Transformation Interface

-Andy Bulka, August 2001
abulka@netspace.net.au

Abstract

The Transformation Interface is both a design pattern and an architectural pattern. It is an *interface* or *layer*, which offers services to transform an object into another format and back again. A pair of methods on some class e.g. `load/save` are often the signature of a Transformation Interface.

The 'impedence mismatch' between class instances in memory and relational databases can be viewed as an object-to-table format 'transformation' problem. In fact, persisting an object to disk and back again is one of the most common transformations - as is 'transforming' an object to and from a human manipulable/editable format in a GUI. Other examples of object transformations are: saving and loading an object to and from XML format, saving/restoring or sending/receiving an object using streaming or serialization, encryption/decryption, and perhaps even making and restoring a memento of an object.

Because there are so many implementations of such an abstract idea of 'transformation', I have set up the context to be quite specific (implementing a three-tiered architecture of GUI-to-Model mapping and Model-to-Persistent storage mapping) – in order to more concretely explain the pattern to the reader. Consequently, this paper focuses on how the Transformation Interface pattern resolves the forces that appear in the context of implementing such a three-tiered architecture, rather than discussing too much the nature of the transformation interface itself, or in studying examples of all its incarnations.

This paper includes working code to both a persistence layer and a GUI display mechanism - both implementations of the Transformation Interface Pattern, which can then be 'plugged together' to form the main hub of a complete application architecture.

Context

You are writing a number of applications, of different sizes, some in different languages and operating systems, all of which require that require that certain objects be saved to disk and also be displayed in a GUI. You are looking for a pattern that describes a general approach to implementing these 'core transformations'.

Problem

How do you design the transformation of an object into persistent, graphical and potentially other formats, in a way that is abstract and language/class-library neutral so that you have the flexibility to take advantage of the language facilities and class libraries available in each particular implementation context?

You want to implement any such solution as simply as possible.

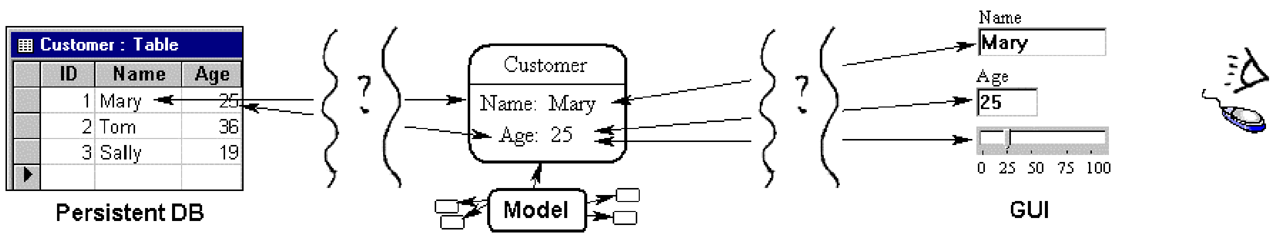


Figure 1. How do we implementing the mapping layers required by the above application architecture?

Forces

- **Necessity:** Implementing both persistence of ‘model’ objects and the display of objects in a GUI are core ‘transformational’ tasks - one or both are usually necessary in almost all software applications. You want to minimise the time spent implementing these necessary, common tasks.

- **Complexity**: When implementing a presentation layer (e.g. rendering object attributes in a visual way), a key requirement is to keep graphical knowledge out of the model, decoupling the model data and GUI. The Model-View-Controller (MVC) pattern achieves this using a sophisticated set of classes and behaviour. Similarly, when implementing a persistence layer, there are numerous complexities to consider.

  Thus both MVC-like and Persistence frameworks are notoriously complex to design and implement, and even obtaining and learning third party frameworks is work that you would rather avoid or keep to a minimum.

  You are prepared to forgo the optimal efficiency of an ideal but possibly complex solution for the simplicity of a possibly less efficient solution.

- **Data-awareness:** Off-the-shelf data-aware GUI controls are easy to implement, requiring no code - providing both persistence and GUI display. However, off-the-shelf data-aware GUI controls usually do not map naturally to application domain objects residing in memory, rather they map to and are restricted to displaying persisted, relational database table data. They also may be difficult to ‘tame’ when applications and display requirements get more complicated. Data-aware-frameworks and off-the-shelf funky data-aware widgets are not guaranteed to be available in all your programming contexts.

  Building custom data-aware controls and supporting frameworks solve some of these limitations – allowing data-aware controls that work with your particular domain object model. Such custom data-aware solutions usually require substantial design and implementation overhead.

- **Transparency and Control**: Code that painstakingly maps attributes of one format into the other format is boring to write (since it is often repetitive and detailed work) however it gives the programmer full and total control over when and how the mapping behaviour occurs. Data-aware control frameworks attempt to automate such specific mapping code, but almost always introduce some mystery into the data-to-GUI synchronisation process (unless the documentation and implementation is very good).

