

Assembling Classes at Runtime

By Peter Hut (p.h.hut@student.rug.nl) and Zef Hemel (z.hemel@student.rug.nl)

Abstract: This paper will describe the problem of assembling classes at runtime. Four solutions will be considered and their advantages and disadvantages explored. The solutions are: the Type Object pattern, the Adaptive Object-Model, the UML Virtual Machine and using a Dynamic Language.

1. The problem

Dynamic and configurable systems are the upcoming trend. Most systems are demanded to be flexible and easily extensible. This is needed to adapt the system to changing business rules (Rouvellou, 1998). An example of this is a system designed for administration of measurements done in a hospital. There are many different possible measurements. It is not feasible to design a subclass for each type of measurement. Furthermore it is very likely that new types of measurements need to be added to the system after deployment. The adding of new types of measurement should be possible for non-programmers. Additionally, the system has to keep running at all times. So shutting down the system, adding the new classes, recompiling the system and then starting it up again is not an option. The solution for this problem is assembling classes at runtime.

As in this example, there are three main reasons to need runtime assembling of classes. They are:

- The number of subclasses is unknown upfront
- The number of subclasses is huge
- Changes to the system have to be made without the system going down

2. Four solutions

For these problems four possible solutions exist:

- The Type Object Pattern
- Adaptive Object-Models
- The UML Virtual Machine
- Using a Dynamic Language

Each of these solutions will be discussed separately and at the end the solutions will be compared to one another.

2.1. The Type Object Pattern

The Type Object Pattern is a simple way to assemble classes at runtime (Johnson, 1998). An example will describe how it works.

A library contains a lot of different books. It also has multiple copies of one book. Some information about such a book, like whether it has been borrowed and by whom, is different for each copy. So therefore, an instance of the Book class is necessary for each copy. This leads to duplication of information.

A possible solution is to create a class for each title (for example 1984Book, TomSawyerBook and TheTimeMachineBook). In the class itself common data for all copies is stored; each instance represents a copy and stores data such as who has borrowed it.

This solves the data duplication problem, but is not very elegant. Another problem is that for every single title a subclass has to be created. This is not only a lot of work, but would also mean that for every addition of a title a new class has to be written.

The Type Object pattern solution suggests creating two classes: Title and Book. The Title class would contain data common to all copies of that title and the Book class contains a reference to the Title class and stores data specific to that particular copy (such as who borrowed it).

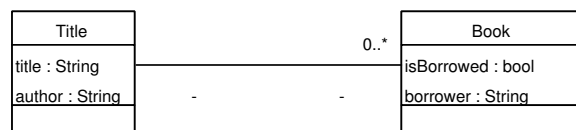


Figure 1 – Type Object pattern example.

Now, when a new copy of a book comes in, a new instance of the Book class can be created that references the Title it belongs to. If a new title comes in all that has to be done is the creation of an instance of the Title class and an instance of the Book class for each copy that came in.

Figure 2 gives the more general structure of the Type Object pattern. The Type Object pattern has two concrete classes, one that represents objects

and another that represents their type. Each Object-class instance has a pointer to its corresponding TypeObject object.

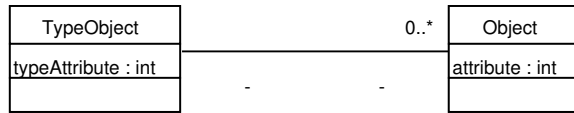


Figure 2 - The Type Object pattern.

2.2. Adaptive Object-Models

If it is not only necessary to create new classes at runtime, but also to customise the attributes, associations and the behaviour of these classes, then the Type Object pattern isn't fully suitable. In the library example no new attributes have to be defined, these are all known upfront, but in some cases the attributes that the objects will have is unknown. In many cases it is then possible to use an Adaptive Object-Model.

To make the customisation of attributes possible, the Type Object pattern is combined with the Property pattern (Foote, 1998). The property pattern makes it possible to dynamically add attributes to classes. The two patterns combined lead to the architecture as represented by Figure 3.

