METHODS & TOOLS

Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**

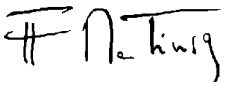
Winter 2006 (Volume 14 - number 4) www.methodsandtools.com

The Real Project Skills

If you look at training material on software project management, you often find what I will call "classical" techniques: how to list every task that should be performed, to create the schedule of the project, to control its progress, to recognise the associated risks, etc... In addition, agile approaches have put in evidence some techniques or tools like short iterations, daily meetings or burndown charts. With this situation, you can have the impression that project management is mainly a technical issue. Don't get me wrong: all these technical considerations are important and they are useful tools for the project manager, but I think that the most essential skills for a successful project are beyond those technical aspects.

We like to say that we are **engineering** software, but **people** are the essential contributors to the success of software development project. We can be initially impressed by the reassuring technical frameworks provided by approaches like UML/UP. Processes to develop software are clearly defined and several models are used to capture requirements in a structured form. This could seem superior to unstructured requirements and easier to use and understand than some "social" approaches. With a little more experience, I think that use case modelling is worth nothing if you are not able to elicit requirements from your customers and understand them. The "social" aspects of a project are often those that decide of its success. Is it better to know how to produce a Gantt chart or how to negotiate successfully? Will you prefer to be good at drawing a burndown chart or to motivate a team?

We should get beyond our technical vision of software project management and take more time to explore and develop the social and psychological dimensions. I hope that the article on stakeholder analysis in this issue will help you to progress in this long journey. As 2006 comes to its end, I wish you the best for all your projects in 2007.



Inside

Using Stakeholder Analysis in Software Project Management

Bas de Baar, webmaster@softwareprojects.org
http://www.SoftwareProjects.org

Every software professional that has been part of more than one project knows for sure: no two projects are the same. Different circumstances make most software projects unique in several aspects. And with different situations come different approaches to handle project life effectively: there are multiple ways to "do" a project. Different circumstances *require* different approaches.

Although a project is to a large part defined by the required end results and technology used, the main determining factor of what makes a project different from another is *people*. The entire process of software project management is strongly stakeholder-driven. It is their wishes, fears, dreams—their *stakes*—that determine the course of the project. You have to handle a project to really grasp the impact of people on your endeavour. You have to "live" a project to know the force of political games and power trips. You have to lead a team to deliver a project under time pressures to appreciate the constructive power of motivated people or the destructive power of demotivated team members.

In a project, it is the *people* that are the main cause of problems. Time schedules, financial projections, and software goals may be abstractions, but it is the flesh-and-blood people whose work determines your project's status. It's the programmer that misses a deadline and holds up everyone else, it's the financial manager that goes berserk if you can't produce some good budgetary indications, and it's the key user that doesn't give a darn but didn't tell you about his dismal lack of motivation; these are the folks who can cause serious trouble.

Stakeholders and the Software Project Manager's Problem

So, as a software project manager, you should really focus on the stakeholders. You should be guided by their fears and their wishes. A *stakeholder* can be a project team member, an employee of the user organization, or a senior manager. It can be virtually anyone, as long as that person has something to do with the project.

Here is the central problem the software manager is faced with, appropriately named "the software project manager's problem," as explained by Barry W. Boehm and Rony Ross [1]. They believe that everyone affected by the project, directly or indirectly, has something to say, again directly or indirectly, and will do so. All of them want to get the best from this project for themselves personally or for their organization. It is the job of the software project manager to see that everyone gets what he or she wants, in one way or another. He has to "make everyone a winner."

Of course, this is easier said than done. You have to act like a psychoanalyst and get in touch with everyone's deeper feelings. Most technical project leaders would be running for the door at this moment. To make it less fuzzy, we can attach some project management lingo to it:

"The project manager can use stakeholder analysis to determine the stakes and expectations of the stakeholders, and adopt the project organization and feedback mechanisms according to the desired outcomes."

Expectations, Interests and Requirements

Stakeholder analysis is a technique to identify and analyze the stakeholders surrounding a project. It provides information on stakeholders and their relationships, interests, and expectations. A proper analysis of the stakeholders will help you to construct a project approach suited to the situation and will allow you to negotiate better with the stakeholders.

What people (and therefore your project stakeholders) really, really, really want is what can be termed their interests or, as I sometimes call them, their stakes (hence the name “stakeholder”). With fears there is a stake to lose, and with wishes there is something to gain.

In this context, I consider interests as the aspects that drive people. Before you start drawing your “interest evaluation diagram” or some other tool or technique, be aware that in general these interests are hardly ever communicated. It is pure mind stuff, all inside the head of the owner. A four-year-old boy may share his true interests with you, but the fifty-year-old greying accountant will tell you nothing.

If no one will tell you anything, what is the point? People will tell you *something* if you ask them. They will tell you they want an ice cream cone, a new hyperspeed Internet uplink, or a new financial software package. In essence, they tell you what they expect. It is a statement created by *themselves* about a desired situation: their *expectations*.

If I emphasize that expectations are a one-sided communication, then there must be something else as well; enter *requirements*. Requirements are a set of statements negotiated among a group of people. They can be the original expectations, if all agree on the statement itself, but more often than not, requirements consist of some consensus of conflicting expectations.

It sounds simple, but getting the expectations is one thing and discovering their corresponding stakes is another. Why bother anyway? What is it worth? A lot. You can't effectively change the stakes, but you can alter the set of requirements as long as the new requirements continue to support the stakes. In this way, there is room to negotiate a set of requirements for the project that poses no conflict, matches the stakes, and thus makes everyone a winner!

Consider this example: a stakeholder formulates an expectation for the software project; for example, senior management states that "The project should be finished before the end of August." The project manager then has to deal with this time frame. When this deadline is no problem, he can rest assured; however, it is a software project, so the deadline typically will be a problem. The way to handle it is to get some information on the stakes that prompted this requirement to be formulated in the first place.

Perhaps it is the old "I don't want to lose face when my projects get delayed" concern. That being the case, the project manager can offer alternatives that don't violate the stake, like keeping the deadline but postponing a subsystem. Chances are good that alternative requirements that keep supporting the stakes will be accepted—maybe not easily, but project managers should do something to earn their money.

So, it seems to be valuable to dig deeper into the souls of your stakeholders. It sounds all very misty and cloudy, but keep remembering why you must do it:

- Expectations are assessable and can be influenced. However, you should stay true to the interests of people; they will determine the amount of leverage you have to change the expectations without setting a stakeholder on the warpath.
- Requirements have to stay in line with what people are expecting. If stakeholders find out the requirements don't fit their expectations, you have a major problem.
- Knowledge about the stakeholders and their expectations and interests helps you shape the project organization (structure, authority, and responsibility).
- It is a very good risk analysis strategy to see where the potential problems will be.

Stakeholder Analysis

The actual steps for a stakeholder analysis are:

- Stakeholder identification
- Stakeholder expectations and interests
- Stakeholder influence and role in the project

Stakeholder Identification

This first step is concerned with the question "Who are the stakeholders?" For this, you basically draw maps of people or groups and their relationships. You start with two names on a whiteboard and before you know it, you are drawing on the walls.

Stakeholder Expectations and Interests

The more difficult step is this one; here we get the socio-psycho stuff. For expectations it is fairly straightforward: just ask. You can ask in person or via mail or email. Create some variations on the question, so that it is not too obvious what you are trying to find out.

Stakeholder interests are another thing. Trying to elicit their interests is always guesswork, deducting them from other information. For this, there are two types of approaches: (1) using a checklist to assist your thinking about the stakeholder and (2) plotting people in small models that help determine the way to approach them.

For the first type, consider a list with questions like: “Is he satisfied with his current job?”, “Is he covering up his own incompetence?” and “Does he want a bigger office?” Thinking about these questions help you build an image of the person’s interests. Consider the list at the end of this article as good starting point.

Using small models, the second type, can be easier than it sounds. For this, you have to plot a stakeholder in a dimension, and by doing that, you get an idea of how to approach the stakeholder. Dimensions can include: “How much in favour of the project”, “Process or Content-oriented” and “Group or Individual-oriented.”

Stakeholder Influence and Role in the Project

The insights about the stakeholders will assist us also to construct the *project organization*:

- Do we have to include the stakeholders in the organization?
- If so, is it wise to grant stakeholders great influence or should we give them positions where they can do no harm?
- How can we construct stakeholders' job descriptions in such a way that they are as motivated as possible?

What Does This Bring The Project Manager?

In the end, what does all this analyzing and guessing bring to us? That is a good question, and here is the answer:

1. A list of all the stakeholders
2. An idea about each stakeholder's relative importance and influence
3. Insight into what stakeholders want out of the project
4. Insight into what makes stakeholders tick
5. An idea about whether stakeholders will work against or for the project

Apart from an improved ability to negotiate better requirements-sets, this information provides the basis for two major project management tools: the project organization (mentioned as the last step of stakeholder analysis) and the feedback mechanisms.

Feedback mechanisms

If stakeholders provide requirements to a project, they are very interested to know what happens to them. Are they going to be met, or will they be ignored? Keeping your stakeholders in the loop, reassuring your stakeholders about what is happening with their stakes/requirements is a smart thing to do.

There are many project management techniques and artifacts available just for the purpose of feedback. Often, it is unclear from the methods that you use what is the purpose of the feedback. In the list below are some artefact samples that are common in most methods, and what kind of feedback each provides.

Requirements definition	Feedback to the users how their requirements are noted after talking, analyzing and negotiation
Functional design	Feedback to the user how their requirements are translated to a new system
Prototype	Feedback to the user how their requirements are translated to a new system
Schedule	Feedback on constraint "time"
Budget	Feedback on constraint "cost"

Closing

Every improvement you make with the goal of improving your ability to do projects better should involve how to handle the human element more effectively, as this is the area where we can realize the biggest gains.

Stakeholder analysis is one technique that can assist in this. To sharpen your knife you can improve your sensitivity in this area, create your own checklists and read up on the various models available to you. Finally, just being aware of stakes and their effects on your project in itself is a huge benefit to your software project management efforts.

Checklist Stakeholder Interests

Personal	
Work Orientation versus Family Orientation	*Is the person more focused on the work environment or the home front?*
Satisfaction with Current Job	*Important to know when creating a job description. If not satisfied, perhaps include some new, more exciting content to the job.*
Satisfaction with Current Organization	*A broader aspect than "satisfaction with job": do you need to be aware of some resentment a person might have against the organization?*
Desire to Gain More Skill/Knowledge in a Certain Area	*Giving someone the opportunity to improve competency in a certain area is a great motivator.*
Sufficient Appreciation	*Showing some appreciation can boost a person's performance.*
Reduction/Expansion of Workload	*Does someone want to do more or less? As a project manager, you don't always have the authority to influence this, but you might give it a shot.*
Reduction/Expansion of Responsibility	*Actually, comments are the same as above.*
Infection by Not-Invented-Here Syndrome	*Does someone have a great need to be involved in something to accept it? If yes, and you need the person, then ensure involvement.*
Relatedness (Interpersonal/Social)	
Recognition of Knowledge Among Peers Outside the Organization	*If someone has a strong need for recognition among his or her own peers, try to incorporate it into the job.*
Recognition of Job Competence within the Organization (Hierarchy)	*Does the person feel his or her work goes unnoticed in the organization? Does the person want to keep a higher profile?*
Covering Up Own Incompetence (Don't Rock the Boat)	*Some people try to maintain a low profile and avoid attracting any attention to themselves to cover up some incompetence.*
Boosting Another's Reputation (Sponsorship)	*It is nice to get some insight into who is friends with whom and who works to give others a good reputation. Sponsors can work nicely together, but be sceptical about the judgements made about each other.*

Undermining Another's Reputation	*Actually, the opposite of the above situation: who is trashing whom? Try to avoid putting these antagonists together.*
Attempt to Move to Another Job within the Organization	*Will not do much about own job or assignment, but can be found meddling in the desired area where he or she has no authority.*
Attempt to Build an Empire within the Organization	*Very dynamic personality and worth soliciting for maximum influence in the project, if only for sheer number of personal connections.*
Attempt to Maintain an Empire within the Organization	*Looks the same as person mentioned above, but has a more defensive attitude and will try to avoid as much change as possible.*
Attempt to Increase Sphere of Influence within Organization	*This person makes assignments bigger than they actually are.*
Existence (Material Interests)	
Desire for More Money	*Need I say more?*
Desire for More Tools	*Any questions?*
Desire for a Bigger Office	*Look for a real-estate agent.*

References

[1] Boehm, Barry W. and Rony Ross. "Theory-W Software Project Management Principles and Examples." *IEEE Transactions on Software Engineering.* 15.7 (1989): 902-916.

Software Product Line Engineering with Feature Models

Danilo Beuche, Danilo.Beuche@pure-systems.com,
Mark Dalgarno, mark@software-acumen.com,
pure-systems http://www.pure-systems.com/
Software Acumen, http://www.software-acumen.com/

Although the term *"Software Product Line Engineering"* is becoming more widely known, there is still uncertainty among developers about how it would apply in their own development context. In this article we tackle this problem by describing the design and automated derivation of the product variants of a Software Product Line using an easy to understand, practical example.

