Applying the Evolving Frameworks Pattern Language to Multi-View Software Design Tools

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

Abstract

The *Evolving Frameworks Pattern Language* (EFPL) describes the development of software frameworks over time. Based on our experience of the evolution of the JViews framework over almost ten years, we verify in this paper the ability of the EFPL to describe the evolutionary development process of our framework. JViews is a mature framework for constructing visual and textual editing environments that support multiple views, notations and users.

We have found the EFPL patterns describe the evolution of our framework well. However, as some of our experiences (and that of others) are not covered by the existing patterns, we propose additions to the pattern language. We hope our experiences with the framework, and the patterns, will be useful to developers of software design tools, frameworks and pattern languages.

Keywords: Patterns, Frameworks, Integrated Environments

1 Introduction

The *Evolving Frameworks Pattern Language* (EFPL) [1] is a collection of inter-related patterns that describe the development of large-scale software frameworks over time. According to the EFPL, a typical software framework is initially formed from three example applications, whose commonalities are generalised into an initial white-box, reusable framework. Evolution of the framework typically results in movement to a black-box approach (using aggregation rather than inheritance to use the framework classes). Further enhancements usually include the development of reusable component libraries, identification of reuse "hot spots", and the use of fine-grained objects in the framework. Visual builders and language tools to support generation of classes using the framework may be longer-term developments. Each of these evolutionary characteristics is captured in the EFPL as a related Design Pattern.

In this paper we describe the application of the EFPL to the evolution of our JViews framework [2]. We developed JViews over many years as a framework for the construction of multiple view, multiple user editing environments. It has undergone many extensions and enhancements, and is now a large, mature framework. Hence it is an ideal candidate to calibrate the EFPL against. Our aim here is to both validate that the EFPL captures well the evolution of complex software frameworks, and to propose the addition of several new patterns to the EFPL based on our experience with JViews.

The following section describes the Evolving Frameworks Pattern Language, including a categorisation of its patterns. We then introduce the JViews framework, giving an overview of its evolution. The bulk of the paper examines each of the patterns in EFPL and their applicability to our experiences in evolving JViews. We then discuss areas where EFPL could be improved, based on our own work with JViews and the reported results of others. This leads us to propose a number of new patterns for inclusion into EFPL. We finish with conclusions and future work.

2 The Evolving Frameworks Pattern Language

Roberts and Johnson [1] introduce software frameworks as "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". They developed the Evolving Frameworks Pattern Language (EFPL) to describe the possible evolution of such frameworks. The 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".

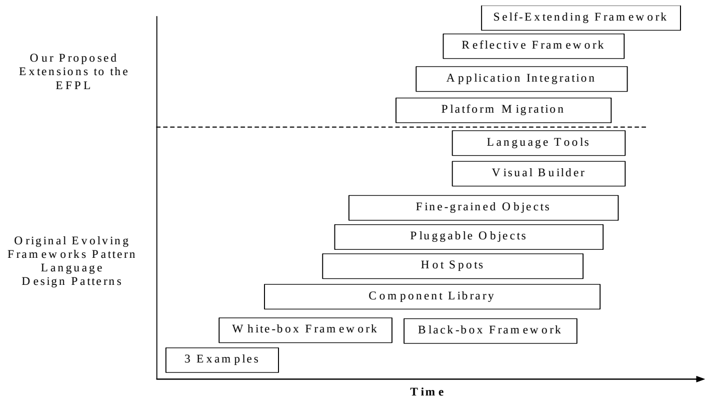

Fig. 1: Patterns in the Evolving Frameworks Pattern Language.

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

- An initiator pattern, called "Three examples", which describes the genesis of a framework via generalisation 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", and "Fine Grained Objects", which describe how the structure of a framework changes and matures over time, improving the framework's reusability, maintainability and generality.
- High level tool patterns, "Visual Builder" and "Language Tools", describing evolution of high level generation and debugging tools to simplify construction of applications.
- Our suggestions for EFPL extensions, including "Platform Migration", "Application Integration", "Reflective Framework" and "Self-extending Framework". These further extend a framework's capabilities and follow from our experience with JViews evolution.

