Luca Cabibbo

Architettura dei Sistemi Software

Verificabilità

dispensa asw260

ottobre 2023

Testing leads to failure,
and failure leads to understanding.

Burt Rutan

- Riferimenti

- Luca Cabibbo. **Architettura del Software**: Strutture e Qualità. Edizioni Efesto, 2021.
  - Capitolo 12, Verificabilità
- Bass, L., Weber, I., and Zhu, L. **DevOps**: A Software Architect's Perspective. Addison-Wesley, 2015.
- Humble, J. and Farley, D. **Continuous Delivery**: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, 2010.
- Newman, S. **Building Microservices**: Designing Fine-Grained Systems. O'Reilly, 2015.
- Richardson, C. **Microservices Patterns**: With examples in Java. Manning, 2019.

- Obiettivi e argomenti

- Obiettivi
  - presentare la qualità della verificabilità
  - presentare alcune modalità di verifica dei sistemi software
  - illustrare alcune tattiche per la progettazione per la verificabilità
- Argomenti
  - verificabilità
  - progettare per la verificabilità
  - discussione

* Verificabilità

- **Verifica** (*testing*) del software
  - è un'investigazione che viene eseguita per ottenere informazioni sulla qualità di un sistema o componente software
  - viene svolta mediante l'esecuzione di test del sistema o componente, per valutare una o più proprietà (funzionali e di qualità) di interesse
  - non consente di verificare se il software è "corretto"
    - "program testing can be used to show the presence of bugs, but never to show their absence!" [Edsger W. Dijkstra]

Verificabilità

- **Verificabilità** (*testability*)
  - la facilità con cui è possibile definire dei test per dimostrare la presenza di guasti software (errori software) nel sistema
  - misura la probabilità che il sistema software fallisca durante la prossima esecuzione dei test – nell'ipotesi che nel software ci sia almeno un guasto/errore
  - dunque, un sistema software è verificabile se rivela facilmente i suoi guasti

Verificabilità

- La verificabilità è importante – anche perché ha impatto su altre qualità del software
  - i test sostengono l'affidabilità, la disponibilità, la sicurezza (safety) e la sicurezza informatica (security) – e più in generale la riduzione dei rischi
  - i test consentono di ottenere un feedback sul sistema software in corso di sviluppo – il team di sviluppo può poi rispondere a questo feedback per migliorare il software e il sostegno alle qualità del software

Verificabilità

- La verificabilità è importante – anche perché ha impatto su altre qualità del software
  - il testing richiede una buona percentuale del costo complessivo di sviluppo di un buon sistema software – spesso tra il 30% e il 50% – che è opportuno cercare di ridurre
  - malgrado lo sforzo (e il costo) richiesto per la loro scrittura, i test automatizzati possono consentire dei risparmi, prevenendo problemi di affidabilità e di disponibilità
  - la mancanza di un insieme opportuno di test automatizzati costituisce un “debito tecnologico” di un sistema software

Verifica e validazione

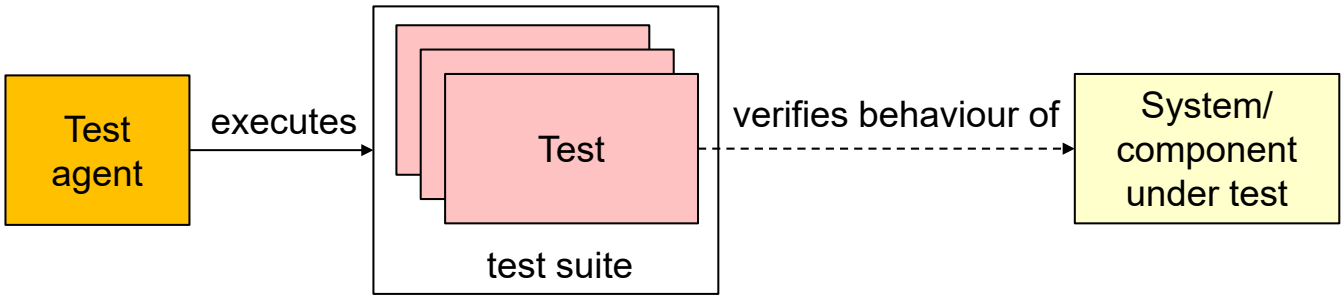
- La *verifica* e la *validazione* sono due attività per “controllare” un prodotto, servizio o sistema
  - la **verifica** (*verification*) riguarda la valutazione della conformità di un prodotto, servizio o sistema con i requisiti, le specifiche o delle condizioni imposte
  - la **validazione** (*validation*) riguarda l’assicurarsi che il prodotto, servizio o sistema corrisponda ai bisogni degli utenti o delle altre parti interessate
- Si tratta di nozioni correlate ma comunque diverse
  - qui ci concentriamo soprattutto sulla verifica

