I Princìpi dell'Ingegneria del Software

a cura di **Angelo Furfaro**
da "Ingegneria del Software, Fondamenti e Princìpi"
Ghezzi, Jazayeri, Mandrioli

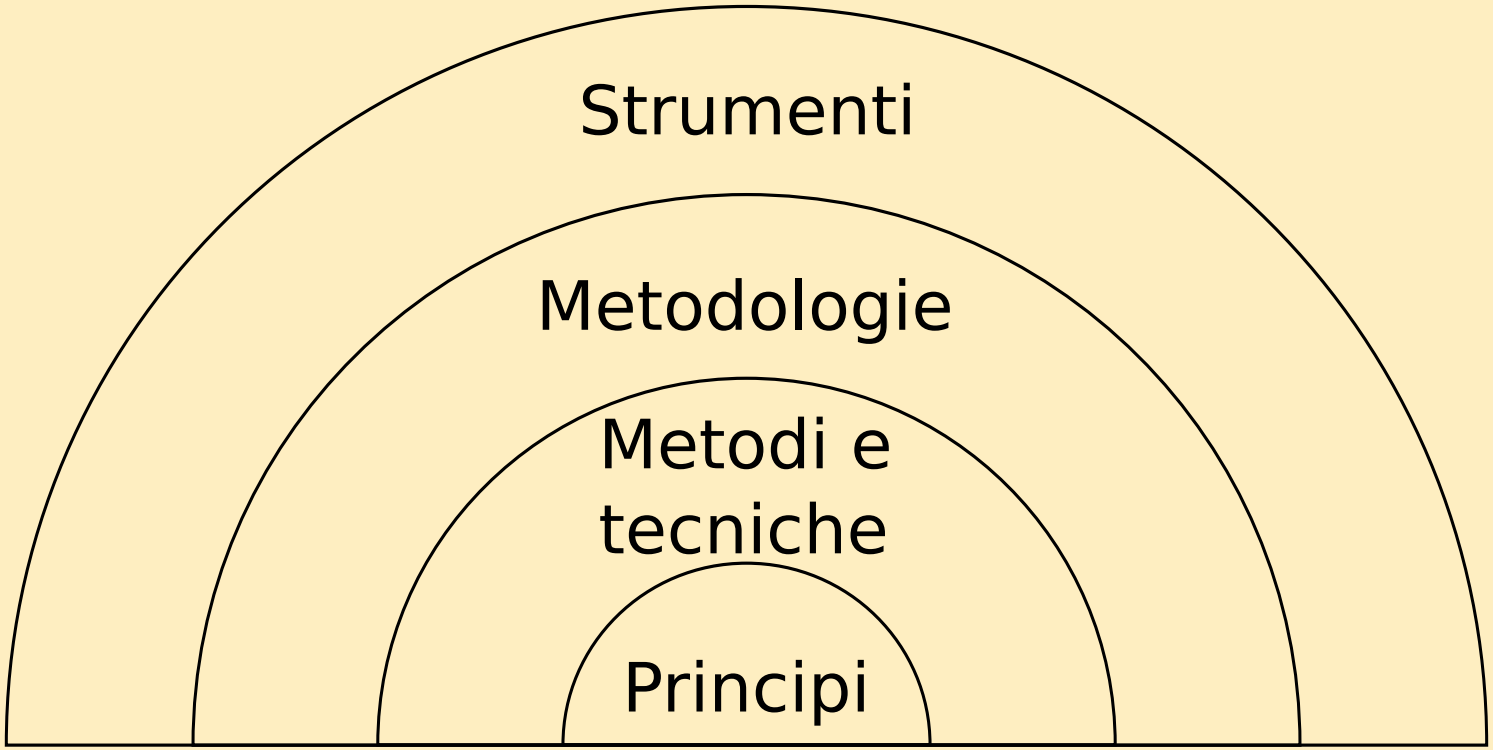
Dipartimento di
Ingegneria Informatica, Elettronica, Modellistica e Sistemistica
Università della Calabria, 87036 Rende(CS) - Italy
Email: a.furfaro@unical.it
Web: http://angelo.furfaro.dimes.unical.it

Introduzione

- I princìpi illustrati di seguito costituiscono la base dei metodi, delle tecniche, delle metodologie e degli strumenti dell'ingegneria del software
- Tali princìpi possono essere adoperati in tutte le fasi dello sviluppo del software
- La modularità è il principio fondamentale nella progettazione del software
- I princìpi si applicano sia al processo che al prodotto
- I princìpi sono messi in pratica attraverso i metodi e le tecniche
  - metodi e tecniche sono spesso *confezionati* assieme per formare una *metodologia*
  - l'uso di una metodologia può essere imposto dagli *strumenti* (*tools*) adottati

