GUIDA OPERATIVA ESAME - TEORIA TULLIO

STRUTTURA ESAME

Formato: Domande aperte (NON quiz multipli)
Tempo: ~20-25 minuti per domanda teoria
Punteggio: 4-5 punti per domanda (su 30 totali)
Valutazione: Precisione terminologica + Struttura ISO 12207 + Analisi critica

---

PATTERN DOMANDE RICORRENTI (ULTIMI 15 ANNI)

TIPO 1: Verifica vs Validazione (90% probabilità)

FORMULE POSSIBILI:

- "Facendo riferimento allo standard ISO/IEC 12207, discutere la differenza di obiettivi, tecniche e strategie di conduzione tra i processi di verifica e validazione"
- "Indicare le differenze (natura, finalità, collocazione) che intercorrono tra le attività di verifica e validazione"
- "Discutere la differenza di obiettivi, attività coinvolte, strategie di conduzione e strumenti, tra i processi di verifica e validazione"

TEMPLATE RISPOSTA (600-800 parole):

```
VERIFICA (Software Verification)

Definizione ISO/IEC 12207:
Processo che fornisce prove oggettive che i risultati di un particolare
segmento del ciclo di vita dello sviluppo software soddisfano tutti i
requisiti specificati.

Obiettivo:
Accertare che l'esecuzione delle attività non abbia introdotto errori
nel prodotto. Risponde alla domanda: "Did I build the system right?"

Natura e Collocazione:
- Processo CONTINUO applicato ad ogni segmento temporale (fase)
- Interesse INTERNO al fornitore
- Si applica a prodotti intermedi e baseline
- Supporta il successo della validazione

Strategie di conduzione:
1. ANALISI STATICA (senza esecuzione):
```

   - Inspection: lettura mirata con presupposti, verificatori separati
   - Walkthrough: lettura critica ampio spettro, gruppi misti

2. ANALISI DINAMICA (con esecuzione):
   - Test di unità: singole unità architetturali
   - Test di integrazione: componenti integrate
   - Test di sistema: sistema completo vs requisiti

Tecniche e Strumenti:
- Ciclo PDCA per miglioramento continuo
- Tracciamento requisiti verso componenti
- Controllo configurazione (baseline e repository)
- Metriche di qualità (efficacia, efficienza)

Attività coinvolte:
- Revisione documenti prodotti
- Esecuzione test automatizzati
- Verifica conformità a standard
- Approvazione baseline associate a milestone

VALIDAZIONE (Software Validation)

Definizione ISO/IEC 12207:
Conferma mediante esame e presentazione di prove oggettive che le specifiche software sono conformi alle esigenze dell'utente e agli usi previsti.

Obiettivo:
Accertare che il prodotto realizzato corrisponda alle attese. Risponde alla domanda: "Did I build the right system?"

Natura e Collocazione:
- Processo FINALE (self-fulfilling prophecy)
- Interesse ESTERNO (coinvolge committente)
- Si applica al prodotto completo
- Conferma conformità requisiti utente

Strategie di conduzione:
1. COLLAUDO FORMALE:
   - Eseguito dal committente su casi di prova contrattuali
   - Valore contrattuale
   - Conclusione della commessa

2. TRACCIAMENTO REQUISITI:
   - Dimostrazione che ogni user need è soddisfatto
   - Approccio top-down (need → requisiti SW)
   - Approccio bottom-up (requisiti SW → need)

Tecniche e Strumenti:

```
- Test funzionali con committente
- Verifica scenari d'uso
- Tracciabilità completa requisiti
- Accettazione formale

Attività coinvolte:
- Dimostrazione copertura requisiti utente
- Esecuzione test accettazione
- Consegna prodotto finale
- Firma contrattuale


RELAZIONE TRA VERIFICA E VALIDAZIONE

La verifica prepara e garantisce il successo della validazione. Senza
verifica continua, ci si affaccia alla validazione senza certezza
dell'esito.

Differenze chiave:
- QUANDO: Verifica continua vs Validazione finale
- CHI: Verifica interna vs Validazione con committente
- COSA: Verifica processo vs Validazione prodotto
- PERCHÉ: Verifica assicura vs Validazione accerta

Nel progetto didattico:
[Qui inserisci 2-3 righe sulla tua esperienza concreta]
- Es: "La verifica è stata applicata tramite inspection documenti
  e test automatizzati ad ogni baseline RTB/PB. La validazione
  è avvenuta al collaudo finale tramite dimostrazione scenari al
  committente."
```

TIPO 2: Strategia Testing/Verifica Progetto (80% probabilità)

FORMULE POSSIBILI:

- "Illustrare la strategia di testing adottata nel proprio progetto didattico, valutarne l'efficacia e discutere brevemente se e come avrebbe potuto essere migliore"
- "Proporre concisamente una strategia di verifica tramite test sostenibile per un progetto di taglia analoga al progetto didattico"
- "Illustrare concisamente la strategia di verifica tramite test adottata nel progetto didattico (quali tipi di test, quali obiettivi, quale grado di automazione). Discutere gli spazi di miglioramento rilevati"

TEMPLATE RISPOSTA (500-700 parole):

STRATEGIA DI TESTING ADOTTATA

Tipi di test implementati:

1. TEST DI UNITÀ
Obiettivi:
- Verificare correttezza singole unità architetturali
- Isolamento dipendenze tramite mock
- Copertura logica interna

Oggetti del test:
- Singole classi/moduli senza dipendenze significative
- Componenti con correttezza non accertabile "a vista"

Ingressi/Uscite:
- Input: casi limite, valori normali, valori errati
- Output: asserzioni su risultati attesi (oracolo)

Grado di automazione:
- Automazione completa tramite framework [Jest/JUnit/PyTest]
- Esecuzione automatica ad ogni commit (CI/CD)
- Coverage obiettivo: 80% statement coverage

2. TEST DI INTEGRAZIONE
Obiettivi:
- Verificare corretta integrazione componenti
- Accertare interfacce tra moduli

Oggetti del test:
- Componenti integrate secondo architettura
- Flussi dati tra componenti

Strategia:
- Approccio bottom-up (dai moduli ai sottosistemi)
- Test API endpoints
- Test database integration

Grado di automazione:
- Parzialmente automatizzato
- Test manuali per UI integration

3. TEST DI SISTEMA
Obiettivi:
- Verificare copertura requisiti
- Validare scenari d'uso end-to-end

Oggetti del test:
- Sistema completo
- Requisiti funzionali e non funzionali

Strategia:
- Test scenari da analisi requisiti
- Test accettazione con proponente
- Test regressione prima di rilasci

Grado di automazione:
- Test E2E automatizzati per flussi critici
- Test manuali per validazione UX

