1997 OOPSLA Conference on Developing Successful Object Oriented Frameworks

Position Papers

Table of Contents

The CTP Framework

Tim Tjarks
Member of Technical Staff
Lucent Technologies/Bell Labs
tjarks@lucent.com

For the past 14 years, I have been developing tools to test telephone switching systems. During that time, I have been through a number of different application developments which are geared toward doing a specific kind of testing (background load generation, protocol monitoring, automated regression, etc) on specific types of switches or signaling links (analog call processing, ISDN, wireless networks, etc). The common way of doing business had always been to develop to the specific needs of the customer at hand, and so often redeveloped the same functionality in multiple products.

Four years ago, I began working in a team of three to define means for reusing components in multiple testing products, in order cut development time and take advantage of previous work. This led to the development of the (misnamed) Common Testing Platform (CTP), first as a Shlaer-Mellor analysis model and subsequently as a framework implemented in C++, for development of stimulus/response style testing products.

Our first attempt at development based off the CTP was successful at meeting a customer need, but failed in producing reusable components. This is mostly attributed to developing specific concrete classes that fulfilled the analysis model before spending additional time to develop abstract base classes to realize the framework communications. The classes developed performed their functions, but depended on too much knowledge of the working of other components to be separable from the specific application.

After the completion of the above mentioned development, the same small team which developed the initial analysis model revised the model to take into account lessons learned. As we began realizing the model as a framework and developing the abstract classes which, we expanded the team slightly to bring in additional department members to begin learning how to use and maintain the framework.

At the beginning of 1997, we began developing two new testing applications, building upon the framework classes completed at the end of 1996. One of these applications was a new ground-start project, while the second was to build upon parts of some of our legacy applications, re-engineering the components to take advantage of the CTP framework. Both of these applications are intended for use by customers internal to Lucent. The ground-start project has now reached its first customer release, while the legacy-based project has just begun development in earnest.

The CTP framework is intended for use by developers internal to our department in developing testing applications. The more junior members of the framework team have become the framework development team, while I and the other senior members have either become the application development teams (in the case of the ground start, the application team has but two members) or begun working with an application development team to gain acceptance of the framework (I have moved over to become project lead of the legacy-based project). The work projections for our department have us moving all development to be based on the CTP framework by the end of this year, yet acceptance by developers is slow. This is mostly due to inexperience with object-oriented development, and thus difficulty understanding the framework model underlying the intended new developments. The sentence may also be read as a failure of the framework design team to document the framework that is understandable without having been through the analysis process.

The CTP framework provides the communication model for the objects which work within the framework, and is suitable for describing how code and machine object modules can be fit together to form an application. Another aspect that I have been struggling with is how to create documentation frameworks which allow modules to own pieces of the end-user visible documentation (user-level requirements, users manuals, tutorials, etc) and construct the customer documentation from reusable components as easily as we can the executable image. Toward that end, I have been writing requirements for the aforementioned legacy-based project as modules corresponding to the executable components that will be developed. However, without first developing some form of documentation framework which these modules will then serve, the end result lacks a coherent flow which the customer can read and follow. This has led me to spending additional time interpreting the document to the customer, trying to relate the individual sections into a single, understandable whole.

Tim Tjarks received a BS in mathematics and physics from Iowa State University in 1980 and an MS in computer science from Purdue University in 1981. He has been with Bell Laboratories (now Lucent Technologies) since 1980, doing development of device drivers, remote target debuggers, and stimulus/response test systems for telephony applications.

The Definition of Framework: A Usage Survey and Proposal

John Artim, and Tony Leung
OOCL (USA), Inc.
(jartim@acm.org)
Tony Leung, IBM
(leungtk@us.ibm.com)

Last year's OOPSLA workshop on frameworks included a discussion of the definition of a framework. This discussion revealed a great deal of variability in the use of the term. For many participants, this term indicated the intended usage of a set of related code components as well as implying the degree of robustness and reusability of these components. Other participants, one of the authors included, favored definitions of framework that were more technical in nature - that is, definitions of framework that precisely and reflectively define a framework's content in terms of other, already precisely defined object-oriented concepts such as class and method.

Workshop participants each acted as though the concept of framework had long ago been pinned down with precision yet the degree of disagreement over its definition was disturbing. .In preparing for this year's framework workshop, we decided to research definitions of the term, framework starting with the dictionary.

The American Heritage Dictionary of the English Language defines framework as:

frame.work (fram?-wurk?) noun 1. A structure for supporting or enclosing something else, especially a skeletal support used as the basis for something being constructed. 2. An external work platform; a scaffold. 3. A fundamental structure, as for a written work or a system of ideas.

Definition (3) seems very close to the object-oriented (and, in many ways, more generally the programmer's) use of the term, framework. This strong similarity between the English sense of framework and our specialized use of the term may explain why we can so successfully use the word without agreeing on a specific, technical definition.

The next step was to scan through a number of years of OOPSLA proceedings to see how the term framework was used in our formal communications. We quickly found many uses of the term but none of these papers offered a definition or a citation to a paper that did provide such a definition.

There was, however, a great deal of similarity in the way the term was used (which may go some way to explain why we all use the term as though it has been defined). Every reference found described an object-oriented framework in terms of the classes (the fundamental components in our system of ideas) of which that framework is composed. Most authors either explicitly or implicitly made a visibility distinction using terms like public and private.

This seemed a good time to check on a few of the object-oriented analysis and design methodologists to see what they had to offer. James Rumbaugh and his coauthors in the OMT book, discussed the notion of framework but seemed to be restricting the discussion to frameworks as architectural pattern of design. Grady Booch, in his own object-oriented methods book, offered a more generic definition of framework: **Framework —** *A collection of classes that provide a set of services for a particular domain; a framework thus exports a number of individual classes and mechanisms that clients can use or adapt.*

This definition adds the notion of different client (consumer) viewpoints for the classes that make up a framework. Some clients are only interested in using the framework as intended. Other clients may want to adapt the framework, to one degree or another altering its structure to suit some new, possibly related purpose.

Ivar Jacobson and his co-authors discuss reusable components at some length. They define three kinds of components: white box components, black box components and frameworks.

A white box component is one which requires an extensive understanding of its design and the application domain to create a variation of that component for a similar use. Black box components are used as-is - that is, they are instantiated with only a knowledge of their method interface semantics and without any knowledge of their internal design. Finally, Jacobson, Et. al define a design framework as a special form of a white box component used as a skeleton within which to build an application. They go on to further qualify the design framework as specifying an abstract block or subsystem.

These object-oriented methodologists have identified two key viewpoints from which to document a framework and they have implied a third. We have clients of frameworks who wish to use it as intended, primarily though not exclusively limiting themselves to instantiating black-box components. We have clients whose use varies enough from the framework design that some modification or elaboration of the essential patterns of collaboration of the framework classes is needed. Finally, by implication we have a third viewpoint, the framework's creator and subsequent maintainers.

So, where does this leave us? I would argue that a framework is a collection of classes whose pattern of collaboration satisfies some usually well-documented purpose. A framework has a continuum of users ranging from simple re-use with little or no understanding of the underlying pattern of design to framework creator establishing a design's pattern of collaboration. Somewhere in the middle of this continuum is the interesting case of the informed consumer who must extend or elaborate on the pattern of collaboration following some anticipated patterns.

One aspect left out of this discussion is that frameworks are seldom used in isolation. In the extreme case, best illustrated by the system of frameworks introduced by the Taligent company, a set of frameworks are engineered to work together. That is, the pattern of collaboration of each framework includes points of collaboration with other frameworks. In our opinion, one of the largest problems facing commercial practice is that you never need a single framework in isolation, you almost always need two or more frameworks.

Initially, it might look as though the choice of these frameworks, each addressing its own requirements and each residing in a separate portion of the system architecture, can be treated as independent design choices. But that seldom seems to be the case. Once the first framework is chosen constraints are placed on the choice of subsequent frameworks. In practice, this means some or all of the frameworks must either be modified or else adapter bridgework must be built to span them.

Most of these sources have been consistent in defining the content of a framework. I would suggest a framework can be defined as a set of classes that embody a well understood and documented pattern of collaboration that supports one or more equally well documented purposes. These well documented purposes cover a continuum from instantiation of components of the framework only, to creation of a new framework based on the first framework's pattern of collaboration. This definition imposes a number of requirements on framework documentation.

First of all, the documentation must be able to show the user how to instantiate that framework. The framework class or classes that can be directly instantiated must be identified along with the appropriate constructor/destructor methods. If instances of a set of classes are intended to be used compositionally, the methods used to compose these instances must also be identified. The methods that represent the semantics (more loosely, the behavior) of these classes must also be identified. Typically, examples (scenarios) of use of these classes would be needed to visualize their intended collaboration.

Finally, many users would benefit from some expression of the visible pattern of collaboration seen in the intended use of these classes. As with classes, some frameworks are, at least in part, abstract. That is, one or more classes within the framework must be sub-classed in order to use the framework as designed. These classes must be identified. For each of these abstract classes, the methods that must be overridden must also be separately identified as well as the methods intended to be used by the concrete subclass and those methods to be considered a part of the concrete class's public interface.

One or more examples of well-formed concrete classes should be provided. These examples should fit into overall examples of use of the framework. The abstract classes should also be called out in a visual representation of the pattern of collaboration of the framework's classes. Some frameworks are designed to be extended by other frameworks. When we speak of a framework such as MVC we often are referring to it as a simple and abstract pattern of design. But, in the case of an implementation of MVC such as ParcPlace/Digitalk's VisualWorks, the real MVC framework has many component parts including subsidiary frameworks such as systems of wrapper classes, visual builders, etc.

Some aspects of the implementation imply intended extensibility - adding new widgets for example. Plugging in a new widget into this structure can be seen as adding a new framework. Expressing the responsibilities of a new widget as a framework has the advantage of binding together the set of related classes needed to add that widget. It can also help by showing the dependencies of the new widget on other widget frameworks. Even in cases where the widget adds only a single class to the system, that class may impose the need to add class extensions to existing classes in the system - this could also be documented.

By documenting both the intended ways in which the components of this MVC implementation are meant to be extended and the extensions themselves, framework developers and consumers both benefit. Framework developers benefit because their responsibilities are well defined. At present, these responsibilities are left to the developer to discover or are documented in fragmentary and disjointed ways. The advantage to the framework consumer is that they can see which frameworks are intended to be compatible with which other frameworks. And, if these framework-to-framework extensions and collaborations are well documented, novel combinations of framework can be better evaluated prior to installation and initial test. The need to experiment with the novel combination is not eliminated but it is better controlled.

Finally, the pattern of collaborations within each framework should also be documented. This can be for the benefit of the maintainer of the framework or it may be of use to some subsequent designer who wants to learn more about previously (well) designed frameworks.

To summarize, framework can be concrete (its classes are directly instantiated) or abstract (requiring the addition of one or more concrete classes). Extensions to an abstract framework can be simple (isolated concrete classes) or elaborate. Elaborate extensions imply the creation of a whole new framework that plugs into the existing framework in a pre-determined way. A framework can be designed with an intended collaboration with other frameworks in mind. Were these properties reflected in a meta-model, such as the meta-model described in the Unified Modeling Language OMG proposal, it would be possible for CASE tool vendors to provide a means of visualizing the documentation for a framework provided by a third party. It would also be much easier to coordinate use of frameworks from multiple vendors . A meta-model such as this would greatly simplify the commercial application of frameworks both from the standpoint of the frameworks providers and the consumers of these products.

John Artim works as User Interface Architect in the information services department of OOCL (USA), Inc., a global transportation company. John has worked on the design and development of user interface, CASE tool meta-model, and application frameworks and has been a consumer of frameworks as well. He is interested in advancing the state of the practice in user interface support of knowledge workers. *Tony Leung* is currently working on a database management product based on the Tivoli Framework at IBM's Santa Teresa Laboratory. His past projects included developing frameworks to support Object-oriented GUI development, an OO Analysis and Design tool for Frameworks, and an internal administration tool framework for DB2 Universal Database.

Developing Successful Object-Oriented Frameworks.

Mathew Schemenaur
mathew.Schemenaur@pscmail.ps.net

Building frameworks for the enterprise presents a slightly different set of concerns. It usually results in a smaller audience for the framework. The technical staff must be concerned with ROI issues.

Framework — Enterprise based frameworks provide a service to project teams within one company. Usually, this company produces several systems that share a similar problem domain. The framework implements a domain specific architecture. It captures the entities, relationships and dynamics that are common to a group of applications. For example, all Oil Field Automation projects could share a common design that is specialized for each project.

Sharing a common design across multiple project teams reduces development cost and shortens the delivery cycle once the framework is mature. Cost of Reuse Reusable software will cost more to develop than non-reusable software. It requires a greater investment in planning, communication, testing and training.

A framework requires the development team to investigate the needs of multiple clients and predict the future needs of clients. This alone is more expensive than the typical "throw away" design performed on most projects. Once the software is built, it must be productionized. This means the team needs to develop documentation and training classes. The documentation of a framework must be very detailed and consistent with the implementation.

To achieve a return on your investment in a framework, you need to reuse it at least three times. If you do not have at least three client projects that will reuse the framework, do not build it. Instead, use a less formal reuse library.

