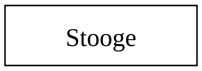
Module One Object-Oriented Programming

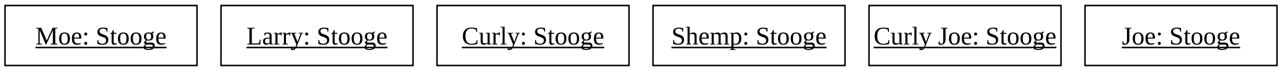
Classes and Objects

Classes are blueprints, objects are constructs. The class definition defines what you can do to the object, and what information you can get from it. All the objects that are instances of a class can be treated identically. They are interchangeable entities. If you’ve written an object-oriented program that can add a text caption to an instance of a class that represents a PNG image, it doesn’t matter it you give it a picture of Richard Nixon or a pomegranate.

In UML, a class is represented by a box. This class represents a stooge, who is a member of the comedy group "The Three Stooges":



Objects, which are instances of the class, are also represented by boxes, but the name is underlined, and a colon separates the object name from the class name. Here are some Stooges (the third stooge position was filled by different performers over time):

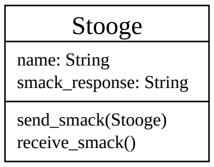


Attributes and Operations

The class definition describes the attributes (data) and methods (behavior) that are part of the class.

The stooges often smacked each other, although Moe did most of the smacking. This smacking is part of stooge behavior, so we’ll represent it with a method called send_smack(Stooge). The Stooge in parentheses indicates that the send_smack() method accepts one argument, which is the stooge that is to be smacked. For example, Moe.smack(Curly) indicates that you want Moe to smack Curly.

Just as a stooge needs to know when you want him to smack another stooge, he also needs to know when he’s been smacked. So, we’ll also have the receive_smack() method, which you can think of as the stooge’s reaction to being smacked. Each stooge has a unique reaction to being smacked, so we need an attribute, smack_response. In addition to the smack_response attribute, each stooge will naturally have a name.



Stooge.java

```
public class Stooge {

  // This stooge’s name.
  String name;

  // Response to being smacked.
  String smack_response;

  // Constructor.
  //
  public Stooge(String stooge_name) {
      name = stooge_name;
  }

  // Set this stooge’s response.
  //
  void setResponse(String new_response) {
      smack_response = new_response;
  }

  // Send a smack to another stooge.
  //
  void sendSmack(Stooge target) {
      target.receiveSmack();
  }

  // Receive a smack from another stooge.
  //
  void receiveSmack() {
      System.out.println(
        name + ": " + smack_response);
  }

}
```

Stooge.pm

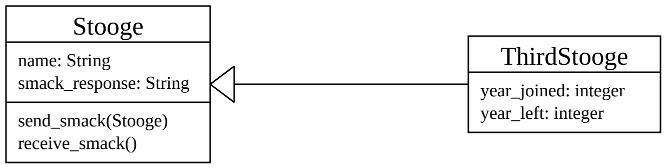
```
package Stooge;

# The constructor.
#
sub new {
  my ($proto, $name) = @_;
  my $class = ref($proto) || $proto;
  my $self = {};
  # This stooge’s name.
  $self->{name} = $name;
  # Response to being smacked.
  $self->{smack_response} = "";
  bless($self, $class);
  return $self;
}

# Set this stooge’s response.
#
sub set_response {
  my $self = shift;
  $self->{smack_response} = shift;
}

# Smack another stooge.
#
sub send_smack {
  my ($self, $target) = @_;
  $target->receive_smack();
}

# Be smacked by another stooge.
#
sub receive_smack {
  my ($self) = @_;
  print "$self->{name}: $self->{smack_response}\n";
}
1;
```

Stooge.py

```
class Stooge:

    # The constructor.
    #
    def __init__(self, name):
        self.name = name
        self.smack_response = ’’

    # Set this stooge’s response.
    #
    def setResponse(self, response):
        self.smack_response = response

    # Send a smack to another stooge.
    #
    def sendSmack(self, stooge):
        stooge.receiveSmack()

    # Receive a smack from another stooge.
    #
    def receiveSmack(self):
        print "%s: %s" % (self.name,
                          self.smack_response)
```

Generalization

At any given time, there were always three stooges. Of the three, Larry and Moe were always present. To differentiate between types of stooge, we can define a class that is identical to Stooge with the exception of two extra attributes that indicate the years he joined and left the stooges. This class, ThirdStooge, is a subclass of Stooge. Everything that you can say about a Stooge can also be said about a ThirdStooge. **Stooge is a *generalization* of ThirdStooge.**



By the same logic, ThirdStooge is a *specialization* of Stooge. Generalization is often implemented using a feature called inheritance. Using inheritance, a subclass inherits all the methods and attributes from its parent class, or superclass. The subclass can, as with ThirdStooge, define additional features, such as methods or attributes.

Here are the ThirdStooge subclasses of Stooge in Java, Perl, and Python:

ThirdStooge.java

```
public class ThirdStooge extends Stooge {

  int year_joined; // year he joined the stooges.
  int year_left;   // year he left the stooges.

  // Invoke the superclass constructor.
  //
  public ThirdStooge(String stooge_name) {
      super(stooge_name);
  }

  void setYearJoined(int year) {
      year_joined = year;
  }

  int getYearJoined() {
      return year_joined;
  }

  void setYearLeft(int year) {
      year_left = year;
  }

  int getYearLeft() {
      return year_left;
  }

}
```

ThirdStooge.py

```
# import the Stooge class from Stooge.py
from Stooge import Stooge

class ThirdStooge(Stooge):

    # The constructor.
    #
    def __init__(self, name):

        # Invoke superclass constructor.
        Stooge.__init__(self, name)

        # Initialize year values.
        self.year_joined = 0
        self.year_left   = 0

    def setYearJoined(self, year):
        self.year_joined = year

    def getYearJoined(self):
        return self.year_joined

    def setYearLeft(self, year):
        self.year_left = year

    def getYearLeft(self):
        return self.year_left
```

ThirdStooge.pm

```
package ThirdStooge;
use Stooge;
@ISA = qw(Stooge);

# The constructor.
#
sub new {
  my $proto = shift;
  my $class = ref($proto) || $proto;

  # Invoke the superclass constructor.
  #
  my $self = $class->SUPER::new(@_);

  # Additional properties.
  #
  $self->{year_joined} = "";
  $self->{year_left} = "";

  bless($self, $class);
  return $self;
}

sub set_year_joined {
    my $self = shift;
    $self->{year_joined} = shift;
}
sub get_year_joined {
    my $self = shift;
    return $self->{year_joined};
}
sub set_year_left {
    my $self = shift;
    $self->{year_left} = shift;
}
sub get_year_left {
    my $self = shift;
    return $self->{year_left};
}
1;
```

Polymorphism

Generalization lets you substitute a ThirdStooge anywhere a Stooge is needed. For example, Moe can smack Larry because Larry is a Stooge. Moe can also smack Shemp, Curly, Curly Joe, or Joe, because a ThirdStooge can be used anywhere a Stooge is needed. This property of replaceability is known as polymorphism. The reverse is not true: if you tried to find out what year a Stooge left or joined the Three Stooges, you would be at a loss. Only a ThirdStooge has that sort of information.

polyStooge.java

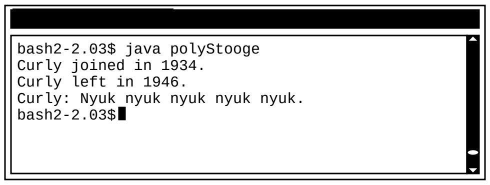
```
public class polyStooge {

  public static void main(String argv[]) {

    // Create Moe and Curly
    //
    Stooge moe = new Stooge("Moe");
    ThirdStooge curly = new ThirdStooge("Curly");
    curly.setResponse("Nyuk nyuk nyuk nyuk nyuk.");
    curly.setYearJoined(1934);
    curly.setYearLeft(1946);

    // Get info about curly.
    //
    System.out.println("Curly joined in " +
                       curly.getYearJoined() + ".");
    System.out.println("Curly left in " +
                       curly.getYearLeft() + ".");

    // Moe smacks Curly.
    //
    moe.sendSmack(curly);
  }
}
```

polyStooge.pl

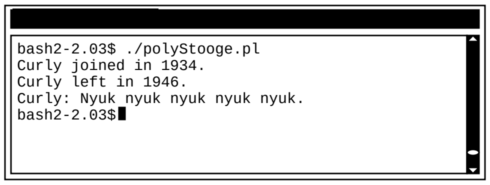
```
#!/usr/bin/perl -w

# Import classes from Stooge.pm and ThirdStooge.pm.
#
use Stooge;
use ThirdStooge;

# Create Moe and Curly
#
$moe = new Stooge("Moe");
$curly = new ThirdStooge("Curly");
$curly->set_response("Nyuk nyuk nyuk nyuk nyuk.");
$curly->set_year_joined(1934);
$curly->set_year_left(1946);

# Info about curly.
#
print "Curly joined in ",
    $curly->get_year_joined(), ".\n";
print "Curly left in ",
    $curly->get_year_left(), ".\n";

$moe->send_smack($curly); # Moe smacks Curly.
```

polyStooge.pl

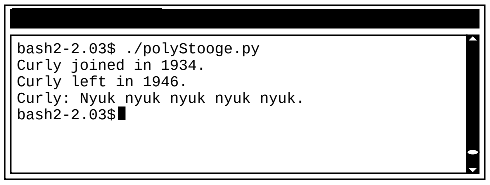
```
#!/usr/bin/python

# import classes from Stooge.py and ThirdStooge.py.
#
from Stooge import Stooge
from ThirdStooge import ThirdStooge

# Create Moe and Curly.
#
moe = Stooge("Moe")
curly = ThirdStooge("Curly")
curly.setResponse("Nyuk nyuk nyuk nyuk nyuk.")
curly.setYearJoined(1934);
curly.setYearLeft(1946);

# Info about curly.
#
print "Curly joined in %d." % curly.getYearJoined()
print "Curly left in %d." % curly.getYearLeft()

moe.sendSmack(curly) # Moe smacks Curly.
```



```
bash2-2.03$ java polyStooge
Curly joined in 1934.
Curly left in 1946.
Curly: Nyuk nyuk nyuk nyuk nyuk.
bash2-2.03$
```



