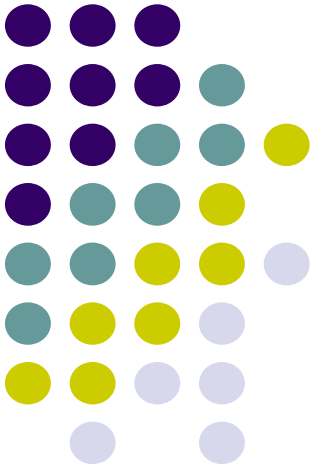
Università di Napoli Federico II - Dip. di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Corso di Laurea in Ingegneria Informatica

# Corso di Programmazione

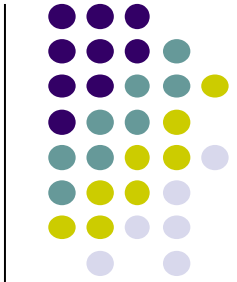
## *Programmazione modulare: concetti base*



# Il problema

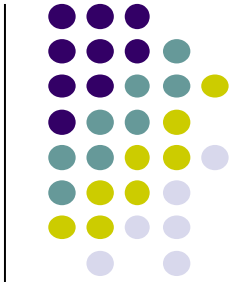
- Lo sviluppo di sistemi software complessi (programmazione in grande o “in the large”) richiede di:
  - poter suddividere il lavoro tra diversi gruppi di lavoro e di contenere i tempi e i costi di sviluppo che sono in generale molto alti
  - Garantire la qualità del software in termini di:
    - Affidabilità
    - Prestazioni
    - Modificabilità
    - Estensibilità
    - Riutilizzo
- La programmazione modulare è un necessario approccio alla ***programmazione in grande,*** in quanto consente gestire la complessità del sistema da realizzare suddividendolo in parti tra loro correlate.

# Modularizzazione



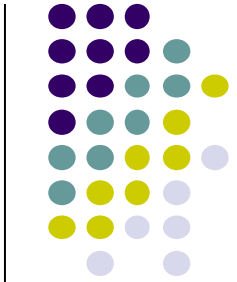
- La ***modularizzazione*** è la suddivisione in parti (moduli) di un sistema, in modo che esso risulti più semplice da comprendere e manipolare
- Essa determina la definizione di una “architettura”, che descrive l’organizzazione del sistema in parti e le interconnessioni tra di esse
- Gran parte dei **sistemi complessi** sono modulari. Ad esempio, un elaboratore:
  - Sottosistema di elaborazione
  - Sottosistema di memorizzazione
  - Sottosistema di comunicazione
  - Rete di interconnessione

# Modularizzazione e Astrazione



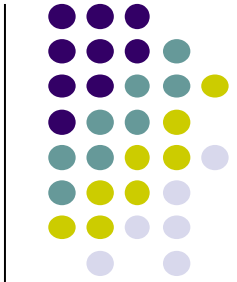
- La suddivisione di un sistema in parti richiede che ciascuna di esse realizzi un aspetto o un comportamento ben definito all'interno del sistema
- E' necessario a questo scopo che venga effettuato un processo di astrazione

# Il processo di astrazione



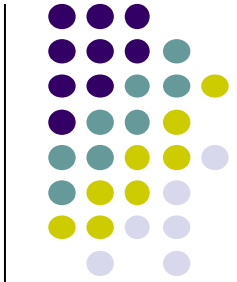
- L'*astrazione* è il processo che porta ad individuare e considerare le proprietà rilevanti di un'entità, ignorando i dettagli inessenziali
  - Le proprietà prese in considerazione definiscono una particolare vista dell'entità (che può in generale essere considerata secondo viste diverse)
- Esempio: una persona
  - vista "anagrafica":
    - Nome, cognome, data di nascita, luogo di nascita, residenza ...
  - visita "clinica":
    - Temperatura corporea, peso, pressione arteriosa, ...

