oggetto Relazione progetto Programmazione a Oggetti

gruppo Nardo Silvio, mat. 1222011

titolo Logic Gates

Introduzione

Logic Gates è un ambiente di simulazione di circuiti logici che permette di collegare tra loro componenti (quali porte logiche, interruttori e lampadine) e monitorare il loro comportamento in funzione dei collegamenti effettuati e degli interruttori premuti.

Due componenti possono essere collegati tra loro mediante dei “cavi”, i quali non si collegano direttamente al componente, bensì ai suoi “pin”.

Ogni componente, in base alla sua natura, ha un numero variabile di pin di input e di output,

per esempio ogni componente di tipo interruttore ha zero pin di input e un pin di output, ad esso si potrà collegare un cavo.

Dopo aver aperto il programma, la schermata si divide in tre parti:

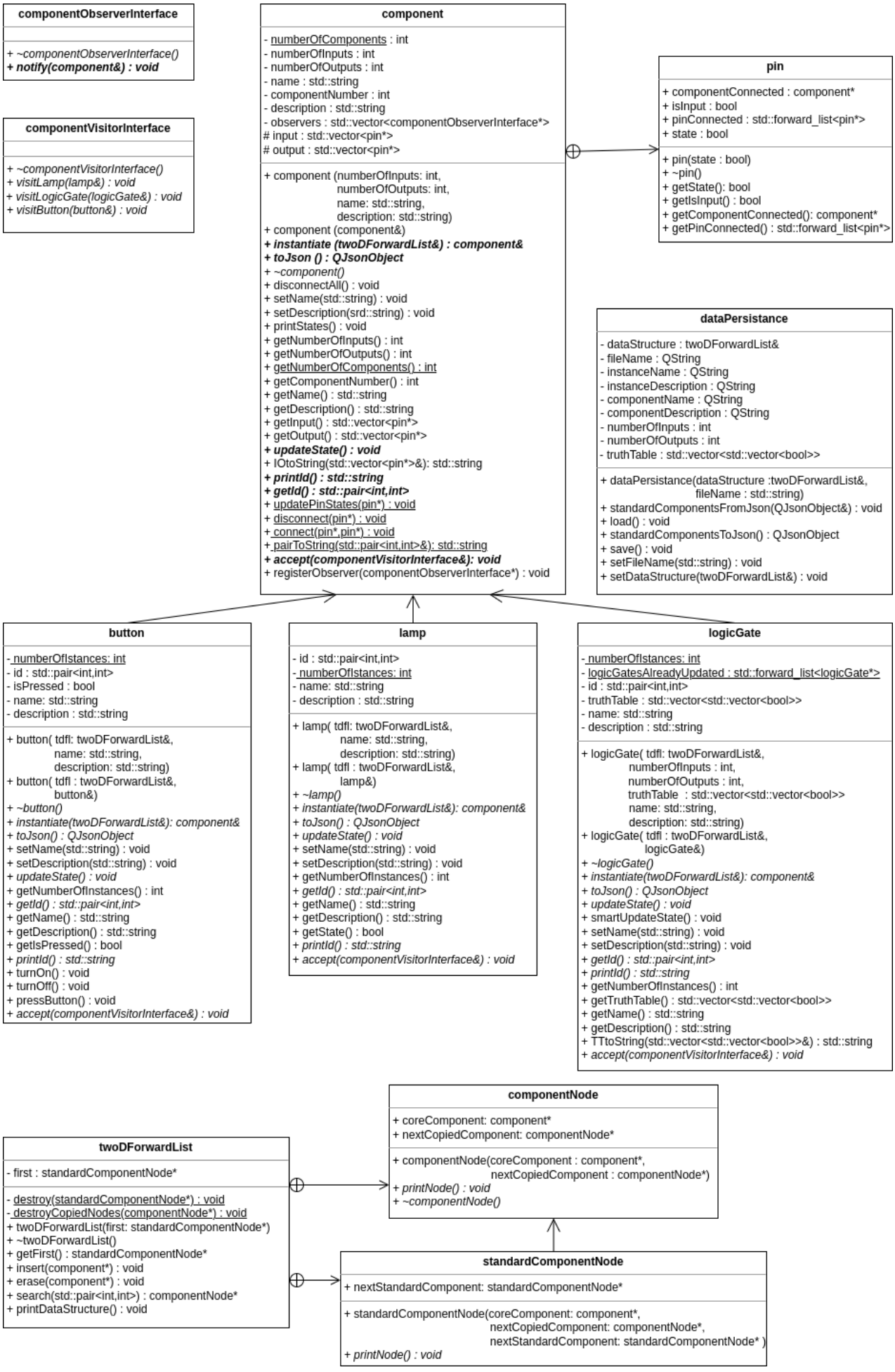
- Al centro è presente l’area di lavoro (soprannominata “workbench”) dove verranno collocati tutti i componenti generati e dove avverranno tutti i collegamenti.
- Una schermata laterale sinistra mostra i componenti standard ovvero le tipologie di diversi elementi che possiamo duplicare per poi utilizzare nel circuito.
- Una schermata laterale destra mostra tutte le informazioni relative ad un componente precedentemente selezionato e permette di eliminarlo.

