Knowledge Base Techniques in Web Applications: a Tutorial for Genworks' GDL/GWL

David J. Cooper, Jr.

February 16, 2004

Contents

1 Introduction **1**
1.1 Knowledge Base Concepts According to Genworks 1
1.2 Goals for this Tutorial 1
1.3 What is GDL/GWL? 2
1.4 Why GDL (what is GDL good for?) 3
1.5 What GDL is not 3

2 GDL Syntax **5**
2.1 Define-Object 5
2.2 Making Instances and Sending Messages 6
2.3 Objects 7
2.4 Sequences of Objects and Input-slots with a Default Expression 8
2.5 Summary 9

3 GWL Syntax **11**
3.1 Testing your GWL Installation 11
3.2 GWL:Define-Package 12
3.3 Basic Usage 12
3.4 Page Linking 12
3.5 Form Handling 13
3.6 Publishing URIs for GWL Objects 18
3.7 Higher-level Apps and Graphics 18

4 Example 1: Personal Ledger **19**
4.1 Main Ledger Object 19
4.2 Objects for Accounts and Transactions 24
4.3 Using the Main Ledger Object 24
4.4 Making a Web Interface with GWL 25
4.5 Summary 29

5 Example 2: Simplified Android Robot **33**
5.1 Main UI Sheet for the Robot 33
5.2 Robot Geometry 34
5.3 Using the App 40

6 Example 3: School Bus **43**
6.1 Toplevel Assembly 43
6.2 Interior of School Bus 43
6.3 Causing a Rule Violation 52

Chapter 1

Introduction

1.1 Knowledge Base Concepts According to Genworks

You may have an idea from college or from textbooks that Knowledge Base Systems, or Knowledge *Based* Systems, are a broad or fuzzy set of concepts somehow related to AI which "do not translate into anything practical." Or you may have heard jabs at the pretentious-sounding name, Knowledge-based Engineering as in: "you mean as opposed to Ignorance-based Engineering?" To the contrary, we hope you will agree that our concept of a KB system is simple and practical, and in this tutorial our goal is to make you comfortable and excited about the ideas we have implemented in our flagship system, GDL/GWL.

Our definition of a *Knowledge Base System* is an object-oriented programming environment which implements the features of *Caching* and *Dependency tracking*. Caching means that once the KB has computed something, it might not need to repeat that computation if the same question is asked again. Dependency tracking is the flip side of that coin — it ensures that if a cached result is *stale*, the result will be recomputed the next time it is *demanded*, so as to give a fresh result.

1.2 Goals for this Tutorial

This manual is designed as a companion to a live two-hour GDL/GWL tutorial, but you may also be reading it on your own. In either case, the basic goals are:

- Get you excited about using GDL/GWL
- Enable you to judge whether GDL/GWL is an appropriate tool for a given job
- Arm you with the ability to argue the case for using GDL/GWL when appropriate
- Prepare you to begin maintaining and authoring GDL/GWL applications, or porting apps from similar KB systems into GDL/GWL.

This tutorial assumes a basic familiarity with the Common Lisp programming language. If you are new to Common Lisp: congratulations! You have just discovered an exciting and powerful new tool. Many resources are available to get you started in CL — for starters, we recommend Basic Lisp Techniques[1], which was prepared by the author of this tutorial. Most of the Common Lisp used in this tutorial is covered in the first few chapters of Basic Lisp Techniques, and there you will also find pointers to more exhaustive CL tutorials and reference material.

1.3 What is GDL/GWL?

GDL is an acronym for the General-purpose Declarative Language. GWL is an acronym for the Generative Web Language. GWL ships along with GDL in a layered, modular fashion. For the remainder of this tutorial, we will sometimes refer only to GDL, and this should usually be taken to mean the bundled package which includes GWL.

GDL is a superset of ANSI Common Lisp, and consists mainly of automatic code-expanding extensions to Common Lisp implemented in the form of macros. When you write, let's say, 20 lines in GDL, you might be writing the equivalent of 200 lines of Common Lisp. Of course, since GDL is a superset of Common Lisp, you still have the full power of the CL language at your fingertips whenever you are working in GDL.

Since GDL expands into CL, everything you write in GDL will be compiled "down to the metal" to machine code with all the optimizations and safety that the tested-and-true CL compiler provides. This is an important distinction as contrasted to some other so-called KB systems on the market, which are really nothing more than interpreted scripting languages which often fall over with a "thud" when pushed to compute something more demanding than simple parameter-passing.

GDL can also be considered a *declarative* language. When you put together a GDL application, you write and think mainly in terms of objects and their properties, and how they depend on one another in a direct sense. You do not have to track in your mind explicitly how one object or property will "call" another object or propery, in what order this will happen, etc. Those details are taken care of for you automatically by the language.

Because GDL is object-oriented, you have all the features you would normally expect from an object-oriented language, such as

- Separation between the *definition* of an object and an *instance* of an object
- High levels of data abstraction
- The ability for one object to "inherit" from others
- The ability to "use" an object without concern for its "under-the-hood" implementation

GDL supports the "message-passing" paradigm of object orientation, with some extensions. Since full-blown ANSI CLOS (Common Lisp Object System) is always available as well, the Generic Function paradigm is supported as well. Do not be concerned at this point if you are not fully aware of the differences between these two paradigms[2].

[1] BLT is available at `http://www.franz.com/resources/educational_resources/cooper.book.pdf`

[2] See Paul Graham's ANSI Common Lisp, page 192, for an excellent discussion of the Two Models of Object-oriented Programming.

1.4 Why GDL (what is GDL good for?)

- Organizing and interrelating large amounts of information in ways not possible or not practical using conventional languages or conventional relational database technology alone;
- Evaluating many design or engineering alternatives and performing various kinds of optimizations within specified design spaces;
- Capturing the procedures and rules used to solve repetitive tasks in engineering and other fields;
- Applying rules to achieve intermediate and final outputs, which may include virtual models of wireframe, surface, and solid geometric objects.

1.5 What GDL is not

- A CAD system (although it may operate on and/or generate geometric entities);
- A drawing program (although it may operate on and/or generate geometric entities);
- An Artificial Intelligence system (although it is an excellent environment for developing capabilities which could be considered as such);
- An Expert System Shell (although one could be easily embedded within it).

Without further ado, then, let's turn the page and get started with some hands-on GDL...

Chapter 2

GDL Syntax

2.1 Define-Object

Define-object is the basic macro for defining objects in GDL. An object definition maps directly into a Lisp (CLOS) class definition.

The `define-object` macro takes three basic arguments:

- a *name*, which is a symbol;
- a *mixin-list*, which is a list of symbols naming other objects from which the current object will inherit characteristics;
- a *specification-plist*, which is spliced in (i.e. doesn't have its own surrounding parentheses) after the mixin-list, and describes the object model by specifying properties of the object (messages, contained objects, etc.) The specification-plist typically makes up the bulk of the object definition.

Here are descriptions of the most common keywords making up the specification-plist:

input-slots specify information to be passed into the object instance when it is created.

computed-slots are really cached methods, with expressions to compute and return a value.

objects specify other instances to be "contained" within this instance.

functions are (uncached) functions "of" the object, i.e. they are actually methods which discriminate on their first argument, which is the object instance upon which they are operating. GDL functions can also take other non-specialized arguments, just like a normal CL function.

Figure 2.1 shows a simple example, which contains two input-slots, `first-name` and `last-name`, and a single computed-slot, `greeting`. As you can see, a GDL Object is analogous in some ways to a `defun`, where the input-slots are like arguments to the function, and the computed-slots are like return-values. But seen another way, each attribute in a GDL object is like a function in its own right.

```
(define-object hello ()
  :input-slots (first-name last-name)

  :computed-slots
  ((greeting (format nil "Hello, ~a ~a!!"
                     (the first-name)
                     (the last-name)))))
```

Figure 2.1: Example of Simple Object Definition

The referencing macro `the` shadows CL's `the` (which is a seldom-used type declaration operator). `The` in GDL is a macro which is used to reference the value of other messages within the same object or within contained objects. In the above example, we are using `the` to refer to the values of the messages (input-slots) named `first-name` and `last-name`.

Note that messages used with `the` are given as symbols. These symbols are unaffected by the current Lisp `*package*`, so they can be specified either as plain unquoted symbols or as keyword symbols (i.e. preceded by a colon), and the `the` macro will process them appropriately.

2.2 Making Instances and Sending Messages

Once we have defined an object such as the example above, we can use the constructor function `make-object` in order to create an *instance* of it. This function is very similar to the CLOS `make-instance` function. Here we create an instance of `hello` with specified values for `first-name` and `last-name` (the required input-slots), and assign this instance as the value of the symbol `my-instance`:

```
GDL-USER(16): (setq my-instance
                (make-object 'hello :first-name "John"
                                    :last-name "Doe"))
#<HELLO @ #x218f39c2>
```

As you can see, keyword symbols are used to "tag" the input values, and the return value is an instance of class `hello`. Now that we have an instance, we can use the macro `the-object` to send messages to this instance:

```
GDL-USER(17): (the-object my-instance greeting)
"Hello, John Doe!!"
```

`The-object` is similar to `the`, but as its first argument it takes an expression which evaluates to an object instance. `The`, by contrast, assumes that the object instance is the lexical variable `self`, which is automatically set within the lexical context of a `define-object`.

```
(define-object city ()
  :computed-slots
  ((total-water-usage (+ (the hotel water-usage)
                         (the bank water-usage))))
  :objects
  ((hotel :type 'hotel
          :size :large)
   (bank  :type 'bank
          :size :medium)))
```

Figure 2.2: Object Containing Child Objects

Like `the`, `the-object` evaluates all but the first of its arguments as package-immune symbols, so although keyword symbols may be used, this is not a requirement, and plain, unquoted symbols will work just fine.

For convenience, you can also set `self` manually at the CL Command Prompt, and use `the` instead of `the-object` for referencing:

```
GDL-USER(18): (setq self
                (make-object 'hello :first-name "John"
                                    :last-name "Doe"))
#<HELLO @ #x218f406a>

GDL-USER(19): (the greeting)
"Hello, John Doe!!"
```

In actual fact, `(the ...)` simply expands into `(the-object self ...)`.

2.3 Objects

The `:objects` keyword specifies a list of “contained” instances, where each instance is considered to be a “child” object of the current object. Each child object is of a specified type, which itself must be defined with `define-object` before the child object can be instantiated.

Inputs to each instance are specified as a plist of keywords and value expressions, spliced in after the object’s name and type specification. These inputs must match the inputs protocol (i.e. the input-slots) of the object being instantiated. Figure 2.2 shows an example of an object which contains some child objects. In this example, `hotel` and `bank` are presumed to be already (or soon to be) defined as objects themselves, which each answer the `water-usage` message. The *reference chains*:

```
(the hotel water-usage)
```

```
(defparameter *presidents-data*
    '((:name
       "Carter"
       :term 1976)
      (:name "Reagan"
       :term 1980)
      (:name "Bush"
       :term 1988)
      (:name "Clinton"
       :term 1992)))

(define-object presidents-container ()

  :input-slots
  ((data *presidents-data*))

  :objects
  ((presidents :type 'president
               :sequence (:size (length (the data)))
               :name (getf (nth (the-child index) (the data)) :name)
               :term (getf (nth (the-child index) (the data)) :term))))
```

Figure 2.3: Sample Data and Object Definition to Contain U.S. Presidents

and

```
(the bank water-usage)
```

provide the mechanism to access messages within the child object instances.

These child objects become instantiated *on demand*, meaning that the first time these instances or any of their messages are referenced, the actual instance will be created *and* cached for future reference.

2.4 Sequences of Objects and Input-slots with a Default Expression

Objects may be *sequenced*, to specify, in effect, an array or list of object instances. The most common type of sequence is called a *fixed size* sequence. See Figure 2.3 for an example of an object which contains a sequenced set of instances representing U.S. presidents. Each member of the sequenced set is fed inputs from a list of plists, which simulates a relational database table (essentially a "list of rows").

