Applying and Evolving the Evolving Frameworks Pattern Language

J.G. Hosking, J.C. Grundy, W.B. Mugridge
Department of Computer Science
University of Auckland, New Zealand
{john,john-g,rick}@cs.auckland.ac.nz

Abstract

We describe the appropriateness of the evolving framework pattern language to our experience in developing the JViews framework for constructing multiple view, multiple notation visual/textual editing environments. This framework has been developed over almost ten years. Because it is quite mature, it is an ideal candidate for calibrating the pattern language against. We have found the pattern language to very appropriately describe the evolution of the framework, but propose a number of extensions to the pattern language based on our experience with JViews and our reported results of the development of other frameworks.

1 Introduction

The Evolving Frameworks Pattern Language [1] describes a collection of patterns useful for constructing large-scale software frameworks together with a typical process connecting the usage of those patterns. In this paper we describe the application of this pattern language to the JViews framework for constructing multiple view, multiple user component editing environments [2]. We have been developing this framework for close to 10 years. Because it is a large and mature framework, developed over many years, it is an ideal candidate to calibrate the Evolving Frameworks Pattern Language against.

The following section describes the Evolving Frameworks Pattern Language (EFPL), including a categorisation of its patterns. We then introduce the JViews framework, providing a rapid overview of its evolution. The bulk of the body of the paper examines each of the patterns in EFPL and their applicability or otherwise to the JViews framework experience. We then discuss areas where EFPL could be extended, based on our recent work with JViews, and reported results of others.

This paper describes experience with using an existing Pattern Language, rather than defining a new Pattern Language. This type of paper is not common at Patterns conferences, and hence there is not an accepted style for presentation. We have adapted the pattern writing pattern language of [3] to suit presentation of experience. This suggests that an extension to that Pattern Language for describing experience with pattern languages would be appropriate. We discuss this in more detail in the final section of this paper.

2 The Evolving Frameworks Pattern Language

Roberts and Johnson describe frameworks to be "reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate". Because frameworks are expensive to develop, Roberts and Johnson developed a pattern language describing the evolution of a framework. They point out however that the pattern language describes "a common path that frameworks take, but it is not necessary to follow the path to the end to have a viable framework". This is because some frameworks die out or, for deployment reasons, remain at the white box stage.

Figure 1, from EFPL, shows the patterns making up the pattern language plotted against time. Although not explicitly described as such in the pattern language description, this consists of four groups of patterns:

- An initiator pattern, "Three examples", describing the genesis of a framework from an initial set of applications
- Architecture patterns, "White Box Framework" and "Black Box Framework", which describe the overall architectural form of a framework at two points in its evolution
- Transformation patterns, "Component Library", "Hot Spots", "Pluggable Objects", Fine Grained Objects", which describe ways in which the structure of a framework changes and matures over time.

- High level tool patterns, “Visual Builder” and “Language Tools”, which describe the evolution of high level generation and debugging tools that make construction of applications using a framework simpler.

The running example Roberts and Johnson used to illustrate the pattern language was the evolution of the MVC framework [4], supplemented by examples from the Runtime Systems Expert framework [5].

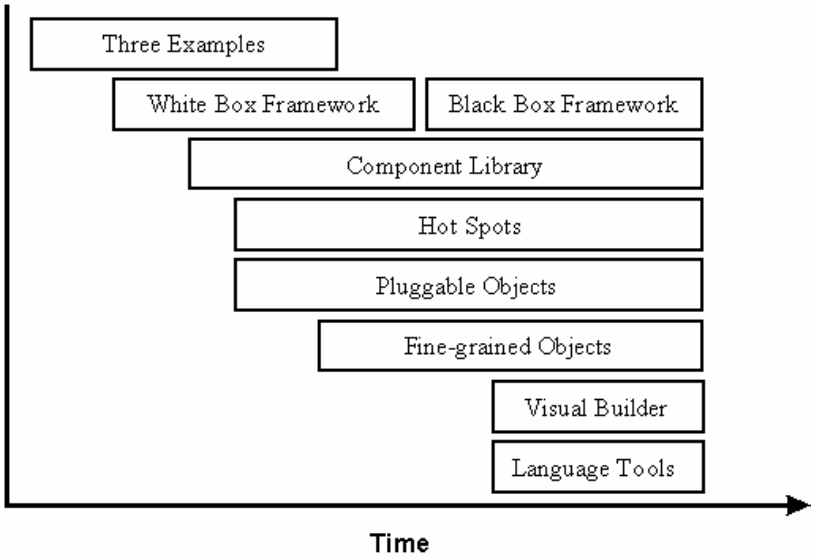

Figure 1: Patterns in the Evolving Frameworks Pattern Language

3 The JViews Framework

The JViews framework has evolved over close to 10 years [2,6,7,8]. Its aim is to support the design and implementation of visual environments supporting multiple views with different representations, and where consistency between the views is required. A typical example of such an environment might be a CASE tool supporting various types of UML diagram.

The framework provides support for specification and implementation of: an underlying shared repository, for storing environment state; the information represented in the various views; consistency management and mappings between views; and visual representation and manipulation of elements in the views. The underlying abstraction of JViews is the change propagation and response graph (CPRG) [7]. Each significant object in the environment is represented by a graph component. Components have attributes representing state. Edges in the graph represent relationships between components, and are themselves components. Changes to the components or the graph structure are captured in the form of discrete change description objects, which are propagated along inter-object relationships. Components may respond to, store, or re-propagate received change descriptions.

The JViews framework provides an implementation of the CPRG abstraction, together with support for constructing 3-layer applications, based on CPRGs. Figure 2 shows the basic architecture. The *display* layer represents visual objects, which can be manipulated by the end user. The *view* layer provides an abstraction of these as a collection of CPRGs, one for each view. Inter-view relationship objects link the view layer CPRGs with the base layer CPRG, which provides a shared repository of environment data. Changes at the display level modify a view CPRG. The change is propagated to the base layer CPRG via the inter-view relationship links and the base layer re-propagates the effects to other view CPRGs. A later extension of the framework supports a fourth layer, which coordinates activities between multiple base level CPRGs, allowing several JViews-based applications to interact with one another in a relatively seamless fashion.

