UNIVERSITÀ
degli STUDI
di CATANIA

Dipartimento di Matematica ed Informatica

Dottorato di Ricerca in Matematica ed Informatica
Settore Scientifico Disciplinare INF/01

SIRENE
un linguaggio di programmazione visuale per la didattica della programmazione

Tesi di Dottorato di:
Dott. **Guido Averna**

Il Tutor: ***Chiar.mo***
Prof. **Domenico Tegolo**

Il Coordinatore: ***Chiar.mo***
Prof. **Giovanni Russo**

Il Co-Tutor: ***Chiar.mo***
Prof. **Biagio Lenzitti**

Ciclo XXX
Anno conseguimento titolo: 2019

A mia madre Irene
ed ai miei piccoli grandi amori
Ambra, Luna, Luce, Perla e Selene

C'è una forza motrice più forte del vapore,
dell'elettricità, dell'energia atomica:
la volontà.
Albert Einstein

Ringraziamenti

La tesi di un Dottorato di ricerca è il frutto e la sintesi di un lungo percorso non semplice da affrontare. Durante questo percorso, mi sono state vicine alcune persone che mi hanno guidato, sorretto o semplicemente incoraggiato nell'andare avanti e nel non desistere.

Desidero ringraziare sinceramente il mio Tutor, il prof. Domenico Tegolo, Professore del Dipartimento di Matematica ed Informatica di Palermo, ed il mio Co-Tutor, il prof. Biagio Lenzitti, Professore del Dipartimento di Matematica ed Informatica di Palermo, per avermi indirizzato, spronato e fatto crescere durante questo cammino. La loro guida ed il loro supporto sono stati essenziali.

Desidero ringraziare Maria per essermi sempre stata accanto, per avermi sempre ben consigliato, supportato e "sopportato" durante questi anni, dimostrandomi il suo amore.

Desidero ringraziare i miei tre cari fraterni amici Nino, Nino† e Pippo, per non avermi mai abbandonato nelle difficoltà che si sono manifestate sia prima sia durante questo tragitto.

Ringrazio i miei cari amici Arabella e Giuseppe i quali mi hanno pazientemente incoraggiato con la loro simpatia ogni giorno durante questi anni.

Ringrazio il Direttore Don Domenico Saraniti, il Preside Prof. Nicola Filippone, il Dipartimento di Matematica, Fisica ed Informatica dell'Istituto Salesiano "Don Bosco - Villa Ranchibile" di Palermo, ed in particolare il prof. Alessio Sofia, docente di Informatica, per i loro insegnamenti e per avermi permesso di condurre la sperimentazione utile all'arricchimento di questo lavoro.

Altresì, desidero ringraziare i professori ed i ricercatori del Dipartimento di Matematica ed Informatica di Palermo, con cui mi sono confrontato, per avermi dato spunti, idee o semplicemente un conforto, durante questa mia crescita personale e

professionale.

Guido Averna

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Dalla metà del ventesimo secolo vi è stato un aumento della diffusione dei personal computer dovuto ad una crescente domanda nel mercato di questo elaboratore grazie alla sua riduzione sia di prezzo e sia di dimensioni. Questo ha permesso, a molte persone, di poter utilizzare i computer per le più svariate finalità e non soltanto per attività di tipo strettamente scientifico o accademico. La facilità di utilizzo, dovuta alle interfacce Uomo-Macchina, e le potenzialità di utilizzo dei personal computer hanno portato le comunità scolastiche ed accademiche, nazionali ed internazionali, a sviluppare corsi di programmazione ad ogni livello. In questi anni, dunque, sono stati sviluppati molti strumenti utili all'implementazione di programmi: molti di questi usano un paradigma text-based mentre solo pochi strumenti usano un paradigma visuale, o iconico. Questo lavoro propone un framework iconico di programmazione utile all'insegnamento ed all'apprendimento dell'arte della programmazione, a livello sia concettuale e sia implementativo, in un contesto collaborativo e distribuito in rete sfruttando le più recenti tecnologie per una condivisione, in tempo reale, dell'ambiente di lavoro e degli algoritmi e programmi sviluppati tra il docente, o "l'esperto", e gli studenti, o "i principianti". Si discuterà delle difficoltà comuni che hanno gli studenti dei corsi di programmazione e lo stato dell'arte attuale degli strumenti iconici di sviluppo utili all'insegnamento e all'apprendimento della programmazione. Dunque, si presenterà un framework iconico, SIRENE, che possiede caratteristiche real-time, è cooperativo, usabile in ogni dispositivo ed in internet ed è progettato per l'insegnamento e l'apprendimento della programmazione. Si discuterà della sua implementazione, argomentando anche delle varie difficoltà implementative riscontrate e le loro soluzioni. Si perverrà, infine, alla conclusione analizzando una sperimentazione ed i suoi risultati e la realizzazione del framework visuale, dei suoi possibili usi pratici, delle sue potenzialità e delle sue possibili estensioni.

Capitolo 1

Linguaggi di programmazione visuale: stato dell'arte

1.1 Corsi di programmazione e linguaggi di programmazione

Esiste un vasto numero di linguaggi di programmazione per computer e in generale per sistemi di elaborazione. Ognuno di essi possiede una serie di proprie utili caratteristiche che li differenziano.
I linguaggi di programmazione possono essere classificati in:

- linguaggi di programmazione testuali, denominati Text Programming Language (TPL).
- linguaggi di programmazione visuale, o iconica, denominati Visual Programming Language (VPL).

Per ogni linguaggio di programmazione sono stati creati vari ambienti di sviluppo che forniscono utili strumenti e risorse per lo sviluppo di algoritmi e programmi.
I corsi di programmazione che le scuole, di ogni grado e livello, e le università hanno introdotto nei loro piani di formazione, devono adeguarsi sia al linguaggio di programmazione adottato sia agli strumenti didattici utilizzabili durante il corso.
In ogni corso di programmazione, l'insegnante deve sviluppare, e potenziare, negli studenti una capacità fondamentale per il corretto apprendimento della programmazione: il pensiero computazionale. Il pensiero computazionale, originariamente usato da Seymour Papert nel suo [48] e successivamente formalizzato da Jeannette Wing nel suo [60], è un insieme di processi mentali che permettono la comprensione di un problema per trovare una o più soluzioni. Vi sono tipicamente quattro fasi nella formalizzazione di un problema:

1. Astrazione.
2. Decomposizione.
3. Riconoscimento di schemi ripetuti.
4. Costruzione di un algoritmo risolutivo di un problema.

La fase di astrazione permette di concentrarsi sull'essenza del problema e delle sue caratteristiche chiavi. La fase di decomposizione scompone il problema in sotto problemi più semplici da affrontare. La fase di riconoscimento di pattern verifica se esistono delle soluzioni, anche similari, già conosciute. Mentre la fase algoritmica fornisce una o più procedure risolutive del problema.
Sebbene sia nato in ambito informatico, la capacità del pensiero computazionale non è confinata esclusivamente a questo settore, ma riguarda anche altri ambiti scientifici e non scientifici: ad esempio la matematica, la fisica ma anche il latino, etc. Uno strumento utile per la scomposizione dei problemi usato da molti insegnanti, non necessariamente di informatica, è la mappa concettuale, di cui se ne discuterà brevemente nel successivo paragrafo. I principali obiettivi dei corsi di programmazione sono:

- sviluppare le capacità di comprensione, di astrazione, di analisi e di risoluzione dei problemi;
- realizzare un programma usando uno specifico linguaggio di programmazione.

Ciò implica la necessità di sviluppare il pensiero computazionale , il problem solving, l'astrazione concettuale di un problema, della sua soluzione implementativa e della sua traduzione in uno specifico linguaggio di programmazione [29]. Gli studenti riscontrano un alto grado di difficoltà nel comprendere, astrarre ed analizzare un problema e nel realizzare una sua soluzione e, quindi, sviluppare il pensiero computazionale. Gli studenti focalizzano principalmente la loro attenzione sulla sintassi del linguaggio di programmazione studiato, tralasciando l'aspetto concettuale e generale di un problema. Così il docente spesso spende la maggior parte del tempo del corso sulla spiegazione della corretta sintassi del linguaggio oggetto di studio a discapito dell'astrazione concettuale di un problema e delle sue soluzioni algoritmiche.
I linguaggi di programmazione di tipo TPL sono largamente usati, soprattutto da quei professionisti che hanno già acquisito le abilità e le tecniche che permettono loro di sviluppare progetti, anche complessi, con una certa facilità e familiarità. Diversamente, i meno esperti, come gli studenti che si approcciano per la prima volta ad un linguaggio di programmazione, considerano i TPLs come linguaggi ostici e criptici da comprendere e con i quali la codifica di un programma risulta troppo

difficile e ardimentosa da implementare. Invece, i Visual Programming Languages includono anche un'interfaccia grafica semplice da comprendere e supportata da simboli grafici che esprimono un significato specifico o una precisa entità, e quindi sono più semplici ed immediati da comprendere a livello cognitivo.

1.2 Mappe Concettuali

Le mappe concettuali [27, 32, 47, 59, 46] non sono strumenti di programmazione ma di apprendimento, tuttavia sono utili strumenti per la didattica. Una mappa concettuale rappresenta una organizzazione della conoscenza di un generico argomento. Questa organizzazione è sintetica e serve a focalizzare l'attenzione sui concetti chiave dell'argomento affrontato. L'organizzazione è rappresentata da una struttura espressa mediante la connessione di oggetti visuali. Tali oggetti sono tipicamente dei riquadri con all'interno un'immagine, un testo significativo di un aspetto, o di un elemento chiave, di un argomento e sono associati ad uno o più specifici argomenti. La connessione tra questi oggetti avviene mediante l'uso di archi, generalmente direzionati, verosimilmente a dei grafi orientati e ai diagrammi. La differenza con quest'ultima tipologia è che nelle mappe concettuali non necessariamente la conoscenza deve partire da un preciso elemento e/o non necessariamente vi è un unico flusso. Le mappe concettuali sono utilissimi strumenti didattici e pedagogici. Per la loro semplicità di utilizzo ed utilità, sono spesso utilizzate in tutti i casi di alunni che necessitano di Bisogni Educativi Speciali (BES) o con Disturbi Specifici di Apprendimento (DSA). Vengono anche utilizzate per l'insegnamento in assenza di alunni con disturbi particolari. Per la loro importanza e grande utilità, sono state introdotte nelle scuole, di ogni ordine e grado, e normate da leggi nazionali (Legge n. 170 dell'8 ottobre 2010, Decreto Ministeriale 12 Luglio 2011).

1.3 Linguaggi visuali di programmazione

I linguaggi di programmazione iconica sono validi ed utili strumenti che possono supportare l'insegnante a focalizzare l'attenzione degli studenti sulle caratteristiche dei problemi e sulle tecniche necessarie per le loro possibili soluzioni. Ciò facilita la comprensione dei problemi e del loro problem solving, creando un'astrazione concettuale del problema e della soluzione che è possibile realizzare in ogni linguaggio di programmazione. Quindi, i linguaggi iconici aiutano a sviluppare il pensiero computazionale.
Nel secolo scorso sono stati sviluppati diversi linguaggi di programmazione visuale,

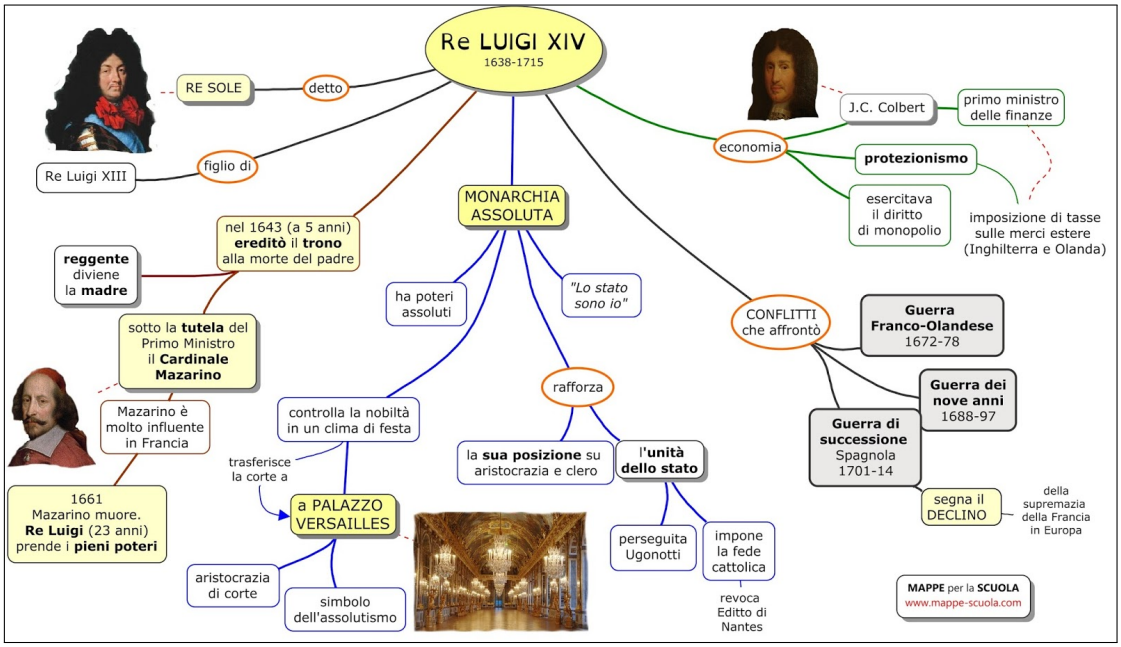


Figura 1.1: Esempio di Mappe Concettuali.

per le più svariate finalità, non necessariamente al solo fine di creare programmi. Infatti, alcuni linguaggi visuali sono stati sviluppati per facilitare la manipolazione dei dati all'interno dei database, ne è un esempio VQL in [39], il quale permette di ottenere informazioni sia sul modello del database sia sui dati all'interno di questi. Per esempio, per eseguire interrogazioni sui database si usano tre semplici primitive grafiche. La concatenazione di questi elementi permette di realizzare le interrogazioni sui database. Altri linguaggi sono stati sviluppati per aiutare i programmatori durante le varie fasi di sviluppo dei linguaggi di programmazione visuale come AgentsSheets in [51]. AgentsSheets è un linguaggio visuale che permette di realizzare linguaggi di programmazione visuali orientati nel dominio, ovvero consente di creare linguaggi iconici che permettono agli utilizzatori finali di usare facilmente il linguaggio in quanto questo linguaggio iconico viene realizzato esplicitamente per gli utilizzatori finali e per uno specifico contesto e dominio di applicazione. Essenzialmente, AgentsSheets è un linguaggio iconico che consente lo sviluppo di altri linguaggi visuali mediante il dialogo e l'interazione tra il programmatore, che crea il linguaggio, e gli utilizzatori finali, che usano questo linguaggio. Mediante una serie di decisioni di icone, regole scelte ed usi del linguaggio tra i due attori (programmatori ed utenti finali) e raffinamenti successivi, si sviluppa il linguaggio iconico.
Altri linguaggi visuali sono stati sviluppati per creare programmi e interfacce grafiche, come nel caso di Fabrik in [28]. Fabrik è un linguaggio visuale attraverso

il quale è possibile manipolare una serie di immagini primitive, chiamate anche box, per creare componenti ed applicazioni. La manipolazione avviene attraverso la concatenazione mediante l'uso di fili, ovvero archi, tra queste icone primitive e dati. La concatenazione delle icone crea un diagramma di tipo data flow e rappresenta il comportamento del componente realizzato.
Nel prossimo paragrafo si discuterà brevemente di famosi linguaggi di programmazione visuali.

1.4 Classificazione dei linguaggi di programmazione visuale

I VPLs possono essere distinti secondo due classificazioni [9, 50, 34, 62, 11, 30]: sulla base alle loro caratteristiche operative e grafiche o sulla base d'uso e dello stile grafico.
Nel seguito si descriveranno entrambe le classificazioni.

1.4.1 Classificazione mediante caratteristiche operative e grafiche

La classificazione in base alle caratteristiche operative e grafiche dei VPLs può essere sintetizzata come segue:

1. Linguaggi puramente iconici.
2. Sistemi iconici e testuali ibridi.
3. Sistemi con modello programmazione-per-esempi.
4. Sistemi orientati ai vincoli.
5. Sistemi basati sulle finestre.

In generale, alcuni programmi possono rientrare in più di una famiglia di questa classificazione, in accordo alla loro modalità d'uso.
Nei linguaggi puramente visuali la programmazione avviene manipolando oggetti visuali appartenenti interamente all'ambiente di sviluppo, e non viene generato alcun codice sorgente testuale da dover essere compilato in un altro linguaggio di programmazione. Esempi classici di linguaggi iconici riconducibili a questa categoria sono Pict [20] e Prograph [13]. Un altro esempio, di recente implementazione, è Scratch [37, 38, 52].
I sistemi ibridi iconici e testuali forniscono un approccio visuale alla programmazione traducendo un linguaggio iconico in un linguaggio testuale, così da ottenere

un codice sorgente testuale partendo da un linguaggio più astratto e meno formale. Alcuni esempi rientranti in questa categoria sono Rehearsal World [15, 16], Mit App Inventor [18, 21, 61, 49], Raptor [12].
Diversamente, i sistemi con modello di programmazione per esempi sono linguaggi in cui ciascun programma fa riferimento ad esempi definiti precedentemente. Un esempio può essere un'icona creata da un utente e "richiamata", come se fosse una funzione, per essere usata successivamente. Esempi di questo genere sono Pygmalion [54, 55, 15] e LabView [58].
I sistemi orientati ai vincoli sono, tipicamente, linguaggi iconici che simulano un qualche vincolo come, ad esempio, leggi naturali della fisica. Esempi molto conosciuti di questi linguaggi sono Thinglab [8, 7], Sketchpad [56, 15] e linguaggi, che sono anche ambienti di sviluppo, come alcuni motori di simulazione fisica per giochi, ad esempio Unreal Engine [33] e Unity3D [14].
Infine, i sistemi basati sulle finestre sono linguaggi visuali che forniscono una rappresentazione del programma che si sviluppa utilizzando una sequenza di finestre. Questa sequenza rappresenta il funzionamento di un programma come una successione temporale di finestre, o istantanee, del programma stesso. Un esempio di questi linguaggi è Form/3 [10].

1.4.2 Classificazione sulla base d'uso e dello stile grafico

La seconda classificazione, che descrive i linguaggi di programmazione visuale mediante l'uso e lo stile grafico, considera quattro famiglie di visualizzazione:

1. Control flow.
2. Data flow.
3. State flow.
4. Topology.

La famiglia dei linguaggi Control flow è stata da sempre utilizzata per la descrizione del comportamento di un algoritmo mediante la successione di un insieme di azioni, opportunamente ordinate, che esplicitano il funzionamento dell'algoritmo stesso. La rappresentazione grafica di questi linguaggi di programmazione visuale è stata un valido strumento di supporto per gli sviluppatori. La raffigurazione grafica degli elementi di questa famiglia è fornita da una serie di oggetti visuali connessi tra di loro mediante degli archi, o frecce. Ogni icona può avere una freccia in entrata e/o una in uscita. Alcune icone hanno delle frecce che generano un

annidamento, come gli oggetti che esprimono il concetto di ciclo o gli oggetti che esplicitano il concetto di condizione.
I linguaggi Control flow si dividono in: flowcharts e diagrammi di Nassi-Shneiderman [34], quest'ultima è una rappresentazione funzionale dei flowcharts in una raffigurazione grafica compatta.
Un valido esempio di programma per i diagrammi flowchart è Raptor, mentre un tipico esempio per i diagrammi di Nassi-Shneiderman è Scratch.
A differenza della famiglia dei linguaggi Control flow, che si concentra sulla descrizione del flusso di controllo, la famiglia di linguaggi visuali Data flow mostra una rappresentazione visuale di come i dati vengono manipolati da un programma durante la sua esecuzione e delle azioni che l'applicativo svolge. In questa rappresentazione gli oggetti visuali sono, generalmente, dati e costrutti elementari. Il flusso è esplicitato mediante l'uso di archi orientati. Esempi ben noti sono Labview e Microsoft VPL.
La famiglia dei diagrammi State flow fornisce una rappresentazione visuale di come un sistema cambia in accordo agli eventi generati sul sistema. Gli oggetti rappresentano uno stato del sistema e gli archi connettono tra loro gli stati in accordo ad una relazione di transizione tra stati. Esempi noti sono alcuni moduli di Unreal Engine e Unity3D.
Nella famiglia di linguaggi Topology, la struttura visuale degli oggetti viene raffigurata mediante l'uso di schemi o moduli interconnessi tra loro mediante delle frecce. Questi schemi e moduli vengono raffigurati mediante dei box, o finestre, e possono contenere codice o dati. Pygmalion e Form/3 sono esempi di questa tipologia di linguaggio.

1.5 Linguaggi di programmazione visuale noti in letteratura

Nel seguito si illustreranno brevemente alcuni linguaggi di programmazione visuale.

1.5.1 Sketchpad

I linguaggi di programmazione visuale nascono formalmente con Sketchpad [56, 15]. Questa applicazione venne definita come la prima applicazione di computer grafica e fornì, nolto prematuramente, i concetti di "oggetti" ed "istanze" nella programmazione, molti anni prima dei linguaggi di programmazione object oriented. Inoltre, Sketchpad introdusse il concetto di interazione uomo-macchina e di interfaccia grafica, realizzando una prima rudimentale versione di Graphical User Interface (GUI). Sketchpad permetteva di disegnare alcuni tipi di oggetti

chiamati primitive, ad esempio quadrati, cerchi e linee, in una grafica 2D usando un display con assi x-y. Tramite queste primitive era possibile creare delle istanze di oggetti. Gli oggetti primitivi sono punti, segmenti, cerchi eds altre costruzioni più complesse che possono essere definite dall'utente e derivate da quelle primitive. La programmazione in Sketchpad avviene definendo visualmente relazioni spaziali e geometriche tra gli oggetti visuali. Queste relazioni spaziali definiscono nuovi oggetti che possono essere chiamati nuovamente primitive. In un linguaggio di programmazione ad oggetti, si potrebbe accostare il significato di primitiva al significato di classe di un oggetto. La definizione di oggetti visuali più complessi avviene definendo nuovi tipi di primitive, in logica spaziale, e tali primitive sono associate univocamente ad uno specifico grafico, detto "*schizzo*". In un certo senso, Sketchpad può essere considerato come il precursore di Geogebra. Sketchpad è di grande interesse e rilievo in quanto introduce il concetto di ereditarietà mediante il quale modificando un tipo di primitiva automaticamente vengono modificate anche tutte le sue istanze di un oggetto e tutti gli altri oggetti che derivano da quel tipo di primitiva, ovvero modificando una relazione tra gli oggetti di una primitiva, ogni oggetto definito con questa primitiva subisce la stessa modifica. Sketchpad è un'applicazione per desktop e non permette la cooperazione.

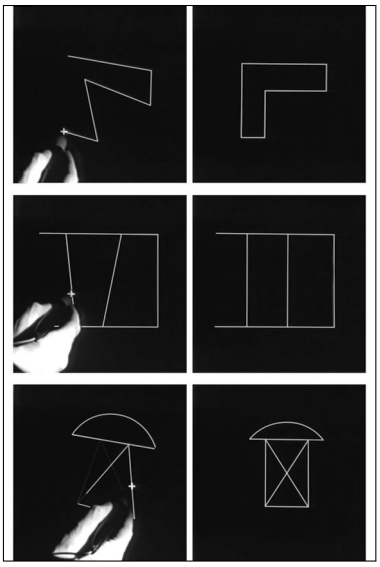


Figura 1.2: Esempio di Sketchpad.

1.5.2 Pygmalion

Pygmalion [54, 55, 15] è uno dei primi programmi iconici che permettono di programmare usando le icone come linguaggio di sviluppo. Sviluppato successivamente a Sketchpad, il suo approccio alla programmazione è di tipo "programmazione per esempi". Lo sviluppatore utilizza oggetti visuali, icone, per definire funzioni. Ogni icona rappresenta un'azione visuale. Le funzioni, definite dall'utente, sono anch'esse delle icone che attuano un comportamento. Questo comportamento è definito dallo sviluppatore nel seguente modo: per definire un'icona, l'utente deve cliccare inizialmente su un'icona "define" il quale pone Pygmalion in uno stato di "remembering", ovvero, il programma inizia a memorizzare ogni singola azione eseguita in successione dall'utente. Queste azioni, definiscono il comportamento della funzione che viene "memorizzata". Quindi, se l'utente inserisce alcune icone all'interno dell'area visuale, il sistema memorizzerà queste azioni per eseguirle quando la funzione definita sarà richiamata. Se l'utente trascina alcune variabili in una posizione dello schermo, il sistema lo memorizzerà. Anche la digitazione di valori è memorizzata. La funzione è definita quando l'utente clicca sull'icona "done" ed indica a Pygmalion che non è necessario memorizzare ulteriori azioni per questa funzione. Per Pygmalion, la programmazione consiste nel memorizzare le azioni che lo sviluppatore esegue per definire le funzioni, per poi ripeterle quando queste funzioni vengono richiamate dallo sviluppatore. Quindi, la programmazione avviene per dimostrazione di esempi che lo sviluppatore realizza e l'utilizzo di funzioni significa far ripetere a Pygmalion la stessa sequenza di azioni che lo sviluppatore ha eseguito. Questo linguaggio rappresenta un primo tentativo, riuscito, di linguaggio di programmazione visuale. Anche Pygmalion è un'applicazione per desktop e non permette la cooperazione.

1.5.3 Labview

Labview [58] risulta essere uno dei linguaggi di programmazione visuale maggiormente riusciti e longevi. Sviluppato nel 1986 inizialmente per l'automazione dei processi industriali, per la loro idealizzazione, progettazione ed i loro test. Fornisce due tipi di visualizzazioni parallele: un pannello frontale, dove vengono visualizzati graficamente uno o più strumenti (manopole, interruttori, led, etc); ed un altro pannello dove si sviluppa, mediante un linguaggio di programmazione visuale di tipo data flow, il funzionamento del pannello frontale. Il pannello frontale serve a dare al programma i valori richiesti ed a testare le funzionalità implementate mediante il linguaggio di programmazione. Gli elementi visuali del linguaggio sono icone che rappresentano dati, variabili, costanti, strutture più complesse o altre funzioni. Gli oggetti visuali del linguaggio vengono collegati tra di loro mediante

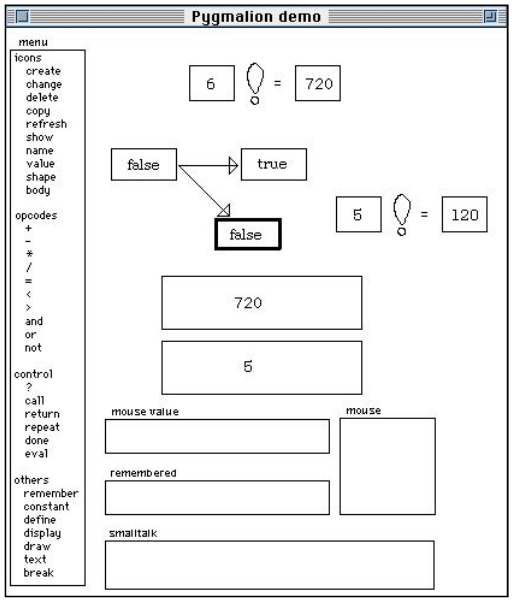


Figura 1.3: Esempio di Pygmalion.

archi. Alcune icone possono avere uno o più archi incidenti e/o uscenti oppure nessuno. Generalmente, le variabili e le costanti sono valori forniti dall'utente mediante la manipolazione degli oggetti visuali all'interno del pannello frontale. Al variare dei valori degli strumenti il programma viene avviato e fornisce un output in qualche altro strumento del pannello frontale. Tipicamente Labview viene utilizzato per la creazione di prototipi di strumenti industriali. Anche Labview è un'applicazione stand-alone, ovvero è un programma sviluppato per essere eseguito in un computer e non permette la cooperazione del progetto, in tempo reale.

1.5.4 Scratch

Scratch [37, 38] è un linguaggio di programmazione visuale, sviluppato in linguaggio Flash, ed è uno dei primi linguaggi di programmazione visuale che viene eseguito nel web. È attualmente, quasi, l'unico linguaggio di programmazione visuale utilizzato per l'insegnamento della programmazione, generalmente dalla scuola primaria alla scuola secondaria di secondo grado. Scratch è sia un linguaggio di programmazione visuale sia un ambiente di sviluppo. Il linguaggio iconico fornisce un insieme di istruzioni, procedure ed azioni, già implementate, da eseguire su degli oggetti, immagini, come sprite od icone, associate a variabili. Lo sviluppo di algoritmi avviene incastrando delle icone tra di loro, generando, per ogni algoritmo, un diagramma iconico simile ai diagrammi di Nassi-Shneiderman.

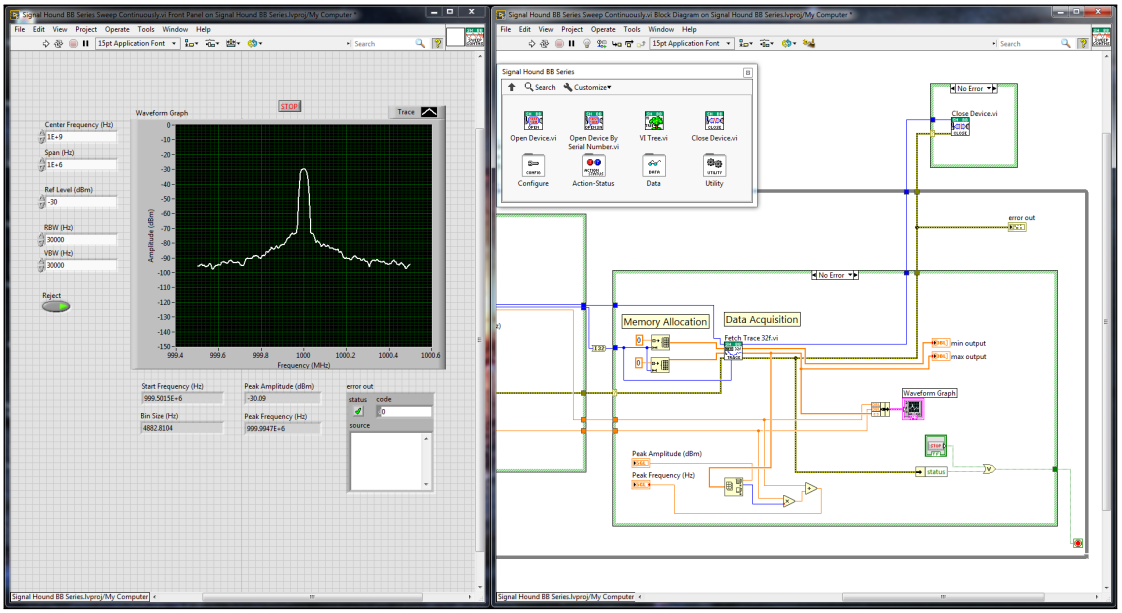


Figura 1.4: Esempio di Labview.

Le procedure sono invocate quando si verifica un evento. Scratch non fornisce un codice sorgente visuale. Permette di sviluppare ed eseguire programmi direttamente nell'ambiente visuale stesso. Permette la condivisione di un progetto, tuttavia non permette la cooperazione in tempo reale tra gli utenti. Scratch risulta essere un linguaggio di programmazione visuale utile, permettendo ad utenti meno esperti di familiarizzare, giocando, con la programmazione. Tuttavia, gli autori stessi in [38] riferiscono che "... Tuttavia, l'astrazione procedurale è una delle 'potenti idee' dell'informatica e le procedure hanno un valore pratico come modo per strutturare il codice man mano che i progetti crescono. Il team di Scratch sta valutando la possibilità di reintrodurre le procedure come modo per gli utenti di definire i propri blocchi di comando.". Scratch, dunque, avvicina gli utenti inesperti alla programmazione senza però fornire una consapevole astrazione concettuale di programmazione utile all'implementazione di procedure, e quindi allo sviluppo del pensiero computazionale. Inoltre, Scratch non fornisce un codice sorgente del programma sviluppato, e quindi gli utenti non sono consapevoli di come in realtà viene tradotto il linguaggio iconico in un linguaggio compilativo.

1.5.5 MIT App Inventor

È un linguaggio di programmazione iconico derivato da Scratch. MIT App Inventor [18, 21, 61, 49] permette lo sviluppo di programmi per il sistema operati-

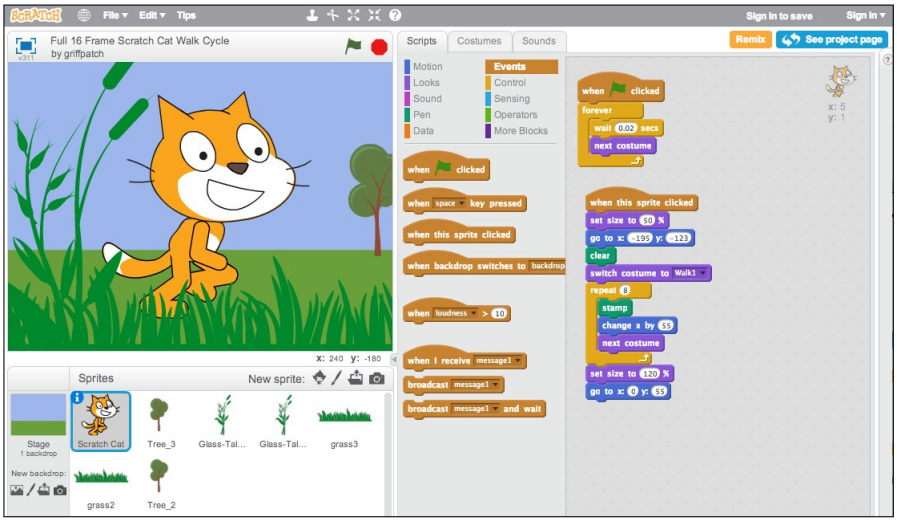


Figura 1.5: Esempio di Scratch.

vo Android. Fornisce due modalità di visualizzazioni di un progetto: una sezione "designer", dove lo sviluppatore inserisce alcuni oggetti con cui l'utente finale deve interfacciarsi; ed un'altra sezione chiamata "Blocks", dove il programmatore crea gli algoritmi mediante l'uso del linguaggio visuale di Scratch. Anche MIT App Inventor è un programma che funziona in un browser[1], tuttavia anche in questo caso non vi è la possibilità della cooperazione con altri utenti. Inoltre, quando si sviluppano molti algoritmi, o pochi ma con una discreta dimensione e complessità, lo sviluppatore può confondersi a causa della complessa visualizzazione degli algoritmi che non rende immediata la comprensione di quali siano gli algoritmi (che si stanno vedendo) e quando vengono utilizzati.

1.5.6 Raptor

Raptor [12] è un linguaggio di programmazione visuale che utilizza uno stile diagrammatico control flow. È stato sviluppato esclusivamente per i computer, e non permette la cooperazione. Gli oggetti visuali, le icone chiamate "Simboli", che Raptor mette a disposizione per programmare sono 6: l'assegnazione, la chiamata alle funzioni, l'input, l'output, la selezione ed il ciclo. L'assegnazione permette di assegnare valori alle variabili. La chiamata alle funzioni permette di usare funzioni

[1]Un browser, anche detto web browser, è un applicativo che permette la navigazione in internet utilizzando, generalmente, il protocollo Http, HyperText Transport Protocol, e traduce un testo, scritto nel linguaggio di formattazione HTML5, visualizzandolo come una pagina web.

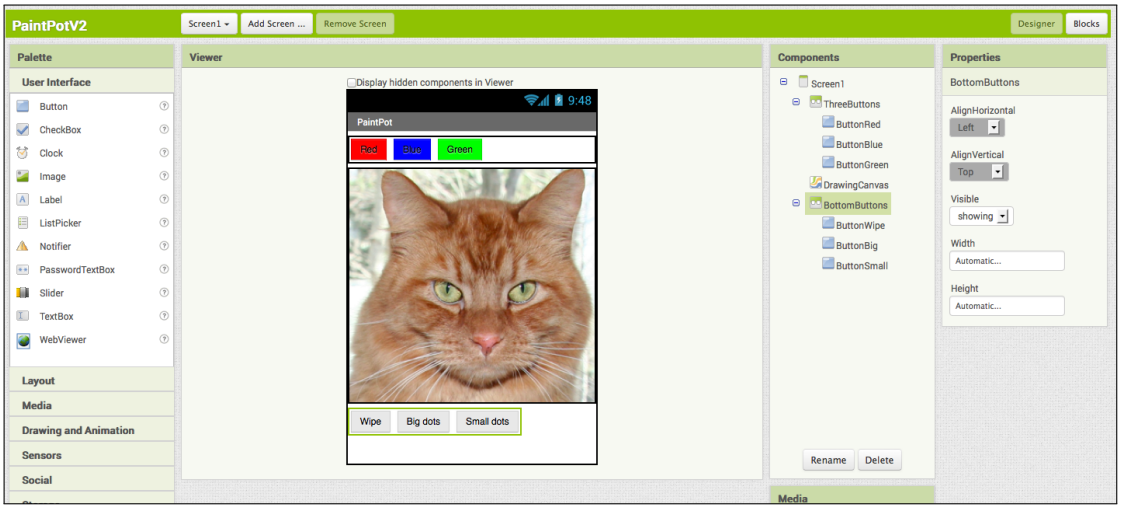


Figura 1.6: Esempio di sviluppo di interfaccia grafica con Mit App Inventor.

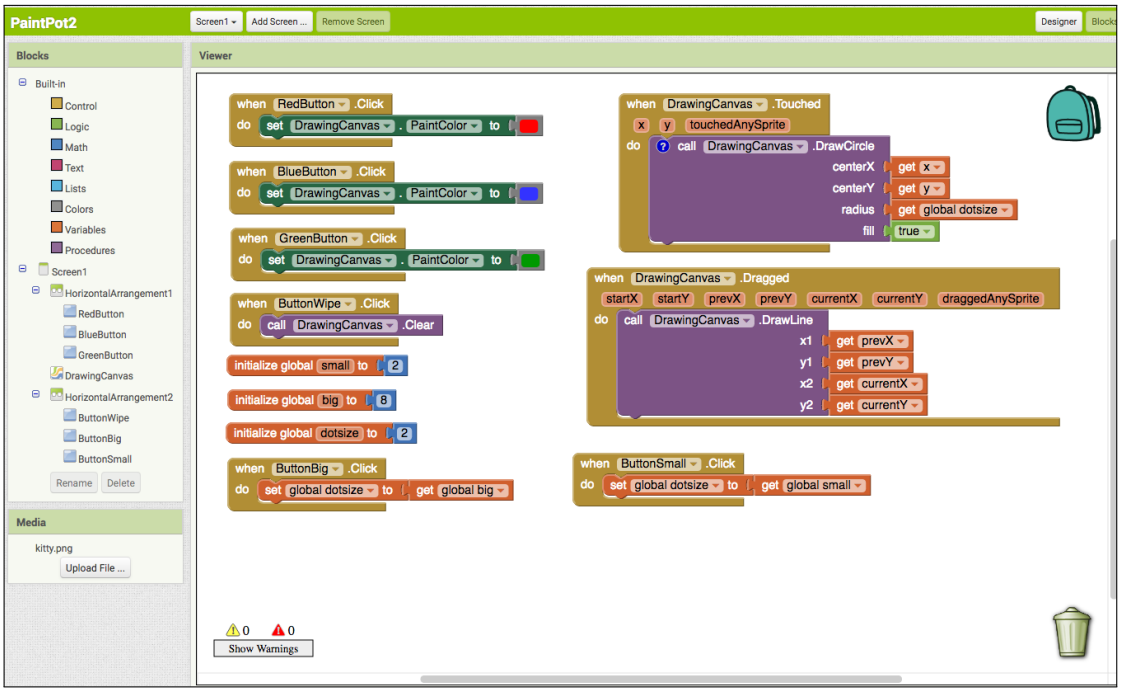


Figura 1.7: Esempio di sviluppo di algoritmi con Mit App Inventor.

definite dall'utente o fornite dal sistema stesso. Il simbolo di input permette di acquisire valori e di assegnarli alle variabili. L'icona di output stampa stringhe o numeri in una finestra chiamata "console" (non è una console di sviluppo ma una semplice finestra di output). L'icona di selezione è un costrutto visuale che

simboleggia una condizione. Il simbolo di ciclo permette di realizzare cicli diagrammatici. Inoltre, Raptor può generare un codice sorgente testuale del programma sviluppato iconicamente, e fornisce una astrazione concettuale dell'algoritmo, tuttavia non permette di familiarizzare con funzioni proprie di uno specifico linguaggio di programmazione in quanto la programmazione è molto, forse troppo, astratta. Inoltre, non è permesso all'utente di modificare la posizione delle icone che vengono inserite durante lo sviluppo visuale, ovvero l'utente non può spostare liberamente in tutta l'area visuale le icone a proprio piacimento (queste vengono posizionate automaticamente dal programma), ma può solo spostare l'ordine delle icone nel flusso del diagramma. Raptor è anch'esso un programma standalone per computer e non permette la cooperazione.

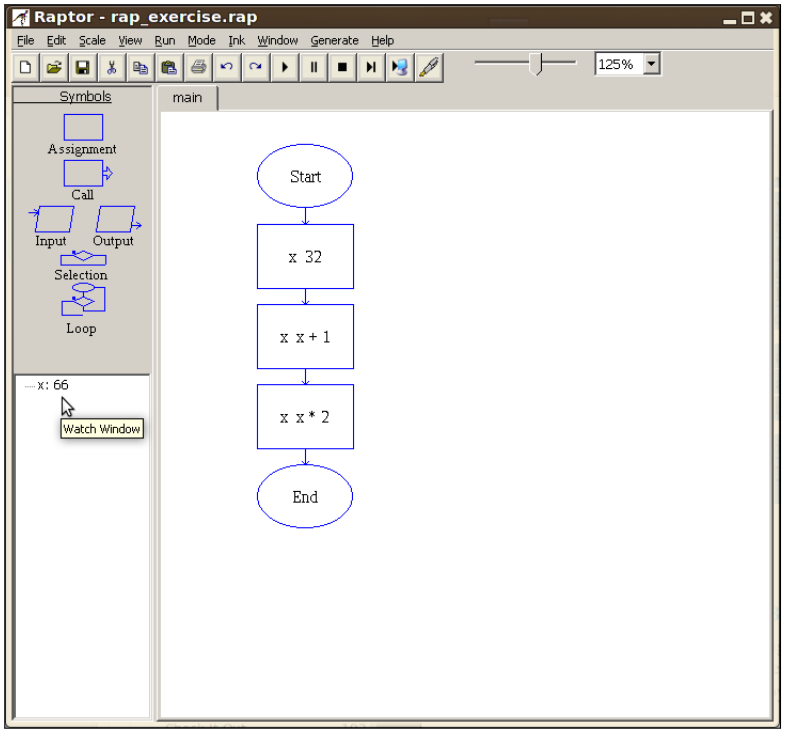


Figura 1.8: Esempio di Raptor.

1.6 Ragioni di sviluppo di un nuovo framework iconico

Molti dei programmi descritti, ed in generale presenti in letteratura, possiedono caratteristiche utili in accordo alle finalità che si sono previste. Tuttavia, al

meglio della mia personale conoscenza, non è stato ancora implementato un linguaggio di programmazione visuale utile ad insegnare l'arte della programmazione ed a sviluppare e potenziare il pensiero computazionale. Inoltre, molti programmi sono stati sviluppati principalmente per desktop o computer portatili, limitandone fortemente l'usabilità. Altri linguaggi sono stati sviluppati per essere utilizzati attraverso programmi web browser, ma sono, ancora, limitati ad un uso standalone. Inoltre, nessuno di tutti i linguaggi di programmazione visuale permette la cooperazione, in tempo reale, tra gli utenti, non consentendo di usufruire di metodologie didattiche innovative o, almeno, diverse dall'insegnamento e dall'apprendimento tradizionale. Il lavoro di questa tesi è l'analisi e lo sviluppo di un framework innovativo, capace di sopperire alle profonde mancanze, sopra discusse, dei linguaggi di programmazione. Tale linguaggio di programmazione visuale rientra nella classificazione dei linguaggi di programmazione dei sistemi iconici e testuali ibridi, i sistemi con modello di programmazione per esempi ed i linguaggi di programmazione di tipo control flow. Nel secondo capitolo si discuterà delle caratteristiche che un linguaggio di programmazione iconico, dedicato all'insegnamento e all'apprendimento della programmazione, deve possedere; nel terzo capitolo si discuterà dello sviluppo del linguaggio iconico; nel quarto capitolo verranno discusse le funzioni più importanti del framework e delle sue componenti principali. Nel quinto capitolo si discuterà ed analizzerà un case study, che è stato utilizzato per eseguire una sperimentazione del framework con i suoi risultati. Nel sesto capitolo si discuteranno le conclusioni e le possibili estensioni e lavori futuri. Nelle appendici A e B si discute, rispettivamente, dell'implementazione della componente server e client del framework. Infine, nelle appendici C e D si forniscono, rispettivamente, il manuale d'uso del framework e la sua installazione in un ambiente linux.

