
iss26

Release 1

Antonio Natali

Feb 16, 2026

CONTENTS:

1	Preludio	1
1.1	C'è ancora possibilità di lavoro?	1
2	Cosa si intende per Sistema	5
2.1	Descrivere e classificare	5
2.2	Sistemi semplici	6
2.3	Sistemi complicati	6
2.4	Sistemi complessi	6
2.4.1	Il termine 'emergenza'	6
2.5	Sistemi caotici	7
2.5.1	Macchine, agenti, organismi	7
2.6	Come si descrive un sistema	9
2.6.1	Visione BlackBox	9
2.6.2	Visione WhiteBox	9
2.6.3	Dimensioni	10
2.6.4	Visione sistemica	10
2.7	Perchè i sistemi complessi	10
2.7.1	La sinergetica	12
2.7.2	Teoria delle reti	12
2.8	Verso i sistemi software	13
3	Cosa vuol dire computare	15
3.1	Il motto di Alan Kay	15
3.2	Dai modelli ai linguaggi	16
3.3	Macchine astratte	16
3.4	La Macchina di Minsky	17
3.5	Problemi non riducibili (Indecidibili)	17
3.5.1	Il Teorema di Rice	18
3.6	Cosa significa 'costruire'	18
3.6.1	Computazionalmente completo	18
3.7	Teoria Algoritmica dell'Informazione	19
3.7.1	Applicazione all'Intelligenza Artificiale	21
3.8	Evoluzione del concetto di Computazione	21
3.8.1	Verso la definizione di 'sistemi che calcolano'	22
3.9	Famiglie di formalismi	23
3.9.1	Alcuni dei promotori	23
3.9.2	Formalismi funzionali	23
3.9.3	Formalismi concorrenti	24
3.9.4	Formalismi distribuiti	25
3.9.5	Formalismi per la mobilità	25

3.9.6	Formalismi stocastici	25
3.9.7	Formalismi field-based e global behavior	26
3.9.8	Formalismi per sistemi complessi e dinamiche non lineari	26
3.9.9	Oltre la macchina di Von Neumann	27
3.9.10	Quantum computer	27
3.10	Modelli formali: necessari ma non sufficienti	28
3.11	La coroutine un nuovo vecchio concetto	29
3.12	Un esempio: CSP e GO	30
3.12.1	CSP: Processi sequenziali	30
3.12.2	Come CSP ha influenzato Go	31
3.13	Modelli e linguaggi	32
4	I linguaggi	33
4.1	Linguaggi naturali	33
4.2	I Large Language Models	34
4.3	Linguaggi di programmazione	34
4.3.1	Una frase misteriosa	35
4.3.2	Sintassi e semantica	35
4.3.3	Linguaggi per descrivere linguaggi	35
4.3.4	La analisi sintattica	36
4.3.5	Descrivere la semantica	36
4.4	Compilatori e interpreti	37
4.5	General Processing Programming Languages	37
4.5.1	Stili (paradigmi) di programmazione	38
4.6	Gli attori come nuovo paradigma?	40
4.7	Non determinismo	40
4.8	Influenza di un LdP sul progettista del software	41
4.9	L'astrazione	41
4.9.1	Realtà vs. Modello	41
4.9.2	L'Abstraction gap	42
4.9.3	Modelli e Linguaggi di modellazione	42
4.9.4	UML	43
4.9.5	Estensioni di UML	45
4.10	Domain-Specific Languages (DSL)	45
4.10.1	DSL Interno	45
4.10.2	DSL Esterno	46
4.11	I DSL e i Digital Twins	47
4.12	Verso i sistemi software	47
5	Sistemi software	49
5.1	Viste di un sistema software	49
5.1.1	Vista esterna	50
5.1.2	Vista interna	50
5.1.3	Vista sommersa	50
5.2	Componenti software	50
5.2.1	Componenti software di base	50
5.2.2	Oltre gli oggetti	51
5.3	Un viaggio nei sistemi software	51
5.3.1	Punti di vista	51
5.3.2	Sistemi Embedded/IoT	52
5.4	Capire sperimentando	53
5.4.1	Capire interagendo	53
5.4.2	Interagire con un sistema software	53
5.4.3	Capire implica opportuni livelli di conoscenza	54

5.4.4	Conoscere per comunicare	54
5.5	Sistemi software come servizi (ma non solo)	55
5.5.1	L'infrastruttura	55
5.5.2	Le API	56
5.5.3	Representational State Transfer (REST)	57
5.5.4	Dai servizi ai microservizi	58
5.5.5	CoAP	58
5.5.6	WebSocket	59
5.6	Sistemi visti dall'interno	62
5.6.1	Design patterns	62
5.7	Architettura di un sistema	64
5.7.1	Definizione Strutturale	65
5.7.2	Definizione Pragmatica	65
5.7.3	Evoluzione delle architetture	65
5.7.4	Big Ball of Mud	66
5.7.5	Principi SOLID	66
5.7.6	Clean Architecture	67
5.7.7	Bounded Context	68
5.8	Costruiamo sistemi software	69
5.8.1	Il filo conduttore	69
5.9	Processi di costruzione del software	70
5.10	Il testing	71
5.10.1	Il nostro metodo di lavoro	72
6	Game of Life di Conway	75
6.1	Il gioco in JavaScript	76
6.2	Il gioco produce complessità	76
6.2.1	Il gioco è Turing-completo	77
6.3	Il problema della vista del gioco	77
6.4	Il gioco come caso di studio	78
6.4.1	Analisi del problema GofLife	78
6.4.2	Progetto ConwayLife26	78
6.4.3	LifeController	79
6.4.4	Il processo di evoluzione del gioco	79
6.4.5	La interfaccia IOutDev	80
6.4.6	La interfaccia GameController	80
6.4.7	MainConwayLifeJava.java	80
6.4.8	Una GUI con Swing	81
6.4.9	Configurazione di GofLife con Swing	81
6.4.10	Da un programma a un servizio	82

PRELUDIO

L'idea di **Ingegneria** pone enfasi sulla

costruzione consapevole e motivata di artefatti basata su analisi, progettazione, sviluppo, distribuzione e manutenzione di prodotti

Gli *ingegneri del software* sono abituati ad avvelersi di molti diversi strumenti per costruire *Sistemi software*: linguaggi di programmazione, framework, librerie, ambienti di sviluppo, etc.

Oggi esiste un nuovo, potente strumento: le Chat basate su *Intelligenza Artificiale (IA)*.

Sempre più spesso, le IA generative (come ChatGPT, Gemini, Claude, ecc.) vengono utilizzate per assistere gli sviluppatori nel processo di scrittura del codice, nella generazione di documentazione e molto altro.

Ad esempio, posso chiedere a ChatGPT di generare un programma che realizza il gioco *Conway Life*. A tal fine imposto il seguente **prompt**:

"Genera un programma che implementa il gioco di Conway Life.
Il programma deve utilizzare una griglia 10x10 e visualizzare lo stato della griglia ad ogni iterazione.
Includi commenti esplicativi nel codice."

- Il risultato di questa richiesta a **ChatGpt** è un programma *Python* riportato in *ConwayLife.py*.

Posso anche ottenere una versione in JavaScript, specificando questa richiesta nel prompt. Il file *conway25Gemini.js* riporta il risultato del codice generato da **Gemini**.

Modificando ulteriormente il prompt, è possibile generare programmi che eseguono il gioco in un browser web, utilizzando HTML5 e *canvas* per la visualizzazione grafica o che utilizzano librerie grafiche come *Pygame* per una visualizzazione più avanzata in Python.

1.1 C'è ancora possibilità di lavoro?

E' noto che quando una attività diventa riproducibile da una macchina, cessa di avere valore per gli esseri umani e, automaticamente cessano di avere valore le persone che continuano a svolgere quell'attività.

Dunque, la professione del costruttore di software rischia di essere messa in crisi. Proviamo a chiedere alla IA 'cosa ne pensa':

"L'avvento della IA generativa potrebbe mettere in crisi la professione del programmatore di computer.
Vorrei avere una panoramica delle ragioni che possono rendere meno probabile questo fenomeno o, meglio, le attività nel campo della costruzione del software che possono ancora essere svolte in modo migliore da un essere umano"

Note: Risponde Gemini

È vero che l'avvento dell'IA Generativa ha sollevato preoccupazioni sulla professione di programmatore.

Tuttavia, la maggior parte degli esperti concorda che l'IA sia destinata a diventare uno strumento di potenziamento piuttosto che un sostituto completo, soprattutto nelle fasi più critiche e astratte del ciclo di vita del software.

Ecco una panoramica delle attività in cui l'essere umano mantiene un vantaggio distintivo e in cui è meno probabile che venga sostituito dall'IA.

Comprensione del Dominio e Astrazione

L'IA generativa (come ChatGPT o Gemini o Copilot) eccelle nella **traduzione sintattica** (scrivere codice da istruzioni chiare), ma fallisce nella **comprensione semantica** profonda del dominio aziendale o scientifico.

- **Identificazione del Problema Reale:** Un ingegnere del software umano è fondamentale per interagire con il cliente, comprendere i requisiti ambigui, definire i *casi limite* (edge cases) e distinguere ciò che il cliente *vuole* da ciò di cui il cliente *ha bisogno*. L'IA può scrivere codice per una funzione, ma non può ancora guidare una riunione di *discovery* per definire i modelli di business.
- **Architettura e Design a Livello Olistico :** L'IA può generare codice per singoli componenti o **Microservizi**, ma la creazione di un'**architettura software** coerente, scalabile, e che rispetti vincoli di costo e tempo (*Design Patterns, Olarchia, Coreografia vs. Orchestrazione*) richiede un giudizio e una visione sistemica che l'IA non possiede.

Gestione della Complessità e Integrazione

I sistemi software reali sono raramente applicazioni *greenfield* (nuove da zero); sono complessi intrecciati di sistemi legacy, API di terze parti e vincoli operativi.

- **Integrazione di Sistemi Legacy:** Il codice generato dall'IA è spesso ottimale, ma si scontra con la necessità di integrarsi con un codice legacy vecchio di 20 anni, spesso non documentato e con *side effects* non ovvi. Solo un umano può **decifrare e mantenere** in modo sicuro queste interfacce complesse.
- **Debug di Sistemi Distribuiti:** Identificare la causa radice di un *bug* in un sistema distribuito (che coinvolge rete, database, latenza e interazioni asincrone) richiede ragionamento causale e ipotesi basate su esperienza, non solo sul pattern matching del codice.

Fattore Umano ed Etica

La programmazione non è solo un esercizio tecnico, ma un atto sociale, etico e di gestione del rischio.

- **Responsabilità Etica e Legale:** Chi è responsabile se un codice generato dall'IA causa danni (es. in un sistema finanziario o medico)? L'ingegnere umano mantiene la **responsabilità finale** per la validazione, il test e l'implementazione sicura del codice.
- **Leadership e Mentoring:** L'IA non può gestire un team, mediare conflitti, fare *mentoring- a sviluppatori junior*, o difendere una decisione tecnica davanti a stakeholder non tecnici. Queste sono soft skills cruciali nella professione.

Attività Specifiche in cui l'Umano eccelle

Ecco le aree dove il programmatore sposta il suo valore aggiunto:

Area di Attività	Ruolo Umano Distintivo
<i>Ingegneria dei Requisiti</i>	Interazione diretta con gli utenti, conversione di necessità vaghe in specifiche tecniche formali (definizione dell'obiettivo).
<i>Architettura Software</i>	Selezione del paradigma (OOP, Funzionale, Olonico), progettazione di schemi di database e definizione delle interfacce tra i macro-servizi.
<i>Validazione e Test Strategico</i>	Non solo scrivere unit test (che l'IA può fare), ma definire scenari di test complessi e non banali, test di carico e strategie di <i>chaos engineering</i> .
<i>Sicurezza e Hardening</i>	Identificazione di vulnerabilità logiche e attacchi a livello di business. L'IA può scrivere codice, ma un umano deve garantire la robustezza contro gli intenti malevoli .
<i>Innovazione e Ricerca</i>	L'IA è addestrata su dati esistenti (ciò che è stato fatto). La creazione di nuovi algoritmi , paradigmi o linguaggi di programmazione per risolvere problemi non ancora risolti sembra richiedere ancora creatività umana.

In conclusione, l'IA Generativa solleverà il programmatore dai compiti di scrittura di codice *boilerplate* e di *routine* (codifica), ma amplificherà la necessità di concentrarsi sulla **soluzione dei problemi**, sulla **gestione della complessità** e sul **giudizio etico-architettonale**.

La professione non scompare, ma si **eleva** da programmatore a **ingegnere del software di sistema**.

COSA SI INTENDE PER SISTEMA

Il termine ‘Sistema’ deriva da una parola greca che significa “ciò che è posto insieme” o “un insieme organizzato”.

Non esiste una singola definizione formale universalmente accettata di “**sistema**” poichè la parola evoca un concetto categoriale che assume significati diversi in fisica, ingegneria, biologia, sociologia, informatica.

Un prima, possibile definizione ‘informale’ può essere:

Un sistema è un insieme di elementi interconnessi o interagenti, organizzati in modo coerente per raggiungere un determinato scopo (o per svolgere una funzione specifica).`

2.1 Descrivere e classificare

Descrivere significa **selezionare e organizzare informazione** per rappresentare un oggetto o un sistema **in modo utile**, spesso *compattato*, privilegiando **relazioni e regole** rispetto a elenchi e dettagli.

Una descrizione può assumere la forma di una **classificazione**, cioè:

classificare = assegnare ogni elemento di un insieme a una categoria, in base a criteri scelti.

Dobbiamo essere consapevoli che ogni classificazione è **costruita**, e dipende dagli **scopi** e dalla **cultura** dell’osservatore. Non esistono categorie “naturali” e assolute: *decidere cosa distinguere e aggregare* è sempre **in-interpretazione**. Le categorie non “esistono” nelle cose. Le creiamo noi per raggiungere uno scopo.

Nel tentativo di comprendere il mondo, si è affermata una tassonomia molto diffusa che distingue i sistemi in *semplici, complicati, complessi e caotici*. Questa classificazione permette di riconoscere la *natura* di un sistema e di scegliere gli strumenti più adatti per analizzarlo o progettarelo.

Questa tassonomia serve non tanto per incasellare i sistemi, quanto per: guidare il modo in cui li **modelliamo**, quali strumenti formali utilizziamo e quali aspettative abbiamo sul loro comportamento.

A ciascuna delle categorie possiamo far corrispondere una diversa classe di fenomeni e diverse possibili azioni di governo.

2.2 Sistemi semplici

Caratteristica: comportamento prevedibile, descrivibile con poche regole lineari; facile da modellare e prevedere.

I sistemi semplici provocano fenomeni semplici. Il principio causa-effetto indica, a partire dall'effetto, qual è la causa che lo determina.

Azioni: Un'azione di intervento sul sistema consiste nell'applicare un protocollo risolutivo predefinito. E' il regno delle persone **executive** che applicano **Best Practices**. *Esempio:* hostess di un aereo.

2.3 Sistemi complicati

Caratteristica: Molti elementi e molte relazioni, ma ingegnerizzabile—decomponibile, prevedibile; molte possibili casue per un dato fenomeno.

Questi sistemi provocano fenomeni complicati, per i quali occorre studio e analisi.

Azioni: Un'azione di intervento consiste nell'analizzare le possibili cause, pianificare l'intervento risolutivo e realizzarlo. Sono necessarie persone **expert** capaci di definire a priori un modello di funzionamento, in base al quale pianificare l'intervento. *Esempio:* pilota di un aereo a motore.

2.4 Sistemi complessi

Caratteristica: Molti elementi, relazioni non lineari, adattamento, emergenza; il comportamento non è predicibile solo dalle parti, ma **emerge** dalle interazioni.

2.4.1 Il termine 'emergenza'

Con il termine **emergenza** intenderemo sempre indicare l'idea di

Affioramento/manifestazione di nuovi comportamenti o proprietà non riducibili alle proprietà o comportamenti di singoli componenti presi isolatamente.

Un sistema complesso non può essere 'controllato'; può al massimo essere 'perturbato'.

Azioni: Un'azione di intervento non può essere basata su un modello a-priori, ma richiede l'azione come scelta/scommessa, apprendimento (degli effetti) e adattamento continuo (**try and learn**). *Esempio:* pilota di un aliante.

E' l'azione (**agency**) costruttiva a generare conoscenza. Quando non si conosce il modello di comportamento di un fenomeno, **agire** significa sperimentare e questo per gli esseri umani (dice il biologo *Stuart Kauffman*) significa generare nuovi artefatti e modelli chiamando in causa competenze e approcci diversi, con una proliferazione di attività eterogenee.

2.5 Sistemi caotici

Caratteristica: Estremamente sensibile alle condizioni iniziali (effetto farfalla); può essere anche “semplice” dal punto di vista strutturale, ma impossibile da prevedere a lungo termine.

Per i sistemi che provocano fenomeni caotici non c'è mai ripetizione di alcuna traiettoria e non è possibile apprendimento.

Azioni: L'unica strategia è azione-adattamento continuo. *Esempio:* pilota che cerca di atterrare in condizioni meteo avverse.

2.5.1 Macchine, agenti, organismi

Per un ingegnere (del software, ma non solo) è anche utile introdurre una ulteriore distinzione/classificazione tra sistemi, distinguendo tra *macchine, agenti, agenti intelligenti e organismi*. Ma prima, premettiamo alcune puntualizzazioni:

- **Le categorie non sono necessariamente disgiunte né nette:** un sistema può trovarsi “a metà” tra macchina e agente oppure tra agente e agente intelligente, a seconda delle caratteristiche reali.
- **Dipendenza dalla scala di osservazione:** un “organismo” visto dall'esterno può mostrarsi come “molti agenti intelligenti” o come “un agente intelligente”: la classificazione dipende da **chi osserva, come osserva, e perché**.
- **Arbitrarietà e finalità:** come per ogni classificazione, le categorie riflettono aspettative progettuali o analitiche: cambiano se vogliamo modellare un software, un sistema biologico, un network sociale, un ecosistema.
- **Pericolosità di antropomorfismi:** attribuire “intelligenza”, “vita”, “autonomia” a sistemi software è spesso metaforico: bisogna chiarire cosa intendiamo con quei termini, per evitare ambiguità o aspettative irrealistiche.

Macchina

Una macchina è un sistema chiuso, deterministico e creato artificialmente per raggiungere un obiettivo specifico.

Sistema fatto di parti che esistono l'una per l'altra.

Spesso non varia il proprio comportamento se non in modi predeterminati.

Le relazioni tra le parti sono lineari e l'analisi è di tipo scompositivo (riduzionistico). La modifica di una macchina è di tipo reattivo (si aggiunge o toglie una parte)

Sistema cibernetico

Un sistema cibernetico è un sistema che regola il proprio comportamento tramite meccanismi di controllo e retroazione (feedback), adattandosi alle variazioni dell'ambiente per mantenere uno stato o conseguire uno scopo.

Il termine cibernetica deriva da una parola greca che significa “timoniere”, “colui che guida”.

Norbert Wiener (1894-1964), matematico statunitense, è considerato il padre della cibernetica. Per Wiener, ciò che accomuna sistemi naturali e artificiali è la capacità di:

percepire → confrontare → correggere

Il ciclo percezione → controllo → azione → percezione è l'essenza del sistema cibernetico, che offre modelli generali di regolazione e controllo capaci di unificare biologia, ingegneria, matematica, neuroscienze, formalizzando il concetto di **feedback**.

Agente

Sistema (hardware-software) che interagisce con un ambiente: percepisce input (sensori o messaggi), produce output (attuatori o messaggi), ha un ciclo percezione - azione.

Opera in modo autonomo e prende decisioni senza essere controllato.

Ogni agente è un sistema cibernetico avanzato. Non ogni sistema cibernetico è un agente.

L'idea di Russell & Norvig è che un agente sia un sistema che mappa percezioni in azioni per massimizzare una misura di performance.

Questa definizione generalizza l'idea cibernetica: i sistemi di Wiener erano soprattutto regolatori; gli agenti introducono scelte orientate al futuro.

Occorrono precise assunzioni:

- Esistenza di un **ambiente con regolarità**: se l'ambiente fosse completamente caotico *nessuna strategia sarebbe possibile*
- Esistenza di **obiettivi (telos** Aristotelico) e metriche di successo: un agente senza obiettivo è una macchina. Gli obiettivi possono essere espliciti (goal logici), normativi, funzioni di utilità nella Reinforcement Learning (RL)
- Capacità di **previsione** e (spesso) **modello del mondo**: **modelli espliciti** (mondo rappresentato → **GOFAI** (*Good Old-Fashioned Artificial Intelligence*)), **stimatori statistici** (policy without model → **RL** (*Reinforcement learning*)), **pattern reattivi** (modello implicito → **Stigmergia**)

Il modello può essere **interno o distribuito**. Nelle società di agenti l'ambiente stesso può essere *parte della memoria collettiva*.

Agente intelligente

Un agente dotato di capacità di decisione flessibile, adattamento, ragionamento o apprendimento: cioè un sistema che, oltre a reagire, elabora conoscenza, valuta opzioni, può perseguire obiettivi.

Risulta efficace in una varietà di situazioni/ambienti.

Una definizione data da **Albus** nel 1991 è: agente capace di agire in modo appropriato in un ambiente incerto (ma regolare) dove le azioni appropriate sono quelle che aumentano la possibilità di successo.

Organismo

Un organismo è un sistema aperto, dinamico che interagisce con l'ambiente e si adatta ad esso.

Sistema (vivente) le cui parti esistono anche l'una per mezzo dell'altra.

Esibisce *autopoiesi*, *omeostasi*, *adattamento evolutivo*, *autorganizzazione*: un insieme di sottosistemi integrati, capace di mantenerne l'identità, di auto-regolarsi, di evolvere.

Le relazioni tra le parti sono non lineari, con feedback complessi e l'analisi è di tipo olistico (si privilegia il tutto).

La modifica di un organismo è di tipo proattivo/evolutivo (si auto-organizza).

2.6 Come si descrive un sistema

La descrizione di un sistema può essere affrontata da diverse prospettive: una visione interna (analitica o **WhiteBox** o *come è fatto*), una visione esterna (o **BlackBox** o *come si usa*) e una visione **globale** o *sistemica* per sistemi complessi, che descrive le proprietà ‘emergenti’.

2.6.1 Visione BlackBox

La ‘visione’ dall’esterno di un sistema è legata al modo con cui è possibile interagire (scambiare informazione) con il sistema.

- La semantica Black Box di un **sistema fisico** è la funzione matematica o l’equazione differenziale che lega l’Input all’Output nel tempo.

Una funzione del tipo $O(t) = f(I(t))$ definisce il significato del sistema per l’osservatore. Questo approccio è fondamentale per la Progettazione di *Sistemi di Controllo*, dove il controllore (l’osservatore) deve interagire con la Black Box basandosi esclusivamente sui segnali di ingresso e uscita per manipolare il suo comportamento.

- La semantica Black Box di un **sistema software**, è legata all’output emesso dal sistema sollecitato da un certo input; un osservatore del sistema non è interessato quanti thread Java, quali protocolli di routing o quali algoritmi il sistema utilizza internamente.

L’interazione è legata al concetto di **Interfaccia Pubblica** del sistema (ad esempio, le *Le API* di un servizio, la *Interface* di un object, la *signature* di una funzione). Questa interfaccia funge da contratto tra il sistema e il mondo esterno, stabilendo l’unico modo permesso per scambiare informazioni.

L’atto di nascondere i dettagli implementativi ed esporre solo il comportamento essenziale è noto come **As-trazione** o Occultamento dell’Informazione (**Information Hiding**).

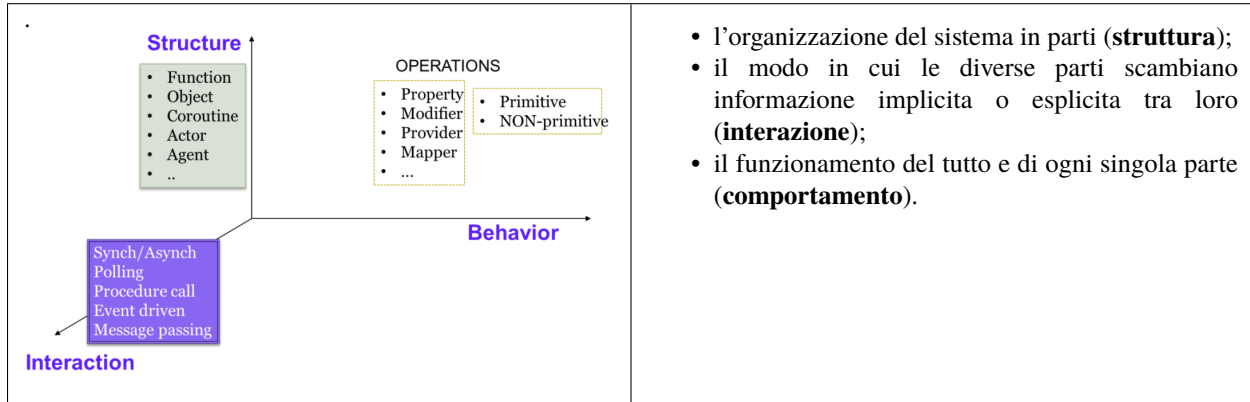
2.6.2 Visione WhiteBox

Per descrivere un sistema ‘dall’interno’ occorrono **almeno tre dimensioni**: la dimensione **strutturale**, la dimensione della **interazione** e la dimensione del **comportamento** dei singoli componenti.

Struttura	Descrive la disposizione fisica o logica degli elementi; i nodi e i collegamenti. Risponde a: Quanti sono? Come sono connessi?	Distingue tra sistemi semplici (poche connessioni) e sistemi complicati o complessi (molte connessioni).
Comportamento	Descrive Le regole interne e le capacità di elaborazione di ogni singolo elemento del sistema. Risponde a: Cosa fa ogni parte?	Distingue i componenti rigidi/prevedibili da quelli autonomi/adattativi (essenziale per l’autoorganizzazione).
Interazione	Descrive la natura, la forza e la frequenza degli scambi di informazione/energia tra i componenti. Risponde a: Come si influenzano a vicenda?	Cruciale per distinguere complicato (interazioni lineari e fisse) da complesso (interazioni non lineari e dinamiche) e caotico (estrema sensibilità).

2.6.3 Dimensioni

Un sistema software si può quindi pensare collocato in uno spazio concettuale a tre dimensioni specificato attraverso uno dei tanti *Linguaggi di programmazione*:



2.6.4 Visione sistemica

Questo livello descrive le proprietà che appaiono solo quando il sistema viene visto nel suo insieme, quali la resilienza, l'adattabilità, il grado di prevedibilità e le proprietà emergenti.

E' un livello di descrizione tipico dei sistemi complessi, indipendente dal tipo di componente. Pone il focus su pattern di interazione, topologie, coordinazione, flussi informativi.

More is different

Quando un sistema cresce in dimensione, diventa diverso in natura. *Philip W. Anderson* sintetizza già nel 1972 questa idea con il celebre motto “**More is different**”. L'aumento del numero di elementi porta all'emergere di proprietà globali non prevedibili analizzando i singoli componenti.

Le proprietà di un insieme di molti elementi non sono riducibili alla somma delle proprietà dei singoli elementi.

Tali proprietà richiedono un diverso linguaggio descrittivo, focalizzato sulle interazioni e sugli schemi collettivi, aprendo la strada a una visione sistemica del software. Per sistemi complessi software — come gli ecosistemi di servizi distribuiti o gli insiemi massivi di attori — questa visione è oggi indispensabile per comprendere e progettare comportamenti emergenti, robustezza e adattamento.

