# 3 An REA-Based Example Application

*By Christian Vibe Scheller*

In this chapter I will show you just how easy it is to use REA for developing software applications. I will do so by developing a simple order website, where Joe's customers can order pizzas. The finished webpage will look like this:

**Fig. 92.** Joe's web shop

The customer enters his order by first entering his name and address. This allows Joe's Pizzeria to know where to deliver the pizzas and to whom. If the customer is already registered in the system, he can press the link labeled "already a customer?" This will cause the web page to display the customer's address without the customer having to type it himself.

The customer proceeds to enter his order by specifying which pizzas he wants to order and how many. The web page responds by calculating the total amount the customer has to pay for the order.

Finally, the customer presses the submit button. Only then will all the order information be stored in the database. In a real web application the customer would then have to specify credit card information, etc., but we will skip this part for the sake of simplicity.

## 3.1 Representing the Metamodel

A special concern when implementing an application based on the REA model is that the REA model exists on two separate levels of abstraction (the application model and the metamodel).

As a general rule we should not mix two levels of abstraction in the same source code. While it is possible to do so in programming languages that support reflection, it is almost always the case that the reflection code and the reflected code resides in different components.

We need to make a choice: If we implement the application model, we will just have to map the concepts of the metamodel as well as possible to the existing metamodel of the programming language (e.g. by using inheritance to represent metamodel elements or by using attributes to describe metadata). If we implement the metamodel however, the application model becomes data and we are basically developing our own programming language.

I can see the benefits of both approaches: I find that the first approach is easy to explain and understand whereas most developers get scared by the second approach. The second approach, however, results in a model that captures the deep knowledge of the business model in a much more profound way.

We will look into the approach of implementing the metamodel in chapter *An Aspect-Based Example Application* at the end of Part II of this book, but for now we will stick to implementing the application model.

## 3.2 Component Model

Let us start out by defining the components that we want to build our application from. The dependencies between the different components are shown in Fig. 93.

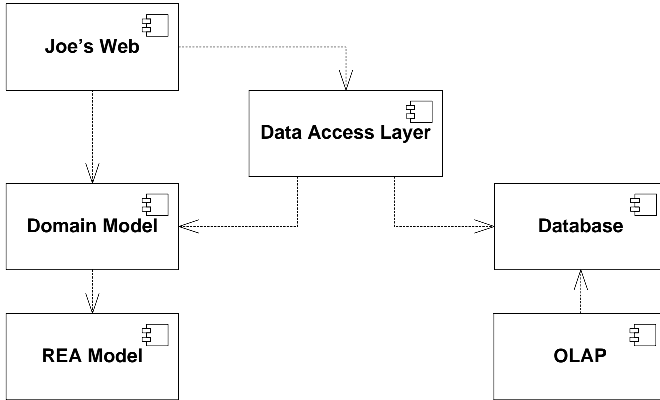



**Fig. 93.** Component model of the REA sample application

*REA model* defines the underlying REA model. Classes such as Order and Customer will inherit from base classes defined in this component. The REA model component will be designed with reusability in mind, so it can be reused in other REA-based applications.

*Domain model* contains all the entities that make up Joe's Pizzeria. In a real-life application the domain model would contain everything including purchase, production, salaries, etc., but in our small sample application we will only model sales orders and customers. We will make the design rule that all classes in the domain model must inherit from one of the base classes in the REA model component.

*Joe's Web* is the actual web site that the customers will be visiting when they want to order pizzas. Joe's web consists of a number of web pages running on a web server. As a design rule we will not put any business logic directly in this component. All the business logic will instead be placed in the domain model and REA model components.

*Data Access Layer* is responsible for retrieving objects from the database as well as storing objects in the database. The process of transforming a domain object to its database equivalent is often referred to as O/R mapping. While O/R mapping tools exist, in the case of this simple web application we will just be writing the code ourselves.

*Database* is where the data (orders, customers, etc.) gets stored. The database is the only persistent component in the application, so if we want our data to be available over time we need to put it in the database.

*OLAP* – In our sample application we would like to provide Joe with all kinds of information about his business: What kinds of pizzas are the most popular? Are sales going up or down? Etc. In my opinion an OLAP cube is the ideal tool for this kind of information.

