UNIONE EUROPEA

FONDO EUROPEO DI
SVILUPPO REGIONALE

Ministero Sviluppo Economico

REGIONE PUGLIA
DIPARTIMENTO SVILUPPO ECON
INNOVAZIONE, ISTRUZION
FORMAZIONE E LAVORO

TECHNICAL REPORT 2.2

METODI E TECNICHE PER LA MANUTENZIONE FULL REUSE

INDICE

1. STORIA DEL DOCUMENTO 4

2. INTRODUZIONE 7

3. INTERAZIONI 11

3.1 INTEGRAZIONE DI APPLICAZIONI 15
3.1.1 Integrazione applicazioni software 17
3.2 TECNOLOGIE CHE REALIZZANO L'INTEGRAZIONE APPLICAZIONI 21
3.2.1 Le API 22
3.2.2 SOA 23
3.2.3 Web Service: architettura service-oriented 29
3.2.4 Web Service: ReSTful 36
3.2.5 I principi REST 38
3.2.6 REST ed il protocollo HTTP 38
3.3 RESOURCE ORIENTED ARCHITECTURE 43
3.3.1 OpenAPI Specification (Swagger) 47

4. COLLABORAZIONE CON LE PARTI INTERESSATE 52

5. CICLO DI VITA DEL SOFTWARE 57

5.1 MODELLI DEL CICLO DI VITA DEL SOFTWARE IN BASE ALL'APPROCCIO DEL TESTING 57
5.1.1 Modelli sequenziali 57
5.1.2 Modello incrementale 59
5.1.3 Modello iterativo 60
5.1.4 Modello a spirale (o per prototipi) 61
5.1.5 Modello RUP (Rational Unified Process) 61
5.1.6 Modelli con approccio Agile 63
5.1.7 Differenze tra TDD, ATDD e BDD 75

6. RIUSO NELLA MANUTENZIONE 78

6.1 MANUTENZIONE 78
6.2 UN QUADRO DI RIUTILIZZO 81
6.3 CONFRONTARE I MODELLI USANDO IL QUADRO DI RIUTILIZZO 84
6.4 IL PARADIGMA DELLA METRICA DELLA DOMANDA DELL'OBIETTIVO 87

6.4.1 Applicazione del paradigma della metrica della domanda dell'obiettivo 87
6.5 ABILITATORI DEL RIUSO 91

7. SVILUPPO ORIENTATE AL RIUSO E RALLENTAMENTO DEL DEGRADO NELLA MANUTENZIONE 95

7.1 IL RIUTILIZZO IN LETTERATURA 96
7.2 MODELLO DI MANUTENZIONE DEL FULL REUSE 98
7.3 PROGETTO CONV E PROGETTO FR 102
7.3.1 Analisi dei dati 105
7.4 MODELLO DI SVILUPPO ORIENTATO AL RIUTILIZZO (ROD) 115
7.5 RAPID SOFTWARE DEVELOPMENT PROCESS (RADO) 116
7.6 PROGETTO CONV E PROGETTO ROD 117
7.6.1 Disegno del caso di studio 118
7.6.2 Esecuzione 119
7.6.3 Analisi dei dati 120

8. BIBLIOGRAFIA 132

1. STORIA DEL DOCUMENTO

DATA	AUTORE	CONTENUTI & MODIFICHE
Giugno 2019	Responsabile di progetto	Canovaccio del TR e piano delle attività
Da Giugno 2019 a Settembre 2019	Alluvatti Giovanni Maria Altamura Mario Genua Giovanni Minniti Federica Villani Gianluca Lingua Gianluca	Redazione capitolo 3
Da Giugno 2019 a Settembre 2019	Danaila Mihai Cosmin Izzio Giovanni Piccolella Antonio Speciale Francesco Ludergnani Marco Natale Marco	Redazione capitolo 4
Ottobre 2019	Responsabile di Progetto + autori dei capitoli 3 e 4	Fusione dei capitoli 3 e 4
Ottobre 2019	Responsabile di Progetto + autori dei capitoli 3 e 4	Cap. 3e4- Verifica e validazione-evidenza di eventuali rilievi. Revisione del canovacci.
Ottobre 2019	Responsabile di Progetto + autori del capitolo 3	Superamento dei rilievi emersi dalla V&V e Redazione finale del cap. 3
Ottobre 2019	Responsabile di Progetto + autori del capitolo 4	Superamento dei rilievi emersi dalla V&V e Redazione finale del cap. 4
Da Ottobre a Novembre 2019	Alluvatti Giovanni Maria Lingua Gianluca Genito Piera Orioli Francesco	Redazione capitolo 5

Da Ottobre a Novembre 2019	Danaila Mihai Cosmin Speciale Francesco Ludergnani Marco Orso Emanuele	Redazione capitolo 6
Dicembre 2019	Responsabile di Progetto + autori dei capitoli 5 e 6	Fusione dei capitoli 5 e 6
Dicembre 2019	Responsabile di Progetto + autori del capitolo 5	Cap. 5- Verifica e validazione- evidenza di eventuali rilievi
Dicembre 2019	Responsabile di Progetto + autori del capitolo 6	Cap. 6- Verifica e validazione- evidenza di eventuali rilievi
Dicembre 2019	Responsabile di Progetto + autori del capitolo 5	Superamento dei rilievi emersi dalla V&V e Redazione finale del cap. 5
Dicembre 2019	Responsabile di Progetto + autori del capitolo 6	Superamento dei rilievi emersi dalla V&V e Redazione finale del cap. 6
Da Dicembre 2019 a Gennaio 2020	Alluvatti Giovanni Maria Bernardini Walter Genito Piera Lingua Gianluca Orioli Francesco Orso Emanuele Speciale Francesco Pagano Simone Villani Gianluca	Redazione capitolo 7
Gennaio 2020	Responsabile di Progetto + autori del capitolo 7	Cap. 7- Verifica e validazione- evidenza di eventuali rilievi
Gennaio 2020	Responsabile di Progetto + autori del capitolo 7	Superamento dei rilievi emersi dalla V&V e Redazione finale del cap. 7
Febbraio 2020	Ser&P+ Responsabile progetto	Revisione del documento sulla base delle evidenze emerse durante l'avanzamento del progetto – Fusione, redazione

		e pubblicazione del Rapporto finale.

2. INTRODUZIONE

Partendo dal presupposto che in iMOLE si ha l'obiettivo di massimizzare il riutilizzo dell'esperienza sotto forma di conoscenza, processi, prodotti e strumenti, il processo di manutenzione è idealmente orientato al riutilizzo del software e, quindi, della conoscenza accumulata in esso, con particolare riferimento alla documentazione. In generale, il processo di manutenzione trasforma un prodotto in esercizio in un nuovo prodotto che è l'evoluzione del precedente in uno o più dei seguenti aspetti: i servizi messi a disposizione degli utilizzatori che si adeguano a nuovi contesti legislativi e/o operativi; la struttura architetturale, le tecnologie utilizzate per la sua implementazione. Il prodotto originario viene "trasportato" nel nuovo contesto arricchendolo, se necessario, di ulteriori funzionalità e caratteristiche tecniche che devono mantenere e, se possibile, aumentare il suo "valore" per tutte le parti interessate.

Per attuare un vero riuso all'interno del processo di manutenzione, nel tempo sono stati effettuati diversi tentativi basati sulla definizione di processi, strumenti e regole utilizzabili solo in particolari contesti che vedevano ciclo di sviluppo del software, conteso applicativo e tecnologie definite e non modificabili rendendo il tutto non riutilizzabile in contesti differenti [Cybulski][Agresti and McGarry][Krueger 1989]. Invece, in un lavoro del 1990 Victor Basili [Basili, 1989], ha presentato un paradigma organizzativo di alto livello, *Full Reuse-Model*, per la manutenzione in cui un'organizzazione può apprendere dalle attività di sviluppo e manutenzione e quindi applicare tale paradigma a diversi modelli di processo di manutenzione.

Secondo il Full Reuse-Model la prima attività da eseguire nelle fasi di analisi, progettazione e codifica di un sistema è quella di riutilizzare requisiti, la progettazione e il codice appropriati da qualsiasi versione precedente del vecchio sistema. L'applicazione di tale modello presuppone quindi un repository di documenti e componenti che definiscono le versioni precedenti del sistema corrente e di sistemi simili. In questo modello il riutilizzo è esplicito e le decisioni che esso richiede che si facciano durante il processo di manutenzione assicurano che la qualità del sistema software decada molto lentamente nel tempo. Ciò implica che la vita di un sistema software si allunghi e che, quindi, l'investimento per la sua produzione rientri ampiamente con il suo esercizio.

Il problema della manutenzione di un sistema software e della corrispondente qualità si pone in modo ancora più rilevante quando il

software da produrre e sviluppare è una infrastruttura piuttosto che di una applicazione.

Lo scopo dei modelli per la gestione del ciclo di vita di una applicazione è quello di consentire la produzione minimizzando tempi e costi assicurando che la qualità del prodotto sia adeguata all'uso che si deve fare dell'applicazione. Per la gestione del ciclo di vita di un'applicazione software l'efficacia è nella riduzione del tempo di rilascio di un prodotto, dei suoi aggiornamenti e patch di sicurezza.

Il modello di gestione del ciclo di vita di un'infrastruttura richiede un approccio diverso. Lo scopo della gestione di un'infrastruttura include oltre alla efficacia di sviluppo delle singole applicazioni di cui si compone anche il bisogno di supportare l'operatività di diversi gruppi di lavoro altamente specializzati e mutevoli nel tempo che devono operare sulle diverse componenti e su diversi aspetti della stessa infrastruttura. In altre parole, l'infrastruttura rappresenta la base permanente utilizzata, per tempi lunghi, per attuare le attività di sviluppo, di test, di rilascio e di manutenzione delle sue componenti e per soddisfare le diversificate esigenze delle molte classi di utilizzatori che seguono percorsi diversi di cambiamenti.

Data la natura della infrastruttura, i progetti di realizzazione di una infrastruttura e di evoluzione della stessa sono tipicamente molto più estesi rispetto ai progetti di un'applicazione, il che significa che cicli di rilasci evolutivi se non curano il riuso adeguato delle componenti della struttura corrente della stessa infrastruttura possono impedire il raggiungimento completo degli obiettivi della stessa. A titolo di esempio, l'introduzione di una nuova tecnologia per l'erogazione di un servizio web all'interno di una infrastruttura consolidata richiede l'esecuzione di una serie di attività (scouting delle tecnologie, identificazione della tecnologia, implementazione della tecnologia, integrazione della tecnologia con l'infrastruttura esistente, rilascio della nuova infrastruttura, formazione all'uso) la cui esecuzione impatta su molte fasi dello sviluppo del software, su diversi aspetti dello stesso software che sono tutti da documentare, su diversi gruppi di sviluppo e di utilizzatori.

Queste esigenze dei modelli di gestione delle infrastrutture hanno richiesto nuove tecnologie per la produzione del software e per la sua manutenzione che concilino le esigenze descritte per realizzare la produzione e la evoluzione efficaci di un'infrastruttura. La produzione di un'infrastruttura e la sua evoluzione devono basarsi sui seguenti principi:

1. **Individui e riuso**. Per progettare e realizzare una infrastruttura è necessario avere come riferimento le interazioni che avvengono tra i gruppi di lavoro, interazioni che comprendono la comunicazione diretta, governata dalle API, la messaggistica, le interdipendenze a livello di sistema, e i processi di test and release allo scopo di efficientare le attività di sviluppo e manutenzione di un sistema. Per questo motivo particolare attenzione deve essere rivolta agli strumenti e alle tecnologie di gestione degli strumenti utili a standardizzare le interfacce di comunicazione tra persone, macchine e persone-macchine.

2. **Collaborazione col cliente**, **specifiche di integrazione**. Le specifiche di integrazione rappresentano il modo in cui i team responsabili dell'infrastruttura gestiscono le dipendenze tra le componenti e gli aspetti dell'infrastruttura, come ad esempio le policy di sicurezza, i service-level agreement e persino le API pubblicate. Il coinvolgimento dei clienti dà loro la possibilità di conoscere e partecipare alla definizione al cambiamento delle policy e delle interfacce associate ai sistemi. Un forte anello di congiunzione con il cliente è da ritrovarsi nei test automatici. Da sempre i contratti con il cliente hanno trovato applicazione nella definizione dei requisiti dell'applicazione e nei test di accettazione dei sistemi. I requisiti altro non sono che i desiderata del cliente da utilizzarsi per la realizzazione del prodotto software, mentre i test di sistema e accettazione sono il mezzo utilizzato per certificare che il prodotto realizzato risponde alle esigenze manifestate nei requisiti. In questo caso è necessario disporre di un modello del ciclo di vita che preveda in modo esplicito il ruolo partecipativo per il cliente / committente.

3. **Software funzionale**. Essendo l'infrastruttura la base necessaria per l'esecuzione di tutte le attività, essa deve essere utilizzabile e funzionante 24/7/365, ed inoltre deve essere progettata in modo tale che possa adattarsi gradualmente alle nuove esigenze piuttosto che rispondere a cambiamenti drastici. Dal punto di vista della strategia infrastrutturale, "funzionale" significa che le differenti componenti dell'infrastruttura devono offrire il comportamento atteso dall'utente finale in uno scenario prestazionale prevedibile. Per raggiungere l'obbiettivo della funzionalità le componenti dell'infrastruttura, compreso l'ambiente di sviluppo, devono avere tutti la stessa configurazione e devono essere utilizzabili da tutti allo stesso modo, in questo vengono in aiuto le tecnologie quali cloud e container, ma è sempre più stringente l'esigenza di un processo di manutenzione che garantisca il riuso del patrimonio software disponibile dell'infrastruttura.

Il presente rapporto tecnico continua con:

- l'analisi delle interazioni e le conseguenti esigenze di integrazione delle applicazioni;
- le tecnologie che realizzano l'integrazione delle applicazioni;
- la collaborazione tra le parti interessate e come questa impatti sulla qualità del software;
- i modelli di ciclo di vita del software e come si sono evoluti al passaggio dalla produzione e sviluppo di un'applicazione ai corrispondenti processi per un'infrastruttura;
- l'approfondimento del riuso nella manutenzione e dei suoi abilitatori;
- i processi di sviluppo orientati al riuso ed al rallentamento del degrado nella manutenzione;
- la bibliografia.

3. INTERAZIONI

Per quanto riguarda le attività collegate alla manutenzione di una Applicazione di Impresa (nel prosieguo AI) l'iterazione tra i gruppi di lavoro ruota, come detto, attorno al fulcro della produzione di nuove funzioni ottenute con differenti strategie: Scrittura integrale, Composizione, Orchestrazione, Mista.

La categoria "**Scrittura integrale**" implica l'esecuzione di tutto il processo di produzione a partire dalla definizione delle capacità di cui deve essere dotata l'AI, sino alla scrittura del codice necessario per creare il sistema software che automatizza tali caratteristiche. Questo modello di sviluppo è applicato raramente nel caso di AI e più in generale nella realizzazione di grandi sistemi, in quanto i tempi di realizzazione, normalmente, differiscono dal tempo utile per l'immissione nel mercato della stessa AI (Time to market).

Parlare di AI sviluppate utilizzando il modello denominato "**Composizione**" significa parlare di "sviluppo basato su componenti" (Component-Based Development, CBD nel seguito). Il CBD è un approccio alla costruzione di grandi sistemi software e di AI basato sull'integrazione di componenti software pre-esistenti (acquisiti da terze parti o sviluppati in precedenza). CBD implica l'industrializzazione del processo di produzione del software basato sull'assemblaggio di componenti prefabbricate e già verificate, riducendo costi e tempi per l'immissione sul mercato del prodotto finito.

Le problematiche aperte in questo settore sono molteplici: la mancanza di modelli per la valutazione del costo per la realizzazione di una componente; la definizione dei processi per la selezione e la valutazione delle componenti che consentono di supportare la scelta in fase di progetto. In questa fase è importante valutare la qualità delle componenti da scegliere in quanto tale qualità influirà su quella del sistema che le utilizzerà o che le integrerà. Da ciò derivano altri problemi: stabilire se lo sviluppo per componenti è vantaggioso rispetto ad agli approcci classici; stabilire se comporta dei costi di realizzazione maggiori o minori.

La categoria "**Orchestrazione**" racchiude tutti quei sistemi il cui sviluppo è basato sulla logica dei processi. In altre parole, la realizzazione dell'applicazione parte dalla formalizzazione di un processo di business che attraverso un insieme di traduzioni successive trasforma tale descrizione in un linguaggio comprensibile da un esecutore automatizzato. Inoltre, durante le trasformazioni avviene anche un processo di arricchimento in cui a ogni

attività sono associate componenti software che le eseguono. I passi principali per realizzare un AI con questo modello di sviluppo sono:

- Individuazione dei requisiti di business specifici del committente attraverso la formalizzazione del processo di business che esso intende automatizzare;
- Individuazione e selezione delle componenti utili a realizzare le fasi del processo da automatizzare;
- Specializzazione o costruzione ex novo delle componenti le cui capacità non siano coperte da alcuna componente disponibile;
- Assemblaggio delle componenti individuate e specializzazione delle stesse rispetto all'applicazione d'impresa che automatizza il processo di business del committente.

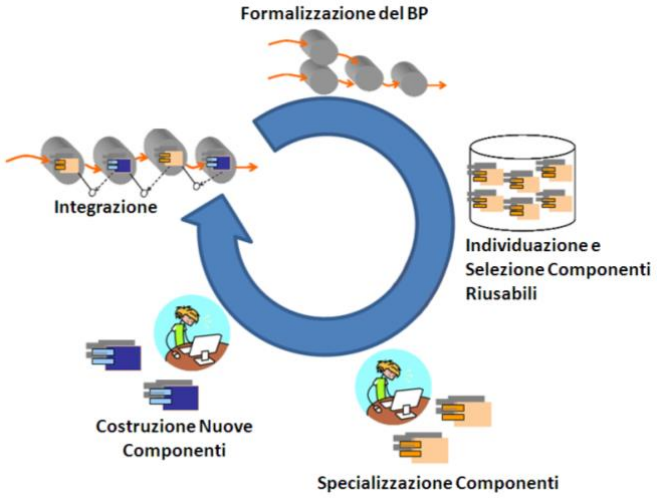

Figura 3.1 Fasi del modello per composizione

La categoria "**Mista**" racchiude le AI che sono state realizzate con almeno due differenti modelli di sviluppo. In altri termini, in questa tipologia ricadono tutte le applicazioni in cui non è stessa utilizzata una sola metodologia, bensì alcune parti sono state scritte ex novo, altre sono state sviluppate per composizione di sistemi e così via. Anche questo modello di sviluppo è poco utilizzato in quanto rende di difficile attuazione la pianificazione e la gestione del ciclo di sviluppo del sistema. Inoltre, questo modello rende di difficile attuazione la manutenzione del sistema realizzato.

I modelli di sviluppo "Composizione", "Orchestrazione" e "Mista", hanno come fattore comune il riuso del software, tale pratica ha come principale

obiettivo la riduzione dei costi di produzione, dei tempi di sviluppo e l'incremento della qualità. Ovviamente ci sono nuove problematiche che si accompagnano alla pratica del riuso, che devono essere considerate:

- L'incremento nei costi di gestione del sistema;
- La mancanza di strumenti di supporto al riuso;
- La necessità di gestire una o più librerie di componenti;
- La necessità di disporre di specialisti per le operazioni di integrazione e adattamento di componenti.

La granularità delle componenti da riutilizzare, in contesti reali, è differente. Difatti, dipendentemente dalle esigenze, è possibile riusare intere applicazioni, sottosistemi, componenti, oggetti, librerie e servizi web. Infine, i possibili approcci adottabili in caso di riuso, riportato nella figura seguente, sono molteplici e costituiscono una caratteristica da considerare in tutti i modelli descritti ad eccezione del modello "**Scrittura integrale**".

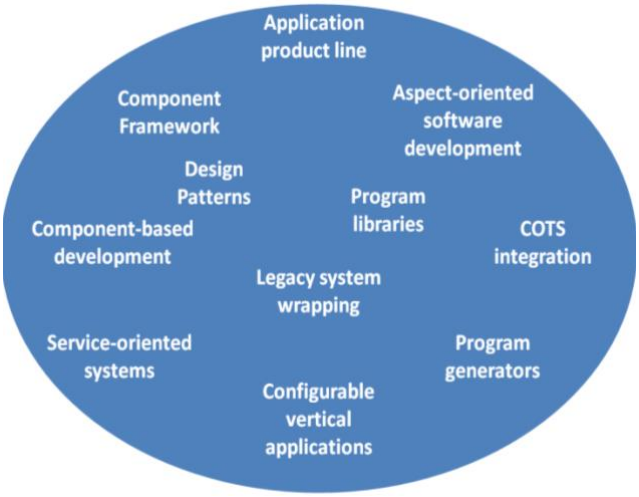

Figura 3.2 Approcci per il riuso

In letteratura sono presenti molteplici definizioni di componenti software. Per la prospettiva del progetto i-mole, ovvero quella del riuso delle stesse abbiamo utilizzato come riferimento principale le definizioni formulate da Szyperski e Messerchmitt nel lavoro 0 e da Brown e Wallnau nel lavoro 0. Szyperski e Messerchmitt hanno definito cinque criteri che una componente software deve avere per essere ritenuta tale. In linea con la prospettiva del presente lavoro a questi ne è stata aggiunta una, ovvero la resilienza. Difatti secondo questa visione i criteri da rispettare sono:

- **Multi-uso**: la capacità di poter essere utilizzata in diversi modi;
- **Indipendenza**: non deve dipendere dal particolare dominio applicativo nel quale è stata realizzata o utilizzata;
- **Componibilità**: deve esser possibile integrare la componente con altre componenti o con altri sistemi software;
- **Incapsulamento**: attraverso l'interfaccia della stessa, devono essere descritte solo le possibili relazioni con il mondo esterno, mentre tutto quello che è interno alla componente deve rimanere nascosto;
- **Autonomicità**: si riferisce alla capacità che un sistema software ha di gestire sé stesso, senza bisogno di "intelligenza", cioè di un controllo cosciente, come evidenziato nel lavoro 0. Una componente autonomica è caratterizzata da diversi elementi chiave, di seguito sono riportati solo gli elementi che una componente dovrebbe avere.
    - **Conoscenza di sé stesso**. Deve avere una conoscenza dettagliata delle sue componenti, dei suoi stati, delle sue capacità, delle sue connessioni, delle proprie risorse e di quelle che potrebbero essere utilizzate.
    - **Capacità di autoconfigurazione.** Ovvero, la capacità di riallocare le risorse a disposizione se necessario.
    - **Capacità di curare sé stesso**. La capacità di scoprire i problemi o i potenziali problemi, identificarne le cause e trovarne i rimedi.
    - **Sfida ai sistemi ermetici**. Per definizione una componente autonomica non può essere una soluzione proprietaria.
- **Resilienza**: è la capacità di un sistema di adattarsi alle condizioni d'uso e di resistere all'usura in modo da garantire la disponibilità dei servizi erogati. I contesti di riferimento sono quelli relativi alla business continuity (la capacità dell'azienda di continuare ad esercitare il proprio business a fronte di eventi catastrofici che possono colpirla) e al disaster recovery (l'insieme di misure tecnologiche e processi organizzativi atti a ripristinare sistemi, dati e infrastrutture necessarie all'erogazione di servizi di business a fronte di gravi emergenze). È definibile anche come una somma di abilità, capacità di adattamento attivo e flessibilità necessaria per adottare nuovi comportamenti una volta che si è appurato che i precedenti non funzionano. Alcune delle tecniche utilizzate per dotare una applicazione della resilienza sono le tecniche di ridondanza. Ad esempio, supponendo di disporre di un floppy disk HD ormai alla fine della su vita utile e quindi con un elevato rischio di guasto, allora la resilienza può essere ottenuta duplicando la risorsa a disposizione per la scrittura dei dati. Ad esempio, si

possono scrivere i dati su più dischi in modo da garantirsi contro la rottura di un singolo disco.

Brown e Wallnau descrivono una componente nel seguente modo: "*significativa, approssimativa, indipendente e sostituibile parte di un sistema che soddisfa una chiara funzione, nel contesto di una ben definita architettura*".

Nel mercato dell'ICT sono presenti numerose tipologie di componenti che, in seguito alle esperienze maturate durante le attività di ricerca, sono state catalogate nel seguente modo:

- **COTS** (**C**ommercial **O**ff-**T**he-**S**helf): prodotto software realizzato da terze parti, disponibile nel mercato per soddisfare un particolare requisito;
- **Open Source**: prodotto software per il quale il codice sorgente è disponibile e modificabile da tutti;
- **Free For not Commercial Use**: sono i software liberi per i quali l'utilizzo a fini di lucro è vietato;
- **In-house**: componenti sviluppate all'interno della organizzazione che la utilizza;
- **White box**: sono tutte le componenti software per le quali è disponibile il codice sorgente ma non è modificabile.

3.1 Integrazione di applicazioni

Quando si parla di "integrazione di applicazioni", si fa riferimento ad un concetto di ampie dimensioni: integrare applicazioni di tipo stand-alone non è la stessa cosa che integrare applicazioni client-server.

Per l'integrazione di una componente in una AI è necessario dapprima individuare le componenti da assemblare per la realizzazione della AI, in seguito queste sono amalgamate in modo tale da interagire tra loro, è ovvio che questo è possibile se le componenti scelte sono predisposte per poter essere integrate.

Le problematiche che possono scaturire sono differenti, proprio perché per "integrare" è necessario prendere in considerazione diversi aspetti.

Gli aspetti principali da considerare sono quattro: portabilità, manutenibilità, l'interoperabilità e resilienza.

- **Portabilità**: la portabilità di un componente software è un adattamento o una modifica della componente stessa, volto a consentirne l'uso in

un ambiente di esecuzione diverso da quello originale. L'adeguamento della componente può essere un'attività complessa e costosa che dipende dalla distanza tecnologica fra l'ambiente di origine e l'ambiente di destinazione, dal tipo di componente software da "portare", e dagli strumenti con cui esso è stato costruito. Un componente software è portabile quando l'operazione di adeguamento è semplice (e quindi poco costosa). In questo senso, la portabilità si può considerare un caso particolare di riusabilità del software.

- **Manutenibilità**: in seguito del rilascio di un prodotto software possono essere richieste le operazioni per la correzione o per l'evoluzione del prodotto rilasciato queste operazioni prendono il nome di manutenzione del software. La manutenibilità ha due significati, il primo associato al concetto statistico, il secondo come requisito. Sul piano *statistico*, la manutenibilità è la probabilità di un sistema, in assegnate condizioni di funzionamento, di essere riportato in uno stato nel quale possa svolgere la funzione richiesta. Ciò avviene quando la manutenzione è realizzata nelle condizioni determinate, e con procedure e mezzi prescritti. La manutenibilità esprime invece un importante *requisito* di progetto di un sistema, che definisce la sua capacità di essere facilmente **ripristinato** qualora sia necessario realizzare un intervento di manutenzione. In riferimento allo sviluppo di applicazioni d'impresa basato sul riuso di componenti software preesistenti, è bene sottolineare che l'evoluzione della AI non impatta necessariamente su tutte le componenti di cui è costituita. Al contrario l'evoluzione di una delle componenti impatta sulla AI. In definitiva la manutenzione del software è un processo fondamentale perché garantisce il funzionamento degli applicativi rilasciati dai quali dipende la sopravvivenza della AI prima, e della stessa organizzazione che le utilizza dopo.
- **Resilienza**: in questo contesto viene considerata al fine dell'integrazione parte del concetto della resilienza, precedentemente descritto, ovvero la capacità di un sistema di adattarsi alle condizioni d'uso e di ripristinare lo stato iniziale. Riprendendo le tecniche di ridondanza, la resilienza a cui siamo interessati in questo contesto è la capacità di un sistema di ritornare allo stato iniziale quando si verificano errori bloccanti e quindi continuare a lavorare. Dal punto di vista delle metodologie e delle tecniche infrastrutturali presenti in letteratura e utilizzate per far sì che un'applicazione software risulti essere resiliente sono i processi di Continuous Integration e

Continuous Delivery saranno gli elementi principali su cui si fonda l'infrastruttura i-mole. Una trattazione di dettaglio di questi due processi sarà seguita nel prossimo capitolo che tratta la parte del "software funzionale".

- **Interoperabilità**: l'interoperabilità è la capacità di un sistema o di un prodotto informatico di cooperare e di scambiare informazioni o servizi con altri sistemi in maniera più o meno completa e priva di errori, con i livelli di affidabilità desiderati e con ottimizzazione delle risorse. Obiettivo dell'interoperabilità è facilitare l'interazione fra sistemi differenti, lo scambio e il riutilizzo delle informazioni anche fra sistemi informativi non omogenei. Il termine interoperabilità è utilizzato in ambito tecnologico per indicare un elevato grado di sinergia di sistemi diversi con lo scopo di offrire servizi o funzionalità nuove. Il termine è quindi direttamente legato alla consolidata tendenza di far convergere su alcune tecnologie evolute una vasta gamma di servizi. Quanto ai servizi che si possono reperire sul web, l'interoperabilità ha due accezioni:
  - **interoperabilità *tecnica***, è la capacità di sistemi o componenti diversi, di scambiarsi informazioni, utilizzare informazioni scambiate e di lavorare insieme per svolgere un compito comune;
  - **interoperabilità *concettuale,*** è la relazione a livello semantico tra sistemi differenti, per esempio, banche dati diverse.

3.1.1 Integrazione applicazioni software

Assicurare la possibilità di far comunicare tra loro applicazioni eterogenee (cioè sviluppate su framework, piattaforme e/o sistemi operativi, linguaggi e tecnologie diverse) sta diventando ormai un elemento chiave per l'evoluzione dei sistemi informatici. Le esigenze possono essere limitate al semplice scambio di dati o evolversi fino alla possibilità di utilizzare metodi offerti da sistemi remoti per integrare servizi di terzi nella propria applicazione.

Le soluzioni di integrazione riguardano sia l'implementazione di applicazioni o di sistemi informatici creati ex novo, sia la gestione delle tecnologie aziendali preesistenti, in modo da non intaccare gli investimenti anteriori.

Quando si parla di integrazione, è bene quindi definire di che tipo di integrazione stiamo parlando, dal momento che esistono diversi "livelli di profondità", ovvero:

1. **User interface integration**: l'integrazione a livello di User Interface non è di fatto un'integrazione nel termine stretto del termine. Si opera affinché le due applicazioni (indipendenti tra di loro) presentino la medesima interfaccia utente condividendo quindi il medesimo "look". Questo tipo di integrazione è consigliabile laddove non è necessario distinguere da quale applicazione (integrante o integrata) provengano i dati. Questo approccio ha il vantaggio di non essere oneroso sia in termini tecnici che di tempo permettendo l'interoperabilità tra applicazioni eterogenee.
2. **Object/RPC integration**: l'integrazione Object/RPC prevede di affrontare l'integrazione a livello di oggetti distribuiti (EJB, CORBA, DCOM, .NET Remoting). I dati sono passati come parametri dei metodi di business con opportuni componenti di disaccoppiamento (Proxy e Adapter) per incapsulare/delimitare le zone integranti. Un grosso vantaggio di questo approccio è di essere type safety (ossia immune da errori relativi ai tipi) e di facile gestione per quanto riguarda la propagazione dei dati, delle eccezioni e dei codici d'errore.

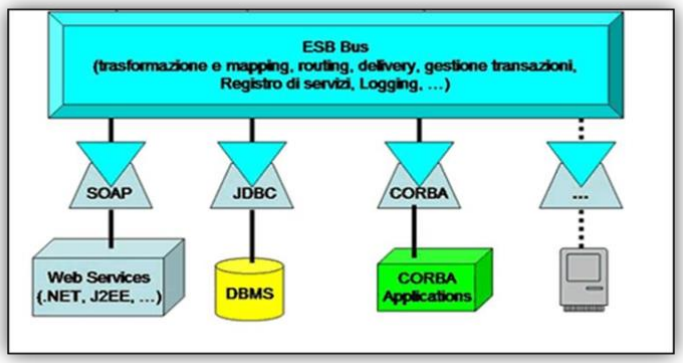

Figura 3.3 – Object/RPC integration

3. **Message Integration**: per integrare le applicazioni mediante la Message Integration, i dati sono inviati sotto forma di messaggi. Le applicazioni devono essere logicamente in grado di scambiarsi i messaggi (JMS, SOAP, ecc). Supponendo di dover integrare piattaforme diverse tra loro, come ad esempio J2EE e .NET, bisogna trovare un fattore comune di interoperabilità, basato ad esempio, su Web Services o più semplicemente su SOAP. Al contrario delle architetture basate su ORB, l'integrazione message-oriented permette un forte

disaccoppiamento tra i sistemi integranti e integrati per la natura intrinsecamente asincrona dei sistemi di messaging.

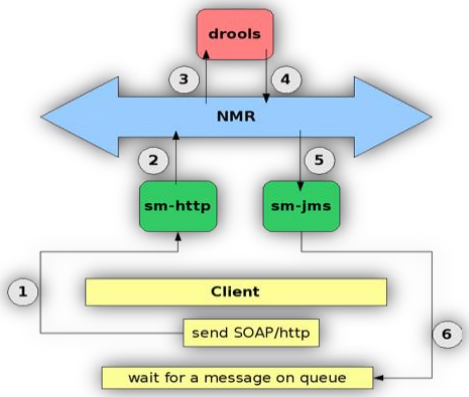

Figura 3.4 – Integrazione per messaggi

4. **Data Integration**: nel caso di Data Integration i dati sono passati tra i vari sistemi utilizzando il meccanismo data/record-oriented e adattati da un formato di un sistema all'altro. La Java Connector Architecture di SUN ricade proprio in questa categoria.

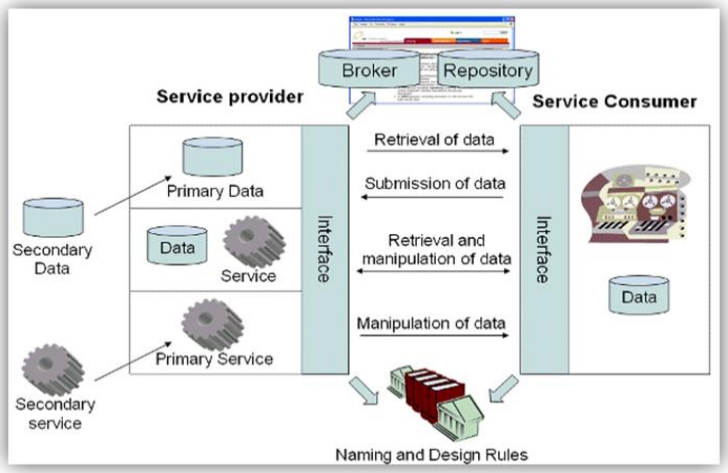

Figura 3.5 Service Provider e Service Consumer

Tra i vari aspetti da considerare in relazione all'integrazione delle applicazioni, sicuramente sta acquisendo sempre maggiore spessore un nuovo concetto, cioè quello del **middleware**.

Sfruttando il middleware, si ha la possibilità di poter utilizzare uno strumento molto efficace, mediante il quale è possibile integrare più applicazioni.

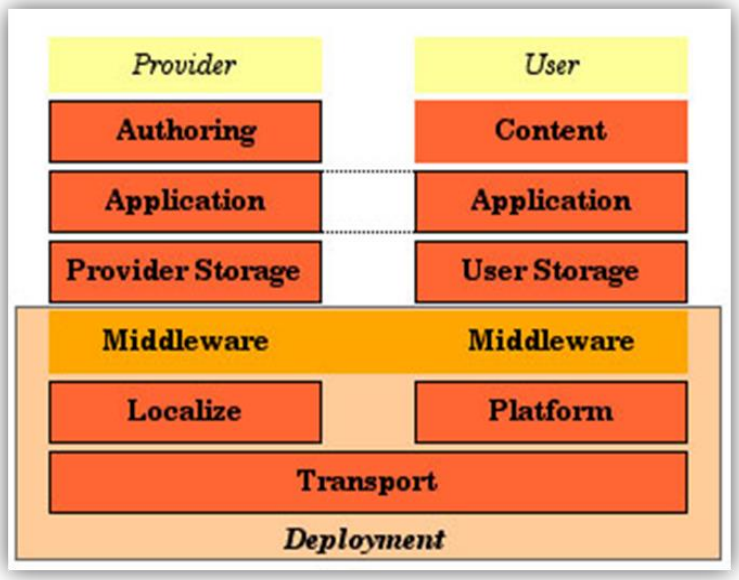

Figura 3.6 Middleware

Il termine Middleware letteralmente è "lo strato software posto nel mezzo". In linea con questa definizione, sono identificabili come Middleware un numero vastissimo di software completamente differenti fra loro (es. Application Server, MOM, JDBC, Screen Scraper).

