Patterns and Anti-Patterns in Modelica

Dr. Michael M Tiller
Emmeskay, Inc.
Plymouth, MI, USA
mtiller@emmeskay.com

Abstract

In 1977, Christopher Alexander, Sara Ishikawa and Murray Silverstein published the book "A Pattern Language: Towns, Buildings, Construction" [1]. Although the topic of the book was architecture, it inspired Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in their approach to pattern based software development. This ultimately led to the publication, in 1994, of the book "Design Patterns: Elements of Reusable Object-Oriented Software" [2] (also known as the "Gang of Four" or "GoF" book) which launched a major movement in the software development community toward pattern based software design. The idea behind the pattern movement is to formally identify sound design solutions to common problems.

Since the publication of "Design Patterns" there have been numerous books published on the topic of software patterns. Several of these books dealt with the sub-topic of anti-patterns [3,4]. In contrast to a normal pattern, anti-patterns are an attempt to identify common bad practices and ways they can be refactored using sound design patterns.

The emphasis of the pattern community is, understandably, on object-oriented languages with procedural semantics. This paper will build on previous work [5] identifying patterns in Modelica. These design patterns include how medium properties can be handled in a flexible way, how to deal with systems with varying causality and differential index, idealized plant control and, finally, coordination between models. In addition, this paper includes some extensive discussion of anti-patterns to avoid redundant code, awkward data management and inflexible models.

This paper continues the discussion on patterns within the Modelica community with the hope that this will encourage others to contribute patterns of their own. One obvious benefit of such efforts will be additional resources for Modelica developers to make the process of developing models in Modelica easier. In addition, we expect that many of the patterns discussed will also generate proposals for improving the Modelica language through new features and semantics.

Keywords: patterns, anti-patterns

1 Background

When Alexander *et. al.*, published their work, each pattern included four principle aspects, the pattern name, the context in which the pattern applied, the problem the pattern attempted to address and the proposed solution. This paper will focus primarily on the problem and solution.

In September of 2006, Mark Dominus wrote an essay in his blog [6] in which he concluded with the following statement:

> *"Patterns are signs of weakness in programming languages. When we identify and document one, that should not be the end of the story. Rather, we should have the long-term goal of trying to understand how to improve the language so that the pattern becomes invisible or unnecessary."*

This statement triggered quite a bit of controversy and many people argued with this assertion, not the least of which was Ralph Johnson [7], co-author of the original "Design Patterns" book who argued that patterns are simply manifestations of high level concepts beyond the scope of language semantics.

In this paper the assumption will be that the truth lies somewhere in between. Some patterns are simply manifestations of design decisions made in the development of a given language. Other patterns appear to address missing expressiveness in the underlying language. In some cases, patterns are simply introduced to encourage consistency and readability above and beyond what is really the purview of language designers. Along with the patterns themselves some discussion will be included indicat-

ing to what degree each pattern (or anti-pattern, as the case may be) could be mitigated by changes in the language or standard library.

2 Design Patterns

2.1 Architecture Pattern

2.1.1 Problem

While building models to support a variety of systems and/or subsystems a large collection of models with many structural similarities have been developed. Adding additional models involves constructing models either by copy and pasting large chunks from previous models or dragging and dropping the complete model from scratch.

There are two distinct issues being discussed. The first is the amount of work required to create a new model. The second is about redundancy between models. This pattern focuses on the former and the latter is discussed as part of the DRY anti-pattern in Section 3.1.

2.1.2 Solution

When significant structural similarities exist between system or subsystem models then these models can be formulated in terms of architectures. In doing so, each model becomes simply a variation of the architecture with the various interfaces replaced by implementations that are appropriate for that specific model.

For example, building vehicle models by simply dragging all the usual constituents (*e.g.* engine, transmission, chassis, *etc*) into a diagram can be quite time consuming and tedious whereas building them as variations from a standard vehicle architecture (e.g. [8]) can greatly reduce the overhead of creating and managing such models.