Metodi e Tecniche

- **Metodi**: linee guida generali che governano l'esecuzione di un attività attraverso un approccio rigoroso, sistematico e disciplinato
- **Tecniche**: hanno carattere più meccanico ed un'applicabilità più limitata



I princìpi chiave

- Rigore e formalità
- Separazione degli interessi
- Modularità
- Astrazione
- Anticipazione del cambiamento
- Generalità
- Incrementalità

Rigore e Formalità

- Lo sviluppo del software è un'attività creativa e pertanto si tende ad essere poco precisi e/o accurati
- Un adeguato livello di rigore è necessario al fine di ottenere prodotti affidabili, controllarne i costi ed aumentare la fiducia nel loro corretto funzionamento
- Il rigore non limita la creatività
- La formalità è il più alto livello di rigore (richiede che il processo di sviluppo sia guidato e valutato con leggi matematiche)
- Occorre determinare il giusto livello di rigore
- La formalità offre garanzie e vantaggi maggiori ma in alcuni casi è troppo complessa e non applicabile in pratica:
  - Parti critiche, come lo scheduler di un sistema operativo real-time, necessitano di una descrizione *formale* del loro funzionamento ed una dimostrazione *formale* della loro correttezza
  - La derivazione sistematica dei dati in ingresso per le procedure di test è un approccio *rigoroso* ma non formale
  - La documentazione *rigorosa* dei passi di sviluppo aiuta la gestione del progetto e la verifica della tempistica del processo

Rigore e Formalità

- La fase di programmazione adotta un approccio formale
- I programmi sono oggetti formali, scritti in linguaggi di programmazione la cui sintassi e semantica sono definite formalmente
- I programmi possono essere manipolati automaticamente dai compilatori
- La formalità nella programmazione favorisce la verificabilità e l'affidabilità dei prodotti software
- Il rigore e la formalità possono essere applicati anche nelle fasi di progettazione, specifica e verifica

Separazione degli interessi

- Per dominare la complessità è necessario separare i vari aspetti del problema concentrandosi su ciascuno di essi in maniera separata
- Occorre innanzitutto isolare gli aspetti scarsamente correlati tra di loro
- Esistono vari modi di separazione:
  - **Temporale**: la separazione delle attività in diversi periodi temporali consente una pianificazione precisa ed evita gli inconvenienti legati al continuo cambiamento
  - **Qualitativa**: ad esempio correttezza ed efficienza di un programma possono essere considerate separatamente curando la seconda qualità dopo aver assicurato la prima

Separazione degli interessi

- Un altro tipo di separazione riguarda la capacità di affrontare separatamente diverse parti del sistema. Da questo concetto deriva il principio di modularità.
- Un potenziale inconveniente legato alla separazione degli interessi è la minore possibilità di effettuare ottimizzazioni globali
- Un'importante applicazione di questo principio è la separazione tra aspetti relativi al dominio del problema da quelli relativi al dominio dell'implementazione
- La separazione degli interessi può fornire una base per la separazione delle responsabilità nel trattare i vari aspetti

Modularità

- Un sistema complesso può essere suddiviso in parti più semplici detti *moduli*
- Un sistema composto da moduli è detto *modulare*
- Consente di applicare la separazione degli interessi in due fasi:
  1. Si possono trattare i dettagli di ciascun modulo separatamente ignorando i dettagli degli altri
  2. Si possono esaminare le caratteristiche complessive di tutti i moduli e le loro relazioni per integrarli in un sistema coerente
- Se ci si concentra prima sui moduli e poi sulla loro composizione si ha un approccio **bottom-up**
- Se si procede prima alla scomposizione in moduli e successivamente sulla progettazione di ciascuno di essi si ha un approccio **top-down**

Modularità

La modularità fornisce quattro tipi di benefici fondamentali:

- La capacità di scomporre un sistema complesso in parti più semplici (*divide et impera*)
- La capacità di comporre un sistema complesso a partire dai moduli esistenti
- La capacità di comprendere un sistema in funzione delle sue parti
- La capacità di modificare un sistema modificando solo un piccolo insieme delle sue parti