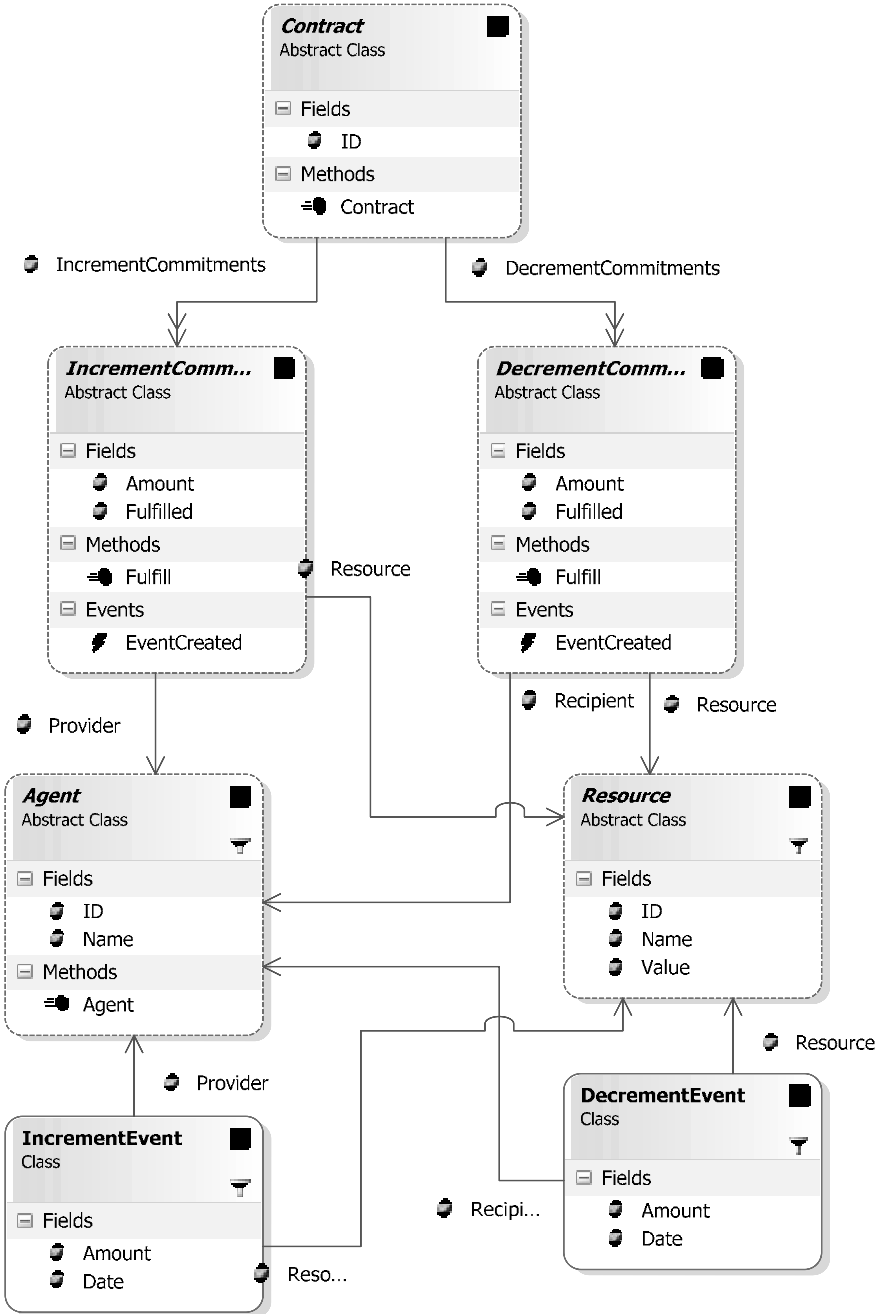



**Fig. 94.** REA Model Component

## 3.3 The REA Model Component

Fig. 93 shows the REA model that we will be basing our application on. As can be seen, the model is not a complete REA model. This is because we don't need concepts such as duality in our sample application. The simplification of the REA model is a pattern in itself called *MODELING COMPROMISE*.

Each object in the REA model is defined as an abstract base class. When we later define our domain model, each of the domain objects is going to inherit from one of these base classes. The exception to this rule is the Event class, which does not have a domain counterpart.