Capitolo 2

Caratteristiche di un linguaggio di programmazione visuale per la didattica della programmazione

2.1 Caratteristiche dei linguaggi di programmazione iconici

Uno tra i principali obiettivi a cui tendono i linguaggi di programmazione visuali è quello di svincolare gli sviluppatori dai vincoli della programmazione testuale. Ognuno di essi possiede alcune caratteristiche grafiche utili [9, 50, 34, 62, 11, 30] e coinvolgenti vari aspetti cognitivi [19, 36, 35, 42, 41, 40, 44, 43, 45]. Sulla base di tali caratteristiche l'utente apprende diverse modalità di programmazione.

2.2 Caratteristiche grafiche ed aspetti cognitivi

È importante per la programmazione che i linguaggi iconici abbiano le seguenti caratteristiche grafiche:

- Manipolazione diretta degli elementi grafici.
- Visualizzazione del software.
- Visualizzazione delle informazioni.
- Rappresentazione e ragionamento diagrammatico.
- Simulazioni grafiche.

- Semantica e sintassi iconica.
- Chiarezza di definizione.
- Concretezza.
- Immediatezza.
- Feedback visuale.
- Astrazione concettuale.

2.2.1 Manipolazione diretta degli elementi grafici

La manipolazione diretta è un'azione che l'utente può eseguire direttamente sugli elementi visuali, ovvero sulle icone, di un linguaggio iconico. Essa permette la modifica delle informazioni memorizzate negli oggetti visuali, interagendo direttamente col sistema, e apporta modifiche visuali immediatamente riscontrabili agendo anche da feedback visuale. Questa interazione produce un cambiamento nel sistema, localizzato alla sola icona o nell'intero sistema. Questo cambiamento deve essere immediatamente rappresentato visualmente producendo nell'utilizzatore un'associazione univoca tra immagine ed azione. Al fine di poter essere percepiti come reali, i linguaggi iconici devono permettere di modificare direttamente i dati assegnando ad essi valori differenti in qualunque momento della programmazione iconica. Questo permette all'interazione uomo-macchina di rappresentare, cognitivamente, un'azione diretta eseguita su un dato o una struttura di dati, come ad esempio l'assegnazione di valori a variabili e costanti.

2.2.2 Visualizzazione del software

L'importanza della veste grafica è fondamentale. La visualizzazione quanto più sarà semplice e pulita tanto più sarà maggiormente facile comprendere le varie connessioni tra gli oggetti ed il flusso logico delle istruzioni e dati. Pertanto, la visualizzazione del software implicherà un ridotto sforzo cognitivo ma, nel contempo, un immediato ed intuitivo apprendimento del comportamento logico di un algoritmo e delle varie componenti del programma in fase di sviluppo, soprattutto quando gli algoritmi risultano essere complessi.

2.2.3 Visualizzazione delle informazioni

La simbologia iconica deve avere una capacità di sintesi anche nella sua veste grafica e quindi deve visualizzare le informazioni di ogni icona anche nel suo

aspetto grafico. Così, icone diverse devono evidenziare un significato diverso. Le icone possono rappresentare dati o funzioni. Icone simili in aspetto grafico devono differenziarsi dal contenuto testuale visualizzato. Così, sebbene due icone abbiano stessa forma e colore, per essere differenti devono contenere un testo differente, altrimenti devono considerarsi equivalenti, dal punto di vista del linguaggio iconico.

2.2.4 Rappresentazione e ragionamento diagrammatico

Nei linguaggi testuali gli algoritmi, ed in generale i programmi, vengono sviluppati attraverso una precisa sequenza di azioni in successione. Un linguaggio di programmazione iconico deve utilizzare un aspetto grafico semplice e di immediata comprensione, non soltanto dal punto di vista delle nozioni fornite, ma anche dal punto di vista delle relazioni che intercorrono tra gli oggetti visuali. In una rappresentazione diagrammatica le relazioni sono esplicitate in maniera tale che l'utente possa immediatamente capire le interazioni tra le varie icone e, quindi, il significato dell'insieme delle icone. Questa rappresentazione ottiene una serie di vantaggi: le implicazioni logiche della sequenza di azioni è più comprensibile, la comprensione del comportamento di un algoritmo diviene più intuitivo, infine la modifica delle proprietà delle icone, come valori o azioni, permette una più facile condiscendenza dell'effetto sull'intero sistema, o anche su una porzione di esso.

2.2.5 Simulazioni grafiche

In un linguaggio di programmazione visuale la simulazione grafica è molto importante in quanto fornisce una riproduzione del funzionamento del programma iconico sviluppato. L'importanza della simulazione grafica deriva dal fatto che gli utenti possono avere la percezione che lo sviluppo, per mezzo del codice iconico, del programma non è solo astratto ma è concreto e produce un programma reale e tangibile che può essere eseguito. Un linguaggio visuale di programmazione che permette di simulare, o di visualizzare, il risultato finale (come ad esempio il programma compilato) del processo di sviluppo svolto, offre una caratteristica importante in quanto rende tangibile e concreto lo sviluppo iconico producendo un programma reale.

2.2.6 Semantica e sintassi iconica

I Visual Programming Languages, come nei linguaggi di programmazione testuali, devono possedere una precisa semantica e sintassi. Le icone non possono e non devono essere associate casualmente, ma con un preciso riferimento come nei linguaggi testuali. Ad esempio, ipotizzando che un linguaggio iconico abbia

regole di sintassi in cui alcuni elementi non possono essere collegati tra loro ed altri elementi si, allora questo linguaggio iconico deve prevedere, e vietare, casi di configurazione iconica in cui siano collegati tra loro elementi grafici che violino queste regole. Anche la semantica del linguaggio iconico è importante. Gli elementi visuali devono essere realizzati in maniera tale da rendere immediato il significato dell'icona senza alcuna ambiguità.

2.2.7 Chiarezza di definizione

Nella rappresentazione diagrammatica, le relazioni tra gli oggetti visuali avvengono tramite l'uso di archi e frecce. Ogni oggetto è collegato ad uno o più oggetti a seconda della natura dell'oggetto stesso ed alle regole del suo linguaggio. Un oggetto è collegato ad un altro oggetto mediante un arco, mentre la freccia definisce una direzione, come nei grafi direzionali, ma esplicita univocamente il significato di consequenzialità delle azioni, ovvero se due oggetti A e B sono collegati mediante un arco che indica una direzione da A verso B, questo arco definisce una univoca consequenzialitá tra le due icone senza ambiguità. Come detto, un algoritmo sviluppato in linguaggio testuale è una sequenza corretta e precisa di azioni. Alcune azioni possono essere inserite in diversi modi. Ad esempio, se si dovesse eseguire la stampa del nome e cognome di un utente, è necessario che l'algoritmo chieda all'utente di inserire il suo nome e il suo cognome e solo successivamente l'algoritmo può stampare il nome e il cognome. Per un algoritmo di questo genere è ininfluente conoscere il nome prima o dopo il cognome, ma è necessario che l'algoritmo li conosca entrambi prima di doverli stampare. Usando i linguaggi testuali, gli sviluppatori possono definire gli algoritmi inserendo, a loro piacimento, le dovute azioni in una successione che loro ritengono più opportuna. Un linguaggio iconico deve poter prevedere la stessa semplicità e la stessa chiarezza nella definizione degli algoritmi, senza ambiguità ed al contempo fornendo un alto grado di flessibilità del suo utilizzo.

2.2.8 Concretezza

In un linguaggio di programmazione visuale, le icone sono una parte di esso e gli oggetti devono essere percepiti come oggetti reali e "fisici", ovvero concreti, ma in uno spazio non necessariamente tridimensionale o "reale". Questo avviene attraverso l'uso della corretta associazione e corrispondenza dei valori inseriti nelle varie icone utilizzate, che corrispondono, quindi, ad un reale contesto circoscritto e coerente al tipo di oggetto visuale, ed avviene, inoltre, per mezzo delle azioni che possono essere eseguite sugli oggetti e per mezzo delle modifiche che tali azioni provocano negli e tra gli oggetti e nell'intero sistema, come ad esempio se due oggetti A e B fossero collegati mediante un arco, allora lo spostamento nello spazio

esercitato sull'oggetto B dovrebbe modificare automaticamente la lunghezza di questo arco.

2.2.9 Immediatezza

Questa caratteristica può essere espressa come una distanza, breve, che sussiste tra uno fine e l'azione esercitata per conseguire tale fine. Questa caratteristica è importante in quanto più si dilata il tempo necessario per ottenere questo scopo, maggiore sarà il calo cognitivo nell'associare l'azione al fine. Ovvero, nella programmazione avviene un'associazione uno a uno tra un problema reale, o concettuale, e il corrispettivo problema computazionale. Maggiore è la vicinanza tra questi due elementi, in termini di distanza, più facilmente essi sono associabili. Questa distanza può utilizzare una metrica consona al sistema sviluppato ed al linguaggio iconico realizzato. Questa metrica potrebbe essere il tempo necessario ad ottenere uno scopo mediante un'azione su un'icona, oppure il numero di finestre aperte per modificare un valore in una struttura dati, ecc. L'immediatezza è una proprietà importante per i VPLs [22, 25, 23, 24] ed è necessario che sia opportunamente valutata nello sviluppo di un linguaggio di programmazione iconico.

2.2.10 Feedback visuale

Il feedback visuale è una caratteristica importante in un linguaggio testuale [20], ma lo è ancor di più in un linguaggio iconico. In un linguaggio visuale, il feedback visuale rappresenta la visualizzazione automatica degli effetti delle azioni esercitate sugli oggetti visuali. Il feedback visuale è fortemente associato alla manipolazione diretta degli oggetti iconici. Manipolare un'icona, ovvero modificarne ad esempio il suo valore, deve ottenere un cambiamento di stato nell'oggetto stesso. Questo cambiamento deve essere mostrato attraverso una modifica nell'icona o nel sistema visuale. Inoltre, questo cambiamento deve essere immediato, o, al più, manifestato mediante una transizione visuale delimitata da un tempo prestabilito, e certamente non casuale.

2.2.11 Astrazione concettuale

Un linguaggio di programmazione visuale, che voglia sviluppare e potenziare il pensiero computazionale, deve essere anche concettuale. L'astrazione concettuale è il significato di un concetto o di una idea. L'importanza della semplificazione iconografica e dell'associazione delle icone a concetti precisi è un aspetto fondamentale e non trascurabile. Nei linguaggi testuali, generalmente, il nome di una variabile, costante o funzione, suggerisce concettualmente uno specifico significato

oppure il tipo di dato, ad esempio, *int* di una variabile suggerisce che tale variabile può assumere, concettualmente, qualunque valore all'interno del range dei numeri *int*. In un linguaggio di programmazione visuale, le icone devono essere associate ad un concetto ben definito. Infatti, gli elementi visuali dei linguaggi simbolici ed iconografici sono associati ad uno specifico significato, derivato dal contesto o dal linguaggio stesso. Ad esempio, il simbolo + può assumere diversi significati: in un contesto matematico questo simbolo assume il significato di operazione di addizione, o in un contesto artistico può significare una croce greca, o ancora in un contesto religioso potrebbe significare la croce cristiana, etc. Dunque, l'astrazione concettuale è una caratteristica importante e da valutare attentamente durante la costruzione di un linguaggio di programmazione visuale per la didattica della programmazione.

2.2.12 Caratteristiche di un linguaggio iconico completo

Lo sviluppo e il potenziamento del pensiero computazionale è importante, se non fondamentale, per lo sviluppo di buoni programmi ed algoritmi corretti. Un linguaggio di programmazione visuale utile per la didattica, per le scuole di ogni ordine e grado e per l'università, dovrebbe possedere ognuna delle precedenti caratteristiche elencate per sviluppare e potenziare il pensiero computazionale. Tuttavia, questo potrebbe non bastare: il solo sviluppo del pensiero computazionale non è una condizione sufficiente per imparare l'arte della programmazione, ma lo è per il ragionamento. Come precedentemente detto nel paragrafo 1.1, gli studenti hanno anche difficoltà nell'apprendimento della sintassi e della semantica di un linguaggio di programmazione. Quindi, sarebbe auspicabile che ad un linguaggio di programmazione iconico sia associato un linguaggio di programmazione visuale che possa aiutare gli studenti a comprendere un linguaggio di programmazione non iconico aiutandoli ad apprendere la corretta sintassi e semantica di un linguaggio di programmazione specifico. In tale maniera, l'insegnamento e l'apprendimento dell'arte della programmazione sarebbe facilitato e completo.

Capitolo 3

Sviluppo di un framework di programmazione visuale per la didattica della programmazione

3.1 Scelta del modello di visualizzazione del linguaggio iconico

Nei precedenti capitoli si sono introdotti alcuni linguaggi di programmazione visuali e le relative caratteristiche, e le caratteristiche che dovrebbe possedere un linguaggio di programmazione visuale per lo sviluppo e il potenziamento del pensiero computazionale e per l'insegnamento e l'apprendimento della sintassi e della semantica di un linguaggio di programmazione. Lo sviluppo visuale diagrammatico risulta essere ragionevolmente corretto per la rappresentazione dei diversi passi computazionali che un algoritmo deve eseguire. Questa rappresentazione, molto schematica nel raffigurare le azioni eseguite in un algoritmo, aiuta lo studente a sviluppare un insieme di processi cognitivi logico-risolutivi propri del pensiero computazionale, formandolo e supportandolo. La scelta di utilizzo del modello control flow, appare, quindi, corretta. Tuttavia, come già detto nel paragrafo 2.2.12, solo tale modalità non è sufficiente: è necessario supportare il linguaggio iconico con un altro modello di visualizzazione per aiutare l'insegnante ad insegnare, ed allo studente ad apprendere, la sintassi e la semantica. Considerando che ogni linguaggio di programmazione ha una propria sintassi, ma che molti possiedono una sintassi di linguaggio C-like, si è scelto di voler adottare il linguaggio C, come linguaggio di basso livello. Inoltre, il linguaggio C è uno dei linguaggi più diffusi e studiati. Inoltre, in accordo al TIOBE index[1] il linguaggio C, attualmente, è tra i primi 5

[1]La definizione di Tiobe index può essere trovata al seguente indirizzo `https://www.tiobe.com/tiobe-index/programming-languages-definition/`, mentre la classifica dei linguaggi di

linguaggi di programmazione più diffusi.
La combinazione di due linguaggi visuali, uno rappresentativo del comportamento logico degli algoritmi da sviluppare e l'altro rappresentativo dei costrutti propri del linguaggio C sembra essere appropriato in quanto mira a completare l'offerta formativa dell'insegnamento e dell'apprendimento dell'arte della programmazione. Infatti, attraverso l'uso dei diagrammi di flusso è possibile sviluppare concettualmente un algoritmo, modellarlo ed implementarlo. Inoltre, attraverso l'uso di un linguaggio che rappresenta la sintassi e la semantica è possibile apprendere la corretta scrittura di un codice testuale di programmazione. In questa maniera, il discente focalizza contemporaneamente la sua attenzione sui più importanti concetti della programmazione: l'astrazione concettuale di un problema, una sua risoluzione e la relativa traslazione in un linguaggio di programmazione.

3.2 Carenze generali dei linguaggi di programmazione visuale

Gran parte dei linguaggi di programmazione visuale, e i loro ambienti di sviluppo, è stata sviluppata per essere eseguita localmente nei computer e nei laptop, ed in alcuni casi anche su tablet e smartphone, quindi per essere eseguiti su specifici sistemi operativi, mentre pochi linguaggi sono stati sviluppati per essere utilizzati su un browser, indipendentemente dal dispositivo e dal sistema operativo utilizzato. Alcuni linguaggi permettono la condivisione dei progetti realizzati, ma non in tempo reale: l'utente deve salvare il progetto e fornirlo ad altri utenti in un secondo momento. Alcuni linguaggi iconici forniscono l'analogo codice sorgente di tipo testuale degli algoritmi iconici sviluppati, ma questo non è condiviso con gli altri utenti. Comunque, al meglio della propria conoscenza, tutti i linguaggi di programmazione visuale non permettono una collaborazione e cooperazione in tempo reale tra altri utenti del sistema, ovvero gli utenti non possono contemporaneamente collaborare e cooperare all'implementazione del progetto. Questa è una grossa lacuna nel campo della didattica della programmazione, in quanto non è possibile realizzare gruppi di lavoro cooperanti per lo sviluppo di algoritmi e programmi con gli strumenti attuali. In questo modo, ogni utente deve attendere un altro per continuare lo sviluppo dell'applicazione, nel caso di utilizzo dello stesso file.

programmazione, in accordo al Tiobe index, può essere trovate al seguente indirizzo `https://www.tiobe.com/tiobe-index/`.

3.3 SIRENE

Oggetto e contributo del progetto di questo dottorato di ricerca è la definizione di un framework iconico utile all'insegnamento e all'apprendimento dell'arte della programmazione e dello sviluppo e del potenziamento del pensiero computazionale. La sua realizzazione su browser permette di fornire solide basi di programmazione dal punto di vista dell'apprendimento della sintassi e della semantica di un linguaggio di programmazione testuale che nello specifico risulta essere il linguaggio C. Il framework è denominato "SIRENE", Shared InteReactive ENvironment for Encoding, ovvero ambiente interattivo condiviso per la codifica. SIRENE è utilizzabile attraverso l'uso di un qualunque browser che supporti i linguaggi HTML5 e CSS3 ed è sviluppato in accordo al modello internet client/server. La componente client viene eseguita in qualunque browser mentre la componente server è installata su un computer che usa qualunque sistema operativo. Il client, quindi, è eseguibile in qualunque dispositivo computer, tablet o smartphone a prescindere dal sistema operativo utilizzato. Questa caratteristica permette a SIRENE di essere universalmente utilizzabile. SIRENE, inoltre, permette la condivisione, con tutti gli utenti, dell'ambiente visuale di sviluppo, permettendo di cooperare e collaborare allo sviluppo del programma contemporaneamente ed in tempo reale. Infatti, durante l'uso di SIRENE gli utenti possono interagire col sistema ed ogni interazione esercitata da un utente con il sistema può essere immediatamente visualizzata da ogni altro utente collegato. Questa caratteristica rende SIRENE unico nel suo genere, poiché esso è l'unico framework capace di permettere la cooperazione, l'interazione e lo scambio, in tempo reale, di know how tra tutti gli utenti, e, potenzialmente, di accelerare lo sviluppo e il potenziamento del pensiero computazionale grazie non soltanto all'utilizzo di un linguaggio iconico capace di mostrare la logica di un algoritmo, ma anche grazie all'interazione tra utenti in cui lo scambio di conoscenza permetterebbe di raggiungere soluzioni algoritmiche migliori. Inoltre, il potenziamento del pensiero computazionale e l'apprendimento della sintassi e della semantica dovrebbero essere garantiti in quanto per realizzare un algoritmo gli utenti sono vincolati a sviluppare sia il comportamento logico di un algoritmo e sia le assegnazioni o le espressioni che sono associate ad ogni icona del linguaggio di SIRENE. Queste caratteristiche focalizzano l'attenzione degli utenti esattamente sui due principali problemi esposti nel paragrafo 1.1. Come già accennato nel paragrafo 1.6, SIRENE può essere considerato un linguaggio di programmazione visuale ibrido, in quanto oltre al linguaggio iconico per la rappresentazione concettuale degli algoritmi è stato implementato un altro linguaggio iconico/simbolico/testuale per la aiutare la comprensione della sintassi e della semantica del linguaggio C. L'usabilità di SIRENE in ambito web, facilita l'insegnamento di tipo tradizionale in aula o laboratorio e permette l'insegnamento della programmazione anche in ambito e-learning. Infatti, la sua caratteristica è quella di essere un ambiente web

based condiviso in tempo reale con tutti gli utenti. Ciò permette di espandere le lezioni oltre un ambiente fisico ristretto, quale un'aula o un laboratorio, realizzando lezioni anche in luoghi fisicamente distanti rendendoli virtualmente vicini. SIRENE, quindi, colma la carenza dei linguaggi di programmazione visuale, in quanto realizza, dal lato didattico, lo sviluppo e il potenziamento del pensiero computazionale e l'insegnamento della sintassi e della semantica, e realizza, dal lato tecnico, la cooperazione e la collaborazione in tempo reale tra tutti gli utenti, in ambienti anche diversi ed utilizzando qualunque dispositivo che supporti l'uso di un browser. Nel seguito del capitolo vengono descritte le tecnologie usate per sviluppare SIRENE.

3.4 Panoramica delle tecnologie usate per sviluppare SIRENE

SIRENE è stato sviluppato usando due linguaggi di programmazione, Javascript e Pascal, per lo sviluppo del sistema client/server e due linguaggi di formattazione [2], HTML5 e CSS3, per lo sviluppo dell'interfaccia visuale del framework. Inoltre, sono state utilizzate alcune librerie e framework di comunicazione quali "INDY" (libreria scritta in Pascal realizzata dal gruppo "The Indy Project") e "Node.js"[3], "Espress.js" e "Socket.io". Nel seguito vengono descritte le sopracitate tecnologie e linguaggi usati per lo sviluppo di SIRENE.

3.4.1 Il linguaggio di programmazione Javascript

Javascript è un linguaggio di programmazione interpretato. È il linguaggio di programmazione più diffuso in ambito web per le sue proprietà, caratteristiche e standardizzazione. Fu inventato da Brendan Eich nel 1995. Successivamente fu standardizzato nel 1997 dalla ECMA, European Computer Manufacturers Association International, un'associazione internazionale che si occupa della standardizzazione di diversi linguaggi, tra cui il Javascript e i sistemi di comunicazione. La ECMA rilasciò varie versioni di Javascript, chiamato anche ECMAScript, a causa della necessità di standardizzarlo come linguaggio comune ed a causa delle necessità di inserire nuove proprietà e caratteristiche del linguaggio. Standardizzato

[2]Un linguaggio di formattazione è un linguaggio che permette di organizzare contenuti o elementi visuali, come ad esempio testi ed immagini, in maniera tale che un applicativo, come il browser, visualizzi i contenuti e li disponga correttamente nella pagina da visualizzare.

[3]In accordo alle "Trademark Guidelines of the Node.js Foundation" attualmente reperibili all'indirizzo `https://nodejs.org/static/documents/trademark-policy.pdf`: "Node.js is a trademark of Joyent, Inc. and is used with its permission. We are not endorsed by or affiliated with Joyent".

ormai alla versione 8, la ECMAScript 2018, Javascript è un linguaggio interpretato ed è utilizzato da qualunque browser. Esso può gestire e manipolare ogni oggetto del linguaggio di formattazione HTML5 di cui se ne parlerà in dettaglio nel seguito del capitolo. Javascript permette di avere accesso agli oggetti HTML5, modificando le loro proprietà e comportamenti. Le caratteristiche di Javascript quali la sua facilità d'uso, la sua debole tipizzazione, le sue strutture di controllo e di condizioni, la facilità di creare funzioni e di gestire gli eventi, la sua gestione dei dati, lo hanno reso il linguaggio di programmazione universale in ambito web supportato da qualunque browser di qualsiasi sistema operativo. Javascript, grazie alla sua popolarità ed alla sua caratteristica di essere supportato da qualunque browser di qualsiasi dispositivo, è stato scelto per essere utilizzato come uno dei due linguaggi per sviluppare SIRENE. Javascript utilizza alcune interfacce che vengono rese disponibili dai browser: le interfacce DOM, Document Object Model[4]. Come si vedrà nel prossimo capitolo, in SIRENE, questo linguaggio di programmazione viene utilizzato prevalentemente per la creazione, la manipolazione e la gestione di tutti gli oggetti dell'interfaccia grafica del client di SIRENE e di ogni comunicazione tra i client e il server. Inoltre, viene utilizzato anche per la gestione di comunicazione con un componente server di SIRENE, J-Manager, discusso nel seguito, che permette la gestione dei processi dei programmi generati, compilati ed eseguiti da SIRENE. Mediante l'uso di Javascript è possibile realizzare una tecnica di programmazione web conosciuta come Ajax, Asynchronous JavaScript and XML. L'uso di questa tecnica, permette la creazione di web applications interattive, ma soprattutto permette l'aggiornamento dinamico ed interattivo delle pagine web, modificando soltanto porzioni del codice HTML, senza ricaricare l'intera pagina e senza interrompere la navigazione dell'utente. Questa tecnica avviene in background e mediante uno scambio di informazioni tra il server e il web browser e si avvale dell'uso combinato tra HTML, l'utilizzo del DOM, l'uso dell'interfaccia API[5] XMLHttpRequest (che permette lo scambio di informazioni tra server e browser), ed il formato di comunicazione XML o Json[6].

[4]Il Document Object Model è uno standard del W3C, World Wide Web Consortium, nativamente supportato da tutti i browser e permette di accedere e manipolare gli elementi dei documenti HTML usando Javascript

[5]L'API è l'acronimo di Application Programmable Interface, ovvero interfaccia di programmazione di un'applicazione, ed è un insieme di funzioni e procedure, raggruppate in un oggetto detto interfaccia, messe a disposizione per lo sviluppatore.

[6]JavaScript Object Notation, è un formato di raggruppamento di dati ed è usato in internet per la comunicazione di dati.

3.4.2 Node.js, Espress.js e Socket.io

Node.js [57] è un progetto della Node.js Foundation ed è un programma basato sul motore V8, un applicativo scritto in linguaggio C++ e sviluppato da Google. Esso interpreta files scritti in linguaggio Javascript e realizza motori server e files client scritti in Javascript per creare applicazioni internet performanti e robuste. Inoltre, Node.js interpreta ed esegue files scritti in linguaggio Javascript per realizzare un modello client/server. I files risiedono nel server e vengono diversificati in accordo al loro utilizzo. Alcuni moduli funzionali definiscono il comportamento del server. Node.js è un applicativo cross-platform, ovvero può essere utilizzato in più sistemi operativi. Alcuni esempi di sistemi operativi in cui Node.js può essere eseguito sono: Windows, OSX e Linux. Questa caratteristica permette di eseguire lo stesso codice sorgente Javascript su più sistemi operativi senza sostanziali modifiche. Permette inoltre l'accesso al filesystem del sistema operativo che lo ospita. Node.js è composto da due componenti: una componente lato server e una lato client. La componente server permette di eseguire le attività, definite task, ovvero le operazioni che il server dovrà eseguire. La componente client è un file che il server invia ai client con cui si connette, in cui sono presenti tutte le funzioni utili al collegamento con il server. Queste sue caratteristiche hanno reso Node.js un utile strumento per implementare i protocolli di comunicazione tra le componenti client e server di SIRENE.
Anche Express.js è un progetto della Node.js Foundation e può essere considerato un modulo utilizzabile da Node.js. Esso fornisce una serie di utili funzionalità di comunicazione che sfruttano il protocollo HTTP e permettono di realizzare applicazioni Middleware[7]. Mediante Espress.js è possibile scrivere file che realizzano tutte le funzionalità server che saranno interpretate da Node.js. È dunque possibile definire funzionalità server HTTP a cui i client possono richiedere o inviare informazioni. Socket.io è un altro modulo che può essere aggiunto a Node.js. Esso permette di implementare sistemi server altamente performanti realizzando sistemi di comunicazione real time tra i client e il server. L'avvio della comunicazione può avvenire dal client, ma anche dal server, ovvero in maniera simile ai sistemi Push messagging[8], dove il server invia un messaggio ad uno o più client senza che questi lo abbiano richiesto. Socket.io permette di realizzare la stessa modalità di comunicazione dei sistemi Push messagging. Usando Socket.io è possibile realizzare chat real time, sistemi push messaging, ecc. Inoltre, Socket.io è capace di verificare se un client che cerca di connettersi dispone della tecnologia dei Websoc-

[7] Le applicazioni Middleware sono un insieme di programmi che definiscono una o più interfacce di comunicazione che si collocano tra un insieme di utenti ed un sistema applicativo al fine di proteggere tale sistema da un potenziale uso improprio da parte degli utenti.

[8] Il Push messagging è una tecnica di comunicazione che permette lo scambio di dati, detti messaggi, in tempo reale o quasi reale, tra un server ed uno o più client.

ket[9] oppure necessita di utilizzare alcune tecniche di connessione tra il client e il server che richiedono maggiori oneri prestazionali, come, ad esempio, l'Ajax long pooling, ovvero una tecnica di comunicazione Ajax che mantiene una connessione http attiva durante la quale in qualsiasi momento è possibile ricevere e mandare dati. Questa tecnica, però, è onerosa dal punto di vista prestazionale, sia sul client e sia sul server, poiché il canale di comunicazione viene sovraccaricato di pacchetti di dati, che spesso risultano pacchetti di dati utili al solo mantenimento della connessione perennemente attiva ed a causa del browser che deve sempre controllare la ricezione di questi pacchetti. La tecnica Ajax long pooling era nata come tentativo di colmare la mancanza di socket utilizzabili direttamente nei browser, ed in tale maniera era possibile simulare gli attuali Websocket per ricevere i dati "quasi" in real-time. In realtà la connessione, nell'Ajax long pooling, non è realtime ma quello che si verifica è una continua richiesta di dati in una connessione, tra il client ed il server, mantenuta forzatamente attiva.

3.4.3 Il linguaggio di programmazione Pascal di Lazarus

Il linguaggio Pascal [31] è uno dei più vecchi, ma ancora usati, linguaggi di programmazione. È universalmente considerato il linguaggio di programmazione da utilizzare per la didattica, e per questa sua caratteristica lo si userà come paradigma nella stesura di parte del linguaggio testuale generato usando il linguaggio visuale di SIRENE. Il Pascal è stato inventato da Niklaus Wirth. Wirth, docente di informatica, durante il suo periodo di insegnamento, aveva notato che mancava un linguaggio di programmazione algoritmico dotato di strutture dati avanzate, quindi inventò il Pascal, ovvero un linguaggio di programmazione algoritmica inserendovi il concetto di programmazione strutturata. Il Pascal è un linguaggio fortemente tipizzato, ciò significa che ogni istanza di un tipo non può variare la sua definizione. Con le successive modifiche apportate al linguaggio, sono state introdotte alcune caratteristiche e funzionalità che permettono ad alcuni tipi, sotto opportune condizioni, di essere usati come altri tipi. Il Pascal fu adottato come linguaggio da varie aziende commerciali e gruppi no profit. Lazarus [53] è un ambiente di sviluppo e permette di sviluppare programmi utilizzando il linguaggio Pascal. Lazarus è stato iniziato nel 1999 da Cliff Baeseman, Shane Miller, e Michael A. Hess, e prese il nome di "Lazarus project". Lazarus è un ambiente di sviluppo cross-platform, ovvero è un ambiente di sviluppo utilizzabile in molti sistemi operativi e permette di creare programmi per altrettanti sistemi operativi. Attualmente sono supportati i seguenti sistemi operativi: Windows (tutte le versioni), WinCE, Unix, Beos, BSD, Linux, Solaris, Macintosh, Android, IOS, Amiga,

[9] Nel seguito del capitolo si discuterà di cosa è un websocket.

Atari, OS2, MSDOS. Inoltre, Lazarus può utilizzare la libreria INDY, una libreria scritta in linguaggio Pascal realizzata dal gruppo "The Indy Project". Anche questa libreria è cross-platform.
Il motivo per il quale si è scelto di utilizzare il Pascal di Lazarus, come secondo linguaggio di programmazione per sviluppare SIRENE, è che Lazarus mette a disposizione degli sviluppatori una vasta gamma di funzioni e procedure ed oggetti, utilizzabili per sviluppare programmi funzionanti in tutti i sistemi operativi sopra citati senza dover modificare il codice sorgente. Inoltre, Lazarus è un programma in continuo sviluppo e miglioramento, infatti è aggiornato quasi ogni 6 mesi ed è un applicativo con licenza GPL linking exception per il rilascio del software generato con Lazarus. Per queste motivazioni, Lazarus e il Pascal sono stati scelti per sviluppare una componente fondamentale di un modulo, di intercomunicazione tra processi, del server di SIRENE. Inoltre, l'uso del linguaggio di programmazione di basso livello, come il Pascal, è necessario per la gestione della comunicazione dei messaggi scambiati tra la componete server di SIRENE e i programmi compilati ed eseguiti sul server, in quanto Node.js non gestisce bene lo scambio di dati con i processi eseguiti, mentre il Pascal, può farlo.

3.4.4 HTML5

L'HTML5 è un linguaggio di formattazione per la realizzazione di pagine web. L'HTML5 fornisce un insieme di miglioramenti e utilità dalla versione precedente, l'HTML4. In particolare, in HTML5 sono stati introdotti alcuni elementi utili come i Canvas e gli SVG (Scalable Vector Graphics), elementi grafici che permettono di essere modellati come grafici bitmap, o vettoriali nel caso degli elementi SVG, e permettono di realizzare animazioni. L'HTML5[10] prevede anche la manipolazione di questi elementi direttamente dal linguaggio Javascript. Gli elementi visuali di HTML sono istanze di oggetti HTML. Le istanze vengono inserite nella pagine web tramite una sintassi ben definita, che generalmente è:

$$<\textbf{tag}\ \ \textit{alcuni attributi o eventi} > \textit{altro codice HTML} <\!/\textbf{tag}>$$

dove **tag** deve essere il tipo di elemento. Ogni elemento HTML possiede molti attributi comuni ad altri elementi, mentre alcuni attributi sono specifici di pochi elementi. Uno dei più importanti attributi comuni è *id*, infatti esso permette di identificare univocamente l'elemento HTML all'interno di tutto il documento, o pagina, web. Mediante questo attributo Javascript e il linguaggio di formattazione

[10]Nel seguito si scriverà HTML per riferirsi ad HTML4 e HTML5 indifferentemente, se non diversamente specificato.

CSS3, di cui se ne discuterà successivamente, possono avere accesso ad ogni elemento HTML per manipolarli più facilmente. Un altro attributo importante per quasi tutti gli elementi HTML è l'attributo *class* e permette all'elemento HTML di ricevere un insieme di informazioni utili per il suo aspetto grafico da una classe inserita nella pagine web o da un file scritto in linguaggio CSS3. La sintassi, in generale, per valorizzare gli attributi è la seguente

attributo="*valore da inserire*"

Questa sintassi assegna il contenuto tra virgolette come valore di quell'attributo. Alcuni particolari attributi sono gli eventi. Gli eventi permettono di eseguire un determinato codice Javascript, o una funzione Javascript, quando si verifica un particolare evento nella pagina web visualizzata. Un tipico evento è *onclick* che si verifica ogni volta che si esegue il click, col tasto sinistro del mouse, su un qualsiasi elemento visuale HTML. In questo caso, se un oggetto HTML ha una funzione Javascript, ad esempio la funzione "**cliccato(event)**", associata all'evento *onclick* allora quando un utente cliccherà sopra questo elemento il browser eseguirà la funzione **"cliccato()"** e, il browser, passerà alla funzione **"cliccato()"** l'oggetto *event*, il quale include una serie di informazioni come: il tipo di evento, l'oggetto, chiamato *target* che ha innescato l'evento, le coordinate della pagina web dove l'utente ha cliccato, e molte altre informazioni. Inoltre, gli elementi Canvas e SVG forniscono un insieme di funzioni per visualizzare forme geometriche: linee, cerchi, quadrati, rettangoli e poligoni. Il corretto uso di questi permette di creare strutture grafiche complesse e di notevole interesse. Questi due elementi hanno caratteristiche diverse: l'elemento Canvas ha una dimensione e risoluzione definite e il testo deve essere disegnato nell'area grafica dell'oggetto, ma è un elemento che si presta bene ad essere utilizzato nei giochi ed in tutte quelle occasioni in cui devono essere inseriti ed usati molti elementi grafici; diversamente, gli elementi SVG sono indipendenti dalla risoluzione, in quanto disegnano grafici vettoriali, i suoi elementi grafici vengono realizzati mediante l'inserimento di sotto-elementi HTML di SVG , come *circle*, *rect*, ecc. Tale caratteristica causa una maggiore lentezza nel rendering dell'elemento SVG a causa dell'interpretazione dei suoi sotto-elementi. A causa di questa lentezza, si esclude l'utilizzo degli elementi SVG nei giochi o ambienti nei quali si prevede un largo uso di essi.
L'HTML5 aggiunge anche un'interfaccia di comunicazione internet chiamata WebSocket. Un WebSocket crea un canale di comunicazione full-duplex, ovvero un canale bidirezionale, usando una connessione internet con protocollo TCP. Le comunicazioni avvengono mediante la porta 80 del TCP. Ciò permette di poter realizzare comunicazioni anche in presenza di firewall nella intranet. Le interfacce WebSocket sono implementate ed ottimizzate direttamente nei browser. Inoltre,

ogni browser, pur implementando l'interfaccia WebSocket personalizzata, fornisce comuni funzioni e metodologie per la comunicazione in internet. Grazie alle caratteristiche precedentemente citate, l'HTML5 è stato scelto come linguaggio per la realizzazione grafica di SIRENE e per la realizzazione di comunicazione in internet tramite l'uso di Socket.io lato client.

3.4.5 CSS3

Il linguaggio CSS3, ovvero Cascading Style Sheets versione 3[11], è un linguaggio di formattazione che permette di realizzare regole di stili grafici da associare ad uno o piu elementi HTML, ovvero in cascata. Esso fornisce un insieme di proprietà grafiche da dover valorizzare. Valorizzare una proprietà significa assegnare un valore coerente al tipo di proprietà, ad esempio nel CSS è frequente assegnare un valore alla proprietà *background-color* ed il suo valore deve essere un colore. Attraverso la valorizzazione delle proprietà del CSS è possibile realizzare aspetti grafici, anche complessi, per gli elementi visuali HTML. Il CSS nasce dall'esigenza dello sviluppatore di impostare una precisa formattazione del contenuto nelle pagine web, indipendentemente dalle preferenze dei browser. Infatti, prima dell'avvento del CSS, non era possibile impaginare il contenuto delle pagine web. Quando questa impossibilità divenne una forte limitazione nella realizzazione di una impaginazione corretta e personalizzata, gli sviluppatori di browser cercarono di sviluppare appropriate funzioni, proprie dei browser, che potessero rispondere a questa esigenza. Ogni sviluppatore di browser implementava una sua propria modalità di funzionamento per la formattazione grafica. Da ciò, si realizzò una gara tra sviluppatori di browser, denominata "guerra tra browser", ovvero una sorta di gara tra sviluppatori nella quale si confrontavano le potenzialità dei browser. Il browser vincitore sarebbe stato il browser che forniva più funzionalità e che forniva maggiori capacità di visualizzazione. Da qui nacque l'esigenza di realizzare un unico linguaggio di formattazione grafica standardizzato tra tutti i browser e dunque nacque il CSS.
Il linguaggio CSS permette di manipolare velocemente un elemento specifico, insiemi di oggetti che afferiscono alla stessa classe o allo stesso tipo. Per esempio, è possibile manipolare attributi grafici di uno specifico elemento visuale HTML inserendo la definizione di una regola di stile grafico, nella pagina web o in un file CSS caricato da una pagina web, attraverso l'uso della seguente sintassi e semantica:

 # "identificatore elemento"
{ *...proprietà: valore; ...* **}**

[11]Esistono altre versioni di CSS: la 1 e la 2. Nel seguito si scriverà CSS indifferentemente.

Per manipolare la grafica di tutti gli elementi HTML che associano al proprio attributo *class* la classe *NomeClasse*, si utilizza la seguente modalità:

.“NomeClasse”
{ *...proprietà CSS: valore; ...* **}**

Diversamente, per manipolare graficamente tutti gli elementi visuali HTML dello stesso tipo di *tag*, si scrive la seguente:

“tag”
{ *...proprietà CSS: valore; ...* **}**

Il linguaggio CSS permette anche di poter impostare più di un formato di stile grafico per elemento, infatti è possibile inserire come attributo *class* due o più classi separate da spazio, in questo caso l'ultima classe sovrascrive gli attributi valorizzati dalla prima. Oppure è possibile inizialmente settare un attributo per tutti i tipi di elementi, e successivamente settare altri attributi per gli elementi. Per poter impostare uno o più attributi per tutti gli elementi si utilizza la seguente forma:

***{** *...proprietà CSS: valore; ...* **}**

dove “*” indica tutti gli elementi di una pagina web.

Capitolo 4

Sviluppo del linguaggio iconico di SIRENE

4.1 Linguaggi visuali in SIRENE

SIRENE è stato realizzato per sviluppare e potenziare il pensiero computazionale, ridurre le difficoltà degli studenti nell'individuare soluzioni a problemi computazionali e, contemporaneamente, ad apprendere più facilmente la sintassi e la semantica di un linguaggio di programmazione, quale può essere il linguaggio C. Si è scelto di progettare e di realizzare due linguaggi visuali per risolvere entrambi i problemi. Il modello Control flow è stato adottato per la realizzazione di un linguaggio iconico adatto a rappresentare il comportamento concettuale ed astratto di un algoritmo. L'altro linguaggio visuale è stato realizzato per rappresentare le espressioni matematiche, condizionali, le assegnazioni e la definizione di funzioni da chiamare. Nei seguenti paragrafi saranno discussi entrambi i linguaggi visuali e la loro implementazione.

4.2 Linguaggio iconico e concettuale di SIRENE

Il linguaggio iconico di SIRENE (per brevità nel seguito si indicherà linguaggio), è un linguaggio visuale che permette di comprendere il funzionamento e il comportamento di un algoritmo. Questo linguaggio è una combinazione tra le mappe concettuali e i diagrammi Control Flow. Gli elementi iconici hanno una veste grafica simile agli elementi tradizionali del Control Flow, con qualche differenza. Tra i vari modelli, presenti in letteratura utili per realizzare gli elementi visuali, si è scelto il Control Flow in quanto affine alle caratteristiche del linguaggio in oggetto e descritte nel paragrafo 2.2:

- i diagrammi Control flow schematizzano correttamente i passi degli algoritmi, fornendo una chiara visione del comportamento algoritmico;
- è un modello che permette facilmente di modificare gli schemi diagrammatici;
- i diagrammi Control flow sono utilizzati in svariati ambiti, anche industriali, per rappresentare i processi logici;
- è un modello in cui è possibile manipolare facilmente e velocemente gli elementi visuali fornendo un feedback immediato ed intuitivo.

Generalmente, negli elementi visuali dei diagrammi Contro Flow utilizzati per rappresentare gli algoritmi è uso comune inserire porzioni di codice di programmazione al fine di identificare l'azione computazionale (istruzione o assegnazione del calcolo di una espressione) associata all'icona. Il linguaggio di SIRENE è leggermente differente: non viene inserito una parte del codice sorgente dell'algoritmo all'interno delle icone, ma una frase che identifica il contenuto e la spiegazione dell'azione associata a quell'icona. Questa modalità di visualizzazione è tipica delle mappe concettuali e permette all'utente di associare un concetto, anche complesso, ad un elemento grafico. Pertanto, le frasi all'interno di un'icona giustificano l'uso e l'esistenza dell'icona stessa all'interno dell'algoritmo grafico. Questa rappresentazione grafica permette di sfruttare le potenzialità grafiche sia dei diagrammi di flusso sia delle mappe concettuali. Gli schemi grafici sono schemi orientati nel flusso dell'algoritmo ed ogni elemento grafico sintetizza l'azione che svolge e, come nelle mappe concettuali, ogni elemento grafico viene associato mentalmente ad una specifica azione. Inoltre, ogni elemento visuale è associato ad un preciso costrutto del linguaggio C.
Un altro elemento importante di tale linguaggio è il flusso. Nei linguaggi di programmazione testuale esiste il concetto di flusso di un algoritmo, ovvero un algoritmo inizia da una posizione ben specificata, detta radice della funzione, all'inizio della sezione dichiarativa del codice testuale, e continua seguendo una sequenza ben specificata di azioni da eseguire. Analogamente, in SIRENE il flusso inizia sempre da un'icona *Start* e continua seguendo una sequenza direzionata di archi che, nel loro insieme, connettono i vari elementi grafici, formando un percorso orientato e non ambiguo terminando nell'ultimo elemento, non necessariamente unico, denominato pozzo in cui non sono presenti archi uscenti.

4.2.1 Sviluppo del flusso iconico di SIRENE

Lo sviluppo di un linguaggio di programmazione iconico necessita di un flusso logico di elementi visuali posti in una precisa e ben definita sequenza. Questa sequenza deve essere non ambigua. Per emulare questa caratteristica si è studiato

come gli ambienti di sviluppo di programmazione testuale forniscono questo concetto. Tipicamente, in ogni ambiente di sviluppo la scrittura del codice di una funzione avviene mediante l'inserimento di una serie di istruzioni sequenziali, ovvero cominciando immediatamente dopo la definizione della funzione scendendo alla fine di questa. Si è, dunque, scelto di emulare la stessa funzionalità di concetto di flusso, realizzando l'icona *Start* quale punto di inizio non ambiguo del flusso di qualsiasi funzione. Pertanto, ogni funzione avrà una sua icona *Start* ed a questa si collegheranno tutti gli altri elementi visuali. Quest'approccio è conforme allo standard utilizzato nei diagrammi di flusso Contro Flow. I costrutti visuali di SIRENE vengono, quindi, collegati tra di loro mediante gli archi i quali costituiscono gli elementi di collegamento del flusso. Questa rappresentazione garantisce la continuità del flusso ma anche un controllo sugli elementi del flusso in diversi modi:

- Ogni costrutto nel flusso ha visibilità con quali altri costrutti può collegarsi. Questa caratteristica del linguaggio concettuale non permette all'utente di collegare costrutti già inseriti nel flusso dell'algoritmo iconico. Inoltre, da ogni elemento visuale possono esistere un solo arco incidente su di esso e un solo arco uscente per orientare il flusso. Alcuni costrutti iconici, quali la condizione e le icone che corrispondono ai cicli, permettono una ramificazione del flusso e possono, quindi, avere uno o due archi uscenti a seconda del tipo di costrutto iconico.