Note the following from this example:

- In order to sequence an object, the input keyword `:sequence` is added, with a list consisting of the keyword `:size` followed by an expression which must evaluate to a number.
- In the input-slots, `data` is specified together with a default expression. Used this way, input-slots function as a hybrid of computed-slots and input-slots, allowing a *default expression* as with computed-slots, but allowing a value to be passed in on instantiation or from the parent, as with an input-slot which has no default expression. A passed-in value will override the default expression.

2.5 Summary

This GDL syntax overview has been kept purposely brief, covering the fundamentals of the language in a dense manner. On one hand, it is not meant to be a comprehensive language reference; on the other hand, do not be concerned if you are still unsure about some of the terminology. The upcoming examples will revisit and further expand many of the topics covered here, and at some point a coherent picture should begin to emerge.

At that point it will be like riding a bicycle, and there will be no going back.

Chapter 3

GWL Syntax

GWL (Generative Web Language) consists essentially of a set of mixins and a few functions which provide a convenient mechanism to present KB objects defined in GDL through a standard HTTP/HTML web user interface. GWL ships as a standard component of the commercial GDL product, and is available on the GDL Trial Edition CD as well.

GWL is designed to operate in conjunction with AllegroServe[1] and its companion HTML generating facility, htmlgen.

This chapter describes basic GWL usage and syntax. It assumes familiarity with the underlying base language, GDL, covered in the previous chapter.

3.1 Testing your GWL Installation

After you have installed according to `install.htm`:

1. Make sure you have started AllegroServe with:

   ```
   (net.aserve:start :port 9000)
   ```

   (or any port of your choice)

2. In any standard web browser, go to:

   ```
   http://<hostname>:<port>/demos/robot
   ```

   e.g.:

   ```
   http://localhost:9000/demos/robot
   ```

You should see a page with a simple robot assembly made from boxes. If the robot shows up as well as the orange graphical ”compass” below the graphics viewport, your installation is likely working correctly.

[1]AllegroServe is an open-source webserver from Franz Inc, available at http://opensource.franz.com

3.2 GWL:Define-Package

The macro `gwl:define-package` is provided for setting up new working GWL packages.

Example:

```
(gwl:define-package :gwl-user)
```

The `:gwl-user` package is an empty, pre-defined package for your use if you do not wish to make a new package just for scratch work.

For real projects it is recommended that you make and work in your own GWL package.

3.3 Basic Usage

To present a GDL object instance as a web page requires two simple steps:

1. mix in `base-html-sheet` or a subclass thereof
2. define a function in the object called `write-html-sheet`

The `write-html-sheet` function should typically make use of the htmlgen `html` macro. It does not need to take any arguments.

Please see the htmlgen documentation (at `http://opensource.franz.com`, with the AllegroServe distribution, or in `<gdl-home>/doc/aserve/`) for full details on the use of htmlgen.

The code for a simple example object with its `write-html-sheet` presentation function is shown in Figure 3.1. This example contains two optional input slots with values for Name and Term of a president, and creates a simple HTML table displaying this information. As outlined above, in order to make an instance and display this object through a web browser, you would visit the URI:

```
http://<host>:<port>/make?object=gwl-user::president
```

3.4 Page Linking

Creating hypertext links to other pages in the page hierarchy is usually accomplished with the built-in GDL function of `base-html-sheet` called `write-self-link`. This GDL function, when called on a particular page instance, will write a hypertext link referring to that page instance.

These hypertext links are published by AllegroServe "on the fly" (as a side-effect of being demanded), and are made up from the unique root-path of the target object, as well as an instance-id which identifies the particular object instance which is the "root" of the relevant page hierarchy. This is necessary because GWL maintains a table of root-level instances. Each root-level instance will usually correspond to one "user" or session. However, in general, there can be a many-to-many relationship between user sessions and root-level instances.

```
(define-object president (base-html-sheet)
  :input-slots
  ((name "Carter")
   (term 1976))

  :functions
  ((write-html-sheet
    ()
    (html
     (:html
      (:head (:title (format nil "Info on President: ~a" (the name))))
      (:body
       (:table
        (:tr (:td "Name") (:td "Term"))
        (:tr (:td (:princ (the name))) (:td (:princ (the term))))))))))))
```

Figure 3.1: Basic Usage

The instance-id is generated randomly. On a publicly-accessible website, the maximum instance-id should be set to a very large number to decrease the likelihood of a malicious visitor being able to "guess" the instance-id of another user. The maximum is set with the parameter `gwl:*max-id-value*`.

Figures 3.2 and 3.3 show the code for making a page with a list of links to pages representing individual U.S. presidents, resulting in a web page which should resemble Figure 3.4. Note the call to the `write-self-link` function inside the `dolist` in Figure 3.2. This results in an HTML list item being generated with a hyperlink for each "president" child object.

Note also the use of the `write-back-link` function in `presidents-display`. This will generate a link back to the `return-object` of the object, which defaults to the object's `parent`.

3.5 Form Handling

Forms are generated using the GWL macro`with-html-form`. You wrap this macro around the HTMLgen code which creates the contents of the form:

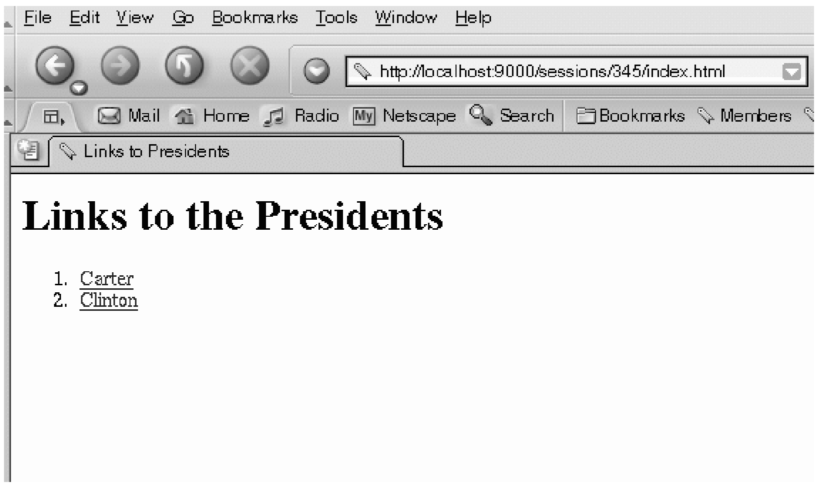
```
(with-html-form ()

  ;; the body of your form goes here

   )
```

```
(define-object presidents-container (base-html-sheet)
  :input-slots
  ((data '((:last-name "Carter" :term 1976)
           (:last-name "Clinton" :term 1992))))
  :objects
  ((presidents :type 'president-display
               :sequence (:size (length (the data)))
               :last-name (getf (nth (the-child index)
                                     (the data))
                                :last-name)
               :term (getf (nth (the-child index)
                                (the data))
                           :term)))
  :functions
  ((write-html-sheet
    ()
    (html
     (:html
      (:head (:title "Links to Presidents"))
      (:body
       (:h1 "Links to the Presidents")
       (:ol
        (dolist (president (list-elements (the presidents)))
          (html
           (:li (the-object president (write-self-link)))))))))))))
```

Figure 3.2: Making a List of Links

```
(define-object president-display (base-html-sheet)
  :input-slots
  (last-name term)

  :computed-slots
  ((strings-for-display (the last-name)))

  :functions
  ((write-html-sheet
    ()
    (html
     (:html
      (:head (:title (:princ (the last-name))))
      (:body
       (:h1 (:princ (the last-name)))
       "Term: " (:princ (the term))
       (:p (the (write-back-link)))))))))
```

Figure 3.3: Link Target


File Edit View Go Bookmarks Tools Window Help
http://localhost:9000/sessions/345/index.html
Mail Home Radio Netscape Search Bookmarks Members
Links to Presidents

Links to the Presidents

1. Carter
2. Clinton

Figure 3.4: Presidents Container Page with Links

The above code snippet would be included in a `write-html-sheet`function of a page definition.

By default, the same object which generates the form will also respond to the form, and is also the object which will have its settable slots modified based on form fields (i.e. html "inputs") of the same name. You can override the default by specifying a `bashee` and/or `respondent`as slots in the requestor object (i.e. the object which is generating the form), for example:

```
:computed-slots
((respondent (the some-other-object))
 (bashee (the yet-another-object)))
```

Any `:settable` computed-slots in the object may be specified as input values (i.e. with `:input` tags) in the form. GWL will automatically infer their types and do appropriate conversion. If the type of a slot can vary, it is best to make its default be a string, then have your application read from the string (with the `read-safe-string` function).

Note that only those input values which have actually changed (according to `equalp`) will be set into the corresponding computed-slot upon form submission. Ones which remain the same will be left alone (to avoid unnecessary dependency updating in the model).

Any `:input` values in the form whose name does not match one of the `:settable` computed-slots in the object will still be collected, but rather than being set into its own named slot, it will be returned as part of the special `query-plist` message when the response page's `write-html-sheet` method is invoked. `Query-plist` is a plist containing keywords representing the form field names, and values which will be strings representing the submitted values.

If you want to do additional processing, the following functions are provided for `base-html-sheet`:

before-set! This is invoked before the "bashee" is modified with any new form values.

after-set! This is invoked after the "bashee" is modified with any new form values.

before-present! This is invoked after the "bashee" is modified with any new form values, but before the page content is returned to the web client.

after-present! This is invoked after the page content is returned to the web client.

By default, these functions are empty, but you can override them to do whatever extra processing you wish.

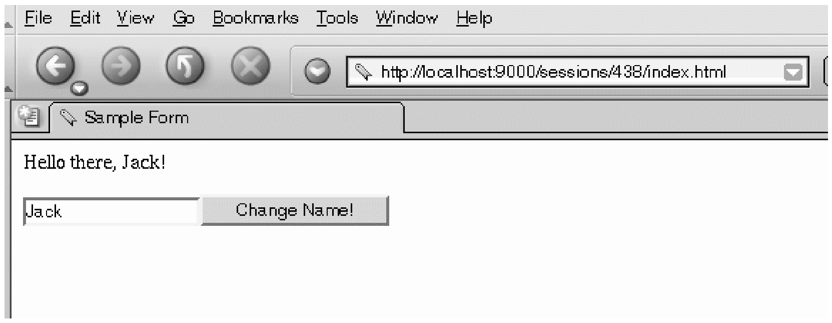
Figure 3.5 shows an object which both generates and responds to a simple form, with the corresponding web page shown in Figure 3.6. The form allows the user to type a name to override the default "Jack," and reflects the submitted name in the form page upon response.

To instantiate this object in a web browser, you would visit the URI:

```
http://<host>:<port>/make?object=gwl-user::hello-form
```

```
(define-object hello-form (base-html-sheet)

  :computed-slots ((username "Jack" :settable))

  :functions
  ((write-html-sheet
    ()
    (html
     (:html
      (:head (:title "Sample Form"))
      (:body
       (:p "Hello there, " (:princ (the username)) "!")
       (:p (with-html-form ()
            ((:input :type :text :name :username
                     :value (the username)))
            ((:input :type :submit :name :submit
                     :value " Change Name! "))))))))))
```

Figure 3.5: Hello Form


File Edit View Go Bookmarks Tools Window Help
http://localhost:9000/sessions/438/index.html
Sample Form
Change Name!

Hello there, Jack!

Jack

Figure 3.6: Hello Form

3.6 Publishing URIs for GWL Objects

You can publish a URI for a given object, to avoid having to type the “make?” expression, using the AllegroServe `publish` function and the GWL `gwl-make-object` function, as per the following example:

```
(publish :path "/demos/bus"
         :function #'(lambda(req ent)
                        (gwl-make-object req ent "bus:assembly")))
```

In this example, the “bus” object would now be instantiated simply by visiting the URI:

`http://<host>:<port>/bus`

3.7 Higher-level Apps and Graphics

GDL/GWL also has the ability to generate and display standard wireframe geometric entities. The complete list of currently available primitive geometric objects is available in the GDL documentation set. Currently the only documented way to display geometry in a GWL application is by using the higher-level mixin `application-mixin`. Two complete examples of the use of this mixin, the Robot and the School Bus, are given in Chapters 5 and 6. Here we will just touch on the basics of how to use this mixin.

