**Università degli Studi di Brescia**

2011/2012

**Diagnosi minimali con mascheramento comprensive degli ingressi di un sistema**

**Elaborato di Ingegneria del Software**

**Patrizio Bertoni, Massimo Bono, Michael Maghella**

Sommario

[GLOSSARIO DEI TERMINI DI DOMINIO 2](#_Toc311725241)

[GLOSSARIO DEI TERMINI DI INTERFACCIA 3](#_Toc311725242)

[REGOLE DEL MODELLO 3](#_Toc311725243)

[Definizioni 3](#_Toc311725244)

[Lista dei Warning 4](#_Toc311725245)

[Lista degli Errori 4](#_Toc311725246)

# PREFAZIONE SUL GLOSSARIO

Nello sviluppo dell’elaborato, si è deciso di mantenere un lessico bilingua ( Italiano, Inglese ) seppur in modo isolato a seconda del contesto e della produzione. In particolare, si è deciso di utilizzare:

Inglese per il codice sorgente e i diagrammi di specifica UML, ovvero il materiale che possa un giorno essere condiviso e modificato a fini di progettazione da terzi;

Italiano per la documentazione stessa, il manuale, la JavaDoc, i casi d’uso, l’interfaccia con l’utente e in generale ogni elemento discorsivo.

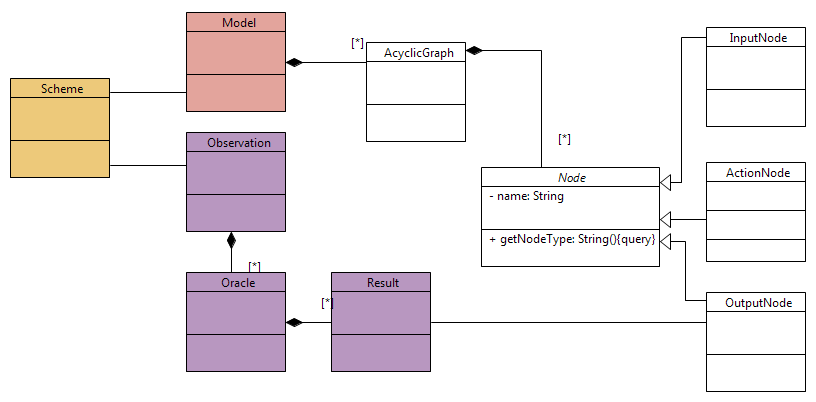
Si lascia al lettore con l’autorità di accedere ad entrambe le tipologie dei contenuti la responsabilità della traduzione ed associazione dei concetti e termini del glossario.

# GLOSSARIO DEI TERMINI DI DOMINIO

1. Schema insieme { Modello , Osservazione }
   1. Modello grafo disconnesso, unione di zero o più Grafi .
      1. Grafo grafo connesso aciclico di Forme, connesse da Flussi .
         1. Nodo
            1. nodoIngresso
            2. nodoUscita
            3. nodoAzione
         2. Flusso link tra due Nodi distinti, disciplinato dalle [Regole](#_REGOLE_DEL_MODELLO).
   2. Osservazione vettore di Oracoli, ognuno associato a un singolo Grafo .
      1. Oracolo vettore di Risultati, ognuno associato a un singolo nodoUscita .
         1. Risultato valore appartenente al dominio {“OK”,”KO”,”OKM”} .

*Note: il nome “Oracolo” è stato deprecato nell’implementazione, esiste solo per coerenza rispetto alla specifica di connettività.*

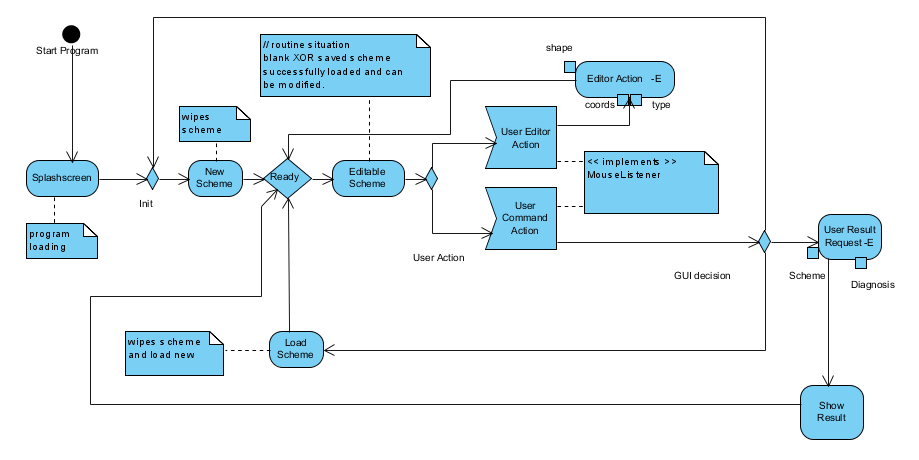
Le definizioni appena viste hanno una rappresentazione equivalente nel seguente diagramma delle classi UML, utilizzato in prospettiva concettuale.



1. Azioni
   1. di Comando insieme:

* Nuovo Schema
* Open Schema
* Salva Schema
* Salva Schema Come..
* Calcola Diagnosi Minimali
  1. di Modifica al Modello insieme:
* nuovo NodoIngresso
* nuovo NodoUscita
* nuovo NodoAzione
* nuovo Flusso
* cancella Nodo
  1. di Modifica all’Osservazione relativo a ogni singolo Risultato, insieme:
* settaOK
* settaKO
* settaOKM

Il seguente diagramma di attività UML rende un’idea del funzionamento delle azioni descritte.



# GLOSSARIO DEI TERMINI DI INTERFACCIA

Vista del Modello: area della finestra del programma dedicata alla rappresentazione e alla modifica del Modello.

Vista dell’Osservazione: area della finestra del programma dedicata alla rappresentazione e alla modifica dell’Osservazione.

Pannello di Comando: area della finestra del programma dedicata alla scelta delle azioni di Modifica di Modello, azioni di Comando, varie (help, menu)

# REGOLE DEL MODELLO

che disciplinano il modo in cui i vari tipi di nodi possono interagire tra loro.

## Definizioni

Si definisce un MODELLO ERRATO un modello virtuale ( quindi relativo a uno stato che non deve essere mai raggiunto dalla struttura dati ) un modello che presenta almeno un errore e zero o più warnings.

*Intuitivamente un modello ERRATO è un modello in cui è presente almeno un’incoerenza che, eccetto l’eliminazione della causa dell’incoerenza stessa, non può essere in alcun modo risolta dall’utente.*

Si definisce uno MODELLO WARNING un modello che NON è ERRATO ma che presenta un numero non nullo di warning attivi.

Uno modello WARNING rappresenta un sistema descritto da una struttura dati non sicura, cioé una struttura dati consentita ma che non permette il calcolo della diagnosi.

*Ad esempio all’inizio l’utente vuole inserire un NODO d’azione; l’interprete consente l’azione ma avvisa che tale struttura non può essere calcolata in quanto un NODO d’azione deve essere parte integrante di un GRAFO ( lo schema non è corretto)*

*Intuitivamente un modello WARNING è un modello in cui è presente almeno un’incoerenza che può essere risolta dall’utente con una particolare serie di azioni oltre l’eliminazione della causa del warning stesso.*

Si definisce uno MODELLO CORRETTO un modello che non è un modello ERRATO né WARNING.

Tale modello è dunque l’unico che permette il calcolo della diagnosi.

## Regole di Warning

Il Modello è Warning se è vera almeno una delle seguenti:

1. NODI
2. NON ESISTE ALCUN INGRESSO.
3. NON ESISTE ALCUNA AZIONE.
4. NON ESISTE ALCUNA USCITA.
5. UN INGRESSO NON HA ALCUN FLUSSO USCENTE.
6. UN USCITA NON HA ALCUN FLUSSO ENTRANTE.
7. UN AZIONE NON HA ALCUN FLUSSO ENTRANTE
8. UN AZIONE NON HA ALCUN FLUSSO USCENTE.