One increasing trend in software development is the need to develop multiple, similar software products instead of just a single individual product. There are several reasons for this. Products that are being developed for the international market must be adapted for different legal or cultural environments, as well as for different languages, and so must provide adapted user interfaces. Because of cost and time constraints it is not possible for software developers to develop a new product from scratch for each new customer, and so software reuse must be increased. These types of problems typically occur in portal or embedded applications, e.g. vehicle control applications [Ste04]. *Software Product Line Engineering* (SPLE) offers a solution to these not quite new, but increasingly challenging, problems [Cle01].

The basis of SPLE is the explicit modelling of what is **common** and what **differs** between product variants. Feature Models [Kan90], [Cza00] are frequently used for this. SPLE also includes the design and management of a variable software architecture and its constituent (software) components.

This article describes how this is done in practice, using the example of a Product Line of meteorological data systems. Using this example we will show how a Product Line is designed, and how product variants can be derived automatically.

Software Product Lines

However, before we introduce the example, we will take a small detour into the basis of SPLE. The main difference from "normal", one-of-a-kind software development is a logical separation between the development of core, reusable software assets (the *platform*), and actual applications. During application development, platform software is selected and configured to meet the specific needs of the application.

The Product Line's commonalities and variabilities are described in the *Problem Space*. This reflects the desired range of applications ("product variants") in the Product Line (the "domain") and their inter-dependencies. So, when producing a product variant, the application developer uses the problem space definition to describe the desired combination of problem variabilities to implement the product variant.

An associated *Solution Space* describes the constituent assets of the Product Line (the "platform") and its relation to the problem space, i.e. rules for how elements of the platform are selected when certain values in the problem space are selected as part of a product variant. The four-part division resulting from the combination of the problem space and solution space with domain and application engineering is shown in Figure 1.

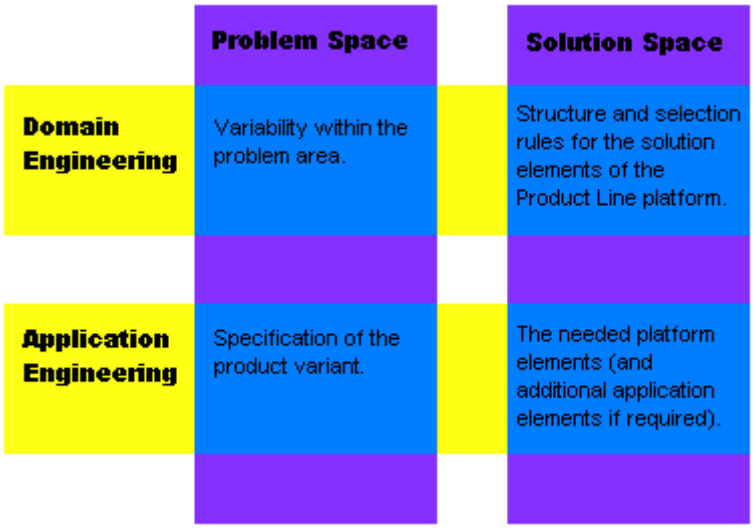


Figure 1 Overview of SPLE activities

Several different options are available for modelling the information in these four quadrants. The problem space can be described e.g. with *Feature Models*, or with a *Domain Specific Language* (DSL). There are also a number of different options for modelling the solution space, for example component libraries, DSL compilers, generative programs and also configuration files. [Cza00].

In the rest of this article we will consider each of these quadrants in turn, beginning with Domain Engineering activities. We will first look at modelling the problem space - what is common to, and what differs between, the different product variants. Then we will consider one possible approach for realising product variants in the solution space using C++ as an example. Finally, we will look at how Application Engineering is performed by using the problem and solution space models to create a product variant. In reality, this linear flow is rarely found in practice. Product Lines usually evolve continuously, even after the first product variants have been defined and delivered to customers.

Our example Product Line will contain different products for entry and display of meteorological data on a PC. An initial brainstorming session has led to a set of possible differences (*variation points*) between possible products: meteorological data can come from different sensors attached to the PC, fetched from appropriate Internet services or generated directly by the product for demonstration and test purposes. Data can be output directly from the application, distributed as HTML or XML through an integrated Web server or regularly written to file on a fixed disk. The measurements to make can also vary: temperature, air pressure, wind velocity and humidity could all be of interest. Finally the units of measure could also vary (degrees Celsius vs. Fahrenheit, hPa vs. mmHg, m / s vs. Beaufort).

Modelling the Problem Space

We will now convert the informal, natural-language specification of variability noted above into a formal model, in order to be able to process it. Specifically, we will use a *Feature Model*. Feature models are simple, hierarchical models that capture the commonality and variability of a Product Line. Each relevant characteristic of the problem space becomes a *feature* in the model. A definition of the term "feature" is given in Definition 1.

Features are an abstract concept for describing commonalities and variabilities. What this means precisely needs to be decided for each Product Line.

A feature in this sense is a characteristic of a system relevant for some Stakeholder. Depending on the interest of the Stakeholders a feature can be for the example a requirement, a technical function or function group or a non-functional (quality) characteristic.

Definition 1 Features

Feature models have a tree structure, with features forming nodes of the tree. The arcs and groupings of features represent feature variability. There are four different types of feature groups: *“mandatory"*, *“optional"*, *"alternative"* and *“or”*. When specifying which features are to be included in a variant the following rules apply: If a parent feature is contained in a variant,

- all its *mandatory* child features must be also contained ("n from n"),
- any number of *optional* features can be included ("m from n, 0 < = m<=n"),
- exactly one feature must be selected from a group of *alternative* features ("1 from n"),
- at least one feature must be selected from a group of *or* features ("m from n, m>1").

Unfortunately, no single standard has yet been agreed for the graphical notation of feature models. However, in the literature, the graphical notation of the original FODA method [Ste04] is common.

However this is representable with standard text tools and graph libraries only with difficulty. Therefore in this article a simplified notation has been used. *Alternatives* and groups of *or* features are represented with traverses between the matching features. In this representation both colour and box connector are used independently to indicate the type of group. Our notation is shown in Figure 2.

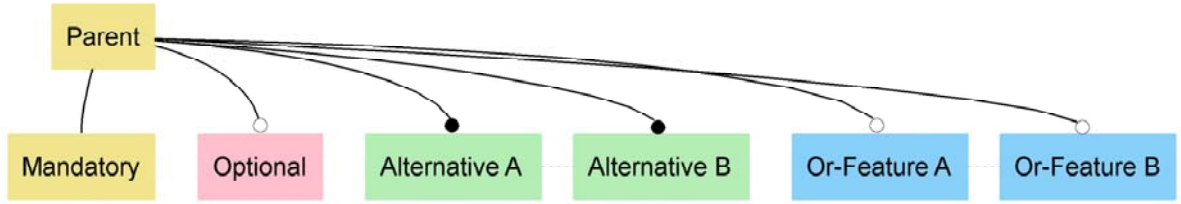


Figure 2 Structure and notation of feature models

Using this notation, our example feature model, with some modifications, is shown in Figure 3:

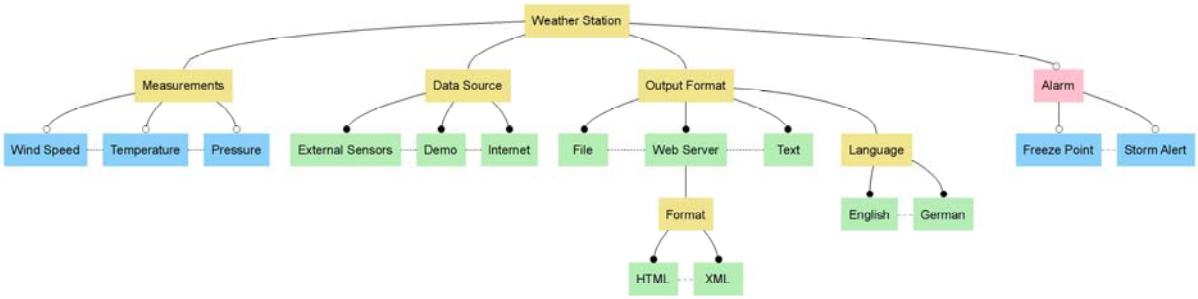


Figure 3 Feature model for our meteorological Product Line

Each Feature Model has a root feature. Beneath this are three *mandatory* features – "Measurements", "Data Source" and "Output Format". Mandatory features will always be included in a product variant if their parent feature is included in the product variant. Mandatory features are not variable in the true sense, but serve to structure or document their parent feature in some way. Our example also has *alternative* features, e.g. "External Sensors", "Demo" and "Internet" for data sources. All product variants must contain one and only one of these alternatives.

At this stage we can already see one advantage that feature modelling has over a natural-language representation - it removes ambiguities - e.g. whether an individual variant is able to process data from more than one source. When taking measurements any combination of measurements is meaningful and at least one measurement source is necessary for a sensible weather station, to model this we use a group of *Or*. Usually simple *optional* features are used, such as the example of the freezing point alarm. Refining the model hierarchy can also make further improvements. So the strict choice between Web Server output formats - HTML or XML – can be made explicit.

Feature models also support transverse relationships, such as *requires* and *mutually exclusive*, in order to model additional dependencies between features other than those already described. So, in the example model, a selection of the “Freeze Point” alarm feature is only meaningful in connection with the temperature measurement capability.

This can be modelled by a *"Freeze Point" requires "Temperature"* relationship (not shown in the figure). However, such relations should be used sparingly. The more transverse relations there are, the harder it is for a human user to visualize connections in the model.

When creating a feature model it can be difficult to decide exactly how problem space variabilities are to be represented in the model. In this case it is best to discuss this further with the customer. It is usually better to base these discussions around the feature model, since such models are easier for the customer to understand than textual documents and / or UML models. Formalising customer requirements in this way offers significant advantages later in Product Line development, since many architectural and implementation decisions can be made on the basis of the variabilities captured in the feature model. In the example, the use of the output format XML and HTML can be clarified. The model explicitly defines that the choice of output format is only relevant for Web Server, a format selection is not possible for File or Text output. However, in the context of a discussion of the feature model it could be decided that HTML is also desirable for the on-screen (Window) representation and could also be applicable for file storage.

This results in the modified feature model shown in Figure 4.

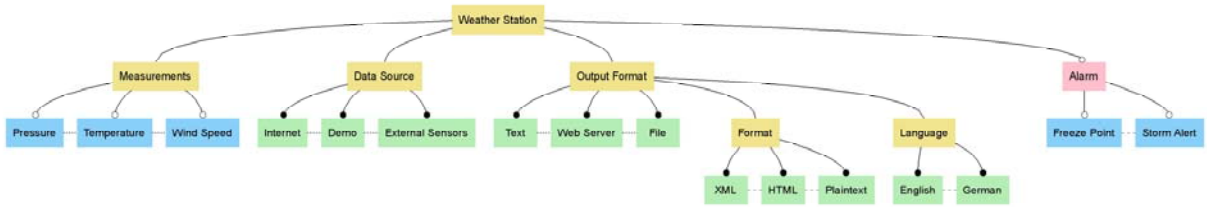


Figure 4 Enhanced Feature Model for our meteorological Product Line

We have added “Plaintext” to the existing features; this was implicitly assumed for output to the screen or to a file. We have modelled the mutual exclusion of XML and screen display (“Text”) using a (transverse) relationship between these features (not shown).

The previous discussion describes the basic feature model approach commonly found in the literature, but a number of people have extended this basic approach. To complement the so-called *hard relations* between features (*requires* and *conflicts)* the weakened forms *recommends* and *discourages* have been added to many feature model dialects. A few tools also support the association of named attributes with features. This allows numeric values or enumerated values to be conveniently associated with features e.g. the wind force required to activate the storm alarm could be represented as a "Threshold" attribute of the feature "Storm Alert".

An important and difficult issue in the creation of feature models is deciding which problem space features to represent. In the example model, it is not possible to make a choice from the available hardware sensor types (e.g. use of a PR1003 or a PR2005 sensor for pressure). So, when specifying a variant, the user does not have direct influence on the selection of sensor types. These are determined when modelling the solution space. If the choice of different sensor types for measuring pressure is a major criterion for the customer / users, then appropriate options would have to be included in the feature model. This means that the features in the problem space are not a *1:1*-illustration of the possibilities in the solution space, but only represent the (variable) characteristics relevant for the users of the Product Line. Feature models are a user-oriented (or marketing-oriented) representation of the problem space, not the solution space.

After creating the problem space model we can use it to perform some initial analysis. For example, we can now calculate the upper limit on the number of possible variants in our example Product Line. In this case we have 1,512 variants (the model in Figure 2 only has 612 variants). For such a small number of variants the listing of all possible variants can be meaningful. However, the number of variants is usually too high to make practical use of such an enumeration.

Modelling the Solution Space

In order to implement the solution space using a suitable variable architecture, we must take account of other factors beyond the variability model of the problem space. These include common characteristics of all variants of the problem space that are not modelled in the feature model, as well as other constraints that limit the solution space.

