



**Politecnico
di Torino**

POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica

A.A. 2020/2021

Sessione di Laurea Dicembre 2021

**Dal testo al codice: uno strumento per
incoraggiare l'analisi degli esercizi di
programmazione**

Relatori

Luigi De Russis

Juan Pablo Sáenz

Fulvio Corno

Candidato

Andrea Bruno

Ringraziamenti

Ringrazio i miei relatori per il loro scrupoloso impegno e per avermi guidato con pazienza e professionalità nel processo di preparazione e stesura di questa tesi.

Un ringraziamento particolare va anche alla mia famiglia, ai miei amici e a tutte le persone che mi hanno supportato durante il percorso universitario.

Indice

Elenco delle tabelle	VI
Elenco delle figure	VII
Acronimi	X
1 Introduzione	1
1.1 Obiettivi della tesi	2
1.2 Struttura del documento	3
2 Revisione della letteratura	5
2.1 Abitudini e difficoltà degli studenti	5
2.2 Strumenti didattici per il supporto alla programmazione	7
2.3 Strumenti e IDE professionali	9
2.4 Sommario	14
3 Needfinding	16
3.1 Interrogativi	16
3.2 Metodologia	18
3.2.1 Questionario	19
3.2.2 Intervista	19
3.2.3 Modalità di svolgimento dell'osservazione	20
3.3 Analisi dei risultati	20
3.3.1 Questionari	20
3.3.2 Interviste e conclusioni	25
4 Progettazione	29
4.1 Definizione delle caratteristiche dell'applicazione	29
4.2 Alternative e scelte progettuali	32
4.3 Progetto dell'applicazione	34
4.3.1 Funzionalità principali	35

4.3.2	Funzionalità complementari	36
4.3.3	Funzionalità legate al professore	37
5	Implementazione del sistema	41
5.1	Backend	41
5.2	Frontend	41
5.2.1	Login	41
5.2.2	Home page	42
5.2.3	Creazione esercizio	43
5.2.4	Visualizzazione esercizio	44
5.2.5	Risoluzione esercizio	45
6	Valutazione	49
6.1	Pianificazione	49
6.1.1	Task	49
6.1.2	Domande post-test	51
6.2	Svolgimento dei test	52
6.3	Risultati	52
6.3.1	Task	53
6.3.2	Domande post-test	54
6.3.3	Possibili modifiche all'interfaccia	55
7	Conclusioni	57
7.1	Limitazioni	57
7.2	Sviluppi futuri	58
A	Questionario del Needfinding	59
B	Intervista del Needfinding	62
C	Questionario SUS	64
D	Script per il Test di Usabilità	65
D.1	Introduzione	65
D.2	Spiegazione del <i>think-aloud</i>	66
D.3	Task	66
D.4	Post-Task	68
	Bibliografia	69

Elenco delle tabelle

3.1	Rapporto uomo/donna dei partecipanti alle osservazioni	21
3.2	Età dei partecipanti alle osservazioni	21
3.3	Voti dei partecipanti alle osservazioni	22
6.1	Elenco dei task	50
6.2	Misure ottenute dai risultati dei task	53

Elenco delle figure

2.1	Interfaccia di Python Tutor. A sinistra viene scritto il codice, a destra sono visualizzate le strutture dati. Immagine proveniente dall'articolo di Guo [8]	9
2.2	Proposta per il plugin per Atom di Henley et al. Immagine proveniente dall'articolo di Henley et al. [10]	10
2.3	Interfaccia di PlanIt!. Immagine proveniente dall'articolo di Milliken et al. [11]	10
2.4	Interfaccia di AlgoPlan. Immagine proveniente dall'articolo di Choi et al. [12]	11
2.5	Interfaccia di Synectic. Immagine proveniente dall'articolo di Adeli et al. [13]	12
2.6	Interfaccia di Hipikat. Immagine proveniente dall'articolo di Čubranić et al. [14]	13
2.7	Interfaccia di SketchLink. Immagine proveniente dall'articolo di Baltés et al. [15]	14
3.1	Mediane delle risposte sull'analisi del testo dell'esercizio	22
3.2	Mediane delle risposte sul supporto alla comprensione del testo	23
3.3	Mediane delle risposte sulla scrittura del codice	23
3.4	Mediane delle risposte sulle difficoltà incontrate	24
3.5	Mediane delle risposte sulla risoluzione dei problemi di comprensione	25
3.6	Mediane delle risposte sull'auto-valutazione	25
3.7	Alcune proposte di interfaccia degli studenti	28
4.1	Interazioni principali all'interno dell'interfaccia	35
4.2	Wireframe dell'interfaccia prima di poter programmare	36
4.3	Wireframe dell'interfaccia completa	37
4.4	Consigli dal professore sulla decomposizione	38
4.5	Gestione dei file	38
4.6	Esecuzione del programma	39
4.7	Creazione dell'esercizio	39

4.8	Visualizzazione dell'esercizio	40
5.1	Pagina di login per studenti	42
5.2	Pagina di login per professori	42
5.3	Pagina principale per studenti	43
5.4	Pagina principale per professori	44
5.5	Pagina di creazione di un esercizio	44
5.6	Pagina di visualizzazione di un esercizio	45
5.7	Pagina di risoluzione di un esercizio, senza frammenti di codice . . .	46
5.8	Domanda a cui lo studente ha risposto	46
5.9	Pagina di risoluzione di un esercizio, modalità completa	47
5.10	Finestra per la visualizzazione e modifica dei <i>file</i> di testo	47
5.11	Finestra per la visualizzazione della soluzione proposta dal professore	48
5.12	Pagina di risoluzione di un esercizio quando viene eseguito il programma	48

Acronimi

IDE

Integrated Development Environment

CS1

Computer Science 1 (corso introduttivo di Informatica)

SLC

Smart Learning Content

MUR

Ministero dell'Università e della Ricerca

REST

Representational State Transfer

DBMS

Database Management System

SUS

System Usability Scale

Capitolo 1

Introduzione

Imparare a programmare è un percorso complesso che coinvolge sia aspetti tecnici (ad esempio l'apprendimento della sintassi e di un'ampia gamma di nozioni di informatica) che logici e metodologici (tra cui lo sviluppo delle capacità di analisi, comprensione e decomposizione dei problemi da risolvere).

Usati per apprendere e affinare tali competenze, gli esercizi di programmazione sono uno strumento fondamentale a disposizione degli studenti neofiti di informatica. Gli esercizi consistono in un testo scritto in linguaggio naturale, al cui interno viene proposto un problema da risolvere scrivendo un programma. Tramite la risoluzione di questi esercizi, lo studente non solo mette in pratica le regole di sintassi apprese dalla teoria, ma impara ad adottare buone pratiche di programmazione e strategie di *problem solving*.

Lo svolgimento di questi problemi richiede che lo studente segua una serie di passaggi, tutti importanti per raggiungere il risultato corretto e consolidare le proprie conoscenze. Il punto di partenza è la comprensione del testo dell'esercizio, in cui lo studente esamina il problema e ne estrae le richieste specifiche. Una volta fatto questo, si può passare alla traduzione delle richieste del problema (espresso in linguaggio naturale) in concetti di natura informatica. Una volta analizzato il problema e decisa la strategia da seguire, si può passare alla scrittura effettiva del programma e alla risoluzione di eventuali errori (il cosiddetto *debugging*).

Gli esercizi di programmazione sono solitamente risolti all'interno di classici IDE, programmi che forniscono un ambiente di sviluppo integrato in cui gli studenti possono scrivere il loro programma all'interno di un editor di testo (spesso dotato di funzionalità avanzate come suggerimenti per il completamento di una parola o lo spostamento rapido tra i punti nel testo in cui un certo elemento è definito e utilizzato), eseguirlo per verificarne il corretto funzionamento ed essere notificati riguardo a possibili errori di sintassi o di esecuzione del codice. Negli ultimi anni, tali strumenti hanno assunto nuove forme e un numero sempre maggiore di funzionalità, come ad esempio la visualizzazione delle strutture dati presenti nel

codice, o la verifica automatica della correttezza del programma tramite una serie di test. In molti casi questi servizi sono implementati come applicazioni web, che hanno il vantaggio di essere facilmente accessibili e di non richiedere installazioni, oltre a nascondere la complessità dell'ambiente di esecuzione del programma.

Nonostante siano fondamentali per la corretta risoluzione di un esercizio di programmazione, i passaggi preliminari di analisi e traduzione di un esercizio sono spesso ignorati o sottovalutati dagli studenti, i quali tendono a scrivere direttamente il loro programma procedendo per tentativi ed errori. Senza una corretta e completa analisi del problema, però, gli studenti si trovano a scrivere del codice senza riflettere sull'approccio corretto da seguire, producendo un risultato incompleto o di qualità più bassa.

Tale inclinazione può essere osservata anche negli IDE e nelle applicazioni interattive descritti in precedenza. Questi strumenti svolgono certamente un ruolo importante nel processo di apprendimento degli studenti, ma nella maggioranza dei casi il supporto fornito dall'applicazione si concentra sulla scrittura del codice, lasciando scoperte le fasi di analisi del testo e di passaggio da richieste in linguaggio naturale ad azioni da implementare in logica informatica. Questi passaggi, apparentemente non direttamente legati alla scrittura del codice, sono fondamentali per fare in modo che lo studente rifletta su come implementare al meglio la propria soluzione e tragga il massimo beneficio dalla sua sessione di esercitazione.

Queste considerazioni costituiscono il punto di partenza di questa tesi, in cui si prenderanno in considerazione le abitudini e le necessità degli studenti universitari che seguono corsi introduttivi di informatica.

1.1 Obiettivi della tesi

L'obiettivo della tesi è supportare gli studenti neofiti di informatica nelle fasi di analisi e decomposizione degli esercizi di programmazione.

A tale scopo sono state quindi individuate le necessità e le difficoltà incontrate dagli studenti nel compiere tali operazioni, oltre alle funzionalità fornite da applicazioni attualmente disponibili con caratteristiche rilevanti ai fini della tesi.

Riscontrato il bisogno di un'applicazione che fornisca supporto in queste fasi preliminari dello svolgimento di un esercizio, si è poi progettato, implementato e validato il prototipo di un'interfaccia implementata come applicazione web che metta in pratica le informazioni ottenute in precedenza.

Nello specifico, il lavoro svolto può essere suddiviso in quattro fasi:

Revisione della letteratura: In questa fase sono state analizzate le caratteristiche di strumenti e applicazioni per la didattica o che valorizzino l'analisi del problema da risolvere, nonché le pubblicazioni che approfondiscono l'esperienza

di risoluzione di un esercizio di programmazione da parte di uno studente di informatica.

Questa fase ha costituito la fondazione teorica per i passi successivi, esaminando i comportamenti tipici degli studenti e sottolineando l'importanza di sviluppare un'applicazione pensata appositamente per il supporto all'analisi degli esercizi.

Needfinding: Per approfondire le informazioni ottenute dalla letteratura, è stata progettata e svolta una raccolta di dati (sotto forma di questionari e interviste) per approfondire le abitudini e le problematiche che gli studenti incontrano mentre svolgono un esercizio di programmazione.

La raccolta dati ha permesso di ottenere informazioni non emerse dallo studio della letteratura, principalmente legate al flusso di lavoro seguito dagli studenti e alle motivazioni dietro alle loro scelte o preferenze.

Progettazione e implementazione: In questa fase è stata progettata e implementata un'applicazione web che ha messo in pratica le informazioni acquisite nei passi precedenti.

La progettazione dell'interfaccia, in particolare, ha permesso di tradurre i dati precedentemente ottenuti in funzionalità per l'applicazione, permettendo di arrivare a un'interfaccia usata come guida nella progettazione del prototipo.

Valutazione: Per finire l'applicazione è stata valutata tramite dei test di usabilità, per mettere alla prova le varie funzionalità offerte e individuare possibili miglioramenti futuri.

1.2 Struttura del documento

I prossimi capitoli di questo documento seguiranno il percorso logico descritto nel paragrafo precedente.

Il Capitolo 2 espone i risultati ottenuti dalla revisione della letteratura. Si parte dalle pubblicazioni che mettono in evidenza le necessità e le difficoltà degli studenti nelle prime fasi di risoluzione di un problema. Si passa poi a una panoramica degli strumenti per il supporto all'apprendimento attualmente in uso, per poi finire con l'esaminare più in generale applicazioni e IDE professionali per valutare quali soluzioni sono state adottate per supportare la comprensione del problema da affrontare.

Il Capitolo 3 descrive il processo di progettazione ed esecuzione della raccolta dati. Vengono definite le informazioni che si vogliono ottenere da questo studio, si descrivono le modalità con cui sono stati svolti questionari e interviste e si analizzano i risultati ottenuti.

Nel Capitolo 4, tutte le informazioni ottenute in precedenza vengono utilizzate per definire i requisiti principali dell'applicazione. In seguito si discute delle alternative per implementare tali requisiti, motivando le scelte finali. Il capitolo si conclude descrivendo il progetto dell'applicazione, contenente caratteristiche e funzionalità che sono state incorporate nel successivo processo di implementazione.

Il Capitolo 5 passa in rassegna le caratteristiche dell'applicazione, esplorando tecnologie, scelte implementative e soluzioni adottate.

Il Capitolo 6 è dedicato alla descrizione dei test di usabilità sull'applicazione trattata nel capitolo precedente. Per prima cosa si parla della definizione di tali test, per poi passare a come sono stati svolti e finendo con l'analizzarne i risultati.

Il Capitolo 7 serve a racchiudere le considerazioni conclusive, sottolineando gli obiettivi raggiunti e le limitazioni della tesi, e indicando possibili sviluppi futuri del progetto.

Capitolo 2

Revisione della letteratura

Il primo passo per supportare gli studenti durante la fase di analisi di un esercizio di programmazione è analizzare il modo in cui loro affrontano tali esercizi, e in particolare quali sono le criticità su cui si può intervenire e le tecniche già adottate da altri per risolvere problemi simili.

2.1 Abitudini e difficoltà degli studenti

Programmare è un processo complesso, che richiede l'applicazione di un gran numero di conoscenze e competenze affinate col tempo e l'esperienza. Non bisogna però commettere l'errore di associare l'atto del programmare alla sola scrittura del codice: esso è infatti il prodotto finale, non il processo. Se questo è vero per la programmazione in generale, vale ancora di più per la risoluzione di un esercizio di programmazione, che prevede che si analizzi a fondo il testo del problema e che si scelga il modo più appropriato per risolverlo in ogni sua parte.

L'esercizio di programmazione, quindi, è prima ancora un esercizio di *problem solving* (inteso come processo di analisi logica e di risoluzione di un problema di natura non necessariamente informatica), come riportato dall'articolo di Lishinski et al. [1] in cui è stato evidenziato l'impatto delle abilità logiche degli studenti nei corsi universitari introduttivi di informatica. Nello studio viene rilevata la correlazione tra capacità di *problem solving* e abilità nella programmazione degli studenti di corsi di CS1, mostrando come la prima possa essere un fattore predittivo della seconda tanto quanto altri parametri come la media dei voti. Nello stesso articolo si evidenzia anche la differenza tra abilità di basso livello (conoscenza della sintassi, tracciamento e spiegazione del codice) e di alto livello (effettiva scrittura del programma): le prime sono necessarie per possedere le seconde, ma non sono sufficienti, sottolineando l'importanza del *problem solving*.

Analizzando più nel dettaglio le capacità pratiche che servono per risolvere un problema di informatica, Soloway [2] sottolinea come le capacità di decomposizione, analisi e ricomposizione siano il fulcro della creazione di un programma e del processo di apprendimento. Andando avanti, Soloway spiega anche come questo processo di composizione e coordinamento dei componenti di un programma sia un problema per gli studenti principianti, più che la sintassi.

È dunque auspicabile che gli studenti utilizzino e acquisiscano familiarità con queste tecniche di analisi e decomposizione degli esercizi, ma all'atto pratico questi passaggi vengono spesso evitati.

Arvanitis et al. hanno condotto uno studio [3] in cui si è riscontrato che gli studenti tendono a non decomporre il problema da risolvere, preferendo un approccio a tentativi ed errori mirato unicamente alla scrittura del codice. Tra gli studenti osservati, solo il 35% ha applicato strategie di astrazione e decomposizione per progettare la propria soluzione al problema, mentre l'85% ha optato per un approccio *build-and-fix* (scrittura del codice senza alcuna pianificazione concettuale). Lo studio si conclude sottolineando che una volta che gli studenti adottano un approccio *build-and-fix*, hanno difficoltà ad applicare tecniche di astrazione, e spesso per loro il *problem solving* si traduce nell'eliminazione degli errori di compilazione.

In uno studio per capire quali siano le strategie di *problem solving* comunemente adottate dagli studenti di informatica, Falkner et al. [4] analizzano le strategie di auto-regolazione degli studenti. Questi ultimi, infatti, tendono a pianificare il loro processo di *problem solving* solo se strettamente necessario, rallentando così il loro processo di apprendimento. Approfondendo l'analisi quantitativa, emerge che la decomposizione del problema è praticata, ma la maggior parte delle strategie adottate dagli studenti si concentrano sui processi di sviluppo del codice (le due categorie sono state riscontrate con una frequenza del 16.24% per la decomposizione e del 54.5% per lo sviluppo del codice). Osservando le strategie infruttuose legate nello specifico alla programmazione, la maggioranza delle volte (frequenza del 57.1%) esse sono riconducibili allo scrivere il codice prima della fase di pianificazione.

In questo contesto, per poter aiutare gli studenti ad affrontare in modo più strutturato gli esercizi di programmazione, è anche utile comprendere il perché di alcune delle pratiche scorrette che adottano.

L'effetto di alcuni fattori interni che influenzano gli studenti neofiti può infatti ridurre la loro motivazione nel proseguire il corso di studi, fino a portarli ad allontanarsi dalla materia. Questi effetti sono stati documentati da Carbone et al. [5] in uno studio in cui sono state raccolte interviste semi-strutturate e descrizioni scritte da parte degli studenti. Alcuni dei risultati proposti nell'articolo permettono di approfondire problematiche già citate in precedenza. È emerso che per studenti neofiti, suddividere un esercizio in sottoproblemi è complesso in quanto non hanno un ampio repertorio di tecniche per affrontare i singoli problemi individuati, e in alcuni casi non pensano neanche alla possibilità di poter suddividere il problema

originale, a meno che non sia il professore a sollecitarli. In riferimento alla pratica generale di analizzare il testo di un esercizio prima di scrivere il codice, si fa notare che quando gli studenti percepivano di star impiegando troppo tempo per risolvere un problema (tenendo in conto anche altri loro impegni di studio), interrompevano la fase di comprensione del problema per passare a strategie che percepivano come più efficienti, come copiare blocchi di codice dagli appunti.

Allo stesso modo, lo studio di Gorson et al. [6] propone una prospettiva diversa sui fattori che influenzano gli studenti dei corsi di CS1: il loro auto-valutarsi negativamente tende a ridurre la percezione delle loro abilità nella programmazione, portando anche all'abbandono della disciplina. In particolar modo, alcune di queste auto-valutazioni negative sono dovute anche a concezioni errate delle aspettative nei loro confronti e verso le pratiche di programmazione professionale. Gli studenti, quando si valutano, utilizzano spesso dei criteri che loro considerano negativi, ma che in realtà sono buone pratiche di programmazione e comportamenti adottati da programmatori professionisti (tra cui lo spendere tempo nel pianificare la propria soluzione prima di passare alla scrittura del codice). I dati raccolti durante lo studio mostrano come il momento descritto come “*Does not understand the problem statement*” (“Non comprende il testo del problema”) sia fonte di autocritica nel 78.97% dei casi, mentre “*Stopping programming to plan*” (“Interrompere la programmazione per pianificare”) e “*Spending time palnning at the beginning*” (“Impiegare del tempo all’inizio per pianificare”) molto meno (rispettivamente 15.42% e 18.22%), ma comunque tutti e tre sono stati evidenziati da una percentuale non trascurabile di studenti. Tutti e tre i momenti sono inoltre fortemente correlati alla percezione errata che gli studenti hanno avuto nei confronti del lavoro dei professionisti: gli studenti pensano che i professionisti non compiano quegli errori o non abbiano bisogno di pianificare in anticipo il loro lavoro, e di conseguenza si giudicano negativamente se si trovano in quelle situazioni.

2.2 Strumenti didattici per il supporto alla programmazione

Gli studenti di informatica possono avvalersi del supporto di una vasta gamma di strumenti digitali pensati per affrontare temi e problematiche di natura diversa.

