-Prodotto: Rappresenta cosa realizziamo (software). Il prodotto software ha 3 caratteristiche principali: -*Intangibile:* difficile da descrivere e valutare. –*Malleabile:* può essere trasformato e dotato di nuove funzionalità. –*Human Intensive:* non comporta nessun processo manifatturiero tradizionale. Le qualità del prodotto riguardano il software stesso e sono sempre valutabili
-Processo: Rappresenta come realizziamo il prodotto. Esistono diversi modelli in cui organizzare le fasi di sviluppo. Le qualità del processo riguardano i metodi utilizzati per lo sviluppo del software. Le qualità del processo influiscono su quelle del prodotto. Le qualità sono: -*Produttività*: misura l'efficienza del processo di produzione del software in termini di velocità di consegna del software. –*Tempestività*: misura la capacità del processo di produzione del software di valutare e rispettare i tempi di consegna del prodotto. –*Trasparenza:* misura la capacità del processo di produzione del software di capire il suo stato attuale e tutti i suoi passi.
-Qualità del software: Sono utili per la valutazione del software, e sono classificate in Esterne e Interne, strettamente connesse in quanto non è possibile ottenere le qualità esterne se il software non gode delle qualità interne. *–Esterne (Black Box View)*: Sono quelle percepite da un osservatore esterno che esamina il prodotto come se fosse una scatola nera visibile agli utenti; riguardano soprattuto le funzionalità del prodotto. Alcune qualità esterne sono: correttezza, affidabilità, efficienza, usabilità, portabilità, interoperabilità, robustezza. *–Interne (White Box View)*: Sono quelle percepite da un osservatore che esamina il processo o il prodotto come se fosse una scatola trasparente; non sono visibili agli utenti e riguardano soprattutto le caratteristiche legate allo sviluppo del software. Alcune qualità interne sono: riusabilità, verificabilità, facilità di manutenzione.
-Correttezza: Un software è corretto se rispetta le specifiche di progetto. E' una qualità assoluta in quanto non esiste il concetto di "grado di correttezza" e di "gravità di deviazione", e in quanto specifiche sbagliate possono dipendere da requisiti incorretti o da errori nella conoscenza del dominio di applicazione. Se le specifiche sono espresse formalmente, la correttezza può essere definita formalmente, quindi può essere provata con un teorema (verifica), oppure valutata attraverso dei contro-esempi (testing).
-Affidabilità: Un software è affidabile se si comporta "come previsto". Questa qualità può essere valutata matematicamente come la "probabilità di assenza di fallimenti in un dato intervallo di tempo"; inoltre, se le specifiche sono corrette, tutto il software corretto è affidabile, ma non vale il viceversa.
-Efficienza: Un software è efficiente se usa intelligentemente le risorse di calcolo (ad es. memoria, tempo di processamento, metodi di comunicazione). Questa qualità può essere valutata attraverso analisi di complessità o la simulazione di scenari critici; può inoltre influenzare la scalabilità, in quanto: una soluzione che funziona in piccolo può non funzionare in grande; cambia la tecnologia; e può influenzare la facilità d'uso.
-Usabilità: Facilità d'uso da parte dell'utente. E' una qualità difficile da valutare in quanto molto soggettiva; comporta la definizione di *expected user*, e influisce molto sulle interfacce utente (visuale vs testuale).
-Portabilità: Un software è portabile se può funzionare su più piattaforme.
-Interoperabilità: Fa riferimento all'abilità di un sistema di coesistere e cooperare con altri sistemi.
-Robustezza: Un software è robusto se si comporta in modo ragionevole anche in circostanze non previste dalle specifiche di progetto (es.input incorretti, rotture di dischi, ecc.).
-Riusabilità: Un software è riusabile se può essere usato, in tutto o in parte, per costruire nuovi sistemi.
-Verificabilità: Possibilità di dimostrare a posteriori la correttezza o altre caratteristiche del software.
-Facilità di manutenzione: Facilità nel realizzare adattamenti o evoluzioni; agio nel corregere gli errori. Un software è facile da mantenere se: -è strutturato in modo tale da facilitare la ricerca degli errori (*modifiche correttive*); -la sua struttura permette di aggiungere nuove funzionalità al sistema (*modifiche perfettive*); -la sua struttura permette di adattarlo ai cambiamenti del dominio applicativo (*modifiche adattative*).
-Sicurezza: E' una proprietà relativa e non assoluta del sistema; il suo significato è relativo al contesto di applicazione, e può essere definita rispondendo alla domanda: "sicurezza contro cosa e da chi?". E' una qualità che implica lo stabilire una policy di sicurezza. Un *software sicuro*, oltre alle qualità comuni a tutti i software, deve godere delle seguenti qualità (*security goals*): *-Prevenzione:* Anticipare possibili attacchi; prevenire i fattori di vulnerabilità. *–Tracciabilità*: E' il meccanismo che consente di stabilire in modo inequivocabile la relazione di causa/effetto tra elementi, eventi, o processi. E' utile per riparare ad un attacco. *–Controllo (o auditing):* Per auditing si intende il processo di controllo di un sistema, effettuato sulla base del confronto tra le attività svolte sul sistema con le politiche e le procedure stabilite al fine di determinare la loro conformità, suggerende eventualmente l'opportunità di introdurre delle migliorie. Non è una tecnica di prevenzione, ma è utile per dissuadere da potenziali attacchi. *–Monitoraggio:* E' auditing in tempo reale. Sistemi per il monitoraggio possono causare un elevato numero di falsi allarmi. E' possibile monitorare un programma a diversi livelli: approcci semplici, cioè controllo di segnature note che identificano un attacco in corso, e approcci complessi, cioè monitoraggio del codice mediante asserzioni. *–Privatezza:* E' il diritto di un individuo a stabilire se, come, quando e a chi l'informazione che lo riguarda può essere rilasciata. *–Confidenzialità:* Assicura che certi servizi e informazioni siano accessibili solo ad utenti autorizzati. *–Sicurezza a più livelli:* Alcuni tipi di dati e informazioni sono più segrete di altre, quindi è necessario disporre di sistemi in grado di gestire la segretezza di dati e informazioni a più livelli. *–Anonimato:* Va intesa come la proprietà di mantenere segreta (o non accessibile pubblicamente) l'origine di certi dati. Il software prende spesso decisioni non anticipate e programmate sull'anonimato (il Global Identifier di Microsoft, e il sistema Carnivore dell'FBI). Architetti e sviluppatori debbono pensare attentamente a cosa avviene dei dati collezionati dai propri programmi. *–Autenticazione:* Va intesa come la proprietà di conoscere l'identità di chi accede ad un servizio. E' una proprietà critica per la sicurezza, in quanto un software sicuro quasi sempre include elementi di autenticazione. Scarsa è l'autenticazione sul web, dove la tecnologia SSL assicura la protezione dei dati ma non garantisce l'identità del server a cui si è connessi. Esistono parecchi modi per garantirla in relazione al software di applicazione. *–Integrità*: Nel contesto della sicurezza, integrità va intesa come "rimanere lo stesso", ossia non modificato da quando creato. In sistemi sicuri, le informazioni e le risorse non devono essere modificate, cancellate o distrutte in modo non autorizzato o improprio.
-Processo di produzione: La sequenza di operazioni che viene esguita per progettare, realizzare, consegnare e modificare un prodotto software.
-Ciclo di vita di un software: L'insieme di stadi in cui il sistema viene a trovarsi.
-Modelli di processo: Determinano l'organizzazione delle diverse fasi di sviluppo.
-Fasi di sviluppo: Le attività del processo sono organizzate in: *-Studio di fattibilità:* Scopo di questa fase è la produzione di un documento detto Feasibility Study Document (*FSD*) che valuta i costi e i benefici della applicazione proposta. L'FSD deve contenere la definizione del problema, le possibili soluzioni e le relative motivazoni e, per ognuna delle soluzioni proposte, la stima dei benefici, dei costi, delle risorse richieste e dei tempi di consegna. *–Specifica (o analisi dei requisiti):* Ha lo scopo di determinare le funzionalità richiesta dal cliente e le proprietà del software in termini di performance, facilità d'uso, portabilità, facilità di