1. Instead of `base-html-sheet`, mix in `application-mixin` into the object definition you wish to publish via the web.

2. Collect the objects whose leaves you wish to display as geometry in a computed-slot named`ui-display-list-objects`.

Following the above steps will result in a page with a default user interface which will display your graphics in the center. This page, and each of its components, are highly customizable, and we will look at some of the available customizations in the examples in Chapters 5 and 6.

Chapter 4

Example 1: Personal Ledger

In this chapter we will describe a simple personal accounting ledger application. First we will build the core objects necessary to keep track of accounts and transactions; then we will layer a web user interface on top of these objects to allow for convenient end-user access.

I have chosen to build this application in base GDL/GWL, without the use of any database[1] or other general-purpose mixins. Clearly, one could greatly reduce the amount of code required for an application like this by using "utility" mixins for tasks such as database or filesystem access and standard GUI templates.

While it certainly does not implement the most efficient accounting/ledger algorithms possible, this application will give us a small taste of the power of the caching and dependency-tracking features of a KB system.

4.1 Main Ledger Object

The full source for the Ledger application is available in the GDL application directory under the `gwl-samples` directory. We include partial hardcopy in this tutorial, but if you wish to try running the example yourself you should use the code from the CD rather than trying to type it in from this hardcopy (also, some changes may have occured since the preparation of this tutorial).

The ledger application needs write access to files under the `gwl-samples/ledger/data/` directory, so you may need to open up permissions on these files, or copy the entire `ledger/` directory to a location where you have write access, such as your home directory.

Figure 4.2 shows the input-slots and computed-slots of our main object, named conventionally "assembly" (in the `:ledger` Lisp package). The two inputs each default to a file holding the beginning data set for Accounts and Transactions respectively. Figure 4.1 shows a sampling of the first few lines from typical data files. These data files are given the "lisp" extension simply so that they will come up in Lisp-mode in Emacs for ease of hand-editing – they are not meant to be compiled as Lisp program source code.

The `account-data` and `transaction-data` computed-slots read and hold the contents of these data files for use in initializing the actual `accounts` and `transactions` object sequences, which the ledger application will use. The `account-indices` and `transaction-indices`

[1]Paul Graham is fond of reminding us that "the filesystem is already a database."

```
("Index" "Name" "Description" "Account Number"
 "Account Type" "Account Class" "Beginning Balance")
(0 "DFCU" "Dearborn Federal Credit Union" "999-6969-999"
 "Checking" "Asset/Liability" 25000)
(1 "Waterhouse" "Waterhouse Taxable" "555-7979-555"
 "Savings" "Asset/Liability" 12500)

  ...


("Index" "From Account" "To Account" "Date" "Amount" "Payee")
(0 0 1 3243976597 1000 "djc")
(1 0 1 3243976772 1500 "djc")

  ...
```

Figure 4.1: Contents of Account and Transaction Data Files

collect up the unique indices from the individual acounts and transactions, used to compute the "next" index when adding a new "record." The `net-worth` and `profit` slots compute the sums of Asset/Liability accounts and Income/Expense accounts, respectively (the sign on `profit` is reversed to make Income items appear positive and Expense items appear negative). Finally, the `balances-hash-table` slot is a hash table keyed on the account indices, computing the current balance of each account. This value is passed into the actual account objects for later use.

Figure 4.3 shows the two sub-objects contained within the ledger assembly object. Each of these is specified as a `sequence`, based on the initial data read from the data files. Because these sequences are specified by a list of `:indices`, rather than by a fixed `:size`, they can be programmatically modified by inserting and deleting sequence elements. In this example, such insertions or deletions would be analogous to row operations on a relational database table.

Note that the `current-balance` is accessed from the `balances-hash-table`, a computed-slot listed in Figure 4.2. The final segment of our main ledger object is listed in Figure 4.4. Here, we specify some functions on the ledger object which accept arguments and perform some side-effect on the object. Note that GDL `:functions` are distinguished from `:computed-slots` by two main characteristics:

1. They can accept arguments, a "lambda list," as with a normal CL function.

2. Their return-values are not cached or dependency-tracked. Their bodies are evaluated every time they are referenced (called).

The `add-transaction!` function adds a new transaction element to the `transactions`

```
(define-object assembly (base-object)

  :input-slots
  ((account-data-file "~/genworks/gwl-apps/ledger/data/accounts.lisp")
   (transaction-data-file "~/genworks/gwl-apps/ledger/data/transactions.lisp"))

  :computed-slots
  ((account-data (let (result)
                   (with-open-file (in (the account-data-file))
                     (do ((account (read in nil nil) (read in nil nil)))
                         ((null account) (nreverse result))
                       (push account result)))))

   (transaction-data (let (result)
                       (with-open-file (in (the transaction-data-file))
                         (do ((transaction (read in nil nil) (read in nil nil)))
                             ((null transaction) (nreverse result))
                           (push transaction result)))))

   (account-indices (mapcar #'(lambda(account) (the-object account index))
                            (list-elements (the accounts))))

   (transaction-indices (mapcar #'(lambda(transaction)
                                    (the-object transaction index))
                                (list-elements (the transactions))))

   (net-worth (let ((result 0))
                (dolist (account (list-elements (the accounts)) result)
                  (when (eql (make-keyword (the-object account account-class))
                             :asset/liability)
                    (incf result (the-object account current-balance))))))

   (profit (let ((result 0))
             (dolist (account (list-elements (the accounts)) result)
               (when (eql (make-keyword (the-object account account-class))
                          :income/expense)
                 (decf result (the-object account current-balance))))))

   (balances-hash-table (let ((ht (make-hash-table)))
                          (dolist (account (list-elements (the accounts)))
                            (setf (gethash (the-object account index) ht)
                              (the-object account beginning-balance)))
                          (dolist (transaction (list-elements
                                                (the transactions)) ht)
                            (decf
                             (gethash (the-object transaction from-account) ht)
                             (the-object transaction amount))
                            (incf
                             (gethash (the-object transaction to-account) ht)
                             (the-object transaction amount)))))) ...
```

Figure 4.2: Input-Slots and Computed-Slots of Ledger

```
  ...

:objects
((accounts :type 'account
           :sequence (:indices (mapcar #'first (rest (the account-data))))
           :data (nth (the-child index) (rest (the account-data)))
           :current-balance (gethash (the-child index)
                                     (the balances-hash-table))
           :headings (first (the account-data)))

 (transactions :type 'transaction
               :sequence (:indices (mapcar #'first
                                           (rest (the transaction-data))))
               :data (nth (the-child index) (rest (the transaction-data)))
               :headings (first (the transaction-data))))


  ...
```

Figure 4.3: Sub-Objects of Ledger

object sequence, and immediately calls the `save-transactions!` function to update the transactions data file[2].

The most important thing to note here is that when a new transaction is added using the `add-transaction!` function, any other message (e.g. computed-slot, object, etc.) which in any way depends upon the `transactions` sequence of objects will now automatically recompute and return a fresh value the next time it is demanded. For example, the `profit` and `net-worth` messages will now return updated values the next time they are demanded. But the recomputation of a message will happen if and only if the message is actually demanded. In a very large object tree, for example, thousands of messages in hundreds of objects might depend on a certain value.

When that value is changed, however, the system does not incur the computational overhead of updating all these thousands of dependent items at that time. Perhaps only a few dozen of these thousands of dependent items will ever be accessed. Only those few dozen will need to be computed.

This is one of the obvious distinctions between a conventional “spreadsheet” application and a knowledge base application — a spreadsheet is generally not scalable to very large problems or models, because changing a value forces the user to wait for an all-or-nothing update of the entire sheet.

[2] In a more robust application, the in-memory insertion and the updating of the external file should be done as an indivisible unit with “unwind” capability, to ensure that either the whole thing succeeds or the whole thing fails.

```
 ...

:functions
((add-transaction! (&key from-account to-account date amount payee)
  (when (or (not (member from-account (the account-indices)))
            (not (member to-account (the account-indices)))
            (not (numberp amount)) (not (stringp payee))
            (not (ignore-errors (decode-universal-time date))))
    (error "One or more invalid arguments given to add-transaction!"))
  (let ((new-index (1+ (if (null (the transaction-indices)) 0
                         (apply #'max (the transaction-indices))))))
    (the transactions (insert! new-index))
    (the (transactions new-index)
      (set-slot! :data (list new-index from-account
                             to-account date amount payee))))
  (the save-transactions!))

 (save-transactions! ()
  (with-open-file (out (the transaction-data-file)
                   :direction :output :if-exists :supersede
                   :if-does-not-exist :create)
    (print (first (the transaction-data)) out)
    (dolist (transaction (list-elements (the transactions)))
      (print (the-object transaction data) out))))


    ;; ... Similarly for add-account! and save-accounts!

 )))
```

Figure 4.4: Functions of Ledger

```
(define-object account (base-object)
  :input-slots
  (data headings current-balance)

  :computed-slots
  ((name (second (the data)))
   (description (third (the data)))
   (account-number (fourth (the data)))
   (account-type (fifth (the data)))
   (account-class (sixth (the data)))
   (beginning-balance (seventh (the data)))))

(define-object transaction (base-object)
  :input-slots
  (data headings)

  :computed-slots
  ((from-account (second (the data)))
   (to-account (third (the data)))
   (date (fourth (the data)))
   (amount (fifth (the data)))
   (payee (sixth (the data)))))
```

Figure 4.5: Account and Transaction Object Definitions

4.2 Objects for Accounts and Transactions

Figure 4.5 shows the object definitions for `account` and `transaction`. These are simple objects which essentially receive some inputs and make them available as messages.

4.3 Using the Main Ledger Object

We can use the ledger object we have put together so far by typing commands at the Common Lisp prompt. First, we will make an instance of a ledger object, conveniently setting this instance to the variable `self`:

```
LEDGER(10): (setq self (make-object 'assembly))
#<ASSEMBLY @ #x7367516a>
```

Now we can use “the” referencing to call various messages in this instance:

```
LEDGER(11): (the profit)
140
```

```
LEDGER(12): (the net-worth)
37640
LEDGER(13): (the balances-hash-table)
#<EQL hash-table with 7 entries @ #x7367fc1a>
LEDGER(14): (the (accounts 0) current-balance)
28140
LEDGER(15):
```

Now we will add a transaction, and confirm that it affects our `profit` and `net-worth`:

```
LEDGER(15): (the (add-transaction! :from-account 0 :to-account 4
                                   :date (get-universal-time)
                                   :amount 250 :payee "djc"))
NIL
LEDGER(16): (the profit)
-110
LEDGER(17): (the net-worth)
37390
LEDGER(18):
```

If we had wrapped the CL `time` macro around for example the call to the `profit`, we would have been able to see that indeed it was recomputed the second time we called it, since something it depends upon had been modified. If we call it again immediately, without having changed anything, `time` would show us that it returns virtually instantaneously without causing any substantial work to be done, since its value is now cached and the cache is still fresh.

4.4 Making a Web Interface with GWL

Now that we have built and tested our main ledger "engine," let's make it more accessible to casual users by layering a web user interface on it. One way to do this is to create a new toplevel object which *contains* the ledger engine, and specifies an assembly of objects to represent the web pages in our interface. Figure 4.6 shows the definition of the top level, or "home page," of our web application.

It contains the actual ledger assembly, or "engine," as well as child objects, which represent sub-pages in our website, corresponding to an account listing, a transaction listing, a form for adding a transaction, and a form for adding an account. Our toplevel user interface sheet also contains a `write-html-sheet` presentation function, shown in Figure 4.7. This presentation function also serves as a response function to the forms for adding accounts and transactions – for this reason, it contains two blocks of code, before the actual html page generation, which take care of actually processing any added transaction or account.