La frase di Anderson diventa anche un principio di progettazione:

Quando i componenti diventano molti, ciò che conta non è cosa fanno, ma come interagiscono

2.7 Perché i sistemi complessi

Per un ingegnere del software è necessario avere consapevolezza di cosa sia un sistema complesso perché:

- i sistemi software moderni non sono solo più macchine gerarchiche e deterministiche: sono ecosistemi di servizi, microservizi, agenti, attori, componenti autonomi.
- il software moderno è quindi distribuito, eterogeneo, autonomo, adattivo, connesso in rete: cioè possiede molte delle proprietà osservate nei sistemi complessi naturali.

Mentre il ‘pensiero sistemico’ iniziò la sua ascesa negli anni Venti del secolo scorso, la scienza ha cominciato a parlare esplicitamente e a fondo di “sistemi complessi” con le teorie di *Prigogine* (anni Settanta) e con l’emergere della matematica non lineare (caos, frattali, dinamica dei sistemi, formalizzata dagli anni Ottanta in poi), portando alla nascita della **Scienza della Complessità** negli anni Novanta.

Si può usare la metafora di una montagna: per molti anni la scienza ha concentrato l’attenzione sulla “**fisica dell’ordine**” (sistemi semplici) e sulla “**fisica del disordine**” (sistemi con moltissime variabili non correlate), lasciando la “complessità organica” (la vetta del monte) in gran parte inesplorata fino alla fine del XX secolo.

La possibilità di *scalare la vetta della complessità* si è aperta con l’avvento dei computer e delle nuove tecniche matematiche, che hanno permesso di esplorare la “scala mesoscopica” dei fenomeni biologici e sociali.

Alcuni tratti ricorrenti in questo nuovo campo disciplinare sono riassunti nella tabella che segue:

Tratto distintivo	Significato
Molte entità interagenti	Un sistema è composto da un gran numero di componenti autonomi che agiscono localmente. Esempi: Formicai, reti di microservizi, reti neurali biologiche o artificiali.
Interazioni non lineari	Le interazioni non si sommano in modo proporzionale: piccole cause possono produrre grandi effetti e viceversa. Esempi: Mappa logistica, epidemie, amplificazione in reti sociali.
Retroazioni (feedback)	Gli effetti delle azioni dei componenti ritornano a influenzarli, creando cicli di amplificazione o stabilizzazione. Esempi: Controllo ormonale, circuiti di feedback nei sistemi autonomi e nei sistemi ecologici.
Auto-organizzazione	L’ordine globale emerge spontaneamente dalle interazioni locali, senza un controllo centralizzato. Esempi: Sincronizzazione delle lucciole, flocking degli uccelli, clustering spontaneo in robotica swarm.
Transizioni ordine/disordine	Il sistema può passare bruscamente da stati stabili a stati caotici o viceversa, spesso in corrispondenza di soglie critiche. Esempi: Transizioni di fase nella materia, collasso di reti infrastrutturali, criticalità auto-organizzata.
Emergenza di pattern globali	Comportamenti collettivi non prevedibili dalle singole parti, ma derivanti dalle loro interazioni. Esempi: Pattern del Game of Life, onde di traffico, comportamenti collettivi nei social network.

Inoltre:

Tratto distintivo	Significato
Adattamento	Il sistema modifica la propria struttura o comportamento in risposta all’ambiente o a pressioni interne. Esempi: Algoritmi genetici, ecosistemi, sistemi software self-adaptive.
Sensibilità alla topologia delle interazioni	La forma della rete di connessioni influisce profondamente sulla dinamica del sistema. Esempi: Reti small-world (diffusione rapida), reti scale-free (robuste ma fragili ai nodi-hub), reti multilivello.
Dinamiche collettive lontane dall’equilibrio	Il sistema opera in condizioni non stazionarie, mantenute da flussi costanti di energia, materia o informazione. Esempi: Reazioni chimiche oscillanti (Belousov–Zhabotinsky), mercati finanziari, sistemi distribuiti event-driven.

2.7.1 La sinergetica

Una delle cornici interpretative fondamentali per comprendere la natura dei sistemi complessi è la **sinergetica**, introdotta da **Hermann Haken**, l'inventore del *Laser*.

La sinergetica descrive come l'auto-organizzazione emerga tramite parametri d'ordine, modi instabili e il principio di asservimento, secondo cui il comportamento collettivo domina e coordina quello dei singoli componenti. E' uno dei "ponti" più eleganti tra:

- fenomeni fisici quali: auto-organizzazione, emergenza, dinamiche collettive, transizioni ordine/disordine, asservimento

e

- l'ingegneria di sistemi autonomi e distribuiti, in cui un insieme di microservizi o attori autonomi può generare spontaneamente *parametri d'ordine* quali sincronizzazione, consenso, oscillazioni, propagazioni, *asservimento* tramite orchestrazione e coreografia, *transizioni critiche* collegabili alla resilienza dei microservizi, ai circuit breakers, alle cascading failures, alle reti scale-free.

Il video: [Il Sorprendente Segreto della Sincronizzazione](#) fornisce una panoramica di alcuni interessanti (e fondamentali) fenomeni di sincronizzazione naturale. Se ne consiglia vivamente la visione.

2.7.2 Teoria delle reti

Un modo ritenuto efficace per descrivere sistemi (complessi) e la loro dimensione globale/emergente è ricorrere ai modelli di rete (o teoria dei grafi).

Griglie e grafi

Una **griglia**, o reticolo regolare (lattice), è un tipo specifico di grafo caratterizzato da una topologia altamente regolare e locale.:

- ogni nodo è collegato allo stesso numero fisso di vicini (es. 4 in una griglia 2D, 6 in una 3D)
- l'informazione si diffonde lentamente e in modo prevedibile
- non mostrano un'emergenza forte o un comportamento a "piccolo mondo"; descrivono sistemi complicati o al massimo semplici-complessi.

Un **grafo** generico non ha restrizioni sulla sua topologia

- il numero di connessioni per nodo varia enormemente
- possono esistere "salti" o hub che collegano nodi lontani
- descrivono sistemi complessi e autoorganizzati. Mostrano emergenza, resilienza e vulnerabilità (dipendente dalla topologia).

La teoria delle reti basata su grafi è la **grammatica formale della complessità**, e permette di spostare la discussione dalla specificità dei componenti (Java, router, gene) alla topologia che determina il comportamento globale (complesso, autoorganizzato, caotico).

Ad esempio, modelli che descrivono come l'organizzazione (l'ordine) emerge in reti complesse sono:

1. Reti Casuali (*Erdős-Rényi*)

Queste reti servono da base di confronto. Le connessioni sono disposte in modo puramente casuale.

- **Limitazione:** Non descrivono adeguatamente la maggior parte dei sistemi reali (Internet, reti biologiche, reti sociali) che hanno strutture molto più organizzate.

2. Reti a Piccolo Mondo (*Watts e Strogatz*)

Le **Small world network** hanno un alto coefficiente di clustering (i miei amici sono amici tra loro, come nelle reti sociali) ma anche una corta distanza media tra i nodi (di veda la proprietà dei **sei gradi di separazione**).

- **Significato:** Favoriscono la diffusione efficiente dell'informazione o delle malattie.

3. Reti Scale-Free (*Barabási-Albert*)

Questo è il modello che meglio descrive Internet e molte reti biologiche.

- **Caratteristica:** La distribuzione dei gradi dei nodi segue una **legge di potenza** (**powerlaw**). Ci sono pochissimi nodi con un numero estremamente elevato di collegamenti (gli hub) e molti nodi con pochissimi collegamenti.
- **Meccanismo:** Nascono da un meccanismo di **preferential attachment** (*effetto San Matteo*: “i ricchi diventano più ricchi”).
- Proprietà Emergenti:
 - **Robustezza:** La rete è estremamente resistente al fallimento casuale dei nodi (perché la maggior parte dei nodi non sono hub).
 - **Vulnerabilità:** È estremamente vulnerabile a un attacco mirato contro gli hub.

2.8 Verso i sistemi software

La citata progressiva evoluzione dei sistemi software verso ecosistemi costituiti da molteplici componenti autonomi che possono presentare proprietà simili a quelle dei sistemi naturali, non invalida il fatto che il software nasce in relazione alla costruzione di macchine universali capaci di ‘computare’.

Ciò che si modifica è il **concetto stesso di ‘computazione’** e a questo tema verrà dedicato il capitolo successivo.

COSA VUOL DIRE COMPUTARE

Per descrivere qualcosa occorre un linguaggio.

La necessità di modellare l'imprevedibile e l'emergente ha richiesto, nel caso dei sistemi fisici, nuovi strumenti matematici e, nel caso dei sistemi software, nuovi paradigmi computazionali.

Per i **sistemi fisici**, la *nuova matematica* è basata su dinamica non lineare (attrattori, biforcazioni, caos), automi cellulari e sistemi discreti, teoria dei frattali, teoria delle reti complesse, sinergetica. Questi formalismi non sostituiscono la matematica classica, ma la estendono e la rendono applicabile a fenomeni emergenti con molti gradi di libertà.

In modo analogo, per i **sistemi software** l'approccio classico alla progettazione e costruzione di sistemi (*OOP* + *UML* + programmazione sequenziale/imperativa) risulta insufficiente per affrontare problemi in cui vi sono molti componenti che operano come enti autonomi che interagiscono scambiandosi messaggi, in assenza di memoria comune.

Al concetto di **oggetto** come tipico componente-base per molti linguaggi general purpose (**GPL**) occorre affiancare oggi concetti quali **attore** e **agente** come enti autonomi che interagiscono a messaggi. Questi concetti permettono di 'disgregare' le tradizionali **architetture monolitiche** dei sistemi a favore di architetture distribuite, quali quelle, oggi molto diffuse, basate su **microservizi**.

3.1 Il motto di Alan Kay

Come ingegneri del software, è necessario capire i dettagli e le motivazioni della continua evoluzione dei linguaggi di programmazione, assumendo il quadro concettuale di riferimento enfatizzato da **Alan Kay** (il promotore del linguaggio **Smalltalk**, uno dei progenitori della *OOP*):

Un linguaggio di programmazione è come ali per i nostri pensieri,
ma è anche le catene che non ci permettono di volare più **in** alto.

Questo riassume l'idea che i linguaggi (di programmazione, ma non solo) ci consentono di **esprimere concetti** che altrimenti sarebbero inesprimibili (le **ali**), ma le loro limitazioni intrinseche (la loro sintassi/semantica) possono impedirci (le **catene**) di concepire soluzioni più radicali o potenti.

3.2 Dai modelli ai linguaggi

L'evoluzione dei linguaggi di programmazione (LdP) si fonda su una lunga storia di idee matematiche che hanno cercato di definire in modo preciso che cosa significhi **calcolare**. Prima ancora che il software esistesse come disciplina autonoma, matematici, logici e filosofi si trovarono di fronte a un quesito fondamentale: **esiste un limite intrinseco a ciò che una procedura meccanica può fare?**

La risposta a questa domanda richiedeva di esplicitare che cosa fosse, esattamente, una “procedura meccanica”. Ovvero: un **modello formale** del calcolo.

È in questo contesto che nasce, negli **anni '30** del Novecento, la **Macchina di Turing** (proposta da [Alan M. Turing](#)), insieme ad altri formalismi equivalenti proposti quasi in parallelo:

- il **λ calcolo** di Alonzo Church,
- le **funzioni ricorsive** di Gödel, Herbrand e Kleene,
- i **sistemi di riscrittura** di Post e Markov.

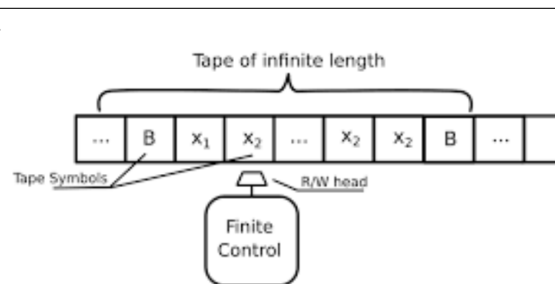
Pur diversi nella forma, questi modelli condividono una sorprendente convergenza: definiscono tutti la stessa classe di funzioni calcolabili. Da qui il **teorema di Church-Turing** e la tesi filosofica omonima: tutte le nozioni ragionevoli di “calcolo effettivo” coincidono.

Questa prima stagione dei formalismi nasce dunque **prima dei linguaggi di programmazione**, e non per scopi ingegneristici, ma **per rispondere a domande di logica matematica e fondamentali**: cosa può essere calcolato? cosa non può esserlo? cosa significa “*algoritmo*”?

I primi linguaggi di programmazione, come Fortran, Lisp, Algol, arriveranno *dopo*, come reinterpretazione pragmatica dei formalismi astratti. In alcuni casi (come Lisp) il riferimento alla teoria è diretto; in altri (come C o Java) è implicito.

3.3 Macchine astratte

Spesso si qualifica il comportamento di un componente facendo riferimento a tipi diversi di *macchine astratte*, iniziando dalla **Turing Machine (TM)**, che individua, in stile imperativo, le mosse-base (*primitive*) di un automa che permette di realizzare qualunque calcolo (!).



Un tipo meno potente di macchina astratta rispetto alla **TM** è il **pushdown automaton (PDA)** che però è l'automa di riferimento per il riconoscimento delle frasi di un linguaggio con *sintassi context-free*, di tipo 2 nella *gerarchia di Chomsky*, che introdurremo nella sezione *Macchine astratte per riconoscere linguaggi*.

3.4 La Macchina di Minsky

La macchina di Turing (TM) definisce un insieme di mosse con riferimento a un nastro e una testina. Una classe di macchine astratte che si dimostrano equivalente alla TM è quella delle macchine a registro o *Counter machine*. Una macchina di questa classe possiede un numero finito (N) di registri (contatori) (chiamati R_1, R_2, \dots, R_N).

La *Macchina di Minsky* è stata proposta da *Marvin Minsky* nel **1961**, dimostrando soli due registri ($N=2$) bastano per simulare qualsiasi TM (e quindi eseguire qualsiasi algoritmo).

La *Macchina di Minsky* è caratterizzata da solo due tipi di istruzioni:

INC(R_i, j)	Incrementa il registro R_i di 1. Salta all'istruzione j . (Senza condizione)
DEC_JUMP(R_i, j, k)	Se il registro $R_i > 0$, decrementalo di 1 e salta all'istruzione j . Altrimenti (se $R_i = 0$), salta all'istruzione k

Notiamo come questo modo di esprimere le operazioni elementari che stanno alla base del calcolo è più vicina alla nostra idea di computer come macchina capace di eseguire un insieme di mosse elementari, funzionando come **interprete** del *Linguaggio assembly*.

3.5 Problemi non riducibili (Indecidibili)

Vi sono problemi per cui è stato dimostrato che una TM (o formulazioni equivalenti) **non** può produrre una risposta in un tempo finito. Tra i più noti di questi ricordiamo:

- Il Problema dell'Arresto (**Halting Problem**): Data una descrizione di un programma (o TM) e il suo input, esisterà un algoritmo in grado di determinare se quel programma terminerà (si arresterà) o continuerà a funzionare indefinitamente (loop infinito)?
- Il Problema della Decidibilità (**Entscheidungsproblem**): Esiste un algoritmo che possa decidere, data una proposizione logica espressa nella logica del primo ordine, se quella proposizione è universalmente valida (vera in ogni modello)?
- Equivalenza dei Linguaggi Liberi dal Contesto (**CFG Equivalence**): Date due grammatiche libere dal contesto (CFG), generano esattamente lo stesso linguaggio?

L'*Entscheidungsproblem* fu posto da *David Hilbert* nel 1928. Sotto la guida di *Hilbert*, *John von Neumann*, nei suoi anni giovanili, si fece portabandiera dell'approccio assiomatico della matematica, riconoscendo subito il valore dei contributi di *Kurt Gödel* (Teoremi di incompletezza) che evidenziarono l'impossibilità di conseguire una dimostrazione completa della coerenza dell'aritmetica nel contesto del pensiero matematico.

L'indecidibilità di un problema P_2 viene di solito dimostrata riducendolo a un problema P_1 che è già noto come indecidibile.

La maggior parte delle prove di indecidibilità nel campo dei linguaggi di programmazione o delle proprietà dei programmi fa riferimento a un potente strumento formale basato sul concetto di riduzione: il *Teorema di Rice*.

3.5.1 Il Teorema di Rice

Il Teorema afferma che:

Qualsiasi proprietà *non banale* sul linguaggio (o sulla funzione) calcolato da una Macchina di Turing è indecidibile.

- **Proprietà:** Una caratteristica che un programma può avere (es. “termina sempre”, “stampa sempre 10 numeri”, “non entra mai in loop”).
- **Non banale:** Una proprietà che non è posseduta da tutti i programmi né da nessun programma. (Ad esempio, “essere un programma” è banale; “essere un programma che calcola π ” è non banale).

Esempio: Vogliamo dimostrare che decidere se un programma C++ stamperà “Hello World” è indecidibile.

Definiamo la proprietà P: “Il programma stampa ‘Hello World’”. P è non banale (esiste un programma che lo fa, e uno che non lo fa). Per il Teorema di Rice, la decisione se un dato programma possiede la proprietà P è indecidibile. Non si può costruire un algoritmo generale per questo.

3.6 Cosa significa ‘costruire’

Dunque, i formalismi servono non solo per costruire, ma anche per **sapere cosa possiamo o non possiamo ottenere**:

Questi limiti sono fondamentali per gli ingegneri: **modellano il territorio** in cui è possibile progettare.

Sulla base dei concetti introdotti, possiamo cercare di catturare meglio il significato del termine ‘costruire’ nell’ambito del software:

Costruire significa combinare elementi di base secondo regole definite per ottenere nuove entità o comportamenti.
Quando queste combinazioni sono lineari, otteniamo strutture prevedibili.
Quando invece le interazioni tra componenti generano proprietà non deducibili dai singoli, la costruzione diventa sorgente di complessità ed emergenza.

Notiamo che gli elementi-base nella costruzione di sistemi software sono le istruzioni interpretabili da un computer, che, per la *tesi di Church Turing*, definiscono anche di fatto ciò che è ‘costruibile’ mediante il software.

3.6.1 Computazionalmente completo

In informatica teorica si usa spesso il concetto di **Turing completezza**:

Note: Un linguaggio è Turing-completo (*Computazionalmente completo*) se può simulare qualsiasi macchina di Turing.

Questo significa che:

- può **leggere** e **scrivere** memoria arbitraria
- può eseguire **controllo del flusso** (cicli, ramificazioni, ricorsione)
- può comporre operazioni in modo **illimitato**

I GPL conservano queste proprietà anche quando offrono la capacità di costruire sistemi basati su componenti quali oggetti attori, etc. Ciascun componente viene *alla fine* eseguito (grazie ai compilatori e interpreti) da istruzioni macchina e dunque mantiene la Turing-completezza del sistema.

Ci sono modelli di calcolo o anche:ref:Domain-Specific Languages (DSL) **deliberatamente** non universali, per tre motivi principali:

- **Verificabilità** : SL reattivi senza ricorsione → analisi statica garantita
- **Sicurezza**: Smart Contracts (e.g. [Tezos Michelson](#)) → prevenzione infinite loops
- **Semplificazione concettuale** DSL dichiarativi per configurazione (SQL senza ricorsione)

3.7 Teoria Algoritmica dell'Informazione

La **Teoria Algoritmica dell'Informazione (AIT)** può essere vista come la versione “metrica” della teoria di Turing. Se Turing si occupava di ciò che può essere fatto (**possibilità**), la AIT si occupa di quanto costa farlo in termini di bit (**complessità**). In sintesi:

Teoria della Computabilità	Teoria Algoritmica dell'Informazione
Si chiede: “Esiste un algoritmo per questo?”	Si chiede: “Qual è l'algoritmo più breve per questo?”
Si concentra sull'esistenza del calcolo.	Si concentra sulla compressione del calcolo.
Risultato chiave: Il Problema dell'Arresto.	Risultato chiave: L'Incomputabilità della Complessità.

Fondata negli anni '60 da *Andrej Kolmogorov*, *Gregory Chaitin* e *Ray Solomonoff*, la **AIT** definisce l'informazione non come una probabilità statistica (come faceva Shannon), ma come **struttura e contenuto**. Riportiamone i punti salienti:

- **pattern**

Nell'AIT, un oggetto è considerato “strutturato” (e quindi contenente un pattern) se può essere compresso. Data un stringa di dati X :

- Se X è casuale, la sua descrizione più breve è la stringa stessa (non c'è pattern).
- Se X contiene un pattern, esiste un algoritmo/programma (P) molto più corto di X che può generarlo. Questo è il modello formale di ciò che il filosofo **Daniel Dennett** chiama “leverage” (leva predittiva): il pattern è reale perché permette di “risparmiare” informazione senza perdere il contenuto essenziale.

- **La Complessità di Kolmogorov**

Il concetto cardine è che la complessità di un oggetto (come una configurazione del gioco Conway Game of Life) è definita dalla **lunghezza del programma più breve che lo può produrre**.

$$K(s) = \min \{ |p| : U(p) = s \}$$

Dove s è la stringa, p è il programma e U è una macchina di Turing universale.

- **Bassa Complessità (Senso)**: Una griglia piena di “Block” stabili ha una K molto bassa, perché si può scrivere: “*Disegna un quadrato ogni N pixel*”.
- **Alta Complessità (Caos)**: Una griglia di rumore casuale ha una K altissima, perché l'unico modo per descriverla è elencare lo stato di ogni singolo cella: “*Cella 1 viva, 2 morta, 3 morta...*”.

La Complessità di Kolmogorov K stabilisce un limite oggettivo alla comprimibilità di un sistema. Anche se diversi osservatori usano diversi linguaggi di programmazione, la lunghezza della descrizione più breve differisce solo per una costante fissa.

- **La Casualità come Incomprimibilità**

Per l'AIT, un oggetto è **casuale** se non può essere compresso. Se non trovi un algoritmo più breve della stringa stessa per descriverla, allora quella stringa non ha “senso” o “struttura” logica; è rumore puro.

Questo si ricollega al fatto che il “senso” emerge quando l'osservatore trova una **compressione algoritmica** (una regola) in ciò che vede. Vedere un “Glider” nel Conway Game of Life è un atto di compressione: invece di bit sparsi, si ‘vede’ un'unica “entità” che si muove.

- **L'Induzione di Solomonoff e il Rasoio di Occam**

Solomonoff ha usato l'AIT per formalizzare matematicamente il Rasoio di Occam. La teoria dice che:

“Tra tutte le ipotesi che spiegano i dati osservati, quella più probabile è la più corta (quella con la minore complessità algoritmica).”

Nel Conway Game of Life, se si vede uno schema che sembra muoversi, il cervello sceglie l'ipotesi “È un oggetto semovente” perché è algoritmicamente più semplice che pensare “Sono centinaia di pixel indipendenti che cambiano stato per puro caso in modo coordinato”. **Il ‘senso è la spiegazione più economica.**

- **Il Numero Omega di Chaitin**

Gregory Chaitin ha portato la teoria verso l'ignoto con , la “probabilità di arresto”. Ω è un numero reale che esprime la probabilità che un programma generato casualmente si fermi.

- Ω è **incalcolabile** e racchiude in sé la soluzione a ogni problema matematico.
- Rappresenta il limite estremo della conoscenza: esiste una saggezza (informazione) che nessuna logica può estrarre in modo sistematico.

Senza un osservatore che cerchi la compressione, l'universo è solo una stringa incompressibile di bit casuali.

- **Il Problema della “Non-Computabilità”**

Un punto critico discusso nella letteratura del 2025 è che la complessità di Kolmogorov è non computabile (non esiste un algoritmo generale per trovarla sempre). Questo riflette l'idea di Dennett secondo cui non esiste una “prospettiva divina” per identificare tutti i pattern; la loro scoperta dipende dalla nostra capacità (limitata ma efficace) di trovare scorciatoie matematiche e predittive nel caso dei dati.

- **La Complessità di Kolmogorov come Misura della Realtà**

- **Definizione di Pattern:** Negli studi recenti, un pattern dennettiano è definito come una descrizione compressa dei dati. Se un sistema ha una complessità di Kolmogorov $K(x)$ significativamente inferiore alla lunghezza dei dati grezzi, allora quel sistema contiene un “real pattern”.
- **Ontologia algoritmica:** La realtà di un'entità (come un “glider” nel Gioco della Vita) non dipende dalla sua sostanza, ma dal fatto che l'algoritmo per descrivere il sistema diventa drasticamente più corto se includiamo quel pattern nella nostra “ontologia”.

- **Il Modello del “Minimum Description Length” (MDL)**

Nel volume del 2026 di Dennett *Real Patterns in Science and Nature* (MIT Press), i ricercatori utilizzano il principio del *Minimum Description Length* per risolvere il dibattito tra realismo e strumentalismo:

- **Pattern e Rumore:** MDL permette di bilanciare la precisione (aderenza ai dati) e la semplicità (brevità della descrizione).
- **Criterio di Oggettività:** Un pattern è considerato “reale” se e solo se la sua inclusione nel modello riduce la lunghezza totale della descrizione, pur “pagando” il costo di ignorare un po' di rumore.

3.7.1 Applicazione all'Intelligenza Artificiale

Ricerche pubblicate tra il 2025 e l'inizio del 2026 collegano i pattern di Dennett al funzionamento dei *Large Language Models* (LLM):

- **Emergenza:** Le capacità di ragionamento “zero-shot” dei modelli transformer sono interpretate come l'estrazione di pattern reali (strutture semantiche e logiche) da moltitudini di dati grezzi.
- **Codifica Predittiva:** La mente umana e l'IA sono viste come “motori di compressione” che estraggono pattern reali per minimizzare l'errore di previsione, una tesi che unifica l'epistemologia di Dennett con le neuroscienze computazionali moderne.

3.8 Evoluzione del concetto di Computazione

Con l'evoluzione dei computer, è diventato evidente che i sistemi software reali non si riducono a funzioni matematiche che trasformano un input in un output.

Dal 1940 al 1970, il computer era visto essenzialmente come l'implementazione fisica della Macchina di Turing o della Macchina di Von Neumann. A un computer veniva affidato il compito di eseguire un **Algoritmo**, cioè una sequenza finita e ben definita di istruzioni per risolvere un problema specifico o eseguire un calcolo.

Un computer classico (CPU) opera eseguendo istruzioni codificate (linguaggio macchina), agendo come un **interprete di un linguaggio formale**. Ogni livello software (dal linguaggio di alto livello fino al microcodice) è un'astrazione che viene *interpretata in istruzioni più fondamentali* (elementi di base).

Dopo il 1970, con l'avvento dei sistemi interattivi, delle reti e di Internet, i problemi da risolvere non erano più solo di calcolo, ma di organizzazione e interazione con il mondo reale, con gli utenti e con altre macchine. Vi sono state ‘forze’ che hanno sprinto verso nuove frontiere:

- **Complessità Crescente:** I programmi sono diventati troppo grandi per essere gestiti come un unico algoritmo, portando alla necessità di modularità, astrazione e occultamento dell'informazione, come negli oggetti software e nei (micro)servizi.
- **Requisiti di Interazione:** L'informatica si è spostata dal calcolo puro all'interazione continua (sistemi operativi, reti, database, interfacce utente). La “correttezza” non è solo la correttezza algoritmica, ma la correttezza comportamentale in un ambiente dinamico.
- **Ingegneria del Software:** Si è riconosciuta la necessità di discipline ingegneristiche per gestire l'intero ciclo di vita del software, superando la sola logica algoritmica.

Emergono nuove esigenze:

- **concorrenza** e processi che evolvono in parallelo;
- **comunicazione** tra entità autonome;
- **distribuzione** e assenza di una memoria centrale;
- **mobilità** di codici e canali;
- **stocasticità** e incertezza;
- **comportamenti globali emergenti** da interazioni locali.

