ReDiA-VeriFInt

(Realizzatore Diagrammi Automatico con Verifica Formale Integrata)

Tesina di

Metodi Formali nell'Ingegneria del Software

a.a. 2007-2008

A cura di:

Fabio D'Aprano, Claudio Di Ciccio

(fdaprano@gmail.com, dc.claudio@gmail.com)

Supervisore:

Prof. Toni Mancini

DIS

(Dipartimento di Informatica e Sistemistica)

SAPIENZA - Università di Roma

Cos'è ReDiA-VeriFInt

- Il progetto pilota dell'applicazione **CASE**
  - Un ambiente di sviluppo integrato
    - in grado di assistere il progettista/programmatore in tutte le fasi di progettazione e realizzazione delle proprie applicazioni
      - dalla raccolta ed analisi dei requisiti alla produzione di codice implementativo
      - validando continuamente le sue scelte progettuali/implementative

ReDiA-VeriFInt ora

- Gestisce la modellazione di **Diagrammi UML delle Classi *Concettuali***
- Traduce i diagrammi gestiti in un equipollente insieme di formule della **logica del prim'ordine**
- Gestisce la verifica di proprietà formali del diagramma così *tradotto*
  - attraverso un **dimostratore automatico** (***Prover9***)
- È in grado di **disegnare** il Diagramma UML delle Classi Concettuali
  - tramite primitive *a basso livello*

Obiettivi del progetto

- Gestire la traduzione da insieme di oggetti ad insieme di formule...
- ... fornendo un *framework* valido per tutti i diagrammi...
- ... facilmente estendibile...
  - **Strutturazione**
  - **Interoperabilità**
  - **Modularità**

Soluzioni: *framework*: Oggetti → XML → input dimostratore (1 / 2)

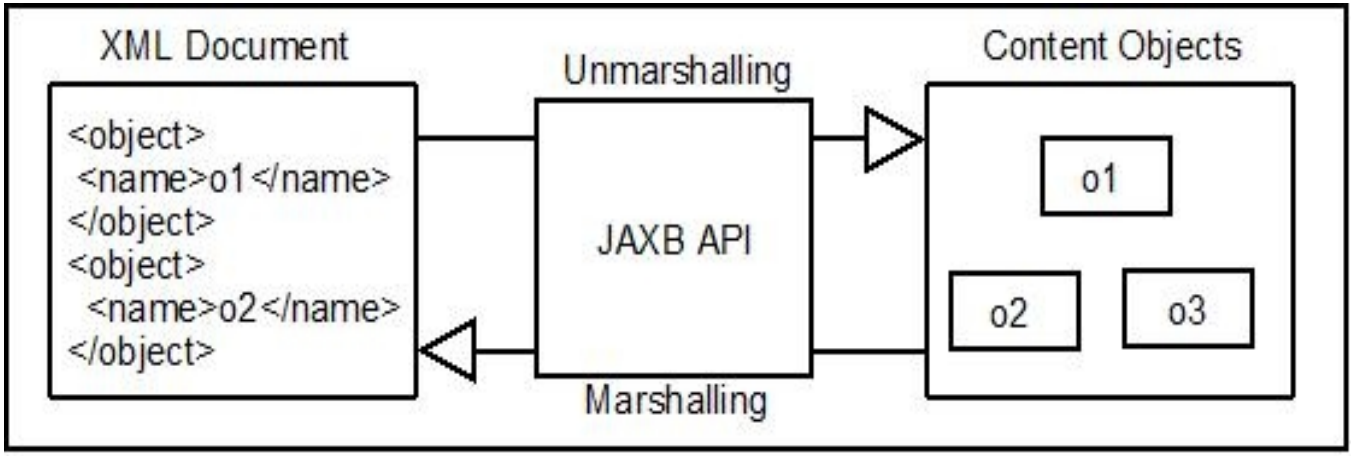
- Tra
  - la **rappresentazione ad oggetti** dei componenti del diagramma e
  - la **trasposizione in formato testo** come input per dimostratori automatici
- si frappone
  - la **rappresentazione XML** dei dati strutturati
- usando
  - ***JAXB* 2.0** (traduzione oggetti → XML)
  - **XSLT** 1.0 (XML → testo di input dimostratori)

Soluzioni: *framework*: Oggetti → XML → input dimostratore (2 / 2)

- **XML** è un linguaggio usato in vastissimi ambiti
  - uno standard che ha compiuto 10 anni nel 2008
- XML è un formato leggibile ed importabile/esportabile in vari formati
  - può fungere da DTO!
- viene demandata a XSLT, interamente, la logica di traduzione
  - non una sola linea di codice *hard-wired* nelle classi Java!

Cos'è JAXB?

- **J**ava **A**rchitecture for **X**ML **B**inding
  - Consente ***marshalling***, ***unmarshalling*** degli oggetti
  - Vantaggi di JAXB 2.0
    - No *SAX*, No *DOM* → no *parser*!
    - Sono sufficienti delle *Java Annotation* nel codice delle classi!
  - Indicato per dati con struttura complessa o mutevole



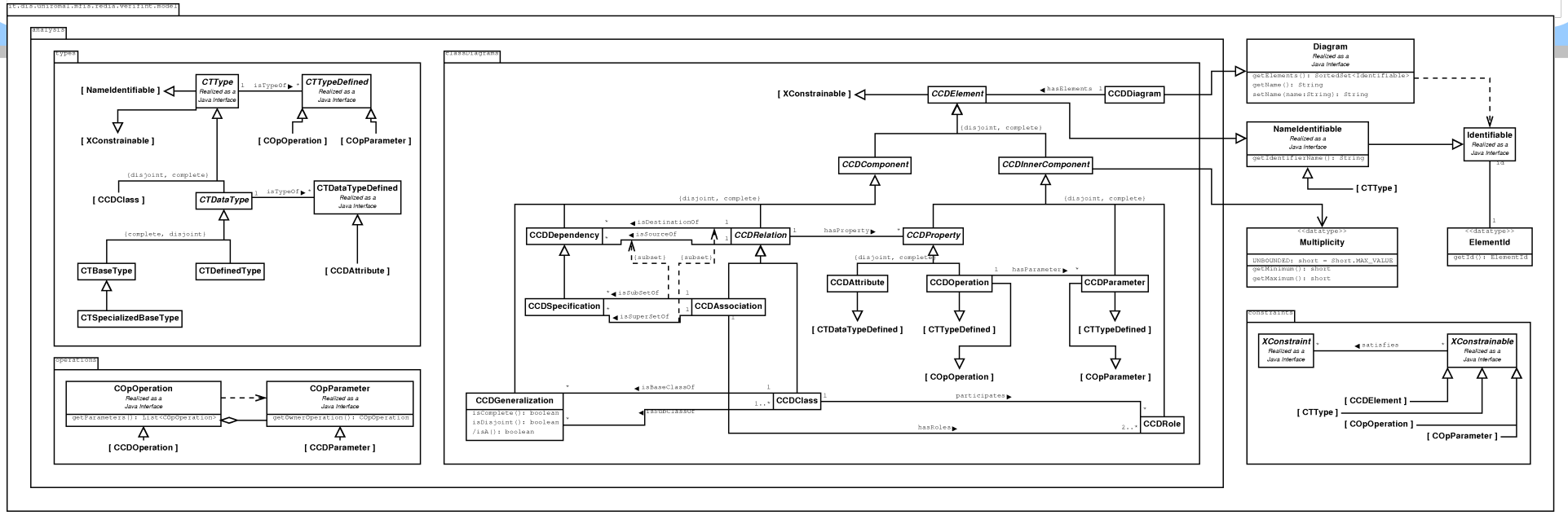
Cos'è XSLT?

- e**X**tensible **S**tylesheet **L**anguage **T**ransformations: un linguaggio standardizzato *W3C*
  - segue la **sintassi XML**
  - detta le regole di trasformazione ad un processore, il quale
    - in ***input*** riceve un **documento XML**
    - in ***output*** può produrre
      - un documento **XML**
      - un documento **HTML**
      - un documento in puro **testo**
- Può essere utile, dato un file XML, a produrre una vasta gamma di file di testo
  - input per dimostratori automatici di teoremi
  - righe di codice!

Soluzioni: architettura MVC

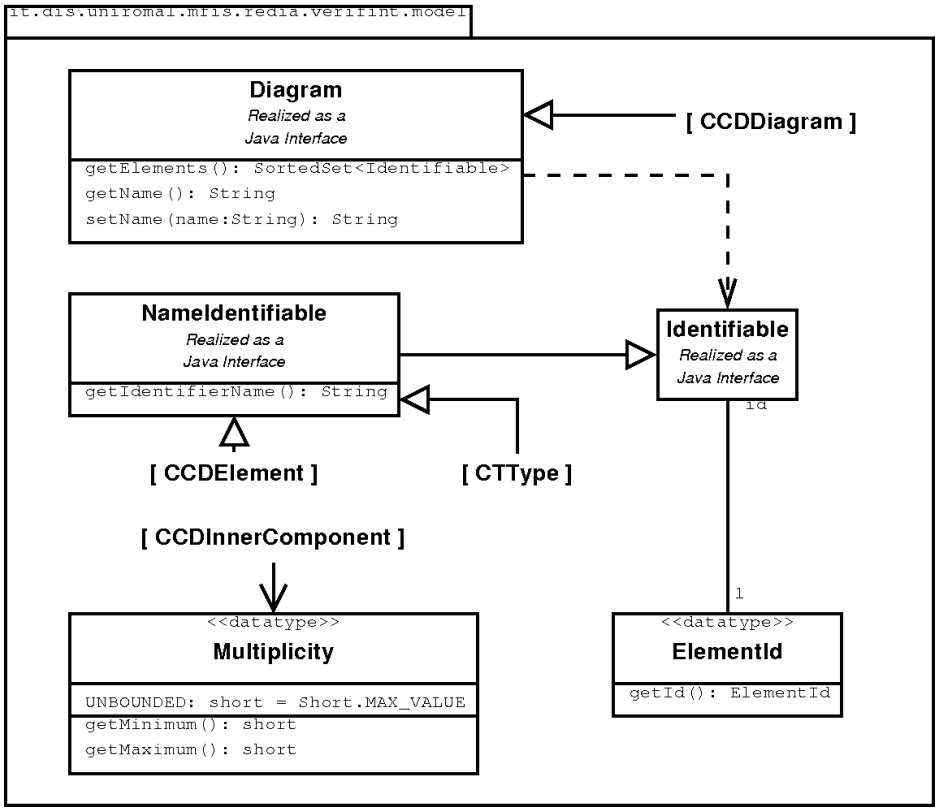
- ***M**odel **V**iew **C**ontroller*
- Garantisce interoperabilità, modularità, strutturazione
- Le classi sono divise in *package* omonimi:
  - `it.dis.uniroma1.mfis.redia.verifint.model`
  - `it.dis.uniroma1.mfis.redia.verifint.view`
  - `it.dis.uniroma1.mfis.redia.verifint.control`