As can be seen from the diagram, the *Agent* class has two fields. The first field is the *ID* which is a unique identifier for the *Agent* class. The main purpose of the *ID* field is to identify the agent record in the database as well as to solve the ambiguity that would otherwise occur if two agents were to have the same name. The *Name* field is also a kind of identifier of the agent but it is less strict than the *ID* in that it is not necessarily unique. On the other hand the *Name* is the identifier that humans use: "Did John Doe receive his pizzas?" Joe might ask. Anyway, here is the code:

```
public abstract class Agent {
    public int ID;
    public string Name;
}
```

Just like agents, *Resources* contain an *ID* and *Name*. In addition a resource has a *Value* which is defined as the value in US dollars of a single unit of the resource, i.e., the price of a single pizza:

```
public abstract class Resource {
        public int ID;
        public string Name;
        public double Value;
}
```

The *Contract* class contains an *ID* field and two collections: A collection of increment commitments and a collection of decrement commitments.

```
public abstract class Contract {
    public int ID;
    public List<IncrementCommitment> IncrementCommitments = …
    public List<DecrementCommitment> DecrementCommitments = …
}
```

First of all it is worth noting that the *Increment Commitment* class, unlike the Agent, Resource and Contract classes, does not have an ID. This is because commitments do not have identities – after all what is the difference between receiving ten dollars and receiving five dollars and then another five dollars? Another thing worth noting is that the commitment classes contain a fulfillment mechanism:

Once a certain commitment is fulfilled, the application can call the commitment object's *Fulfill()* method. This will cause the commitment to change its *Fulfilled* field to *true* and will also cause the commitment to generate an economic event based on its own information. Since the commitment itself does not know what to do with this economic event, it will pass it to the calling application using the *EventCreated* delegate.

```
public abstract class IncrementCommitment {
    public Resource Resource;
    public double Amount;
    public Agent Provider;
    public bool Fulfilled = false;
    public event IncrementEventCreatedHandler EventCreated;

    public void Fulfill() {
        Fulfilled = true;
        IncrementEvent e = new IncrementEvent(this);
        EventCreated(e);
    }
}
```

Basically, *decrement commitments* are identical to increment commitments except they have a recipient instead of a provider. While writing this chapter I was debating with Pavel whether decrement and increment commitments should actually be modeled as different classes or if they should rather be merged into a single generic commitment class. In the end we decided that the semantic difference between the two types of commitments is so important to the whole REA model that they should be kept separate.

The *Increment Event* and *Decrement Event* will be generated by the REA model component whenever a commitment is marked as fulfilled by calling its *Fulfill()* method.

```
public class IncrementEvent {
    public DateTime Date;
    public Resource Resource;
    public double Amount;
    public Agent Provider;

    public IncrementEvent(IncrementCommitment commitment) {
        Date = DateTime.Now;
        Resource = commitment.Resource;
```

```
            Amount = commitment.Amount;
            Provider = commitment.Provider;
        }
}
```