manutenzione, ecc. La fase di specifica richiede interazione con il cliente e comprensione delle proprietà del dominio del problema, e produce il *RASD* (Requirements Analysis and Specification Document). Il RASD deve permettere al cliente di verificare le caratteristiche specificate, e deve consentire al progettista di procedere allo sviluppo dell'architettura del software. Le proprietà richieste dal RASD sono precisione, compeltezza e consistenza, e dovrebbe includere un User Manual preliminare, e un System Test Plan, ovvero la definizione delle modalità di test del sistema. Nella fase di specifica occorre descrivere cosa fare e non come farlo, rispondendo alle 5 W (who, why, what, where, when). , progettazione (o design), impementazione e test dei moduli, integrazione e test del sistema, consegna, manutenibilità. *–Progettazione:* Scopo di questa fase è la produzione di un documento contenete la descrizione dell'architettura del software sia a livello globale che a livello dei singoli moduli. La fase di design produce il *Design Document*: definizione delle componenti (o moduli), relazioni tra le componenti, interazione tra le componenti. Gli obiettivi della progettazione sono lo sviluppo concorrente e la separazione delle responsabilità. *–Implementazione e test dei moduli:* E' la fase in cui i programmi vengono realizzati dai programmatori i quali si occupano di effettuare i test sulle funzionalità inerenti i singoli moduli. Per ogni componente si fornisce codifica, documentazione, e specifica dei test effettuati. *–Integrazione e test del sistema:* Questa fase ha lo scopo di assemblare il codice prodotto dai diversi gruppi di programmatori e verificarne l'effettiva compatibilità risolvendo eventuali errori derivanti dall'interazione di due o più moduli. *–Consegna:* In questa fase il sistema viene distribuito agli utenti che ne verificano il funzionamento, individuando eventuali malfunzionamenti o dissimilarità rispetto alle specifiche di progetto. La consegna avviene in due fasi: *-Beta Testing:* il software viene distribuito ad un insieme selezionato di utenti allo scopo di effettuare testi in casi reali. Gli errori riscontrati vengono corretti prima della distribuzione effettiva. *–Distribution:* il software viene definitivamente rilasciato agli utenti. Gli errori che vengono riscontrati vengono solitamente corretti nelle versioni successive o tramite l'utilizzo di appositi software correttivi (patch). *–Manutenibilità:* E' la fase che include tutti i cambiamenti (correzione + evoluzione) che seguono la distribuzione del software. Spesso comporta più del 50% del costo totale del software.

-Modelli di processo: I cicli di vita differiscono principalmente nel modo e nell'ordine con cui le attività sono realizzate. I modelli di ciclo di vita sono classificati in modelli a cascata, modelli evolutivi, e modelli a spirale.

-Modelli a cascata (Waterfall): Rappresenta lo standard per la produzione di sistemi informatici. L'output di ogni fase rappresenta l'input della fase successiva. *Limiti:* ne esistono molte varianti e ciascuna organiazzazione tende a definire il "proprio"; il modello implica il completamento di una fase prima di passare alla fase successiva; è un processo black box e non consente prototipazione; non tiene in considerazione il fatto che i requisiti cambiano (cambia il contesto e le specifiche potrebbero risultare sbagliate).

-Evoluzione (come affrontarla): L'idea di base è di disporre i processi incrementali e di poter avere un feedback (fondamentale per consentire controlli e cambiamenti anticipati) sugli incrementi.

-Processi flessibili: Capaci di adattarsi ai cambiamenti, in particoalre ai cambiamenti dei requisiti e della specifica. Permettono il riciclo, ovvero modificare prima il progetto e poi cambiare il codice o applicare cambiamenti in maniera consistente in tutti i documenti. Un processo flessibile consente *validazione* e *verifica.* I processi flessibili esistono in molte forme: *-Prototipazione:* E' basato sul principio "do it twice"; un prototipo è un modello approssimato dell'applicazione che ha lo scopo di fornire feedback necessari a individuare con esattezza tutte le caratteristiche del sistema e tutti gli errori di progettazione effettuati. Per evitare che l'adozione di questo modello comporti costi troppo elevati è necessario che i progettisti siano coscienti della funzione del prototipo e che tutte le fasi del processo siano basate su un criterio di massima riusabilità delle componenti sviluppate: massimizzare la modularizzazione e ingegnerizzare le sole componenti critiche nel prototipo. *Modello a fasi di release (o Incremental Delivery):* E' basato sul principio "early subset, early delivery, ... , early feedback. Si parte da sottoinsiemi critici sui quali richiedere il feedback del cliente. L'incremental delivery è ancor oggi attuato. *–Modello a spirale:* E' un processo ciclico tra le seguenti attività: analisi dei rischi, sviluppo (specifica, design, codifica), e validazione. Ad ogni ciclo il costo aumenta, proprio come in una spirale. Questo modello ingloba prototipazione e approccio iterativo in quanto: inizia sviluppando un piccolo prototipo; continua seguendo un mini-processo waterfall, principalmente per raccogliere i requisiti; quindi, il primo prototipo è revisionato; nei loop successivi, il team di progetto realizza altri requisiti, design, implementazione e revisione; prima di iniziare un nuovo ciclo, fa l'analisi dei rischi; il mantenimento è semplicemente una forma di sviluppo continuo. Il modello a spirale è il modello di sviluppo ottimale per poter effettuare un'efficace analisi di sicurezza: prevede al suo interno la fase di analisi dei rischi, è flessibile, prevede un raffinamento successivo, e consente la validazione già nelle fasi iniziali con la possibilità di integrare soluzioni trovate a posteriori.

