Verifica e validazione: analisi statica

IS

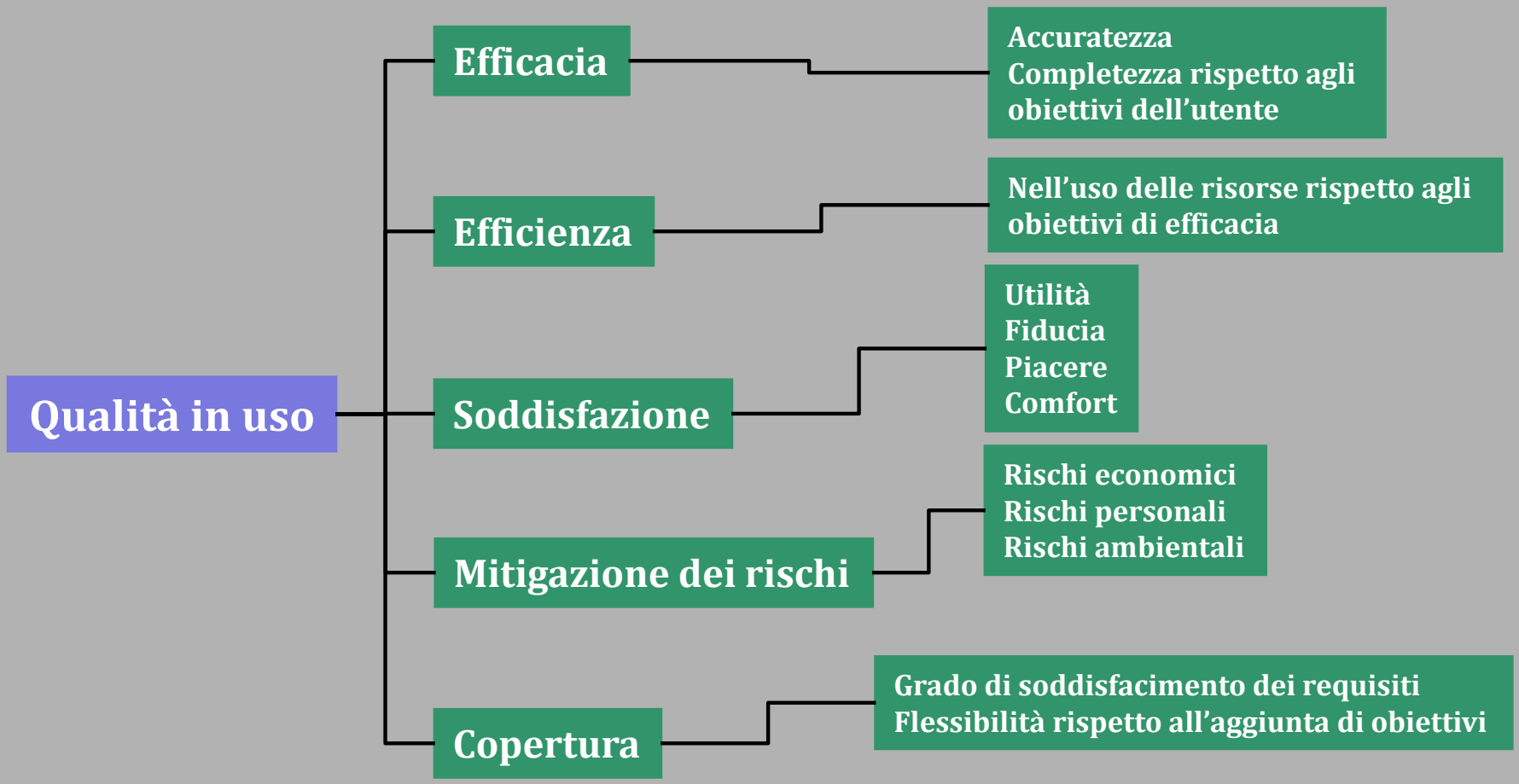
Anno accademico 2023/2024
Ingegneria del Software

