizia l'implementazione. Questa fase coinvolge la traduzione della struttura software progettata in un programma eseguibile. Gli sviluppatori scrivono il codice sorgente in base ai dettagli forniti dalla progettazione, tenendo conto di aspetti quali l'efficienza, la manutenibilità e la qualità del codice. Durante l'implementazione, vengono utilizzati linguaggi di programmazione e strumenti di sviluppo per creare il software in conformità con i requisiti stabiliti nella specifica del sistema.

Attività del processo di progettazione

- Progettazione architetturale
- Specifica astratta
- Progettazione dell'interfaccia
- Progettazione dei componenti
- Progettazione delle strutture dei dati
- Progettazione degli algoritmi

Il processo di progettazione del software

The software design process



Modified from Sommerville's originals

Software Engineering, 7th edition. Chapter 11

Slide 4

Architettura software

Progettazione architettonica

La progettazione architettonica comporta l'identificazione dei sotto-sistemi e la specifica dei framework di controllo e comunicazione. Una descrizione dell'architettura del software è l'output di questo processo di progettazione.

Strutturazione del sistema

La strutturazione del sistema è una fase che riguarda la decomposizione del sistema in sotto-sistemi che interagiscono tra loro.

L'architettura è normalmente espressa attraverso diagrammi a blocchi e linee che forniscono una panoramica della struttura del sistema. Questi diagrammi sono utili per la comunicazione con gli stakeholder e la pianificazione del progetto.

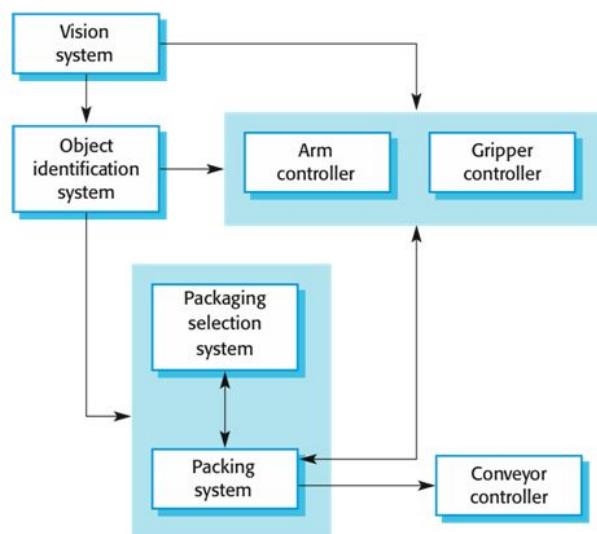
Si possono sviluppare modelli più specifici che mostrano come questi sotto-sistemi condividono dati, sono distribuiti e come si interfacciano tra loro.

Diagrammi a Blocchi e Linee

I diagrammi a blocchi sono un modo comune di rappresentare la progettazione architetturale di un sistema. Sono molto astratti e tendono a non mostrare dettagli come la natura delle relazioni tra i vari componenti o le proprietà visibili esternamente dei sotto-sistemi. Tuttavia, sono molto utili per la comunicazione con gli stakeholder del progetto e per la pianificazione del progetto, in quanto offrono una visualizzazione chiara della struttura generale del sistema.

Sistema di controllo del robot di imballaggio

Figure 6.1 The architecture of a packing robot control system



Vantaggi di un'architettura esplicita

Un'architettura esplicita comporta numerosi vantaggi che possono influire positivamente sulla gestione e lo sviluppo di un sistema:

1. **Comunicazione con gli stakeholder:** L'architettura può svolgere un ruolo cruciale come punto focale per le discussioni e le interazioni con gli stakeholder del sistema. Fornisce una rappresentazione chiara e condivisa di come il sistema è strutturato e funziona, facilitando la comunicazione tra sviluppatori, utenti, e altri attori interessati.
2. **Analisi del sistema:** Un'architettura esplicita consente l'analisi dettagliata del sistema per determinare se è in grado di soddisfare i suoi requisiti non funzionali. Questa analisi può evidenziare eventuali carenze o problematiche nell'architettura che devono essere affrontate durante lo sviluppo. Inoltre, aiuta a valutare le prestazioni, la sicurezza, la disponibilità e altre caratteristiche critiche del sistema.
3. **Riutilizzo su larga scala:** Un altro grande vantaggio di un'architettura ben definita è il potenziale per il riutilizzo. Una volta sviluppata e validata, l'architettura può essere applicata a una serie di sistemi simili o correlati. Questo risparmio di tempo e risorse può essere significativo, specialmente quando si tratta di sistemi complessi.

Architettura e caratteristiche del sistema

L'architettura del sistema gioca un ruolo fondamentale nell'influenzare le diverse caratteristiche del sistema. Ecco alcune considerazioni chiave relative a specifiche caratteristiche:

1. **Prestazioni:** Per ottenere prestazioni ottimali, è importante localizzare le operazioni critiche e minimizzare le comunicazioni tra i componenti del sistema. L'uso di componenti di grandi dimensioni anziché quelli granulari può contribuire a ottimizzare le prestazioni, riducendo l'overhead delle comunicazioni.
2. **Sicurezza:** L'architettura del sistema è fondamentale per garantire la sicurezza. L'utilizzo di un'architettura a strati, con gli asset critici posizionati nei livelli interni, può contribuire a isolare e proteggere questi elementi importanti da potenziali minacce esterne.
3. **Disponibilità:** Per migliorare la disponibilità del sistema, è essenziale includere componenti ridondanti e meccanismi di tolleranza ai guasti nell'architettura. Questi elementi aiutano a garantire che il sistema rimanga operativo anche in caso di guasti hardware o software.
4. **Manutenibilità:** La manutenibilità è agevolata dall'uso di componenti granulari e sostituibili. Questa caratteristica consente di apportare modifiche e aggiornamenti al sistema in modo più efficiente, riducendo il rischio di impatti negativi su altre parti dell'architettura.

Un'architettura esplicita è quindi un elemento chiave nella progettazione di sistemi complessi, offrendo un quadro chiaro e strutturato per la realizzazione e la gestione di sistemi efficaci, sicuri e ad alte prestazioni.

Decisioni di progettazione architetturale

Le decisioni di progettazione architetturale sono fondamentali nella creazione di un sistema software. Queste decisioni definiscono l'aspetto generale e la struttura dell'applicazione e possono influenzare notevolmente il suo comportamento e le prestazioni. Alcune delle decisioni di progettazione architetturale includono:

- **Esistenza di un'architettura generica:** La prima decisione riguarda l'adozione di un'architettura generica, se possibile, che possa servire come base per l'applicazione. Questo può semplificare lo sviluppo e migliorare la manutenibilità del sistema.
- **Distribuzione del sistema:** La decisione su come distribuire il sistema è cruciale. Si tratta di determinare se il software verrà eseguito su un singolo computer o su una rete di computer e come le componenti interagiranno tra loro.
- **Stili architetturali:** Gli stili architetturali rappresentano modelli comuni di organizzazione dei componenti di un sistema. La scelta di uno stile architetturale dipende dalle esigenze specifiche dell'applicazione.
- **Approccio di strutturazione del sistema:** L'approccio di strutturazione del sistema riguarda la modalità di organizzazione dei componenti, dei moduli o delle classi che costituiscono il sistema. Questa decisione influisce sulla modularità e sulla manutenibilità del sistema.
- **Decomposizione del sistema:** La decomposizione del sistema riguarda la scomposizione del sistema in moduli o componenti più piccoli, ognuno con un compito specifico. Questa suddivisione facilita lo sviluppo e il testing.
- **Strategia di controllo:** La strategia di controllo definisce come le componenti del sistema interagiscono e coordinano tra loro durante l'esecuzione. Può riguardare aspetti come la sincronizzazione dei processi o il flusso di controllo.
- **Valutazione della progettazione architetturale:** È necessario definire come verrà valutata la progettazione architetturale per garantire che soddisfi gli obiettivi e i requisiti del sistema.
- **Documentazione dell'architettura:** La documentazione dell'architettura è essenziale per comunicare le decisioni prese e la struttura dell'applicazione a tutti gli stakeholder. Deve essere chiara, completa e facilmente comprensibile.

Modelli architetturali

I modelli architetturali vengono utilizzati per documentare la progettazione architettonica. Ci sono diversi tipi di modelli, tra cui:

- **Modello strutturale statico:** Questo modello mostra i principali componenti del sistema e le relazioni tra di essi. È una rappresentazione statica dell'architettura del sistema.
- **Modello di processo dinamico:** Questo modello rappresenta la struttura del processo del sistema, mostrando come le componenti interagiscono dinamicamente durante l'esecuzione.
- **Modello di interfaccia:** Questo modello definisce le interfacce esposte dai sotto-sistemi o componenti del sistema. Specifica come le diverse parti del sistema comunicano tra loro.
- **Modello delle relazioni:** I modelli delle relazioni possono includere rappresentazioni come modelli di flusso dati, che mostrano come i dati si spostano attraverso il sistema e le relazioni tra i sotto-sistemi.
- **Modello di distribuzione:** Questo modello mostra come i sotto-sistemi sono distribuiti tra i computer fisici o le risorse di rete. È importante per i sistemi distribuiti.

Spesso, durante il processo di progettazione architettonica, vengono creati diversi modelli architettonici, ognuno dei quali fornisce una prospettiva specifica sull'architettura del sistema, aiutando a comunicare e comprendere meglio la struttura del sistema.

Organizzazione del sistema

L'organizzazione del sistema riflette la strategia di base utilizzata per strutturare un sistema. Alcuni stili organizzativi comuni includono:

Modello del repository

Il modello del repository è un approccio nell'ambito dei sistemi informativi in cui diversi sotto-sistemi devono scambiare dati tra di loro. In questo modello, i dati condivisi sono conservati in un database centrale o repository e possono essere accessibili da tutti i sotto-sistemi. Ogni sotto-sistema mantiene il proprio database e passa dati esplicitamente ad altri sotto-sistemi quando necessario. Questo approccio è comunemente utilizzato quando si ha bisogno di condividere grandi quantità di dati tra i vari componenti del sistema.

Vantaggi del modello repository:

- È un modo efficiente per condividere grandi quantità di dati tra i sotto-sistemi.
- I sotto-sistemi non devono preoccuparsi di come vengono prodotti i dati, poiché la gestione dei dati è centralizzata, inclusi aspetti come il backup e la sicurezza.
- Il modello di condivisione dei dati viene pubblicato come uno schema del repository, rendendo chiaro come i dati sono strutturati e accessibili.

Svantaggi del modello repository:

- I sotto-sistemi devono concordare su un modello dati specifico del repository, il che può richiedere compromessi inevitabili.
- L'evoluzione dei dati nel repository può essere difficile e costosa, poiché le modifiche devono essere coordinate tra tutti i sotto-sistemi.
- Questo modello non supporta politiche di gestione specifiche per i dati in modo flessibile.
- La distribuzione efficiente del sistema basato su un repository centralizzato può risultare complicata.

In sintesi, il modello del repository è efficace per condividere grandi quantità di dati tra i sotto-sistemi, ma richiede una pianificazione accurata e può comportare sfide nella gestione e nell'evoluzione dei dati condivisi.

Modello client-server

Il modello client-server è un tipo di sistema distribuito che illustra come i dati e l'elaborazione sono distribuiti tra una serie di componenti. In questo modello, un insieme di server autonomi fornisce servizi specifici, come ad esempio la stampa o la gestione dei dati, mentre un insieme di client utilizza questi servizi. La comunicazione tra client e server avviene attraverso una rete che consente ai client di accedere ai server.

Caratteristiche del modello client-server:

- **Modello di sistema distribuito:** Il modello client-server mostra come i dati e l'elaborazione sono distribuiti in una serie di componenti.
- **Un insieme di server autonomi:** Questi server forniscono servizi specifici, come la stampa, la gestione dei dati, ecc.
- **Un insieme di client che chiamano questi servizi:** I client utilizzano i servizi forniti dai server per svolgere le proprie attività.
- **Una rete che consente ai client di accedere ai server:** La comunicazione avviene attraverso una rete che collega i client ai server.

Vantaggi del modello client-server:

- **La distribuzione dei dati è semplice:** La struttura del modello semplifica la gestione dei dati condivisi.
- **Utilizza in modo efficace sistemi in rete:** Questo modello sfrutta in modo efficiente le reti, il che potrebbe ridurre la necessità di hardware costoso.
- **Facilità di aggiungere nuovi server o aggiornare server esistenti:** Il modello permette una facile espansione e aggiornamento dei server.

Svantaggi del modello client-server:

- **Nessun modello dati condiviso:** I sotto-sistemi possono utilizzare diverse organizzazioni dei dati, il che potrebbe causare inefficienze nello scambio di dati.
- **Gestione ridondante in ciascun server:** Ogni server richiede una gestione separata, aumentando la complessità.
- **Nessun registro centrale dei nomi e dei servizi:** Trovare e individuare i server e i servizi disponibili all'interno del sistema può risultare difficile.

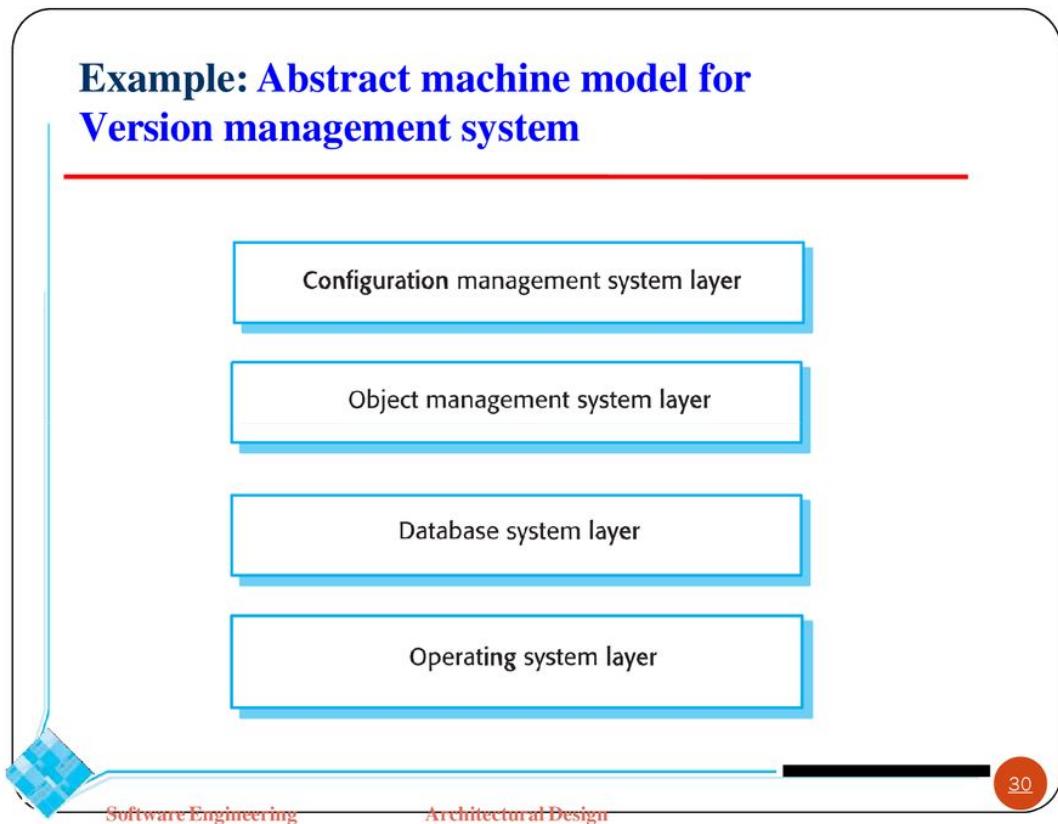
In sintesi, il modello client-server favorisce la modularità e la scalabilità, ma richiede attenzione alla gestione dei dati condivisi e alla configurazione dei server.

Modello di macchina astratta o a strati

Il modello di Macchina Astratta o a Strati organizza il sistema in strati o livelli di astrazione. Ogni strato offre un insieme ben definito di funzionalità e interagisce solo con strati adiacenti. Questo favorisce la separazione delle preoccupazioni e la manutenibilità.

- **Usato per modellare l'interfacciamento dei sotto-sistemi:** Questo modello è utilizzato per modellare come i sotto-sistemi interagiscono tra loro tramite interfacce ben definite.
- **Organizza il sistema in un insieme di livelli (o macchine astratte) ciascuno dei quali fornisce un insieme di servizi:** Il sistema è suddiviso in livelli, o macchine astratte, ciascuno dei quali offre un insieme specifico di servizi.
- **Supporta lo sviluppo incrementale dei sotto-sistemi in diversi livelli:** L'approccio a strati permette lo sviluppo incrementale, in modo che le modifiche all'interfaccia di un livello influiscano solo su quello adiacente.
- **Tuttavia, spesso è artificiale strutturare i sistemi in questo modo:** In alcuni casi, la strutturazione a strati può sembrare artificiale e complicare la progettazione.

Sistema di gestione delle versioni



Sotto-sistemi e moduli

Nell'ambito dell'ingegneria dei sistemi, due concetti importanti da considerare sono i sotto-sistemi e i moduli. Questi concetti aiutano a suddividere e organizzare un sistema complesso in unità più gestibili, contribuendo a definire chiaramente le relazioni e le funzionalità all'interno del sistema.

Sotto-sistemi

Un sotto-sistema è un componente che costituisce un sistema a tutti gli effetti. La caratteristica distintiva di un sotto-sistema è la sua operatività indipendente: può funzionare autonomamente senza dover dipendere da altri sotto-sistemi per eseguire le sue funzioni. In pratica, ciò significa che un sotto-sistema ha la capacità di operare come un sistema separato all'interno di un contesto più ampio. Questo concetto è fondamentale per definire le diverse parti di un sistema complesso e stabilire chiaramente le relazioni tra di esse.

Moduli

I moduli sono componenti di sistema progettati per collaborare all'interno di un sistema più grande che forniscono servizi ad altri componenti, e sebbene possano essere interconnessi tra loro per offrire un servizio più ampio, non costituirebbero un sistema completo e autonomo da soli. Invece, i moduli aiutano a suddividere le funzionalità in unità più piccole e gestibili all'interno di un sistema complesso, semplificando la progettazione e la manutenzione.

Decomposizione modulare

La decomposizione modulare è un ulteriore livello strutturale in cui i sottosistemi vengono suddivisi in moduli più piccoli. Esistono due principali modelli di decomposizione modulare:

1. **Modello ad Oggetti:** In questo modello, il sistema è suddiviso in oggetti che interagiscono tra loro. Gli oggetti rappresentano le componenti del sistema e comunicano tra loro per realizzare le funzionalità richieste. Questo approccio è spesso utilizzato in programmazione orientata agli oggetti.
2. **Modello a Pipeline (o Flusso Dati):** In questo modello, il sistema è suddiviso in moduli funzionali che trasformano input in output. Ogni modulo svolge una specifica operazione e i dati vengono passati attraverso una sequenza di moduli, simile a come l'acqua scorre in una tubazione. Questo approccio è efficace per gestire il flusso dei dati all'interno del sistema.

È importante notare che, quando possibile, le decisioni relative alla concorrenza dovrebbero essere ritardate fino a quando i vari moduli non sono stati implementati. La scelta di come gestire la concorrenza tra moduli può influire notevolmente sulle prestazioni e sulla correttezza complessiva del sistema.

La decomposizione modulare è un approccio fondamentale nello sviluppo software che aiuta a organizzare complessi sistemi informatici in modo più gestibile e comprensibile.

Stili di decomposizione modulare

Gli stili di decomposizione modulare sono un'importante strategia nell'organizzazione dei sistemi. Questo approccio mira a suddividere i sotto-sistemi in moduli per affrontare la complessità, migliorare la manutenibilità e promuovere il riuso. La chiave è creare moduli con funzioni specifiche e ridurre le dipendenze tra di essi. Questa struttura modulare è strettamente legata all'organizzazione generale del sistema, influenzando direttamente la sua architettura complessiva.

Modello ad oggetti

I modelli ad oggetti sono un approccio di progettazione che struttura un sistema in un insieme di oggetti poco accoppiati, ognuno con interfacce ben definite. Questa decomposizione orientata agli oggetti riguarda l'identificazione delle classi di oggetti, dei loro attributi e delle operazioni che possono essere eseguite su di essi. Quando questi modelli vengono implementati, gli oggetti vengono creati da queste classi, e un qualche modello di controllo viene utilizzato per coordinare le operazioni degli oggetti.

Vantaggi del modello ad oggetti:

- **Poco accoppiamento tra gli oggetti:** La loro implementazione può essere modificata senza influenzare altri oggetti nel sistema.
- **Rappresentazione di entità reali:** Semplifica la progettazione di software in grado di modellare situazioni complesse.

Svantaggi del modello ad oggetti:

- **Problematiche nelle modifiche dell'interfaccia:** Modifiche all'interfaccia degli oggetti possono avere un impatto su altre parti del sistema.
- **Complessità nella rappresentazione di entità complesse:** Possono richiedere una modellazione sofisticata.

Pipelining orientato alle funzioni

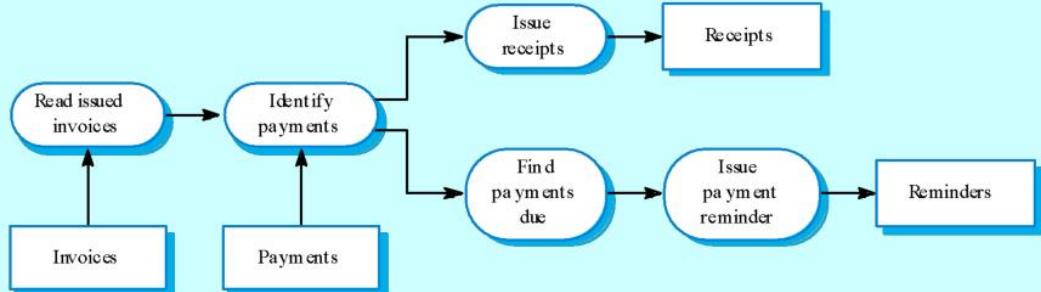
La modellazione del flusso di lavoro orientato alle funzioni coinvolge trasformazioni funzionali che elaborano i loro input per produrre output.

Questo approccio può essere descritto come un modello a tubo e filtro, analogo a quanto si riscontra nella shell UNIX.

Variante comuni di questo approccio esistono, ma quando le trasformazioni sono sequenziali, si configura un modello sequenziale batch che trova ampio utilizzo nei sistemi di elaborazione dati.

È importante notare che questo modello non è particolarmente adatto per sistemi interattivi. In tali casi, potrebbero essere necessari approcci diversi per gestire l'interazione con gli utenti.

Example: Invoice processing system



Software Engineering

Architecture design

Slide 35

Caratteristiche del modello a pipeline

Riutilizzo delle trasformazioni: Una delle principali qualità di questo modello è la sua capacità di supportare il riutilizzo delle trasformazioni.

Organizzazione intuitiva: Il modello a pipeline offre un'organizzazione intuitiva per la comunicazione con gli stakeholder.

Facilità di aggiunta di nuove trasformazioni: La facilità di aggiungere nuove trasformazioni è un'altra caratteristica positiva di questo modello.

Versatilità nell'implementazione: Dal punto di vista dell'implementazione, il modello a pipeline è relativamente semplice da realizzare sia come sistema concorrente che sequenziale.

Sfide del modello a pipeline

Formato comune per il trasferimento dei dati: Tuttavia, è importante notare che il modello a pipeline richiede un formato comune per il trasferimento di dati lungo il percorso della pipeline.

Limitazioni nell'interazione basata su eventi: Inoltre, il modello a pipeline è meno adatto per supportare l'interazione basata su eventi.

Stili di controllo

Gli stili di controllo sono un aspetto importante dell'ingegneria dei sistemi che riguarda il flusso di controllo tra i sotto-sistemi. Questi stili di controllo sono distinti dal modello di decomposizione del sistema, che descrive come il sistema è suddiviso in sotto-sistemi o componenti. Due degli stili di controllo più comuni sono:

- **Controllo centralizzato**
- **Controllo basato su eventi**

Controllo Centralizzato

In un sistema con un controllo centralizzato, un sotto-sistema specifico assume la responsabilità globale per il controllo e coordina l'avvio e l'arresto degli altri sotto-sistemi. Due modelli di controllo centralizzato comuni sono:

- **Modello chiamata-ritorno:** il controllo inizia nella parte superiore di una gerarchia di subroutine e si sposta verso il basso. Questo modello è applicabile a sistemi sequenziali in cui il flusso di controllo segue un percorso lineare.
- **Modello di gestione:** che è adatto ai sistemi concorrenti, un componente specifico del sistema controlla l'avvio, l'arresto e il coordinamento degli altri processi del sistema. Anche se questo modello è spesso utilizzato in sistemi concorrenti, può essere implementato in sistemi sequenziali utilizzando dichiarazioni di tipo "case."

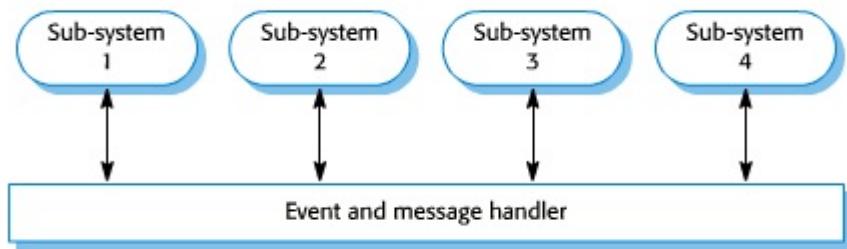
Sistemi basati su eventi

I sistemi basati su eventi sono gestiti da eventi generati esternamente, il cui tempismo è al di fuori del controllo dei sotto-sistemi che elaborano l'evento. Ci sono due principali modelli basati su eventi:

Modelli di diffusione

In questo modello, un evento viene trasmesso a tutti i sotto-sistemi. Qualsiasi sotto-sistema in grado di gestire l'evento può farlo. Questo modello è utile quando gli eventi devono essere elaborati da più parti del sistema, e la decisione su quale sotto-sistema gestirà un evento può variare dinamicamente. La politica di controllo non è incorporata nell'evento e nel gestore dei messaggi. I sottosistemi decidono quali eventi siano di loro interesse.

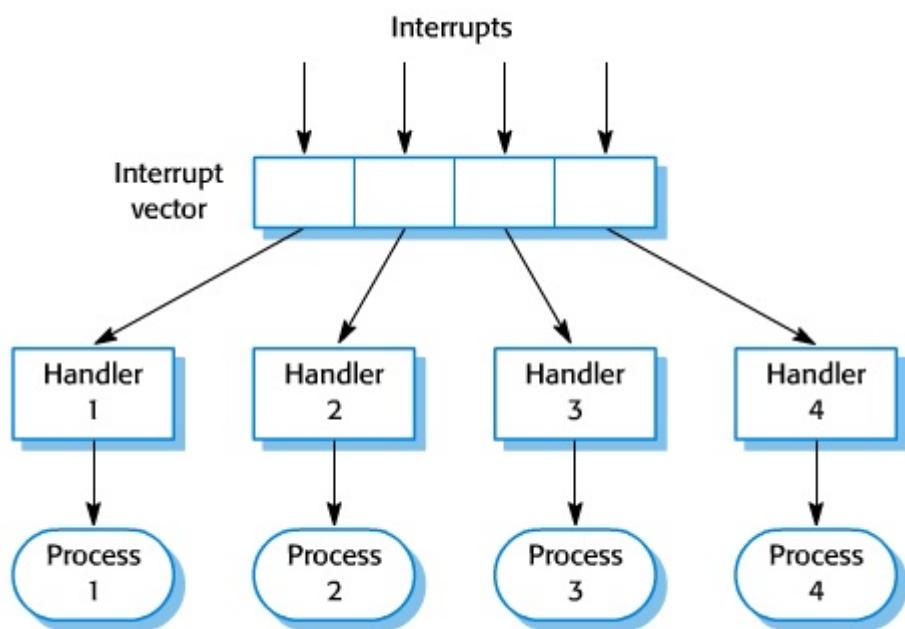
Trasmissione selettiva



Modelli basati su interruzioni

I modelli basati su interruzioni sono spesso adottati nei sistemi in tempo reale, in cui le interruzioni hardware vengono gestite da specifici gestori per garantire una risposta immediata agli eventi esterni. Questo approccio è essenziale quando è necessario rispettare rigorosi requisiti temporali e garantire che il sistema risponda in modo deterministico. Ciascun tipo di interruzione è associato a un gestore definito, semplificando la gestione degli eventi. Tuttavia, la complessità di programmazione e la validazione accurata dei gestori sono sfide significative in questi sistemi.

Controllo guidato dagli interrupt



Architetture dei sistemi distribuiti

Tipi di sistema

- **Sistemi personali:** Questi sistemi non sono distribuiti e sono progettati per funzionare su un computer personale o una stazione di lavoro. Sono tipicamente utilizzati da un singolo utente o da pochi utenti che accedono direttamente alla macchina.
- **Sistemi embedded:** Questi sistemi funzionano su un singolo processore o su un gruppo integrato di processori. Sono comunemente incorporati in dispositivi come telefoni cellulari, elettrodomestici, automobili e altri oggetti quotidiani. Sono progettati per svolgere funzioni specifiche.
- **Sistemi distribuiti:** Questi sistemi coinvolgono l'esecuzione del software di sistema su un gruppo debolmente integrato di processori che collaborano tra loro tramite una rete. Questa architettura permette la condivisione di risorse tra i computer che compongono il sistema.

Sistemi distribuiti

Il concetto di sistema distribuito è oggi ubiquitario nell'ambito dell'informatica. In un sistema distribuito, l'elaborazione delle informazioni non è limitata a una singola macchina, ma è distribuita su diversi computer. Questo approccio è diventato cruciale per i sistemi informatici aziendali, offrendo notevoli vantaggi.

Vantaggi dei sistemi distribuiti

- **Condivisione delle Risorse:** La condivisione di risorse comprende sia risorse hardware che software. Questa condivisione favorisce l'efficienza nell'utilizzo delle risorse disponibili.
- **Apertura:** I sistemi distribuiti sono aperti, il che implica l'utilizzo di attrezzature e software da diversi fornitori, promuovendo così la flessibilità e l'interoperabilità.
- **Concorrenza:** L'elaborazione concorrente è una caratteristica essenziale dei sistemi distribuiti. Questo tipo di elaborazione consente di migliorare le prestazioni, poiché più nodi della rete possono eseguire operazioni parallelamente.
- **Scalabilità:** I sistemi distribuiti possono essere facilmente ampliati o scalati per aumentare la loro capacità. Questo si ottiene aggiungendo nuove risorse o nodi alla rete in modo dinamico.
- **Tolleranza ai Guasti:** I sistemi distribuiti sono progettati per continuare a operare anche dopo la verifica di un guasto. Questa capacità di tolleranza ai guasti è cruciale per garantire la disponibilità continua dei servizi.

Svantaggi dei sistemi distribuiti

- **Complessità:** I sistemi distribuiti tendono ad essere più complessi rispetto a quelli centralizzati. La gestione di una rete di nodi richiede una pianificazione e un monitoraggio più attenti.
- **Sicurezza:** A causa della loro esposizione su reti, i sistemi distribuiti possono essere più suscettibili agli attacchi esterni. È necessario implementare rigorose misure di sicurezza per proteggere i dati e i servizi.
- **Gestione:** La gestione di un sistema distribuito richiede più sforzi rispetto a un sistema centralizzato. Questo include il monitoraggio delle risorse, la gestione dei guasti e l'implementazione di aggiornamenti su più nodi.
- **Imprevedibilità:** La risposta di un sistema distribuito può variare in base all'organizzazione della rete e al carico della stessa. Questa imprevedibilità richiede un'attenta progettazione e gestione per garantire le prestazioni desiderate.

Architetture dei sistemi distribuiti

Le architetture dei sistemi distribuiti forniscono una struttura organizzativa per le applicazioni che devono funzionare su più nodi o dispositivi di un sistema. Esistono diverse tipologie di architetture, tra cui le seguenti:

- **Architetture Client-Server:** in queste architetture i servizi distribuiti sono chiamati dai client. Questi servizi possono essere trattati in modo diverso rispetto ai client che li utilizzano. In altre parole, i client conoscono i server, ma i server non necessariamente conoscono i client. Questo modello permette una distribuzione flessibile delle responsabilità tra i nodi del sistema.
- **Architetture di oggetti distribuiti:** queste architetture eliminano la distinzione tra client e server. In questo modello, qualsiasi oggetto nel sistema può fornire e utilizzare servizi da altri oggetti. Questo approccio offre una maggiore flessibilità e un'interazione più dinamica tra i componenti del sistema.

Architetture multiprocessore

Un'architettura multiprocessore è un modello di sistema distribuito che è composto da processi multipli che possono (ma non devono) essere eseguiti su diversi processori.

Questo approccio offre maggiore capacità di parallelismo, utile in applicazioni in tempo reale per garantire il rispetto di scadenze e migliorare l'affidabilità.

La distribuzione dei processi può essere preordinata o può essere gestita da un dispatcher.

Architetture client-server

Le architetture client-server suddividono le applicazioni in server (fornitori di servizi) e client (utilizzatori di servizi).

I client conoscono i server, ma non è necessario che i server conoscano i client.

Client e server sono processi logici, e la mappatura dei processori ai processi può variare, ad esempio, un server può essere eseguito su più macchine o più server possono essere ospitati su una singola macchina. Questa architettura è comune nelle applicazioni distribuite e in rete.

Architettura dell'applicazione a strati

L'architettura a strati è un approccio comune nella progettazione dei sistemi software. Questo approccio organizza il sistema in diversi livelli o strati, ognuno con un ruolo specifico nel funzionamento dell'applicazione.