## Regole di Errore

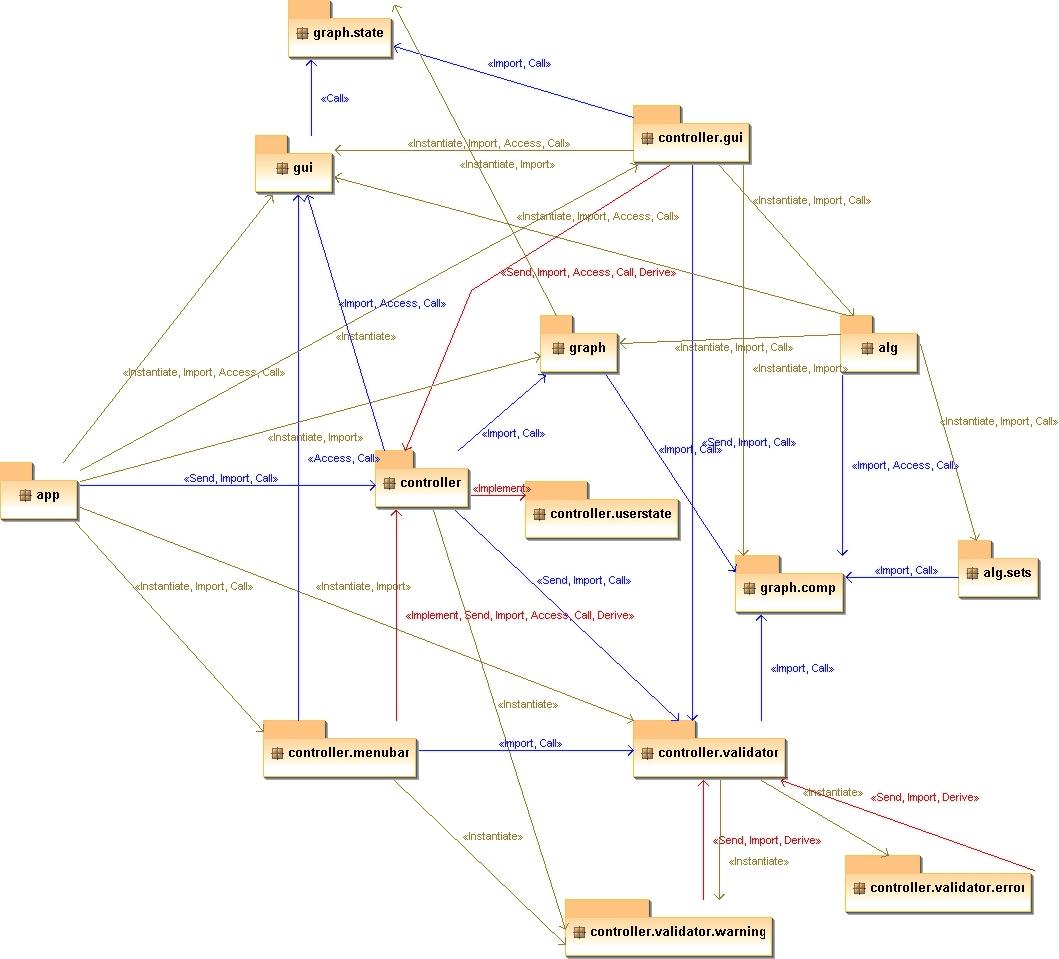
Il Modello è Errato se è vera almeno una delle seguenti:

1. NODI
   1. UN INGRESSO HA ALMENO UN FLUSSO ENTRANTE.
   2. UN USCITA HA ALMENO UN FLUSSO USCENTE.
   3. UN USCITA HA PIÚ DI UN FLUSSO ENTRANTE.
2. FLUSSI
   1. SE FLUSSI IN USCITA DA AZIONE OR DA INGRESSO SONO PIU' D'UNO , CIASCUNO DI ESSI HA COME DESTINAZIONE UN NODO DISTINTO
   2. UN FLUSSO NON PUÓ COLLEGARE DIRETTAMENTE UN INGRESSO CON UN USCITA
   3. UN AZIONE COLLEGATA A UN USCITA NON HA ALTRI FLUSSI USCENTI
   4. IL GRAFO E’ SEMPRE ACICLICO

# **DESCRIZIONE DEI PACKAGE**

Nota bene: i seguenti diagrammi UML delle classi e dei package sono disponibili in formato raster e vettoriale sul cd dell’elaborato, per una migliore consultazione. Gli screenshot qui presenti sono da intendersi a scopo illustrativo.

## Diagramma UML dei Package



## app

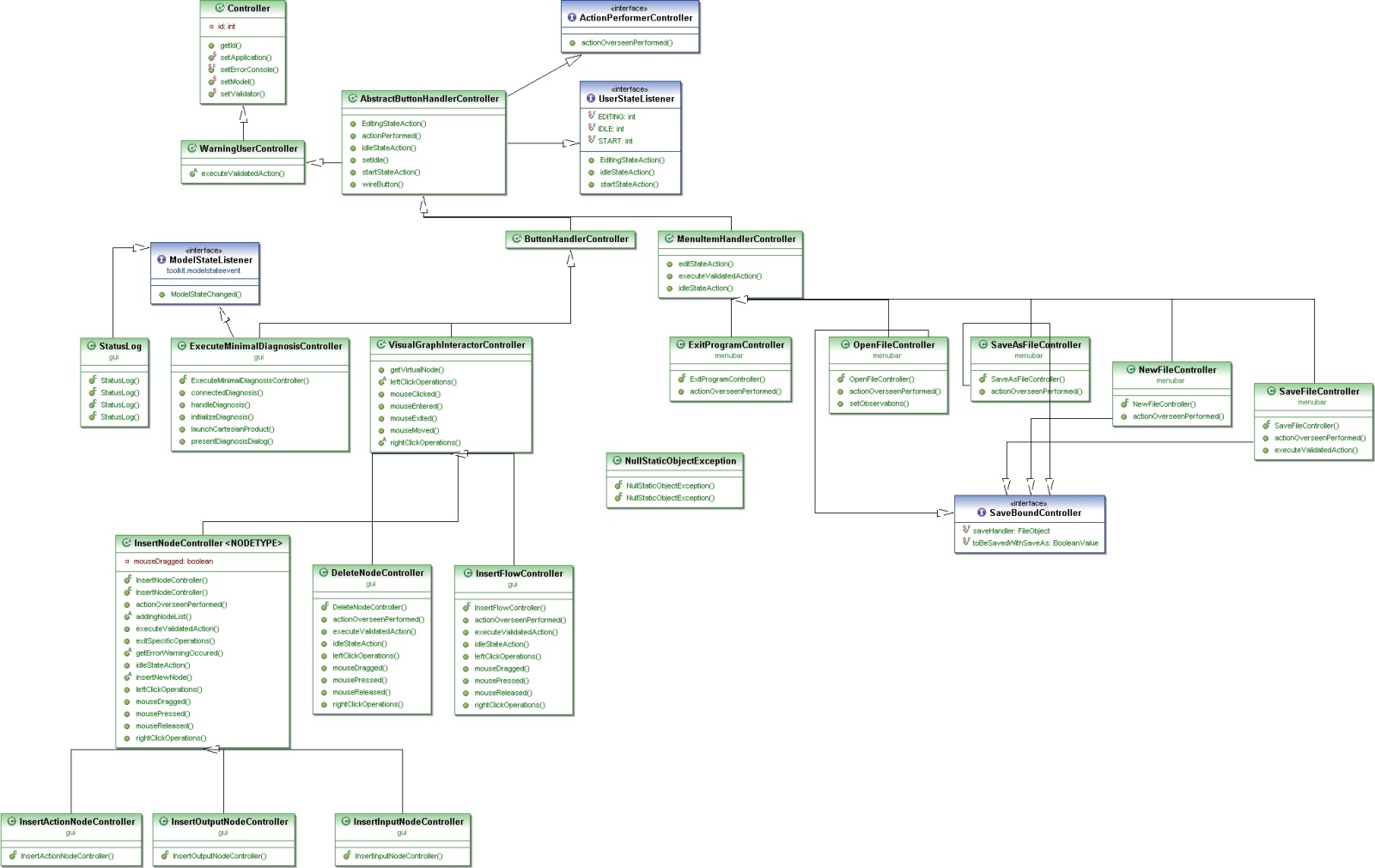
Contiene solamente la classe “driver”, l’applicazione che crea e inizializza i moduli fondamentali del programma:

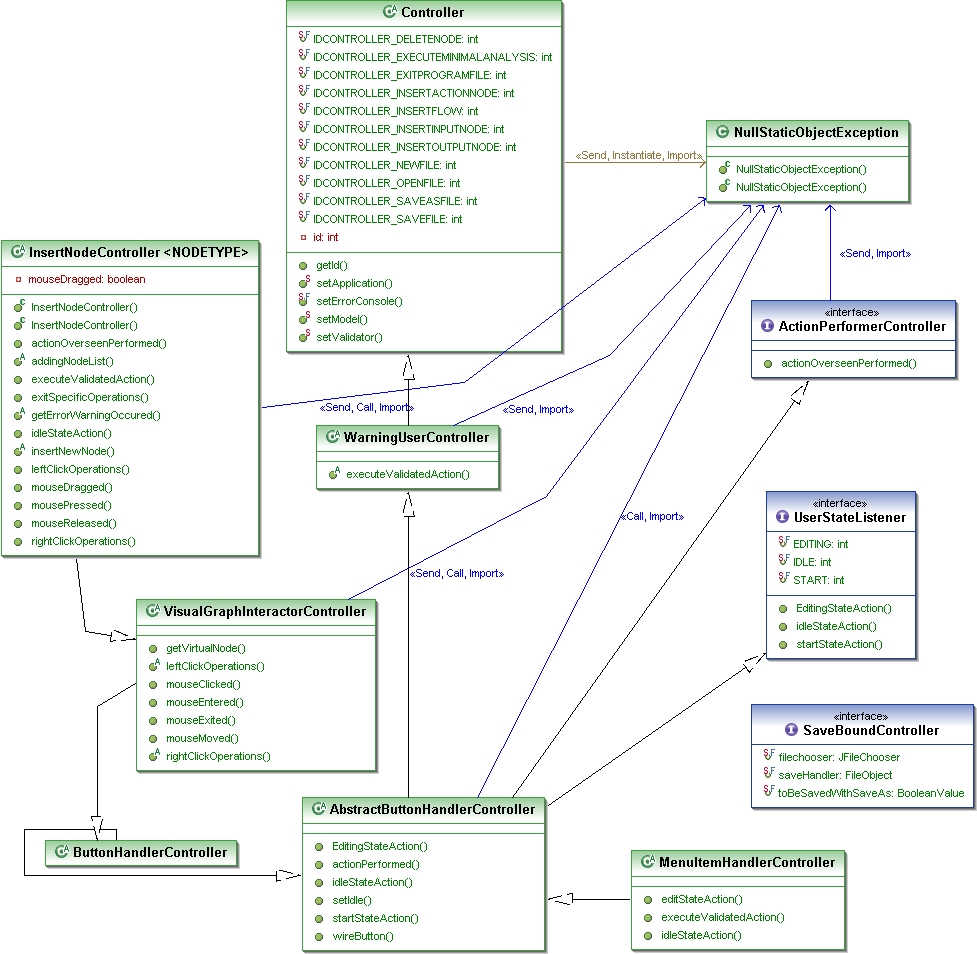
* Model l’oggetto che rappresenta il modello in analisi.
* GUI la graphic user interface scelta per l’interazione con l’utente. Altre implementazioni del problema possono prendere in considerazione interfacce di tipo console; vi è un pacchetto vuoto preposto, per future implementazioni.
* Validator vedere la [specifica](#_controller.validator).

## **controller**

Si occupa dell’operazione da eseguire sotto gli stimoli della GUI. Modifica lo Schema e/o la GUI stessa. Esempi:

* *La pressione del tasto GUI ‘crea nodo’ genera un evento che viene catturato da una classe specifica di controller che implementi listener.  
  Questa elabora una risposta specifica per l’oggetto ascoltato, che potrà opzionalmente coinvolgere metodi di classi nel package graph (come la creazione di un nuovo nodo).*
* *La pressione del tasto GUI ‘elimina nodo’ genera un evento che viene catturato da una classe specifica di controller che implementi listener.  
  Questa una risposta specifica per l’oggetto ascoltato, che potrebbe essere costituita da  
   un Jframe che messaggia l’utente con “scrivi il nodo da eliminare”  
  l’utente sceglie il nodo da eliminare (come?)  
  il controller rileva questa selezione e può interrogare il validator sulla consistenza del nuovo modello.*



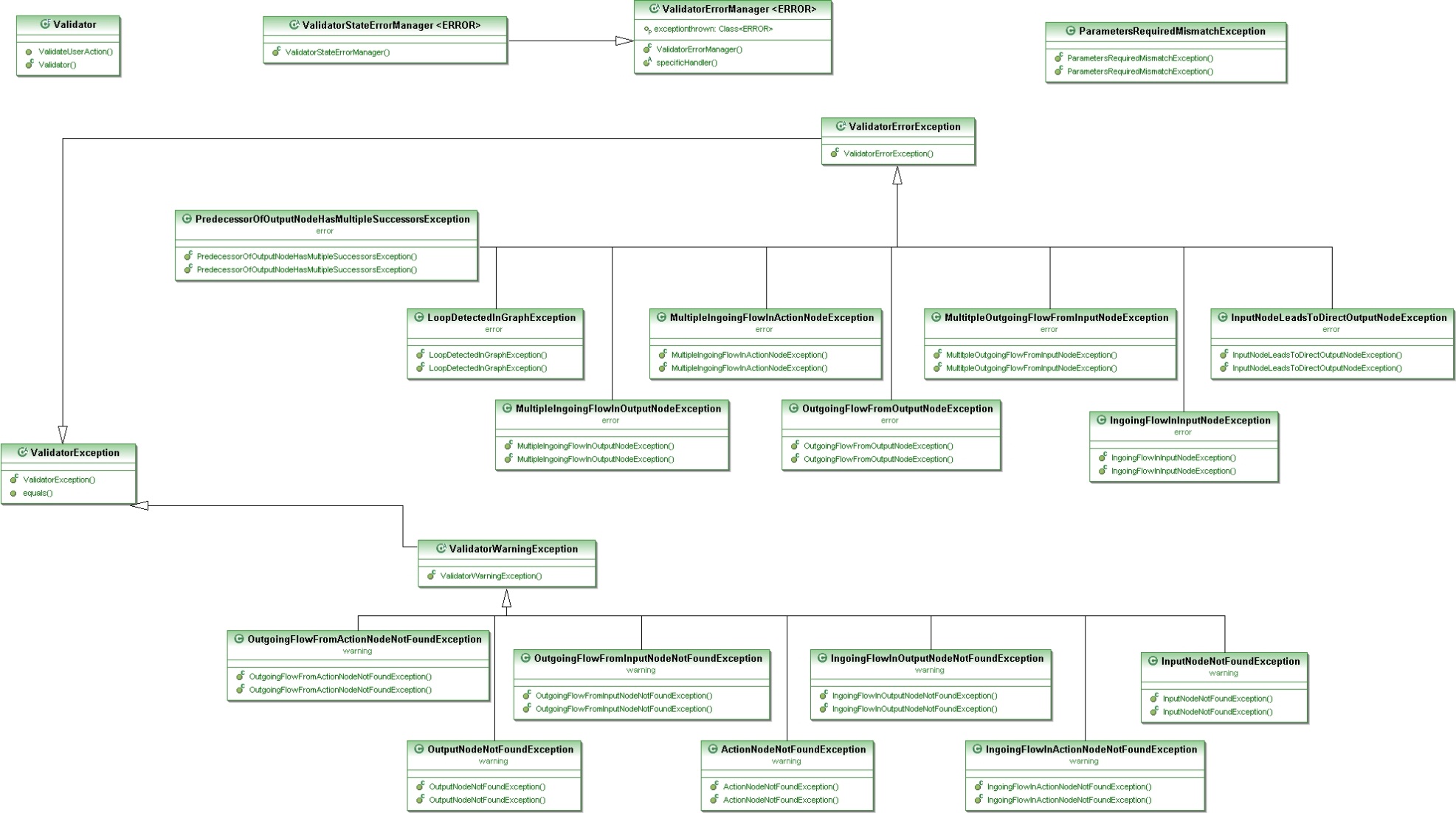
**

### controller.validator

Gestisce il controllo di validità del sistema a seguito di una modifica del Modello fatta dall’utente, in accordo con le asserzioni specificate nel paragrafo [Regole](#_REGOLE_DEL_MODELLO).

Le sue classi vengono richiamate dalle classi Controller quando un’azione GUI deve essere verificata.

A seconda della tipologia di azione editing dell’utente, il Controller pilota il Validator passandogli la serie di controlli che devono essere eseguiti contestualmente all’azione intrapresa. Il Validator dunque come prima cosa copia l’istanza di grafo passatogli e su di essa esegue la modifica utente, a prescindere dalla sua correttezza. A questo punto vengono eseguiti tutti i controlli passatigli dalle classi controller in modo vettorizzato. (chiavi id). Ogni controllo ritorna un booleano, e la presenza di almeno un errore genererà lo stato errore, che verrà comunicato all’utente. La copia del grafo verrà dunque distrutta.

Se invece non si ha nessun errore, il Validator restituirà il nuovo grafo, valido o in warning.

