Capitolo 7

Laboratorio virtuale

7.1 Il laboratorio virtuale di Tecnologie per la sicurezza

7.1.1 Motivazioni

Per acquisire una solida conoscenza nel settore ingegneristico non è sufficiente ascoltare lezioni e leggere dispense. E', infatti, anche indispensabile verificare di persona ciò che si appreso, sia indagandone le implicazioni ad un livello d'astrazione più basso di quello necessariamente usato dal docente, sia sperimentandone l'applicabilità in casi che lui non ha trattato.

Questa attività di *learning by doing* richiede la disponibilità di appositi strumenti, da un lato simili, anche se più semplici, di quelli che serviranno nell'attività professionale, da un altro lato molto diversi, dovendo rendere visibili informazioni di dettaglio sulle prestazioni dei componenti di un sistema complesso, sicuramente molto formative per chi impara il mestiere del progettista, ma anche inutilmente pesanti per chi lo esercita già.

La realizzazione fisica di laboratori ove svolgere tale forma di apprendimento ha nel passato trovato forti vincoli di costo, di tempo e di spazio. Oggi le tecnologie ICT, i progetti open source ed il dato di fatto che quasi tutti gli studenti hanno un loro PC hanno creato i presupposti per risolvere il problema delle esercitazioni con laboratori virtuali indirizzati alla verifica sperimentale di specifiche discipline.

7.1.2 Tecnologia, obiettivi formativi e funzionalità

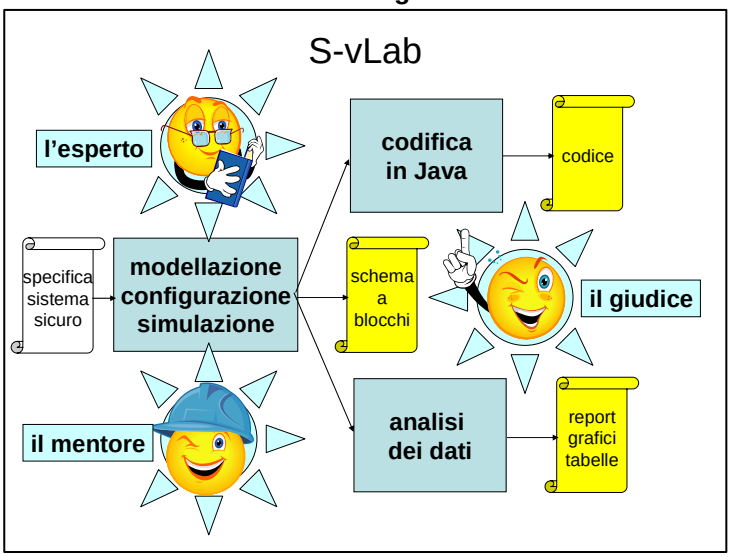
Il laboratorio virtuale S-vLab (*Security Virtual Laboratory*)[64] è un software scritto in Java (per poter essere installato su qualsiasi piattaforma), è organizzato a plug-in (per presentare spiccate caratteristiche di modularità ed estendibilità), è fortemente basato sulla tecnologia open source della Comunità di Eclipse (per potersi avvalere dei complessi servizi di esecuzione e cordinamento dei processi da essa sviluppati), è reso disponibile sulla piattaforma di e-learning AlmaChannel (per usufruire dei servizi di autenticazione, di cooperazione e di condivisione da essa offerti) e, facendo riferimento al modello del blended learning, realizza la componente asincrona dell'apprendimento delle nozioni impartite in presenza nell'insegnamento di Tecnologie per la sicurezza.

S-vLab persegue quattro obietti formativi:

1. rendere familiare la programmazione in Java[65] di meccanismi e di servizi per la sicurezza informatica;
2. rafforzare la conoscenza sul come deve essere organizzato un sistema sicuro;
3. evidenziare gli oneri computazionali degli algoritmi e dei protocolli per la sicurezza;
4. fare toccare con mano l'effetto di alcuni attacchi intenzionali.

Lo studente, secondo l'approccio costruttivista, è il vero protagonista degli esperimenti che eseguirà in laboratorio: egli infatti è parte attiva dapprima nella scelta degli obiettivi formativi, poi nella definizione dell'esperimento, poi ancora nel modo con cui condurlo ed infine nella interpretazione dei dati che ha così ottenuto.

Per rendere possibile questo modo di operare, S-vLab offre allo studente la possibilità di avvalersi liberamente di **tre differenti strumenti** e la possibilità di eseguire al meglio l'attività sperimentale dialogando continuamente con **tre differenti figure virtuali** di docente.



Il primo strumento consente di **modellare**, **configurare** e **simulare** un sistema sicuro. Il secondo strumento fornisce supporto all' **analisi dei dati** ottenuti dalla simulazione. Il terzo strumento offre tutte le funzionalità necessarie per **codificare** il sistema allo studio.

La figura virtuale dell'**esperto** aiuta nella consultazione di manuali, di standard e di buone pratiche.

La figura virtuale del **mentore** notifica ogni errore commesso nell'uso degli strumenti e suggerisce come eliminarlo.

La figura virtuale del **giudice**, impiegando un "profilo" dello studente continuamente aggiornato, suggerisce l'esperimento da fare in laboratorio, osserva e giudica come viene svolto e valuta quanta nuova conoscenza viene così acquisita.

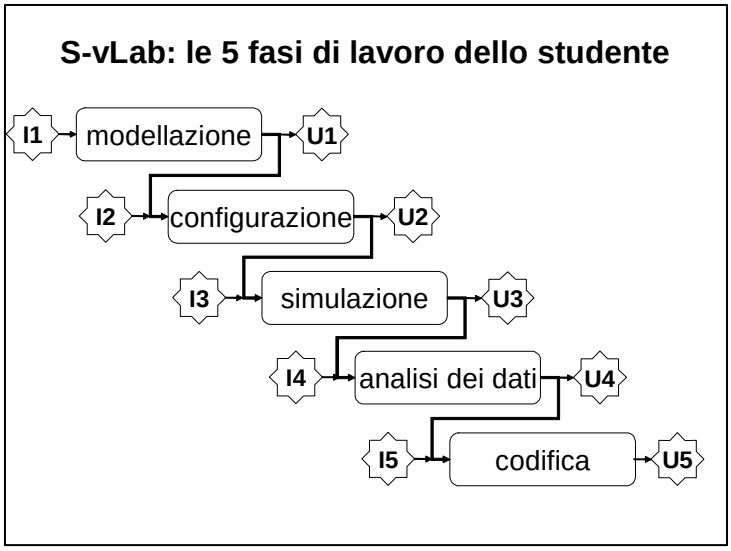
[64] L'attuale versione (S-vLab 2.0) recepisce ed amplia le funzionalità del precedente prototipo vLab 1.0, sperimentato con successo negli a.a. 2004-05 e 2005-06.

[65] Una futura versione consentirà anche l'uso di differenti linguaggi di programmazione.

7.1.3 Fasi di un esperimento e codice dei colori

Come devono essere adoperati gli strumenti di S-vLab?

Lo studente deve prima di tutto concordare con il giudice un suo obiettivo di apprendimento, tipicamente l'analisi del comportamento di un certo sistema sicuro al fine di verificarne una o più proprietà (integrità, confidenzialità, autenticità, disponibilità, ecc.).



Da queste specifiche (I1 in figura), prende le mosse una prima fase di lavoro, la **modellazione**, durante la quale lo studente deve costruire lo **schema a blocchi** (U1) del sistema a partire da un preassegnato insieme di componenti primitivi.

Nella fase di lavoro successiva ogni componente deve essere **configurato**, attribuendo valori specifici ai **parametri** che caratterizzano il suo comportamento: a questo punto il modello è completamente definito (U2) ed è possibile procedere ad esperimenti sul suo comportamento una volta che siano stati introdotti e configurati componenti che gli forniscono l'input, che ne raccolgono l'output e che simulano l'esecuzione di attacchi attivi.

Durante la terza fase, quella di **simulazione**, lo studente gestisce l'esperimento, osservando e facendo progredire passo passo il **flusso dei dati** all'interno del modello (U3). In generale i prefissati obiettivi di apprendimento richiedono l'esecuzione di più esperimenti, caratterizzati ciascuno da una appropriata ridefinizione dei dati d'ingresso e/o dei parametri dei componenti del sistema.

Il ruolo attivo dello studente nell'apprendimento si accentua durante le due ultime fasi di lavoro, di norma eseguite una dopo l'altra.

Durante la fase di **analisi dei dati** lo studente ottiene da S-vLab una raccolta completa dei risultati della simulazione e concentra la sua attenzione sul loro valore, per verificare o per estrapolare **leggi generali di comportamento** (U4) dei componenti di suo interesse.

Forte di questa conoscenza, un prerequisito indispensabile per chi dovrà progettare sistemi complessi, lo studente si può impegnare in una quinta ed ultima fase di lavoro: la **codifica** del sistema di cui ha definito la struttura e studiato il comportamento. Per aiutarlo, S-vLab gli fornisce la **lista di statement** Java (I5) che descrive tutte le attività di elaborazione richieste nel sistema, quali ha potuto via via individuare durante l'esecuzione delle precedenti fasi di modellazione, di configurazione e di simulazione. Aspetti salienti da affrontare in questa fase sono la decomposizione del listato in processi eseguibili su macchine diverse e la definizione delle loro modalità di comunicazione. Il risultato finale (U5) sarà dunque un insieme di file ".java" e ".class".

Quanto detto non deve essere necessariamente fatto in un'unica sessione di lavoro. S-vLab consente infatti allo studente di interrompere l'attività di laboratorio in qualsiasi momento e di riprenderla successivamente.

Un secondo importante aiuto è quello di rendere evidente sullo schermo in quale fase di lavoro ci si trova.

Stato di un componente e codice dei colori

Fase	Stato del blocco	Contorno	Input (se c'è)	Output (se c'è)
modellazione	presente	nero		
	con input	nero	nero	
	con output	nero	nero	nero
configurazione	configurato	rosso	nero	nero
simulazione	eseguibile	rosso/verde	rosso/verde	nero/verde
	eseguito	verde	verde	rosso
	considerato	verde	verde	verde
	riconfigurato	rosso	verde	nero

A tal fine S-vLab modifica via via il colore (o nero, o verde, o rosso) del contorno e delle connessioni di ogni componente.

Ciascuno di questi ultimi ha una **forma** (ottagonale per i meccanismi per la sicurezza, rettangolare per le operazione su stringhe di bit e circolare per gli elemento di input/output), un **nome** (una sigla seguita da un numero di istanza) e, in generale, delle **connessioni** di input e di output.

Ogni componente ha anche uno **stato** che individua le azioni su esso già fatte e quelle ancora fare. Come evidenziato nella tabella a lato, ad ogni stato corrisponde un colore del contorno e delle connessioni. L'evoluzione dello stato di ogni componente si può dedurre dalla seconda colonna, procedendo dall'alto verso il basso.

7.2 Modellazione

Si è già detto che S-vLab impiega l'astrazione dello schema a blocchi per descrivere la struttura del sistema sicuro di cui si intende analizzare il comportamento.

Prendiamo ora atto che alcuni modelli sono già disponibili sotto forma di **file**: in questo caso l'utente deve solo selezionare quello di suo interesse, attendere la comparsa dell'immagine nell'area di disegno e passare poi alla fase di configurazione.

Lo studente ha però anche la possibilità di costruire modelli diversi da quelli predisposti dal docente e più adatti a fargli conseguire suoi personali obiettivi formativi: a tal fine deve dapprima selezionare su una palette e trascinare nell'area di disegno, uno dopo l'altro, i componenti che gli interessano; successivamente deve stabilire come interagiscono all'interno del sistema che vuole analizzare, connettendone opportunamente i punti d'ingresso e d'uscita. Un comando di "salva con nome" consente di memorizzare su un file quanto si è disegnato sullo schermo, fermo restando il fatto che la struttura può essere modificata durante le successive fasi di lavoro.

Nei successivi paragrafi, con frasi e con esempi, vengono fornite informazioni più dettagliate sugli strumenti disponibili e sulle loro modalità d'uso.