VALUTAZIONE EFFICACIA

Punti di forza:
- Automazione test unità ha permesso rilevazione precoce difetti
- CI/CD ha garantito non-regressione
- Coverage metrics hanno guidato sviluppo

Limiti riscontrati:
- Test integrazione poco automatizzati (alto effort manuale)
- Mock dependencies complesse da mantenere
- Test E2E fragili (timeout, race conditions)

POSSIBILI MIGLIORAMENTI

1. AUMENTARE AUTOMAZIONE INTEGRAZIONE
Come: Framework contract testing (Pact/Spring Cloud Contract)
Beneficio: Riduzione effort manuale, maggiore affidabilità

2. MIGLIORARE STABILITÀ E2E
Come: Pattern wait-for-condition invece di timeout fissi
Beneficio: Eliminare flakiness, test più affidabili

3. TEST DI MUTAZIONE
Come: Introdurre mutation testing per validare efficacia test
Beneficio: Verificare che test rilevano effettivamente difetti

4. PROPERTY-BASED TESTING
Come: QuickCheck/Hypothesis per test proprietà
Beneficio: Esplorazione automatica edge cases

SOSTENIBILITÀ

Per progetto taglia analoga:
- Priorità: Test unità (ROI alto, effort basso)
- Test integrazione mirati su interfacce critiche
- Test sistema su scenari core (Pareto 20/80)
- Automazione progressiva (iniziare manuale, automatizzare ripetitivi)

```
Rapporto costi/benefici:
- Test unità: costo iniziale medio, beneficio continuo alto
- Test integrazione: costo medio, beneficio medio-alto
- Test E2E: costo alto, beneficio variabile (prioritizzare critici)

Risorse minime:
- Framework testing + CI/CD setup: 1 sprint
- Scrittura test: 30-40% tempo sviluppo feature
- Manutenzione test: 10-15% effort continuo
```

TIPO 3: Tracciamento Requisiti (60% probabilità)

FORMULE POSSIBILI:

- "Descrivere la tecnica di classificazione e tracciamento dei requisiti adottata nel proprio progetto didattico e discuterne l'efficacia e i limiti riscontrati"
- "Relazione tra tracciamento requisiti e validazione"

TEMPLATE RISPOSTA (400-600 parole):

```
TECNICA DI TRACCIAMENTO REQUISITI

Classificazione adottata:

GERARCHIA IDENTIFICATIVI:
- R[tipo].[categoria].[numero]
  Es: RF.1.2 = Requisito Funzionale, categoria 1, numero 2
      RNF.2.3 = Requisito Non Funzionale, categoria 2, numero 3

Tipi di requisiti:
- RF: Requisiti Funzionali (servizi che sistema deve fornire)
- RNF: Requisiti Non Funzionali (vincoli prestazioni, qualità)
- RV: Requisiti di Vincolo (tecnologie, standard obbligatori)

Struttura tracciamento:
- Livello 1: User Needs (bisogni stakeholder)
- Livello 2: Requisiti Utente (cosa utente si aspetta)
- Livello 3: Requisiti Software (come sistema soddisfa)
- Livello 4: Componenti architetturali (implementazione)


STRUMENTI UTILIZZATI

Database relazionale:
- Tabella Requisiti: ID, Descrizione, Tipo, Priorità, Stato
```

- Tabella Dipendenze: RequisitoID, DipendeDA
- Tabella Tracciabilità: RequisitoID, ComponenteID

Matrice di tracciabilità:
- Requisiti (righe) × Componenti (colonne)
- Celle: ✓ se componente soddisfa requisito

Tool:
- [Jira/Redmine/Excel/Database custom]
- Query SQL per verificare copertura
- Script automatici per validazione consistenza

EFFICACIA RISCONTRATA

Benefici:
- Identificazione precoce requisiti mancanti
- Verifica impatto modifiche (change impact analysis)
- Dimostrazione copertura per validazione
- Supporto decisionale prioritizzazione

Approccio utilizzato:
- TOP-DOWN: User Need → Requisiti SW (analisi)
- BOTTOM-UP: Requisiti SW → User Need (verifica copertura)

Esempio pratico:
Need N1: "Utente deve poter cercare prodotti"
  ├─ RF.1.1: "Sistema fornisce barra di ricerca"
  ├─ RF.1.2: "Sistema filtra risultati per categoria"
  └─ RF.1.3: "Sistema ordina risultati per rilevanza"
    └─ Componenti: SearchBar, FilterEngine, RankingAlgorithm

LIMITI RISCONTRATI

1. EFFORT MANUTENZIONE
Problema: Aggiornamento matrice dopo ogni modifica requisiti
Impatto: Rischio inconsistenze se non aggiornato
Mitigazione: Script validazione automatica in CI

2. GRANULARITÀ REQUISITI
Problema: Requisiti troppo fini → esplosione tracciamenti
Soluzione: Raggruppamento in feature logiche

3. REQUISITI IMPLICITI
Problema: Bisogni non esplicitati non tracciati
Soluzione: Brainstorming con proponente, analisi dominio

4. TRACEABILITY TOOL OVERHEAD
Problema: Tool complex vs beneficio per progetto piccolo

```
Soluzione: Approccio pragmatico (Excel/DB semplice sufficiente)


RELAZIONE CON VALIDAZIONE

Il tracciamento è essenziale per validazione perché:
- Dimostra COMPLETEZZA: ogni need coperto da requisiti
- Dimostra NECESSITÀ: ogni requisito giustificato da need
- Permette COLLAUDO: test accettazione basati su tracciabilità
- Fornisce EVIDENZA OGGETTIVA: prove conformità per committente

Durante collaudo:
"Il sistema soddisfa need N1? Sì, perché implementa RF.1.1,
RF.1.2, RF.1.3 verificati in componenti X, Y, Z tramite test
TS.1.1, TS.1.2, TS.1.3 con esito positivo."


MIGLIORAMENTI POSSIBILI

Per progetti futuri:
- Integrazione con tool requirements management (Jama, Doors)
- Tracciamento bidirezionale automatico codice ⇔ requisiti
- Dashboard metriche tracciabilità real-time
- Tool diff per impact analysis automatico
```

TIPO 4: Modelli Sviluppo (50% probabilità)

FORMULE POSSIBILI:

- "Spiegare la differenza tra modello iterativo e incrementale, fornendo elementi per comprendere sia l'uno che l'altro. Descrivere le condizioni al contorno che rendano preferibile l'uno o l'altro"
- "Sviluppo incrementale vs agile: differenze, somiglianze, quando preferire uno o l'altro"

