Alma Mater Studiorum · Università di Bologna
Campus di Cesena

SCUOLA DI SCIENZE
Corso di Laurea in Ingegneria e Scienze Informatiche

Progettazione e Sviluppo di un Transpiler da Scratch/Snap! a Wiring/Arduino: il Progetto Smart T-Shirt come Caso di Studio

Elaborato nell'ambito del corso
di
Programmazione di Sistemi Embedded

Relatore:
Prof. ALESSANDRO RICCI

Presentata da:
PAOLO VENTURI

Correlatori:
Dott.ssa LAURA TARSITANO
Ing. ANGELO CROATTI

III Sessione di Laurea
Anno Accademico 2016-2017

PAROLE CHIAVE

Pensiero Computazionale
Progetto COGITO
Scratch/Snap!
Progetto Smart T-Shirt
Sistemi Embedded
Arduino
Transpiler
Progetto Snap2ino

A ciò che di più caro ho nel mondo:
la mia famiglia, i miei amici ...

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Negli ultimi anni il termine *Pensiero Computazionale* è andato sempre più diffondendosi nel contesto dell'istruzione internazionale e nazionale, quale attitudine e processo mentale attraverso il quale poter risolvere problemi di varia natura e complessità nella vita quotidiana, e non solo, attraverso metodi e strumenti propri del ragionamento informatico-algoritmico. Le ragioni alla base di tale diffusione, e dell'importanza di insegnare questo approccio mentale fin dall'infanzia, sono molteplici, così come i suoi sostenitori. Da una parte ci sono le grandi istituzioni. Queste vedono nell'educazione digitale, nel coding e nel pensiero computazionale gli strumenti attraverso i quali sviluppare nei giovani le abilità alla base di quell'atteggiamento di *lifelong learning* che sarà sempre più necessario avere in futuro. Un futuro caratterizzato da prospettive lavorative e professionali sempre più incerte, nel quale sarà impensabile poter affrontare la quotidianità con le sole conoscenze pregresse, e nel quale rivestirà vitale importanza lo sapersi destreggiare anche nell'ambito di discipline differenti da quella di provenienza e specializzazione. Dall'altra, invece, c'è chi vede nel pensiero computazionale la chiave per aprire nuove prospettive a livello pedagogico, relativamente a come i bambini vedono se stessi e il mondo che li circonda, e come strumento di crescita ed espressione personale formidabile, grazie al quale rendere ai bambini più facile e naturale esprimere se stessi, comunicare e interagire con gli altri, porsi domande e imparare a gestire problemi via via sempre più complessi. Tra questi ultimi, in particolare, non posso non ci tare tutti coloro dietro al progetto COGITO: insegnanti e Dirigente Scolastico della Scuola Primaria *Carducci* di Cesena, il Prof. Alessandro Ricci, l'Ing. Angelo Croatti, e la Dott.ssa Laura Tarsitano. Nato proprio con lo scopo di diffondere il pensiero computazionale in alcune classi 4e e 5e Cesenati, COGITO, facendo tesoro degli insegnamenti del padre Seymour Papert e del suo erede Mitchel Resnick, attraverso l'utilizzo del calcolatore, del *coding* e di ambienti digitali protetti (i *Micromondi*), mira a favorire l'esplorazione e l'apprendimento di concetti e competenze interdisciplinari, nonché l'ideazione e la creazione di

artefatti informatici digitali. A tal proposito, alla vigilia del terzo anno di attività, si è dato il via a un progetto "spin-off", *Smart T-Shirt*, con l'obiettivo di includere fra i contenuti offerti da COGITO quello di veder portato nel mondo reale quanto realizzato dai bambini sui tablet attraverso Snap!. In particolare, come suggerisce il nome, *Smart T-Shirt* vuole rendere possibile ai bambini poter programmare la propria "maglietta intelligente", ossia dotata di una matrice di led sul petto, attraverso l'ambiente Snap! e il *Micromondo della Matrice*. Di qui la necessità di consentire la programmazione del dispositivo embedded mediante il quale poter controllare la matrice di led, attraverso il medesimo linguaggio e ambiente che i bambini hanno imparato a conoscere e utilizzare nelle varie attività di *coding* e pensiero computazionale svolte in questi anni. Data la non esistenza di un *framework* in grado di rispondere appieno alle esigenze del progetto *Smart T-Shirt*, ha origine, a questo punto, un secondo progetto "spin-off": *il transpiler Snap2ino*. Nato con l'obiettivo di consentire la conversione di quanto definito attraverso il *Micromondo della Matrice* in uno sketch Wiring, e controllare così facendo la matrice di led fisica attraverso Arduino Uno, *Snap2ino*, a seguito di un'opportuna generalizzazione del suo utilizzo, rappresenta, in commistione con Snap!, un framework all'avanguardia circa la possibilità di programmare qualsiasi sistema embedded basato su Wiring, in maniera semplice, veloce e intuitiva. Attraverso il supporto ad aspetti avanzati come l'esecuzione di task concorrenti e gli eventi, *Snap2ino* consente, inoltre, la modellazione di veri e propri *sistemi ad agenti*, modellati come Sprite comunicanti tra loro, nonché il supporto all'integrazione di librerie Wiring esistenti, di sensori e attuatori compatibili con la piattaforma Arduino e la possibilità di definire, di propria mano, cosa si vuole che un dato blocco Snap! faccia una volta convertito. Tali caratteristiche, e le potenzialità che da esse derivano, fanno di questo progetto un vero e proprio framework aperto e adatto all'utilizzo nei più svariati contesti (*making, atelier digitali/creativi, coding e pensiero computazionale*), rendendolo, quindi, potenzialmente significativo per l'intera comunità internazionale in generale.

La presente tesi di laurea affronta, quindi, il progetto e lo sviluppo del *transpiler Snap2ino*, nato nel contesto del progetto *Smart T-Shirt* che ne rappresenta il caso di studio, con l'obiettivo di poter convertire progetti realizzati attraverso il linguaggio e l'ambiente Snap! in progetti basati sul framework Wiring, con riferimento al caricamento e all'esecuzione di questi sulla nota piattaforma a microcontrollore Arduino. Composta da due parti, la tesi vede affrontati, nella prima, il contesto applicativo e quello tecnologico nei quali collocare il caso di studio *Smart T-Shirt*, nella seconda, invece, gli aspetti progettuali inerenti a quest'ultimo e al progetto *Snap2ino*.

In particolare, il Capitolo 1 tratta il concetto di *Pensiero Computazionale*, a partire dal significato di questo termine e dall'importanza dell'insegnamento di questo fin dall'infanzia, di colui che ne è considerato il "padre", Seymour Papert, dell'erede di questo, Mitchel Resnick, e dei contributi apportati da questi nell'ambito dell'istruzione attraverso l'utilizzo delle tecnologie. Viene infine descritto il progetto COGITO, nato a Cesena con l'intento di applicare e portare avanti gli studi di Papert e Resnick, rappresentante il contesto applicativo del caso di studio. Il Capitolo 2, riguardante invece il contesto tecnologico, tratta le tecnologie e i dispositivi attraverso i quali poter venire in contro alle esigenze del progetto *Smart T-Shirt*, nonché le piattaforme esistenti utilizzate nel medesimo contesto applicativo, l'*Educational*, tra le quali poter collocare il caso di studio. Segue poi la seconda parte, aperta dal Capitolo 3. In questo è descritto il caso di studio *Smart T-Shirt*, gli obiettivi di questo, la fase di progettazione, il team e il mio contributo all'interno di questo. Infine, viene trattato, negli ultimi due capitoli, il progetto *Snap2ino*, rappresentante il vero contributo progettuale insito in questa tesi di laurea, sviluppato seguendo un approccio incrementale e attraverso la definizione di due successivi prototipi: il primo, specifico per la conversione del *Micromondo della Matrice*, trattato nel Capitolo 4 mentre il secondo, più generico e in grado di convertire potenzialmente qualsiasi progetto Snap!, trattato invece nel Capitolo 5. Completa la tesi una sezione dedicata alle conclusioni generali e agli sviluppi futuri.

Una nota che è importante fare, prima di potersi inoltrare nella lettura della tesi, riguarda il ruolo di *Scratch* in questa. Infatti, nonostante il titolo faccia chiaro riferimento a questo, oltre che a Snap!, Scratch è tuttavia inserito esclusivamente per via della maggiore "visibilità" rispetto al primo, grazie alla quale rendere immediatamente chiaro ciò di cui si andrà a parlare, ossia *l'abilitazione all'utilizzo di linguaggi visuali a blocchi per la programmazione di sistemi embedded*. Nonostante, quindi, la tesi, e il contributo progettuale ad essa relativo, sia incentrata esclusivamente su Snap!, nella sezione relativa agli sviluppi futuri verrà comunque sottolineato quanto circoscritte e poco complicate possano essere le modifiche da apportare al transpiler realizzato, al fine di renderlo compatibile anche con Scratch.

Parte I

Contesto applicativo e tecnologico del caso di studio

Capitolo 1

Contesto applicativo: dal Pensiero Computazionale al progetto COGITO

In questo primo capitolo viene trattato quello che è il contesto applicativo in cui si colloca il caso di studio Smart T-Shirt.

Si parte innanzitutto da quello che è il concetto di *Pensiero Computazionale* (1.1): da chi e quando è stato introdotto, che significato ha oggi e, soprattutto, che potenziale ha per la scuola del futuro. Sono in seguito introdotti due dei principali fautori del pensiero computazionale, Seymour Papert e Mitchel Resnick (1.2), il loro contributo a livello teorico (il *Costruzionismo* di Papert e il framework del *Lifelong Kindergarden* di Resnick) e tecnologico: i linguaggi LOGO e Scratch (1.3).

Infine, viene presentato il progetto COGITO (1.4), a partire dal quale è in seguito nato il progetto Smart T-Shirt, e che, nato a Cesena partendo proprio dai concetti definiti dal *Costruzionismo* di Papert e dall'approccio introdotto da Resnick (*Learn to code, code to learn*), sta riscontrando grande successo nelle scuole primarie tra gli insegnanti e, soprattutto, tra gli alunni.

1.1 Pensiero Computazionale

Concetto che sempre più negli ultimi anni si è visto diffondersi è quello di *Pensiero Computazionale*. Ciononostante, la stragrande maggioranza delle persone, anche del settore, non conosce o da per scontato di sapere, seppur marginalmente, di che cosa realmente si tratta e quali implicazioni potrebbe avere nel futuro dell'istruzione e delle prossime generazioni in tutto il mon-

do. Dunque, quando è nato, chi lo ha proposto per primo, cosa significa e, soprattutto, perché è così importante per la scuola del futuro?

1.1.1 Nascita e definizione del Pensiero Computazionale

Il concetto di *Pensiero Computazionale* è stato introdotto per la prima volta dal matematico, filosofo, pedagogista e informatico statunitense, nonché padre del *Costruzionismo*, Seymour Papert nel 1980 nel suo libro *Mindstorms* [1]. Successivamente, nel 1996 Papert riprese tale concetto nel contesto di LOGO, il linguaggio di programmazione da lui sviluppato al MIT (*Massachusetts Institute of Technology*) con l'obiettivo non solo di insegnare la programmazione ai bambini ma, soprattutto, come strumento per agevolarne e migliorarne l'apprendimento.

Per quanto riguarda, invece, il significato di tale termine, oggi ancora non esiste una definizione universalmente condivisa. Quella che però sembra mettere d'accordo la maggior parte degli esperti è la definizione formulata dalla dottoressa Jeannette Wing, direttrice del Dipartimento di Informatica della *Carnegie Mellon University*, in un articolo del 2006 dal titolo *Computional Thinking* [2]. Da tale articolo, infatti, si può evincere quello che per la dottoressa Wing è il significato di tale concetto attraverso i seguenti punti fondamentali.

Il Pensiero Computazionale:

- È riformulare un problema apparentemente difficile in uno che sappiamo come risolvere, magari semplicemente riducendolo, incorporandolo, trasformandolo o simulandolo.

- È pensare utilizzando i meccanismi di astrazione e decomposizione quando si ha ad avere a che fare con un compito o un sistema complesso e di grandi dimensioni. È separazione dei concetti. È saper scegliere il metodo di rappresentazione più appropriato e modellare gli aspetti più importanti di un problema, in modo da renderlo più facilmente trattabile.

- Rappresenta un atteggiamento e una competenza universalmente applicabili da chiunque, non solo dagli informatici.

- Va dalla risoluzione di problemi e la progettazione di sistemi, a capire certi comportamenti umani, affidandosi ai concetti fondamentali e gli strumenti mentali propri dell'informatica.

Da quanto definito si può quindi asserire come il *Pensiero Computazionale* altro non è che *un'attitudine e un processo mentale applicabili da chiunque e in qualunque contesto, e che attraverso metodi e strumenti specifici, propri del ragionamento informatico-algoritmico, consente di risolvere problemi di varia natura e complessità.*

Sempre nello stesso articolo, la dottoressa Wing ci tiene, inoltre, a sottolineare ciò che caratterizza il *Pensiero Computazionale*, definendo, nello specifico, *cosa sia*, e *cosa*, invece, *non sia*.

Il Pensiero Computazionale è:

- *Concettualizzazione, non programmazione*: essere informatico non significa essere programmatore. Pensare come un informatico, quindi, non significa tanto saper programmare quanto saper pensare a più livelli di astrazione.

- *Un fondamento, non un'abilità meccanica*: un'abilità fondamentale è qualcosa che ogni essere umano dovrebbe saper utilizzare nella società odierna, in quanto arricchente la propria esperienza di vita, e che non può essere vista, quindi, come un'abilità meccanica e ripetitiva.

- *Un modo in cui gli umani, e non i computer, dovrebbero pensare*: Il *Pensiero computazionale* dovrebbe essere un modo attraverso cui l'uomo risolve i problemi di ogni giorno. Ciò non significa che gli umani debbano iniziare a pensare come i computer. I calcolatori, beninteso, sono estremamente stupidi e noiosi, e solo attraverso la genialità, la creatività e l'immaginazione dell'uomo diventano strumenti utili e interessanti.

- *Un insieme di idee, non di artefatti*: Non saranno tanto gli artefatti che riusciremo a realizzare attraverso il *Pensiero Computazionale* a cambiare drasticamente le nostre vite, ma come, attraverso questo nuovo metodo di pensiero, affronteremo e risolveremo i problemi e le difficoltà di ogni giorno, come organizzeremo le nostre giornate e come comunicheremo e interagiremo con le altre persone.

Alla luce di quanto detto, risulta allora facile notare quanto ogni aspetto della nostra vita, attraverso questo nuovo metodo di pensiero, possa trarne un beneficio significativo e palpabile. Ciononostante, quanto è davvero importante sensibilizzare l'opinione pubblica e mobilitare le istituzioni affinché tale approccio sia introdotto, quanto prima possibile, nello sviluppo e nell'istruzione dei bambini?

1.1.2 L'importanza del Pensiero Computazionale

Il termine "computazionale" può certamente indurre a pensare che il *Pensiero Computazionale* sia un'abilità utile solo a chi ha fatto dell'informatica la propria professione. Tuttavia, come già accennato, si tratta invece di una *competenza fondamentale che tutti dovrebbero possedere.*

In particolare, al di là dei già accennati benefici generali riconducibili all'utilizzo del pensiero computazionale nella vita quotidiana, vi sono due importanti ragioni - una di carattere pedagogico, l'altra legata al mondo del lavoro e delle prospettive professionali future - per le quali è importante infondere tale metodo di ragionamento fin dall'infanzia.

A livello pedagogico, attraverso il pensiero computazionale è possibile aprire nuove prospettive relativamente a come i bambini vedono se stessi e il mondo che li circonda, risultando essere uno strumento di crescita ed espressione personale formidabile. Per i bambini sarebbe quindi più facile e naturale esprimere se stessi, comunicare e interagire con gli altri, porsi domande e imparare a gestire problemi via via sempre più complessi [3, pp. 28, 29].

A livello lavorativo, invece, vi è l'ormai universale consapevolezza che per riuscire ad affrontare nel modo ottimale il proprio futuro professionale, ottenendo risultati significativi, i giovani debbano, quanto prima possibile, *imparare a imparare.* Abilità acquisibile grazie alla pratica del pensiero computazionale. In una società caratterizzata da una così rapida evoluzione tecnologica e culturale, infatti, non è più concepibile limitarsi a fornire risposte preconfezionate ai quesiti che ci si palesano davanti ogni giorno. Le informazioni e la conoscenze richieste sono sempre più in costante e rapido mutamento, al punto tale che risulterà probabilmente impensabile, negli anni avvenire (ma già oggi nei settori tecnici), poter affrontare il proprio lavoro solo con quelle scolastiche e accademiche: ciò che gli studenti universitari imparano oggi tra cinque anni sarà probabilmente obsoleto, in maniera particolare per le discipline tecniche. Inoltre, è davvero difficile pensare che gli studenti di oggi che domani troveranno un lavoro, riescano a mantenerlo fino alla fine della propria carriera lavorativa, come accadeva in passato ai loro padri o nonni. È molto più probabile, invece, che nel prossimo futuro i giovani siano chiamati a cambiare lavoro piuttosto frequentemente, dovendo imparare a destreggiarsi anche nell'ambito di discipline differenti da quella di provenienza e specializzazione. In vista di quello che sarà un futuro caratterizzato, quindi, da un elevato *staff turnover*[1], si ha la necessità di

[1] **staff turnover:** in ambito lavorativo, si intende la sostituzione (ricambio) del personale.

porre le generazioni future, quanto prima possibile, in un atteggiamento di *lifelong learning*, facendo loro acquisire quelle abilità che gli consentiranno di sviluppare un'attitudine mentale utile ad affrontare problemi di ogni genere e ordine. E quale palestra migliore se non l'esercizio quotidiano del pensiero computazionale per raggiungere tale obiettivo? [4]

1.1.3 Il Pensiero Computazionale nella scuola del futuro

Sono in molti oggi a credere che il pensiero computazionale costituisca la quarta abilità di base oltre al saper leggere, scrivere e fare di calcolo. Ed è per questo motivo che va facendosi strada sempre più la convinzione che il pensiero computazionale debba essere insegnato a ogni bambino. Perché così come leggere, scrivere e contare sono abilità che è importante imparare fin da bambini, per la scuola del futuro sarà necessario rendere anche il *pensiero computazionale un'abilità da apprendere ed esercitare fin dai primi anni* [4, *La programmazione come strumento per sviluppare il pensiero computazionale*].

Fortunatamente, in tale direzione si sono mosse negli ultimi anni diverse iniziative internazionali e nazionali. Un esempio fra tutti è *Code.org*[2], un progetto statunitense no-profit, nato per diffondere il pensiero computazionale e il coding in tutto il mondo, attraverso corsi dedicati a insegnanti e studenti e con un approccio orientato al *problem solving*.

In Italia, le principali iniziative volte a far crescere le competenze digitali e a diffondere il pensiero computazionale all'interno delle scuole, invece, sono: la legge della *Buona Scuola*, entrata in vigore il 16 luglio 2015, e la versione "nostrana" di Code.org: *Programma il Futuro*[3]. Quest'ultima in particolare, avviata dal *Miur* in collaborazione con il *Consorzio Interuniversitario Nazionale per l'Informatica*, nasce proprio per incoraggiare la diffusione del coding e del pensiero computazionale nella scuola primaria e secondaria, con l'obiettivo di fornire sostegno a tutte quelle scuole che vogliano cimentarsi con questa nuova dimensione dell'insegnamento e dell'apprendimento. Attraverso il sito del progetto, *Programma il Futuro* sostiene l'esperienza di Code.org, quale punto di riferimento in tutto il mondo per quando riguarda il coding a livello di istruzione, traducendo i contenuti di quest'ultimo e rendendo disponibile, al supporto degli insegnati, percorsi strutturati, videolezioni, forum di discussione e, per chiunque in generale, corsi caratterizzati da un taglio prettamente pratico, semplice e intuitivo, come: esercizi di vario genere e giochi di difficoltà crescente, attraverso i quali imparare a scrivere

[2] `https://code.org`
[3] `https://programmailfuturo.it`

(per mezzo della programmazione visuale basata sulla composizione di opportuni blocchi funzionali) sequenze di istruzioni sempre più complesse, con l'obiettivo di far compiere ai personaggi del gioco i compiti richiesti [5, pp. 15-18].

In generale, la direzione che sempre più istituzioni ed enti, sia a livello nazionale sia internazionale, stanno cercando di delineare è quindi quella nella quale, all'interno delle scuole, il ruolo rivestito dai dispositivi, dal calcolatore e dal coding sarà sempre più fondamentale nel contesto dell'insegnamento. In particolare, attraverso tali strumenti sarà possibile assistere al passaggio da un'istruzione basata sulla *trasmissione della conoscenza*, dal docente all'allievo, a una basata sulla *creazione della conoscenza* attraverso l'esperienza e l'errore. Il computer, strumento catalizzatore del *Pensiero computazionale*, avrà lo scopo di migliorare e arricchire l'apprendimento in tutte le materie scolastiche. Proprio Seymour Papert, nella sua teoria *Costruzionista*, definisce il computer come un nuovo mezzo di apprendimento, tramite il quale gli studenti possono essere creatori del proprio processo di crescita, smettendo di essere soggetti passivi (che subiscono la tecnologia), divenendo soggetti attivi e creatori di contenuti. Il computer quindi, da semplice macchina per la gestione delle informazioni, divenendo uno strumento per apprendere, scoprire, mettersi in gioco e imparare dai propri errori, potrà essere la base della *Scuola del futuro*.

1.2 Seymour Papert e Mitchel Resnick

Nel paragrafo 1.1 è stato largamente trattato il concetto di *Pensiero Computazionale* e dell'importanza di questo per le generazioni future. Si è inoltre citato più volte quello che è stato il maggiore fautore di tale metodo di pensiero, nonché padre del *Costruzionismo*, Seymour Papert e i cui ideali sono oggi portati avanti da Mitchel Resnik e dal suo *Lifelong Kindergarden*.

1.2.1 Seymour Papert e il Costruzionismo

Seymour Papert[4], matematico, filosofo, pedagogista e informatico statunitense, è conosciuto principalmente come il "padre" del *Costruzionismo* e del LOGO, linguaggio di programmazione sviluppato come supporto all'apprendimento, nonché per essere stato il maggiore esperto al mondo circa l'utilizzo della tecnologia nell'insegnamento e nell'apprendimento dei bambini.

[4] 29 febbraio 1928 - 31 luglio 2016.

Nato e cresciuto in Sudafrica, Seymour Papert si trasferisce nel 1954 a Cambridge dove svolge, fino al 1958, ricerche nel campo della matematica. Successivamente, durante un periodo di studio a Ginevra, collabora col noto psicologo e pedagogista Jean Piaget circa le teorie sull'apprendimento. Questa esperienza sarà per Papert estremamente determinante, perché lo condurrà, non solo, a pensare alla matematica come ad una disciplina che può permettere ai bambini di diventare consapevoli dei processi con cui imparano e pensano, ma soprattutto perché gli consentirà di sviluppare una propria teoria sull'apprendimento: il *Costruzionismo*. Agli inizi degli anni '60 Papert entra al MIT (*Massachusetts Institute of Tecnology*) dove fonda con Marvin Minsky il Laboratorio di Intelligenza Artificiale. Nel 1963 realizza, con i suoi collaboratori, il linguaggio di programmazione LOGO, con l'obiettivo, non solo, di insegnare ai bambini a programmare, ma soprattutto come strumento per agevolarne e migliorarne l'apprendimento. Negli anni, grazie alla diffusione e il successo che tale linguaggio ha avuto, il LOGO è diventato sempre più il punto di riferimento per chiunque volesse usare il computer con i bambini, evidenziando l'importanza del coding nell'istruzione.

Secondo la teoria *Costruzionista* definita da Papert, la Scuola dovrebbe smettere di essere il luogo in cui *trasmettere* la conoscenza, dal docente all'allievo, e diventare quello in cui *costruirla*, attraverso la realizzazione dei cosiddetti artefatti cognitivi[5]. In tale contesto, un ruolo fondamentale sarebbe rivestito, in particolare, dal calcolatore e dal coding, attraverso i quali rendere il bambino, da soggetto passivo della tecnologia, un soggetto attivo e creatore di conoscenza. Attraverso tali strumenti, il bambino verrebbe infatti messo in una condizione nella quale potersi mettere in gioco e scoprire da solo le conoscenze di cui avere bisogno, restituendogli la possibilità di apprendere in maniera naturale e spontanea, nonché consentendogli di acquisire una maggiore fiducia e consapevolezza delle proprie possibilità, esprimendo se stesso attraverso la realizzazione degli artefatti cognitivi.

Proprio come strumento per raggiungere gli obiettivi posti dalla teoria costruzionista, Papert realizzò il linguaggio LOGO. Attraverso questo, Papert era sicuro di poter insegnare ai bambini, attraverso il computer e il coding, ad *imparare a imparare*, lasciando loro lo spazio e il tempo per la costruzione degli artefatti cognitivi, e da questi riflettere sui processi, o gli errori, che sarebbero potuti scaturirne. Proprio all'interno di tale contesto, il concetto di *errore* trova, secondo la teoria di Papert, un significato completamente nuovo. Esso non deve essere più visto come una debolezza, una mancanza,

[5]**artefatto cognitivo:** artefatto software o hardware nella cui realizzazione si ha un miglioramento del processo di apprendimento.

un qualcosa di negativo ma, piuttosto, come un occasione per comprendere meglio ciò che si è fatto, o non si è fatto, divenendo quindi un'opportunità di crescita e miglioramento personali [5, 1.2].

La cosa più importante da ricordare di Papert, quindi, è senza ombra di dubbio l'impatto che il suo pensiero e la sua attività di ricerca hanno avuto per le teorie dell'apprendimento, caratterizzate, alle fondamenta, dall'uso delle nuove tecnologie come strumento di apprendimento attivo [6].

1.2.2 Mitchel Resnick e il Lifelong Kindergarden

Il testimone di Papert è stato raccolto dal suo ex allievo e responsabile del *Lifelong Kindergarten* del MIT MediaLab, Mitchel Resnick[6], al quale si deve la realizzazione del famosissimo linguaggio Scratch[7].

Laureato in Fisica nel 1978 presso l'università di Princeton e PhD in Computer Science presso il MIT, è cofondatore del progetto *Computer Clubhouse*, programma gratuito di apprendimento post-scuola riservato ai giovani delle comunità a basso reddito o con difficoltà. Attualmente si occupa di ricerca nel campo dell'apprendimento presso il MIT Media Lab , dove sviluppa nuove tecnologie e attività per impegnare persone di qualunque età, ma soprattutto i bambini, in esperienze creative di apprendimento.

Per Resnik, proprio come per il suo mentore Papert, l'attività di coding e, in generale, la realizzazione di opere cognitive per mezzo di questa è di fondamentale importanza all'interno dell'istruzione dei bambini e delle generazioni future. Le ragioni per le quali, secondo Resnick, il coding è tanto importante possono ritrovarsi in un articolo [7, p. 62], pubblicato sul *Communications of the ACM*, nel quale lo stesso Resnick parla di Scratch:

"Perché programmare? È diventato un luogo comune identificare i giovani come nativi digitali *a causa della loro apparente agilità nell'interagire con le tecnologie digitali. Ma il fatto che siano considerati dei* nativi digitali *li rende davvero fluenti con le nuove tecnologie? Anche se interagiscono con i media digitali tutto il tempo, pochi sono davvero in grado di creare i propri giochi, le proprie animazioni o simulazioni. In effetti è quasi come se potessero "leggere" senza però saper "scrivere". Ciò che manca è quindi la capacità di sfruttare tali tecnologie per progettare, creare e inventare cose nuove, per i quali è necessario il codig. Saper programmare porta in genere diversi benefici. Per esempio, espande notevolmente la portata di ciò che puoi creare (e come puoi esprimere te stesso) con il computer. Inoltre consente di estendere notevolmente la portata di ciò che puoi imparare. In particolare, il coding*

[6] https://www.media.mit.edu/people/mres/overview
[7] http://scratch.mit.edu

supporta il Pensiero Computazionale*, grazie al quale poter imparare importanti strategie di problem-solving e progettazione (come la modularizzazione e il design iterativo) e i cui benefici vanno ben oltre ai domini dell'informatica. Infine, poiché la programmazione coinvolge la creazione di rappresentazioni esterne dei tuoi processi di problem-solving, fornisce l'opportunità di riflettere sul proprio pensiero, nonché a riflettere sull'opinione che abbiamo di noi stessi".*

Proprio per perseguire gli obiettivi di diffusione del pensiero computazionale, del coding a scuola e, in generale, per lo studio del cosiddetto *Creative Learning*, Resnick ha fondato il team *Lifelong Kindergardern.* Fondando le proprie radici sui principi promulgati dal maestro Papert col *Costruzionismo*, vanta tra le opere più significative, oltre al linguaggio Scratch, la realizzazione di un framework per l'insegnamento del pensiero computazionale e la valutazione dell'apprendimento conseguito. Tale framework, fondandosi sulla convinzione che i bambini possano acquisire il pensiero computazionale programmando storie interattive e videogiochi (gli artefatti di cui parlava Papert), individua in questo tutta una serie di concetti, pratiche e attitudini che lo caratterizzano [4, 8].

Concetti del pensiero computazionale

Sequenza: attività che può essere espressa attraverso una serie consecutiva di singoli step o istruzioni.

Ciclo: meccanismo per eseguire più volte la medesima sequenza in maniera iterativa.

Evento: verificarsi di un'azione che causa lo scatenarsi di un'altra.

Parallelismo: esecuzione di sequenze di istruzioni differenti allo stesso tempo.

Condizione: possibilità di prendere decisioni sulla base del verificarsi di determinate situazioni.

Operatore: costrutto per la manipolazione di numeri e stringhe di caratteri.

Dati: valori che possono essere salvati, recuperati e modificati durante l'esecuzione di un programma.

Pratiche del pensiero computazionale

Essere incrementali e iterativi: approcciare la risoluzione di un problema per passi successivi, in modo da venire in contro a possibili evoluzioni future dello stesso.

Testare e debuggare: individuare problemi ed errori e correggerli.

Riusare (pattern recognition): riconoscere come alcune parti di soluzione possono essere riusate nella stessa o riapplicate a problemi simili.

Remixare (copiare per migliorare): prendere spunto dalle idee e dal codice di altre persone per costruire cose più complesse di quelle che si sarebbero potute realizzare per conto proprio, dando una spinta alla propria creatività.

Astrarre: ridurre la complessità di un problema, con lo scopo di far emergere l'idea principale, mantenendo solo alcuni aspetti e tralasciandone altri.

Modularizzare: scomporre un problema complesso in problemi più semplici, dalla cui risoluzione poi risolvere anche il problema complesso.

Attitudini del pensiero computazionale

Esprimere se stessi: utilizzare la tecnologia come mezzo per esprimere se stessi, la propria creatività, le proprie idee e mettersi in gioco.

Essere connessi: saper comunicare e lavorare con gli altri per raggiungere un obiettivo o una soluzione condivisa.

Porre domande: avere una mente vigile, attraverso la quale interrogarsi in continuazione su come un qualsiasi oggetto nel mondo reale possa funzionare.

1.3 Dal LOGO ai linguaggi Scratch e Snap!

Papert, col suo *Costruzionismo*, sosteneva che alla base di un migliore e più naturale apprendimento la mente umana avesse bisogno di creare artefatti cognitivi, ovvero rappresentazioni reali del mondo con cui interagisce. Solo attraverso la realizzazione di tali artefatti l'uomo sarebbe stato in grado di apprendere ciò che realmente era importante e in maniera più duratura, focalizzandosi su quegli aspetti emozionali che sarebbero scaturiti nella propria mente, frutto dell'interazione fra l'esperienza che avrebbe vissuto in quel momento e le esperienze pregresse. Papert identifica, in particolare, nel

computer e nel coding gli strumenti fondamentali per la realizzazione di tali artefatti.