7.2.1 La palette e l'area di disegno

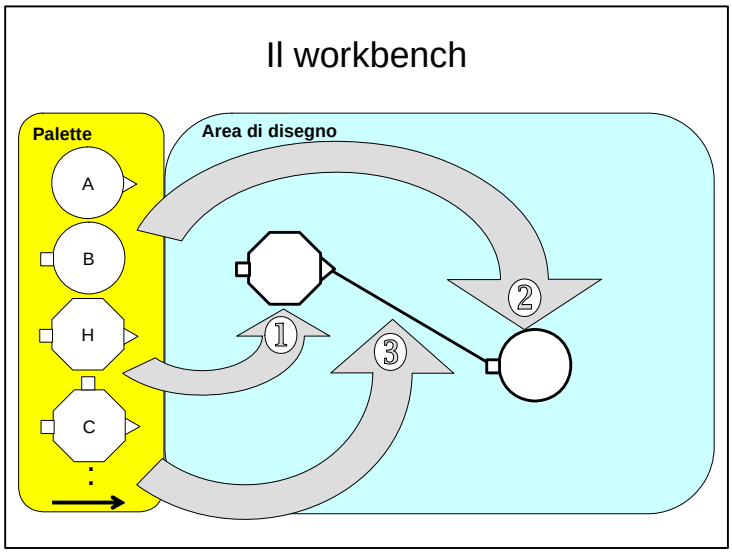
La palette e l'area di disegno appaiono al centro dello schermo dell'utente ed hanno come supporto l'editor grafico GEF (*Graphical Editing Framework*) di Eclipse.

La palette contiene i simboli grafici dei principali **meccanismi crittografici** (*PRNG, Generatore di chiavi, Hash, Cifrario, Firma digitale*) e dei **personaggi** (*Alice, Bob, Eva*) che tradizionalmente svolgono le funzioni di input, di output e di attacco in un sistema sicuro.

Per consentire indagini ad un livello di astrazione più basso, la palette contiene anche i simboli grafici delle **operazioni** tipicamente chiamate in causa dagli algoritmi crittografici (*Confronto bit a bit, Somma modulo due, Esponenziazione modulare, Memorizzazione, Conversione di tipo di dato*).

La palette contiene infine i simboli grafici di due comandi:

- la **freccia**, che consente di tracciare una connessione tra un punto d'uscita di un componente ed un punto d'ingresso di un altro,
- la **BOX**, che consente di racchiudere al suo interno uno schema di cui occorre iterare molte volte la simulazione o che si vuole considerare come un componente in un più alto livello di astrazione.



La figura mostra l'inizio di una fase di modellazione: l'utente seleziona con un click uno dei componenti della palette e lo trascina nel punto desiderato dell'area di disegno, ove rimane selezionato fino a quando non si clicca su un altro punto del workbench. Ad ogni componente trascinato nell'area di disegno S-vLab attribuisce lo stato "**presente**" ed il colore nero al contorno del suo simbolo grafico.

L'ordine con cui i componenti vengono trascinati nell'area di disegno e la loro disposizione sono scelti liberamente dall'utente.

Per tracciare una connessione, l'utente deve dapprima selezionare la freccia sulla palette, poi fare click sull'uscita (il simbolo è un triangolino) del componente che invierà il dato ed infine fare click sull'ingresso (il simbolo è un quadratino) di quello che lo dovrà ricevere.

Una volta che tutti i punti d'ingresso di un componente sono statti connessi a nodi d'uscita di altri componenti (le connessioni sono di colore nero), S-vLab gli assegna lo stato "**con input**" ed il suo contorno non presenta modifiche di colore. La connessione dei suoi nodi d'uscita con nodi di input di altri componenti (tratti neri) porta il componente nello stato "**con output**", ancora senza modifiche di colore.

7.2.2 Le viste

Due "viste", sincronizzate con l'editor grafico, aiutano lo studente durante il lavoro di modellazione, di configurazione e di simulazione. Nella **vista del mentore** compaiono sia la segnalazione di eventuali errori commessi dall'utente, sia, quando possibile, il suggerimento per eliminarli. La vista include anche un pulsante, premendo il quale è possibile consultare la guida online di S-vLab.

La **vista delle proprietà** (*properties view*) si riferisce sempre al componente **attualmente selezionato** nell'area di disegno.

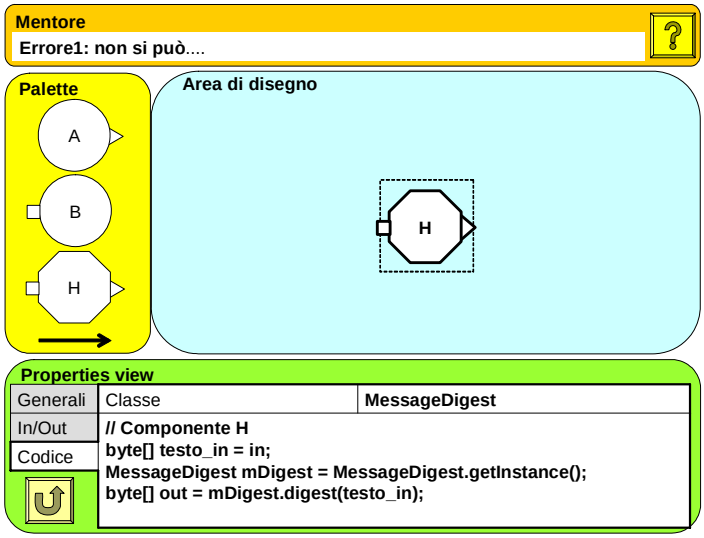
Le proprietà sono suddivise in tre schede (Generali, In/Out, Codice) che l'utente può visualizzare una alla volta cliccando sulla relativa etichetta.

La scheda delle proprietà **Generali** contiene il nome del componente, il numero d'ordine dell'istanza sull'area di disegno, lo stato in cui si trova. Tutti questi dati sono definiti automaticamente da S-vLab e normalmente non sono d'interesse per l'utente.

La scheda delle proprietà di **Input/Output** contiene il valore e le proprietà dei dati provenienti dalle connessioni con altri componenti, il valore dei parametri che l'utente ha digitato/selezionato per specificare il comportamento desiderato ed il valore e le proprietà dei risultati che S-vLab ottiene quando gli è richiesto di eseguire il codice Java associato al componente. Il valore dei dati introdotti dall'utente deve essere confermato premendo un apposito comando di "set". Questa scheda è d'interesse dell'utente durante le fasi di configurazione e di simulazione.

La scheda della proprietà **Codice** è selezionata come default in fase di modellazione e mostra la codifica in Java del comportamento del componente.

ESEMPIO N.1 (modellazione) **–** In figura è mostrato il primo passo della costruzione di un banco di prova per verificare il comportamento degli algoritmi di hash.



Una volta che il componente H è stato trascinato nell'area di disegno, l'utente può controllare sulla vista delle proprietà se è proprio quello che desiderava installare.

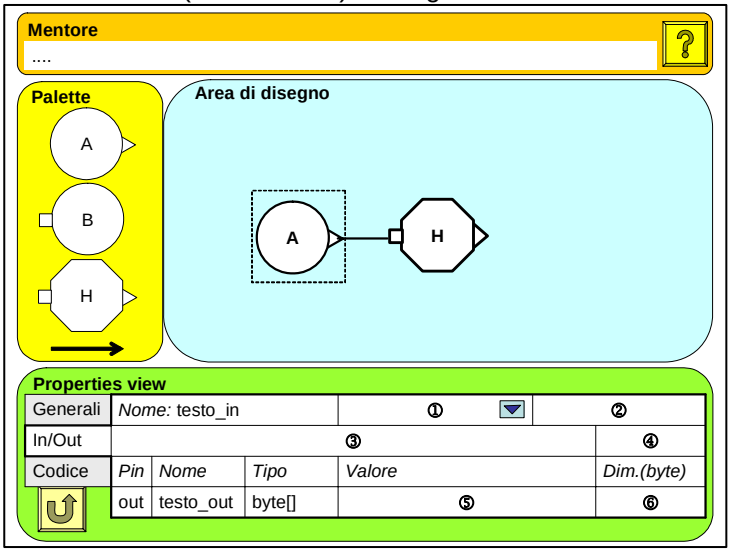
Un primo contributo all'apprendimento delle estensioni di Java per la sicurezza è fornito dalla scheda Codice. Si noti che il componente accetta e restituisce array di byte, che la classe di riferimento è *MessageDigest* e che l'invocazione del metodo *getInstance* richiedetà una opportuna introduzione di parametri all'interno delle parentesi.

N.B. Qualsiasi dubbio sull'uso di Java può essere risolto in ogni momento, chiedendo all'esperto di accedere al manuale on-line.

7.2.3 Il componente Alice

In Crittografia è Alice il personaggio che introduce in un sistema sicuro l'informazione da proteggere. S-vLab assegna tale compito al componente A, che sarà quindi presente in ogni modello.

ESEMPIO N.1 (modellazione) – In figura è mostrato il secondo passo della costruzione del banco di prova per algoritmi di hash: l'utente ha introdotto un componente A per disporre di un array di byte da fornire in ingresso a H.



In figura sono evidenziati i diversi campi della scheda In/Out di A.

Nel campo 1, tramite una combo-box, l'utente dovrà indicare se vuole ottenere il dato d'uscita a partire da una stringa o da un file.

Nel campo 2, se è stato scelto come sorgente il file, l'utente dovrà indicarne il percorso d'accesso.

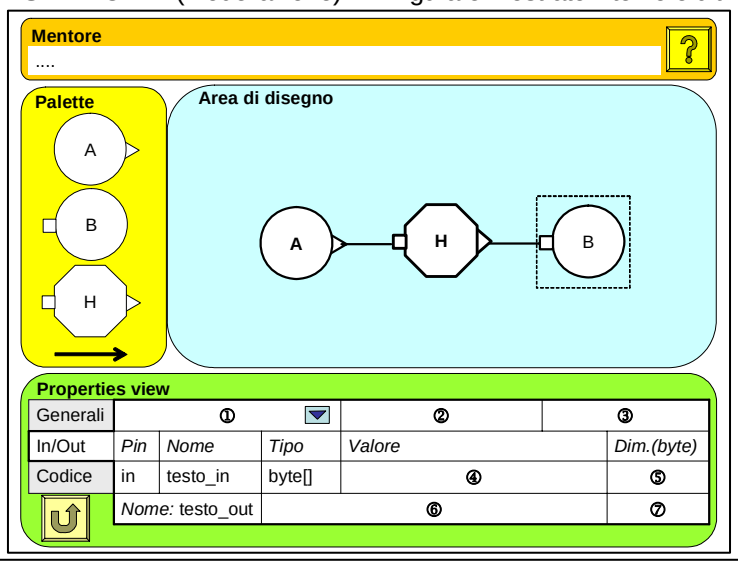
Il campo 3 è di digitazione del testo, se è stata scelta l'opzione stringa, di sola visualizzazione, se è stata scelta l'opzione file.

Nel campo 4 S-vLab scrive la dimensione (in caratteri nel caso della stringa, in byte nel caso del file) del testo_in; anche i campi 5 e 6 sono di competenza di S-vLab e conterranno, al momento dell'esecuzione del componente, il dato che verrà trasmesso ad H e la sua dimensione.

7.2.4 Il componente Bob

Il componente B ha il compito di convertire e poi visualizzare il dato generato da un meccanismo di sicurezza. Il ruolo è quello del personaggio Bob che la Crittografia prevede come destinatario dei messaggi inoltrati su un canale insicuro. Di norma ogni modello di S-vLab prevede un componente B, per consentire all'utente di prendere atto della trasformazione eseguita dal sistema sull'informazione immessa dal componente A.

ESEMPIO N.1 (modellazione) – In figura è mostrato il terzo e ultimo passo della costruzione del banco di prova per algoritmi di hash: l'utente ha ora introdotto un componente B per poter controllare a vista il valore dell'impronta calcolata da H. In figura è mostrata la scheda In/Out del componente B.



In fase di configurazione l'utente deve indicare nel campo 1, tramite una combo-box, se vuole solo visualizzare il testo_in sotto forma di stringa o se vuole anche salvarlo in un file; nel secondo caso, nei campi 2 e 3, deve essere indicata la modalità di scrittura ed il percorso d'accesso.

A seguito di un comando di esecuzione dato al componente H, S-vLab riempirà il campo 4 (testo_in come array di byte) ed il campo 5 (dimensione dell'array). Quando anche B sarà stato eseguito, il campo 6 conterrà la rappresentazione del testo_in voluta dall'utente ed il campo 7 la sua dimensione.