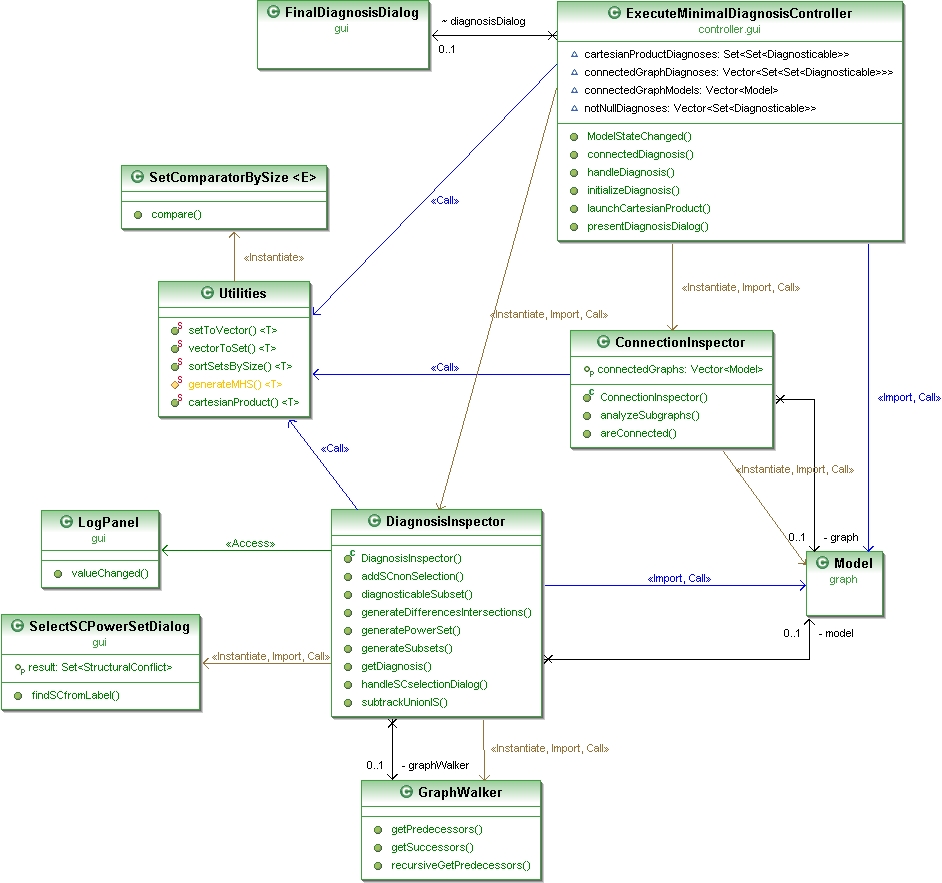
## alg

Utilizza il package jgrapht.alg, contenente algoritmi attinenti alla teoria dei grafi, per sviluppare l’implementazione del problema di diagnosi minimale con mascheramento comprensiva degli ingressi.  
I metodi di uso generale sono stati resi statici, parametrici e refactorizzati verso la classe Utilities.

* *versione con prodotto cartesiano tra grafi disconnessi (sperimentale)*  
   Ogni Grafo appartenente al Modello deve costruire, nell’ExecuteMinimalDiagnosisController, un’istanza della classe DiagnosisInspector, delegata alla generazione delle sue proprie Diagnosi. Ottenuti tutti i relativi insiemi di diagnosi, ne verrà fatto il prodotto cartesiano.
* *Versione regredita al caso generico (realmente implementata)*

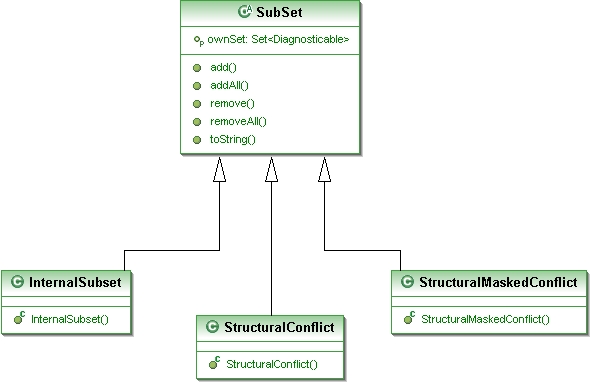
Il Modello, anche se disconnesso, viene interpretato come un Grafo unico. Allora esiste un solo  
DiagnosisInspector, ove rientrano tutte e sole le Diagnosi previste.

Nella progettazione del package, la sua natura algoritmica e multi-purpose (il problema dei Minimal Hitting Set potrebbe essere comune a moltissime applicazioni) ha spinto a parametrizzare ove possibile il codice.



### alg.sets

Contiene le strutture che modellano i sottoinsiemi cardine dell’algoritmo: InternalSubset, StructuralConflict e StructuralMaskedConflict. Ereditano da una classe astratta in comune, SubSet.



## console

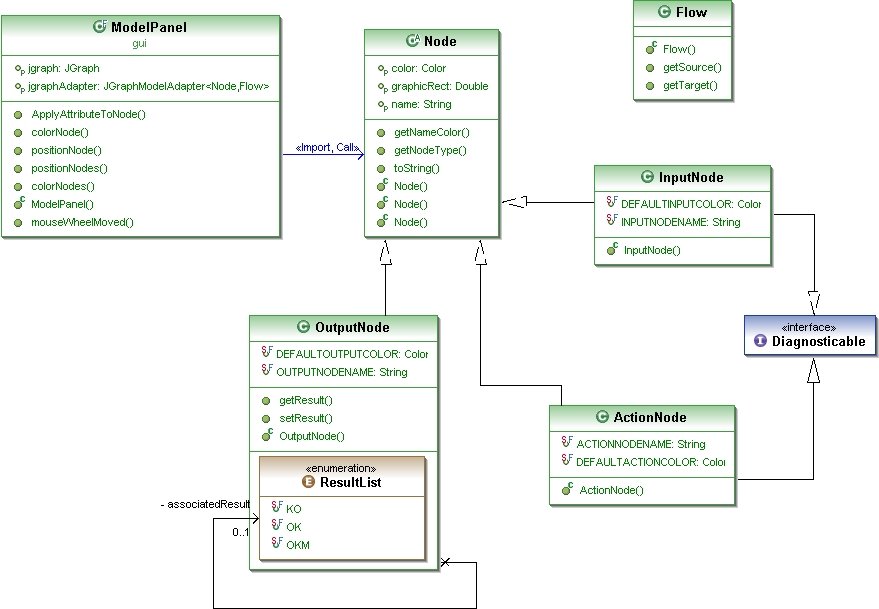
Package attualmente vuoto, suggerisce un futuro embedding delle classi di logica in una Console User Interface, anziché una GUI.

