# Design Patterns e Dimostrazioni

Uso di design patterns per automatizzare la dimostrazione di una formula proposizionale usando regole inferenziali derivanti dalla deduzione naturale

# Logica Proposizionale

- **FBF** (**F**ormule **B**en **F**ormate)
- Valutazione
- Conseguenza Logica
- Deduzione Naturale
- Deducibilità
- Correttezza & Completezza
- Rappresentazione in Sequenti
- Deduzione Naturale in Sequenti
- Regole Invertibili

# FBF

$$Atom = \{ s \in \Sigma^* | \Sigma = \{ a, ..., z, A, ..., Z, 0, ..., 9 \} \}$$

$$Absurd = \#$$

$$FBF =_{min} \left\{ f | \begin{array}{l} f \in Atom \\ f \text{ is } Absurd \\ f = (! f_1) \text{ con } f_1 \in FBF \\ f = (f_1 \, op \, f_2) \text{ con } f_1, f_2 \in FBF \wedge op \in \{ >, \wedge, \vee \} \end{array} \right\}$$

# Valutazione

Una valutazione è una funzione:

$$v : FBF \rightarrow \{0,1\}$$

Ogni valutazione definisce un assegnamento sugli atomi che poi viene esteso alle formule in generale. Consideriamo che:

$$\forall\, a \in Atom.v(a) = a^{v} \in \{0,1\}$$

$$Prese\ \ a\ , b \in FBF :$$

$$v(\#) = 0 \qquad\qquad v((!\, a)) = 1 - v(a)$$

$$v((a \wedge b)) = min\{v(a), v(b)\}$$

$$v((a \vee b)) = max\{v(a), v(b)\}$$

$$v((a > b)) = max\{1 - v(a), v(b)\}$$

# Conseguenza Logica

Si dice che una formula *f* è conseguenza di una formula *F* (*F* |= *f*) se:

$$\forall\, v : FBF \rightarrow \{0,1\} . v(F) = 1 \;\; allora \;\; v(f) = 1$$

Si dice poi che una formula *f* è conseguenza logica si un insieme di formule *KB* se:

$$\forall\, v : FBF \rightarrow \{0,1\} . (\forall\, F \in KB \subseteq FBF . v(F) = 1) \;\; allora \;\; v(f) = 1$$

Le uniche formule conseguenza logica dell'insieme vuoto sono tautologie.
Se *KB* ha come conseguenza logica l'assurdo allora *KB* è insoddisfacibile.

# Deduzione Naturale

Insieme di regole inferenziali che permettono di dimostrare un teorema da un certo numero di assunzioni.

Esistono due tipi di regole:

- Regole di eliminazione: tutte quelle regole che permettono di fare un passo nella deduzione semplificando il teorema da dimostrare. Solitamente eliminano un connettivo.
- Regole di introduzione: Permettono di introdurre una formula per proseguire nella dimostrazione

# Deduzione Naturale

## Regole

### REGOLE DI INTRODUZIONE

$$\frac{A \qquad B}{A \,\&\, B}$$

$$\frac{A}{A \mid B} \qquad \frac{B}{A \mid B}$$

$$\frac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A > B}$$

$$\frac{\begin{array}{c}[A] \\ \vdots \\ \#\end{array}}{!\,A}$$

### REGOLE DI ELIMINAZIONE

$$\frac{A \,\&\, B}{A} \qquad \frac{A \,\&\, B}{B}$$

$$\frac{A|B \qquad \begin{array}{c}[A] \\ \vdots \\ C\end{array} \qquad \begin{array}{c}[B] \\ \vdots \\ C\end{array}}{C}$$

$$\frac{A \qquad A > B}{B}$$

$$\frac{A \qquad !A}{\#}$$

$$\frac{\#}{A}$$

### REDUCTIO AD ABSURDUM

$$\frac{\begin{array}{c}[!\,A] \\ \vdots \\ \#\end{array}}{A}$$

# Scaricare le formule

Le regole di introduzione di implicazione e negazione, la RAA e l'eliminazione della disgiunzione permettono di scaricare alcune formule.

Se una formula è scaricabile potrà essere usata nella deduzione. Nel momento in cui dovrà essere dimostrata ci limiteremo a scaricarla.

In quest'ottica possiamo leggere la regola di introduzione dell'implicazione come:

*“Volendo dimostrare A > B non devo far altro che dimostrare B assumendo vera A”*

In pratica una formula scaricabile viene considerata già dimostrata

# Deducibilità

Una formula *f* si dice deducibile da un insieme di formule *KB* (*KB* |- *f*) sse esiste un albero in deduzione naturale tale per cui le foglie non scaricate stanno tutte in *KB*.

```
(!A | C),(!B | C)|-((A | B) > C)

            [A]      [! A]                     [B]      [! B]
            --------------                     --------------
                  #                                  #
                -----                              -----
      (!A | C)    C      [C]          (!B | C)       C       [C]
      ------------------------        ---------------------------
[A | B]           C                              C
---------------------------------------------------
                  C
             -----------
             (A | B) > C
```

# Correttezza & Completezza

La deduzione naturale risulta essere un sistema deduttivo corretto e completo rispetto alla conseguenza logica.
Quindi una formula *f* è conseguenza logica di un insieme di formule *KB* sse da *KB* posso dedurre *f* usando la deduzione naturale.

$$KB \models f \quad \text{sse} \quad KB \vdash f$$

# Rappresentazione in Sequenti

La deduzione naturale è uno strumento molto elegante e simula in un certo modo il ragionamento matematico. Tuttavia non è facile da utilizzare e richiede una parte di ingegno per trovare “la giusta strada”. Per semplificarci la vita la cosa migliore da fare è utilizzare ad ogni passo dell'albero l'intera proposizione (KB |- f) e ogni volta che viene travata una regola che permetta di scaricare una formula si aggiunge la formula alle premesse. Otteniamo così una rappresentazione in sequenti della deduzione naturale. In questo modo risulta essere molto più semplice scegliere la regola da applicare.

# Deduzione naturale in sequenti

Le regole della deduzione naturale rappresentate in sequenti sono:

REGOLE DI INTRODUZIONE

$$\frac{\text{KB|-A} \qquad \text{KB|-B}}{\text{KB|-A \& B}}$$

$$\frac{\text{KB|-A}}{\text{KB|-A | B}} \qquad \frac{\text{KB|-B}}{\text{KB|-A | B}}$$

$$\frac{\text{KB,A|-B}}{\text{KB|-A > B}}$$

$$\frac{\text{KB,A|-\#}}{\text{KB|-! A}}$$

REDUCTIO AD ABSURDUM

$$\frac{\text{KB,!A|-\#}}{\text{KB|-A}}$$

REGOLE DI ELIMINAZIONE

$$\frac{\text{KB|-A \& B}}{\text{KB|-A}} \qquad \frac{\text{KB|-A \& B}}{\text{KB|-B}}$$

$$\frac{\text{KB|-A|B} \qquad \text{KB,A|-C} \qquad \text{KB,B|-C}}{\text{KB|-C}}$$

$$\frac{\text{KB|-A} \qquad \text{KB|-A > B}}{\text{KB|-B}}$$

$$\frac{\text{KB|-A} \qquad \text{KB|-!A}}{\text{KB|-\#}}$$

$$\frac{\text{KB|-\#}}{\text{KB|-A}}$$

# Invertibilità

Una regola di un sistema deduttivo di dice invertibile se assumendone le conclusioni si può dimostrare le premesse della regola stessa; nel sistema deduttivo privato della regola in questione. Un sistema si dice invertibile se lo sono tutte le sue regole.

La deduzione naturale e la rappresentazioni in sequenti non sono sistemi invertibili.

L'invertibilità garantisce che una proposizione sia soddisfacibile sse lo è/sono la/le proposizione/i ottenute applicando una regola

# Regole invertibili

ho aggiunto un assioma indispensabile e spiegato cosa sono le cose usate

Le seguenti regole sono invertibili e possono essere ricondotte ad un albero di deduzione naturale rappresentato in sequenti.

**REGOLE DI INTRODUZIONE**

$$\frac{KB|\text{-}A \qquad KB|\text{-}B}{KB|\text{-}A \;\&\; B}$$

$$\frac{KB,!\,A|\text{-}B}{KB|\text{-}A \mid B}$$

$$\frac{KB,A|\text{-}B}{KB|\text{-}A > B}$$

$$\frac{KB,!D|\text{-}a}{KB,(!\,a)|\text{-}D}$$

$$KB,\#|\text{-}A$$

$$KB,A|\text{-}A$$

**REGOLE DI ELIMINAZIONE**

$$\frac{KB,A,B|\text{-}C}{KB,A \;\&\; B|\text{-}C}$$

$$\frac{KB,A|\text{-}C \qquad KB,B|\text{-}C}{KB,A \mid B|\text{-}C}$$

$$\frac{KB,!A|\text{-}C \qquad KB,B|\text{-}C}{KB,A > B|\text{-}C}$$

$$\frac{KB,A|\text{-}\#}{KB|\text{-}!\,A}$$

Con A,B,C FBF, a Atomo, e D FBF non Atomica

# Osservazioni

Con le regole presentate riusciamo ad avere un sistema di deduzione corretto e completo che però risulta molto facile da automatizzare. Lo svantaggio principale è la perdita di "naturalezza". Infatti si perdono alcune regole il cui uso è tipico nelle dimostrazioni; prima tra tutte il *modus ponens*.

Tuttavia con queste regole la dimostrazione può essere automatizzata.