7.2.5 Il data base degli eventi

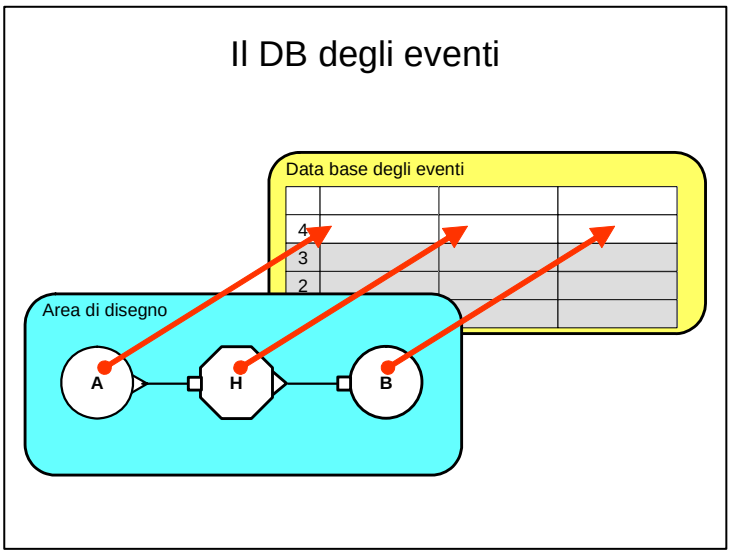
La raccolta ed interpretazione dei dati sperimentali non può essere basata sul solo componente B.

Un utente può, infatti, aver bisogno di sapere non tanto **il risultato di un singolo esperimento,** quanto la **sequenza di risultati** che il sistema gli fornisce in risposta ad una sequenza di modifiche apportate ai parametri di alcuni dei suoi componenti.

ESEMPIO – Il banco di prova per algoritmi di hash, modificando via via i parametri di A, consente di verificare la apparente indipendenza dei bit dell'impronta dai bit del messaggio da cui discende.

Lo stesso banco di prova, questa volta modificando i parametri di H, consente di mettere a confronto le prestazioni di algoritmi di compressione diversi e di provider diversi.

Consultando le schede di In/Out dei componenti al termine di ogni esperimento, è senz'altro possibile annotarsi su un foglio i risultati che si stanno cercando, ma è anche vero che questo modo di procedere può creare errori e scoraggiare l'utente.



Per semplificare il lavoro di **raccolta** dei dati sperimentali, S-vLab si preoccupa di salvare automaticamente, ad ogni comando di esecuzione, le proprietà di **tutti** i componenti dello schema a blocchi.

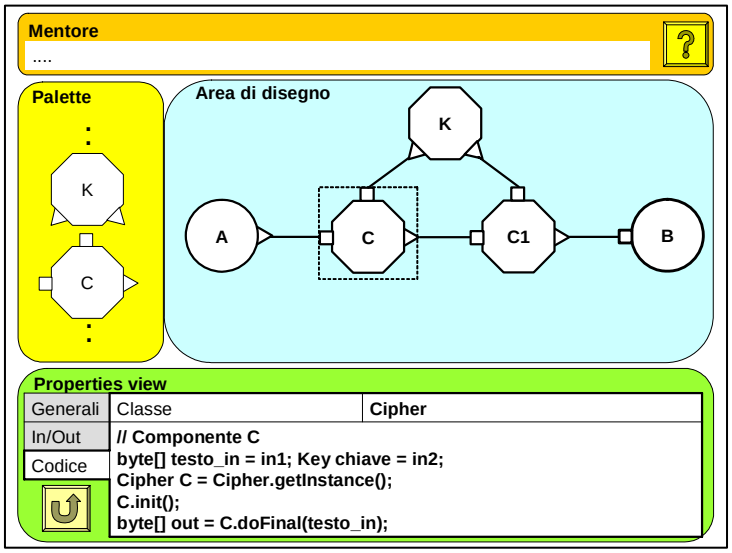
Per accedere a tale **data base degli eventi** (e ad una serie di strumenti di analisi), lo studente deve soltanto premere un pulsante e comunicare così a S-vLab la sua volontà di avviare la fase di analisi dei dati,.

Apposite **query** consentiranno all'utente di selezionare i dati di suo interesse ed appositi programmi gli forniranno, sotto forma di tabelle e di grafici, le correlazioni che sta cercando.

In 7.5 vedremo alcuni esempi, fermo restando che la cosa migliore per impadronisi di questo strumento è usarlo.

7.2.5 Casi di studio

ESEMPIO N.2 (modellazione) **-** La predisposizione e la configurazione di un banco di prova di un cifrario mette in luce un aspetto particolare di S-vLab. La palette prevede un unico componente primitivo **C** per proteggere la riservatezza delle comunicazioni tra Alice e Bob ed un unico componente **K** per definire la chiave in uso. Tale scelta semplifica la fase di modellazione (vedi figura a lato), ma richiede anche una particolare attenzione durante la fase di configurazione: occorrerà, infatti, configurare C in modo da fargli eseguire la **cifratura** del testo in chiaro fornito da A, C1 (la nuova sigla gli è stata attribuita quando è stato istanziato) in modo da fargli eseguire la **decifrazione** del testo cifrato e K in modo da rendere disponibili o due chiavi **identiche**, se il cifrario è simmetrico, o una chiave **privata** ed una **pubblica**, se è asimmetrico.

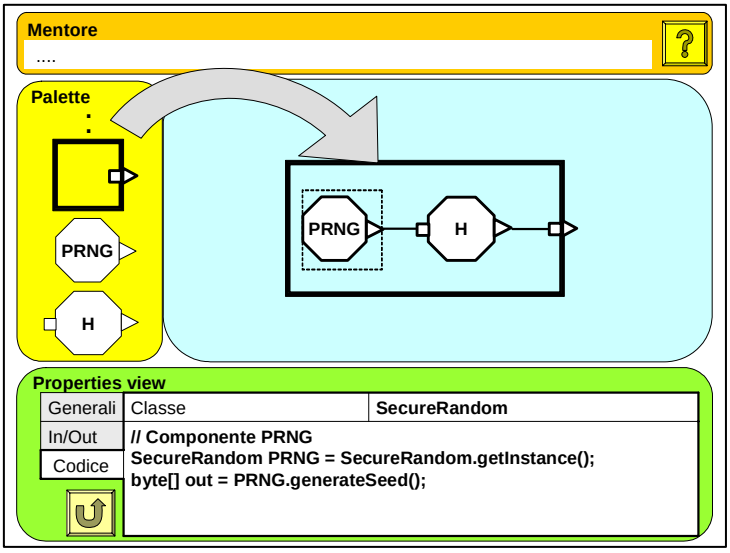


Si noti nella scheda Codice di C sia la necessaria numerazione degli ingressi prevista da S-vLab, sia la per ora incompleta invocazione dei metodi *getInstance* e *init*.

ESEMPIO N.3 (modellazione) - Per verificare se il comportamento degli algoritmi crittografici di compressione è proprio quello di un **oracolo casuale** occorre fare un'analisi statistica delle impronte generate da un componente H in risposta a moltissimi dati casuali inviatigli da un componente PRNG (si noti in figura il suo codice Java).

A differenza dell'esempio N.1, in questo caso è indispensabile inserire il banco di prova all'interno di una **BOX**.

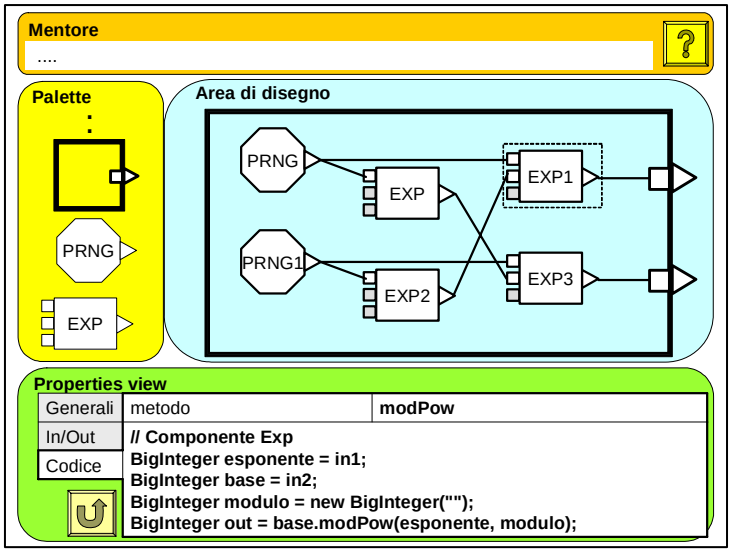
Con questo accorgimento, infatti, si può affidare interamente a S-vLab il compito di **iterare il numero di volte desiderato** (lo si dovrà indicare in fase di configurazione) la generazione del dato casuale ed il calcolo della sua impronta. L'uscita della BOX è un **convertitore di tipo di dato**: l'accorgimento è utile quando l'utente vuole immettere nel DB dati di tipo diverso da quello con cui sono stati generati all'interno della BOX.



ESEMPIO N.4 (modellazione) – Un differente impiego della BOX è mostrato in figura. L'intendimento è realizzare un nuovo componente che metta a disposizione due copie di una chiave segreta eseguendo uno scambio anonimo D-H.

Il modello richiede quattro componenti EXP (la sigla ricorda l'operazione di **esponenziazione modulare** che eseguiranno). I nodi di input di tale componente sono dedicati ad introdurre, a partire dal più in alto, l'esponente, la base ed il modulo. E' però possibile assegnare tali valori anche tramite properties view: in questo caso i relativi punti d'ingresso assumono un colore scuro.

In figura è mostrato il codice associato a EXP1. Si noti che è previsto di considerare il modulo come parametro e che il risultato è un *BigInteger:* per trasformarlo in *SecretKey* occorrerà quindi configurare il convertitore di tipo di dato sulle uscite della BOX.



7.3 Configurazione

Ogni componente di S-vLab corrisponde ad una funzione i cui argomenti sono specificati dal valore dei dati presenti sui suoi ingressi e dal valore dei parametri impostati dall'utente nella sua scheda di In/Out.

La fase di configurazione riguarda specificatamente questi ultimi. Per avviarla si deve selezionare con **un click** il componente ed accedere ai campi di scrittura previsti nella sua finestra di proprietà.

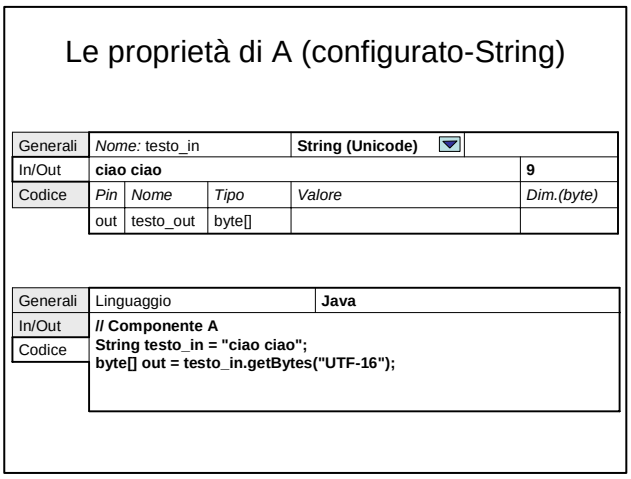
In alcuni casi i parametri devono essere digitati, in altri selezionati su liste accessibili tramite combo-box. Se l'utente ha dei dubbi sul modo di operare, può interrogare la figura virtuale del mentore; se invece ha dei dubbi sul significato e/o sul valore di un parametro, può chiedere l'aito dell'esperto premendo l'apposito pulsante. Il cambio temporaneo della prospettiva gli consente di chiedere accesso alla risorsa che gli viene indicata come la più appropriata, di consultarla e di esportare le informazioni che gli servono.