La *Macchina di Turing* non basta più per descrivere questi fenomeni. Non è sbagliata: è solo **inadatta** al nuovo dominio.

Il concetto di computazione si è ampliata progressivamente, passando dall'idea di calcolo simbolico/algoritmico a nuove forme, quali:

- **Computazione come processo strutturato** (anni 60-80) in cui la computazione non è più solo trasformazione di simboli, ma organizzazione di processi concettuali, legati alla evoluzione del software (Programmazione strutturata, oop, etc.).
- **Computazione come concorrenza e comunicazione** (anni 70-oggi) in cui computazione diventa interazione, non più solo calcolo e calcolare significa “scambiare informazione”, non eseguire funzioni matematiche
- **Computazione come sistema complesso** (anni 90-oggi) in cui la computazione viene vista come entità che cooperano o competono, eventi asincroni, comportamento emergente, auto-organizzazione

I paradigmi computazionali oggi studiati dal [Natural computing](#), vengono astratti da fenomeni naturali diversi come l’auto-organizzazione, l’evoluzione Darwiniana, il comportamento di gruppo (Swarm intelligence), il sistema immunitario, fino alla idea (*Zuse-Fredlink*) che l’informazione sia più fondamentale della materia o dell’energia e che l’universo stesso sia un computer quantistico (*Seth Lloyd*) che computa il suo proprio comportamento.

Si assiste così a una esplosione di nuovi formalismi, ognuno dei quali nasce per rispondere a una “forza storica” precisa:

- **la concorrenza** → [CCS](#) (Milner) e [CSP](#) (Hoare), π -calcolo ([Pi calcolo](#));
- **il comportamento reattivo** → automi, transizione di stato, model checking;
- **la computazione distribuita** → Actor Model (Hewitt, Agha), Join Calculus;
- **la mobilità** → Mobile Ambients (Cardelli), π -calcolo mobile;
- **il caso e la variabilità** → π -calcolo stocastico (Priami), Markov Processes;
- **la computazione globale** → Field Calculus, Aggregate Computing (Beal, Viroli).

In ognuno di questi casi, il formalismo nasce *prima* dei linguaggi di programmazione che ne derivano. Ad esempio:

- **Lisp** come incarnazione del λ -calcolo;
- **Erlang, Akka, Orleans** come discendenti dell’Actor Model;
- **Go, Rust async, CSP libraries** ispirate a CSP;
- **Proto, Scafi** ispirati al Field Calculus;
- **AmbientTalk, Klaim** ispirati alla mobilità dei processi.

3.8.1 Verso la definizione di ‘sistemi che calcolano’

La nozione di computazione, nata per descrivere ciò che una macchina può calcolare, si è progressivamente trasformata in una teoria di ciò che un sistema può far emergere.

Il termine ‘*computazione*’ diventa **polisemico** e perde la nitidezza logico-matematica originaria.

Si può dire che oggi una *computazione* non è più intesa come la valutazione di una funzione, ma come l’evoluzione controllata di uno stato collettivo sotto vincoli.

Una estrema sintesi dell’evoluzione storica è che si passa:

1. **dal calcolo come trasformazione di funzioni,**
2. **al calcolo come parallelismo e interazione,**
3. **al calcolo come comportamento collettivo,**
4. **al calcolo come fenomeno emergente.**

Questo cambiamento è profondamente legato alla trasformazione dell’ingegneria del software: oggi non costruiamo più singoli algoritmi, ma **sistemi distribuiti, microservizi, reti di attori, sistemi software complessi**.

3.9 Famiglie di formalismi

Nel corso della storia dell'informatica teorica, i formalismi si sono evoluti in risposta a esigenze sempre nuove: definire cosa significhi calcolo, modellare la concorrenza, gestire sistemi distribuiti non affidabili, comprendere il ruolo della mobilità, descrivere l'emergenza di comportamenti globali.

Questa evoluzione ha dato vita a una serie di *famiglie* di formalismi, ciascuna caratterizzata da un'idea guida, da un ambito di applicazione privilegiato e da risultati teorici che ne precisano potere espressivo e limiti.

L'insieme delle famiglie mostra come i formalismi non siano strumenti puramente accademici: sono la **grammatica fondamentale** con cui l'ingegneria del software comprende e progetta sistemi sempre più complessi, autonomi, distribuiti e adattivi.

Ogni famiglia è nata in risposta a una tensione storica concreta, e ogni risultato teorico ha contribuito a definire il perimetro del possibile e dell'impossibile in informatica.

3.9.1 Alcuni dei promotori

Dalla *Macchina di Turing* ai calcoli concorrenti, fino ai modelli per sistemi complessi, la storia dei formalismi è la storia dell'evoluzione della computazione stessa: da operazioni meccaniche a fenomeni collettivi autonomi.

Ricordiamo alcuni degli studiosi che hanno aperto le diverse strade che oggi possiamo percorrere:

- **Alan Turing**, la computazione come processo meccanico sequenziale.
- **Alonzo Church**, la computazione come trasformazione simbolica (λ -calcolo).
- **Stephen Kleene, Gödel, Herbrand**, computazione come ricorsione.
- **Emil Post, Markov**, riscrittura e manipolazione di stringhe.
- **Robin Milner, Tony Hoare**, processi concorrenti e comunicazione.
- **Carl Hewitt, Gul Agha**, attori come entità autonome.
- **Luca Cardelli, Andrew Gordon**, mobilità e confini (*Ambient Calculus*).
- **Luca Cardelli, Gierz, Milner**, transizioni di stato e sistemi dinamici.
- **Radhika Nagpal, Beal, Viroli**, computazione di campo e self-organization.
- **Christopher Langton, Stuart Kauffman, Per Bak**, sistemi complessi e auto-organizzazione.

Queste figure non hanno solo inventato formalismi: hanno introdotto **nuovi modi di pensare** il calcolo; vediamone una sintetica panoramica.

3.9.2 Formalismi funzionali

La prima grande famiglia è quella dei formalismi funzionali, nata negli **anni '30** con Alonzo Church e la definizione del λ **calcolo**. L'idea fondamentale è che la computazione possa essere vista come *applicazione di funzioni a valori*, senza memoria mutabile né effetti collaterali.

Questa visione è profondamente radicata nella logica matematica e nella teoria delle funzioni ricorsive.

Questa famiglia di formalismi serve principalmente a:

I formalismi funzionali servono principalmente a:

- descrivere la computazione in modo puramente simbolico,
- ragionare su trasformazioni e riduzioni,

- garantire proprietà come la *referential transparency*,
- dimostrare equazioni e invarianti senza preoccuparsi dello stato.

Sono fondamentali per discipline come la programmazione funzionale, gli strumenti di ottimizzazione dei compilatori e i linguaggi basati su funzioni pure (Haskell, ML).

Principali risultati teorici

- **Equivalenza Church–Turing:** il λ -calcolo è Turing-completo.
- **Teorema di Church–Rosser (confluenza):** l'ordine di riduzione delle espressioni non influisce sul risultato finale (se questo esiste).
- **Indecidibilità della normalizzazione:** non si può decidere in generale se un termine del λ -calcolo ha una forma normale.
- **Sistema di tipi:** introduzione dei tipi semplici (Church) e dei sistemi polimorfi, fondamentali per linguaggi moderni.

Questi risultati fondano l'intera disciplina della programmazione funzionale moderna.

3.9.3 Formalismi concorrenti

A partire dagli **anni '60 e '70**, con lo sviluppo dei sistemi time-sharing, emerge il problema della **concorrenza**. Robin Milner, Tony Hoare ed altri introducono formalismi che trattano i processi *non* come funzioni, ma come entità dinamiche in comunicazione.

Nascono così **CSP**, **CCS** e, più tardi, il π **calcolo**.

Questa famiglia di formalismi serve principalmente a:

Questi formalismi servono a:

- modellare processi che evolvono in parallelo,
- formalizzare comunicazioni sincrone/asincrone,
- verificare sicurezza (deadlock-freedom) e liveness,
- definire equivalenze comportamentali (bisimulazione),
- ragionare sul nondeterminismo operativo.

Sono indispensabili per sistemi real-time, middleware concorrenti, modelli actor e linguaggi come Erlang, Occam, Go (canali CSP-like).

Principali risultati teorici

- **Bisimulazione** (Milner, Park): due processi sono equivalenti se non distinguibili da un osservatore esterno.
- **Teoria delle transizioni strutturali:** permette di definire formalmente il comportamento di processi complessi con regole modulari.
- π **calcolo Turing-completo** con scambio dinamico di canali.
- **Risultati di impossibilità:** certi pattern di mobilità non sono esprimibili in CCS senza estensioni.

Questi risultati segnano il passaggio decisivo dal calcolo come funzione al calcolo come *interazione*.

3.9.4 Formalismi distribuiti

Con lo sviluppo delle reti (ARPANET, Internet) emerge la necessità di modellare sistemi in cui nessun processo possiede un punto di vista globale. *Hewitt* introdusse l'**Actor Model** già negli **anni '70**

Questa famiglia di formalismi serve principalmente a:

- modellare comunicazione asincrona senza memoria condivisa;
- trattare fallimenti, partizioni di rete, ritardi imprevedibili;
- descrivere algoritmi distribuiti e protocolli;
- caratterizzare il comportamento di microservizi e cluster.

Principali risultati teorici

- **Teorema FLP (Fischer-Lynch-Paterson)**: impossibile garantire consenso deterministico in un sistema asincrono con un fallo possibile.
- **Teorema CAP**: impossibile garantire contemporaneamente **Consistenza, Disponibilità e Tolleranza alla Partizione**.
- **Determinismo con attori isolati**: gli attori sono Turing-completi ma con un forte controllo sull'interferenza.

Questi risultati hanno avuto impatto diretto sul design dei sistemi cloud e dei Sistemi a microservizi.

3.9.5 Formalismi per la mobilità

L'attenzione alla **mobilità** (di codice, processi, canali) emerge negli **anni '90** con gruppi come quello di *Luca Cardelli* e *Andrew Gordon*. Nascono così formalismi come il **Mobile Ambients** e le estensioni mobili del π -calcolo.

Questa famiglia di formalismi serve principalmente a:

- descrivere sistemi che cambiano struttura dinamicamente;
- modellare agenti mobili, protocolli ad hoc, migrazione di processi;
- rappresentare reti pervasive o IoT.

Principali risultati teorici

- **Il mobile π calculus** è più espressivo di **CCS**;
- **alcune proprietà di sicurezza sono indecidibili** in ambienti mobili;
- **gerarchie di potere espressivo** tra modelli nominali e strutturali;
- **equivalenza con automi di mobilità** in vari casi.

Questi formalismi sono fondamentali nella modellazione di sistemi IoT e mobile-cloud.

3.9.6 Formalismi stocastici

La necessità di modellare comportamenti probabilistici, biologici o di rete porta alla nascita (**anni 2000**) del **π calcolo stocastico**, delle **CTMC** (*continuous-time Markov chain*) e **DTMC** (*Discrete-time Markov chain*), dei modelli Markoviani e dei sistemi probabilistici.

Questa famiglia di formalismi serve principalmente a:

- descrivere sistemi soggetti a rumore, variabilità o fallimenti probabilistici;
- associarne tempi medi, probabilità di stati, rischi;

- supportare model checking probabilistico.

Principali risultati teorici

- **ergodicità** e **stazionarietà** delle catene di Markov;
- **decidibilità** del model checking probabilistico su DTMC/CTMC finite;
- **indecidibilità** per modelli con memoria illimitata;
- **calcolo Turing-completo** in alcune estensioni probabilistiche del λ -calcolo.

Questi formalismi sono oggi fondamentali per sistemi autonomi, robotics e protocolli di rete.

3.9.7 Formalismi field-based e global behavior

Dagli **anni 2000-2010** nasce una nuova esigenza: modellare **comportamenti globali emergenti** a partire da moltissime entità locali. È la famiglia dei formalismi “a campo”, come il **Field Calculus**, il linguaggio **Proto**, e le estensioni **aggregate computing**.

Il Field Calculus non calcola valori, ma calcola configurazioni spaziali, dando come risultato un pattern stabile.

Questa famiglia di formalismi serve principalmente a:

- descrivere sistemi auto-organizzanti;
- modellare propagazione, stabilizzazione e gradienti;
- controllare comportamenti globali senza un coordinatore centrale;
- formalizzare sistemi complessi su larga scala: swarm robotics, reti self-organizing, large IoT.

Principali risultati teorici

- **equivalenza del field calculus** con un λ **calcolo distribuito** strutturato su spazio e tempo;
- **teoremi di stabilizzazione** in presenza di mobilità e fallimenti;
- **limiti della computazione spaziale**: impossibile ottenere global properties con vicinato insufficiente;
- **convergenza garantita** per classi di programmi auto-stabilizzanti.

Questa famiglia rappresenta il ponte più diretto tra sistemi complessi naturali e sistemi complessi software.

3.9.8 Formalismi per sistemi complessi e dinamiche non lineari

Modelli come **automi cellulari**, **agent-based models**, **reti complesse** e **sistemi dinamici non lineari** derivano da un'altra tradizione: fisica, biologia, teoria del caos. Il tema centrale è la *dinamica collettiva*.

Questa famiglia di formalismi serve principalmente a:

- analizzare emergenze non lineari;
- studiare caos, biforcazioni, auto-organizzazione;
- modellare popolazioni di agenti, ecosistemi artificiali, fenomeni di sincronizzazione.

Principali risultati teorici

- **Turing-completezza** dell'automa cellulare [Rule 110](#);
- **Teorema di Li-Yorke**: il caos è inevitabile in certi sistemi dinamici;
- **criticità auto-organizzata** (Bak, Tang, Wiesenfeld) produce distribuzioni a [Legge di potenza](#);
- **limiti alla predicibilità** (sensibilità alle condizioni iniziali).

Questi formalismi ispirano tecniche di simulazione su larga scala e modelli per “sistemi complessi software”.

3.9.9 Oltre la macchina di Von Neumann

L’architettura di Von Neumann — processore centrale (CPU) + memoria separata + esecuzione sequenziale di istruzioni — rimane il modello di riferimento per la maggior parte dei computer moderni. Tuttavia, pone limiti ben noti:

- buona flessibilità generale
- colli di bottiglia nell’accesso alla memoria
- limitata parallelizzazione
- inefficienza energetica con carichi massivamente paralleli

Da qui deriva la nascita di **nuove architetture**, orientate a superarne i limiti su compiti specifici. Non è questa la sede per entrare nei dettagli di queste nuove architetture. Riportiamo solo un quadro sintetico delle alternative oggi più diffuse:

Architettura	Paradigma	Applicazioni	Limitazioni
GPU	SIMD/SIMT massivo	Grafica, ML, fisica	Poco efficiente per logica complessa
TPU	Matrici / systolic arrays	Inferenza e training AI	Poco efficiente per logica complessa
Quantum	Superposizione e entanglement	Problemi esponenziali selezionati	Rumore, scalabilità, algoritmi specifici

3.9.10 Quantum computer

La computazione legata sulle leggi della **Meccanica quantistica** (avanzata da *Richard Feynman* nel 1982 e poi formalizzata da *David Deutsch* nel 1985) ha promosso lo sviluppo del **Computer Quantistico (QC)** e del **Quantum computing**.

Una convergenza di idee emerse tra la fine degli anni ‘70 e l’inizio degli anni ‘80, è culminata con l’indicazione di *Richard Feynman* (1982) che un computer basato sulle leggi della meccanica quantistica potesse essere lo strumento necessario per simulare altri sistemi quantistici.

Su questa base, nel 1985, *David Deutsch* formalizzò l’idea di una Macchina di Turing Universale Quantistica, dimostrando che i principi quantistici potevano essere applicati all’elaborazione dell’informazione, aprendo la via all’idea di **Computer Quantistico (QC)**.

Nel 1994, *Peter Shor* sviluppò di un algoritmo quantistico in grado di fattorizzare numeri interi in modo esponenzialmente più veloce dei migliori algoritmi classici.

Da quello momento, la ricerca nel campo del calcolo quantistico è progredita rapidamente, portando allo sviluppo di prototipi di computer quantistici e alla sperimentazione di algoritmi quantistici per problemi specifici.

Mentre la programmazione classica lavora con istruzioni che cambiano lo stato da 0 a 1 o viceversa, la programmazione quantistica lavora con i **qubit**:

- Un qubit esiste in una sovrapposizione di stati, una combinazione probabilistica di 0 e 1.
- Programmare un QC significa controllare e manipolare queste probabilità per allineare gli stati quantistici in modo che, al momento della misurazione, lo stato desiderato (la soluzione) abbia la probabilità massima.

Ciò che rende il computer quantistico qualitativamente diverso da un computer classico è l’**entanglement**.

L'entanglement è una correlazione quantistica che si verifica quando lo stato di due o più qubit è così profondamente legato che non è possibile descrivere lo stato di ciascun qubit indipendentemente, anche se sono separati da grandi distanze.

Quando un algoritmo quantistico crea entanglement tra N qubit, non sta semplicemente lavorando su N bit indipendenti. Sta lavorando su un unico stato quantistico collettivo che può codificare 2^N combinazioni classiche contemporaneamente.

Questo significa che, grazie all'entanglement, le porte quantistiche (**Quantum Gates**) applicate durante l'elaborazione modificano simultaneamente tutte e 2^N le possibili soluzioni. Questo è il fondamento del parallelismo quantistico che accelera esponenzialmente la risoluzione di alcuni problemi.

La programmazione quantistica viene eseguita utilizzando linguaggi o framework come Qiskit (IBM) o Cirq (Google), che traducono istruzioni ad alto livello in circuiti di *Quantum Gates*.

Ovviamente, la computazione quantistica apre nuove prospettive e nuovi campi di ricerca teorica e applicate. Tuttavia, allo stato attuale, il computer quantistico non sostituisce quello classico, ma fornisce una capacità di calcolo qualitativamente diversa per specifiche classi di problemi.

Il calcolo di un QC è probabilistico, unitario e non clonabile (No-Cloning Theorem).

Note: La crescente miniaturizzazione dei transistor che compongono un computer 'classico' implica l'insorgere di alcuni fenomeni spiegabili solo con la [Meccanica quantistica](#). Ricordiamo al proposito il **Tunneling Quantistico**:

- responsabile di correnti di dispersione (**leakage current**), che generano calore e sprecano energia.
 - usato per spostare gli elettroni dentro e fuori le "floating gate" che intrappolando elettroni in modo da permettere la memorizzazione dei dati nelle **memorie flash** (usate negli SSD e nei telefoni).
-

3.10 Modelli formali: necessari ma non sufficienti

I modelli formali sono importanti perché isolano l'essenza concettuale dei sistemi permettendo di capirli, compararli e verificarli in modo rigoroso e universale.

I modelli formali sul calcolo non sono sufficienti perché non dicono **come** realizzare efficientemente un sistema: sono i **linguaggi di programmazione** che colmano il divario, traducendo principi astratti in meccanismi concreti, efficienti e utilizzabili nella pratica.

Consideriamo, ad esempio, il **processo**, spesso introdotto come concetto fondamentale legato alla computazione.

In **CSP**, *π*_{calcolo} **calcolo**, ecc.:

- ciò che esiste sono **processi matematici**, non thread o coroutine;
- un processo è un'entità astratta, definita da un insieme di transizioni o eventi;
- lo stato del processo cambia tramite regole formali ("esegue un evento", "comunica", "si sincronizza").

In questi modelli:

- **non esistono stack**,
- **non esistono scheduler**,
- **non esiste il concetto di sospensione volontaria** come *yield* o *await*,
- **non c'è distinzione tra thread pesante o coroutine leggera**.

CSP dice:

- Esistono processi che comunicano per canali, in modo sincrono.

Il π -calcolo dice:

- I processi possono scambiarsi anche canali, supportando mobilità.

Ma nessuno dei due dice:

- come si deve gestire lo stack,
- come evitare di creare migliaia di thread pesanti,
- come sospendere una funzione senza bloccare un thread OS.

Per implementare *processi concettuali* a basso costo, servivano meccanismi pragmatici. Tradizionalmente, il concetto di processo è realizzato usando *Thread* del sistema operativo o *Thread più leggeri* gestiti da *scheduler*. Tuttavia questi, di solito: **i)** richiedono molta memoria (stack da ~1 MB) **ii)** sono gestiti da kernel **iii)** hanno costi elevati di switching.

La possibilità pratica di attivare processi si riduce quindi a poche migliaia di *Thread* in uno stesso sistema, mentre molte applicazioni moderne richiedono **decine o centinaia di migliaia di attività concorrenti**.

Meccanismi implementativi quali le **callback** o le **promesse/future** si sono rivelati non ideali (problema del “callback hell” e della frammentazione del codice).

Un concetto/meccanismo che risale agli albori dell’informatica — introdotto negli **anni ‘60** come meccanismo di cooperazione tra routine — poi quasi dimenticato con l’affermarsi dei *thread di sistema* è quello di **coroutine**.

Questo meccanismo viene oggi riscoperto e potenziato perché fornisce una soluzione più leggera, espressiva ed efficiente alla crescente esigenza di concorrenza su larga scala.

3.11 La coroutine un nuovo vecchio concetto

La “rinascita” delle coroutine nasce dagli stessi problemi che avevano motivato la loro invenzione negli anni ‘60:

- la necessità di gestire altissima concorrenza (i thread OS sono troppo pesanti per scalare a centinaia di migliaia o milioni di attività)
- la necessità di scrivere codice asincrono leggibile (superare callback hell, future annidati, codice frammentato)
- la necessità di modelli di concorrenza più sicuri (riduzione di deadlock e race, privilegiando message passing)

Introdotte in linguaggi pionieristici come **Simula 67** (uno dei padri della **OOP** e dei primi concetti di concorrenti cooperativi), **Modula-2** e **Modula-3** (di **Niklaus Wirth**, padre anche del linguaggio **Pascal**) e alcune versioni evolute del *Lisp*, il concetto è stato oggi riscoperto e potenziato perché fornisce una soluzione più leggera, espressiva ed efficiente alla crescente esigenza di concorrenza su larga scala.

Esso ritorna in linguaggi come:

- Go (goroutine, 2009)
- Kotlin (coroutine suspend, 2017)
- Python async/await (2015)
- Java virtual threads (Loom), che di fatto sono coroutine del linguaggio (2023)
- C# async/await (2012)
- JavaScript async/await (2017)

3.12 Un esempio: CSP e GO

CSP (*Communicating Sequential Processes*) è un modello formale introdotto da **Tony Hoare (1978)** per descrivere sistemi concorrenti. È usato per *specificare, analizzare e verificare- protocolli di comunicazione tra processi.

Riportiamo un quadro sintetico delle idee fondamentali di CSP

- Comunicazione sincrona per messaggi
- Canali come entità di prima classe
- Composizione parallela dei processi
- Nessuna memoria condivisa
- Sincronizzazione implicita nella comunicazione
- Modello matematico verificabile

3.12.1 CSP: Processi sequenziali

Le caratteristiche chiave di CSP (in sintesi) sono:

Caratteristica	Descrizione
Processi sequenziali	Il sistema è formato da processi , ciascuno dei quali è sequenziale: esegue azioni una dopo l'altra.
Comunicazione per canali	<ul style="list-style-type: none"> • La comunicazione è sincrona: chi invia e chi riceve devono essere entrambi pronti → la comunicazione “si completa insieme”. • Il canale è un'astrazione matematica: niente buffer (di default). Esempio : <pre>P = c ! x // processo P invia x sul canale c Q = c ? y // processo Q riceve un valore su c</pre> La comunicazione avviene solo quando entrambi i processi sono pronti.
Composizione concorrente	La concorrenza si ottiene tramite composizione parallela : $P \parallel Q$ P e Q eseguono in parallelo e devono <i>sincronizzarsi</i> sugli eventi comuni.
Scelta	Ci sono due tipi di scelta: <ul style="list-style-type: none"> • Scelta esterna (determinata dall'ambiente) • Scelta interna (non osservabile)
Verifica formale	CSP fornisce un <i>algebra formale</i> per ragionare su: <ul style="list-style-type: none"> • deadlock • livelock • race conditions • equivalenza comportamentale È molto usato in contesti safety-critical.

3.12.2 Come CSP ha influenzato Go

Il linguaggio **Go**, creato da Google (Pike, Thompson, Griesemer), incorpora esplicitamente concetti provenienti da CSP.

La frase di Rob Pike è celebre:

Do not communicate by sharing memory; instead, share memory by communicating.

Go reinterpreta CSP in modo **pragmatico**, rendendolo adatto alla programmazione industriale. Le principali differenze sono riassunte nella seguente tabella:

Elemento	CSP	Go
Comunicazione	Sincrona	Sincrona + buffer (asincrona)
Semantica	Matematica, formale	Pratica, implementativa
Processi	Astratti, infiniti	Goroutine, mappate su thread
Verifica formale	Centrale	Non è obiettivo del linguaggio
Scopo	Specifica e modellazione	Programmazione applicativa

Go si ispira a CSP, ma **non è una sua implementazione formale**.

Go-routine come “processi CSP”

Le **goroutine** sono entità concorrenti leggere, analoghe ai *processi di CSP*: indipendenti, sequenziali, comunicano tramite canali.

I canali di Go sono ispirati direttamente ai canali CSP

Nei canali Go:

```
ch := make(chan int)
```

la comunicazione può essere **bloccante** (sincrona), come in CSP.

```
ch <- x // invio
y := <-ch // ricezione
```

Il completamento avviene solo quando entrambi sono pronti (*sincronizzazione implicita*, come in CSP).

Ma Go introduce una differenza importante: i canali **possono essere bufferizzati**:

```
make(chan int, 10)
```

Dunque in Go, la comunicazione può essere anche *asincrona*, quindi più flessibile.

Select

La Select di Go deriva dalla scelta esterna di CSP

```
select {
  case x := <-ch1:
    ...
  case ch2 <- y:
    ...
  default:
```

(continues on next page)

(continued from previous page)

```
}    ...
```

Questo è la versione di Go dell'operatore di *scelta esterna di CSP*.

Composizione concorrente

La frase CSP: ``P || Q``

diventa in Go:

```
go f() // esegue f in parallelo
```

La concorrenza nasce dalla creazione di *goroutine* indipendenti.

Assenza di memoria condivisa “esplicita”

Go permette la memoria condivisa, ma lo *stile idiomático* promuove il modello CSP:

- evitare il locking manuale
- preferire canali ai mutex

3.13 Modelli e linguaggi

Un modello formale fonda la potenza espressiva, mentre il linguaggio di programmazione governa l'uso pratico di tale potenza.

La storia rende evidente che a un singolo modello corrispondono di fatto molti linguaggi, ciascuno diverso per architettura e filosofia.

Note: Si pensi ad esempio al λ -calcolo e ai linguaggi Lisp, Scheme, ML, Haskell, F\#, Scala, Clojure, etc.

La proliferazione segue spesso l'emergere di nuovi problemi e ogni linguaggio ‘congela’ una visione del mondo e una *teoria implicita* del sistema che si vuole costruire.

I modelli formali delimitano ciò che è possibile; ma i linguaggi di programmazione proliferano perché ogni epoca, dominio e comunità ha bisogno di un modo diverso per rendere quel possibile effettivamente costruibile.

E' quindi il momento di esplorare più da vicino il mondo dei linguaggi.

I LINGUAGGI

E' importante avere chiara la distinzione tra *linguaggio naturale* e *linguaggio di programmazione* e il ruolo che questo può avere nella definizione e costruzione di oggetti e 'sistemi artificiali'.