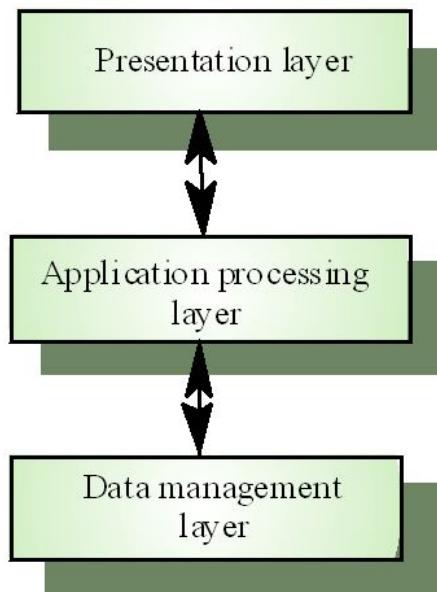
Il **livello di presentazione** si occupa della presentazione dei risultati ai utenti e della raccolta degli input. È il punto di interazione diretta con gli utenti, ed è cruciale che sia progettato in modo intuitivo e user-friendly per garantire un'esperienza utente positiva.

Il **livello di elaborazione delle applicazioni**, noto anche come logica di business, rappresenta il cuore dell'applicazione. Questo strato gestisce le funzionalità specifiche dell'applicazione, ad esempio, in un sistema bancario, le operazioni bancarie come l'apertura di un conto o la chiusura di un conto. Qui si trova la logica che elabora i dati, esegue calcoli e gestisce il flusso delle operazioni.

Il **livello di gestione dei dati** è responsabile della gestione dei database di sistema. In questo strato, i dati necessari per il funzionamento dell'applicazione vengono memorizzati e recuperati. La progettazione dei database e la gestione delle query rientrano in questo livello.

L'architettura a strati è un approccio modulare che permette di separare le diverse componenti dell'applicazione. Questo semplifica lo sviluppo, il mantenimento e l'aggiornamento del software poiché ciascun livello può essere modificato o sostituito senza influire negativamente sugli altri. Questo approccio è ampiamente utilizzato nell'ingegneria del software per garantire una struttura organizzata e scalabile delle applicazioni.

Application layers



©Ian Sommerville 2000

Software Engineering, 6th edition. Chapter 10

Slide 60

Modelli di client leggero e client pesante

In un modello di client leggero, tutta l'elaborazione dell'applicazione e la gestione dei dati vengono effettuate sul server. Il client è responsabile solo dell'esecuzione del software di presentazione.

Nel modello di client pesante, il server è responsabile solo della gestione dei dati. Il software sul client implementa la logica dell'applicazione e le interazioni con l'utente del sistema.

Modello client leggero

Il modello client leggero (thin-client) è spesso utilizzato quando i sistemi legacy vengono migrati verso architetture server-client. In questo caso, il sistema legacy funge da server a sé stante con un'interfaccia grafica implementata su un client. Tuttavia, uno dei principali svantaggi di questo modello è che comporta un carico di elaborazione pesante sia sul server che sulla rete.

Modello client pesante

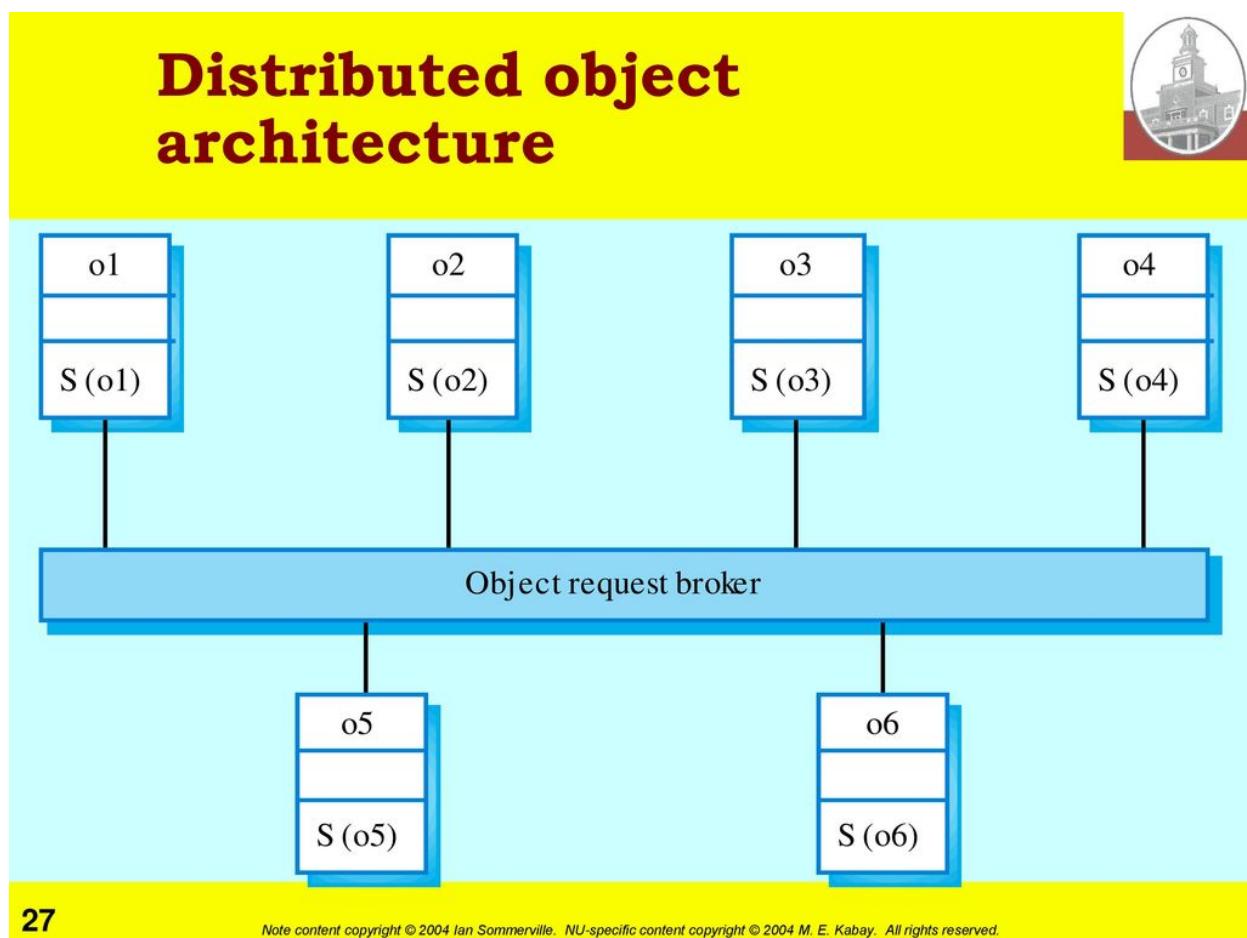
Nel modello cliente pesante (fat-client), una maggiore elaborazione viene delegata al client poiché l'elaborazione dell'applicazione viene eseguita localmente. Questo modello è particolarmente adatto per i nuovi sistemi client-server (C/S) in cui le capacità del sistema client sono note in anticipo. Tuttavia, il modello cliente grasso è più complesso rispetto al modello thin client, soprattutto per la gestione. Le nuove versioni dell'applicazione devono essere installate su tutti i client.

Architettura a tre livelli

In un'architettura a tre livelli, ciascuno dei livelli di architettura dell'applicazione può eseguire su un processore separato. Questo approccio consente migliori prestazioni rispetto a un approccio di client leggero ed è più semplice da gestire rispetto a un approccio di client pesante. Inoltre, questa architettura è altamente scalabile, poiché è possibile aggiungere server aggiuntivi per far fronte all'aumento delle richieste.

Architettura degli oggetti distribuiti

In un'architettura di oggetti distribuiti, non c'è distinzione tra client e server. Ogni entità distribuibile è considerata un oggetto che fornisce servizi ad altri oggetti e riceve servizi da altri oggetti. La comunicazione tra questi oggetti avviene attraverso un sistema middleware chiamato object request broker. Tuttavia, le architetture di oggetti distribuiti sono più complesse da progettare rispetto ai sistemi client/server tradizionali.



Vantaggi dell'architettura degli oggetti distribuiti

I vantaggi di un'architettura degli oggetti distribuiti includono la capacità di ritardare le decisioni su dove e come dovrebbero essere forniti i servizi, rendendo il sistema altamente aperto e flessibile. Inoltre, questa architettura è altamente scalabile e consente la riconfigurazione dinamica del sistema con oggetti che possono migrare attraverso la rete quando necessario.

CORBA

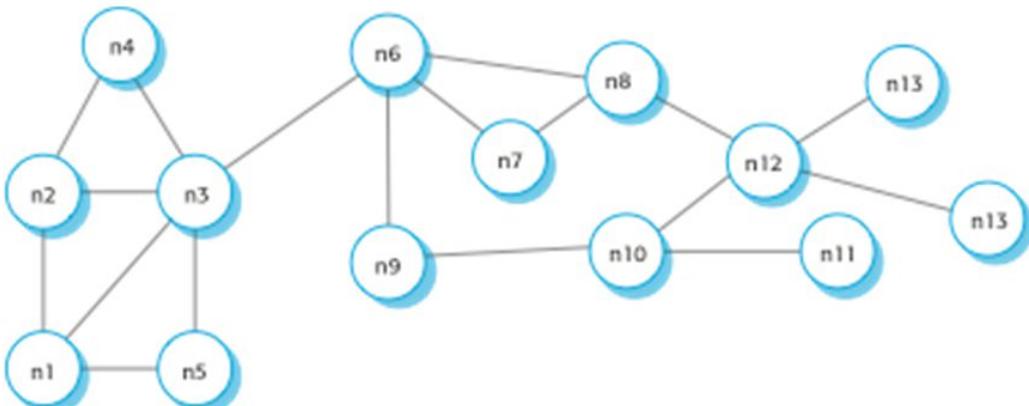
CORBA (Common Object Request Broker Architecture) è uno standard internazionale per un Object Request Broker (ORB), che è un middleware utilizzato per gestire le comunicazioni tra oggetti distribuiti. CORBA è essenziale per il calcolo distribuito a due livelli. A livello di comunicazione logica, il middleware consente agli oggetti su computer diversi di scambiare dati e informazioni di controllo. A livello di componenti, CORBA fornisce una base per lo sviluppo di componenti compatibili e ha stabilito standard di componenti CORBA.

Architetture peer-to-peer

I sistemi peer-to-peer (p2p) rappresentano una modalità di organizzazione decentralizzata, in cui ogni nodo di una rete può contribuire eseguendo calcoli e condividendo risorse. Questo tipo di architettura mira a sfruttare la potenza di calcolo e la capacità di archiviazione di un gran numero di computer connessi in rete. Inizialmente, la maggior parte dei sistemi peer-to-peer era orientata verso utilizzi personali, come condivisione di file o risorse tra utenti, ma negli ultimi anni, c'è stato un notevole aumento nell'uso commerciale di questa tecnologia.

Architettura peer-to-peer decentralizzata

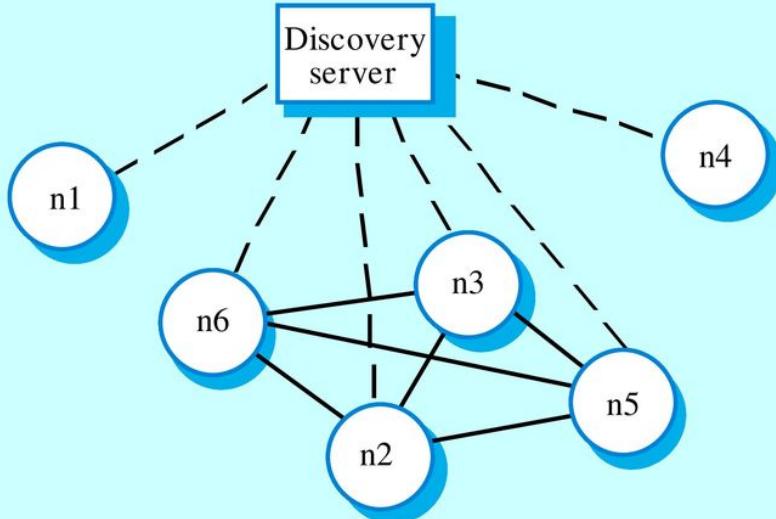
A decentralized p2p architecture



Nell'architettura peer-to-peer decentralizzata, non esiste un nodo centrale di controllo. Ogni nodo nella rete ha pari diritti e responsabilità. Questa modalità consente una maggiore resistenza ai guasti e un'ampia distribuzione della gestione delle risorse e dei calcoli. Un esempio ben noto di architettura decentralizzata è BitTorrent, un protocollo per la condivisione di file in cui ciascun peer può caricare e scaricare dati dagli altri peers senza la necessità di un server centrale.



Semi-centralized p2p architecture



46

Note content copyright © 2004 Ian Sommerville. NU-specific content copyright © 2004 M. E. Kabay. All rights reserved.

Nell'architettura peer-to-peer semi-centralizzata, anche se alcuni aspetti sono decentralizzati, esiste un certo grado di controllo centralizzato. Ad esempio, una rete peer-to-peer potrebbe avere nodi che svolgono il ruolo di "supernodi" o "tracker" per facilitare la ricerca e il reindirizzamento dei peer verso i contenuti richiesti. Questo approccio è spesso utilizzato per migliorare l'efficienza nella ricerca dei contenuti, ma può comportare una maggiore vulnerabilità a problemi tecnici o legali.

Architetture orientate ai servizi

Le architetture orientate ai servizi sono basate sulla nozione di servizi che possono essere resi disponibili ed accessibili attraverso il web. Un servizio web è un'implementazione standard per esporre componenti riutilizzabili su Internet. Ad esempio, un servizio di presentazione delle dichiarazioni fiscali potrebbe aiutare gli utenti nella compilazione delle dichiarazioni fiscali e nel loro invio alle autorità fiscali.

Standard dei servizi

I servizi web sono basati su standard concordati che utilizzano XML come linguaggio di scambio dei dati e possono essere eseguiti su qualsiasi piattaforma e scritti in qualsiasi linguaggio di programmazione. Alcuni dei principali standard nell'ambito dei servizi web includono:

- **SOAP (Simple Object Access Protocol)**: Un protocollo basato su XML per la comunicazione tra applicazioni su diverse piattaforme.
- **WSDL (Web Services Description Language)**: Un linguaggio standard per descrivere i servizi web, consentendo agli utenti di comprenderne la funzionalità e gli input/output.
- **UDDI (Universal Description, Discovery and Integration)**: Un protocollo e un registro per scoprire servizi web e per la loro integrazione nelle applicazioni.

Questi standard contribuiscono a garantire l'interoperabilità tra servizi web, consentendo alle applicazioni di comunicare e collaborare in modo efficace.

Progettazione orientata agli oggetti

Sviluppo orientato agli oggetti

Lo sviluppo orientato agli oggetti (OOD) è una pratica fondamentale nella programmazione e nell'ingegneria del software che si basa su concetti e principi legati all'orientamento agli oggetti.

Analisi, Progettazione e Programmazione Orientate agli Oggetti

L'analisi, la progettazione e la programmazione orientate agli oggetti sono tre fasi distinte ma strettamente collegate nello sviluppo del software.

- **OOA (Analisi Orientata agli Oggetti)**: Questa fase comporta lo sviluppo di un modello di oggetti del dominio dell'applicazione. In altre parole, si tratta di comprendere e identificare le principali entità e oggetti rilevanti nel contesto dell'applicazione software.
- **OOD (Progettazione Orientata agli Oggetti)**: In questa fase, si sviluppa un modello di sistema orientato agli oggetti che mira a implementare i requisiti emersi dall'analisi. L'obiettivo è definire le classi, le relazioni e le interfacce necessarie per realizzare l'applicazione.
- **OOP (Programmazione Orientata agli Oggetti)**: Qui, il modello OOD viene effettivamente tradotto in codice utilizzando un linguaggio di programmazione orientato agli oggetti, come Java o C++. Questo passaggio coinvolge la creazione delle istanze delle classi e l'implementazione dei comportamenti degli oggetti.

Caratteristiche dell'OOD

L'OOD è caratterizzato da alcune importanti proprietà e pratiche:

- **Gli Oggetti come Astrazioni:** Gli oggetti nell'OOD sono astrazioni di entità reali o del sistema, e sono progettati per essere autonomi, cioè contengono informazioni sul loro stato e sulla loro rappresentazione.
- **Espressione della Funzionalità:** La funzionalità dell'applicazione è espressa in termini di servizi degli oggetti. Questo significa che gli oggetti svolgono compiti specifici e comunicano tra loro attraverso il passaggio di messaggi.
- **Eliminazione delle Aree di Condivisione dei Dati:** Nell'OOD, le aree di condivisione dei dati tra oggetti vengono eliminate o minimizzate. Questo riduce il potenziale per errori dovuti a modifiche accidentali dei dati.
- **Comunicazione tramite Passaggio di Messaggi:** Gli oggetti comunicano tra loro attraverso il passaggio di messaggi, piuttosto che condividere dati direttamente. Questo migliora l'incapsulamento e l'indipendenza degli oggetti.
- **Possibilità di Distribuzione e Parallelismo:** Gli oggetti possono essere distribuiti su diverse macchine o eseguiti in modo sequenziale o parallelo, il che offre maggiore flessibilità nella progettazione dei sistemi.

Vantaggi dell'OOD

L'utilizzo dell'OOD comporta diversi vantaggi:

- **Manutenzione più Semplice:** Gli oggetti sono entità autonome, il che semplifica la manutenzione poiché le modifiche a un oggetto possono essere apportate senza influenzare altri componenti del sistema.
- **Riutilizzabilità:** Gli oggetti sono componenti potenzialmente riutilizzabili. Questo significa che le classi e i metodi sviluppati in un progetto possono essere applicati anche in altri contesti, riducendo la duplicazione del codice.
- **Metafora Naturale:** Spesso, c'è una corrispondenza naturale tra le entità del mondo reale e gli oggetti del sistema. Questa metafora naturale rende più intuitivo comprendere e progettare il software.

In conclusione, l'OOD è un approccio fondamentale nell'ingegneria del software che consente la progettazione e l'implementazione di sistemi software in modo modulare, flessibile e manutenibile, utilizzando oggetti come mattoni fondamentali. Questo porta a una migliore comprensione del dominio dell'applicazione, una progettazione efficiente e un codice più robusto e riutilizzabile.

Linguaggio di modellizzazione unificato (UML - Unified Model Language)

Negli anni '80 e '90, sono state proposte diverse notazioni per descrivere progetti orientati agli oggetti. Queste notazioni hanno cercato di fornire un modo strutturato per rappresentare e comunicare i concetti associati all'analisi e alla progettazione dei sistemi orientati agli oggetti. Tra queste notazioni, una delle più influenti e ampiamente adottate è l'**Unified Modeling Language** (UML).

Processo di progettazione orientato agli oggetti

Il processo di progettazione orientato agli oggetti coinvolge lo sviluppo di numerosi modelli di sistema diversi che servono a rappresentare le diverse parti del sistema e le relazioni tra di esse.

Questi modelli richiedono molto sforzo per lo sviluppo e la manutenzione, e per i sistemi piccoli potrebbero non essere convenienti.

Tuttavia, i modelli di progettazione offrono un meccanismo di comunicazione essenziale in progetti di grandi dimensioni, sviluppati da gruppi diversi. Per queste applicazioni complesse, i modelli di progettazione aiutano a garantire che tutti i membri del team abbiano una comprensione chiara del sistema e dei suoi componenti.

Fasi del Processo

Nel processo di ingegneria dei requisiti, ci sono diverse fasi chiave da affrontare per garantire il successo dello sviluppo del sistema. Queste fasi non sono vincolate a un processo proprietario specifico, come il Rational Unified Process (RUP), ma sono applicabili in generale a diverse metodologie di sviluppo. Le principali fasi del processo includono:

- 1. Definizione del Contesto e Modalità d'Uso del Sistema:** Questa fase mira a sviluppare una comprensione delle relazioni tra il software in fase di progettazione e il suo ambiente esterno. È essenziale per stabilire i confini del sistema e identificare i sistemi esterni con cui interagirà.
- 2. Progettazione dell'Architettura del Sistema:** In questa fase, si sviluppa un'architettura di alto livello per il sistema. Questa architettura definisce la struttura globale del sistema e come le sue parti interagiranno tra loro.
- 3. Identificazione degli Oggetti Principali del Sistema:** Si tratta di individuare gli oggetti chiave che costituiranno il sistema. Questi oggetti sono spesso basati sulle entità del dominio o sulle funzionalità rilevanti.
- 4. Sviluppo di Modelli di Progettazione:** Questa fase implica la creazione di modelli dettagliati che rappresentano il comportamento dei singoli oggetti o delle classi di oggetti nel sistema. Questi modelli contribuiscono a definire come gli oggetti risponderanno agli stimoli esterni.
- 5. Specificazione delle Interfacce degli Oggetti:** In questa fase, vengono definite le interfacce e le relazioni tra gli oggetti. Questo contribuisce a stabilire come gli oggetti collaboreranno all'interno del sistema.

Contesto del sistema e modello di utilizzo

Per comprendere appieno come un sistema software in fase di progettazione si relazioni con il suo ambiente circostante, è essenziale considerare due aspetti fondamentali: il "Contesto del Sistema" e il "Modello di Utilizzo del Sistema".

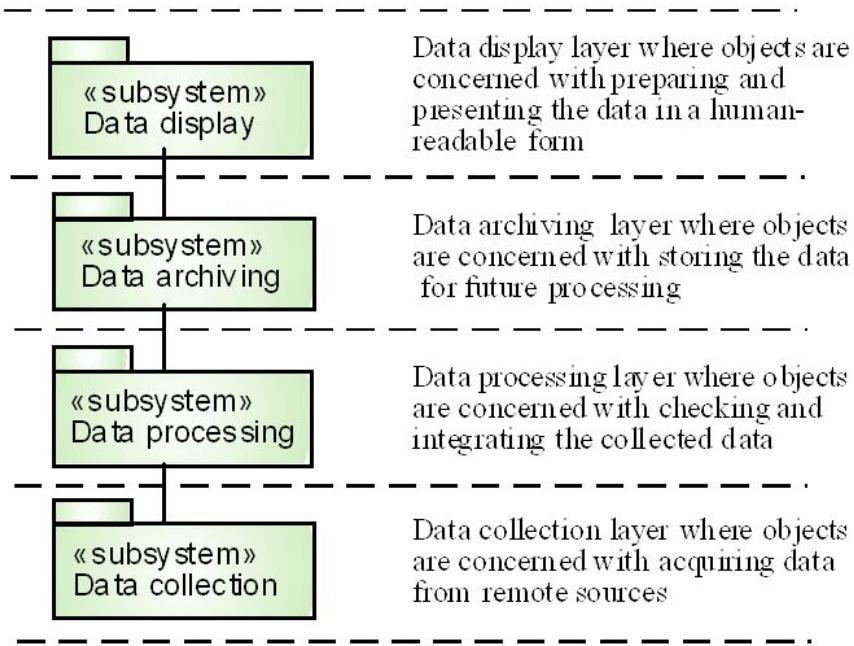
Contesto del sistema

Il "contesto del sistema" è un modello statico che mira a descrivere i rapporti tra il sistema software che stiamo progettando e altri sistemi che coesistono nell'ambiente circostante. Questo modello di contesto del sistema rappresenta una panoramica visiva delle interazioni tra il sistema che stiamo sviluppando e gli altri componenti o sottosistemi con cui deve interagire. Solitamente, utilizziamo un modello a livello di sottosistema per mostrare chiaramente come il nostro sistema si integra con il contesto circostante.

Modello di utilizzo

Il "modello di utilizzo del sistema" è invece un modello dinamico che evidenzia come il nostro sistema interagirà con l'ambiente esterno. Questo tipo di modello si concentra sugli aspetti operativi e dinamici delle interazioni. In genere, i "casi d'uso" vengono utilizzati per rappresentare queste interazioni. Ogni "caso d'uso" rappresenta uno specifico scenario o flusso di lavoro in cui il sistema viene utilizzato, indicando chiaramente gli attori coinvolti e le azioni che si verificano. Questi casi d'uso aiutano a catturare in modo dettagliato come gli utenti e altri sistemi interagiranno con il nostro sistema software.

Layered architecture



Slide 23

L'architettura a strati è un approccio comune utilizzato nella progettazione dei sistemi software. Questo approccio organizza un sistema in strati o livelli distinti, ognuno dei quali ha un compito specifico. Ecco una panoramica dei quattro principali strati dell'architettura a strati:

Data Collection

Questo strato rappresenta il livello più basso dell'architettura a strati ed è responsabile della raccolta dei dati da diverse fonti. Queste fonti possono includere sensori, dispositivi di acquisizione dati, input utente e altro ancora. Il compito principale di questo strato è acquisire i dati grezzi e prepararli per essere elaborati nei livelli superiori.

Data Processing

Il secondo strato è il livello di elaborazione dei dati. Qui, i dati grezzi raccolti dal livello di raccolta vengono elaborati e trasformati in una forma più utile. Questo strato può coinvolgere calcoli, filtraggio, normalizzazione e altre operazioni di manipolazione dei dati. L'obiettivo è preparare i dati per l'analisi o l'utilizzo in strati superiori.

Data Archiving

Il terzo strato, noto come livello di archiviazione dei dati, gestisce la conservazione a lungo termine dei dati. I dati elaborati e utili vengono archiviati in modo da poter essere recuperati in futuro, se necessario. Questo strato è responsabile della gestione dei database o dei sistemi di archiviazione dati a lungo termine. Garantisce che i dati siano disponibili e conservati in modo sicuro per scopi storici o di conformità.

Data Display

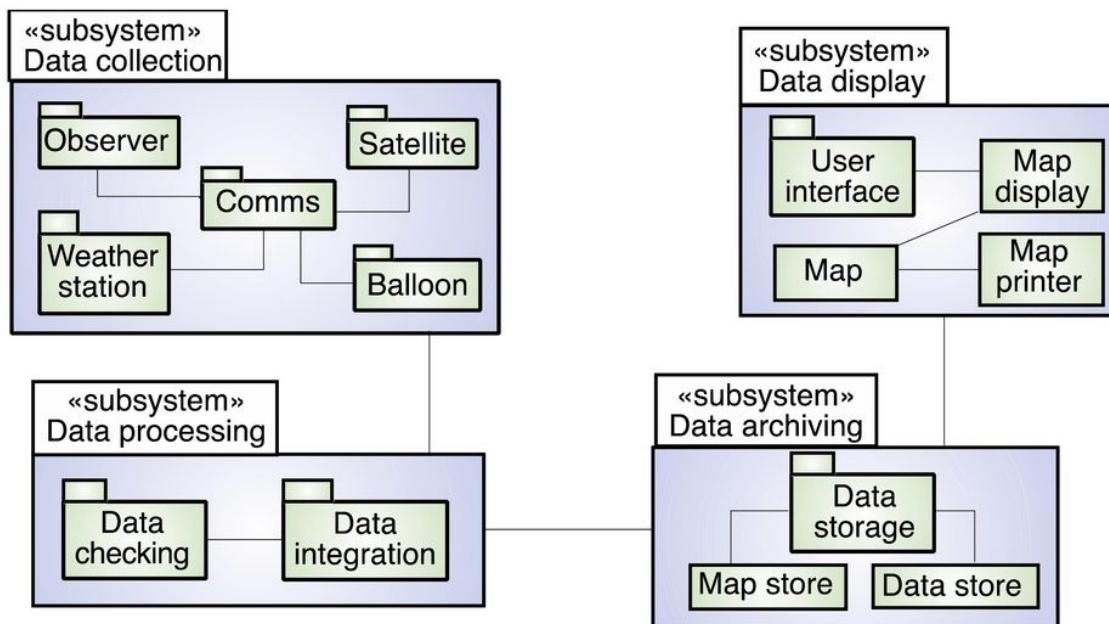
Il livello superiore è il livello di visualizzazione dei dati, che è responsabile della presentazione dei dati agli utenti o ad altre parti interessate. Questo strato può coinvolgere la creazione di interfacce utente, report, dashboard o altre forme di visualizzazione dei dati. L'obiettivo principale è consentire agli utenti di comprendere e interagire con i dati in modo significativo.

L'architettura a strati offre numerosi vantaggi, tra cui la separazione delle responsabilità, la facilità di manutenzione, la scalabilità e la riutilizzabilità dei componenti. Ogni strato ha un compito specifico e interagisce con gli strati adiacenti in modo ben definito. Questa struttura chiara rende più semplice la progettazione, lo sviluppo e la gestione dei sistemi software complessi.

Tieni presente che l'architettura a strati può variare in base alle esigenze specifiche del sistema e può includere ulteriori sottolivelli o componenti. La chiave è la separazione delle responsabilità e la definizione di interfacce chiare tra gli strati.

Sottosistemi in un sistema di mappatura metereologico

Subsystems in the weather mapping system



C-S 446/546

6

Un sistema di mappatura meteorologica, utilizzato per monitorare, prevedere e visualizzare le condizioni meteorologiche, di solito è composto da vari sottosistemi che lavorano insieme per raccogliere dati, elaborare informazioni e fornire previsioni accurate legate al meteo. Questi sottosistemi possono variare a seconda della complessità e dello scopo del sistema di mappatura meteorologica. Ecco alcuni comuni sottosistemi presenti in tali sistemi:

1. Sottosistema di Raccolta Dati:

- **Sensori Meteorologici:** Questo sottosistema include vari tipi di sensori, come anemometri, barometri, termometri e pluviometri, che raccolgono dati su temperatura, umidità, velocità e direzione del vento, pressione atmosferica e precipitazioni.
- **Sistemi Radar:** La tecnologia radar viene utilizzata per rilevare precipitazioni, movimenti delle tempeste ed eventi meteorologici gravi.
- **Satelliti Meteorologici:** I satelliti in orbita catturano immagini, dati di temperatura e altre informazioni atmosferiche per fornire una prospettiva globale.
- **Palloncini Meteorologici:** Questi strumenti sono dotati di sensori per misurare temperatura, umidità e pressione a diverse altitudini nell'atmosfera.

2. Sottosistema di Trasmissione e Ricezione Dati:

- **Infrastruttura di Comunicazione:** I dati meteorologici raccolti da sensori, satelliti e altre fonti vengono trasmessi ai centri dati centrali tramite vari metodi di comunicazione.
- **Ricezione e Elaborazione Dati:** I centri dati ricevono e elaborano le informazioni in ingresso, garantendo la loro qualità e integrità.

3. Sottosistema di Elaborazione e Analisi Dati:

- **Modelli Meteorologici:** Complessi modelli informatici utilizzano i dati raccolti per prevedere e simulare i modelli meteorologici. I modelli di previsione meteorologica numerica (NWP) sono comunemente utilizzati per questo scopo.
- **Assimilazione Dati:** Questo processo integra dati osservazionali in tempo reale nei modelli meteorologici per migliorare l'accuratezza.
- **Analisi Dati Meteorologici:** I meteorologi analizzano i dati per creare previsioni meteorologiche, monitorare eventi meteorologici gravi e emettere avvisi.

4. Sottosistema di Visualizzazione e Display Meteorologico:

- **Mappe Meteorologiche:** Gli strumenti di visualizzazione creano mappe meteorologiche che mostrano le condizioni attuali, le previsioni e gli avvisi.
- **Interfacce Utente Grafiche (GUI):** Interfacce interattive consentono ai meteorologi e al pubblico di visualizzare e interagire con i dati meteorologici.
- **Applicazioni Web e Mobile:** Le informazioni meteorologiche sono rese accessibili al pubblico attraverso siti web e app mobili.

5. Sottosistema di Monitoraggio e Avviso Eventi Meteorologici Gravi:

- **Rilevazione Eventi Meteorologici Gravi:** Strumenti e algoritmi specializzati monitorano le condizioni favorevoli a eventi meteorologici gravi, come tornado e uragani.
- **Sistemi di Avviso:** Sistemi automatizzati generano e diffondono avvisi meteorologici al pubblico, ai soccorritori e alle agenzie governative.

6. Sottosistema di Archiviazione e Recupero Dati Storici:

- **Basi di Dati Archiviate:** Questo sottosistema archivia dati meteorologici storici per riferimento, ricerca e analisi delle tendenze.
- **Interfacce di Recupero Dati:** Gli utenti possono accedere a dati meteorologici passati per varie applicazioni, inclusi studi climatici.

7. Sistema di Supporto alle Decisioni (DSS):

- **Supporto alle Decisioni Meteorologiche:** Questo sottosistema fornisce informazioni meteorologiche in tempo reale ai responsabili delle decisioni per sostenere la pianificazione e gli sforzi di risposta, specialmente in situazioni critiche come le catastrofi naturali.

8. Integrazione del Sistema di Informazioni Geografiche (GIS):

- **Mappatura Geospaziale:** Integra i dati meteorologici con le informazioni geografiche per creare rappresentazioni spaziali dei modelli meteorologici e dei loro impatti.

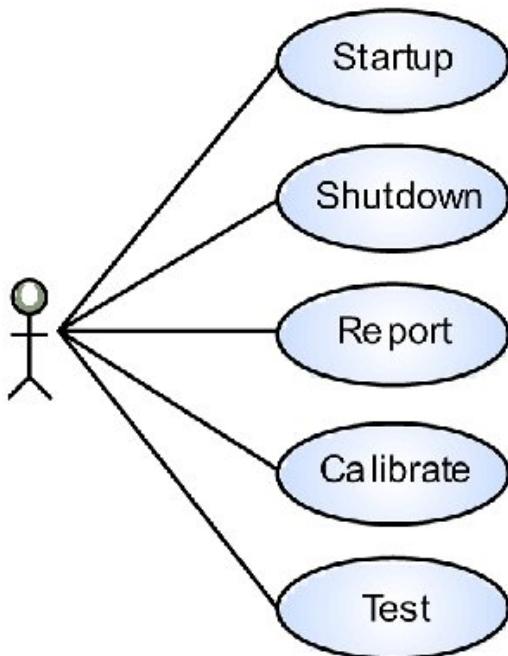
9. Feedback degli Utenti e Assicurazione della Qualità dei Dati:

- **Controllo Qualità:** Garantisce l'accuratezza e l'affidabilità dei dati raccolti e diffusi.
- **Feedback degli Utenti e Segnalazione:** Consente agli utenti di fornire feedback sull'accuratezza e l'utilità delle informazioni meteorologiche.

Questi sottosistemi lavorano insieme per creare un sistema di mappatura meteorologica completo che fornisce informazioni essenziali sia per i professionisti meteorologi che per il pubblico in generale.