Tullio Vardanega, tullio.vardanega@unipd.it

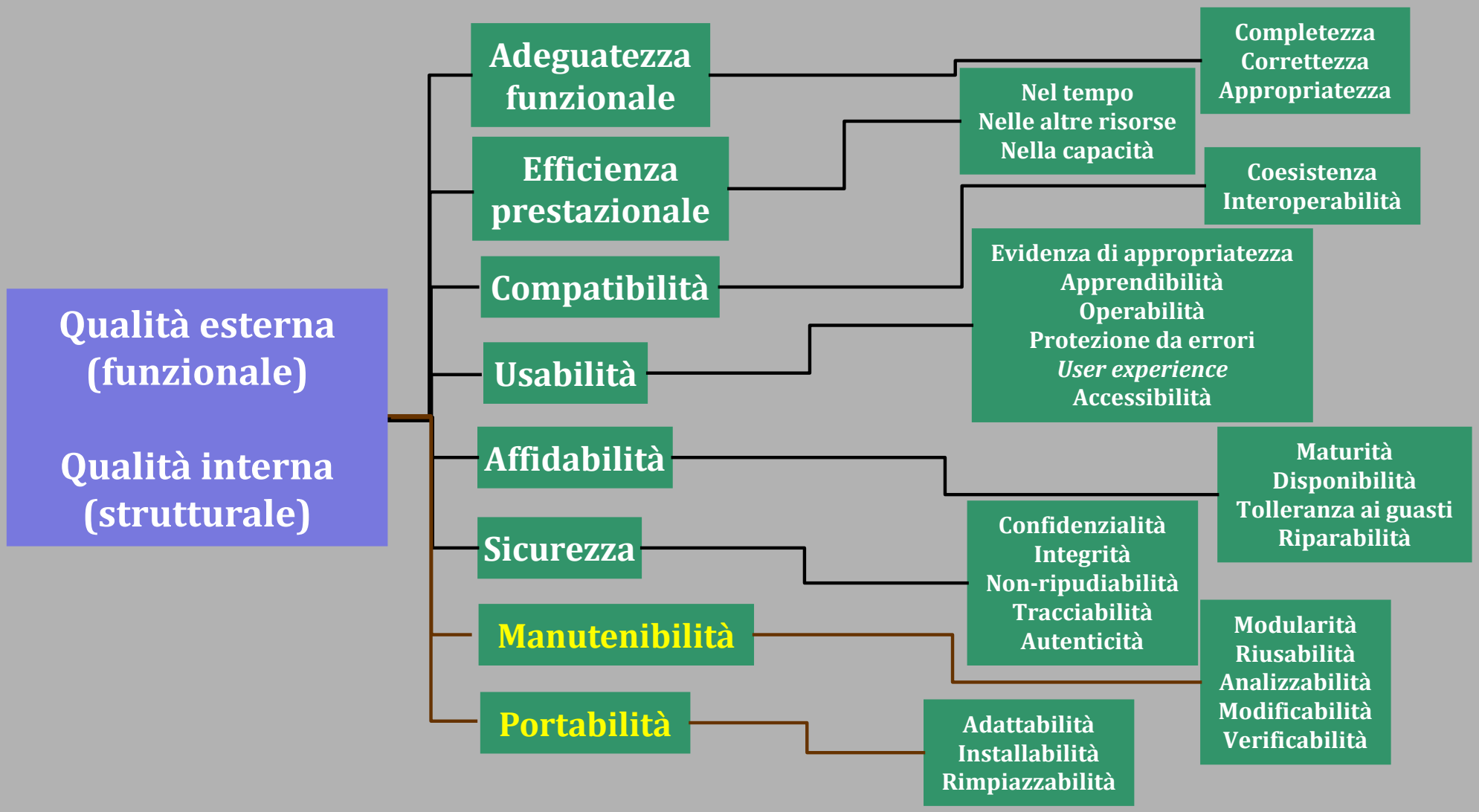
Premessa – 1/2

- **Un SW di qualità *deve* possedere**
  - **Tutte le capacità funzionali attese, che specificano cosa il sistema debba fare**
  - **Tutte le caratteristiche non-funzionali necessarie affinché il sistema lavori sempre come previsto**
- **Dimostrarlo richiede accertare il possesso di svariate proprietà**
  - **Di costruzione: architettura, codifica, integrazione**
  - **D'uso: esperienza utente, precisione, affidabilità**
  - **Di funzionamento: prestazioni, robustezza, sicurezza**