These typically include the programming languages that can be used, the development environment and the application deployment environment(s). Different factors affect the choice of mechanisms to be used for converting from variation points in the solution space. These include the available development tools, the required performance and the available (computing) resources, as well as time and money. For example, use of configuration files can reduce development time for a project, if users can administer their own configurations. In other cases, using preprocessor directives (***#ifdef***) for conditional compilation can be appropriate, e.g. if smaller program sizes are required.

There are many possibilities for implementation of the solution space. Very simple variant-specific model transformations can be made with model-driven software development (MDSD) tools by including information from feature models in the Model-Transformation process. [Voel05] gives an example using the *OpenArchitectureWare* model transformer. Aspect-oriented programming (AOP) can also be used as a means for the efficient conversion of variabilities in the solution space. Product Lines can also be implemented naturally using "classical" means such as procedural or object-oriented languages.

Designing a variable architecture

A Product Line architecture will only rarely result directly from the structure of the problem space model. The solution space which can be implemented should support the variability of the problem space, but will not necessarily be a *1:1* correspondence with the architecture. The mapping of variabilities can take place in various ways.

In the example Product Line we will use a simple object-oriented design concept implemented in C++. A majority of the variability is then resolved at compile-time or link-time; runtime variability is only used if it is absolutely necessary. Such solutions are frequently used in practice, particularly in embedded systems.

The choice of which tools to use for automating the configuration and / or production of a variant plays a substantial role in the design and implementation of the solution space. The range of variability, the complexity of relations between problem space features and solution constituents, the number and frequency of variant production, the size and experience of the development team and many further factors play a role. In simple cases, the variant can be produced by hand, but automated tools in the form of Excel and / or small configuration scripts, and also model transformers, code generators or variant management systems will speed production.

One approach for modelling and mapping of the solution space variability is to use a separate *Solution Model* to model the solution space, to associate solution space elements with problem space features, and to support the automatic selection of solution space elements when constructing a product variant. This separation of concerns also has the advantage of allowing both models to evolve independently.

Solution models have a hierarchical structure, consisting of logical items of the solution architecture, e.g. components, classes and objects. These logical items can be augmented with information about "real" solution elements such as source code files, in order to enable automatic production of a solution from a valid feature model configuration (more on this later). For each solution model element, a rule is created to link it to the solution space. For example, the Web Server implementation component is only included if the Web Server feature has been selected from the problem space. To achieve this, a ***hasFeature('Web Server')*** rule is attached to the "Web Server" component. Any item below “Web Server” in the solution model can only be included in the solution if the corresponding **Web Server** feature is selected.

In our example, an architectural variation point arises, among other possibilities, in the area of data output. Each output format can be implemented with an object of a format-specific output class. Thus in the case of HTML output, an object of type ***HtmlOutput*** is instantiated, and with XML output, an ***XmlOutput*** object. There would also be the possibility here of instantiating an appropriate object at runtime using a *Strategy* pattern. However, since the feature model designates only the use of alternative output formats, the variability can be resolved at compile-time and a suitable object can be instantiated using code generation for example.

In our example solution space a lookup in a text database is used to support multiple natural languages. The choice of which database to use is made at compile-time depending on the desired language. No difference in solution architectures can be detected between two variants that differ only in the target language. Here the variation point is embedded in the data level of the implementation.

In many cases, managing variable solutions only at the architectural level is insufficient. As it has already been mentioned above, we must also support variation points at the implementation level, i.e. in our case at the C++ source code level. This is necessary to support automated product derivation. The constituents of a solution on the implementation level, like source code files or configuration files which can be generated, can also be entered in the solution model and associated with selection rules.

So the existence of the Web Server component in a product variant is denoted using a ***#define*** preprocessor directive in a configuration *Header* file. In addition, an appropriate abstract variation point variable "WEB SERVER" must first be created in the solution model. The value of this variable is determined by a ***Value*** attribute. In our case, this value is always *1* if the variable is contained in the product variant. An item can now be assigned to this abstract variable. This item also possesses attributes (***file, flag***), which are used during the transformation of the model into "real" code. The meaning of the attributes is determined by a transformation selected in the generation step.

Separating the logical variation point from the solution makes it very simple to manage changes to the solution space. For example, if the same variation point requires an entry in a *Makefile*, this could be achieved with the definition of a further source element, below the variation point "WEB SERVER".

Deriving product variants

The solution model captures both the structure of the solution space with its variation points and the connection of solution and problem space. Not only is the separation of these two spaces important, but also the direction of the connection, since problem space models in most cases are much more stable than solution spaces; the linkage of the solution space to the problem space is more meaningful than the selection of solution items by rules in the problem space. This also increases the potential for reuse, since problem space models can simply be combined with other (new, better, faster) solutions.

Now we have all the information needed to create an individual product variant. The first step is to determine a valid selection of characteristics from the feature model. In the case of some tools, the user is guided towards a valid and complete feature selection, and for a large feature model this can reduce the time to create a complete and consistent selection by an order of magnitude. Once a valid selection is found, the specified feature list, and the solution model serve as input for the production of a variant model. Then, as is described above, the rules of the individual model items are checked. Only items that have their rules satisfied are included in the finished solution.

Open Issues in SPLE

So far we have highlighted some of the most common issues that will be encountered when working with Software Product Lines. In this section we highlight some additional issues.

Even with visualization support from specialist tools the visual representation of very complex model structures is not a completely solved problem. Larger feature models can have several hundred features and the solution space can have several thousands or more constituents. Thus it can be hard to understand the implications of modifications to these models just through use of model diagrams.

Issues around Product Line evolution are also very important. Evolution must be managed since changes that positively affect one or more variants could have a negative effect on other variants and these issues may only show up when variants are produced long after the changes have been implemented.

Finally, testing a Product Line also represents a significant challenge. Most Product Lines offer more potential variability than is in use at any one time, and testing all possible variants is usually impossible and in some cases a waste of time. Testing just those variants that are produced is already a difficult problem where there is a high number of variants. However, it is still necessary to co-ordinate testing with variant production in some way. One approach is to create test asset variants as one does for the (software) solution space variants – effectively creating a parallel test solution space that is driven from the Feature model. Reduction of test effort is still an open issue though for many Product Lines.

Closing Remarks

The example presented in this article shows how the variability of the problem space of a Product Line can be described very simply using feature models. The automated production of solution variants is the logical next step, for which we have shown one example. In this example we have highlighted some of the approaches and issues that need to be considered when using Software Product Lines. These are covered in more depth in [Bos00], a standard work on Software Product Lines, for example.

References and Links

[Bos00] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*, Addison-Wesley, 2000

[Cle01] P. Clements, L Northrop, *Software Product Lines: Practices & Patterns*, Addison-Wesley 2001 (see also ***www.sei.cmu.edu/productlines/framework.html***

[Cza00] K. Czarnecki, U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000

[Kan90] K. Kang, et al., *Feature Oriented Domain Analysis (FODA) Feasibility Study*, Technical report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, 1990

[Ste04] M. Steger et al., *Introducing PLA RK Bosch Gasoline of System: Experiences and Practices*, in: Proc. of the Software Product Line Conf. 2004, S. 34-50

[Voel05] M. Voelter, *Variantenmanagement in the context of MDSD*, in: JavaSpektrum 5/05

Creating a Domain-Specific Modeling Language for an Existing Framework

Juha-Pekka Tolvanen, jpt@metacase.com
MetaCase, http://www.metacase.com

Domain-specific languages are a natural extension to code libraries and frameworks, making their use faster, easier and more consistent. This article describes how to define modeling languages on top of a library or a framework.

Developers usually agree that it does not make sense to write all code character by character. After coding some features they usually start to find similarities in the code and patterns that seem to repeat. For most developers it would then make sense to focus just on the unique functionality, the differences between the various features and products, rather than wasting time and effort re-implementing similar functionality again and again. Avoiding reinventing the wheel is good advice for a single developer, but even more so if colleagues too are implementing almost identical code.

One well-known practice is to move the common parts into a reusable library, or make them into a framework that is used as a basis for developing features or even whole products. Application developers can then get to know the framework and its architecture, find out its class hierarchies, learn its APIs and study the programming model required. The use of a framework is supported by documentation, application examples and guidelines. The bad news is that both research and practice show that for most developers it is often hard to use the framework and find the right component for the right task. There are usually also interdependencies that need to be understood, optional services and multiple alternative ways to provide the same functionality. It takes time to learn these and become enough of an expert on the framework to use it well.

Domain-Specific Modeling (DSM) provides a solution: it allows the framework developer to hide the details of the framework by raising the level of abstraction on which applications are built. The basic concepts of the framework are represented as the available kinds of objects in a new, domain-specific modeling language. Application developers can model the applications using these high-level concepts and generate working code that takes best advantage of the framework.

This automation is possible because both the modeling language and generators need only fit the requirements of the specific framework and application type. Automation minimizes routine work, makes many typical errors impossible and leads to a fundamentally faster development process. A domain-specific language can also cover the rules of the architecture and component use, guaranteeing that the framework is used correctly. Developers are thus guided to follow the best practices – those the experienced framework developer knows work well.

DSM example

Before we look at how to create such a language, let’s first illustrate domain-specific modeling with an example. For this purpose we use a well known domain: a digital wristwatch. Suppose your developers make the watch applications, such as a stopwatch or time setting and display. These applications are implemented on a top of a framework that provides common services like time calculations, showing an icon, ringing an alarm etc. The framework expects that a particular programming model is applied when making the application and calling the framework services.

Before any new features can be implemented, or existing ones modified, developers must design how they should work in the watch problem domain. This involves applying the terms and rules of the watch, such as buttons, alarms, display icons, states and user's actions. DSM applies these very same concepts directly in the modeling language.

An example of such a modeling language is illustrated in Figure 1. This modeling language is used to design and create various time related applications. The sample design model represents the time setting feature: the actions a user can make by pressing buttons, the display elements blinking, and the actions changing the time.

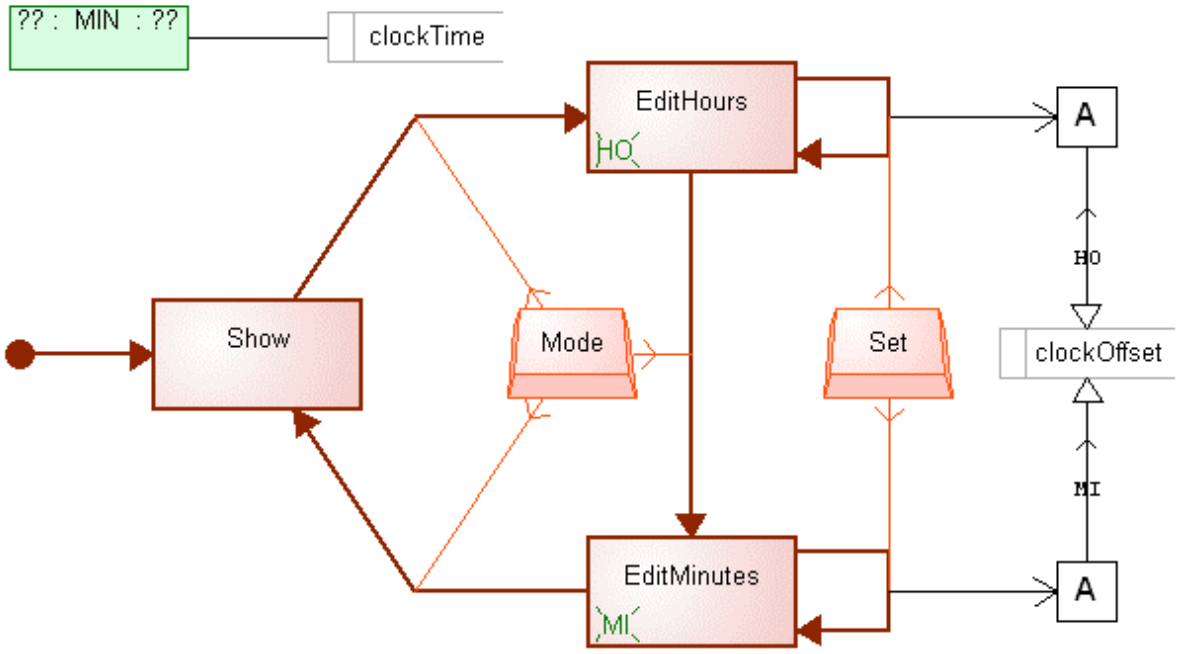


Figure 1. Modeling a time setting feature

Although the modeling language raises the level of abstraction and totally hides the details of the framework services from the developer, it is still clearly based on the underlying watch architecture and its framework. The design models describe the missing information that fills the variation points in the framework to build applications. With this language, developers can focus purely on the design of the application itself. Complete and working application code – using the services of the framework – is generated automatically.

Having a new modeling language, rather than just a model, allows us to make many different watch applications. Figure 2 illustrates a model of a world time application for the same watch framework, built using this same modeling language. This application is for a slightly more advanced watch, with an extra Mode button that allows the user to exit this application, or switch between editing the minutes or hours. The green display function at the top calculates the time to display: no longer just the clockTime, which is set in the Time application, but the clockTime plus a time zone offset which is set here.

