Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM

Steven Kelly
stevek@metacase.com

1 Introduction

Over the last year or two, interest in Domain-Specific Modeling has grown tremendously: a few examples will suffice. Bill Gates has said visual modeling will be the most significant innovation in the next 10 years, reducing coding 'by a factor of five' (http://www.adtmag.com/article.asp?id=9166). Microsoft has unveiled the Whitehorse domain-specific modeling editors for Visual Studio, with an SDK to come that will allow developers to build their own DSM editors. IBM has released the Eclipse Modeling Framework and Graphical Editor Framework, offering another way to build your own DSM editor. Software Development Magazine chose MetaCase's MetaEdit+ DSM environment as a finalist in this year's Jolt software productivity awards.

The Eclipse frameworks and MetaEdit+ offer significantly different approaches to defining DSM support. In this paper, we examine both approaches and see how they shape up as tools for DSM developers. First however we must explain what DSM is; for an overview of how it relates to other similar topics, see the glossary.

DSM is about generating full code directly from models, making software development 5-10 times faster. The only way of doing so that has worked in practice is to make both the modeling language and generators domain-specific. Attempts to make a completely generic modeling language and generators have failed, as might be expected. Raising the level of abstraction always means sacrificing fine control and complete generality, for the more important end of productivity. After all, nobody expects control over the machine code run by a given Java statement, nor is J2EE a good environment for 3D graphics engines.

Different camps have different views on just how specific the DSM languages should be. At one end of the scale stands the OMG, who would have liked to see everybody use unadulterated UML. The OMG have now tacitly admitted that full code generation from UML is not going to happen, and they are pinning their hopes on MDA (model-driven architecture). At its most basic, this involves transforming one UML model into another UML model, possibly several times and possibly automatically, then automatically generating substantial code from the final model.

A manufacturer-sponsored study of UML-based MDA showed productivity increases of 35% (http://www.compuware.co.uk/pressroom/news/21072003_02.htm). While good, that is far from the 500%-1000% consistently found with DSM. MDA proponents envisage higher forms of MDA incorporating elements of DSM, and these may offer some more hope. In these, the base UML can be extended with domain-specific enhancements, or even replaced with new MOF-based metamodels. However, experiences with the former have found current tools lacking the necessary extensibility, and no tools support the latter.

Simply put, UML is not domain-specific, and UML tools were not designed to support changing UML. Trying to build domain-specific models in a UML tool is – at best – like trying to write English in a Spanish version of Word. The GUI labels are all wrong, and the tool keeps trying to correct your input into something valid for its

language. Worse, since the tool is parsing your input according to the wrong language, any attempt at translation or code generation will be fraught with difficulties.

This in part explains the lack of adoption of DSM, despite all its promise: existing CASE tools simply cannot support it. Without tool support, any modeling language is largely useless, and certainly no code can be generated. Building a CASE tool for your own modeling language is prohibitively expensive. Even for a simple language, full CASE support would take man-years to build from scratch.

That is where tools for building DSM editors come in. These tools allow you to build a completely new modeling language, editor, and code generator for that domain. The tools can be divided into two classes: code frameworks such as Eclipse's EMF and GEF, and metaCASE tools such as MetaEdit+. The frameworks are just that: pure code, with utility functions and classes useful for building graphical CASE tools. In contrast, metaCASE tools already implement generic graphical CASE behavior, and the user supplies the concepts and symbols via the tool's GUI.

2 Eclipse

Since IBM released its Eclipse IDE and tools as open source, the Java framework has gathered developers at an impressive rate. Although the Eclipse 'ecosystem' of plugin developers is in no small part funded directly by IBM grants, the spirit is perhaps more one of enlightened philanthropy than crass commercialism. While Sun is still trying to decide how best to relate to this new community, Eclipse is already making fruitful contributions to academic research. Measuring the commercial impact of something that is free is always difficult, but 18 million downloads is a big number in anyone's book.

