AN ERD TOOL

Ron McFadyen
Applied Computer Science
University of Winnipeg
Winnipeg, Manitoba, Canada R3B 2E9
r.mcfadyen@uwinnipeg.ca

Abstract

This paper discusses a graphical ERD editor that was developed using Eclipse, the Eclipse Modeling Framework, the Graphical Editing Framework, and the Eclipse Rich Client Platform. We discuss the need for the editor, aspects of its development within Eclipse, and its use by students. The ERD editor has been used in a 2000-level undergraduate course that covers database design and some features of MS Access, and will be used in further courses at our University. Because of the architecture that the Graphical Editing Framework provides the tool has also been used in a software architecture course to provide examples of software design patterns.

1 Introduction

At our university entity-relationship (ER) modeling is a topic in several database courses. The text [1] used in our 3000- and 4000-level courses is a standard university text where the notation for ER Modeling is based on the work of Peter Chen [2]. The major artifact produced in the modeling process is an entity-relationship drawing (ERD); a sample ERD is given in Figure 1.

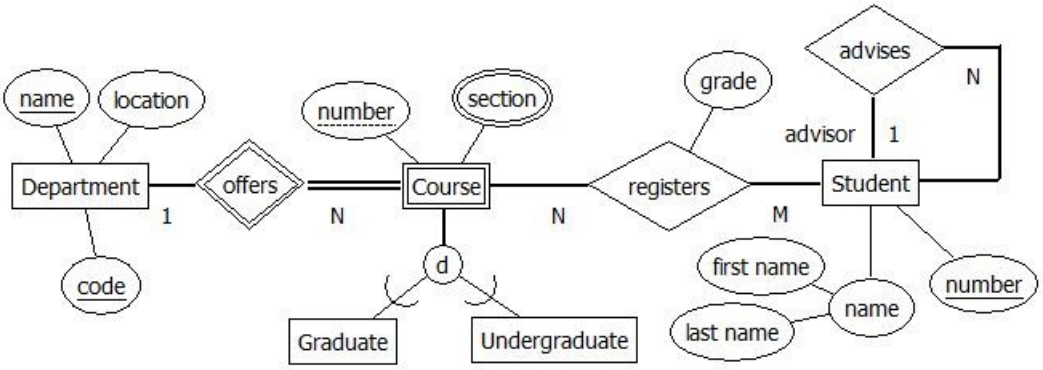


Figure 1: A sample ERD.

We are not aware of a software tool for creating ERDs that provides the exact notation as presented in the text [1]. Our experience has been that different instructors take different approaches for when students work on ERDs: paper and pen, a generic drawing tool such as MS PowerPoint, commercially available tools such as the Erwin Data Modeler [3] and Visual Paradigm [4]. Paper and pen, and generic drawing tools do reflect how models may have their beginnings in practice (such as on the back of the envelope) but there is no feedback or enforcement of notation. Tools such as Erwin and Visual Paradigm were not designed for academic use. When starting one of these tools the beginner is presented with a graphical user interface that is complex and powerful (Figure 2), whereas an academically oriented tool can have a simplified and focussed interface (Figure 3).

At some point introducing students to commercial tools is worth while but at the beginning a student is best served with a focussed drawing tool that reflects the textbook presentation. In [5] it is reported that students preferred to use an academically oriented ERD tool called TerraER over a more complex tool named DBDesigner. Our tool has similarities to TerraER but provides the student with a different look and feel and additionally provides for composite attributes, recursive relationships and an undo/redo capability.

In section 2 we briefly discuss our requirements and in sections 3, 4, and 5 we discuss aspects of the development environment. We do not go into much detail as there are several publications that describe Eclipse and various frameworks [7, 8, 9, 10]. For us the most significant resource has been a tutorial series by Arieh Bibliowicz [12] that takes one through the development process step-by-step. In section 6 we present more examples of usage and in section 7 we give our conclusions and next steps.

