UML-based Approach for Documenting and Specializing Frameworks Using Patterns and Concern Architectures

Imed Hammouda, Mika Pussinen, Mika Katara, and Tommi Mikkonen
Institute of Software Systems, Tampere University of Technology
P.O.Box 553, 33101, Tampere, Finland
emails: {imed, mtp, clark, tjm}@cs.tut.fi

ABSTRACT

Documenting an application framework is a non-trivial task. The most challenging part is the specialization interface that is used to derive specialized versions of the framework. Application developers using the framework should be able to grasp easily the associated classes and their collaborations. Patterns have provided partial support but in the case of highly complicated software platforms, with plethora of conventions, we can easily recognize important concerns that cut across the patterns. In order to better separate the concerns related with specialization interfaces of frameworks, we propose an approach based on concern architectures and UML. Using the approach, the level of abstraction in framework deployment is lifted from patterns to concerns, thus contributing to shortened learning time and shortened application development time. A real-world example illustrating the approach is presented.

1. INTRODUCTION

Application frameworks, defining a common software architecture for a family of applications, are a proven software development approach. A framework consists of predefined classes, some of which may be abstract, and their relationships that facilitate application development for some specific domain.

An important and essential step in application development using a framework is to implement the specialization interfaces (abstractions) of the framework. An acknowledged way of implementing these interfaces is to use *patterns*. First, the individual patterns should be identified. Then, the patterns are put in a unified scenario forming a pattern language for using the framework (see for instance [13, 10]).

Moreover, a framework is meant to be reused. Therefore, clear instructions on how to use it should be provided. A good framework documentation should describe the following three points of view: 1) the purpose of the framework, 2) how to use the framework, and 3) the overall structure of the framework and its main interactions. In the following, we will concentrate on the two latter points.

Concerns, or any conceptual matters of interest, related with frameworks usually deal with features involving collaboration of several objects. Therefore, we do not consider a classical object-oriented approach where classes would be used as primary units of modularity in framework documentation strong enough. This would not enable definition of collaboration between multiple classes, thus overlooking points 2) and 3) above. The next logical step is to use patterns as already mentioned. However, patterns describe concepts of the solution domain rather those of the problem domain. Thus, important concerns stemming from requirements, for instance, can cut across several patterns. The use of patterns thus alleviates the problem but does not solve it.

We propose lifting the level of abstraction from patterns to concerns. Firstly, patterns are used to define the rules and conventions pertaining to the software architecture enforced by a framework. Secondly, to advance separation of concerns, *concern architectures* [14] are used to structure sets of patterns and their dependencies to match different concerns. Then, in order to provide an implementation for a concern, one only has to specialize the framework according to the patterns associated with the concern. This enables aspect-oriented architecting where each specialization step produces a more detailed UML model of the application.

In this paper we introduce an approach following the above scheme for application development using frameworks. The main contributions of this paper are the use of concern level concepts to document and specialize frameworks, and the real-world example illustrating the approach. We also add rigor to the related pattern descriptions and present our vision for future tool support.

2. FROM PATTERNS TO CONCERNS

In this section, we start by discussing conventional uses of patterns in software development and then move forward to present a graphical notation for so called specialization patterns. Finally, a new way of annotating the specialization interfaces of frameworks using such patterns and concerns is introduced.

2.1 Conventional Patterns

Patterns have been widely used in the software community to share proven design solutions and coding standards. Usu-

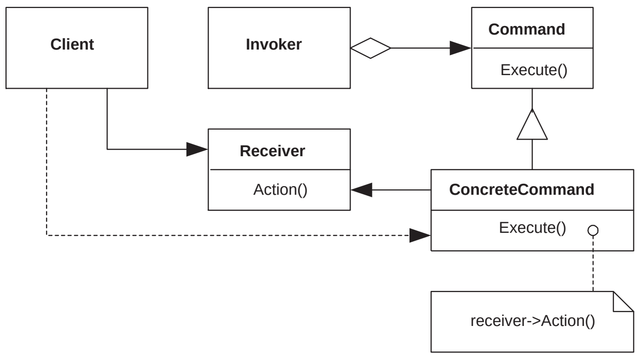


Figure 1: Structure of the Command Pattern

ally, when a new pattern is realized and has been successfully applied, it is published to the outside world in order to be used by other software architects. For the pattern to be correctly used, clear instructions on how to instantiate it should be provided. In most of the cases, the pattern diagram alone leaves the user with no clue on how to proceed with the pattern instantiation process. This can be considered as a serious problem because any pattern misuse can lead to serious design defects or program malfunctions.

Because patterns represent general, and proven design solutions, they should not be bound to a particular implementation or restricted to a specific platform. Instead, a higher level of abstraction is needed. Typically, patterns are documented in terms of UML class diagrams. The class diagram, shown in Figure 1, for example, shows the structure of the Command pattern [9]. The participants of the pattern are: 1) *Invoker*, it asks the command to perform the request, 2) *Receiver*, it knows how to perform the right action, 3) *Command*, it declares the interface how to execute the operation, 4) *ConcreteCommand*, it implements the execute operation and defines a binding between a *Receiver* object and an action, 5) Client, creates a *ConcreteCommand* object and sets its receiver. The pattern structure as shown in the figure expresses the general pattern specification by listing the basic properties. However, for a person not familiar with the pattern, it is hard to figure out how to use it.

2.2 Pattern Role Diagrams

We utilize a pattern concept called a specialization pattern [10]. This is a concrete implementation of the general pattern concept. A specialization pattern is composed of several roles. Each role corresponds to one concrete element (class, method, field). Roles may have properties like dependencies on other roles, cardinality, constraints, and templates. To illustrate these properties, let us consider two pattern roles A and B that stand for class roles. Role A is said to be dependent upon role B if the binding (associating the role to concrete classes) of the role A depends on the binding of the role B. The cardinality of the role A specifies the number of concrete elements that may play the role of the pattern role A. An example constraint on the pattern role A could be an "inheritance constraint" meaning that the class which plays the pattern role A should extend the class which plays the role B.