Casi d'uso per una stazione metereologica

Use-cases for the weather station



1. Startup (Avvio):

- Attivazione del sistema: L'utente avvia la stazione meteorologica.
- Inizializzazione dei sensori: Il sistema avvia e inizializza i sensori per la raccolta dei dati meteorologici.
- Connessione alla rete: La stazione si connette alla rete per la trasmissione dei dati o per l'accesso ai dati meteorologici in remoto.

2. Shutdown (Arresto):

- Arresto del sistema: L'utente o il sistema stesso spegne la stazione meteorologica.
- Spegnimento dei sensori: I sensori vengono spenti in modo sicuro.
- Chiusura della connessione di rete: La stazione interrompe la connessione alla rete.

3. Report (Generazione di Rapporti):

- Raccolta dei dati: Il sistema raccoglie dati meteorologici da vari sensori.
- Elaborazione dei dati: I dati vengono elaborati per generare un rapporto completo.
- Creazione di report: Il sistema genera un rapporto meteorologico che include le informazioni sulla temperatura, l'umidità, la velocità del vento, ecc.
- Condivisione dei rapporti: Il rapporto può essere condiviso con gli utenti o con altri sistemi attraverso varie modalità, come la pubblicazione online o l'invio via e-mail.

4. Calibrate (Calibrazione):

- Calibrazione dei sensori: L'utente o il sistema avvia il processo di calibrazione per garantire l'accuratezza dei dati rilevati dai sensori.
- Regolazione dei parametri: Vengono regolati i parametri dei sensori, ad esempio per compensare eventuali deviazioni o errori.

5. Test (Test):

- Esecuzione di test diagnostici: L'utente o il sistema avvia test diagnostici per verificare il corretto funzionamento dei sensori e del sistema nel complesso.
- Registrazione dei risultati: I risultati dei test vengono registrati e possono essere visualizzati dall'utente o utilizzati per il monitoraggio dello stato della stazione.
- Notifica di problemi: Se i test rilevano anomalie o problemi, la stazione può generare notifiche o avvisi per l'utente o il personale tecnico responsabile della manutenzione.

Questi casi d'uso rappresentano alcune delle attività comuni associate a una stazione meteorologica e illustrano come questa possa essere avviata, spenta, utilizzata per generare rapporti meteorologici, calibrata e testata per garantire il corretto funzionamento e l'accuratezza dei dati.

Descrizione del caso d'uso

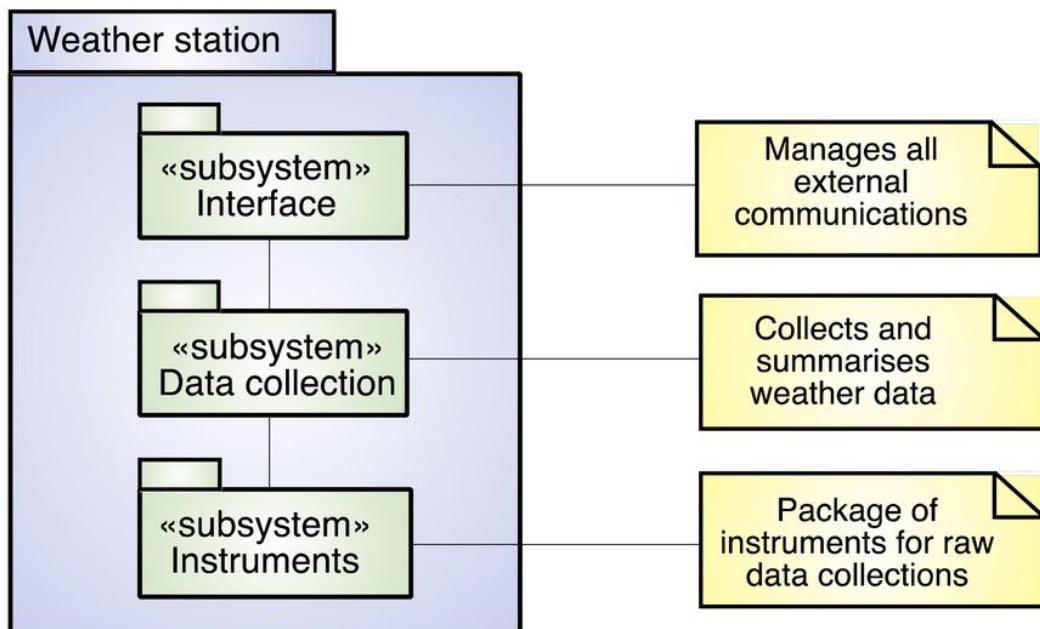
| | |
|------------|---|
| Sistema | Stazione metereologica |
| Caso d'uso | Report |
| Attori | Sistema di raccolta dati meteorologici, Stazione meteorologica |
| Dati | La stazione meteorologica invia un riepilogo dei dati meteorologici rilevati raccolti dagli strumenti nel periodo di raccolta al sistema di raccolta dati meteorologici. I dati inviati sono le temperature massime minime e medie del suolo e dell'aria, la pressione atmosferica massima, minima e media, la velocità del vento massima, minima e media, le precipitazioni totali e la direzione del vento campionati a intervalli di 5 minuti. |
| Stimolo | Il sistema di raccolta dei dati meteorologici stabilisce un collegamento modem con la stazione meteorologica e richiede la trasmissione dei dati. |
| Risposta | I dati riepilogati vengono inviati al sistema di raccolta dati meteorologici. |
| Commenti | Alle stazioni meteorologiche viene solitamente chiesto di effettuare il rapporto una volta all'ora, ma questa frequenza può differire da una stazione all'altra e può essere modificata in base alle esigenze future. |

Progettazione architetturale

Una volta comprese le interazioni tra il sistema e il suo ambiente, queste informazioni vengono utilizzate per progettare l'architettura del sistema. È importante sottolineare che in un modello architettonico ideale, ci dovrebbero essere di solito meno di sette entità. La progettazione architettonica svolge un ruolo cruciale nell'organizzare il sistema in modo da soddisfare i requisiti funzionali e non funzionali. La scelta dell'architettura giusta influenzera notevolmente il successo del progetto.

Architettura stazione metereologica

Weather station architecture



C-S 446/546

10

L'architettura di una stazione metereologica solitamente include tre componenti principali:

1. **Subsystem Interface:** Questo gestisce l'interazione con il mondo esterno e consente agli utenti di accedere ai dati metereologici.
2. **Subsystem Data Collection:** Questo è responsabile della raccolta, elaborazione e memorizzazione dei dati metereologici provenienti da sensori.
3. **Subsystem Instruments:** Questo include i sensori e gli strumenti utilizzati per raccogliere dati metereologici.

Identificazione degli Oggetti

L'identificazione degli oggetti (o delle classi di oggetti) rappresenta una delle sfide più complesse nella progettazione orientata agli oggetti. Non esiste una "formula magica" per condurre l'identificazione con successo. Invece, tale processo si fonda sull'abilità, sull'esperienza e sulla conoscenza specifica del dominio di applicazione da parte dei progettisti di sistemi. Inoltre, l'identificazione degli oggetti è un processo iterativo, che richiede spesso diverse iterazioni per essere completato in modo soddisfacente. È improbabile raggiungere una soluzione ottimale già al primo tentativo.

Approcci all'Identificazione

Ci sono vari approcci utilizzati per l'identificazione degli oggetti, ognuno dei quali può essere adatto a situazioni diverse:

1. **Approccio Grammaticale:** Questo metodo si basa su una descrizione del sistema in linguaggio naturale. Ad esempio, il metodo HoodOOD fa ampio uso di un approccio grammaticale per l'identificazione degli oggetti.
2. **Approccio Basato su Oggetti Tangibili:** Questo approccio si concentra sulla definizione degli oggetti basandosi su elementi tangibili e concreti presenti nel dominio dell'applicazione. Gli oggetti possono rappresentare oggetti fisici o astratti.
3. **Approccio Comportamentale:** Qui, l'identificazione degli oggetti avviene considerando quali oggetti partecipano a specifici comportamenti o processi all'interno del sistema. Ciò può includere il riconoscimento degli oggetti coinvolti in attività, funzioni o operazioni.
4. **Analisi basata su Scenari:** In questo approccio, vengono esaminati scenari o casi d'uso specifici dell'applicazione per identificare gli oggetti, gli attributi e i metodi necessari per soddisfare i requisiti. Questo può aiutare a individuare gli oggetti e le loro interazioni in situazioni reali di utilizzo.

Ogni approccio ha le proprie applicazioni e può essere utilizzato in combinazione, a seconda delle esigenze del progetto. Inoltre, l'identificazione degli oggetti potrebbe richiedere iterazioni e rifiniture mentre si sviluppa una comprensione più approfondita del sistema.

Modelli di progettazione

I modelli di progettazione aiutano a visualizzare e pianificare come gli oggetti e le classi di oggetti interagiscono tra loro, oltre a descrivere la struttura statica e le interazioni dinamiche all'interno del sistema.

Modelli Statici

I modelli statici sono focalizzati sulla rappresentazione della struttura statica del sistema. Questi modelli descrivono le classi di oggetti e le relazioni tra di esse. Alcuni esempi di modelli statici includono diagrammi delle classi UML (Unified Modeling Language) che mostrano le classi, gli attributi e le associazioni tra di esse. Questi modelli aiutano gli sviluppatori a comprendere la struttura del sistema e le relazioni tra le diverse entità.

Modelli Dinamici

I modelli dinamici, d'altra parte, si concentrano sulle interazioni e le dinamiche del sistema. Questi modelli descrivono come gli oggetti interagiscono tra di loro durante l'esecuzione del software. Diagrammi di sequenza UML e diagrammi di attività sono esempi di modelli dinamici che mostrano le sequenze di azioni e i flussi di controllo tra oggetti durante l'esecuzione del software. Questi modelli sono utili per comprendere il comportamento del sistema e le transizioni tra diversi stati.

Esempi di modelli di progettazione

Nel contesto dell'ingegneria del software, esistono diversi modelli di progettazione che aiutano gli sviluppatori a concepire, strutturare e comprendere l'architettura di un sistema software. Questi modelli svolgono un ruolo fondamentale nella pianificazione e nell'implementazione dei sistemi software. Alcuni dei modelli di progettazione più comuni includono:

Modelli di Sottosistemi

I modelli di sottosistemi sono utilizzati per mostrare i raggruppamenti logici di oggetti all'interno del sistema in sottosistemi coerenti. Questi sottosistemi rappresentano una suddivisione logica dell'applicazione in parti più gestibili e comprensibili. Questo approccio consente ai team di sviluppo di lavorare su parti specifiche del sistema in parallelo, semplificando la manutenzione e il debug.

Modelli di Sequenza

I modelli di sequenza descrivono la sequenza delle interazioni tra gli oggetti nel sistema. Questi diagrammi aiutano a visualizzare come gli oggetti collaborano e comunicano tra loro per eseguire determinate funzionalità del sistema. Questi modelli consentono di tracciare le chiamate tra oggetti e di comprendere come fluiscono i dati all'interno dell'applicazione.

Modelli di Macchine a Stati

I modelli di macchine a stati mostrano come gli oggetti nel sistema cambiano il loro stato in risposta a eventi specifici. Questi modelli sono utili per la progettazione di sistemi reattivi e comportamentali. Mostrano chiaramente come il sistema risponde agli stimoli esterni e come si evolve nel tempo.

Altri Modelli

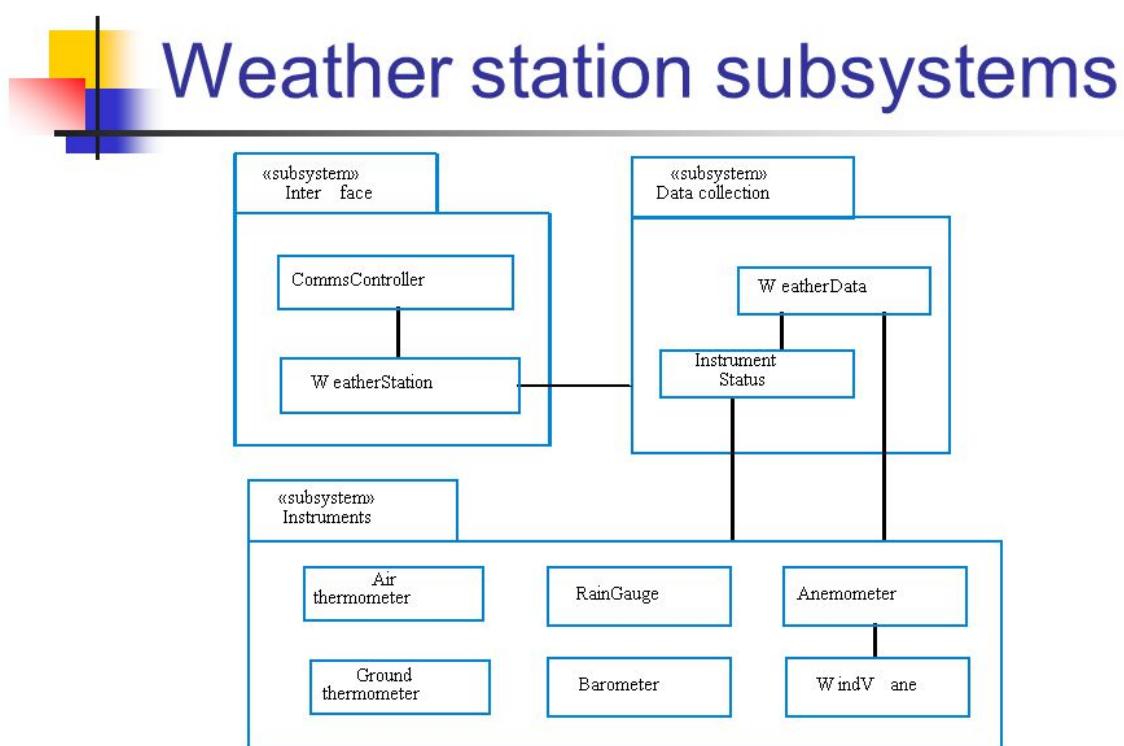
Oltre a questi modelli, ci sono altri modelli di progettazione utilizzati in base alle esigenze specifiche del progetto. Alcuni esempi includono i modelli di casi d'uso, che descrivono come gli utenti interagiscono con il sistema, i modelli di aggregazione che mostrano le relazioni tra gli oggetti e i modelli di generalizzazione che evidenziano la gerarchia di classi.

Modelli di sottosistemi

I modelli di sottosistemi servono a mostrare come la progettazione sia organizzata in gruppi di oggetti correlati logicamente. Nell'UML (Unified Modeling Language), questi sottosistemi vengono mostrati utilizzando i pacchetti, che rappresentano un costrutto di encapsulamento. Questo modello di organizzazione è di natura logica, e va sottolineato che l'effettiva organizzazione degli oggetti all'interno del sistema potrebbe differire da questa rappresentazione logica.

In altre parole, i pacchetti nei modelli UML fungono da contenitori logici per gli oggetti correlati all'interno del sistema, ma come questi oggetti vengono fisicamente implementati o organizzati nel codice o nell'architettura del sistema potrebbe variare. Il vantaggio di questa rappresentazione logica è che facilita la comprensione e la gestione delle relazioni e delle dipendenze tra i diversi componenti del sistema, contribuendo a garantire che la progettazione sia organizzata in modo coerente e che le responsabilità siano chiaramente definite.

Sottosistemi di una stazione metereologica



Modelli di sequenza

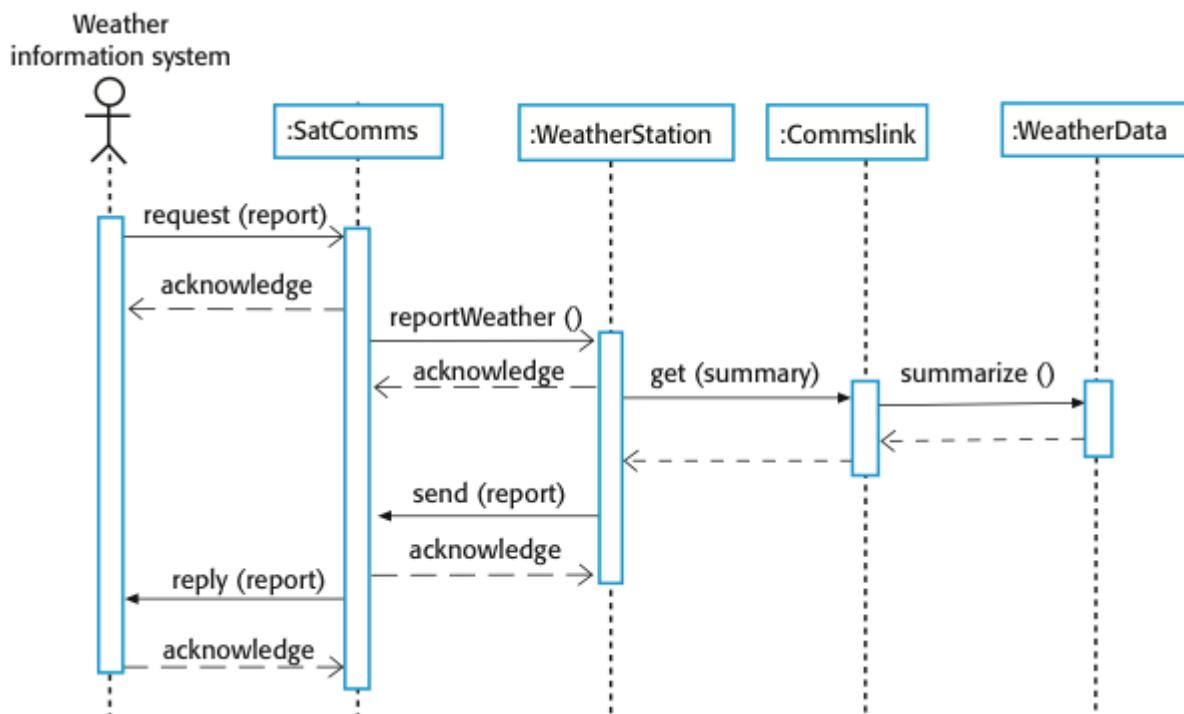
I modelli di sequenza rappresentano la sequenza delle interazioni degli oggetti all'interno di un sistema software. In questi modelli, gli oggetti sono disposti orizzontalmente nella parte superiore del diagramma, mentre il tempo è rappresentato verticalmente. Pertanto, la lettura dei modelli avviene dall'alto verso il basso.

Le interazioni tra gli oggetti sono rappresentate da frecce etichettate. Differenti stili di frecce possono essere utilizzati per rappresentare diversi tipi di interazioni. Ad esempio, le frecce possono indicare il flusso di controllo, la comunicazione di dati o altri tipi di interazione tra gli oggetti.

Ogni oggetto coinvolto nel sistema è tracciato verticalmente lungo una "linea di vita". All'interno di queste linee di vita, un sottile rettangolo può essere utilizzato per rappresentare il momento in cui l'oggetto è attivo o controlla il sistema. In altre parole, mostra il periodo in cui l'oggetto è coinvolto o esegue un'azione specifica all'interno del contesto dell'interazione.

Questi modelli di sequenza sono ampiamente utilizzati nell'analisi e nella progettazione del software per visualizzare e comprendere come gli oggetti interagiscono tra loro durante l'esecuzione di un sistema. Sono uno strumento utile per catturare la dinamica delle interazioni e delle comunicazioni tra gli oggetti all'interno di un sistema software.

Sequenza di raccolta dei dati



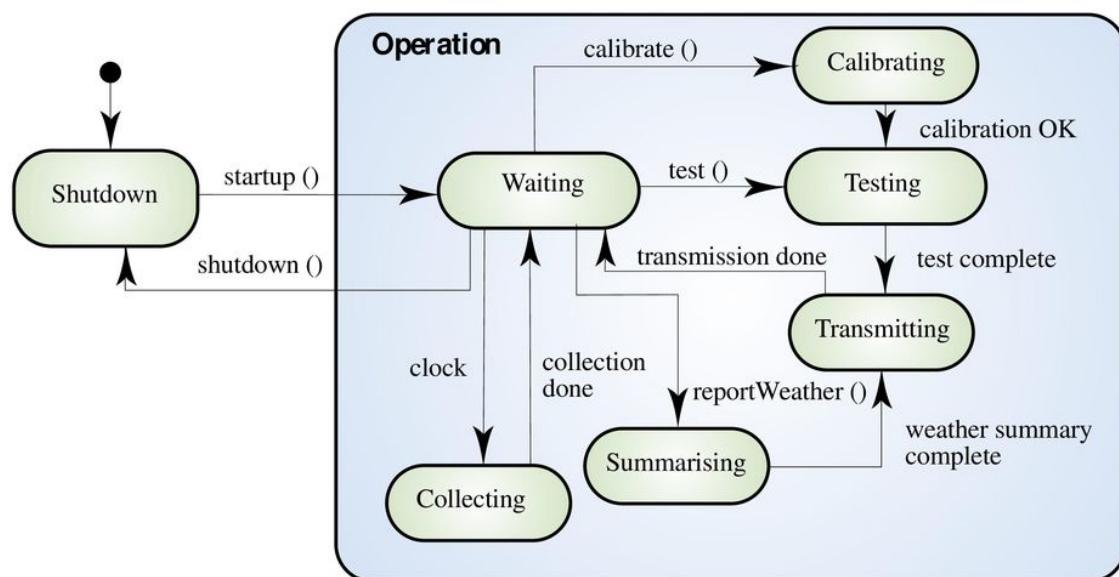
Diagrammi di stato

I diagrammi di stato mostrano lo stato corrente di un oggetto e come questo stato può cambiare in risposta a richieste specifiche o eventi. Essi possono includere:

1. La risposta a un messaggio: Ad esempio, un oggetto nello stato "Spegnimento" risponde al messaggio "Avvio", innescando una transizione verso uno stato di "Accensione".
2. Stato di attesa: Un oggetto può essere in uno stato di "attesa", indicando che è in attesa di ulteriori messaggi o eventi prima di compiere ulteriori azioni.
3. Transizioni di stato: Questi diagrammi mostrano come avvengono le transizioni tra uno stato e l'altro in risposta a eventi specifici.
4. Stati specifici: Possono includere stati speciali che rappresentano situazioni particolari, come uno stato di "calibrazione" in risposta a un comando.
5. Eventi di sistema: Alcune transizioni di stato possono essere innestate da eventi di sistema, come segnali di orologio.

Diagramma di stato della stazione meteorologica

Weather station state diagram



Specifiche delle interfacce degli oggetti

Nel processo di progettazione software, è fondamentale specificare adeguatamente le interfacce degli oggetti per consentire una progettazione parallela degli oggetti e dei componenti del sistema.

I progettisti dovrebbero evitare di concentrarsi troppo sulla rappresentazione dell'interfaccia stessa. Invece, l'attenzione dovrebbe essere rivolta principalmente all'interno dell'oggetto, dove la complessità dell'implementazione può essere nascosta agli altri componenti del sistema. Questo principio di astrazione è cruciale per garantire che gli oggetti siano modulari e che le loro interfacce rimangano stabili anche se l'implementazione interna cambia.

Un aspetto interessante delle interfacce degli oggetti è la loro capacità di consentire a un oggetto di avere più di un'interfaccia. Ciò significa che è possibile rappresentare diversi "punti di vista" sui metodi forniti dall'oggetto. Questo concetto di "punti di vista" offre una maggiore flessibilità nella progettazione, in quanto consente di esporre solo un sottoinsieme specifico delle funzionalità dell'oggetto a determinati componenti del sistema.

Per la specifica delle interfacce degli oggetti, esistono diverse metodologie. L'Unified Modeling Language (UML) è una delle più comuni e utilizza i diagrammi delle classi per rappresentare chiaramente l'interfaccia dell'oggetto. Tuttavia, questa specifica non è limitata all'UML, poiché molti linguaggi di programmazione, come Java, offrono strumenti e convenzioni per rappresentare in modo pratico queste interfacce direttamente nel codice sorgente.

Interfaccia di una stazione meteorologica

```
interface WeatherStation{
    public void WeatherStation();

    public void startup();
    public void startup(Instrumenti i);

    public void shutdown();
    public void shutdown(Instrument i);

    public void reportWeather();

    public void test();
    public void test(Instrument i);

    public void calibrate(Instrument i);

    public int getID();
} //WeatherStation
```

Sviluppo e mantenimento

Sviluppo software

Panorama dello sviluppo software

Il panorama dello sviluppo software comprende diverse metodologie e approcci, tra cui:

- **Lo Sviluppo Software Rapido (RAD)**
- **Lo Sviluppo Agile**
- **L'Ingegneria del Software basata sul Riutilizzo**
- **L'Ingegneria del Software basata sui Componenti**

Sviluppo Software Rapido (RAD)

Lo Sviluppo Software Rapido (RAD) si concentra sulla creazione rapida di prototipi e versioni funzionanti del software, consentendo risposte rapide ai requisiti dei clienti.

Le aziende possono essere disposte ad accettare software di qualità inferiore se è possibile una consegna rapida delle funzionalità essenziali

Requisiti

A causa dell'ambiente in evoluzione, è spesso impossibile giungere ad un insieme stabile e coerente di requisiti di sistema.

Un modello di sviluppo a cascata, che richiede una sequenza lineare di fasi, diventa spesso impraticabile, poiché richiede che tutti i requisiti siano definiti prima di procedere.

Pertanto, l'*approccio iterativo*, che consente aggiornamenti e adattamenti continui del software, diventa il metodo preferito per consegnare rapidamente software in un ambiente in evoluzione.

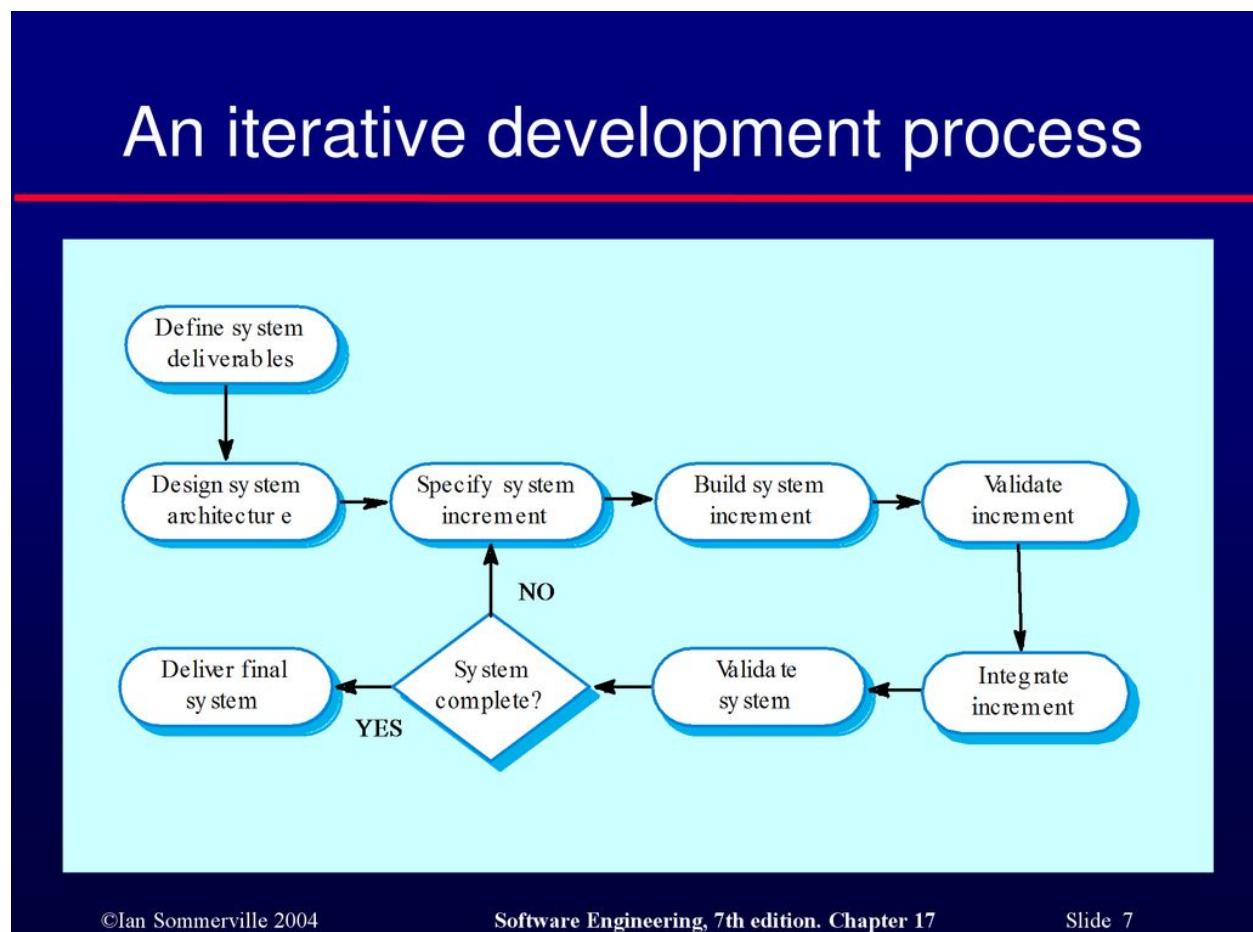
In questa modalità, lo sviluppo è suddiviso in cicli più brevi, consentendo di affrontare meglio i cambiamenti nei requisiti e di consegnare funzionalità utili in tempi più brevi.

Caratteristiche dei Processi RAD

Il processo di sviluppo RAD (Rapid Application Development) presenta alcune caratteristiche distinctive:

- Le fasi di specifica, progettazione e implementazione avvengono in modo concorrente, accelerando il ciclo di sviluppo.
- Si minimizza la documentazione dettagliata, concentrando gli sforzi direttamente sulla creazione del software.
- Il progetto è suddiviso in incrementi, piccoli pezzi funzionali del sistema, con il coinvolgimento attivo degli utenti finali nella valutazione e nel feedback.
- Le interfacce utente del sistema sono sviluppate in modo interattivo, con la collaborazione diretta tra gli sviluppatori e gli utenti.

Un processo di sviluppo iterativo



Processo di sviluppo incrementale

Vantaggi dello sviluppo incrementale

Lo sviluppo incrementale offre una serie di vantaggi che possono migliorare il processo di sviluppo del software:

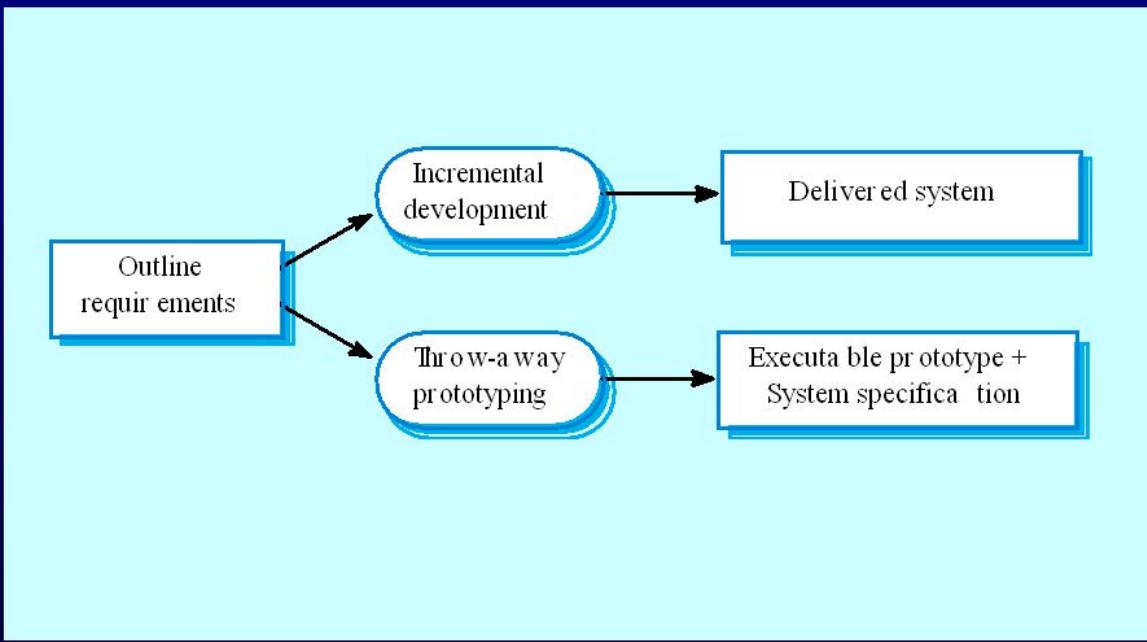
- **Consegna accelerata dei servizi ai clienti:** Questo approccio consente di consegnare in tempi brevi incrementi funzionali ai clienti. Ogni incremento fornisce le funzionalità di massima priorità, consentendo ai clienti di iniziare a utilizzare parti del sistema molto presto.
- **Coinvolgimento degli utenti con il sistema:** Gli utenti sono coinvolti fin dall'inizio, il che aumenta le probabilità che il sistema soddisfi i loro requisiti. Inoltre, gli utenti si sentono più coinvolti e hanno l'opportunità di fornire feedback regolare, contribuendo a orientare lo sviluppo verso le loro esigenze specifiche.

Problemi dello sviluppo incrementale

Nonostante i vantaggi, lo sviluppo incrementale può comportare alcune sfide che devono essere affrontate:

- **Problemi di gestione:** La valutazione del progresso e l'individuazione dei problemi possono risultare difficili. La mancanza di documentazione dettagliata che mostri cosa è stato fatto può rendere complicata la gestione del progetto.
- **Problemi contrattuali:** Nei progetti di sviluppo incrementale, è comune non avere una specifica completa all'inizio del progetto. Questo può portare a complicazioni nei contratti, richiedendo l'adozione di forme di contratto più flessibili che si adattino al processo incrementale.
- **Problemi di convalida:** La mancanza di una specifica completa può rendere difficile determinare cosa testare. Senza una definizione chiara delle funzionalità e dei requisiti, il processo di convalida può risultare complicato.
- **Problemi di manutenzione:** Il continuo cambiamento e l'aggiunta di nuove funzionalità possono avere un impatto sulla struttura del software. Questo rende il sistema più complesso da mantenere e può aumentare i costi di sviluppo nel lungo periodo.