-Analisi di sicurezza (analisi e gestione dei rischi): E' importante per riuscire a bilanciare tra sicurezza e funzionalità: è indispensabile quindi per decidere quali obiettivi realizzare e quali proprietà e qualità garantire. L'analisi di sicurezza è un processo *ortogonale* al processo di sviluppo del software, è il suo modello di sviluppo ottimale, è quello a spirale.

-Ingengere della sicurezza: E' una nuova figura professionale che si occupa di analisi di sicurezza ortogonalmente al processo di sviluppo; si richiede profonda conoscenza di sviluppo del software, e competenze di sicurezza.

-Artifatti e Stakeholders: L'analisi di sicurezza comporta l'impiego di tutti gli artifatti (documenti dei requisiti, documenti dell'architettura e del design, modelli, ecc.), e degli stakeholder (utenti, clienti, ingegneri del software/della sicurezza, esperti di dominio) coinvolti come risorse per l'identificazione dei rischi.

-Sicurezza e requisiti: Nella definizione dei requisiti connessi alla sicurezza occorre identificare ciò che va protetto, da chi va protetto e per quanto tempo, e anche quanto vale mantenere certi dati protetti. I requisiti devono essere enunciati in modo chiaro e completo; il documento dei requisiti deve specificare cosa un sistema deve e non deve fare, ma anche perchè deve farlo.

-Analisi dei rischi: Ha l'obiettivo di identificare i rischi possibili sulla base dei requisiti di sicurezza stabiliti, e di valutare strategie per prevenirli o affrontarli (l'analisi dei rischi influisce sul processo di auditing). I rischi vanno classificati in base alla loro severità (è una classificazione context-sensitive importante per allocare risorse). E' difficile comparare sistemi in temrini di sicurezza, in quanto non esiste una metrica "assoluta" ma linee guida molto generali per la valutazione e la comparazione di sistemi sicuri.

-Sicurezza e Specifica: Dai requisiti si ricava la specifica che deve anche integrare eventuali possibili soluzioni a rischi individuati. L'analisi di sicurezza continua in fase di specifica; è molto importante avere una specifica dettagliata (che descriva la reazione del sistema a circostanze critiche), formale (completa e non ambigua, ma anche chiara e comprensibile), ed eseguibile per avere un feed-back immediato. Se nuovi rischi sono individuati, vanno riflessi nei requisiti.

-Sicurezza e Design: L'analisi di sicurezza nella fase di design consiste principalmente nell'identificare: come i dati passano da una componente all'altra; utenti, ruoli e diritti che sono esplicitamente dichiarati o implicitamente supposti dal design; ogni soluzione potenzialmente applicabile a problemi individuati; la relazione di *trust* di ciascuna componente. Il *security plan* va attuato in fase di design per essere effettivo.

