Application Facades

Deep in the bones of Object-Oriented programming is notion of building a set of classes that mimics the objects in the “real world”. That is we try to analyze the way people think about the world and let the classes in our programs model the way an expert in a domain thinks. Many of the books on OO analysis and design talk about developing this domain model. To do anything with the domain model we need to put information into and out of it, typically through a Graphical User Interface (GUI). There is not so much written about that part of object-oriented design.

This article serves many purposes, but the first purpose is to address this issue of the relationship between a GUI and the underlying model. I hold to the principle that user interfaces should lie on the outside of the system and be invisible to the classes that model the problem. This keeps the often varying UI functionality away from the domain classes. The domain classes will model the domain, the UI classes handle the UI — simple and separate responsibilities.

I go further than this, and divide the UI classes into two: a presentation class and an application facade class. The presentation class is the class that handles all the UI work. The application facade class is responsible for talking to the domain model and getting the information to the presentation class in exactly the form that the presentation class requires. In this way the presentation class needs to know nothing about what is going on in the model, it only handles the UI work.

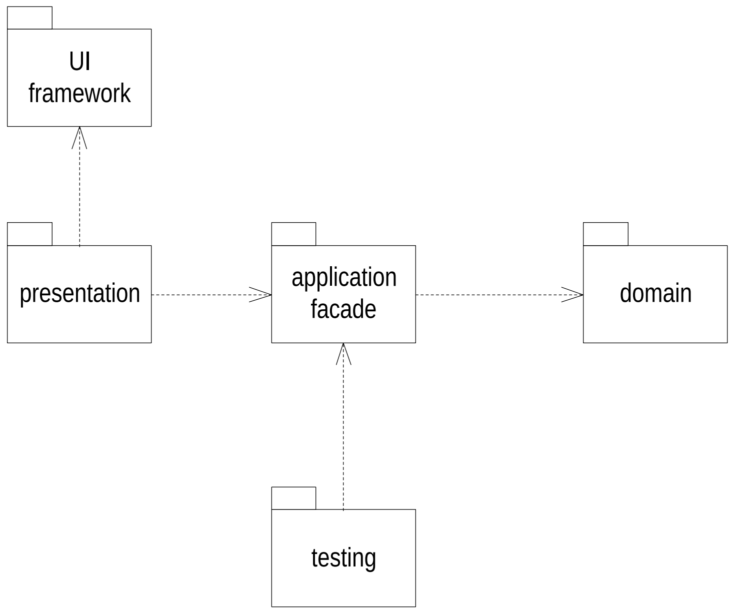


Figure 1 The general structure of packages and dependencies

Figure 1 shows a UML [UML] class diagram of the general structure of packages and dependencies I use. The key points are:

- The presentation package does not see the domain package
- The application facade package does not see the UI framework
- The testing package does not need to see the presentation package.

The benefits we get from this approach are:

- We have split the UI classes into two sections with clear responsibilities for each. This makes each class simpler and easier to understand and maintain.

- We can choose to separate the tasks of coding the presentation and application facade classes. Those who code the application facade need to understand the domain package but need know nothing about coding UI classes, the presentation programmers need to know about the UI but not about the details of the domain. If the domain classes and the UI framework are complex, as they often are, this makes it much easier for programmers to be found and trained.
- We can test most of the system without using the UI. Testing through the UI is generally awkward and it is difficult to set up and maintain the testing scripts. By testing through the application facade only we make it much easier to set up an automatic testing system which is essential to any well managed project. There is still some testing of the UI that is needed, but the task is greatly reduced as in that testing we are only concerned with the way the UI works, not how it interacts with the domain classes.

This article will explore how to do this in practice, with examples in Java. I discussed the principles of this in chapter 12 and 13 of [Fowler], but did not provide any code examples. This article will should help dispel that problem. For the domain model I chose to take some of the ideas of observation and measurement from chapters 3 and 4 of [Fowler]. So this article also illustrates some examples of implementing those patterns.

This article also uses much the same material as that in the Java example in UML Distilled.

An Example Problem

Consider how a hospital's computer systems might get at various observations they have made about a patient. You could have a patient class with attributes for all the different types of observations (height, blood type, heart rate, etc) but there would be thousands of such attributes: too many to have as attributes of a patient class. So we can get around this by using the Observation and Measurement patterns from [Fowler]. For the purposes of our discussion we want to be able record quantitative (height, 6 feet) and qualitative (blood group A) statements about the patient. We also want to be able to assign qualitative statements depending on a measurement. Thus is we record a person is breathing at a rate of 23 breaths a minute we should be able to automatically make the qualitative statement that that is a fast breathing rate.

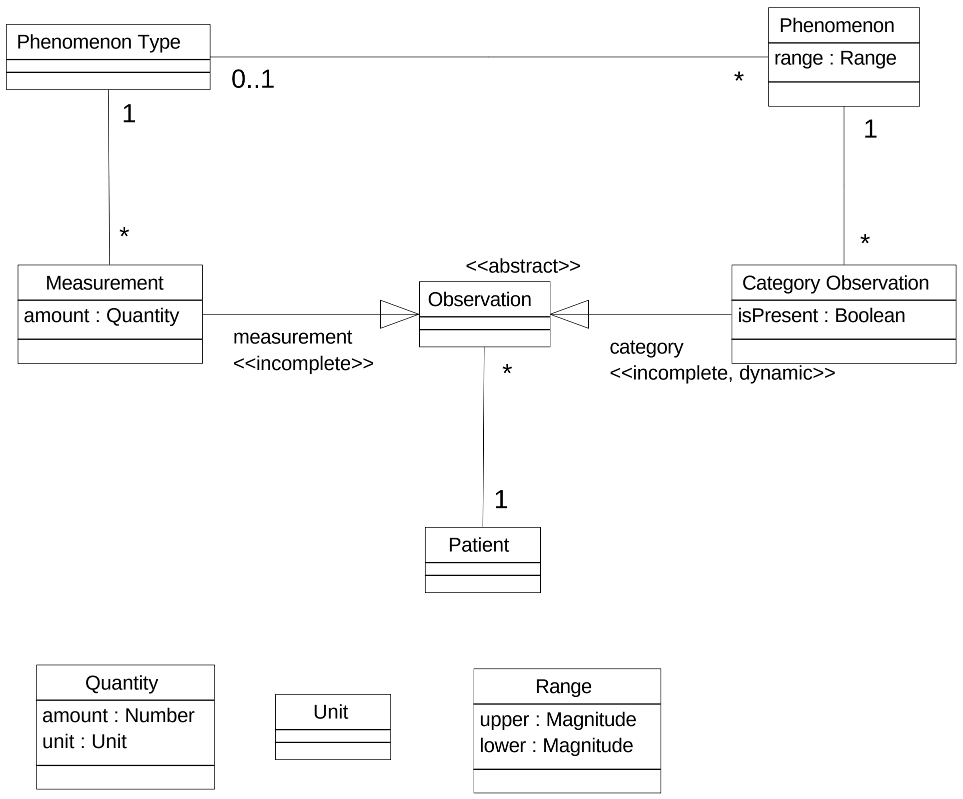


Figure 2 Conceptual UML diagram for the domain of this example

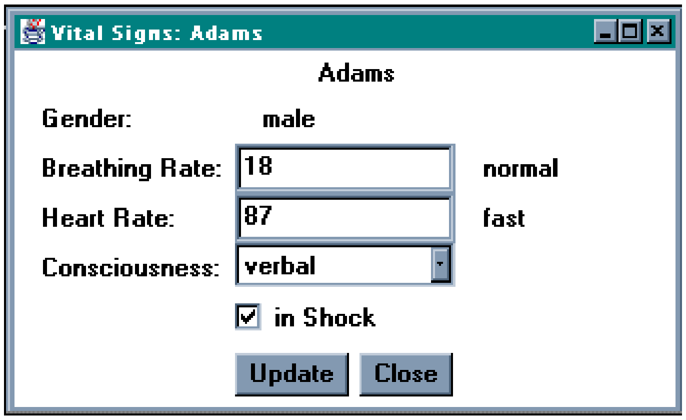
Figure 2 shows a conceptual model to support this kind of behavior. Before we dive into it I need to stress that word *conceptual*. This model is not what the classes look like, rather it is an attempt to model the concepts inside a doctor's head. It is similar to the classes, but as we shall see we have to change them a bit in the implementation. I have used several patterns here from [Fowler], specifically *Quantity*, *Measurement*, *Observation*, *Range*, and *Phenomenon with Range*. I'll discuss how the model works here, but I won't discuss the justification for why I'm doing that way, that I will leave to the book.