Selling the Framework — Focusing a framework within one enterprise creates a unique set of issues. Employees within a company compete for recognition and promotions. This can create a destructive conflict between the framework team and the client teams. The people issues can be more challenging than the technical issues.

The management must create an atmosphere of trust and respect between the framework team and the client teams. It helps to select respected individual from the client community as members of the framework team. This prevents the clients from viewing the framework team as a set of outsiders. It helps to let the advance clients to built some parts of the framework. This builds a sense of ownership for the more advance developers. The top technical developers in the client communicate must be champions of the framework.

Release Schedules — Living within the same enterprise makes it easy for the framework team to be responsive to the need of the client community. At the same time, it is more difficult to manage an effective release schedule.

The management of the framework team can take two approaches to the release schedules. First, the framework team can determine when functionality is needed by the clients and schedule releases when the clients need the functionality. Do not start with the client date and work backwards. Instead prioritize work based on when it is needed and package a release based on the need of the client.

Second, management could choose a fixed interval release schedule. Deliver an update to the framework on six month intervals. Prioritize functionality by the need of the clients, but maintain the release schedule. The clients can choose to wait for the framework functionally or to built it themselves. The second approach requires less management but increases the blame factor. The client projects will blame the framework team for delaying their development efforts.

Mathew Schemenaur , Perot Systems I have lead the development of object oriented systems for 12 years. I have worked on a variety of frameworks for companies including Shell Oil, Shell Development, AMR, _ Enterprise Oriented Frameworks My focus has been on custom frameworks for companies. I have built frameworks for Oil Field Automation, Interactive Exploration Systems and Transportation Capacity Planning.

Confessions Of A Framework Zealot

Brian Foote

Department of Computer Science
University of Illinois at Urbana-Champaign
3253 DCL
1304 W. Springfield Ave.
Urbana, IL 61801

(217) 244-6108 (Office) (217) 244-6869 (FAX)
foote@cs.uiuc.edu

This is a story about a framework. Indeed, it’s a story about a successful framework, albeit an unusual one. It’s about a real framework, not a research framework, or at least not a research framework in the customary sense. Oh, it was developed primarily in a university research context, and it’s for conducting scientific research, but it’s not a "research framework", in the sense that it is itself a research project or experiment. No, this framework was developed for real, paying customers, by a real, professional framework developer. That programmer was yours truly. The remainder of this tale takes a candid look at my experiences with this framework, and, in the process of doing so, attempts to cover as many of the workshop’s laundry list of "points of interests" as it can.

I’ve noted with satisfaction that the organizers of this workshop are insisting on war stories from real framework developers, and are requiring mere researchers to go to extra lengths to establish their bona fides. Now, this is not to say that I’m not also one of those framework researchers. Indeed, for the last several years, I’ve worn two hats: framework researcher and framework developer. I’ve been doing research on object-oriented frameworks for twelve years, and have written and spoken on the topic in a number of forums during this time. This paper however, is my first attempt to describe my experiences, both good and bad, with trying to practice professionally what I had theretofore preached.

OSIRIS

This framework is called OSIRIS. It’s a not particularly original name, but it’s sonorous and memorable. I had an acronymic justification ready for it (something like Object-Oriented Scientific Realtime Instrumentation System) but, to date, no one has asked about it. It is written in C++, and runs under Windows 3.11 and Windows 95. It is built using Borland’s C++ and makes fairly extensive use of another framework: Borland’s OWL GUI framework. It also includes a dash of older C and x86 assembly language code. The assembly language code performs device driver level functions, while the C code performs numerical tasks.

OSIRIS is a domain-specific object-oriented framework for building psychophysiological realtime experimental control and data acquisition applications. The basic idea is that one places electrodes on the heads of experimental subjects, and records their brain waves as they perform simple experimental tasks. These data are digitized using an A/D converter, and stored in digital form. The nature of the tasks, as well as the data acquisition strategy, varies from experiment to experiment.

By virtue of the nature of experimental research itself, the requirements placed on the experimental software change rapidly. A particular study may be run and published over a period of weeks. The challenge for the software designer is to write applications that encompass as wide a range of experimental possibilities as possible, and to make it easy to write new applications when existing ones are not sufficiently flexible to cope with new scientific needs.

The framework was constructed on and off over several years during the period from 1992 to 1996. Line-of-code metrics can be particularly misleading when one is discussing object-oriented frameworks. However, they can be useful up to a point as a gross indication of scale. The OSIRIS source files weigh-in at 53877 lines of code. By contrast, the OWL 4.53 GUI framework itself is about 41000 lines long. To be fair, there is more internal documentation in OSIRIS than in OWL. Of course, I'd argue that when it comes to frameworks, shorter is better anyway, and can make a case that OSIRIS would be shorter were it more mature. A typical application, DIM, built using OSIRIS contains 8563 lines of code.

The framework has been used to construct four distinct, major applications, for researchers at three different universities. Each application, in turn, has evolved to encompass the needs of between one to five distinct families of experiments. Customary laboratory programming practices would have employed distinct programs for each experiment (not application) and each program would have included its own versions of much of the code in the framework as well.

SOME HISTORY

This tale really begins during the late '70s, when I was working for a well-funded group of researchers at the Department of Psychology at UIUC, and had begun to grow weary of writing nearly the same application over and over again as requirements changed. Of course, I learned how to design and build libraries of reusable subroutines (we used structured Fortran and assembly language during that era) and functions, and how to cleverly parameterize programs to straddle as many potential requirements as I could. Parameterization allowed me to push design decisions out onto the user. If I could think of more than one way to do something, or a way of dealing with a particular request that could obviously be generalized to encompass a range of likely future requests, I'd put this in a user configurable table of parameters, and let the users change these themselves.

Usually, users would be delighted to find that a change that usually required a new program could be made on the spot by changing a parameter. I'd be pleased as well, of course, when my anticipation of the potential need for such generality spared me additional work.

This strategy also helped to forestall what I could see would become the bane of my existence: a phenomenon I later dubbed metastisization [Foote1988], or the proliferation of many nearly, but not exactly identical versions of the same applications. This proliferation usually came about as a result of expedient, cut-and-paste alteration to existing, working applications, often at the hands of clever researchers who were, nevertheless, relatively unskilled programmers. I could see that the maintenance burden associated with such unbounded proliferation could consume time that might be better spent writing (for instance) fancier displays for the new graphics hardware that was starting to emerge.

This seemed like an ignominious fate. Surely, some combination of cleverness and laziness could triumph over this tedium and drudgery. (Of course, I later realized that the usual solution to this problem was to change jobs once these sorts of problems started to arise.) My belief at the time was that there had to be a better way, and in 1981, at the National Computer Conference in Chicago, I came across something that looked like it might be the answer: object-oriented programming.

It was hard to miss: Xerox had decided to unleash Smalltalk-80, and held a high-profile event to introduce their porting partners. Xerox was also showing the Star Workstation, which gave many attendees their first glimpses of the sort of windowed desktops that were destined to take over the world. Intel was showing the iAPX432, a super-CISC processor with object-oriented instructions, which ran a crude, Smalltalk-like language. Other vendors were showing bitmapped workstation displays for the first time.

I returned home, and tried to find out all I could about objects, bitmapped workstations, and these processors of the future. What I found out was that objects had some remarkable properties. You could use inheritance to share code and data that always stayed the same, and put the just things that changed out at the edges. Because of dynamic binding, you could use any object so constructed anywhere where its shared repertoire might be invoked. Here, perhaps, was the answer to my problem of how to avoid having to write pretty much the same program over and over again.

Of course, mere mortals working with minicomputers had no access to the sorts of languages, tools and displays I’d seen at McCormack Place. Even so, I set spent the next several years working on battery of the psychophysiological applications we were constructing as part of a "mini-OEM" laboratory data acquisition system we’d decided to sell to other researchers. Over 30 of these $50,000/$100,000 DEC LSI/11-based "Pearl" systems were sold during this period.

The "battery" software needed to accommodate a wide range of potential research needs, which change quickly, and are, by definition, difficult to anticipate. I tried to engineer these programs to be as reusable as I could. I pushed as much sharable code as I could into library functions, and made them as customizable as I could by exposing parameter editors (which exposed hundreds of parameters). I also crafted the core of applications so that they shared the same code base, using a home-built preprocessor. By doing so, I was intentionally trying to crudely glean the benefits I thought object-oriented inheritance would give me, had I access to such tools.

The preprocessor also produced a table of dynamic meta-information that allowed users and programs to access parameters by name at runtime. These tables tracked data types, legal ranges, and user help information. (Here were the roots of my eventual interest in object-oriented reflection.)

Over the next several years, I read as much O-O literature as I could, and even acquired an Apple Lisa so as to get my hands on Classcal, ObjectPascal, and, eventually an exceedingly slow version of Smalltalk. I also decided to resume my graduate education in earnest, and came across a junior professor, Ralph Johnson, who was eager to let me explore my object-oriented generic laboratory application idea in Smalltalk.

The initial result of this collaboration was a Smalltalk framework that implemented a simulation of the battery applications I'd built over the previous several years at work. Every line of this framework was presented in "Designing to Facilitate Change with Object-Oriented Frameworks" [Foote1988]. The principal conclusion of this work was that the idea of constructing a reusable, generic skeleton application for this domain out of dynamically sharable objects, that could serve as the nucleus for a family of related applications, really was feasible. By specializing different classes in the framework, specific applications could be derived from the generic core with relatively little code.

This desultory but amusing work also contained a lengthy discussion of how frameworks evolve. I was frankly surprised that despite years of domain experience, my initial design for framework evolved very substantially as I incorporated requirements from new applications, and as I exploited opportunities to glean yet more general objects from my code. I also didn't initially expect the degree to which inheritance seemed to be supplanted by delegation and forwarding to components as my design became more mature. However, such a strategy allowed the kinds of dynamically pluggability for objects that my old parameter editors had made possible for simpler data in my earlier laboratory applications.

The emergence of distinct architectural identities for these objects, which would allow them to themselves serve as loci for specialization and evolution, enhanced their reuse potential. Some of these findings were discussed in [Johnson & Foote 1988].

DOCUMENTATION: CATALOG, COOKBOOK, ANATOMY, AND SOURCE

I know of no successful framework that has achieved significant popularity that doesn't come with a book, and full source code. In order to use, and reuse, the classes and components a framework exposes, programmers have to understand both the black-box interfaces, and when need be, the white-box internals of the framework. At various times, clients need:

1. A catalog, to help them determine what, if any, wares the framework has to offer them
2. A cookbook, to show them how to use elements of the framework, and how they fit together
3. An anatomy that exposes the detailed internals of the frameworks, such that they can be modified and extended.

There is no substitute for this sort of detailed, multi-level documentation if a framework is to flourish. Developers seem to find it easier to provide the detailed anatomy, or reference, documentation than they do the catalogs and cookbooks. Good examples, and comprehensible overviews that give one a sense of how the pieces fit, and why I (as a potential client) might care are harder to come by.

The source code is the ultimate, irrefutable, utterly operational authority on how the framework behaves. The source has the final say as to how the framework works. If it doesn't work the way the client wants, he or she can extend or change it (at the cost of maintaining these enhancements).

Genuinely good documentation is essential if a framework is to prosper and endure. Note that while I have claimed that OSIRIS is a successful framework, I have not claimed that it has achieved significant popularity. There is adequate reference documentation for the low-level realtime portions of the OSIRIS framework, and there is, of course, commercial and even third-party documentation for OWL. However, I have been to date unable to find the time to document the framework, nor anyone who is interested in underwriting such an effort. One reason for this is that the perpetually looming possibility of major overhauls makes it easy to rationalize procrastination. This is, of course, a trap. The lack of a suitable patron to pay for the documentation is a rationale that stands up better to sober analysis. Any discussion of framework economics must include the cost of producing the documentation, and the cost of keeping it up to date, and the cost of distributing it. The Web can certainly be of assistance with the latter two items.

DOCUMENTATION: MENTORING

An alternative to lavish documentation is to be the "walking manual" yourself. One-on-one mentoring with new framework clients is not a good permanent substitute for documentation. Clients don't mind it as long as it is plentiful, but your typical solo virtuoso may find that this role becomes tedious, and that, in any case, that it robs the development process of precious uniprocessor cycles. I've found myself playing this role with the OSIRIS framework. Since there really hasn't been that much independent development undertaken with OSIRIS, this burden has thus far been tolerable.

SOLO VIRTUOSO

Jim Coplien, in his Generative Development Process Pattern Language, describes a pattern, Size the Organization, which suggests "ten" as the answer to the question of how big a software development team should be. A subsequent pattern suggests another solution to this problem: "one". This is the Solo Virtuoso pattern.

The notion of the "virtuoso" software developer comes from Richard Gabriel. The idea is essentially that if one (presumably good) person can do the job, you can avoid the communication, organizational, and political costs of a large group by doing the job this way. There is, of course, the implication the term "virtuoso" that the developer who plays this role must be exceptionally skilled.

Various investigators report differences of up to two orders of magnitude in the skill levels of software professionals. My impression has always been that some of these differences are overdrawn, and result from correctable deficiencies such as a lack of empowerment, poorly partitioned projects, inadequate training, domain inexperience, poor communication, and a lack of incentives.