Incremental development and prototyping



©Ian Sommerville 2004

Software Engineering, 7th edition. Chapter 17

Slide 19

Obiettivi Conflittuali

L'obiettivo dello sviluppo incrementale è quello di consegnare un sistema funzionante agli utenti finali. Questo approccio inizia con i requisiti meglio compresi e si concentra su incrementi successivi per fornire valore aggiunto.

L'obiettivo della prototipazione usa e getta è diverso, poiché mira a convalidare o derivare i requisiti del sistema. La prototipazione inizia spesso con requisiti meno compresi, creando prototipi iterativi per esplorare e raffinare i requisiti.

Prototipazione del Software

Un prototipo rappresenta una versione iniziale di un sistema che viene impiegata per dimostrare concetti e sperimentare opzioni di progettazione.

Un prototipo rappresenta una risorsa sia nella raccolta dei requisiti che nella loro convalida. I prototipi consentono agli stakeholder di vedere una rappresentazione tangibile del sistema in sviluppo, rendendo più semplice la comprensione dei requisiti e la valutazione delle funzionalità richieste.

I prototipi sono strumenti fondamentali per esplorare opzioni di design e sviluppare l'aspetto e il comportamento dell'interfaccia utente. Creare un prototipo di interfaccia utente consente ai

progettisti di testare diverse idee e interazioni utente prima di impegnarsi in uno sviluppo a lungo termine. Questo processo di progettazione iterativa porta a risultati più soddisfacenti e adattati alle esigenze degli utenti.

I prototipi possono anche svolgere un ruolo importante nel processo di testing. Possono essere utilizzati per eseguire test di confronto, consentendo agli sviluppatori e ai tester di confrontare il comportamento atteso del sistema con il funzionamento effettivo. In questo modo, è possibile identificare e risolvere tempestivamente i difetti.

Prototipi usa e getta

I prototipi dovrebbero essere scartati dopo lo sviluppo poiché non costituiscono una buona base per un sistema di produzione. Ciò è dovuto a diversi motivi.

- **Potrebbe essere impossibile adattare il sistema per soddisfare i requisiti non funzionali**

Questo è spesso il caso, poiché, i prototipi sono sviluppati rapidamente e con l'obiettivo principale di dimostrare il funzionamento delle funzionalità chiave. Non tengono conto di aspetti come le prestazioni, l'affidabilità o la sicurezza, che sono invece fondamentali per un sistema di produzione.

- **I prototipi di solito non sono documentati**

Questo significa che manca una documentazione dettagliata che sarebbe necessaria per sviluppare ulteriormente il sistema o per formare nuovi membri del team. Senza documentazione, il passaggio da un prototipo a un sistema di produzione può essere estremamente difficile.

- **La struttura del prototipo è di solito degradata attraverso cambiamenti rapidi**

Poiché i prototipi sono spesso soggetti a frequenti modifiche per soddisfare i requisiti in evoluzione, la loro struttura interna può diventare caotica e non adatta per una transizione senza problemi a un sistema di produzione.

- **Il prototipo probabilmente non rispetterà gli standard di qualità dell'organizzazione**

La qualità di un prototipo spesso viene sacrificata in nome della velocità di sviluppo e della dimostrazione delle funzionalità principali. Questo rende il prototipo inadatto per essere utilizzato come base per un sistema di produzione che deve rispettare rigorosi standard di qualità.

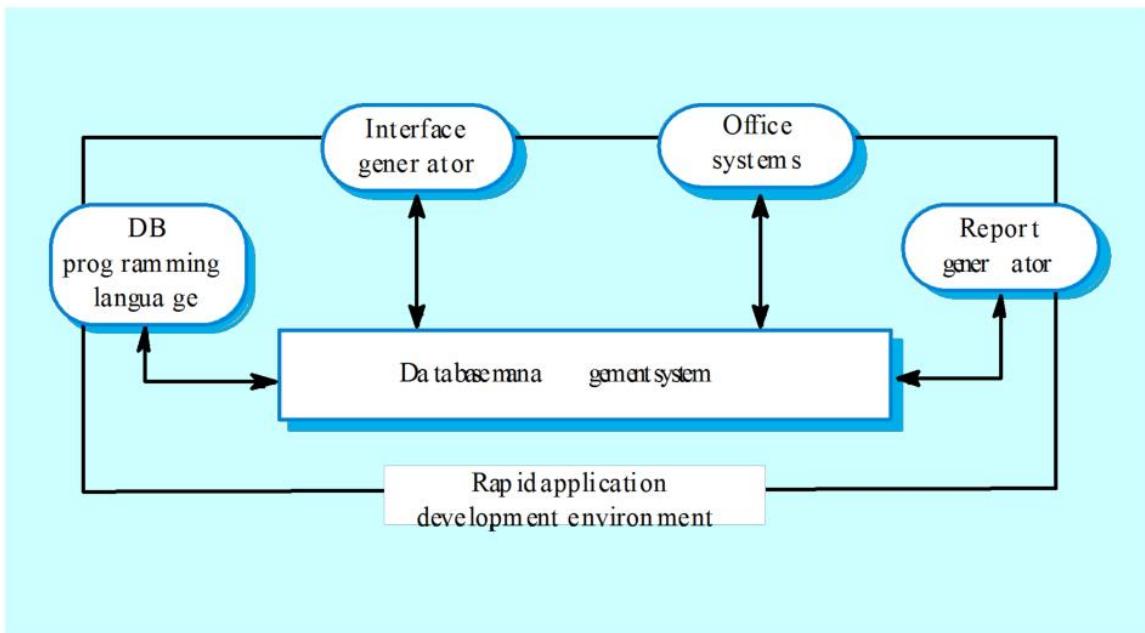
Ambienti RAD

L'ambiente RAD (Rapid Application Development) è stato progettato con l'obiettivo di sviluppare applicazioni aziendali ad alto contenuto di dati, mettendo una forte enfasi sulla programmazione e sulla presentazione delle informazioni da un database.

Strumenti:

- Linguaggio di programmazione del database
- Generatore di interfaccia (come SceneBuilder)
- Collegamenti a software per ufficio
- Generatori di report

A RAD environment



Riutilizzo COTS

Un approccio efficace allo sviluppo rapido è configurare e collegare sistemi COTS esistenti. Ad esempio, un sistema di gestione dei requisiti potrebbe essere realizzato utilizzando: un database per memorizzare i requisiti, un elaboratore di testi per catturare i requisiti e formattare i report, e un foglio di calcolo per la gestione della tracciabilità.

Questo approccio sfrutta l'uso di componenti COTS (Commercial Off-The-Shelf) per costruire un sistema senza dover sviluppare ogni parte da zero. In questo caso, il database, l'elaboratore di testi e il foglio di calcolo sono componenti software disponibili commercialmente, che possono essere facilmente integrati per soddisfare le esigenze del sistema di gestione dei requisiti.

L'utilizzo di soluzioni COTS offre diversi vantaggi, tra cui la riduzione dei tempi di sviluppo, la riduzione dei costi e la possibilità di sfruttare software già consolidati e ampiamente utilizzati.

Tuttavia, è importante valutare attentamente la compatibilità e l'integrazione tra le diverse componenti COTS per garantire che funzionino in modo sinergico e rispettino i requisiti specifici del sistema.

Metodi agili

I metodi agili sono stati sviluppati come risposta al disappunto causato dagli oneri associati ai metodi tradizionali di progettazione e sviluppo software. Questi metodi si distinguono per alcune caratteristiche chiave:

- Gli approcci agili mettono una forte enfasi sulla scrittura di codice funzionante piuttosto che dedicare una quantità eccessiva di tempo alla progettazione dettagliata. Questo non significa che la progettazione sia trascurata, ma piuttosto che si tratta di un processo più leggero e snodato, adattato alle esigenze del progetto.
- Un secondo aspetto distintivo è l'approccio iterativo seguito dagli sviluppatori. Gli sviluppatori lavorano su piccole porzioni di software alla volta, invece di cercare di produrre il sistema completo in un'unica iterazione. Questo approccio iterativo consente di consegnare rapidamente un software funzionante, il che è particolarmente utile quando i requisiti sono in continua evoluzione.
- Inoltre, i metodi agili si sforzano di adattarsi alle modifiche nei requisiti del progetto. Questo aspetto è cruciale, dato che i requisiti possono cambiare durante lo sviluppo, specialmente in ambienti aziendali in rapida evoluzione. La flessibilità è un pilastro fondamentale dei metodi agili.

I metodi agili sono spesso considerati più adatti per i sistemi aziendali di dimensioni moderate o piccole, o per i prodotti software destinati a personal computer (PC). Questa adattabilità ai progetti di dimensioni diverse è uno dei motivi per cui gli approcci agili hanno guadagnato popolarità.

Principi dei metodi agili

| Principio | Descrizione |
|-----------------------------------|---|
| Coinvolgimento del cliente | Il cliente dovrebbe essere strettamente coinvolto durante il processo di sviluppo. Il suo ruolo è fornire e stabilire le nuove specifiche di sistema e valutare le iterazioni del sistema. |
| Consegna incrementale | Il software è sviluppato a incrementi con il cliente che specifica i requisiti da includere in ciascun incremento. |
| Le persone non il processo | Le competenze del team di sviluppo dovrebbero essere riconosciute e sfruttate. Il team dovrebbe essere lasciato libero di sviluppare il proprio modo di lavorare senza processi prescrittivi. |
| Accogliere il cambiamento | Ci si aspetta che i requisiti del sistema cambino e si progetta il sistema in modo che possa accogliere questi cambiamenti. |
| Mantenere la semplicità | Concentrarsi sulla semplicità sia nel software in fase di sviluppo che nel processo di sviluppo utilizzato. Ovunque possibile, lavorare attivamente per eliminare la complessità dal sistema. |

Problemi con i Metodi Agili

Difficoltà nel Coinvolgere i Clienti

Un problema comune è la difficoltà nel mantenere l'interesse dei clienti coinvolti nel processo agile. Poiché questi metodi richiedono un coinvolgimento continuo e diretto dei clienti, potrebbe essere complicato garantire che i clienti siano costantemente disponibili per le discussioni e la valutazione dei progressi.

Adeguata Preparazione dei Membri del Team

Non tutti i membri del team possono essere adeguatamente preparati per l'intensità e l'interazione diretta richieste dai metodi agili. Alcuni sviluppatori potrebbero preferire approcci più strutturati o avere difficoltà a gestire i rapidi cambiamenti e la collaborazione stretta con i clienti.

Prioritizzazione in Presenza di Diversi Stakeholder

Quando ci sono diversi stakeholder coinvolti nel progetto, la prioritizzazione dei cambiamenti e delle nuove funzionalità può diventare un problema. Determinare quali richieste dovrebbero avere la precedenza può richiedere tempo e sforzi significativi.

Mantenimento della Semplicità

Uno dei principi fondamentali degli approcci agili è quello di mantenere il software semplice. Questo, tuttavia, richiede un lavoro aggiuntivo poiché è più facile aggiungere complessità che mantenerla ridotta. Mantenere la semplicità richiede una rigorosa attenzione all'essenziale e una costante valutazione dei requisiti e delle funzionalità.

Sfide Contrattuali

I contratti possono rappresentare una sfida nei metodi agili, come in altri approcci allo sviluppo iterativo. La natura fluida e adattabile dei progetti agili può entrare in conflitto con contratti che richiedono requisiti e tempi fissi. Questa sfida richiede la negoziazione di contratti più flessibili e la comprensione da parte dei clienti e dei fornitori di come funziona lo sviluppo agile.

Extreme Programming

Extreme Programming (XP) è uno dei metodi agili più noti e ampiamente utilizzati. Si caratterizza per un approccio estremo allo sviluppo iterativo, con alcune pratiche chiave che lo contraddistinguono.

XP promuove la costruzione di nuove versioni del software più volte al giorno. Questo significa che il team di sviluppo lavora costantemente all'evoluzione del prodotto, consentendo aggiornamenti frequenti per adattarsi alle esigenze in continua evoluzione.

Inoltre, XP stabilisce il rilascio di incrementi del software ai clienti ogni due settimane. Questo ritmo di rilascio regolare consente ai clienti di vedere i progressi e fornire feedback in modo tempestivo, facilitando la comunicazione e l'adattamento alle esigenze degli utenti.

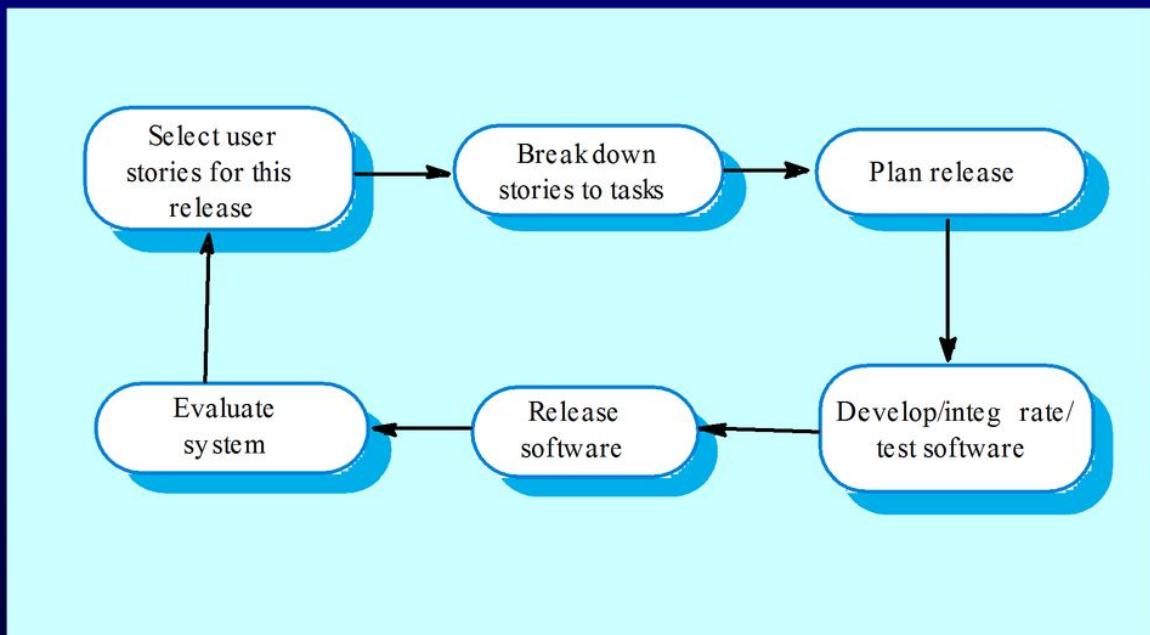
Un principio fondamentale di Extreme Programming è la rigorosa esecuzione dei test. Prima che una costruzione (build) possa essere accettata, tutti i test devono essere eseguiti con successo. Questo approccio mira a garantire che il software sia privo di errori e che le modifiche apportate non abbiano effetti indesiderati sul sistema.

Complessivamente, Extreme Programming promuove la collaborazione, la comunicazione continua e la qualità del software attraverso la sua pratica "estrema" di sviluppo iterativo.

Pratiche di Extreme Programming

| | |
|------------------------------------|---|
| Pianificazione incrementale | I requisiti sono registrati su Story Cards, e le card da includere in una release sono determinate dal tempo disponibile e dalla loro priorità relativa. Gli sviluppatori suddividono queste storie in "task" di sviluppo. |
| Rilasci piccoli | Si sviluppa prima il set minimo di funzionalità che fornisce valore aziendale. I rilasci del sistema sono frequenti e aggiungono incrementalmente funzionalità al primo rilascio. |
| Progettazione semplice | Viene effettuata una progettazione sufficiente per soddisfare i requisiti correnti e nient'altro. |
| Sviluppo orientato ai test | Viene utilizzato un framework di test automatizzati (es. JUnit) per scrivere i test per una nuova funzionalità prima che essa sia implementata. |
| Refactoring | Ci si aspetta che tutti gli sviluppatori refattorizzino continuamente il codice non appena vengono trovate possibili migliorie al codice. Ciò mantiene il codice semplice e manutenibile. |
| Programmazione in coppia | Gli sviluppatori lavorano in coppia, verificando il lavoro l'uno dell'altro e fornendo il supporto per fare sempre un buon lavoro (più facile trovare errori). |
| Proprietà collettiva | Le coppie di sviluppatori lavorano su tutte le aree del sistema, in modo che non si sviluppino isole di competenza, e tutti i membri del team possiedono e conoscono tutto il codice. Chiunque può modificare qualsiasi cosa. |
| Integrazione continua | Non appena il lavoro su un compito è completo, esso viene integrato nell'intero sistema. Dopo ogni integrazione, devono passare tutti i test unitari nel sistema. |
| Andatura sostenibile | Grandi quantità di straordinari non sono considerate accettabili, poiché l'effetto è spesso quello di ridurre la qualità del codice e la produttività. |
| Cliente in sede | Un rappresentante dell'utente finale del sistema (il Cliente) dovrebbe essere disponibile a tempo pieno per il team XP. In un processo di extreme programming, il cliente fa parte del team ed è responsabile di fornire nuovi requisiti da implementare. |

The XP release cycle



©Ian Sommerville 2004

Software Engineering, 7th edition. Chapter 17

Slide 6

Scenari di requisiti

In XP, i requisiti sono espressi come scenari o stories dell'utente che descrivono come il sistema dovrebbe essere utilizzato. Gli scenari sono scritte su schede in linguaggio naturale. Questi compiti sono la base delle stime di programmazione e pianificazione. Il cliente sceglie quali scenari devono essere incluse nel prossimo rilascio, in base alla sue priorità e alle stime di pianificazione.

Story card for document downloading

Downloading and printing an article

First, you select the article that you want from a displayed list. You then have to tell the system how you will pay for it - this can either be through a subscription, through a company account or by credit card.

After this, you get a copyright form from the system to fill in and, when you have submitted this, the article you want is downloaded onto your computer.

You then choose a printer and a copy of the article is printed. You tell the system if printing has been successful.

If the article is a print-only article, you can keep the PDF version so it is automatically deleted from your computer.

Testing in XP

Nel contesto di Extreme Programming (XP), il processo inizia con la scrittura dei test. Questo significa che prima di scrivere il codice, vengono creati test unitari per verificare il comportamento previsto del software. Questo approccio promuove l'affidabilità del software fin dall'inizio.

Durante lo sviluppo incrementale, i test sono sviluppati basandosi su scenari realistici, che rappresentano situazioni d'uso del software. Questi scenari guidano la creazione dei test, garantendo che il software soddisfi efficacemente le esigenze degli utenti.

Gli utenti partecipano attivamente nello sviluppo e nella validazione dei test. Questo coinvolgimento assicura che i test rispecchino le aspettative degli utenti finali e che il software sia allineato alle loro effettive necessità.

Per automatizzare il processo di testing, si utilizzano i "test harnesses" (strumenti di automazione dei test). Questi strumenti eseguono automaticamente i test su tutti i componenti del software ogni volta che viene creata una nuova versione, garantendo un controllo costante sulla qualità del software durante lo sviluppo.

Task cards for document downloading

Task 1: Implement principal workflow

Task 2: Implement article catalog and selection

Task 3: Implement payment collection

Payment may be made in 3 different ways. The user selects which way they wish to pay. If the user has a library subscription, then they can input the subscriber key which should be checked by the system. Alternatively they can input an organisational account number. If this is valid, a debit of the cost of the article is posted to this account. Finally, they may input a 16 digit credit card number and expiry date. This should be checked for validity and, if valid a debit is posted to that credit card account.

Test case description

Test 4: Test credit card validity

Input:

A string representing the credit card number and two integers representing the month and year when the card expires

Tests:

Check that all bytes in the string are digits
Check that the month lies between 1 and 12 and the year is greater than or equal to the current year
Using the first 4 digits of the credit card number check that the card issuer is valid by looking up the card issuer table. Check credit card validity by submitting the card number and expiry date information to the card issuer

Output:

OK or error message indicating that the card is invalid

Programmazione in Coppia

Nell'Extreme Programming (XP), la pratica della "Programmazione in coppia" è centrale. Questo significa che i programmatori lavorano a coppie, collaborando fisicamente nello sviluppo del codice. Questa metodologia promuove la condivisione della responsabilità del codice e diffonde la conoscenza all'interno del team. Inoltre, offre un processo informale di revisione del codice poiché ogni riga di codice è esaminata da più di una persona.

Un beneficio chiave della programmazione in coppia è la promozione del refactoring. Poiché più occhi sono puntati sul codice, il team può facilmente riconoscere opportunità per migliorare la struttura e l'efficienza del software. Questo conduce a un codice più pulito e manutenibile.

È interessante notare che le misurazioni suggeriscono che la produttività nello sviluppo con la programmazione in coppia è simile a quella di due persone che lavorano in modo indipendente. Questo dimostra l'efficacia di questa pratica nel mantenere un alto standard di qualità del codice mentre si mantiene una produttività sostenibile.

Riutilizzo del Software

Spesso, i sistemi vengono progettati componendo componenti esistenti che sono stati utilizzati in altri sistemi. Questa pratica di riutilizzo sistematico del software può portare a software di migliore qualità, sviluppato in modo più rapido e a costi inferiori.

Ingegneria del software basata sul riuso

Riutilizzo di sistemi applicativi

Questo tipo di riutilizzo comporta l'utilizzo dell'intero sistema applicativo, che può essere incorporato senza modifiche in altri sistemi (riutilizzo COTS) o sviluppando famiglie di applicazioni che condividono componenti essenziali.

Riutilizzo di componenti

Qui, si tratta di riutilizzare parti di un'applicazione, che possono variare da sottosistemi a singoli oggetti. Questo permette di utilizzare componenti software specifici all'interno di diverse applicazioni.

Riutilizzo di oggetti e funzioni

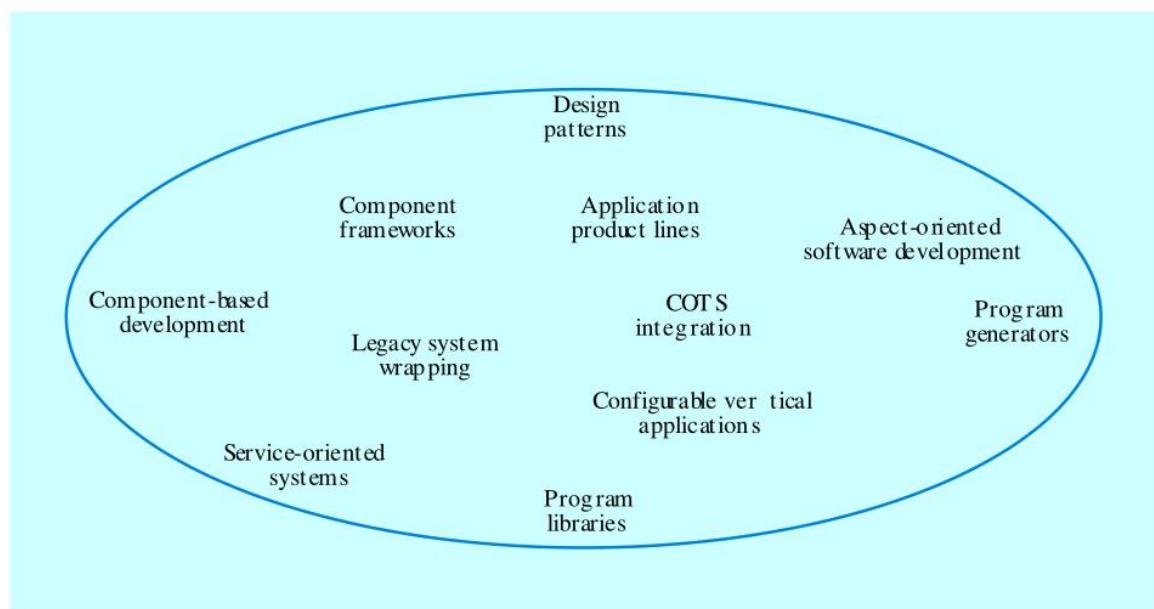
Questa forma di riutilizzo coinvolge componenti software che implementano un singolo oggetto o una funzione ben definita. Questi oggetti o funzioni possono essere riutilizzati in diverse parti di un sistema o in sistemi separati.

Il panorama del riuso

Esistono molteplici metodologie per il riutilizzo del software, e queste possono variare notevolmente in base alle esigenze specifiche del progetto. L'ingegneria del software offre un ampio spettro di strumenti e tecniche che possono essere adottati per il riutilizzo.

Il riutilizzo del software è possibile a vari livelli, che vanno dalle funzioni software più semplici fino a sistemi applicativi completi. Questa flessibilità consente ai professionisti dell'ingegneria del software di adattare il riutilizzo alle esigenze specifiche di un progetto, ottimizzando l'efficienza e la qualità del lavoro.

The reuse landscape



Reuse approaches 1

| | |
|-----------------------------|--|
| Design patterns | Generic abstractions that occur across applications are represented as design patterns that show abstract and concrete objects and interactions. |
| Component-based development | Systems are developed by integrating components (collections of objects) that conform to component-model standards. This is covered in Chapter 19. |
| Application frameworks | Collections of abstract and concrete classes that can be adapted and extended to create application systems. |
| Legacy system wrapping | Legacy systems (see Chapter 2) that can be <i>wrapped</i> by defining a set of interfaces and providing access to these legacy systems through these interfaces. |
| Service-oriented systems | Systems are developed by linking shared services that may be externally provided. |

Modified from Sommerville's originals

Software Engineering, 7th edition. Chapters 18 & 19

Slide 13

1. **Design Patterns:** Sono astrazioni generiche che si verificano attraverso diverse applicazioni. I design pattern mostrano oggetti astratti e concreti e le loro interazioni, e sono utili per risolvere problemi di design comuni e per comunicare soluzioni efficienti e eleganti.
2. **Component-based Development:** Questo approccio si concentra sullo sviluppo di sistemi attraverso l'integrazione di componenti (collezioni di oggetti) che sono conformi a standard di modelli di componenti. Questo metodo promuove il riuso e l'intercambiabilità dei componenti.
3. **Application Frameworks:** Si tratta di raccolte di classi astratte e concrete che possono essere adattate ed estese per creare sistemi applicativi. Gli application frameworks forniscono una struttura standard che può essere personalizzata per un'applicazione specifica.
4. **Legacy System Wrapping:** Alcuni sistemi legacy (vedi Capitolo 2 del testo di riferimento) possono essere "incapsulati" definendo un set di interfacce che permettono l'accesso alle funzionalità dei sistemi legacy attraverso queste interfacce. Questo permette di riutilizzare sistemi vecchi senza doverli riscrivere completamente.
5. **Service-oriented Systems:** Questi sistemi sono sviluppati collegando servizi condivisi che possono essere forniti esternamente. Il paradigma Service-oriented architecture (SOA) è fondamentale in questo contesto per permettere il riutilizzo dei servizi in diverse applicazioni e contesti.

Reuse approaches 2

| | |
|--------------------------------------|---|
| Application product lines | An application type is generalised around a common architecture so that it can be adapted in different ways for different customers. |
| COTS integration | Systems are developed by integrating existing application systems. (Special case of CBSE.) |
| Configurable vertical applications | A generic system is designed so that it can be configured to the needs of specific system customers. |
| Program libraries | Class and function libraries implementing commonly-used abstractions are available for reuse. |
| Program generators | A generator system embeds knowledge of a particular types of application and can generate systems or system fragments in that domain. |
| Aspect-oriented software development | Shared components are woven into an application at different places when the program is compiled. |

Modified from Sommerville's originals

Software Engineering, 7th edition. Chapters 18 & 19

Slide 14

- 1. Application Product Lines:** Questo approccio si concentra sulla creazione di un tipo di applicazione attorno a un'architettura comune che può essere adattata in modi diversi per diversi clienti. Consente di sviluppare prodotti software in modo più efficiente utilizzando un insieme comune di componenti software con la capacità di essere personalizzati per soddisfare esigenze specifiche.
- 2. COTS Integration:** Si tratta dello sviluppo di sistemi mediante l'integrazione di sistemi applicativi esistenti. COTS sta per "Commercial Off-The-Shelf" e si riferisce a prodotti software standardizzati disponibili sul mercato che possono essere acquisiti e integrati in altri sistemi.
- 3. Configurable Vertical Applications:** Questi sono sistemi generici progettati in modo da poter essere configurati per soddisfare le esigenze di clienti specifici del sistema. Sono applicazioni verticali perché sono specializzate per un particolare settore o tipo di attività.
- 4. Program Libraries:** Le librerie di programmi contengono classi e funzioni che implementano astrazioni comunemente utilizzate e sono disponibili per il riutilizzo. Queste librerie possono ridurre significativamente il tempo di sviluppo poiché forniscono soluzioni predefinite per compiti comuni.
- 5. Program Generators:** Sistemi generatori che incorporano la conoscenza di tipi particolari di applicazioni e possono generare sistemi o frammenti di sistemi in quel dominio. Questi possono automatizzare parti dello sviluppo del software e sono spesso utilizzati in ambienti dove le applicazioni simili vengono ripetutamente create con variazioni.

6. Aspect-oriented Software Development: Questo approccio si concentra sullo sviluppo software che separa le preoccupazioni incrociate (aspects) dal resto dell'applicazione. Componenti condivisi (aspects) vengono inseriti in un'applicazione in diversi punti quando il programma viene compilato, consentendo una maggiore modularità e riutilizzo del codice.

Riutilizzo dei concetti

Il riutilizzo concettuale del software comporta il seguimento delle decisioni progettuali prese dallo sviluppatore originale del componente. Questo significa che, quando si utilizza un componente software, si aderisce alla sua progettazione originale per garantire la coerenza.

I principali approcci per il riutilizzo concettuale includono l'utilizzo di "design patterns" (modelli di progettazione) e la programmazione generativa. I design patterns forniscono soluzioni comuni a problemi ricorrenti di progettazione del software, mentre la programmazione generativa si concentra sulla creazione automatica di codice basata su specifiche regole o modelli. Entrambi questi approcci aiutano a massimizzare il riutilizzo concettuale del software.

Design Patterns

I "design patterns" sono un modo per riutilizzare conoscenze astratte su un problema e la sua soluzione nell'ambito dell'ingegneria del software. Un design pattern rappresenta una descrizione del problema e l'essenza della sua soluzione. Deve essere sufficientemente astratto da poter essere riutilizzato in contesti diversi. Spesso, i design patterns si basano sulle caratteristiche degli oggetti, come l'ereditarietà e il polimorfismo, per fornire soluzioni flessibili e riutilizzabili a problemi comuni di progettazione del software.

Tipi di generatori di programmi

Esistono diversi tipi di generatori di programmi che coinvolgono il riutilizzo di modelli e algoritmi standard per generare automaticamente il codice del software. Questi tipi includono:

- **Generatori di applicazioni per l'elaborazione dei dati aziendali:**
Questi generatori sono utilizzati per creare software che si occupa dell'elaborazione dei dati aziendali, facilitando lo sviluppo di applicazioni aziendali comuni.
- **Generatori di parser e analizzatori lessicali per l'elaborazione del linguaggio:**
Questi strumenti sono progettati per generare codice che analizza e interpreta il linguaggio, spesso utilizzati nell'elaborazione dei linguaggi di programmazione.
- **Generatori di Codice all'interno degli strumenti CASE (Computer-Aided Software Engineering):**
Questi generatori sono inclusi negli strumenti CASE e sono utilizzati per automatizzare la generazione di codice in base ai modelli di progettazione.

Il riutilizzo basato su generatori di programmi è considerato molto vantaggioso in termini di costo ed è particolarmente efficace in specifici domini applicativi. Inoltre, rispetto alle approcci basati su componenti, può risultare più semplice per gli utenti finali. Tuttavia, la sua applicabilità potrebbe essere limitata a un numero ristretto di domini applicativi specifici.

Framework applicativi

I framework delle applicazioni sono progetti di sotto-sistemi costituiti da una raccolta di classi astratte e concrete e dalle interfacce tra di esse. L'implementazione del sotto-sistema avviene aggiungendo componenti per completare parti del progetto e istanziando le classi astratte presenti nel framework. I framework sono entità di dimensioni moderate che possono essere riutilizzate.

Riutilizzo del sistema applicativo

Il riutilizzo di sistemi applicativi comporta la riutilizzazione di interi sistemi applicativi, spesso attraverso:

- **Configurazione di un sistema:** Questo coinvolge la configurazione di un sistema esistente per adattarlo alle esigenze specifiche.

Integrazione di due o più sistemi: In questo caso, si integrano due o più sistemi distinti per formare un nuovo sistema completo.

Nel contesto del riutilizzo di sistemi applicativi, due approcci comuni includono:

- **Integrazione di prodotti COTS (Commercial Off-The-Shelf):** Questo coinvolge l'utilizzo di prodotti software commerciali esistenti e la loro integrazione per soddisfare i requisiti dell'applicazione.
- **Sviluppo di linee di prodotti:** Questo approccio prevede lo sviluppo di famiglie di applicazioni che condividono componenti essenziali, consentendo un elevato grado di riutilizzo.

L'obiettivo è massimizzare il riutilizzo dei sistemi applicativi esistenti per migliorare l'efficienza e ridurre i costi nello sviluppo di nuove applicazioni.

Riutilizzo dei prodotti COTS

Il riutilizzo di prodotti COTS (Commercial Off-The-Shelf) coinvolge l'utilizzo di sistemi commerciali completi che offrono un'API (Application Programming Interface). Questi sistemi COTS sono spesso applicazioni complete e pronte all'uso.

Un approccio valido nello sviluppo di sistemi di alcuni tipi, come i sistemi di E-commerce, è la costruzione di sistemi complessi mediante l'integrazione di sistemi COTS. Il principale vantaggio di questa strategia è la possibilità di sviluppare applicazioni più velocemente e, di solito, a costi di sviluppo inferiori.

L'utilizzo di prodotti COTS consente di sfruttare soluzioni software già esistenti e consolidate sul mercato, riducendo la necessità di sviluppare tutto da zero e accelerando il processo di implementazione.

Problemi di integrazione del sistema COTS

L'integrazione di sistemi COTS (Commercial Off-The-Shelf) può comportare alcune sfide:

Mancanza di Controllo sulla Funzionalità e le Prestazioni:

Poiché i sistemi COTS sono soluzioni preconfezionate, potrebbe mancare il controllo diretto sulla funzionalità e le prestazioni. Ciò significa che i sistemi COTS potrebbero non essere altrettanto efficaci come sembrano inizialmente.