TEMPLATE RISPOSTA (400-600 parole):

```
SVILUPPO INCREMENTALE

Caratteristiche:
- Costruzione per AGGIUNTE successive
- Ogni incremento AGGIUNGE funzionalità
- Mai rimuovere o rifare quanto già fatto
- Pianificazione incrementi iniziale

Processo:
1. Definizione requisiti completi all'inizio
```

2. Divisione requisiti in incrementi
3. Implementazione incremento 1 (subset funzionalità)
4. Rilascio incremento 1
5. Implementazione incremento 2 (aggiunge a incremento 1)
6. Rilascio incremento 2
... fino a prodotto completo

Vantaggi:
- Feedback anticipato su parte funzionante
- Rilasci graduali, rischio distribuito
- Valore consegnato progressivamente
- Riduzione big bang integration

Svantaggi:
- Difficile cambiare architettura iniziale
- Requisiti devono essere stabili
- Vincoli architetturali iniziali rigidi

SVILUPPO ITERATIVO

Caratteristiche:
- Costruzione per RAFFINAMENTI successivi
- Ogni iterazione PUÒ RIFARE/MODIFICARE precedente
- Iterazioni possono essere DISTRUTTIVE
- Pianificazione adattiva

Processo:
1. Definizione requisiti iniziali (anche parziali)
2. Implementazione versione 1 (prototipo/PoC)
3. Valutazione con stakeholder
4. MODIFICA/RIFATTORIZZAZIONE versione 1 → versione 2
5. Valutazione versione 2
6. MODIFICA versione 2 → versione 3
... fino a soddisfazione requisiti

Vantaggi:
- Adattabilità a requisiti emergenti
- Possibilità correzione errori architetturali
- Apprendimento progressivo dominio
- Riduzione rischio requisiti sbagliati

Svantaggi:
- Possibile rilavorazione (waste)
- Difficile stimare effort totale
- Rischio refactoring continuo senza avanzamento

SVILUPPO AGILE (Iterativo + Incrementale)

Agile combina entrambi:
- INCREMENTALE: ogni sprint aggiunge funzionalità
- ITERATIVO: ammette refactoring e cambiamenti

Caratteristiche distintive:
- Sprint brevi (1-4 settimane)
- Iterazioni potenzialmente distruttive
- Feedback continuo stakeholder
- Requisiti emergenti accettati

Differenza con incrementale puro:
"Entrambi guardano nella stessa direzione, ma lo sviluppo
agile contempla il rischio di iterazioni distruttive, che
invece lo sviluppo incrementale cerca in ogni modo di evitare."

CONDIZIONI PREFERENZA

PREFERIRE INCREMENTALE quando:
- Requisiti STABILI e ben compresi
- Architettura chiara dall'inizio
- Basso rischio cambiamenti
- Necessità rilasci graduali prevedibili
- Team esperti nel dominio

Esempio: Sistema gestionale consolidato, specifiche IEEE

PREFERIRE ITERATIVO quando:
- Requisiti INCERTI o emergenti
- Dominio poco conosciuto
- Alto rischio requisiti sbagliati
- Necessità esplorare soluzioni
- Prototipazione e PoC

Esempio: Startup innovativa, prodotto nuovo mercato

PREFERIRE AGILE quando:
- Requisiti variabili (mercato dinamico)
- Feedback frequente stakeholder critico
- Team piccoli coesi
- Possibilità refactoring continuo
- Time-to-market importante

Esempio: Webapp consumer, ambiente competitivo

SCONSIGLIATO ITERATIVO quando:
- Contratti a prezzo fisso
- Deadline rigide
- Budget limitato (rischio overrun)
- Dominio safety-critical (costi refactoring alti)

```
IBRIDAZIONI PRATICHE

Molti progetti combinano approcci:
- Incrementale per architettura core (stabile)
- Iterativo per features innovative (sperimentali)
- Agile per sviluppo sprint
- Waterfall per documentazione contrattuale

Nel progetto didattico:
- TB: approccio iterativo (PoC, esplorare)
- PB: approccio incrementale (costruire su base verificata)
- Sprint interni: pratiche agile (standup, retrospettive)
```

TIPO 5: Processi ISO/IEC 12207 (40% probabilità)

FORMULE POSSIBILI:

- "Fissando l'attenzione sulla definizione di processo associata allo standard ISO/IEC 12207, indicare quali processi sia possibile e opportuno istanziare su un progetto di taglia analoga al progetto didattico, e con quale istanziazione concreta"

TEMPLATE RISPOSTA (400-600 parole):

```
DEFINIZIONE PROCESSO (ISO/IEC 12207)

"Insieme di attività correlate e coese che trasformano ingressi
in uscite"

Caratteristiche processo di qualità:
- EFFICACE: raggiunge obiettivi prefissati
- EFFICIENTE: usa minimo risorse necessarie
- MISURABILE: metriche quantitative definite


PROCESSI PRIMARI (ciclo di vita prodotto)

1. ACQUISIZIONE
Istanziazione progetto didattico:
- Studio capitolato d'appalto
- Valutazione fattibilità tecnica/economica
- Definizione vincoli contrattuali
- Output: Scelta capitolato, impegni assunti

2. FORNITURA
```

```
Istanziazione:
- Pianificazione progetto (tempi, costi, risorse)
- Gestione baseline e milestone (TB, PB, CA)
- Consegna prodotto finale
- Output: Prodotto + documentazione

3. SVILUPPO (core)
Sottoprocessi istanziati:

3.1 Analisi Requisiti
- Acquisizione bisogni stakeholder
- Classificazione requisiti (RF, RNF, RV)
- Tracciamento requisiti
- Output: Analisi dei Requisiti v1.0.0 (TB), v2.0.0 (PB)

3.2 Progettazione Architetturale
- Definizione architettura logica
- Diagrammi UML (classi, sequenza, attività)
- Scelta design pattern
- Output: Specifica Tecnica v1.0.0 (TB), v2.0.0 (PB)

3.3 Progettazione Dettaglio
- Design componenti singoli
- Interfacce e API
- Diagrammi dettagliati
- Output: Manuali tecnici

3.4 Codifica
- Implementazione codice
- Rispetto standard codifica
- Code review
- Output: Codice sorgente

3.5 Integrazione
- Composizione componenti
- Test integrazione
- Build automatizzato
- Output: Sistema integrato

3.6 Test Qualifica
- Test sistema
- Verifica requisiti
- Collaudo
- Output: Sistema validato

4. MANUTENZIONE
Istanziazione (limitata progetto didattico):
- Correzione bug post-PB
- Miglioramenti pre-CA
- Output: Versioni successive
```