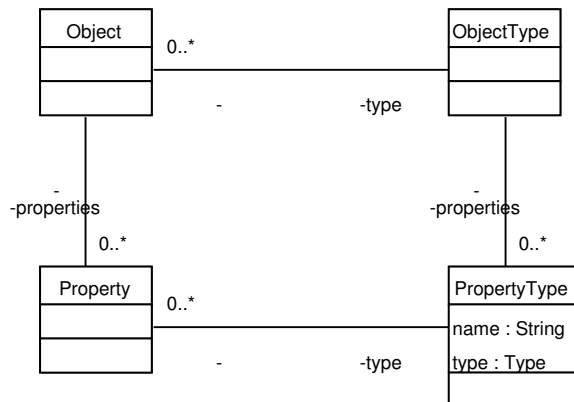


Figure 3 – Type Object and Property pattern combined.

The ObjectType object needs to store a list of properties (and their types) that its instances will have. And each Object needs to store a list of values for these properties.

To handle relationships between objects two subclasses of PropertyType are introduced: AssociationType and AttributeType. And two subclasses of Property: Association and Attribute. See Figure 4 - Handling associations (Yoder, 2003).

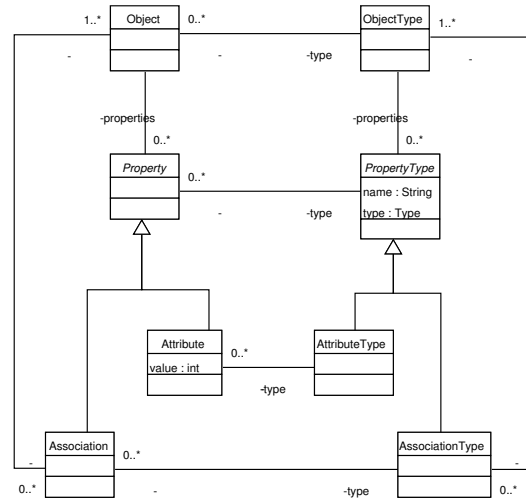


Figure 4 - Handling associations (Yoder, 2003)

To define the behaviour of an object, the Strategy pattern can be used. Strategies can, for example, be used to validate the values of properties.

In general a Strategy is an object that represents an algorithm. The Strategy pattern defines a standard interface for a family of algorithms so that clients can work with any of them. If an object's behaviour is defined by one or more strategies then that behaviour is easy to change. (Yoder, 2001)

Figure 5 is a UML diagram of applying the Type Object pattern twice with the Property pattern and then adding Strategies (called Rules in the diagram).

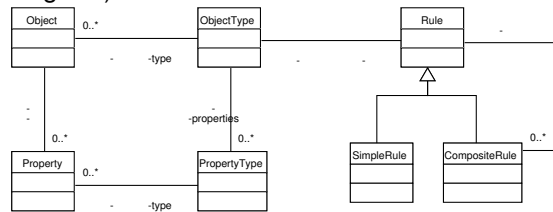


Figure 5 - Handling strategies.

To show how Adaptive Object-Models can be applied, an example will be used from (Yoder, 2001). In a hospital, measurements are done on people. The number of different kinds of measurements is very large and not known upfront. That's why it should be possible for hospital personnel (not just programmers) to define new kinds of measurements. Each measurement has different properties and relations.

To implement these requirements, the Adaptive Object-Model is used. This results in the class

diagram in Figure 6. There are two types of Observations. One type consists of discrete values such as blood type or eye colour (Trait). The other has continuous values such as weight or height (Measurement).

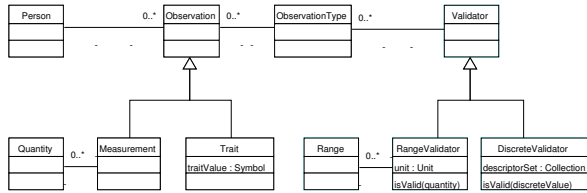


Figure 6 - Hospital example.

To create a new kind of observation a new instance of ObservationType has to be created. To be able to validate the values of the measurements, the Strategy Pattern is applied. The validators can be used by the ObservationType to check whether the value of an Observation is legal. Because of the two kinds of observations, for Measurements and traits, there are two kinds of validators. One represents a range of Measurements and one represents a set of Traits. A validator is just an algorithm telling whether a value is valid.

2.3. The UML Virtual Machine

The UML Virtual Machine is a totally different approach to software development, based on the Model Driven Architecture (MDA). MDA encourages efficient use of system models in the software development process and it supports reuse of best practices when creating applications. The main idea is to develop as much as possible of the application through design and less through implementation.