Problemi di Interoperabilità tra Sistemi COTS:

Diversi sistemi COTS possono fare diverse assunzioni, rendendo difficile l'integrazione tra di essi. Questi sistemi possono avere protocolli di comunicazione diversi o dipendere da tecnologie diverse.

Assenza di Controllo sull'Evolutività del Sistema:

L'evoluzione dei sistemi COTS è controllata dai fornitori dei prodotti e non dagli utenti del sistema. Questo significa che potresti non avere controllo sul modo in cui il sistema evolve nel tempo.

Supporto da Parte dei Fornitori COTS:

I fornitori di sistemi COTS potrebbero non fornire supporto per l'intera durata di vita del prodotto, il che potrebbe comportare problemi di manutenzione e aggiornamento.

È importante prendere in considerazione queste sfide quando si decide di integrare sistemi COTS in un progetto, e pianificare di conseguenza per affrontarle e mitigare gli effetti.

Linee di prodotti software

Le "linee di prodotto software" o "famiglie di applicazioni" sono applicazioni che presentano funzionalità generiche e possono essere adattate e configurate in diverse varianti. L'adattamento può includere:

Configurazione di Componenti e Sistemi: Questo coinvolge la personalizzazione delle componenti e del sistema per soddisfare specifici requisiti o casi d'uso.

Aggiunta di Nuove Componenti al Sistema: È possibile estendere il sistema aggiungendo nuove componenti che ampliano la funzionalità o le capacità dell'applicazione.

Selezione da una Libreria di Componenti Esistenti: In alternativa, è possibile scegliere componenti preesistenti da una libreria di componenti disponibili per costruire l'applicazione desiderata.

Modifica delle Componenti per Soddisfare Nuovi Requisiti: Talvolta, è necessario apportare modifiche alle componenti esistenti per adattarle a nuovi requisiti o scenari di utilizzo.

Le linee di prodotto software sono un approccio flessibile per sviluppare una serie di applicazioni correlate che condividono una base comune ma possono essere adattate alle esigenze specifiche dei clienti o degli utenti finali.

Sistemi ERP

I sistemi ERP (Enterprise Resource Planning) rappresentano un tipo di sistema generico che supporta i processi aziendali comuni come ordini e fatturazione, produzione, ecc. Questi sistemi sono ampiamente utilizzati nelle grandi aziende e costituiscono probabilmente la forma più comune di riutilizzo del software.

Il nucleo generico di un sistema ERP viene adattato includendo moduli specifici e incorporando conoscenze sui processi aziendali e sulle regole operative. Questa personalizzazione consente agli ERP di soddisfare le esigenze specifiche di un'azienda, anche se si basano su una base comune.

Sviluppo basato su componenti

Lo sviluppo basato su componenti, noto come Component-Based Software Engineering (CBSE), è un approccio allo sviluppo del software che si basa sul riutilizzo del software. Questa metodologia è emersa come risposta alla difficoltà dell'approccio di sviluppo orientato agli oggetti nel supportare un riutilizzo efficace. Le classi di oggetti singoli risultano spesso troppo dettagliate e specifiche per il riutilizzo efficiente.

I componenti nel contesto del CBSE sono più astratti rispetto alle classi di oggetti e possono essere considerati come fornitori di servizi autonomi e indipendenti. Questo livello di astrazione consente una maggiore flessibilità nell'uso e nel riutilizzo dei componenti all'interno di diverse applicazioni.

CBSE essenziale

Nel contesto dell'ingegneria del software basata su componenti (CBSE), ci sono quattro elementi essenziali:

1. Componenti Indipendenti Specificati tramite Interfacce:

I componenti sono unità di software indipendenti che vengono specificati attraverso le loro interfacce. Queste interfacce definiscono come i componenti possono essere utilizzati e interagiti con altri elementi del sistema.

2. Standard per i Componenti per Agevolare l'Integrazione:

È importante avere standard e convenzioni per i componenti in modo che possano essere facilmente integrati in diverse applicazioni. Questi standard definiscono come i componenti comunicano e si integrano tra loro.

3. Middleware per Supportare l'Interoperabilità dei Componenti:

Il middleware è un livello di software che facilita la comunicazione e l'interoperabilità tra i componenti. Fornisce una base tecnologica comune per il funzionamento dei componenti all'interno di un sistema.

4. Processo di Sviluppo Orientato al Riutilizzo:

Il processo di sviluppo deve essere orientato al riutilizzo, il che significa che deve essere progettato per massimizzare il riutilizzo dei componenti esistenti e facilitare la creazione di nuovi componenti riutilizzabili.

CBSE e principi di progettazione

Oltre ai vantaggi del riutilizzo del software, l'ingegneria del software basata su componenti (CBSE) si basa su solidi principi di progettazione del software:

- **Indipendenza dei componenti:** I componenti sono progettati per essere indipendenti l'uno dall'altro, il che significa che le modifiche apportate a un componente non influenzano gli altri. Questa separazione favorisce la manutenibilità e la robustezza del sistema.
- **Nascondimento delle implementazioni dei componenti:** Le implementazioni interne dei componenti sono nascoste e non visibili all'esterno attraverso le interfacce. Questo principio di nascondimento migliora la modularità e la manutenibilità del sistema.
- **Comunicazione tramite interfacce ben definite:** La comunicazione tra i componenti avviene attraverso interfacce ben definite, il che significa che i componenti devono seguire regole chiare per scambiare informazioni. Ciò favorisce l'interoperabilità e la comprensibilità del sistema.

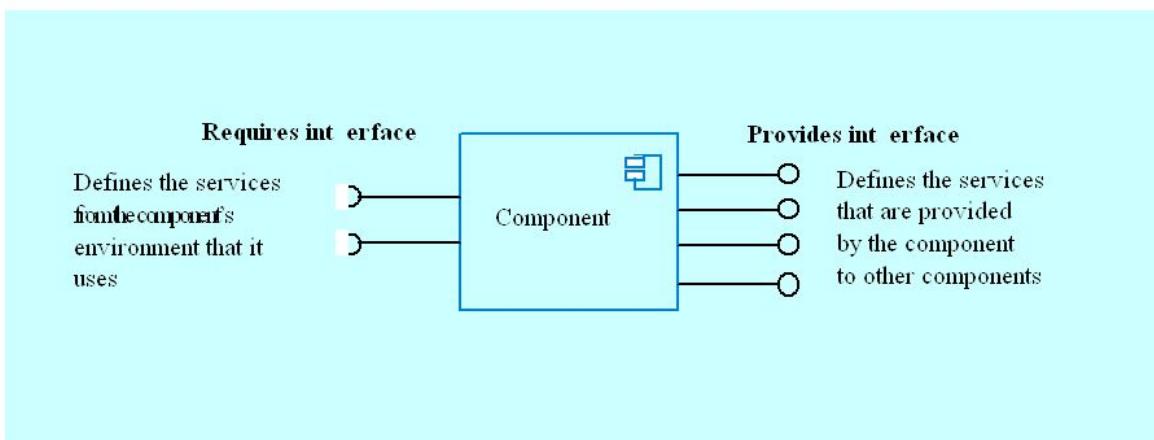
Piattaforme Condivise dei Componenti: La condivisione di piattaforme tra i componenti riduce i costi di sviluppo, in quanto più componenti possono sfruttare la stessa infrastruttura o tecnologia di base, evitando la duplicazione degli sforzi di sviluppo.

Componenti

I componenti forniscono un servizio senza considerare dove il componente è in esecuzione o il linguaggio di programmazione utilizzato. Ecco alcune caratteristiche chiave dei componenti:

- **Componente come entità esecutiva indipendente:** Un componente è un'entità esecutiva indipendente che può essere composta da uno o più oggetti eseguibili. Questa indipendenza consente ai componenti di essere utilizzati in diversi contesti senza dover conoscere i dettagli interni.
- **Pubblicazione dell'interfaccia del componente:** L'interfaccia del componente è resa pubblica, il che significa che è disponibile per essere utilizzata da altre parti del sistema o da sistemi esterni. Tutte le interazioni con il componente avvengono attraverso questa interfaccia pubblicata.

Component interfaces



©Ian Sommerville 2006

Software Engineering, 8th edition. Chapter 19

Slide 14

Componenti e oggetti

I componenti e gli oggetti nel contesto dell'ingegneria del software hanno alcune differenze chiave:

Componenti sono Entità Deployable: I componenti sono entità deployable, il che significa che possono essere distribuite e utilizzate in diversi contesti. Gli oggetti, d'altra parte, sono spesso considerati come parte di un'applicazione in esecuzione.

Componenti non Definiscono Tipi: I componenti non definiscono tipi di dati specifici come gli oggetti. Invece, si concentrano su servizi o funzionalità forniti attraverso le loro interfacce pubbliche.

Implementazioni dei Componenti sono Opache: L'implementazione interna dei componenti è nascosta e opaca attraverso le loro interfacce. Gli oggetti possono essere utilizzati direttamente, consentendo l'accesso ai loro dettagli interni.

Componenti sono Indipendenti dal Linguaggio: I componenti sono progettati per essere indipendenti dal linguaggio di programmazione, il che significa che possono essere utilizzati in sistemi sviluppati in diversi linguaggi.

Standardizzazione dei Componenti: Spesso, i componenti seguono standard o convenzioni specifiche per garantire l'interoperabilità e il riutilizzo in diversi contesti.

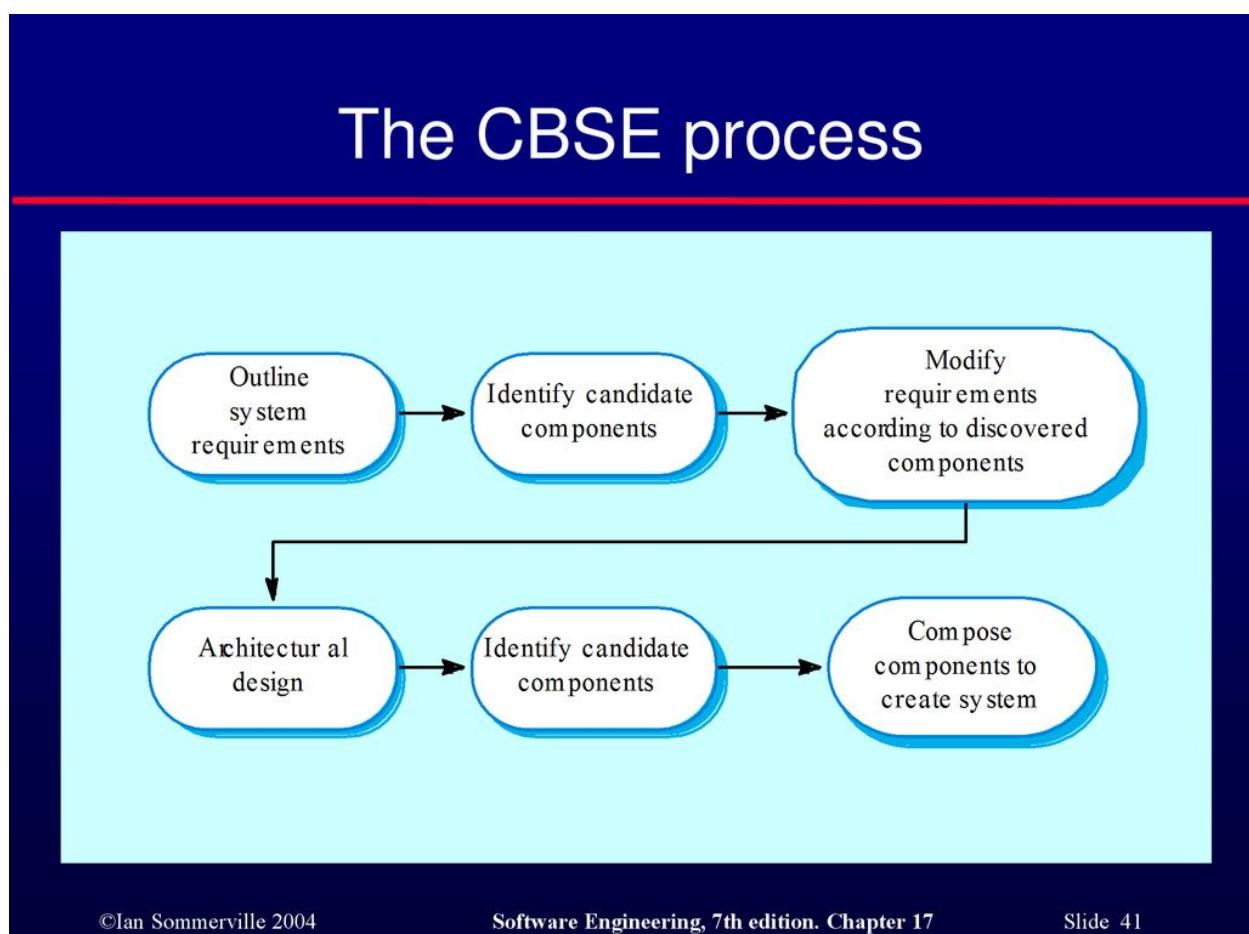
Modelli di componenti

Un modello di componente è una definizione di standard per l'implementazione, la documentazione e la distribuzione di componenti software. Ecco alcuni esempi di modelli di componente:

- **Modello EJB (Enterprise Java Beans):** Questo modello è specifico per lo sviluppo di componenti in Java ed è ampiamente utilizzato nelle applicazioni enterprise. Definisce come i componenti devono essere implementati e utilizzati in ambienti Java.
- **Modello COM+ (Modello .NET):** Questo modello è legato all'ecosistema .NET di Microsoft ed è utilizzato per la creazione e l'utilizzo di componenti in applicazioni basate su .NET.
- **Modello Corba Component:** Questo modello è basato sulla tecnologia CORBA (Common Object Request Broker Architecture) e offre un modo di definire e utilizzare componenti in sistemi distribuiti.

Il modello di componente specifica come le interfacce dovrebbero essere definite e quali elementi dovrebbero essere inclusi nella definizione di un'interfaccia. Questi modelli aiutano a stabilire standard e convenzioni che rendono più semplice lo sviluppo, la documentazione e la distribuzione di componenti software.

Il processo CBSE



Evoluzione del software

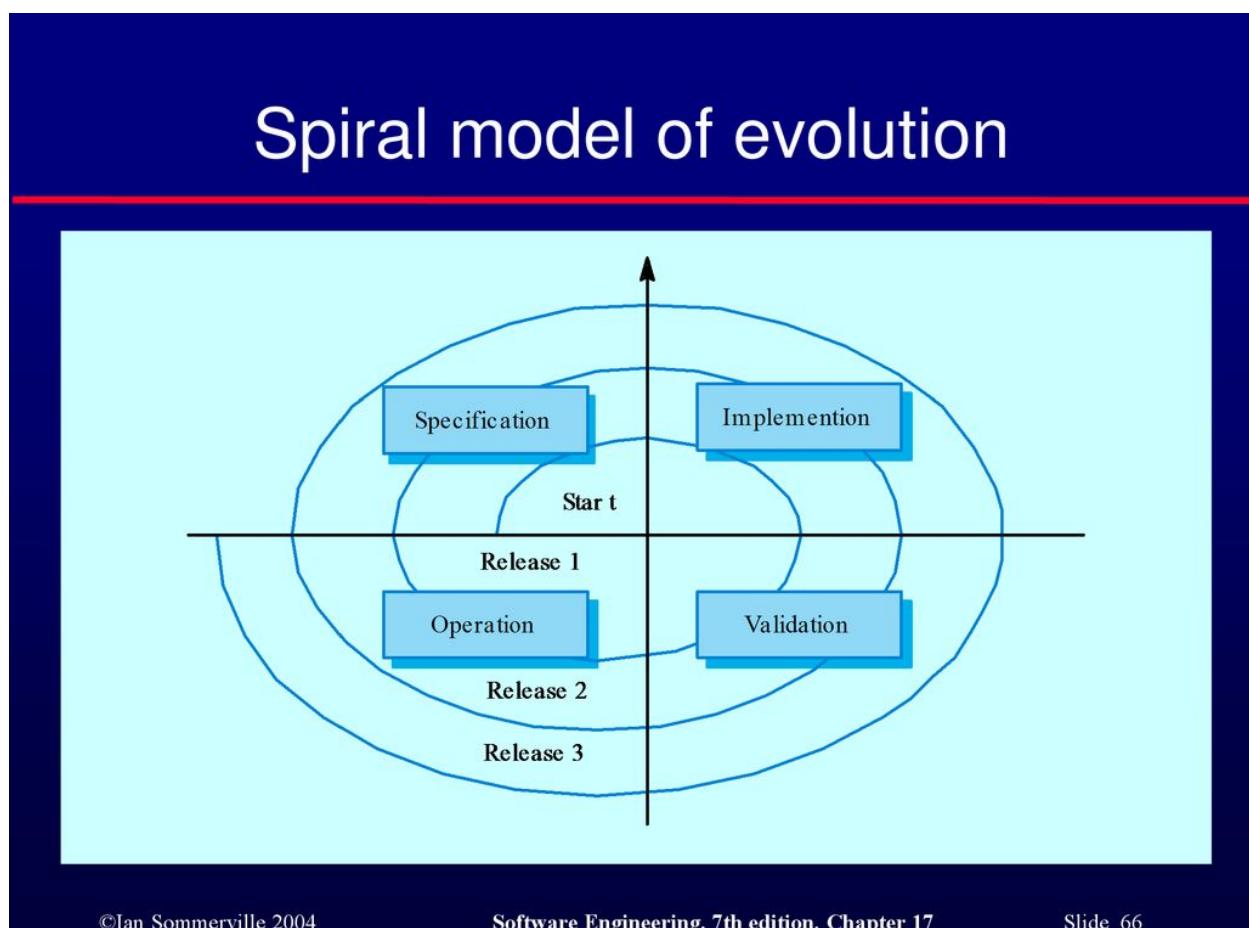
Modifica del software

Il cambiamento del software è inevitabile e può derivare da diverse cause, tra cui:

- Emergenza di nuovi requisiti quando il software è in uso.
- Cambiamenti nell'ambiente aziendale.
- Necessità di correggere errori.
- Aggiunta di nuovi computer ed equipaggiamento al sistema.
- Miglioramento delle prestazioni o della affidabilità del sistema.

Un problema chiave per le organizzazioni è l'implementazione e la gestione del cambiamento nei loro sistemi software esistenti. Affrontare con successo il cambiamento richiede pianificazione, gestione e una comprensione chiara dei processi di manutenzione e aggiornamento del software.

Modello a spirale dell'evoluzione



Dinamiche di evoluzione del programma

La dinamica dell'evoluzione del programma è lo studio dei processi di cambiamento dei sistemi software nel tempo. Lehman e Belady hanno proposto una serie di "leggi" basate su importanti studi empirici, applicate a tutti i sistemi durante il loro sviluppo. Tuttavia, queste "leggi" sono considerate più come osservazioni sensate piuttosto che leggi rigorose. Sono particolarmente applicabili a grandi sistemi sviluppati da grandi organizzazioni e potrebbero essere meno rilevanti in altri contesti.

Queste leggi, o osservazioni, aiutano a comprendere i processi di cambiamento del software e possono essere utilizzate come guida per la gestione e l'evoluzione dei sistemi software nel tempo.

Leggi di Lehman

Lehman's Laws

| Law | Description |
|-----------------------------|---|
| Continuing change | A program must change or become progressively less useful |
| Increasing complexity | As program changes, its structure becomes more complex Extra resources are required. |
| Large program evolution | System attributes, (eg, size, time between releases) is ~invariant for each system release. |
| Organizational stability | A program's rate of development is ~constant. |
| Conservation of familiarity | The incremental change in each release is ~constant. |
| Continuing growth | The functionality has to continually increase to maintain user satisfaction. |
| Declining quality | Quality of systems appear to be declining unless adapted to changing environments |
| Feedback system | Evolution processes involve feedback systems for product improvement |

Manutenzione del software

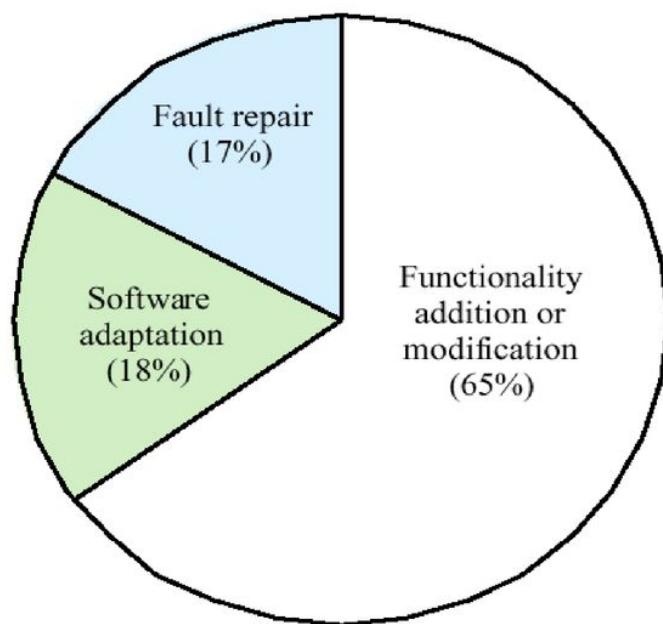
La manutenzione del software è il processo di modifica di un programma dopo che è stato messo in uso. Tipicamente, la manutenzione non comporta modifiche sostanziali all'architettura del sistema. Le modifiche vengono attuate attraverso la modifica dei componenti esistenti e l'aggiunta di nuovi componenti al sistema.

La manutenzione del software è inevitabile perché l'ambiente in cui opera il software è in costante cambiamento, e di conseguenza anche i requisiti possono cambiare. Affinché un sistema software rimanga utile nell'ambiente in cui è utilizzato, è necessario che subisca manutenzione.

Inoltre, la manutenzione del software è una parte essenziale del ciclo di vita del software che segue lo sviluppo iniziale e può durare per un periodo molto più lungo. È importante pianificare e gestire efficacemente la manutenzione del software per garantire che il sistema rimanga efficiente e in grado di soddisfare le esigenze in evoluzione degli utenti.

Distribuzione degli sforzi di manutenzione

Distribution of maintenance effort



Fattori dei costi di manutenzione

Ci sono diversi fattori che influenzano i costi di manutenzione del software:

- **Stabilità del team**

I costi di manutenzione tendono a diminuire se lo stesso personale è coinvolto per un certo periodo. La familiarità con il sistema può ridurre i tempi di diagnosi e correzione degli errori.

- **Responsabilità contrattuale**

Se i creatori del sistema non hanno alcuna responsabilità contrattuale per la manutenzione, potrebbe non esserci un incentivo per progettare il sistema in modo che sia facilmente modificabile in futuro.

- **Competenze del Personale**

Il personale di manutenzione spesso ha meno esperienza rispetto ai creatori del sistema e può avere conoscenze limitate nel dominio del sistema. Ciò può aumentare i tempi di risoluzione dei problemi.

- **Età e Struttura del Programma**

Con il tempo, i programmi tendono a degradarsi nella loro struttura e diventare più complessi da comprendere e modificare. L'età del software può quindi influenzare i costi di manutenzione.

Questi fattori evidenziano l'importanza di una pianificazione adeguata per la manutenzione del software e l'importanza di progettare il software con una visione a lungo termine, considerando i costi futuri di manutenzione.

Processi di evoluzione

I processi di evoluzione del software dipendono da diversi fattori, tra cui:

- **Tipo di software in manutenzione**

Il tipo di software che viene mantenuto influenzera il processo di evoluzione. Ad esempio, i processi per il software embedded in un dispositivo medico saranno diversi da quelli per un'applicazione web.

- **Processi di sviluppo utilizzati**

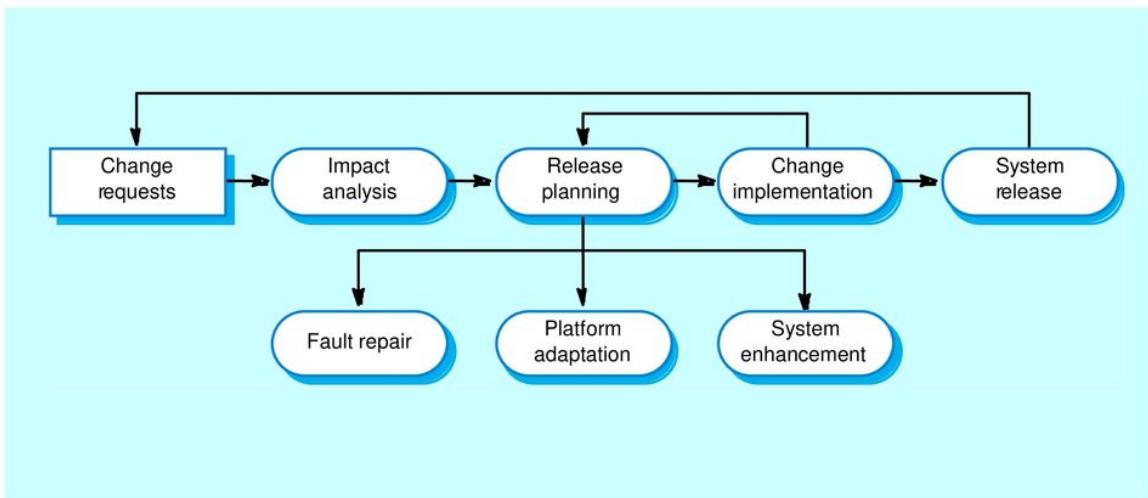
Il modo in cui il software è stato originariamente sviluppato influenzera come avviene la sua evoluzione. Se il software è stato sviluppato utilizzando metodi agili, ad esempio, il processo di evoluzione potrebbe essere più flessibile.

- **Competenze ed esperienza delle persone coinvolte**

Le competenze e l'esperienza del personale coinvolto nella manutenzione del software sono fondamentali. Il personale esperto può gestire meglio le modifiche e l'evoluzione del sistema.

Le proposte di cambiamento sono il motore dell'evoluzione del sistema. L'identificazione e l'evoluzione delle modifiche continuano per tutta la durata del sistema software, poiché le esigenze dei clienti cambiano e nuove sfide emergono nel tempo. La gestione efficace di questo processo è essenziale per garantire che il software rimanga utile e adattato alle esigenze in evoluzione.

The system evolution process



Sommerville, Ch. 21

1. **Change Requests:** Il processo inizia quando vengono ricevute richieste di cambiamento. Queste possono essere correzioni di errori, miglioramenti o aggiornamenti del sistema per soddisfare nuove esigenze o tecnologie.
2. **Impact Analysis:** Dopo aver ricevuto una richiesta di cambiamento, viene condotta un'analisi d'impatto per comprendere le implicazioni della modifica proposta sul sistema esistente.
3. **Release Planning:** Una volta compreso l'impatto, si procede con la pianificazione del rilascio. In questa fase si decide quando e come le modifiche verranno implementate, tenendo conto di fattori come risorse, tempistiche e rischi.
4. **Change Implementation:** Le modifiche pianificate sono poi effettivamente implementate nel sistema. Questa fase può includere la scrittura di nuovo codice, la modifica di quello esistente e la verifica che le modifiche funzionino come previsto.
5. **System Release:** Infine, dopo un'attenta implementazione e test, il sistema aggiornato viene rilasciato e messo a disposizione degli utenti.

Il diagramma mostra anche tre attività che possono avvenire in parallelo o come parte del processo di evoluzione:

- **Fault Repair:** La riparazione di errori che sono stati identificati nel sistema.
- **Platform Adaptation:** L'adattamento del sistema a nuove piattaforme hardware o software.
- **System Enhancement:** Il miglioramento del sistema aggiungendo nuove funzionalità o migliorando quelle esistenti.

Questo processo è iterativo e ciclico, il che significa che dopo il rilascio del sistema, nuove richieste di cambiamento possono iniziare un altro ciclo del processo di evoluzione del sistema.

Reingegnerizzazione del sistema

La ri-ingegnerizzazione del sistema è un processo che comporta la ristrutturazione o la riscrittura di parte o di tutto un sistema legacy senza cambiarne la funzionalità. Questo approccio è utile quando alcune parti di un sistema, ma non tutte, richiedono frequenti operazioni di manutenzione.

La ri-ingegnerizzazione coinvolge l'aggiunta di sforzi per rendere tali parti più facili da mantenere. Questo può includere la ristrutturazione del codice, la documentazione aggiornata e altre attività per migliorare la manutenibilità del sistema.

L'obiettivo principale della ri-ingegnerizzazione del sistema è mantenere e migliorare il sistema esistente, preservando la sua funzionalità, ma rendendolo più efficiente, comprensibile e facilmente manutenibile nel tempo.

Vantaggi della ri-ingegnerizzazione

I vantaggi della ri-ingegnerizzazione del sistema includono:

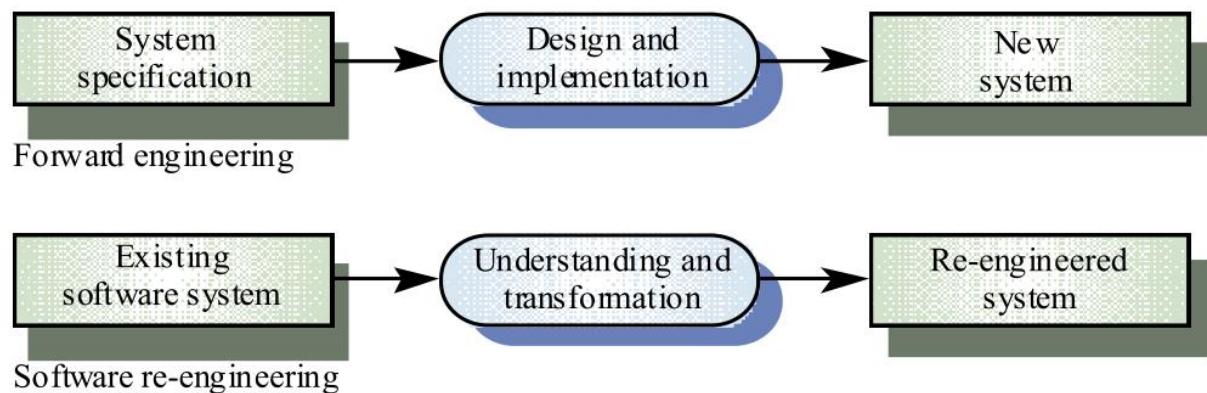
- **Riduzione del rischio**

La creazione di nuovo software comporta un alto rischio. Potrebbero sorgere problemi durante lo sviluppo, problemi legati al personale e problemi di specifiche. La ri-ingegnerizzazione, che mira a mantenere la funzionalità esistente, riduce il rischio di introdurre nuovi problemi significativi.

- **Riduzione dei costi**

I costi associati alla ri-ingegnerizzazione sono spesso significativamente inferiori rispetto ai costi di sviluppare nuovo software da zero. Questo può rappresentare un risparmio considerevole per le organizzazioni, specialmente quando è necessario mantenere funzionalità esistenti.

Forward engineering and re-engineering



Attività del processo di reingegnerizzazione

Le attività del processo di ri-ingegnerizzazione includono:

- **Traduzione del codice sorgente**

Questa attività coinvolge la conversione del codice sorgente esistente in un nuovo linguaggio di programmazione. Può essere necessario se si vuole migrare il software su una piattaforma o tecnologia differente.

- **Reverse Engineering (Ingegneria Inversa)**

Qui si analizza il programma esistente per comprenderlo. L'obiettivo è acquisire una comprensione dettagliata del software esistente, spesso attraverso la lettura e l'analisi del codice sorgente.

- **Miglioramento della struttura del programma**

Questa attività mira a ristrutturare automaticamente il codice per renderlo più comprensibile. Questo può coinvolgere la ridenominazione delle variabili, la semplificazione della logica e altre modifiche per migliorare la leggibilità.

- **Modularizzazione del programma:**

Qui si riorienta la struttura del programma in modo modulare. Ciò implica la suddivisione del programma in moduli o componenti più piccoli e gestibili, il che facilita la manutenzione futura.

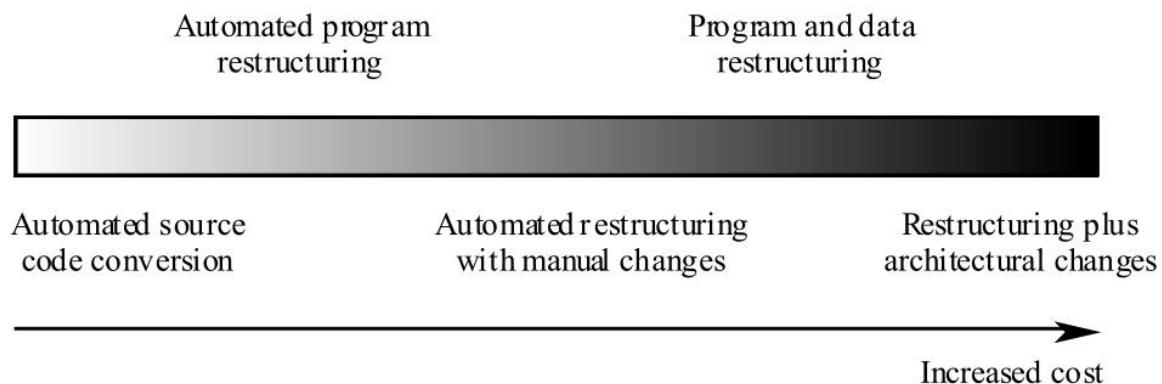
- **Ri-ingegnerizzazione dei dati:**

Questa attività coinvolge la pulizia e la ri-strutturazione dei dati all'interno del sistema. Può essere necessaria per garantire che i dati siano coerenti, ben organizzati e adatti alle esigenze del sistema.

Queste attività lavorano insieme per migliorare il software esistente, rendendolo più comprensibile, manutenibile e adatto alle esigenze attuali dell'organizzazione.