PROCESSI SUPPORTO

5. DOCUMENTAZIONE
Istanziazione:
- Norme di Progetto: regole team
- Piano di Progetto: pianificazione
- Piano di Qualifica: strategie V&V
- Glossario: terminologia
- Manuali utente/sviluppatore
Output: Documenti di processo e prodotto

6. GESTIONE CONFIGURAZIONE
Istanziazione:
- Git + GitHub/GitLab
- Versioning semantico (X.Y.Z)
- Branch strategy (gitflow)
- Tag per baseline (TB, PB)
Output: Repository configurato

7. VERIFICA
Istanziazione:
- Inspection documenti (checklist)
- Test automatizzati (unit, integration, E2E)
- Code review
- Verifica metriche qualità
Output: Rapporti verifica, test reports

8. VALIDAZIONE
Istanziazione:
- Collaudo con proponente
- Dimostrazione scenari d'uso
- Accettazione formale
Output: Verbale collaudo

9. REVISIONE CONGIUNTA
Istanziazione:
- Revisioni RTB, PB, CA con proponente/committente
- Presentazione stato avanzamento
- Feedback e modifiche
Output: Verbali revisione

10. GESTIONE QUALITÀ
Istanziazione:
- Definizione metriche (Piano di Qualifica)
- Monitoraggio cruscotto qualità
- Azioni correttive
Output: Dashboard metriche, report qualità

```
PROCESSI ORGANIZZATIVI

11. GESTIONE PROGETTO
Istanziazione:
- Pianificazione sprint/periodi
- Allocazione risorse
- Tracking avanzamento
- Gestione rischi
Output: Gantt, burndown, risk register

12. GESTIONE INFRASTRUTTURA
Istanziazione:
- Setup ambiente sviluppo
- CI/CD pipeline
- Server deployment
Output: Infrastruttura configurata

13. MIGLIORAMENTO PROCESSO
Istanziazione:
- Retrospettive sprint
- Analisi problemi ricorrenti
- Adattamento Norme di Progetto
Output: Norme aggiornate, lesson learned


ISTANZIAZIONE CONCRETA PROGETTO DIDATTICO

Processi prioritari (effort 80%):
- Sviluppo (Analisi, Progettazione, Codifica, Test): 50%
- Verifica (inspection, test): 20%
- Documentazione: 15%
- Gestione Configurazione: 10%
- Altri: 5%

Processi ridotti/adattati:
- Acquisizione: semplificato (no contratti legali)
- Manutenzione: limitato post-PB
- Formazione: peer learning invece di formale

Flusso dipendenze:
Acquisizione → Fornitura → Sviluppo (con Verifica continua)
→ Validazione → Consegna
```

TIPO 6: Metriche Qualità (30% probabilità)

FORMULE POSSIBILI:

- "Presentare almeno due metriche significative per la misurazione della qualità della progettazione software e del codice. Specificare quale sia stata utilizzata nel proprio progetto didattico e con quale esito"
- "Metriche di qualità per progettazione e codice: obiettivi, criteri di valorizzazione, possibilità di automazione"

TEMPLATE RISPOSTA (400-600 parole):

```
METRICHE PROGETTAZIONE SOFTWARE

1. INSTABILITÀ (I)
Formula: I = Ce / (Ca + Ce)
Dove:
- Ce (Efferent Coupling): dipendenze in uscita
- Ca (Afferent Coupling): dipendenze in ingresso

Obiettivo:
- Misurare stabilità componente rispetto a cambiamenti
- Valutare bilanciamento responsabilità

Criteri di valorizzazione:
- I ≈ 0: componente STABILE (dipende da molti, dipende da pochi)
- I ≈ 1: componente INSTABILE (dipende da pochi, dipende da molti)
- Ideale: componenti core stabili (I basso), componenti leaf instabili

Possibilità automazione:
- Tool: SonarQube, NDepend, JDepend
- Analisi statica codice
- Report automatici in CI/CD

Applicazione progetto:
"Nel nostro progetto, componente DatabaseLayer aveva I=0.2
(stabile, usato da molti componenti business logic), mentre
UIComponents aveva I=0.8 (instabile, dipende da molti ma non
usato da altri). Questo era desiderabile architetturalmente."

2. ACCOPPIAMENTO (Coupling)
Tipi misurabili:
- CBO (Coupling Between Objects): numero classi accoppiate
- RFC (Response For Class): numero metodi invocabili

Obiettivo:
- Minimizzare dipendenze tra componenti
- Favorire modularità e manutenibilità

Criteri di valorizzazione:
- CBO basso (< 5-7): buona modularità
- CBO alto (> 15): accoppiamento eccessivo, refactoring necessario
```

Possibilità automazione:
- Analizzatori statici (PMD, Checkstyle)
- Metriche Eclipse, IntelliJ
- Gate quality in pipeline

Applicazione progetto:
"Abbiamo monitorato CBO con SonarQube. Classi Controller
avevano CBO=8-12 (accettabile per coordinamento), mentre
Entities avevano CBO=2-4 (bene, oggetti dati isolati)."

3. COESIONE (Cohesion)
Metrica: LCOM (Lack of Cohesion of Methods)
Formula: LCOM = (P - Q) se P > Q, altrimenti 0
Dove:
- P: coppie metodi che NON condividono attributi
- Q: coppie metodi che condividono attributi

Obiettivo:
- Misurare quanto metodi di una classe siano correlati
- Alta coesione = responsabilità ben definita

Criteri di valorizzazione:
- LCOM = 0: coesione perfetta (tutti metodi usano attributi comuni)
- LCOM alto: classe fa troppe cose (viola SRP)

Possibilità automazione:
- SonarQube LCOM4
- Metriche IDE
- Threshold-based alerting

Applicazione progetto:
"Classe UserService inizialmente aveva LCOM=45 (faceva
autenticazione + profilo + notifiche). Dopo refactoring in
3 classi separate, LCOM sceso a 5-8 per classe."

METRICHE QUALITÀ CODICE

4. COPERTURA CODICE (Code Coverage)
Tipi:
- Statement Coverage: % righe eseguite
- Branch Coverage: % branch if/switch coperti
- Path Coverage: % percorsi esecuzione

Obiettivo:
- Valutare adeguatezza test
- Identificare codice non testato

Criteri di valorizzazione:

- Minimo: 70% statement coverage
- Target: 80-85% statement + 70% branch
- Evitare: 100% coverage fine a sé (test inutili)

Possibilità automazione:
- Tool: JaCoCo, Istanbul, Coverage.py
- Report HTML interattivi
- Gate quality (fail build se < threshold)

Applicazione progetto:
"Obiettivo 80% statement coverage. Raggiunto 83% overall,
con core business logic a 92%. UI layer a 65% (test E2E
manuali non contati). Gate in CI: fail se < 75%."