Say we want to record that Martin is breathing at 23 breaths per minute. We would do this by creating a measurement object linked to the patient object that represents Martin. The phenomenon type of this measurement object would be called "breathing rate". The amount in a measurement would be handled by a quantity object with amount of 23 and unit of "breaths per minute".

To say that Martin's breathing is fast we would create a category observation, again with Martin as the patient. The category observation would be linked to a phenomenon of "fast breathing rate" which in turn would be linked to the phenomenon type of "breathing rate". If the "fast breathing rate" phenomenon has a range, we should be able to automatically tell if it applies to a breathing rate of 23.

The way the Figure 2 works a single observation object can be both a measurement and a category observation at the same time (since the generalization arrows carry different labels). Also a measurement can begin life as a plain measurement and become a category observation as well later (indicated by the {dynamic} constraint). The combination of the {abstract} constraint on observation and the {incomplete} constraints on its subtypes implies that an observation can be either a measurement, or a category observation, or both; but it may not be neither. This is a conceptual picture that we will not be able to directly implement as Java does not allow us quite this flexibility in typing.

A model along the lines of this is very suitable for a hospital example because it will scale to the thousands of phenomena that are observed in a hospital setting. For an individual use, however, it is not so suitable. An individual use may want a simpler screen entirely, along the lines of that in Figure 3. Here the user does not want to bother with knowing about observation objects, they just want to assign a value to some patient attribute.



Gender: male
Breathing Rate: 18 normal
Heart Rate: 87 fast
Consciousness: verbal
[✓] in Shock

Figure 3 A sample screen showing a simpler view of patient information

Our task is to implement the model in Figure 2 yet provide a UI of the form of Figure 3. We will do this by creating an application facade that converts from Figure 2 to a form ready for Figure 3 and a presentation object that gives the display in Figure 3. I'm not making any claims about the practical usefulness of a screen like Figure 3, the screen is purely a sample to discuss the software principles.

Implementing Quantity

Faced with this kind of situation many people would represent a heart rate with a number. I prefer to always include units with this kind of dimensioned value, hence my use of the *Quantity* pattern. Implementing the Quantity pattern is fairly straightforward in any object-oriented language.

```
public class Quantity {
  private double _amount;
  private Unit _unit;
```

Although we use a double for the internal amount we can provide constructors for different initialization options. (Note that by convention I use a leading underscore on all fields.)

```
  public Quantity (double amount, Unit unit) {
     requireNonNull(unit);
     _amount = amount;
     _unit = unit;
  };

  public Quantity (String amountString, Unit unit)  {
     this (new Double(amountString).doubleValue(), unit);
  };

  public Quantity (int amount, Unit unit) {
     this (new Double(amount).doubleValue(), unit);
  };

  protected void requireNonNull(Object arg) {
     if (arg == null) throw new NullPointerException();
  };
```

The quantity class needs a unit class, which for the purposes of this example need only know its name. A class that has a name is a common need in these circumstances, so I have an abstract class, DomainObject, for it.

```
public DomainObject (String name)   {
     _name = name;
  };

  public String name ()      {
     return _name;
  };
  protected String _name = "no name";
};

public class Unit extends DomainObject
```

Registrar

Another core behavior we will need is to get hold of specific objects without using global variables for example the unit “breaths per minute”. I need unit to be an *Entry Point* [Fowler] for my objects so I can just refer to the “breaths per minute” unit by going something like `Unit.get(“breaths per minute”)`. I can implement this in two ways: either by having a static variable in the unit class, or by having a Registrar object that manages these entry points. I prefer the Registrar as it is easier to manage. The Registrar is a Singleton [Gang of Four] which manages several entry points.

```
public class Registrar          {

  private static Registrar _soleInstance = new Registrar();
```

Each entry point is a Hashtable. Since the Registrar manages several entry points it keeps each entry point as the value in a Hashtable indexed by some useful name, usually the name of the class that is acting as the entry point.

```
public class Registrar          {

  private static Registrar _soleInstance = new Registrar();
  private Dictionary _entryPoints = new Hashtable();

  private void addObj (String entryPointName, DomainObject newObject)     {
     Dictionary theEntryPoint = (Dictionary) _entryPoints.get(entryPointName);
     if (theEntryPoint == null)      {
        theEntryPoint = new Hashtable();
         _entryPoints.put(entryPointName, theEntryPoint);
     };
     theEntryPoint.put(newObject.name(), newObject);
  };

  private DomainObject getObj (String entryPointName, String objectName)  {
     Dictionary theEntryPoint = (Dictionary) _entryPoints.get(entryPointName);
     assertNonNull (theEntryPoint, "No entry point present for " + entryPointName);
     DomainObject answer = (DomainObject) theEntryPoint.get(objectName);
     assertNonNull (answer, "There is no " + entryPointName + " called " + objectName);
     return answer;
  };

  private void assertNonNull(Object arg, String message) {
     if (arg == null) throw new NullPointerException(message);
  };
```

I use *Lazy Initialization* [Beck] if a client wants to store a value into an entry point collection that I have not used yet. To make it a little easier to use the registrar I put some static methods on the class.

```
  public static void add (String entryPoint, DomainObject newObject)      {
     _soleInstance.addObj(entryPoint, newObject);
  };

  public static DomainObject get (String entryPointName, String objectName)       {
     return _soleInstance.getObj(entryPointName, objectName);
  };
```

However to make it easier for programmers I use methods on the appropriate classes to get values in and out of the registrar:

```
public class Unit extends DomainObject {

  public static Unit get (String name)      {
     return (Unit) Registrar.get("Unit", name);
  };

  public Unit persist()      {
        Registrar.add("Unit", this);
        return this;
    };
```

You will see that I do the same thing for other entry point classes in the system. I would be inclined to make `persist` and `get` part of an interface. Unfortunately if I did that then the return type of `get` would have to be defined to be something like `Object` and I could not override it to something more specific within a class. This would result in a lot of casting, so I just duplicate those two methods on any entry point class. Of course a programmer could use the `Registrar` directly, but then I have to remember the collection name. I try not to clutter up my memory with stuff like that.

Phenomenon and Phenomenon Type

Now we will turn to phenomenon and phenomenon type. An example of these classes might be blood group, which we would describe by a single phenomenon type of ‘blood group’ with phenomena of ‘blood group A’, ‘blood group B’, ‘blood group O’, and ‘blood group A/B’. A phenomenon type need not have phenomena. We might not choose to put ranges on people’s height, in that case the phenomenon type of ‘height’ would have no phenomena. So it makes sense to have a simple construction for phenomenon type.

```
  public class PhenomenonType extends DomainObject    {

    public PhenomenonType (String name)        {
        super (name);
    };
```

A phenomenon may also exist alone, such as the phenomenon ‘shock’.

```
  public class Phenomenon extends DomainObject        {

    public Phenomenon (String name)    {
        super (name);
    };
```

Of course the interesting case is when we have to link them together. Typically we may want to do this by giving an array of names to a phenomenon type:

```
        PhenomenonType sex = new PhenomenonType("gender").persist();
        String[] sexes = {"male", "female"};
        sex.setPhenomena (sexes);
```

We now need to think about how the conceptual association between phenomenon type and phenomenon should be implemented. For a full discussion see the *Implementing Associations* pattern [Fowler]. In this case I am going to implement it by *Pointers in Both Directions*. Many people shy away from these ‘back pointers’, but I find that if they are used appropriately they do not cause trouble. One source of trouble lies in keeping them up to date and in sync. To deal with this I always ensure that the updating is controlled by one side of the association — in this case the phenomenon. Since we only do this at creation time, the behavior is in an alternative constructor for phenomenon.

```
  public class Phenomenon extends DomainObject        {
    private PhenomenonType _type;
    public Phenomenon (String name, PhenomenonType type)      {
        super (name);
        _type = type;
        _type.friendPhenomenonAdd(this);
    };

  public class PhenomenonType extends DomainObject    {
    private Vector _phenomena = new Vector();
    void friendPhenomenonAdd (Phenomenon newPhenonenon)       {
        // RESTRICTED: only used by Phenomenon
        _phenomena.addElement(newPhenonenon);
    };
```

The phenomenon constructor really needs privileged access to phenomenon type here, a good use of C++’s friend construct. We don’t have friends in Java. I could make phenomenon type’s field be of package visibility, but I prefer to keep my data private. So I create a special method using the word “friend” to communicate its special purpose.