Sometimes the way to win a medal isn't by writing superior code. I like to think I'm a pretty good software developer (who doesn't), but, in any case, the solo style was imposed on me by necessity. I can, however, give confident, though subjective, testimony as to some of the advantages of this approach.

Autonomy: I controlled the product, and the architecture, (though not always the process). In particular, decisions regarded architecture and evolution were made in my head. It is well know that communication among team members is slow, and consumes an inordinate amount of time. As a solo act, these decisions were made intra-cranially. Thought is usual much faster than talk (except when one engages in an extended debate with one's self). Most importantly, when I wanted to change an interface, I was, as both client and provider, able to weigh the cost of complying with an interface change vs. retaining backward compatibility myself. Further, as author of both the clients and the core framework, I was intimately familiar with the ramifications of such changes on both.

Style: As a solo act, you get to do it your way. I found that it's useful to code in a way so that your code is recognizable amidst the OWL code and sample templates I used. Code Ownership, as (for instance) Coplien has observed, contributes to pride in workmanship, accountability, and easier maintenance, and conceptual and architectural integrity. I no more want to someone else to edit a piece of code that I'm responsible for maintaining and improving than I want to arrive at work and find my office and desk rearranged. If somebody moves something, I want to know it, and I want to know why. With code, I'll often want to re-assimilate it my way. I don't advocate this for a collaborative code, of course, but as a solo architect-builder, I have the luxury of not dealing with it. Indeed, I'd bean advocate of stylistic standards for collaborative work that encourage a sense that a body of code is recognizably <our> code, and not the exclusive bailiwick of any one individual. (I've been contemplating a writing proto-pattern (to be named SCHOOL UNIFORMS) that would make this case, but need to collect more evidence in order to do so.)

A significant additional consequence of the solo approach is that one reaps what one sows. It's hard to retain a slash-and-burn mentality toward one's work when you know that you'll be the guy who has to clean up after your mistakes. Conversely, if you take the time to make your framework general, you are the guy on Easy Street the next time you need to build a new application. There are, of course some downsides to this approach. If one has developed any design blind-spots, there is no one else to spot them. If one has tendency towards lily gilding, there is no one to arrest it. The fact that my clients are dependent on an irreplaceable architect-builder troubles them considerably more than it does me.

ECONOMICS

The OSIRIS experience has brought me face-to-face with the cold dirty economic facts of framework development life. Who pays for refactoring? Who pays for architecture? Who benefits? Who owns what? How do you avoid giving away the store? Especially if refactoring and architecture is expensive, how do you charge for it?

My usual approach was to treat refactoring as a precursor to additional development in a particular part of the framework. This way, the current client, who was to be the first beneficiary of the change, paid for the time. (I've always billed time, and not deliverables, when doing this sort of work. I've also used license fees.) Of course, a case can be made that every subsequent client is the benefit of some of these reusable elements, and that the cost might be somehow amortized. Fortunately, in the small potatoes world of OSIRIS, this never became a serious issue.

It is easy to imagine these issue being a far more serious concern to larger projects. Some of the refactoring in OSIRIS was done on my own time, as "sweat equity" investments in the framework. While I might have preferred that some benefactor underwrite this work, such effort is unencumbered by deliverable pressures, and gives one a basis for making certain ownership claims as well. Another serious issue is ownership. If you deliver a framework with source and examples, may your colleagues go into completion with you? They have everything they need to do so. What of the issue of code that was developed for a particular client, but emerges as general, sharable code.

In the academic world, some funding agencies consider code developed on their dime as government property. One needs to be aware of licensing, copyright, and ownership issues to handle these questions, especially as the stakes grow larger. Proprietary client concerns are a problem too. A particular client may not be thrilled that your passion for crafting reusable artifacts based on his or her requirements will technically empower his or her competitors.

So far, I've been able to keep such "competitive sensitive" material in the client applications, but it's not hard to imagine it becoming a more serious problem in a broader framework. Scheduling refactoring is another key issue. Blending it with development, and treating it as a background task are two ways of dealing with it. Convincing management that a long term investment in architecture really will pay off down the road is an enduring difficulty. Ralph Johnson has estimated that about a third of a development groups time should be devoted to refactoring. This sounds about right to me.

CONFESSIONS

* Frameworks are hard to build

There is simply no denying that it takes time, skill, and resources to get these right. You can't just commission one. You have to make a commitment to pursuing, even cultivating frameworks to reap the reuse rewards. You can't design a really good one up-front, unless you can leverage pre-existing domain experience.

I am bemused by the hubris of those who think they can. If you've worked in the area for a while, you may have the knowledge to discern the right abstractions already. If you are working in an area where frameworks and libraries existing already, then there are shoulders to stand on. One the other hand, if you are working in a domain where architecture has been unexplored or undiscussed, you've got a long row to hoe. Initial analysis usually yields a decent, but superficial collection of objects that model the surface structure of the domain, which can grow down. Initial implementation efforts grow up in response to linguistic and resource driven forces. In between is the realm of the truly reusable objects, which become discernable only as successive efforts to redeploy it are made, and the code is refactored to embody these insights. It is from this process that frameworks emerge. We've written on a number of occasions about this process [Foote 1998][Johnson & Foote 1988][Foote & Opdyke 1994][Roberts &Johnson 1996].

*Domain knowledge is essential

You need to find a skilled software architect who is willing to become extremely familiar with the domain in question, and spend time interacting with domain experts, in order to glean the right objects from amongst the pyrite.

From the standpoint of the architect him or herself, this constitutes a significant commitment, and investment. Framework design is not a casual pursuit, and one needs to choose one's target domain wisely. (Now, if only I'd gotten involved with realtime securities trading')

Over-design is a trap. Top-down frameworks can be structural straightjackets. You can paint yourself into corners with architecture that your purely functional pieces would not force you into. This is one reason why white-box phases are a natural part of the early evolution of a framework. Form (architecture) really does follow function (features, data, and code), and mature components emerge from a "messy kitchen" architectural phase, where you scatter ingredients all over the counter before consigning them to there place in the stew.

Of course, it's a mistake to say that it's futile to attempt to design any architectural elements at all early in the lifecycle. Particularly in those cases where the architect has prior experience in the domain, reasonable initial conjectures can be made. My advice: take the obvious, don't pan for the clever. That can come later, when real opportunities to exploit such insights arise.

Frameworks evolve, as do species, when they are <stressed>by the environment. They can even be <cultivated> by selecting successive challenges for them that seem representative of the range of applications they might be expected to ultimately encounter. Reuse erodes structure, and structure can impede evolution. Consolidation and refactoring later in the lifecycle are essential to arrest these entropic tendencies. It is late in the lifecycle when the experience that can tell you what abstractions are necessary to straddle existing architectural requirements can be exploited.

C++ is hard to refactor. Having cut my object-oriented teeth on Smalltalk, I elected to use C++ to construct OSIRIS because using Smalltalk for the sorts of hard realtime tasks that my public demanded was impractical. In 1991, 286 and 386 era processors could not provide the RAM or CPU horsepower necessary to handle the response and processing requirements this domain posed under Smalltalk. There were licensing, cost, and training issues too. By contrast, C++ compilers were beginning to emerge with environments and performance that seemed promising. Existing client Fortran and C skills seemed more likely to map to a C++ environment than to Smalltalk. Porting existing computational functions to C++ would be relatively simple, and those that had already been coded in C would be immediately usable. (In hindsight, one of the reasons for C++'s success (and it was successful, whatever you may think of it's prospects now), was it's ability to allow C programs to become C++ programs without requiring a total rewrite.) Furthermore, I had sketched the design in Smalltalk, and, with this in hand, could proceed to re-implement my framework in C++ with my simulation as a roadmap.

By-and-large, this was, in fact, what I did, and, by-and-large, it worked. As such, this would seem to vindicate those who advocate prototyping in languages like Smalltalk, while building production code using blue-collar languages like C++. I could, in a different frame of mind, and for a different audience, extol the virtues of this strategy, but this not my purpose at the moment. Instead, in hindsight, I'm struck most by the difference in programming tactics the two languages seem to demand. My sense is that it took me between 3 and 5 times as long to construct the same functionality in C++ as in Smalltalk. These numbers are somewhat subjective, but are based in part on logs I kept for billing purposes during the construction of OSIRIS. In particular, <refactoring> C++ code is a much more tedious process than in Smalltalk.

The combination of two factors makes this so: the declarative redundancy C++ needs to do type-checking, and the paucity of tools to assist in this task. The combination of the two makes one must less "quick on one' s feet" when it comes to performing refactorings.

Refactorings are changes to the system that enhance it's architectural integrity and reusability rather than add capabilities or enhance performance. Refactoring is the means by which architecture emerges as a framework evolves, Refactoring a C++ program entailed running around all over the system dealing with calls, and type declarations, and the like. More changes and more opportunities for inconsistency meant more chances to accidentally break something. Even a refactoring as simple as changing a member name might be so time consuming as to be avoided.

The aggregate effect of this was that I found myself adopting a much more deliberate, conservative approach to refactoring than I had in Smalltalk. The eventual, cumulative effect was to change the character of the way the framework evolved. Architectural change took place in coarser, larger grained, more deliberate, less frequent increments than it did in Smalltalk. It became easy to defer more speculative exploration, and to defer obviously worthwhile change, for fear of collateral damage. One had to not try to break the framework unless one had time to fix it. Indeed, there are, even now, a bevy of deferred refactoring opportunities in OSIRIS waiting for enough of my time to realize them. I have thought it likely that had my framework <research> been conducted in C++ rather than Smalltalk, our conclusions would have been quite different.

The power of refactoring and our ideas about reuse itself, and the emergence of black box components from white box inheritance-based precursors might have escaped us. The experience has given me insight into why the C++ development world seemed so bureaucratic, and even as to why the pattern's movement has resonated so clearly in it. The point at which one goes from being able to handle things oneself, to where you need a team, would seem to come considerably sooner in C++.

In the case of C++, better tools could still help a lot. (Java would seem to sit somewhere between C++ and Smalltalk with respect to both linguistic cumbersomeness and tool support.) Certainly, there are other forces at work here. Foremost among them is that OSIRIS, unlike my Smalltalk simulation, was a real, working system facing real requirements. The world doesn't pull its punches to make your architecture work out better. With my simulation, some of the architectural heavy lifting had been borrowed from my battery experience.

With OSIRIS, the novel domain challenges really have been new ones to me. * Multifunction components are more complex than single function components, so this complexity must be managed. It's harder to come up with tools that do a range of jobs well, rather than one. There are times when this shouldn't even be attempted. The art of framework construction is in being able to recognize how and when exploit opportunities to factor common, reusable elements from among similar components, and consign that which distinguishes them to separate subclasses or components.

Frameworks really do work, but it’s a long hard road to get there.

My overarching conclusion is having been there, frameworks really do work. Reuse is real, and it’s powerful. In my case, there is simply no way that I could have constructed and maintained the number of distinct, complicated applications I did without factoring the common parts into a framework. When you polish or repair a core element of a framework, everyone benefits, almost retroactively. This, in turn, amortizes the cost of this sort of effort across all its beneficiaries. Once one has distilled the generic essence of a domain, and embodies it in a framework, one really can deploy new applications in a fraction of the time it would otherwise take. (The line of code measurements discussed earlier are good rough indications of the scale of these efforts.)

In my case, that made bidding certain jobs economically viable, where they otherwise would not have been. As an individual, framework has been an incredible lever, and has enabled me to compete as a solo act in a domain where small armies are now being deployed. How might one bottle and scale this’ Small, commando teams of similarly inclined architect-builders could wor?

A "skunk-works" mentality seems to be helpful. This seems to be the model employed in so-called hyper-productive organizations. It’s hard for me to imagine framework development being done in the big-shop assembly line style, without factoring the task along the grain of the domain into these sorts of teams.

For me, the reward is in being able to craft quality artifacts. I’m paid to produce working programs, but, I must confess, my consuming concern is with artifacts themselves. The code is where I live, and when my nest is littered with spaghetti code squalor, it’s not a pretty place to be. Reuse is motivated by a particular combination of a sloth and craftsmanship. If there is one thing more satisfying that a thousand line of code weekend, it’s a<negative> thousand line of code weekend. These are not unusual occurrences during framework development. If there is anything worse than rewriting the same application over and over again, it’s maintaining a bushel of them.

Frameworks give us a way out of this quagmire. Ten years ago, I said that if my alternatives are to roll the same rock up the same hill everyday or leave a legacy of polished, tested, general components as the result of my toil, I know what my choice will be. I still believe that.

Brian Foote has over twenty years of professional programming and consulting experience in the realm of realtime scientific systems and applications. In addition, since 1985, he has also been engaged in research on object languages and frameworks, as well as software reuse, software evolution, reflection, patterns, and software architecture, and has taught and published in these areas. He is one of approximately 30 people to have attended every OOPSLA conference to-date. This highly unusual combination of practical experience and research training converges in his enduring interest in where good code comes from. Brian is currently a Visiting Research

Programmer at the University of Illinois at Urbana-Champaign. He received his MS in Computer Science therein 1988, and his BS in 1977.