5. COMPLESSITÀ CICLOMATICA
Formula: M = E - N + 2P
Dove:
- E: archi control flow
- N: nodi
- P: componenti connesse

Obiettivo:
- Misurare complessità logica metodo
- Identificare hot spot manutenzione

Criteri di valorizzazione:
- M ≤ 10: complessità accettabile
- 10 < M ≤ 20: complessità moderata, monitorare
- M > 20: refactoring necessario

Possibilità automazione:
- Analisi statica (SonarQube, Radon, Lizard)
- Report per metodo/classe
- Trend analysis

Applicazione progetto:
"Metodo validateInput aveva M=24 (troppo complesso).
Refactoring in sotto-metodi validateEmail, validatePassword,
validatePhone ha ridotto a M=8, 6, 7 rispettivamente."

6. DUPLICAZIONE CODICE
Metrica: % linee duplicate
Soglie:
- < 3%: ottimo
- 3-5%: accettabile
- > 5%: problematico

Obiettivo:
- Identificare violazioni DRY
- Favorire riuso

```
Possibilità automazione:
- CPD (Copy-Paste Detector) di PMD
- SonarQube duplication
- Automatic refactoring suggestions

Applicazione progetto:
"Inizialmente 7% duplicazione (form validation ripetuta).
Estratto ValidationUtils, ridotto a 2.8%."


SCELTA METRICHE PROGETTO

Priorità:
1. Code Coverage (fondamentale, facile automatizzare)
2. Complessità ciclomatica (previene debito tecnico)
3. Duplicazione (rapido fix, alto ROI)
4. Coupling/Cohesion (design quality, refactoring costoso)

Dashboard qualità:
- SonarQube integrato in CI/CD
- Metriche monitorate ad ogni commit
- Quality gate pre-merge: coverage ≥ 75%, duplicazione ≤ 3%,
  complessità ≤ 15

Esito:
"Il monitoring continuo ha permesso identificare early debt
tecnico. Refactoring guidato da metriche ha migliorato
maintainability index da C (fair) a A (excellent)."
```

TIPO 6: GANTT vs PERT (30% probabilità)

FORMULE POSSIBILI:

- "Discutere le differenze informative tra i diagrammi PERT e Gantt"
- "Fornire definizione del diagramma di Gantt, discutere finalità, modalità d'uso, efficacia e punti deboli"
- "Diagrammi di supporto alla pianificazione: PERT e Gantt"

TEMPLATE RISPOSTA (400-600 parole):

```
DIAGRAMMA DI GANTT

Definizione:
Strumento di supporto alla gestione progetti che mostra le attività
come barre orizzontali su asse temporale. Inventato da Henry L. Gantt
```

(1917).

Struttura:
- Asse orizzontale: tempo (giorni/settimane/mesi)
- Asse verticale: attività di progetto
- Barre orizzontali: durata e sequenza attività
- Possibile sovrapposizione barre (pianificato vs effettivo)

Informazioni visualizzate:
1. DURATA attività (lunghezza barra)
2. SEQUENZIALITÀ (attività consecutive)
3. PARALLELISMO (attività sovrapposte temporalmente)
4. AVANZAMENTO (confronto pianificato/effettivo)
5. RESPONSABILI (chi fa cosa)

Vantaggi:
- Lettura immediata stato avanzamento
- Comprensibile a stakeholder non tecnici
- Visualizzazione intuitiva sovrapposizioni
- Facile aggiornamento manuale

Limiti:
- NON gestisce dipendenze tra attività
- Non mostra cammino critico
- Non evidenzia slack time
- Difficile capire impatto ritardi su altre attività

DIAGRAMMA PERT

Definizione:
Program Evaluation and Review Technique - metodo statistico per determinare tempi attività mediante diagrammi reticolari (grafi).

Struttura:
- Nodi: eventi/milestone (inizio/fine attività)
- Archi: attività (consumano tempo e risorse)
- Grafo orientato con dipendenze temporali

Informazioni visualizzate:
1. DIPENDENZE temporali tra attività
2. CAMMINO CRITICO (sequenza attività più lunga)
3. SLACK TIME (margine temporale per ogni attività)
4. DATA MINIMA evento (earliest)
5. DATA MASSIMA evento (latest)

Calcoli associati:
- Stima a 3 valori: ottimistico, probabile, pessimistico
- Tempo atteso = (ottimistico + 4×probabile + pessimistico) / 6
- Cammino critico: attività con slack = 0

Vantaggi:
- Gestisce esplicitamente dipendenze
- Identifica attività critiche
- Permette analisi what-if (impatto ritardi)
- Supporta ottimizzazione allocazione risorse

Limiti:
- Complessità maggiore vs Gantt
- Meno intuitivo per non esperti
- Necessita aggiornamento attento dipendenze

DIFFERENZE INFORMATIVE

Aspetto	GANTT	PERT
Focus	Durata e sequenza	Dipendenze e criticità
Visualizzazione	Barre su timeline	Grafo reticolare
Dipendenze	Non gestite	Esplicite (archi)
Cammino critico	Non visibile	Identificato
Slack time	Non calcolato	Calcolato
Lettura	Immediata	Richiede analisi
Utenti	Tutti stakeholder	Project manager
Aggiornamento	Semplice	Complesso (ripropagazione)

GANTT risponde a: "Quando si fa cosa?"
PERT risponde a: "Cosa dipende da cosa? Dove non posso ritardare?"

RELAZIONE TRA I DUE

Non sono alternativi ma complementari:
- PERT per PIANIFICAZIONE (identificare criticità, ottimizzare)
- GANTT per COMUNICAZIONE (mostrare avanzamento, coordinare)

Nel progetto didattico:
"Abbiamo utilizzato PERT in fase iniziale per identificare
cammino critico (Analisi Requisiti → Progettazione → Codifica
componente X), calcolando che ritardi > 3 giorni su Analisi
avrebbero impattato milestone RTB. Successivamente abbiamo
usato Gantt settimanalmente per monitorare avanzamento e
comunicare stato a team e proponente."