Descrizione del modello

Il modello logico si divide in quattro aspetti principali:

1. La generazione dei componenti in base alla loro natura.
2. La struttura dati “twoDForwardList” utilizzata per organizzare tutti i componenti.
3. La persistenza dei dati gestita dalla classe “dataPersistance”.
4. La presenza degli strumenti Visitor e Observer utili per interfacciarsi con una parte grafica.

Di seguito viene riportato il diagramma UML delle classi del modello logico:


componentObserverInterface
+ ~componentObserverInterface()
+ notify(component&) : void
componentVisitorInterface
+ ~componentVisitorInterface()
+ visitLamp(lamp&) : void
+ visitLogicGate(logicGate&) : void
+ visitButton(button&) : void
component
- numberOfComponents : int
- numberOfInputs : int
- numberOfOutputs : int
- name : std::string
- componentNumber : int
- description : std::string
- observers : std::vector<componentObserverInterface*>
input : std::vector<pin*>
output : std::vector<pin*>
+ component (numberOfInputs: int, numberOfOutputs: int, name: std::string, description: std::string)
+ component (component&)
+ instantiate (twoDForwardList&) : component&
+ toJson () : QJsonObject
+ ~component()
+ disconnectAll() : void
+ setName(std::string) : void
+ setDescription(srd::string) : void
+ printStates() : void
+ getNumberOfInputs() : int
+ getNumberOfOutputs() : int
+ getNumberOfComponents() : int
+ getComponentNumber() : int
+ getName() : std::string
+ getDescription() : std::string
+ getInput() : std::vector<pin*>
+ getOutput() : std::vector<pin*>
+ updateState() : void
+ IOtoString(std::vector<pin*>&): std::string
+ printId() : std::string
+ getId() : std::pair<int,int>
+ updatePinStates(pin*) : void
+ disconnect(pin*) : void
+ connect(pin*,pin*) : void
+ pairToString(std::pair<int,int>&): std::string
+ accept(componentVisitorInterface&): void
+ registerObserver(componentObserverInterface*) : void
pin
+ componentConnected : component*
+ isInput : bool
+ pinConnected : std::forward_list<pin*>
+ state : bool
+ pin(state : bool)
+ ~pin()
+ getState(): bool
+ getIsInput() : bool
+ getComponentConnected(): component*
+ getPinConnected() : std::forward_list<pin*>
dataPersistance
- dataStructure : twoDForwardList&
- fileName : QString
- instanceName : QString
- instanceDescription : QString
- componentName : QString
- componentDescription : QString
- numberOfInputs : int
- numberOfOutputs : int
- truthTable : std::vector<std::vector<bool>>
+ dataPersistance(dataStructure :twoDForwardList&, fileName : std::string)
+ standardComponentsFromJson(QJsonObject&) : void
+ load() : void
+ standardComponentsToJson() : QJsonObject
+ save() : void
+ setFileName(std::string) : void
+ setDataStructure(twoDForwardList&) : void
button
- numberOfIstances: int
- id : std::pair<int,int>
- isPressed : bool
- name: std::string
- description : std::string
+ button(tdfl: twoDForwardList&, name: std::string, description: std::string)
+ button(tdfl : twoDForwardList&, button&)
+ ~button()
+ instantiate(twoDForwardList&): component&
+ toJson() : QJsonObject
+ setName(std::string) : void
+ setDescription(std::string) : void
+ updateState() : void
+ getNumberOfInstances() : int
+ getId() : std::pair<int,int>
+ getName() : std::string
+ getDescription() : std::string
+ getIsPressed() : bool
+ printId() : std::string
+ turnOn() : void
+ turnOff() : void
+ pressButton() : void
+ accept(componentVisitorInterface&) : void
lamp
- id : std::pair<int,int>
- numberOfIstances: int
- name: std::string
- description : std::string
+ lamp(tdfl: twoDForwardList&, name: std::string, description: std::string)
+ lamp(tdfl : twoDForwardList&, lamp&)
+ ~lamp()
+ instantiate(twoDForwardList&): component&
+ toJson() : QJsonObject
+ updateState() : void
+ setName(std::string) : void
+ setDescription(std::string) : void
+ getNumberOfInstances() : int
+ getId() : std::pair<int,int>
+ getName() : std::string
+ getDescription() : std::string
+ getState() : bool
+ printId() : std::string
+ accept(componentVisitorInterface&) : void
logicGate
- numberOfIstances: int
- logicGatesAlreadyUpdated : std::forward_list<logicGate*>
- id : std::pair<int,int>
- truthTable : std::vector<std::vector<bool>>
- name: std::string
- description : std::string
+ logicGate(tdfl: twoDForwardList&, numberOfInputs : int, numberOfOutputs : int, truthTable : std::vector<std::vector<bool>> name: std::string, description: std::string)
+ logicGate(tdfl : twoDForwardList&, logicGate&)
+ ~logicGate()
+ instantiate(twoDForwardList&): component&
+ toJson() : QJsonObject
+ updateState() : void
+ smartUpdateState() : void
+ setName(std::string) : void
+ setDescription(std::string) : void
+ getId() : std::pair<int,int>
+ printId() : std::string
+ getNumberOfInstances() : int
+ getTruthTable() : std::vector<std::vector<bool>>
+ getName() : std::string
+ getDescription() : std::string
+ TTtoString(std::vector<std::vector<bool>>&) : std::string
+ accept(componentVisitorInterface&) : void
componentNode
+ coreComponent: component*
+ nextCopiedComponent: componentNode*
+ componentNode(coreComponent : component*, nextCopiedComponent : componentNode*)
+ printNode() : void
+ ~componentNode()
twoDForwardList
- first : standardComponentNode*
- destroy(standardComponentNode*) : void
- destroyCopiedNodes(componentNode*) : void
+ twoDForwardList(first: standardComponentNode*)
+ ~twoDForwardList()
+ getFirst() : standardComponentNode*
+ insert(component*) : void
+ erase(component*) : void
+ search(std::pair<int,int>) : componentNode*
+ printDataStructure() : void
standardComponentNode
+ nextStandardComponent: standardComponentNode*
+ standardComponentNode(coreComponent: component*, nextCopiedComponent: componentNode*, nextStandardComponent: standardComponentNode*)
+ printNode() : void