While the main focus of Eclipse has been on its IDE for Java programmers, two major Eclipse tool projects offer help for modelers too. The Eclipse Modeling Framework (EMF) allows you to input your desired data model, and can generate simple table-based editors and an XMI schema for such models. The Graphical Editor Framework (GEF) supplies functions and classes useful for specifying graphical editors for Eclipse data. Although you can use EMF and GEF separately, building DSM support requires both – and of course Eclipse. There is no support for building stand-alone editors. We looked at the latest Eclipse 3.0 release candidate, RC1, and the corresponding EMF and GEF versions from www.eclipse.org. As our experience on three CASE tool construction projects has taught us that the graphics are always more complicated than the model data, we will spend more time on GEF.

2.1 EMF

The main function of EMF is to provide a data entry and storage environment following a schema that you supply. EMF does not use the OMG MOF standard, although the design of the Ecore data model it uses was influenced by MOF. In the EMF team's experience of building modeling tools MOF was found to lack necessary features. This mirrors our own experience of MOF: to pick just one example, the lack of support for n-ary relationships is an astonishing oversight. The most probable explanation for that, and most likely a large proportion of the problems, is MOF's heritage. MOF was created simply to be able to model UML in UML, and as such was simply a UML subset. Whilst later versions have tried to give it an existence of its own, it remains clearly tied to its parent. Other groups that have tried to use MOF as a meta-metamodel seem to have faced the same problem, invariably coming up with their own extensions to MOF. EMF's Ecore is thus no exception.

Your schema, or metamodel, can be fed to EMF in several formats. The native format is an XMI file, but EMF can also read Rational Rose class models, annotated Java files, or XSD files, providing these follow its restrictions.

Based on this input, EMF.Codegen can generate a bare bones editor for data following the schema. The editor uses classes from the EMF.Edit framework to provide standard table and property sheet views. If the generated editor is not sufficient, you can add your own code. Providing it is marked correctly, the generator will not overwrite your code when the schema is updated – although of course it will not update it to reflect the schema changes either.

You can also build your own code directly on top of the EMF.Edit framework, and that indeed seems to be the way many developers go. An interesting Norwegian project, part of www.pats.no, has been working on cell phone service engineering. Their original solutions used a State Machine pattern, with both state actions and legal transitions hand-coded in Java. To reduce the work and improve the chances of validating the resulting system, they wanted to have the state machine represented in a model.

For the simpler parts of the system, they were able to use the EMF code generation framework on annotated Java and even an existing XSD file. For the more table-based state editor, they are hand-coding on top of the EMF.Edit framework. Currently that part is 4000 lines of code, and requires about another three months' work. The rest of the editors and property sheets are another few thousand lines of code, with significant parts generated. In total the project has taken about six man-months so far.

2.2 GEF

EMF only provides part of the solution for DSM: data storage, property sheets, tree or table-based browsing, and a code generation framework. GEF provides the graphical support needed for building a diagram editor on top of the EMF framework. Diagrams bring two vital additions to the modeling experience. Firstly, the human brain is much better at quickly interpreting and remembering a graphical diagram than text, trees or tables. Secondly, diagrams show multiple relationships between objects much better than text or table formats. Whilst a tree format can show one simple kind of relationship well, it cannot handle object reuse or other relationships.

Strangely, GEF is not particularly designed to take advantage of EMF. The only thing they really share is their integration with Eclipse change notification. GEF uses a Model-View-Controller pattern, where the Model can be an EMF model, or something entirely different. Whilst using GEF, the intention is thus that the Model is largely ignored, and the main work happens in the Controller. Such a heavy Controller in an MVC framework is somewhat unusual. In many ways it might have been better to establish an MVC framework entirely within GEF, with its own true Model to represent facts like the co-ordinates of a model element. This graphical Model could then have received change notifications from the EMF model.