Model (1 / 7): Sguardo d'insieme



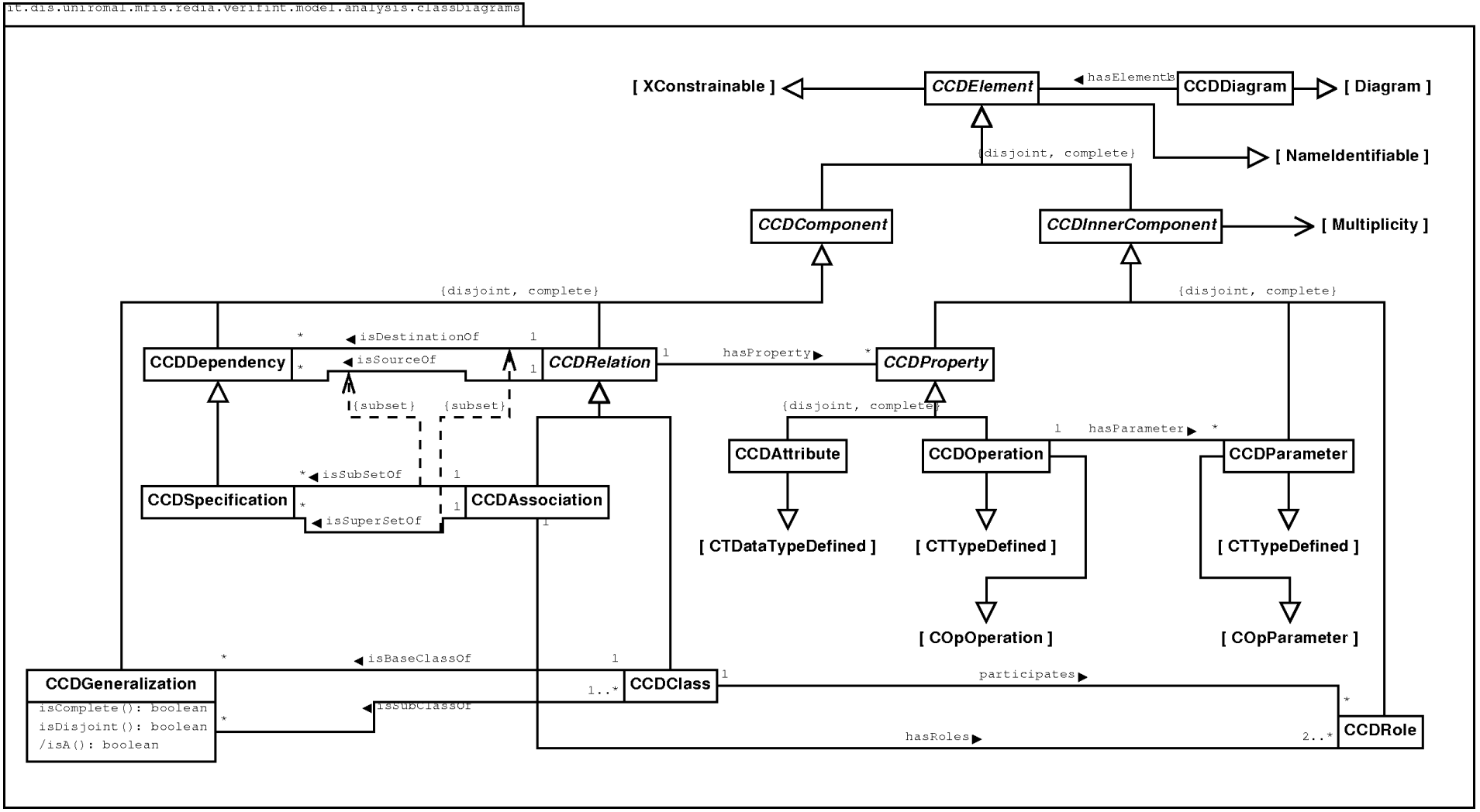
- Basato sullo studio di M. Proni per la sua tesi
  - progetto *OOPS*, seguito dal Prof. Mancini
- Il *package* viene diviso a sua volta in *sub-package*, ognuno dei quali rappresentante una differente semantica per gli oggetti classificati
  - Elementi del Diagramma delle Classi, Tipi, Operazioni, Vincoli

Model (2 / 7): Ambito generale



- `Diagram`
  - ogni diagramma estenderà questa classe (implementerà l'interfaccia)
  - ha una collezione di:
- `Identifiable`
  - prevede la generazione di un *ID* (`ElementId`)
- `NameIdentifiable`
  - autogenera un **nome univoco**
    - che verrà usato come **simbolo di predicato**!

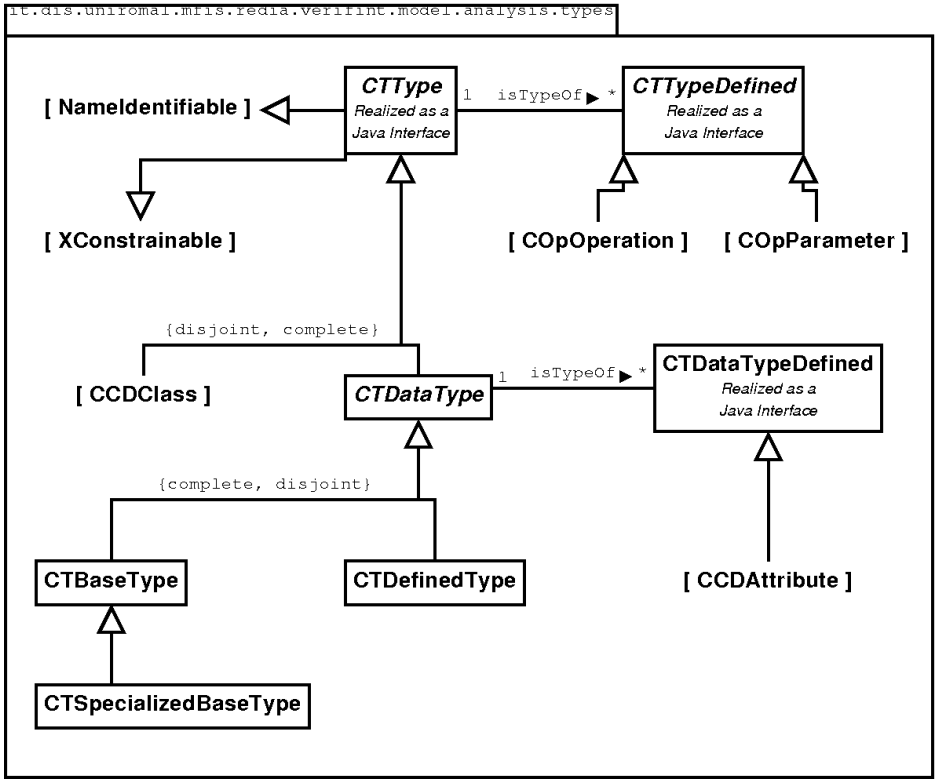
Model (3 / 7): Diagramma Concettuale delle Classi (1 / 2)



Model (4 / 7): Diagramma Concettuale delle Classi (2 / 2)

- `CCDDiagram` è il collettore di elementi `CCDElement`
- Da `CCDElement` ha inizio la gerarchia di ogni componente del Diagramma Concettuale delle Classi
  - possono essere
    - elementi *autosufficienti* (e.g. classi)
      - `CCDComponent`
    - elementi la cui identificazione prevede un riferimento ai precedenti (e.g. attributi)
      - `CCDInnerComponent`
- A questo livello, dei componenti si descrive una semantica relativa all'appartenenza e al ruolo all'interno del diagramma
  - ad esempio, le classi sono anche tipi per i parametri, ma ciò non viene dettagliato in questa parte, bensì nella prossima