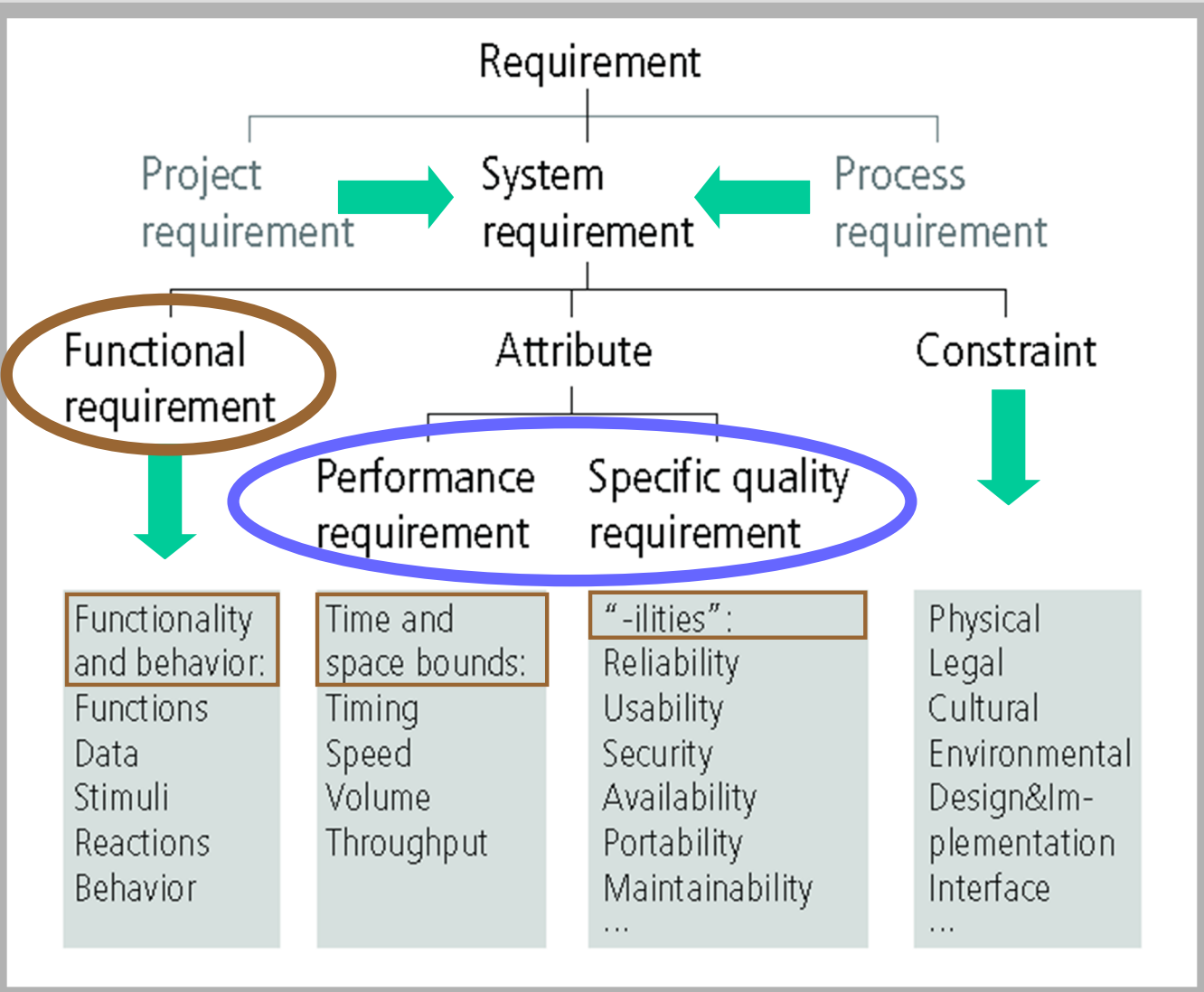
Richiami – 1/4


Qualità in uso
Efficacia
Accuratezza
Completezza rispetto agli obiettivi dell'utente
Efficienza
Nell'uso delle risorse rispetto agli obiettivi di efficacia
Soddisfazione
Utilità
Fiducia
Piacere
Comfort
Mitigazione dei rischi
Rischi economici
Rischi personali
Rischi ambientali
Copertura
Grado di soddisfacimento dei requisiti
Flessibilità rispetto all'aggiunta di obiettivi