The GEF Controller is called an EditPart, and for each EMF model element class you will normally need to create a corresponding EditPart class. EditParts have a Figure, which is their graphical view, implemented in the lower-level Draw2D graphical framework. Designing a symbol for your DSM language thus consists of writing Java code for its individual lines and curves, which can be painstaking work. Often a second Figure will be necessary for display when a model element is being moved, e.g. to show the symbol grayed out.

EditParts respond to events by way of an EditPolicy: most EditParts require their own EditPolicy class. The job of the EditPolicy is to turn the event request into a Command. GEF uses the Command pattern to implement an undo stack: all changes to data must happen through Commands, and each Command must store its own undo information on the stack, and implement an undo method. Unfortunately, whilst EMF also has the same pattern, they are implemented in different namespaces, and so cannot be used together. Instead, the developer must maintain EMF undo information separately from GEF undo information, and try to maintain consistency between them.

3 MetaEdit+

MetaEdit+ is the most widely-used commercial metaCASE tool, first released in 1995. The 4.0 SR1 version used here is available from www.metacase.com. MetaEdit+ was built on the principle that all CASE tools are essentially the same: you can put objects on a diagram, fill in their properties, connect them with relationships, and move them around. All that really changes between different modeling languages is what the object types look like, what properties they have, and how you can connect them.

The MetaEdit+ toolset thus includes generic CASE behavior for objects and relationships, including a Diagram Editor, Object and Graph Browsers, and property dialogs. The DSM developer need only specify his modeling language: e.g. creating a new object type, giving it a name and choosing which property types it has. A vector-based Symbol Editor allows you to define your object and relationship symbols, or reuse existing symbols. There is no need for any hand coding, nor is any CASE tool code generated. The MetaEdit+ editors simply follow the defined language in a similar way to how Word follows its templates.

In addition to the CASE editing functionality, MetaEdit+ also includes XML import and export, an API for data and control access to MetaEdit+ functions, and a generic code generator. The code generator uses a DSL that allows the DSM developer to specify how to walk through models and output their contents along with other text. This makes defining code generators simple, with one line of a code generator definition corresponding to several lines in the scripting languages sometimes used for this purpose. As the generator has no preconceptions about the modeling language, code language, or framework the code will run on top of, the DSM developer has complete freedom to produce the best code possible from the models.

4 Comparison

We had originally intended to implement a small DSM language from scratch in both Eclipse and MetaEdit+. However, the code for the simplest GEF-only editor weighs in at 130KB (65 files, nearly 5,000 lines) for three object types. At 50 lines of code a day, a simple COCOMO calculation gives an estimation of over six man-months to complete a similar project. As implementing a language from scratch was thus out of the question, we decided to give Eclipse a slight advantage by taking one of its samples and re-implementing it in MetaEdit+. An unexpected benefit of this is that the Eclipse version of the sample was thus built by Eclipse experts, and the MetaEdit+ version by a MetaEdit+ expert, leveling the playing field.

As the simplest sample was rather small for a DSM, and behaved somewhat oddly graphically, we took the second sample. This was a fairly simple Logic Gate language: you could connect together AND, OR and XOR gates and link them to form circuits with

LED displays and voltage sources (Figure 1). The example was again pure GEF, with no EMF for data storage or editing: there were essentially no data values to edit.