With this behavior in place I can now implement `setPhenomena()`:

```
  public void setPhenomena (String[] names) {
     for (int i = 0; i < names.length; i++)
        new Phenomenon (names[i], this);
     };
```

I don't use arrays that much in C^{++} or Java, so it gave me a nostalgic thrill to write a classic C for loop.

Creating Observations

Now its time to think how we going to create an observation, and in particular how we are going to deal with that awkward to implement classification of observation. Well I'm going to duck it. I'm going to have an observation class and a measurement class. The observation class will have the link category observation behavior folded into it. Simple and it will work for this situation. And by ducking a more complicated implementation I am actually passing on an important lesson. Don't try to come up with a clever way to do something if a simple way works fine. So this approach has limitations. Always ask yourself if you can live with the limitations — if you can you should. You can always make it more complicated later.

```
public class Observation extends DomainObject      {

  protected Phenomenon _phenomenon;
  private boolean _isPresent;
  private Date _whenObserved;

  public Observation (Phenomenon relevantPhenomenon, boolean isPresent, Patient patient,
                      Date whenObserved)     {
     _phenomenon = relevantPhenomenon;
     _isPresent = isPresent;
     patient.observationsAdd(this);
     _whenObserved = whenObserved;
  };
```

In this case I'm not doing any two-way pointer stuff. Observation keeps the pointer to phenomenon, but patient keeps the pointers to observation. I use a vector for the set of observations that a patient has, as I don't know the size in advance. I find I rarely know the size of things in advance, so I use arrays rarely. Vectors are on the whole easier to use, although the downcasting gets up my nose after a while.

```
public class Patient extends DomainObject   {

  private Vector _observations = new Vector();
  public void observationsAdd (Observation newObs) {
     _observations.addElement(newObs);
  };
```

First Steps in the Facade

Now we have enough to begin to consider setting up a facade. The sample window in Figure 3 is based on your vital signs, so I will call it a VitalsFacade. When I construct a facade I give it a *subject*: a reference into the domain model. In this case the subject is the patient.

```
package vitalsFacade;
import observations.*;

public class VitalsFacade {
  private Patient _subject;
  public VitalsFacade (Patient subject)     {
     _subject = subject;
  };
```

All the other classes were in the observations package, so it needs to import them.

We can begin the facade by providing some simple queries for the patient's name and gender. The name is very easy.

```
  public String name() {
     return _subject.name();
  };
```

The gender is a little bit more complicated. We may have more than one observation of gender. We will assume that we are not going to have to deal with contradictory observations (in reality we do have to deal with them, but I can cut it out of the scope of this exercise). So we need to find the latest observation of a phenomenon whose phenomenon type is “Gender”, and then return the string that describes the phenomenon. We need to return a string for that is all the presentation class can understand. As well as navigating through the domain model, the facade is responsible for converting the types to the simple types that the UI classes understand.

We could put all of this behavior on the facade, but it is better to delegate much of it to the patient.

```
Class VitalsFacade {
  public String gender() {
     return _subject.phenomenonOf("gender").name();
  };
```

The patient will take this and find the latest observation of the requested phenomenon type. If there is none it will just hand back null.

```
class patient {
  public Phenomenon phenomenonOf(String name)      {
     return phenomenonOf(PhenomenonType.get(name));
  };

  public Phenomenon phenomenonOf(PhenomenonType phenomenonType)    {
     return (latestObservation(phenomenonType) == null ?
        null :
        latestObservation(phenomenonType).phenomenon());
  };
```

To find the latest observation the patient first finds all the observations that exist for a given phenomenon type, which it supplies as an enumeration.

```
  public Enumeration observationsOf(PhenomenonType value)  {
     Vector result = new Vector();
     Enumeration e = observations();
     while (e.hasMoreElements())    {
        Observation each = (Observation) e.nextElement();
        if (each.phenomenonType() == value) result.addElement(each);
     };
     return result.elements();
  };

  public PhenomenonType phenomenonType()    {
     return _phenomenon.phenomenonType();
  };

  public Enumeration observations() {
     return _observations.elements();
  };
```

We could speed this up by keeping our observations in a hashtable indexed by phenomenon type. If I ever have to performance tune this I will profile it to see, but for this exercise I’m more interested in the slow and simple approach. Once we have the enumeration of the appropriate observations we can find the latest.

```
  public Observation latestObservation(PhenomenonType value)      {
     return latestObservationIn (observationsOf(value));
  };

  private Observation latestObservationIn(Enumeration observationEnum)    {
     if (!observationEnum.hasMoreElements()) return null;
     Observation result = (Observation) observationEnum.nextElement();
     if (!observationEnum.hasMoreElements()) return result;
     do {
        Observation each = (Observation) observationEnum.nextElement();
        if  (each.whenObserved().after(result.whenObserved())) result = each;
     }
     while (observationEnum.hasMoreElements());
     return result;
  };
```

Handing back null is actually rather awkward. It means the method `gender` will break if it gets a null sent back to it. To fix this I would need to put a conditional in place

```
  public String gender() {
     return (_subject.phenomenonOf("gender") == null) ?
        null :
        _subject.phenomenonOf("gender").name();
  };
```

And I have to do this everywhere I use `Patient.phenomenonOf()`. That is downright boring, and I don't like boring programming, not just because its boring, but also because it tends to be very error prone. Fortunately there is a savior — the *Null Object* pattern (which will be published in the PloPD 3 book). To use a null object all I need to do is create a subclass of phenomenon called null phenomenon, and override the behavior of phenomenon to do suitable default things.

```
class NullPhenomenon extends Phenomenon {

  public String name () {
    return "";
  };
}
```

I can now return a null phenomenon instead.

```
  public Phenomenon phenomenonOf(PhenomenonType phenomenonType) {
    return (latestObservation(phenomenonType) == null ?
      new NullPhenomenon() :
      latestObservation(phenomenonType).phenomenon());
  };
```

The nice thing about this is that no client of `Patient.phenomenonOf()` is even aware that a null phenomenon class exists. The class is not declared public and is invisible to the client programs. There are cases when we might need to see if we have a null phenomenon, we can do this by adding a `isNull()` method to phenomenon. Using null objects can do a lot to simplify code, use it whenever you are standing on your head with your arms crossed trying to deal with null responses to questions.

Testing the Facade

Now we actually have enough to test the facade. My approach to testing is to test frequently after each step, and to build up my test code so that I have a solid set of regression tests to run at any time. I've actually run a bit long here without testing, so its time to catch up.

The basis of testing is a tester class what will get run when we test, so it needs a main method. The main looks rather complicated, this is because of Cafe's habit of printing to the standard output, and then closing the standard output screen before you can see what was printed. With this code I get to read it and dismiss it with a return, even if I get a runtime exception.

```
public class Tester {

  public static void main (String[] argv) {
     try {
     System.out.println (new Tester().test());
     } catch (Exception e) {
        System.out.println ("Exception while executing: " + e.toString());
     };
     try {
        System.in.read();
     } catch (java.io.IOException e) {};
  };
```

The main action lies in the test method.

```
  TestResult test() {
    setup();
    testPhenomena();
    return _result;
  };
```

OK, so I lied the test method is just three method calls, but to start delving into the depth let me explain what this test result class is. Essentially it is just a mechanism to storing up details of failed tests and printing them out at the end.

```
public class TestResult {
  private Vector _failures = new Vector();
```

We add details about failures with a descriptive string.

```
  public void addFailure (String message) {
    _failures.addElement (message);
```

```
   };
```

When we are done we print either the list of failures or the comforting “OK”.

```
   public String toString()   {
      if (isFailure())
         return "Failures\n" + failureString();
      else
         return "OK";
   };

   public boolean isFailure() {
      return ! _failures.isEmpty();
   };

   public String failureString() {
      String result = "";
      Enumeration e = _failures.elements();
      while (e.hasMoreElements()) {
         result += "\t" + e.nextElement() + "\n";
      };
      return result;
   };
```

To use all of this we begin by setting up our genders in the setup method.

```
   Public void setup() {
      PhenomenonType sex = new PhenomenonType("gender").persist();
      String[] sexes = {"male", "female"};
      sex.setPhenomena (sexes);
```

We then create a person and make him male

```
      new Patient("Adams").persist();
      new Observation
         (PhenomenonType.get("gender").phenomenonNamed("male"),
         true,
         Patient.get("Adams"),
         new Date (96, 3, 1));
```

Then we test to see if we are reading what we are supposed to be reading.

```
   Public void testPhenomena() {
   try {
         VitalsFacade facade = new VitalsFacade (Patient.get("Adams"));
         if (!facade.name().equals("Adams")) _result.addFailure("adams without name");
         if (!facade.gender().equals("male")) _result.addFailure("adams not male");
      } catch (Exception e) {
         String message = "Exception " + e.toString();
         if (e.getMessage() != null) message += " " + e.getMessage();
         _result.addFailure (message);
         e.printStackTrace();
      };
```

This way we capture the results of many tests and print the results of them all. This is a pretty simple framework, and on a real project I would expect to see something rather better. But this one is enough for my needs here.