To illustrate the pattern language, Roberts and Johnson use as a running example the evolution of the Model-View-Controller (MVC) framework [3], supplemented by examples from the Runtime Systems Expert framework [4].

3 The JViews Framework

The JViews framework has evolved over 10 years [2], [5], [6], [7]. It aims to support the design and implementation of visual environments that support multiple views with different representations and inter-view consistency. 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.
- Visual representation and manipulation of elements in the views.

The underlying abstraction of JViews is the Change Propagation and Response Graph (CPRG) [6]. Each significant object in the environment is represented by a graph component. Components have attributes representing state. Graph edges represent relationships between components, and are themselves components. Changes to components or the graph structure are captured as 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 supports the construction of three-layer applications based on CPRGs. Fig. 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.

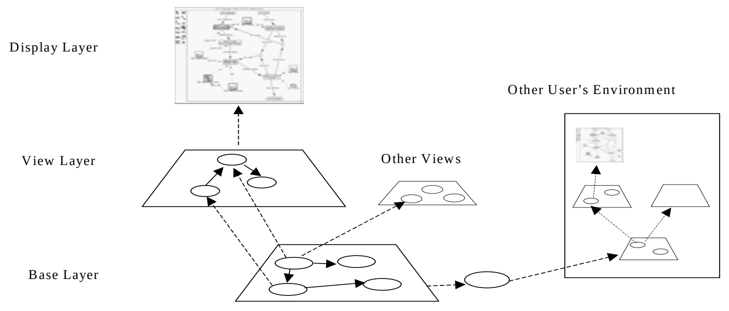

Fig. 2: JViews 3 layer architecture

Fig. 3 shows an application that was developed using an early version of JViews [7]. This integrates two tools, SPE and Serendipity. SPE is a CASE tool that supports editing of class diagrams for the visual design of object oriented programs (1), together with textual class implementations generated from, and kept consistent with, the visual designs (2). Serendipity is a workflow and process modelling tool (3). This allows specification and implementation of complex hierarchical process models, including the roles and artefacts that are applicable to various stages in a process. In the combined environment, Serendipity process models can be enacted. Changes made in SPE are then recorded against the currently active process stage (4).

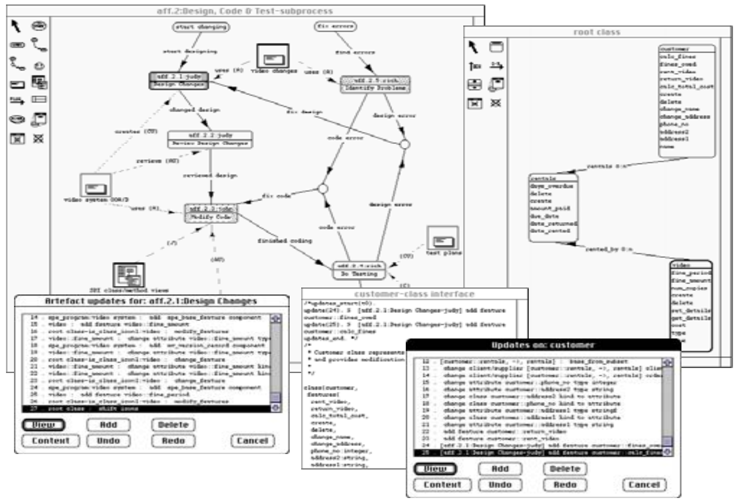

Fig. 3: SPE/Serendipity integrated environment.

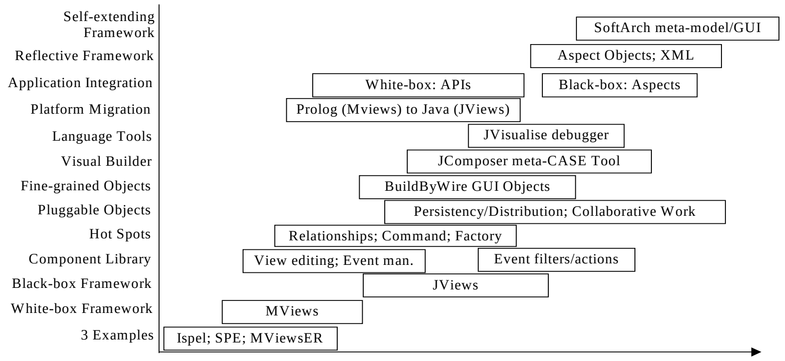

Fig. 4. An overview of the evolution of the JViews framework.