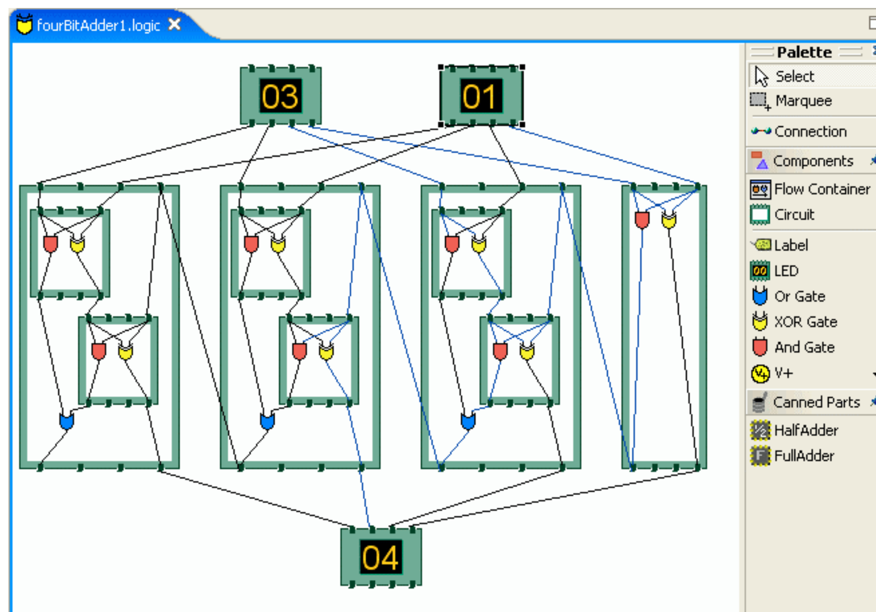

Figure 1: the Logic example implemented in Eclipse

There was also no code generation from the Logic models, but about 5% of the code added on some simulation behavior. A connection 'wire' could show its true/false status by changing color, and an LED display could show the values of its inputs. In a normal DSM scenario, it would be more likely that code could be generated from the model, and running that code could show the values with different input conditions.

The GEF Java code for the sample was 332KB (120 files, over 10,000 lines). Part of the code for the LED display type definition is shown in Listing 1, and part of its display code in Listing 2. To make a 'clean room' implementation in MetaEdit+, we looked at the resulting editor rather than its Java code. The basic concepts of the DSM language were clear from the type palette, and by playing with the example model we found out how the gates could be connected. For instance, there was a distinction between 'in' and 'out' ports on each gate, and connections had to be from an 'out' port to an 'in' port. These kinds of rules are one thing that distinguishes DSM editors from simple drawings in PowerPoint or Visio.

It took about 15 minutes to specify the eight object types along with their port and connection types and rules. With the author's limited graphical skills, drawing and fine-tuning the symbols in the MetaEdit+ Symbol Editor took an additional 45 minutes. The LED object type and symbol definitions are shown in Figure 2. The total of one hour included building the same example model as in Eclipse: a useful step in MetaEdit+, allowing us to test the language while building it.