Analizzando questi aspetti nel dettaglio:

1)

I componenti derivano tutti dalla classe astratta “component” la quale rappresenta un generico componente collegabile al circuito. Da essa vengono definite tre sottoclassi: button, lamp e logicGate. Le prime due rappresentano rispettivamente i vari input (button o interrutore) e output (lamp o lampadina) del circuito; la classe logicGate invece, esprime maggiormente il potenziale di questo programma:

essa rappresenta una porta logica generica definita in fase di costruzione da numero di input, numero di output e tabella di verità. Così facendo operatori logici differenti come l’AND, l’OR e il NOT sono rappresentati tutti dalla stessa classe logicGate, cambiando solo i parametri con cui viene istanziata.

Per gestire al meglio i collegamenti tra componenti non è stato necessario creare la classe cavo (nel modello logico) ma è presente la classe “pin”, annidata a component, la quale è incaricata di tenere traccia dei collegamenti e dello stato che essi propagano.

Mentre i pin di input possono essere collegati al massimo con un altro pin alla volta, i pin di output possono collegarsi ad un numero indefinito di altri pin.

Ogni sottocalsse di component ha un proprio univoco identificativo “id” composto da due interi: il primo identifica il tipo del componente mentre il secondo identifica il numero di istanza. Questa meccanica viene spiegata più approfonditamente al punto successivo.

La scelta di avere necessariamente identificativi univoci ha portato alla rimozione forzata dei costruttori di copia standard che genererebbero ambiguità.

Durante la costruzione di un componente (sia per i componenti standard che per le loro copie), esso viene automaticamente inserito nella struttura dati.

2)

La struttura dati adottata per ordinare tutti i componenti utilizzati è una forward list a due dimensioni (il nome della classe twoDForwardList deriva proprio dalla doppia dimensionalità).

Il programma prevede infatti che ci sia una lista di “componenti standard” presenti ciascuno una sola volta e non utilizzabili se non per generare, da una copia di sé sessi, i cosiddetti “componenti copiati” (o “componenti istanziati”). Questi ultimi sono quelli che effettivamente verranno generati nell’area di lavoro e potranno essere collegati per generare il circuito desiderato.