REFERENCES

[Coplien 1994] James O. CoplienA Generative Development Process Pattern LanguageFirst Conference on Pattern Languages of Programs (PLoP '94) Monticello, Illinois, August 1994Pattern Languages of Program Design edited by James O. Coplien and Douglas C. Schmidt Addison-Wesley, 1995

[Foote 1988] Brian Foote (advisor: Ralph Johnson) Designing to Facilitate Change with Object-Oriented Frameworks Master Thesis (advisor: Ralph Johnson) University of Illinois, 1988

[Foote & Opdyke 1994] Brian Foote and William F. Opdyke Life-cycle and Refactoring Patterns that Support Evolution and Reuse First Conference on Pattern Languages of Programs (PLoP '94) Monticello, Illinois, August 1994
Pattern Languages of Program Design edited by James O. Coplien and Douglas C. Schmidt Addison-Wesley, 1995

[Johnson & Foote 1988] Ralph E. Johnson and Brian Foote Designing Reusable Classes Journal of Object-Oriented Programming Volume 1, Number 2, June/July 1988pages 22-35
[Roberts & Johnson 1996] Don Roberts and Ralph E. Johnson Evolve Frameworks into Domain-Specific Languages PLoP '96 submission

Application Development Using Our Meta-Repository Based Framework.

Martine Devos and Michel Tilman

mdevos@argo.be
mtilman@argo.be

1. Abstract

This position paper describes experience with a repository-based framework for evolutionary software development, designed, developed and used at Argo to support its administration. Argo is a semi-government organization managing several hundred public schools. It uses this framework to develop its applications, which share a common business model, and require database, electronic document, workflow and Internet functionality.

The framework uses a (meta-)repository to capture formal and informal knowledge of the business model. Part of this knowledge is explicitly modeled; other practices are less formally specified. To model, configure and manage the repository, we provide the user with several high-level tools that require only the business model to get the system up and running. With the end-user tools, users select applications, enter and view data, query the repository, display, print and export the results of a query, access the thesaurus and manage electronic documents, workflow processes and task assignments.

Configuration and administration tools enable users to define private and shared views, to store queries for later re-use, to edit the object model, to set up processes, to define authorization and business rules, and to add or remove end-user functionality. These tools have no hard-coded knowledge of a particular business model. They consult the repository at run-time, querying the meta-model for dynamic behavior. Changes can be made at run-time and are immediately available to clients. Thus we effectively separate descriptions of an organization's business logic from the application functionality.

Using our framework, we develop applications in an iterative and incremental way, even interactively. In fact, we build increasingly complete specifications of applications that can be executed at once.

2. Context

Work on the framework started 3 years ago at Argo, a semi-government organization that manages the public schools (non-denominational) within the Flemish community in Belgium. Argo consists of a central administration and a few hundred semi-autonomous local sites (boards and schools). Argo requested both end-user applications and a framework for developing these applications. In the past this has been confined to the central administration, but this will include other services as well in the near future, mainly towards the local sites.

We started with three pilot applications to provide input for the framework requirements and design:

· a documentation center application, requiring flexible search and retrieve of data and documents · support of the central board's decision procedures
· an application to scan, index and route large volumes of documents sent by the local boards.

Since then, several end-user applications have been delivered and new ones are in various phases of development. In addition, we created some applications to support the development process, such as bug reporting and follow-up, training session management and framework documentation management.

Halfway the project a pending re-organization was effectively carried through. Although the nature and extent of the re-organization were largely unknown at the start of the project, we were able to adapt the existing applications through re-configuration without additional coding. We are now in the process of extending the framework; most notably to give schools and local boards access to the repository through the Internet. Within this process we further refine and re-design many framework components.

The persistency component, for instance, has been completely re-designed and re-implemented, to address new requirements and performance issues. End-users at Argo are gradually taking over management and configuration of the applications from the development team. Key-users teach and write part of the documentation. User groups organize workshops to discuss how to put the technology to better use.

This process was initially driven by small teams of highly motivated pioneer-users, and has gradually started to embrace the end-user community at large. Given the high degree of computer-illiteracy within Argo at the start of the project, we feel this to be a significant achievement on behalf of the end-users.

2.1. Project Requirements

The technology had to support applications containing a mixture of database, electronic document and workflow management, and had to give local sites access to the central repository in the near future. Users needed access to a common business model, respecting standardized policies (e.g. indexing vocabulary of documentation, and global validation and authorization rules), but should have the freedom to adapt the technology to their most appropriate working practices. To cater for changing needs, the system had to forego hard coding as much as possible. At the end of the project, Argo users ought to be able to develop applications themselves. Hence the deliveries did not only include end-user applications, but also the necessary tools to empower users to model and configure applications.

These tools had to be part of an object-oriented framework. The technology had to support iterative and incremental development through the use of prototypes. Even more, it had to act as a catalyst for a learning process when introducing this new technology, thus effectively complementing a Business Process Re-engineering project at Argo.

2.1.1. End-user application requirements

Argo wanted access to the applications to be integrated in one environment, using the same (logical) repository and electronic document storage. Authorization control had to enable administrators to grant and deny access to sensitive information in a very controlled way. Access privileges could be context- and contents-sensitive. Database Users ought to be able to enter, modify, remove, query, list, browse, report, print, import and export data. They needed tools to set up appropriate views, and to store queries for later re-use. Electronic document management Users needed to create, index, search, edit, annotate, print, export and process documents. These could be either electronic (such as files or E-mail) or paper documents, in which case they might have to be scanned. Indexing and searching should use structured information, thesaurus keywords and full-text-indexing. Optical character recognition was required to extract text from scanned documents. Users should be able to manage document versions and to choose the most appropriate representation of a document for a given job. Workflow Users had to be able to plan and handle incoming tasks, to keep track of outgoing task assignments and to manage workflow processes, which can be more or less well defined or completely ad-hoc.

Argo wanted workflow functionality to offer users opportunities, rather than restricting them. Remote access Users at Argo needed remote access to applications, data, documents and processes in the central repository in several ways:

· using an off-line version of the system through import / export of documents and data
· using the system on-line by means of a direct dial-up connection, e.g. modem or ISDN ·
using E-mail and Web browsers to access the repository through the Internet.

2.1.2. Configuration and administration tools requirements

Argo wanted be able to define the object model, constraints and behavior, to configure and manage applications, views and stored queries, to define and manage workflow processes, to set up access control, to manage the database, and to configure caching and backup of documents. Automated tasks had to be managed by a generic process manager, e.g. handling migration of electronic documents to and from optical disk, and getting E-mail into and out of the system. Configuration of (new) automated tasks had to be provided for.

2.1.3. Framework requirements

The end-user and configuration tools were to be developed using an object-oriented framework satisfying the following requirements:

· It should allow Argo to develop end-user applications through modeling and configuration, rather than through coding.
· To easily support changing needs, as few assumptions as possible about a particular business model had to be hard-coded. Rather, this knowledge should be dynamically available from a central repository.

2.1.4 Existing infrastructure

The end-user applications and configuration tools had to operate in a Windows / Novell / Oracle / cc: Mail / Internet environment.

3 Argo framework approach

Our approach aims to combine the high-level modeling power of CASE-tools with the open-ended nature of object-oriented frameworks. It relies on the following observations:

· Users tend to describe their requirements in terms of how they currently work. It is difficult for them to picture how this new technology can or will influence their future working practices. Hence the need to try out, to learn and to correct the software being delivered. This requires extensive prototyping. Too often, however, prototypes are used to validate the design, rather than help users and developers capture and explore the real requirements. Prototypes can be expensive, as they usually have to be thrown away. This is particularly true when developing many applications, each one requiring several iterations to get the requirements straight. · Whereas the business model is particular to each organization at a particular point in time [Sto94], end-user or configuration functionality, such as data-entry, querying, reporting and document management, is all the more generic.
· Extensions or enhancements to the functionality can often be made into re-usable assets, independent of the actual business model, and vice versa.

3.1 Architecture

The framework uses a (meta-)repository to capture formal and informal knowledge of a particular business model, including organization structure, roles, data, documents, processes, applications, rules, and personal and shared working practices. Part of this knowledge is explicitly represented in the object model, other practices are less formally specified, e.g. through custom scripts or via the population of the repository.

To model, configure, tailor and manage the repository, we provide the user with several high-level tools that require only the business model to get the system up and running:

· End-user tools enable users to select applications, to enter and view data, to query the repository, to display, print and export the results of a query, to access the thesaurus, to manage electronic documents and to manage workflow processes and task assignments.
· Configuration and administration tools enable users to define private and shared views, to store queries for later re-use, to edit the object model, to set up processes, to define authorization and business rules, and to add or remove end-user functionality.

These tools have no hard-coded knowledge of a particular business model. They consult the repository at run-time, querying the meta-model for dynamic behavior. Changes made at run-time are immediately available to clients. Hence we effectively separate descriptions of an organization's business model from the application functionality. To support this behavior, the repository contains both object- and meta-level knowledge. The meta-model, used to describe the actual object model, can be expressed in terms of itself and is stored in the repository too.

Another important design goal of the framework is to support a bootstrapping process, whereby administration and configuration tools are gradually expressed in terms of the framework itself, i.e. replacing most hard-wired administration and configuration tools through configured applications. This should lead to a small but powerful kernel of generic and orthogonal tools. Benefits In contrast to most CASE-tool approaches, the business model is not used to generate an end-user application; neither do we code (apart from some custom scripting) applications. This results in increased flexibility and maintainability, particularly when developing in an iterative way. Using a (meta-)repository in combination with ready-built generic tools effectively decreases the gap between specification, development and use of the applications [Dev96].

We do not use nor do we need throwaway prototypes. Instead, we build increasingly complete specifications of end-user applications. These specifications are available for immediate execution. As all components, tools, and model and functionality of the (meta-) repository are part of the same environment, they can evolve over time within the framework as needed. This allows us to add new tools and functionality, to change the meta-model and to cope with future technologies and media.

3.2. Building applications

Starting from the model in the repository at a certain moment, we build applications by going through of a set of steps. These steps can be performed in any order, even interactively. This enables the development process to unfold incrementally. Hence, we can try out and correct ideas dynamically, together with our users. Starting with a model, which initially contains only meta-model and system objects, developing an end-user application implies:

· Extend or refine the object model. If new structures or modifications are required, we update the model with the necessary object types, association types and global constraints. If needed, we add behavior. The object model, shared by all applications, provides cohesion. We store this object model in the repository to make it dynamically available for the tools. This way, we can set up arbitrary business models without re-coding tools or applications.
· Set up application environments. This step starts with defining a view on the shared business model: objects and properties to be accessed, created and queried. Once this basic environment has

been stored in the repository and access privileges have been set, it is available for immediate use. The user can log on, choose this environment, enter data, query the repository, list or print results. He can keep track of task assignments, manage electronic documents and access the thesaurus. Functionality can be added or removed. The environment can be further refined for individual or shared use through views, queries and subenvironments.
· Extend or refine authorization rules. Authorization rules are not tied to any specific model and can be used to set up both very fine- and course-grained access. Authorization rules, just as action rules, can be context- and contents-sensitive. Some rules apply to everyone; other rules apply to groups of people whose members are determined at run-time. This way, authorization rules support cohesion, while giving considerable flexibility in supporting different team cultures.
· Define action rules. These rules capture business-semantics, set defaults, perform extra validation, filter information and add functionality. Action rules can cover an entire organization, particular functions or can be limited to a specific application environment. Hence action rules give extensive support to autonomous teams while preserving overall consistency.
· Define workflow process templates. These specify default scenarios that the user typically follows in the context of a business process. Users can (usually) deviate from these scenarios and snap back into the pre-defined flow later on. Processes can be configured incrementally, e.g. between departments first, and within each department later on. Depending on team culture, these individual subprocesses are more or less strictly defined. In addition, we add automated tasks and dedicated private or shared in- / outbaskets to manage incoming and outgoing work. This process is illustrated in the following picture. Various tools are drawn at the bottom. Incoming arrows denote input for the tools; outgoing arrows specify the products.

3.3. Framework evolution

When developing applications with the framework we identify:

· Working practices specific to the organization. We create re-usable assets in the repository, re-factor the object model, or we develop more specific end-user and configuration applications using the existing tools.
· Practices which transcend applications. We change the framework functionality, and re-factor the framework [Foo95,Rob96].
· Functionality specific to a particular application. Depending on its nature, we change the functionality of the repository, the framework or both. For instance, some applications require a simple procedure to automatically generate standard reply letters based on repository information. In this case, we store extra behavior in the repository. If a new external tool is required, we usually add a component to the framework.
· Technological needs and opportunities. We use subframeworks for components that are strongly coupled to technology, such as the persistency layer and the document storage. These subframeworks can easily be replaced.
· Additional functionality, to support new types of applications, e.g. Internet applications. This way, the framework explicitly supports evolution and re-use at different levels [Til96].

4. Framework components

We broadly classify the framework components in the following groups:

· the repository component: the object model, system objects and end-user objects
· front-end components: end-user and configuration tools, such as forms, document viewers and the object model editor
· back-end components: the persistency component, document-storage, background process managers, the E-mail gateway and the OCR-engine
· Central components: the central core component, the application manager, the document session manager, the authorization rule-base, and the background process manager board, the action rule manager and managers for various other system objects. We refer the reader for a full description of these components to http://www.argo.be/intranet/aiv and to the enclosed screendumps (in particular for the next section).