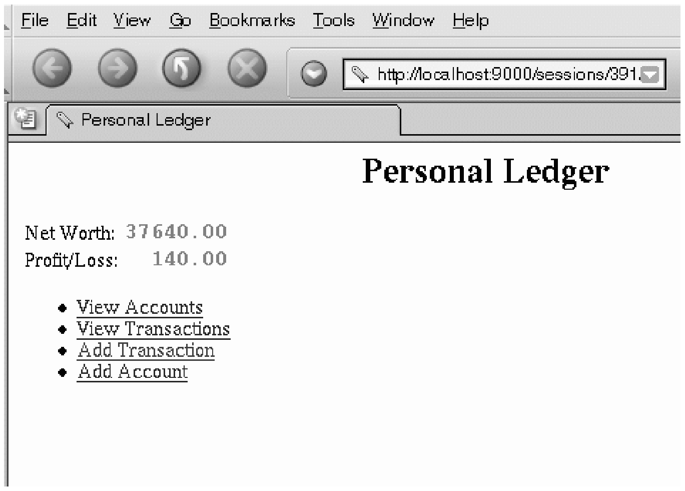
The form values for these will show up as plist values in the `query-plist` slot, since they are not specified as named slots in the objects which generate the forms (the transaction form is shown in Figure 4.11, and the accounts form is not listed here but is similar).

```
(define-object html-assembly (base-html-sheet)

  :computed-slots
  ((accounts-ht (let ((ht (make-hash-table)))
                  (dolist (account (list-elements
                                    (the ledger accounts)) ht)
                    (setf (gethash (the-object account index) ht)
                      account)))))

  :objects
  ((ledger :type 'assembly)

   (view-accounts :type 'view-accounts
                  :account-sequence (the ledger accounts))

   (view-transactions :type 'view-transactions
                      :transaction-sequence (the ledger transactions)
                      :pass-down (accounts-ht))

   (add-transaction :type 'add-transaction
                    :main-sheet self
                    :pass-down (accounts-ht))

   (add-account :type 'add-account
                :main-sheet self))
 ...
```

Figure 4.6: Slots and Object for Web Interface

```
:functions
((write-html-sheet
  ()
  (let ((plist (the add-transaction query-plist)))
    (when plist
      (the ledger
        (add-transaction!
         :from-account (read-safe-string (getf plist :from-account))
         :to-account (read-safe-string (getf plist :to-account))
         :date (iso-to-universal (getf plist :date))
         :amount (read-safe-string (getf plist :amount))
         :payee (getf plist :payee))))
    (the add-transaction (:set-slot! :query-plist nil)))
  (let ((plist (the add-account query-plist)))
    (when plist
      (the ledger (add-account!
                   :name (getf plist :name)
                   :description (getf plist :description)
                   :account-number (getf plist :account-number)
                   :account-type (getf plist :account-type)
                   :account-class (getf plist :account-class)
                   :beginning-balance (read-safe-string
                                       (getf plist :beginning-balance)))))
    (the add-account (:set-slot! :query-plist nil)))
  (html
   (:html
    (:head (:title "Personal Ledger"))
    (:body
     (:h2 (:center "Personal Ledger"))
     (:p (:table
          (:tr (:td "Net Worth:")
               ((:td :align :right)
                (:b (:tt ((:font
                           :color (gethash (if (minusp (the ledger net-worth))
                                               :red :green-lime) *color-table*))
                          (:princ (number-format (the ledger net-worth) 2)))))))
          (:tr (:td "Profit/Loss:")
               ((:td :align :right)
                (:b (:tt ((:font
                           :color (gethash (if (minusp (the ledger profit))
                                               :red :green-lime) *color-table*))
                          (:princ (number-format (the ledger profit) 2)))))))))
     (:p (:ul (:li (the view-accounts
                     (write-self-link :display-string "View Accounts")))
              (:li (the view-transactions
                     (write-self-link :display-string "View Transactions")))
              (:li (the add-transaction
                     (write-self-link :display-string "Add Transaction")))
              (:li (the add-account
                     (write-self-link :display-string "Add Account"))))))))))
```

Figure 4.7: Output Function for Web Interface

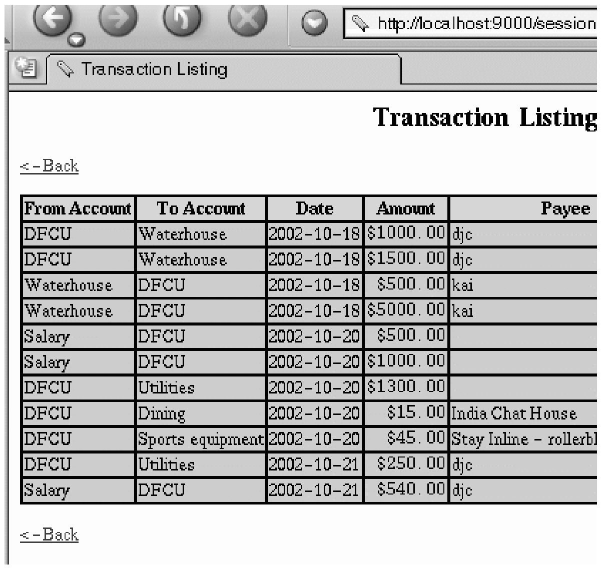

File Edit View Go Bookmarks Tools Window Help
http://localhost:9000/sessions/391.
Personal Ledger

Personal Ledger

Net Worth: 37640.00
Profit/Loss: 140.00

- View Accounts
- View Transactions
- Add Transaction
- Add Account

Figure 4.8: Main Screen of Ledger



Transaction Listing

<--Back

From Account	To Account	Date	Amount	Payee
DFCU	Waterhouse	2002-10-18	$1000.00	djc
DFCU	Waterhouse	2002-10-18	$1500.00	djc
Waterhouse	DFCU	2002-10-18	$500.00	kai
Waterhouse	DFCU	2002-10-18	$5000.00	kai
Salary	DFCU	2002-10-20	$500.00	
Salary	DFCU	2002-10-20	$1000.00	
DFCU	Utilities	2002-10-20	$1300.00	
DFCU	Dining	2002-10-20	$15.00	India Chat House
DFCU	Sports equipment	2002-10-20	$45.00	Stay Inline - rollerbl
DFCU	Utilities	2002-10-21	$250.00	djc
Salary	DFCU	2002-10-21	$540.00	djc

<--Back

Figure 4.9: Transaction Listing Sheet

Figure 4.10 defines an object which takes two inputs and computes two simple slots, but most of all it defines a presentation method to generate an html table listing all transactions entered to date. An example is seen in Figure 4.9. Figure 4.11 defines a presentation function to create a fillout-form for adding a transaction, and Figure 4.12 shows a sample rendition of this form.

4.5 Summary

Please note that we have built this ledger using only core GDL/GWL, for illustrative purposes. Several parts of the app have been written from scratch, which otherwise could be handled by using simple utilities or libraries for things such as filesystem/database access and user interface templates.

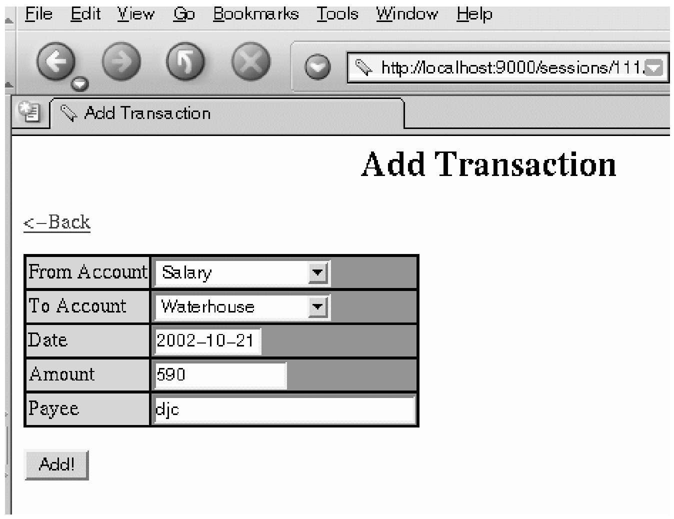
The main point of this example is to show that KB technology can be useful even for applications which might be considered "simple" or "mainstream" — even the simplest of applications tend to grow ever-changing and ever-expanding requirements, and it pays in the end to use a development environment which can absorb these requirements gracefully and with ease.

```
(define-object view-transactions (base-html-sheet)
  :input-slots (transaction-sequence accounts-sequence)

  :computed-slots ((transactions (list-elements
                                  (the transaction-sequence)))
                   (headings (the-object (first (the transactions))
                                         headings)))
  :functions
  ((write-html-sheet
    ()
    (html
     (:html
      (:head (:title "Transaction Listing"))
      (:body
       (:h2 (:center "Transaction Listing"))
       (:p (the (:write-back-link)))
       (:p ((:table :bgcolor :black)
            ((:tr :bgcolor :yellow)
             (dolist (heading (rest (the headings)))
               (html (:th (:princ heading)))))
            (dolist (transaction (the transactions))
              (html
               ((:tr :bgcolor (gethash :grey-light-very
                                       *color-table*))
                (dolist (slot (list :from-account :to-account
                                    :date :amount :payee ))
                  (let* ((raw-value
                          (the-object transaction (evaluate slot)))
                         (value (case slot
                                  ((:from-account :to-account)
                                   (the :accounts-sequence
                                     (get-member raw-value) :name))
                                  (:date (iso-date raw-value))
                                  (:amount (number-format raw-value 2))
                                  (otherwise raw-value))))
                    (html ((:td :align (case slot (:amount :right)
                                             (otherwise :left)))
                           (case slot (:amount
                                       (html
                                        (:tt (format *html-stream*
                                                     "$~$" value))))
                                 (otherwise
                                  (html (:princ value))))))))))))
       (:p (the (:write-back-link))))))))
```

Figure 4.10: Sheet for Transaction Listing

```
(define-object add-transaction (base-html-sheet)
  :input-slots (accounts-sequence main-sheet)

  :computed-slots ((respondent (the main-sheet)))

  :functions
  ((write-html-sheet
    ()
    (html
     (:html
      (:head (:title "Add Transaction"))
      (:body
       (:h2 (:center "Add Transaction"))
       (:p (the (:write-back-link)))
       (with-html-form
        (:p
         ((:table :bgcolor :black)
          (:tr
           ((:td :bgcolor :yellow) "From Account")
           ((:td :bgcolor (gethash :green-spring *color-table*))
            ((:select :name :from-account)
             (mapcar #'(lambda(account)
                         (html ((:option
                                 :value (the-object account index))
                                (:princ (the-object account name)))))
                     (list-elements (the accounts-sequence))))))
          (:tr
           ((:td :bgcolor :yellow) "To Account")
           ((:td :bgcolor (gethash :green-spring *color-table*))
            ((:select :name :to-account)
             (mapcar #'(lambda(account)
                         (html ((:option
                                 :value (the-object account index))
                                (:princ (the-object account name)))))
                     (list-elements (the accounts-sequence))))))
          (:tr
           ((:td :bgcolor :yellow) "Date")
           ((:td :bgcolor (gethash :green-spring *color-table*))
            ((:input :type :text :name :date :size 12
                     :value (iso-date (get-universal-time))))))
          (:tr ((:td :bgcolor :yellow) "Amount")
               ((:td :bgcolor (gethash :green-spring *color-table*))
                ((:input :name :amount :type :text :size 15))))
          (:tr ((:td :bgcolor :yellow) "Payee")
               ((:td :bgcolor (gethash :green-spring *color-table*))
                ((:input :name :payee :type :text :size 30))))))
        (:p ((:input :type :submit :name :add-transaction
                     :value " Add! "))))))))))
```

Figure 4.11: Form for Adding a Transaction


File Edit View Go Bookmarks Tools Window Help
http://localhost:9000/sessions/111
Add Transaction
<–Back
Add!