# L’importanza delle astrazioni



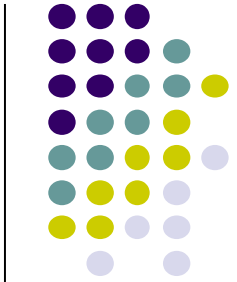
- Problemi complessi possono essere gestiti e risolti facendo uso di astrazioni, ad esempio sviluppando un modello della realtà/sistema ad un (appropriato) livello di astrazione:
  - Sufficientemente dettagliato da renderne significativa l’analisi
  - Non tanto dettagliato da rendere troppo onerosa l’analisi
  - I linguaggi di programmazione fanno uso di diverse astrazioni, ad esempio i TIPI di dato, i sottoprogrammi, etc…

# Meccanismi di astrazione



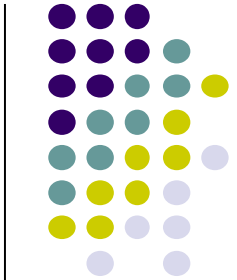
- I meccanismi di astrazione più diffusi sono:
  - Astrazione sul controllo
  - Astrazione sui dati
- L'astrazione fondamentale dei linguaggi procedurali è l'astrazione sul controllo che consiste nell'astrarre una data funzionalità dai dettagli della sua implementazione
- L'astrazione fondamentale dei linguaggi a oggetti è l'astrazione sui dati che consiste nell'astrarre le entità (oggetti) costituenti il sistema, descritte in termini di una struttura dati e delle operazioni possibili su di essa

# Il concetto di modulo



- Un modulo in un sistema software è un componente che può essere utilizzato per *realizzare una astrazione*
- In generale può esportare all'esterno:
  - servizi/funzioni ("astrazione sul controllo")
  - dati ("astrazione sui dati")
  - identificativi
- A sua volta può importare dati, funzionalità e identificativi da altri moduli
- E' definito come una unità di compilazione
- Definisce un AMBIENTE DI VISIBILITA'

# Realizzazione di un modulo

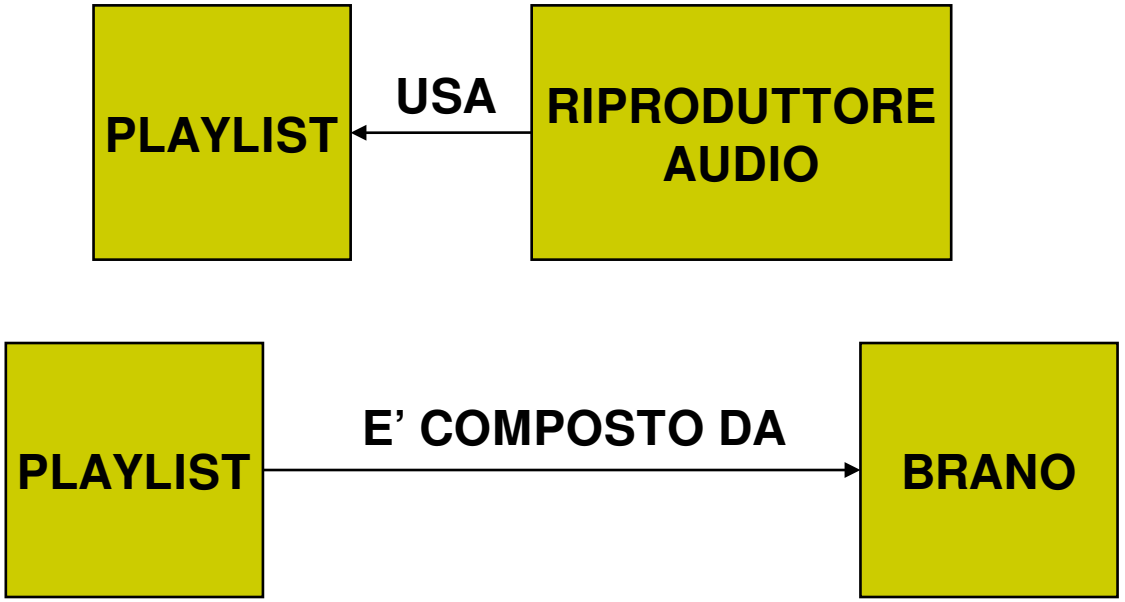


- È dotato di una chiara separazione tra:
  - *Interfaccia*
  - *Corpo*
- L'interfaccia specifica "cosa" fa il modulo (l'astrazione realizzata) e "come" si utilizza
- Il corpo descrive "come" l'astrazione è realizzata