Richiami – 2/4


Qualità esterna (funzionale)
Qualità interna (strutturale)
Adeguatezza funzionale
Efficienza prestazionale
Compatibilità
Usabilità
Affidabilità
Sicurezza
Manutenibilità
Portabilità
Completezza
Correttezza
Appropriatezza
Nel tempo
Nelle altre risorse
Nella capacità
Coesistenza
Interoperabilità
Evidenza di appropriatezza
Apprendibilità
Operabilità
Protezione da errori
User experience
Accessibilità
Maturità
Disponibilità
Tolleranza ai guasti
Riparabilità
Confidenzialità
Integrità
Non-ripudiabilità
Tracciabilità
Autenticità
Modularità
Riusabilità
Analizzabilità
Modificabilità
Verificabilità
Adattabilità
Installabilità
Rimpiazzabilità

Richiami – 3/4


Requirement
Project requirement
System requirement
Process requirement
Functional requirement
Attribute
Constraint
Performance requirement
Specific quality requirement
Functionality and behavior:
Functions
Data
Stimuli
Reactions
Behavior
Time and space bounds:
Timing
Speed
Volume
Throughput
"-ilities":
Reliability
Usability
Security
Availability
Portability
Maintainability
...
Physical
Legal
Cultural
Environmental
Design&Im-plementation
Interface
...

Richiami – 4/4

- **E.W. Dijkstra, *On the role of scientific thought* :**
  - ***The task of "making a thing satisfying our needs", as a single responsibility, is split into two parts:***

  1) Stating the properties of a thing, by virtue of which, it would satisfy our needs
  [ANALISI E SPECIFICA DEI REQUISITI]

  2) Making a thing that is guaranteed to have the stated properties
  [PROGETTAZIONE E CODIFICA DELLA SOLUZIONE]

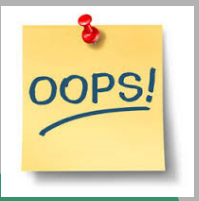
- **La verifica – incluso il tracciamento – accerta e assicura che 2) soddisfi 1)**

Premessa – 2/2

- La codifica deve *aiutare* la verifica, non ostacolarla
  - Pochi linguaggi la facilitano attivamente
  - Per questo serve imporre disciplina di programmazione
- L'uso di funzionalità «oscure» ostacola l'accertamento di integrità
  - La programmazione non può essere ottimistica (non sono sicuro ma spero che funzioni …)
- Le norme di codifica devono bilanciare la ricchezza di funzionalità con le garanzie di integrità