So far its not much of test, there is only a minimal amount of behavior going on. But the point is to test like those people in Chicago allegedly used to vote — early and often. A simple testing framework makes life easier. As you go on you can add more tests, but you never throw tests away. Not unless you like driving at night with your lights off.

Updating through the facade

So far we are only looking at using the facade in a simple case, to get a value of a phenomenon where there is only one in our test case, and we are not trying to update. So for a more complicated example lets take the level of consciousness phenomenon type. This is a simple classification of how conscious someone is. There are four values: alert (talking normally), verbal (responds to verbal stimuli), pain (responds to painful stimuli - such as tweaking the ear) and unresponsive (even to ear tweaking). In the model this simply works as a phenomenon type with four phenomena.

With the facade the first route we can try is to work with methods on VitalsFacade. For level of consciousness we will want to get the current value, update the current value, and also find out what the legal values are so we can display them in a pop-up menu.

```
public String levelOfConsciousness ()
public void levelOfConsciousness (String newValue)
public Enumeration levelOfConsciousnessValues ()
```

With the case of gender we just asked the patient directly for its information. We can do the same here, but as things get a bit involved it can be useful to store a value in the facade. It generally makes it easier to see what is going on. So we store the current value in a string.

```
private String _levelOfConsciousness;
```

We put the information into the string when we load the facade.

```
public void load()        {
   _levelOfConsciousness = levelOfConsciousnessRetrieve();

private String levelOfConsciousnessRetrieve() {
   return _subject.phenomenonOf("level of consciousness").name();
};
```

So far this very similar to the gender case we looked at before. To carry out an update we begin updating the value in the facade. When we do this we want to check we are getting a legal value for the level of consciousness. We need a way to find out what the legal values are. We could do this by just asking phenomenon type for its phenomena, but we also need a way to get from the strings of these values to the phenomena. To serve these two needs we would really like a hashtable, keyed by the strings displayed in the pop-menu with values as the phenomenon types. Phenomenon type can easily give us such a table.

```
class VitalsFacade {
  private Hashtable levelOfConsciousnessTable()    {
     return PhenomenonType.get("level of consciousness").mappingTable();
  };

Class PhenomenonType {
  public Hashtable mappingTable() {
     Hashtable result = new Hashtable();
     Enumeration e = phenomena();
     while (e.hasMoreElements())    {
        Phenomenon each = (Phenomenon) e.nextElement();
        result.put (each.name(), each);
     };
     return result;
  };
```

To check we are getting a legal value, we just ask the table for its keys

```
public void levelOfConsciousness (String newValue) {
   if (! levelOfConsciousnessTable().containsKey(newValue))
      throw new IllegalArgumentException("Illegal level of consciousness");
   _levelOfConsciousness = newValue;
};
```

We throw a runtime exception here, rather than a matched exception, because we don't really expect an illegal update value. Any window using this interface should really restrict the choices to the four legal values we give. In Design By Contract [Meyer] terms the fact that `newValue` is one of the four is really part of the pre-condition of `levelOfConsciousness(String)`, and thus it is not the responsibility of levelOfConsciousness to do any checking. Thus callers of levelOfConsciousness should not expect any checking. Its difficult to show that explicitly in Java, as we can't use compiler directives (as in C^{++}), let alone proper operation pre-conditions as they have in Eiffel. I suppose I could something of the kind

```
if (PRE_CONDITION_CHECKS_ON) {
```

and rely on the compiler to optimize away any problems. The reality is that I'm still undecided about how to handle pre-conditions in Java.

A useful thing to consider here is what we might do if we wanted to show something other than the standard names for the four phenomena. If we wanted to show something else the facade can build the hashtable itself with the alternative display strings. Each facade could potentially have its own display strings, and use the hashtables to map back to the internal phenomena.

Anyway back to the main plot. Once we have the new value in the facade's field we then need to save it into the domain model.

```
public void save() {
   if (_date == null) _date = new Date();
   levelOfConsciousnessUpdate();

private void levelOfConsciousnessUpdate() {
   if (_levelOfConsciousness == null) return;
   new Observation ((Phenomenon)
      levelOfConsciousnessTable().get(_levelOfConsciousness),
      true, _subject, _date);
};
```

We use the table again to get the actual phenomenon we want. Each time we create a observation we need a date. We can explicitly set a date into the facade (we will do this for testing) or the computer assigns the current date.

So again we need to test. The test routine now has four distinct stages

```
TestResult test() {
   loadPhenomena();
   setupAdams1();
   setupAdams2();
   testPhenomena();
   return _result;
};
```

`LoadPhenomena` sets up the various phenomenon types and other knowledge objects that are loaded permanently into the registrar. `SetupAdams1` and `setupAdams2` do two stages of changes to our sample patient.

```
private void setupAdams1() {
   new Patient("Adams").persist();
   new Observation
      (PhenomenonType.get("gender").phenomenonNamed("male"),
      true,
      Patient.get("Adams"),
      new Date (96, 3, 1));
   VitalsFacade facade = new VitalsFacade (Patient.get("Adams"));
   facade.date(new Date (96,3,1,4,0));
   facade.levelOfConsciousness("pain");
   facade.save();
};

private void setupAdams2() {
   VitalsFacade facade = new VitalsFacade (Patient.get("Adams"));
   facade.date(new Date (96,3,1,6,0));
   facade.levelOfConsciousness("verbal");
   facade.save();
};
```

This gives us two observations to choose the latest from.

```
private void testPhenomena () {
   try {
      VitalsFacade facade = new VitalsFacade (Patient.get("Adams"));
      facade.load();
      if (!facade.name().equals("Adams")) _result.addFailure("adams without name");
      if (!facade.gender().equals("male")) _result.addFailure("adams not male");
      if (!facade.levelOfConsciousness().equals("verbal"))
                                 _result.addFailure("adams not verbal");
   } catch (Exception e) {
      String message = "Exception " + e.toString();
      if (e.getMessage() != null) message += " " + e.getMessage();
      _result.addFailure (message);
      e.printStackTrace();
   };
};
```

As we continue this example we will see these set up and test routines steadily grow.

Adding a Measurement

So far we have looked at qualitative observations. Now it is time to look at the quantitative observations. Although the conceptual model gives me the outline of how I expect to do them, I haven't yet coded up the Measurement class.

```
public class Measurement extends Observation{
  private Quantity _amount;
  private PhenomenonType _phenomenonType;
```

Measurement will need a different constructor, one that provides a quantity and a phenomenon type rather than a phenomenon. In this case I am setting some of the fields of the superclass. I could use a superclass constructor for this, but its not really a constructor, and using a constructor would communicate the wrong thing to the reader of this code. So I've added an initialize method to observation for those fields that are really observation's responsibility to deal with. The initialize method is marked protected to communicate the fact that it is intended for subclasses only.

```
Class Measurement
  public Measurement (Quantity amount, PhenomenonType phenomenonType, Patient patient,
                              Date whenObserved)
      initialize (patient, whenObserved);
      _amount = amount;
      _phenomenonType = phenomenonType;
      checkInvariant();
      _phenomenon = calculatePhenomenonFor(_amount);
  };

  private void checkInvariant () {
      assertNonNull (_phenomenonType, "Null phenomenon type");
      assertNonNull (_amount, "Null amount");
  };

Class Observation
  protected void initialize (Patient patient, Date whenObserved)   {
      patient.observationsAdd(this);
      _whenObserved = whenObserved;
  };
```