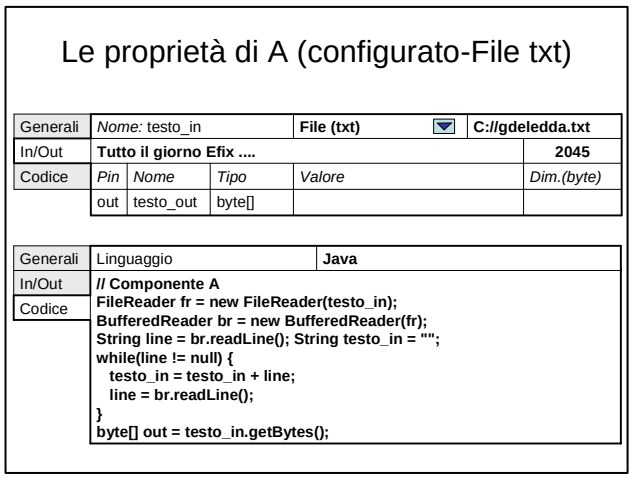
Una volta che i parametri sono stati tutti definiti, l'utente dà il **comando di set** ed il componente assume lo stato di "**configurato**" : il suo bordo diventa rosso, colore che segnala la possibilità di sottoporgli dati d'ingresso e quindi di renderlo "**eseguibile**": Fanno eccezione i blocchi privi d'ingresso (il blocco di input Alice, il PRNG, il Generatore di chiavi) per cui gli stati di configurato e di eseguibile coincidono. Al comando di set S-vLab aggiorna anche la scheda Codice, rendendo così visibile l'effetto nel listato delle scelte operate dall'utente.

L'ordine con cui i componenti vengono configurati è arbitrariamente definito dall'utente.

7.3.1 – Configurazione di A

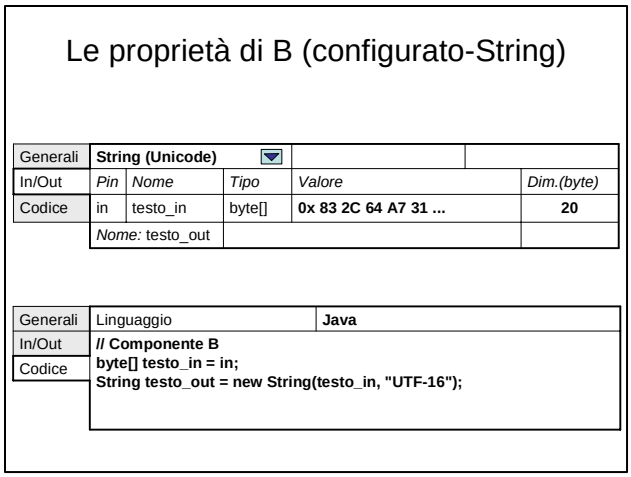
Le sottostanti figure evidenziano due alternative: sulla sinistra viene mostrata la decisione di fare emettere ad A una stringa di caratteri Unicode digitata dall'utente (scheda In/Out) e la corrispondente codifica in Java del comportamento (scheda Codice); sulla destra sono invece mostrate le schede In/Out e Codice nell'ipotesi di fargli emettere il contenuto del file di cui si è fornito il percorso.

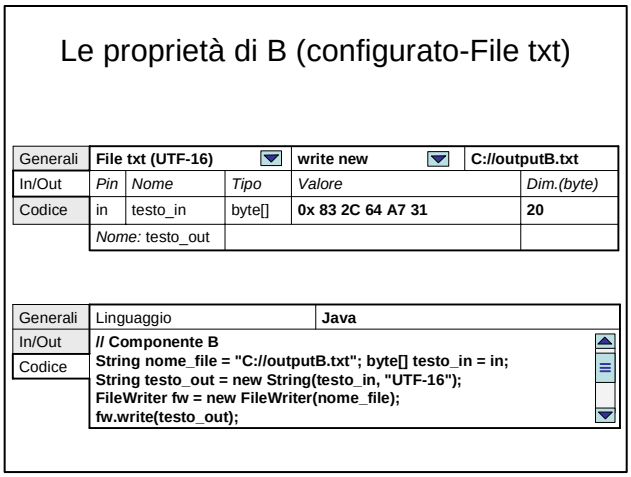




7.3.2 – Configurazione di B

Le sottostanti figure evidenziano due alternative: sulla sinistra è mostrata la scelta di rendere soltanto visibile l'input di B, sulla destra, quella di salvarlo anche su un file preesistente o generato al momento.





7.3.3 – Configurazione di H

Le proprietà di H (configurato)

Generali	Provider			**BC**		
In/Out	Algoritmo			**SHA-1**		
Codice	*Pin*	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	in	testo_in	byte[]	**0x A5 64 2C …**	**128**	
	out	digest	byte[]			

Generali	Classe	**MessageDigest**
In/Out	**// Componente H** **byte[] testo_in = in;** **MessageDigest mDigest = MessageDigest.getInstance("SHA-1","BC");** **byte[] out = mDigest.digest(testo_in);**	
Codice		

Per configurare il componente H (esempi N.1 e N.3) occorre selezionarlo sulla vista dell'editor grafico, aprire la scheda In/Out della vista delle proprietà, scegliere sulle liste mostrate nelle combo-box le sigle che individuano **Provider** e **Algoritmo** e dare il comando di set. In figura è mostrata la scelta del provider BC e dell'algoritmo SHA-1.

In figura si ipotizza anche che il componente a monte di H (A, se ci si riferisce all'esempio N.1) sia già stato non solo configurato, ma anche eseguito; la scheda In/Out di H evidenzia di conseguenza il valore del dato presente sull'ingresso *in*.

Si noti nella scheda Codice che S-vLab ha automaticamente inserito all'interno delle parentesi del metodo *getInstance* le stringhe "SHA-1" e "BC".

7.3.4 – Configurazione di K e di C

Nell'esempio N.2, una volta configurati A e B, bisogna occuparsi di K e di C.

Quando K è configurato come generatore della chiave segreta di un cifrario simmetrico (v. figura sulla sinistra) assume la sigla **KG** (KeyGenerator); quando invece genera la coppia chiave privata-chave pubblica di un cifrario asimmetrico (v. figura sulla destra) assume la sigla **KPG** (KeyPairGenerator).

Le proprietà di K (configurato – KG)

Generali	Provider			**BC**		
In/Out	Algoritmo / Dimensione chiave (bit)			**TripleDES**	**112**	
	Pin	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	out1	chiave_segr	SecretKey			
	out2	chiave_segr	SecretKey			

Generali	Classe	**KeyGenerator / KeyPairGenerator**
In/Out	**// Componente K** **KeyGenerator KG = KeyGenerator.getInstance("TripleDES", "BC");** **KG.init(112);** **SecretKey chiave_segr = KG.generateKey();** **SecretKey out1 = chiave_segr; SecretKey out2 = chiave_segr;**	
Codice		

Le proprietà di K (configurato – KPG)

Generali	Provider			**BC**		
In/Out	Algoritmo / Dimensione chiave (bit)			**RSA**	**512**	
	Pin	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	out1	chiave_priv	PrivateKey			
	out2	chiave_pub	PublicKey			

Generali	Classe	**KeyGenerator / KeyPairGenerator**
In/Out	**// Componente K** **KeyPairGenerator KPG = KeyPairGenerator.getInstance("RSA", "BC");** **KPG.initialize(512); KeyPair keypair = KPG.generateKeyPair();** **PrivateKey out1 = keypair.getPrivate();** **PublicKey out2 = keypair.getPublic();**	
Codice		

Nella figura a sinistra è mostrata una possibile configurazione di C come blocco di cifratura **simmetrica** (la sigla diventa **Es**); a destra è mostrata la configurazione richiesta da C1 per eseguire la funzione inversa (sigla **Ds**).

Le proprietà di C (configurato – Es)

Generali	Provider / Direzione			**BC**	**Cipher.ENCRYPT**	
In/Out	Algoritmo			**TripleDES** **ECB** **PKCS5Padding**		
Codice	*Pin*	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	in1	testo_in	byte[]	**0x 3B E5 81 4C...**	**1024**	
	in2	chiave	SecretKey			
	out	testo_out	byte[]			

Generali	Classe	**Cipher**
In/Out	**// Componente Es** **byte[] testo_in = in1; SecretKey chiave = in2;** **Cipher Es = Cipher.getInstance("TripleDES/ECB/PKCS5Padding", "BC");** **Es.init(Cipher.ENCRYPT_MODE, chiave);** **byte[] out = Es.doFinal(testo_in);**	
Codice		

Le proprietà di C1 (configurato – Ds)

Generali	Provider / Direzione			**BC**	**Cipher.DECRYPT**	
In/Out	Algoritmo			**TripleDES** **ECB** **PKCS5Padding**		
Codice	*Pin*	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	in1	testo_in	byte[]			
	in2	chiave	SecretKey			
	out	testo_out	byte[]			

Generali	Classe	**Cipher**
In/Out	**// Componente Ds** **byte[] testo_in = in1; SecretKey chiave = in2;** **Cipher Ds = Cipher.getInstance("TripleDES/ECB/PKCS5Padding", "BC");** **Ds.init(Cipher.DECRYPT_MODE, chiave);** **byte[] out = Ds.doFinal(testo_in);**	
Codice		

Nella figure sottostanti sono mostrate le configurazioni richieste da C e da C1 quando devono eseguire rispettivamente una cifratura ed una decifrazione **asimmetrica**: in questo caso le sigle diventano **Ea** e **Da**.

Le proprietà di C (configurato – Ea)

Generali	Provider / Direzione		**BC**		**Cipher.ENCRYPT**	
In/Out	Algoritmo		**RSA**	**None**	**PKCS1Padding**	
Codice	*Pin*	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	in1	testo_in	byte[]	**0x 3B E5 81 4C...**	**511**	
	in2	chiave	PublicKey			
	out	testo_out	byte[]			

Generali	Classe	**Cipher**
In/Out	**// Componente Ea** **byte[] testo_in = in1; PublicKey chiave = in2;** **Cipher Ea = Cipher.getInstance("RSA/None/PKCS1Padding", "BC");** **Ea.init(Cipher.ENCRYPT_MODE, chiave);** **byte[] out = Ea.doFinal(testo_in);**	
Codice		

Le proprietà di C1 (configurato – Da)

Generali	Provider / Direzione		**BC**		**Cipher.DECRYPT**	
In/Out	Algoritmo		**RSA**	**None**	**PKCS1Padding**	
Codice	*Pin*	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	in1	testo_in	byte[]			
	in2	chiave	PrivateKey			
	out	testo_out	byte[]			

Generali	Classe	**Cipher**
In/Out	**// Componente Da** **byte[] testo_in = in1; PrivateKey chiave = in2;** **Cipher Da = Cipher.getInstance ("RSA/None/PKCS1Padding", "BC");** **Da.init(Cipher.DECRYPT_MODE, chiave);** **byte[] out = Da.doFinal(testo_in);**	
Codice		

N. B. Anche il componente SIGN, qui non trattato per brevità, modifica la sua sigla dopo la configurazione. SIGN corrisponde alla classe astratta *Signature* e può essere configurato o per eseguire la firma di un testo con una chiave privata (in questo caso assume la sigla **S**) o per verificarla con la corrisponte chiave pubblica (in questo caso assume la sigla **V**).

7.3.5 – Configurazione di PRNG, di EXP e di BOX

Le proprietà di PRNG (configurato)

Generali	Provider		**BC**			
In/Out	Algoritmo		**SHA1PRNG**			
Codice	Tipo di seed		**generateSeed**			
	Dim. (byte)		**64**			
	Seed iniziale					
	Pin	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	out	prng_out	byte[]			

Generali	Classe	**SecureRandom**
In/Out	**// Componente PRNG** **SecureRandom PRNG =** **SecureRandom.getInstance("SHA1PRNG","BC");** **byte[] out = PRNG.generateSeed(64);**	
Codice		

Le proprietà di EXP (configurato)

Generali	*Pin*	*Nome*	*Tipo*	*Valore*	*n.° di "1"*	*Dim.(B)*	*Tempo (ms)*
In/Out	in1	esponente	BigInteger			**56**	
Codice	in2	base	BigInteger			**56**	
	in3	modulo	BigInteger	**132...**		**56**	
	out	out	BigInteger			**56**	

Generali	Linguaggio	**Java**
In/Out	**// Componente Exp** **BigInteger esponente = in1;** **BigInteger base = in2;** **BigInteger modulo = new BigInteger("132... ");** **BigInteger out = base.modPow(esponente, modulo);**	
Codice		

Negli esempi 3 e 4 occorre configurare un componente PRNG. Vediamo come si deve operare, ipotizzando al solito che il componente sia stato precedentemente selezionato.

Tramite le apposite combo-box della scheda In/Out l'utente deve scegliere provider, algoritmo e tipo di *seed*.

In un campo apposito l'utente deve anche digitare la dimensione in byte del dato casuale di cui chiede la generazione e questo senza dover mettere in conto che il sottostante algoritmo SHA-1 produce 20 byte alla volta.