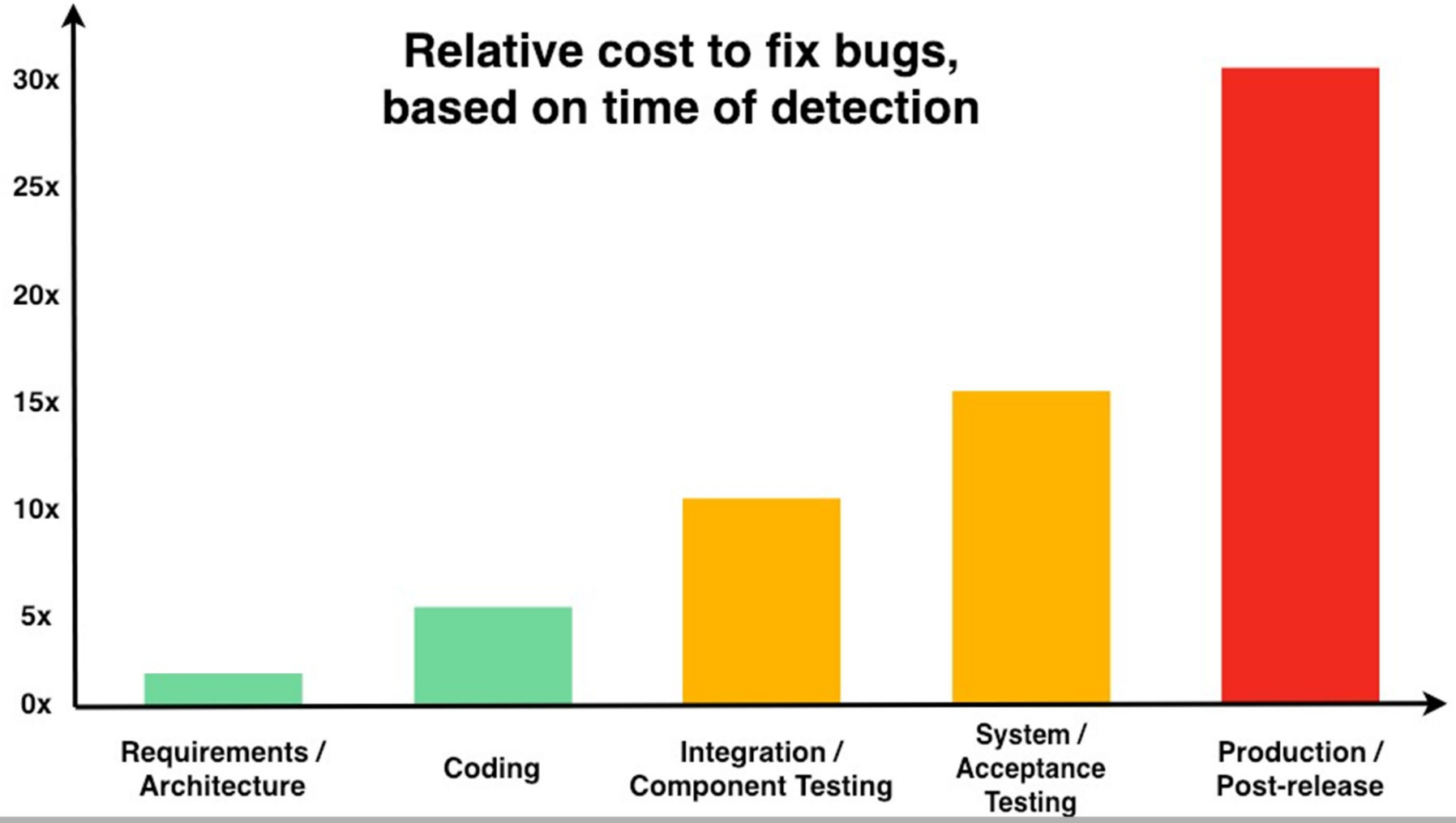
Scrivere programmi verificabili – 1/3

- Dotarsi di norme di codifica coerenti con le esigenze di verifica
  - Promuovendo buone prassi e ponendo vincoli sui costrutti di programmazione inappropriati
  - Verificandone attivamente il rispetto
- La verifica retrospettiva è <u>insufficiente</u>



- Il costo di rilevazione e correzione di errori cresce con l'avanzare dello sviluppo

Costo di correzione di errori nel SW



Scrivere programmi verificabili – 2/3

- **L'approccio reattivo alla verifica è ingenuo, pigro, ottimistico**
  - ***Hoping for correctness by correction***

- **È più saggio sostenere lo sviluppo con la verifica sistematica: approccio costruttivo**
  - ***Pursuing correctness by construction***

Scrivere programmi verificabili – 3/3

- Regolamentare l'uso del linguaggio di programmazione tramite principi da riflettere nelle Norme di Progetto
  1. Per assicurare comportamento predicibile
  2. Per usare buoni principi di programmazione
  3. Per ragioni pragmatiche
- Vediamo ciascuna di queste tre dimensioni

1. Comportamento predicibile

- **Codice sorgente senza ambiguità**
  - **Effetti laterali (p.es. di sottoprogrammi)**
    - Invocazioni della stessa azione che producano effetti diversi
  - **Ordine di elaborazione e inizializzazione**
    - L'effetto del programma può dipendere dall'ordine di **elaborazione** o l'ordine di **esecuzione** delle sue parti
    - Esempio: impredicibilità nell'attivazione di *thread*
  - **Modalità di passaggio dei parametri**
    - La scelta di una modalità di passaggio (per valore, per riferimento) può influenzare l'esito dell'esecuzione

Funziona?

```
class Swapper{
  public static void swap(int Left, int Right)
  {
    int tmp = Left;
    Left = Right;
    Right = tmp;
  }

  public static void main(String args[])
  {
    int Source = 1;
    int Destination = 3;
    swap(Source, Destination);
  }
}
```

In Java, i nomi sono riferimenti, ma le chiamate sono per valore!