5. Bootstrapping process

The bootstrapping process is driven by our observations that most configuration tools essentially manage specific types of objects in the repository, to which we need to add some extra functionality, validation, consistency rules and specific caching.

Often, an alternative plug-in property editor is all that is required to manage these objects easily through end-user tools. Examples To bootstrap the system, we initially developed a hard-wired object model editor. About one year into the project we were able to configure end-user tools for accessing the object model. Now we are enhancing this application to support the functionality offered by the new persistency component. While access to in- /outbaskets (these keep track of incoming and outgoing task assignments) is still managed through a dedicated application, baskets are defined through regular forms. We applied a bit of white-box re-use to make the query editor fit in the property editors scheme, and we modeled the basket definition explicitly. Its properties include a query expression specifying the basket's contents.

Most of our latest tools are configured applications, or are developed with configuration in mind. Examples of the former are the object behavior and action rules applications, and the background process administration tool to be developed in the near future. The graphical workflow and object model editors, on the other hand, are designed as alternative property editors for use in forms. Reflection In this bootstrap process we are increasingly using existing tools to build new tools, typically configuration tools.

Reflection [Foo96] is a useful technique to support this kind of approach. Reflection, however, is typically considered too obscure and difficult, and is often associated with the run-time behavior of applications. Yet, given the framework approach, this idea of bootstrapping is very natural: the system already contains so much tools to build applications, that we would usually take a step back by not trying to re-use this functionality. The locality of change that can be achieved with reflection is also a very important aspect.

6. Technical specifications

The framework is implemented in a Windows / Novell / Oracle / cc:Mail / Internet environment and is based on the class library in the VisualWorks\Smalltalk 2.5 environment. Most of the tools have been developed in Smalltalk, but some external components have been integrated:

· the ODBC-interface used by the persistency component
· desktop applications (such as Word, WordPerfect and Excel), by means of DDE
· Kofax image rendering and scanning library, transparently integrated in the framework
· Calera OCR module · E-mail (SMPT- en POP-3 protocol) based on the sockets communication library
· Mires full-text-indexing engine
· Jukebox technology (HP), driven by the framework document caching manager.

The framework itself comprises about 1000 classes and has been implemented by a team of 5-6 core developers.

7. Using the framework tools

We give a short overview of a documentation center application, a decision procedure application and an incoming mail registration application. How the user interacts with applications and the various tools is illustrated by means of several screen dumps (in annex in PDF-version)

Documentation center

With the documentation center application, users manage authors, publishers, and several types of books, periodicals, editions and other kinds of documentation, including the framework documentation. We offer two environments. Librarians for administration purposes use the first one. A simpler subenvironment is destined for regular end-users.

Some documentation exists on paper. In this case we maintain references to the physical location. Otherwise the documentation is made accessible on-line. This may include scanned documents.

Part of the documentation is also available on the Internet, in an alternative representation, such as PDF. Indexing and querying rely extensively on thesaurus and full-text-indexing. Librarians manage distribution lists to notify users automatically about new documentation. They also keep track of loans.

In addition to the standard printing facilities, librarians print documentation in ISBD-specific format. The object model is complex. To reduce complexity and make the application more dynamic, we use some explicitly typed objects and associations, e.g. typed "replaces" and "is an addendum to" associations between documentation. We can do this easily since our tools query both meta- and object-level information, which gives us considerable flexibility to (re-)design the model.

Workflow processes

Argo does not perceive workflow as a means to single-mindedly automate existing processes. Too many exceptions exist in the office to prescribe the flow in most cases. Although some processes do follow a strict plan, in general, users need the freedom to deviate from this plan. And in some cases no fixed plan exists at all.

The central board decision procedure formally describes the process for submitting a dossier for approval by the central board, starting from a department's manager. The preparation phase prior to this process is however subject to each department's culture, which in some cases is very team-oriented, and in other cases is inspired by a hierarchical structure. The general manager has the right to advise on the dossier, but he is not part of the actual process flow. To keep him informed upon upcoming dossiers, notifications are sent when the process enters a particular phase.

When decisions result in a concrete action plan, management needs to check when and if the goals are actually met, by monitoring the state of specific data, documents and processes. Use of ad-hoc processes is illustrated by the application for handling incoming (paper) mail. Users register these documents in the system and send appropriate task assignments to handle the dossiers. The actual process flow depends on the contents of the documents.

8. Conclusion

The main goal of the framework design was a small kernel of generic components acting dynamically upon the repository. Hard coding was to be avoided as much as possible. Initially, we focused on achieving this goal for end-user applications. We developed hard-wired administration and configuration tools to help us bootstrap the system. As the framework evolved, and the tools became more flexible, we re-used components of end-user tools in some of our administration tools. In a third phase, we started to replace some of the hard-wired tools with applications configured in the system.

The framework configuration functionality will be enhanced, in order to further increase the expressiveness of the tools, and to allow end-users to adapt the tools to their own needs even better, thus obviating the need for developers to a larger degree. Some additional components are being developed, most notably to make the repository Internet-aware. The main component will act both as a client of the repository and as a generic Internet application server, using a VisualWave-based approach. This will empower Argo to develop Internet applications through modeling and configuration too.

At first sight, one might wonder whether this approach does not make the design more complex, or affects performance negatively. We do not feel that the design is more complex. In fact, the framework goals help us focus more clearly on the responsibilities of the various components, in particular with regards to re-use and evolution. As is often the case, reifying implicit responsibilities actually makes the design simpler. And the bootstrapping principle is an additional asset in validating the design. Although we initially encountered some performance problems, these were not essentially related to the approach as such, but to the use of a persistency component that was not really suited for our purposes.

We must be aware, however, that the flexibility of the system allows end-users to build e.g. equally good and bad queries. Proper training is of primary importance, as is the need to initially hide some of the more complex elements of the tools from novice users. After three years, we are convinced that the approach is a valid one. End-user applications can be developed iterative and incrementally, even interactively. We do not code (apart from some custom scripting) and we do not generate end-user applications. Neither do we use or need throwaway prototypes. Instead, we build increasingly complete specifications of end-user applications. These specifications are available for immediate execution. In this way, we help close the gap between specification, development and use of the applications.

Some Observations - Keepers and don’ts

· We found inspiration in the Framework and Application Prototyping teams as described by Goldberg and Rubin. We have a special Framework team, that is highly interacting with what we call the Configurator team, that uses the framework to build end-user applications. These teams work together intensively (and recently we decided to have two members working on both teams) at scenarios, both real and imaginary. Both teams work with the end-user in workshops.
· In every new version we pay more attention to consistent use and documentation of idioms, coding and design patterns and find that very useful and rewarding.
· Project teams are organized along patterns for organization, in particular James O. Coplien's Generative Development Process Pattern Language. We have been working along these lines for more than a year and at the end of the former project, and start of the new, we organized a two days workshop with the team, evaluating does and don'ts with this pattern language. During that workshop additional patterns were selected from related languages.
· Recently we started using Alistair Coburn's Project Risk Reduction Patterns, along with his Medical catalog of Project management patterns.
· In all our projects we use workshops. Rather than using classic interview techniques, we make use of creative techniques (metaphores, story telling, system envisioning, scenario writing, the solution-after-next concept and games)
· Our approach involves active participation of users, not only in analysis, testing of releases, but in the actual writing of end-user documentation and in the training of their peers. This is related to the nature of our framework and the way end-user applications are build and tuned. Lacking in our approach is a good case for this way of working - even with several other organizations (in different business) interested in the use of the framework. We feel rather isolated among more traditional project managers. We had several severe clashes with a company involved in the development. We miss experience (and documentation) in defending our way of working and leading an OO project, with investment in reuseable assets and flexibility.

References

[Dev96] Martine Devos and Michel Tilman, Design and Implementation of a Business Modeling Framework using Smalltalk, Object Technology'96, 1996

[Foo95] Brian Foote and William F. Opdyke, Lifecycle and Refactoring Patterns that Support Evolution and Reuse, Pattern Languages of Program Design, Addison-Wesley, 1995

[Foo96] Brian Foote and Joseph Yoder, Evolution, Architecture, and Metamorphosis, Pattern Languages of Program Design, Addison-Wesley, 1996

[Rob96] Don Roberts and Ralph Johnson, Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks http://st-www.cs.uiuc.edu /users/droberts/evolve.html

[Sto94] Frank Stowell and Duane West, Client-Led Design, A systemic approach to Information System Definition, McGraw-Hill, 1994

[Til96] Michel Tilman and Martine Devos, Object-Orientation and Evolutionary Software Engineering, Position paper for the OOPSLA'96 Workshop on Object-Oriented Software Evolution and Reengineering, 1996

Biographic information: Martine Devos has been IS manager at Argo since 1992. Previously she has worked as teacher in Economics, teaching assistant in Computer science, as technical consultant and project leader to the Belgian Minister of Education, the Civil Service and Argo. She is chair of the IS managers workgroup of MOVI, representing all Flemisch semi-government organizations. As IS manager she initiated, and coördinated the development of mentioned framework and several applications using it. Her special interests are framework development, the use of patterns in organization, the human side of IS and facilitating workshops. Michel Tilman graduated in Mathematics at the Brussels Free University in 1981, in the field of Functional Analysis. He joined the Mathematics Department at the Faculty of Applied Sciences as teaching-assistant, where he was involved in research on Non-Archimedian analysis. In 1985 he joined the new Programming Technology Lab at the Faculty of Sciences, where he specialized in models for reflective, parallel object-oriented programming systems. In 1989 he became research manager at SoftCore, a company developing electronic document and workflow management systems. He led several research projects regarding the use of object-oriented frameworks and application servers in cooperative work and workflow. He has been involved in the Argo project as Senior system architect from the start. Papers of related interest: Martine Devos and Michel Tilman, Design and Implementation of a Business Modeling Framework using Smalltalk, Object Technology'96, 1996 Martine Devos and Michel Tilman, Business Modeling using an Object-Oriented Framework and Meta-Repository Architecture, OOPSLA'96 Demonstrations, 1996 Martine Devos and Michel Tilman, Object-Orientation and Evolutionary Software Engineering, Position paper for the OOPSLA'96 Workshop on Object-Oriented Software Evolution and Reengineering, 1996 Martine Devos, System Envisioning, Position paper for the OOPSLA'96 Workshop on System Envisioning, 1996 Martine Devos and Michel Tilman, Application development using a Meta-repository based Framework, ECOOP '97 Demonstrations, 1997 Martine Devos, A repository-based Framework approach to application development, SAI seminar on Object oriented frameworks for business, Katholieke Universiteit Leuven, 1997

Developing Successful OO Frameworks

Einar Dehli
Vice President, Process Owner, Technology, Computas AS Phone: +47 67 54 11 11, Fax: +47 67 54 10 11 E-mail: ed@computas.no, URL: http://www.computas.no/ Address: Leif Tronstads plass 6, P.O. Box 444, N-1301 Sandvika, Norway

First, some general background information is in place, to set the context: Computas AS (named Computas Expert Systems A.S in the old days) is a major competence centre in Scandinavia for knowledge technology, object technology and user interaction. The major part of Computas services is contract work for large client organisations, primarily systems development with a knowledge management focus. Key success factors are skilled and motivated personnel, efficient project methodology, and a growing library of reuse components and frameworks. Efficient maintenance and development of frameworks and components are crucial for the continued success and further growth we aim at.

The company culture we have managed to build in Computas is, if not unique, far from common-place. As one of three co-founders back in 1985, it has been an exciting experience to take part in building a company like this from scratch. Computas' mission is to develop innovative business solutions for knowledge management. Apart from a strong technology foundation, our approach has been to foster a stimulating work environment where employees (and client participants) can combine creativity, skills and technology with the plain hard work which is always required to actually deliver anything useful.

The company currently employs 120 people. More than 90% have university degrees and background in advanced computer sciences, artificial intelligence, expert systems, physics, philosophy and psychology. Employees are seen as the main asset of the company. In a hot job market for IT specialists we still manage to recruit and keep the "best" brains, without offering excessive salaries. In the last three years we have had an annual growth rate of 50%, with only 2-3% turn-over. And, profit margins have been positive. Thus, a success story so far!

Computas has a flat, process oriented organisation, with all employees in a single resource pool. Assignments of roles and responsibilities are temporary, both within projects and in the organisation. To stay focused - doing systems development from a knowledge management perspective, and maybe also just to be a bit different - our system developers are all entitled Knowledge Engineers.

Most employees spend almost all their time on fixed price or hourly paid client projects. But, since we have managed to get acceptance for running most projects in our own offices, communication with framework specialists, mentors and between projects is greatly facilitated. This has been useful in early phases of our framework development, but certainly does not scale up to the level of reuse we plan for in the coming 2-3 years.

In addition to using various third party frameworks, like G2 and other AI-oriented solution frameworks, we maintain two major internal object-oriented frameworks: BriX, based on COM, C++/Java and other Microsoft technology, for complex technical domains, and Expert WorkFlow (internally denoted Sara), based on Smalltalk. Solutions based on the Expert WorkFlow framework provide users with active support in executing business processes. Expert WorkFlow will be presented as an experience report at this year's OOPSLA (see extract below), and we have also been invited to present a demo.