1.3.1 Il linguaggio LOGO

Creato più di cinquant'anni fa, LOGO è a tutti gli effetti il *linguaggio del costruzionismo* in quanto nato proprio per sperimentare sul campo le teorie di Papert, il suo ideatore. La prima volta che Papert parla del linguaggio LOGO è nel suo libro *Mindstorms* [1] (nome oggi associato a una famosa linea di giocattoli LEGO e scelto proprio in onore e memoria del contributo che Seymour diede al produttore di giocattoli danese - 2.3.2), nel quale, oltre a descrivere i benefici dell'alfabetizzazione informatica nell'istruzione primaria e secondaria, ne descrive le caratteristiche e le potenzialità.

Noto per essere forse il linguaggio più famoso nell'ambito dell'educazione scolastica, LOGO è caratterizzato dall'essere modulare, estendibile, interattivo e flessibile, consentendo all'utente di utilizzare sia i comandi testuali preesistenti, sia quelli creati e personalizzati dall'utente stesso. Papert ha sempre pensato che il LOGO sarebbe stato un linguaggio tanto semplice da poter essere usato dai bambini, quanto capace di costrutti complessi adatti alle esigenze dei programmatori più "adulti" [6].

La programmazione con il LOGO non è mai fine a se stessa ma sempre pensata in relazione ad un dato progetto relativo alle discipline più svariate: dalla matematica, la lingua, la musica, fino alla realizzazione di un videogioco o di un robot. Proprio da questa caratteristica del LOGO nasce il concetto di *Micromondo*, ossia un ambiente di apprendimento appositamente realizzato per esplorare e apprendere una data disciplina e materia, e che rivede nella *Geometria della Tartaruga* quello indubbiamente più conosciuto e usato, all'epoca, nelle scuole di tutto il mondo. Nata originariamente come robot movente per mezzo di comandi impartiti attraverso un calcolatore (fig. 1.1), la Tartaruga, trasferita poi sul monitor del computer, divenne in seguito uno strumento estremamente potente e versatile, adatto all'insegnamento ai bambini della matematica e della geometria in maniera creativa. Questi, infatti, impersonandosi direttamente nella tartaruga, erano in grado di realizzare disegni e immagini digitali partendo semplicemente dalla direzione da seguire e dal numero di passi da percorre per formare i vari lati delle figure da disegnare, sperimentando, quindi, nuovi approcci alla risoluzione dei problemi matematici [9, 10].

L'approccio alla programmazione in LOGO, in particolare, consiste nel provare, testare, ricostruire quello che non va e riprovarlo, consentendo di imparare a sistemare un ragionamento piuttosto che scartarlo tutto in bloc-

Figura 1.1: La Tartaruga originariamente creata da Seymour Papert.

co come sbagliato. Il LOGO è stato infatti pensato per permette all'utente di realizzare quanto desiderato per tentativi piuttosto che al primo colpo, spingendo questo a sperimentare cose nuove, piuttosto che mettere assieme quelle conosciute. In tale contesto, in particolare, l'elemento dell'*errore* non deve essere più visto come qualcosa di "spaventoso" e di bloccante per l'apprendimento ma, anzi, come la scintilla per quel meccanismo di ricerca e individuazione dell'errore, chiamato *debugging*, attraverso il quale poter comprendere e risolvere quanto c'è di sbagliato. In definitiva, la realizzazione di soluzioni in LOGO è quindi caratterizzata da un approccio iterativo che partendo da soluzioni base e parziali, spinge l'utente a raffinare queste per passi successivi fino ad arrivare a quella che è la soluzione desiderata [5, p. 6].

1.3.2 Scratch

Mitchel Resnik e il suo *Lifelong Kindergardern*, a partire dalle necessità che andavano maturando nei giovani del *Computer Clubhouse*, e riprendendo alcune delle caratteristiche che hanno fatto del LOGO il più influente linguaggio di programmazione in ambito educativo (modularità, semplicità, flessibilità, organizzazione in *Micromondi*), lanciarono nel 2007 la prima versione dell'ormai noto a tutti linguaggio Scratch. Ambiente di programmazione visuale e orientato agli oggetti, Scratch avrebbe dovuto consentire ai ragazzi di creare in maniera semplice e intuitiva le proprie storie animate e i propri giochi.

Come enuncia Resnick in [7, p. 60], quando col suo team iniziarono a muovere i primi passi nella vita del progetto Scratch l'obiettivo era fin da

subito ben chiaro e definito:

"Volevamo sviluppare un approccio alla programmazione che avrebbe attratto qualunque genere di persone, anche quelle che non si sarebbero mai immaginate nei panni di un programmatore. Volevamo rendere semplice per chiunque, a prescindere dall'età, dal background culturale e dagli interessi, la programmazione delle proprie storie interattive, i propri videogiochi, le proprie animazioni e simulazioni, nonché condividere tali opere con chiunque volessero".

Sin dal momento del lancio pubblico nel maggio 2007, il sito web di Scratch [8] è diventato subito il centro nevralgico di una vibrante comunità online, ricco di persone sempre pronte a condividere, discutere e collaborare ai progetti di chiunque altro all'interno della community, seguendo quello che è divenuto ormai il loro slogan principale "Immagina, Programma, Condividi". Proprio quest'ultimo rimarca quanto sia importante e fondamentale il ruolo della condivisione, nonché quello della creatività, nella filosofia alla base del progetto Scratch.

Oggi Scratch conta una community di giovani sviluppatori (Scratchers[9]) estesa in tutto il mondo e che ogni giorno carica sul sito migliaia di nuovi progetti, rendendo disponibile gratuitamente il sorgente di questi a chiunque ne sia interessato, e facendo di Scratch il cosiddetto "YouTube dei media interattivi". È proprio attraverso la realizzazione e la condivisione di questi progetti che gli Scratchers possono imparare importanti concetti alla base della matematica, dell'informatica e di molte altre discipline, il tutto facendo uso di un pensiero creativo, sistemico e collaborando reciprocamente (tutte abilità essenziali nel 21esimo secolo). In effetti, come dichiara Resnick [7], l'obiettivo del progetto Scratch, in vero, non è mai stato quello di rendere le persone degli informatici o dei programmatori di professione ma di coltivare una nuova generazione di pensatori creativi e sistemici, in grado di esprimere le proprie idee attraverso lo strumento del calcolatore e della programmazione. Obiettivo che si spera venga presto raggiunto, dal momento che oggi Scratch risulta essere lo strumento più efficace per le attività di coding e pensiero computazionale all'interno delle scuole di tutti i gradi.

Caratteristiche

Scratch è un linguaggio, nonché un ambiente, di programmazione visuale completo[10], con supporto alla programmazione concorrente e orientato agli

[8] `http://scratch.mit.edu`

[9] **Scratcher:** appellativo dato agli sviluppatori in ambiente Scratch.

[10] **completo:** si intende la capacità di poter scrivere qualsiasi programma.

oggetti e agli eventi. Si contraddistingue per la semplicità e l'immediatezza con le quali è possibile interagire con le funzionalità messe a disposizione dallo stesso. La programmazione per mezzo di Scratch è caratterizzata, infatti, dalla realizzazione del codice del programma attraverso l'utilizzo di una serie di blocchi componibili (fig. 1.2), ai quali corrisponde una data funzione e colorati a seconda della categoria di appartenenza (tab. 1.1). In base a come tali blocchi sono connessi tra loro è possibile quindi realizzare un comportamento completamente diverso. Fu proprio il lavoro svolto e l'esperienza maturata dal team *Lifelong Kindergardern* in collaborazione con Lego (2.3.2) a convincerli che una metodologia di programmazione basata sulla costruzione del codice attraverso la composizione di blocchi opportuni (come i mattoncini dei Lego), avrebbe garantito la massima immediatezza, facendo capire intuitivamente all'utente quali blocchi poter utilizzare in un dato contesto (i.e. la sintassi del linguaggio) [7, p. 63].

Figura 1.2: Classico "Hello, World!" realizzato in Scratch.

L'ambiente Scratch è suddiviso in 4 sezioni principali (fig. 1.3): lo Stage (in alto a sinistra), l'area degli Sprite[11] ed degli oggetti grafici (in basso a sinistra), il menu dei blocchi (centrale) e, infine, l'area di scripting (a destra).

Stage: rappresenta il "palcoscenico" dove gli Sprite e gli altri oggetti grafici possono prendere vita, nonché dove vedere i risultati del codice realizzato semplicemente premendo il relativo pulsante di "avvio dell'esecuzione" (quello con la bandierina verde), col quale mandare in esecuzione (concorrente) tutti gli script associati ai vari Sprite.

Area degli Sprite: contiene i vari elementi grafici (spesso attori) con i quali il codice può interagire. Selezionando uno Sprite è possibile associarvi uno o più script semplicemente trascinando i blocchi desiderati dal menu relativo all'area di scripting, componendo così lo script/comportamento voluto.

Menu dei blocchi: contiene i vari blocchi a disposizione, opportunamente suddivisi per categoria di appartenenza.

[11] **Sprite:** agente (o meglio "attore") il cui comportamento può essere programmato attraverso l'assegnamento di uno o più script.

Colore	Categoria	Descrizione
	Movimento	Per muovere gli Sprite e cambiare gli angoli.
	Aspetto	Per controllare la visibilità, i costumi e l'output.
	Suono	Per eseguire brani e sequenze audio programmabili.
	Penna	Per realizzare disegni e per la grafica.
	Variabili e liste	Per creare, rimuovere e manipolare variabili.
	Situazioni	Per la gestione degli eventi e dell'esecuzione degli script.
	Controllo	Per il controllo del flusso di esecuzione (strutture di selezione, ripetizione, etc).
	Sensori	Per percepire eventi da Sprite e utente.
	Operatori	Per effettuare operazioni matematiche, logiche e di confronto.
	Altri blocchi	Per il controllo delle periferiche e azioni personalizzate.

Tabella 1.1: Categorie di blocchi in Scratch.

Area di scripting: selezionando uno Sprite, se questo ha associato del codice, è possibile visualizzare e modificare gli script associati a questo nell'area a destra, nonché provare, in maniera isolata, un qualsiasi script o blocco con un semplice "doppio clic" su di esso, in modo da vederlo in azione nell'anteprima.

Scratch consente la realizzazione di una vasta gamma di progetti differenti (si va dai videogiochi alle newsletters interattive, dalle simulazioni scientifiche ai tour virtuali, da scenette di danza a tutorial interattivi) nonché un elevato grado di personalizzazione di questi, consentendo all'utente di caricare contenuti personali, come foto, audio, etc, all'interno del progetto e integrarli col resto del codice in maniera estremamente semplice e veloce. Infine, tutti i progetti realizzati possono essere comodamente condivisi sul sito di Scratch attraverso l'ambiente di sviluppo, in modo che chiunque possa vederlo, eseguirlo e collaborare per migliorarlo.

La versione attuale, la 2.0[12] (rilasciata nel Maggio del 2013), ha visto, tra le varie novità, l'introduzione della possibilità da parte dell'utente di definire i

[12] https://wiki.scratch.mit.edu/wiki/Scratch_2.0

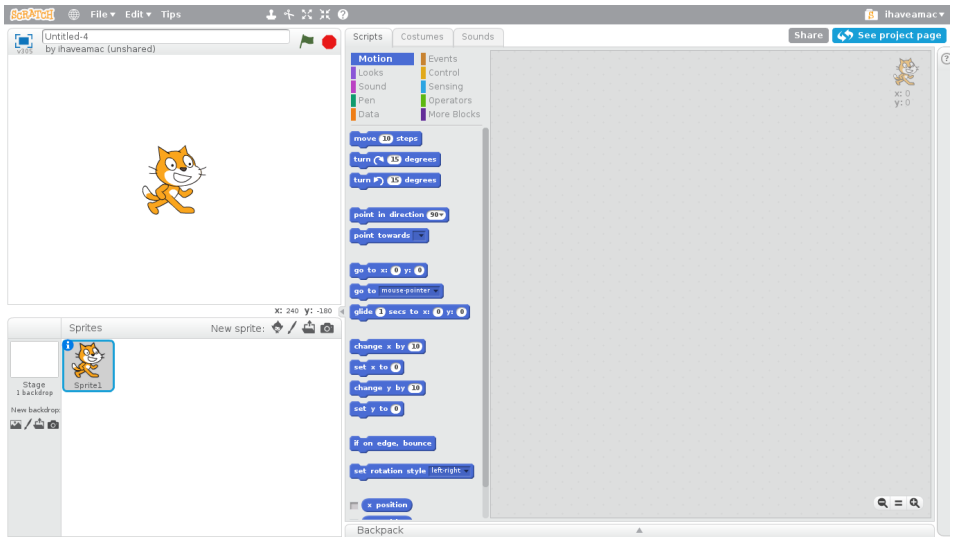


Figura 1.3: IDE di sviluppo di Scratch 2.0.

propri blocchi all'interno di un progetto. Con la prossima versione[13] (prevista per l'estate 2018), invece, sarà finalmente possibile utilizzare Scratch anche su dispositivi mobili (smartphone e tablet) grazie al porting del progetto da AdobeFlash ad HTML5.

1.3.3 Snap!

Snap! (formalmente BYOB - Build Your Own Blocks)[14] è un linguaggio di programmazione open source, visuale e caratterizzato da un'interfaccia drag-and-drop per la definizione del codice, che consente la programmazione direttamente da browser web (e quindi anche da dispositivi mobili) attraverso l'omonimo ambiente di sviluppo.

Creato da Jens Mönig e Brian Harvey dell'Università della California a Berkeley, la prima versione fu rilasciata nel 2011 quando venne usato per il corso introduttivo di Computer Science: *The Beauty and Joy of Computing*. Nel Dicembre 2014, 100 scuole superiori di New York City introdussero il corso *The Beauty and Joy of Computing* come nuovo corso di AP utilizzando proprio Snap! [5].

Snap! può essere visto essenzialmente come una re-implementazione estesa di Scratch al quale viene aggiunta la possibilità di costruire i propri blocchi, funzionalità introdotta in Scratch solo a partire dalla versione 2.0 nel 2013. Infatti, mentre da un lato importa da Scratch l'organizzazione dell'ambiente

[13] `https://wiki.scratch.mit.edu/wiki/Scratch_3.0`.
[14] `http://snap.berkeley.edu/index.html`.

di sviluppo, l'interfaccia drag-and-drop, gli strumenti per l'animazione, e le metafore visuali per la semantica dei blocchi, dall'altro aggiunge la possibilità di creare nuovi blocchi (da cui il nome BYOB - Build Your Own Blocks) di complessità arbitraria, definendone la categoria di appartenenza, il nome, il tipo e la lista dei parametri.

Oltre a questo, Snap! è caratterizzato da alcuni elementi non presenti originariamente in Scratch e presi in prestito dal linguaggio Scheme[15], come liste, procedure e continuazioni *first-class*. Quest'ultima caratteristica è essenziale per proiettare l'utilizzo di Snap! oltre la didattica e renderlo utilizzabile anche a livello accademico [11].

Liste *first-class* Uno dei limiti di Scratch, come linguaggio di programmazione per l'educazione informatica, è che le liste che mette a disposizione non sono *first-class*. Ciò significa che, ad esempio, non è possibile avere una lista di liste, ma non solo. Rendere le liste *first-class* rende invece possibile in Snap! la definizione di qualsiasi tipologia di struttura dati: *alberi, heaps, hash tables, dizionari*, etc.

Procedure *first-class* Un'altra limitazione di Scratch sta nell'assenza di procedure *first-class*. In Snap! la presenza di queste rende possibile all'utente la realizzazione di proprie di strutture di controllo (p.e. loop-cycle, non presente nativamente) nonché la possibilità di definire le *funzioni di ordine superiore*[16], potente meccanismo della programmazione funzionale.

Continuazioni *first-class* Un esempio di tipo di dati che esiste dietro le quinte di quasi tutti i linguaggi di programmazione è lo *stack frame*, la struttura dati che tiene traccia delle chiamate a procedura attive e delle loro variabili locali. Prendendo la scia di Scheme, in Snap! è stato reso possibile accedere a questi dati da parte del programmatore sotto forma di un tipo di dati *first-class* chiamato *Continuazione*. Usando le continuazioni l'utente è in grado di implementare meccanismi di controllo non locali come *catch-throw* e la gestione dei *threads*.

La commistione Scratch-Scheme dalla quale è originato Snap! rendono questo uno strumento davvero potente, utilizzabile perfettamente sia

[15]**Scheme:** linguaggio di programmazione funzionale dialetto del Lisp (List Processor), una famiglia di linguaggi di programmazione con implementazioni sia compilate sia interpretate, associata nel passato ai progetti di intelligenza artificiale.

[16]**funzione di ordine superiore**: funzione in grado di prendere come argomento e/o restituire come risultato, un'altra funzione.

in ambito didattico, dai bambini, sia in ambito hobbistico dai non esperti di informatica, grazie alla semplicità ereditata da Scratch, ma anche dagli informatici e dagli esperti del settore grazie ai costrutti avanzati messi a disposizione.

1.4 Progetto COGITO: *learn to code, code to learn*

Quando si impara a leggere e a scrivere si aprono molte opportunità di imparare cose nuove. Una volta imparato a leggere allora si potrà leggere per imparare. Imparare a programmare è la stessa cosa. Se si impara a programmare allora si potrà programmare per imparare: *learn to code, code to learn* [12]. Attraverso il *coding* si può certamente imparare come funziona un calcolatore, tuttavia questo sarebbe solo la punta dell'iceberg. Imparando a programmare si imparano molte altre cose: si imparano strategie di apprendimento per risolvere problemi sempre più complessi, realizzare progetti e comunicare idee. È importante evidenziare come queste non siano abilità utili solo ai programmatori ma a chiunque. Il coding è visto come estensione della capacità di scrivere e creare. Si insegna ai bambini a scrivere non perché diventino scrittori così come non gli si insegna a fare calcoli per diventare matematici, analogamente non gli si insegna il coding perché diventino informatici ma per permettere loro di acquisire un nuovo e più creativo modo di pensare, attraverso il quale poter realizzare nuovi tipi di cose [5, p. 11].

A partire da questa visione, nonché da quella costruzionista di Papert, nasce nel 2015, in un contesto di iniziative internazionali (p.e. *Code.org*[17]) e nazionali (p.e. il progetto *Programma il Futuro*[18] promosso dal MIUR), il progetto COGITO[19].

Nato come iniziativa a partire dal Prof. Alessandro Ricci e l' Ing. Angelo Croatti della facoltà di Ingegneria e Scienze Informatiche, in accordo col Dirigente Scolastico e alcuni insegnanti del 3o Circolo Didattico di Cesena (Scuola Primaria "Carducci"), COGITO[20] si pone come obiettivo quello di sviluppare nei bambini il pensiero computazionale nonché favorire l'esplorazione e l'apprendimento di concetti e competenze interdisciplinari attraverso l'utilizzo del calcolatore, del coding e di ambienti digitali protetti (i micromondi). Questo progetto nasce quindi dalla necessità di insegnare i concetti e le tecniche fondanti della programmazione, identificabili nel cosiddetto *Pensiero Computazionale*, in modo che fungano da supporto nello sviluppo delle

[17] `https://code.org`
[18] `https://www.programmailfuturo.it`
[19] `http://apice.unibo.it/xwiki/bin/view/Cogito/WebHome`
[20] **COGITO:** dal latino *pensare, riflettere.*

capacità di ragionamento logico, nella risoluzione di problemi, nell'ideazione e nella creazione di artefatti informatici digitali, nonché come sostegno all'apprendimento delle altre discipline scolastiche.

1.4.1 Strumenti e locali

Snap!

La piattaforma che si è deciso di utilizzare per lo svolgimento delle varie attività è Snap!. Tra le varie caratteristiche aggiuntive di questo rispetto a Scratch (1.3.3), quelle che più di tutte hanno fatto preferire il primo al secondo sono:

- la possibilità di poter utilizzare l'ambiente Snap! sui dispositivi mobili (quindi i tablet, a lezione), cosa non ancora pienamente possibile in Scratch[21];

- la possibilità di nascondere certi blocchi, rendendo disponibili nel progetto solo quelli necessari (sia base, sia definiti dall'utente), consentendo la creazione di ambienti virtuali didattici *ad hoc* (i *Micromondi*) caratterizzati da un proprio linguaggio per la programmazione (l'insieme di blocchi resi disponibili) e un proprio livello di astrazione, cosa non possibile in Scratch.

Li2Lab: Little Turtle Lab

Le attività si svolgono sia nelle aule tradizionali sia nell'aula TEAL[22] denominata Li2Lab (LittleTurtle Lab), appositamente inaugurata per le attività relative al coding e al pensiero computazionale. Quest'ultima, in particolare, è composta da banchi componibili, facili da spostare e adattare in base alle esigenze didattiche. All'interno del Li2Lab sono presenti circa una ventina di tablet, essenziali nello svolgimento delle varie attività da parte dei bambini, distribuiti ai vari gruppi di 2-3 ragazzi così da stimolare la cooperazione e il lavoro di gruppo.

[21] `https://wiki.scratch.mit.edu/wiki/Scratch_on_Tablets`, ultimo accesso 22 Gennaio 2018.

[22] **TEAL:** *Technology Enhanced Active Learning*, ossia *Apprendimento Attivo Abilitato Attraverso la Tecnologia*, è un metodo definito dal MIT che coniuga le lezioni frontali, le attività di laboratorio e l'attivismo pedagogico per dare vita a un'apprendimento arricchito e basato sulla collaborazione.

1.4.2 Attività

Primo anno

Le attività si sono svolte a cadenza periodica ogni due settimane con lezioni esclusivamente nel Li2Lab. Il primo anno è stato caratterizzato, nella prima parte, dall'introduzione ai bambini (due cassi seconde e due terze) del pensiero computazionale e, in generale, del coding, attraverso percorsi appositamente selezionati tra quelli proposti da Code.org e Programma il Futuro. Nella seconda parte ci si è, invece, concentrati nella realizzazione di "cartoline animate" in Snap!, a partire dalle quali poi arrivare a realizzare programmi di varia natura, richiamanti di volta in volta argomenti e tematiche che i bambini avevano già visto in altre materie.

Secondo anno

Mentre il primo è stato caratterizzato da una forte impronta "sperimentale", in quanto sia gli insegnati che i bambini stavano approcciando un modo del tutto nuovo di affrontare la didattica, durante il secondo anno si è cercato il più possibile di raggiungere due obiettivi ben precisi:

1. Consolidare ed estendere le competenze alla base del pensiero computazionale (algoritmo, programma, costrutti di selezione e ripetizione, variabili, approccio top-down alla decomposizione dei problemi, etc.) introdotte in quelle che erano le due cassi seconde e terze del primo anno;

2. Attuare fin da subito l'integrazione con le altre discipline scolastiche attraverso l'introduzione di ambienti didattici virtuali aventi come oggetto concetti e contenuti delle specifiche materie: i cosiddetti *Micromondi* (1.3.1, terzo capoverso, riga 5). Nell'ambito della programmazione in Snap!, questi sono composti da uno scenario (lo Stage), da uno o più attori (gli Sprite) e da un sottoinsieme di blocchi che definiscono il *linguaggio* di quel particolare micromondo. Sarà poi attraverso quest'ultimo che i bambini di volta in volta programmeranno lo specifico comportamento desiderato per i vari attori, attraverso la realizzazione dello/degli script necessario/i. Tra i vari micromondi utilizzati, i principali sono: *Tarta-Matematica* (per la risoluzione di semplici problemi), *Micromondo del Disegno* (per la realizzazione figure geometriche anche complesse), *Tarta-Parole* (per la manipolazione delle stringhe), *Micromondo della Griglia* (reticolo nel quale poter fare dei disegni specificando, secondo il sistema cartesiano, le coordinate

delle celle da colorare e, attraverso le componenti RGB, il colore da attribuire a queste) e *Videogioco Pioggia nel Pineto* vero proprio gioco che ha vista i bambini direttamente impegnati nella sua realizzazione.

Si è inoltre deciso di introdurre una lezione aggiuntiva rispetto a quella nel Li2Lab. A partire dal secondo anno, infatti, viste le possibili lacune e i dubbi irrisolti che durante il primo anno potevano formarsi stando fermi un'intera settimana, si è deciso di svolgere, in quella che era la settimana di "pausa", una lezione nelle aule tradizionali in modo da ripassare e rinforzare gli argomenti non chiari affrontati durante la precedente lezione nel Li2Lab.

Nel secondo anno, forti dell'esperienza maturata durante il primo, è stato inoltre possibile organizzare un piano d'esecuzione più coordinato con i vari insegnanti, in modo da affrontare nella maniera più parallela possibile i concetti visti nelle altre materie e le relative attività del progetto COGITO, in modo che i bambini potessero trarne il massimo beneficio.

Terzo anno: *il Micromondo della Matrice*

Il terzo anno (corrente) del progetto COGITO vede l'introduzione, tra gli altri, del *Micromondo della Matrice* (fig 1.4). Questo, riprendendo i concetti e gli elementi visti nel *Micromondo della Griglia* del secondo anno, è caratterizzato, anziché da un reticolo di celle, da una maglietta (lo *Stage*) con, sul petto, un reticolo di pallini neri, rappresentanti semplici led di una matrice 8x8.

Proprio come nel *Micromondo della Griglia* era possibile disegnare figure, specificando quali celle colorare (attraverso le relative coordinate) e il colore da assegnare a queste (secondo le componenti RGB), in questo micromondo è possibile fare analogamente programmando lo *Sprite* fornita di *default*, attraverso la definizione di un'apposito *script* che accenda i led desiderati. A tal proposito, per creare/ripulire la matrice e per accendere i led, sono disponibili due *blocchi custom* appositamente realizzati per questo micromondo: *prepara matrice*, per pulire (ridisegnare) la matrice, e *accendi led x: y: (colore r: g: b:)*, per accendere il led in coordinate *x, y* del colore *r, g, b*. Oltre a questi blocchi *custom*, sono messi a disposizione numerosi blocchi base (per il controllo di flusso, la manipolazione delle variabili, eseguire operazioni - matematiche, di confronto e logiche - e alcune funzioni di utilità), rappresentanti il linguaggio utilizzabile dai bambini per la definizione dei propri disegni e delle proprie animazioni (fig. 1.5).

A differenza di qualsiasi altro micromondo realizzato nel contesto del progetto COGITO, col *Micromondo della Matrice*, la cui attività principale consiste nel programmare una matrice di led posta su di una maglietta

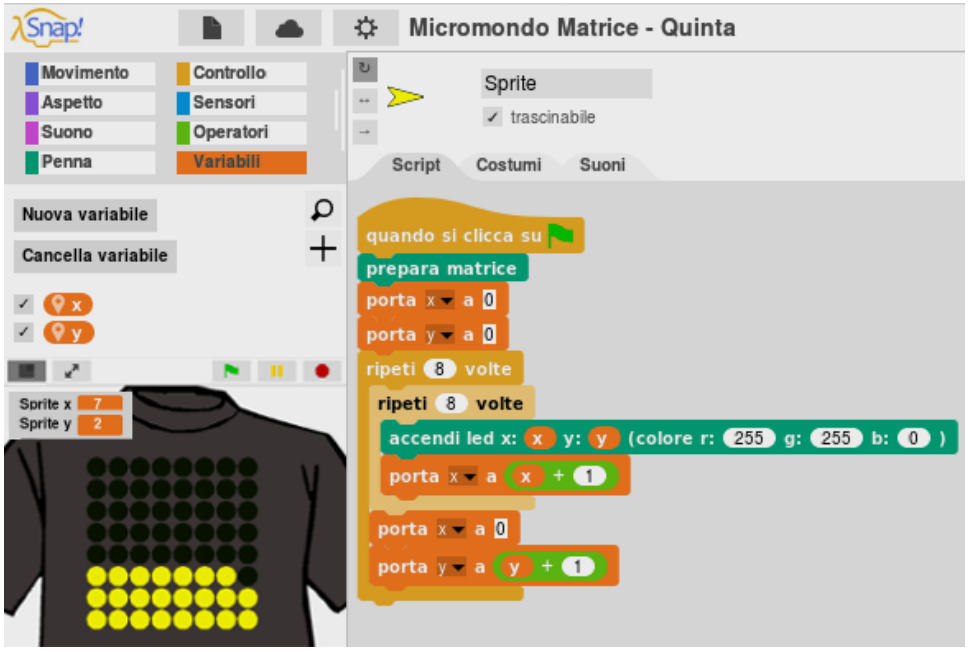


Figura 1.4: Ambiente Snap! con un semplice esercizio fatto nel contesto del *Micromondo della Matrice*, nelle classi quinte. A sinistra si evidenzia, in alto, la palette delle categorie di blocchi selezionabili e la vista sulle variabili definite, in basso, lo *Stage*, composto dalla maglietta con la matrice di led sul petto, sottostante al valore delle variabili (sulla sinistra) e ai comandi d'esecuzione (sulla destra). A destra, invece, è presentata la *script-view*, contenente lo script associato all'unico *Sprite* del micromondo.

(*smart*), si è voluto andare oltre a quanto i bambini avrebbero definito e visto sui tablet, consentendo loro la possibilità di programmare una *vera maglietta dotata di una matrice di led sul petto*: la *Smart T-Shirt*. L'obiettivo principale delle attività da svolgere in questo micromondo è, nello specifico, quello di riuscire, entro la fine dell'anno scolastico e attraverso un approccio incrementale (ossia realizzando disegni sempre più complessi ed elaborati), a realizzare animazioni e giochi di luce, fino ad arrivare a vere e proprie coreografie. A partire da queste, opportunamente trasferite su una vera matrice fisica (applicata sul petto della *Smart T-Shirt* - cap. 3), sarà poi possibile presentare, durante la festa di fine anno scolastico, un piccolo spettacolo nel quale i bambini, indossando ognuno una di queste magliette, daranno vita ad un gioco di luci e animazioni mai visto prima nelle scuole.

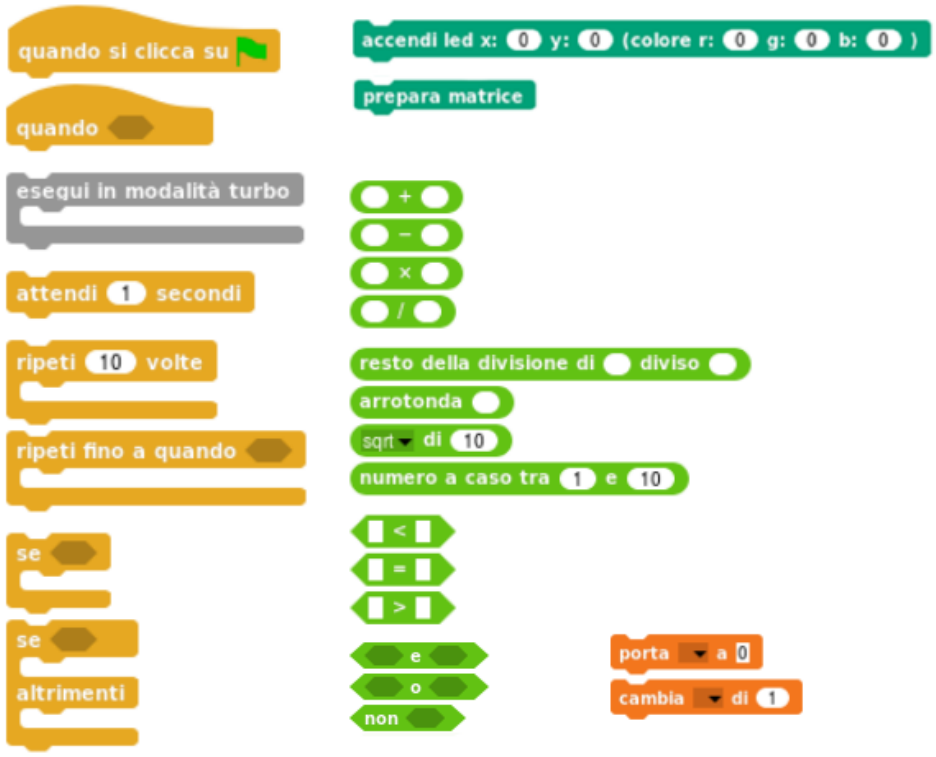


Figura 1.5: Blocchi costituenti il linguaggio utilizzabile nel *Micromondo della Matrice*.