Figure 2 shows an application developed using an early version of JViews [8]. This combines two tools in an integrated fashion. The first is a CASE tool (SPE), which supports a form of class diagrams for visual design of object oriented programs, together with textual class implementations generated from, and kept consistent with, the visual designs. The other tool is a workflow and process modelling tool, Serendipity, illustrated in Fig. 3. This allows specification and implementation of complex hierarchical

process models, such as a model for a software design process, including specification of roles and artefacts applicable for various stages in the process. In this combined environment, Serendipity process models can be enacted, and then changes made within the SPE CASE tool are recorded against the currently active process stage (using the four layer mechanism discussed above), providing a history of changes made at particular stages of the software process.

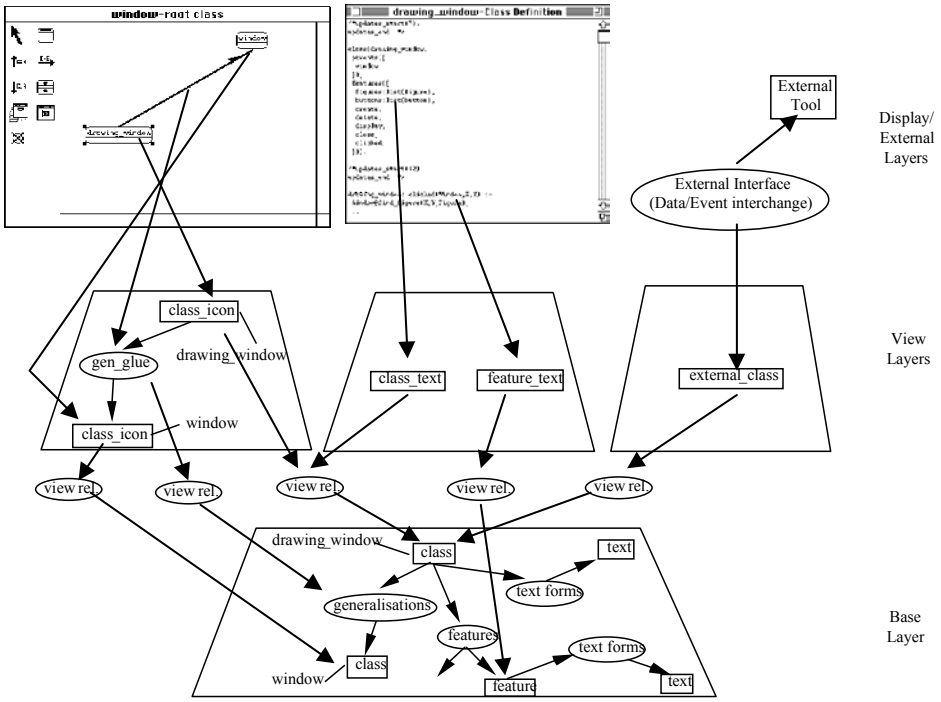

Figure 2: JViews 3 layer architecture

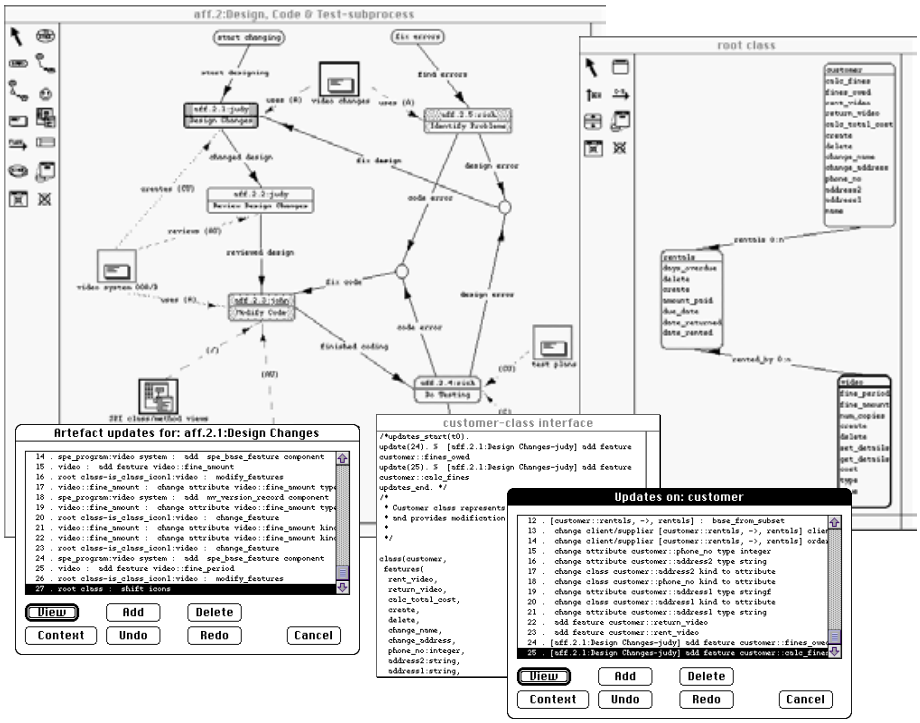

Figure 3: SPE/Serendipity ISDE

JViews (originally called MViews) was initially implemented in Snart, a locally developed object oriented variant of Prolog [6]. Subsequently the framework was re-implemented initially in C++ and then later in Java. Many applications have been constructed using JViews, and a large suite of support tools have been developed for constructing JViews-based applications. In the following sections, we describe the evolution of JViews in more detail, relating that evolution to the patterns in the EFPL. In each section, we provide brief summary descriptions of the EFPL patterns. For a fuller description the reader is referred to [1].