Partendo da un puntatore “first”, la struttura dati collega lungo una forward list tutti i componenti standard (aventi il secondo numero dell’id pari a zero); ciascuno di essi ha un puntatore che collega in una propria forward list ogni componente generato dalla copia di tale standard.

Così facendo si velocizza la ricerca dei componenti, la quale avviene tramite id, e si possono gestire i componenti copiati in maniera differente sapendo che derivano dallo standard (ad esempio il modello logico prevede che nel caso venga rimosso un componente standard dalla struttura dati, vengano automaticamente rimossi ed eliminati anche tutti i componenti copiati da esso).

I componenti standard e i componenti copiati sono puntati da nodi della struttura dati tra loro differenti:

i componenti copiati sono ciascuno dentro ad un “componentNode”, mentre i componenti standard sono in uno “standardComponentNode”.

La scelta di avere una struttura dati derivante da liste linkate piuttosto che da vettori deriva dalla considerazione che durante l’esecuzione del programma è molto probabile che avvengano numerose eliminazioni di componenti le quali richiederebbero una continua compattazione della memoria in caso dell’utilizzo dei vettori, il tutto a discapito di una funzione di ricerca più lenta ma comunque sufficientemente veloce.

Questa struttura permette quindi di inserire,rimuovere e cercare specifici elementi partendo dall’id, il quale identifica lo specifico componente standard con il primo numero e il componente copiato con il secondo.

3)

La presenza della struttura dati twoDForwardList è necessaria anche per gestire in maniera ordinata e controllata la persistenza dei dati. Quest’ultima avviene tramite l’apposita classe “dataPersistance”.

Maggiorni informazioni relative a questa classe sono presenti nella sezione “Persistenza dei dati” della relazione.

4)

Il modello logico è indipendente dall’implementazione grafica ma mette a disposizione delle classi utili per agevolare il recupero di informazioni tramite il polimorfismo e per notificare alla parte grafica eventuali modifiche dei dati in maniera efficiente.

Queste classi sono rispettivamente “componentVisitorInterface” per l’implementazione di visitor e “componentObserverInterface” per l’implementazione di observer.

In questa versione del programma non sono utilizzati observer ma la classe relativa è stata ugualmente implementata in vista di eventuali sviluppi futuri.

Polimorfismo

Il polimorfismo è stato usato principalmente mediante i visitor:

la classe “componentVisitorInterface” ha due sottoclassi,“componentInfoVisitor” e “componentViewVisitor”, utilizzate dalla parte grafica per ottenere in maniera polimorfa le informazioni necessarie a generare i widget grafici relativi rispettivamente alle informazioni di un componente da mostrare nella barra laterale destra dello schermo e all’aspetto che un componente deve avere nell’area di lavoro.

Il polimorfismo è stato utilizzato per la conversione dei componenti in formato JSON per la persistenza dei dati, in quanto componenti di natura diversa richiedono un diverso numero di informazioni per poter essere caricati dal programma (per esempio i componenti di tipo logicGate necessitano della tabella di verità).

Un altro importante utilizzo del polimorfismo è il metodo virtuale “updateState” che se chiamato su un componente di tipo lampadina dovrà controllare l’input e cambiare il proprio stato, se chiamato su logicGate dovrà analizzare la tabella di verità e aggiornare lo stato dei pin di output, richiamando l’updateState sui componenti ad essi collegati.

Persistenza dei dati

La persistenza dei dati avviene tramite l’apposita classe “dataPersistance” la quale gestisce il caricamento da file e il salvataggio su file dei componenti standard.
Caricamento e salvataggio avvengono rispettivamente all’inizio e alla fine dell’esecuzione del programma in maniera automatica e il formato utilizzato per lo scambio dei dati è il JSON.

Il file “savedComponents.json” è l’attuale riferimento per il caricamento e il salvataggio del programma ed è predisposto per contenere una serie di componenti standard “di partenza”:

“interruttore standard” è una normale costruzione della classe button, “lampadina standard” è una normale costruzione della classe lamp, l’AND, l’OR e il NOT sono costruzioni della calasse logicGate con le relative tabelle di verità che simulano il comportamento degli operatori “and”, “or” e “not” della logica booleana.