4.1 Linguaggi naturali

Il linguaggio naturale (LN) come l'italiano, l'inglese, ecc. è lo strumento primario degli esseri umani per interagire con il mondo e per *categorizzarlo*. Il suo ruolo non è solo descrittivo, ma anche costitutivo e sociale, in quanto un LN non è un semplice specchio che riflette oggetti esistenti.

Un LN è piuttosto un filtro categoriale che organizza e rende gestibile un flusso continuo di percezioni. Le impostazioni filosofiche estreme suggeriscono che l'esistenza degli "oggetti" come li conosciamo (con i loro confini e le loro identità) è un prodotto di questo filtro linguistico. In questa visione, gli oggetti non esistono "in sé" ma solo come costrutti mentali e si pone il problema di cosa sia "reale".

Nonostante la sua potenza, il linguaggio naturale presenta alcune caratteristiche limitative:

1. **Vaghezza e Ambiguità:** Molte parole hanno confini sfocati (*fuzzy boundaries*). Non esiste un confine netto che determini, ad esempio, quando un "cespuglio" diventi un "albero" o quando un "sasso" sia abbastanza grande da essere chiamato "roccia". Questo è il limite della precisione.
2. **Dipendenza dal Contesto:** La descrizione di un oggetto è sempre incompleta e dipendente dal contesto. Descrivere un "tavolo" in fisica richiede la massa e la composizione molecolare; descriverlo in un negozio di mobili richiede lo stile e il prezzo. Nessuna descrizione è totale.
3. **Problema del Riferimento:** Il linguaggio non può esaurire tutte le proprietà di un oggetto. Possiamo solo riferirci a un sottoinsieme di proprietà (**l'intensione**) per identificare la collezione di oggetti (**l'estensione**).

Gli oggetti reali (**things-in-themselves**) possono esistere fisicamente (cioè, esistono atomi e onde), ma la loro divisione in categorie discrete e utilizzabili (es. "pianta", "bottiglia", "confine", "matrimonio") è un atto imposto dal linguaggio e dalla cultura.

Per molti, non c'è un modo "oggettivo" di dividere il mondo in oggetti. L'atto di riferimento linguistico a un oggetto è sempre vincolato al sistema linguistico e concettuale che stiamo usando (la cosiddetta *underdetermination of reference* del filosofo Willard Van Orman Quine).

In ogni caso, diverse evidenze empiriche indicano che un LN influenza o orienta l'attenzione e la cognizione, rendendo più facile o frequente pensare in certi modi. Ad esempio, il linguaggio fornisce le categorie che gli umani usano per interpretare e comunicare la realtà dello spazio e del tempo e queste sono diverse in contesti culturali diversi (*Ipotesi Sapir-Whorf*).

Tuttavia, l'ambiguità delle frasi dei linguaggi naturali (ad esempio: "*Ho visto un uomo su una pianta col cannocchiale*") sono caratteristiche intrinseche e necessarie per la flessibilità e l'espressione umana e può anche essere vista come un vantaggio:

1. **Efficienza e Velocità di Trasmissione.** utilizzando la conoscenza condivisa e il contesto.
2. **Creatività, Espressione Artistica e Ironia.** Nuovi significati e usi delle parole emergono proprio grazie alla loro flessibilità e ambiguità, consentendo al linguaggio di evolvere per adattarsi a nuove realtà sociali e tecnologiche.
3. **Flessibilità e Adattabilità Sociale.**
 - Mantenere relazioni positive, di negoziare o di evitare il conflitto.
 - Identificazione del Contesto (Pragmatica): L'ambiguità ci costringe a interagire con l'ambiente e con l'altro per risolverla.
 - Il processo di disambiguazione contestuale è una funzione cruciale della nostra intelligenza e della nostra capacità di adattarci a situazioni diverse.

4.2 I Large Language Models

Gli ormai onnipresenti **Large Language Models (LLM)** si fondano su concetti di statistica e probabilità applicate al linguaggio naturale.

Un **LLM** è una rete neurale trasformatrice (**Transformer**), addestrata su miliardi di parole e testi provenienti da Internet, libri e altre fonti.

Il modello non lavora su singole lettere o parole intere, ma su **token** (una parola intera, una parte di una parola, un segno di punteggiatura o uno spazio).

Quando il modello elabora un *token* nel **prompt** dato in input dall'utente, non lo guarda isolatamente, ma valuta quanto sono importanti tutti gli altri token nel prompt per “dare significato” al token corrente. Questo modo di procedere è denominato **attenzione** e costruisce una rappresentazione numerica finale del prompt che include il contesto e l'importanza di ogni parte del testo di input.

L'ultima rappresentazione numerica del prompt funge da stato iniziale. A partire da questo stato, il modello fa una **previsione del token successivo** utilizzando una tecnica chiamata campionamento stocastico o **sampling** (spesso controllato da un parametro denominato **Temperatura**).

Il processo si ripete, ma questa volta, il nuovo input per la previsione successiva è l'intero prompt più il token appena generato.

Con questo modo di procedere, un **LLM** non “capisce” nel senso umano. Il suo successo deriva dalla sua capacità di:

- **Imitare la Sintassi e la Semantica del Testo Umano:** Avendo analizzato enormi quantità di testo, ha imparato quali sequenze di parole sono statisticamente coerenti e significative.
- **Mantenere il Contesto:** Grazie al meccanismo di attenzione, ogni token generato è coerente non solo con l'ultima parola, ma con l'intero contesto del prompt e della conversazione precedente.

4.3 Linguaggi di programmazione

I linguaggi di programmazione (**LdP**) sono costruiti per essere l'esatto opposto dei linguaggi naturali in termini di precisione e ambiguità.

Questi linguaggi ‘artificiali’ non si preoccupano di come l'essere umano percepisce il mondo esterno. La loro **ontologia** (gli oggetti che esistono) è definita esplicitamente dal programmatore:

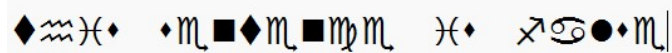
- Gli oggetti sono tipi di dati (**Int**, **String**, **Person**).
- Lo spazio e il tempo sono gestiti da astrazioni matematiche precise (indirizzi di memoria, timestamp, cicli di CPU).

- Non c'è spazio per la “percezione” individuale dei concetti.

Se si usa un LdP, l'hardware sottostante (il *computer*) esegue la logica allo stesso modo; ciò che cambia è l'efficienza e la facilità con cui l'umano esprime quella logica nel linguaggio usato.

4.3.1 Una frase misteriosa

Supponiamo che un esploratore abbia trovato una incisione rupestre che si presenta come segue:



Cosa dovrà fare l'esploratore per comprenderne il significato? Certo dovrà occuparsi di sintassi e di semantica.

4.3.2 Sintassi e semantica

Ogni frase di un LdP ubbidisce a una precisa sintassi, espressa da regole definite con notazioni formali.

La frase di un LdP ha sempre una interpretazione univoca che permette al computer di **agire** in un unico modo: questo modo di agire costituisce la **semantica** della frase.

- **Sintassi:** Riguarda le regole strutturali che definiscono se una frase è legalmente formata (es. le parentesi sono bilanciate? I punti e virgola sono al posto giusto?). Se una frase non rispetta la sintassi, è un errore e il computer non può agire.
- **Semantica:** Riguarda il significato della frase sintatticamente corretta e le azioni che ne derivano. La semantica è, in sostanza, la descrizione rigorosa e formale dell'effetto che l'esecuzione di una frase ha sullo stato della macchina.

4.3.3 Linguaggi per descrivere linguaggi

C'è una connessione fondamentale tra linguaggi, macchine e la necessità di notazioni formali per definire e studiare i linguaggi in modo non ambiguo.

Un **computer** è un sistema fisico. Ogni istruzione di un LdP si traduce in una serie di transizioni di stato all'interno di circuiti e memoria. Il **linguaggio macchina** di un computer è direttamente **interpretabile** dalla CPU. Tutto il codice di alto livello deve essere ricondotto a questo linguaggio di basso livello per l'esecuzione.

Prima che il computer possa eseguire le frasi di un LdP, deve **riconoscere** tali frasi (fare il **parsing**) cioè determinare se la frase costituisce una sequenza di simboli valida di quel linguaggio.

Per descrivere le forme lecite assumibili dalle frasi di un LdP (**sintassi**) si usano notazioni (dunque ulteriori linguaggi) dette **Grammatiche Formali** costituite da un sistema di regole che delineano matematicamente un insieme (di solito infinito) di sequenze finite di simboli (*stringhe*) appartenenti ad un ***alfabeto**- anch'esso finito.

Macchine astratte per riconoscere linguaggi

La **Gerarchia di Chomsky** è un insieme di classi di grammatiche formali che generano linguaggi formali. Un concetto fondamentale nell'informatica teorica è la corrispondenza tra la **Gerarchia di Chomsky** e una **Gerarchia di Macchine Astratte** in grado di riconoscere i linguaggi.

Note: In informatica, una **Macchina astratta** è un modello teorico di sistema computazionale.

La corrispondenza tra tipi di grammatiche e macchine astratte in grado di riconoscere i linguaggi generati può essere riassunta come segue:

Tipo di Chomsky	Nome Grammatica	Macchina Astratta	Complessità di Riconoscimento
Tipo0	Unrestricted	Macchina di Turing	Indecidibile
Tipo1	Context-Sensitive	Macchina di Turing Limitata Linearmente (LBA)	Decidibile
Tipo2	Context-Free	Automa a Pila (Pushdown Automaton, PDA)	Decidibile, Tempo Polinomiale
Tipo3	Regular	Automa a Stati Finiti (Finite Automaton, FA)	Decidibile, Tempo Lineare

Il riconoscimento delle frasi di un LdP deve avvenire rapidamente. Quindi la sintassi di tutti gli LdP oggi uso è descritta grammatiche **Context-Free** o **Regular**.

4.3.4 La analisi sintattica

Le **grammatiche formali** definiscono in modo non ambiguo e matematicamente rigoroso quali sequenze di simboli (parole chiave, identificatori, operatori, ecc.) costituiscono un programma sintatticamente corretto.

Note: La notazione più fondamentale è la *Forma di Backus-Naur (BNF)* e la *BNF Estesa (EBNF)*. Questa notazione definisce la sintassi attraverso un insieme di regole di produzione. Ad esempio

```
<assegnazione> ::= <variabile> = <espressione> ;
<espressione> ::= <termine> | <espressione> + <termine>
```

Le grammatiche sono essenziali per la costruzione dei traduttori (**compilatori** e **interpreti**):

- *Analisi Lessicale (Lexing)*: Il testo viene scomposto in **token** (simboli terminali).
- *Analisi Sintattica (Parsing)*: Il *parser* utilizza le regole della grammatica per verificare se la sequenza di *token* è strutturalmente corretta. Se le regole vengono soddisfatte, il parser costruisce un **Albero Sintattico Astratto (AST)**, che è la rappresentazione strutturata del codice che verrà poi usata per definire la semantica (il significato e l'azione) del programma.

Sono stati creati tool che generano automaticamente un *parser* partendo dalla specifica di un linguaggio di programmazione descritta con la **BNF**, ad esempio **Yacc** (*Yet Another Compiler Compiler*).

4.3.5 Descrivere la semantica

Si distinguono diversi tipi di semantica formale, ciascuno con i suoi metodi e applicazioni specifiche.

Note: La semantica operativa (**Operational Semantics**) definisce il significato di un programma descrivendo come viene eseguito. L'attenzione è posta sui passaggi elementari di calcolo che un'istruzione innesca.

- *Formalismo Principale*: Regole di transizione.
- *Utilità principale*: Progettazione di interpreti e macchine virtuali

La semantica denotazionale (**Denotational Semantics**) definisce il significato di un programma associando (o "denotando") a ogni costrutto una funzione matematica pura.

- *Formalismo Principale*: Teoria dei Domini e Punti Fissi (Fixed Points).

- *Utilità principale:* Dimostrazione di equivalenza tra programmi

La semantica assiomatica (**Axiomatic semantics**) definisce il significato di un programma specificando le proprietà logiche che devono essere vere prima e dopo l'esecuzione dell'istruzione.

- *Formalismo Principale:* La Logica di Tony Hoare.
- *Utilità principale:* Verifica e validazione di programmi

Il tipo di semantica più utilizzato nella pratica ingegneristica e didattica della programmazione è la **Semantica Operazionale**, per i seguenti motivi:

1. **Corrispondenza Diretta con l'Implementazione:** Le regole della semantica operativa (es. "Quando incontri un'istruzione IF, valuta prima la condizione, se è vera esegui il blocco 'then', altrimenti il blocco 'else'") sono quasi un blueprint diretto per l'implementazione di un interprete o di un compilatore. È la forma più vicina a come una macchina esegue i calcoli.
2. **Facilità di Comprensione e Debug:** I programmatori sono abituati a tracciare l'esecuzione del codice passo dopo passo, proprio come farebbe un debugger. La semantica operativa formalizza questa tracciabilità, rendendo intuitivo capire cosa accadrà ad ogni istruzione.
3. **Definizione di Standard:** Molte specifiche di linguaggi di programmazione, come C, C++, e Java (in termini di esecuzione della Java Virtual Machine o JVM), si basano implicitamente o esplicitamente su un modello operativo per definire il comportamento del linguaggio in vari scenari.

4.4 Compilatori e interpreti

Un **compilatore** è un programma che traduce l'intero codice sorgente di un LdP ad alto livello in un programma equivalente in linguaggio macchina (o in un formato eseguibile intermedio), prima della sua esecuzione.

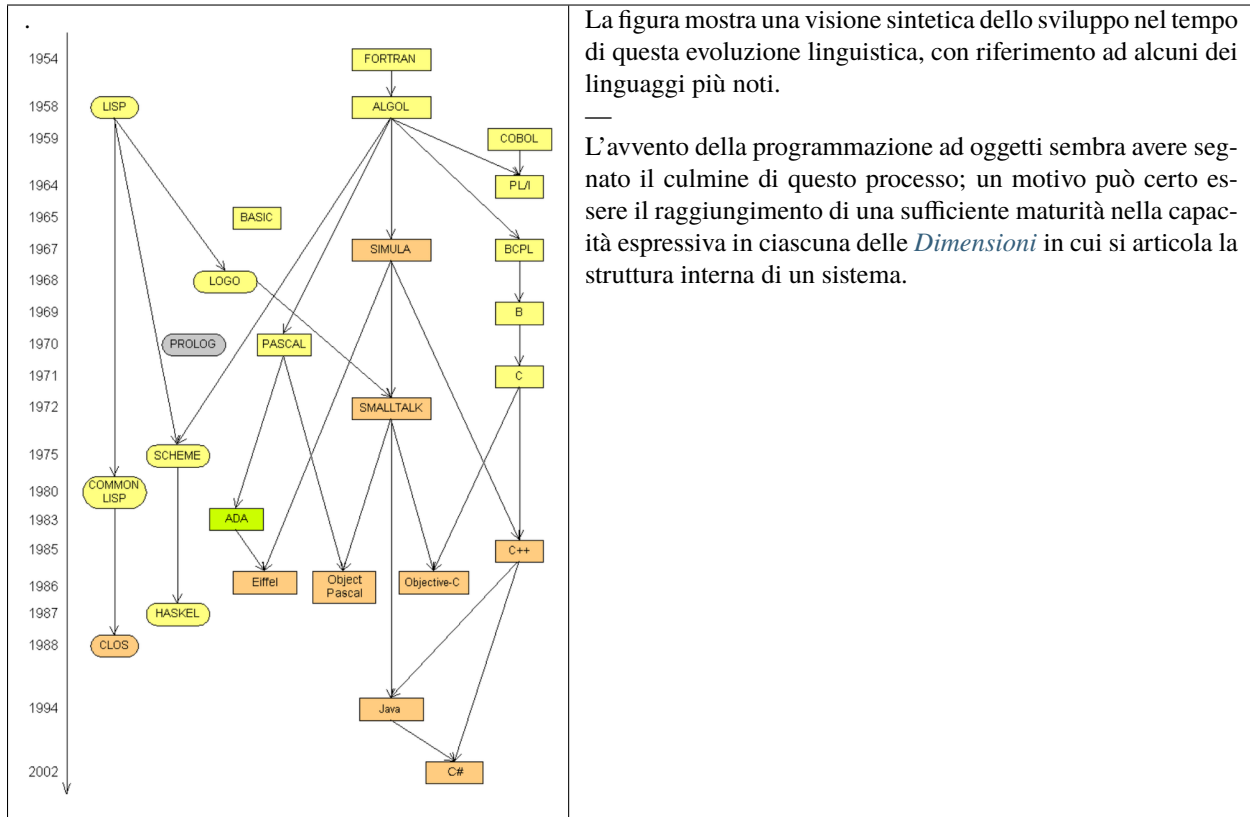
Un **interprete** è un programma che esegue direttamente il codice sorgente istruzione per istruzione, traducendo e valutando le frasi una dopo l'altra.

Molti LdP usano approcci ibridi, come:

- Java → compilazione in bytecode + interpretazione/ottimizzazione **JIT** (*Just-In-Time Compilation*)
- Python → bytecode + interprete (CPython)
- JavaScript → interprete + compilazione dinamica **JIT** nei browser

4.5 General Processing Programming Languages

Il numero totale di linguaggi di programmazione (inclusi linguaggi esoterici, accademici e non più in uso) è stimato in circa 2500.



Tuttavia, il numero di Linguaggi di Programmazione Generale (GPL) attivamente utilizzati in ambiente industriale è molto più limitato, probabilmente tra 20 e 50. Solo una decina di questi domina oltre l'80% del mercato dello sviluppo; tra essi vi sono:

Python, Java, Javascript, C, C++, C#, TypeScript, GO

Questi GPL sono quasi sempre utilizzati in combinazione con **framework** e **librerie** che non solo forniscono funzionalità/utilità ma che possono anche alzare il livello di astrazione. Discuteremo questo aspetto più avanti, nella sezione *L'astrazione*.

4.5.1 Stili (paradigmi) di programmazione

I linguaggi di programmazione possono essere raggruppati in stili o paradigmi, che definiscono il **modo concettuale** in cui si struttura un programma per risolvere un problema.

Paradigma Imperativo

Il paradigma imperativo si concentra sul come un programma debba operare. Si basa sul concetto di **stato** e su istruzioni (o comandi) che modificano sequenzialmente tale stato (variabili, memoria). Il codice descrive una serie di **azioni esplicite** che il computer deve eseguire.

Come Linguaggio Rappresentativo possiamo citare il C.

Caratteristiche:

- **Stato Mutabile:** Le variabili possono essere modificate.
- **Flusso di Controllo:** Utilizzo massiccio di costrutti come *if/else*, *for*, *while*,

- **Procedure/Funzioni:** Raggruppano istruzioni per il riutilizzo.

Paradigma Orientato agli Oggetti (OOP)

Derivato dal paradigma imperativo, l'Object Oriented Programming (**OOP**) organizza il software attorno a **oggetti**, che combinano **dati** (attributi) e le **funzioni** (metodi) che operano su quei dati. L'obiettivo è modellare entità del mondo reale.

Come Linguaggio Rappresentativo possiamo citare il **C++** o **Java**.

Caratteristiche:

- **Inscapsulamento** (Encapsulation): Raggruppamento di dati e metodi che operano su di essi.
- **Ereditarietà** (Inheritance): Le nuove classi ereditano le proprietà delle classi esistenti.
- **Polimorfismo** (Polymorphism): Oggetti diversi possono rispondere allo stesso input in modi diversi.
- **Astrazione** (Abstraction): Nascondere la complessità implementativa all'utente.

Paradigma Funzionale (FP)

Il paradigma funzionale tratta il calcolo come la valutazione di **funzioni matematiche** (non nel senso di procedure imperative). Si concentra sul **cosa** calcolare, evitando cambiamenti di stato e dati mutabili. Promuove l'uso di **funzioni pure**.

Come Linguaggio Rappresentativo possiamo citare il **Lisp/Scheme** o **Haskell**.

Caratteristiche:

- **Immutabilità:** I dati non cambiano una volta creati.
- **Funzioni Pure:** Una funzione restituisce sempre lo stesso output per lo stesso input e non produce effetti collaterali (non modifica lo stato esterno).
- **Funzioni di Ordine Superiore:** Funzioni che accettano altre funzioni come argomenti o le restituiscono come risultato.

Paradigma Logico (Dichiarativo)

Questo paradigma si basa sulla **logica formale**. Il programmatore **non** specifica *come* trovare la soluzione, ma descrive il problema in termini di **fatti** e **regole**. Il motore di esecuzione (l'interprete *risolutore*) cerca le risposte che soddisfano queste regole.

Come Linguaggio Rappresentativo possiamo citare il **Prolog**

Caratteristiche:

- **Asserzioni (Facts):** Dichiarazioni vere sul dominio del problema.
- **Regole (Rules):** Definizioni di nuove verità basate su fatti esistenti.
- **Query:** Domande poste al sistema per dedurre nuove informazioni.
- **Backtracking:** Il meccanismo di ricerca automatico per trovare le soluzioni.

Note: Molti linguaggi moderni (come *Python*, *JavaScript* e *Scala*) sono **multi-paradigma**, supportando elementi di imperativo, OOP e funzionale.

4.6 Gli attori come nuovo paradigma?

L'**Actor Model**, ideato da *Carl Hewitt* nel 1973 e reso popolare da linguaggi come **Erlang** e framework come **Akka** (implementato in *Scala* e *Java*), offre una filosofia di calcolo che molti vedono come un nuovo stile di programmazione per l'era del multicore e del cloud.

Il focus non è più su cosa fare (Funzionale) o come cambiare lo stato (Imperativo), ma su come le entità computazionali (Attori) interagiscono in un ambiente massivamente concorrente e potenzialmente fallibile.

In un modello computazionale ad attori, la priorità è la comunicazione e la gestione dei messaggi.

- **Inscapsulamento Forte e Isolamento** Ogni Attore è un'entità completamente incapsulata. È “strutturalmente” un contenitore di stato e comportamento, ma il suo stato è rigorosamente isolato. Questo significa che la struttura interna di un Attore non è visibile o accessibile all'esterno. L'unico modo per interagire è attraverso l'invio di messaggi asincroni.
- **Il Messaggio come contratto** L'attenzione si sposta sulla definizione dei messaggi che verranno scambiati. Questi messaggi definiscono l'interfaccia pubblica e il contratto di **interazione** tra gli Attori. La logica del sistema diventa una complessa rete di conversazioni e risposte a messaggi, piuttosto che una gerarchia di chiamate a metodi sincroni.
- **Concorrenza per Composizione** Invece di preoccuparsi di *Lock* e *Thread* per gestire la concorrenza all'interno di un singolo componente ci si concentra su come comporre Attori indipendenti che lavorano in parallelo e coordinano il loro lavoro tramite lo scambio di messaggi.

Un linguaggio come **Akka** non solo sposta il focus sull'interazione, ma fornisce anche gli strumenti e i modelli comportamentali intrinseci (isolamento, asincronicità, supervisione) che risolvono le sfide più difficili della creazione di moderni **Microservizi** resilienti e scalabili.

4.7 Non determinismo

Se una qualche frase di un LdP induce un computer ad agire in due o più modi diversi, si ha una esecuzione non deterministica.

Esempi di forme di non determinismo si hanno:

- **Edsger Dijkstra** ha definito un costrutto teorico che introduce la **scelta non-deterministica** (usato per prove formali di correttezza)

```
if C1 -> S1 [] C2 -> S2 fi.
```

Se entrambe le condizioni (C_1 e C_2) sono vere, il sistema sceglie non deterministicamente se eseguire S_1 o S_2.

- in **Prolog**: La semantica di Prolog non specifica l'ordine in cui il motore di inferenza deve provare le regole o i fatti per soddisfare un goal. Il motore sceglie non deterministicamente (anche se le implementazioni reali seguono un ordine predefinito) un percorso, e se fallisce, “torna indietro” (**backtracking**) e prova un'alternativa.
- In **Akka**: L'ordine in cui gli attori ricevono i messaggi e l'esatto momento in cui elaborano i messaggi è non-deterministico.

Ad esempio, se l'Actor A invia due messaggi all'Actor B, non c'è garanzia che B elabori il Messaggio 1 prima del Messaggio 2 se c'è traffico di rete o ritardi. Questo non-determinismo può essere visto come un forza del modello, poiché costringe gli sviluppatori a scrivere codice resiliente a qualsiasi temporizzazione.

4.8 Influenza di un LdP sul progettista del software

Come già evidenziato da *Il motto di Alan Kay*, la struttura di un LdP può influenzare notevolmente il modo in cui il programmatore modella e risolve un problema. Ad esempio:

- **Prolog** (Logico): Incoraggia a pensare in termini di fatti e regole di deduzione, costringendo il programmatore a definire cosa è vero.
- **Java/C++** (OOP): Incoraggia a modellare il mondo in termini di oggetti, classi e relazioni gerarchiche.
- **Lisp/Scala/Haskell** (Funzionale): Incoraggia a pensare in termini di trasformazione di dati immutabili e composizione di funzioni, riducendo gli effetti collaterali.
- **Akka** (Attori): Incoraggia a pensare a un sistema come una collezione di (Micro)servizi che interagiscono usando Messaggi ed Eventi

La scelta del linguaggio impone un modello mentale e un modello di oggetto specifico:

- **OOP**: induce a pensare: “Quali entità esistono nel sistema e quali azioni possono eseguire su sé stesse?” Questo porta a un modello in cui lo stato (la realtà modellata) è distribuito e potenzialmente volatile.
- **FP**: induce a pensare: “Quali trasformazioni (funzioni) devo applicare a un insieme di dati immutabili per ottenere il risultato desiderato?” Questo porta a un modello più robusto per la gestione della complessità e della concorrenza.
- **Actor**: induce a pensare: “Quali attori esistono nel sistema e quali messaggi si devono scambiare e in quale modo?” Questo porta a un modello di sistema a (nano/micro)servizi come avremo modo di approfondire in seguito.

Il linguaggio non ha creato alcun oggetto reale, ma la sua struttura ha creato la sua rappresentazione digitale (*Funzione, Classe, Actor*) e ha dettato la logica di sistema (mutazione di stato vs. trasformazione pura vs. scambio di messaggi) che il software seguirà.

In questo senso, un LdP **orienta la soluzione architetturale** che il progettista sceglie per mappare un problema del mondo reale nella logica computazionale.

4.9 L'astrazione

Un LdP crea i **modelli di enti** con cui la logica del sistema opera. Questa fase definisce il **concetto di astrazione in informatica** e la sua relazione con la realtà, o, meglio, con l'insieme delle rappresentazioni umane del mondo perceptio come 'reale'.

4.9.1 Realtà vs. Modello

- “Oggetti Reali” (Il **Dominio del Problema**): Sono le entità, i concetti e le interazioni che esistono al di fuori del codice. Ad esempio, una persona, un conto bancario, un evento fisico (come un click del mouse) o una transazione finanziaria. Questi esistono indipendentemente dalla applicazione.
- “Modelli di Oggetti” (Il **Dominio della Soluzione**): Sono le rappresentazioni, o astrazioni, di quegli oggetti reali create all'interno del linguaggio di programmazione.

Quando si utilizza la parola “automobile” nel linguaggio naturale, si evoca un'immagine vaga e ricca di connotazioni. Quando si definisce una Classe Automobile in C++ o Java, si crea un modello preciso che ha solo le proprietà (velocità, colore, motore) e i comportamenti (*accelera, frena*) che sono stati esplicitamente codificati nel linguaggio.

- In C, un modello potrebbe essere la *struct* (struttura).
- In C++, un modello può essere la *Classe automobile*.

- In Prolog, fatti e le regole logiche definiscono le relazioni che coinvolgono l'automobile.
- In Akka, un modello potrebbe essere l'*Actor automobile* che differisce da tutti i precedenti in quanto esprime l'automobile come un ente intrinsecamente attivo.