Lo studio portato avanti da Brusilovsky et al. [7] mostra l'importanza, la pervasività e la varietà dei contenuti interattivi per l'apprendimento (chiamati *Smart Learning Content* o SLC nell'articolo). Questo studio comprende l'analisi di strumenti e pubblicazioni relativi agli SLC, integrandola con una serie di questionari rivolti a educatori nel campo della *Computer Science*.

Gli SLC sono definiti come materiali per l'apprendimento che presentino almeno un qualche grado di interattività, e vengono classificati in base al tipo di *input*

che ricevono, al modo in cui processano i dati ricevuti e all'*output* che producono. Le applicazioni d'interesse per questa tesi, essendo relative alla risoluzione di esercizi, sono caratterizzate dal massimo livello di interattività (codice scritto dall'utente, esecuzione del programma dato in *input* e restituzione del risultato, oltre ad altre interazioni con il testo del problema), e costituiscono il 70% degli strumenti analizzati. Nello specifico, le applicazioni con caratteristiche simili a quelle proposte in questa tesi (nell'articolo indicate con "*coding*") rappresentano il 20% degli strumenti totali. Nei questionari è anche emerso il grado di integrazione nei corsi di informatica di tali applicazioni altamente interattive, suddivise in base al tipo di funzionalità offerta. In generale la maggioranza degli educatori ha riportato di utilizzarle (dimostrando quindi che questi strumenti sono ben integrati nei processi di apprendimento), e in particolare gli strumenti più in linea con gli obiettivi della tesi (indicati nei questionari come *programming tools*) sono utilizzati dal 30% degli educatori, e almeno provati dal 50%.

Gli strumenti interattivi per il supporto all'apprendimento sono dunque diffusi e utilizzati nell'ambito della didattica. Tali strumenti sono inoltre caratterizzati da obiettivi e funzionalità piuttosto diversi tra loro, e alcuni di essi sono di particolare interesse nell'ambito di questa tesi, nonostante nessuno di essi presenti tutte le caratteristiche auspiccate.

Tra le applicazioni il cui scopo principale è legato alla fase di programmazione, alcune offrono unicamente un compilatore per eseguire il programma dello studente, mentre altre partono dal codice per offrire spunti anche sull'interpretazione e comprensione del problema nel suo complesso.

Ad esempio, Python Tutor [8] (disponibile online come applicazione web [9], l'interfaccia è mostrata in Figura 2.1) permette l'esecuzione del proprio codice (nel suo complesso o una riga alla volta) e la visualizzazione in tempo reale delle strutture dati utilizzate al suo interno. Tale strumento permette allo studente di astrarsi dalla sintassi del proprio programma e di analizzare a più alto livello i dati che sta manipolando.

Un altro modo per portare uno studente a riflettere sul proprio processo risolutivo durante la scrittura di un programma è quello proposto da Henley et al. [10], nel cui articolo viene proposto un plugin per l'editor Atom (mostrato in Figura 2.2) che mira a correggere le convinzioni errate dello studente proponendo domande sul comportamento del suo codice, generando poi spiegazioni e codice che testi il corretto funzionamento di quella porzione del programma. Nell'articolo viene evidenziato come gli studenti leghino la correttezza di un programma più all'assenza di errori di compilazione (correttezza sintattica) che al suo reale comportamento (correttezza semantica): uno strumento del genere, nonostante sia comunque pensato per il supporto alla correzione del proprio programma, può portare uno studente a riflettere maggiormente sulla sua soluzione al problema.

Ugualmente, alcuni strumenti per la didattica sono stati ideati per supportare

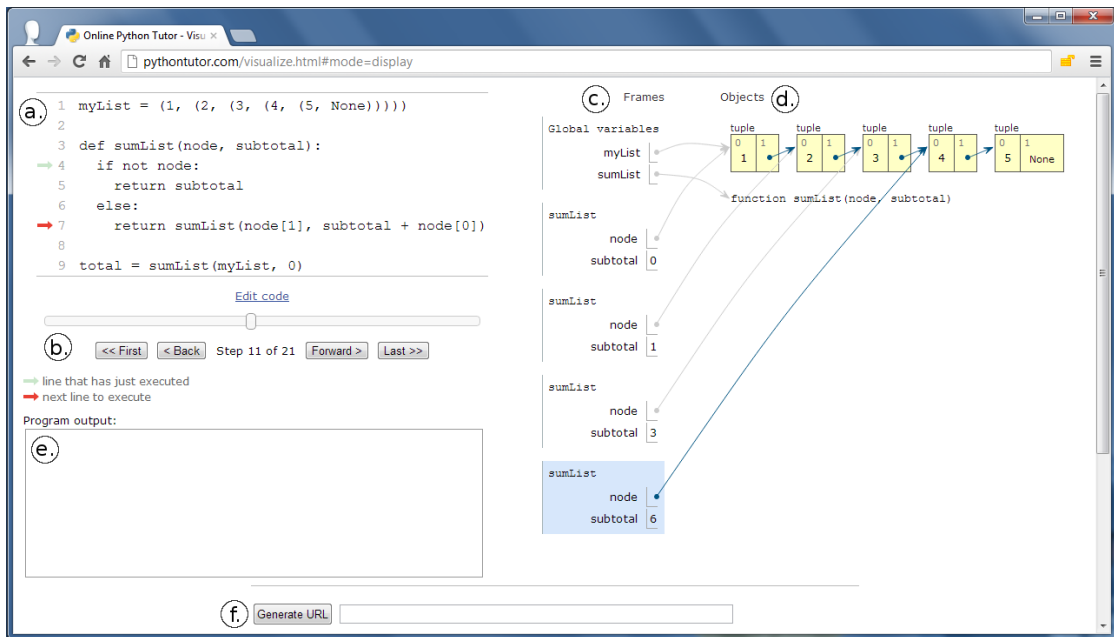


Figura 2.1: Interfaccia di Python Tutor. A sinistra viene scritto il codice, a destra sono visualizzate le strutture dati. Immagine proveniente dall'articolo di Guo [8]

gli studenti anche nella progettazione della propria soluzione.

Un esempio rilevante è PlanIt! [11], un'interfaccia che permette agli studenti di pianificare il proprio lavoro: al suo interno si ha la possibilità di arricchire il proprio progetto con una descrizione, gruppi di variabili ed elementi del sistema e una lista di obiettivi e passi da seguire per implementare la propria soluzione. L'interfaccia è mostrata in Figura 2.3.

AlgoPlan [12] è invece uno strumento che incoraggia e supporta gli studenti a strutturare il processo di *problem solving* suddividendo il loro lavoro in sottoproblemi. L'interfaccia (riportata in Figura 2.4) permette la creazione di un diagramma di sottoproblemi in cui si ha una descrizione ad alto livello dei passi da seguire per arrivare alla soluzione, partendo da uno scheletro fornito insieme al problema. Una volta completata la progettazione, lo studente può scrivere il proprio programma, associando ogni linea di codice a un punto specifico del diagramma (e quindi a uno specifico sottoproblema).

2.3 Strumenti e IDE professionali

La programmazione in ambienti professionali possiede connotazioni estremamente diverse rispetto alla risoluzione di esercizi di informatica: il numero di requisiti, il tempo e le conoscenze necessarie allo sviluppo e la natura intrinsecamente

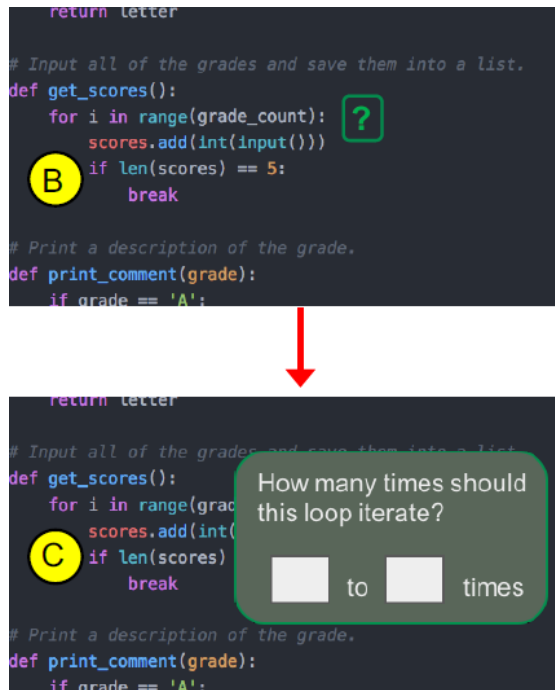


Figura 2.2: Proposta per il plugin per Atom di Henley et al. Immagine proveniente dall'articolo di Henley et al. [10]

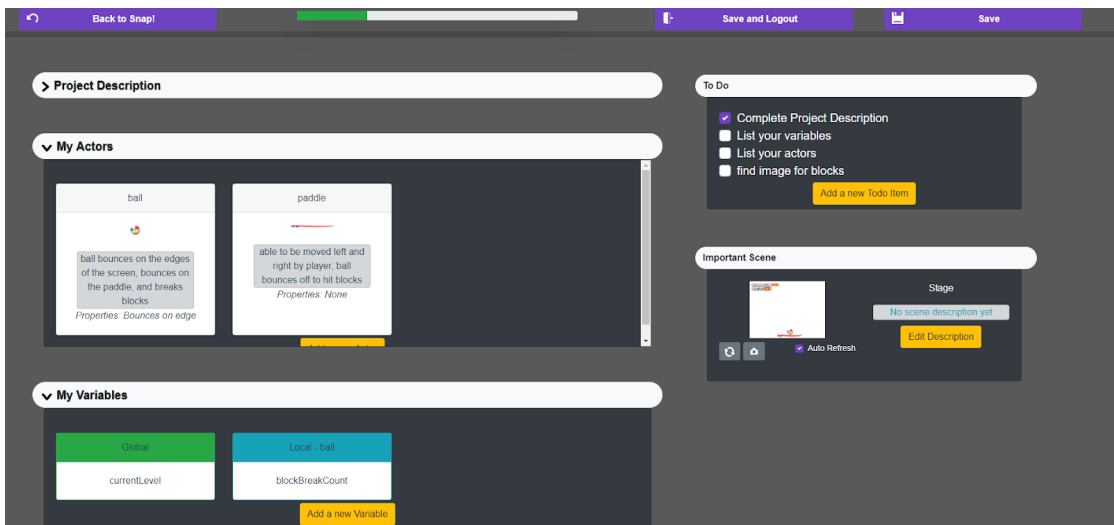


Figura 2.3: Interfaccia di PlanIt!. Immagine proveniente dall'articolo di Milliken et al. [11]

multidisciplinare e collaborativa di un progetto in un ambito lavorativo sono solo

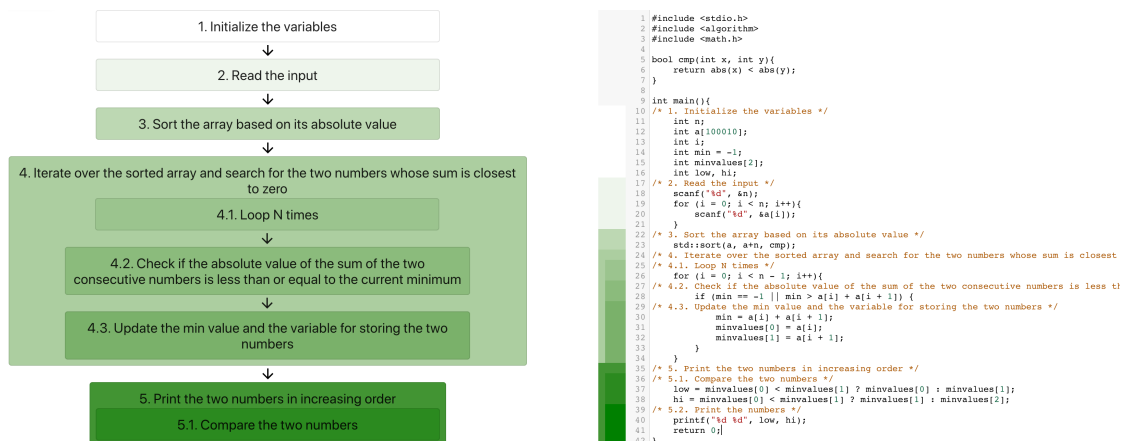


Figura 2.4: Interfaccia di AlgoPlan. Immagine proveniente dall’articolo di Choi et al. [12]

alcuni aspetti che rendono questo scenario estremamente più complesso. Alcune pratiche possono però accomunare gli studenti alle prime armi e i professionisti del settore (seppur su scale nettamente diverse), tra cui la necessità di analizzare e suddividere il proprio problema, nonché l’importanza assunta dagli artefatti esterni al codice nel processo di programmazione. Il processo di derivazione e suddivisione dei requisiti in ambito professionale segue un processo che difficilmente può essere messo in parallelo con quello usato da uno studente, ma la gestione degli artefatti può fornire informazioni utili.

Nel caso dello studente, a parte eventuali schemi o diagrammi, l’unico artefatto che non sia il programma è dato dal testo dell’esercizio, che costituisce l’insieme completo di requisiti per quel progetto in miniatura. Un progetto in ambito lavorativo prevede invece una enorme quantità di artefatti esterni al codice, come per esempio resoconti, diagrammi, documentazione, descrizioni di *bug*, messaggi e email. Nonostante queste differenze, la pratica professionale di legare artefatti di varia natura al codice può essere messa in relazione all’associazione di una sezione del testo di un problema alla rispettiva implementazione nel codice. L’esistenza di strumenti e IDE innovativi usati nell’ambito professionale a tale scopo mostra come tale pratica sia auspicabile, e l’analisi di tali applicazioni può fornire informazioni e spunti rilevanti nell’ambito di questa tesi.

In questo contesto, Synectic si propone come IDE non tradizionale che presenta sezioni di codice, documentazione e altri file legati tra loro da collegamenti e annotazioni. Nello studio di Adeli et al. [13] vengono evidenziate le motivazioni che hanno portato alla progettazione di questa interfaccia (mostrata in Figura 2.5), una descrizione delle sue caratteristiche e uno studio per verificarne la validità.

Nell’articolo si fa notare che chi è da poco entrato a far parte di un progetto



Figura 2.5: Interfaccia di Synectic. Immagine proveniente dall'articolo di Adeli et al. [13]

ha maggiore bisogno di avere a portata di mano tutte le informazioni rilevanti (non solo il codice) e i legami tra loro, mentre gli IDE tradizionali si focalizzano su combinazioni di finestre con singoli nuclei funzionali comprendenti solo codice. Tramite l'uso di Synectic si vogliono quindi coprire tre aspetti fondamentali: l'interazione con artefatti esterni al codice, prossimità spaziale di informazioni rilevanti e visualizzazione delle relazioni tra artefatti. In particolare questo strumento si

propone di migliorare l'esperienza di programmazione: mantenere modelli mentali ha un costo cognitivo, e per alleggerirlo i programmatori che usano IDE tradizionali non hanno altra scelta se non preservare informazioni e collegamenti utili nel codice o nei commenti. Lo studio finale ha inoltre valutato le prestazioni di programmatori che hanno usato Synectic per eseguire dei *task*, confrontando i risultati con un gruppo di controllo che ha usato Eclipse, un IDE tradizionale: i risultati hanno mostrato aumento dell'accuratezza, riduzione del carico cognitivo e una maggiore usabilità.

Hipikat [14] è un altro strumento che mette in risalto l'importanza degli artefatti che non siano il codice. Proposto come plugin per Eclipse, riconosce e genera automaticamente connessioni tra il codice e altri artefatti (in particolare documentazione di *versioning*, tracciamento degli *issue*, canali di comunicazione e documentazione online). In questo caso l'interfaccia, mostrata in Figura 2.6 presentata una lista di riferimenti partendo da un singolo file.

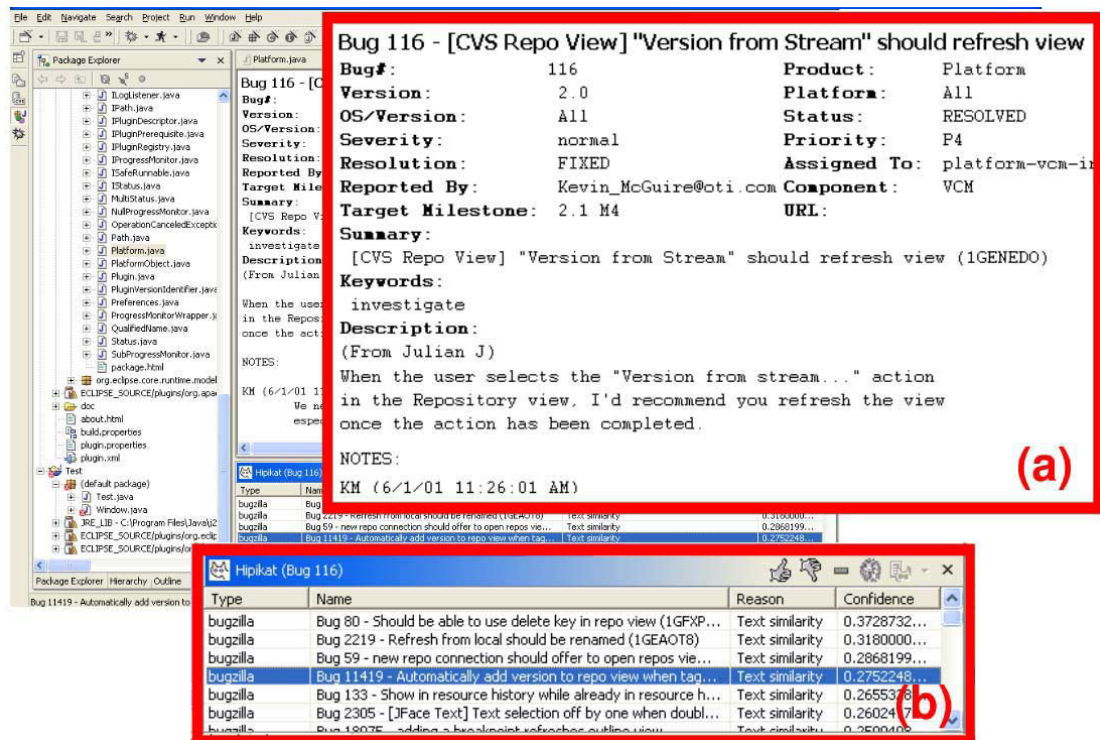


Figura 2.6: Interfaccia di Hipikat. Immagine proveniente dall'articolo di Čubranić et al. [14]

Un altro esempio di interfaccia per la gestione di artefatti e la loro connessione alle sezioni rilevanti di codice è SketchLink [15]. In questo caso gli artefatti presi in considerazione sono schemi e diagrammi prodotti dal programmatore, i quali

vengono messi in relazione con le linee di codice a cui fanno riferimento. Gli schemi possono essere fotografati tramite un'apposita applicazione web, per poi essere integrati nel codice grazie a un plugin per IntelliJ IDEA, un IDE per Java. In Figura 2.7 è mostrata l'interfaccia di SketchLink.

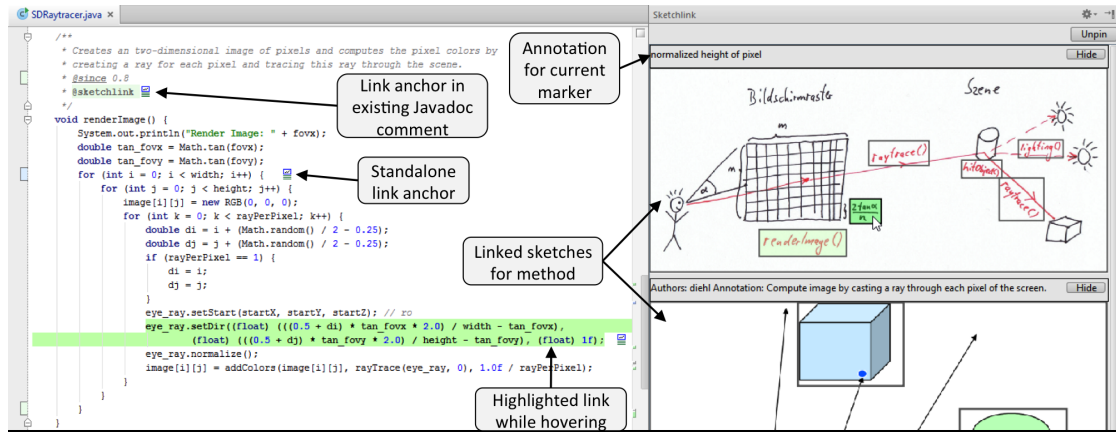


Figura 2.7: Interfaccia di SketchLink. Immagine proveniente dall'articolo di Baltes et al. [15]

2.4 Sommario

Le pubblicazioni esaminate nei paragrafi precedenti mostrano come l'analisi del testo di un esercizio e decomposizione in sottoproblemi siano pratiche fondamentali nel processo di apprendimento degli studenti dei corsi introduttivi di informatica, sottolineandone l'importanza nei processi di scrittura e di apprendimento, nonché degli effetti positivi sulla qualità dei programmi prodotti.