2. Principi di programmazione

- **Riflettere l'architettura (*design*) nel codice**
  - Usare programmazione strutturata per esprimere componenti, moduli, unità come da progettazione, e facilitare l'integrazione
- **Separare le interfacce dall'implementazione**
  - Fissare bene le interfacce a partire dall'architettura logica
  - Esporre le interfacce, nascondere l'implementazione
- **Massimizzare l'incapsulazione (*information hiding*)**
  - Usare membri privati e metodi pubblici per l'accesso ai dati
- **Usare tipi specializzati per specificare dati**
  - La composizione e la specializzazione aumentano il potere espressivo del sistema di tipi del programma

3. Considerazioni pragmatiche

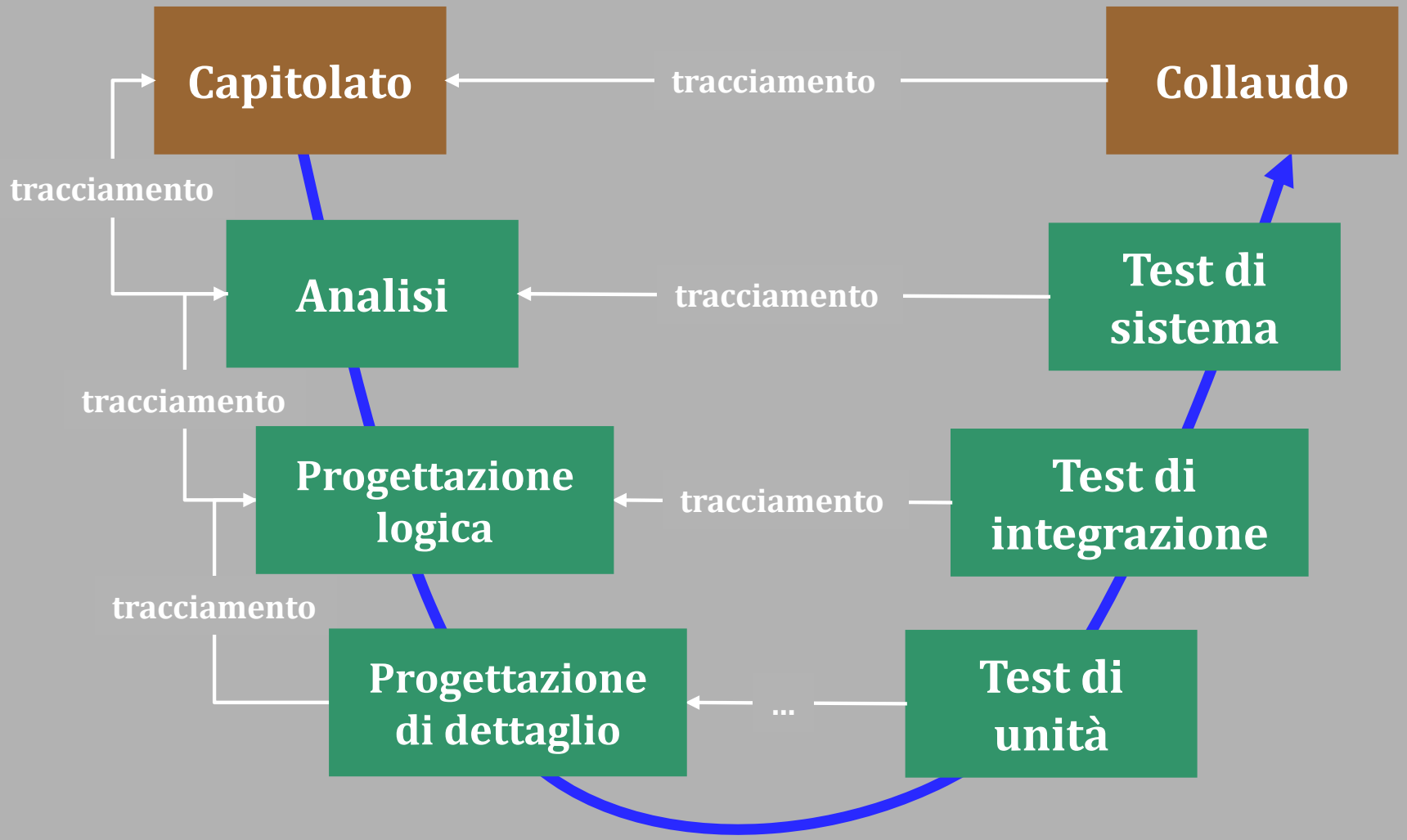
- **L'efficacia dei metodi di verifica è funzione della qualità di strutturazione del codice**
  - **Esempio: una procedura con un solo punto di uscita facilita l'analisi del suo effetto sullo stato del sistema**
- **La verifica di un programma mette in relazione segmenti di codice con porzioni di specifica**
  - **La verificabilità è dipende inversamente dall'ampiezza del contesto oggetto di verifica**
    - Più ampio il contesto, più difficile e costoso verificare: confinare *scope* e visibilità
  - **Una buona architettura facilita la verifica**
    - P.es. tramite incapsulazione dello stato e controllo di accesso