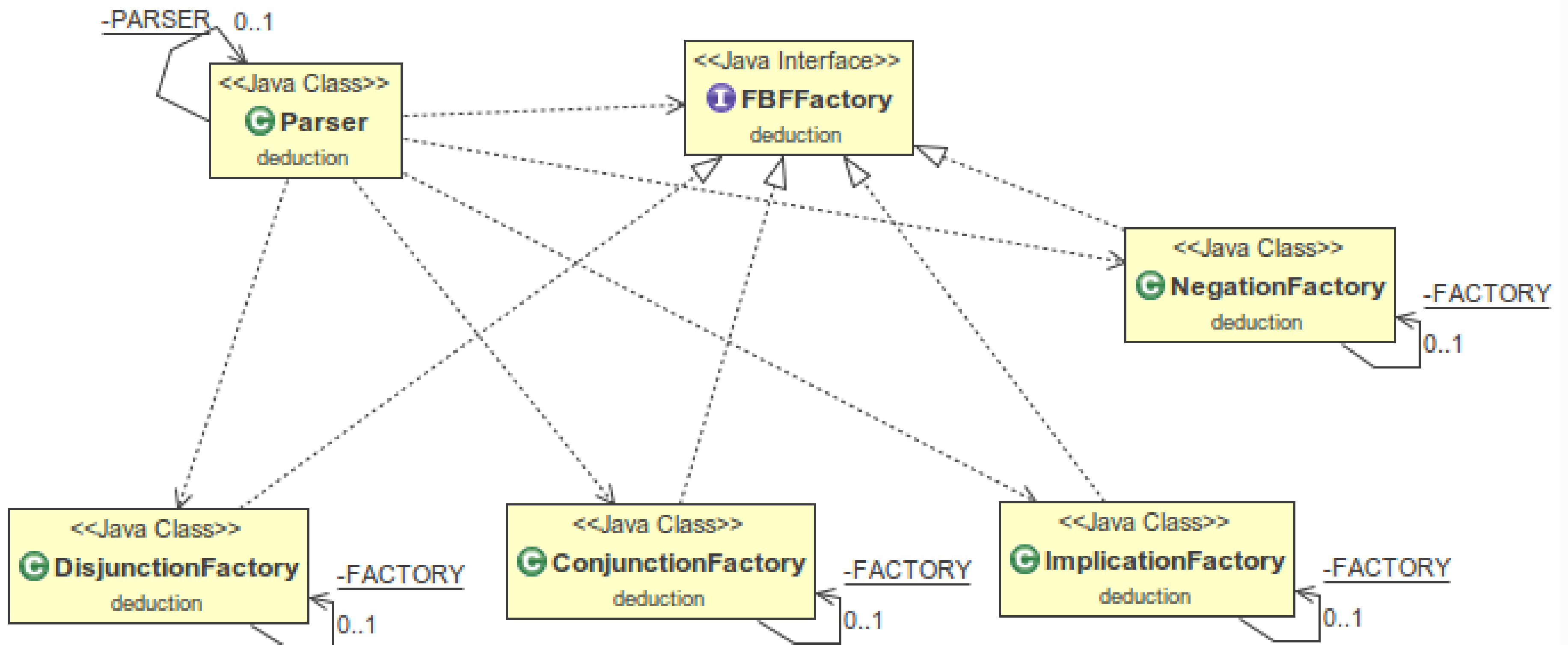
# Automatizzazione

**Invertibilità**
Garantisce che applicando una regola le proposizioni ottenute sono valide sse lo è la originale;
**Diminuzione dei connettivi**
Ad ogni applicazione delle regole i connettivi (utilizzabili) diminuiscono quindi prima o poi non ci saranno più regole applicabili
**Raggiungimento di un caso base**
Applicando le regole raggiungeremo un caso in cui sulla sinistra abbiamo solo letterali o l'assurdo e sulla destra un atomo o l'assurdo. Questo può essere considerato un caso baso. Per la decidibilità di questo tipo di proposizione si applicano i due assiomi.

# Classi e Pattern

- Parsing
- FBF
- Proposizioni
- Regole
- Proposizioni e Regole
- Deduzione
- Valutazione

# Parsing



# Parser

La classe Parser ha una duplice funzionalità, crea le FBF e le proposizioni iniziali. Il secondo compito è di natura marginale in quanto non fa altro che dividere una stringa in premesse e conclusioni, dividere le premesse e mandarle tutte al parser che le trasforma in FBF.

# Parser

Il parsing delle formule è eseguito con una tecnica simile ad un parser LR. Vengono usate due stack una contiene le formule già parsate l'altra le factory con le quali creare nuove formule. Il procedimento è semplice:

- Se si trova un atomo seguito da un connettivo binario allora si mette l'atomo nella stack delle formule parsate e la factory relativa al connettivo nello stack delle factory;
- se si trova una negazione si inserisce tra le formule parsate un null e la factory della negazione nello stack delle factory;
- quando si trova un atomo seguito da una parentesi chiusa si crea una nuova formula con tutti i connettivi che hanno un livello di annidamento maggiore o uguale al livello di annidamento dell'atomo in questione.

# Parser

(a & b)

[],[]->[a],[]->[a],[&]->[(a & b)],[]

(! a)

[],[]->[null],[!]->[(! a)],[]

((a & b) | (! (b > c)))

[],[]->[a],[]->[a],[&]->[(a & b)],[]->

[(a & b)],[|]->[null,(a & b)],[!,|]->

[b,null,(a & b)],[!,|]->

[b,null,(a & b)],[>,!,|]->

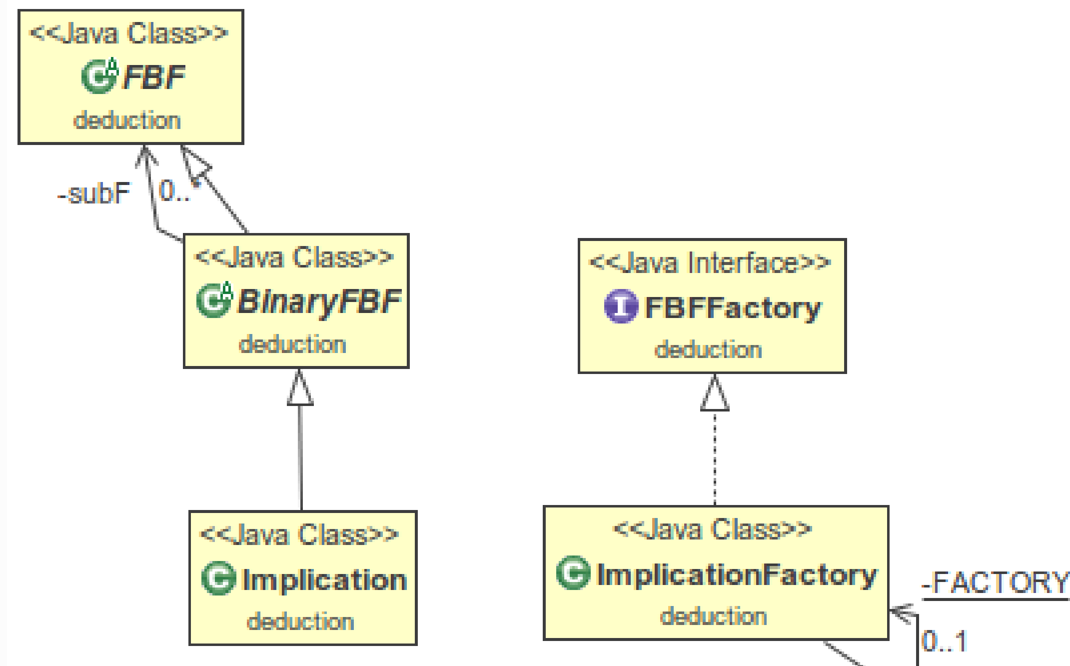
[(b > c),null,(a & b)],[!,|]->

[(! (b > c)),(a & b)],[|]->

[((a & b) | (! (b > c)))],[]

# Parser

Come detto precedentemente il parser usa delle factory per rimandare la creazione di una formula. La possibilità di fare ciò viene garantita dall'uso di Factory Method. In questo modo quando trovo la seconda parte di una formula posso richiamare il factory senza conoscerne il tipo specifico. I Factory Method seguono lo schema:

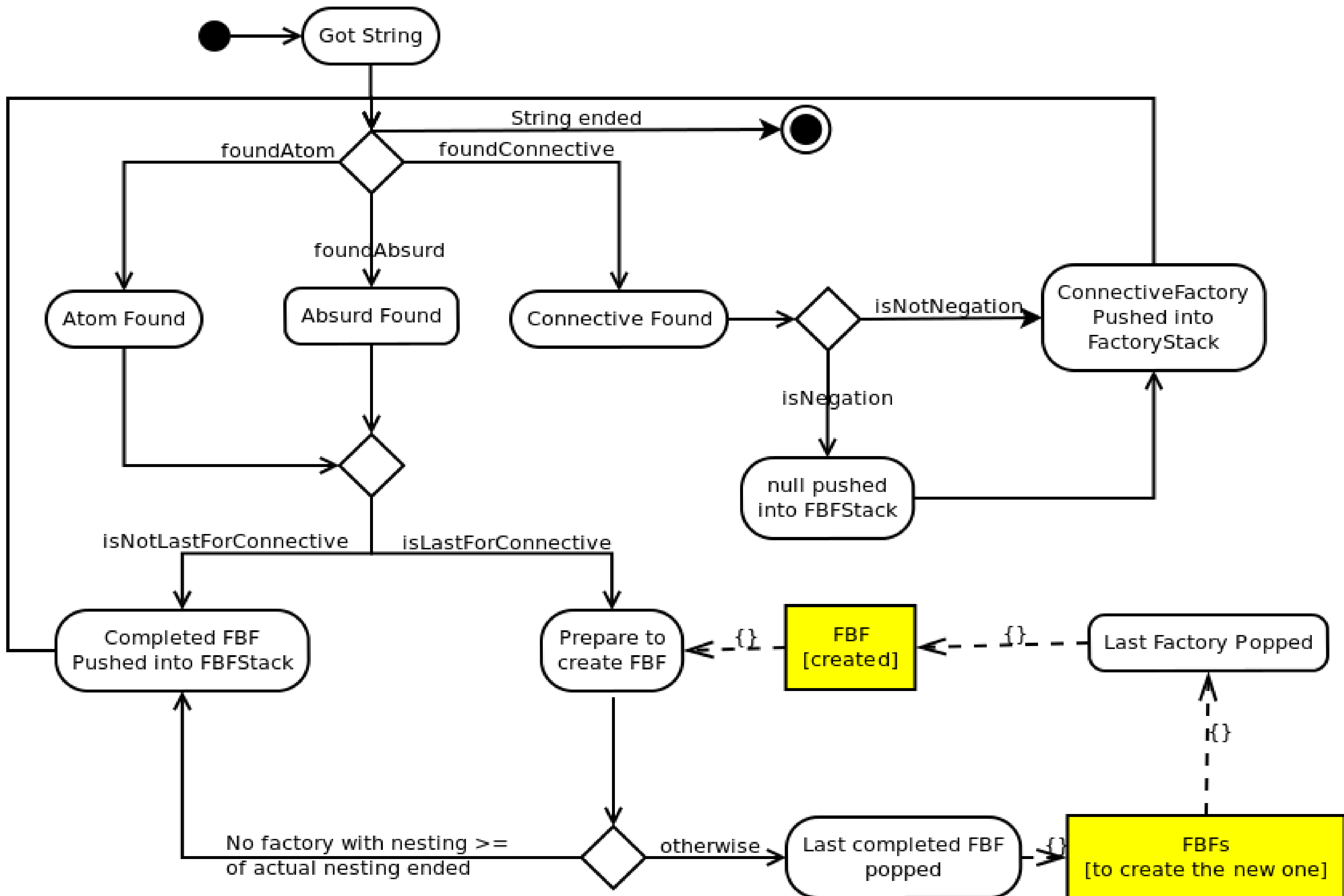


# Parser

Per le formule atomiche e l'assurdo non sono state create factory perché non c'é molto da fare per la loro creazione. Tuttavia è stato isolato il metodo che parsa e crea atomi e assurdi per poterlo lasciare eventualmente ad una personalizzazione successiva in stile template.