2.2 Singleton Pattern

2.2.1 Problem

When building libraries of models it is sometimes necessary to design the library in such a way that there is a single instance somewhere that includes a definitive reference for some information. The basic idea is that within some scope there is exactly one such instance. The challenge is not simply how to access that "singleton" object but how to design the library so that this is handled well for users.

2.2.2 Solution

In languages like Java and C++, the use of the `static` qualifier on members provides a language supported mechanism for ensuring uniqueness within a given program. The closest equivalent in Modelica would be a variable declared as `constant`. However, the values of constants cannot be changed so while semantically similar, this is not adequate to achieve the singleton pattern. Instead, the use of `inner` and `outer` qualifiers is a more common choice. By referring to an `inner` instance it is possible for all outer references to act simply as "pointers" to a single object. The use of `inner` and `outer` has an additional advantage (or disadvantage, depending on how strict you need to be) which is that they can be nested inside each other.

Two immediate examples of the singleton pattern can be found in the Modelica Standard Library. The first is in the Multibody library. The design of the library is such that it requires that there is exactly one instance of the so-called "world" object in the system to provide a reference coordinate system. Another example, which exploits the ability to nest one subsystem (requiring its own internally unique singleton) inside another subsystem, can be seen in the StateGraph library [9].

An example where the use of `inner` and `outer` is not currently sufficient is in dealing with "many to many" interactions. For example, consider a model of the solar system. Each planet exerts a gravitational force on all the others. While it is possible to implement each gravitational force as an individual component that connects between every combination of planet instances in a system, it is more convenient and scalable to have some kind of (singleton) intermediary component that is somehow aware of all planet instances and can, within the context of that single model, handle all interactions. Similar "many to many" requirements can be found in systems where collisions are possible between multiple bodies.

2.3 Medium Model Pattern

The medium model pattern is more generally called the "abstract factory" or "kit" pattern. However in Modelica the most common use is to represent medium properties. For this reason the name "medium model" is used since it is more familiar to the target audience of this paper having appeared in previous work [10, 11].

2.3.1 Problem

In a nutshell, the medium model pattern shows up in models that include multiple configurable types that must be, in some way, consistent with each other. As already mentioned, this is something that occurs often when characterizing the medium of a given fluid system. The configurable types typically include (but are not limited to) connector definitions and some kind of property evaluation model. The essential point is that many assumptions about a fluid bind the definition of the connectors and the property evaluation together (*e.g.* the number of species). For example, it would not make sense to combine the connector a multi-species gas with the properties of oil.

2.3.2 Solution

As mentioned previously, this approach is called the "abstract factory" pattern in other languages and it is usually achieved through abstract methods that return instances abstract types. The consistency is assured by the implementation of the abstract factory. Because Modelica lacks methods or even any appreciably dynamic object creation, the same effect is achieved in Modelica using replaceable packages.

By using replaceable packages, it is possible for models to reference constants and types defined in the "constraining package" defined or implied in the replaceable definition. A given "implementation" (*e.g.* a specific medium) can then redefine these types and constants in a consistent way (*e.g.* so they all represent the same medium). The following sample code demonstrates the use of this pattern. First, an abstract model of the medium must be defined:

```
partial model AbstractMedium
  constant Integer n "# of Species";
  connector Fluid
    Pressure p;
    flow MassFlowRate m_dot;
    MassFraction Xi[n-1];
    flow MassFlowRate mXi_dot[n-1];
  end Fluid;

  partial block Properties
    input Pressure p;
    input MassFraction Xi[n-1];
    output SpecificEnergy u;
    output SpecificEnthalpy h;
  end Properties;
end AbstractMedium;
```