In the following sections, we relate the evolution of JViews to the EFPL design patterns. We have adapted the pattern writing pattern language of [8] to do this. In each section, we briefly summarise an EFPL pattern (indented) and detail our relevant experience with the development of JViews. For the complete description of the EFPL patterns, the reader is referred to [1].

To give a context for much of the discussion in the following sections, Fig. 4 outlines the evolution over time of the JViews framework. It shows some of the evolutionary milestones of JViews against the EFPL patterns (along with our additional four patterns) on the vertical axis.

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 matches exactly the initial development of JViews. Initially we developed a multiple view class-diagramming tool, Ispel [5]. From this we abstracted a general framework, MViews (subsequently JViews). We used MViews to develop an early version of SPE, combining diagrammatic and textual programming [5]. To demonstrate the generality of the framework and to extend it further, we developed an entity-relationship modelling tool, MviewsER. This supports visual ER diagrams and textual relation specifications, with consistency maintained between the two types of view [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, the second extending the first to provide better textual modelling support. The third application (the ER tool) was different in nature to the first two, supporting ER rather than 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. The initial application, Ispel, was developed as a Masters thesis project. Tackling a larger development would have been impossible in the time available. Each subsequent application formed natural material for a research paper, a suitable payoff in the academic domain.

5 Architecture patterns

EFPL introduces two architectural patterns, the “White Box Framework” and the “Black Box Framework”. 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. This formed our initial white-box framework, MViews, implemented in an OO-Prolog [5].

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). The APIs couldn’t readily be frozen. At this point, 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. This is as recommended in the implementation section of the White Box Framework pattern.

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.*

In a black-box framework, composition of provided software components can reduce or eliminate programming effort. Dynamic modification of an application based on such a framework is enabled through adding or replugging such reusable components [10]. Categorisation can help organise components so that the right ones can be found. However, to have an appropriate collection of reusable components at the right granularity requires a maturing of the framework through application of the transformation patterns of the next section.

Our second-generation framework, JViews, has rapidly matured into a Black Box framework [2]. Much of a JViews-based application can be constructed by composition of stable, fine-grained components. We have obtained enormous benefits from adopting a component approach. Most notably, we are 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].

There are still parts of the system that require programming as opposed to composition; the most common approach to implementing these is through inheritance. However, there is considerable tool support to assist with this (see Section 7).

6 Transformation patterns

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

Many concrete classes were implemented in MViews for use in SPE, particularly for graphical icons and event handling code. Many 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 event representation and handling. At the time the framework was re-implemented in Java to produce JViews, the opportunity was taken to trim out little used classes to avoid the overhead of re-implementation. We also began developing a reusable library of event handling building blocks ("filters and actions") [2] to aid construction of common event-handling approaches.

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 using the initial MViews white box framework. Identifying these caused a reorganisation of the code to localise impact of the hot spots. For example, the Command pattern was used extensively to encapsulate menu interaction behaviour and Factory Methods and Abstract Factories to encapsulate CPRG component generation.

Much of the code for managing event handling was originally distributed. We found that much of this code could be moved into the relationship classes, which implement the edges in the CPRGs. Common categories of behaviour were then observed, primarily related to change description propagation. Refactoring led to several generic relationship classes, each representing a common type of behaviour that was needed when linking CPRG nodes together. Relationship specialisations for aggregation, inter-component attribute dependency, and multiple views (view-of) were rapidly developed and added to the component library.

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 differences between individual instances.*

This pattern was commonly applied in the early development of MViews. 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.

When MViews was re-implemented as JViews, using Java, many individual classes were collapsed together and turned into JavaBean components with settable properties for customisation. Other examples of pluggable objects include those for persistency support, distributed system support, and view consistency management.