Gli studenti tendono però a preferire metodi meno strutturati di risoluzione degli esercizi per vari motivi. In alcuni casi gli studenti pensano che la correttezza del loro programma sia data dall'assenza di errori di compilazione, portando dunque ad applicare un approccio a tentativi ed errori. In altri casi lo studente incontra difficoltà nell'analizzare e suddividere un esercizio, oppure non prende in considerazione l'idea di poterlo fare, o ancora lo ritiene una perdita di tempo. Alcuni studenti, infine, pensano che prendersi del tempo per analizzare un esercizio sia un indicatore di una loro scarsa capacità di programmazione, una pratica negativa che non viene utilizzata dai professionisti.

È inoltre emerso che gli strumenti didattici per il supporto alla programmazione sono attualmente in uso e apprezzati. Nonostante il numero e varietà di tali applicazioni, non ne è emersa una con tutte le caratteristiche auspiccate negli obiettivi di questa tesi.

In tal senso, le applicazioni usate in ambito didattico esaminate in precedenza hanno proposto funzionalità come la visualizzazione delle strutture dati, la proposta di domande per guidare lo studente, la pianificazione del proprio lavoro e la suddivisione dell'esercizio in sottoproblemi. Gli strumenti e IDE professionali hanno invece messo in risalto in modi diversi l'importanza del collegare il proprio codice con artefatti e documenti che ne forniscano requisiti e contesto.

Alcune delle caratteristiche emerse negli strumenti esaminati (tra cui la pianificazione della soluzione, la suddivisione in sottoproblemi e la connessione tra sezioni del problema e punti specifici del codice) sono in linea con quelle ricercate nell'ambito della tesi, e ciò mostra l'interesse verso queste caratteristiche e propone spunti interessanti per le prossime fasi di progettazione.

Capitolo 3

Needfinding

La revisione della letteratura descritta nel Capitolo 2 ha evidenziato come gli studenti abbiano problemi nel passaggio dal testo al codice, e come spesso saltino la fase iniziale di comprensione e decomposizione del problema per passare alla scrittura del programma. Per ottenere informazioni più specifiche è stato condotto uno studio (composto da un questionario e un'intervista) che ha permesso di approfondire le abitudini di programmazione degli studenti e, più nello specifico, il loro approccio al *problem solving* durante lo svolgimento dei loro esercizi di informatica. Tali informazioni saranno utili al prossimo passaggio, ovvero la definizione di funzionalità specifiche per lo strumento da progettare.

3.1 Interrogativi

Gli obiettivi principali di questa osservazione sono:

- individuare motivazioni per l'uso o meno di approcci strutturati agli esercizi
- individuare strategie comunemente adottate per l'analisi e comprensione dei testi
- individuare strategie comunemente adottate per la traduzione dell'esercizio in codice
- individuare problematiche e difficoltà comuni nell'analisi e traduzione del testo, e strategie comuni per risolverle
- esaminare relazioni tra i risultati precedenti e la valutazione (sia oggettiva che auto-valutazione) delle prestazioni dello studente

In particolare, lo studio si propone di ripercorrere il processo di risoluzione di un esercizio di programmazione, approfondendo tre momenti principali:

La fase iniziale, in cui lo studente considera con che approccio affrontare il problema. In questa fase ci si chiede come mai gli studenti tendano a non adottare una strategia di decomposizione del problema:

- la considerano una pratica negativa, che denota scarsa abilità nella programmazione?
- non pensano che sia utile, preferendo un approccio a tentativi ed errori?
- hanno troppa fretta di svolgere l'esercizio, e quindi considerano inutile spendere tempo sull'analisi?
- non considerano la possibilità di analizzare il problema in questo modo?
- hanno difficoltà nell'analisi e decomposizione del testo? E in questo caso, la difficoltà è dovuta:
 - alla comprensione del testo?
 - all'individuazione di sezioni importanti o parole chiave?
 - al mappare concetti di programmazione a richieste in linguaggio naturale?

Rispondere a queste domande permetterà di approfondire le necessità degli studenti: se dovessero emergere problemi specifici, essi dovranno essere considerati nello sviluppo; se il problema principale è la bassa propensione alla pianificazione, la presenza stessa dell'interfaccia potrebbe risultare uno stimolo per lo studente ad adottare pratiche di analisi e decomposizione.

Lo svolgimento, in cui lo studente comprende il testo e lo traduce in codice. In particolare, ci si chiede come gli studenti analizzino e decompongano un problema:

- annotano o evidenziano il testo del problema? Se sì, come?
- affiancano al testo altri tipi di artefatti? Se sì, quali?
 - diagrammi di flusso
 - schemi logici (ad esempio simili a diagrammi entità-relazione)
 - rappresentazione di strutture dati
 - rappresentazione degli stati del sistema partendo da specifici input
- annotano il codice da loro scritto? Se sì, come?
 - se annotano anche il testo, aggiungono dei riferimenti alla relativa annotazione sull'esercizio?
- dividono prima tutto il problema, o scrivono il codice di una sezione appena la individuano?

Comprendendo le pratiche comuni di analisi e risoluzione di un esercizio, si può creare un'interfaccia che segua in modo più naturale il meccanismo di risoluzione usato comunemente dagli studenti.

La verifica, in cui lo studente affronta dubbi ed errori e confronta la propria soluzione con il professore o altri studenti. In questo caso ci si chiede come lo studente preferisca agire durante la fase di analisi dell'esercizio (ignorando dunque eventuali problemi nel codice o *debugging*):

- riflettendo autonomamente?
- cercando online?
 - solo per dubbi sintattici, o anche su strategie di risoluzione?
- confrontandosi con altri studenti?
 - riguardo al testo?
 - confrontando il codice?
- confrontandosi con professore o esercitatore?
 - per ricevere aiuto nella suddivisione (completa o solo in parte)?
 - per ricevere aiuto nella scelta di strutture dati, algoritmi o approcci da utilizzare?
 - per fare un confronto con esempi di input e relativi output desiderati?

In caso di problematiche specifiche, ci si può interrogare su tecniche di validazione della soluzione proposta dallo studente, o strumenti di assistenza.

Potrebbe inoltre emergere l'utilità di integrare elementi di altri editor educativi, come per esempio l'interazione con altri studenti, la presentazione di correzioni o guide verso la soluzione corretta o l'utilizzo di input di prova per verificare la soluzione proposta.

La visualizzazione delle strutture dati, seppur pensata per la risoluzione di dubbi all'interno del codice del programma, è anche legata alla fase di analisi e pianificazione dell'esercizio, dunque può essere presa in considerazione giudicando i risultati delle osservazioni nel loro complesso.

3.2 Metodologia

Questo studio è stato condotto con un approccio misto, in cui a ogni studente è stato chiesto di compilare un questionario e di rispondere a una serie di domande.

Dato che uno degli obiettivi di questo studio è comprendere quali sono le strategie e abitudini degli studenti per la risoluzione di un problema, si sarebbe potuto optare anche per l'osservare gli studenti mentre risolvono un esercizio di programmazione.

Questo metodo non avrebbe però permesso di ottenere dati su tutti gli aspetti e le fasi dello svolgimento di un problema a causa dell'intrinseca variabilità che avrebbe introdotto. Uno dei problemi dovuti a tale metodo è che la difficoltà di un esercizio è in larga parte soggettiva, e dunque uno stesso problema potrebbe risultare semplice per qualcuno (e dunque non richiedere particolari strategie di analisi per essere risolto anche correttamente), mentre potrebbe essere molto complesso per qualcun altro (inducendo stress nello studente e alterando o bloccando il suo processo di *problem solving*). Un altro fattore da considerare è che anche la scelta di un esercizio introduce variabilità: a seconda della tipologia del problema scelto, uno studente potrebbe scegliere di adoperare solo alcune delle strategie che impiega di solito (ad esempio, un esercizio che si focalizza sulle condizioni logiche potrebbe non permettere di valutare come uno studente affronta la scelta delle strutture dati).

3.2.1 Questionario

I dati raccolti dallo studio sono di natura qualitativa, quindi non si prevede il ricorso ad analisi statistiche per estrarre informazioni dai risultati ottenuti. Il questionario iniziale è dunque servito per porre una serie di domande in maniera standardizzata e al di fuori dell'intervista, in modo tale da non allungarla troppo con domande a risposta chiusa.

Le sezioni introduttive del questionario (sesso, età, voto dell'esame di informatica) è servita a valutare la rappresentatività degli studenti intervistati (seppure, come detto precedentemente, in maniera qualitativa).

La prima domanda effettiva ha permesso di contestualizzare le risposte al questionario in base allo stile di risoluzione degli esercizi adottato dallo studente, cioè il suo analizzare o meno il testo del problema prima di scrivere il programma.

Le domande successive hanno previsto gruppi di affermazioni da valutare su una scala Likert a cinque livelli, e in alcuni casi è stata aggiunta una domanda a risposta aperta per permettere ulteriori commenti.

Il testo completo del questionario è riportato nell'Appendice A.

3.2.2 Intervista

Le domande dell'intervista sono state pensate per essere di natura più generale rispetto a quelle del questionario, in modo tale da portare lo studente a discutere più nel dettaglio le sue risposte.

La maggior parte delle domande prevede dei possibili *follow-up* (quesiti aggiuntivi) per approfondire il discorso portato avanti dallo studente, e dipendono dalla risposta data alla specifica domanda.

Nell'ultima domanda viene chiesto allo studente di immaginare e disegnare un'interfaccia che pensa lo aiuterebbe a risolvere esercizi di informatica. Questa domanda è pensata per capire più in dettaglio il loro processo di risoluzione e quali elementi di una futura interfaccia sarebbero a loro più familiari.

In generale, data la natura discorsiva delle interviste, le domande possono essere alterate, aggiunte o saltate in caso di risposte al questionario o all'intervista stessa che meritano di essere approfondite.

Il testo completo dell'intervista è riportato nell'Appendice B.

3.2.3 Modalità di svolgimento dell'osservazione

Sono stati contattati dieci studenti al primo anno di laurea triennale del Politecnico di Torino, con il requisito di aver superato l'esame del corso di Informatica. Dato che il corso non prevede la registrazione della frequenza a lezioni o laboratori, tale restrizione è servita per assicurarsi che lo studente avesse almeno familiarità con la programmazione. I partecipanti sono stati selezionati tra gli studenti del corso di Informatica tenuto dal professore Corno.

Le osservazioni si sono svolte in modalità remota. Per prima cosa agli studenti è stato chiesto di compilare il questionario. Il questionario è stato fornito sotto forma di link verso un documento online, contenente come domanda iniziale l'accettazione del modulo di consenso informato fornito quando lo studente è stato contattato per lo studio. Una volta terminata la compilazione, gli studenti hanno risposto alle domande dell'intervista. Poiché l'ultima domanda prevedeva la rappresentazione dello schizzo di un'interfaccia, è stato chiesto agli studenti di mostrare il loro disegno su carta tramite la propria telecamera. L'immagine è stata poi salvata tramite *screenshot*. In alcuni casi, quando preferito dallo studente, è stato prodotto direttamente uno schema disegnato a mano su un editor di disegno digitale.

3.3 Analisi dei risultati

3.3.1 Questionari

Domande iniziali: Età, sesso e voto all'esame di Informatica degli studenti che hanno risposto al questionario sono risultati ragionevolmente nella media e senza differenze apprezzabili tra chi ha riportato di analizzare il testo degli esercizi e chi ha invece detto di iniziare a scrivere il programma.

Il rapporto uomo/donna (Tabella 3.1) è stato di 80/20, rispetto al 69.2/30.8 riportato dal MUR [16] per le immatricolazioni al Politecnico di Torino nel 2020/21.

L'età media (Tabella 3.2) dei partecipanti è risultata essere di 19.3 anni, rispetto ai 19.5 anni riportati da AlmaLaurea [17] per gli studenti laureati nel 2020 (vengono citati un'età media di laurea di 23.7 anni e una durata media degli studi di 4.2 anni)

Il voto medio all'esame di Informatica (Tabella 3.3) è stato 27.7, rispetto alla media di 25.9 dei voti d'esame per il corso del professore Corno per l'anno 2020/21 [18]. La valutazione "30 e Lode" è stata considerata come 30 nel calcolo della media.

	Uomini	Donne
Chi inizia a scrivere	80%	20%
Chi analizza il testo	80%	20%
Complessivo	80%	20%

Tabella 3.1: Rapporto uomo/donna dei partecipanti alle osservazioni

	Media (anni)	Deviazione standard	Massimo	Minimo
Chi inizia a scrivere	19.6	0.55	20	19
Chi analizza il testo	19	0	19	19
Complessivo	19.3	0.48	20	19

Tabella 3.2: Età dei partecipanti alle osservazioni

Approccio generale: Il 50% degli studenti analizza l'esercizio prima di risolverlo, il 50% scrive direttamente il codice del programma.

Analisi del testo dell'esercizio: Indipendentemente dall'approccio generale, la maggior parte degli studenti effettua la lettura di tutto il testo dell'esercizio prima di iniziare.

Chi analizza il problema tende anche a prendere appunti sul testo, al contrario di chi inizia col codice.

Non è emerso alcun pattern legato agli appunti nei commenti del codice. Indipendentemente dall'approccio generale, gli studenti hanno fornito sia risposte positive che negative.

Chi analizza il problema tende a suddividere l'esercizio in sottoproblemi, mentre chi inizia scrivendo il codice ha fornito opinioni miste.

Le mediane delle risposte fornite dagli studenti sono riportate nella Figura 3.1

	Media	Deviazione standard	Massimo	Minimo
Chi inizia a scrivere	28	2.12	30L	25
Chi analizza il testo	27.4	2.97	30L	23
Complessivo	27.7	2.45	30L	23

Tabella 3.3: Voti dei partecipanti alle osservazioni

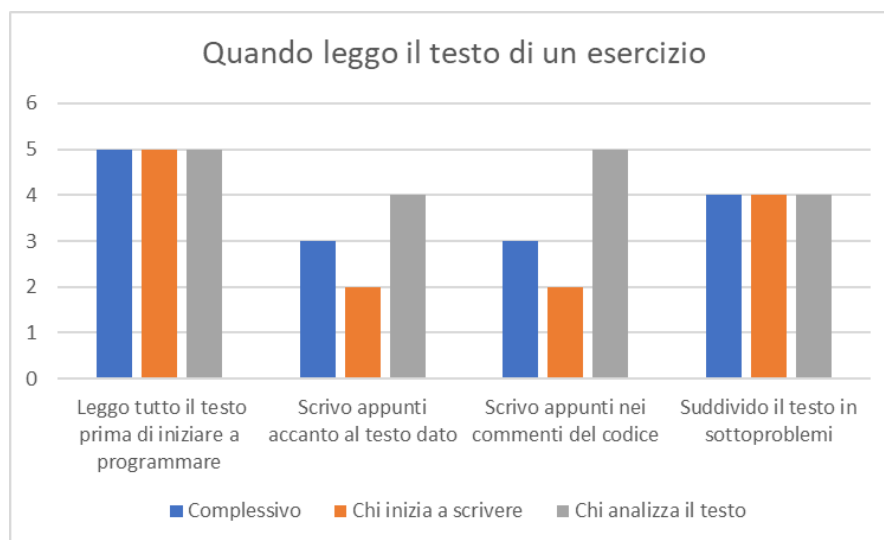


Figura 3.1: Mediane delle risposte sull'analisi del testo dell'esercizio

Supporto alla comprensione del testo: I diagrammi di flusso non sono mai utilizzati.

La rappresentazione delle strutture dati è invece apprezzata da chi analizza il testo dei problemi, mentre chi inizia col codice ha riportato di non utilizzare queste rappresentazioni.

Non è emerso alcun pattern legato al tenere traccia delle variabili simulando a mano l'esecuzione del codice. Indipendentemente dall'approccio generale, gli studenti hanno fornito sia risposte positive che negative.

Le mediane delle risposte fornite dagli studenti sono riportate nella Figura 3.2

Scrittura del codice: Gli studenti ritornano spesso a leggere il testo dell'esercizio, indipendentemente dall'approccio generale che utilizzano per risolvere gli esercizi.

Chi analizza il testo del problema tende anche a ritornare spesso a consultare gli appunti che ha scritto, al contrario di chi inizia scrivendo il codice.

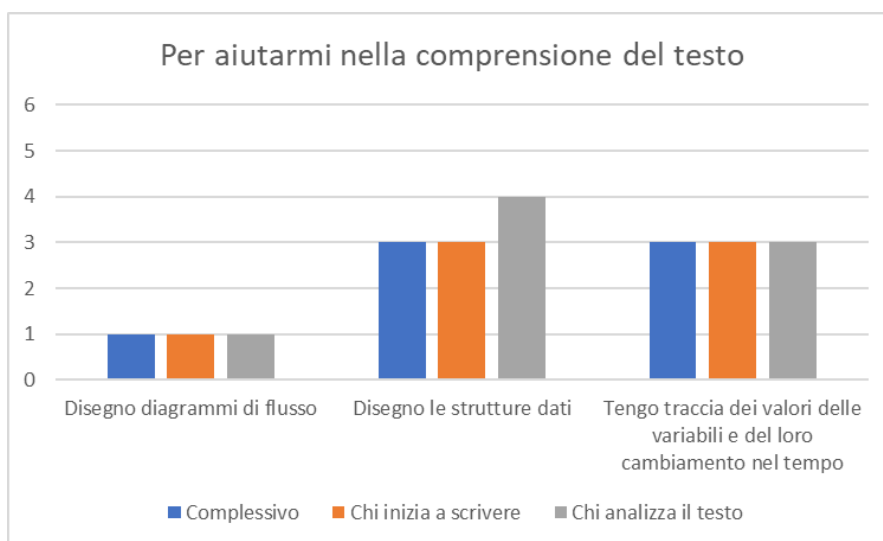


Figura 3.2: Mediane delle risposte sul supporto alla comprensione del testo

Al contrario, nessuno studente tende a consultare nuovamente i diagrammi che ha disegnato in precedenza.

Gli studenti hanno infine riportato una tendenza ad alternare la lettura di una sezione del testo e la scrittura del relativo codice.

Le mediane delle risposte fornite dagli studenti sono riportate nella Figura 3.3

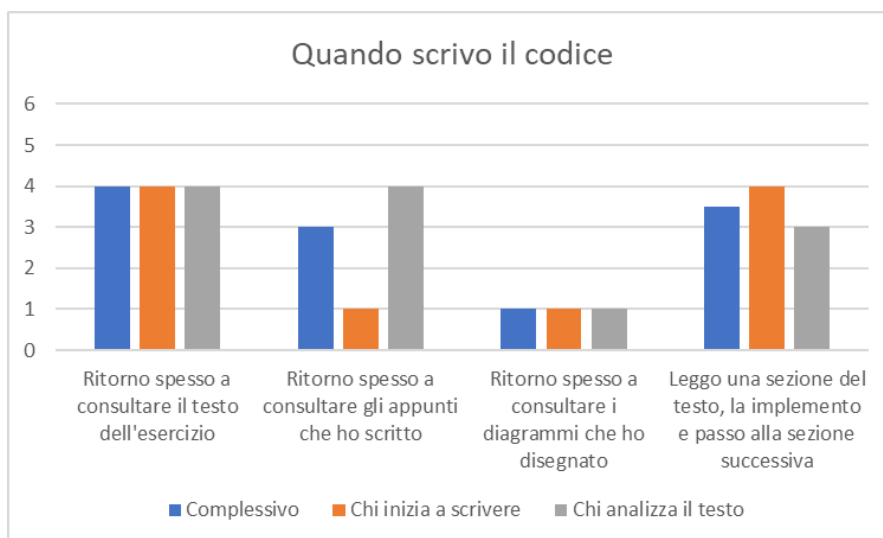


Figura 3.3: Mediane delle risposte sulla scrittura del codice

Difficoltà: In questo caso le risposte al questionario sono state poco significative, in quanto tutte le risposte tendono verso il negativo. Le risposte all'intervista su questo stesso argomento chiariscono quale sia la tendenza degli studenti.