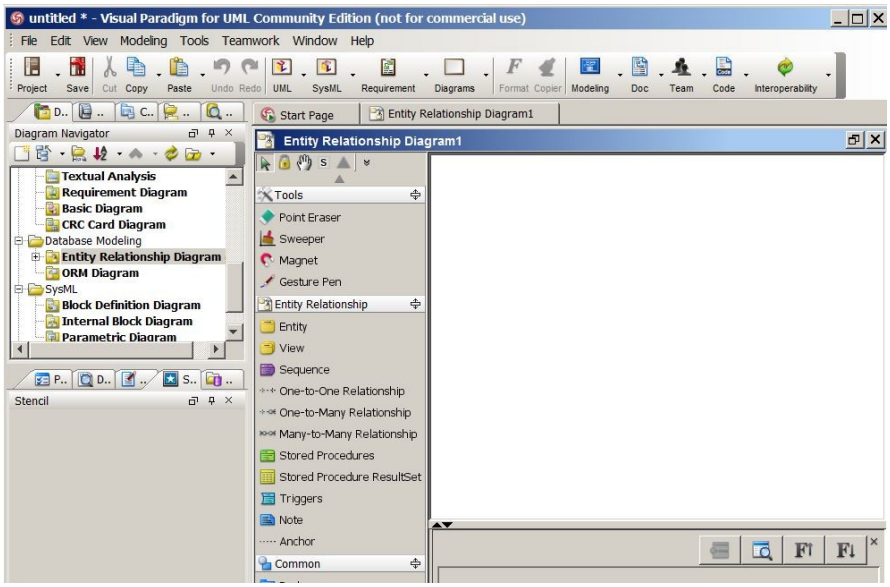


Figure 2: User interface for Visual Paradigm Community Edition.

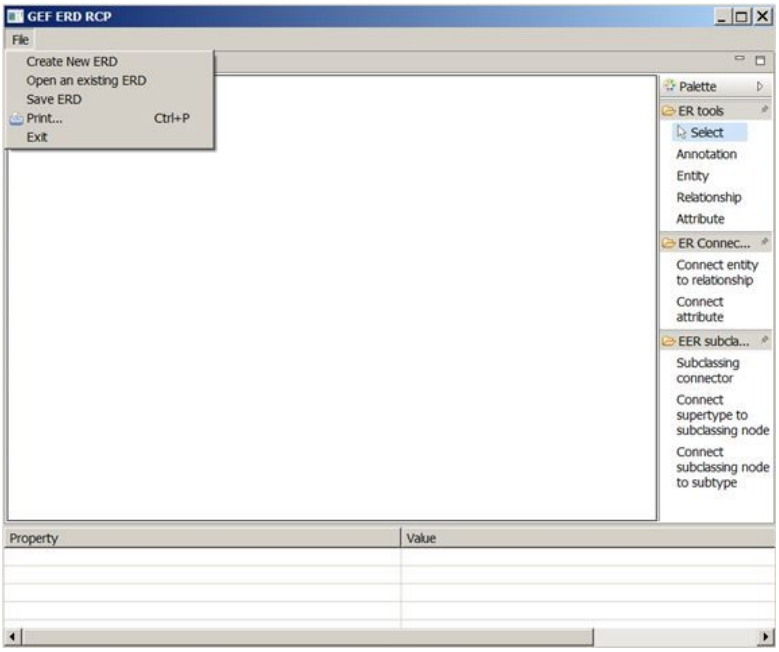


Figure 3: Simple UI for ERD tool.

2 Requirements

Our requirement is to draw ERDs with a tool that is easy to use and readily available to our students. If one reviews entity relationship modeling in the text [1] a number of basic concepts must be included. The symbols used to represent these concepts in the Chen notation are shown in Figure 4; many of these concepts/symbols are used in Figure 1. A brief discussion of these concepts follows.