Verifica, test e verificabilità

- La verifica di un sistema software avviene in genere mediante la definizione e l'esecuzione di un ampio insieme di test
  - per progettare per la verificabilità, è necessario conoscere le possibili tipologie di test e le loro caratteristiche – e poi capire come l'architettura di un sistema software possa sostenere la realizzazione dei test

- Test

- Il test di un sistema è basato sulla definizione e l'esecuzione di un ampio insieme di test individuali
  - ciascun test individuale si basa sull'utilizzo (esecuzione) del sistema o di un suo componente da parte di un agente



  - per ogni test, l'agente fornisce un input prefissato al sistema, osserva l'output prodotto e lo confronta con l'output desiderato
  - il sistema può *passare il test* oppure *non passare il test* (*fallire il test*)

Scenari di test

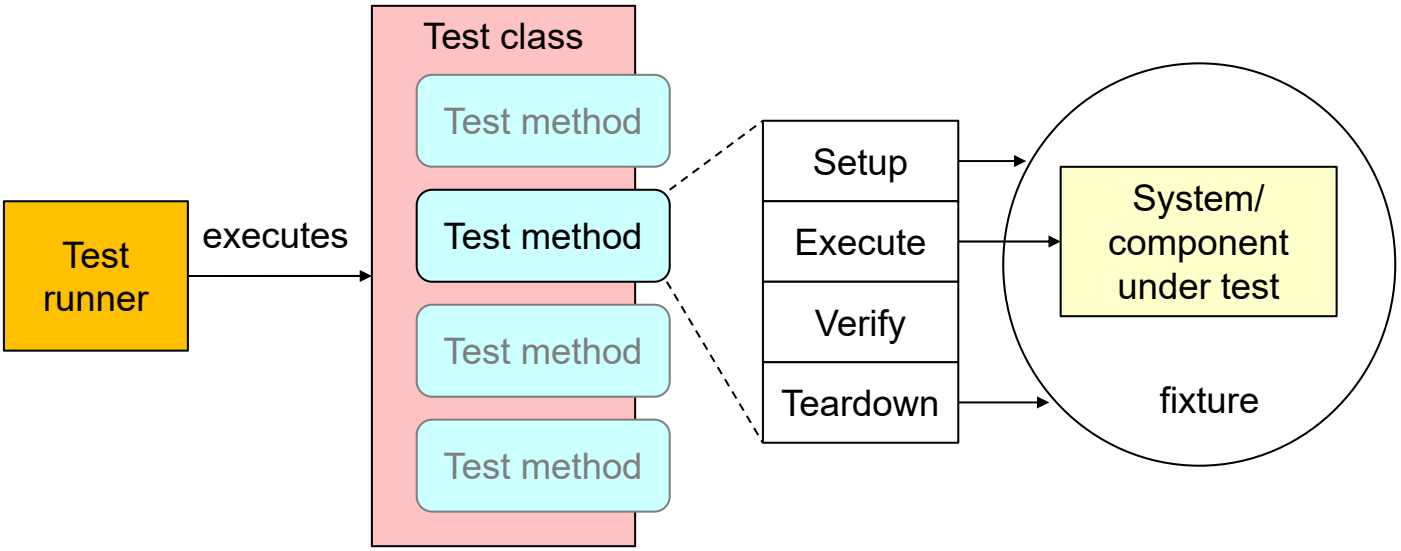
- Ciascun test individuale rappresenta in genere uno scenario di utilizzo per il sistema oppure di un suo componente
  - i diversi test possono riguardare
    - aspetti funzionali e aspetti non funzionali
    - scenari di successo e scenari di fallimento