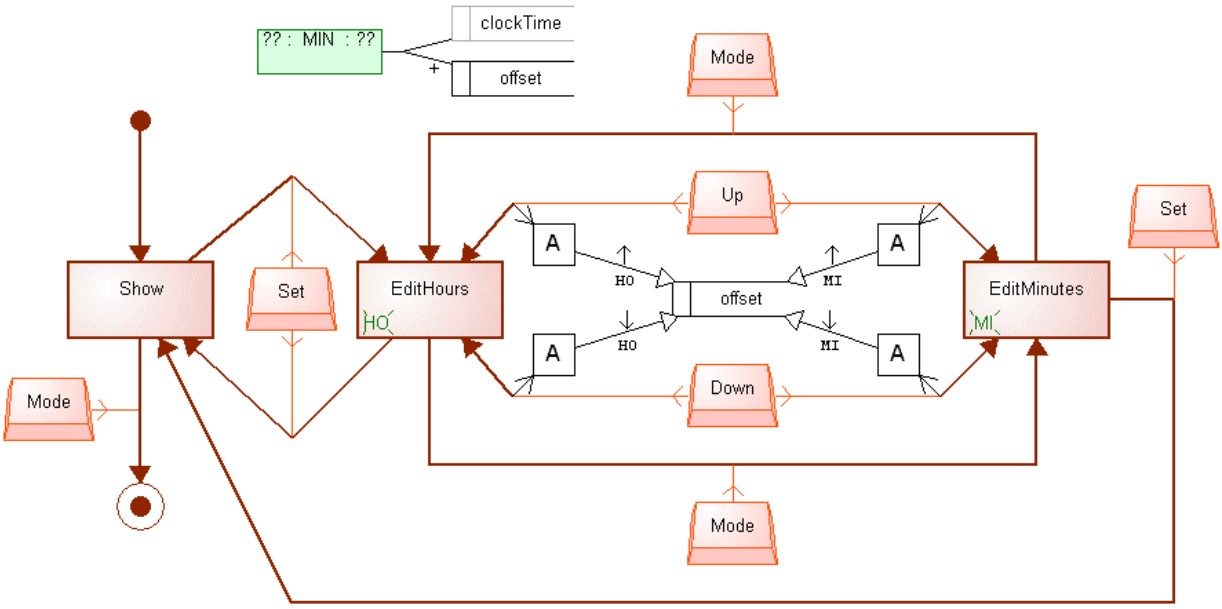


Figure 2. World time application

Figure 3 specifies a basic stopwatch application. The model uses a couple of features of the modeling language which were not used in Figures 1 and 2: straight arithmetic calculations on time variables, and different display functions for different states. The display function for the Running state is the lower green object, and calculates the time to display as the current system time minus the time the stopwatch was started.

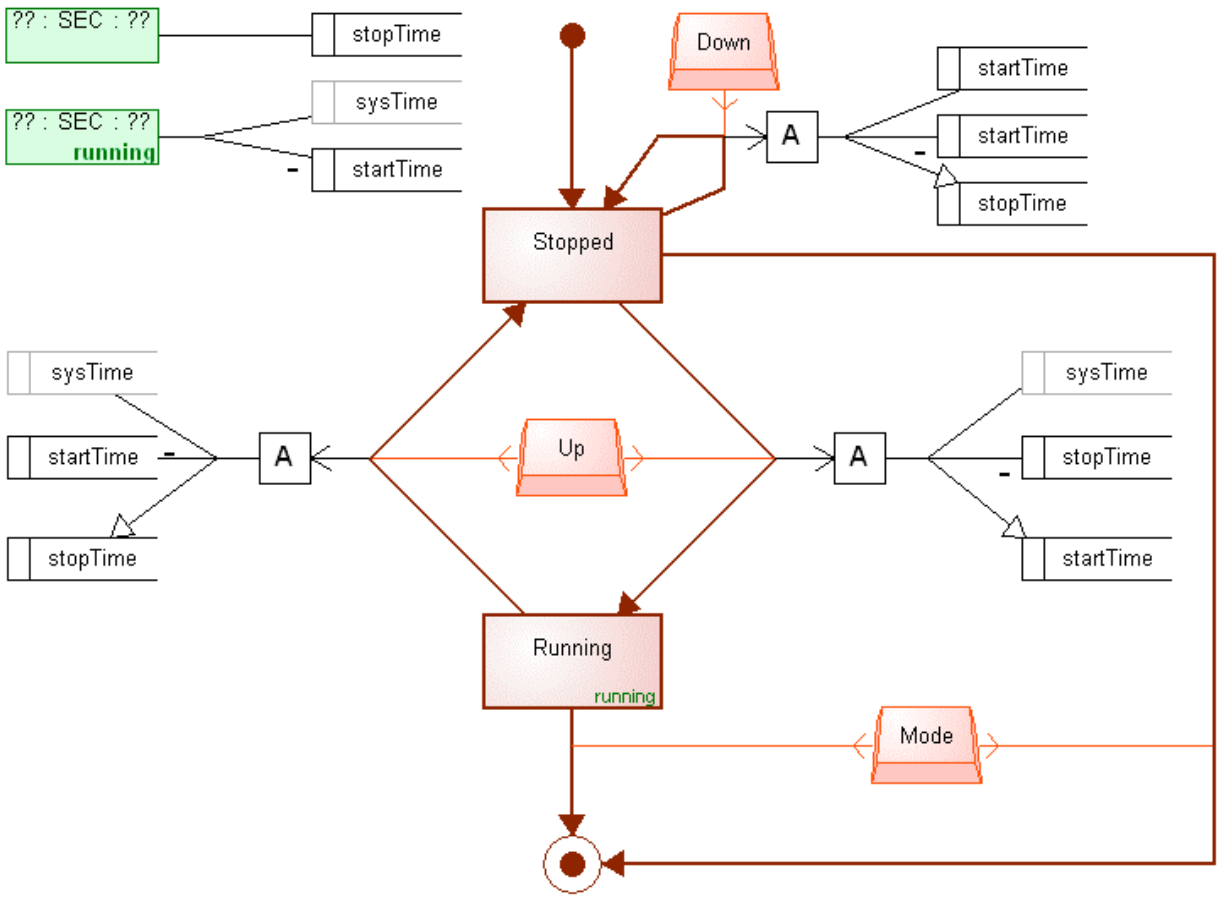


Figure 3. Design of a simple stopwatch application

These applications are not static: they can be developed further just as with source code. If we want an improvement, we simply edit the model. Consider for example Figure 4, which extends the stopwatch application to add lap time functionality, and turns an icon on and off to show when the stopwatch is running.

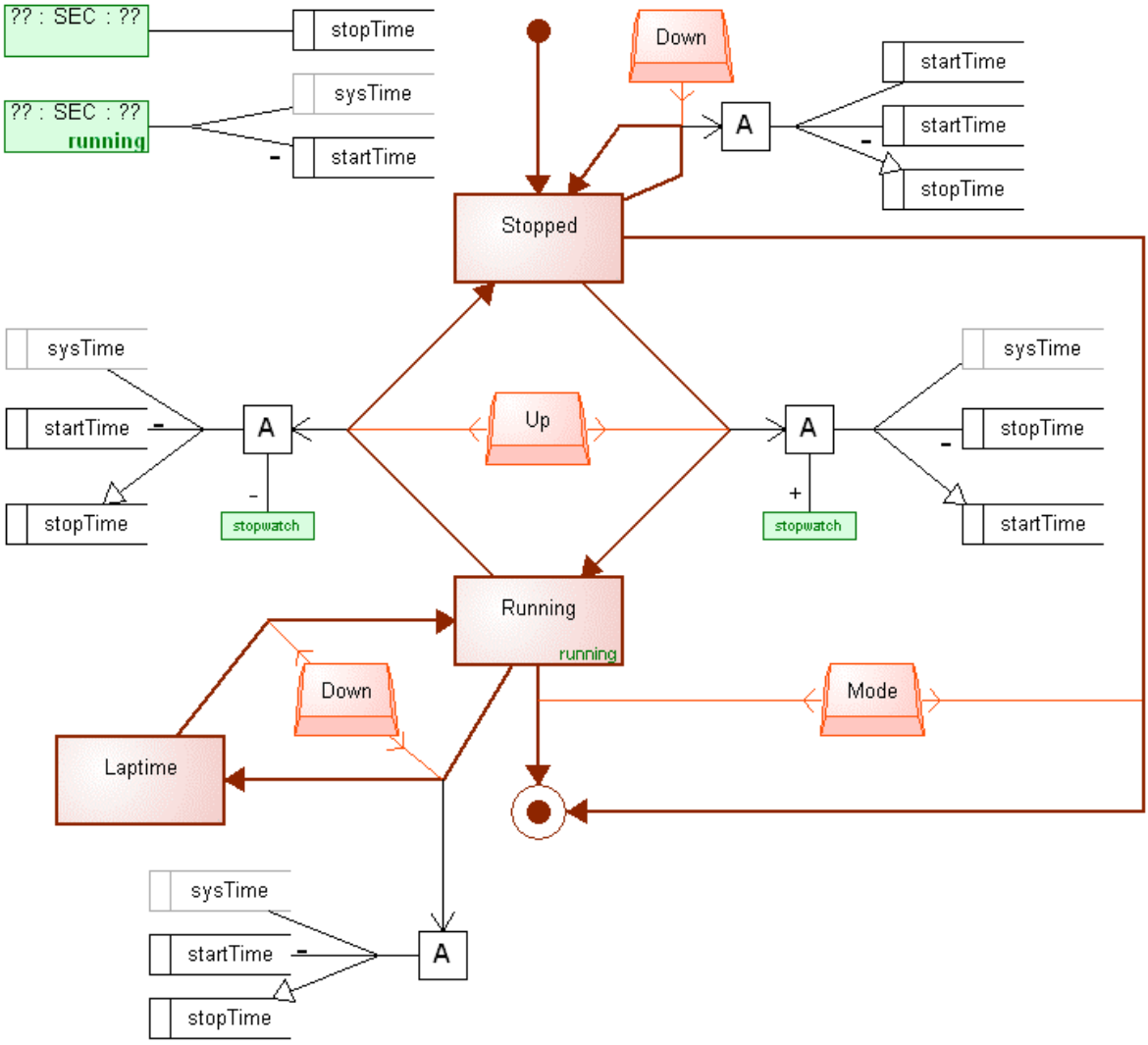


Figure 4. Design of an advanced stopwatch application

Let's look next how the modeling languages are defined. We'll use the watch framework and its related modeling language as an example later in the article.

Steps for defining a DSM language

An incremental approach is often the best way to build a domain-specific language: first define a small part of the language, then model with it, define generators, make changes to the language, model some more etc. After getting it working for the main parts of the framework, we extend the language to cover other parts and finally the whole framework. This is also a natural approach for cases where the framework and supporting components are not yet completely available.

We can divide the process for defining modeling languages and their generators into four phases:

1. Identifying abstractions and how they work together
2. Specifying the language concepts and their rules (metamodel)
3. Creating the visual representation of the language (notation)
4. Defining the generators for model checking, code, documentation, etc.

Usually the process proceeds in this order, iterating back as necessary. We focus here on the second step: specifying the modeling language. The first step, abstractions, is largely already set by the component framework or the library. The framework thus nicely bounds the task of creating the modeling language: we need only support what the developers of the framework have chosen to support. The task remaining for the language designer is thus to make the framework and its services easily available for application engineers.

When defining the modeling language we should not, however, think of models as directly visualizing code or framework API calls. Instead, we should seek higher-level abstractions than current programming languages offer: otherwise we will just be reinventing the flowcharts used to document C programs! In addition to the framework itself, sample applications built on it help us to see how the framework is expected to be used. The sample applications act also as good input and test cases when defining code generators.

Raising the level of abstraction with a language not only improves productivity but also enforces correct use of the framework. This is what every architect hopes for. Putting these rules and guidelines right in the language specification makes it impossible to specify applications or features that are not based on the framework. It also saves developers from having to refer constantly to in-house “design guidelines” documents – which are probably out of date anyway.

Defining the modeling language

Language definition deals with specifying the modeling concepts and possible connections between them. A common starting point is to use existing ways of modeling that are used to talk about that kind of application, as opposed to its implementation. It is particularly important to avoid modeling languages like UML class diagrams, which are clearly implementation-focused and not at all specific to your problem domain.

In the simplest cases, when all possible functionality is already implemented by the framework, the variation can be specified as a list or tree of parameters. In our digital wristwatch example this approach could be used to capture the number of buttons and icons. Usually it is not this simple, since there are also interdependencies among the parameters: these call for at least wizards or decision tables to specify the possible variation.

Neither of these, however, is adequate as most variation is more complex than choosing among alternatives. For example rather than just choosing if the watch has an icon or not we need to know also the circumstances when the icons are used. Also, this approach can only work if all code is already written: it is no help in the most typical situations where we do not yet have the application code.

Parameter lists or feature selections are not adequate when variation goes beyond static configuration to deal with dynamic or behavioral code. These cases require models more complicated than a simple list or tree, whose every element is known in advance. The modelers must be able to create new elements and link them into new structures. We saw an example of such a modeling language in Figures 1–4. Whereas a list or tree with 5 Boolean choices will allow at most 32 variants, with such a language there is no limit to the number of variants and indeed new applications that can be built.