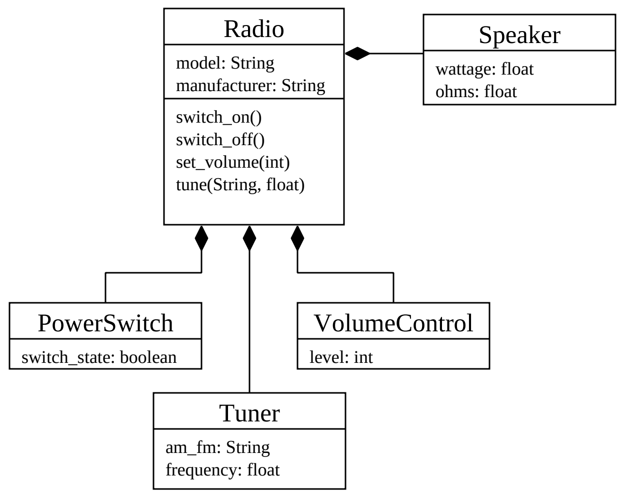
```
bash2-2.03$ ./polyStooge.pl
Curly joined in 1934.
Curly left in 1946.
Curly: Nyuk nyuk nyuk nyuk nyuk.
bash2-2.03$
```



```
bash2-2.03$ ./polyStooge.py
Curly joined in 1934.
Curly left in 1946.
Curly: Nyuk nyuk nyuk nyuk nyuk.
bash2-2.03$
```

Composition

When you need to model a part-whole relationship, you can use composition to indicate that an instance of one class is part of an instance of another class. Leaving the Stooges behind, let’s look at an example of composition. In this example, we see a Radio class that is composed of a few essential parts: a Speaker, Tuner, Volume Control, and Power Switch.



Radio.pm

```
use Class::Template;

# Use Class::Template to create definitions for
# the PowerSwitch, Tuner, VolumeControl, and
# Speaker classes.
#
struct(
    PowerSwitch => {
        switch_state => ’$’,
    }
);
struct(
    Tuner => {
        am_fm     => ’$’,
        frequency => ’$’,
    }
);
struct(
    VolumeControl => {
        level => ’$’,
    }
);
struct(
    Speaker => {
        wattage => ’$’,
        ohms    => ’$’,
    }
);

# The Radio Class.
#
package Radio;

sub new {
    # Standard Perl OO setup.
    #
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {};
    bless ($self, $class);

    # Create the PowerSwitch, Tuner, VolumeControl,
    # and Speaker.
    #
    $self->{power_switch} = new PowerSwitch();
    $self->{tuner} = new Tuner();
    $self->{volume_control} = new VolumeControl();
    $self->{speaker} = new Speaker();

    # Set reasonable defaults for the Speaker.
    #
    $self->{speaker}->wattage(3.75);
    $self->{speaker}->ohms(8);
    return $self;
}

# Turn the radio on.
#
sub switch_on {
    my $self = shift;
    $self->{power_switch}->switch_state(1);
}

# Turn the radio off.
#
sub switch_off {
    my $self = shift;
    $self->{power_switch}->switch_state(0);
}

# Set the volume.
#
sub set_volume {
    my $self = shift;
    my $level = shift;
    $self->{volume_control}->level($level);
}

# Tune the radio.
#
sub tune {
    my $self = shift;
    my ($frequency, $band) = @_;
    $self->{tuner}->am_fm($band);
    $self->{tuner}->frequency($frequency);
}
1;
```

Radio.java

```
// Define PowerSwitch, Tuner, VolumeControl,
// and Speaker.
//
class PowerSwitch {
  boolean switch_state = false;
}

class Tuner {
  String am_fm;
  double frequency;
}

class VolumeControl {
  int level;
}

class Speaker {
  double wattage;
  double ohms;
}

// The Radio class.
//
public class Radio {

  PowerSwitch power_switch;
  Tuner tuner;
  VolumeControl volume_control;
  Speaker speaker;

  public Radio() {

      // Create the PowerSwitch, Tuner,
      // VolumeControl, and Speaker.
      //
      power_switch = new PowerSwitch();
      tuner = new Tuner();
      volume_control = new VolumeControl();
      speaker = new Speaker();

      // Set reasonable defaults for the Speaker.
      //
      speaker.wattage = 3.75;
      speaker.ohms = 8;
  }

  // Turn the radio on.
  //
  public void switchOn() {
      power_switch.switch_state = true;
  }

  // Turn the radio off.
  //
  public void switchOff() {
      power_switch.switch_state = false;
  }

  // Set the volume.
  //
  public void setVolume(int level) {
      volume_control.level = level;
  }

  // Tune the radio.
  //
  public void tune(double frequency, String band) {
      tuner.am_fm = band;
      tuner.frequency = frequency;
  }
}
```

Radio.py

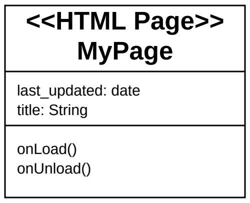
```
# Define PowerSwitch, Tuner, VolumeControl,
# and Speaker.
#
class PowerSwitch:
  def __init__(self):
      self.switch_state = 0

class Tuner:
  def __init__(self):
      self.am_fm = ’’
      self.frequency = 0

class VolumeControl:
  def __init__(self):
      self.level = 0

class Speaker:
  def __init__(self):
      self.wattage = 0
      self.ohms = 0

# The Radio class.
#
class Radio:

  def __init__(self):

      # Create the PowerSwitch, Tuner,
      # VolumeControl, and Speaker.
      #
      self.power_switch = PowerSwitch()
      self.tuner = Tuner()
      self.volume_control = VolumeControl()
      self.speaker = Speaker()

      # Set reasonable defaults for the speaker.
      #
      self.speaker.wattage = 3.75
      self.speaker.ohms = 8

  # Turn the radio on.
  #
  def switchOn(self):
      self.power_switch.switch_state = 1;

  # Turn off the radio.
  #
  def switchOff(self):
      self.power_switch.switch_state = 0;

  # Set the volume.
  #
  def setVolume(self, level):
      self.volume_control.level = level

  # Tune the radio.
  #
  def tune(self, frequency, band):
      self.tuner.am_fm = band
      self.tuner.frequency = frequency
```

UML In Web Apps

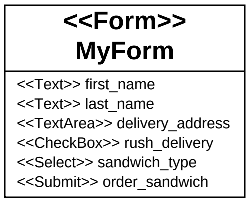
UML was originally developed to facilitate object-oriented design, but it has been adapted to many other areas.

Let's look at how you can use UML in web applications. Much of the notation we'll use here was directly inspired by Jim Conallen's Modeling Web Application Architectures with UML (Communications of the ACM, October 199, Vol. 42. No. 10).

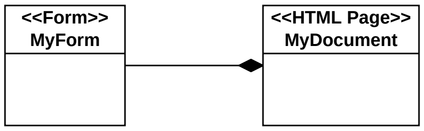
You'll find that web applications aren't always made of truly object-oriented components. For example, a CGI script is usually a small program with a minimal amount of branching and decisions. While a CGI script may make use of objects (as we'll see in some examples), it's usually just a simple program. But, it needs to be modeled, so we'll model it using the UML class notation. Here is a gallery of some of the notations we'll be using:



An HTML page is represented in this document with the <<HTML Page>> stereotype. Important attributes that are displayed on the page may be visible, as well as JavaScript objects or methods.


<<Form>>
MyForm
<<Text>> first_name
<<Text>> last_name
<<TextArea>> delivery_address
<<CheckBox>> rush_delivery
<<Select>> sandwich_type
<<Submit>> order_sandwich

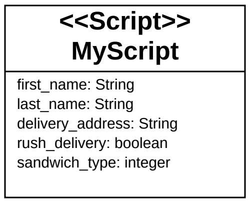
A form is represented in this document with the <<Form>> stereotype. An HTML page can contain one or more forms, which users can use to submit information to a CGI script. Forms are always enclosed in an HTML page. You can express this relationship with composition:



Unless we are developing an HTML page of some significance, we'll omit it in some of our examples. So, if you see a form that isn't part of a page, we'll assume that it needs a minimal HTML document around it.

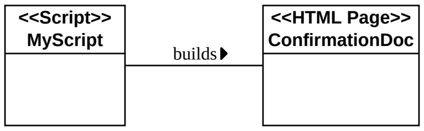
Each form includes various HTML elements, which are shown as attributes of the form. We use the UML stereotype notation to indicate a particular HTML element.

HTML forms generally don't have any methods: the action of the <<Submit>> button is represented as an association between the form and its target script.

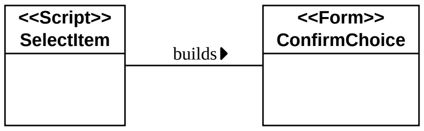


We use the <<Script>> stereotype to represent a server-side script, such as a CGI script or servlet. You can represent arguments to the script as properties, and you may find it useful to include internal methods that the script uses.

A script can be instantiated when a user submits a form, follows a link, or types the script's URL into their browser's location bar. You'll always see a script chained to its output, which is usually an HTML document (although it can be another content type, such as a GIF image).