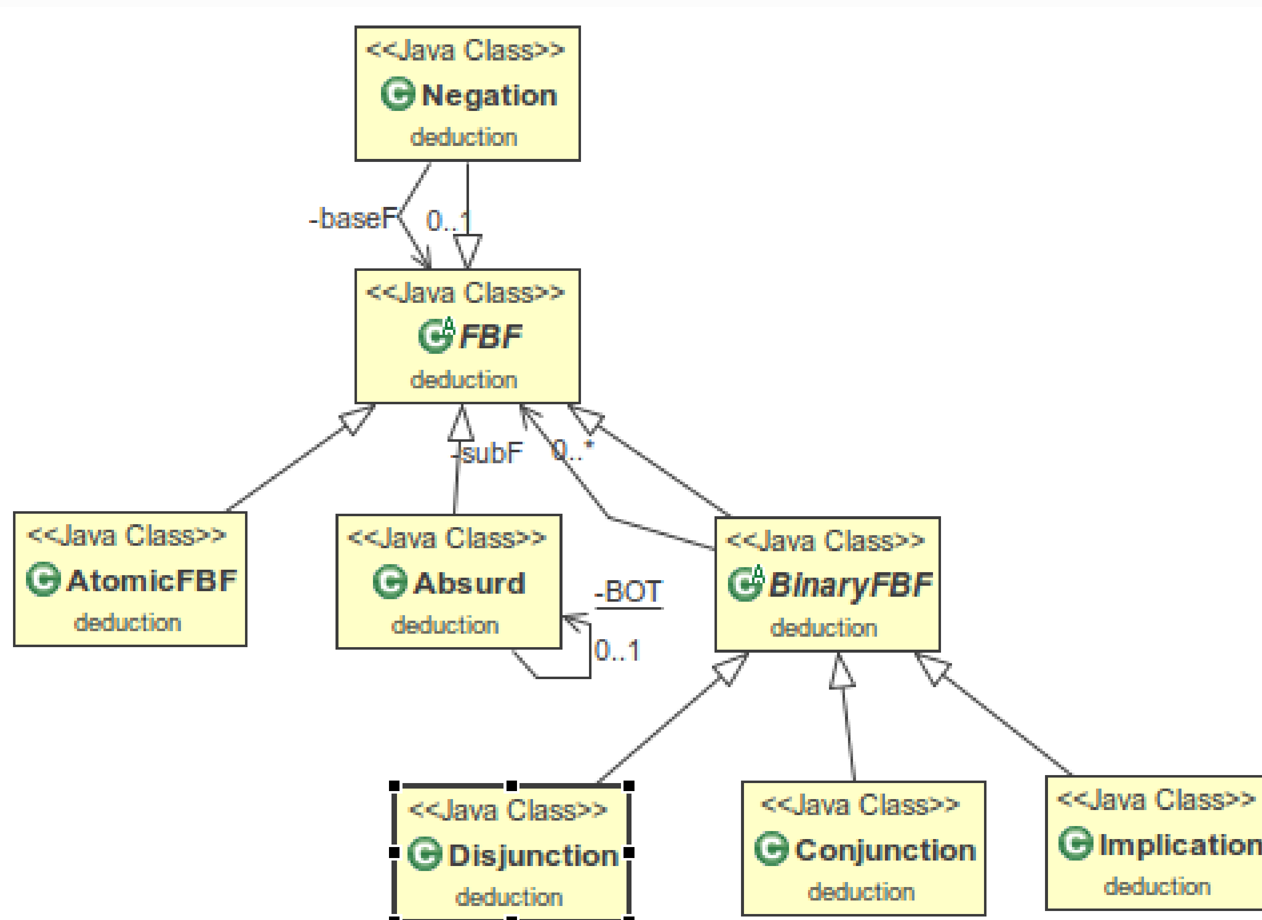
Tutte le factory sono singleton in quanto non si ha necessità che ne vengano create più di una.

# Parser
## Diagramma di attività



# FBF

Le formule hanno una naturale struttura ad albero perciò per la loro implementazione è stato usato Composite che ne rappresenta perfettamente la struttura.



# FBF

Come si può notare Atom e Absurd sono le foglie mentre Negation, Conjunction, Disjunction e Implication sono nodi. Negation è distinta dalle altre perché avrà solo un figlio.

Absurd è un singleton in quanto avremo un solo assurdo. Non c'è motivo per averne altri.

# Proposizioni

Una proposizione sarà rappresentata come

**a,(a | b),(b > c)|-(b | c)**

**Compiti**

- Memorizzare le formule (premesse e conseguenza);
- Realizzare in base alle formule un passo di deduzione.

**Cosa fanno:**

Mantengono tutte le formule e ne scelgono una per un passo di deduzione

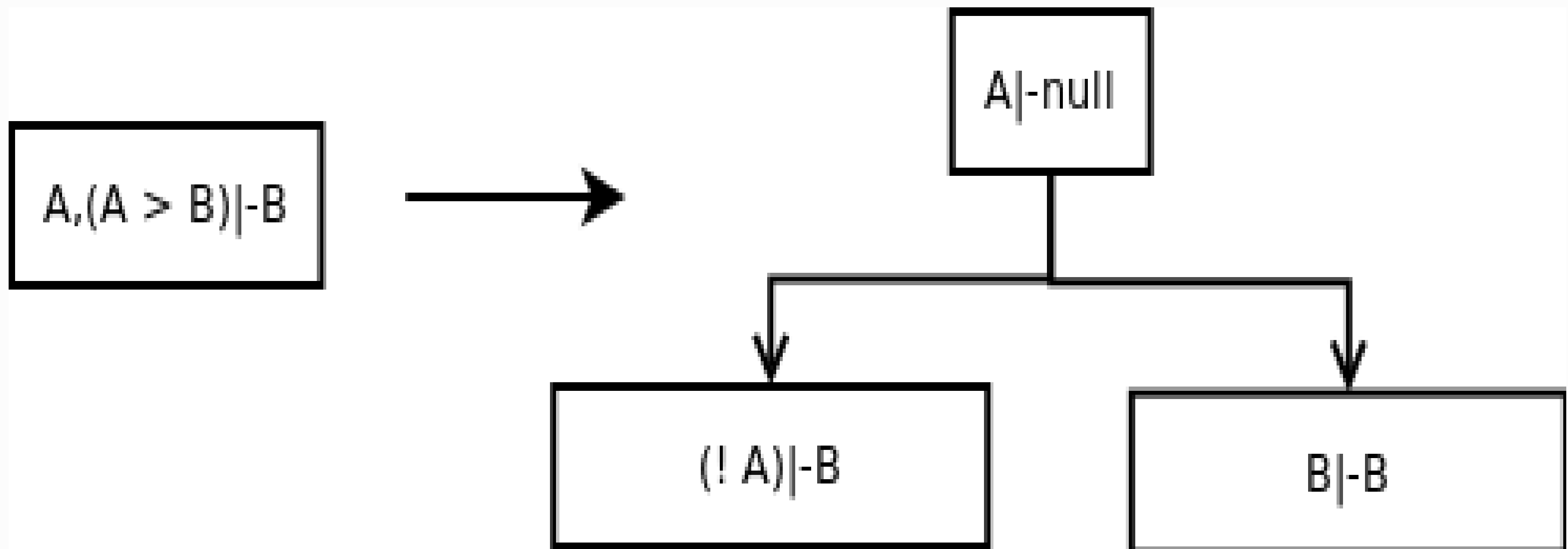
**Come**

- Se la regola applicata non produce branching la proposizione si modifica secondo la regola;
- Se la regola produce branching la proposizione viene decorata con altre proposizioni per non duplicare immediatamente tutte le formule.

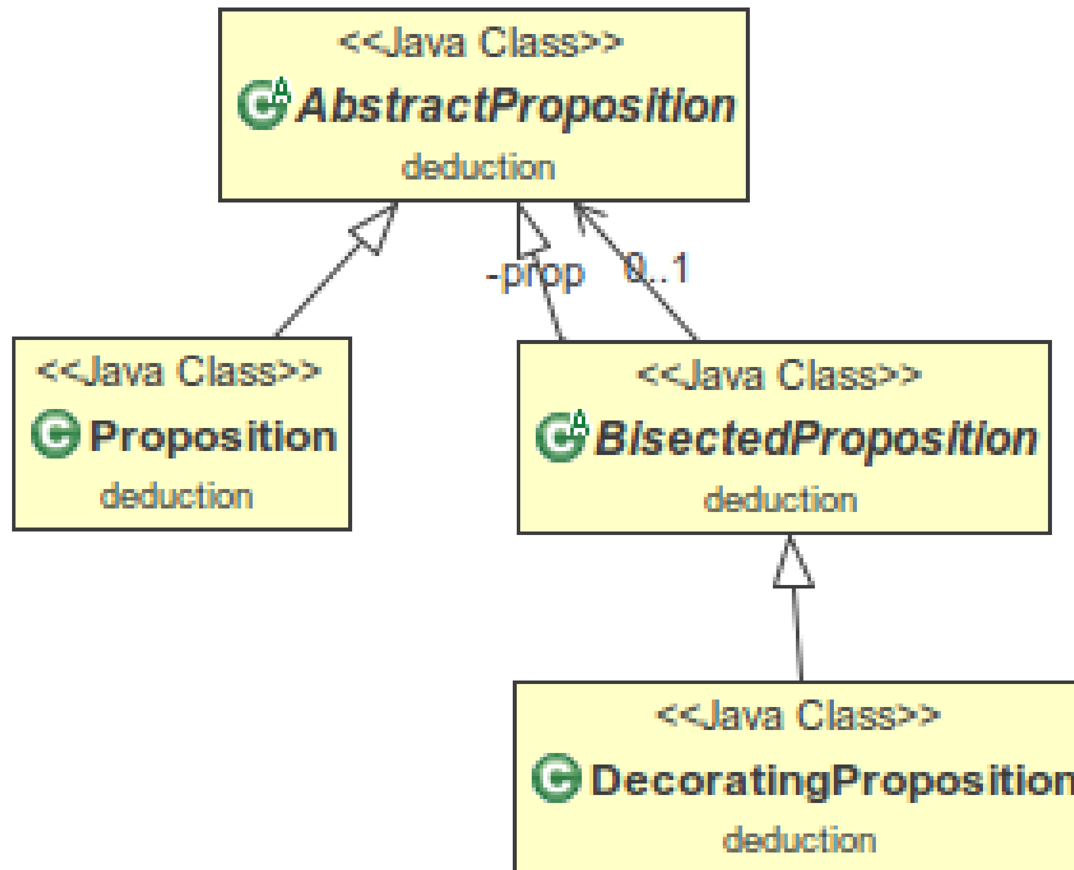
# Proposizioni

**Decorator**

Ogni proposizione originata da un branching decora la formula che l'ha originata invece di crearcne una completamente nuova. Ogni volta che una proposizione viene decorata la sua conclusione viene posta a null perché non dovrà più essere considerata.