Funzionalità implementate

Funzionalità legate al modello logico presenti nel progetto:

- la possibilità di creare componenti standard e da essi poter generare componenti “istanziati” in grado di creare collegamenti tra i loro pin in modo da propagare correttamente i segnali emessi inizialmente dagli interruttori
- gli interruttori possono essere “premuti” tramite l’apposito metodo che automaticamente chiamerà a cascata metodi “updateState” che modificheranno correttamente lo stato dei pin collegati al circuito ed eventualmente le lampadine poste alla fine di esso.
- I componenti di tipo logicGate possiedono un controllo maggiore in fase di “updateState” che tiene traccia di tutti i logicGate già aggiornati e previene quindi loop ricorsivi
- la possibilità di inserire,rimuovere e cercare uno specifico componente nella struttura dati creata appositamente, il quale focus è quello di organizzare tutti i componenti distinguendoli tra standard e non
- il sistema di persistenza dei dati su file in formato JSON
- è disponibile un set di metodi (tra cui overloading degli operatori di output) per stampare tutte le informazioni utili al monitoraggio della struttura dati e dei suoi elementi (molto utilizzati nelle fasi di test).
- Possibilità di “personalizzare” i componenti creati modificandone nomi e descrizioni

(Gli ultimi due punti descrivono features attualmente strettamente legate al modello logico e non all’interfaccia grafica)

Funzionalità grafiche presenti nel progetto:

(quando si fa riferimento a click del mouse si intende un click del tasto sinistro del mouse)

- la possibilità di generare componenti di varia natura premendo i relativi riquadri informativi presenti in una barra laterale a sinistra dello schermo. Tali componenti verranno generati con i loro relativi pin: i pin di input sono situati nella parte sinistra del componente, i pin di output sono invece situati nella parte destra
- la possibilità di selezionare con il mouse un elemento precedentemente generato per poterne visualizzare le informazioni in una barra laterale a destra dello schermo
- il cursore del muse cambia in base a dove si trova e in caso venga premuto/rilasciato
- la possibilità di eliminare un elemento selezionato premendo sul tasto “elimina componente” nella barra destra
- la possibilità di eliminare un elemento selezionato premendo il tasto backspace della tastiera
- le barre laterali (quella sinistra relativa ai componenti standard e quella destra relativa alle informazioni dei componenti) possono essere nascose/mostrate premendo il tasto adiacente a ciascuna barra
- l’area di lavoro è navigabile spostandola tenendo premuto il mouse
- gli elementi generati nell’area di lavoro sono in essa spostabili tenendo premuto il mouse su di essi (drag and drop). Inoltre l’elemento che sta venendo spostato acquisisce temporaneamente la priorità a livello grafico, quindi nel caso venga sovrapposto ad altri elementi esso verrà mostrato sopra di essi
- sia nella generazione che nello spostamento dei componenti nell’area di lavoro, avvengono verifiche che impediscono a due componenti di venire rilasciati sulla stessa area sovrapponendosi, in tale evenienza l’ultimo elemento rilasciato viene spostato nella prima area libera sottostante
- i componenti di tipo button (gli interruttori) possono cambiare il loro stato da accesi a spenti e viceversa con un doppio click del mouse su di essi e questa azione comporterà un cambiamento grafico di tale componente al fine di riconoscerne lo stato
- le lampadine cambiano colore in caso di modifica del loro stato, così da renderlo intuitivo
- premendo con il mouse su un pin, da esso verrà generato un cavo collegato al cursore del mouse fintanto che non si verifica un altro click. Se il secondo click avviene su un pin valido (un collegamento tra due pin di input o tra due pin di output non è valido) allora l’estremità del cavo precedentemente collegata al cursore del mouse viene collegata a tale pin; se il secondo click avviene in una qualsiasi altra posizione, il cavo precedentemente generato viene eliminato
- relativamente alla generazione di cavi: se un click del mouse avviene su un pin di input già collegato ad un cavo (i pin di input non possono essere collegati a più cavi contemporaneamente) allora il cavo precedentemente collegato verrà eliminato per essere sostituito da quello nuovo. Questo implica che se si desidera eliminare un cavo collegato tra due pin, bisogna fare un doppio click sul pin di input collegato ad esso.

- nel caso in cui un componente collegato a cavi venga eliminato, verranno eliminati anche tutti i cavi a lui collegati

Rendicontazione ore

Attività	Ore Previste	Ore Effettive
Studio e progettazione	10	5
Sviluppo del codice del modello	15	20
Studio del framework Qt	10	10
Sviluppo del codice della GUI	15	20
Test e debug	5	8
Stesura della relazione	5	4
totale	60	**67**

Il monte ore è stato leggermente superato poiché, non conoscendo bene il framework Qt, lo sviluppo della GUI si è rivelato più impegnativo di quanto preventivato, richiedendo diverso tempo per capire come implementare le varie features grafiche.