1.5 Sommario

In questo primo capitolo è stato definito il concetto di *Pensiero Computazionale* nonché largamente discusso quanto possa essere importante, per le generazioni future, l'introduzione di questo all'interno delle scuole. Sono inoltre state introdotte le figure di Seymour Papert e Mitchel Resnick, importanti fautori del pensiero computazionale, e quanto per loro fosse fondamentale il ruolo del computer e del coding nell'istruzione dei bambini come catalizzatori del pensiero computazionale. A tal proposito, sono stati presentanti i due potenti linguaggi Scratch e Snap! che, partendo da quello che è stato il più famoso linguaggio di programmazione per la didattica (il LOGO), sono oggi il punto di riferimento per chiunque voglia approcciare un bambino al mondo della programmazione. Infine è stato descritto il progetto COGITO, iniziativa della facoltà di Ingegneria e Scienze Informatiche per diffondere il pensiero computazionale nelle scuole primarie Cesenati, rappresentante il contesto applicativo del progetto Smart T-Shirt (cap. 3).

Nel capitolo successivo, partendo dal concetto di *sistema embedded*, sarà invece trattato il contesto tecnologico del caso di studio.

Capitolo 2

Contesto tecnologico: Sistemi Embedded programmabili

Mentre nel primo capitolo è stato trattato il contesto applicativo del caso di studio, in questo viene trattato quello che è il contesto tecnologico. Quest'ultimo, nello specifico, è da inquadrare, da un lato, nelle tecnologie e nei dispositivi disponibili in grado di venire in contro alle esigenze e ai requisiti del caso di studio, dall'altro, nelle piattaforme esistenti utilizzate nel medesimo contesto applicativo, ovvero l'*Educational*, tra le quali poter collocare il progetto Smart T-Shirt.

Nello specifico, viene innanzitutto trattata la tecnologia dei *Sistemi Embedded*, alla base della fase di prototipazione del caso di studio, partendo dalla definizione del termine, le origini, le caratteristiche, le tipologie esistenti e alcuni dei principali ambiti di utilizzo, tra qui *l'Internet of Things* (2.1). Viene in seguito posto particolare focus sul progetto *Arduino*, basato su sistemi a microcontrollore, sulla storia e le caratteristiche di questo, nonché sulla scheda più famosa offerta da tale piattaforma (utilizzata per la prototipazione del caso di studio): *Arduino Uno* (2.2).

Infine, vengono trattate alcune delle più famose piattaforme e tecnologie embedded programmabili oggi utilizzate in ambito Educational: *Raspberry Pi*, *LEGO Mindstorm* e *WeDo* (2.3).

2.1 Sistemi Embedded

Partendo dalla definizione, un *sistema embedded* altro non è che un sistema di elaborazione *special-purpose*, ossia specializzato nello svolgimento di un dato compito e in un dato contesto, facente parte (i.e. integrato, *embedded*) in un sistema o un dispositivo più grande. Costituito da una componen-

te hardware e, generalmente, da una software (*embedded software*), il ruolo principale, come parte del sistema in cui è integrato e per conto di questo, è in genere quello di interagire con il mondo fisico mediante opportuni sensori e attuatori, notificando gli eventi e gli stimoli provenienti dall'ambiente ed eventualmente reagendo a questi sotto indicazione del sistema [13, 14].

2.1.1 Storia

I primi progetti e utilizzi dei sistemi embedded sono da ricollegarsi ai settori aerospaziale e militare degli anni '60. In ambito aerospaziale, in quegli anni veniva utilizzato, come computer di bordo nel *Programma Apollo* della *NASA*, l'*Apollo Guidance Computer* (AGC), incaricato del controllo delle manovre di allunaggio e decollo nel modulo lunare e in quello di comando. Nel settore militare, invece, trovava impiego il primo sistema embedded prodotto in massa, l'*Autonetics D-17B*, incaricato della gestione del sistema di guida del missile balistico intercontinentale *Minuteman I*[1], e in grado di consentire il miglioramento della precisione del missile attraverso la modifica dell'algoritmo di guida.

In seguito, nel rispetto alla famosa *Legge di Moore*[2], i sistemi embedded subirono una tale riduzione di costi, così come un'enorme crescita della capacità di calcolo e delle funzionalità offerte, da trovare sempre maggiore impiego anche in settori differenti da quello militare e aerospaziale. Il primo microprocessore monolitico[3] progettato per essere messo in commercio fu, ad esempio, l'*Intel 4004* (15 Novembre 1971). Questo, nonostante la necessità di chip di memoria e altra logica di supporto esterni, riuscì a trovare particolare applicazione a bordo di calcolatrici e altri sistemi di piccole dimensioni.

Verso la fine degli anni settanta, i microprocessori ad 8 bit, nonostante necessitassero sempre di logiche di supporto esterno, erano divenuti ormai la norma, tanto che sempre più applicazioni, favorite anche dalla caduta libera dei prezzi in quel periodo, cominciarono a preferirli rispetto alle metodologie personalizzate di progetto dei circuiti logici.

Verso la metà degli anni ottanta, un maggiore grado di integrazione permise poi il montaggio di altri componenti, in precedenza collegati esternamente (come memoria permanente e volatile, pin di I/O ed eventuali altri blocchi specializzati), sullo stesso chip del processore, dando vita ai *micro-*

[1] `http://minutemanmissile.com/missileguidancesystem.html`

[2] **Legge di Moore:** famosa legge in ambito elettronico e informatico formulata dall'imprenditore Gordon Moore, secondo la quale: "*La complessità di un microcircuito, misurata ad esempio tramite il numero di transistori per chip, raddoppia ogni 18 mesi circa*".

[3] **microprocessore monolitico:** sistema di elaborazione (*Central Processing Unit*) completo, costituito da un unito circuito integrato.

controllori. Caratterizzati da un costo per componente relativamente basso, l'utilizzo dei microcontrollori come alternativa alla realizzazione di interi circuiti logici dedicati divenne molto più interessante, favorendone la diffusione di massa. In breve tempo, ci fu un'esplosione nel numero di sistemi embedded distribuiti sul mercato, così come delle componenti fornite da vari produttori per facilitarne la progettazione. Vennero prodotti, ad esempio, molti circuiti integrati con un'interfaccia seriale (come il noto I^2C[4]), rispetto a quella parallela e più tradizionale, in quanto consentivano l'interfacciamento con i microcontrollori attraverso un minor numero di connettori.

Verso la fine degli anni ottanta, i sistemi embedded rappresentavano la regola piuttosto che l'eccezione per quasi tutti i dispositivi elettronici, tendenza che tuttora sembra continuare [14].

2.1.2 Caratteristiche

Per quanto riguarda le caratteristiche peculiari che, in generale, fanno di un dispositivo di elaborazione un sistema embedded, le principali sono [13]:

Funzionalità specifiche: un sistema embedded è tipicamente programmato per eseguire ripetutamente uno specifico compito/applicazione, consentendo a design time di minimizzare le risorse necessarie e massimizzarne l'efficienza e la robustezza.

Risorse limitate: tutti i sistemi embedded sono caratterizzati da vincoli progettuali circa le risorse disponibili (ram, rom, cpu, etc) molto più stringenti rispetto ai tradizionali sistemi di elaborazione.

Progettazione orientata all'efficienza: alla base di un qualsiasi dispositivo embedded vi è una progettazione orientata a minimizzare i consumi, le dimensioni e il peso, i costi, le dimensioni del codice da eseguire, nonché a garantire un'esecuzione il più efficiente possibile del dispositivo.

Affidabilità e sicurezza: per via delle caratteristiche sopracitate, importanti applicazioni dei dispositivi embedded riguardano contesti "critici" (*safety-critical*), come l'ambito biomedico, logistico, industriale e di controllo, nei quali è necessario un certo grado di affidabilità e disponibilità (quindi corretto e continuo funzionamento) nonché di sicurezza (ovvero funzionamento che non rechi danno a utenti o ambiente e nel rispetto delle norme di sicurezza e privacy).

[4] I^2C: *Inter Integrated Circuit*, è un sistema di comunicazione seriale, bifilare, master-slave e sincrono molto diffuso nell'ambito embedded.

Reattività e funzionamento *real-time*: i contesti di funzionamento critico precedentemente accennati, oltre a necessitare di un'adeguata garanzia di affidabilità e sicurezza, richiedono, a seconda dei contesti, anche un certo grado di reattività agli stimoli provenienti dall'ambiente in cui operano nonché un'elaborazione di questi, ed eventualmente delle relative reazioni, senza ritardi e quindi in un contesto *real-time*.

2.1.3 Architettura hardware e software

Come ogni sistema di elaborazione, anche i dispositivi embedded sono caratterizzati da una determinata architettura hardware e software, dipendenti fortemente dallo specifico dominio applicativo, dalle funzionalità richieste e dai requisiti di progetto. A differenza però dei sistemi *general-purpose*[5], nei sistemi embedded solo l'architettura hardware è sempre presente, mentre quella software, rappresentata da sistema operativo e software applicativo, in certi contesti è presente solo parzialmente o completamente assente.

Architettura hardware e tipologie di sistema embedded

Facendo riferimento all'architettura hardware, sempre presente a prescindere dalla tipologia di dispositivo e dall'ambito di utilizzo, essa è generalmente caratterizzata da: processore (CPU), memoria volatile (RAM), memoria persistente (ROM/FLASH), interfacce di comunicazione e I/O, circuiti e logiche specifiche per il dato dominio applicativo nonché un certo parco sensori/attuatori.

Relativamente a queste componenti, a seconda delle specifiche tecniche di progetto (p.e. prestazioni, consumi, etc), delle funzionalità e dei requisiti, corrispondono, in genere, particolari vincoli circa la tipologia e le caratteristiche di questi, che determinano quale tipologia di dispositivo embedded sia più indicato nel dato contesto applicativo. Tra queste componenti, l'elemento più importante da tenere in considerazione, in quanto da questo derivante la specifica tipologia di dispositivo embedded, nonché l'effettivo dominio di applicazione, è il processore. Quest'ultimo, nell'ambito dei sistemi embedded, può essere distinto principalmente in tre differenti tipologie, a seconda della flessibilità, le prestazioni offerte e i consumi [13]:

[5] **general-purpose**: fa riferimento a quella classe di sistemi di elaborazione in grado di essere utilizzati in differenti contesti applicativi. Fanno parte di tale categoria i desktop computer e i laptop.

Processori general-purpose (microprocessori): caratterizzati da un'architettura e un ISA (*Instruction Set Architecture*[6]) definiti, presentano il miglior grado di flessibilità grazie alla possibilità di programmarne il comportamento attraverso l'esecuzione di un'apposito software applicativo, al prezzo, però, di consumi maggiori e prestazioni inferiori. Vengono generalmente utilizzati in sistemi general-purpose, come PC e Laptop, nei quali è richiesta l'esecuzioni di diverse tipologie di applicazioni senza la necessità di elevate prestazioni o consumi minimi.

Processori single-purpose: costituiti da circuiti digitali progettati per implementare specifiche funzionalità, realizzano il comportamento richiesto esclusivamente attraverso la componente hardware e senza necessità di sistema operativo o software applicativo. In questo modo, sono in grado di offrire le massime prestazioni e la miglior efficienza in fatto di consumi, al prezzo di una maggiore complessità in fase di progettazione e una scarsissima flessibilità. Un esempio di tali processori (nonché di dispositivi embedded) sono gli ASICs (*Application-Specific Integrated Circuits*) e le FPGA (*Field Programmable Gate Array*). Mentre i primi sono circuiti appositamente progettati e realizzati per una specifica applicazione, le FPGA, molto meno costose rispetto ai primi, sono composte da un circuito base predefinito e programmabile al computer, mediante apposito software e linguaggio (p.e. Verilog, VHDL), sul quale poter definire il comportamento desiderato attraverso l'implementazione delle funzioni logiche richieste.

Application-Specific Processors (ASIP): aventi caratteristiche riconducibili ad entrambe le tipologie già trattate, ne rappresentano una via intermedia garantendo il giusto compromesso tra ottime prestazioni, consumi ridotti e flessibilità. Ciò è reso possibile, da un lato, dalla possibilità di definire il comportamento desiderato mediante la programmazione di un opportuno software applicativo, dall'altro, dalle elevate prestazioni ottenibili dall'hardware ottimizzato, per la data categoria di applicazioni a cui sono rivolti. Fanno parte di questa categoria i microcontrollori (MCU), i SoC (*System-on-a-Chip*[7]) e i DSP (*Digital*

[6]**Instuction Set Architecture:** set di istruzioni elementari eseguibili da un processore.

[7]**System-on-a-Chip:** letteralmente *Sistema su Circuito Integrato*, consiste in un chip integrante un'intero sistema. Generalmente, in tali sistemi si trovano integrati, oltre al processore centrale, il chipset, eventuali controller di memoria, la circuiteria di I/O e il sotto sistema video.

Signal Processor[8]). I primi due in particolare, MCU e SoC, caratterizzano due delle tipologie di sistema embedded più diffuse e utilizzate in circolazione: i sistemi a microcontrollore, detti *single-board microcontroller*, dotati di sola MCU e i sistemi a microprocessore, chiamati *single-board computer*, dotati generalmente di SoC.

Architettura software: sistema operativo e software applicativo

A seconda dell'architettura hardware utilizzata, e ovviamente del contesto applicativo e delle specifiche di progetto, vi sarà la presenza o meno del livello applicativo e/o del sistema operativo, e quindi una particolare architettura software.

Nel caso di dispositivi caratterizzati da processore *single-purpose*, dal momento che tutte le funzionalità richieste possono essere implementate via hardware, attraverso la progettazione e la realizzazione di circuiti digitali specifici, non è presente, in genere, né il software applicativo (a meno del caso di programmazione delle FPGA), e neppure il sistema operativo.

Nel caso invece dei processori *general-purpose* e *application-specific*, si possono trovare sia dispositivi caratterizzati dall'implementazione delle funzionalità richieste solo tramite software applicativo, sia attraverso il supporto di un sistema operativo apposito. Mentre per i processori *general-purpose* è necessariamente richiesto un livello applicativo che ne definisca le funzionalità, generalmente supportato da un apposito sistema operativo, per i processori *application-specific* esistono invece diverse alternative. Nel caso di sistemi basati su microcontrollore (come Arduino - 2.2), ad esempio, spesso è sufficiente il solo software applicativo (*firmware*) per implementare le funzionalità richieste, senza la necessità di un sistema operativo di supporto. Nel qual caso, invece, si abbia la necessità di gestire contesti di lavoro e funzionalità complesse (gestione della concorrenza, della rete, del file system, della condivisione delle risorse, etc.), oppure si abbia a che fare con sistemi basati su SoC (come Raspberry Pi - 2.3.1), occorre invece fare riferimento a un sistema operativo. Quest'ultimo, appositamente progettato per l'ambito embedded e real-time, si distingue dai tradizionali sistemi operativi per quella che è la maggiore compattezza e affidabilità, la migliore gestione delle limitate risorse hardware a disposizione, nonché per la maggiore reattività agli eventi e le particolari politiche di schedulazione studiate per i contesti

[8]**Digital Signal Processor:** letteralmente *Processore di Segnale Digitale*, è un processore dedicato e ottimizzato per eseguire in maniera estremamente efficiente sequenze di istruzioni ricorrenti (come ad esempio somme, moltiplicazioni e traslazioni), nell'ambito dell'elaborazione numerica dei segnali digitali.

real-time. Alcuni esempi sono: Free-RTOS, RT Linux ed Embedded Linux, Windows CE e Windows IoT Core nonché i mobile-os come Android, iOS, WindowsPhone, etc. È importante sottolineare come, oltre ai vari vantaggi derivanti dall'utilizzo di un sistema operativo (gestione delle risorse in generale: interruzione ed eventi, dispositivi I/O, timer, memoria, sincronizzazione, comunicazione inter-task e loro schedulazione, etc), l'utilizzo di questo porta anche all'abilitazione di opportuni *framework/middleware* in grado di fornire servizi ed API aggiuntive, utili per lo sviluppo, l'esecuzione e il debugging delle varie applicazioni embedded che si può voler realizzare (p.e. Java Embedded ME e Real-Time Java e varie piattaforme per l'IoT) [15].

2.1.4 Applicazioni dei sistemi embedded e Internet of Things

Mentre in origine il dominio di applicazione dei dispositivi embedded era, come già citato, confinato al settore militare e aerospaziale, con gli anni questi hanno avuto sempre maggiore diffusione nei più svariati settori. Partendo dall'elettronica di consumo (smartphone, videocamere) passando agli *home appliances* (termostati, impianti di illuminazione intelligenti, forno a microonde), all'*office automation* (fotocopiatrici, stampanti), e al *business equipment* (registratori di cassa, sistemi di allarme), fino a giungere all'*automotive* (sistema di trasmissione, navigazione, etc) e all'automazione e il controllo industriale (PLC[9], sistemi SCADA[10], etc).

Particolare applicazione dei sistemi embedded è, però, quella dei cosiddetti oggetti intelligenti: *smart things*. Nel corso degli anni, infatti, sempre più oggetti quotidiani hanno acquisito una capacità di calcolo non nativa, dovuta all'integrazione, in questi, di sistemi dedicati (embedded), in grado di renderli capaci di percepire l'ambiente circostante, di interagire con esso e persino di comunicare con altri dispositivi "intelligenti" per mezzo di un'opportuna rete: l'*Internet of Things* [16].

[9] **PLC:** *Programmable Logic Controller*, letteralmente *Controllore Logico Programmabile*, è un particolare controllore industriale specializzato nella gestione e controllo dei processi industriali.

[10] **SCADA:** *Supervisory Control And Data Acquisition*, letteralmente *Controllo di Supervisione ed Acquisizione Dati*, indica un sistema informatico distribuito per il monitoraggio e il controllo infrastrutturale e di processo industriali.

Internet of Things

Termine introdotto nel 1999 da Kevin Ashton, fondatore del *Auto-ID Center*[11] presso il *MIT*, nel contesto di alcuni progetti relativi alle tecnologie *RFID* e le relative applicazioni, l'*Internet of Things* (IoT, letteralmente *Internet delle cose*) rappresenta l'estensione della rete Internet al mondo degli oggetti e dei luoghi concreti. Mentre in origine l'obiettivo di Ashton era semplicemente quello di sfruttare la tecnologia RFID per automatizzare l'inserimento su Internet di dati real-time, provenienti dal mondo fisico mediante opportuni sensori, oggi il termine *Internet of Things* va ben oltre alla semplice identificazione degli oggetti. Oggi significa integrare una certa intelligenza (capacità computazionale) in oggetti di uso quotidiano in modo da renderli "*smart*" e in grado di fare molto più di ciò per cui erano stati originariamente pensati.

Attraverso opportune tecnologie abilitanti per la comunicazione (come Ethernet, Wifi, reti radio e dati, Bluetooth/BLE e ZigBee) e l'identificazione (come NFC, RFID e iBeacon) è infatti possibile creare un sistema pervasivo e interconnesso (*Pervasive Computing*). A partire da quest'ultimo, opportunamente coadiuvato da tecnologie e servizi basati sul *Cloud*, nonché da tecniche e architetture per la memorizzazione, l'elaborazione e l'accesso ai dati proprie del *Big Data*, si apre la strada a un gran numero di applicazioni possibili che spaziano dalla Logistica alla Domotica, dall'*Healthcare* alla sorveglianza e il monitoraggio nelle *Smart Cities*, fino ad arrivare all' Industria 4.0 e non solo [17, 18].

2.2 Arduino

Nell'intera scena dei sistemi embedded a disposizione, uno dei più famosi e utilizzati, specialmente in ambito prototipale, *maker* e *hobbista*, è sicuramente Arduino.

Arduino è una piattaforma di prototipazione elettronica *open-source*[12], basata su hardware e software flessibili e facili da usare. Progetto italiano nato all'*Ivrea Interaction Design Institute* come facile strumento per la prototipazione rapida, Arduino è rivolto ad artisti, designer, hobbisti e a

[11] **Auto-ID Center:** letteralmente *Centro di Identificazione Automatica*, è un'organizzazione di ricerca globale no-profit indipendente, con sede al MIT. A questa si deve la nascita della tecnologia RFID (*Radio-Frequency IDentification*) per l'identificazione/memorizzazione automatica di informazioni relative ad oggetti/animali/persone per mezzo di particolari tag (etichette elettroniche).

[12] **open-source:** termine usato per indicare un artefatto (generalmente software) non protetto da copyright e liberamente modificabile dagli utenti.

chiunque sia interessato a create oggetti o ambienti interattivi, senza però avere esperienza in ambito elettronico o di programmazione.

Le varie schede messe a disposizione della piattaforma Arduino sono tutte in grado di interagire con l'ambiente in cui si trovano, ricevendo informazioni da una grande varietà di sensori, controllando luci, motori e altri attuatori, attraverso il microcontrollore di cui sono dotate. Quest'ultimo, in particolare, può essere programmato utilizzando il linguaggio di programmazione Arduino (basato su *Wiring*[13]) e l'ambiente di sviluppo Arduino IDE (basato su *Processing*[14]), consentendo la realizzazione sia di semplici progetti standalone, sia di progetti più complessi e comunicanti con software in esecuzione su altri dispositivi.

Mentre l'ambiente di sviluppo può essere scaricato gratuitamente, la piattaforma hardware è distribuita agli hobbisti solo attraverso Internet e in versione pre-assemblata. Tuttavia, siccome i progetti di riferimento dell'hardware, così come il software, sono open-source, chiunque può liberamente costruirsi il proprio clone o adattare i progetti esistenti a seconda delle proprie esigenze [19, 20].

2.2.1 Storia

Arduino (derivato dal nome del bar che i fondatori erano soliti frequentare) è un progetto italiano nato nel 2005 all'*Ivrea Interaction Design Institute* ad opera del team composto da Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino e David Mellis, con lo scopo di rendere disponibile, nei progetti degli studenti di *Interaction Design*, un device per il controllo che fosse più economico rispetto agli altri sistemi di prototipazione disponibili all'epoca.

I progettisti sono riusciti nel loro intento creando una piattaforma di semplice utilizzo e, al tempo stesso, in grado di permettere una significativa riduzione dei costi rispetto a molti dei concorrenti sul mercato. Da semplice strumento per facilitare la realizzazione dei progetti in merito all' *Interaction Design*, il progetto Arduino ottenne ben presto successo anche in altri ambiti, tanto da ricevere una menzione d'onore nella sezione *Digital Communities del 2006 Ars Electronica Prix* e superando, nell'ottobre 2008, la quota di 50.000 esemplari venduti in tutto il mondo.

Non appena raggiunta una comunità più ampia, il progetto Arduino ha iniziato a evolversi per adattarsi alle nuove esigenze e sfide, differenziando

[13] **Wiring:** framework open-source per la programmazione di microcontrollori, `http://wiring.org.co`.

[14] `https://processing.org/`

la sua offerta da semplici schede a 8 bit a prodotti per applicazioni IoT, indossabili, stampa 3D e ambienti embedded. Oggi sul sito del progetto è disponibile un'ampia gamma di prodotti ufficiali che spazia dalle schede elettroniche e i moduli (schede di dimensioni minori) agli *shield*[15] di espansione fino a veri e propri kit.

Divenendo nel corso degli anni sempre più il cuore di migliaia di progetti in tutto il mondo, dagli oggetti di uso quotidiano a strumenti scientifici complessi, oggi Arduino può vantare una comunità mondiale di produttori - studenti, hobbisti, artisti, programmatori e professionisti - riunitasi attorno a questa piattaforma open-source e che contribuisce ogni giorno con un'incredibile quantità di conoscenza, resa accessibile a chiunque, di grande aiuto sia per i principianti che per gli esperti [19, 20].

2.2.2 Caratteristiche

Il progetto Arduino, quindi, abilita la possibilità a chiunque, indipendentemente dal background informatico ed elettronico, di poter realizzare le proprie idee attraverso una piattaforma semplice, compatta, estendibile, pronta all'uso e supportata online da una vastissima comunità e da un numero di progetti e tutorial in costante crescita. Detto questo, volendo riassumere le caratteristiche peculiari alla base del successo ottenuto negli anni da questo progetto, rispetto a molte altre soluzioni rivali, si possono citare le seguenti [19]:

Economicità: Le schede Arduino sono relativamente economiche rispetto ad altre piattaforme a microcontrollore concorrenti. La versione meno costosa del modulo Arduino può essere assemblata a mano, e anche i moduli Arduino pre-assemblati costano comunque meno di $50 (Arduino Uno REV - 2.2.3 - attualmente costa 20 Euro circa[16]).

Cross-platform: Il software Arduino IDE funziona sui più comuni sistemi operativi (Windows, Macintosh OSX e Linux) mentre la maggior parte dei sistemi a microcontrollore concorrenti sono limitati a Windows.

Ambiente di programmazione semplice e chiaro: Il software Arduino IDE è tanto facile da usare per i principianti, quanto flessibile da consentire anche agli utenti avanzati di trarne vantaggio. Inoltre, essendo

[15] **shield:** elemento progettato per essere collegato a una scheda in modo da fornirne funzionalità aggiuntive.

[16] `https://store.arduino.cc/arduino-uno-rev3`, ultimo accesso 30 Gennaio 2018.

basato sull'ambiente di programmazione *Processing*, risulta più confortevole a tutti quegli utenti (insegnanti e studenti in particolare) che hanno già una certa familiarità con quest'ultimo.

Software open-source ed estensibile: Il software Arduino è pubblicato sotto licenza GNU GPL (*General Public License*)[17] e pertanto disponibile per l'estensione da parte di chiunque, all'interno della community e abbastanza esperto, voglia dare il suo contributo al progetto. Inoltre, il linguaggio può essere esteso attraverso librerie scritte in C++, piuttosto che aggiungendo codice AVR-C (sul quale il progetto Arduino si basa) direttamente negli *sketch*[18], in modo da venire in contro alle proprie esigenze di progetto.

Hardware open-source ed estensibile: I progetti delle schede Arduino, similmente ai sorgenti software, sono pubblicati sotto una licenza *Creative Commons*[19], consentendo a progettisti di circuiti esperti di creare la propria versione del modulo, estenderlo e migliorarlo. Anche gli utenti relativamente inesperti possono eventualmente creare la propria versione del modulo su *breadboard*[20], in modo da capire come funziona e magari risparmiare denaro.

2.2.3 Arduino Uno

Nel contesto del caso di studio, la prototipazione (discussa nella sezione 3.3) si è valsa dell'utilizzo della piattaforma Arduino, fin'ora trattata, e nello specifico del modello Uno. La scelta di questo è dovuta, da un lato, nella longevità del progetto e la consolidata comunità che vi ruota attorno (riconducibili quindi a una grande disponibilità di documentazione e librerie), dall'altro, alle caratteristiche tecniche che lo rendono lo strumento perfetto per il caso di studio (prezzo esiguo, potenza, consumi e funzionalità in regola con la specifica dei requisiti).

Arduino Uno, il cui nome contrassegna la versione della piattaforma hardware e software, è la prima di una serie di schede USB-compatibili nonché il modello di riferimento per l'intera piattaforma Arduino. Grazie alla sem-

[17] **GNU General Public License:** licenza fortemente *copyleft* (ovvero liberamente utilizzabile, diffondibile e modificabile, nel rispetto di alcune condizioni essenziali) per il software libero. Quest'ultimo, nello specifico, deve rimanere libero indipendentemente dalla successione e il numero di modifiche apportate.

[18] **sketch:** come viene chiamato un progetto/programma in ambiente Arduino.

[19] `https://creativecommons.org/licenses/by-nc-sa/3.0/it/deed.it`

[20] **breadboard:** chiamata anche *basetta sperimentale*, è uno strumento utilizzato per creare prototipi di circuiti elettrici.

plicità e all'immediatezza caratteristiche della piattaforma Arduino, nonché alla robustezza e l'esperienza maturata nella comunità in tutti questi anni (che la rendono la scheda più utilizzata e documentata della piattaforma Arduino), essa rappresenta la miglior scheda con la quale potersi approcciare al mondo dell'elettronica e della programmazione per la prima volta [21].

Caratteristiche

La scheda, caratterizzata dal microcontrollore Atmel ATmega328P, dispone di 14 pin di input/output digitali (di cui 6 utilizzabili come uscite PWM[21]), 6 ingressi analogici, un cristallo di quarzo generante una frequenza di 16 MHz, un connettore USB, un jack di alimentazione, un'header ICSP(*In-Circuit Serial Programming*) per la programmazione e un pulsante di ripristino. Il tutto nelle dimensioni di una carta di credito e per un peso di soli 25g.

Per quanto riguarda le caratteristiche specifiche del microcontrollore, l'ATmega328P, abbiamo:

- CPU AVR8 con architettura RISC[22] a 8 bit (per i dati e 16 bit per le istruzioni), operante a 16 MHz e con 32 registri general-purpose;
- 2KB di memoria volatile (SRAM);
- 32KB di memoria persistente (Flash) per il firmware, di cui 0.5 usati per il *bootloader*[23];
- 1KB di memoria persistente (EEPROM) per i dati.

Inoltre, come la maggior parte dei microcontrollori e DSP in circolazione, anche l'ATmega328P è caratterizzato da un'architettura Harvard, ossia basata sulla separazione fisica della memoria contenente il codice (Flash) da

[21] **PWM:** letteralmente *Pulse-width Modulation*, è una tecnica utilizzata per emulare un segnale analogico in uscita a partire da uno digitale opportunamente modulato, ovvero come tensione media variabile dipendente dal rapporto tra la durata (*duty cycle*) dell'impulso positivo (p.e. 5V) e quello negativo (p.e. 0V).

[22] **RISC:** *Reduced Instruction Set Computer*, indica una tipologia di architetture per microprocessori caratterizzata dalla maggiore semplicità e linearità rispetto alla controparte CISC (*Complex Instruction Set Computer*). Tale semplicità è ricollegabile sopratutto al set di istruzioni (ISA) del processore che, essendo più semplici, possono essere eseguite in meno cicli di clock (1-2 generalmente) rispetto alle controparti per architettura CISC. Inoltre, l'architettura RISC è caratterizzata da un numero maggiore di registri interni e cache, riducendo la necessità di fare accessi a memoria esterna e limitando, quindi, i ritardi ad essi relativi.

[23] **bootloader:** essendo la scheda sprovvista di sistema operativo, il bootloader, pre-caricato all'interno della stessa, è il software adibito al caricamento e al lancio dell'esecuzione degli sketch compilati attraverso il computer.

quella contenente i dati (SRAM), consentendo l'utilizzo di tecnologie e tagli differenti per le due distinte memorie. La ragione di fondo di tale architettura sta nel voler ottimizzare al massimo l'esecuzione del codice, consentendo, in genere in un'unico ciclo, il *fetch* (caricamento) simultaneo dei dati e delle istruzioni da eseguire.

Altre importanti caratteristiche dell'ATmega328P sono:

Gestione degli eventi: attraverso due appositi pin digitali (2 e 3), la scheda Uno offre la possibilità di implementare sistemi reattivi agli eventi attraverso l'assegnamento di un apposito ISR (*Interrupt Service Routine* - ossia un blocco di codice eseguibile) come reazione ad un dato tipo di segnale (HIGH, LOW, RISING, FALLING e CHANGE) sul pin specificato.