In conclusione: **la programmazione è un esercizio di modellazione**. I linguaggi di programmazione sono strumenti formali e rigorosi che permettono di creare rappresentazioni manipolabili di entità per poter applicare una logica e risolvere un problema, senza cadere nell'ambiguità o nella relatività del linguaggio umano.

4.9.2 L'Abstraction gap

L'**Abstraction Gap** (*Divario di Astrazione*) è la distanza concettuale tra:

- Il **Dominio del Problema**: I concetti, le regole e le entità come sono compresi e descritti dagli esperti del dominio (es. un economista, un biologo, un esperto fiscale) usando il Linguaggio Naturale.

Esempio in LN: "Ogni cliente della banca deve avere al più un conto corrente in rosso".

- Il **Dominio della Soluzione**: Il modo in cui tali concetti devono essere mappati in strutture, tipi e logiche eseguibili nel *Linguaggio di Programmazione General Purpose* (GPL).

Esempio in LdP (Java/C++): Il concetto di "Cliente della banca" può essere modellato come un *Actor* che possiede uno o più (riferimenti a) *Oggetti* di classe *ContoCorrente* oppure come una riga di una tabella "Clienti" in un database relazionale in relazione con la tabella *ContoCorrente* o in vari altri modi.

Perché esiste l'Abstraction Gap

Nei GPL più comuni **non esistono** tipi nativi come *Microservizio*, *ConttoCorrent*, *ContrattoAssicurativo*, o *EquazioneTermodinamica*. Il programmatore deve tradurre manualmente il linguaggio del dominio nel linguaggio GPL, un processo che richiede tempo, è soggetto a errori di interpretazione e spesso diluisce l'intenzione originale del dominio.


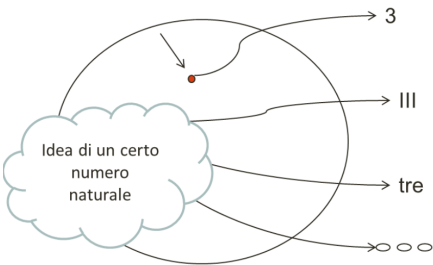
4.9.3 Modelli e Linguaggi di modellazione

Un **Linguaggio di modellazione** è un **Linguaggio formale** che può essere utilizzato per descrivere (modellare) un sistema di qualche natura.

Nel linguaggio comune, il termine *modello* è spesso usato per denotare un'astrazione di qualcosa che esiste nella realtà, come ad esempio il modello che posa per un artista, una riproduzione in miniatura, un esempio di modo di svolgere un'attività, una forma da cui ricavare vestiti, un ideale da seguire, etc..

Alcuni (tra cui gli ingegneri) intendono per modello un sistema matematico o fisico che ubbidisce a specifici vincoli e che può essere utilizzato per descrivere e comprendere un sistema (fisico, biologico, sociale, etc.) attraverso relazioni di analogia.

Modello=rappresentazione dell'essenza di un sistema

<ul style="list-style-type: none"> • Nei processi di costruzione del software, il termine modello va inteso come un insieme di concetti e proprietà volti a catturare aspetti essenziali di un sistema, collocandosi in un preciso spazio concettuale. • Per l'ingegnere del software un modello costituisce una visione semplificata di un sistema che rende il sistema stesso più accessibile alla comprensione e facilita il trasferimento di informazione e la collaborazione tra persone.  <ul style="list-style-type: none"> • Ma ... • L'insieme dei modelli che descrivono un sistema dovrebbe formare una descrizione completa, coerente, consistente e non (troppo) ridondante. 	 <p>Quattro diverse rappresentazioni espresse in quattro differenti linguaggi dello stesso concetto (modello).</p>
--	--

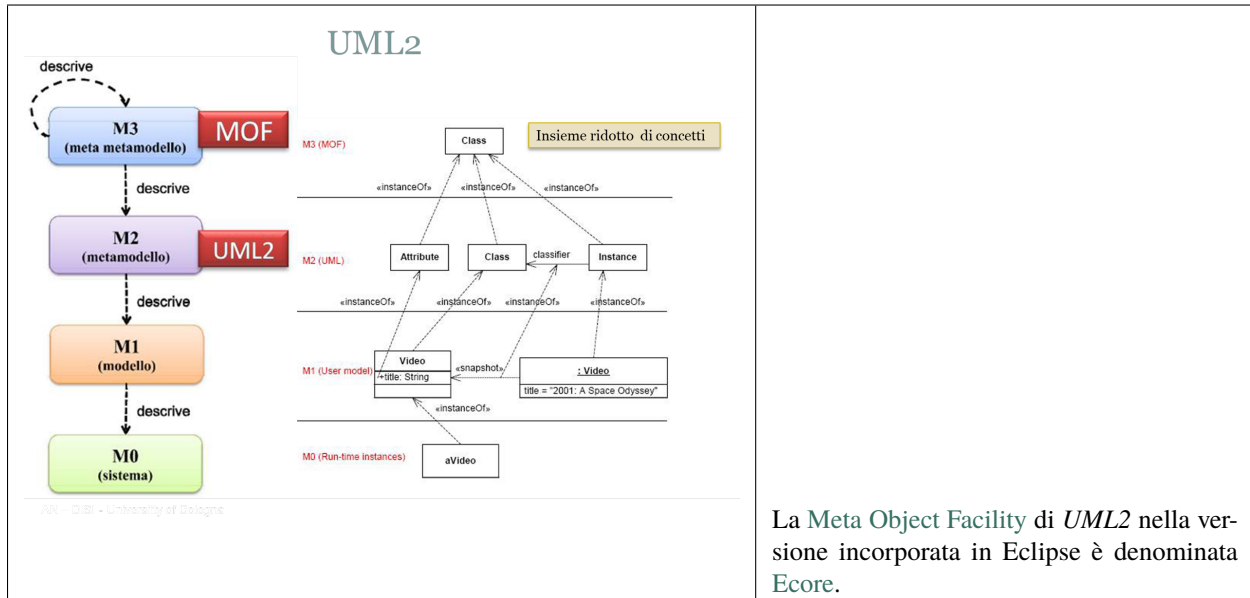
Nel concepire un modello come visione semplificata di un sistema software si assume che il sistema abbia già una sua esistenza concreta. In alcune fasi di lavoro (in particolare nella fase di analisi) il sistema è il modello; un raffinamento o una variazione del modello corrisponde in questo caso ad una variazione del sistema.

La produzione esplicita di modelli si rivela utile in quanto i diversi attori di un processo di produzione di software (committenti, analisti, progettisti, utenti, etc) operano a diversi livelli di astrazione e con fini diversi.

Definendo opportuni modelli del sistema da realizzare, in ogni fase del processo di produzione l'attenzione può essere focalizzata sugli aspetti rilevanti in quella fase, utilizzando una forma di comunicazione comprensibile ad attori diversi. Per garantire coesione e interoperabilità, si cerca di individuare regole di corrispondenza e di trasformazione automatica tra modelli.

4.9.4 UML

Il **Linguaggio UML** (*Unified Modeling Language*) è un linguaggio di modellazione e di specifica di sistemi software basato sul *Paradigma Orientato agli Oggetti (OOP)*. Il linguaggio è definito in termini di un **linguaggio di meta-modellazione** denominato MOF (si veda *Meta Object Facility*) .



Nel contesto dell'ingegneria del software, UML viene usato soprattutto per **descrivere** il dominio applicativo di un sistema software e/o il comportamento e la struttura del sistema stesso.

- **UML** è nato per la documentazione e il disegno: è enorme, include diagrammi di stato, di sequenza e molti concetti puramente grafici o metodologici. È spesso "complicato" perché deve coprire ogni possibile scenario di progettazione.
- **Ecore** è nato per l'implementazione: è un sottoinsieme di UML (spesso chiamato **EMF - Essential MOF**) ottimizzato per generare codice Java efficiente. In pratica, Ecore è ciò che resta di UML quando si toglie tutto ciò che non serve a definire strutture dati concrete.

Nel seguito, useremo *Ecore* per formalizzare la struttura di un nostro linguaggio di modellazione 'custom': Il linguaggio qak.

Il modello è strutturato secondo un **insieme di viste** che rappresentano diversi aspetti della cosa modellata (funzionamento, struttura, comportamento e così via), a scopo sia di analisi sia di progetto, mantenendo la tracciabilità dei concetti impiegati nelle diverse viste.

Il **Linguaggio UML** permette di definire modelli al termine di ogni fase del processo di sviluppo software:

- *Use Case Diagrams*: permettono di esprimere il risultato della fase di analisi dei requisiti;
- *class diagrams, sequence diagrams, activity diagrams, statechart diagram* permettono di esprimere la struttura e il comportamento il sistema al termine della analisi del problema e del progetto;
- *deployment diagrams* permettono di esprimere le modalità di distribuzione del prodotto.

4.9.5 Estensioni di UML

Tecnicamente, UML non è limitato al *Paradigma Orientato agli Oggetti (OOP)*, in quanto permette **estensioni** che lo possono rendere capace di modellare anche sistemi basati sul *Modello ad Attori*. Inoltre esistono **Profili UML** definiti da comunità di sviluppatori, come è stato nel caso di Akka.

Un modello UML standard è una specifica, mentre un modello **xUML** (*Executable UML*) o **fUML** (*Foundational UML*) è una rappresentazione formale che può essere utilizzata da un tool (come una piattaforma **MDE**: *Model-Driven Engineering*) per:

- **Interpretazione:** Eseguire direttamente il modello per simulare o testare il sistema.
- **Trasformazione** (Generazione di Codice): Generare codice sorgente eseguibile completo (codice della piattaforma) dal modello astratto.

Questi aspetti sono però una sorta di ‘add on’ al corpo principale di UML e il linguaggio qak che introdurremo più avanti vuole fornire uno strumento più adatto (in quanto esplicitamente orientato allo scopo) ad abbattere i costi di produzione di un sistema distribuito.

4.10 Domain-Specific Languages (DSL)

L'Abstraction gap è un dato di fatto strutturale nell'informatica e i *Domain-Specific Languages (DSL)* rappresentano oggi la tecnica più potente per riconciliare il ricco e complesso linguaggio di un certo dominio applicativo con i rigidi e universali GPL.

I DSL sono linguaggi creati per esprimere in modo conciso e non ambiguo i concetti e le regole di un dominio applicativo specifico. Il loro scopo primario è alzare il livello di astrazione per colmare *L'Abstraction gap* in modo più espressivo di quanto possa fare una libreria o un framework di un GPL.

SQL (Structured Query Language) È un primo esempio di **DSL**, strettamente **dichiarativo**, poiché si concentra sulla descrizione del **risultato desiderato** (i dati da recuperare) piuttosto che sulla sequenza di passi per ottenerlo.

Caratteristiche:

- **Non Procedurale:** Non si specificano i cicli o le condizioni di flusso.
- **Set-Oriented:** Le operazioni agiscono su insiemi di dati (tabelle).
- **Focus sui Dati:** La logica è incentrata su come manipolare e interrogare i dati relazionali.

I DSL possono essere classificati in base alla loro implementazione:

4.10.1 DSL Interno

Un DSL costruito all'interno di un GPL esistente (l'Host Language) viene anche detto **Embedded DSL**. Utilizza la sintassi e la semantica dell'Host per creare astrazioni di livello superiore.

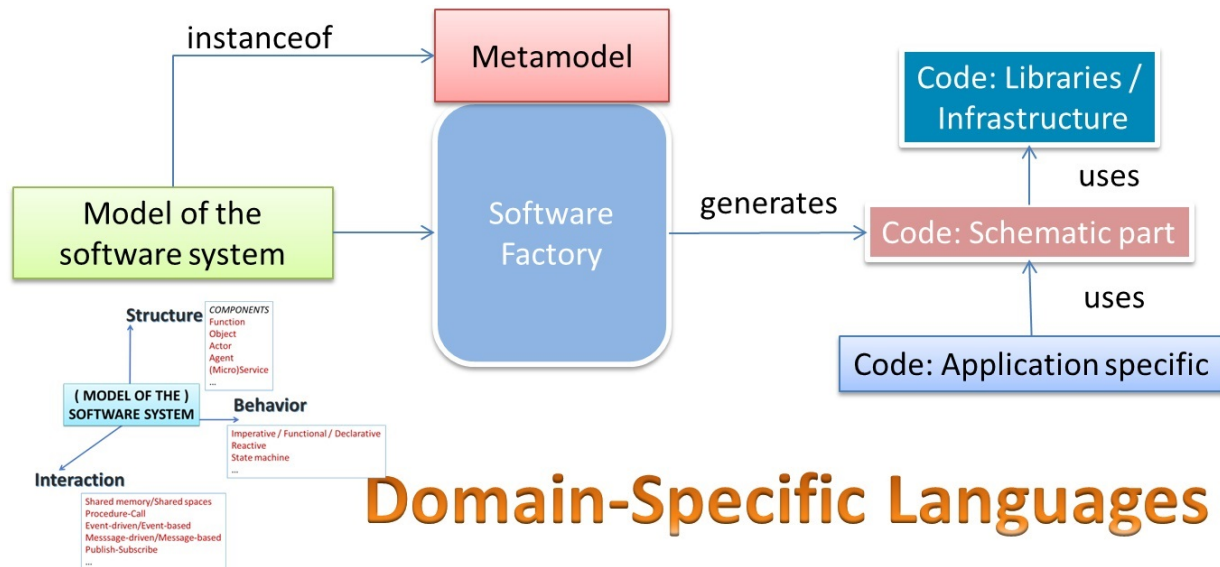
- **Vantaggio:** Facile integrazione, sfrutta l'infrastruttura dell'Host Language (es. compilatore, debugger).
- **Esempi:** LINQ in C#, le API Funzionali di Scala (es. le collezioni), l'uso di Builder Pattern in Java.

4.10.2 DSL Esterno

Si tratta di un linguaggio completamente nuovo, con la sua sintassi, parser e motore di esecuzione.

- **Vantaggio:** Massima espressività, può essere letto e scritto direttamente dagli esperti del dominio, totale indipendenza dal GPL.
- **Esempi:** SQL (per interrogare database), Regex (per pattern matching), YAML/JSON (per la configurazione).

Il framework **Xtext** si basa sull'ecosistema **Eclipse** e offre strumenti per ampliare Eclipse con appositi plugin per fornire un editor guidato dalla sintassi, e un generazione di codice (Java, Kotlin o altro) che realizza la semantica del linguaggio.



I DSL traducono direttamente concetti del LN in costrutti formali che sono facili da comprendere sia dagli esperti del dominio che dai programmatori.

Esempio: Nonostante Java non abbia una parola chiave “Microservizio”, un Framework (come *Akka* o *Spring Cloud*) può essere visto come un insieme di astrazioni che creano un DSL interno:

- Si usano annotazioni come `@Service` o si definiscono `Actor` (come in *Akka*) che mappano direttamente il concetto di unità isolata del microservizio.
- Questo “Linguaggio” all’interno del Framework permette agli sviluppatori di pensare e scrivere in termini di servizi anziché solo in termini di classi e thread.

Il linguaggio qak che introdurremo più avanti costituisce una *DSL Esterno* che permette di esprimere in modo diretto concetti (attore, messaggio, etc.) particolarmente rilevanti per la costruzione di sistemi software distribuiti a microservizi.

Vantaggi Principali a livello Applicativo

L’uso di DSL per colmare *L’Abstraction gap* offre vantaggi sostanziali:

- **Riduzione dell’Ambiguità:** I concetti del dominio vengono espressi in modo formale e non ambiguo (es. “calcolaInteressi” nel DSL fiscale è univoco, a differenza dell’LN).
- **Aumento della Produttività:** I programmatori lavorano con astrazioni più vicine al problema, riducendo la quantità di “boilerplate code” (codice standardizzato e ripetitivo).
- **Manutenibilità:** Quando le regole del dominio cambiano (es. una nuova legge fiscale), si modificano solo le definizioni nel DSL (che è ad alto livello), senza toccare gran parte della logica di basso livello del GPL.

4.11 I DSL e i Digital Twins

I DSL consentono agli esperti di un dominio (ingegneri meccanici, fisici, chimici) di esprimere la logica e le proprietà di un asset fisico senza dipendere da programmatori software o da GPL.

Questo aspetto è particolarmente rilevante per un sistemi software che svolge il ruolo di **Gemello Digitale (Digital Twin o DT)** di un sistema fisico.

Il termine *Digital Twin* è stato introdotto nel 2003 da *Michael Grieves* per denotare la replica digitale di una qualsiasi Entità Fisica che viene vista come la fonte dei dati su cui il **DT** opera per realizzare alcune funzionalità chiave:

- **Monitoraggio in Tempo Reale:** Il DT è alimentato da streaming data (telemetria) che riflette costantemente lo stato operativo, la salute e le prestazioni dell'asset fisico.
- **Simulazione e Previsione:** Permette di eseguire scenari What-If ("Cosa succede se...") sul gemello virtuale senza rischiare danni all'entità fisica.
- **Diagnostica e Ottimizzazione:** Viene utilizzato per identificare le cause profonde dei problemi e per testare le modifiche operative prima di implementarle nel mondo reale.
- **Manutenzione Predittiva:** Una delle applicazioni più potenti: il DT analizza le prestazioni attuali, le confronta con i modelli di degrado e prevede esattamente quando e dove un componente fallirà, permettendo la manutenzione prima del guasto.

Il DT fornisce una rappresentazione concreta di un *Sistema complesso*. Questo riduce l'Abstraction gap tra l'utente finale (che lavora con l'asset fisico) e il programmatore (che scrive il codice di controllo).

In questo campo, i DSL non sono solo strumenti di codifica, ma sono strumenti indispensabili per la **modellazione concettuale** che rendono il DT più accurato, manutenibile e interoperabile.

Un DSL può servire anche come linguaggio intermedio standardizzato per descrivere e integrare i dati provenienti da fonti eterogenee di dati generati da sistemi di controllo industriali (*PLC, SCADA*).

I DT sono intrinsecamente basati su **eventi**, guidati dal flusso continuo di dati **IoT (Internet of Things)**. I sistemi software che li gestiscono devono essere costruiti con architetture reattive e asincrone (come l'Actor Model o l'architettura a microservizi) per gestire l'alta velocità e la bassa latenza richieste.

Il DT eleva il ruolo del programmatore da semplice codificatore a **modellatore di sistemi e ingegnere di dominio**. Il valore non risiede più solo nello scrivere codice, ma nel:

- **Creare Modelli Accurati:** Sviluppare modelli matematici e fisici che replichino fedelmente il comportamento del mondo reale.
- **Ingegneria dei Dati:** Progettare data pipelines efficienti e sistemi di time-series database per gestire e pulire il volume massiccio di dati in tempo reale generato dal gemello.

4.12 Verso i sistemi software

Il *Digital Twin*, quindi, è un caso emblematico di come l'obiettivo di molti moderni sistemi software si sposta verso la proattività, la previsione e la conoscenza olistica dello stato operativo. La transizione dagli algoritmi ai sistemi software evocata nella *Evoluzione del concetto di Computazione* trova qui il suo più evoluto compimento.

Ma prima può essere opportuno affrontare il tema dei sistemi software in modo incrementale, procedendo dal semplice al complicato, fino a raggiungere livelli tipici dei sistemi complessi.

SISTEMI SOFTWARE

In ambito informatico, un Sistema è spesso descritto come:

Una combinazione strutturata di componenti che raccolgono, elaborano, memorizzano e distribuiscono informazioni per supportare servizi e processi decisionali e di controllo.

Storicamente, il software nasce come l'insieme di **istruzioni-macchina** che deve essere eseguito da un computer su un insieme di **dati** per realizzare un algoritmo.

L'evoluzione di linguaggi ad alto livello ha visto la rapida introduzione di costrutti capaci di esprimere semplici forme di aggregazioni di istruzioni e dati (componenti software).

L'uso del termine **sistema software** enfatizza, in genere, prodotti costruiti o studiati in termini della organizzazione e della interazione tra i componenti che formano il prodotto.

Quando si ha a che fare con un sistema software, è utile distinguere **due livelli**:

1. **Descrizione esterna** — come il sistema parla con l'esterno: API, messaggi, contratto di servizio.
2. **Comportamento interno** — come ogni componente (oggetto, attore, microservizio) evolve nel tempo: stato, transizioni, reazioni ai messaggi, logica interna.

Molti strumenti coprono solo il primo livello (descrizione dell'interfaccia — es. [OpenAPI](#)), altri solo il secondo (codice, FSM, logica interna). Ma per sistemi non banali servono entrambi in modo coerente.

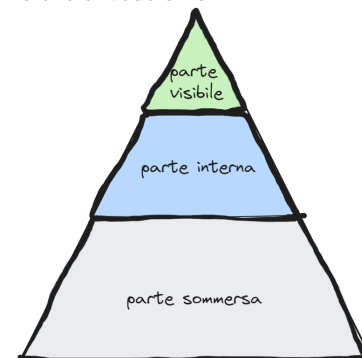
5.1 Viste di un sistema software

Un sistema software può essere descritto da diversi punti di vista:

L'osservatore di un sistema software può parlare di un sistema considerando diversi punti di vista

- *Vista esterna*
- *Vista interna*
- *Vista sommersa*

Quello che si vede e non



5.1.1 Vista esterna

Il sistema viene visto come una entità accessibile attraverso una *Le API* (**Application Programming Interface**) intesa come il punto di contatto che consente l'interazione tra codici in esecuzione

5.1.2 Vista interna

Il sistema viene visto come un insieme di enti computazionali (funzioni, oggetti, processi, etc.) che operano interagendo tra loro e con il mondo esterno (clienti, dispositivi, etc.) usando adeguati supporti

5.1.3 Vista sommersa

Il sistema è l'ultimo livello (*ayer*) di uno stack i cui livelli sottostanti (librerie, infrastruttura, etc.) forniscono il supporto alla esecuzione.

Nel corso degli anni sono stati proposti e usati diversi tipi di componenti software, che hanno contribuito a formare lo **spazio concettuale** dei linguaggi di programmazione di alto livello.

5.2 Componenti software

In generale, il concetto di **componente software** può essere definito come:


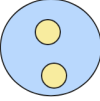
una unità autonoma e riutilizzabile di codice che svolge una funzione specifica *all'interno di un sistema software*, promuovendo la riutilizzabilità, la modularità e la manutenibilità *del* codice

Questa idea di base ha subito una profonda evoluzione nel corso degli anni, tanto che occorrerebbe un corso ad hoc per descriverla.

Trascurando molti rilevanti aspetti, noi cercheremo di delinearne alcuni punti importanti sia sul piano concettuale, sia sul piano pragmatico, relativi alle moderne pratiche di costruzione di sistemi software.

5.2.1 Componenti software di base

I linguaggi di programmazione più diffusi permettono di costruire sistemi software composti da due principali specie di componenti-base:

 funzione	Una funzione rappresenta un componente che permette la definizione parametrica di istruzioni e la loro esecuzione mediante trasferimento di controllo. <i>Linguaggio di riferimento:</i> JavaScript (Node.js)
pojo  oggetto	Un oggetto incapsula uno stato e un insieme di funzioni (metodi), la cui esecuzione avviene mediante chiamata di procedura (trasferimento di controllo). <i>Linguaggio di riferimento:</i> Java. Un oggetto può essere dotato di comportamento autonomo incapsulando un Thread .

I POJO

Nel seguito useremo il termine **POJO** (*Plain Old Java Object*) per denotare componenti software ‘vecchio stile’ che interagiscono mediante trasferimento del controllo (chiamate di procedura).

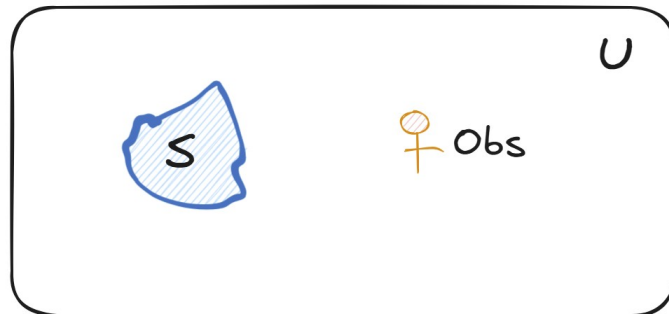
5.2.2 Oltre gli oggetti

Lo sviluppo delle reti informatiche e di Internet ha promosso la costruzione di **sistemi software distribuiti** su più nodi di elaborazione, su ciascuno dei quali possono essere eseguiti programmi espressi mediante linguaggi diversi.

- La costruzione di **sistemi software distribuiti eterogenei** è resa possibile dall’uso (entro funzioni ed oggetti) di protocolli di comunicazione (come UDP, TCP, HTTP, MQTT, CoAP, etc.) e richiede logicamente componenti capaci di interagire (spesso in modo asincrono) mediante **scambio di messaggi** e non più mediante trasferimento di controllo con chiamate di procedura.
- Ciò ha portato alla definizione di nuovi tipi di componenti software quali gli *Attori*, gli agenti software (intelligenti), i *Microservizi*, le *Funzioni serverless*, etc.

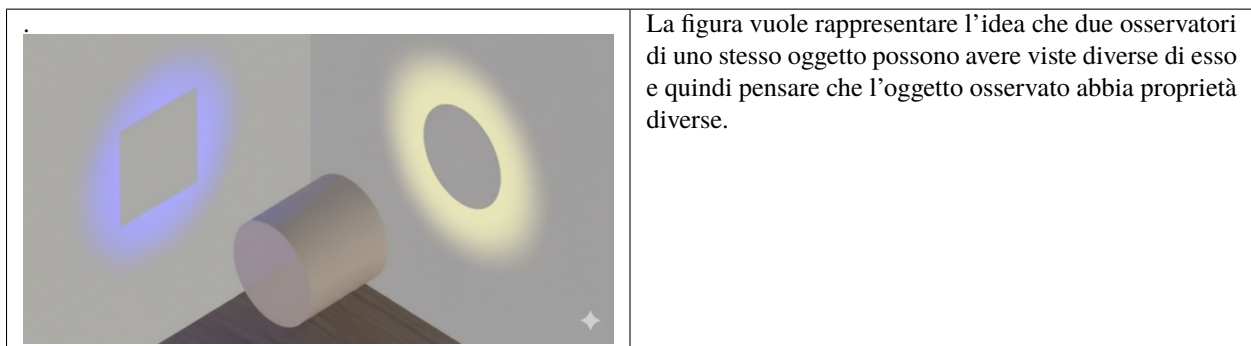
5.3 Un viaggio nei sistemi software

In questa sezione cercheremo di parlare, in linguaggio naturale, di un sistema denominato **S** supponendo che sia un *sistema hardware/software* con cui sia possibile interagire via rete.



Il sistema **S** è posto in un ‘universo’ (**U**) in cui si trova anche un osservatore (umano o meno) **Obs**, che può interagire con S in molti modi diversi.

5.3.1 Punti di vista



La figura vuole rappresentare l’idea che due osservatori di uno stesso oggetto possono avere viste diverse di esso e quindi pensare che l’oggetto osservato abbia proprietà diverse.

Dire che **S** è un sistema **hardware/software** induce a pensare che sia costruito usando un **computer** (hardware) capace di elaborare **istruzioni** (software) che determinano il comportamento osservabile del sistema.

Il *sistema globale* $S + Obs$ è un sistema distribuito di cui può essere difficile *parlare in modo preciso* senza un opportuno linguaggio capace di **esprimere gli aspetti salienti della interazione/comunicazione**, lasciando sullo sfondo dettagli importanti, ma non essenziali.

