SEZIONE

A

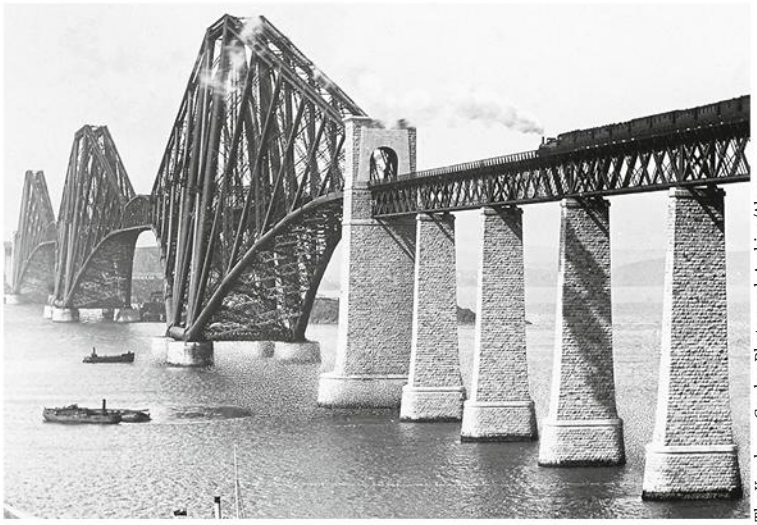
Ciclo di vita del software, UML e linguaggio C++

A1

Ciclo di vita e ingegneria del software

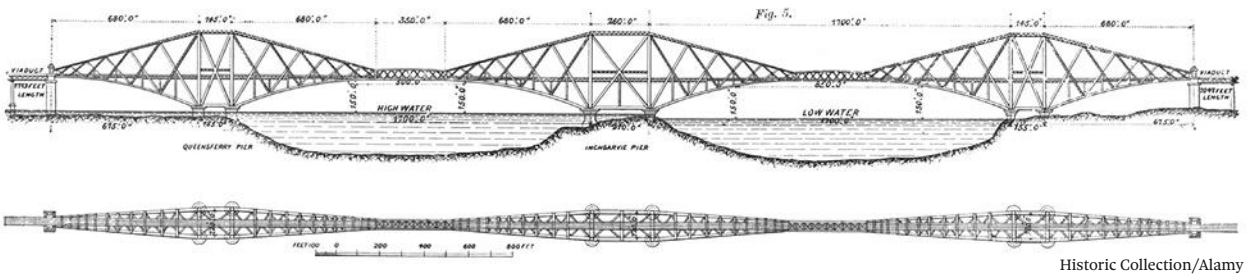
Il Forth Rail bridge, a Edimburgo, fu costruito in poco più di sette anni, dal 1883 al 1890.

FLHC 114A/Alamy



The Keasbury-Gordon Photograph Archive/Alamy

Come tutte le attività complesse, la costruzione del ponte richiese un'accurata progettazione della struttura e una precisa pianificazione delle fasi di realizzazione.



Historic Collection/Alamy

Verosimilmente gli ingegneri John Fowler e Benjamin Baker seguirono il tradizionale processo di realizzazione delle grandi opere dell'ingegneria civile, in cui ogni fase del processo segue necessariamente la fase precedente:

- progettazione;
- costruzione;
- manutenzione.

Il processo di realizzazione di un'applicazione complessa non è dissimile ed è schematizzato nel famoso **modello a cascata** del ciclo di vita del software (FIGURA **1**).

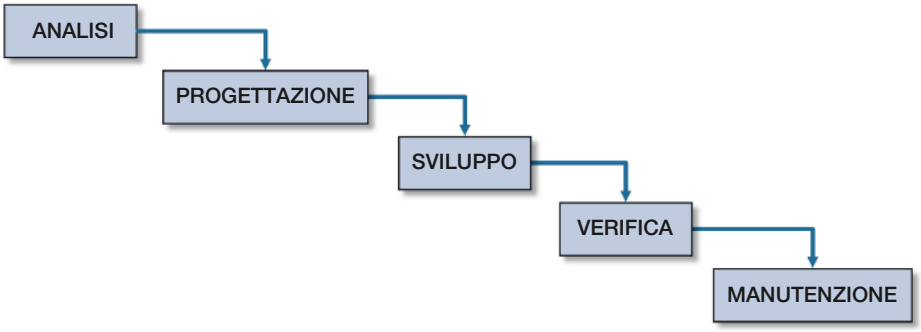


FIGURA **1**

Il **ciclo di vita del software** è l'insieme delle attività e delle azioni da intraprendere per realizzare un progetto software.

Ciascuna fase del ciclo di vita a cascata del software (*software waterfall life-cycle*) produce uno specifico output che rappresenta l'input della fase successiva (TABELLA **1**):

TABELLA **1**

Fase	Descrizione	Output
Analisi	La fase di analisi ha lo scopo di identificare i requisiti dell'applicazione da realizzare.	Requisiti
Progettazione	Nella fase di progettazione (*design*) viene definita l'architettura del software che deve essere sviluppato, avendo come scopo la decomposizione in componenti.	Architettura
Implementazione	In questa fase viene sviluppato e testato (*development*) il codice dei singoli componenti software che costituiscono l'applicazione, utilizzando uno o più linguaggi di programmazione.	Codice sorgente/ eseguibile da verificare
Verifica	L'integrazione dei componenti, la corretta esecuzione del codice sviluppato e il fatto che l'applicazione realizzata soddisfi i requisiti previsti devono essere puntualmente verificati (*testing*).	Codice sorgente/ eseguibile da rilasciare
Manutenzione	Successivamente al rilascio dell'applicazione software realizzata è spesso necessario correggere errori (*bug*) non rilevati in fase di verifica (manutenzione correttiva) o integrare funzionalità non previste in fase di analisi e identificazione dei requisiti (manutenzione evolutiva).	Versioni aggiornate del codice sorgente/ eseguibile

Documentazione di un prodotto software

Oltre allo sviluppo del codice, ogni progetto software richiede la redazione di un'adeguata documentazione: la **documentazione tecnica** (per esempio la descrizione dell'architettura e delle interazioni tra i componenti o la procedura di installazione e configurazione) è destinata agli sviluppatori che devono mantenere il prodotto, mentre la **documentazione utente** (per esempio il manuale d'uso) è destinata agli utilizzatori dell'applicazione.

OSSERVAZIONE Le fasi di analisi e di verifica dei requisiti coinvolgono normalmente il committente del progetto software. La maggior parte dei progetti software prevede infatti lo sviluppo di un'applicazione su commessa e non di un prodotto da immettere sul mercato. In questo caso il committente è un soggetto pubblico o privato che richiede la realizzazione di un software con determinate caratteristiche per soddisfare le proprie esigenze e i cui utenti sono il proprio personale e/o i propri clienti.

ESEMPIO Una scuola che intenda informatizzare la gestione delle risorse condivise (per esempio: aule e spazi speciali, strumenti o dispositivi particolari, ...) necessita di un sistema software che consenta ai docenti di prenotare per una data e un orario specifico la risorsa necessaria e che al tempo stesso ne visualizzi la disponibilità tenendo conto delle prenotazioni già effettuate in precedenza. Il sistema dovrebbe inoltre consentire alla dirigenza della scuola di risolvere eventuali conflitti di prenotazione da parte dei docenti o di impostare dei criteri di priorità (per esempio ai docenti di Inglese per il laboratorio linguistico, o ai docenti di Informatica per i laboratori computer).
La scuola commissionerà la realizzazione dell'applicazione software a un'azienda d'informatica che potrà in seguito renderla disponibile in forma di servizio esposto sulla rete Internet gestendo il riconoscimento dei docenti mediante *username* e *password*, oppure fornirla alla scuola che ne curerà l'installazione e l'accesso mediante la propria rete privata. Nella fase di analisi saranno documentati i requisiti che faranno parte integrante del contratto di affidamento della commessa e che saranno verificati dal committente al momento in cui il servizio risulta disponibile, o dell'installazione.