Add Transaction

From Account	Salary
To Account	Waterhouse
Date	2002–10–21
Amount	590
Payee	djc

Figure 4.12: Form for Adding a Transaction

Chapter 5

Example 2: Simplified Android Robot

This chapter describes and shows the complete code for a Simplified Android Robot[1] implemented in GDL/GWL. Here we introduce the use of geometric primitive objects, as well as the use of the higher-level `application-mixin` first introduced in Chapter 3.

We also introduce the concept of the *view*, which allows separation of presentation functions (e.g. for HTML output) from the core object definition.

5.1 Main UI Sheet for the Robot

Figure 5.1 defines the toplevel user interface sheet for the robot. It specifies several `:settable` computed-slots which the user will end up being able to set through an HTML form. It also specifies the `robot` child object, whose leaves contain the actual geometry (boxes in this case) of the robot.

Figure 5.2 shows the definition of a *view* which is defined for the `html-format` output format, and the `robot-assembly` GDL object. Rather than being associated with a single type as with normal GDL objects, views are associated with *two* types – an output format and a normal GDL object type. The `:output-functions` defined within the view will therefore be associated with the *combination* of the given output-format and the given GDL object type. In this case, we are specifying the `model-inputs :output-function` to be applied to the combination of the `html-format` output-format and the `robot-assembly` GDL object type.

The `application-mixin` contains a default (essentially blank) `model-inputs` function, and here we are overriding it to do something specific, namely to display html input fields for the slots in our `robot-assembly` which we wish the user to be able to alter through a form.

By default, the `application-mixin` will display the output from the `model-inputs` function in the upper-right area of the user interface sheet, as seen in Figure 5.3. These

[1]The "Simplified Android Robot" is a traditional example used for pedagogical purposes in computer graphics, and as far as we know it has its origins in Computer Graphics: Principles and Practice by Foley, Feiner, and Van Dam.

```
(define-object robot-assembly (application-mixin)

  :computed-slots
  ((width 5 :settable)
   (length 2 :settable)
   (height 10 :settable)
   (head-angle 0 :settable)
   (body-angle 0 :settable)
   (arm-angle-right 0 :settable)
   (arm-angle-left 0 :settable)
   (pincer-distance-right (single-float
                           (number-round (* .15 3/5 (the width)) 3))
                          :settable)
   (pincer-distance-left (single-float
                          (number-round (* .15 3/5 (the width)) 3))
                         :settable)
   (image-format (the view-object image-format))
   (strings-for-display "Robot Assembly")
   (ui-display-list-objects (the robot)))

  :objects
  ((robot :type 'robot
          :pass-down (:head-angle
                      :body-angle :arm-angle-right :arm-angle-left
                      :pincer-distance-right :pincer-distance-left))))
```

Figure 5.1: UI Sheet for Robot

input fields are automatically contained inside an appropriate HTML form entity - when using `application-mixin`, there is no need for application-level code to generate the HTML form tag or the `:respondant` or `:bashee` hidden fields described in Chapter 3 with plain `base-html-sheet`.

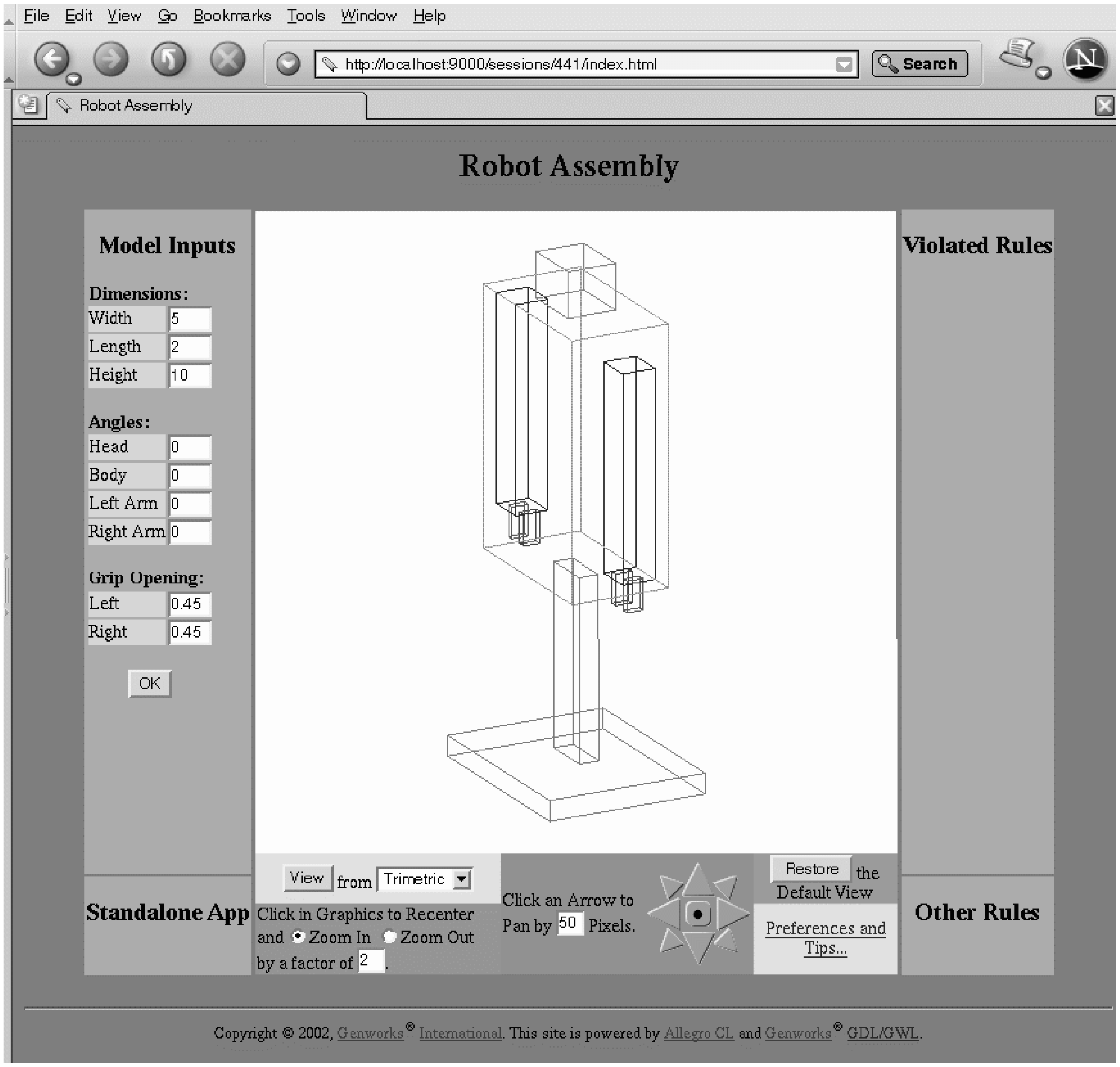
5.2 Robot Geometry

The actual robot is made up of two child objects, its `base` and its `body`, as shown in Figure 5.4. Figure 5.5 shows the definition of the body, and 5.7 shows the definition of the base. The body is made up of a torso (a box), a head (a box), and two arms, whose definition is shown in Figure 5.6. The `:settable` :computed-slots from the toplevel UI sheet come into the `robot` as input-slots. These serve as parameters for the rest of the robot hierarchy.

The positioning and orientation of each child object are specified by passing `:center` and `:orientation` into the child part:

```
(define-view (html-format robot-assembly)()
  :output-functions
  ((model-inputs
    ()
    (html
     ((:table  :cellpadding 0)
      (:tr ((:td :colspan 2) (:b "Dimensions:")))
      (dolist (slot (list :width :length :height))
        (html
         (:tr ((:td :bgcolor :yellow)
               (:princ (string-capitalize slot)))
              (:td ((:input :type :string :name slot :size 5
                            :value (format nil "~a"
                                           (the (evaluate slot)))))))))
      (:tr ((:td :colspan 2) :br))
      (:tr ((:td :colspan 2) (:b "Angles:")))
      (dolist (angle '(("Head" :head-angle)("Body" :body-angle)
                      ("Left Arm" :arm-angle-left)
                      ("Right Arm" :arm-angle-right)))
        (html
         (:tr ((:td :bgcolor :yellow) (:princ (first angle)))
              (:td ((:input
                     :type :string :name (second angle)
                     :size 5 :value
                     (format nil "~a" (the (evaluate (second angle))))))))))
      (:tr ((:td :colspan 2) :br))
      (:tr ((:td :colspan 2) (:b "Grip Opening:")))
      (dolist (side (list :left :right))
        (html
         (:tr
          ((:td :bgcolor :yellow) (:princ (string-capitalize side)))
          (:td
           ((:input
             :type :string
             :name (format nil "pincer-distance-~a" side)
             :size 5
             :value
             (the (evaluate (make-keyword
                             (format nil "pincer-distance-~a" side))))))))))
      (:tr ((:td :colspan 2) :br))
      (:tr ((:td :colspan 2 :align :center)
            ((:input :type :submit :value " OK " :name :refresh))))))))
```

Figure 5.2: Inputs Section for UI Sheet



http://localhost:9000/sessions/441/index.html

Robot Assembly

Model Inputs

Dimensions:

Width	5
Length	2
Height	10

Angles:

Head	0
Body	0
Left Arm	0
Right Arm	0

Grip Opening:

Left	0.45
Right	0.45

Click in Graphics to Recenter and [✓ Zoom In] Zoom Out by a factor of 2.

Click an Arrow to Pan by 50 Pixels.

Restore the Default View

Copyright © 2002, Genworks® International. This site is powered by Allegro CL and Genworks® GDL/GWL.

Figure 5.3: Default Robot

```
(define-object robot (base-object)

  :input-slots
  (head-angle
   body-angle
   arm-angle-right
   arm-angle-left
   pincer-distance-right
   pincer-distance-left)

  :computed-slots
  ((display-controls (list :color :green-lime)))

  :objects
  ((base :type 'robot-base
         :height (* (the :height) 0.4)
         :width (* (the :width) 0.2)
         :length (* (the :length) 0.2)
         :center (translate (the :center) :down (* (the :height) 0.3)))
   (body :type 'robot-body
         :height (* (the :height) 0.6)
         :center (translate (the :center) :up (* (the :height) 0.2))
         :orientation
         (alignment :right
                    (rotate-vector-d
                     (the (:face-normal-vector :right))
                     (the :body-angle)
                     (the (:face-normal-vector :top))))
         :pass-down (:head-angle
                     :arm-angle-right :arm-angle-left
                     :pincer-distance-left :pincer-distance-right))))
```

Figure 5.4: Toplevel of Robot Geometry

:center is given as a 3D point, and causes the child object to treat this point as its center.