Similarly to working with observations, we need to get the latest measurement for a phenomenon type for a particular patient. Again I've used the *Null Object* pattern to give me a null quantity if I don't have a measurement.

```
  public Quantity latestAmountOf(PhenomenonType value)     {
      Measurement latestMeasurement = (Measurement) latestObservation(value);
      return (latestMeasurement == null) ?
         new NullQuantity() :
         latestMeasurement.amount();
  };
```

In the facade, I will again store the display value in a variable and pull it in when I load the facade. I will start work with heart rate.

```
Class VitalsFacade {
  private String _heartRate;

  public String heartRate() {
      return _heartRate;
  };

  private String heartRateRetrieve() {
       _value = _facade.subject().latestAmountOf
                              (PhenomenonType.get("Heart Rate")).amountString();
  };

class Quantity {
  public String amountString() {
       return String.valueOf(_amount);
  };

Class NullQuantity {
  public String amountString() {
       return null;
  };
```

When updating the GUI will allow users to type in whatever numbers they like. Of course they might not just type in numbers, they might add letters too. So the facade should check that its update string is properly numeric. If there is a problem it will just set the value to null.

```
   public void heartRate(String newRate)     {
      _heartRate = (isNumeric(newRate) ? newRate : null);
   };

   private boolean isNumeric(String value)   {
       for (int i = 0; i < value.length(); i++) {
           char eachChar = value.charAt(i);
           if (Character.isDigit(eachChar)) continue;
           if ((i == 0) & (eachChar == '-')) continue;
           if (eachChar == '.') continue;
           return false;
       };
       return true;
  };
```

There is room for improvement in the numeric checking, but I’ll leave that as an exercise for the reader. What I would have preferred to do was to try creating a number and handle any resulting exception.

```
      try {
         Float.valueOf(value);
      } catch (NumberFormatException e)      {
         return false;
      }
      return true;
```

Sadly, however, if you go `Float.valueOf(“8m1”)` it does not throw an exception, it merely creates the number 8. It makes a certain sense, but was not what I wanted.

Now I have a numeric string in `_value`, so I am ready to create the measurement.

```
   private void heartRateUpdate () {
      if (_heartRate == null) return;
     new Measurement (new Quantity (_heartRate, Unit.get("beats/min")),
                      PhenomenonType.get ("Heart Rate"),
                      _subject,
                      _date);
   };
```

In all of this I have to have some policy about when and what should be updated. Should I only update observations that have changed, or should I create observations for all the non-null values? For this example I am creating an observation for each non-null item on the window, whether or not it has changed since the last time. This is complicated by the fact that the changes to the domain model may not be visible from this facade. If I have a measurement of 82 beats per minute now, and another measurement of 82 beats per minute in ten minutes; I cannot tell if there is two measurements or one measurement from this UI. If I was looking through another UI at a plot of heart rate over time I would see a difference. Different facades would use different policies. You just need to make sure the effects on the domain model are what a user who is aware of the domain model would expect.

I can now add tests for this new behavior.

```
   private void setupAdams1() {
      new Patient("Adams").persist();
      new Observation
         (PhenomenonType.get("gender").phenomenonNamed("male"),
         true,
         Patient.get("Adams"),
         new Date (96, 3, 1));
      VitalsFacade facade = new VitalsFacade (Patient.get("Adams"));
      facade.date(new Date (96,3,1,4,0));
      facade.levelOfConsciousness("pain");
      facade.heartRate("92");
      facade.save();
   };

   private void setupAdams2() {
      VitalsFacade facade = new VitalsFacade (Patient.get("Adams"));
      facade.date(new Date (96,3,1,6,0));
      facade.levelOfConsciousness("verbal");
      facade.heartRate("87");
```

```
        facade.save();
    };

    private void testPhenomena () {
        try {
           VitalsFacade facade = new VitalsFacade (Patient.get("Adams"));
           facade.load();
           if (!facade.name().equals("Adams")) _result.addFailure("adams without name");
           if (!facade.gender().equals("male")) _result.addFailure("adams not male");
           if (!facade.levelOfConsciousness().equals("verbal")) _result.addFailure("adams
not verbal");
           if (facade.heartRate() == null) _result.addFailure ("Heart Rate is null");
           if (!facade.heartRate().equals("87")) _result.addFailure("incorrect heart rate");
```

Creating Ranges

The next step is to infer a phenomenon from a measurement. I might want to know if the heart rate value I have put in is fast, average, or slow. A professional doctor or nurse would not need a computer to tell them that, of course. Indeed they would probably be irritated by the computer saying that. But after all you are probably not a doctor or nurse and there are phenomena where they would find such an indication useful.

We know from the conceptual model that we can do much of what we want to do by giving a phenomenon a range. We could do this by giving a phenomenon an upper and lower value, but I prefer to use the *Range* pattern [Fowler]. This suggests making a class that handles all of the range like behavior. In some languages I can make a single range class, but since Java is both strongly typed and does not have paramaterized types it's easier to make a specific quantity range class.

```
  public class QuantityRange {
    private Quantity _start;
    private Quantity _end;
    private boolean _isStartInclusive;
    private boolean _isEndInclusive;
```

When you do a range with continuous variables, like real numbers, then Booleans that allow expressions such as $12 < x \leq 15$ are very useful. If you are dealing with discreet things, such as integers, then you don't need them.

```
    public QuantityRange (Quantity start, boolean isStartInclusive, Quantity end,
                                                   boolean isEndInclusive) {
        _start = start;  // null implies no lower bound
        _end = end;             //null implies no upper bound
        _isStartInclusive = isStartInclusive;
        _isEndInclusive = isEndInclusive;
    };

    public QuantityRange (Quantity start, Quantity end) {
        this (start, true, end, true);
    };

    public static QuantityRange unlimited() {
        return new QuantityRange (null, null);
    };
```

Once you have a range, you can use it to figure out whether a quantity lies within its bounds.

```
    public boolean includes (Quantity value) {
        if (_end != null) {
           if (value.isGreaterThan(_end)) return false;
           if (!_isEndInclusive && value.equals(_end)) return false;
        };
        if (_start != null) {
           if (value.isLessThan(_start)) return false;
           if (!_isStartInclusive && value.equals(_start)) return false;
        };
        return true;
    };
```

Of course to do this you need to add the appropriate operations to quantity.

```
    public boolean equals(Quantity arg) {
        return _unit.equals(arg.unit()) && (_amount == arg.amount());
    };

    public boolean isLessThan(Quantity arg) {
        requireSameUnitsAs(arg);
```

```
      return _amount < arg.amount();
   };

   public boolean isLessThanOrEqualTo(Quantity arg) {
      requireSameUnitsAs(arg);
      return isLessThan(arg) || equals(arg);
   };

   public boolean isGreaterThan(Quantity arg) {
      requireSameUnitsAs(arg);
      return !isLessThanOrEqualTo(arg);
   };

   public boolean isGreaterThanOrEqualTo(Quantity arg) {
      requireSameUnitsAs(arg);
      return !isLessThan(arg);
   };

   private void requireSameUnitsAs(Quantity arg) {
      if (!_unit.equals(arg.unit())) throw new IllegalArgumentException();
   };
```

You could argue that these should have been developed earlier, since we knew we would probably need them. But I prefer not to code unless I need to, because sometimes you code things you think you will need that you end up not needing. An exception to this is when you have long build times, or complicated inter-team communications. Then it is worth getting the interfaces sorted out earlier.

I put the tests for quantity range in the class itself, since its pretty well self contained.

```
   public static void main () {
      String result = "";
      Unit cm = new Unit ("cm");
      QuantityRange range1 = new QuantityRange (new Quantity (100,cm), new Quantity
 (200,cm));
      if (range1.includes(new Quantity(201, cm)))  result += "201 within range\n";
      if (range1.includes(new Quantity(200.0001, cm)))  result += "200.0001 within
 range\n";
      if (!range1.includes(new Quantity(200, cm)))  result += "200 not within range\n";
      if (!range1.includes(new Quantity(199, cm)))  result += "199 not within range\n";
      if (!range1.includes(new Quantity(101, cm)))  result += "101 not within range\n";
      if (!range1.includes(new Quantity(100, cm)))  result += "100 not within range\n";
      if (range1.includes(new Quantity(99.999, cm)))  result += "99.999 within range\n";
      if (range1.includes(new Quantity(99, cm)))  result += "99 within range\n";

      QuantityRange range2 = new QuantityRange (new Quantity (250,cm), false, new Quantity
 (400,cm), false);
      if (range2.includes(new Quantity(249.99, cm)))  result += "249.99 within range\n";
      if (range2.includes(new Quantity(250, cm)))  result += "250 within range\n";
      if (!range2.includes(new Quantity(250.0001, cm)))  result += "250.0001 not within
 range\n";
      if (!range2.includes(new Quantity(399.9999, cm)))  result += "399.9999 not within
 range\n";
      if (range2.includes(new Quantity(400, cm)))  result += "400 within range\n";
      if (range2.includes(new Quantity(400.0001, cm)))  result += "400.0001 within
 range\n";
      if (range2.includes(new Quantity(401, cm)))  result += "401 within range\n";

      if (result == "") result = "OK";
      System.out.println(result);
   };
```

This is an even simpler kind of testing framework than the one I'm using in general. It is a useful approach if you don't want extra classes. Putting tests in a class like this also helps a reader understand what the class does.