-Sicurezza ed implementazione: L'implementazione è una fase critica per la sicurezza perché molti attacchi sono possibili proprio a causa di fattori tecnici ed umani connessi alla fase di implementazione. Compito dell'ingegnere della sicurezza è quello di: identificare le linee guida per la stesura del codice, ed effettuare il controllo del codice (code audit).
-Sicurezza e testing: Per sistemi sicuri si distingue tra: *-testing funzionale:* comporta il mettere alla prova un sistema per determinare se il sistema fa ciò che si suppone debba fare in circostanze normali. *–testing di sicurezza:* comporta il mettere alla prova un sistema allo stesso modo in cui può provarlo un utente malizioso, ossia sfruttando i punti di debolezza del sistema. *Code Coverage* rimane ancora un metodo importante per il testing di sicurezza, in quanto permette di scoprire se esistono parti di codice che non vengono mai eseguite (codice non eseguito può infatti avere seri bachi sfruttabili per attacchi di successo).
-Standard per la sicurezza: *-Orange Book:* pubblicato nel 1985 in collaborazione tra il NSA e il Dept. Of Defence Trusted Computer System Evaluation Criteria. *–Common Criteria* (versione 2): è uno standard ISO che definisce un insieme di classi e di comportamenti di sicurezza progettate per essere opportunamente combinate per definire profili di protezione per ogni tipo di prodotto IT, incluso hardware, firmware, software.
-Common Evaluation Methodology: Definisce le modalità di valutazione di un sistema in base a criteri comuni. *Processo di valutazione:* 1) Wall Street produce un profilo di protezione per il firewall usato per proteggere le macchine che contengono informazioni finanziarie dei clienti; 2) Il profilo viene valutato in base alle Common Evaluation Methodology per garantire che esso sia consistente e completo; 3) Ottenuta la valutazione, il profilo viene pubblicato (*target di sicurezza*); 4) Un vendor realizza una sua versione (*target di valutazione*) di firewall dotato del profilo di protezione definito da Wall Street; 5) Il *target di valutazione* viene inviato ad un istituto accreditato per la valutazione rispetto al *target di sicurezza*: -L'istituto applilca la Common Evaluatoin Methodology per determinare in base ai common criteria se il target di valutazione soddisfa il target di sicurezza; -Se la valutazione ha esito positivo, la documentazione di testing dell'istituto viene inviata al NAVLAP (National Voluntary Lavoratory Accreditation Program) per controllarne la correttezza; -Se gli atti vengono apporvati, il prodotto ottiene la *certificazione* di prodotto valutato in base ai common criteria.
-Limiti dei Common Criteria: La valutazione della sicurezza è un'attività molto più complessa dell'applicare un target di sicurezza ad un target di valutazione. I problemi di sicurezza sono context-sensitive, e quindi è difficile valutarli in base a criteri comuni. Inoltre, come molti standrard, definiscono il "cosa" e non il "come", mentre nel campo della sicurezza è fondamentale sapere *come* affrontare problemi di sicurezza e gestire i rischi.
-Valutazione delle smart cards: Dal 2001 lo Smart Card Security User's Group sta cercando di definire dei Common Criteria per la valutazione delle smart cards. Questo processo converge lentamente per due motivi: 1) Esistono due profili per la sicurezza: -quello europeo, che mira maggiormente alla protezione fisica dei dati; -quello americano, che mira maggiormente alla protezione software e alla prevenzione di attacchi noti. 2) La maggior parte degli enti che lavorano sulla sicurezza delle smart cards non sono interessati ai Common Criteria.
-Ciclo di vulnerabilità: E' una sequenza di eventi che si susseguono molto comunemente nel mondo della sicurezza: va dalla determinazione alla eliminazione di una vulnerabilità del software. E' un ciclo ortogonale al ciclo di vita del software, e solitamente accade dopo la sua distribuzione. *Sequenza di eventi:* 1)Viene scoperta una *nuova vulnerabilità* in un pezzo di software. 2) I *cattivi* analizzano l'informazione e sfruttano la vulnerabilità per lanciare attacchi contro il sistema o la rete. 3)I *buoni* cercano una soluzione al problema: analizzano la vulnerabilità, sviluppano una soluzione e la testano in un ambiente controllato, distribuiscono la soluzione. 4)Se la vulnerabilità è seria, o gli attacchi sono gravi, anche i media ne danno pubblica informazione con catastrofiche conseguenze. 5)La *patch* distribuita (spesso rilasciata come parte di update automatico), viene installata agli addetti. 6)Tencici della sicurezza analizzano frammenti di codice per scoprire vulnerabilità simili. *Complicazioni:* Sistemi e reti vulnerabili raramente vengono riparate durante il ciclo di vita della vulnerabilità: la patch sono ricevute come parte di una versione aggiornata del software. Il software maligno rilasciato via Internet può sfruttare la vulnerabilità dei sistemi causando e propagando danni senza misura.
-Attacco: Un attacco ad un sistema è ogni atto malevolo contro un sistema o un complesso di sistemi. Nell'ambito di un attacco vanno distinti (*fasi di un attacco)*: -*Goal:* il danno causato al sistema; -*Sottogoal:* raggiungere un goal può richiedere raggiungere prima dei sottogoal; -*Attività:* le attività che un attaccante deve svolgere per raggiungere uno o più sottogoal; -*Eventi*: le attività citate, possono rivelarsi eventi di attacco; -*Conseguenze:* il diretto risultato di un evento; -*Impatto:* effetti di business. *Esempio di attacco:* ATTACKER → -Rubare denaro (goal); -Ottenere accesso non autorizzato (sottogoal); -Usare password rubata (attività); -Controlli d'accesso violati (eventi); -Mancanza di...(conseguenze); -Perdita di guadagno (impatto); → TARGET.
-Modalità di attacco: Il *come* un attacker attacca un sistema, dipende spesso dal *perchè* e dalla *competenza* in materia di sicurezza del safecracker. Per rafforzare la sicurezza di un'applicazione è importante domandarsi *cosa* può accadere, e *come* può accadere.
-Tipi di attacco: In fase di: -*Progettazione* (mentre si sta progettando l'applicazione); -*Implementazione* (mentre si sta scrivendo l'applicazione); -*Operazione* (dopo che l'applicazione è in produzione).
-Attacchi in fase di progettazione: *-Man-in-the-middle attack:* Accade quando un attacker intercetta una trasmissione di rete tra due host, e si spaccia per una delle due parti coinvolte nella transazione, possibilimente aggiungendo ulteriori direttive nel dialogo. *Difesa:* usare intensivamente tecniche di *encryption* (in particolare autenticazione crittografica forte), ed usare *session checksum* e *shared secrets* come cookies (usare ssh invece che telnet, criptare i file usando utilities come PGP o Entrust). *–Race condition attack:* Certe operazioni comuni ad applicazioni software sono, dal punto di vista del computer, costituite da passi discreti (anche se noi le riteniamo atomiche). Il tempo per completare questi passi apre una finestra durante cui un utente malizioso può compromettere la sicurezza. Un esempio è l'operazione: 1.controllare se un file contiene comandi sicuri da eseguire; 2.eseguirlo. Un utente malizioso potrebbbe sostituire il file tra i passi 1 e 2. *Difesa:* Comprende la differenza tra operazioni atomiche (indivisibili) e non-atomiche, ed evitare le ultime a meno che non si sia sicuri che non abbiano delle implicazioni non sicure. Se non si è sicuri che un'operazione sia atomica, assumere che non lo sia, ossia che il sistema operativo possa eseguirla in due o più passi interrompibili. *–Replay attack:* Se un attacker riesce a catturare o ottenere il record di una intera transazione tra un programma client ed un server, ha la possibilità di "riprodurre" parte della conversazione a scopo sovversivo. *Difesa:* come per il man-in-the-middle attack. In più, introdurre in ogni dialogo alcuni elementi (come un sequence identifier) che differisca da sessione in sessione in modo da far fallire il byte-for-byte replay. *–Sniffer attack:* Uno *sniffer* è un programma che silenziosamente memorizza tutto il traffico spedito su una rete locale. Gli sniffer sono talvolta tool legittimi con fini diagognistici, ma sono anche utili agli attaccanti per memorizzare usernames e passwords trasmesse in chiaro. *Difesa:* come sistemista, attente configurazione della rete, e uso di "switched" network routers; come programmatore, massimizzare l'uso della crittografia. *–Session hijacking attack:* Sfruttando le debolezze del protocollo TCP/IP, un attaccante può riuscire a prendere il controllo (o *hijacking*) di una connessione già stabilita. Esistono molti tool che sono stati scritti e

distribuiti su Internet per implementare questo tipo di attacco. *Difesa:* E' un attacco di rete da cui un'applicazione software riesce difficilmente a difendersi. La crittografia è di aiuto, e se un attento logging fornisce abbastanza informazioni sulla sessione, procedure operazionali possono essere in grado di rilevare un attacco di hijacking dopo che si è verificato. *–Session killing attack:* Sezioni illegittime di TCP/IP possono terminare se una delle due parti che comunicano invia il packet *TCP reset*. Un attaccante potrebbe essere in grado di forgiare gli indirizzi di una delle due parti e resettare prematuramente la connessione. Un attacco di questo tipo può essere suato per distruggere una comunicazione, o per prendere il controllo di una parte della trasmissione. *Difesa:* E' difficile a livello di applicazione potersi difendere da questi tipi di attacchi alla rete. Tuttavia, l'applicazione potrebbe essere in grado di controbattere gli effetti dell'attacco ristabilendo la connessione interrotta.

-Attacchi in fase di implementazione: *Buffer overflow attack:* Molti linguaggi di programmazione consentono di allocare un buffer di lunghezza fissa per una stringa di caratteri ricevuta in input, ad es. come argomento a linea di comando. Una condizione di *buffer overflow* si verifica quando l'applicazione non effettua un adeguato *bounds checking* alla stringa ed accetta più caratteri di quanto sia possibile memorizzare nel buffer. In molti casi, un attaccante può causare un overflowed del buffer e forzare il programma ad eseguire comandi o azioni non autorizzate. *Difesa:* Usare un linguaggio di codifica (come Java) che esclude overflows per design; altrimenti evitare di leggere stringhe di testo di lunghezza indeterminata in buffer di lunghezza fissa a meno che non puoi leggere in modo sicuro sottostringhe di lunghezza specificata che possono essere contenute nel buffer. *–Back door attack:* Molti sistemi di applicazioni vengono compromessi da attacchi introdotti durante la scrittura del codice. Può accadere, per es., che un programmatore furfante scriva all'interno dell'applicazione del codice speciale che consenta in seguito di bypassare il controllo d'accesso. Tali punti di accesso speciali sono detti *back doors*. *Difesa:* Adottare procedure di garanzia delle qualità che controllano la presenza di back doors sull'intero codice. *–Parsing error attack:* Le applicazioni spesso accettano input da utenti remoti senza controllare opportunamente i dati in input. Il controllo (o *parsing*) dei dati in input ai fini della sicurezza è importante per bloccare gli attacchi. Un famoso esempio di errore di parsing riguardava server web che non controllavano le richieste con all'interno sequenze "../", e ciò consentiva all'attaccante di raggiungere parte delle directory del filesystem il cui accesso dovrebbe essere proibito. *Difesa:* Si raccomanda il riuso di codice esistente, scritto da una specialista, che è stato opportunamente controllato, testato e mantenuto. Quando si scrive del codice, bisogna tener conto che è molto più sicuro controllare che ogni carattere di input compaia in una lista di caratteri "safe", piuttosto che comparare ciascun carattere con quelli di una lista di caratteri "pericolosi".