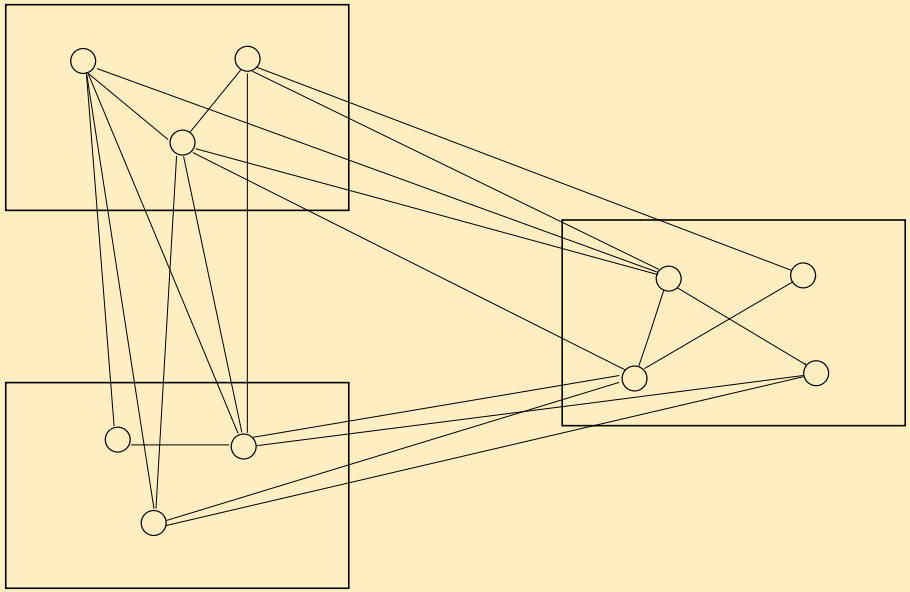
Modularità

Alta coesione e basso accoppiamento

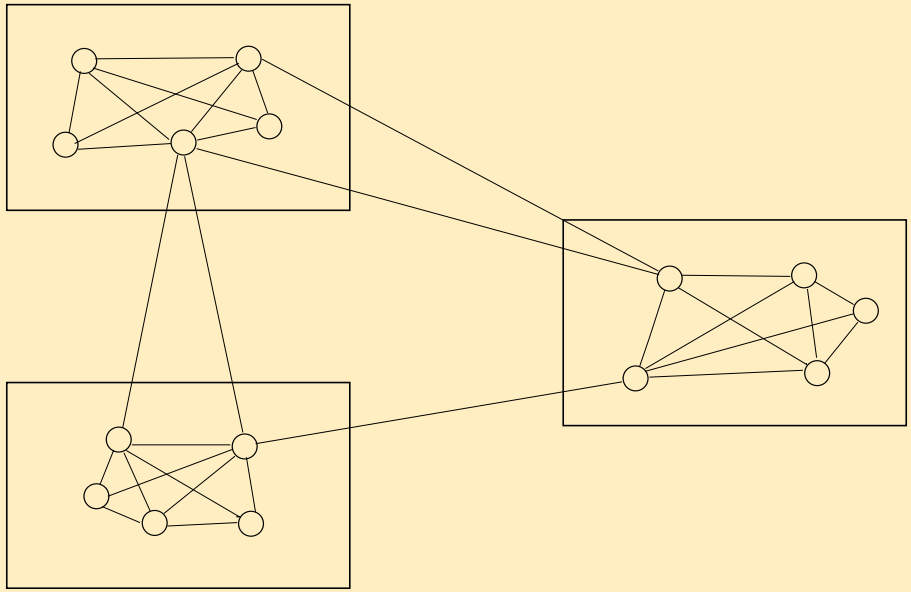
- Per ottenere i benefici della modularità è necessario che i moduli abbiano un'alta coesione interna ed un basso accoppiamento tra di essi
- Un modulo è altamente coeso se tutti i suoi elementi sono strettamente connessi
- L'accoppiamento caratterizza le relazioni tra i moduli.
- Un basso accoppiamento consente di analizzare, capire, modificare, testare o riusare ciascun modulo separatamente
- Strutture modulari con alta coesione e basso accoppiamento consentono di vedere i moduli come delle *black box* quando si descrive la struttura complessiva e vedere ciascuno nei suoi dettagli quando è necessario analizzarne la struttura interna

Alta coesione e basso accoppiamento

Forte accoppiamento



Alta coesione e basso accoppiamento



Astrazione

- L'astrazione consente di identificare gli aspetti fondamentali di un fenomeno e di ignorare i dettagli
- È un caso particolare di separazione degli interessi
- Il tipo di astrazione dipende dallo scopo