- La connessione degli elementi visuali mediante archi aumenta la leggibilità concettuale e migliora la comprensione dell'algoritmo e la comprensione e visualizzazione delle modifiche che dovranno essere apportate all'algoritmo.

Il flusso del linguaggio iconico di SIRENE permette di annidare altri costrutti tra di loro. In questo caso il colore degli archi del flusso cambieranno di colore in accordo all'indice di annidamento. Per la visualizzazione dei livelli di annidamento sono stati definiti 10 colori:

1. White
2. Red
3. Green
4. Blue
5. Silver
6. Fuchsia

7. Yellow
8. Lime
9. Aqua
10. Purple

Se un utente dovesse realizzare più di 10 livelli in uno stesso annidamento, il colore degli archi ricomincia dal primo colore definito, ovvero il *white*, e così via.

4.2.2 Costrutti di SIRENE

I costrutti di SIRENE corrispondono ai costrutti fondamentali dei linguaggi di programmazione. Alcuni costrutti presenti negli attuali linguaggi testuali non sono stati inseriti nel linguaggio a causa della poca leggibilità del codice iconico che ne deriverebbe ed a causa dell'esistenza di altri costrutti iconici che possono sostituirli, come ad esempio il costrutto condizionale *Switch* che può essere sostituito da una sequenza di costrutti condizionali *If..else* in cascata ed annidati. Inoltre, in accordo al teorema di Bohm e Jacopini [6] non è fondamentale usare il costrutto *Goto* per poter produrre un buon codice sorgente nella programmazione strutturata, in quanto può essere fuorviante e, quindi, anche questo costrutto non è stato inserito. Per la stessa ragione, non sono stati inseriti, nel linguaggio iconico di SIRENE, anche i costrutti tipici come: l'*Etichetta*, il *Break*, il *Continue*. Gli elementi del linguaggio visuale e concettuale di SIRENE sono 8, non considerando gli archi del flusso, 6 dei quali identificano i costrutti più usuali presenti nei linguaggi di programmazione:

- costrutto condizionale *If..else*;
- costrutto ciclico *For*;
- costrutto ciclico *While*;
- costrutto ciclico *Do..while*;
- costrutto di uscita da una funzione *Exit*;
- costrutto del risultato fornito da una funzione *Return*;

Gli ulteriori 2 elementi visuali del linguaggio di SIRENE corrispondono a:

- Assegnazione di un valore, o di una espressione, ad una variabile. Questo costrutto viene chiamato "Assegnazione".
- Chiamata di una istruzione. Questo elemento rappresenta una "Istruzione".

4.2.3 Caratteristiche degli elementi visuali del linguaggio iconico di SIRENE

Ogni elemento iconico in SIRENE ha alcune proprietà visuali che lo differenziano senza ambiguità da tutti gli altri elementi. Queste caratteristiche sono:

- Colore del contorno.
- Forma dell'icona.
- Testo contenuto nell'icona.
- Creazione di una immagine custom per le istruzioni.
- Associazione di un codice testuale, che nello specifico fa riferimento al linguaggio di programmazione C.

4.2.4 Colore del contorno

Ogni icona del linguaggio concettuale di SIRENE ha, generalmente, un colore differente di contorno. Alcune icone possono avere stesso colore ma stile del contorno e/o forma dell'icona differenti. Questo concetto è fondamentale in quanto è il primo elemento di distinzione tra le icone e fornisce un primo impatto visuale per rappresentare l'elemento iconico e la memorizzazione del tipo di costrutto.
L'icona *Start* è l'unico elemento iconico che possiede un contorno rosso. Poiché dall'icona *Start* inizia il flusso logico e concettuale di ogni algoritmo, essa diviene l'elemento fondamentale del flusso iconico e richiede un colore preciso di differenziazione da tutti gli altri elementi visuali affinché quest'icona risulti immediatamente individuabile.
Un altro elemento visuale che possiede un colore differente dagli altri è il costrutto iconico condizionale. Questo usa un colore giallo, colore al quale viene generalmente associato il significato di "attenzione", ed a causa di questo significato viene usato in quest'icona.
I costrutti visuali ai quali viene associato il concetto di "ciclo" hanno colori differenti, ognuno per indicare un tipo di ciclo differente: il costrutto *For* iconico, equivalente al ciclo "for" nei linguaggi testuali, usa il colore verde; il costrutto iterativo condizionale *While* utilizza il colore blu; mentre l'elemento *Do..while* utilizza il colore fucsia.
Il costrutto visuale che identifica il concetto di "assegnazione" usa un colore arancio.
L'elemento visuale che è associato alle "istruzioni" assume un colore del bordo viola. Vedremo che questo costrutto ha potenzialità visuali differenti rispetto alle altre icone.

Il costrutto iconico *Return* utilizza un colore verde
Un caso a parte è il colore nero. Questo viene usato nell'elemento visuale del linguaggio iconico al quale è associato il concetto di uscita da una funzione, ovvero l'*Exit*.

4.2.5 Forma degli elementi iconici

Oltre al colore, le icone del linguaggio concettuale di SIRENE si differenziano per la forma. Alcune icone hanno forma uguale ma colori differenti nei bordi e testo differente all'interno dell'icona. L'icona *Start* ha una piccola forma rettangolare con angoli smussati, ed è la più piccola icona del linguaggio concettuale, proprio per indicare il punto di inizio del flusso logico dell'algoritmo iconico.

Figura 4.1: Icona *Start*.

In accordo allo standard utilizzato nei diagrammi di flusso, l'icona condizionale ha una forma romboidale ad indicare una condizione e ramificazione nel flusso dell'algoritmo.

Figura 4.2: Icona condizionale *If..else*.

I costrutti di *Assegnazione* e di *Istruzione* hanno entrambi una forma rettangolare, per rappresentare concettualmente l'assegnazione di una espressione ad una variabile e l'utilizzo di una istruzione.

I costrutti iconici che rappresentano i "cicli" hanno una forma quadrata con angoli smussati. Questo indica che è presente un blocco di azioni che viene ripetuto un numero definito di volte. Poiché questo blocco di azioni può avere una lunghezza variabile, si è scelto di utilizzare la forma rettangolare per questi tre costrutti.

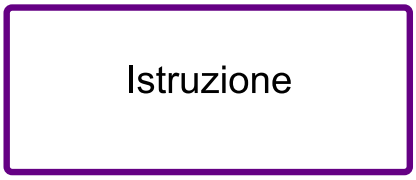


Figura 4.3: Icona *Istruzione*.

Figura 4.4: Icona *Assegnazione*.

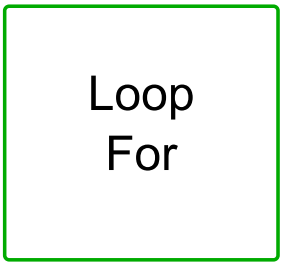


Figura 4.5: Icona *For*.

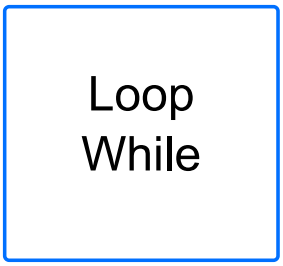


Figura 4.6: Icona *While*.

Un'altra forma rettangolare, più grande di quella dell'icona *Start* ma più piccola di quelle dell'*Assegnazione* e dell'*Istruzione*, è quella dell'icona *Return*. Generalmente, il costrutto *Return* fornisce un valore già calcolato o un valore ottenuto da una espressione, ed è per questo motivo che viene rappresentato graficamente da una forma rettangolare piccola rispetto agli altri costrutti rettangolari.

Diversamente, la forma circolare è stata utilizzata per il solo costrutto di uscita

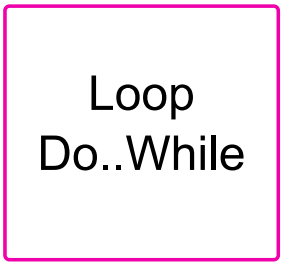


Figura 4.7: Icona *Do..while*.

Figura 4.8: Icona *Return*.

da una funzione, ovvero il costrutto iconico che rappresenta l'*Exit*. L'elemento *Exit* possiede un bordo tratteggiato per indicare una interruzione o un'uscita da una funzione.

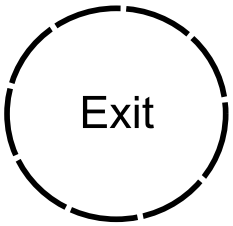


Figura 4.9: Icona *Exit*.

4.2.6 Testo nelle icone

Ogni costrutto visuale in SIRENE permette di avere un testo che esplicita e motiva l'uso e l'esistenza di quell'icona all'interno dell'algoritmo visuale, una sorta di commento proprio dell'icona. Quest'aspetto è molto importante in quanto si associa un significato preciso ad un elemento visuale, esattamente come avviene nelle mappe concettuali. Infatti, questa tipologia di rappresentazione sfrutta le potenzialità dei due tipi di intelligenze logico-matematica e visuale-spaziale della teoria delle intelligenze multiple di Gardner [26, 17] e permette di memorizzare e comprendere più facilmente il comportamento dell'algoritmo iconico che si intende sviluppare.

4.2.7 Immagini custom per le istruzioni

Le immagini sono potenti strumenti per fornire un'informazione o un significato precisi. L'unico costrutto che può visualizzare un'immagine, al posto della propria icona, è il costrutto *Istruzione*. L'immagine può essere specificata dall'utente durante la definizione di una funzione usando SIRENE. Un'immagine definita per una funzione può essere un'informazione immediata dell'azione associata all'icona quando si visualizza ed ispeziona l'algoritmo iconico, ad esempio SIRENE visualizza sempre un cestino nell'area di programmazione iconica e l'azione esplicitata da questo cestino è facilmente comprensibile, ovvero la rimozione di un'icona dall'area del codice visuale.

4.2.8 Associazione di un codice testuale

Il linguaggio concettuale di SIRENE possiede una caratteristica importante: ogni elemento visuale è univocamente associato al testo di un costrutto di un linguaggio di programmazione, che nel caso specifico è il "C", ed è associato, in maniera biunivoca, ad una porzione di codice testuale visualizzata durante lo sviluppo dell'algoritmo iconico. Ciò permette di associare il significato dell'azione di un'icona del linguaggio di SIRENE ad un codice testuale.

4.3 Linguaggio iconico di espressione

Il linguaggio iconico e di espressione di SIRENE è stato sviluppato per aiutare l'apprendimento e la comprensione della sintassi e della semantica del linguaggio di programmazione C. Per agevolare l'insegnamento e l'apprendimento è stata implementata la possibilità di realizzare espressioni in codice testuale C utilizzando oggetti visuali che corrispondono agli elementi e ad istruzioni del linguaggio C. Per fare questo, è stata sviluppata una specifica sezione per realizzare le espressioni. Gli elementi di tale sezione sono i nomi delle variabili, delle costanti e dei parametri di una funzione, delle funzioni standard del linguaggio C e delle funzioni realizzate usando il linguaggio iconico, gli operatori aritmetici, logici, relazionali, e bitwise, oltre all'inserimento di dati primitivi come testo o dati numerici.
Le espressioni sono realizzate mediante l'accostamento di questi elementi visuali. L'accostamento, o concatenazione, di questi oggetti crea espressioni semplici o complesse e permette l'annidamento di ulteriori espressioni tramite l'uso delle parentesi.
Le espressioni iconiche sono usate per definire le espressioni per l'assegnazione di valori alle variabili o ai parametri e per definire espressioni condizionali. Un caso particolare consiste nella chiamata delle funzioni. In questo caso, si può realizzare un'espressione usando soltanto una istruzione. Nel caso in cui una funzione

dovesse richiedere uno o più parametri per il corretto funzionamento allora ogni parametro (che richiede un passaggio per valore e non per riferimento) può essere gestito come un'espressione comunemente realizzata con SIRENE. Tecnicamente, ogni parametro è il valore fornito da una espressione, una variabile, una costante o un altro parametro, ed è, quindi, possibile inserire espressioni come parametri per le funzioni.
Le espressioni vengono costruite in accordo alla sintassi e semantica del linguaggio C. Questa rappresentazione del linguaggio C aiuta lo studente a non commettere errori comuni, quali:

- inserire nomi errati di funzioni, di variabili, di costanti o parametri;
- inserire valori errati durante l'assegnazione di valori di tipo: (unsigned) char, (unsigned) short, (unsigned) int, (unsigned) long, float e double;
- inserire un numero errato di apertura e chiusura parentesi;
- sbagliare l'inserimento degli operatori relazionali nelle espressioni condizionali;
- inserire un numero errato di parametri richiesti da una funzione;
- inserire una scorretta sintassi nelle espressioni. Questa caratteristica è opzionale a discrezione del docente.[1]

Ogni espressione è associata ai costrutti del linguaggio iconico creando una relazione biunivoca tra il codice testuale costruito tramite il linguaggio iconico di espressione e i costrutti del linguaggio.
Inoltre, la realizzazione di espressioni attraverso l'uso del linguaggio di SIRENE stimola fortemente lo studente nel creare espressioni corrente in quanto il discente deve concentrarsi nell'usare correttamente la semantica e la sintassi del linguaggio iconico.
L'uso combinato dei due linguaggi iconici permette di realizzare un vero codice sorgente testuale in linguaggio C di un programma e stimola lo studente a concentrarsi a sviluppare concettualmente un programma e a costruire espressioni ed assegnazioni corrette.
Il codice sorgente associato all'espressione è visibile in tempo reale in un Text Editor, in cui è inibita la possibilità di modificare il codice se non modificando l'accostamento degli oggetti visuali.

[1] Il docente può decidere se SIRENE deve aiutare il discente nel comprendere se un'espressione realizzata è sintatticamente corretta o meno.

Ogni elemento visuale ha caratteristiche proprie, con numero di parametri statico o dinamico a seconda della definizione delle funzioni. Inoltre, alcune funzioni hanno un comportamento dinamico, ad esempio le funzioni omologhe alle funzioni del linguaggio *printf* e *scanf* cambieranno dinamicamente il numero di parametri richiesti in funzione del testo inserito. Inoltre, se alcuni parametri richiesti sono parametri che devono passati per riferimento in questo caso il codice sorgente associato alla funzione inserirà automaticamente il simbolo "**&**" prima della variabile, costante o parametro.
In aggiunta, ogni qual volta si inserisce una funzione in una espressione, SIRENE modifica il codice sorgente del file generato includendo una dipendenza al file che definisce la funzione inserita.

4.4 Parser del linguaggio iconico

Per la corretta costruzione del codice sorgente in linguaggio C sono stati sviluppati due parser: uno per il linguaggio concettuale ed uno per il linguaggio visuale delle espressioni. In generale, nel seguito, si indicherà parser iconico, o semplicemente parser, il parser che analizza il linguaggio di SIRENE.
Il parser del linguaggio necessita di iniziare l'analisi di ogni funzione partendo dall'icona *Start*. Seguendo il flusso visuale, il parser costruisce il linguaggio C in accordo al tipo di icona che sta analizzando ed alle informazioni in essa memorizzate. Ad esempio, l'analisi dell'icona *For*, creerà il codice sorgente

for **(** *espressione/i di assegnazione; di condizione; di incremento* **)**
{ *codice sorgente annidato* **}**

in cui le espressioni di assegnazione, di condizione e di incremento sono il codice sorgente delle espressioni realizzate tramite l'uso del linguaggio visuale di espressione di SIRENE e memorizzate all'interno dell'icona stessa, mentre la sezione altro codice consiste del codice sorgente testuale associato alle ulteriori icone che sono state collegate all'icona *For* tramite annidamento.
Un'altra icona adibita al concetto di ciclo è l'icona *While* per la quale il parser iconico fornirà il codice sorgente

while **(** *espressione condizionale* **)**
{ *codice sorgente annidato* **}**

dove tra le parentesi tonde sarà presente un'espressione condizionale generata dal parser visuale di espressione e tra le parentesi graffe sarà inserito un ulteriore

codice sorgente annidato ed associato ad altre icone annidate visualmente all'icona *While*.
Diversamente, se il parser iconico dovesse analizzare l'ultima icona che rappresenta il costrutto di ciclo in SIRENE, ovvero l'icona *Do..while*, allora il codice sorgente generato sarà

do **(** *codice sorgente annidato* **)**
while **{** *espressione condizionale* **}**

dove tra le parentesi tonde sarà presente un'espressione condizionale generata dal parser visuale di espressione e tra le parentesi graffe sarà inserito un ulteriore codice sorgente annidato ed associato ad altre icone annidate visualmente all'icona *While*.
L'attività del parser iconico sull'icona *If..else* fornirà un codice sorgente simile a

if **(** *espressione condizionale* **)**
{ *altro codice sorgente* **}**
else
{ *altro codice sorgente* **}**

in cui la sezione "*espressione condizionale*" è costituita dal codice sorgente testuale associato agli elementi visuali del linguaggio visuale di espressione di SIRENE e memorizzato all'interno dell'icona *If..else*, mentre le sezioni "*altro codice sorgente*" costituiscono il codice sorgente testuale associato alle icone annidate nelle ramificazioni del costrutto iconico *If..else*. Il parser, analizzando l'icona *If..else*, verificherà l'esistenza delle ramificazioni del flusso uscenti da quest'icona: se non esistesse una ramificazione o esistesse solo la ramificazione "*Then*" allora il codice sorgente associato all'icona è costituito da una sola sezione di parentesi graffe aperte e chiuse, {} con all'interno ulteriore codice sorgente generato da altre icone annidate; nel caso in cui esistesse la ramificazione dell'"*Else*" allora il codice sorgente associato all'icona *If..else* contiene entrambe le ramificazioni con ulteriore codice sorgente generato da altre icone annidate.
Per le altre icone, il parser del linguaggio concettuale fornisce il codice sorgente generato dal parser del linguaggio visuale di espressione e memorizzato nell'icona stessa.
Una caratteristica intrinseca ed utile del parser iconico è che il codice testuale segue il flusso dell'algoritmo iconico: questo significa che è possibile avere all'interno di un algoritmo iconico parti di codice visuale non agganciate al flusso e quindi non inserite nel codice testuale. Infatti, nel caso in cui si volesse confrontare il funzionamento di parti differenti di algoritmi iconici è necessario solo collegare o

scollegare attraverso gli archi le icone. Ad esempio, per verificare l'evoluzione di un algoritmo di ordinamento in cui è possibile realizzare porzioni di codice iconico in maniera diversa, è possibile collegare prima una sezione di codice per visualizzare iconograficamente il suo effetto, e, successivamente, scollegare questa porzione dal flusso dell'algoritmo per collegarne, al flusso, una seconda, ma diversa, porzione iconica e verificarne il diverso funzionamento.
Oltre al parser iconico del linguaggio è stato sviluppato un parser iconico per il linguaggio di espressione di SIRENE. Il parser del linguaggio visuale di espressione analizza elementi visuali associati alle istruzioni, variabili, costanti, parametri ed operatori e fornisce un codice sorgente testuale in accordo al tipo di elemento visuale che si sta analizzando. Nel caso in cui il parser dovesse analizzare un operatore, esso fornirà un codice sorgente coincidente all'operatore analizzato. Diversamente se il parser dovesse analizzare una variabile (in caso di una costante o di un parametro il funzionamento è coincidente) esso verifica se tale variabile è inserita all'interno di una funzione: se non dovesse essere all'interno di una funzione allora il parser fornisce il nome della variabile; se invece la variabile dovesse essere all'interno di una funzione allora il parser verifica se la posizione parametrica della funzione, in cui la variabile è inserita, richiede un passaggio per valore o per riferimento: nel caso in cui il passaggio sia di valore, allora il parser fornisce un codice sorgente uguale al nome della variabile, altrimenti il parser fornisce un codice sorgente uguale al nome della variabile preceduto, in questo caso, dall'operatore "&".
Nel caso in cui il parser stesse analizzando una funzione, od istruzione, esso verifica quanti parametri questa funzione richiede ed inserisce il numero corretto (e richiesto dalla definizione della funzione) di spazi e virgole all'interno delle parentesi tonde della funzione. Questa caratteristica è molto importante in quanto aiuta lo studente a fornire il corretto numero di variabili da passare alle funzioni. Inoltre, il parser verifica in quale libreria la funzione è stata sviluppata, ed inserisce il nome della libreria, automaticamente, tra le dipendenze del file in cui si sta sviluppando l'algoritmo.
La formalizzazione del linguaggio di programmazione iconico proposto è stato affrontato attraverso la definizione della sua grammatica e dei relativi simboli. I simboli sono rappresentati dalle icone elementari e la grammatica contex-free è stata definita come una Grammar of Visual Language (grammatica del linguaggio visuale):

$$GVL(N,\ T,\ S,\ P)$$

dove

- N = l'insieme delle icone non terminali;

- $T = T_1 \bigcup T_{WS}$;
- S = Simbolo dello Start, o punto iniziale;
- P = insieme delle regole di produzione;
- T_1 = I(1): I=(I, (WS,t), I_p(WS,t), Type(t), Relazione(WS,t)), UI;
- T_{WS} = insieme delle operazioni sulla WS da modificare;
- I=(I,WS,t), I_p(WS,t), Type(I), RelAction(WS,t));
- I_p = parte fisica di I;
- Type(I) = l'insieme di Strutture dati di I;
- Relazione = la relazione tra WS e I;
- UI= Spazio Universale dell'icona;

4.5 Visualizzatore del codice sorgente

Il codice sorgente testuale generato dal parser iconico è inserito e mostrato in un visualizzatore opportunamente realizzato nella sezione destra ad ogni area visuale del codice iconico ed anche in una sezione generale dove viene visualizzato tutto il codice sorgente di uno specifico File. Questo visualizzatore permette di vedere il codice sorgente ggiornato in modo sincrono da SIRENE durante lo sviluppo del programma visuale, ciò permette un'associazione visiva tra l'azione che un'icona svolge e il suo significato testuale. Tali visualizzatori permettono di rendere visibile solo il codice sorgente e non permettono di modificarlo. Il visualizzatore si occupa anche di gestire e mostrare le variabili e le costanti. La visualizzazione delle variabili e delle costanti avviene in una tipologia Pascal-like, ovvero la dichiarazione (e il relativo codice sorgente) di ogni variabile e costante è inserita nel codice sorgente dell'algoritmo all'inizio di ogni funzione. Sebbene il linguaggio C permetta di dichiarare una variabile e costante in qualsiasi locazione di una funzione, è stata scelta questa metodologia in quanto il raggruppamento visuale di tutte le costanti e variabili all'inizio della funzione permette all'utente di avere una corretta e precisa consapevolezza delle costanti e variabili usate nell'algoritmo.

4.6 Caratteristica cooperativa di SIRENE

SIRENE è un ambiente condiviso per la programmazione. SIRENE è stato sviluppato per colmare la lacuna che rende fortemente limitati ogni altro ambiente di sviluppo: SIRENE è l'unico ambiente di sviluppo iconico cooperativo in tempo reale. La cooperazione permette agli studenti di poter interagire tra di loro in tempo reale, permettendo a tutti gli utenti di sviluppare qualunque algoritmo insieme e contemporaneamente. La natura cooperativa di SIRENE permette anche di vedere ciò che gli altri utenti stanno realizzando. Attualmente, nessun ambiente di sviluppo permette di modificare in maniera cooperativa, e in tempo reale, un programma. SIRENE rompe questo schema: ogni utente può interagire, osservare e manipolare gli algoritmi che un altro utente (o gruppo di utenti) sta realizzando. La manipolazione di ogni algoritmo si ripercuote su ogni utente che sta assistendo alla lezione. Inoltre, la potenzialità di questa caratteristica permette di realizzare configurazioni di metodologie pedagogiche sempre più complesse, portando il docente a creare metodologie innovative e sperimentali in cui l'unico limite è la creatività del docente stesso. Nei prossimi paragrafi si discuteranno alcune caratteristiche aggiuntive che la cooperazione di SIRENE permette di realizzare.

4.7 Interazione con i programmi compilati con SIRENE

SIRENE è un framework sviluppato per l'apprendimento dell'arte della programmazione. Il framework sarebbe incompleto se non mostrasse il risultato materiale dello sviluppo iconico, ovvero i programmi eseguibili. SIRENE, permette di compilare il codice sorgente e permette agli utenti di interagire direttamente con i programmi attraverso il web. Tutti gli utenti possono visualizzare contemporaneamente gli output dei programmi e possono inviare gli input richiesti dai programmi tramite un'apposita interfaccia grafica sviluppata in modalità terminale. Questa interfaccia è composta da due sezioni: la prima che permette di verificare lo stato di compilazione e successivamente, a compilazione correttamente avvenuta, permette di mostrare tutti gli output del programma; la seconda sezione permette di inviare gli input richiesti da un programma usando il tasto invio, come in una console per un sistema operativo, o usando un bottone HTML "invia". SIRENE gestisce, in tempo reale, l'invio degli input richiesti dal programma da parte degli utenti e contemporaneamente la visualizzazione degli output inviati dal programma a tutti gli utenti.

4.8 Caratteristiche uniche di SIRENE

SIRENE ha molte caratteristiche utili. Una delle principali caratteristiche di SIRENE è la possibilità di espletare lezioni in internet in luoghi differenti ed in tempo reale, infatti l'uso del browser permette di assistere alle lezioni senza essere fisicamente nello stesso luogo dove avviene l'insegnamento. Questa è una caratteristica unica di SIRENE e non implementata in altri framework. Inoltre, l'uso del browser permette di assistere alle lezioni utilizzando qualsiasi computer, telefonino, palmare o tablet. Infatti, per ogni sistema operativo esiste un web browser che permette di navigare in internet e, quindi, permette l'uso di SIRENE.
Un'altra caratteristica unica di SIRENE, e che nessun altro framework possiede, è che ogni modifica degli algoritmi mediante l'uso del linguaggio iconico, sia quello concettuale e sia quello di espressione, è immediatamente visualizzata da ogni altro utente. Ogni modifica della posizione di un'icona, o del suo contenuto o del suo collegamento o scollegamento dal flusso dell'algoritmo è visualizzata istantaneamente e contemporaneamente da tutti gli utenti. Inoltre, ogni aggiunzione e cancellazione e modifica ai nomi del programma, dei files, delle funzioni, parametri, variabili e costanti è anch'essa istantaneamente e contemporaneamente visualizzata da tutti gli utenti. Questa caratteristica avviene mediante l'invio di messaggi sincronizzati tra tutti gli utenti a cura del server. Questa metodologia permette una conoscenza coerente dello sviluppo ed una esatta copia dei programmi a livello sia visuale per gli utenti e sia informativo per SIRENE assicurando che non vi siano differenze di informazioni tra tutti gli utenti. Questa caratteristica, unica nel suo genere, permette a tutti gli utenti di cooperare e collaborare alla risoluzione di un algoritmo. Infatti, ogni utente può interagire con il sistema e, quindi, con tutti gli altri utenti. Questo permette di ricreare diverse metodologie pedagogiche e di poterne sperimentare ulteriori, per la didattica della programmazione. Inoltre, SIRENE permette ad un numero elevatissimo di utenti di accedere alle lezioni in quanto, grazie all'uso di Ajax, i dati di aggiornamento della visualizzazione del sistema sono confinati a pochi dati, infatti generalmente i dati, ovvero i messaggi inviati in internet dal server di SIRENE verso ogni client, e viceversa, non superano generalmente la dimensione di 200 byte. Questi messaggi contengono tutte le informazioni essenziali affinché ogni client crei, autonomamente, la stessa azione nella propria area di visualizzazione.
Un'altra caratteristica unica è che SIRENE permette agli studenti di focalizzarsi sia sull'aspetto concettuale dello sviluppo di un algoritmo, al fine di comprenderne l'esatto comportamento, sia sull'aspetto dell'apprendimento della corretta sintassi e semantica di un linguaggio di programmazione, che nello specifico è il C.
Un'altra caratteristica unica di SIRENE è che permette di compilare il codice sorgente generato in linguaggio C, di eseguire il programma generato dalla fase di compilazione e di permettere a tutti gli utenti di interagire con questo programma,

infatti SIRENE gestisce automaticamente l'invio di tutti gli output del programma verso i client e gestisce l'invio verso il programma di tutti i dati dei client necessari come input.
Nell'appendice C si discuterà di come utilizzare SIRENE, mentre nell'appendice D viene discussa l'installazione di SIRENE nel sistema operativo Ubuntu versione 18.10.
Nell'appendice A si discuteranno i dettagli implementativi della componente Server di Sirene. Infine, nell'appendice B si esamineranno i dettagli implementativi del linguaggio iconico insieme alla componente Client di SIRENE.

Capitolo 5

Sperimentazione e risultati dell'uso di SIRENE nel campo dell'istruzione

5.1 Riferimento europeo

Come accennato nel paragrafo 1.1, i corsi di linguaggi di programmazione sono stati inseriti nelle scuole di ogni ordine e grado, incluse le università. In Europa, ogni nazione ha sviluppato un proprio piano di istruzione. Pertanto, questi piani di istruzione possono essere differenti tra le varie nazioni, anche in maniera sostanziale e profonda. All'inizio, questa differenziazione non permetteva di comprendere facilmente, in prima analisi, le differenze e le somiglianze dei livelli di istruzione tra le varie nazioni. A causa di ciò, il 23 aprile del 2008 il Parlamento Europeo e il Consiglio dell'Unione Europea ratificarono la "Raccomandazione 2008/C 111/01/CE del Parlamento Europeo e del Consiglio del 23 aprile 2008 sulla costituzione del Quadro europeo delle qualifiche per l'apprendimento permanente" che definisce il quadro europeo delle qualifiche, anche detto European Qualifications Framework (EQF). Questo realizza un modello ed uno strumento utile per la comprensione del livello di formazione posseduto da un individuo all'interno della comunità europea. Inoltre, è uno strumento idoneo all'equiparazione dei livelli di istruzione tra le nazioni europee.
Nell'allegato 1 dell'EQF vengono definiti i concetti di "*conoscenze*", "*abilità*" e "*competenze*":

- *Conoscenze*: "risultato dell'assimilazione di informazioni attraverso l'apprendimento. Le conoscenze sono un insieme di fatti, principi, teorie e pratiche

relative ad un settore di lavoro o di studio. Nel contesto del Quadro europeo delle qualifiche le conoscenze sono descritte come teoriche e/o pratiche".

- *Abilità*: "indicano le capacità di applicare conoscenze e di utilizzare know-how per portare a termine compiti e risolvere problemi. Nel contesto del Quadro europeo delle qualifiche le abilità sono descritte come cognitive (comprendenti l'uso del pensiero logico, intuitivo e creativo) o pratiche (comprendenti l'abilità manuale e l'uso di metodi, materiali, strumenti)".

- *Competenze*: "comprovata capacità di utilizzare conoscenze, abilità e capacità personali, sociali e/o metodologiche, in situazioni di lavoro o di studio e nello sviluppo professionale e personale. Nel contesto del Quadro europeo delle qualifiche le competenze sono descritte in termini di responsabilità e autonomia".

Nell'allegato 2 dell'EQF, chiamato "Descrittori che definiscono i livelli del Quadro europeo delle qualifiche", viene specificata una griglia in cui sono presenti 8 livelli di formazione ed istruzione e le relative conoscenze, abilità e competenze possedute da ciascun livello.
La raccomandazione del Parlamento Europeo e il Consiglio dell'Unione Europea definisce anche ulteriormente che gli stati membri devono recepire tale raccomandazione ed uniformare a questa il proprio livello di formazione ed istruzione. L'Italia, quindi, recepisce la raccomandazione nel 20 dicembre 2012 attraverso l'accordo, a conferenza unificata, denominato"La Conferenza permanente per i rapporti tra lo Stato, le Regioni e le Provincie Autonome di Trento e Bolzano" che descrive l'"Accordo sulla referenziazione del sistema italiano delle qualificazioni al Quadro Europeo delle qualifiche per l'apprendimento permanente (EQF) di cui alla Raccomandazione del Parlamento Europeo e del Consiglio del 23 aprile 2008". L'allegato B di questo accordo, nominato "Quadro sinottico di referenziazione delle qualificazioni pubbliche nazionali ai livelli del Quadro Europeo delle qualificazioni per l'apprendimento permanente" descrive e relaziona formalmente i livelli dell'EQF alle qualifiche conseguite con il sistema d'istruzione italiano, vedi tabella 5.1.

Attraverso questo schema qualunque cittadino di uno stato membro della comunità europea può confrontare il suo livello di istruzione con quello italiano.

5.2 Sperimentazione di SIRENE

Per la sperimentazione di SIRENE si è reso necessario definire alcuni parametri: il numero e la tipologia di utenti da prendere in considerazione ovvero gli studenti ed il loro grado di istruzione, la somministrazione agli studenti in un periodo di

Livello EQF	Tipologia di qualificazione
Livello 1	Diploma di licenza conclusiva del I ciclo di istruzione
Livello 2	Certificazione delle competenze di base acquisite in esito all'assolvimento dell'obbligo di istruzione
Livello 3	Attestato di qualifica di operatore professionale
Livello 4	Diploma professionale di tecnico Diploma liceale Diploma di istruzione tecnica Diploma di istruzione professionale Certificato di specializzazione tecnica superiore
Livello 5	Diploma di tecnico superiore
Livello 6	Laurea Diploma Accademico di I livello
Livello 7	Laurea Magistrale Diploma Accademico di II livello Master universitario di I livello Diploma Accademico di specializzazione (I) Diploma di perfezionamento o master (I)
Livello 8	Dottorato di ricerca Diploma accademico di formazione alla ricerca Diploma di specializzazione Master universitario di II livello Diploma Accademico di specializzazione (II) Diploma di perfezionamento o master (II)

Tabella 5.1: Livelli delle qualificazioni italiane

uso del framework ed, infine, lo strumento utile per comprendere come validare il framework.

5.2.1 Scelta del campione di studenti per la sperimentazione

Gli studenti, scelti per rappresentare il campione da prendere in esame per la sperimentazione di SIRENE, sono alunni di una classe di liceo scientifico che frequentano il terzo anno, quindi studenti che sono a metà tra il livello 3 e 4 dell'EQF. La scuola del liceo scientifico scelto è l'Istituto Salesiano "Don Bosco" - Villa Ranchibile, il quale è un Istituto Scolastico Paritario Cattolico con Scuola Secondaria di primo grado e Scuola Secondaria di secondo grado dei seguenti indirizzi scolastici: Liceo Scientifico, Liceo Classico e Tecnico Economico. Il docente della scuola, con il quale si è collaborato durante la sperimentazione, è il Prof. Alessio Sofia. Gli studenti, presi in considerazione per la sperimentazione di SIRENE, sono 12 ed hanno seguito un percorso di Alternanza Scuola-Lavoro[1] (ASL). L'obiettivo di tale percorso è permettere agli studenti di acquisire i principi di base del linguaggio di programmazione C per poi espletare un periodo di tirocinio in un'azienda di informatica di Palermo. In quest'ottica, si è scelto l'utilizzo di SIRENE come strumento per un case study: imparare il costrutto condizionale *If..else* ed i costrutti dei cicli *For* e *While* ed il costrutto sequenza. Gli studenti hanno utilizzato il framework in 3 incontri, ciascuno della durata di 2 ore. Durante ogni incontro i ragazzi dovevano sviluppare in gruppo (ciascun gruppo era composto da 2 studenti) la soluzione di alcuni problemi usando SIRENE.

5.2.2 Modalità e scelta dello strumento per validare SIRENE

Al fine di verificare la validità di SIRENE, si è reso necessario somministrare agli studenti un test prima e dopo l'utilizzo del framework. L'obiettivo di tale test è di verificare lo stato di miglioramento del pensiero computazionale degli studenti. Per determinare la differenza del livello del pensiero computazionale degli studenti si è somministrato lo stesso test a tutti gli studenti, sia prima sia dopo l'uso di SIRENE, si precisa che: a conclusione del test iniziale non è stato fornito alcun risultato delle risposte fornite. In questa maniera, gli studenti non erano

[1] L'alternanza scuola-lavoro è stata introdotta dalla legge n. 53 del 28 marzo 2003, decreto legislativo n. 77 del 15 aprile 2005, decreto legislativo n. 81 del 9 aprile 2008, decreti del Presidente della Repubblica n. 87, n. 88 e n. 89 del 15 marzo 2010, direttiva del MIUR n. 4 e n. 5 del 16 gennaio 2012, decreto legge n. 104 del 12 settembre 2013, decreto legislativo n. 81 del 15 giugno 2015, legge n. 107 del 13 luglio 2015, decreto legislativo n. 150 14 settembre 2015.

Livello	Tipologia di alunni
KiloBebras	alunni delle scuole primarie (8-10 anni circa)
MegaBebras	alunni delle classi prima e seconda delle scuole secondarie di primo grado (10-12 anni circa)
GigaBebras	alunni delle classi terze delle scuole secondarie di primo grado (12-13 anni circa)
TeraBebras	alunni del biennio delle scuole secondarie di secondo grado (13-15 anni circa)
PetaBebras	alunni del triennio delle scuole secondarie di secondo grado (15-18 anni circa)

Tabella 5.2: Livelli di quesiti di Bebras

in grado di determinare quali fossero le risposte corrette così da poter rieseguire successivamente lo stesso test e determinare la differenza tra i due test, iniziale e finale.
La scelta dello strumento per validare il framework è ricaduta in "Bebras"[2]. Bebras nasce nel 2004 in Lituania ed "è un'organizzazione internazionale che ha lo scopo di promuovere nelle scuole gli aspetti scientifici dell'informatica. I giochi Bebras sono accessibili agli studenti delle scuole primarie e secondarie anche senza nessuna specifica conoscenza pregressa. I problemi presenti nel sistema propongono reali situazioni informatiche, che richiedono di interpretare informazioni, manipolare strutture discrete, elaborare dati e ragionare algoritmicamente". Le caratteristiche dei problemi proposti ai ragazzi hanno permesso di scegliere Bebras come strumento utile alla determinazione di un punteggio del pensiero computazionale dei studenti. Bebras presenta diverse edizioni annuali dei quesiti proposti e cinque livelli di difficoltà, ogni livello è specifico alla fascia di età ed anno di scuola, vedi tabella 5.2.2.

Poiché il campione è rappresentato da studenti del terzo anno di un liceo scientifico, si è scelto di usare il livello "PetaBebras" edizione 2017/18. In questa edizione di quesiti, vi sono 10 domande nelle quali gli studenti devono riuscire a rispondere in maniera corretta. Il punteggio massimo è 37, in caso di risposta corretta a tutte le domande. È ammesso un punteggio parziale per ogni domanda. Per alcune domande il punteggio massimo è 3, mentre per altre è 4. In soli due casi il punteggio massimo è 5. La somma totale delle risposte non può superare il valore 37. I quesiti di Petabebras edizione 2017/18 affrontano un insieme di problemi informatici mascherati da problemi comuni.
Nel primo quesito gli studenti devono affrontare un problema di calcolo di una fun-

[2]L'indirizzo di Bebras è `https://bebras.it/`.

zione hash, in cui è necessario inserire alcuni oggetti all'interno di quattro scatole in maniera deterministica.
Nel secondo quesito gli studenti affrontano il problema dell'apprendimento automatico, ovvero di machine learning. Gli studenti devono decidere se alcuni oggetti, in base alle loro caratteristiche e ad alcune semplici regole, possono essere classificati in una di due classi.
Nella terza domanda gli studenti devono risolvere un problema di programmazione. Individuare la corretta sequenza di azioni che un ipotetico robot deve eseguire per svolgere un'attività.
La quarta domanda è associata ad un problema di calcolo combinatorio. Individuare la corretta sequenza di un sistema binario.
Nella quinta domanda viene proposto un problema di ricerca operativa. Individuare la corretta sequenza di oggetti che ottimizza una pianificazione.
La sesta domanda è associata ad un problema di regular expression. Individuare la conseguenza di un'azione legata ad uno string matching.
Nel settimo quesito gli studenti affrontano un problema di compressione dati. Individuare la corretta sequenza fornita da un algoritmo di compressione.
L'ottavo quesito è in relazione con un problema sui grafi e programmazione. Individuare la corretta sequenza di istruzioni affinché un robot possa eseguire un corretto cammino in un grafo.
La nona domanda affronta un problema di programmazione imperativa e di un sistema binario. Modificare in maniera corretta lo stato di alcuni oggetti, arrivando ad una configurazione finale precisa partendo da una configurazione qualsiasi.
Infine, nella decima domanda viene proposto un altro problema di pattern recognition. Gli alunni devono riconoscere alcune caratteristiche specifiche affinché si possano identificare un insieme di oggetti in maniera univoca.
Ognuno dei primi 5 quesiti fornisce un punteggio massimo di 3 punti. Dal sesto all'ottavo quesito si ottiene un massimo di 4 punti per ciascuna risposta esatta. Mentre ognuna delle ultime due domande permette di ottenere un punteggio massimo di 5 punti.

5.3 Risultati

I dati globali ottenuti dai test prima e dopo la sperimentazione sono riportati in tabella 5.3. In figura 5.1 è riportato il grafico del punteggio dei test sia prima sia dopo gli incontri in cui gli studenti hanno utilizzato SIRENE: nelle ascisse viene riportato il numero dello studente e nelle ordinate il punteggio ottenuto. In colore blu viene evidenziato il punteggio globale, per singolo studente, del test iniziale, mentre il punteggio del test finale per singolo studente viene messo in evidenza dal colore arancione.

Studente	Prima	Dopo
1	32	37
2	31	37
3	21	26
4	18	22
5	13	37
6	20	31
7	25	27
8	24	36
9	34	37
10	17	22
11	15	33
12	26	37

Tabella 5.3: Punteggi globali dei test prima e dopo la sperimentazione

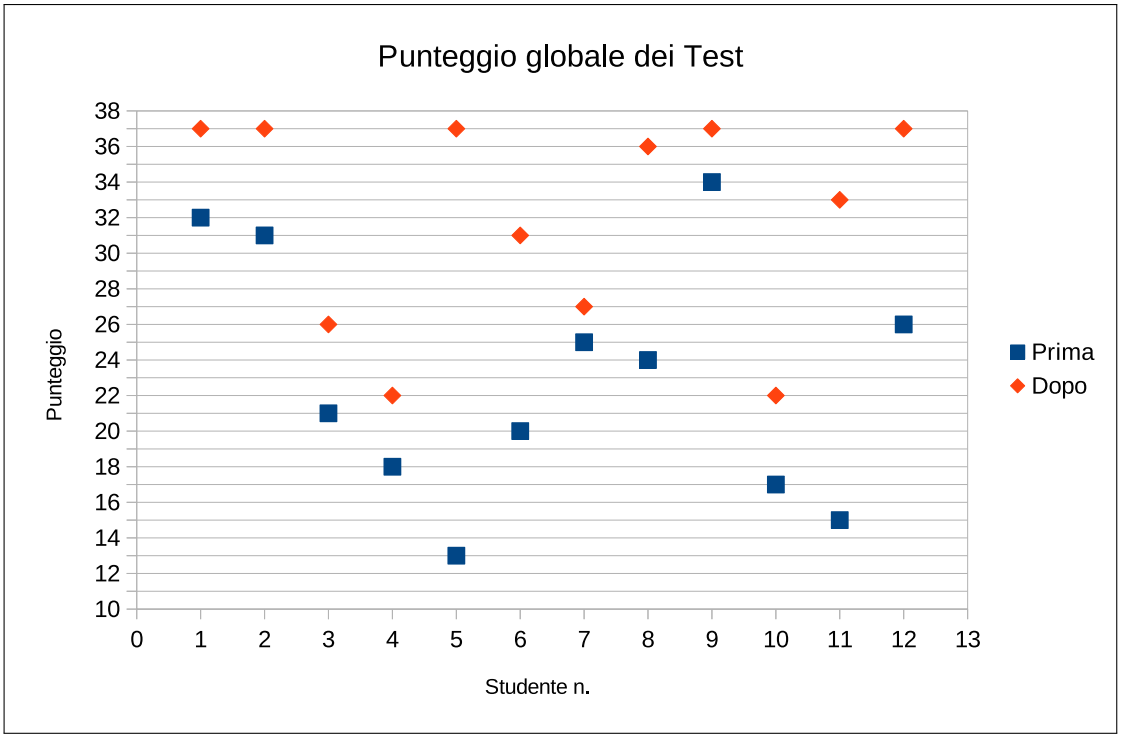


Figura 5.1: Grafico dei punteggi globali dei test.