A more recent use of this pattern was in the development of plug-and-play collaborative work components [11]. These were added to JViews to allow users to choose various collaborative work facilities, collaborative editing, highlighting, view element locking, messaging and shared versioning, to plug into a JViews-based system at run-time. This proved a successful approach. It improved greatly on the earlier MViews-based equivalents, which were hard-coded into MViews framework classes, included whether they were needed or not [12]

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 MViews, graphics components were initially "large", representing, say, an entire class icon. These were extensively decomposed when developing JViews 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 [13], 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. One type of tool supports the composition of applications using the fine grained components in the, now, black box framework. Another type of tools supports understanding of 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 for JViews, to assist in composing components and generating applications, as illustrated in Fig. 4. JComposer is a tool that allows high level specification and generation of CPRG components [14]. It comprises several visual languages. One is for specifying components and relationships, similar in approach to UML class diagrams. A data flow based visual language is used for specifying dynamic behaviour, by composing CPRG components and prepackaged filter and action components. These are linked together by event flows, where the events are encapsulated as CPRG change descriptions.

JComposer enables the visual specification and generation of the bulk of the base and view layers of a JViews application. Further programming is needed, however, to compete the application. This is handled by the generation of two classes for each component. The first, abstract class contains the bulk of the generated code and is not altered by the programmer. The second inherits from the first class and contains method stubs; once modified, it is not regenerated. This separation allows regeneration of parts of the environment, while retaining the hand coded modifications, another application of the Hot Spot pattern.

BuildByWire is a tool for visually composing elements used in the display layer of a JViews application [2], [13]. This adopts a composition metaphor, with discrete user interface elements being composed using constraints as the "glue". The resulting visual elements are constructed as reusable JavaBean components, suitable for the CPRG API. This allows them to be integrated with the JComposer views, permitting display and view layers to be visually composed together. The tool generates view editors, which allow end users to instantiate and manipulate instances of the visual components. Composite property sheets are constructed for composite user interface elements (allowing attributes of aggregated elements to be exposed or hidden, renamed, etc).

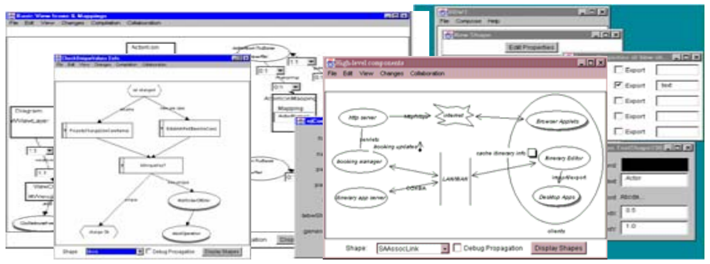

Fig. 4: JComposer tool specifying components of a CPRG (left top) and a task automation agent (left bottom), BuildByWire specifying a UML Actor icon (right top), and SoftArch specifying a high level architectural view (right bottom)

Recently, we have developed another visual tool, SoftArch, which allows high level architectural specification of JViews based environments [15]. 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 patterns, 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 was 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 objects 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 (Fig. 5, left) uses a similar notation to the JComposer CPRG specification tool, replacing generic component icons with actual instances.

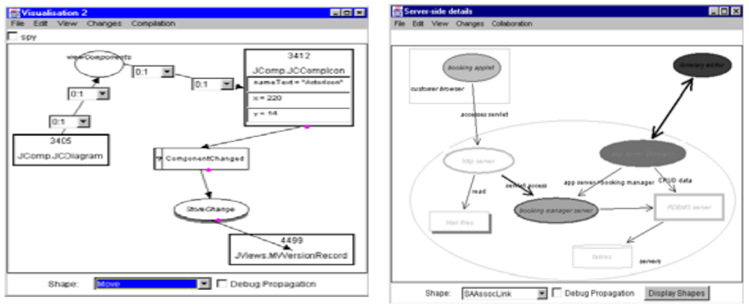

Fig. 5: JVisualiser used to visualise and extend a JViews application (left) and SoftArch visualising execution behaviour of an application at an abstract level (right)

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 (Fig. 5 right).