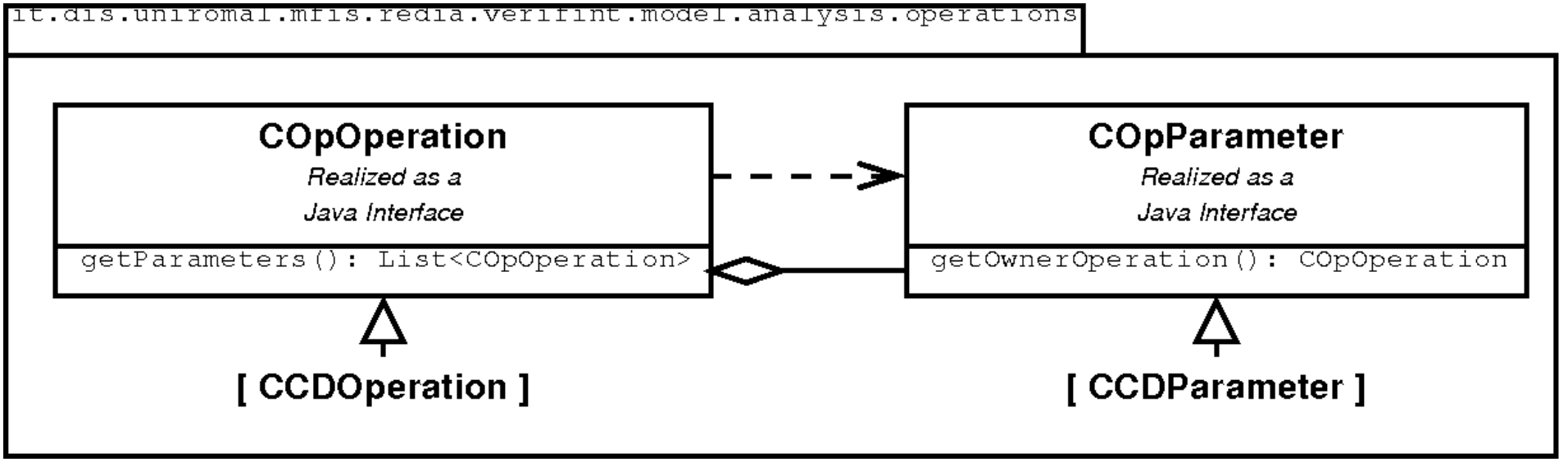
Model (5 / 7): Tipi



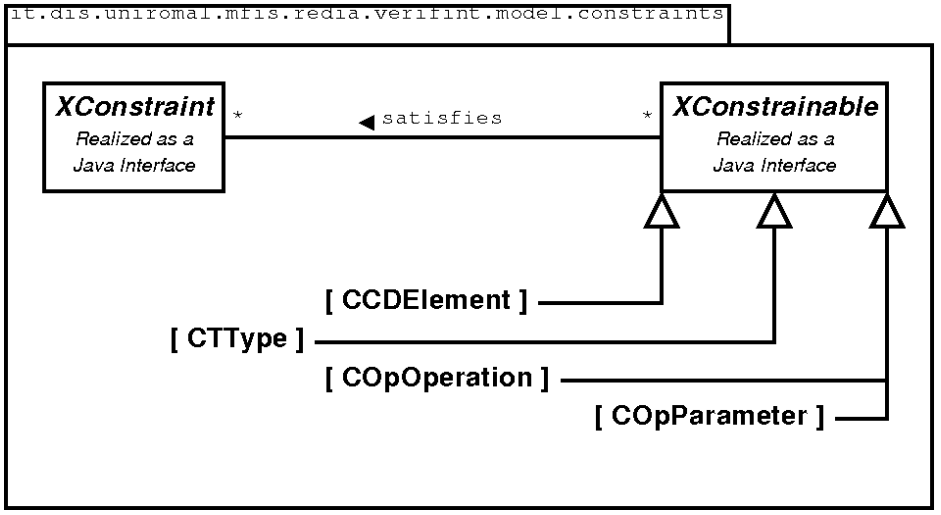
- Questa sezione verrà estesa ed approfondita dai prossimi sviluppatori

- `CTType`
  - genitore della gerarchia
    - `CCDClass`
      - (v. slide precedente)
    - `CTDataType`
  - la gerarchia serve affinché si possa rappresentare il fatto che
    - gli attributi possono essere di un tipo di dato
    - i parametri di ingresso o di ritorno delle operazioni possono essere di qualunque tipo (anche delle classi, intese come tipi)

Model (6 / 7): Operazioni



Model (7 / 7): Vincoli



- Questa sezione rappresenta essenzialmente il punto di aggancio per gli sviluppatori che renderanno gestiti dal programma anche i vincoli
- I vincoli saranno classi figlie di `XConstraint`
- Ogni componente può essere soggetto a vincoli
  - `XConstrainable`

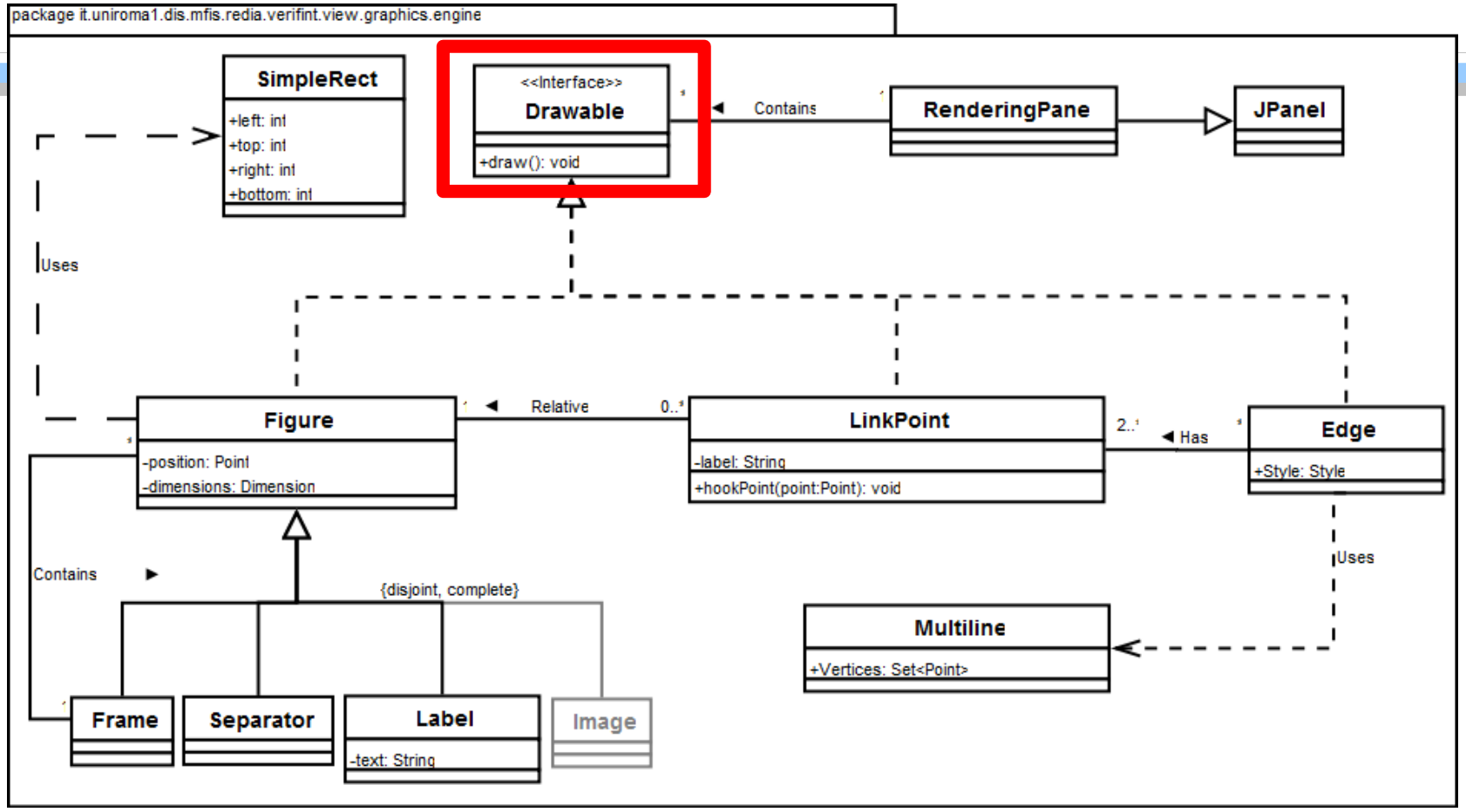
View (1 / 8): Sguardo d'insieme

- Questo livello contiene la logica orientata alla sola interazione con l'utente finale
- Tutta la logica applicativa, e la gestione dei flussi di dati, si trova a livello *Control*
- Lo strato è diviso in due ambiti
  - **Motore grafico**
    - per il ***rendering*** dei diagrammi a video
    - `it.uniroma1.dis.mfis.redia.verifint.view.graphics.engine`
  - **Widget di I/O**
    - per la visualizzazione e l'aggiornamento dei dati tramite ***interfaccia grafica***
    - `it.uniroma1.dis.mfis.redia.verifint.view.widgets`

View (2 / 8): Motore grafico (1 / 6)

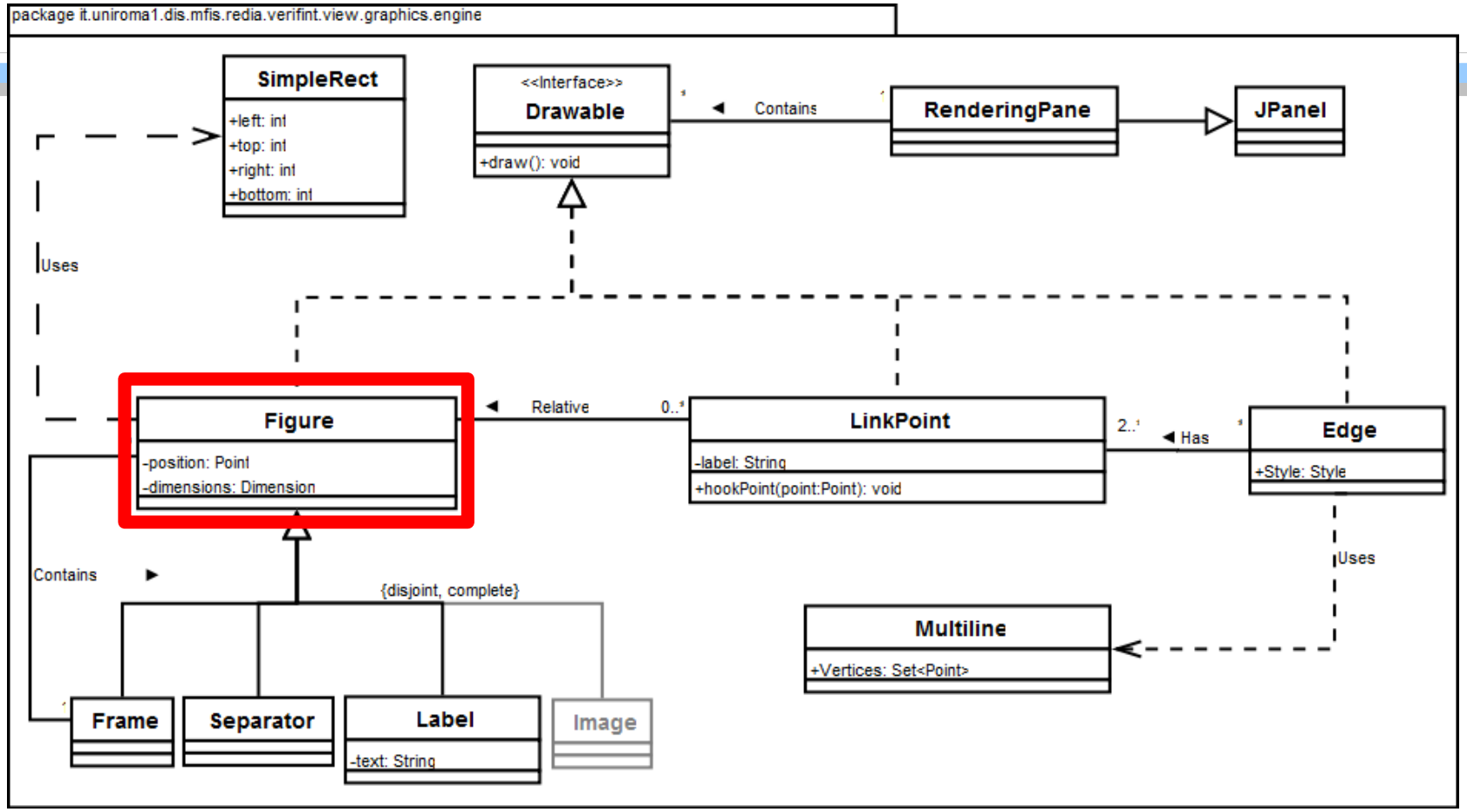
- Offre un supporto alla visualizzazione dei diagrammi
- Opera a basso livello
  - no semantica UML
- Progettato in modo da essere estendibile per il disegno di ogni tipo di diagramma