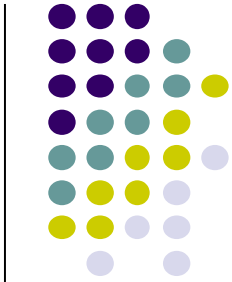

**Modulo**

# Relazioni tra moduli

- “usa”: M1 usa M2 se importa dati e/o funzionalità esportati da M2
- “è composto da”: M1 è costruito a partire da altri moduli secondo una metodologia di sviluppo “top down” o “bottom up”

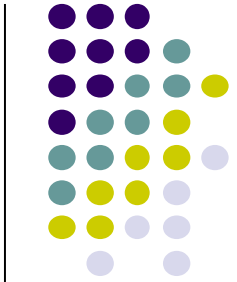


# Modularizzazione



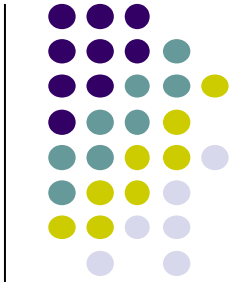
- L'organizzazione in moduli di un sistema software è un problema trattato dall'ingegneria del software
- E' possibile organizzare in moduli secondo diverse metodologie di sviluppo (Top-down, Bottom-up)
- Possono essere adottati diversi criteri che tengano conto di alcuni elementi, tra i quali:
  - Information hiding
  - coesione
  - accoppiamento

# Metodologie di sviluppo



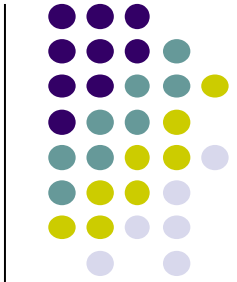
- Top down: si parte dall'alto, considerando il problema nella sua interezza e si procede verso il basso per raffinamenti successivi fino a ridurlo ad un insieme di sottoproblemi elementari
- Bottom up: si risolvono singole parti del problema, senza averne necessariamente una visione d'insieme, per poi risalire procedendo per aggiustamenti successivi fino ad ottenere la soluzione globale.

# Information hiding



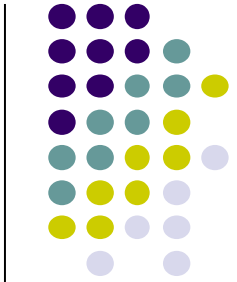
- Consiste nel nascondere e proteggere (*incapsulare*) alcune informazioni di una entità all'interno del modulo
- Alle informazioni protette l'accesso da parte degli utilizzatori del modulo è fornito in maniera controllata
- L'accesso ad esse è possibile solo attraverso l'interfaccia

# Information hiding: esempio



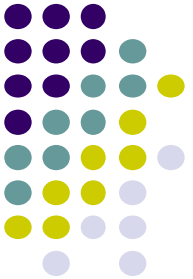
- Un modulo realizza l'astrazione dati fornendo un tipo "persona"
- Una persona è caratterizzata al livello di astrazione scelto da Codice Fiscale, Nome, Cognome, Data e Luogo di nascita, Residenza...
- Il modulo esporta attraverso la sua interfaccia il tipo "persona" ed alcune operazioni che è possibile effettuare su variabili di quel tipo, ad esempio inizializzazione dei dati della persona, stampa dei dati etc...
- Non è possibile per l'utente del modulo modificare o leggere i dati di una (variabile) persona se non utilizzando le funzionalità esportate dal modulo

# Information hiding: criteri



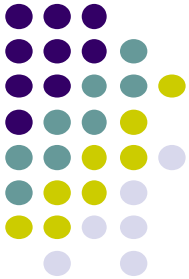
- Devono essere progettate definendo con attenzione cosa esportare e cosa nascondere
- Le interfacce devono esporre solo ciò che è strettamente necessario
- Devono essere stabili, cioè possibilmente non devono subire cambiamenti nel tempo: se l'interfaccia non cambia, le informazioni nascoste possono essere modificate senza che questo influisca sulle altre parti del sistema di cui l'entità fa parte

# Accoppiamento



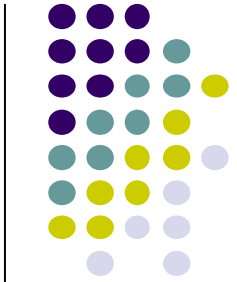
- Due moduli si dicono *debolmente accoppiati* se le dipendenze tra di essi sono minimizzate (non si sono create dipendenze non volute o non necessarie);
- Ad esempio: limitare al massimo l'uso di variabili globali, visibili e utilizzabili da più moduli poiché creano dipendenze non facilmente controllabili