1 Ingegneria del software e metodologie di sviluppo

Fino a circa trent'anni fa quasi un quarto di tutti i grandi progetti software veniva interrotto prima della conclusione e più della metà veniva terminato in notevole ritardo o con costi molto maggiori di quelli inizialmente previsti. Queste statistiche – unite al fatto che frequentemente il software realizzato non soddisfava i requisiti del committente – non erano accettabili per la moderna industria del software: la situazione era nota almeno dal 1968, anno in cui il Comitato scientifico della NATO organizzò un convegno a Garmisch, in Germania, sulla «crisi del software». È solo con l'avvento del nuovo millennio che l'adozione sistematica delle metodologie dell'ingegneria del software ha introdotto miglioramenti significativi nella capacità di progettare e realizzare software di qualità.

ESEMPIO Uno dei più noti esempi di progetti software dall'esito disastroso è dato dallo sviluppo del sistema di controllo dello smistamento dei bagagli nell'aeroporto di Denver, in Colorado (USA), uno dei più grandi del mondo.
La complessità di realizzazione del software, distribuito su centinaia di computer, che doveva gestire lo spostamento di migliaia di convogli guidati automaticamente su una rete di binari sotterranei estesa decine di chilometri, ha ritardato di quasi due anni – dal 1993 al 1995 – l'inaugurazione dell'aeropor- ▶

to, comportando perdite economiche che hanno superato il milione di dollari al giorno.
La conclusione della commessa ha in ogni caso richiesto una riprogettazione sostanziale del sistema di smistamento dei bagagli, in cui è stato abbandonato il controllo completamente automatico in favore di una più tradizionale movimentazione gestita da operatori.
Tutte le analisi condotte su questo disastro dell'industria del software concordano sul fatto che i responsabili dell'aeroporto e della ditta fornitrice del sistema ne hanno – in mancanza di una rigorosa disciplina progettuale – sottostimato l'enorme complessità realizzativa; in particolare, la mancanza di una metodologia condivisa ha rapidamente condotto al caos, tanto che alcuni responsabili organizzativi e tecnici hanno cambiato lavoro o incarico.

L'**ingegneria del software** è la disciplina tecnologica e gestionale relativa alla realizzazione sistematica e alla manutenzione di un prodotto o servizio software rispettando tempi e costi preventivati.

Il **ciclo di vita** è uno dei concetti fondamentali dell'ingegneria del software, ma la rigida sequenzialità delle fasi prevista dal processo a cascata risulta impraticabile nei progetti reali. I problemi che emergono in fase di verifica e di manutenzione correttiva comportano frequentemente modifiche al codice e, di conseguenza, una ripetizione parziale della fase di implementazione. Inoltre la necessità di eseguire la manutenzione evolutiva delle funzionalità del software può anche determinare la ripetizione dell'intero processo di sviluppo a partire dalle fasi di analisi e progettazione. Per questi motivi il ciclo di vita a cascata è stato con il tempo superato per flessibilità e praticabilità da processi di tipo incrementale e iterativo.

Un processo di sviluppo è **incrementale** se il software che ne costituisce il risultato viene realizzato in versioni successive, ciascuna delle quali evolve dalla precedente accrescendone le funzionalità, fino a soddisfare tutti i requisiti previsti. La realizzazione di varie versioni del software implica l'**iterazione** delle tradizionali fasi di analisi, progettazione, implementazione e verifica.

OSSERVAZIONE Non deve sorprendere che anche la fase di analisi in cui si definiscono i requisiti del sistema software da realizzare sia inclusa nell'iterazione: molte esperienze di sviluppo di grandi progetti software dimostrano che i requisiti sono estremamente soggetti a cambiare nel corso del progetto stesso. È infatti difficile stabilire a priori in modo corretto e completo i requisiti di un'applicazione e, di conseguenza, è estremamente probabile che la valutazione di un prototipo da parte del committente suggerisca un adattamento incrementale dei requisiti iniziali.

La FIGURA **2** (a pagina seguente) chiarisce perché il processo incrementale e iterativo sia noto anche come **modello a spirale** del ciclo di vita del software.

Pur condividendo un ciclo di vita a spirale di tipo incrementale e iterativo, esistono molte metodologie per lo sviluppo di un progetto software; tra le più adottate dalle grandi aziende vi è **RUP** (*Rational Unified Process*), mentre **PSP** (*Personal Software Process*) è stata specificatamente ideata per

Prodotti e servizi software

È sempre più comune il caso che l'oggetto finale di un progetto software non sia il rilascio al committente – o la vendita ai clienti – di un programma, ma l'erogazione di un servizio.
Per quanto basati su prodotti estremamente complessi, la cui manutenzione correttiva ed evolutiva è continua, siti come *Google* e *Amazon* rendono disponibili su web le proprie funzionalità senza richiedere ai propri utenti l'acquisto o l'installazione di un'applicazione software.
Sempre più spesso questa soluzione viene adottata per sostituire, anche da parte di soggetti pubblici e privati, le funzionalità di applicazioni software tradizionali.

i singoli sviluppatori. In ogni caso queste metodologie di sviluppo, anche se estremamente diverse tra loro, sono entrambe molto disciplinate e dettagliate, e la loro applicazione a un progetto software richiede strumenti specifici e una formazione dedicata.

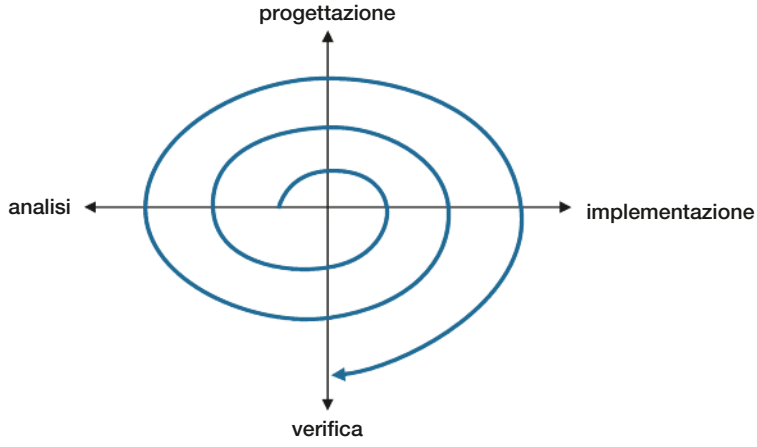


FIGURA **2**

Negli ultimi vent'anni si sono affermate forme meno strutturate di metodologie di sviluppo che hanno assunto la denominazione di **metodologie agili** e che, pur con molte differenze, condividono un manifesto comune pubblicato nel 2001:

> *Stiamo scoprendo modi migliori di creare software,*
> *sviluppandolo e aiutando gli altri a fare lo stesso.*
> *Grazie a questa attività siamo arrivati a considerare importanti:*
>
> ***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*
>
> *Ovvero, fermo restando il valore delle voci a destra,*
> *consideriamo più importanti le voci a sinistra.*

Tra i firmatari del manifesto figurano i nomi di noti sviluppatori e autori di testi di programmazione di fama internazionale; il sito del manifesto (www.agilemanifesto.org) riporta i princìpi informali a cui ogni metodologia agile dovrebbe attenersi.

Per assicurare la qualità del codice rilasciato tutte le metodologie agili enfatizzano il ruolo del ***testing*** del software prodotto fino a teorizzare la predisposizione del codice di test prima ancora dello sviluppo del codice dell'applicazione da realizzare (*test-driven development*).

Un'altra pratica condivisa dalle metodologie agili che deriva direttamente dalla focalizzazione dell'attenzione sulla qualità del codice prodotto è il ***refactoring***: anche il codice sorgente che genera il codice eseguibile corretto e conforme ai requisiti viene continuamente ristrutturato allo scopo di migliorarne la coerenza e la flessibilità in fase di manutenzione correttiva ed evolutiva.

DevOps

La sempre maggiore diffusione di applicazioni software fruibili come servizi erogati mediante server virtuali e accessibili dagli utenti attraverso la rete, rende indispensabile una stretta collaborazione tra chi sviluppa le applicazioni stesse, chi ne gestisce l'installazione e la configurazione e chi ne effettua il monitoraggio. Questa modalità di distribuzione del software come servizio (nota come SaaS, *Software as a Service*) permette di realizzare uno degli obiettivi fondamentali delle metodologie agili, la possibilità di rilasciare continuamente aggiornamenti del software. Il termine *DevOps* (contrazione di *Development* e di *Operations*) viene spesso utilizzato per indicare gli strumenti tecnologici e le metodologie organizzative che facilitano l'interazione tra i responsabili dello sviluppo del software e i responsabili della gestione dei servizi.