Based on this abstract medium model, component models can then be written that rely on information from the medium model but without knowledge of what specific medium model is being used:

```
model Component
  replaceable package MediumModel =
    AbstractMedium;
  MediumModel.Fluid c;
  MediumModel.Properties props(
    p=c.p,X=c.X);
equation
  // equations in this component
  // can reference the pressure
  // at the connector, c.p, or
  // properties of the fluid,
  // e.g. props.h
end Component;
```

Finally, an implementation of the medium model can be created by extending from the abstract medium model:

```
package RealMedium
  extends AbstractMedium(nspecies=2);
  redeclare model extends Properties
  equation
    // This model may include things
    // like property calculations or
    // an equation of state.
  end Properties;
end RealMedium;
```

One usability issue with this pattern is that when it is used in conjunction with the transport of physical information or behavior it is somewhat counter intuitive since the redefinitions of the medium model are propagated from "top down" when users think, at least conceptually, that the information should be propagated through connections. For example, the Component model in the previous sample code would need to be instantiated with a modification specifying the medium model, e.g.

```
Component comp(
  redeclare package MediumModel =
   RealMedium);
```

whereas most users would expect that "somehow" the type of medium was dictated by what the instace was connected to. While this is not an issue with the

pattern in general, it is an important consideration for language designers and tool vendors.

2.4 Adapter Pattern

2.4.1 Problem

When working with architectures, it is necessary for the subsystem models to be developed so that they satisfy the interface prescribed by the architecture. However, there are many cases where the subsystem model might be developed independently from an architecture and as a result it does not conform to any specific interface. This situation may come about because the subsystem models were developed before the architecture or perhaps they were developed in an architecturally neutral way to avoid dependence on a particular architecture or to support multiple architectures.

2.4.2 Solution

In these circumstances, it may be necessary to develop adaptor components. Such components provide a mapping from the interface that the subsystem currently has to the interface that is to be supported. There are two variations of this pattern. In the first case, the subsystem is developed independently from any particular interface. In this case, the development of an adapter for the subsystem is a "one time only" process since other subsystems are unlikely to share the exact same interface (and if they do, they should probably be refactored as described in Section 3.1).

The other case is where the subsystem has been developed according to a specific interface (one that presumably other subsystems satisfy). In this case, a general adaptor could be constructed that maps one interface onto another. Such an adaptor could then be used as an adaptor for multiple subsystems. This kind of adaptor pattern can also be used to implement compatibility between comparable interfaces across different architectures.

The following code fragment shows an example of how the adaptor pattern is implemented. First, let us consider the one potential (and greatly simplified) interface for a vehicle model:

```
partial model VehicleInterfaceA
  RealOutput vehicle_speed;
end VehicleInterfaceA;
```

Several vehicle models might be developed using this interface, *e.g.*

```
model Vehicle1
 extends VehicleInterfaceA(
   vehicle_speed=…);
end Vehicle1;

model Vehicle2
 extends VehicleInterfaceA(
   vehicle_speed=…);
end Vehicle2;
```

Now consider an alternative vehicle model interface and system architecture definition:

```
partial model VehicleInterfaceB
 RealOutput v_vehicle;
end VehicleInterfaceB;

partial model ArchitectureB
 replaceable VehicleInterfaceB vehicle;
 …
end ArchitectureB;
```

An adaptor between the different interfaces could be developed as follows:

```
model VehicleAdaptor_A2B
 extends VehicleInterfaceB;
 replaceable VehicleInterfaceA vehicle;
equation
  connect(vehicle.vehicle_speed,
          v_vehicle);
end VehicleAdaptor_A2B;
```

Using this adaptor it is possible to build a system that utilizes `ArchitectureB` but uses an implementation of `VehicleInterfaceA` as follows:

```
model System
 extends ArchitectureB(
   redeclare VehicleAdaptor_A2B(
     redeclare Vehicle1 vehicle));
end System;
```

2.5 Parametric Behavior Pattern

2.5.1 Problem

In acausal modeling most components tend to describe the flow of some conserved quantity explic-

itly in terms of the across variables (*e.g.* i=v*R). Other components describe the flow of conserved quantities implicitly in terms of constraints (*e.g.* an ideal voltage). However, it is often quite useful to be able to describe components that describe the flow of conserved quantities in terms of both implicit and explicit relations depending on the state of the component. The simplest example of such a component is an electrical diode which either allows no current (explicit case) or no voltage drop (implicit case). Another slightly more complicated case would be a clutch which computes transmitted torque explicitly in terms of dynamic friction when disengaged or slipping but computes torque implicitly in terms of a kinematic relation when locked.