- **Portability and Adaptability**: Any single persistence framework or GUI display solution is unlikely to be portable to all contexts, languages, operating systems and projects faced in a programmer's career. However an abstract solution in the form of a design pattern could be used in all contexts where the GUI and persistence transformation problems occur.

Forces involved in data-aware solutions

Relational database-style data-aware controls are often compelling solutions, and may be all that is required in certain contexts, for example, where portability and transparency forces are weak, and you are only dealing with persistence to relational databases.

However, one day you might be writing a stockmarket simulator in the Python language, which requires objects to be streamed to file, and requires the complex display of objects in a Java swing GUI under Linux and then later requires you to re-implement the display of those same objects in a Delphi GUI form under Windows. In this example, the forces of 'adaptibility' and 'control' are strong, and the 'data-aware' force is weak and thus traditional data-aware controls are not going to adequately resolve the forces.

Another major issue with data-aware controls is that in such frameworks, database tables, not objects are first class citizens - you don't even need to define classes to use data-aware frameworks (but of course you need to define your tables). This complicates the task of an object oriented developer who wants to model his or her domain by defining classes with both attributes and behaviour, and wants to make these classes first class citizens within their overall design. Ironically the developer usually proceeds by demoting the relational database to a mere storage facility and is forced to implement a persistence layer which transforms (loads/saves) the objects in memory to rows in tables. Data aware controls could still conceivably be used, as long as all relevant objects are persisted to disk immediately before the data aware GUI form is invoked.

Solution

Define an interface featuring a pair of *transformational methods.* One method converts the attributes of an object into the foreign format. The second method converts the foreign format back into attributes of the object. Maintain a reference (e.g. pointer or integer) to the other format in either the original object or the transformed object (e.g. a domain object in memory might store an integer ID reference of the record in a table that persists that domain object).

The two methods of the class implementing the Transformation Interface will contain all the (possibly detailed) mapping code required to transform an object into another format.

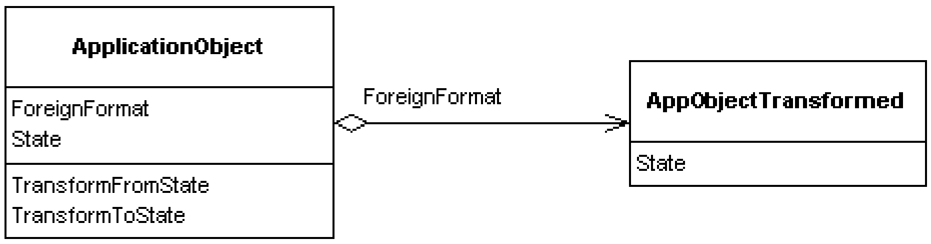


Figure 2. Transformation Interface Pattern.
The object on the left can be transformed into the object on the right, and back again.

On which class to implement the Transformation Interface

You have some flexibility as to where to put the Transformation Interface mapping implementation (the two methods and property). In our working implementations, below, the two persistence layer transformation interface methods (save/load) for the business object are implemented on the *business object.* However, the presentation layer transformation interface methods (display/repopulate) are implemented on the *form (*see *figure 4).*

There are numerous design alternatives available, depending on your situation. For example we can instead use an *adapter* class to implement transformation interface methods. *Figure 5* illustrates using a mediating adaptor class to implement the GUI transformation interface methods on behalf of the form. Even the persistence transformation interface methods of business objects could alternatively be implemented on a separate 'writer' class - that knows about the internals of the class that it is persisting/transforming. (See the related patterns section, Serializer pattern, for an even more generic (but more complex) way of implementing a writer class).

Regardless of this implementation flexibility, there are serious considerations to keep in mind when deciding where to put the transformational interface. For example since you want to keep knowledge of the GUI out of the model, you would not implement a GUI transformational interface on a business object. However a *persistence* transformation interface *is* often implemented on the business object - though this is not as objectionable since how an object correctly persists itself is arguably a direct concern of the business object. In any case, such persistence functionality can alternatively be implemented on an adapter object, as mentioned above.

Thus one way that the mapping layers required in *Figure 1.* could be implemented by having both the business object and GUI form implement Transformation Interfaces.

Class Feature	*Description*
Id	Pointer to some foreign format e.g. an integer database ID or pointer to a GUI widget
TransfromTo()	Transformation method e.g. `saveToDb` () or `displayObjectInGui() or encrypt()`
TransformFrom()	Transformation method e.g. `loadFromDb` () or `rePopulateObjectFromGUI() or decrypt()`
State	The data attribute(s) of the business object e.g. Name, Phone, Address etc.

Figure 3. A transformation interface consists of two methods and one attribute.