Approcci di reingegnerizzazione

Re-engineering approaches



©Ian Sommerville 2000

Software Engineering, 6th edition. Chapter 28

Slide 13

La slide illustra diversi approcci al re-engineering del software, ordinati in base al costo crescente:

1. **Automated Source Code Conversion:** Questo approccio utilizza strumenti automatici per convertire il codice sorgente da un linguaggio, database, o piattaforma ad un altro. È generalmente il meno costoso poiché richiede il minimo intervento umano.
2. **Automated Program Restructuring:** Qui, il codice sorgente viene trasformato automaticamente per migliorarne la struttura senza cambiare il suo comportamento funzionale. Questo può includere la riformattazione del codice, l'organizzazione di moduli, l'ottimizzazione delle performance e la rimozione di codice ridondante.
3. **Automated Restructuring with Manual Changes:** Questo approccio combina tecniche di re-engineering automatico con modifiche manuali. Dopo la restrutturazione automatica, gli sviluppatori intervengono per fare aggiustamenti che non possono essere automatizzati.
4. **Program and Data Restructuring:** Questo metodo va oltre la semplice ristrutturazione del programma e include anche la ristrutturazione dei dati associati. Può comportare la migrazione di database, la modifica degli schemi di dati, e l'adattamento del codice per utilizzare i nuovi formati dei dati.

5. **Restructuring Plus Architectural Changes:** L'approccio più costoso, che non solo ristruttura il programma e i dati ma include anche cambiamenti all'architettura software. Questo potrebbe significare l'introduzione di nuovi pattern architettonici, la riscrittura di componenti per migliorare la manutenzione e la scalabilità, o l'adattamento del sistema per nuovi requisiti tecnologici.

La barra che va dal grigio chiaro al nero rappresenta l'aumento del costo associato a ciascun approccio. Il re-engineering più avanzato richiede più risorse e tempo, aumentando così il costo totale del progetto.

Evoluzione del sistema legacy

Le organizzazioni che dipendono dai sistemi legacy devono scegliere una strategia per l'evoluzione di questi sistemi. Ecco alcune opzioni comuni:

- **Scarto completo del sistema:**
In questa strategia, l'organizzazione abbandona completamente il sistema legacy e modifica i processi aziendali in modo che il sistema non sia più necessario. Questa è una scelta drastica che può essere presa se il sistema è obsoleto o troppo costoso da mantenere.
- **Continuare la manutenzione:**
Alcune organizzazioni possono scegliere di continuare a mantenere il sistema legacy se è ancora funzionale e svolge un ruolo critico nei processi aziendali. Questa strategia può essere adottata se i costi di manutenzione sono accettabili e il sistema continua a fornire valore.
- **Trasformare il sistema tramite ri-ingegnerizzazione:**
In questa strategia, il sistema legacy viene sottoposto a un processo di ri-ingegnerizzazione per migliorarne la manutenibilità e l'efficienza. Questo può essere un'opzione se il sistema è fondamentale ma la sua struttura è obsoleta o difficile da mantenere.
- **Sostituire il Sistema con un nuovo sistema:**
In alcuni casi, l'organizzazione può optare per la sostituzione completa del sistema legacy con un nuovo sistema. Questa scelta può essere giustificata se il sistema legacy non può essere adeguatamente migliorato o se un nuovo sistema offre significativi vantaggi in termini di funzionalità o efficienza.

La strategia scelta dovrebbe dipendere dalla qualità del sistema legacy e dal suo valore per l'organizzazione. È importante valutare attentamente le opzioni e prendere decisioni informate basate sulle esigenze aziendali e tecniche.

Categorie del sistema legacy

Le categorie dei sistemi legacy possono essere suddivise in base alla loro qualità e al loro valore aziendale. Ecco una panoramica delle quattro categorie:

- **Bassa Qualità, Basso Valore Aziendale:** Questi sistemi sono caratterizzati da una bassa qualità e offrono un valore aziendale limitato. In genere, la migliore opzione per questi sistemi è il loro abbandono completo.

- **Bassa Qualità, Alto Valore Aziendale:** Questi sistemi forniscono un contributo importante all'attività aziendale, ma richiedono costi significativi per la manutenzione. In questo caso, dovrebbe essere considerata l'opzione di ri-ingegnerizzazione o sostituzione con un sistema adeguato, se disponibile.
- **Alta Qualità, Bassa Valore Aziendale:** Questi sistemi sono di alta qualità ma offrono un valore aziendale limitato. Le opzioni per questi sistemi includono la sostituzione con sistemi Commercial Off-The-Shelf (COTS), l'abbandono completo o la manutenzione.
- **Alta Qualità, Alto Valore Aziendale:** Questi sistemi sono di alta qualità e svolgono un ruolo critico nell'attività aziendale. La strategia migliore in questo caso è continuare l'operatività del sistema attraverso la manutenzione normale del sistema.

Le decisioni sulla gestione dei sistemi legacy dovrebbero essere prese in base a queste categorie, valutando attentamente la qualità e il valore aziendale di ciascun sistema per determinare la strategia più appropriata.

Verifica e validazione

Verifica vs Validazione

Verifica:

La verifica risponde alla domanda "Stiamo costruendo il prodotto correttamente?". Si tratta di un processo che mira a confermare che il software è stato sviluppato in conformità con le specifiche stabilite. In altre parole, verifica che il software soddisfi i requisiti tecnici e le specifiche tecniche. La verifica si concentra sulla correttezza tecnica del software.

Convalida:

La convalida risponde alla domanda "Stiamo costruendo il prodotto giusto?". Si concentra sulla conferma che il software soddisfi le esigenze e le aspettative degli utenti finali. La convalida si concentra sulla funzionalità del software e sul fatto che effettivamente risolva i problemi o le esigenze degli utenti.

Il processo V&V

La verifica e la validazione sono un processo che riguarda l'intero ciclo di vita del software e devono essere applicate in ogni fase del processo di sviluppo del software. Questo processo ha due obiettivi principali:

1. Scoperta di difetti nel sistema:

La verifica e la validazione sono utilizzate per individuare difetti o errori nel sistema software. Questi difetti possono riguardare problemi tecnici, errori di implementazione o discrepanze rispetto ai requisiti.

2. Valutazione dell'utilità e dell'utilizzabilità del sistema:

Oltre alla scoperta di difetti, la verifica e la validazione mirano anche a valutare se il sistema è utile e utilizzabile in una situazione operativa. Questo significa assicurarsi che il software non solo funzioni correttamente, ma soddisfi anche le esigenze degli utenti e sia adatto all'uso previsto.

Obiettivi del V&V

Gli obiettivi della verifica e della validazione (V&V) includono:

1. Stabilire la Fiducia che il Software sia Adeguato allo Scopo:

La V&V mira a fornire una certa fiducia che il software sia adatto all'uso previsto. Questo non significa che il software deve essere completamente privo di difetti, ma piuttosto che deve essere abbastanza buono per soddisfare il suo scopo previsto.

2. Non Necessariamente Privi di Difetti:

È importante sottolineare che l'obiettivo della V&V non è necessariamente quello di avere un software completamente privo di difetti. La perfezione assoluta può essere irraggiungibile, ma il software deve essere accettabile per il suo utilizzo previsto.

3. Buono Abbastanza per l'Uso Previsto:

La qualità del software deve essere tale da consentire il suo utilizzo effettivo e senza gravi problemi. L'importante è che il software soddisfi le esigenze degli utenti e sia adatto alle attività per cui è stato sviluppato.

Fiducia nella V&V

La fiducia richiesta nella verifica e nella validazione (V&V) del software dipende da diversi fattori, tra cui:

1. Scopo del Sistema, Aspettative degli Utenti e Ambiente di Marketing:

La fiducia richiesta nella V&V del software dipende dallo scopo del sistema, dalle aspettative degli utenti e dall'ambiente di marketing. Questi fattori influenzano quanto il software debba essere accuratamente verificato e convalidato.

2. Funzione del Software:

La funzione specifica del software gioca un ruolo importante nella determinazione del livello di fiducia richiesto. Il software critico per l'organizzazione, che svolge un ruolo fondamentale nelle operazioni aziendali, richiede un livello di fiducia più elevato rispetto al software meno critico.

3. Aspettative degli Utenti:

Le aspettative degli utenti possono variare a seconda del tipo di software. Per alcune categorie di software, gli utenti possono avere aspettative relativamente basse in termini di funzionalità o prestazioni.

4. Ambiente di Marketing:

L'ambiente di marketing può influenzare la priorità tra la consegna rapida del prodotto sul mercato e la ricerca di difetti nel programma. In alcuni casi, è più importante rilasciare il prodotto sul mercato il prima possibile, anche se ciò significa tollerare alcuni difetti minori.

Verifica statica e dinamica

Le attività di verifica e validazione nel processo di sviluppo del software possono essere suddivise in due principali categorie:

1. Ispezioni del Software:

Le ispezioni del software sono attività che si concentrano sull'analisi della rappresentazione statica del sistema per scoprire problemi. Questa è una forma di verifica statica, che significa che non coinvolge l'esecuzione del software, ma piuttosto l'analisi della sua rappresentazione documentale, come specifiche, documentazione del codice, diagrammi, ecc. Le ispezioni del software coinvolgono tipicamente revisioni manuali del materiale documentario per individuare difetti o problemi potenziali. Possono anche essere supportate da strumenti di analisi dei documenti e del codice.

2. Test del Software:

Il testing del software è un'attività che si concentra sull'esecuzione e sull'osservazione del comportamento del prodotto software. Questa è una forma di verifica dinamica in cui il software viene eseguito con dati di test e il suo comportamento operativo viene osservato e confrontato con le aspettative. Il testing può rilevare problemi come errori di logica, malfunzionamenti e prestazioni inadeguate.

Entrambe queste attività sono essenziali per garantire la qualità del software. Le ispezioni del software consentono di individuare difetti nel materiale documentario e nelle specifiche, mentre il testing del software verifica il comportamento effettivo del software in esecuzione. Entrambe le forme di verifica e validazione sono complementari e contribuiscono a garantire che il software sia corretto e adatto all'uso previsto.

Test del programma

È importante notare alcune considerazioni chiave sulla validazione del software:

- 1. Rivelazione di Errori, Non Assenza:** La validazione del software può rivelare la presenza di errori o problemi nel comportamento del software, ma non può garantire l'assenza completa di errori. Anche dopo un rigoroso processo di validazione, potrebbero ancora esistere difetti non rilevati.
- 2. Tecnica per i Requisiti Non Funzionali:** La validazione del software è l'unica tecnica adatta per testare i requisiti non funzionali. Questi requisiti riguardano aspetti come le prestazioni, la scalabilità, la sicurezza e l'usabilità, e richiedono l'esecuzione effettiva del software per valutarli. Al contrario, la verifica statica è più adatta per i requisiti funzionali e l'analisi dei documenti.

3. Complemento alla Verifica Statica: La validazione del software dovrebbe essere utilizzata in combinazione con la verifica statica per ottenere una copertura completa nella verifica e nella validazione (V&V) del software. La verifica statica si concentra sull'analisi della rappresentazione statica del software, mentre la validazione dinamica coinvolge l'esecuzione effettiva del software. Entrambe le tecniche sono necessarie per garantire una corretta qualità del software.

Tipi di test

Nel contesto del testing del software, ci sono due approcci principali:

1. Defect Testing (Testing dei Difetti):

Il testing dei difetti è progettato per scoprire difetti o problemi nel sistema software. In questo tipo di testing, si cercano attivamente difetti, e un test dei difetti è considerato riuscito se rileva la presenza di difetti nel sistema. L'obiettivo principale è individuare e segnalare difetti o problemi che devono essere corretti.

2. Validation Testing (Testing di Convalida):

Il testing di convalida è progettato per dimostrare che il software soddisfa i suoi requisiti. In questo tipo di testing, l'obiettivo è verificare che il software implementi correttamente i requisiti specificati. Un test di convalida è considerato riuscito se dimostra che un requisito è stato implementato correttamente.

Entrambi questi approcci sono importanti nel processo di testing del software. Il testing dei difetti aiuta a individuare problemi nel software, mentre il testing di convalida verifica che il software risponda alle aspettative e ai requisiti dell'utente. Entrambi sono complementari e contribuiscono a garantire la qualità complessiva del software.

Test e debug

È importante comprendere le distinzioni tra il testing dei difetti, la verifica, la validazione e il debugging nel processo di sviluppo del software:

1. Defect Testing (Testing dei Difetti):

Il testing dei difetti è un processo che mira a scoprire la presenza di difetti o problemi nel software. È una parte della verifica e della validazione (V&V) del software e si concentra sulla ricerca attiva di difetti. Un test dei difetti è riuscito se rileva la presenza di difetti nel sistema.

2. Debugging:

Il debugging è un processo separato che si verifica dopo che sono stati identificati difetti durante il testing. Si concentra sulla localizzazione e sulla correzione degli errori specifici nel codice del programma. Il debugging coinvolge la formulazione di ipotesi sul comportamento del programma e il testing di queste ipotesi per individuare e risolvere gli errori nel sistema.

3. Verifica e Validazione (V&V):

La verifica e la validazione sono processi più ampi che riguardano l'intero ciclo di vita del software. La verifica è la conferma che il software è stato sviluppato correttamente in base alle specifiche, mentre la validazione si concentra sul fatto che il software soddisfi le esigenze degli utenti. Entrambi questi processi sono finalizzati a stabilire la presenza di difetti nel programma.

Il piano di test del software

The software test plan

The testing process

A description of the major phases of the testing process. These might be as described earlier in this chapter.

Requirements traceability

Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

Tested items

The products of the software process that are to be tested should be specified.

Testing schedule

An overall testing schedule and resource allocation for this schedule. This, obviously, is linked to the more general project development schedule.

Test recording procedures

It is not enough simply to run tests. The results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it been carried out correctly.

Hardware and software requirements

This section should set out software tools required and estimated hardware utilisation.

Constraints

Constraints affecting the testing process such as staff shortages should be anticipated in this section.

Ispezioni del software

Le ispezioni del software sono un processo che coinvolge persone nell'esame della rappresentazione sorgente del software con l'obiettivo di scoprire anomalie e difetti. Ecco alcune caratteristiche chiave delle ispezioni del software:

- Scoperta di Anomalie e Difetti:** L'obiettivo principale delle ispezioni del software è individuare anomalie e difetti nel materiale documentario o nella rappresentazione del sistema, come specifiche, progettazione, dati di configurazione, dati di test, ecc. Gli ispettori cercano attivamente problemi che richiedono correzioni.

2. **Non Richiedono l'Esecuzione del Sistema:** A differenza del testing del software, le ispezioni non richiedono l'esecuzione del sistema. Possono essere condotte prima dell'implementazione del software, durante la fase di progettazione o persino durante la fase di definizione dei requisiti. Questo significa che possono essere applicate in fasi iniziali del processo di sviluppo.
3. **Applicabili a Diverse Rappresentazioni del Sistema:** Le ispezioni del software possono essere applicate a qualsiasi rappresentazione del sistema, compresi requisiti, progettazione, dati di configurazione e dati di test. Questo le rende un'attività versatile nel processo di sviluppo del software.
4. **Efficaci nella Rilevazione di Errori:** Le ispezioni del software sono state dimostrate come un metodo efficace per scoprire errori nel programma. Attraverso l'esame critico dei documenti e delle rappresentazioni del sistema, è possibile individuare problemi che altrimenti potrebbero essere trascurati.

Successo dell'ispezione

Le ispezioni del software presentano alcune differenze rispetto al testing del software, che le rendono una pratica utile e complementare nel processo di verifica e validazione (V&V) del software:

1. **Rivelazione di Diversi Difetti:** Nelle ispezioni del software, è possibile scoprire molti difetti diversi in un'unica sessione di ispezione. Questo perché le ispezioni coinvolgono il confronto e l'esame critico dei documenti e delle rappresentazioni del sistema, il che può portare alla rilevazione di vari tipi di errori in diverse parti del materiale documentario. Nel testing del software, un difetto può mascherare un altro, il che significa che potrebbero essere necessarie più esecuzioni di test per scoprire diversi difetti.
2. **Conoscenza del Dominio e della Programmazione:** Gli ispettori delle ispezioni del software sono spesso esperti nel dominio di applicazione specifico e nella programmazione. Questo significa che hanno familiarità con i tipi di errori che comunemente si verificano in quel dominio o con le pratiche comuni di programmazione. Questa conoscenza approfondita li aiuta a individuare difetti in modo più efficace rispetto a un test generico.

Ispezioni e test

È importante sottolineare che le ispezioni e il testing del software sono tecniche complementari, non in competizione tra loro, nel processo di verifica e validazione (V&V) del software. Ecco alcune considerazioni chiave:

1. **Complementarietà:** Le ispezioni e il testing sono entrambi importanti per garantire la qualità del software, ma si concentrano su aspetti diversi del processo di V&V. Le ispezioni sono particolarmente utili per verificare la conformità con specifiche tecniche e possono individuare difetti in documenti e rappresentazioni del sistema. Il testing, d'altra parte, è essenziale per verificare il comportamento effettivo del software in esecuzione, testando requisiti funzionali e non funzionali.

2. Utilizzo Combinato: La migliore pratica è utilizzare sia ispezioni che testing durante il processo di V&V. Le ispezioni possono individuare difetti nelle specifiche e nei documenti prima dell'implementazione, consentendo una correzione preventiva. Il testing è essenziale per verificare il comportamento reale del software. Insieme, queste due tecniche coprono un'ampia gamma di aspetti della qualità del software.

3. Limitazioni delle Ispezioni: Le ispezioni possono verificare la conformità con specifiche tecniche e requisiti documentati ma potrebbero non essere in grado di verificare la conformità con le reali esigenze del cliente. Inoltre, le ispezioni non sono adatte per testare caratteristiche non funzionali come prestazioni, usabilità, sicurezza, ecc.

Ispezioni del programma

Le ispezioni del software sono caratterizzate da un approccio formalizzato alle revisioni dei documenti e sono specificamente orientate alla rilevazione dei difetti, non alla loro correzione. Ecco alcune delle principali caratteristiche delle ispezioni del software:

- 1. Approccio Formalizzato:** Le ispezioni del software seguono un processo strutturato e formalizzato per condurre le revisioni dei documenti e delle rappresentazioni del sistema. Questo processo definisce chiaramente i passaggi da seguire, le regole e le responsabilità degli ispettori.
- 2. Orientate alla Rilevazione dei Difetti:** La principale finalità delle ispezioni del software è rilevare difetti o problemi nei materiali documentari, come specifiche, progettazione, codice, ecc. Questo significa che gli ispettori cercano attivamente problemi senza preoccuparsi di risolverli durante l'ispezione stessa.
- 3. Tipi di Difetti:** I difetti individuati durante le ispezioni del software possono includere errori logici, anomalie nel codice che potrebbero indicare una condizione errata (ad esempio, una variabile non inizializzata) o non conformità agli standard del processo.
- 4. Non Correzione durante l'Ispezione:** Durante un'ispezione, l'obiettivo principale è individuare difetti e problemi, non risolverli. La correzione dei difetti individuati può avvenire in fasi successive del processo di sviluppo.

Presupposti per l'ispezione

Per condurre ispezioni del software in modo efficace, ci sono alcune condizioni e considerazioni importanti da tenere in considerazione:

- 1. Specifiche Precise:** È essenziale avere una specifica precisa e ben definita del software oggetto dell'ispezione. Senza una specifica chiara, gli ispettori potrebbero avere difficoltà nell'identificare i difetti o i problemi.
- 2. Conoscenza degli Standard dell'Organizzazione:** I membri del team incaricati dell'ispezione devono essere familiari con gli standard dell'organizzazione. Questi standard possono riguardare la formattazione dei documenti, le convenzioni di denominazione, le procedure di revisione, ecc.

3. **Disponibilità di Codice o Altre Rappresentazioni del Sistema:** È necessario avere accesso a codice sorgente o altre rappresentazioni del sistema, come specifiche di progettazione o documenti di requisiti. Questi materiali sono oggetto dell'ispezione.
4. **Checklist degli Errori:** È utile preparare una checklist degli errori o dei problemi noti che gli ispettori possono utilizzare durante l'ispezione. Questa checklist può aiutare a guidare l'analisi e garantire che nessun aspetto critico venga trascurato.
5. **Accettazione della Spesa Iniziale:** La gestione dell'organizzazione deve comprendere che le ispezioni del software possono comportare un aumento dei costi nelle fasi iniziali del processo di sviluppo. Tuttavia, questo investimento iniziale spesso porta a risparmi significativi in termini di correzioni e problemi rilevati in fasi successive.
6. **Non Uso per Valutazione del Personale:** Le ispezioni del software non dovrebbero essere utilizzate come strumento per valutare o giudicare il personale. L'obiettivo principale delle ispezioni è la rilevazione dei difetti e il miglioramento della qualità del software, non la valutazione delle prestazioni individuali.

Rispettare queste condizioni può contribuire a garantire che le ispezioni del software siano condotte in modo efficace e producano risultati significativi nel miglioramento della qualità del software.

Procedura di ispezione

Il processo di ispezione del software segue generalmente una serie di passaggi chiave, che possono includere:

1. **Presentazione dell'Overview del Sistema:** L'ispettore del software o il responsabile dell'ispezione presenta un'overview del sistema o dell'elemento del software che verrà ispezionato. Questo può includere una panoramica dei requisiti, della progettazione o del codice.
2. **Distribuzione dei Materiali:** I materiali pertinenti, come il codice sorgente, i documenti di progettazione o altre rappresentazioni del sistema, vengono distribuiti in anticipo all'ispettore o al team di ispezione. Gli ispettori esaminano questi materiali in preparazione all'ispezione.
3. **Ispezione:** L'ispezione effettiva viene condotta dal team di ispezione. Durante l'ispezione, gli ispettori esaminano attentamente i materiali alla ricerca di difetti, errori di logica, non conformità agli standard e altri problemi.
4. **Registrazione degli Errori:** Gli errori scoperti durante l'ispezione vengono registrati in modo dettagliato. Questo può includere la natura dell'errore, la sua posizione nei materiali, la sua gravità e altre informazioni pertinenti.
5. **Modifiche e Correzioni:** Dopo che l'ispezione è stata completata e gli errori sono stati identificati, vengono apportate modifiche per correggere gli errori scoperti. Queste correzioni vengono fatte per garantire che il software sia conforme ai requisiti e agli standard.

6. Eventuale re-ispezione: In alcuni casi, potrebbe essere necessaria una re-ispezione se ci sono stati errori significativi o se le modifiche apportate sono state sostanziali. La re-ispezione serve a verificare che gli errori siano stati correttamente risolti e che il software sia ora conforme.

Questo processo di ispezione del software è finalizzato a identificare e risolvere difetti nel software e migliorare la sua qualità complessiva. La collaborazione tra il team di ispezione, gli sviluppatori e altri stakeholder è fondamentale per il successo di questo processo.

Utilizzo dell'analisi statica automatica

Le ispezioni del software sono particolarmente preziose quando si utilizzano linguaggi come C, che hanno una debole verifica dei tipi e quindi molti errori possono sfuggire al compilatore. In linguaggi con una debole verifica dei tipi, gli errori possono verificarsi a causa di incompatibilità nei tipi di dati utilizzati, e questi errori potrebbero non essere rilevati durante la compilazione.

Tuttavia, in linguaggi come Java, che dispongono di una verifica dei tipi più rigorosa, molti errori di tipo vengono rilevati durante la compilazione stessa. Ciò significa che è meno probabile che si verifichino errori di tipo nei codici Java rispetto a quelli scritti in C. Di conseguenza, le ispezioni possono essere meno cost-effective per linguaggi con una forte verifica dei tipi come Java.

In generale, l'efficacia delle ispezioni del software può variare in base al linguaggio di programmazione utilizzato e alla complessità del software stesso. Tuttavia, indipendentemente dal linguaggio, le ispezioni possono comunque rivelare altri tipi di difetti, come errori logici o discrepanze rispetto ai requisiti, che non sono direttamente correlati alla verifica dei tipi. Pertanto, le ispezioni rimangono una pratica importante per migliorare la qualità del software in qualsiasi contesto di sviluppo.

Verifica e metodi formali

I metodi formali sono una potente tecnica di verifica statica che può essere utilizzata quando si dispone di una specifica matematica dettagliata del sistema. Ecco alcune considerazioni chiave sui metodi formali:

1. **Specifiche Matematiche:** I metodi formali richiedono una specifica matematica precisa del sistema. Questa specifica definisce in modo rigoroso il comportamento desiderato del sistema, utilizzando notazioni matematiche o logiche.
2. **Verifica Statica:** I metodi formali sono una forma di verifica statica, il che significa che l'analisi viene eseguita senza l'esecuzione effettiva del software. Questa analisi si basa sulla specifica matematica e può rilevare errori o problemi potenziali nel software.
3. **Analisi Matematica Dettagliata:** L'analisi matematica dei metodi formali è estremamente dettagliata. Gli esperti esaminano in modo rigoroso la specifica matematica e utilizzano tecniche matematiche avanzate per dimostrare che il software rispetta questa specifica.

4. **Formulazione di Argomenti Formali:** Nel contesto dei metodi formali, vengono spesso formulati argomenti formali per dimostrare che il software rispetta la specifica. Questi argomenti possono includere dimostrazioni matematiche che il software soddisfa tutte le condizioni specificate nella specifica.
5. **Ultima Forma di Verifica:** I metodi formali sono considerati una delle forme più rigorose di verifica. Quando vengono applicati correttamente, possono fornire una certezza molto alta che il software sia corretto rispetto alla specifica.

Tuttavia, l'uso dei metodi formali richiede competenze matematiche avanzate e può essere molto dispendioso in termini di tempo e risorse. Di conseguenza, viene spesso utilizzato per sistemi critici in cui la sicurezza e l'affidabilità sono di massima importanza, come nell'industria aerospaziale o nella produzione di dispositivi medici.

Sviluppo software per camere bianche

Il processo Cleanroom è un approccio allo sviluppo del software che si concentra sulla prevenzione dei difetti anziché sulla loro rimozione. Prende il nome dalla sala pulita (cleanroom) utilizzata nella produzione di semiconduttori, in cui è fondamentale evitare la contaminazione per ottenere risultati affidabili. Ecco alcune caratteristiche chiave del processo Cleanroom:

1. **Sviluppo Incrementale:** Il processo Cleanroom si basa su uno sviluppo incrementale, il che significa che il software viene costruito in piccoli passi iterativi. Ogni incremento è rigorosamente verificato prima di procedere al successivo.
2. **Specifiche Formali:** Le specifiche formali del software vengono utilizzate per definire in modo preciso il comportamento del sistema. Queste specifiche sono matematicamente rigorose e forniscono una base solida per la progettazione e lo sviluppo.
3. **Verifica Statica con Argomenti di Correttezza:** Una parte significativa della verifica avviene in modo statico attraverso l'uso di argomenti di correttezza formale. Gli esperti esaminano la specifica e formulano argomenti formali per dimostrare che il software rispetta la specifica.
4. **Test Statistico:** Una volta che il software è stato implementato e verificato staticamente, viene sottoposto a test statistico per determinare la sua affidabilità. Questi test utilizzano campioni di dati per valutare le prestazioni del software e la sua conformità agli standard di affidabilità.

L'obiettivo principale del processo Cleanroom è creare software con la massima affidabilità possibile, evitando la presenza di difetti fin dall'inizio. Questo approccio è particolarmente adatto per applicazioni critiche in cui la sicurezza e l'affidabilità sono di primaria importanza, come nel settore aerospaziale o nella produzione di software per dispositivi medici.

Maggiori dettagli sul test del software

Il processo di test

Nel contesto del testing del software, ci sono due tipi di testing importanti: il testing dei componenti e il testing del sistema. Ecco una panoramica di entrambi:

Testing dei Componenti:

- Il testing dei componenti riguarda la verifica delle singole unità o moduli di un software. Un componente può essere una funzione, una classe o qualsiasi altra unità di codice autonomo.
- Di solito, il responsabile del testing dei componenti è lo stesso sviluppatore del componente. Questo è noto come "white-box testing" o "glass-box testing", poiché il tester ha familiarità con il codice sorgente e la struttura interna del componente.
- I test dei componenti sono spesso derivati dall'esperienza del programmatore e mirano a verificare che ogni componente funzioni correttamente in isolamento.

Testing del Sistema:

- Il testing del sistema coinvolge il testing di gruppi di componenti integrati per creare un sistema completo o un sottosistema. L'obiettivo è verificare che i componenti lavorino insieme in modo coerente.
- Solitamente, il testing del sistema è responsabilità di un team di testing indipendente, separato dagli sviluppatori del software. Questo aiuta a identificare errori o problemi che potrebbero non essere stati evidenti per gli sviluppatori.
- I test del sistema sono basati sulla specifica del sistema, che definisce il comportamento previsto dell'intero sistema o sottosistema.

Entrambi i tipi di testing sono essenziali per garantire la qualità del software. Il testing dei componenti si concentra sulla correttezza dei singoli elementi, mentre il testing del sistema verifica l'integrazione e il funzionamento complessivo del software. Combinate, queste attività aiutano a individuare e correggere difetti in modo efficace prima che il software venga rilasciato.

Testare gli obiettivi del processo

Nel contesto del testing del software, ci sono due tipi principali di testing che svolgono ruoli diversi:

Validation Testing (Testing di Convalida):

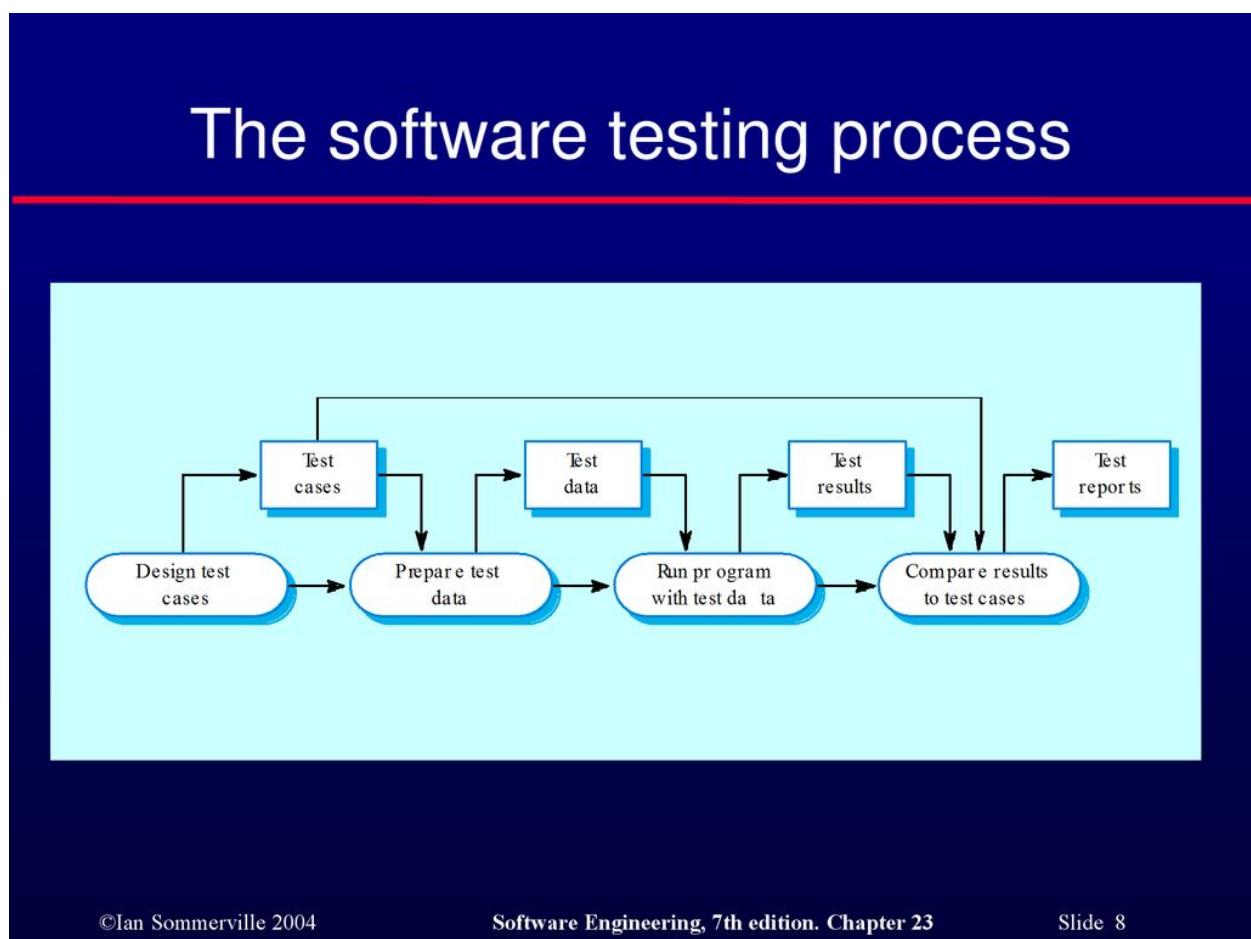
- Il testing di convalida è finalizzato a dimostrare al cliente e agli sviluppatori del sistema che il software soddisfa i suoi requisiti e le aspettative degli utenti.
- Un test di convalida riuscito dimostra che il sistema funziona come previsto e che è idoneo all'uso previsto.
- In altre parole, il testing di convalida verifica se il software "sta costruendo il prodotto giusto".

Defect Testing (Testing dei Difetti):

- Il testing dei difetti ha l'obiettivo di scoprire errori o difetti nel software, dove il comportamento del sistema è incorretto o non conforme alla sua specifica.
- Un test dei difetti riuscito è un test che fa eseguire al sistema un comportamento errato e quindi mette in evidenza un difetto nel sistema.
- In sintesi, il testing dei difetti evidenzia la presenza di difetti, ma non la loro assenza. Rivelando difetti nel software, il testing dei difetti aiuta a individuare problemi che richiedono correzioni.

Entrambi questi tipi di testing sono importanti per garantire la qualità del software. La convalida verifica che il software soddisfi i requisiti e le aspettative degli utenti, mentre il testing dei difetti identifica problemi nel comportamento del software. Insieme, questi approcci aiutano a migliorare l'affidabilità e l'idoneità del software all'uso previsto.

Il processo di test del software



Testare le politiche

L'idea che solo il testing esaustivo possa dimostrare che un programma sia privo di difetti è un concetto importante ma spesso irraggiungibile nella pratica. Ecco alcune considerazioni su questo concetto:

1. Testing Esaustivo: Il testing esaustivo, noto anche come "testing completo" o "testing di tutti i percorsi", implica la verifica di tutte le possibili combinazioni di input e scenari di esecuzione del software. Questo approccio teoricamente garantirebbe che nessun difetto rimanga non rilevato.