I Middleware sono suddivisi in tre macro-aree:

- Basic Middleware.
- Platform Middleware.
- Integration Middleware.

I *Basic Middleware* forniscono architetture consistenti per l'integrazione tra dati ed applicazioni. Esempi di questa tipologia sono (Webshpere MQ, Sonic MQ, TIBCO/Rendezvous, FioranoMQ, Bea MessageQ, ecc) ed i Middleware di Data Management come ODBC e JDBC.

I *Platform Middleware* rappresentano la tipologia più complessa di Middleware, e tra questi troviamo gli Application Server (IBM WebSphere, BEA WebLogic, JBoss, ecc), i TP Monitor (BEA Tuxedo, IBM CICS, IMS), gli ORB (Borland VisiBroker, IONA ORBIX) ed i Portal Server.

Spesso gli Application Server forniscono, integrandole direttamente, le funzionalità tipiche di TP e ORB.

Gli *Integration Middleware* sono pensati invece per connettere database ed applicazioni sviluppati su architetture differenti. Esempi di questa tipologia

sono gli Integration Broker (IBM MQSeries Integrator; Tibco ActiveEnterprise, Microsoft BizTalk Server; Sonic ESB, Fiorano ESB, ecc), i Business Process Managers (IBM MQSeries Workflow, Tibco InConcert2000, Vitria Technology BusinessWare Automator, Fiorano BIS, Bea WLI), i Gateways (MOM, Database e Platform) e gli Adapter.

Le più comuni funzionalità fornite da un gruppo di prodotti necessari per integrare componenti sono:

- **Resource Adapter (RA)**: risolve i problemi di comunicazione. Quasi tutti i fornitori EAI (Enterprise Application Integration) forniscono Adapter proprietari (sincroni e asincroni) per la comunicazione con specifici EIS (Enterprise Application Software).
- **Messaging brokers**: tipicamente opera come messaging layer (MOM) e può operare sia in modalità point-to-point che publish/subscribe.
- **Integration Broker**: si occupa del data mapping dei dati e dell'intelligent routing (content-based e/o publish-and-subscribe). Di solito il data mapping è l'operazione più costosa nell'integrazione visto che tipicamente i dati acquisiti da un RA in un formato proprietario devono essere trasformati (sintatticamente e/o semanticamente) secondo le esigenze del business object richiedente.
- **Workflow**: in questo contesto si intende come il coordinamento di business object e messaggi nell'esecuzione di processi di business. I tool specializzati nella gestione del workflow di processi di business sono denominati Business Process Manager (BPM).
- **Monitor delle attività di Business:** l'analisi della storia degli eventi di business per tracciare l'accuratezza, la qualità e la sicurezza delle attività di business istanziate.

3.2 Tecnologie che realizzano l'integrazione applicazioni

Le tecnologie maggiormente utilizzate da chi si occupa di integrare applicazioni sono relative all'utilizzo delle API, dei web services e dell'architettura SOA.

Le applicazioni e i Web services devono quindi essere costruiti in maniera tale da poter permettere un semplice scambio di messaggi contenenti le funzioni o i dati elaborati attraverso di esse. La flessibilità è poca e la manutenzione è difficile, inoltre la riusabilità di questi servizi è praticamente nulla.

3.2.1 Le API

Le **A**pplication **P**rogramming **I**nterface (Interfaccia di Programmazione di un'Applicazione), sono “procedure” disponibili al programmatore, di solito raggruppate per formare un set di strumenti specifici per un determinato compito.

Le API permettono di evitare ai programmatori la scrittura di tutte le funzioni di cui si ha bisogno.

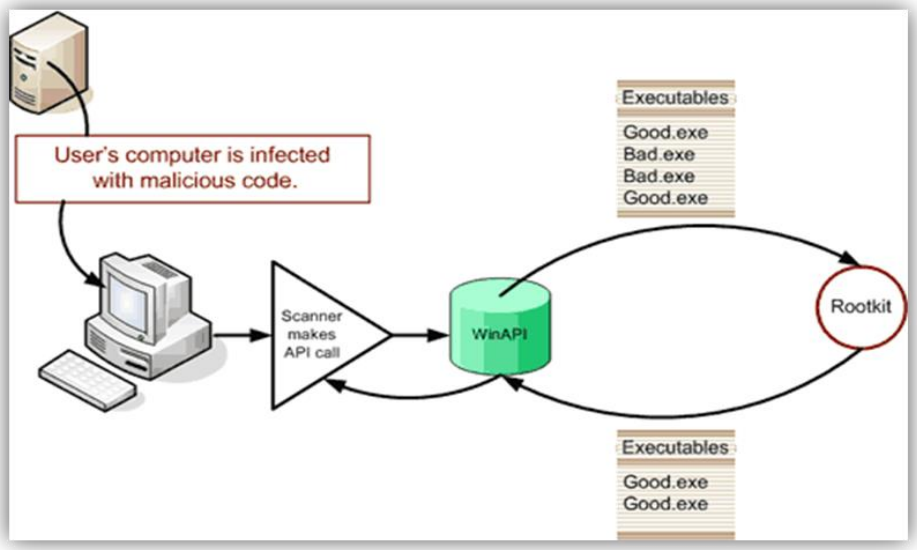

Figura 3.7 L'uso delle API

Le interfacce spesso consistono in una serie di funzioni, procedure, variabili e strutture dati. Una buona API fornisce una "scatola nera" o un livello di astrazione che evita al programmatore di sapere come funzionano ad un livello più basso.

Alcune API, come quelle standard di un sistema operativo, sono implementate come una libreria separata e distribuite con il sistema operativo. Altre richiedono a chi pubblica il software, di integrare l'API direttamente nell'applicazione. Questo costituisce un'ulteriore distinzione rispetto all'esempio precedente. Le API di Microsoft Windows sono fornite con il sistema operativo e chiunque può utilizzarle, a differenza dei software per i sistemi embedded, come le console per videogiochi, le quali generalmente ricadono nella categoria in cui le API sono integrate con l'applicazione.

Le **Windows API**, nonostante siano scritte, per ragioni di prestazioni, in un mix di linguaggio C e assembler, presentano un complesso modello orientato

agli oggetti con una struttura molto uniforme ed uno stile che è stato di ispirazione per molti altri progetti.

La struttura di base delle Windows API è rimasta pressoché invariata da Windows 1.0 ad oggi. Vi sono tre gruppi principali di API: *kernel*, *GDI* e *user*.

Le API *kernel* forniscono alle applicazioni un'interfaccia di alto livello ai servizi del kernel del sistema operativo (gestione della memoria, dei processi, sincronizzazione, ecc).

Le API *GDI* (*Graphics Device Interface*) costituiscono la libreria grafica dei sistemi Windows.

GDI virtualizza tutti i dispositivi grafici (monitor, stampanti, plotter) in modo da avere un'interfaccia omogenea (chiamata *Device Context*) tra le differenti tipologie di dispositivi. Le API *user* (da *user interface*) forniscono i servizi di interfaccia grafica, basati sui concetti di "finestra" e di "messaggio".

Un ulteriore esempio pratico è fornito dalle **Google Api.** Il celebre motore di ricerca Google, fra i tanti servizi che offre ha messo a disposizione degli utenti e degli sviluppatori una serie di API, da poter utilizzare gratuitamente. In particolare, è possibile scegliere fra diverse API, le quali offrono servizi di vario genere. Ad esempio, con la Google Search API, è possibile ricreare all'interno del proprio sito/sistema web, il potente motore di ricerca che utilizza Google stesso. Inoltre, sono presenti altre utilissime API come quella relativa all'utilizzo delle mappe (Google Maps API) ed una delle ultime rilasciate da Google, cioè la Google Contacts API, mediante la quale è possibile far dialogare le proprie applicazioni con il database della "rubrica" dei contatti degli utenti.

3.2.2 SOA

Una Service-Oriented Architecture (SOA) 0 è un modello architetturale per la creazione di sistemi residenti su una rete che focalizza l'attenzione sul concetto di servizio. Un sistema costruito seguendo la filosofia SOA e costituito da applicazioni, chiamate servizi, ben definite ed indipendenti l'una dall'altra, che risiedono su più computer all'interno di una rete (ad esempio la rete interna di una azienda o una rete di connessione fra più aziende che collaborano: intracompany e intercompany network). Ogni servizio mette a disposizione una certa funzionalità e può utilizzare quelle che gli altri servizi hanno reso disponibili, realizzando, in questo modo, applicazioni di maggiore complessità.

SOA e una forma particolare di Distributed System; un Distributed System consiste di vari agenti software distinti che devono lavorare insieme per svolgere alcuni compiti. Inoltre, gli agenti in un sistema distribuito non operano nello stesso ambiente di calcolo, quindi devono comunicare per mezzo di stack di protocolli hardware/software su una rete. Questo significa che le comunicazioni in un sistema distribuito sono intrinsecamente meno veloci e affidabili rispetto a quelle che utilizzano invocazione diretta del codice e memoria condivisa. Ciò ha importanti implicazioni architetturali perché i sistemi distribuiti richiedono che gli sviluppatori (di infrastruttura e applicazioni) considerino la latenza, fattore imprevedibile dell'accesso remoto, e tengano presenti questioni relative alla concorrenza e la possibilità di fallimenti parziali.

L'astrazione delle SOA non è legata ad alcuna specifica tecnologia, ma semplicemente definisce alcune proprietà, orientate al riutilizzo e all'integrazione in un ambiente eterogeneo, che devono essere rispettate dai servizi che compongono il sistema. In particolare, un servizio dovrà:

- Essere ricercabile e recuperabile dinamicamente. Un servizio deve poter essere ricercato in base alla sua interfaccia e richiamato a tempo di esecuzione. La definizione del servizio in base alla sua interfaccia rende quest'ultima (e quindi l'interazione con altri servizi) indipendente dal modo in cui è stato realizzato il componente che lo implementa.
- Essere auto contenuto e modulare. Ogni servizio deve essere ben definito, completo ed indipendente dal contesto o dallo stato di altri servizi.
- Essere definito da un'interfaccia ed indipendente dall'implementazione. Deve cioè essere definito in termini di ciò che fa, astraendo dai metodi e dalle tecnologie utilizzate per implementarlo. Questo determina l'indipendenza del servizio non solo dal linguaggio di programmazione utilizzato per realizzare il componente che lo implementa ma anche dalla piattaforma e dal sistema operativo su cui e in esecuzione: non è necessario conoscere come un servizio e realizzato ma solo quali funzionalità rende disponibili.
- Essere debolmente accoppiato con altri servizi (loosely coupled).
- Un'architettura e debolmente accoppiata se le dipendenze fra le sue componenti sono in numero limitato. Questo rende il sistema flessibile e facilmente modificabile.

- Essere reso disponibile sulla rete attraverso la pubblicazione della sua interfaccia (in un Service Directory o Service Registry) ed accessibile in modo trasparente rispetto alla sua allocazione. Essere disponibile sulla rete lo rende accessibile da quei componenti che ne richiedono l'utilizzo e l'accesso deve avvenire in maniera indipendente rispetto all'allocazione del servizio. La pubblicazione dell'interfaccia deve rendere noto anche le modalità di accesso al servizio.
- Fornire un'interfaccia possibilmente a "grana grossa" (coarse-grained). Deve mettere a disposizione un basso numero di operazioni, cioè poche funzionalità, in modo tale da non dover avere un programma di controllo complesso. Deve essere invece orientato ad un elevato livello di interazione con gli altri servizi attraverso lo scambio di messaggi. Per questo motivo e per il fatto che i servizi possono trovarsi su sistemi operativi e piattaforme diverse e necessario che i messaggi siano composti utilizzando un formato standard largamente riconosciuto (Platform Neutral). I dati che vengono trasmessi attraverso i messaggi possono essere costituiti sia dal risultato dell'elaborazione di un certo servizio sia da informazioni che più servizi si scambiano per coordinarsi fra loro.
- Essere realizzato in modo tale da permetterne la composizione con altri. Nell'architettura SOA le applicazioni sono il risultato della composizione di più servizi. E per questo motivo che ogni servizio deve essere indipendente da qualsiasi altro, in modo tale da ottenere il massimo della riusabilità. La creazione di applicazioni o di servizi più complessi attraverso la composizione dei servizi di base viene definita Service Orchestration.

Queste, dunque, le caratteristiche di un sistema di tipo SOA, di cui adesso passiamo a descrivere il funzionamento.

3.2.2.1 Come funziona una SOA e vantaggi dell'architettura

Gli attori di un sistema SOA sono quattro:

- Service Provider
- Service Consumer
- Service Contract
- Service Registry

Il **Service Provider** è un'entità che mette a disposizione un qualche servizio. Tale servizio, per poter essere trovato da altre entità che vogliono utilizzarlo, deve essere reso visibile sulla rete e quindi pubblicato. A tal fine il Service Provider comunica al Service Registry le informazioni relative al servizio, perché vengano memorizzate. Il Service Registry possiede quindi le informazioni, come URL e modalità di accesso, di tutti i servizi disponibili. Nel momento in cui un Service Consumer dovrà utilizzare un servizio farà a richiesta delle informazioni ad esso relative al Service Registry che fornirà il relativo Service Contract. Con queste informazioni il Service Consumer potrà comunicare direttamente con il Service Provider ed utilizzare il servizio. In figura sono riportate le interazioni fra le entità appena descritte.

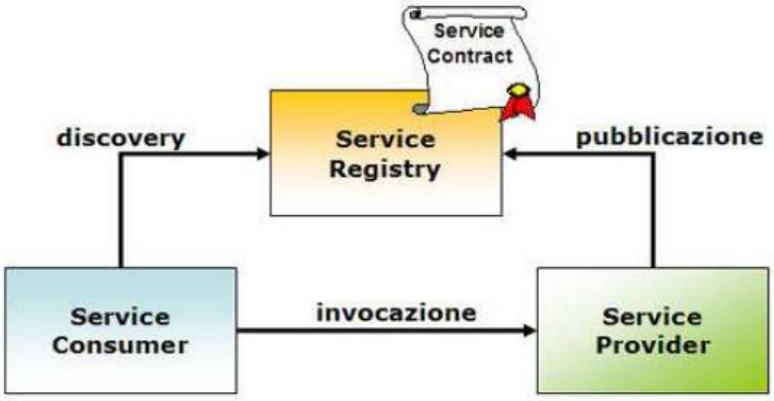

Figura 3.8 I ruoli nel SOA

Tutte queste interazioni passano attraverso quella che in figura viene genericamente definita Rete di Comunicazione, la quale in un'implementazione reale di una SOA può essere costituita sia da Internet sia da una intranet. SOA definisce, dunque, le caratteristiche che i componenti facenti parte di un sistema devono avere al fine di poter definire quest'ultimo un'architettura orientata ai servizi.

Spostandosi su un punto di vista più ampio e chiaro che l'adozione di un'architettura SOA investe tutti gli aspetti associati alla definizione di un Information System (metodologici, di processo, organizzativi ed anche tecnologici). L'obiettivo centrale e comunque quello di guidare la definizione di un'Architettura Enterprise ispirata e modellata in funzione dei driver di Business.

Possiamo quindi aggiungere che i servizi devono essere business-driven, quindi dovranno essere definiti assieme agli esperti funzionali e di dominio.

I requisiti di essere auto contenuti, stateless e coarse-grained orientano comunque il disegno in questa direzione e definire il servizio come componente che abbia realmente un significato per gli esperti funzionali (cioè non sia un componente architetturale fine a sé stesso) e importantissimo ai fini del riuso del servizio stesso. Il concetto di servizio e di composizione di servizi può essere quindi compreso in maniera semplice anche da chi tradizionalmente e al di fuori delle problematiche architetturali.

I servizi devono essere essenzialmente modellati in funzione del valore aggiunto che essi portano al core Business dell'azienda: non tutto deve essere necessariamente modellato in stile SOA.

D'altra parte, le problematiche trasversali (es. Sicurezza, Manutenibilità) dovrebbero essere viste dall'infrastruttura applicativa come servizi.

È bene notare come SOA non specifichi un'architettura in senso stretto, ma si ponga obiettivi molto più ampi:

- Definire pratiche e Roadmap di adozione che guidino l'evoluzione tecnologica e organizzativa dei Sistemi Enterprise;
- Specificare un insieme di regole per il disegno dei servizi con la definizione delle relative interfacce e delle modalità di interazione tra essi (sia per i nuovi servizi, sia per l'evoluzione delle piattaforme applicative esistenti);
- Definire pratiche per guidare gli impatti organizzativi e culturali indotti migliorare il livello di condivisione, comunicazione, coordinamento tra i ruoli che tradizionalmente si occupano di Business e gli Architect (es. attraverso la centralità del concetto di "Processo" di Business);
- Contribuire al soddisfacimento delle strategie IT di Business SOA di fatto non propone concetti architetturali nuovi, quanto una visione integrata ed ampliata di principi architetturali già adottati nel passato e di problematiche affrontate in altri ambiti.

SOA indirizza quindi in maniera corretta (anche se non rivoluzionaria) il design in termini di riutilizzo e di integrazione, di fatto producendo servizi con un maggiore valore aggiunto.

Riassumendo, i vantaggi reali dell'architettura SOA sono:

- Indirizzamento in maniera corretta del design di servizi in termini di riutilizzo e di integrazione;
- Non esistenza di alcun legame con nessuna tecnologia in particolare.

È importante notare che a livello implementativo la tecnologia utilizzata per lo sviluppo dei servizi SOA non è determinante finché vengono rispettate queste caratteristiche.

All'interno di un sistema SOA e possibile distinguere 7 layers come mostrato nella figura seguente.

1. Bottom layer: Contiene i sistemi operazionali (applicazioni e sistemi esistenti: ERP, CRM, Applicazioni legacy;
2. Component layer: basato su tecnologie a container e componenti;
3. Service Layer: fornisce i servizi basandosi sui componenti e attori degli strati sottostanti. Agli strati superiori vengono forniti i servizi;
4. Business Process Choreography: e lo strato che compone i servizi per implementare gli use case ed i processi di business;
5. Presentation: e lo strato che permette l'esposizione dei processi ottenuti dallo strato precedente tramite componenti di presentazione;
6. Integration Architecture: e l'infrastruttura che permette l'accesso e la composizione dei servizi fornendo meccanismi di integrazione. Gli ESB (Enterprise Service Bus) sono definiti a questo livello, che deve quindi gestire per ogni servizio i ruoli definiti (service consumer, provider, contract, registry);
7. QoS, Security, Management, Monitoring: tools di controllo e gestione dell'infrastruttura.

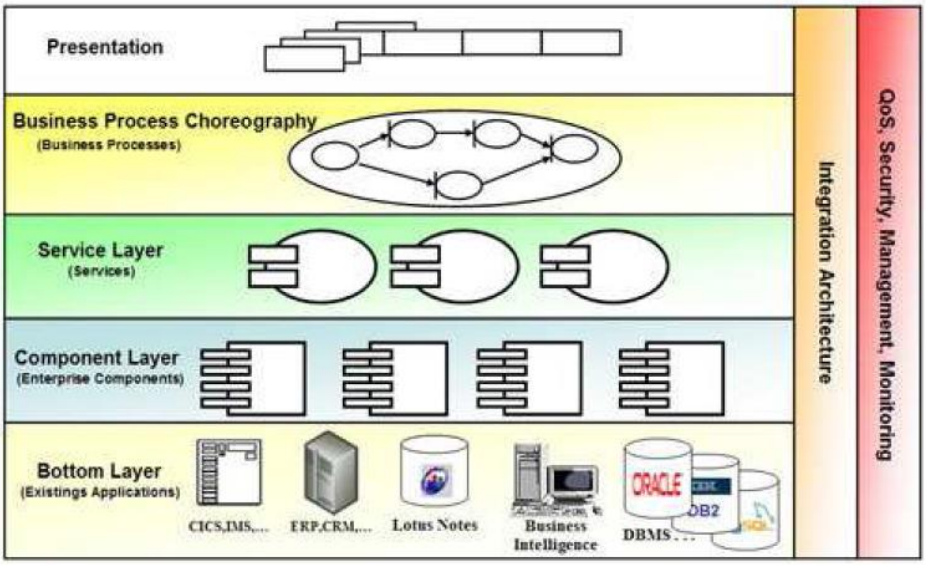

Figura 3.9 Layer SOA

Costruire un'applicazione SOA significa quindi identificare, disegnare, orchestrare servizi all'interno dei layers definiti in maniera il più possibile indipendente dalla tecnologia utilizzata.

Dopo aver descritto cos'e l'architettura SOA ed il funzionamento di un sistema di questo tipo, vediamo adesso cosa sono i Web Services, quali tecnologie utilizzano ed il loro legame con SOA.

3.2.3 Web Service: architettura service-oriented

Un **Web Service** è una componente software progettata per supportare l'interoperabilità tra diversi elaboratori dislocati in differenti reti.

I Web Service forniscono un'interfaccia (descritta in un formato standard, ovvero il Web Services Description Language) tramite la quale è possibile per mezzo di appositi "messaggi" inclusi in una "busta" (la più famosa è SOAP) utilizzare i servizi disponibili. I messaggi sono solitamente trasportati tramite il protocollo HTTP e formattati secondo lo standard XML.

Ne deriva che una delle caratteristiche alla base dell'utilizzo dei Web Service è il "disaccoppiamento" che l'interfaccia standard esposta dal Web Service rende possibile fra il sistema utente e il Web Service stesso. Questo ha un grande valore per chi si trova a dover lavorare con sistemi basati sui web service poiché le modifiche di cui necessitano i servizi o le applicazioni che li utilizzano possono essere attuate in maniera trasparente a patto che rimangano invariate le interfacce di comunicazione; tale flessibilità consente la creazione di sistemi software complessi costituiti da componenti svincolati l'uno dall'altro e consente una forte riusabilità di codice ed applicazioni già sviluppate. Infatti, i web service e lo sviluppo di SOAP hanno consentito a differenza di EDI (Electronic Data Interchange), RPC (Remote Procedure Call), e altri tipi di API, la creazione di interfacce per l'utilizzo di funzionalità distribuite in rete.

Il consorzio OASIS e il W3C sono i principali responsabili dell'architettura e della standardizzazione dei Web Service.

L'architettura dei web service può essere meglio descritta dalla cosiddetta pila protocollare, riportato nella figura seguente:

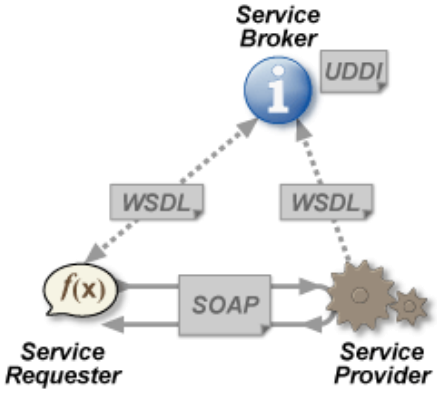

Figura 3.10 Pila protocollare dei web services

La pila protocollare dei Web Service è l'insieme dei protocolli di rete utilizzati per definire, localizzare, realizzare e far interagire tra di loro i Web Service. Essa è principalmente composta da quattro aree:

- Trasporto del servizio: responsabile per il trasporto dei messaggi tra le applicazioni in rete, include protocolli quali HTTP, SMTP, FTP, XMPP ed il recente Blocks Extensible Exchange Protocol (BEEP).
- XML Messaging: tutti i dati scambiati sono formattati mediante tag XML in modo che gli stessi possano essere utilizzati ad entrambi i capi delle connessioni; il messaggio può essere codificato conformemente allo standard SOAP, come anche utilizzare JAX-RPC, XML-RPC o REST.
- Descrizione del servizio: l'interfaccia pubblica di un Web Service viene descritta tramite WSDL (Web Services Description Language). Si tratta di un linguaggio basato su XML usato per la creazione di documenti descrittivi delle modalità di interfacciamento ed utilizzo del Web Service.
- Elencazione dei servizi: si tratta della centralizzazione della descrizione e della localizzazione dei Web Service in un registro comune, la quale permette la ricerca ed il reperimento in maniera veloce dei Web Service disponibili in rete; a tale scopo viene attualmente utilizzato il protocollo UDDI.

Oltre queste quattro aree vengono utilizzati altri protocolli standard quali:

- WS-Security: il protocollo Web Services Security protocol è stato adottato come standard OASIS; tale standard permette l'autenticazione degli utenti e la confidenzialità dei messaggi scambiati con l'interfaccia del Web Service;

- WS-Reliability: si tratta di specifiche basate su SOAP e accettate come standard OASIS che soddisfano la richiesta di messaggi affidabili, la quale costituisce una richiesta critica per alcune delle applicazioni che utilizzano i Web Service (come, ad esempio, transazioni monetarie o applicazioni di E-commerce).

Accanto al concetto di web service, risultano fondamentali altri tre concetti: WSDL, UDDI e SOAP.

3.2.3.1 Web Services Description Language

La creazione della documentazione è una delle parti che richiede più tempo nei progetti di sviluppo di applicazioni web e, spesso, chi non è coinvolto in questo processo fatica a comprenderne l'importanza per la manutenzione e l'ulteriore sviluppo dei sistemi. Il valore di questo processo, però, risulta evidente soprattutto nel momento in cui un gruppo esterno a quello di sviluppo tenta di riutilizzare o migliorare l'applicazione.

Allo scopo di generare una documentazione funzionale, nasce il linguaggio formale **Web Services Description Language (WSDL).** Il **WSDL** viene strutturato seguendo le caratteristiche dell'XML, utilizzato per la creazione di documenti per la descrizione di Web Service.

Mediante il WSDL può essere descritta l'interfaccia pubblica di un Web Service, ovvero creata una descrizione basata su XML, di come interagire con un determinato servizio. Un documento WSDL contiene, relativamente al Web Service descritto, informazioni su:

- cosa può essere utilizzato (le operazioni messe a disposizione dal servizio);
- come utilizzarlo (il protocollo di comunicazione da utilizzare per accedere al servizio, il formato dei messaggi accettati in input e restituiti in output dal servizio ed i dati correlati), ovvero i vincoli (bindings) del servizio;
- dove utilizzare il servizio (cosiddetto endpoint del servizio che solitamente corrisponde all'indirizzo – in formato URI – che rende disponibile il Web Service).

Le operazioni supportate dal Web Service e i messaggi che esso può scambiare con l'esterno sono descritte in maniera astratta e quindi collegate ad uno specifico protocollo di rete e ad uno specifico formato.

Un documento WSDL è costituito da 4 elementi principali:

- portType: le operazioni (i metodi) esposti dal WebService;
- message: i messaggi utilizzati dal servizio;
- types: i tipi di dato utilizzati dal servizio;
- binding: i protocolli di comunicazione utilizzati.

Esistono quattro tipi possibili di operazioni:

- One-way: l'operazione riceve una richiesta ma non fornisce risposta;

Esempio "One-way"

```
<message name="newTermValues">
  <part name="term" type="xs:string"/>
  <part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
      <operation name="setTerm">
            <input                              name="newTerm"
message="newTermValues"/>
      </operation>
    </portType >
```

In questo esempio è possibile notare come l'operazione "setTerm" permette l'utilizzo di un parametro in entrata ma non restituisce alcun output.

- Request-response: l'operazione riceve una richiesta e fornisce una risposta.

Esempio "Request-response"

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
    <operation name="getTerm">
      <input message="getTermRequest"/>
     <output message="getTermResponse"/>
     </operation>
</portType>
```

In questo esempio è possibile notare come l'operazione "getTerm" richiede in input un messaggio chiamato "getTermRequest" e restituisce in output un messaggio chiamato "getTermResponse".

- Solicit-response: l'operazione spedisce una richiesta e attende una risposta.
- Notification: l'operazione spedisce una richiesta e non attende risposta.

La formalizzazione di un contratto di servizio avviene attraverso l'utilizzo di una specifica standard che prende il nome di WSDL (Web Services Description Language). Il WSDL permette l'utilizzo automatico di un servizio, ovvero se due servizi utilizzano la stessa interfaccia (WSDL) allora possono essere richiamati in alternativa con la sola limitazione che rispondono con nomi diversi (indirizzi web differenti). Ne consegue che se si dispone di più servizi concorrenti, l'AI che li utilizza ha maggiore flessibilità, nel senso che

possono essere implementate policy di utilizzo che dipendentemente dalla disponibilità, dal carico di lavoro e dalle esperienze pregresse decidono quale tra i servizi utilizzare.

3.2.3.2 Universal Description Discovery and Integration

L'UDDI (Universal Description Discovery and Integration), è uno degli standard alla base del funzionamento dei web service, ed è definibile come un "registry", cioè una base di dati ordinata e indicizzata, basata su XML e indipendente dalla piattaforma hardware. Esso permette alle aziende la pubblicazione dei propri dati e dei servizi offerti su internet.

UDDI è un'iniziativa Open Source, sviluppata a cavallo tra il 1999 e il 2000, sponsorizzata dall'OASIS, la quale offre la possibilità di ricercare, scoprire ed interrogare servizi offerti sul web, adottando le procedure stabilite per il loro utilizzo, da parte delle stesse aziende che li mettono a disposizione di tutti.

Una "registrazione" UDDI consiste, infatti, di tre diverse componenti:

- Pagine bianche (White Pages): indirizzo, contatti (dell'azienda che offre uno o più servizi) e identificativi;
- Pagine gialle (Yellow Pages): categorizzazione dei servizi basata su tassonomie standardizzate;
- Pagine verdi (Green Pages): informazioni (tecniche) dei servizi forniti dall'azienda.

3.2.3.3 Simple Object Access Protocol

SOAP (Simple Object Access Protocol), è un protocollo leggero per lo scambio di messaggi tra componenti software. La parola object lascia intendere che l'uso del protocollo dovrebbe effettuarsi secondo il paradigma della programmazione orientata agli oggetti.

SOAP è la struttura operativa estensibile e decentralizzata che può operare sopra varie pile protocollari per reti di computer. I richiami di procedure remote possono essere modellati come interazione di diversi messaggi SOAP.

Nonostante SOAP possa utilizzare tutti i protocolli di Internet, il protocollo HTTP è il più comunemente utilizzato e l'unico ad essere stato standardizzato dal W3C. Uno dei punti di forza di SOAP sta nel fatto che si

basa sul metalinguaggio XML e la sua struttura segue la configurazione Head-Body, analogamente all'HTML. Il segmento opzionale Header contiene meta-informazioni come quelle che riguardano il routing, la sicurezza e le transazioni. Il segmento obbligatorio Body trasporta il contenuto informativo e questo deve seguire uno schema definito dal linguaggio XML Schema.

Esempio richiesta informazioni di un client

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails
xmlns="http://warehouse.example.com/ws">
      <productId>827635</productId>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

Esempio risposta del server

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse
xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productName>Toptimate 3-Piece Set</productName>
        <productId>827635</productId>
          <description>3-Piece luggage set.  Black Polyester.
```

```
        </description>
       <price>96.50</price>
       <inStock>true</inStock>
     </getProductDetailsResult>
   </getProductDetailsResponse>
 </soap:Body>
</soap:Envelope>
```

3.2.4 Web Service: ReSTful

Fino ad ora sono stati descritti i servizi web orientati a SOA come protocollo di comunicazione per l'integrazione di applicazione orientata a metodi ed interfacce (application-to-application) che utilizzano standard definiti dal W3C per comunicare e interagire (WSDL, SOAP, UDDI). Questi non coprono tutte le possibili metodiche di progettazione per i servizi web. Esiste, infatti, un altro tipo di realizzazione per servizi basato su un architettura ReST, la cui unica differenza tra un servizio ReSTful e un normale sito internet è il destinatario dei messaggi di risposta, un servizio è costruito per essere utilizzato direttamente da altre applicazioni alle quali sono indirizzate le informazioni contenute in questi messaggi che saranno poi elaborate, di contro un sito web è costruito per essere visualizzato da un utente e le informazioni per essere lette da persone.

Un servizio web ReSTful, infatti, è completamente comprensibile semplicemente guardando il tipo di richiesta HTTP e a chi è indirizzata.

I servizi REST (chiamati **Representational State Transfer**) sono stati presentati e trattati nel 2000 da Roy Thomas Fielding come argomento della sua tesi di dottorato (PhD) "Architectural Styles and the Design of Network-based Software Architecture", questo stile architetturale è stato sviluppato in parallelo allo standard HTTP 1.1, estensione della versione precedente, a cui contribuì Fielding stesso.

I servizi di tipo ReST sono visti come rivoluzionari in quanto in grado di rendere il codice riutilizzabile e i sistemi molto più semplici da realizzare.

Uno dei casi di studio utilizzati come riferimento per la reingegnerizzazione di una applicazione per mezzo di servizi ReST è riportato in “Reengineering Legacy Systems with ReSTful Web Service” 0.

In estrema sintesi i motivi chiave che hanno reso i ReST vincenti rispetto ai SAOP sono:

- la riusabilità delle funzionalità. Il riuso del codice è l’aspetto fondamentale di questo lavoro ed è un aspetto che con SOAP non è possibile ottenere.

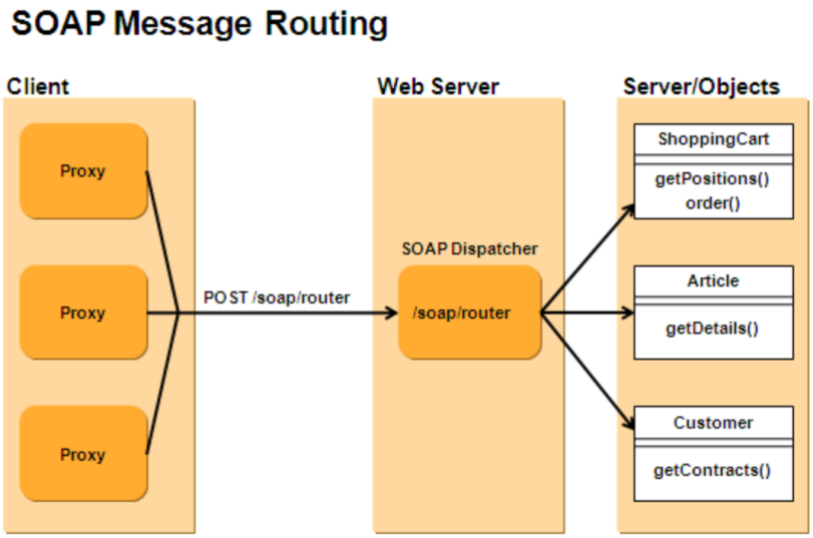

Figura 3.11 SOAP Message Routing

- Possibili “mash-up”. I mash-up consentono di creare applicazioni composite e consentono l'integrazione rapida di applicazioni web.
- La rapidità e semplicità nell’effettuare aggiornamenti e modifiche anche in questo è una caratteristica fondamentale dei servizi REST.

Nella logica REST esistono alcune difficoltà come, ad esempio, quella di riuscire a identificare le risorse e vedere a queste non come dati ma come “cose RESTful”. Le risorse sono, per superare tale ostacolo, identificabili tramite URI che devono essere progettati per avere un’alta interoperabilità e scalabilità.

3.2.5 I principi REST

Lo stile architetturale REST richiede che sei principi (anche chiamati vincoli) vengano rispettati affinché un sistema rispecchi tale architettura:

- **Client-Server**: I client ed il server sono **separati** gli uni dagli altri. In questo modo i client non dovranno preoccuparsi della **persistenza** dei dati, migliorando così la **portabilità** del codice, allo stesso modo il server non dovrà preoccuparsi di doversi **interfacciare** correttamente con ogni client, favorendo così in termini di **scalabilità** che sarà più facile.
- **Uniformità delle interfacce**: Risorse univoche sono identificare da URL e sono esse stesse differenti dalla rappresentazione inviate al client (XML, JSON, HTML). Il client può manipolare una specifica risorsa attraverso le rappresentazioni fornite a cui essi hanno accesso (permessi). Ogni messaggio scambiato tra il client ed il server è autodescrittivo ed include abbastanza informazioni per descrivere come debba essere processato.
  Gli ipermedia, siano essi ipertesti o collegamenti fra di essi fungono come mezzo per il trasferimento dello stato.
- **Interazioni stateless**: nessuno contesto relativo al client è memorizzato lato server durante le richieste. Tutte le informazioni necessarie al servizio sono contenute nel URL, parametri (query), corpo (body) o intestazioni (header).
- **Conservabilità (cacheability)**: i client possono conservare le risposte. Esse dovranno definirsi tali o no per prevenire che il client invii dati inopportuni durante le richieste successive.
- **Stratificazione**: In ogni momento un client non è a conoscenza di essere connesso ad un servizio finale (endpoint) o ad uno intermedio. Questo strato aiuta a far rispettare le politiche di sicurezza e migliorare la scalabilità del servizio attraverso funzionalità di bilanciamento del carico (load balancing)
- **Codice su richiesta**: Opzionalmente, il server può estendere la funzionalità di un client trasferendo ad esso codice eseguibile.