Si noti nella scheda Codice che le stringhe dell'algoritmo e del provider vanno incluse tra parentesi nell'invocazione del metodo *getInstance*. La dimensione del dato casuale è invece un parametro del metodo *generateSeed*.

Nell'esempio N.4, pag.151, occorre configurare quattro componenti EXP. In figura si fa riferimento a EXP1: i valori dell'esponente e della base arriveranno sugli ingressi *in1* e *in2* quando saranno eseguiti i due componenti disposti a monte (PRNG e EXP2); il valore del modulo, per non complicare troppo il disegno, è in questo caso digitato dall'utente.

Tale scelta si ripercuote anche nel listato della scheda Codice. Sono infatti due soli gli assegnamenti di valore tra variabili esterne e variabili interne; il valore del modulo appare, infatti, come stringa nella dichiarazione *new BigInteger*.

Si noti che la variabile *out* è un *BigInteger*: per impiegarla come chiave occorrerà prevedere a valle una conversione di tipo.

Le proprietà di BOX (configurato)

Generali	Ciclo		**for**	*n.° iteraz.:* **400**
In/Out	*Pin*	*Nome*	*Tipo interno*	*Tipo esterno*
Codice	out1	box_out	**byte[]**	**byte[]**
		Valore:	*Dim:*	*Tempo:*

Generali	Linguaggio	**Java**
In/Out	**// Componente BOX** **SecureRandom PRNG =** **SecureRandom.getInstance("SHA1PRNG","BC");** **MessageDigest mDigest = MessageDigest.getInstance("SHA-1","BC");** **Vector temp = new Vector();** **for (int i=0; i<400; i++) {** **byte[] PRNG_out = PRNG.generateSeed(20);** **byte[] H_testo_in = PRNG_out;** **byte[] H_out = mDigest.digest(H_testo_in);** **temp.add(i, H_out); i++; }** **byte[] out1; temp.toArray(out1); byte[] box_out = out1;**	
Codice		

Presentando gli esempi N.3 e N.4 si è detto che anche il componente BOX deve essere configurato.

In figura si fa riferimento alla BOX che deve far compiere 400 esecuzioni consecutive alla coppia PRNG e H (esempio N.3, pag.151).

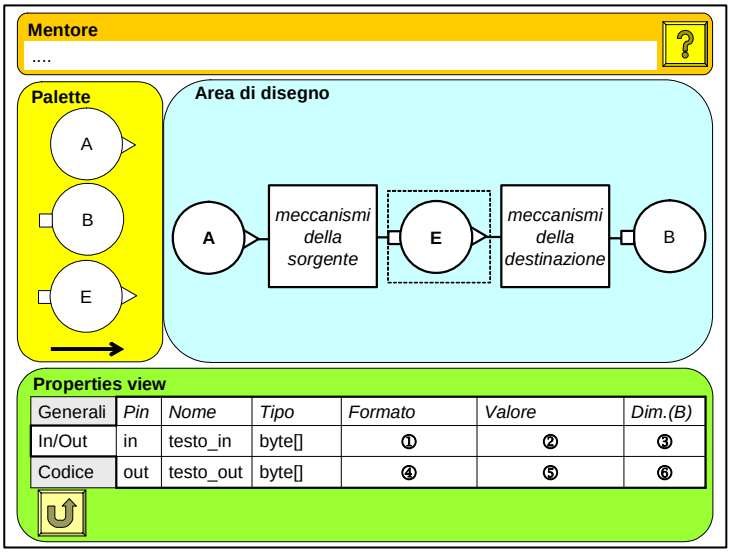
A tal fine l'utente deve scegliere il ciclo *for* nella combo-box del primo campo in alto della scheda In/Out e deve poi digitare 400 nel campo del n° di iterazioni.

L'utente si deve anche occupare della conversione di tipo sempre disponibile sulle uscite di una BOX. Il tipo interno è definito da S-vLab, prendendo atto del componente interno che genera il dato d'uscita; il tipo esterno è definito dall'utente, tenendo conto delle elaborazioni che intende fare al di fuori. Nel caso in esame non è prevista la conversione.

7.4 Simulazione

7.4.1 – La relazione ingresso/uscita

La fase di simulazione intende dare allo studente la possibilità di valutare criticamente la **relazione ingresso/uscita** di un sistema sicuro e di ciascuno dei suoi componenti.



Per quanto riguarda il comportamento dell'intero sistema, un ruolo importante hanno la scelta della configurazione del componente A, tipicamente impiegato per immettere l'informazione da proteggere, la presa d'atto, con un componente B, dell'informazione in uscita dal sistema e la predisposizione su una qualche connessione interna di un componente E, che consente di simulare l'esecuzione di un attacco attivo da parte di un intruso.

In figura è riportata la scheda In/Out di Eva nell'ipotesi che testo_in e testo_out siano array di byte. Nei campi 1 e 4 l'utente deve dire a S-vLab come visualizzare i messaggi d'ingresso e d'uscita. I campi 2, 3, 6 sono riempiti dal sistema. Nel campo 5 l'utente deve digitare il testo_out; per facilitargli il compito, il campo contiene inizialmente il testo_in.

Le proprietà di PRNG (eseguito)

Generali	Provider		**BC**			
In/Out	Algoritmo		**SHA1PRNG**			
Codice	Tipo di seed		**generateSeed**			
	Dim. (byte)		**64**			
	Seed iniziale					
	Pin	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	out	prng_out	byte[]	**0x 58 F1 B9 D9 AE..**	**64**	**4.909842**

Generali	Classe	**SecureRandom**
In/Out	**// Componente PRNG** **SecureRandom PRNG =** **SecureRandom.getInstance("SHA1PRNG","BC");** **byte[] out = PRNG.generateSeed(64);**	
Codice		

Per quanto riguarda il comportamento di ciascun componente, è essenziale prendere atto del contenuto della sua scheda di In/Out dopo il comando di esecuzione.

In questo momento, infatti, S-vLab è in grado di presentare **dimensione e valore** di tutti i **dati d'ingresso,** di tutti i **parametri** e di tutti i **dati d'uscita**; può inoltre indicare il **tempo** che ha impiegato per calcolarli.

In figura è mostrato il caso semplice del PRNG (non ci sono ingressi e c'è una sola uscita). Nel prendere atto dei dati sperimentali occorre sempre un atteggiamento critico: in questo caso ad esempio, eseguendo due prove, si può verificare che la prima esecuzione di un PRNG è più lunga della seconda, dovendo all'inizio JDK costruirsi un *random pool* da cui attingere il seme del generatore.

7.4.2 – Modalità operative

La fase di simulazione inizia mettendo in esecuzione, uno dopo l'altro e con un **doppio click**, i componenti privi d'ingresso e preventivamente configurati. Lo stato di "**eseguito**" che viene così raggiunto dal componente è evidenziato dal passaggio da colore rosso a colore verde del suo contorno e dal colore rosso attribuito alle connessioni che trasportano i suoi dati di output ad altri componenti.

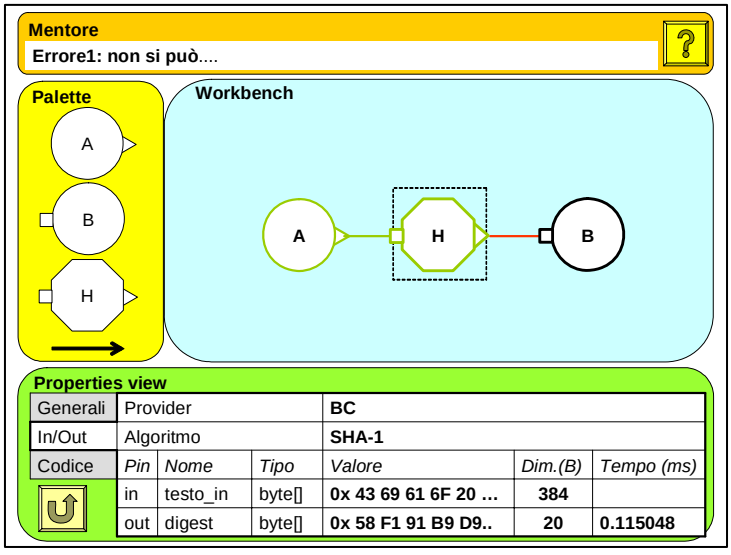
A questo punto altri blocchi configurati diventano "**eseguibili**": colore rosso del simbolo, colore rosso (o verde, come vedremo tra poco) delle loro connessioni d'ingresso, colore nero (o verde, come vedremo) delle loro connessioni d'uscita. Un **doppio click** su uno qualsiasi di questi blocchi lo mette in esecuzione. Al termine lo stato diventa "**eseguito**", il simbolo grafico diventa verde, le connessioni d'ingresso diventano verdi (i dati di input sono stati elaborati ed i componenti che li hanno inviati devono prenderne atto entrando nello stato "**considerato**"), le connessioni d'uscita diventano rosse (dato ancora da elaborare). Nuovi componenti diventano così eseguibili ed ulteriori doppi click fanno progredire passo passo il flusso dei dati all'interno del modello.

Quando tutte le connessioni e tutti i blocchi di uno schema hanno colore verde è possibile procedere ad una nuova simulazione. A tal fine l'utente, a seconda dell'indagine che sta conducendo, deve introdurre nel modello o **nuovi dati** o **nuovi parametr**i: in entrambi i casi deve accedere con un click al componente desiderato e modificarne le proprietà.

Al termine il componente diventa "**riconfigurato**", il colore del suo contorno è rosso e quello delle sue connessioni d'uscita è nero. Una volta che si è operata la riconfigurazione di tutti i componenti che la richiedono, è possibile rientrare in simulazione seguendo le modalità descritte in precedenza.

7.4.3 – Casi di studio

ESEMPIO N.1 (simulazione) **-** Riprendiamo in considerazione il banco di prova dell'algoritmo di hash, supponendo che il componente A sia già stato eseguito. A seguito di un doppio click su H ed una volta terminata la simulazione, il suo contorno passa dal colore rosso al colore verde e la connessione con B diventa rossa per segnalare che occorrerà completare l'esperimento mettendolo in esecuzione.



Si noti che a questo punto sulla scheda In/Out della vista delle proprietà di H appaiono nella riga *out* sia il valore dell'impronta del dato fornitogli da A, sia il tempo impiegato dall'algoritmo SHA-1 per calcolarla.

Facciamo l'ipotesi che il banco di prova sia stato predisposto per confrontare le prestazioni di diversi algoritmi di hash.

Dopo la prova illustrata in precedenza l'utente dovrà prendere atto del tempo di esecuzione (ad es. annotandolo su un foglio), riconfigurare H con il secondo algoritmo, porlo in esecuzione, prendere atto del secondo tempo di esecuzione, riconfigurare H, ecc. ecc.

In figura è mostrato il **report** che uno studente può costruirsi al termine delle prove.

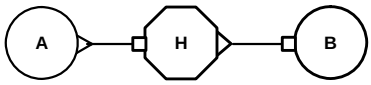
Nella parte alta ha chiesto ed ottenuto da S-vLab lo schema a blocchi che ha costruito.

Sotto, ha giustamente deciso di mostrare il valore e la dimensione della stringa di cui ha calcolato le varie impronte.

Sotto ancora, consultando di volta in volta la scheda di In/Out di H, ha completato il report con un tabulato contenente il provider, l'algoritmo, l'impronta, la dimensione ed il tempo di esecuzione di ciascuno degli esperimenti che ha fatto.

La costruzione del report, con minor fatica, può essere rinviata alla fase di analisi.