STRUMENTI COMPLEMENTARI

Spesso si usa anche:
- WBS (Work Breakdown Structure): scompone attività gerarchicamente
- Burndown chart: in contesti agile, mostra lavoro rimanente

```
- Kanban board: visualizza flusso work items

Sequenza tipica:
1. WBS: scomporre progetto in attività
2. PERT: analizzare dipendenze, identificare criticità
3. GANTT: pianificare timeline, comunicare
4. GANTT aggiornato: tracking continuo


EFFICACIA E LIMITI (esperienza progetto)

Gantt - Punti di forza:
- Usato settimanalmente per standup, allineamento veloce
- Stakeholder capivano immediatamente stato
- Facile spot parallelismi migliorabili

Gantt - Limiti riscontrati:
- Dipendenza "Analisi → Progettazione" non esplicita
- Ritardo Analisi non mostrava impatto cascata
- Dovevamo spiegare verbalmente criticità

PERT - Punti di forza:
- Identificato subito che "Setup CI/CD" bloccava "Test Integrazione"
- Slack positivo su "Documentazione Manuale Utente" usato per buffer

PERT - Limiti riscontrati:
- Aggiornamento oneroso (modifiche dipendenze frequenti)
- Team junior faticava a leggere grafo
- Abbandonato dopo PB per overhead


POSSIBILI MIGLIORAMENTI

Per progetti futuri:
- Tool integrato (Microsoft Project, Jira) che sincronizza Gantt-PERT
- PERT automatico da task dependencies
- Dashboard combina: Gantt (timeline) + heatmap criticità
- Gantt interattivo mostra dipendenze al click
```

TIPO 7: INSPECTION vs WALKTHROUGH (25% probabilità)

FORMULE POSSIBILI:

- "Illustrare differenze per obiettivi e modalità di svolgimento delle tecniche di inspection e walkthrough"
- "Confrontare inspection e walkthrough come tecniche di verifica statica"

TEMPLATE RISPOSTA (400-500 parole):

```
INSPECTION

Definizione:
Tecnica di verifica statica (lettura codice senza esecuzione) con
approccio MIRATO basato su presupposti di errore.

Obiettivi:
- Rivelare presenza di difetti specifici
- Eseguire lettura MIRATA del codice
- Ricerca focalizzata su presupposti (error-prone patterns)

Agenti:
- Verificatori DISTINTI e SEPARATI da programmatori
- Ruoli: moderatore, lettore, registratore
- Fronte contrapposto: chi verifica ≠ chi sviluppa

Strategia:
Focalizzare ricerca su PRESUPPOSTI di errore:
- Checklist difetti comuni (es. null pointer, buffer overflow)
- Pattern noti problematici (es. concorrenza, memory leak)
- Standard violabili (coding conventions)

Fasi inspection:
1. PIANIFICAZIONE:
   - Selezione materiale da verificare
   - Preparazione checklist
   - Assegnazione ruoli

2. DEFINIZIONE LISTA CONTROLLO:
   - Checklist difetti tipici dominio
   - Presupposti basati su esperienze passate

3. LETTURA CODICE:
   - Individuale, preparazione anticipata
   - Focus su punti checklist
   - Annotazione difetti trovati

4. MEETING INSPECTION:
   - Presentazione difetti trovati
   - Discussione focalizzata
   - NO risoluzione immediata (solo identificazione)

5. CORREZIONE DIFETTI:
   - Programmatore corregge
   - Follow-up verifica correzioni

6. DOCUMENTAZIONE:
```

   - Rapporto formale difetti
   - Metriche (difetti/KLOC, tempo)

Caratteristiche:
- RAPIDO (focus mirato)
- RIPETIBILE (checklist standardizzata)
- MISURABILE (metriche difetti)

WALKTHROUGH

Definizione:
Tecnica di verifica statica con approccio AMPIO basato
su simulazione esecuzione.

Obiettivi:
- Rivelare difetti di ogni tipo (largo spettro)
- Eseguire lettura CRITICA del codice
- Simulare esecuzioni possibili

Agenti:
- Gruppi MISTI ispettori/sviluppatori
- Ruoli DISTINTI ma collaborativi
- Più cooperativo che contrapposto

Strategia:
Percorrere codice SIMULANDO esecuzioni:
- Tracciare flussi dati
- Provare casi d'uso
- Esplorare branch alternativi

Fasi walkthrough:
1. PIANIFICAZIONE:
   - Selezione codice
   - Preparazione scenari test
   - Convocazione partecipanti

2. LETTURA CODICE:
   - Collettiva, durante meeting
   - Programmatore guida attraverso codice
   - Simulazione "dry run" manuale

3. DISCUSSIONE:
   - Apertura totale (qualunque difetto)
   - Brainstorming problemi potenziali
   - Dialogo costruttivo

4. CORREZIONE DIFETTI:
   - Discussione possibili fix
   - Programmatore implementa

5. DOCUMENTAZIONE:
   - Rapporto informale/formale
   - Lessons learned

Caratteristiche:
- PIÙ LENTO (esplorazione completa)
- COLLABORATIVO (team misto)
- EDUCATIVO (condivisione conoscenza)

CONFRONTO DIRETTO

AFFINITÀ:
- Entrambi controlli STATICI (no esecuzione)
- Entrambi basati su desk check
- Programmatori e verificatori esistono in entrambi
- Documentazione formale risultati

DIFFERENZE:

Aspetto	INSPECTION	WALKTHROUGH
Approccio	Mirato (presupposti)	Ampio (esplorativo)
Base	Checklist predefinita	Esperienza collettiva
Agenti	Separati (verificatori ≠ sviluppatori)	Misti (team unico)
Modalità	Lettura individuale + meeting	Lettura collettiva durante meeting
Velocità	Più rapido	Più lento
Cooperazione	Contrapposto	Collaborativo
Output	Lista difetti specifica	Discussione ampia
Ripetibilità	Alta (checklist)	Media (dipende esperienza)
Efficacia	Alta su difetti noti	Alta su difetti nuovi

QUANDO PREFERIRE

INSPECTION quando:
- Codice critico (safety, security)
- Pattern errori noti nel dominio
- Tempo limitato, focus efficienza
- Standard da rispettare rigorosamente
- Team esperto con checklist consolidate

WALKTHROUGH quando:
- Codice innovativo (dominio nuovo)
- Necessità condivisione conoscenza
- Team junior (educazione)
- Requisiti complessi da verificare copertura
- Collaborazione sviluppatori-verificatori importante

```
NEL PROGETTO DIDATTICO

"Abbiamo usato INSPECTION per documenti formali (Analisi
Requisiti, Specifica Tecnica) con checklist 15 punti basata
su Norme di Progetto (es. formato riferimenti, completezza
sezioni obbligatorie). Tempo medio: 20 min/documento.

Abbiamo usato WALKTHROUGH per codice componenti core,
dove programmatore presentava architettura e team simulava
scenari edge case. Questo ha rivelato race condition non
prevista in checklist. Tempo medio: 1h/componente.

Efficacia: Inspection più efficiente per difetti formali
(95% trovati), Walkthrough più efficace per difetti logici
complessi."
```

TIPO 8: CONFIGURAZIONE e VERSIONAMENTO (20% probabilità)

FORMULE POSSIBILI:

- "Discutere almeno 2 varianti della nozione di configurazione di prodotto/sistema. Indicare relazione tra configurazione e versionamento"
- "Gestione configurazione: varianti, obiettivi, relazione con versionamento"

TEMPLATE RISPOSTA (400-500 parole):

```
NOZIONE DI CONFIGURAZIONE

Definizione generale:
Insieme coerente e consistente di componenti (file sorgenti,
eseguibili, librerie, documenti) che costituiscono una versione
specifica di un sistema software.


VARIANTI CONFIGURAZIONE

1. CONFIGURAZIONE DI SVILUPPO (Development Configuration)

Caratteristiche:
- Include codice sorgente, script build, dipendenze sviluppo
- Versionata in repository (Git)
- Potenzialmente instabile
- Componenti: .java/.cpp, package.json, Makefile, test
```

Obiettivo:
- Permettere sviluppo collaborativo
- Tracciare evoluzione codice
- Supportare integrazione continua

Esempio:
- Branch feature/login con: LoginController.java,
  LoginService.java, login.test.js, dipendenze (JWT lib v2.1)

2. CONFIGURAZIONE DI PRODOTTO (Product/Release Configuration)

Caratteristiche:
- Include SOLO componenti deployment
- Eseguibili compilati, librerie runtime, asset
- Stabile, testata, validata
- Componenti: .jar/.exe, .so/.dll, config files, DB schema

Obiettivo:
- Distribuire versione funzionante
- Garantire riproducibilità deployment
- Supportare rollback se necessario

Esempio:
- Release v2.3.1: app.jar (dipendenza: spring-boot 3.2.0),
  application.yml, schema.sql v15

3. CONFIGURAZIONE DI BASELINE

Caratteristiche:
- Snapshot approvato formalmente
- Associato a milestone progetto
- Immutabile (frozen)
- Tracciabilità requisiti → componenti

Obiettivo:
- Riferimento stabile per verifiche successive
- Punto partenza cambiamenti controllati
- Evidence per revisioni formali

Esempio:
- Baseline RTB (v1.0.0): Analisi Requisiti v1.0.0,
  Specifica Tecnica v1.0.0, PoC commit abc123

RELAZIONE CONFIGURAZIONE - VERSIONAMENTO

Il versionamento è il MECCANISMO per gestire configurazioni:

```
VERSIONAMENTO fornisce:
- Identificazione univoca (v1.2.3, commit hash)
- Storico modifiche (changelog, git log)
- Possibilità recupero versioni precedenti
- Tracciamento dipendenze tra versioni

CONFIGURAZIONE usa versionamento per:
- Assemblare componenti versioni specifiche
- Garantire coerenza (tutte dipendenze versioni compatibili)
- Riprodurre esattamente ambiente passato

Relazione: CONFIGURAZIONE = insieme componenti VERSIONATI

Esempio concreto:
```

Configurazione Release v2.1.0:

- backend.jar v2.1.0 (Git commit: def456)
- frontend-bundle.js v2.0.5 (Git commit: ghi789)
- database-schema v14 (migration: 20240115_add_users)
- nginx.conf v3 (configurazione routing)

```

Ogni componente ha VERSIONE, insieme formano CONFIGURAZIONE.


GESTIONE CONFIGURAZIONE (Configuration Management)

Processo che gestisce:
1. IDENTIFICAZIONE configurazioni (naming, tagging)
2. CONTROLLO modifiche (change control)
3. STATO accounting (tracking versioni)
4. AUDIT/VERIFICA (compliance baseline)

Strumenti:
- VCS (Git, SVN): versionamento sorgenti
- Artifact repository (Nexus, Artifactory): binari versionati
- Configuration management tools (Ansible, Chef): configurazioni infra
- Dependency management (Maven, npm): gestione dipendenze versionate


OBIETTIVI

1. RIPRODUCIBILITÀ:
   Ricreare esattamente configurazione passata

2. TRACCIABILITÀ:
   Sapere quali componenti in quale versione
```

```
3. CONSISTENZA:
   Evitare incompatibilità dipendenze

4. ROLLBACK:
   Tornare a configurazione funzionante

5. PARALLELISMO:
   Multiple configurazioni coesistenti (dev/staging/prod)


NEL PROGETTO DIDATTICO

"Abbiamo gestito 3 tipi configurazioni:

1. SVILUPPO: Branch Git per feature, con package.json
   versioning dipendenze. Ogni sviluppatore aveva
   configurazione locale consistente.

2. BASELINE: Tag Git per milestone (v1.0.0-RTB, v2.0.0-PB)
   referenziando commit documenti + codice. Configurazione
   frozen presentata a revisioni.

3. PRODOTTO: Release v1.0.0 in Docker container con
   versioni esplicite dipendenze (Node 18.16, PostgreSQL 14.2).

Relazione versionamento: Semantic versioning (X.Y.Z) per
documenti e codice. Configurazione baseline identificata
da tag Git, riproducibile tramite `git checkout v1.0.0-RTB`.

Efficacia: Rollback immediato dopo bug critical (ritorno
a v1.2.3 in 5 minuti). Limite: overhead manutenzione
changelog per ogni componente."
```

TIPO 9: RUOLI DI PROGETTO (15% probabilità)

FORMULE POSSIBILI:

- "Descrivere i ruoli di progetto e le loro responsabilità"
- "Ruoli in un progetto software: responsabile, amministratore, analista, progettista, programmatore, verificatore"

TEMPLATE RISPOSTA (300-400 parole):

```
RUOLI DI PROGETTO
```

ISO/IEC 12207 non definisce ruoli ma PROCESSI. I ruoli sono
istanziazione organizzativa di responsabilità sui processi.

1. RESPONSABILE DI PROGETTO (Project Manager)

Responsabilità:
- Pianificazione attività e allocazione risorse
- Gestione timeline, budget, rischi
- Coordinamento team
- Interfaccia con committente/proponente
- Approvazione decisioni strategiche

Processi gestiti:
- Gestione progetto (planning, tracking)
- Gestione rischi
- Gestione comunicazioni stakeholder

Criticità:
- Accentratore decisioni, richiede visione d'insieme
- Responsabile successo/fallimento progetto

2. AMMINISTRATORE (Administrator)

Responsabilità:
- Gestione infrastruttura sviluppo
- Configurazione repository, CI/CD
- Controllo versioni e baseline
- Gestione documentazione formale
- Setup ambienti (dev/staging/prod)

Processi gestiti:
- Gestione configurazione
- Gestione infrastruttura
- Documentazione

Criticità:
- Garanzia riproducibilità, tracciabilità
- Setup iniziale determina efficienza futura