Tracciamento – 1/3

- **Dimostra completezza ed economicità del prodotto**
  - **Nessun requisito dimenticato**
  - **Nessuna funzionalità superflua**

- **Va applicato**
  - **Su ogni passaggio della specifica (ramo discendente)**
  - **A ogni passo di implementazione (ramo ascendente)**
- **Va automatizzato il più possibile**
  - **Per limitarne il costo all'aumentare della sua intensità**

Tracciamento – 2/3



Tracciamento – 3/3

- **Tracciare i requisiti su progettazione di dettaglio e codifica aiuta a valutare il costo di verifica**
  - **Assegnare N requisiti elementari a 1 singolo modulo SW richiede N procedure di prova per quel modulo**
    (1 prova per 1 requisito aiuta a rendere le prove decidibili)
  - **Al crescere di N crescono la criticità e il costo di quel modulo**
- **Maggiore il potere espressivo di un costrutto, maggiore la sua complessità di esecuzione, maggiore il costo di dimostrarlo corretto**
  - **Basso potere espressivo: addizione tra interi, …**
  - **Alto potere espressivo: attivazione di *thread*, invocazione di API esterne, …**

Tipi di analisi statica del codice

A. Flusso di controllo
B. Flusso dei dati
C. Flusso dell'informazione
D. Esecuzione simbolica
E. Verifica formale del codice
F. Verifica di limite
G. Uso dello *stack*
H. Comportamento temporale
I. Interferenza
J. Codice oggetto

Prima di e in aggiunta all'analisi dinamica

Analisi di flusso di controllo

- **Per accertare**
  - **Logica: l'esecuzione avverrà nella sequenza specificata**
  - **Visibilità e propagazione: il codice è ben strutturato**
- **Per localizzare codice non raggiungibile**
- **Per identificare rischi di non terminazione**
  - **L'analisi dell'albero delle chiamate (*call-tree analysis*) mostra se l'ordine di chiamata corrisponda alla specifica**
  - **E segnala la presenza di ricorsione diretta o indiretta**
  - **La modifica di variabili di controllo delle iterazioni è fonte di vulnerabilità rispetto alla terminazione**

Analisi di flusso dei dati

- Per accertare che nessun cammino d'esecuzione del programma acceda a variabili non valorizzate
  - Concentrando l'analisi di flusso di controllo sulla sequenza di accesso alle variabili e le sue modalità (lettura, scrittura)
- Per rilevare possibili anomalie
  - Scritture successive senza letture intermedie
  - Letture che precedano scritture
- Per accertare l'assenza di variabili globali
  - E di altre violazioni al principio di incapsulazione

Analisi di limite

- Per verificare che i valori del programma restino sempre entro i limiti del loro tipo e della precisione desiderata
- L'*overflow* produce valori maggiori del massimo rappresentabile
  - Può causare eccezioni o silenziosamente produrre valori errati
- L'*underflow* produce valori più piccoli del minimo rappresentabile
  - Può causare eccezioni o grande perdita di precisione
- Rispetto dei limiti nell'accesso a strutture dati (*range checking*)
- Alcuni linguaggi permettono di assegnare limiti statici a tipi discreti per facilitare verifica sulle corrispondenti variabili
  - Più difficile farlo con tipi enumerati e reali