View (3 / 8): Motore grafico (2 / 6)



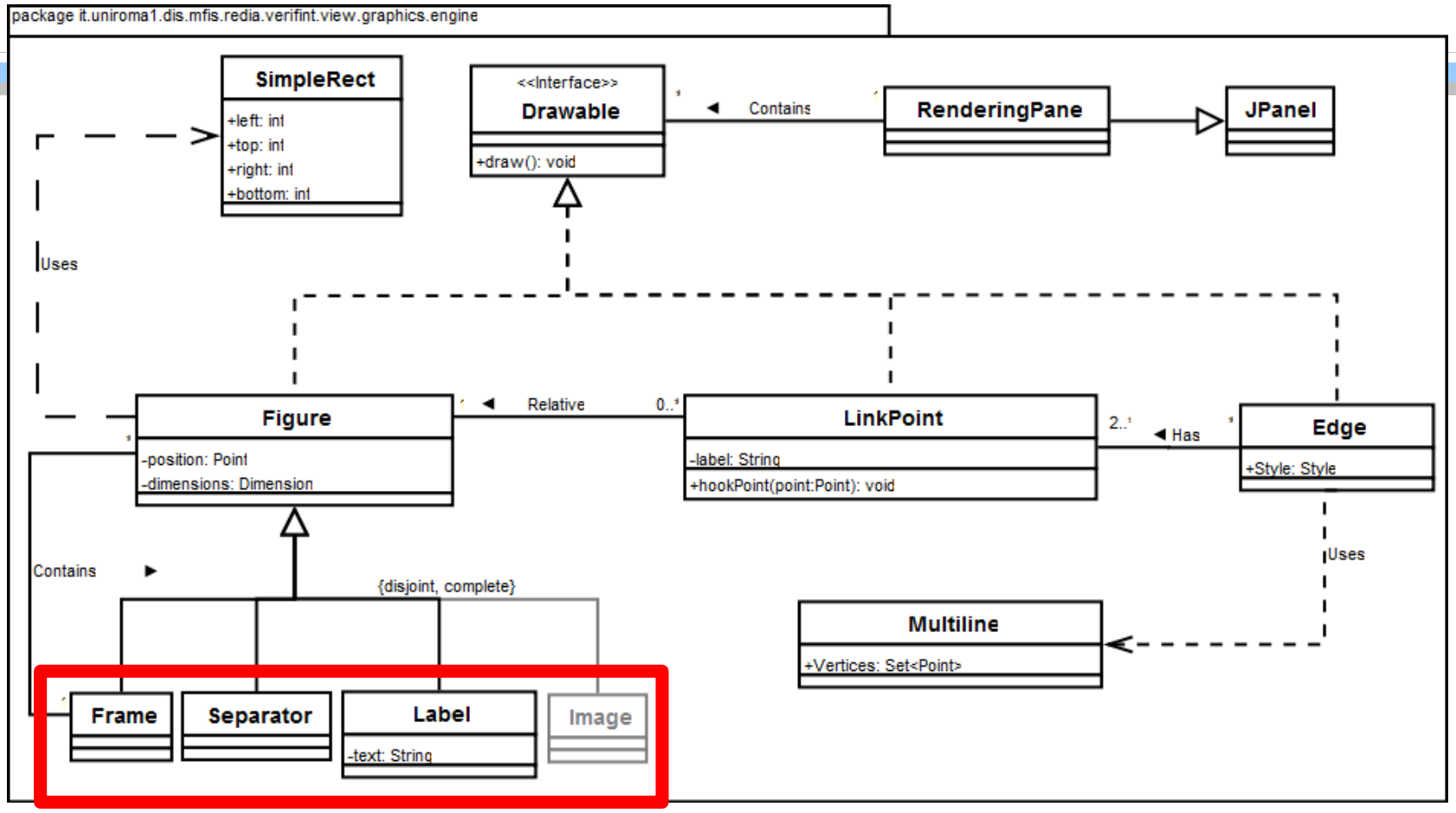
- Interfaccia `Drawable`:
  - Definisce un oggetto generico *disegnabile* su schermo
  - Metodo `draw()`

View (4 / 8): Motore grafico (3 / 6)



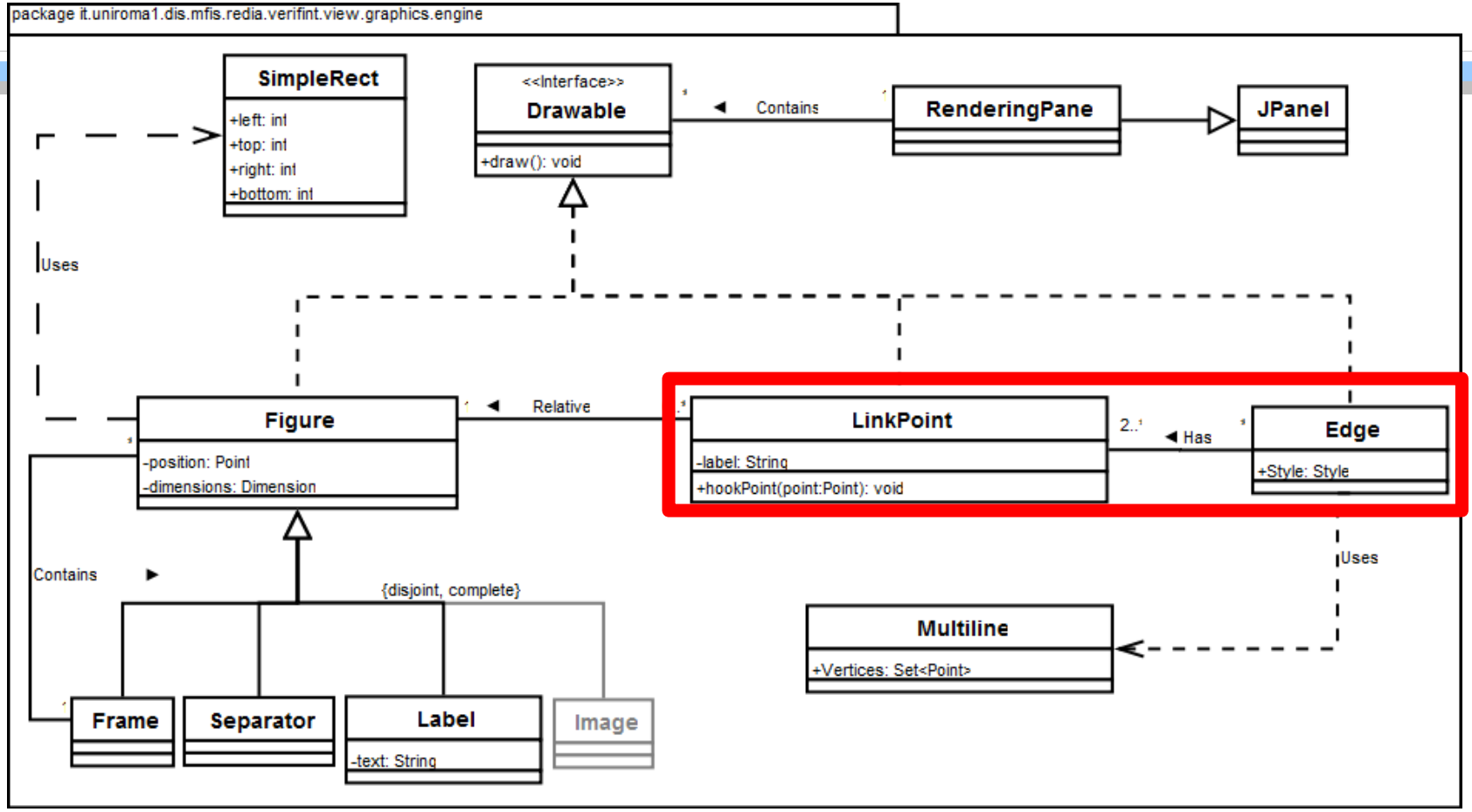
- Classe `Figure`:
  - Definisce un oggetto contenuto in una "*bounding box*"

View (5 / 8): Motore grafico (4 / 6)