In assenza di ulteriori informazioni, **S** potrebbe essere visto come un **servizio**, pensando che il sistema sia in grado di **fornire funzionalità a clienti** (umani o programmi) che ne fanno richiesta usando protocolli di rete standard, come HTTP e WS (*WebSocket*) o altri.

Queste sommarie indicazioni sono già sufficienti a dare una idea di cosa stiamo parlando comprensibile a **molti** esseri umani, ed in particolare a coloro che sono abituati a usare **Internet**. Osserviamo però che questo modo di ‘capire’ cosa sia **S** era **praticamente impossibile** solo pochi decenni fa, quando i *computer* e *Internet* non erano così diffusi e facilmente accessibili.

Tuttavia, anche in comunità umane in cui il termine **servizio** (software) è ormai di uso comune, non è certo scontato che sia a tutti ‘chiaro’ cosa sia un *servizio*, come sia fatto e come funzioni. Inoltre, anche i più esperti del settore potrebbero trovarsi in difficoltà di fronte a domande del tipo:

Domanda1: che differenza c’è tra un servizio e un normale **programma** che gira su un computer? Certo si può presumere che un servizio sia un programma, ma allora cosa lo rende un **servizio**?

Domanda2: che differenza c’è tra un *servizio* e un **microservizio** ?

In attesa di rispondere a domande di questo tipo, al momento osserviamo solo che:

- **tutti i sistemi software servono a qualcosa ma non tutti sono servizi**

5.3.2 Sistemi Embedded/IoT

I computer oggi in uso sono spesso dotati di dispositivi di interazione con il mondo fisico (si pensi a un *RaspberryPi* e ai suoi PIN **GPIO** (*General Purpose IO*) e rappresentano una categoria qualitativamente diversa dai computer generici che elabora solo dati.

Un **sistema embedded** è un sistema di calcolo dedicato a svolgere una o poche funzioni specifiche, integrato all’interno di un prodotto o dispositivo fisico. Quando questi sistemi sono connessi in rete — spesso a Internet — e possono interagire con altri dispositivi o servizi digitali, si parla di **sistemi IoT** (*Internet of Things*).

RaspberryPi

Il *Raspberry Pi* è una piccola scheda computer, economica e a basso consumo, molto utilizzata per realizzare sistemi embedded e IoT grazie a:

- un sistema operativo completo (generalmente Linux)
- interfacce di comunicazione evolute (Wi-Fi, Bluetooth, Ethernet)
- GPIO per la connessione diretta a sensori e attuatori fisici
- una comunità e un ecosistema di librerie molto vasti

Sistemi cibernetici

Un computer interfacciato con il mondo fisico rappresenta un **sistema cibernetico** la cui funzione primaria è modificare o preservare lo stato del mondo reale. Come tale, è un sistema — naturale, artificiale o misto — capace di raggiungere o mantenere uno stato desiderato (un obiettivo):

- percependo l'ambiente attraverso sensori,
- valutando l'errore rispetto all'obiettivo,
- agendo sull'ambiente tramite attuatori,
- chiudendo il ciclo tramite un meccanismo di **feedback**.

5.4 Capire sperimentando

Il problema di capire/definire cosa sia S può essere affrontato seguendo un approccio analogo a quello di un ricercatore che vuole studiare un nuovo oggetto fisico. In questo caso, il ricercatore potrebbe iniziare a porsi alcune domande del tipo:

1. cosa fa questo oggetto?
2. come funziona?
3. come è fatto?
4. quali sono le sue proprietà più interessanti?

Notiamo che le domande 1 e 2 implicano una certa **osservazione** dell'oggetto in azione, mentre le domande 3 e 4 richiedono una certa **analisi** dell'oggetto, magari smontandolo per vedere come è fatto dentro.

5.4.1 Capire interagendo

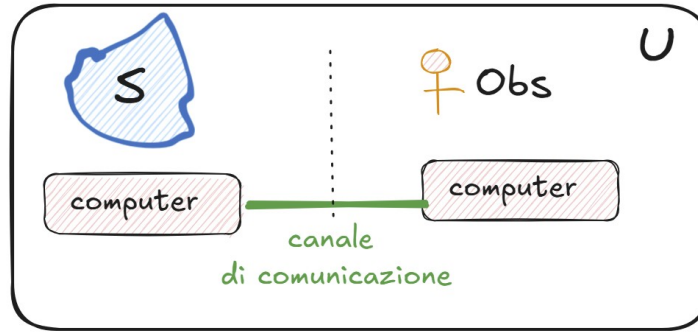
Sembra ragionevole pensare che, prima di 'smontare' un oggetto, sia meglio intergere con l'oggetto in una *fase di 'pura osservazione'*, per cercare di ricavarne informazioni utili per il suo utilizzo e per **inferire**, se possibile, alcune delle sue proprietà.

Siamo qui di fronte a una delle problematiche più interessanti e controverse della scienza: come si fa a studiare un oggetto nuovo di cui non si sa nulla? Anche una questione che sembra banale, come la *'semplice (!) osservazione'*, è oggi un tema di ricerca avanzata in molti campi, dopo che la **meccanica quantistica** ha mostrato che l'atto stesso dell'osservazione può influenzare (prima ancora di 'smontarlo') l'oggetto osservato o addirittura **determinarne alcune proprietà**.

Fortunatamente, per i nostri scopi, potremo trascurare queste proprietà che sembrano caratterizzare il mondo in cui viviamo, almeno fino a quando non dovremo occuparci di sistemi software che usano il *Quantum computer*.

5.4.2 Interagire con un sistema software

Nel caso in cui S sia un sistema software, è implicito il fatto che la interazione richiede l'uso di un computer e/o di una rete di comunicazione.



5.4.3 Capire implica opportuni livelli di conoscenza

L'informazione scambiata tra un sistema software *S* e *Obs* è di solito rappresentata in forma di stringa usando caratteri quali UTF-8. Ma questo dettaglio di basso livello è piuttosto **irrilevante per catturare gli aspetti salienti** della comunicazione.

Questo aspetto è ancora più evidente considerando che noi ‘sappiamo’ che *S* (o almeno il suo *hardware* è senza dubbio formato da *atomi* (su questo punto ci sarebbe da approfondire ...); tuttavia nessun essere umano penserebbe che questo livello di conoscenza sia necessario (o meglio **il livello ‘giusto’**) per capire *S*.

In altri termini, per capire *S* non è oggi necessario (al momento almeno) studiare discipline come la *fisica* o la *chimica* o la *biologia* o la *psicologia*. Pensandoci bene, non è nemmeno necessario studiare **calcolatori elettronici**, perchè è implicito che quello che fa *S* è determinato dal *software*, che ha nell'*hardware* del computer, la sua premessa indispensabile, il suo ‘supporto vitale’ ma non gli elementi necessari per fornire una adeguata ‘spiegazione’ di *S*.

Potrebbe anche non essere nemmeno necessario addentrarsi nella parte di software nota come **sistema operativo**: per capire pienamente *S* sarebbe però necessario conoscere il **linguaggio di programmazione** (o i linguaggi) con cui è stato specificato il comportamento di *S*: al più è necessario conoscere anche il rapporto tra il linguaggio e il primo livello di software limitrofo sottostante.

Ma prima di ‘aprire il sistema’ per vedere come è fatto ‘dentro’, proviamo a capirlo seguendo l’idea meno invasiva di **comunicare** con esso.

Tenendo conto del moderno punto di vista scientifico, non possiamo dimenticare che osservatore e osservato sono parte dello stesso ‘universo’ e che il funzionamento di *S* potrebbe essere legato non solo al modo in cui l’osservatore interagisce direttamente con esso, ma anche da altri fattori, ad esempio la temperatura e la pressione dell’ambiente, flare solari, etc.

Tuttavia è ragionevole pensare che, per studiare *S*, l’osservatore possa limitarsi a interagire con esso usando un computer come unico ‘strumento di interazione’, senza preoccuparsi troppo di altri fattori esterni (come ha proposto di fare per primo *Galileo*, al fine di distinguere ciò che è necessario da ciò che può essere ignorato in prima istanza).

5.4.4 Conoscere per comunicare

Per poter **comunicare proficuamente** con *S*, l’osservatore *Obs* deve conoscere:

1. il **protocollo di comunicazione (P)** che permette al computer che supporta *S* di ricevere/trasmettere informazione e quindi di interagire con il computer usato da *Obs*;
2. il **vocabolario/linguaggio** che *S* usa per denotare l’informazione trasmessa/ricevuta come **messaggi** usando *P*;
3. il **legame logico della interazione** che si stabilisce in ogni particolare tipo di comunicazione tra due enti.

La interazione come vincolo

La necessità di conoscere il **legame logico della interazione** nasce dal fatto che vi sono varie forme di comunicazione e che queste comportano diverse forme di interazione logica tra gli enti comunicanti. Ad esempio, il sistema S potrebbe essere:

1. capace di rispondere a **richieste** provenienti da uno o più Obs, inviando la risposta a chi ha fatto la richiesta. Se due Obs diversi inviano richieste a S, ognuno di loro si aspetta di ricevere una risposta pertinente alla propria richiesta, senza confusioni.

Note: Notiamo che l'invio di una richiesta non implica che Obs si blocchi in attesa della risposta. in tal caso si avrebbe una **operazione sincrona** e il canale di comunicazione rimarrebbe aperto fino alla risposta.

Obs potrebbe inviare la richiesta come **operazione asincrona**; in tal caso il canale di comunicazione potrebbe chiudersi subito dopo l'invio della richiesta e S dovrebbe trovare modi opportuni per inviare la risposta che Obs si aspetta e che elaborerà quando sarà 'disposto a farlo'.

2. **emettere informazioni** (dette **eventi**) in modo spontaneo verso il mondo esterno, che potrebbero essere 'captate' da uno o più Obs (si pensi ad esempio a un servizio che avvisa su allarmi, congestione o blocchi stradali, etc.) Questa forma di comunicazione viene detta **Publish-subscribe** e viene supportata da broker come RABBITMQ, Mosquitto, HiveMQ, Kafka.
3. capace di elaborare messaggi inviati in modo **fire-and-forget**, impegnandosi a gestirli come **comandi da eseguire** e/o a restituire risultati in forma di **notifiche/eventi**.

Ognuna delle forme di comunicazione citate costituisce un **vincolo** tra gli enti comunicanti, che devono agire ciascuno in modo coerente con l'altro. E' simile al tipo di vincolo che lega due danzatori.

Anche questo semplice e vago elenco di possibilità mostra come possa essere difficile *parlare in modo preciso* di un sistema software senza un opportuno linguaggio capace di **esprimere gli aspetti salienti della interazione/comunicazione**, lasciando sullo sfondo dettagli importantissimi, ma non essenziali per la definizione e la comprensione del *sistema globale S + Obs*.

5.5 Sistemi software come servizi (ma non solo)

Il sistema software S risulta accessibile a Obs attraverso l'uso di un particolare sistema fisico (il **computer**) che ha la capacità di inviare/ricevere informazioni usando reti cablate e/o wireless e opportuni **protocolli** di comunicazione quali HTTP, TCP, MQTT, WS ed altri.

5.5.1 L'infrastruttura

Perché il computer usato da Obs (**Client**) possa interagire con S occorre disporre, allo stato attuale della tecnologia, di un ponte di comunicazione.

- **Networking:** Entrambi i computer devono essere sulla stessa rete (LAN) o connessi tramite VPN/Internet. Occorre conoscere l'Indirizzo IP del computer su cui gira S.
- **Wrapping:** Se S è un programma "chiuso" (es. uno script che gira solo da terminale), occorre aggiungere uno strato software (un *Framework* come **Flask** per Python, **Express** per JS, SpringBoot o ASP.NET) che riceva le chiamate su rete e le "passi" a S.
- **Port:** occorre scegliere un "canale" numerico (es. **8080**) su cui il server rimarrà in ascolto.
- **Firewall:** Sul computer di S, occorre aprire la porta scelta nel firewall per permettere il traffico in entrata.

I framework

Un **framework** è una struttura logica di supporto su cui uno sviluppatore può costruire software.

A differenza di una semplice libreria, il framework fornisce un'architettura predefinita e un insieme di strumenti, componenti e regole che guidano lo sviluppo, permettendo di non dover riscrivere codice comune da zero

Note: Molti framework gestiscono le comunicazioni client-server, solitamente via HTTP/HTTPS o [WebSocket](#); tra questi: Node.js (Express), FastAPI (Python), javalin, Spring Boot (Java).

Nel campo dei Web, *React* e *Angular* sono framework per **Frontend**, mentre Django (Python), Laravel (PHP), Spring (Java) sono framework per **Backend**.

Le caratteristiche principali di un framework includono:

- **Inversione del controllo (IoC):** A differenza delle librerie, dove il programmatore chiama il codice esterno, in un framework è la struttura stessa a chiamare il codice scritto dall'utente nei punti appropriati.
- **Estendibilità:** Permette di personalizzare specifiche parti del software per aggiungere funzionalità.
- **Codice predefinito:** Include soluzioni già pronte per compiti ripetitivi (come la gestione dei database o l'autenticazione degli utenti).

Un volta stabilita una connessione, il computer **Client** è analogo ai tipi di strumenti telescopi, microscopi, acceleratori di particelle, etc.) usati dagli scienziati.

Ovviamente, la conoscenza di quale protocollo (**P**) usa **S** per interagire con il mondo esterno non è sufficiente: occorre anche conoscere la *Sintassi e semantica* delle 'frasi' veicolate tramite **P**. In altre parole, **S** definisce un **proprio vocabolario**, o più in generale un **proprio linguaggio** che **Obs** deve conoscere per interagire in modo proficuo con **S**.

5.5.2 Le API

Nei sistemi software accessibili in rete, di solito si esegue un mapping tra le capacità interne del sistema e un'interfaccia esterna, l'**API** (*Application Programming Interface*) che stabilisce il "contratto" che definisce quali informazioni il sistema può ricevere e cosa risponderà.

Senza **API standardizzate**, non ci sarebbe un accordo condiviso su come le diverse applicazioni o i servizi software comunicano. I programmatori di due applicazioni separate dovrebbero parlare tra loro per determinare ogni volta come sviluppare lo scambio di dati.

Di solito una API specifica almeno:

- **Input/Output:** Quali argomenti accetta una specifica funzionalità di **S**? Che tipo di dati restituisce? (Es. una stringa, un numero, un'immagine?).
- **Protocollo:** Il più comune, nei servizi software, è **HTTP**, seguendo lo stile architetturale *Representational State Transfer (REST)* (**Representational State Transfer**).

Aspetti importanti in relazione all'uso di un sistema sono anche:

1. Gestione dello Stato

- **Stateless:** Ogni richiesta è indipendente (es. una funzione che calcola la radice quadrata).
- **Stateful :** **S** ricorda le azioni precedenti (es. un carrello della spesa). In questo caso, il client deve inviare un "ID sessione" per farsi riconoscere.

2. Concorrenza

S è in grado di gestire più richieste contemporaneamente? Se due client chiamano la stessa funzione nello stesso istante, S mette in coda le richieste? Se S non è multi-thread, il *Iframework* che funge da wrapper della *L'infrastruttura* dovrà realizzare e gestire la coda.

Al giorno d'oggi esistono diverse soluzioni consolidate per descrivere in modo formale (le interfacce di) servizi software, alcune delle quali, largamente usate, sono descritte nel capitolo APIDescription.

Non tutte sono “formalismi puri” in senso teorico, ma piuttosto **linguaggi di descrizione contrattuale** (interface description languages, IDL / API description languages).

5.5.3 Representational State Transfer (REST)

REST è uno stile architetturale per sistemi distribuiti. L'espressione “representational state transfer” e il suo acronimo, REST, fu introdotto nel 2000 nella tesi di dottorato di **Roy Fielding**.

REST rappresenta un sistema di trasmissione di dati su HTTP senza ulteriori livelli (quali ad esempio **SOAP**). I sistemi REST non prevedono il concetto di sessione, ovvero sono **stateless**.

REST prevede che la scalabilità del Web e la sua crescita siano risultati di pochi principi chiave di progettazione:

- lo stato dell'applicazione e le funzionalità sono divisi in **risorse web**
- ogni risorsa è unica e indirizzabile usando sintassi universale per uso nei link ipertestuali

Il funzionamento di REST prevede una struttura degli **URL** (*Uniform Resource Locator*) ben definita che identifica univocamente una **risorsa** o un insieme di risorse e l'utilizzo dei metodi HTTP specifici per il recupero di informazioni, per la modifica e per altri scopi (**OPTIONS**, ecc.):

- **GET**: “Dammi le informazioni” (non modifica nulla).
- **POST**: “Crea qualcosa di nuovo” (invia dati).
- **PUT/PATCH**: “Aggiorna qualcosa che esiste già”.
- **DELETE**: “Cancella una risorsa”.

REST è ideale per creare un'API pubblica per sviluppatori esterni o un servizio-web semplice. È lo **standard universale** e il più facile da testare.

HATEOAS

HATEOAS è l'acronimo di **Hypermedia as the Engine of Application State** ed è un componente dell'architettura e della progettazione delle API RESTful.

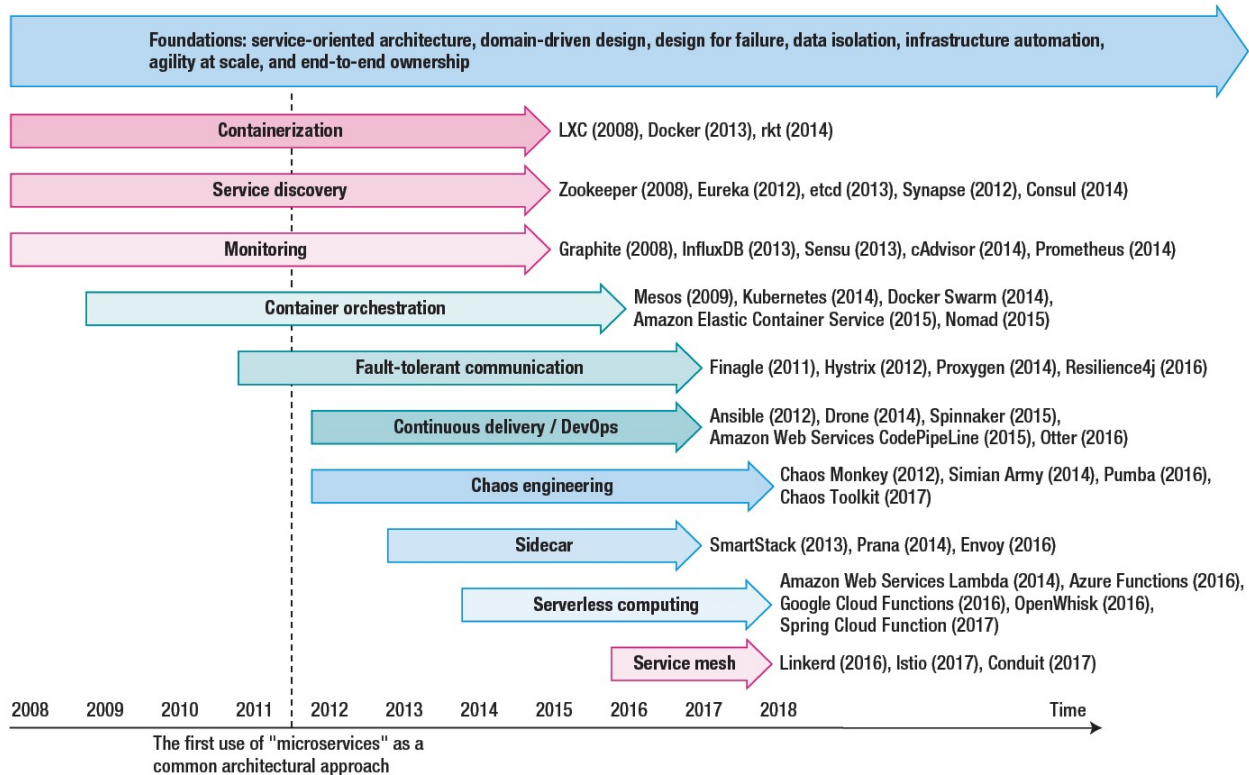
Utilizzando l'architettura HATEOAS, un client potrà accedere all'API di un'applicazione tramite una semplice chiamata URL RESTful. Qualsiasi ulteriore azione che il client desideri intraprendere sarà abilitata dai dati restituiti dal server nella chiamata originale.

I “dati”, all'interno della risposta, che consentono questo cambiamento di stato sono semplici collegamenti ipermediali. Ciò consentirà al client di passare da uno stato dell'applicazione al successivo semplicemente interagendo con i dettagli contenuti nelle risposte del server.

5.5.4 Dai servizi ai microservizi

I sistemi software composti da microservizi rappresentano l'apice di un'evoluzione che ha visto i sistemi informatici trasformarsi da semplici programmi a complicate infrastrutture distribuite.

Le principali tappe che hanno segnato questo cambiamento possono essere visualizzate e riassunte come riportato in [mshistory](#).



Le motivazioni che spingono le aziende verso i microservizi sono molteplici e spesso legate alla necessità di gestire sistemi complessi e scalabili, migliorare la resilienza, accelerare l'innovazione, e supportare la trasformazione digitale.

Il tema verrà approfondito nel capitolo dedicato ai Sistemi a microservizi

Oltre HTTP

Considerati i limiti di HTTP, vi sono molte applicazioni in cui è preferibile utilizzare strumenti di interazione più efficienti ed ottimizzati, come il protocollo *CoAP* o, più recentemente, gRPC.

5.5.5 CoAP

Il protocollo *CoAP* (**C**onstrained **A**pplication **P**rotocol) è stato progettato specificamente per portare il paradigma RESTful (risorse, metodi, URI) in ambienti dove l'overhead di HTTP e TCP sarebbe insostenibile.

Mentre HTTP è il protocollo del **Web delle persone**, CoAP è il protocollo del **Web delle cose (IoT)**.

REST tradizionale usa HTTP su TCP. Questo comporta un "peso" notevole per dispositivi alimentati a batteria o con poca memoria. CoAP ottimizza questa struttura agendo su tre livelli:

1. Dal Testo al Binario

HTTP è testuale (leggibile dagli umani), il che lo rende verboso. CoAP è binario e utilizza un header fisso di soli 4 byte. In CoAP, un intero messaggio può stare in meno di 10-20 byte.

2. Da TCP a UDP

TCP richiede un “handshake” (3 passaggi) per stabilire la connessione e gestisce la ri-trasmissione in modo complesso. CoAP gira su UDP, che è “senza connessione”. CoAP implementa la propria affidabilità (*Messaggi Confirmable*) solo quando serve, risparmiando energia e traffico di rete.

3. Non solo Request/Response

In REST/HTTP, per un aggiornamento, occorre fare polling (chiedere continuamente). CoAP introduce il **meccanismo Observe**: un client si “iscrive” a una **risorsa** e il server invia un aggiornamento solo quando il valore cambia. È un ibrido tra REST e Publish/Subscribe.

Messaggi Confirmable

I messaggi Confirmable (**CON**) rappresentano il meccanismo principale per garantire in CoAP l'affidabilità della comunicazione sopra il protocollo di trasporto UDP, che di per sé non è affidabile.

Un messaggio contrassegnato come Confirmable richiede una conferma esplicita dal destinatario.

- **Acknowledge (ACK)**: Quando il server riceve un messaggio CON, deve rispondere con un pacchetto ACK.
- **Message ID**: Per collegare correttamente la conferma al messaggio originale, l'ACK deve contenere lo stesso Message ID del messaggio CON inviato.
- **Ritrasmissione**: Se il mittente non riceve l'ACK entro un determinato intervallo (timeout), il messaggio CON viene inviato nuovamente. Questo processo continua fino a quando non arriva la conferma o viene raggiunto il numero massimo di tentativi.

Se il server CoAP può elaborare la richiesta immediatamente, include i dati della risposta direttamente nel pacchetto ACK (**Piggybacked Response**). Altrimenti, invia prima un ACK vuoto (per fermare le ritrasmissioni del client) e, una volta pronti i dati, invia un nuovo messaggio CON verso il client, che a sua volta dovrà rispondere con un ACK (**Separate Response**).

5.5.6 WebSocket

Il protocollo *WebSocket* (**WS**) è nato nato per risolvere il limite della comunicazione unidirezionale di HTTP.

WS consente a due o più computer di comunicare tra loro in modo full-duplex su una singola **connessione TCP**. È uno strato molto sottile su TCP che trasforma un flusso di byte in un flusso di messaggi (testo o binario).

- Con REST, per sapere se ci sono novità, occorre chiedere continuamente (**Polling**). In WebSockets, si apre la connessione una volta e il server emette i dati appena sono pronti. Ciò aumenta l'efficienza.
- Tuttavia, gestire migliaia di connessioni sempre aperte è costoso in termini di memoria del server. REST è più efficiente per operazioni “mordi e fuggi” (es. leggere un articolo di un blog). Dunque, **WS** non è un sostituto totale di REST:
- **WS** ha come uso ideale: Chat, notifiche live, cruscotti finanziari, giochi online.

Nella sua forma più semplice,

un WebSocket è solo un canale di comunicazione tra due applicazioni

e non deve essere necessariamente coinvolto un browser. Tuttavia l'uso più comune di WebSocket è facilitare la comunicazione tra un'applicazione server e un'applicazione basata su browser, avendo il vantaggio di realizzare comunicazioni asincrone bidirezionali e in tempo reale.

Mentre in HTTP ogni singola richiesta porta con sé il “metodo” (GET/POST) e il “path” (/xxx), nel protocollo WebSocket, una volta stabilita la connessione, i messaggi sono solo **sequenze di byte o testo** senza alcun metadato di instradamento.

Sorge un problema: una volta che aperta una **WS** verso un servizio, occorre sapere cosa inviare e cosa aspettarsi. Senza uno standard, l'unico modo è leggere il codice Java o qualche documentazione scritta a mano.

A questo scopo è stato introdotto AsyncAPI, di cui parliamo nell'apposito capitolo.

La fase di connessione nelle WS

Ogni WebSocket inizia la sua vita come una normale richiesta **HTTP GET**.

- Il client invia: *GET /xxx HTTP/1.1* con l'intestazione *Upgrade: websocket*.

Questo **instradamento della connessione** apre la via a diverse forme di *Instradamento del messaggio*, in accordo a diverse tecniche che permettono a un **server WebSocket** di gestire molteplici funzionalità sulla stessa porta.

Instradamento del messaggio

Volendo inviare ordini diversi (es. “fai questo”, “fai quello”) sulla **stessa** connessione WebSocket, il protocollo non aiuta perché non ci sono “header” nei messaggi. Per risolvere questo “vuoto” di informazioni, i server e gli sviluppatori usano tre strategie:

- A. Il modello “**Un path, una funzione**”

È la soluzione più semplice. Creiamo connessioni diverse per scopi diversi.

- Connessione A -> /xxx1
- Connessione B -> /xxx2

Il “routing” avviene solo all'inizio. È adeguato se le funzionalità sono logicamente separate.

- B. **Sotto-protocolli** (Sub-protocols)

Il protocollo WebSocket permette di definire un “sotto-protocollo” (es. **STOMP** o **MQTT**). Questi aggiungono un piccolo header a ogni messaggio che dice: “Questo messaggio è per la coda X”. Ciò complica il codice del client.

- C. L'**Envelope Pattern** (Il “**Facade**” logico)

Visto che il protocollo è “muto”, avvolgiamo il messaggio in una ‘busta’ (*envelope*) che contiene le istruzioni di routing. È lo standard nell'industria moderna.

Esempio di envelope JSON:

```
{
  "action": "xxx1",
  "data": "il mio input"
}
```

In questo caso, un server riceve il messaggio come stringa generica, lo decodifica e usa uno *switch* nel codice Java per decidere cosa fare.