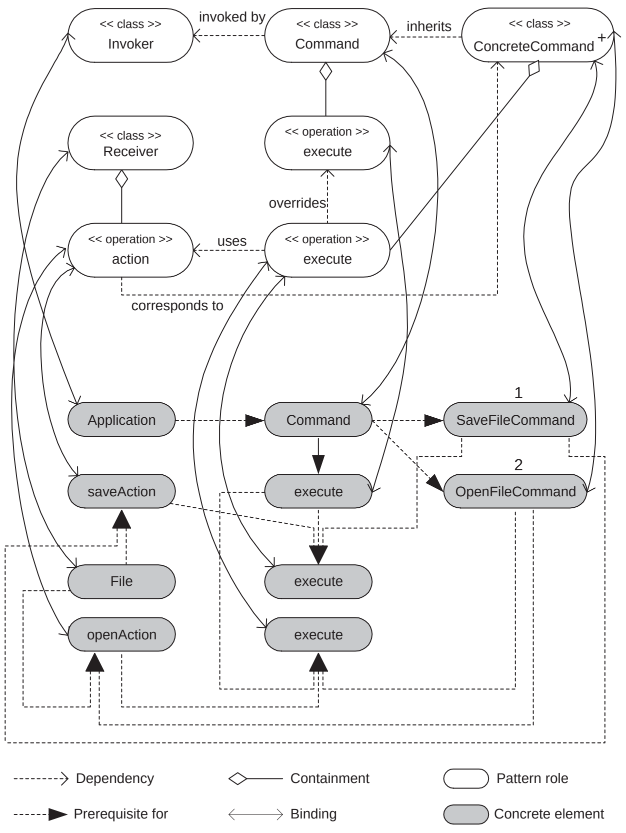


Figure 2: Pattern casting diagram

Specialization patterns are put into a unified scenario annotating the specialization interface of a given object-oriented application framework. During framework specialization phase, the pattern roles are either bound to new elements, for example by creating a new UML class for a class role, or bound to existing UML elements. The action of associating pattern roles with concrete program elements represents a task that the framework user should perform.

In order to elevate the comprehensiveness of the pattern reuse, we introduce in Figure 2 a role diagram, that we call a pattern casting diagram, of the command pattern discussed earlier. Role diagrams have been used in several previous works. In [20], role diagrams are used to document object collaboration based patterns. The proposed notation, however, does not give the reader clear instructions on how to instantiate the pattern. In [10], a role-based graph representation is used to clarify the pattern instantiation steps. The problem with this representation is the ambiguity of the used notation. The arrow notation, for example, represents both a dependency and a containment relation. Also the type of the pattern roles is not indicated in the graph.

In Figure 2, we have used a "pseudo UML" notation on purpose. Instead of using standard extension mechanisms (stereotypes, tagged values and constraints) the goal has been to better illustrate the concepts related with specialization patterns[1]. In the figure, the nodes, marked with a white color, depict the pattern roles. The *Invoker* role, for example, stands for any concrete element that may play the class role *Invoker* in Figure 1. The type of the role is marked by a stereotype. The edges in the upper part of the figure denote the dependencies between the roles. There are two kinds of dependencies: 1) the dependency from role *execute* on the role *Command*, which is marked with a diamond-ended line, represents the containment relationship between the elements that may play these two roles, 2) the dependency from the role *execute* to the role *action*, which is marked with a light-arrow-ended dashed line stands for a logical relationship. In this case, any element that plays the role *execute* should call the corresponding element that plays the role *action*. The multiplicity symbol ('1' for exactly one, '?' for optional, '*' for zero or more, '+' for at least one), that comes along the role name, indicates the allowed number of concrete elements that may play that role. For instance, there should be at least one element that plays the *ConcreteCommand* role. If not otherwise indicated, the multiplicity of the role is 1.

The bottom part of Figure 2 shows the casting diagram of the Command pattern. By casting, we mean the mechanism of associating concrete elements to the pattern roles. The dark-grey nodes depict the actual concrete elements that are bound to the pattern roles. The concrete element *Application*, represented by a dark-grey node, plays the role of *Invoker*. This is marked by the double-arrowed line between *Application* and *Invoker*. In addition, there are two elements that play the role *ConcreteCommand*. This is a direct implication of the '+' multiplicity symbol associated with this role. As a next step, the user might want to provide a third *ConcreteCommand* element, named *NewFileCommand*, for creating new files. In this case, where several concrete elements play the same pattern role, the order of the binding is indicated by an integer index. Moreover, the dark-headed arrows in this part of the figure denote the order how the bindings should be performed. For instance, the binding between the concrete element *Application* and the role *Invoker* is a prerequisite for the binding of the concrete element *Command* and its role.

The pattern casting diagram explained above offers several benefits over conventional approaches:

Simple task lists By task we mean a simple action that adds an element or enforces a property on the model. Tasks are kept simple enough so that their immediate result is easy to track and can be changed if needed. The directed edges in the casting diagram which map concrete elements to pattern roles or to other concrete elements can be seen as simple specialization tasks.

Expressiveness Properties such as dependencies between pattern roles and cardinalities along with their implications can be easily read from and written to our diagram notation.

Computer-aided construction process The idea of expressing the pattern specialization steps as a simple task list has an important practical implication: A tool could be used to maintain the task list implementing the pattern role properties and checking possible constraint violations. Repairing constraint violations can also be treated as specialization tasks. We found the different types of pattern documentation discussed in the other approaches too abstract to be implemented. Without tool support, the pattern specialization process could become too complex and risky in the sense that wrong specialization steps can lead to serious design defects that may not be easily detected.

[1]It should be noted that these pattern descriptions are not visible to the application developer using the framework.