Testing e *refactoring* sono aspetti strettamente connessi nella pratica dello sviluppo software, perché l'introduzione deliberata di modifiche al codice già verificato può essere effettuata solo se la ripetizione dei test di conformità ai requisiti è di semplice attuazione, preferibilmente mediante strumenti automatici.

1.1 *Scrum*

Pur volendo evitare la rigida strutturazione delle metodologie tradizionali, alcune metodologie agili sono state formalizzate dai loro autori; una fra le più note e diffuse è ***Scrum***, il cui nome identifica il pacchetto di mischia del gioco del rugby con l'esplicita intenzione di enfatizzare il ruolo del gruppo di sviluppatori nel raggiungere uno scopo comune. Gli ideatori di *Scrum*, Jeff Sutherland e Ken Schwaber, lo hanno presentato per la prima volta nel 1995 e, dal 2010, pubblicano gli aggiornamenti della relativa guida sul sito https://scrumguides.org.

OSSERVAZIONE *Scrum* viene principalmente usato nel campo dello sviluppo software, ma è una metodologia generale per guidare il lavoro comune di un gruppo di persone a risolvere in modo incrementale e iterativo un problema complesso.

Scrum è un processo di sviluppo incrementale basato sull'iterazione di fasi definite *sprint* della durata massima di un mese ed è adatto a piccoli team di lavoro composti tipicamente da meno di dieci sviluppatori, da uno *Scrum master* che ha il compito di coordinare («servire» nella terminologia di *Scrum*) il team e da un *product owner* che rappresenta il committente del prodotto, o servizio, software da realizzare e che ne definisce le caratteristiche e i relativi requisiti in un documento noto come *product backlog*.

Gli ideatori di *Scrum* hanno fin dall'inizio definito i suoi tre pilastri fondamentali che lo *Scrum master* ha il compito di concretizzare:

- trasparenza: il processo e il prodotto del lavoro del team devono essere sempre visibili al team stesso e ai rappresentanti del committente;
- ispezione: i risultati del lavoro del team devono essere controllati spesso per individuare eventuali scostamenti dagli obiettivi;
- adattamento: gli scostamenti individuati devono produrre immediati cambiamenti (delle caratteristiche o dei requisiti del prodotto/servizio, degli strumenti o delle metodologie utilizzate dagli sviluppatori, della pianificazione effettuata, ...).

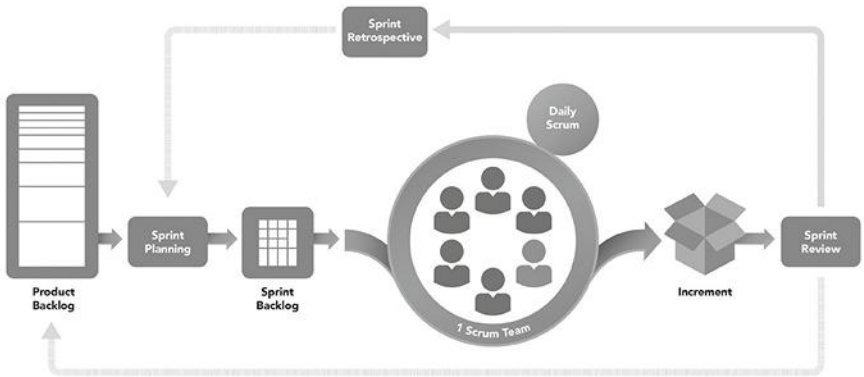
Ogni singolo *sprint* è a sua volta articolato nelle seguenti fasi e ha lo scopo di perseguire un incremento valutabile del prodotto/servizio (*sprint goal*) (FIGURA 3, a pagina seguente):

- ***sprint planning***: evento della durata massima di una giornata lavorativa in cui sono selezionate e pianificate le caratteristiche e i relativi requisiti da implementare nel corso dello *sprint* e inserite nel cosiddetto *sprint backlog*;

Beta *testing*

La verifica di un prodotto software da immettere sul mercato prevede normalmente una fase di test interno (**alfa test**) seguita da una fase di test esterno effettuata da utenti potenziali e volontari (**beta test**) che riportano al produttore i *bug* riscontrati prima del rilascio o della commercializzazione.

- ***daily Scrum***: in cui gli sviluppatori implementano i requisiti delle caratteristiche presenti nello *sprint backlog*; nel corso dello *sprint* ogni giorno il team effettua una breve riunione con il *product owner* per verificare i progressi e aggiornare lo *sprint backlog*;
- ***sprint review***: al termine di ogni *sprint* in questo evento, della durata massima di mezza giornata lavorativa, viene valutato l'incremento apportato al prodotto/servizio ed eventualmente modificato il *product backlog* se dalla valutazione emergono nuove caratteristiche o diversi requisiti da implementare;
- ***sprint retrospective***: al termine di ogni *sprint* in questo evento, della durata massima di mezza giornata lavorativa, viene valutata la qualità del lavoro svolto in relazione agli individui e alle loro relazioni e ai processi e agli strumenti utilizzati ed eventualmente individuati i cambiamenti da apportare o perseguire.



Fonte: Scrum.org Framework @ 2020 Scrum.org

FIGURA **3**

Molti *Scrum master* realizzano lo *sprint backlog* nella forma di una superficie nell'ambiente di lavoro del team – normalmente una lavagna o una parete – in cui i requisiti che definiscono le caratteristiche del prodotto/servizio sono rappresentati da post-it adesivi riposizionabili in tre diverse colonne: «da fare», «in lavorazione» e «fatto» (FIGURA **4**):

Sprint backlog			
Caratteristiche	**Da fare**	**In lavorazione**	**Fatto**
Prori solbleda vemare			
Soela lubozira ontae			
Prrio soble ltoma			

FIGURA **4**

OSSERVAZIONE Collocare un requisito nella colonna «fatto» richiede un accordo nel team di sviluppo su cosa si intende per implementazione completa di un requisito, aspetto che può includere la revisione da parte di altri sviluppatori, il superamento di varie tipologie di test, la conformità a regole relative al codice sorgente, la produzione di documentazione, ...

Nell'ultima versione della guida di *Scrum*, pubblicata nel 2020, gli autori hanno elencato cinque valori che il team dovrebbe condividere: impegno, focalizzazione, apertura, rispetto e coraggio.

2 Linguaggio di modellizzazione UML

Una tecnica generalmente adottata per mitigare il rischio di insuccesso di un progetto software consiste nella decomposizione dell'applicazione da sviluppare in componenti indipendenti, ciascuno di complessità limitata e testabile unitariamente. La decomposizione del software avviene a vari livelli, ma nel caso del codice è resa naturale dall'adozione di un linguaggio di programmazione e di una progettazione orientati agli oggetti (OOP, *Object Oriented Programming* e OOD, *Object Oriented Design*).

Il linguaggio di modellizzazione **UML** (*Unified Modeling Language*[1]) – la cui prima versione fu definita nel 1996 a opera di Grady Booch, Jim Rumbaugh e Ivar Jacobson, noti come *los tres amigos* – è un insieme di formalismi grafici che consentono di definire e documentare mediante vari tipi di diagrammi i diversi aspetti di un sistema software, in particolare se realizzato secondo il paradigma a oggetti.

1. Il termine «unificato» ricorda come la versione originale del linguaggio di modellizzazione derivi dall'integrazione di proposte precedenti degli autori.