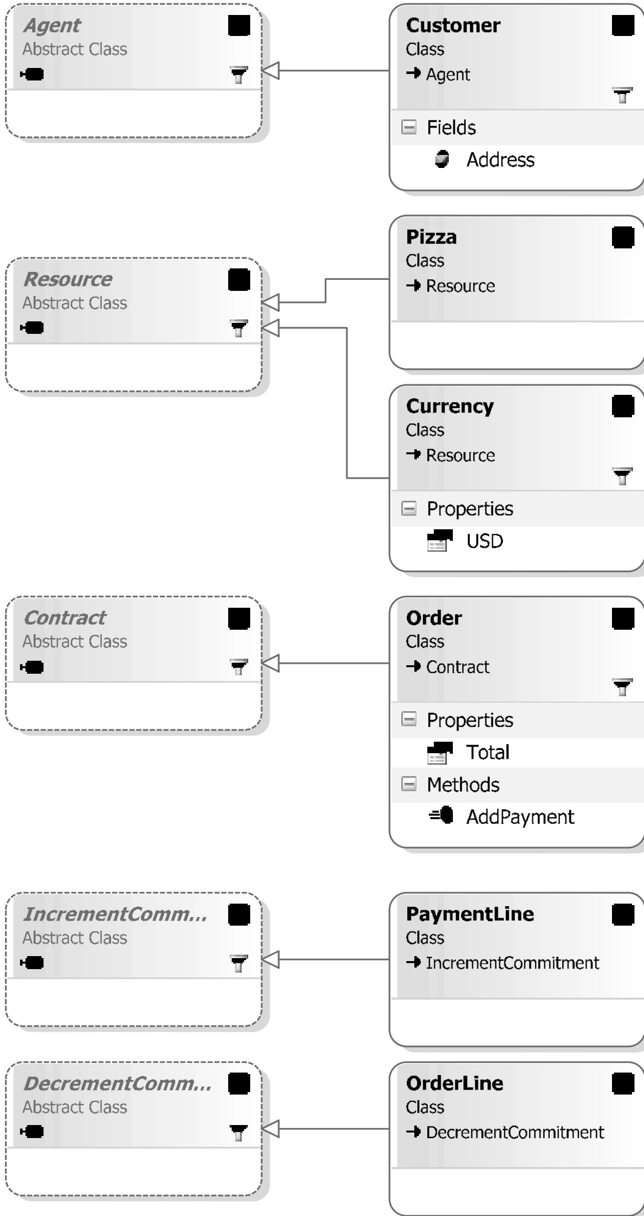



**Fig. 95.** Domain Model Component

## 3.4 The Domain Model Component

The diagram in Fig. 95 shows the domain model component. It is worth noting that the model does not contain any associations between domain classes. This is because all the associations are inherited from the REA model component. It can also be seen that each domain class inherits from a corresponding REA class.

*A Customer* is basically an agent. An *Address* field has been added so that Joe will know where to deliver the *Pizzas*.

```
public class Customer : Agent {
    public string Address;
}
```

*Pizzas* are resources.

```
public class Pizza : Resource {
}
```

The *Currency* class is needed because the REA model expects every commitment and event to have a *Resource*. The *Currency* class represents monetary value. In reality, only one type of currency will be used in the application, namely US dollars, so we implement a singleton pattern.

```
public class Currency : Resource {
    private Currency() {}

    public static Currency USD {
        get {
            Currency usd = new Currency();
            usd.ID = 0;
            usd.Name = "USD";
            usd.Value = 1;
            return usd;
        }
    }
}
```

An *Order Line* is a decrement commitment where Joe's Pizzeria commits itself to deliver a given number of pizzas of a specific type to a customer.

```
public class OrderLine : DecrementCommitment {
}
```

A *Payment* is an increment commitment where the *Customer* commits himself to pay Joe's Pizzeria a certain amount of currency.

```
public class PaymentLine : IncrementCommitment {
}
```

The *Order* class is the only class in the domain model component that adds something that could reasonably be called business logic. The order is able to calculate the total amount (in USD) that the customer should pay for his pizzas. The order can also add a payment line based on this total to its incoming commitments.

```
public class Order : Contract {
    public double Total {
        get {
            double total = 0;
            foreach (OrderLine line in DecrementCommitments) {
                total += line.Amount * line.Resource.Value;
            }
            return total;
        }
    }

    public void AddPayment(Customer customer, Currency currency) {
        IncrementCommitments.Clear();
        PaymentLine line = new PaymentLine();
        line.Amount = Total;
        line.Resource = currency;
        line.Provider = customer;
        IncrementCommitments.Add(line);
    }
}
```

All in all the domain model component consists of only 28 lines of code (not including blank lines and closing brackets).

## 3.5 The Database

The database is designed to mimic the domain model as closely as possible, see Fig. 96. All fields have the same name and are of the same data type as in the domain model. A few exceptions are necessary, however, due to the nature of databases:

- In the domain model order lines and payment lines are part of an order. In the database this is modeled by adding an order ID to each order line and payment line.
- In the domain model resources, providers and recipients are references to resource and agent objects. In the database, resource ID, provider ID or recipient ID are foreign keys to the pizza and customer tables.