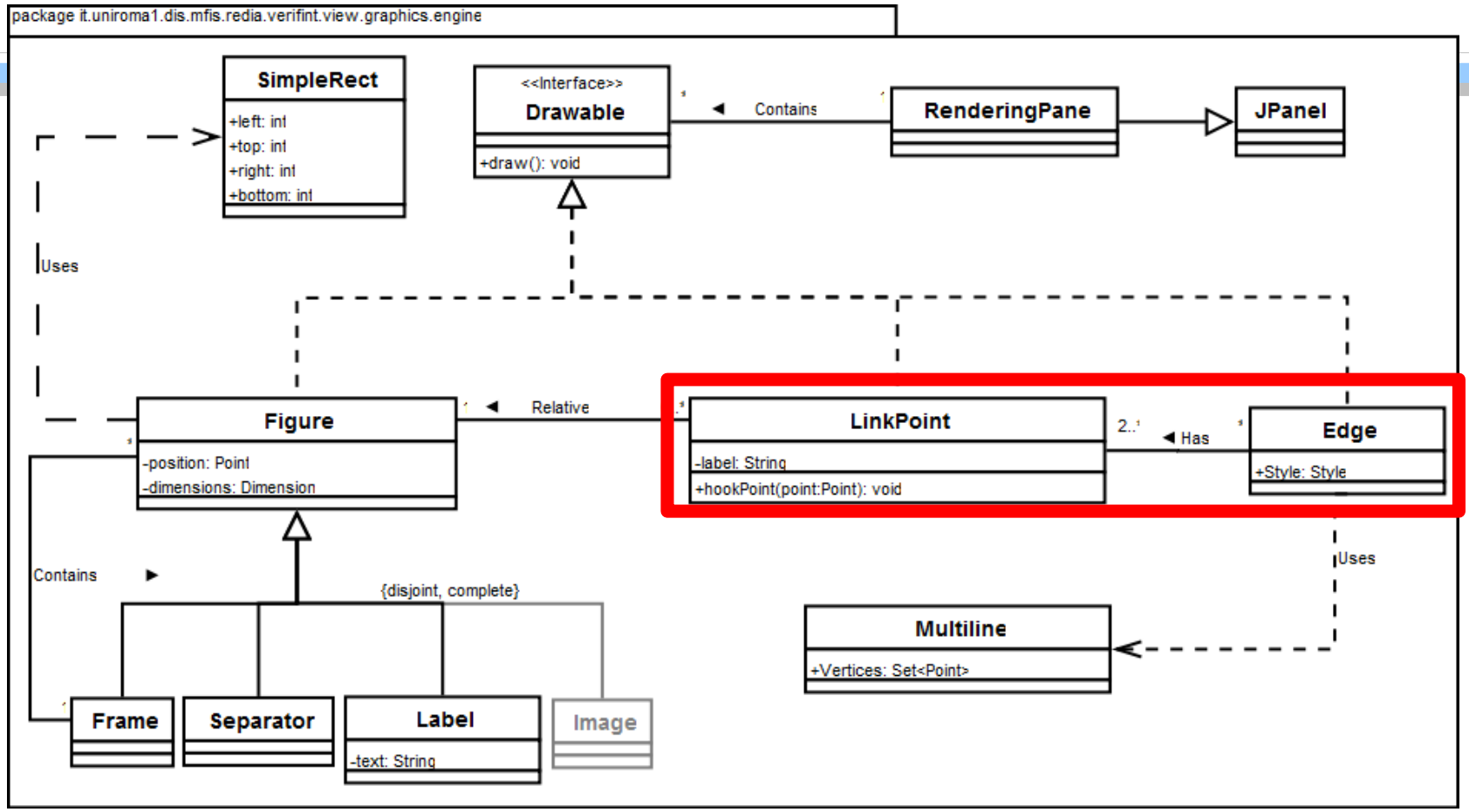
- Classi figlie di `Figure`:
  - Frame: per la rappresentazione grafica di classi e interfacce, contiene oggetti `Label` e `Separator`
  - Possibile implementazione di altri oggetti grafici, ad esempio `Image`

View (6 / 8): Motore grafico (5 / 6)



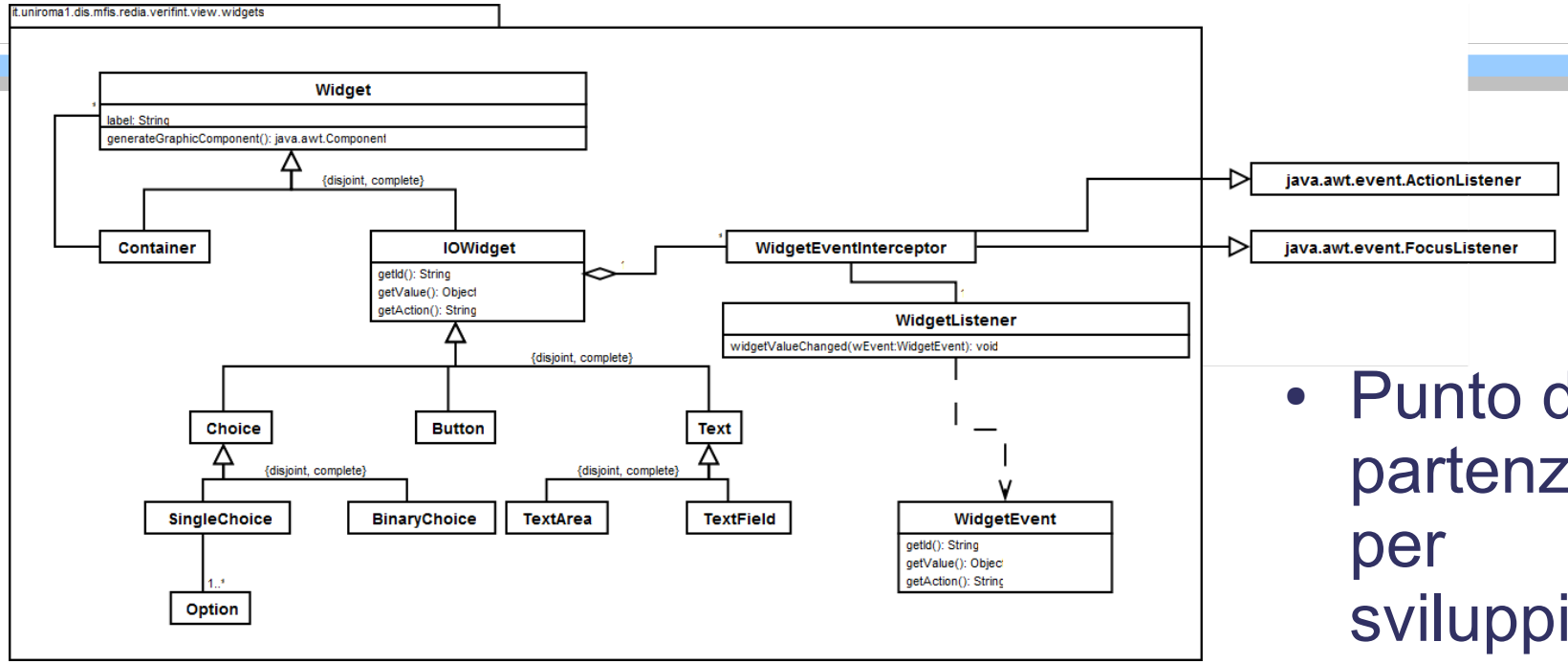
- Classi `LinkPoint` e `Edge`:
  - Un `Edge` rappresenta un arco di collegamento e ha da 2 a N `LinkPoint`

View (7 / 8): Motore grafico (6 / 6)



- Ogni `LinkPoint` e' riferito ad una sola Figure
- Il `LinkPoint` gestisce dinamicamente la posizione dei punti di aggancio degli archi, tramite il metodo `hookPoint()`

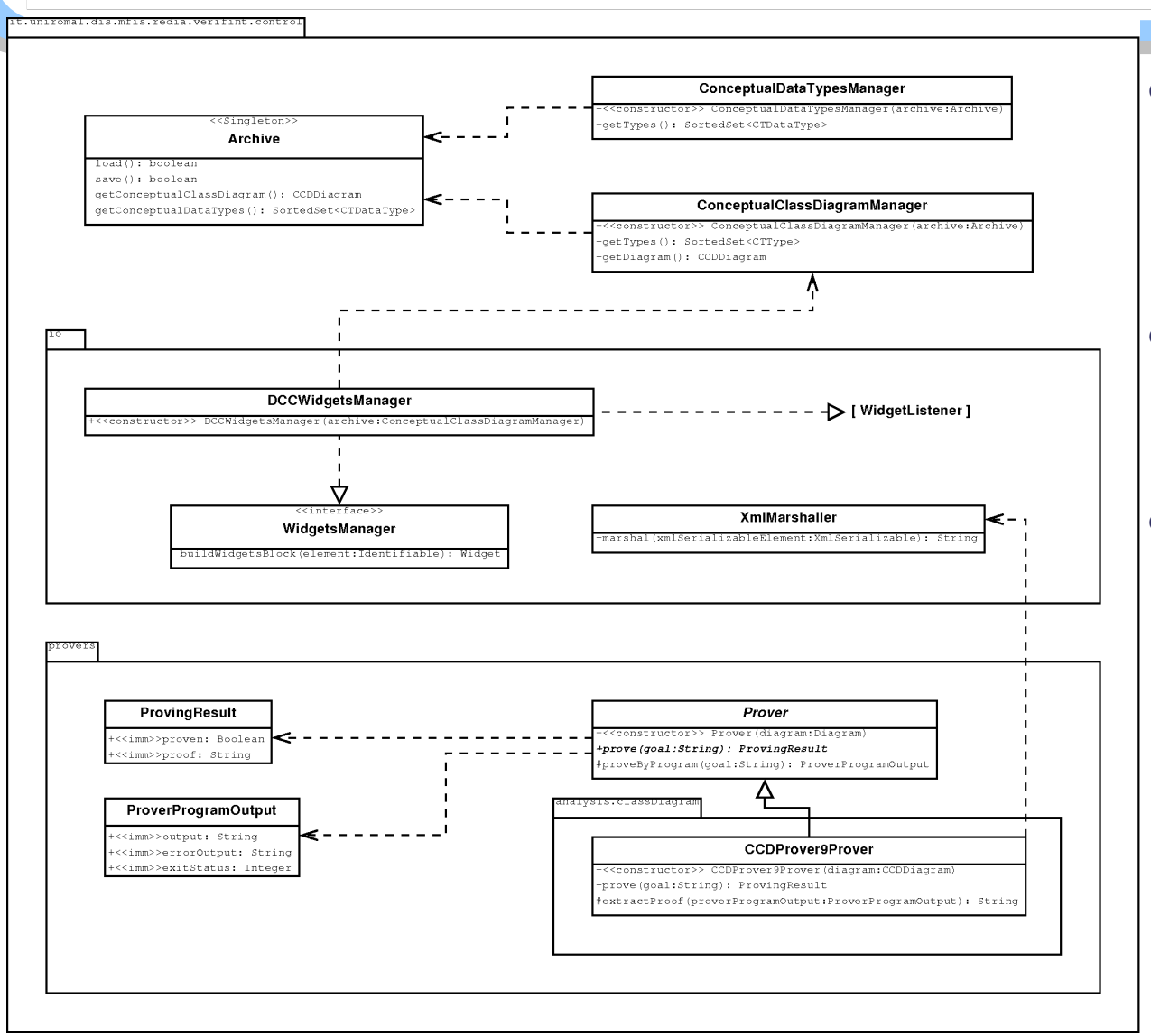
View (8 / 8): Widget



- Punto di partenza per sviluppi futuri!