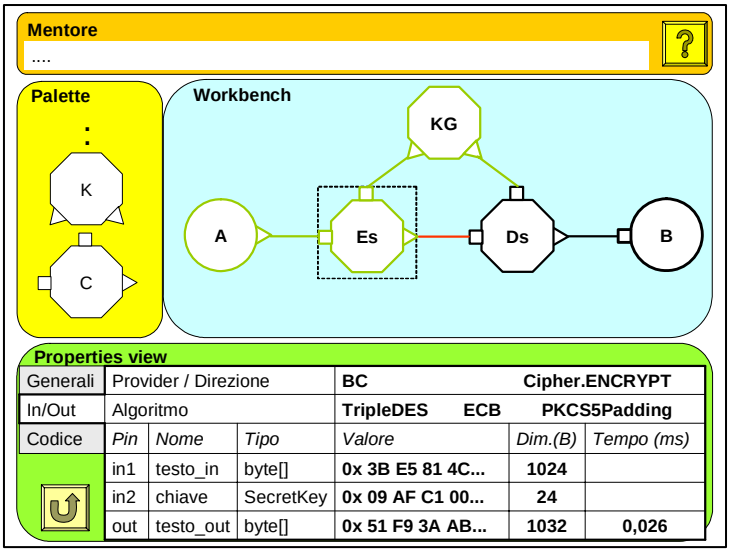
L'analisi dei dati del banco di prova



Input: **0x 43 69 61 6F 20 63 69 61 6F 20 72 61 67 61 7A 7A 69 2C 20 76 65 64 69 ...**
Dimensione (byte): **384**

Provider	Algoritmo	Impronta	Dim. (bit)	Tempo (ms)
BC	SHA-1	0x 58 F1 91 B9 D9 AE B3 AB 95 ..	160	0.008960
SUN	SHA-1	0x 58 F1 91 B9 D9 AE B3 AB 95 ..	160	0.011289
BC	SHA-256	0x 7D E2 4C F8 86 73 99 C4 CA ..	256	0.012290
BC	SHA-512	0x D4 19 31 81 48 8B A1 BE 29 ...	512	0.016876
BC	Tiger	0x BA FB F8 B4 41 6E 49 FD AF ..	192	0.027099
BC	Whirlpool	0x 77 9E 5E 95 FC 90 D1 1B 18 ..	512	0.044419

ESEMPIO N.2 (simulazione)- Supponiamo di aver configurati per il Triple DES i componenti K e C del banco di prova di un cifrario, che K sia già stato eseguito e che nello stesso stato si trovi anche A.



Generali	Provider / Direzione		BC		Cipher.ENCRYPT	
In/Out	Algoritmo		TripleDES	ECB	PKCS5Padding	
Codice	*Pin*	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	in1	testo_in	byte[]	**0x 3B E5 81 4C...**	**1024**	
	in2	chiave	SecretKey	**0x 09 AF C1 00...**	**24**	
	out	testo_out	byte[]	**0x 51 F9 3A AB...**	**1032**	**0,026**

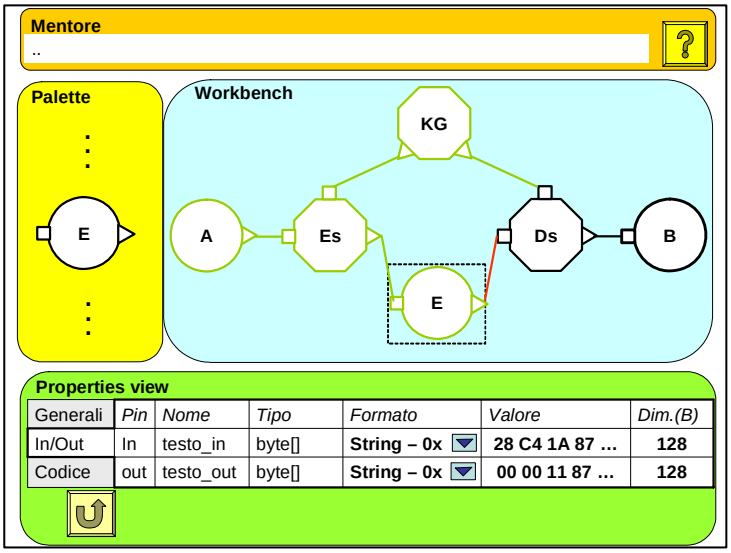
Una volta dato il doppio click su Es, se tutto è stato correttamente impostato, appare il colore verde sul suo contorno ed il colore rosso sulla connessione con Ds; si ricordi che ciò segnala l'avvenuta produzione del testo cifrato e la possibilità di eseguire il componente a valle.

Contemporaneamente sulla scheda In/Out di Es (v. figura a lato) vengono mostrati i dati di **input**, il dato di **output** ed il **tempo di esecuzione**.

Se a questo punto lo studente vuole provare algoritmi di cifratura diversi, deve riconfigurare K e C e ritornare in simulazione.

Il procedimento per generare un report su questi esperimenti è ancora quello che è stato descritto nell'esempio precedente.

Se vuole invece verificare gli effetti di un **attacco attivo** sul canale di comunicazione tra A e B, deve ritornare in modellazione ed inserire un componente **E** tra **Es** e **Ds**.



Generali	*Pin*	*Nome*	*Tipo*	*Formato*	*Valore*	*Dim.(B)*
In/Out	In	testo_in	byte[]	**String – 0x**	**28 C4 1A 87 ...**	**128**
Codice	out	testo_out	byte[]	**String – 0x**	**00 00 11 87 ...**	**128**

Supponiamo che Es sia stato appena eseguito e che venga selezionato Eva. Si noti nella properties view che l'utente ha voluto violare l'integrità del testo cifrato, modificandone solo i primi tre byte.

Per prendere atto degli effetti dell'attacco, l'utente dovrà ora eseguire prima E, poi Ds ed infine B.

Prove significative si potranno ottenere con successive riconfigurazioni delle funzioni di cifratura e di decifrazione: cambiando soltanto la modalità d'uso del cifrario prescelto, lo studente è, infatti, in grado di verificare su B il risultato dell'attacco a TDES nei casi di elaborazione a blocchi (modalità ECB, CBC) e di elaborazione a flusso (modalità CFB, OFB, CTR).

ESEMPIO N. 3 (simulazione) – La BOX impiegata per generare automaticamente 400 impronte di dati casuali è dotata di una sua properties view, come abbiamo già visto, se pure solo in parte, quando ne abbiamo definito la configurazione (v. pag. 155).

Le proprietà di BOX (eseguito)

Generali	...				
In/Out		*N.°*	*Valore:*	*Dim (B):*	*Tempo:*
Codice	out1	**0**	**0x 58 F1 91 B9 D9 AE B3 AB 95 ...**	**20**	**4.917**
		1	**0x 49 D9 77 EA AC 6D B0 E7 08 ...**	**20**	**2.138**
		2	**0x CF C3 3F 3A 87 FC 54 7D C6 ...**	**20**	**2.374**
		3	**0x 4B 30 99 73 5C D7 09 F9 28 ...**	**20**	**2.012**
		...			
		398	**0x 8D 95 EE F4 D3 C0 D5 BE 25 ...**	**20**	**1.992**
		399	**0x 34 22 6D CF 9C DA E7 C0 5B ...**	**20**	**2.100**

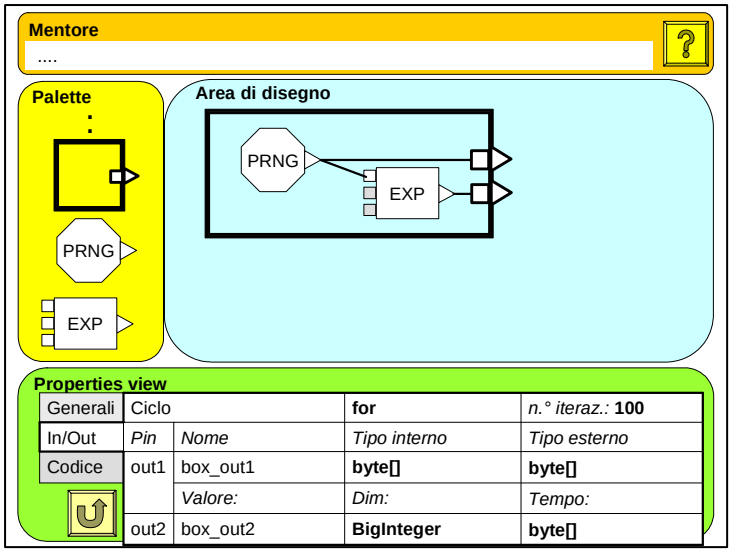
La scheda di In/Out di questo componente è, infatti, decisamente più "lunga" di quella di tutti gli altri, dovendo evidenziare i risultati ed i tempi di esecuzione di ogni iterazione.

Nella figura a lato si suppone di aver fatto doppio click sul contorno della BOX e ci si limita a mettere in evidenza quella parte della scheda di In/Out che visualizza i 400 risultati della simulazione (... per vederli c'è però voluta un po' di pazienza!).

Si noti che S-vLab ha provveduto a numerarli con il valore corrente dell'indice di iterazione, per facilitare eventuali successive analisi sul data base degli eventi.

ESEMPIO N.4 (simulazione) – Al termine della simulazione di tutti i componenti dello scambio anonimo D-H è possibile verificare che i due dati in uscita dalla BOX sono identici.

Uno studente curioso potrebbe però anche chiedersi cosa succede se la base di EXP e di EXP2 non è un generatore del campo di Galois del numero primo usato come modulo. S-vLab gli fornisce la risposta con poca fatica: è, infatti, sufficiente modificare a caso il dato presente su *in2* nelle schede di In/Out di EXP, EXP2 per prendere atto che l'eguaglianza non è più verificata.



Nella figura a lato è delineato un approccio diverso alla costruzione di un nuovo componente complesso: prima di costruirlo interamente, si simulano, separatamente, le sue parti componenti.

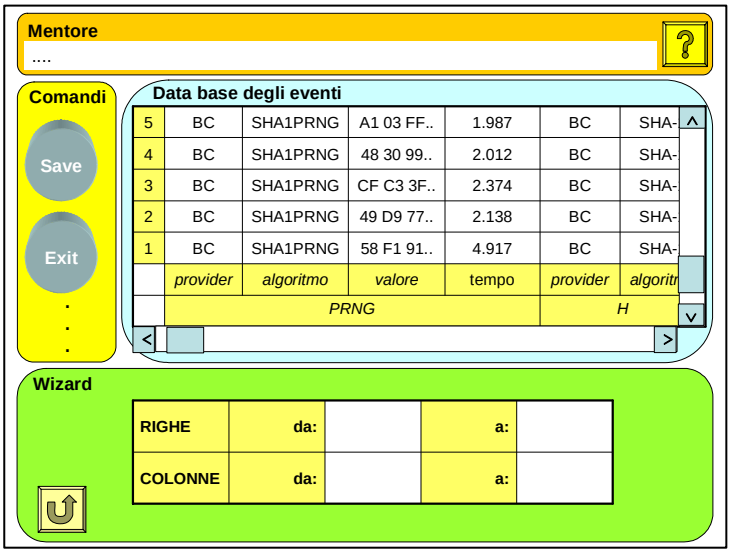
Questa filosofia *bottom-up* applicata al caso dello scambio DH comporta di inserire in una BOX soltanto la parte che genera una coppia chiave privata-chiave pubblica e di replicarla con un "copia & incolla" solo dopo aver analizzato bene la relazione di causa/effetto dell'esponenziazione modulare.

Molto istruttivo è, ad esempio, iterare un centinaio di volte il calcolo delle chiavi per poter poi individuare (a vista o, molto meglio, con i tool disponibili nella fase di analisi dei dati) la dipendenza del tempo di esecuzione dal valore e dalla dimensione dell'esponente.

7.5 Analisi dei dati

7.5.1 – Selezione ed elaborazione dei risultati della simulazione

L'avvio della fase di analisi dei dati sperimentali richiede una "prospettiva" diversa da quella utilizzata nelle tre fasi precedenti ed avviene quindi solo quando lo studente, premendo un apposito pulsante, dichiara conclusa la fase di simulazione.



A questo punto S-vLab gli rende visibili il **data base degli eventi**, una serie di **pulsanti di comando** ed un **wizard** che lo aiuta a selezionare le righe e le colonne contenenti i dati in quel momento di suo interesse.

Se l'utente vuole rimandare la fase di analisi, deve preoccuparsi soltanto di salvare il data base in un suo file prima di uscire; successivamente potrà, infatti, ripristinare la situazione di partenza selezionando la fase di analisi sul menù principale di S-vLab.

Se vuole invece eseguire l'analisi, deve formulare una *query* con l'aiuto del wizard.