3.2.6 REST ed il protocollo HTTP

Lo stile architetturale REST, figura seguente, a differenza del SOAP non è indipendente dal protocollo di trasporto su cui esso è trasmesso. Abbiamo

visto come SOAP possa viaggiare attraverso HTTP come su SMTP, i sistemi REST invece sono progettati sul protocollo HTTP.

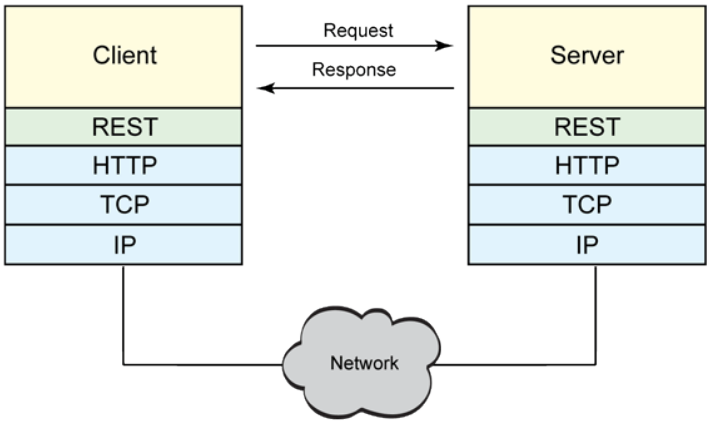

Figura 3.12 Architettura REST tramite HTTP

Abbiamo in precedenza elencato i vincoli posti dalle architetture di tipo REST: per soddisfare completamente tali vincoli è stato scelto, ed esteso, uno dei protocolli in più in uso, se non vitale, che li soddisfacesse tutti: il protocollo HTTP.

Si può dire che REST non sia nient'altro se non la formalizzazione dell'architettura del web stesso, dove gli URL identificano univocamente le risorse ed i metodi (o verbi) forniti dal protocollo stesso ne permettano la manipolazione CRUD delle stesse mentre i codici di stato comunichino, appunto, lo stato al client (ricordiamo che il mantenimento dello stato è compito dei client). Per fare ciò si è reso necessario estendere lo standard in uso ai tempi (1.0)

I principali metodi (verbi) utilizzati dall'architettura sono:

- **GET: Legge** una risorsa;
- **POST: Crea** una nuova risorsa;
- **PUT: Aggiorna** una risorsa esistente;
- **DELETE: Cancella** una risorsa esistente. Quest'ultima operazione è idempotente, ovvero diverse richieste di cancellazione della medesima risorsa, verranno interpretate come una singola.

Mentre per quanto riguarda i codici di stato essi sono identificati da numeri composti da 3 cifre, di cui la prima indica la categoria di stato:

- **Informativi (1xx)**: Introdotta dalla versione 1.1, questa classe di codici indica una risposta provvisoria da parte del server.
- Composta da una sola entità "Status-Line" e terminata da una linea vuota, può contenere ulteriori header opzionali.
- **Successo (2xx)**: Indicano che la richiesta del client è stata ricevuta, interpretata ed accettata.
- **Redirezione (3xx)**: Questa classe di codici indica che è necessaria un'ulteriore azione da parte del client per poter soddisfare la richiesta. L'azione richiesta poterebbe essere svolta dal client stesso senza l'interazione dell'utente se e solo se il metodo usato nella seconda richiesta è GET o HEAD. Il client dev'essere in grado di riconoscere cicli di redirezione infiniti che potrebbero generare traffico di rete elevato dovuto al altrettanto elevato di redirezioni. Nella versione precedente questo limite era fissato a 5 mentre ora è compito del client implementare tale limite.
- **Richieste malformate (4xx)**: Dovuto a richieste non valide come risorse inesistenti, campi dello header mancanti, ecc.
- **Errori interni (5xx)**: Causate da errori avvenuti all'interno del server mentre cercava di soddisfare la richiesta. In questa categoria ricadono errori dovuti a funzionalità non implementate (501), sovraccarico o manutenzione del server (502) o mancato supporto alla versione HTTP richiesta (505).

I codici di stato più comuni sono:

- **OK (200):** Ritornato dai verbi GET, PUT e DELETE. Conferma che la richiesta effettuata ha avuto successo.
- **Created (201):** Ritornato dal verbo POST. La richiesta è stata soddisfatta e la nuova risorsa è stata creata. La nuova risorsa creata può essere referenziata dall'identificatore univoco (URI) contenuto nel campo Location nella risposta.
  La risposta **deve** includere una lista di posizioni dalle quali il client può scegliere quella più appropriata. Il formato dell'entità creata è inoltre specificato dal campo Content-Type nell'intestazione della risposta.
  Se l'elaborazione della richiesta comporta la necessità di ulteriore tempo da parte del server, il server stesso provvederà a ritornare un altro codice di accettazione della richiesta (**202 - Accepted**).
  Mostriamo ora alcuni esempi di quanto detto. Il client richiede la creazione di una risorsa (in questo caso un link per un feed atom):

```
POST /edit/ HTTP/1.1
Host: example.org
User-Agent: Thingio/1.0
Authorization: Basic ZGFmZnk6c2VjZXJldA==
Content-Type: application/atom+xml;type=entry
Content-Length: nnn
Slug: First Post

<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
 <title>Atom-Powered Robots Run Amok</title>
 <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
 <updated>2003-12-13T18:30:02Z</updated>
 <author><name>John Doe</name></author>
 <content>Some text.</content>
</entry>
```

Il server risponde con codice 201 comunicando la nuova risorsa:

```
HTTP/1.1 201 Created
Date: Fri, 7 Oct 2005 17:17:11 GMT
Content-Length: nnn
Content-Type: application/atom+xml;type=entry;charset="utf-8"
Location: http://example.org/edit/first-post.atom
ETag: "c180de84f991g8"
<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
 <title>Atom-Powered Robots Run Amok</title>
```

<id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>

<updated>2003-12-13T18:30:02Z</updated>

<author><name>John Doe</name></author>

<content>Some text.</content>

<link rel="edit"

href="http://example.org/edit/first-post.atom"/>

</entry>

- **No Content (204):** Ritornata dal metodo POST. Sta ad indicare che il server ha soddisfatto la richiesta e la risposta non contiene nessuno corpo ma potrebbe contenerne negli header. Questo genere di risposta indica al client che i cambiamenti richiesti sull'entità sono avvenuti senza che lui debba necessariamente richiedere nuovamente richiedere al server la stessa.

- **Not found (404):** Ritornabile da tutti i metodi quando la risorsa a cui si fa riferimento non è presente.
  In tal caso si avrà una risposta simile a questa:

  HTTP/1.1 404 Not Found

  Date: Sat, 28 Nov 2015 17:23:39 GMT

  Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.1e-fips PHP/5.4.16

  Content-Length: 233

  Connection: close

  Content-Type: text/html; charset=iso-8859-1

  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

  <title>404 Not Found</title>

  <h1>Not Found</h1>

The requested URL was not found on the server. If you entered

the URL manually please check your spelling and try again.

- **Conflict (409)**: Ritornata dal metodo POST o PUT, indica che la richiesta non può essere completata per un conflitto con lo stato corrente della risorsa. Questo codice prevede che sia l'utente a risolvere tale conflitto e rinviare la richiesta contenente la risorsa corretta. Le differenze tra lo stato noto attualmente del client e quello presente sul server sono ritornate nella risposta nel formato definito dal campo Content-Type dell'header.

In conclusione, a valle dell'inquadramento dei servizi ReSTful possiamo concludere dicendo che nel contesto ReSTful si possono identificare due possibilità con cui cliente e fornitore comunicano:

- servizi ibridi: rispetta tutti i vincoli definiti da ReST. Vengono visti come una via di mezzo tra servizi ROA e RPC.
- servizi Resource-Oriented: pienamente aderenti ai principi definiti dall'architettura ROA. Ogni funzionalità e informazione offerta dal servizio viene esposta come risorsa.

Nella pratica quando ci si riferisce ad un servizio ReSTful si prende in considerazione la seconda tipologia di implementazione nonostante in alcuni casi esistono implementazioni che utilizzano i servizi ibridi. Questo avviene perché i servizi ibridi non sfruttano appieno le potenzialità offerte dallo HTTP, caratteristica che invece è il punto di forza dei servizi basati su architetture Resource-Oriented.

3.3 Resource Oriented Architecture

Il Web, col passare degli anni per rispondere alle esigenze dei degli utenti si è molto evoluto e trasformato mutando la sua conformazione di "semplice" contenitore di pagine statiche in una infrastruttura "rete" di connessione di risorse digitali e fisiche (Internet Of Things).

Oggi è normale pensare che tutto può essere connesso alla rete e può scambiare dati con altri e questo nuovo modo di vedere il Web, ha portato ad un nuovo approccio architetturale chiamato Resource Oriented Architecture.

Resource Oriented Architecture (conosciuta anche con l'acronimo di ROA), è un architettura software specifica per i servizi ReSTful il cui scopo è di rendere questi ultimi più simili possibile al resto del web e per questa ragione in ROA sono presenti i principi ReST e regole non per la progettazione web utili a rendere espliciti e chiari gli aspetti sino a questo momento lasciati alla libera interpretazione dei progettisti) 0.

Come per ReST, la risorsa è il fondamento della Resource Oriented Architecture. Essa identifica tutti i concetti che vengono esposti come servizi e fornisce un punto di accesso per il loro utilizzo (URI). In ROA qualsiasi funzionalità offerta da un servizio deve essere esposta come risorsa. Questo significa che la caratteristica principale in comune a tutti i servizi ROA è la modalità con cui il client comunica al server le sue intenzioni. Difatti, ogni richiesta deve essere conforme ad una data struttura e deve fornire le informazioni necessarie al server per soddisfare le richieste suddivise nel modo seguente.

- HTTP fornisce method information, il quale definisce lo scopo della richiesta. Il messaggio di richiesta HTTP formato da quattro parti: riga di richiesta (request line), sezione header (informazioni aggiuntive), riga vuota (CRLF: i 2 caratteri carriage return e line feed), body (corpo del messaggio) e nella request line indica quale sarà l'azione eseguita (ovvero il metodo) a seguito della soddisfazione della richiesta oltre a URI e versione del protocollo.
- L'URI deve fornire scoping information ovvero indicare l'oggetto della richiesta (ad esempio la pagina web che si intende ottenere) su quale risorsa il client intende operare e gli eventuali parametri necessari.

Anche per le Resource Oriented Architecture il fulcro attorno al quale ruota tutto è la risorsa. Dato che una risorsa è esposta pubblicamente tramite un URI, un servizio di questo tipo fornisce un URI per ogni aspetto utile, dando al client un punto di accesso diretto. Questa caratteristica è definita URI-addressable resources e rappresenta una delle proprietà più importanti per l'utente finale. Questa proprietà consente al client di accedere in modo diretto alla risorsa, in mancanza di questa sarebbe

impossibile accedere alla risorsa a meno che non si riesca a trovare un modo alternativo indiretto.

Scendendo su un piano più pragmatico, pensiamo ad una applicazione che ha necessità di accedere direttamente ai risultati di una ricerca inviando una sola richiesta se ci sono risorse indirizzabili farlo è semplice, ad esempio utilizzando i servizi messi a disposizione da Google (https://www.google.com/search?q=resource+oriented+architecture) se la risorsa ricerca di google invece non fosse identificata con un URI questo non sarebbe possibile.

Inoltre, grazie alla caratteristica di addressability che rende possibile recuperare in qualsiasi momento tramite un riferimento definito una risorsa si possono adottare i meccanismi classici di cache che consentono di ottenere un risparmio della banda utilizzata. In mancanza della caratteristica di addressability pur salvando le risorse nella cache non si potrebbero riesumare in quanto non ci sarebbe una associazione tra quanto salvato e la richiesta che si sta effettuando.

- ogni URI, nella architettura ROA, deve descrive chiaramente la risorsa a cui si riferisce e deve avere una struttura che varia in modo chiaro e prevedibile. Questa pur non essendo un principio previsto per i servizi ReST risulta essere ampiamente utilizzato sul web anche per i servizi ReST allo scopo di facilitare l'utilizzo delle risorse messe a disposizione dai servizi.
- ogni risorsa, lato server, può essere implementata in modo tale da avere diverse rappresentazioni, ad esempio esporre la data corrente diversi formati. In questo caso, il client che vuole utilizzare la risorsa, effettuare la richiesta, deve essere in grado di far sapere al server a quale delle rappresentazioni possibili della risorsa vuole accedere. Il client per richiedere la rappresentazione desiderata ha due possibili strategie da perseguire.
    - Includere all'interno del campo accept dell'header HTTP il formato con cui il server deve rispondere.
    - Includere il formato richiesto all'interno dell'URI, ciò significa che quest'ultimo contiene tutto quello che è necessario al server per soddisfare la richiesta.
      Relativamente alla architettura Resource Oriented il metodo preferenziale è quello dell'URI, anche se questa strategia rischia di rendere poco chiara l'identificazione delle risorse tramite URI in quanto due differenti URI (due nomi diversi)

identificheranno la stessa risorsa. Questo problema viene superato esponendo un URI che fa riferimento alla risorsa standard denominata “forma platonica” da utilizzarsi nel caso in cui ci si voglia riferire alla risorsa senza badare al formato.

- ogni richiesta è autonomica e isolata. Questo significa che ogni richiesta deve contenere tutte le informazioni necessarie per permettere al server di soddisfarla. Il server per poter soddisfare la richiesta non deve avere bisogno di altre informazioni oltre a quelle presenti nella chiamata né tantomeno avere bisogno di ricordare chiamate precedenti.
- L’assenza della memoria di stato (stateless) è la caratteristica che l’architettura ROA utilizza del protocollo HTTP che utilizza come canale di comunicazione.
  In aggiunta a questo la proprietà di stateless consente di migliorare le prestazioni infrastrutturali dei fornitori di servizi. Difatti, essendo ogni richiesta indipendente da altre richieste ed essendo autonomica è possibile distribuire i servizi su più server senza doversi preoccupare di gestire dipendenze tra i diversi servizi.
- Completezza ReSTful. La risposta fornita da un servizio ReSTful solitamente, se necessario, oltre a contenere i dati richiesti contengono i riferimenti per accedere alle risorse collegate. In altre parole, nella risposta del server sono riportati tutti gli URI che servono per accedere ad altri stati della risorsa stessa oppure ad altre risorse collegate.
  Un servizio per essere totalmente ReSTful restituisce l’elenco di tutte le risorse collegate e questo assicura la scalabilità del sistema. In mancanza di questa caratteristica, e quindi nel caso in cui le risorse collegate non siano comprese nella risposta, il server dovrebbe fornire, al client, informazioni con metalinguaggi o altro per consentirgli in autonomia e automaticamente di costruire gli URI delle risorse collegate. In questo caso, il client dovrebbe essere in grado di capire il metalinguaggio e di automatizzare la creazione degli URI e questa creazione non potrebbe che essere effettuata ad-hoc di volta in volta. È evidente che nel servizio viene modificato anche la sola modalità di costruzione del pattern dell’URI si dovrà provvedere a modificare anche il client in modo tale che riesca a ricostruire il nuovo pattern, ottenendo nei fatti un forte accoppiamento tra client e server.

Lo scopo di questa proprietà è proprio quello di risolvere il problema dell'accoppiamento client e server fornendo dei link diretti URI che sono immediatamente utilizzabili dai client.

- Standardizzazione servizi resource oriented. Gli utenti che utilizzano i servizi RO devono essere in grado di prevedere il risultato di ogni metodo a loro disposizione, ovvero ognuno degli utenti deve accedere ad una risorsa nel medesimo modo. A tale scopo ROA definisce una specifica interfaccia uniforme che fornisce un significato ad ogni metodo 0:
  - GET: serve per ottenere una rappresentazione dello stato della risorsa di destinazione.
  - PUT: richiesta di creazione o modifica di una risorsa. Imposta lo stato della risorsa di destinazione sullo stato definito dalla rappresentazione inclusa nella richiesta.
  - DELETE: richiesta di eliminazione una risorsa.
  - POST: serve per richiedere alla risorsa di destinazione l'elaborazione della rappresentazione racchiusa nella richiesta. Viene utilizzato per creare una nuova risorsa da una già esistente, quest'ultima assume il ruolo di genitore di quella nuova.

Per concludere, considerando che per il contesto in cui i-mole si muove, l'unica scelta possibile è di utilizzare servizi web che consentono l'integrazione ed il riuso di componenti software orientati alle risorse, e quindi utilizzare una architettura ROA e servizi di tipo ReST. In questa visione, un ulteriore elemento atto ad agevolare il riuso nella piattaforma i-mole è la tecnologia swagger che serve per documentare i servizi ReST e agevolare il lavoro di integrazione effettuato dai progettisti del sistema. Di seguito si riportano le motivazioni che ci hanno portato a scegliere lo swagger come strumento di documentazione dei servizi ReST API.

3.3.1 OpenAPI Specification (Swagger)

Abbiamo visto come le Application Programming Interface sono il presupposto principale nel momento cui si vuole realizzare un'architettura scalabile e manutenibile. Le API in questa visione sono anche lo strumento utilizzato per consentire a terze parti di utilizzare alcuni nostri servizi oppure di estendere le funzionalità dei sistemi software aziendali. Ad esempio, nel caso in cui si volesse integrare nel proprio sito web, e-

commerce o altro una mappa per dare indicazioni, geolocalizzate della propria posizione o della propria azienda, agli utenti questo può essere fatto in modo semplice e veloce utilizzando le API di Google che consentono di integrare nei propri sistemi le mappe di Google Maps.

A livello commerciale moltissime aziende hanno, fatto delle API un importante asset aziendale ad altro reddito, a titolo di esempio basti pensare ad Amazon, noto per il negozio online, che nei servizi web ha uno dei suoi core asset aziendali più importanti, figura seguente. AWS è l'acronimo di Amazon Web Services che identifica l'insieme dei servizi che la stessa Amazon vende a terzi, tra cui ci sono:

- Calcolo
- Rete
- Distribuzione dei contenuti
- Archiviazione e distribuzione dei contenuti
- Basi di dati
- Deployment
- Servizi applicativi
- Analitiche

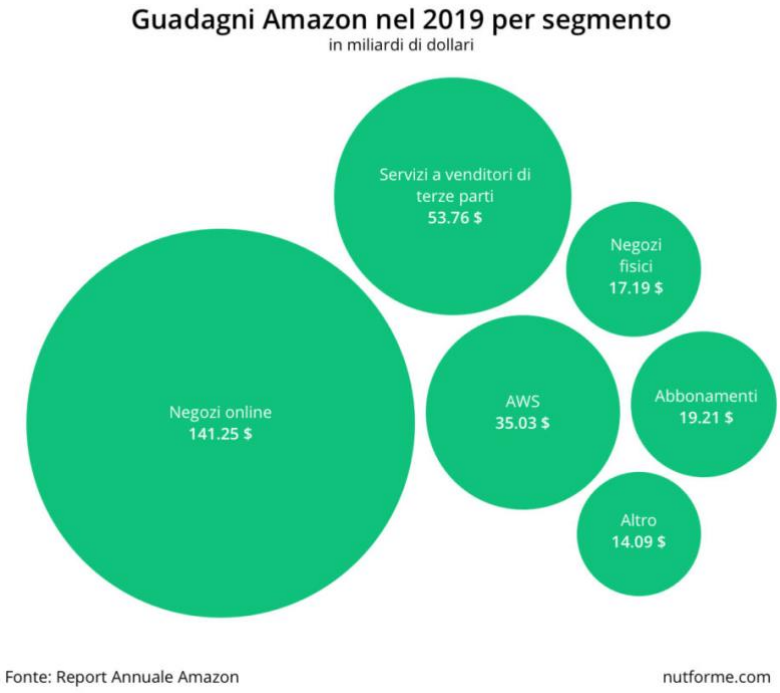

Figura 3.13 Asset aziendali Amazon

Scendendo da un piano generale, che rappresenta quanto a livello di mercato le API si siano rivelate vincenti, ad un piano più pragmatico un aspetto di interesse da osservare è legato alla documentazione delle API. In altri termini, affinché uno sviluppatore possa utilizzare le interfacce messe a disposizione, in modo pubblico o privato, è necessario che esista una

documentazione comprensibile e completa utilizzabile per capire come utilizzare tali interfacce.

Prima dell'avvento di ReST, creare una documentazione comprensibile ed utilizzabile che consentisse, a sua volta, di utilizzare delle API era un compito arduo a causa della diversità e numerosità dei linguaggi di programmazione utilizzabili o dei diversi modelli di programmazione impiegabili. Questo perché, in un contesto di forte variabilità e poliedricità la documentazione non solo doveva occuparsi di descrivere le Api e l'utilizzo ma doveva descrivere e prevedere tutti gli scenari dipendenti dai fattori esterni come, ad esempio, il linguaggio di programmazione.

Con l'avvento di ReST, utilizzato come modello di programmazione per lo sviluppo delle API, è stato possibile rendere più lineare la creazione della documentazione anche perché indipendente dal linguaggio di programmazione. I primi approcci alla documentazione delle API, descritti in precedenza, si sono basati sull'utilizzo del Web Services Description Language WSDL e la successiva evoluzione Web Application Description Language (WADL) ma questi non sono mai riusciti ad affermarsi, come linguaggio standard, in quanto considerati macchinosi e complessi a causa del formato XML utilizzato.

Sulla scia di interesse per i servizi web e di avversione per le tecnologie di scrittura della documentazione esistenti nel 2011 nasce il progetto SWAGGER il cui scopo è quello di massimizzare la generazione automatica della documentazione delle API REST e minimizzare la scrittura manuale di questa da parte del programmatore.

Grazie a questa scelta, Swagger è divenuto in breve tempo la tecnologia più diffusa per la documentazione delle interfacce di programmazione delle applicazioni ma più in particolare la tecnologia più utilizzata per le API ReST.

Nel novembre 2015, SmartBear Software, la società che ha sviluppato Swagger e che ne ha la proprietà, ha dato il suo contributo alla creazione di una nuova iniziativa: la OpenAPI Initiative. All'avvio di tale collaborazione, Swagger è stato ribattezzato con il nome di OpenAPI Specification, anche se ad oggi la tecnologia, viene ancora identificata tra gli addetti ai lavori, nella documentazione tecnica (libri e articoli sul web) e negli articoli di ricerca con il nome originario Swagger.

Utilizzare l'insieme di tecnologie e strumenti di cui consta Swagger e che sono stati realizzati in riferimento alla specifica OpenAPI sono, attualmente, il modo migliore per documentare le API REST, in quanto rendono possibile

generare documentazione che può essere letta da macchine e al contempo documentazione interattiva che può essere letta facilmente dagli umani. I principali tool, di tipo open source, di cui si compone Swagger sono:

- Swagger Editor: un editor utilizzabile per descrivere le API REST in linguaggio JSON o YAML all'interno del browser ed avere una anteprima real-time della documentazione prodotta.
- Swagger UI: una collezione di asset HTML, CSS e Javascript generata automaticamente dalla documentazione della API (che deve essere conforme allo standard OpenAPI).
- Swagger Codegen: genera automaticamente stub per la parte client e server a partire dalla specifica OpenAPI.

Arrivando al processo di produzione, la documentazione di una API può essere approcciata in due differenti modi: Code First; Design First. L'approccio Code First prevede che si effettui prima la realizzazione del codice sorgente della API e in un secondo momento a partire dal codice Swagger è in grado di ricavare direttamente la documentazione, indipendente dal linguaggio di programmazione utilizzato, che può essere interpretata sia dalle macchine che dagli umani.

L'approccio Design First prevede invece che sia generata prima la documentazione della API e in un secondo momento attraverso il generatore automatico di Swagger viene prodotto il codice della API documentata. Questa conversione automatica della documentazione in codice sorgente della API è effettuata per mezzo di generatori di codice Swagger, generatori che esistono per tutti i linguaggi di programmazione più utilizzati.

Utilizzando l'approccio Design First con Swagger di genera automaticamente il codice sorgente, e questo potrà essere integrato manuale in base alle esigenze del caso. Nel caso in cui si siano effettuate delle integrazioni manuali che generano errori in fase di compilazione Swagger notificherà tali errori utilizzando un sistema di integrazione continua.

Concludendo, Swagger è lo infrastrutturale che meglio si adatta alle esigenze del progetto i-mole e che consente di automatizzare le attività di integrazione e di iterazione a tutti i livelli sposando la logica di iterazione dalla by contract: macchina-macchina; macchina-uomo; uomo-uomo e così via.

Inoltre, l'ambiente per la produzione di codice documentato con Swagger è disponibile in modo open sul sito Swagger ufficiale (https://swagger.io/) da

dove quindi è possibile scaricare i tool decritti in precedenza: Swagger Editor; Swagger UI; Swagger Codegen.

4. COLLABORAZIONE CON LE PARTI INTERESSATE

A valle delle necessità legate alle problematiche di manutenzione per riuso del processo di sviluppo è necessario integrare il modello di processo di sviluppo e manutenzione con un approccio progettazione human-centred design. Questo garantisce nell'ambito di i-Mole la possibilità di arrivare ad una soluzione software che pur badando al riuso e quindi alla sostenibilità economica della realizzazione e della evoluzione della stessa, risulta essere attrattiva da parte dell'utilizzatore finale in quanto si avvale degli approcci che guardano allo sviluppo badando all'usabilità per gli utenti finali human-centred design.

Queste due necessità che sono state prese in considerazione per definire il processo e che sono diventate al contempo il punto di forza del modello da applicare.

- **Orientamento alla qualità del prodotto**, sostanziato dall'utilizzo in fase di validazione del codice prodotto delle verifiche dettate dalle direttive ISO/IEC 25010:2011. Più in dettaglio, questa valutazione è eseguita ogni qualvolta viene prodotta una parte di codice o viene effettuata una modifica del codice esistente, in un'ottica di continuous integration, utilizzando strumenti automatici di analisi statica del codice, ISO 25010 compliant: Fortify Static Code Analyzer, KIUWAN, CAST e SONARQUBE.
- **Orientamento all'usabilità del prodotto** sostanziato dall'utilizzo di validatori automatici che consentono di verificare la rispondenza del sistema software prodotto ai criteri di accessibilità e fruibilità definiti dalla legge n. 04/2004 e dal regolamento di attuazione n.75/2005, oltre che all'attuazione del protocollo eGLU2.1 che consente di verificare l'usabilità dei siti web e dei servizi online delle PA. Per il validare i principi del protocollo eGLUE2.1 saranno utilizzate le linee guida rilasciate dal Dipartimento della funzione pubblica (http://www.funzionepubblica.gov.it/sites/funzionepubblica.gov.it/files/documenti/GLU/Protocollo_eGLU_2_1_19082015_DEF.pdf) mentre per le verificare la rispondenza delle interfacce ai criteri di accessibilità e fruibilità definiti dalla legge n.4/2004 e dal regolamento di attuazione n.75/200, lo strumento utilizzato è il seguente (http://www.validatore.it/vamola_validator/checker/index.php).
- **Orientamento all'utente del processo di sviluppo**, questa caratteristica è ottenuta grazie alla integrazione del processo HCD con il processo

tradizionale. Difatti, l'obiettivo delle linee guida dell'HCD per gli appalti web (http://www.funzionepubblica.gov.it/sites/funzionepubblica.gov.it/files/linee_guida_appalti_hcd_beta01_0.pdf), rilasciate dalla presidenza del consiglio dei ministri ed inserito nella agenda digitale è proprio quello di eseguire la progettazione sistema software orientata all'utente.

- **Flessibilità di sviluppo**, questa caratteristica è fondamentale per il framework SCRUM, difatti è previsto il "Team Scrum" ovvero un gruppo di lavoro formato da un gruppo di persone che scelgono come meglio compiere il lavoro organizzandosi e coordinandosi al proprio interno e hanno tutte le competenze necessarie per realizzare il lavoro senza dover dipendere da nessuno al di fuori del team. Il modello di team in Scrum **è progettato per ottimizzare la flessibilità, la creatività e la produttività**. I Team Scrum rilasciano i prodotti in modo iterativo e incrementale, massimizzando le opportunità di feedback. I rilasci incrementali di prodotto "Fatto" garantiscono che una versione potenzialmente utile del prodotto funzionante sia sempre disponibile.

Scendendo su un piano più operativo, al processo SCRUM tradizionale, di cui si riporta una descrizione di seguito, vengono apportate delle migliorie grazie alla metodologia HCD la metodologia proposta prevede:

1. **Customer Committee**. Uno dei ruoli chiave nel processo è il Product Owner. Il Product Owner rappresenta tutte le parti interessate ed è la voce del cliente. Lui o lei è responsabile di assicurare i risultati del team costituiscono valore per il business. In i-mole si è pensato di introdurre il concetto di Customer Committee che include un Product Owner selezionato all'interno del team SCRUM e almeno due persone referenti del cliente, preferibilmente un utente finale e un esperto di dominio aziendale. Il Customer Committee è attivamente coinvolto nel prodotto, e nella definizione degli Sprint Backlogs e degli Sprint Review.
2. **Inception**. In processi basati su metodologia SCRUM, lo sviluppo di un software di solito inizia partendo degli input forniti dagli stakeholders. In i-mole la fase iniziale consiste nelle interviste svolte ai soggetti interessati, e nello studio del dominio applicativo. In questo modo, si uniscono le esigenze e il modus operandi solitamente adottato dalla comunità HCI, che di solito si riferiscono quasi esclusivamente a utenti finali, lead-user e clienti durante la definizione dei requisiti, con il

modus operandi degli ingegneri del software, che si riferiscono prevalentemente a studio di settore, legacy systems in uso dai competitors, studio di mercato ecc... con contatti limitati con l'utente o con i clienti finali.

3. **Sprint n.0.** In un processo di tipo SCRUM, durante questa fase viene progettata un'architettura preliminare dell'applicazione, e sono identificati sia i moduli principali che i moduli secondari. Tuttavia, lo sviluppo del software e il codice sorgente del software di solito avvengono nella fase di sviluppo durante l'esecuzione degli Sprint successivi. Il processo definito, invece, identifica in modo esplicito un "n.0 Sprint" in cui è realizzato un prototipo di alto livello. L'architettura e l'infrastruttura del sistema di base sono sviluppati con l'obiettivo di avere le fondamenta su cui poggiare facilmente i mattoni prodotti durate i successivi Sprints. L'obiettivo è quello di ottenere uno scheletro del sistema software altamente modulare con servizi software di base come i dati, meccanismi di comunicazione, reporting, servizi di calcolo, ecc.. che permettono un efficace ed efficiente sviluppo di moduli software e di funzioni durante il prossimo Sprint. Un altro obiettivo importante è quello di ridurre il più possibile il tempo necessario per avere una prima versione del software di lavoro.
4. **SCRUM Islands**. Per lo svolgimento delle attività si deve considerare in genere un Team di 4 persone o meno. Uno di loro è il Product Owner incluso nel Customer Committee che ricopre anche il ruolo di membro del team (coinvolto nella progettazione e sviluppo di software) quando non è impegnato in alcuna attività. Un altro membro del team è di solito un graphic designer con una conoscenza di base delle tecniche di HCI. Gli altri sono degli ingegneri del software. Quando un progetto inizia, i membri del team selezionati vengono assegnati ad una "SCRUM Island". Un SCRUM Island è un sito virtuale in cui tutti i partecipanti possono collaborare attivamente. Questa soluzione permette di massimizzare il flusso di informazioni, evitando i gap di gestione, e di migliorare la comunicazione continua e la collaborazione tra i membri del team. Per questo motivo il Daily Standup Meeting non è adottato. Se un membro è coinvolto in più di un progetto, lui/lei deve spostarsi tra i diversi ambienti collaborativi ovvero le isole. Ovviamente l'organismo di gestione superiore dell'intera linea adi prodotto deve cercare di minimizzare la condivisione del personale tra i singoli prodotti.

5. **1 Week Time Boxed Sprint**. In i-mole si adottano sprint con una durata fissa di 1 settimana. Ogni settimana gli output ottenuti da ogni sprint sono rivisti nei successivi sprint, mentre gli arretrati sono ridefiniti includendo i nuovi compiti da eseguire insieme a quelli che sono ancora aperti e non chiusi. Così la durata dello sprint non è il risultato di una stima ex-ante ma è predefinito. Solo le stime riguardanti l'intero progetto sono effettuate dal Customer Committee alla fine di ogni sprint, durante l'affinamento del Product Backlog. Ciò è possibile grazie al coinvolgimento attivo dell'eventuale cliente nel processo.
6. **(IN)Sprint Review**. Il processo include una continua verifica e validazione (V&V) delle attività. Questa attività si estende temporalmente lungo tutto lo sprint. Quando una prima bozza di lavoro del software è disponibile (dopo lo Sprint n.0 o qualche sprint più tardi), gli utenti finali del Customer Committee sono invitati a utilizzare il sistema e fare una sorta di prova utente o di test del sistema. Questo permette di valutare le funzionalità sviluppate durante uno sprint e verificare la loro corretta implementazione, e l'integrazione con ciò che già esiste. I risultati delle attività di V&V sono poi discussi e analizzati all'interno della Sprint Review alla fine degli Sprint. Qui il Customer Committee è coinvolto nell'analisi dei risultati, sia dal punto funzionale che dal punto di vista tecnico, passando attraverso i problemi operativi e quelli di business.
7. **Project Retrospective**. Un processo di SCRUM in genere include la Sprint Retrospective che è un evento ufficiale nella metodologia SCRUM in cui tutte le parti (Product Owner, SCRUM Master, Team) coinvolte nello sviluppo cercano di migliorare il processo evidenziando i punti di forza e di debolezza. Nel processo di i-mole non avviene questo. SCRUM significa auto-organizzante, ovvero team inter-funzionale in grado di organizzare il lavoro in modo autonomo. Grazie all'adozione del Customer Committee, (IN) Sprint Review e SCRUM Island c'è un ottimo flusso di informazioni, un buon livello di comunicazione, un buon coordinamento e una valutazione tempestiva da ciascun membro del team e dei clienti. Così la squadra può condividere, analizzare e migliorare il processo continuamente, senza la necessità di un evento formale Sprint Retrospective. Quello che i-mole fa è una retrospettiva di progetto. Qui tutti i dati di progetto disponibili vengono analizzati, i punti di forza e di debolezza sono evidenziati e le opportunità e le iniziative di miglioramento sono definiti ed eseguiti. In tal modo, ci si riferisce alle ben note buone pratiche e approcci della comunità di

Software Engineering ispirato al Learning Organization and Experience Factory Models. Questo permette di avere un ciclo di feedback all'interno dell'intera organizzazione che, consente di stimolare la crescita della società. Un altro punto chiave di questo evento è il coinvolgimento attivo dei clienti nell'analisi retrospettiva per mezzo di focus group tematici. Durante il focus group i clienti sono informalmente intervistati al fine di esplorare le opportunità di miglioramento della qualità del prodotto, tutto questo al fine di soddisfare il cliente stesso. I feedback raccolti vengono poi utilizzati a livello globale per migliorare le proprie strategie, gestione, e organizzazione del proprio processo.

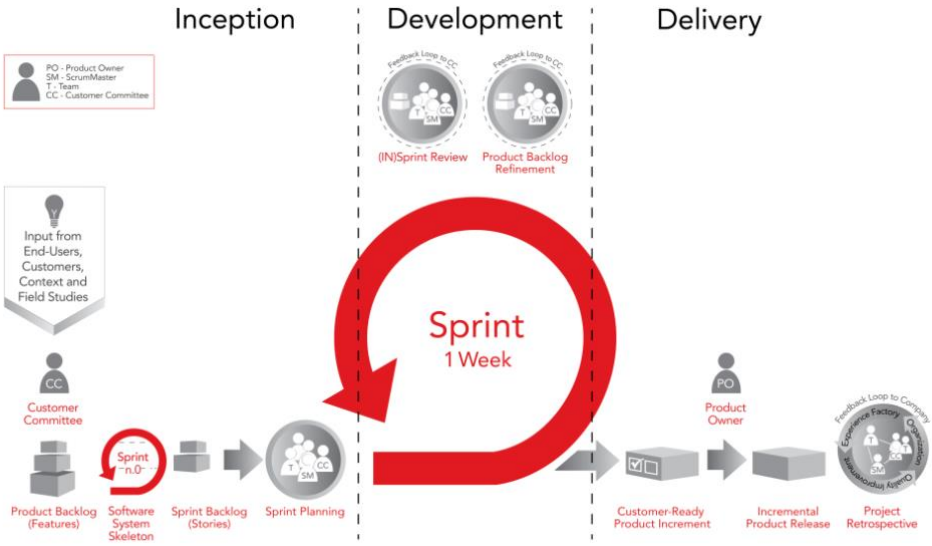

Tabella 4.1 SCRUM Process tailored

5. CICLO DI VITA DEL SOFTWARE

5.1 Modelli del Ciclo di vita del Software in base all'approccio del testing

In base alla tipologia del processo scelto, di conseguenza, saranno pianificate ed eseguite tutte le attività stabilite.