2.3 Specialization Patterns as Aspects

In aspect-oriented software development (AOSD) [2] the focus is on modularity. The idea is to provide means for encapsulation of concerns cutting across several classes or other units of decomposition. The conventional way of building software suffers from the so called "tyranny of the dominant decomposition" [23]. The tyranny is due to the fact that there are always some concerns, i.e. conceptual matters of interest, for instance features, that cut across the units of decomposition.

The tyranny strikes when there is no way to provide one-to-one match between concerns and the units of design and/or code treating them. As an example, consider an implementation of a feature involving several objects of different classes. On the one hand, when changes are needed concerning the feature, the right classes must be identified so that the changes can be done. On the other hand, as the objects may also implement other features, special care must be taken not to break those.

In an object-oriented setting, because requirements are scattered around the classes, the tyranny causes misalignment between requirements, design and code. This has been speculated to cause problems with traceability, comprehensibility, maintainability, low reuse, high impacts of changes and reduced concurrency in development [5].

A known solution [16, 15, 14] to these problems at the design level is to consider architectural descriptions to have two dimensions, one consisting of the classes of the system, and another with the concerns cutting across the classes. On the one hand, the former dimension defines what is usually called the software architecture. On the other hand, the latter dimension gives rise to a so called *concern architecture* (also known as aspect architecture [14]).

In the UML context, the two-dimensional view on architecture was first deployed in [14]. A concern architecture as defined in [14] consists of aspects and concerns modeled as stereotyped packages, and dependencies between the aspects. Because some of the aspects may be shared by more than one concern, concerns do not own the corresponding aspects, instead they are all imported. For legibility, rather than using standard package icons for concerns, they are depicted as encircling lines surrounding the corresponding aspects (see Figure 3).

Specialization patterns, as described in the previous section, are used for annotating the specialization interfaces of frameworks. When a pattern is instantiated, a more spe-

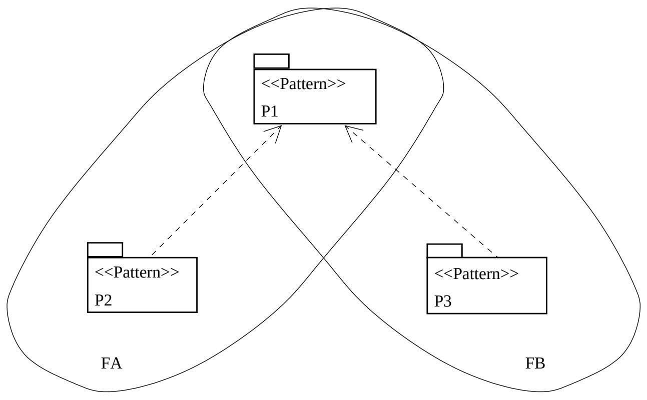


Figure 3: Concern architecture of patterns

cialized version of the framework results. To provide better separation of concerns for specialization interfaces, due to the cross-cutting nature of specialization patterns, they are treated like aspects in the concern architecture explained above. In other words, we propose the concern architecture documenting specialization interface of a framework to consist of specialization patterns and dependencies between them, as well as concerns grouping the patterns.

In more detail, a dependency arrow (see Figure 3) between two patterns implies a partial order for instantiating them. In principle, the patterns can be applied in any order. However, in practice, a well chosen order of application simplifies the casting process involved. For instance, when a pattern is instantiated, a new class can be defined to be bound to a pattern role. If another pattern is instantiated after that by binding its role to the same class, a dependency between the patterns is formed.

To illustrate the use of patterns in defining the concern architecture, an example of a very simple specialization interface implementing two concerns corresponding to features FA and FB is depicted in Figures 3 and 4. In Figure 3, the patterns, their dependencies and the concerns are depicted. In Figure 4, the corresponding UML models and their specialization relationships are shown. On the one hand, the former figure relates the patterns to concerns in order to document the abstractions used in a framework. On the other hand, the latter figure documents the effects of applying the patterns in the order implied by the dependencies depicted in the former. The scheme is described in more detail in the following.

Let us assume that the specialization process is started from scratch, denoted by an empty UML model D in Figure 4. First, pattern P1 is instantiated by defining the concrete classes, objects, and methods etc. needed and binding those to the pattern roles. Obviously, because we started from scratch, there are no predefined elements to bind to the pattern roles. The model obtained by instantiating the pattern specializes D and is denoted by D_{P1}.

Let us then consider the branch on the left hand side of the Figure 4. Pattern P2 is instantiated by binding some of the units defined by D_{P1} to P2's roles. Additionally, new ele-

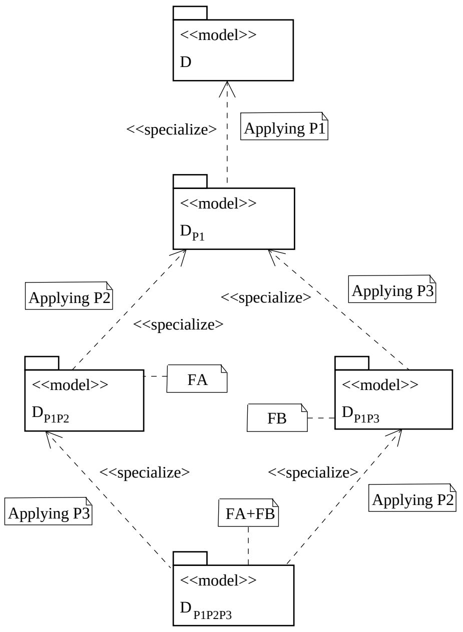


Figure 4: UML models corresponding to the application of the patterns

ments might need to be defined. Those are bound to the remaining unbound roles. This way a new model D_{P1P2}, a specialization of D_{P1}, is created. Similarly, pattern P3 is instantiated after P2, creating D_{P1P2P3} treating both features. However, the instances of P2 and P3 are independent, as seen from Figure 3. The independence implies that the application order of the patterns could have been D_{P1P3P2} instead of D_{P1P2P3}. This option is illustrated by the branch on the right hand side of Figure 4.