Using known models of computation

Since the watch applications are state based and event-driven we could take state machines as the basis for our language definition. This model of computation can then be enriched with the domain concepts and rules.

Figure 5 illustrates the basic definition (metamodel) of a state machine. The metamodel describes that each state machine can have one Start object initiating the state machine with a transition leading to a state. The State object has two properties: its name that is unique inside the model and a description text. A State can be connected with a Transition to another State or to a Stop object (both grouped together here into one “Objects in binding” group, since they are both bound in the same role in this Transition).

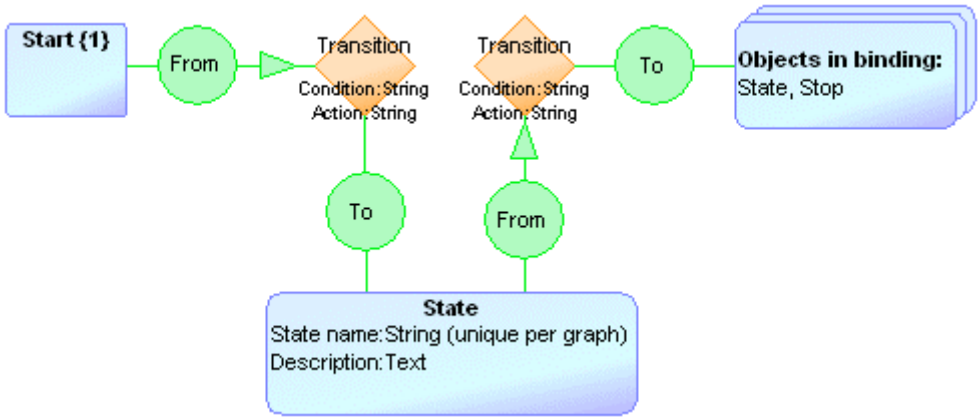


Figure 5. Basic metamodel of a state machine

A language also has rules that constrain how models can be made and modified, to ensure model correctness. These rules are specified in the metamodel along with the language concepts. For example in state machines, as specified also in the metamodel in Figure 5, a Start state can have no incoming transitions, and in each state machine there can only be one start state.

Extended languages with domain-specific concepts and rules

We can now enrich our modeling language with watch-specific concepts and rules. This means mapping the domain concepts and framework services to modeling elements. Some of them can be seen as the objects in the modeling language while others could be better captured as object properties, relationships, sub-models or links to models in other languages.

For example, in a watch each application state can have a display function that calculates what time to show: the current time for a Time application, the time plus a time zone offset for a World Time application etc. Another watch characteristic is that rather than a general concept of event, transitions in a watch are caused by pressing buttons.

An explicit concept of Button is thus added to the modeling language specification. Figure 6 shows an extended metamodel where State has two additional properties. A Display that is shown when a state is active and a Time unit (hour, min, sec, hundredth) to indicate if certain part of the time is blinking – e.g. when that part of the time is being edited.

Transitions are modified to describe watch applications more precisely: users pressing buttons of the watch trigger transitions and during a transition an action can be performed. Both of these are not mandatory: the metamodel in Figure 6 has role cardinality 0,1 in transitions related to Action and Button concept.

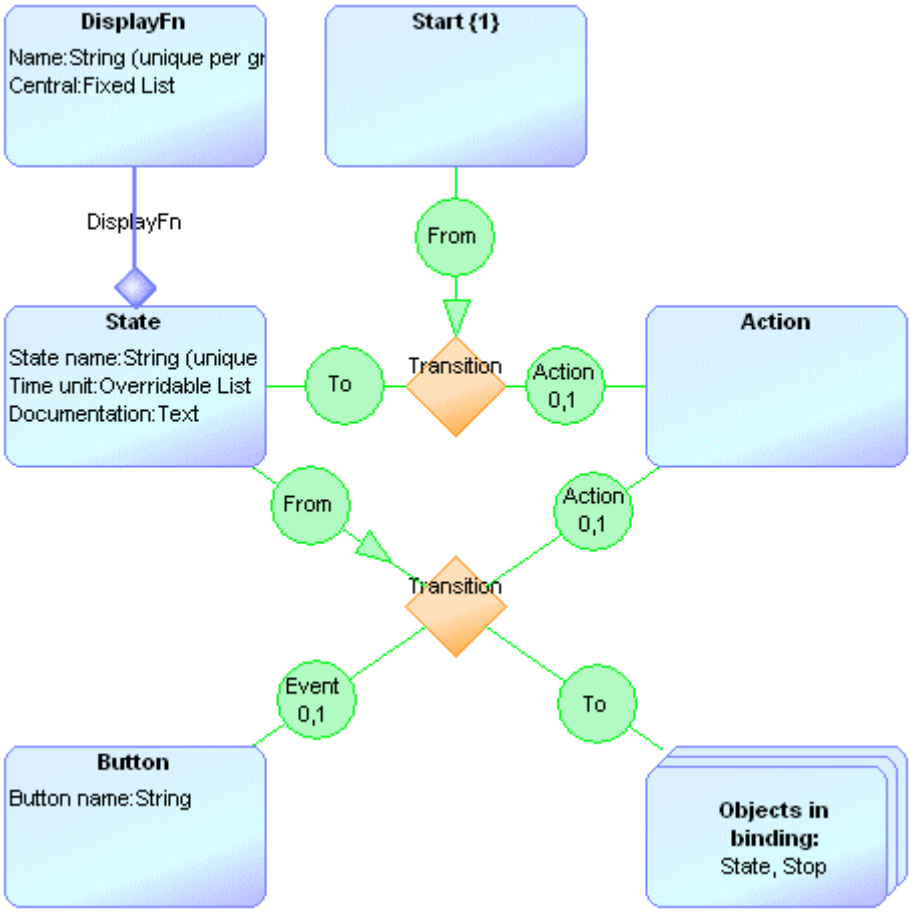


Figure 6. Extended state machine with domain-specific transitions

In a similar manner, the other domain concepts and framework services can be added to the language specification. For example the framework gives services to use icons: they can be turned on or off. This is specified in the metamodel (Figure 7) with a relationship between an action and an icon.

The relationship also has a property to specify which icon service is used (on, off). Another framework service is ringing an Alarm to trigger a special transition to a State. The metamodel in Figure 7 also includes other framework services like setting and calculating time via variables, describing how the DisplayFn calculates the time and how the Alarm is set.

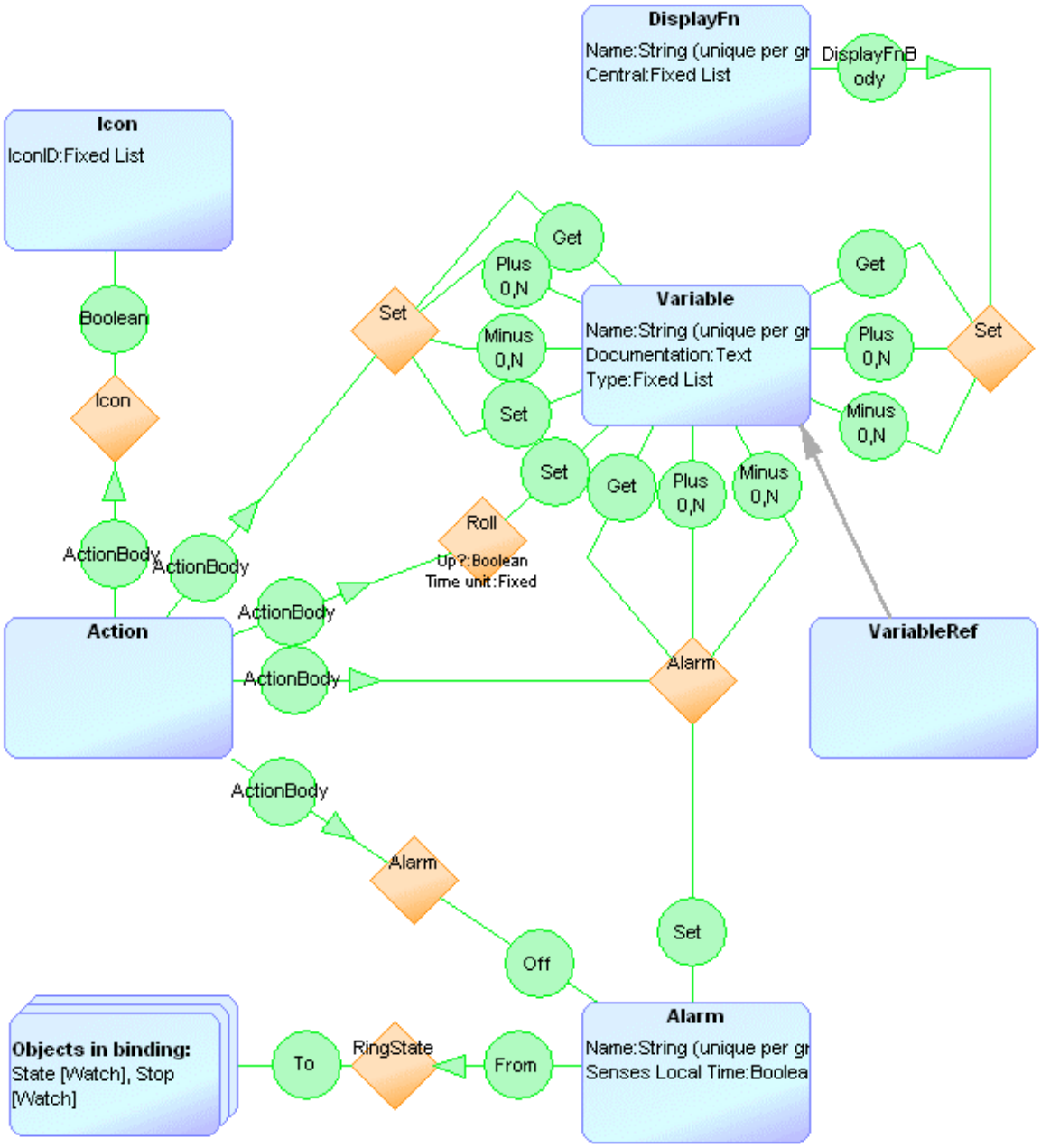


Figure 7. Extended state machine with watch-specific actions

This language specification is formal so it can be instantiated and used as a modeling language. Actually all models illustrated in Figures 1-4 are instances of the metamodel. Design models are therefore always based on the concepts and rules set in the metamodel.

Notation makes a language visible

A pure metamodel, however, is not enough as the language also needs to have notation and visual presentation, usually a diagram, but sometimes a matrix, table, or plain text. The best source for the notation is the application domain. Using familiar symbols from the application domain makes the models easier to read, remember, understand and check. One good way is to look at designs that developers draw when they have complete freedom, e.g. on whiteboards or in presentation slides. Here they can apply the concepts and visualizations that they find most suitable for describing their applications. DSM should mimic and apply these concepts and notations in the modeling language.

In our sample language we have used the notation of basic state machine and extended it with domain concepts like buttons, alarms and icons (see Figures 1-4). We also use coloring to categorize different areas of concern according to the MVC (model-view-controller) architecture: green for the views (icons, displays), red for control path (buttons, triggers, alarms), and black for the underlying data model (time manipulation). Dark red is used for elements that have aspects of both data model and control.

Running the language definition

Once a metamodel, or a part of it, is defined, it should be tested with sample applications. For this purpose we need tools that provide editors for modeling with the defined languages. Best is if the tool for language definition and language use is the same. This makes language definition agile: we can modify the language (metamodel) while using it.

Tools should not, however, be restricted only to editor creation based on metamodels. They should guide in defining the metamodel and recognize if the language definition is incomplete (i.e. illegal in some way) and if a particular language can be integrated with other languages. Tools should also allow sharing language versions among developers and updating existing models based on language changes. This is crucial in practice since companies simply cannot afford to wait for weeks or months before getting a new language version – and then having to update previously made models by hand.

Fortunately, tools are already available that provide good support for this: whilst a quick prototype editor could be built in a man-year or even man-months, to make a usable, reliable tool that developers will accept requires a far greater amount of work. Using a tool that already offers these features, an experienced developer in a company can thus tailor design languages and generators to a specific domain. Other developers can then design with the resulting DSM languages and tools, and generate actual products from their models.

Concluding remarks

Domain-specific languages are a natural extension to frameworks, making them better able to be used. Domain-specific languages work well with frameworks since both focus on a specific application area and on creating similar applications. The investment you need to make first is to build the domain language and related generators. This investment has been found small compared to the payback. For example, the implementation of the watch modeling language as presented above took two days. An additional five days were needed to design the architecture and assemble the framework Java components (without any prior Java experience), and one day to implement the code generator. This development effort also produced the first working watch model. From then on, new watch models could be implemented and tested in less than twenty minutes.

In addition to greatly speeding up development, domain-specific languages on top of frameworks may also offer other benefits:

- Routine work in framework use can be minimized
- The complexity of the framework can be hidden
- Rules embedded into the language can better guarantee that the architecture is followed
- The resulting applications will be of better quality, since they are built based on the rules set by the experienced developers who created the languages

- Changes in the framework can be made more easily available by changing the language in which applications are described. Applications using the newer framework can then simply be regenerated from existing models.