-Attacchi in fase di operazione: *-Denial-of-service:* Si verifica quando un utente legittimo si vede "negato un servizio" richiesta. Un sistema di applicazioni, un host, o una rete possono essere resi inutilizzabili per via di una cascata di richieste di servizio, o persino di un flusso di input ad alta-frequenza. In un attacco negazione-servizio su ampia-scala, l'attaccante può usare hosts su Internet precedentemente compromessi come piattaforme di passaggio per l'assalto. *Difesa:* Progettare il software e in particolare l'allocazione delle risorse in modo tale che l'applicazione faccia moderata richiesta di risorse del sistema (come spazio su disco o numero di file aperti). Quando si progettano grandi sistemi, prevedere il monitoraggio dell'utilizzo delle risorse, e un modo che consenta al sistema il trabocco di caricamente eccessivo (cioè il software non dovrebbe lamentarsi o morire nel caso in cui si esauriscano le risorse). *–Default accounts attack:* Molti sistemi operativi e programmi applicativi sono configurati, per default, con username e password "standard" (es.guest/guest). Username e password di defautl permettono facile accesso a potenziali attaccanti che conoscono o riescono ad indovinare queste informazioni. *Difesa:* Rimuovere gli accounts di default; controllare nuovamente dopo aver installato nuovo software o nuove versioni di software esistente (infatti scripts di installazione possono talvolta reinstallare questi tipi di accounts di default). *–Password cracking attack:* Gli attaccanti ripetutatmente indovinano password scelte male, per mezzo di speciali programmi di cracking: questi programmi usano speciali algoritmi e dizionari di parole e frasi d'uso comune per indovinare centinaia di migliaia di parole; password deboli, come nomi comuni, date di nascita, o contenenti parole "secret" o "system", possono essere indovinate da questi programmi nella frazione di un secondo. *Difesa:* Come utente, scegliere password intelligenti. Come programmatore, usare tool per richiedere passwords robuste, cioè facili da ricordare e difficili da indovinare. Le norme per una buona scelta di password sono: offuscare le parole in un linguaggio sconosciuto; usare combinazioni di caratteri formate da lettere iniziali (o finali) di ciascuna parola di una frase che si ricordi; includere simboli di punteggiatura o numeri. Bisognerebbe inoltre evitare il riuso di passwords già in fase di design, usando metodi alternativi di autenticazione, come dispositivi biometrici e smart cards.

-Problema Y2K: Molti software gestivano le date memorizzando solo le ultime due cifre. Dall'anno 2000 questi sistemi andavano aggiornati in modo di gestire correttamente gli anni. Alla vigilia del capodanno 2000 c'era molto timore che potesse esserci qualche malfunzionamento che avrebbe potuto bloccare le attività più critiche e scatenare un effetto dominio. Non è accaduto nulla grazie alla "lezione del Y2K". Due motivi principali hanno contribuito al successo dell'operazione Y2K: 1.C'era una scadenza ben precisa, nota a tutti e non posticipabile. 2.Presa di coscienza del problema, in quanto il problema era noto a tutti: una ditta che non avrebbe aggiornato il proprio sistema, sarebbe potuta essere responsabile oggettivamente; la popolazione e i media erano informati e preoccupati del problema. *Cosa si può imparare dal Y2K?* –Riuso di tecniche sviluppate per l'Y2K (dopo l'attacco dell'11 settembre, diverse ditte hanno usato procedure e tecniche messe a punto per l'Y2K per ripristinare il servizio. –Rivedere la nostra concezione strategica del software all'interno della società.

-Architettura sicura:Insieme dei principi d'alto livello e delle decisioni per la progettazione di sistemi da ritenersi sicuri. -garantisce un sufficiente grado di sicurezza(troppa sicurezza può risultare dannosa);-è un framework per la progettazione sicura(secondo i 4 principi:proteggi(protect),scoraggia(deter),scopri(detect) e reagisci(react));-è definita mediante un insieme di documenti(certificano le decisioni e i principi);-deve essere riusabile.

-Politica di sicurezza(policy):E' diversa da una architettura sicura. Stabilisce:-le regole su chi puo avere accesso ed a che tipo di dati;-le procedure di testing necessarie a certificare che l'applicazione puo girare in rete in modo sicuro. Una politica di sicurezza va stabilita a priori ed usata come linea guida per definire l'architettura dell'applicazione.

-Documenti d'architettura:Un'architettura dovrebbe definire:organizzazione del programma,strategie di cambiamenti,decisione su acquisto vs sviluppo, principali strutture dati,algoritmi chiave,oggetti principali,funzionalità generica,processamento dell'errore(correttivo o determinativo),robustezza attiva o passiva e tolleranza ai guasti(fault tolerance).

-Principi di un'architettura sicura: I principi base per definire architetture sicure sono: iniziare facendo domande,decidere che livello di sicurezza è sufficiente,identificare le assunzioni,progettare con in mente il nemico,conoscere e rispettare la chain of trust,definire privilegi adeguati,testare le azioni possibili contro la policy,suddividere la progettazione in moduli,non offuscare le informazioni,mantenere in memoria lo stato minimale,garantire un adeguato livello di fault tolerance,pianificare l'error-handing,applicare il principio di graceful degradation, garantire che un fallimento lasci il sistema in una configurazione

sicura,applicare misure di sicurezza adeguate all'utente,responsabilizzare l'utente nelle sue azioni,controllare e limitare il consumo delle risorse,garantire la possibilità di ricostruire gli eventi,pianificare livelli multipli di difesa,non considerare una applicazione come monolitica,riusare SW certificato sicuro,utilizzare tecniche di ingegneria standard,progettare la sicurezza sin dalle fasi iniziali del ciclo di vita.