Examples of Transformations

- From an object in memory to a record in a table on disk
  e.g. `load/save`

- From an object to a GUI representation of that object
  e.g. `display/repopulate`

- From an string attribute of an object to a text control widget in a GUI
  e.g. part of the mapping code inside `display/repopulate`

- A Delphi `TStringList` class streams its contents to and from a file
  e.g. `saveToFile/loadFromFile`

- A Java class is serialised to a data stream
  e.g. `Serialiser.store(obj, astream) / obj = Serialiser.load(astream)`

- An object into its memento
  e.g. `CreateMemento() / SetMemento( m )`

Consequences

Resolution of Forces

All the forces are adequately resolved by this pattern (though see limitations section, below). The necessary persistence and GUI functionality has been implemented relatively simply compared with potentially complex MVC-like and persistence-layer schemes. We have avoided non portable, proprietary data-aware components in favor of a flexible, homegrown and fully OO design.

The force of total transparency and control is satisfied because the exact code that achieves the transformation is visible to the programmer and fully encapsulated inside the two transformation methods.

The solution is pragmatic, quick to build and easy to debug. It is a common solution used in the real world – as opposed to theoretically optimal solutions that are never actually implemented. For example, you might like a sophisticated MVC (Model View Controller) framework in your next application, but you don't have the time to build and debug it, so you use Transformation Interface, which usually does the job (subject to various limitations, below).

Limitations

Code Maintenance: Transformation Interface method code, whilst transparent, needs to be created and maintained manually, which is tedious work. However it could be argued that these 'mappings' need to be specificied in *some* way – whether by lines of code, or by entering information into dialog boxes, or even by drawing and dragging. So as long as the implementation code is syntactically economical (which it usually is – consisting of multiple, simple, one line mapping statements e.g. `dbrec.AGE := self.Age`) then there is no redundancy or extra work in maintaining code manually. Nevertheless, some sort of automated code generation approach would probably be welcome where there are large amounts of straightforward mappings between standard data types.

Granularity: A limitation of a single pair of transformational methods is that they act at a high level of granularity – e.g. they display *all* the fields in a form, or they save *all* the fields of a record in a database – whether the object attributes has changed or not. More sophisticated transformational methods could be smart enough to check for ‘changed’ and ‘dirty’ flags to avoid unecessary work. In addition, you might consider factoring the transformation method behaviour into sub-methods which are intelligently called as required. You could even implement Observer somewhere in your design and communicate *exactly* what changed to the transformation method (e.g. as a parameter). All these enhancements introduce additional complexity – so unless the context demands optimisation, it is usually sufficient to use the original simple, brute force approach and map all attributes whether they have changed or not.

Modality Issues: The above granularity considerations are closely related to issues of ‘modality’. The basic Transformation Interface solution assumes that GUI forms are displayed *modally*, or that we have *exclusive access* to the database records we are writing to. For example, by displaying a GUI form modally we guarantee that the information in the form will not change whilst the user has the form open. Usually the modality requirement is not a limitation – in fact modal forms are common, as is the requirement for exclusive access to database records. Even other transformations like streaming and serialization usually require exclusive access to a data streams or to the files that are being written to.

However if your application includes some sort of real-time display, or the design allows two GUI forms representing the same model data to be active at the one time, then a fine-grained, MVC (Model-View-Controller)-like ‘observer’ based design might be worth considering so that every time an individual business object attribute changes, the relevant GUI control widget will be notified and updated. This is not just a granularity issue (making transformations as small and as efficient as possible), but a ‘timing of update’ issue (*when* are the transformations performed).

Treating forms modally on the other hand keeps things very simple – opening the form causes the transformation to the GUI, and hitting OK triggers the reverse transformation.

Implementation

Implementations of this pattern involve adding two methods and one reference (e.g. pointer field/property) to the appropriate class (*see solutions section, above, for a discussion on how to pick this class*). Typically you would rename the methods to suit the mapping situation. Three styles of implementation are also available

- just add the methods and property to the appropriate class
- as above, but define the methods as implementing an *abstract class* or *interface*
- add the methods and property to the base class and override these methods in sub-classes

The last two implementation styles allow polymorphic iteration over descendants of the base class *e.g.* a `ModelManager` object opens a data stream or file, then iterates through all the business objects in its collection, calling `.save` on each object.

Initialization

Before displaying information in a form, the form usually needs to be created and displayed. Before writing an object’s attribute information to a database, the database needs to be correctly opened. Simlar issues arise with files and streams etc. These initializations can be placed in client code, manager objects, constructor methods or separate methods (static or otherwise), depending on the circumstances. In the example presentation layer shown later in this paper (*figure 6*), the adaptor class which implements the Transformation Interface creates the form in its contructor method.

Attribute Mapping

The key issue in a two-way mapping is how to send the data (attributes of an object) from one layer or format to another. This is handled by the code in the transformation methods – *that is the sole purpose of transformation methods*. The implementing programmer needs to type in the necessary code to painstakingly map each attribute of the object onto the other format. The code in the transformation methods can be relatively straightforward or can encapsulate and hide the possibly fiddly details of the transformation.