Concern architectures raise the level of abstraction from specialization patterns to concerns. For instance, if the feature FB needs to be examined in isolation, the corresponding model D_{P1P3} can be easily identified and P2 discarded. The concerns help in identifying the corresponding patters when needed. If FA needs to be changed and the changes occur only in P2, we can replace P2 with some other pattern P4, for instance. This enables us to solve the problem without touching P3. However, if the changes occur in P1, the overlapping concern FB tells us to be careful also with P3.

The concern architecture encapsulates the pattern relationships in the sense that each dependency in the concern architecture can be mapped to some kind of pattern relationship. This should facilitate the specialization of a framework. It is important to acknowledge that one could define different concern architectures for the same framework depending on the angle of interest towards the framework. Each concern architecture would then map to a different set of patterns.

The approach enables using concern level concept for documenting frameworks. In the case where patterns are used to capture architectural conventions in UML, each specialization step produces a more detailed UML model of the application conforming to the architecture. This will be illustrated by the example in the next section.

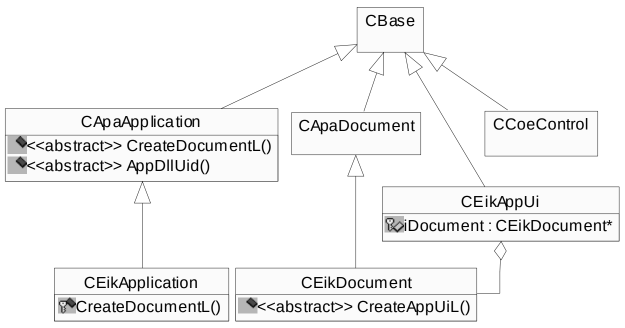


Figure 5: Symbian application architecture

3. EXAMPLE: SYMBIAN FRAMEWORK

The described approach is targeted to be applicable especially in highly complicated software platforms with plethora of conventions. One this kind of a platform is Symbian operating system [24] targeted for mobile devices such as PDAs and smart phones. It contains lots of conventions and mandatory relationships between classes that an application has to conform to. The relationships are demonstrated in the following with a simplified example of an application built on the Symbian application framework. This includes a deployment example of the domain specific version of the more general Model-View-Controller (MVC) model.

3.1 Symbian Application Framework

Figure 5 depicts the framework classes that are used as a base for a basic Symbian application architecture. An application conforming to the architecture consists of five classes: an application class (subclass of CEikApplication), an engine class (subclass of CBase), a document class (subclass of CEikDocument), a controller class (subclass of CEikAppUi), and a view class (subclass of CCoeControl).

Symbian applications are compiled into Dynamic Link Libraries (DLLs). A basic application consists of two DLLs, one for the engine class (and other classes needed by it) and one for the application itself. The engine DLL contains reusable and user interface independent application logic. The use of DLLs enables also binary level reuse (with some restrictions). Both DLLs must have a unique identifier that is referenced in code via a predefined constant that must conform to the Symbian naming conventions. There are also other DLL related conventions not present in regular C++ applications. A few mandatory functions are not methods of any class but should be compiled into a DLL. For instance, each DLL must implement an entry point function called E32Dll, and each application DLL an additional function called NewApplication.

3.2 Concern Architecture

We have developed a set of useful specialization patterns by examining basic Symbian applications. Based on those, one possible concern architecture of the Symbian framework is depicted in Figure 6. It consists of six patterns and several dependencies. This structure has been split into several patterns for better separation of concerns. The Symbian application architecture (shown in detail in Figure 5) depicted on the top of the figure describes the essential classes of the application domain. These classes define the basis on which the patterns will be instantiated on.

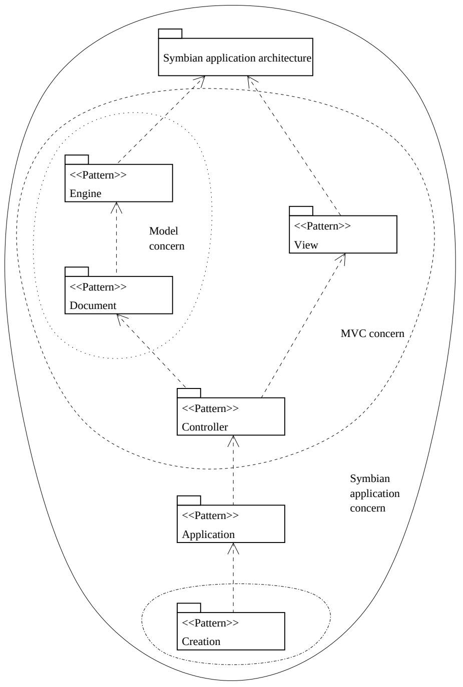


Figure 6: The concern architecture

In the concern architecture, the Symbian application concern itself can be seen to include all the patterns of the framework. This is intuitive because if any of the patterns would be left uninstantiated, the application would not conform to the Symbian architecture. The concern corresponding to the MVC model consists of patterns Engine, Document, View and Controller, of which the two former ones constitute the Model concern.

The Creation pattern forms an interesting concern of its own. Without it, the framework would be sound in the object-oriented sense, but in the domain of the Symbian platform, however, invalid. In the following, we show how the patterns are instantiated and the framework is specialized concern-by-concern.

3.3 Model Concern

The Model concern includes patterns Engine and Document. Figure 7 represents the pattern role diagram related to the Engine pattern. The diagram in the figure (as well as the rest of the diagrams in the section) uses the notation presented in Figure 2.