Le mediane delle risposte fornite dagli studenti sono riportate nella Figura 3.4

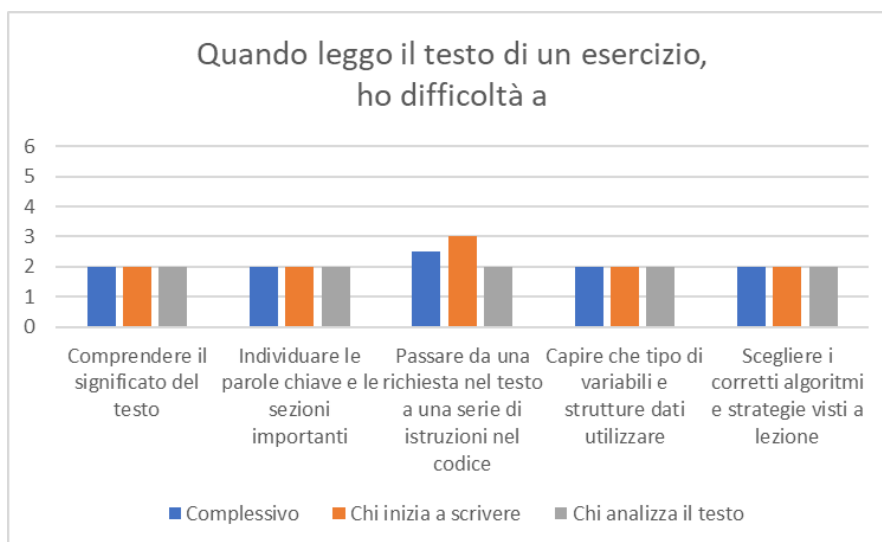


Figura 3.4: Mediane delle risposte sulle difficoltà incontrate

Risoluzione dei problemi di comprensione: Gli studenti tendono a non cercare online per chiarire i loro dubbi di comprensione.

Gli studenti tendono invece a consultare slide e/o appunti, questo viene fatto principalmente da chi analizza il testo dell'esercizio.

Il confronto con i colleghi è un altro approccio utilizzato, anche in questo caso, principalmente da chi analizza il testo dell'esercizio.

Chi analizza il problema tende anche a chiedere aiuto a professori o esercitatori, mentre chi inizia col codice ha spesso riportato di non consultarsi con loro.

Le mediane delle risposte fornite dagli studenti sono riportate nella Figura 3.5

Auto-valutazione: In questo caso le risposte al questionario sono state poco significative, in quanto tutte le risposte tendono verso il positivo. Le risposte all'intervista su questo stesso argomento chiariscono quale sia la tendenza degli studenti.

Le mediane delle risposte fornite dagli studenti sono riportate nella Figura 3.6

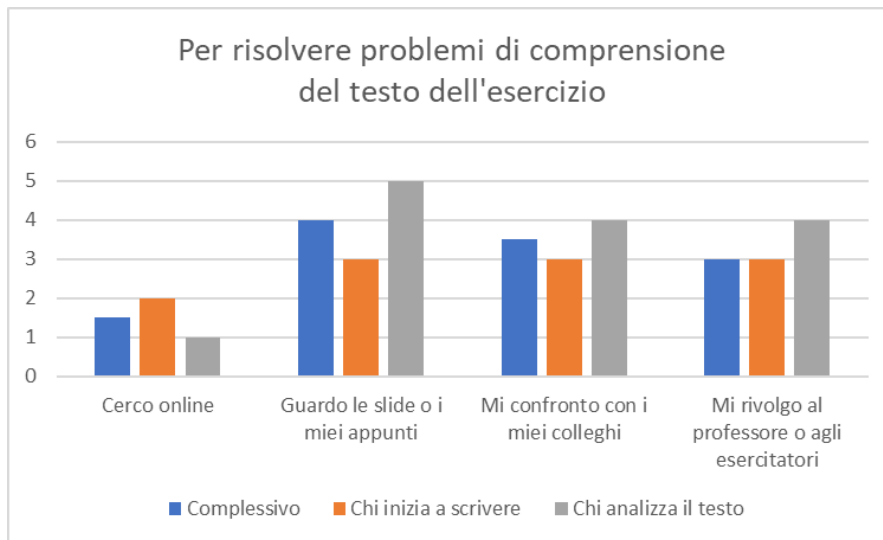


Figura 3.5: Mediane delle risposte sulla risoluzione dei problemi di comprensione

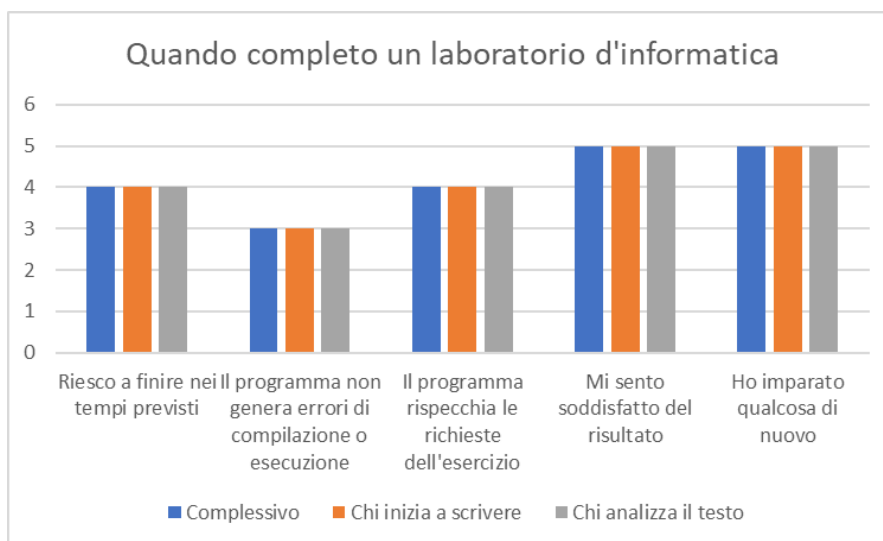


Figura 3.6: Mediane delle risposte sull'auto-valutazione

3.3.2 Interviste e conclusioni

I risultati dei questionari, insieme alle risposte date dagli studenti durante le interviste, hanno portato alle seguenti conclusioni:

Approccio generale: Il 50% degli studenti intervistati tende a scrivere il codice senza analizzare, almeno in parte, il testo dell'esercizio. In alcuni casi l'approccio utilizzato è misto, ma comunque senza una precisa pianificazione. Questo

è in linea con la letteratura, in quanto il fenomeno è considerato uno dei vari problemi che si possono riscontrare, ma non quello prevalente.

Analisi del testo dell'esercizio: Gli studenti, in particolar modo quelli che analizzano il testo del problema prima di programmare, tendono a preferire annotazioni sul testo dell'esercizio, piuttosto che nel codice. Gli studenti non usano il codice per tenere traccia del loro processo di *problem solving*: commentano il codice principalmente perché suggerito dal professore e per agevolare la correzione del codice da parte sua, oppure per tenere a mente il ruolo di funzioni o strutture dati più complesse.

Le annotazioni sul testo del problema tendono a essere di tipo grafico (sottolineature o suddivisioni) piuttosto che appunti in forma scritta. Alcuni studenti hanno detto che non annotano il testo del laboratorio in quanto preferiscono leggerlo da computer.

Gli studenti che analizzano con attenzione il testo tendono anche a suddividerlo in sottoproblemi, mentre chi preferisce partire scrivendo il codice salta del tutto o in parte questo passaggio, oppure lo fa a mente.

Supporto alla comprensione del testo: Gli studenti intervistati hanno riportato vari tipi di annotazioni e schemi esterni al testo dell'esercizio, anche se nessuno di questi è stato menzionato con una frequenza nettamente superiore agli altri. Tra questi schemi è interessante citare la rappresentazione delle strutture dati e lo pseudocodice, mentre nessuno utilizza i diagrammi di flusso. Questi schemi, come evidenziato dalle domande successive, tendono comunque a non essere consultati dopo essere stati scritti.

Scrittura del codice: Durante la scrittura del codice, la maggior parte degli studenti torna spesso sul testo dell'esercizio, mentre è più raro che torni spesso a consultare schemi o appunti che ha scritto, in quanto vengono usati principalmente per chiarire le idee in caso di dubbi su come procedere.

Sul questionario è emersa una tendenza ad alternare la lettura di una sezione del testo con la scrittura della relativa parte del programma. Questa tendenza è stata confermata nelle interviste dagli studenti che partono a scrivere il programma, mentre la maggior parte degli studenti che analizzano il testo hanno riportato di non alternare analisi del testo e scrittura del codice. Probabilmente il contesto della conversazione ha chiarito meglio la domanda stessa.

Gli studenti che preferiscono partire a scrivere il programma hanno espresso una preferenza verso questo metodo di analisi e scrittura alternata. Alcuni studenti hanno però detto di non essere pienamente soddisfatti da questo metodo, in quanto fa perdere di vista il contesto generale del problema. Uno

studente ha anche raccontato che questo è stato un problema quando ha sostenuto l'esame di Informatica: dopo la prima metà dell'esame si è accorto di aver usato un approccio sbagliato e ha dovuto ricominciare da zero.

Difficoltà: Sia dai questionari che dalle interviste non è emersa una difficoltà specifica e comune a un gran numero di studenti. In generale, alcuni studenti hanno detto che i loro dubbi e difficoltà sono solitamente legati alla scelta di algoritmi e strutture dati. Questi studenti pensano che tale difficoltà sia dovuta principalmente a mancanza di esperienza o esercitazione, e in un caso alla mancanza di familiarità con la teoria. In due casi le difficoltà erano legate alla comprensione della richiesta nel testo, ma uno dei due studenti ha anche detto che di solito questo problema è risolto dagli esempi fatti nel testo stesso.

Risoluzione dei problemi di comprensione: Gli studenti tendono a chiedere aiuto a colleghi o al professore, raramente online e non consultano molto spesso le slide. Chi si confronta con altri studenti, di solito lo fa per *debug* o risoluzione di problemi nel codice. Chi chiede aiuto a professore o esercitatori cerca solitamente suggerimenti o consigli su come impostare il programma.

Pro e contro dei metodi di risoluzione: Per gli studenti che analizzano approfonditamente il testo i vantaggi sono la visione d'insieme, il processo logico chiaro e la bassa probabilità di dover tornare indietro a riscrivere il programma; lo svantaggio principale è che la fase di analisi porta via del tempo. In alcuni casi questi studenti hanno detto di aver provato metodi più basati su tentativi ed errori, ma hanno trovato il procedimento meno chiaro e hanno scritto programmi di qualità più bassa.

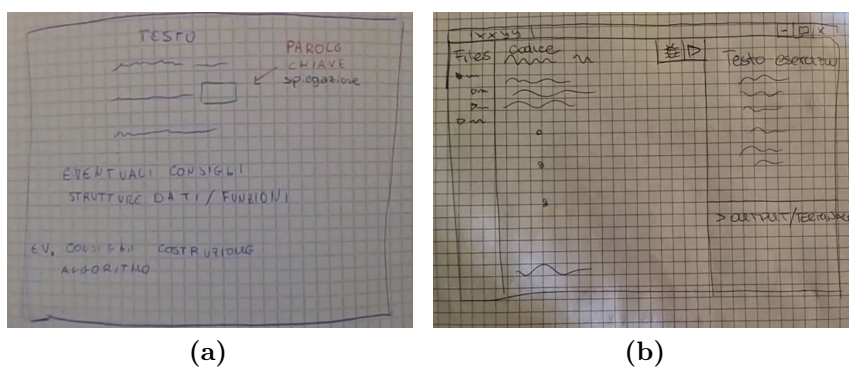
Per chi parte scrivendo il programma, i vantaggi sono la flessibilità e la velocità (se il codice di partenza risulta corretto), mentre gli svantaggi comprendono la tendenza a rimanere bloccati in soluzioni più complesse del necessario e il rischio di dover ripensare e implementare nuovamente la struttura del programma. Questi studenti non hanno provato altri approcci alla risoluzione dei laboratori.

Proposte di interfaccia: Le interfacce immaginate dagli studenti presentano caratteristiche ricorrenti, tra cui:

- testo a fianco del codice per una visione sinottica: alcuni studenti hanno detto di aver già lavorato in modo simile, ma in modi artificiosi come avere due finestre aperte contemporaneamente, o copiando il testo del laboratorio nei commenti del programma
- supporto alla scelta delle strutture dati, ad esempio tramite consigli o analisi del codice

- supporto generale alla programmazione, ad esempio con editor di codice con colorazione della sintassi, indentazione assistita, bottoni grandi e semplici, documentazione a schermo relativa alla sintassi/funzione che si sta usando

Nella Figura 3.7 sono riportati tre degli schizzi proposti dagli studenti. Questi disegni forniscono un esempio delle caratteristiche che gli studenti si aspetterebbero da un'interfaccia che li aiuti nell'analisi e svolgimento di un esercizio di programmazione, e in generale riflettono i risultati ottenuti nell'arco di tutta l'osservazione. In particolare, gli schizzi 3.7b e 3.7c mostrano esempi di visualizzazione del testo del problema a fianco del codice, mentre lo schizzo 3.7a sottolinea l'interesse dello studente verso funzionalità di ricerca delle parole chiave nel testo e di supporto nella scelta di strutture dati e algoritmi da utilizzare.



(a)

(b)



(c)

Figura 3.7: Alcune proposte di interfaccia degli studenti

Capitolo 4

Progettazione

4.1 Definizione delle caratteristiche dell'applicazione

Le informazioni ottenute dalla letteratura e da questionari e interviste fatti a studenti di informatica (descritti rispettivamente nei Capitoli 2 e 3) hanno contribuito a definire una serie di caratteristiche che verranno utilizzate come punto di partenza per la progettazione di un'applicazione per il supporto all'analisi degli esercizi di programmazione:

Suddivisione del testo in sottoproblemi: I questionari hanno mostrato che la suddivisione in sottoproblemi è praticata dagli studenti, e in particolare da quelli che analizzano il testo del problema. Tale tendenza è stata confermata dalle interviste, in cui tre studenti hanno detto di suddividere il problema.

Questa pratica trova riscontro anche nella letteratura: come già evidenziato da più pubblicazioni [2, 3, 4, 5], è auspicabile che gli studenti decompongano l'esercizio: se non incoraggiati, tendono a sottovalutare l'importanza di tale passaggio e a programmare in modo non strutturato.

Avere il testo del problema accanto al codice: Sia i questionari che le interviste hanno rilevato che la maggior parte degli studenti torna spesso a rileggere il testo dell'esercizio, e nella maggior parte dei casi le interfacce immaginate dagli studenti prevedevano un qualche tipo di visualizzazione sinottica del testo dell'esercizio e del codice.

Tale approccio è stato esaminato anche durante la revisione della letteratura attraverso l'IDE Synectic [13] (usato in ambito professionale), che utilizza una visualizzazione combinata del codice e dei relativi artefatti.

Analisi del problema prima di iniziare a scrivere il programma: Le interviste agli studenti hanno suggerito che sarebbe opportuno prevedere una fase di analisi dell'intero problema prima di iniziare a scrivere il programma, per evitare che lo studente legga una parte del testo, scriva il codice per la sua risoluzione e passi a leggere la parte successiva del problema.

Gli studenti che utilizzano strategie di analisi e decomposizione del problema tendono ad analizzare il testo dell'esercizio una volta e all'inizio, per evitare di dover tornare indietro a correggere eventuali problemi di logica durante la scrittura del programma. Questi studenti hanno anche riportato di aver provato a svolgere degli esercizi usando un approccio senza analisi preliminare, ma hanno trovato più difficile capire come risolvere il problema e una qualità inferiore della loro soluzione.

Chi invece non utilizza strategie di analisi tende ad alternare la lettura di una sezione del testo e la scrittura della porzione di programma che la implementa, prima di leggere e capire le richieste successive del problema. Questi studenti hanno spesso riferito che in molti casi sono costretti a cambiare gran parte dell'implementazione della loro soluzione perché durante il suo sviluppo si rendono conto di non aver compreso a fondo il problema.

Gli articoli di Falkner et al. [4] e di Corson et al. [6], esaminati nella revisione della letteratura del Capitolo 2, hanno inoltre evidenziato l'importanza di analizzare a fondo il problema prima di iniziare a programmare, sottolineando come tale pratica sia spesso sottovalutata dagli studenti.

Processo di analisi relativamente veloce: Le interviste hanno mostrato come il tempo passato ad analizzare il testo del problema sia un fattore preso in considerazione dagli studenti nella scelta del loro approccio al *problem solving*.

Chi tende a non analizzare e decomporre il problema indica la velocità come uno dei vantaggi, a patto che l'implementazione scelta sia corretta (se invece hanno sbagliato approccio al problema, devono spendere molto tempo a re-implementare la soluzione). Gli studenti che analizzano il testo dell'esercizio considerano invece il tempo impiegato come un fattore negativo.

Nel Capitolo 2, l'articolo di Carbone et al. [5] ha sottolineato l'effetto negativo del tempo impiegato per risolvere un problema di informatica: se lo studente percepisce di starne impiegando troppo, rischia di attuare strategie non strutturate e meno efficaci.

Supporto nella scelta di algoritmi e strutture dati: Questionari e interviste non hanno riscontrato difficoltà prevalenti tra gli studenti coinvolti nello studio. Tra le problematiche menzionate, la più citata è stata la scelta di algoritmi e strutture dati, dato corroborato dalle interfacce proposte dagli studenti.

Tale difficoltà trova riscontro anche nella letteratura. Lo studio di Lahtinen et al. [19] mostra come la comprensione delle strutture dati sia complessa per gli studenti, anche se meno rispetto a fattori come la strutturazione del problema e la sua decomposizione. È inoltre emersa una forte correlazione tra questi tre fattori: lo studente impara tutte queste cose facilmente, o ha problemi con tutte.

Smith et al. [20] riportano le conclusioni tratte dal loro studio su un gruppo di studenti che hanno svolto una serie di esercizi giornalieri: il 53% dei partecipanti ha riscontrato come problema principale lo sviluppo di algoritmi o rappresentazione di dati, dato nettamente superiore rispetto ad altre difficoltà legate alla sintassi o al *debugging* (rispettivamente emersi nel 14% e 10% dei casi).

Annotazioni sul testo del problema: Gli studenti che analizzano il testo del problema tendono anche ad annotarlo, a differenza di chi inizia scrivendo il programma. Gli studenti hanno inoltre riferito di preferire annotazioni di tipo grafico come le sottolineature, piuttosto che commenti di tipo testuale. Tali annotazioni sono utilizzate per evidenziare parole chiave o passaggi importanti all'interno del testo.

Supporto da parte del professore: Durante le interviste, gli studenti hanno detto di risolvere le loro difficoltà ricorrendo all'aiuto di slide, altri studenti o professori. Se i primi due sono spesso consultati per risolvere dubbi legati al codice o al *debugging*, professori ed esercitatori sono chiamati in causa per difficoltà legate all'analisi del testo e alla scelta delle strategie risolutive da adottare.

In alcuni casi, le informazioni ottenute da questionari e interviste hanno evidenziato delle funzionalità da non considerare all'interno dell'applicazione, poiché utilizzate prevalente nella fase di correzione del codice, oppure in quanto generalmente poco usate.

Una di queste funzionalità è il supporto ad annotazioni sul programma scritto dallo studente, offerto come potenziamento e alternativa a commenti di tipo classico all'interno del codice. Dalle interviste è emerso che gli studenti di solito commentano il codice per avere un promemoria del funzionamento di quella sezione del programma, oppure per fornire chiarimenti se il professore dovesse visionare il codice.

Un altro caso è quello del supporto per la creazione di schemi o diagrammi: sono strumenti piuttosto usati dagli studenti, ma è stato riportato che molto spesso questi schemi servono solo per risolvere dei dubbi temporanei o per prendere delle decisioni, quindi una volta disegnati non vengono più consultati.

È stato infine considerato il supporto di altri studenti: il confronto con il punto di vista dei propri colleghi è stato riscontrato piuttosto spesso, ma nella maggior parte dei casi riguarda dubbi sul codice, problemi di sintassi o *debug*.

4.2 Alternative e scelte progettuali

Alcune delle caratteristiche emerse dall'analisi dello scorso paragrafo si prestano ad essere progettate in un numero diverso di modi. Di seguito verranno proposte le varie alternative, indicandone punti di forza e di debolezza e scegliendo infine la strategia più adatta.

Suddivisione del testo in sottoproblemi: Durante la suddivisione del testo in sottoproblemi, si prevede che lo studente selezioni all'interno dell'esercizio una parte del testo che corrisponda al sottoproblema individuato. Questa operazione può essere effettuata con due livelli di granularità: si possono selezionare le righe appartenenti alla sezione (granularità a livello di riga), o si può evidenziare il testo e assegnarlo alla sezione (granularità a livello di carattere). Nonostante la selezione a livello di riga possa rendere più chiara la differenza tra creazione di un sottoproblema e annotazione del testo (in cui è necessario selezionare il testo, come ad esempio per la sottolineatura), la riga non è un'unità logica del problema. Si possono infatti avere casi in cui più sottoproblemi si trovano nello stesso paragrafo, o viceversa, in cui una richiesta particolarmente complessa venga spiegata da più righe.