4 Three examples pattern

This is the initiator pattern for EFPL. The **context** is that you have decided to develop a framework for a particular problem domain and the **problem** is how to start designing the framework?

A summary of the **forces** is:

- People develop abstractions by generalizing from concrete examples.
- Having an initial framework makes it easier to develop more examples.
- Projects that take a long time to deliver anything tend to get cancelled, so build something to deliver from the start.

The **solution** is to develop three applications that you believe that the framework should help you build. These applications should pay for their own development.

This pattern matched exactly the initial development of the JViews framework. Initially we developed a multiple view class-diagramming tool, Ispel [6]. From this we abstracted a general framework, MViews (subsequently JViews), which we used to develop an early version of the SPE CASE tool, combining diagrammatic and textual programming [6]. To demonstrate the generality of the framework and to extend it further, we developed an entity-relationship modelling tool, which support visual ER diagrams and textual relation specifications, with consistency maintained between them [9].

In the rationale for this pattern, EFPL argues that "the general rule is: build an application, build a second application that is slightly different from the first, and finally build a third application that is even more different than the first two". The first two applications we developed were both to support tools for object modelling, with the second extending the first to provide better textual modelling support. The third application (the ER diagrammer) was somewhat different in nature to the first two, which both supported object modelling. This allowed us to tease out support for OO modelling environments from that required for other types of environment

EFPL argues that the initial three applications pay for the development of the framework. In our case, money was not a motivating factor as the framework was being developed as a research prototype in an academic environment. However, two related factors provided some motivation for the approach. Firstly, the initial application was developed as a Masters thesis project. Tackling a larger development would have been impossible in the time available. Secondly, each application formed the natural material for a research paper, the equivalent to a monetary payoff in the academic domain.

5 Architecture patterns

EFPL introduces two architectural patterns, being the "White Box Framework" and the "Black Box Framework". These are interesting in that the problem and forces are identical, but the differing contexts lead to different solutions. The **problem** is that some frameworks rely heavily on inheritance, while others relay on polymorphic composition; which should you use?

The **forces** involved are:

- Inheritance results in strong coupling between components, but it lets you modify the components that you are reusing, so you can change things that the original designer never imagined you would change.
- Making a new class involves programming.
- Polymorphic composition requires knowing what is going to change

- Composition is a powerful reuse technique, but it is difficult to understand by examining static program text.
- Compositions can be changed at runtime.
- Inheritance is static and cannot be easily changed at runtime.

5.1 White Box Framework

For the **White Box Framework**, the **context** is that you have started to build your second application. The **solution** in this case is to choose inheritance and build a white box framework [10] by generalizing from the classes in the individual applications.

In our JViews development, we abstracted from the initial Ispel application, separating out classes that provided general support for multiple view environments from those specific and concrete classes related only to the Ispel CASE tool. The relationship between the abstract and concrete classes was implemented as an inheritance hierarchy with method overriding and abstract method implementation used to modify or implement behaviour. In the rationale for this pattern in EFPL, the argument is made that "at this point in the life cycle, you probably do not have enough information to make an informed decision as to which parts of the framework will consistently change across applications and which parts will remain constant". This was quite correct in our case, as we had no clear idea of what the significant building blocks were, and were continually adding new mechanisms to the framework (eg for persistency, collaboration support) meaning that APIs couldn't readily be frozen.

At this point, and as recommended in the implementation section of this pattern, we used the *programming-by-difference* approach of [10] to obtain a clearer idea of what classes were changing with each new application and what could be abstracted into the framework.

5.2 Black Box Framework

For the second architecture pattern, the **Black Box Framework**, the **context** is that you are developing pluggable objects by encapsulating hot spots and making fine-grained objects. The latter are the transformational patterns of EFPL, ie the framework has been applied for a while and the "building blocks" are better understood.

The **solution** in this case is to choose to use inheritance to organize your component library but to use composition to combine the components into applications.

Thus you construct a black-box framework, ie one where you can reuse components by plugging them together and not worrying about how they accomplish their individual tasks [10]. In EFPL, the argument is that hierarchies are good for organising categories of components, and hence are useful to find things in component libraries, but that using composition of components avoids programming and allows dynamic modification of the application. However, to have an appropriate collection of reusable components at the right granularity requires a maturing of the framework via an application of the transformation patterns.

JViews is rapidly maturing into a Black Box framework. Much of a JViews-based application can be constructed by composition of stable, fine-grained components, most particularly the user interface components [2]. However, there are still parts of the system that require programming as opposed to composition, and the most common approach to implementing these is via inheritance still. However, there is considerable tool support (see later) to assist with this. We have obtained enormous benefits from adopting a component approach, most notably that of being able to dynamically modify and extend JViews-based environments by end user addition of components. As a result, JViews-based environments are highly configurable, and task automation agents can be rapidly constructed [2].

6 Transformation patterns

This collection of patterns guides the evolution of the framework from the initial raw White Box architecture to the more mature Black Box architecture. These patterns identify and exploit commonalities in code and shift the basis of the architecture from inheritance to composition.

6.1 Component library

The **context** for this pattern is that you are developing the second and subsequent examples based on the white box framework.

The **problem** is that similar objects must be implemented for each problem the framework solves and so how do you avoid writing similar objects for each instantiation of the framework.

The **forces** involved are:

- Bare-bones frameworks require much effort to reuse, while things that work out of the box are much easier. A good library of concrete components makes a framework easier to use
- Its hard to tell initially what components will be reused. Some will be problem specific - some will be reused most times

The solution is to start building a simple library of concrete components and add extra ones as you need them. Add all components initially and later remove ones that never get reused. These are still useful as they give examples of how to use the framework.