-Criteri di scelta di tecnologie sicure nei diversi ambiti: LINGUAGGIO DI PROGRAMMAZIONE: la scelta del linguaggio è fondamentale,i fattori di scelta includono efficienza,potere espressivo,portabilità,familiarità e sicurezza(C,C++,Java,Visual Basic,..). L'efficienza spesso porta a scegliere C e C++. Altri linguaggi possono essere veloci quasi quanto il C/C++ e quindi alcune volte è vantaggioso sacrificare un po' di efficienza per altri vantaggi come sicurezza,affidabilità,ecc. Non tutti i linguaggi offrono lo stesso livello di sicurezza,infatti i programmi C tendono ad essere inaffidabili,in programmi interpretati(VB,Perl..)sono normalmente sicuri ma possono contenere codice errato mai eseguito,i programmi scritti in Java sono molto sicuri a spese dell'efficienza,della facilità con cui scrivere i programmi e introducono i rischi di sfruttare in modo improprio o insicuro l'information hiding del linguaggio(es. private) per la sicurezza. I vantaggi di Java per quanto riguarda la sicurezza sono:no buffer overflow e no errori di memoria, ha un controllo forte e statico dei tipi,ha bisogno di un framework per eseguire codice non fidato in un sendbox e dispone di librerie per la sicurezza. PIATTAFORME AD OGGETTI DISTRIBUITE: molte applicazioni client/server si basano su piattaforme distribuite ad oggetti. Lo standard proposto dall'OMG è CORBA che offre un ottima sicurezza tramite comunicazione sicura(cifrata),autenticazione dei client,meccanismo di controllo di accesso. SISTEMA OPERATIVO: -separazione tra kernel space e user space(i programmi girano in user space e il sistema operativo in kernel space,le richieste di accesso a risorse passano attraverso il SO);-separazione tra processi(un processo non può leggere i dati di un altro,solo in SO avanzati). TECNOLOGIE DI AUTENTICAZIONE: La scelta della tecnologia di autenticazione è cruciale.Oltre a user e password si può implementare un'autenticazione basata sull'host, sugli oggetti fisici,oppure un'***autenticazione biometrica***: usa caratteristiche fisiche personali come impronte digitali,faccia,voce.. Pro e contro: non possono essere dimenticate né perse ,richiedono HW costoso,sono uniche ma non segrete, non sono revocabili e problemi di privacy.

-Domande iniziali utili in fase di progettazione: -sulle vulnerabilità: cosa può andar male,cosa si sta cercando di proteggere,chi pensiamo possa tentare di compromettere la sicurezza,qual è il punto più debole della nostra difesa; -sulle risorse:abbiamo a disposizione un'architettura sicura,abbiamo accesso a librerie di SW riutilizzabile,quali linee guida e standard abbiamo a disposizione,esistono esempi simili a cui ispirarsi; -sul SW: a che punto della catena di trust è il SW,esistono applicazioni critiche successive che fanno affidamento sul SW per l'autenticazione,esistono librerie a livello superiore che potrebbero fornire input non affidabili,chi sono gli utenti legittimi,chi ha accesso al SW(sorgente o eseguibile),si prevede un aumento o diminuzione degli utenti in futuro,che impatto questo cambiamento avrebbe sulle assunzioni iniziali,qual è l'ambiente in cui il SW viene eseguito. –sui goals:qual è l'impatto che avrebbe un attacco alla sicurezza,quali attività del SW(efficienza,accessibilità,riuso..) vengono compromesse dalle misure di sicurezza adottate,se l'utente ignora o viola le misure di sicurezza come dovrebbe reagire il sistema e come vengono rilevate queste violazioni.

-Livello di sicurezza:Il grado di sicurezza richiesto è connesso al rischio ed al costo delle contromisure.Rendere l'applicazione il piu sicura possibile puo causare il fallimento della progettazione.Occorre identificare costi ed esplicitare possibili compromessi.

-Identificazione delle assunzioni:E' una fase molto critica della sicurezza.Esempio assunzioni sull'autenticazione: -Shared Secret per autenticare componenti SW; -metodi biometrici per sostanze umane o sangue.

-Progettare con in mente il nemico(Adversary principle):Progettare il SW come se il nemico più astuto lo attaccherà,anticipando gli attacchi di avversari intelligenti,razionali e irrazionali e prevedendo le possibili solution.E' impossibile progettare con in mente il nemico senza il realistico senso di cosa il nemico potrebbe voler attaccare del SW.

-Conoscere e rispettare la chain of trust: Non invocare programmi non fidati da programmi fidati(spesso violato quando un'applicazione invoca un comando di sistema per eseguire un'operazione).Un programma non deve delegare l'autorità di fare un'azione senza delegare anche la responsabilità di controllare se l'azione è appropriata.Un'applicazione rispetta la chain of trust se:valida tutto quello che riceve in input,non passa tasks a identità meno-fidate,emette informazioni tanto valide e sicure quanto quelle prodotte dalle altre risorse.

-Privilegi adeguati: Un'applicazione necessita di adeguati privilegi e accessi per operare.Seguire il *principio del minimo privilegio*:per leggere un record non aprire il file con accesso anche in scrittura,per creare un file in una directory condivisa fare uso del "group access" o delle ACL.

-Test delle azioni contro la policy: Il *principio di mediazione* completa dice che occorre testare,passo per passo,ogni azione che è possibile tentare contro la policy.La mediazione completa è necessaria a garantire che le decisioni prese in un dato momento dal SW sono conformi al settaggio di sicurezza aggiornato del sistema.

-Offuscamento delle informazioni:Il *principio del non offuscamento* dice che nascondere la modalità di funzionamento di un componente SW o il registro in cui è memorizzato un particolare parametro di policy è pericoloso.La segretezza non è affidabile come unico mezzo di difesa,ma puo essere utile per ingannare un attaccante.

-Mantenere lo stato minimale: Stato=informazione che un programma deve tenere in memoria durante una transazione o una esecuzione di un comando.Il principio del *minimal retained state* dice che un programma deve tenere in memoria lo stato minimale cosi da rendere difficile ad un utente malizioso il poter modificare le variabili di stato o creare stati di programma non permessi,consentendo in tal modo azioni di programma anomale.

-Livello di fault tolerance:Per avere certificazione CERT un apparato deve: *identificare le funzionalità critiche*(uso delle 3 R):-Resistance,la capacità di impedire un attacco;-Recongnition,la capacità di riconoscere attacchi e la misura del danno;-Recovery,la capacità di fornire servizi ed assetti essenziali durante un attacco e ripristinare tutti i servizi dopo l'attacco; *valutare il ruolo del SW nel contesto della fault tolerance and continuity planning della compagnia*.

-Appropriato Error Handling:Molti sistemi vengono compromessi a causa di improprio trattamento di errori inaspettati.Una corretta pianificazione dell'error handling coinvolge:l'architetto che decide la progettazione generale del trattamento di errori;il designer che decide come l'applicazione rileva un fallimento,come discrimina tra vari casi ed il meccanismo di risposta;il programmatore che cattura le condizioni di decision/triggering e realizza il design;l'operatore che controlla se i processi critici vengono interrotti e opportuni messaggi appaiono in console o vengono generati file di log.

-Graceful degradation:Quando si verifica un guasto il sistema non si ferma ma continua ad operare in modo ristretto o con funzionalità ridotte.