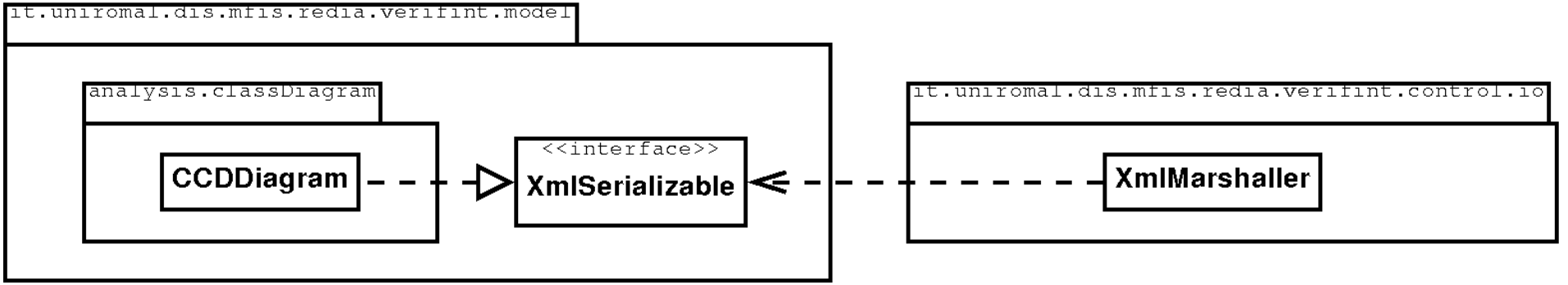
- `Widget`
  - L'idea: rappresentare oggetti grafici interattivi *ad alto livello*
    - Ogni `Widget` genera un oggetto compatibile con *Java AWT* (*Swing*) mediante `generateGraphicComponent()`
    - `WidgetEventInterceptor` cattura gli eventi ne demanda la gestione al *Listener* di livello *Control* (`WidgetListener`)

Control (1 / 4): Sguardo d'insieme



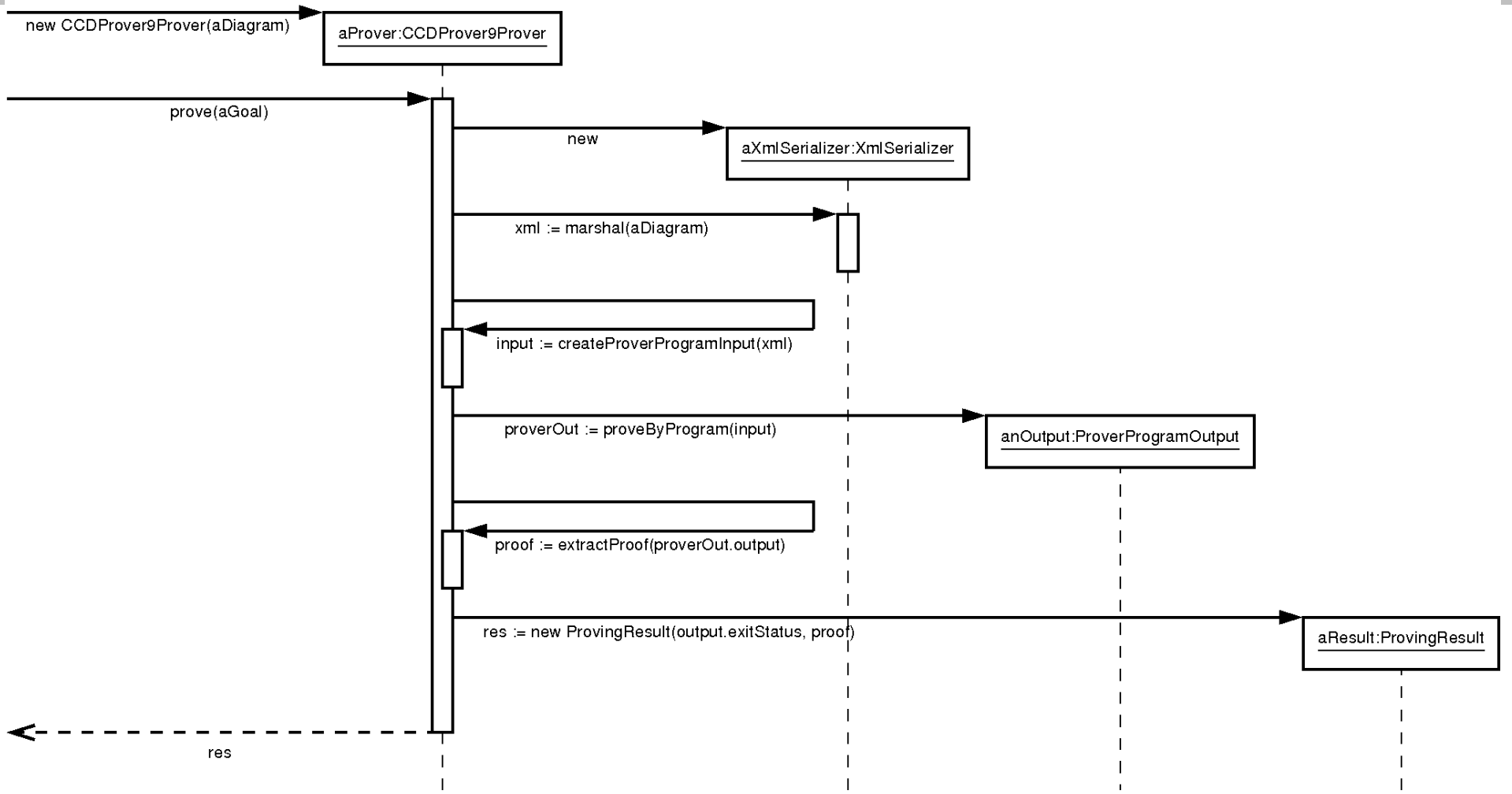
- `Archive`
  - contenitore dei dati applicativi
- `XXXManager`
  - gestori dei dati
- *sub-package*
  - `io`
    - gestione dei **`Widget`**, e del ***marshalling*** *XML*
  - `provers`
    - gestione delle verifiche automatiche delle **proprietà** dei diagrammi

Control (2 / 4): Serializzazione XML



- `XmlMarshaller` gestisce oggetti `XmlSerializable`
  - ogni diagramma è un `XmlSerializable`
    - anche altri tipi di diagramma dovranno esserlo
  - non deve implementare metodi diversi per serializzare diagrammi diversi
  - segue la tecnica di separazione dei *behaviour* tramite interfacce già vista in precedenza

Control (3 / 4): Serializzazione XML e traduzione input dimostratore



Realizzazione: Model (1 / 2): regole di serializzazione XML (JAXB)

```
public abstract class CCDProperty extends
    CCDInnerComponent {

  ...

  @XmlJavaTypeAdapter(

    type=CTType.class,

   value=CTTypeXmlImplementation.Adapter.class)

  @XmlAttribute(required=true)

  @XmlIDREF

  protected CTType type;


  @XmlTransient

  protected CCDRelation ownerRelation;

  ...

}
```

- Tramite l'inserimento di annotazioni *JAXB*, nelle classi del livello *Model* sono implicitamente dettagliate le regole di serializzazione *XML*
  - Si definiscono i ***DTO***
    - Incapsulamento delle informazioni!
  - Segue la tecnica usata nell'O/R mapping in *Java Persistence API*

Realizzazione: Model (2 / 2): implementazione

- Ulteriore separazione in ***meta-package***
  - e.g. `_dataTypes`, `_interfaces`
- Uso delle tecniche **ClasseAssociazione**
  - per le associazioni con *responsabilità doppia*
    - consente il mantenimento della coerenza dei dati aggiungendo o sottraendo *handle* ad ambedue le istanze
      - e.g. `CCDRelation_hasProperty_CCDProperty.addLink(CCDRelation, CCDProperty)`
- Uso del pattern **Factory**
  - laddove comparissero, su associazioni con responsabilità doppia, *vincoli di partecipazione*
    - costruttori ad accesso ristretto al *package* di appartenenza per le classi
      - e.g. `CCDClass_Factory` ha il metodo:
        `public createInstance(CCDDiagram, String): CCDClass`
        mentre il costruttore di `CCDClass` è ad accesso *ristretto al package*:
        `CCDClass(String)`

Realizzazione: Control: serializzazione XML

```
public class XmlMarshaller {
  public String marshal(XmlSerializable xmlSerializableElement) {
    Object xmlSerializableObject = (Object)xmlSerializableElement;
    StringWriter marshalledInstanceWriter = new StringWriter();
    try {
      JAXBContext jaxbContext = JAXBContext.newInstance(
          xmlSerializableObject.getClass());
      Marshaller marshaller = jaxbContext.createMarshaller();
      ...
      marshaller.marshal(
          xmlSerializableElement, marshalledInstanceWriter);
    }
    catch (Exception e) { ... }
    return marshalledInstanceWriter.toString();
  }
}
```

- servono ben poche righe di codice, a livello *Control*, per effettuare il *marshalling*!
  - perché le regole di serializzazione sono state definite negli oggetti stessi
  - le regole di serializzazione di altri elementi, per altri diagrammi, saranno incapsulati in quelle classi, e non in estensioni di `XmlMarshaller`
    - **information hiding**

Realizzazione: Control: Traduzione XML → input per *Prover9* (1 / 2)

```
public abstract class Prover {

  protected String createProverProgramInput(String diagramXml) {

    ...

    try {

      // read the XML input

      Source xmlSource = new StreamSource(new StringReader(diagramXml));

      // read the XSLT sheet

      Source xsltSource = new StreamSource(new FileReader(

              System.getProperty("user.dir") + this.getXsltPath()));

      // transform the XML following the XSLT templates

      TransformerFactory transFactory = TransformerFactory.newInstance();

      Transformer trans = transFactory.newTransformer(xsltSource);

      StringWriter newXmlWriter = new StringWriter();

      trans.transform(xmlSource, new StreamResult(newXmlWriter));

      ...

    }    catch (Exception e) { ... } ... }

}
```

- Seguendo la stessa tecnica, viene delegato ai soli fogli *XSLT* il compito di descrivere le regole di traduzione