Encapsulating complex mappings

One of the benefits of complete transparency of implementation is the ability to encapsulate any messy mapping code under the ultra-simplicity of *a pair of transformational methods*.

For example, a `load` transformational method, when reading an integer field from a database, may need to validate that the value makes sense then map/cast it into an enumerated type that the business object actually uses. The transformational method in the other direction will map the enumerated type back into a regular integer to be stored in the database.

Similarly, a pair of GUI transformational methods would contain the either straightforward or fiddly code to represent object attributes in widgets of some sort, and back again – the details of which will vary depending on the GUI widgets involved.

Furthermore, complex mappings might require that aspects of a single object attribute be displayed in separate GUI widgets. Conversely, a number of object attributes might be combined and displayed in a single GUI widget (e.g. a series of numbers plotted on a graph/canvas widget). Whether the code is simple or complex, it is encapsulated in the transformational methods, and the programmer knows where to go to modify and control the mapping behaviour.

Foreign keys and pointers

A further example of how transformation methods can encapsulating complex mappings is the case of say, loading a composite object from a database. The `.LoadFromDb` transformation method for the composite class e.g. `Customer` would house the code necessary to convert, the database table record into an object. Any aggregated sub-object e.g. `CustomerAddress` would be referred to using a foreign key (an integer id). So in order to fully construct the composite domain object e.g. `Customer`, both the actual object (`Customer`) and the object it aggregates (`CustomerAddress`) will need to be created and a pointer set from the composite object to the aggregated sub-object. In addition, a number of database queries need to be made in order to extract all the necessary data for both `CustomerAddress` and `CustomerAddress.` The transformation method can encapsulate all that needs to be done.

Note that ideally, such transformational methods would typically call other transformational methods to load in the sub-objects rather than trying to do everything themselves – this is just common sense factoring. e.g. If class `Customer` aggregates the class `CustomerAddress`, then `Customer.LoadFromDb` will at some stage call `CustomerAddress.LoadFromDb.` One needs to be careful to transform/load objects in the right order, since loading one object may rely on another object already being loaded. Some of the considerations discussed here are similar to those of ‘deep streaming’ (see Serializer pattern).

Introducing Observer

To get ‘real-time’ updates of changes to a foreign format’s state, (e.g. a form want to be notified when there are changes to a business object attributes), the class implementing the transformation interface should also implement Observer. It can then subscribe and be notified of changes to the foreign format. To take advantage of this notification process, the transformation method should be modified to take a parameter telling it what changed e.g. `display( somespec )`, so that the `display` method doesn’t work hard updating *all* parts of the state – only what changed..

In a GUI display scenario, another alternative way of implementing more granular and efficient transformations (mappings / screen updates) is to create an adapter object for each business object or even for every business object attribute, and create appropriate manager objects to coordinate all the adaptors. Here we start to equal fine grained MVC-like efficiency, but begin to take on some of its complexity. On the positive side, by using Transformation Interface throughout these more complex composite designs, you help to reduce their complexity a little - for example, by insisting that all relevant classes implement standard transformation method pairs.

The “Persistence Layer” Transformation

Running Example of a Persistence Layer

The following is a small but fully working implementation of a persistence layer. The Transformation Interface is implemented on the business class as a pair of load/save methods, which are overloaded in the `Customer` and `Supplier` classes. Written in the pseudo-code-like, syntactically minimalist language Python: (See *Appendix* for easy tips on reading Python)

```
import shelve, string

class PersistenceTransformationInterface:            # Abstract
    def save(self):
        pass
    def load(self):
        pass
    def getId(self):
        pass

class DbPersistenceTI( PersistenceTransformationInterface ):
    NextKeyId = 1     # class variable
    def AllocateNextId(self):
        result = DbPersistenceTI.NextKeyId
        DbPersistenceTI.NextKeyId += 1
        return str(result)
    def getId(self):
        return self.dbKeyId

class BusinessObject( DbPersistenceTI ):
    def __init__(self):                              # Constructor
        self.dbKeyId = self.AllocateNextId() + ',' + \
                     self.__class__.__name__
```

```
class Model:
    def __init__(self, file):
        if not file:
            raise 'NoFileSpecified'
        self.Filename = file
        self.BOlist = []
    def Add(self, obj):
        self.BOlist.append(obj)
        return self
    def SaveAll(self):
        db = shelve.open(self.Filename,'c')
        for obj in self.BOlist:
            obj.save(db)
        db.close()
    def LoadAll(self):
        db = shelve.open(self.Filename,'r')
        self.BOlist = []
        keyz = db.keys()
        for dbkey in keyz:
            classref = string.split(dbkey,',')[1]
            obj = eval(classref)()    # Create appropriate object
            obj.dbKeyId = dbkey
            obj.load(db)
            self.BOlist.append(obj)
        db.close()
        return self

class Customer( BusinessObject ):
    def __init__(self, name=''):
        BusinessObject.__init__(self)    # Call inherited
constructor
        if name:
            self.FirstName, self.Surname = string.split(name)
    def save(self,db):
        db[ self.dbKeyId ] = (self.FirstName, self.Surname)
    def load(self,db):
        self.FirstName, self.Surname = db[ self.dbKeyId ]
```