:orientation is given as a 3-by-3 rotational transformation matrix, and causes the child object to adjust its six `face-normal-vector`s accordingly. As with this example, this transformation matrix is usually created using the `alignment` function, which allows you to align up to three faces of the child object with up to three given vectors. The first vector will be taken exactly, and the second and third vectors will be taken for their orthogonal components to the previous ones.

```
(define-object robot-body (base-object)
  :input-slots
  (head-angle arm-angle-left arm-angle-right
   pincer-distance-left pincer-distance-right
   (shoulder-height (* (the :torso :height) 0.1)))

  :computed-slots
  ((display-controls (list :color :blue-steel-light)))

  :objects
  ((torso :type 'box :height (* (the :height) 0.85)
          :width (* (the :width) 0.7)
          :center (translate (the :center)
                             :down (- (half (the :height))
                                      (half (the-child :height)))))
   (head :type 'box :display-controls (list :color :magenta)
         :height (- (the :height) (the :torso :height))
         :width (* (the :width) 0.25) :length (half (the :length))
         :center (translate (the :center) :up
                            (- (half (the :height)) (half (the-child :height))))
         :orientation (alignment :right
                                 (rotate-vector-d
                                  (the (:face-normal-vector :right))
                                  (the :head-angle)
                                  (the (:face-normal-vector :top)))))
   (arms :type 'robot-arm :sequence (:size 2)
         :side (ecase (the-child :index) (0 :left) (1 :right))
         :width (half (- (the :width) (the :torso :width)))
         :length (/ (the :length) 3)
         :height (- (the :torso :height) (twice (the :shoulder-height)))
         :center (translate-along-vector
                  (the-child :shoulder-point)
                  (the-child (:face-normal-vector :bottom))
                  (half (the-child :height)))
         :orientation
         (alignment :bottom
                    (rotate-vector-d (the (:face-normal-vector :bottom))
                                     (the-child :angle)
                                     (the (:face-normal-vector :left)))
                    :right (the (:face-normal-vector :right)))
         :shoulder-point
         (translate (the :torso (:edge-center :top (the-child :side)))
                    (the-child :side) (half (the-child :width)) :down
                    (the :shoulder-height))
         :angle (ecase (the-child :side) (:left (the :arm-angle-left))
                       (:right (the :arm-angle-right)))
         :pincer-distance (ecase (the-child :side)
                            (:left (the :pincer-distance-left))
                            (:right (the :pincer-distance-right))))))
```

Figure 5.5: Robot Body

```
(define-object robot-arm (base-object)

  :input-slots
  (side
   shoulder-point
   angle
   pincer-distance)

  :computed-slots
  ((display-controls (list :color :blue)))

  :objects
  ((arm :type 'box)
   (thumb :type 'box
          :display-controls (list :color :red)
          :width (the :hand :width)
          :height (the :hand :height)
          :length (the :hand :length)
          :center (translate (the :hand :center)
                             (the :side) (the :pincer-distance)))
   (hand :type 'box
         :display-controls (list :color :green)
         :center (translate (the :center) :down
                            (+ (half (the :height))
                               (half (the-child :height)))
                            (the :side)
                            (- (- (half (the :width))
                                  (half (the-child :width)))))
         :height (* (the :height) 0.15)
         :width (* (the :width) 0.2)
         :length (half (the :length)))))
```

Figure 5.6: Robot Arm

```
(define-object robot-base (base-object)

  :objects
  ((leg :type 'box
        :height (* 0.9 (the :height))
        :center (translate (the :center) :up
                           (- (half (the :height))
                              (half (the-child :height)))))
   (foot :type 'box
         :height (* 0.1 (the :height))
         :width (twice (twice (the :width)))
         :length (twice (twice (twice (the :length))))
         :center (translate (the :center) :down
                            (- (half (the :height))
                               (half (the-child :height)))))))
```

Figure 5.7: Robot Base

5.3 Using the App

Figures 5.8 and 5.9 show some examples of a user having interacted with the application, resulting in a specified standard 3D view of the robot or the robot's body parts rotated to specified angles. Note also the extra hyperlinks at the upper-left in Figure 5.8, which are a result of the GDL/GWL session being in *development mode*. Development mode can be entered as follows:

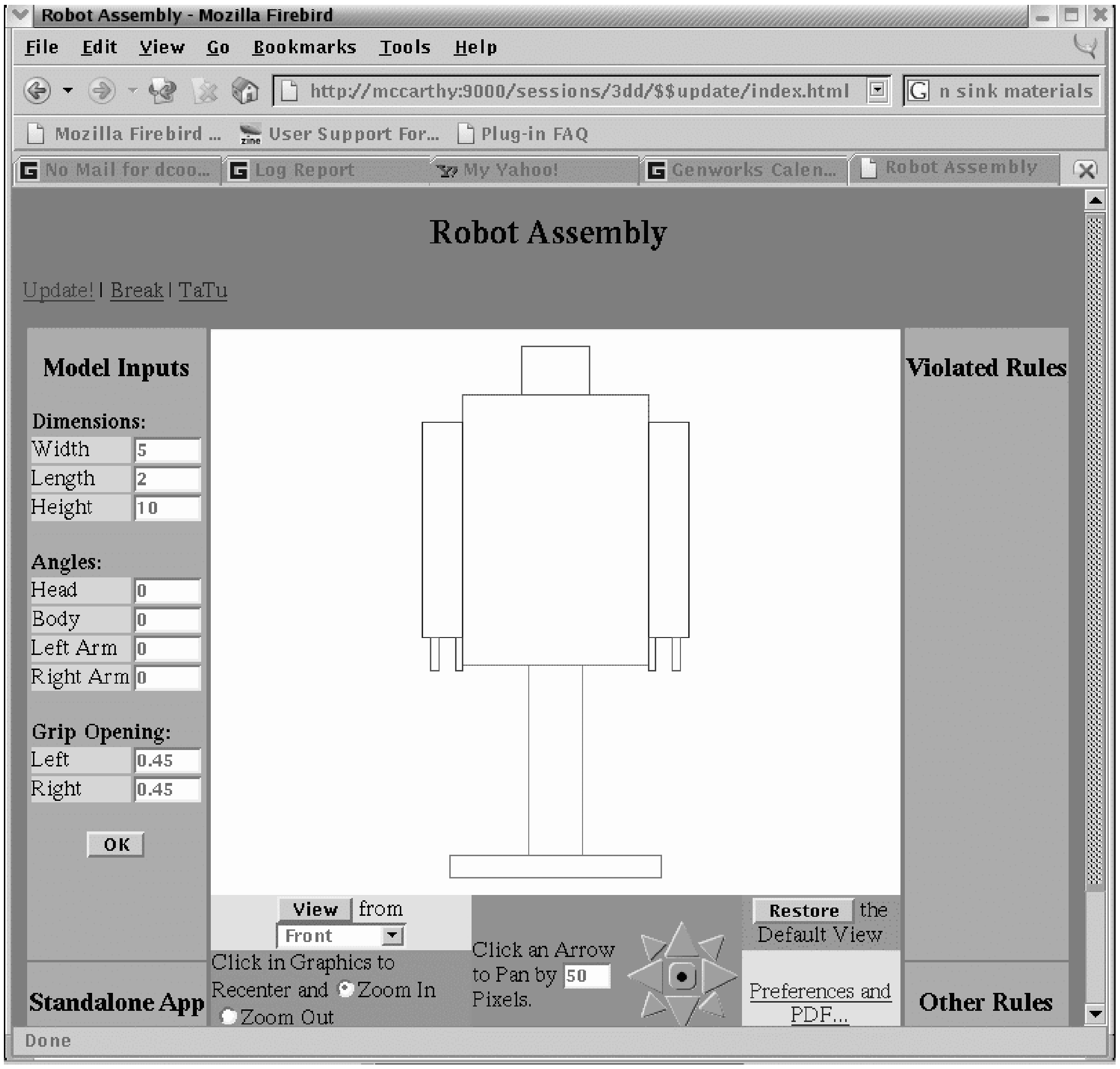
```
(setq gwl:*developing?* t)
```

The three standard links provided by development mode are as follows:

Update! will essentially re-instantiate the object hierarchy, taking into account any new or altered definitions you have compiled since the objects were last demanded. However, any `:settable` slots which have been altered will retain their values to the extent feasible.

Break will cause a Common Lisp break level to be entered, with the parameter `self` set to the object instance corresponding to the current web page.

TaTu will respond with the development view of the object, as described in the file `tatu.txt`.


Robot Assembly - Mozilla Firebird
File Edit View Go Bookmarks Tools Help
http://mccarthy:9000/sessions/3dd/$$update/index.html
n sink materials
Mozilla Firebird ... User Support For... Plug-in FAQ
No Mail for dcoo... Log Report My Yahoo! Genworks Calen... Robot Assembly
Done

Robot Assembly

Update! | Break | TaTu

Model Inputs

Dimensions:

Width	5
Length	2
Height	10

Angles:

Head	0
Body	0
Left Arm	0
Right Arm	0

Grip Opening:

Left	0.45
Right	0.45

OK

Violated Rules

View from Front

Click in Graphics to Recenter and [✓ Zoom In] Zoom Out

Click an Arrow to Pan by 50 Pixels.

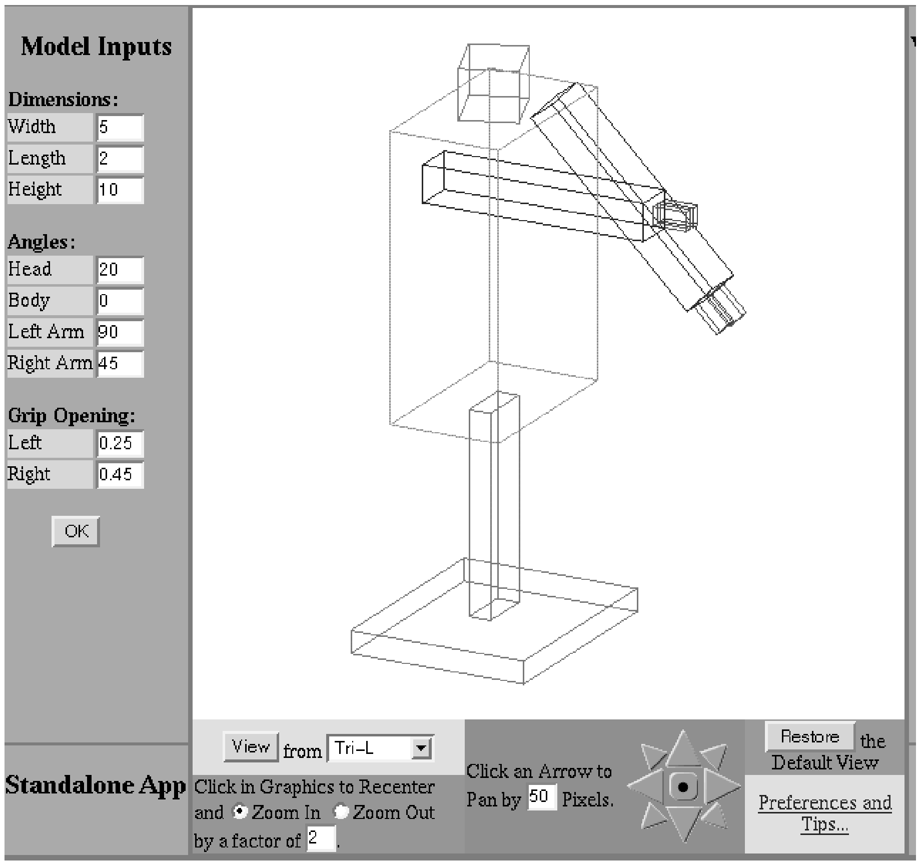
Restore the Default View

Preferences and PDF...

Standalone App

Other Rules

Figure 5.8: Front View of Robot


Model Inputs

Dimensions:

Width	5
Length	2
Height	10

Angles:

Head	20
Body	0
Left Arm	90
Right Arm	45

Grip Opening:

Left	0.25
Right	0.45

OK
Standalone App
View from Tri–L
Click in Graphics to Recenter and [✓ Zoom In] Zoom Out by a factor of 2.
Click an Arrow to Pan by 50 Pixels.
Restore the Default View
Preferences and Tips...

Figure 5.9: Robot with some non-Default Angles

Chapter 6

Example 3: School Bus

In this chapter we leave you with another example of a geometric GDL/GWL application, heavy on code examples, with a bit of explanation sprinkled in between. This School Bus example introduces the use of the `node-mixin` primitive object, which is similar to `application-mixin` which we met in the last chapter. But `node-mixin` is used to *contain* other instances of either `node-mixin` or `application-mixin`, and automatically collects up any `ui-display-list-objects` or rule objects (i.e. objects of type `gwl-rule-object`) from its descendants.

6.1 Toplevel Assembly

Figure 6.1 defines the toplevel assembly consisting of a chassis, body, and interior.