Tecnologie: JAXB (1 / 2): utilizzare JAXB 2.0: annotazioni

- *JAXB* 2 i basa su istruzioni Java denominate ***annotazioni***
  - Annotazioni più utilizzate:
    - **`@XmlRootElement`**
      - Si riferisce a una classe e indica che essa sarà l'elemento radice del documento XML
    - **`@XmlAccessorType(XmlAccessType)`**
      - Si riferisce ad una classe e, tramite il valore `XmlAccessType`, definisce quali elementi di tale classe saranno serializzati
        - distinguendo tra attributi, proprietà pubbliche, ...
    - **`@XmlElement`**
      - Si riferisce a una proprietà di una classe e definisce un *mapping* con un elemento *XML* (prevalendo sulla politica definita da `XmlAccessorType`)
    - **`@XmlTransient`**
      - Come per `XmlElement`, si riferisce a una proprietà di una classe specificando che essa non avrà alcun corrispondente su *XML* (prevalendo sulla politica definita da `XmlAccessorType`)

Tecnologie: JAXB (2 / 2): *marshalling* e *unmarshalling*

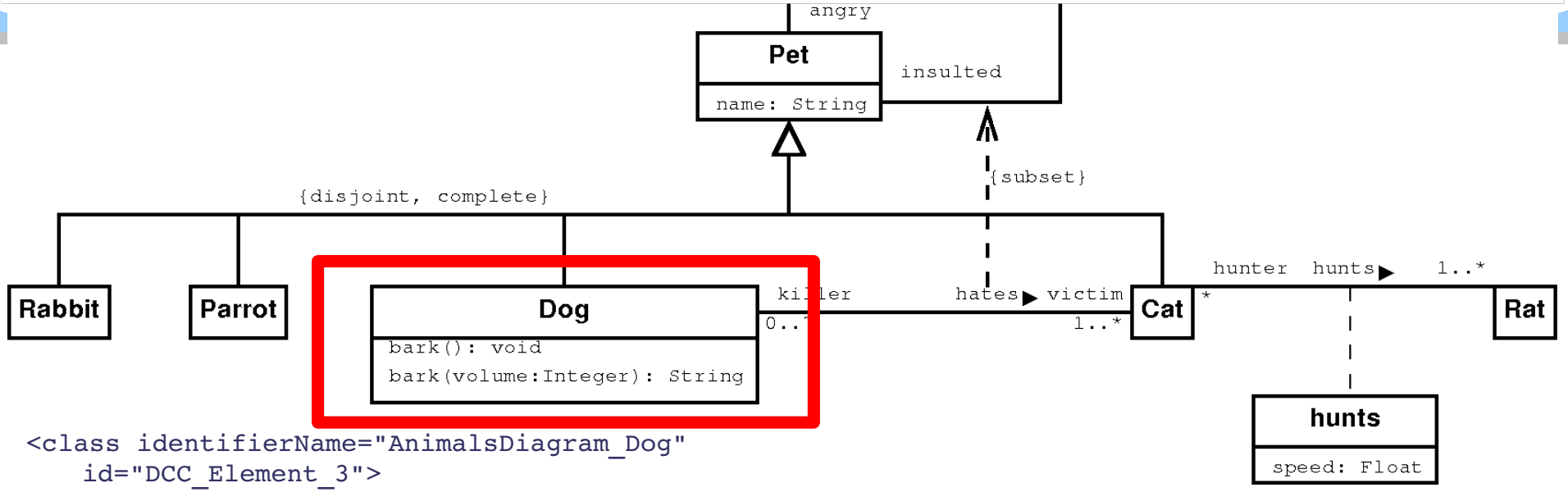
- *Marshalling*

```
Marshaller marshaller = context.createMarshaller();
marshaller.marshal(
    xmlSerializableObject,
    new FileOutputStream("output.xml"));
```

- *Unmarshalling*

```
Unmarshaller unmarshaller = context.createUnmarshaller();
MyXmlSerializableObject xmlSerializableObject =
    (MyXmlSerializableObject)
    unmarshaller.unmarshal(
        new File("input.xml"));
```

JAXB in azione




```
<class identifierName="AnimalsDiagram_Dog"
    id="DCC_Element_3">
 <name>Dog</name>
 <properties>
  <operation
  identifierName="AnimalsDiagram_Dog_bark_void"
    id="DCC_Element_4">
   <name>bark</name>
   <multiplicity maximum="1" minimum="0"/>
   <inputParameters/>
   <returnTypeString>void</returnTypeString>
  </operation>
<operation type="String"
    identifierName="AnimalsDiagram_Dog_bark_vol
    ume_Integer_String"
  id="DCC_Element_5">
   <name>bark</name>
   <multiplicity maximum="1" minimum="0"/> ...
```

```
... <inputParameters>
    <parameter type="Integer"
    identifierName=
    "AnimalsDiagram_Dog_bark_volume_Integer_Str
    ing__parameter__volume"
    id="DCC_Element_6">
     <name>volume</name>
     <multiplicity maximum="1" minimum="0"/>
    </parameter>
   </inputParameters>
   <returnTypeString>String</returnTypeString>
  </operation>
 </properties>
</class>
```

Tecnologie: XSLT (1 / 3): cos'è XSLT

- **XSLT** (e**X**tensible **S**tylesheet **L**anguage **T**ransformations) è
  - un linguaggio standardizzato dal W3C
  - scritto secondo la sintassi XML
  - consente di trasformare la struttura di un documento XML
- L'applicazione più consueta di *XSLT* è convertire un documento *XML* in un documento *XHTML*
  - qualunque formato di testo è ammesso in uscita al processamento *XSLT*
    - anche testo semplice
    - come nel nostro caso, un documento la cui sintassi sia quella delle direttive di input di un dimostratore automatico di teoremi

Tecnologie: XSLT (2 / 3): processori, potere computazionale

- Il ruolo del **processore** *XSLT* è
  - applicare le regole di trasformazione
    - date da un foglio di stile *XSLT*
  - ad un documento di origine *XML*
- *XSLT* è un **linguaggio dichiarativo**
  - descrive il *trasferimento dei dati*
  - senza istruzioni procedurali
    - si possono dichiarare variabili, ma non aggiornare i loro valori in seguito.
- Consente la **ricorsione** → è **Turing-completo**
  - per questo, possiamo delegare ad esso interamente la manipolazione dei documenti *XML* (tesi di *Church-Turing*)
    - senza usare una sola riga di codice Java inerente regole di trasformazione

Tecnologie: XSLT (3 / 3)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="">
<xsl:output method="text" encoding="utf-8"
    omit-xml-declaration="yes" indent="no"/>
    <!-- regola sul nodo root! -->
    <xsl:template match="/">
        <!-- output testuale -->
        <xsl:text><![CDATA[Del testo]]></xsl:text>
        <!-- output ricavato da un valore -->
        <xsl:value-of select="ccdDiagram/name"/>
        <!-- applicazione di altre regole! -->
        <xsl:apply-templates select="//properties"/>
    </xsl:template>

    <xsl:template match="properties">
    <!-- altri processamenti -->
    ...
    </xsl:template>

    <!-- altre regole -->
</xml:stylesheet>
```

- Le regole sono elencate in *template*
- le espressioni di *selezione* e *test* sono espresse in ***XPath***
- Esistono altri costrutti che consentono processamenti condizionati
  - `<xsl:for-each/>`
  - `<xsl:if/>`
  - `<xsl:choose>`
    `<xsl:when/>`
    `</xsl:choose>`

XSLT in azione

- Input XML

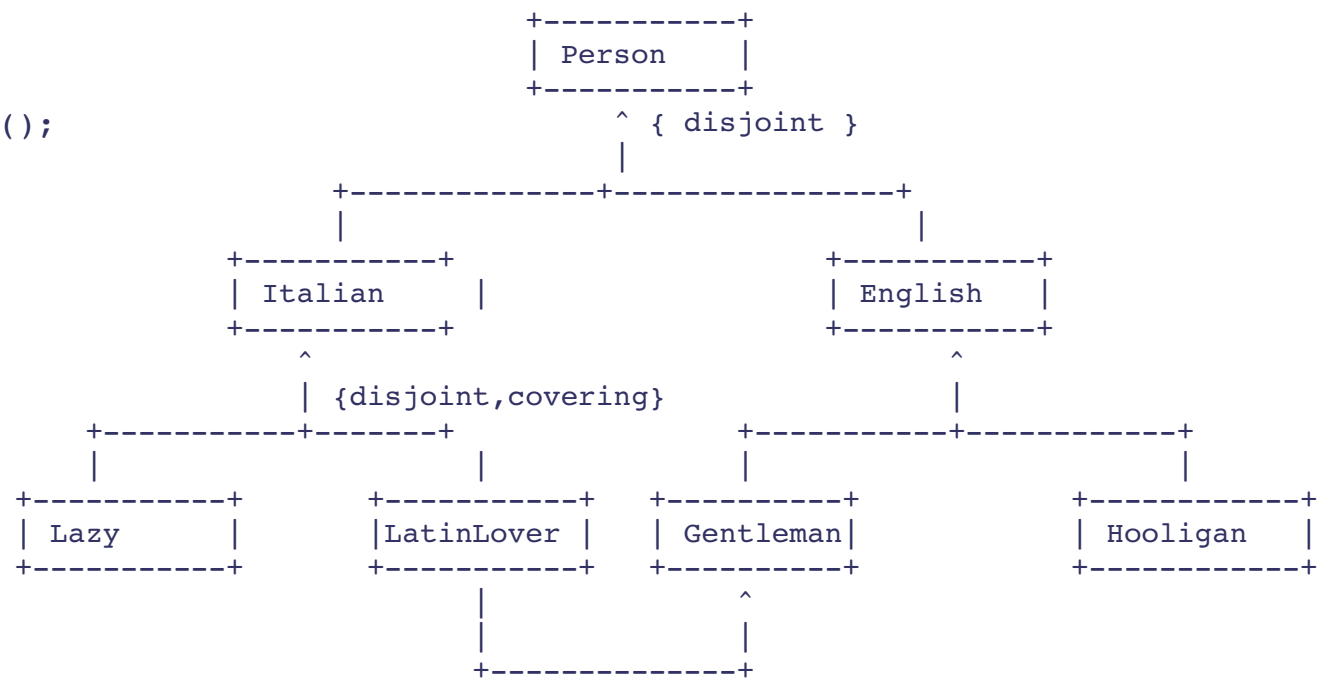
```
<generalization
complete="true" disjoint="true"
identifierName="AnimalsDiagram__Pet__generalize
    s__Dog_Cat_Parrot_Rabbit"
id="DCC_Element_10">
 <name></name>
 <subClasses>
  <subClass>AnimalsDiagram_Dog</subClass>
  <subClass>AnimalsDiagram_Cat</subClass>
  <subClass>AnimalsDiagram_Parrot</subClass>
  <subClass>AnimalsDiagram_Rabbit</subClass>
 </subClasses>
 <baseClass>AnimalsDiagram_Pet</baseClass>
</generalization>
```

- Output testuale