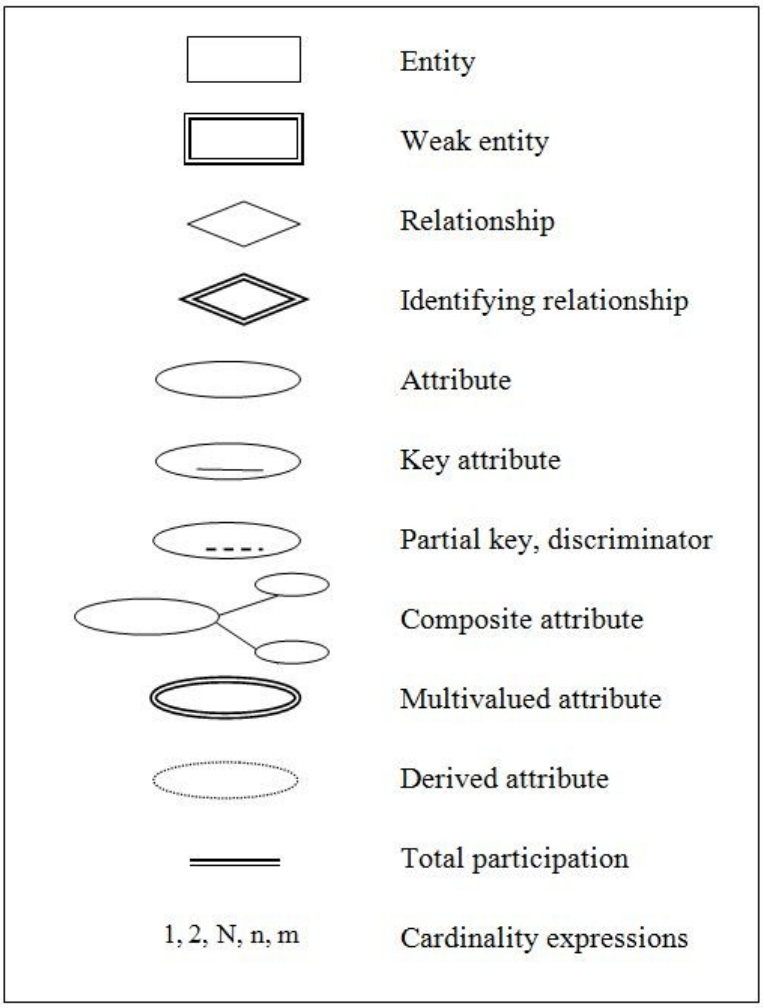


Figure 4: Symbols used in the Chen notation.

Entity types: it must be possible to draw the rectangular shape of an entity type and give it a name. Entity types represent things, persons, places in a design.

Weak entity types: For some entity types their existence may depend on some other entity. For example, a course entity may only exist if a corresponding department entity also exists. Such entity types are called *weak* entity types and are depicted with a double-lined rectangle.

Relationship types: it must be possible to draw the diamond shape for a relationship type and enter a phrase indicating how two or more entity types are related. Additionally relationship types are connected via lines to the pertinent entity types (some entity type may play more than one role).

Recursive relationships: Relationship types that involve the same entity type more than once are called recursive. In our experience recursive relationships are challenging for students of database design. For example, a student *advises* a student is a recursive relationship involving the student entity type twice.

Identifying: An identifying relationship is one where a primary key of one entity type is used to identify another entity type. A typical example arises when a course number on its own is not enough to identify a course – in addition an identifier for the department is necessary. For example, different departments have unique department codes, but different departments may have courses of the same number: ACS-2814 and MATH-2814.

Cardinality: Each relationship line has a cardinality that reflects the number of instances of an entity type that can be related to another entity instance via the relationship. There are two choices, one or many; The cardinality *one* is shown with a *1* and a the cardinality *many* is shown with a letter such as *N* or *M*. For example, a department may offer *many* courses, but a course is offered by only *one* department.

Role names: To enhance understanding of an entity type's role in a relationship the entity type can be given a role name. For instance a student can be referred to as an *advisor*.

Participation: For each entity type involved in a relationship one must specify if an instance must participate or not (total vs partial participation). Partial participation is shown with a single line and total participation is shown with a double line. For example, a course *must participate* in relationship with a department.

Attributes: It must be possible to draw an oval with a name to represent an attribute. Attributes can be further indicated as:

Composite – attributes can be organized hierarchically (a student's name *comprises* a first name and last name).

Derived – must be shown with a dotted-lined oval.

Multi-valued – must be shown with a double-lined oval (a course may be delivered in *multiple* sections such as section 001, section 002, section 003).

Key - shown with a solid underline (a department is known by its name or by a code).

Partial key – shown with a dashed underline.

Supertypes: We must be able to show an entity type as a supertype of one or more entity types (its subtypes). Subtypes with respect to a supertype can be overlapping or disjoint, and we also need to be able to show a discriminating attribute for subtypes. For example courses are either graduate courses or undergraduate courses.

3 The Eclipse Modeling Framework (EMF)

EMF has been used to develop a data model for the ERD editor and to generate Java code based on that model. Figure 5 shows some basic features of the meta-model: that it must contain information about entity types, relationship types, and attributes, and that binary links connect these features (for example: an attribute may be composed of other attributes, an entity type may be connected to a relationship type, an attribute can be connected to an entity type or a relationship type).

The code generation facility of EMF was used to generate Java code for the model. The generated code includes a basic editor, a factory with methods for creating objects, interfaces for each model component, and concrete implementations corresponding to each interface.

