

Object-Oriented Modeling with OptimJ

OptimJ is a radically new optimization modeling language designed as a Java extension with Eclipse integration, in contrast with all existing modeling languages that are home-brewed domain-specific languages.

In this whitepaper, we show how the object-oriented techniques inherited from Java enable abstraction and reuse of optimization code.

OptimJ technical overview

OptimJ™ is an extension of the Java™ programming language with language support for writing optimization models and powerful abstractions for bulk data processing. The language is supported by programming tools under the Eclipse™ 3.2 environment.

- OptimJ is a programming language :
 - OptimJ is an extension of Java 5
 - OptimJ operates directly on Java objects and can be combined with any other Java classes
 - The whole Java library is directly available from OptimJ
 - OptimJ is interoperable with standard Java-based programming tools such as team collaboration, unit testing or interface design.
- OptimJ is a modeling language :
 - All concepts found in modeling languages such as AIMMS™, AMPL™, GAMST™, MOSEL™, MPL™, OPL™, etc., are expressible in OptimJ.
 - OptimJ can target any optimization engine offering a C or Java API.
- OptimJ is part of a product line of numerous domain-specific Java language extensions, a novel approach of “*integration at the language level*” made possible by Ateji proprietary technology.



The Diet Model

Our running example will be the Diet model, a canonical example that appears in the first pages of most textbook about optimization techniques. The diet problem consists in finding an appropriate mix of foods satisfying minimum and maximum weekly intakes of vitamins, for a minimal cost.

Textbook version

Here is how the Diet model is typically presented in a textbook. We have adapted the version given in *"The AMPL book"* to the OptimJ syntax, changing a few surface details but keeping the same overall presentation. First we have the model data:

```
// vitamin names
String[] VTMN = { "A", "B1", "B2", "C" };
// food names
String[] FOOD = { "BEEF", "CHK", "FISH", "HAM",
                  "MCH", "MTL", "SPG", "TUR" };
// cost of each food
double[] cost = { 3.19, 2.59, 2.29, 2.89,
                  1.89, 1.99, 1.99, 2.49 };
// minimum and maximum intakes of each food
float[] f_min = { 0, 0, 0, 0, 0, 0, 0, 0 };
float[] f_max = { 100, 100, 100, 100, 100, 100, 100, 100 };
// minimum and maximum intakes of each vitamin
float[] n_min = { 700, 700, 700, 700 };
float[] n_max = { 10000, 10000, 10000, 10000 };
// amount of each vitamin brought by each food
float[][] amt = {
    { 60, 8, 8, 40, 15, 70, 25, 60 },
    { 20, 0, 10, 40, 35, 30, 50, 20 },
    { 10, 20, 15, 35, 15, 15, 25, 15 },
    { 15, 20, 10, 10, 15, 15, 15, 10 }};
```

Then the decision variables, constraints and objective:

```
var double[] Buy[int j : FOOD] in f_min[j] .. f_max[j];

minimize
    sum {int j : FOOD} { cost[j] * Buy[j] };

constraints {
    forall(int i : VTMN) {
        n_min[i] <= sum {int j : FOOD} { amt[i][j] * Buy[j] };
        sum {int j : FOOD} { amt[i][j] * Buy[j] } <= n_max[i];
    }
}
```

This formulation is fine for a textbook and is a perfect example of the power of algebraic modeling languages: it is concise while easily readable, close to the mathematical formulation of the problem.

One thing that is missing in the initial textbook version is the code to actually solve this model and print a solution. In OptimJ, a **model** is just a specialized Java **class**, and like any other class it can have fields, methods and constructors. First we define a **model** containing the data and the problem given above, and provide this code inside a **main()** method as follows:

```
model DietModel // a model a just a special kind of class
  solver lpsolve // the solver name is provided here
{
  ... data ...
  ... variables, constraints and objective ...

  public static void main(String[] args)
  {
    // instanciate the model (this is like instanciating a class)
    DietModel m = new DietModel();

    // extract the model into the solver
    m.extract();

    // solve the model
    if (m.solve()) {
      // print the objective value
      System.out.println("objective value " + m.objValue());

      // print the value of each variable
      for(int j : m.FOOD) {
        System.out.println(
          m.FOOD[j] +
          " " +
          m.value(m.Buy[j])
        );
      }
    } else {
      System.out.println("no solution");
    }
  }
}
```

Identify model parameters

In the previous version, all model parameters are given inline as field initializations in the source code:

- no difference is made between which fields are parameters and which are not
- the only parameter-passing mechanism is modifying fields from the outside

Indeed, this is how today's modeling languages are designed. Most are interpreted languages where parameter-passing is actually a text-replacement operation, generating a new source code with different inline parameters, after what the whole source code is interpreted again.

When a mechanism is available for modifying fields from the outside (this is required for instance in column generation algorithms where one must deal with multiple instances of multiple models), this leads to spaghetti code where it quickly becomes impossible to track what has been initialized or not. This is a sure recipe for introducing very nasty bugs.