```
%%% [Generalization]: Dog + Cat + ... < Pet
%%%
all X ( AnimalsDiagram_Dog(X) ->
    AnimalsDiagram_Pet(X) ).
all X ( AnimalsDiagram_Cat(X) ->
    AnimalsDiagram_Pet(X) ).
all X ( AnimalsDiagram_Parrot(X) ->
    AnimalsDiagram_Pet(X) ).
all X ( AnimalsDiagram_Rabbit(X) ->
    AnimalsDiagram_Pet(X) ).
%%%     >       Disjointness
all X ( AnimalsDiagram_Dog(X) -> -
    AnimalsDiagram_Cat(X) ).
all X ( AnimalsDiagram_Dog(X) -> -
    AnimalsDiagram_Parrot(X) ).
all X ( AnimalsDiagram_Dog(X) -> -
    AnimalsDiagram_Rabbit(X) ).
all X ( AnimalsDiagram_Cat(X) -> -
    AnimalsDiagram_Parrot(X) ).
all X ( AnimalsDiagram_Cat(X) -> -
    AnimalsDiagram_Rabbit(X) ).
all X ( AnimalsDiagram_Parrot(X) -> -
    AnimalsDiagram_Rabbit(X) ).
%%%     >       Completeness
all X ( AnimalsDiagram_Pet(X) -> (
AnimalsDiagram_Dog(X) |
AnimalsDiagram_Cat(X) |
AnimalsDiagram_Parrot(X) |
AnimalsDiagram_Rabbit(X)
)
).
```


Un esempio: perché non può esistere un latin-lover (1 / 3)

```
CCDDiagram diagram =    // diagramma
    new CCDDiagram("EnglishAndItalians");
CCDClass_Factory  // factories
class_factory =
    new CCDClass_Factory();
CCDGeneralization_Factory
generalization_factory =
    new CCDGeneralization_Factory();
// classi
CCDClass person_class =
    class_factory.createInstance(
        diagram, "Person");
CCDClass italian_class =
    class_factory.createInstance(
        diagram, "Italian");
CCDClass english_class =
    class_factory.createInstance(
        diagram, "English");
... // altre classi
// generalizzazioni
Set<CCDClass> itaEnBrothers =
    new TreeSet<CCDClass>();
itaEnBrothers.add(italian_class);
itaEnBrothers.add(english_class);
generalization_factory.createInstance(
    diagram, true, false,
    person_class, itaEnBrothers);
... // altre generalizzazioni ...
generalization_factory.createInstance(
    diagram, gentleman_class,
    latinLover_class);
```



Un esempio: perché non può esistere un latin-lover (2 / 3)

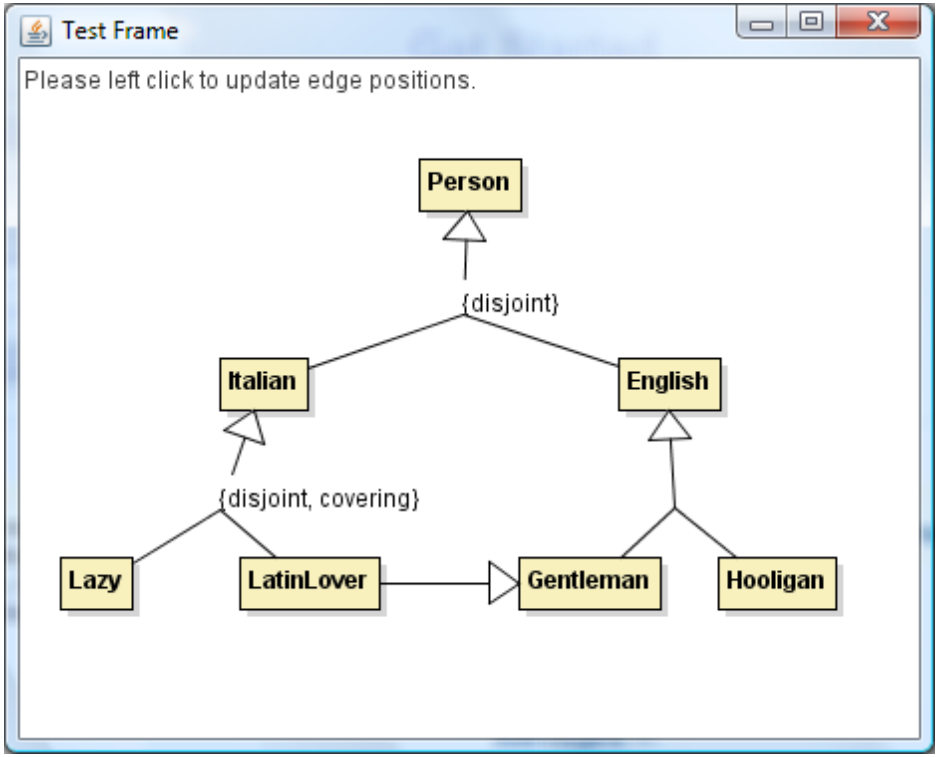
```
TreeSet<Drawable> drawableObjects =
new TreeSet<Drawable>();

Frame person_frame = new Frame(
new Point(200, 50));
Label person_label = new Label("Person");
person_label.setFont(RenderingPane.BOLD_FONT);
person_label.setTextAlignment(
Label.ALIGNMENT_CENTER);
person_frame.getFigure().add(student_label);
drawableObjects.add(person_frame);

Frame italian_frame = new Frame(
new Point(100, 150));
... // altri frame

ArrayList person_subClasses = new ArrayList();
person_subClasses.add(italian_frame);
person_subClasses.add(english_frame);
... // altre liste di sottoclassi

Edge person_hierarchy = new Edge(
person_subClasses, person_frame);
person_hierarchy.setStyle(Edge.STYLE_ISA);
person_hierarchy.setLabelName("{disjoint}");
drawableObjects.add(person_hierarchy);
... // altre gerarchie
```



Un esempio: perché non può esistere un latin-lover (3 / 3)

```
// prepare the diagram to test
CCDDiagram testDiagram =
    prepareEnglishItalianTestDiagram();
// marshal the given diagram in XML
String xmlString = new
    XmlMarshaller().marshal(testDiagram);
/* the formula to prove about the given diagram:
 * no latin lover can exist! */
String testGoal =
    "all X (-EnglishAndItalians_LatinLover(X)).";
// proving
ProvingResult proof =
new CCDProver9Prover(testDiagram).prove(testGoal);
// from now on, print the results
System.out.println(xmlString);
// ...
System.out.println(testGoal);
// ...
System.out.println(proof.proven);
// ...
System.out.println(proof.proof);
```

```
cdc@ubuntu.cdc.dis.uniroma1.it: ~/workspace-MFIS/ReDiA-VeriFInt
File  Edit  View  Terminal  Tabs  Help
============================== prooftrans ============================
============================== Prover9 ===============================
Prover9 (32) version Dec-2007, Dec 2007.
Process 7595 was started by cdc on ubuntu.cdc.dis.uniroma1.it,
Tue Sep 30 11:59:25 2008
The command was "exe/./prover9".
============================== end of head ===========================

============================== end of input ==========================

============================== PROOF =================================

% -------- Comments from original proof --------
% Proof 1 at 0.00 (+ 0.00) seconds.
% Length of proof is 15.
% Level of proof is 4.
% Maximum clause weight is 2.
% Given clauses 0.

1 (all X (EnglishAndItalians_Gentleman(X) -> EnglishAndItalians_English(X))).  [
assumption].
3 (all X (EnglishAndItalians_LatinLover(X) -> EnglishAndItalians_Gentleman(X))).
  [assumption].
6 (all X (EnglishAndItalians_Italian(X) -> -EnglishAndItalians_English(X))).  [a
ssumption].
8 (all X (EnglishAndItalians_LatinLover(X) -> EnglishAndItalians_Italian(X))).
[assumption].
12 (all X -EnglishAndItalians_LatinLover(X)).  [goal].
13 -EnglishAndItalians_LatinLover(x) | EnglishAndItalians_Gentleman(x).  [clausi
fy(3)].
14 -EnglishAndItalians_Gentleman(x) | EnglishAndItalians_English(x).  [clausify(
1)].
20 -EnglishAndItalians_Italian(x) | -EnglishAndItalians_English(x).  [clausify(6
)].
21 -EnglishAndItalians_LatinLover(x) | EnglishAndItalians_Italian(x).  [clausify
(8)].
28 -EnglishAndItalians_LatinLover(x) | EnglishAndItalians_English(x).  [resolve(
13,b,14,a)].
29 EnglishAndItalians_LatinLover(c1).  [deny(12)].
31 -EnglishAndItalians_LatinLover(x) | -EnglishAndItalians_English(x).  [resolve
(21,b,20,a)].
34 EnglishAndItalians_English(c1).  [resolve(28,a,29,a)].
36 -EnglishAndItalians_English(c1).  [resolve(31,a,29,a)].
37 $F.  [copy(36),unit_del(a,34)].

============================== end of proof ==========================
```

Futuri sviluppi

- **Estensioni**
  1. Supporto alla definizione di **vincoli** esterni
     - in **OCL** (**O**bject **C**onstraint **L**anguage)
  2. Definizione e gestione dei **tipi** definiti dall'utente
  3. Supporto per la traduzione del diagramma in input per **altri dimostratori automatici**
  4. **Attivazione automatica del dimostratore**
     - per verificare tesi *ad hoc* al momento dell'aggiunta/modifica di componenti del diagramma
- **Perfezionamenti**
  1. Completamento dell'**interfaccia grafica** e delle classi di gestione
  2. Connessione dei componenti dell'***engine* grafico** al diagramma memorizzato
     - primitive più ad alto livello
  3. Creazione di un'**interfaccia utente** grafica
  4. Conclusione delle regole di trasposizione in formule della logica del prim'ordine dei componenti del diagramma delle classi