In cases where a script generates a simple form, we'll chain it directly to the form object, which implies that there is a minimal HTML document when none is specified:



Module Two Patterns

Patterns were first defined in Christopher Alexander's A Pattern Language (Oxford University Press, 1977), as solutions to recurring architectural problems. In the 1980s, Kent Beck and Ward Cunningham applied Alexander's patterns to object-oriented design. Patterns were highly popularized in the book Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, Helm, Johnson, and Vlissides, Addison Wesley-Longman, 1995). Patterns pass on knowledge from experts to other practitioners: they distill experience into a form that can be easily applied to new manifestations of familiar problems.

We'll look at some design patterns (patterns that are applied during the design process), and later we'll apply them to the design of an example system.

Design patterns are not algorithms, reusable code, or data structures. They are descriptions of objects and collaborations that you can apply in a specific context.

Elements of a Pattern

There are various formats for patterns, and we use a very simple one in this tutorial:

Name:
This is a descriptive name that should communicate a lot about the pattern. The idea is to increase your vocabulary with words that communicate complex concepts quickly.

Problem:
This describes the context in which the pattern should be applied. Often, an example is used to illustrate the problem.

Solution:
The solution describes the objects, relationships, responsibilities, and collaborations that are part of the pattern. It is essentially an abstract template that can be applied to the problem in context. Superimpose the solution on your problem to light the way toward an implementation.

Example Pattern: Observer

Let's look at the Observer pattern from Design Patterns (Gamma).

Problem: Suppose you are building a system to handle auctions. In any given auction, you have one object to represent the Item, and any number of objects to represent the Bidders. When the price of the auction changes, how do you notify each Bidder of the new price without coupling the Bidders too closely to the Items?

Solution: The Observer pattern can be applied in a situation such as this. This pattern proposes a collaboration where changes in one object (the subject) cause notifications to be sent to other objects (the observers).
A typical way to implement Observer is to define a Subject and Observer class, and let your implementations inherit from those classes.

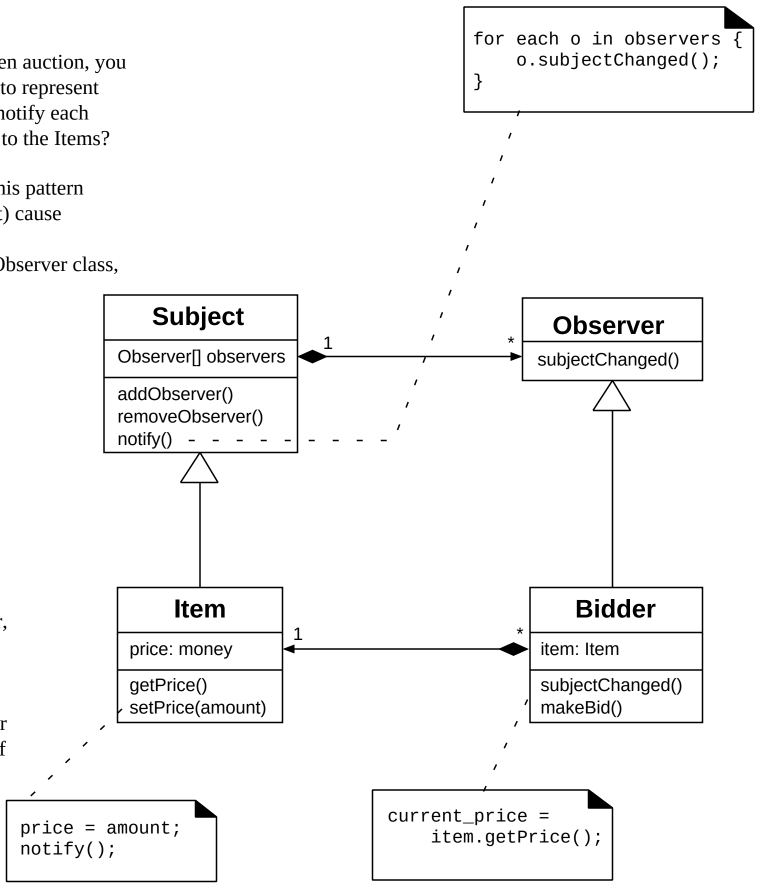
Subject: The Subject class should define the following methods:
AddObserver(Observer)
Attaches an observer.

DropObserver(Observer)
Removes an observer.

notify()
For each observer, send a subjectChanged() message.

Observer: The Observer class only needs a ***subjectChanged()*** method. One way to implement this is to define a class, such as Bidder, as a subclass of Observer.

It's up to you to define how the Observer will respond to the subjectChanged() message. More likely than not, the Observer will invoke one of the subject's methods to obtain the value of any attributes that may have changed (such as the price of an item in an auction).



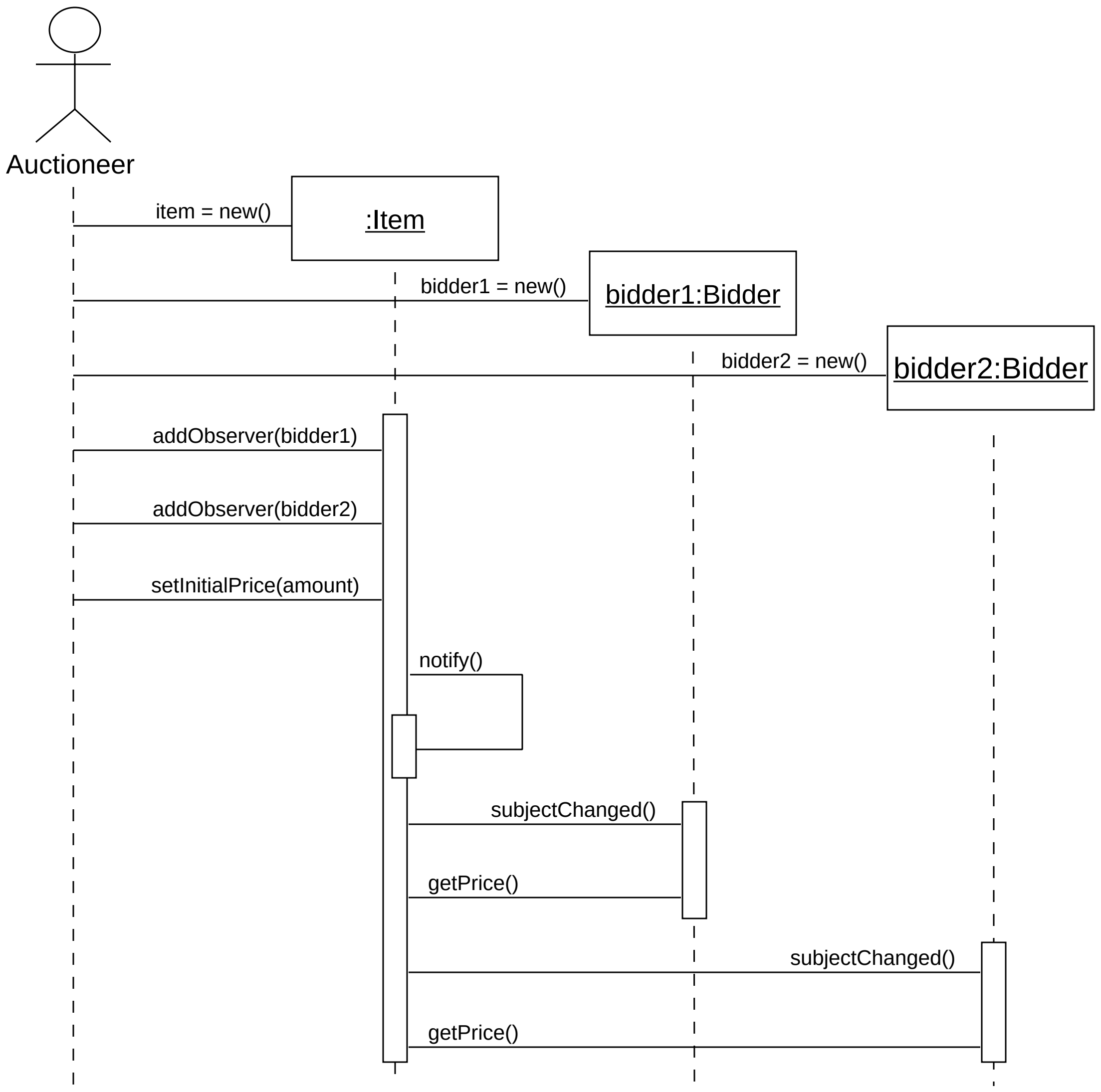
```
for each o in observers {
    o.subjectChanged();
}
```

```
price = amount;
notify();
```

```
current_price =
    item.getPrice();
```

Example class diagram for Observer.

Let’s look at another type of UML diagram, the system sequence diagram, to see how Observer would work in practice:



Module Three
OO Analysis and Design

Use Cases

Use cases are formal descriptions of scenarios that take your system through a process from start to finish. For example, here is a use case that describes how a user might shop at an online store:

Use Case: Select a Product

Summary: Our customer visits the store and selects a product category, after which she is presented with a list of products. Next, she chooses a product and quantity, which is added to her shopping cart. After this, she can either keep shopping, or proceed to the checkout.

User Action	System Response
1. Select a product category.	**2.** Display a descriptive list of products and their price within the selected category.
3. Choose a product and quantity.	**4.** Put the product and quantity in the shopping cart.
	5. Display the contents of the shopping cart with prices and total cost. Give the customer the opportunity to shop some more or check out.

Use cases are a logical starting point for building a system. They can be used to state or expand upon the requirements for the system. Once you have a use case, the tricky part is to get from the Use Case to something that specifies the classes you'll use in your application.

Bookshelves are brimming with texts that tell you how to design a system using UML and use cases. Let's look at a simple approach that takes us from use cases to class diagrams.

Identifying Concepts

The first thing we'll do is try to identify concepts that are part of the use case. Craig Larman's Applying UML and Patterns includes some excellent techniques for discovering this sort of information. However, we'll use one of the simplest techniques: noun phrase identification.