2.5.2 Solution

One "easy" way to describe such behavior is to compromise on the ideal nature of the behavior. For example, where an ideal diode might describe the implicit and explicit behavior using the equations v=0 and i=0, respectively, a compromise model sacrifices the idealization might use the equations v=G*i and i=v*R, where G is chosen to be very small (to approximate the v=0 case) and R is chosen to be very large (to approximate the i=0 case). The result of this compromise is that the behavior is now completely explicit in nature. However, another consequence of this "easy" solution is that the system of equations is very likely to be poorly conditioned which means the system will be stiff and slow to simulate.

A "better" solution (from the modeler's perspective at least) is to capture the ideal behavior somehow. Not only is this possible but it can be a very elegant and useful way to approach such problems. The basic premise (which is presented in greater detail in [12]) is to introduce a third variable and describe the behavior of the original variables in terms of the third parametric variable. This approach is frequently used in geometric applications where it is not possible to use a particular coordinate axis as an independent variable to describe a line or surface. The same issue is present, for example, in a diode where it is not possible to write current explicitly in terms of voltage nor is it possible to write voltage explicitly in terms of current. However, it is possible to write both in terms of a third parametric variable, *e.g.*

```
off = s<0;
v = if off then s*unitV else 0;
i = if off then 0 else s*unitC;
```

where unitV and unitC are defined as follows:

```
import Modelica.SIunits.Voltage;
import Modelica.SIunits.Current;
constant Voltage unitV=1;
constant Current unitC=1;
```

Analysis of this parametric approach shows that describing this kind of behavior is not simply an issue with the expressiveness of the underlying modeling language but with the solution method. While some basic solution techniques exist to deal with component models that are either implicit or explicit, the ability of a component to function in both ways creates additional complications for the underlying solver.

One such complication is that switching between two different sets of equations during a simulation always brings with it the risk that the differential index of the system might change. As such, the posed problem could be a variable index system. In fact, a clutch model normally leads to a variable index system when modeled using the parametric behavior pattern. However, by understanding this in advance it is possible to differentiate the equations such that the index is no longer variable. For this reason, it would be very useful if investigation into this issue showed that a general algorithm could be developed along similar lines. Such an algorithm would most likely benefit from language features that directly supported this pattern.

2.6 Perfect Control Pattern

2.6.1 Problem

Physical models typically include sensors and actuators and these are in turn normally connected to some kind of control system. One of the burdens that model developers face is to provide some kind of actuator control strategy in addition to the base physical models. In many cases, the model developer is not particularly interested in the dynamics of the controller but they need some function in the model to determine how the actuator will behave and so therefore implementation of controls is unavoidable. Such implementations often take time both to construct and calibrate and many times they do not add any significant value to the model.

2.6.2 Solution

It is important to point out that this pattern is very specific to cases where the model developer simply wants a very good controller but they don't

need to be very concerned about how such a control strategy would actually be deployed or implemented in hardware. In these specific circumstances, it is often possible to rely on a "perfect" control strategy to control the device. For example, consider a simple SISO plant model defined as follows:

```
model PlantModel
  input Real u;
  output Real y;
protected
  Real dy = der(y);
equation
  2*der(dy) + dy + 4*y = u;
end PlantModel;

model ClosedLoop
  PlantModel plant;
protected
  Real ybar = max(0,time-2);
equation
  plant.u = 10*(ybar-plant.y);
end ClosedLoop;

model PerfectControl
  PlantModel plant;
protected
  Real ybar = max(0,time-2);
equation
 ybar = plant.y;
end PerfectControl;
```

The simulation results from both types of control can be seen in Figure 1. The basic idea of this pattern is rather than including an explicit equation for the command to the system an equation prescribing the output is used. This equation for the output acts as an implicit equation for the input. It should be pointed out that this type of approach is limited to cases where the plant model is sufficiently invertible.