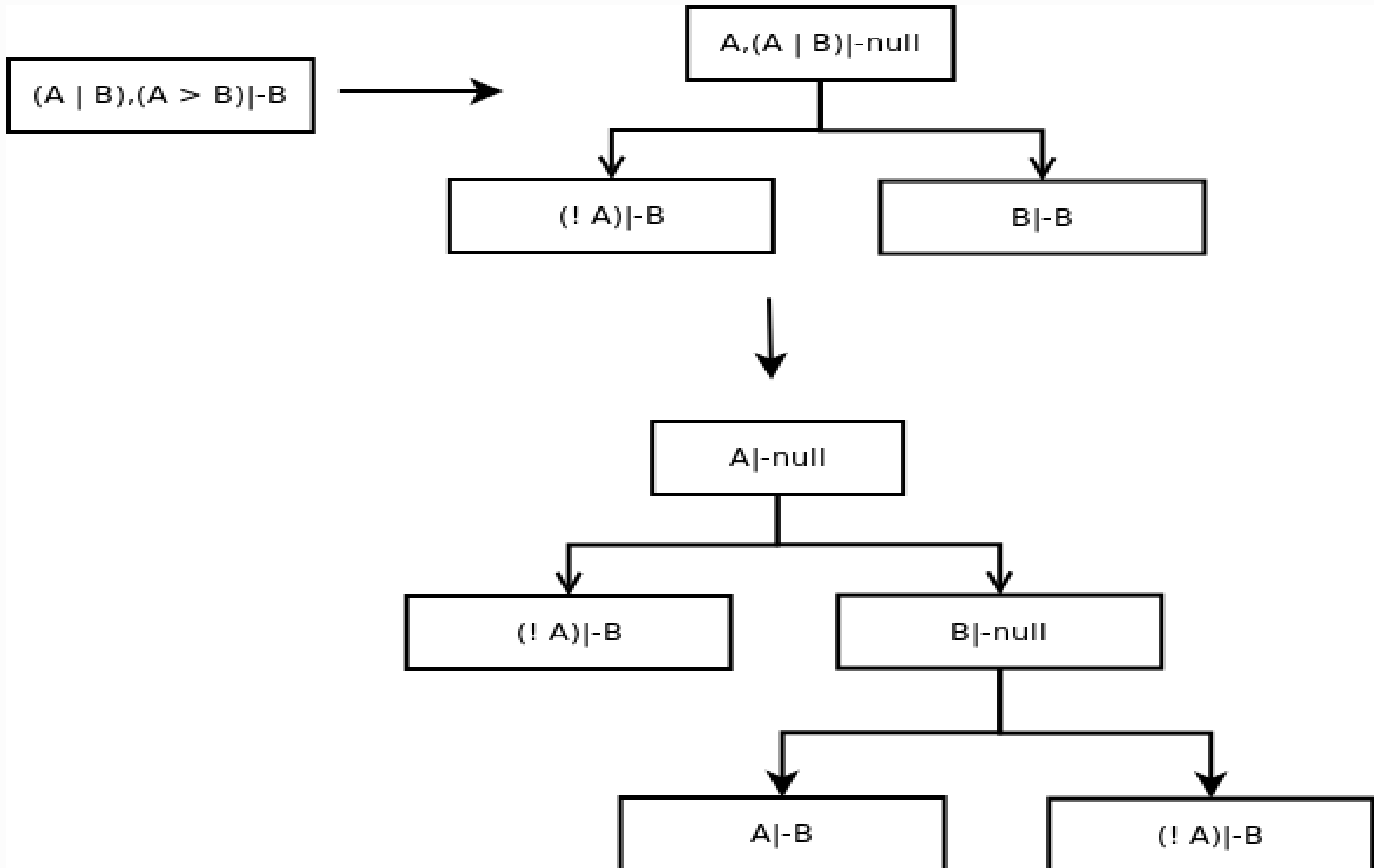
# Proposizioni



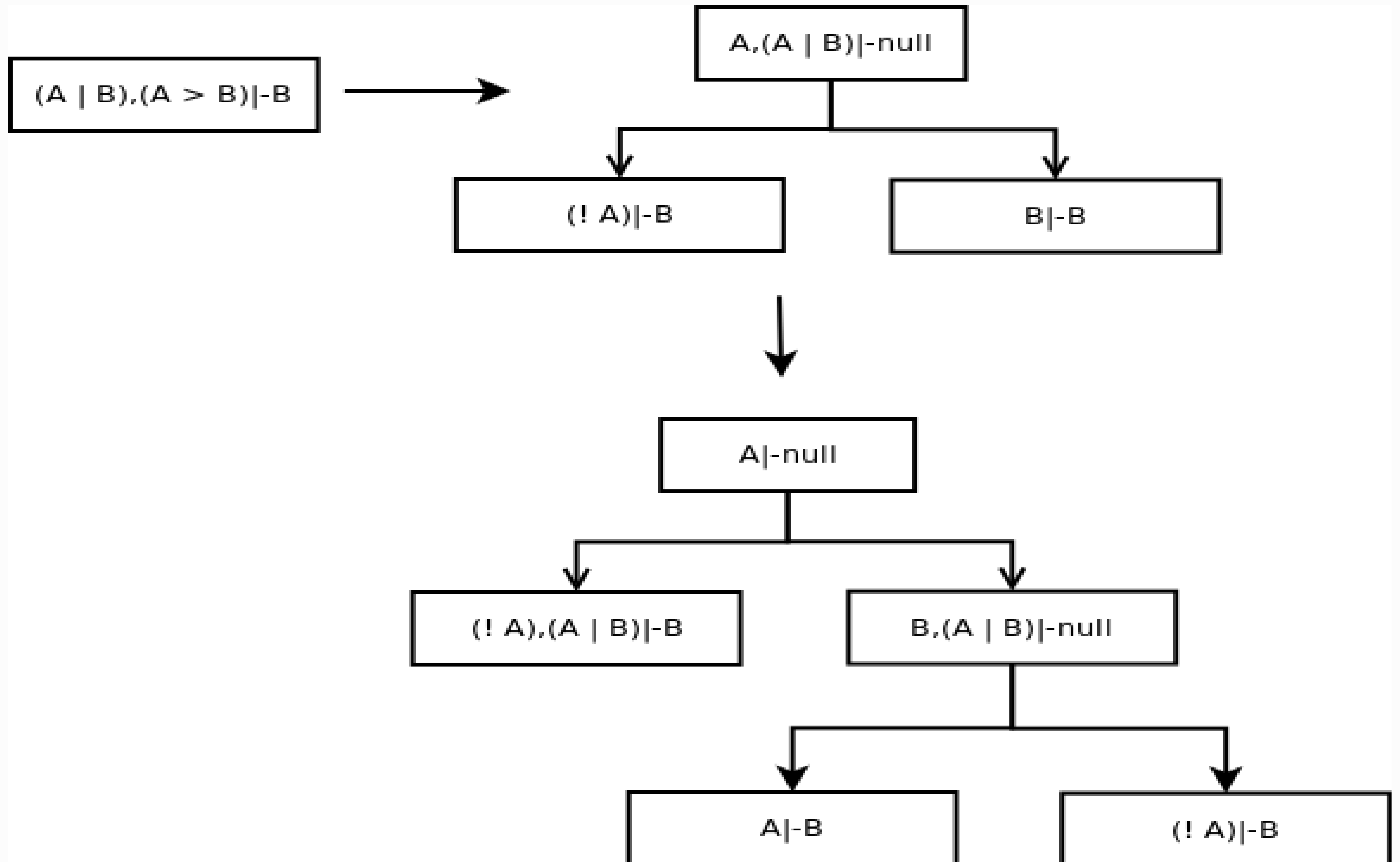
# Proposizioni

Proposition rappresenta la proposizione iniziale su cui si vuole ragionare ogni volta che si ha branching si produce delle DecoratingProposition che mantengono le formule non processate.
Questo però crea un problema. Infatti nel momento in cui si va ad applicare una formula che viene mantenuta nello stato condiviso questa deve essere inviata a quei rami che non l'hanno ancora usata.
Quindi si rende necessario che la proposizione decorata mandi alle decoranti la formula in questione.
Per poter realizzare questo è stato utilizzato un Observer.

# Proposizioni
senza observer

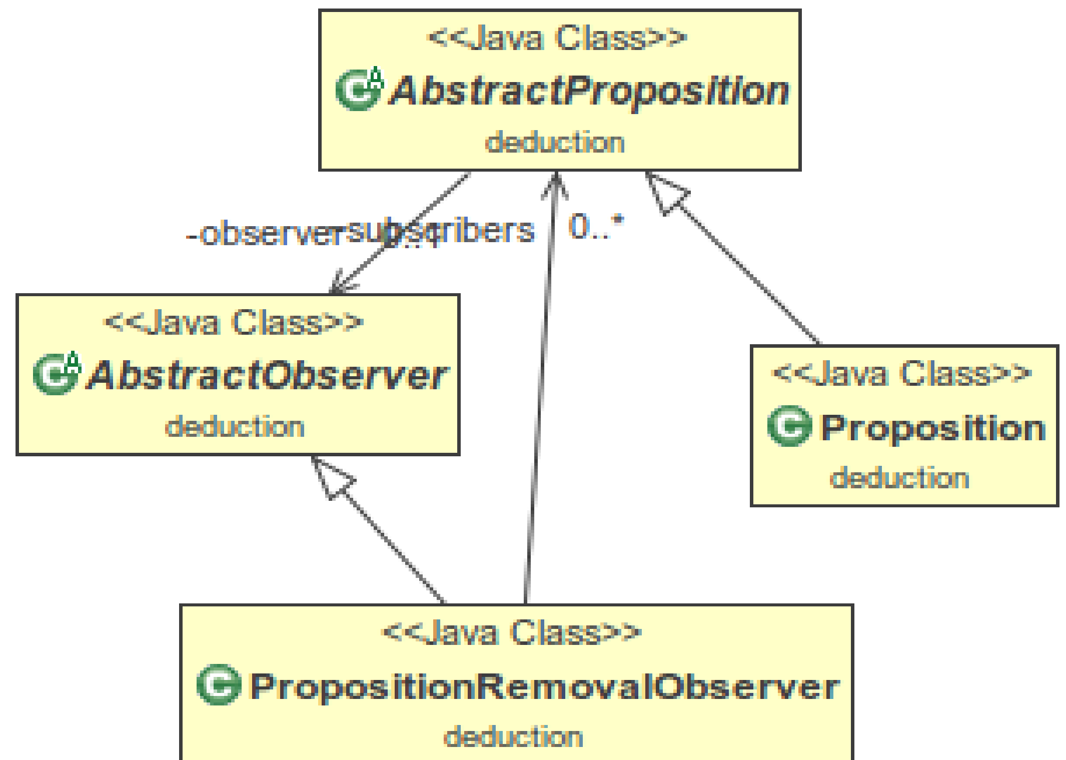


# Proposizioni
con observer



# Proposizioni

Ogni volta che una AbstractProposition viene decorata si assegna a questa un Observer. Nel momento in cui una formula viene rimossa dalla decorata questa lo comunica all'observer che la distribuisce alle decoranti. Le decoranti vengono passate all'observer al momento della creazione di questo. L'assegnamento, e quindi la creazione, di un observer avviene ogni volta che l'applicazione di una regola causa branching (decorazione).



# Regole

Le regole rappresentano gli strumenti coi quali eseguire deduzioni. Hanno un duplica scopo:

- Eseguire un passo di deduzione;
- Garantire che l'applicazione sia consentita.