These benefits are not easily, if at all, available for developers using traditional manual programming practices: reading textual documentation about the framework, browsing components in a library, or trying to follow a (hopefully) shared understanding of a common architecture. In short, if you are using or developing a framework you should consider defining a modeling language on top of it. It bears mentioning that defining a domain-specific modeling language is not only an interesting challenge—it is also a lot of fun!

The sample language and related code generators for Java and C are available from jpt@metacase.com upon request.

ARM yourself for Enterprise Application Development

Mark Collins-Cope, info@ratio.co.uk
Ratio Group, http://www.ratio.co.uk

If you have been involved in large scale software development for any length of time, you will surely recognise the following symptoms of application architecture and design decay:

- you keep having to *cut'n'paste repetitive code* (along, of course, with any associated bugs), as proper *re-use* of code is not a viable option. Poor code factoring makes this impossible;
- this in turn leads to *duplication* existing all over the place – making a functional single simple functional change then requires amendments to be made all over the place;
- the *chaotic package structure* of the application means there is a prohibitively expensive learning curve for new staff – there's no clarity of responsibility in what package does what;
- everyone keeps *treading on everyone else's toes* when making changes to the system – lack of any consistent packaging rules make intelligent work scheduling difficult;
- *automated testing is difficult*, it's impossible to test any package in isolation due to poor dependency management;
- *bug fixing is difficult,* spaghetti-like class dependencies make it difficult to track the source of an error,
- all in all, the whole application seems to be *unstable*, simple functional changes require large code rewords, and everything breaks all the time;

In this article I discuss an architectural reference model (ARM) for large scale applications, that I've successfully employed on three large enterprise application developments in the last few years (two in the finance sector, and one in on-line auctioning).

The ARM is made up of five architectural strata (Interface, Application, Domain, Infrastructure and Platform). The overriding purpose of the ARM is to provide a clear set of rules for large scale application decomposition, that encourage separation of concerns, maximises re-use of code (primarily) within the application and (secondarily) across applications, that leads to good code factoring without duplication, and increases application stability in the face of changing requirements.

To achieve this end, the ARM has an associated, and fairly easy to apply, set of rules which force a coherency of structure on your application, leading to: greater clarity of responsibility – as to which code does what, improved code factoring – reducing duplication within the code, increased consistency in packaging rules, more manageable dependencies between packages, thus mitigating to some degree the problems outlined above.

1. Introducing the ARM

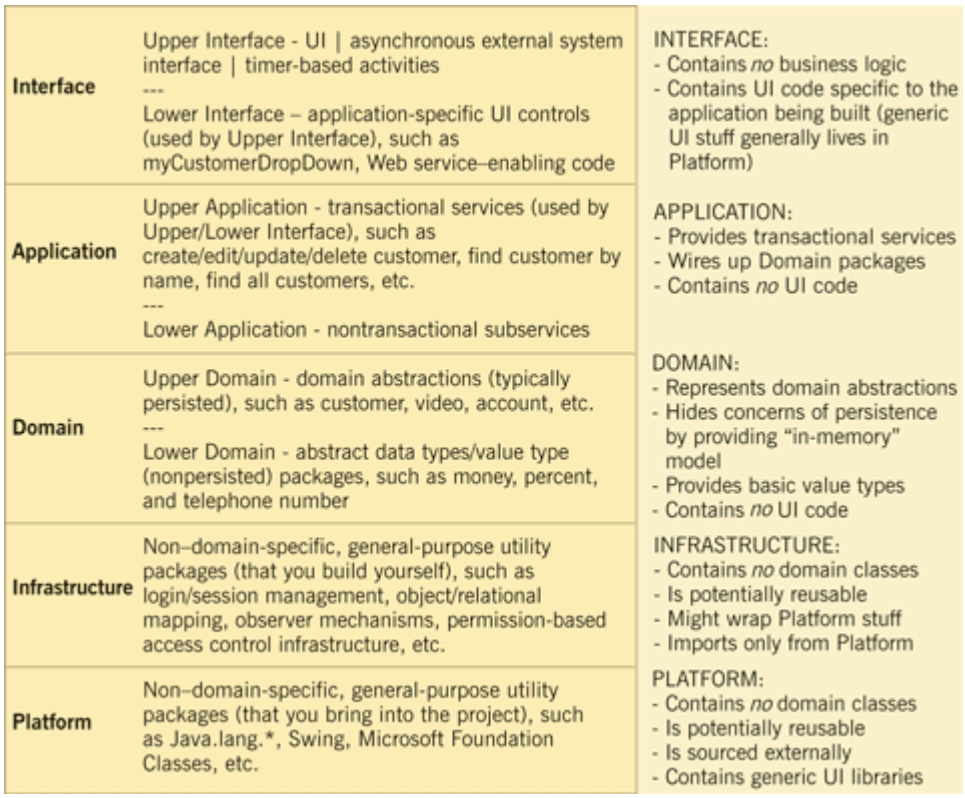


Figure 1 – The five strata of the architectural reference model – each of which acts as a placeholder for one or more packages. The stratum in which a package "lives" is determined by what the classes within the package do, and what other packages the classes depend on.

1.1. Application sub-division

The ARM – see figure 1 - is based on five fundamental architectural strata – each of which I will describe in some detail in the coming sections. Each stratum acts as a *placeholder* for the packages that, combined together, will make up the source code to your application. It's important to note that the stratum are *not* themselves packages, but – as per the rules in the following section – help to determine what should and shouldn't be in any individual package.

The ARM has two sets of associated rules: general rules – which apply across the whole model, and stratum specific rules, which apply to each stratum individually:

1.2. The rules (general)

- *strata are divided according to functionality and 'natural' dependencies.*
  Each stratum has an associated set of responsibility guidelines (a.k.a. concerns) which determine what the packages and classes within it are allowed to do. Each stratum is 'naturally' dependent on strata below it – in the sense that the functionality of the classes within it will be built using the facilities provided by lower stratum classes and packages.

- *depend downwards*.
  Packages in a given stratum are only allowed to depend on - import from - packages in the same or lower stratum. Dependencies to packages in the same stratum should be avoided where this does not unduly increase complexity.

- *packages can only live in one stratum.*
  A package that contains some classes that belong in one stratum, and some that belong in another, is designated to live in the higher stratum. Well factored applications tend to have a greater number of classes within the lower strata – indeed one objective of the ARM is to put a focus on this type of factoring – so there's a clear implication here that if a package cross strata boundaries you should consider restructuring it. Applications with all or most of the packages in the higher strata – as per this rule – tend to be poorly factored and often exhibit the problems discussed at the beginning of this article.

1.3. The rules (stratum specific)

- *Platform underpins the application development.*
  Platform contains packages or utilities that are acquired externally to support the development. Typical examples include Jakarta Struts, java.lang.*, Swing, Microsoft Foundation Classes, .NET GUI libraries, etc. Although not discussed greatly in this article choice of Platform technology is a key to ensuring project success –hence its inclusion in this model.

- *Infrastructure neither contains nor depends on domain-specific code*
  Packages in *Infrastructure* are those which contain general purpose (non-domain specific) classes that provide utility functionality applicable to many different types of application. Infrastructure may only depend on (import from) platform. Typical examples include: general purpose object/relational mapping code (persistence); general purpose observer mechanisms; general purpose group based security mechanisms; and thin wrappers imposing a restricted API on Platform functionality.

- *Domain contains domain specific classes (often called entities)*
  Packages in *Domain* contain domain specific abstractions that you would typically find in an entity relationship or domain model. User and/or external system interface/presentation code is specifically prohibited. In the context of enterprise applications, Domain should provide the illusion (abstraction) that all domain classes are in memory, hiding the details of any persistence mechanism. Domain may also hide other Infrastructure concerns, where this does not introduce needless complexity.

  The upper domain stratum contains packages made up of "reference" objects – those typically persisted in their own right (with primary key, etc.) in a relational database. Typical examples include: a customers package (providing access/update functionality to the set of customers), an accounts package, etc. The lower domain stratum often contains packages that provide "value" objects – those that are typically contained by value as attributes of upper domain classes. Examples include: telephone number, date, money, etc.

- *Application provides a service oriented architecture*
  Packages in *Application* provide a set of application specific transactional services that the Interface stratum uses to query and/or update the application state. Application packages typically "wire-up" or provide the "linking glue" for decoupled Domain packages (see figure 4). The upper Application stratum will provide transaction services like: createCustomer, getAllCustomers, createAccount, getAccountsByCriteria, etc. Lower Application packages, if present, will contain utility sub-services that are non-transactional, but which are used to construct the transactional services provided by the upper Application.

- *Interface packages make the application do something!*
  At its most fundamental, the purpose of the Interface stratum is to interact with the outside world, and at its request, call the Application stratum to change the application's internal state. Most commonly the Interface holds packages that provide application specific user interface – that is - user interface classes that are particular to the application in question (*not* general purpose UI toolkit classes). Interface may also contain code to parse application specific file-interchange formats (e.g. a custom XML format), deal with requests from external systems (web-enabling code), or application specific timer related activities.

  The Upper Interface typically contains complete user interface dialogs such as the "create a new customer" screen, or the "find a video" screen. Lower Interface, if present, may contain application specific user-interface "widgets" such as a "find video by name" pane (one used on multiple screens), or an account-type drop-down listbox (containing values such as 'current', 'savings', etc). Lower Interface widgets are thus used to build upper Interface user interfaces.

2. Video stores – a case study

That is enough theory for the moment. I will now show you an example based on a video store application. Note that the apparently simple set of requirements – see figures 2 and 3 - are sufficient to pose some architectural complexity.

- customers can register for email or text message alerts that tell them when a video is available to rent
- customers can reserve a video by replying to a text message alert
- customers can search for videos by partial title, and reserve a video (using the GUI)

Figure 2 – requirements fragment for iteration 3 of the video store application. The application has two type of user interface – one GUI based and one mobile phone based. Figure 3 shows the associated domain model for this application.

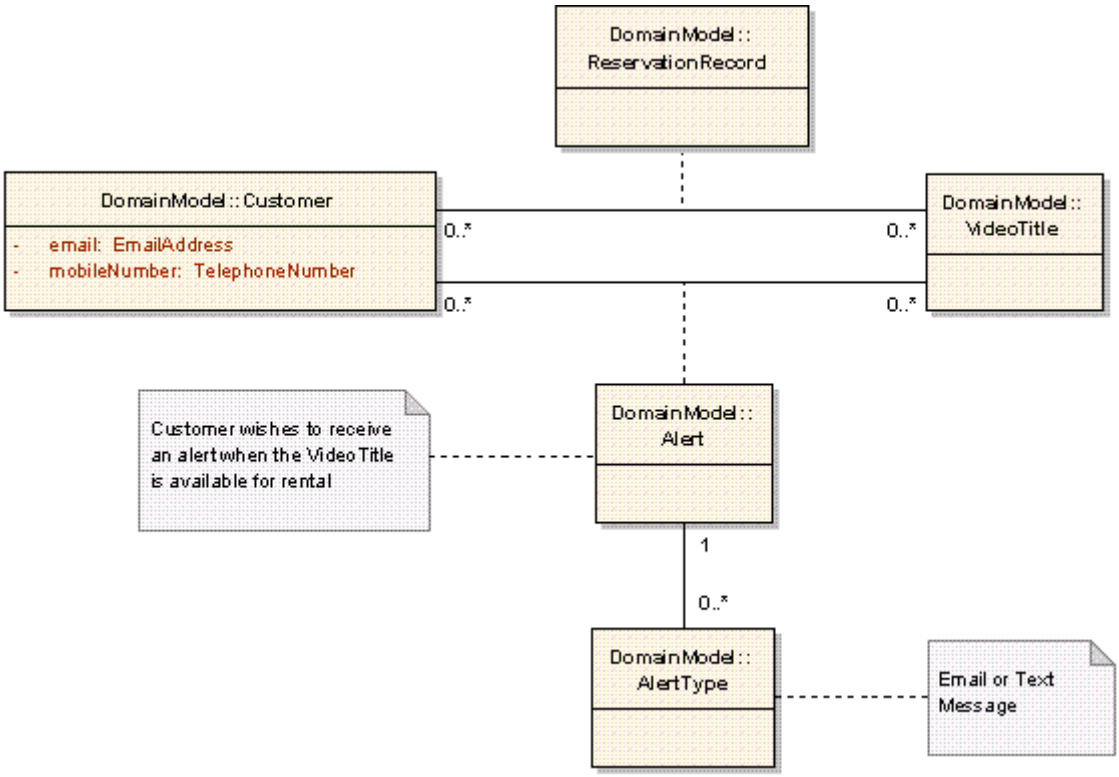


Figure 3 – domain model for the video store application.

Figure 4 shows the architecture resultant from iteration 3 of the video stores project.