In order to instantiate the pattern, the application developer has to first provide a concrete class for the role *Engine*. As the role is bound to a concrete element (Figure 7), new tasks offering alternative instantiation paths are provided. In the next step, the user can either provide a name of the code module that is used as the DLL entry point for the engine DLL, or generate constructor or destructor for the *Engine*. The third alternative is to perform a task that repairs an inheritance constraint violation by adding inheritance rela-

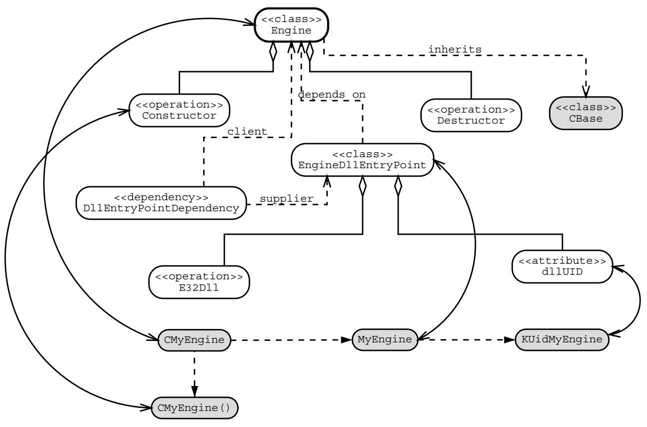


Figure 7: The Engine pattern and instantiation steps

tionship between the concrete instance of the role *Engine* and class CBase.

In the example casting scenario, shown also in Figure 7, the user has decided to provide the DLL entry point and unique identifier for the DLL, and moreover, the constructor for the *Engine*. As the user performs the rest of the tasks, this results in the UML model depicted in Figure 8. The information related to *EngineDllEntryPoint* that is not a part of any class is modeled as a separate class marked with DllEntry stereotype.

The instantiation of the Document pattern depicted in Figure 9 starts by binding *Engine* role into a concrete class, in order to relate the Document pattern with the already instantiated Engine pattern. The user has to bind the *Engine* role of the pattern into the already created instance (in this example case into CMyEngine class). This is the reason for the dependency between the patterns in Figure 6.

After the user has located the class playing the *Engine* role, it is possible for the user to create a class for the *Document* role and perform subsequent tasks that refer to the *Engine* role. The *Document* role, for one, offers possibility for other classes to get reference to *Engine*. This is provided by the public method called Model that is represented by the *Model* role, which returns a reference to the class playing the *Engine* role. Additionally, it is the responsibility of the document class to create the engine class. However, this concern is separated into the Creation pattern explained in subsection 3.7 and does not have to be considered at this stage. The instantiation of the Document pattern completes the binding of the Model concern into the Symbian application architecture.

3.4 View Concern

The View concern corresponds to the View pattern. The instantiation of the View pattern represented in Figure 9 is carried out in the similar fashion as in the earlier cases. Noteworthy at this phase is that the View concern is only bound to the Symbian application architecture and does not yet have any references to other pattern instances (such as those of the Model concern).

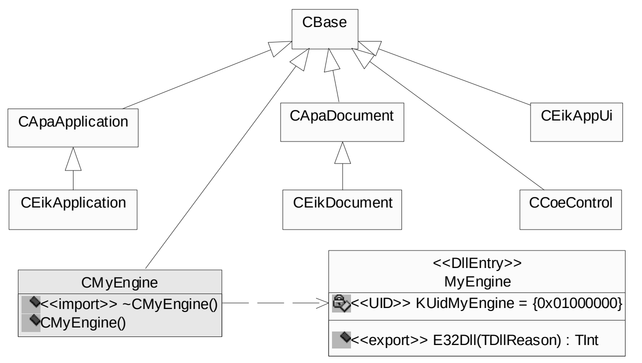


Figure 8: The UML model after the instantiation of the Engine pattern

3.5 Controller Concern

The Controller pattern implements the Controller concern. Figure 9 represents the pattern that glues together the Model and View concerns. The instantiation of the pattern proceeds in a similar fashion as in the earlier phases.

It is the responsibility of the Controller pattern to act as an intermediary between the Engine and View patterns. For example, direct menu commands from the user of the application are handled by *Controller*. Commands have direct implications on the internal state of the application. The state is usually stored into a separate class owned by *Engine* or it can be queried directly from *Engine*. It is the responsibility of the *Controller* to inform *View* if the user actions will have implications on the user interface. Alternatively, *View* can directly register to listen state changes happening in *Engine*. As *View* gets informed about the changes in the state of the application, it must have means to query the current state from *Engine*. Other types of user actions, e.g. keyboard and mouse events, are handled directly by *View* which calls appropriate methods of *Engine*, and requests state changes after user actions have been processed.

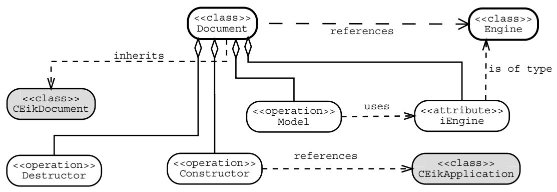
3.6 Application Concern

The Application pattern, handling the Application concern, shown in Figure 9 does not bring much new compared to the earlier patterns. The nature of the pattern is not very crosscutting and it is bound to other patterns only via the Symbian application architecture (Figure 5). Because of that, the Application pattern could have been bound before the other patterns in the deployment order. However, in this example we have selected this order, because it is more intuitive to instantiate it on top of the MVC than the other way around.

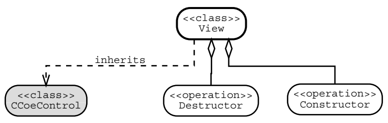
The Application pattern introduces a task of a new kind. After the inheritance relationship between CEikApplication and *Application* has been created, the user is asked to provide the *AppDllUid* method that implements the abstract method defined in CApaApplication class (Figure 5).