To test drive the above code, we run the following:

```
    model = Model('mydata.dat')
    cust1 = model.Add( Customer('Fred Flinstone') )
    cust2 = model.Add( Customer('Anastasia Jones') )
    model.SaveAll()
```

then in a later session ....

```
    m = Model('mydata.dat').LoadAll()
    for businessObject in m.BOlist:
        print businessObject
```

which yeilds:

```
    <__main__.Customer instance at 01A10B9C> Anastasia Jones
    <__main__.Customer instance at 01A10B7C> Fred Flinstone
```

Success! We have just implemented a working persistence layer/framework using about *forty* lines of code (To aid clarity, `Model.Remove` is not listed, and the `Model.LoadAll()` lacks a few lines of code which sync the `DbPersistentTI.NextKeyId` to the maximum id read in, plus one). The above implementation includes an additional *ten* lines of code for the `Customer` class which is used in our test code, directly above).

How additional classes participate in persistence

To add new business classes that are capable of persisting themselves, descend a new class from class `BusinessObject` and add the four lines of code implementing the Transformation Interface methods (*e.g.* the load/save method pair). This is all that is required to participate in this persistence framework. Here is an implementation of the `Supplier` class:

```
class Supplier( BusinessObject ):
        def __init__(self, name='', address='', phone=''):
            BusinessObject.__init__(self)
            self.Companyname = name
            self.Address = address
            self.Phone = phone

        def save(self,db):
            db[ self.dbKeyId ] = (self.Companyname, self.Address, \
                                  self.Phone)
        def load(self,db):
            self.Companyname, self.Address, self.Phone = \
                                  db[ self.dbKeyId ]
```

Note that the Python `shelve` module provides database like access using language native dictionary like syntax e.g. `shelfFileObject[ stringkey ] = data`. Examine the `save/load` methods - above.

Test drive creating and persisting Customers and Suppliers with the following code:

```
    model = Model('mydata.dat')
    cust1 = model.Add( Customer('Fred Flinstone') )
    cust2 = model.Add( Customer('Anastasia Jones') )
    supplier = model.Add( Supplier('McDonald Farms Ltd.') )
    model.SaveAll()
```

Ok, our customers and supplier are created and have persisted to the file 'mydata.dat'. Later, to retrieve all customers out of the database and back into being objects in memory, we:

```
    model = Model('mydata.dat').LoadAll()
```

Then we can display all the customers in the model using a loop:

```
    for businessObject in model.BOlist:
        print businessObject
```

which yields:

```
    <__main__.Customer instance at 01A10A9C> Anastasia Jones
    <__main__.Customer instance at 01A1097C> Fred Flinstone
    <__main__.Supplier instance at 01A10A6C> McDonald Farms Ltd.
```

Two customers and one supplier object – correct!

The “Presentation Layer” GUI Transformation

Now we will implement the ‘mirror transformational task’ of providing a GUI for a business object. The object or model is ‘transformed’ to and from a human manipulable/editable format in a GUI (Graphical User Interface).

Sample Implementation #1 - Put the transformational methods on the form class

A common way to implement the GUI Transformation Interface is to add the required two methods and attribute to a GUI form class:

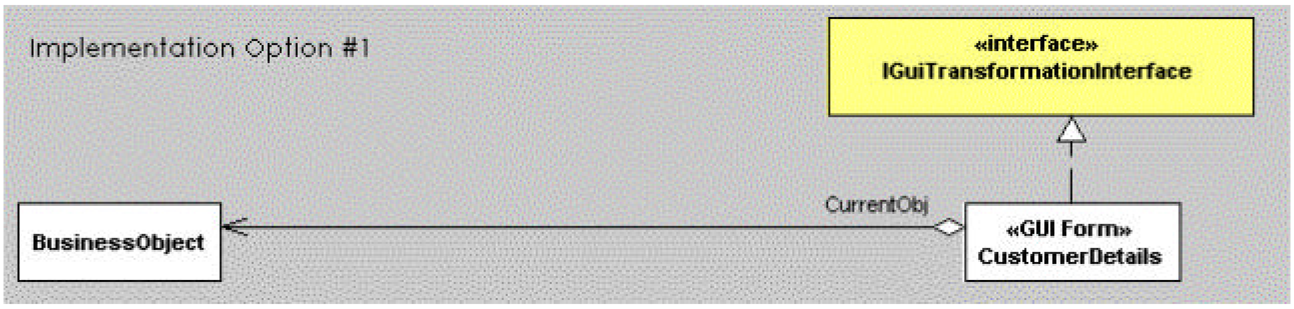


Figure 4. A GUI form implementing the transformation interface.