Considerando il breve periodo di soli tre incontri dell'uso di SIRENE, i risultati ottenuti sono incoraggianti. Giova ricordare che il massimo punteggio che si può ottenere rispondendo correttamente a tutte le 10 domande è 37. Dai dati, nel pri-

mo test la media degli studenti è stata di 23, con una deviazione standard di 6.86. Mentre, nel test finale la media ottenuta è stata di 31.83, con una deviazione standard di 6.06. La differenza della media tra il secondo ed il primo test è di 8.83 e la differenza della deviazione standard tra i due test è di quasi un punto, esattamente 0.80. Da questi dati, sembrerebbe che l'uso di SIRENE abbia migliorato il pensiero computazionale degli studenti. Infatti, tutti gli studenti hanno migliorato le loro risposte ai singoli quesiti e la media del punteggio del secondo test è notevolmente incrementata. Inoltre, l'abbassamento della deviazione standard induce a pensare che l'uso di SIRENE abbia diminuito la differenza del pensiero computazionale tra gli studenti. Questo è esattamente ciò che ci si aspettava. Inoltre, è interessante evidenziare il caso dello studente n. 5. Nel primo test, lo studente ha ottenuto un punteggio di 13, mentre nel secondo test ha ottenuto un punteggio di 37. A prima vista, si è pensato ad un caso di outlier. Un risultato simile può essere fatto per lo studente numero 11, il quale ha migliorato il suo punteggio da 17, nel primo test, a 33, nel secondo test. Questo potrebbe essere spiegato dal fatto che gli studenti sono indotti a riflettere durante l'uso di SIRENE ed a concentrarsi sull'aspetto concettuale logico-risolutivo di un problema, ed a tradurre una soluzione usando il linguaggio di SIRENE. Ciò potrebbe migliorare negli studenti la consapevolezza della propria capacità di problem solving.
È utile eseguire un'analisi dei punteggi degli studenti sulle singole risposte. In figura 5.2 è mostrato il grafico che visualizza i punteggi degli studenti dei test iniziale e finale.
Nella prima domanda, gli studenti hanno mostrato un andamento quasi identico sia nel primo test sia nel secondo test, ad eccezione di due studenti i quali hanno invertito le risposte: uno ha migliorato, l'altro ha leggermente peggiorato. Nessuno studente, in questo quesito, ha fallito pienamente in entrambi i test.

Nel secondo quesito, vedi figura 5.3, 5 studenti hanno migliorato il loro punteggio, mentre due studenti hanno peggiorato il punteggio. Gli altri studenti hanno risposto nel secondo test nella stessa maniera di come avevano risposto nel primo test. Anche in questo quesito, nessuno studente ha fallito pienamente in entrambi i test.

Nel terzo quesito, vedi figura 5.4, due studenti hanno migliorato il loro punteggio. Gli altri, invece, hanno risposto in entrambi i test nella stessa maniera. Soltanto uno studente ha fallito completamente in questo quesito in entrambi i test.

Nella quarta domanda, vedi figura 5.3, 5 studenti hanno migliorato il loro punteggio nel secondo test, mentre per gli altri studenti il risultato è rimasto invariato. Anche in questa domanda, soltanto uno studente ha fallito completamente in entrambi i test.

Nella quinta domanda, vedi figura 5.6, gli unici 4 studenti, che avevano sba-

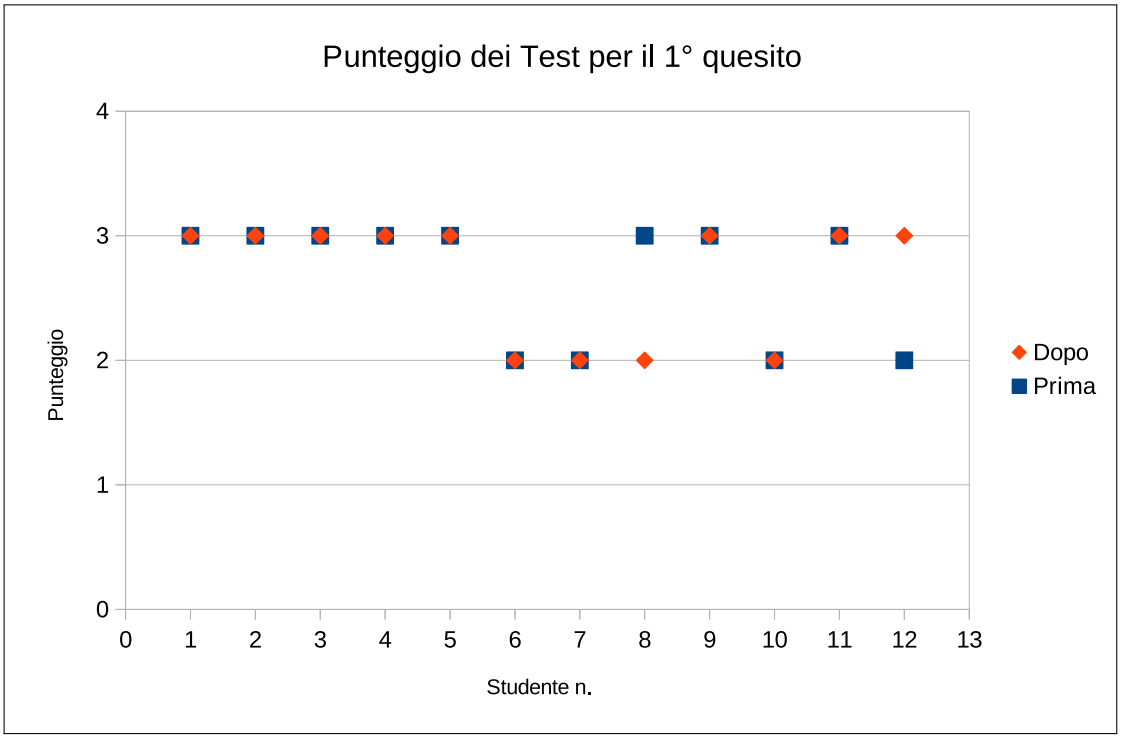


Figura 5.2: Grafico del punteggio dei Test per il quesito 1.

gliato nel primo test, hanno migliorato pienamente il loro risultato. Nel secondo test, tutti gli studenti hanno risposto correttamente in questa domanda.

Dalla figura 5.7, nel sesto quesito 3 studenti hanno migliorato pienamente il loro risultato, mentre tutti gli altri studenti hanno invariato la loro risposta. In questa domanda soltanto uno studente ha fallito il quesito in entrambi i test.

Dalla figura 5.8, nella settima domanda 3 studenti hanno migliorato pienamente il loro risultato, mentre un solo studente ha peggiorato il risultato. Anche in questo quesito soltanto uno studente ha fallito in entrambi i test.

Nell'ottavo quesito, vedi figura 5.9, 5 studenti hanno migliorato pienamente il loro risultato, mentre soltanto uno studente ha peggiorato il risultato fallendo il test. Tutti gli altri studenti non hanno variato le risposte. In questa domanda, 4 studenti hanno fallito completamente in entrambi i test.

Dalla figura 5.8, nella nona domanda si evince che tre studenti hanno pienamente migliorato, mentre soltanto due hanno fallito il secondo test. Tutti gli altri studenti sono rimasti invariati nelle loro risposte. In questo quesito nessuno studente ha fallito in entrambi i test.

Nel decimo quesito, vedi figura 5.11, hanno migliorato pienamente 7 studenti, mentre soltanto uno studente ha fallito il secondo test. Anche in questo quesito

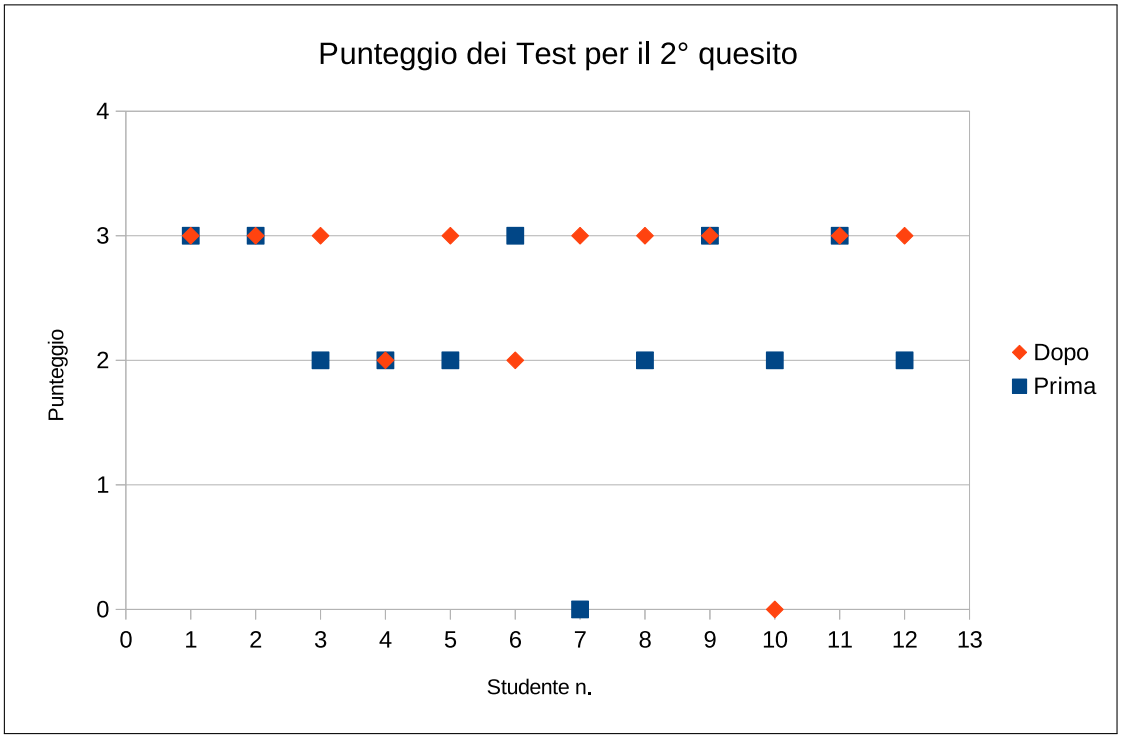


Figura 5.3: Grafico del punteggio dei Test per il quesito 2.

nessuno studente ha fallito in entrambi i test.

Dai grafici, si evince che in ogni domanda dei test, gli studenti migliorano il risultato dopo avere utilizzato SIRENE.

5.4 Valutazione di SIRENE da parte degli studenti

Alla fine dei tre incontri di utilizzo di SIRENE, è stato chiesto agli studenti una loro valutazione sull'utilizzo del framework. Tutti gli studenti hanno riferito che usare il framework è stato relativamente semplice sin dal primo utilizzo. Superata una fase iniziale, gli studenti hanno preso facilmente confidenza con lo strumento e riuscivano a realizzare i programmi in maniera autonoma ed intuitiva. Alcuni studenti hanno riferito di aver apprezzato molto positivamente la visualizzazione del codice sorgente vicino all'area in cui si sviluppa il comportamento degli algoritmi realizzati con il linguaggio di SIRENE. Inoltre, gli studenti hanno apprezzato la modalità della costruzione delle espressioni con il linguaggio di SIRENE e la visualizzazione del codice sorgente. Un solo studente ha riferito, inizialmente, di

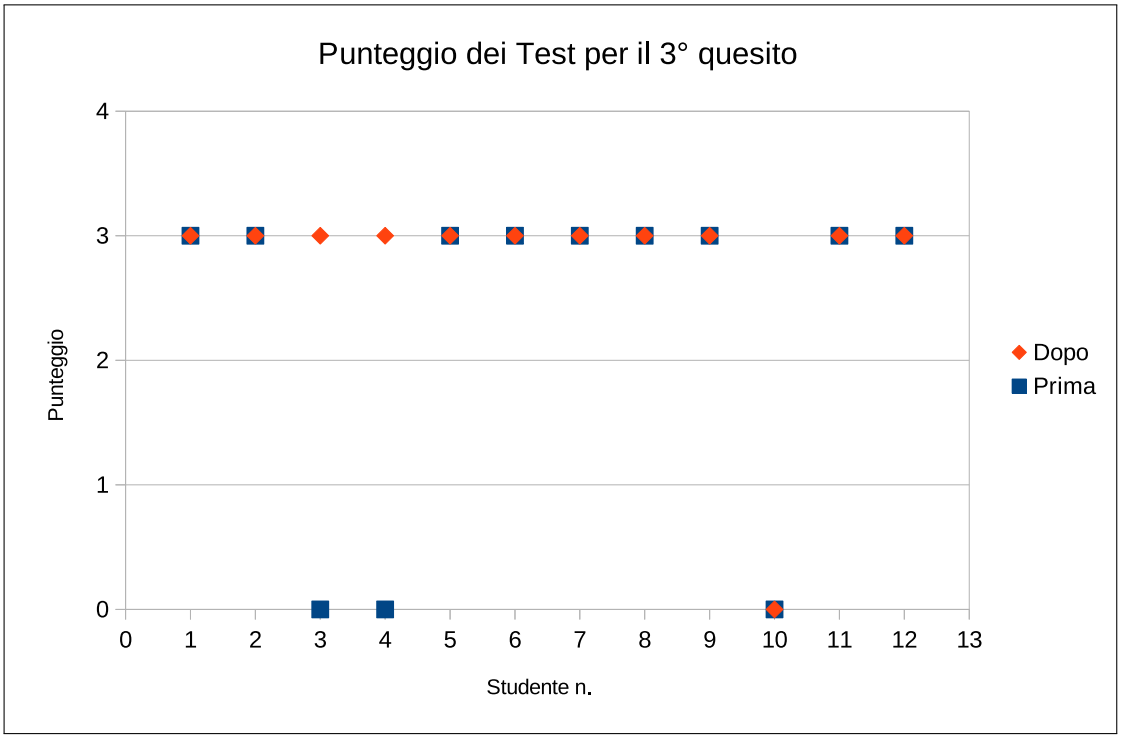


Figura 5.4: Grafico del punteggio dei Test per il quesito 3.

voler utilizzare un classico ambiente testuale, ma di aver anche lui apprezzato (alla fine degli incontri) la semplicità e la modalità di utilizzo di SIRENE. Pertanto tutti gli studenti ritengono utile avere la possibilità di interagire con i programmi realizzati, potendone verificare la correttezza del loro operato.
In definitiva, gli studenti hanno affermato che l'esperienza dell'uso del framework è stata positiva.

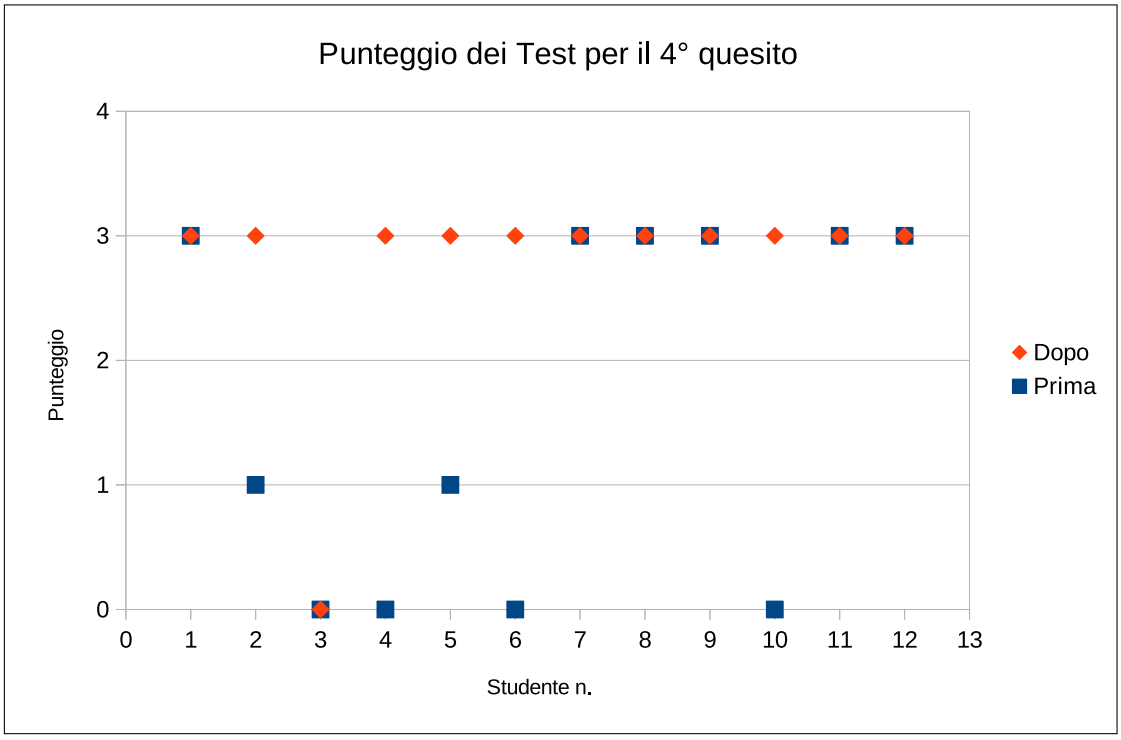


Figura 5.5: Grafico del punteggio dei Test per il quesito 4.

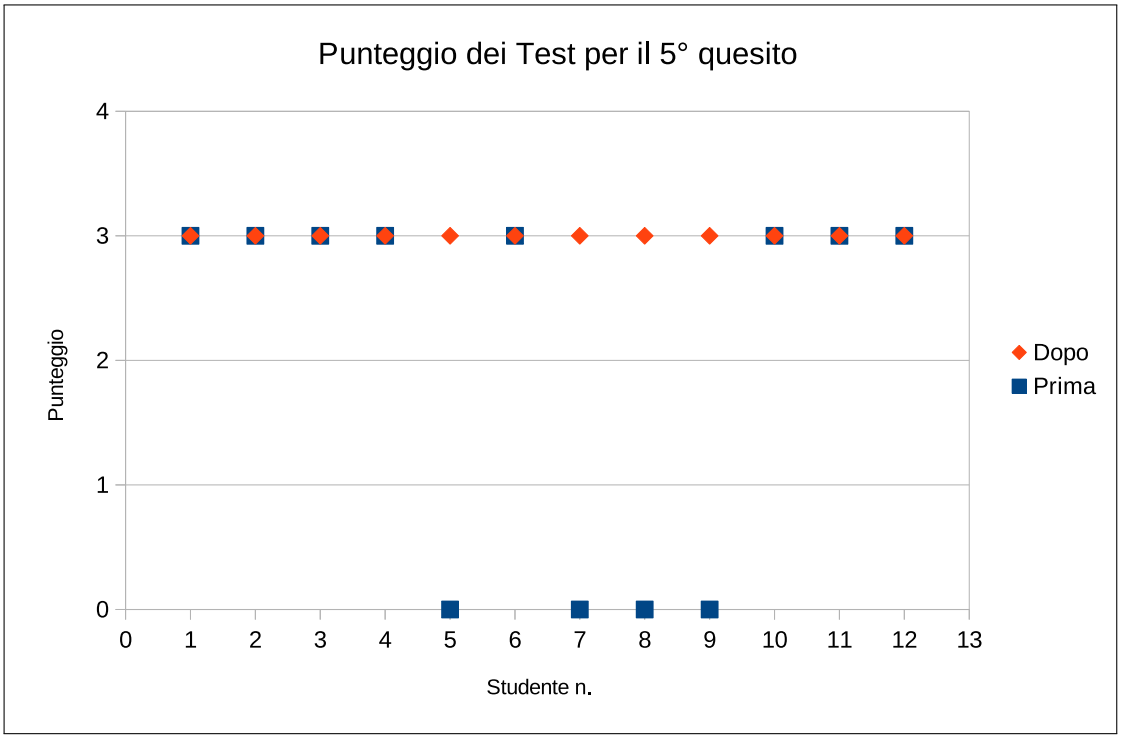


Figura 5.6: Grafico del punteggio dei Test per il quesito 5.

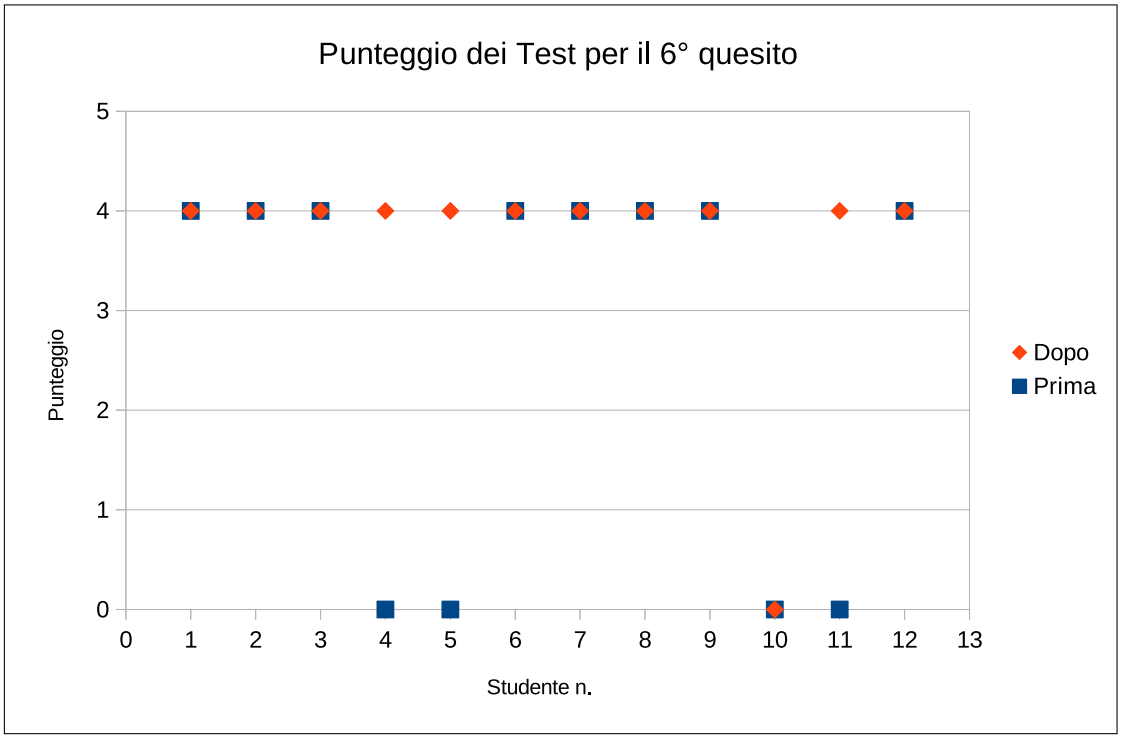


Figura 5.7: Grafico del punteggio dei Test per il quesito 6.

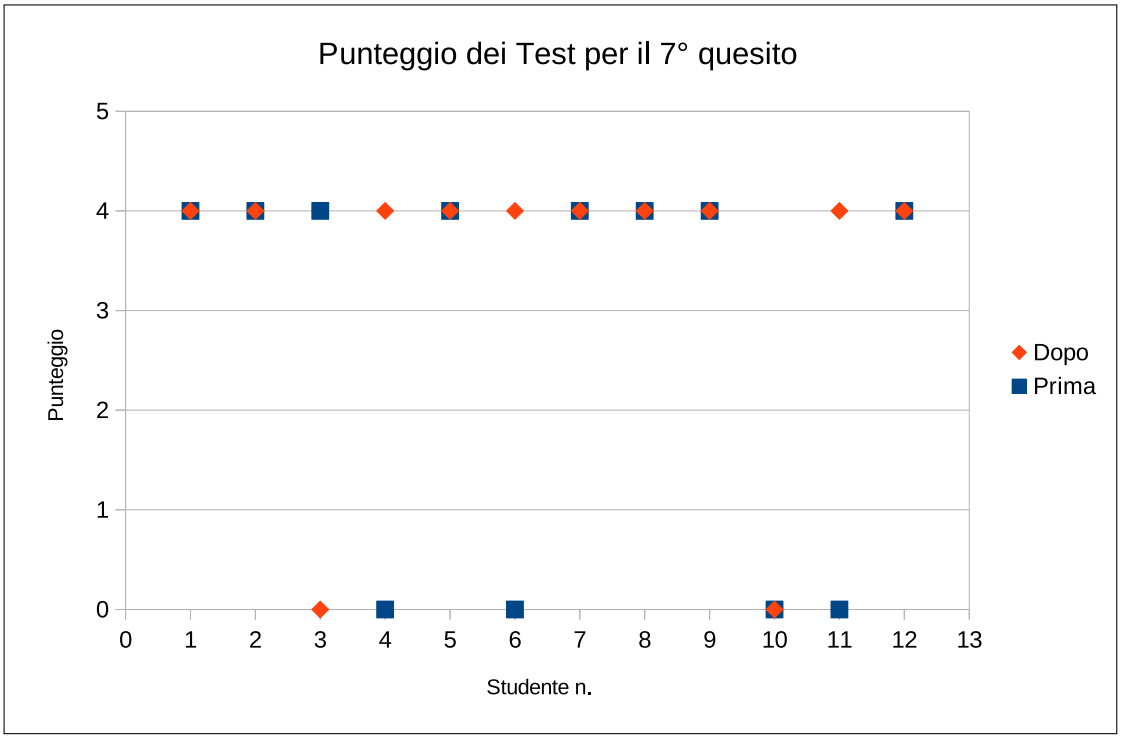


Figura 5.8: Grafico del punteggio dei Test per il quesito 7.

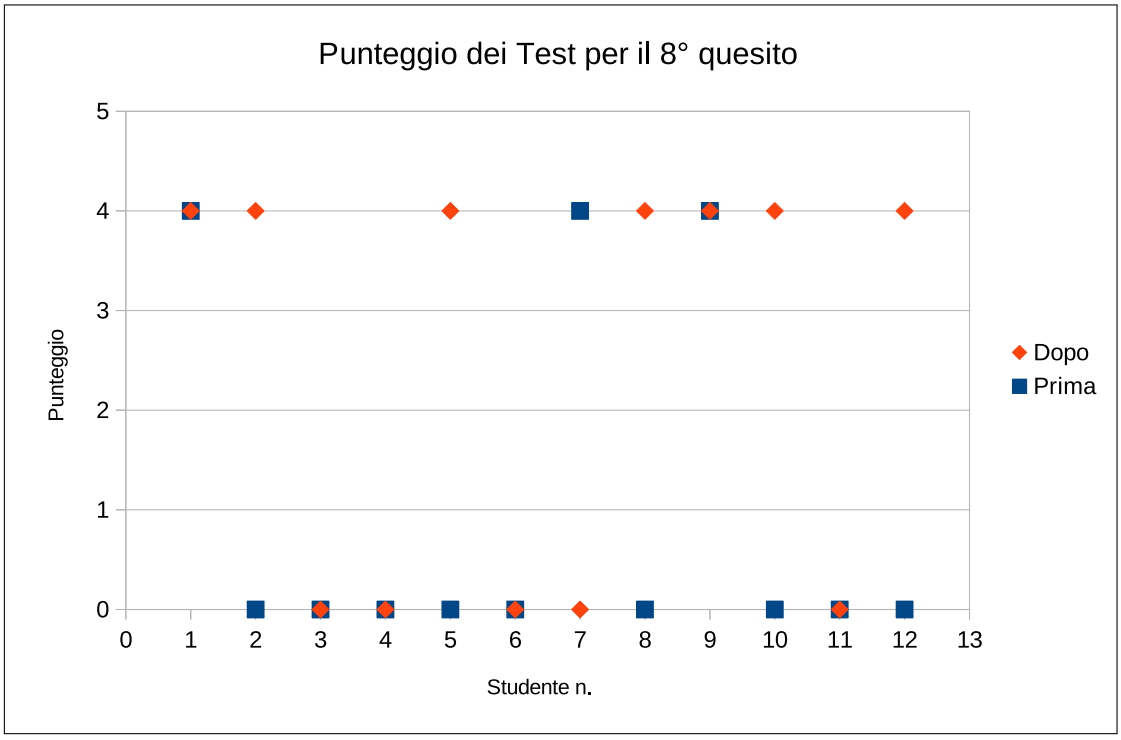


Figura 5.9: Grafico del punteggio dei Test per il quesito 8.

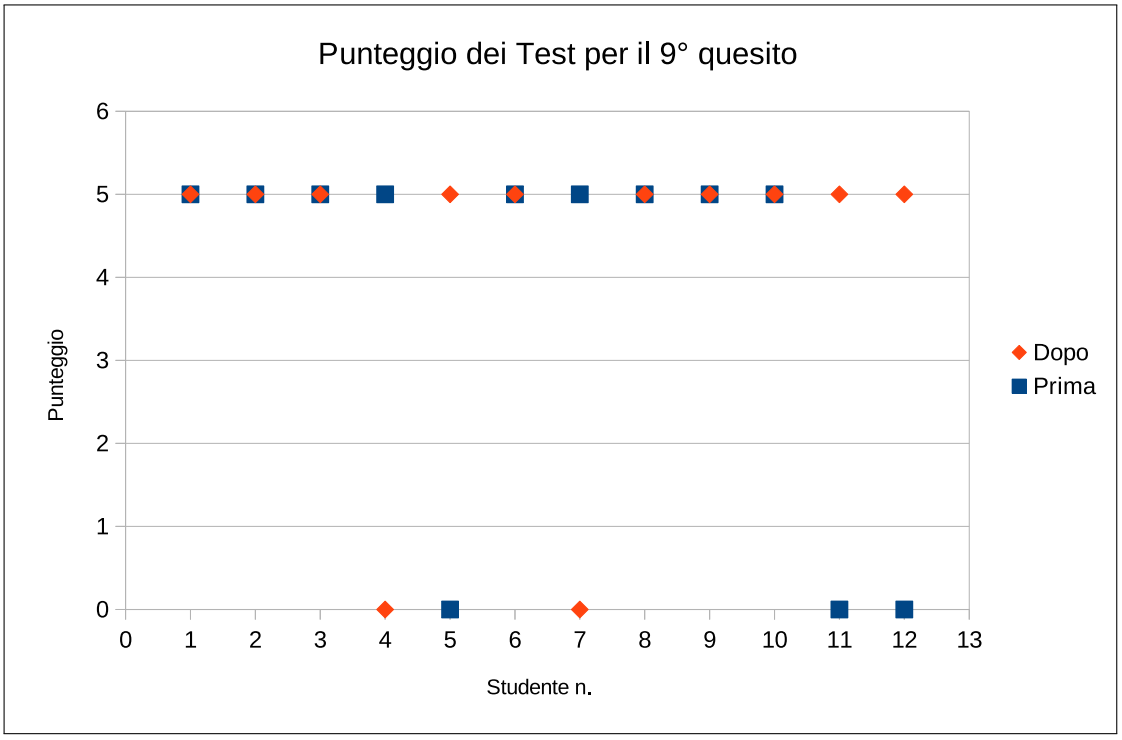


Figura 5.10: Grafico del punteggio dei Test per il quesito 9.

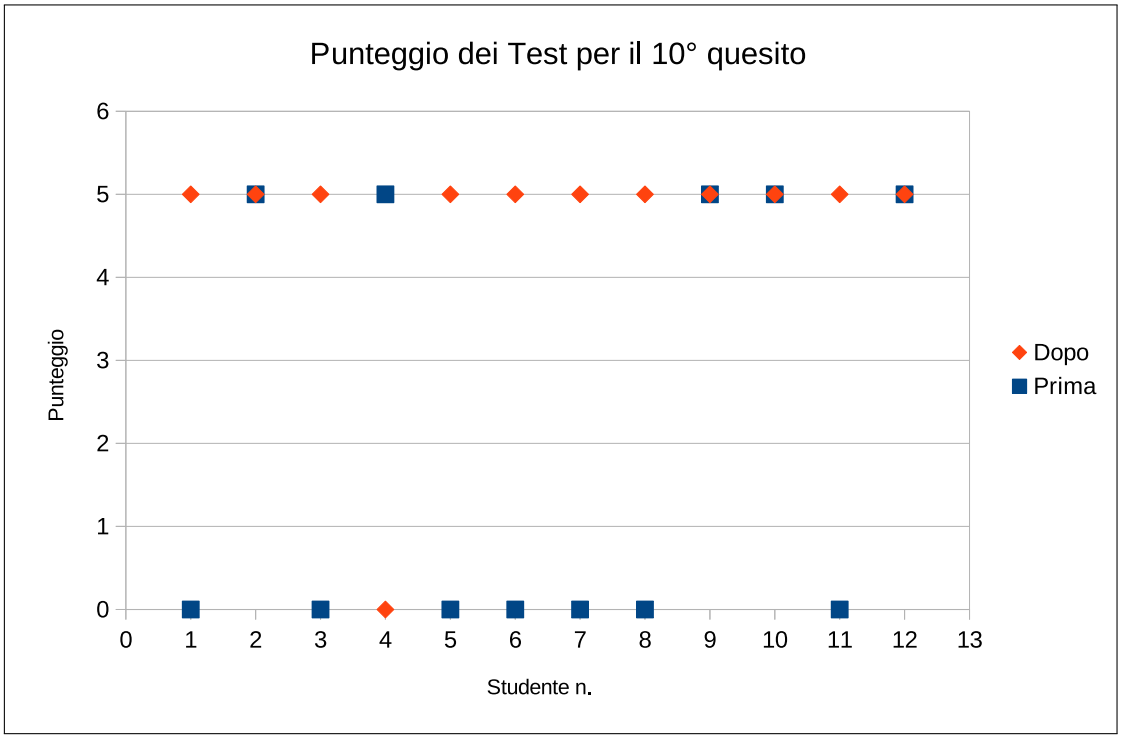


Figura 5.11: Grafico del punteggio dei Test per il quesito 10.

Capitolo 6

Conclusioni

6.1 Analisi finale

Si è discusso del funzionamento del linguaggio iconico e delle caratteristiche principali di SIRENE assicurando una visione precisa delle caratteristiche uniche di SIRENE ed, in generale, del suo funzionamento. Si è discusso delle caratteristiche grafiche del linguaggio iconico, del funzionamento del parser iconico e delle caratteristiche di condivisione e di interazione del framework.
Questo ha permesso la disseminazione a conferenza sei seguenti articoli:

- "*An Iconic Framework for Learning the Art of Programming*"[4], Proceedings of "International Conference on e-learning", "e-Learning'16", Bratislava (Slovacchia), 8-9 Settembre 2016, ISSN: 2367-6698 (print), ISSN: 2367-6701 (cd-rom), ISSN: 2367-6787 (online).

- "E-Learning and art of programming: a context oriented to"[5], Proceedings of "International Conference on Education and New Learning Technologies", "EDULEARN17", Barcellona (Spagna), 3-5 Luglio 2017, ISBN: 978-84-697-3777-4, ISSN: 2340-1117.

- "SIRENE - Framework sull'arte della programmazione"[1], Proceedings of "Giornate di studio dell'Insegnante di MATematica 2017", "GIMat 2017", 20-21 Ottobre 2017, Palermo (Italia), ISSN 1: 1592-4424, ISSN 2: 1592-5137.

- "Iconic framework for cooperative coding"[3], Proceedings of "International Conference on Computer Systems and Technologies", "CompSysTech'18", 13-14 Settembre 2018, Ruse (Bulgaria), ACM ISBN 978-1-4503-6425-6.

- "SIRENE, framework per l'insegnamento della matematica"[2], Proceedings of "Giornate di studio dell'Insegnante di MATematica 2018", GIMat 2018", 19-20 Ottobre 2018, Catania (Italia), ISSN 1: 1592-4424, ISSN 2: 1592-5137.

Sirene ha ricevuto il premio "*Crystal Prize*" come "*best paper*" nella conferenza "CompSysTech'18" avvenuta in Ruse (Bulgaria). Inoltre, SIRENE è stato proposto durante un seminario della conferenza *Gi.Mat. 2018*" (Giornate di studio dell'Insegnante di MATematica 2018 in Catania nelle date 19 e 20 ottobre 2018) come strumento multidisciplinare per l'insegnamento della matematica e per lo sviluppo del pensiero computazionale.
SIRENE è l'unico linguaggio visuale di programmazione che permette di essere utilizzato in modalità cooperativa tra gli utenti per sviluppare programmi ed il Client Device di SIRENE è utilizzabile in qualsiasi dispositivo dotato di un browser, ovvero dalla quasi totalità dei sistemi operativi, mentre il Server Device di SIRENE è eseguibile in un vasto numero di sistemi operativi senza la necessità di modifiche sostanziali nel codice. Inoltre, SIRENE permette di seguire lezioni in una stessa aula ed in aule differenti, in modalità e-learning.
SIRENE è stato implementato per l'insegnamento dell'arte della programmazione, focalizzando l'attenzione sia dal punto di vista dell'implementazione concettuale ed astratta degli algoritmi e sia dal punto di vista dell'apprendimento della corretta sintassi e semantica del linguaggio di programmazione testuale come il C.
SIRENE, inoltre, permette di ottenere il codice sorgente in linguaggio C associato al linguaggio iconico e permette inoltre di compilare questo codice sorgente e di eseguire i relativi eseguibili. Infine, SIRENE permette a tutti gli utenti di interagire con i programmi realizzati con il linguaggio iconico e compilati, gestendo l'invio di tutti gli output dei programmi e la ricezione degli input per i programmi mandati dai Client Device.
I risultati sperimentali di SIRENE sono stati positivi: gli studenti che hanno partecipato alla sperimentazione hanno riferito di avere apprezzato l'uso del framework. Inoltre, gli studenti sono stati sottoposti ad un test iniziale e finale dopo l'uso di SIRENE. Il confronto di questi test è risultato incoraggiante: tutti gli studenti hanno avuto un notevole miglioramento, vedi paragrafo 5.3. Dai dati ottenuti, sembrerebbe che l'uso di SIRENE migliori e potenzi il pensiero computazionale.

6.2 Lavori futuri

Sarebbe auspicabile eseguire ulteriori fasi di test in diverse classi scolastiche e/o universitarie, per verificare la bontà del sistema nonchè per il miglioramento del sistema stesso. Inoltre, sarebbe interessante verificare se l'uso di SIRENE permetta agli studenti di realizzare una migliore e maggiore auto-consapevolezza di analisi e

problem solving. Inoltre, sono prese in considerazione ulteriori ampliamenti delle caratteristiche quali:

- Miglioramento della gestione dei gruppi degli studenti.
- Capacità di visualizzazione "step-by-step" del funzionamento durante l'esecuzione dei programmi, evidenziando le icone ed il codice sorgente nei Client device.
- Capacità di debugging delle applicazioni.
- Capacità di rivedere una lezione eseguita. Questa caratteristica è simile alla video registrazione di una lezione, tuttavia la dimensione, in termini di Byte, sarà notevolmente inferiore.
- Inserimento dell'audio nelle lezioni.
- Realizzazione di una chat, con condivisione di eventuali files, per la comunicazione tra gli utenti che seguono una stessa lezione.
- Inserimento di più modalità di utilizzo delle lezioni: modalità tradizionale (in cui vi è un unico utente, ad esempio l'insegnante, che può eseguire le azioni in SIRENE mentre gli altri utenti possono solo seguire), modalità di tutoring (peer tutoring, peer to peer, ecc.) e modalità d'interazione con le lezioni miste tra tradizionale e collaborativo per permettere agli insegnanti di realizzare nuove forme sperimentali di metodologie d'insegnamento.

Sarà presa in considerazione la possibilità di inserire ulteriori caratteristiche in base alle esperienze fornite dagli utenti i quali utilizzeranno SIRENE durante le ulteriori fasi di test.

Appendice A

Implementazione del Server di SIRENE

A.1 Componenti di SIRENE

SIRENE è costituito da due moduli funzionali, presenti all'interno del computer che ospita l'intero sistema. La componente Server di SIRENE, denominata Sirene Server Device Module (SSDM, nel seguito per brevità ci si riferirà esplicitamente ad esso come Server Device), ha il compito di ricevere ed inviare messaggi verso tutti i Client, verso i programmi che verranno sviluppato ed eseguiti, nonché ha anche il compito della compilazione dei codici sorgenti creati; l'altro modulo definisce il Client di SIRENE, denominato Sirene Client Device Module (SCDM, nel seguito per brevità ci si riferirà esplicitamente ad esso come Client Device), che ha la funzione di gestire gli utenti, include la Graphic User Interface di SIRENE e garantisce e gestisce l'invio dei messaggi dal e verso il Server Device. Nel seguito del capitolo si discuterà dell'implementazione di Sirene, partendo dalla componente Server e terminando con la descrizione della componente Client, vedi fig. A.1.

A.2 Il Server di SIRENE

Il Server SSDM si occupa della gestione della ricezione e dell'invio dei messaggi verso tutti i Client, la compilazione del codice sorgente e l'esecuzione dei programmi compilati, infine dell'interazione di SIRENE con i programmi compilati e con i Client. Il Server Device di SIRENE è composto da due moduli cooperanti per il corretto funzionamento del framework: J-Communicator e J-Manager.

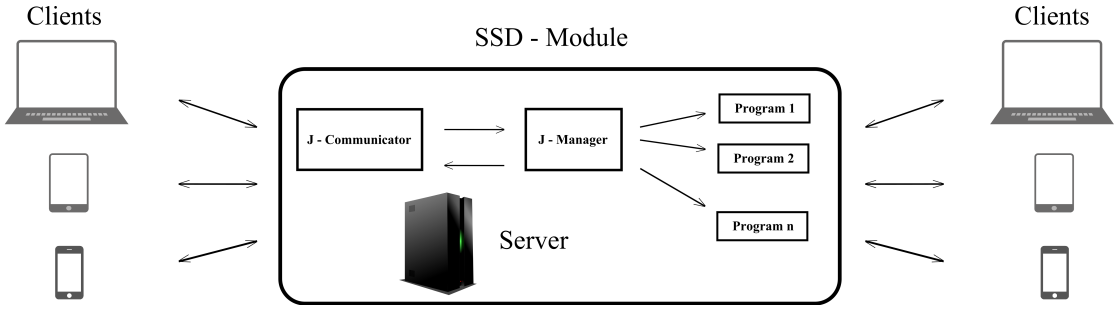


Figura A.1: Architettura di Sirene.

A.2.1 Componente J-Communicator

J-Communicator è il componente Server dell'SSDM che permette la gestione di tutti i messaggi inviati dai Client e la compilazione dei sorgenti dei file generati mediante l'uso del linguaggio iconico descritto nel paragrafo 4.2. Basato sulle potenzialità offerte da Node.js, Express.io e Socket.io, gestisce la connessione dei Client e garantisce la comunicazione dei messaggi di sincronizzazione tra tutti i Client. Gestisce, inoltre, la comunicazione con J-Manager per l'invio, ai Client connessi ad una stessa attività o lezione, di tutti gli output generati dai programmi in esecuzione. Inoltre garantisce l'invio di tutti i dati forniti in input dai Client verso i programmi che richiedono informazioni per continuare la loro esecuzione. J-Communicator è un modulo ad hoc e descrive le attività da realizzare. Esso realizza due Server: uno per per la comunicazione con i Client e uno per la comunicazione con J-Manager.