Test automatizzati e test manuali

- Un'importante classificazione
  - *test automatizzati*
    - basati su degli opportuni programmi di test – *test suite* o *test harness*
  - *test manuali*
    - spesso necessari per verificare ciò che non è stato ancora o che non può essere verificato in modo automatizzato
      - per validare i requisiti di uso del sistema (*showcases*)
      - per identificare altri test che poi andranno effettuati in modo automatizzato (*exploratory testing*)
      - per verificare l'usabilità del sistema (*usability testing*)

- Test automatizzati

- Un test automatizzato ha lo scopo di verificare un singolo caso o scenario di test
  - usando dei framework opportuni, ciascun test viene implementato mediante un metodo di test

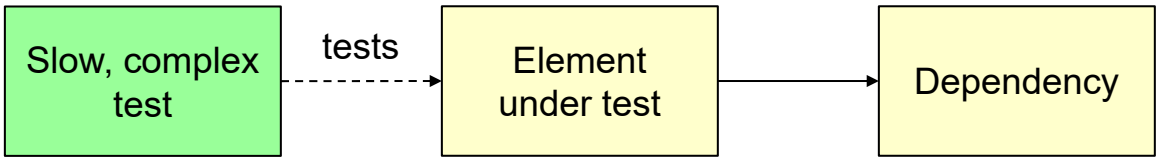


Test automatizzati

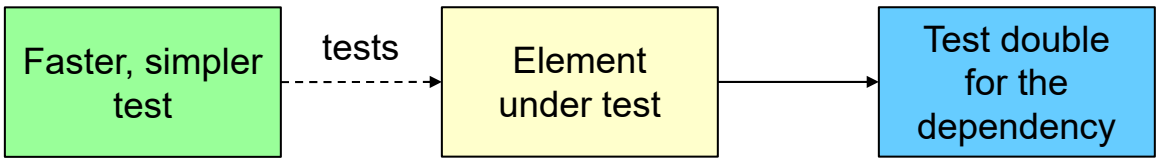
- Alcune osservazioni
  - la definizione di test automatizzati richiede
    - di poter controllare lo stato degli elementi coinvolti nel test (setup e teardown)
    - di poter chiedere all'elemento sotto test di eseguire il comportamento desiderato tramite la sua interfaccia (execute)
    - di poter osservare e ispezionare lo stato dell'elemento da verificare (verify)

- Test socievoli e test solitari

- Alcuni test hanno lo scopo di verificare il funzionamento di singoli elementi – altri servono per verificare le interazioni (all'interfaccia) tra due o più elementi (ad es., A e B)
  - due modalità per definire ed eseguire questi ultimi test
    - *test socievoli*



    - *test solitari*



Test double

- In un test, un *test double* ("controfigura per il test") è un elemento fittizio, realizzato appositamente per il test, che simula una dipendenza dell'elemento da verificare
  - *fake object* – un elemento con un'implementazione parziale o semplificata
  - *stub* – programmato per fornire risposte predefinite
  - *spy* – uno stub che inoltre registra dati sulle chiamate che riceve
  - *mock* – un elemento pre-programmato (durante la preparazione) con riferimento alla specifica delle chiamate attese durante il test – può anche verificare se ha effettivamente ricevuto le chiamate attese, oppure sollevare eccezioni se riceve chiamate inattese

- Tipi di test

- Il test di un sistema software è di solito basato su diversi tipi di test – a granularità diversa e con obiettivi e caratteristiche differenti
  - test unitari
  - test di integrazione
  - test di componenti
  - test end-to-end
  - test di accettazione