BriX

BriX is an architecture based on COM. BriX is the result of more than 3 years architectural effort in the 100 person+ Nauticus project, developed by CX and DNV for DNV. DNV is an independent, autonomous foundation with the objective of safeguarding life, property and the environment. With 300 offices in 100 countries, DNV serves customers in the maritime and process industries worldwide. (I'm sorry, but it turned out I didn't have any suitable BriX documentation readily available in electronic form. Below follows some plain text extracts from a PowerPoint presentation, just to convey a flavour of what BriX is all about).

About Nauticus

Product Model:

- Detailed model of all parts of a ship relevant for classification-related tasks
- Core model: objects with conceptual relevance, other representations derived from the core model (views)
- The same model throughout the ship's life-cycle
- Storage of relevant product-related information & knowledge
- A rich core representation can be idealised into task-specific representations, still preserving coherency between different representations

Why a Product Model?

- Immediate, easy access to relevant information.
- Product logic as part of the product model supports complex tasks
- Same model, but different "views" used in different tasks
- Faster analysis, more qualified judgements
- Experience storage and retrieval
- Consistent, up-to-date information
- Consistent use of terms and concepts
- Knowledge management & development in a complex domain

Nauticus Domain Characteristics

Need for horizontal integration across a wide domain
- Integration with external systems
- Need integration at different levels:
- Conceptual integration (unified conceptual domain model)
- Domain Object model (business objects as instances of domain concepts)
- Tool integration

- User interface integration
- Work-flow integration
- Infrastructure integration
- Geographical distribution

What is needed

- Desktop as a generic "task container"
- Can host any ask and the components required by the task
- Tasks as self-contained components
- Declares what tool components it needs in order to execute
- Common Information Services
- A set of task-independent services for uniform access to common information
- Architecture Vision
- Task-oriented system
- Allow dynamic configuration of tasks
- Dynamic binding of components at run-time:
- Tools are small components which are bound into the user's task on demand
- "Lego components": Standardised interfaces makes composition of tools easier and faster
- Purification of architecture roles makes architecture components replacable and the architecture scalable and extensible

What are components?

Software components:

- Self-contained objects
- Designed, managed and distributed as a separate image (DLL, EXE, Java class)
- Simple   Complex   Industry-standard interfaces for composition and integration
- "Software bus" Examples: The PC   Stereo   MIDI (digital music instruments)
- Cheap, replacable component architectures
- Makes solutions simpler to build and customise

Why components?

- "LEGO-like"standard, "plug-in" interfaces
- Scalability: Many small components loaded when needed by the task, not one big application
- Flexible task-centric system
- Faster to build new applications, can reuse existing components
- Complex components assembled from simpler components
- Assembly vs. "handicraft"
- Component Object Model COM = Software Components + Distributed object technology

Basis for a system-wide object-oriented software architecture COM/ActiveX Keywords COM:

- Software components

- Distributed objects
- Local-remote transparency
- System object model
- Encapsulation
- Implementation reuse
- Polymorphism
- Language-independence
- Originated on Windows
- Emerging on Mac, Unix and mainframes
- ActiveX and OLE:
- A set of COM-based frameworks (client and server) supporting highly dynamic applications

A family of COM-based frameworks:

- Visual Editing
- OLE Controls etc.
- Basis for Microsoft's "Internet architecture"
- The COM Architecture Local-remote transparency
- Component Objects
- A COM object may implement several interfaces
- An interface has a fixed set of methods
- Reference counting

Polymorphism:

- Two objects may implement the same interface differently
- Implementation reuse: Aggregation or delegation
- Complete encapsulation
- COM and Object-Oriented Design

COM enforces two object-oriented design-principles:

- Program to an interface, not an implementation
- Favor object composition over class inheritance [for implementation reuse] ("Design Patterns", Gamma et. al 1995) Result: Reusable, "black-box" software components

- Tight integration with loose coupling
- Focus on role models rather than object models
- Roles map naturally to COM interfaces
- Types specify a set of interfaces (mandatory/optional) that classes of that type must implement
- Objects can easily adopt new roles
- Simpler, more extensible object-oriented designs

Nauticus Architecture Overview - Desktop Architecture

Desktop: A container of tasks
A task declares to the desktop what tools to use

Explicit representation of tasks:

- A task may contain subtasks
- A job is a container of tasks
- Tasks and tools linked into the desktop on demand

Desktop Architecture

- Jobs as assemblies of tasks
- No need for applications!
- Jobs designed through a workflow management system and stored as "job templates"
- Jobs are mobile: can be mailed to co-workers
- Common Information Services, Meta Model
- Uniform access to a hybrid set of data sources
- Queries
- Browsing
- Meta model = grammar for the object models
- Concept, e.g. Damage   Type, e.g. Corrosion   Instance, e.g. recording of Corrosion on a particular ship

Different information providers may implement their objects differently, but they will be exposed through the same interfaces Software Factory Environment for building and assembly of software with maximum efficiency and quality

Development Environment Setup

- Development tools/utilities/"wizards"
- Frameworks (e.g. for tool development)

Documentation Experience summary –

The good news

- COM promotes good object-oriented design principles
- COM simplifies pattern implementations
- A language-neutral object model (C++, Java, Visual Basic, Delphi, Smalltalk)
- Location transparency
- Already a large COMponent market (e.g. Active controls)
- System object model on Windows
- Support from an increasing number of development tools
- Java and COM - a good marriage
- Teams work more independently
- Enables a new breed of software: componentised, dynamic, task-driven

Experience summary –

The bad news

- Learning curve for C++ programmers (C++ object model lacks the interface abstraction)
- Debugging is more difficult (due to the design principle enforced by COM)
- Components need to be thoroughly tested before published

Advice:

Employ "zero defect" principle for component development

Summary

Distributed components ready for prime time!
When designing a system, focus on clear separation of architecture roles
Integration of components through interfaces - reuse existing interfaces whenever possible

Expert WorkFlow

An object-oriented framework with components for building knowledge based workflow systems (extract from OOPSLA Paper). Our aim with Expert WorkFlow has been to build knowledge based workflow systems that support the execution of business processes. The processes are modeled as routines consisting of tasks. The user performs the tasks, while the system controls and manages the workflow execution. Thus, the main purpose of these systems is to ensure correct performance and increase efficiency by providing easy access to relevant information.

After having developed knowledge based and expert systems for 12 years, our experience is that expert system tools (often object-oriented) are too specialised and closed for a lot of problems. The same applies to many tools supporting workflow. For instance, traditional expert systems are not adequate in domains governed by complex sets of laws and regulations. However, by implementing knowledge based and workflow mechanisms in a general purpose object-oriented programming system, we have found a powerful way of efficiently developing intelligent workflow systems. The power and efficiency come from the combination of and tight integration between expert system technology, object oriented technology and the focus on direct computer support for the user organisation's business processes that workflow systems have.

The general purpose object-oriented programming environment we are talking about is, for good reasons, Smalltalk. Since we started developing the first system of this kind in 1994, we have built several systems with our framework. In the OOPSLA presentation we present examples from two of these.

The first called Helene , is a support system for officials in the child support reclaiming office of the Norwegian National Insurance Institution. The second called BL96 , is a support system for handling the processes of criminal proceedings in the Norwegian Police Department and the Public Prosecuting Authority. Both systems are currently in full operation and showing very good results.

The Helene system has about 65 users and the BL96 system about 1500 users at 15 different police districts in Norway, increasing to 8000 users at all 54 districts by the end of 1998.

Starting from Helene, the framework has evolved in our development team. The framework consists of some black-box components and some white box elements — the glue that binds the components together and makes a complete system work. We will give a brief overview of the framework, describe the key components, present our main idea that concerns user interaction in a system of this kind and comment on our experiences with the implementation tool used.

There are three main views of the framework architecture: A business knowledge view, a more technical application view and a systems building view.

The technical application view is a five-layer architecture of an Expert WorkFlow application. In a client/server setting, these layers can be partitioned between the clients and a server, or several servers. As a consequence, a system built with the framework can be implemented as a traditional fat-client application with only data stored on a server, a single application server solution (two- or three-tiered) with the application partitioned between the clients and the application server, or a multi-tiered distributed solution with the application distributed across the clients and several interacting application servers.

The systems building view can be regarded as a synthesis of the two other views. The separation between the framework and the application specific parts of an Expert WorkFlow-system can be thought of as a cartridge model , like a Nintendo player with different cartridges. Building an application based on the Expert WorkFlow framework thus consists of building the cartridge .

The elements that must be built are the domain model, the business processes, the rule base and the application specific views. These elements can be regarded as pluggable into the framework. In a real world situation this is somewhat simplified, since you may want to make modifications to some elements in the framework when building a specific application. This is done through classical object-oriented subclassing at key points in the framework. You may also want to integrate other components into the application, e.g. for communication with some mainframe system. The aforementioned Helene and BL96 systems both communicate with mainframe systems, but using different communication components.

The process of building the cartridge is, in knowledge engineering words, the process of knowledge acquisition and representation. We also regard as knowledge acquisition the object-oriented analysis task, i.e. building the domain model. With the framework at hand, it is possible to do this process in an incremental and very rapid manner. As soon as the first few processes and a part of the domain model are defined, they are instantly implemented in the system and act as a basis for discussions and further development.

One of the key strengths of the Expert WorkFlow framework, inherited from the knowledge engineering tradition, is that the application specific business processes and rules are data and not code. This means that business processes and rules are defined and maintained separately from application code. For the systems developed with the framework, this quality has been one of the most valuable, and it gets even more valuable over time when the system is in operation and continuous maintenance.

To some extent, also the definition of the domain model is represented as data in a meta model. However, objects with application specific behavior require classes and methods, i.e. code, to be added. In fact, even the definitions of some of the simpler user interface views are separated out as data, and the contents and layout of these views are built dynamically from that data at runtime. Editing the processes and rules is done with authoring tools.

Object oriented CASE tools can be used as authoring tools for the domain model. The key strength of the Expert WorkFlow framework is the combination of and tight integration between object technology, expert system technology and workflow technology. This integration is obtained by implementing expert system technology and workflow technology in a dynamic and general purpose object-oriented programming language

Smalltalk.

The workflow component is implemented in a relatively straight-forward manner by representing process definitions, activities, active processes and activity instances as Smalltalk objects. This is also the case for the rule reasoning component. The rules and the processes are data to the system. Here Smalltalk's flexible ability to store message names as data and perform them at runtime is used to keep the tight integration between the rule and process data and the object model of the domain.

Smalltalk has proved to be a very well suited tool to implement the desired mechanisms. The level of abstraction and the dynamic capabilities of the language are two key reasons for making this relatively easy and straight forward.

Other dynamic high level object-oriented languages, e.g. CLOS or Self, could also have been used, but they lack the same level of maturity in client/server commercial tools, professional development environments, extensive libraries for integrating with other systems etc.

Einar Dehli is a co-founder of Computas AS, and is currently Vice President and Process Owner, Technology. He graduated with a Master's degree in Computing Sciences from the Norwegian University of Science and Technology in Trondheim in 1979. For the last 15 years he has been actively involved in a broad range of activities related to knowledge based systems design and object oriented software development. His experience range from company management roles, most recently Process Owner, Human Resources, to systems development and project management. He is an experienced Lisp and Smalltalk developer, with experience from national and European research projects, as well as industrial client projects. On behalf of Computas Management he has closely followed the development of the BriX framework over the last three years. His primary responsibility now is Project Manager for the next generation Expert WorkFlow framework, which includes porting the software from Digitalk Visual Smalltalk Enterprise to IBM VisualAge for

Smalltalk, and componentisation for better integration with other Computas frameworks based on ActiveX/COM/C++ (BriX) and Java/Corba. He also manages a pre-project for establishing a next generation corporate memory support system in Computas.

A Case Study Of Object-Oriented Development That A Framework Is Developed And Is Used

Takeo Hayase
TOSHIBA Corporation

1. Introduction

This report describes the process that a framework for Automatic Teller Machine, the ATM framework, is developed and is used, and explain the feature of this framework.

High reusability is a major purpose of Object-Oriented (OO) development. Using a framework as means of this purpose has been done. If an appropriate framework exists already, it is used. On the other hand, if it does not exist, it is necessary to develop original framework. However, there are not many examples that a framework is developed and is used.

We describe the ATM framework as a case study of OO development that a framework is developed and is used.

2. Target Application

ATM is a machine dealing with cash in financial institutions. An application for ATM has following characteristics.

(1) Specifications are different every financial institution
(2) Specifications are changed frequently
(3) Comfortable operability is expected
(4) High reliability is necessary
(5) Opened architecture is expected.

In a normal case, ATM naturally deals with services for a customer and for a person in charge. In addition, in an abnormal case it deals with them in order to secure reliability. This case holds large portion of specifications.

An application for ATM is required to control devices in order to deal with services quickly. Therefore, specifications are very complicated to advance services smoothly even if anything happens by any timing. Complicated specifications are described by using state transition diagrams mainly. State transition diagrams extend over several thousand pieces. Total size of source code exceeds about one million steps. In this way, an application for ATM is very complicated and large.

3. Requirements and approaches

Requirements to develop an application for ATM are as follows.

(1) Promoting development efficiency in complicated and large systems

(2) Improving reusability
(3) Securing performance in real-time control systems

Approaches for these requirements are described respectively.