## graph

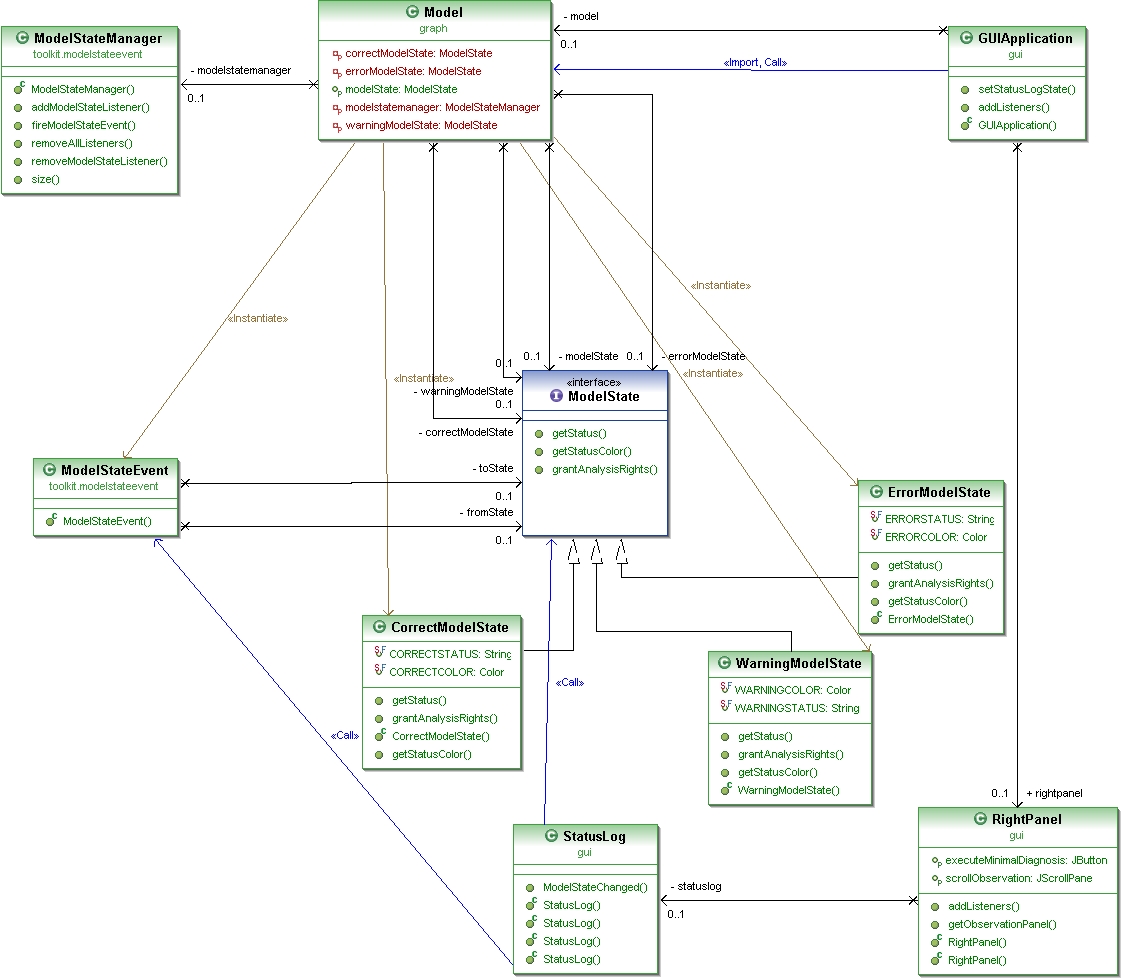
Interfaccia i packages nativi dell’elaborato con quelli forniti da [jgraph e jgrapht](#_JGRAPH_E_JGRAPHT:).  
In graph è contenuto l’oggetto Model utilizzato dal software. Esso descrive completamente il Modello, che ricordiamo nella teoria dei Grafi essere un *grafo diretto aciclico*. Implementa la gerarchia di JgraphT, ed è l’oggetto effettivamente salvato e caricato in quegli scenari d’uso.

### graph.comp

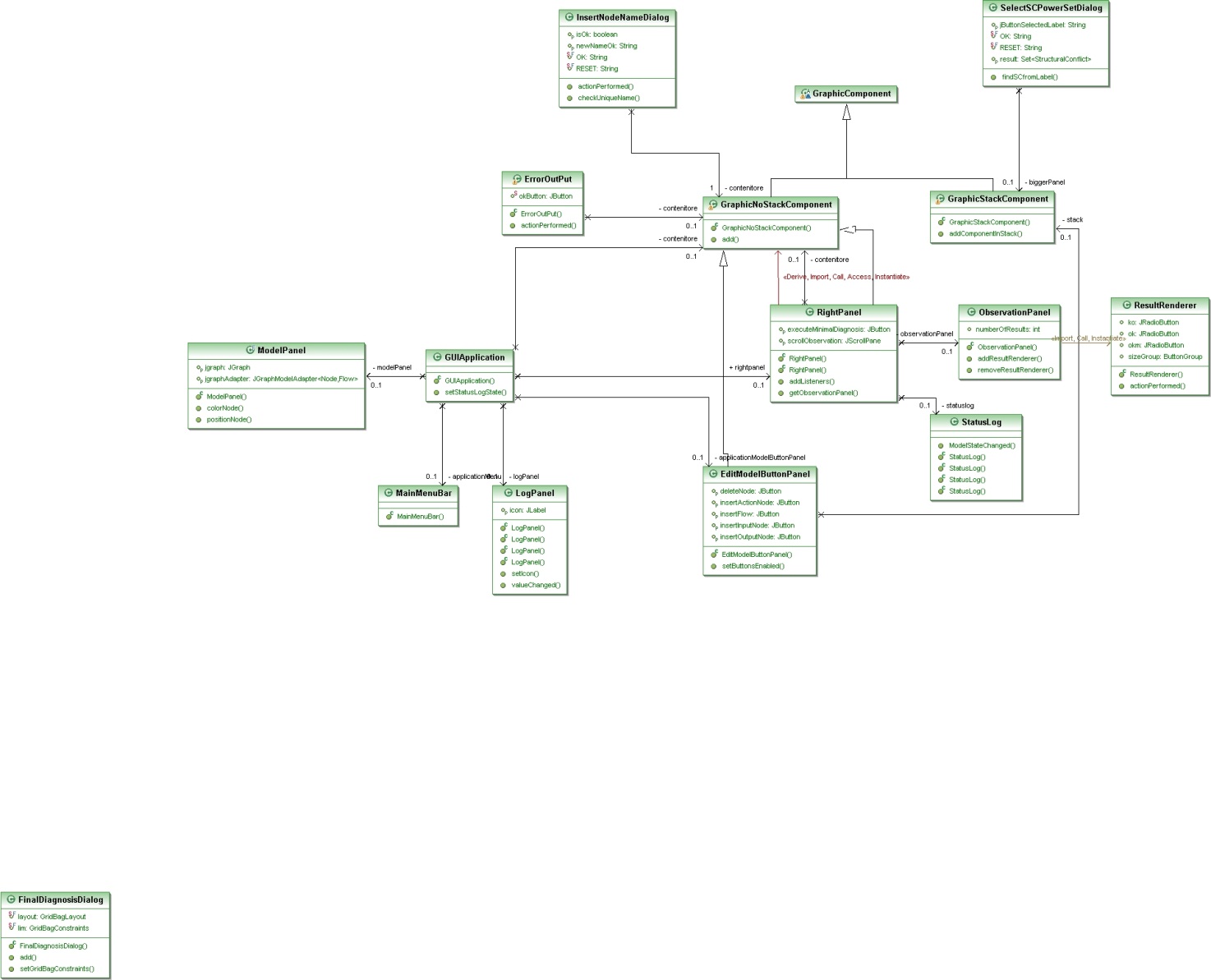
Contiene le classi che rappresentano i componenti del modello: i nodi d’ingresso, d’azione, d’uscita, e i flussi che li collegano. Il ModelPanel, l’oggetto grafico che descrive il Model, interagisce con essi per disegnarli correttamente.



### graph.state

Implementa il Design Pattern [State](#_State). Un Modello possiede contemporaneamente un’istanza per ogni stato possibile della FSM atta a descriverlo, oltre a un’istanza contente lo stato effettivo dello stesso.  
Notare che lo stato Error è virtuale, perché non è mai raggiunto dal Modello vero (potrebbe essere raggiunto da una copia temporanea del Modello preposta alla verifica di validità).

## gui

Implementa l’interfaccia grafica a finestre con l’utente, e fornisce comandi visivi per le azioni sullo schema.

# JGRAPH E JGRAPHT: DUE POTENTI RISORSE

JgraphT e Jgraph sono due differenti librerie software, distribuite sotto licenza GNU e intese per scopi diversi.

JgraphT è centralizzata su strutture dati e algoritmi;  
Jgraph si occupa del rendering e dell’editing guidato dalla GUI.

Le due librerie sono in questo senso complementari, sebbene Jgraph sia antecedente a JgraphT, e possono essere usate insieme tramite un oggetto JgraphModelAdapter fornito da JgraphT. Questa classe fornisce i servizi che permettono a un grafo JgraphT di essere usato come il modello visto e controllato da Jgraph.  
Le modifiche effettuate sulla struttura JgraphT sottostante sono propagate al modello Jgraph se e solo se il grafo JgraphT è un grafo di tipo Listenable.  
La comodità di JgraphT è la sua natura parametrica, che permette di utilizzare come Nodi e Flussi (nel pacchetto chiamati rispettivamente Vertex e Edges) qualsiasi coppia di classe si desideri.  
Si è quindi potuto mantenere il livello logico concettualmente ben distinto dal codice JgraphT e ancor più dal codice grafico Jgraph; questa è stata una condizione imprescindibile nella scelta di appoggio a software esterno, che non condizionasse le scelte progettuali.