- Test unitari

- I *test unitari* hanno lo scopo di verificare il corretto funzionamento di un'"unità" – una piccola parte verificabile del software, dotata di funzionamento autonomo
  - possono essere eseguiti direttamente nell'ambiente di sviluppo (nel PC dello sviluppatore)
  - sono in genere veloci da eseguire
  - vengono usati soprattutto per verificare in isolamento elementi indipendenti
  - possono essere sia solitari che socievoli

Test unitari

- Consideriamo un servizio applicativo S che gestisce i propri dati mediante un repository R e a cui è possibile accedere mediante un adattatore A

  - quali test unitari (solitari e socievoli) sono utili?
  - ad es., si possono verificare il servizio S e l'adattatore A
  - i test unitari non vengono però utilizzati per verificare l'accesso ai servizi infrastrutturali – come l'accesso al database

Test solitari e test double

- Alcune osservazioni
  - i test solitari (basati su stub e mock) hanno lo scopo di verificare alcune dipendenze e relazioni tra elementi diversi
  - i diversi elementi vengono verificati in isolamento – viene anche verificato che un elemento sia in grado di interagire con gli elementi del sistema da cui dipende
  - i test solitari sono sostenuti dall'uso di framework per la creazione di test double

Test solitari

- Alcune osservazioni
  - i test double sostengono diversi benefici – come l'isolamento e la velocizzazione dell'esecuzione del test
  - i test solitari, basati su test double, richiedono di poter configurare i collegamenti (le dipendenze) tra gli elementi del sistema
    - richiedono che gli elementi interagiscano solo con riferimento alle loro interfacce e l'uso di meccanismi di iniezione delle dipendenze

- Test di integrazione

- I *test di integrazione* hanno lo scopo di verificare che un elemento del sistema possa interagire con i servizi infrastrutturali (ad es., il database e il message broker) e con altri servizi applicativi
  - utilizzano dei servizi infrastrutturali reali (come chiamate REST o lo scambio di messaggi) – vanno dunque eseguiti in un ambiente di test opportuno
  - per velocizzarli, spesso si verificano individualmente solo gli "adattatori" del componente di interesse – ma non la logica di business né l'interazione con altri componenti
  - sono importanti nel test di un'architettura – perché hanno lo scopo di verificare le interazioni tra elementi architetturali, alla loro interfaccia

Test di integrazione

- Alcune osservazioni
  - i test di integrazione richiedono di poter avviare alcuni servizi infrastrutturali o applicativi (magari sostituiti da test double) in un ambiente di test, spesso distribuito e sicuramente separato e isolato dall'ambiente di produzione

Test di integrazione basati su contratti

- Un approccio alla realizzazione dei test di integrazione è il *consumer-driven contract test* ("test a contratti guidato dai consumatori")
  - ogni scenario di interazione (tra più elementi del sistema) viene specificato sotto forma di un "contratto" (in modo "dichiarativo")
  - ogni contratto viene poi usato per generare dei test double eseguibili, per verificare separatamente, ma in modo coerente, i partecipanti a quell'interazione

Test di integrazione basati su contratti

- Un approccio alla realizzazione dei test di integrazione è il *consumer-driven contract test* ("test a contratti guidato dai consumatori")
  - ad es., un contratto potrebbe essere relativo alla chiamata di un'operazione REST di un servizio S da parte di un suo client (consumatore) – e specificato come una coppia richiesta HTTP/risposta JSON
  - il contratto viene poi usato per generare dei test double eseguibili, per verificare in modo coerente
    - il fornitore del servizio S – per verificare che a fronte di quella richiesta fornisca quella risposta
    - i consumatori di S – per verificare come si comportano se quando fanno quella richiesta ottengono quella risposta
  - sono utili soprattutto quando sono possibili evoluzioni autonome nell'implementazione di S e dei suoi consumatori

- Test end-to-end

- I *test end-to-end* hanno lo scopo di verificare il collegamento complessivo tra gli elementi del sistema, a partire da interazioni esterne al sistema – ad es., da una richiesta fatta da un utente tramite UI o un'API fino all'accesso alla base di dati
  - verificano direttamente l'integrazione e le interazioni tra gli elementi del sistema
  - richiedono un ambiente di test simile all'ambiente di produzione – ma separato da esso
  - sono sostenuti da opportuni framework – ad es., per simulare il comportamento dell'utente tramite la UI
  - sono solitamente lenti da eseguire