(1) Separating with common specifications and specifications that are not common We separate with specifications that depend on a difference of financial institutions and services, and specifications that do not depend on it. By this, complicated and large specifications are represented easily. Common specifications are implemented reusable components.
(2) Developing the ATM framework. Using this framework can achieve advanced reusability. This framework has fundamental mechanisms so that ATM application is worked.
(3) Using multi-process / multi-thread environment. Using this environment can control devices concurrently. Performance of ATM is secured by the ATM framework that contains this environment.

4. Developing and using the ATM framework

We developed the ATM framework which contains the fundamental behavior of ATM and which has multi-process / multi-thread environment.

4.1 Analysis

We analyzed ATM domain and listed objects. We divided four categories (Service, Media, Unit, and GUI), and defined association between category. These categories have following role.

<Service> "Service" category has a role to represent how services flow. Objects in this category have a scenario every service. Objects in this category send messages to objects in "Media" category and wait for messages from objects in "Media" category. This category contains objects such as "Payment", "Deposit", and "Transfer".

<Media> "Media" category has a role to represent how media behave. Media are logical resources in ATM. Objects in this category receive messages from objects in "Service" category, send messages to objects in "GUI" and "Unit" category, and send messages to "Service" category. This category contains objects such as "Card", "Passbook", and "Password".

<Unit> "Unit" category has a role to represent how units (devices) are controlled. Units are physical resources in ATM. Objects in this category receive messages from objects in "Media" category and send messages to "Media" category. This category contains objects such as "Card", "Passbook", and "Receipt".

<GUI> "GUI" category has a role to represent how GUI is showed and changed. Objects in this category have a scenario every service. Objects in this category receive messages from objects in "Media" category and send messages to "Media" category. This category contains objects such as "Payment", "Deposit", and "Transfer".

4.2 Design

An OO specialist and a domain analysis specialist decided the architecture. Two project members discussed what kinds of classes contain in the ATM framework. This Framework is refined while developing part of the ATM application as a prototype. We developed three times of the prototype. We mainly designed mechanisms for message communication between objects on multi-process / multi-thread environment and for change state of an object. The behavior of ATM is described as state transition of an individual object. We expanded "State pattern" that is design patterns.

4.3 Implementation

We mainly implemented mechanisms for message communication between objects and for state transition on multi-process / multi-thread environment by C++ language.

4.4 Use

The ATM framework is used black-box components. An application for ATM consists of classes in the ATM framework and classes that inherit a class in this framework.

5. Feature of the ATM Framework

The ATM framework is divided into three parts. This framework contains about 80 classes.

(1) Common classes
(2) ATM domain classes
(3) General-purpose classes

(1) Common classes

These classes are commonly used in control systems. The common characteristics of objects in control systems are to react to messages, to act anything, and to send messages to the other objects. On the other hand, the difference characteristics of each object in this system are to how those behave. Therefore, we divided a class to communicate messages and a class to define behavior. The former class is called "Actor". The latter class is called "Mechanism". "Actor" class has thread facility. "Mechanism" class receives messages to transmit from "Actor" class. The derived class of "Actor" is made every category (Service, Media, Unit, and GUI). This is because communication partners are limited every category. The derived class of "Mechanism" has a specific mechanism. For example, "STD" class has a mechanism for state transition, and "Transaction" class has a mechanism for transaction notification.

(2) ATM domain classes

These classes are implemented to represent the feature of ATM. ATM domain classes represent specifications that do not depend on a difference of financial institutions and services. As an example of common behaviors, there is a class called "Card" and "Passbook" in "Media" category. The behavior of "Card" class is different by financial institutions or services. However, there is actually the common behavior that does not depend on financial institutions and services. The common behavior of "Card" class is as follows. (a) Sending a message to take in a card to "Unit" category. (b) Interpreting logically the magnetism physical information of the card, which has been sent from "Unit" category. (c) Sending the new logical information of the card to "Unit" category. (d) Sending a message to drain the card to "Unit" category.

(3) General-purpose classes

These classes are primitive such as a list, a queue, or an array. We avoided using the class library of ISV (Independent Software Vendor) and developed positively these classes. This is because we make the responsibility clear and establish original interfaces.

6. Conclusion

This report describes the process that the ATM framework is developed and is used, and explain the feature of this framework. The process that ATM framework is developed and is used, is as follows.

(1) Analyzed ATM domain and extracted four category (Service, Media, Unit, GUI), and defined association between category.
(2) Designed mainly mechanisms for message communication between objects and for Change State of an object.
(3) Implemented designed classes on multi-process / multi-thread environment by C++ language.
(4) Used the ATM framework as black-box components.

The ATM framework consists of three components.

The feature of these components is as follows.

(1) Common classes are used commonly in control system.
(2) ATM domain classes do not depend on financial institutions and services.
(3) General-purpose classes are developed newly to make the responsibility clear and to establish original interface.

By using the ATM framework, the period of developing ATM application decreases largely after the second development of ATM application. This is because the first development of ATM application contains the ATM framework. "Common classes" of the ATM framework are used commonly in control systems. These classes will be refined by adapting other control systems.

Takeo Hayase received the Master of Engineering degrees in mechanical Engineering from Sophia University, Tokyo, in 1992. From 1992 to 1996 he has been with TOSHIBA Corporation, where he has worked on object-oriented technology(OOT) at System and Software Engineering Laboratory. He is now working on OOT at System Integration Technology Center. His current research interests include architecture, class library, framework, and design patterns. Takeo Hayase System Integration Technology Center, Toshiba Corporation, 3-22, Katamachi, Fucyu-shi, Tokyo, 183 JAPAN tel: 011-81-423-40-6368 fax: 011-81-423-40-6013 e-mail: hayase@sitc.toshiba.co.jp ************

Framework Development and Delivery Position Statement

Paul Dyson
University of Essex

 For the past three years I have been working towards a PhD in the field of software architecture, specifically looking at the development of software frameworks. During this time I've produced a number of small example frameworks in order to investigate some of the technical and human issues faced by framework developers, and have worked with part of the development team of a large, commercial, financial-services framework - observing and reflecting on the development process and actively participating in the documentation of two key sub-frameworks for transaction-processing on a mainframe server (referred to as the Transactions framework) and error handling.

In both of these, very different, environments I was interested in how architecture impacts on framework development. In the case of the example frameworks I developed as part of my research project this mainly focused on how a framework's architecture is directly inherited by all of the systems instantiated from it.

In order to develop a good framework we need to look at the required non-functional characteristics of potential instantiations as well as their functional requirements. In the case of the financial services framework, the impact of architecture was most evident in the way that the designs of the sub-frameworks were influenced by the architecture of the wider framework, at least as much as by the need to implement a good abstraction of the potential instantiations. In order to develop a good framework we need to consider the requirements for the framework as a system in its own right as well as the requirements for it as an abstraction of all its potential instantiations.

Through the work I carried out documenting the financial-services framework I became very interested in how we document frameworks – there's no point developing a "good" framework if no-one can understand or use it.

For the Transactions framework I produced three separate documents: a user's manual, a maintainer's/developer'as manual, and an architectural overview. Although these three documents shared a lot of information, their formats, use of examples, and focus were very different.

In order to document frameworks well we need to separate out the concerns of the developers from the concerns of the users. I was also very interested in the way the financial-services framework was delivered - the developers of each sub-framework worked directly with its users and took responsibility for some of the instantiation themselves. In this environment the user's could tap the developer's expertise directly and the developers could see how the user's coped (or didn't) with their framework. In order to deliver frameworks well we need to consider that, sometimes, good documentation isn't enough.

Underpinning both of these areas of interest is the belief that we need to recognise, record, and learn from experiences in the development and delivery of good frameworks. There seems to be a distinct lack of concrete guidelines for framework developers on how to find good abstractions, how to implement these abstractions as good frameworks, what aspects of the framework to document for which audience, etc. Without some sharing of experience and knowledge we will find it hard to move on from the current state of practice where many frameworks are discarded because they are completely unusable. Impact of architecture and domain analysis on framework development

I tend to look on a framework from three related, but distinct, perspectives: as the implementation of a generic architecture, as an abstraction of a domain of systems, and as a system in its own right.

Frameworks as abstractions of domains of systems — A framework can be instantiated to a number of systems in a particular domain. In the case of GUI frameworks like MacApp and MFC the domain is applications that run on their respective operating systems; for the Transactions framework mentioned above the domain is server-side transactions in financial-services applications.

A good framework needs to be a good abstraction of all the systems in its domain. When we start out developing a framework we usually abstract from a number of example systems (either actual, visualised, or known from experience), and develop the framework as an abstraction of these systems over a number of iterations, but the framework isn't only going to be instantiated to these systems - they already exist so what would be the point

This is where domain analysis methods play a role. Domain analysis considers domains of systems to identify the scope of the domain, the language of the domain, and the nature of the systems in the domain [Arango&Prieto-Diaz91]. Domain analysis informs, and is informed by, the example systems. If we only consider the examples then they effectively define the domain - we will only be able to instantiate the framework to systems that are exactly like these examples (potentially, only the examples themselves). If, however, we use the examples as a starting point and feed domain analysis information into the framework development process we can hope to develop a framework that is representative of the whole domain rather than just some subset of systems used as examples.

Frameworks as implementations of generic architectures — Domain analysis traditionally focuses on the function of systems - a library system, a hotel system and a car-rental system can be said to be part of a domain of systems that lend or hire resources to customers. Software architecture considers the structure of a system rather than its function. We have seen that the choice of architecture for a system has serious implications for its non-functional characteristics such as size, performance, robustness to certain changes [Dyson&Anderson97], ease of integration and safety [Shaw95].

A domain-specific architecture is a structure that should suit a particular group of systems - e.g. the adaptive intelligent systems (AIS) architecture for controlling autonomous robots, mobile software agents, and intelligent patient monitoring systems described in [Hayes-Roth+95]. This architecture has a number of non-functional characteristics, such as fast processing of sensor inputs and rapid switching of contextual information, desirable in these types of systems. It is used as a reference when developing the architecture of a specific AIS - the article describes its use in the development of three different office-automation robots.

We can view a framework as an implementation of such an architecture. The architecture describes the components of a system, what their roles and responsibilities are (without saying exactly how these responsibilities are carried out), how they relate to each other, and how they communicate. This is essentially what a framework does. We inherit a framework's structure when we instantiate it so we must also inherit all of its non-functional characteristics. This means that, as well as a good abstraction of a functional domain, a framework must also be the implementation of a good architecture for the systems in this domain.

Frameworks as systems in their own right — As well as an abstraction of a domain of systems and the implementation of an architecture, a framework can be considered to be an incomplete system, one where we fill in some "gaps" to make it complete. As such we need to consider the design and implementation of a system in its own right. Domain analysis combined with abstraction from examples will help to reveal the key abstractions in the domain (abstractions such as Transaction, PersistentObject, MessageToServer and ServerBuffer in the server-side transaction-processing domain) and explicit architectural consideration will help to reveal the distribution of responsibility and nature of communication between these key abstractions. But a nontrivial framework needs to be implemented with more than just the key abstractions in the domain. Consider the Transaction abstraction.

In a system where communication between the client and the server is carried out synchronously the Transaction has some state information associated with it. The State pattern in the design patterns book [Gamma+95] describes the benefits and limitations of implementing state as a separate object and so helps us to develop the framework as a system in its own right - we introduce a TransactionState class not because it is a key abstraction in the domain of server-side transaction-processing, or because it gives us the non-functional characteristics we require, but because, in this instance, it gives us a "better" system design. Similarly, we know that, in a synchronous and single (client-side) threaded environment, we should only have a single point of communication between the client and server: ServerBuffer. This is a good application of the Singleton pattern. It is in the development of a framework as a system in its own right that I see design patterns having their greatest input: they don't help us to find good domain abstractions, they help us to implement them better and so lead to a framework with a good design.

Tying the perspectives together

These three perspectives are distinct but very closely related. The perspective of a framework as an abstraction informs the perspective of the framework as an implementation of an architecture, and vice-versa, because we are dealing with object-oriented systems and such systems are very explicit about their structure. However, these two perspectives can be in tension with each other. The Transactions framework can be instantiated both to a transaction querying the balance of a customer's account and to a transaction transferring funds between two customer accounts. These transactions are functionally similar (a transaction is encoded and passed to the server, the server processes the transaction, the server response is processed by the client, and the client's local model is updated to reflect changes in the server) but have very different non-functional requirements.

In the case of the balance query, speed of processing is the major concern because we don’t want to keep the customer hanging around on the phone, at the desk, or at the ATM. In the case of the funds-transfer, security and correctness are the major considerations - within reason, we don’t really care how long it takes to go through the system as long as the correct amount of money is transferred between the correct accounts.

We can cater for both of these issues in the framework by introducing priority queuing and various levels of security for transactions, but this increases the complexity of the framework. Maybe we should leave such considerations to the instantiation of the framework Maybe we should have two or more separate frameworks for different types of transactions These are difficult decisions to take but can make the difference between having a good framework (or frameworks) and a bad one.

Both the perspectives of a framework as an abstraction and as an implementation of an architecture consider the externally observable properties of the framework. The perspective of a framework as a system in its own right considers the internal properties. This is an important view to take because it can make the difference between a framework that struggles under the weight of the abstract relationships and interactions between its classes and one that executes cleanly and efficiently. However, it must be constrained by the other two views - whatever the final design for the framework is, it must capture the key abstractions of the domain and it must structure them in such a way as to realise the non-functional requirements.