In JViews, many concrete classes were implemented for use in SPE, particularly for the graphical icons. Many of these were adapted or generalised for use in MViewsER using programming by difference or other adaptation mechanisms. Gradually a stable underlying collection of reusable classes developed. Classes that were successfully reused from the initial concrete applications included those supporting event histories, view editing and change description representation and management.

At the time the framework was re-implemented in Java, the opportunity was taken to trim out some of the little used classes to avoid the overhead of their re-implementation.

6.2 Hot Spots

The **context** for this pattern is that you are adding components to the component library.

The **problem** is that as you develop applications similar code gets reused over and over again. These code locations are called “hot spots”. How do you eliminate this similar code?

The **forces** are:

- If changeable code is scattered it’s difficult to trace and change
- If changeable code is in a common place flow of control can be obscure

The **solution** is to separate code that changes from code that doesn’t, encapsulating the changing code in objects. Composition can then be used to select the appropriate behaviour rather than having to subclass. Appropriate design patterns are used to encapsulate changes.

Hot spots were commonly found when applying the initial JViews white box framework. Noticing these caused a reorganisation of the code to localise the impact of the hot spots. Much of the code for managing change descriptions was originally distributed. In many cases the code could be isolated into the relationship classes, ie the classes implementing the edges in the CPRGs. Once this isolation occurred various categories of behaviour were observed to be common, leading to the development of a collection of generic relationship classes. Each class in the collection represented a different type of behaviour commonly needed when linking CPRG nodes together. The differences were primarily related to the ways in which the links managed change description propagation. Thus relationship specialisations for aggregation, inter-component attribute dependency, and multiple views (view-of) were rapidly developed and added to the component library. As other examples, the Command pattern was used extensively to encapsulate menu interaction behaviour and Factory Methods and Abstract Factories were used to encapsulate CPRG component generation.

6.3 Pluggable Objects

The **context** for this pattern is that you are adding components to your component library.

The **problem** is that most of the subclasses differ in trivial ways (eg only one method overridden). How do you avoid having to create trivial subclasses?

The **forces** involved are:
- New classes increase system complexity
- Complex sets of parameters make classes difficult to understand and use

The **solution** is to design adaptable subclasses that can be parameterised with messages to send, code to evaluate, colours to display, buttons to hide, etc. Use state variables to represent the differences between individual instances.

This pattern was commonly applied in the early development of JViews. The relationship classes, again provide an example of this. Many of the generic relationship classes could be reused directly, with initialisation parameters used to specify the individual usage differences. For example the generic inter-component attribute dependency relationship was parameterised with the names of the parent and child attributes that were to be kept consistent. A larger example arose when JViews was re-implemented in Java. At that time, many individual classes were collapsed together and turned into JavaBean components with settable properties for customisation. Other examples of pluggable objects developed, include those for persistency support, distributed system support, and view consistency management.

6.4 Fine-grained objects

The **context** for this pattern is that you are refactoring your component library to make it more usable.

The **problem** is how far should you go in dividing objects into smaller ones.

The **forces** involved are:
- The more objects in the system the harder it is to understand
- Small objects allow applications to be constructed by composing small objects together so little programming is required

The **solution** is to keep breaking objects into smaller pieces until it doesn't make sense to divide further - ie decide on the "atomic" level for this domain. The **rationale** is that frameworks will ultimately be used by domain experts so tools will be developed to compose objects automatically. Hence, it's more important to avoid programming than to avoid lots of objects.

In JViews, graphics components were initially "large" representing, say, an entire class icon. These were extensively decomposed so that more generic components for individual lines, boxes, text boxes, and, more interestingly, various types of positional constraint, etc were developed. These supported the design of new types of GUI element by composition. However, this level of refinement necessarily went hand in hand with the development of BuildByWire [11], a GUI element construction tool that could be used to visually compose GUI elements together, and supported libraries of pre-composed elements. Without this tool the level of granularity would have been too fine as the code needed to program the composition, and the complexity caused by the shear numbers of component used would have outweighed any advantage gained by adopting the pure composition approach.

7 High–level tool patterns

The following two patterns describe the evolution of tools to support rapid use of the framework, by providing assistance with the composition of applications using the fine grained components in the, now, black box framework, and tools to support understanding the execution behaviour of applications constructed using the framework.

7.1 Visual builder

The **context** is that you have a black box framework and applications are constructed by composing objects. Behaviour is now determined entirely by interconnection of components. Application is now in two parts:
- Script to connect parts and turn them on
- Behaviour of parts (provided by framework)

The **problem** is that the connection script is very similar between applications. How do you simplify its construction?

The forces involved are:

- Compositions are complex and difficult to understand
- Building tools is costly, but domain experts don't want to be programmers

The **solution** is to construct a visual language and environment to construct the script. This generates the code for the application.

For JViews we have developed a number of visual tools, illustrated in Fig. 4, to assist in composing components and generating applications. JComposer, is a tool that allows high level specification and generation of CPRG components [12]. It comprises several visual languages. One is a language for specifying components and relationships, that is similar in approach to UML class diagrams. Another is a data flow based visual language for specifying dynamic behaviour, by composing CPRG components and prepackaged filter and action components, linked together by event flows, where the events are encapsulated as CPRG change descriptions. These two tools allow visual specification and generation of the bulk of the base and view layers of a JViews application. Further programming is, however, needed to compete the application. This is handled by generation of two classes for each component, the first contains many method stubs, and is intended to be user modified. This inherits from the second, which contains the bulk of the generated code. This separation allows regeneration of parts of the environment, while retaining the hand coded modifications, another application of the Hot Spot pattern.