Despite this limitation, this is a useful pattern that can be used in conjunction with some surprisingly complex systems. For example, this approach is the same approach that is employed to create "backward" drive cycle models (models where the vehicle speed is prescribed and the system resolves the torque required to meet the speed profile). In addition, this same pattern can be used in conjunction with actuators like clutches and valves.

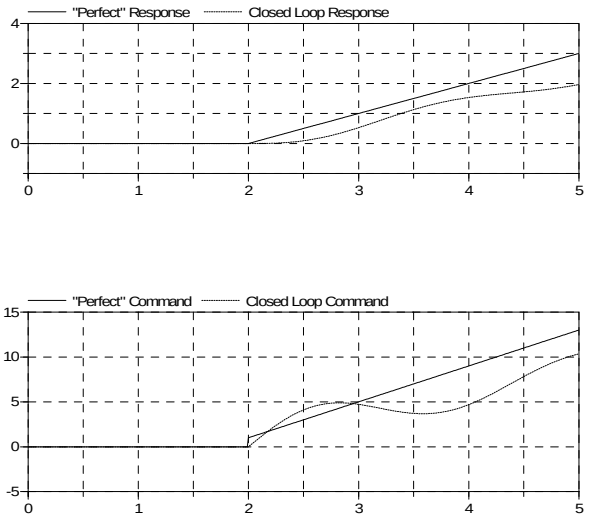

Figure 1: Example of "Perfect" Control Pattern

3 Anti-Patterns

Patterns are primarily useful for intermediate to advanced users who, having written some substantial amounts of code, are able to recognize the emergence of patterns and are interested in understanding how patterns can help them be more productive (as well as improve consistency and readability among project members).

However, Modelica is still a relatively new technology with many new users. As a result, anti-patterns are probably at least as important as patterns. The reason is that anti-patterns can help novices to recognize weaknesses in code they have written. As such, anti-patterns are almost immediately applicable. This section introduces several anti-patterns and discusses refactoring approaches associated with each pattern.

This is not to say that anti-patterns only apply to novice users. Because Modelica improves developer productivity, it is very easy to write a large volume of code only to realize in hindsight that some anti-patterns have developed. As a result, the material in this section is applicable to a wide range of users. As such, the material in the anti-patterns section should be of particular interest to tool vendors since refactoring typically requires tool support.

3.1 DRY Anti-Pattern

3.1.1 Problem

By far, the most common anti-pattern is the use of "copying and pasting" model code between models. While this happens for a wide variety of reasons most of them are ultimately because users are not aware of the various mechanisms within Modelica

for code reuse. In software development there is something known as the "DRY principle" where DRY is an acronym for "Don't Repeat Yourself". The DRY anti-pattern is one where the DRY principle has not been followed.

The reason that the DRY principle is so important (and which has led to the motto that "redundancy is the root of all evil") is that redundancy creates many problems. Not only does it lead to inefficiency when building models it also means significantly more work when maintaining those same models.

3.1.2 Solution

While this is a very common anti-pattern, the good news is that Modelica contains a rich supply of language features to help combat it. The first language feature all users should become familiar with is inheritance (specifically, the `extends` keyword). Once developers understand inheritance they should investigate the architecture pattern (described previously in this paper) which hinges on the `replaceable` and `redeclare` keywords.