# Coesione



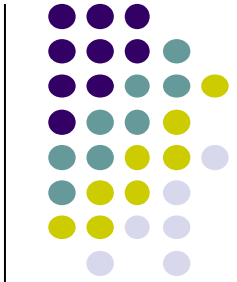
- Un modulo è fortemente coeso se incapsula un insieme di caratteristiche omogenee, sufficientemente indipendenti da altri moduli
- Un esempio di modulo fortemente coeso è un modulo che realizza un' unica astrazione (Persona, Ordina)

# Coesione e Accoppiamento: criteri



- Nello sviluppo software coesione e accoppiamento suggeriscono criteri contrastanti, tra i quali è opportuno valutare un tradeoff
- L'ideale sarebbe riuscire ad ottenere una alta coesione tra i moduli ed un basso accoppiamento, ma:
  - Un livello molto alto di coesione dei moduli genera in generale una eccessiva crescita delle dipendenze tra di essi
  - Un livello molto basso di accoppiamento può determinare uno scadere della qualità delle astrazioni realizzate dai moduli

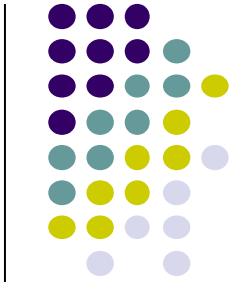
# Un esempio in pseudo-codice



```
/*interfaccia*/
module interface PERSONA
import cin, cout from iostream;
end import;
export type hidden Persona;
Persona CreaPersona(const char *; const char *, const char *);
void StampaDatiPersona();
void ModificaResidenza(const char *);
<altre operazioni...>
end export;
end interface

/*corpo */
module implementation PERSONA
typedef struct {char Nome[20]; char Cognome[20]; char Residenza[40];} Persona;
Persona crea(const char * N; const char * C, const char * R) {
/* codice funzione */
}
void StampaDatiPersona()  {
/* codice funzione */
}
void ModificaResidenza(const char * R){
/* codice funzione */
}
<definizioni di altre funzioni>
end implementation
```

# Nell'esempio

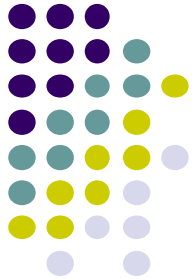


- E' esplicitamente definito un modulo PERSONA
- Sono esplicitate la sezione di interfaccia e il corpo
- E' esplicitamente definito cosa viene esportato e cosa viene importato
- E' realizzato l'information hiding:
  - Sui tipi: il tipo persona definito dal modulo è esplicitamente incapsulato nel modulo, l'utente potrà dichiarare variabili del tipo Persona ma *non ha la visibilità della dichiarazione del tipo* (tipo opaco) nè ha necessità di sapere come è realizzato.
  - Sulle funzioni: l'implementazione delle operazioni è esplicitamente incapsulata nel corpo del modulo, l'utente ha visibilità dei prototipi esportati dall'interfaccia e attraverso di essi potrà invocare le funzioni.

# Vantaggi

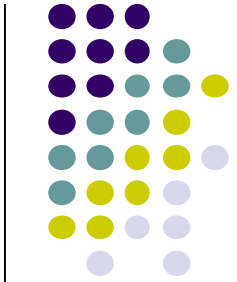
- Seguendo i criteri a cui abbiamo accennato si riesce ad ottenere che:
  - L'implementazione del modulo dipenda in modo minimo dall' "esterno"
  - Le modifiche apportate ad un modulo impattino in modo minimo sull' "esterno"
- Ciò comporta i seguenti vantaggi:
  - Riuso
  - Modificabilità
  - Estensibilità
  - Manutenibilità
  - Leggibilità
  - Protezione
  - Sviluppo separato