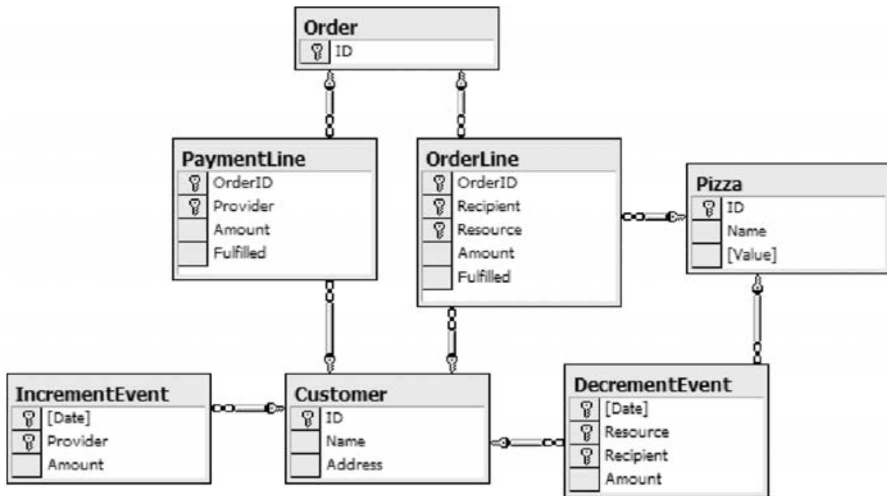



**Fig. 96.** The database

## 3.6 The Data Access Layer

The data access layer contains a single static class with a number of methods for retrieving and saving data to the database, see Fig. 97.

These methods are extremely simple so I will not waste too much space listing all the code. Here is a single example showing the code for the *GetPizzas()* method:

```
public static Dictionary<int, Pizza> GetPizzas() {
    Dictionary<int, Pizza> pizzas = new Dictionary<int, Pizza>();
    using (SqlConnection connection = new SqlConnection("…")) {
        connection.Open();
        SqlCommand command = new SqlCommand("select number, name,
                                    price from pizza", connection);
        SqlDataReader reader = command.ExecuteReader();
        while (reader.Read()) {
            Pizza pizza = new Pizza();
            pizza.ID = reader.GetInt32(0);
            pizza.Name = reader.GetString(1);
            pizza.Price = (double) reader.GetDecimal(2);
            pizzas.Add(pizza.ID, pizza);
        }
    }
    return pizzas;
}
```

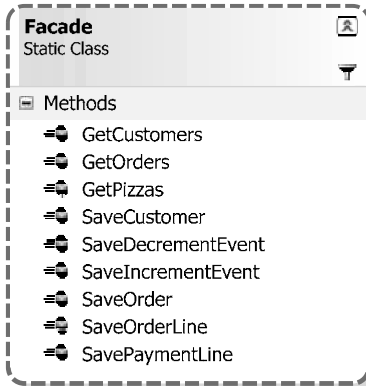



**Fig. 97.** The data access layer

One of the interesting features of the data access layer is that it is responsible for saving the economic events generated by the commitments. It does so by attaching an event handler to the order lines and payment lines in the *GetOrders()* method:

```
public static Dictionary<int, Order> GetOrders() {
…
…
OrderLine line = new OrderLine();
order.DecrementCommitments.Add(line);
line.EventCreated +=
    new DecrementEventCreatedHandler(OrderLine_EventCreated);
…
…
PaymentLine line = new PaymentLine();
order.IncrementCommitments.Add(line);
line.EventCreated +=
    new IncrementEventCreatedHandler(PaymentLine_EventCreated);
…
…
}

static void OrderLine_EventCreated(DecrementEvent e) {
    SaveDecrementEvent(e);
}

static void PaymentLine_EventCreated(IncrementEvent e) {
    SaveIncrementEvent(e);
}
```

## 3.7 Joe's Web

Now that all the underlying components are in place we are ready to develop the user interface.

The order web page is developed in ASP.Net and uses the page's *View-State* to store the order and customer objects between post backs. This is extremely convenient when you base your development on a domain model.

```
public partial class CreateOrder : System.Web.UI.Page {
    Order Order;
    Customer Customer;

    protected void Page_Load(object sender, EventArgs e) {
        if (!IsPostBack) {
            Order = new Order(Facade.GetNextOrderID());
            OrderNumberLabel.Text = Order.ID.ToString();
            Customer = new Customer(Facade.GetNextCustomerID());
            foreach (Pizza pizza in Facade.GetPizzas().Values) {
                ListItem item =
                    new ListItem(pizza.Name, pizza.ID.ToString());
                ResourceList.Items.Add(item);
            }
            ViewState.Add("order", Order);
            ViewState.Add("customer", Customer);
        } else {
            Order = (Order) ViewState["order"];
            Customer = (Customer)ViewState["customer"];
            foreach (OrderLine line in Order.DecrementCommitments) {
                AddOrderLineTableRow(line);
            }
        }
    }
```