A.2.2 Server di comunicazione di J-Communicator con i Client Device

Il Server per la comunicazione con i Client gestisce le comunicazioni con i Client e con J-Manager. Esso implementa una serie di funzionalità, ognuna di queste viene eseguita al verificarsi di un evento specifico. Questi eventi coincidono con una particolare stringa presente all'interno dei messaggi ricevuti dai Client. Attualmente è stata implementata soltanto la modalità "*Cooperativa_sincrona*".
Il Server di comunicazione con i Client possiede una serie di eventi: "*Connection*", "*HTML*", "*Azione*", "*Comando*", "*Collega*", "*Compila*", "*Aggiorna_Html*", "*Invia_Input*". Per ridurre il carico di lavoro di J-Communicator, soprattutto nei casi in cui ci sia un numero elevato di utenti connessi ed anche un numero elevato di confronti tra stringhe, sono state implementate funzionalità semplici ma efficaci per l'inoltro di tutti i messaggi di sincronizzazione tra i Client: "*Azione*", "*Coman-*

do". Generalmente, un tipico messaggio, come protocollo di applicazione, inviato da un Client Device al Server Device, (ricordo che i Client possono collegarsi solo con il componente J-Communicator) è composto, in formato JSON, come nel seguente esempio:

```
{
    Evento: Stringa_Evento,
    Messaggio:
    {
        Funzione_Nome_Funzione,
        Parametri:
        {
            Parametro1: Valore_del_Parametro1,
            Parametro2: Valore_del_Parametro2,
            ...
            ParametroN: Valore_del_ParametroN
        }
    }
};
```

Questa tipologia di protocollo permette di velocizzare la ricerca delle azioni che J-Communicator deve svolgere. Il valore dell'attributo "*Evento*" corrisponde alla stringa che evidenzia l'evento che J-Communicator deve svolgere, ovvero la funzione che deve eseguire. La stringa presente nel campo "*Messaggio*" contiene il messaggio che J-Communicator deve inviare a tutti i Client connessi alla lezione. Questa tipologia di protocollo viene utilizzata nella quasi totalità delle azioni e comandi che J-Communicator deve eseguire. Nello specifico: gli eventi *Azione*, *Comando* usano il seguente protocollo di azioni: J-Communicator deve semplicemente eseguire il controllo del tipo di evento, se azione o comando, per eseguire la corretta funzione: se l'evento rappresenta *Azione* allora il messaggio viene inviato a tutti i Client che eseguiranno la funzione inserita nel campo *Funzione* inserendo automaticamente come unico parametro l'oggetto "*Parametri*" del messaggio, in questa maniera i Client e J-Communicator non devono eseguire alcun controllo ma solo le istruzioni. Questa modalità agevola l'esecuzione dell'invio delle istruzioni sia da parte del Client Device sia del Server Device. Inoltre, questo protocollo non necessita di implementare alcuna funzionalità ulteriore per l'invio e lo smistamento dei messaggi da parte sia del Client Device sia del Server Device. Nel Client Device bisognerà implementare soltanto la funzione da dovere invocare quando il Client Device riceve il messaggio.
Nell'ipotesi in cui l'evento contenesse la stringa *Comando* allora il campo "*Funzione*" nel protocollo si chiamerà "*Comando*" e J-Communicator capirà che deve

eseguire un controllo poiché necessita di eseguire una particolare azione, come inviare messaggi a J-Manager o eseguire qualche altra azione. I comandi attualmente implementati, ma facilmente estendibili, sono "*Resetta*", "*Inviato_Input*", "*Termina_Programma*". Usando il comando *Resetta*, viene eseguito il reset del codice HTML per tutti i Client Device e per il Server Device. Al comando *Inviato_Input*, J-Communicator esegue l'invio dell'input ricevuto da un Client verso il componente J-Manager (che si occuperà di inviarlo al programma in attesa di input) e successivamente invia lo stesso input a tutti i Client Device per simulare la funzionalità di un classico Terminal, o shell, che visualizza i dati immessi nella finestra di prompt. Infine, in seguito ala ricezione del comando *Termina_Programma*, J-Communicator invia un messaggio a J-Manager ordinando di chiudere il programma (nel seguito paragrafo A.2.4 si spiegherà in quale maniera) e, successivamente, invia lo stesso comando a tutti i Client Device affinché chiudano il loro simulatore del Terminal, o shell, di promt.
In ogni caso, il comando viene sempre inviato ai Client Device i quali eseguiranno il *Comando*, ovvero la funzione specificata dal comando, semplicemente richiamandolo, come nel caso di *Azione* (il meccanismo è similare). Inoltre, per ogni *Azione* o *Comando*, J-Communicator inserisce in *Parametri* il nome dell'utente che ha sollevato l'evento in J-Communicator e il momento temporale in formato Timestamp. Questi due dati saranno importanti per la corretta creazione di ogni oggetto dinamico, in quanto ogni utente può eseguire quell'evento soltanto in un istante specifico, maggiori dettagli sono riportati nel paragrafo B.7, dedicato al Client Device. Inoltre, sempre nel paragrafo B.7 viene spiegato nel dettaglio come *Funzione* e *Comando*, vengono eseguiti per tutti i tipi di funzioni con qualunque parametro, in maniera automatica, veloce e snella.
All'evento *Connection*, J-Communicator permette ad un Client di connettersi con SIRENE, soltanto se il Client può fornire credenziali per l'accesso. Se l Client non riesce a fornire credenziali valide, la connessione viene rifiutata, ovvero viene chiusa. Diversamente, al Client vengono assegnate un'insieme di attributi utili alla gestione e alla comunicazione con questo Client ed inoltre, al Client vengono assegnate una o più specifiche attività, o lezioni (nel seguito ci si riferisce a lezioni ed attività per descrivere la medesima cosa), alle quali può partecipare. La modalità della partecipazione alle lezioni dipende dai permessi che un tutor, o insegnante, fornisce ai Client. Questi permessi possono essere: "*Cooperativa_sincrona*", "*Cooperativa_asincrona*", "*Sola_Visione*", "*Negato*". In modalità *Cooperativa_sincrona*, SIRENE permette a tutti i Client Device, che hanno questo permesso, di eseguire le stesse azioni fatte dagli altri utenti, ovvero ogni azione eseguita da un Client in quella lezione verrà eseguita automaticamente, e in tempo reale, da tutti gli altri utenti che hanno il permesso *Cooperativa_sincrona* nella stessa lezione. Quindi, se un utente cliccasse su un tab per visualizzare una funzione allora tutti gli altri

Client visualizzeranno la stessa funzione, e così via.
All'evento *Aggiorna_Html*, J-Communicator aggiorna il testo HTML, memorizzandolo nella variabile globale "*testo*", presente all'interno dello spazio comune ai Client Device. Ciò permette di avere una visualizzazione comune per tutti i Client Device nel caso in cui questi dovessero collegarsi successivamente all'inizio della lezione o dovessero ricollegarsi nuovamente. Questo dato, *testo*, viene sempre aggiornato dal Client che ha eseguito l'ultima azione ed inviato a J-Communicator.
All'evento *Html*, J-Communicator comprende che il Client Device chiede di ottenere il codice HTML aggiornato, precedentemente descritto, della Visual Code Area sincronizzata con tutti i Client. Quindi, J-Communicator invia il testo HTML al Client Device affinché possa avere la stessa visualizzazione della lezione aggiornata e sincronizzata. Tale evento è molto raro in quanto viene eseguito soltanto quando il Client Device si connette o si riconnette.
All'evento *Compila*, J-Communicator riceve nel messaggio i file generati dal Client Device che solleva l'evento. Quindi, J-Communicator salva i file ricevuti e li compila. Se durante la fase di compilazione dovesse avvenire qualche errore di compilazione allora J-Communicator manda un messaggio a tutti i Client Device fornendo anche l'informazione della mancata compilazione. Diversamente, se la compilazione va a buon fine allora J-Communicator crea un client che stabilisce una connessione con il Server di J-Manager e manda un messaggio a J-Manager ordinando di eseguire il programma e di gestire tutti i suoi input ed output. Quindi, J-Communicator manda un messaggio a tutti i Client Device aggiornando il Web-Terminal (web-shell) di simulazione del prompt con il testo "*Successful compilation. Program Starts.*".
All'evento *Invia_Input*, J-Communicator riceve un messaggio contenente i dati da dover inserire nella Web-Terminal, di simulazione del prompt, quindi invia a tutti i Client Device questo messaggio, incluso chi ha sollevato l'evento in quanto la sincronizzazione dei messaggi avviene per mezzo di chiamate di funzioni e parametri. Nel paragrafo B.7 viene spiegata nel dettaglio questa modalità di gestione dei Client.

A.2.3 Server di comunicazione di J-Communicator con J-Manager

All'avvio del Server Device, J-Communicator inizialmente crea un Server tcp nella porta 5000 del localhost, ovvero nell'indirizzo '127.0.0.1'. Questo permette di poter comunicare soltanto con applicazioni in esecuzione nella stessa macchina ed evita che applicazioni esterne possano collegarsi ad esso. Nello specifico, l'unica applicazione permessa ad accedere a questo Server è J-Manager. Questo Server

è stato realizzato per la comunicazione con il client del modulo J-Manager. Per il Server localhost vengono implementate diverse funzionalità, ognuna di queste viene eseguita alla ricezione di una determinata stringa all'interno di un messaggio inviato da J-Manager. Questa stringa è all'interno di un attributo chiamato *Comando* e può avere i seguenti valori: "*Collegati*", "*Inviato_Output*", "*Richiesta_Input*", "*Terminato*".
Alla ricezione del comando *Collegati*, il Server istanzia un client che si collega al Server di J-Manager per stabilire una comunicazione bidirezionale, "*Server J-Communicator - Client J-Manager*" e "*Server J-Manager - Client J-Communicator*". Tale comunicazione permetterà di poter scambiare messaggi tra i due moduli. Il client di J-Communicator si collegherà alla porta specificata all'interno del messaggio che contiene la stringa *Collegati*.
Alla ricezione del comando *Inviato_Output*, J-Communicator esegue la funzione **Invia_Output_a_Clients** per inviare a tutti i Client connessi la stringa contenuta all'interno del messaggio ricevuto. Questa stringa contiene l'output generato dai programmi che deve essere inviata a tutti i Client che seguono quell'attività o quella lezione. Eseguita la funzione **Invia_Output_a_Clients** viene inviata una stringa a J-Manager affinché faccia continuare l'esecuzione del programma.
Alla ricezione del comando *Richiesta_Input*, J-Communicator invia un messaggio a tutti i Client per metterli a conoscenza che un programma richiede un input per continuare la sua esecuzione. I Client riceveranno questo testo e renderanno editabile l'area designata della GUI per l'invio di un input.
Alla ricezione del comando *Terminato*, J-Communicator invia un messaggio a tutti i Client per avvertirli che il programma è terminato, e quindi saranno invitati a chiudere il loro Terminal di simulazione del prompt.
La funzione **Avvia_Programma**, di tipo booleano, crea il processo del programma il cui nome è il valore della stringa *Nome* e avvia il processo ponendolo nello stato sospeso. Questa funzione restituisce il valore *True* se non è avvenuto alcun errore, diversamente restituirà *False*.
La funzione **Invia_Input**, di tipo booleano, ha come parametro di input un testo inviato dai Client Device e lo invia al processo in pausa. In questa funzione, J-Manager scrive il valore del testo ricevuto, *Testo*, nella pipe dello standard input del processo figlio, quindi se non è avvenuto alcun errore allora la funzione restituisce *True*, altrimenti restituisce *False*. Il processo sarà risvegliato dopo la terminazione di tale funzione.

A.2.4 Componente J-Manager

J-Manager è un componente sviluppato interamente in Pascal utilizzando l'IDE[1] Lazarus. Il suo funzionamento è simile ad un Terminal o ad una shell, con la differenza che invia i dati di visualizzazione e riceve i dati di input tramite socket e protocollo TCP. Ci si può domandare il perché è stato scelto un linguaggio di programmazione diverso da Javascript e C per implementare questa componente e perché è stato necessario crearla. La necessità consiste nel risolvere tre problemi, e le loro soluzioni rispondono alle precedenti domande. Il primo problema consiste nell'impossibilità di poter gestire adeguatamente la comunicazione utilizzando delle "pipe". Le pipe sono canali di comunicazione interprocesso monodirezionali utili allo scambio di dati e quindi di informazioni. Node.js non riesce a gestire adeguatamente il flusso di esecuzione di un programma utilizzando le pipe. Infatti, il flusso di esecuzione di un programma è asincrono tra gli output ed input. In un terminal generalmente il flusso di esecuzione di un programma appare ben definito nella sua esecuzione. Ciò è dovuto alla sincronizzazione delle pipe di input ed output del programma. Ogni programma, in qualunque sistema operativo, usa 3 pipe di comunicazione: uno per lo standard output, uno per lo standard input e l'ultimo per lo standard error. La pipe di output contiene i dati che il processo invia ad un altro processo, generalmente è una stringa. La pipe di input contiene i dati utili al programma per la sua corretta esecuzione, questi dati generalmente sono numeri o stringhe. La pipe di error contiene gli errori che il processo solleva durante la sua esecuzione, simile alla pipe di output ma contiene solo messaggi di errore. Node.js non riesce a sincronizzare correttamente la gestione delle tre pipe in maniera tale da rendere interattiva l'esecuzione dei programmi. Si è dunque reso necessario utilizzare un linguaggio di programmazione di basso livello per risolvere il problema, infatti la soluzione consiste nel riuscire a sospendere il processo del programma ogni qualvolta che il programma invia un messaggio di output e di errore, e richiede un input per continuare correttamente la sua esecuzione. Generalmente, i programmi che riescono a gestire correttamente l'esecuzione delle applicazioni sono i terminal o le shell a riga di comando fornite dai sistemi opertivi. I terminal o le shell sono applicazioni che permettono di eseguire altri programmi sincronizzandone il flusso delle loro pipe, creando una tecnica simile a quella di debugging, ovvero fermano l'esecuzione del programma quando questo invia un messaggio, di output o di errore, o richiede un input. Per eseguire questo approccio, i terminal o le shell (nel prosieguo per brevità si parlerà di terminal o shell in maniera indifferente per il solo scopo di esplicitazione) devono utilizzare alcune funzioni del sistema operativo messe a disposizione tramite chiamate di sistema di tipo API (Application Programmable Interfaces). Queste funzioni permettono di

[1] Integrated Development Environment

inviare segnali tra processi al fine di eseguire vari comportamenti. Tipicamente, un sistema operativo possiede funzioni per l'invio di segnali tra processi. Queste funzioni richiedono l'identificativo del processo univocamente assegnato dal sistema operativo, tipicamente un numero intero definito "*PID*" (Process IDentifier), ed un valore numerico che specifica il tipo di segnale inviato. Vi sono vari tipi di segnali che è possibile inviare tra processi, i più comuni sono i segnali di debugging, di pausa, di "wake" (ovvero di risveglio dallo stato, ad esempio, da una pausa), di stop, di kill, di interruzione e due segnali "*custom*" utili all'implementazione di segnali propri usati dagli sviluppatori. Quelli utili per i nostri scopi sono: due tipi di segnale custom chiamati segnale "*User1*" e "*User2*", che servono a distinguere il tipo di di dato, output se inviato o input se richiesto dal programma; il segnale di pausa che permette ad un processo di addormentarsi in attesa di essere svegliato e che permette di sospendere l'esecuzione del programma; il segnale di wake che sveglia il processo di un programma, se dormiente, per permettere ad esso di continuare la sua normale esecuzione; il segnale di kill che chiude il programma immediatamente a prescindere se il processo ha terminato la sua esecuzione o no. Il Pascal è un linguaggio di programmazione di basso livello che può agevolmente risolvere questo problema, infatti esso può effettuare le corrette chiamate di sistema per la corretta gestione dei segnali da ricevere ed inviare ai processi.
Il secondo problema consiste nel rendere il codice sorgente di J-Manager compilabile, ed eseguibile, in qualsiasi sistema operativo. Il linguaggio C permette di essere compilato in qualsiasi sistema operativo, tuttavia è necessario modificare alcune porzioni del codice sorgente quando si cerca di compilare questo codice sorgente su sistemi operativi diversi, es. Windows e Linux. Questo è dovuto alla naturale definizione delle funzioni del sistema operativo per la gestione dei segnali interprocesso che non risultano equivalenti, in termini implementativi e d'uso, tra i vari sistemi operativi. Lazarus risolve questo problema, in quanto è un applicativo eseguibile in tantissimi sistemi operativi, vedi il paragrafo 3.4.3, e può compilare facilmente il codice sorgente scritto in Pascal. Inoltre, Lazarus mette a disposizione una serie di funzioni per la corretta gestione dei segnali senza la necessità di modificare il codice sorgente. Il compilatore di Lazarus gestisce le condizioni di compilazione su sistemi operativi diversi, garantendo l'esecuzione del modulo J-Manager.
Il terzo problema consiste nell'instaurare un canale di comunicazione tra J-Communicator e J-Manager. La comunicazione tra questi due componenti del Server Device è fondamentale per una delle principali caratteristiche di SIRENE: permettere a tutti gli utenti di interagire con i programmi. Per fare questo è necessario che J-Communicator e J-Manager possano dialogare al fine di inviare i relativi output dei programmi ai Client Device e di ricevere tutti gli input dai Client Device e mandarli correttamente ai programmi attivi. Anche in questo caso sarebbe possibile utilizzare il linguaggio C per sviluppare J-Manager e permettergli

di comunicare con J-Communicator tramite socket con protocollo TCP, tuttavia anche in questo caso sarebbe necessario modificare porzioni di codice per rendere compilabile il codice sorgente, da un sistema operativo all'altro, a causa della non uniformità delle funzioni che i sistemi operativi mettono a disposizione tramite le proprie API. Lazarus risolve anche questo problema in quanto le librerie di *Indy Project* sono compilabili in tutti i sistemi operativi compatibili con *Lazarus*, vedi il paragrafo 3.4.3, ed i componenti "*TidTCPClient*" e "*TidTCPServer*" di *Indy Project* permettono di realizzare facilmente un client (usando il componente *TidTCPClient*) che possa collegarsi al server di comunicazione di J-Communicator e di realizzare un server (usando il componente *TidTCPServer*) che permetta al client, di comunicazione, di J-Communicator di dialogare con J-Manager, questo senza eseguire alcuna modifica del codice sorgente di J-Manager.
J-Manager gestisce l'output dei processi e l'invio degli input ai processi. Per sincronizzare il flusso degli standard output ed input dei programmi, ogni programma, che deve inviare un output, inizialmente deve scrivere nella pipe dello standard output e successivamente mandano un segnale con valore numerico 10 ("*SigUser1*", o segnale *User1*) a J-Manager, ed immediatamente ogni programma si pone in stato di pausa. Invece, quando un processo richiede un input, manda un segnale a J-Manager con valore numerico 12 ("*SigUser2*", o segnale *User2*) e si pone nello stato di pausa finché J-Manager non ha scritto nel pipe dello standard input del processo e risveglia il processo dormiente, quindi il processo puócontinuare la sua esecuzione prendendo il valore di input ricevuto.
In sintesi, quando un processo ha inviato un output questo si mette subito in stato di *Wait* e quando un processo deve ricevere un input, questo invia un segnale a J-Manager emettendo la richiesta di un input e subito si mette in stato di *Wait*. Questa metodologia garantisce che il flusso di esecuzione dei programmi sia corretto e che vi sia una corretta sincronia tra standard output ed input. J-Manager gestisce questo ciclo di pause e di segnali e realizza quello che può essere definito un "*Web-Terminal*" o una "*Web-Shell*".

A.2.5 Sviluppo di J-Manager

J-Manager usa tre attributi per la classe "*TForm*"[2] che implementa la finestra di visualizzazione dell'applicazione ed una variabile globale per la gestione di alcuni dati. I primi due attributi della classe *TForm* sono istanze del client del server (il primo di tipo *TidTCPClient* e nominato "*tcp*", e il secondo di tipo *TidTCPServer* e chiamato "*serv*" di J-Manager e permettono la comunicazione con J-Communicator. Il terzo attributo è un'istanza del processo, di tipo "*TProcessUTF8*" e chiamato "*processo*", che permette di eseguire e gestire

[2]In Lazarus, la classe *TForm* è la classe di base che implementa le finestre delle applicazioni.

i processi. La variabile globale è un'istanza di "*TCriticalSection*", una classe che implementa le sezioni critiche per evitare problemi di Deadlock e di sincronizzazione di eventi quando si usano risorse, gestite all'interno di questa sezione critica, da parte di Thread concorrenti. Vi sono sei funzioni globali dichiarate ed implementate **Ricevi_Output**, **Richiedi_Input**, **Gestisci_Fine_Processo**, **Crea_Gestore_Output**, **Crea_Gestore_Input**,
Crea_Gestore_Terminazione.
La funzione **Ricevi_Output** è una funzione che gestisce il segnale di tipo *SigUser1*, equivalente al valore numerico 10, e gestisce la ricezione di questo segnale da parte del processo figlio (il programma che J-Manager esegue). Questa funzione viene attivata quando il processo figlio ha scritto nella pipe dello standard output alcuni dati. In questa funzione, inizialmente, si controlla che il processo figlio sia in pausa, se non lo è si aspetta che esso entri in stato di pausa, quindi preleva il flusso dei dati dallo standard output del programma in esecuzione e lo invia, attraverso il *tcp*, al server di comunicazione di J-Communicator. Fatto questo, la funzione risveglia il processo figlio per permettergli di continuare la sua esecuzione.
La funzione **Richiedi_Input** è una funzione che gestisce il segnale di tipo *SigUser2*, equivalente al valore numerico 12, e gestisce la ricezione di questo segnale da parte del processo figlio. Questa funzione viene attivata quando il processo figlio chiede di ricevere un input da parte dei Client Device, in questo caso J-Manager, controlla sempre se il processo figlio è in pausa, altrimenti attende che esso sia in pausa e, quindi, invia un messaggio a J-Communicator, tramite il suo oggetto *tcp*, per indicargli di avvertire i Client Device che il programma è in attesa di input.
La funzione **Gestisci_Fine_Processo** è una funzione che gestisce il segnale di terminazione del programma, in questo caso, la terminzione del processo figlio. Questa funzione avviene quando il programma termina. Generalmente, il programma viene terminato direttamente da J-Manager, in quanto, attraverso l'uso dell'oggetto *processo*, il processo viene gestito interamente durante il suo ciclo di vita. Tuttavia, può capitare che per qualche motivo, il sistema operativo decide di terminare il programma, in questo caso, quest'ultimo invia un segnale a J-Manager che lo intercetta ed invia un messaggio a J-Communicator per avvertire i Client Device che il programma è stato terminato.
La funzione **Crea_Gestore_Output** è una funzione che imposta la funzione **Ricevi_Output** come unico gestore del segnale *SigUser1*, ovvero ogni segnale con valore numerico 10 viene intercettato da J-Manager e gestito da questa funzione.
La funzione **Crea_Gestore_Input** è una funzione che imposta la funzione **Richiedi_Input** come unico gestore del segnale *SigUser2*, ovvero ogni segnale con valore numerico 12 viene intercettato da J-Manager e gestito da questa funzione.
La funzione **Crea_Gestore_Terminazione** è una funzione che imposta la funzione **Richiedi_Input** come unico gestore del segnale di terminazione, ovvero ogni

segnale di terminazione viene intercettato da J-Manager e gestito da questa funzione.
La finestra dell'applicazione di J-Manager implementa un insieme di procedure e funzioni per la gestione del Server di J-Manager e la corretta gestione dei messaggi inviati a questo server. Le procedure sono: **ServerExecute**, **preleva**, **Form_Create**. Mentre le funzioni sono: **Avvia_Programma** e **Invia_Input**.
La procedura **ServerExecute** viene eseguita quando il Server di J-Manager riceve un flusso di dati dal client di comunicazione di J-Communicator. In questa funzione, inizialmente si entra all'interno della sezione critica, affinché non sia possibile per ulteriori thread in esecuzione accedere alla porzione di codice ed alle sue risorse contemporaneamente, e quindi si invoca la funzione **preleva**'. Alla fine si esce dalla sezione critica per permettere ad un eventuale altro thread di poter continuare la sua esecuzione.
La procedura **preleva** viene eseguita quando il server di J-Manager deve leggere i dati che il client di comunicazione di J-Communicator ha inviato. Questi dati sono stringhe in formato JSON. Questo messaggio ha il seguente formato

{
    Comando: *valore*,
    Parametro: *valore*
};

Il parametro *Comando* può assumere quattro valori: "*avvia*", "*Inviato_Input*", "*continua*", "*Termina_Programma*".
Se il comando è *avvia* allora il campo *Parametro* è chiamato "*nome_ programma_completo*" ed il suo valore sarà il nome del file compilato da J-Communicator. Quindi, J-Manager chiama la funzione **Avvia_Programma** passandogli come parametro il valore di *nome_programma_completo* ed avvia il processo (messo in stato di pausa immediatamente dalla funzione **Avvia_Programma**.
Se il comando è *Inviato_Input*, allora il campo *Parametro* è chiamato "*input*" ed il suo valore sarà il testo che i Client Device inviano a J-Communicator quando un programma chiede di ricevere un input. In questo caso, viene eseguita la funzione **Invia_Input** passandogli come parametro il valore di *input* ed infine J-Manager risveglia il processo che, in attesa di input, si era messo in stato di sonno.
Se il comando è *continua* allora significa che J-Communicator ha eseguito un invio dell'output di un programma e quindi, dopo aver inviato a tutti i Client Device il testo dell'output del programma, comunica a J-Manager di risvegliare il processo dormiente. In questo caso il messaggio non contiene il campo *Parametro*.
Se il comando è *Termina_Programma* allora significa che J-Communicator ha ricevuto l'ordine, da un Client Device, di terminare il processo. Questo può avvenire

quando un Client Device vuole interrompere l'esecuzione del programma per qualunque motivazione oppure perché il programma è terminato ma è posto in uno stato definito "*zombie*", ovvero il processo ha completato interamente la sua esecuzione e si pone in stato di attesa prima di essere concluso definitivamente. Quindi, J-Manager termina il processo. Anche in questo caso il messaggio non contiene il campo *Parametro*.
Infine, la procedura **FormCreate** viene chiamata quando J-Manager viene eseguito per la prima volta, quindi, J-Manager crea la sezione critica e la rende utilizzabile e setta i gestori dei segnali per gestire correttamente i segnali inviati dai processi figli.

A.3 Analisi del Server Device di SIRENE

Si sono analizzate le difficoltà implementative per la realizzazione del Server Device di SIRENE e le rispettive soluzioni sviluppate. Infine si è discussa l'implementazione e la logica delle funzioni principali del Server Device di SIRENE. Nell'appendice B si discuterà dell'implementazione del linguaggio iconico, nonché delle difficoltà di realizzazione e le loro soluzioni implementative.

Appendice B

Implementazione del linguaggio iconico di SIRENE

B.1 Sviluppo del linguaggio iconico di SIRENE

Il linguaggio concettuale di SIRENE è stato sviluppato utilizzando due linguaggi: HTML5 e Javascript. Gli elementi HTML sono elementi visualizzati da ogni browser. Le specifiche dell'HTML5 forniscono utili oggetti da cui partire per sviluppare il linguaggio iconico. Questi elementi si prestano bene ad essere modellati, anche visualmente utilizzando il Javascript e il CSS, per realizzare oggetti visuali che possano rappresentare gli elementi del linguaggio iconico.
Prima di definire lo sviluppo del linguaggio iconico è utile analizzare alcuni aspetti tecnici e alcune criticità dell'utilizzo del HTML e di Javascript.

B.2 Funzionalità e caratteristiche di HTML e Javascript

In generale, gli elementi HTML hanno una insieme di attributi, alcuni sono generici per tutti gli elementi HTML, altri invece sono specifici per alcuni elementi, ma tutti gli elementi permettono di definire attributi "*custom*", ovvero attributi che non esistono nelle specifiche di definizione dell'HTML ma possono essere definite dal programmatore semplicemente inserendole nel codice HTML come il seguente esempio

attributo_custom="*valore da inserire*"

Gli attributi *custom* degli elementi HTML permettono di memorizzare alcune informazioni utili ed importanti. Questi attributi, generalmente, vengono ignorati dai browser in quanto non essendo standard non è possibile prevedere il loro significato o il loro comportamento. Tuttavia, questi attributi sono spesso utili quando è necessario memorizzare informazioni che possono essere successivamente, ad esempio, utilizzare dal linguaggio Javascript. Javascript ha la capacità di interagire con gli elementi dell'HTML attraverso l'uso del DOM e può facilmente ottenere l'accesso ad un elemento HTML e manipolarlo modificando i valori dei suoi attributi. Inoltre, Javascript può aggiungere un attributo ad un qualsiasi oggetto HTML in due modi: utilizzando la funzione **setAttribute** che inserisce un attributo ed il suo valore all'interno del documento HTML, oppure scrivendo *oggetto.attributo_custom*, in questo caso l'attributo è aggiunto all'oggetto (nel modello DOM) ma non è visibile nel documento HTML, ovvero significa che il browser "ricorda" che è stato aggiunto questo attributo all'elemento ma non deve essere inserito nel documento HTML. Ad esempio, usando la seguente riga di codice in Javascript possiamo accedere ad un elemento HTML di nome "*nome_elemento*" usando la variabile "*elemento*":

```
var elemento = document.getElementById('nome_elemento');
```

Oppure, la riga successiva inserisce e valorizza nel documento HTML l'attributo "*nuovo_attributo1*", dell'elemento, con "*valore*", se l'attributo non esiste allora "*nuovo_Attributo1*" verrà creato e valorizzato:

```
elemento.setAttribute('nuovo_attributo1','valore');
```

Diversamente, la riga successiva inserisce e valorizza nel documento DOM, ma non nell'HTML, l'attributo "*nuovo_attributo2*", dell'elemento, con "*valore*", se l'attributo non esiste allora "*nuovo_Attributo2*" verrà creato e valorizzato:

```
elemento.nuovo_attributo2 = 'valore');
```

Per completezza, in Javascript si ottiene il valore dell'attributo "*nuovo_attributo1*" di un elemento HTML scrivendo:

```
elemento.getAttribute('nuovo_attributo1');
```

Javascript permette anche di aggiungere una nuovo attributo ad un elemento, o modificare un attributo di un elemento esistente, utilizzando la funzione "**defineProperty**" dell'oggetto "*Object*". La caratteristica di questa funzione è che permette di definire gli attributi in una modalità più avanzata e completa valorizzando alcune caratteristiche all'attributo: "*configurable*", "*enumerable*", "*writable*", "*value*". La caratteristica *configurable* controlla se l'attributo che si sta definendo può essere cancellato dall'oggetto e se le altre proprietà di definizione (*writable* e *value*) possono essere cambiate. Se *configurable* è impostato a *True* allora è possibile modificare i valori o eliminare l'attributo, altrimenti no.
La caratteristica *enumerable* verifica se l'attributo può essere visibile quando si usa il ciclo "*For..in*", di Javascript, oppure se l'attributo può essere visibile dalla funzione "**keys**" dell'oggetto *Object*. Se *enumerable* è impostato a *True* allora l'attributo è possibile essere visto, ed utilizzato, nei casi precedentemente detti.
La caratteristica *writable* permette, o non permette, di modificare il valore di un attributo. Se questa caratteristica è impostata a *True* è possibile modificare il valore di un attributo, diversamente se impostato a *False* allora l'attributo si comporterà come una costante per quell'elemento.
Oltre alla definizione di un attributo per mezzo delle sue proprietà, è possibile usare la funzione **defineProperty** utilizzando le sue due funzioni "**get**" e "**set**". Queste funzioni permettono di avere una gestione più flessibile quando si cerca di ottenere, o di cambiare, il valore di un attributo. Quando si cerca di ottenere il valore di un attributo, Javascript chiama automaticamente la funzione **get** che è stata definita, e quando si deve modificare il valore di un attributo, Javascript chiama automaticamente la funzione **set** per settarne il valore. Di seguito due esempi d'uso:

```
Object.defineProperty(obj, 'key',
{
        enumerable: False,
        configurable: False,
        writable: False,
        value: 'static'
});
```

In questo caso, per l'oggetto "*obj*" è stato definito l'attributo "*key*" : il valore è "*static*", non è enumerabile, non è configurabile e neanche cancellabile, e non è possibile modificare il suo valore.

```
Object.defineProperty(obj, 'key',
{
        get()
        {
                return chiave;
        },
        set(value)
        {
                chiave = value;
        }
});
```

In questo caso, per l'oggetto *obj* è stato definito l'attributo *key* in una modalità più flessibile: quando si cerca di ottenere il valore di *key*, viene chiamata la funzione **get** definita per l'attributo *key* e restituisce il valore di *chiave* (per brevità l'attributo *chiave* lo consideriamo già esistente nell'elemento), mentre se si vuole impostare il valore di *key*, allora verrà chiamata la funzione **set** definita per *key* nella quale il valore "*value*" viene memorizzato in chiave. In realtà, nella funzione **get** è possibile anche restituire un valore calcolato da un'espressione e non necessariamente un valore di qualche attributo già definito. Con le funzioni **get** e **set** è possibile definire attributi che in realtà non esistono, ma che sono il risultato di due chiamate di funzioni dentro le quali è possibile calcolare e restituire il valore di espressioni. Inoltre, non è necessario definire obbligatoriamente entrambe le funzioni, ma se ne può definire anche una soltanto, ad esempio: se si usa solo la funzione **get** nella definizione dell'attributo, allora si otterrà un attributo virtuale che fornisce soltanto sempre lo stesso valore (se i dati che usa **get** non variano), se invece si usa soltanto la funzione **set** allora si definisce un attributo che è solo possibile modificare (generalmente questa metodologia viene usata, nei linguaggi di programmazione in generale, per inserire valori che devono essere validati).
SIRENE fa largo uso della funzione **defineProperty** dell'oggetto *Object* in quanto ogni oggetto in SIRENE possiede un attributo, sia nel documento HTML sia nel DOM, chiamato "*Tipo_HTML*", non modificabile e neanche cancellabile, che definisce univocamente il tipo di oggetto e tutte le sue funzionalità, e come questo oggetto deve essere manipolato e gestito. Inoltre, questo attributo, essendo non modificabile e non cancellabile, realizza un alto grado di sicurezza e protezione da manipolazioni non consentite nel documento HTML modificando i dati dalle funzionalità implementate nei web browser.

B.3 Sirene Client Device Module e suoi oggetti

SIRENE Client Device Module è composto da numerosi file scritti in Javascript, alcuni in CSS ed uno soltanto in HTML5. Il file scritto in HTML, **Sirene.html**, realizza l'impostazione gerarchica degli oggetti, visibili e non, che verranno inseriti nel documento. In SIRENE ogni oggetto, ad eccezione di quelli nel file HTML, viene creato in Javascript ed inserito nel documento HTML in modalità dinamica, ovvero creato dinamicamente. Ogni oggetto ha un suo tipo che specifica in maniera univoca il tipo di oggetto, le modalità di funzionamento dell'oggetto, le modalità per ottenere l'accesso a quell'oggetto e le modalità di gestione dell'oggetto, nonché le modalità di visualizzazione di quell'oggetto. Questo tipo di dato viene memorizzato nell'attributo *custom* HTML e DOM *Tipo_HTML*. Questo è impostato all'inizio, ovvero quando viene istanziato l'oggetto, e reso immediatamente non scrivibile, non cancellabile, non modificabile, non enumerabile e non configurabile. È stato necessario creare questo attributo in quanto le pagine HTML possono essere facilmente modificabili usando alcune funzionalità dei web browser messe a disposizione per gli sviluppatori, oltre ad avere una corretta gestione e conoscenza di ogni singolo elemento, creato dinamicamente, all'interno del documento HTML. Queste funzionalità dei web browser per sviluppatori permettono di modificare facilmente il contenuto HTML di una pagina web. È quindi possibile che qualche utente malintenzionato voglia modificare alcuni comportamenti degli oggetti in una pagina web per diffondere a tutti i client un comportamento anomalo. SIRENE non permette questa possibilità. Il comportamento e la gestione degli oggetti avviene attraverso una perfetta corrispondenza dell'attributo *Tipo_HTML* e non viene permessa la modifica a livello DOM, infatti anche se un utente provasse a modificare il *Tipo_HTML* a livello HTML, al primo tentativo di utilizzo dell'oggetto SIRENE reimposta automaticamente il corretto valore dell'attributo poiché ogni oggetto viene controllato a livello DOM e, se il valore tra il modello DOM e quello HTML differisce, allora questo viene reimpostato al valore originario producendo una reinizializzazione dell'oggetto. Questo viene realizzato attraverso il corretto uso di due funzioni sviluppate in Javascript: "**Setta_Elemento**" e "**Prendi_Tipo_HTML**". Ogni volta che si crea un oggetto HTML esso viene subito passato come parametro alla funzione **Setta_Elemento** che rende permanenti il suo tipo, gli eventi e le funzionalità ad esso associati, mentre quando si vuole ottenere il tipo di oggetto HTML, per eseguire alcune azioni sull'oggetto e sul sistema, si passa l'elemento HTML alla funzione **Prendi_Tipo_HTML** che recupera il corretto valore del tipo di oggetto e lo setta nuovamente, se necessario. Inoltre, quando un utente inizia a seguire una lezione già avviata, SIRENE recupera tutto l'HTML della lezione e imposta il corretto **Tipo_HTML** per tutti gli oggetti della lezione rendendoli permanenti. Non è dunque possibile modificare in alcun modo il tipo di oggetto e le loro funzionalità e gestioni. La funzione

Setta_Elemento è di notevole importanza in quanto risolve anche il seguente problema: quando un utente si collega (o ricollega) per la prima volta ad una lezione, egli riceve tutto il codice HTML aggiornato della lezione, e riceve anche gli oggetti *Canvas*. Questi oggetti per essere visualizzati devono essere "disegnati", infatti non basta semplicemente inserire l'oggetto nella pagina web ma essi devono essere necessariamente disegnati tramite l'uso di Javascript in ogni browser. La funzione **Setta_Elemento** risolve questo problema, e nel paragrafo B.8 se ne discuterà la soluzione.
La maggioranza degli oggetti utilizzati in SIRENE sono composti da 3 tipi di elementi visuali HTML: *div*, *span* e *Canvas*. Gli elementi *div* si prestano bene a realizzare contenitori di altri oggetti HTML, mentre gli elementi di tipo *span* vengono utilizzati anche come contenitori ma come elementi che possono essere accostati, nella stessa posizione orizzontale, ad altri elementi (diversamente dagli oggetti *div* i quali si spostano verticalmente, nel loro comportamento grafico di default, e non si accostano ad altri elementi HTML). Gli oggetti *Canvas* invece possono essere manipolati graficamente, realizzando grafici anche complessi, tramite le funzioni Javascript. Ad ogni oggetto creato in SIRENE viene impostato l'attributo *id* in maniera univoca. Generalmente in SIRENE, l'*id* è una stringa realizzata per concatenazione da 3 elementi: il *Tipo_HTML*, il momento di quando viene realizzata l'azione in SIRENE e il nome dell'utente che realizza l'azione di creazione, dove il momento è il valore del tempo, in formato timestamp, del server di SIRENE di quando riceve il messaggio di creazione di un oggetto (ad esempio un nuovo progetto). In questa maniera, ogni Client Device può autonomamente creare lo stesso oggetto con lo stesso *id*, oltre che con gli stessi attributi. Questo permette a tutti i client di SIRENE di poter eseguire le stesse azioni sugli stessi oggetti contemporaneamente e di ottenere tramite Javascript l'accesso allo stesso elemento grazie all'*id*.
Generalmente, negli oggetti HTML l'attributo *classname* definisce la classe di formattazione (visualizzazione) dello stile grafico dell'oggetto. In SIRENE il tipo di oggetto e il tipo di *classname* tipicamente coincidono. La valorizzazione dell'attributo *classname* non ha effetto negli elementi *Canvas*, poiché questi vengono normalmente disegnati tramite l'uso di Javascript, che disegna l'oggetto *Canvas* manipolando il suo "*contesto*". Questa caratteristica non dipende da SIRENE. Per utilizzare i valori delle classi CSS negli oggetti *Canvas* è possibile utilizzare una funzione Javascript chiamata "**getComputedStyle**" che ricevere come parametro l'elemento *Canvas* HTML e ottiene tutti gli attributi valorizzati nella classe CSS sotto forma di attributi di un oggetto stesso. Ad esempio

```
var style = getComputedStyle(elemento);
```

in questo caso, *style* è l'oggetto che ricevere tutti gli attributi della classe dell'elemento HTML *elemento*. Si potrà accedere ai singoli attributi valorizzati come nel seguente esempio:

```
width = parseInt(style.width);
```

così facendo è possibile modificare, parzialmente, lo stile grafico di una Canvas tramite CSS.
Il file che contiene tutta la struttura HTML della pagina web di Sirene è **Sirene.html**.
Mentre i files Javascript che realizzano le funzionalità del Client Device sono 15:

- Eventi.js
- Azioni.js
- Rete.js
- Gestione_Forme.js
- Forme.js
- Gestione_Assegnazioni.js
- Gestione_Codice.js
- Gestione_Espressioni.js
- Gestione_For.js
- Gestione_Progetto.js
- Gestione_Scheda_Array.js
- Gestione_Schede.js
- Gestione_Terminale.js
- Menu.js
- Comuni.js

Invece, i files CSS che contengono lo stile di ogni oggetto sono 8:

- Assegnazione.CSS
- Comune.CSS
- Espressione.CSS
- Forme.CSS
- Gestione_For.CSS
- Menu.CSS
- Progetto.CSS
- Schede.CSS

Nel seguito del capitolo si discuterà dei file, e delle loro funzioni più importanti, che compongono il Client Device di SIRENE.