To use this technique, simply highlight the noun phrases in the use case:

User Action	System Response
1. Select a product category.	**2.** Display a descriptive list of products and their price within the selected category.
3. Choose a product and quantity.	**4.** Put the product and quantity in the shopping cart.
	5. Display the contents of the shopping cart with prices and total cost. Give the customer the opportunity to shop some more or check out.

Once you have this list, you need to decide which nouns are first-class concepts in the use case, and which are merely attributes. There might be some unstated concepts lurking there, so you need to strike a balance between being a detective and an inventory: find what lurks below the surface, but don't invent stuff. Let's be cautious at first, and only model things that are in the list. Here are all the candidate concepts:

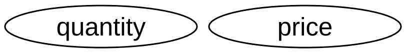
Noun Phrase	Concept Name
product category, selected category	ProductCategory
descriptive list of products	ProductList
product	Product
quantity	Quantity
price, prices, total cost	Price
shopping cart, contents of the shopping cart	ShoppingCart
customer	Customer

Identify Attributes

Now, let’s try to find the attributes.

Step One: Primitive Data Types

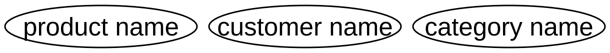
Primitive data types, such as strings or numbers are almost always attributes. So, let’s categorize quantity and price as attributes:



Which concepts do these attributes belong to? It seems that price belongs to a product, since it’s the price for that product, but what about quantity? Let’s revisit this in a few paragraphs...

Step Two: Names

Many things need names, even though it’s not explicitly specified. At the very least, we’ll need name attributes for product categories, products and customers:



Step Three: Implied Attributes

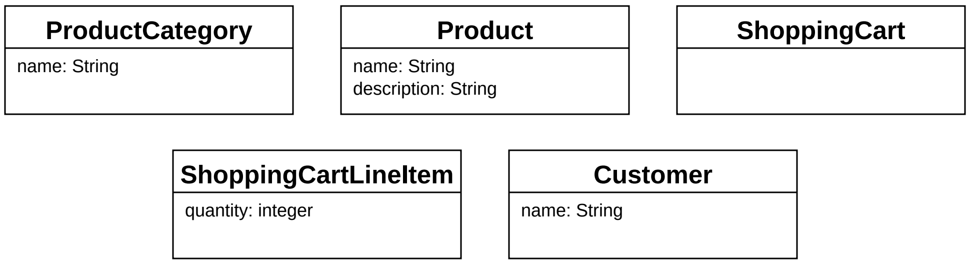
Most of the noun phrases are very precise, and name only one thing. However, there are two that we can’t be so sure of. The first is "descriptive list of products." What does this mean? Unfortunately, the use case is not specific about this, so in the real world, you’d need to ask your user community. Let’s pretend they told us that the descriptive list is the name of the product, its price, and a short description. This is no problem, we only need one more attribute:

Diagram the Concepts

The second noun phrase that seems to imply more than one thing is "contents of the shopping cart." At first guess, we might assume that a shopping cart contains a product, and this is partially correct. If we were to go with that assumption, we’d need to have quantity as an attribute of product. This has the drawback of coupling the product to the shopping cart. If there is any way to keep the product unaware of the shopping cart, the system will be more extensible. So, I suggest an intersection between the shopping cart and the product, a shopping cart line item. However, this isn’t really an attribute, but a concept that has attributes of its own. So, we’ll add it to the list of concepts (shown here without attributes):

Noun Phrase	**Concept Name**
product category, selected category	ProductCategory
product, descriptive list of products	Product
shopping cart	ShoppingCart
contents of the shopping cart	ShoppingCartLineItem
customer	Customer

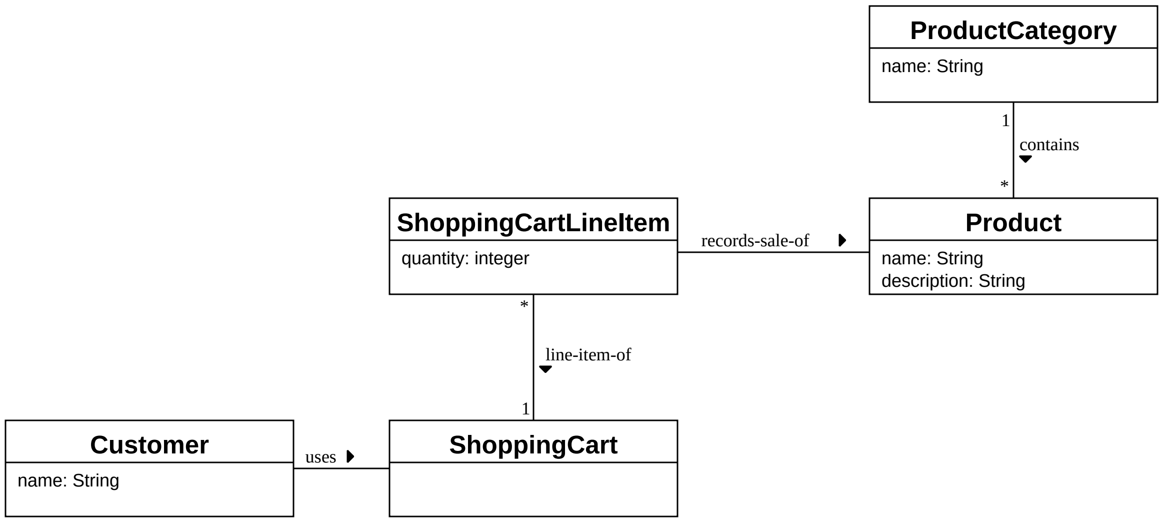
Next, let’s draw out class figures for the concepts, and include their attributes. Our diagram is starting to resemble the class diagrams we looked at in the previous module.



Identify Relationships

Next, we need to figure out how all of these are related. You can start with a list of relationship categories, and attempt to find relationships that fit into those categories. This list below is similar to one found in Larman:

Category	**Relationships**
A contains B	ProductCategory contains Products
A is a line item of B	ShoppingCartLineItem is a line item in ShoppingCart
A knows or records B	ShoppingCartLineItem records sale of Product
A communicates with B	(none found)
A uses or owns B	Customer uses/owns ShoppingCart
A communicates with B	(none found)



Module Four
User Interface Development

Designing the User Interface

To develop the user interface, we need to revisit the use case, and decide how to satisfy its requirements. Because of the peculiarities of web applications, each user action that is sent to the server will run a script on the server. Each script will, in turn, result in a new page or form being generated.

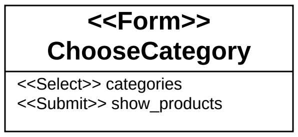
Note that this is the most general case: in many instances, you can use client-side scripting to eliminate round-trips to the server. However, the following example assumes that all user requests and system responses involve communications between the browser and server.

Let's look at each of the steps in the use case, and consider what we need to do to satisfy the requirements in each step.

Use Case: Select a Product

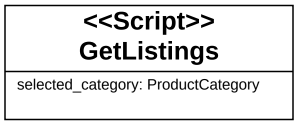
Summary: Our customer visits the store and selects a product category, after which she is presented with a list of products. Next, she chooses a product and quantity, which is added to her shopping cart. After this, she can either keep shopping, or proceed to the checkout.

User Action	**System Response**
1. Select a product category.	**2.** Display a descriptive list of products and their price within the selected category.
3. Choose a product and quantity.	**4.** Put the product and quantity in the shopping cart.
	5. Display the contents of the shopping cart with prices and total cost. Give the customer the opportunity to shop some more or check out.



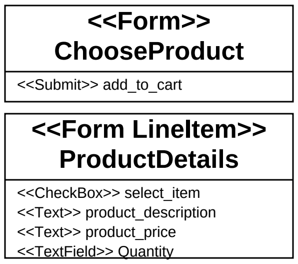
Step 1
Summary: A User Action that requires a client-side form,

We need a form that lets the user select a product category. Let's call this form ChooseCategory, and give it a popup menu to choose a category (a <SELECT> element) and a submit button to show the products.



Step 2
Summary: A System Response that requires a script.

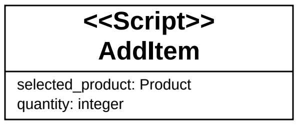
We need a script to respond to the user's action. Let's call this script GetListings. It only has one attribute, the category that we selected in ChooseCategory.



Step 3
Summary: A User Action that requires a client-side form.

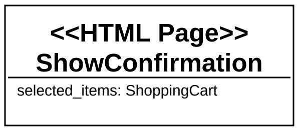
This step requires a form so the user can specify a product and quantity. It must be built by the script in the previous step. We'll call this form ChooseProduct.

The form consists of multiple product descriptions, so we'll need one or more line items that includes: a product description, price, a text field to specify the quantity of the product, and a checkbox to indicate that the item should be added to the shopping cart. Each line item appears in ProductDetails, which is a line item of ChooseProduct.



Step 4
Summary:A System Response that requires a script.

We need a script to respond to the user's action. This script, AddItem, has two attributes: selected_product, and quantity.