Di seguito si riportano i modelli del ciclo di vita del software che possono essere raggruppati in cinque modelli di seguito elencati e spiegati. 0**Error! Reference source not found.**

5.1.1 Modelli sequenziali

Questo tipo di modelli ha un flusso di attività lineari e sequenziali e ogni fase viene eseguita solo se i risultati della fase precedente sono stati raggiunti e completati.

Di seguito vengono descritti due tipi di modelli sequenziali.

1. Modello waterfall (a cascata)
2. V-Model

1) Modello waterfall

Il modello a cascata, inventato nei tardi anni '50 e divenuto popolare all'inizio degli anni '70, ha le seguenti caratteristiche. **Error! Reference source not found.**

- Esiste un unico rilascio nel software, disponibile alla fine del ciclo di vita.
- È un modello rigido e sequenziale in cui non esistono sovrapposizioni tra le fasi.
- Ogni fase è definita e stabile, soprattutto la fase iniziale ovvero quella dei requisiti.
- Se l'esito della fase di testing è negativo significa che ci sono discrepanze tra il "ciò che il cliente voleva" e "ciò che è stato fatto", ed in questo caso bisognerà necessariamente tornare nella fase precedente o addirittura all'inizio del ciclo di vita procurando, oltre a ritardi temporali, danni economici di notevole importanza.

Di seguito,nella figura 5.1, una rappresentazione grafica del modello sequenziale waterfall.

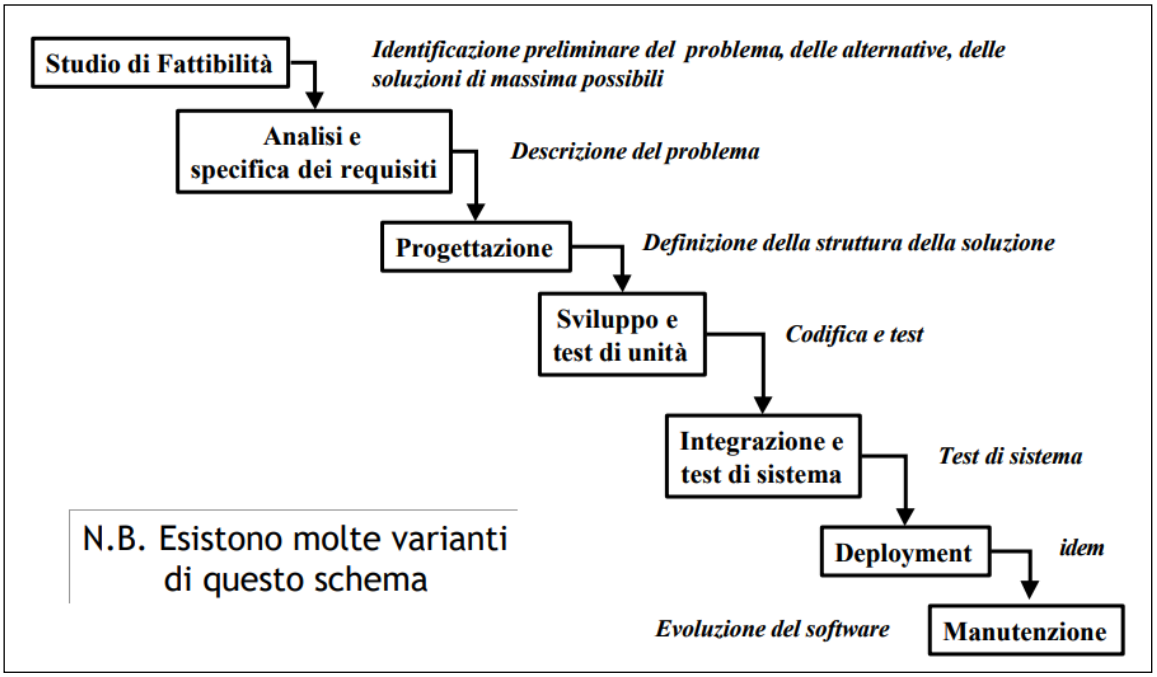

Figura 5.1 Modello Waterfall (a cascata)

2) Modello V-Model

Il modello V-Model, originariamente diffuso come standard dal Ministero della difesa tedesco nel 1992, è una variante del modello a cascata dove ogni livello di sviluppo è associato alla corrispondente fase di test; il processo di testing è integrato nel processo di sviluppo adottando, quindi, il principio del Testing anticipato.

L'esecuzione dei singoli livelli di test, associati alla relativa fase di sviluppo, è sequenziale.

Inoltre, descrive il processo di sviluppo dal punto di vista funzionale definendo un insieme di attività e risultati che devono essere prodotti; questo tipo di modello è adottato così da molte compagnie contestualizzandolo in vari contesti specifici di applicazione.

Il V-Model, dunque, evidenzia come i problemi scoperti sul lato destro della V determinano la riesecuzione di attività sul lato sinistro.

Di seguito,nella figura 5.2, una rappresentazione grafica del V-Model.

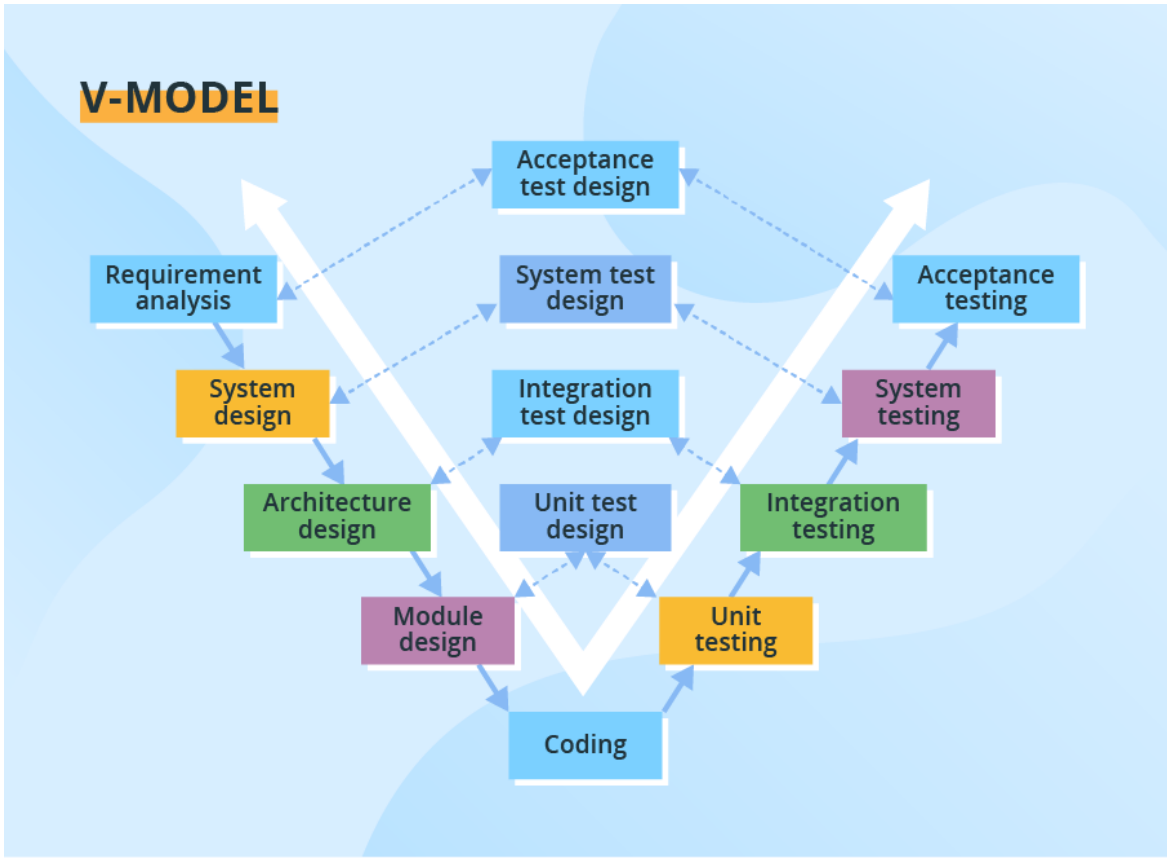

Figura 5.2 V-Model

Per ovviare alle limitazioni del modello a cascata, sono stati adottati nuovi e diversi approcci ossia il modello incrementale e il modello iterativo.

La differenza tra i due modelli viene illustrata qui di seguito. **Error! Reference source not found.**

5.1.2 Modello incrementale

Nel modello incrementale le attività progettuali da eseguire (analisi, progettazione, sviluppo e test) vengono eseguiti a "pezzi" creando dei sotto-progetti con una crescita incrementale delle funzionalità software.

Solitamente vengono implementate prima i requisiti considerati di maggior valore per gli stakeholder cosicché ogni versione aggiunge nuove funzionalità.

Si illustra una figura che ritrae il modello incrementale.

Figura 5.3 Modello incrementale

5.1.3 Modello iterativo

Nel modello iterativo, invece, vengono realizzati una serie di prototipi e di attività da eseguire (analisi, progettazione, sviluppo e test), via via più vicini al sistema finale.

La prima iterazione si concentra sul sistema creato in modo primitivo e semplificato, le successive iterazioni rilasciano funzionalità che sono un sottoinsieme crescente delle funzionalità totali.

Fondamentale è il feedback anticipato dagli stakeholder per verificarne la correttezza e l'adeguatezza.

In questo tipo di modello, dunque, già dal primo prototipo sono presenti le funzionalità di base che vengono progressivamente raffinate e migliorate.

Di seguito si illustra una figura che rappresenta il modello iterativo.

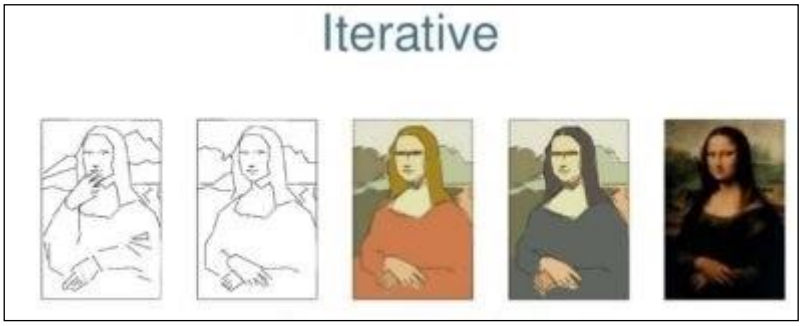

Figura 5.4 Modello iterativo

Di seguito vengono illustrati due modelli, entrambi presentano contemporaneamente attività svolte in modo incrementale ed iterativo. **Error! Reference source not found.**

5.1.4 Modello a spirale (o per prototipi)

Il modello a spirale è rappresentato mediante una spirale in cui ogni voluta rappresenta un raffinamento/completamento del sistema rispetto alla voluta precedente.

Essendo di natura iterativa/incrementale, il sistema è sviluppato in modo ciclico (iterativo ed incrementale) producendo versioni sempre più raffinate del sistema finale.

Il percorso va in senso orario partendo dal centro dell'origine della spirale ed ogni iterazione prevede la ripetizione di tutte le fasi del processo di sviluppo.

Di seguito,figura 5.5, si illustra una figura che rappresenta il modello a spirale.

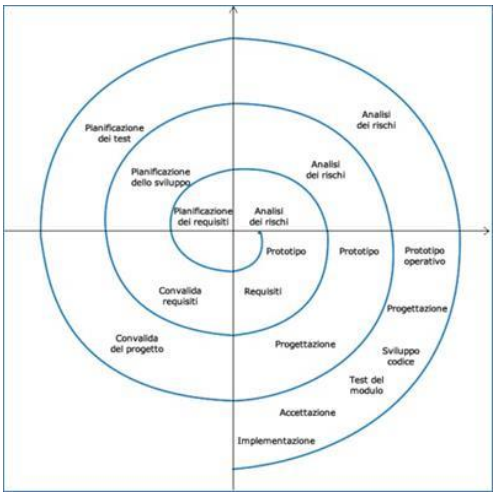

Figura 5.5 Modello a spirale

5.1.5 Modello RUP (Rational Unified Process)

Il modello RUP, facente parte dei modelli di tipo iterativo/incrementale, è la sintesi metodologica dei metodi che hanno portato alla definizione dell'UML (Unified Modeling Language).

Lo Unified Process è composto dalle seguenti quattro fasi.

1. Ideazione
    a. definisce l'ambito, la portata e gli obiettivi del progetto;
    b. stabilisce il budget, i vincoli e le tempistiche.
2. Elaborazione: specifica le caratteristiche e baseline dell'architettura
    a. definisce e crea una baseline dell'architettura eseguibile;

   b. dimostra che l'architettura scelta supporterà la visione di progetto con costi e tempi ragionevoli.
3. Costruzione: realizza il prodotto
   a. gestisce e monitora le risorse, i tempi e i costi di sviluppo;
   b. completa le versioni eseguibili del prodotto software raggiungendo una qualità adeguata in tempi brevi;
4. Transizione: esegue le attività necessarie per consegnare il prodotto agli utenti finali
   a. prepara la messa in esercizio del sistema;
   b. verifica che tutti gli utenti finali siano in grado di comprendere e di utilizzare il sistema.

Ogni iterazione è un mini progetto di durata fissa e costante (time-boxed) che può variare da 1 a 3 mesi; le prime iterazioni, invece, possiedono pochi requisiti da implementare e servono per calibrare la velocità dell'iterazione.

Si specifica che ogni iterazione è tendenzialmente lunga dato che è considerata un sottoinsieme crescente dell'insieme complessivo delle funzionalità.

Terminata un'iterazione, vengono stabilite delle milestones dove si analizzano i feedback ricevuti ovvero risultati ottenuti, le richieste di cambiamento e si aggiorna il piano di progetto definendo il piano delle attività dell'iterazione successiva.

Qui sotto si rappresenta, mediante figura, il modello Rational Unified Process.

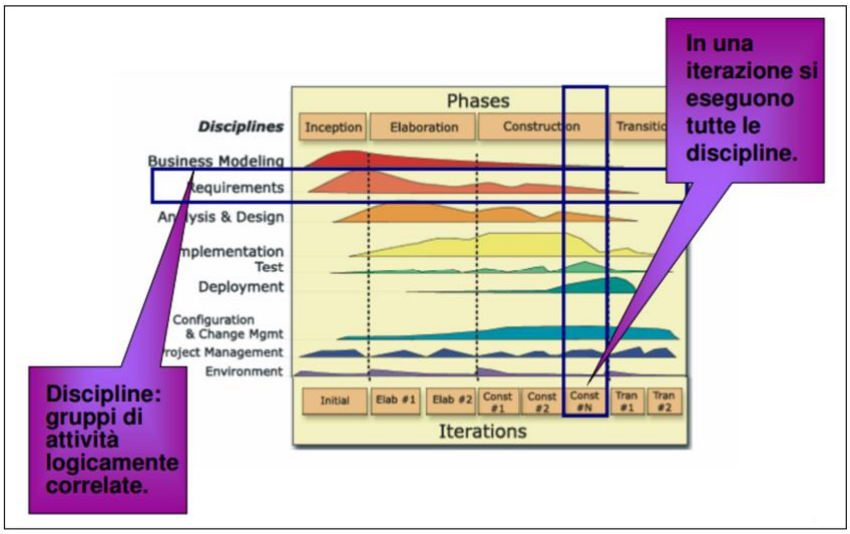

Figura 5.6 Modello RUP

5.1.6 Modelli con approccio Agile

Il modello con approccio Agile venne ideato nel febbraio 2001 durante un incontro presso la località sciistica "The Lodge at Snowbard", nelle montagne Wasatch dello Utah (stato federale degli Stati Uniti d'America – USA); più precisamente, diciassette persone si riunirono per discutere, condividere e parlare delle necessità di trovare un'alternativa ai processi di sviluppo software pesanti basati su documentazione e di scoprire modi migliori per creare software, sviluppandolo e aiutando gli altri a fare lo stesso.

Dopo aver documentato e racchiuso il tutto, all'interno del "Manifesto Agile Sviluppo Software" firmato da tutti i partecipanti, sono emersi i seguenti punti cardine. **Error! Reference source not found.**

- Gli individui e le interazioni più che i processi e gli strumenti;
- il software funzionante più che la documentazione esaustiva;
- la collaborazione col cliente più che la negoziazione dei contratti;
- rispondere al cambiamento più che seguire un piano.

Contestualmente, sono stati fissati i 12 principi su cui verte il mondo Agile di seguito elencati. **Error! Reference source not found.**

1. La nostra massima priorità è soddisfare il cliente rilasciando software di valore, fin da subito e in maniera continua;
2. accogliamo i cambiamenti nei requisiti, anche a stadi avanzati dello sviluppo. I processi agili sfruttano il cambiamento a favore del vantaggio competitivo del cliente;
3. consegniamo frequentemente software funzionante, con cadenza variabile da un paio di settimane a un paio di mesi, preferendo i periodi brevi;
4. committenti e sviluppatori devono lavorare insieme quotidianamente per tutta la durata del progetto;
5. fondiamo i progetti su individui motivati. Diamo loro l'ambiente e il supporto di cui hanno bisogno e confidiamo nella loro capacità di portare il lavoro a termine;
6. una conversazione faccia a faccia è il modo più efficiente e più efficace per comunicare con il team ed all'interno del team;
7. il software funzionante è il principale metro di misura di progresso;
8. i processi agili promuovono uno sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere indefinitamente un ritmo costante;

9. la continua attenzione all'eccellenza tecnica e alla buona progettazione esaltandone l'agilità;
10. la semplicità, ovvero l'arte di massimizzare la quantità di lavoro non svolto, è essenziale;
11. le architetture, i requisiti e la progettazione migliori emergono dai gruppi di lavoro che si auto-organizzano;
12. a intervalli regolari il team riflette su come diventare più efficace, dopodiché regola e adatta il proprio comportamento di conseguenza.

In estrema sintesi l'approccio Agile identifica. 0

"*È un insieme di metodi di sviluppo del software che si contrappone al tradizionale, perché si concentra sull'obiettivo da consegnare al cliente che chiede sempre più le seguenti particolarità.*

- *Tempi più brevi;*
- *alta frequenza dei rilasci;*
- *codice corretto e di qualità;*
- *attenzione al "Time To Market" ovvero il tempo che intercorre tra l'ideazione e progettazione di un prodotto e il suo effettivo ingresso sul mercato*".

Di seguito una figura che mostra, dal punto di vista concettuale, la differenza tra i modelli "tradizionali" e la filosofia Agile.

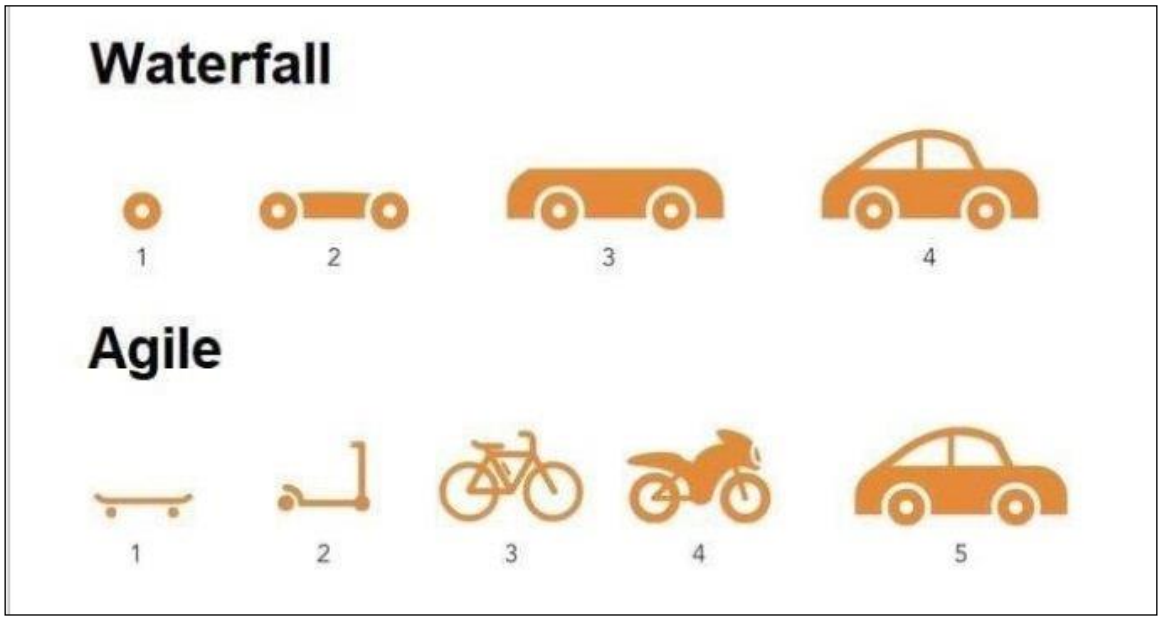

Figura 1 Differenza concettuale tra modello Waterfall e modello Agile

Oggigiorno, la metodologia Agile, considerata da molti come una filosofia, è sicuramente una delle più utilizzate all'interno del mondo IT poiché rappresenta la sfida di "produrre software a costi inferiori ed in tempi brevi".

A differenza delle altre metodologie precedentemente illustrate, si notano le seguenti differenze tra il modello Agile e i restanti modelli raccolte in forma tabellare. 0

Modelli sequenziali/iterativi	Modello Agile
Analizzano il progetto nella sua totalità per prevedere funzionalità future compiendo un grande sforzo nelle fasi di analisi	Considerano e completano velocemente piccole parti del progetto, ottenendo subito risultati tangibili, per fare in modo di terminare il progetto attraverso iterazioni successive
Si adotta un approccio predittivo ovvero la previsione di funzionalità necessarie	Si adotta un approccio adattativo ovvero l'adattamento in futuro del progetto in base alle necessità dell'utente
Utente coinvolto in fase di analisi, di collaudo per l'accettazione finale o in milestones specifiche	Utente coinvolto in modo costante e parte integrante del team di progetto
Requisiti stabiliti in anticipo e non mutabili	Requisiti modificabili frequentemente durante il progetto secondo le esigenze del cliente

Le modifiche sono vincolate o integrate ad intervalli periodici	Le modifiche sono integrate in tempo reale durante la consegna
Feedback utente alla fine del progetto	Feedback utenti rapidi e presenti durante ogni fase del progetto
Seguono un piano	Rispondono al cambiamento
Budget allocato all'inizio	Budget allocato in modo ciclico

Tabella 5.1 Differenza tra modelli sequenziali/iterativi e modelli agili

Dopo aver illustrato in modo teorico i concetti della filosofia Agile, si approfondisce l'argomento entrando nei dettagli delle attività quotidiane che costituiscono questa importante filosofia.

Altre metodologie e approcci di tipo Agile

Facendo riferimento alla filosofia Agile, all'interno di questa, possiamo trovare varie metodologie e approcci che sono i seguenti. **Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.**

- Scrum
- Test Driven Development (TDD)
- Acceptance Test Driven Development (ATDD)
- Behavior Driven Development (BDD)
- DevOps

5.1.6.1 Scrum

La metodologia Scrum è un modello di processo per produrre software ottenendo il massimo valore utile nel minor tempo.

Dà la possibilità al cliente di ispezionare in modo rapido e ciclico, ogni 3-4 settimane, versioni funzionanti del software.

Il cliente definisce le funzioni da realizzare e le relative priorità; il team di sviluppo decide quotidianamente il modo migliore per produrre le funzionalità con la più alta priorità.

Ogni 3-4 settimane nasce una nuova versione che viene esaminata per decidere se continuarne lo sviluppo con un altro sprint o produrne un rilascio.

Le caratteristiche del processo Scrum possono essere raggruppate in categorie di seguito elencate e illustrate mediante tabella.

I. Ruoli (product owner, scrum master e team);
II. rituali (sprint, sprint planning, daily scrum meeting, sprint review e sprint retrospective);
III. prodotti di lavoro (Product backlog, sprint backlog, Tasks e Scrumboard).

Ruolo	Attività
Product Owner	È la figura che rappresenta gli stakeholder; responsabile del valore del prodotto; definisce le caratteristiche del prodotto; scrive il product backlog sotto forma di user stories; decide le date e i contenuti del rilascio.
Scrum Master	È il responsabile dell'aderenza alle pratiche Scrum: figura di coach e mentore; assicura il benessere del team; supporta la cooperazione di ruoli e funzioni.
Team	È un team cross-funzionale auto-organizzato composto mediamente da 5-9 persone che coprono vari ruoli. Developer, tester, UX/UI Design, DBA ecc. ecc.); i membri del team possono avere qualche skill personale ma l'intero team è responsabile dello sviluppo.

Tabella 5.1 Ruoli del processo Scrum

All'interno del processo Scrum, ci sono i seguenti rituali che sono descritti in forma tabellare.

Rituale	Definizione
Sprint	È un'iterazione di durata fissa (time-boxed) e l'intero processo Scrum è composto da una serie di sprint. Ogni sprint incluse design, codifica e test. Durata: 2-4 settimane max
Sprint Planning	Pianificazione di gruppo dove si stimano le attività che saranno svolte in base alla velocità con la quale devono essere svolte. Vengono definiti i criteri di uscita (DoD) per tutti gli elementi dello sprint backlog. Durata: 1 ora a settimana
Daily Scrum Meeting (o stand-up)	Riunione quotidiana, in piedi, dove ogni membro del Team comunica cosa ha fatto, cosa pianifica per oggi e che tipo di impedimenti ha riscontrato. Durata: 15 minuti max, in piedi
Sprint Review	Il team presenta ciò che ha ottenuto con lo sprint riguardante il prodotto: cosa è stato o non è stato completato in questo sprint. In maniera informale, viene presentata una demo a tutto il team Durata: 4 ore max
Sprint Retrospective	Riguarda il processo: cosa è andato bene e quali impedimenti sono stati rilevati durante lo sprint Potenzialmente partecipano tutti gli stakeholder (anche committenti e utenti) per analizzare le informazioni con lo scopo di migliorare la maturità del processo (lessons learned). Durata: 3 ore max.

Tabella 5.2 Rituali svolti durante il processo Scrum

All'interno del processo Scrum vengono creati i seguenti manufatti descritti mediante tabella.

Prodotto di lavoro	Scopo
Product Backlog	Il product backlog è un set di requisiti in forma di user stories, ordinate per priorità e valore di business; vengono redatti dal Product Owner.
Sprint Backlog	È un set ordinato di attività che il team seleziona durante lo sprint planning e che si impegna di portare al termine alla fine dello sprint; lo sprint backlog, dunque, è un sottoinsieme del product backlog da implementare durante lo sprint.
Tasks	Attività atomiche richieste su base "volontaria", ovvero i membri del team di lavoro selezionano il lavoro da fare su scelta personale.
Scrumboard	Rappresentazione grafica che tiene traccia delle attività che devono essere svolte, che sono in svolgimento e che sono state svolte tenendo conto delle priorità. Utile per monitorare l'andamento delle attività in modo grafico e per identificare potenziali colli di bottiglia.

Tabella 5.3 Manufatti prodotti nel processo Scrum

Si illustrano, tramite figura, i prodotti di lavoro creati durante il processo Scrum per poter avere in modo chiaro le tipologie e le relative sottotipologie.

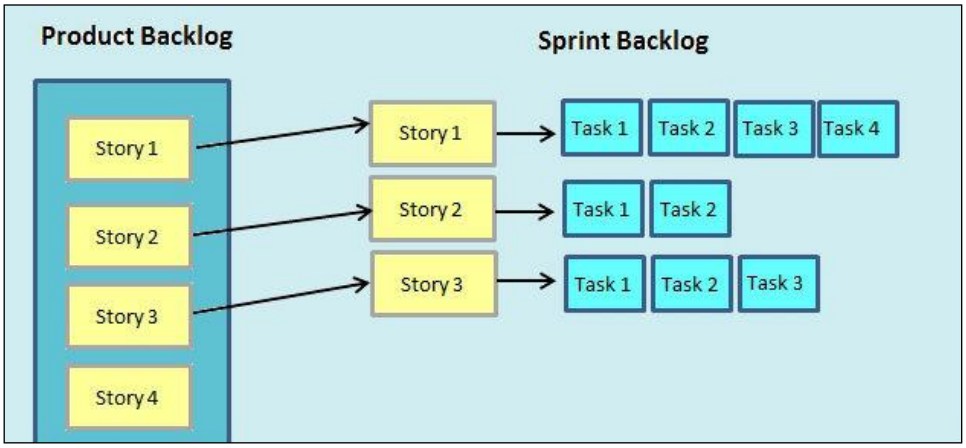

Figura 2 Product Backlog, Sprint Backlog, Tasks

Nella seguente figura viene rappresentato il processo Scrum.

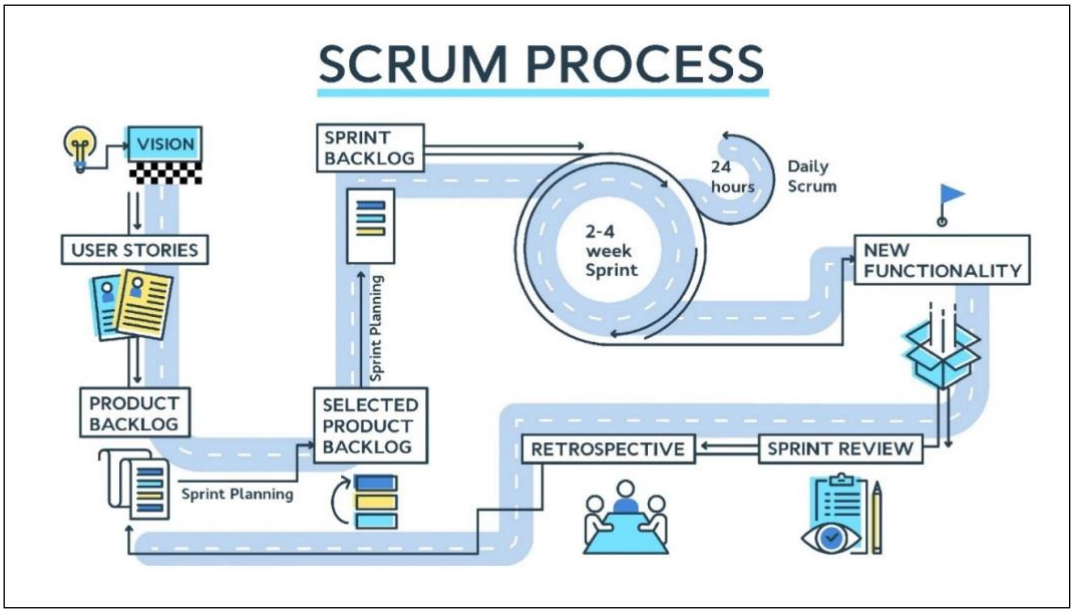

Figura 3 Processo Scrum

5.1.6.2 DevOps

Con questo termine si fa riferimento ad una metodologia di sviluppo software che enfatizza al massimo la collaborazione, la comunicazione e l'integrazione tra sviluppatori software e professionisti IT al fine di garantire il successo dell'intera organizzazione.

La metodologia DevOps, letteralmente "Development Operation" che significa rispettivamente "Sviluppo" e "Messa in produzione", promuove un set di processi ed accorgimenti atti a favorire e a migliorare la comunicazione e

collaborazione tra i reparti fra gli sviluppatori e la parte IT Operation (in questo caso coloro che si occupano della messa in produzione/ rilascio); processi ed accorgimenti dipendono dall'organizzazione e dagli obiettivi che si intendono perseguire.

L'obiettivo principale di DevOps è creare un'organizzazione in grado di fornire prodotti, processi e servizi software, facilitandone la distribuzione continua in tempi rapidi.

I principali processi di sviluppo software continui sono di seguito elencati.

a) Continuous Integration, ovvero la strategia di sviluppo software che minimizza durata ed effort di ogni integrazione del codice generando, anche in modo quotidiano, una maggiore efficienza grazie a test automatizzati idealmente applicabili a tutti i livelli di test.
b) Continuous Delivery, letteralmente "Consegna continua" è l'intero processo che indica l'esecuzione contemporanea ed automatizzata delle fasi di sviluppo, rilascio, feedback e gestione della qualità in brevi intervalli ciclici.
c) Continuous Deployment, è l'insieme dei concetti di Continuous Integration – CI e Continuous Delivery – CD dove ogni modifica viene rilasciata in modo completamente automatizzato in ambiente di produzione;
d) Microservices, ovvero piccoli servizi autonomi e modulari che rappresentano determinate funzionalità rilasciati in modo autonomo.

DevOps è stato ben accolto all'interno del mondo ingegneristico del software e ha ricevuto particolari attenzioni per quanto concerne la letteratura.

I report annuali "State of DevOps" mostrano che il numero di team DevOps è aumentato dal 16% nel 2014 al 19% nel 2015, successivamente al 22% nel 2016 fino ad arrivare al 27% nel 2017. **Error! Reference source not found.**

Un esempio pratico di questa metodologia potrebbe essere una collaborazione stretta tra:

- sviluppatori e Tester al fine di predisporre una suite test composta da test automatici e manuali a garanzia e verifica di quanto sviluppato;
- sviluppatori e sistemisti al fine di predisporre uno script automatico che permetta il rilascio in vari ambienti del software sviluppato.

Di seguito una figura che illustra il principio DevOps.

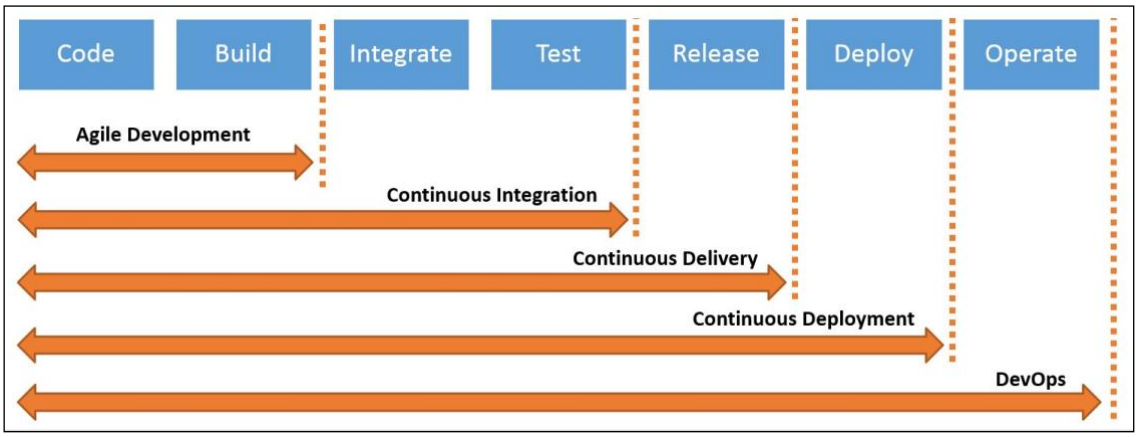

Figura 4 Pratica DevOps

5.1.6.3 Test Driven Development (TDD)

Il Test Driven Development è una pratica agile di sviluppo software che è stata utilizzata sporadicamente per decenni; i primi riferimenti del suo utilizzo sono del Progetto Mercury NASA all'inizio degli anni '60.

Gli obiettivi dei TDD sono:

- Prevenzione dei malfunzionamenti;
- l'eliminazione dei malfunzionamenti;
- la tolleranza agli errori.

Con il TDD, prima di implementare il codice, lo sviluppatore o chi preposto scrive test di unità automatici per le nuove funzionalità che sono in procinto di essere implementate.

Dopo aver scritto i test case, lo sviluppatore implementa il minimo pezzo di codice necessario per superare i test cases progettati.

Lo sviluppatore scrive qualche test cases, implementa il relativo codice, implementa il successo codice e così via.

Considerati i requisiti funzionali, gli ingegneri del software TDD implementano il codice attraverso rapide iterazioni nel seguente modo.

- Scrittura di un piccolo numero di casi di unit test automatizzati;
- esecuzione dei nuovi casi di unit test per assicurarsi che falliscono poiché non c'è ancora codice da eseguire;
- implementazione del codice che dovrebbe consentire il superamento dei nuovi casi di unit test;

- riesecuzione dei nuovi casi di unit test per assicurarsi che ora passino con il codice implementato;
- periodicamente (e.g. una volta al giorno, in funzione della grandezza della base di codice) rieseguire tutti i casi di test nella base di codice per garantire che il nuovo codice non causi imprevisti/interruzioni di alcun caso di test eseguito in precedenza (test di regressione).