3.7 Creation Concern

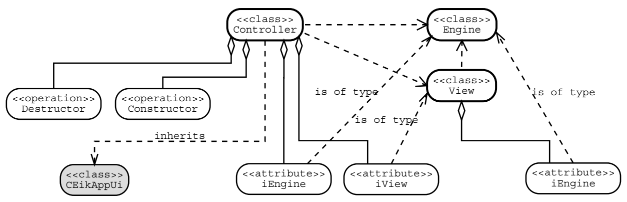
Creation pattern depicted in Figure 9 finalizes the Symbian application concern and ties the class instances into the creation chain. The chain is started when the operating system



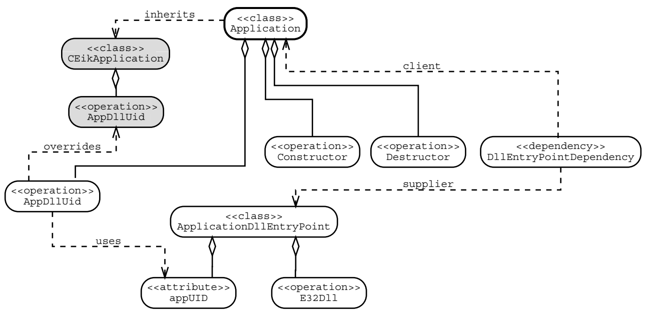
The Document Pattern



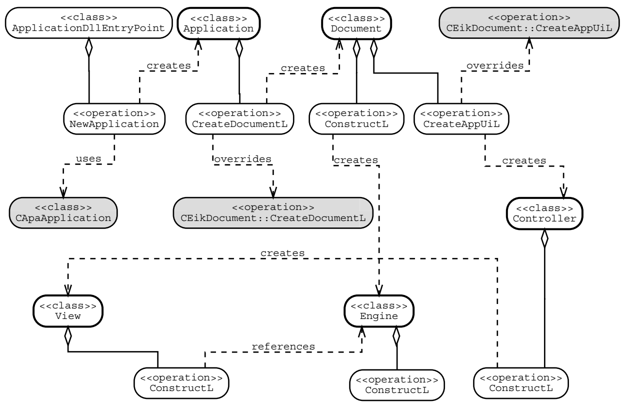
The View Pattern



The Controller Pattern



The Application Pattern



The Creation Pattern

Figure 9: The other patterns

creates *Application* using the NewApplication function implemented in the application DLL. After that, *Application* creates *Document* which, for one, creates *Engine* and *Controller*. Finally, it is the responsibility of *Controller* to create *View*. Naturally, this creation chain introduces relationships between roles.

The Creation pattern is not built around any central role. Instead, one has to first locate all the concrete class instances that play the five main roles of the application. The rest of the tasks just add methods that are needed in the creation chain.

3.8 Concluding Remarks on the Example

The whole Symbian application concern is covered after all the patterns have been instantiated. The resulting UML model is depicted in Figure 10.

By specializing concern-by-concern the application developer gets a good grip on the framework. However, the presented example constituted of mandatory tasks which all had default implementations i.e. tasks could have been performed automatically by using default names and properties for the concrete implementations of the roles. Thus, our approach empowers a generative approach for standard portions. Naturally, this is not always the case, and even in this example at least application specific unique identifiers (*appUID* and *dllUID*) should be provided, and the names of the classes should describe the target application.

Patterns can also be used to describe frameworks containing optional tasks that might be fulfilled or left undone. Whether the application developer decides to perform those tasks or not, the final application conforms to the architectural restrictions and is a legal Symbian application. For instance, our example could have been enhanced to include optional tasks for overriding the default implementations of some of the methods provided by class CCoeControl. In more detail, the pattern developer could have added optional tasks to the View pattern to guide the application developer to define his/her own implementation for handling key events (OfferKeyEventL method), and for processing pointer events (HandlePointerEventL method). The application developer could then implement both methods or just one of them, depending on the needs of the application, and on the capabilities of the target mobile device.

4. TOOL VISION

We have started a tool development project in order to proof our concept. The UML model is represented in Rational Rose [19] and the specialization itself is guided by tailor-made views and dialogs. These views and dialogs are built on the Eclipse environment [8].

Our target is to gain a reasonable level of case tool independence. For runtime representation of the UML model we use xUMLi [1] which conforms to the version 1.4 of the UML meta model [18]. xUMLi provides services for transferring data to and from Rational Rose. Rose Extensibility Interface (REI) is used for data transfer between Rose and xUMLi models. The internal representation of the model is not the same as used by the graphical representation, and the selected case tool is controlled via a well-defined interface that could be ported also for other case tools. Especially the Eclipse community may provide an alternative to Rational

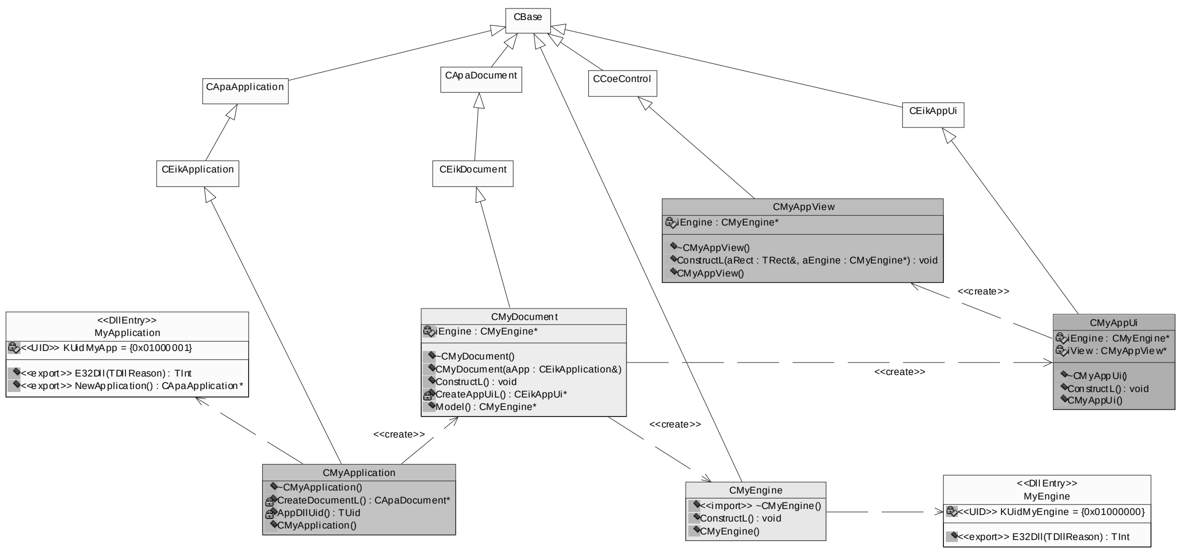


Figure 10: The UML model after the instantiation of the Creation pattern

Rose that is better integrated to Eclipse and allows more interactive use. Related to this, Rational XDE [22] could be a future choice.