Annotazioni sul testo del problema: Ci sono vari modi per permettere agli studenti di annotare il testo e mettere in evidenza delle parole chiave. Alcuni metodi, come il grassetto e la sottolineatura, non presentano particolari problemi. Altri metodi più ingombranti dal punto di vista grafico, come ad esempio l'evidenziatore, rischiano di competere troppo per l'attenzione dello studente e riempire la schermata con troppi colori. Rispetto a un normale editor di testo, l'interfaccia di questo strumento prevede la creazione di sezioni che dovranno rappresentare l'elemento logico più importante e visibile all'interno dell'esercizio.

Analisi del problema prima di scrivere il programma: Stimolare lo studente ad analizzare il testo prima di programmare è uno degli obiettivi principali dell'applicazione. Un modo per raggiungere questo scopo è avere una fase iniziale in cui allo studente viene presentato solo il testo dell'esercizio (e in cui può suddividere in sottoproblemi e scegliere la sua strategia di risoluzione), con la possibilità in seguito di poter passare alla visualizzazione dell'editor per il codice. Un'altra soluzione è quella di permettere allo studente di scrivere

del codice solo se associato a uno specifico sottoproblema precedentemente individuato (invece che avere un unico editor in cui scrivere il programma, lo studente avrebbe più blocchi in cui scrivere, e ciascuno di questi blocchi esisterebbe in funzione di un sottoproblema).

Queste due proposte non si escludono a vicenda e, anzi, applicarle insieme permette di mitigare le loro criticità. Far scrivere allo studente del codice solo se associato a un sottoproblema rischia di spingere a un'analisi del problema a blocchi (in cui lo studente legge e individua il primo sottoproblema, interrompe l'analisi per scrivere del codice e poi prosegue con la lettura del testo), fenomeno che questo strumento mira esplicitamente a evitare. Se però allo studente viene prima fornita una pagina in cui non può ancora programmare, la fase di analisi e decomposizione può essere svolta nel suo complesso (eventualmente il passaggio alla pagina di scrittura del codice può essere subordinato al raggiungimento di qualche criterio minimo di analisi del problema). In modo simile, adottare solo l'uso di una pagina iniziale per l'analisi del problema potrebbe portare lo studente a percepire il processo come uno spreco di tempo e quindi a cercare di saltarlo o a passare il prima possibile alla programmazione, ma legando la decomposizione del problema alla scrittura del codice, il processo iniziale di analisi assume un ruolo attivo e di supporto anche nella fase di programmazione.

Supporto da parte del professore: Per fornire suggerimenti e indicazioni su come decomporre il testo dell'esercizio, il professore può preparare una propria suddivisione del problema che potrà essere usata dagli studenti come riferimento. Tale suddivisione, considerabile come una proposta di soluzione, può essere direttamente visualizzabile o può essere usata internamente per fare un confronto automatico con la decomposizione proposta dallo studente.

Questa seconda opzione ha però diversi punti critici legati al criterio di confronto tra le due suddivisioni: due soluzioni valide potrebbero comunque essere molto differenti tra loro, e in generale due decomposizioni potrebbero essere logicamente equivalenti ma avere granularità diversa (una sezione di testo che in una soluzione è definita come unico sottoproblema, in un'altra soluzione potrebbe essere scomposta ulteriormente). La comunicazione di tale confronto costituisce inoltre un altro problema: per evitare di mostrare la soluzione proposta dal professore, bisognerebbe fornire allo studente un qualche valore o percentuale ottenuti in funzione della somiglianza (ammesso che si possa definire, come detto prima) tra le due soluzioni. Tale informazione, oltre che difficilmente utilizzabile dallo studente, potrebbe portare all'assunzione sbagliata che la soluzione fornita dal professore sia corretta in senso assoluto e sia l'unica soluzione possibile.

Supporto nella scelta di algoritmi e strutture dati: Scegliere gli algoritmi e le strutture dati da adottare nelle varie fasi della risoluzione di un esercizio è un passo importante nell'analisi di un problema.

Un modo per incoraggiare gli studenti a operare tali scelte è quelli di legarle alla definizione dei sottoproblemi, facendo indicare allo studente la strategia da adottare nel sottoproblema creato. Un campo libero da compilare rischia però di essere un supporto troppo vago (lo studente potrebbe, ad esempio, impiegarlo per commenti non legati alla strategia di risoluzione), mentre una soluzione più strutturata, come ad esempio proporre una serie di strategie o strutture dati tra cui scegliere, avrebbe il problema di dover presentare una lista completa di possibili proposte, correndo comunque il rischio che l'idea dello studente non sia tra quelle immaginate dal professore. Legare la scelta di una strategia risolutiva alla fase di decomposizione, indipendentemente dall'implementazione, rischia di essere problematico o confondere lo studente, in quanto in molti casi un sottoproblema non è associabile in modo chiaro a una struttura dati o a una specifica strategia.

Per supportare lo studente in questa fase, e in generale per guidarlo nella fase di analisi e comprensione dell'esercizio, il professore può invece proporre delle domande a risposta multipla o aperta. Le domande a risposta aperta possono aiutare a chiarire in anticipo possibili dubbi dello studente o a portarlo a scegliere una soluzione tra varie possibilità, mentre quelle a risposta chiusa possono aiutare lo studente a descrivere la propria soluzione più nel dettaglio. La domanda ha inoltre il vantaggio di essere posizionata dal professore in punti del testo ritenuti più critici, in cui lo studente potrebbe avere più dubbi, oppure in punti che richiedono la scelta di algoritmi o strutture dati più complesse.

4.3 Progetto dell'applicazione

Alla luce di quanto emerso in precedenza, sono stati definiti comportamento e funzionalità dell'applicazione.

L'applicazione supporta due tipi di utenti: i professori (autorizzati a caricare nuovi esercizi) e gli studenti (che possono risolvere gli esercizi presenti nel sistema).

Le interazioni principali all'interno dell'applicazione sono mostrate nella Figura 4.1. Le pagine (mostrate come nodi all'interno del grafico) possono essere suddivise in tre categorie: il login, la home page e la pagina per l'esercizio di programmazione. Una volta autenticato in base al suo ruolo (studente o professore), l'utente ha accesso alla pagina principale, in cui può scegliere di risolvere un problema di informatica (se studente), oppure creare un nuovo esercizio o visualizzarne uno già esistente (se professore).

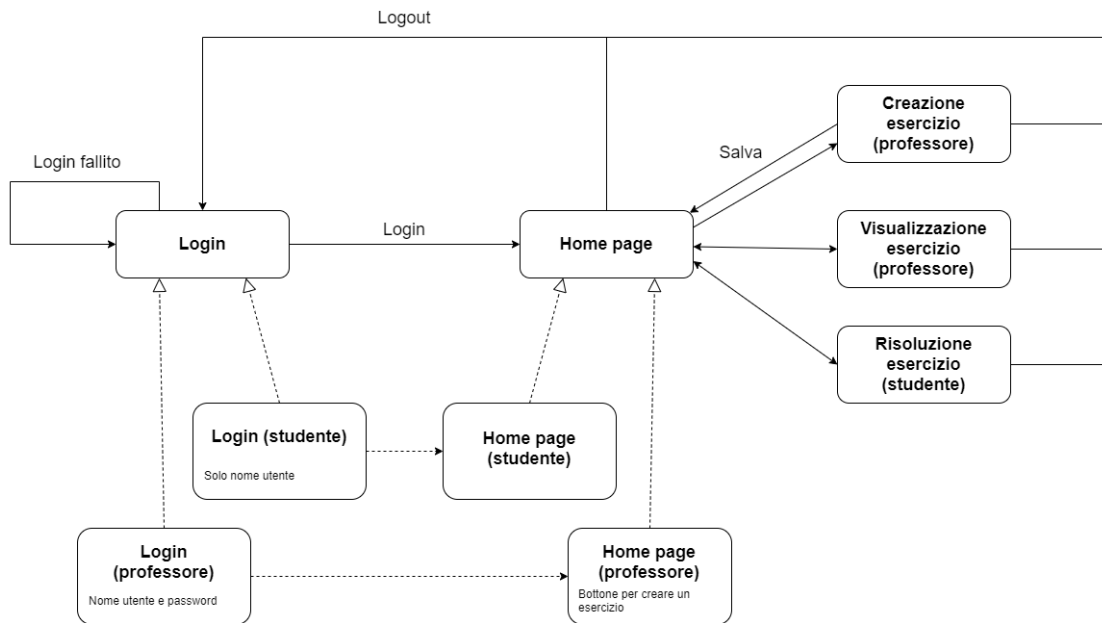


Figura 4.1: Interazioni principali all'interno dell'interfaccia

4.3.1 Funzionalità principali

L'interfaccia di risoluzione dell'esercizio permetterà allo studente di prendere visione del testo dell'esercizio e di suddividerlo in sottoproblemi evidenziando il testo stesso. Partendo da questi sottoproblemi, lo studente potrà creare dei frammenti di codice (ciascuno associato a uno specifico sottoproblema) che costituiranno poi il suo programma.

Per incoraggiare l'analisi del problema, inizialmente lo studente interagirà solo con il testo dell'esercizio, senza scrivere il programma. Uno sketch dell'interfaccia è proposto nella Figura 4.2. In questa schermata lo studente può creare ed eliminare sezioni all'interno del testo, corrispondenti ai sottoproblemi individuati dallo studente e ciascuna con un colore diverso. Lo studente può anche annotare il testo (con grassetto e sottolineatura) e rispondere alle domande proposte dal professore, che serviranno da autovalutazione e guida nei punti centrali dell'esercizio.

Una volta analizzato il problema, allo studente verrà permesso di passare alla schermata completa (mostrata in Figura 4.3) in cui è possibile scrivere il proprio programma sotto forma di frammenti di codice. Ogni frammento di codice viene assegnato a un sottoproblema in fase di creazione (tale assegnazione può essere cambiata in ogni momento), ma a ogni sottoproblema possono essere assegnati più frammenti, in modo tale da permettere allo studente di sperimentare con più soluzioni diverse.

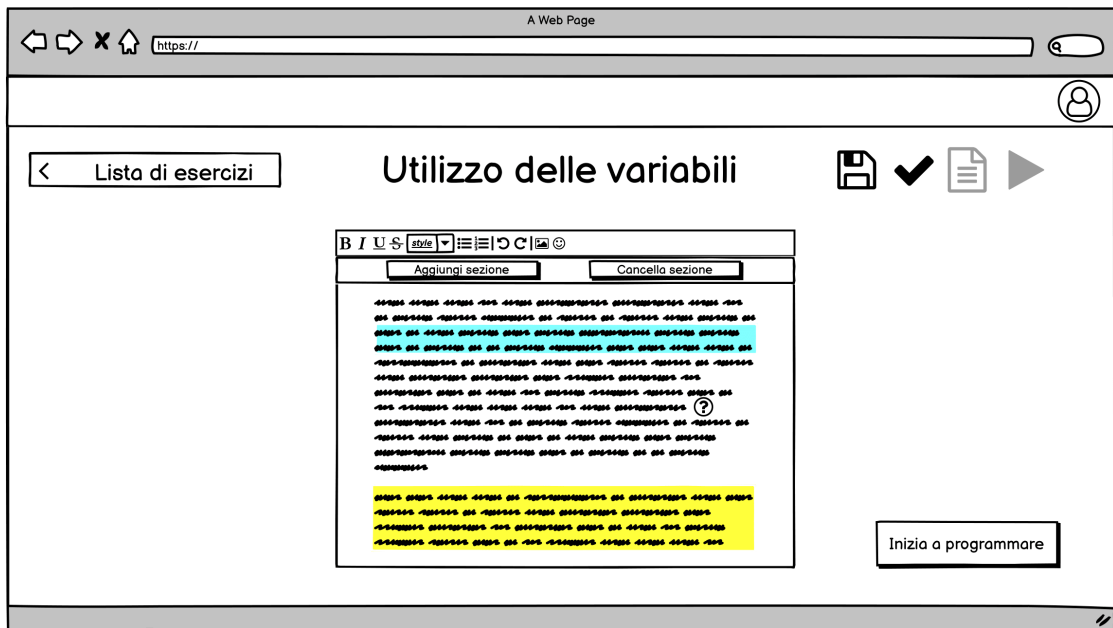


Figura 4.2: Wireframe dell'interfaccia prima di poter programmare

I frammenti di codice possono appartenere alla categoria del codice provvisorio o definitivo (rispettivamente seconda e terza colonna nella Figura 4.3), durante l'esecuzione del programma verranno presi in considerazione solo i frammenti di codice definitivi, presi nell'ordine indicato dallo studente. I frammenti possono essere spostati tramite *drag-and-drop* in modo tale da cambiare il loro ordine o spostarli da una colonna all'altra.

4.3.2 Funzionalità complementari

Lo studente ha inoltre a disposizione altre funzionalità. Tramite i bottoni in alto a destra, lo studente può (in ordine da sinistra a destra):

- salvare i propri progressi in ogni momento
- visualizzare consigli di decomposizione del problema proposti dal professore (per evitare che sia sempre disponibile, questa funzione potrà essere usata solo se lo studente avrà dimostrato di aver analizzato sufficientemente il problema). L'interazione è mostrata in Figura 4.4
- creare, modificare ed eliminare *file* testuali da usare come *input* e *output* all'interno del proprio programma (possibile nella visualizzazione completa, visto che non avrebbe senso interagire con i *file* prima di poter scrivere il programma). L'interazione è mostrata in Figura 4.5

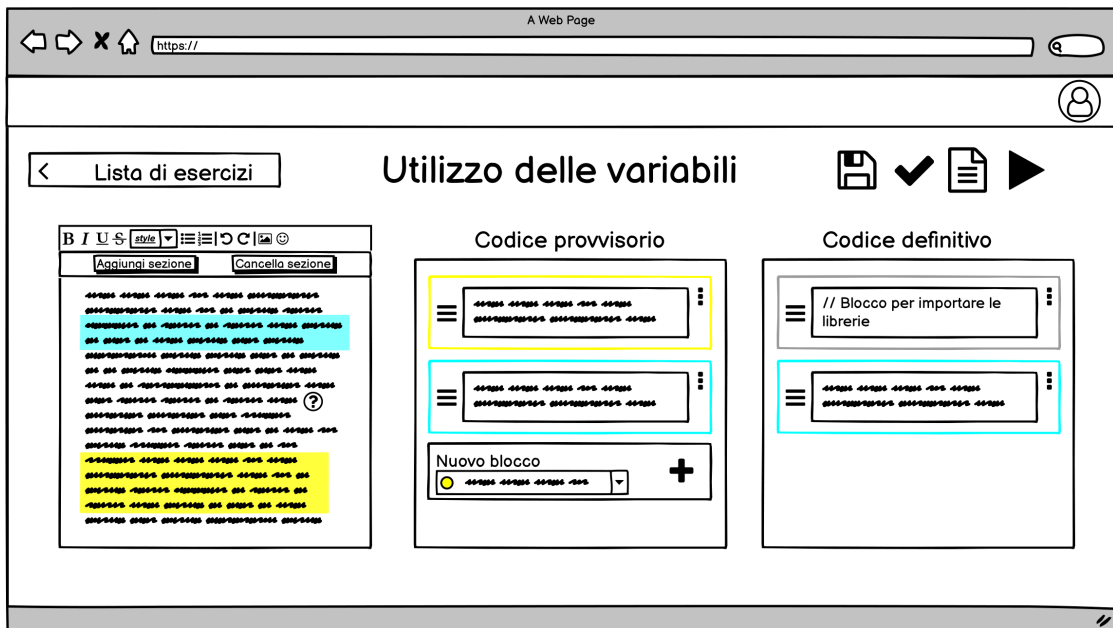


Figura 4.3: Wireframe dell'interfaccia completa

- eseguire il proprio programma (costituito dall'insieme ordinato dei frammenti di codice definitivi) e visualizzare l'*output* prodotto (funzionalità disponibili solo nella visualizzazione completa). L'interazione è mostrata in Figura 4.6

4.3.3 Funzionalità legate al professore

All'interno dell'applicazione il professore può creare nuovi esercizi e visualizzare gli esercizi creati in modalità di sola lettura.

Nella Figura 4.7 è mostrato uno sketch dell'interfaccia per la creazione di un esercizio. Nella finestra a sinistra il professore potrà scrivere il testo del problema, suddividerlo in sezioni (usate dal sistema per visualizzare i consigli allo studente) e creare domande. Nella finestra a destra sarà possibile modificare e cancellare le domande, inserire eventuali opzioni per domande a risposta multipla e aggiungere un commento visualizzabile dallo studente dopo aver risposto.

Una volta creato l'esercizio, esso sarà disponibile per essere visualizzato in una schermata il cui sketch è mostrato in Figura 4.8. In questa pagina sarà possibile leggere il testo del problema, visualizzare la suddivisione in sezioni e le domande con relative opzioni (se la domanda è a risposta multipla) e commento.