Determining a Phenomenon for a Measurement

With ranges in place we can now set up some phenomena for heart rate. First we need to give phenomenon a range.

```
   private QuantityRange _range;
   public void range (QuantityRange arg) {
      _range = arg;
   };
```

Then set three ranges for heart rate when we load the phenomena in the tester.

```
  private void loadPhenomena()      {
     PhenomenonType sex = new PhenomenonType("gender").persist();
     String[] sexes = {"male", "female"};
     sex.setPhenomena (sexes);
     PhenomenonType loc = new PhenomenonType("level of consciousness").persist();
     new Phenomenon("alert", loc);
     new Phenomenon("verbal", loc);
     new Phenomenon("pain", loc);
     new Phenomenon("unresponsive", loc);
     setupHeartRates();
  };

  private void setupHeartRates() {
     Unit bpm = new Unit("beats/min").persist();
     PhenomenonType heartRate = new PhenomenonType("Heart Rate").persist();
     new Phenomenon ("slow", heartRate).range(new QuantityRange
                    (null, false, new Quantity (60, bpm), false));
     new Phenomenon ("normal", heartRate).range(new QuantityRange
                    (new Quantity (60, bpm), true, new Quantity (80, bpm), true));
     new Phenomenon ("fast", heartRate).range(new QuantityRange
                    (new Quantity (80, bpm), false, null, false));
  };
```

The system now knows what ranges exist, so the next step is to do the assignment. The obvious place to do this is when we create a measurement.

```
  public Measurement (Quantity amount, PhenomenonType phenomenonType, Patient patient,
 Date whenObserved)
     initialize (patient, whenObserved);
     _amount = amount;
     _phenomenonType = phenomenonType;
     checkInvariant();
     _phenomenon = calculatePhenomenonFor(_amount);
  };
```

The measurement does this by asking the phenomenon type which phenomenon includes the range.

```
  public Phenomenon calculatePhenomenonFor(Quantity arg) {
     return _phenomenonType.phenomenonIncluding(arg);
  };
```

The phenomenon type asks each phenomenon if it includes the range, returning the first one that does. If none do, it returns null, which is appropriate for the measurement anyway.

```
 Class PhenomenonType {
  public Phenomenon phenomenonIncluding (Quantity arg) {
     Enumeration e = phenomena();
     while (e.hasMoreElements()) {
        Phenomenon each = (Phenomenon) e.nextElement();
        if (each.includes(arg)) return each;
     };
     return null;
  };

 class Phenomenon {
  public boolean includes (Quantity arg) {
     return (_range == null ? false : _range.includes(arg));
  };
```

The approach works whether or not we have defined ranges for the phenomena. The interaction diagram below sums up the messages that are going on.

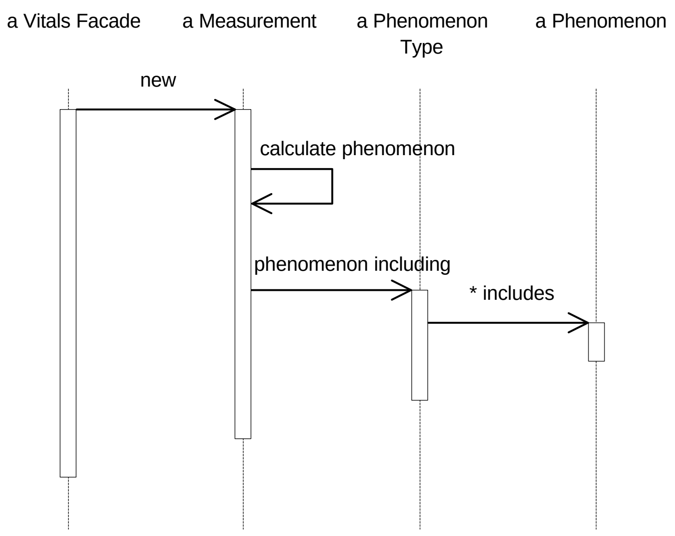


Figure 4 An interaction diagram for assigning a phenomenon to a measurement

As far as the UI is concerned the string that describes the phenomena is just another attribute of the facade, one that is not connected with the number. The facade provides a separate pair of operations for the UI to use to get at the information.

```
private String _heartRateName;

public String heartRateName() {
   return _heartRateName;
};
```

To load the stored attribute, the facade uses the same approach as for level of consciousness.

```
private String heartRateNameRetrieve() {
   return _subject.phenomenonOf("Heart Rate").name();
};
```

Even the facade is unaware of measurement's cleverness. I like it when bits of programs don't know what other bits are doing.

Refactoring the Level of Consciousness

Adding the breathing rate is pretty easy, just copy the code for heart rate and change a few names. Easy to say and easy to do, but it leaves a bad taste in my mouth and gives me an uneasy feeling. Also the vitals facade class is getting a little bit too complicated for my liking. Kent Beck says you should "listen to your code". Of course when he says things like that my first thought is that he's been living near San Francisco too long. The serious point is that when it starts getting too awkward to use the code because the code is getting in the way, then its time to take a scalpel to the code, in this case it's time for vitals facade to get on the operating table.

The underlying problem is that vitals facade is getting too complex. It's got these fields each of which has several methods working on it. The methods tend to work on only one field. That's a sign for breaking up the class by creating satellites. This refactoring step takes one or more fields and puts it in a separate class which is linked to the original. Those methods that touch the moved fields get moved too. We will begin by making a satellite of level of consciousness. First we define the new class and move over the field. I'm also going to give the new class a pointer back to its facade.

```
class LocAttribute {

  private String _value;
  private VitalsFacade _facade;

  LocAttribute(VitalsFacade facade) {
      _facade = facade;
```

```
   };
```

Next we take the methods over. I'll start with the accessors for the UI.

```
   void value (String newValue) {
      if (! tableOfValues().containsKey(newValue))
         throw new IllegalArgumentException("Illegal level of consciousness");
      _value = newValue;
   };

   String value() {
      return _value;
   };

   Enumeration legalValues () {
      return tableOfValues().keys();
   };

   private Hashtable tableOfValues()  {
      return PhenomenonType.get("level of consciousness").mappingTable();
   }
```

I've renamed them as I've gone. As far as the UI classes are concerned I don't want to change anything, so I want to still have the same accessors present in the facade, but these will now be simple delegates.

```
   public String levelOfConsciousness () {
      return levelOfConsciousnessAttribute().value();
   };

   public void levelOfConsciousness (String newValue) {
      levelOfConsciousnessAttribute().value(newValue);
   };

   public Enumeration levelOfConsciousnessValues () {
      return levelOfConsciousnessAttribute().legalValues();
   };
```

I need to decide how I will represent the link from vitals facade to its satellites. I shall use a hashtable.

```
  private Hashtable _attributes = new Hashtable();
```

The facade loads up the hashtable when it is created.

```
  public class VitalsFacade {
   public VitalsFacade (Patient subject)      {
      _subject = subject;
      setupAttributes();
   };

   void setupAttributes() {
       _attributes.put ("LevelOfConsciousness", new LocAttribute (this));
   };
```

The accessors then use a common method to get hold of the attribute.

```
   private LocAttribute levelOfConsciousnessAttribute() {
      return (LocAttribute) _attributes.get("LevelOfConsciousness");
   };
```

That handles the communication between the UI and facade with satellite. The UI does not know that the world has changed. Now to re-stitch the facade to the domain model. The facade will still do loads and saves, but in each case it will pass on the load messages to the satellites.

```
   public void load()         {
      date = null;
      loadAttributes();

   void loadAttributes () {
      Enumeration e = _attributes.elements();
      while (e.hasMoreElements()) {
         LocAttribute each = (LocAttribute) e.nextElement();
         each.load();
      };
   };
```

Exactly the same thing happens with a save. On a load the locAttribute pulls in values from the domain model.

```
    public void load() {
       _value = _facade.subject().phenomenonOf("level of consciousness").name();
    };
```

The save is quite similar.

```
  public void save() {
       if (_value == null) return;
       new Observation ((Phenomenon) tableOfValues().get(_value), true,
                                _facade.subject(), _facade.date());
    };
```

Spinning off a satellite like this is quite easy. There is an immutable single valued link from the satellite to its host. You don't always need the back pointer from the satellite to the host, but in this case it seems worthwhile, both for the subject and for the date. Having the tests is essential here because I can now run them to make sure I didn't break anything.