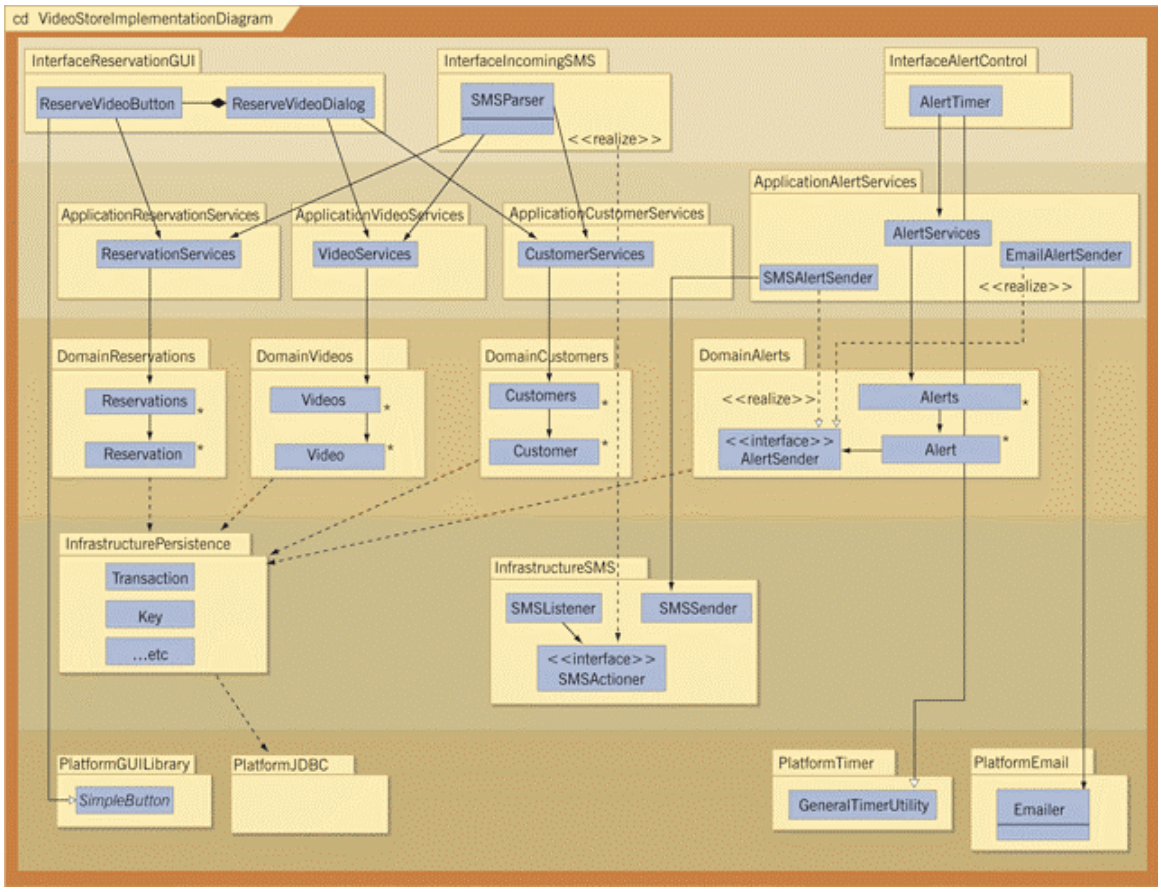


Figure 4 – detailed view of the architecture of the video store system, with strata superimposed. Package names also indicate the stratum of a package. One infrastructure package (SMS) has been culled from a previous project. The Persistence package (which manages the object/relational mapping in a domain independent fashion) is shown only in outline.

2.1. Interface

Interface contains three packages, each of which is responsible for kicking the application into life:

- InterfaceReservationGUI – responsible for the UI for reserving a video,
- InterfaceIncomingSMS – responsible for reserving videos in the event a customer responds to an SMS alert, parsing the SMS text and creating a reservation for the video.
- InterfaceAlertControl – responsible for sending alerts periodically.

Notice that all three of these packages depend - due to either inheriting from a class or implementing an interface - either on Infrastructure or Platform packages. In all three cases, control is passed to Interface code from Infrastructure or Platform code via a call-back (using a technique called Inversion of Control or Dependency Injection).

This is not an uncommon situation, and occurs when code is factored to remove application and domain specific dependencies – leading to better code factoring and making the SMS package, in particular, usable in multiple contexts. The ARM gives context to this type of code factoring.

2.2. Application

Application contains four decoupled packages, each of which provides a set of services that can be used by Interface code. Note that ReservationServices code in figure 4 is used by the both the SMSParser and the InterfaceReservationGUI packages – each packages provides a different user interface to the same underlying functionality. This type of code factoring is a key motivation in the separating the concerns of Interface and Application.

2.3. Domain

Domain contains four packages – one for each entity identified in the domain model – each of which follows a common project pattern (see figure 5).

What is particularly noteworthy here is that each package in Domain is completely decoupled from the others. None of the Domain packages depend on each other: Videos don't know about Customers, Customers don't know about Videos, and perhaps most surprisingly Reservations (see figure 6) don't know or depend directly on either.

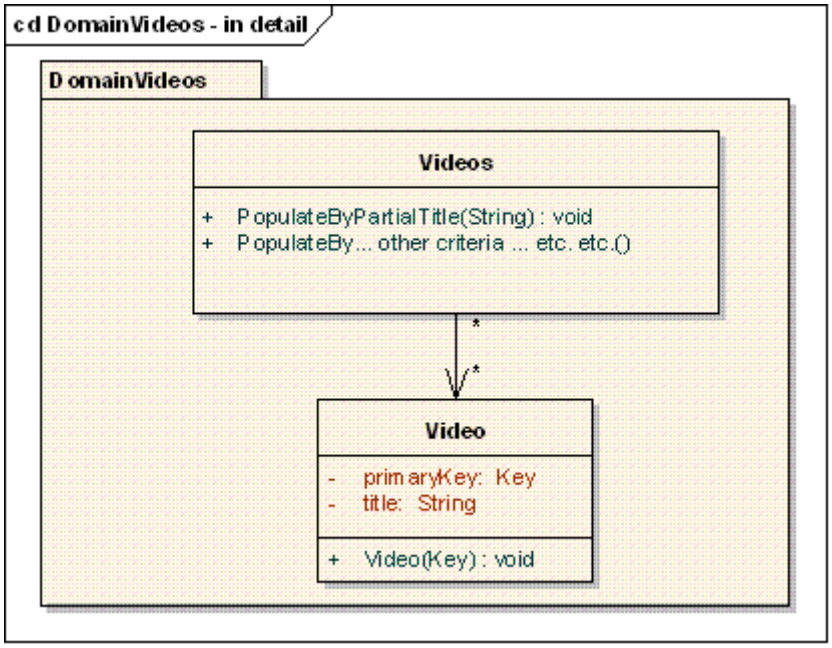


Figure 5 – detailed view of the contents of the DomainVideos package. Videos provides the mechanism by which collections of video Keys can be returned to Application code. Individual Keys are then used to instantiated individual Videos. The other domain packages in this example follow this pattern.

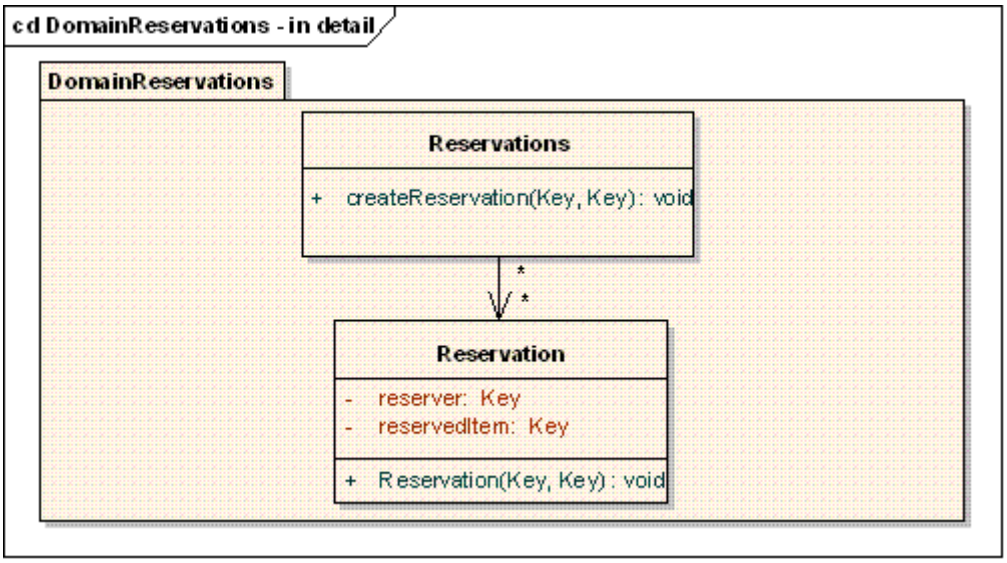


Figure 6 – The DomainReservation package is completely decoupled from DomainVideo or DomainCustomer packages. This is achieved using a simple technique, namely by the Reservation class storing only the Keys of the reservedItem and the reserver. In this case, it is up to Application code determine what it does with these Keys. A variety of other techniques can be used to achieve Domain package decoupling. For example, Application code may often wire up Domain packages using Adapters. Note that the Key class shown here is imported from the Persistence package (see figure 4).

The DomainAlerts package both keeps track of alerts and sends them when asked to do so. In figure 4, it doesn't actually know the mechanism(s) by which an alert can be sent, so it provides an interface (of the Java type) which is implemented by ApplicationAlertServices. If only one alert mechanism was needed, this might be considered needless complexity.

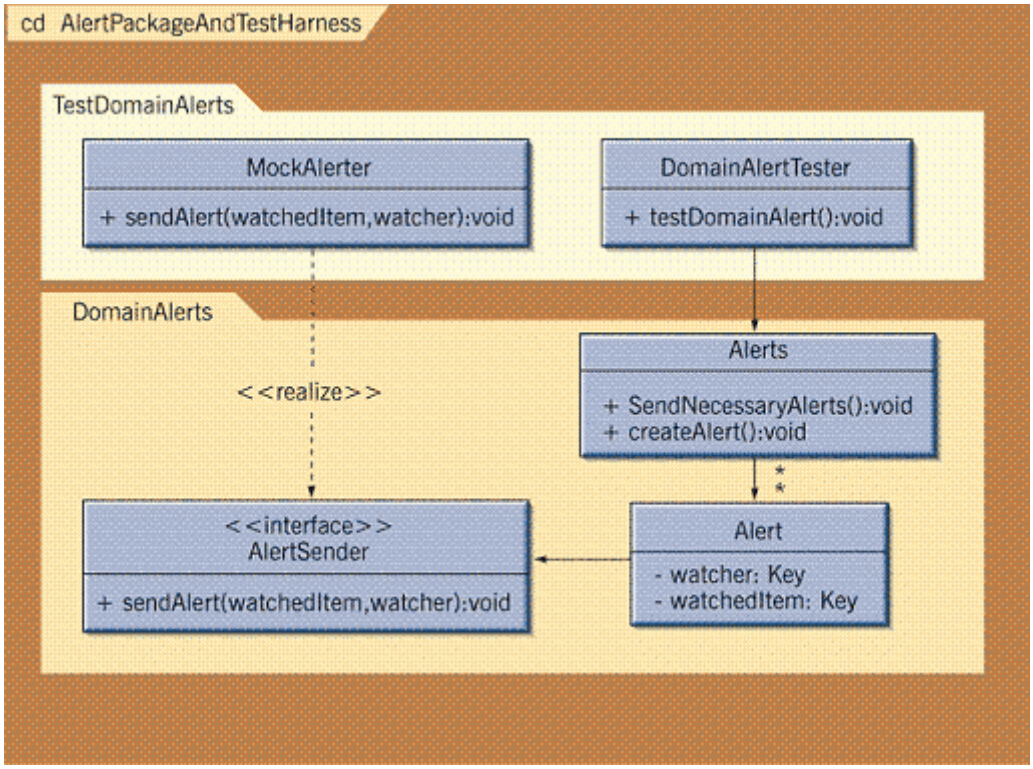


Figure 7 – TestDomainAlerts masquerading as an Application package – adopting a client code perspective. To test DomainAlerts in isolation, TestDomainAlerts provides a MockAlerter, which enables it to verify that alerts are being invoked correctly. This is an example of dependency injection in action.

However, Alerts can be sent by email or SMS, so this design results in less code being required overall, and has the additional benefit of making automated testing an easier proposition (see figure 7). The design is now also inherently extensible. It is not atypical to see examples of Domain packages having their behaviour customised, in this fashion, by Application packages – indeed, this is one of the drivers for the separation of Domain and Application.

2.4. Infrastructure

Infrastructure contains two packages. The InfrastructurePersistence package is worthy of an article in its own right – but for our purpose here let's just say that it manages the interface to a relational database, exports the Key class (and probably a Versioned Key class) and Transaction (unit of work) class, and makes sure things are stored in the database. As shown, both Application and Domain classes rely on Persistence facilities directly. In particular each service exported by Application packages will use Transaction class facilities.

Of more interest to us here is the InfrastructureSMS package. A package providing general purpose SMS facilities is an obvious candidate for inclusion in Infrastructure – and could be genuinely re-used across many projects. In the example shown, a SMSListener class is instantiated to listen on an appropriate number (i.e. the number on which we are expecting to receive incoming text messages). Incoming messages are forwarded to the SMSActioner interface, which in this example is realised by the SMSParser class in the InterfaceIncomingSMS package. See figure 8.