-Fallimento sicuro:Il principio di *fault safely* dice che in caso di fallimento un sistema deve terminare in una configurazione sicura(esempio:se fallisce il programma di una porta a chiusura elettronica la porta deve rimanere aperta)
-Misure di sicurezza adeguate all'utente:Il principio dell'*accettabilità psicologica* dice che è importante usare modelli mentali e paradigmi tratti dal mondo reale.Se le misure di sicurezza di un sistema sono cosi onerose ed irritanti che l'utente le evita la sicurezza del sistema è compromessa.
-Responsabilità individuali: Il principio di *accountable individuals* dice che un'architettura di successo deve garantire che gli individui siano responsabili delle proprie azioni.Questo principio richiede che ogni utente abbia ed usi un account personale,sia difficile per un utente spacciarsi per un altro,deve essere chiaramente stabilita la responsabilità della sicurezza delle risorse coinvolte.
-Limite sulle risorse:Il principio di *resource-consumption limitation* dice che si devono usare le funzionalità che il SO fornisce per limitare il consumo di risorse del sistema.La limitazione del consumo di risorse deve essere combinata con un significativo ripristino da errore(error recovery).
-Ricostruzione di eventi:Il *principio di auditability* dice che deve essere possibile ricostruire la sequenza di eventi che hanno condotto certe azioni chiave.Questo principio prevede che l'applicazione e il sistema di host creino e mantengano gli audit logs.
-Livelli multipli di difesa:Fornire difesa in profonidità è meglio che affidarsi ad un'unica barriera(esempio:richiedere ad un utente di avere permessi appropriati e password prima di accedere ad un file)
-Monolicità di una applicazione:-garantire sicurezza ad ogni fase dello sviluppo;-nell'analisi delle minacce e della chain of trust considerare l'intera collezione di applicazione SW interoperanti,il supporto SW,la connessione di rete,e l'HW;-un insieme di decisioni di design ragionevoli puo combinarsi in modo irragionevole.
-Riuso di SW sicuro:Molte corporazioni di grandi dimensioni hanno politiche di riuso del codice e repository di codice certificato sicuro.Esperti di sicurezza dibattono da anni se è piu sicuro il SW di dominio pubblico(SW open source) oppure il SW proprietario(SW closet source). Non assumere che un programma od algoritmo sia sicuro da attacchi perché pubblicamente disponibile.
-Semplicità di progettazione:Un principio base dell'ingegneria sicura è realizzare sistemi che siano piu semplici possibile.Sistemi semplici sono piu facili da progettare e testare.
-Modularizzazione:Il *principio di modularizzazione* dice di:suddividere la progettazione in moduli,definire con precisione i punti d'interfaccia tra moduli,limitare privilegi e risorse ai moduli che realmente ne necessitano.Funzioni che richiedono privilegi speciali vengono isolate e codificate in separati semplici moduli.
-Tecniche ingegneristiche:Le tecniche di sviluppo sono un fattore critico per la sicurezza del SW.Software sicuro richiede buona progettazione e buone tecniche di progettazione.Molti attacchi sono dovuti a mancanza di progettazione,debolezze umane e scarsa capacità di programmazione.Una buona architettura di sicurezza può eliminare o mitigare il primi 2 fattori ma non il terzo.
-TCP SYN FLOODS: Dall'analisi degli attacchi SYN flood del protocollo TCP,i seguenti principi di progettazione sono stati violati:progettare con in mente il nemico,garantire un adeguato livello di fault tolerance,il principio di graceful degradation,self-controllo del consumo di risorse.
-Caso di studio javaSandBox:I principi su cui si basa l'architettura di java sono: sicurezza dei tipi del linguaggio,verifica statica del bytecode,controllo runtime dei permessi e accesso mediato alle risorse.In JDK 1.0 le applet non possono accedere a risorse locali e vengono eseguite in un SendBox e le applicazioni possono fare tutto. In JDK 1.1 le applet firmate sono trattate come le applicazioni.In Java 2 l'architettura sono piu sensibili e personalizzabili.
-Macchine a stati finiti: Una macchina a stati finiti(FSM) è una notazione formale che permette la rappresentazione astratta del comportamento del sistema.Le FSM hanno una rigorosa definizione matematica e una intuitiva rappresentazione grafica tramite diagrammi di stati.Si evince che le FSM modellano le configurazioni istantanee di un sistema tramite stati (nodi) e le operazioni del sistema tramite transizioni(archi).Le operazioni possono ricevere input (FSM di base) ed eventualmente produrre output(FSM estese).Sono utilizzate per modellare GUIs,protocolli di rete,applicazioni web,..Non tutti i requisiti pero possono essere specificati con una FSM,tra i quali requisiti real time,requisiti riguardanti performance,e requisiti riguardanti tipi di computazioni.Il modello FSM di un sistema non è unico.Una FSM è una tupla(S,I,δ) con S insieme finito di stati,I insieme finito di eventi di input e $\delta=S*I \rightarrow S$: funzione di transizione.
-Macchine a stati finiti con output:Macchina di Mealy se è una FSM che produce un output per ciascuna transizione;Macchina di Moore se è una FSM che produce un output per ciascun stato.La macchina di Mealy è una tupla(S,I,O,δ,γ) con S insieme finito di stati,I insieme finito di eventi di input,O insieme finito di eventi di output, $\delta=S*I \rightarrow S$: funzione di transizione, $\gamma=S*I \rightarrow O$ funzione di output.Spesso è anche fissato lo stato iniziale $s_0 \in S$.Una macchina di Moore è una tupla (S,I,O,δ,γ,F) con S insieme finito di stati,I insieme finito di eventi di input,O insieme finito di eventi di output, $\delta=S*I \rightarrow S$: funzione di transizione, $\gamma=S*I \rightarrow O$ funzione di output e $F \subseteq S$ insieme di stati finali.Un evento di output puo anche essere una azione della macchina.E' utile vedere una FSM di mealy come la tupla (S,I,O,T) con con S insieme finito di stati,I insieme finito di eventi di input,O insieme finito di eventi di output e T insieme finito di transizioni.Una transazione è una tupla(s,i,o,s').
-Limiti dell'FSM:E' possibile rappresentare solo un numero finito di stati.La composizione di piu FSM è una FSM con K1*k2*… stati(crescita esponenziale).Per superare i limiti di composizionalità dell'FSM sono state definite opportune estensioni(tra cui le statecharts di UML) che sono dotate dei concetti di sottomacchina e permettono composizione sequenziale,composizione parallela e modularità.
-Proprietà delle specifiche:La descrizione informale di un sistema SW è data in termine di REQUISITI. I requisiti specificano cosa un sistema SW deve fare,e non come deve farlo.
-Requisiti:Requisiti funzionali:specificano una funzione che il sistema deve compiere;Requisiti non funzionali(solitamente globali):Prestazioni,affidabilità,efficienza,portabilità,modificabilità;Requisiti inversi(relativi alla sicurezza):specificano operazioni che il sistema non deve fare;Specifiche di progetto/implementazione:legati alla tecnologia(es.www,XML,java).
-Specifica dei requisiti:La specifica dei requisiti è una descrizione completa e non ambigua dei requisiti.Si fa attraverso un'analisi iterativa e cooperativa del problema,richiede l'impiego di linguaggi formali o semiformali e il controllo della comprensione del problema che si era aggiunta.
-Qualità delle specifiche: -*Chiarezza* : la specifica deve descrivere quanto più chiaramente possibile i termini e le operazioni coinvolte.-*Non ambiguità*: la specifica non deve generare interpretazioni ambigue.-*Consistenza*: la specifica non deve contenere contraddizioni.-*Completezza*: il processo descritto dalla specifica deve essere definito in modo completo e dettagliato.Si divide in :

completezza interna, la specifica deve definire ogni concetto nuovo e ogni terminologia usata(definire un glossario) e *completezza esterna*, che è la completezza rispetto ai requisiti; *Incrementalità*: la specifica viene sviluppata in più passi successivi;*Comprensibilità*: la specifica, in quanto contratto tra committente e produttore, deve essere intuitiva e comprensibile per il cliente.