If the user presses the *Already a customer* link, see Fig. 92, the web page will search the database for a customer with the correct name and then use that customer as the recipient for the order lines. The web page will also display the customer's address information:

```
protected void AlreadyCustomer_Click(object sender, EventArgs e) {
    foreach (Customer customer in Facade.GetCustomers().Values) {
        if (customer.Name == NameTextBox.Text) {
            Customer = customer;
            AddressTextBox.Text = customer.Address;
            ViewState.Add("customer", Customer);
            break;
        }
    }
}
```

When the user presses the *add to order* button, the web page will generate an order line based on the information that the user has entered and then add that order line to the order object:

```
protected void AddToOrder_Click(object sender, EventArgs e) {
    OrderLine line = new OrderLine();
    line.Amount = double.Parse(QuantityTextBox.Text);
    line.Resource = Facade.GetPizzas()[ResourceList.SelectedValue];
    line.Recipient = Customer;
    Order.DecrementCommitments.Add(line);
    AddOrderLineTableRow(line);
    ViewState.Add("order", Order);
    TotalAmountLabel.Text = Order.Total.ToString("#.00");
}
```

The final piece of code that we need for our web page is the code behind the *Submit your order* button:

```
protected void Submit_Click(object sender, EventArgs e) {
    Order.AddPayment(Customer, Currency.USD);
    Customer.Name = NameTextBox.Text;
    Customer.Address = AddressTextBox.Text;
    Facade.SaveCustomer(Customer);
    Facade.SaveOrder(Order);
    Response.Redirect("MainPage.aspx");
}
```

Now everything is in place and Joe is ready to receive orders from his customers.

## 3.8 The Fulfillment Page

Once the customer has submitted the order, Joe needs to keep track of it. He needs to know whether the customer has received his pizzas and whether he has paid for them or not. For this purpose the system contains a fulfillment page, illustrated in Fig. 98.

Order no. 10022

Order lines

| Resource | Quantity | Recipient | Fulfilled |
|---|---|---|---|
| Pizza Salsiccia | 2 | John Doe | ☐ |
| Pizza Pollo e Pesto | 3 | John Doe | ☐ |

Payment

| Resource | Amount | Provider | Fulfilled |
|---|---|---|---|
| USD | 55,75 | John Doe | ☐ |

Save Changes

**Fig. 98.** The fulfillment web page

By checking the checkboxes, Joe can mark a specific order line or payment line as *Fulfilled*. The fulfillment page supports scenarios where the customer pays up front for his pizzas as well as scenarios where the customer pays on delivery. At least in the area where I live, both these scenarios occur regularly.

Less realistic is the fact that Joe can partly fulfill an order, but only by providing all the pizzas of a specific type at once. This flaw is caused by the simplified fulfillment mechanism we implemented in the REA model.

Behind the scenes the fulfillment page is using the same domain model, data access layer and database as the order web page. When Joe presses the *Save Changes* button, the web page runs through all checkboxes and calls the associated order line or payment line's fulfill method if necessary:

```
protected void SaveChanges_Click(object sender, EventArgs e) {
    for(int i=0; i < OrderLineTable.Rows.Count; i++) {
        CheckBox checkbox =
            (CheckBox) OrderLineTable.Rows[i].Cells[3].Controls[0];
        if (checkbox.Checked &&
            !Order.DecrementCommitments[i].Fulfilled) {
            Order.DecrementCommitments[i].Fulfill();
        }
    }
    for (int i = 0; i < PaymentLineTable.Rows.Count; i++) {
        CheckBox checkbox =
            (CheckBox) PaymentLineTable.Rows[i].Cells[3].Controls[0];
        if (checkbox.Checked &&
            !Order.IncrementCommitments[i].Fulfilled) {
            Order.IncrementCommitments[i].Fulfill();
        }
    }
    Facade.SaveOrder(Order);
    Response.Redirect("MainPage.aspx");
  }
```

This eventually causes decrement events and increment events to be stored in the database, see Fig. 99:

DecrementEvent : Table

| Date | ResourceID | RecipientID | Amount |
|---|---|---|---|
| 2/21/2005 | 15 | 1002 | 1 |
| 8/1/2005 | 13 | 1002 | 1 |
| 6/13/2005 | 10 | 1002 | 3 |
| 10/11/2005 | 11 | 1002 | 2 |
| 4/13/2005 | 12 | 1001 | 1 |
| 10/19/2005 | 12 | 1002 | 1 |
| 9/18/2005 | 10 | 1003 | 5 |
| 6/20/2005 | 15 | 1005 | 2 |
| 3/4/2005 | 10 | 1004 | 2 |
| 7/15/2005 | 11 | 1002 | 2 |
| 8/17/2005 | 13 | 1001 | 3 |
| 9/7/2005 | 12 | 1003 | 1 |
| 11/13/2005 | 10 | 1004 | 1 |
| 9/8/2005 | 11 | 1005 | 1 |

Record: 1

**Fig. 99.** Decrement event table

## 3.9 The OLAP Cube

Now it is time to generate some management reports based on our event data.

To make it really simple let us just add a simple Microsoft Access pivot table on top of each of the event tables. While this is not a real OLAP cube it still provides us with the same basic functionality.

The definition of the cube based on decrement events is in Fig. 100.

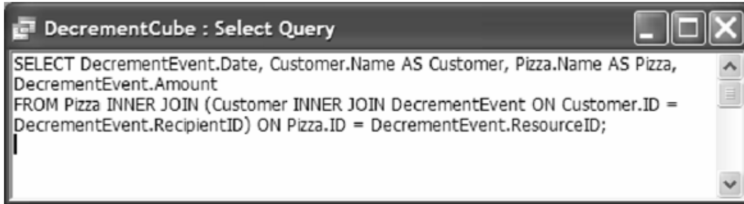
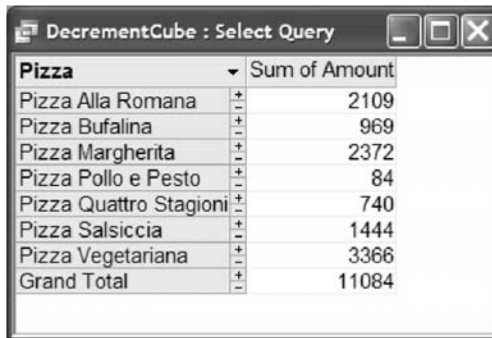



**Fig. 100.** Definition of the decrement event table

We can use this cube to get simple sales statistics based on Joe's pizza sales.



DecrementCube : Select Query

| Pizza | Sum of Amount |
|---|---|
| Pizza Alla Romana | 2109 |
| Pizza Bufalina | 969 |
| Pizza Margherita | 2372 |
| Pizza Pollo e Pesto | 84 |
| Pizza Quattro Stagioni | 740 |
| Pizza Salsiccia | 1444 |
| Pizza Vegetariana | 3366 |
| Grand Total | 11084 |

**Fig. 101.** Pizza sales

It is probably easier to see the results if we present them as a bar chart in Fig. 102.

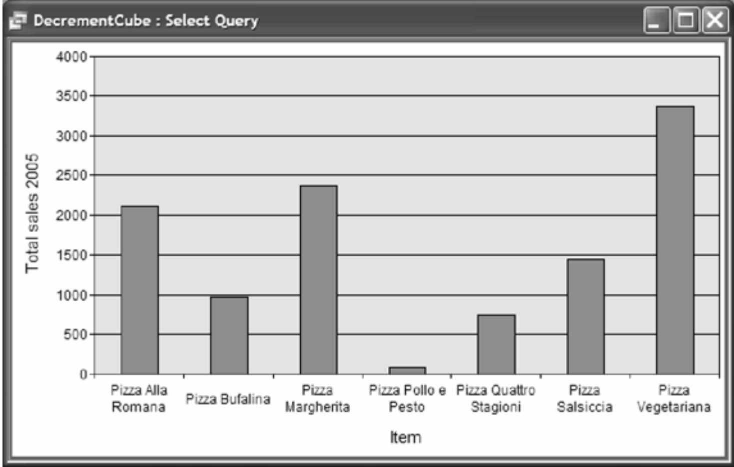



**Fig. 102.** Pizza sales bar chart

Based on these figures Joe should probably remove the *Pizza Pollo e Pesto* from his menu and instead consider adding more vegetarian pizzas.