One issue that prevents addressing this anti-pattern is tool support for refactoring. This manifests itself in several ways. First, it should be possible for users to change the names of components and/or classes and be assured that all references that use those names are also adjusted (ideally even if they are not even currently loaded). Furthermore, refactoring of existing code often involves the exercises of identifying commonality between existing models, composing base classes that contain this common code and then extending the original models from the base classes. Without tool support, such refactoring can be very time consuming.

3.2 Kitchen Sink Anti-Pattern

3.2.1 Problem

Another common anti-pattern is the "kitchen sink" anti-pattern. There are two variations of this pattern. For component models, the anti-pattern manifests itself as component models with too many equations by lumping several distinct types of behavior together into a single component. For subsystem models, the anti-pattern manifests itself in diagrams with an unnecessarily large number of components.

3.2.2 Solution

In both of these cases, a "divide and conquer" approach is required. For the component variation, this means building component models that heed Occam's Razor, "*entia non sunt multiplicanda praeter necessitatem*". In practical terms, this means building component models that attempt as much as possible to describe individual effects (*e.g.* inertia, compliance, dissipation, *etc*).

In the case of subsystem models, refactoring is typically a matter of nesting some tightly coupled subset of components into a subsystem of their own. Again, tool support is an issue here. Simulink has a very convenient feature to take a group of selected components and lump them into a subsystem model. Modelica tool vendors would do well to recognize the value of such functionality (and users would do well to remind them).

3.3 Literal Data Overload Anti-Pattern

3.3.1 Problem

Modelica supports a wide range of ways to deal with data handling. In theory, users can bring data in from an external database, they could read it from external files, *etc*. However, the simplest way to import data into Modelica models is to enter it literally (*e.g.* `parameter Real table[:,2] = [0, 1; 1, 2; 2, 3; …]`). While there is nothing wrong with this *per se*, it leads very quickly to the literal data overload anti-pattern. The pattern is characterized by the tendency of models to rely on literal data. While this is acceptable for simple component models, this creates two problems with more complex models. The first complication is that entering tables of data is often quite inconvenient. The second complication is that often times any given parameter cannot be changed independently. For example data associated with a given electric motor might bring together the rotor inertia, internal resistance, bearing friction, etc into a set of parameters. If a different motor is to be used, it is not simply a matter of changing a single parameter value but the entire set must be exchanged for another consistent set representing a different motor.

3.3.2 Solution

Both issues of entering literal data and parameter set consistency can be handled by creating records to represent such parameter sets and including the literal data only in the context of the record definitions. In addition, it is advisable to make use of the `choices` annotation so tools understand how the data will be used. The result of such refactoring is that users will only see opaque references to complex and/or voluminous data sets rather than vast expressions containing literal data. It is also a advisable to provide useful descriptions of the data sets so tools

can provide users with clear descriptions of available choices.

3.4 Parameter Data Overload Anti-Pattern

3.4.1 Problem

The previous anti-pattern addresses some of the issues associated with models that require large amounts of data. While the aggregation prescribed for refactoring reduces the number of individual parameters a complex system with many components can still contain large numbers of parameter sets (and even the aggregations themselves may have an unwieldy number of parameters). The result is parameter dialogs that contain large numbers of parameter values and/or choices. In these cases, further consolidation doesn't make sense (since we do not want to aggregate data together that is actually independent or unrelated) as a way to address the overload.

3.4.2 Solution

In cases where aggregation is not an appropriate remedy the standard annotations for grouping parameters by tab and group can be utilized. Rather than aggregate the data, the result of using the tab and group directives is to organize the data into a "tree" (i.e. the data is presented in a hierarchy where the first layer is determined by the tab and the next layers is determined by group). In particular, common parameters should be organized such that they appear in the default tab and less common parameters are assigned to later tabs. Tab labels are also an important consideration since users should be able to determine quickly, based on the name, whether they need to look in a particular tab.