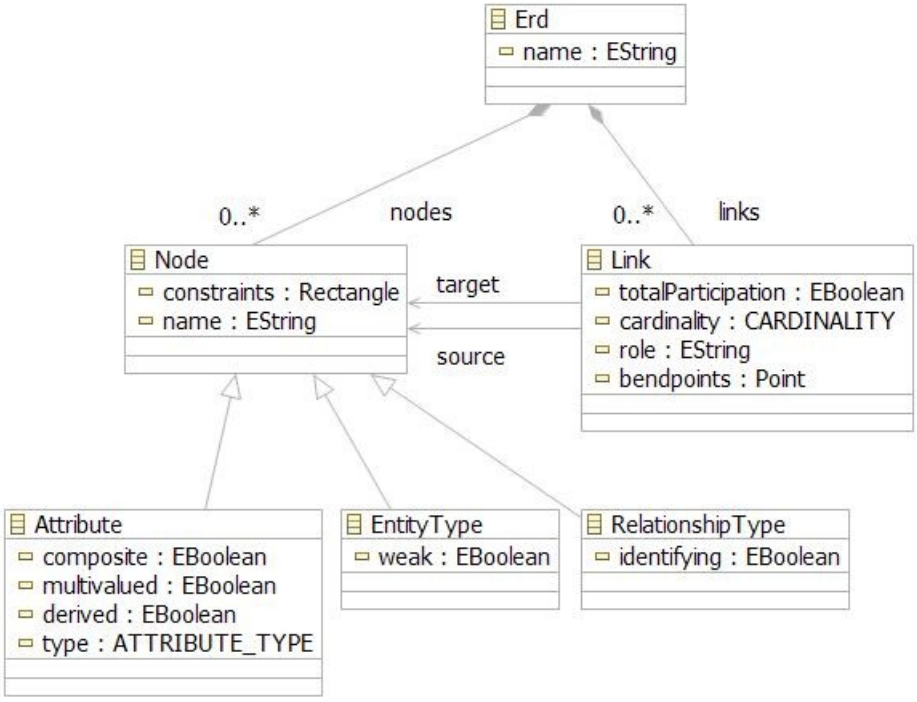


Figure 5: Part of a meta-model for an ERD editor.

4 The Graphical Editing Framework (GEF)

GEF provides the basic framework for the graphical editor. Typically (and in our case) the developer begins by extending a base class: the *graphical editor with flyout palette*. This class has various responsibilities such as configuring the palette, and creating, saving, and opening a model.

The developer must create the various tools comprising the palette. For our editor this amounted to deciding on the basic object creation facilities such as creating an annotation, creating an entity type, creating a relationship type, creating an attribute, connecting an entity type to a relationship type, etc. (see Figure 6).

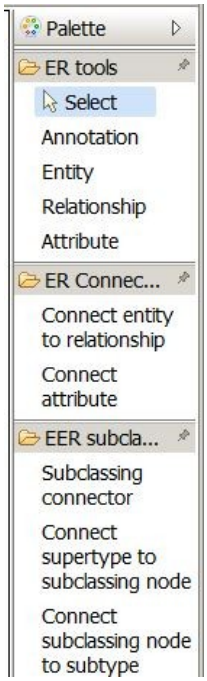


Figure 6: Palette tools for creating model components.

GEF is based on the Model-View-Controller (MVC) design pattern [11] (see Figure 7). In a GEF-based application there are many pieces to MVC the developer must provide which, of course, are broken down into 3 broad areas: model, view, and controller. The model contains the data, the view comprises the figures the user sees on the canvas, and the controller manages any interaction between model and view. For instance, if the user changes the drawing the controller will recognize the change and inform the model of such changes so the model will update itself. If some property in the model changes then the controller will learn of this and send requests to the view (so it will update itself). The GEF framework provides for notifications based on the Observer design pattern [11].

The Controller is multi-part. For each model element, including the overall diagram being created, there must be a corresponding 'Edit Part' which functions as the specific controller for the model element and its view components. GEF provides a 'Root Edit Part" that controls the overall interaction: for instance, when a canvas event occurs, the Root Edit Part determines which specific Edit Part should handle the event.