I principali diagrammi previsti dalla versione attuale del linguaggio sono riportati nella TABELLA **2**:

TABELLA **2**

Diagramma	Descrizione
Casi d'uso	Diagramma *comportamentale* di descrizione delle funzionalità di un sistema software in termini di utenti (denominati «attori», termine con il quale si denotano sia una persona sia un sistema informatico) e di azioni eseguibili.
Classi	Diagramma *strutturale* che descrive l'architettura di un sistema software in termini di classi con i relativi attributi e metodi e delle relazioni tra esse. Il diagramma delle classi è una metodologia di progettazione a oggetti estremamente diffusa.
Oggetti	Diagramma *strutturale* che rappresenta lo stato completo o parziale di un sistema software in un istante della propria esecuzione in termini di oggetti istanziati dalle classi e delle loro relazioni e dipendenze.
Stati	Diagramma *comportamentale* che rappresenta le transizioni di stato di un componente software.
Attività	Diagramma *comportamentale* che illustra le fasi del flusso di controllo di un sistema software.
Sequenza	Diagramma di *interazione* che descrive la comunicazione tra i componenti di un sistema software in termini di «messaggi» (termine con il quale si intende l'invocazione di metodi).
Componenti	Diagramma *strutturale* che documenta l'architettura di un sistema software in termini di componenti e delle loro dipendenze.
Deployment	Diagramma *strutturale* che documenta la distribuzione delle componenti di un sistema software in esecuzione su diversi dispositivi hardware (esempio: server web, server applicativo, server database, client).

ESEMPIO

Il diagramma UML dei casi d'uso di FIGURA **5** rappresenta in forma semplificata le funzionalità consentite al cliente di un sito di vendita online (l'attore cliente è raffigurato con la classica icona grafica stilizzata). La visualizzazione dei commenti dei clienti relativi a uno specifico prodotto è un'estensione facoltativa della funzionalità di visualizzazione dei dettagli di un prodotto, mentre l'operazione di pagamento è necessariamente inclusa nell'operazione di acquisto.

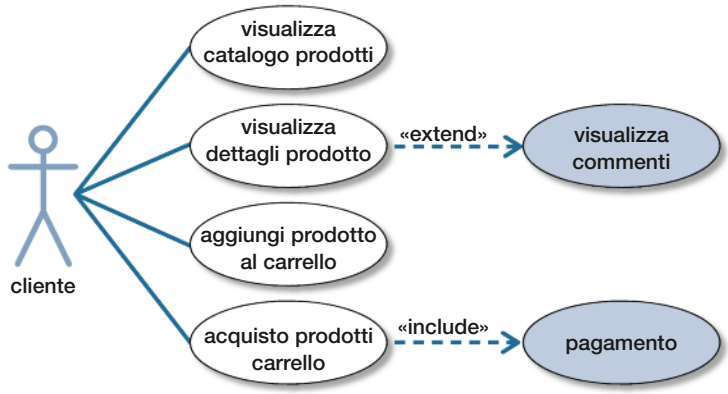


FIGURA **5**

ESEMPIO

Il diagramma UML delle classi come quello riportato in FIGURA **6** rappresenta in forma semplificata l'architettura software di un sito che commercializza libri, giornali e riviste periodiche (la classe `Prodotto` costituisce una loro generalizzazione astratta): il catalogo raccoglie tutti i prodotti disponibili, mentre un carrello comprende i soli prodotti che un cliente intende acquistare.

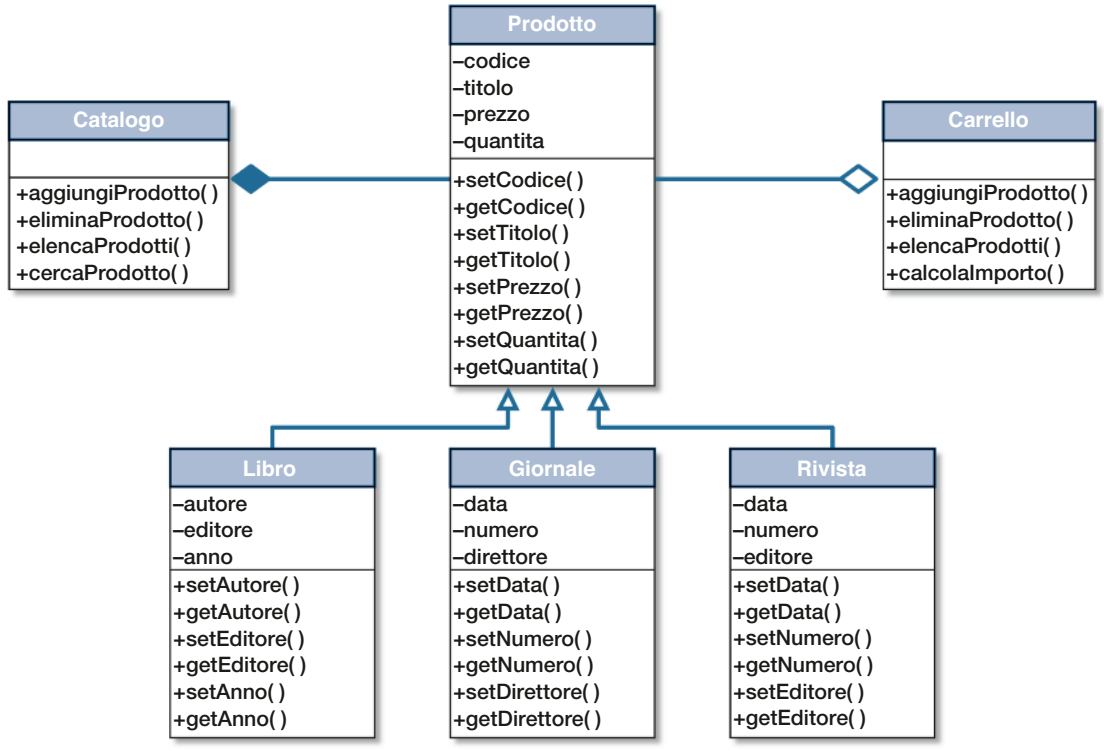


FIGURA **6**

Gli elementi di questo tipo di diagramma UML corrispondono al concetto di classe dei linguaggi di programmazione orientati agli oggetti, con i relativi attributi e metodi: la relazione di generalizzazione corrisponde all'ereditarietà tra classi, mentre le relazioni di composizione corrispondono alla definizione di attributi istanze di altre classi aventi la cardinalità specificata.

ESEMPIO

Un diagramma UML di sequenza come quello riportato in FIGURA **7** illustra l'interazione nel tempo (che trascorre dall'alto verso il basso) tra i componenti di un sistema software per la vendita online di prodotti, in particolare – se riferito a un linguaggio di programmazione orientato agli oggetti – mostra la sequenza temporale di invocazione dei metodi tra oggetti di classi diverse.

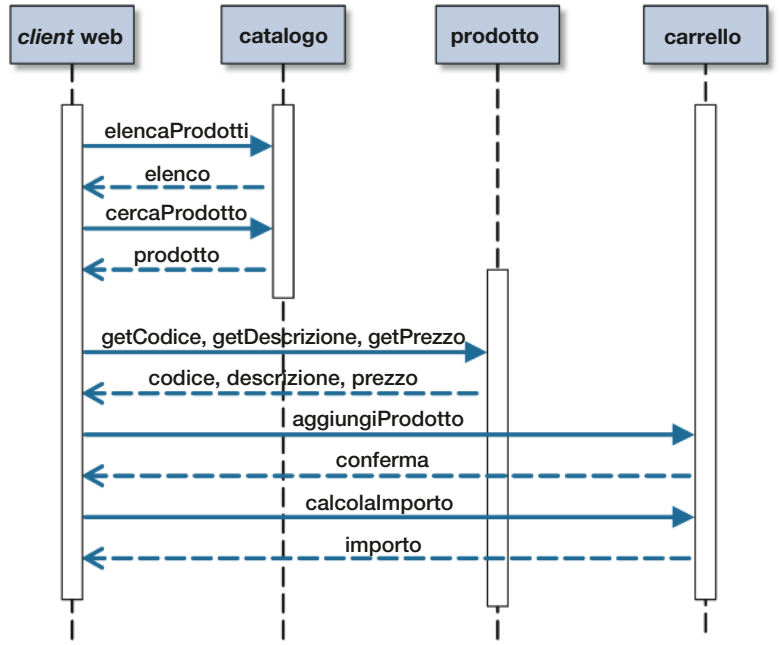


FIGURA **7**

ESEMPIO

Il seguente diagramma UML delle attività (FIGURA **8**) illustra il comportamento dell'utente di un sito web di vendita online: il diagramma ha lo scopo di dettagliare il flusso delle azioni e delle scelte operate tra l'inizio e la fine delle attività.