Typically you would also create a static class method for the form class that takes care of all the initialization details including the invocation/showing and disposal of the form. This initialization code might even be implemented inside the form’s constructor, taking as a parameter the business object(s) or model to display.

After creating/initializing the form, the startup method or constructor must eventually call the transformational method

```
  displayObjectInGui( businessobject )
```

in order to actually map the object attributes to particular widgets on the GUI form.

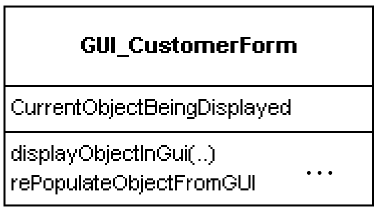


Figure 5. A GUI form class implementing the transformation interface.

After the user has edited the widgets on the form and clicks OK, the

```
  repopulateObjectFromGUI
```

transformational method should be called. This method will iterate through all the relevant widgets on the form, and write their values back into attributes of the business object. Then the form is closed.

If Cancel is clicked, then `repopulateObjectFromGUI` *is not* called, and the form is simply closed

Form as mediator

In this first, sample implementation of a persistence layer transformation, the form acts as a handy central mediator, providing a shared namespace, housing the transformation methods (usually implemented as methods of the form class) and also owning all the form widgets. The form usually also houses the event methods that respond to user interactions. Alternatively, some of these roles can be taken over by an adaptor class (see next working example, implementation #2).

Working Example Implementation #2 - Put the transformational methods on mediating adapter class

Sometimes it is more convenient to place the transformation interface on a separate mediating adapter object rather than on the form itself:

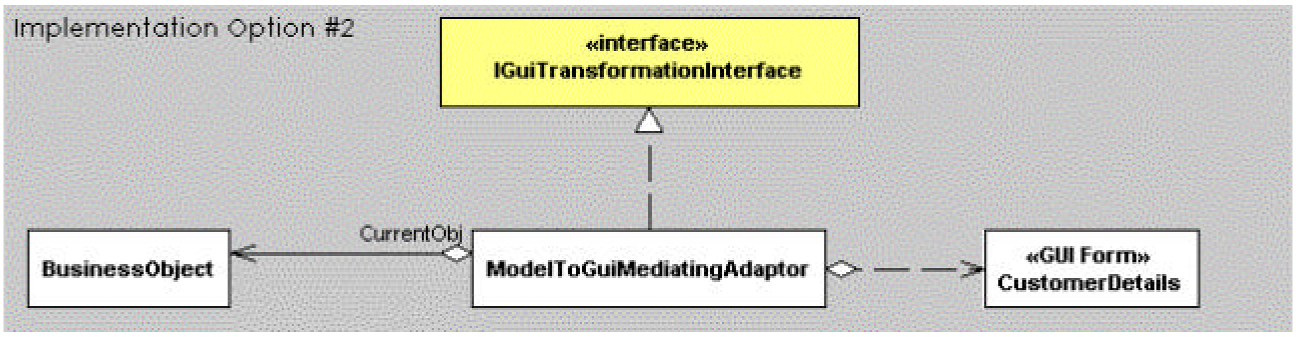


Figure 6. Rather than implementing the transformation interface on the GUI form itself, the transformation interface can be implemented by a mediating adapter class.

Use an adapter class when you cannot or choose not to sub-class forms. For example, your forms or GUI widgets may be difficult to subclass. Perhaps you want the flexibility of being able to change adapters without affecting forms.

The `ModelToGuiMediatingAdapter` class implements the Transformation Interface, and thus features two transformational methods which encapsulate all the (possibly complex) code for mapping attributes of the business object to particular widgets on the GUI form – and back again.

Optionally, if the adaptor class also implements Observer, it can be notified of specific attribute changes in the business object or model (See MGM pattern). The following implementation code works *without* the need for Observer, using only the simple Transformation Interface pattern. Despite the brevity of the code, this code is fully functional, and fits on less than a page.

Running Example of a Presentation Layer

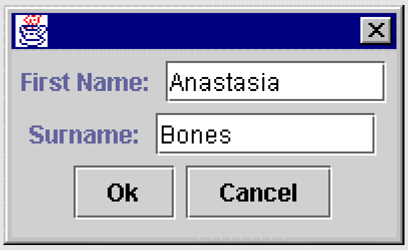
The following is an implementation of a GUI display framework that uses the simple Transformation Interface approach. A customer object is created and a form is displayed in which the user can edit the attributes of that customer business object.