UML is a well-known standard for designing and specifying software. It knows different diagrams, such as the class diagram to define the structure of the software and other diagrams to specify the behaviour of the software. The most used diagrams for behaviour are state chart diagrams and collaboration diagrams. In version 1.5 of UML (OMG, 2003), action semantics were added to allow full specification of the behaviour of software. Additionally the Object Constraint Language (OCL) can be used to define constraints on objects.

It's very common to first design an application in a UML application (like Visio or Rational Rose) and then implement it in some programming language. Sometimes the UML diagrams are

used to automatically generate some of the code, interfaces and stubs for the classes and methods.

Executable UML (Mellor, 2002) takes this a step further. It eliminates the implementation phase. The Executable UML application will read the UML specification and compile it to executable code. This, however, does still mean that the system has to be restarted each time the object model is adapted. So it can't assemble classes at runtime.

UML Virtual Machines (Riehle, 2001) on the other hand are different. UML Virtual Machines will read the UML specification and interpret it on the fly. While the application is running, the UML specification can be changed. New classes, attributes and associations can be added and behaviour can be defined, all without shutting down the application. Algorithmic detail can be added as hand-programmed policy classes that fit into a well-defined extension architecture.

The UML Virtual Machine, in contrast to the Type Object pattern or Adaptive Object-Models, is not a set of patterns that can be used, but rather a runtime environment that runs the application. It's a product that can be bought. At this moment, though, there is no working implementation of the UML Virtual Machine. (Riehle, 2001) is working on one, but it's only slowly progressing. There's also a group at the university of Massachusetts (Lall, 2004) working on an implementation (expecting to deliver in May, 2005).

This system could be used as a solution to assemble classes at runtime, as the UML specification can be changed at runtime, including defining new classes. The problem is that a UML tool will have to be used to design the new classes, which not every layman will understand. On the other hand, it is imaginable that an application can be created that interfaces with the UML Virtual Machine and exposes a user-friendly GUI to the user to define new classes, associations and behaviours.

Unfortunately there is no working implementation of the UML Virtual Machine yet. Therefore only guesses can be made about its opportunities.

2.4. Dynamic Languages

Dynamic Languages such as Python and Ruby allow runtime assembling of classes and at runtime adaptation of classes and objects, by design; it is part of the dynamic nature of the

languages (Mertz, 2003). Everything that can be done at “compile” time can be done at runtime.

It is possible to generate classes and methods; and attach new properties and methods to classes and individual objects. To show this, a simple Python example is included in Listing 1 at the end of this paper. The example implements a couple of elements of the hospital example as presented in section 2.2.

3. Comparing the solutions

The table on the next page gives a quick overview of the differences between the discussed solutions. In the next sections we’ll discuss what each row means.

3.1. Design complexity

Design in this context means how much the complexity of the design of your application increases because of adding class assembling features.

Before the Type Object pattern or an Adaptive Object-Model can be used effectively it is necessary to understand the principles of the chosen method. These are not always obvious. In practice both will result in more design complexity, in particular Adaptive Object-Models, which requires quite a complex class structure.

As an application can be designed for a UML Virtual Machines as usual and the changing of classes at runtime comes with using the UML Virtual Machine, the design complexity is low.

Because class assemblage is very common and natural in a Dynamic Language, no special design tricks have to be applied. Therefore the design complexity of adding features to assemble classes at runtime is low, but the availability of these features might not be obvious from the design.

3.2. Implementation complexity

When the Adaptive Object-Model is used it will result in a more difficult architecture than when only the Type Object pattern is used. Consequently the first will be harder to implement.

When it is possible to use the UML Virtual Machine as a component off the shelf, not much implementation work is needed, except maybe a graphical user interface to allow laymen to create new structures. But of course, this all depends on the implementation of the UML Virtual Machine, which does not exist as of yet. If an entire UML Virtual Machine has to be implemented, the

implementation complexity would become very high.

To use a Dynamic Language, familiarity with the language is necessary and a Dynamic-Language interpreter has to be used or integrated in the application. Implementing the assembling of classes at runtime is very simple.

3.3. Implementation language constraints

For the Type Object, Adaptive Object-Model and UML Virtual Machine no particular language features are required. For the Dynamic Language solution to be used, a Dynamic Language is necessary (obviously).