Di seguito una rappresentazione grafica del processo TDD facendo riferimento alla cardinalità dei suddetti punti descritti.

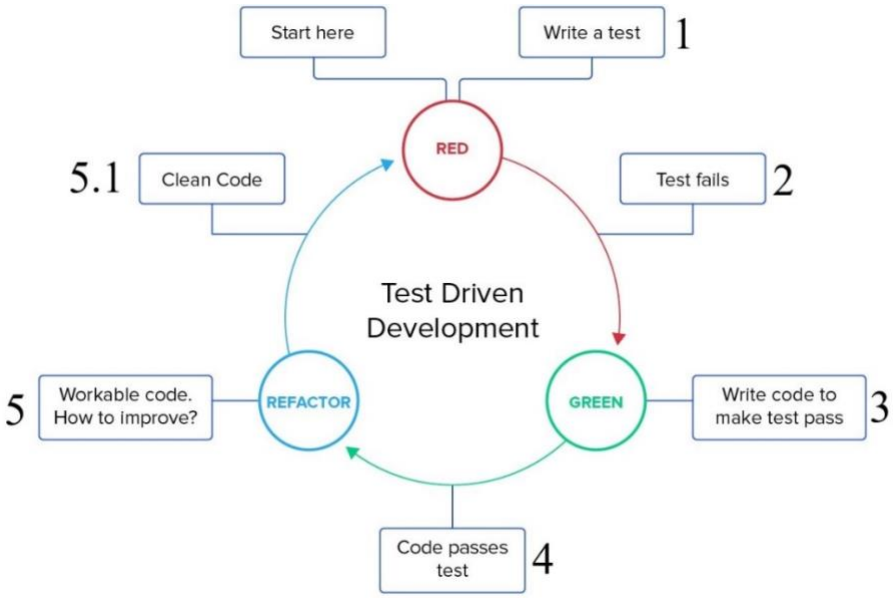

Figura 5.11 Test Driven Development

5.1.6.4 Acceptance Test Driven Development (ATDD)

La ATDD è una metodologia di sviluppo collaborativa basata sulla comunicazione fra i clienti, gli sviluppatori e i tester utilizzando uno pseudo-linguaggio di dominio dei clienti per comprenderne i requisiti.

Lo scopo degli ATDD, visto da differenti prospettive (cliente, sviluppatori, tester) che collaborano per comprendere come deve essere implementato il codice, è regolato da cinque steps di seguito illustrati.

- Selezionare la User Story;
- scrivere il test di accettazione;
- implementare la User Story;
- eseguire il test di accettazione;

- refactor.

I vantaggi dell'ATDD sono i seguenti.

- Perfezionamento della comunicazione e della collaborazione tra gli stakeholder;
- condivisione della comprensione del significato corretto e preciso dell'implementazione;
- miglioramento della copertura delle aspettative di business;
- feedback rapidi.

Gli svantaggi dell'ATDD, invece, sono i seguenti.

- Nuove metodologie che richiedono rigore e disciplina;
- trovare il giusto equilibrio tra persone/processo/strumento.

5.1.6.5 Behavior Driven Development (BDD)

BDD, inizialmente proposto da Dan North, è una sintesi e raffinamento delle pratiche di Ingegneria del Software che aiutano i team a generare e rilasciare codice di elevata qualità in tempi rapidi.

Questi test possono essere scritti dai tester o dai technical managers e servono per una piena e totale comprensione del sistema e dei comportamenti dei moduli ma dal punto di vista del business e dell'utente.

Lo scopo è sull'aspetto comportamentale del sistema, considerando gli scenari invece dei test.

Il processo BDD è simile al processo TDD e segue i seguenti steps.

- Scrive uno scenario;
- esegue lo scenario che fallisce;
- scrive il test che corrisponde alle specifiche dello scenario;
- scrive il minimo pezzo di codice necessario per far passare il test e infine lo scenario;
- ottimizza il codice (refactor) per eliminare duplicati.

I vantaggi del BDD sono i seguenti.

- Perfezionamento della comunicazione fra sviluppatori, tester e product owners;
- comprensione da parte di un pubblico sempre più ampio essendo un test di natura non tecnico;

- rapidità di comprensione e apprendimento dato che i BDD sono spiegati mediante un linguaggio semplice e comprensibile da tutti gli stakeholders.

Gli svantaggi del BDD, invece, sono i seguenti.

- Per lavorare BDD è richiesta un'esperienza pregressa con i TDD;
- BDD non sono compatibili con l'approccio sequenziale waterfall;
- se i requisiti non sono opportunamente specificati, BDD possono non essere efficaci.

Di seguito si illustra una figura che rappresenta Behavior Driven Development.

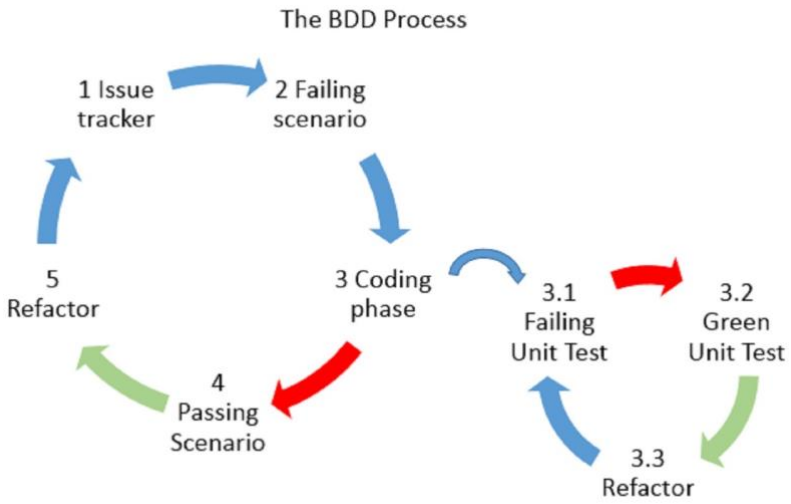

Figura 5 Behavior Driven Development – BDD

5.1.7 Differenze tra TDD, ATDD e BDD

Dopo aver descritto le tre tipologie di Test, di seguito vengono illustrate le più evidenti differenze. **Error! Reference source not found.**

Test Driven Development (TDD)	Acceptance Test Driven Development (ATDD)	Behavior Driven Development (BDD)
Focalizzato sugli aspetti implementativi e tecnici del sistema	Focalizzati sulle prospettive diverse tra gli stakeholder (utenti, sviluppatori e tester) e sulla qualità esterna del sistema	Focalizzati sull'aspetto comportamentale del sistema
Unit testing	User Acceptance Test (UAT)	Requisiti
Tendono verso un approccio tecnico	Tendono verso un approccio tecnico come i TDD	Tendono verso un approccio dal punto di vista del business/cliente

Tabella 5.4 differenze tra TDD, ATDD e BDD

Il seguente documento ha il seguente scopo:

- formalizzare il processo di sviluppo utilizzato in i-mole per una più chiara comprensione delle attività coinvolte in tale processo

A tale scopo il documento è suddiviso in 2 sezioni:

la sezione 1 presenta una visione generale del processo attraverso diagrammi di processo

la sezione 2 completa i diagrammi con una descrizione testuale e puntuale di tutte le attività del processo.

6. RIUSO NELLA MANUTENZIONE

Se si considera che il software dovrebbe essere sviluppato con l'obiettivo di massimizzare il riutilizzo dell'esperienza precedente sotto forma di conoscenza, processi, prodotti e strumenti, allora il processo di manutenzione è logicamente ideale per un processo di sviluppo del software orientato al riutilizzo. Esiste una varietà di modelli di riutilizzo.

La natura del software è che può essere modificato senza l'uso di strumenti fisici come cacciaviti e saldatori. Questo ha portato al falso presupposto che la manutenzione sia facile e poco costosa. Chiaramente niente potrebbe essere più lontano dalla verità.

La maggior parte dei sistemi software sono complessi e la modifica richiede una profonda comprensione dei requisiti funzionali e non funzionali, la mappatura delle funzioni ai componenti del sistema, e l'interazione dei componenti. Senza una buona documentazione dei requisiti, del design e del codice per quanto riguarda la funzione, la tracciabilità e la struttura, la manutenzione diventa un compito difficile, costoso e soggetto a errori. Già nel 1976, Belady e Lehman riferirono dei problemi con l'evoluzione dell'OS 360 [Belady]. La letteratura è piena di esperienze simili e lezioni apprese [Grady,Lientz,Rombach,Tracz].

6.1 Manutenzione

La manutenzione consiste in diversi tipi di attività: la correzione dei difetti esistenti nel sistema, l'adattamento del sistema a un ambiente operativo che cambia, per esempio, nuovi terminali, modifiche al sistema operativo, ecc. Il nuovo sistema è come il vecchio sistema ma diverso in un insieme specifico di caratteristiche. Si può vedere la nuova versione del sistema come una modifica *del* vecchio sistema o un nuovo sistema che riutilizza molti dei componenti del vecchio sistema. Anche se questi due punti di vista hanno molti aspetti in comune, sono abbastanza diversi per quanto riguarda i modelli di processo usati e i loro effetti sugli ambienti futuri.

Infatti, si può identificare almeno tre modelli di processo associati alla manutenzione a seconda delle caratteristiche della modifica:

- il modello quick fix
- il modello di miglioramento iterativo
- il modello di full reuse

Tutti e tre i modelli riutilizzano il vecchio sistema e quindi sono orientati al riutilizzo. Quale modello dovrebbe essere scelto per ogni particolare modifica è una combinazione di decisioni tecniche e di gestione.

Modello quick fix. Il modello quick fix implica prendere il sistema esistente, di solito solo il codice, e fare i necessari cambiamenti al codice sorgente e alla documentazione che lo accompagna, ad esempio i requisiti, il design, e ricompilare il sistema come una nuova versione. Questo può essere semplice come un cambiamento a qualche componente interno, per esempio una correzione di errore che coinvolge un singolo componente o un cambiamento strutturale o anche qualche miglioramento funzionale Qui il riutilizzo è implicito.

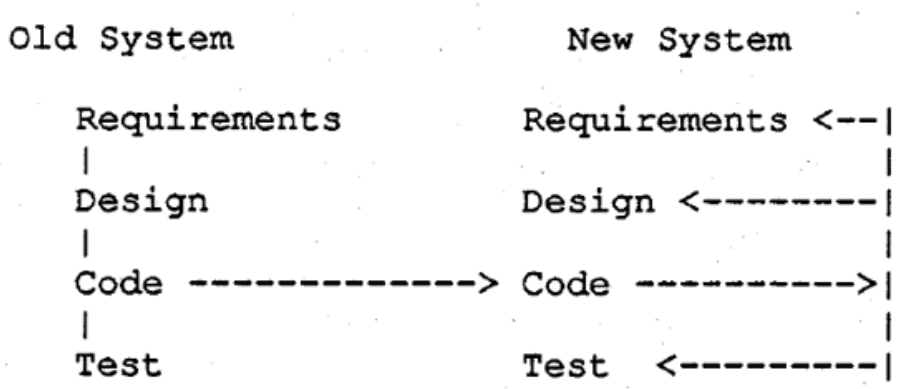

Figura 6.1 Modello quick fix

Modello di miglioramento iterativo. Il modello di miglioramento iterativo [Basili75] è un modello evolutivo che è stato proposto per lo sviluppo del software in ambienti in cui l'insieme completo dei requisiti per un sistema non era completamente compreso o l'autore non sapeva come costruire il sistema completo. Sebbene sia stato proposto come modello di sviluppo, si adatta bene alla manutenzione. Il modello di processo coinvolge:

- Partendo dai requisiti di sistema esistenti, dal design, dal codice, dai test e dai documenti di analisi
- Rielaborare a partire dal documento appropriato in base all'analisi del sistema esistente, propagando i cambiamenti attraverso l'intero set di documenti
- Ad ogni passo del processo evolutivo, continuando a riprogettare, in base all'analisi.

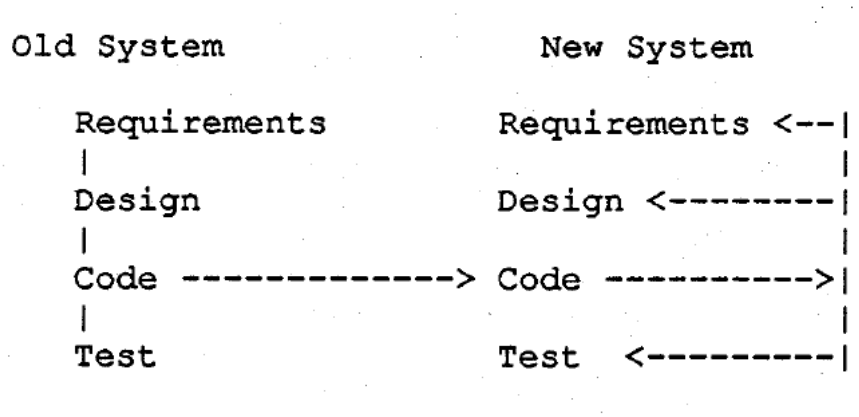

Figura6.2 Modello di miglioramento iterativo

Per vedere questo come un modello di manutenzione, si suppone che l'implementazione iniziale sia il sistema nel suo stato attuale nel processo di manutenzione evolutiva. Il processo presuppone che l'organizzazione di manutenzione abbia la capacità di analizzare il prodotto esistente, caratterizzare l'insieme proposto di modifiche, e riprogettare la versione corrente dove necessario per le nuove capacità. Di nuovo, il riutilizzo è implicito.

Modello di processo di full reuse. Mentre il miglioramento iterativo inizia con la valutazione del sistema esistente per la riprogettazione e la modifica, un modello di processo di full reuse inizia con l'analisi dei requisiti e la progettazione del nuovo sistema, con il concetto di riutilizzare qualsiasi requisito, progettazione e codice siano disponibili dal vecchio sistema. Il modello di processo di riutilizzo coinvolge:

- Avviare i requisiti per il nuovo sistema, riutilizzando il più possibile il vecchio sistema
- Costruire un nuovo sistema usando componenti del vecchio sistema o di altri sistemi disponibili nel repository sviluppando nuovi componenti dove appropriato.

Qui il riutilizzo è esplicito, il confezionamento dei componenti precedenti è necessario e l'analisi è necessaria per la selezione dei componenti appropriati.

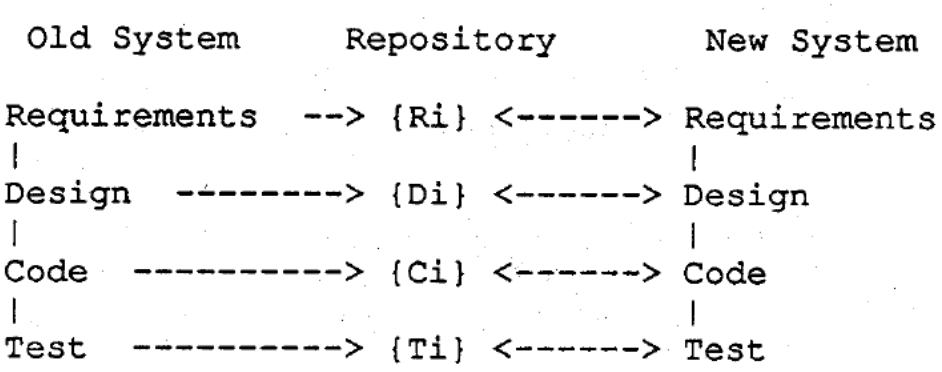

Figura 6.3 modello di full reuse

La differenza tra gli ultimi due approcci è più una questione di prospettiva. Il modello di full reuse libera lo sviluppatore a progettare la soluzione rispetto all'insieme disponibile di soluzioni di sistemi simili. Entrambi gli approcci incoraggiano la riprogettazione, ma il modello di full reuse suggerisce un forum più ampio e può portare allo sviluppo di più componenti riutilizzabili per sistemi futuri, mentre il modello di miglioramento iterativo suggerisce l'adattamento di un sistema esistente per le estensioni date.

6.2 Un quadro di riutilizzo

L'esistenza di diversi modelli di manutenzione solleva diverse domande. Qual è il modello più appropriato per un particolare ambiente? un particolare sistema? un particolare insieme di modifiche? il compito da svolgere? Come posso migliorare ogni fase del modello di processo che ho scelto? Come posso minimizzare il costo complessivo e massimizzare la qualità complessiva?

Per rispondere a queste domande si ha bisogno di un modello dell'oggetto di riutilizzo, il processo di adattamento di quell'oggetto per la sua applicazione di destinazione, e l'oggetto riutilizzato all'interno della sua applicazione di destinazione. Un semplice modello per il riutilizzo è dato nella figura 6.4. In questo modello, un oggetto è qualsiasi processo o prodotto software e una trasformazione è l'insieme delle attività che vengono eseguite nel riutilizzare quell'oggetto. Dato che l'ambito della nuova applicazione è compreso, le fasi sono:

- identificare i pezzi candidati riutilizzabili del vecchio oggetto
- capirli
- modificandoli secondo le nostre esigenze

- integrandoli nel processo

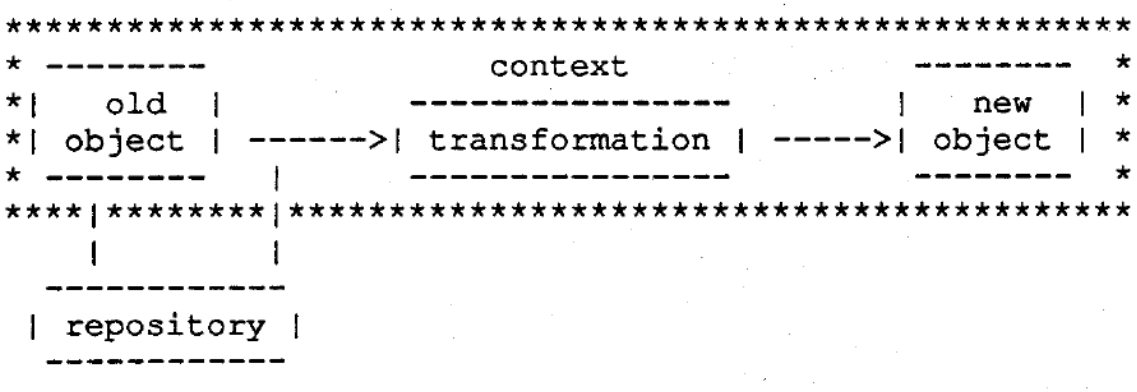

Figura 6.4 Un semplice modello reuse

Per completare il modello, si ha bisogno di un quadro per categorizzare gli oggetti, le trasformazioni e il contesto. Il quadro dovrebbe coprire varie categorie, ad esempio oggetto di riutilizzo: processo, prodotto. All'interno di ogni categoria ci sono vari schemi di classificazione per i prodotti, ad esempio documento dei requisiti, modulo di codice, piano di test, e processo: ad esempio stima dei costi, analisi dei rischi, progettazione.

Ci sono una varietà di approcci al riutilizzo e schemi che classificano l'oggetto del riutilizzo [Biggerstaff,Bowen,Freeman,IEE,Jones,Prieto-Diaz,Shaw].

Le dimensioni dell'oggetto includono:

- *Tipo di oggetto di riutilizzo:* Qual è una caratterizzazione dell'oggetto di riutilizzo candidato? Esempi di categorie e classificazioni sono: processo (ad es. progettazione, test) e prodotto (ad es. applicazione, strumento).
- *Autosufficienza*: Quanto è indipendente e comprensibile l'oggetto di riutilizzo candidato? Esempi di categorie e classificazioni sono: indipendenza sintattica (es. strettamente accoppiato), indipendenza semantica (es. funzionalità simili), e precisione della specifica (es. formale, informale).
- *Qualità dell'oggetto di riutilizzo:* Quanto è buono l'oggetto di riutilizzo candidato? Esempi di categorie e classificazioni sono: maturità (ad es. appena sviluppato, usato in un'applicazione) complessità (ad es. bassa complessità ciclomatica) capacità di rivivere (ad es. nessun fallimento durante l'uso precedente).

Le dimensioni del contesto includono:

- *Dominio dei requisiti*: Quanto sono simili i domini dei requisiti degli oggetti di riutilizzo candidati e i progetti attuali o futuri? Alcune categorie e classificazioni di esempio sono: applicazione (es. software di supporto a terra per satelliti), distanza (es. stessa applicazione, algoritmi simili/differente focus del problema).
- *Dominio della soluzione*: Quanto sono simili il processo di evoluzione che ha portato agli oggetti di riuso candidati e quelli usati nei progetti attuali e futuri? Alcune categorie e classificazioni di esempio sono: modello di processo (per esempio modello a cascata) metodo di progettazione (per esempio decomposizione delle funzioni) linguaggio di programmazione (per esempio FORTRAN).
- *Meccanismo di trasferimento della conoscenza*: come vengono passate le informazioni sugli oggetti di riutilizzo candidati e il loro contesto ai progetti attuali e futuri? Un esempio di classificazione è: esseri umani (per esempio un sottoinsieme del team di sviluppo che fa manutenzione, un team separato che fa manutenzione).

Le dimensioni di trasformazione includono:

- *Tipo di trasformazioni*: Come caratterizziamo le attività di trasformazione da eseguire? Alcune categorie e classificazioni di esempio sono: Percentuale di cambiamento (es. 0%, 5%), direzione (es. da generale a specifico del dominio, da specifico del progetto a specifico del dominio), meccanismo di modifica (es. verbatim, parameter ization, template-based, unconstrained) e meccanismo di identificazione (es. per nome, per requisiti funzionali).
- *Integrazione delle attività:* Come integriamo le attività di trasformazione nello sviluppo del nuovo sistema? Alcune categorie e classificazioni di esempio sono: attività di fase eseguite nella pianificazione del nuovo sviluppo (per esempio stima dei costi, analisi dei rischi), costruzione (per esempio sviluppo dei requisiti), analisi (per esempio test).
- *Qualità trasformata*: Qual è il contributo dell'oggetto di riuso nel contesto del nuovo sistema rispetto agli obiettivi fissati per esso? Esempi di categorie e classificazioni sono: affidabilità (es. nessun guasto associato a quel componente) e prestazioni (es. soddisfa il requisito di tempistica).

6.3 Confrontare i modelli usando il quadro di riutilizzo

Quando si applica il quadro di riutilizzo al processo di manutenzione, ci si concentra su un insieme di oggetti di riutilizzo che sono i documenti del prodotto. Si confrontano i vari modelli secondo le dimensioni date.

Considerate le dimensioni dell'oggetto di riuso:

Per quanto riguarda il tipo di oggetto di riutilizzo, l'oggetto dei modelli quick fix e di miglioramento iterativo è l'insieme dei documenti che rappresentano il vecchio sistema. L'oggetto del modello di full reuse è il repository che include il vecchio sistema.

Per quanto riguarda l'autosufficienza, tutti i modelli dipendono dall'unità di cambiamento. Il modello di correzione rapide dipende da quanto l'evoluzione ha avuto luogo poiché l'entropia può aver destrutturato il sistema. Nel miglioramento iterativo, il sistema evoluto dovrebbe migliorare per l'applicazione specifica e per l'insieme appropriato di cambiamenti, l'unità di cambiamento dovrebbe essere più visibile. Nel modello di full reuse, il sistema evoluto dovrebbe migliorare rispetto all'indipendenza degli oggetti di riutilizzo per l'applicazione generale, a seconda della qualità e della maturità del repository.

Per quanto riguarda la qualità dell'oggetto di riuso, il modello quick fix offre poca conoscenza della qualità del vecchio oggetto. Nel miglioramento iterativo, la fase di analisi fornisce una giusta valutazione della qualità rispetto alla particolare applicazione. Nel full reuse, si ha una valutazione della qualità dell'oggetto di riutilizzo attraverso diversi sistemi.

Considerate le dimensioni del contesto:

Per quanto riguarda il dominio dei requisiti, il modello quick fix e il modello di miglioramento iterativo presuppongono la stessa applicazione, di fatto lo stesso progetto. Il modello di full reuse permette una variazione gestibile nel dominio dell'applicazione, a seconda di ciò che è disponibile nel repository.

Per quanto riguarda il dominio della soluzione, il modello quick fix presuppone che durante la manutenzione esista la stessa struttura della soluzione come durante lo sviluppo. Non c'è nessun cambiamento nel design di base o nella struttura del nuovo sistema. Nel modello di miglioramento iterativo, poiché la riprogettazione fa parte del modello, è permessa qualche modifica alla struttura

della soluzione. Il modello di full reuse permette grandi differenze nella struttura della soluzione, cioè una riprogettazione completa è possibile passando dalla decomposizione funzionale alla progettazione orientata agli oggetti.

Per quanto riguarda il meccanismo di trasferimento delle conoscenze, il modello quick fix e il miglioramento iterativo funzionano meglio con le stesse persone. Il modello di full reuse può compensare il fatto di avere un team diverso, supponendo di avere specialisti dell'applicazione e un archivio di oggetti di riutilizzo ben documentato.

Considerate le dimensioni di trasformazione:

Per quanto riguarda il tipo di attività, il modello quick fix usa tipicamente un approccio di ricerca del codice sorgente, lettura per la comprensione, modifica non vincolata e ricompilazione. Il miglioramento iterativo inizia tipicamente con una ricerca attraverso il documento più rilevante, cambiandolo e continuando attraverso i documenti successivi usando una varietà di meccanismi di modifica. Il full reuse utilizza una ricerca nella libreria e una varietà di meccanismi di modifica a seconda del tipo di cambiamento. Qui la modifica è fatta off-line.

Per quanto riguarda l'integrazione delle attività, nel modello quick fix, tutte le attività sono eseguite allo stesso tempo. Il miglioramento iterativo associa le attività a tutte le normali fasi di sviluppo. Nel modello di full reuse, l'identificazione dei pezzi riutilizzabili candidati è fatta durante la pianificazione del progetto e le altre attività sono fatte durante lo sviluppo.

Per quanto riguarda la qualità trasformata, il modello quick fix di solito funziona meglio su piccole modifiche ben contenute poiché il loro effetto sul sistema può essere compreso e verificato nel contesto. Il miglioramento iterativo è più appropriato per cambiamenti più grandi dove la fase di analisi può fornire una migliore valutazione dell'effetto completo dei cambiamenti. Il full reuse è appropriato per grandi cambiamenti e grandi riprogettazioni. Qui, l'analisi e la storia precedente delle prestazioni degli oggetti di riutilizzo supportano la qualità.

Date queste differenze, si possono fornire alcune analisi dei vari modelli di processo di manutenzione e raccomandare dove potrebbero essere più applicabili. Ma prima, si discute sulla relazione tra i modelli di processo di sviluppo e di manutenzione. In un certo senso lo sviluppo può essere considerato un sottoinsieme della manutenzione. Gli ambienti di manutenzione differiscono

da quelli di sviluppo per quanto riguarda i vincoli sulla soluzione, la domanda del cliente, la tempestività della risposta e l'organizzazione.

La maggior parte delle organizzazioni di manutenzione sono impostate per il modello quick fix ma non per i modelli di miglioramento iterativo o di processo di riutilizzo. Questo perché rispondono alla tempestività, ad esempio un guasto del sistema deve essere riparato immediatamente, o una richiesta del cliente, ad esempio una modifica della funzionalità del sistema. Chiaramente questi sono punti di forza per il modello di soluzione rapida. Ma le debolezze del modello sono che la modifica è di solito una patch, non ben documentata, la struttura del sistema è stata parzialmente distrutta, il che rende l'evoluzione futura del sistema difficile e piena di errori, e non è compatibile con i processi di sviluppo. Questo modello è meglio usato quando la tempestività e la necessità del cliente sono dominanti e ci sono poche possibilità che il sistema venga modificato di nuovo.

Il modello di miglioramento iterativo permette la riprogettazione in modo che la struttura del sistema si evolva e la modifica futura sia più facile. Si concentra sul sistema particolare, rendendolo il migliore possibile. È compatibile con i modelli di processo di sviluppo. Gli svantaggi sono che è un approccio più costoso e possibilmente meno tempestivo (nel breve periodo) rispetto al modello quick fix e fornisce poco supporto per i generici o i sistemi simili futuri. È un buon approccio da usare quando il prodotto avrà una lunga vita e si evolverà nel tempo. In questo caso, se anche la tempestività è un vincolo, il modello quick fix può essere usato come una patch e il modello di miglioramento iterativo può essere usato per il cambiamento a lungo termine, sostituendo la patch.

Il modello di processo di full reuse fornisce al manutentore una prospettiva più ampia, si concentra sullo sviluppo a lungo raggio per un insieme di prodotti e ha l'effetto collaterale di creare componenti riutilizzabili di tutti i tipi per sviluppi futuri. È compatibile con i modelli di processo di sviluppo e, in effetti, è il modo in cui vorremmo che tali modelli evolvessero. Lo svantaggio è che è più costoso nel breve periodo, non è appropriato per piccole modifiche, ma può essere usato insieme ad altri modelli. È meglio usarlo quando viviamo in multi-ambienti di prodotti o sviluppo generico dove la linea di prodotti ha una lunga vita.

La valutazione data sopra è informale e intuitiva. Questo è dovuto al fatto che è un'analisi qualitativa. Per fare un'analisi quantitativa si hanno bisogno di modelli quantitativi degli oggetti di riutilizzo, delle trasformazioni e del contesto. Si ha

bisogno di un quadro di misurazione per caratterizzare attraverso la categorizzazione e la classificazione, la valutazione, la previsione e la motivazione per supportare la gestione e le decisioni tecniche. Per fare questo si applica il paradigma obiettivo/domanda/metrica ai modelli.

6.4 Il paradigma della metrica della domanda dell'obiettivo

Il paradigma obiettivo/domanda/metrica (GQM) [Basili86, Basili88B, Basili84] rappresenta un approccio sistematico per fissare gli obiettivi del progetto (su misura per le esigenze specifiche di un'organizzazione), definendoli in modo operativo e trattabile, raffinandoli in un insieme di domande quantificabili che implicano un insieme specifico di metriche e dati da raccogliere. La trattabilità di questo processo d'ingegneria del software permette l'analisi dei dati raccolti e delle metriche composte nel contesto appropriato delle domande e dell'obiettivo originale. Questo contesto supporta il feedback (integrando aspetti analitici e costruttivi) e l'apprendimento (definendo la procedura di sintesi appropriata per le esperienze di livello inferiore in quelle di livello superiore).

Il processo di definizione degli obiettivi e la loro raffinazione in domande quantificabili è complesso e richiede esperienza. Per supportare questo processo, è stato sviluppato un set di modelli per la definizione degli obiettivi e un set di linee guida per derivare domande e metriche [Basili88A].

Gli obiettivi sono definiti in termini di scopo, prospettiva e ambiente Esistono diverse serie di linee guida per definire le domande relative al prodotto e al processo. Le domande relative al prodotto sono formulate allo scopo di definire il prodotto (per esempio, attributi fisici, costo, cambiamenti e difetti, contesto), definire la prospettiva di qualità di interesse (per esempio, affidabilità, facilità d'uso), e fornire un feedback dalla particolare prospettiva di qualità. Le domande relative al processo sono formulate allo scopo di definire il processo (qualità d'uso, dominio d'uso), definire la prospettiva di qualità di interesse (per esempio, riduzione dei difetti, economicità d'uso), e fornire un feedback dalla particolare prospettiva di qualità.

6.4.1 Applicazione del paradigma della metrica della domanda dell'obiettivo

Applicando il paradigma obiettivo/domanda/metrica, si definiscono gli obiettivi del processo di manutenzione e si articolano i problemi associati alla scelta del

modello di processo appropriato, fornendo alla direzione le domande a cui bisogna rispondere per prendere decisioni intelligenti, capire i compromessi ed eseguire l'analisi dei rischi. Ci sono una varietà di obiettivi che possiamo generare. Per esempio: determinare quale modello di processo dovrebbe essere scelto per un particolare prodotto, migliorare le nostre prestazioni o evolvere una migliore definizione di uno qualsiasi dei modelli per una particolare linea di prodotti.

In quello che segue si genera un obiettivo campione per la manutenzione e si fornisce una lista parziale delle domande coinvolte. Alcune delle risposte saranno ovvie, sia nelle misure che richiedono di prendere o nelle informazioni richieste dagli esperti; altre no. Così un obiettivo per la manutenzione nel contesto del quadro del riuso potrebbe essere:

Scopo:

Valutare i requisiti dei nuovi prodotti per riutilizzare il più possibile i prodotti disponibili.

Prospettiva:

Esaminare il costo e l'evoluzione futura dello sviluppo dal punto di vista dell'organizzazione.

Ambiente:

Insieme ai fattori ambientali standard, come il fattore risorse, i fattori problema, si presta particolare attenzione alle tre dimensioni di contesto del quadro di riutilizzo.

Dominio dei requisiti:

Si stanno usando oggetti di prodotto dello stesso dominio applicativo, anche se si ha la possibilità di scegliere componenti candidati da altri domini applicativi.

Dominio della soluzione:

Questo definisce i modelli di processo, i metodi e gli strumenti che sono stati usati nello sviluppo del prodotto esistente. Se gli stessi processi devono essere usati per il progetto evoluto, allora non ci sono problemi di riutilizzo. Tuttavia, il modello di riutilizzo ci permette di cambiare i processi (e quindi eventualmente la struttura del prodotto) al costo di riutilizzare meno del progetto precedente. Se ci devono essere cambiamenti, allora dobbiamo valutare il costo della modifica del

processo e del prodotto risultante rispetto ai guadagni per il cambiamento del processo.

Meccanismo di trasferimento delle conoscenze:

Se il gruppo di manutenzione è lo stesso del gruppo di sviluppo, allora non c'è bisogno di trasferire le conoscenze. Se sono diversi, allora ci sono preoccupazioni che devono essere valutate rispetto all'applicazione, al processo e alla conoscenza del prodotto dei manutentori e ai tipi di documentazione disponibili.

Definizione del prodotto:

Nel considerare il prodotto, in realtà se ne ha diversi. Il nuovo prodotto da costruire, cioè la nuova versione del sistema, e le vecchie versioni più qualsiasi altro sistema che sia rilevante.

6.4.1.1 Dimensioni del prodotto

Nuovi prodotti:

Quanti requisiti ci sono in totale per il nuovo sistema?

Vecchi prodotti:

Qual è la mappatura dei requisiti ai componenti del sistema? Qual è la misura della complessità della tracciabilità?

Quanto sono indipendenti i componenti da modificare?

Qual è la complessità del sistema e dei singoli componenti del sistema?

Deposito:

Quali componenti candidati sono disponibili nel repository e quali sono le loro classificazioni di contesto, trasformazione e oggetto?

Differenza tra nuovo e vecchio:

Quanti requisiti ci sono che non sono nel vecchio sistema? (Categorizzare per dimensione, nuovo vs. modifica del vecchio vs. cancellazione del vecchio, ecc.)

Quanti componenti devono essere cambiati, aggiunti, cancellati? (suddivisi per dimensione e tipo di cambiamento)

6.4.1.2 Modifiche/difetti

Quanti errori, difetti, guasti (categorizzati per classe) sono associati ai requisiti e ai componenti che devono essere cambiati?

Qual è il profilo di qiei cambiamenti al sistema originale prima di questo cambiamento?

6.4.1.3 Costo

Qual è il costo della comprensione dei nuovi requisiti?

Qual è il costo stimato per costruire un nuovo sistema, riutilizzando l'esperienza e le parti del vecchio progetto?

Qual era il costo del vecchio sistema in totale? Qual era il costo di ogni versione?

Qual è il costo stimato della modifica del vecchio sistema per soddisfare i nuovi requisiti?

6.4.1.4 Contesto del cliente

Come verrà usato il nuovo sistema?

Quali sono le potenziali modifiche future basate sulla nostra analisi dei profili dei clienti, le modifiche passate e lo stato delle tecnologie?

Prospettiva:

costo ed evoluzione futura dello sviluppo

Modello di prospettiva: costo della modifica del design del sistema rispetto alle modifiche future previste Parametri:

- il tempo di vita del sistema
- il costo dell'evoluzione futura del sistema
- il costo dell'evoluzione del vecchio sistema rispetto alla ricostruzione da vecchie parti

Feedback:

- Il modello è appropriato?
- Come si può migliorare il modello?
- Come si possono migliorare le stime?
- Come migliorare le classificazioni?

- Come si possono migliorare le attività?

Il paradigma obiettivo/domanda/metrica permette di sviluppare altri obiettivi per il riutilizzo. Questi possono essere sviluppati sia che l'oggetto di riutilizzo sia un processo o un prodotto.