Figure 11 depicts the user interface. The main user interface consists of three views: Architecture, Pattern and Documentation View. The Architecture View is used to create new, export, organize and save existing software and concern architectures, and patterns. The Pattern View is used to show and perform tasks needed to instantiate the patterns developed for target software architecture. The Pattern View is used for pattern development as well. The Documentation View plays an important role in guiding the user and providing rationales for each task to be performed.

User interaction during the instantiation phase occurs mainly via the Pattern View, illustrated in Figure 12. Roles that have already been bound to concrete classes are shown in the left pane whereas unperformed tasks related to the selected role are shown in the right pane. Tasks may guide the user to provide a concrete UML element to play a certain role or to repair a constraint violation. A constraint violation task may, for example, inform the user that a certain class must inherit some other class. This kind of constraint violation is repaired automatically as the user performs the task. Constraint violations that cannot be repaired automatically, require model modifications from the user. For instance, the user might be guided to correct the name of a concrete element to conform to a name constraint.

Tasks that guide the user to bind roles to concrete elements invoke dialogs. A couple of example dialogs are also represented in Figure 12. These kind of dialogs provide a way to generate new UML elements or locate existing ones from a model. The latter option is necessary because sometimes a single concrete UML element may play multiple roles. Also, the ability to locate elements enables the reuse of existing UML models.

5. RELATED WORK

The idea of regarding patterns as task lists and checking of pattern integrity originates from JavaFrames (previously known as Fred [11, 10]). In that context, the emphasis was on developing Java programs at the level of program code, whereas here, we have lifted the level of abstraction to UML and architecting with patterns instead of coding with ones. We assume that this will give us improved possibilities to use the approach in large scale systems.

An aspect-oriented technique for implementing design patterns has been proposed in [17]. The authors propose a way to separate concerns related with design patterns from those of the application core. The design patterns are then implemented using an aspect-oriented programming language. We are applying a similar idea but at a higher abstraction level. Their work is closer to the implementation world; the design patterns are implemented using programming languages. In our approach, we express the design patterns in terms of specialization patterns keeping closer to the design world. Moreover, our technique is targeted to enhance the documentation and specialization of frameworks. Besides, we back our approach with a prototype tool support.

The most closely matching tool related to our proposal is Rational XDE [22]. In XDE, patterns are modeled using UML's template collaborations. After the deployment of the pattern, the product of the deployment is considered just as a UML model and the integrity of a pattern is no more supervised after the deployment phase. One can, for example, check that a produced class diagram is a valid UML model, but there is no way to ensure that after user modifications the model still conforms to the previously applied pattern. In our approach the conformance to the architecture expressed by patterns is supervised all the time. As a user makes a change that breaks pattern integrity a new task will appear informing about the violation and telling how to fix it.

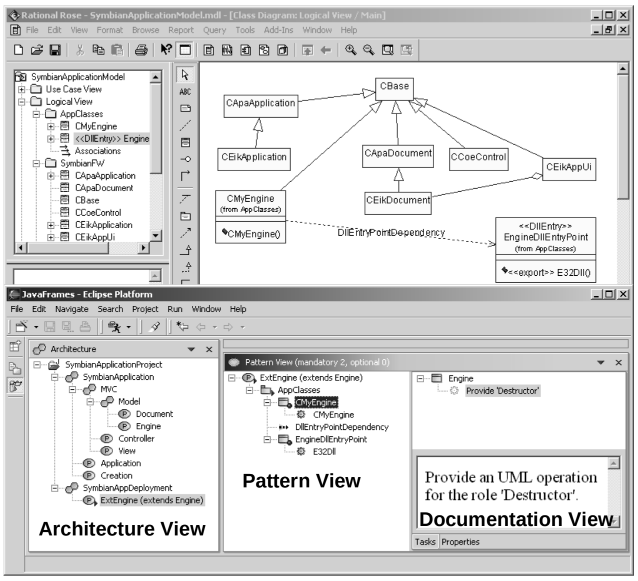

Rational Rose - SymbianApplicationModel.mdl - [Class Diagram: Logical View / Main]
CBase
CApaApplication
CEikApplication
CApaDocument
CCoeControl
CEikAppUi
CEikDocument
CMyEngine
DllEntryPointDependency
<<DllEntry>> EngineDllEntryPoint
JavaFrames - Eclipse Platform
Pattern View (mandatory 2, optional 0)
ExtEngine (extends Engine)
Provide 'Destructor'
Provide an UML operation for the role 'Destructor'.
Architecture View
Pattern View
Documentation View