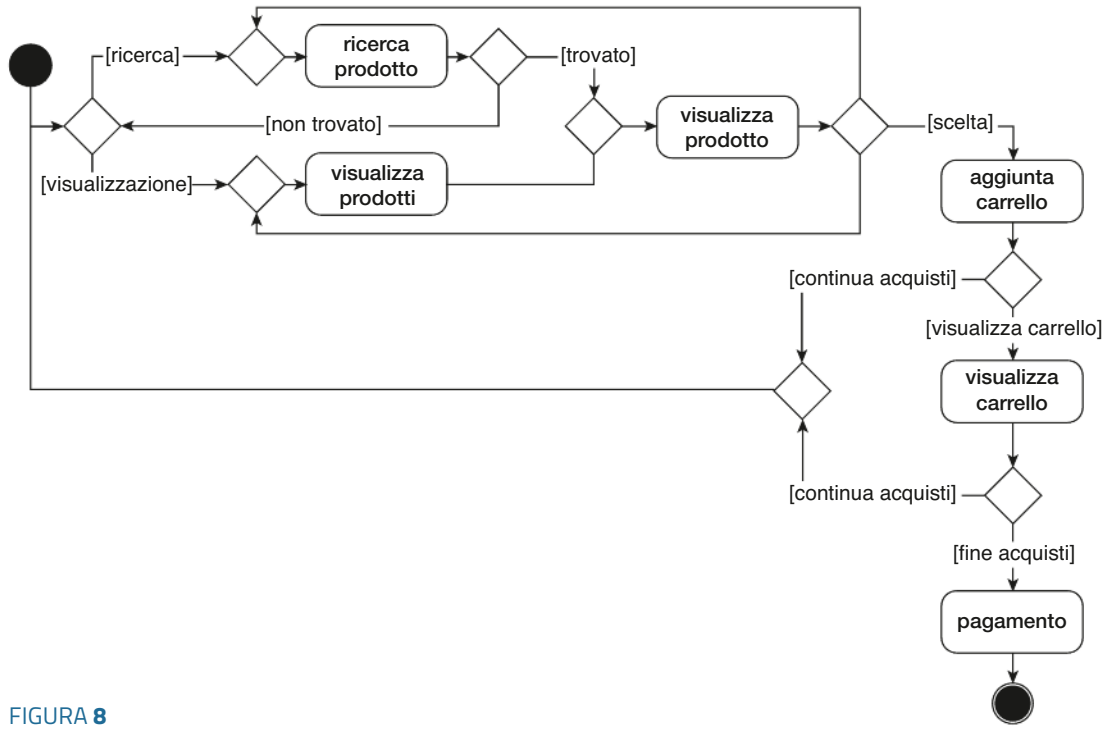


FIGURA **8**

3 Qualità del software e *pattern*

Uno standard internazionale[2] identifica le seguenti caratteristiche di qualità di un prodotto software (TABELLA 3):

2. ISO/IEC 25010:2011.

TABELLA 3

Compatibilità	Coesistenza con altri software e livello di interoperabilità.
Funzionalità	Presenza delle funzionalità che soddisfano i requisiti stabiliti e/o impliciti.
Affidabilità	Mantenimento delle prestazioni in condizioni definite e/o per un periodo di tempo stabilito.
Usabilità	Livello di impegno richiesto per l'impiego da parte di varie tipologie di utenti.
Efficienza	Relazione tra il livello di prestazione e le risorse hardware/software utilizzate in condizioni definite.
Manutenibilità	Livello di impegno richiesto per la modifica del codice.
Portabilità	Facilità di esecuzione su piattaforme diverse.
Sicurezza	Capacità di garantire i dati e di verificare gli accessi.

Dopo l'adozione di una metodologia e di un ciclo di sviluppo, uno dei fattori che maggiormente influenzano le caratteristiche di qualità di un prodotto software è il ricorso a soluzioni standardizzate per i problemi di progettazione e di implementazione. Il concetto di ***pattern*** nasce nel contesto dell'architettura di edifici e aree urbane a cura dell'architetto Cristopher Alexander, che lo definì come una soluzione a un problema ricorrente di progettazione «in modo che la soluzione sia applicabile milioni di volte senza doverla applicare allo stesso modo nemmeno due volte»; è lo stesso Alexander che stabilisce un possibile formato di pubblicazione di un *pattern*:

- nome del *pattern*;
- descrizione del problema ricorrente;
- descrizione del contesto di applicazione e delle «forze» in gioco;
- descrizione della soluzione proposta;
- descrizione del contesto risultante dall'applicazione della soluzione proposta.

È auspicabile, inoltre, documentare il *pattern* con esempi e casi noti di applicazione.

OSSERVAZIONE Le «forze» in gioco nel contesto di applicazione di un *pattern* rappresentano le motivazioni per adottare una soluzione piuttosto che un'altra e sono spesso, nei problemi reali, opposte: la soluzione proposta deve quindi tenerne conto, risultando in un contesto in cui queste trovano un effettivo equilibrio.

L'iniziale adozione del concetto di *pattern* nel contesto della progettazione e dell'implementazione del software è dovuta a Kent Beck e Ward Cunningham, che li utilizzarono come strumento didattico per la programmazione a oggetti fin dagli anni Ottanta del secolo scorso.

Ma la diffusione dei *pattern* tra i progettisti di software e i programmatori avvenne nel 1995, con la pubblicazione del famoso testo *Design Pattern:*

Antipattern

The gang of four ha coniato il termine ***antipattern*** per definire un problema ricorrente nello sviluppo software che dovrebbe essere evitato.
Uno dei più famosi – e purtroppo comuni! – è *reinventing the wheel* (*and make it square*): il rifiuto da parte dei progettisti e degli sviluppatori software a usare soluzioni sperimentate (i *pattern* appunto) porta spesso a soluzioni faticose ed errate!

elementi per il riuso di software orientato agli oggetti, a cura di Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, che sono ormai noti a tutti gli informatici come *the gang of four*. Il testo presenta 23 *pattern* per la progettazione del software che sono considerati fondamentali nella progettazione/programmazione orientata agli oggetti, classificati in tre categorie:

- **creazionali**: *pattern* relativi alla creazione di oggetti a partire da classi o oggetti;
- **strutturali**: *pattern* relativi alle modalità di composizione delle classi o degli oggetti;
- **comportamentali**: *pattern* relativi alle interazioni tra oggetti.

Nel seguito sono introdotti alcuni *pattern* che non appartengono a nessuna delle categorie individuate da *the gang of four* – si tratta di *pattern* di tipo metodologico e di uso generale – allo scopo di esemplificare il concetto.

PATTERN

Nome	***Keep It Simple.***
Problema	È difficile comprendere un sistema software quando diviene complesso.
Contesto e forze	La complessità genera numerosi errori e rende un sistema software difficilmente mantenibile. Sono desiderabili sistemi software più comprensibili, più mantenibili, più flessibili e meno esposti agli errori.
Soluzione	La progettazione del software non è un processo casuale: ci sono infatti molti fattori da prendere in considerazione. La progettazione dovrebbe essere la più semplice possibile, ma non di più. Questa soluzione facilita l'ottenimento di un sistema più facilmente comprensibile e mantenibile nel tempo, ma non significa che i requisiti e le caratteristiche (compresi i requisiti e le caratteristiche interne) debbano essere tralasciati in favore della semplicità. I progetti software più eleganti sono spesso i più semplici ed essenziali. Semplice non significa *quick and dirty*, e infatti il raggiungimento della semplicità richiede spesso molta attività concettuale e numerose iterazioni.
Contesto risultante	Il software è più comprensibile, più mantenibile e meno esposto agli errori.

PATTERN

Nome	***Make it run, make it right, make it fast, make it small.***
Problema	Quando ottimizzare un progetto software?
Contesto e forze	Si sta affrontando un nuovo progetto e la soluzione richiesta deve normalmente essere «migliore, più veloce e più economica» ed è necessaria una strategia di sviluppo che bilanci le necessità del cliente o del mercato e organizzative. L'ottimizzazione ha costi sia a breve sia a lungo termine. L'ottimizzazione precoce porta con sé il rischio di alti costi per risultati limitati: gli sviluppatori normalmente non sono in grado di prevedere quali saranno gli *hot-spot* del progetto e posticipare l'ottimizzazione fino al momento in cui essi non sono noti si è spesso dimostrato saggio («Molti errori di progettazione e programmazione sono stati compiuti in nome dell'efficienza più che per ogni altro singolo motivo, compresa la stupidità cieca»; «Il miglioramento dell'efficienza spesso viene ottenuto a costo di sacrificare qualsiasi altra desiderabile caratteristica del prodotto»).