Si considerano i seguenti esempi:

- Valutare l'attività di modifica all'interno del processo di riutilizzo per migliorarlo. Esaminare il costo e la correttezza dell'oggetto risultante dal punto di vista del cliente.
- Prevedere il modello di processo di manutenzione appropriato per eseguire quello corretto. Esaminare il suo costo rispetto alle esigenze del cliente e le future evoluzioni del sistema dal punto di vista dell'azienda.
- Valutare il metodo standard di progettazione aziendale per valutare come avrebbe dovuto essere adattato al progetto attuale. Esaminare la sua efficacia dal punto di vista del designer.
- Valutare i componenti del prodotto esistente per determinare se riutilizzarli. Esaminare la loro indipendenza e adeguatezza funzionale dal punto di vista del loro uso in sistemi futuri.
- Prevedere la capacità di un insieme di componenti di codice da integrare nel sistema attuale dal punto di vista dello sviluppatore.
- Motivare lo sviluppo di un insieme di componenti riutilizzabili al fine di ingegnerizzarli per il riutilizzo. Esaminare la struttura di ricompensa dal punto di vista del manager e dello sviluppatore.

6.5 Abilitatori del riuso

Ci sono una varietà di meccanismi di supporto necessari per raggiungere il massimo riutilizzo che non sono stati sufficientemente enfatizzati in letteratura. Per supportare queste attività c'è bisogno di un paradigma di miglioramento che aiuti l'organizzazione a valutare, imparare e migliorare il processo e il prodotto software, un ambiente di evoluzione orientato al riutilizzo che motivi e supporti il riutilizzo, e un supporto automatizzato per quel modello e per il processo di misurazione e valutazione.

Il paradigma del miglioramento: Il paradigma del miglioramento [Basili86] è un modello di processo organizzativo di alto livello in cui l'organizzazione impara come migliorare il proprio prodotto e processo. All'interno di questo modello

l'organizzazione dovrebbe imparare a prendere decisioni migliori su quale modello di processo usare per la manutenzione dei loro futuri prodotti software basandosi sull'apprendimento delle prestazioni passate.

Il paradigma è definito come segue:

1. *Pianificazione*. Ci sono tre attività integrate alla pianificazione che vengono applicate iterativamente:

- Caratterizza l'ambiente attuale del progetto. Fornisce un'analisi quantitativa dell'ambiente e un modello del progetto nel contesto di quell'ambiente. Nel contesto della manutenzione, la caratterizzazione dovrebbe fornire dati sulle dimensioni del prodotto, dati sui cambiamenti e sui difetti, dati sui costi e sul cliente, dati di contesto per le versioni precedenti del sistema da modificare, informazioni su quali classi di componenti candidate sono disponibili nel repository per il nuovo sistema, e qualsiasi feedback informativo da progetti precedenti sull'esperienza con i diversi modelli per i tipi di modifiche richieste.
- Impostare gli obiettivi e raffinarli in domande e metriche quantificabili usando il paradigma dell'obiettivo/domanda/metrica, per il successo delle prestazioni del progetto e il miglioramento rispetto alle prestazioni del progetto precedente. Questo consiste in un'analisi top-down degli obiettivi che decompone iterativamente gli obiettivi di alto livello in sotto obiettivi dettagliati. L'iterazione termina quando ha prodotto sotto-obiettivi che possiamo misurare direttamente. Per la manutenzione questo comporta lo sviluppo di specifici G/QJM come specificato nella sezione precedente.
- Scegliere e adattare il modello di costruzione appropriato per questo progetto e i metodi e gli strumenti di supporto per soddisfare gli obiettivi del progetto rispetto all'ambiente caratterizzato. La comprensione quantitativa dell'ambiente ci permette di scegliere il modello di processo appropriato e di mettere a punto i metodi e gli strumenti necessari per essere più efficaci. Per esempio, conoscere l'effetto delle precedenti applicazioni dei vari modelli e metodi di manutenzione nella creazione di nuovi progetti da vecchi sistemi ci permette di scegliere e mettere a punto il modello di processo appropriato e i metodi che sono stati storicamente più efficaci nella creazione di nuovi sistemi del tipo richiesto da vecchie versioni e componenti nel repository.

2. *Analisi.* Analizzare i dati per valutare le pratiche attuali, determinare i problemi, registrare i risultati e fare raccomandazioni per il miglioramento. Si deve condurre l'analisi dei dati durante e dopo il progetto. Il paradigma obiettivo/domanda/metrica fornisce la tracciabilità dagli obiettivi alla metrica e viceversa. Questo permette di interpretare la misurazione nel contesto, assicurando un'analisi più semplice e mirata. Le misure operative guidate dagli obiettivi forniscono una struttura per il tipo di analisi necessaria.

3. *Leaming e feedback*. Questa fase comporta l'organizzazione e la codifica dell'esperienza quantitativa e qualitativa acquisita dal progetto corrente in una base di informazioni aziendali per aiutare a migliorare la pianificazione, lo sviluppo e la valutazione dei progetti futuri. I risultati della fase di analisi e interpretazione possono essere riportati all'organizzazione per cambiare il modo in cui fa affari sulla base di successi e fallimenti esplicitamente determinati. In questo modo, si può imparare come migliorare la qualità e la produttività, e come migliorare la definizione e la valutazione degli obiettivi. Si può iniziare il prossimo progetto armati dell'esperienza acquisita da questo e dai progetti precedenti. Per esempio, capire i problemi associati ad ogni nuova versione di un sistema, fornisce intuizioni sulla necessità di riprogettazione e ri-sviluppo.

Un ambiente orientato al riuso: Il riuso può essere realizzato più efficacemente in un ambiente che supporta il riuso [Basili88B ,Biggerstaff,IEE]. Gli ambienti di ingegneria del software forniscono cose come un database di progetto e supportano l'interazione delle persone con metodi, strumenti e dati di progetto. Tuttavia, l'esperienza non è controllata dalla banca dati del progetto o posseduta dall'organizzazione. Il riutilizzo esiste solo implicitamente.

Si deve essere in grado di incorporare il modello del processo di riutilizzo nel contesto dello sviluppo. Si devono combinare i modelli di sviluppo e manutenzione per massimizzare le dimensioni del contesto. Si deve integrare la caratterizzazione, la valutazione, la previsione e la motivazione nel processo. Si ha bisogno di supportare l'apprendimento e il feedback per rendere il riutilizzo fattibile. Si propone che il modello di riutilizzo possa esistere nel contesto del paradigma di miglioramento, rendendo possibile supportare tutti i requisiti di cui sopra.

Il progetto TAME: Il paradigma di miglioramento e il modello di processo orientato al riuso richiedono un supporto automatizzato per la base di dati, l'esperienza codificata e il repository di progetti priot e componenti riutilizzabili

[Basili88A,Basili88B,IEE]. Si ha bisogno di automatizzare il più possibile il processo di misurazione, e fornire un ambiente di strumenti per manager e ingegneri per sviluppare obiettivi specifici del progetto, e generare definizioni operative basate su questi obiettivi che specifichino le metriche appropriate necessarie per la valutazione. La valutazione e il feedback non possono essere fatti in tempo reale senza un supporto automatizzato. Il supporto automatizzato aiuterà nell'analisi post mortem.

L'obiettivo del sistema TAME [Basili88A] è di istanziare e integrare i paradigmi di miglioramento e di obiettivo/domanda metrica e aiutare nell'adattamento del processo di sviluppo del software. Ma può anche supportare il modello di processo orientato al riuso. Il modello di ambiente TAME contiene meccanismi di base per supportare l'apprendimento sistematico e il riutilizzo. Per aiutare l'apprendimento sistematico fornisce il supporto per la registrazione dell'esperienza, la generazione off-line o l'adattamento dell'esperienza, e la formalizzazione dell'esperienza. Per aiutare il riutilizzo sistematico supporta i meccanismi per usare l'esperienza esistente e la generalizzazione o l'adattamento on-line dell'esperienza candidata. In questo modo cerca di integrare sia l'apprendimento che il riutilizzo in un modello di evoluzione globale.

L'applicazione del concetto di sistema TAME alla manutenzione fornirà un meccanismo per scegliere il modello di processo di manutenzione appropriato per un particolare progetto e fornirà dati per aiutare a imparare come fare un lavoro migliore di manutenzione.

7. SVILUPPO ORIENTATE AL RIUSO E RALLENTAMENTO DEL DEGRADO NELLA MANUTENZIONE

Tutti i sistemi software sono soggetti a degrado durante la loro vita. Più precisamente, l'affidabilità diminuisce a causa della propensione all'errore della manutenzione, la struttura del sistema è impattata da una manutenzione non accurata legata a restrizioni di tempo, la tracciabilità tra i documenti e tra i documenti e il codice si perde a causa di un'analisi di impatto non appropriata delle richieste di manutenzione. Inoltre, come conseguenza dei vari interventi effettuati sul sistema, la manutenibilità diminuisce. Quindi, sebbene la manutenzione di un sistema software sia di primaria importanza per mantenere il sistema allineato alle continue esigenze e richieste di chi lo utilizza, essa ha inevitabilmente un alto impatto sulla qualità e ne causa il degrado. Questo avviene indipendentemente dal processo di manutenzione adottato.

Esperienze all'interno della comunità dell'ingegneria del software hanno affrontato tali problemi [Lehman, Visaggio97, Visaggio01, France]. Come conseguenza dei continui interventi su un sistema software, la manutenzione diventa più costosa e meno accurata. Allo stesso tempo, il sistema riduce il suo valore economico. È quindi importante investire su un processo di manutenzione che sia in grado di ridurre o almeno controllare il degrado della qualità del software.

Tra i vari tipi di processi di manutenzione, un approccio interessante è stato introdotto nei primi anni '90 da Basili [Basili90]. Egli lo chiama "modello di full reuse". Lo schema di questo processo, descritto in [Basili90], è rappresentato nella figura 7.1. Si tratta di un processo di manutenzione che adotta il riutilizzo di componenti software sviluppati in precedenza che sono stati adeguatamente memorizzati in un repository e sono recuperabili durante le attività di manutenzione. Un punto debole è il fatto che in letteratura non sembrano essere riportate esperienze relative alla sperimentazione in ambito industriale del modello.

Come riportato in [Basili90], "Per un riutilizzo efficace, è necessario essere in grado di incorporare il modello del processo di riutilizzo nel contesto dello sviluppo. È necessario combinare i modelli di sviluppo e manutenzione per massimizzare le dimensioni del contesto. È necessario integrare la caratterizzazione, la valutazione, la previsione e la motivazione nel processo". In altre parole, il successo del riutilizzo nella manutenzione è una conseguenza dell'adozione del riutilizzo nella fase di

sviluppo. Secondo[Baldassarre03] , il riutilizzo è un fattore chiave per ridurre il degrado della qualità causato dalla manutenzione.

Nel lavoro [Baldassarre03] si sperimenta il modello di manutenzione del riuso completo in un ambiente industriale, concentrandonsi sulla seguente domanda di ricerca:

"Il full reuse riduce la degradazione del software e migliora il controllo della sua qualità, rispetto a un processo di manutenzione convenzionale basato sul miglioramento iterativo?".

La domanda di ricerca che confronta il miglioramento iterativo con un modello quick fix è stata studiata empiricamente in [Visaggio01]. I risultati hanno evidenziato che il processo di miglioramento iterativo era più efficace del quick fix.

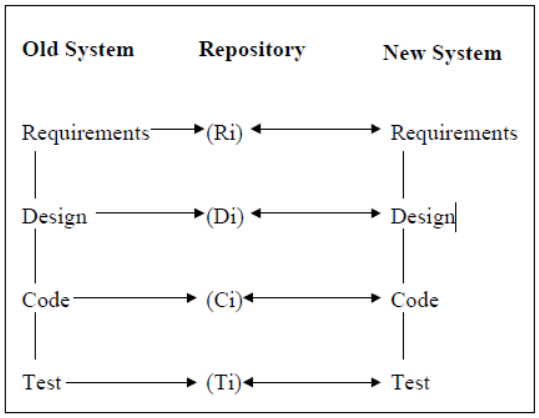

Figura 7.1 Modello Full Reuse

La prova della risposta alla domanda di ricerca precedentemente definita è stata ottenuta attraverso un'indagine sul campo su due progetti realizzati in una piccola-media impresa (PMI), sulla base di due diversi processi di manutenzione che sono stati confrontati.

7.1 Il riutilizzo in letteratura

Negli ultimi anni la comunità scientifica e industriale ha focalizzato la loro attenzione su tecnologie e tecniche che si concentrano su concetti di riuso in grado di ridurre i costi di produzione e manutenzione del software e controllare sia la produttività che il degrado della qualità. Questa continua evoluzione ha portato a paradigmi come Product Lines [Knauber] e Aspect Oriented Programming [Wiley] passando per la programmazione Object Oriented [Booch].

Un fattore costante in ognuno di questi processi è che considerano il concetto di "riuso" come un punto di forza e lo usano come linea guida per strutturare un sistema software.

Il riutilizzo del software non è nuovo alla comunità dell'ingegneria del software. È stato introdotto da Dough McIlroy in [McIlroy] alla fine degli anni '60. Negli anni '80 sono stati presentati e discussi i progetti relativi a questo tema. Negli anni successivi l'interesse di diversi governi e industrie per il riutilizzo è stato dimostrato attraverso i numerosi progetti che sono stati finanziati e realizzati [Martin, IEE]. Le comunità scientifiche e industriali hanno condotto altri studi su questo argomento nel corso degli anni, come mostrato in [Prieto-diaz, Frakes, Pighin, Toffolon]; sono stati considerati diversi punti di vista, come le barriere all'adozione del riutilizzo del software [Sherif], o l'estrazione di un insieme di componenti riutilizzabili analizzando quelli esistenti disponibili [Caldiera].

Ci sono molte definizioni e tipi di riutilizzo riportati in letteratura. Per quanto riguarda il riutilizzo, si riferisce alla definizione data in [Morisio]: "Il riutilizzo del software è la pratica sistematica di sviluppare software da uno stock di blocchi di costruzione, in modo che le somiglianze nei requisiti e/o nell'architettura tra le applicazioni possano essere sfruttate per ottenere benefici sostanziali in termini di produttività, qualità e prestazioni aziendali". È adeguatamente integrato nell'intero processo, sia esso di sviluppo o di manutenzione, e si riferisce a tutti gli artefatti prodotti, a partire dalle specifiche dei requisiti, includendo la progettazione, il codice e i casi di test. Il modello Full reuse a cui ci si riferisce è coerente con questa definizione.

Il riutilizzo ad hoc è comunemente e informalmente chiamato "taglia e incolla". Non si è d'accordo sul fatto che questo possa essere considerato una forma di riutilizzo. Infatti, se un pezzo di codice di programma viene "tagliato" e poi "incollato" in un altro programma, la porzione di codice che è presente in entrambi i programmi dipende dal contesto del programma e non solo dalle interfacce di comunicazione con la parte rimanente del programma. Pertanto, quando uno dei due programmi viene modificato nella parte relativa al codice "tagliato e incollato", la modifica non si applica necessariamente a entrambi i programmi. Di conseguenza, anche se il codice è stato riutilizzato, ciò non implica che anche la manutenzione sia riutilizzabile. Inoltre, supponendo che quando una parte di codice che è stata "tagliata" da un programma e "incollata" in molti altri, la modifica nella parte "tagliata" sia adeguata per tutte le altre parti "incollate", è difficile mantenere la tracciabilità di dove sono

state incollate le parti tagliate. Pertanto, molte correzioni non sono estese ad altri programmi e rimangono limitate al caso specifico in cui sono state fatte.

Molti lavori in letteratura considerano il riuso del software rispetto al processo di sviluppo e ne analizzano i benefici.In particolare, [Morisio00, Morisio02A, Rine, Morisio02B, 27, Lim, Joos, Poulin] sottolineano come il riutilizzo di componenti (o moduli) software esistenti impatti su molti fattori di qualità come: riduzione del tempo di ciclo, perché si deve fare meno lavoro per produrre i vari componenti; miglioramento della qualità e migliore affidabilità del software, perché i componenti riutilizzati sono già testati quando vengono selezionati per il riutilizzo e poi vengono ritestati se sono soggetti a ulteriori modifiche, maggiore produttività, perché è necessario meno sforzo per produrre lo stesso prodotto finale

Tuttavia, l'introduzione del riuso sistematico come pratica comune all'interno di un processo di manutenzione, è ancora una questione molto poco studiata anche se a nostro parere è un fattore chiave per ridurre il degrado del software dovuto alla manutenzione continua. In [Basili90] Basili afferma come la manutenzione possa essere trattata come un processo di sviluppo orientato al riuso.

L'articolo [Baldassarre03] fornisce la prova di come il modello di manutenzione di full reuse permetta di ridurre il degrado del sistema.

7.2 Modello di manutenzione del Full reuse

In [Basili90] vengono proposti tre diversi tipi di modelli di riutilizzo. I principali concetti comuni ad ogni modello di riutilizzo possono essere riassunti nei seguenti passi:

- identificazione dei componenti del software;
- analisi dei componenti per vedere se sono adatti alle esigenze specifiche del contesto in cui devono essere applicati;
- adattamento dei componenti al contesto;
- integrazione;

Basili afferma che "Qui il riutilizzo è esplicito, è necessario il packaging dei componenti esistenti e l'analisi per selezionare i componenti appropriati". Ciò che caratterizza il modello del full reuse è il fatto che utilizza un repository per memorizzare, cercare e recuperare i componenti software da riutilizzare.

Seguendo queste considerazioni e adottandole come linee guida, [Baldassarre03] ha definito un processo di manutenzione full reuse (RApid software Maintenance prOcess) che utilizza i componenti del repository per lo svolgimento delle attività di manutenzione, e uno strumento di supporto automatico al processo in grado di gestire il repository di componenti riutilizzabili.

Prima di descriverli, è il caso di illustrare cosa si intende per full reuse: Al tempo T0, si considera che S1 e S2 siano due implementazioni di un sistema software che sfrutta moduli riutilizzabili (Figura7.2). Si Suppone che al tempo T1 venga fatta una richiesta di manutenzione (MR) per un difetto [df21] in S2 e comporta la modifica dei moduli M3 e M31. Le versioni aggiornate di questi moduli sono memorizzate nel repository (Figura 7.3). In seguito, al tempo T2 viene fatto un MR per un difetto [df12] trovato in S1. L'attività di manutenzione implica il lavoro sul modulo M1 e può comportare il lavoro su una versione più recente del modulo memorizzato nel repository (Figura 7.4). Dopo la manutenzione e la memorizzazione del modulo nel repository (T3), tutti i moduli che compongono il sistema sono aggiornati con le loro ultime versioni. In altre parole, M3 e M31 sono aggiornati senza che ci sia un MR specifico per loro in S1 (Figura 7.5).

In sintesi, si assume[Baldassarre03] che il modello del full reuse permette di controllare la qualità del sistema mantenuto grazie all'aggiornamento dei suoi moduli. Inoltre rallenta la degradazione della qualità grazie al fatto che i moduli sono costantemente testati e perché il riutilizzo include tutti gli artefatti prodotti, che sono sempre aggiornati.

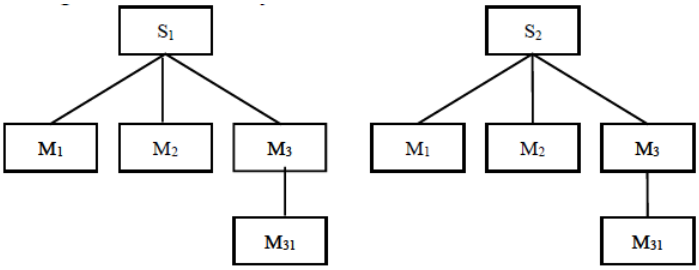

Figura 7.2 Due implementazioni di un sistema software

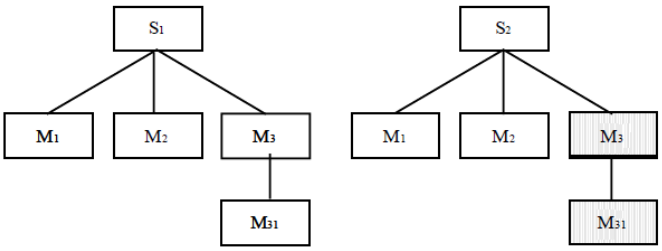

Figura 7.3 MR per S2

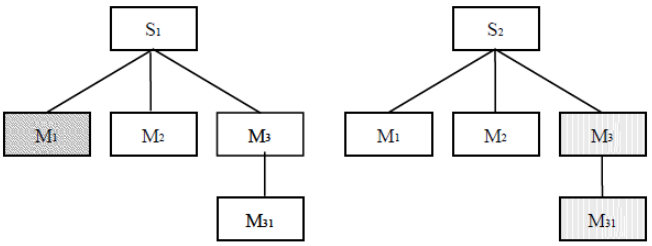

Figura 7.4 MR per S1

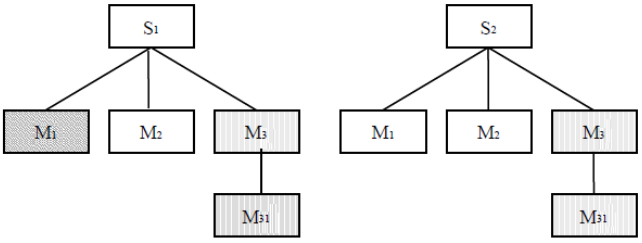

Figura7.5 Aggiornamento di sistema per S2

Le caratteristiche principali del processo di manutenzione del Full reuse (RAMO) che è stata definita e adottata all'interno della PMI considerata nell'indagine è rappresentato nella figura 7.6[Baldassarre03]. Può essere riassunto nei seguenti passi:

- viene ricevuta una richiesta di manutenzione (MR) di un sistema software in uso;
- dopo un'analisi d'impatto primaria, il manutentore identifica la lista dei moduli che sono impattati e richiedono manutenzione (*analisi d'impatto primaria*);

- per ogni modulo, l'ultima versione aggiornata viene recuperata dal repository. Se corrisponde e soddisfa la MR, nel senso che corregge il difetto, allora la vecchia versione del modulo che contiene il difetto viene sostituita con quella aggiornata (*progetto di modifica*).
- "Nessuna corrispondenza" nel repository significa che il modulo in esame non è mai stato modificato in momenti precedenti con riferimento allo specifico MR. In questo caso, il modulo è modificato "a mano" (*esecuzione di modifica*), cioè la modifica si riferisce ai modelli utilizzati per definire il modulo stesso, quindi la lista dei modelli viene recuperata dal repository e sottoposta a manutenzione per correggere i difetti. Una volta completato, il modulo aggiornato viene memorizzato nel repository. In questo modo il repository contiene le ultime versioni di tutti i moduli che compongono il sistema software.
- Si noti che il riutilizzo del modulo implica il riutilizzo della documentazione, del codice e dei casi di test. Pertanto, prima che un modulo sia aggiornato e memorizzato nel repository, è adeguatamente testato (*pianificazione dei test di regressione, esecuzione dei test, eliminazione dei difetti*).
- una volta che tutti i moduli impattati dalla MR sono stati modificati, ogni modulo rimanente del sistema software viene aggiornato con la sua ultima versione nel repository. In questo modo il sistema beneficia degli interventi di manutenzione effettuati in momenti precedenti su altri moduli che non sono direttamente coinvolti nella richiesta di manutenzione corrente. Da questo punto in poi i termini "RAMO" e "manutenzione full reuse" saranno usati indifferentemente.

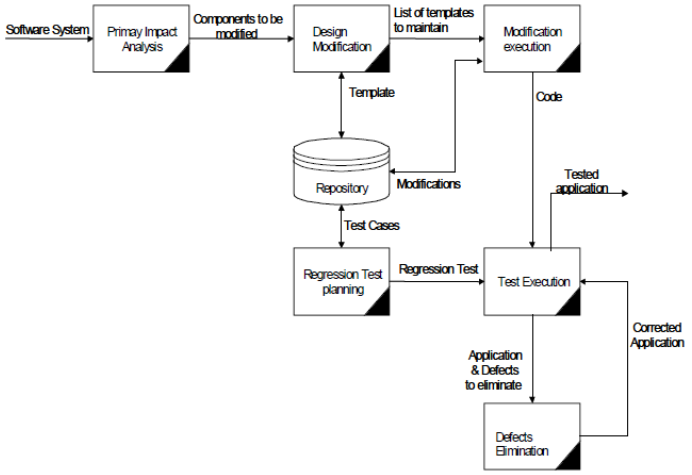

Figura7.6 RAMO: Software rapido prOcesso di manutenzione

Il supporto automatizzato è importante per un efficace full reuse, come affermato in [Basili90]. Questo è stato anche considerato in [Pighin, Toffolon] Per questo scopo si è sviluppato uno strumento chiamato RAPID (Rapid aPplication maIntenance and Development).

Tra le sue funzioni, lo strumento RAPID gestisce il deposito attraverso tre funzioni principali: RETRIEVE, MAINTENANCE e FULL REUSE.

- La funzione *RETRIEVE* cerca nel repository i moduli riutilizzabili come conseguenza di una richiesta di manutenzione. Come risultato di una ricerca con la funzione retrieve, se il repository contiene un modulo perfettamente corrispondente alla richiesta viene estratto, altrimenti fornisce il modulo che corrisponde parzialmente alla richiesta. In questo caso deve essere adattato dall'autore al contesto specifico in cui deve essere utilizzato. Infine, se nessun modulo corrisponde alla richiesta, lo strumento fornisce un modello che lo sviluppatore deve completare per generare il modulo.
- La funzione *MAINTENANCE* aggiorna il repository. Ogni volta che un modulo viene completato, questa funzione estrae le sue caratteristiche essenziali (descrizione, input al modulo, output prodotto) e le memorizza nel repository.
- Per quanto riguarda la funzione *FULL REUSE*, ogni volta che un modulo viene identificato nel repository per essere riutilizzato, questa funzione estrae tutte le informazioni ad esso associate come codice, casi di test e documentazione. La funzione aggiorna anche il sistema in manutenzione con le ultime versioni di tutti i componenti. Questo è possibile perché i moduli memorizzati nel repository sono caratterizzati da un nome di modulo, una descrizione, un input al modulo, un output prodotto e infine la data dell'ultimo aggiornamento.

7.3 Progetto CONV e Progetto FR

Per eseguire l'indagine sul campo [Baldassarre03] ha selezionato due progetti di manutenzione, tipici dell'organizzazione. Si trattava di interventi correttivi, adeguati e attività di manutenzione evolutiva e sono stati identificati, rispettivamente, come Progetto CONV e Progetto FR, per riferirsi a un processo di riutilizzo convenzionale e completo utilizzato durante l'esecuzione. Il primo era un'applicazione per la conversione automatica di valuta. Il secondo consisteva in un sistema software per procedure bancarie come il recupero del credito. Secondo [Fenton], saranno indicati

come progetti fratelli. In altre parole, il RAMO sarà applicato al Progetto FR in studio, mentre il progetto gemello (Progetto CONV) utilizzerà il processo di manutenzione convenzionale già in uso nella PMI.

La situazione ideale sarebbe stata quella di mantenere lo stesso sistema software in entrambi i processi del full reuse e di manutenzione convenzionale (miglioramento iterativo). Questo non è possibile in un campo industriale per ovvie ragioni di economia di produzione. Si noti che i due progetti erano progetti in corso eseguiti dalla PMI quindi, noi non poteva controllare tutte le variabili indipendenti riguardanti lo studio.

Il sistema mantenuto durante il progetto CONV comprende 13.528 COBOL LOC distribuiti in 675 moduli totali, espressi come somma di paragrafi cobol, sezioni e routine esterne, e fa uso di un database DB2 che comprende 69 tabelle, per un totale di 1.277 campi. Il sistema mantenuto durante il Progetto FR comprende 155.945 LOC COBOL distribuiti in 3.000 moduli totali, tra i quali 231 sono moduli riutilizzati. Inoltre il sistema fa uso di un database SQL che comprende 95 tabelle, per un totale di 1.411 campi.

Il progetto CONV comprendeva 76 richieste di manutenzione, e più precisamente 26 sono state soddisfatte nel primo trimestre, 25 nel secondo trimestre e 25 nel terzo. Il progetto FR comprendeva 60 richieste di manutenzione, e più precisamente 18 sono state soddisfatte nel primo trimestre, 21 nel secondo trimestre e 21 nel terzo.

Gli sviluppatori della PMI, coinvolti in entrambi i progetti, possono essere considerati come aventi la stessa esperienza e capacità. Infatti, sono stati raggruppati a caso e assegnati ai due progetti selezionati.

I programmatori assegnati al progetto che coinvolge i processi di manutenzione del full reuse sono stati addestrati e qualificati per l'uso sia dello strumento che del processo di manutenzione di full reuseprima di iniziare l'indagine.

L'indagine è stata organizzata in tre serie sperimentali di tre mesi ciascuna. Sono schematizzati come segue:

Progetto CONV: processo di manutenzione convenzionale

Ra1 Ra2 Ra3

Progetto FR: processo di manutenzione completo di riutilizzo (RAMO)

Rb1 I Rb2 I Rb3

Come affermato in [Juristo], le variabili di risposta sono spesso usate in modo intercambiabile con il concetto di metrica. In questo senso, durante l'esecuzione, le variabili di risposta sono state valutate attraverso le metriche specificate in un piano di misurazione secondo gli obiettivi perseguiti nello studio. Sono state definite seguendo l'approccio GQM [Basili94]. Il modello di qualità utilizzato è rappresentato nella tabella 7.1. La qualità complessiva del sistema è stata valutata attraverso parametri di produttività, manutenzione e qualità del prodotto.

Tutte le metriche sono state raccolte su base trimestrale. Dopo ogni esecuzione le metriche relative a entrambi i progetti sono state confrontate al fine di valutare gli effetti del full reuserispetto al processo di manutenzione convenzionale utilizzato nel PMI per quanto riguarda il degrado della qualità del software

Object Maintenance process	**Purpose** Evaluate	**Quality Factor:** overall system quality	**Viewpoint** Developer	**Context** full reuse & conv maint.
Primary Quality Model: **Characterization Metrics** **Q1**: What is the developer's experience? M_1: Knowledge of the maintenance process M_2: Experience with programming language **Process Metrics** **Q2**: What is the productivity of the maintenance process? M_3: added LOC (total LOCs belonging to the specific modules of (S) that are not present in (S_{before}) and are contained in (S_{after}) M_4: deleted LOC (total LOCs belonging to the specific modules of (S) that are present in (S_{before}) and no longer included in (S_{after}); M_5: reused LOC (total number of LOCs added and deleted with refer to all the reused modules of (S) M_6: actual modified LOC ($M_6 = M_3 + M_4$) M_7: apparent modified LOC ($M_7 = M_3 + M_4 + M_5$) M_8: effort spent in person hrs M_9: Apparent Productivity ($M_9 = M_7 / M_8$) M_{10}: Actual Productivity ($M_{10} = M_6 / M_8$) M_{11}: Apparent Maintenance Effort M_{12}: Actual Maintenance Effort **Product Metrics** **Q3**: What is the quality of the software system? M_{13}: Total number of modules (M) M_{14}: Cyclomatic Complexity of each module (C_i) M_{15}: Integration Cyclomatic Complexity (ICC) of the system $ICC = (\sum_{i=1}^{M} Ci) - M + 1$ M_{16}: Total number of tables in the databases M_{17}: Database Structure Hiding (number of modules that know the structure of the tables and access them)				
N.B. S=the maintained system S_{before} = the version of the system before being maintained S_{after} = the version of the system after being maintained				

Tabella 7.1 GQM per il caso studio

Essendo i progetti diversi, sono stati utilizzati fattori di normalizzazione per rendere le metriche comparabili e indipendenti dal contesto. L'interpretazione è stata effettuata secondo le linee di base e i fattori di variazione fissati per le metriche nel GQM. Inoltre, le schede di astrazione [Briand] sono state utilizzate come supporto per identificare le azioni di miglioramento (I) che sono state applicate ai soggetti del Progetto FR.

7.3.1 Analisi dei dati

Le metriche sono state suddivise in due classi: la classe delle metriche di processo e le metriche di prodotto, come appare dalla tabella 7.1.

Per ragioni di comparabilità, i valori delle metriche di processo sono tutti normalizzati rispetto al numero totale di LOC prodotte durante la corsa sperimentale considerata.

Per indagare se il processo di manutenzione del full reuse, piuttosto che quello convenzionale, rallenta il degrado della qualità del sistema, si è analizzato [Baldassarre03] i valori delle metriche in entrambi i progetti relativi alla produttività, allo sforzo di manutenzione e alla qualità del prodotto. Inoltre, le ipotesi nulle e alternative sono state formulate come segue per le metriche interessate Mi del piano GQM:

Hi0: Non c'è differenza tra i valori della metrica Mi nel processo di manutenzione convenzionale (cioè, nel progetto CONV) e il processo di manutenzione del full reuse(cioè, nel progetto FR).

Hia: C'è una differenza tra i valori della metrica Mi nel processo di manutenzione convenzionale e del full reuse.

Poiché i due gruppi di osservazioni da confrontare sono indipendenti l'uno dall'altro, avremmo potuto usare il test t per campioni indipendenti. Tuttavia, poiché l'ipotesi di normalità non è stata sempre rispettata, si è deciso di utilizzare il test U di Mann-Whitney. Questa alternativa non parametrica può essere applicata quando ci sono due campioni da popolazioni possibilmente diverse con le seguenti ipotesi [Conover]:

- entrambi i campioni sono campioni casuali delle rispettive popolazioni;
- oltre all'indipendenza all'interno di ogni campione, esiste un'indipendenza reciproca tra i due campioni;
- la scala di misurazione è almeno ordinale.

Nel caso in esame [Baldassarre], tutte e tre le ipotesi precedenti sono state soddisfatte.

Poiché il valore di ogni metrica Mi in entrambi i progetti è stato osservato 3 volte, cioè un'osservazione per ogni esecuzione sperimentale, è necessario verificare che non esista un effetto di interazione tra trattamento e variabile indipendente. In altre parole, è necessario verificare se le eventuali differenze tra i valori delle metriche nei due progetti siano dovute solo allo specifico processo di manutenzione, e non ad altre ragioni, come ad esempio l'effetto dell'apprendimento. A questo scopo, è stato eseguito il test di Wilcoxon, che può "essere usato in situazioni in cui le osservazioni sono accoppiate, per vedere se la seconda variabile casuale nella coppia ha la stessa mediana della prima" [Conover]. I presupposti per applicare questo test sono i seguenti:

- la distribuzione di ogni osservazione è simmetrica
- le osservazioni sono reciprocamente indipendenti
- le osservazioni hanno tutte la stessa mediana
- la scala di misura delle osservazioni è almeno un intervallo

Nel caso in esame [Baldassarre03], tutte e quattro le ipotesi precedenti sono state soddisfatte.

7.3.1.1 Metriche di processo

Le seguenti metriche di processo sono state raccolte e analizzate per valutare gli effetti della manutenzione del riutilizzo completo sulla qualità complessiva del sistema, rispetto a un processo convenzionale.

Produttività apparente

La produttività apparente è una metrica indiretta che rappresenta la quantità di codice modificato in una persona a seguito di una richiesta di manutenzione. Per codice modificato intendiamo LOC aggiunte e cancellate e LOC riutilizzate, cioè la porzione di codice che diventa parte del sistema mantenuto in seguito al recupero dei moduli dal repository. Questa misura riflette la situazione in cui il caso di studio è stato eseguito.

La figura 7.7 mostra l'andamento nei 3 trimestri della produttività apparente durante le attività di manutenzione, per entrambi i progetti CONV e FR. Dalla figura emergono le seguenti considerazioni: in primo luogo, c'è una maggiore quantità di produttività

apparente nel processo di riuso completo piuttosto che in quello convenzionale. Questo è dovuto al fatto che il riuso avviene sistematicamente nel primo processo grazie al repository di moduli riutilizzabili e perché fa parte del processo di manutenzione stesso. Mentre nel processo convenzionale l'unico riutilizzo che avviene è fortuito. In secondo luogo, in entrambi i casi c'è una diminuzione della produttività apparente durante i tre trimestri. Questa è una tendenza attesa in quanto è noto che gli interventi continui hanno un impatto sulla qualità complessiva del sistema, rendendo la manutenzione più onerosa e lunga. Tuttavia, sottolinea che c'è più degrado nel periodo osservato nel processo convenzionale piuttosto che in quello di riutilizzo totale. Ancora una volta, questo può essere attribuito all'effetto del riutilizzo. In altre parole, il riutilizzo sistematico dei moduli nel processo di riutilizzo completo assicura che la qualità del sistema sia preservata. Questo perché il riutilizzo di un modulo coinvolge tutti gli artefatti prodotti e la documentazione e non è limitato solo al codice. Quindi i moduli riutilizzabili sono costantemente testati e aggiornati.