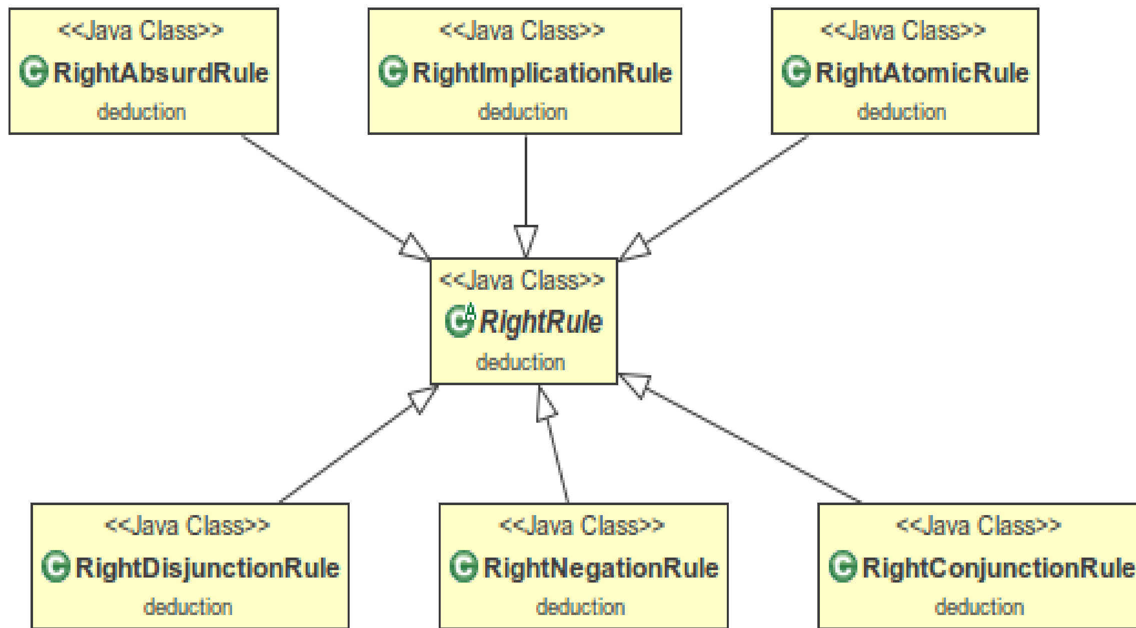
Per fare ciò ogni regola deve considerare sia la formula sulla quale la regola viene applicata sia la proposizione che contiene la formula presa in considerazione.

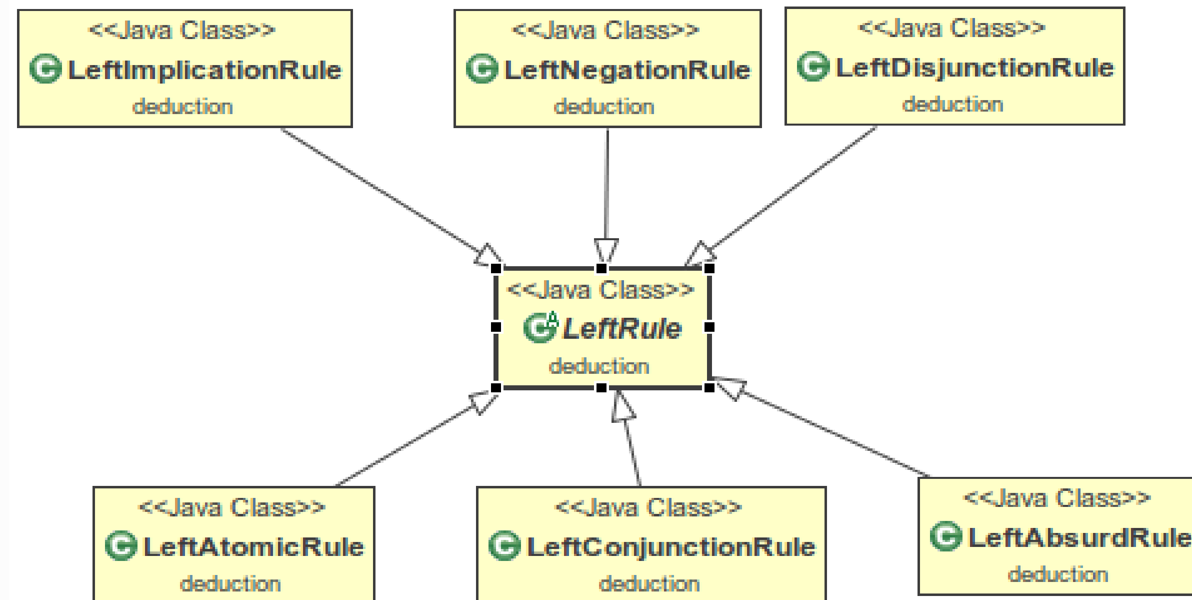
# Regole

Le regole sono distinte in LeftRule e RightRule. Le prime rappresentano le regole che si applicano a premesse di una proposizione, le seconde si applicano alla conclusione. Le due classi sono nettamente distinte tra loro perché rappresentano oggetti molto differenti e non devono in nessun caso essere confuse.

LeftRule e RightRule sono classi astratte che definiscono la struttura generica delle regole. Le relative implementazioni rappresentano le regole stesse.

# Regole



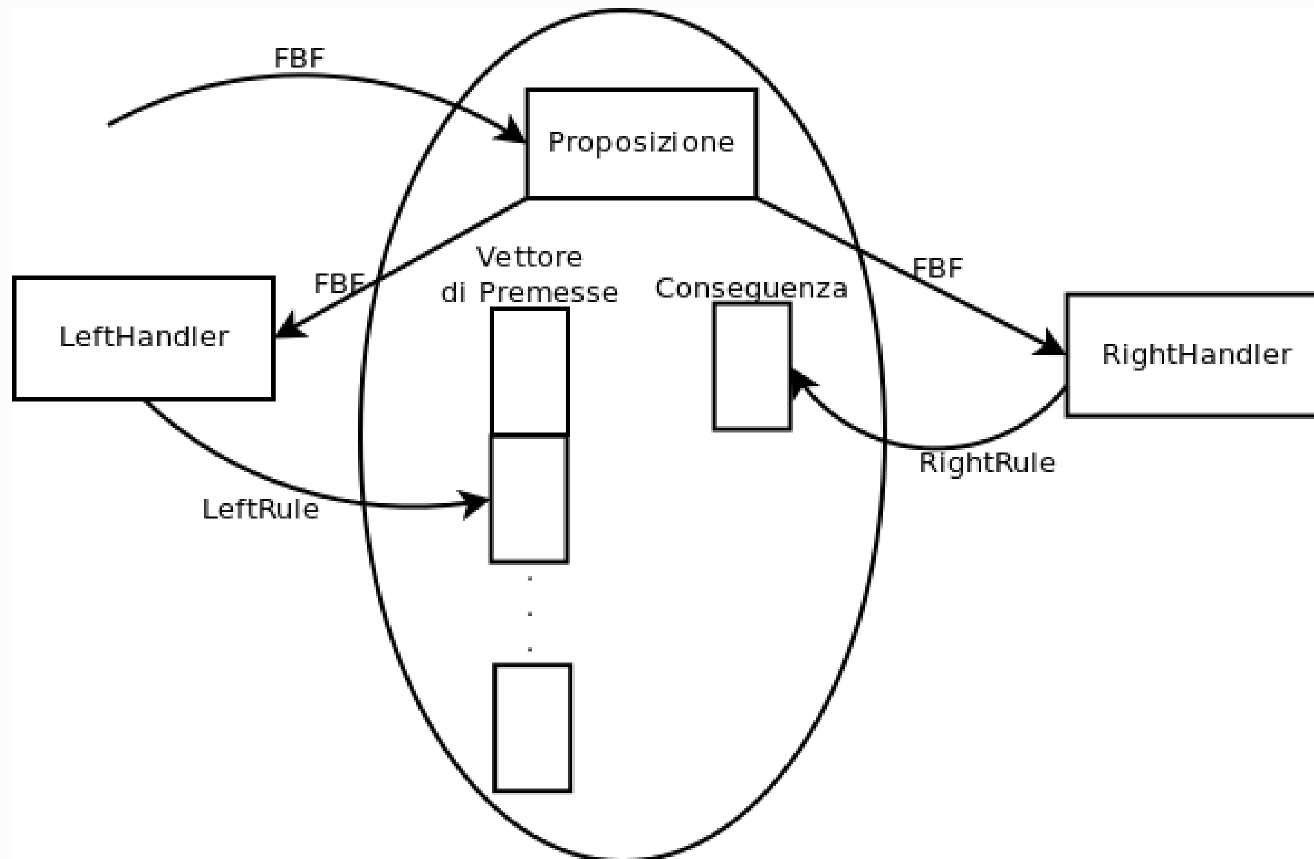
# Proposizioni e Regole

A questo punto abbiamo tutti gli ingredienti per procedere con la deduzione. Come visto finora le proposizioni dovrebbero avere un certo numero di formule sulla sinistra e una formula sulla destra. Questa modalità ha diversi inconvenienti:

- ✗ Le proposizioni devono conoscere profondamente la struttura sia delle formule che delle regole;
- ✗ Ogni volta che si procede nella dimostrazione occorre controllare tutte le possibili regole applicabili e scegliere la più "opportuna";
- ✗ Aggiungere una nuova regole comporta una modifica radicale delle operazioni delle proposizioni.

# Proposizioni e Regole

Per ovviare ai problemi presentati, si mette nelle proposizioni regole specificate e al momento necessario si richiama il metodo deduce della formula.



# Proposizioni e Regole

Quanto visto finora permette di risolvere il primo problema. Infatti in questo modo riusciamo ogni volta a prendere una regola, chiamare deduce su questa senza preoccuparsi di cosa veramente facciamo.
Il secondo problema resta sospeso. Infatti finora non sappiamo come scegliere in modo opportuno una regola. Per prima cosa definiamo opportuno. Per opportuno si intende applicare prima le regole applicabili che non causano branching. Poi tra le branching applicare quelle che ne fanno meno. Nel nostro caso abbiamo solo branching di grado 2, però lasciamo la possibilità di introdurre regole con grado di branching diverso. Inoltre l'unica regola che potrebbe non essere applicabile è la regola sinistra per la negazione che richiede che a destra ci sia un atomo o l'assurdo.