3. ANALISTA (Analyst)

Responsabilità:
- Acquisizione e analisi requisiti
- Classificazione requisiti (RF, RNF, RV)
- Tracciamento requisiti
- Interfaccia con stakeholder per chiarimenti
- Validazione copertura needs

Processi gestiti:
- Analisi requisiti
- Studio fattibilità
- Gestione requisiti

Criticità:
- Comprensione dominio fondamentale
- Errori analisi propagano a tutte fasi successive

4. PROGETTISTA (Designer/Architect)

Responsabilità:
- Progettazione architettura software
- Scelta design pattern
- Definizione interfacce componenti
- Documentazione scelte tecniche
- Technology Baseline

Processi gestiti:
- Progettazione architetturale
- Progettazione dettagliata

Criticità:
- Decisioni architetturali difficilmente reversibili
- Bilanciamento requisiti funzionali/non funzionali/vincoli

5. PROGRAMMATORE (Programmer/Developer)

Responsabilità:
- Codifica componenti
- Unit testing
- Rispetto standard codifica
- Documentazione codice (commenti, API doc)
- Debug

Processi gestiti:
- Codifica
- Test unità (auto-verifica)

Criticità:
- Qualità codice impatta manutenibilità
- Maggior numero persone nel team

6. VERIFICATORE (Verifier/Quality Assurance)

Responsabilità:

- Verifica documenti (inspection/walkthrough)
- Test integrazione e sistema
- Controllo conformità standard
- Validazione baseline
- Approvazione prodotti intermedi

Processi gestiti:
- Verifica
- Validazione (con committente)
- Controllo qualità

Criticità:
- DEVE essere indipendente da chi produce
- Garanzia qualità, gateway per milestone

SOVRAPPOSIZIONE RUOLI

Progetto grande: persone dedicate per ruolo
Progetto piccolo (es. didattico): stessa persona più ruoli

VINCOLO: Verificatore SEMPRE separato da programmatore
(conflitto interessi, perdita oggettività).

Possibili sovrapposizioni accettabili:
- Analista + Progettista (continuità requisiti → design)
- Progettista + Programmatore (comprensione architettura)
- Responsabile + Amministratore (coordinamento)

Sovrapposizioni VIETATE:
- Programmatore + Verificatore proprio codice

NEL PROGETTO DIDATTICO

"Team 6 persone, ruoli rotati per sprint:

- Responsabile: pianificazione Gantt, risk management
- Amministratore: Git branching strategy, CI/CD GitHub Actions
- Analista (2 persone): interviste proponente, Analisi Requisiti
- Progettista: architettura a layer, scelta pattern (MVC, Strategy)
- Programmatore (tutti): codifica componenti assegnati
- Verificatore (rotazione): mai verifica proprio codice

Rotazione ruoli ogni 2 settimane permetteva:
- Esperienza completa ciclo vita
- Riduzione bottleneck (no single point of failure)

Limite: overhead rotazione (curva apprendimento ogni cambio)"

CHECKLIST PRE-CONSEGNA

Prima di consegnare la risposta:

- [ ] Ho risposto A TUTTE le parti della domanda?
- [ ] Ho usato terminologia ISO/IEC 12207 precisa?
- [ ] Ho fornito definizioni esplicite quando richiesto?
- [ ] Ho incluso esempi concreti dal progetto didattico?
- [ ] Ho analizzato criticamente (pregi E difetti)?
- [ ] La risposta è strutturata in paragrafi chiari?
- [ ] Ho usato elenchi puntati per chiarezza quando opportuno?
- [ ] Lunghezza adeguata (non troppo breve, non oltre 1 pagina)?

ERRORI FATALI DA EVITARE

1. **Rispondere solo parzialmente**: Se chiede "obiettivi, tecniche e strategie", devi coprire TUTTE E TRE
2. **Non citare ISO/IEC 12207**: Quando richiesto, è OBBLIGATORIO
3. **Essere vago**: "Abbiamo fatto dei test" → ❌ | "Test unità con JUnit, 82% coverage, CI/CD" → ✓
4. **Non fare analisi critica**: Tullio vuole pregi E difetti, non solo elenco
5. **Dimenticare esperienza progetto**: Quando chiede "nel vostro progetto", devi parlare del TUO progetto specifico

---

STRATEGIA TEMPORALE

20-25 minuti per domanda:

- 3 min: Leggere domanda 2 volte, identificare parti richieste
- 2 min: Schema mentale risposta (elenco punti da coprire)
- 12-15 min: Scrivere risposta strutturata
- 3-5 min: Rilettura, verifica copertura parti, correzioni

Se finisci prima:

- Rileggi domanda: ho risposto a TUTTO?
- Aggiungi 1-2 esempi concreti in più
- Verifica terminologia precisa

Se sei indietro:

- Priorità: coprire TUTTE le parti richieste (anche sinteticamente)
- Meglio risposta completa breve che risposta incompleta lunga
- Salta elaborazioni, vai al sodo

LINGUAGGIO E STILE

Usa:

- Terminologia tecnica precisa (verifica, validazione, baseline, milestone)
- Struttura paragrafi con intestazioni chiare
- Elenchi puntati per chiarezza
- Esempi concreti numerici ("83% coverage" non "alta coverage")

Evita:

- Linguaggio colloquiale ("abbiamo fatto delle robe")
- Ripetizioni ("la verifica... la verifica... la verifica...")
- Vaghezze ("abbastanza buono", "in generale")
- Frasi troppo lunghe (max 2 righe)

Esempio BUONO:

```
La verifica è un processo continuo applicato ad ogni baseline
del progetto. Nel nostro caso, abbiamo istanziato:
- Inspection documenti (checklist 15 punti)
- Test unità automatizzati (JUnit, 83% coverage)
- Test integrazione API (Postman, 45 test)
Questo ha permesso identificare il 78% dei difetti prima della
validazione finale.
```

Esempio CATTIVO:

```
La verifica è una cosa che serve per controllare che va tutto
bene e l'abbiamo fatta nel nostro progetto facendo dei test e
delle revisioni che sono andate abbastanza bene in generale e
ci hanno aiutato.
```

NOTE FINALI

Tullio valuta:

1. **Precisione terminologica** (30%): Usi termini ISO/SEMAT corretti?
2. **Completezza risposta** (30%): Copri tutte le parti richieste?
3. **Profondità analisi** (20%): Vai oltre l'elenco, analizzi criticamente?
4. **Esperienza concreta** (20%): Esempi reali dal progetto?

Non valuta:

- Calligrafia/estetica
- Lunghezza assoluta (conta completezza)
- Opinioni personali non motivate

In sintesi: Tullio vuole dimostrare che COMPRENDI profondamente concetti ISO 12207 e SAI APPLICARLI criticamente. Non basta conoscere, devi RAGIONARE.