For the ER drawing tool, the Model is a meta-model that describes the information represented in an ERD. The View comprises a number of figures, one per model element. The developer decides how each model element is to be viewed on the canvas and (roughly speaking) codes one class per figure. For example, as we require an Attribute to be seen as an oval we have the constructor shown in Figure 8. Then depending on the characteristics of the Attribute the user may alter some properties (for a multi-valued attribute the second ellipse becomes visible and inset; for a derived attribute the outline will be a dotted line).

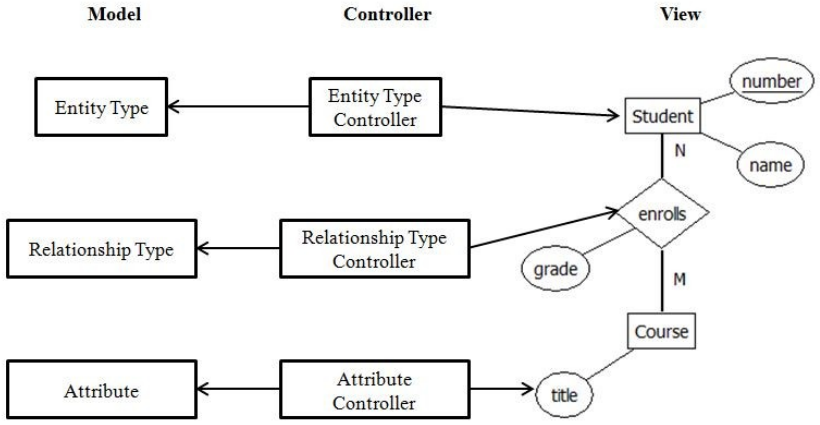


Figure 7: GEF is based on the Model-View-Controller design pattern.

```
public AttributeFigure() {
    setLayoutManager(new XYLayout());
    ellipse = new Ellipse();
    e2 = new Ellipse();
    underline = new Polyline();
    add(ellipse, new Rectangle(0, 0, -1, -1));
    add(e2, new Rectangle(0, 0, -1, -1));
    e2.setVisible(false);
    nameLabel = new Label();
    nameLabel.setTextAlignment(PositionConstants.CENTER
    );
    add(nameLabel, new Rectangle(0, 0, -1, 20));
    add(underline);
    underline.setVisible(false);
```

Figure 8: Constructor for an Attribute Figure.

Each Edit Part specifies editing policies that determine how various actions are carried out. For instance, if an ERD component is being moved on the canvas a command object (see Figure 9) is created to cause the movement which the GEF framework will manage in accordance with the Command design pattern [11] (and hence provide for undo and redo capability). Editing policies control the connections allowed between two objects (for instance, starting from an entity type our editor allows one to draw a connection line to a relationship type, but starting at a relationship type one cannot draw any connection line).

```
protected Command createChangeConstraintCommand(
                  EditPart child, Object constraint)
{
     ErdNodeChangeConstraintCommand command = new
            ErdNodeChangeConstraintCommand();
     command.setNode((ErdNode) child.getModel());
     command.setNewConstraint((Rectangle)
     constraint);
```

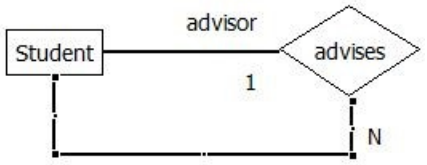
Figure 9: Command object created for changing location.

GEF provides basic figures such as rectangles, ellipses, labels and lines. In some cases more elaborate visualizations are required. For example, *polylines* can be used to create the Chen diamond figure used for a relationship type (see Figure 10), and figures can be composed of other figures using various layout strategies.

```
public RtypeFigure() {
    // create a diamond shape for a relationship type
    setLayoutManager(new XYLayout());
    Rectangle r = new Rectangle(10, 10, 50, 50);
    ps = new ScalablePolygonShape();
    ps.setStart(r.getTop());
    ps.addPoint(r.getTop());
    ps.addPoint(r.getLeft());
    ps.addPoint(r.getBottom());
    ps.addPoint(r.getRight());
    ps.addPoint(r.getTop());
    ps.setEnd(r.getTop());
    ps.setPreferredSize(r.getSize());
    ps.setForegroundColor(ColorConstants.black);
    ps.setLineStyle(SWT.LINE_SOLID);
    ps.setLineWidth(1);
    add(ps);
```

Figure 10: Creating a custom figure.