(A | B),(A & B)|-B

A,(A & B)|-null

(! A)|-B

B|-B

A|-null

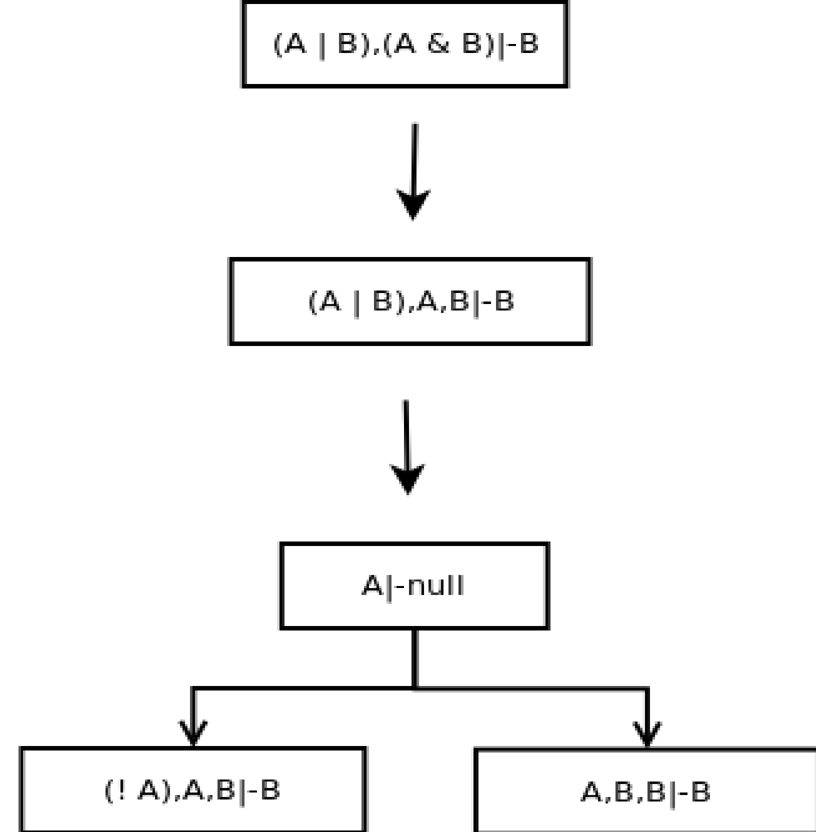
(! A),(A & B)|-B

A,B,B|-B

A|-null

(! A),A,B|-B

A,B,B|-B




Come si può vedere cambiare l'ordine di applicazione delle regole può modificare il numero di applicazioni necessarie per completare la dimostrazione.

# Proposizioni e Regole

Per avere la possibilità di seguire questa politica si è scelto di usare nelle proposizioni non un vettore di regole ma un vettore di vettori. Il vettore conterrà in posizione *i* le regole con branching degree *i*. Gli atomi e l'assurdo (sia a destra che a sinistra) avranno branching degree 0. I letterali negativi a sinistra avranno anche loro branching degree 0.

0
[a,b,(!b),#]

1
[(a & b),(!(b > c))]

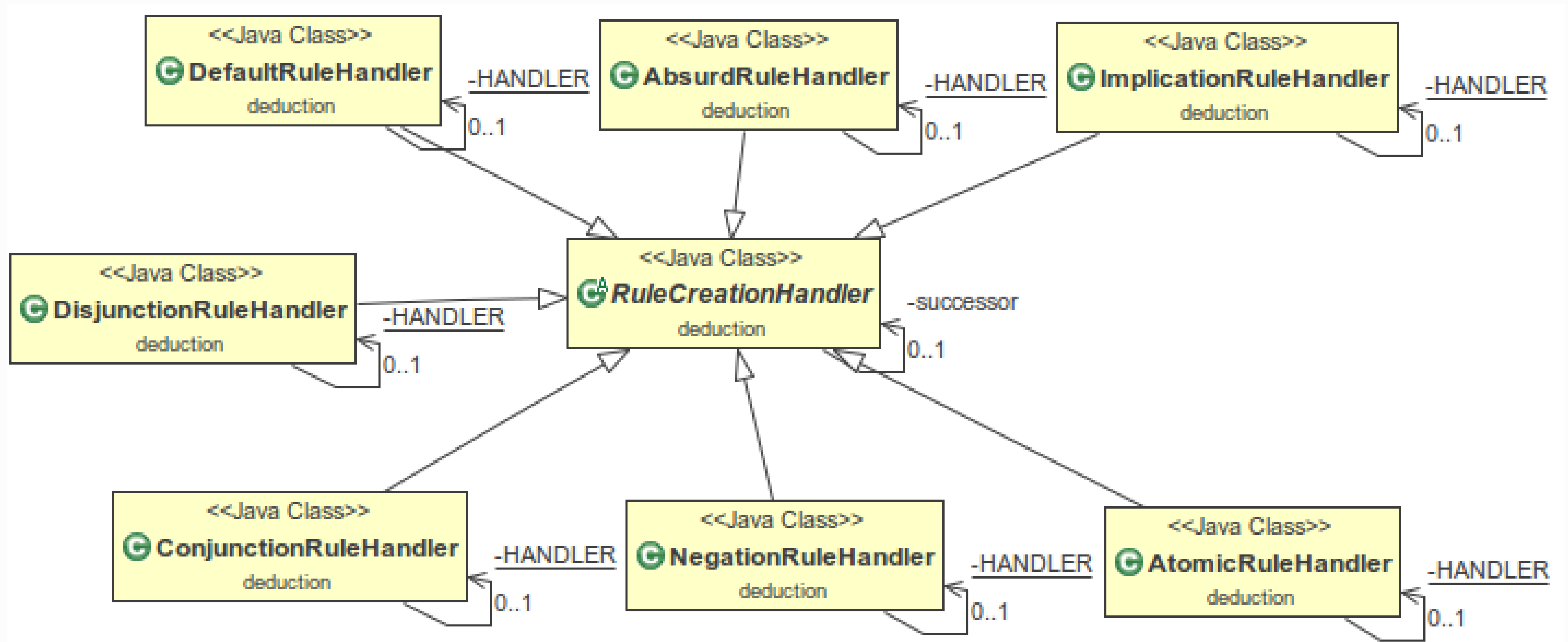
2
[(a | b),((!b) > c),((a & c) | b)]

3
[(a | b | c)]

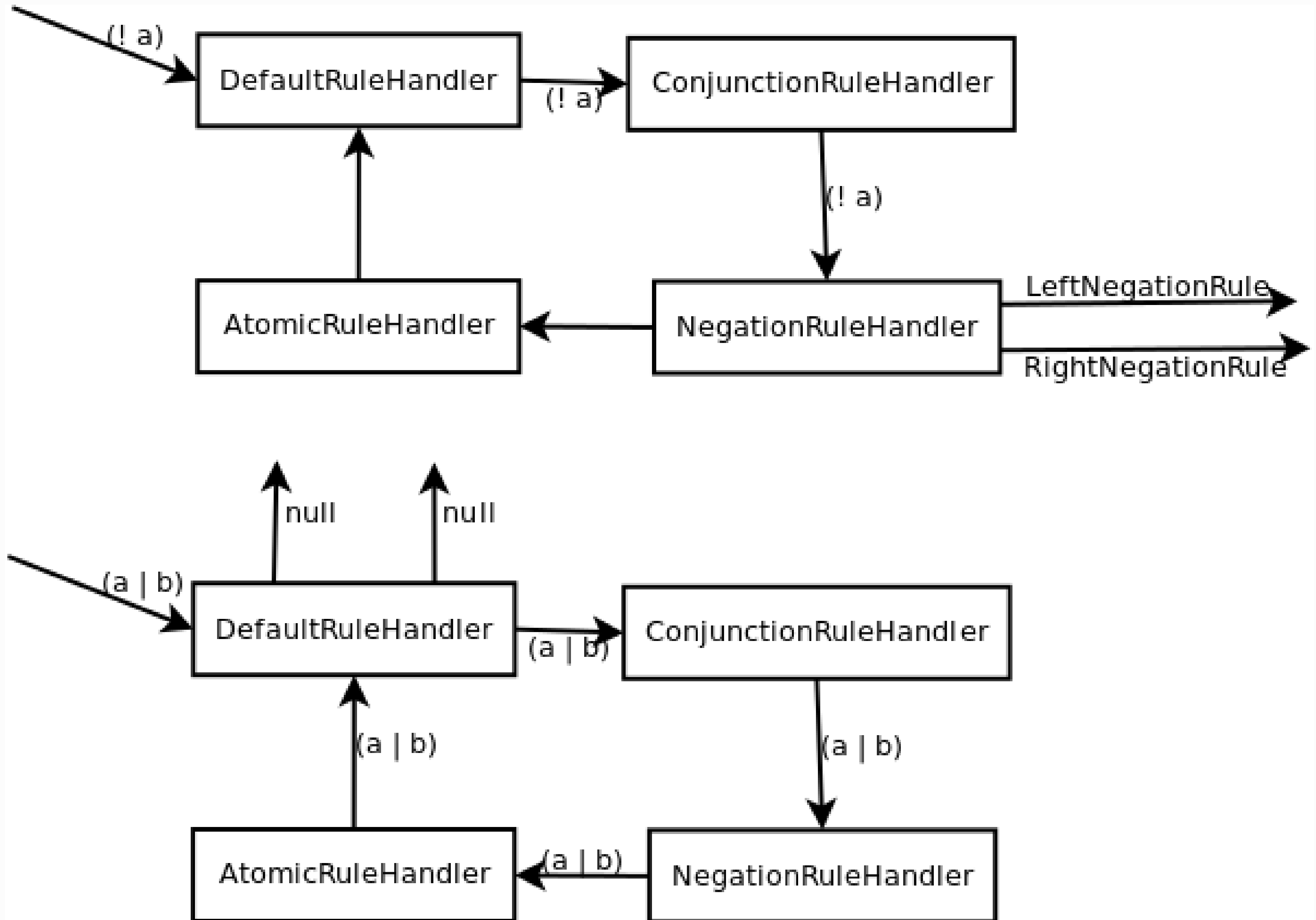
# Proposizioni e Regole

Infine per permettere alle proposizioni di astrarre completamente dalle regole è stato usato un Chain of Responsability di AbstractFactory. Le abstract factory sono singleton e sono sottoclassi di CreationRuleHandler. Viene specificata una classe DefaultRuleHandler che non riconosce nessuna FBF. Questa è sempre considerata la testa e la coda della catena (circolare). Il suo compito è quello di dirigere la giusta richiesta (left/right creation) e di segnalare che la formula non è riconosciuta.