8 Critique and Extension of Evolving Frameworks

It is clear from the previous sections that EFPL describes the development of the JViews framework well. All patterns were applicable and the contexts, forces, solutions and rationale appropriate. However, some of our experiences were not captured by the pattern language. From our experience with the development of JViews and our understanding of the development of other frameworks, we believe that additions to the EFPL can be made that others may find useful. We note that Ruping [16] has developed a pattern language for developing a framework in parallel with the first application. This can also be considered a variant or extension of EFPL.

We propose the addition of four new patterns to EFPL:

- "Platform migration", an extra transformation pattern
- "Integrating applications", for composing several applications developed with the framework
- "Reflective Framework", an architecture pattern allowing framework components to be self-describing
- "Self-extending Framework", where the framework itself can be run-time extended with new abstractions.

The existing EFPL "Visual builder" and "Language tools" patterns can be used with the third and fourth of these new patterns to provide greater framework reuse support.

8.1 Platform migration

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 (ie implementation language, set of application libraries, and/or operating system API) that is either unacceptable to a large fraction of potential end users or constrains further development of the framework due to performance or technology limitations.

The **forces** are:

- As 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 with advantageous technology features 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.
- By this stage of development the software architecture of the framework has, to a large extent, stabilised and is becoming well understood.
- 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 retaining as much of the underlying design architecture of the original implementation as is possible while using the technology advantages offered by the new platform to improve performance or capability of the framework.

The **rationale** is that a framework needs to be used to make the cost of development worthwhile. If a large fraction of potential framework users (ie software developers using the framework to construct partial or complete applications) consider the platform to be inappropriate, the framework will die through lack of adoption, or its growth may be severely limited. In addition, unpopular or inaccessible platforms are unlikely themselves to be growing and incorporating new technological features thus making the task of extending the framework increasingly more difficult. If this is the case, the relative cost of shifting platforms must be weighed against the opportunity cost of not shifting. As the architecture of the framework has now stabilised, the important structures and mechanisms are well understood at an abstract platform independent level. This means that re-implementation costs can be minimised through reuse of this abstract architecture together with appropriate parts of the framework design.

Although this pattern may on the face of it seem somewhat trite, it encapsulates one of the most important choice points that can be faced in the development of a framework. The decision to re-implement comes about when a number of the forces described above coalesce together and overcome the strong emotional and cost inertia against redevelopment.

Implementation of this pattern can be a time consuming and expensive operation. Reusing as much of the existing architecture and design as possible minimises the redevelopment cost. A possible preliminary step that eases the redevelopment is to refactor the original framework to separate platform specific and platform independent components, using patterns such as Bridge to achieve this. Implementation also provides an opportunity and strong motivation 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 these changes. Facilities available in the new platform, notably graphics support, may spur the development of Fine Grained Objects. Application of these concurrent changes can lead to a period of accelerated evolution of the framework.

One of the most significant **liabilities** in using this pattern is that framework users may be forced to migrate existing code investment to the new platform. This may involve considerable investment on their part and may thus be used as an excuse to stop using the framework.

Examples

JViews was originally developed (as MViews) using a bespoke OO Prolog developed by ourselves and limited to Macintosh OS. This platform was convenient for rapid prototyping, but clearly limited the potential audience for our framework. As we scaled up the size of application using the framework, the platform caused 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. We also exploited the, at that time, new JavaBeans technology to create large numbers of pluggable components. Following conversion to Java the original implementation was no longer supported.

Brad Myers' group developed Garnet, a user interface development framework, implemented in Common Lisp [17]. 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 [18]. As part of this development, code associated with individual windowing platforms was isolated and a portable graphic events manager (GEM) developed [18]. Similarly commands were reformulated as Command Objects. Both are examples of Hot Spot application during migration.

The Standard Template Library (STL) is a C++ framework for data structures [19]. 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) [20], 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. A further Java-based example is the listener model, which is a conversion and adaptation of the SmallTalk MVC framework into the Java platform.