GEF provides a feature called bendpoints that can be set for a connection line. The bendpoints are a sequence of points used to form connected line segments (Figure 11). This is useful for recursive relationships to allow lines to be customized by the user. By default the two lines connecting the same entity type and relationship type are one on top of the other. By selecting one, clicking a midpoint and dragging, one of the lines is easily reshaped. Using an Eclipse feature called property sheets the ERD drawing tool shows properties of components on the canvas. Using the property sheet view the drawing tool easily displays the properties of the selected component for the user to view or change. Some properties, such as the (x, y) location of a figure and the (x, y) locations of bendpoints, were hidden from the user so they could not be manipulated directly (Figure 11).



Property	Value
Cardinality	N
Role	
Total Participation	false

Figure 11: Viewing properties of a line having two bendpoints and three segments.

When figures are joined via lines as in an ERD, the drawing tool must determine where (called an anchor point) a connection line contacts the figure. In the case of a diamond shaped figure one would like the line to make a connection at a point on the line that outlines the diamond (as opposed to a default anchor position on a bounding rectangle for the diamond shape). This requires a custom anchor for the figure that determines the point of contact. With respect to Figure 11 the relationship lines connect to points on the lines forming the diamond shape. In Figure 1 there is an attribute *grade* connected to a different location on the diamond shape.

5 Creating a Rich Client Platform (RCP)

When one creates a graphical editor it is created, tested, and run in the Eclipse IDE environment which is very complex. To simplify the student's experience a standalone application was generated that greatly simplifies the use of the ERD drawing tool. At the heart of the generation process is a product configuration file and an Eclipse Product Export Wizard. In our case the result is a zip file which, when expanded, contains all necessary modules and a Java runtime environment – the user runs the *erd.exe* file shown in Figure 12 and the application starts as depicted in Figure 13.

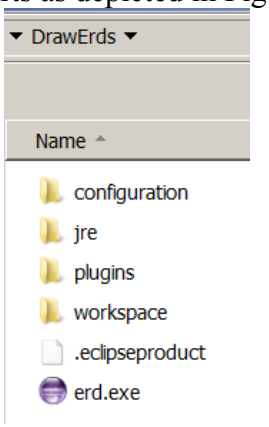


Figure 12: The ERD drawing tool with its own runtime environment.

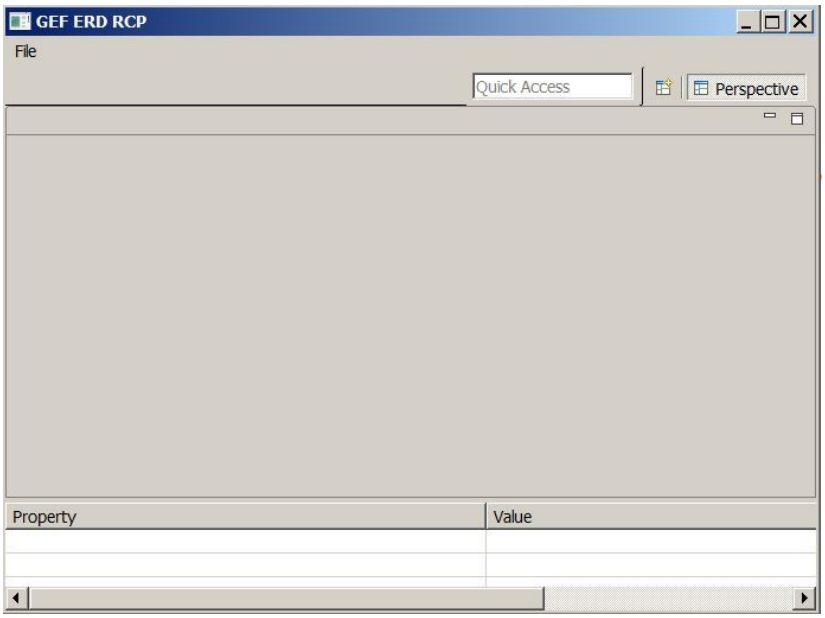


Figure 13: The ERD drawing tool on startup.

6 Further Examples of Usage

In our student laboratory environment the tool is a desktop icon that is also available via a remote desktop facility. When the tool starts the user is presented an interface with limited functionality where the user can create a new ERD, open an existing ERD, etc. as seen in Figure 14.

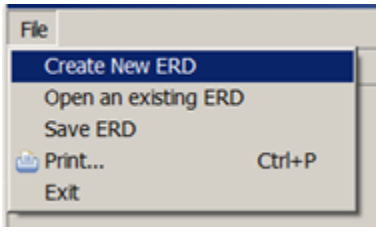


Figure 14: One menu to choose from.