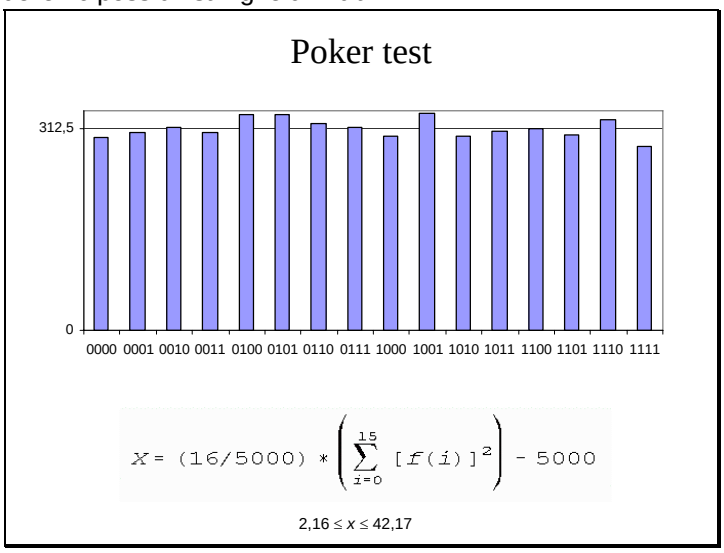
I dati selezionati appariranno nella vista posta al centro dello schermo; nella vista sottostante gli verrà proposto o l'uso di una delle modalità di elaborazione già disponibili o, in alternativa, l'accesso ad un ambiente di sviluppo (il plug-in BIRT della Comunità di Eclipse) in cui potrà programmare ed eseguire i calcoli che ritiene più appropriati.

In entrambi i casi i risultati verranno tabulati e/o graficati. Lo studente potrà o inserirli in un report, o salvarli in un file per esaminarli in seguito; è naturalmente possibile salvare anche le funzioni che ha creato per analizzarli.

7.5.2 – Caso di studio

Per dare un'idea più concreta su quello che si può fare, assumiamo come caso di studio l'esempio N.3, ricordando che il banco di prova aveva l'obiettivo di verificare se l'algoritmo di compressione SHA-1 si comporta o meno come un *random oracle*.

Supponiamo di aver selezionato sul data base i valori di 125 impronte consecutive per avere a disposizione un totale di 20.000 bit: l'uso di un campione di questa lunghezza è, infatti, uno standard fissato dal NIST per l'esecuzione di test statistici. Supponiamo inoltre di aver richiesto a S-vLab il **grafico delle occorrenze** delle 16 possibili stringhe di 4 bit.



In figura è mostrato il risultato: si noti che il n° delle occorrenze di ciascuna stringa è sempre molto prossimo a 5000/16=312,5.

Assumendo la stringa di quattro bit come variabile aleatoria, si può dunque affermare che la sua distribuzione di probabilità è uniforme.

Per essere più sicuri di quest'ultima affermazione, si può chiedere a S-vLab di eseguire il **poker test** del NIST(v. pag.28), cioè di fare il calcolo indicato in figura e di verificare se il risultato ricade nell'intervallo 2,16-42,17.

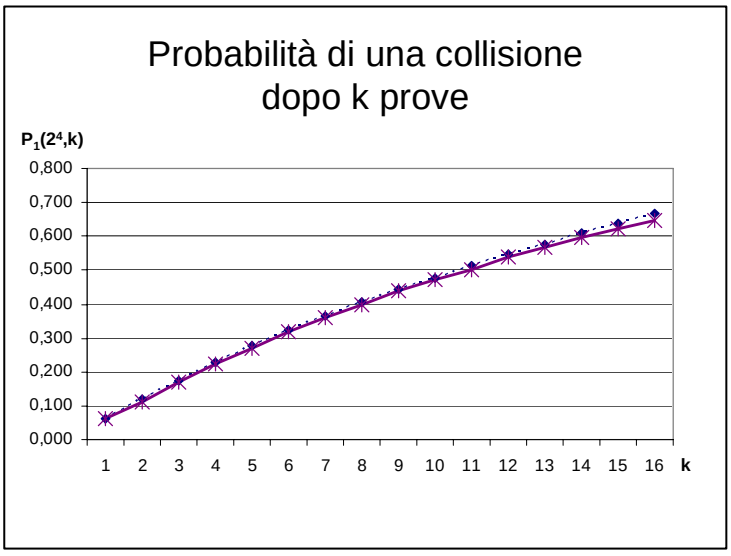
Nel caso in esame si ottiene 10,4, valore che determina quindi il superamento del test. S-vLab consente anche di eseguire i test del **monobit** e del **run**, che in questo caso vengono tutti superati. SHA1 è dunque un ottimo oracolo casuale.

La stringa di 20.000 bit può essere impiegata anche per verificare sperimentalmente le formule di pag. 33, che forniscono rispettivamente, per una sorgente casuale, la probabilità di individuare, con **k esperimenti**, o una stringa identica ad una prefissata e scelta a caso, o due stringhe identiche.

Facciamo ancora riferimento a stringhe di quattro bit. I calcoli da richiedere a BIRT non sono troppo complicati, ma non vale qui la pena di esaminarli in dettaglio[66]: accontentiamoci dunque di una descrizione a parole.

Nel primo caso occorre occorre preliminarmente attribuire un indice alle 5.000 stringhe di 4 bit, poi calcolare la differenza degli indici per tutte le coppie di stringhe identiche, poi valutare quante coppie hanno "distanza" 1, quante "distanza" 2, ecc..

Dividendo i numeri così ottenuti per 5.000 si ottiene una stima della probabilità di ogni valore di "distanza". La somma delle stime relative alle distanze 1, 2, 3..,k fornisce infine una stima della probabilità di trovare una collisione facendo k esperimenti.



In figura è mostrato il grafico di tali risultati: con undici prove la probabilità di trovare la collisione è circa 0,5; occorrono più di sessanta prove per essere quasi certi di ritrovare il valore prefissato.

Per valutare la correttezza di questa analisi è utile prendere come termine di paragone la funzione $P_1(2^4,k)$. In questo caso le stringhe sono però molto corte e non si può quindi usare l'approssimazione indicata a pag.33; il grafico di figura si riferisce quindi ai primi tre termini dello sviluppo in serie.

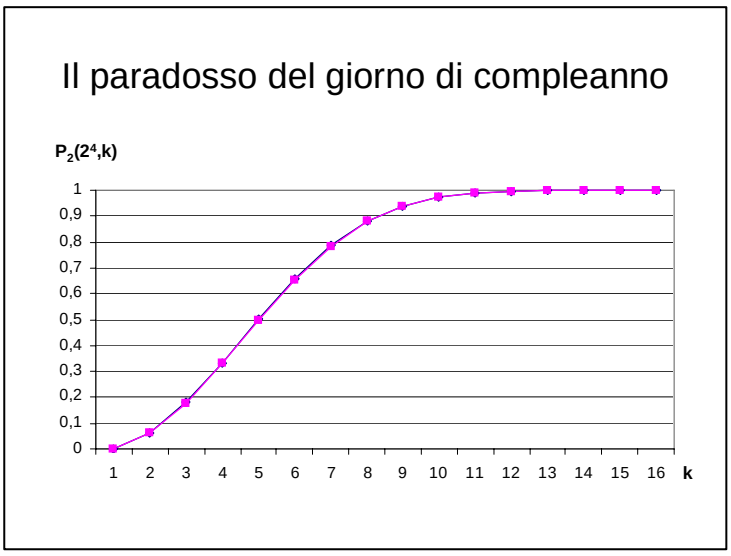
La quasi perfetta sovrapposizione delle due curve dimostra ancora una volta che SHA1 è un ottimo oracolo casuale.

Risultati del tutto simili si otterrebbero anche con gli altri algoritmi crittografici di compressione disponibili in S-vLab.

Nel secondo caso occorre calcolare, a partire da ciascuna delle 5.000 stringhe di 4 bit, quante stringhe successive occorre esaminare per trovarne due in collisione.

Si procede poi come indicato in precedenza, valutando quante volte si è trovato 2, 3, ecc., stimando poi la probabilità di occorrenza di ciascun valore e sommando infine le stime relative alle distanze 1, 2, .., k..

[66] I sorgenti e gli eseguibili di tutte le funzioni, realizzati da Stefano Maggiore, sono disponibili nel sito del corso.



In figura sono mostrate la curva che si ottiene dai dati sperimentali e quella che discende dalla formula:

$$P_2(2^4,k)= 1-(2^4! / (2^4-k)!2^{4k})$$

indicata a pag.33.

In realtà si vede una curva sola, essendo pressoché perfetta la sovrapposizione dei dati sperimentali e dei dati teorici.

Con 5 tentativi si ha probabilità 0,5 di trovare una coppia in collisione, con 10 tentativi si è molto vicini alla certezza di trovarla!

Abbiamo così verificato sperimentalmente il cosiddetto paradosso del giorno di compleanno: la formula approssimata di pag.33

$$k = (2\ln(2))^{1/2} (2^n)^{1/2}$$

indica, infatti, che occorre fare 4,7 tentativi per avere la probabilità del 50% di trovare una coppia in collisione.

7.6 Codifica

La funzionalità di supporto alla codifica in Java di un sistema sicuro è realizzata da S-vLab attraverso la generazione di:

1. **estratti di codice didattico**, ciascuno corrispondente ad un componente presente nel modello e consultabile attraverso la scheda Codice contenuta nella property view del componente stesso;
2. un **listato complessivo**, relativo all'intero sistema studiato, riportato all'interno di un file che può essere visualizzato e modificato dall'editor dell'IDE Eclipse accessibile dal laboratorio.

Le due tipologie di codice didattico differiscono, oltre che per contenuto informativo e completezza, anche per la fase di lavoro in cui risultano accessibili agli utenti del laboratorio.

Gli statement relativi a ciascun componente di uno schema, infatti, vengono visualizzati durante le attività di modellazione, configurazione e simulazione.

Il listato complessivo viene invece visualizzato solo quando l'utente ha terminato la fase di simulazione ed ha deciso di passare alla prospettiva di **codifica** premendo l'apposito pulsante. Il file contenente il listato ha estensione ".java" ed è automaticamente incluso da S-vLab all'interno di un **progetto** dell'ambiente di sviluppo Java della piattaforma Eclipse.

7.6.1 Gli estratti di codice didattico

Lo scopo della visualizzazione degli statement relativi ad un componente presente nell'area di disegno è quello di fornire agli utenti un'indicazione rapida, efficace e corretta su come possa essere codificata la funzione svolta dal componente.

Poiché gli estratti di codice presentati sono incentrati sul singolo blocco e focalizzati su una specifica funzione, l'**indipendenza** dal resto dello schema diventa una caratteristica imprescindibile: per ottenerla, è importante che gli statement dipendano soltanto dalle variabili interne al blocco di interesse. In questa fase, quindi, l'acquisizione di eventuali dati di **input** prodotti da altri elementi del sistema studiato verrà rappresentata mediante l'assegnamento alle variabili interne del valore di alcune variabili esterne associate all'astrazione del *pin* del componente.

ESEMPIO – Nel paragrafo 7.3.5, a pag. 154, è stata esaminata la configurazione del componente EXP nel caso in cui i primi due input, rispettivamente l'esponente e la base, vengano prodotti da due blocchi a monte. Gli statement Java risultanti includono infatti due assegnamenti tra le opportune variabili interne (caratterizzate da un nome esplicativo, riportato nella colonna "nome" della scheda In/Out delle proprietà) ed i pin corrispondenti.

L'estratto di codice corrispondente a ciascun componente viene caricato all'interno della property view, in una versione incompleta, già all'atto del trascinamento del blocco nell'area di disegno. Come si è visto nel paragrafo 7.3, la **completezza** della sequenza di statement dipende però dalla successiva fase di **configurazione** del componente: i parametri che l'utente imposta attraverso la scheda di In/Out, infatti, si riflettono anche sul listato parziale come argomenti di metodi o funzioni o, in alcuni casi, come discriminanti fra più (sequenze di) statement alternativi.

ESEMPI – Una volta impostati il provider e l'algoritmo del componente **H** (v. pag.153), le opzioni scelte diventano gli argomenti del metodo *MessageDigest.getInstance(…)*.

Provider, algoritmo, direzione, modalità di cifratura e padding scelto completano lo statement associato al componente **C** (v. pagg. 153-154) nei metodi *Cipher.getInstance()* e *Cipher.init()*;

Il componente **EXP** (v. pag. 154) modifica l'estratto di codice ad esso corrispondente a seconda che i dati di input vengano impostati attraverso la finestra delle proprietà o siano invece ricevuti da connessioni.

Il componente **A** (v. pag. 152), a seconda che riceva l'input in forma di stringa digitata dall'utente nell'apposito campo della property view, o che invece lo importi da un file di cui l'utente ha specificato il percorso, presenta due listati significativamente diversi.