Il routing non lo fa il protocollo, lo fai il codice applicativo.

javalin server

Javalin è quello che in gergo viene definito un micro-framework per Java e Kotlin. Javalin è per Java quello che **Express.js** è per *Node.js* o **Flask** è per *Python*.

A differenza di **Spring Boot**, che è un “ecosistema” vasto e spesso pesante, *Javalin* è progettato per essere estremamente leggero, semplice da imparare e privo di configurazioni (niente file XML o annotazioni @).

Note: Dalla **versione 6**, *Javalin* è diventato molto modulare, per cui è opportuno usare la dipendenza **javalin-bundle**, che include tutto il necessario per far girare WebSockets e rendering di template.

Come opera javalin

Javalin sfrutta il fatto che ogni WebSocket inizia la sua vita come una normale richiesta **HTTP GET**.

1. Il client invia: *GET /xxx1 HTTP/1.1* con l'intestazione *Upgrade: websocket*.
2. Javalin legge il path */xxx1* come se fosse una normale pagina web.
3. In quel momento, Javalin “stacca” quella specifica connessione e la associa internamente al blocco di codice che hai scritto per */xxx1*.

Da questo punto tutti i messaggi che arrivano su quella connessione verranno passati alla funzione *xxx1*. **Il protocollo non instrada i messaggi, il server ricorda a quale “binario” appartiene la connessione.**

In sintesi

Javalin usa le informazioni dell'HTTP iniziale per separare i flussi. Una volta che la connessione è aperta, il protocollo WebSocket è effettivamente un “tubo cieco”:

- Volendo **ordine e semplicità**, è opportuno usare path diversi (*/xxx1*, */xxx2*).
- Volendo **un solo path**, occorre aggiungere le informazioni di instradamento dentro il messaggio (solitamente JSON) come detto in *Instradamento del messaggio*.

Javalin:

- è **Imperativo/Bloccante**: segue il modello “One Thread per Request”. Quando arriva una richiesta, viene assegnato un thread che rimane occupato finché la funzione non ha finito. È facile da leggere e da debuggare perché il codice segue un flusso lineare.
- è un Framework Web: È progettato quasi esclusivamente per fare HTTP e WebSocket. È un “guscio” attorno al *Server Jetty*.
- è ideale per trasformare una funzione semplice in un servizio accessibile

Server Jetty

Jetty è un server web e un container di servlet Java, leggero, gratuito e open source, gestito dalla Eclipse Foundation

A differenza di server più “monolitici” come Apache Tomcat, *Jetty* è nato per essere incorporato direttamente nel codice di un'applicazione.

È ottimizzato per gestire un numero elevatissimo di connessioni simultanee con un basso consumo di memoria, rendendolo ideale per applicazioni ad alta concorrenza.

Server Vert.x

Un server javalin non è ideale se un sistema

- deve gestire decine di migliaia di connessioni WebSocket aperte contemporaneamente (es. una chat massiva o un sistema di notifiche push)
- ha bisogno di resilienza: se un componente fallisce, gli altri devono continuare a girare isolati
- vuole usare più linguaggi di programmazione

Una alternativa è impostare il server usando **Vert.x**:

- è **Reattivo/Non-bloccante**: si basa sul modello Event Loop (simile a Node.js). Un solo thread (o pochissimi) gestisce migliaia di connessioni contemporaneamente. Se la funzione richiesta deve attendere un dato, il thread non si blocca, ma passa subito a servire un'altra richiesta.
- è un **Toolkit per sistemi distribuiti**: Non serve solo per il Web. Può essere usato per gestire protocolli TCP, UDP, DNS, database, o persino per far comunicare diversi parti di software tramite un **Event Bus** interno.
- non è solo un server, ma un insieme di **Verticles** (piccoli componenti isolati) che si scambiano messaggi.

5.6 Sistemi visti dall'interno

Guardare un sistema software 'dal di dentro' significa poter capire meglio come è fatto e quali altre proprietà abbia, oltre a quelle esposte nelle API.

Ovviamente l'interno di un sistema software è costituito da un insieme di *istruzioni* (codice) scritte in qualche linguaggio di programmazione e non è raro che, per molti sistemi, la lettura del codice risulti intricata e *complicata* da capire.

Ma se un sistema software è 'fatto bene' il codice rivela subito una precisa **architettura** che ne rende la comprensione più semplice.

5.6.1 Design patterns

I Design Pattern sono l'equivalente dei "modelli predefiniti" o delle "best practice" consolidate nel mondo dell'ingegneria del software. Invece di reinventare ogni volta la ruota per risolvere un problema ricorrente, si utilizza una **soluzione architetturale** che è già stata testata, analizzata e raffinata da migliaia di sviluppatori.

Un pattern non è codice da copiare e incollare. È piuttosto un **progetto concettuale** che descrive come risolvere un problema in una determinata situazione.

Se l'architettura è la "strategia" (come dividiamo il sistema in grandi blocchi?), i design pattern sono la "tattica" (come facciamo interagire questi blocchi in modo opportuno?).

I pattern vengono solitamente divisi in famiglie principali. Ricordiamo qui le tre categorie classiche, la cui conoscenza diamo per scontata.

Agli anni 90 del secolo scorso risale l'idea di *pattern*, culminata nella pubblicazione nel 1995 dell'ormai famoso testo sui *Design Pattern* della così detta *Gang-of-Four* (**GoF**): *Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides*).

- **Interfaces**

Adapter Facade
Composite Bridge

- **Responsibility**

Singleton Observer
Mediator Proxy
ChainOfResponsibility
Flyweight

- **Construction**

AbstractFactory Buidler
FactoryMethod Prototype Memento

- **Operation**

TemplateMethod State
Command Interpreter

- **Extension**

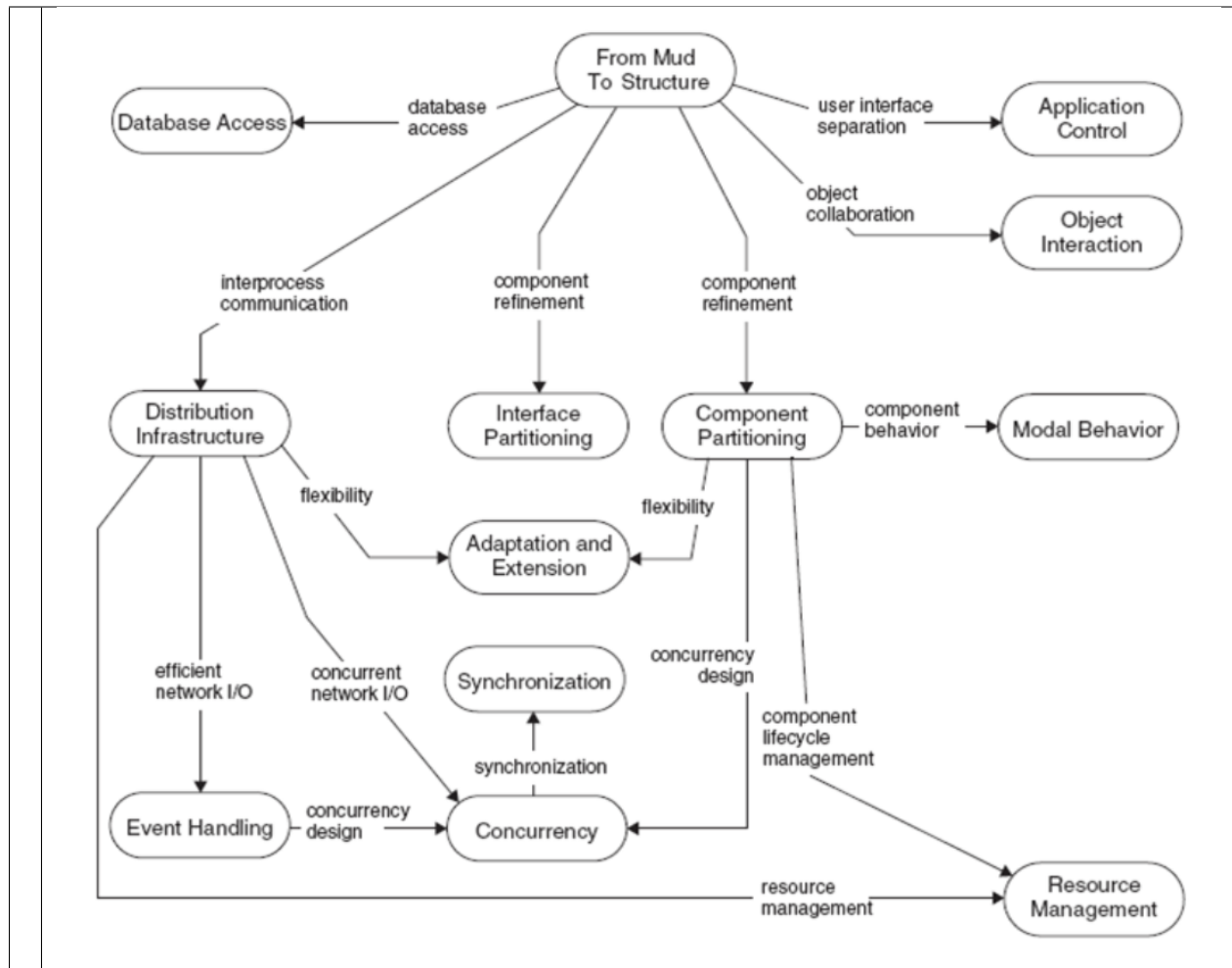
Decorator Iterator Visitor

Un errore comune è cercare di “forzare” i pattern ovunque. Questo porta alla cosiddetta **Over-engineering**: il codice diventa inutilmente complicato. Un bravo architetto software usa i pattern solo quando la semplicità non è più sufficiente.

Dopo la introduzione dei pattern **GoF**, si sono susseguiti molte altre proposte sui design pattern. I riferimenti più noti sono i cinque testi POSA sulle *Pattern oriented Software Architectures* e i convegni PLoP (*Pattern Languages of Programming*).

Pattern POSA

Si riportano qui i pattern POSA (*Pattern oriented Software Architectures*) principali, che intendono introdurre strutture nei sistemi software, per superare il così detto *Big Ball of Mud*.



Anche se non approfondiremo in modo sistematico questi pattern, ritroveremo molte delle problematiche che motivano i vari pattern nei casi di studio di sistemi distribuiti che affronteremo, ponendo particolare attenzione sui Sistemi a Microservizi.

5.7 Architettura di un sistema

Definire l'architettura software è sempre stato complicato perché il termine oscilla tra la struttura tecnica e il processo decisionale. Oggi, la comunità scientifica e professionale converge principalmente su due definizioni che si completano a vicenda.

5.7.1 Definizione Strutturale

Lo Standard ISO/IEC/IEEE 42010 recita:

- L'architettura software è l'organizzazione fondamentale di un sistema, definita dai suoi componenti, dalle relazioni tra di essi e con l'ambiente, e dai principi che ne guidano il progetto e l'evoluzione.

5.7.2 Definizione Pragmatica

Nel mondo dello sviluppo Agile e dei Microservizi, si sposta l'attenzione dal “disegno” alle scelte strategiche:

- L'architettura è l'insieme delle decisioni significative su come è costruito un sistema software; sono le decisioni **difficili da cambiare** in un secondo momento

Un'altra frase celebre nel settore dice:

- L'architettura è ciò che resta del sistema quando si rimuovono tutti i dettagli implementativi.

Ovvero, è lo **scheletro concettuale** che permette di ragionare sul comportamento del software.

5.7.3 Evoluzione delle architetture

L'evoluzione delle architetture software è una storia di *astrazione crescente* e di *decentramento*. Ogni fase è nata per risolvere un limite della precedente, passando da blocchi di codice indivisibili a ecosistemi di funzioni atomiche che “vivono” nel cloud.

Un sintetico quadro di questa evoluzione può essere tracciato come segue:

1. Il Monolite (Anni '60 - '80)

Tutto il software risiede in un unico blocco logico. Interfaccia utente, logica di business e accesso ai dati sono strettamente intrecciati. Il software gira su un unico server (Mainframe)

2. Client-Server (Anni '90)

Con l'arrivo dei PC e delle reti locali, l'architettura si spacca in due (**2-Tier**). Il Client (PC dell'utente) contiene l'interfaccia e la logica, mentre il Server gestisce solo il Database.

3. Architettura N-Tier / Layered (Anni 2000)

L'esplosione del Web induce a dividere il monolite in “strati” (**layer**) logici. Si aggiunge uno strato intermedio (**3-Tier**):

Presentation (Browser) → Logic (Server Applicativo) → Data (Database)

Questo modello ha dominato l'era di Java EE e .NET, portando alla definizione di pattern come il *Model-View-Controller (MVC)*.

4. SOA - Service Oriented Architecture (Metà anni 2000)

Il sistema è un insieme di “servizi” che comunicano tramite un bus centrale (**ESB - Enterprise Service Bus**) I servizi sono grandi e pesanti, spesso legati a protocolli complicati come SOAP e XML.

5. Microservizi e Cloud-Native (2010 - Oggi)

Il sistema è diviso in decine o centinaia di servizi minuscoli e autonomi. Ogni servizio ha il suo DB, viene “im-pacchettato” in **Docker** e orchestrato da **Kubernetes**. La comunicazione è leggera (REST, gRPC, WebSocket).

6. Serverless ed Event-Driven (Il Presente)

L'architettura non è più fatta di server sempre accesi, ma di singole funzioni che si attivano solo in risposta a un evento (**FaaS Function as a Service**) gestito da un *Cloud Provider*.

5.7.4 Big Ball of Mud

Il concetto di **Big Ball of Mud** (*Grande Palla di Fango*) è stato evocato per la prima volta in un saggio di *Brian Foote* e *Joseph Yoder* nel 1997. Tuttavia, come problema sistematico, si è manifestato in momenti diversi dell'evoluzione architettonica, diventando particolarmente critico durante la transizione tra gli anni '80 e '90.

Nel 1997, quando Foote e Yoder pubblicarono il loro saggio, nonostante i nuovi Design Pattern, la maggior parte del software reale non era affatto “pulito”.

La fretta, i requisiti che cambiano continuamente e il turnover degli sviluppatori portavano naturalmente il codice verso la “Palla di Fango”, cioè verso uno stato di entropia massima del software: un sistema che cresce senza un'architettura definita, dove ogni parte dipende da ogni altro parte.

Nella storia del software, il “Big Ball of Mud” si manifesta ogni volta che la velocità di sviluppo prevale sulla disciplina architettonica. Senza uno **sforzo cosciente** per applicare principi organizzativi (come i *Principi SOLID* o le *Clean Architecture*, il sistema, inevitabilmente, si presenta internamente una palla di fango.

L'architettura è costosa, ma la mancanza di architettura costa molto di più.

5.7.5 Principi SOLID

I principi SOLID sono nati nel contesto della *Programmazione Orientata agli Oggetti* (OOP), ma oggi sono considerati principi universali di progettazione software, applicabili a livello di architettura, microservizi e persino organizzazione dei dati.

SRP-Single responsibility

Single responsibility principle: in OOP, una classe dovrebbe avere una, e una sola, ragione per cambiare.

- Concetto Universale: **Coesione**. Ogni componente del sistema deve fare (bene) una cosa sola.
- In Architettura: Un microservizio (o un modulo del Monolite Modulare) deve gestire una sola funzionalità di business. Se il servizio “Ordini” gestisce anche la “Generazione PDF delle fatture” e il “Tracking GPS”, si sta violando l'SRP a livello macroscopico.

OCP-Open/closed

Open-closed principle: in OOP, una classe dovrebbe essere aperta per l'estensione ma chiusa per la modifica.

- Concetto Universale: **Estensibilità**. Usare interfacce o “hook” per permettere l'evoluzione del sistema senza toccare il codice sorgente originale.
- In Architettura: Un sistema deve permettere l'aggiunta di nuove funzionalità senza riscrivere il nucleo esistente. Si pensa ai Plugin o alle Estensioni dei browser.

LSP-Liskov substitution

Liskov's substitution principle: in OOP, gli oggetti in un programma dovrebbero essere sostituibili con istanze dei loro sottotipi senza alterare la correttezza di quel programma.

- Concetto Universale: **Intercambiabilità**. Il rispetto del “contratto” garantisce che il sistema non collassi sostituendo un componente.
- In Architettura: Questo è il principio cardine delle Interfacce API e dei Contratti (come AsyncAPI). Se si sostituisce il servizio xxx1 versione A con la versione B, il client non deve accorgersi della differenza.

ISP-Interface segregation

Interface segregation principle: in OOP, molte interfacce specifiche del client sono migliori di un'interfaccia generica.

- Concetto Universale: **Minimo Privilegio Informativo**. Fornire solo ciò che serve per quel compito specifico (molto rilevante nella sicurezza e nelle performance).
- In Architettura: Evitare i “Modelli Dati Giganti”. Se un client ha bisogno solo dell'indirizzo di un utente, non inviargli l'intero oggetto JSON con password, storico acquisti e preferenze.

DIP-Dependency inversion

Dependency inversion principle: si dovrebbe dipendere dalle astrazioni, non dalle implementazioni.

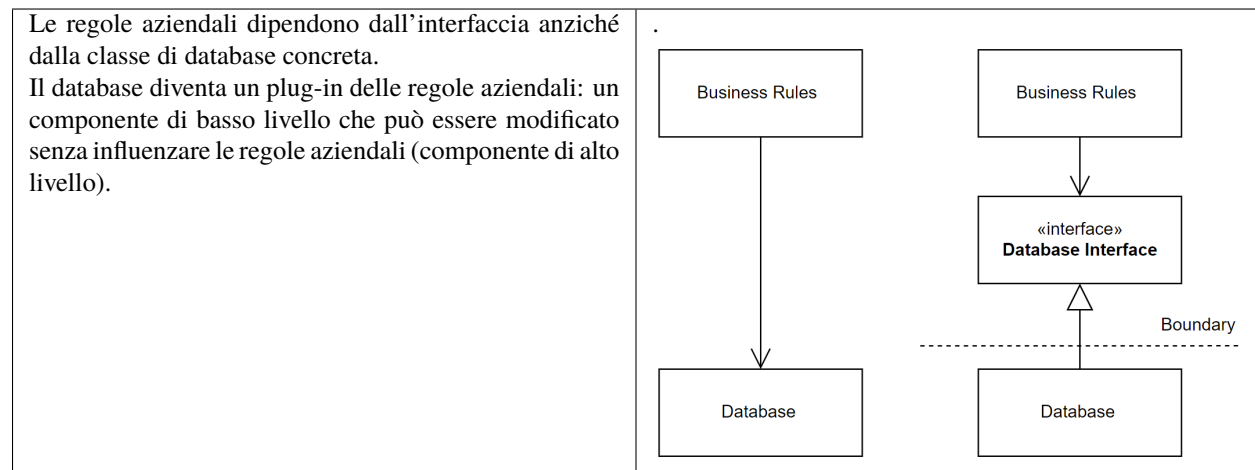
- Concetto Universale: **Disaccoppiamento**. L'astrazione protegge la logica di business dalle tecnologie volatili.
- In Architettura: È il fondamento della *Architettura Port/Adapter*. Il “cuore” logico della applicazione non deve dipendere dal database specifico (MySQL o MongoDB) o dal protocollo (WebSocket o CoAP). Sono il database e il protocollo che devono “adattarsi” alle porte del sistema.

Se il codice sorgente di un componente di alto livello dipende da quello di un componente di basso livello, i cambiamenti nei componenti di basso livello si diffonderanno al componente di alto livello. Pertanto, poniamo un confine tra i due, usando il polimorfismo per invertire il flusso logico.

5.7.6 Clean Architecture

E' un'architettura che segue i principi SOLID. L'idea chiave è quella di utilizzare il *principio di inversione delle dipendenze* per tracciare confini tra componenti di alto livello e componenti di basso livello. Questo crea un'architettura “plug-in” che mantiene il sistema flessibile e manutenibile.

Vediamo un esempio, preso da :ref:Clean Architecture:

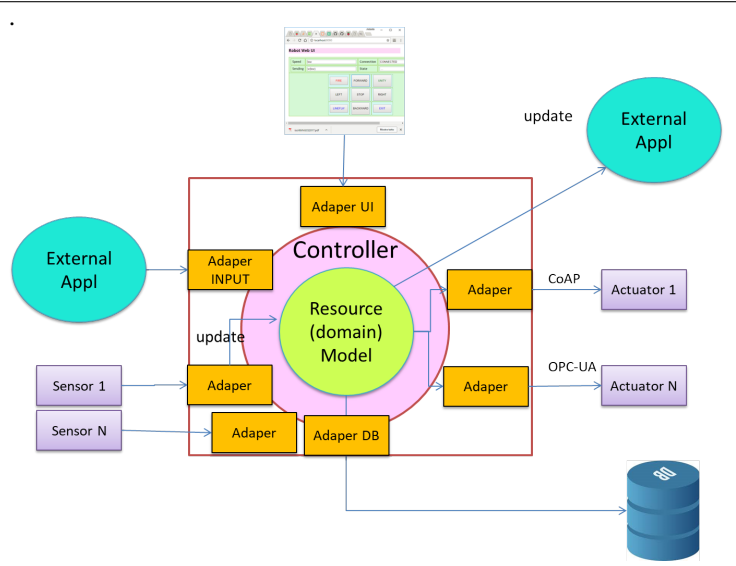


Architettura Port/Adapter

L'architettura **esagonale** (anche nota come *Port-Adapter*) è, tra le tante Architetture di sistemi software oggi in uso, quella cui noi faremo particolare attenzione.

Si identificano quattro layer principali

- **Entities:** oggetti che contengono la logica aziendale critica.
- **Use-cases:** regole aziendali specifiche dell'applicazione
- **Interface adapters:** gateway, presentatori e controller. Ad esempio, questo livello contiene l'architettura MVC della GUI e anche oggetti che trasformano i dati tra il formato del database e i casi d'uso.
- **Frameworks and drivers:** framework web, database, la vista di MVC



5.7.7 Bounded Context

Il concetto di Bounded Context è stato introdotto da Eric Evans nel 2003 e da allora è diventato uno dei pilastri del **Domain Driven Design (DDD)** ed ha avuto grande influenza anche sull'evoluzione dei Sistemi a Microservizi, dove il concetto è stato ripreso e applicato in modo molto concreto.

Il **Bounded Context** rappresenta un *confine esplicito* entro il quale un modello di dominio è definito e applicato in modo coerente. All'interno di questo perimetro, termini, regole di business e significato dei concetti sono univoci e condivisi dal team che lavora su quel contesto.

Lo stesso termine può avere significati diversi in contesti differenti, senza creare ambiguità, proprio grazie alla separazione dei confini.

Uno degli aspetti chiave è la **gestione della complessità**: suddividendo un sistema ampio in più Bounded Context, si evitano modelli di dominio "onnicomprendivi" e difficili da mantenere. Ogni contesto si concentra su una specifica area del dominio, favorendo chiarezza, evoluzione indipendente e maggiore qualità del software.

Il Bounded Context è anche fondamentale per l'**allineamento tra business e tecnologia**. Definendo un *linguaggio ubiquo* valido solo all'interno del contesto, sviluppatori ed esperti di dominio comunicano in modo più efficace, riducendo incomprensioni e traduzioni errate dei requisiti.

- Ad esempio, se un sistema software amplia le proprie funzionalità, si potrebbe avere una parte che si occupa di Vendite e una di Spedizioni. Nel Bounded Context "Vendite", la parola *Prodotto* implica "prezzo e descrizione". Nel Bounded Context "Spedizioni", *Prodotto* implica "peso e dimensioni".

Infine, i Bounded Context hanno un ruolo centrale nell'**architettura dei sistemi distribuiti**, come i microservizi: spesso esiste una forte corrispondenza tra un Bounded Context e un servizio, permettendo indipendenza di deploy, scalabilità mirata e isolamento dei cambiamenti.

Bounded Context e Clean Architecture

Una volta stabilito che siamo dentro un Bounded Context (ad esempio “Vendite”), il codice può essere organizzato secondo la Clean Architecture.

Mentre il Bounded Context (dal Domain-Driven Design) definisce i **confini orizzontali** (le diverse aree di business), la Clean Architecture definisce i **confini verticali** (all’interno di ogni singola area, tra logica e tecnologia).

Un (micro)servizio ben progettato è:

Un’unità di deploy che racchiude un singolo Bounded Context, organizzata internamente secondo la Clean Architecture.

5.8 Costruiamo sistemi software

Il testing

Il noto fisico *Richard Feynmann* (Premio Nobel per la fisica nel 1965 per l’elaborazione dell’elettrodinamica quantistica) ha detto:

Quello che non posso creare, non lo saprò mai capire

Se è vero che *non creare implica non capire*, allora **capire implica creare**.

Il filosofo napoletano *Giambattista Vico* (1668-1744) esprime il motto **Verum ipsum factum** (*la verità è nello stesso fare*) sostenendo che l’uomo può conoscere perfettamente solo ciò che egli stesso ha costruito (come la geometria, la matematica o la storia), mentre non potrà mai capire appieno la natura. Il principio è considerato un esempio di *epistemologia costruttivista*, dove il soggetto conoscente è la causa di ciò che conosce.

Seymour Papert, co-creatore del linguaggio *Logo*, ha coniato il termine **Costruzionismo**:

L’apprendimento avviene in modo particolarmente efficace quando le persone sono impegnate nella costruzione di un prodotto reale.

In accordo a queste visioni, nel seguito affronteremo la progettazione e costruzione di un insieme di sistemi software, scelti in modo da ‘capire meglio’ i temi esposti in precedenza in questo capitolo (e anche per approfondire la comprensione dei sistemi complessi).

5.8.1 Il filo conduttore

Il nostro filo conduttore sarà il gioco *Game of Life di Conway* che nasce come un sistema **semplice da capire e costruire**, ma che presenta **risultati complessi**.