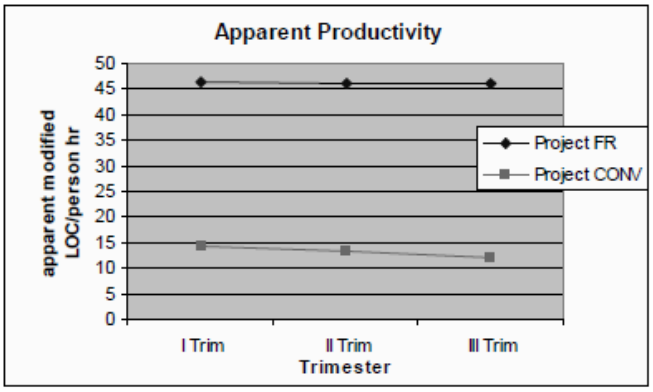

Figura 7.7 Produttività apparente della manutenzione

La tabella 7.2 riporta i valori del livello p ottenuti dal test di Wilcoxon applicato alla produttività apparente per l'esecuzione delle attività di manutenzione.

Project	Trimester	vs. Trimester	p-level
CONV	Trimester 1	Trimester 2	0,637736
	Trimester 2	Trimester 3	0,057845
FR	Trimester 1	Trimester 2	0,198894
	Trimester 2	Trimester 3	0,159234

Tabella 7.2 Valori di p-level del test di Wilcoxon applicato alla produttività apparente

Poiché in tutti i casi il valore di p-level è superiore alla soglia convenzionale del 5%, allora non esiste una differenza significativa tra le mediane di due osservazioni successive.

La tabella 7.3 riporta i valori del livello p ottenuti dal test Mann-Whitney U, applicato alla produttività apparente per l'esecuzione delle attività di manutenzione.

Trimester	p-level
Trimester 1	0,004855
Trimester 2	0,009880
Trimester 3	0,000001

Tabella 7.3 Valori di p-level del test Mann- Whitney U applicato alla produttività apparente

Il test rivela una differenza significativa tra i due gruppi rispetto al valore della produttività apparente nei due progetti.

Infine, è stato considerato un valore complessivo dell'aumento di produttività, rispetto all'intero periodo di osservazione, grazie al riutilizzo. È stato calcolato come la differenza della produttività globale apparente nel processo di riutilizzo completo e quello convenzionale, diviso per quest'ultimo. Si è ottenuto un aumento del 248%.

Produttività effettiva

La produttività effettiva è intesa come la quantità di codice modificato per ora persona, a seguito di una richiesta di manutenzione, senza contare il riutilizzo. Pertanto, in questo caso il codice modificato è inteso come la quantità totale di LOC aggiunte e cancellate. Questa misura ci dà un'idea della produttività senza considerare il riutilizzo.

La figura 7.8 mostra l'andamento nei 3 trimestri della produttività effettiva per l'esecuzione della manutenzione, sia per il progetto CONV che per FR. Dalla figura emergono le seguenti considerazioni: in primo luogo, l'andamento nei tre trimestri è simile per entrambi i progetti, questo era atteso, considerando che la quantità di riutilizzo è stata esclusa dall'analisi. La somiglianza può essere attribuita al fatto che i programmatori assegnati ai progetti avevano la stessa esperienza di programmazione, capacità e conoscenza dei domini applicativi. Questo è probabilmente il motivo per cui i valori sono praticamente gli stessi nel primo trimestre (13,35 LOCs/hr) nel progetto FR contro (13,15) nel progetto CONV. In secondo luogo, la produttività diminuisce più velocemente durante i tre trimestri rispetto al processo convenzionale piuttosto che a quello di riutilizzo completo. La tendenza che emerge dal grafico indica che il sistema coinvolto nel Progetto FR è più facile da mantenere. Questo può essere, ancora una volta, legato agli effetti del processo di riuso completo in quanto garantisce che i moduli nel repository siano costantemente testati e aggiornati, e quindi di qualità superiore. Di conseguenza, quando si soddisfa ogni richiesta di manutenzione i programmatori implicano meno sforzo. La maggiore qualità dei moduli, e quindi della documentazione e del codice, assicura che la manutenzione abbia meno effetti negativi sul degrado del

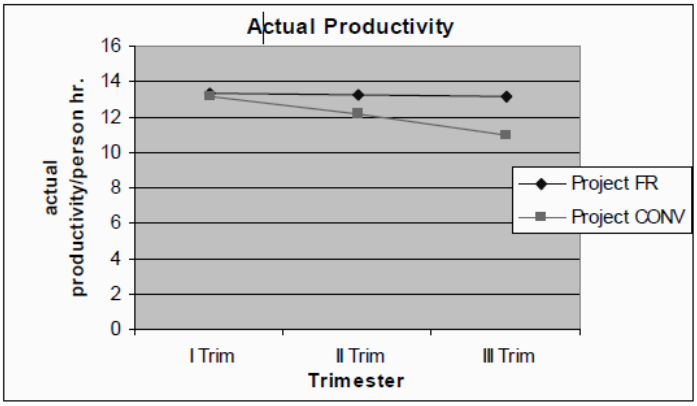

Figura 7.8 Produttività effettiva della manutenzione

La tabella 7.4 riporta i valori del livello p ottenuti dal test di Wilcoxon applicato alla produttività effettiva per l'esecuzione delle attività di manutenzione.

Project	Trimester	vs Trimester	p-level
CONV	Trimester 1	Trimester 2	0,581232
	Trimester 2	Trimester 3	0,087537
FR	Trimester 1	Trimester 2	0,280539
	Trimester 2	Trimester 3	0,527731

Tabella 7.4 Valori di p-level del test di Wilcoxon applicato alla produttività effettiva

Poiché in tutti i casi il valore di p-level è superiore alla soglia convenzionale del 5%, allora non esiste una differenza significativa tra le mediane di due osservazioni successive.

La tabella 7.5 riporta i valori del livello p ottenuti dal test Mann-Whitney U, applicato alla produttività effettiva per l'esecuzione delle attività di manutenzione.

Trimester	p-level
Trimester 1	0,008650
Trimester 2	0,087
Trimester 3	0,3659

Tabella 7.5 Valori di p-level del test Mann-Whitney U applicato alla produttività effettiva

Il test rivela una differenza significativa tra i due gruppi rispetto al valore della produttività effettiva nei due progetti solo nel primo trimestre. Differenze non significative si ottengono negli altri trimestri. Questi risultati erano attesi, in quanto confermano che un aumento della produttività è legato al riutilizzo. Infatti, in questo caso i moduli nel repository sono costantemente controllati per quanto riguarda la loro qualità, con conseguenti impatti sulla qualità complessiva del sistema.

Sforzo di manutenzione apparente e reale

La figura 7.9 mostra l'andamento nei 3 trimestri dello sforzo di manutenzione speso per l'esecuzione di tutte le attività di manutenzione utilizzando il Full Reuse Process. Questa metrica è stata raccolta solo per il progetto FR. In particolare, il suo scopo era quello di evidenziare le differenze nello sforzo di manutenzione se non fosse stato applicato il riuso totale. Per comprendere il grafico sono necessarie le seguenti definizioni:

- per *sforzo* di *manutenzione apparente* intendiamo lo sforzo speso dai programmatori del progetto FR per eseguire tutte le richieste di manutenzione (MR) durante i tre trimestri. In questa misura si tiene conto degli effetti del riutilizzo. Durante i tre trimestri una stessa MR può essere stata fatta varie volte. Pertanto, lo sforzo di manutenzione apparente per ogni MR è contato come la somma dello sforzo necessario per eseguire ogni MR. Ovviamente, la prima volta che la richiesta è stata soddisfatta, ha implicato uno sforzo maggiore. Mentre, in tutte le volte successive lo sforzo è stato minore in quanto ha beneficiato del riutilizzo di moduli precedenti. Così per esempio MR1 è stato fatto due volte durante i tre trimestri e ha richiesto uno sforzo di 23 ore la prima volta, e solo 2 ore la seconda. Quindi lo sforzo apparente di manutenzione è stato di 25 ore.
- per *sforzo di manutenzione effettivo,* intendiamo lo sforzo che sarebbe stato implicito nell'esecuzione di ogni MR se il riutilizzo non fosse stato adottato. Senza il riutilizzo, ogni MR deve essere esaminato da zero come se fosse la prima volta che viene realizzato. In seguito a queste considerazioni è stato calcolato come lo sforzo per realizzare il MR la prima volta, moltiplicato per le volte che il MR è stato realizzato per tutti i trimestri. Così, per esempio, nel caso di MR1, lo sforzo effettivo di manutenzione è stato di 46 ore (23 ore*2MR).

Era insensato misurare questa metrica anche nel progetto CONV, in quanto lo sforzo di manutenzione apparente e quello effettivo erano gli stessi. In questo caso infatti il MR non ha beneficiato di nessun tipo di riutilizzo se non quello fortuito.

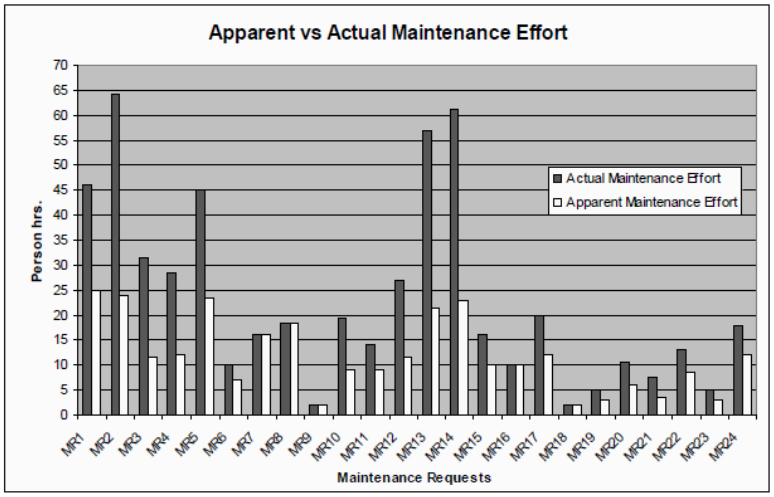

Figura 7.9 Sforzo di manutenzione apparente e reale nel progetto FR

Come si può vedere dall'istogramma, il riutilizzo permette di risparmiare una grande quantità di sforzo per eseguire le attività di manutenzione durante i trimestri. I casi in cui lo sforzo apparente e quello effettivo sono uguali sono quelli in cui c'è stata una sola richiesta di MR.

7.3.1.2 Metriche di prodotto

I risultati positivi ottenuti rispetto alle metriche di processo sono stati ulteriormente indagati attraverso le metriche di prodotto al fine di verificare i benefici del riutilizzo completo sulla riduzione del degrado della qualità. Le metriche di prodotto considerate sono riportate nella tabella 1. I risultati sono descritti di seguito.

Complessità ciclomatica di integrazione (ICC)

Questa misura esprime la complessità del sistema. Come si sa, più un sistema è complesso, più è difficile da mantenere. Pertanto, ICC è un indicatore della qualità complessiva del sistema.

La misura è stata raccolta utilizzando la formula riportata nella tabella 7.1. È stata calcolata su tutti e tre i trimestri per monitorare il progressivo degrado del sistema. L'andamento della complessità ciclomatica di integrazione è mostrato nella figura 7.10.

Come si può vedere dal grafico, la complessità del sistema aumenta più nel progetto CONV che in FR. Questo può essere interpretato come un sintomo di degradazione del sistema. Pertanto, il sistema soggetto al processo di manutenzione convenzionale è caratterizzato da un degrado maggiore rispetto a quello soggetto alla manutenzione di riuso totale. Una minore complessità in tutto il periodo di osservazione in Project FR è attribuibile ai moduli riutilizzabili (codice e documentazione) memorizzati nel repository e costantemente aggiornati.

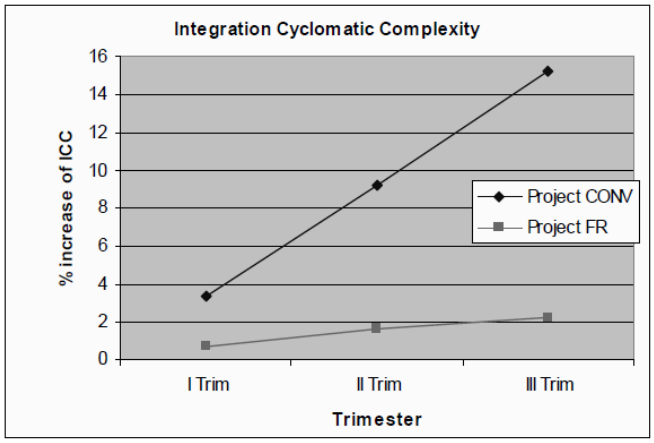

Figura 7.10 Complessità ciclomatica dell'integrazione

La tabella 7.6 riporta i valori del p-level ottenuti dal test di Wilcoxon applicato alla ICC per l'esecuzione delle attività di manutenzione.

Project	Trimester	vs. Trimester	p-level
CONV	Start	Trimester 1	0,000008
	Trimester 1	Trimester 2	0,000010
	Trimester 2	Trimester 3	0,000000
FR	Start	Trimester 1	0,105655
	Trimester 1	Trimester 2	0,007136
	Trimester 2	Trimester 3	0,000102

Tabella 7.6 Valori di p-level del test di Wilcoxon applicato a ICC

Come si può vedere dalla tabella, per quanto riguarda il Progetto CONV, il valore di p-level è inferiore alla soglia convenzionale del 5%, quindi esiste una differenza significativa tra le mediane di due osservazioni successive. Questo riflette le nostre aspettative in quanto è plausibile aspettarsi che nel processo convenzionale la complessità complessiva del sistema aumenti a causa dei continui interventi di manutenzione. La differenza significativa da un trimestre all'altro è un sintomo che il sistema si sta degradando.

Per quanto riguarda il Progetto FR, non ci sono differenze significative attraverso i trimestri tranne che per il secondo e il terzo. Questo è un risultato atteso in quanto il riutilizzo completo "rallenta" ma non "evita" il degrado del sistema, quindi una diminuzione della qualità era in qualche modo prevista.

Nascondere la struttura del database

Questa misura rappresenta quanti moduli conoscono la struttura delle tabelle e vi accedono. I dettagli sono rappresentati nella figura 7.11.

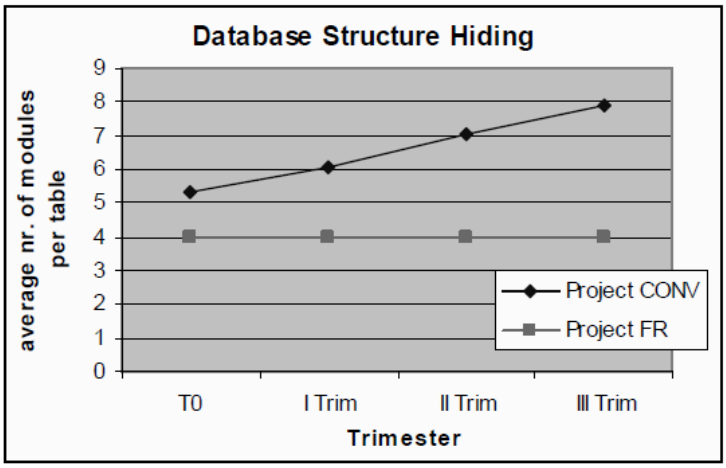

Figura 7.11 Nascondare la struttura del database

In caso di riutilizzo completo, ogni tabella è accessibile da quattro moduli a causa del fatto che lo strumento RAPID costruito per supportare il sistema, accede ad ogni tabella attraverso quattro moduli noti come CRUD. In altre parole, un modulo per ciascuna delle operazioni eseguite: creare, leggere, aggiornare, cancellare. Questo valore dipende solo dal numero di tabelle che compongono il sistema ed è quindi lo stesso per tutti e tre i trimestri. Per la manutenzione convenzionale il numero di moduli per ogni tabella dipende dalle decisioni di progettazione prese dagli sviluppatori durante lo sviluppo del sistema. La tendenza aumenta durante il periodo di osservazione. In altre parole, il numero medio di moduli che accedono ad ogni tabella aumenta da un trimestre all'altro.

La tabella 7.7 riporta i valori del livello p ottenuti dal test di Wilcoxon applicato alla struttura del database nascosto per l'esecuzione delle attività di manutenzione. L'analisi statistica è stata effettuata solo per il progetto CONV. Era insensato considerarla per il FR essendo il valore medio sempre quattro.

Project	Trimester	Vs. Trimester	p-level
CONV	Start	Trimester 1	0,000438
	Trimester 1	Trimester 2	0,000040
	Trimester 2	Trimester 3	0,000197

Tabella7.7 valori di p-level del test di Wilcoxon applicato al nascondiglio della struttura del database

I risultati mostrano che in tutti i casi il valore di p- level è inferiore alla soglia convenzionale del 5%, quindi esiste una differenza significativa tra le mediane di due osservazioni successive. Questo risultato conferma il fatto che un aumento continuo del numero di moduli che accedono ad ogni tabella del database è un sintomo di degradazione della qualità del sistema.

7.4 Modello di sviluppo orientato al riutilizzo (ROD)

Il riutilizzo avviene ogni volta che un artefatto esistente viene utilizzato per lo sviluppo o la manutenzione del prodotto finale. La caratteristica principale dello sviluppo orientato al riutilizzo proposto è che utilizza un repository comune per memorizzare, cercare e recuperare artefatti software esistenti da riutilizzare per sviluppare nuovi requisiti software. I passi del modello possono essere riassunti come segue: identificazione degli artefatti software da un repository comune; analisi degli artefatti per vedere se sono appropriati per le esigenze specifiche del contesto in cui devono essere applicati; adattamento dei componenti al contesto; integrazione. Così, nella definizione di un nuovo artefatto, quelli esistenti recuperati dal repository possono corrispondere parzialmente o totalmente alla specifica del problema. Nel caso specifico di ROD:

- prima di sviluppare un artefatto software da zero, il repository viene cercato per i componenti che sono appropriati per gli scopi della specifica del problema. Tutti gli artefatti nel repository sono caratterizzati da una descrizione, parametri di input e di output. Tali informazioni sono necessarie per recuperare artefatti riutilizzabili;
- il risultato di una ricerca può essere: "total match" quando l'artefatto identificato nel repository corrisponde alla richiesta. Quindi l'artefatto viene riutilizzato interamente; "partial match" in questo caso l'artefatto recuperato dal repository corrisponde alla richiesta solo in parte. Quindi, deve essere adattato alle caratteristiche specifiche del requisito da sviluppare. In questo caso il riutilizzo è solo parziale perché parte dell'artefatto deve essere sviluppato; "nessuna corrispondenza" significa che nessun artefatto nel repository è una soluzione appropriata alla specifica del problema dato. In questo caso l'artefatto deve essere interamente sviluppato da zero.

Seguendo queste considerazioni e adottandole come linee guida, abbiamo definito un processo di sviluppo orientato al riuso chiamato "RApid software Development prOcess" (RADO) che utilizza i moduli del repository per sviluppare parti del sistema

software. Quindi RADO è un'istanza di un modello ROD. Da qui in poi RADO e ROD saranno usati in modo equivalente.

7.5 RApid software Development prOcess (RADO)

Il modello orientato al riuso, presentato nel paragrafo precedente, usa *modelli* per sviluppare un artefatto software. Questo evita di dover sviluppare codice da zero e porta all'adozione di documentazione e standard di codifica attraverso i "template". Nel nostro contesto un template è uno scheletro di un artefatto, descritto a diversi livelli di astrazione. È memorizzato nel repository ed è composto da una parte fissa, indipendente dall'applicazione, e una parte che deve essere definita dallo sviluppatore e specializzata al prodotto software finale in cui è inclusa. Quindi, lo sforzo richiesto consiste nello specializzare il contenuto del template al contesto specifico. Più precisamente, distinguiamo tra: *template di architettura*: specificano un artefatto da un punto di vista architetturale; *template di documentazione*: specificano la documentazione relativa ad un artefatto; *template di casi di test*: per definire l'insieme dei casi di test per l'artefatto.

Le caratteristiche principali del RADO sono rappresentate nella figura 7.12.

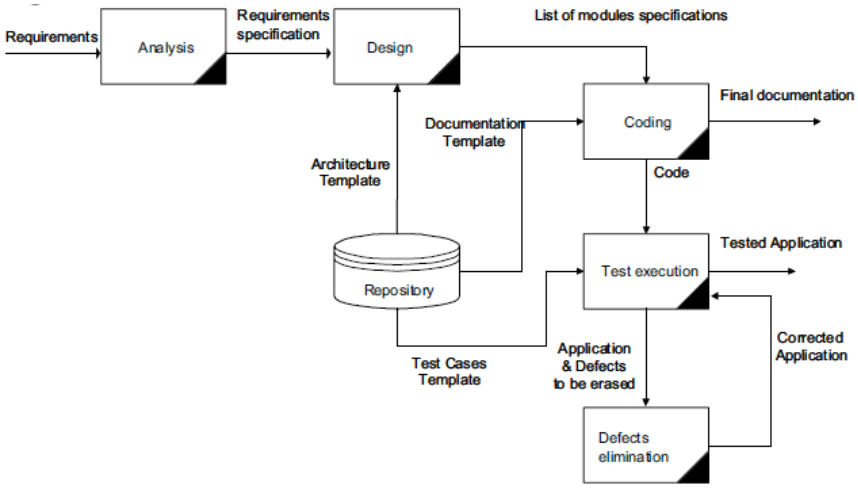

Figura 7.12: RADO: RApid software Development prOcess

Può essere riassunto nei seguenti passi: vengono specificati i requisiti software del sistema da sviluppare; le specifiche del problema ottenute vengono utilizzate nella fase di progettazione per cercare possibili artefatti recuperabili da riutilizzare o, in caso di assenza, per istanziare l'apposito modello di architettura recuperato dal repository; l'output di tale fase è una lista di specifiche di artefatti; durante la fase successiva viene effettuata la codifica facendo uso sia delle specifiche fornite dalla

fase precedente sia dell'apposito modello di documentazione, recuperato dal repository; se è stata trovata una corrispondenza totale nella fase precedente, la documentazione è già definita, così come il suo codice. L'output di tale fase è la documentazione finale del sistema e il suo codice; il codice ottenuto viene poi testato durante la fase successiva, che fa uso del modello dei casi di test. Inoltre, in caso di corrispondenza totale, i casi di test sono già impostati, quindi devono essere solo eseguiti e non progettati. Altrimenti devono essere adattati al contesto, in caso di corrispondenza parziale, o completamente definiti a partire da un modello in caso di mancata corrispondenza. I possibili difetti vengono rimossi durante l'ultima fase.

Secondo [5], il nostro obiettivo di ricerca è stato formalizzato come segue: *Analizzare* il processo ROD, *Allo scopo di* confrontare, *Rispetto alla* qualità generale del sistema, *Dal punto di vista dell'*ingegnere del software, *Nel contesto dello* sviluppo del software con e senza riuso.

7.6 Progetto CONV e Progetto ROD

La qualità complessiva del sistema è considerata sia da una prospettiva di prodotto che di processo. Dato questo obiettivo, si è definito [Baldassarre05] le seguenti domande di ricerca:

RQ1: Che impatto ha il ROD sulla qualità del prodotto?

RQ2: Che impatto ha la ROD sulla qualità del processo?

RQ3: Che impatto ha la ROD sulla FRM?

Per RQ1 e RQ2 è stato effettuato il test delle ipotesi, mentre per RQ3 si fanno considerazioni in base ai risultati ottenuti in [Baldassarre03].

L'indagine sul campo consisteva nell'applicare il processo proposto a progetti industriali. Più precisamente, sono stati scelti due progetti in corso, tipici di una PMI italiana. I due progetti sono identificati come Progetto CONV e Progetto ROD, e si riferiscono ad un processo di sviluppo convenzionale (waterfall) e orientato al riuso adottato. Il primo progetto consisteva in un'applicazione per la conversione automatica della valuta. Il secondo era un sistema software per la gestione di procedure bancarie come il recupero del credito.

È chiaro che una situazione ideale sarebbe stata quella di applicare entrambi i processi (CONV e ROD) a un solo progetto. Questo avrebbe permesso un maggior livello di controllo su tutte le variabili durante lo studio. D'altra parte, questo avrebbe

avuto un impatto sui costi e sull'impegno dell'indagine perché l'impresa avrebbe dovuto investire il doppio delle risorse sullo stesso progetto sviluppato con due processi diversi. Tale situazione è abbastanza irrealistica, soprattutto in una PMI dove le risorse economiche e umane sono limitate e i vincoli di bilancio determinano la necessità di innovazione, compatibilmente con le esigenze di produzione. Questo ha determinato la preferenza per indagare l'innovazione utilizzando progetti di produzione ordinari piuttosto che un progetto sperimentale controllato ad hoc. Per le stesse ragioni, sono state utilizzate metriche familiari e valutate nell'impresa per evitare di essere invadenti rispetto ai processi di supporto alla valutazione della qualità della PMI. In ogni caso, nonostante queste limitazioni, si è del parere che i risultati ottenuti possano essere interessanti per la comunità scientifica e industriale.

Inoltre, questo studio di caso è stato organizzato come una strategia di ricerca comparativa, cioè confrontando i risultati dell'uso di un metodo di sviluppo (ROD) con quelli dell'uso dell'altro approccio (CONV). Al fine di evitare pregiudizi e assicurare la validità interna, abbiamo organizzato lo studio su una solida base secondo [Fenton]. I due progetti sono simili nel linguaggio di implementazione (COBOL), nelle specifiche e nella tecnica di progettazione.

I soggetti coinvolti nel caso di studio sono stati selezionati tra gli sviluppatori della PMI e assegnati casualmente ad ogni progetto. Questo è stato possibile perché l'indagine è stata pianificata prima che i progetti iniziassero effettivamente. I programmatori assegnati al progetto ROD sono stati addestrati ed esperti nell'uso dello strumento RAPID e del processo ROD prima di iniziare l'indagine. I soggetti assegnati allo sviluppo convenzionale erano già abili in quanto avevano usato il processo nello stesso numero di progetti precedentemente svolti nella PMI. In questo caso, non era necessaria alcuna formazione.

7.6.1 Disegno del caso di studio

L'indagine è stata organizzata in tre serie sperimentali di tre mesi ciascuna. Il disegno è rappresentato come segue:

TR_{x1} O_1 TR_{x2} O_2 TR_{x3} O_3

Dove x={progettoCONV O progettoROD} e y={i-esimo trimestre di osservazione | i=1...3}. Ogni TR_{xy} contiene l'insieme delle specifiche dei requisiti che sono state sviluppate nel trimestre. Essendo progetti in corso, le specifiche sviluppate non possono essere pianificate e decise a priori, piuttosto, corrispondono ai requisiti

effettivi che sono stati richiesti durante ogni progetto. Ogni O_y corrisponde all'osservazione delle variabili dopo un trimestre di lavoro, cioè tutte le metriche raccolte ogni trimestre su entrambi i progetti.

Inoltre, dopo ogni esecuzione le metriche relative a entrambi i progetti sono state confrontate (CONV e ROD) al fine di valutare gli effetti del riutilizzo orientato rispetto al processo di sviluppo convenzionale utilizzato nella PMI per quanto riguarda la qualità del software.

7.6.2 Esecuzione

Durante l'esecuzione, le variabili di risposta sono state valutate attraverso metriche [Fenton, Juristo] specificate in un piano di misurazione secondo l'obiettivo perseguito nello studio. Tali metriche sono state definite secondo il GQM approccio [5, 6]. Un frammento del modello di qualità utilizzato è rappresentato nella tabella 7.8.

Essendo l'indagine un caso di studio condotto in un contesto industriale su progetti in corso, siamo stati inevitabilmente influenzati nella scelta delle metriche da raccogliere. Le metriche scelte sono quelle familiari al PMI, raccolte sistematicamente e che meglio rappresentano l'attributo di indagine della "qualità del sistema software". È chiaro che l'introduzione, da parte [Baldassarre05], di altre metriche avrebbe inevitabilmente introdotto un overhead nell'esecuzione del processo e rischiato di falsare i risultati. Date le metriche disponibili, la qualità complessiva del sistema è stata considerata da due prospettive: processo e prodotto. La qualità del processo è stata valutata attraverso la produttività. La qualità del prodotto è stata misurata attraverso la complessità del sistema.

Object	Purpose	Quality Factor:	Viewpoint	Context
ROD Model	Evaluating its impact	overall system quality	Software Engineer	Software development.

Primary Quality Model:

Process Metrics

Q1: What is the productivity of the ROD process?

M_1: written LOC (total LOCs that are written when developing the requirement)

M_2: reused LOC (total number of LOCs that belong to previously developed artifacts and have been retrieved from the repository during development of the software requirement)

M_3: effort spent in person hrs

M_4: Apparent Productivity: $M_4 = (M_1 + M_2)/ M_3$

M_5: Actual Productivity: $M_5 = M_1 / M_3$

M_6: Total LOC (per requirement of the ROD software system)

M_7: Reuse Rate: $M_7 = M_2 / M_6$

Product Metrics

Q2: What is the quality of the software system developed with ROD?

M_8: Total number of artifacts (M^{ROD})

M_9: Cyclomatic Complexity of each artifact (C_i^{ROD})

M_{10}: Mean Cyclomatic Complexity (MCC^{ROD}) of the system

$$MCC^{ROD} = (\sum_{i=1}^{M^{ROD}} C_i^{ROD}) / M^{ROD}$$

Confirming Quality Model:

Process Metrics

Q3: What is the productivity of the conventional (CONV) development process?

M_{11}: written LOC (total LOCs that are written when developing the requirement)

M_{12}: reused LOC (total number of LOCs that belong to previously developed artifacts and have been used during development of the new software requirement)

M_{13}: effort spent in person hrs

M_{14}: Apparent Productivity: $M_{14} = (M_{11} + M_{12})/ M_{13}$

M_{15}: Actual Productivity: $M_{15} = M_{11} / M_{13}$

M_{16}: Total LOC (per requirement of the CONV software system)

Product Metrics

Q4: What is the quality of the software system developed with CONV?

M_{17}: Total number of artifacts (M^{CONV})

M_{18}Cyclomatic Complexity of each artifact (C_i^{CONV})

M_{19}: Mean Cyclomatic Complexity (MCC^{CONV}) of the system

$$MCC^{CONV} = (\sum_{i=1}^{M^{CONV}} C_i^{CONV}) / M^{CONV}$$

Tabella 7.8 GQM per il caso studio

Il piano di misurazione è strutturato in due parti: modello di qualità primario e modello di qualità di conferma. Nella prima, si misurano le metriche relative al modello ROD. La seconda si riferisce alle metriche del processo di sviluppo CONV. Queste metriche sono utilizzate per confrontare e confermare i valori delle metriche del modello di qualità primario. I valori di metrica raccolti sono stati interpretati secondo il linee di base fissate per le metriche nel GQM.

7.6.3 Analisi dei dati

Per capire gli impatti del modello ROD sul ciclo di vita del software, abbiamo considerato sia le metriche di processo che quelle di prodotto. A questo scopo, si è analizzato e confrontato i valori delle metriche in entrambi i progetti. Considerando l'obiettivo della ricerca, le ipotesi sono state formulate come segue per le metriche rilevanti Mi del piano GQM:

Hi0: Non c'è differenza tra i valori della metrica Mi nel processo CONV e nel processo ROD.

Hia: C'è una differenza tra i valori della metrica Mi nel processo convenzionale e ROD.

Queste ipotesi sono state formulate e testate per RQ1 e RQ2. Mentre per la RQ3 si riportano considerazioni, secondo i risultati precedentemente valutati in [Baldassarre03] su come lo sviluppo orientato al riuso impatti sulla successiva manutenzione di un sistema software

Poiché i due gruppi di osservazioni da confrontare sono indipendenti l'uno dall'altro, avremmo potuto usare il test t per campioni indipendenti. Tuttavia, poiché l'ipotesi di normalità non è stata sempre rispettata, si è deciso di utilizzare l'alternativa non parametrica [Conover] del test Mann- Whitney U.

Poiché il valore di ogni metrica Mi in entrambi i progetti è stato osservato 3 volte (cioè un'osservazione per ogni esecuzione sperimentale) è necessario verificare che non esista un effetto di interazione tra trattamento e variabile indipendente. In altre parole, è necessario verificare se le eventuali differenze tra i valori delle metriche in ciascuno dei due progetti siano dovute solo allo specifico processo di manutenzione, e non ad altre ragioni, come ad esempio l'effetto maturazione. A questo scopo, è stato scelto il test di Wilcoxon per gruppi di campioni dipendenti [Conover]. Per la verifica delle ipotesi, sia nei test per gruppi dipendenti che in quelli per gruppi indipendenti, il valore Ļ è stato fissato al 5%. Una rappresentazione grafica dei test effettuati durante l'intera indagine è rappresentata nella figura 7.13.

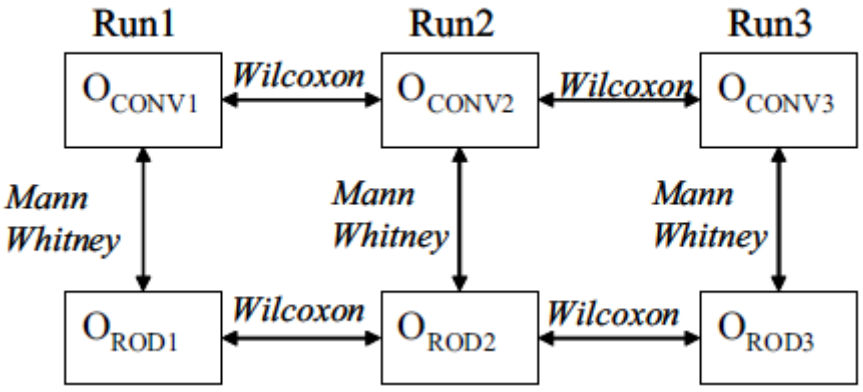

Figura7.13 Schema del test statistico

L'analisi dei dati include: statistiche descrittive per rappresentare graficamente i dati di ogni progetto lungo i tre trimestri di osservazione; risultati del test delle ipotesi

cioè il livello p del test Mann-Whitney U, che indica se ci sono differenze significative nelle osservazioni tra i due trattamenti (gruppi campione indipendenti); il livello p del test Wilcoxon, che indica se ci sono differenze significative nelle osservazioni in due trimestri successivi dello stesso trattamento (gruppi campione dipendenti). Questo è un indicatore dell'effetto di maturazione.

7.6.3.1 Metriche di prodotto

Per rispondere alla RQ1, durante l'esecuzione sono state raccolte metriche di prodotto per entrambi i progetti. Queste sono state analizzate su base trimestrale e miravano a misurare la qualità del software sviluppato. In particolare, la qualità del prodotto è stata valutata attraverso la complessità ciclomatica McCabe di ogni modulo [Watson].

Complessità ciclomatica media (MCC)

Questa misura esprime la complessità del sistema. Più un sistema è complesso, più è difficile capire e riutilizzare i suoi artefatti. Perciò MCC è usato come un indicatore della qualità generale del sistema.

La misura è stata raccolta utilizzando la formula riportata nella tabella 1, ottenuta come sommatoria della complessità di ogni modulo diviso il numero totale di moduli sviluppati nello specifico trimestre di osservazione. È stata calcolata lungo ogni trimestre per monitorare la progressiva complessità del sistema, per entrambi i progetti. La figura 7.14 sintetizza la complessità del sistema software per trimestre (Mean Cyclomatic Complexity - MCC).

Come si può vedere dal grafico, durante tutto il periodo di osservazione, la complessità media complessiva del sistema software CONV è sempre più alta che in ROD. Inoltre, tra un trimestre e l'altro la complessità dello sviluppo convenzionale aumenta più rapidamente che per ROD. Questo significa che la qualità del prodotto è più alta nello sviluppo ROD e che, per CONV, il sistema software diventa più complesso nel tempo.

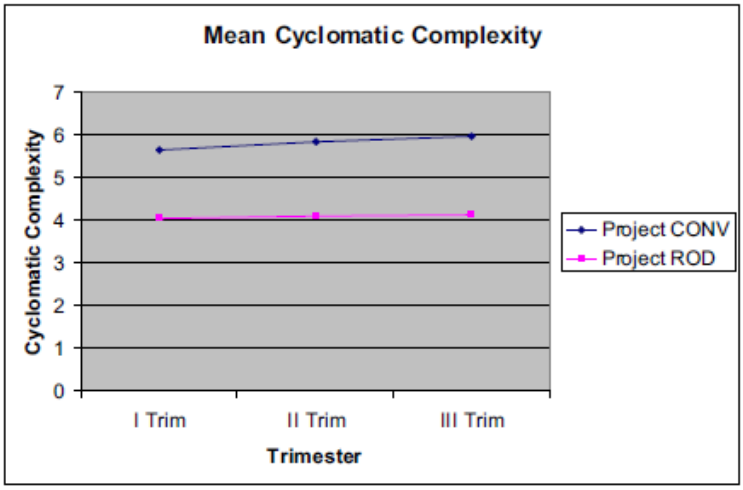

Figura 7.14 Complessità cicolomatica media

I risultati del test di Mann Whitney (tabella 7.9), eseguiti tra i trattamenti hanno evidenziato, come previsto, che le differenze tra i trattamenti sono statisticamente significative.