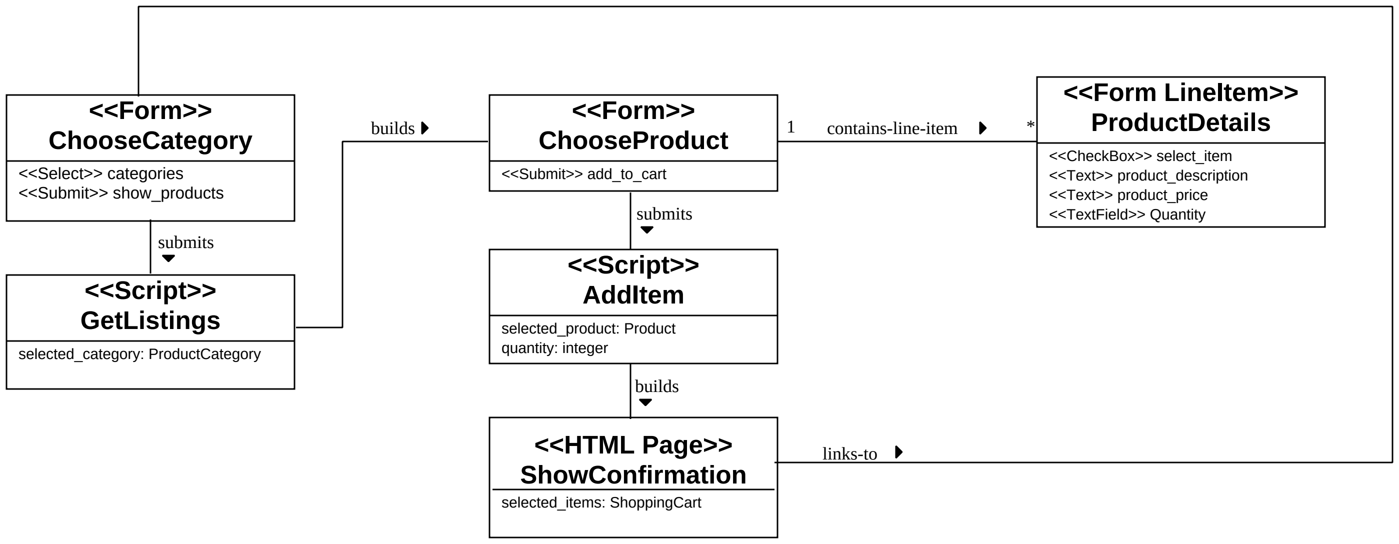


Step 5
Summary: Information that is presented to the user.

After the user adds an item to their cart, we need to show them a confirmation page to let them know the item made it into the cart. This page lets the user go back to shopping or check out and pay for his or her purchases, which implies that it has a link to the entry point of this use case and the Check Out use case (which we haven't seen yet). This page should have been generated by the script in the previous step.

Assemble the Pieces

Now, let's put all the pieces together. We need to make sure that the confirmation page links back to the entry point (ChooseCategory).



Look for More Relationships

How do we hook this up to the conceptual model we developed earlier? We need to look for more relationships, as we did in the previous module. Let's take a first pass at this. Think through each step in the use case and consider how the user interface objects relate to the items in the conceptual model.

Category	Relationships
A contains B	ChooseCategory contains ProductCategory GetListings contains ProductCategory ProductDetails contains Product AddItem contains Product ShowConfirmation contains ShoppingCart
A uses or owns B	AddItem puts a Product in ShoppingCart *

Forms, Pages, and Relationships

In general, forms are pretty dumb - the scripts that build them have access to system objects, but the forms that are built can't interact with system objects, unless you develop some client-side processing that makes round-trips to the server. As a rule, a form won't have a relationship with anything other than its builder and its target. So, if you see a page or a form that appears to need knowledge of a concept, you need to move that knowledge into the form's builder.

What about the ChooseCategory form? As it stands right now, it's just a form, and it's not built by a script. By the rule just stated above, it can't have knowledge of the concepts, so, we need to add a new script, the CategoryBuilder, which builds the ChooseCategory form.

We'll move relationships according to the following rules:

```
ChooseCategory -> CategoryBuilder
ChooseProduct -> GetListings
ProductDetails -> GetListings
ShowConfirmation -> AddItem
```

Category	Relationships
A contains B	CategoryBuilder contains ProductCategory GetListings contains ProductCategory GetListings contains Product AddItem contains Product AddItem contains ShoppingCart
A uses or owns B	AddItem puts a Product in ShoppingCart

Don't Restate Derived Relationships

You can avoid a lot of clutter and conceptual confusion if you eliminate relationships that can be derived. For example, GetListings has relationships with ProductCategory and Product. Because a ProductCategory contains Products, GetListings doesn't need a separate relationship with Product. It only needs a relationship with ProductCategory, from which you can derive the other relationship.

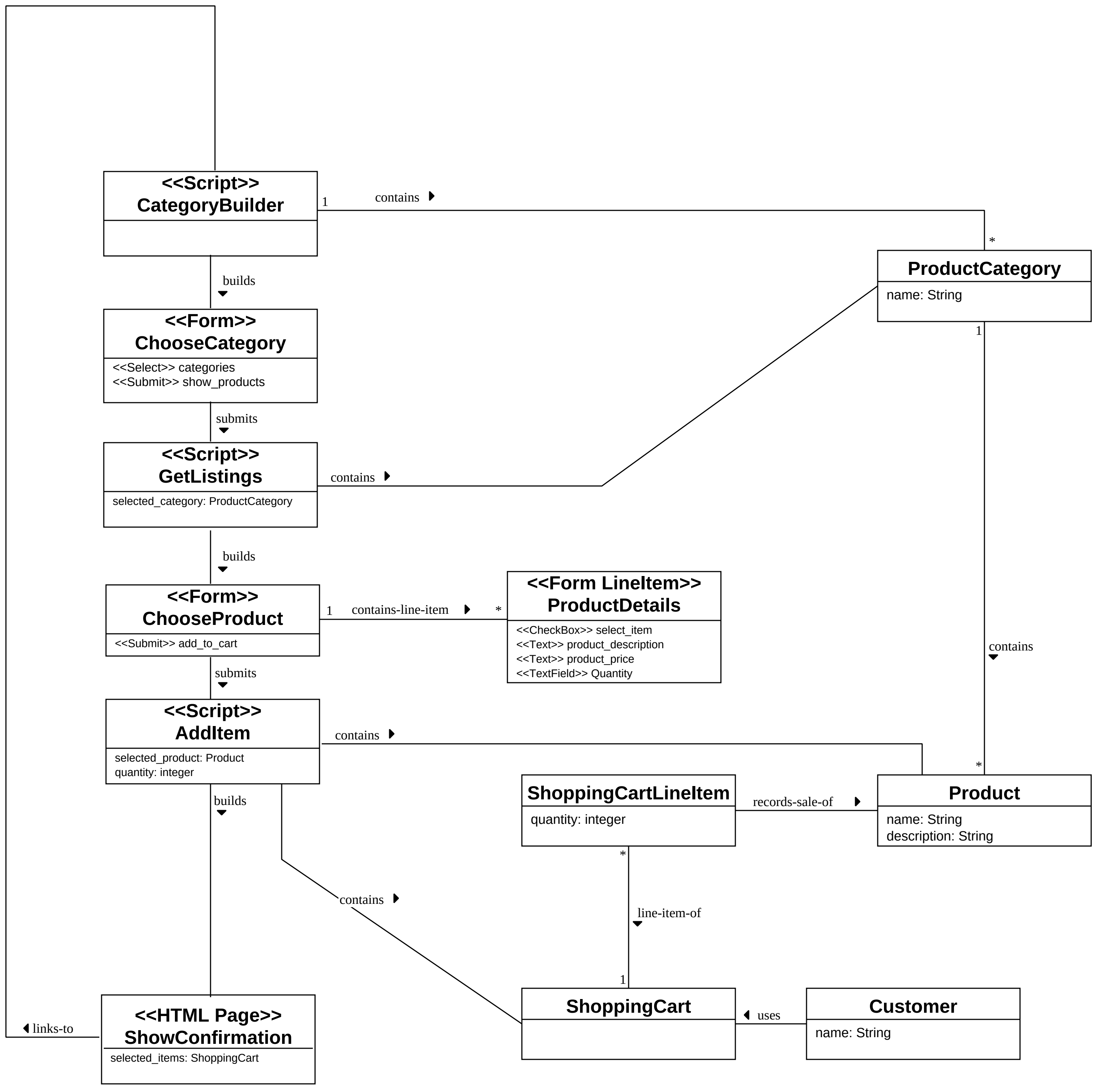
We can also do away with the uses/owns relationship between AddItem, Product, and ShoppingCart, since AddItem now has relationships with both concepts.

Category	Relationships
A contains B	CategoryBuilder contains ProductCategory GetListings contains ProductCategory AddItem contains Product AddItem contains ShoppingCart

* Why don't we have a relationship between AddItem and the ShoppingCartLineItem? Following the Low Coupling pattern, let's minimize the number of relationships that AddItem has with various concepts. It already is coupled to the Product and ShoppingCart - when we design the application code, let's see if we can make things happen without having to tell AddItem about ShoppingCartLineItems.

Putting it All Together

Now, here is our revised conceptual model, including both the user interface components and domain concepts.



Module Five Developing a Prototype

Next, we’re going to build the first prototype for this system. You might be alarmed at this point that we haven’t even considered the database at this point, but it is for the best, and will result in a stronger application. The rationale for this is Brown and Whitenack’s Table Design Time Pattern, from their Crossing Chasms paper (1996). This pattern applies when you are designing an object-oriented system that is not connected to a legacy database (or is connected to a legacy database that has a flexible design. Table Design Time suggests that your conceptual model is, in fact, a first pass at a database design. This pattern goes on to suggest that the tables should be designed after you have implemented the object model in an architectural prototype. As we’ll see, this will work out rather well when we go on to develop the database.

Designing Collaborations

Before we can go on to code, we need to figure out how the objects will collaborate with one another. We already know how they are related, and the use case will be our guide as we discover collaborations. We’ll also use patterns to help us find the right collaborations.

Let’s walk through the use case, step by step.

As we examine each step, ask yourself what needs to be done to make this happen? Examine the object model, and look for examples of the following responsibilities:

* What objects must be created (or destroyed)?
* What associations must be formed (or broken) between objects?
* What attributes must be modified?

Use Case: Select a Product

Summary: Our customer visits the store and selects a product category, after which she is presented with a list of products. Next, she chooses a product and quantity, which is added to her shopping cart. After this, she can either keep shopping, or proceed to the checkout.