Il componente **B** (v. pag. 152) presenta sequenze di statement differenziate per la semplice visualizzazione del risultato all'interno della scheda In/Out e per il salvataggio su file.

La funzione svolta dal componente **K** viene realizzata utilizzando la classe *KeyGenerator* o *KeyPairGenerator* a seconda che l'algoritmo scelto sia simmetrico o asimmetrico.

Il componente **PRNG** può essere utilizzato per generare un seed oppure dei byte casuali[67]: nel primo caso si ricorrerà al metodo *generateSeed()* della classe Java *SecureRandom*, il secondo caso comporterà invece l'invocazione del metodo *nextBytes()*. I due metodi prevedono modalità diverse per la specifica del numero di byte (di seed o casuali) desiderati: *generateSeed()*, infatti, accetta come argomento un intero; *nextBytes()*, invece, riceve un array della dimensione voluta, all'interno del quale collocherà i byte prodotti. E' infine possibile inizializzare il PRNG con un seed dato: la riga di codice corrispondente a questa funzione comparirà o meno nel listato parziale a seconda che l'utente imposti o meno un valore per il parametro. Le due figure seguenti mostrano le schede In/Out e Codice del componente PRNG per le due possibili modalità di configurazione, differenti per tipo di seed; inoltre, mentre nel primo caso l'utente ha scelto di non inizializzare il PRNG, nel secondo caso ha impostato un valore per il parametro "Seed iniziale": di conseguenza, il listato parziale conterrà uno statement in più dedicato a questa funzione.

Le proprietà di PRNG (generateSeed)

Generali	Provider		**BC**			
In/Out	Algoritmo		**SHA1PRNG**			
Codice	Tipo di seed		**generateSeed**			
	Dim. (byte)		**64**			
	Seed iniziale					
	Pin	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	out	prng_out	byte[]	**0x 58 F1 B9 D9 AE..**	**64**	**4.909842**

Generali	Classe	**SecureRandom**
In/Out	**// Componente PRNG**	
Codice	**SecureRandom PRNG =**	
	SecureRandom.getInstance("SHA1PRNG","BC");	
	byte[] out = PRNG.generateSeed(64);	

Le proprietà di PRNG (nextBytes)

Generali	Provider		**BC**			
In/Out	Algoritmo		**SHA1PRNG**			
Codice	Tipo di seed		**nextBytes**			
	Dim. (byte)		**64**			
	Seed iniziale		**0x 41 85 C6 0E 32 B1 9F 05 A4 21 79 4B ...**			
	Pin	*Nome*	*Tipo*	*Valore*	*Dim.(B)*	*Tempo (ms)*
	out	prng_out	byte[]	**0x B7 23 9C 11 0E ..**	**64**	**4.909842**

Generali	Classe	**SecureRandom**
In/Out	**// Componente PRNG**	
Codice	**SecureRandom PRNG =**	
	SecureRandom.getInstance("SHA1PRNG","BC");	
	PRNG.setSeed(new byte[] { 0x41, 0x85, 0xC6, 0x0E, 0x32, 0xB1, 0x9F..	
	byte[] out = new byte[64];	
	PRNG.nextBytes(out);	

7.6.2 Il listato complessivo

Il listato complessivo è associato al sistema sicuro che lo studente ha in precedenza modellato, configurato e simulato e viene composto automaticamente da S-vLab per concatenazione degli estratti di codice dei componenti dello schema.

Le varie sequenze di statement sono inframezzate da linee di codice aggiuntive che rappresentano le connessioni per il passaggio dei dati fra le varie sezioni. L'ordinamento sequenziale dei diversi moduli viene stabilito grazie ad un'analisi statica, effettuata sull'intero schema, dei vincoli di precedenza imposti dalle connessioni, e quindi dall'acquisizione degli input necessari.

L'obiettivo formativo del listato è quello di fornire allo studente uno spunto per la realizzazione di un prototipo software. Il listato, infatti, è costituito da un codice esemplificativo, corretto ma incompleto e non ottimizzato: in generale non risulta quindi né compilabile, né eseguibile prima che siano apportate le opportune modifiche nell'ambiente di sviluppo integrato con il laboratorio.

In particolare, il codice didattico:

1. include una sezione di ***import***, non necessariamente esaustiva: gli studenti devono quindi aggiungere le chiamate alle librerie mancanti;

[67] La differenza tra la generazione di un seed (necessario, ad esempio, per inizializzare un altro PRNG) e di byte casuali sta nell'algoritmo utilizzato: nel primo caso, la classe SecureRandom di Java ricorre allo stesso algoritmo che usa per inizializzare se stessa; nel secondo caso ricorre invece all'algoritmo scelto dall'utente (in genere SHA1PRNG).

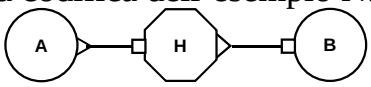
2. non prevede la gestione delle **eccezioni**: sono gli utenti a dover inserire gli opportuni blocchi *try / catch*, in seguito alla consultazione della documentazione di riferimento (javadocs) o all'individuazione, in fase di simulazione, dei componenti che in alcune condizioni possono non terminare con successo la loro esecuzione;
3. introduce **ridondanza** in corrispondenza del passaggio dei dati da un modulo all'altro: la voluta indipendenza degli estratti di codice associati ai componenti impone, infatti, di realizzare ogni connessione con due successivi assegnamenti (da variabile esterna a pin e da pin a variabilie interna), cosa che dovrebbero essere eliminata se si vuole ottenere un codice più compatto e più efficiente.

ESEMPIO N.1 (codifica) – Il listato complessivo del codice associato al banco di prova di un algoritmo di hash è illustrato in figura. Scorrendolo si evince che il componente **A** è stato configurato in modo da generare un array di byte a partire dalla stringa "ciao ciao" codificata in Unicode, che **H** ne calcola l'impronta con l'algoritmo SHA-1 del provider BouncyCastle e che **B** visualizza in formato esadecimale la stringa ottenuta a partire da un array di byte ricevuto in input.

Si noti (vedi la freccia chiara in figura) che in testa al listato S-vLab ha già previsto l'importazione della classe *MessageDigest* del package **java.security,** la sola chiamata in causa in questo caso.

Se i componenti A o B fossero stati configurati in modo da gestire dei file, sarebbe stato compito degli utenti aggiungere (prima della compilazione del listato) le chiamate alle librerie aggiuntive necessarie.

La codifica dell'esempio N.1




```
import java.security.MessageDigest;

// Componente A
String testo_in_A1 = i1_A1;
byte[] testo_out_A1 = testo_in_A1.toByteArray();
byte[] o1_A1 = testo_out_A1;

byte[] i1_H1 = o1_A1;

// Componente H
byte[] testo_in_H1 = i1_H1;
MessageDigest mDigest = MessageDigest.getInstance("SHA-1","BC");
byte[] digest_H1 = mDigest.digest(testo_in_H1);
byte[] o1_H1 = digest_H1;

byte[] i1_B1 = o1_H1;

// Componente B
String testo_in_B1 = i1_B1;
byte[] testo_out_B1 = byteToHex(testo_in_B1);
byte[] o1_B1 = testo_out_B1;
```

Si noti anche (vedi le frecce scure in figura) che il codice didattico **non è ottimizzato**: l'assemblaggio diretto dei moduli software contenuti nelle schede Codice di A, H e B ha, infatti, prodotto due assegnamenti inutili nel codice complessivo. Infine, poiché l'invocazione del metodo *getInstance()* della classe MessageDigest può, in alcuni casi, lanciare delle eccezioni, è opportuno che gli studenti completino il listato inserendo gli opportuni blocchi *try / catch*.

Il listato didattico generato dal laboratorio virtuale, essendo pensato per introdurre gli utenti alle classi e ai metodi di riferimento per meccanismi e servizi di sicurezza, trascura completamente gli aspetti sistemistici e di deployment delle applicazioni in cui tali servizi andranno utilizzati. Per questo motivo, un utente che desideri sviluppare un prototipo software a partire dal codice esemplificativo, dovrà dedicare un'attenzione particolare alla sua riorganizzazione in moduli separati e interagenti e curare gli aspetti di comunicazione, sincronizzazione e distribuzione.

ESEMPIO N.2 (codifica) – La parte più significativa del codice del banco di prova di un cifrario è riportata in figura. Al blocco K è stata richiesta la generazione di una chiave di 112 bit per il Triple DES, che viene di conseguenza chiamato in causa anche nei successivi metodi di cifratura e di decifrazione.

Si noti che questa volta il listato, anziché includere le singole classi utilizzate, importa i due interi package di riferimento: **java.security** e **javax.crypto**.

Mancano, ancora, eventuali chiamate a librerie per la gestione dell'input/output.

Tale listato costituisce solo un punto di partenza per la fase di codifica. Nel caso specifico è necessaria una decomposizione del programma in (almeno) due moduli eseguibili su macchine diverse: è infatti indispensabile separare la fase di cifratura da quella di decifrazione, collegarle con un opportuno canale di comunicazione, prevedere un metodo sicuro per la condivisione della chiave e gestire dinamicamente l'input/output.

La codifica dell'esempio N.2

```
import java.security.*;
import javax.crypto.*;

// Componente A
...
// Componente K
KeyGenerator KG1 = KeyGenerator.getInstance("TripleDES");
KG1.init(112); SecretKey chiave_segr_KG1 = KG1.generateKey();
SecretKey o1_KG1= chiave_segr_KG1; SecretKey o2_KG1= chiave_segr_KG1;

SecretKey i2_Es1 = o1_KG1; SecretKey i2_Ds2 = o2_KG1;

// Componente Es
byte[] testo_in_Es1 = i1_Es1; SecretKey chiave_Es1 = i2_Es1;
Cipher Es1 = Cipher.getInstance("TripleDES/ECB/PKCS5Padding");
Es1.init(Cipher.ENCRYPT_MODE, chiave_Es1);
byte[] testo_out_Es1 = Es1.doFinal(testo_in_Es1);
// Componente CsD
byte[] testo_in_Ds2 = i1_Ds2; SecretKey chiave_Ds2 = i2_Ds2;
Cipher Ds2 = Cipher.getInstance("TripleDES/ECB/PKCS5Padding");
Ds2.init(Cipher.DECRYPT_MODE, chiave_Ds2);
byte[] testo_out_Ds2 = Ds2.doFinal(testo_in_Ds2);
// Componente B
...
```

7.6.3 La prospettiva di sviluppo software

Alla fase di codifica di un prototipo, all'interno del laboratorio virtuale, è dedicata la prospettiva di sviluppo software di S-vLab, che coincide con l'ambiente di sviluppo offerto dalla piattaforma Eclipse.

All'interno di questa prospettiva, gli utenti possono usufruire di tutti i vantaggi offerti dall'IDE: tra i principali vi sono

- la consultazione delle viste di Console e di Errors,
- l'attivazione della prospettiva di Debug,
- l'installazione di eventuali strumenti aggiuntivi per la pianificazione di test e per il monitoraggio delle performance del codice,
- l'accesso all'help online,
- il completamento automatico del codice,
- la sintassi evidenziata per lo specifico linguaggio di programmazione in uso (di default è Java, ma installando gli opportuni plug-in è possibile personalizzare l'ambiente di sviluppo per altri linguaggi).

Per una completa descrizione dell'ambiente IDE Eclipse si rimanda alla documentazione ufficiale distribuita gratuitamente dalla Comunità Eclipse attraverso il sito www.eclipse.org.

Per poter sfruttare appieno le potenzialità di questa prospettiva, è necessario creare un progetto (Java, nel nostro caso) e lavorare al suo interno, creando dei nuovi file oppure importando il listato esemplificativo prodotto da S-vLab al termine di un esperimento.

A questo punto, gli utenti possono editare il codice, decomporlo e completarlo con le necessarie procedure di comunicazione appoggiandosi ad eventuali risorse aggiuntive supportate dall'ambiente di sviluppo.

ESEMPIO – Tra le varie possibilità a cui gli studenti possono ricorrere per ottimizzare ed espandere il listato esemplificativo così da costruire dei prototipi significativi, sono particolarmente interessanti:

- SignedObject e SealedObject;
- estensione a infrastrutture di comunicazione basate su diverse tecnologie (RMI, scambio di messaggi supportati da middleware JMS, contesto mobile J2ME, …);
- introduzione e implementazione di pattern per la sicurezza;
- gestione della persistenza;
- utilizzo di librerie e package di supporto e confronto tra provider e soluzioni alternative.

.