B.4 Sirene.html e la gerarchia HTML di visualizzazione

Il file **Sirene.html** contiene tutta l'intera ed unica pagina web in cui SIRENE viene eseguito. Questa pagina descrive la gerarchia degli oggetti HTML. Il file contiene la lista di file da cui dipende, sia files Javascript sia CSS, e presenta anche la dipendenza dal file **Socket.io.js** che è generato dai frameworks Node.js, Express.io e Socket.io del Server Device di SIRENE e contiene tutte le funzionalità di comunicazione websocket.
Il corpo *body* del file **Sirene.html** è la radice di tutti gli elementi visuali della pagina ed ha due eventi: "*onload*" e "*onmouseup*", valorizzati rispettivamente con le funzioni "*Inizializza_Tutto*" e "*Azione_Window*". Queste due funzioni sono fondamentali per il corretto funzionamento di alcune caratteristiche. Si discuterà di queste due funzioni in dettaglio nel paragrafo B.5 dedicato al file **Eventi.js**. La funzione **Inizializza_Tutto** è sempre chiamata ogni volta, e soltanto una volta, che si accede a SIRENE e viene rimossa subito dopo essere stata chiamata. Diversamente, la funzione **Azione_Window** non viene mai rimossa ma mantenuta in quanto serve per azionare alcuni eventi *custom* non esistenti nell'HTML.
Il file contiene anche una gerarchia di librerie e di funzioni del linguaggio C. La radice di questa libreria è l'oggetto *Librerie* e contiene il suo *id* univoco (tutti gli attributi *id* di ogni oggetto HTML è univoco in SIRENE) e il suo stile CSS settato in "invisibile", cosicché nel caso in cui qualche browser dovesse visualizzare questo elemento, e i suoi figli, non gli sarà permesso. *Librerie* è una collezione

di oggetti *Libreria*. Ogni *Libreria* ha diversi attributi: un proprio *id*, un proprio nome della libreria *Nome* valorizzato con il nome di una libreria standard C, ad esempio *stdio*; due attributi *Tempo* e *Proprietario* che non sono valorizzati ma sono inseriti per compatibilità di implementazioni successive con le librerie *custom* realizzate con il linguaggio iconico di SIRENE, ed un attributo *Standard* settato ad 1 per indicare che la libreria è una libreria standard del C, per le librerie *custom* (ovvero quelle realizzare con SIRENE) questo attributo è settato a 0. Ogni *Libreria* ha un figlio chiamato *Funzioni* il quale è un insieme di oggetti *Funzione*. Ogni *Funzione* possiede un proprio *id*, un proprio *Nome* (ad esempio *printf*), un proprio *tipo*, che rappresenta il tipo di dato restituito dalla funzione ad esempio *int*, ed un attributo *Tipo_Terminale* il quale rappresenta se il tipo di funzione è una funzione di output, di input oppure nessuna delle due precedenti. *Funzione* contiene un figlio chiamato *Parametri* ed ha un attributo *id*, un attributo *numero_obbligatorio* il quale è valorizzato con un valore intero non negativo ed indica il numero di parametri obbligatori da inserire durante lo sviluppo del codice iconico, un attributo *numero_facoltativo* il quale rappresenta il numero facoltativo di parametri da dover inserire e può essere valorizzato da un valore numerico che va da *-1* ad un numero positivo intero, questo attributo indica il numero facoltativo da poter inserire: se il suo valore è *-1* significa che possono essere inseriti infiniti parametri (un caso è il printf che, teoricamente, può ricevere un numero indefinito di parametri a seconda del contenuto della stringa che viene inserita all'inizio), invece se è valorizzato con un valore da *0* in poi, potrà contenere un numero di parametri facoltativi pari a questo valore). *Parametri* è una collezione di oggetti *Parametro*. Ogni *Parametro* ha un *id*, un *Nome* che indica il nome del parametro, un *Tipo* che identifica il tipo di dato, ad esempio *void*, ed un attributo *Riferimento* che indica se quel parametro deve essere passato per valore o come riferimento: nel caso in cui questo attributo sia valorizzato con *0* allora il parametro è passato per valore, diversamente se è valorizzato con *1* allora significa che il parametro viene passato per riferimento e si inserirà automaticamente il simbolo "*&*" nel codice sorgente quando viene chiamata la funzione durante l'uso del linguaggio visuale, ovvero quando viene creata una funzione o assegnazione, vedi i paragrafi C.5.1 e C.5.2. La gerarchia di *Librerie* è la seguente:

Librerie (id, style)
  libreria (id, Nome, Tempo, Proprietario, Standard)
    Funzioni (id)
      Funzione (id, Nome, Tipo, Tipo_Terminale)
        Parametri (id, numero_obbligatorio, numero_facoltativo)
          Parametro (id, Nome, Tipo, Riferimento)

tra le parentesi sono indicati gli attributi degli elementi.
All'interno del file HTML di SIRENE, è presente un classico menù per la creazione, apertura, salvataggio e chiusura dei progetti, file e funzioni, nonché di un'azione di compilazione ed esecuzione del codice sorgente testuale generato. Questo menù è un oggetto *div* HTML, con attributo *id* valorizzato a '*Contenitore_Menu*' e contiene un insieme di elementi che costituiscono il menù nella sua interezza.
Il menù è contenuto in un importante oggetto *div* che ha un *id* valorizzato a "*Contenitore*". Questo contenitore racchiude ogni struttura grafica di SIRENE ed è importante perché è un punto di riferimento per il framework, infatti esso possiede alcuni attributi che vengono valorizzati durante l'uso di SIRENE e forniscono informazioni per la corretta gestione del sistema. Questi attributi sono: l'attributo "*Progetto_Attivo*", il quale è dinamicamente valorizzato con l'*id* del Progetto che si sta sviluppando; l'attributo "*File_Attivo*", il quale è valorizzato con l'*id* del file che si sta manipolando; l'attributo "*Funzione_Attiva*", il quale è valorizzato con l'*id* della funzione che si sta implementando; infine due attributi che identificano due oggetti da nascondere, generalmente alcune finestre o vari menù aperti durante lo sviluppo del codice iconico e che necessitano di essere memorizzati per essere gestiti correttamente.
L'altro oggetto visuale importante, e che è contenuto in *Contenitore*, è un *div* ed ha l'*id* valorizzato a "*Contenitore_Progetti*" e la sua funzione è quella di contenere tutti i progetti da sviluppare. *Contenitore_Progetti* racchiude una collezione di elementi aventi l'attributo *Tipo_HTML* valorizzato a "*Contenitore_Progetto*", ciascuno di questi elementi contiene un intero progetto sia visuale che informativo. Il file **Sirene.html** inizialmente contiene solo gli elementi fin qui descritti, per completezza, della descrizione di tutta la gerarchia degli elementi HTML della pagina, si descriverà in questo paragrafo lo sviluppo completo di una pagina in SIRENE. Durante la fase di sviluppo, *Contenitore_Progetto* racchiude molti elementi HTML annidati per costituire la GUI di SIRENE. Questi elementi HTML costituiscono un progetto. La gerarchia di un *Contenitore_Progetto* è la seguente:

```
Contenitore_Progetto
├── Progetto
├── Contenitore_Schede_File
├── Contenitore_View_Progetto
└── Contenitore_Corpo_File
```

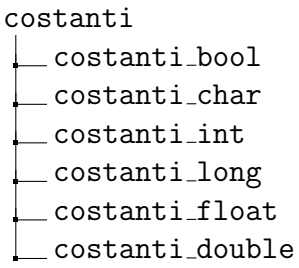
Ogni oggetto *Progetto* di un programma in SIRENE consiste di due componenti: una di tipo informativo di tutta la struttura del programma e una di tipo visuale che costituisce l'interfaccia Uomo-Macchina e permette la manipolazione di tutte le informazioni del progetto. Il progetto, dal punto di vista informativo,

consiste in una serie di elementi inseriti con un preciso tag che corrisponde, ciascuno, all'oggetto stesso, ovvero: il tag di ogni progetto è *progetto*, il tag dei files è *files*, il tag di un file è *file*, il tag di una funzione è *funzione*, e così via. Di seguito la completa gerarchia di un progetto nella sua struttura informativa:

```
progetto
└── files
    └── file
        ├── funzioni
        │   └── funzione
        │       ├── parametri
        │       │   └── parametro
        │       ├── variabili
        │       ├── costanti
        │       ├── enumerazioni
        │       └── strutture
        ├── variabili
        ├── costanti
        ├── enumerazioni
        ├── strutture
        └── utenti
```

Inoltre, gli oggetti *variabili* e *costanti*, degli oggetti *file* e *funzione*, possiedono ulteriori discendenti. La loro struttura informativa è la seguente:

```
variabili
├── variabili_bool
├── variabili_char
├── variabili_int
│   └── dato
├── variabili_long
├── variabili_float
└── variabili_double
```



È possibile avere un numero indefinito di elementi di tipo *parametro* all'interno degli elementi *parametri*, e di elementi *dato* sia nelle variabili sia nelle costanti. Nel seguito invece vi è la rappresentazione di ogni oggetto con gli attributi in esso inseriti tra parentesi:

progetto (id, nome, tempo, utente, modalita, ultimo_tempo, ultimo_utente, tipo_HTML)
files (id, progetto_id, tipo_HTML)
file (id, nome, tempo, proprietario, ultimo_tempo, ultimo_utente, progetto_id, standard, esterna, tipo_HTML, includes)
funzioni (id, , file_id, tipo_HTML)
funzione (id, nome, tipo, tempo, proprietario, ultimo_tempo, ultimo_utente, progetto_id, file_id, esterna, tipo_terminale, codice_visuale_HTML, tipo_HTML, includes, codice_testo_HTML)
parametri (id, funzione_id, tipo_HTML)
parametro (id, funzione_id, nome, tipo, dimensione, referenza, array, tipo_HTML)
variabili (id, funzione_id, tipo_HTML)
variabili_bool (id, funzione_id, tipo_HTML, dato)
variabili_char (id, funzione_id, tipo_HTML, dato)
variabili_int (id, funzione_id, tipo_HTML, dato)
dato (id, tempo, utente, funzione_id, dato, tipo, dimensione, nome, valore, array, referenza)
variabili_long (id, funzione_id, tipo_HTML, dato)
variabili_float (id, funzione_id, tipo_HTML, dato)
variabili_double (id, funzione_id, tipo_HTML, dato)
costanti (id, funzione_id, tipo_HTML)
costanti_bool (id, funzione_id, tipo_HTML, dato)
costanti_char (id, funzione_id, tipo_HTML, dato)
costanti_int (id, funzione_id, tipo_HTML, dato)
costanti_long (id, funzione_id, tipo_HTML, dato)
costanti_float (id, funzione_id, tipo_HTML, dato)
costanti_double (id, funzione_id, tipo_HTML, dato)

enumerazioni (id, funzione_id, tipo_HTML)
strutture (id, funzione_id, tipo_HTML)
variabili (id, file_id, tipo_HTML)
variabili_bool (id, file_id, tipo_HTML, dato)
variabili_char (id, file_id, tipo_HTML, dato)
variabili_int (id, file_id, tipo_HTML, dato)
dato (id, tempo, utente, file_id, dato, tipo, dimensione, nome, valore, array, referenza)
variabili_long (id, file_id, tipo_HTML, dato)
variabili_float (id, file_id, tipo_HTML, dato)
variabili_double (id, file_id, tipo_HTML, dato)
costanti (id, file_id, tipo_HTML)
costanti_bool (id, file_id, tipo_HTML, dato)
costanti_char (id, file_id, tipo_HTML, dato)
costanti_int (id, file_id, tipo_HTML, dato)
costanti_long (id, file_id, tipo_HTML, dato)
costanti_float (id, file_id, tipo_HTML, dato)
costanti_double (id, file_id, tipo_HTML, dato)
enumerazioni (id, file_id, tipo_HTML)
strutture (id, file_id, tipo_HTML)
utenti (id, progetto_id, tipo_HTML)

Ogni elemento della gerarchia di un progetto ha un *id* univoco, un *nome* assegnato dall'utente, un *Tipo_HTML* che identifica il tipo di oggetto HTML. L'attributo *progetto_id* identifica l'*id* del progetto; l'attributo *file_id* identifica l'*id* del file; l'attributo *funzione_id* identifica l'*id* della funzione. Molti elementi hanno anche un attributo *tempo* che identifica il momento temporale di creazione e l'attributo *proprietario*, ovvero il nome dell'utente che ha creato quell'oggetto (progetto, file o funzione). Gli attributi *ultimo_tempo* e *ultimo_utente* identificano rispettivamente il momento temporale dell'ultima modifica eseguita e l'utente che l'ha fatta. Gli elementi *file* e *funzione* hanno l'attributo *esterna* per indicare se il file, o funzione, devono essere dichiarati esterni e quindi è necessario creare il file con estensione "*.h*" per indicare gli "headers" delle funzioni. L'attributo *includes* si trova sia nell'elemento *files* sia nell'oggetto *funzione* ed indica le dipendenze da altri files, nello specifico: questo attributo in *funzione* indica che l'implementazione delle istruzioni inserite, durante la creazione iconica, risiedono in altri files; diversamente questo attributo in *file* indica tutte le dipendenze da tutti i files che devono essere inserite negli include del codice sorgente. L'attributo *standard* è inserito solo nell'elemento *file* per identificare che il file non appartiene alla libreria standard del linguaggio C, ma è stato sviluppato utilizzando il linguaggio iconico

di SIRENE. L'attributo *tipo_terminale* dell'elemento *funzione* è già stato discusso precedentemente in questo capitolo. Gli attributi *codice_visuale_HTML* e *codice_testo_HTML* forniscono il codice, rispettivamente, HTML e testuale. L'attributo *dato* negli elementi *variabili_bool*, *variabili_char*, *variabili_int*, *variabili_long*, *variabili_float*, *variabili_double*, *costanti_bool*, *costanti_char*, *costanti_int*, *costanti_long*, *costanti_float*, *costanti_double* identifica il tipo di dato: bool, char, int, long, float e double. Gli attributi *array*, *dimensione*, *tipo* e *referenza* negli elementi *parametro* e *dato* indicano, rispettivamente, se l'elemento è un array e, in caso affermativo, la sua dimensione, il tipo (bool, char, ecc). Invece, l'attributo *valore* indica il valore assegnato alla variabile o costante in fase di dichiarazione.
Il progetto, dal punto di vista della GUI, è invece composto da una serie di elementi HTML *div* e *span* e *Canvas* annidati, che sono all'interno dei restanti elementi di *Contenitore_Progetto*: *Contenitore_Schede_File*, *Contenitore_Schede_File* e *Contenitore_Schede_File*.
Contenitore_Schede_File assume la seguente gerarchia HTML:

```
Contenitore_Schede_File
└── Contenuto_Schede_File
    ├── Tab_Scheda_Progetto_View
    └── Tab_Scheda_File
```

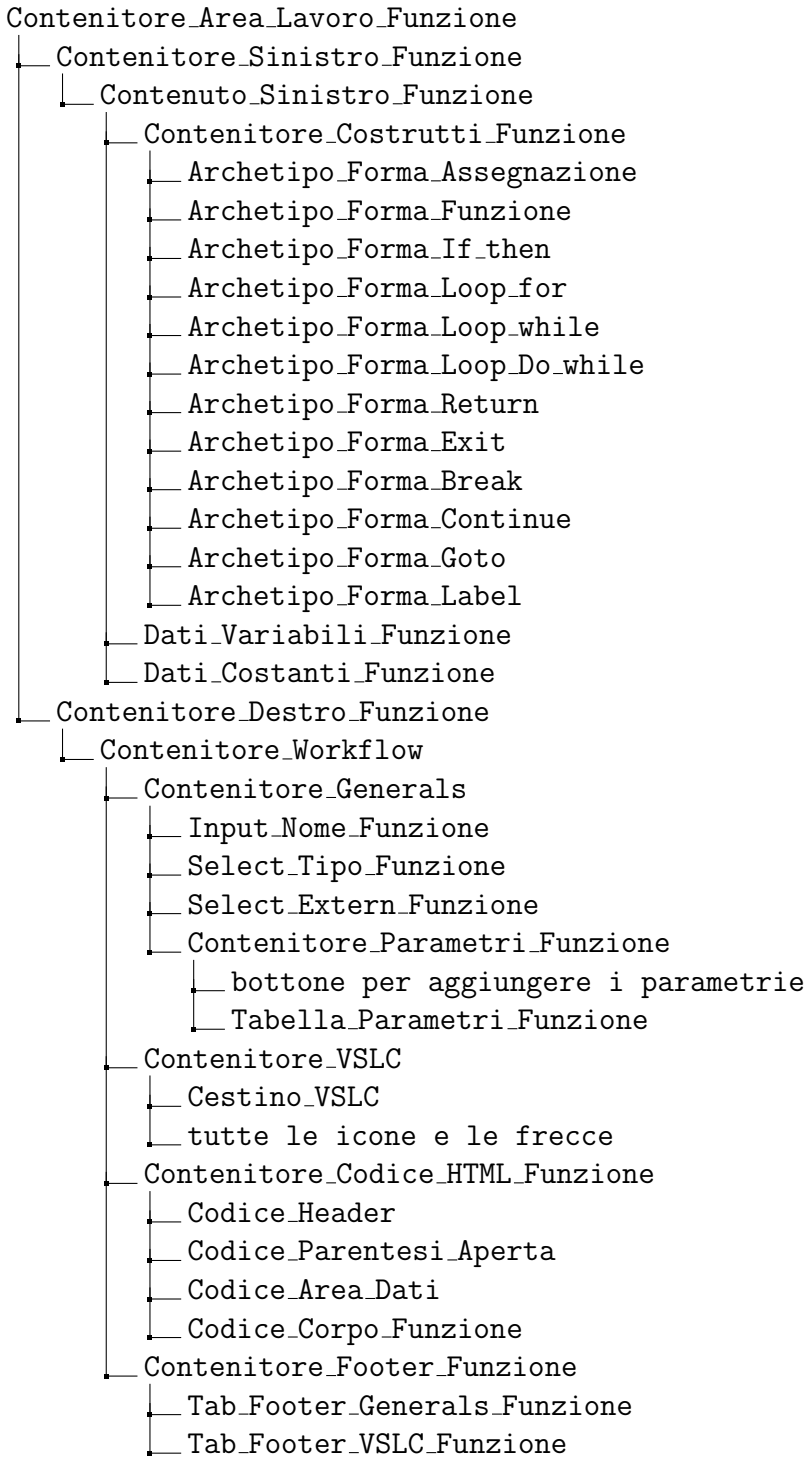
Il *Contenitore_Schede_File* contiene un insieme di tab e permette di navigare tra i files creati iconicamente. Il primo Tab, di ogni elemento *Progetto*, permette di visualizzare l'oggetto *Contenitore_View_Progetto* ed è sempre creato quando si crea un nuovo progetto. Gli altri tab vengono creati quando si crea un nuovo file: ogni tab visualizza il corrispondente oggetto *Contenitore_Corpo_File*. L'oggetto *Contenitore_View_Progetto* contiene due elementi importanti ed ha una semplice gerarchia HTML:

```
Contenitore_View_Progetto
└── Contenitore_File_View_File_list
    ├── Input_Nome_Contenitore_View_Progetto
    └── Tabella_Lista_File_Progetto
```

Questo contenitore *div* permette di inserire, o modificare, il nome del progetto editando l'elemento *Input_Nome_Contenitore_View_Progetto*, ed inoltre permette di vedere tutti i files creati nella tabella HTML *Tabella_Lista_File_ Progetto*.
Contenitore_Corpo_File è un elemento visuale molto più complesso e comprende 3 elementi:

```
Contenitore_Corpo_File
├── Contenitore_Schede_Funzione
├── Contenitore_File_View_Lavoro
└── Contenitore_Area_Lavoro_Funzione
```

Contenitore_Corpo_File contiene un contenitore di schede dei files: l'oggetto *Contenitore_Schede_Funzione* in cui vi è una serie di tab, ogni tab permette di visualizzare un'area di lavoro *Contenitore_Area_Lavoro_Funzione*. Mentre, il primo Tab di *Contenitore_Schede_Funzione* permette di visualizzare il contenuto di *Contenitore_File_View_Lavoro*. Quest'ultimo elemento contiene altri due contenitori: il primo che contiene una serie di bottoni HTML, i quali permettono di mostrare un oggetto che contiene variabili e costanti, dichiarati all'interno del file; il secondo contenitore permette di definire il nome del file e di visualizzare tutte le funzioni create all'interno di quel file, oltre a tutto il codice sorgente, in linguaggio C, del file stesso.
Contenitore_Area_Lavoro_Funzione è un elemento HTML *div* che contiene l'area di lavoro in cui si sviluppa utilizzando il linguaggio iconico. La sua gerarchia è:



Contenuto_Sinistro_Funzione contiene tutte le icone del linguaggio iconico ed

un insieme di pulsanti per inserire variabili e costanti all'interno della funzione.
In *Contenitore_Generals* sono contenuti tutti gli elementi HTML che permettono la definizione della funzione, come il suo nome, il suo tipo, la possibilità di rendere una funzione utilizzabile all'esterno del file ed i parametri della funzione.
Invece, in *Contenitore_VSLC* si realizza il comportamento logico delle funzioni utilizzando le icone del linguaggio iconico. Le icone vengono inserite in quest'area tramite il meccanismo del "drag&drop".
Il contenitore *Contenitore_Codice_HTML_Funzione* visualizza il codice sorgente associato ad ogni icona che è nel flusso dell'algoritmo e, nella totalità, di tutta la funzione.
L'ultimo contenitore, il *Contenitore_Footer_Funzione* contiene due tab che permettono di visualizzare, rispettivamente, il *Contenitore_Generals* e *Contenitore_VSLC*.

B.5 File Eventi.js

Questo file contiene un insieme di alcune funzioni utili utilizzate durante le azioni eseguite usando SIRENE ed il linguaggio iconico. La prima funzione che il file **Sirene.html** esegue è la funzione **Inizializza_Tutto** e viene eseguita quando il file HTML è caricato completamente. All'avvio di questa funzione, la funzione **Inizializza_Tutto** viene rimossa dal codice HTML, in quanto è necessario chiamarla solo una volta (all'avvio). Questo permette a SIRENE di rendere il suo codice non facilmente comprensibile. La funzione **Inizializza_Tutto** inizializza il socket di comunicazione tra il Client Device e il Server Device chiamando la funzione **Inizializza_Socket** del file **Rete.js**, e successivamente viene chiamata la funzione **Ottieni_HTML**, anch'essa definita in **Rete.js**, infine viene impostato l'attributo *Tipo_HTML* di ogni singolo elemento HTML del codice ricevuto, chiamando la funzione **Setta_HTML** del file **Gestione_Progetto.js**. Queste tre funzioni saranno esaminate in dettaglio successivamente. In questa maniera ogni utente potrà seguire la lezione in maniera corretta, visualizzando correttamente tutti gli elementi che gli altri utenti visualizzano ed ottenendo un codice HTML privo di tentativi malevoli di modificare i comportamenti degli oggetti HTML.
In questo file vi sono alcune funzioni importanti per lo sviluppo di un algoritmo iconico: **drag_Archetipi**, **allowDrop_Archetipi** e **drop_Archetipi**. Queste funzioni vengono eseguite a livello locale nel Client Device, ed implementano il meccanismo del "drag&drop": soltanto alla fine della funzione **drop_Archetipi** viene inviato un messaggio di sincronizzazione a tutti gli altri Client Device. In questo messaggio vengono inseriti tutti gli elementi utili alla ricostruzione dell'icona, trascinata e rilasciata nella Visual Code Area, da creare e del tipo di icona e la sua posizione in cui deve essere creata. Questo messaggio viene mandato anche all'utente che ha inviato il messaggio, in quanto la ricezione del messaggio stesso

permette di eseguire l'azione.
Vi sono altre 3 funzioni importanti per lo spostamento delle icone nella Visual Code Area: **startDrag**, **dragDiv** e **stopDrag**. Queste implementano il meccanismo di spostamento delle icone all'interno della Visual Code Area, mediante un sistema simile al "drag&drop". Anche queste funzioni agiscono a livello locale e soltanto la funzione **stopDrag** invia un messaggio di sincronizzazione a tutti i Client Device per eseguire la stessa azione. Inoltre, la funzione **startDrag** verifica se l'utente ha cliccato sopra un'icona con il tasto destro del mouse: se questo è vero si inibisce la visualizzazione del context menù (il meù di default che ogni browser visualizza quando un utente clicca il tasto destro del mouse) e crea e visualizza il menù di azioni per il collegamento con le altre icone. Questo viene realizzato mediante un messaggio di sincronizzazione inviato a tutti i i Client Device, incluso il Client Device che espleta l'azione, affinché tutti creino e realizzino la stessa visualizzazione. In questa maniera, ogni Client Device che segue la lezione visualizza un menù a tendina per l'esecuzione di un collegamento.
Una funzione interessante è la funzione **Azione_Window** che viene invocata ogni volta che un utente clicca sul browser. Questa funzione inizialmente serviva per nascondere alcune finestre dinamiche, successivamente si è voluto utilizzarla per realizzare un insieme di funzionalità utili che vengono invocate in seguito ad un evento *custom* realizzato e non esistente nell'HTML, ed è eseguito soltanto per ogni elemento visuale che abbia l'attributo *custom onazione_click*. Nella pratica, la funzione verifica se l'elemento che ha perso il focus o lo ha ricevuto, possiede questo attributo: se questa condizione è vera allora viene eseguita la funzione associata all'attributo, altrimenti non esegue niente.
Un'altra funzione che necessita di essere discussa è la funzione **Metti_in_Top Most**. Questa funzione è stata realizzata per risolvere il problema della mancanza di una funzione nell'HTML e nel Javascript che permette di porre un'icona visuale sopra le altre icone. Le icone quando vengono cliccate necessitano di essere posizionate davanti alle altre per essere visualizzate dagli utenti. In Javascript e nell'HTML non esiste una funzione che permetta questa azione, a meno che non si utilizzi la proprietà *z-index* del CSS ma sarebbe eccessivamente complicato ed eccessivamente oneroso, in termini prestazionali, aggiornare tutti gli elementi visuali nel documento HTML. Tuttavia, questa azione può essere espletata facilmente usando Javascript ed HTML nella seguente condizione: sia l'oggetto cliccato chiamato *elemento* e sia il padre di questo oggetto chiamato *padre* allora se *elemento* è l'unico figlio di *padre* allora *elemento* è già visualizzato correttamente. Diversamente se *padre* ha tanti figli, oltre ad *elemento*, allora per porre *elemento* visualizzabile sopra tutti gli altri oggetti si deve porre semplicemente l'*elemento* in fondo alla lista della gerarchia dei figli, ovvero si pone come ultimo figlio. Questa funzione consiste in una sola riga di codice:

elemento.parentElement.appendChild(*elemento*)

B.6 File Azioni.js

Questo file contiene la quasi totalità delle funzioni che realizzano i messaggi di sincronizzazione da inviare al Server Device ed a tutti i Client Device. Tutti i messaggi di sincronizzazione hanno la forma descritta nel paragrafo A.2.2. Ogni funzione di questo file descrive un *Evento* per una generica azione, comando oppure un evento specifico, per il Server Device. Per ogni azione o comando, il Server Device invia a tutti i Client Device l'oggetto *Messaggio* contenuto nel messaggio di sincronizzazione, il quale contiene, a sua volta, un insieme di elementi chiamato *Parametri*. Questi elementi sono indispensabili per la corretta esecuzione della stessa azione da parte di tutti i Client Device. Inoltre, l'oggetto *Messaggio* contiene l'elemento *Funzione* che corrisponde al nome della funzione che deve essere eseguita dal Client Device quando riceve il messaggio. Questa funzionalità è stata sviluppata in maniera molto semplice e snella al fine di evitare di sovraccaricare il Server Device, sia nell'invio dei dati sia nel carico di lavoro per il processore, ed inoltre evita di dover implementare una funzionalità di string matching per ogni funzione descritta nel parametro *Funzione*, e che deve essere eseguita dal Client Device in occasione della ricezione di ogni messaggio di sincronizzazione. Per l'invio del messaggio di sincronizzazione, ogni funzione chiama le funzioni **Invia_Azione** ed **Invia_Comando** implementati nel file **Rete.js**, mentre il Client Device implementa nel file **Rete.js** la funzione di ricezione del messaggio e la chiamata automatica della funzione da eseguire, descritta all'interno del messaggio ricevuto.

B.7 File Rete.js

Questo file è uno dei più importanti per realizzare la comunicazione tra i Client Device e il Server Device, ed indirettamente la comunicazione tra un Client Device con tutti gli altri Client Device. Le funzioni sono semplici ma molto adattabili a molteplici funzionalità. La prima funzione è **Inizializza_Socket** nella quale si inizializza il web socket di comunicazione con il Server Device e, successivamente, imposta due importanti, ma semplici, funzioni, che vengono eseguite quando si riceve una determinata stringa che funziona da evento per il socket: *azione* e *comando*. Queste due funzioni, che ricevono i messaggi di sincronizzazione dal Server Device, chiamano tutte le funzioni relative ai messaggi di sincronizzazione,

ed il loro contenuto più significativo è la seguente riga di codice:

```
window[data.Funzione](data.Parametri);
```

dove *data.Funzione* è l'elemento *Funzione* inserito nel messaggio di sincronizzazione e corrisponde al nome della funzione che i Client Device devono eseguire, e *data.Parametri* corrisponde ai *Parametri* inseriti nel messaggio di sincronizzazione e corrisponde ai parametri che la funzione necessita per essere eseguita; mentre *data* è il messaggio di sincronizzazione nella sua interezza, vedi il paragrafo A.2.2. La semplicità di questa sola riga di codice realizza un ipotetico inserimento di un costrutto *switch..case* in Javascript, con un numero di *case* che corrisponde al numero delle funzioni implementate, ed inoltre usando l'oggetto *window* si evita il rischio di possibili dimenticanze di inserimento di altri *case* (anche per sviluppi futuri) durante l'implementazione del Client Device permettendo allo sviluppatore di focalizzare la sua attenzione soltanto sullo sviluppo del codice sorgente che realizza il messaggio di sincronizzazione, e sulla funzione da eseguire che gestisce il messaggio ricevuto. Inoltre, questa metodologia permette al Server Device di evitare l'esecuzione di un controllo di funzioni, e quindi di evitare un onere computazionale da parte del suo processore, soprattutto nei casi in cui un numero elevato di Client Device è collegato al Server Device. Il Server Device deve soltanto verificare se il Client Device, che ha inviato il messaggio, ha i diritti per poter inviare quel messaggio.
La funzione **Invia_Azione** è simile alla funzione **Invia_Comando**. Le funzioni **Invia_Azione** e **Invia_Comando** inviano, rispettivamente, il messaggio di sincronizzazione di *azione* e di *comando*. L'unica differenza consiste nel fatto che il Client Device (che esegue la funzione **Invia_Azione**) alla fine dell'invio del messaggio riceve il messaggio dall'evento *azione* del socket. Quindi finita la gestione di tale evento, torna alla esecuzione della funzione **Invia_Azione** ed aggiorna il contenuto HTML del progetto nel Server Device, chiamando la funzione **Aggiorna_HTML** e mandando il codice HTML al Server Device.
La funzione **Ottieni_HTML** viene eseguita quando un Client Device si collega per la prima volta al Server Device, e quindi richiede tutto il codice HTML della lezione. La funzione **Ottieni_HTML** esegue questo meccanismo, e quando il Client Device ottiene la lezione (dal Server Device) in formato HTML, setta il *Tipo_HTML* di tutti gli oggetti presenti nella lezione.
La funzione **Invia_Input** è importante in quanto serve ad inoltrare al Server Device gli input verso i programmi i quali sono in attesa di ricevere un input per continuare la loro esecuzione.
Infine, la funzione **Compila** invia un messaggio al Server Device con all'interno

tutti i file in linguaggio C necessari per la compilazione dei programmi.

B.8 Files Forme.js e Gestione_Forme.js

Questi due files contengono alcune funzioni che creano e gestiscono il codice iconico, che permette di costruire il comportamento logico degli algoritmi. Queste funzioni permettono di creare tutti gli elementi visuali HTML *Canvas*, i quali sono di 2 tipi: i costrutti iconici del linguaggio di programmazione visuale e le frecce che formano il flusso iconico degli algoritmi, descritti nel paragrafo 4.2.
La funzione **Inserisci_Forma** crea un'icona e le fornisce un insieme di attributi, tra i quali i più importanti sono: il *testo* da visualizzare nell'icona, vedi paragrafo 4.2.6; il *Livello_Annidamento* il quale è importante per la corretta visualizzazione del colore delle icone che formano il flusso della funzione, vedi paragrafo 4.2.1; ed una serie di attributi che definiscono gli archi incidenti ed uscenti dall'icona: *Arco_Avanti*, *Arco_Indietro*, *Arco_Annidato_Indietro*, *Arco_Annidato_Avanti*. I costrutti iconici che corrispondono ai costrutti dei cicli hanno anche l'attributo *Arco_Annidato_Avanti*, mentre il costrutto iconico condizionale ha anche gli attributi *Arco_Then* e *Arco_Else*. Creato l'oggetto HTML, la funzione **Inserisci_Forma** chiama la funzione **Setta_Elemento**, già discussa. Questa funzione esegue una verifica sugli oggetti *Canvas*, ovvero verifica se l'oggetto è: un *Archetipo*, ovvero se l'oggetto visuale HTML è stato trascinato dalla sezione dei costrutti dentro la Visual Code Area; oppure se l'oggetto è una icona all'interno della Visual Code Area; infine verifica se l'icona è un arco. Eseguita la corretta verifica, la funzione **Setta_Elemento** setta i corretti eventi per il tipo di oggetto HTML, e quindi la logica di utilizzo per l'icona. A prescindere dal tipo di icona, **Setta_Elemento** chiama la funzione generica **Disegna_Stato_Forma**, che a sua volta chiama la funzione appropriata per disegnare il tipo di elemento *Canvas*, passato come parametro.
Le frecce consistono di 5 elementi: la parte iniziale, che assume la forma di un pallino, viene chiamata *Pallino*; la parte finale, che assume la forma di un triangolo, è chiamata *Freccia*; ed infine 3 parti intermedie che vengono chiamate *Arco_0*, *Arco_1* e *Arco_2*. I 5 elementi, che compongono l'arco, vengono creati usando le funzioni **Crea_Arco**, **Crea_Arco_Annidato**, **Crea_Arco_ThenOrElse** e **Crea_Arco_Etichetta_a**. Ognuna di queste funzioni individua l'icona di partenza e di destinazione ed inserisce l'arco chiamando rispettivamente le funzioni **Inserisci_Arco**, **Inserisci_Arco_in_ Annidamento**, **Inserisci_Arco_in_ThenOrElse**. Queste funzioni creano gli archi, rispettivamente, nel flusso dell'algoritmo iconico, durante la creazione di un blocco annidato nel caso dei cicli, oppure la creazione di una ramificazione in seguito all'uso del costrutto iconico condizionale.

Oltre alla creazione delle icone, sono state create le funzioni per la rimozione degli archi, che viene eseguita successivamente ad una esplicita azione dell'utente tramite menù a tendina o in seguito alla cancellazione di un'icona e, necessariamente, di tutti gli archi ad essa associati. Per espletare questa funzionalità sono state create le funzioni **Elimina_Archi**, **Elimina_Arco** e **Elimina_Arco_Annidato**. La funzione **Elimina_Archi** verifica l'esistenza di archi associati ad un'icona, se ve ne sono, per ogni arco, viene invocata la funzione **Elimina_Arco** o **Elimina_Arco_Annidato** per rimuovere tutti gli elementi *Canvas* che costituiscono l'arco.
L'uso del linguaggio iconico permette di spostare le icone in vari punti della Visual Coce Area. Queste icone possono avere archi incidenti ed uscenti da esse, è stato quindi necessario realizzare un algoritmo, la funzione **Aggiorna_Arco**, abbastanza complessa, per la corretta visualizzazione di tutti gli archi collegati alle icone. Questo algoritmo considera le posizioni delle icone di partenza e di terminazione di un arco. In base alla posizione delle icone viene creata la visualizzazione dell'arco, ovvero la dimensione e posizione di *Pallino*, *Freccia*, *Arco_0*, *Arco_1* e *Arco_2*. La funzione **Aggiorna_Arco** viene invocata dalla funzione **Aggiorna_Archi** per ogni arco associato all'icona da spostare.
Infine, come detto precedentemente, gli archi possono assumere colori differenti in base al loro livello di annidamento. Per realizzare questa caratteristica sono state sviluppate le funzioni **Aggiorna_Tutti_i_Livelli** e **Aggiorna_Livello_In_Cascata**. La funzione *Aggiorna_Livello_In_Cascata* aggiorna il numero ed il colore del livello di annidamento di ogni icona, e i relativi archi, partendo dall'icona, passata come parametro, fino alla fine del flusso iconico. La funzione **Aggiorna_Tutti_i_Livelli** determina l'icona *Start* della Visual Code Area dell'algoritmo iconico ed invoca la funzione **Aggiorna_Livello_In_Cascata** passandole come parametro l'icona *Start*.

B.9 File Gestione_Progetto.js e Gestione_ Schede.js

Questi files si occupano della gestione dei progetti, sia dal punto di vista della GUI sia dal punto di vista informativo. Essi creano tutta la struttura di un progetto e del *Contenitore_Progetto* indicato nel paragrafo B.4. I files gestiscono anche gli eventi dei click del mouse sui vari Tab dei progetti, files e funzioni inerentemente la navigazione e la visualizzazione dei progetti, files e funzioni, nonché la gestione della creazione, modifica e rimozione di progetti, file e funzioni, parametri, variabili e costanti delle funzioni.

Alcune delle funzioni più importanti sono **Setta_HTML**, **Setta_Elemento** e **Setta_Elementi_Figli**. Le prime due funzioni sono state ampiamente discusse nei precedenti paragrafi. La funzione **Setta_Elementi_Figli** prende in input un elemento e per ogni suo figlio, e discendente anche non diretto, viene invocata la funzione **Setta_Elemento**.
Altre due funzioni importanti sono due: **Verifica_Comportamento_Tipo_ Costante_Variabile** e **Aggiorna_Corpo_Dati_a_File**.
La funzione **Verifica_Comportamento_Tipo_Costante_Variabile** esegue un controllo sui valori numerici inseriti negli elementi visuali HTML *input* per verificare la corretta valorizzazione delle variabili e costanti numeriche. Mentre la funzione **Aggiorna_Corpo_Dati_a_File**, crea il codice HTML da inserire nella sezione *Codice_HTML_File_Dati* del contenitore *Contenitore_File_View_ Contenitore_Codice_HTML_Funzione* che visualizza il codice sorgente in linguaggio C di ogni file, e che corrisponde alle variabili e costanti del file.

B.10 File Gestione_Espressioni.js e Gestione_ Assegnazioni.js

Nel paragrafo B.8 si è discussa l'implementazione del linguaggio iconico che permette di costruire il comportamento logico degli algoritmi. In questo paragrafo si discuterà del linguaggio iconico che permette di costruire le espressioni e le assegnazioni e permette anche di focalizzare l'attenzione dello studente nella sintassi e semantica di un linguaggio di programmazione. Per la gestione delle espressioni ed assegnazioni si è resa necessaria la creazione di una finestra all'interno della pagina di SIRENE. Questa finestra è stata realizzata in una modalità che nelle applicazioni dei sistemi operativi viene definita: "*modalità dialogo*", ovvero una speciale modalità di visualizzazione delle finestre, chiamata "*modale*", in cui l'utente vede apparire una finestra nello schermo. Tale finestra occupa una porzione dello schermo e disabilita la possibilità all'utente di interagire con la restante parte dello schermo visualizzata, opacizzandone visivamente il contenuto. Per emulare questa caratteristica utilizzando soltanto elementi HTML si è utilizzato un elemento *div* che occupa l'intero schermo usando la proprietà *z-index* dello stile CSS. Questa proprietà imposta l'oggetto HTML in un layer, detto anche "*strato*", diverso dagli altri elementi. La funzione del layer è simile a quella di un piano intersecante una ipotetica asse Z in tre dimensioni, in cui l'utente vede gli oggetti. Maggiore è il valore della proprietà *z-index* più vicino sarà questo piano virtuale all'utente. Inoltre, per simulare la modalità "*modale*" è necessario anche utilizzare la segue proprietà e suoi valori:

background-color: rgba(0,0,0,0.4);

dove rgba identifica una funzione di codifica del colore del CSS che, in questo caso, specifica il colore nero (dai primi 3 valori rgb), mentre l'ultimo valore '0.4' identifica la modalità "*alphablending*", ovvero l'opacità dell'oggetto, al 40%. Il risultato, dell'accoppiamento delle due proprietà appena descritte, è di avere uno sfondo dello schermo opacizzato di nero e non selezionabile. La finestra di interazione della selezione è posta nel centro di questo oggetto *div* ed è possibile interagire con essa. La funzione che crea la finestra delle espressioni e delle assegnazioni è **Crea_Espressione_Funzione**. Questa funzione genera la seguente gerarchia di codice HTML:

```
Espressione_Modale
└── Espressione o Assegnazione
    ├── Sezione_Chiudi_Espressione
    │   └── span_Sezione_Chiudi_Espressione
    ├── Sezione_Contenuto_Espressione
    │   ├── Sezione_Lato_Sinistro_Espressione
    │   └── Espressione_Sezione_Area
    │       ├── Espressione_Sezione_Visual_Code
    │       └── Espressione_Sezione_Textual_Code
    └── Espressione_Bottoni
```

Inoltre, gli elementi *Sezione_Lato_Sinistro_Espressione* e *Espressione_ Sezione_Area* possiedono le seguenti strutture gerarchiche:

```
Espressione_Sezione_Area
├── Espressione_Sezione_Titolo_Icona
├── Espressione_Sezione_Titolo_Descrizione_Icona
├── Asssegnazione_Sezione_Variabile_e_Assegnazione
├── Asssegnazione_Sezione_Variabile_Area
└── Asssegnazione_Sezione_Assegnazione_Area_Scelta
```

```
Sezione_Lato_Sinistro_Espressione
├── Espressione_Funzioni
├── Espressione_Dati
├── Espressione_Dati_Manuali
├── Espressione_Operatori_Aritmetici
├── Espressione_Operatori_Relazionali
├── Espressione_Operatori_Logici
└── Espressione_Operatori_Bitwise
```

Il secondo elemento di *Espressione_Modale* può essere di tipo *Espressione* oppure di tipo *Assegnazione*. La differenza consiste che nella finestra contrassegnata con *Assegnazione* è presente la sezione *Asssegnazione_Sezione_ Variabile_e_Assegnazione* ed il suo contenuto, mentre nel tipo *Espressione* tale sezione non è presente. Poiché in HTML non esiste la caratteristica del polimorfismo, si è dovuto ricorrere alla differenziazione data dal valore dell'attributo *Tipo_HTML* ed aggiunzione del contenitore *Asssegnazione_Sezione_Variabile_e_Assegnazione*[1] nel caso in cui il secondo elemento della gerarchia sia di tipo *Assegnazione* e conseguente differenziazione della gestione delle due finestre per la creazione del codice sorgente in linguaggio C.
La finestra modale può essere chiusa soltanto utilizzando i bottoni presenti nella sezione *Sezione_Chiudi_Espressione* oppure in *Espressione_Bottoni*.
Il contenitore *Sezione_Contenuto_Espressione* è la sezione in cui l'utente crea le espressioni e le assegnazioni alle variabili. Nella *Sezione_Lato_Sinistro_ Espressione* vi è un insieme di elementi HTML. Tali elementi permettono di costruire tutte le espressioni (nel seguito, per brevità, si intenderà espressioni anche per assegnazioni, se non diversamente specificato). Ogni oggetto in questa sezione può essere trascinato e rilasciato all'interno dell'area *Espressione_Sezione_Visual_Code*, e durante la costruzione delle espressioni viene riportato il codice sorgente ad esso associato nella sezione *Espressione_Sezione_Textual_Code*.
Sono presenti alcuni contenitori chiamati: *Espressione_Funzioni*, *Espressione_Dati*, *Espressione_Dati_Manuali*, *Espressione_Operatori_Aritmetici*, *Espressione_Operatori_Relazionali*, *Espressione_Operatori_Logici*, *Espressione_Operatori_Bitwise*. Ognuno di questi contenitori racchiude gli elementi visuali HTML che possono essere trascinati e rilasciati, tramite il meccanismo "drag&drop" nel contenitore *Espressione_Sezione_Visual_Code*. All'interno del contenitore *Espressione_Funzioni* è presente un selettore con menù a tendina che permette di scegliere il file. Questo selettore ha un evento "*onchange*" scatenato quando un utente cambia il suo contenuto, ed invoca la funzione **Espressione_Opzione_Change**, per mezzo di un messaggio di sincronizzazione a tutti i Client Device. In quest'ultima funzione **Espressione_Opzione_Change** vengono creati dinamicamente, dentro una tabella, una lista di tutte le funzioni che sono proprie della libreria standard di C, implementate in SIRENE, e di tutte le funzioni create mediante il linguaggio iconico. Questi elementi possono essere trascinati e rilasciati, come specificato precedentemente.
La funzione **Crea_Espressione_Funzione** genera anche *Espressione_Dati* che è il contenitore racchiude tutti i dati: parametri, variabili e costanti utilizzabili all'interno dell'espressione. Questi dati possono essere i parametri della funzione,

[1] Questa sezione, ed il suo contenuto, è presente soltanto se il secondo elemento della gerarchia è di tipo *Assegnazione*.

le variabili e costanti globali o locali della funzione. In questa funzione, viene anche creato il contenitore *Espressione_Dati_Manuali* che racchiude un insieme di oggetti che permettono di inserire manualmente valori attraverso la digitazione. Gli altri oggetti visuali degli altri contenitori hanno lo stesso simbolo omologo agli operatori aritmetici, logici, relazionali e bitwise. Nella sezione *Sezione_Lato_Sinistro_Espressione* troviamo, dunque, tutti gli elementi utili per costruire una espressione e permettere all'utente di focalizzare la sua attenzione nello sviluppo di una corretta espressione.
Per la creazione di espressioni omologhe al linguaggio C sono stati affrontati due problemi: il primo problema riguarda l'annidamento delle espressioni, ovvero la possibilità di creare espressioni da passare come parametri dentro alcune funzioni, questo annidamento può essere virtualmente infinito nel linguaggio C; il secondo problema consiste nello spostare gli elementi dentro il contenitore *Espressione_Sezione_Visual_Code* in maniera da dare amplia flessibilità all'utente di poter manipolare e costruire l'espressione stessa, ed anche di poter spostare elementi da una espressione annidata ad un'altra, e così via. Per risolvere agevolmente questi problemi, mantenendo l'omologa flessibilità del linguaggio C, è stata adottata la seguente soluzione: ogni oggetto rilasciato nella *Espressione_Sezione_Visual_Code* genera un evento che invoca la funzione **Espressione_Drop**, o **Assegnazione_Drop** nel caso di assegnazione, il quale crea un oggetto HTML *span*, chiamato *nuovo_elemento* e che racchiude 3 elementi *span*: un elemento a sinistra chiamato *span_sinistro*, un elemento centrale chiamato *span_corpo* ed un elemento a destra chiamato *span_destro*. Se un oggetto rilasciato è un operatore (logico, relazionale, aritmetico o bitwise) allora *span_corpo* contiene lo stesso simbolo dell'operatore. Invece, gli oggetti che sono gli omologhi delle parentesi, conterranno anche un altro oggetto *span* che permette di annidare altre espressioni dentro le parentesi. La funzione **Espressione_Drop**, o **Assegnazione_Drop** nel caso di assegnazione, invoca anche la funzione **Setta_Elemento** e **Setta_Elementi_Figli** per *nuovo_elemento*. Se l'oggetto rilasciato è, invece, un parametro, o variabile o costante, allora l'elemento visuale *span_corpo* contiene il nome della variabile e l'oggetto *nuovo_elemento* avrà un attributo, chiamato *rif* che contiene l'*id* del parametro, variabile o elemento. Questo attributo *rif*, discusso in seguito, permette di mantenere aggiornato il valore del nome del parametro, variabile o costante, quando questo viene cambiato in qualunque momento durante lo sviluppo iconico del programma. Inoltre, Se l'oggetto HTML si riferisce ad una funzione allora viene inserito un altro elemento, chiamato *Espressione_span_nome_funzione*, prima di *span_corpo*, ed, in questo caso, *span_corpo* racchiude una coppia di parentesi, una aperta e l'altra chiusa, con all'interno un eventuale altro oggetto *span* contentente da zero a più campi (che identificano i parametri in accordo alla definizione della funzione) in cui si possono inserire altre espressioni e *Espressione_span_nome_funzione* contiene il

nome della funzione rilasciata. Questi campi in cui si possono creare altre espressioni permettono di poter invocare funzioni inserendo anche ulteriori parametri o ulteriori espressioni annidate, garantendo la stessa flessibilità che il linguaggio C permette quando vengono inserite espressioni come parametri da fornire alle funzioni. In questa maniera, viene implementata l'omologa flessibilità di annidamenti proprie delle espressioni e delle assegnazioni come in un linguaggio di programmazione classico.
Al fine di risolvere il problema di spostare gli oggetti visuali a destra o a sinistra degli altri elementi visuali all'interno del contenitore *Espressione_Sezione_Visual_Code*, gli oggetti *span_sinistro* e *span_destro* hanno l'evento *dragover* valorizzato (tramite la funzione **Setta_Elemento**) rispettivamente con le funzioni **Azione_Espressione_Drag_Move_Sinistra** e **Azione_Espressione_Drag_Move_Destra**. Queste due funzioni vengono invocate ogni volta che un oggetto, trascinato dall'utente, si trova sopra di esse e il loro funzionamento è quello di spostare fisicamente l'oggetto trascinato a sinistra di *nuovo_elemento* o alla destra di *nuovo_elemento*. Questa coppia di funzioni assicura il corretto spostamento degli oggetti nella posizione desiderata dall'utente durante il trascinamento e permette anche di annidare oggetti dentro altre espressioni, o fare uscire oggetti da espressioni annidate. Infatti, ogni coppia di parentesi inserita, o creata dinamicamente da SIRENE, può contenere un elemento *span* chiamato *Espressione_span_Parametri_corpo* che permette di inserire altre espressioni usando il codice iconico. Inoltre, le due funzioni invocano la funzione **Espressione_Codice_da_Stesso_Livello**, o nel caso di assegnazione invocano la funzione **Assegnazione_Codice_da_Stesso_Livello**, che realizzano un parser per l'espressione creata dinamicamente e di cui se ne discuterà successivamente.
L'oggetto *Espressione_span_Parametri_corpo* ha associati 3 eventi: *ondragenter*, *ondragleave* e *ondragover* che vengono invocati quando l'utente sta trascinando un oggetto e si strova col puntatore del mouse sopra *Espressione_span_Parametri_corpo*. I primi due eventi, *ondragenter* e *ondragleave*, eseguono rispettivamente le funzioni **Azione_Espressione_Drag_Enter** e **Azione_Espressione_Drag_Leave** che inviano un messaggio di sincronizzazione a tutti i Client Device per eseguire le funzioni **Espressione_Drag_ Enter** e **Espressione_Drag_Leave**. Queste due funzioni permettono di modificare il colore di fondo di un elemento *Espressione_span_Parametri_corpo* per indicare che si sta trascinando un oggetto dentro una specifica sezione dove è presente un'espressione, e queste due funzioni servono ad inserire gli elementi visuali dentro un'espressione annidata.
Invece, allo stesso modo, l'evento *ondragover* invoca la funzione **Azione_ Espressione_Drag_move_in_Elemento** che permette di eseguire in tutti i Client Device, la funzione **Espressione_Drag_move_in_Elemento**. Questa funzione inserisce l'oggetto trascinato dentro l'espressione annidata.