Esempio: orologio digitale

- **Scopo: settaggio del tempo**
  La descrizione del funzionamento dell'interfaccia utente di un orologio digitale, ovvero (gli effetti della pressione dei tasti) *astrae* dai dettagli interni dell'orologio
- **Scopo: riparazione**
  L'orologio può essere *astratto* come una scatola che può essere aperta per sostituire la batteria

Astrazione

Esempio: circuito elettrico

- Può essere rappresentato in termini di componenti (resistenze, capacità, etc.) ciascuno descritto da equazioni definisce un'astrazione ideale del dispositivo
- Le equazioni ignorano i dettagli che producono effetti trascurabili (ad es. le connessioni tra i componenti sono considerate ideali)

L'astrazione genera modelli

- Quando, ad esempio, si analizzano i requisiti si ottiene un modello della potenziale applicazione
- Il modello può essere una descrizione formale o semiformale
- Per mezzo del modello ottenuto è possibile ragionare sul sistema
- Se un modello può essere *simulato* può fornire informazioni sul (comportamento del) sistema prima che esso venga realizzato

Astrazione

Astrazione nei linguaggi di programmazione

- I linguaggi di programmazione sono astrazioni costruite sull'hardware
- I costrutti dei linguaggi consentono di scrivere programmi ignorando i dettagli che implementano a basso livello la loro semantica
- Ciò consente al progettista di concentrarsi sulla soluzione del problema piuttosto che sull'istruire il calcolatore ad eseguire una sua risoluzione

Astrazione nei processi

- Si consideri l'esempio della stima dei costi per la realizzazione di una nuova applicazione
- Un modo possibile consiste nell'individuazione di alcuni fattori chiave quali ad esempio: il numero di progettisti necessari e la dimensione attesa del prodotto
- La stima può essere ottenuta estrapolandola dai costi di sistemi *simili* nei fattori chiave individuati *astraendo* in tal modo dalle differenze nei dettagli

Anticipazione del Cambiamento

- La capacità di supportare l'evoluzione del software richiede l'abilità di anticipare i potenziali cambiamenti futuri
- Costituisce la base per l'evolvibilità del software
- La capacità di evolvere deve essere pianificata con estrema cura
- I progettisti devono tentare di identificare i possibili cambiamenti e predisporre il progetto per favorire la loro attuazione in modo agevole
- La riusabilità è un'altra qualità influenzata dall'anticipazione del cambiamento
- L'anticipazione del cambiamento richiede adeguati strumenti per la gestione delle varie versioni e revisioni del software
- L'anticipazione del cambiamento è fondamentale anche nei processi (ad es. la previsione degli effetti del cambio di personale)

Generalità

- Ogni volta che si cerca di risolvere un problema, si cerca di scoprire quale è il problema più generale che si nasconde dietro quello specifico
- Alcune volte il problema più generale è più complesso, ma spesso è più semplice di quello originale
- La soluzione del problema generale è potenzialmente più riusabile
- La soluzione generalizzata può essere più costosa (velocità di esecuzione, memoria utilizzata, tempo di sviluppo)
- Occorre bilanciare la generalità rispetto ai costi ed all'efficienza della soluzione

Incrementalità

- Si parla di incrementalità quando un processo procede per passi (incrementi) che permettono di raggiungere l'obiettivo attraverso approssimazioni successive
- Nel caso del software si intende che l'applicazione desiderata si ottiene come risultato di un processo evolutivo

Esempi

- Si consegna in anticipo un sottoinsieme delle funzionalità di un sistema per ottenere al più presto feedback dagli utenti; nuove caratteristiche sono aggiunte in modo incrementale
- Si cura per prima il supporto alle funzionalità, quindi ci si dedica alle prestazioni
- Si consegna inizialmente un prototipo del prodotto che viene quindi raffinato incrementalmente in prototipi successivi fino a trasformarlo nel prodotto finale

Caso di studio: Compilatore

- Un compilatore è un prodotto critico: dalla sua correttezza dipende la correttezza della applicazioni generate per mezzo di esso
- Lo sviluppo rigoroso è di importanza cruciale
- La costruzione di un compilatore è un area in cui sono stati sviluppati dei metodi sistematici e formali di progettazione, ad esempio:
  - BNF per la descrizione formale della sintassi del linguaggio
  - applicazione della teoria degli automi e dei linguaggi formali