The following code is written in the clear, syntactically minimalist language of JPython - which is Python running on a Java VM, with seamless access to all Java classes, including swing! (See *Appendix* for easy tips on reading Python code).

```
class GuiTransformationInterface:
  def displayObjectInGui(self, obj):
      self.currentObj = obj
  def rePopulateObjectFromGui(self):
      pass
  def getCurrentObjectBeingDisplayed(self):
      return self.currentObj

class Customer:
  def __init__(self, name=''): # Pass customer name to Constructor
      if name:
          self.Firstname, self.Surname = string.split(name)

from pawt import swing
import string, java

class CustomerAsSwingForm(GuiTransformationInterface): # Model-Gui Mediator
  def __init__(self, customer):
    self.frame = swing.JDialog ( swing.JFrame(), 1)
    self.frame.contentPane.layout = java.awt.FlowLayout()
    addwidget = lambda obj, topane : topane.add(obj)
    pane = self.frame.contentPane
    self.edFirstname = swing.JTextField("",10)
    self.edSurname = swing.JTextField("",10)
    buttonOk = swing.JButton('Ok', actionPerformed=self.ok)
    buttonCancel = swing.JButton('Cancel', actionPerformed=self.cancel)
    addwidget(swing.JLabel("First Name: "),pane)
    addwidget(self.edFirstname,pane)
    addwidget(swing.JLabel("Surname: "),pane)
    addwidget(self.edSurname,pane)
    addwidget(buttonOk,pane)
    addwidget(buttonCancel,pane)
    self.displayObjectInGui(customer)
    self.frame.setSize(200,120)

  def show(self): self.frame.show()

  def close(self):
    self.frame.setVisible(0)
    self.frame.dispose()

  def ok(self,e): self.rePopulateObjectFromGui() ; self.close()
  def cancel(self,e): self.close()


  def displayObjectInGui(self, obj):
    GuiTransformationInterface.displayObjectInGui(self, obj) # call inheritec
    self.edFirstname.text = obj.Firstname
    self.edSurname.text = obj.Surname

  def rePopulateObjectFromGui(self):
    self.getCurrentObjectBeingDisplayed().Firstname = self.edFirstname.text
    self.getCurrentObjectBeingDisplayed().Surname = self.edSurname.text
```

The following test code creates a customer and displays it in a form, allowing user to edit.

```
cust = Customer('Anastasia Bones')
CustomerAsSwingForm(cust).show()    # Display the modal form
```



The user edits the customer form then clicks Ok or Cancel.

The form represent the attributes of the Customer Object passed to it.

```
print 'Customer is now: ', cust.Firstname, cust.Surname
java.lang.System.exit(0)
```

Creating more forms

For every GUI form required in your application, simply create a new mediating adapter class which implements the transformational methods in the required way to move object attribute data to the widgets and out again. All the widget specific code required to achieve this mapping is nicely encapsulated in the code of the transformational methods of the adapter class.

The combined architecture

In this paper, we have mentioned how the Transformation Interface metaphor is applicable to many 'transformational situations'. Our specific examples show the implementation of both persistence and GUI displays - using the same pattern! (there may be an comprehensibility advantage in this for both designers and implementers). Also, because of this broad applicability, the Transformation Interface arguably has an architectural aspect about it. For example, the above working implementations of a persistence layer and GUI display mechanism can be 'plugged together' to form the hub of a complete application architecture.

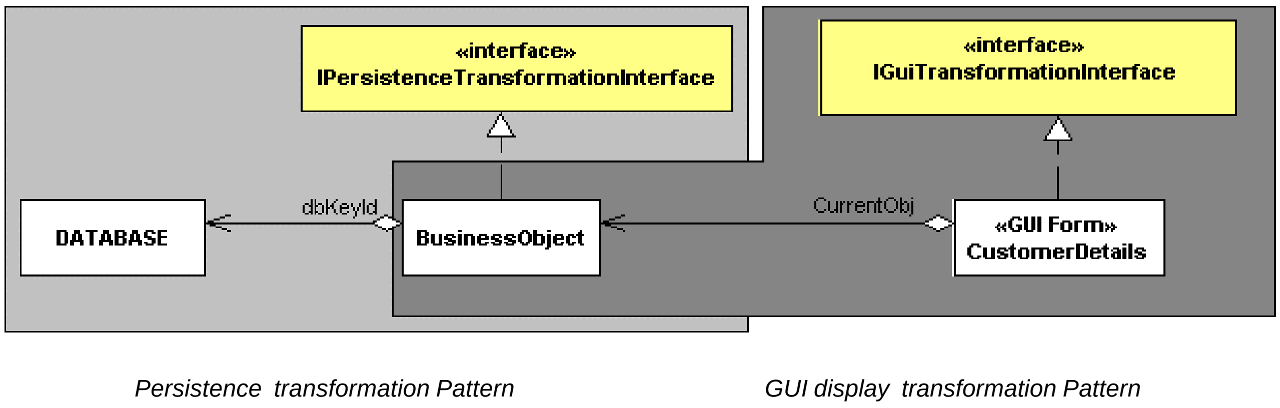

«interface»
IPersistenceTransformationInterface
«interface»
IGuiTransformationInterface
DATABASE
dbKeyId
BusinessObject
CurrentObj
«GUI Form»
CustomerDetails
Persistence transformation Pattern
GUI display transformation Pattern

Figure 7. Applying the transformation interface pattern twice, provides a solution to both business object persistence and GUI display functionality.