- Test di componenti

- I *test di componenti* hanno lo scopo di verificare il comportamento complessivo di singoli componenti architetturali
  - intuitivamente, a metà strada tra i test di integrazione e quelli end-to-end
  - come i test di integrazione, utilizzano servizi infrastrutturali reali – dunque richiedono un ambiente di test opportuno
  - diversamente dai test di integrazione, viene testato tutto il componente (la logica di business insieme agli adattatori)
  - a differenza dei test end-to-end, quando si testa un componente A che deve interagire con un componente B, il componente B viene comunque sostituito da un test double

- Test di accettazione

- I *test di accettazione* (*user acceptance test*, *UAT*) hanno lo scopo di verificare il funzionamento complessivo del sistema – considerato a scatola nera e dal punto di vista dell'utente – sulla base di un insieme di scenari di accettazione per il sistema
  - sono test end-to-end
  - possono essere relativi sia a scenari funzionali che a scenari di qualità (*quality assurance*, *QA*)
  - l'ambiente di test deve essere il più possibile simile all'ambiente di produzione
  - molti test di accettazione dovrebbero essere automatizzati – ma spesso sono necessari anche dei test manuali

- Test per scenari di qualità

- I test per i diversi attributi di qualità sono spesso realizzati usando tecniche e strumenti specifici
  - ad es., test per le prestazioni

- Test di regressione

- I test di regressione (*regression test*) sono utilizzati per verificare nuovamente un sistema software – dopo un cambiamento nel sistema o in una sua parte
  - per verificare che il cambiamento non abbia introdotto una *regressione* nel sistema
  - dovrebbero essere automatizzati – possono comprendere tutti i test automatizzati del sistema, o un loro sottoinsieme rappresentativo
  - dovrebbero essere eseguiti periodicamente – e comunque dopo ogni cambiamento significativo del software

- Quadrante dei test

Business facing

Acceptance testing *Did we build the right thing?* Automated (e.g., Fit-Finesse)	**Exploratory testing** *Usability;* *how can I break the system?* Manual
Unit testing *Did we build it right?* Automated (e.g., xUnit)	**Property testing** *Performance; scalability;* *availability; security* Tools