2. Impossibilità del Testing Esaustivo: Tuttavia, il testing esaustivo è spesso impossibile da realizzare nella pratica per diverse ragioni:

- Complessità: La maggior parte dei software è così complessa che il numero di possibili combinazioni di input diventa enorme. Eseguire test per tutte queste combinazioni richiederebbe un tempo e una risorsa infiniti.
- Risorse Limitate: Le risorse (tempo, budget, personale) per il testing sono sempre limitate, quindi è necessario pianificare e prioritizzare il testing in modo efficace.
- Comportamento Inaspettato: A volte il software può comportarsi in modi inaspettati o imprevedibili, rendendo difficile la definizione di tutti i casi di test.

3. Testing Policies: Le politiche di testing definiscono l'approccio da seguire nella selezione dei test di sistema. Queste politiche possono aiutare a garantire che almeno alcuni aspetti critici del software vengano testati in modo rigoroso. Le politiche possono includere requisiti come testare tutte le funzioni accessibili tramite i menu o testare combinazioni di funzioni accessibili tramite lo stesso menu.

Test del sistema

Il testing di integrazione è una parte importante del processo di sviluppo del software e coinvolge l'integrazione dei componenti per creare un sistema o un sotto-sistema. Ecco alcune informazioni chiave sul testing di integrazione:

1. Scopo del Testing di Integrazione: Il suo scopo principale è verificare che i diversi componenti del software, quando integrati, funzionino correttamente insieme. Questo è importante perché il software spesso è composto da molte parti diverse, e il loro comportamento congiunto è fondamentale per il funzionamento complessivo del sistema.

2. Fasi del Testing di Integrazione: Il testing di integrazione si svolge generalmente in due fasi principali:

- **Testing di Integrazione:** In questa fase, il team di testing ha accesso al codice sorgente del sistema e testa il sistema mentre i componenti vengono integrati. L'obiettivo è individuare eventuali problemi di interoperabilità tra i componenti e garantire che essi si interfaccino correttamente.
- **Testing di Rilascio:** In questa fase, il team di testing testa l'intero sistema come una "scatola nera", senza avere accesso al codice sorgente. L'obiettivo è verificare che il sistema funzioni come previsto dall'utente finale.

3. Obiettivi del Testing di Integrazione: Durante il testing di integrazione, si cercano difetti legati all'interazione tra i componenti, come errori di comunicazione, dati non correttamente trasmessi o problemi di sincronizzazione. Inoltre, si verifica che le funzionalità del sistema funzionino come previsto quando i componenti vengono integrati.

4. Ambito del Testing: Il livello di dettaglio e il grado di integrazione testati possono variare. Ad esempio, il testing di integrazione può essere condotto a livello di modulo (singoli componenti) o a livello di sistema (integrazione di tutti i componenti). La scelta dipende dalle esigenze del progetto.

In generale, il testing di integrazione è una pratica cruciale per garantire che il software funzioni senza problemi quando tutti i suoi componenti sono combinati. Aiuta a individuare e risolvere problemi di integrazione prima che il software venga rilasciato agli utenti finali.

Test d'integrazione

L'integrazione del software può essere affrontata utilizzando diverse strategie, tra cui l'integrazione dall'alto verso il basso (top-down) e l'integrazione dal basso verso l'alto (bottom-up). Ecco una spiegazione di queste strategie:

1. Top-Down Integration (Integrazione dall'Alto verso il Basso): In questa strategia, si inizia sviluppando la struttura principale o il "scheletro" del sistema, noto come "sistema stub" o "driver." Quindi, i componenti individuali vengono gradualmente aggiunti e integrati nella struttura principale. Questa strategia segue una gerarchia, partendo dal livello più alto del sistema e scendendo ai livelli inferiori.

- **Vantaggi:** Questo approccio può aiutare a identificare e risolvere problemi architetturali o di progettazione all'inizio del processo di integrazione. Inoltre, è utile quando si desidera testare prima le funzionalità principali del sistema.
- **Svantaggi:** Potrebbe richiedere più tempo per integrare tutti i componenti, in quanto le parti inferiori del sistema potrebbero dover aspettare che la struttura principale sia pronta. Inoltre, potrebbe non essere appropriato per sistemi in cui i componenti inferiori sono già completi o critici.

2. Bottom-Up Integration (Integrazione dal Basso verso l'Alto): In questa strategia, si inizia integrando i componenti di base o infrastrutturali, noti come "sottomoduli" o "componenti leaf." Successivamente, i componenti funzionali vengono gradualmente aggiunti sopra di essi. Questa strategia parte dai componenti di base e si muove verso l'alto nella gerarchia del sistema.

- **Vantaggi:** Questo approccio può essere più veloce nell'integrare i componenti, poiché i componenti di base sono spesso più semplici e possono essere pronti prima dei componenti funzionali. Inoltre, è adatto per sistemi in cui i componenti di base sono essenziali.
- **Svantaggi:** Potrebbero emergere problemi di progettazione o architetturali solo quando i componenti funzionali vengono integrati, il che potrebbe richiedere ulteriori sforzi per risolverli.

3. Incremental Integration (Integrazione Incrementale): In questa strategia, l'integrazione viene eseguita gradualmente e incrementalmente man mano che i componenti sono pronti. Non è vincolata a una struttura specifica dall'alto verso il basso o dal basso verso l'alto e mira a semplificare la localizzazione degli errori.

- **Vantaggi:** Questo approccio permette di iniziare l'integrazione quando i componenti sono pronti, indipendentemente dalla loro posizione nella gerarchia. Riduce la necessità di aspettare che una parte specifica del sistema sia completata prima di iniziare l'integrazione.
- **Svantaggi:** La gestione dell'ordine e della sequenza di integrazione può richiedere una pianificazione attenta per garantire un processo senza problemi.

La scelta tra queste strategie dipende dalle specifiche esigenze del progetto, dalla complessità del sistema e dalla disponibilità dei componenti. Spesso, nel mondo reale, si utilizzano combinazioni di queste strategie per gestire l'integrazione del software in modo efficiente.

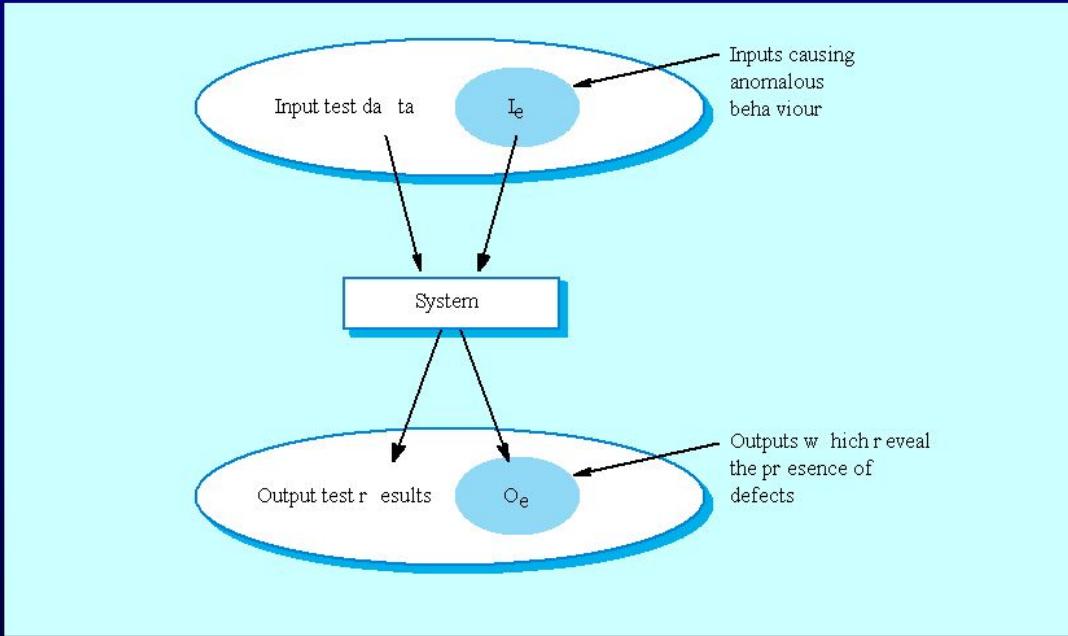
Test di rilascio

Release testing è il processo di testare una versione del sistema che verrà distribuita ai clienti. L'obiettivo principale di questo tipo di testing è aumentare la fiducia del fornitore (chi sviluppa il software) che il sistema soddisfi i suoi requisiti. Ecco alcuni punti chiave relativi al release testing:

1. **Obiettivo Primario:** L'obiettivo principale del release testing è garantire che il sistema sia pronto per essere distribuito ai clienti. Questo significa assicurarsi che il software sia stabile, funzionante e che soddisfi i requisiti specificati nella documentazione del sistema.
2. **Black-Box o Functional Testing:** Nel release testing, di solito si utilizza l'approccio del black-box o functional testing. Questo significa che i tester si concentrano sull'esecuzione delle funzionalità del sistema senza conoscere i dettagli interni dell'implementazione. Si basano sulla specifica del sistema per progettare e eseguire i test.
3. **Accesso alla Specifica:** I tester hanno accesso alla specifica del sistema, che è un documento che descrive il comportamento atteso del sistema e i requisiti che deve soddisfare. Utilizzano questa specifica come base per pianificare e condurre i test.
4. **Ignoranza dell'Implementazione:** Nel release testing, i tester non devono conoscere i dettagli tecnici dell'implementazione del sistema. Questo li aiuta a valutare il sistema dal punto di vista dell'utente finale e a concentrarsi sulle funzionalità e sul comportamento previsto.
5. **Identificazione dei Problemi:** Se durante il release testing vengono rilevati problemi, come bug o difetti, questi vengono documentati e successivamente affrontati dal team di sviluppo. È importante risolvere questi problemi prima di distribuire il sistema ai clienti.
6. **Fase Finale del Ciclo di Vita:** Il release testing è di solito una delle ultime fasi del ciclo di vita dello sviluppo del software, che precede la distribuzione del sistema ai clienti. È fondamentale per garantire che il sistema sia stabile e che soddisfi i requisiti prima di essere reso disponibile per l'utilizzo effettivo.

Test della scatola nera

Black-box testing



©Ian Sommerville 2004

Software Engineering, 7th edition. Chapter 23

Slide 15

Linee guida per i test

Le linee guida per il testing sono suggerimenti per il team di testing al fine di aiutarli a selezionare test che rivelino difetti nel sistema. Questi suggerimenti sono utili per progettare test efficaci che possono mettere alla prova il sistema in modo completo e identificare potenziali problemi. Di seguito, sono forniti alcuni esempi di linee guida per il testing:

1. **Scegliere input che costringano il sistema a generare tutti i messaggi di errore:** Questo significa progettare input che mettano alla prova tutte le possibili condizioni di errore nel sistema. Ad esempio, se il sistema gestisce dati numerici, è importante testare input non numerici per verificare come il sistema reagisce.
2. **Progettare input che causino il superamento dei buffer:** I buffer sono aree di memoria utilizzate dal software per archiviare dati temporanei. Forzare input che superano i limiti di questi buffer può rivelare problemi di gestione della memoria o potenziali violazioni della sicurezza.
3. **Ripetere lo stesso input o serie di input più volte:** Questo può mettere alla prova la stabilità e l'affidabilità del sistema durante un uso prolungato o in condizioni di carico elevato.
4. **Forzare la generazione di output non validi:** Verificare come il sistema gestisce situazioni in cui dovrebbe generare output non validi o risultati incoerenti. Questo può rivelare problemi di logica o di controllo nel software.

5. **Forzare risultati di calcolo troppo grandi o troppo piccoli:** Testare come il sistema gestisce valori estremamente grandi o piccoli. Questo può rivelare problemi di overflow o underflow nei calcoli.

Queste sono solo alcune delle linee guida che possono essere seguite nel processo di testing. L'obiettivo principale è sviluppare una strategia di testing completa che copra tutti gli aspetti critici del sistema e identifichi i difetti in modo efficace. Le linee guida possono variare a seconda del tipo di sistema, dei requisiti specifici e degli obiettivi di testing.

Casi d'uso

I casi d'uso possono costituire una base importante per derivare i test di un sistema. Possono aiutare a identificare le operazioni da testare e a progettare i casi di test necessari. Ecco come i casi d'uso possono essere utilizzati per sviluppare i test di sistema:

1. **Identificazione delle operazioni da testare:** Ogni caso d'uso rappresenta un'interazione specifica tra l'utente e il sistema. Ogni passo all'interno di un caso d'uso corrisponde a un'operazione che può essere testata. Identificando i passi chiave all'interno dei casi d'uso, è possibile determinare quali operazioni devono essere testate.
2. **Progettazione dei casi di test:** Una volta identificate le operazioni da testare, è possibile progettare i casi di test corrispondenti. Ogni caso di test dovrebbe coprire uno scenario specifico o un flusso di lavoro all'interno del caso d'uso. Questi casi di test dovrebbero includere input appropriati e aspettative di output.
3. **Creazione dei dati di input e definizione delle aspettative:** Per ogni caso di test, è necessario definire i dati di input necessari e specificare quali output ci si aspetta dal sistema. Questi dati di input possono essere derivati dai passi del caso d'uso e dalle informazioni associate.
4. **Verifica dell'interazione tra operazioni:** Nei casi d'uso complessi o nei sistemi che coinvolgono interazioni tra più operazioni, è possibile utilizzare diagrammi di sequenza associati per identificare come le operazioni interagiscono tra loro. Questo può aiutare a definire i test che verificano la corretta sequenza e l'interazione delle operazioni.

Test delle prestazioni

La parte di testing di rilascio può coinvolgere anche il testing delle proprietà emergenti di un sistema, come le prestazioni e l'affidabilità. Ecco alcune considerazioni su come vengono gestiti questi aspetti:

1. **Testing delle prestazioni:** Il testing delle prestazioni è essenziale per assicurarsi che il sistema funzioni in modo efficiente e soddisfi i requisiti di prestazioni. Di solito, questo coinvolge la pianificazione di una serie di test in cui il carico di lavoro viene gradualmente aumentato fino a quando le prestazioni del sistema diventano inaccettabili. Questo tipo di test può rivelare problemi come il sovraccarico della CPU, il collasso del server o i tempi di risposta lenti.
2. **Testing dell'affidabilità:** Il testing dell'affidabilità mira a valutare la stabilità e la robustezza del sistema. Questo può coinvolgere test di resistenza, in cui il sistema viene sottoposto a

stress prolungati per identificare eventuali fallo o problemi di affidabilità. Inoltre, i test di recupero da errori possono essere eseguiti per verificare come il sistema gestisce le situazioni di errore e il ripristino.

3. **Testing di sicurezza:** Se il sistema gestisce dati sensibili o operazioni critiche per la sicurezza, è importante includere il testing di sicurezza come parte del processo di rilascio. Questo tipo di test mira a identificare vulnerabilità e fallo di sicurezza che potrebbero essere sfruttate da potenziali minacce.
4. **Testing di scalabilità:** Se il sistema è destinato a crescere nel tempo o deve gestire carichi di lavoro variabili, è importante testare la sua capacità di scalare in modo adeguato. Questo può coinvolgere test di scalabilità orizzontale o verticale per valutare come il sistema gestisce l'aumento del traffico o dei dati.
5. **Testing delle interfacce:** Se il sistema interagisce con altri sistemi esterni o componenti, è fondamentale testare le interfacce per garantire che le comunicazioni avvengano in modo corretto e affidabile.
6. **Testing del carico:** Il testing del carico mira a valutare come il sistema si comporta sotto carichi di lavoro pesanti. Questo può rivelare problemi di risorse, come la saturazione della banda o il consumo eccessivo di memoria.

Prove di stress

Il testing di stress è un'attività critica nel processo di verifica e validazione del software, specialmente per i sistemi complessi e distribuiti. Ecco alcune considerazioni aggiuntive sul testing di stress:

1. **Esercizio oltre il carico massimo previsto:** Il testing di stress comporta l'esecuzione del sistema oltre il carico massimo previsto durante l'operazione normale. Questo può aiutare a identificare i limiti del sistema e i potenziali problemi che potrebbero emergere in situazioni eccezionali o di sovraccarico.
2. **Verifica del comportamento di fallimento:** Durante il testing di stress, è importante verificare il comportamento di fallimento del sistema. Un sistema ben progettato dovrebbe evitare il fallimento catastrofico, il che significa che, anche quando sottoposto a stress estremi, dovrebbe essere in grado di gestire il carico in modo tale da evitare la perdita di servizio o dati inaccettabili.
3. **Rilevazione di difetti nascosti:** Il testing di stress può rivelare difetti nascosti che non emergono durante l'operazione normale del sistema. Ad esempio, potrebbero emergere problemi di concorrenza, perdita di dati o comportamenti imprevisti sotto stress.
4. **Rilevamento di problemi di scalabilità:** I sistemi distribuiti, in particolare, possono mostrare problemi gravi quando la rete è sovraccarica o quando si verificano problemi di comunicazione. Il testing di stress è cruciale per identificare questi problemi di scalabilità e risolverli in modo appropriato.
5. **Rispetto ai requisiti di affidabilità:** Il testing di stress è in linea con i requisiti di affidabilità del sistema. Un sistema affidabile dovrebbe essere in grado di gestire situazioni di stress

senza subire danni irreversibili o causare interruzioni critiche nei servizi.

Test delle componenti

Il testing delle componenti o delle unità è una fase critica del processo di verifica e validazione del software. Ecco alcune informazioni aggiuntive sul testing delle componenti o delle unità:

1. **Processo di test delle componenti:** Questo processo si concentra sulla verifica delle singole componenti software in isolamento. Le componenti possono essere funzioni, metodi, classi o anche componenti composti più complessi. L'obiettivo è assicurarsi che ciascun componente funzioni correttamente e produca i risultati desiderati.
2. **Processo di testing dei difetti:** Il testing delle componenti è principalmente un processo di testing dei difetti. L'obiettivo principale è scoprire eventuali difetti o errori nelle componenti. Questi difetti possono includere problemi di logica, errori di implementazione, errori di sintassi o qualsiasi altra anomalia che potrebbe influenzare il comportamento della componente.
3. **Livelli di componenti:** La complessità delle componenti può cambiare; possono essere semplici funzioni o metodi che eseguono operazioni specifiche, oppure classi che contengono attributi e metodi. Anche componenti composti più complesse possono essere soggette a testing delle unità se hanno interfacce definite per l'accesso alla loro funzionalità.
4. **Isolamento delle componenti:** Durante il testing delle componenti, è importante isolare il componente in esame dagli altri componenti del sistema. Questo significa che qualsiasi dipendenza esterna dovrebbe essere simulata o sostituita con mock o stub, in modo da concentrarsi esclusivamente sul componente in esame.
5. **Ripetibilità:** Il testing delle componenti dovrebbe essere ripetibile. Ciò significa che i risultati dei test dovrebbero essere coerenti quando si eseguono gli stessi test più volte. Questa ripetibilità aiuta a garantire che le componenti siano affidabili e stabili.
6. **Strumenti di testing:** Esistono strumenti di testing appositamente progettati per il testing dei componenti, che semplificano il processo e consentono l'automatizzazione dei test.

Test delle classi di oggetti

La copertura completa dei test di una classe è una parte essenziale del processo di testing delle unità. Ecco alcune considerazioni aggiuntive sulla copertura completa dei test delle classi:

1. **Test di tutte le operazioni associate:** La copertura completa dei test di una classe richiede che vengano testate tutte le operazioni o metodi associati a quell'oggetto o a quella classe. Questo include la chiamata e l'esecuzione di ciascun metodo con input appropriati per verificare che essi producano i risultati attesi.
2. **Impostazione e interrogazione degli attributi:** È importante testare anche l'impostazione e l'interrogazione di tutti gli attributi dell'oggetto. Questo assicura che gli attributi vengano gestiti correttamente e che le operazioni di get e set funzionino come previsto.

3. **Esercitare l'oggetto in tutti gli stati possibili:** Un oggetto può esistere in vari stati durante il suo ciclo di vita. È necessario esercitare l'oggetto in tutti questi stati per assicurarsi che si comporti correttamente in ciascuno di essi. Questo può includere situazioni di inizializzazione, stato normale e situazioni eccezionali.
4. **Ereditarietà e complessità:** Quando si utilizza l'ereditarietà nelle classi, il testing delle classi può diventare più complesso. Poiché le sottoclassi ereditano comportamenti e attributi dalle superclassi, è importante testare anche il comportamento delle sottoclassi in relazione alle superclassi. Questo può richiedere la progettazione di test più sofisticati per coprire tutti gli scenari possibili.
5. **Strumenti di test:** L'uso di strumenti di testing può semplificare notevolmente il processo di copertura completa dei test delle classi. Questi strumenti possono automatizzare i test e generare report dettagliati sulla copertura dei test.

Test dell'interfaccia

Il testing dell'interfaccia è una fase cruciale nel processo di verifica e validazione del software, con l'obiettivo principale di rilevare difetti derivanti da errori nelle interfacce o da assunzioni non valide sulle interfacce. Questo tipo di testing è particolarmente rilevante nello sviluppo orientato agli oggetti, poiché gli oggetti sono definiti dalle loro interfacce. Ecco alcune considerazioni aggiuntive sull'interface testing:

1. **Definizione dell'interfaccia:** Prima di condurre il testing dell'interfaccia, è essenziale avere una chiara comprensione delle interfacce coinvolte. Questo include l'identificazione delle interfacce di oggetti, classi o moduli e la comprensione delle loro specifiche e delle aspettative di utilizzo.
2. **Scenari di test:** Durante il testing dell'interfaccia, è importante progettare una serie di scenari di test che coprano tutti gli aspetti critici delle interfacce. Ciò potrebbe includere test di input validi e non validi, test di errori, test di limiti e test di performance delle interfacce.
3. **Isolamento delle interfacce:** Nell'ambito del testing dell'interfaccia, è possibile isolare le interfacce per testarle separatamente. Ciò consente di identificare eventuali problemi specifici legati a un'interfaccia particolare senza essere influenzati da altri componenti del sistema.
4. **Verifica dell'integrazione:** Inoltre, è importante testare come le diverse interfacce interagiscono tra loro quando vengono utilizzate in combinazione. Questo può rivelare problemi di compatibilità o conflitti tra le interfacce.
5. **Test di ereditarietà:** Nel contesto dell'ereditarietà, nelle classi o negli oggetti, è necessario verificare che l'ereditarietà delle interfacce funzioni correttamente. Ciò significa testare come le sottoclassi o gli oggetti derivati utilizzano e implementano le interfacce ereditate dalle superclassi.
6. **Strumenti di automazione:** Come per molti aspetti del testing del software, l'uso di strumenti di automazione può semplificare il processo di testing dell'interfaccia. Gli strumenti possono aiutare a generare casi di test, eseguire i test e raccogliere dati di test in modo efficiente.

7. Documentazione e tracciabilità: Assicurarsi di documentare i risultati del testing dell'interfaccia e di mantenere una tracciabilità completa tra i casi di test, i difetti rilevati e le interfacce coinvolte. Questo aiuta a gestire e risolvere eventuali problemi in modo più efficiente.

Errori dell'interfaccia

Gli errori legati all'interfaccia possono verificarsi per vari motivi e possono influenzare il comportamento del sistema. Ecco alcuni tipi comuni di errori legati all'interfaccia:

1. **Misuso dell'interfaccia:** Questo errore si verifica quando un componente chiamante utilizza un'altra interfaccia in modo scorretto. Ad esempio, potrebbe passare i parametri in un ordine errato o fornire dati non validi all'interfaccia. Questo può portare a comportamenti imprevisti o errori nel sistema.
2. **Malinteso dell'interfaccia:** In questo caso, il componente chiamante ha delle supposizioni errate sul comportamento dell'interfaccia chiamata. Queste supposizioni errate possono portare a errori di progettazione o a comportamenti non desiderati nel sistema. È importante che le interfacce siano ben documentate per evitare malintesi.
3. **Errori di tempismo:** Questo tipo di errore si verifica quando il componente chiamante e il componente chiamato operano a velocità diverse o seguono temporizzazioni diverse. Ciò può causare problemi come l'accesso a informazioni obsolete o la sincronizzazione inadeguata tra i componenti.

È fondamentale condurre test mirati per rilevare questi errori legati all'interfaccia durante il processo di sviluppo del software. Inoltre, una documentazione chiara delle interfacce e una buona comunicazione tra i team di sviluppo possono contribuire a prevenire o mitigare questi tipi di errori.

Linee guida per i test dell'interfaccia

Ecco alcune linee guida importanti per la progettazione dei test, che possono aiutare a rilevare e prevenire errori nei componenti del software:

1. **Test dei limiti:** Assicurarsi di progettare test in modo che i parametri passati a una procedura chiamata siano agli estremi dei loro intervalli validi. Questo può rivelare problemi di gestione dei limiti o di overflow/underflow.
2. **Test con valori estremi:** Testare sempre i parametri dei puntatori con puntatori nulli. Questo è particolarmente importante per evitare errori di dereferenziazione dei puntatori nulli.
3. **Test di fallimento:** Progettare test che inducano il componente a fallire. Questo è importante per verificare che il componente gestisca correttamente gli errori e le eccezioni.
4. **Test di stress:** Nel caso di sistemi basati su passaggio di messaggi, utilizzare test di stress per valutare come il sistema si comporta sotto carichi di lavoro intensi o con comunicazioni errate.
5. **Variazione dell'ordine di attivazione:** Nei sistemi a memoria condivisa, variare l'ordine in cui i componenti vengono attivati. Questo può rivelare problemi di concorrenza o di accesso simultaneo ai dati condivisi.

Queste linee guida aiutano a garantire una copertura completa dei casi di test e a individuare potenziali errori nei componenti del software. È importante pianificare accuratamente i test e includere una varietà di scenari per garantire che il software sia robusto e affidabile.

Progettazione del caso di prova

Il processo di progettazione dei casi di test è cruciale per garantire che il sistema venga testato in modo efficace ed efficiente. Ci sono diverse approccio per la progettazione dei casi di test, tra cui:

1. **Testing basato sui requisiti:** Questo approccio coinvolge la progettazione dei casi di test in base ai requisiti specificati per il sistema. Ogni requisito deve essere associato a uno o più casi di test che verificano se il requisito è stato soddisfatto. Questo assicura che il software sia testato rispetto alle specifiche richieste.
2. **Testing per partizioni:** In questo approccio, si suddivide l'input in diverse partizioni o categorie e si progettano casi di test per ogni categoria. Ad esempio, se un input può essere un numero intero positivo o negativo, progetteresti casi di test per entrambe le partizioni.
3. **Testing strutturale:** Questo approccio si concentra sulla progettazione dei casi di test in base alla struttura interna del codice sorgente. Può includere la creazione di casi di test che eseguono specifiche sezioni di codice o percorsi di esecuzione. È spesso utilizzato per testare la copertura del codice.

Ogni approccio ha i suoi vantaggi e può essere utilizzato in combinazione per garantire una copertura completa dei casi di test. La progettazione dei casi di test richiede una buona comprensione del sistema e dei suoi requisiti, nonché una pianificazione accurata per garantire che tutti gli aspetti critici siano testati.

Test basati sui requisiti

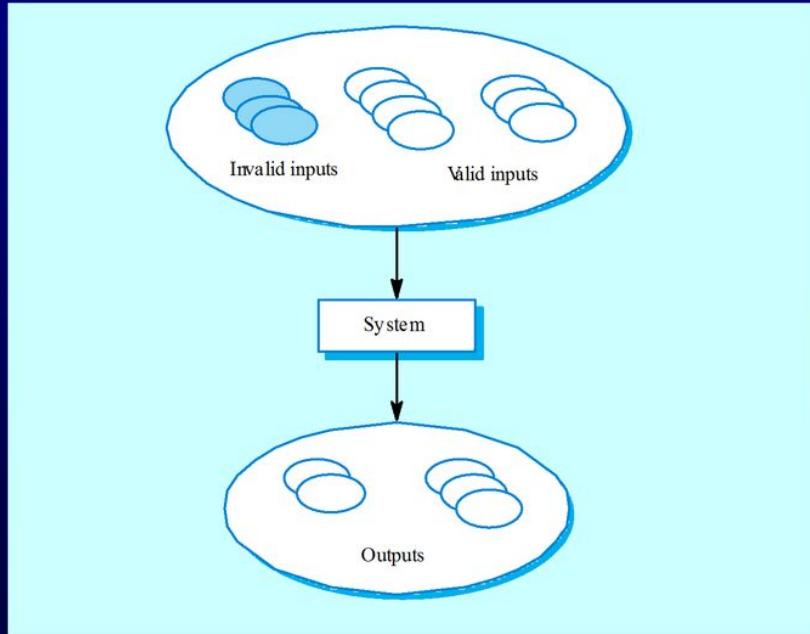
Il principio che i requisiti dovrebbero essere testabili è fondamentale nell'ingegneria dei requisiti. Questo significa che i requisiti dovrebbero essere formulati in modo tale da poter essere verificati attraverso test o altri mezzi. Quando i requisiti sono testabili, diventa possibile sviluppare casi di test specifici per ogni requisito, consentendo una valida validazione del sistema.

L'approccio di testing basato sui requisiti consiste nel prendere ciascun requisito e derivare un set di casi di test che coprano quel requisito in particolare. Questo aiuta a garantire che il software soddisfi effettivamente ciascun requisito e che tutti i requisiti siano testati. È un metodo sistematico per assicurarsi che il software sia in linea con ciò che è stato richiesto.

Tuttavia, è importante notare che il testing basato sui requisiti è solo uno dei metodi di progettazione dei casi di test. È spesso combinato con altri approcci come il testing strutturale e il testing per partizioni per ottenere una copertura completa dei casi di test.

Partizionamento per equivalenze

Equivalence partitioning



©Ian Sommerville 2004

Software Engineering, 7th edition. Chapter 23

Slide 31

Linee guida per i test (sequenze)

Le linee guida fornite riguardano l'approccio al testing di software che coinvolge sequenze o insiemi di dati, come ad esempio array o elenchi. Ecco come puoi applicare queste linee guida:

- 1. Testare con sequenze che hanno solo un valore:** Quando si testa il software, è importante includere casi in cui la sequenza contiene solo un elemento. Questo aiuterà a verificare se il software gestisce correttamente le situazioni con sequenze minime.
- 2. Usare sequenze di diverse dimensioni in test diversi:** Assicurati di progettare casi di test che coinvolgono sequenze di diverse dimensioni. Ad esempio, potresti testare una sequenza con tre elementi, una con dieci elementi e una con cento elementi per verificare che il software gestisca correttamente varie dimensioni di dati di input.
- 3. Derivare test in modo da includere il primo, il medio e l'ultimo elemento della sequenza:** È importante testare il comportamento del software con l'accesso al primo, al medio e all'ultimo elemento della sequenza. Questo può rivelare eventuali problemi legati all'indicizzazione o alla gestione dei dati in diverse parti della sequenza.
- 4. Testare con sequenze di lunghezza zero:** Non dimenticare di includere casi di test in cui la sequenza ha lunghezza zero, ovvero è vuota. Questo è importante per verificare che il software gestisca correttamente situazioni in cui non ci sono dati nella sequenza.

In generale, queste linee guida ti aiuteranno a progettare casi di test più completi ed efficaci quando lavori con dati sequenziali nel tuo software. Assicurati di adattare queste linee guida alle specifiche

esigenze e caratteristiche del tuo software per ottenere risultati di testing accurati.

Test del percorso

Il testing basato sui percorsi (path testing) è una tecnica che mira a garantire che ogni percorso attraverso il programma venga eseguito almeno una volta durante il processo di testing. Questo approccio è particolarmente utile per identificare potenziali errori nelle decisioni del programma e nella gestione del flusso di controllo. Ecco alcuni punti chiave riguardanti il testing basato sui percorsi:

1. **Obiettivo principale:** L'obiettivo principale del path testing è coprire tutti i possibili percorsi di esecuzione attraverso il programma. Questo significa che ogni possibile combinazione di ramificazioni condizionali e istruzioni deve essere testata almeno una volta.
2. **Flusso grafico del programma:** Il punto di partenza per il path testing è la creazione di un grafico di flusso del programma. Questo grafico rappresenta il programma come un insieme di nodi e archi, dove i nodi rappresentano le decisioni del programma e gli archi rappresentano il flusso di controllo tra le istruzioni.
3. **Nodi delle decisioni:** I nodi del grafico di flusso del programma corrispondono alle istruzioni del programma che contengono condizioni decisionali (ad esempio, istruzioni "if" o "switch"). Ogni possibile ramo di esecuzione viene rappresentato come un percorso tra i nodi di decisione.
4. **Creazione dei casi di test:** Una volta creato il grafico di flusso del programma, è possibile progettare casi di test che coprano ciascun percorso attraverso il programma. Ogni caso di test dovrebbe essere progettato in modo da attraversare un percorso specifico nel grafico.
5. **Esecuzione dei casi di test:** I casi di test vengono quindi eseguiti utilizzando il programma. Durante l'esecuzione, è importante tenere traccia dei percorsi effettivamente coperti dai test e dei risultati ottenuti.
6. **Identificazione di percorsi non coperti:** Se durante il processo di testing si scopre che alcuni percorsi non sono stati coperti, è necessario progettare ulteriori casi di test per garantire una copertura completa.

Il testing basato sui percorsi può essere un'approccio efficace per rilevare errori nel flusso di controllo del programma e garantire una maggiore affidabilità del software. Tuttavia, può richiedere un'analisi dettagliata del codice e la progettazione di un insieme completo di casi di test, il che può essere oneroso in termini di tempo e risorse.

Testare l'automazione

L'automazione dei test è un'importante pratica per ridurre il tempo e i costi associati alla fase di test del software. Alcuni punti chiave sull'automazione dei test includono:

1. **Costi della fase di test:** La fase di test può essere costosa in termini di tempo e risorse umane. L'automazione dei test mira a ridurre questi costi automatizzando l'esecuzione dei casi di test.