Useremo questo gioco come *caso di studio* per costruire sistemi organizzati secondo architetture diverse:

- *Game of Life di Conway*: un ‘normale’ programma Java ad oggetti.
- Una GUI HTML per Game of Life: evoluzione del sistema precedente in un sistema distribuito in stile MVC.
- Game of Life con protoattori: una prima sperimentazione del concetto di attore, limitata alla costruzione del controllore del gioco nella versione distribuita.
- Game of Life ad attori: una versione in cui tutte le celle sono attori/microservizi autonomi che producono il risultato tramite scambio di messaggi

Altre sperimentazioni

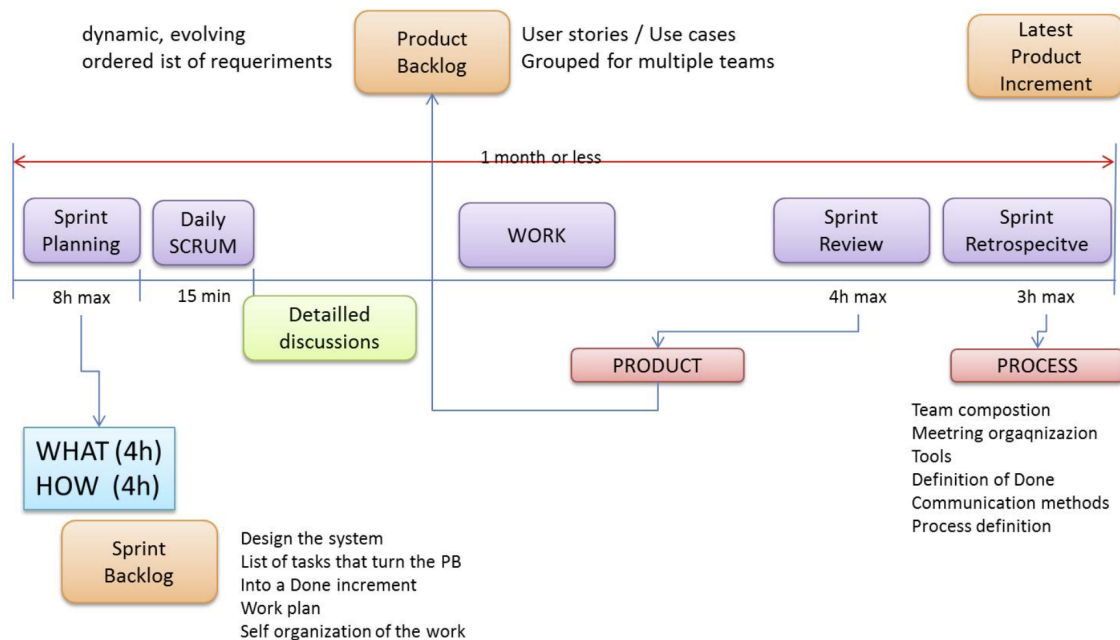
Questo caso di studio sarà affiancato dalla costruzione di altri sistemi:

- Da una funzione a un servizio: come costruire non solo prodotti a sè stanti, ma anche librerie e framework capaci di rendere i sistemi distribuiti indipendenti dai dettagli tecnologici dei vari protocolli di comunicazione
- Costruzione del sistema_S in qak: come definire un modello eseguibile del sistema precedente con Il linguaggio qak, da noi creato per dare una **proiezione semantica** alle librerie introdotte in precedenza
- firefly: ogni lucciola è un attore che sincronizza la propria emissione di luce con tutte le altre lucciole
- IOT su Raspberry
- Robot (virtuale e/o reali) situati

5.9 Processi di costruzione del software

Oggi si ritiene che un **approccio INCREMENTALE** sia indispensabile per aggredire i problemi applicativi e per mettere in luce in primis gli aspetti PIU' IMPORTANTI e PIU' CRITICI, senza perdersi in dettagli che possono distogliere l'attenzione per formare un quadro generale di riferimento.

L'approccio **AGILE** descritto in **SCRUM Guide** è oggi un punto di riferimento.



I vari **SPRINT** dovrebbero effettuare uno **ZOOMING** entro i **MACRO-COMPONENTI** della architettura logica del sistema, innescando un **processo ITERATIVO** di analisi, progetto, sviluppo e testing di quel componente o del **SOT-TOSISTEMA** che lo **SPRINT** vuole costruire.

5.10 Il testing

L'obiettivo della fase di testing è quello di trovare il maggior numero di situazioni che conducono ad errori con il minimo numero di casi di prova.

La fase di testing del software è attività imprescindibile per garantirne la qualità ed è anche indispensabile per garantire all'utente una *user experience* soddisfacente.

In [Software testing IBM](#) leggiamo che:

- Negli anni '90 c'è stata una transizione dai test a un processo più completo denominato *controllo di qualità*, che copre l'intero ciclo di sviluppo del software e riguarda i processi di pianificazione, progettazione, creazione ed esecuzione di casi di test.
- I test del software sono stati tradizionalmente separati dal resto dello sviluppo, ma molti team di sviluppo usano ora una metodologia nota come **test continuo**.

E' importante sottolineare che i test **non vanno pensati ed eseguiti dopo** aver scritto il codice, ma sono modi per chiarire/formalizzare i requisiti di una applicazione e anche per organizzare il lavoro di produzione.

Un *piano di testing* (**TestPlan**) ha di solito lo scopo di stabilire la gerarchia con cui i test vanno eseguiti, i metodi di esecuzione, i criteri di accettazione e prevede diverse tipologie di test, tra cui:

- **Unit test** (Test di unità o di modulo) Ha l'obiettivo di individuare gli errori nel singolo modulo software.
- **Integration Test** (Test di integrazione) Ha l'obiettivo di individuare gli errori nel software quando tutti i moduli che compongono un sottosistema o l'intero sistema vengono fatti lavorare assieme.
- **System Test** (Test di sistema) Ha l'obiettivo di garantire che il prodotto software nel suo complesso soddisfi completamente i requisiti iniziali. E' un collaudo interno.
- **Functional Test** (Test legato ai requisiti) verifica che il sistema implementi correttamente le funzionalità specificate, producendo gli output attesi a fronte di input noti, indipendentemente da come il sistema è implementato internamente. In altre parole, verifica **cosa fa** il sistema, **non come** lo fa.
- **User Acceptance Test** (Test di accettazione) Ha l'obiettivo di valutare la rispondenza dell'applicazione software rispetto ai requisiti espressi inizialmente nel contratto e di ottenere l'accettazione formale del cliente di quanto realizzato. Il test viene effettuato dal cliente nel suo ambiente di test, e da un gruppo di utenti.
- **Alpha test e Beta test** Un *alpha test* è un test preliminare di un'applicazione software anche non ancora completa, eseguito da alcuni potenziali utenti rappresentati da un team.

Un *beta test* ha l'obiettivo di far valutare al cliente, prima della distribuzione ufficiale del sistema, la reale funzionalità, completezza ed operatività dell'applicazione.

- **Regression Test** (test di non regressione) Ha l'obiettivo di verificare a valle di una manutenzione, dopo che un bug è stato individuato e corretto e sono stati eseguiti esattamente gli stessi test che erano stati effettuati quando era stato individuato il problema.
- **Stress test** (test di carico) Ha l'obiettivo di determinare il punto di rottura di un sistema software, oltre il quale si verificano instabilità del sistema, perdita dei dati o interruzione del servizio.

Sul problema del testing si trova moltissimo materiale in rete. Riportiamo qui alcuni riferimenti utili sul 'piano pragmatico':

- [Software testing](#) (wikipedia)
- [Automated-Testing-Plan FAQ](#)

Il *test continuo* è parte dell'approccio [DevOps](#), in cui sviluppo e operazioni collaborano per l'intero ciclo di vita del prodotto. L'obiettivo è quello di accelerare la fornitura del software, bilanciando al tempo stesso i costi, la qualità e i rischi.

L'obiettivo di definire fin dalle fasi di analisi un **TestPlan** con test automatizzabili di un sistema applicativo, introduce di fatto, per il progettista del sistema, un **nuovo requisito**: *rendere l'applicazione osservabile*, o meglio:

rendere verificabili, da parte di una macchina, gli effetti di un sistema software

5.10.1 Il nostro metodo di lavoro

1. Partiamo dall'insieme di **requisiti forniti dal committente** ed espressi in linguaggio naturale
2. Impostiamo lo SPRINT₀ come una **analisi dei requisiti** che mira a esprimerli in modo comprensibile anche alla macchina introducendo:
 - un modello (eseguibile) dell'architettura del sistema desunta dai requisiti
 - un (primo) *TestPlan* di **Functional Tests**
3. Impostiamo una **analisi del problema** che mira a definire:
 - una **architettura logica** del sistema come risultato dall'analisi
 - un piano di lavoro
 - una possibile estensione ai piani di testing funzionali
 - la definizione del *primo SPRINT* della produzione
4. Impostiamo ogni SPRINT come un **sottoprogetto** con sua propria spiegazione e testing.
 - Lo SPRINT $n+1$ parte dai risultati dello sprint n e costruisce un sottosistema (funzionante) che sarà l'input dello SPRINT $n+2$.
5. Facciamo seguire ad ogni SPRINT una fase di **Sprint review** i cui meeting saranno
 - **un nuovo modo di concepire le ore di ricevimento-studenti**
6. Per ogni SPRINT, compiliamo un documento (si veda La documentazione del lavoro) per **rendere esplicite** le conoscenze, le decisioni, i modelli, etc. introdotti nello SPRINT, fornendo indicazioni all'utente finale su come eseguire lo SPRINT.

Il diario di bordo

Ogni progetto da noi impostato, deve avere un *diario di bordo*, organizzato secondo un preciso **template**. Questo diario deve essere compilato **in itinere** con indicazioni **sintetiche ed essenziali**, man mano si svolge il processo di produzione e *non dopo* avere sviluppato il codice.

Il template del diario

- Introduction
 - Requirements

 - Copy here the EXACT text given by the customer.
 - Requirement analysis
 - Problem analysis
 - Test plans
 - Project
 - Testing
 - Deployment
 - Maintenance

(continues on next page)

(continued from previous page)

- About the team (massimo 3 worker)
 - CognomeNomeMatricola del worker1
 - ...

GAME OF LIFE DI CONWAY

Affrontiamo la costruzione di un sistema software ispirato al famoso automa cellulare di **John Horton Conway**: il **Game of Life** (*GofLife*).

Questo gioco viene usato come *Il filo conduttore* del nostro “*approccio costruttivistico*” alla comprensione dei problemi inerenti la realizzazione a la natura dei sistemi software moderni. Infatti il gioco implica un sistema **semplice** da capire e costruire, ma che presenta **risultati complessi**.

Il mondo di *GofLife* è una griglia infinita di celle quadrate. Ogni cella può trovarsi in soli due stati: VIVA (accesa) o MORTA (spenta).

GofLife è un “zero-player game”: la sua evoluzione è determinata interamente dallo stato iniziale della griglia. Ad ogni “tick” di un clock (**generazione** o **Epoch**), si applicano quattro **regole** basate sui vicini di ogni cella:

- Una cella viva con meno di due celle vive adiacenti muore (per isolamento).
- Una cella viva con due o tre celle vive adiacenti sopravvive alla generazione successiva.
- Una cella viva con più di tre celle vive adiacenti muore (per sovrappopolazione).
- Una cella morta con esattamente tre celle vive adiacenti diventa una cella viva (per riproduzione).

Nel caso di griglia infinita, ogni cella ha esattamente 8 vicini (orizzontali, verticali e diagonali). Ovviamente, in una implementazione software, la griglia sarà finita e le celle potranno avere **3,5, o 8** vicini.

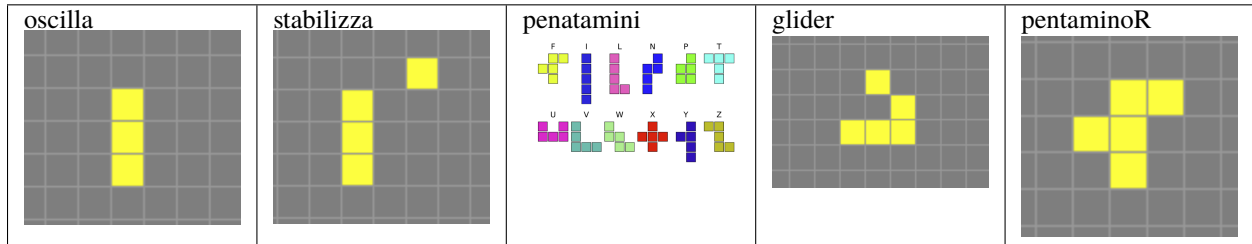
Nonostante le regole parlino solo di “celle e vicini”, **osservando una vista del sistema** emergono (si veda emergenza) strutture che sembrano avere una propria identità (si ricordi quanto detto per “Vedere un Glider” in *Teoria Algoritmica dell’Informazione*), come:

- **Still Lifes**: Forme statiche che non cambiano (come rocce).
- **Oscillatori**: Forme che mutano ciclicamente (come un battito cardiaco).
- **Spaceships** (Astronavi): Aggregati che “espellono” delle strutture che viaggiano all’infinito lontano dal centro della griglia. Il più famoso è il **Glider** (Aliante).

E’ importante sottolineare che le strutture *Still Lifes* **sembrano** rimanere immutabili. Infatti esse sono **continuamente rigenerate** dal ‘motore’ del gioco, sempre uguali a se stesse; si veda: Evoluzione di GofLife.

6.1 Il gioco in JavaScript

Collegandosi al sito <https://playgameoflife.com/> si può sperimentare il gioco. Riportiamo qui alcune configurazioni iniziali interessanti:



I 12 **pentamini** esistenti si stabilizzano nei tre modi citati: Still Lifes, Oscillatori e Astronavi (Glider).

Il **Pentamino-R** è importante perché ha dimostrato che non era possibile prevedere a occhio nudo se una forma sarebbe cresciuta all'infinito o meno. È il riflesso visivo del Problema della *Fermata di Turing*: non si può sapere se un programma si fermerà (si stabilizzerà) senza farlo girare. Viene definito un *Matusalemme*, perché esplode in un caos ribollente che dura per ben 1103 generazioni prima di stabilizzarsi.

6.2 Il gioco produce complessità

Nonostante la semplicità delle regole di base, la previsione del comportamento a lungo termine del Gioco della Vita è un problema che ha affascinato matematici e informatici per decenni e, nonostante numerosi studi, non esiste ancora una soluzione definitiva e generale, per alcuni motivi:

- **Complessità esponenziale:** Il numero di possibili configurazioni di una griglia di celle aumenta in modo esponenziale con la dimensione della griglia stessa. Questo rende impraticabile un'analisi esaustiva di tutte le possibili evoluzioni.
- **Comportamento caotico:** Piccole variazioni nella configurazione iniziale possono portare a risultati completamente diversi nel lungo periodo, rendendo difficile stabilire delle leggi generali che governino l'evoluzione del sistema.
- **Emergenza:** Dal semplice insieme di regole del gioco emergono comportamenti complessi e imprevedibili, come la formazione di strutture stabili, oscillanti o in continua evoluzione.

Anche con una griglia finita, prevedere il comportamento a lungo termine di una configurazione iniziale rimane un *problema complesso*. Il numero di possibili configurazioni, anche se finito, può essere enorme, rendendo impraticabile un'analisi esaustiva.

Inoltre, a differenza di molti altri sistemi fisici, il Gioco della Vita **non è sempre reversibile**. Ciò significa che, data una configurazione finale, non è sempre possibile risalire univocamente alla configurazione iniziale che l'ha generata. Molte configurazioni finali possono avere più configurazioni iniziali che portano allo stesso risultato.

Dunque, determinare la configurazione iniziale che porta a una data configurazione finale nel Gioco della Vita è un *problema complesso* e, in generale, non esiste una soluzione algoritmica efficiente.

Il gioco è un esempio di sviluppo e **auto-organizzazione**. È interessante per scienziati, matematici e economisti osservare il modo in cui schemi complessi possono emergere dall'implementazione di regole assai semplici.

6.2.1 Il gioco è Turing-completo

È stato dimostrato che nel *Game of Life* si possono costruire circuiti logici (porte AND, OR, NOT) ed è quindi *Computazionalmente completo*. Ciò significa che, con una griglia abbastanza grande e la giusta configurazione iniziale, si potrebbe costruire un computer capace di eseguire qualsiasi algoritmo che un PC attuale può eseguire.

6.3 Il problema della vista del gioco

Se visualizziamo l'evoluzione dello stato delle celle con una sequenza di messaggi su una console, come ad esempio:

```
cell x=0y=0:false
cell x=0y=1:true
cell x=0y=2:false
...
```

possiamo capire che *LifeController* sta girando, ma non avremo certo alcuna immediata percezione delle configurazioni 'interessanti'.

Il "senso" delle configurazioni non risiede solo nei dati, ma anche nella struttura dello spazio in cui questi dati vengono proiettati. Se la proiezione non rispetta la geometria delle regole del gioco, l'emergenza (la nascita delle forme) svanisce per l'osservatore.

Per fare in modo che lo spazio della rappresentazione sia "sintonizzato" sulla metrica delle leggi del gioco occorre una matrice 2D.

La matrice non è solo un modo "comodo" per vedere il gioco; è la chiave di **decrittazione** che permette alla logica di Conway di manifestarsi come "forma".

Note: Il filosofo **Daniel Dennet** utilizza il Game of Life come metafora centrale nel suo saggio del 1991, *Real Patterns*, per illustrare come entità complesse possano essere considerate "reali" senza dover essere riducibili a oggetti fisici fondamentali.

La descrizione dello stato delle celle come sequenza di stringhe è esaustiva ma priva di sintesi; non permette di "vedere" il comportamento globale del sistema in modo efficiente.

Per Dennett, il **glider** che si muove su una rappresentazione a matrice è un "pattern reale" perché riconoscerlo permette una **compressione predittiva**: fornisce un vantaggio informativo enorme rispetto alla descrizione atomica delle celle.

I glider dunque "esistono"? La risposta di Dennet è che sono reali tanto quanto i nostri desideri o le nostre credenze. Non sono entità fisiche separate, ma sono configurazioni oggettive della materia che diventano visibili (e reali) solo quando adottiamo un certo livello di astrazione necessario per navigare la complessità del mondo.

Dennet sostiene che noi identifichiamo modelli adottando diverse posizioni nei confronti di un sistema:

- **Posizione fisica:** previsione basata su leggi fisiche e microdettagli.
- **Posizione di progettazione:** previsione basata sull'organizzazione funzionale (ad esempio, "questa parte serve per il raffreddamento").
- **Posizione intenzionale:** prevedere il comportamento attribuendo credenze, desideri e razionalità.

Il collegamento tra i "Real Patterns" di Dennett e la **Teoria algoritmica dell'informazione** è diventato un pilastro della filosofia della scienza contemporanea, con importanti sviluppi nel 2025 e 2026 che formalizzano questa intuizione.

Per un approfondimento, si veda la *Teoria Algoritmica dell'Informazione* come l'apparato matematico che formalizza le idee di Dennet.

6.4 Il gioco come caso di studio

Nei capitoli seguenti faremo riferimento al gioco *Game of Life* (**GofLife**) come caso di studio di sistemi software, dapprima concentrati e poi distribuiti. L'elenco delle varie versioni è riportato in *Il filo conduttore*:

Iniziamo costruendo il gioco assumendo come tecnologia di riferimento il *paradigma ad oggetti* e il linguaggio Java. L'obiettivo è anche mettere a punto gli strumenti (Eclipse, Gradle, Docker, etc.) che useremo per la costruzione del software. A tal fine si veda: Primi passi operativi.

6.4.1 Analisi del problema GofLife

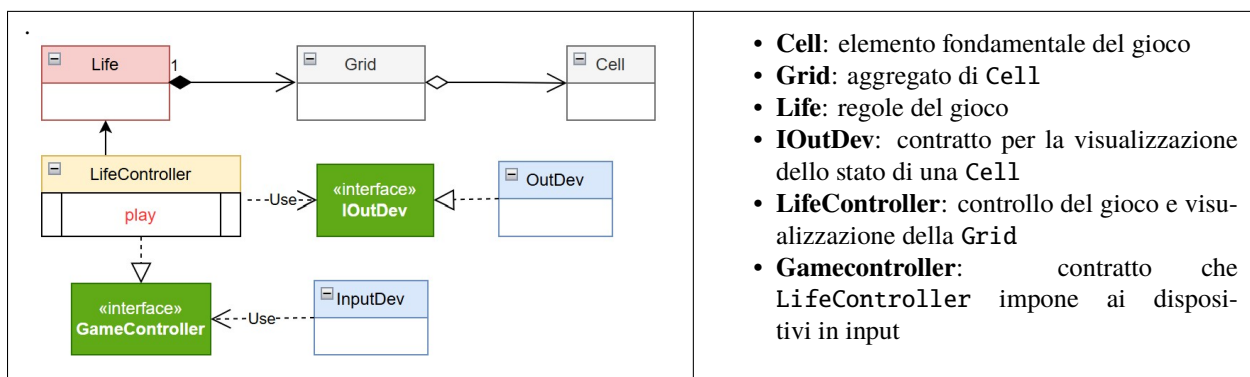
Tratandosi di un gioco zero-player con controllo esterno, il sistema

- deve essere **proattivo**, in quanto deve realizzare il gioco come un comportamento autonomo
- deve essere **anche reattivo**, in quanto deve percepire e gestire i comandi di configurazione e di start/stop che l'utente invierà mediante un dispositivo di input
- deve fornire una rappresentazione tenendo conto di quanto detto in *Il problema della vista del gioco*

6.4.2 Progetto ConwayLife26

In accordo ai principi di Separazione delle Responsabilità e *SRP-Single responsibility* proponiamo una architettura software basata sui seguenti componenti software:

1. Modelliamo il dominio con le classi Java *Cell*, *Grid*, *Life*
2. Definiamo un componente attivo *LifeController* che ha la responsabilità di far evolvere il gioco
3. Definiamo *La interfaccia IOutDev* per la visualizzazione dello stato di una *Cell* e/o di tutta la *Grid*
4. Definiamo *La interfaccia GameController* per la interazione dei dispositivi di input con *LifeController*
5. Definiamo i componenti di output che implementano *IOutDev* e quelli di input che si relazionano con *LifeController* mediante l'interfaccia *GameController*.



6.4.3 LifeController

Osserviamo che `LifeController` è un componente

- **proattivo**, in quanto fa evolvere il “zero-player game” attraverso il thread **play**
- **reattivo**, in quanto dovrebbe essere capace di:
 - percepire comandi per definire la configurazione iniziale della Grid
 - percepire comandi di `start/stop/clear` del gioco emessi da un utente (umano) mediante un dispositivo di input. A questo fine, `LifeController` implementa *La interfaccia `GameController`*

Osserviamo anche che i dispositivi di output e di input possono essere di tipi diversi. In particolare, esamineremo le seguenti possibilità:

1. Dispositivo di output basato su console (`System.out`); lasciato come esercizio al lettore
2. Dispositivo di output (e input) basato su GUI swing: si veda *Una GUI con Swing*
3. Dispositivo di output (e input) basato su GUI HTML/JS:
 - con rappresentazione granulare della Grid; si veda *Una griglia granulare*
 - con rappresentazione sintetica della Grid` mediante canvas; si veda *Una griglia globale*

Il punto importante consiste nel fare in modo che `LifeController` **non dipenda** da una specifica implementazione di `IOutDev`.

```
public class LifeController implements GameController {
    public LifeController( Life game, IOutDev outdev ){
        ...
    }

    protected void play() {
        new Thread() {
            public void run() {
                //Evoluzione del gioco in epoche successive
                while( running ) {
                    ...
                }
            }
        }.start();
    } //play
}
```

6.4.4 Il processo di evoluzione del gioco

Il thread **play** fa evolvere il gioco dalla configurazione iniziale in epoche successive. Nella nostra implementazione attuale, il thread rimane **running** fino a quando:

- la configurazione corrente diventa stabile
- non ci sono più celle vive
- l'utente invia un comando `stop` per fermare l'evoluzione del sistema

Tuttavia si potrebbe pensare di **non fermare** `play` nei primi due casi, in quanto:

- l'utente che controlla il gioco potrebbe introdurre nuove celle ‘vive’ tra una epoca e l'altra

- il thread *play* stesso potrebbe generare nuove celle ‘vive’ in modo casuale, simulando un meccanismo che nella teoria dei campi della fisica moderna è collegato all’idea di ‘vuoto fisico’ della teoria dei campi come entità dinamica, che, come asserito dai mistici orientali, è la ‘realtà prima’ che dà origine a tutte le forme del mondo fenomenico.

6.4.5 La interfaccia IOutDev

```
package main.java.conway.domain;

public interface IOutDev {
    void display(String msg);           //For HMI
    void displayCell(Cell cell, Grid grid);
    void displayGrid(Grid grid);
    void close();
}
```

- Il metodo `display(String msg)` consente di visualizzare messaggi di testo (per una migliore **HMI** - *Human-MachineInterface*)
- Il metodo `displayCell(Cell cell, Grid grid)` consente di visualizzare lo stato di una singola cella (all’interno della griglia)
- Il metodo `displayGrid(Grid grid)` consente di visualizzare lo stato di tutta la griglia

6.4.6 La interfaccia GameController

```
package main.java.conway.domain;

public interface GameController {
    void onStart();
    void onStop();
    void onClear();
    void switchCellState(int x, int y);
    void setOutDev(IOutDev outdev); //injection
}
```

- I metodi `onStart`, `onStop` e `onClear` consentono di avviare, fermare e azzerare il gioco.
- Il metodo `switchCellState` consente di commutare lo stato di una cella (viva/morta). Usato in particolare nella fase di inizializzazione.
- Il metodo `setOutDev` consente di ‘iniettare’ una specifica implementazione di `IOutDev` nel `LifeController`.

6.4.7 MainConwayLifeJava.java

I diversi approcci per la realizzazione di dispositivi di output e input vengono sperimentati con il programma **MainConwayLifeJava.java** nel package `main.java.conway.domain`.

```
public void configureTheSystemWithMockOutdev() { ... }
public void configureTheSystemWithSwing() { ... }
public void configureTheSystemWithHtmlWs(boolean pageexternal){ ... }
```

(continues on next page)

(continued from previous page)

```

public static void main(String[] args) {
MainConwayLifeJava app = new MainConwayLifeJava();
    //app.configureTheSystemWitMockOutdev();
    app.configureTheSystemWithSwing();
    //app.configureTheSystemWithHtmlWs(false); //apre la pagina automaticamente
    //app.configureTheSystemWithHtmlWs(true); //occorre localhost:8080
}

```

Il programma propone diverse procedure di configurazione del sistema, tra cui :

- *Configurazione di GofLife con Swing*
- Configurazione di GofLife con HTML (più avanti)

6.4.8 Una GUI con Swing

Un primo modo per superare il limite dell'uso della console come dispositivo di output, è introdurre una visualizzazione di tipo grafico, realizzando una GUI in Java con Swing. Un dispositivo di questo tipo è definito nella classe `main/java/conway/io/ConwayLifeGridClaude.java` il cui codice è stato proposto da **Claude AI**.

Note: Il prompt dato a Claude (**requisiti**): Vorrei una classe Java che realizza la rappresentazione grafica di una griglia quadrata di 20x20 celle usando Swiing. La griglia è relativa al gioco ConwayLife e quindi le celle possono assumere solo due colori (white - cella morta e red - cella viva). La classe deve fornire metodi invocabili dall'esterno per visualizzare tutta la griglia e per commutare il colore di una cella (da white a red e viceversa). Nella rappresentazione grafica devono comparire anche due pulsanti START e STOP, facendo click sui quali la classe invia un comando a un componente esterno (un controllore del gioco)

6.4.9 Configurazione di GofLife con Swing

Rispetto al codice proposto da Claude, abbiamo aggiunto i metodi per implementare l'interfaccia `IOutDev` e la 'iniezione' dell'istanza di `LifeController`.

```

package main.java.conway.domain;
public class MainConwayLifeJava {
    ...
public void configureTheSystemWithSwing() {
    Life life          = new Life( 20,20 );
    IOutDev swinggui   = new ConwayLifeGridClaude( ); //dispositivo di output (e anche
↳di input)
    GameController cc  = new LifeController(life, swinggui) ; //un GameController che
↳deve usare un IOutDev
    ((ConwayLifeGridClaude) swinggui).setController(cc);      //iniezione del
↳controller nella GUI
    //Il sistema termina quando si chiude la swinggui
}

```

Problemi di deployment

Questa implementazione funziona bene in locale, ma non è adatta a essere eseguito in un container Docker.

Infatti, Java Swing tenta di disegnare la GUI, interroga il sistema operativo per trovare un Display Server (come *X11* su Linux o *Window Manager* su Windows). Non trovandolo, Java solleverà l'eccezione **`java.awt.HeadlessException`**.

Mettere una GUI Swing in Docker va contro la filosofia “Cloud Native”, se non altro perchè l’immagine Docker diventa enorme, dovendo includere tutte le librerie grafiche.

6.4.10 Da un programma a un servizio

Meglio quindi realizzare la GUI del gioco come una pagina Web (HTML/JS) come faremo in Una GUI HTML per Game of Life.

Prima però è opportuno capire che la introduzione di una pagina HTML cambia la natura del sistema, in quanto implica la introduzione di supporti capaci di realizzare interazioni via rete tra componenti software.

Il gioco diventa molto simile a un ‘servizio’ e può quindi essere utile approfondire le implicazioni di questa trasformazione nel prossimo capitolo.