# Creazione delle regole



# Creazione delle regole



# Proposizioni Regole

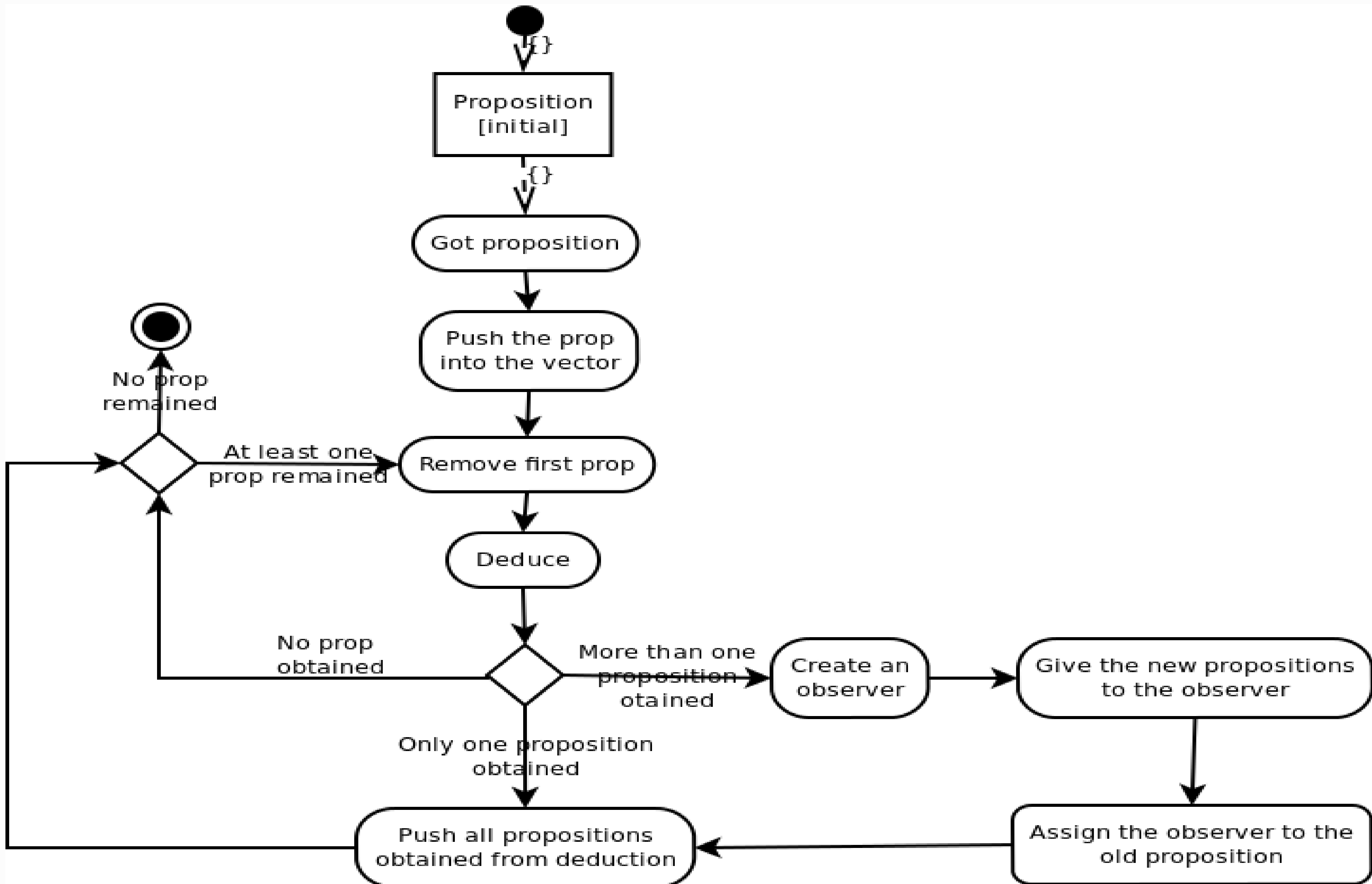
In questo modo siamo riusciti ad avere un vettore di regole-premessa ed una regola-conclusione. La proposizione non dovrà fare altro che pescare dalle premesse la regola con branching degree minore e richiamare su questa deduce per produrre proposizioni secondo le regole di deduzione. A questo punto siamo in grado di eseguire un passo di deduzione a partire da una proposizione. Vediamo infine come eseguire e completare la deduzione.

# Deduzione

Tutta la procedure di deduzione viene affidata alla classe Deduction. Vediamo come opera:

- Prende una proposizione, la inserisce in un vettore e inizia la deduzione;
- Ad ogni passo rimuove dalla testa del vettore una proposizione;
- Applica una regola per procedere (di un passo) nella deduzione;
- Ottiene la/le nuova/e proposizione/i;
- Assegna alla vecchia proposizione un observer al quale sono iscritte le proposizioni prodotte (solo con branching > 1);
- Reinserisce nel vettore tutte le proposizioni che possono essere non valide.

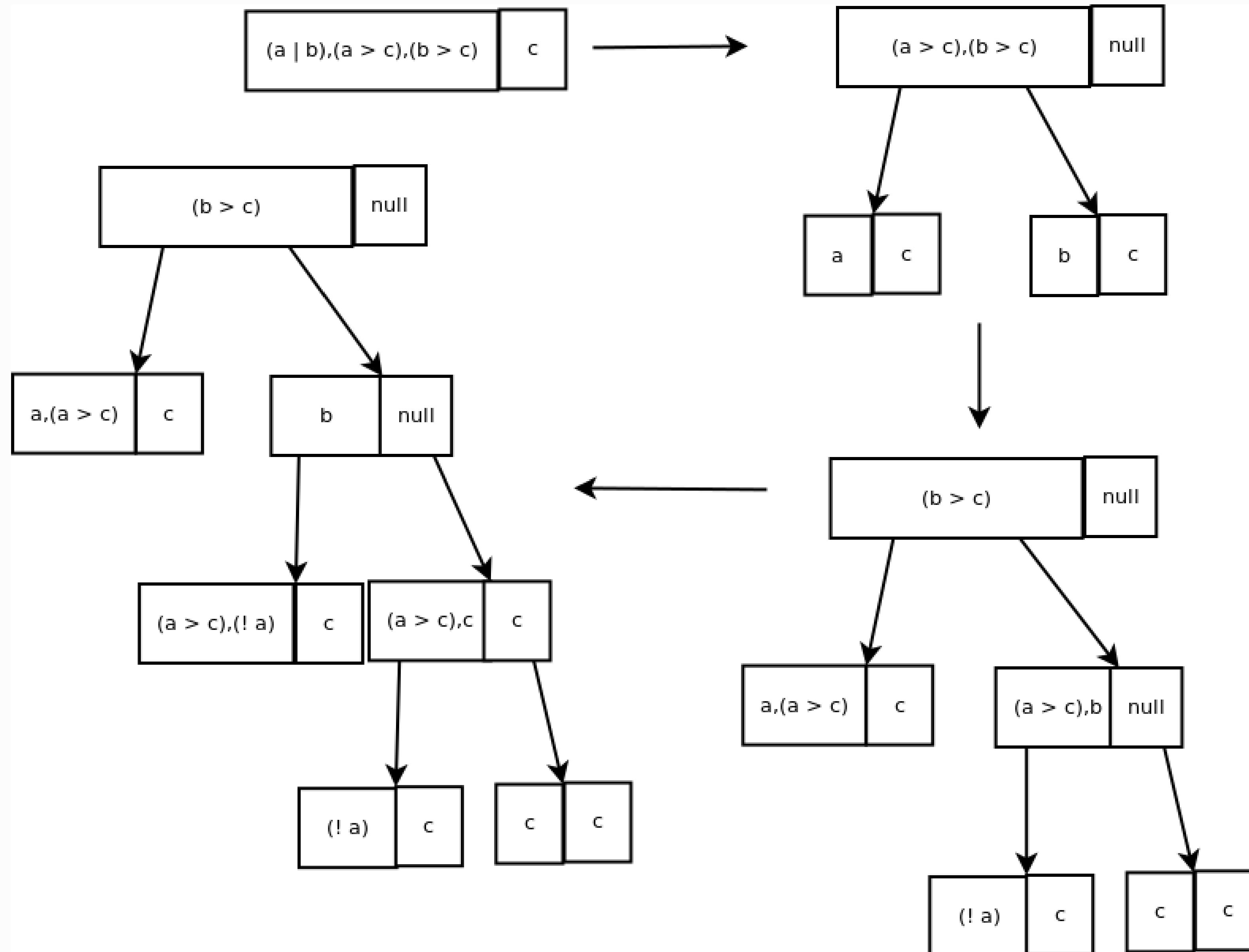
# Deduzione



# Deduzione
## Rimozione delle formule

Ricordiamo innanzitutto che le proposizioni non hanno formule ma regole. Dato che le nuove proposizioni sono create come decorazioni occorre che l'observer comunichi loro la rimozione di regole nella proposizione decorata. Come visto precedentemente questo ci garantisce che niente sia perso. Però ha l'inconveniente che può accadere che la rimozione non sia eseguita correttamente e si riapplichi una regola già applicata.

# Deduzione
## Rimozione delle formule



# Deduzione
## Rimozione delle formule

Per ovviare a questo problema occorre reiterare l'eliminazione della regola applicata finché possibile. Questo favorisce anche in un altro modo. Infatti se una formula è ripetuta più volte le ripetizioni vengono eliminate perché inutili.

# Valutazioni

La parte finale del progetto riguarda come valutare se una proposizione sia effettivamente valida o meno. Per fare ciò si ricorre alla classe Evaluation e AbstractEvaluationChecker. Evaluation permette di creare una valutazione semplice (assegnamenti a atomi). L'evaluation checker invece cerca di creare una valutazione da un insieme di formule e una formula.