Gestione del tempo: la gestione del tempo è di fondamentale importanza nella realizzazione di comportamenti *time-oriented*: misurazione di intervalli di tempo, segnali PWM, realizzazione di timeout e allarmi, realizzazione di forme di *multi-tasking* con *preemptive scheduling*, etc. Nel micro ATmega328P sono presenti tre timer interni (timer0, timer2 a 8bit e timer1 a 16bit) implementati come contatori e incrementati ad ogni ciclo di clock o multiplo di esso. Attraverso questi, è quindi possibile gestire opportune temporizzazioni nell'esecuzione di certe routine e task, nonché implementare un sistema *time-driven* attraverso un'opportuno ISR in modalità CTC (*Clear Time on Compare Match*[24]).

Supporto ai più utilizzati standard/bus di comunicazione seriale: il supporto agli standard/bus di comunicazione seriale più diffusi garantisce grande versatilità e un maggior numero di domini applicativi per i quai poter utilizzare la scheda Uno come sistema di controllo. L'ATmega328P, in particolare, supporta sia standard asincroni sia sincroni (più complessi ma anche piu veloci rispetto ai primi). I primi comprendono USB e TTL (multiplexata sui pin 0 -RX e 1 -TX) mentre i secondi I^2C (nativo sui pin analogici A4 -SDA e A5 -SCL) e SPI (sui pin 10 -SS, 11 -MOSI, 12 -MISO e 13 -SCK).

Infine, per quanto concerne l'alimentazione della scheda, questa, operante ad un voltaggio di 5V, può essere alimentata sia per mezzo dell'ingresso USB

[24] **Clear Time on Compare Match:** è una modalità di gestione delle interruzioni relative ai timer. Questa si basa sul lanciare un'interruzione ogni volta che il contatore associato al timer raggiunge un certo valore specificato.

(quindi attraverso PC o power bank a 5V), sia attraverso una batteria (o altro alimentatore) collegata per mezzo dell'apposito jack (a 7-12V) [21, 22].

Programmazione

La programmazione di Arduino Uno avviene mediante un sistema esterno (PC) con il quale editare e compilare, attraverso l'IDE di Arduino (oppure Eclipse, opportunamente configurato, o anche AVR Studio), gli sketch da caricare sulla scheda. Quest'ultima, in particolare, essendo sprovvista di sistema operativo, non fa altro che eseguire il binario caricato, caratterizzato da un "main-loop" contenente le istruzioni da eseguire ripetutamente. Al fine di poter trasferire sulla scheda gli sketch realizzati, è pre-caricato nella memoria Flash un bootloader adibito alla comunicazione col PC (tramite USB) nonché al caricamento del binario compilato affinché sia immediatamente eseguito. Alternativamente all'utilizzo di un computer per la programmazione del micro, è possibile utilizzare l'interfaccia ICSP (*In-Circuit Serial Programmer*) dedicata, attraverso la quale programmare la scheda mediante un'apposito dispositivo "programmer" esterno.

Il framework utilizzato, in generale, per la programmazione delle schede Arduino mediante computer è, come in precedenza accennato, Wiring. Quest'ultimo, basato su GNU GCC, è composto da una serie di librerie custom per il controllo dell'hardware e permette la programmazione dei microcontrollori per mezzo del linguaggio C/C++ (e la relativa compilazione del codice attraverso l'AVR-gcc), consentendo la realizzazione di sketch e librerie ad alto livello in grado di poter essere utilizzate in tutte le schede della piattaforma Arduino [21, 22].

2.3 Tecnologie programmabili in ambito Educational

Nonostante il nome "Smart T-Shirt" possa far pensare che il caso di studio abbia come destinazione il mondo dei *wearables* (di consumo), l'obiettivo principale del progetto è invece quello di realizzare uno strumento che funga da supporto al progetto COGITO, collocandosi quindi, più propriamente, nel dominio del *Pensiero Computazionale* e del *Costruzionismo*, del *coding* e della *Digital Education* in generale. Infatti, data la natura, gli obiettivi e gli strumenti alla base del progetto (diffondere il pensiero computazionale attraverso il coding, interagire col mondo fisico - la matrice di led - attraverso un dispositivo embedded - Arduino - programmato mediante un linguaggio a blocchi come Scratch/Snap!), *Smart T-Shirt* rientra indubbiamente nel-

la categoria di tecnologie/dispositivi programmabili, utilizzabili in ambito Educational.

Anche in tale settore, e in particolare nel contesto dello STEM[25], negli ultimi anni hanno infatti avuto ampia diffusione, trovando sempre maggior impiego, i dispositivi e le tecnologie embedded programmabili, attraverso le quali poter introdurre agli studenti concetti più avanzati e "accattivanti", come l'interazione col mondo fisico, senza tuttavia la necessità di essere degli esperti di elettronica.

Il panorama delle piattaforme embedded utilizzabili in ambito Educational, in particolare, ha visto negli ultimi anni la nascita di numerosi nuovi progetti (come micro:bit[26], CodeBug[27], PicoBoard[28], etc) che vanno ad affiancarsi a piattaforme consolidate e con più maturata esperienza. Tra queste ultime, e tra le più famose e utilizzate, troviamo sia piattaforme a basso costo e basate su microcontrollore come Arduino, già ampiamente trattato nella sezione 2.2.3, o su microprocessore come *Raspberry Pi*, sia piattaforme più costose come il già accennato *LEGO Mindstorms* (1.3.1) e *LEGO WeDo*.

2.3.1 Raspberry Pi

Raspberry Pi è una serie di piccoli *single-board computer*[29] sviluppata nel Regno Unito dalla *Raspberry Pi Foundation* per promuovere l'insegnamento dell'informatica di base nelle scuole e nei paesi in via di sviluppo.

Presentata pubblicamente per la prima volta nel Febbraio del 2012, la scheda è progettata per ospitare sistemi operativi basati sia su kernel Linux (come Raspbian, Ubuntu MATE, Snappy Ubuntu, etc) sia su altri sistemi operativi come RISC OS, FreeBSD e Windows 10 IoT Core. In quanto *single-board computer*, Raspberry risulta ottima per la prototipazione dei sistemi embedded avanzati, per i quali le soluzioni basate su microcontrollore non fornirebbero sufficienti capacità di elaborazione e memoria. Oltre all'ambito della prototipazione, però, Raspberry trova comunque innumerevoli altre

[25] **STEM:** acronimo che sta per *Science, Technology, Engineering and Mathematics*, indica l'insieme di corsi e scelte educative volte a incrementare la competitività e le competenze in campo scientifico e tecnologico nei giovani. Nato nei primi anni 2000, si basa su un approccio interdisciplinare nel quale gli studenti vengono spinti a creare connessioni tra le nozioni apprese nello studio delle discipline tecnico-scientifiche e la realtà concreta, con l'obiettivo di preparare le nuove generazioni alla cosiddetta *New Economy*.

[26] `http://microbit.org/`

[27] `http://www.codebug.org.uk/`

[28] `https://wiki.scratch.mit.edu/wiki/PicoBoard`

[29] **Single-Board Computer:** scheda caratterizzata da un SoC (*System on a Chip*) con prestazioni e funzionalità paragonabili ad un tradizionale desktop-computer ma a costi, ingombro e consumi notevolmente inferiori.

applicazioni: si va dal settore educativo, con progetti di elettronica e coding, a quello making e videoludico (*Retropie*[30]), fino ad un uso più ordinario come l'elaborazione di testi, la navigazione in Internet o addirittura vedere video in alta risoluzione, proprio come si farebbe su un comune PC desktop.

Come Arduino, col quale condivide la motivazione alla nascita del progetto, ovvero la diffusione dell'insegnamento dell'informatica nelle scuole, oggi Raspberry può vantare, insieme al grande successo e la grande diffusione, una vasta comunità e una grandissima quantità di progetti e riferimenti in rete, risultando, al pari del microcontrollore Made in Italy per eccellenza, una delle piattaforme più diffuse per la prototipazione dei sistemi embedded e l'IoT [23, 24, 25].

Caratteristiche

Negli anni sono stati rilasciati diversi modelli (Zero, A e B, in ordine crescente di complessità, prestazioni e costi) e generazioni di schede Raspberry Pi, differenti per la capacità della memoria, le prestazioni in generale e il supporto alle periferiche, tutte caratterizzate da un SoC Broadcom integrante, oltre al processore ARM, anche un co-processore grafico. Le varie schede sono alimentate attraverso microUSB (5V), con consumi variabili dai 100mA-350mA per il modello Zero, 200mA per il modello A+ di I gen, fino ad arrivare ai 300mA-1.34A per il modello B di III gen, e caratterizzate da prezzi piuttosto contenuti che vanno dai $5 fino ai $35.

La velocità della CPU varia da 700 MHz, per le prime generazioni, a 1.2 GHz (con architettura quad-core a 64 bit - ARM Cortex-A53), per la terza, mentre la RAM da 256 MB a 1 GB. Per quanto riguarda l'archiviazione del sistema operativo e dei programmi, non viene utilizzato alcun hard-disk ma semplici schede SD nelle dimensioni SDHC o MicroSDHC. In ciascun modello sono inoltre presenti da una a quattro porte USB, per la comunicazione col PC e le periferiche (come tastiera, mouse, etc), una porta HDMI, per l'uscita video, e un jack standard da 3,5 mm per l'uscita audio. Per quanto riguarda, invece, l'interfacciamento a "basso-livello" col mondo fisico e l'esterno, sono disponibili un certo numero di pin GPIO (*General Purpose Input-Output*), in grado di supportare anche alcuni dei più comuni protocolli di comunicazione in ambito embedded come I^2C. Sempre a proposito di comunicazione, si sottolinea come i più recenti modelli (Pi 3 e Pi Zero W) trovino anche integrati sia il modulo Wi-Fi 802.11n sia quello Bluetooth.

Per quanto riguarda i sistemi operativi supportati dai vari modelli, nonostante sia possibile utilizzare sistemi operativi di terze parti come Ubuntu,

[30] `https://retropie.org.uk/`

Windows 10 IoT, RISC OS, etc, Raspberry Pi Foundation fornisce direttamente Raspbian, una propria distribuzione Linux basata su Debian, raccomandando ad ogni modo, soprattutto ai neofiti, l'installazione del sistema operativo desiderato per mezzo dell'apposito distro-manager NOOBS[31](*New Out Of Box Software*). Infine, per quanto riguarda la programmazione, sono consigliati i linguaggi Python e Scratch, nonostante ne siano supportanti anche molti altri [24, 26].

Educational Mission

Fin dalla nascita del progetto, ad opera di Eben Upton della Broadcom e un gruppo di insegnanti, accademici e appassionati di computer, mossi dalla necessità di voler diffondere nelle nuove generazioni la curiosità verso il coding e la computazione, il principale obiettivo perseguito dalla Raspberry Pi Foundation è sempre stato quello di diffondere nelle persone le abilità di coding e digital-making:

"La missione di Raspberry Pi Foundation è quella di mettere il potere del digital making nelle mani delle persone di tutto il mondo. I nostri programmi di formazione gratuiti fanno parte della nostra strategia per raggiungere questa missione impegnativa: riteniamo che tutti dovrebbero avere l'opportunità di sviluppare le proprie competenze informatiche e di digital making" [27].

A tal proposito, nel 2014 è stato assunto un certo numero di membri della comunità, tra cui ex insegnanti e sviluppatori di software, per rendere disponibili sul proprio sito Web, in maniera assolutamente gratuita, una vasta serie di risorse e contenuti per l'apprendimento. La nuova area del sito[32], appositamente progettata per insegnanti, studenti e maker, è infatti piena di materiale e progetti gratuiti per l'apprendimento del coding, del pensiero computazionale e del making, il tutto sotto licenza *Creative Commons*. Per gli insegnanti, ad esempio, sono resi disponibili interi schemi di lavoro, completi di piani di lezione, collegati al curriculum di Informatica nel Regno Unito. Per tutti coloro che invece desiderano imparare da soli, è disponibile tutto l'occorrente necessario ad approcciarsi con una scheda Raspberry Pi, a partire dagli usi che se ne possono fare e come utilizzarla, nonché numerosi contenuti e-learning e guide passo-passo per la creazione dei vari progetti messi a disposizione su sito [24, 28].

La Raspberry Pi Foundation ha inoltre creato un corso di formazione specifico per tutti quegli insegnati che vogliono cimentarsi nella nuova era dell'insegnamento. *Picademy*, questo il nome del programma di sviluppo

[31] `https://www.raspberrypi.org/documentation/installation/noobs.md`
[32] `https://www.raspberrypi.org/education`

professionale gratuito e *face-to-face* offerto, è rivolto a tutti quegli educatori intenzionati a intraprendere un percorso di *digital making* e *computing*, indipendentemente dal ruolo, la disciplina insegnata e il grado della scuola in cui insegnano. Attraverso un programma di formazione di due giorni, tenuto nel Regno Unito e nel Nord America, i vari partecipanti possono acquisire le conoscenze e le abilità di cui avere bisogno per iniziare ad utilizzare il computer, e le tecnologie in genere, nell'ambito di un'insegnamento attivo, sicuro e creativo, con l'opportunità di entrare a far parte di una grande comunità di professionisti e appassionati del *digital making* [24, 27, 29].

2.3.2 LEGO Mindstorms e WeDo

La famosissima casa di giocattoli danese, LEGO, ha introdotto ormai da parecchi anni la possibilità di animare le proprie costruzioni per mezzo di appositi blocchi programmabili. L'integrazione nei blocchi LEGO della possibilità di essere programmati è infatti da ricondursi ai primi anni '80 e alla collaborazione di LEGO col Professor Seymour Papert (1.2.1), al quale si deve una partnership di oltre 30'anni tra LEGO Group e il MIT Media Lab [30].

Questa collaborazione con Papert, nonché con Mitchel Resnick e tutto il MIT Media Lab in generale, aveva l'obiettivo di realizzare un progetto che consentisse di unire la praticità e la creatività caratteristici della composizione dei famosi blocchi per costruzioni con la versatilità e l'espandibilità del linguaggio LOGO (1.3.1), consentendo ai bambini la possibilità di automatizzare e controllare le costruzioni che realizzavano. Da tale collaborazione nacque LEGO/Logo, oggi precursore della linea Mindstorms, consistente in un ambiente computerizzato in grado di portare le attività di progettazione e invenzione all'interno delle scuole. Usando LEGO/Logo, uno studente poteva ad esempio costruire una macchina con i tradizionali pezzi LEGO (inclusi però ingranaggi, motori e sensori), collegare questa a un computer e scrivere un programma per controllarla attraverso il Logo (opportunamente esteso per supportare i blocchi-sensore, blocchi-attuatore, etc). L'obiettivo del progetto non era solo quello di fornire un contesto significativo e motivante nel quale trattare i concetti scientifici tradizionali, ma anche quello di consentire agli studenti l'esplorazione di idee progettuali e ingegneristiche raramente affrontate nei contesti scolastici tradizionali [31, p.14].

Oltre che per tale collaborazione, Seymour Papert è stato, in generale, un partner molto importante per LEGO Group per via delle sue idee e le sue teorie circa l'apprendimento dei bambini, tanto da essere il primo ad aver ricevuto la nomina di *Professore LEGO per la ricerca sull'apprendimento.*

Gli studi di Papert su come i bambini percepiscono il mondo hanno infatti, da una parte, influenzato in maniera unica e duratura la visione che l'azienda aveva per i bambini, dall'altra, ispirato in LEGO tutta una filosofia che ha guidato alcuni dei suoi prodotti più influenti e che continua tuttora a essere una straordinaria ispirazione per tutte le entità, i fan e le comunità LEGO che supportano la creatività, il gioco e l'apprendimento dei bambini. Secondo Papert:

"La conoscenza che i bambini sperimentano quando interagiscono con la tecnologia non è separata dalla loro passione per i giocattoli e dalle cose che fanno prima di andare a scuola. I bambini imparano meglio quando sono impegnati attivamente nella costruzione di qualcosa che ha un significato personale per loro, sia esso un poema, un robot, un castello di sabbia o un programma per computer".

Lo stesso vale per i giochi LEGO. Essi rappresentano un modo pratico e mentale di apprendere le cose attraverso il gioco, insegnando ai bambini a imparare in maniera costruttiva e creativa, in mondo da poterlo fare durante tutto il corso della loro vita. Secondo LEGO:

"L'approccio pratico all'apprendimento consiste nel fornire agli studenti l'opportunità di sperimentare ciò che li circonda come una forma di risoluzione dei problemi attraverso il coding. Ciò favorisce lo sviluppo di creatività e collaborazione, motivazione e autodirezione. Si tratta di improvvisazione, scoperta ed esperienze di apprendimento ludico con rilevanza nel mondo reale. La combinazione unica tra la costruzione attraverso i familiari mattoncini LEGO e l'utilizzo di un software di codifica di facile utilizzo, rendono il coding divertente e pertinente per gli studenti delle scuole elementari e medie, aiutandoli ad avere successo nelle varie sfide del curriculum STEM" [30, 32].

A tal proposito, LEGO mette oggi a disposizione due linee di giocattoli appositamente dedicate: *Mindstorms*, la cui prima versione è stata rilasciata nel 1998, e *WeDo*, rilasciato nel 2006. Queste, partendo entrambe dalla famigliarità che i bambini già hanno nell'utilizzare i blocchi componibili per realizzare le proprie costruzioni, aggiungono la componente programmabile (embedded) attraverso la quale poter automatizzare e controllare le proprie opere. Mindstorms, nonostante sia pensato principalmente per le scuole medie, trova applicazione nelle mani di ogni genere di appassionato, mentre WeDo, appositamente progettato per la scuola elementare, è lo strumento perfetto per far approcciare i bambini provenienti dal tradizionale mondo LEGO ai nuovi e particolari blocchi in grado di animarlo.

Mindstorms

Mindstorms[33], originariamente ispirata dalle idee di Seymour Papert, è una linea di prodotti LEGO piuttosto costosa (qualche centinaia di dollari per kit) che combina i tradizionali mattoncini per costruzioni con particolari mattoncini programmabili dotati di motori elettrici, sensori, e pezzi di *LEGO Technic* (come ingranaggi, assi e parti pneumatiche), per costruire robot e altri sistemi automatici e interattivi [33]. Oltre alla versione commerciale, orientata all'ambito ludico e maker, LEGO realizza anche una versione alternativa del kit, chiamata *LEGO Mindstorms Education*[34], appositamente progettata per l'utilizzo scolastico e l'ambito STEM. Quest'ultima, partendo dallo stesso blocco programmabile, si differenzia da quella commerciale non tanto per le caratteristiche hardware, come potrebbe essere un differente parco sensori, quanto per l'ambiente di sviluppo utilizzato. Nella versione progettata per le scuole sono infatti previsti, oltre ad apposito materiale didattico, specifici percorsi formativi nell'ambito delle sfide STEM, mancanti invece nella versione commerciale.

A prescindere dalla versione di riferimento, Il blocco più importante all'interno di ogni kit Mindstorms è quello programmabile e di controllo, rappresentato, a partire dalla terza generazione (2013), dall'*EV3*. Dotato di sistema operativo Linux, l'EV3 è caratterizzato da un processore ARM9 da 300 MHz, 64 MB di RAM e 16 MB di memoria Flash, espandibili con scheda Micro SD fino ad un massimo di 32 GB. Per quando riguarda le interfacce di comunicazione con l'esterno, è invece dotato di un connettore Mini-USB 2.0 e di un modulo Bluetooth integrato, per l'interfacciamento col computer, nonché di un'apposita porta USB per la connessione del modulo Wi-Fi e per la comunicazione (in cascata) con altri blocchi EV3 (fino ad un massimo di 4). Per quanto concerne poi il parco sensori e attuatori, sono disponibili due tipologie di servo-motori (LARGE e MEDIUM) nonché una vasta gamma di sensori, tra cui: giroscopio, sensore tattile, sensore di luminosità e colore, sensore a ultrasuoni, sensore di temperatura e sensore a infrarossi (utilizzato, oltre che per il rilevamento della prossimità degli oggetti, anche per il controllo remoto). Per poter comandare i vari motori e sensori, l'EV3 presenta poi quattro uscite per i primi e altrettante entrate per secondi, tutte caratterizzate da connettori RJ12. L'EV3 è infine dotato di display LCD monocromatico con risoluzione 178x128 pixel, quattro pulsanti di navigazione frontali, un altoparlante, alimentazione per mezzo di batterie AA

[33] Versione commerciale: `https://www.lego.com/it-it/mindstorms`;

[34] `https://education.lego.com/en-us/middle-school/intro/c/ev3-everyone-can-code`

e funzionalità Auto ID per il riconoscimento dei vari componenti hardware esterni (sensori, attuatori, altri EV3 Bricks etc) [34].

Per quanto concerne, invece, la programmazione dei prodotti Mindstorms, LEGO mette a disposizione esclusiva per la versione Education due ambienti appositamente studiati per seguire i percorsi didattici del curriculum STEM. Attraverso tali ambienti è infatti possibile accedere ai percorsi di coding, problem solving, progettazione di robot, nonché quelli più avanzati relativi alla sperimentazione scientifica e all'ambito spaziale. Questi due ambienti sono l'*EV3 Software Lab*, per l'ambito desktop e i sistemi operativi Windows e MacOS, e l'*EV3 Programming App*, per i dispositivi mobili (iPad, e tablet Android e Windows 10) e i Chromebook. Nonostante questa seconda versione, più "leggera", ponga alcune limitazioni circa i percorsi didattici affrontabili e gli strumenti per lo sviluppo e la programmazione disponibili, come la versione desktop offre però un'ambiente di programmazione caratterizzato da un linguaggio visuale e un'interfaccia *drag-and-drop*, dominanti ormai nell'ambito educativo (Scratch, Snap!, etc), attraverso il quale definire il comportamento desiderato per il proprio progetto in maniera estremamente semplice e intuitiva [35]. Il linguaggio messo a disposizione è composto da varie categorie di blocchi funzionali componibili, attraverso i quali definire il comportamento voluto. Attraverso tali categorie, identificate da un colore specifico a seconda della funzione, i vari blocchi sono distinti in: *Action Blocks*, per il controllo dei vari attuatori (motori, display, altoparlante, etc), *Flow Blocks*, per il controllo del flusso di esecuzione (cicli, blocchi condizionali, etc), *Sensor Blocks*, per il controllo dei vari sensori, *Data Blocks*, per le variabili, gli operatori e le funzioni matematiche, e infine *Advanced Blocks*, per particolari funzionalità avanzate (come accesso al Filesystem, Bluetooth, etc). Oltre ai blocchi predefiniti, gli ambienti di sviluppo consentono inoltre la definizione e l'aggiunta di propri blocchi (*My Blocks*), attraverso i quali rendere il codice riusabile e modulare [34]. Alternativamente alle piattaforme fornite da LEGO appena trattate, per la programmazione del dispositivo è possibile utilizzare l'interfaccia fisica presente direttamente sul blocco EV3, piuttosto che linguaggi e ambienti alternativi sia visuali come LabVIEW[35], OpenRoberta[36], e lo stesso Scratch (1.3.2), sia testuali come RobotC[37] e EV3 Basic[38] ed EV3dev[39].

[35] http://www.legoengineering.com/platform/labview
[36] https://www.open-roberta.org/lab
[37] http://www.robotc.net
[38] https://sites.google.com/site/ev3basic
[39] http://www.ev3dev.org

WeDo

Oltre a Mindstorms, orientato agli studenti delle scuole medie, LEGO Education ha introdotto, a partire dal 2006, WeDo, una piattaforma appositamente progettata per l'utilizzo nelle scuole elementari. Disponibile oggi solo nella versione 2.0, WeDo, combinando i mattoncini LEGO per le costruzioni con un'ambiente di sviluppo facile e intuitivo, consente di portare la robotica educativa e il coding nelle scuole elementari, mettendo a disposizione degli insegnanti percorsi didattici preimpostati e progetti guidati pronti all'uso.

Il kit base, acquistabile a poco meno di $200, consiste, oltre ai mattoncini per le costruzioni, di un mattoncino programmabile, chiamato *SmartHub*, un servo-motore, un sensore di movimento e uno di inclinazione, nonché del software per la programmazione e dei contenuti didattici per l'insegnate. Lo SmartHub, ovvero il blocco principale di ogni progetto, è caratterizzato da un modulo Bluetooth 4.0/BLE integrato, attraverso il quale interfacciarsi col dispositivo di programmazione, due porte di I/O per l'interfacciamento con i sensori, i motori ed eventualmente altri due SmartHub, e una superficie RGB programmabile. La programmazione dello SmartHub può avvenire sia attraverso l'apposito ambiente di sviluppo, simile a quello per Mindstorms Education, disponibile per desktop e dispositivi mobili, sia attraverso il linguaggio Scratch. Attraverso l'ambiente di sviluppo messo a disposizione, lo SmartHub può essere programmato esattamente come si farebbe in Mindstorms, ossia in maniera facile e intuitiva mediante l'apposita interfaccia *drag-and-drop* e il linguaggio a blocchi messi a disposizione. A supporto degli insegnati, è infine reso disponibile un *Curriculum Pack* contenente più di 40 ore di attività pratiche, lezioni e strumenti di valutazione già pronti all'uso [36, 37].

2.4 Sommario

In questo secondo capitolo è stato trattato il contesto tecnologico in cui collocare il caso di studio Smart T-Shirt. Partendo dalla definizione, la storia, le caratteristiche e gli utilizzi dei Sistemi Embedded, è stato trattato il progetto italiano Arduino e la sua più famosa scheda a microcontrollore Arduino Uno. Dopo la definizione delle specifiche tecniche di tale scheda, utilizzata durante la fase di prototipazione del caso di studio, sono state trattate alcune delle più famose piattaforme per la programmazione di sistemi embedded utilizzate oggi in ambito Educational: *Raspbarry Pi*, *LEGO Mindstorm*, e *WeDo*.

Una volta trattati il contesto applicativo (progetto COGITO e diffusione del pensiero computazionale mediante il coding) e tecnologico (sistemi embedded programmabili in ambito Educational) nei quali collocare il caso di studio, seguono nella seconda parte della tesi la trattazione di questo e del progetto realizzato. Partendo dall'introduzione del progetto Smart T-Shirt (cap. 3), nato all'interno del contesto di COGITO e rappresentante il caso di studio, segue infatti il progetto derivante dal mio contributo all'interno del team Smart T-Shirt nonché il reale apporto progettuale insito in questa tesi di laurea: *Snap2ino* (cap. 4).

Parte II

I progetti Smart T-Shirt e Snap2ino

Capitolo 3

Il progetto Smart T-Shirt

Una volta descritto il contesto applicativo (cap. 1) e tecnologico (cap. 2) in cui collocare il progetto Smart T-Shirt, è dunque possibile definire, relativamente al primo, l'idea che ne è alla base e l'obiettivo perseguito, relativamente al secondo, le ragioni che hanno portato alla scelta delle tecnologie descritte.

Partendo da quelle che sono le generalità del progetto, viene descritta l'idea che ne è alla base, l'obiettivo e in fine il ruolo nel contesto del progetto COGITO (3.1). In seguito, è descritto l'approccio allo sviluppo del progetto, i requisiti e le relative funzionalità principali (3.2), nonché gli elementi critici e centrali all'interno della progettazione dello stesso (3.3). Infine, viene definito il ruolo del mio contributo all'interno del progetto e nel team Smart T-Shirt (3.4).

3.1 Idea, descrizione del progetto e obiettivo

Con l'introduzione del *Micromondo della Matrice*, nel terzo anno del progetto COGITO (1.4.2), si è deciso di offrire ai bambini la possibilità di andare oltre a quanto realizzato in Snap! sui tablet, rendendo loro possibile averne un riscontro concreto anche nel mondo fisico. Di qui l'idea di realizzare una maglietta in grado di mostrare quanto i bambini avrebbero realizzato durante le lezioni attraverso Snap!, per mezzo di una matrice di led appositamente applicata sul petto di questa.

Nato dal team formato dal Prof. Alessandro Ricci, l'Ing. Angelo Croatti e la Dottoressa Laura Tarsitano, il progetto Smart T-Shirt rappresenta la controparte fisica di quello che è il *Micromondo della Matrice* nel progetto COGITO. Infatti, mentre quest'ultimo rappresenta l'ambiente di programmazione con cui definire il comportamento dei led (i.e. il disegno, l'anima-

zione, etc), il progetto Smart T-Shirt ha come obiettivo quello di realizzare, in maniera incrementale, la maglietta vera e propria con la matrice sul petto.

In particolare, il team mira a fornire a ognuno dei bambini una di queste magliette "smart", consentendo loro la possibilità di programmarle e personalizzarle come meglio desiderano, attraverso uno strumento tanto semplice quanto potente come Snap!. Una volta personalizzate, le magliette potranno poi essere indossate dai vari bambini durante la festa di fine anno scolastico, mettendo in scena un vero e proprio spettacolo di luci e animazioni come mai visto prima nelle scuole.

Come già accennato nella sezione 2.3, tale progetto, quindi, non si pone come destinazione il mondo dei *wearables* di consumo, ma, piuttosto, quello delle tecnologie e delle piattaforme programmabili e utilizzabili in ambito Educational. Infatti, la Smart T-Shirt non è studiata tanto come vero e proprio indumento da sfoggiare, quanto come strumento di supporto alle attività di *coding* e *pensiero computazionale* nell'ambito del progetto COGITO.

3.2 Approccio allo sviluppo, funzionalità e requisiti

Come già accennato, l'approccio scelto nello sviluppo di tale progetto è di tipo incrementale e caratterizzato dal ricorso alla *prototipazione evolutiva*[1] nell'implementazione delle varie funzionalità richieste. Infatti, quello della matrice di led sul petto della maglietta è solo il primo passo di un progetto che può evolvere ulteriormente. I passi successivi potrebbero ad esempio spaziare dall'integrazione di opportuni sensori (p.e. movimento, prossimità, etc), in modo da poter interagire con l'animazione della matrice, fino ad arrivare ad integrare elementi più interessanti, come moduli bluetooth/wifi, in modo da far comunicare ciascuna Smart T-Shirt e realizzare un vero e proprio *sistema ad agenti*[2].

Per quanto riguarda, tuttavia, gli obiettivi per l'anno scolastico in corso, la realizzazione del primo prototipo si limita alla semplice maglietta con la matrice di led sul petto, mostrante l'animazione programmata dal bambino.

A tal proposito, il prototipo deve consentire la programmazione della matrice di led attraverso il *Micromondo della Matrice* (1.4.2) realizzato nell'ambito del progetto COGITO, in modo da consentire ai bambini l'utilizzo

[1] **prototipazione evolutiva:** per le varie funzionalità da implementare/testare si realizza un unico prototipo che viene fatto evolvere nel tempo fino a diventare il prodotto finale.

[2] **sistema ad agenti:** insieme di agenti (entità caratterizzate da un certo grado di autonomia) situati in un certo ambiente e interagenti tra loro mediante un'opportuna organizzazione.

di strumenti e metodologie familiari e con i quali trovarsi a proprio agio nella fase di programmazione e personalizzazione della propria maglietta. La sfida, e la principale criticità del progetto, consiste quindi nel consentire persino a un bambino di programmare e personalizzare la propria maglietta, in maniera facile, intuitiva e attraverso un linguaggio altrettanto facile e intuitivo come Snap!.

Infine, siccome tale maglietta ha come target di utenza i bambini delle scuole elementari, occorre prestare la massima attenzione all'aspetto relativo all'usabilità, da intendersi, in questo particolare contesto, come la combinazione di comodità, mobilità e, soprattutto, sicurezza da garantire durante l'utilizzo da parte del bambino.