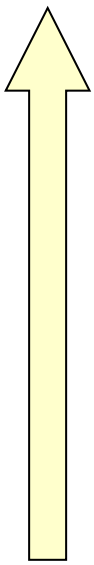
Support programming

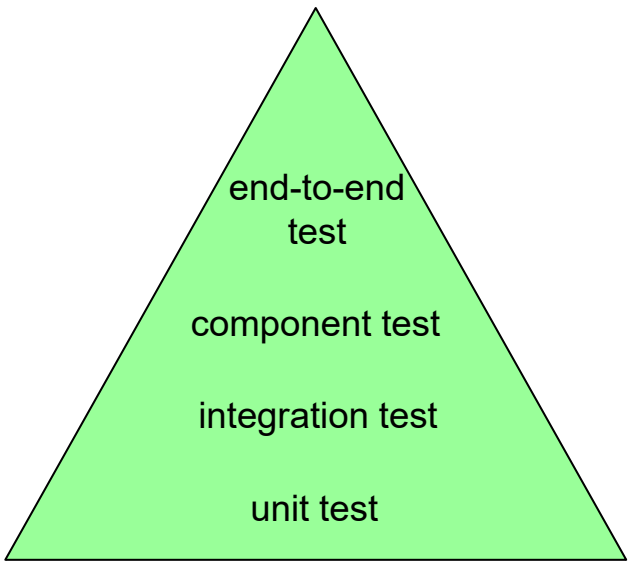
Critique product

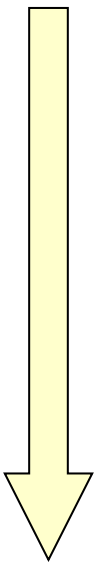
Technology facing

- Piramide dei test

increasing scope
more confidence







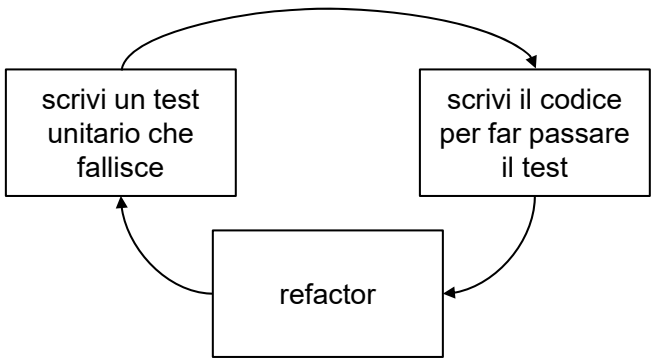
faster
better isolation

- Test-Driven Development

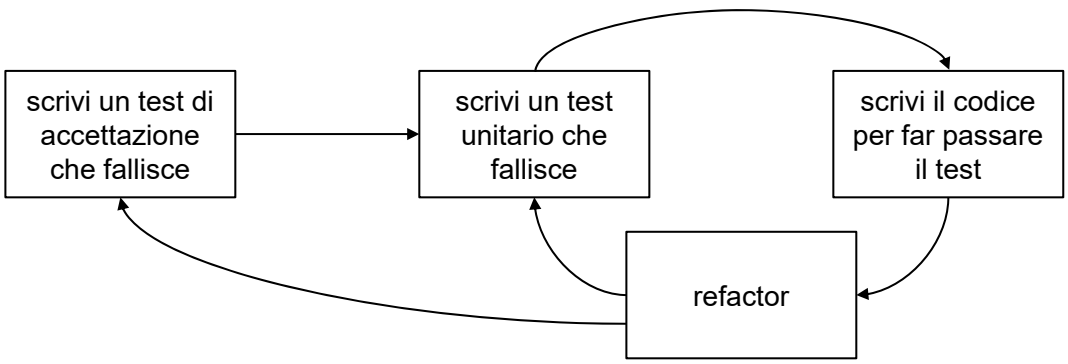
- Una pratica agile è lo sviluppo guidato dai test (*TDD*, *Test-Driven Development*)
  - prima di scrivere il codice per un pezzo di funzionalità, viene scritto un test automatizzato per la funzionalità
  - poi viene implementata quel pezzo di funzionalità, con l'obiettivo di far passare il test
  - quando il test passa, viene effettuato il refactoring, per migliorare la qualità del codice
  - questi passi vengono ripetuti

Test-Driven Development

- Il ciclo del TDD



- Il doppio ciclo del TDD



Test-Driven Development

- Vantaggi del TDD
  - vengono scritti i test per tutto il codice
  - soddisfazione dello sviluppatore
  - i test consentono una verifica automatica e ripetibile del software
  - fiducia nei cambiamenti – il TDD e il refactoring si sostengono a vicenda
  - chiarimento dell'interfaccia e del comportamento

- Verifica in produzione

- Le pratiche moderne suggeriscono di verificare un servizio o sistema software anche quando viene rilasciato in produzione, o anche durante la sua esecuzione
  - smoke testing – per verificare se un servizio o un'applicazione è stata avviata, subito dopo il suo deployment
  - early release testing – ad es., beta testing, nel canary release, nell'A/B testing
  - error detection – usare il monitoraggio per verificare in modo continuo il comportamento del sistema e i suoi indicatori di prestazioni
  - live testing – verificare il comportamento del sistema (sempre mediante il monitoraggio) in presenza di perturbazioni – l'esempio più comune è la Simian Army di Netflix