Figure 11: User interface

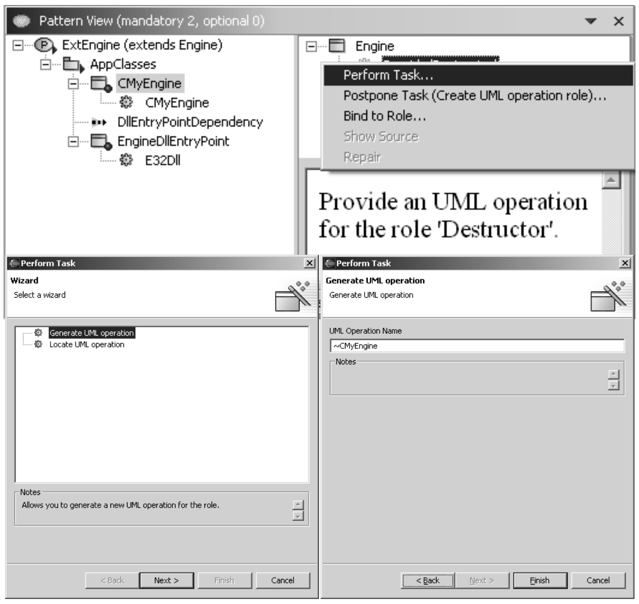


Figure 12: Pattern view and dialogs for user input

The Catalysis [7] method also has some common ideas with our approach, most of which can be traced back to the DisCo method [12, 6]. The specialization process used in our approach corresponds to refinement in Catalysis. However, we have not adapted joint actions [3, 4] into UML. Furthermore, unlike Catalysis, we are able to separate important concerns explicitly at the design level in the terms of patterns by using the concern architecture, as shown in the example. Similarly to Catalysis, experiences with formal method DisCo have had an influence on our approach. In particular, the way concern architectures are treated in this paper closely resembles specification architectures used in the DisCo method [15].

6. DISCUSSION

Using our approach, we see three kinds of advantages over the conventional techniques for framework documentation. Firstly, the explicitly defined concerns help in documenting the purpose of the framework and the incorporated patterns. Secondly, on the one hand, the pattern casting diagrams associated with the patterns describe how to instantiate them. On the other hand, the dependencies between the patterns in the concern architecture define their instantiation order. Thirdly, the pattern casting diagrams specify the internal structure of a pattern. These diagrams could be displayed when the user "zooms-in" to the pattern included in the concern architecture.

Furthermore, using specialization patterns for architecting with frameworks and UML, we can deploy the advantages of guided and task-based specialization support without forcing an application developer to think strictly in terms of programming languages. We believe that with this approach, the design can advance so that we first use more platform-independent patterns, and then move towards platform-dependent ones. The latter ones can be annotated with code templates that bind the pattern to a used programming language. Then, the mapping to a programming language occurs at code template level, and an application developer can choose whether he/she wants to look deeper into the implementation or refer to the more abstract models expressed in UML instead. These code templates that bind model into a certain language of choice are evaluated in a separate code generation phase that provides the code skeleton for a specialized application.

In a more formal sense, we also see an option to define a pattern calculus, which could be used to conjoin patterns even without a base design on top of which they are applied. A step to this direction has been sketched in [21, 14]. Also, our approach should be fairly easy to generalize to other kinds of pattern systems as well.

The expected benefits of the introduced approach come from the better separation of concerns at a higher level of abstraction. This is combined with accurate knowledge about the target software architecture together with its rules and conventions. We use specialization patterns to capture architectural knowledge, and aspect-oriented design practices to document explicitly the (possibly overlapping) sets of patterns treating each concern. The UML model that is generated during the specialization process offers a communication media for developers, and can be studied at any level of specialization. We believe that the benefits of the approach can be seen as shortened learning time and shortened application development time. Also, there are good chances for better quality and maintainability of the application software.

7. REFERENCES

[1] J. Airaksinen, K. Koskimies, J. Koskinen, J. Peltonen, P. Selonen, M. Siikarla, and T. Systä. xUMLi, towards a tool-independent UML processing platform. In *Proc. NWPER*, Copenhagen, Denmark, August 2002.

[2] Aspect-oriented software development WWW site. at

URL `http://aosd.net`.

[3] R.J.R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *TOPLAS*, 10(4):513–554, Oct 1988.

[4] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3:73–87, June 1989.

[5] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. *ACM SIGPLAN Notices*, 34(10):325–339, October 1999.

[6] DisCo WWW site. at URL `http://disco.cs.tut.fi`.

[7] Desmond F. D'Souza and Alan C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[8] Eclipse WWW site. At URL `http://www.eclipse.org`.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Annotating reusable software architectures with specialization patterns. In *Proc. the Working IEEE/IFIP Conference on Software Architecture*, pages 171–180, Amsterdam, 2001.

[11] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Generating application development environments for Java frameworks. In *Proc. of the 3rd International Conference on Generative and Component-Based Software Engineering*, pages 163–176, Erfurt, Germany, 2001. Springer-Verlag.

[12] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proc. ICSE'90*, pages 63–71. IEEE CS Press, 1990.

[13] Ralph E. Johnson. Documenting frameworks using patterns. In *Proc. OOPSLA'92*, pages 63–76. ACM Press, 1992.

[14] M. Katara and S. Katz. Architectural views of aspects. In *Proc. AOSD 2003*, Boston, MA, USA, March 2003. ACM Press.

[15] M. Katara and T. Mikkonen. Aspect-oriented specification architectures for distributed real-time systems. In *Proc. the Seventh IEEE International Conference on Engineering of Complex Computer Systems*, pages 180–190, Skövde, Sweden, June 2001. IEEE CS Press.

[16] T. Mikkonen. The two dimensions of an architecture. A position paper in the First Working IFIP Conference on Software Architecture, February 1999. San Antonio, Texas, USA.

[17] N. Noda and T. Kishi. Implementing design patterns using advanced separation of concerns. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, FL, USA, 2001.

[18] OMG. Unified Modeling Language Specification, Version 1.4, September 2001.

[19] Rational Rose WWW site. at URL `http://www.rational.com/products/rose/index.jsp`.

[20] Dirk Riehle. Composite design patterns. In *Proc. OOPSLA 1997*, pages 218–228, 1997.

[21] M. Sihman and S. Katz. A calculus of superimpositions for distributed systems. In *Proc. AOSD 2002*, pages 28–40, Enschede, The Netherlands, April 2002.

[22] IBM Rational Software. Rational XDE. At URL `http://www.rational.com/products/xde/index.jsp`.

[23] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proc. ICSE'99*, pages 107–119, Los Angeles, CA, USA, May 1999. ACM Press.

[24] M. Tasker, J. Allin, J. Dixon, M. Shackman, T. Richardson, and J. Forrest. *Professional Symbian Programming: Mobile Solutions on the EPOC Platform*. Wrox Press Inc, 2000.