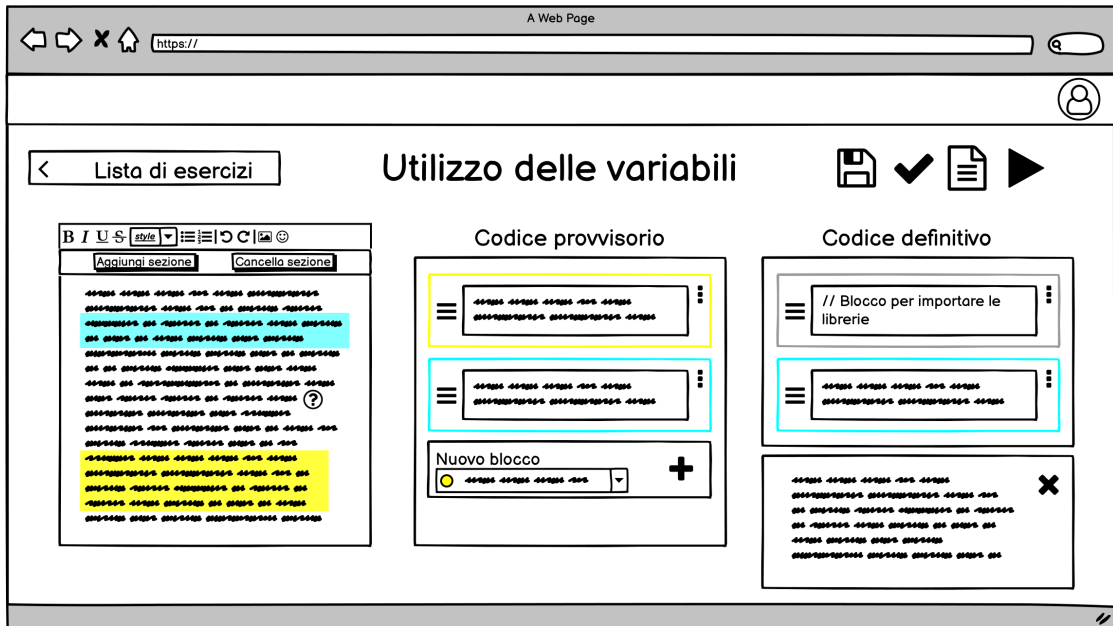


Figura 4.6: Esecuzione del programma

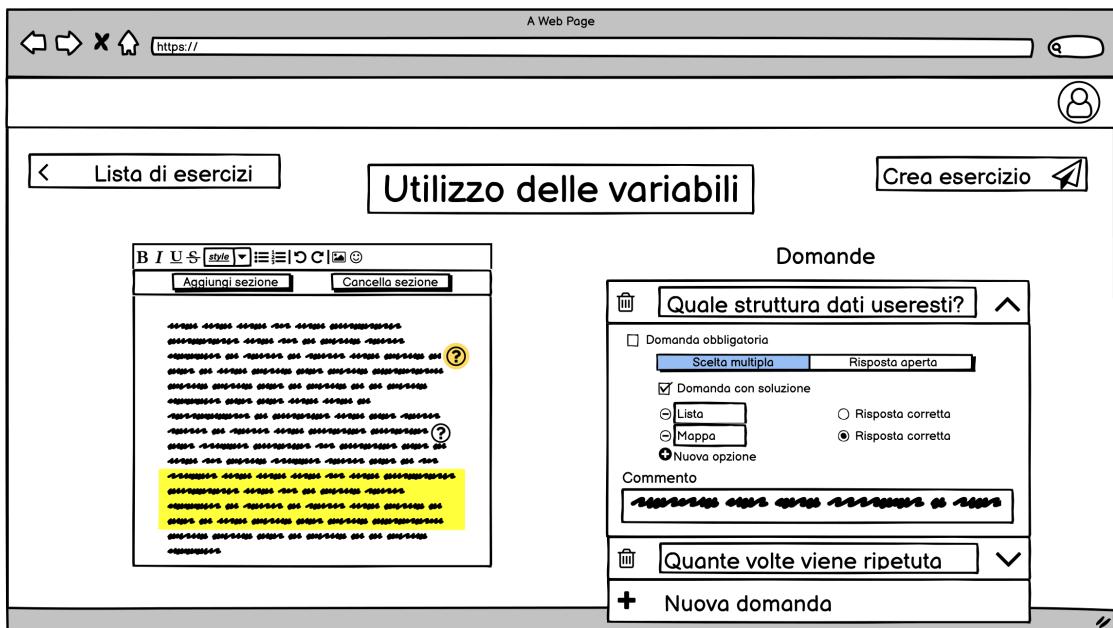


Figura 4.7: Creazione dell'esercizio

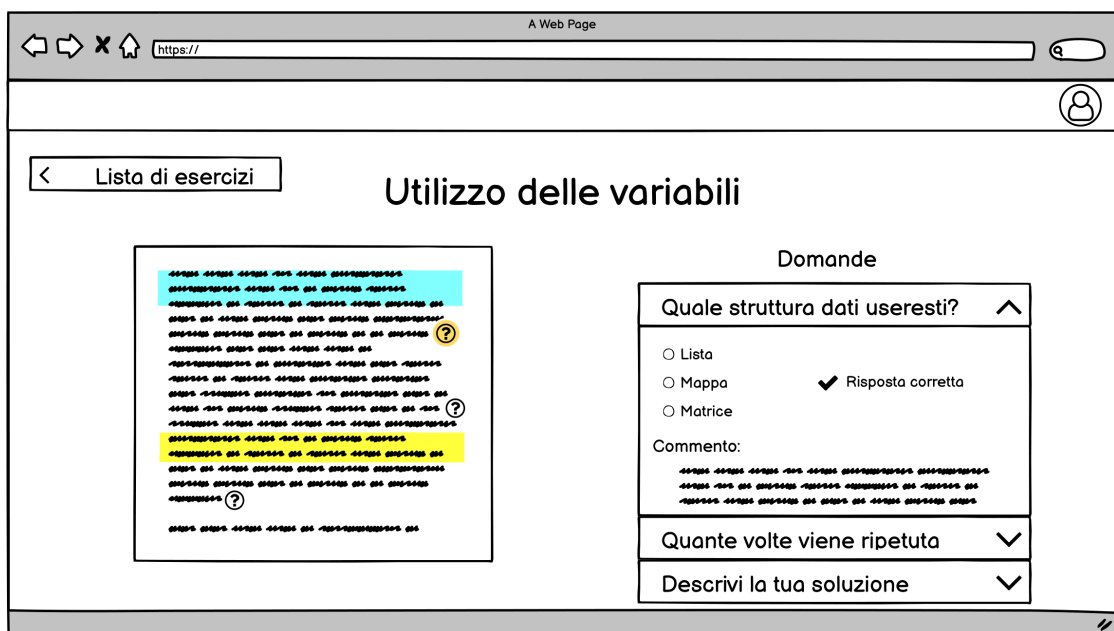


Figura 4.8: Visualizzazione dell'esercizio

Capitolo 5

Implementazione del sistema

I requisiti e le funzionalità definiti nel Capitolo 4 sono stati implementati sotto forma di applicazione web. Il progetto è stato scritto in JavaScript: il backend si basa su Express, mentre il frontend usa React.

5.1 Backend

Il backend si occupa di memorizzare i dati del sistema e permetterne l'aggiornamento, e comunica con il frontend tramite un'interfaccia REST.

Il backend si avvale di un database che contiene informazioni su utenti, esercizi caricati dai professori e soluzioni salvate dagli studenti. I dati sono salvati all'interno di un database non relazionale *in-memory* sfruttando la libreria NeDB (che al suo interno replica comportamento e funzionalità di MongoDB): data la natura prototipale del sistema, le collezioni vengono salvate all'interno di un file binario invece che in un DBMS.

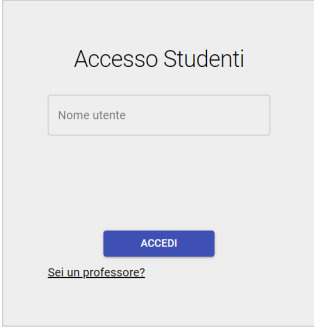
La scelta di un database non relazionale è dovuta alla natura stratificata dei dati, rappresentabili in modo più efficace con degli oggetti (basti pensare che ogni esercizio contiene liste di domande e sottoproblemi, i cui elementi sono a loro volta oggetti complessi).

5.2 Frontend

5.2.1 Login

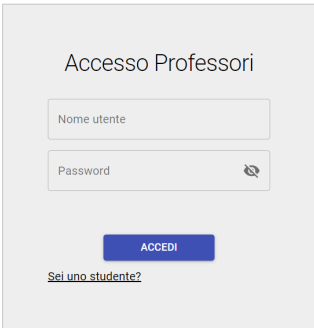
La prima pagina che viene presentata all'utente è quella di login, in cui si può effettuare l'accesso in base alle proprie credenziali e ruolo.

Gli studenti accedono utilizzando solo il loro nome utente (schermata in Figura 5.1), mentre i professori hanno bisogno di inserire nome utente e password (schermata in Figura 5.2).



The screenshot shows a login form titled "Accesso Studenti". It features a single text input field labeled "Nome utente". Below the input field is a blue button labeled "ACCEDI". At the bottom of the form, there is a link labeled "Sei un professore?".

Figura 5.1: Pagina di login per studenti



The screenshot shows a login form titled "Accesso Professori". It features two text input fields: "Nome utente" and "Password". The "Password" field includes a small eye icon for toggling visibility. Below the input fields is a blue button labeled "ACCEDI". At the bottom of the form, there is a link labeled "Sei uno studente?".

Figura 5.2: Pagina di login per professori

5.2.2 Home page

In questa pagina l'utente può navigare nella lista di esercizi presenti nel sistema.

Gli studenti possono scegliere di risolvere uno degli esercizi disponibili. Se l'esercizio è già stato iniziato, lo svolgimento riprenderà dall'ultima versione salvata. I professori possono invece selezionare un esercizio per visualizzarlo in modalità di sola lettura.

I professori possono inoltre creare un nuovo esercizio tramite il bottone presente in alto a sinistra nella scelta degli esercizi.

La versione per studenti della schermata è mostrata in Figura 5.3, mentre quella per professori in Figura 5.4



Figura 5.3: Pagina principale per studenti

5.2.3 Creazione esercizio

In questa pagina il professore può creare un nuovo esercizio, mettendolo a disposizione degli studenti.

Come si può vedere nella Figura 5.5, nella colonna a sinistra si trova l'editor di testo in cui il professore può scrivere il testo del problema, creare domande e selezionare i sottoproblemi per la soluzione da proporre allo studente.

La colonna centrale è dedicata alla compilazione delle domande create in precedenza. Il professore può determinare le caratteristiche della domanda (può decidere se è obbligatoria o opzionale, a scelta multipla o a risposta aperta, se possiede o meno una soluzione), oltre che scriverne il testo, eventuali opzioni (se a scelta multipla) e un commento che lo studente può visualizzare dopo aver risposto.

Nella colonna a destra il professore può scrivere dei commenti associati ai sottoproblemi individuati nel testo, in modo tale da rendere più chiare le sue scelte di decomposizione del problema.



Figura 5.4: Pagina principale per professori

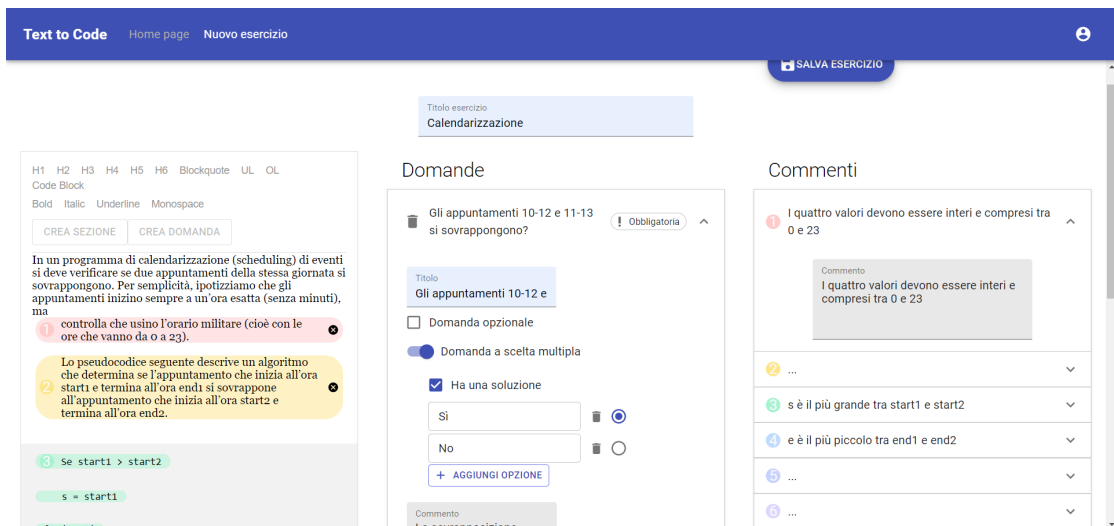


Figura 5.5: Pagina di creazione di un esercizio

5.2.4 Visualizzazione esercizio

Oltre a poter creare un esercizio, il professore può aprire problemi precedentemente creati in modalità di sola lettura (mostrata nella Figura 5.6).

In questa pagina è mostrato il testo dell'esercizio, completo di domande e delle sezioni di testo create dal professore, le quali costituiscono la soluzione proposta allo studente.

In aggiunta al testo del problema, si possono visualizzare testo e caratteristiche

delle domande, nonché i commenti (se presenti) ai singoli sottoproblemi.

The screenshot shows the 'Calendarizzazione' exercise page. On the left, there is a text block describing the scheduling problem with two numbered annotations: '1' pointing to a note about military time and '2' pointing to a note about the algorithm. Below the text is a code editor with several lines of code, including conditional statements like 'Se start1 > start2' and 's = start1'. The central 'Domande' section contains a question: 'Gli appuntamenti 10-12 e 11-13 si sovrappongono?' with an 'Obbligatoria' tag. Below the question are two options: 'Si' (marked as 'Soluzione corretta') and 'No'. A comment box below the options contains the text: 'La sovrapposizione avviene nell'intervallo 11-12'. The right 'Commenti' section shows a list of seven comment boxes. The first one contains the text: 'I quattro valori devono essere interi e compresi tra 0 e 23'. The other comment boxes are empty.

Figura 5.6: Pagina di visualizzazione di un esercizio

5.2.5 Risoluzione esercizio

Quando lo studente apre un esercizio, la pagina non permette la scrittura del codice, ma solo l'analisi e decomposizione del programma (schermata mostrata nella Figura 5.7). In questo momento lo studente può annotare il testo (con grassetto e sottolineatura), rispondere alle domande ed evidenziare i sottoproblemi creando sezioni all'interno del testo. Nella Figura 5.8 è mostrata una domanda a cui lo studente ha risposto: è possibile leggere un commento lasciato dal professore e, se la domanda prevede una risposta corretta, viene segnalato se la risposta è giusta o sbagliata.

Una volta che lo studente avrà risposto a tutte le domande obbligatorie, sarà possibile entrare nella versione completa della pagina (in Figura 5.9), in cui si potrà scrivere e testare il proprio programma.

Per scrivere il programma è necessario creare dei frammenti di codice, assegnando ciascuno di essi a una specifica sezione creata nel testo del problema. A ogni sezione può invece corrispondere più di un frammento di codice. I frammenti possono far parte del codice provvisorio (colonna centrale) o di quello definitivo (colonna a destra): i frammenti del codice definitivo, in ordine, costituiranno il testo del programma in fase di esecuzione. È inoltre possibile spostare (tramite *drag-and-drop*) i frammenti da una colonna all'altra e riordinarli all'interno della colonna, oltre che assegnare un frammento a una sezione di testo diversa e cancellare il frammento. Dato che le istruzioni di *import* non appartengono nello specifico a



Figura 5.7: Pagina di risoluzione di un esercizio, senza frammenti di codice

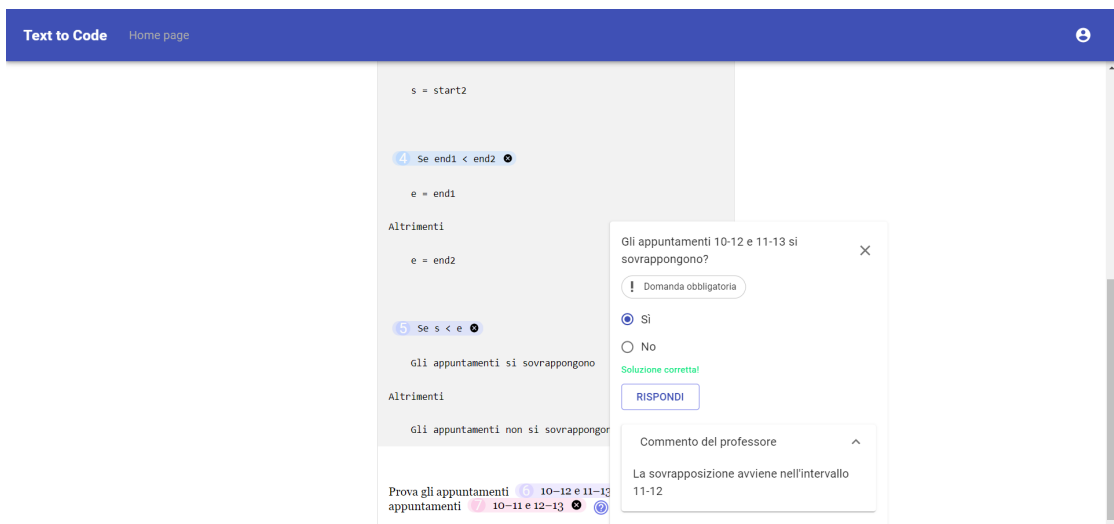


Figura 5.8: Domanda a cui lo studente ha risposto

nessun sottoproblema, è presente un blocco di codice statico all'inizio della colonna definitiva.

Tramite i comandi in alto a destra, lo studente può compiere altre operazioni: è possibile (in ordine, da sinistra a destra) salvare il proprio esercizio, interagire con i propri *file* di testo, visualizzare la soluzione fornita dal professore ed eseguire il codice.

Nella finestra dedicata ai *file* di testo (mostrata nella Figura 5.10) è possibile creare, modificare e cancellare documenti in formato testuale che potranno essere

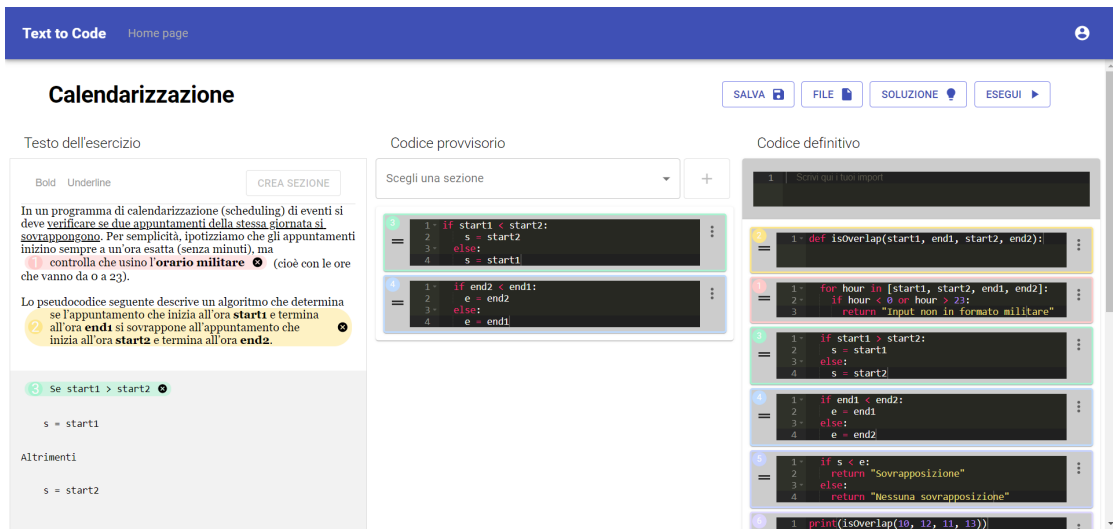


Figura 5.9: Pagina di risoluzione di un esercizio, modalità completa

usati come *input* e *output* durante l'esecuzione del programma. La finestra con

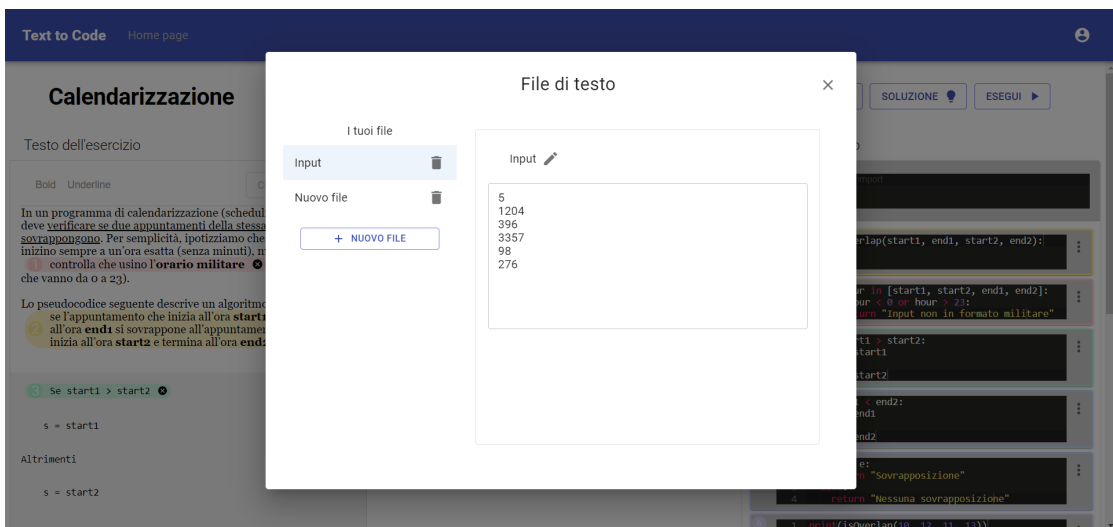


Figura 5.10: Finestra per la visualizzazione e modifica dei file di testo

la soluzione proposta dal professore (in Figura 5.11) permette allo studente di vedere una possibile decomposizione del problema, con la possibilità di visualizzare (facendo *hover* su un sottoproblema) eventuali commenti lasciati dal professore. Quando lo studente esegue il proprio programma, nella pagina viene aggiunta una nuova colonna (visibile nella Figura 5.12) in cui vengono mostrati il testo completo del programma e il relativo output. L'assenza di suddivisione in frammenti permette



Figura 5.11: Finestra per la visualizzazione della soluzione proposta dal professore

di avere una numerazione corretta delle linee di codice, importante per individuare eventuali errori durante la fase di *debug*.