3.4. Application-embedded domain knowledge

When the Type Object pattern is used the classes that can be assembled at runtime can only be changed in minimal ways and are, for the most part, designed with the domain knowledge in mind. Therefore a lot of domain knowledge is embedded in the application.

When an Abstract Object-Model or Dynamic Language is used, even the attributes of the classes can be changed at runtime. This means the system can still be adapted to a domain in which it is used, even while it is running. Therefore when designing a system using an Abstract Object-Model or Dynamic Language, only limited domain knowledge has to be embedded in the application, as a lot can be defined at runtime. This also depends on the tools the system has available to change or add classes at runtime. In most cases the purpose of using an Abstract Object-Model would be to let the user adapt the system to domain in which it is used.

The UML Virtual Machine, like Dynamic Languages, allows you to assemble any kind of class at runtime as desired, with any kind of behaviour desired. Therefore very little domain knowledge has to be embedded into the application upfront.

3.5. Change properties/association at runtime

With an Abstract Object-Model, UML Virtual Machine and Dynamic Language it is possible to add and remove properties of a class. With just the Type Object pattern it is not.

3.6. Change behaviour at runtime

The Type Object pattern does not allow you to change behaviour at runtime, for an Adaptive Object-Model the behaviour can only be changed by using the Strategy pattern (see Figure 5). The user, in many cases, can choose one of pre-

	Type Object pattern	Adaptive Object-Models	UML Virtual Machine	Dynamic Languages
Design complexity	Medium	High	Low	Low
Implementation complexity	Low	High	Low/Very High	Low
Implementation language constraint	None	None	None	Should be dynamic
Application-embedded domain knowledge	Much	Little	Much/Little	Little
Change properties /association at runtime	No	Yes	Yes	Yes
Change behaviour at runtime	No	Limited	Yes	Yes
Flexibility	Moderate	High	Very High	Very High
Runtime overhead	Low	Medium	High	Medium

defined strategies which is somewhat limiting, but enough in many cases.

Depending on the chosen way to implement behaviour in the UML Virtual Machine, the system's behaviour can be fully changed at runtime. In a Dynamic Language it is possible to generate code at runtime and instantly interpret it. So any kind of logic can be generated at runtime.

3.7. Flexibility

Using the Type Object pattern means that the new classes that can be generated will have pre-defined properties and behaviour.

When an Abstract Object-Model is used the properties of the (at runtime assembled) classes can be modified and behaviour can be chosen from pre-defined strategies.

The UML Virtual Machine allows the creation of whole new programs and there are no constraints. The same goes for the Dynamic Language approach.

3.8. Runtime overhead

If the Type Object pattern is used instead of a normal class-subclass relationship it would mean slightly more runtime overhead than usual, as the objects need to keep track of their TypeObject-relation themselves. Also some requests to the object might need to be forwarded to its TypeObject.

If, on the other hand an Adaptive Object-Model would be used it, would give more overhead than the Type Object pattern, as it keeps track of properties and relationships as objects.

When the UML Virtual Machine is used, the object model is re-implemented on the object model of the implementation language. On top of that the application is run. This will of course mean a lot more overhead compared to implementing the application directly in the implementation language.

For Dynamic Languages, the runtime overhead is hard to determine. It highly depends on the implementation of the Dynamic Language's interpreter. Most implementations work like the Adaptive Object-Models, others emit machine code at runtime for the at-runtime generated classes. In that case there is hardly any runtime overhead.

4. Conclusion

If a system needs to be created in which:

- there are a large amount of sub-classes of one class;
- the number of sub-classes upfront is not known; or
- it is necessary to make changes without the system going down;

then a solution can be used that allows the assembling or change of classes at runtime. Four solutions and their advantages and disadvantages were discussed.

The Type Object pattern is quite simple and can be used in any design but offers limited flexibility.

The Adaptive Object-Model is more flexible and allows the system to be adapted later to better fit

organizational changes. This means less domain knowledge is embedded into the application compared to a normal situation or when the Type Object pattern is used. The drawback of using an Adaptive Object-Model is that the system will be more difficult to understand.

An entirely different solution is to use a UML Virtual Machine. The design of the system will be the same as usual, but it would be possible to also give the user the option to add and change classes at runtime. From this the same advantage is obtained as from an Adaptive Object-Models, but not the disadvantage that the design is more complex than usual. The problem is that currently no finished implementation is available of a UML Virtual Machine and implementing one would mean quite an investment.