User Action	System Response
1. Select a product category.	**2.** Display a descriptive list of products and their price within the selected category.
3. Choose a product and quantity.	**4.** Put the product and quantity in the shopping cart.
	5. Display the contents of the shopping cart with prices and total cost. Give the customer the opportunity to shop some more or check out.

Use Case Step One: Select a Product Category

Here is a list of responsibilities. Can you think of anything else?

1. A CategoryBuilder script was created.
2. A collection of ProductCategories was created.
3. The collection was associated with the CategoryBuilder.
4. A ChooseCategory form was created.
5. For each ProductCategory, an option was added to ChooseCategory.categories.

In typical object-oriented applications, the question of "who creates what?" can take a good amount of deliberation. In a web application, it’s a little easier, since the object model dictates much of that. Let’s revise the list to eliminate ambiguity where we can:

1. The web server created a CategoryBuilder script.
2. A collection of ProductCategories was created. (BY WHO?)
3. The collection was associated with the CategoryBuilder. (BY WHO?)
4. The CategoryBuilder created a ChooseCategory form.
5. For each ProductCategory, an option was added to ChooseCategory.categories. (BY WHO?)

Patterns for Responsibility Assignment

We still have some unassigned responsibilities, so let’s turn to some patterns to assign them.

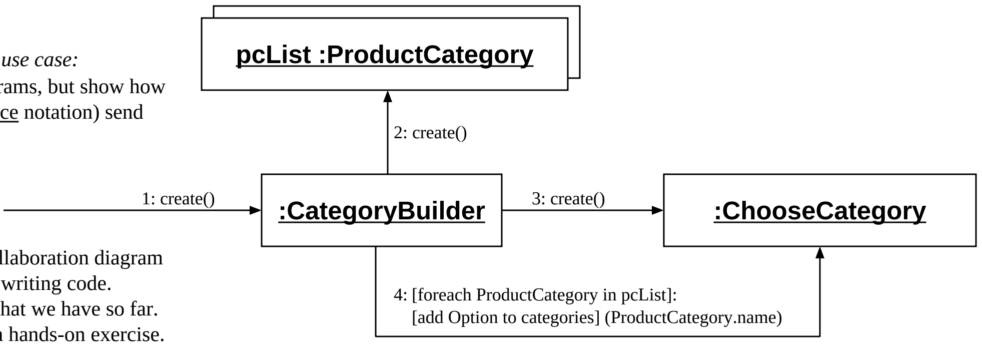
For **#2**, we need to turn to Larman’s Creator pattern. This pattern answers the question, "who is responsible for creating an instance of an object?" Because the CategoryBuilder contains (and ultimately uses) the ProductCategories, Creator suggests that CategoryBuilder is responsible for creating the collection of ProductCategories.

For **#3**, we can turn to Larman’s Expert pattern, which suggests that responsibility falls to the object that has the most information needed to fulfill that responsibility. Since CategoryBuilder created the collection, it already knows about it. So, not only does it have the information (knowledge of the collection), it already has a containment relationship with the collection, so we can actually eliminate that step!

The Expert pattern also helps us with **#5**: the CategoryBuilder has knowledge both the ChooseCategory form and the ProductCategory collection, so CategoryBuilder is the logical choice for this responsibility.

1. The web server created a CategoryBuilder script.
2. The CategoryBuilder created a collection of ProductCategories.
3. The CategoryBuilder created a ChooseCategory form.
4. For each ProductCategory, CategoryBulder added an option to ChooseCategory.categories.

Here is a collaboration diagram for Step 1 of the use case:
Collaboration diagrams look similar to class diagrams, but show how instances of classes (shown with the :class_instance notation) send messages to one another.

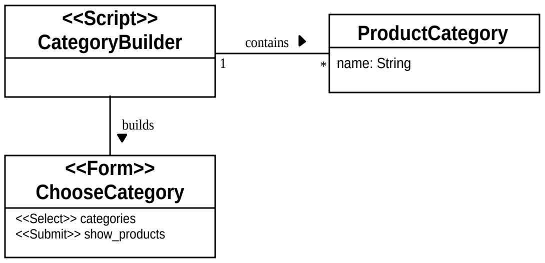

pcList :ProductCategory
2: create()
1: create()
:CategoryBuilder
3: create()
:ChooseCategory
4: [foreach ProductCategory in pcList]:
[add Option to categories] (ProductCategory.name)

Normally, you should continue developing this collaboration diagram until the use case is finished, and then move on to writing code. However, we'll take a detour here to implement what we have so far. Then, you'll continue working on the use case as a hands-on exercise. I've included part of the class diagram here as a refresher.

Now, let's build each object.

CategoryBuilder/ChooseCategory

We need to create a script that creates a collection of product categories and then builds the ChooseCategory form. How do we get a collection of product categories? If we use the ProductCategory class, we're stuck, because by default, it will only contain a constructor to generate one instance of ProductCategory. Instead, we need to define an extra helper method, fetch_all(), which creates a collection of all ProductCategories. We'll see how this is done when we look at ProductCategory.



Depending on which language you choose, you'll implement the collaboration slightly differently. For example, Perl's CGI.pm lets you create all the options in one fell swoop by using the popup_menu method, so there is no need to iterate over each element in the collection. Here are examples in Perl, Java, and Python:

CategoryBuilder.java

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Reference: Collaboration Diagram for Step 1 in
// the Select a Product use case.
//
// 1: (create CategoryBuilder) is realized when this
//    servlet is executed.
//
public class CategoryBuilder extends HttpServlet {

  public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
     throws ServletException, IOException
  {
    // Start the document.
    //
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML><HEAD>");
    out.println("<TITLE>Choose a Category</TITLE>");
    out.println("</HEAD><BODY>");

    // 2: create the ProductCategory collection.
    //
    ProductCategory[] all_categories;
    all_categories = ProductCategory.fetchAll();

    // 3: Create ChooseCategory.
    //
    out.println("<FORM NAME=\"ChooseCategory\">");

    // 4: Add each product category to categories.
    //
    out.println("Choose Category: ");
    out.println("<SELECT NAME=\"categories\">");
    for (int i = 0; i < all_categories.length; i++) {
        out.println("<OPTION>" +
            all_categories[i].getName());
    }
    out.println("</SELECT>");

    // Create the <<Submit>> show_products attribute.
    //
    out.println("<INPUT TYPE=\"submit\" " +
                " VALUE=\"Show Products\"" +
                " NAME=\"show_products\">");

    out.println("</FORM>");
    out.println("</BODY></HTML>");
    out.close();
  }

}
```

CategoryBuilder.py

```python
#!/usr/bin/python
#
# Reference: Collaboration Diagram for Step 1 in
# the Select a Product use case.
#
# 1: (create CategoryBuilder) is realized when this
#    script is executed.

import ProductCategory
import cgi

# Start the document.
#
print "Content-type: text/html"
print
print "<HTML><HEAD><TITLE>Choose a Category"
print "</TITLE></HEAD><BODY>"

# 2: Create the ProductCategory collection.
#
all_categories = ProductCategory.fetchAll()

# 3: Create ChooseCategory.
#
print '<FORM NAME="ChooseCategory">'

# 4: Add each product category to categories.
#
print "Choose Category: "
print '<SELECT NAME="categories">'

i = 0
while i < len(all_categories):
    print "<OPTION>%s" % all_categories[i].getName()
    i = i + 1
print "</SELECT>"

# Create the <<Submit>> show_products attribute.
#
print '<INPUT TYPE="submit"    \
        VALUE="Show Products"  \
        NAME="show_products">'

# Finish the form and the HTML document.
#
print "</FORM></BODY></HTML>"
```

CategoryBuilder.pl

```perl
#!/usr/bin/perl -w
#
# Reference: Collaboration Diagram for Step 1 in
# the Select a Product use case.
#
# 1: (create CategoryBuilder) is realized when
#     this script is executed.

use CGI qw(:standard);
use ProductCategory;

# 2: create the ProductCategory collection.
#
use vars qw(@all_categories);
@all_categories = ProductCategory::fetch_all();

print header(); # start the document.
print start_html("Choose a Category");

# 3: create ChooseCategory.
#
print start_form( -name   => "ChooseCategory",
                  -action => undef);

# 4: (add each product category to categories).
#
#   4a: Extract the name from each object.
#
my @names = map { $_->get_name() } @all_categories;
#
#   4b: Create the <<Select>> categories attribute
#       from the ChooseCategory class diagram.
#
print "Choose Category: "; # label for the popup
print popup_menu( -name   => 'categories',
                  -values => \@names);

# Create the '<<Submit>> show_products' attribute.
#
print submit( -name  => 'show_products',
              -value => 'Show Products');

print end_form(); # finish the form.
print end_html(); # finish the HTML document.
```

ProductCategory

ProductCategory is a simple object. It's only got one field, an accessor method to go with it, and a simple constructor. In addition, we've also got the convenience method fetch_all (or fetchAll in Java and Python, just to go with local convention), which returns an array of ProductCategory items. Currently, fetch_all() only simulates a database. In the next module, we'll make it real.

Here is ProductCategory in Perl, Java, and Python:

ProductCategory.py

```
class ProductCategory:

  # The constructor.
  def __init__(self, n):
    self.name = n

  # Return the name.
  def getName(self):
    return self.name

# Retrieve all the product categories.
def fetchAll():
  db = []
  db.append( ProductCategory("Computers"))
  db.append( ProductCategory("Toys"))
  db.append( ProductCategory("Books"))
  db.append( ProductCategory("Music"))
  return db
```

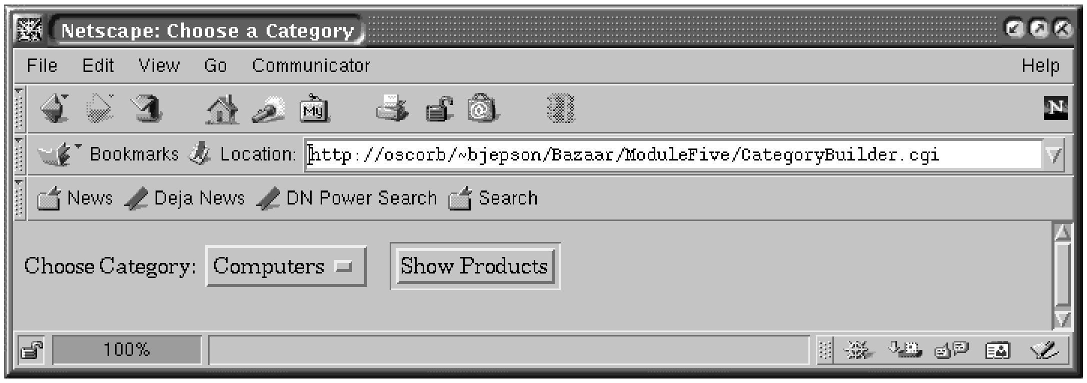
ProductCategory.java

```
public class ProductCategory {

  String name;

  // The constructor.
  //
  public ProductCategory(String n) {
      name = n;
  }

  // Get this category's name.
  public String getName() {
      return name;
  }

  // Retrieve all the product categories.
  //
  static public ProductCategory[] fetchAll() {
      // Create some fake data. Eventually, it will
      // come from a real database.
      //
      ProductCategory[] db = {
          new ProductCategory("Computers"),
          new ProductCategory("Toys"),
          new ProductCategory("Books"),
          new ProductCategory("Music")
      };
      return db;
  }

}
```

ProductCategory.pm

```
package ProductCategory;

# The constructor.
sub new {

  my $proto = shift;
  my $name  = shift;

  my $class = ref($proto) || $proto;
  my $self = {};
  $self->{name} = $name;

  bless ($self, $class);

}

# Get this category's name.
sub get_name {
  my $self = shift;
  return $self->{name};
}

# Retrieve all the product categories.
sub fetch_all {
  return @db;
}

# Create some fake data. Eventually, this
# will come from a real database.
use vars qw( @db );
@db = (
        new ProductCategory("Computers"),
        new ProductCategory("Toys"),
        new ProductCategory("Books"),
        new ProductCategory("Music"),
);
1;
```

Here is a snapshot of the page, followed by a dump of the HTML that was generated:


Netscape: Choose a Category
File Edit View Go Communicator Help
Bookmarks Location:
News Deja News DN Power Search Search
100%

http://oscorb/~bjepson/Bazaar/ModuleFive/CategoryBuilder.cgi

Choose Category: Computers [Show Products]

```
<HTML>

<HEAD>
<TITLE>Choose a Category</TITLE>
</HEAD>

<BODY>

<FORM METHOD="POST"  ENCTYPE="application/x-www-form-urlencoded" NAME="ChooseCategory">
Choose Category: <SELECT NAME="categories">
  <OPTION  VALUE="Computers           ">Computers
  <OPTION  VALUE="Books               ">Books
  <OPTION  VALUE="Toys                ">Toys
  <OPTION  VALUE="Music               ">Music
</SELECT>
<INPUT TYPE="submit" NAME="show_products" VALUE="Show Products">
</FORM>

</BODY>
</HTML>
```

Module Six Persistence

For a web application, persistence comes in two forms: stuff that is persistent forever (or at least until it is destroyed somehow) and stuff that is persistent for the duration of a user's session. We'll call these long-term and session persistence.

How do we determine what needs to be persistent? In general, you only need to look at the original conceptual model for candidates. Here is a list of potential candidates for persistence:

ProductCategory
Product
ShoppingCart
ShoppingCartLineItem
Customer

Let's look at each of these, and decide what type of persistence we need. In doing so, we'll ask some questions:

1. Do all users have access (read or write) to this object?
   If the answer is yes, then this object probably needs to have long-term persistence.
   Examples: Product catalog, news articles, and guest book entries.

2. Is this object part of a transaction that the system needs to remember?
   If the answer is yes, then the object needs long-term persistence.
   Example: One-click purchases, purchases, product registration.

3. If this object is related to a transaction of some sort, will the system forget it if the user decides to abort the transaction?
   If the answer is yes, then the object needs only session persistence until the user commits to the transaction.
   Examples: Purchases, creation of a new user, and product registration.

Can you think of some other questions that would be useful to ask?

Here are the answers I came up with:

Object	**Q1**	**Q2**	**Q3**	**Persistence**
ProductCategory	Yes	No	No	Long-term
Product	Yes	No	No	Long-term
ShoppingCart	Yes	Yes	Yes	Session, long-term when user checks out.
ShoppingCartLineItem	Yes	Yes	Yes	Session, long-term when user checks out.
Customer	No	Yes	No	Long-term.

Client or Server-Side Persistence?

In general, long-term persistence is handled on the server-side, and short-term persistence is handled on the client-side. There may be some exceptions to this, but we'll proceed with this simple division of responsibility.

As the title of this talk suggests, databases are going to show up some place or another. And here it comes: we're going to use a database to implement persistence on the server side.

Modern web browsers include support for cookies, which are small bits of information stored on the browser. These are very useful for storing session data, so we will use cookies to store anything that needs to be stored on the client-side. Some web application APIs, such as Java Servlets, include session management features that can make it easier to work with session persistence.

Developing the Database

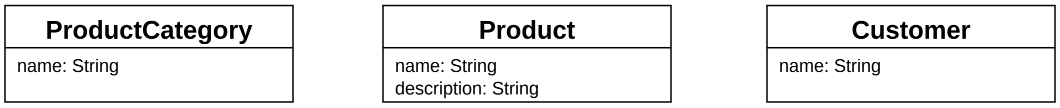
Database design is a slightly involved topic, and we're going to take a shortcut here to avoid a lengthy digression.

You can use the items in your conceptual model as the basis of your database design. From here, it is best if you apply the principles of database design known as the normal forms. The purpose of the normal forms is to optimize the structure of your database for storage and retrieval. In applying the normal forms, you may rearrange the way objects and their attributes are structured, and you may even introduce new objects to your conceptual model.

Because we are looking at a simple example, and in the interests of keeping this tutorial manageable, we are going to use the conceptual model as the guide for creating the database. In practice, you should employ the normal forms before creating the database. The suggested reading list at the end of this booklet lists some good database texts that cover the normal forms.

Let’s look at the three objects that will have long-term persistence.

We do not include the ShoppingCart or ShoppingCartLineItem here, simply because it does not need long-term persistence in this use case. For the purposes of this example, we are only considering the needs of the current use case. When we complete this use case, we’ll consider another use case as the more hands-on portion of this tutorial. As you work through that use case, you can build upon the implementation of the current use case, which you’ll then look at as a prototype.



The Customer object will be off by itself. As is the case with many online stores, the customer won’t have to identify herself until she checks out. So, as far as this use case is concerned, the customer is an anonymous actor. But, we will define the customer object for completeness’ sake.

What does it take to represent one of these classes in the database? On the database server, you need only define the table using SQL. Here is the definition for ProductCategory. Notice that we’ve added another attribute, the id. In general, you should have a unique identifier for every database table, which distinguishes one row from another. This identifier is also called a primary key.

```
CREATE TABLE productcategory (name CHAR(20), id INT)
```

Now, we can insert some sample data with the following SQL statements:

```
INSERT INTO productcategory VALUES (’Computers’, 100)
INSERT INTO productcategory VALUES (’Books’,     110)
INSERT INTO productcategory VALUES (’Toys’,      120)
INSERT INTO productcategory VALUES (’Music’,     130)
```

The Easy Way

There are two ways to make the ProductCategory persistent. One is the easy way: we’ll mess with the object definition, and plug some database connectivity into its fetch_all() method.

The second way is more involved, and we’ll look at it after the easy way.

In the easy way, we rewrite ProductCategory’s fetch_all method to make a database connection, issue an SQL statement, and create ProductCategory objects using the data retrieved from the database. Here is an example in Perl:

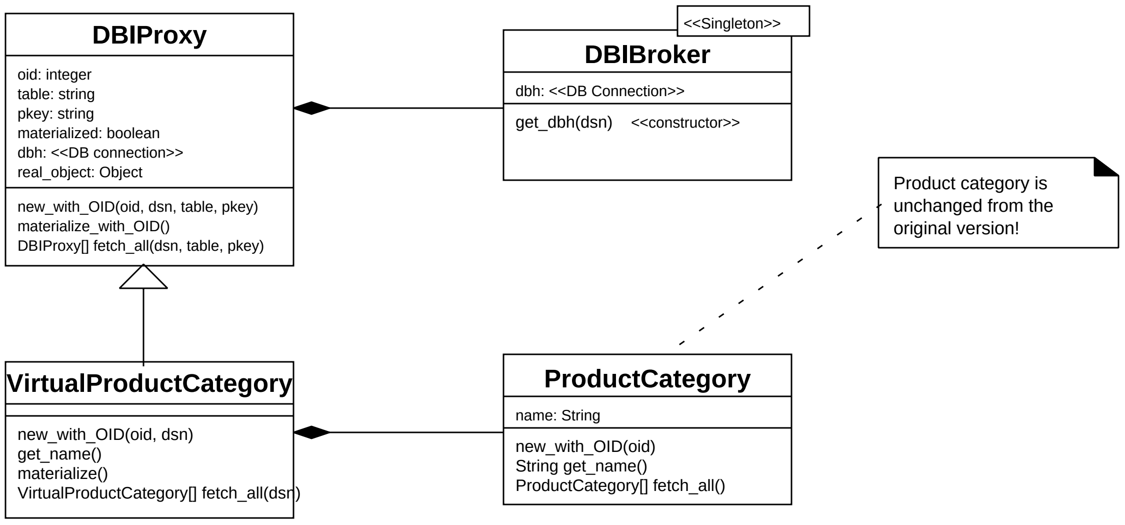
```
# Retrieve all the product categories.
#
use DBI;
sub fetch_all {