Related Patterns

Serializer, Riehle, Siberski, Bäumer, Megert and Züllighoven. PLoPD 3, 1997 page 293. Serializer deals with the context of converting objects and arbitrarily complex data structures into different data representations and back, with an emphasis on streamed and persistent data formats - making no reference to its applicability in a GUI presentation layer context.

The Serializer pattern is a much more complex solution than Transformation Interface, and resolves different forces. The Serializable interface consists of a transformational method pair `writeTo(Writer)/readFrom(Reader)` but these do not encapsulate any format specific mapping code, rather they are coded specifically to persist themselves, but are coded *generically* in terms of a reader or writer class which reads and writes object attributes via generic operations like

```
  writer->writeString( customername)
  writer->writeInteger( age )
```

Neither the readers, writers, nor the object being serialized know anything about any particular data format – specific concrete readers and writer classes override `writeString and writeInteger` etc. in order to implement these generic operations for a particular data format. Thus reader and writer objects know nothing about the actual application classes they are serializing. The benefit of this complex design pattern is that new concrete reader and writer objects for different representational formats can be seamlessly plugged in as needed.

The Transformation Interface is a simpler pattern, with less classes required to implement it – though the pattern does require that the transformation methods be re-implemented for each new format. The Transformation Interface pattern encourages the transformational method code to be customising to the transformational task at hand, which may involve things like taking object attributes and storing them in GUI widget attributes or even may involve special calculations and drawings onto a graphic canvas area. The serializer pattern's transformation are limited to reading and writing value types like string and integer etc. to an abstract stream.

Model Gui Mediator (MGM), Bulka, A. KoalaPlop2000, RMIT Australia departmental technical report, http://www.cs.rmit.edu.au/reports/2000/00-7.html. MGM implements a GUI display transformation interface but also implements Observer to gain the benefits of more granular control of screen updates.

Known Uses

Reason! Software for critical thinking (1998-2000), Melbourne University, Australia.

RIPS 2001 payroll system uses load/save method pairs for persistence and similar GUI methods pairs for all its modal forms. Used by retiree investment company.

The Delphi class library (VCL) offer `loadFromFile/SaveToFile` method pairs for many component classes e.g. TBitmap, TPicture, and TStrings classes. They are also available for some data-aware components (TBlobField, TMemoField,
and TGraphicField), for other graphic formats (TGraphic, TIcon, and TMetaFile), for OLE (Object Linking and Embedding) containers, and for the TreeView and other Windows common controls.

APPENDIX

Easy Tips on Reading Python Code

Python is a simple, straightforward and elegant language. It uses standard conventions of accessing methods and properties and is fully OO. Types are associated with objects not variables, so you don't need to declare variables. Functions are called like `afunction(param1, param2)` and objects are created from classes the same way e.g. `o =`

`MyClass()`. Python is case sensitive.

There are no `begin end` reserved words or `{ }` symbols in Python to indicate code blocks – this is done through indentation. The colon in `if lzt:` simply means ‘then’. The idiom of testing objects (rather than expressions) in an if statement makes sense as python treats empty lists, None and 0 as false.

Python understands named parameters e.g. In a method call, `afunction(From=None)` means you are passing None *(null / nil)* as the ‘From’ parameter, whilst in a method definition `From=None` means that if the caller does not supply this parameter, then it will default to None.

The first argument of all methods defined inside classes must be ‘self’. This argument fills the role of the reserved word *this* in C++ or Java or *self* in Delphi. Most languages supply this parameter (a reference to the current instance) implicitly whereas Python makes this concept explicit. At runtime this parameter is supplied by the system, not the caller of the method, thus the `def AddOrder(self, order)` method in reality takes *one* parameter when calling it: `AddOrder( order )`.

The statement `pass` means *do nothing*.

You can return multiple items at once *e.g.* `return (2, 50)` and also assign multiple items at once *e.g.* `x, y, z = 0` or even expressions like `result, status, errmsg = myfunction(1, 90)`.

Other class files/modules are imported using `import somefile`. `__init__` methods are simply constructors. Finally, a *lambda* is just a one line function that reads `functionname = lambda paramlist : returnedexpression`.

Both **Python** and JPython (Java Python, now called **Jython**) are open source, free and available from www.python.org

-Andy Bulka
abulka@netspace.net.au
6/32 Loller street,
Brighton VIC 3186, Australia
Ph: +613-9593-1389