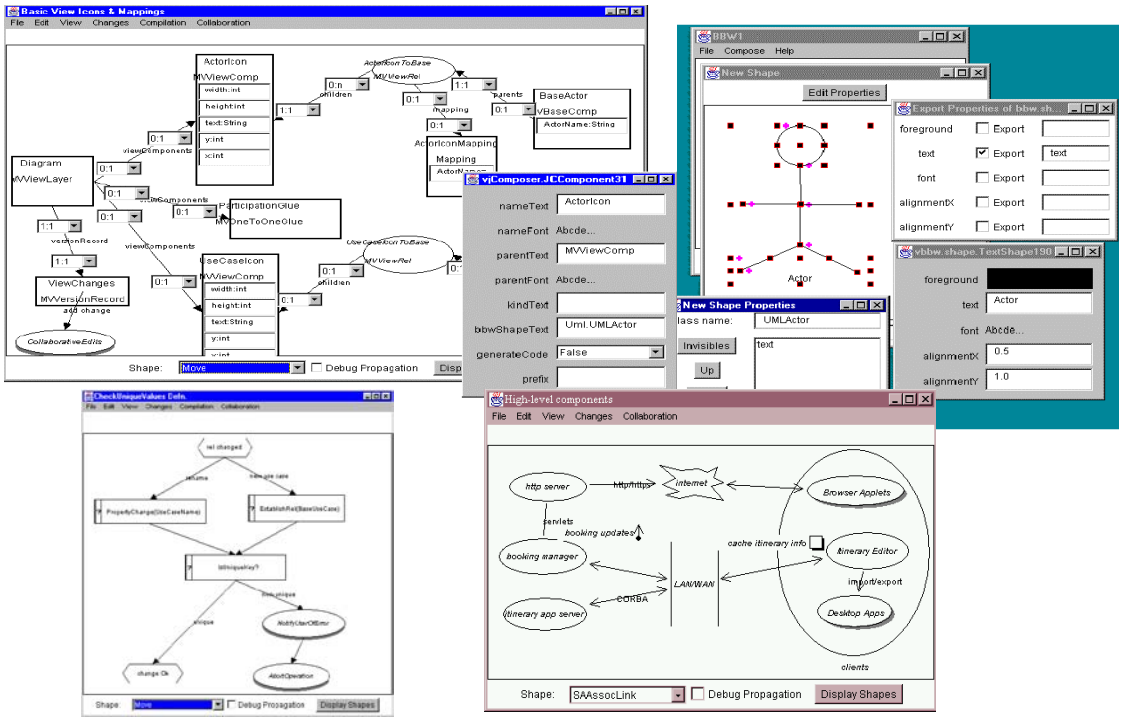

Figure 4: JComposer tool specifying components of a CPRG (left top) and a task automation agent (left bottom), BuildByWire specifying a UML Actor icon (right top), and SoftArch specifying a high level architectural view (right bottom)

BuildByWire is a tool for visually composing elements for use in the display layer of a JViews application [2,11]. This very strongly adopts the composition metaphor, with discrete user interface elements being composed together using constraint components as the "glue". The visual elements are constructed as reusable JavaBean components. The tool also allows construction of the view editors, generating panels that allow end users to instantiate and manipulate instances of the visual components. The tool also provides assistance in constructing a composite property sheet (allowing attributes of aggregated elements to be exposed or hidden, renamed, etc) and CPRG API. The latter allows the components to be incorporated into JComposer specifications, permitting display and view layers to be visually composed together. Recently, we have added another visual tool, SoftArch, which allows high level architectural specification of JViews based environments [2]. This supports refinement to JComposer-based CPRG component specifications.

Development of these visual tools, particularly BuildByWire, went hand in hand with application of the transformation pattern, particularly Fine Grained Objects and Pluggable Objects. Without the motivation of construction of the visual tools, the decomposition of components into fine-grained objects would not have occurred. The complexity of scripting was definitely the motivation for development of JComposer. Much of the work in connecting together elements of a CPRG is repetitive and mundane, but sufficiently different each time that parameterisation approaches were inadequate and a more generative approach required.

7.2 Language Tools

The **context** for this pattern is that you have created a builder

The **problem** is that visual builders create complex composites. How do you inspect and debug these

The **forces** that exist are:

- Existing tools are inadequate as they don't provide information at the right abstraction level
- Building good tools takes time

The **solution** is to create specialised debugging and inspection tools.

The **rationale** here is that because the atomic building blocks of your application are now the pluggable object and fine-grained components of your black box framework, you need tools that allow you to visualise application behaviour in terms of these abstractions.

We have aggressively applied this pattern in all of our visual language work. Our strong feeling is that as you are applying the Visual Builder pattern, you should concurrently apply the Language Tools pattern to construct visualisation tools to match your visual construction tools. In our case, we have found that a desirable approach in constructing such language tools is to attempt as much as possible to reuse the visual abstractions adopted in the corresponding visual builder. Our JVisualiser tool (Figure 5, left) uses a similar notation to the JComposer CPRG specification tool, replacing generic component icons with actual instances. One advantage of developing pluggable objects is that our visualisation tools can often also be used to dynamically extend an environment by "plugging in" precomposed components. The event flow notation is used to link these new components with existing ones, allowing dynamic generation of new software agents. These may be used to monitor behaviour (eg compile statistics, record certain types of changes, etc) or to extend the environment with new user-specified task automation agents. The SoftArch tool also has a visualisation tool that uses the design refinement links to allow high level architecture components to capture and visualise in appropriate ways, eg using colour to indicate frequency of method calls, events associated with components implementing them (Figure 5 right).