2. **Strumenti di automazione:** Esistono diverse suite di strumenti e framework progettati per l'automazione dei test. Un esempio noto è JUnit, che è ampiamente utilizzato per l'automazione dei test nell'ambito dello sviluppo Java.
3. **Sistemi aperti:** La maggior parte delle suite di automazione dei test sono progettate come sistemi aperti, in modo da poter essere adattate alle esigenze specifiche di un'organizzazione. Ciò consente di personalizzare i test in base ai requisiti del progetto.
4. **Integrazione:** Talvolta può essere difficile integrare completamente le suite di automazione dei test con altri ambienti di sviluppo e analisi chiusi. È importante pianificare attentamente l'integrazione per massimizzare i benefici dell'automazione dei test.

L'automazione dei test può migliorare notevolmente l'efficienza dei processi di test, consentendo una maggiore copertura dei test, una ripetizione più rapida dei test e una rilevazione più tempestiva dei difetti nel software. Tuttavia, richiede un investimento iniziale per la scrittura dei casi di test automatizzati e la configurazione degli strumenti.

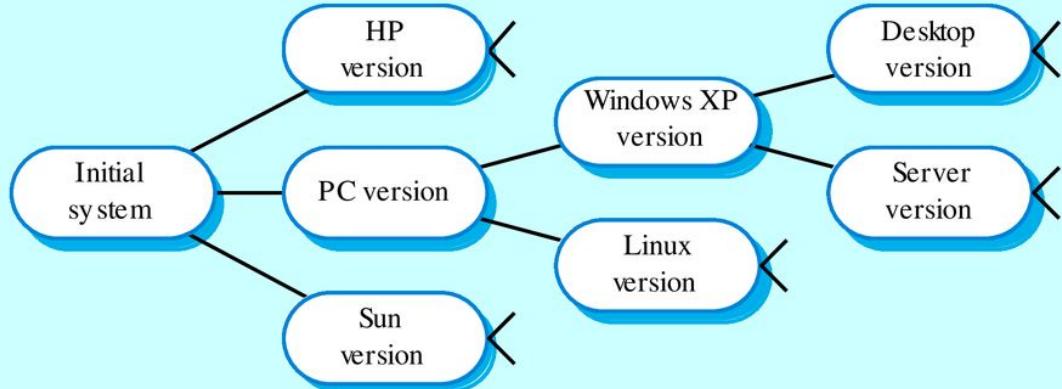
Gestione della configurazione

La gestione della configurazione (CM) è una parte critica del processo di sviluppo del software che mira a gestire i sistemi software in evoluzione. Ecco alcuni punti chiave sulla gestione della configurazione:

1. **Creazione di nuove versioni:** I sistemi software cambiano per adattarsi a diverse macchine, sistemi operativi, offrire funzionalità diverse o soddisfare requisiti specifici degli utenti. La CM si occupa di gestire questo processo di cambiamento.
2. **Controllo dei costi e degli sforzi:** La CM mira a controllare i costi e gli sforzi associati alle modifiche apportate a un sistema software. Questo è particolarmente importante quando si sviluppano sistemi complessi o critici.
3. **Procedure e standard:** La CM implica lo sviluppo e l'applicazione di procedure e standard per gestire i sistemi software. Questi includono il controllo delle versioni, la documentazione, la tracciabilità delle modifiche e altro ancora.
4. **Parte della gestione della qualità:** La CM è spesso vista come parte integrante della gestione della qualità generale del software. Assicura che i cambiamenti siano gestiti in modo coerente e che la qualità del software sia mantenuta.
5. **Baselines:** Quando un sistema software è rilasciato al controllo della configurazione, spesso è chiamato "baseline". Questo rappresenta uno stato iniziale da cui inizia il processo di evoluzione.

Famiglie di sistema

System families



7

Pianificazione della gestione della configurazione

La gestione della configurazione è un aspetto critico nella gestione di progetti software, specialmente per progetti complessi. Qui sono alcune considerazioni importanti nella pianificazione della gestione della configurazione:

- 1. Identificazione dei prodotti:** Prima di tutto, è necessario identificare tutti i prodotti generati dal processo software. Questi possono includere specifiche, design, codice sorgente, dati di test, manuali utente e altro ancora. Ogni prodotto deve essere identificato in modo univoco.
- 2. Controllo delle versioni:** Ogni prodotto deve essere sottoposto a controllo delle versioni per tenere traccia delle modifiche nel tempo. Questo assicura che si possano recuperare versioni precedenti dei prodotti, se necessario, e che sia possibile monitorare le modifiche apportate.
- 3. Baselines:** Le baselines rappresentano una snapshot dei prodotti in un punto specifico del tempo. Le baselines sono utilizzate come riferimenti stabili per confrontare le versioni successive dei prodotti.
- 4. Gestione delle modifiche:** È necessario stabilire un processo per gestire le modifiche ai prodotti. Questo include la registrazione delle modifiche, l'approvazione delle modifiche e la loro implementazione in modo controllato.
- 5. Gestione delle configurazioni:** La gestione delle configurazioni coinvolge la gestione dei prodotti, delle baselines e delle modifiche. Un sistema di gestione delle configurazioni (SCM) può essere utilizzato per automatizzare questo processo.

6. **Tracciamento delle modifiche:** È importante tenere traccia di chi ha apportato le modifiche, quando e perché. Questo può essere cruciale per la risoluzione dei problemi e per garantire la responsabilità.
7. **Documentazione:** Tutto il processo di gestione della configurazione deve essere documentato in modo chiaro e accessibile. Questo include procedure, politiche e registrazioni.

La pianificazione della gestione della configurazione è essenziale per garantire che tutti i prodotti software siano tracciati, controllati e gestiti in modo efficiente durante l'intero ciclo di vita del progetto.

Il piano CM

La gestione della configurazione (CM) è un aspetto critico nello sviluppo software per garantire che il software sia gestito, controllato e documentato in modo efficace durante tutto il suo ciclo di vita. Il documento che definisce le politiche e le procedure per la gestione della configurazione è noto come "Piano di Gestione della Configurazione" o "Piano di CM". Questo documento svolge un ruolo chiave nel garantire la coerenza e l'integrità del software.

Ecco alcuni degli elementi che un Piano di Gestione della Configurazione può includere:

1. **Tipi di documenti:** Specifica quali tipi di documenti devono essere gestiti all'interno del processo di CM, ad esempio specifiche dei requisiti, progettazione, codice sorgente, documentazione di test, ecc.
2. **Schema di denominazione dei documenti:** Stabilisce una convenzione di denominazione per i documenti, in modo che siano facilmente identificabili e rintracciabili.
3. **Responsabilità:** Chi è responsabile dell'attuazione delle procedure di CM e della creazione delle basi di configurazione.
4. **Politiche di controllo delle modifiche:** Descrive come le modifiche al software devono essere gestite, incluse le procedure di revisione, approvazione e registrazione delle modifiche.
5. **Gestione delle versioni:** Definisce come le diverse versioni del software saranno identificate, registrate e mantenute.
6. **Registrazione delle CM:** Specifica quali record devono essere mantenuti, inclusi registri di modifiche, registri di revisione e altre informazioni relative alla configurazione.
7. **Strumenti CM:** Indica gli strumenti o software utilizzati per assistere il processo di CM e le limitazioni nell'uso di tali strumenti.
8. **Processo di utilizzo degli strumenti:** Spiega come utilizzare gli strumenti CM per implementare le politiche e le procedure di CM.
9. **Database di CM:** Descrive come i dati di configurazione saranno registrati e gestiti in un database di CM dedicato.

10. **Altri aspetti:** Può includere altre informazioni pertinenti, come la gestione di software esterno, procedure di audit del processo, ecc.

Il Piano di Gestione della Configurazione è essenziale per garantire la coerenza e la tracciabilità del software durante il suo sviluppo, manutenzione e distribuzione. È uno strumento di controllo fondamentale per garantire che il software soddisfi i requisiti e le aspettative degli utenti.

Identificazione dell'elemento di configurazione

Gli schemi di denominazione dei documenti sono estremamente importanti in progetti di grandi dimensioni per garantire che i documenti siano univocamente identificati, organizzati e gestiti correttamente. Alcuni punti chiave da tenere a mente per la progettazione di uno schema di denominazione dei documenti includono:

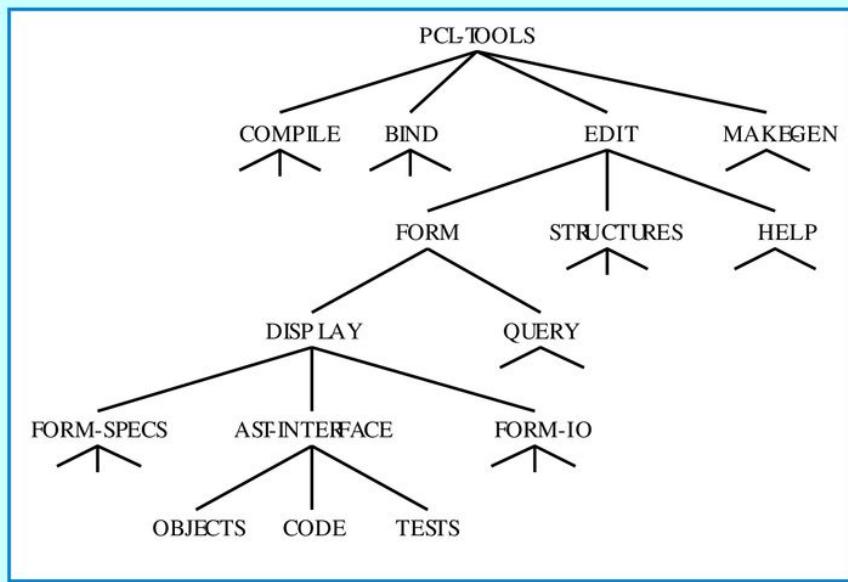
1. **Identificazione univoca:** Assicurarsi che ogni documento abbia un nome univoco. Questo è fondamentale per evitare confusione e problemi di gestione.
2. **Struttura gerarchica:** L'uso di un approccio gerarchico con nomi a più livelli, come l'esempio "PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE", può rendere più chiaro il posizionamento del documento nell'ambito del progetto.
3. **Relazioni tra documenti:** Progettare lo schema in modo che documenti correlati abbiano nomi correlati. Questo aiuta a identificare facilmente documenti simili o collegati.
4. **Configurazione e gestione:** Utilizzare un database di configurazione collegato alle risorse del progetto per tenere traccia dei documenti e delle loro versioni. Questo semplifica la gestione dei documenti nel tempo e aiuta a garantire che le versioni corrette siano utilizzate.
5. **Documenti obbligatori:** Identificare i documenti che devono essere conservati per l'intera durata del software e assegnare loro nomi specifici o percorsi nella struttura dei nomi.
6. **Pianificazione e comunicazione:** Comunicare chiaramente lo schema di denominazione dei documenti a tutti i membri del team per garantire che sia seguito in modo uniforme.
7. **Flessibilità:** Assicurarsi che lo schema sia sufficientemente flessibile da gestire l'evoluzione del progetto nel tempo.

Uno schema di denominazione dei documenti ben progettato è un elemento fondamentale per una gestione documentale efficace in progetti complessi.

Gerarchia di configurazione

Configuration hierarchy

15



Gestione dei cambiamenti

La gestione dei cambiamenti è un aspetto critico nello sviluppo del software poiché i sistemi software sono soggetti a continui cambiamenti e richieste di modifica da parte degli utenti, degli sviluppatori e delle forze di mercato. Ecco alcune considerazioni importanti sulla gestione dei cambiamenti:

- 1. Richieste di cambiamento:** Le richieste di cambiamento possono provenire da diverse fonti, inclusi gli utenti finali che richiedono nuove funzionalità o correzioni di bug, gli sviluppatori che riconoscono la necessità di miglioramenti tecnici o le dinamiche di mercato che richiedono adattamenti.
- 2. Tracciamento dei cambiamenti:** Un aspetto fondamentale della gestione dei cambiamenti è il tracciamento di tutte le richieste di modifica. Ogni richiesta dovrebbe essere registrata, documentata e assegnata a un responsabile per la sua valutazione e implementazione.
- 3. Prioritizzazione dei cambiamenti:** Non tutte le richieste di cambiamento possono essere implementate immediatamente. È importante stabilire una priorità per le richieste in base alla loro importanza e urgenza. Ciò aiuta a garantire che i cambiamenti critici vengano affrontati prima.
- 4. Valutazione dell'impatto:** Prima di implementare un cambiamento, è necessario valutare l'impatto che avrà sul sistema esistente. Questa valutazione dovrebbe considerare sia gli aspetti tecnici che quelli finanziari.

5. Comunicazione: È essenziale comunicare chiaramente i cambiamenti agli stakeholder interessati, inclusi gli utenti finali, i membri del team di sviluppo e altri interessati. La comunicazione efficace aiuta a evitare fraintendimenti e resistenze al cambiamento.

6. Testing e verifica: Ogni cambiamento dovrebbe essere testato accuratamente per garantire che non introduca nuovi errori o problemi nel sistema. La verifica è un passo fondamentale per assicurarsi che il sistema continui a funzionare in modo affidabile dopo il cambiamento.

7. Documentazione: Ogni cambiamento dovrebbe essere documentato in modo completo, compreso il motivo del cambiamento, la sua implementazione e l'impatto sul sistema. La documentazione aiuta a mantenere un registro accurato delle modifiche apportate.

La gestione dei cambiamenti è un processo continuo e dinamico che richiede una pianificazione e un controllo attenti per assicurare che i cambiamenti vengano gestiti in modo efficace e che il sistema software rimanga affidabile e in linea con le esigenze degli utenti.

Revisione dei cambiamenti

La revisione dei cambiamenti da parte di un gruppo esterno è una pratica importante per garantire che le modifiche apportate a un sistema siano effettivamente vantaggiose dal punto di vista strategico e organizzativo. Alcuni punti chiave da tenere a mente riguardo a questo processo sono:

- 1. Revisione indipendente:** È essenziale che il gruppo incaricato della revisione dei cambiamenti sia indipendente dal team responsabile del progetto o del sistema. Questo garantisce un'analisi obiettiva e imparziale delle modifiche proposte.
- 2. Valutazione strategica:** La revisione dovrebbe concentrarsi sulla valutazione strategica delle modifiche. Questo significa valutare se le modifiche sono allineate agli obiettivi aziendali e se comportano vantaggi strategici a lungo termine. Non si tratta solo di valutare aspetti tecnici, ma di considerare l'impatto complessivo sul business.
- 3. Involgimento di parti interessate:** È importante coinvolgere rappresentanti sia dal cliente che dal contraente nella revisione. Ciò garantisce che le prospettive di entrambe le parti siano prese in considerazione e che le decisioni siano prese in modo collaborativo.
- 4. Change Control Board (CCB):** Il gruppo responsabile della revisione dei cambiamenti può essere chiamato Change Control Board (CCB). Questo gruppo ha il compito di valutare e autorizzare i cambiamenti in base a criteri strategici e organizzativi.

La revisione dei cambiamenti è una pratica importante per evitare modifiche non necessarie o controproducenti, garantendo che ogni modifica apporti valore e contribuisca agli obiettivi aziendali complessivi.

Registro delle modifiche

Un registro delle modifiche, noto anche come "change log" o "changelog," è un documento o una sezione di un documento che tiene traccia delle modifiche apportate a un componente, come il codice sorgente di un software o un documento di progetto. Ecco alcune informazioni tipiche che dovrebbero essere incluse in un registro delle modifiche:

1. **Descrizione della modifica:** Questo campo dovrebbe indicare brevemente cosa è stato cambiato. Ad esempio, "Risolto un bug di convalida del modulo di login."
2. **Rationale:** Qui dovrebbe essere spiegato il motivo per cui la modifica è stata apportata. Può includere spiegazioni su problemi o requisiti specifici che hanno portato alla modifica.
3. **Autore:** Chi ha apportato la modifica dovrebbe essere registrato. Questo può essere il nome dell'individuo o il nome del team responsabile.
4. **Data:** La data in cui la modifica è stata implementata.
5. **Numero di versione:** La versione del componente in cui è stata apportata la modifica. Questo può essere utile per tenere traccia delle modifiche nel tempo.
6. **Altre informazioni pertinenti:** A seconda delle esigenze, potrebbe essere necessario includere ulteriori dettagli, come riferimenti a problemi o ticket di tracciamento, o informazioni sulla revisione o l'approvazione della modifica.

L'inclusione di un registro delle modifiche è una pratica comune nel controllo delle versioni del software e nel documentare le modifiche apportate a documenti importanti. Aiuta a tenere traccia delle modifiche nel tempo, a capire perché sono state apportate e a chi è stata attribuita la responsabilità.

Informazioni sull'intestazione del componente

Component Header Information

```
// PROTEUS project (ESPRIT 6087)
//
// PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE
//
// Object: PCL-Tool-Desc
// Author: G. Dean
// Creation date: 10th November 1998
//
// © Lancaster University 1998
//
// Modification history
// Version          Modifier Date      Change      Reason
// 1.0      J. Jones   1/12/1998  Add header  Submitted to CM
// 1.1      G. Dean    9/4/1999  New field  Change req. R07/99
```

Versioni/varianti/rilasci

- **Versione (Version):** Una istanza di un sistema che è funzionalmente distinta in qualche modo dalle altre istanze del sistema. In altre parole, una versione è una variante del sistema che ha delle differenze funzionali rispetto ad altre versioni.
- **Variante (Variant):** Un'istanza di un sistema che è funzionalmente identica ma non funzionalmente distinta dalle altre istanze del sistema. In sostanza, una variante è una versione del sistema che ha le stesse funzionalità di altre varianti ma può variare in termini di configurazione, personalizzazione o altre caratteristiche non funzionali.
- **Rilascio (Release):** Un'istanza di un sistema che è distribuita agli utenti al di fuori del team di sviluppo. Un rilascio è una versione o una variante del sistema che è stata considerata pronta per l'uso da parte degli utenti finali ed è stata distribuita per l'utilizzo effettivo.

Identificazione della versione

Il processo di identificazione delle versioni è fondamentale per gestire e tenere traccia delle diverse versioni dei componenti software. Esistono tre tecniche di base per l'identificazione delle versioni:

1. **Numerazione delle versioni:** Questa tecnica coinvolge l'assegnazione di un numero univoco a ciascuna versione del componente. Solitamente, si utilizza una convenzione di numerazione che può includere numeri principali, numeri minori e, talvolta, numeri di revisione. Ad esempio, "1.0.1" potrebbe rappresentare la prima versione principale, la prima sottoversione e la prima revisione di un componente.
2. **Identificazione basata sugli attributi:** In questo approccio, le versioni dei componenti sono identificate in base a specifici attributi o proprietà. Ad esempio, potresti identificare una versione in base alla data di rilascio o a una firma digitale associata. Questo metodo può essere utile quando si desidera includere informazioni aggiuntive nella identificazione delle versioni.
3. **Identificazione basata sul cambiamento:** Questo metodo si concentra sugli specifici cambiamenti o aggiornamenti apportati a un componente. Ogni modifica al componente viene registrata in modo univoco e l'identificazione della versione può essere basata su tali cambiamenti. Questo è particolarmente utile quando si desidera tracciare i singoli cambiamenti nel sistema.

La scelta della tecnica dipenderà dalle esigenze specifiche del progetto e dalla complessità del sistema. In molti casi, viene utilizzata una combinazione di queste tecniche per fornire un'identificazione completa e univoca delle versioni dei componenti software.

Gestione dei rilasci

Il processo di gestione delle release è fondamentale per garantire che il software sia costantemente aggiornato e soddisfi le esigenze degli utenti. Ecco alcune considerazioni chiave sulla gestione delle release:

1. **Incorporazione di cambiamenti:** Le release devono includere i cambiamenti necessari dovuti agli errori scoperti dagli utenti o alle modifiche hardware. Questi cambiamenti sono spesso

noti come correzioni di bug o patch. È importante avere un meccanismo per tracciare e gestire questi cambiamenti in modo da assicurarsi che siano inclusi nelle release appropriate.

2. **Nuova funzionalità:** Oltre alle correzioni di bug, le release possono anche incorporare nuove funzionalità o miglioramenti al software. Questi possono essere basati su nuovi requisiti degli utenti o sulle esigenze di mercato. La pianificazione delle release dovrebbe includere la valutazione di quale nuova funzionalità è prioritaria e quando dovrebbe essere introdotta.
3. **Pianificazione delle release:** La pianificazione delle release è il processo di determinare quando rilasciare una nuova versione del software. Questa decisione può dipendere da vari fattori, tra cui la gravità dei bug da correggere, l'urgenza delle nuove funzionalità e la disponibilità delle risorse di sviluppo. È importante avere una strategia chiara per stabilire quando una release è pronta per essere distribuita.
4. **Versioning:** Un aspetto importante della gestione delle release è la versioning del software. Ogni release dovrebbe essere identificata da un numero di versione univoco in modo che gli utenti possano facilmente identificare quale versione stanno utilizzando e quale è la più recente.
5. **Comunicazione:** È essenziale comunicare chiaramente con gli utenti riguardo alle nuove release. Questo include la documentazione delle modifiche apportate, la risoluzione di problemi noti e l'offerta di supporto per l'aggiornamento. Una buona comunicazione può contribuire a evitare confusione e frustrazione tra gli utenti.

La gestione delle release è un processo continuo che richiede una pianificazione accurata e una gestione efficace delle risorse. È fondamentale per garantire che il software rimanga affidabile e competitivo nel mercato.

Rilasci di sistema

Una release del software è molto più di un semplice insieme di programmi eseguibili. Può comprendere vari elementi che contribuiscono a garantire il corretto funzionamento e la facilità d'uso del software. Ecco alcuni degli elementi che possono essere inclusi in una release del software:

1. **File di configurazione:** Questi definiscono come la release è configurata per un'installazione specifica. Possono includere impostazioni come le preferenze dell'utente, le connessioni di rete, i percorsi dei file e altre configurazioni rilevanti.
2. **File di dati:** Alcuni software richiedono dati specifici per funzionare correttamente. Questi dati possono essere inclusi nella release o possono essere scaricati o generati successivamente durante l'installazione o l'utilizzo del software.
3. **Programma di installazione:** Questo è un programma o uno script che semplifica il processo di installazione del software sul hardware di destinazione. Può automatizzare le configurazioni, copiare i file necessari e gestire altre attività di installazione.
4. **Documentazione:** La documentazione è essenziale per gli utenti del software. Può includere manuali utente, guide all'installazione, documentazione tecnica e altro materiale informativo per facilitare l'utilizzo del software.

5. **Imballaggio:** L'imballaggio del software è importante per la distribuzione e la presentazione del prodotto. Questo può includere la progettazione di copertine, etichette e materiali promozionali per il software.

6. **Supporto:** Le release del software spesso includono informazioni su come ottenere supporto tecnico o assistenza in caso di problemi o domande degli utenti. Questo può includere dettagli sui canali di supporto, come l'assistenza telefonica o l'assistenza via e-mail.

7. **Distribuzione:** Le release del software possono essere distribuite in vari formati, tra cui CD o DVD fisici o come file di installazione scaricabili dal web. La scelta del metodo di distribuzione dipende spesso dalla portata del software e dalle preferenze dell'utente.

Problemi di rilascio

La gestione delle release del software è una parte critica del processo di sviluppo e distribuzione del software. È importante tener conto delle esigenze e delle preferenze dei clienti quando si pianifica e si distribuisce una nuova release del sistema. Alcuni punti da considerare includono:

1. **Ascoltare i clienti:** È fondamentale raccogliere feedback dai clienti per comprendere le loro esigenze e preferenze. Prima di pianificare una nuova release, è importante comunicare con i clienti per capire se desiderano nuove funzionalità o se sono soddisfatti con la versione attuale.

2. **Versionamento:** Utilizzare una strategia di versionamento chiara per identificare chiaramente le diverse release del software. Ad esempio, è comune utilizzare numeri di versione come "1.0", "2.0", ecc. In questo modo, i clienti possono identificare facilmente quale versione del software stanno utilizzando.

3. **Opzioni di aggiornamento:** Fornire ai clienti opzioni di aggiornamento flessibili. Alcuni potrebbero preferire di rimanere con la versione attuale, mentre altri desiderano adottare la nuova release. Consentire agli utenti di scegliere quale versione installare o offrire un processo di aggiornamento agevole può essere vantaggioso.

4. **Compatibilità all'indietro:** Se possibile, mantenere la compatibilità all'indietro con le versioni precedenti del software. Ciò significa che le funzionalità e i dati esistenti dovrebbero continuare a funzionare correttamente con la nuova release, evitando interruzioni per gli utenti.

5. **Gestione dei file:** Come sottolineato, è importante che tutti i file necessari per una release vengano adeguatamente gestiti. Assicurarsi che tutti i file richiesti siano inclusi nella nuova release e che non vi siano conflitti o problemi di installazione.

6. **Documentazione:** Fornire documentazione chiara e aggiornata per guidare gli utenti nell'installazione e nell'utilizzo della nuova release. Questo aiuta a evitare confusione e problemi durante l'aggiornamento.

Strategia di rilascio del sistema

| Fattore | Descrizione |
|---------|-------------|
|---------|-------------|

| Fattore | Descrizione |
|---|--|
| Qualità tecnica del sistema | Se vengono segnalati gravi difetti del sistema che influiscono sul modo in cui molti clienti utilizzano il sistema, potrebbe essere necessario rilasciare una versione di correzione dei difetti. Tuttavia, i difetti minori del sistema possono essere riparati mediante il rilascio di patch (spesso distribuite su Internet) che possono essere applicate alla versione corrente del sistema. |
| Cambiamenti di piattaforma | Potrebbe essere necessario creare una nuova versione di un'applicazione software quando viene rilasciata una nuova versione della piattaforma del sistema operativo. |
| Quinta legge di Lehman (vedi Capitolo 21) | Questo suggerisce che l'incremento di funzionalità incluso in ogni versione è approssimativamente costante. Pertanto, se è stata rilasciata una versione del sistema con significative nuove funzionalità, potrebbe essere necessario rilasciare una versione di correzione. |
| Concorrenza | Potrebbe essere necessario un nuovo rilascio del sistema perché è disponibile un prodotto concorrente. |
| Requisiti di marketing | Il dipartimento marketing di un'organizzazione potrebbe aver fatto un impegno affinché le versioni siano disponibili in una data specifica. |
| Proposte di modifica del cliente | Per sistemi personalizzati, i clienti potrebbero aver presentato e pagato per un insieme specifico di proposte di modifica del sistema e si aspettano un rilascio del sistema non appena queste sono state implementate. |

Costruzione del sistema

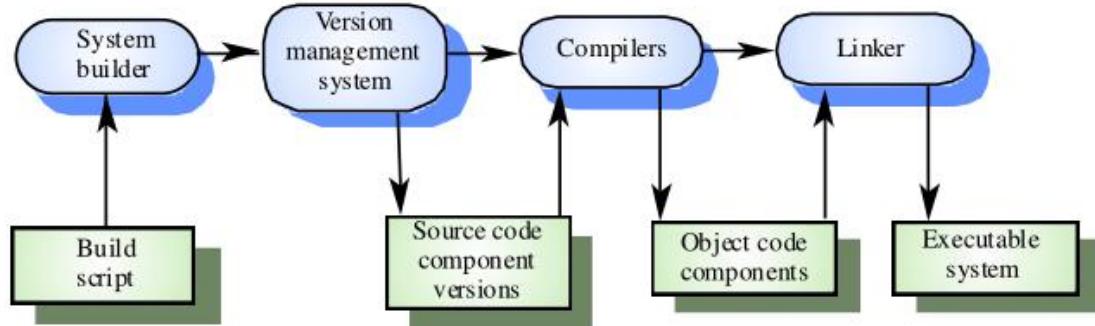
Il processo di "build" (compilazione) è una fase cruciale nello sviluppo del software, durante la quale i componenti del software vengono trasformati in un sistema eseguibile. Ecco alcune considerazioni chiave legate a questo processo:

- Compilazione:** Durante la fase di compilazione, il codice sorgente scritto in un linguaggio di programmazione viene tradotto in linguaggio macchina o in un linguaggio intermedio, a seconda del linguaggio e dell'ambiente di sviluppo utilizzati. Questo passaggio è essenziale per rendere il software eseguibile sulla piattaforma di destinazione.
- Linking:** Dopo la compilazione, i file risultanti (spesso chiamati file oggetto) devono essere collegati tra loro e con le librerie di sistema per creare un'eseguibile completo. Il processo di linking collega le diverse parti del software in modo che possano cooperare correttamente durante l'esecuzione.
- Automazione:** Oggi, la maggior parte dei progetti di sviluppo del software utilizza strumenti di automazione per semplificare il processo di build. Questi strumenti sono spesso configurati attraverso script di build che definiscono quali componenti devono essere inclusi, come devono essere compilati e come devono essere collegati.
- Dipendenze:** Il processo di build deve gestire le dipendenze tra i componenti. Ad esempio, se un componente dipende da una libreria esterna, il processo di build deve assicurarsi che

questa libreria sia disponibile e collegata correttamente.

5. **Ambiente di build:** L'ambiente di build è la configurazione specifica in cui avviene il processo di build. Può variare in base al sistema operativo, al compilatore e alle librerie utilizzate. Mantenere un ambiente di build coerente e ben documentato è essenziale per garantire la riproducibilità del processo di build.
6. **Testing della build:** Dopo la creazione di un'eseguibile, è importante condurre test di build per verificare che il software generato funzioni correttamente e sia privo di errori critici introdotti durante il processo di compilazione e linking.

System building



Strumenti CASE per la gestione della configurazione

Gli strumenti CASE (Computer-Aided Software Engineering) svolgono un ruolo cruciale nella gestione della configurazione (CM) dei progetti software. Ecco alcune funzionalità chiave che questi strumenti offrono per la gestione della configurazione:

1. **Controllo delle versioni:** Gli strumenti CASE consentono di tenere traccia delle diverse versioni di ogni componente software. Questo è fondamentale per garantire che sia possibile tornare indietro a una versione precedente in caso di problemi e per gestire le modifiche in modo controllato.
2. **Gestione delle modifiche:** Consentono di documentare e gestire le modifiche apportate al software. Questo include la registrazione delle richieste di modifica, l'approvazione, l'attuazione e la verifica delle modifiche.

3. **Integrazione con il sistema di compilazione:** Spesso, gli strumenti CASE sono integrati con sistemi di compilazione e automazione del processo di build. Ciò assicura che le versioni corrette dei componenti vengano utilizzate durante la compilazione.
4. **Tracciamento delle dipendenze:** Gli strumenti permettono di identificare e gestire le dipendenze tra i componenti software. Questo è importante per assicurarsi che le modifiche a un componente non causino problemi in altri componenti dipendenti.
5. **Gestione dei rilasci:** Consentono di pianificare e registrare i rilasci delle diverse versioni del software. Questo aiuta a garantire che le versioni corrette vengano distribuite agli utenti finali.
6. **Audit e reporting:** Forniscono strumenti per condurre audit sulla configurazione e generare report sulla gestione della configurazione. Questi report sono utili per verificare la conformità alle procedure e per l'analisi delle tendenze.
7. **Collaborazione:** Spesso, gli strumenti CASE offrono funzionalità di collaborazione, consentendo a team distribuiti di lavorare insieme in modo sincronizzato sulla gestione della configurazione.
8. **Sicurezza:** Assicurano che solo utenti autorizzati possano apportare modifiche o accedere a determinate parti del software o della configurazione.
9. **Backup e ripristino:** Forniscono strumenti per il backup e il ripristino dei dati di configurazione in caso di perdita o danneggiamento.

Strumenti CASE

Gli strumenti di gestione del cambiamento (Change Management Tools) e gli strumenti di gestione delle versioni (Version Management Tools) sono essenziali per una gestione efficace del ciclo di vita del software. Ecco una panoramica di come questi strumenti possono essere utili:

Strumenti di gestione del cambiamento:

1. **Form editor:** Questo strumento consente di creare e modificare documenti e moduli specifici per la gestione dei cambiamenti. Questi documenti possono essere utilizzati per registrare dettagli sulle richieste di cambiamento, inclusi dettagli come la descrizione del cambiamento, la priorità, il responsabile e la data prevista per l'implementazione.
2. **Workflow system:** Un sistema di flusso di lavoro aiuta a definire e monitorare i passaggi coinvolti nella gestione dei cambiamenti. Determina chi è responsabile di cosa, quali approvazioni sono necessarie e il flusso generale dei cambiamenti attraverso il processo.
3. **Change database:** Questo strumento funge da repository centralizzato per registrare tutte le richieste e i dettagli dei cambiamenti. Consente una gestione organizzata e il tracciamento delle richieste.
4. **Change reporting system:** Questo sistema genera report e analisi sulla gestione dei cambiamenti. Può fornire informazioni sullo stato dei cambiamenti, le tendenze e le metriche chiave per valutare l'efficacia del processo di gestione dei cambiamenti.

Strumenti di gestione delle versioni:

1. **Version and release identification:** Questi strumenti consentono di assegnare un numero di versione univoco a ciascuna release del software. Questo aiuta a tracciare quale versione è attualmente in uso e quale è la più recente.
2. **Storage management:** Gestisce la memorizzazione delle diverse versioni del software e le differenze tra di esse. Questo è importante per consentire il ripristino di versioni precedenti se necessario e per mantenere un registro delle modifiche apportate a ciascuna versione.
3. **Independent development:** Questi strumenti consentono a più sviluppatori di lavorare contemporaneamente su diverse parti del software senza interferenze. Ciò è particolarmente utile in progetti di sviluppo collaborativo.
4. **Project support:** Fornisce supporto per la gestione di progetti software complessi. Questi strumenti possono aiutare a coordinare le attività tra i membri del team e garantire che il progetto proceda in modo efficace.
5. **Dependency specification language and interpreter:** Questi strumenti aiutano a gestire le dipendenze tra le diverse componenti del software. Consentono di definire le relazioni tra le parti del software e di garantire che le modifiche a una componente non causino problemi in altre parti del sistema.
6. **Distributed compilation:** Questo strumento consente la compilazione distribuita del codice sorgente, il che può accelerare notevolmente il processo di sviluppo.
7. **Derived object management:** Gestisce gli oggetti derivati dal codice sorgente, come i file eseguibili e le librerie, e ne traccia le versioni e le dipendenze.