The last solution that was discussed, Dynamic Languages, allows for a normal design and implement of a system, and also allows the addition of features to allow users to make runtime changes to the classes. The disadvantage in this case is the restriction to a limited group of implementation languages.

5. References

Foote, B., Yoder, Y.W., "Metadata and Active Object Models", Proceedings of Plop98. Technical Report #wucs-98-25. Washington University Department of Computer Science. URL:
http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P59.pdf (1998)

Johnson, R., Wolf, B., "Type Object", Pattern Languages of Program Design 3, Addison Wesley. (1998)

Lall, A., Malinowski, A., Qureshi, M., Solaiappan, K., "UML Virtual Machine", URL:
<http://umlvm.cs.umb.edu> (2004)

Mellor, S.J., Balcer, M.J., "Executable UML: A Foundation for Model Driven Architecture", Addison-Wesley Pub Co; 1st edition, ISBN 0201748045 (2002)

Object Management Group Inc., "Unified Modeling Language specification 1.5" (2003)

Mertz, D., "A Primer on Python Metaclass Programming", URL:
<http://www.onlamp.com/pub/a/python/2003/04/17/metaclasses.html> (2003)

Riehle, D., Fraleigh S., Bucka-Lassen, D., and Omorogbe, N., "The Architecture Of A UML virtual machine", Proceedings of OOPSLA'01. ACM Press, New York, 327-341. (2001)

Rouvellou, I., Degenaro, L., Rasmus, K., Ehnebuske, D., McKee, B., "Extending business objects with business rules". Proceedings on Software Engineering: Education & Practice. Page(s): 238 - 249. (1998)

Yoder, J. W., Johnson, R., "The Adaptive Object-Model Architectural Style". URL:
<http://www.adaptiveobjectmodel.com/WICSA3/ArchitectureOfAOMsWICSA3.htm> (2003)

Yoder, J. W., Balaguer, F., Johnson, R., "Intriguing technology from OOPSLA: Architecture and design of adaptive object-models", December 2001 ACM SIGPLAN Notices, Volume 36 Issue 12

Listing 1: Runtime assembling of classes in Python 2.2+

```
class InvalidInputException(Exception):
    """The exception that's raised on invalid input"""
    pass

def generateClass():
    """Generates an empty class"""
    class Dummy(object):
        pass
    return Dummy
```

```

def generateDiscreteValidatingSetter(attr, allowedValues):
    """Generates a setter that validates discrete values"""
    def validateDiscrete(self, value):
        if value in allowedValues:
            setattr(self, '__'+attr, value)
        else:
            raise InvalidInputException
    return validateDiscrete

def generateRangeValidatingSetter(attr, lowerBound, upperBound):
    """Generates a setter that range validates a value"""
    def validateRange(self, value):
        if lowerBound <= value and value <= upperBound:
            setattr(self, '__'+attr, value)
        else:
            raise InvalidInputException
    return validateRange

def generateGetter(attr):
    """Generates a simple getter"""
    def getter(self):
        return getattr(self, '__'+attr)
    return getter

# Generate classes
BodyLengths = generateClass() # Composite class
LengthMeasurement = generateClass() # General length measurement

# Add validating properties to LengthMeasurement
LengthMeasurement.value = property(generateGetter('value'), \
    generateRangeValidatingSetter('value', 0, 300))
LengthMeasurement.unit = property(generateGetter('unit'), \
    generateDiscreteValidatingSetter('unit', ['meters', 'inches', 'feet']))

# Instantiate and use the classes the natural way
lengths = BodyLengths()
lengths.arms = LengthMeasurement()
lengths.arms.value = 30
lengths.arms.unit = 'inches'

# And now the same, the dynamic way
# Obtain property names/values from somewhere
propname = 'fullbody'
valuepropname = 'value'
valuepropvalue = 1.85
unitpropname = 'unit'
unitpropvalue = 'meters'
# Use them
setattr(lengths, propname, LengthMeasurement())
setattr(getattr(lengths, propname), valuepropname, valuepropvalue)
setattr(getattr(lengths, propname), unitpropname, unitpropvalue)

# Assign invalid values to properties
lengths.fullbody.value = 500 # Exception
lengths.fullbody.unit = 'yards' # Exception

```