	Lo sviluppo incrementale ha un effetto estremamente positivo sul morale degli sviluppatori: ottenere qualcosa che funziona in una fase iniziale del processo di sviluppo e che cresce progressivamente mantiene l'entusiasmo e la visione ad alti livelli. La correttezza dei prodotti è una caratteristica estremamente desiderabile, prodotti software veloci ottengono una valutazione positiva da parte del committente e i progetti «piccoli» sono più facilmente distribuibili e mantenibili.
Soluzione	Adottare un ciclo di sviluppo iterativo che ordini gli obiettivi come di seguito: 1. *make it run* (garantire il funzionamento); 2. *make it right* (garantire il rispetto dei requisiti); 3. *make it fast* (ottimizzare le prestazioni); 4. *make it small* (ottimizzare la struttura interna).
Contesto risultante	La ricerca dell'efficienza non impatta eccessivamente sulla buona progettazione dell'architettura; il morale del gruppo di sviluppo rimane elevato e solo le ottimizzazioni realmente necessarie sono effettuate.

4 Lo studio di un caso

Le metodologie di progettazione e le tecniche di sviluppo illustrate nel seguito del testo saranno esemplificate in relazione allo studio di un caso realistico: analisi, progettazione, implementazione e verifica di un'applicazione software denominata *Expense-manager* che consenta di gestire una lista di progetti alle spese dei quali partecipano nel tempo più persone. L'applicazione consentirà di:

- creare ed eliminare i progetti;
- visualizzare i pagamenti effettuati dai partecipanti a un progetto;
- registrare i pagamenti effettuati da una persona in relazione a un progetto;
- determinare il bilancio dei pagamenti effettuati dalle persone in relazione a uno specifico progetto.

I CONCETTI CHIAVE

CICLO DI VITA DEL SOFTWARE. Il ciclo di vita del software è l'insieme delle attività e delle azioni da intraprendere per realizzare un progetto software.

MODELLO A CASCATA. Il modello a cascata (*waterfall*) del ciclo di vita del software prevede le seguenti fasi rigorosamente sequenziali (tra parentesi è indicato il relativo output): analisi (requisiti), progettazione (architettura), implementazione (codice da verificare), verifica (codice da rilasciare) e manutenzione (versioni aggiornate del codice).

INGEGNERIA DEL SOFTWARE. L'ingegneria del software è la disciplina tecnologica e gestionale relativa alla realizzazione sistematica e alla manutenzione di un prodotto o servizio software rispettando tempi e costi preventivati.

PROCESSI DI SVILUPPO INCREMENTALI. Un processo di sviluppo è incrementale se il software che ne costituisce il risultato viene realizzato in versioni successive, ciascuna delle quali evolve dalla precedente accrescendone le funzionalità fino a soddisfare tutti i requisiti previsti. La realizzazione di varie versioni del software implica l'iterazione delle tradizionali fasi di analisi, progettazione, implementazione e verifica.

METODOLOGIE AGILI. Le metodologie di sviluppo agili enfatizzano il ruolo del ***testing*** del software prodotto fino a teorizzare la predisposizione del codice di test prima ancora dello sviluppo del codice dell'applicazione da realizzare (*test-driven development*). Inoltre le metodologie agili promuovono il ***refactoring***: anche il codice sorgente che genera il codice eseguibile corretto e conforme ai requisiti viene continuamente ristrutturato allo scopo di migliorarne la coerenza e la flessibilità in fase di manutenzione correttiva ed evolutiva.

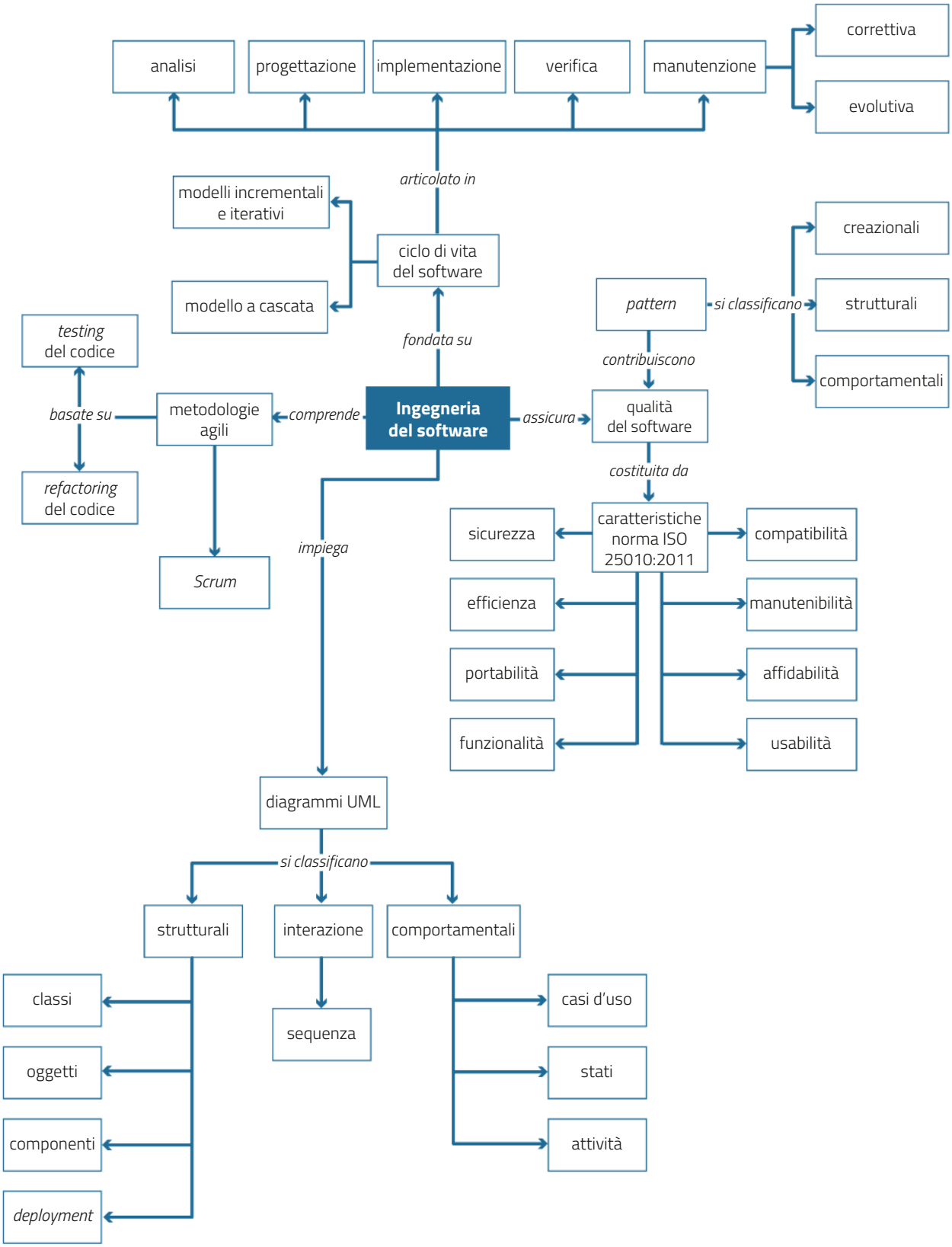
SCRUM. *Scrum* è un processo di sviluppo incrementale basato sull'iterazione di fasi definite *sprint* della durata massima di un mese ed è adatto a piccoli team di lavoro composti tipicamente da meno di dieci sviluppatori, da uno *Scrum master* che ha il compito di coordinare («servire» nella terminologia di *Scrum*) il team e da un *product owner* che rappresenta il committente del prodotto, o servizio, software da realizzare e che ne definisce le caratteristiche e i relativi requisiti in un documento noto come *product backlog*.

UML. Il linguaggio di modellizzazione UML (*Unified Modeling Language*) è un insieme di formalismi grafici che consentono di definire e documentare mediante vari tipi di diagrammi i diversi aspetti di un sistema software, in particolare se realizzato secondo il paradigma a oggetti. I principali diagrammi previsti dalla versione attuale del linguaggio sono: casi d'uso, classi, oggetti, stati, attività, sequenza e componenti.