3.3 Progettazione

Una volta definito l'obiettivo, le funzionalità e i requisiti del progetto Smart T-Shirt, ossia che cosa realizzare, giunge il quesito su come fare. La risposta a tale domanda è data dalla fase di progettazione del caso di studio, riguardante, essenzialmente, i due seguenti aspetti: da un lato, quello più cruciale, la programmabilità della maglietta, dall'altro, l'usabilità, ossia la comodità, mobilità e sicurezza nell'utilizzo della stessa

Programmabilità

Per quanto riguarda la programmabilità della maglietta, l'aspetto principale da gestire è la scelta della tecnologia/dispositivo da utilizzare per la gestione della matrice. In particolare, il dispositivo deve consentire l'interfacciamento con una generica matrice di led (come nel nostro caso la *Adafruit NeoPixel Matrix 8x8*[3]) ed essere quanto più possibile compatibile col linguaggio Snap!. Inoltre, il dispositivo candidato deve essere tanto piccolo e maneggevole da poter essere integrato facilmente nella maglietta senza dare fastidio al bambino e in grado di poter essere alimentato da un semplice power-bank da 5V tascabile. La scelta di tale dispositivo va quindi a ricadere nel settore embedded (cap. 2), caratterizzato da dispositivi compatti e a basso consumo, perfetti per il caso di studio della Smart T-Shirt.

Nel sito ufficiale di Snap![4], nella sezione dedicata ai dispositivi compatibili, spiccano, tra tutti, la piattaforma Arduino (largamente trattata nella sezione 2.2) e Raspberry Pi (2.3.1). A tal proposito, nonostante la grande

[3] `https://www.adafruit.com/product/1487`
[4] `http://snap.berkeley.edu/index.html`

community, la quantità e la varietà di librerie, nonché di tutorial, a disposizione in rete relativamente a entrambi i progetti, la scelta è ricaduta sulla scheda Arduino Uno (2.2.3). Questa, direttamente disponibile presso la sede universitaria, presenta infatti numerosi vantaggi rispetto alla controparte basata su microprocessore. Oltre ai minori costi e consumi, infatti, la piattaforma Arduino, essendo basata sul framework Wiring, consente sia una maggiore generalità e portabilità dell'idea (dovute al fatto di poter migrare facilmente dalla scheda Uno a versioni più compatte come Arduino Micro e Nano, e in generale verso tutti i sistemi a microcontrollore basati sul medesimo framework), sia maggiore precisione circa la gestione e il controllo del tempo e dei GPIO.

Tuttavia, per quanto riguarda le soluzioni proposte[5] sul sito, circa l'interfacciamento tra Snap! e Arduino, queste non risultano essere sufficientemente adeguate alle esigenze del caso di studio. Tra le principali limitazioni vi è il supporto ad un limitato set di blocchi per la programmazione del microcontrollore, nonché il mancato supporto nativo alle librerie di terze parti (incluse quelle per il controllo della matrice di led). Tali limitazioni sono dovute al fatto che le soluzioni proposte si limitano semplicemente a far dialogare Arduino e Snap!, attraverso un firmware apposito basato sullo standard *Firmata*[6], e ad eseguire il mapping a run-time dei blocchi Snap! con le procedure predefinite nel firmware. Inoltre, nonostante l'eventuale possibilità di modificare i firmware descritti a seconda delle nostre esigenze, questi, basandosi sullo standard Firmata, necessitano comunque di un collegamento continuo tra il dispositivo e il computer ausiliario sul quale è in esecuzione Snap!, limitando possibili evoluzioni e applicazioni future del caso di studio.

Alternativamente alle soluzioni basate sulla comunicazione tra Arduino e il computer, il framework *Snap4Arduino* consenta, in aggiunta, una vera e propria conversione degli script Snap! in semplici sketch per Arduino. Ciononostante, anche tale soluzione risulta presentare notevoli limitazioni. Il problema principale di questa alternativa sta nel fatto che, ad oggi, impone una notevole restrizione circa la tipologia e la struttura dei progetti Snap! che possono essere convertiti, limitando fortemente il dominio applicativo per il quale utilizzarla. Infatti: [7]

1. È possibile esportare esclusivamente singoli script, escludendo quindi

[5] s2a_fm (https://github.com/MrYsLab/s2a_fm) e Snap4Arduino (http://snap4arduino.rocks).

[6] http://firmata.org/wiki/Main_Page

[7] http://blog.s4a.cat/2015/06/09/Snap4Arduino-Arduino-sketch-generation.html

a priori la possibilità di modellare sistemi concorrenti composti da più script e Sprite.

2. Ogni script che si vuole esportare in Arduino necessità del blocco *"quando si clicca su [bandierina verde]"* come primo blocco, escludendo ulteriormente la modellazione di sistemi reattivi agli eventi (ossia con script che hanno, ad esempio, come primo blocco *"quando<condizione>"*).

Vista la non esistenza di un framework in grado di soddisfare appieno le esigenze attuali e future del progetto Smart T-Shirt, ci si pone davanti una questione cruciale: Come proseguire? In risposta a questa, le strade possibili sono essenzialmente le tre di seguito riportate:

- Accontentarsi delle soluzioni, seppur incomplete e limitate, precedentemente trattate.
- Cambiare la piattaforma software (Snap!) e/o quella hardware (Arduino).
- Cercare un modo alternativo per consentire a Snap! di essere eseguito sulla piattaforma Arduino, senza la necessità di un dispositivo ausiliario col quale dover costantemente dialogare.

Relativamente alla prima, si andrebbe in contro a limitazioni non accettabili e che minerebbero troppo, come già detto, possibili evoluzioni future del progetto e le applicazioni associate. Con la seconda opzione, nel caso si sostituisse Snap!, si andrebbe ad interrompere la continuità che da oltre due anni e mezzo vi è nell'utilizzo di questo all'interno del progetto COGITO, mentre nel caso si sostituisse Arduino (p.e. con Raspberry Pi), verrebbero meno tutti i vantaggi da esso derivanti e in precedenza descritti. La terza opzione, quella scelta, risulta invece essere quella con le maggiori potenzialità, tanto da poter trovare significativa utilità, in generale, in un contesto di utilizzo internazionale. In particolare, la realizzazione di un simile framework, nato dalla commistione di una delle piattaforme più utilizzate e famose in ambito embedded/IoT e Making, quale Arduino, e un linguaggio tanto semplice e immediato, quanto potente, come Snap!, potrebbe trovare impiego nel supportare qualsiasi progetto nel quale si sfrutti la programmazione a blocchi in ambito di sistemi embedded/IoT, in particolare nel mondo making e degli atelier digitali/creativi [8], nonché all'interno del contesto di pensiero computazionale e coding nelle scuole, ma non solo.

[8] `http://www.istruzione.it/scuola_digitale/prog-atelier.shtml`text

Ciò che la terza opzione richiede, al fine di consentire, nel modo più efficace possibile, l'esecuzione di codice Snap! sulla piattaforma Arduino (e in generale Wiring), è la realizzazione di un apposito *transpiler*[9] in grado di convertire i progetti realizzati in Snap! (a partire dal *Micromondo della Matrice*) in progetti compatibili per Arduino e Wiring, necessità a partire dalla quale nasce il progetto *Snap2ino* (cap. 4).

Usabilità

Per quanto riguarda, infine, il secondo aspetto da gestire in fase di progettazione, ossia l'usabilità, questa è da vedere principalmente, come già detto, come la commistione di comodità, mobilità e sicurezza di utilizzo della Smart T-Shirt.

Per quanto riguarda gli aspetti di comodità e mobilità, la scelta di un dispositivo compatto come Arduino consente un certo grado di integrazione nell'outfit di un bambino, garantendo un buon livello di comodità e mobilità. Tale integrazione può avvenire, ad esempio, attraverso la realizzazione di una custodia appositamente progettata a contenente sia la matrice sia il dispositivo controllore, appendibile al collo del bambino attraverso un laccio opportuno. I vantaggi di questa soluzione, rispetto alla realizzazione di una maglietta apposita (nella quale la matrice sarebbe cucita sopra), sono diversi. Da un lato vi è la possibilità di rendere "smart" non solo qualsiasi indumento, senza la necessità di dover indossare per forza la maglietta con la matrice (per la quale sarebbe, inoltre, sconsigliato il lavaggio, onde evitare eventuali danni alla matrice o ai collegamenti), ma, più in generale, qualsiasi oggetto sul quale poter appendere la custodia (porta della camera da letto, armadio, etc). Inoltre, attraverso tale soluzione, il bambino è libero di muoversi senza doversi preoccupare di scucire la matrice dalla maglietta o di scollegare i cavi collegati al controllore (che, onde evitare di essere scomodo sul petto del bambino, andrebbe tenuto, ad esempio, in tasca).

Per quanto concerne, invece, l'aspetto della sicurezza, la soluzione della custodia, efficace nel garantire comodità e mobilità al bambino, risulta vincente anche in questo frangente. Infatti, racchiudendo tutti i componenti elettronici all'interno di un'unico contenitore (ignifugo), si garantisce il massimo livello di isolamento e sicurezza, evitando che cavi (più o meno liberi) nel corpo della maglietta, e in generale il controllore, entrino in contatto col tessuto della maglietta.

[9] **transpiler:** programma in grado di convertire sorgenti scritti in un dato linguaggio, in sorgenti di un linguaggio differente.

3.4 Il mio contributo

Vista la non esistenza di un framework che soddisfi appieno le esigenze attuali e future del progetto, il mio ruolo all'interno del team Smart T-Shirt è quello di realizzare il *transpiler* richiesto dalla fase di progettazione. Questo, oltre a dover generare progetti Wiring a partire da sorgenti Snap!, deve consentire, in particolare, l'estensione dei blocchi supportati, attraverso la possibilità di mappare un qualsiasi blocco custom realizzato in Snap! con la relativa implementazione C/C++ eseguibile in Wiring, nonché pieno supporto a sistemi concorrenti, composti da più script/Sprite, e agli eventi.

In particolare, rispetto ai progetti esistenti e precedentemente citati, si sottolineano le seguenti caratteristiche peculiari:

- Possibilità di definire i propri blocchi (lato Snap!) e di mappare questi con le effettive implementazioni per il dispositivo di destinazione (basato su Wiring), consentendo la definizione del proprio "linguaggio" e del proprio "livello di astrazione", al di là dei blocchi predefiniti che il framework stesso può dare.

- Possibilità di utilizzare librerie e codice Wiring esistenti, e i relativi sensori e attuatori, rendendo il sistema quanto più possibile aperto.

Tali caratteristiche, e le potenzialità che da esse derivano, fanno di questo progetto, come già detto, un vero e proprio framework aperto e adatto all'utilizzo nei più svariati contesti (making, atelier digitali/creativi, coding e pensiero computazionale), rendendolo, quindi, potenzialmente significativo per l'intera comunità internazionale in generale.

3.5 Sommario

In questo primo capitolo progettuale è stato introdotto e descritto, partendo dall'idea, il ruolo all'interno del progetto COGITO, le funzionalità e gli obiettivi perseguiti, il progetto Smart T-Shirt. In particolare, per quanto riguarda l'aspetto progettuale relativo alla programmazione delle magliette, è stato accennato il progetto *Snap2ino* (cap. 4), ovvero il transpiler necessario a mappare i micromondi realizzati in Snap! in generici sketch per il framework Wiring (con riferimento al microcontrollore Arduino), rappresentante il contributo progettuale insito in questa tesi e che verrà trattato nel prosieguo.

Capitolo 4

Da Snap! ad Arduino con Snap2ino

Nel precedente capitolo è stato introdotto il progetto Smart T-Shirt e discussa la progettazione relativamente all'aspetto programmatico e di usabilità di questo. Per quanto riguarda il primo aspetto, ossia la possibilità di programmare la maglietta, si è deciso di utilizzare la piattaforma hardware Arduino (scheda Uno), per il controllo della matrice, e il linguaggio Snap!, per la definizione del comportamento dei led. In particolare, vista la mancanza di un framework in grado di venire in contro alle esigenze presenti e future del caso di studio, è emersa la necessità di realizzare un apposito *transpiler* in grado di convertire i progetti realizzati in Snap!, con riferimento al *Micromondo della Matrice*, in sketch eseguibili su Arduino Uno, con riferimento al framework Wiring.

In questo capitolo, viene quindi trattato il progetto del suddetto transpiler, *Snap2ino*, partendo da una descrizione dell'approccio allo sviluppo (4.1), passando poi all'analisi (4.2), alla progettazione (4.3), con riguardo circa la scelta delle tecnologie utilizzate (4.3.1), e infine al testing (4.4) e a una breve guida all'uso, corredata dalle funzionalità, dalle limitazioni e dai vincoli di quanto realizzato (4.5). In particolare, questo capitolo si concentra sul primo prototipo, progettato per un determinato contesto di utilizzo, a partire dal quale verrà poi progettato e implementato il secondo prototipo, più generico e versatile, trattato nel capitolo successivo.

4.1 Descrizione

Come in precedenza accennato, il progetto *Snap2ino* ha lo scopo di realizzare un *transpiler* in grado di convertire il comportamento definito per

mezzo del linguaggio Snap!, con riferimento al *Micromondo della Matrice* (1.4.2), in uno equivalente che sia eseguibile dalla scheda Arduino Uno (con riferimento al framework Wiring), ossia in sketch in linguaggio C/C++. Come per il progetto Smart T-Shirt, anche per Snap2ino si segue un processo di sviluppo incrementale e basato sulla prototipazione evolutiva. In particolare, data la versatilità di Snap!, dovuta alla possibilità di definire i propri *Micromondi*[1], e la complessità che questi possono avere, attraverso aspetti avanzati come la *gestione degli eventi* e l'*esecuzione concorrente* di script e Sprite, la realizzazione del transpiler prevede lo sviluppo di due prototipi.

Il primo, trattato in questo capitolo, è un prototipo di base progettato per rispondere ad esigenze specifiche, come la conversione di un singolo script realizzato nel contesto del *Micromondo della Matrice* e attraverso i blocchi (base e alcuni custom) predefiniti.

Il secondo, trattato invece nel capitolo 5, ha come obiettivo principale, da una parte, l'integrazione di funzionalità in grado di rendere il transpiler realizzato (il primo prototipo) più generico possibile, come il supporto alla conversione di un qualsiasi *Micromondo* definito (composto da un numero qualsiasi di script e Sprite e dotato di propri blocchi base e custom), dall'altra, il supporto ad aspetti avanzati come l'esecuzione di script e Sprite in maniera concorrente e la modellazione di sistemi ad eventi. Questi ultimi due aspetti, in particolare, caratterizzano la parte più critica dell'intero progetto Snap2ino, fin'ora realizzato. Essi, infatti, richiedono che sulla scheda Arduino Uno, sprovvista di sistema operativo, sia possibile eseguire più task concorrentemente, garantendo, inoltre, una *schedulazione* quanto più possibile fedele a quella realizzata da Snap!, in modo che il comportamento riscontrato sulla matrice fisica e quanto ottenuto nell'anteprima dell'ambiente Snap! sia il medesimo.

4.2 Analisi

Il requisito principale che ogni transpiler deve soddisfare è, in generale, quello di poter convertire i sorgenti scritti in un dato linguaggio di provenienza A, in sorgenti in un certo linguaggio di destinazione B. Per quanto

[1] **Micromondo:** a partire dalla definizione data da Seymour Papert nel contesto al linguaggio LOGO (1.3.1, terzo capoverso, riga 5), si intende un ambiente di apprendimento digitale appositamente realizzato per esplorare e apprendere una data disciplina e materia. In particolare, nell'ambito del progetto COGITO (1.4.2, Secondo anno, punto 2), si intende un generico progetto Snap!, composto da: uno scenario (lo Stage), uno o più attori (gli Sprite), uno o più script per la definizione del comportamento dei vari attori e un insieme di blocchi (base e/o *custom*) che definiscono il *linguaggio* utilizzabile in tale micromondo.

concerne il caso di studio Smart T-Shirt, tuttavia, il transpiler da realizzare non dovrà limitarsi alla mera conversione dei sorgenti da A a B (da Snap! a C/C++), ma consentire che, una volta convertiti, questi siano eseguiti nel rispetto del *modello d'esecuzione* originario, ossia quello dell'ambiente Snap!, in modo tale da ottenere, su Arduino, il medesimo comportamento sperimentato sul pc. Ciò significa dover considerare come convertire non solo gli aspetti *sintattici* dei sorgenti Snap!, nelle rispettive controparti Wiring (C/C++), ma anche il *modello d'esecuzione* utilizzato per eseguirli, ossia gli aspetti *semantici* del linguaggio.

Segue quindi la descrizione dell'organizzazione dei progetti in Snap! e del relativo *modello d'esecuzione* generali, a partire dai quali definire, poi, quelli specifici per il *Micromondo della Matrice*. Grazie a questi ultimi saranno in seguito condotte l'analisi sintattica del linguaggio, focalizzata esclusivamente sui costrutti rappresentanti gli elementi significativi per la conversione, e l'analisi dell'output da generare, che andrà a contenere i costrutti del linguaggio Snap! e il *modello d'esecuzione* di questo, mappati con le rispettive controparti in Wiring.

4.2.1 Organizzazione dei progetti in Snap! e modello d'esecuzione

Partendo dall'organizzazione generale di un progetto Snap!, questo è caratterizzato da una o più Sprite (attori) contenenti al proprio interno la definizione delle variabili locali (allo Sprite) e/o globali (condivise tra tutti gli Sprite) e di uno o più script attraverso i quali definire il comportamento di questi. Gli script, a loro volta composti a partire dai blocchi messi a disposizione per quel determinato progetto, vengono eseguiti concorrentemente dall'ambiente, consentendo agli Sprite, per i quali sono stati definiti, di comportarsi in maniera autonoma rispetto agli altri. Il modello d'esecuzione utilizzato per gli script, in particolare, è quello ad eventi, caratterizzato dall'esecuzione di ciascuno di questi attraverso il semplice "click" del mouse sopra di essi, piuttosto che in risposta a particolari condizioni specificate per mezzo del blocco posto loro come testata (p.e. i blocchi *"quando si clicca su [bandierina verde]"* o *"quando<condizione>"*). Lo scenario d'uso generale per un progetto, invece, è quello di mandare in esecuzione gli Sprite attraverso la pressione del pulsante *"bandierina verde"*, scatenante l'esecuzione concorrente di tutti gli script con primo blocco *"quando si clicca su [bandierina verde]"*, ai quali segue l'esecuzione di quelli dipendenti da particolari eventi o condizioni (p.e. con blocco testata *"quando<condizione>"*). In particolare, mentre i primi vengono eseguiti una volta soltanto (ossia a

seguita dell'evento "click" sulla *"bandierina verde"*) i secondi sono eseguiti ogniqualvolta la condizione che specificano è verificata, se questi non sono già in esecuzione.

Al di là delle caratteristiche e dello scenario generali appena descritti, analizzati in dettaglio solo nel secondo prototipo (5.2.2), ora, per quanto concerne il caso specifico del *Micromondo della Matrice*, si suppone che il progetto sia composto da una singola Sprite, un solo script e le sole variabili locali. Per quanto riguarda, invece, il *modello d'esecuzione*, questo consisterà nell'eseguire l'unico script presente, una sola volta, a seguito della pressione del pulsante *"bandierina verde"*. Inoltre, escludendo per questo primo prototipo gli scenari multi-script/Sprite e di esecuzione condizionata/a eventi degli script, è possibile ignorare i blocchi *"esegui in modalità turbo"*, utilizzato in contesti multi-script/Sprite, e *"quando<condizione>"* tra quelli da convertire (fig. 1.5).

4.2.2 Analisi sintattica del *Micromondo della Matrice*

Una volta definita l'organizzazione specifica del progetto Snap! *Micromondo della Matrice*, è ora possibile passare all'analisi del sorgente da esso derivante (rappresentante l'input da convertire per il tanspiler), attraverso la possibilità offerta dall'ambiente Snap! di esportare i progetti realizzati, come singoli file in formato `.xml`. A partire da questi, e quindi dalla *sintassi xml*, è possibile analizzare e individuare quali informazioni, relativamente al progetto esportato, siano contenute e come queste siano organizzate tra loro. In particolare, tra le molte informazioni contenute, è possibile individuare ciò che effettivamente ha significato per la conversione, ossia: gli Sprite definiti, le relative variabili locali e gli script in esso contenuti, e le variabili globali. Nel caso specifico del *Micromondo della Matrice* (e dell'esercizio presentato - fig. 1.4), ossia di un progetto Snap! composto da una sola Sprite, un solo script e le sole variabili locali, è possibile individuare, rimuovendo le informazioni superflue per il caso di studio, una prima porzione di codice xml, rappresentante proprio queste ultime, in corrispondenza del sotto-tag `/variables/` e una seconda, rappresentante lo script definito, in corrispondenza del sotto-tag `/scritps/script/`, entrambe contenute nello Sprite collocato in `project/stage/sprites/sprite/`:

File: Micromondo Matrice - Quinta.xml

```
<!-- VARAIBILI LOCALI DEFINITE E VALORE ATTUALE -->
<variables>
```

```
    <variable name="x">
        <l>2</l>
    </variable>
    <variable name="y">
        <l>7</l>
    </variable>
</variables>

<!-- CORPO DELLO SCRIPT -->
<script>
    <!-- BLOCCHI SENZA ARGOMENTI -->
    <block s="receiveGo"></block>
    <custom-block s="prepara matrice"></custom-block>

    <!-- BLOCCHI CON ARGOMENTI -->
    <block s="doSetVar">  <!-- assegnamento di variabile -->
        <l>x</l>  <!-- variabile da valorizzare -->
        <l>0</l>  <!-- valore -->
    </block>
    <block s="doSetVar">
        <l>y</l>
        <l>0</l>
    </block>

    <!-- CICLO: RIPETI 8 VOLTE -->
    <block s="doRepeat">
          <l>8</l>
          <script>
              <block s="doRepeat">
                  <l>8</l>
                   <script>
                       <custom-block s="accendi led x: %n y: %n
                                  (colore r: %n g: %n b: %n )">
                          <block var="x"></block>
                          <block var="y"></block>
                          <l>0</l>
                          <l>255</l>
                          <l>0</l>
                        </custom-block>

                        <!-- assegnamento ad x di x+1 -->
                        <block s="doSetVar">
                            <l>x</l>
                            <block s="reportSum">
                                <!-- accesso a variabile -->
                                <block var="x"></block>
                                <l>1</l>
                            </block>
                        </block>
```

```
                </script>
            </block>
            <block s="doSetVar">
                <l>x</l>
                <l>0</l>
            </block>
            <block s="doSetVar">
                <l>y</l>
                <block s="reportSum">
                    <block var="y"></block>
                    <l>1</l>
                </block>
            </block>
        </script>
    </block>
</script>
```

Com'è possibile notare dal listato presentato, la prima sezione contiene le variabili locali definite, racchiuse dal tag `<variables>`, caratterizzate da un nome (l'attributo `name`) e dal valore al momento dell'esportazione (il sotto-tag `<l>`). La seconda sezione invece, costituente il corpo dello script realizzato in Snap! (racchiuso dal tag `<script>`), si compone di una serie di blocchi base (`<block>`) e di blochi definiti dall'utente (`<custom-block>`), caratterizzati da un nome (l'attributo `s`) ed, eventualmente, uno o più argomenti in input (definiti come sotto-tag). Alternativamente al proprio nome, i blocchi base possono specificare il nome di una variabile definita, attraverso l'attributo `var`, per rappresentarne l'utilizzo come argomento di un altro blocco.

Partendo dall'analisi del listato proposto, e considerando ora tutti blocchi utilizzabili nel *Micromondo della Matrice* (fig. 1.5), è possibile derivare lo schema UML (fig. 4.1) rappresentante l'organizzazione generale che i tag xml significativi possono avere, circa le due sezioni di codice appena individuate, rappresentante la *sintassi* per il sorgente (filtrato) da convertire. In tale schema, si evidenzia la presenza di alcuni *stereotipi*, usati per meglio definire l'organizzazione sintattica degli elementi sotto esame:

- *«xml tag»*: identifica una classe rappresentante un tag xml;
- *«inner content»*: identifica un'associazione di composizione, legante un tag *genitore* agli eventuali tag *figli* (*subtag*);
- *«inner value»*: identifica il contenuto *testuale* (quindi non un tag) contenuto all'interno di un tag.

Per l'UML definito, si precisa come le istanze di *Literal* (ossia i tag nella forma `<l>valore-alfanumerico<l>` oppure `<l><bool>true-false</bool></l>`),

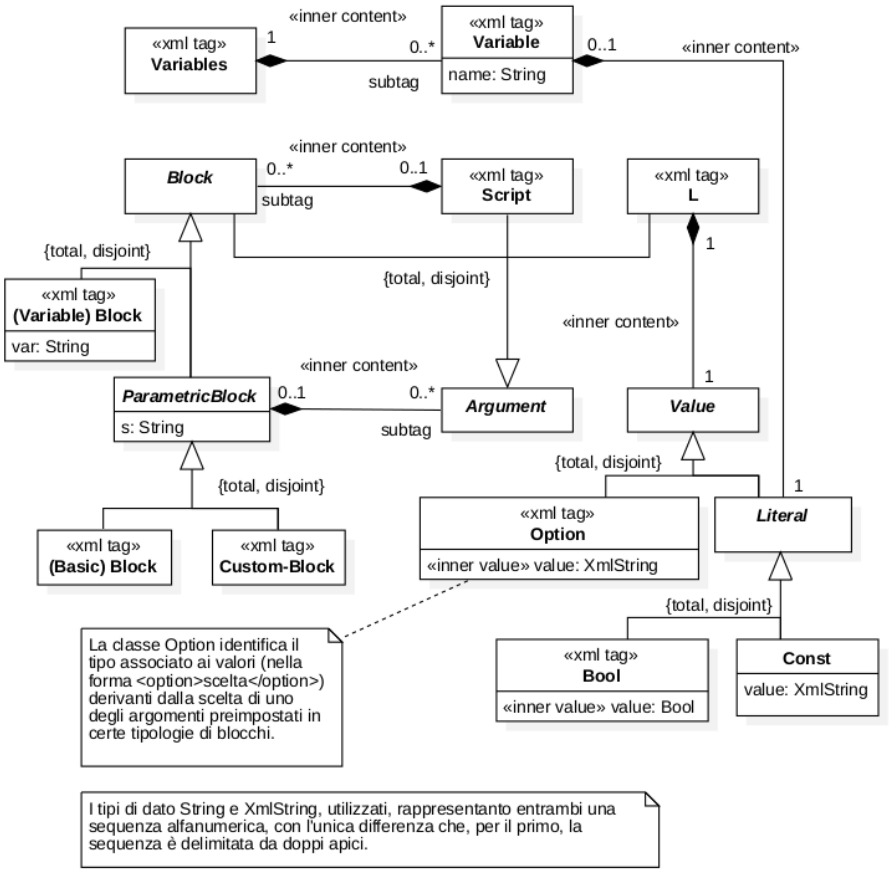


Figura 4.1: Organizzazione sintattica dei tag xml (significativi) per un generico progetto/*Micromondo* Snap! esportato.

così come quelle di *Block*, siano associate, rispettivamente, a Variable e Script opzionalmente, poiché rappresentanti, esclusivamente, o i valori assunti dalle variabili, le prime, e i blocchi contenuti in uno script, le seconde, o gli argomenti in input di un blocco.

Oltre a questo, è anche importante notare che, nell'UML definito, non sono specificati vincoli circa il tipo (e il numero) degli argomenti accettati da un dato blocco e, in generale, non sia tenuto conto della compatibilità tra il tipo di un blocco contenuto (ossia un *subtag*) e il relativo blocco genitore (p.e. è ammesso che un blocco *ripeti sempre* sia argomento di un blocco che accetta solo valori interi). Questo perché si presuppone che l'input fornito corrisponda a un micromondo già validato dall'ambiente Snap! e quindi corretto. Da ciò ne consegue, quindi, che il transpiler da realizzare non deve effettuare alcun controllo circa la correttezza (*semantica*) del sorgente

passato.

In definitiva, quindi, a seguito dell'analisi sintattica effettuata è possibile stabilire quanto segue.
Il sorgente in input da processare consiste in un file in formato xml, rappresentante il Micromondo della Matrice validato e corretto. Questo, tolte le informazioni superflue per la conversione, si compone di due sezioni, rispettanti la sintassi definita dallo schema della figura 4.1. La prima rappresenta le variabili locali definite e il valore di queste al momento dell'esportazione, la seconda lo script eseguibile, composto da una serie di blocchi Snap! di cui il primo è sempre "quando si clicca su [bandierina verde]".

4.2.3 Analisi dell'output da generare: *Lo sketch Wiring*

Infine, per quanto riguarda l'output da generare, al di là del linguaggio di destinazione (il C/C++), per il quale è nota la *sintassi*, ciò che effettivamente occorre definire è come organizzare i/il sorgenti/e che, una volta compilati/o, devono/e essere eseguiti/o sulla scheda Uno, nonché il relativo *modello d'esecuzione* utilizzato per questi. In particolare, dal momento che gli sketch di Arduino si basano su Wiring, ciò che occorre conoscere è la relativa *organizzazione e strutturazione dei sorgenti* richiesta da tale framework, e come questo esegua il codice in essi contenuto.

Partendo dall'organizzazione generale dei sorgenti in Wiring, questa è la medesima di in un generico progetto C/C++, ossia caratterizzata da un file principale e uno o più file contenenti eventuali variabili esterne e le definizioni (`.c/cpp`) e i prototipi (*header file* `.h/hpp`) delle procedure/funzioni richiamate da questo. Per quanto concerne, invece, la *strutturazione dei sorgenti*, ossia l'organizzazione del codice in essi contenuto, mentre quella per i file contenenti le procedure/funzioni e gli *header file* è la medesima dei progetti C/C++ tradizionali, quella del sorgente principale è invece la seguente:

```
File: expected_output.ino

#include<header_1.h>
#include<header_2.h>
// ...
#include<header_N.h>

// defizione delle variabili globali:
int i;

// procedura chiamata automaticamente (solo una volta),
// prima della procedura loop(), per settare variabili,
```

```
// instaziare oggetti, etc.
void setup() { // body }

// procedura chiamata automaticamente (ripetutamente),
// contenente le operazioni da eseguire.
void loop() { // body }
```

In tale listato è possibile notare, analogamente a come si farebbe per qualsiasi progetto C/C++, l'inclusione dei vari *header file* e la parte di definizione delle variabili globali. Ciò che tuttavia differisce da un classico `main.c/.cpp` è la presenza delle due procedure `setup()` e `loop()` qui definite. La prima, eseguita solo una volta e prima di `loop()`, ha lo scopo di settare i valori iniziali delle variabili globali, piuttosto che istanziare eventuali oggetti C++, mentre la seconda, eseguita ripetutamente, contiene invece tutte le istruzioni che si vuole far eseguire dal microcontrollore.

L'aspetto fondamentale da sottolineare circa queste due procedure è come siano caratteristiche del tipo di *modello d'esecuzione* utilizzato per i progetti basati su Wiring, ossia il modello a *loop di controllo*, anche detto *super loop*. Questo, basato sull'esecuzione continua e ripetuta delle istruzioni contenute all'interno di un'apposita procedura (`loop()` in questo caso), è in genere utilizzato, nei contesti embedded, per la modellazione del cosiddetto *polling*, ossia quel comportamento di interrogazione ripetuta e continua circa la presenza di eventi o condizioni particolari, in risposta ai quali reagire.