Refactoring the Heart Rate

The level of consciousness was easy, now its time for the heart rate. In this case we have both the heart rate and the breathing rate to consider, and they do look very similar. I feel a superclass coming on. I'll shout "down boy" to that feeling for the moment. First spin off the satellite, then do the generalization. The steps are familiar, first create the satellite.

```
  class HeartRateAttribute {
    private String _value;
    private VitalsFacade _facade;

    HeartRateAttribute(VitalsFacade facade) {
        _facade = facade;
    };
```

Move the accessing methods for the UI. `IsNumeric` gets moved too, but it looks just the same.

```
    String value() {
       return _value;
    };

    void value (String newValue) {
       _value = (isNumeric(newValue) ? newValue : null);
    };
```

The facade methods become simple delegates

```
    public String heartRate() {
       return heartRateAttribute().value();
    };

    public void heartRate(String newRate)     {
       heartRateAttribute().value(newRate);
    };

    private HeartRateAttribute heartRateAttribute() {
          return (HeartRateAttribute) _attributes.get("HeartRate");
      };
```

And finally the methods to load and save the attribute

```
    public void load() {
        _value = _facade.subject().latestAmountOf
                       (PhenomenonType.get ("Heart Rate")).amountString();
    };

    public void save()  {
       if (_value == null) return;
       new Measurement (new Quantity (_value, Unit.get("beats/min")),
                       PhenomenonType.get ("Heart Rate"),
                       _facade.subject(),
                       _facade.date());
    };
```

Even easier the second time. The only thing to be careful of is the loading and saving loops in vital facade. They need to cast to something that understands load and save. This calls for an interface.

```
  interface FacadeAttribute {
      void load();

      void save();
  };

  class LocAttribute implements FacadeAttribute {
  class HeartRateAttribute implements FacadeAttribute {
```

That makes the load and save loops much better. I could have used a superclass here, but I felt that all I really wanted was the typing of that load and save method, so an interface seems the easiest.

```
void loadAttributes () {
        Enumeration e = _attributes.elements();
        while (e.hasMoreElements()) {
            FacadeAttribute each = (FacadeAttribute) e.nextElement();
            each.load();
        };
    };
```

In my mind this desire to generalize the measurements is like a dog eager for attention. Well now its time for walkies. What is different between the heart rate attribute class and the breathing rate attribute class? It's the values for the phenomenon type and for the units. So I don't actually need to do inheritance here, I can just create one class which takes those values as parameters. The parameters will get set in the constructor.

```
class MeasurementAttribute implements FacadeAttribute{
  private String _value;
  private VitalsFacade _facade;
  private PhenomenonType _phenomenonType;
  private Unit _updateUnit;

  MeasurementAttribute (     VitalsFacade facade,
                             PhenomenonType phenomenonType,
                             Unit updateUnit) {
     _facade = facade;
     _phenomenonType = phenomenonType;
     _updateUnit = updateUnit;
  };
```

I change the load and save routines to use the new fields.

```
  public void load() {
      _value = _facade.subject().latestAmountOf(_phenomenonType).amountString();
  };

  public void save()  {
     if (_value == null) return;
     new Measurement (new Quantity (_value, _updateUnit),
                     _phenomenonType,
                     _facade.subject(),
                     _facade.date());
  };
```

So now I can set up the heart rate and breathing rates easily.

```
  void setupAttributes() {
      _attributes.put ("BreathingRate",
              new MeasurementAttribute (    this,
                                            PhenomenonType.get ("Breathing Rate"),
                                            Unit.get("breaths/min")));
      _attributes.put ("HeartRate",
              new MeasurementAttribute (    this,
                                            PhenomenonType.get ("Heart Rate"),
                                            Unit.get("beats/min")));
      _attributes.put ("LevelOfConsciousness", new LocAttribute (this));
};
```

Was it better to do it in two steps instead of one? My argument is that it is, because you can test after each step and spot any mistake more easily. I'll confess that in reality I didn't, and regretted it. One day I'll learn.

Now we can put our attention to the heart rate name and breathing rate name attributes. We could do these as separate satellites, they are completely separate in the facade after all. But we do know that they are linked in the domain model, so I'm prepared to use the information. First I add a method to measurement attribute to give the name.

```
  String name() {
     return _facade.subject().phenomenonOf(_phenomenonType).name();
```

```
   };
```

And then turn the facade method into a simple delegate

```
   public String heartRateName() {
      return heartRateAttribute().name();
   };
```

That was so easy I'm tempted to do the same for level of consciousness, but I will wait until I have a need, like building another choice box style attribute. Don't let that stop you though.

Checking the Validity of a Measurement

There are some restrictions on our measurements. We shouldn't enter negative heart rates, and numbers over (say) a 1000 are probably too large. We should put some checking in for those. In those cases the checks should be done by the domain model, by the phenomenon type class. The valid ranges will vary with the phenomenon type, so it seems the right place to look for something.

```
  Class PhenomenonType {
   private QuantityRange _validRange;

   public void validRange (QuantityRange arg) {
      _validRange = arg;
   };

   public boolean validRangeIncludes (Quantity arg) {
      if (_validRange == null) return true;
      return _validRange.includes(arg);
   };
```

We can check the valid ranges when we create the measurement.

```
   public Measurement (Quantity amount, PhenomenonType phenomenonType, Patient patient,
  Date whenObserved)
        throws OutOfRangeException{
      if (! phenomenonType.validRangeIncludes(amount)) throw new OutOfRangeException();
      initialize (patient, whenObserved);
      _amount = amount;
      _phenomenonType = phenomenonType;
      checkInvariant();
      _phenomenon = calculatePhenomenonFor(_amount);
   };
```

Now we have altered the interface of measurement creation to throw the exception. It is right here to throw a matched exception, the client should be prepared to handle the exception. So what should we do? The easiest thing is for the measurement attribute to set its value to null.

```
   public void save()  {
      if (_value == null) return;
      try {
          new Measurement (   new Quantity (_value, _updateUnit),
                              _phenomenonType,
                              _facade.subject(),
                              _facade.date());
      } catch (OutOfRangeException e) {
         _value = null;
         return;
      };
   };
```

That gives subtle but clear feedback on what went wrong.

The Shock Check Box

For the shock check box we have a slightly different case. The UI will want a Boolean attribute to work with in this case. Vitals facade will oblige, and delegate to a satellite.

```
   public boolean hasShock() {
      return shockAttribute().value();
   };
   public void hasShock (boolean newValue) {
      shockAttribute().value(newValue);
   };

   private ShockAttribute shockAttribute() {
      return (ShockAttribute) _attributes.get("Shock");
   };
```

We build the satellite in the usual way.

```
  class ShockAttribute implements FacadeAttribute {
   private Boolean _value;
   private VitalsFacade _facade;

   ShockAttribute (VitalsFacade facade) {
       _facade = facade;
   };
```

We can’t use the same interface for finding the value of the phenomenon, for shock is a phenomenon without a phenomenon type. What the facade wants to do is to ask the patient if he is in shock.

```
   public void load () {
       _value = _facade.subject().isPresent(Phenomenon.get("Shock"));
   };
```

The patient finds the latest observation of that phenomenon to answer the question.

```
  public Boolean isPresent (Phenomenon value) {
      return (latestObservation(value) == null ?
         null :
         new Boolean(latestObservation(value).isPresent()));

     };
```

The patient can answer null, which means “I have no information”. The check box, however, wants true or false, so the facade has to decide how to translate that information when asked. In this case it treats null as false.

```
  Class ShockAttribute
   public boolean value() {
      return (_value == null) ? false : _value.booleanValue();
   };
```

We don’t have unknown problem when updating.

```
  Class ShockAttribute
   public void save () {
       if (_value == null) return;
          new Observation (    Phenomenon.get("Shock"),
                               _value.booleanValue(),
                               _facade.subject(),
                               _facade.date());
     };
```

Adding the UI

Now it’s time to actually put the UI code in place. I could have built the UI incrementally as I did the facade, but I wanted to wait before learning all about awt. The UI is a single class VitalsWindow. To refresh your memory here’s what it looks like.