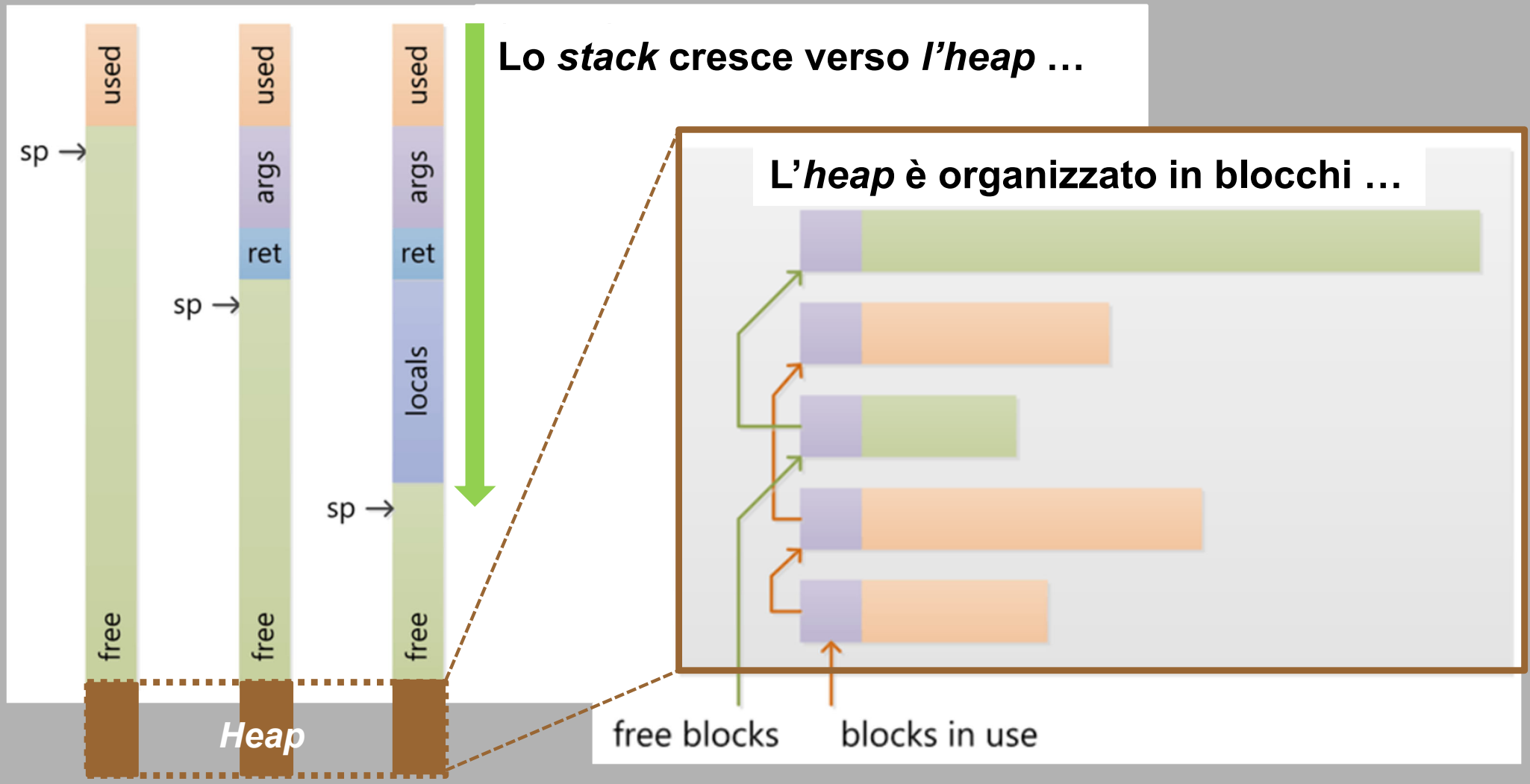
Analisi d'uso di *stack*

- Per determinare la massima domanda di *stack* richiesta a tempo d'esecuzione in relazione con la dimensione della memoria assegnata all'esecuzione del programma
- Per verificare che non vi sia rischio di collisione tra *stack* e *heap* per qualche esecuzione

Stack & *heap* – 1/2

- Lo *stack* è la memoria usata per ospitare dati locali e indirizzi di ritorno generati dal compilatore alla chiamata di sottoprogrammi
  - Ogni flusso di controllo (*main*, *thread*) ha il suo *stack*
  - La sua dimensione cresce con l'annidamento di chiamate
  - I dati in esso hanno chiare regole di visibilità e ciclo di vita
- L'*heap* è la memoria usata per tutto il resto (globale)
  - Dimensione fissata prima dell'avvio del programma
  - Contenuto determinato dalla quantità di oggetti globali creati durante l'esecuzione del programma
  - Regole di visibilità e ciclo di vita difficili da controllare

Stack & heap – 2/2



Analisi temporale

- Per studiare le dipendenze temporali (latenza) tra le uscite del programma e i suoi ingressi
  - Per verificare che il valore giusto sia prodotto al momento giusto
- Limiti espressivi dei linguaggi e delle tecniche di programmazione complicano questa analisi
  - Iterazioni prive di limite statico (*while*)
  - Creazione dinamica di variabili (*new*)
  - ...