QUALITÀ DEL SOFTWARE. Le caratteristiche che concorrono alla qualità di un prodotto software sono: la compatibilità, la funzionalità, l'affidabilità, l'usabilità, l'efficienza, la manutenibilità, la portabilità e la sicurezza.

PATTERN. Un *pattern* è una soluzione a un problema ricorrente in un dato contesto della progettazione o dello sviluppo software; la documentazione di un *pattern* prevede: il nome, la definizione del problema, la descrizione del contesto e delle forze in gioco, la descrizione della soluzione e del contesto risultante dall'applicazione del *pattern*.

RIPASSA CON LA MAPPA



QUESITI

1 **Il ciclo di vita del software è l'insieme delle attività e delle azioni...**

A da intraprendere per realizzare un progetto software.

B da intraprendere per commercializzare un prodotto software.

C da intraprendere per minimizzare il rischio di ritardo e di sforamento del costo nella realizzazione di un progetto software.

D ripetute allo scopo di migliorare la qualità di un prodotto software riducendone i difetti.

2 **L'ingegneria del software è...**

A la disciplina tecnologica e gestionale relativa alla realizzazione sistematica e alla manutenzione di un prodotto o servizio software rispettando tempi e costi preventivati.

B un modello di ciclo di vita del software alternativo al modello a cascata.

C la disciplina tecnologica e gestionale che le grandi aziende applicano per verificare i prodotti software prima del loro rilascio.

D la disciplina tecnologica e gestionale che le aziende committenti di un prodotto software applicano per verificare la soddisfazione dei requisiti.

3 **Associare le fasi del ciclo di vita del software ai relativi output:**

analisi	codice aggiornato
progettazione	codice da verificare
sviluppo	codice rilasciato
verifica	requisiti
manutenzione	architettura

4 **Un modello di sviluppo incrementale e iterativo...**

A prevede un unico rilascio del prodotto software al termine di tutte le attività di analisi, progettazione, implementazione e verifica.

B prevede il rilascio di varie versioni del prodotto software con funzionalità accresciute ripetendo le attività di analisi, progettazione, implementazione e verifica.

C è un controsenso: le attività di analisi, progettazione, implementazione e verifica sono necessariamente sequenziali.

D prevede la ripetizione da parte di gruppi di lavoro distinti delle attività di analisi, progettazione, implementazione e verifica in modo che il committente possa sempre selezionare il risultato migliore.

5 **Indicare gli aspetti privilegiati da una metodologia di sviluppo agile.**

A Documentazione rigorosa del codice.

B Verifica puntuale della funzionalità del codice.

C Rigidità dei requisiti iniziali.

D Miglioramento continuo del codice.

6 **Ordinare temporalmente le fasi di uno *sprint* della metodologia *Scrum*:**

..... *daily Scrum*
..... *review*
..... *planning*
..... *retrospective*

7 **Il linguaggio UML è...**

A il linguaggio di programmazione adatto per l'adozione di una metodologia di sviluppo agile.

B un insieme di formalismi grafici che consentono di definire e documentare i diversi aspetti di un sistema software.

C un insieme di formalismi grafici che consentono di definire e documentare i requisiti di un sistema software.

D un formalismo grafico che può essere convertito nel codice di un linguaggio di programmazione a oggetti automaticamente.

8 **Associare le tipologie dei diagrammi dei casi d'uso alla fase del ciclo di vita del software in cui sono tipicamente realizzati:**

classi	analisi
attività	progettazione
casi d'uso	sviluppo
sequenza	verifica
	manutenzione

9 **Ordinare le voci standard per la pubblicazione di un *pattern*:**

..... esempi
..... problema
..... nome
..... contesto
..... soluzione

10 **Il *pattern* «*Keep it Simple*»...**

A afferma che i requisiti di un prodotto software possono essere sacrificati al fine di realizzare una progettazione semplice.
B afferma che la progettazione di un prodotto software deve essere la più semplice possibile consentita dai requisiti.
C afferma che i requisiti di un prodotto software devono essere definiti in modo che la progettazione risultante sia semplice.
D Nessuna delle precedenti risposte è corretta.

11 **Il *pattern* «*Make it run, make it right, make it fast, make it small*»...**

A afferma che l'ottimizzazione di un prodotto software non deve essere effettuata.
B afferma che l'ottimizzazione di un prodotto software è un'attività fondamentale da svolgersi sempre in fase di sviluppo e verifica.
C afferma che l'ottimizzazione di un prodotto software è un'attività che deve essere svolta dopo lo sviluppo e la verifica.
D Nessuna delle precedenti risposte è corretta.

anyhow
comunque

customer
cliente

delivery
consegna

to embrace
comprendere, abbracciare

to entail
comportare

fault
guasto

feature
caratteristica

improvement
miglioramento

lightweight
leggero

manufacturing
produzione

overlooked
trascurato

purpose
scopo

pursuit
inseguimento

to seek
cercare, ricercare

stuck
bloccato

to tackle
affrontare

throwaway
usa e getta

training
formazione

widespread
diffuso

Understanding Agile

"*Agility is the ability to both create and respond to change in order to profit in a turbulent business environment.*"

Jim Highsmith (2002)

The first programmable computers appeared during the first half of the twentieth century. Programming them was a challenge, because nobody had ever done it before. As the capabilities of the machines expanded, more and more complex tasks were asked of them. By the late 1960s it was already apparent that large-scale programming wasn't easy. The challenge now lay in the complexity of the thing being built.

A NATO conference in 1968 coined the term *software crisis* to describe the problems that companies, governments and the military faced. The new industry responded by inventing software engineering and attempting to make programming into an engineering discipline. Since nobody really knew the right way to programme or manage a development effort, multiple methodologies and notations were proposed in an attempt to bring discipline and engineering rigour to the problem.

The crisis was never really resolved and after 20-30 years the term fell into disuse – after all, most crises last for hours, days or maybe months, not decades. But software engineering and methodologies stuck. The problem was that software projects were still unpredictable, still delivered late – if at all – and regularly over budget. If anything, the problem had got worse because computers, and thus software, were far more widespread than in 1968.

In the late 1990s a new stream of thought appeared. Several different figures in the software development community came up with similar ideas at about the same time. Kent Beck, Ward Cunningham, Alistair Cockburn, Jim Highsmith, Ken Schwaber and others proposed a new breed of "lightweight" methodologies.

Many of the lightweight proponents had previously been involved in the software Patterns community, and before that the object orientation movement. This meant that many of them knew each other and had been exposed to similar ideas and thinking. [...] It quickly became apparent that although some of the details differed, the new methodologies had a lot in common. Eventually, many of the key movers in the lightweight methodology movement came together and proposed the Agile manifesto (see topic box). So Agile software development was born. The proposed methodologies still existed in their own right, but they had a collective name.

A Manifesto for Agile Software Development

We're uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

That is, while there's value in the items on the right-hand side, we value the items on the left-hand side more.

The Roots of Agile Thinking

The thinking behind Agile methodologies isn't hard to find. Indeed, much Agile thinking is simply common sense or good management practice. In the pursuit of engineering rigour, modern management thinking had sometimes been overlooked.

The first thing that the lightweight methodologies did was to simplify the development activities. The previous generation of methodologies had tackled complexity with complexity, so simplicity was high on the list of influences.

Agile development has also been influenced by the arguments of Phil Crosby

(*Quality is Free*) and W. Edwards Deming (see the topic box). Agile developers seek to find faults early in the development cycle. In doing so, costly and disruptive re-work can be eliminated from the later stages of development. The next aspect to be embraced concerned the dirty little secret of software engineering: people. While architecture, engineering, process, notation and tools tend to dominate the literature and teaching of software engineering, there has always been an understanding that it is people who make the real difference. Even by the 1960s, it was apparent that some programmers were simply far more productive than others. [...] The Agile manifesto and methodology writers took notice of these ideas and put people centre stage in lightweight methodologies.