Caso di studio: Compilatore

Separazione degli interessi

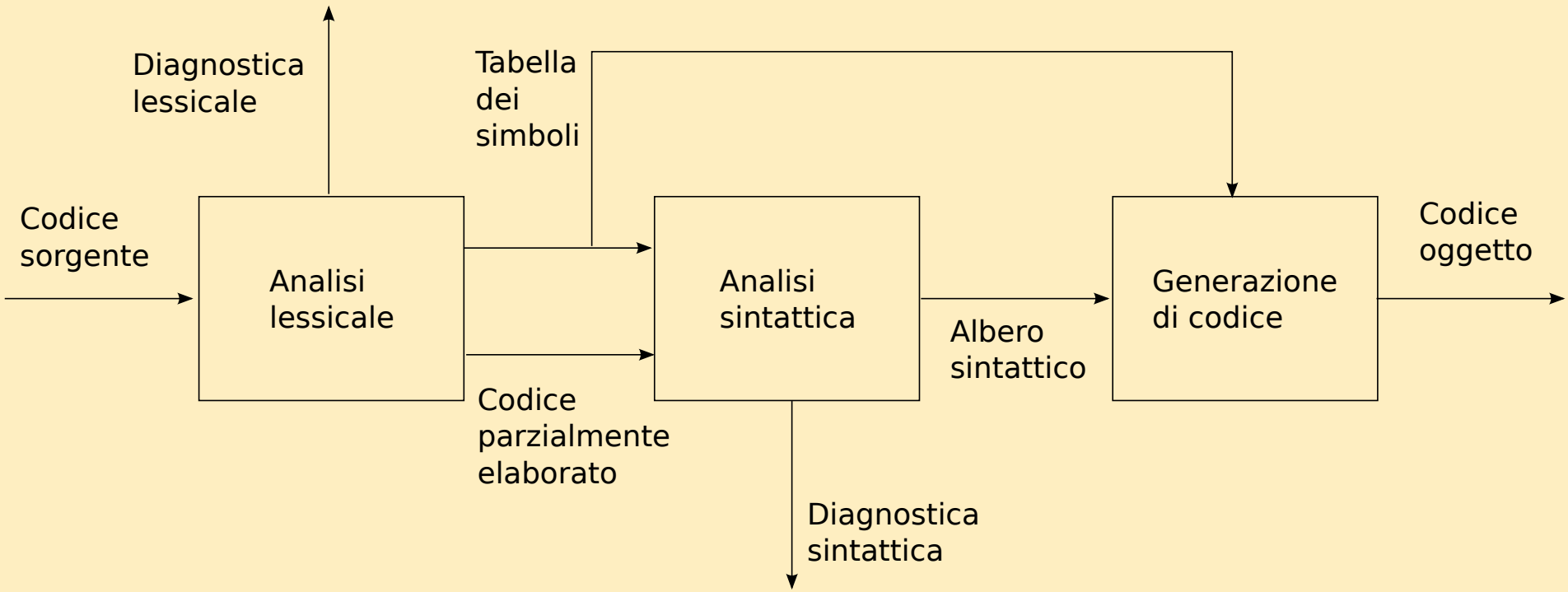
- L'efficienza del compilatore e l'amichevolezza della sua interfaccia utente sono due aspetti importanti
- L'efficienza riguarda la velocità nell'analisi del codice sorgente e della traduzione, l'utilizzo limitato della memoria e il tempo di esecuzione del codice generato
- L'amichevolezza dell'interfaccia riguarda ad esempio la precisione e completezza dei messaggi diagnostici e la facilità di interazione tramite GUI
- Questi due aspetti dovrebbero essere analizzati e gestiti separatamente

Modularità

- Il processo di compilazione viene decomposto in fasi:
  1. Analisi lessicale
  2. Analisi sintattica (parsing)
  3. Generazione di codice
- Le fasi possono essere associate a moduli distinti