Considerando ora l'organizzazione del progetto e il *modello d'esecuzione* definiti per il *Micromondo della Matrice* (4.2.1), è possibile notare come questi possano essere facilmente mappati (*mapping semantico*) nelle rispettive controparti definite per Wiring, attraverso il solo sorgente `.ino`. Infatti, per quanto riguarda lo script, dovendo questo essere eseguito una sola volta, è possibile inserire il codice derivante dalla conversione di quest'ultimo all'interno della procedura `loop()`, vincolando l'esecuzione di questo a una singola volta. Per quanto riguarda, invece, l'evento scatenante l'esecuzione di questo, ossia il "click" sulla *"bandierina verde"* dell'ambiente Snap!, questo è facilmente mappabile, concettualmente, con la pressione del pulsante di avvio/reset di Arduino. Infine, per quanto concerne le variabili locali definite in Snap!, dovendo queste essere accessibili allo script, e quindi alla procedura `loop()`, è possibile rappresentare queste attraverso semplici variabili globali nel sorgente Wiring principale, definite e inizializzate a seconda del tipo del valore assunto al momento dell'esportazione. A tal proposito, l'unica restrizione rispetto a Snap! è che il tipo delle variabili così convertire sia immutabile, a differenza di quanto può invece avvenire in quest'ultimo.

Alla luce di quanto emerso dall'analisi appena effettuata, è dunque possibile definire quanto segue.
L'output da generare consiste nel solo sorgente Wiring principale (.ino). Per ogni variabile locale definita nel Micromondo della Matrice, il sorgente deve contenere una variabile globale che la rappresenta, definita con tipo concorde a quello desumibile dal valore specificato nel file xml (presupponendo che questo non cambi mai) e inizializzata con un valore standard*, scelto per il tipo di dato desunto. Infine, il comportamento dello script da convertire deve essere posizionato all'interno della procedura loop(), facendo in modo che questo sia eseguito una sola volta.*

4.2.4 Definizione dei requisiti

Volendo riassumere quanto emerso dalla fase di analisi, è quindi possibile definire, come quadro generale circa i requisiti richiesti per il primo prototipo, quanto segue. *Il transpiler Snap2ino deve essere in grado di prendere come sorgente, in input, il file xml rappresentante il progetto Snap! "Micromondo della Matrice", validato e corretto. Questo deve poi essere filtrato in modo da mantenere solo le informazioni di interesse da convertire, ossia la sezione rappresentante lo script definito e quella delle relative variabili locali. Infine, deve convertire tali informazioni generando, secondo la sintassi del linguaggio C/C++, lo sketch Wiring, rappresentato dal solo file .ino, contenente la definizione delle variabili globali rappresentanti quelle locali definite in Snap!, e il comportamento dello script collocato all'interno della procedura loop(), facendo in modo che sia eseguito una sola volta.*

4.3 Progettazione

Descritti l'input da processare e l'output da generare, è ora necessario stabilire come passare dal primo all'ultimo, ossia convertire le informazioni (il codice) definite attraverso la sintassi xml, in uno sketch Wiring equivalente e compilabile per la piattaforma Arduino.

Partendo dall'analisi precedentemente effettuata, i requisiti emersi da questa possono essere identificati nei seguenti punti, caratterizzanti il funzionamento del transpiler da realizzare:

1. Acquisizione del sorgente da processare e filtraggio delle informazioni (tag) significative in esso contenute (4.3.2).

2. Lettura e comprensione delle informazioni filtrate (4.3.3).

3. Conversione delle informazioni filtrate e generazione dei file richiesti, costituenti lo sketch da compilare e caricare su Arduino (4.3.4).

Di queste tre punti, il vero cuore della fase di progettazione del transpiler da realizzare risiede però negli ultimi due, ossia nel riuscire a leggere e capire le informazioni filtrate, e a convertire queste in maniera equivalente nel linguaggio di destinazione. Per poter fare ciò, si è ricorso alla definizione di un DLS (*Domain-Specific Language*) rappresentante il linguaggio composto dalle informazioni filtrate, e chiamato come il transpiler da realizzare: *Snap2ino*.

Domain-Specific Languages Col termine DSL di intende un qualsiasi linguaggio di programmazione, o di specifica, appositamente realizzato per l'utilizzo all'interno di uno specifico dominio applicativo. A differenza dei linguaggi di programmazione *general purpose* (come il C o Java), non sono realizzati con lo scopo di poter risolvere un qualsiasi genere di problema, ma una ben precisa classe e tipologia in maniera più semplice e veloce rispetto ai primi. Alcuni esempi sono SQL, Mathematica, HTML e molti altri. Una volta realizzato un programma, o specifica, in un particolare DSL, questo può poi essere interpretato o compilato, a seconda delle esigenze, secondo uno specifico linguaggio *general purpose*, piuttosto che essere manipolato e processato, nel caso rappresenti semplicemente una particolare organizzazione di dati, da particolari sistemi specifici [38, p. 7].

Detto questo, implementare un DSL significa sviluppare un programma che sia in grado di *leggere* un sorgente scritto in quel particolare linguaggio domain-specific, *farne il parsing, processarlo* e, possibilmente, *interpretarlo o convertirlo in un altro linguaggio* [38, p. 9].

Grammatica e Parsing Il punto di partenza nell'implementazione di un qualunque DSL è la definizione della *grammatica*, ossia l'insieme di regole attraverso le quali definire la *sintassi* del linguaggio. Solo attraverso la definizione della grammatica del proprio linguaggio è infatti possibile leggere un programma scritto in tale linguaggio, nonché verificarne la correttezza. In particolare, prima di poter leggere il programma, occorre innanzitutto verificare che sia effettivamente codificato nel linguaggio che ci aspettiamo, ossia che ne rispetti la *sintassi*.

Per fare questo occorre, in prima battuta, scomporre il sorgente dato in tanti piccoli *token*, ossia elementi atomici del linguaggio, come: *parole chiave* (p.e. `class` in Java), *identificatori* (come il nome della classe) o *nomi simbolici* (come il nome di una variabile), o ancora *operatori* (aritmetici,

logici, parentesi, etc) e *valori letterali* (p.e costanti numeriche, stringhe, etc). Il processo che consente di convertire generiche sequenze di caratteri in una sequenza di *token* è detta *analisi lessicale* (*lexical analysis*) ed è eseguita ad opera di un *analizzatore lessicale* (a.k.a *lexer*) attraverso opportune *Regular Expressions*[2].

Una volta ottenuta la sequenza di *token* dal file sorgente, è quindi possibile verificare se questi rispettano la *struttura sintattica* definita dalla *grammatica* del nostro linguaggio, nel processo chiamato *parsing* [38, pp. 10, 11].

Abstract Syntax Tree, Validator e Code Generator Oltre all'analisi sintattica, un'altra fondamentale attività svolta durante il *parsing* è la generazione del cosiddetto *Abstract Syntax Tree*, ossia una particolare struttura dati ad albero, utilizzata per la rappresentazione in memoria del programma *parsato*, nella quale ogni nodo rappresenta un costrutto utilizzato del linguaggio.

Il motivo principale alla base della generazione e l'utilizzo di tale struttura dati risiede nel fatto che l'analisi sintattica di un programma non è sufficiente a garantirne la piena correttezza, in particolare la correttezza *semantica*. Infatti, solo potendo accedere a una rappresentazione in memoria del programma è possibile effettuare importanti verifiche *semantiche* come: il *type checking*, ossia la verifica del corretto utilizzo dei tipi di dato (negli assegnamenti, nelle operazioni, etc), e lo *scope checking*, ossia il controllo sulla visibilità e gli accessi (di/a variabili, campi, metodi, etc), per il quale è, in genere, richiesto il supporto di un'apposita *tabella dei simboli*[3].

Una volta verificata la *correttezza semantica* del programma, ad opera del cosiddetto *validator* e attraverso l'ispezione dei nodi dell'AST, è poi possibile eseguire l'ultima importante fase operativa: *l'interpretazione/generazione del codice* [38, p. 12]. Questa, eseguita ad opera di un componente chiamato *code generator* e attraverso l'ispezione della rappresentazione in memoria del programma *parsato* e validato (AST), consiste nella generazione dei sorgenti richiesti, con riferimento all'organizzazione del codice richiesta per questi, attraverso il *mapping* dei costrutti definiti nel programma *parsato*, contenuti nell'AST, nelle controparti equivalenti per il linguaggio di destinazione.

[2]**Regular Expressions:** letteralmente *Espressioni Regolari*, sono sequenze di simboli (caratteri) utilizzate per l'identificazione di sottostringhe, stringhe o insieme di stringhe.

[3]**tabella dei simboli:** struttura dati contenente le informazioni su tutti gli elementi simbolici incontrati durante l'analisi del sorgente, come: nome, visibilità (scope), eventuale tipo, etc.

4.3.1 Tecnologie, linguaggi e architetture utilizzate

Domain-Specific Languages con Xtext

Per l'implementazione del DSL richiesto si è ricorso a *Xtext* [38], framework utilizzato per implementare linguaggi di programmazione e DSL in *Eclipse*[4]. La peculiarità di Xtext risiede nella possibilità di realizzare il proprio DSL molto velocemente, partendo dalla semplice definizione della *grammatica*, e senza la necessità di dover definire, noi stessi, le strutture dati per i nodi e le regole per la generazione dell'AST. Partendo dalla grammatica realizzata, infatti, il framework genera automaticamente le classi Java rappresentanti i nodi dell'AST, le regole per la generazione di quest'ultimo, il *lexer*, il *parser* e, addirittura, l'editor Eclipse per il linguaggio definito, lasciando allo sviluppatore la definizione del *validator* e del *code generator*.

Dei vari artefatti che Xtext è in grado di generare per noi, particolare focus va però riposto sulle classi utilizzate per i nodi dell' AST. In particolare, per quanto concerne la definizione di quest'ultimo, Xtext si basa sull'utilizzo dell'EMF (*Eclipse Modelling Framework*)[5], attraverso il quale, partendo dalla grammatica definita, inferisce un metamodello EMF (modello *Ecore*) per il linguaggio da realizzare, per poi generare, attraverso le funzionalità offerte dall'EMF, le varie interfacce e le classi Java rappresentanti i costrutti definiti per quest'ultimo, ossia il *modello* EMF per la rappresentazione dei programmi in memoria. In questo modo, Xtext consente allo sviluppatore la possibilità di ispezionare il programma in memoria (l'AST) attraverso i tradizionali costrutti e strumenti messi a disposizione dal linguaggio Java.

Infine, è importante notare come il framework, nonostante i vari artefatti autogenerati, consenta comunque un elevato grado di personalizzazione, fornendo allo sviluppatore la possibilità di modificare il processo di generazione, attraverso il file di configurazione `.mwe2` (fig. 4.2), e consentendogli di mettere mano direttamente al codice autogenerato. In particolare, per quanto riguarda quest'ultima possibilità, il meccanismo di generazione degli artefatti, onde evitare che il codice personalizzato dal programmatore venga sovrascritto da successive ri-generazioni, separa il codice autogenerato, e che verrà sovrascritto a ogni nuova generazione, da quello non rigenerabile secondo il cosiddetto *Generation Gap Pattern*. Così facendo, Xtext consente all'utente di avere consapevolezza circa quali file poter modificare, senza il rischio che vengano sovrascritti, e quali no [38, p. 17, 33-37].

[4] **Eclipse:** noto ambiente di sviluppo integrato multi-linguaggio e multi-piattaforma, liberamente distribuito sotto i termini della *Eclipse Public License*.

[5] `https://www.eclipse.org/modeling/emf`

JUnit

Durante tutta la fase sviluppo della *grammatica* del DSL *Snap2ino* e del *code generator*, fondamentale importanza è stata rivestita dall'attività di testing svolta, caratterizzata dall'utilizzo del framework *JUnit*.

Linguaggi utilizzati

- *Xtext*[6]: oltre a essere il nome del framework utilizzato per l'implementazione del DSL necessario, è anche il linguaggio (o meglio dsl) mediante il quale se ne è definita la *grammatica*.

- *Xtend*[7]: dialetto di Java consigliato dal framework Xtext, è stato utilizzato per l'implementazione del *code-generator*, delle relative classi di supporto e per la realizzazione dei test.

- *Java*: utilizzato per la realizzazione dell'*entry point* del transpiler, a partire dal quale mandare in esecuzione quest'ultimo e al quale passare il sorgente da filtrare e convertire.

Architettura e organizzazione del progetto

Per quanto concerne l'architettura e l'organizzazione del progetto utilizzate, si è fatto riferimento a quella predisposta dal framework Xtext, consistente nella suddivisione dell'intero progetto in 5 sotto-progetti. Di questi, quelli effettivamente utilizzati sono `org.xtext.example.snap2ino` e `org.xtext.example.snap2ino.tests`.

Il primo (fig. 4.2), rappresentante il progetto principale, è suddiviso in diverse directory contenenti, in particolare: i sorgenti non rigenerabili (`src`), i sorgenti Java e Xtend autogenerati (`src-gen` e `xtend-gen`), le librerie utilizzate e il metamodello EMF inferito dalla grammatica (`Snap2ino.ecore`). In particolare, per quanto riguarda i sorgenti non rigenerabili (`src`), seguono i principali package nei quali sono suddivisi:

- `org.xtext.example.snap2ino`: contenente, evidenziati, il file di configurazione per la generazione degli artefatti (`GenerateSnap2ino.mwe2`) e la definizione della grammatica (`Snap2ino.xtext`).

- `org.xtext.example.snap2ino.generator`: contenente le classi rappresentanti il transpiler vero e proprio, ossia l'*entry point* al quale passare

[6] `https://www.eclipse.org/Xtext/`
[7] `https://www.eclipse.org/xtend/`

il file da processare e che ne filtra il contenuto (`Main.java`), il *code generator* richiamato da questo (`Snap2inoGenerator.xtend`) e una classe contenente alcuni parametri di configurazione (`CodeGeneratorSettings.xtend`).

- `org.xtext.example.snap2ino.utils`: contenente una classe di utilità per il supporto all'ispezione e alla conversione di alcuni elementi del modello EMF (`ModelUtiliesTest.xtend`).

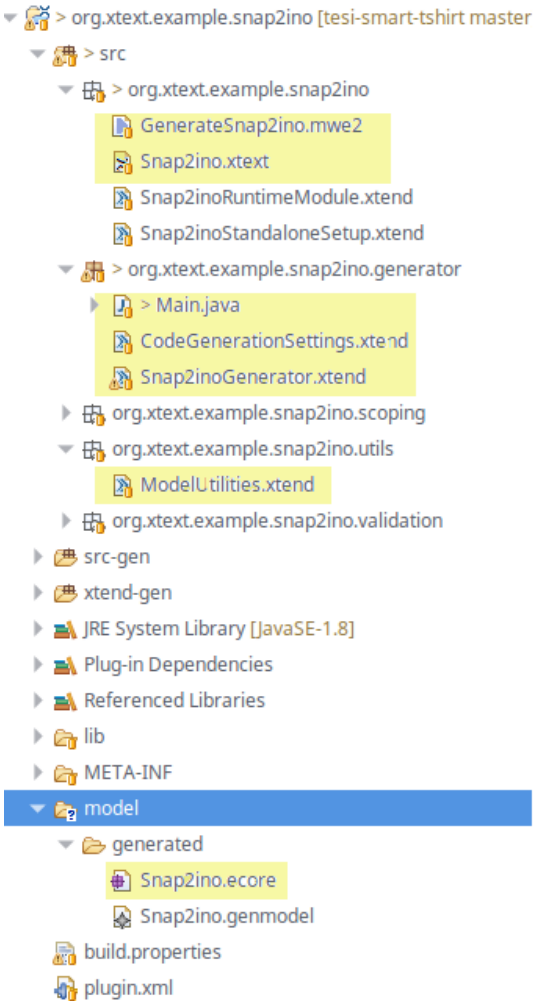


Figura 4.2: Organizzazione del progetto principale per il DSL Snap2ino.

Il secondo (fig. 4.3), con una composizione e un'organizzazione delle directory pressoché analoga al primo, è invece destinato al confinamento delle varie classi contenenti i test effettuati sugli elementi più cruciali implementati: il *code generator* (`CodeGeneratorTest.xtend`), la classe di utilità

per l'ispezione/conversione del modello EMF (`ModelUtiliesTest.xtend`) e la grammatica (`Snap2inoParsingTest.xtend`).

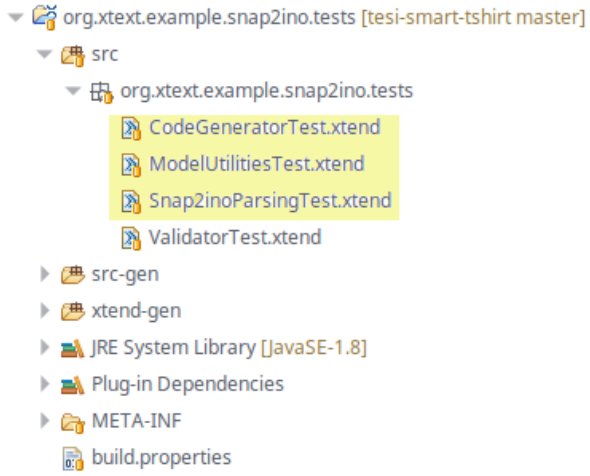


Figura 4.3: Organizzazione del progetto contenente i test per il DSL Snap2ino.

4.3.2 Acquisizione del sorgente e filtraggio delle informazioni

Per quanto riguarda i requisiti del primo punto, questi sono soddisfatti per mezzo della classe `Main.java` (fig. 4.2) definita. Questa, rappresentante l'*entry point* del transpiler (esportato come file `.jar`), consente all'utente l'utilizzo di *Snap2ino* tramite terminale (ed eventualmente l'integrazione in altri sistemi), accettando come argomento il file `.xml` rappresentante il *Micromondo della Matrice* esportato e da convertire. A partire dal file passato, una volta verificato che il formato sia quello atteso (`.xml`), vengono eliminate le informazioni (i tag xml) superflue e non previste/gestite dalla grammatica, in modo da ottenere un contenuto conforme alla sintassi del DSL atteso. Tale operazione avviene mediante l'uso delle *RegEx* (*Regular Expressions*), attraverso le quali poter rimuovere il contenuto testuale compreso: dall'inizio del documento al tag `<variables>`, dalla chiusura del tag `</variables>` alla definizione del primo script (`<script>`) e, infine, dalla chiusura di tale tag (`</script>`) alla fine del documento. Oltre a queste, vengono rimosse anche eventuali altre informazioni (attributi) superflue rimanenti a seguito della prima fase di filtraggio. Il risultato che si ottiene è un contenuto testuale costituito dalle sole sezioni descritte in precedenza (4.2.2) e conformi al DSL *Snap2ino*. A partire da tale contenuto è poi generato il relativo file in formato `.snap2ino`, utile sia per l'analisi dell'organizzazione del micro-

mondo passato, sia come punto di partenza per la successive fasi di: *lettura, comprensione, conversione e generazione dell'output.*

4.3.3 Lettura e comprensione delle informazioni filtrate

Una volta generato, a partire dal *Micromondo della Matrice* esportato, il file `.snap2ino`, conforme all'organizzazione descritta in fase di analisi dell'input (4.2.2), per poterlo convertire in uno sketch Wiring è prima necessario riuscire a leggerlo e capirlo. Come in precedenza accennato, per poter fare ciò si è ricorso alla definizione di un'apposito *Domain-Specific Language* (*Snap2ino*), per mezzo del framework Xtext, rappresentante il linguaggio composto dalle informazioni filtrate. In particolare, grazie al supporto offerto da Xtext (generazione automatica delle classi Java rappresentanti i nodi dell'AST, delle regole per la creazione di questo, del *lexer* e del *parser*) si è reso necessario definire esclusivamente la *grammatica* per questo DSL. A partire da questa, e dal *metamodello EMF* da essa inferito, è poi possibile ispezionare la rappresentazione in memoria dei sorgenti `.snap2ino` in modo da generare, nell'ultima fase, le controparti C/C++ dei i vari costrutti in essi definiti.

Definizione della grammatica

Avendo come riferimento l'*organizzazione dei tag xml* definita durante la fase di analisi dell'input (fig. 4.1), si è proceduto alla realizzazione della grammatica necessaria (fig. 4.2, `Snap2ino.xtext`), attraverso il DSL Xtext, in maniera incrementale e aggiungendo, man mano, costrutti in grado di rendere il metamodello EMF (fig. 4.2, modello `Snap2ino.ecore`), inferibile dalle regole definite, sempre più simile al modello UML per l'organizzaizone dei tag significativi realizzato durante la fase di analisi. In particolare, si è partiti dalla definizione di una semplice grammatica, composta dalla sola sezione delle variabili, di tipo intero, e dallo script, composto da blocchi base e blocchi custom senza argomenti. Successivamente è stato esteso il supporto al tipo delle variabili, definendo ConstValue (rappresentante valori costanti, ossia interi e float, con e senza segno, e stringhe - considerandoli come sequenze alfanumeriche XmlString) e BoolValue, e integrato il supporto agli argomenti in input per i blocchi (partendo dagli appena definiti ConstValue e BoolValue, includendo poi anche valori Option, altri blocchi e, infine, gli Script).

Una volta raggiunto il grado di similarità cercato, tra il metamodello EMF inferito e il modello dei tag significativi, la *grammatica completa* per il primo prototipo risulta essere la seguente:

```
XMLModel:
    variables = Variables
    script = Script
;

Variables:
    {Variables} '<variables>'
                    variables += Variable*
                '</variables>'
;

Variable:
     '<variable' 'name=' name = STRING '>'
        innerContent = Literal
     '</variable>'
;

Script:
     {Script} '<script>'
                 blocks += Block*
             '</script>'
;

Block:
    VariableBlock | BasicBlock | CustomBlock
;

VariableBlock:
     '<block' 'var=' variable = STRING '>''</block>'
;

BasicBlock:
    '<block' 's=' name = STRING '>'
        args += (Argument)*
    '</block>'
;

CustomBlock:
    '<custom-block' 's=' name = STRING '>'
        args += (Argument)*
    '</custom-block>'
;

Argument:
     Script | Block | Option | Literal
;

Option:
     '<l>'
```

```
        '<option>' value = XmlString'</option>'
     '</l>'
;

Literal:
    BoolValue | ConstValue
;

BoolValue:
    {BoolValue} '<l>'?
                    ((value ?= '<bool>true</bool>') |
                              '<bool>false</bool>')
                '</l>'?
;

// Includes strings (i.e. chars sequence without double quotes)
// and signed integers/floating point numbers.
ConstValue:
    '<l>' value = XmlString '</l>'
;

XmlString:
    ID XmlString?
;

/*
* Custom ID defined to include both signed integer and signed
* floating values.
* NB: < and >  symbols are excluded due to avoid ambiguity
* with the xml syntax.
*/
@Override
terminal ID:
    ( '\\' . | !('\\'|'"'|'\''|' '|'\t'|'\r'|'\n'|'<'|'>') )+
;
```

Le due principali differenze evidenziabili rispetto al modello definito in fase di analisi riguardano essenzialmente:

- l'inclusione del tag `<l></l>` direttamente nel costrutto Option e Literal, eliminando quindi le classi L e *Value*;
- il collasso della classe *ParametricBlock* nei rispettivi BasicBlock e CustomBlock.

Inoltre si evidenzia come, analogamente al modello definito per i tag xml (fig. 4.1), anche la grammatica realizzata non prenda in considerazione gli specifici blocchi forniti dal linguaggio Snap!, o il numero/tipo degli argomenti che questi possono, o meno, richiedere. Infatti, si è voluto rendere la

grammatica la più generica possibile per avere, da una parte, una definizione della stessa più semplice e concisa, dall'altra, una maggiore flessibilità circa la gestione dei possibili blocchi utilizzabili nel micromondo da processare. In più, è bene notare come la definizione di regole specifiche per i singoli blocchi possibili sia pressoché inutile, dal momento che si presuppone che il sorgente da processare non sia realizzato a mano ma derivante direttamente dall'esportazione del *Micromondo della Matrice* validato dall'ambiente Snap!, e quindi privo di errori.

4.3.4 Conversione delle informazioni filtrate e generazione dell'output

Una volta filtrate le informazioni di interesse e generato il relativo file `.snap2ino` per contenerle, ad opera dell'*entry point* del transpiler (la classe `Main.java`) nella prima fase, e una volta definita la *grammatica* attraverso la quale poter leggere e comprendere il DSL (*Snap2ino*) contenuto in tale file, e quindi ottenerne la rappresentazione in memoria (AST) per l'ispezione, è possibile passare all'ultima fase operativa, ossia la *conversione e generazione dell'output*.

Come già accennato più volte, per la realizzazione del proprio DSL (inteso come programma attraverso il quale poter *leggere, parsare, processare e interpretare/convertire* un sorgente scritto nel proprio linguaggio *domain-specific*), il framework Xtext richiede, oltre alla definizione della grammatica, solo l'implementazione dei propri *validator* e *code generator*. Tralasciando il primo, non necessario per tale prototipo dal momento che si presuppone che l'input filtrato, e precedentemente validato ed esportato direttamente dall'ambiente Snap!, sia *semanticamente corretto*, il *code generator*, incaricato della conversione e generazione dell'output (lo sketch Wiring), è il componente richiesto per quest'ultima fase operativa, ed è rappresentato dalla classe `Snap2inoGenerator.xtend`.

Snap2ino Generator

Basato sull'implementazione di default fornita da Xtext, nel linguaggio Xtend, il comportamento del generatore di codice consiste, essenzialmente, nelle seguenti attività:

- ispezione del modello EMF (i.e. dell' AST), creato dal framework a seguito del *parsing* al sorgente `.snap2ino`, a sua volta, generato e passato a Xtext dalla classe `Main.java`;

- mapping dei costrutti presenti nel modello EMF ispezionato con le controparti definite per il linguaggio C/C++, rispettando la *sintassi* di quest'ultimo;

- generazione dei sorgenti (lo sketch `.ino`) contenenti i costrutti mappati, rappresentanti il *Micromondo* convertito, secondo quanto definito in sede di analisi (4.2.3).

In particolare, il *code generator* deve, innanzitutto, definire il contenuto delle varie sezioni individuate in sede di analisi dell'output (4.2.3), ossia quella degli *header* (o meglio dell'inclusione degli *header file*), delle *variabili globali*, e delle procedure `setup()` e `loop()`, e successivamente generare, a partire da questo, lo sketch rappresentante il *Micromondo della Matrice* convertito, con medesimo nome (ma senza spazi) ed estensione `.ino`, e un apposito file `config.h`, contenente i parametri di default utilizzati per la configurazione della matrice e dei pin di Arduino.

Per quanto concerne la definizione del contenuto di suddette sezioni, nello specifico, a parte gli *header*, la procedura `setup()` e alcune *variabili globali di supporto*, per cui la definizione è predefinita, ossia indipendente dal contenuto effettivo del sorgente da convertire, per le altre variabili globali (rappresentanti quelle locali definite nel *Micromondo*) e il corpo della procedura `loop()` è necessaria, invece, l'ispezione del modello EMF passato, in modo da convertire i costrutti in esso contenuti nelle rispettive controparti C/C++. A tal proposito, accedendo al campo `xmlModel.variables.variables` del modello EMF, sarà infatti possibile definire il contenuto per la sezione delle variabili globali (non di supporto), mentre accedendo ai blocchi componenti lo script, ossia il campo `xmlModel.script.blocks`, sarà invece possibile definire il corpo della procedura `loop()`.

Definizione degli *header* della procedura *setup()* e delle *variabili di supporto* La prima sezione da definire è quella degli *header*. Questa, indipendente dal contenuto dell'input fornito, consiste nell'inclusione (`#include <...>`) delle librerie necessarie per il funzionamento della matrice (Adafruit_GFX, Adafruit_NeoMatrix e Adafruit_NeoPixel) e del file `config.h`. Quest'ultimo, in particolare, contiene i parametri utilizzati, all'interno della procedura `setup()`, per la configurazione di default della matrice e dei pin di Arduino. Oltre a questi, sempre dentro alla procedura `setup()`, sono inoltre inizializzati tutti gli eventuali altri oggetti utilizzati dalle controparti C/C++ dei blocchi da convertire, definiti, assieme ad alcune variabili di supporto, nella sezione delle variabili globali.

Conversione delle variabili locali A differenza degli *header*, della procedura `setup()` e delle *variabili di supporto*, per la definizione delle variabili globali, rappresentanti quelle locali definite in Snap!, è necessaria l'ispezione del modello EMF generato e la conversione dei costrutti in esso contenuti nelle controparti C/C++. In particolare, occorre innanzitutto accedere, attraverso il campo `xmlModel.variables.variables`, a tutte le variabil locali definite nel *Micromondo della Matrice* passato. Successivamente, la conversione di queste, nelle controparti C/C++, consiste nel *definire* e *inizializzare* le *variabili globali* che le andranno a rappresentare nello sketch Wiring. Per fare ciò, il *code generator* opera, per ogni variabile definita:

- la conversione del nome, in uno conforme alla sintassi richiesta dal C/C++ (p.e. sostituendo gli spazi con degli *underscore*);
- l'inferenza del tipo, tra *int, float, bool o String*, a partire dal valore assunto al momento dell'esportazione;
- l'assegnamento di un valore di inizializzazione *standard*, concorde col tipo scelto (*0* per i tipi int e float, *false* per il tipo bool e *""* per il tipo String).

Conversione dello script Come per le variabili globali (non di supporto), anche per la definizione del corpo della procedura `loop()` è necessario accedere al modello EMF del sorgente passato. Attraverso questo, e in particolare accedendo al campo `xmlModel.script.blocks`, è possibile ispezionare i blocchi rappresentanti lo script Snap! che, una volta convertiti in C/C++, potranno poi essere inseriti all'interno della suddetta procedura, ed eseguiti, come specificato in sede di analisi e mediante il controllo di un opportuno *flag*, una sola volta.

In particolare, per quanto concerne la conversione di tali blocchi, questa avviene attraverso un costrutto di *switching* col quale mappare, per ciascuno dei blocchi definiti nel *Micromondo della Matrice* (fig. 1.5), la relativa implementazione in C/C++. Nel fare ciò, occorre, tuttavia, poter prima identificare i singoli blocchi, e, siccome la grammatica definita non prevede costrutti specifici per ciascuno di essi, questo è possibile farlo attraverso i seguenti elementi:

- *il tipo/la classe di appartenenza* del blocco, ossia `VariableBlock`, `BasicBlock` e `CustomBlock`;
- *il nome del blocco*, ossia il campo `s`.

Attraverso *l'analisi della classe* è infatti possibile, in prima battuta, distinguere i blocchi che rappresentano il semplice utilizzo di una variabile (`VariableBlock`), per i quali il mapping consiste nel semplice nome della variabile referenziata (con la stessa sintassi utilizzata in sede di *definizione* di questa), da quelli che devono invece essere mappati con procedure/funzioni o altri costrutti più complessi del linguaggio C/C+. Per questi ultimi in particolare, suddivisi in `BasicBlock` e `CustomBlock`, l'identificazione può poi essere fatta attraverso il confronto del nome (il campo `s`) ad essi associato. Attraverso questo, infatti, è possibile mappare, per un preciso blocco, il codice (ossia la funzione, la procedura o l'insieme di istruzioni C/C++) rappresentante il medesimo comportamento, nonché convertire nel modo più opportuno gli eventuali argomenti disponibili (contenuti nel campo `args`). Nello specifico, a partire dal nome del blocco da convertire, è possibile stabilire il numero e il tipo degli argomenti ad esso associati e, di conseguenza, come convertirli. In particolare, nel caso l'argomento da convertire sia, a sua volta, un blocco o uno script (ossia un'insieme di blocchi), per questo verrà fatta una conversione *ricorsiva* (ossia una conversione del/i blocco/chi e dei relativi argomenti, a loro volta ricorsivamente convertibili), mentre nel caso sia un argomento Option o Literal, a questo verrà semplicemente sostituito il valore del relativo campo `value` (che nel caso del tipo `Const` - utilizzato sia per valori numerici sia per stringhe - vedrà l'aggiunta delle virgolette nel caso non rappresenti effettivamente un numero).

Per maggiore dettaglio, seguono ora i blocchi Snap! mappati e la relativa conversione in linguaggio C/C++.