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

8.2 Integrating 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, or integrate third-party applications with an application developed with the framework.

The **forces** involved are:

- Applications take time to develop, so reusing them by integrating simple applications into more complex ones 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 framework has been designed to produce stand-alone applications without thought to integrating other tools.

The **solution** is to modify the framework architecture to allow multiple applications, both those developed using the framework and third-party applications, to interact with one another, via suitable APIs or other abstractions. This involves developing data, control flow, and user interface integration strategies that were not obvious when the framework was initially developed to allow this integration to be performed in a seamless fashion. There is a tension in design here between providing an open API that allows for tight integration and complex interaction between the applications versus the need to ensure integrity of the framework.

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 rapidly becomes compelling, so architectural changes are required to make integration possible.

Implementation of this pattern may involve considerable rework. In combining several distinct applications developed with a framework, persistency and user interface integration are often needed. Integration of persistency or object indexing mechanisms (ie mechanisms for low level object access, particularly associated with persistent or distributed objects), may require multiple name space support. For user interface integration, the development of Pluggable Objects and the application of the Hot Spot and Fine Grained Object patterns may be needed to address code redundancy and to combine interface functionality. A common approach is to use the Wrapper pattern to develop an API.

Component technology has made it common to construct applications developed within the framework as components. External tools can also be wrapped as components, using standard component interaction mechanisms to mediate integration. Schmidt et al [21] discuss a number of patterns, such as Component Configurator, Interceptor and Extension Interface that can be applied in these cases. Refactoring is often an essential first step in order to identify useful Shearing Layers and to avoid construction of a Big Ball of Mud [22].

Examples

Having developed the SPE and MViewsER applications, we immediately saw benefit in combining these to form an integrated 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 a fourth, coordination layer to the framework. The latter allowed change descriptions to be routed between separate CPRGs. Extension of the change response mechanism to allow change descriptions to be intercepted both before and after taking affect allowed sophisticated control integration to be simply effected [7]. This approach was developed considerably as the framework matured; for example we have developed pluggable objects that support collaborative work activities in a seamless and transparent fashion [12].

The MetaMOOSE framework for CASE tool construction, similar in application domain to JViews, originally supported only one application, Subsequent development added a shared repository to allow multiple applications generated by the framework to interact with one another [23]. The C2 development framework, which supports distributed inter-application messaging support, was an evolution of the Chiron-1 framework which lacked such support [24].

After developing several stand-alone 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 OS scripting and messaging architecture to the external tools. When converting JViews to Java, we exploited JavaBeans technology to turn JViews applications into JavaBeans. Exposing the JViews event management system as part of this approach allowed for powerful interaction and control, but caused occasional problems with "rogue" third party applications. Field [25], developed from Pecan [26] used wrappers, together with a proprietary message bus to integrate tools together. Oz [27], developed from Marvel [28], also used Wrappers to integrate other tools.

Related patterns are all of the EFPL transformation patterns and White Box and Black Box Framework patterns. Pluggable Objects is a pattern commonly used to implement this integration, using Wrapper patterns.

8.3 Reflective framework

The **context** is that you have a mature black box framework.

The **problem** is that you want applications developed using the framework to be extensible at run-time i.e. new components can be plugged in to extend the application's functionality. You may want third-party application integration to be similarly dynamic.

The **forces** are:

- Extending an application's functionality dynamically is hard
- In order to do this automatically, components must be self-describing
- Newly added components should appropriately share user interface, distribution, persistency and other related services of components already in use
- Behaviour of newly-added or re-configured objects must be validated

The **solution** is to provide a reflection mechanism for the framework components which allow run-time discovery of component characteristics and run-time plug-and-play and configuration of components.

The **implementation** of this pattern is difficult and the framework needs to be designed with support for it in mind. A mechanism for representing component knowledge is required, along with abstractions to query this data, configure components and validate component compositions.