# Meccanismi e Linguaggi



- La strutturazione di un programma in moduli deve essere supportata da opportuni meccanismi offerti dal linguaggio di programmazione
- Non tutti i linguaggi sono “modulari”, nel senso di offrire costrutti espliciti per la strutturazione dei programmi in moduli. Alcuni linguaggi modulari sono: ADA, Mesa, Modula2
- Il C non è un linguaggio modulare perché non offre costrutti espliciti, una programmazione modulare è possibile mediante un uso auto-disciplinato degli strumenti del linguaggio
- Il C++ e JAVA offrono meccanismi espliciti a supporto dell’information hiding, ma non prevede costrutti per la definizione esplicita dei moduli

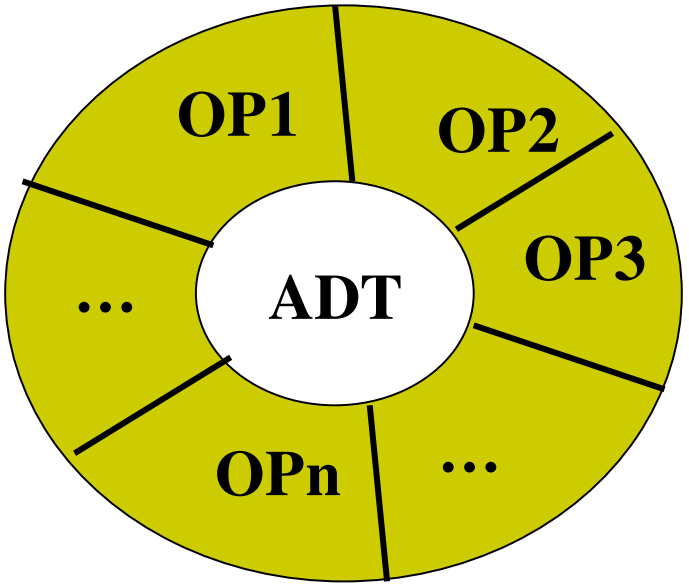
# Utilizzo dei Moduli



- Per realizzare LIBRERIE software: per offrire collezioni coese di procedure e funzioni
- Per realizzare astrazioni di dato, utilizzando in maniera disciplinata gli strumenti offerti dal linguaggio.
- ***Per implementare Tipi di Dati Astratti, avendo a disposizione meccanismi forniti dal linguaggio per l'information hiding***

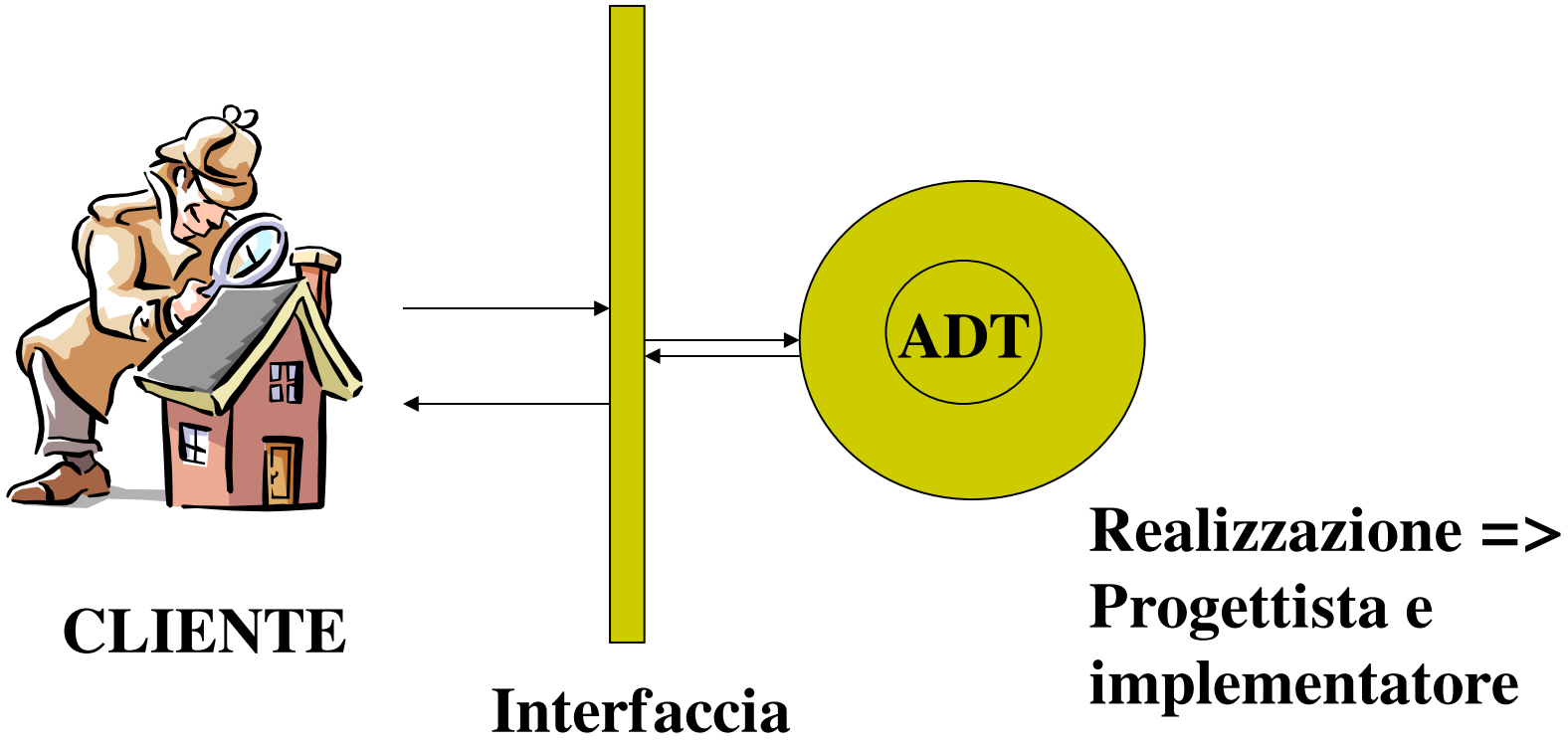
# Definizione (ADT)