Caso di studio: Compilatore

Rappresentazione della struttura modulare

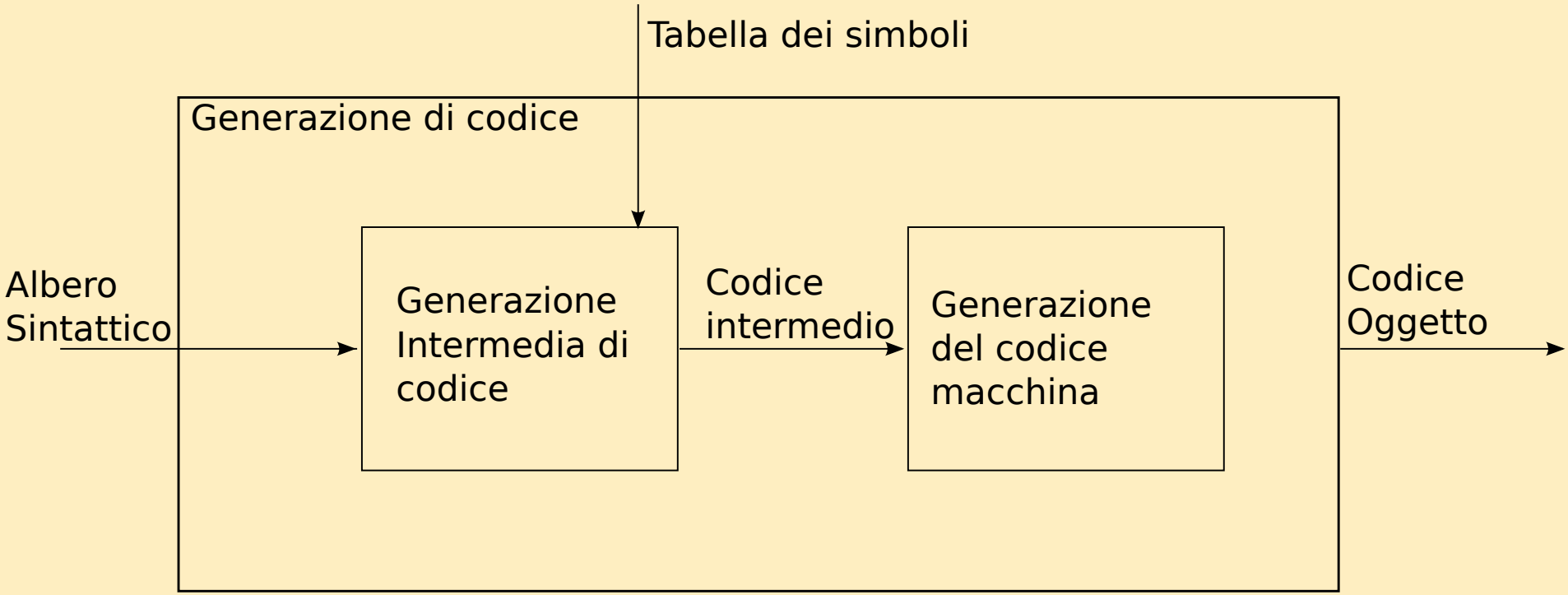


Le scatole rappresentano i moduli

Le linee orientate rappresentano le interfacce

La decomposizione modulare può essere iterata

Ulteriore modularizzazione del modulo di generazione del codice



Caso di studio: Compilatore

Astrazione

Applicata in vari casi

- Sintassi astratta per ignorare dettagli sintattici quali **begin**...**end** rispetto a {...} per delimitare sequenze di istruzioni (blocchi)
- Generazione di codice macchina intermedio (ad es., Java Bytecode) per ottenere la portabilità del codice

Anticipazione del cambiamento

- Si considerano possibili estensioni e cambiamenti nel linguaggio sorgente (dovuti alle decisioni dei comitati di standardizzazione):
  - Nuove versioni del processore target con istruzioni nuove e più potenti
  - Introduzione di nuovi dispositivi di I/O che richiedono nuovi tipi di istruzioni
- Il linguaggio Pascal introdusse una definizione rigida delle istruzioni di I/O fallendo nell'anticipazione del cambiamento. Linguaggi quali C, ADA, Java incapsulano le istruzioni di I/O in librerie standard

Caso di studio: Compilatore

Generalità

- Parametrizzazione rispetto all'architettura della macchina target (introducendo codice intermedio)
- Sviluppo di strumenti per la generazione di compilatori (*compiler compilers*) invece di un singolo compilatore (es. lex, yacc, gcc)

Incrementalità

Sviluppo incrementale

- Si consegna prima un versione parziale del compilatore che riconosce solo un sottoinsieme del linguaggio sorgente, e quindi, attraverso consegne successive, si forniscono insiemi più estesi del linguaggio
- Si consegna inizialmente un compilatore con basso livello di diagnostica e senza ottimizzazioni. Le versioni successive migliorano il livello di diagnostica ed introducono le opportune ottimizzazioni