# SCELTA DEL MODELLO DI PROGETTO

E’ stato adottato un metamodello a spirale per definire un processo proprio del gruppo di lavoro, nuovo al metodo ingegneristico applicato alla progettazione del software.   
Dopo una grande fase di brainstorming iniziale, si è passati a stendere gli Use Case per familiarizzare col rapporto che si desiderava creare con l’esterno, e successivamente una prima serie di diagrammi UML: classi e attività. Sebbene sia prioritario progettar la logica prima dell’interfaccia all’esterno, la spesa di qualche ora per parlar dei casi d’uso ha motivato fortemente il gruppo.  
La lettura del codice offerto da [Jgraph](#_JGRAPH_E_JGRAPHT:) ha imposto uno stop, per un’attenta verifica del suo possibile utilizzo, senza che fosse persa nessuna libertà di progetto degli ingegneri del software. La conferma è stata positiva: le classi di logica si appoggiano a Jgraph, ma sono state parametrizzate per essere più versatili possibile.  
Procedendo nella scrittura del codice, si sono rivalutate molte scelte, che han portato a redigere un secondo insieme di diagrammi UML, mentre i casi d’uso sono rimasti pressochè invariati.  
La successiva fase implementativa ha visto sincronizzarsi il lavoro degli ingegneri del sofware, che ha goduto di un buon livello di indipendenza grazie all’appoggio presso un SVN (un repository remoto).  
La seconda voluta della spirale è stata costituita dal [refactoring](#_REFACTORING) e dalla manutenzione perfettiva.

# SPECIFICHE

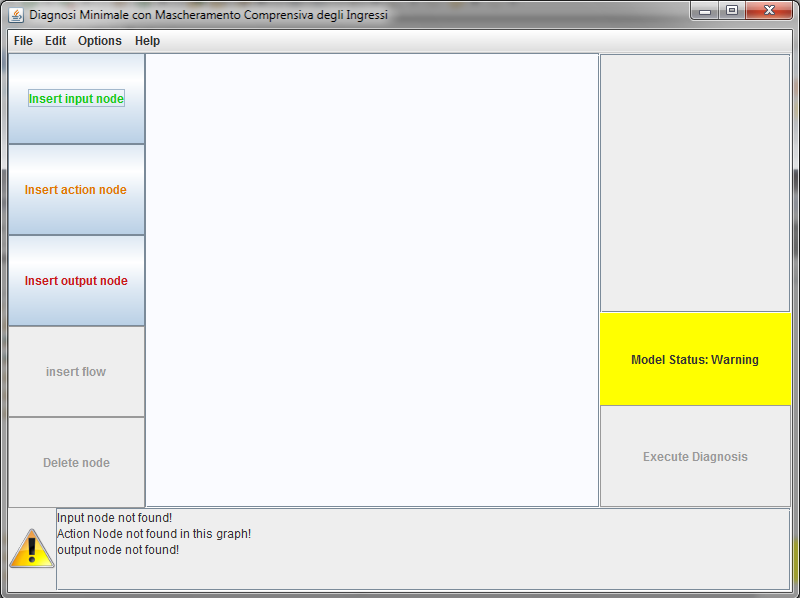
Sono state inizialmente definite le *specifiche del software,* ovvero la descrizione accurata del comportamento del sistema verso l’ambiente esterno. Tali specifiche sono state verificate rispetto ai requisiti utente e funzionali (ovvero confrontate con la consegna fornita).  
Successivamente sono state stese le *specifiche di design,* riguardanti l’architettura del software vera e propria. Essa è stata verificata rispetto alle specifiche del software.

# CASI D’USO

Si veda il pdf “Casi d’Uso” allegato a questa Documentazione.

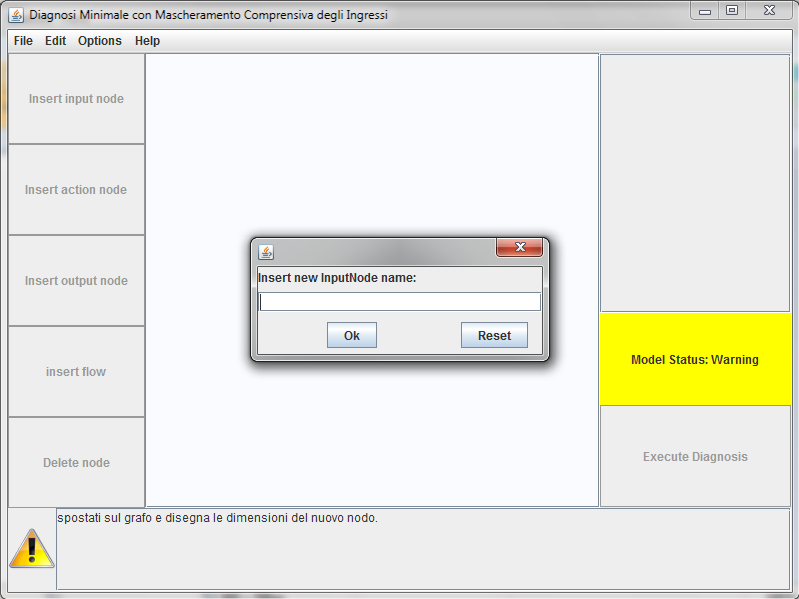
# MANUALE DI UTILIZZO DEL SOFTWARE

*Schermata iniziale del programma*



## Inserire un Input/Action/Output Node

Cliccare su Insert InputNode/ Insert ActionNode/ Insert OutputNode, sulla toolbar a sinistra. Cliccare con mouse sx su un qualsiasi punto nella vista grafica. Rilasciare se si vuole un nodo di dimensioni standard; tenere premuto e trascinare per definire una dimensione propria.



Inserire un nominativo univoco per il nuovo nodo. In caso il nominativo scelto non sia univoco, un messaggio arresterà l’operazione finché non verrà inserito uno valido o l’azione venga abortita chiudendo la finestra di dialogo.

## Cancellare un Node

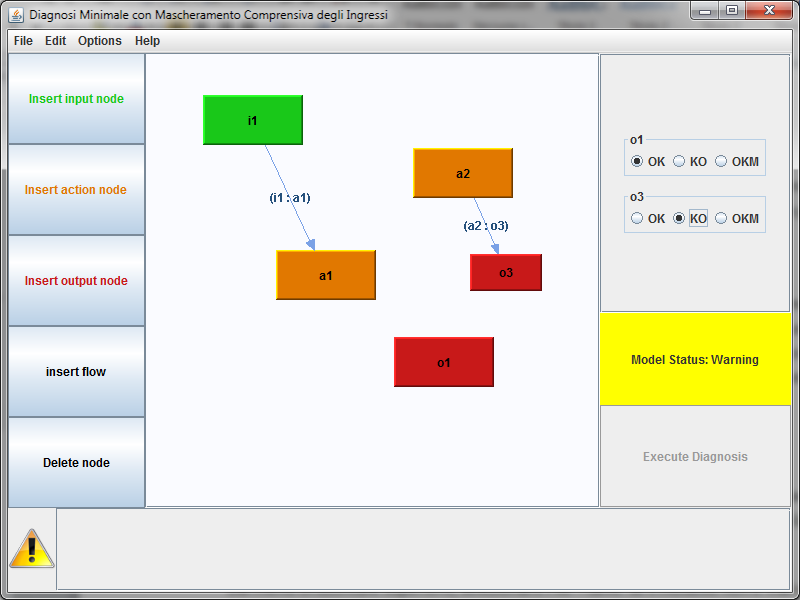
Cliccare su Delete Node, nella toolbar a sinistra. Cliccare con mouse sx su un qualsiasi Node nella vista grafica. Se invece si desidera abortire la cancellazione, cliccare con mouse dx su un qualsiasi punto della vista grafica. Attenzione: i flussi direttamente connessi al Node verranno eliminati insieme ad esso.  
Notare che l’azione è impossibile se nel Modello sono presenti zero Nodes.

## Inserire un Flow tra due Nodes

Cliccare su Insert Flow, sulla toolbar a sinistra. Selezionare il Node che sarà sorgente del Flow.  
Selezionare il Node che sarà destinazione del Flow.   
Se invece si desidera abortire l’inserimento, cliccare con mouse dx su un qualsiasi punto della vista grafica, in ogni istante.  
 L’inserimento deve avvenire in accordo alle [Regole](#_REGOLE_DEL_MODELLO).  
Notare che l’azione è impossibile se nel Modello sono presenti meno di due Nodes.

## Modificare un’Osservazione

Nel pannello a destra selezionare il bottone associato allo stato d’uscita desiderato, nel radio group relativo all’OutputNode che si vuole modificare. Notare che questa modifica è invisibile poiché impossibile, se non si è prima inserito alcun OutputNode.