Trimester	p-level
Trimester 1 (ROD vs. CONV)	0,001
Trimester 2 (ROD vs. CONV)	0,001
Trimester 3 (ROD vs. CONV)	0,001

Tabella7.9 Valori di p-level del test Mann-Whitney applicato a MCC

La tabella 7.10 riporta i valori del livello p ottenuti dal test di Wilcoxon applicato alla MCC durante lo sviluppo di entrambi i sistemi software lungo i tre trimestri. Il test evidenzia che non esistono differenze significative tra le mediane di due osservazioni successive. Quindi, non c'è alcun effetto di maturazione durante il periodo di sperimentazione, quindi i risultati non sono molto probabilmente influenzati da alcun effetto di maturazione, anche se questo non esclude completamente altre possibili minacce di cui non eravamo a conoscenza e che non abbiamo potuto controllare.

Project	Trimester	vs. Trimester	p-level
CONV	Trimester 1	Trimester 2	0,140656
	Trimester 2	Trimester 3	0,456745
ROD	Trimester 1	Trimester 2	0,159410
	Trimester 2	Trimester 3	0,577238

Tabella 7.10 Valori di p-level del test di Wilcoxon applicato a MCC

7.6.3.2 Metriche di processo

Come ulteriore analisi, si è indagato se il processo ROD è più efficiente di uno a cascata convenzionale. In particolare, sono state misurate le metriche di processo per assicurare che la maggiore qualità del prodotto sia dovuta a un processo efficiente.

La metrica di processo considerata per evidenziare i benefici del processo ROD e capire il suo impatto sulla qualità risultante del sistema software è: la produttività. Questa metrica è stata analizzata solo per gli artefatti relativi a a modelli di architettura, ad es. componenti codificati. La metrica è considerata come la quantità di codice prodotto in ore persona. Si distinguono tra produttività apparente e reale come segue:

Produttività apparente

La produttività apparente, come definito in precedenza, è una metrica indiretta che rappresenta la quantità di codice prodotto in ore persona secondo ogni requisito delle specifiche di sviluppo.

La figura 7.15 e la figura 7.16 mostrano l'andamento nei 3 trimestri della produttività apparente durante le attività di sviluppo, sia per il progetto CONV che per il ROD, rappresentati attraverso i box plot.

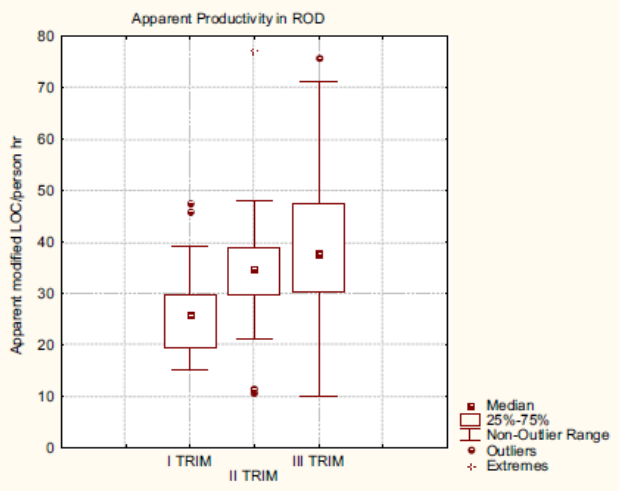

Figura 7.15 Produttività apparente in ROD

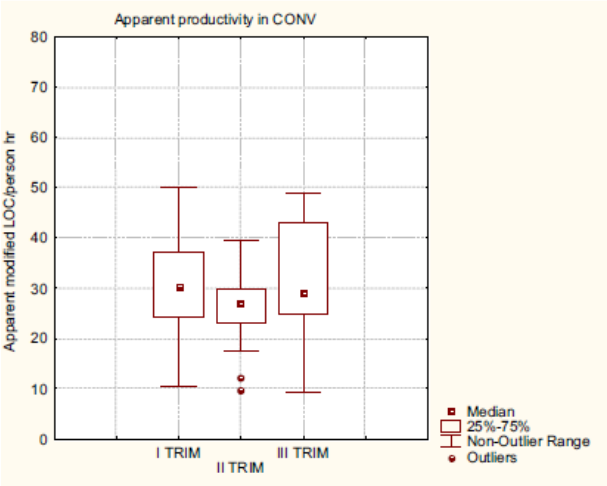

Figura 7.16 Produttività apparente in CONV

Dalle cifre emergono le seguenti considerazioni: nel complesso, c'è una maggiore quantità di produttività apparente nel processo orientato al riutilizzo piuttosto che in quello convenzionale. Questo perché, nel primo caso il riutilizzo avviene sistematicamente ed è una parte integrata del processo. Più precisamente, nel primo trimestre ROD è inferiore a CONV, questo è attribuibile al fatto che il processo di sviluppo era nuovo per gli sviluppatori e che, non essendo il repository ancora popolato di componenti riutilizzabili, il riuso era più difficile da mettere in pratica. Dopo un primo periodo, i benefici del riuso vengono percepiti e questo si traduce in un aumento della produttività del processo ROD, per cui si producono più moduli e si implica un minore sforzo. La tabella 7.11 riporta i risultati del test Mann-Whitney U. Tutti i trimestri evidenziano differenze significative. In particolare, nel primo trimestre CONV è più performante di ROD. Questo era un risultato atteso perché nel primo trimestre il repository non è popolato, il tasso di riutilizzo è basso e gli sviluppatori stanno iniziando a familiarizzare con il processo. Infatti, l'effetto di maturazione nel processo ROD, come sarà mostrato nella tabella 7.12, conferma una differenza significativa nei due trattamenti per il secondo e terzo trimestre.

Trimester	p-level
Trimester 1 (ROD vs. CONV)	0,0392
Trimester 2 (ROD vs. CONV)	0,0052
Trimester 3 (ROD vs. CONV)	0,0216

Tabella 7.11 Valori di p-level del test Mann-Whitney U applicato alla produttività apparente

La tabella 7.12 riporta i valori del livello p ottenuti dal test di Wilcoxon applicato alla produttività apparente per lo sviluppo dei requisiti del sistema.

Project	Trimester	vs. Trimester	p-level
CONV	Trimester 1	Trimester 2	0,1329
	Trimester 2	Trimester 3	0,1476
ROD	Trimester 1	Trimester 2	**0,0298**
	Trimester 2	Trimester 3	0,4548

Tabella7.12 Valori di p-level del test di Wilcoxon applicato alla produttività apparente

Come si può vedere tra il primo e il secondo semestre del processo ROD il livello p è inferiore alla soglia fissata. Questo, come commentato in precedenza, dipende dal fatto che gli sviluppatori hanno imparato ad usare sia il processo che lo strumento per attuare il riuso.

Produttività effettiva

La produttività effettiva è intesa come la quantità di codice sviluppato per ora persona, secondo un requisito, senza contare il riutilizzo. Quindi, in questo caso, per codice sviluppato si intende la quantità totale di LOC scritte. Questa misura ci dà un'idea della produttività senza considerare il riutilizzo.

La figura 7.17 e la figura 7.18 mostrano la tendenza attraverso i 3 trimestri della produttività effettiva per lo sviluppo, in entrambi i progetti. Ancora una volta sono stati usati i box plots per tracciare le serie di dati. Nel progetto CONV la produttività effettiva è il riutilizzo avviene è solo fortuito e non entra nel nostro concetto di riutilizzabilità.

D'altra parte, nel progetto ROD la produttività è più alta tra il primo e il secondo trimestre perché più codice viene scritto ex-novo. Mentre, tra il secondo e il terzo trimestre, un numero maggiore di moduli nel repository porta a un maggiore riutilizzo e a meno codice scritto per ogni requisito software sviluppato, questo fa diminuire la produttività effettiva perché si verifica un maggiore riutilizzo.

La tabella 7.13 riporta i valori del livello p ottenuti dal test Mann-Whitney U.

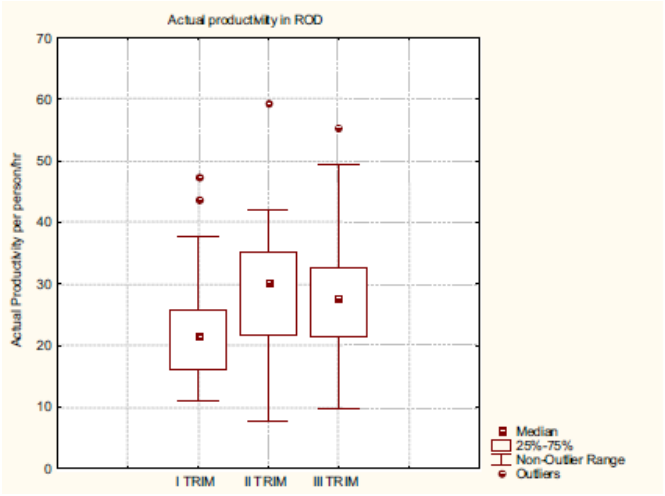

Figura 7.17 Produttività effettiva in ROD

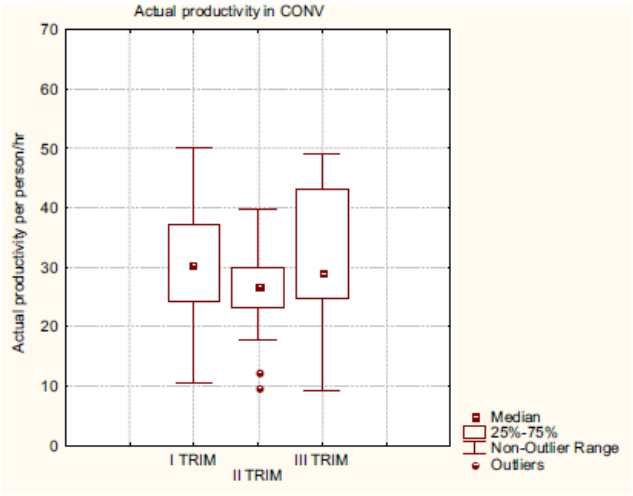

Figura 7.18 Produttività attuale in ROD

Trimester	p-level
Trimester 1 (ROD vs. CONV)	0,0025
Trimester 2 (ROD vs. CONV)	0,2048
Trimester 3 (ROD vs. CONV)	0,5696

Tabella 7.13 Valori di p-level del test di Mann-Whitney U applicato alla produttività effettiva

Il test rivela una differenza significativa solo tra trattamenti nel primo trimestre. Ciò riflette l'andamento del grafico precedentemente commentato e conferma anche che i punti di forza del modello orientato al riutilizzo sono legati all'adozione sistematica

del riutilizzo. Infatti, una volta riutilizzato è escluso, le osservazioni si riferiscono alla produttività media di uno sviluppatore che nei processi maturi (come mostrato nella prossima tabella) tende ad oscillare all'interno di un intervallo definito. Questo intervallo, come appare nella precedente apparente produttività è superato solo grazie al riuso. Inoltre, in questa analisi, dove il riuso è stato escluso, i benefici percepiti non emergono più.

La tabella 7.14 riporta i valori del test di Wilcoxon applicato alla produttività effettiva.

Project	Trimester	vs Trimester	p-level
CONV	Trimester 1	Trimester 2	0,1329
	Trimester 2	Trimester 3	0,1476
ROD	Trimester 1	Trimester 2	0,0792
	Trimester 2	Trimester 3	0,6639

Tabella 7.14 Valori di p-level del test di Wilcoxon applicato alla produttività effettiva

In tutti i casi il valore del livello p è maggiore della soglia fissata. Quindi, una differenza significativa tra le mediane di due osservazioni successive non esiste. Ancora una volta, questo risultato conferma che non c'è stato alcun effetto di maturazione tra i trimestri.

Tendenza dei depositi e tasso di riutilizzo

La figura 7.19 mostra il contenuto del repository alla fine di ogni trimestre in termini di come è stato popolato con moduli riutilizzabili. Inizialmente, il contenuto del repository è limitato ai modelli. Man mano che i requisiti del software vengono sviluppati, i modelli vengono recuperati dal repository e gli artefatti vengono implementati insieme al loro codice, documentazione e casi di test. Questi sono poi memorizzati nel repository e possono essere usati per implementare altri requisiti. Si può vedere dal grafico che dopo il primo trimestre, il contenuto del repository cresce progressivamente a causa del fatto che sempre più moduli riutilizzabili possono essere implicati per sviluppare nuovi requisiti software. Infatti, si osserva un picco verso il terzo trimestre.

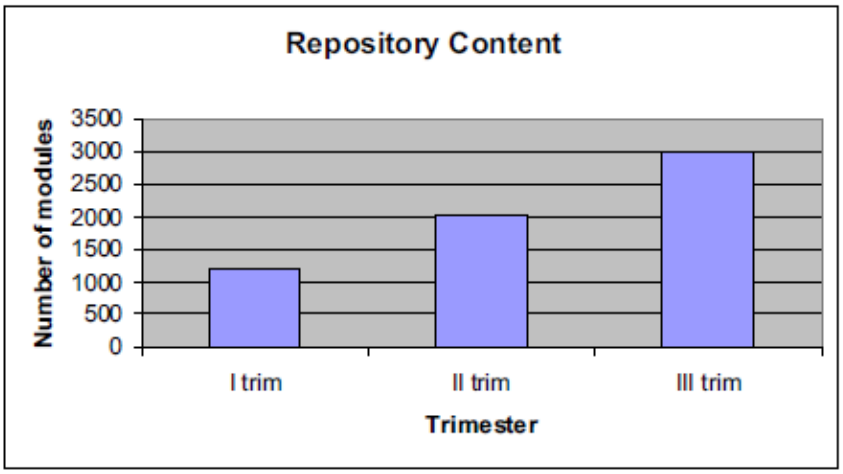

Figura 7.19: andamento del contenuto del repository

Dato l'andamento del repository lungo i tre trimestri, è interessante analizzare quanto riutilizzo è stato adottato per implementare ogni requisito software all'interno dei tre trimestri. Questo ci dà un'idea dell'impatto del riuso sullo sviluppo. La metrica utilizzata per rappresentare tali informazioni è il *Reuse Rate*, valutato come il rapporto tra la LOC riutilizzata e la LOC totale di ogni requisito software sviluppato. Misura la percentuale di codice riutilizzato, beneficiando dei moduli e dei modelli memorizzati nel repository, per sviluppare ogni nuovo requisito. I risultati sono rappresentati nella figura 7.20. Il grafico indica un aumento costante nel tempo. Queste metriche sono state raccolte e discusse solo per il progetto ROD. Era insensato considerare questi aspetti per il processo CONV.

L'andamento evidenziato nella figura 7.20 è conforme a quanto osservato nella figura 7.19 ed è coerente con il comportamento mostrato rispettivamente nelle figure 7.15, 7.16, 7.17, 7.18. Infatti, man mano che il repository si popola di moduli riutilizzabili, il tasso di riutilizzo per lo sviluppo di ogni requisito aumenta. Questo implica che la produttività apparente in ROD (figura 7.15) aumenta man mano che il repository si popola. Inoltre, per quanto riguarda la produttività reale (figura 7.16), tranne che per il primo trimestre, essa diminuisce man mano che il contenuto del repository aumenta con nuovi elementi riutilizzabili.

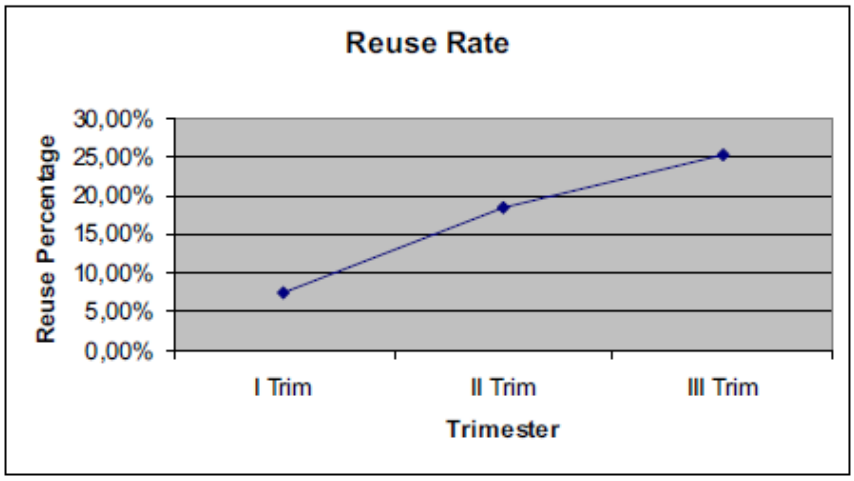

Figura 7.20: Tasso di riutilizzo dei requisiti del software

7.6.3.3 Impatto sulla manutenzione completa del riutilizzo

Dopo aver valutato i risultati precedenti[bBaldassarre05], si è indagato se un sistema software, sviluppato utilizzando un modello orientato al riuso è più facile da mantenere. Queste considerazioni hanno permesso di rispondere alla RQ3. In questo senso, dopo essere stati sviluppati, i due sistemi sono stati sottoposti a manutenzione utilizzando un processo di miglioramento iterativo (CONV) da un lato e un processo di Full Reuse Maintenance (FRM) dall'altro. Durante il periodo di indagine, sono state raccolte metriche per monitorare l'andamento della qualità del prodotto in seguito alle attività di manutenzione. Maggiori dettagli sono riportati in [Baldassarre03]. Di seguito vengono delineati i risultati principali: si è analizzato gli impatti del Reuse Oriented Development (ROD) sulla Full Reuse Maintenance (FRM) attraverso entrambe le metriche di prodotto e di processo. Per quanto riguarda le metriche di prodotto, è stata misurata la complessità complessiva del sistema (Integration Cyclomatic Complexity – ICC). I risultati hanno evidenziato che la tendenza della complessità del sistema è aumentata più con il miglioramento iterativo che con FRM. Questo è un sintomo di degradazione del sistema, per cui un sistema soggetto a manutenzione convenzionale è indotto a una maggiore degradazione rispetto a uno soggetto a FRM. Una minore complessità per tutto il periodo di osservazione nel sistema FRM è attribuibile ai moduli riutilizzabili memorizzati nel repository e costantemente aggiornati. Per quanto riguarda le metriche di processo, l'indagine ha evidenziato che l'FRM è più efficiente di uno convenzionale. Questo è stato valutato misurando la produttività nell'esecuzione delle richieste di manutenzione. Più precisamente, c'era una maggiore quantità di produttività

apparente nel processo FRM. Questo è legato al fatto che il riutilizzo avveniva sistematicamente ed era una parte integrata del processo. Utilizza componenti riutilizzabili, memorizzati nel repository e costantemente aggiornati.

I risultati ottenuti in [Baldassarre03] hanno evidenziato che il software sviluppato seguendo una prospettiva "orientata al riutilizzo" è più mantenibile. Infatti, il sistema è "costruito" con moduli che vengono memorizzati in un repository e recuperati per soddisfare nuovi requisiti del software. Il riutilizzo è sistematico ed è integrato nel processo. Quindi i moduli sono costantemente aggiornati e testati, con inevitabile impatto sulla qualità finale. Di conseguenza, FRM può essere applicato solo se un sistema software è stato precedentemente progettato e sviluppato da un punto di vista orientato al riuso, cioè seguendo un processo ROD che popola il repository di artefatti riutilizzabili

8. BIBLIOGRAFIA

AGILE, «Agile Manifesto ISO – Principi» 2001. [Online]. Available: https://agilemanifesto.org/iso/it/principles.html.

Agresti, W. W. and F. E. McGarry (1988). The Minnowbrook Workshop on Software Reuse: A summary report. Software Reuse: Emerging Technology. Tracz W. Washington, D.C., Computer Society Press: 33-40.

Ardimento P., Baldassarre M.T., Bianchi A., Visaggio G., "Empirical Studies as a Means for Technology Transfer", Proceedings of Workshop Series on Empirical Studies in Software Engineering (WSESE) Rome, Sept.2003

Baldassarre M.T., Bianchi A., Caivano D., Visaggio C., "An Industrial Case Study on Reuse Oriented Development", 21st IEEE International Conference on Software Maintenance (ICSM'05), 26-29 Sept. 2005, Budapest, Hungary

Baldassarre M.T., Bianchi A., Caivano D., Visaggio C., "Full Reuse Maintenance Process for Reducing Software Degradation", Proc European Conference on Software Maintenance and Reengineering (CSMR), Benevento, 2003.

Baldassarre M.T., Bianchi A., Caivano D., Visaggio C.A., Stefanizzi M., "Towards a Maintenance Process that Reduces Software Quality Degradation Thanks to Full Reuse" Proc of the 8th Workshop on Empirical Studies of Software Maintenance-WESS, Montreal Canada, Oct 2002, pp. 23-27

Basili R., Rombach H. D. "Towards a Comprehensive Framework for Reuse: A Reuse Enabling Software Evolution Environment," University of Maryland Computer Science Technical Report, UMIACS-TR-88-92, December 1988B.

Basili V. R. , "Quantitative Evaluation of Software Methodology," Dept of Computer Science, University of Maryland, College Parle, TR-1519, July 1985 [also in Proc. of the First Pan Pacific Computer Conference, Australia, September 1986].

Basili V. R., Rombach H. D., Bailey J., and Joo B. G. , "Software Reuse: A Framework," Proc. of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987.

Basili V. R., Rombach H. D. "The TAME Project: Towards Improvement-Oriented Software Environ- ments," IEEE Transactions on Software Engineering, vol. SE-14, no. 6, June 1988A, pp. 758-773.

Basili V. R., Turner A. J. , "Iterative Enhancement A Practical Technique for Software Development," IEEE Transactions on Software Engineering, vol. SE-1, no. 4, pp. 390-396, December, 1975.

Basili V. R., Weiss D. M., "A Methodology for Collecting Valid Software Engineering Data," IEEE Tran- sactions on Software Engineering, vol. SE-10, no. 6, pp. 728-738, November 1984.

Basili V.R., "Viewing Maintenance as Reuse-Oriented Software Development", IEEE Software, gennaio 1990, pp19-25.

Basili V.R., Caldiera G., Rombach H.D., "Goal Question Metric Paradigm", Encyclopedia of Software Engineering, Vol.1, John Wiley & Sons, 1994, pp. 528-532

Basili Victor R, "Maintenance = Reuse-Oriented Software Development", University of Maryland, CS-TR-2244, UMIACS-TR-89-48, May 1989

Basili Victor R., "Software Modeling and Measurement: The Goal/Question/Metric Paradigm," University of Maryland, CS-TR-2956, UMIACS-TR-92-96, September 1992.

Basili Victor, Caldiera Gianluigi, Dieter Rombach H., The Goal Question Metric Approach, 1994.

Basili, J. Musa, "The future engineering of software: a management perspective", IEEE COMPUTER, September 1991, pp.90-96.

Belady L. and Lehman M., "A Model of Large Program Development, IBM Systems Journal, vol.15, no.3, 1976.

Biggerstaff Ted, "Reusability Framework, Assessment, and Directions," IEEE Software Magazine, March 1987, pp.41-49.

Booch G., Object-oriented Analysis and Design with Applications, 2a ed. Benjamin Cummings, 1993.

Bowen T. P., Wigle G. B., Tsai J. T., "Specification of Software Quality Attributes," Technical Report RADC-TR-85-37, Rome Air Development Center, Griffiss Air Force Base, N.Y. 13441-5700, February 1985.

Briand L.C, Differding C.M, Rombach H.D., "Practical Guidelines for Measurement-Based Process Improvement", Software Process-Improvement and Practice, Vol.11, 1996, pp. 253-280.

Brown, Wallnau "Engineering of Component Based Systems", Component Based Software Engineering, IEEE Computer Society Press, 1996, pp.7-15

Bucci G., «Facoltà di Ingegneria - Ingegneria del Software» 2010. [Online]. Available: https://stlab.dinfo.unifi.it/bucci/Teaching/Ingegneria_del_software/12-Ciclo_di_vita.pdf.

Calavaro G. F., «Ingegneria dei Sistemi Software e dei Servizi di Rete» Gennaio 2015. [Online]. Available: https://lists.uniroma2.it/index.html/arc/isssr/2015-01/msg00016/06b-RUP_v1.1.pdf.

Caldiera G., Basili V.R., "Identifying and Qualifying Reusable Software Components", IEEE Computer, Feb. 1991, pp.61-70.

Card D., Comer E., "Why do so many reuse programs fail?", IEEE Software, September 1994, pp.114-115

CEFRIEL - Centro per la Formazione e la Ricerca in Ingegneria dell'Informazione, «Dipartimento di Informatica - Prof. Piuri Vincenzo» 2000. [Online]. Available: http://homes.di.unimi.it/piuri/pages/didattica/SInformatici/mat/ciclo.pdf.

Ciancarini P., «Corso di Ingegneria del Software» 8 Ottobre 2019. [Online]. Available: http://cs.unibo.it/~cianca/wwwpages/ids/5.pdf.

Cybulski , "Introduction to Software Reuse", Jacob 1996/01/01

Communications of the ACM, Volume 46, Issue 10 (October 2003), special issue on "Service-oriented computing".

Conover W.J., Practical Nonparametric Statistics, John Wiley and Sons, 1

Fafchamps D., "Organizational factors and reuse", IEEE Software, September 1994, pp.31-41

Federal Information Processing Standards, "Guideline on Software Maintenance," U.S. Dept. of Commerce/National Bureau of Standards, PIPS PUB 106, June 1984.

Fenton E.N., Pfleeger S.L., Sofware Metrics: A Rigorous and Practical Approach, 2a edizione, Int'l Thomson Computer Press, 1996.

Fielding, Roy (June 2014). "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, Section 4". IETF. Internet Engineering Task Force (IETF). RFC 7231. Retrieved 2018-02-14.

Frakes W.B., Isoda S., "Success Factors of Systematic Reuse", IEEE Software, Sept. 1994, pp. 14-19.

France R., Bieman J.M., "Multi-View Software Evolution: a UML-Based Framework for Evolving Object-Oriented Software", Proc. of the Int. Conf. on Software Maintenance, Firenze Italia, Novembre 2001, pp.386-395.

Freeman P. , "Reusable Software Engineering: Concepts and Research Directions," Proc. of the Workshop on Reusability, September 1983, pp. 63-76.

Gibbs W., "Software's chronic crisis", Scientific American, September 1994, pp.86-95

Goal Question Metric Approach in Software Quality. 17 Aug 2020

Grady R. B., "Measuring and Managing Software Maintenance," IEEE Software, Vol. 4, No. 5, September 1987, pp. 35-45.

Grinter R.E., "From Local to Global Coordination: Lessons from Software Reuse", Proc. of the 2001 Int. ACM SIGGroup Conf. on Supporting Group Work, settembre 2001, pp.144-153.

Griss M., "Software Reuse from Library to Factory", IBM Systems Journal, vol.32, nr. 4, pp.548-566, 1993

Highsmith J., AGILE, «Agile Manifesto History» 2001. [Online]. Available: https://agilemanifesto.org/history.html.

IEEE Software, special issue on 'Tools: Making Reuse a Reality', vol.4, no.7, July 1987.

IEEE Software, special issue on 'Reusing Software', vol.4, no.I, January 1987.

Introduction to Software Reuse, Cybulski, Jacob 1996/01/01

Jones G. A., Prieto-Diaz R., "Building and Managing Software Libraries," Proc. Compsac'88, Chicago, October 5-7, 1988, pp. 228-236.

Joos R., "Software Reuse at Motorola", IEEE Software, settembre 1994, pp. 42-47.

Juristo N., Moreno A.M., Basics of Software Engineering Experimentation, Kluwer Academic Publ., Boston, 2001.

Knauber P., Muthing D., Schmid K., Widen T., "Applying Product Line Concepts in Small and Medium-Sized Companies", IEEE Software, Sept. /Oct. 2000, pp.88-95.

Krueger C.W., "Software Reuse", ACM Computing Surveys, vol. 24, no.2, June 1992, pp.131-183.

Krueger, C. W. (1989). Models of Reuse in Software Engineering, CMU-CS-89-188, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Lehman M.M., "Laws of Program Evolution-Rules and Tools for Programming Management", Proceedings of the Infotech State of the Art Conference, aprile 1978, pp.1-25

Lewis G., Morris E., O'Brien L., Smith D., Wrage L., SMART: The Service-Oriented Migration and Reuse Technique, Technical Note, Integration of Software-Intensive Systems (ISIS) Initiative, Software Engineering Institute at Carnegie Mellon University, September 2005

Lientz B. P., E.B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," Communications of the ACM, Vol 21, No. 6, June 1978, pp. 466-471.

Lim W.C., "Effects of Reuse on Quality, Productivity and Economics", IEEE Software, Sept. 1994, pp.23-30.

Liu Y., Wang Q., Zhuang M. e Y. Zhu, "Reengineering Legacy Systems with RESTful Web Service", 2008.

Lombardi C. e Destri G., «Dipartimento di Matematica e Informatica - cdL Informatica,» 2016. [Online]. Available: http://www.giuliodestri.it/doc/D07_SviluppoSoftwareAgile.pdf.

Maibücher Stefan. Rest web services, 2008. http://www.predic8.com/rest-webservices.htm.

Martin E.W., "Strategy for a DoD Software Initiative",IEEE Computer, marzo 1983, pp.52-59.

Mary Shaw, "Purposes and Varieties of Software Reuse," Proceedings of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July, 1987.

McIlroy D., “Mass-Produced Software Components”, Software Engineering Concepts and Techniques, 1968 NATO Conference on Software Engineering, eds. J.M. Buxton, P. Naur, and B. Randell, Petrocelli/Charter, New York 1969, pp.88-98

Moe M. M., «Comparative Study of Test-Driven Development (TDD), Behavior-Driven Development (BDD) and Acceptance Test-Driven Development (ATDD)» 2019.

Morel B., P.Alexander, “SPARTACAS: automating component reuse and adaptation”, IEEE TSE, vol.30, no.9, September 2004, pp.587-600.

Morisio M., Ezran M.,.Tully C, "Success and Failure Factors in Software Reuse", IEEE Transactions on Software Engineering, vol.28, no.4, April 2002 A, pp.340-357.

Morisio M., Romano D., Stamelos I, "Qualità, produttività e apprendimento nello sviluppo basato su framework: An Exploratory Case Study", IEEE Transactions on Software Engineering, vol.28, no.9, Sept. 2002 B, pp.876-888.

Morisio M., Tully C., Ezran M., "Diversity in Reuse Processes", IEEE Software, luglio-agosto 2000, pp.56-63.

Pighin M., "A New Methodology for Component Reuse and Maintenance", Proc. of the 5th CSMR, March 2001,Lisbon Portugal, pp.196-199

Poulin J.S., "Reuse: Been There, Done That", Comm. of the ACM, Vol.42,No.5, Maggio 1999, pp.98-100.

Prieto-Diaz R., “Status Report: Software Reliability”,IEEE Software, May 1993, pp61-66.

Prieto-Diaz R., Freeman P., "Classifying Software for Reusability," IEEE Software, vol.4, no.I, January 1987, pp. 6-16

Proceedings of the Workshop on Reusability in Programming, Alan Perlis, ed., ITT Programming, Newport, R.I., 1983.

Puppet, «DevOps Research & Assessment» 2017. [Online]. Available: https://services.google.com/fh/files/misc/state-of-devops-2017.pdf.

Rine D.C., "Success Factors for Software Reuse that are Applicable Across Domains and Businesses", Proc. of the 1997 ACM Symposium on Applied Computing, April 1997, pp.182-186.

Rombach H. D.,. Basili V. R, "A Quantitative Assessment of Software Maintenance: An Industrial Case Study," in Proc. Conf. Software Maintenance, Austin, TX, SepL 1987, pp. 134-144.

Rothenberger M.A., Dooley K.J., Kulkarni U.R., Nada N., "Strategies for Software Reuse: a Principal Component analysis of Reuse Practices", IEEE TSE, vol.29, no.8, September 2003, pp.825-837.

Sbroiavacca F., «Dipartimento di Ingegneria e Architettura» 2018. [Online]. Available: https://moodle2.units.it/pluginfile.php/208226/mod_resource/content/0/Slide%20Metodologie.pdf.

Scalera M., «Integrazione e Test di Sistemi Software,» 2018. [Online]. Available: https://elearning.di.uniba.it/mod/resource/view.php?id=9488.

Senapathi M., Buchan J. e Osman H., «DevOps Capabilities, Practices, and Challenges: Insights from a Case Study» 2018.

Sherif K., Vinze A., "A Qualitative Model for Barriers to Software Reuse Adoption", Proc. of the 20th Int. Conf. on Inf. Systems, Jan. 1999, pp.47-64.

Sherif, A.Vinze, "A Qualitative Model for Barriers to Software Reuse Adoption", Proc. of the 20th Int. Conf. on Inf. Systems, gennaio 1999, pp.47-64.

Software Productivity Consortium (SPC), "Reuse Adoption Guidebook", SPC-92051-CMC, Ver.02.00.05, Nov. 1993 Special Issue ESPRIT, IEEE Software, Nov.Dec. 1989.

Special Issue on ESPRIT, IEEE Software, novembre-dicembre 1989. 21] R.Prieto-Diaz, "Status Report: Affidabilità del software",IEEE Software, maggio 1993, pp61-66.

Staringer W., "Constructing applications from reusable components", IEEE Software, September 1994, pp.61-68

Szyperski, Messerschmitt "Component Software: Beyond Object-Oriented Programming" 2nd ed. Addison-Wesley Professional, Boston 2002

The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development. January 1999 - Rini van Solingen and Egon Berghout

Toffolon C., Dakhli S., "Software Artifacts Reuse and Maintenance: An Organizational Framework", Proc of the 2nd CSMR, marzo 1998, Palazzo degli Affari, Italia pp.228- 233.

Tomer A., Goldin L., Kuflik T.,.Kimchi E,Schach S., "Evaluating software reuse alternatives: a model and its application to an industrial case study", IEEE TSE, vol.30, no.9, September 2004, pp.601-612.

Tracz W., "Tutorial on 'Software Reuse: Emerging Technology'," IEEE Catalog Number EH0278-2, 1988.

Vacca M., «Ingegneria dell'Informazione» 2009. [Online]. Available: http://infocom.uniroma1.it/~cdainformazione/uploads/IngegneriaDelSoftware/Modellievolutivi2.pdf.

Visaggio G., "Assessing the Maintenance Process through Replicated, Controlled Experiment", Journal of Systems and Software, Elsevier Science, Vol.44, No.3, 1999, pp.187-197. 16] R.Wiley, T.Elrad, R.E. Filman, Bader A., "Aspect-Oriented programming", Comm. of the ACM, Vol.44 No.10, Oct. 2001, pp.29-32.

Visaggio G., "Comprendere la conoscenza immagazzinata nei sistemi legacy invecchiati per migliorare le loro qualità con un processo di rinnovamento", ISERN-97-26 International Software Engineering Research Network, 1997

Wasmund M., "Implementing Critical Success Factors in Software Reuse", IBM Systems Journal, vol.32, no.4, 1993, pp.595-611.

Watson A.H., McCabe T.J., "Structured Testing: A Design Methodology Using the Cyclomatic Complexity Metric," NIST Special Publication 500-235, D.R. Wallace, ed., NIST Contract 43NANB517266, Sept. 1996.

Williams L., Maximilien M. E. e Vouk M., «Test-driven development as a defect-reduction practice» Denver, CO, USA, USA, 2003.

Xiaolong Jin and Jiming Liu, "From Individual Based Modeling to Autonomy Oriented Computation", in Matthias Nickles, Michael Rovatsos, and Gerhard Weiss (editors), Agents and Computational Autonomy: Potential, Risks, and Solutions, pages 151-169, Lecture Notes in Computer Science, vol. 2969, Springer, Berlin, 2004. ISBN 978-3-540-22477-8

Yau S. S., Nicholl R. A., Tsai J. J.-P., and Liu S.-S., "An Integrated Life-Cycle Model for Software Maintenance," IEEE Transactions on Software Engineering, Vol. SE-14, No. 8, August 1988, pp. 1128- 1144.

Zanella M., «Area disciplinare di Ingegneria» Settembre 2013. [Online]. Available: https://lpg.unibs.it/se/aa2013-2014/3_Processo.pdf.