Examples

We extended the JViews framework with "aspect objects", where "aspects" cross-cut the functional and non-functional behaviour of components to support the description of complex component properties [11]. We have used aspect objects to describe the functional and non-functional properties of a range of JViews-implemented components. Within JViews, support is provided to query component aspects and to configure components by standardised aspect-based interfaces. This is used to support dynamic plug-and-play of collaborative work components [11], which dynamically modify and utilise the user interface, persistency and distribution mechanisms of other components. Others have developed mechanisms for component description [29], [30] and run-time composition [31], [32], in order to support similar aims to this pattern.

8.4 Self-extending 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 developers. The plug-and-play composition of black-box components can only produce limited framework extensibility on its own and programming is needed to extend the framework's main abstractions.

The **forces** are:

- Static coding to change the framework 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, rather than rely on application developers to do this.

The **solution** is to provide a user-editable meta-model and user-tailorable interface components that can be used to dynamically extend the framework. This is a specialisation of the User-Defined Product (UDP) framework [33]. Users can specify new types for the meta-model and new user interfaces for their extended application, as they require, via interface tools rather than programming. This implies a need to understand the ways in which the user will want to use and extend a system; Domain Analysis may assist with this [34].

The **rationale** is that the pressures here are very similar to those that led to constructing a Visual Builder, for applications generated using the framework. However, here the framework itself (and related tools) is to be modified. It's natural in this case to develop a Knowledge level–Operational level split, as outlined by Fowler [35] and construct a meta model to describe the framework and its environment. By allowing this meta model to be visualised and edited, using a visual tool, new modelling abstractions and their properties can be defined. As composition of pluggable objects is by now the most common way of extending or modifying the framework, the changes can be effected dynamically. In addition to new modelling abstractions, users may want to tailor their application interfaces for these extended meta-model abstractions, and should be provided with a mechanism to do this. Some limited interface extension can be realised by automatically incorporating new meta-model types and derived instances in property sheets, but iconic editors and complex dialogues and reports require a user-extensible interface specification.

Implementation can be by a variety of means, but the Dynamic Object Model pattern [36] and User-Defined Product Framework are useful approaches to take. These in turn build on Fowler's Analysis Patterns. Key to the success of this approach is to adequately abstract out a meta-model representation, linking application model components to these meta-model types, specifying user interfaces in terms of (extensible) model elements, and providing sufficiently flexible and tailorable user interface building blocks.

There are a number of **liabilities** associated with use of this pattern. The framework becomes more complex, and hence potentially more difficult to maintain. If the meta level is interpreted, performance problems may result. Implementation may well require application of the Visual Builder pattern to provide necessary meta modelling tools, involving considerable programming.

Examples

The SoftArch modelling tool, which we have developed as one of the Visual Builders for our framework, has a visually editable meta-model and user-tailorable icons [15]. This allows the architecture modeling abstractions and constraints to be modified dynamically. For example, the left-hand example in Fig. 6 shows the visual meta-model editor used to extend and modify the SoftArch framework components. The framework has only the fundamental abstractions of "Component", "Association" and "Annotation". Users create instances of these fundamental "meta-types" for use as types in the SoftArch modelling tool, enabling new architecture abstractions to be dynamically added without programming. Properties of these fundamental abstractions are viewed and edited in an extensible dialogue.

Our other JViews environments are like most conventional CASE tools in that they have fixed meta-models (generated by JComposer) which can only be modified by code re-generation and recompilation. SoftArch also provides a set of tailorable visual abstractions, as shown in the right-hand example in Fig. 6, which users can dynamically tailor to represent new abstractions.

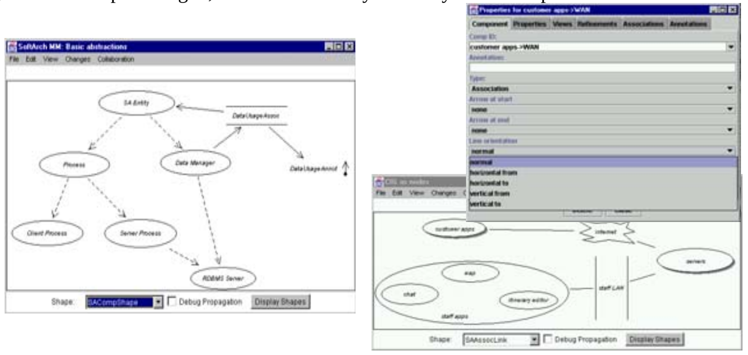

Fig. 6: User editable meta model for SoftArch. Links in the diagram correspond to specialisation (dotted) or relationship (solid).