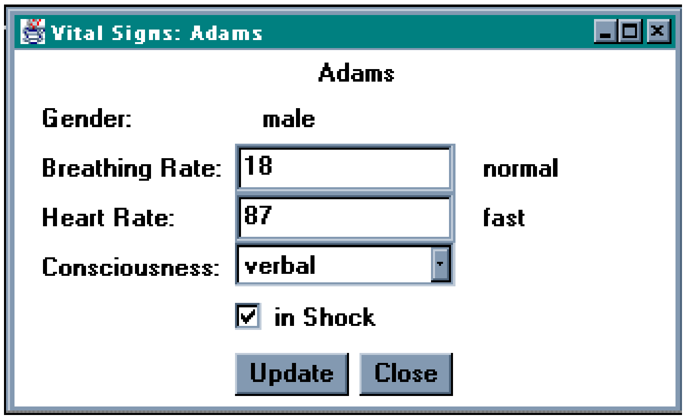


Figure 5 The vitals window

The window has fields for all of its interesting components, that is all the components that change during the life of the window. The static labels, including the patient's name, do not change once the window is created, so I will set them up during creation.

```
package vitalsWindow;
import vitalsFacade.*;
import java.awt.*;
import java.util.*;

class VitalsWindow extends Frame {
     private TextField breathingRate = new TextField ();
    private Label breathingRateName = new Label ();
    private TextField heartRate = new TextField ();
    private Label heartRateName = new Label ();
    private Choice levelOfConsciousness = new Choice ();
    private Checkbox shock = new Checkbox("in Shock");

     private Button updateButton = new Button ("Update");
     private Button closeButton = new Button ("Close");
```

Setting up the window involves several stages in construction.

```
  public VitalsWindow (VitalsFacade facade) {
     _facade = facade;
     loadChoices();
     loadData();
     setLayout(new BorderLayout());
     setTitle ("Vital Signs: " + _facade.name());
     add("North", nameLabel());
     add("Center", readingsPanel());
     add("South", buttonsPanel());
         resize(500,200);
  };
```

The window is given a facade to open on. All presentations would have such a link, their only link into the rest of the system. My first step then is to set up those components that have fairly static choices, in this case the pop up menu for level of consciousness.

```
  private void loadChoices() {
      loadLevelOfConsciousnessChoices();
  };

  private void loadLevelOfConsciousnessChoices() {
         Enumeration consciousnessValues = _facade.levelOfConsciousnessValues();
         while (consciousnessValues.hasMoreElements()) {
             levelOfConsciousness.addItem((String) consciousnessValues.nextElement());
         };
  };
```

I would like to set the values of the choice all at once, but choice does not support that behavior, so its one at a time.

The next step is to load the initial data into the component fields from the facade.

```
private void loadData () {
    _facade.load();
       breathingRate.setText (_facade.breathingRate());
       breathingRateName.setText(_facade.breathingRateName());
       heartRate.setText (_facade.heartRate());
       heartRateName.setText(_facade.heartRateName());
       levelOfConsciousness.select(_facade.levelOfConsciousness());
       shock.setState(_facade.hasShock());
  };
```

Now with all the data present, I can do the layout. The top level layout of the window uses the border layout. The patients name goes at the top ("North"), the rest of the display goes in the middle ("Center"), and the action buttons go at the bottom ("South").

The patient's name is easy. It is the only component in the top section, and it goes in centered horizontally.

```
private Label nameLabel() {
   Label newLabel = new Label (_facade.name());
   newLabel.setAlignment(Label.CENTER);
   return newLabel;
};
```

The majority of the measurements go in the middle, and I use a grid layout with three columns for that.

```
private Panel readingsPanel() {
   Panel newPanel = new Panel();
   newPanel.setLayout(new GridLayout(0,3));
   newPanel.add(new Label ("Gender: "));
   newPanel.add(new Label (_facade.gender()));
   newPanel.add(new Label (""));
   newPanel.add(new Label ("Breathing Rate: "));
   newPanel.add(breathingRate);
   newPanel.add(breathingRateName);
   newPanel.add(new Label ("Heart Rate: "));
   newPanel.add(heartRate);
   newPanel.add(heartRateName);
   newPanel.add(new Label ("Consciousness: "));
   newPanel.add(levelOfConsciousness);
   newPanel.add(new Label (""));
   newPanel.add(new Label (""));
   newPanel.add(shock);
   newPanel.add(new Label (""));
   return newPanel;
};
```

The two buttons go at the bottom

```
private Panel buttonsPanel() {
    Panel newPanel = new Panel();
    newPanel.add(updateButton);
    newPanel.add(closeButton);
    return newPanel;
};
```

Once you get the hang of the layout managers, they are in fact very easy to use. You can easily set up roughly where everything needs to go and the manager takes care of the details. You don't get fine positioning, but I usually don't want to bother my ugly little head with that anyway.

Once the window is created it needs to do something. Really there is only two things it needs to do, as indicated by the buttons.

```
public boolean action (Event theEvent, Object arg) {
    if (theEvent.target == closeButton) {
        closeAction();
        return true;
    };
    if (theEvent.target == updateButton) {
        updateAction();
        return true;
    };
    return false;
};
```

If the user hits close, we just want to forget the whole thing.

```
private void closeAction() {
    System.exit (0);
};
```

I wouldn't recommend that response if the window is part of a larger application, but it's fine for my limited purpose.

The update is nearly as simple, since all the hard work is delegated to the rest of the system.

```
private void updateAction() {
    _facade.breathingRate(breathingRate.getText());
    _facade.heartRate(heartRate.getText());
    _facade.levelOfConsciousness (levelOfConsciousness.getSelectedItem());
    _facade.hasShock(shock.getState());
    _facade.save();
    loadData();
};
```

The use of `loadData` at the end is to ensure that the window is synchronized with the domain model.

There is an awkward interface point as the window stands. If I change the measurement number for breathing rate or heart rate I need to change the string that indicates if is still fast. I can only do this by touching the model. For this case I will take the simple option and clear that text field when I change the number. It will get filled again when I update.

```
public boolean keyDown (Event theEvent, int key) {
    if (theEvent.target == breathingRate) {
        breathingRateChanged();
    };
    if (theEvent.target == heartRate) {
        heartRateChanged();
    };
    return false;
};

private void breathingRateChanged() {
    breathingRateName.setText("");
};

private void heartRateChanged() {
    heartRateName.setText("");
};
```

Summing Up

That was a lot of material, what are the key points to remember?

Firstly the principle of separating the application into well defined layers. Clear separate layers for presentation, application facade, and domain model make the code easier to maintain and test. Most of this application can be operated without the UI. This makes it easier to work with and test.

Its worth spending effort to get the behavior into the right place. That does not mean that the behavior has to start in the right place. Make a sensible first shot, but be prepared to refactor the code to improve it later.

Java is an application development language. The tools are somewhat primitive at the moment (I had terrible trouble with Café's debugger) but it is a full power language, much more than just something for creating applets.

On the subject of applets, this technique should also be useful there. A good way of working across a network is to load up a facade on the server and send it over to the client. There the presentation classes can communicate with the facade and, when necessary, send it back to the server to communicate with the domain model (I refer to this as *stretching* the facade in [Fowler]). You will need to put more state in the facade for this case. This is a technique I've seen several times in networked systems, and I think it would work well over the net.

References

[Beck] Beck K, Smalltalk Best Practice Patterns, Prentice Hall 1997

[Gang of Four] Gamma E, Helm R, Johnson R, and Vlissides J, Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, 1995

[Fowler] Fowler M. Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997

[Fowler, UML] Fowler M, UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley, 1997.

[UML] Booch G, Jacobson I, and Rumbaugh J. The Unified Modelling Language for Object-Oriented Development (version 0.91) Rational Software Corporation <http:\\www.rational.com>