The toplevel mixes in `node-mixin`, and the chassis, body, and interior each mix in `application-mixin`. This results in a high-level user-visible hierarchy, shown as the "Assembly Tree" in the lower-left of Figure 6.3. Note that there is no need to specify a `ui-display-list-objects` slot in the `assembly`. This is because `node-mixin` automatically defines this slot, which appends together the `ui-display-list-objects` from any child objects of appropriate types.

Three toplevel `:settable` computed-slots are also specified, affecting the overall dimensions of the vehicle.

Figure 6.2 defines the `model-inputs` for the toplevel, corresponding to the three `:settable` computed-slots in the `assembly`.

The complete code for the School Bus example is provided on the GDL (and Trial Edition) CD; in this tutorial we provide only the major portion of the interior. The chassis and body are defined similarly, although the chassis in particular contains several interesting examples, which are beyond the scope of this tutorial, of solving somewhat "heavier" engineering problems with GDL/GWL.

6.2 Interior of School Bus

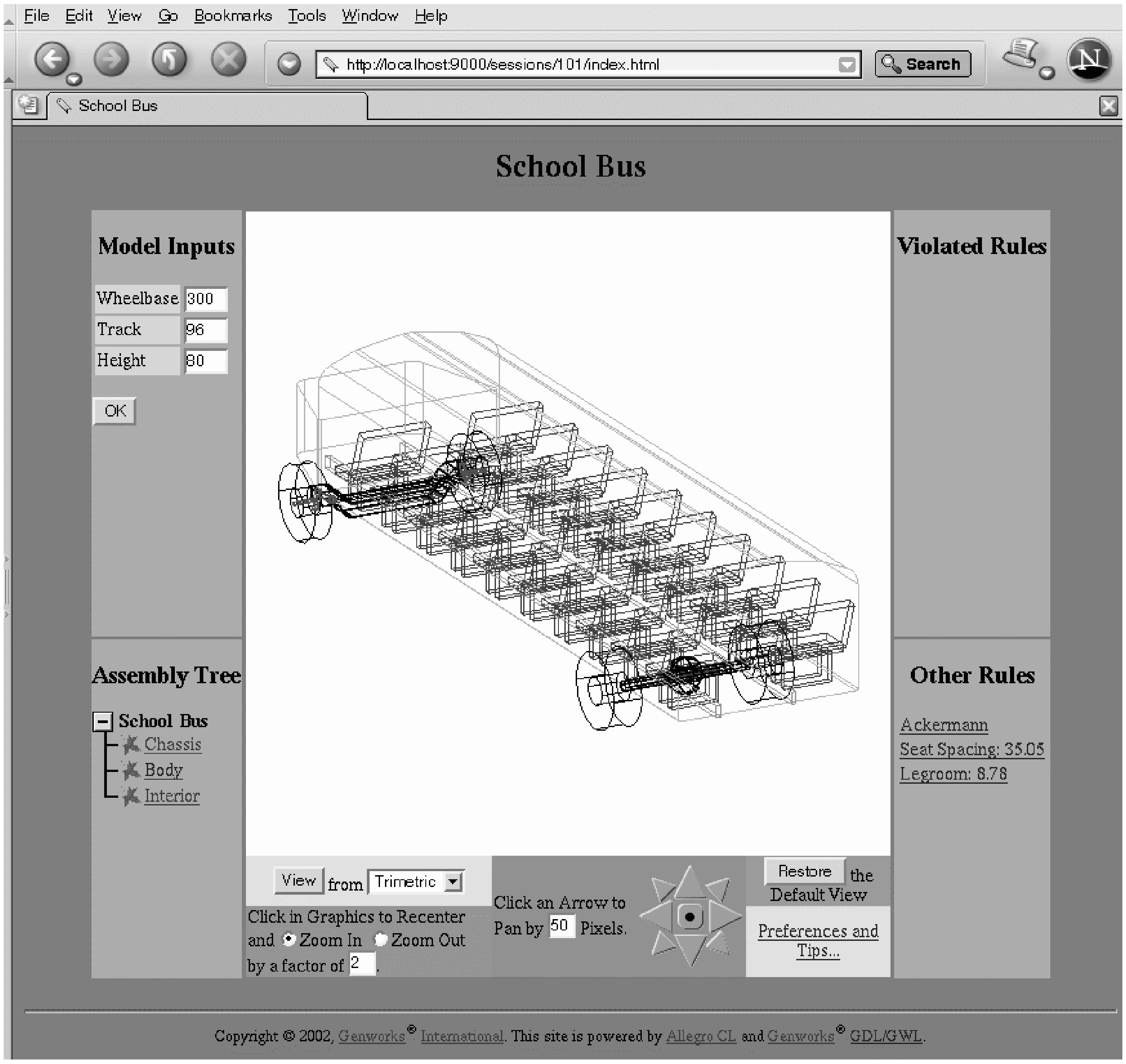
Figures 6.4 and 6.7 define the major components making up the interior of the bus, which for our current purposes consists of the bench seats and a few rules regarding their spacing.

```
(define-object assembly (node-mixin)

  :computed-slots
  ((frame-datum (let ((datum
                       (read-safe-string (string-append
                                          "("
                                          (the frame-datum-m)
                                          ")"))))
                  (translate (the center) :right (first datum) :rear
                             (second datum) :top (third datum))))
   (strings-for-display "School Bus")
   (wheelbase 300 :settable)
   (track 96 :settable)
   (height 80 :settable)
   (frame-datum-m "2000 500 0" :settable))

  :objects
  ((chassis :type 'chassis
            :pass-down (:wheelbase :track)
            :datum (the frame-datum)
            :height 20)
   (body :type 'body
         :pass-down (:wheelbase :track)
         :frame-width (the chassis frame-width)
         :frame-overhang (- (the-child front-overhang)
                            (the chassis front-overhang))
         :firewall-base (translate
                         (the frame-datum) :up
                         (half (the chassis frame-height)) :right
                         (- (the-child cab-width)
                            (the-child frame-overhang))))
   (interior :type 'interior
             :firewall-base (the body firewall-base)
             :width (- (the body width) (the body cab-width))
             :length (the body length)
             :height (the body height))))
```

Figure 6.1: Toplevel Assembly for School Bus

```
(define-view (html-format assembly) nil

  :output-functions
  ((model-inputs
    nil
    (html
     (:p
      (:table
       (:tr ((:td :bgcolor :yellow) "Wheelbase")
            (:td
             ((:input :type :text :size 5 :name :wheelbase :value
                      (the :wheelbase)))))
       (:tr ((:td :bgcolor :yellow) "Track")
            (:td ((:input :type :text :size 5
                          :name :track :value (the :track)))))
       (:tr ((:td :bgcolor :yellow) "Height")
            (:td
             ((:input :type :text :size 5
                      :name :height :value (the :height)))))))
     (:p ((:input :type :submit :name :submit :value " OK ")))))))
```

Figure 6.2: HTML format View of Model Inputs for School Bus Toplevel


File Edit View Go Bookmarks Tools Window Help
http://localhost:9000/sessions/101/index.html
Search
School Bus

School Bus

Model Inputs

Wheelbase	300
Track	96
Height	80

OK

Violated Rules

Assembly Tree

- School Bus
  - Chassis
  - Body
  - Interior

Other Rules

Ackermann
Seat Spacing: 35.05
Legroom: 8.78

View from Trimetric

Click in Graphics to Recenter and [✓ Zoom In] Zoom Out by a factor of 2 .

Click an Arrow to Pan by 50 Pixels.

Restore the Default View

Preferences and Tips...

Copyright © 2002, Genworks® International. This site is powered by Allegro CL and Genworks® GDL/GWL.

Figure 6.3: Toplevel Sheet of School Bus App

In particular, Figure 6.7 defines the `seating-section` which contains the two actual columns of bench seats. This object also contains two *rule objects* which compute certain key pieces of information:

inter-seat-spacing-computation computes the exact spacing from the front of one seat to the front of the next, given the available cabin width (i.e. coach length), the maximumm allowed recline angle of the seat backs, and the number of rows to be fit into the bus. (The actual seat dimensions are taken from defaults in the seat object definition, not listed here).

This spacing value is crucial for two reasons: first, this value is used in order to generate the actual geometric objects that you see in the graphical output. Second, this value is used in the `inter-seat-clearance-check` to compare with the overall length of one seat, to compute how much is left over to be considered "legroom."

inter-seat-clearance-check is a purely diagnostic rule which uses values from the spacing computation rule in order to compute the effective "legroom" between seats. This legroom is compared with the rule's specified `value` (in this case representing the allowed minimum value), to determine whether the rule has violated its condition or not.

It is typical in a KB model to have this kind of tight integration between "rules" and the model itself — when an input to the model is changed, any rules and other objects which directly or indirectly depend on that input will automatically re-evaluate themselves. Rules and objects which are not affected by a given change will avoid the computational work of re-evaluating themselves.

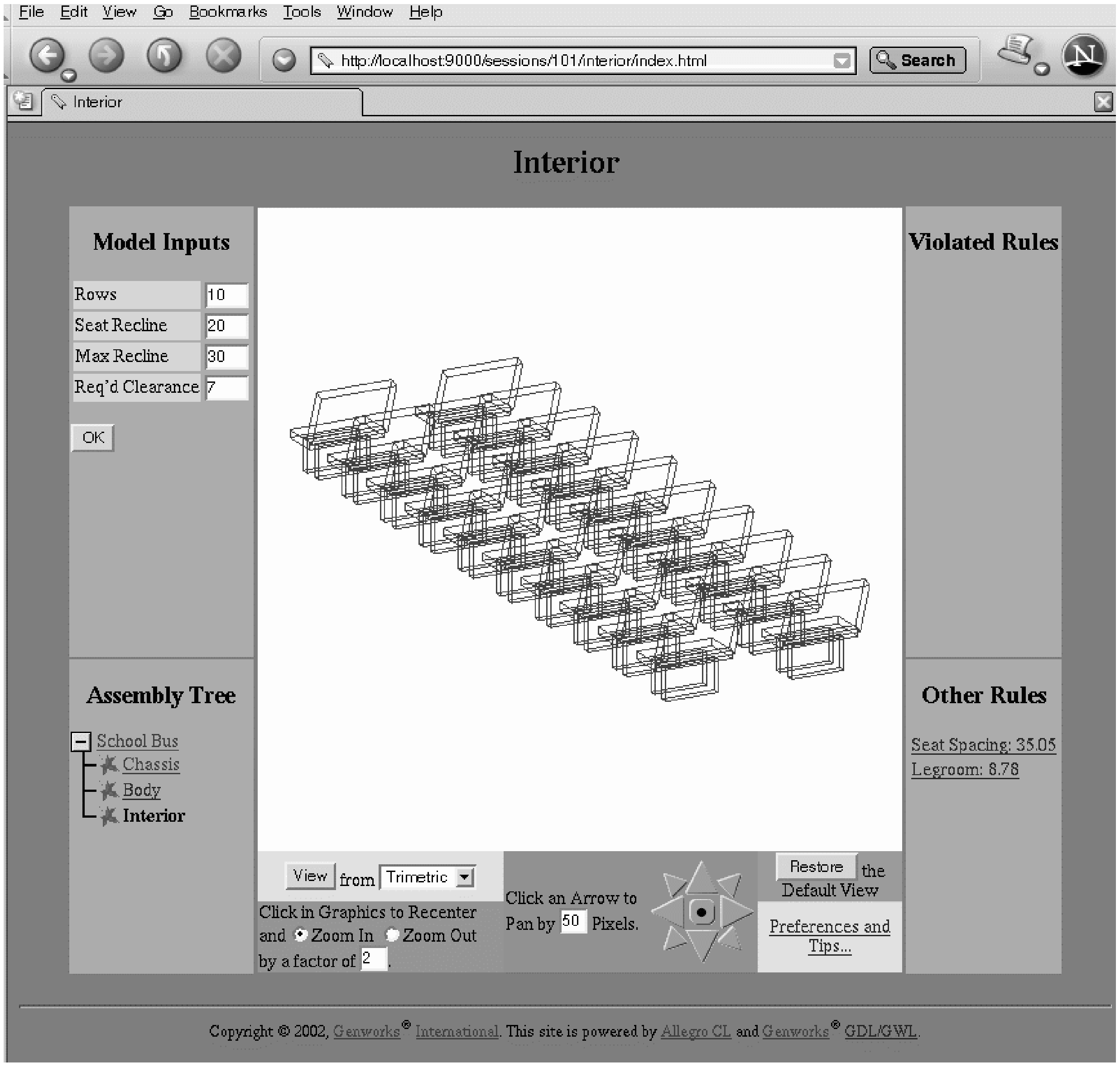
Maintaining this kind of dependency management in a traditional procedural language environment becomes extremely burdensome on the application developer. In a KB environment, however, this dependency management "just happens" as a matter of course.

```
(define-object interior (application-mixin)

  :input-slots
  (firewall-base
   length
   width
   height)

  :computed-slots
  ((ui-display-list-objects (the :sections))
   (number-of-rows 10 :settable)
   (reclined-angle 20 :settable)
   (max-reclined-angle 30 :settable)
   (minimum-inter-seat-clearance 7 :settable))

  :objects
  ((sections :type 'seating-section
             :body-reference-points
             (list :left
                   (translate (the :firewall-base) :front
                              (half (the :length)) :right
                              (the :width))
                   :right
                   (translate (the :firewall-base) :rear
                              (half (the :length)) :right
                              (the :width)))
             :usable-cabin-width (the :width)
             :pass-down (:number-of-rows
                         :reclined-angle :max-reclined-angle
                         :minimum-inter-seat-clearance))))
```