  # Connect to the database.
  #
  my $dsn = shift;
  my $dbh = DBI->connect($dsn);
  die "Connection failed: $DBI::errstr" if $DBI::errstr;

  # Select all the categories.
  #
  my $sql = "SELECT name FROM productcategory";
  my $sth = $dbh->prepare($sql);
  $sth->execute;
  die "SQL failed: $DBI::errstr" if ($DBI::errstr);

  my @all;
  while (my @row = $sth->fetchrow_array) {
      push @all, new ProductCategory($row[0]);
  }

  $sth->finish();
  $dbh->disconnect();
  return @all;
}
```

To use this new version of ProductCategory, we need to change our call to fetch_all() to pass in a database connection string:

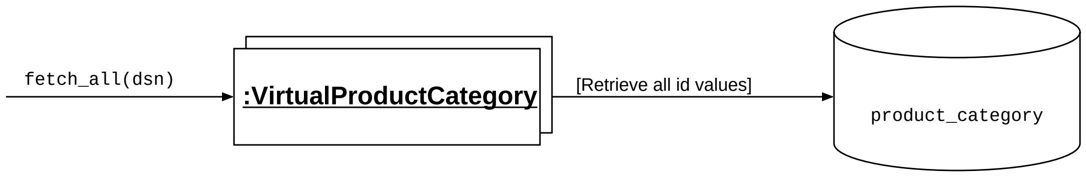
```
@all_categories = ProductCategory::fetch_all("dbi:Pg:dbname=bjepson");
```

The Hard Way

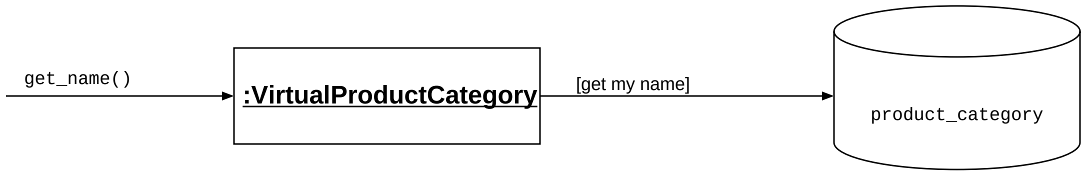
Here is a sample pattern that is loosely based on the ideas presented in Brown and Whitenack's Crossing Chasms. This example should give you a taste of how you could develop your own persistence framework. See the Crossing Chasms paper for a comprehensive pattern language that treats this topic.



Class Diagram.



Fetch all ProductCategory objects.



Wait until we need the name to fetch it from the database.

DBIProxy.pm

```perl
package DBIProxy;
use DBI;

# The constructor.
#
sub new_with_OID {

  my $proto = shift;  # The class.
  my $class = ref($proto) || $proto;

  my $oid  = shift;   # The object id.
  my $dsn  = shift;   # Connection string.
  my $table  = shift; # The table name.
  my $pkey  = shift;  # The primary key name.

  my $self = {};

  $self->{oid}   = $oid;
  $self->{table} = $table;
  $self->{pkey}  = $pkey;
  $self->{materialized} = 0;

  # Connect to the database.
  #
  my $dbh = DBIBroker->get_dbh($dsn);
  die "Connection failed: $DBI::errstr" if $DBI::errstr;
  $self->{dbh}   = $dbh;

  # A pointer to the real object.
  #
  $self->{real_object} = undef;

  bless ($self, $class);
  return $self;

}

# Fetch this object’s attributes from the database.
#
sub materialize_with_OID {

  my $self = shift;
  if (! $self->{dbh} ) {
      die "Cannot materialize without DB connection.";
  }
  my $dbh = $self->{dbh};

  # Select the row that corresponds to this OID.
  #
  my $table = $self->{table};
  my $pkey  = $self->{pkey};
  my $oid   = $self->{oid};
  my $sql = "SELECT * FROM $table WHERE $pkey = $oid";

  my $sth = $dbh->prepare($sql);
  $sth->execute;
  my $attr = $sth->fetchrow_hashref;
  if ($DBI::errstr) {
     die "Materialization failed: $DBI::errstr";
  }

  # Set all the properties.
  #
  foreach (keys %$attr) {
      $self->{properties}->{$_} = $attr->{$_};
  }
  $self->{materialized} = 1;

}
```

```perl
# Fetch all instances of an object from the database, but
# only create virtual instances (deferring materialization
# until we try to access the object).
#
sub fetch_all {

  my $proto = shift;
  my $dsn = shift;
  my $table = shift;
  my $pkey  = shift;

  # Connect to the database.
  #
  my $dbh = DBIBroker->get_dbh($dsn);

  my $sql = "SELECT $pkey FROM $table";
  my $sth = $dbh->prepare($sql);
  $sth->execute;
  if ($DBI::errstr) {
     die "FetchAll failed: $DBI::errstr";
  }

  my @all;
  while (my @row = $sth->fetchrow_array) {
      my $obj =
         $proto->new_with_OID($row[0], $dsn, $table, $pkey);
      push @all, $obj;
  }
  return @all;

}
# A little broker to make sure we all use the same
# DBI connection. Uses the Singleton pattern.
#
package DBIBroker;
use DBI;

use vars qw($dbh);
$dbh = undef;

# Get the DBI connection.
#
sub get_dbh {

  my $proto = shift;  # The class.
  my $class = ref($proto) || $proto;

  my $dsn  = shift;   # Connection string.

  # If the connection doesn’t exist, create it.
  #
  if (!$dbh) {
    $dbh = DBI->connect($dsn);
    die "Connection failed: $DBI::errstr" if $DBI::errstr;
  }
  return $dbh;

}

sub DESTROY {
    $dbh->disconnect();
}
1;
```

VirtualProductCategory.pm

```
package VirtualProductCategory;
use ProductCategory;
use DBIProxy;

@ISA = qw(DBIProxy);

use vars qw($TABLE $PKEY);
$TABLE = 'ProductCategory';
$PKEY  = 'id';

sub new_with_OID {

  my $proto = shift;
  my $class = ref($proto) || $proto;
  my $oid   = shift;
  my $dsn   = shift;

  my $self =
    $class->SUPER::new_with_OID($oid, $dsn, $TABLE, $PKEY);

  bless ($self, $class);
  return $self;

}

# Return the name of this ProductCategory.
#
sub get_name {

  my $self = shift;
  $self->materialize();
  return  $self->{real_object}->get_name;
}

# Materialize this object if it doesn't exist already.
#
sub materialize {

  my $self = shift;
  if (! $self->{materialized} ) {

    # Step 1: Materialize the data.
    #
    $self->materialize_with_OID();

    # Step 2: Create an instance of the real object,
    #         and set all its attributes.
    #
    my $name = $self->{properties}->{name};
    $self->{real_object} = new ProductCategory($name);
  }

}

# Fetch all instances of an object from the database, but
# only create virtual instances (deferring materialization
# until we try to access the object).
#
sub fetch_all {

  my $proto = shift;  # Need classname to find superclass.
  my $class = ref($proto) || $proto;
  my $dsn = shift;    # Data source.

  return $class->SUPER::fetch_all($dsn, $TABLE, $PKEY);

}
1;
```

Example Program

```
use VirtualProductCategory;
$dsn = "dbi:Pg:dbname=bjepson"; # Connection String.

# Get all categories.
#
@all_categories = VirtualProductCategory->fetch_all($dsn);

# Print all category names.
#
print map{ $_->get_name() . "\n" } @all_categories;

# Get one category and print its name.
#
$eg = new_with_OID VirtualProductCategory(100, $dsn);
print "Example (ID=100): ", $eg->get_name(), "\n";
```

CategoryBuilder.cgi

```
#!/usr/bin/perl -w
#
# Reference: Collaboration Diagram for Step 1 in
# the Select a Product use case.
#
# 1: (create CategoryBuilder) is realized when
#     this script is executed.
use CGI qw(:standard);
use VirtualProductCategory;

# 2: create the ProductCategory collection.
#
use vars qw(@all_categories);
my $dsn = "dbi:Pg:dbname=bjepson";
my @all_categories = VirtualProductCategory->fetch_all($dsn);

print header(); # start the document.
print start_html("Choose a Category");

# 3: create ChooseCategory.
#
print start_form( -name   => "ChooseCategory",
                  -action => undef);

# 4: (add each product category to categories).
#
#   4a: Extract the name from each object.
#
my @names = map { $_->get_name } @all_categories;
#
#   4b: Create the '<<Select>> categories' attribute
#       from the ChooseCategory class diagram.
#
print "Choose Category: "; # label for the popup
print popup_menu( -name   => 'categories',
                  -values => \@names);

# Create the '<<Submit>> show_products' attribute.
#
print submit( -name  => 'show_products',
              -value => 'Show Products');

print end_form(); # finish the form.
print end_html(); # finish the HTML document.
```

Recommended Reading

Bowman, Judith S.; Emerson, Sandra L.; Darnovsky, Marcy. 1996 The Practical SQL Handbook: Using Structured Query Language. Addison-Wesley.

Brown, Kyle; Whitenack, Bruce G. 1996. Crossing Chasms. Pattern Languages of Program Design vol. 2. Addison-Wesley

Conallen, Jim. October, 1999. Modeling Web Application Architectures with UML. Communications of the ACM. Vol. 24, No. 10.

Fowler, Martin; Scott, Kendall. 1997. UML Distilled: Applying the Standard Object Modeling Language. Addison-Wesley.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Larman, Craig. 1998. Applying UML and Patterns. Prentice Hall PTR.

Mandel, Theo. 1997. The Elements of User Interface Design. Wiley