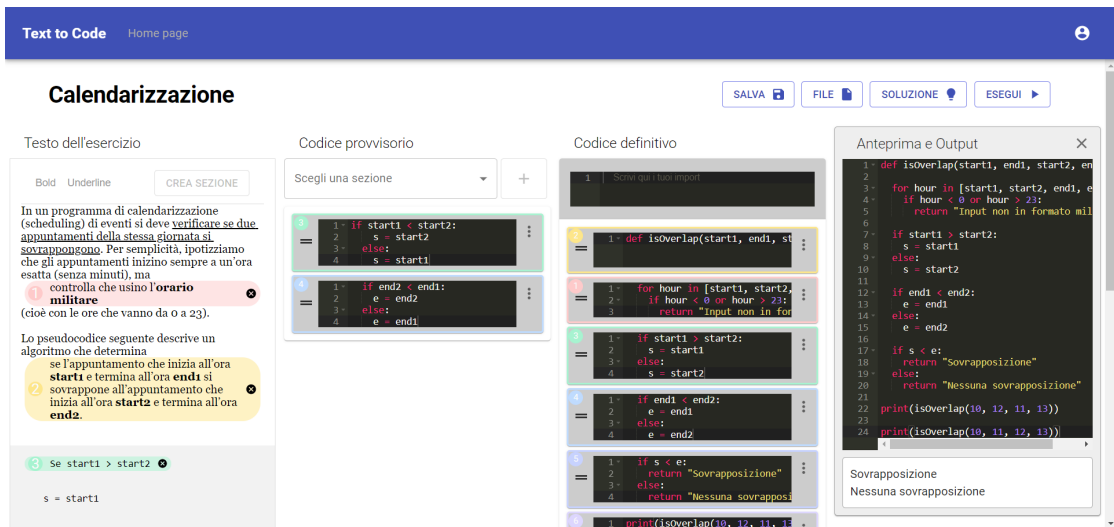


Figura 5.12: Pagina di risoluzione di un esercizio quando viene eseguito il programma

Capitolo 6

Valutazione

Una volta conclusa l'implementazione dell'applicazione descritta nel Capitolo 5, sono stati condotti dei test di usabilità per individuare eventuali problemi di efficacia delle singole funzionalità offerte.

6.1 Pianificazione

Il test si compone di una prima fase in cui viene chiesto allo studente di svolgere una serie di brevi attività (chiamate *task*), ciascuna incentrata su una funzionalità da testare. Una volta effettuati tutti i *task*, allo studente verranno proposte delle domande riepilogative dell'esperienza.

6.1.1 Task

Allo studente è chiesto di svolgere una serie di *task* all'interno di due esercizi, ipotizzando di aver già eseguito l'accesso tramite le proprie credenziali (il login è infatti un'operazione standard che non richiede valutazione). Il primo esercizio, dal titolo “Misura il rettangolo”, non ha dati precedentemente salvati, mentre il secondo, chiamato “Calendarizzazione”, è già stato iniziato e si chiede allo studente di completarlo.

Gli esercizi sono stati scelti tra quelli proposti nei laboratori del corso di Informatica del Politecnico di Torino tenuto dal professore Corno nell'a.a. 2021/2022 [21]. In particolare, “Misura il rettangolo” è tratto dall'esercizio 5 del Laboratorio 2, mentre “Calendarizzazione” è preso dall'esercizio 3 del Laboratorio 1.

Per ogni *task* viene definito un criterio di successo, cioè una serie di requisiti oggettivi che devono essere soddisfatti per ritenere l'attività conclusa correttamente. Ogni *task* ha inoltre un tempo massimo di esecuzione, al termine del quale il *task* viene considerato fallito.

Alcuni dei task prevedono l'impiego del *think-aloud* [22] (“pensare ad alta voce”), una tecnica che prevede che lo studente esprima quello che pensa mentre naviga all'interno della pagina e svolge la sua attività. Considerando che il *think-aloud* mette lo studente in una situazione innaturale, il suo utilizzo è limitato ai task più complessi, per i quali è più importante scoprire i pensieri dello studente.

L'elenco completo dei task è mostrato nella Tabella 6.1.

Task	Testo del task	Criterio di successo e uso del <i>think-aloud</i>	Tempo massimo
1	Apri un esercizio sulla geometria	Apertura dell'esercizio “Misura il rettangolo”	3 min.
2	Individua nel testo dell'esercizio almeno tre sottoproblemi	Creazione di almeno tre sezioni di testo. Uso del <i>think-aloud</i>	5 min.
3	Rispondi alle domande obbligatorie e verifica che le tue soluzioni siano corrette	Risposta solo alle domande obbligatorie, lettura del commento e/o del feedback di risposta giusta o sbagliata (se presente)	5 min.
4	Per ciascun sottoproblema che hai individuato, scrivi un frammento di codice che lo risolva	Passaggio alla visualizzazione completa, creazione di un frammento di codice associato a ciascuna sezione	15 min.
5	Cambia esercizio, aprendo quello chiamato “Calendarizzazione”	Ritorno alla pagina principale, apertura dell'esercizio con nome “Calendarizzazione”	3 min.
6	Scegli un frammento di codice del sottoproblema 2 e assegnalo al sottoproblema 4	Associazione di un frammento di codice della sezione 2 alla sezione 4. Uso del <i>think-aloud</i>	5 min.
7	Associa il frammento di codice che non ha un sottoproblema al sottoproblema corretto	Associazione del frammento di codice senza sezione alla sezione 3. Uso del <i>think-aloud</i>	5 min.
8	Scegliendo un modo per risolvere ogni sottoproblema, esegui il programma e confronta l'output con il risultato che ti aspetti	Scegliere almeno un frammento di codice per sezione (anche più di uno perché l'esecuzione del codice potrebbe anche essere spezzata), eseguire il programma, leggere l'output. Uso del <i>think-aloud</i>	10 min.
9	Confronta la tua soluzione con quella del professore, e leggimi il suo commento sul sottoproblema 1	Aprire la soluzione del professore, individuare le sezioni, leggere il commento della sezione 1. Uso del <i>think-aloud</i>	5 min.

Tabella 6.1: Elenco dei task

Per ogni task verranno valutate le seguenti metriche:

Numero di errori critici: errori che impediscono di completare il task

Numero di errori non critici: errori che permettono comunque il completamento del task, ma in una forma incompleta e/o scorretta

Completion Rate: frazione degli studenti che non ha commesso alcun errore critico

Error-Free Rate: frazione degli studenti che non ha commesso alcun errore, sia critico che non critico

Per valutare il possibile impatto sui test dovuto all'ordine dei task, alcuni di essi verranno proposti in una sequenza diversa per ciascun partecipante al test di usabilità. In particolare i task 2 e 3 possono essere scambiati tra loro senza modificare le condizioni del sistema, così come i task 6 e 7 e i task 8 e 9. In totale si hanno dunque otto ordinamenti diversi dei task.

Si è scelto inoltre di non testare le funzionalità di creazione e modifica dei *file* all'interno di questo test di usabilità, dato che tali processi non riguardano il caso d'uso principale del sistema, e l'introduzione di esercizi di programmazione più complessi avrebbe rischiato di richiedere uno sforzo eccessivo agli studenti (fattore che avrebbe potuto alterare i risultati del test).

Per evitare che lo studente si concentri troppo sulla parte di scrittura del codice (dato che le funzionalità principali dell'applicazione sono legate alla comprensione dell'esercizio), non sono stati quindi scelti esercizi troppo complessi.

6.1.2 Domande post-test

Una volta svolti tutti i task, allo studente verrà chiesto di rispondere a una serie di domande riepilogative sulla sua esperienza con l'applicazione, alcune orali e altre sotto forma di questionario.

Le prime domande orali sono uguali per tutti gli studenti e servono per raccogliere le considerazioni degli studenti. Altre domande possono essere poste sul momento per chiedere chiarimenti su comportamenti specifici dello studente durante lo svolgimento dei task, in modo da ottenere informazioni specifiche su eventuali errori o supposizioni dello studente.

Il questionario utilizzato in questo test è il SUS, un questionario standard per la misura della percezione di usabilità da parte dell'intervistato [23]. La compilazione del questionario prevede di indicare il proprio accordo o disaccordo con dieci affermazioni usando una scala Likert a cinque livelli, e produce un punteggio da 0 a 100 in cui valori al di sopra del 68 sono considerati sopra la media.

Il testo completo del questionario è riportato nell'Appendice C.

6.2 Svolgimento dei test

I partecipanti a questo test di usabilità sono stati otto studenti del Politecnico di Torino che abbiano frequentato e superato il corso di Informatica. Come per lo studio descritto nel Capitolo 3, tale restrizione serve per ipotizzare che lo studente abbia familiarità con la programmazione, dato che non viene registrata la frequenza alle lezioni del corso.

Oltre a questa restrizione, si è evitato di contattare studenti che avessero già partecipato allo studio svolto in precedenza.

Si è evitato di contattare studenti del primo anno in quanto, al momento dello svolgimento di questi test, non avrebbero avuto il tempo necessario per acquisire abbastanza conoscenze e competenze informatiche per poter interagire con naturalezza con l'applicazione.

Per ridurre al minimo la variabilità tra un test e l'altro, è stato creato uno script (disponibile in forma completa nell'Appendice D) da seguire con ogni partecipante: in esso sono definiti i passi da seguire e le parole da usare durante l'arco del test.

I test si sono svolti in modalità remota. Per la compilazione del questionario SUS è stato fornito un link verso un documento online, contenente una domanda introduttiva volta a confermare l'accettazione del modulo di consenso informato fornito allo studente quando è stato contattato per lo studio. I task sono stati svolti tramite controllo remoto dello schermo della macchina su cui è stata fatta partire l'applicazione.

6.3 Risultati

Il test di usabilità svolto col primo studente ha fatto emergere che, nonostante lo script preveda la descrizione iniziale delle finalità dello studio e dell'applicazione, è comunque necessario spiegare quali sono le funzionalità disponibili. In questo primo test, infatti, lo studente non è riuscito a completare alcuni task in quanto non sapeva cosa aspettarsi dall'applicazione.

Per risolvere tale incomprendimento è stata aggiunta una breve descrizione durante l'introduzione ai task:

“Quest'applicazione è pensata per studenti di informatica che si esercitano a programmare, e li guida nella fase di analisi del testo e nella traduzione del testo in codice. Con quest'applicazione puoi suddividere il testo del problema in sottoproblemi, e per ogni sottoproblema puoi creare uno o più frammenti di codice.”

6.3.1 Task

Le misure raccolte per ogni task sono riportate nella Tabella 6.2.

Task	Totale errori critici	Totale errori non critici	Completion rate	Error-free rate
1	0	0	1	1
2	4	5	0.5	0.25
3	0	11	1	0
4	3	7	0.625	0.125
5	0	0	1	1
6	0	3	1	0.625
7	0	1	1	0.875
8	1	5	0.875	0.375
9	2	2	0.875	0.625

Tabella 6.2: Misure ottenute dai risultati dei task

I task con Completion Rate più basso sono quelli in cui più studenti hanno commesso errori critici. È dunque particolarmente importante capirne le cause, in quanto in questi casi gli studenti non sono stati in grado di portare a termine l'obiettivo.

In ordine di Completion Rate crescente, i task che si sono dimostrati più problematici sono stati:

- **Task 2 e Task 4:** i problemi sono stati provocati dalla poca esperienza con l'applicazione. In particolare, gli studenti hanno fatto fatica a capire cosa sono e a cosa servono sezioni di testo e frammenti di codice all'interno dell'applicazione
- **Task 8:** lo studente non ha individuato la possibilità di spostare i frammenti di codice nella colonna definitiva
- **Task 9:** lo studente non ha trovato i commenti lasciati dal professore

I task caratterizzati da un Error-Free Rate basso sono invece quelli in cui molti studenti hanno commesso almeno un errore, che sia critico o non critico. Questo tipo di errori permettono di capire quali sono le problematiche all'interno dell'applicazione che, seppur non abbastanza gravi da interrompere lo svolgimento dell'attività, provocano delle incertezze nello studente.

In ordine di Error-Free Rate crescente, i task che hanno dato più difficoltà sono stati:

- **Task 3:** non è stata chiara la differenza tra domande opzionali e obbligatorie, quindi tutti gli studenti hanno risposto anche alla domanda opzionale
- **Task 2 e Task 4:** come nel caso degli errori critici, non sono stati chiari lo scopo e le funzionalità di sezioni di testo e frammenti di codice
- **Task 8:** sono stati commessi diversi errori, tra cui il più frequente e rilevante è stato il cercare di eseguire il proprio programma senza avere frammenti di codice nella colonna del codice definitivo
- **Task 9:** gli studenti hanno avuto difficoltà nell'accedere alla finestra della soluzione o nel visualizzare i commenti del professore
- **Task 6 e Task 7:** gli studenti hanno avuto problemi nella navigazione, in particolare nel distinguere la pagina di sola analisi del testo da quella con visualizzazione completa. Tali errori sono sempre stati non critici e infatti, una volta ambientati, gli studenti sono sempre stati in grado di portare a termine il task

Il cambio dell'ordine di esecuzione dei task non ha mostrato effetti particolarmente rilevanti. Nel caso dei task 6 e 7 si è notato che gli errori non critici sono stati commessi sempre nel primo tra i due task eseguiti. Questo sarebbe in linea con le considerazioni raccolte durante i test, cioè che lo studente non incontra particolari problemi una volta che ha avuto modo di prendere familiarità con il funzionamento dei frammenti di codice.

6.3.2 Domande post-test

I risultati ottenuti dai questionari SUS sono stati nella maggior parte dei casi nettamente al di sopra della soglia standard di 68. Anche il valore medio dei risultati è stato ampiamente soddisfacente, in quanto ha raggiunto il valore di 80.3. Il punteggio più basso (47.5) è stato dato durante il primo test, in cui non era ancora presente la spiegazione iniziale delle funzionalità del sistema.

Le affermazioni per cui gli studenti hanno riportato punteggi più bassi sono state *“Penso che mi piacerebbe usare questo sistema frequentemente”* (comprensibile, l'applicazione è pensata per studenti alle prime armi, mentre i partecipanti hanno già superato l'esame e sono più esperti), *“Penso che avrei bisogno del supporto di una persona già in grado di utilizzare il sistema”* (come evidenziato in precedenza, alcuni studenti hanno bisogno di un momento iniziale in cui essere guidati) e *“Mi sono sentito a mio agio nell'utilizzare il sistema”* (i punteggi più bassi vengono da alcuni dei partecipanti che hanno avuto problemi nella comprensione delle sezioni e dei frammenti di codice).

Le problematiche emerse più spesso durante le domande orali sono state:

- pensare che le domande nel testo siano dei suggerimenti
- pensare che per creare un frammento di codice si debba scrivere nella sezione degli import (confusione probabilmente dovuta alla presenza dell'editor per inserire gli import vicino al bottone per creare una sezione)
- non comprendere la funzione di trascinamento dei frammenti di codice
- avere dubbi iniziali sul poter creare più frammenti di codice legati alla stessa sezione di testo
- pensare che il bottone per la soluzione proposta dal professore riguardi il codice e non la suddivisione in sottoproblemi
- cercare di eseguire il programma senza frammenti di codice nella colonna del codice definitivo

6.3.3 Possibili modifiche all'interfaccia

I risultati ottenuti da questo test di usabilità hanno permesso di evidenziare una serie di possibili cambiamenti da adottare per migliorare l'usabilità dell'interfaccia, riportati di seguito in ordine di importanza.

La modifica più importante emersa da questi test è la necessità di avere un qualche tipo di introduzione all'applicazione per lo studente, nella forma di una spiegazione o esempio pratico dati dal professore a lezione, oppure tramite tutorial alla prima apertura dell'applicazione. Capire la funzione di sezioni e frammenti di codice la prima volta che si usa l'applicazione può essere complicato, ma i test hanno mostrato che, una volta abituati, gli studenti non hanno problemi a navigare nella pagina. L'introduzione dovrebbe coprire le funzioni base di una sezione di testo e di un frammento di codice, la possibilità di avere più frammenti associati alla stessa sezione e la possibilità di spostare e ordinare i frammenti.

I test hanno inoltre mostrato in modo evidente il bisogno di modificare la presentazione delle domande. La differenza tra domande obbligatorie e opzionali deve essere più evidente (ad esempio usando il tipico asterisco rosso accanto alla domanda, o cambiando il colore della finestra o del suo bordo), e l'icona delle domande all'interno del testo dovrebbe essere cambiata per non far pensare che fornisca dei suggerimenti.

Un altro aspetto che ha margini di miglioramento è la funzionalità di *drag-and-drop*: nonostante vengano usate metafore standard come l'icona di trascinamento e il puntatore a forma di mano, bisognerebbe rendere più evidente la funzionalità di trascinamento dei frammenti di codice e le aree in cui possono essere spostati. Per risolvere il problema si potrebbe provare a modificare l'aspetto delle colonne in cui si possono posizionare i frammenti.

I test hanno mostrato inoltre che bisognerebbe migliorare la presentazione della soluzione proposta dal professore. Si dovrebbe rendere più evidente il fatto che il bottone porti a una proposta di decomposizione del testo e non di scrittura del codice (in questo caso si potrebbero cambiare nome e icona del bottone), e rendere più intuitiva la visualizzazione dei commenti del professore, ad esempio mostrando i commenti accanto alla sezione, o avere un bottone esplicito all'interno della sezione stessa.

Un altro processo migliorabile all'interno dell'applicazione è infine la creazione di un frammento di codice, che dovrebbe essere reso più intuitivo. Alcuni studenti hanno pensato che, per poter creare un nuovo frammento, sia necessario prima scrivere il loro codice nella zona di import delle librerie, data la vicinanza di quest'ultima al bottone per la creazione dei frammenti. In questo caso si potrebbe prevedere un allontanamento di questi due elementi, magari riposizionando in fondo alla colonna gli elementi per la creazione di un frammento di codice.

Le riflessioni degli studenti hanno inoltre portato alla luce delle funzionalità che potrebbe essere interessante valutare in prossime versioni dell'applicazione:

- adottare una strategia più naturale di assegnazione dei numeri alle sezioni nel testo: invece che seguire un criterio puramente incrementale, si potrebbe assegnare il valore più basso tra quelli non utilizzati in quel momento, o dare un valore di un'unità più alto rispetto al numero maggiore attualmente in uso
- poter vedere la propria decomposizione del problema mentre si consulta la soluzione proposta dal professore
- avere la possibilità di scegliere il criterio di ordinamento (come per esempio ordine alfabetico o data di creazione) per gli esercizi proposti nella pagina principale, ed eventualmente poterli visualizzare sotto forma di lista
- poter spostare un frammento di codice da una colonna all'altra (da codice provvisorio a definitivo e viceversa) tramite un apposito bottone, oltre che con il *drag-and-drop*
- poter riconoscere se una domanda è opzionale o obbligatoria senza aver bisogno di aprirla

Capitolo 7

Conclusioni

Il lavoro svolto nell'ambito di questa tesi ha permesso di raggiungere gli obiettivi prefissati, cioè comprendere le difficoltà e le abitudini degli studenti di informatica nella risoluzione di esercizi di programmazione, per poi costruire e validare un'applicazione web pensata a tale scopo.

L'analisi della letteratura ha permesso di evidenziare che pratiche come l'analisi e la decomposizione di un esercizio di programmazione sono estremamente utili per studenti dei corsi introduttivi di informatica, ma spesso tali pratiche non vengono seguite. Dopo un'analisi degli strumenti per il supporto alla programmazione (sia di natura didattica che professionale) attualmente in uso, non è emersa alcuna applicazione che presenti tutte le caratteristiche auspiccate, nonostante la pervasività e la varietà di tali strumenti.

Tramite uno studio con studenti neofiti è stato possibile ottenere informazioni più dettagliate che, insieme a quelle ottenute dalla letteratura, hanno permesso di derivare una serie di funzionalità da implementare all'interno dell'applicazione web. Tra queste possono essere citate la suddivisione del testo in sottoproblemi, la visualizzazione in contemporanea del testo e del codice e l'incoraggiamento ad analizzare il problema prima di scrivere il programma.

L'applicazione è stata infine sviluppata e valutata tramite test di usabilità, da cui sono emersi feedback positivi e possibili occasioni future di miglioramento.

Di seguito viene riportata un'analisi del lavoro svolto, in cui si evidenziano le limitazioni a cui è stato soggetto e possibili sviluppi dei risultati raggiunti in questa tesi.

7.1 Limitazioni

Le osservazioni descritte nel Capitolo 3 hanno fornito informazioni utili sulle abitudini e le problematiche degli studenti di informatica. Il corso di Informatica

del Politecnico di Torino è erogato durante il primo semestre, mentre queste osservazioni sono iniziate durante il secondo. Per questo motivo non è stato quindi possibile coinvolgere studenti che stessero frequentando il corso di Informatica. Anche i test di usabilità del Capitolo 6 (svolti all'inizio del primo semestre dell'Anno Accademico successivo) hanno subito problemi simili, in quanto gli studenti del corso appena iniziato non avrebbero avuto abbastanza conoscenze e competenze per utilizzare con naturalezza l'applicazione.

Data l'assenza di un corso di Informatica svolto durante il periodo delle osservazioni preliminari, è stato inoltre difficile reperire studenti disposti a partecipare allo studio. Il numero di studenti contattati si è rivelato comunque sufficiente a raggiungere gli scopi della tesi, ma un numero maggiore di persone avrebbe permesso di ottenere più dati, eventualmente anche tramite uno studio di natura quantitativa.