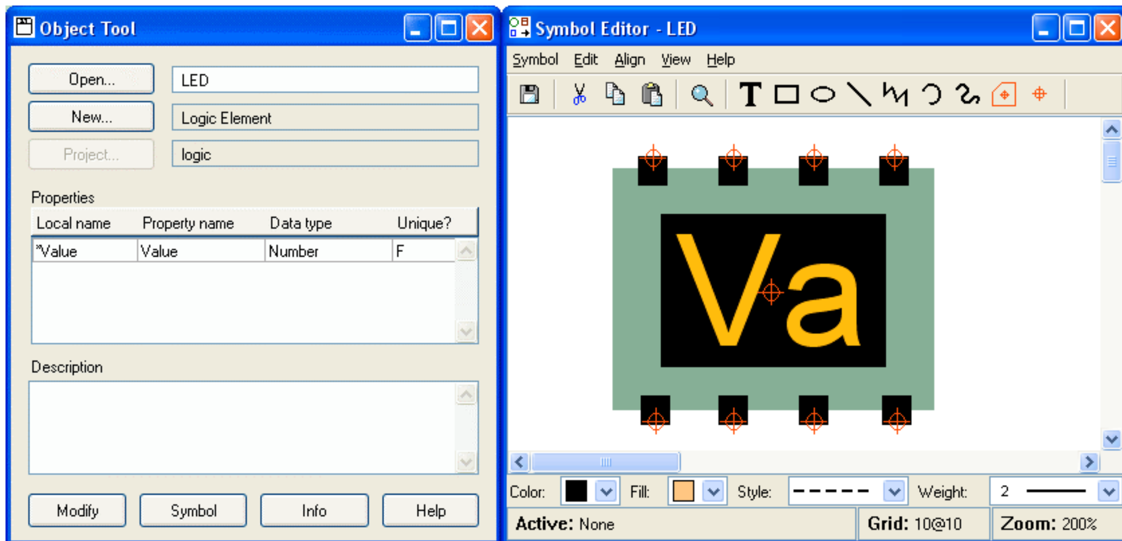

Figure 2: LED object type and symbol definition in MetaEdit+

The resulting editor (Figure 3) was essentially identical to that in GEF, apart from omitting the simulation behavior. That could be added using the MetaEdit+ API with no more code than it took in GEF. Graphical behavior in MetaEdit+ was somewhat better, as connections followed objects as you dragged them (GEF showed only the object outlines while dragging). MetaEdit+ of course also included all its other behavior: browsers, XML import/export, HTML and Word export, multi-user repository etc. Eclipse of course similarly included its own basic behavior, but most of that had nothing to do with modeling: only the XML storage can be considered comparable.

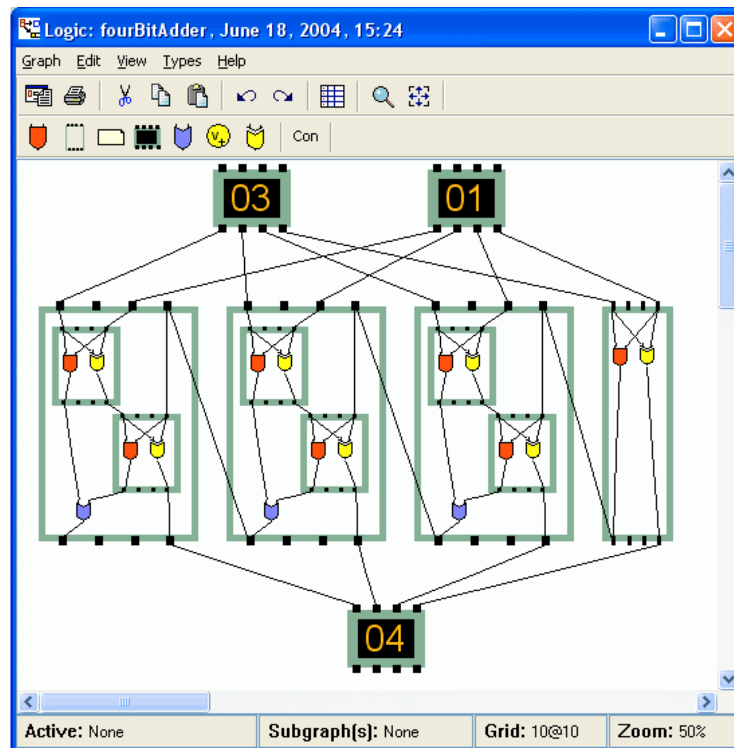

Figure 3: the Logic example implemented in MetaEdit+

5 Conclusion

Clearly, using MetaEdit+ to implement the Logic example was far faster than coding with GEF: one hour, compared to the COCOMO estimate of 13 man-months or 2000 hours for the GEF implementation. Whilst the exact figure is unimportant, its value in terms of cost is relevant when comparing the free Eclipse with the commercial MetaEdit+. The effort to build even this simple editor in GEF would buy you over 10 MetaEdit+ licenses. Full-scale commercial DSM languages have hundreds of types, leading to development costs of millions of dollars: enough to buy several hundred MetaEdit+ licenses.

Part of the benefit of metaCASE tools is in their separation of concerns: the metaCASE tool provider is an expert at building CASE tool functionality, whereas you are the expert in your domain. This natural division of labor is missing from both fixed-method CASE tools and DSM coding frameworks. With a fixed-method CASE tool, you hope the provider is an expert in your domain. With a DSM coding framework like EMF and GEF, you hope your team can become experts in building CASE tools.