We can also have a look at the increment events in Fig. 103.

IncrementCube : Select Query

| Years | Months | Sum of Amount |
|---|---|---|
| ⊟ 2005 | ⊞ Jan | 2404.85 |
| | ⊞ Feb | 5114.95 |
| | ⊞ Mar | 7177.7 |
| | ⊞ Apr | 9402.2 |
| | ⊞ May | 9399.85 |
| | ⊞ Jun | 10147.15 |
| | ⊞ Jul | 10135.35 |
| | ⊞ Aug | 11054.8 |
| | ⊞ Sep | 9664.8 |
| | ⊞ Oct | 10177.2 |
| | ⊞ Nov | 12057.25 |
| | ⊞ Dec | 12390.95 |
| | Total | 109127.05 |
| Grand Total | | 109127.05 |

**Fig. 103.** Cash receipts

Again let's look at the data as a bar chart in Fig. 104.

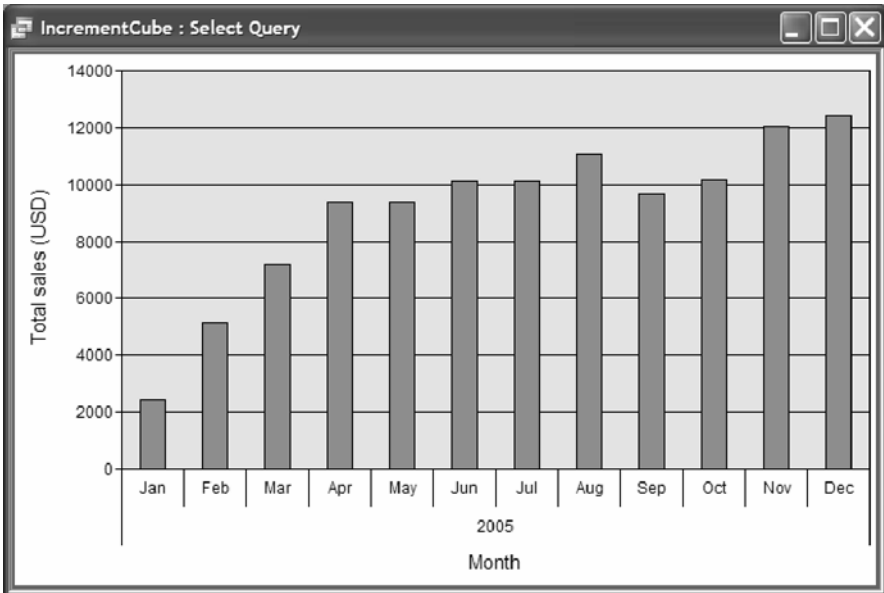



**Fig. 104.** Cash receipts as bar chart

All in all it looks as if things are going well for Joe: sales have been steadily increasing over the year.

## 3.10 Conclusions

Hopefully this example application has shown that it is indeed simple to develop an REA-based business application. The main benefits of doing so are:

- By basing the domain model on a proven and well-understood core model (the REA model) we minimize the risk of design flaws in our application. By demanding that all domain classes inherit from base classes in the REA model we are able to perform a design-time check that the domain model is consistent.
- Due to the fact that we base our domain model on a model that covers a larger set of business cases than the domain model itself, it is relatively easy to extend the domain model at a later time. If for instance Joe decides to track usage of raw materials for making pizza, we know that this will easily fit into the model.

- Because much of the business logic resides in the reusable REA model we can minimize the development effort. In the example application we were able to create a complete domain model for the pizza sales application with only 28 lines of code.

While I strongly recommend that you start using the REA-model there are of course also some caveats that you need to take into consideration:

- If you are developing an application that really is not about resources, events and agents (for instance a document management system), you may end up spending a lot of time trying to "shoehorn" the application into the REA model. It is important to decide early on whether the REA model is applicable.
- While the REA model is very powerful it is also very abstract. If you try to explain your design to a customer or fellow employee, you may find that explaining the underlying REA model is difficult. Trying to hide the fact that you are basing your design on an REA model may also be a bad idea, because major design decisions are based on the decision to use REA (e.g., why should the customer ID be placed on each order line instead of on the order itself).