When the user has chosen to create a new ERD the user is presented with a blank ERD canvas on which to draw an ERD. When the palette is showing, the toolbox is obviously one for drawing ERDs (Figure 15). By clicking on a tool, say *Relationship*, and then clicking on the canvas the user enters the text for that figure directly as shown in Figure 16.

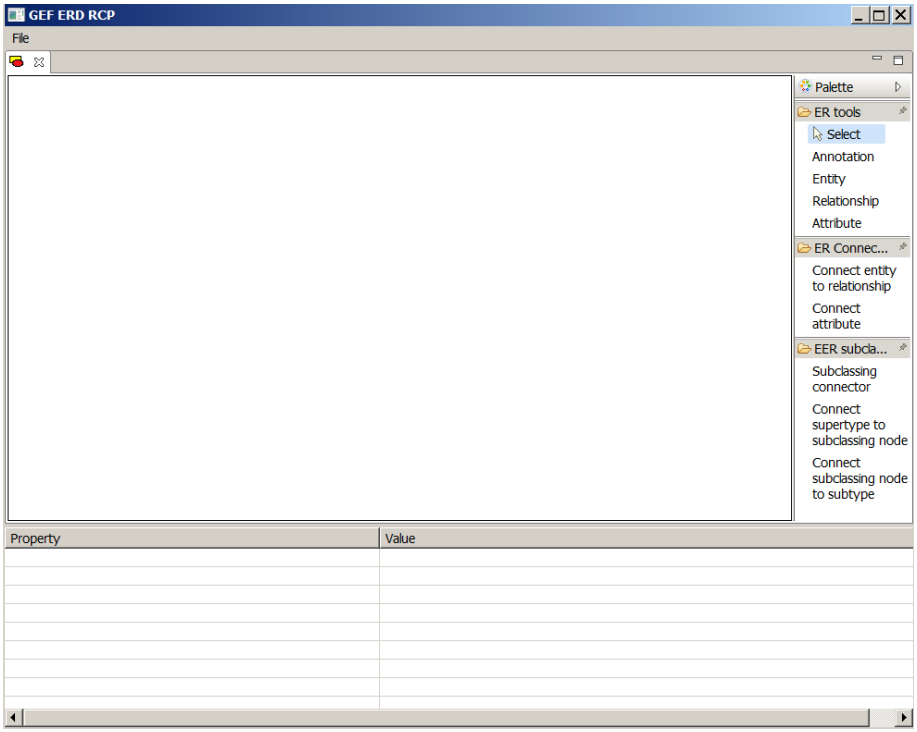


Figure 15: A simple palette of drawing choices.

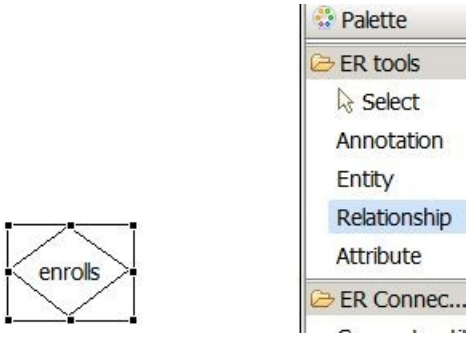


Figure 16: Creating a figure and entering text in place.

When a figure is selected on the canvas the relevant properties for that figure are displayed in a property sheet view below the diagram. For example in Figure 17 the selected figure represents an attribute and the user can change its properties: its name, whether its derived, multi-valued, and whether it is descriptive, a key, or a discriminator.