Other ideas that had been seen to work found their way into the mix too. The Agile development cycle looks a lot more like the classical maintenance cycle than the waterfall model often used for new development work. Maintenance work was often considered the poor relation of big new developments, but it had a better delivery record. Fixing faults and adding new features to software that already works entails less risk than developing something new. Similarly, prototyping had long been practised, but it was often considered bad form to evolve prototypes and so they were usually thrown away. [...]

So Agile has embraced prototyping too. Teams develop some aspect of the product, show it to customers, listen to the feedback and then decide what features to develop next. If customers don't like a feature, it might be dropped or replaced.

Deming's Fourteen Points

W. Edwards Deming was an American statistician, who used statistical techniques during the Second World War to improve American manufacturing output. Following the war, he taught the same techniques to Japanese managers and engineers.

Deming proposed 14 points that he thought would lead to quality improvement:

1. Create consistency of purpose.
2. Adopt new philosophy.
3. Cease dependency on inspection.
4. End awarding business on price.
5. Improve constantly the system of production and service.
6. Institute training on the job.
7. Institute leadership.
8. Drive out fear.
9. Break down barriers between departments.
10. Eliminate slogans and exhortations.
11. Eliminate work quotas or work standards.
12. Give people pride in their job.
13. Institute education and a self-improvement programme.
14. Put everyone to work to accomplish it.

[A. Kelly, "Changing Software Development: Learning to Become Agile", John Wiley & Sons, 2008]

QUESTIONS

a **When and why was the term software crisis used for the first time?**

b **Why were the new development methodologies described in late 1990s defined as «lightweight»?**

c **What is the key factor in lightweight software development methodologies?**

d **Is the classical sequential «waterfall» development cycle the preferred model for Agile software development teams?**

e **What does «prototyping» mean in the context of Agile software development?**

capabilities
capacità

cave art
arte paleolitica

to convey
esprimere

flowchart
diagramma di flusso

A picture is worth a thousand lines of code

[...]

Reviewing kinds of diagrams

There are several kinds of diagrams that you can create. I will quickly review the kinds of diagrams you can create and the kinds of information each of these diagrams is intended to convey.

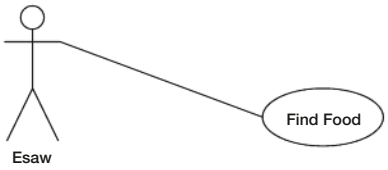


FIGURE **1.2** The "Find Food" use case.

Use case diagrams

Use case diagrams are the equivalent of modern cave art. A use case's main symbols are the *actor* (our friend Esaw) and the *use case oval* (FIGURE **1.2**). Use case diagrams are responsible primarily for documenting the macro requirements of the system. Think of use case diagrams as the list of capabilities the system must provide.

Activity diagrams

An *activity diagram* is the UML version of a flowchart. Activity diagrams are used to analyse processes and, if necessary, perform process reengineering (FIGURE **1.3**).

An activity diagram is an excellent tool for analysing problems that the system ultimately will have to solve. As an analysis tool, we don't want to start solving the problem at a technical level by assigning classes, but we can use activity diagrams to understand the problem and even refine the processes that comprise the problem.

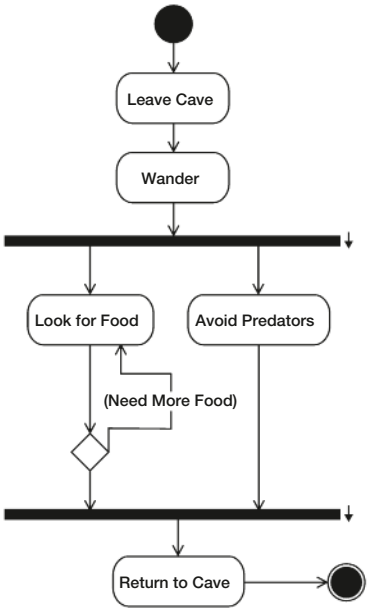


FIGURE **1.3** An activity diagram showing how Esaw goes about finding food.

Class diagrams

Class diagrams are used to show the classes in a system and the relationships between those classes (FIGURE **1.4**). A single class can be shown in more than one class in a diagram, and it isn't necessary to show all the classes in a single, monolithic class diagram. The greatest value is to show classes and their relationships from various perspectives in a way that will help convey the most useful understanding.

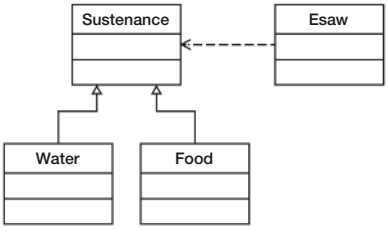


FIGURE **1.4** A single class diagram, perhaps one of many, that conveys a facet of the system being designed.

Class diagrams show a static view of the system. Class diagrams do not describe behaviours or how instances of the classes interact. To describe behaviours and interactions between objects in a system, we can turn to *interaction diagrams*.

Interaction diagrams

There are two kinds of interaction diagrams, the *sequence* and the *collaboration*. These diagrams convey the same information, employing a slightly different perspective. Sequence diagrams show the classes along the top and messages sent between those classes, modeling a single flow through the objects in the system.

Collaboration diagrams use the same classes and messages but are organised in a spatial display.

FIGURE **1.5** shows a simple example of a sequence diagram, and FIGURE **1.6** conveys the same information using a collaboration diagram.

A sequence diagram implies a time ordering by following the sequence of messages from top left to bottom right. Because the collaboration diagram does not indicate a time ordering visually, we number the messages to indicate the order in which they occur.

Some tools will convert interaction diagrams between sequence and collaboration automatically, but it isn't necessary to create both kinds of diagrams. Generally, a sequence diagram is perceived to be easier to read and more common.

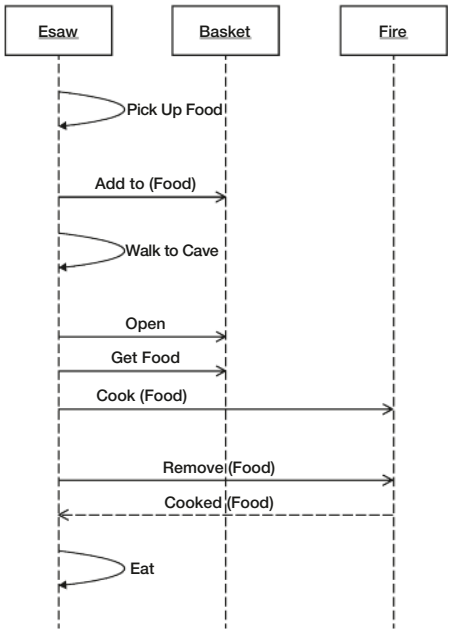


FIGURE **1.5** A single sequence diagram demonstrating how food is gathered and prepared.

State diagrams

Whereas interaction diagrams show objects and the messages passed between them, a *state diagram* shows the changing state of a single object as that object passes through a system. If we continue with our example, then we will focus on Esaw and how his state is changing as he forages for food, finds food, and consumes it (FIGURE **1.7**).

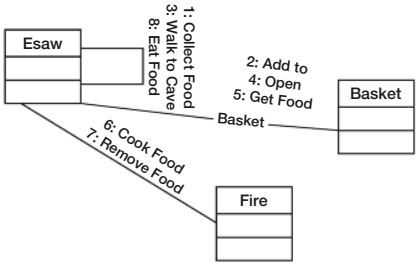


FIGURE **1.6** A collaboration diagram that conveys the same gathering and consuming behaviour.

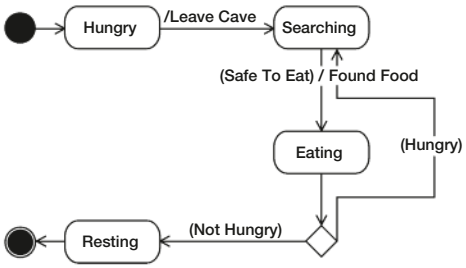


FIGURE **1.7** A state diagram (or state chart) showing the progressive state as Esaw forages and eats.

[P. Kimmel, "UML Demystified – a self-teaching guide", McGraw-Hill, 2005]

QUESTIONS

a **Which kind of UML diagram is used to document the macro requirements of a system?**
b **What are activity diagrams used for?**
c **Do class diagrams show dynamic interactions between classes?**
d **What are the two types of interaction diagrams?**
e **Is a state diagram an interaction diagram?**