Tutte le funzioni di trascinamento descritte non interrompono l'azione di trascinamento dell'oggetto selezionato permettendo un flusso continuo nella modellazione dell'espressione e spostando i vari oggetti visuali dinamicamente.
Nel caso in cui la finestra sia di tipo assegnazione verrà inserito l'elemento *Assegnazione_Sezione_Variabile_e_Assegnazione* riportato nella gerarchia B.10, inclusi gli elemento *Asssegnazione_Sezione_Variabile_Area* ed *Asssegnazione_Sezione_Assegnazione_Area*. Il contenitore *Asssegnazione_Sezione_Variabile_Area* ha due eventi associati: *ondrop* e *ondragover*. Quando l'evento *ondrop* viene innescato, esso verifica se l'oggetto rilasciato è una variabile iconica: se questa condizione risulta vera allora viene verificato se *Asssegnazione_Sezione_Variabile_Area* contiene una variabile, se *Asssegnazione_Sezione_Variabile_Area* non contiene una variabile allora un elemento *nuovo_elemento* è inserito, altrimenti no. Diversamente, nel caso in cui si dovesse trascinare qualunque elemento diverso da una variabile nella sezione *Asssegnazione_Sezione_Variabile_Area* allora questo evento viene scartato e nessun altro elemento è inserito, ad eccezione delle parentesi quadrate soltanto nel caso in cui è già presente una variabile, in questa sezione, e questa variabile sia un array. In questo caso, viene creato un altro elemento *nuovo_elemento* alla destra della variabile e può contenere una espressione generica (costruibile con il meccanismo precedentemente descritto) per indicare l'indice.
La funzione **Espressione_Codice_da_Stesso_Livello** riceve come parametro il contenitore *Espressione_Sezione_Visual_Code*. Questa funzione genera il codice sorgente e realizza il parser del linguaggio iconico di SIRENE finalizzato alla costruzione delle espressioni. Essa analizza ogni elemento di *Espressione_Sezione_Visual_Code* e genera l'omologo codice in linguaggio C. Questa funzione è sia iterativa e sia ricorsiva: è iterativa quando analizza elementi che sono sullo stesso livello di annidamento di una espressione ed è ricorsiva quando, durante l'analisi, incontra una funzione e una parentesi (tonda o quadra) che indicano un ulteriore annidamento dell'espressione. Per ogni funzione iconica analizzata, la funzione **Espressione_Sezione_Visual_Code** esegue una chiamata a se stessa per ricevere il contenuto tra parentesi tonde (se esiste una espressione tra queste parentesi), diversamente prende il nome della funzione e lo memorizza in una stringa, ed alla fine della procedura viene restituito tutto il codice sorgente generato in linguaggio C. Il nome della funzione (da inserire nel codice sorgente) viene trovato prendendo l'*id* dell'oggetto HTML che contiene la funzione memorizzata, in questa maniera, nel caso in cui si modifichi, in qualunque momento, il nome della funzione questa viene automaticamente aggiornata sia nel codice sorgente in linguaggio C e sia nell'elemento visuale del contenitore *Espressione_Sezione_Visual_Code*. Il medesimo meccanismo avviene quando l'elemento analizzato è una variabile, una costante o un parametro. Invece, quando la funzione **Espressione_Codice_da_Stesso_Livello** è all'interno delle parentesi tonde di

una funzione allora **Espressione_Codice_da_Stesso_Livello** analizza il tipo di parametro richiesto da questa funzione e se il parametro richiesto dalla funzione è un parametro per ferimento allora il parser includerà, solo nel codice sorgente del linguaggio C, il simbolo "*&*" prima della variabile, costante o parametro. Inoltre, nel caso in cui il parser, durante la sua analisi, dovesse verificare che gli elementi visuali utilizzati dovessero creare una espressione errata nella sintassi allora gli oggetti visuali inseriti in maniera errata saranno colorati di arancione. In questa maniera il linguaggio iconico fornisce un feedback visuale all'utente durante la costruzione dell'espressione affinché l'utente corregga l'espressione errata.
La funzione **Controlla_Range_Numero** è la funzione che verifica l'input numerico inserito da un utente quando deve valorizzare un campo. In base al tipo di dato (char, short, int, ecc.) questa funzione verifica se il valore inserito rientra nel range del tipo di dato (es '-128..127' nel caso in cui il campo richieda un valore per il tipo char). Questa funzione ritorna *True* se il valore è corretto, altrimenti restituisce *False* ed, in questo caso, inibisce l'ultimo inserimento ripristinando il valore allo stato precedente del campo modificato.

B.11 File Gestione_For.js

Questo file è importante per la gestione dell'icona omologa al ciclo *For* del linguaggio C e viene gestita in una finestra creata dalla funzione **Crea_ Finestra_For**. La gestione del costrutto iconico omologo al ciclo *For* è stata sviluppata in maniera simile a quella della gestione delle espressioni e delle assegnazioni. Tuttavia, è stato necessario creare diverse sezioni poiché nel linguaggio C il costrutto *For* richiede 3 campi : uno contenente una sezione di *inizializzazione*, un altro campo che contiene una sezione *condizionale*, ed infine un campo in cui si modificano i valori delle variabili utilizzate per il ciclo. Per focalizzare l'attenzione dello studente nelle tre differenti sezioni, si è reso necessario costruire tre sezioni omologhe ai campi sopra descritti, ovvero i contenitori *div*:

Espressione_For_Contenitore_Assegnazioni
Espressione_For_Contenitore_Condizioni
Espressione_For_Contenitore_Incrementi

Espressione_For_Contenitore_Assegnazioni funziona da contenitore di tutte le inizializzazioni delle variabili utilizzate. Ogni inizializzazione è un assegnazione indipendente dalle altre e può essere spostata dall'utente, all'interno, tramite trascinamento, come gli oggetti nelle espressioni ed assegnazioni descritte nel paragrafo precedente, ma in questo caso viene trascinata un'assegnazione intera.

Il contenitore *Espressione_For_Contenitore_Incrementi* funziona in maniera omologa per gli incrementi, i quali vengono gestiti come assegnazioni.
Infine, il contenitore *Espressione_For_Contenitore_Condizioni* contiene espressioni che anch'esse possono essere spostate.
L'inserimento di queste assegnazioni ed espressioni avviene per mezzo di un bottone alla sinistra di ogni contenitore. Il codice sorgente risultante dalla costruzione delle assegnazioni ed espressioni create è riportato nel contenitore *Espressione_For_Codice_HTML*. Questo codice viene aggiornato ad ogni azione dell'utente all'interno della finestra per la gestione dell'icona *For*.

B.12 File Gestione_Codice.js

Questo file è fondamentale per la creazione del codice sorgente in linguaggio C e realizza il parser per le icone, vedi paragrafo 4.4. Le funzioni che realizzano questo parser sono:

Funzione_Genera_Codice
Funzione_Genera_Codice_Iterando
Crea_Codice_Testo_HTML

La funzione **Funzione_Genera_Codice** inizialmente prende l'oggetto *Contenitore_VSLC*, ovvero la sezione dove sono inserite tutte le icone, e lo fornisce come parametro alla funzione **Funzione_Genera_Codice_Iterando** dalla quale ricevere, come output, tutto il codice sorgente in linguaggio C ed infine inserisce questo codice nella sezione del codice sorgente all'interno del contenitore *Codice_Corpo_Funzione*, che visualizza il codice testuale della funzione. Successivamente, la funzione **Funzione_Genera_Codice** invoca la funzione **Aggiorna_Includes_in_Funzione** che aggiorna tutte le dipendenze della funzione e le inserisce nel contenitore *Codice_HTML_File_Include* e nel contenitore che visualizza il codice sorgente del file contenente la funzione che si sta sviluppando in maniera iconica. Infine, la funzione **Funzione_Genera_Codice** invoca la funzione **Aggiorna_HTML_Funzione_In_ File** che aggiorna anche il corpo della funzione del file.
La funzione **Funzione_Genera_Codice_Iterando** realizza il parser del linguaggio iconico. Anche questa funzione è una funzione sia iterativa e sia ricorsiva: è iterativa quando il parser analizza un elemento nello stesso livello nel codice iconico, ovvero quando durante l'analisi il parser verifica che l'oggetto visuale è nello stesso livello delle altre icone (icone dello stesso livello hanno il valore dell'attributo *livello_annidamento* uguale); diversamente, è ricorsiva quando

il parser verifica che l'icona ha un codice annidato, ovvero ha l'attributo *arco_annidato_avanti* (se l'icona rappresenta un ciclo) o l'attributo *arco_then* o l'attributo *arco_else* (se l'icona rappresenta una condizione) valorizzati con un valore numerico maggiore del livello di annidamento attuale, allora vi è un annidamento nel flusso iconico (graficamente rappresentato da un cambiamento di colore nell'arco) e quindi un blocco annidato di codice iconico. In questo caso, la funzione **Funzione_Genera_Codice_Iterando** invoca se stessa per proseguire l'analisi ad un livello più profondo di annidamento del flusso. La funzione **Funzione_Genera_Codice_Iterando** restituisce il codice sorgente in linguaggio C sia come testo e sia in formato HTML (per l'inserimento corretto nei contenitori).
La funzione **Crea_Codice_Testo_HTML** crea le sezioni del codice HTML visualizzato in *Contenitore_Codice_HTML_Funzione*. Le sezioni generate sono:

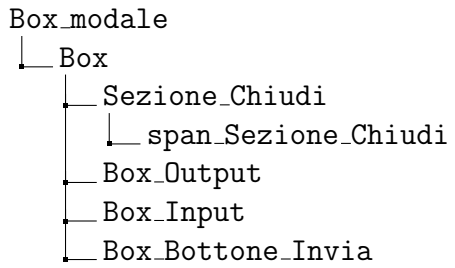
Codice_Header
Codice_Parentesi_Aperta
Codice_Area_Dati
Codice_Corpo_Funzione
Codice_Parentesi_Chiusa

Le sezioni più importanti sono *Codice_Header*, *Codice_Area_Dati*, *Codice_Parentesi_Chiusa*. La sezione *Codice_Header* contiene l'header della funzione e permette di essere facilmente cambiato quando vengono variati il nome, il tipo e i parametri della funzione che si sta sviluppando.
La sezione *Codice_Area_Dati* contiene la dichiarazione (con eventuale inizializzazione) di tutte le variabili e le costanti definite durante lo sviluppo dell'algoritmo iconico.
Mentre, la sezione *Codice_Corpo_Funzione* contiene tutto il codice sorgente associato alle icone durante la fase di sviluppo visuale.
Ognuna di queste sezioni viene aggiornata soltanto quando è necessario modificarle, senza ricostruire e modificare interamente il codice sorgente della funzione quando varia soltanto una parte di esse.
Il file **Gestione_Codice.js** contiene anche una serie di funzioni che servono ad aggiornare l'header della funzione, i suoi dati oppure il corpo della funzione, in accordo alle azioni dell'utente.

B.13 File Gestione_Terminale.js

Questo file implementa l'emulatore di un terminale, un Web terminal o web Shell, usando soltanto l'HTML. La funzione che crea la finestra del Web Terminal è **Crea_Interazione_Programma**. Questa finestra è nominata *Box_modale* ed

ha la seguente gerarchia HTML:



L'oggetto *Box_Output* graficamente somiglia ad un *terminal* di Linux o ad una *shell* di Windows. *Box_Output* visualizza tutti gli output inviati dal Server Device e dai programmi compilati.
La funzione **Crea_Interazione_Programma** crea anche l'oggetto *input*, chiamato *Box_Input*, che serve per inviare gli input necessari ai programmi per proseguire la loro esecuzione.
La funzione **Inserisci_Output** inserisce l'input, ricevuto dal Server Device, nel *Box_Output*.
La funzione **Richiesta_Input** abilita *Box_Input* per permettere l'invio dell'input da parte di un utente al programma in attesa di input.
La funzione **Invia_Input_A_Server** invia il messaggio al Server Device contenente il comando da eseguire e l'input richiesto dal programma in attesa. Se l'invio del messaggio è avvenuto con successo allora viene aggiornato l'HTML del progetto nel Server Device invocando la funzione **Aggiorna_HTML**, altrimenti no. Inoltre, il messaggio include il nome della funzione che il Server Device invierà a tutti i Client Device: se il comando è stato eseguito con successo (ovvero se la funzione **Inviato_Input** è andata a buon fine) allora tutti i Client Device devono disabilitare l'oggetto *Box_Input* fin quando non riceveranno un'altra richiesta di input.
Gli input vengono inviati ogni volta che si preme il tasto "invio" o si clicca sul bottone *Box_Bottone_Invia*. Invece, l'oggetto *span_Sezione_Chiudi* permette di inviare un comando al Server Device che gli impone di terminare il programma con cui si sta interagendo. La chiusura del programma avviene per mezzo di un segnale di *kill* che impone al programma di terminare, a prescindere dallo suo stato di esecuzione.

B.14 File Menu.js

Questo file gestisce la creazione, la visualizzazione e la chiusura dei menù a tendina che vengono visualizzati quando un utente clicca con il mouse sopra un'i-

cona o un elemento che compone un arco. La funzione **Apri_Menu** si occupa di creare e visualizzare il menù quando un utente clicca col tasto destro sopra un'icona e, in accordo al tipo di icona ed ai suoi archi uscenti, verranno visualizzate alcune azioni, ad esempio: se l'icona è l'omologa di un ciclo e non vi sono archi uscenti da quest'icona allora il menù visualizzerà le azioni di "*Collega Avanti*" o "*Collega in Annidamento*"; se invece l'icona è l'omologa di una condizione allora il menù visualizzerà le azioni di "*Collega Avanti*" o "*Collega in Then*" o "*Collega in Else*"; infine per tutte le altre icone, il menù visualizzerà soltanto l'azione di "*Collega Avanti*" per permettere al flusso iconico di continuare. Se l'icona dovesse avere già un arco uscente, non verrà visualizzata l'azione per quel tipo di arco.

B.15 File Comuni.js

Questo file contiene svariate funzioni invocate dai file precedentemente descritti. Vi sono alcune funzioni che ottengono l'accesso ad oggetti HTML inerenti la gestione interna dei progetti, ad esempio le funzioni **Prendi_Progetto**, **Prendi_Libreria**, **Prendi_File**, **Prendi_Files_del_ Progetto**, **Prendi_Funzione**, che ricevono in input il nome dell'oggetto e restituiscono, rispettivamente, l'oggetto HTML del progetto, della libreria, di un file specifico, tutti i files del progetto o di una funzione. Vi sono alcune funzioni che restituiscono uno specifico discendente (anche non diretto nella gerarchia HTML), un figlio o tutti i figli di un oggetto HTML (nella gerarchia HTML), o uno specifico genitore: **Prendi_Figli_Da_Attributo**, **Prendi_Figli_Da_Tipo_ HTML**, **Prendi_Discendenti_Da_Attributo**, **Prendi_Unico_Figlio_Da_Attributo**, **Prendi_Figli**, **Prendi_Discendenti_Da _Tipo_HTML**, **Prendi_Unico_Figlio_Da_Tipo_ HTML**, **Prendi_Unico_Discendente_Da_Tipo_HTML**, **Prendi_Genitore_Con_Tipo_HTML**. Tutte le precedenti funzioni rintracciano uno o più specifici elementi HTML partendo da un oggetto in particolare, passato come primo parametro, o da un attributo e valore ben definiti oppure dallo specifico attributo *Tipo_HTML*. Poiché ogni oggetto in SIRENE ha uno specifico *Tipo_HTML* ognuna delle precedenti funzioni garantisce il rintracciamento dell'oggetto, o oggetti, cercati. Nel caso la ricerca riguardasse uno specifico oggetto, le precedenti funzioni garantiscono il rintracciamento di uno specifico ed univoco oggetto HTML, o di una collezione di essi.
La funzione certamente più usata in SIRENE è **Prendi_Tipo_HTML** che verifica se l'oggetto possiede l'attributo *Tipo_HTML*: se questa condizione dovesse essere falsa allora viene impostato questo attributo tramite l'invocazione della funzione **Setta_Tipo_HTML** ed infine viene restituito il *Tipo_HTML* dell'oggetto.
Vi sono anche le funzioni **Annulla_Evento** e **Cancella_Evento** che annullano entrambe un evento generato dall'utente (ad esempio il click del mouse sopra un

oggetto HTML), in due diverse modalità: la funzione **Annulla_Evento** annulla l'evento a livello del linguaggio Javascript; mentre la funzione **Cancella_Evento** annulla l'evento sia a livello di Javascript e sia dal punto di vista applicativo Browser, ritornando anche il valore booleano *False*.

Appendice C

Utilizzo di SIRENE

C.1 SIRENE: unica pagina di lavoro

Il lato client di SIRENE è composto da un'unica pagina internet dove è possibile realizzare le applicazioni. La pagina è divisa in 5 sezioni:

A) La sezione dei costrutti (Constructs Section) ospita le icone che rappresentano i costrutti visuali del linguaggio di SIRENE. Inoltre, sono presenti alcuni pulsanti per la definizione delle variabili e delle costanti.

B) La sezione dell'area visuale (Visual Area) la quale è divisa in due sottosezioni: la parte generale (General Area), dove si definisce una funzione con il suo tipo, il suo nome ed i suoi parametri; e la parte dell'area del codice visuale (Visual Code Area, o VCA) dove si costruisce il codice visuale che rappresenta il comportamento di un algoritmo.

C) La sezione del codice sorgente (Source Code Area, o SCA), nella quale SIRENE mostra il codice sorgente, in linguaggio testuale, associato ad ogni costrutto iconico della Visual Code Area.

D) La sezione di navigazione (Tab Section), anch'essa divisa in tre sezioni: una nella quale risiedono alcuni "*Tab*" corrispondenti ai progetti, ovvero programmi, che si stanno realizzando; un'altra nella quale risiedono i "*Files*" che compongono un progetto; ed infine la sezione nella quale si mostra l'insieme generale del file corrispondente e di tutte le sue funzioni realizzate in maniera visuale.

E) La sezione dei menù (Menù Section), la quale mostra un classico menù nel quale sono presenti le funzionalità standard di un tipico ambiente di sviluppo: "*New Project*", "*New File*", "*New Function*", "*Run*", ecc.

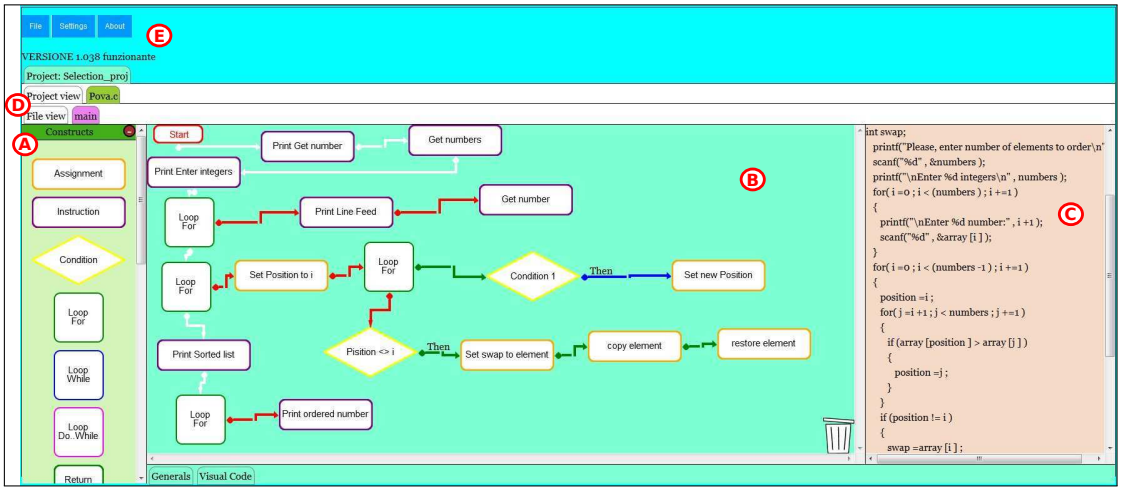


Figura C.1: Esempio dell'algoritmo di Selection sort creato con SIRENE.

C.1.1 Uso dell'ambiente grafico

L'ambiente grafico di SIRENE permette, attualmente, due livelli di utenti: il livello base e quello avanzato. Per scegliere il livello è necessario selezionarlo da un menù a tendina in alto a destra della pagina di SIRENE, vedi figura C.2. In base al livello, alcune operazioni e funzionalità sono limitate o non visualizzate.
Il livello base consiste nel facilitare la creazione di un progetto e nella limitazione dell'utilizzo di alcune funzionalità come l'uso di costanti e determinati tipi di dati messi a disposizione nel livello avanzato: "*int*", "*char*" (inteso come valore), "*long*", "*float*", "*double*". Il tipo di dato *double* viene visualizzato come un tipo "*Numero*", diversamente dal livello avanzato il quale mostra correttamente il tipo di dato. Inoltre, il tipo di dato *char* nel livello base permette di realizzare soltanto stringhe e viene mostrato come "*Stringa*", mentre nel livello avanzato vengono usate due tipi distinti di *char*: uno per le stringhe corrispondenti ad array di *char*, ed uno corrispondente soltanto ai valori numerici. Inoltre, nel livello base non è possibile realizzare espressioni usando nomi di funzioni (messe a disposizione nel livello avanzato) quali "**scanf**", "**printf**", "**getchar**" e "**putchar**", ma è possibile realizzare espressioni usando funzioni più semplici come "**scrivi**" ed "**inserisci**".

C.2 Funzionalità della sezione del menù

Come accennato, il livello base permette di creare facilmente un progetto completo cliccando su *New Project* nell'area del menù: così facendo si creerà un progetto, compreso di *File* ed una funzione "**main**" già predefinite. Per sviluppare un programma con SIRENE, con il livello avanzato, è necessario iniziare creando

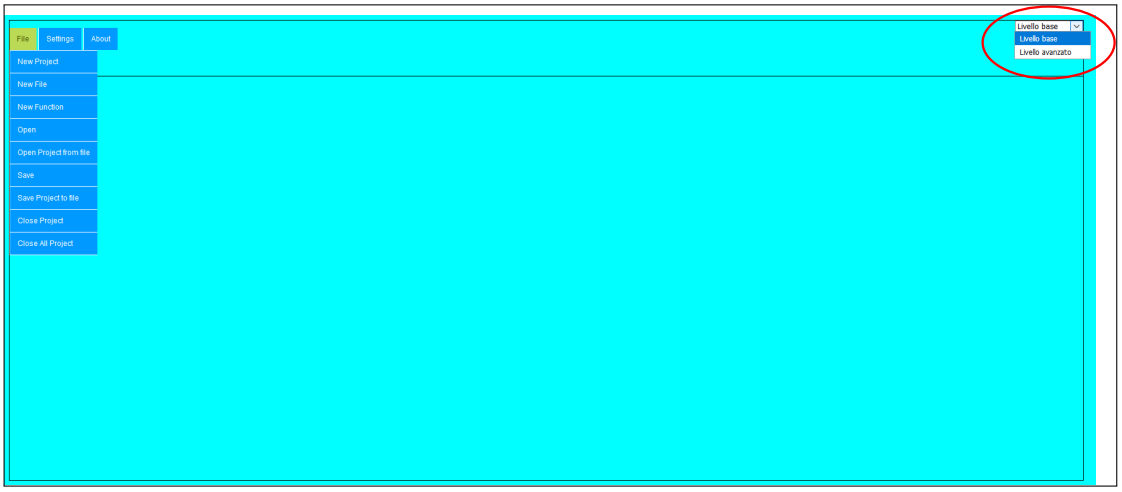


Figura C.2: SIRENE: Menù "*File*" e menù a tendina in cui è possibile selezionare il livello dell'utente.

una pagina, successivamente un file ed infine una funzione usando le rispettive voci nell'area del menù. Ad ogni azione, è necessario valorizzare correttamente i valori del nome del progetto e del file creati, vedi figura C.3. È possibile creare ulteriori progetti, file e funzioni usando le corrispettive funzionalità del menù, sia per il livello base sia per quello avanzato. Il menù permette anche di aprire un progetto salvato e di poterne salvare uno che si sta realizzando. Inoltre, è possibile chiudere un progetto o tutti i progetti aperti, vedi figura C.2.
Il menù permette anche di compilare ed eseguire il programma realizzato, e se ne discuterà nel paragrafo C.6.

C.3 Navigazione nei progetti di SIRENE

La sezione dei *Tab* di SIRENE permette agevolmente di navigare tra i progetti, i files e le funzioni realizzati. La prima fila di *Tab* permette di scegliere quale progetto mostrare all'utente. La seconda sezione dei tab mostra, come primo tab, una schermata, chiamata *Project view*, relativa alla lista di tutti i file che compongono il progetto, mentre gli altri tab corrispondono ai file del progetto, vedi figura C.4. È possibile spostarsi in un file cliccando sul tab corrispondente nella seconda sezione o facendo doppio click sul nome del file mostrato nella prima schermata.
Selezionato il file da visualizzare, viene mostrata una prima schermata, corrispondente al tab *File view*, la quale mostra le variabili (e le costanti nel caso del livello avanzato) definite globalmente. Inoltre, vengono mostrati alcuni dati come il nome del file (modificabile) e una lista di tutte le funzioni definite in quel file. Inoltre,

Figura C.3: SIRENE: esempio di creazione di una funzione.

viene mostrato il codice sorgente associato al file costruito in maniera visuale, quindi anche il codice sorgente di tutte le funzioni. Cliccando sul nome di un file, viene evidenziato, nel codice sorgente, quella parte del codice associata alla funzione corrispondente. È possibile spostarsi in una funzione cliccando su un tab che corrisponde alla funzione o facendo doppio click sul nome di un file nella lista visualizzata.
Infine, quando si vuole visualizzare un file, viene mostrata la Visual Area.

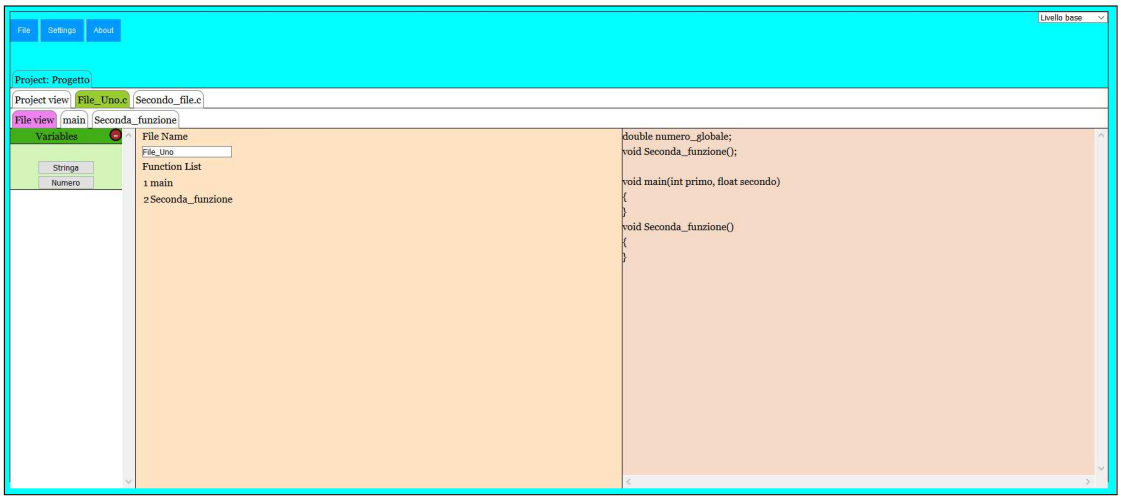


Figura C.4: SIRENE: esempio di visualizzazione di un file.

C.4 Implementazione di una funzione visuale

Un algoritmo visuale, in SIRENE, viene sviluppato definendo le funzioni nella General Area, vedi figura C.3, e sviluppando il comportamento logico di un algoritmo nella Visual Area Code, vedi figura C.1.

C.4.1 Definizione della funzione visuale

Sirene permette di definire le funzioni modificando opportunamente i valori da compilare nella General Area, vedi figura C.3. In questa area è possibile definire il nome di una funzione, il tipo di funzione, dichiarare la funzione come esterna (in questo caso, per la compilazione SIRENE crea anche un file con estensione "*.h*" dove vengono inserite tutte le funzioni rese esterne per altri file), ed inserire i parametri necessari alla funzione.
Per la definizione dei parametri è necessario cliccare sul pulsante "+" (in rosso) adiacente alla dicitura "*Parametri*" (la quale indica il numero di parametri definiti). Viene, quindi, inserita una riga (una per ogni parametro creato) che contiene i campi necessari alla definizione del parametro, quali: il nome, il tipo di parametro, un campo booleano dove indicare se il parametro deve essere passato per valore o per riferimento ("*Pointer*"), un campo booleano che indica se il parametro è un array e, se lo è, un campo dove viene indicata la lunghezza dell'array. È anche possibile eliminare un parametro cliccando nella "x" (in rosso) all'inizio di ogni riga. Contestualmente ad ogni azione che viene eseguita, durante la creazione e modifica di una funzione e dei suoi parametri, viene visualizzato nella Source Code Area anche il codice sorgente della funzione, aggiornato all'ultima azione eseguita.

C.4.2 Sviluppo concettuale della funzione visuale

Per sviluppare il comportamento logico di una funzione, è necessario fare uso dei costrutti iconici messi a disposizione dal linguaggio visuale di SIRENE. Per fare questo, è necessario visualizzare la Visual Code Area e trascinare in essa con il mouse uno dei costrutti iconici mediante il meccanismo "drag&drop". Le icone che possono essere trascinate sono:

- Assegnazione ("*Assignment*").
- Istruzione ("*Instruction*").
- Condizione ("*Condition*").
- I tre Cicli: "*For*", "*While*" e "*Do..While*".
- Ritorno di una funzione ("*Return*").

- Uscita da una funzione ("*Exit*").

Una volta inserita un'icona nella VCA, è possibile spostarla trascinandola nella VCA. Questa operazione è visibile soltanto per chi esegue il trascinamento, vedi figura C.5. Quando l'azione di spostamento è ultimata SIRENE invia un messaggio a tutti i Client connessi per sincronizzare ed aggiornare la posizione di quell'icona. Alla creazione di ogni funzione, viene inserita automaticamente, nella Visual Code Area, l'icona Start, dalla quale il parser iconico inizia ad analizzare il codice iconico. La costruzione del comportamento di una funzione avviene mediante la concatenazione delle icone inserite nella VCA per mezzo di una sequenza di archi. Per collegare due icone si clicca con il tasto destro del mouse sull'icona dalla quale deve partire l'arco. Viene, dunque, mostrato un menù a tendina nella quale sono mostrati i collegamenti possibili per quell'icona, vedi figura C.6. Se nessun collegamento è disponibile allora non viene mostrato nulla. Dopo avere selezionato un'azione è necessario cliccare sull'icona che si vuole collegare. Se l'ultima icona ha già un arco incidente in essa, allora l'azione di collegamento viene annullata. Le azioni di collegamento disponibili sono:

- *Collega Avanti.*
- *Collega in Annidamento.*
- *Collega in Then.*
- *Collega in Else.*

Alcune icone possono avere più tipi di collegamenti. In generale, ogni icona dispone dell'azione 'Collega Avanti', la quale permette di creare un arco tra un'icona ed un'altra.
L'azione *Collega in Annidamento* è disponibile soltanto per le icone che rappresentano i cicli, ovvero le icone quadrate.
Mentre, le azioni *Collega in Then* e *Collega in Else* sono disponibili soltanto per l'icona che rappresenta una ramificazione del flusso, ovvero l'icona che rappresenta la Condizione.
La sequenza degli archi, dunque, rappresenta il flusso iconico e logico di una funzione e permette al parser logico di analizzare correttamente le icone. A causa della possibilità di annidamento o ramificazione del flusso iconico, il colore degli archi viene diversificato in funzione del livello di annidamento o ramificazione, vedi figura C.6 e figura C.7, come già discusso nel paragrafo 4.2.1. Una volta che si è creato un arco, è possibile spostare le icone collegate, semplicemente trascinandole. In questo modo, la direzione e la dimensione degli archi può subire delle modifiche, ovvero gli archi si aggiornano per mantenere collegate visualmente le due icone.

Quest'azione è dinamica, e non necessita di una ulteriore azione da parte dell'utente. Alla fine dello spostamento dell'icona, SIRENE invia un messaggio a tutti i client affinché anch'essi sincronizzino ed aggiornino la posizione dell'icona e di tutti gli archi ad essa incidenti od uscenti.
Le icone inserite nella VCA possono essere eliminate trascinando l'icona sopra il cestino posto nella VCA di ogni funzione. In questo caso, il cestino da bianco diventerà rosso, e tornerà bianco in soli due casi:

1. Quando si rilascia col mouse l'icona trascinata sopra il Cestino, in questo caso viene eliminata l'icona trascinata.

2. Quando è ancora in corso la fase di trascinamento dell'icona e il puntatore del mouse non si trova più sopra il Cestino.

Figura C.5: SIRENE: esempio trascinamento dell'icona *Assegnazione* nella Visual Code Area.

In caso di eliminazione di un'icona, viene eliminato anche il relativo codice sorgente visualizzato nella SCA ed associato all'icona. Inoltre, ogni arco incidente ed uscente da quell'icona viene eliminato. Questo comporta anche la possibilità di eliminare spezzoni di codice sorgente (ma non di codice iconico) se annidati all'icona eliminata, in quanto il parser iconico non riuscirà a raggiungere le icone seguendo il flusso iconico. Questo è un comportamento aspettato.
È possibile anche eliminare soltanto gli archi. Per questo scopo è necessario cliccare col tasto destro sopra un arco specifico e cliccare sulla scritta "*Elimina Arco*" che apparirà.

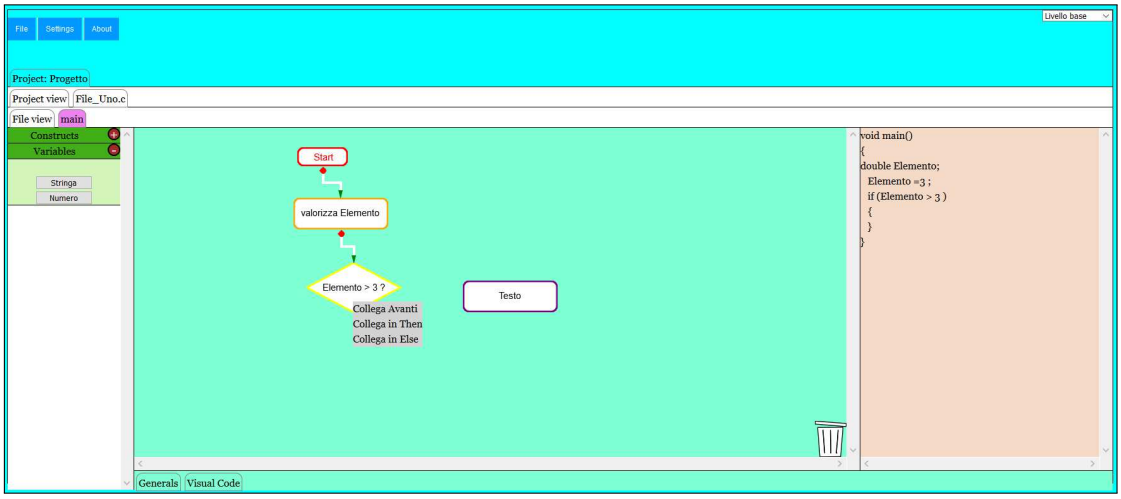


Figura C.6: SIRENE: esempio di collegamento tra due icone.

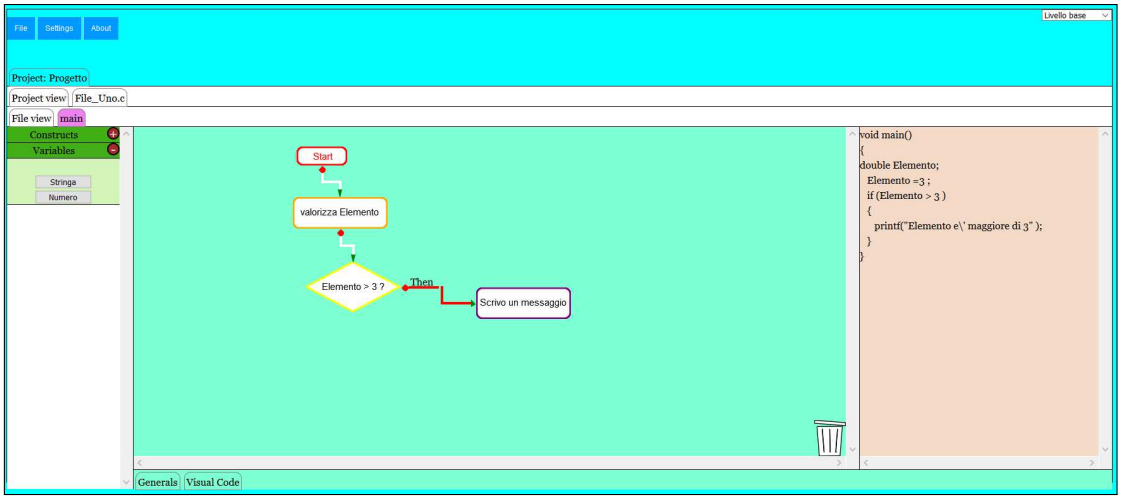


Figura C.7: SIRENE: esempio di cambiamento del colore nel flusso iconico.

C.5 Costruzione delle espressioni ed assegnazioni

Come si è affermato, il linguaggio iconico di SIRENE permette anche di costruire assegnazioni ed espressioni. Per definire le assegnazioni e le espressioni è necessario fare doppio click sulle icone. Esercitata questa azione, apparirà una finestra in accordo al tipo di costruzione che si intende fare. La finestra che permette di creare una assegnazione appare eseguendo un doppio click del mouse sull'icona *Assignment* ed anche durante la definizione del codice associato all'icona ciclica

del *For*.
Invece, le espressioni vengono realizzate eseguendo doppio click sulle icone *Condition*, *Loop While*, *Loop Do..while*, *Return*, ed anche durante la definizione del codice associato all'icona ciclica del *For*. Una espressione fornisce un risultato specifico. Giova ricordare che una condizione altro non è che una espressione il cui risultato è un valore booleano. Per tale motivazione, anche le condizioni sono espressioni ed è per tale motivazione che la visualizzazione della costruzione di una condizione è la stessa di quella di una espressione.

C.5.1 Costruzione di una espressione

La finestra che permette di costruire una espressione viene creata quando si esegue doppio clicl su un'icona, e mostra un insieme di sezioni e campi diversi.
Il primo campo è la definizione della "*Caption*" dell'icona, ovvero la scritta, da valorizzare, che sarà mostrata sopra l'icona associata e definisce l'azione associata cognitivamente all'icona.
Il secondo campo è presente la descrizione, campo "*Description*", questa mostra un testo scritto dall'utente e verrà mostrato quando il mouse rimane sopra l'icona associata.
Il riquadro azzurro, che contiene il cestino, contiene le espressioni che saranno realizzate mediante concatenazione.
Il riquadro in giallo, contiene il codice sorgente associato alla espressione che si sta costruendo, vedi figura C.8.
Diversamente, la colonna a sinistra, di colore grigio, contiene un insieme di regioni che è possibile espandere o comprimere eseguendo, rispettivamente, un click col mouse sul pulsante associato '+' o '-'. Le sezioni sono:

- "*Functions*".
- "*Data*".
- "*Manual data*".
- "*Arithmetic Operators*".
- "*Relational Operators*".
- "*Logical Operators*".
- "*Bitwise Operators*".

Gli elementi scritti in rosso visualizzati in ognuna di queste sezioni possono essere trascinati nel riquadro azzurro per costruire le espressioni.

Espandendo la sezione *Functions*" viene visualizzato un elemento il quale permette di selezionare il tipo di libreria interessata che contiene un insieme di funzioni. Selezionata la libreria, vengono mostrate tutte le funzioni, che sono state create in maniera visuale e quelle che sono di origine standard del linguaggio C.
Nella sezione "*Data*" si trovano tutte le variabili e le costanti che sono state definite dall'utente.
Nella sezione "*Manual Data*", sono raggruppate un insieme di elementi nei quali l'utente può inserire alcuni valori manualmente. È possibile, ad esempio, inserire un valore numerico di tipo *Char*. In questo caso, SIRENE verifica se il valore immesso rientra nel range di valori numerici ammissibili al tipo *Char*: se rientra nel range specifico del *Char* allora SIRENE non corregge il valore immesso dall'utente, diversamente SIRENE esegue una correzione. SIRENE esegue sempre un controllo sui dati inseriti per evitare all'utente di inserire valori fuori range o comunque errati.
Nella sezione "*Arithmetic Operators*" vengono inserite tutte le operazioni aritmetiche, oltre alle parentesi tonde e quadre per permettere la costruzione di espressioni annidate.
Nella sezione "*Relational Operators*" sono stati inseriti tutti gli operatori relazionali.
Troviamo gli operatori logici nella sezione "*Logical Operators*".
Infine, sono presenti gli operatori *bitwise* nella omonima sezione "*Bitwise operators*".
Infine, troviamo i classici bottoni *Ok* e *Cancel*, i quali permettono, rispettivamente, di salvare l'espressione realizzata o di annullare l'operazione di creazione di una espressione. Inoltre, in alto a destra della schermata si trova una "x", nello stile delle finestre di Windows, che permette di annullare l'operazione di creazione di una espressione.

C.5.2 Costruzione di una assegnazione

La finestra che permette di realizzare la creazione di una assegnazione, in maniera visuale, è molto simile a quella che realizza le espressioni. La grande differenza è che al suo interno troviamo una sezione di colore verde nella quale è possibile inserire soltanto una variabile, vedi figura C.9. Questa variabile può essere anche un array, in questo caso è possibile inserire anche una coppia di parentesi quadrate ed inserire, al loro interno, un elemento che indichi l'indice dell'array. Inoltre, nel riquadro verde-acqua, è possibile indicare il tipo di assegnazione che si vuole eseguire:

- La normale assegnazione "=".

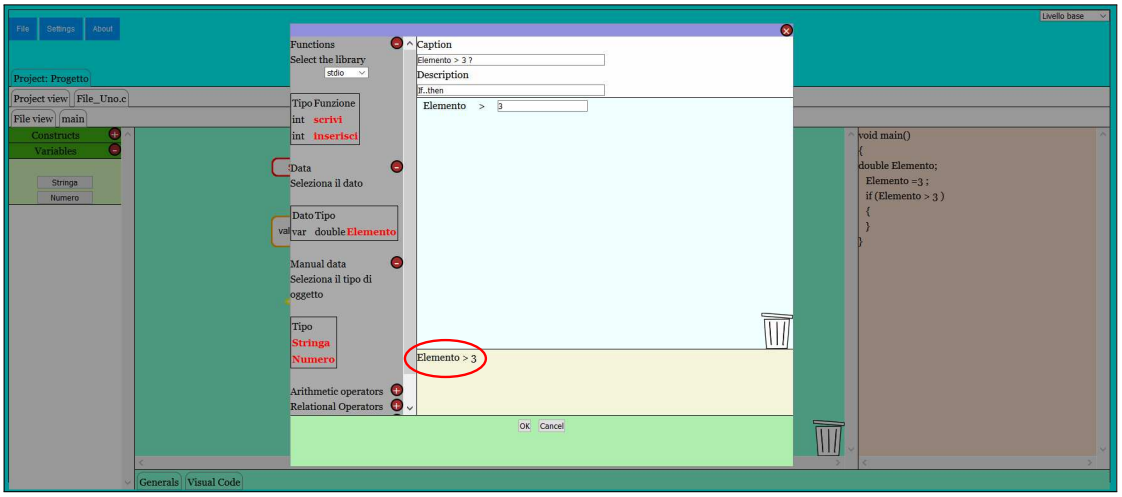


Figura C.8: SIRENE: esempio di costruzione di una condizione.

- L'assegnazione con somma "+=".
- L'assegnazione con sottrazione "-=".
- L'assegnazione con moltiplicazione "*=".
- L'assegnazione con divisione "/=".

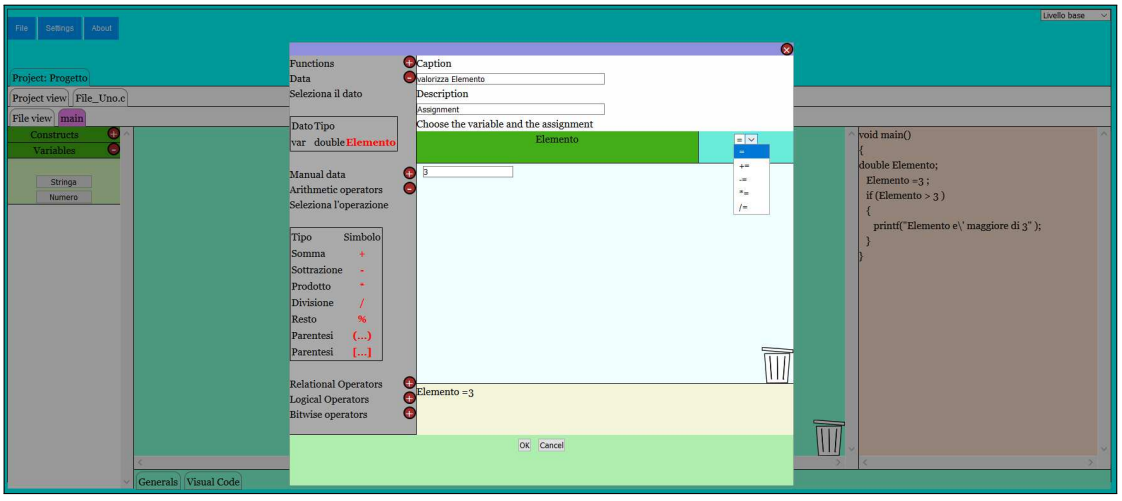


Figura C.9: SIRENE: esempio di costruzione di una espressione.