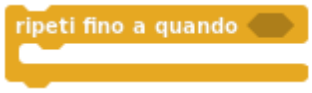

```
// stop, variabile globale di supporto, è inizializzata a false
// a ogni nuova esecuzione.
if (stop) return;
stop = true;
```


```
// «p1», l'argomento passato, può essere un valore costante
// numerico o un qualsiasi blocco ritornante un valore
// numerico.

delay(«p1»)
```


```
// «p1», il primo argomento, può essere un valore costante
// intero o un qualsiasi blocco ritornante un valore intero.
//
// «p2», il secondo argomento, è uno script, ossia l'insieme
// di blocchi da eseguire.

for (int __count__ = «p1»; __count__ > 0; __count__--) {
    «p2»
}
```




```
// «p1», il primo argomento, può essere un valore costante
// booleano o un qualsiasi blocco ritornante un valore
// booleano.
//
// «p2», il secondo argomento, è uno script, ossia l'insieme
// di blocchi da eseguire.

while(!«p1») {
   «p2»
}
```


```
// «p1», il primo argomento, può essere un valore costante
// booleano o un qualsiasi blocco ritornante un valore
// booleano.
//
// «p2», il secondo argomento, è uno script, ossia l'insieme
// di blocchi da eseguire.

if(«p1») {
    «p2»
}
```


```
// «p1», il primo argomento, può essere un valore costante
// booleano o un qualsiasi blocco ritornante un valore
// booleano.
//
// «p2» e «p3», il secondo e il terzo argomento, sono script,
// ossia l'insieme di blocchi da eseguire.

if(«p1») {
    «p2»
} else {
    «p3»
}
```


```
// pixel_matrix è un'istanza della classe Adafruit_NeoMatrix,
// rappresentante la  matrice di led.
//
// «pi», gli argomenti passati, possono essere valori costanti
// interi o blocchi ritornanti un valore intero.
//
// delay() è una procedura predefinita.
//
// DELAY è una costante definita nel file config.h

pixel_matrix.drawPixel(«p1», «p2», pixel_matrix.Color(«p3», «p4»,
                                                      «p5»));
pixel_matrix.show();
delay(DELAY);
```


```
pixel_matrix.clear();
```


```
// «p1» e «p2», gli argomenti passati, possono essere valori
// costanti numerici o qualsiasi blocco ritornante un valore
// numerico.

(«p1» + «p2»)
(«p1» - «p2»)
(«p1» * «p2»)
(«p1» / «p2»)
```


```
// «p1» e «p2», gli argomenti passati, possono essere valori
// costanti numerici o qualsiasi blocco ritornante un valore
// numerico.

(«p1» < «p2»)
(«p1» == «p2»)
(«p1» > «p2»)
```


```
// «p1» e «p2», gli argomenti passati, possono essere valori
// costanti booleani o qualsiasi blocco ritornante un valore
// booleano.

(«p1» && «p2»)
(«p1» || «p2»)
!«p1»
```


```
// «p1» e «p2», gli argomenti passati, possono essere valori
// costanti numerici o qualsiasi blocco ritornante un valore
// numerico.
//
// round e random() sono funzioni predefinite.

(«p1» % «p2»)
round(«p1»)
random(«p1», «p2» + 1)
```


```
// il primo argomento, convertito, rappresenta il nome della
// funzione scelta tra quelle disponibili.
//
// «p2», il secondo argomento, può essere un valore costante
// numerico o qualsiasi blocco ritornante un valore numerico.
//
// ceil, log, log10, exp e pow sono funzioni predefinite.

// «p1» == ceiling:
ceil(«p2»)

// «p1» == ln:
log(«p2»)

// «p1» == log:
log10(«p2»)

// «p1» == e^:
exp(«p2»)

// «p1» == 10^:
pow(10, («p2»))
```


```
// Il primo argomento («p1»), convertito, è il nome della
// variabile da considerare.
//
// «p2», il secondo argomento, può essere un valore costante
// qualsiasi o un blocco ritornante un valore qualsiasi.

«p1» = «p2»
```


```
// Il primo argomento («p1»), convertito, è il nome della
// variabile da considerare.
//
// «p2», il secondo argomento, può essere un valore costante
// numerico o un blocco ritornante un valore numerico.

«p1» += «p2»
```

4.4 Testing

Durante tutta la fase sviluppo della *grammatica* del DSL *Snap2ino* e del *code generator*, di fondamentale utilità è stata l'attività di testing svolta, attraverso il framework *Junit* e le classi di test `Snap2inoParsingTest.xtend`, `CodeGeneratorTest.xtend` e `ModelUtiliesTest.xtend` (fig. 4.3). Per quanto riguarda il testing della *grammatica*, la classe `Snap2inoParsingTest.xtend` racchiude tutti i test derivanti dall'evoluzione della *grammatica* realizzata, a seguito di ogni nuova regola e costrutto definiti, con l'obiettivo di verificare che il *parsing* attraverso questa sia come ci si aspetta, ossia consenta la lettura degli input attesi e il rifiuto di quelli errati, ma soprattutto generi un modello EMF in linea con quello dei tag xml realizzato in fase di analisi (fig. 4.1). Per quanto riguarda, invece, i test del *code generator* e della classe di utilità utilizzata da questo, per l'ispezione e la conversione di alcuni elementi del modello EMF, questi sono contenuti, rispettivamente, nelle classi `CodeGeneratorTest.xtend` e `ModelUtiliesTest.xtend`. Mentre i test per il primo hanno lo scopo di verificare che il codice generato sia come ci si aspetta, ossia rispetti la *sintassi* del C/C++ e sia compilabile, quelli per il secondo servono a verificare la correttezza dei valori restituiti dall'ispezione/conversione di certi elementi del modello EMF generato, come la

conversione del nome delle variabili, l'inferenza del tipo di queste e del valore di inizializzazione.

4.5 Guida utente, funzionalità e limitazioni del primo prototipo

Il prototipo realizzato consiste in un semplice file `.jar` eseguibile da terminale. Questo, dato in input un file `.xml` rappresentante il *Micromondo della Matrice* validato, corretto ed esportato dall'IDE Snap!, produce un progetto Wiring direttamente compilabile e caricabile, come un qualsiasi sketch `.ino`, attraverso Arduino IDE.

Guida utente Assicurarsi di avere installato sul pc sia Arduino IDE sia le librerie Adafruit per il controllo della matrice di led (Adafruit_GFX, Adafruit_NeoMatrix e Adafruit_NeoPixel).

Lanciare, da terminale, il comando `java -jar snap2ino.jar file.xml`, specificando il file `.xml` (esportato direttamente da Snap!) da convertire nello sketch Arduino desiderato. Così facendo si avrà la generazione, in corrispondenza del path del sorgente specificato, di una directory `src-gen/` contenente il file `.snap2ino`, derivato dall'eliminazione di tutti i tag non riconoscibili dal *parser* e utilizzato come input della successiva fase di traduzione, e di una sottodirectory contenente il progetto Wiring generato, ossia il file `.ino` e il file di configurazione `config.h`.

Una volta generato il progetto, questo può essere aperto, compilato e caricato mediante Arduino IDE, come un qualunque altro sketch Wiring.

Funzionalità

- Conversione del *Micromondo della Matrice* in sketch Wiring.
- Generazione automatica del file di configurazione (`config.h`) per ogni progetto generato.
- Supporto ai tipi intero signed (da -32,768 a 32,767), float signed (da 3.4028235E+38 a -3.4028235E+38), booleano e stringa.
- Supporto ai blocchi definiti nel *Micromondo della Matrice* (eccetto *"esegui in modalità turbo"* e *"quando<condizione>"*).
- Supporto a nomi di variabile semplici e composti.

Limitazioni e vincoli

- Mancanza di una GUI.
- Mancato supporto alla manipolazione delle stringhe.
- Mancato controllo del tipo degli argomenti dei blocchi e segnalazione degli eventuali errori di incompatibilità , relativamente alle conversioni C/C++ effettuate, solo in fase di compilazione dello sketch.
- Supporto limitato al solo *Micromondo della matrice*, quindi a uno solo script e un solo Sprite, alle sole variabili locali e ai soli blocchi base e custom predefiniti.
- Richiesta l'installazione di Arduino IDE, per la compilazione e il caricamento dei sorgenti generati, e delle librerie per il controllo della matrice di led (Adafruit_GFX, Adafruit_NeoMatrix, Adafruit_NeoPixel).
- Richiesta l'immutabilità del tipo/utilizzo delle variabili Snap!.
- Richiesta l'inizializzazione di tutte le variabili utilizzate in Snap!, prima dell'esportazione.

4.6 Sommario

In questo capitolo, è stato trattato il progetto del primo prototipo del *transpiler Snap2ino*. Partendo dall'analisi dell'input da processare e dell'output da generare, è poi stata affrontata la fase di progettazione, al cui centro vi è stata la definizione del DSL *Snap2ino* e l'utilizzo del framework Xtext. Attraverso quest'ultimo, è stato infatti possibile realizzare la *grammatica* per il DSL richiesto e il *code generator* attraverso il quale convertire i sorgenti passati e filtrati, rappresentanti il *Micromondo della Matrice*, in sketch Wiring compilabili ed eseguibili sulla scheda Arduino Uno.

Nel capitolo successivo, partendo dal prototipo qui realizzato, verrà affrontato, invece, il progetto del secondo prototipo, caratterizzato dall'aggiunta, rispetto a questo, della possibilità di definire nuovi blocchi in Snap! e il mapping con la relativa implementazione in C/C+, consentendo la conversione di generici *Micromondi*, e dal supporto per l'esecuzione concorrente di Sprite e script.

Capitolo 5

Un framework che evolve: dal *Micromondo della Matrice* a generici micromondi

Nel precedente capitolo è stato trattato il progetto del primo prototipo del transpiler *Snap2ino*, in grado di convertire il *Micromondo della Matrice* - progetto Snap! caratterizzato da un singolo Sprite, uno script, eventuali variabili locali e un insieme predefinito di blocchi base e custom - in uno sketch Wiring eseguibile sulla piattaforma Arduino.

In questo capitolo, viene ora trattato il progetto del secondo prototipo che, partendo dalle funzionalità messe a disposizione dal primo e attraverso le medesime tecnologie, ha l'obiettivo di ottenere un transpiler più generale e in grado di essere utilizzato per la conversione di un generico micromondo/-progetto realizzato in Snap!, composto da un numero arbitrario di script e Sprite e da eventuali variabili sia locali che globali. In particolare, includendo il supporto ad aspetti avanzati come la gestione degli eventi e l'esecuzione di task concorrenti, *Snap2ino* mira a rendere possibile, a partire da questo secondo prototipo, l'utilizzo di Snap! per la programmazione di qualsiasi sistema embedded basato su Wiring. Partendo dalla descrizione del processo di generalizzazione attraverso il quale ottenere gli obiettivi appena citati (5.1), è trattata l'analisi dei requisiti più critici richiesti da questo (5.2) e, infine, affrontata la fase di progettazione attraverso la quale poterli raggiungere (5.3). Completano il progetto, la fase di testing delle modifiche apportate e delle funzionalità aggiunte (5.4) e una breve guida utente corredata da un riassunto circa le principali funzionalità, le limitazioni e i vincoli di questo secondo prototipo realizzato (5.5).

5.1 Processo di generalizzazione

Come si ricorda, la progettazione del primo prototipo è stata guidata da requisiti specifici e mirati alla conversione del solo *Micromondo della Matrice*, ossia di un progetto Snap! composto da un *singolo Sprite, un singolo script, un insieme di variabili locali e di blocchi base e custom predefiniti.* Partendo da questi, il processo di *generalizzazione* da attuare, volto a consentire la conversione di un *generico* micromondo/progetto Snap!, consiste nei tre seguenti punti fondamentali:

- Consentire all'utente la definizione del mapping di un generico blocco Snap! con una propria implementazione C/C++, da utilizzare per la generazione dello sketch Wiring.
- Supportare la conversione di micromondi/progetti composti da più Sprite, più script, da variabili locali e globali, l'esecuzione concorrente di questi e la conversione del blocco base *"esegui in modalità turbo"* (utilizzato in Snap! per l'esecuzione del codice in esso contenuto senza effettuare *script-switching*, nel contesto dell'esecuzione concorrente di più script/Sprite).
- Integrare il supporto agli eventi, ossia all'esecuzione degli script in risposta a particolari condizioni (blocco esegui *"quando<condizione>"*).

Per quanto riguarda il primo punto, consentire all'utente un *mapping personalizzato* dei blocchi utilizzati in Snap! (base o custom) con il codice C/C++ che questo vuole sia eseguito, consente al transpiler di poter mappare, oltre ai blocchi predefiniti del *Micromondo della Matrice*, un qualsiasi set di blocchi definito in Snap! per un dato micromondo/progetto, caratterizzanti il linguaggio messo a disposizione da quest'ultimo. Un meccanismo simile consente, da un lato, l'estensione e la personalizzazione del framework a seconda delle proprie necessità, includendo nuovi blocchi non originariamente previsti e definendo esplicitamente cosa si vuole che *Snap2ino* generi, dall'altro, la possibilità di integrare nel proprio progetto librerie di supporto compatibili con Wiring esistenti, consentendo quindi l'utilizzo dei sensori, degli attuatori e dei protocolli richiesti per i propri scopi. In particolare, quindi, si abilita la possibilità di andare ben oltre al *Micromondo della Matrice*, proiettando *Snap2ino* in un contesto di utilizzo molto più ampio, riguardante, in generale, lo sviluppo di qualsiasi progetto basato su sistemi embedded, supportanti Wiring, attraverso la piattaforma Snap!.

A tal proposito, nell'ottica della modellazione di un qualsiasi sistema embedded, occorre tuttavia considerare come questo possa, in genere, essere

caratterizzato dall'esecuzione concorrente di più operazioni (*task*), eventualmente necessitanti di comunicare tra loro, piuttosto che da un modello di esecuzione a eventi. In particolare, siccome per la modellazione di tali logiche in Snap! è indispensabile l'utilizzo di micromondi/progetti composti da più script e Sprite, dotati, oltre che di quelle locali, di variabili globali attraverso le quali poter comunicare, ne conseguono quindi il secondo e terzo punto. È inoltre importante notare come, attraverso questi ultimi due, oltre a poter consentire la programmazione di qualsiasi sistemi embedded attraverso Snap!, *Snap2ino* consentirà, in aggiunta, la possibilità di modellare veri e propri *sistemi ad agenti*, rappresentati, ad esempio, per mezzo di Sprite comunicanti tra loro mediante semplici variabili globali.

5.2 Analisi

Dei tre punti previsti dal processo di generalizzazione, quello che effettivamente ha necessità di una fase di analisi è il secondo. In particolare, occorre analizzare, innanzitutto, il file xml contenente la rappresentazione del micromondo/progetto esportato, al fine di individuare in esso i tag rappresentanti gli Sprite, le variabili globali e gli script (che per il primo prototipo erano stati esclusi) e poter poi aggiornare, di conseguenza, la grammatica del DSL Snap2ino e il generatore di codice (*mapping sintattico*). Una volta rivisitata la *sintassi* del linguaggio di partenza, occorre poi rianalizzarne la *semantica*, ossia il *modello d'esecuzione* utilizzato dall'ambiente Snap!. In particolare, a differenza di quanto fatto per il primo prototipo, occorre ora analizzare come vengono eseguiti, *concorrentemente*, gli script componenti gli Sprite di un generico micromondo/progetto. Una volta fatto ciò, sarà poi possibile organizzare, di conseguenza, il codice che dovrà essere eseguito sulla piattaforma di destinazione (Wiring/Arduino), in maniera tale da ottenere da questa il medesimo comportamento riscontrato dall'ambiente Snap! (*mapping semantico*).

5.2.1 Analisi sintattica di un generico *Micromondo* Snap!

Partendo dal file xml derivante dall'esportazione di un generico micromondo/progetto Snap!, e filtrando questo in modo da mantenere, questa volta, anche le informazioni riguardanti gli Sprite definiti, i relativi script e le variabili globali, è possibile ottenere il seguente sorgente xml:

```
<!-- ROOT -->
<project>
    <sprites>
```

```
        <!-- SPRITES COMPONENTI IL PROGETTO -->
        <sprite name="Sprite-1" idx="1">
            <variables>
                <!-- VARIABILI LOCALI ALLO SPRITE-1 -->
            </variables>
            <scripts>
                <!-- SCRIPTS DELLO SPRITE-1 -->
                <script>
                    <!-- BLOCCHI DELLO SCRIPT -->
                <script>
                <script>
                    <!-- BLOCCHI DELLO SCRIPT -->
                <script>
            </scripts>
        </sprite>

        <sprite name="Sprite-2" idx="2">
            <variables>
                <!-- VARIABILI LOCALI ALLO SPRITE-2 -->
            </variables>
            <scripts>
                <!-- SCRIPTS DELLO SPRITE-2 -->
                <script>
                    <!-- BLOCCHI DELLO SCRIPT -->
                <script>
            </scripts>
        </sprite>
    </sprites>

    <variables>
          <!-- VARIABILI GLOBALI -->
    </variables>
</project>
```

Da listato presentato, si evidenzia come, a differenza di quanto individuato nella fase di analisi per il primo prototipo (4.2.2), l'intero progetto sia ora contenuto in un'apposito root-tag (`<project>`). All'interno di questo è presente la sezione (il tag `<sprites>`) contenente gli Sprite definiti (`<sprite>`), ognuno dei quali caratterizzato da un nome (l'attributo `name`) e da un'id univoco (l'attributo `idx`) associato in base all'ordinamento nell'area degli Sprite, e quella delle variabili globali definite (il tag `<variables>`). All'interno di ogni Sprite è poi posibile individuare la sezione delle variabili locali (la medesima del primo prototipo) e degli script (racchiusi dal tag `<scripts>`) definiti per quel determinato Sprite.

A partire da tale listato, analogamente a quanto fatto per il primo prototipo (4.1), è possibile definire uno schema UML (fig. 5.1) rappresentante

in maniera più chiara e formale l'organizzazione che i vari tag xml possono avere, utilizzabile poi in seguito come riferimento per l'aggiornamento della grammatica del DSL Snap2ino. In tale schema sono riportati in nuovi tag

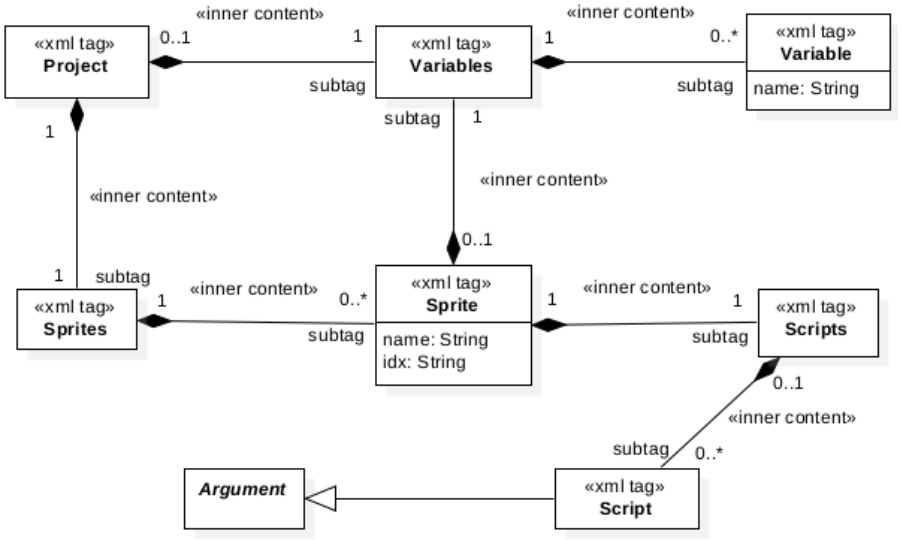


Figura 5.1: Organizzazione dei tag xml (significativi) di un generico micromondo/progetto multi-script/Sprite, esportato dall'ambiente Snap!.

emersi dall'analisi del file xml esportato (Project, Sprites, Sprite e Scripts), e le associazioni di composizione tra questi e tra i principali tag significativi del primo prototipo (Script e Variables in particolare). Si evidenzia, inoltre, come uno Script sia associato opzionalmente a un tag Scripts, in quanto potrebbe essere alternativamente l'argomento di un blocco, e come anche Variables sia opzionalmente un sotto-tag di Project e Sprite, in quanto rappresentante, in maniera esclusiva, o una sezione di variabili *locali* o *globali*.

5.2.2 Analisi del modello di esecuzione dell'ambiente Snap!

Nell'ambito del progetto del primo prototipo è stato introdotto il *modello d'esecuzione* utilizzato dall'ambiente Snap! circa l'esecuzione di un generico micromondo/progetto definito, concentrando tuttavia l'attenzione al caso particolare del *Micromondo della Matrice* (4.2.1). Per quanto concerne gli obiettivi del secondo prototipo, tuttavia, occorre ora scendere maggiormente nel dettaglio circa quanto analizzato nel capitolo precedente, focalizzandosi ora su come vengono effettivamente eseguiti Sprite e script concorrenti.

A tal proposito, si fa quindi riferimento al manuale di Snap! [39, p. 72]:

Snap! can be running several scripts at once, within a single sprite and across many sprites. If you only have one computer, how can it do many things at once? The answer is that only one is actually running at any moment, but Snap! switches its attention from one script to another frequently. At the bottom of every looping block (repeat, repeat until, forever), there is an implicit "yield" command, which remembers where the current script is up to, and switches to some other script, each in turn. At the end of every script is an implicit "end thread" command (a thread is the technical term for the process of running a script), which switches to another script without remembering the old one.

A partire da quanto appena definito, se ne evince come ogni script di uno Sprite sia eseguito, dall'ambiente Snap!, in maniera *cooperativa* rispetto agli altri script dello stesso o di altri Sprite. In particolare, non appena uno script inizia l'esecuzione, questa continua finché non viene eseguito l'ultimo blocco disponibile, dopodiché il controllo viene passato a un altro script. L'unica eccezione a tale funzionamento è per i *cicli*, ossia i blocchi d'iterazione che, nel caso del primo prototipo, sono *"ripeti (N) volte"* e *"ripeti fino a quando<>"* (fig. 1.5). Infatti, al termine di ogni iterazione (a prescindere dal livello di annidamento), l'esecuzione dello script viene messa in pausa in favore di un altro script (*script switching*). Oltre ai blocchi d'iterazione, anche il blocco *"esegui in modalità turbo"* altera la normale esecuzione degli script Snap!. In particolare, i blocchi in esso contenuti sono eseguiti senza che avvenga mai *script switching*, persino nel caso dei blocchi d'iterazione, effettuando questo solo dopo aver eseguito l'ultimo di questi disponibile. A questo punto risulta però fondamentale considerare che, dal momento che i blocchi custom sono generalmente definiti a partire da blocchi preesistenti, e siccome l'analisi del sorgente xml non ispeziona come tali blocchi siano effettivamente definiti (dal momento che se ne lascia l'implementazione in C/C++ all'utente), nel caso questi contengano blocchi che effettuano *script switching* la relativa conversione non porterebbe al medesimo comportamento su Arduino. Per ovviare a ciò, si considera che ogni blocco custom da convertire *wrappi* sempre i propri blocchi con un blocco *"esegui in modalità turbo"*, in modo da essere sicuri che venga eseguito sempre e solo uno *script switching* al termine dell'esecuzione di un blocco custom. Inoltre, siccome l'annidamento di blocchi *"esegui in modalità turbo"* porta ad effettuare *script switching* solo per quello più esterno, in questo modo è possibile utilizzare blocchi custom preesistenti per definirne di nuovi, garantendo lo stesso l'esecuzione di un singolo *script switching*. In definitiva, da tale definizione ne consegue come i blocchi custom siano ora da considerarsi come unità d'esecuzione *atomiche*, all'interno del contesto concorrente descritto.

Per quanto concerne, invece, l'ordine di esecuzione (*schedulazione*) degli script di uno o più Sprite, a seguito di opportuni test eseguiti sull'ambiente Snap! (con riferimento al caso di script con primo blocco *"quando si clicca su [bandierina verde]"* e *"quando<condizione>"* - gli unici considerati per il progetto del secondo prototipo - e all'esecuzione di questi solo tramite la pressione della sopracitata "bandierina verde" - rappresentante l'avvio del microcontrollore), è emerso come questo possa essere determinato nel seguente modo:

1. All'interno di un progetto Snap!, gli script definiti vengono ripartiti tra due code con priorità differente, in base al tipo di blocco che li manda in esecuzione. Nello specifico, vengono inseriti nella coda con priorità massima tutti gli script con primo blocco *"quando si clicca su [bandierina verde]"*, mentre quelli con primo blocco *"quando<condizione>"* in quella con priorità minore. Questi ultimi, in particolare, possono essere schedulati solo dopo che anche l'ultimo script della coda con priorità massima ha eseguito *script switching*.

2. All'interno di ciascuna coda, gli script sono raggruppati in base allo Sprite nel quale sono definiti, e ciascun gruppo è ordinato in base a quando lo Sprite è stato creato, assegnando la massima priorità al gruppo associato allo Sprite più "vecchio" e una via via minor priorità a quelli di ogni nuovo Sprite creato. È importante notare come tale ordinamento tra i vari Sprite (e degli script a questi associati) sia riscontrabile nella collocazione dei relativi tag `<sprite>` come sotto-tag di `<sprites>`.

3. All'interno di ogni gruppo definito, contenente gli script di un dato Sprite, la priorità di ciascuno script è stabilita in base al momento della creazione di questo e a quando questo è stato mosso l'ultima volta, assegnando priorità minima all'ultimo script creato o spostato, e massima a quello più "vecchio" o che non si è spostato da più tempo. Anche in questo caso, la priorità assegnata ai vari script di uno Sprite è riscontrabile nell'ordine con il quale i relativi tag `<script>` sono effettivamente collocati come sotto-tag di `<scripts>`.

Alla luce di quanto emerso dall'analisi del *modello d'esecuzione* utilizzato dal'ambiente Snap!, il cuore della fase di progettazione, per questo secondo prototipo, sarà quindi fare in modo che sulla scheda Arduino Uno, sprovvista di sistema operativo, sia possibile eseguire in maniera *concorrente* ciascuno degli script (*task*) definiti nel micromondo/progetto da convertire, garanten-

do, inoltre, una *schedulazione* quanto più possibile fedele a quella realizzata da Snap!, partendo dal *modello d'esecuzione* utilizzato per il primo prototipo.

5.3 Progettazione

5.3.1 Personalizzazione del mapping

Per quanto riguarda il primo punto caratterizzante il processo di generalizzazione del *transpiler Snap2ino*, ossia consentire all'utente la possibilità di definire egli stesso che codice C/C++ eseguire in corrispondenza della chiamata di un dato blocco Snap!, la soluzione è stata quella di agire sul processo di conversione attuata dal *code generator*. In particolare, mentre per i blocchi definiti per il *Micromondo della Matrice* la conversione di *default* rimane *embedded* nel funzionamento del generatore di codice, per ogni altro blocco si ricorre, invece, all'ispezione di un'apposito file di mapping `custom_blocks.json`. Questo file, autogenerato da *Snap2ino* nel medesimo path se non presente (file di *mapping globale*), contiene il mapping tra le *signature* dei blocchi che si vuole convertire e il nome della relativa funzione o procedura C/C++ che si vuole venga richiamata al posto di questi, evitando limitazioni o problemi imposti dal dover utilizzare nomi identici per i blocchi e per le controparti C/C++. Un esempio di questo, per il blocco *accendi led*, è il seguente:

```
"accendi led x: %n y: %n (colore r: %n g: %n b: %n )": "accendi_led"
```

Attraverso questo meccanismo, ogniqualvolta il code generator si troverà a dover convertire un blocco con nome (il campo `s`) corrispondente ad uno di quelli presenti nel file di mapping (*globale o passato* come argomento al transpiler), inserirà nell'output, al posto di questo, la chiamata alla procedura/funzione da mappare e gli eventuali argomenti (convertiti e nell'ordine definito dal blocco Snap!) per questa. A proposito di questi ultimi, è importante notare come, per ovvi motivi, tale meccanismo escluda la possibilità di mappare blocchi che richiedono come argomenti degli script.

Una volta definito il mapping, all'utente non resta altro che implementare le funzioni/procedure in esso specificate, con numero, ordine e tipo degli eventuali parametri concorde con quello dei rispettivi blocchi da mappare. In particolare, per quanto concerne la definizione di queste, l'utente può scegliere se includerle, manualmente, nel progetto dello sketch autogenerato, piuttosto che utilizzare la libreria di default automaticamente inclusa dal transpiler: `Snap2ino_CustomBlocks.h/cpp`. Questa, fornita assieme a *Snap2ino* (`snap2ino.jar`), contiene, in particolare, la definizione delle procedure e funzioni richiamate dalle controparti C/C++ di alcuni blocchi

convertiti (derivanti dal refactoring del codice che nel primo prototipo era direttamente inserito nello sketch generato). Oltre a queste, l'utente può però includere la definizione delle proprie procedure e funzioni, utilizzate per il mapping personalizzato dei blocchi, nonché definire il proprio *setup* dell'ambiente attraverso la procedura predefinita `custom_setup()`, richiamata automaticamente dalla procedura `setup()` del sorgente principale (`.ino`).

5.3.2 Supporto multi Sprite/script e concorrenza

Dal momento che il secondo prototipo basa il suo funzionamento sul primo realizzato, affinché sia ora possibile supportare la conversione di micromondi/progetti composti da più Sprite/script e variabili sia locali che globali, occorre modificare le tre fasi operative costituenti il funzionamento del transpiler:

1. Acquisizione del sorgente da processare e filtraggio delle informazioni (tag) significative in esso contenute.
2. Lettura e comprensione delle informazioni filtrate.
3. Conversione delle informazioni filtrate e generazione dei file richiesti, costituenti lo sketch da compilare e caricare su Arduino.

In particolare, mentre per il primo punto basta modificare le *RegEx* utilizzate, in modo da includere le nuove informazioni necessarie alla conversione del micromondo/progetto multi-script/Sprite, e generare un file `.snap2ino` conforme alla nuova organizzazione individuata (fig. 5.1), per quanto concerne il secondo e il terzo occorre, rispettivamente:

- *modificare la grammatica del DSL*: in modo da inferire il nuovo metamodello EMF relativo e consentire il *parsing* dei sorgenti `.snap2ino` prodotti dal nuovo processo di filtraggio;
- *modificare il generatore di codice*: in modo da generare tutti gli script e gli Sprite richiesti, con le relative variabili locali e globali, e organizzando il codice così generato in modo da garantirne un'esecuzione, attraverso Arduino Uno, il più possibile conforme a quella attuata da Snap!, in precedenza analizzata (5.2.2).