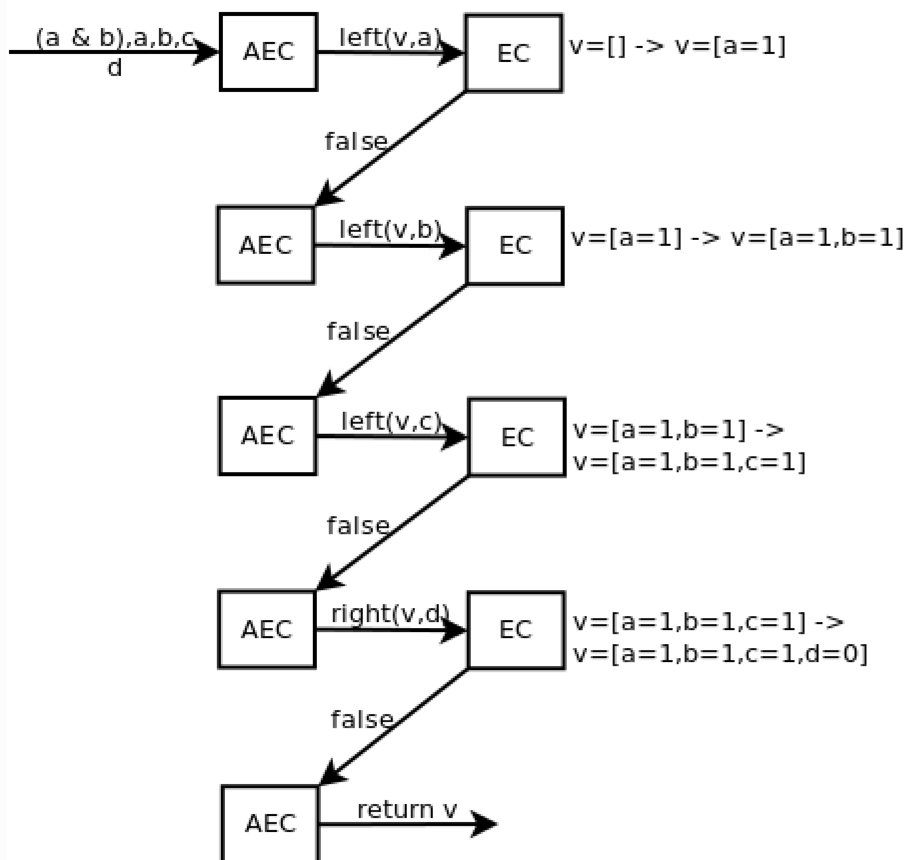
# Valutazioni

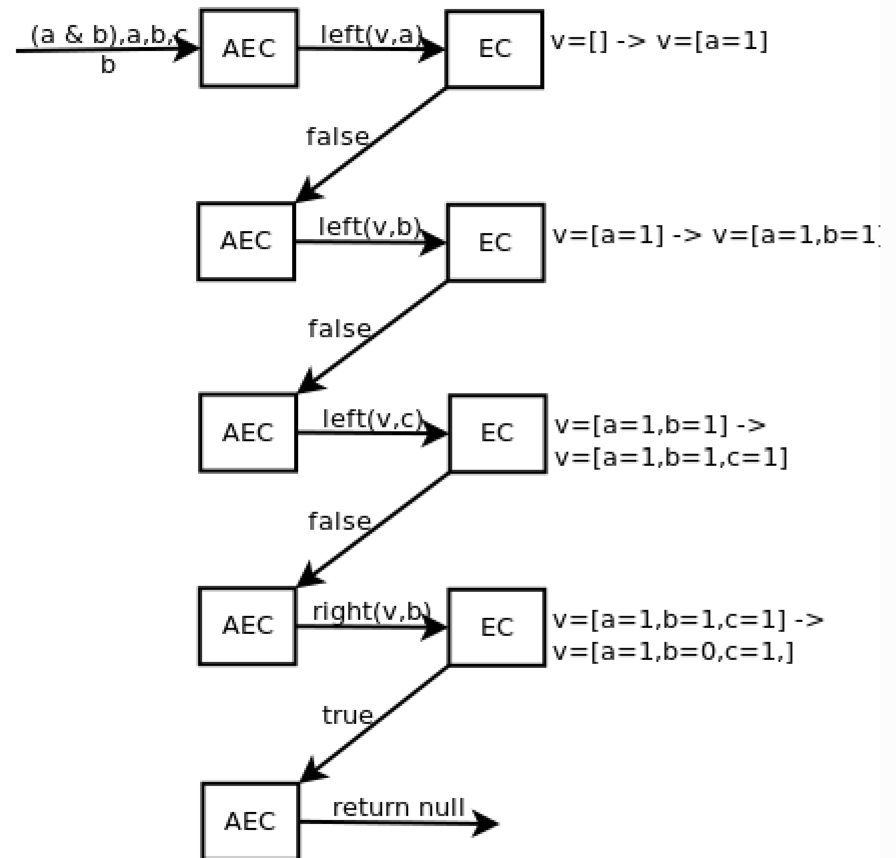
## AbstractEvaluationChecker

Questa classe viene implementata come una template. Definisce la linea guida per la crazione delle valutazioni. Preso un vettore di LeftRule e una RighRule fa valutare ai metodi lasciati astratti solo quelli con branching 0. I metodi valutano secondo un certo criterio una regola con branching zero, la inseriscono nella valutazione che il checker sta costruendo e ritornano true nel caso in cui la valutazione sia risultata contraddittoria. Se ciò avviene il checker ritorna null. Altrimenti la valutazione finale.

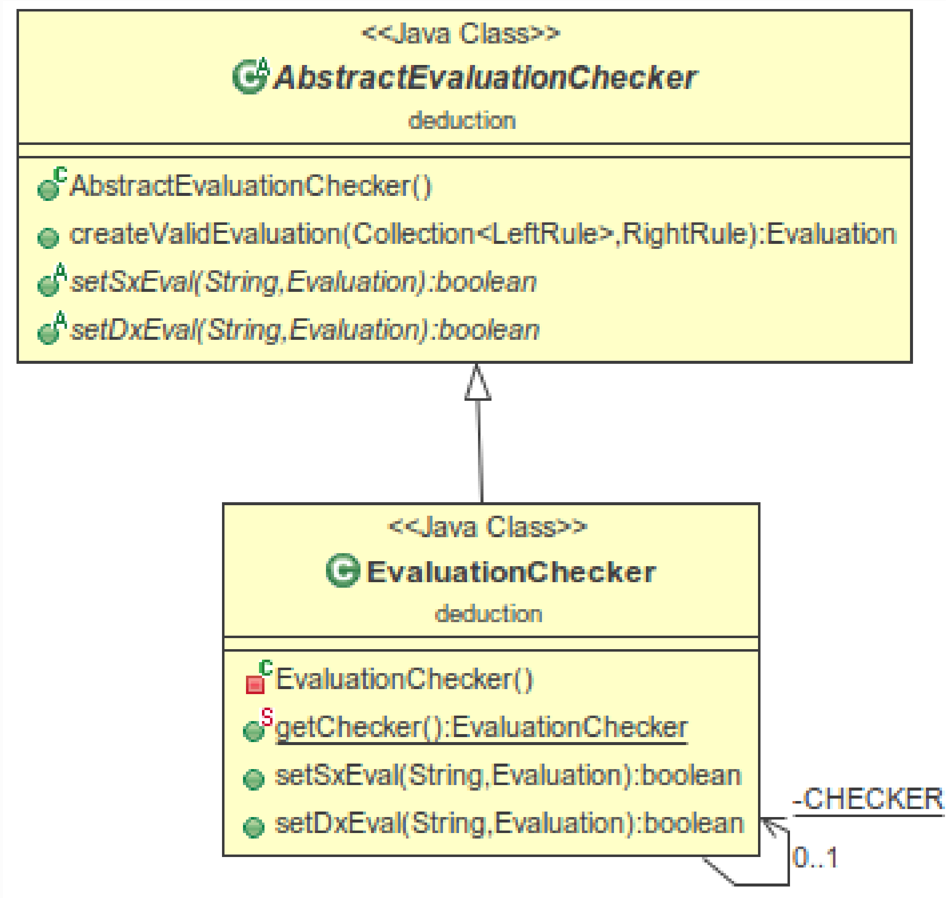
# Valutazioni
## EvaluationChecker

Per questo tipo di deduzione è stato usato EvaluationChecker che implementa AbstractEvaluationChecker. Questo nello specifico valuta a 1 le formule sinistre e a 0 la destra.



# Valutazioni



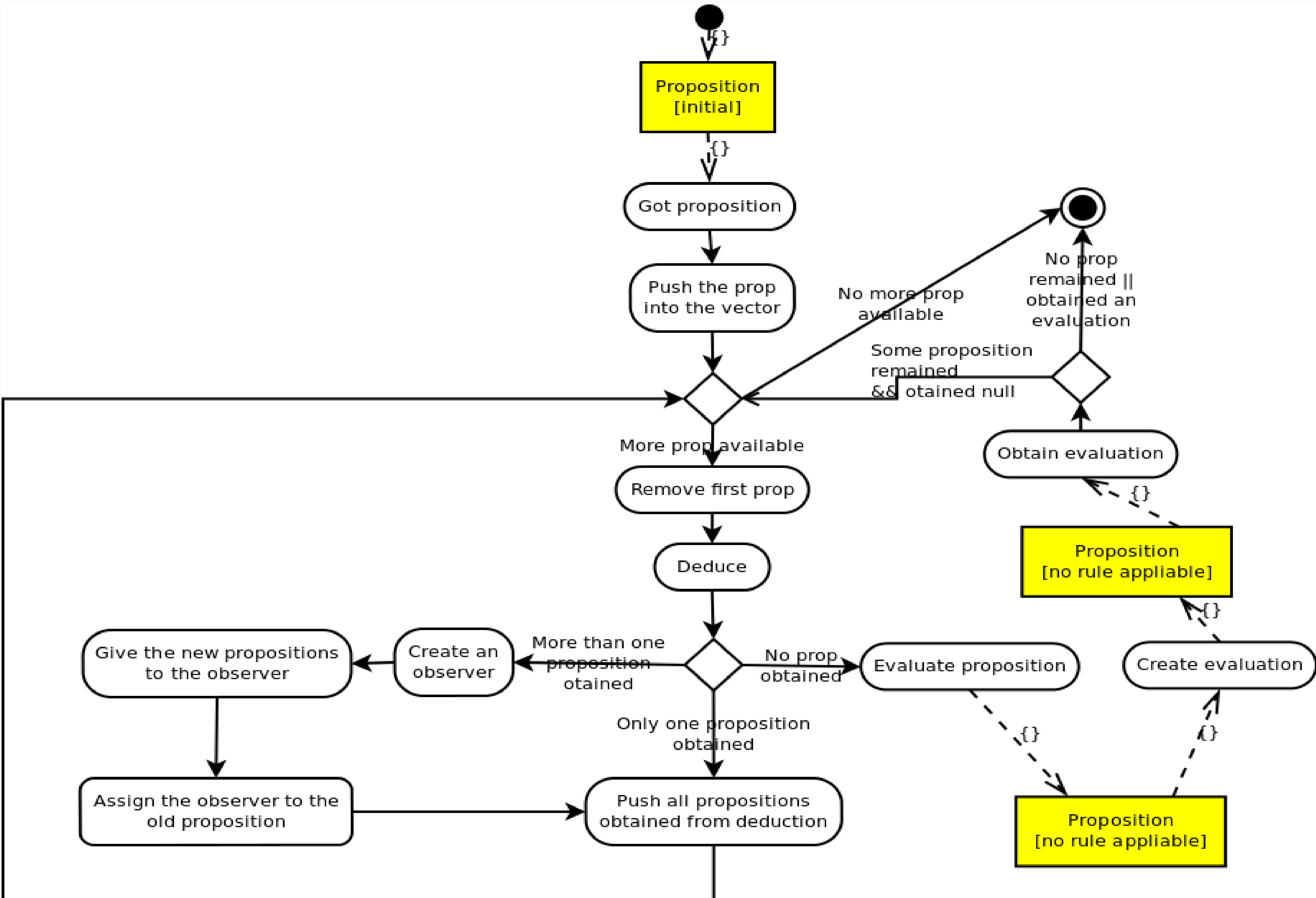
# Deduzione e Valutazioni

Ogni volta che non riesco ad ottenere proposizioni nuove applicando regole devo verificare che la proposizione usata sia valida. Perché sia valida devo ottenere null cercando di creare una valutazione. Se ottengo null significa che non è possibile invalidare la proposizione. Se non ottengo null termino il calcolo affermando che la proposizione originale non è valida.

# Deduzione e Valutazioni

Ogni volta che non riesco ad ottenere proposizioni nuove applicando regole devo verificare che la proposizione usata sia valida.
Perché sia valida devo ottenere null cercando di creare una valutazione. Se ottengo null significa che non è possibile invalidare la proposizione. Devo procedere finché non ho svuotato il vettore delle proposizioni. Se non ottengo null termino il calcolo affermando che la proposizione originale non è valida. Infatti ho dedotto una proposizione non valida, per l'invertibilità non lo è nemmeno l'originale. Inoltre la valutazione che invalida la proposizione dedotta invaliderà anche l'originale.

# Deduzioni e Valutazioni



# Deduzioni e Valutazioni

**Quando non lavorare su una nuova proposizione:**

- ➢Se trovo che la conclusione è contenuta nelle premesse sicuramente la proposizione sarà valida e non è necessario eseguire ulteriori deduzioni su questa.
- ➢Lo stesso accade se trovo una contraddizione tra le premesse.

**Nel progetto:**

- Ogni volta che trovo due letterali contraddittori o l'assurdo nelle premesse non reinserisco la proposizione nel vector delle proposizioni da processare.
- Se la conclusione è un atomo e ho lo stesso atomo tra le premesse non reinserisco la proposizione.

Questi controlli intermedi sono fatti sfruttando la classe EvaluationChecker.

# Migliorie e aggiunte

- Uso di negazioni solo prima di atomi
- Uso di “grandi congiunzioni”, xor, Sheffer,...
- Ottimizzazioni varie per assurdi
- Ecc...