4 Language Implications

Many of the "normal" patterns found in [2] do not appear in this paper. This is primarily because Modelica does not include concepts like pointers and methods which are fundamental to many of the patterns. Furthermore, it has been observed that many of the traditional patterns in software development essentially boil down to adding an additional level of indirection to an abstraction. Since there are very few ways to express this indirection in Modelica, the number of patterns is fairly limited.

One of the lingering questions from this discussion is to what extent these patterns (or lack of patterns) represent deficiencies in the language. For the patterns and anti-patterns that are related to redundant code (*i.e.* Sections 2.1, 3.1 and 3.2) the language is well equipped to address these issues although there are certainly ways that tools can assist model developers in more effectively utilizing those language features.

Although the Singleton pattern is being used in several libraries it is this author's opinion that the semantics of the language do not mesh as well with the pattern and modeler needs. The use of inner and outer in this way has implications for robust model checking and the dependency on inner elements is not easily recognized or represented. In addition, the "many to many" issue mentioned in Section 2.2.2 requires improved expressiveness in the language.

In the case of the medium model pattern, the inability to express type constraints through physical connections is a serious limitation in the language and one that is recognized in the design group. Hopefully this deficiency will be addressed soon.

Section 2.5.2 discusses how behavior can be described parametrically. However, there are many different ways to "phrase" this kind of behavior and they cannot necessarily be easily recognized by tools. Having language elements for describing parametric relationships could not only bring consistency how such behavior is described but it could also allow tools to automatically deal with variable index issues that currently burden developers (equation differentiation, continuity concerns, finite state machines, *etc*).

5 Conclusion

The goal of this paper is to identify common patterns and anti-patterns to help users identify easy solutions for common problems as well as to prompt discussions within the Modelica design group on ways the language can be enhanced to either institutionalize some of the best practices in these patterns or add language features to eliminate the need for these patterns.

References

1. Alexander, C., Ishikawa, S., and Silverstein, M., "A Pattern Language: Towns, Buildings, Construction", Oxford University Press, ISBN 0-19-501919-9, 1977.
2. Gamma, E., Helm, R., Johnson, R. and Vlissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, ISBN 0-201-63361-2, 1994.

3. Brown, J. W., Malveau, R. C. and Mowbray, T. J., “AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis”, John Wiley and Sons, ISBN 0-471-19713-0, 1998.
4. Laplante, P. A., Neill, C. J., “Antipatterns: Identification, Refactoring, and Management”, CRC Press, ISBN 0-8493-2994-9, 2006
5. Clauss, C., Leitner, T., Schneider, A. and Schwarz, P., “Object-oriented Modelling of Physical Systems with Modelica using Design Patterns”, Fraunhofer Institute, 2000
6. Dominus, M., “Design Patterns of 1972”, http://blog.plover.com/prog/design-patterns.html
7. Johnson, R., “Design patterns and language design”, http://www.cincomsmalltalk.com/userblogs/ralph/blogView?entry=3335803396
8. Tiller, M., Bowles, P. and Dempsey, M., “Development of a Vehicle Modeling Architecture in Modelica”, 3rd International Modelica Conference, 2003.
9. Otter, M., °Arz´en, K.-E., Dressler I., “StateGraph-A Modelica Library for Hierarchical State Machines”, 4th International Modelica Conference, 2005.
10. Newman, C. E., Batteh, J. J., Tiller, M., ”Spark-Ignited Engine Cycle Simulation in Modelica”, 2nd International Modelica Conference, 2002. http://www.modelica.org/events/Conference2002/papers/p17_Newman.pdf
11. Elmqvist H., Tummescheit, H., Otter, M., “Object-Oriented Modeling of Thermo-Fluid Systems”, 3rd International Modelica Conference, 2003.
12. Tiller, M. M., “Introduction to Physical Modeling with Modelica”, Kluwer Academic Publishers, ISBN 0-7923-7367-7, 2001.