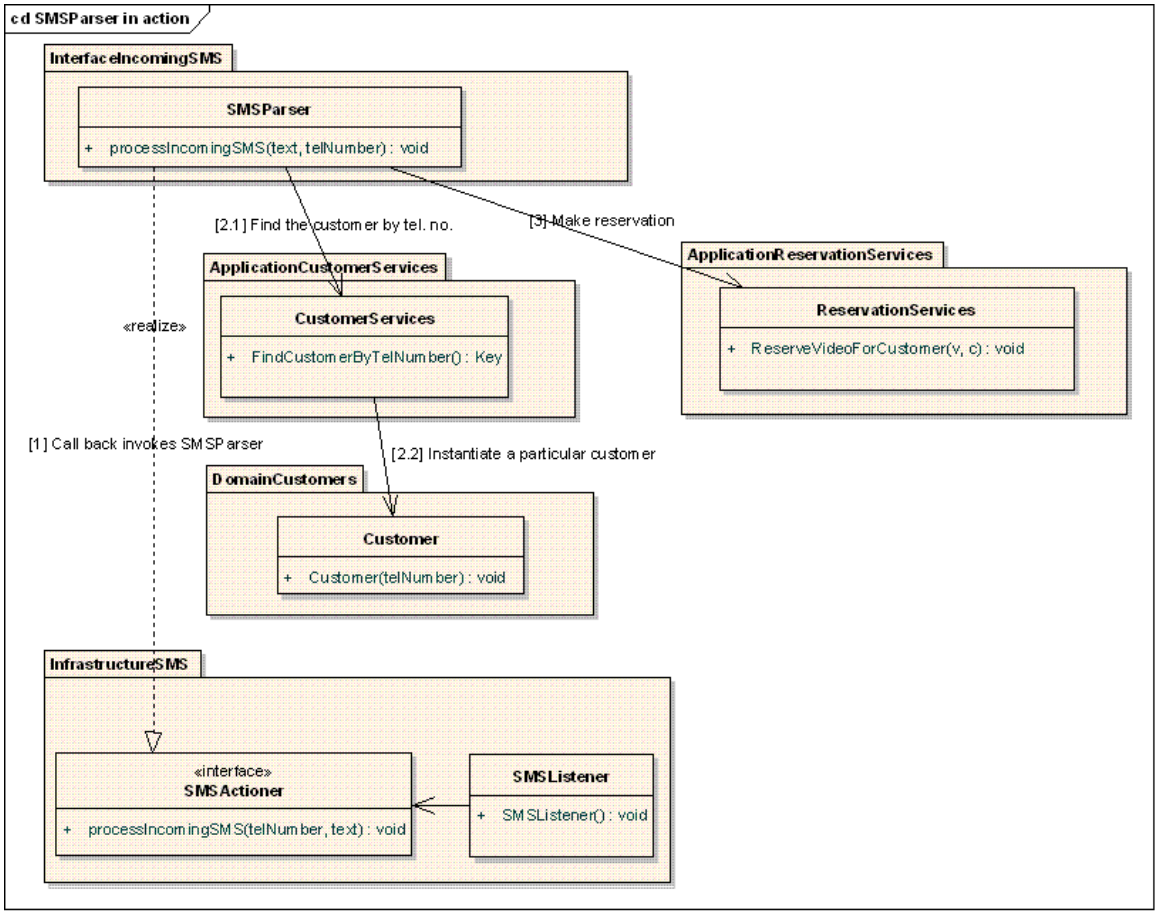


Figure 8 – [1] An incoming SMS invokes a call-back to the InterfaceIncomingSMS package. [2.1] SMSParser.processIncomingSMS parses the incoming message to extract the Video ID to be reserved (contained in the message) and [2.2] looks up the Customer who sent the SMS by their telephone number, before finally [3] making a Reservation.

2.5. Platform

Platform contains the building blocks that underpin the whole development. If you assume the video stores application is written in Java, it will most likely use standard Java libraries to build the GUI, to interface to the database (JDBC), for the basic timing mechanism, and to send email.

3. The ARM revisited

3.1. A conceptual *and* practical model

As you can see from the above example, the reference model has been of conceptual - in terms of assisting in how to think about application structure - and a practical - in terms of concrete package sub-division visible in source code - assistance in getting us to a well structured, well factored application with a coherent and manageable set of dependencies.

Summarising:

The ARM (including its rules) prescribes how to horizontally sub-divide a large application into packages based on the responsibilities assigned to classes, and the natural dependencies that will exist between these classes.

3.2. A deeper look at the strata

So what exactly does it mean that one stratum is "on top of" another in this model? The ARM pulls together three threads of reasoning about good application structure:

- *Dependencies.* Most simply understood is the issue of compile-time dependencies. Packages in a higher stratum import from packages in the same or lower stratum. Put another way, the dependencies always point downwards. Understanding and managing your dependencies is a key feature of flexible application architecture and the ARM is there to help you do this.
- *Functional specificity/neutrality.* The higher the stratum, the more application specific and functionally powerful the operations provided, and the nearer you get to having a complete application. Put another way, the higher you go, the easier it should be to build your specific application with the facilities provided. Conversely, the lower you go, the more work you will have to do to build a specific application, but the more open the range of possible application you could write is.
- *Stability.* The higher the stratum, the less stable (in the face of changing customer requirements) packages become. It's far more likely that an Interface package will change than a Infrastructure package – assuming you've factored your packages correctly – and the whole point of factoring out domain specific functionality from Infrastructure is make it stable against change. There is a *but* here, however. Given the current state of programming language technology, changes in non-functional requirements (say, changing a single user in-memory application to a multi-user database application) can still pull our foundations out from under us. Hence the desire to tie-down non-functional issues as early as possible in a project lifecycle.

Dependency management and functional power/specificity are integrally related concepts, for the pure and simple reason that we build higher level functionality out of lower level functionality - and hence depend on it. Stability is a by-product of factoring out "higher" level functionality that is more likely to be subject to change. In combination, these factors make the ARM a powerful weapon in the enterprise architects' armoury.

3.3. Why five strata?

As you have seen, Platform is the home of the technology base that underpins your application development. Getting your Platform components right can be a project in its own right – hence its inclusion in the ARM - even though it has a lot of similarities with Infrastructure. The distinction between Platform and Infrastructure is, however, very clear: If it is externally sourced and non-domain specific – it is Platform. It is written internally, it is Infrastructure - again, assuming no domain dependencies have crept it.

The distinction between Interface and Application is also fairly clear cut – if a class is directly related to application specific user interface – then it's Interface code; if some code is there to deal with external system integration e.g. web-services – then it's Interface; if the code is dealing with parsing an application specific file format, e.g. an application specific XML format, then it's Interface. But note here, in the same way that user interface code splits into general purpose (Platform) and application specific (Interface), so most XML parsers are Platform. XML itself is domain-neutral – but specific DTDs will require custom Interface code to be written to interpret domain-specific concepts.

The distinction between Application and Domain is sometimes a source of questions. As I said earlier, Application code is there to provide a set of transactional services for Interface code to use. Domain code is there to provide decoupled domain-level abstractions like Book, Account, and so on, which are then combined by Application code to provide application functionality. Without Application packages, decoupled Domain packages would remain forever decoupled – and there would be no application!

The Application/Domain divide is also important from a re-use perspective – and here I am talking about re-using Domain code within the single application discussed. In figure 4 we saw how the DomainAlerts package was customised by the Application layer in two different ways – one to provide text message alerts, the other to provide email alerts. Pushing the common alerts code down into Domain assists achieves re-use through improved code factoring, and also localises most alert related code into the DomainAlert package. This, in turn, will reduce the cost of functional change in the way in which alerts are supposed to operate.

One final motivation for the Application/Domain divide is testability – an example of this is shown in figure 7. Whilst is it both possible and advisable to undertake automated "functional" testing using the service oriented interface provided by the Application stratum, it will difficult, using this approach, to get a high degree of test coverage. Testing domain packages in isolation, as per figure 7, enables you to greatly improve the overall test coverage and hence overall application reliability.

4. Any questions?

- *We do agile development, this seems like big up front design…*
  Design guidance and development process are orthogonal concepts. You might arrive at an architecture by months of forethought, or you can let the ARM assist you in evolving your architecture in an incremental fashion. For the purposes of this article at least, that's up to you!

- *What about the cost (in code terms) of all these strata?*
  The ARM is about packaging. Every line of code you write using the ARM should be absolutely necessary to meet the needs of your customers, the needs of automated testing, and the needs of delivering a high-quality well-factored application that shows intent at a

code level. You don't have to write any code *because* of the ARM – you just have to put the right code into the right package!

- *Dependencies seem to go right across the strata, should not there be a rule saying one stratum can only use the stratum below it.*
  No. Consider the ReservationVideoButton in the example above – which inherits directly from the PlatformGUI package. Applying a rule like this, it would be necessary for each of the three intervening strata to provide a wrapper hiding the facilities provided by the stratum below it. To mandate this would be to mandate what is potentially needless complexity.

- *Is Persistence always Infrastructure / Platform?*
  No, not always. It takes some considerable effort and skill to build a general purpose persistence mechanism, and this may be overkill for a single project. In such circumstances, it is not uncommon to write domain-specific persistence mechanisms (sometimes in the form of brokers for various domain classes). This will tend to lead to some code duplication (it's the duplicate code you would ideally factor out into an Infrastructure package) – but may be necessary. It really does depend on circumstances.

- *Can the ARM be used in conjunction with code generation?*
  Code generation, for example generating domain-specific persistence code in the EJB style, is really an orthogonal concern – but it can be the cause of confusion. If you find you are getting confused – take a look at the code that is generated and see where it fits into the ARM.

- *What about vertical sub-division of the application – the ARM only deals with horizontal sub-division, doesn't it?*
  That is right. As I alluded to earlier, there is a 1-to-many relationship between a stratum and the packages it contains. For completeness, you need some sort of guidance in this sub-division. Help is at hand here in the form of two principles of packaging:

  The *Common Closure Principle* or CCP [RMartin], which states: "package things together that change together". DomainAlerts contains the Alerts, Alert and AlertSender packages precisely because the are highly inter-dependent (highly coupled), and a change to any of them is likely to affect the others.

  The *Common Re-use Principle* or CRP [RMartin], which states: "package things together that are used together". ApplicationAlertServices (see figure 4) contains three classes which are apparently (from the perspective of the relationships *within* this package) unrelated. A closer inspection, however, reveals that AlertServices.SendNecessaryAlerts method cannot be used *without* supplying one or more concrete AlertSenders – in the example shown the EmailAlertSender and SMSAlertSender classes. QED the common re-use principle applies.

- *What practical steps can I take to apply this model on my project?*
  It is remarkably simple to use the ARM to keep track of your project's package structure. All you need is a large whiteboard, which you divide into five horizontal sections as per the strata. Then, draw a package symbol with the package name on the board for each package in your system. Show the dependencies between packages as arrows - all going downwards, of course. This type of diagram is known as a *package map*.

5. Summary

In this article, I have presented an architectural reference model that I have used successfully on a number of Enterprise Applications. The model and its rules are intended to assist you in improving the structure of your code, in particular to ensure greater clarity of responsibility at the package level, to improve overall code factoring, to reduce duplication within code and to enable you to manage your package dependencies effectively.

Coming full circle to the problems that introduced this article:

- the ARM encourages *repetitive code* to be *pushed down* into a lower stratum. Repetitive Application code into Domain or Infrastructure (depending on whether it has domain dependencies or not); repetitive Domain code into Infrastructure, etc.;
- this in turn leads to *less duplication* making functional changes simpler;
- *package structure* is now based on a more coherent set of rules, so it should be easier for new staff to work out what code will live where;
- keeping a package map on the wall means *intelligent work scheduling is easier*;
- *automated testing* and subsequent bug fixing is easier, as dependencies have been brought under greater control, and code factoring has been improved by putting the right code in the right stratum;
- the application is now more stable – pushing non-domain specific code down into infrastructure means it won't be affected by functional changes, and pushing repetitive Application code into Domain leads to greater localisation of changes in the face of varying functional requirements.

You should find out more about the ARM by looking:

- In Chapter 4 of Extreme Programming Examined (Succi & Marchesi - Addison-Wesley, 2001).
- At www.ratio.co.uk/architectural_reference_model.pdf (covers similar ground but with a different angle of attack and example).
- At www.ratio.co.uk/whitepaper_4.pdf and www.ratio.co.uk/whitepaper_7.pdf – two finance related case studies that use the ARM (the terminology used had changed, but the fundamental concepts remain the same).

6. References and credits

[RMartin] – Granularity, Robert C. Martin - C++ Report, 1996.

Thanks are due to Hubert Matthews, who worked with me on previous papers on this topic, and to Andrew Vautier and Anders Nestors of Accenture – on whose 1 Million+ line C++ banking project provided much seed material for the concepts discussed here.

Thanks also to Ilja Preuß, who reviewed earlier drafts of this article, and whose comments were invaluable.

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php