Esempio: Simian Army

- Netflix utilizza un insieme di strumenti di chaos engineering (*Simian Army*) per la verifica di sistemi software nel cloud – ad es.
  - *Chaos Monkey* uccide VM in modo casuale – per verificare la tolleranza ai guasti e la capacità di ripristino del sistema
  - *Latency Monkey* introduce ritardi artificiali nella comunicazione tra processi – per simulare il degrado dei servizi e per comprendere come questo ha impatto sugli utenti
  - *Doctor Monkey* applica dei check alle diverse VM per monitorare il loro stato di salute
  - *Conformity Monkey* cerca e spegne istanze di VM che non aderiscono alle best practice del sistema (ad es., VM che non appartengono a nessun gruppo di auto-scaling)
  - *Security Monkey* cerca e spegne VM che presentano violazioni della sicurezza o vulnerabilità (ad es., gruppi di sicurezza configurati male)

- Discussione

- Quando eseguire i test?
  - i test dovrebbe essere eseguiti in modo più o meno continuo – soprattutto in corrispondenza ai cambiamenti del sistema software o del suo ambiente di esecuzione
    - per fornire un feedback continuo sul software – soprattutto sull'impatto di ciascun cambiamento realizzato
  - può essere utile partizionare i test in una sequenza di gruppi di test
    - per raggiungere un compromesso tra rapidità e accuratezza del feedback

* Progettare per la verificabilità

- Due categorie principali di tattiche per la verificabilità [SAP] – per semplificare la verifica, quando viene completato un incremento (piccolo o grande) del software
  - *control and observe system state* – per controllare e osservare lo stato del sistema
  - *limit complexity* – per limitare la complessità del progetto del sistema
  - un obiettivo fondamentale è sostenere la definizione di test, soprattutto automatizzati, per verificare il sistema software

- Control and observe system state

- Il controllo e l'osservazione hanno un ruolo fondamentale nel test del software – se non si può controllare oppure osservare, allora non si può nemmeno verificare
  - ogni componente software fornisce normalmente dei meccanismi di controllo e osservazione – sotto forma di input e output
  - il testing può beneficiare dalla presenza di ulteriori meccanismi di controllo e osservazione diretta dello stato interno dei componenti

Control and observe system state

- *Executable assertions*
  - le asserzioni sono condizioni che consentono di verificare se un programma è in uno stato corretto oppure erroneo
  - questa tattica supporta l'automazione dei test e, in particolare, l'attività di verifica (verify)
- *Specialized interfaces*
  - introdurre e utilizzare delle interfacce specializzate specifiche per il test – da usare esclusivamente per il test
    - per semplificare le attività di preparazione, verifica e rilascio di un test

Control and observe system state

- *Sandboxing*
  - eseguire il test in un ambiente isolato da quello di produzione
  - una forma comune di sandboxing è la virtualizzazione
    - ad es., la virtualizzazione dell'ambiente di test e l'uso di test double
- *Record/playback*
  - la diagnosi di un guasto è di solito facilitata dalla ripetibilità del guasto – ma non tutti i guasti sono facili da ricreare
  - la "registrazione" dello stato del sistema oppure di uno scenario effettivo di utilizzo del sistema che ha portato a un guasto può consentire di "riprodurre" quel guasto

- Limit complexity

- Il software complesso è difficile da verificare – per questo il progetto del sistema software deve essere semplice
  - le tattiche per la modificabilità sostengono anche la verificabilità
- *Limit structural complexity*
  - evitare o rimuovere dipendenze cicliche
  - isolare e incapsulare le dipendenze (comprese le dipendenze dall'ambiente esterno)
  - più in generale, ridurre le dipendenze tra i componenti
- *Limit behavioral complexity*
  - ad es., limitare il nondeterminismo

* Discussione

- La verificabilità riguarda la facilità con cui è possibile rilevare errori o guasti in un sistema software
  - la verifica di un sistema si basa sull'esecuzione di un ampio insieme di test – di diversi tipi, con obiettivi e caratteristiche differenti
  - sostenere la verificabilità porta a vantaggi di
    - riduzione dei costi della verifica
    - sostegno ad altre qualità del software
  - i test possono essere utilizzati per ottenere un feedback importante su un sistema software in corso di sviluppo
    - è spesso utile anche un feedback relativo all'esecuzione del sistema software nel suo ambiente di produzione – che può essere ottenuto mediante il monitoraggio (discusso in un successivo capitolo)