-Linguaggi per la specifica: -Informali: linguaggio naturale; Semi-Formali(UML): possibilmente grafici; -Formali: formalismi operazionali(definiscono il sistema descrivendone il comportamento come se fosse eseguito da una macchina astratta), formalismi dichiarativi(definiscono il sistema dichiarando le proprietà che esso deve avere).

-Formalismi Operazionali: L'approccio operazionale fornisce una rappresentazione più intuitiva poiché più simile al modo di ragionare degli umani(facilità di realizzazione e di validazione)

-Formalismi Dichiarativi: L'approccio dichiarativo fornisce una rappresentazione che non si presta ad ambiguità, ma è più difficile da comprendere e sviluppare(facilità di verifica).

-Metodi Semi-Formali UML : definisce una notazione grafica, un complesso di viste organizzate in diagrammi, una sintassi mediante meta-modello, una semantica descritta in prosa.

-Viste in UML: - *Use-Case View*: mostra le funzionalità che il sistema dovrebbe fornire così come sono percepite da attori esterni(Black box view).- *Logical View*: analizza l'interno del sistema e descrive come le funzionalità sono fornite (White box view); -*Component View*: mostra l'organizzazione e le dipendenze delle componenti computazionali del sistema; -*Deployment View* :descrive l'allocazione delle parti del sistema SW in una architettura fisica.

-Proprietà delle macchine a stati finiti:

-***Equivalenza di stati:*** due stati di una FSM sono equivalenti se per ogni sequenza di input, le sequenze di output dai 2 stati sono le stesse;

-***Stati distinti:*** 2 stati di una FSM che non sono equivalenti si dicono distinguibili;

-***Minimalizzazione:*** una FSM è minimizzata (o ridotta) se non possiede coppie di stati equivalenti. Esistono algoritmi standard per minimizzare le FSM facendo collassare stati equivalenti in un unico stato;

-***Macchine equivalenti:*** 2 FSM, M1 e M2, si dicono equivalenti se per ogni stato s di M1 esiste uno stato s' di M2 tale che s ed s' sono equivalenti e se per ogni stato s di M2 esiste uno stato s' di M1 tale che s ed s' sono equivalenti.Macchine non equivalenti sono dette distinguibili;

-***K-Equivalenza:*** Due stati si dicono k-equivalenti quando per ogni sequenza di input di lunghezza k le sequenze di output dai 2 stati sono le stesse. 2 stati non k-equivalenti si dicono k-distinguibili;

-***FSM completamente specificate:*** una FSM (S,I,O,T) è completamente specificata se per ogni stato s ed input i,esiste un output o ed uno stato s' tali che (s,i,o,s') è una transizione in T. Come completare FSM non completamente specificate:se non c'è una transizione dello stato s con input i,per *transazioni "implicitamente definite"* bisogna aggiungere una loop transition con output null(s,i,null,s);per *transazioni "indefinite per default"* bisogna aggiungere un insieme di transizioni ((s,i,o,s')|o€O,s'€S);per *transazioni "proibite"* l'input i non puo essere introdotto allo stato s,e la macchina non puo essere completamente specificata.

-***FSM deterministiche***:Una FSM (S,I,O,T) è deterministica se per ogni stato s e input i,esistono al piu un output o ed uno stato s' tali che (s,i,o,s') è una transizione in T.

-***FSM fortemente connesse***: Una FSM (S,I,O,T) è fortemente connessa se per ogni coppia di stati (s,s') esiste una sequenza di input che porta dallo stato s allo stato s'.

-***Inizializzazione***:Una FSM (S,I,O,T) è inizializzata se esiste uno stato s_init in S designato come stato iniziale.

-Synchronizing sequences: Una sequenza di eventi di input e detta una Synchronizing sequences se la sequenza di eventi di input conduce sempre allo stesso stato finale indipendentemente dallo stato da cui si parte; inoltre le Synchronizing sequences sono indipendenti dall'output.

-Homing sequenze: Una sequenza di eventi di input e detta una Homing sequenze se dopo aver applicato la sequenza degli eventi di input, possiamo determinare lo stato finale osservando la sequenza degli eventi di output. Ogni sequenza di output individua univocamente stato iniziale e stato finale. Una sequenza di homing è anche una sequenza di sincronizzazione e non viceversa.

-Distinguishing sequenze(Incertezza stato iniziale): Una sequenza di eventi di input è detta una distinguishing sequence se dopo aver applicato la sequenza degli eventi di input, possiamo determinare lo stato sorgente osservando la sequenza degli eventi di output.

-UIO Sequenze(State Verification): Una sequenza di eventi di input è detta una UIO (Unique input/output) sequence se dopo aver applicato la sequenza degli eventi di input, possiamo determinare se lo stato sorgente è uno specifico stato s osservando la sequenza degli eventi di output.

-EFSM (Macchine a stati finiti estese): Le EFMS estendono le FMS grazie al concetto di variabile. Una EFMS è un tupla composta (S,I,O,V,T) dove: S, indica l'insieme finito di stati; I, insieme finito di eventi di input; O, insieme finito di eventi di output; V, insieme finito di variabili; T, insieme finito di transazioni. Una transazione a sua volta è una tupla (S,I,O,G,A,S') dove G, è il predicato sulle variabili V, detto guardia; A assegnamento ad una variabile, detto azione; ed S' lo stato target.

-Stato Globale EFSM: Per una EFSM (S,I,O,V,T), una coppia (s, σ) è detta stato globale se: s è uno stato, e σ è una valutazione su V.

-Transizione globale EFSM: Per una EFSM (S,I,O,V,T), una tupla ((s, σ),i,o,(s', σ')) è detta una transizione globale se esiste una transizione (s,i,o,g,a,s') tale che: σ soddisfa la guardia g e σ'(v) = σ (exp) dove l'azione a è v:=exp.

-Grafo di raggiungibilità EFSM: Il grafo di raggiungibilità di una EFSM (S,I,O,V,T) è un grafo direzionato in cui: i nodi sono stati globali e gli archi sono le transizioni globali. Se le variabili hanno range finito, il grafo di raggiungibilità di una EFSM è una FSM.