The benefits of the separation also extend to the maintenance phase. As the domain evolves, with a metaCASE tool only the language definition changes, whereas with a code-based framework large areas may need recoding. In practice, this often does not happen, and the hand-coded tool stagnates, drifting ever further from the users' needs. Similarly, improvements in CASE technology will be found in the newer versions of both metaCASE tools and DSM coding frameworks, but with the latter, your own code will often no longer work. We tried a few editors built with the previous Eclipse version: none would work with version 3.0.

The EMF framework offers a good solution for those who want to add a measure of non-graphical modeling into their already strong Eclipse development environment. GEF is a good basis for companies wanting to build a graphical editor that does not follow the normal pattern of CASE tool behavior, e.g. a GUI design tool. By also supporting such behavior, however, it misses the chance to offer optimal support for standard CASE behavior. CASE tools can be built with GEF, but the amount of coding necessary will turn many people away.

If IBM intends EMF and GEF to be a serious alternative to metaCASE tools, more work is needed. An interesting project would be to build a metaCASE tool on top of EMF/GEF. Whether this would be significantly easier than building a metaCASE tool on top of any other XML and graphical framework is hard to predict. An extra amount of work could in theory allow good synchronization between generated code viewed in the Eclipse IDE and the models from which that code was generated. That would however presumably also require large amounts of work by the DSM developer. In any case, this comparison has been of EMF and GEF with metaCASE technology: comparing possible future developments in either is clearly not viable until they actually occur.

To summarize: with metaCASE tools, you are free to concentrate on your modeling language, rather than having to implement how each symbol responds to each possible mouse click and draws each line. With the frameworks, you are free to code any behavior you want: worthwhile if you really need that specific behavior. Both thus form viable ways to build DSM support, and your own situation will determine your choice.

6 Glossary: Areas related to DSM

DSL

Domain-Specific Language: a language, normally textual, specifically created for a narrow range of use. Often parsed to generate code or to configure a more generic application.

DSM

Domain-Specific Modeling: using a graphical modeling language specifically created for building a narrow range of applications, e.g. for a family of products within a single company. Normally includes the idea of full code generation directly from the models, using a domain-specific code generator. See www.dsmforum.org.

Little languages

A DSL built quickly for limited use. Can be either embedded in an existing language, or created from scratch. See Dr. Dobb's March 2004.

MDA

Model Driven Architecture. Officially from OMG, the term is now (mis-)used by different people in different ways. Original focus on being able to design business systems independently of the particular middleware solution chosen, by using one UML model for the high-level design, and another fleshing it out for a specific middleware implementation. Within IBM, the focus has moved to a more DSM-like approach, where modeling language concepts are from the domain, although they still envisage using MOF. See www.omg.org/mda.

MDSD

Model-Driven Software Development. A general term for software development based on models. Often, but not always, domain-specific models with code generation.

MOF

Meta Object Facility. A subset of UML, intended for use as a language for describing modeling languages. Its UML bias restricts it largely to describing languages similar to UML. Adopted as a buzzword by many with varying degrees of conformance to the OMG standard. See www.omg.org/mof.

Software manufacturing

Largely a synonym for DSM/DSL, but focusing more on generation. Criticizes UML's 1:1 mapping of modeling constructs to code artifacts, aiming for 1:20 or more. No desire for reverse engineering – who wants to reverse engineer Assembler to C? See Dr. Dobb's April 2004.

Software factories

The Microsoft view of DSM? Strong focus on round-trip engineering and tight integration with vendor frameworks. Broadly opposed to UML-based MDA and MOF. See article in Software Development, July 2004.