## Ottenere delle Diagnosi

Cliccare su Execute Diagnosis nel pannello a destra. Se nel modello esistono almeno due OutputNodes aventi stato “KO”, verrà presentato un dialogo dove scegliere una sola scelta tra le possibili: queste si riferiscono a insiemi di OutputNodes “KO” diversi richiesti al punto 2) dell’algoritmo. Se si ha un solo OutputNode, questo passo non viene presentato. Verranno poi presentate all’utente le Diagnosi Minimali.  
Notare che l’azione è impossibile se lo stato del modello non è “Correct”.

## Creare un nuovo Schema

Dal menù orizzontale superiore, selezionare File->New. Tutti i Nodes/Flows in lavorazione, se non precedentemente salvati, saranno istantaneamente persi e verrà presentata un’area di lavoro vuota.

## Salvare uno Schema

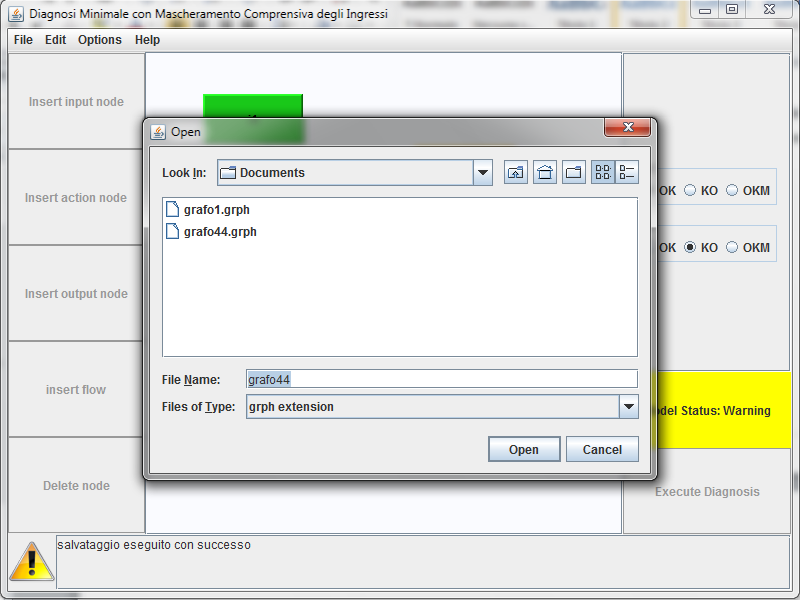
Dal menù orizzontale superiore, selezionare File->Save. Se il modello non è mai stato salvato, avviene una redirezione a [Salvare uno Schema Come](#_Salvare_uno_Schema). Altrimenti, le modifiche correnti al Modello e alle Osservazioni verranno memorizzate nel file che descrive lo Schema.

## Salvare uno Schema Come

Dal menù orizzontale superiore, selezionare File->Save As… . Scegliere all’interno del filesystem una posizione e un nome per il nuovo Schema. Il file verrà salvato con estensione .grph.

## Caricare uno Schema

Dal menù orizzontale superiore, selezionare File->Open . Scegliere all’interno del filesystem una posizione e un nome di uno Schema da caricare. La vista grafica carica lo Schema contenuto in quel file.  
Attenzione: gli unici file visibili dalla form grafica hanno estensione .grph.



# UTILIZZO DI DESIGN PATTERNS

## State

State è un pattern di tipo comportamentale (Behavioural) per modellare degli automi a stati finiti (Finite State Machine). State trova utilizzo nel contesto del Modello, che come prestabilito al paragrafo [Definizioni](#_Definizioni) delle Regole, si trova in ogni istante in uno e uno solo tra gli stati:

* + Correct
  + Warning
  + Error

Appartiene logicamente al package [graph](#_graph), perchè proprietà dell’oggetto Model.

# TESTING

## black box ( gg mm 2012)

L’intera applicazione è stata condivisa con un gruppo di lavoro del tutto all’oscuro dell’implementazione della medesima. I risultati sono stati <completare>

## white box (14 maggio 2012)

L’applicazione è stata testata dal gruppo di sviluppatori per ogni singolo caso d’uso descritto. Il testing si riferisce alla versione 1.3 dopo che un po’ di debugging e di [Refactoring](#_REFACTORING) hanno portato il software a uno stato più stabile della versione 1.0.

### **Azioni di Modifica di Schema**

inserimento di un Input/Action/Output Node

* con dimensioni standard: successo.
* con drag’n’drop dall'angolo superiore sinistro all' angolo inferiore destro: successo.
* con drag and drop dall'angolo inferiore destro all' angolo superiore sinistro:   
   successo, inizialmente molto problematico, poi uniformato al caso precedente.
* con drag oltre il limite della grafica:   
   ci si aspetta che il Model Panel attivi una scrollbar per osservare tutta la finestra: successo.
* in modo che sia sovrapposto ad un altro Nodo:   
   ci si aspetta che si comporti in modo trasparente: successo.
* con un nome già inserito nel Modello:   
   ci si aspetta che non venga aggiunto e che sia presentata una finestra di errore: successo.
* chiusura della finestra di dialogo per l’ottenimento del nuovo nome, nuovo click sx sul ModelPanel e verifica che non venga richiesto di reinserire un nodo: successo.
* verifica della cancellazione del nome mediante il tasto reset, dopo di che completamento inserimento del nome e verifica che il Nodo abbia effettivamente il nome voluto: successo.
* inserimento di un nome non contenibile nelle dimensioni del Nodo (siano di default o draggate):   
   ci si aspetta che il nodo conservi le giuste dimensioni e che il nome venga troncato con dei   
   puntini: successo. Nota Bene, questa funzionalità è implementata da [JGraph](#_JGRAPH_E_JGRAPHT:).
* modifica di Osservazione durante l’attesa di mouse sx per posizione e dimensione Nodo (sia default che drag’n’drop): verifica che la modifica di osservazione venga completata, sostanzialmente che i due flussi di esecuzione non siano concorrenti: successo.
* abort dell’inserimento con click dx sulla Vista del Modello in qualunque istante: successo.
* aggiunta dell'osservazione relativa all'OutputNode appena inserito (in qualunque dei casi di test precedenti) all'ObservationPanel, con stato di default “OK”: successo.

#### inserimento di un flusso

* da InputNode ad ActionNode: successo.
* da ActionNode ad OutputNode: successo.
* da InputNode ad OutputNode, ci si aspetta che non sia consentito (in accordo alle [Regole](#_REGOLE_DEL_MODELLO)):  
   successo, MA nessun messaggio nel LogPanel che segnali l’errore: rimane scritto “clicca sul  
   secondo nodo”
* da ActionNode ad InputNode, ci si aspetta che non sia consentito (in accordo alle [Regole](#_REGOLE_DEL_MODELLO)):  
   successo, MA nessun messaggio nel LogPanel che segnali l’errore: rimane scritto “clicca sul  
   secondo nodo”
* da InputNode a diverso InputNode, ci si aspetta che non sia consentito (in accordo alle [Regole](#_REGOLE_DEL_MODELLO)):  
   successo, MA nessun messaggio nel LogPanel che segnali l’errore: rimane scritto “clicca sul  
   secondo nodo”
* autoflusso su InputNode, ci si aspetta che non sia consentito (in accordo alle [Regole](#_REGOLE_DEL_MODELLO)):  
   successo, MA nessun messaggio nel LogPanel che segnali l’errore: rimane scritto “clicca sul  
   secondo nodo”
* autoflusso su ActionNode, ci si aspetta che non sia consentito (in accordo alle [Regole](#_REGOLE_DEL_MODELLO)):  
   successo, MA nessun messaggio nel LogPanel che segnali l’errore: rimane scritto “clicca sul  
   secondo nodo”
* da ActionNode a diverso ActionNode, ci si aspetta che non sia consentito (in accordo alle [Regole](#_REGOLE_DEL_MODELLO)):  
   successo, MA nessun messaggio nel LogPanel che segnali l’errore: rimane scritto “clicca sul  
   secondo nodo”
* da OutputNode a InputNode, ci si aspetta che non sia consentito (in accordo alle [Regole](#_REGOLE_DEL_MODELLO)):  
   successo, MA nessun messaggio nel LogPanel che segnali l’errore: rimane scritto “clicca sul  
   secondo nodo”
* da OutputNode a ActionNode, ci si aspetta che non sia consentito (in accordo alle [Regole](#_REGOLE_DEL_MODELLO)):  
   successo, MA nessun messaggio nel LogPanel che segnali l’errore: rimane scritto “clicca sul  
   secondo nodo”
* flusso che può creare un grafo ciclico, ci si aspetta che non sia consentito (in accordo alle [Regole](#_REGOLE_DEL_MODELLO)):  
   successo, MA nessun messaggio nel LogPanel che segnali l’errore: rimane scritto “clicca sul  
   secondo nodo”
* attesa dell’inserimento di un nuovo punto per il Nodo da inserire (sia sorgente che destinazione),  
  dopo che è stato selezionato un punto vuoto: successo.
* uscita con mouse dx, in ogni istante: successo.
* modifica di osservazione durante l’attesa di mouse sx per selezione targets del Flow: verifica che la modifica di osservazione venga completata, sostanzialmente che i due flussi di esecuzione non siano concorrenti: successo.
* verifica del corretto passaggio dello stato da “Warning” a “Correct”, in accordo alle [Regole](#_REGOLE_DEL_MODELLO): successo.