The SimplyObjects CASE tool [37] provides an extensible meta-model and tailorable iconic shapes that tool users can extend without need to reprogram the tool. MetaEDIT+ [38] also provides a set of meta-model definition tools, iconic editor tools and tabular report tools allowing it to be tailored without recourse to black-box composition or programming. Rhiele et al [36] describe several other examples when describing the 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 verified the usefulness and effectiveness of the Evolving Frameworks Pattern Language, by calibrating it against our experience in developing the JViews framework over many years. As a result of this experience we have proposed a number of extensions to EFPL that capture other common activities that occur in framework development. We have introduced four new patterns, describing additional evolutionary steps in framework development.

This paper is unusual in the patterns community 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 [8]. 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 came from the University of Auckland Research Committee and the New Zealand Public Good Science Fund. The excellent shepherding of Neil Harrison and the assistance of the KoalaPlop 2001 group who workshopped the initial form of the new patterns proposed in this paper are gratefully acknowledged.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[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] Grundy, J.C. and Hosking, J.G. Engineering plug-in software components to support collaborative work, Software – Practice and Experience, vol. 32, Wiley, pp. 983-1013, 2002.

[12] 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.

[13] 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.

[14] Grundy, J.C. and Hosking, J.G. High-level Static and Dynamic Visualisation of Software Architectures, In Proceedings of the 2000 IEEE Symposium on Visual Languages, Seattle, Washington, Sept. 14-18 2000, IEEE CS Press.

[15] Grundy, J.C. and Hosking, J.G. SoftArch: Tool support for integrated software architecture development, International Journal of Software Engineering and Knowledge Engineering, Vol. 13, No. 2, April 2003, World Scientific, pp. 125-152.

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

[17] 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.

[18] 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.

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

[20] ObjectSpace Inc. http://www.objectspace.com/products/voyager/libraries.asp

[21] Schmidt, D.C., Stal, M., Rohnert, H., and Buschmann, F., "Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects", Wiley, 2000.

[22] Foote, B and Yoder, J, Big Ball of Mud, Chapter 29 of Harrison, N., Foote, B., and Rohnert, H., Pattern Languages of Programme Design 4, Addison Wesley 2000.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] Beugnard, A., Jezequel, J.-M., Plouzeau, N., Watkins, D., Making components contract aware, IEEE Computer, vol.32, no.7, July 1999, pp.38-45.

[30] Khan, K.M. Han, J., Composing security-aware software, IEEE Software, vol.19, no.1, Jan.-Feb. 2002, pp.34-41.

[31] C. Shuckman, L. Kirchner, J. Schummer and J.M. Haake, Designing object-oriented synchronous groupware with COAST. In Proceedings of the ACM Conference on Computer Supported Cooperative Work, ACM Press, November 1996, pp. 21-29.

[32] M. Mezini and K. Lieberherr, Adaptive Plug-and-Play Components for Evolutionary Software Development. In Proceedings of OOPSLA'98, Vancouver, WA (October 1998), ACM Press, pp. 97-116.

[33] Johnson, R and Oakes, J The User-Defined Product Framework, http://st-www.cs.uiuc.edu/users/johnson/papers/udp/UDP.html

[34] Coplien, J., Multi-Paradigm Design for C++, Addison Wesley, 1999.

[35] Fowler, M, 1997, Analysis Patterns: Reusable Object Models, Addison Wesley.

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

[37] Adaptive Arts, SimplyObjects CASE Tool, adaptive-arts.com.

[38] Kelly, S., Lyytinen, K., and Rossi, M., "Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment," In Proceedings of CAiSE'96, Lecture Notes in Computer Science 1080, Springer-Verlag, Heraklion, Crete, Greece, May 1996, pp. 1-21.