7 Listing 1

```
/**

 * Abridged version of LED.java, showing just parts related to property value.
 * Whole file is 3 times as long. In total, the 4 LED classes are 663 lines.
 */

package org.eclipse.gef.examples.logicdesigner.model;

import org.eclipse.gef.examples.logicdesigner.LogicMessages;
import org.eclipse.ui.views.properties.IPropertyDescriptor;
import org.eclipse.ui.views.properties.PropertyDescriptor;
import org.eclipse.ui.views.properties.TextPropertyDescriptor;

public class LED
	extends LogicSubpart
{

public static String P_VALUE = "value";
protected static IPropertyDescriptor[] newDescriptors = null;

static{
	PropertyDescriptor pValueProp = new TextPropertyDescriptor(P_VALUE,
			LogicMessages.PropertyDescriptor_LED_Value);
	pValueProp.setValidator(LogicNumberCellEditorValidator.instance());
	if(descriptors!=null){
		newDescriptors = new IPropertyDescriptor[descriptors.length+1];
		for(int i=0;i<descriptors.length;i++)
			newDescriptors[i] = descriptors[i];
		newDescriptors[descriptors.length] = pValueProp;
	} else
		newDescriptors = new IPropertyDescriptor[]{pValueProp};
}

public Object getPropertyValue(Object propName) {
	if (P_VALUE.equals(propName))
		return new Integer(getValue()).toString();
	if( ID_SIZE.equals(propName)){
		return new String("("+getSize().width+","+getSize().height+")");
	}
	return super.getPropertyValue(propName);
}

public void resetPropertyValue(Object id){
	if (P_VALUE.equals(id))
		setValue(0);
	super.resetPropertyValue(id);
}

public void setPropertyValue(Object id, Object value){
	if (P_VALUE.equals(id))
		setValue(Integer.parseInt((String)value));
	else

		super.setPropertyValue(id,value);
}


}
```

8 Listing 2

```
/**
 * Abridged version of LEDFigure.java, showing just unselected symbol display.
 * Whole file is 4 times as long, plus 65 lines for dragged symbol display.
 */

protected void paintFigure(Graphics g) {
	Rectangle r = getBounds().getCopy();
	g.translate(r.getLocation());
	g.setBackgroundColor(LogicColorConstants.logicGreen);
	g.setForegroundColor(LogicColorConstants.connectorGreen);
	g.fillRectangle(0, 2, r.width, r.height - 4);
	int right = r.width - 1;
	g.drawLine(0, Y1, right, Y1);
	g.drawLine(0, Y1, 0, Y2);

	g.setForegroundColor(LogicColorConstants.connectorGreen);
	g.drawLine(0, Y2, right, Y2);
	g.drawLine(right, Y1, right, Y2);

	// Draw the gaps for the connectors
	g.setForegroundColor(ColorConstants.listBackground);
	for (int i = 0; i < 4; i++) {
		g.drawLine(GAP_CENTERS_X[i] - 2, Y1, GAP_CENTERS_X[i] + 3, Y1);
		g.drawLine(GAP_CENTERS_X[i] - 2, Y2, GAP_CENTERS_X[i] + 3, Y2);
	}

	// Draw the connectors
	g.setForegroundColor(LogicColorConstants.connectorGreen);
	g.setBackgroundColor(LogicColorConstants.connectorGreen);
	for (int i = 0; i < 4; i++) {
		connector.translate(GAP_CENTERS_X[i], 0);
		g.fillPolygon(connector);
		g.drawPolygon(connector);
		connector.translate(-GAP_CENTERS_X[i], 0);

		bottomConnector.translate(GAP_CENTERS_X[i], r.height - 1);
		g.fillPolygon(bottomConnector);
		g.drawPolygon(bottomConnector);
		bottomConnector.translate(-GAP_CENTERS_X[i], -r.height + 1);
	}

	// Draw the display
	g.setBackgroundColor(LogicColorConstants.logicHighlight);
	g.fillRectangle(displayHighlight);
	g.setBackgroundColor(DISPLAY_SHADOW);
	g.fillRectangle(displayShadow);
	g.setBackgroundColor(ColorConstants.black);
	g.fillRectangle(displayRectangle);

	// Draw the value
	g.setFont(DISPLAY_FONT);
	g.setForegroundColor(DISPLAY_TEXT);
	g.drawText(value, valuePoint);
}
```