#### cancellazione di un InputNode/ActionNode/OutputNode

* ci si aspetta che la cancellazione sia strutturale oltre che grafica, e che i Flow direttamente connessi al nodo subiscano la stessa azione: successo.
* ci si aspetta che dopo la cancellazione di un OutputNode venga cancellata l’osservazione associata dall’ObservationPanel: successo.
* modifica di osservazione durante l’attesa di mouse sx per selezione Node da cancellare: verifica che la modifica di osservazione venga completata, sostanzialmente che i due flussi di esecuzione non siano concorrenti: successo.
* attesa dell’inserimento di un nuovo punto per il Nodo da cancellare, dopo che è stato selezionato un punto vuoto: successo.
* uscita con mouse dx, in ogni istante: successo.
* verifica del corretto passaggio dello stato da “Correct” a “Warning”, in accordo alle [Regole](#_REGOLE_DEL_MODELLO): successo.

### Azioni di Comando

#### Menu File

* Exit, dopo la creazione/modifica di Schema (nessun salvataggio è previsto):  
   successo.
* New, dopo la creazione/modifica di Schema (nessun salvataggio è previsto):  
   successo.
* New, dopo il salvataggio di Schema (il file non è sovrascritto):  
   successo.
* disabilitazione del Menu, durante una qualsiasi azione di Modifica del Modello: successo.
* Save As, dopo creazione Schema ci si aspetta che venga visualizzato il dialogo per digitare il nome del file: successo.
* Save, dopo creazione nuovo Schema mai salvato, ci sia aspetta di essere reindirizzati a Save As: successo.
* Save, dopo modifica di Schema precedentemente salvato, ci si aspetta che le modifiche siano state memorizzate: successo.
* Save As, dopo modifica di Schema precedentemente salvato, ci si aspetta che venga aperto il dialogo e si possa salvare un altro file completamente indipendente: successo
* Open, di uno Schema salvato in memoria, ci si aspetta che venga caricato correttamente: successo
* Open, ci si aspetta che eventuali modifiche non salvate dello Schema che era in editazione vengano perse: successo.
* Open, ci si aspetta che l’aggiornamento dello Schema in editazione coinvolga anche la gestione dello Stato (cfr. [Model State](#_State)): successo.
* Diagnosi Minimale, verifica che il pulsante relativo sia enabled se e solo se lo Stato del Modello è corretto: successo.
* Diagnosi Minimale, verifica dell’affidabilità dell’algoritmo:   
   inizialmente l’algoritmo risultava enormemente lento (sull’ordine dei minuti) a causa di un  
   “bubble sort” su una struttura dati piuttosto pesante.   
   Dopo l’implementazione del “quick sort” le performances sono diventate accettabili.  
   Procedendo nel testing è stato chiaro che una voce di ottimizzazione suggerita dalla  
   consegna è stata compresa male e quindi la sua implementazione ha dato luogo ad errori.  
   La regola del prodotto cartesiano tra Diagnosi nel caso di Grafi Sconnessi vale infatti SE E   
   SOLO SE ogni insieme di Diagnosi è non nullo. La parte di logica che gestiva questo (cfr   
   descrizione del package [alg](#_alg)) è stata fatta regredire con successo.

# REFACTORING

## Controller

Nella prima versione dell'albero dei Controller, essi venivano associati ai relativi oggetti nella GUI in modo posizionale .Il costruttore della GUI richiedeva dei generici "Controller", poi associava ad ognuno di essi, in modo posizionale, l'oggetto da pilotare.   
La nuova versione del costruttore GUI ha una lista di Controller, ognuno con il proprio tipo:

da *Controller, Controller, Controller, …* a *NewFileController, OpenFileController, SaveFileController, …*

Il metodo wire(AbstractButton): effettua il binding tra un pulsante grafico e il relativo Controller   
( sostanzialmente richiama addActionListener ), inoltre imposta l’attributo "listenedButton”, che permette di ottenere dal Controller in maniera immediata il pulsante pilotatodal Controller stesso.   
In questo modo la leggibilità del codice aumenta.

La nomenclatura dei metodi nati per compattare il codice lungo tutto l'albero gerarchico dei Controller erano ambigua e poco leggibile per altre risorse umane. Quindi è stato fatto refactoring dei nomi:  
da *buttonaction(ActionEvent)* a *actionOverseenPerformed(ActionEvent)*

Le azioni di Modifica dello Schema, e alcune azioni di Comando (Open, Save) lanciano ognuna un proprio *wizard*, un percorso guidato dove l'utente può eseguire una determinata azione. Il wizard può concludersi anche dopo svariate azioni utente. E’ stata implementata una macchina a stati dell'azione utente, composta da

* idle: l'utente non sta eseguendo alcun wizard
* editing: l'utente sta eseguendo un wizard e quindi non deve essere distratto da altri eventi.

con l'introduzione di questo stato utente è stato possibile attivare/disattivare i pulsanti di azione relativi ad altri wizard per:

* Evitare che l'utente inizi un nuovo wizard mentre si trova dentro a un altro
* Migliorare la leggibilità del codice definendo metodi descriventi in modo chiaro l’evento di terminazione di un wizard
* Modularizzare ove possibile metodi grandi in più metodi piccoli:

*un esempio di questo è nella classe InsertFlowController, ove prima del refactoring era  
 presente un unico metodo mouseClicked, il quale doveva gestire ogni possibile azione utente  
 (rinuncia del wizard, aggiunta di un flow eccetera...).  
 Il metodo (grazie anche alla creazione di metodi ereditati dal padre) è stato diviso in più  
 metodi piccoli come "leftClickOperations",,"rightClickOperations","exitSpecificOperations".*

* Implementata in modo standard la possibilità che l'utente possa rinunciare in qualunque a completare il wizard corrente.  
  Prima del refactoring questo avveniva in diversi modi a seconda del wizard;   
   *per esempio in InsertInputNode l'utente, per rinunciare, doveva chiudere la finestra di scelta  
   nome, in InsertFlow doveva premere il mouse dx mentre in DeleteNode doveva premere con  
   mouse sx una porzione di spazio in cui non esisteva nessun nodo.*  
  Ora l’implementazione standard prevede il click destro del mouse.
* Successivamente all'introduzione dello stato utente, sempre in rapporto all’esclusività del wizard corrente, sono stati eliminati i metodi che impostavano tutti i bottoni attivi/disattivi (principalmente contenuti in ButtonHandlerController), in quanto la gestione sui bottoni non doveva essere più nella classe padre del ButtonHandler, ma nelle singole classi figlie che  
  controllano un pulsante specifico   
  *L'attivazione/disattivazione di pulsanti come InsertFlow e ExecuteMinimalAnalysis infatti segue eventi completamente diversi!*

Soppresse alcune implementazioni ridondanti di interfacce: *UserStateListener,ActionListener,ActionPerformerController erano implementate  
sia in AbstractButtonHandlerController che nella sua classe figlia, ButtonHandlerController*

## **Alg**

Per le classi dell’algoritmo è stata attuata manutenzione perfettiva.   
Lo stile di programmazione, giudicato a posteriori troppo procedurale, è stato modularizzato e pulito.  
A un certo punto è emersa un’errata interpretazione di un’ottimizzazione suggerita nella consegna, a proposito del prodotto cartesiano tra grafi disconnessi. La gestione delle connessioni è quindi degenerata in un Grafo unico, come descritto più accuratamente nel [package alg](#_alg).

## Gui

Il refactoring sulla grafica ha previsto:

* analisi e miglioramento del codice, in particolare della visibilità di classi e metodi
* adattamento ai nuovi [Controller](#_Controller), dati i loro cambiamenti
* aggiunta della classe GraphicComponent e delle sue classi figlie
  + GraphicNoStackComponent
  + GraphicStackComponent

che hanno permesso di eliminare del codice ridondante.

# PRECONDIZIONI, POSTCONDIZIONI E INVARIANTI

Specificate nella Javadoc del progetto e negli Use Case.