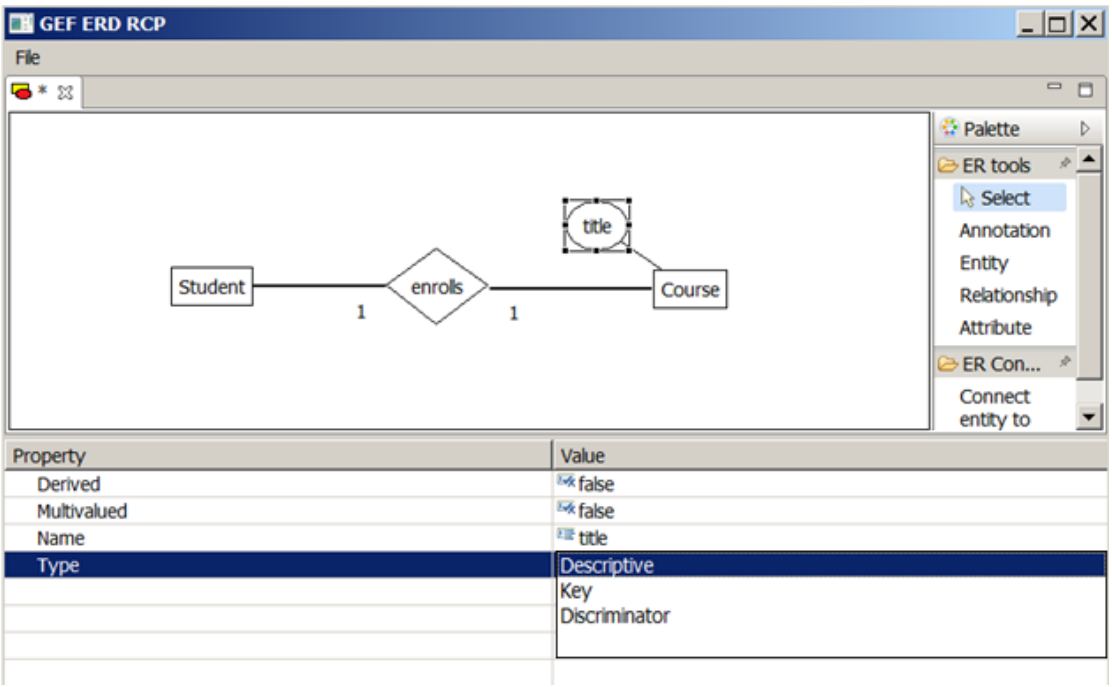


Figure 17: The property sheet view shows properties of the selected component.

7 Conclusions and Future Work

The ERD drawing tool provides a simple interface that is easily understood by students of database design. Developing the tool was the author's first experience with Eclipse, EMF, GEF, and RCP, and so it is expected that several refinements can be made. We feel the ERD Tool provides the basics for creating entity relationship diagrams, but there is much more functionality that can be provided which will form the basis for development going forward. The functionality to be incorporated includes:

Creating a relational view.

An important topic is the transformation (mapping) of an ERD to a relational database. The first step when creating a database is to produce a design, the ERD. The second step is to create a physical database. The recommended relational database can be included as an additional view.

Snap-to geometry.

This feature is found in many drawing programs to facilitate the alignment of figures relative to other figures or to a grid superimposed on the canvas.

ERD validation.

The ERD tool does enforce some ERD drawing rules; for example, one cannot make a connection between two entity types. However, the tool does allow the student to make certain other mistakes; for example, connecting only one entity type to a relationship type when it is required that relationships be n-ary where $n>1$. An option can be developed to check for the correctness of the drawing and give feedback to the student.

8 References

[1] Ramez Elmasri, Shamkant B. Navathe; *Fundamentals of Database Systems*; 2010; Pearson Education; ISBN - 13: 9780136086208.
[2] Peter Chen; *The entity-relationship model – toward a unified view of data*; ACM Transactions on Database Systems; Vol. 1, No. 1, 1976.
[3] Erwin Data Modeler; http://erwin.com/products/data-modeler
[4] Visual Paradigm; http://www.visual-paradigm.com/
[5] Henrique Rocha, Ricardo Terra; *TerraER – an Academic Tool for ER Modeling*; Methods and Tools, Vol. 21, No. 3; pages 38-41, 2013.
[6] DBDesigner; http://www.fabforce.net/dbdesigner4/
[7] Eric Clayberg, Dan Rubel; *Eclipse Plugins*; 3rd edition, Addison-Wesley Professional; 2008; ISBN-13: 978-0321553461.
[8] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks; *EMF Eclipse Modeling Framework*; Addison-Wesley Professional 2nd edition; Dec 16 2008; ISBN-13: 978-0321331885.
[9] Lars Vogel; *Eclipse 4 RCP: The complete guide to Eclipse application development*; ISBN-13: 978-3943747072.
[10] Dan Rubel, Jaime Wren, Eric Clayberg; *The Eclipse Graphical Editing Framework (GEF)*; Addison-Wesley Professional; 2011; ISBN-13: 978-0321718389.
[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Professional; 1994; ISBN-13:9780201633610
[12] Arieh Bibliowicz; *GEF Tutorials*; http://www.vainolo.com/tutorials/gef-tutorials/