Figure 6.4: Interior Assembly Component School Bus

```
(define-view (html-format interior) nil

  :output-functions
  ((model-inputs
    nil
    (html
     (:p
      (:table
       (:tr
        ((:td :bgcolor :yellow) "Rows")
        (:td
         ((:input :type :text :size 5
                  :name :number-of-rows :value
                  (the :number-of-rows)))))
       (:tr
        ((:td :bgcolor :yellow) "Seat Recline")
        (:td
         ((:input :type :text :size 5
                  :name :reclined-angle :value
                  (the :reclined-angle)))))
       (:tr
        ((:td :bgcolor :yellow) "Max Recline")
        (:td
         ((:input :type :text :size 5
                  :name :max-reclined-angle :value
                  (the :max-reclined-angle)))))
       (:tr
        ((:td :bgcolor :yellow) "Req'd Clearance")
        (:td
         ((:input :type :text :size 5
                  :name :minimum-inter-seat-clearance
                  :value (the :minimum-inter-seat-clearance))))))
     (:p ((:input :type :submit :name :submit :value " OK ")))))))
```

Figure 6.5: HTML Format Model Inputs of School Bus Interior


File Edit View Go Bookmarks Tools Window Help
http://localhost:9000/sessions/101/interior/index.html
Search
Interior

Interior

Model Inputs

Rows	10
Seat Recline	20
Max Recline	30
Req'd Clearance	7

OK

Violated Rules

Assembly Tree

School Bus
Chassis
Body
Interior

Other Rules

Seat Spacing: 35.05
Legroom: 8.78

View from Trimetric

Click in Graphics to Recenter and ● Zoom In ○ Zoom Out by a factor of 2.

Click an Arrow to Pan by 50 Pixels.

Restore the Default View

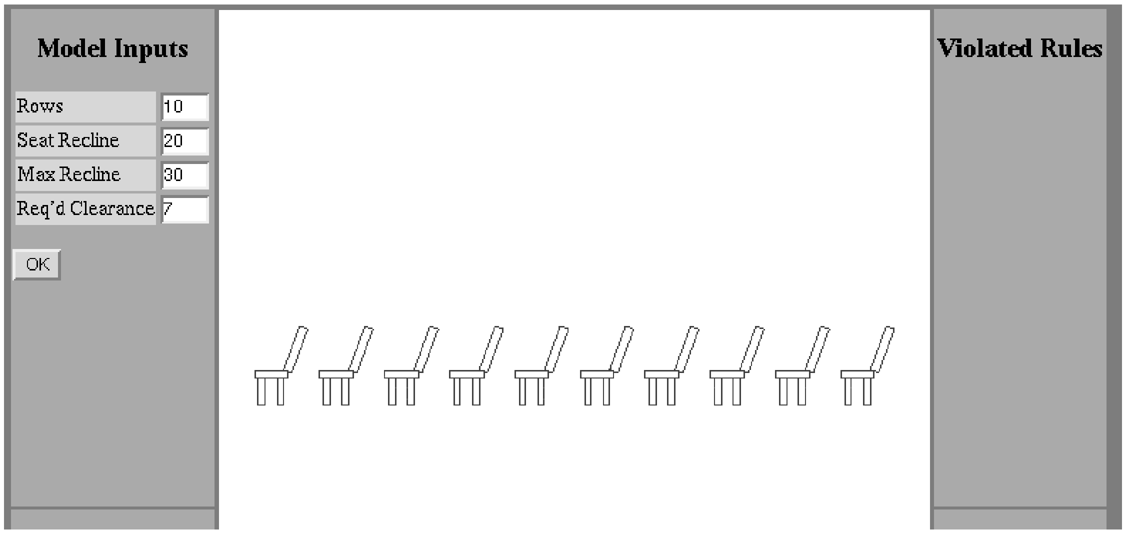
Preferences and Tips...

Copyright © 2002, Genworks® International. This site is powered by Allegro CL and Genworks® GDL/GWL.

Figure 6.6: Interior of School Bus

```
(define-object seating-section (base-object)

  :input-slots
  (fare-class
   usable-cabin-width
   body-reference-points
   max-reclined-angle
   number-of-rows
   minimum-inter-seat-clearance)

  :objects
  ((inter-seat-spacing-computation
    :type 'inter-seat-spacing
    :pass-down (:max-reclined-angle
                :usable-cabin-width :number-of-rows))
   (inter-seat-clearance-check
    :type 'inter-seat-clearance-check
    :inter-seat-spacing (the inter-seat-spacing-computation result)
    :clearance-extent-typical (the inter-seat-spacing-computation
                               clearance-extent-typical)
    :value (the minimum-inter-seat-clearance))
   (sides :type 'seating-side
          :fare-class (the fare-class)
          :sequence (:size 2)
          :side (ecase (the-child index) (0 :left) (1 :right))
          :display-controls (list :color :green)
          :body-reference-point (getf (the body-reference-points)
                                      (the-child side))
          :pass-down (:number-of-rows :reclined-angle)
          :inter-seat-spacing
          (the inter-seat-spacing-computation result)
          :x-max-typical
          (the inter-seat-spacing-computation x-max-typical)
          :x-vector (the (face-normal-vector :right)))))
```

Figure 6.7: Object Definition for Seating Columns



Rows	10
Seat Recline	20
Max Recline	30
Req'd Clearance	7

Figure 6.8: Front View of School Bus Interior

6.3 Causing a Rule Violation

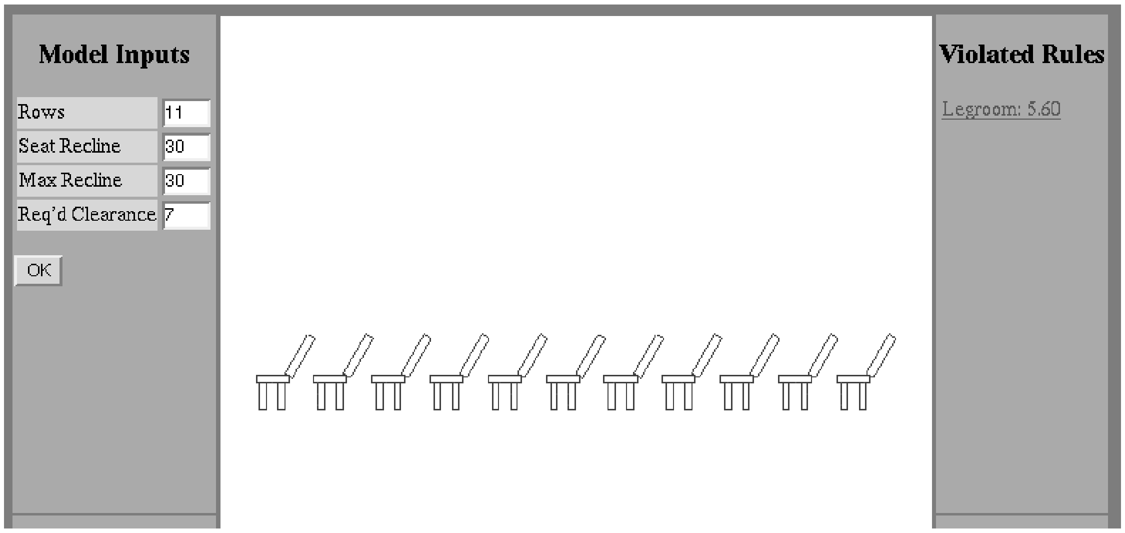
Figure 6.9 shows the state of the interior after a user has changed the number of seating rows to eleven, from the default ten. The user has also changed the displayed recline angle of the seat backs to match the maximum allowed value (30 degrees).

In this state, the seats have redistributed themselves so that they still are spaced evenly in the available length of the coach. However, this has caused a violation in the legroom, defined as the horizontal distance from the rear-center of one seat back to the front of the seat bottom aft of it. The allowed value for this legroom ("Req'd Clearance") is 7, and the current legroom value (as computed by the rule object) is now less than this.

Therefore we have a violation, and the link to the rule shows up in the "Violations" section of the user interface.

For a typical example such as this School Bus, one can imagine dozens or hundreds of other rules. Many such rules can be computed based on information we already have in our model, and others will result in the model being augmented incrementally with new information as needed.

The main point is that we now have a stable, user-friendly, and readily scalable framework in which to represent and grow our "knowledge."



Rows	11
Seat Recline	30
Max Recline	30
Req'd Clearance	7

Legroom: 5.60

Figure 6.9: Interior of School Bus with 11 Rows (Legroom Violation)

Index

:bashee, 34
:center, 34
:indices, 20
:orientation, 34
:respondant, 34
:size, 9, 20

AI, 1
alignment, 37
AllegroServe, 11
application-mixin, 18, 33, 43

base-html-sheet, 12, 34
Basic Lisp Techniques, 2

Caching, 1
Common Lisp, 1
compiled language
  benefits of, 2
computed-slots, 5
containment
  object, 7

declarative, 2
Define-object, 5
dependency management, 47
Dependency tracking, 1
development mode, 40

face-normal-vector, 37
form handling, 13
functions, 5

gwl-make-object, 18
gwl-rule-object, 43
gwl:*developing?*, 40
gwl:*max-id-value*, 13

HTML, 11
htmlgen, 11
HTTP, 11

Ignorance-based Engineering, 1
input-slots, 5
instance-id, 12

Knowledge Base System, 1

macros
  code-expanding, 2
mainstream apps
  KB technology useful for, 29
make-instance, 6
make-object, 6
mixin-list, 5
model-inputs, 33

node-mixin, 43

object sequences, 8
object-orientation
  generic-function, 2
  message-passing, 2
Objects
  sequenced, 8
objects, 5, 7
  child, 7
  contained, 7
  defining, 5
  primitive
    geometric, 33

Paul Graham, 19
publishing
  of GWL URIs, 18

read-safe-string, 16
recomputation

on-demand, 22
reference chains, 7
requirements
ever-expanding, 29
return-object, 13
rules, 47
diagnostic, 47
generative, 47
model
tight integration with, 47
violating, 47

self, 6
sequence, 20
sequences, 8
specification-plist, 5
spreadsheet
KB as distinct from, 22

TaTu, 40
the, 6
the-object, 6
time
using to understand KB dynamics, 25

ui-display-list-objects, 18, 43

view, 33

web user interface
creating, 11
write-back-link, 13
write-html-sheet, 12