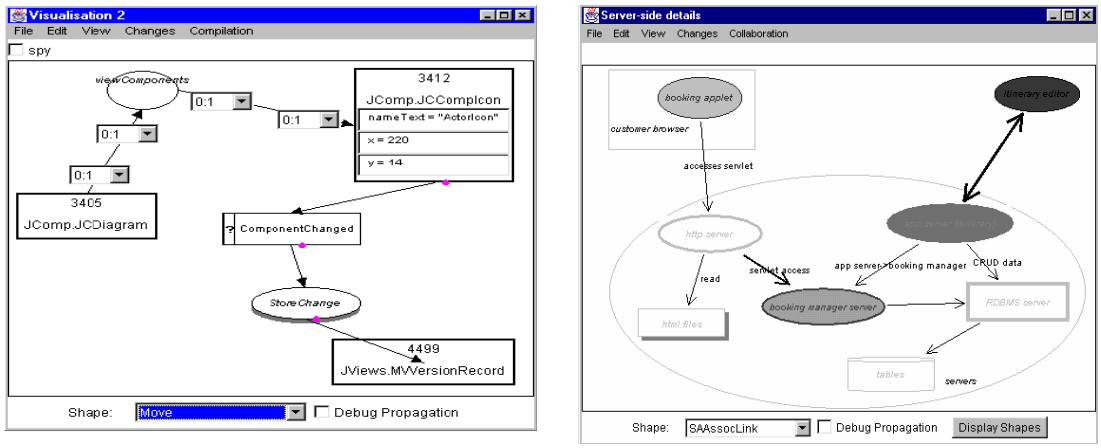

Figure 5: JVisualiser used to visualise and extend a JViews application (left) and SoftArch visualising execution behaviour of an application at an abstract level (right)

8 Critique and Extension of Evolving Frameworks

It is clear from the previous sections that EFPL fits extremely well with the development of the JViews framework. All patterns were applicable and the contexts, forces, solutions and rationale entirely appropriate.

From experience with development of JViews and our understanding of the development of other frameworks, we believe that a number of extensions to EFPL can be made that others may find useful. We note that Ruping [13] has developed a pattern language for developing a framework in parallel with the first application, which could also be considered to be a variant or extension of EFPL. The extensions we propose here are the addition of an additional transformation pattern "Changing Implementation Platform", together with two patterns related to integration of applications developed using the framework with other applications, "Combining applications" and "Integrating third party applications". A final suggested extension is "Reflective Framework", an architecture pattern.

8.1 Changing implementation platform

The **context** for this pattern is that you have constructed a white box framework and have been applying the transformation patterns to partially develop a black box framework.

The **problem** is that the framework is implemented using a software platform that is unacceptable to a large fraction of potential end users.

The **forces** are:

- Because it takes time for a platform to mature, an appropriate choice of implementation platform when you commence may no longer be fashionable or appropriate as the framework matures.
- Significant new languages/platforms arise on a 7-10 year cycle. Frameworks may take many years to mature meaning the chances of a fashionable new platform arising as a framework matures are high.
- A language or platform suitable for rapid prototyping may have been used to develop the initial framework implementation. This may cause performance or accessibility problems as the framework matures.
- Another platform may have technology or ideas that would be desirable to leverage off in further development of the framework.
- Re-implementing a framework on another platform is extremely expensive.
- Supporting multiple implementation platforms is expensive.

The **solution** is to re-implement the framework on a more appropriate or accessible platform.

The **rationale** is that a framework needs end users to make the cost of development worthwhile. If a large fraction of potential end users consider the platform to be inappropriate, the framework will die through lack of adoption, or its growth may be severely limited. If this is the case, the relative cost of shifting platforms must be weighed against the opportunity cost of not shifting.

Implementation of this pattern is a time consuming and expensive operation. It is, however, an opportunity to concurrently apply many of the other transformation patterns, as the cost of implementing them is amortised into the cost of re-coding the framework as a whole, and may reduce that cost. For example, weeding unused components out of the component library, finding appropriate solutions for Hot Spots, and creating pluggable components may all reduce the overall cost of re-coding the framework by reducing the amount of code to translate. The need to migrate platforms may well overcome past inertia against making some of these changes. Facilities available in the new platform, notably graphics support, may well spur the development of Fine Grained Objects.

Examples

JViews was originally developed (as MViews) using a bespoke Object Oriented Prolog developed by ourselves and limited to Macintosh OS. This platform was very convenient for rapid prototyping, but clearly limited the potential audience for our framework. As we scaled up the size of application using the framework, this platform began causing performance problems. For this reason, we re-implemented the framework in C++ and then Java. The time cost of implementing the C++ version meant this never completed. At the time the Java implementation was done, we took the opportunity to make many

changes to minimise the amount of code that required translating. At the same time we exploited the, at that time new, JavaBeans technology to create large numbers of pluggable components. Following conversion to C++ the original implementation was no longer supported.

Brad Myers' group developed Garnet, a user interface development framework, implemented in Common Lisp [14]. This platform caused similar, though less severe, benefits and problems to those we faced with our initial implementation platform. For this reason, Amulet, a C++ framework that "incorporates the best ideas of Garnet" was developed [15]. As part of this development, code associated with individual windowing platforms was isolated and a portable graphic events manager (GEM) developed [15]. Similarly commands were reformulated as Command Objects. Both are examples of Hot Spot application during the migration.

The Standard Template Library (STL) is a C++ framework for data structures [16]. With the advent of Java, there was strong pull from the Java community for an equivalent framework in Java. This led to the development of the Java Generic Library (JGL) [17], a translation of STL into Java. At the time of translation, changes were made to STL to leverage off new technology in Java, such as the built-in iterator classes.

Related patterns are all of the EFPL transformation patterns and White Box and Black Box Framework patterns.

8.2 Combining applications

The **context** for this pattern is that you have developed a white box framework and have started applying transformation patterns to develop a black box framework. You have developed a variety of applications using the framework.

The **problem** is that you wish to integrate together two or more applications developed using the framework.

The **forces** involved are:

- Applications take time to develop, so reusing them by integrating them together makes sense
- From a user perspective an apparently tight integration, with shared user interface and common look and feel is desirable.
- Architectural choices in the early development of the framework will probably mean that tight integration is difficult due to name space conflicts, inadequate componentisation of the user interface, etc.

The **solution** is to modify the framework architecture to allow multiple applications developed using the framework to interact with one another in a seamless fashion.