7.2 Sviluppi futuri

Le limitazioni precedentemente esposte offrono un possibile spunto per degli ampliamenti dei risultati ottenuti in questa tesi. Si potrebbero infatti svolgere ulteriori osservazioni e test dell'interfaccia con studenti frequentanti il corso di Informatica. In un caso simile potrebbero essere considerati anche osservazioni sul campo (ad esempio durante le esercitazioni di laboratorio), test prolungati dell'applicazione (eventualmente comparandone l'efficacia con quella di IDE tradizionali), o test di tipo quantitativo con un numero elevato di partecipanti (ad esempio tramite questionari da condividere a tutti gli studenti del corso, o proposti durante le esercitazioni di laboratorio).

L'applicazione proposta in questa tesi può inoltre essere modificata partendo dai risultati ottenuti dai test di usabilità, oppure ottenendo nuove informazioni con studi più approfonditi, ad esempio tramite valutazioni euristiche.

Quest'applicazione ha inoltre il potenziale per essere utilizzata all'interno di un corso di Informatica come supporto allo studio individuale, oppure come strumento da utilizzare nelle esercitazioni di laboratorio o nelle dimostrazioni pratiche di programmazione durante le lezioni. Dato che lo strumento proposto è attualmente un prototipo, si renderebbero necessarie modifiche per l'uso da parte di studenti e docenti. Tali aggiornamenti comprenderebbero la pubblicazione dell'applicazione in rete, il potenziamento delle misure di sicurezza e l'ampliamento delle funzionalità offerte ai professori.

Appendice A

Questionario del Needfinding

- Ho letto e compreso il modulo di consenso, e acconsento a proseguire con il questionario (*domanda di controllo, la risposta è unica e obbligatoria*)
 - Acconsento

Dati personali

- Sesso (*risposta multipla*)
 - M
 - F
 - Altro
- Età (*risposta aperta*)
- Voto dell'esame di Informatica (*risposta aperta*)

Questionario

1. Quando risolvo l'esercizio di un laboratorio di informatica, preferisco (*risposta multipla*)
 - scrivere direttamente il codice
 - capire e analizzare prima il testo
2. Quando leggo il testo di un esercizio
 - leggo tutto il testo prima di iniziare a programmare (*scala Likert*)

- scrivo appunti accanto al testo dato (*scala Likert*)
 - scrivo appunti nei commenti del codice (*scala Likert*)
 - suddivido il testo in sottoproblemi (*scala Likert*)
3. Per aiutarmi nella comprensione del testo
- disegno diagrammi di flusso (*scala Likert*)
 - disegno le strutture dati (*scala Likert*)
 - tengo traccia dei valori delle variabili e del loro cambiamento nel tempo (*scala Likert*)
 - altro (*risposta aperta*)
4. Quando scrivo il codice
- ritorno spesso a consultare il testo dell'esercizio (*scala Likert*)
 - ritorno spesso a consultare gli appunti che ho scritto (*scala Likert*)
 - ritorno spesso a consultare i diagrammi che ho disegnato (*scala Likert*)
 - leggo una sezione del testo, la implemento e passo alla sezione successiva (*scala Likert*)
5. Quando leggo il testo di un esercizio, ho difficoltà a
- comprendere il significato del testo (*scala Likert*)
 - individuare le parole chiave e le sezioni importanti (*scala Likert*)
 - passare da una richiesta nel testo a una serie di istruzioni nel codice (*scala Likert*)
 - capire che tipo di variabili e strutture dati utilizzare (*scala Likert*)
 - scegliere i corretti algoritmi e strategie visti a lezione (*scala Likert*)
 - altro (*risposta aperta*)
6. Per risolvere problemi di comprensione del testo dell'esercizio
- cerco online (*scala Likert*)
 - guardo le slide o i miei appunti (*scala Likert*)
 - mi confronto con i miei colleghi (*scala Likert*)
 - mi rivolgo al professore o agli esercitatori (*scala Likert*)
 - altro (*risposta aperta*)
7. Quando completo un laboratorio d'informatica

- riesco a finire nei tempi previsti (*scala Likert*)
- il programma non genera errori di compilazione o esecuzione (*scala Likert*)
- il programma rispecchia le richieste dell'esercizio (*scala Likert*)
- mi sento soddisfatto del risultato (*scala Likert*)
- ho imparato qualcosa di nuovo (*scala Likert*)

Appendice B

Intervista del Needfinding

1. Tipicamente, nei laboratori di informatica, come risolvevi gli esercizi che ti venivano dati?

Possibili domande aggiuntive:

- Preferivi scrivere direttamente il codice, o ne analizzavi e decomponevi il testo?
- Come mai?

2. Quando esami il testo di un esercizio, come procedi?

Possibili domande aggiuntive:

- Annoti, evidenzi o suddividi il testo del problema? Se sì, come?
- In questa fase ti aiuti anche commentando il codice?
- Utilizzi anche schemi o diagrammi ausiliari, come per esempio
 - diagrammi di flusso
 - rappresentazione di strutture dati
 - schemi per vedere come cambiano i valori delle variabili nel tempo
 - altro?

Puoi farmi vedere un esempio?

3. Quando inizi a scrivere il codice del programma, come procedi?

Possibili domande aggiuntive:

- Ritorni spesso a guardare il testo/appunti/diagrammi? Come mai?
- Scrivi tutto il programma o alterni lettura del testo e scrittura del codice?

4. Tipicamente quali sono le difficoltà che incontri quando analizzi il testo di un esercizio e inizi a scrivere il tuo programma?
5. Quando incontri delle difficoltà nell'analizzare il testo di un esercizio, come preferisci agire?

Possibili domande aggiuntive:

- (se risponde "online" o "appunti/slide") Per dubbi sintattici o strategie di risoluzione?
- (se risponde "colleghi") In che modo?
- (se risponde "professore/esercitatori") Di solito cosa chiedi:
 - domande relative al testo?
 - possibili algoritmi o approcci da utilizzare?
 - input e relativi output dati dal professore?

6. Secondo te, quali sono gli aspetti positivi e negativi del tuo modo di affrontare un laboratorio di informatica?

Possibili domande aggiuntive:

- Hai provato anche altri metodi? Se sì, come ti sei trovato?
7. Per favore, disegna lo schizzo di un'interfaccia che pensi ti aiuterebbe a svolgere i laboratori di informatica, concentrandoti sulla fase in cui analizzi il testo e inizi a tradurlo in codice. Non ti preoccupare, non c'è bisogno che il disegno sia preciso, bastano anche delle forme semplici.

Appendice C

Questionario SUS

Il questionario prevede che l'utente riporti quanto è d'accordo con le dieci affermazioni presentate. La risposta per ciascuno va da 1 ("Per niente d'accordo") a 5 ("Totalmente d'accordo").

1. Penso che mi piacerebbe usare questo sistema frequentemente.
2. Ho trovato il sistema inutilmente complesso.
3. Ho trovato il sistema molto semplice da usare.
4. Penso che avrei bisogno del supporto di una persona già in grado di utilizzare il sistema.
5. Ho trovato le varie funzionalità del sistema bene integrate.
6. Ho trovato incoerenze tra le varie funzionalità del sistema.
7. Penso che la maggior parte delle persone possano imparare a utilizzare il sistema facilmente.
8. Ho trovato il sistema molto difficile da utilizzare
9. Mi sono sentito a mio agio nell'utilizzare il sistema.
10. Ho avuto bisogno di imparare molti processi prima di riuscire a utilizzare al meglio il sistema.

Appendice D

Script per il Test di Usabilità

D.1 Introduzione

Benvenuto/a! Sono Andrea Bruno, studente del corso di laurea magistrale di Ingegneria Informatica.

Questo test di usabilità mira a valutare l'efficacia di un'interfaccia sviluppata per assistere studenti del corso di informatica del primo anno. L'obiettivo è aiutarli nel passaggio dal testo di un esercizio alla scrittura del codice.

Per prima cosa, voglio sottolineare che questo test di usabilità è volto a valutare l'interfaccia e non te; ogni errore è colpa dell'applicazione, non tua.

Durante questo test di usabilità ti verrà chiesto di svolgere delle attività tramite l'interfaccia: al termine di ogni attività mi notificherai di aver finito, e passeremo a quella successiva. In seguito ti verranno poste delle domande riepilogative sulle tue impressioni. Questo test ha una durata stimata di 40 minuti.

Hai il diritto di ritirarti dalla partecipazione a questo test in ogni momento.

Qualche domanda prima di iniziare?

Risposta alle domande

Adesso ti passerò un link per un questionario da compilare alla fine di questo test, ma ti chiedo di aprirlo adesso per attestare il tuo consenso alla partecipazione a questo test di usabilità.

Consegna del modulo di consenso informato

Consegna del link per il questionario

D.2 Spiegazione del *think-aloud*

Per alcune delle attività ti sarà richiesto di pensare ad alta voce: devi dirmi cosa stai pensando! Può sembrare strano, ma è molto importante, ok?

Adesso ti mostro cosa intendo per “pensare ad alta voce”. Userò Google traduttore per questo esempio.

Aprire Google traduttore, passare a traduzione italiano/inglese

Voglio sapere come si dice “ciao” in francese. Vedo una scritta che mi dice “Inserisci il testo”, quindi mi aspetto di poter scrivere la parola che voglio tradurre se la clicco.

Cliccare sulla scritta e scrivere “ciao”

Vedo che la traduzione fornita a destra non è nella lingua giusta, quindi cerco il modo di passare al francese. In alto a destra noto la scritta “Inglese”, quindi immagino che sia la lingua in cui la parola verrà tradotta.

Cliccare sul bottone “Inglese”

Noto che è comparsa una lista di lingue in ordine alfabetico, quindi la scorro fino a trovare “Francese”

Cliccare su “Francese”

Adesso nel riquadro a destra vedo la traduzione in francese, quindi ho raggiunto il mio obiettivo.

Chiudere Google traduttore

D.3 Task

Quest’applicazione è pensata per studenti di informatica che si esercitano a programmare, e li guida nella fase di analisi del testo e nella traduzione del testo in codice. Con quest’applicazione puoi suddividere il testo del problema in sottoproblemi, e per ogni sottoproblema puoi creare uno o più frammenti di codice.

Adesso possiamo iniziare a svolgere le attività: ciascuna di esse verrà introdotta da una breve descrizione che la contestualizza, poi avrai modo di utilizzare l’applicazione per raggiungere l’obiettivo. Quando pensi di aver terminato l’attività, fammelo sapere.

Task 1: Sei uno studente del corso di informatica, e vuoi esercitarti con un problema sulla geometria.

Apri un esercizio adatto.

Svolgimento del task 1

Task 2: Ora che hai scelto un esercizio, individua nel testo almeno tre sottoproblemi. Con “sottoproblema” intendo una parte del testo che pensi racchiuda una delle richieste del problema.

In quest'attività "pensa ad alta voce"

Svolgimento del task 2

Task 3: Rispondi alle domande obbligatorie e verifica che le tue soluzioni siano corrette.

Svolgimento del task 3

Task 4: Ora che hai analizzato il problema, ti senti pronto/a a programmare.

Per ciascun sottoproblema che hai individuato, scrivi un frammento di codice che lo risolva.

Svolgimento del task 4

Task 5: Cambia esercizio, aprendo quello chiamato "Calendarizzazione".

Svolgimento del task 5

Da questa attività in poi, "pensa sempre ad alta voce"

Task 6: Dato che avevi già iniziato questo esercizio, puoi direttamente passare alla visualizzazione dei frammenti di codice.

Scegli un frammento di codice del sottoproblema 2 e assegnalo al sottoproblema 4.

Svolgimento del task 6

Task 7: Associa il frammento di codice che non ha un sottoproblema al sottoproblema corretto.

Svolgimento del task 7

Task 8: Adesso vuoi verificare il codice che hai scritto.

Scegliendo un modo per risolvere ogni sottoproblema, esegui il programma e confronta l'output con il risultato che ti aspetti.

Svolgimento del task 8

Task 9: Confronta la tua soluzione con quella del professore, e leggimi il suo commento sul sottoproblema 1.

Svolgimento del task 9

D.4 Post-Task

Bene, le attività sono terminate.

Abbiamo quasi finito, ma prima di concludere questo test di usabilità ho bisogno di sapere cosa pensi di questa esperienza.

Ho delle brevi domande da porti:

- Nel complesso, com'è stata la tua esperienza con l'applicazione?
- In quest'applicazione, quali pensi che siano un punto di forza e uno di debolezza?
- Hai qualche commento aggiuntivo?

Debriefing per domande e chiarimenti sullo svolgimento dei task

Per favore, ritorna alla pagina dove hai dato il consenso per questo studio, vai avanti e compila il breve questionario: per ogni affermazione dai un voto su una scala da 1 a 5, dove 1 è “Per nulla d'accordo” e 5 è “Completamente d'accordo”. Una volta risposto alle domande, puoi inviare il questionario.

Aspettare la risposta al questionario SUS

Grazie per il tuo aiuto, spero che ti sia divertito/a!

Bibliografia

- [1] Alex Lishinski, Aman Yadav, Richard Enbody e Jon Good. «The Influence of Problem Solving Abilities on Students' Performance on Different Assessment Tasks in CS1». In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE '16. Memphis, Tennessee, USA: Association for Computing Machinery, 2016, pp. 329–334. ISBN: 9781450336857. DOI: 10.1145/2839509.2844596. URL: <https://doi.org/10.1145/2839509.2844596> (cit. a p. 5).
- [2] E. Soloway. «Learning to Program = Learning to Construct Mechanisms and Explanations». In: *Commun. ACM* 29.9 (set. 1986), pp. 850–858. ISSN: 0001-0782. DOI: 10.1145/6592.6594. URL: <https://doi.org/10.1145/6592.6594> (cit. alle pp. 6, 29).
- [3] T.N. Arvanitis, M.J. Todd, A.J. Gibb e E. Orihashi. «Understanding students' problem-solving performance in the context of programming-in-the-small: an ethnographic field study». In: *31st Annual Frontiers in Education Conference. Impact on Engineering and Science Education. Conference Proceedings (Cat. No.01CH37193)*. Vol. 2. 2001, F1D–20. DOI: 10.1109/FIE.2001.963676 (cit. alle pp. 6, 29).
- [4] Katrina Falkner, Rebecca Vivian e Nickolas J.G. Falkner. «Identifying Computer Science Self-Regulated Learning Strategies». In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. ITiCSE '14. Uppsala, Sweden: Association for Computing Machinery, 2014, pp. 291–296. ISBN: 9781450328333. DOI: 10.1145/2591708.2591715. URL: <https://doi.org/10.1145/2591708.2591715> (cit. alle pp. 6, 29, 30).
- [5] Angela Carbone, John Hurst, Ian Mitchell e Dick Gunstone. «An Exploration of Internal Factors Influencing Student Learning of Programming». In: *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*. ACE '09. Wellington, New Zealand: Australian Computer Society, Inc., 2009, pp. 25–34. ISBN: 9781920682767 (cit. alle pp. 6, 29, 30).

-
- [6] Jamie Gorson e Eleanor O'Rourke. «Why Do CS1 Students Think They're Bad at Programming? Investigating Self-Efficacy and Self-Assessments at Three Universities». In: *Proceedings of the 2020 ACM Conference on International Computing Education Research*. ICER '20. Virtual Event, New Zealand: Association for Computing Machinery, 2020, pp. 170–181. ISBN: 9781450370929. DOI: 10.1145/3372782.3406273. URL: <https://doi.org/10.1145/3372782.3406273> (cit. alle pp. 7, 30).
- [7] Peter Brusilovsky et al. «Increasing Adoption of Smart Learning Content for Computer Science Education». In: *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference*. ITiCSE-WGR '14. Uppsala, Sweden: Association for Computing Machinery, 2014, pp. 31–57. ISBN: 9781450334068. DOI: 10.1145/2713609.2713611. URL: <https://doi.org/10.1145/2713609.2713611> (cit. a p. 7).
- [8] Philip J. Guo. «Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education». In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. Denver, Colorado, USA: Association for Computing Machinery, 2013, pp. 579–584. ISBN: 9781450318686. DOI: 10.1145/2445196.2445368. URL: <https://doi.org/10.1145/2445196.2445368> (cit. alle pp. 8, 9).
- [9] *Python Tutor*. URL: <https://pythontutor.com/> (cit. a p. 8).
- [10] Austin Henley, Julian Ball, Benjamin Klein, Aiden Rutter e Dylan Lee. «An Inquisitive Code Editor for Addressing Novice Programmers' Misconceptions of Program Behavior». In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 2021, pp. 165–170. DOI: 10.1109/ICSE-SEET52601.2021.00026 (cit. alle pp. 8, 10).
- [11] Alexandra Milliken et al. «PlanIT! A New Integrated Tool to Help Novices Design for Open-Ended Projects». In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. SIGCSE '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 232–238. ISBN: 9781450380621. DOI: 10.1145/3408877.3432552. URL: <https://doi.org/10.1145/3408877.3432552> (cit. alle pp. 9, 10).
- [12] Kabdo Choi, Sally Chen, Hyungyu Shin, Jinho Son e Juho Kim. «AlgoPlan: Supporting Planning in Algorithmic Problem-Solving with Subgoal Diagrams». In: *Proceedings of the Seventh ACM Conference on Learning @ Scale*. L@S '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 389–392. ISBN: 9781450379519. DOI: 10.1145/3386527.3406750. URL: <https://doi.org/10.1145/3386527.3406750> (cit. alle pp. 9, 11).

- [13] Marjan Adeli, Nicholas Nelson, Souti Chattopadhyay, Hayden Coffey, Austin Henley e Anita Sarma. «Supporting Code Comprehension via Annotations: Right Information at the Right Time and Place». In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2020, pp. 1–10. DOI: 10.1109/VL/HCC50065.2020.9127264 (cit. alle pp. 11, 12, 29).
- [14] Davor Čubranić e Gail C. Murphy. «Hipikat: Recommending Pertinent Software Development Artifacts». In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. Portland, Oregon: IEEE Computer Society, 2003, pp. 408–418. ISBN: 076951877X (cit. a p. 13).
- [15] Sebastian Baltes, Peter Schmitz e Stephan Diehl. «Linking Sketches and Diagrams to Source Code Artifacts». In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 743–746. ISBN: 9781450330565. DOI: 10.1145/2635868.2661672. URL: <https://doi.org/10.1145/2635868.2661672> (cit. alle pp. 13, 14).
- [16] *Statistiche MUR sulle immatricolazioni del Politecnico di Torino, 2020/21*. URL: <http://ustat.miur.it/dati/didattica/italia/atenei-statali/torino-politecnico> (cit. a p. 20).
- [17] *Statistiche AlmaLaurea sui laureati del Politecnico di Torino, anno di laurea 2020*. URL: <https://www2.almalaurea.it/cgi-php/universita/statistiche/framescheda.php?anno=2020&corstipo=L&ateneo=70032&facolta=tutti&gruppo=tutti&pa=70032&classe=tutti&corso=tutti&postcorso=tutti&isstella=0&isstella=0&presiuui=tutti&disaggregazione=&LANG=it&CONFIG=profilo> (cit. a p. 21).
- [18] *Risultati esami del corso di Informatica del Politecnico di Torino, 2020/21 (consultato in data 1 giugno 2021)*. URL: https://didattica.polito.it/pls/portal30/esami.superi.grafico?p_docente=&p_cod_ins=14BHDLZ&p_a_acc=2021 (cit. a p. 21).
- [19] Essi Lahtinen, Kirsti Ala-Mutka e Hannu-Matti Järvinen. «A Study of the Difficulties of Novice Programmers». In: *SIGCSE Bull.* 37.3 (giu. 2005), pp. 14–18. ISSN: 0097-8418. DOI: 10.1145/1151954.1067453. URL: <https://doi.org/10.1145/1151954.1067453> (cit. a p. 31).
- [20] Neil Smith, Mike Richards e Daniel G. Cabrero. «Summer of Code: Assisting Distance-Learning Students with Open-Ended Programming Tasks». In: *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE 2018. Larnaca, Cyprus: Association for Computing Machinery, 2018, pp. 224–229. ISBN: 9781450357074. DOI: 10.1145/3197091.3197119. URL: <https://doi.org/10.1145/3197091.3197119> (cit. a p. 31).

- [21] *Esercizi di laboratorio del corso di Informatica del Politecnico di Torino del professore Corno, 2020/21 (consultato in data 26 ottobre 2021)*. URL: <https://elite.polito.it/teaching/current-courses/527-14bhd-info?start=3> (cit. a p. 49).
- [22] *Thinking Aloud: The #1 Usability Tool (Jakob Nielsen, Nielsen Norman Group)*. URL: <https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/> (cit. a p. 50).
- [23] *System Usability Scale (SUS), usability.org*. URL: <https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html> (cit. a p. 51).