Aggiornamento della grammatica

Per quanto concerne la nuova grammatica del DSL Snap2ino, questa è stata modificata in modo che la *sintassi* relativa accettasse sorgenti contenenti la nuova organizzazione dei tag xml analizzata, e che il metamodello EMF

inferibile, ossia il modello per la rappresentazione in memoria dei sorgenti da *parsare*, fosse quanto più simile possibile al modello definito in analisi (fig. 5.1):

```
Project:
    '<project>'
        sprites = Sprites
        globalVariables = Variables
    '</project>'
;

Sprites:
    '<sprites>'
        sprites += Sprite*
    '</sprites>'
;

Sprite:
    '<sprite' 'name=' name = STRING 'idx=' idx = STRING '>'
        localVariables = Variables
        scripts = Scripts
    '</sprite>'
;

Variables:
     {Variables} '<variables>'
                     variables += Variable*
                 '</variables>'
;

Variable:
    '<variable' 'name=' name = STRING '>'
        innerContent = Literal
    '</variable>'
;

Scripts:
    {Scripts} '<scripts>'
                  scripts += Script*
              '</scripts>'
;

Script:
    {Script} '<script>'
                 blocks += Block*
             '</script>'
;

Block:
```

```
    VariableBlock | BasicBlock | CustomBlock
;

VariableBlock:
    '<block' 'var=' variable = STRING '>''</block>'
;

BasicBlock:
    '<block' 's=' name = STRING '>'
        args += (Argument)*
    '</block>'
;

CustomBlock:
    '<custom-block' 's=' name = STRING '>'
        args += (Argument)*
    '</custom-block>'
;

Argument:
    Script | Block | Option | Literal
;

Option:
    '<l>'
        '<option>' value = XmlString'</option>'
    '</l>'
;

Literal:
    BoolValue | ConstValue
;

BoolValue:
    {BoolValue} '<l>'?
                    ((value ?= '<bool>true</bool>') |
                                '<bool>false</bool>')
                '</l>'?
;

// Includes strings (i.e. chars sequence without double quotes)
// and signed integers/floating point numbers.
ConstValue:
    '<l>' value = XmlString '</l>'
;

XmlString:
    ID XmlString?
;
```

```
/*
* Custom ID defined to include both signed integer and signed
* floating values.
* NB: < and >  symbols are excluded due to avoid ambiguity
* with the xml syntax.
*/
@Override
terminal ID:
    ( '\\' . | !('\\'|'"'|'\''|' '|'\t'|'\r'|'\n'|'<'|'>') )+
;
```

Com'è possibile notare, le modifiche apportate riguardano essenzialmente l'aggiunta dei costrutti rappresentanti il progetto (`Project`), gli Sprite (`Sprites` e `Sprite`) e gli script (`Scripts`), lasciando inalterati i costrutti definiti per il primo prototipo.

Aggiornamento del generatore di codice

Alla luce di quanto emerso durante l'analisi del *modello d'esecuzione* utilizzato da Snap! (5.2.2), le principali modifiche da attuare, per poter eseguire concorrentemente più script e Sprite analogamente a quanto fatto da questo, riguardano:

- l'organizzazione dello sketch Wiring, ossia la conversione degli script, degli Sprite e delle rispettive variabili locali e globali, in maniera tale che l'esecuzione (*schedulazione*) di questi sia analoga a quella in Snap!;
- la conversione dei blocchi che devono eseguire script switching (*"ripeti (N) volte"* e *"ripeti fino a quando<>"*);
- la conversione di quei blocchi che devono comportarsi diversamente in un contesto concorrente (i blocchi custom e *"attendi (N) secondi"*);
- la conversione del nuovo blocco *"esegui in modalità turbo"*, in modo da consentire l'esecuzione di blocchi che normalmente eseguirebbero *script switching* senza che questo però avvenga, ossia ottenendo un comportamento così come definito nel primo prototipo.

Per quanto concerne tali punti, segue ora la descrizione circa come venga generato un progetto Wiring a partire dal sorgente *parsato*, in modo tale che l'ordine d'esecuzione degli script convertiti sia il medesimo riscontrato su Snap!, cui segue il dettaglio su come le controparti C/C++ che rappresentano tali script, siano definite in modo da consentirne l'esecuzione concorrente attraverso lo *script switching*.

Generazione del progetto Wiring A partire dall'ispezione del modello EMF rappresentante il sorgente *parsato*, la generazione del progetto Wiring (all'interno dell'apposita directory autogenerata), rappresentante il micromondo/progetto Snap! da convertire, è eseguita come segue:

1. Per le variabili globali del micromondo/progetto viene generato un'apposito header file (`global_vars.h`), contenente la dichiarazione (`extern`) di queste, da includere nei file `.cpp` rappresentanti gli Sprite convertiti.

2. Per ogni Sprite definito (già ordinato in base alla priorità di esecuzione come visto in sede di analisi 5.2.2) sono generati un sorgente `.cpp` e un header file `.h`. Il primo contiene la definizione delle procedure rappresentanti gli script (anch'essi già ordinati) dello Sprite, le variabili locali (accessibili solo agli script/procedure di questo - `static` scope) e l'inclusione del file `global_vars.h`, precedentemente generato, e della libreria `Snap2ino_CustomBlocks.h` (5.3.1). Il secondo contiene, invece, i prototipi di suddette procedure, rappresentanti gli script dello Sprite. Partendo dall'ordinamento col quale sono ispezionati/convertiti i vari Sprite e script, è possibile imporre la schedulazione che questi ultimi devono avere semplicemente definendo le procedure che li rappresentano con la firma `void script_[ab][Java-int]()`. Indicante con una lettera (`a` o `b`) la coda di priorità (determinata in base al primo blocco dello script) e con un numero l'ordine di conversione tra tutti gli script, attraverso l'ordinamento della firma delle procedure definite è possibile definire l'ordine di esecuzione che queste devono avere.

3. Infine, è generato il file principale del progetto (`.ino`). Questo contiene, nella sezione degli header, l'inclusione di tutti gli *header file* contenenti i prototipi delle procedure definite per gli script e della libreria `Snap2ino_CustomBlocks.h`, nella sezione delle variabili globali, la definizione delle variabili condivise tra i vari Sprite ed esportate nei rispettivi file `.cpp`, nella procedura `setup()`, la chiamata a `custom_setup` (definito nella libreria di *Snap2ino* importata) e infine, nella procedura `loop()`, la chiamata di tutte le procedure rappresentanti gli script convertiti, ordinate attraverso la propria signature di definizione. È importante notare come quest'ultima procedura, in particolare, sia passata dall'eseguire una sola volta il codice associato all'unico script da convertire, nel primo prototipo, a *scheduler* attraverso il quale susseguire l'esecuzione dei vari *task* derivanti dalla conversione degli script del micromondo/progetto passato.

Esecuzione concorrente e *script switching* Una volta definito l'ordine di esecuzione delle procedure rappresentanti i vari script convertiti, affinché sia possibile effettuare il cosiddetto *script switching*, individuato in sede di analisi, è ora necessario definire un modo per poter sospendere l'esecuzione di uno script, salvarne lo stato e passare a quello successivo, per i seguenti casi: al termine di ogni ciclo (ossia dopo l'esecuzione dell'ultimo blocco disponibile all'interno di *"ripeti (N) volte"* e *"ripeti fino a quando<>"*), dopo il blocco *"esegui in modalità turbo"*, quando si richiama il blocco *"attendi (N) secondi"* (per il quale si vuole modellare un comportamento di attesa non bloccante tutti gli script in esecuzione ma solo quello chiamante) e dopo i blocchi custom.

Il meccanismo escogitato al fine di consentire l'esecuzione concorrente richiesta, su di un dispositivo (Arduino Uno) sprovvisto di sistema operativo, consiste nel definire il corpo di ogni procedura/script come una *macchina a stati finiti sincrona* (*sfsm*), le cui *reazioni* sono scatenate dalla chiamata della procedura stessa ad opera del `loop`. Ciascuna *sfsm* è implementata in C/C++ attraverso il costrutto di selezione `switch`, secondo il modello proposto da *Moore*[1], e alcune variabili (`static`) locali alla procedura, utilizzate per il salvataggio degli stati da una reazione all'altra. Attraverso tale meccanismo è quindi possibile mantenere salvato, tra uno *script switching* e l'altro (rappresentati da semplici chiamate `return`), lo stato di quello che è stato eseguito, di ciò che deve essere rieseguito e di ciò che manca da eseguire.

In particolare, per quanto concerne la definizione del comportamento e della struttura della *sfsm* rappresentante un dato script, si parte da un'organizzazione base attraverso la quale consentire il mapping automatico degli script aventi come primo blocco *"quando si clicca su [bandierina verde]"*, e composta da due soli stati: `case 0`, *a partire dal quale* inserire il codice dei blocchi convertiti, e `default`, rappresenante lo stato di inattività dello script, al quale passare una volta completata l'esecuzione dello stesso.

Una volta definita tale organizzazione base, partendo dallo stato iniziale (`case 0`), il comportamento complessivo della *sfsm* è poi così costruito:

- A ogni blocco che non esegue *script switching* viene sostituita la rispettiva controparte C/C++, così come definita nel primo prototipo.
- Per ogni blocco *"ripeti (N) volte"* e *"ripeti fino a quando<>"* viene creato un nuovo stato, al quale passare al termine dell'esecuzione dello stato precedente. Il nuovo stato creato è caratterizzato, a sua volta,

[1] **Macchina di Moore:** secondo il modello di automa a stati finiti proposto da Edward Moore, le uscite (il comportamento) sono determinate esclusivamente in funzione dello stato e non anche dagli ingressi.

da una *sfsm* (ossia un costrutto `switch`) con un solo stato (`case 0`), all'interno del quale è verificata/aggiornata la condizione di iterazione e inserita la conversione (secondo le regole qui definite) dei blocchi contenuti nel loop da convertire. Ciascuna iterazione del loop convertito consiste nell'esecuzione del codice contenuto negli stati della "sub-*sfsm*" e nella conseguente chiamata (fuori dal costrutto `switch` creato) del `return` (*script switching*).
Segue un'esempio di codice per il blocco *"ripeti fino a quando<>"*:

```
__stateI__ = n;

// «condition» è la condizione specificata per la terminazione
// del ciclo.
// «script» sono i blocchi da convertiti
// secondo le regole definite per la definizione del
// comportamento della sfsm.
case n: {
    static int __stateI+1__ = 0;
    static bool __exitI+1__ = false;
    switch(__stateI+1__) {
        case 0:
        if («condition») {
            __exitI+1__ = true;
            break;
        }
        «script»
    }
    __stateI+1__ = 0;
    if(!__exitI+1__) return;
    __exitI+1__ = false; // reset della variabile poiché
                         // dichiarata static.
}
```

Com'è possibile notate dal listato presentato, per ogni *sfsm* generata (ossia per ogni costrutto `switch`) è definita la propria variabile locale `__stateI__`, in modo da rendere più comprensibile il codice se lo si volesse analizzare (nel caso di più cicli innestati). Inoltre, analogamente a quanto avviene in Snap!, al termine di ogni iterazione, indipendentemente dal fatto che il loop debba essere rieseguito o meno, viene prima fatto *script switching* e solo alla successiva chiamata della procedura (ossia alla riesecuzione dello script) verrà controllata la condizione di iterazione. Infine, si nota come, al termine di ogni iterazione, sia inoltre resettato lo stato della "sub-*sfsm*", in modo da rieseguire il codice contenuto in tutti gli stati di questa all'iterazione successiva. Com'è

possibile immaginare, nel caso del blocco *"ripeti (N) volte"* è presente una variabile aggiuntiva per contare le varie iterazioni, aggiornata dopo ogni verifica del valore e resettata nel caso siano state effettuate tutte le iterazioni richieste. La scelta di utilizzare, per la gestione dei loop, le *sfsm* in maniera ricorsiva è da ricollegarsi alla possibilità di questi di potersi sviluppare attraverso diversi livelli di annidamento (*nesting*), dalla quale è poi derivata la scelta di modellarli, quindi, come entità caratterizzate da un blocco di codice "base" e, ricorsivamente, da uno o più eventuali altri blocchi d'iterazione/loop.

- A ogni blocco *"esegui in modalità turbo"* vengono sostituite le controparti C/C++ (così come definite nel primo prototipo) dei blocchi in esso contenuti, alle quali segue il `return` (*script switching*) e la definizione di un nuovo stato, dal quale far proseguire l'esecuzione della procedura alla prossima chiamata.
  Esempio di codice:

```
// «script», l'argomento passato al blocco, rappresenta
// l'insieme di blocchi convertiti nelle controparti C/C++.
«script»
__stateI__ = n;
return;

case n:
```

- A ogni blocco custom viene sostituita la chiamata alla rispettiva funzione/procedura, alla quale sono passati (convertiti e in ordine) gli argomenti specificati (così come avveniva nel primo prototipo), e alla quale seguono, analogamente come per il blocco *"esegui in modalità turbo"*, il `return` (*script switching*) e la definizione di un nuovo stato, dal quale far proseguire l'esecuzione della procedura alla prossima chiamata.
  Esempio di codice:

```
accendi_led(x, y, 0, 255, 0);
__stateI__ = n;
return;

case n:
```

  È estremamente importante notare, in tale soluzione, che nel caso il blocco debba restituire un qualche valore (ovvero sia mappato con una funzione) questo non può essere utilizzato *direttamente* (lato Snap!)

come argomento di altri blocchi ma occorre assegnare tale valore prima a una variabile di supporto. Il motivo di ciò è evidente e risiede nel fatto che, ad esempio all'interno di un'operazione/espressione, potrebbe generare codice C/C++ sintatticamente e semanticamente non valido.

- Per ogni blocco custom *"attendi (N) secondi"* viene creato un nuovo stato, al quale passare al termine dell'esecuzione del codice associato allo stato precedente. All'interno di questo nuovo stato viene verificato se il tempo trascorso corrisponde o supera quello di attesa. Se la condizione sussiste, l'esecuzione prosegue normalmente, altrimenti, viene chiamato il `return` (*script switching*).
  Esempio di codice:

```
case n:
    ...
    __stateI__ = n+1;

// «p1», l'argomento passato al blocco, indica i secondi di
// attesa.
// millis() è una funzione predefinita di Wiring,
// ritornante il numero di millisecondi trascorsi
// dall'avvio del dispositivo.
case n+1: {
    static unsigned long __waitingTime__ = «p1»*1000;
    static unsigned long __time__ = 0;
    if (__time__ == 0) {
        __time__ = millis();
    }
    if (millis() - __time__ < __waitingTime__) {
        return;
    }
}
```

5.3.3 Supporto agli eventi

Relativamente all'ultimo punto del processo di generalizzazione, ossia l'integrazione del supporto all'esecuzione di script con primo blocco *"quando<condizione>"* (eseguiti al verificarsi della condizione specificata), per come è stato definito il nuovo meccanismo di conversione degli script, è sufficiente definire una struttura base alternativa a quella utilizzata da quest'ultimo per la *sfsm* rappresentante gli script con primo blocco *"quando si clicca su [bandierina verde]"*. In particolare, a differenza di quest'ultima, basta condizionare l'esecuzione del codice, derivante dai blocchi convertiti, mediante l'argomento `condition` specificato e modificare lo stato finale

`default` includendo il reset della variabile memorizzante lo stato corrente, in modo da consentire la riesecuzione del codice al verificarsi di nuovi eventi/condizioni. A tal proposito, occorre notare come questo comportamento porti a una totale insensibilità agli eventi nel mentre che la procedura non è "sensibile" (ossia sta già "reagendo" a un altro evento), e alla conseguente non reazione a questi. In particolare, occorre precisare come questo non sia un bug, ma l'effettivo comportamento tenuto dall'ambiente Snap!
Segue l'organizzazione base per il blocco *"quando<condizione>"*:

```
case 0:
    if(!«condition») return;
    ... // blocchi che non eseguono script-switching convertiti.

... // altri stati creati a seguito di conversione di blocchi
    // che eseguono script-switching.

default:
    __state0__ = 0;
```

5.4 Testing

Come per il progetto del primo prototipo, anche per questo è stata essenziale la fase di testing, volta a verificare il corretto funzionamento a seguito delle modifiche apportate e delle funzionalità integrate. In particolare, per quanto concerne il testing della grammatica, sono stati aggiunti nuovi test e adattati quelli esistenti, al fine di garantire il corretto parsing a seguito delle modifiche apportate a questa. Al di là della grammatica, il vero cuore della fase di test è da ricollegare alla verifica del corretto funzionamento del generatore di codice. Infatti, oltre all'aggiornamento dei test esistenti, sono stati introdotti una serie di nuovi test mirati a verificare la correttezza del meccanismo introdotto per la conversione degli script concorrenti e, in particolare, nella verifica che tutti i blocchi che eseguono *script switching* si integrassero (ed eventualmente innestassero) a dovere tra di loro e con gli altri blocchi. Infine sono stati testati anche l'effettiva conversione dei blocchi Snap! con le controparti C/C++ definite dall'utente, ossia la personalizzazione del mapping tra questi, e la conversione degli script ad eventi.

5.5 Guida utente, funzionalità e limitazioni del secondo prototipo

Come per il primo prototipo, anche per il secondo il transpiler *Snap2ino* consiste in un semplice file `.jar` eseguibile da terminale. A differenza della prima versione, però, ora è possibile convertire in progetto Wiring un *generico* micromondo/progetto Snap! validato e corretto.

Guida utente Assicurarsi di avere installato sul pc, oltre ad Arduino IDE e le librerie Adafruit per il controllo della matrice di led (Adafruit_GFX, Adafruit_NeoMatrix e Adafruit_NeoPixel), anche la libreria offerta dal framework: Snap2ino_CustomBlocks.

Al primo utilizzo del transpiler viene generato automaticamente, se non presente nella directory di questo, il file `custom_blocks.json` (globale) per la definizione del mapping tra i blocchi custom definiti in Snap! e le procedure/funzioni con le quali convertirli. Per modificare il mapping si può agire direttamente sul file globale oppure se ne può sovrascrivere occasionalmente il comportamento (il mapping) semplicemente passando al transpiler, assieme al file `.xml` da convertire, una propria copia come ultimo argomento:
`java -jar snap2ino-compiler.jar source.xml -m mapping_file.json`.
Per quanto riguarda la collocazione delle procedure/funzioni con le quali mappare i blocchi custom definiti, di default si può utilizzare la libreria fornita assieme al compilatore (automaticamente inclusa nei sorgenti autogenerati), nella quale poter definire l'implementazione di queste, oppure includere manualmente i sorgenti/le librerie di interesse nel progetto generato da *Snap2ino*.

Lanciare, da terminale, il comando `java -jar snap2ino.jar file.xml`, specificando il file `.xml` (esportato direttamente da Snap!) da convertire nello sketch Arduino desiderato. Così facendo si avrà la generazione, in corrispondenza del path del sorgente specificato, di una directory `src-gen/` contenente il file `.snap2ino`, derivato dall'eliminazione di tutti i tag non riconoscibili dal *parser* e utilizzato come input della successiva fase di traduzione, e di una sottodirectory contenente il progetto Wiring generato, ossia il file `.ino`, i file `.cpp` e `.h` rappresentanti gli Sprite convertiti e il file `global_vars.h` contenente le variabili globali definite in Snap!.

Una volta generato il progetto, questo può essere aperto, compilato e caricato mediante Arduino IDE, come un qualunque altro sketch Wiring.

Funzionalità

- Conversione di un *generico* micromondo/progetto Snap!, composto da un numero arbitrario di Sprite e script e da variabili sia locali che globali, in un progetto Wiring.
- Personalizzazione del mapping tra i blocchi definiti in Snap! e il codice C/C++ da eseguire al loro posto.
- Supporto ai tipi intero signed (da -32,768 a 32,767), float signed (da 3.4028235E+38 a -3.4028235E+38), booleano e stringa.
- Supporto all'esecuzione concorrente di script e Sprite e agli eventi.
- Supporto a tutti i blocchi definiti in figura 1.5.
- Supporto a nomi di variabile semplici e composti.

Limitazioni e vincoli

- Mancanza di una GUI.
- Mancato supporto alla manipolazione delle stringhe.
- Mancato controllo del tipo degli argomenti dei blocchi e segnalazione degli eventuali errori di incompatibilità , relativamente alle conversioni C/C++ effettuate, solo in fase di compilazione dello sketch.
- Mancato supporto al mapping personalizzato di un blocco (base o custom) Snap! che abbia come parametro/i uno o più Script.
- Richiesta l'installazione di Arduino IDE, delle librerie per il controllo della matrice di led (Adafruit_GFX, Adafruit_NeoMatrix, Adafruit_NeoPixel) e di quella del framework: Snap2ino_CustomBlocks.
- Richiesta l'immutabilità del tipo/utilizzo delle variabili Snap!.
- Richiesta l'inizializzazione di tutte le variabili utilizzate in Snap!, prima dell'esportazione. In particolare, nel caso di variabili condivise tra più script/Sprite si consiglia l'inizializzazione di queste nel primo script eseguito (eventualmente dedicato), analogamente a quanto avviene nel progetto Wiring.
- Richiesto il *wrapping* del corpo di un blocco custom definito in Snap! con il blocco *"esegui in modalità turbo"*.
- Richiesto il salvataggio del valore restituito da blocchi custom (mappati con funzioni) in variabili di supporto per poterlo utilizzare.

5.6 Sommario

In quest'ultimo capitolo progettuale è stato trattato il progetto del secondo prototipo del transpiler *Snap2ino* che, partendo dalle funzionalità messe a disposizione dal primo, ha visto integrarsi in questo il supporto alla conversione di un generico micromondo/progetto realizzato in Snap!, composto da un numero arbitrario di script e Sprite e da eventuali variabili sia locali che globali, e aspetti avanzati come la gestione degli eventi e l'esecuzione di task (script/Sprite) concorrenti. La versione del transpiler così raggiunta non solo è in grado di rispondere, ora, alle possibili evoluzioni e necessità future del caso di studio *Smart T-Shirt*, a partire dal quale tale progetto è nato, ma si proietta ben oltre a questo, rappresentando, unitamente all'ambiente e al linguaggio Snap!, un vero e proprio *framework* utilizzabile nella programmazione di qualsiasi sistema embedded basato su Wiring. A tal proposito, oltre alle conclusioni generali del percorso affrontato con questa tesi di laurea, saranno di seguito descritti, a partire dal contesto COGITO, gli sviluppi e gli obiettivi futuri del framework così realizzato.

Conclusioni e sviluppi futuri

In questa tesi di laurea è stato affrontato il progetto e lo sviluppo di un *transpiler* (*Snap2ino*) in grado di convertire progetti (micromondi) realizzati attraverso il linguaggio e l'ambiente Snap!, in progetti basati sul framework Wiring, con riferimento al caricamento e all'esecuzione di questi sulla nota piattaforma a microcontrollore Arduino.

Nato a partire dalle necessità sorte nel contesto del progetto *Smart T-Shirt*, per il quale sono stati trattati ampiamente, nella prima parte della tesi, il contesto applicativo (*Educational* e progetto COGITO) e quello tecnologico (sistemi embedded programmabili), *Snap2ino* è stato sviluppato attraverso la progettazione e la realizzazione di due prototipi successivi: il primo, in grado di convertire il solo *Micromondo della Matrice*, il secondo, "raffinato" dal primo attraverso un importante processo di *generalizzazione*, in grado invece di convertire un *qualsiasi* micromondo/progetto realizzato in Snap!, attraverso anche il supporto ad aspetti avanzati come la gestione degli eventi e l'esecuzione di task (script/Sprite) concorrenti. La versione attuale del transpiler, unitamente all'ambiente Snap!, fornisce un vero e proprio *framework* per lo sviluppo di qualsiasi progetto in ambito di sistemi embedded/IoT basati su Wiring, nel quale si voglia sfruttare la programmazione visuale a blocchi, con particolare riferimento al mondo *making* e degli *atelier digitali* e *creativi*.

Volendo citare nuovamente alcune delle caratteristiche peculiari rispetto a soluzioni già esistenti (p.e. Snap4Arduino), che rendono il framework realizzato potenzialmente significativo per l'intera comunità internazionale, troviamo:

- La possibilità di definire i propri blocchi su Snap! e come questi debbano essere mappati con le effettive implementazioni per il dispositivo di destinazione, consentendo la definizione del proprio "linguaggio" e del proprio "livello di astrazione", nonché di estendere e personalizzare i blocchi predefiniti supportati dal framework.

- La possibilità di utilizzare librerie e codice Wiring esistenti, e i relativi sensori e attuatori, rendendo il framework quanto più possibile aperto.
- La possibilità di utilizzare il dispositivo programmato (Arduino) senza la necessità che questo sia collegato a un pc nel quale è in esecuzione Snap!.

Sviluppi futuri Per quanto concerne gli sviluppi e gli obiettivi futuri che, a partire dal contesto COGITO, sono previsti per il *framework* così definito (*Snap! + Snap2ino*), si elencano i seguenti:

- Integrazione del supporto allo scambio di messaggi tra più magliette, nel contesto del progetto *Smart T-Shirt*.
- Integrazione del supporto a sensori di vario genere.
- Estensione del target di utilizzo, attraverso la realizzazione di kit *ad hoc*, paragonabili a LEGO Mindstorms e WeDo, composti da sistemi embedded e sensori/attuatori, più o meno specifici per il focus di quel particolare kit, *Micromondi* appositamente definiti e, immancabilmente, da *Snap2ino*.

Infine, un altro fondamentale obiettivo per il futuro è il supporto a Scratch. Relativamente a quest'ultimo, è importante riconsiderare il fatto che Snap! derivi direttamente da questo. In particolare, si nota come entrambi presentino la stessa *semantica* e come le uniche differenze siano limitate all'ordine di schedulazione/esecuzione utilizzato per gli Sprite (semplicemente invertito) e al formato utilizzato per l'esportazione dei progetti (Scratch li esporta come file `.sb2` contenenti la rappresentazione del progetto in formato `.json` anziché `.xml`). Com'è possibile quindi notare, le modifiche da apportare a *Snap2ino*, al fine di renderlo Scratch-compatibile, sono per lo più circoscritte alla sola grammatica del DSL utilizzato, rendendo quanto definito e realizzato per la sviluppo di *Snap2ino* completamente riutilizzabile anche per Scratch.

Bibliografia e sitografia

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

[2] Jeannette M. Wing. Communications of the ACM Vol. 49, No 3: *Computational thinking*. ACM, Marzo 2006.

[3] Michael Lodi, Rita Marchignoli. *EAS e pensiero computazionale. Fare coding nella scuola primaria*. La Scuola, 2016.

[4] Samsung. Smart-Coding: *Pensiero computazionale*. 2015. `http://www.smart-coding.it/wp-content/uploads/2015/02/Computational_Thinking.pdf`, ultimo accesso 14 Gennaio 2018.

[5] Laura Tarsitano. *Pensiero Computazionale e Coding nella scuola primaria: il progetto COGITO*. 2016. `http://amslaurea.unibo.it/14325/1/Pensiero%20Computazionale%20e%20Coding%20nella%20Scuola%20Primaria_Il%20Progetto%20Cogito.pdf`.

[6] Antonio Dini. *Addio a Seymour Papert, papà del Logo e pioniere della tecnologia nell'educazione*. 2 Agosto 2016. `https://www.macitynet.it/addio-seymour-papert-papa-del-logo-pioniere-della-tecnologia-nelleducazione/`, ultimo accesso 19 Gennaio 2018.

[7] Mitchel Resnick and Scratch team. Communications of the ACM Vol. 52, No 11: *Scratch: Programming for all*. ACM, Novembre 2009.

[8] Karen Brennan, Mitchel Resnick. *New frameworks for studying and assessing the development of computational thinking*. MIT Media Lab, 2012. `https://web.media.mit.edu/~kbrennan/files/Brennan_Resnick_AERA2012_CT.pdf`, ultimo accesso 14 Gennaio 2018.

[9] Teresa Magnaterra. *Il mondo del Logo.* `http://www.comune.jesi.an.it/jesicentro/TDC/DISPENSE/LOGO/storia.htm#storia`, ultimo accesso 19 Gennaio 2018.

[10] Mitchel Resnick. Communications of the ACM Vol. 36, No 7: *Behavior Construction Kits.* ACM, Luglio 1993.

[11] *About Snap!.* `http://snap.berkeley.edu/about.html`, ultimo accesso 21 Gennaio 2018.

[12] Mitchel Resnick. *Learn to code, code to learn.* EdSurge, 2013.

[13] Alessandro Ricci. Corso di Sistemi Embedded e IoT a.a 16/17, *Modulo 1.1: Introduzione ai Sistemi Embedded.*

[14] *Sistema embedded.* `https://it.wikipedia.org/wiki/Sistema_embedded`, ultimo accesso 26 Gennaio 2018.

[15] Alessandro Ricci. Corso di Sistemi Embedded e IoT a.a 16/17, *Modulo 4.1: Introduzione ai Sistemi Operativi Embedded e Real-Time.*

[16] Alessandro Ricci. Corso di Sistemi Embedded e IoT a.a 16/17, *Modulo 1.2: Dai Sistemi Embedded all'IoT.*

[17] *Internet delle Cose.* `https://it.wikipedia.org/wiki/Internet_delle_cose`, ultimo accesso 2 Febbraio 2018.

[18] Alessandro Ricci. Corso di Sistemi Embedded e IoT a.a 16/17, *Modulo 6.1: Tecnologie e Protocolli per la Comunicazione nei Sistemi Embedded.*

[19] *Getting started Arduino.* `https://www.arduino.cc/en/Guide/Introduction`, ultimo accesso 29 Gennaio 2018.

[20] *Arduino Playground, un po' di storia.* `https://playground.arduino.cc/Italiano/StoriaDiArduino`, ultimo accesso 29 Gennaio 2018.

[21] *Arduino official store: Arduino Uno Rev3.* `https://store.arduino.cc/arduino-uno-rev3`, ultimo accesso 30 Gennaio 2018.

[22] Alessandro Ricci. Corso di Sistemi Embedded e IoT a.a 16/17, *Modulo 2.1: Sistemi embedded basati su microcontrollore.*

[23] Alessandro Ricci. Corso di Sistemi Embedded e IoT a.a 16/17, *Modulo 4.2: Panoramica Raspberry Pi.*

[24] *Raspberry Pi.* https://en.wikipedia.org/wiki/Raspberry_Pi, ultimo accesso 4 Febbraio 2018.

[25] *Raspberry Pi FAQS.* https://www.raspberrypi.org/help/faqs, ultimo asccesso 4 Febbraio 2018.

[26] *What is a Raspberry Pi?* https://opensource.com/resources/raspberry-pi, ultimo accesso 4 Febbraio 2018.

[27] *Raspberry Pi: Training.* https://www.raspberrypi.org/training/, ultimo accesso 4 Febbraio 2018.

[28] *Welcome to our new website.* https://www.raspberrypi.org/blog/welcome-to-our-new-website/, ultimo accesso 4 Febbraio 2018.

[29] *Picademy.* https://www.raspberrypi.org/training/picademy/, ultimo accesso 4 Febbraio 2018.

[30] The LEGO Foundation. *Honoring Seymour Papert.* http://www.legofoundation.com/nl-nl/newsroom/articles/2016/honoring-seymour-papert, ultimo accesso 6 Febbraio 2018.

[31] Mitchel Resnick, Stephen Ocko and Seymour Papert. Children's Environments Quarterly Vol. 5, No. 4, Children and Interactive Electronic Environments: *Lego, Logo, and Design*, Abstract. Inverno 1988.

[32] *Lego education.* https://education.lego.com/en-us/coding, ultimo accesso 6 Febbraio 2018.

[33] *Lego Mindstorms.* https://it.wikipedia.org/wiki/LEGO_Mindstorms, ultimo accesso 6 Febbraio 2018

[34] *Lego Mindstorms Education EV3: User Guide.* https://le-www-live-s.legocdn.com/sc/media/files/user-guides/ev3/ev3_user_guide_us-ab5d4fc71211c1edc77a7362bab2d88a.pdf?la=en-us, ultimo accesso 6 Febbraio 2018

[35] *Lego Mindstorms Education EV3: Platform & Curriculum Compatibility.* https://education.lego.com/en-us/support/mindstorms-ev3/compatability, ultimo accesso 7 Febbraio 2018

[36] *WeDo 2.0 Education FAQS.* `https://education.lego.com/en-us/support/wedo-2/faqs`, ultimo accesso 7 Febbraio 2018

[37] *LEGO Education WeDo 2.0 Core Set.* `https://education.lego.com/en-us/products/lego-education-wedo-2-0-core-set/45300`, ultimo accesso 7 Febbraio 2018

[38] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, Agosto 2013.

[39] Brian Harvey, Jens Mönig *Snap! Reference Manual.* `https://snap.berkeley.edu/SnapManual.pdf`, ultimo accesso 24 Febbraio 2018