The **rationale** is that when initially developing a framework it is difficult enough to consider the abstractions required for managing one application, let alone factoring in the need for supporting multiple interacting applications in a seamless manner. However, the need to do so will rapidly become compelling, so architectural changes will be required to make this integration possible.

Implementation of this pattern may involve considerable work. One type of area that is commonly affected is persistency or object indexing mechanisms, where inadequate support for multiple name spaces may be an issue. Another is user interface integration and control, where development of Pluggable Objects and application of the Hot Spot and Fine Grained Object patterns may be needed to address inappropriate replication of code or inadequate separation of functionality.

Examples

Having developed the SPE and MViewsER applications, we immediately saw the benefit in combining these to create an integrated design environment supporting multiple modelling paradigms. To do this, however, required changes to the object persistency mechanism, which had a single name space, and also the development of the fourth, coordination layer to the JViews framework. This latter change allowed change descriptions to be routed between separate CPRGs. Extension of the change description response mechanism to allow change descriptions to be intercepted both before and after taking affect allowed sophisticated control integration to be simply effected [8]. This approach was

developed considerably as the framework matured; for example we have been developing pluggable objects that support collaborative work activities in a seamless and transparent fashion [18].

The MetaMOOSE framework for constructing CASE tools, similar in application domain to JViews, originally only supported one application, Subsequent development of the framework added a shared repository to allow multiple applications generated by the framework to interact with one another [19]. The C2 user interface development framework, which supports distributed inter-application messaging support, was an evolution of the Chiron-1 framework which lacked such support [20].

Related patterns are all of the EFPL transformation patterns and White Box and Black Box Framework patterns. The pattern is related to "Third Party Integration", which provides an alternate approach to solving the problem.

8.3 Integrating third party applications

The **context** for this pattern is that you have developed a white box framework and have started applying transformation patterns to develop a black box framework. You have developed a variety of applications using the framework.

The **problem** is that you wish to integrate an application developed using the framework with a third party tool.

The **forces** involved are:

- Applications take time to develop, so reusing them by integrating them together makes sense
- From a user perspective an apparently tight integration, with shared user interface and common look and feel is desirable.
- The framework has been designed to produce stand-alone applications without thought to integrating other tools.

The **solution** is to construct an API for integrating third party tools.

The **rationale** is very similar to that of "Combining Applications. When initially developing a framework it is difficult enough to consider the abstractions required for managing one application, let alone factoring in the need for interacting with other applications in a seamless manner. However, the need to do so will rapidly becomes compelling, so architectural changes will be required to make this integration possible.

Implementation of this pattern can be done in many ways, depending on the infrastructure available in the implementation platform and the needs of the tools being integrated. A common approach is to use the Wrapper pattern to develop an API. The advent of component technology has made it common to either construct applications developed within the framework as components, or wrap external tools as components and use standard component interaction mechanisms to mediate the integration. Because implementation relates strongly to the external environment, this pattern may be applied several times as the framework and external infrastructure evolve.

Examples

Having developed several applications using JViews, we realised that interaction with third party tools, such as word processors, spreadsheets, etc, was desirable. We used Wrappers to implement and convert change descriptions into event messages sent via the underlying OS scripting and messaging architecture to the external tools. When converting JViews to Java, we exploited the newly available JavaBeans technology to turn JViews applications into JavaBeans. Field [21], developed from Pecan [22] used wrappers, together with a proprietary message bus to integrate tools together. Oz [23], developed from Marvel [24], also used Wrappers to integrate other tools.

Related patterns: this pattern is in may ways a specialisation of Pluggable Objects. It commonly uses the Wrapper pattern for implementation. It is closely related to "Combining Applications" and may provide another implementation approach for that pattern's problem.

8.4 Reflective framework

The **context** is that you have a mature black box framework. A Visual Builder and Language Tools have been developed.

The **problem** is that while constructing applications using the framework is fast, due to the visual builder and language tools, extending the framework is slow as it involves static code changes that are typically done by develoepers.

The **forces** are:

- Static coding takes time.
- Because the framework has many fine-grained objects, extension of the framework is often by composition or parameter setting.
- Pluggable Objects can be used, but only to a limited degree as their configurability is limited
- Users of an application developed with the framework want to extend the application's set of modeling abstractions and constraints

The **solution** is to provide a user-editable meta-model that can be used to dynamically extend the framework.

The **rationale** is that the pressures here are very similar to the pressures that led to constructing a Visual Builder, but where the modification is to the framework itself (probably including its tools) rather to an application generated using the framework. It's natural in this case to develop a Knowledge level–operational level split, as outlined by Fowler [25] and construct a meta model to describe the framework and its environment. By allowing this meta model to be visualised and edited, probably using a visual tool, the framework can be modified. As composition of pluggable objects is by now the most common way of extending or modifying the framework, the changes can be effected dynamically.

Implementation can be by a variety of means, but the Dynamic Object Model pattern [26] is a useful approach to take. This in turn builds on Fowler's Analysis Patterns.

Examples
The SoftArch modelling tool we developed as one of the Visual Builders for our framework has a visually editable meta-model. This allows the architecture modeling abstractions and constraints to be modified dynamically. Our other Jviews environments have fixed meta-models (generated by JComposer) which can only be modified by code re-generation and recompilation. Rhiele et al [26] describe several other examples as part of their description of their Dynamic Object Model pattern, including the business model of Argo, the framework used in Dynamo 1 and 2 and the EbXML framework.

9 Summary and conclusions

This paper has provided a verification of the usefulness and effectiveness of the Evolving Frameworks Pattern Language, by calibrating it against experience in developing the JViews framework. As a result of this experience we propose a number of extensions to the pattern language capturing other common activities that occur in framework development.