The parameter-passing mechanism of OptimJ is the same as in Java: define a constructor for the model.

```
// define a constructor for the diet model
DietModel(
    String[] VTMN,
    String[] FOOD,
    double[] cost,
    float[] f_min,
    float[] f_max,
    float[] n_min,
    float[] n_max,
    float[][] amt)
{
    ...
}
```

Now the model parameters are not given inline in the model code anymore, but passed via the constructor. As a first example, we define them inline in the **main()** method:

```
public static void main(String[] args)
{
    // model parameters are now defined here
    // rather than inside the model
    String[] VTMN = { "A", "B1", "B2", "C" };
    String[] FOOD = { "BEEF", "CHK", "FISH", "HAM",
                     "MCH", "MTL", "SPG", "TUR" };
    double[] cost = { 3.19, 2.59, 2.29, 2.89,
                     1.89, 1.99, 1.99, 2.49 };
    float[] f_min = { 0, 0, 0, 0, 0, 0, 0, 0 };
    float[] f_max = { 100, 100, 100, 100, 100, 100, 100, 100 };
    float[] n_min = { 700, 700, 700, 700 };
    float[] n_max = { 10000, 10000, 10000, 10000 };
    float[][] amt = {
        { 60, 8, 8, 40, 15, 70, 25, 60 },
        { 20, 0, 10, 40, 35, 30, 50, 20 },
        { 10, 20, 15, 35, 15, 15, 25, 15 },
        { 15, 20, 10, 10, 15, 15, 15, 10 } };

    // the parameters are now passed explicitly
    // to the model via the constructor call
    DietModel m = new DietModel(VTMN, FOOD, cost, f_min,
                                f_max, n_min, n_max, amt);

    ... same as before ...
}
```

Abstract the model parameters

The diet model constructor now takes eight different parameters. This doesn't sound right, as the eight parameters are closely related. A better formulation is to group all parameters of the diet model into a **DietData** class:

```
public class DietData
{
    final String[] VTMN;
    final String[] FOOD;
    final double[] cost;
    final float[] f_min;
    final float[] f_max;
    final float[] n_min;
    final float[] n_max;
    final float[][] amt;

    ... class constructors come here ...
}
```

The **final** keywords in this example are not mandatory but are a good programming practice. They allow the compiler to check that everything has been initialized properly, thus helping catching potential bugs early. The model constructor must be updated in order to accommodate the change:

```
public DietModel(DietData d)
{
    this.d = d;
}
```

All parameters must now be prefixed, for instance **FOOD** is replaced with **d.FOOD**.

```
var double[] Buy[int j : d.FOOD] in d.f_min[j] .. d.f_max[j];
```

An alternative approach that does not require prefixing is to cache a local copy of the parameters in fields. Again, the **final** keyword is here for ensuring that everything is properly initialized:

```
public DietModel(DietData d)
{
    // cache a local copy of the parameters
    final FOOD = d.FOOD;
    final f_min = d.f_min;
    final f_max = d.f_max;
    ...
}

var double[] Buy[int j : FOOD] in f_min[j] .. f_max[j];
```

Access model data using any Java API

Now that the model parameters have been abstracted away from the model, adding a new data source is simply a matter of writing a new constructor for the `DietData` class. There will be no change in the model itself.

In the following sections, we show three examples where model data is read from three different classical data sources, using standard Java APIs:

- JDBC for accessing SQL databases
- HSSF for accessing Excel sheets
- SAX for accessing XML files

Of course any other available Java API, whether standard or home-grown, can be used as well.

Sending solutions back to the application is done in a similar way using the same APIs, we have omitted this part for brevity.

Abstracting away the diet model parameters into a `DietData` class marks the limit between the 'optimization' part and the 'programming' part. A properly designed model data class provides a perfect means of communication between an optimization expert and a programming expert.

Import data from an SQL database

If this example, the diet model data is stored in a SQL database as three tables 'Foods', 'Amounts' and 'Vitamins'. Here we create a new constructor for the `DietData` class using the standard JDBC API.

```
public DietData(Connection c) throws SQLException
{
    // Read table 'Foods'
    int i = 0;
    Statement statement = c.createStatement();
    ResultSet resultSet = statement.executeQuery(
        "SELECT * FROM Foods");
    statement.close();
    while (resultSet.next()) {
        FOOD[i] = resultSet.getString("food");
        cost[i] = resultSet.getDouble("cost");
        f_min[i] = resultSet.getFloat("fmin");
        f_max[i] = resultSet.getFloat("fmax");
        i++;
    }
    resultSet.close();

    // Read table 'Amounts'
    ... similar code, omitted for brevity ...

    // Read table 'Vitamins'
    ... similar code, omitted for brevity ...
}
```

Import data from an Excel sheet

In this example, the model data is stored in an Excel sheet as shown below:

	A	B	C	D	E	F	G	H	I	J	K	L
14												
15		Foods	Cost			Minimum	Maximum		A	B1	B2	C
16		(Foods)	(Cost)			(FMin)	(FMax)		(Amount)			
17		Beef	\$3,19			0	100		60	10	15	20
18		Chicken	\$2,59			0	100		8	15	20	0
19		Fish	\$2,29			0	100		8	15	10	0
20		Ham	\$2,89			0	100		40	35	10	40
21		Macaroni	\$1,89			0	100		15	15	15	35
22		Meat Loaf	\$1,99			0	100		70	15	15	30
23		Spaghetti	\$1,99			0	100		25	25	15	50
24		Turkey	\$2,49			0	100		60	15	10	60
25												
26		Vitamins				Minimum	Maximum					
27		(Vitamins)				(VMin)	(VMax)					
28		A				700	10000					
29		B1				700	10000					
30		B2				700	10000					
31		C				700	10000					

Having our diet model read data from this sheet is simply a matter of adding the appropriate constructor in the DietData class. Here we use the standard HSSF API from the Apache project.

```
class DietData
{
    // A new constructor to read data from an Excel sheet
    // based on the HSSF API (wb is a reference to the sheet)
    DietData(HSSFWorkbook wb)
    {
        VTMN = HSSFArea.make(wb, "Vitamins").toStringArray();
        FOOD = HSSFArea.make(wb, "Foods").toStringArray();
        cost = HSSFArea.make(wb, "Cost").toDoubleArray();
        f_min = HSSFArea.make(wb, "FMin").toFloatArray();
        f_max = HSSFArea.make(wb, "FMax").toFloatArray();
        n_min = HSSFArea.make(wb, "VMin").toFloatArray();
        n_max = HSSFArea.make(wb, "VMax").toFloatArray();
        amt = HSSFArea.transpose(HSSFArea.make(wb, "Amount")
            .toFloatArrayArray());
    }

    ... as before ...
}
```

Import data from an XML file

Once again, we define an additional constructor to the DietData class, using a standard Java XML API.

```
public DietData(Document document)
{
    NodeList nodes = document.getElementsByTagName("vitamins");

    // locate 'vitamin' nodes and read their properties
    int pos = 0;
    for (int i = 0; i < nodes.getLength(); i++) {
        NodeList vitamins = nodes.item(i).getChildNodes();
        for (int j = 0; j < vitamins.getLength(); j++) {
            Node v= vitamins.item(j);
            if (v.hasAttributes()) {
                NamedNodeMap attributes = v.getAttributes();
                VTMN[pos] = v.getNodeName();
                n_min[pos] = new Float(attributes
                    .getNamedItem("vmin").getNodeValue());
                n_max[pos] = new Float(attributes
                    .getNamedItem("vmax").getNodeValue());
                pos++;
            }
        }
    }

    ... similar code for amounts and vitamins ...
}
```

Abstract the logical structure

A further improvement, typical of object-oriented programming, is to abstract over the logical structure of the model parameters.

Our model parameters consist of eight different arrays with no apparent structure of relationship. The fact that the *i*-th entry in the **f_min** array is the minimum amount for the *i*-th entry in the **FOOD** array is stated nowhere. It is only implicit in the mind of the developer.

This kind of formulation is favored by mathematicians: it is concise, and the possible semantic ambiguities are not considered relevant because of the small size of problems.

From a software engineering perspective however, this is terrible:

- what is implicit in the mind of a programmer may not be implicit, or differently implicit, in the mind of its colleague
- the large size of programs makes it difficult to keep so many implicit assumptions in mind
- any information that is not made explicit cannot be used by the compiler or other software development tools, for instance checking for common mistakes or performing high-level code optimizations

Let us try to improve things. The first question to ask is *"what is vitamin"* ? In an object-oriented perspective, a vitamin is defined by its observable properties. A vitamin has a name, a minimal and a maximal amount. In Java, this is expressed with an interface:

```
interface Vitamin
{
    String name();
    float minAmount();
    float maxAmount();
}
```

An interface has the additional advantage of not committing to a particular implementation choice. A food is defined similarly:

```
interface Food
{
    String name();
    double cost();
    float minQuantity();
    float maxQuantity();
    float amount(Vitamin v);
}
```

The model data now consists simply of a collection of vitamins and a collection of foods:

An Object-Oriented diet model

With the abstractions we have just defined, we can now write our diet model in a truly object-oriented way:

```
var double[] Buy[int j : d.foods]
    in d.foods[j].minQuantity() .. d.foods[j].maxQuantity();

minimize
    sum {int j : d.foods} { d.foods[j].cost() * Buy[j] };

constraints {
    forall(Vitamin v : d.vitamins) {
        v.minAmount() <= sum {int j : d.foods}
            {d.foods[j].amount(v) * Buy[j]};
        sum {int j : d.foods} {d.foods[j].amount(v) * Buy[j]}
            <= v.maxAmount();
    }
}
```

The model is still the same as the initial one, but its formulation does not use hard-wired data structures and arrays anymore. This independence from data representation is what allows us to input the model data from any source that can be modeled in Java.



Try OptimJ

The complete code of the examples shown here is included in the OptimJ samples libraries, available for download at www.ateji.com, where you can also request a free evaluation licence of OptimJ.

Ateji helps you jump-start your first OptimJ projects by providing consulting and education services as needed. Contact us at info@ateji.com