C.5.3 Caso speciale: uso del costrutto For

Quando viene inserita l'icona *For* nella Visual Code Area, SIRENE inserisce automaticamente un numero intero associato a questa icona, vedi figura C.10. Questo numero è utile per eseguire cicli di codice senza la necessità per l'utente inesperto di conoscere la corretta sintassi del linguaggio. In ogni linguaggio di programmazione, il ciclo del costrutto *For* necessita di: una inizializzazione del contatore, o dei contatori, del ciclo; un confronto tra il contatore, o i contatori, del costrutto *For*, ed alcuni vincoli definiti dall'utente; infine, dell'operazione di modifica del valore del contatore, o dei contatori, questa modifica non necessariamente deve essere un incremento o decremento lineare.
Quando si esegue un doppio click con il mouse sopra un icona *For* SIRENE mostra una finestra piuttosto che un'altra in accordo al livello di utente definito. Nel livello base, SIRENE mostra una sezione *Caption* e *Description*, corrispondenti a quelle descritte per la creazione di una espressione. Inoltre, la finestra mostra anche un campo dove inserire il numero di iterazioni da eseguire. Questo semplifica il concetto di ciclo per gli inesperti. Al variare del valore dei cicli, SIRENE mostra il codice sorgente associato al ciclo *For*, nel quale come contatore viene utilizzato il numero intero associato all'icona *For*, vedi figura C.11.
Nel caso in cui il livello dell'utente fosse "avanzato", vedi figura C.12, SIRENE mostra una finestra molto diversa: vi sono sempre i campi *Caption* e *Description*, come negli altri casi, ma troviamo anche tre sezioni ulteriori:

- La sezione dove vengono visualizzate tutte le inizializzazioni delle variabili utilizzate per il ciclo.
- La sezione dove vengono mostrate tutte espressioni che formano le condizioni che permettono il ripetersi dei cicli.
- Una sezione che formalizza come devono essere modificati i valori dei contatori.

In questa maniera, l'utente è vincolato a costruire correttamente ognuna di queste sezioni. Le inizializzazioni e gli incrementi sono assegnazioni di valori ad una variabile e quindi possono essere gestiti come normali assegnazioni con la finestra che permette di costruire una assegnazione. Analogamente, una condizione è una espressione di tipo booleano e viene realizzata mediante l'uso di una finestra che permette la creazione di una espressione. Ad ognuna di queste sezioni vi è adiacente un riquadro il quale contiene un pulsante "+" ed un cestino i quali permettono, rispettivamente, di aggiungere una assegnazione o una condizione alla sezione corrispondente oppure di cancellarle, in quest'ultimo caso mediante trascinamento dell'elemento. Questi elementi sono: inizializzazioni, condizioni e aggiornamenti dei contatori.

Ogni volta, che si inserisce un elemento in una qualsiasi sezione, SIRENE mostra il codice sorgente in basso.
Completano la finestra della costruzione del *For* i due classici bottoni "Ok" e "Cancel" e il bottone "x" in alto a destra della schermata, come già discusso nei precedenti casi.

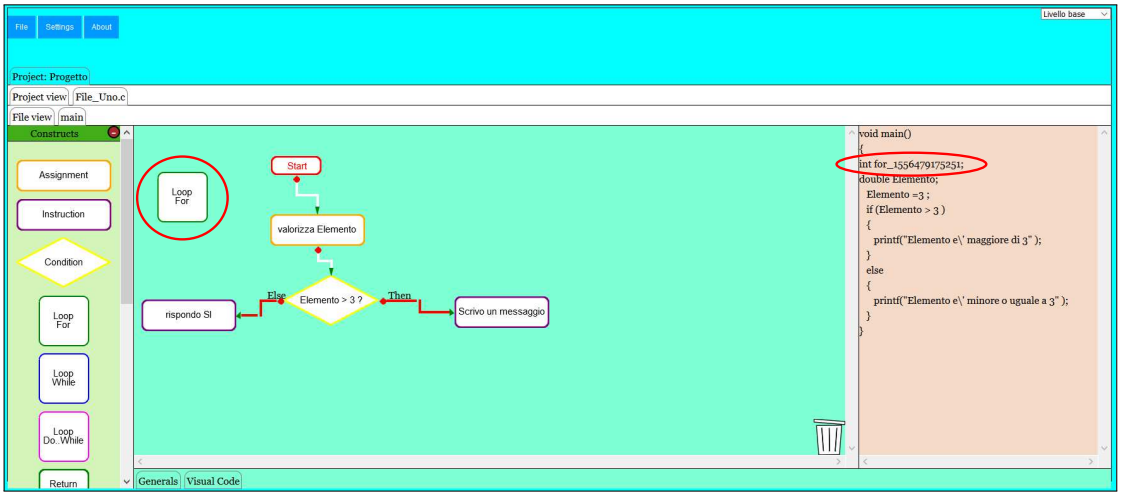


Figura C.10: SIRENE: esempio d'uso del costrutto *For*.

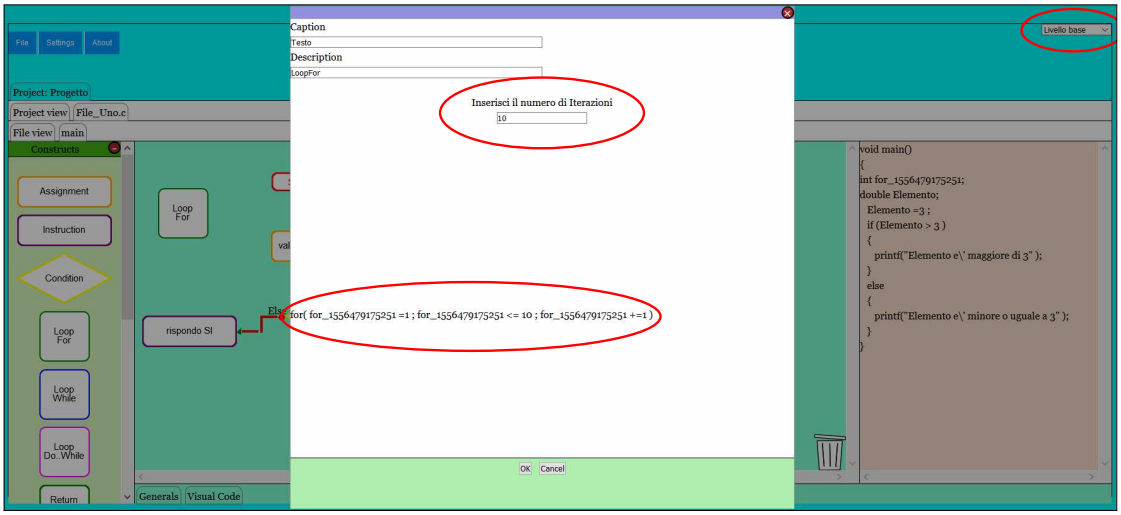


Figura C.11: SIRENE: esempio di visualizzazione della finestra del costrutto *For* per il livello base dell'utente.

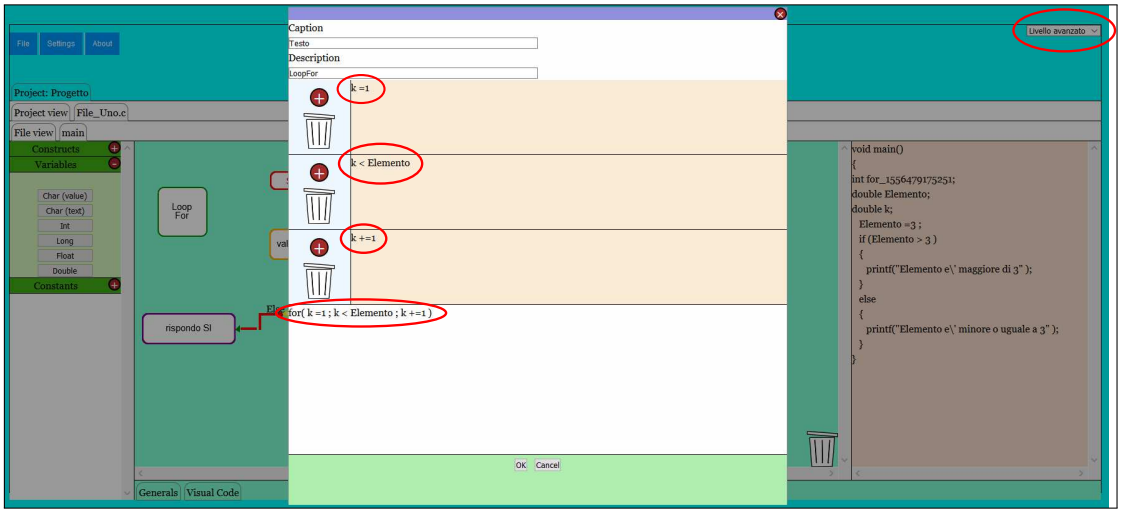


Figura C.12: SIRENE: esempio di visualizzazione della finestra del costrutto *For* per il livello avanzato dell'utente.

C.6 Esecuzione del codice iconico

SIRENE permette di compilare il codice iconico producendo un corretto codice sorgente in linguaggio C, e permette di eseguire il programma compilato. Inoltre, SIRENE permette anche di interagire con il programma in esecuzione mediante una finestra che mostra l'esecuzione del programma in stile "*shell*", come nell'ambiente Windows, o "*terminal*", come in un ambiente linux. Questa caratteristica permette a SIRENE di realizzare una "*web-shell*", o "*web-terminal*".
Per eseguire questa azione, SIRENE mostra una finestra che indica la compilazione del programma in un riquadro nero con 80 colonne. Alla fine della compilazione, il programma viene eseguito e SIRENE mostra, a tutti i client connessi, l'output del programma. Inoltre, SIRENE mostra un campo, inizialmente grigio e non modificabile, nella quale è possibile inserite gli input richiesti dal programma in esecuzione. Quando il programma necessita di ricevere uno o più input, SIRENE modifica il colore, di quest'ultimo campo, da grigio a bianco, specificando così che il programma è in attesa di un input e permette all'utente di inserire alcuni valori all'interno di questo campo. A questo punto, l'utente può interagire col programma inserendo i valori di input richiesti e premere il tasto "*invio*", o cliccare sul bottone "*invia*". Dunque, il campo viene nuovamente posto in stato non modificabile e lo colora nuovamente di grigio.
Per chiudere la finestra, è presente il pulsante "x" in alto a destra. Se si preme questo pulsante quando il programma è in esecuzione allora SIRENE termina il processo del programma eseguito e chiude la finestra del *web-terminal*. Nel caso, in-

vece, in cui il programma ha finito la sua esecuzione SIRENE chiude semplicemente la *web-terminal*.

Appendice D

Installazione di SIRENE

D.1 Esempio di installazione di SIRENE

In questa appendice, viene fornita una spiegazione su come installare le componenti necessarie per realizzare Sirene Server Device Module in un sistema operativo Ubuntu in italiano versione 18.10. Per realizzare SIRENE è necessario installare alcuni strumenti: un webserver, Node.js, Espress.js, Socket.io, Lazarus e la libreria "Indy". Sirene è stato testato usando Node.js versione 6.17.1, Lazarus versione 2.0 e il browser Firefox versione 66.0.3.
Nel seguito vengono descritte le modalità di installazione di queste componenti.

D.2 Installazione di un webserver

Per permettere ai Client di collegarsi con SIRENE, è necessario installare un webserver che permetta l'invio della pagina "**Sirene.html**" ai browser. Per tale finalità è possibile installare, ad esempio, "Apache 2"[1]. Per installare Apache 2 (nel seguito solo Apache) in Ubuntu, è necessario aprire una finestra terminal, cliccando con il tasto destro del mouse sulla "*Scrivania*" di Ubuntu, vedi figura D.1, e digitare i seguenti comandi:

1. sudo apt update
2. sudo apt-get install apache2

Dopo l'esecuzione dei suddetti comandi, viene richiesta la conferma di installazione ed è necessario rispondere con "S". In questa maniera si avvierà l'installazione di Apache. Al termine è necessario eseguire un test di prova, aprendo un

[1]Apache 2 è un server web http sviluppato dalla "The Apache Software Foundation" che ne detiene i diritti. È un webserver open-source e di libero uso ed installazione.

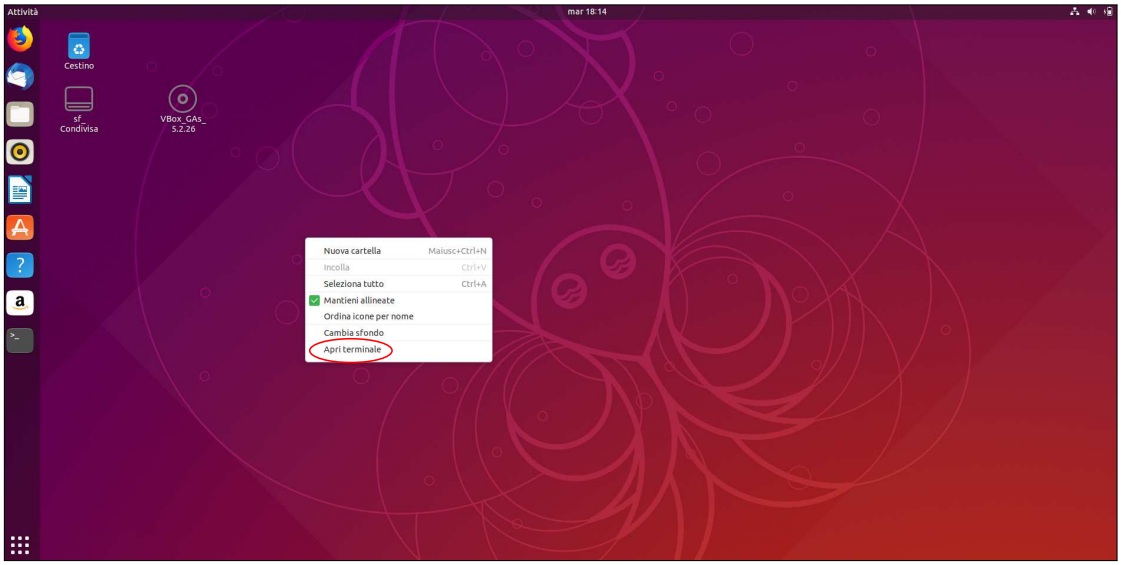


Figura D.1: Apertura di una finestra terminal in Ubuntu.

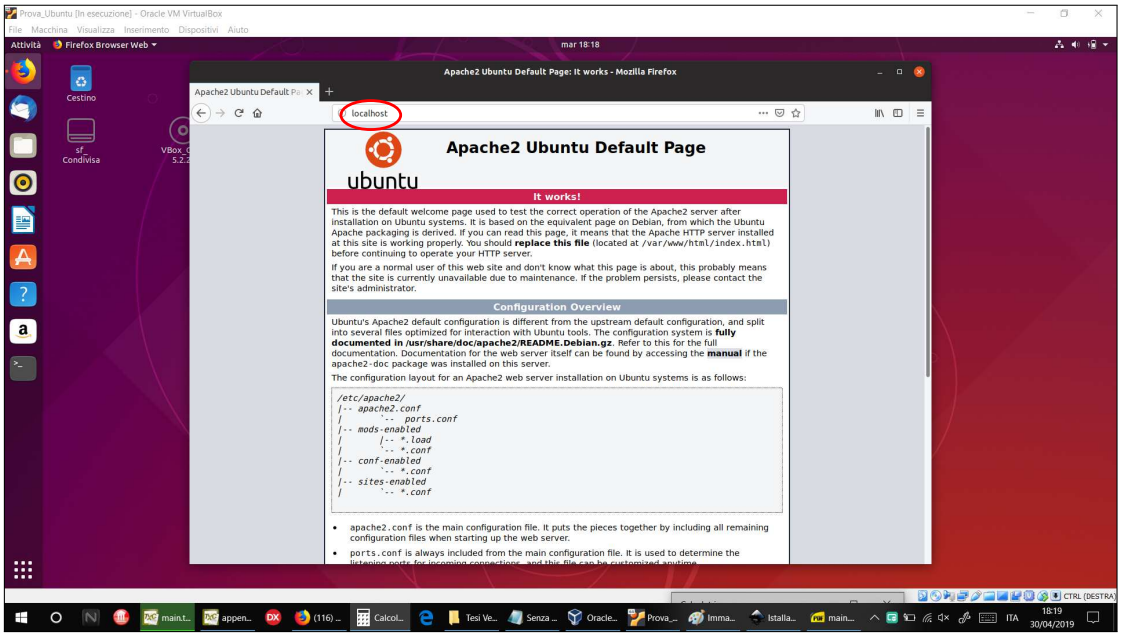


Figura D.2: Verifica della corretta installazione di Apache 2 in Ubuntu.

browser e andando all'indirizzo: "*localhost*". Se l'installazione è andata a buon fine, si aprirà una pagina con su scritto "Apache2 Ubuntu Default Page", vedi figura D.2: l'installazione di Apache è andata a buon fine e si può chiudere il browser. A questo punto è necessario digitare nel terminale i seguenti comandi:

1. cd /var
2. sudo chmod -R 777 www

Fatto questo è necessario andare dentro la cartella "**/var/www/html**" e creare due cartelle:

- "**Exe**".
- "**Temp**".

Quindi, copiare nella cartella "**html**" i seguenti files:

- "**Azioni.js**".
- "**Comuni.js**".
- "**Eventi.js**".
- "**Forme.js**".
- "**Gestione_Assegnazioni.js**".
- "**Gestione_Codice.js**".
- "**Gestione_Espressioni.js**".
- "**Gestione_For.js**".
- "**Gestione_Forme.js**".
- "**Gestione_Progetto.js**".
- "**Gestione_Scheda_Array.js**".
- "**Gestione_Schede.js**".
- "**Gestione_Terminale.js**".
- "**index.js**".
- "**Menu.js**".
- "**Rete.js**".
- "**Sirene.html**".

Infine, copiare nella cartella "**html**" le seguenti cartelle e i loro files:

- "**CSS**".
- "**img**".

A questo punto, si sono inseriti tutti i files necessari per il corretto funzionamento di SIRENE, incluso il componente SIRENE Client Device Module, ed è necessario eseguire nuovamente i seguenti comandi nel terminale:

1. cd /var
2. sudo chmod -R 777 www

Per il corretto funzionamento di SIRENE è necessario modificare due files. Nel file "**Rete.js**" è necessario modificare la settima riga

socket = io.connect('http://192.168.178.31:3701');

nella scritta "*192.168.178.31*" bisogna inserire l'indirizzo IP del server e salvare il file. La stessa azione è necessario eseguirla nella 45 esima ri ga del file "**Sirene.html**", ovvero nella scritta "*192.168.178.31*" bisogna inserire l'indirizzo IP del server e salvare il file.
A questo punto, sarà necessario installare Node.js e gli altri moduli.

D.3 Installazione di Node.js, Espress.js, Socket.io e realizzazione di J-Communicator

Per installare Node.js, Espress.js e Socket.io in Ubuntu, è necessario aprire un terminal e digitare in ordine i seguenti comandi:

1. sudo apt update
2. sudo apt install curl
3. curl -o- `https://raw.githubusercontent.com/creationix/nvm/v0.33.11/install.sh` | bash

Al termine di questa sequenza di comandi si dovrebbe visualizzare una scritta simile alla figura D.3:

A questo punto bisogna chiudere e riaprire il terminal, per rendere permanenti le modifiche, ed eseguire i seguenti comandi:

1. cd /var/www/html

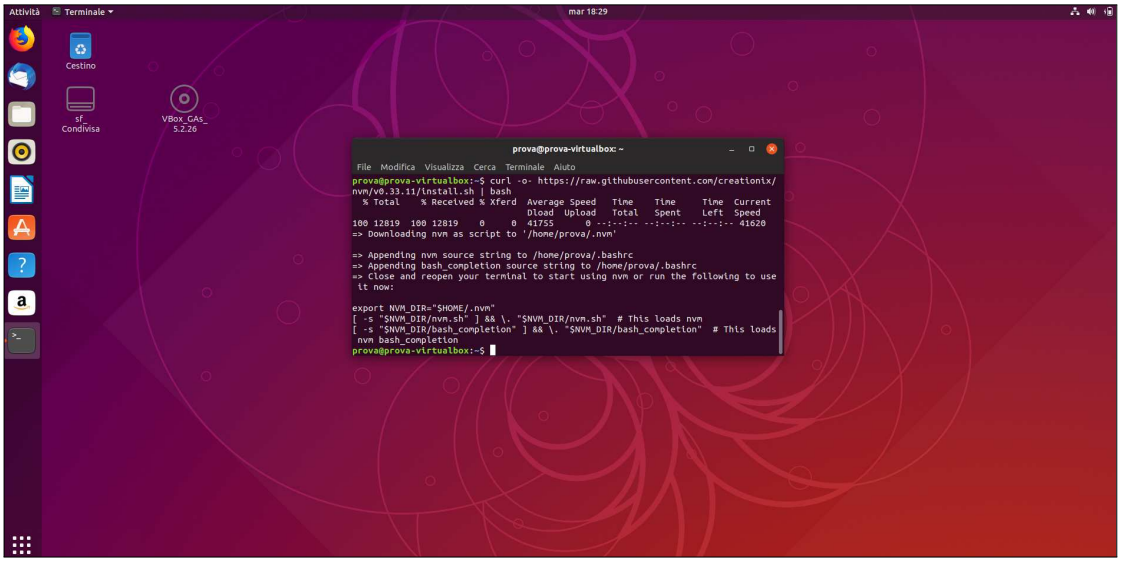


Figura D.3: Esempio di corretta installazione.

2. nvm install 6.17.1
3. npm install socket.io express --save

L'installazione di Node.js, Espress.js e Socket.io è terminata. È possibile verificare se l'installazione è andata a buon fine digitando il seguente comando

node -v

che dovrebbe produrre il seguente risultato

v6.17.1

Con questa installazione si è realizzato il componente J-Communicator del SIRENE Server Device Module. Per completare la realizzazione di SIRENE è necessario installare Lazarus.

D.4 Installazione di Lazarus 2.0, della libreria "Indy" e realizzazione di J- Communicator

Per installare Lazarus 2.0 è necessario creare una cartella "**Lazarus**" nella *Scrivania* di Ubuntu ed andare con un browser alla pagina:

`https://sourceforge.net/projects/lazarus/files/LazarusLinuxamd64DEB/Lazarus2.0.0/`

Da questa pagina internet è necessario scaricare i seguenti file nella cartella appena creata:

1. "**fpc-src_3.0.4-2_amd64.deb**"
2. "**fpc-laz_3.0.4-1_amd64.deb**"
3. "**lazarus-project_2.0.0-0_amd64.deb**"

Scaricati i files è necessario installarli cliccando (singolarmente nell'ordine in cui sarà indicato a breve) con il tasto destro sul file e selezionando la voce "*Apri con installa software*", vedi figura D.4. Si aprirà una schermata nella quale bisognerà cliccare sul bottone "*Installa*". L'ordine di installazione dei files è:

1. "**fpc-src_3.0.4-2_amd64.deb**"
2. "**fpc-laz_3.0.4-1_amd64.deb**"
3. "**lazarus-project_2.0.0-0_amd64.deb**"

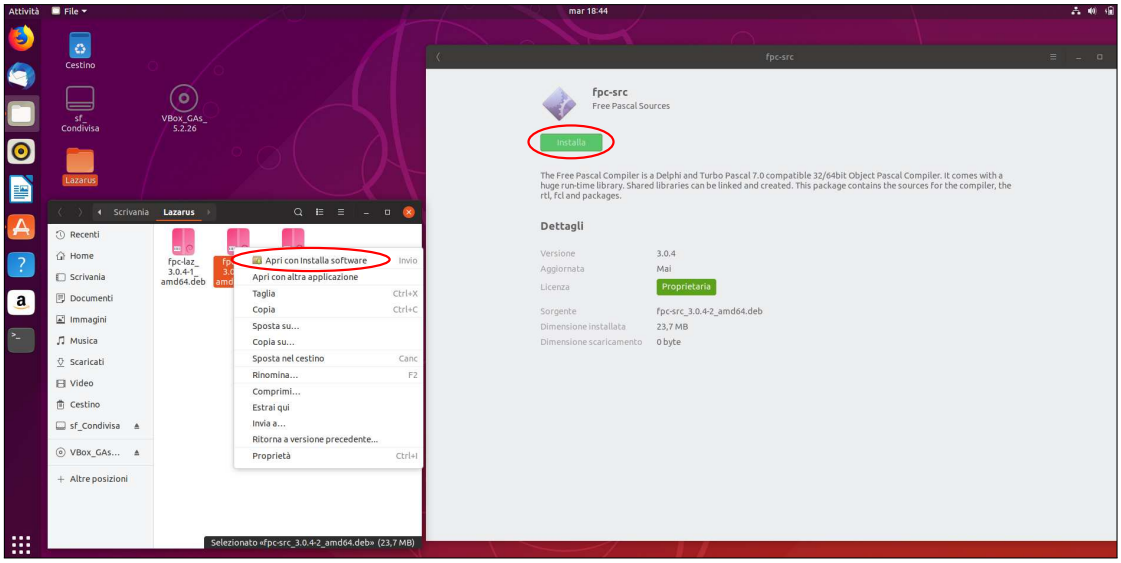


Figura D.4: Installazione di Lazarus.

Se tutto è andato a buon fine Lazarus 2.0 sarà installato. Per aprire Lazarus sarà necessario aprire un terminal e digitare la scritta "*startlazarus*". Si aprirà una schermata nella quale è necessario cliccare (solo per questa volta) il bottone "*Avviare l'IDE*", vedi figura D.5. Aperto Lazarus sarà necessario installare la libreria "**Indy10**". Per fare ciò si vada col mouse sul Menù "*File*" e selezionare

la voce "*Chiudi tutti i file*". Quindi, andare nel Menù "*Pacchetto*" e selezionare la voce "*Online Package Manager*". Si aprirà una schermata nella quale sarà necessario cercare la libreria "**Indy10**", selezionarla e cliccare col mouse sull'icona "*Install*" in basso nella schermata. Si aprirà un piccolo menù nel quale si deve selezionare la voce "*From repository*", vedi figura D.6. Si avvierà l'installazione della libreria "**Indy10**". Al termine dell'installazione, si aprirà una schermata che chiederà "*Vuoi ricostruire Lazarus con il profilo: IDE Normale?*". Cliccare sul bottone "*Si*". A questo punto, Lazarus si ricostruirà ed al suo termine si chiuderà e sarà nuovamente eseguito automaticamente. La libreria "**Indy10**" è installata in Lazarus.

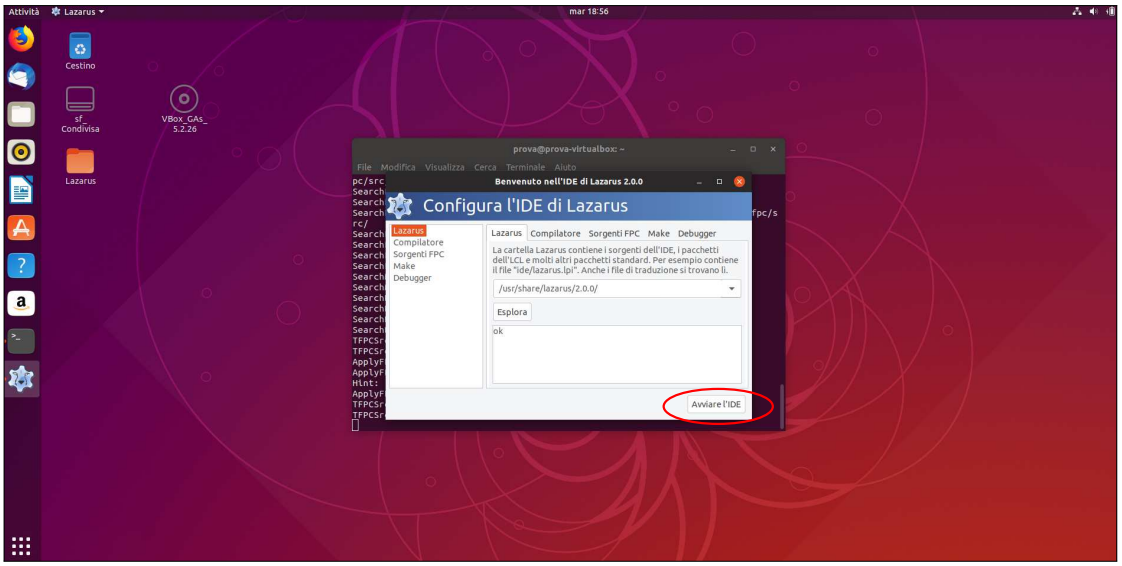


Figura D.5: Primo avvio di Lazarus.

Per creare l'applicazione J-Manager, si vada col mouse sul Menù "*File*" e selezionare sulla voce "*Chiudi tutti i file*". Quindi, creare sulla *Scrivania* di Ubuntu una cartella "**J-Manager**" e copiare dentro questa cartella i seguenti file di SIRENE:

- "**project1.ico**".
- "**project1.lpi**".
- "**project1.lpr**".
- "**project1.lps**".
- "**project1.res**".

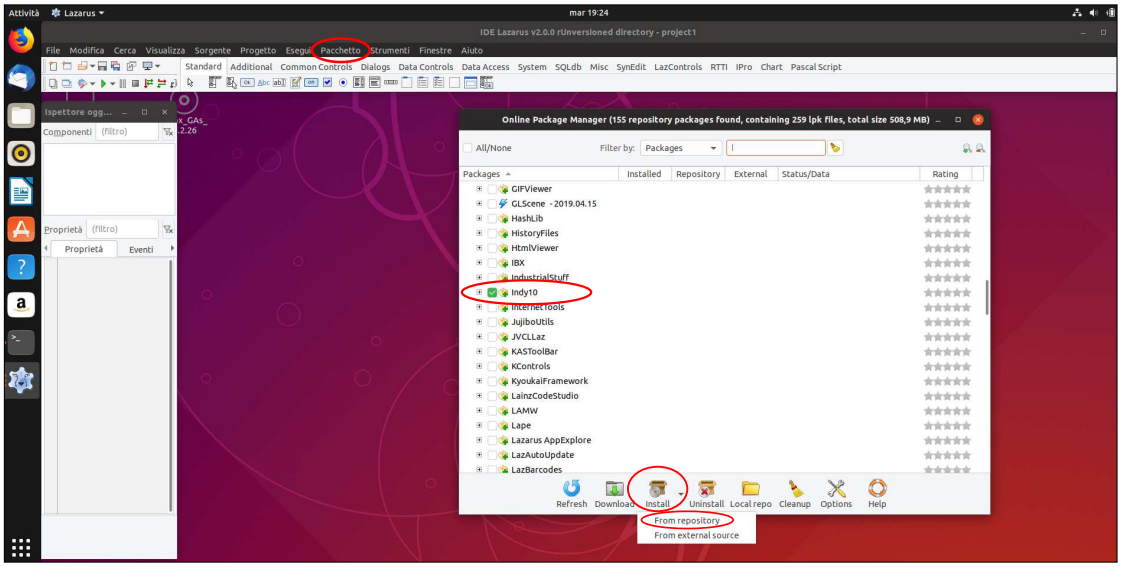


Figura D.6: Installazione della libreria *Indy10*.

- "**unit1.lfm**".
- "**unit1.pas**".

Cliccare due volte sul file "**project1.lpr**" per fare automaticamente tutto il progetto in Lazarus. Andare sul Menù "*Esegui*" e selezionare la voce "*Compila*". Finita la compilazione, tornare sul Menù "*Esegui*" e selezionare la voce "*Costruisci*". Se l'operazione è andata a buon fine si troverà nella cartella "**J-Manager**" il programma "**J-Manager**". A questo punto, copiare il programma "**J-Manager**" nella cartella "**/var/www/html**", ovvero dove sono stati copiati tutti i file "***.js**" e il file "**Sirene.html**".

D.5 Esecuzione di SIRENE

Per eseguire SIRENE, è necessario procedere in ordine con i seguenti passi

1. eseguire prima J-Communicator;
2. eseguire J-Manager.

Per eseguire J-Communicator, andare nella cartella dove sono tutti i file Javascript ed il file "**Sirene.html**", ed apire una finestra terminal da quella posizione. Digitare quindi il comando

node index.js

Se l'esecuzione è andata a buon fine si vedrà visualizzato il seguente messaggio

Listening on port 3701

A questo punto, eseguire il programma "**J-Manager**" cliccando due volte sopra di esso. Si aprirà una finestra, quindi cliccare sul bottone "*avvia server*", vedi figura D.7. Se tutto è andato a buon fine si visualizzerà nell'area in bianco un messaggio simile a

server attivo. Porta: 45347

Il numero della porta può variare. SIRENE è attivo e può essere utilizzato da qualunque client.
Il client per collegarsi necessiterà soltanto di aprire un browser ed andare alla pagina simile a

http://192.168.178.31/Sirene.html

dove al posto della dicitura "*192.168.178.31*" è necessario inserire l'indirizzo IP del server che ospita SIRENE.
A questo punto è possibile utilizzare SIRENE come spiegato nell'appendice C.

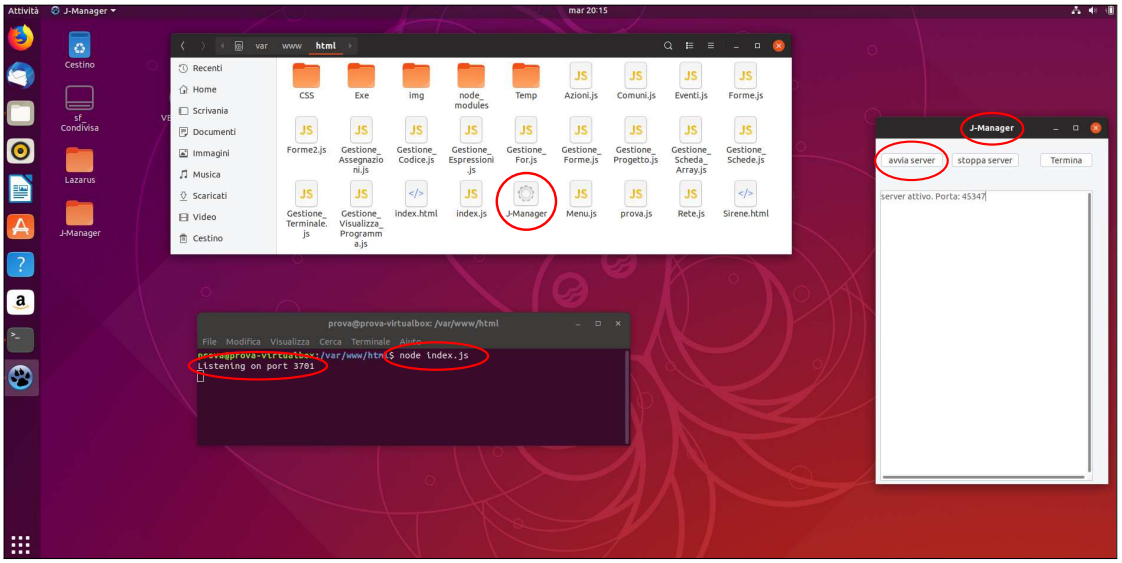


Figura D.7: Avvio di SIRENE.

Bibliografia

[1] Guido Averna. Sirene - framework sull'arte della programmazione. In *Giornate di studio dell'Insegnante di MATematica (GIMAT 17)*, pages 163–164. Benedetto Di Paola, G.R.I.M. - Gruppo di Ricerca sull'Insegnamento/Apprendimento delle Matematiche, 2017.

[2] Guido Averna. Sirene, framework per l'insegnamento della matematica. In *Giornate di studio dell'Insegnante di MATematica (GIMAT 18)*. Benedetto Di Paola, G.R.I.M. - Gruppo di Ricerca sull'Insegnamento/Apprendimento delle Matematiche, 2018.

[3] Guido Averna, Biagio Lenzitti, Davide Taibi, and Domenico Tegolo. Iconic framework for cooperative coding. In *Proceedings of the 19th International Conference on Computer Systems and Technologies*, pages 165–170. ACM, 2018.

[4] Guido Averna, Biagio Lenzitti, and Domenico Tegolo. An iconic framework for learning the art of programming. In *INTERNATIONAL CONFERENCE ON E-LEARNING "e-Learning'16"*, volume 1, pages 73–79. Daniela Chudà, Leon Rothkranz, Angel Smrikarov, Tzvetomir Vassilev, Stoyanka Smrikarova, Yuksel Aliev, 2016.

[5] Guido Averna, Biagio Lenzitti, and Domenico Tegolo. E-learning and art of programming: a context oriented to. In *9th annual International Conference on Education and New Learning Technologies (EDULEARN 17)*, pages 8051–8056. L. Gómez Chova, A. López Martínez, I. Candel Torres, IATED Academy, 2017.

[6] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[7] Alan Borning. Thinglab: an object-oriented system for building simulations using constraints. In *Proceedings of the 5th international joint conference on*

Artificial intelligence-Volume 1, pages 497–498. Morgan Kaufmann Publishers Inc., 1977.

[8] Alan Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):353–387, 1981.

[9] Marat Boshernitsan and Michael Sean Downes. *Visual programming languages: A survey*. Citeseer, 2004.

[10] Margaret M Burnett. Seven programming language issues. In *Visual object-oriented programming*, pages 161–181. Manning Publications Co., 1995.

[11] Margaret M. Burnett and Marla J. Baker. A classification system for visual programming languages. *Journal of Visual Languages and Computing*, 5(3):287–300, 1994.

[12] Martin C Carlisle, Terry A Wilson, Jeffrey W Humphries, and Steven M Hadfield. Raptor: a visual programming environment for teaching algorithmic problem solving. *Acm Sigcse Bulletin*, 37(1):176–180, 2005.

[13] Philip T Cox, FR Giles, and Tomasz Pietrzykowski. Prograph: a step towards liberating programming from textual conditioning. In *Visual Languages, 1989., IEEE Workshop on*, pages 150–156. IEEE, 1989.

[14] Ryan Henson Creighton. *Unity 3D game development by example: A Seat-of-your-pants manual for building fun, groovy little games quickly*. Packt Publishing Ltd, 2010.

[15] Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration*. MIT press, 1993.

[16] WF Fizner and Laura Gould. Rehearsal world: Programming by rehearsal, 1993.

[17] Howard Gardner. *Frames of mind: The theory of multiple intelligences*. Hachette UK, 2011.

[18] Paul Gestwicki and Khuloud Ahmad. App inventor for android with studio-based learning. *Journal of Computing Sciences in Colleges*, 27(1):55–63, 2011.

[19] Janice Glasgow, N Hari Narayanan, and B Chandrasekaran. *Diagrammatic reasoning: Cognitive and computational perspectives*. Mit Press, 1995.

[20] Ephraim P. Glinert and Steven L. Tanimoto. Pict: An interactive graphical programming environment. *Computer*, (11):7–25, 1984.

[21] Jeff Gray, Hal Abelson, David Wolber, and Michelle Friend. Teaching cs principles with app inventor. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 405–406. ACM, 2012.

[22] Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

[23] Thomas RG Green and Marian Petre. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics). GC van der Veer, MJ Tauber, S. Bagnarola and M. Antavolits. Rome, CUD*, pages 167–180. Citeseer, 1992.

[24] Thomas RG Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[25] Thomas RG Green, Marian Petre, and RKE Bellamy. Comprehensibility of visual and textual programs: A test of superlativism against the'match-mismatch'conjecture. *ESP*, 91(743):121–146, 1991.

[26] Lynn Helding. Howard gardner's theory of multiple intelligences. *Journal of Singing*, 66(2):193, 2009.

[27] Gwo-Jen Hwang. A conceptual map model for developing intelligent tutoring systems. *Computers & Education*, 40(3):217–235, 2003.

[28] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik: a visual programming environment. In *ACM SIGPLAN Notices*, volume 23, pages 176–190. ACM, 1988.

[29] Olivera Iskrenovic-Momcilovic. Choice of visual programming language for learning programming. *International Journal of Computers*, 2, 2017.

[30] Robert J. K. Jacob. A state transition diagram language for visual programming. *Computer*, (8):51–59, 1985.

[31] Kathleen Jensen and Niklaus Wirth. *PASCAL user manual and report: ISO PASCAL standard*. Springer Science & Business Media, 2012.

[32] Dona M Kagan. Implication of research on teacher belief. *Educational psychologist*, 27(1):65–90, 1992.

[33] Brian Karis and Epic Games. Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice*, pages 621–635, 2013.

[34] Amir A Khwaja and Joseph E Urban. Adaptation and modification of nassi-shneiderman charts to represent descartes specifications visually. In *Rapid System Prototyping, 1992. Shortening the Path from Specification to Prototype, 1992 International Workshop on*, pages 188–201. IEEE, 1992.

[35] Jill H Larkin. Display-based problem solving. *Complex information processing: The impact of Herbert A. Simon*, pages 319–341, 1989.

[36] Jill H Larkin and Herbert A Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 11(1):65–100, 1987.

[37] John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. Scratch: a sneak preview [education]. In *Creating, connecting and collaborating through computing, 2004. Proceedings. Second International Conference on*, pages 104–109. IEEE, 2004.

[38] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[39] Lil Mohan and Rangasami L. Kashyap. A visual query language for graphical interaction with schema-intensive databases. *IEEE Transactions on Knowledge & Data Engineering*, (5):843–858, 1993.

[40] H Narayanan. A study of diagrammatic reasoning from verbal and gestural data. In *Proceedings of the 16th Annual Conference of the Cognitive Science Society*, pages 652–657. Lawrence Erlbaum Associates, 1994.

[41] N Hari Narayanan and Mary Hegarty. On designing comprehensible interactive hypermedia manuals. *International journal of human-computer studies*, 48(2):267–301, 1998.

[42] N Hari Narayanan and Roland Hübscher. Visual language theory: Towards a human-computer interaction perspective. In *Visual language theory*, pages 87–128. Springer, 1998.

[43] N Hari Narayanan, M Suwa, and H Motoda. Diagram-based problem solving: The case of an impossible problem. In *Proceedings of the 17th annual conference of the Cognitive Science Society*, pages 206–211. Lawrence Erlbaum Hillsdale, 1995.

[44] N Hari Narayanan, Masaki Suwa, and Hiroshi Motoda. How things appear to work: Predicting behaviors from device diagrams. In *AAAI*, pages 1161–1167, 1994.

[45] N Hari Narayanan, Masaki Suwa, and Hiroshi Motoda. Behavior hypothesis from schematic diagrams, 1995.

[46] Joseph D Novak. Concept mapping: A useful tool for science education. *Journal of research in science teaching*, 27(10):937–949, 1990.

[47] Joseph D Novak. *Learning, creating, and using knowledge: Concept maps as facilitative tools in schools and corporations*. Routledge, 2010.

[48] Seymour Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., 1980.

[49] Shaileen Crawford Pokress and José Juan Dominguez Veiga. Mit app inventor: Enabling personal mobile computing. *arXiv preprint arXiv:1310.2830*, 2013.

[50] Georg Raeder. A survey of current graphical programming techniques. *Computer*, (8):11–25, 1985.

[51] Alexander Repenning and Tamara Sumner. Agentsheets: A medium for creating domain-oriented visual languages. *Computer*, 28(3):17–25, 1995.

[52] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[53] Samir Ribic and Maida Beganovic. Cross compilation under lazarus ide. In *Telecommunications Forum (TELFOR), 2013 21st*, pages 1007–1010. IEEE, 2013.

[54] David Canfield Smith. Pygmalion: a creative programming environment. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1975.

[55] David Canfield Smith. Pygmalion: An executable electronic blackboard. In *Watch what I do*, pages 19–48. MIT Press, 1993.

[56] Ivan E Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*, pages 6–329. ACM, 1964.

[57] Stefan Tilkov and Steve Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[58] Jeffrey Travis and Jim Kring. *LabVIEW for everyone: graphical programming made easy and fun*. Prentice-Hall, 2007.

[59] Carol G Williams. Using concept maps to assess conceptual knowledge of function. *Journal for Research in Mathematics Education*, pages 414–421, 1998.

[60] Jeannette M Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.

[61] David Wolber. App inventor and real-world motivation. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 601–606. ACM, 2011.

[62] Guangtong Xue, Qiliang Yang, and Jianchun Xing. A survey of graphical programming language and its applications in intelligent buildings. In *Chinese Automation Congress (CAC), 2017*, pages 6822–6828. IEEE, 2017.