As mentioned in the introduction, this paper is unusual in being an experience paper, both critiquing and extending an existing pattern language. We feel that it would be appropriate for the patterns community to develop a pattern language describing an appropriate structure for such a paper. This paper could usefully be used as an example in abstracting that pattern language. In writing the paper, we drew on the pattern writing pattern language of [3]. We adapted this by summarising the running example in more depth, but abbreviating the descriptions of each of the patterns in the pattern language. An additional activity critiqued the pattern language and proposed extension patterns in a standard pattern manner.

Acknowledgements

Support for parts of this work has come from the University of Auckland Research Committee and the New Zealand Public Good Science Fund.

References

1 Roberts, D., Johnson, R., Evolving Frameworks A Pattern Language for Developing Object-Oriented Frameworks, http://st-www.cs.uiuc.edu/users/droberts/evolve.html

2 Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences, Journal of Information and Software Technology: Special Issue on Constructing Software Engineering Tools, Vol. 42, No. 2, January 2000, pp. 117-128.

3 Meszaros G., and Doble, J., A Pattern Language for Pattern Writing, http://hillside.net/patterns/Writing/pattern_index.html

4 Beck K. *Smalltalk Best Practice Patterns — Volume 1: Coding*. Prentice-Hall, 1996.

5 Durham A., Johnson R. A Framework for Run-time Systems and its Visual Programming Language. In *Proceedings of OOPSLA '96, Object-Oriented Programming Systems, Languages, and Applications*. San Jose, CA. October 1996.

6 Grundy, J.C., and Hosking, J.G., The MViews Framework for Constructing Multi-view Editing Environments, *New Zealand Journal of Computing*, Vol. 4, No. 2, 1993, 31-40.

7 Grundy, J.C., and Hosking, J.G., Mugridge, W.B., Supporting flexible consistency management via discrete change description propagation, *Software - Practice and Experience*, Vol. 26, No. 9, September 1996, Wiley, 1053-1083.

8 Grundy, J.C., and Hosking, J.G., Mugridge, W.B., Inconsistency management for multiple view software development environments, IEEE Transactions on Software Engineering: Special Issue on Inconsistency management in software development, Vol. 24, No. 11, November 1998, IEEE CS Press, pp. 960-981.

9 Grundy, J.C., Venable, J. Providing Integrated Support for Multiple Development Notations, in Proceedings of *CAiSE '95,* Finland, June 1995, Lecture Notes in Computer Science 932, Springer-Verlag, pp. 255-268.

10 Johnson R, Foote B. Designing Reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

11 Mugridge, W.B., Hosking, J.G. and Grundy, J.C. Drag-throughs and attachment regions in BuildByWire, In Proceedings of OZCHI'98, Adelaide, Australia, Dec 1-4 1998, IEEE CS Press, pp. 320-327.

12 Grundy, J.C. and Hosking, J.G. High-level Static and Dynamic Visualisation of Software Architectures, accepted to 2000 IEEE Symposium on Visual Languages, Seattle, Washington, Sept. 14-18 2000, IEEE CS Press.

13 Rüping, A. Building Frameworks and Applications Simultaneously, *Proceedings PLOP 2000*, Washington University Technical Report number: wucs-00-29.

14 Myers, B.A. Giuse, D., Dannenberg, R.B., Zanden, B.V., Kosbie, D., Pervin, E., Mickish, A., and Marchal, P. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. IEEE Computer, Vol. 23, No. 11, November, 1990.

15 Myers, B.A., McDaniel, R.G., Miller, R.C., Ferrency, A.S., Faulring, A., Kyle, B.D., Mickish, A., Klimovitski, A. and Doane. P. The Amulet Environment: New Models for Effective User Interface Software Development, *IEEE Transactions on Software Engineering*, Vol. 23, no. 6. June, 1997. pp. 347-365.

16 Ammeraal, L. STL for C++ Programmers, by. John Wiley, 1996. ISBN 0-471-97181-2.

17 ObjectSpace Inc. http://www.objectspace.com/products/voyager/libraries.asp

18 Grundy, J.C., Hosking, J.G., Mugridge, W.B., Apperley, M.D. A decentralised architecture for software process modelling and enactment, IEEE Internet Computing: Special Issue on Software Engineering via the Internet, Vol. 2, No. 5, IEEE CS Press, September/November, 1998, pp. 53-62.

19 Furguson, R.I., Parrington, N.F., Dunne, P., Archibald, J.M. and Thompson, J.B. MetaMOOSE – an Object-oriented Framework for the Construction of CASE Tools, *Proceedings of CoSET'99*, Los Angeles, 17-18 May 1999, University of South Australia, pp. 19-32.

20 Taylor, R.N. Medvidovic, N., Anderson, K.M. Whitehead, E.J. and Robbins, J.E. A Component- and Message-Based Architectural Style for GUI Software, In *Proceedings of the Seventeenth International Conference on Software Engineering (ICSE17)*, Seattle WA, April 24-28, 1995. pages 295-304.

21 Reiss, S.P., "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, vol. 7, no. 7, 57-66, July 1990.

22 Reiss, S.P., "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, vol. 11, no. 3, 276-285, 1985.

23 Ben-Shaul, I.Z., Heineman, G.T., Popovich, S.S., Skopp, P.D. amd Tong, A.Z., and Valetto, G., "Integrating Groupware and Process Technologies in the Oz Environment," in *9th International Software Process Workshop:The Role of Humans in the Process,* IEEE CS Press, Airlie, VA, October 1994, pp. 114-116.

24 Barghouti, N.S. 1992. Supporting Cooperation in the Marvel Process-Centred SDE. In *Proceedings of the 1992 ACM Symposium on Software Development Environments,* ACM Press, 1992, pp. 21-31.

25 Fowler, M, 1997, Analysis Patterns: Reusable Object Models, Addison Wesley.

26 Riehle, D., Tilman, M. Johnson, R.: Dynamic Object Model, *Proceedings PLOP 2000*, Washington University Technical Report number: wucs-00-29.