- Un tipo di dato astratto (Abstract Data Type: ADT) è una struttura dati incapsulata dalle operazioni consentite su di essa
- Non è possibile accedere alla struttura dati (né in lettura né in scrittura) se non attraverso le operazioni definite su di essa

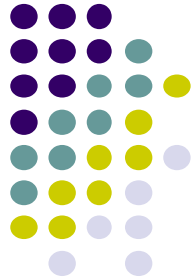


# Metodologicamente:

- Interfaccia
- Realizzazione



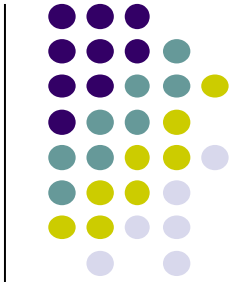
# Clienti e sviluppatori



- Il cliente fa uso delle funzionalità offerte per realizzare procedure di un'applicazione o per costruire operazioni su dati più complessi
- Il progettista e lo sviluppatore devono realizzare le astrazioni e le funzionalità previste per il dato facendo uso di un linguaggio di programmazione o appoggiandosi sulle primitive di un altro tipo di dato già disponibile

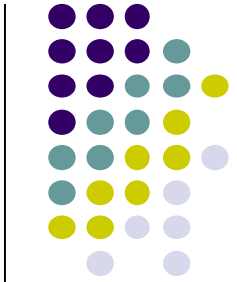
  - **Un progettista o uno sviluppatore può essere il cliente di un altro tipo di dato astratto**

# Vantaggi principali



- La struttura dei dati non può venire alterata da operazioni scorrette
- La realizzazione della struttura dei dati può essere modificata senza influenzare i moduli che ne fanno uso

# Tecnicamente...

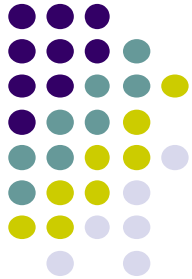


**In C++**

- Meccanismi per l'Information Hiding
- Descrizione (o specifica) dell'interfaccia
- Compilazione separata e inclusione di file (#include)
- C++20 introduce meccanismi linguistisci per la dichiarazione e l'utilizzo di moduli.

**In JAVA**

- Meccanismi per l'Information Hiding
- Packages
- Java 9 ha introdotto la possibilità di introdurre esplicitamente i moduli ed è possibile specificare le relazioni e le dipendenze tra essi ( requires, exports, provides...with, uses e opens)



- Esempi in compilazione separata in C++
- Assegnare Homework