Framework documentation and its impact on delivery

How we document a framework is crucial to how we can use it. A well-designed and documented framework can hide a great deal of the complexity of the relationships and interactions between its abstractions. At its best, a framework presents a well-defined interface to a user which consists of a number of methods to be overridden or defined in an instantiation, along with good descriptions of the sort of application-specific code that belongs to each of these methods. In this situation, the structural aspects of the framework are of secondary consideration to the user - they only need to know which classes to inherit from in order to override the necessary methods. However, such an ideal situation requires that we know what all the potential instantiations of the framework will be. In reality we need to give the user some information about the roles, responsibilities and collaborations of the framework abstractions in order that they can see for themselves how the framework relates the application they want to instantiate from it.

How much of the detail of the framework should we expose? Give the user the full design description of the framework, plus two or three examples of how it can be instantiated, and you risk overwhelming them with detail. Give them only a description of the methods to be overridden and you risk them not being able to see how these methods relate to the application they want to instantiate. Whatever the right balance between hiding and revealing the complexity of the framework is, user documentation needs to be focused on what users are interested in: the function of the framework. Smalltalker’s adopt MVC, not because the Model/View/Controller partitioning of responsibility is good, but because it provides the functionality required for interacting with graphical representations of application elements ... and the Model/View/Controller partitioning is good.

Users of a framework want their instantiated application to do something and so the emphasis on the documentation needs to be on how they can instantiate the framework to do it. This means describing use-cases (how a set of objects collaborate to achieve some externally-observable behaviour) in favour of the behaviour particular to a class or class-cluster.

The emphasis of a user's manual should be very different to that of the maintainer's/developer's documentation. Where as the user focuses on the overall functionality of the framework, and how this can be specialised to the behaviour they require for their instantiation, the developer needs to focus on how classes relate to each other, and how these relationships fit into the wider framework. Hence the collaboration between a number of classes to fulfil some externally observable behaviour becomes a secondary consideration compared to the collaboration of a single class or class cluster with other classes or class clusters because the maintainer/developer is concerned with questions like whether refactoring certain classes in certain ways will break the framework.

Splitting off the concerns of the user and the maintainer/developer is very important in framework documentation – I've seen a number of frameworks where the documentation delivered to the user focuses on the structure of the framework in preference to what it can actually do and these make the user's life very difficult. Similarly, whilst the developer/maintainer can benefit from taking the user's point of view, finding out a class's total responsibilities and collaborations across a number of separate use-case descriptions makes maintenance very hard.

But, even if we get the documentation right, there is still the issue of framework delivery. Most successful 'shrink-wrapped' frameworks have been GUI frameworks such as MacApp and MFC. I think that one of the reasons that these have been successfully delivered without the need for substantial human support is because the abstractions in the framework have a direct and obvious relationship to the concrete refinements of them.

In MacApp [Schmucker86] we have abstractions like TWindow, TApplication and TCommand. We can relate these abstractions to concrete refinements very easily because we can see windows, applications and (the effects of) commands on the screen and if we need a new type of window it is reasonably obvious which abstraction we should be dealing with. With something like the Transactions framework we don't have the same obvious connection between abstract and concrete. A ServerBuffer provides a channel for communication with the server but does it take responsibility for sending and receiving ServerMessages or is it just a passive channel acting a bit like a cache? Only the developer really understands what the ServerBuffer abstraction means and, even with good documentation, it is sometimes very hard to communicate this meaning.

The Transactions framework was very successfully instantiated a number of times. The documentation played an important role in the delivery of the framework to the users but probably more important was the initial placement of part of the development team with the users. The developers provided on-site support for the users and also learned a lot about the usefulness of their framework. As the users became more familiar with the framework, the support role of the developers diminished until they were no longer needed as part of the instantiation team.

Communicating the meaning behind abstractions, where the abstractions are not well known or visible (like the TWindow and TCommand abstractions), is such an intensely human activity that, with the current state of practice in framework documentation and framework support tools, I feel we need a parallel of Coplien’s “Architect also implements” pattern [Coplien95]: ł Developer also instantiates'.

Summary

My position on framework development and delivery can be summarised with a few statements of belief:

- In order to develop a good framework we need to consider it as an abstraction of a domain of systems, rather than just of some example systems.
- A framework implements an architecture and the non-functional characteristics of the framework will be inherited by the systems instantiated from it. A good framework needs to consider the non-functional requirements of the systems in its domain. This may constrain or alter the domain.
- These two views of a framework constrain its development as a system in it own right. However, a good framework needs to be developed as a system in its own right to avoid over- or under-abstraction, performance and size problems.
- Documentation for frameworks should focus on function for users and structure for maintainers/developers.
- Delivery of a framework is, with the current state of practice, an essentially human activity. Good documentation is required in delivery but we should also consider Developer also instantiates.
-
- The issues IŽm particular keen to consider in the workshop are:
-
-
- The use of explicit abstract models as a focal point for the abstraction and architecture perspectives. How can such models be used (or, hoe are they being used)? How do models inform, and how are they informed by, framework implementation? What are the important activities in the development of abstract models?
-
- The techniques and activities for documentation. What constitutes a good document? What constitutes a bad one? How much do these documents impact the delivery of frameworks?
- Human and technological interaction in framework delivery. Do we really need developers to work with instantiation teams? If not, what are the tools and techniques to promote the delivery of 'shrink-wrapped' frameworks? If so, how does this culture affect the organisation and how can we usefully feedback user experiences into the development process?

Paul Dyson, Department of Electronic Systems, University of Essex, Colchester, CO4 3SQ, England.

Email: pdyson@essex.ac.uk, Tel: +44.1206.790234, Fax: +44.1206.872900. Email: pdyson@essex.ac.uk Colchester CO4 3SQ
Web: http://vasawww.essex.ac.uk/~pdyson/ England

Bibliography

[Arango&Prieto-Diaz91] Guillermo Arango and Ruben Prieto-Diaz, Domain Analysis Concepts and Research Directions, in Domain Analysis and Software Systems Modelling (R. Prieto-Diaz and G. Arango eds.), IEEE Computer Society Press, 1991.

[Coplien95] James O. Coplien, A Development Process Generative Pattern Language, in Pattern Languages of Program Design, Coplien and Schmidt (eds.), Wiley and Son, 1995.

[Dyson&Anderson97] Paul Dyson and Bruce Anderson, Architecture and Reuse, Proceedings OOPŽ 97, SIGS Publications, 1997.

[Gamma+95] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Object-Oriented Software, Addison-Wesley, 1995.

[Hayes-Roth+95] Barbara Hayes-Roth, Karl Pfleger, Philippe Lalanda, Philippe Morignot and Marko Balabanovic, A Domain-Specific Software Architecture for Adaptive Intelligent Systems, IEEE Transactions on Software Engineering, IEEE, April 1995.

[Johnson92] Ralph E. Johnson, Documenting Frameworks using Patterns, Proceedings OOPSLA '92, ACM Press, 1992.

[Krasner&Pope88] Glenn E. Krasner and Stephen T. Pope, A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80, Journal of Object-Oriented Programming, SIGS Publishing, August/September 1988.

[Schmucker86] Kurt J. Schmucker, MacApp: An Application Framework, BYTE, McGraw-Hill, August 1986.

[Shaw95] Mary Shaw, Comparing Architectural Design Styles, IEEE Software, IEEE, November 1995.

[Wirfs-Brock90] Allen Wirfs-Brock, Panel Report - Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks, Addendum to Proceedings OOPSLA '90, ACM Press, 1990.

Experiences with Design and Construction of Object Oriented Frameworks

Peter Long,
Software Architect, Rolfe & Nolan Systems Ltd.

Objectives.

The objective of this position paper is to outline briefly some of the experiences that I have gained over the last couple of years whilst working with development projects that have involved the design and construction of OO frameworks. By being invited to the workshop I would like to have the opportunity to share and expand upon these experiences. In addition I am keen to broaden these experiences by hearing the approaches and techniques adopted by other workshop delegates. Projects.

The following experiences are drawn from being closely involved in two large development projects in the business environment, Anugraha and Lighthouse. These are summarised as follows. Project Anugraha involved developing a Smalltalk based client/server application for the purposes of automating the production of corporate tax returns for Ernst & Young (UK). Amongst the many objectives that the application had to fulfil, it had to allow the easy incorporation of regular changes to accommodate amendments to the corporate tax legislation in UK. This was achieved through the development of a number of frameworks and toolkits. These frameworks were used internally within the project and where designed with the objective of requiring minimal effort to develop new business function. The frameworks were used internally amongst a team of eight developers.

The Lighthouse application, developed by Rolfe & Nolan, is a high performance treasury and capital markets enterprise-wide trading and risk management solution. An aspect of this system is a set of toolkits and frameworks which clients can use to customise the system to better fit their business requirements. These frameworks have also been used to develop a core set of business function which is delivered with the application. Lighthouse is a large C++ application incorporating approximately 2,500 classes. Experiences related to topics of interest.

The following sub-section structures some of my experiences during these projects under the headings outlined in the OOPSLA call for participation programme. It also highlights those areas in which I would like to increase my knowledge.

1) The relationship between frameworks and design patterns. Many of the frameworks that we have built incorporate a number of the patterns in Design Patterns, including Observer, Factory, Singleton, Proxy, Command, Flyweight and Template. Some are encapsulated within the framework and not exposed to framework clients being used as part of good design practice. Others are explicit in the framework design and help not only in providing a solution, but also as a communication aid to users of the framework. If clients are familiar with patterns it can enhance the speed with which they are able to learn the framework. For example some of the Lighthouse frameworks use Factories so that clients can extend the function provided by the application.

2) The effects of centralised vs. decentralised framework development and adoption. Unfortunately I do not have much first hand experience in this area. However, as the frameworks in Lighthouse are being adopted by an increasing number of clients it would be beneficial to be appraised of potential issues. It is envisaged that managing the requirements for framework enhancements could prove hard when attempting to balance conflicts in requirements from different clients.
3) Optimal framework development team structure and characteristics. Basically this can be summed up as use your strongest and most experienced developers. Ideally the framework developer should have experience in developing more than one software system in the target domain for the framework. This is to ensure that the framework has the appropriate architecture and has been designed so that it has the sufficient variability and flexibility. In addition, the framework developer must be competent in being able to design and build robust and reliable software. This is important as clients of the framework must have confidence that they are building upon a solid base. The frameworks that we have developed have usually been accomplished by a single developer with the support of a software architect. Where experience of the target domain for the framework may be weak these roles has been supplemented by the use of an OO analyst who will abstract a model defining the target systems that may be built using the framework.
4) Communicating the framework and convincing prospective customers to use it. When communicating the framework we typically provide presentations and support documentation. The presentations are used to provide an overview of the framework which describes the problems that it may solve and at a high-level the design of how it works. It will also walkthrough a solution using the framework. The support documentation is in the form of a framework programmers guide or how to guide. These typically provide, an overview, a concrete example of use, a checklist of tasks and detailed class documentation describing the public interface of the framework. These includes the use of an OO notation, such as OMT, and code fragments. Note that these documents are quite different in form to the design documentation used during framework construction. Convincing prospective customers has involved the use of technical marketing presentations that extol the productivity and reliability virtues of exploiting the framework and production of white papers. Typically the productivity aspects are justified by explaining the tip of the iceberg principal, where the framework is considered to be a like an iceberg. 90% of the detail is submerged below the water line, whilst the 10% above represents the publicly exposed interface upon which clients can build function. Clients are gaining a lot of productivity by being able to leverage the implementation details contained in the framework. These ratios vary from framework to framework.
5) Dealing with feedback between framework developers and framework users. Typically we have found that it takes three development iterations to get a framework correct. This means that the abstractions and structure are stable and are unlikely to change substantially as the framework evolves. These iterations are necessary so that the framework is practical to use. This can only be assessed by using the framework in anger for example in a production environment. In Anugraha, we had a computation engine framework that was core to the application. During our first iteration we produced an extremely flexible solution however the abstractions represented within the framework were too fine grained. This meant that productivity was very low as it took a long time to deliver end-user business function. During the second iteration we effectively built a framework on top of the framework which captured more accurately the business abstractions that we had to support. The third iteration refined

this framework structure further taking on board feedback from users of the framework. For instance providing extra helper functions and additional hooks to customise features.

6) Distribution of framework services   I am keen to hear the experiences of other delegates upon this topic.
7) Any other issues of concern to framework development. In addition to the above topics I would like to discuss the following; Experiences of evolving frameworks. In particularly migrating framework users from one version of the framework to the next. What constraints and issues are caused. Brief Biography After graduating from Kent University with a Computer Science degree I have been working for nearly ten years in the business and financial environment developing software solutions. Until recently I was the software architect for a mission critical OO application for Ernst & Young (UK). I have subsequently joined the software house Rolfe & Nolan Systems Ltd as a software architect and am closely involved in the development of their Lighthouse application.

A Framework Approach

Lee Krause

1 Introduction

Our company - Software Productivity Solutions, Inc. - has been involved in researching and developing domain-oriented frameworks for the last two years. As part of this effort, I have developed a Domain-Oriented Software Architecture Engineering Environment (DOSAEE) to serve as the foundation of our framework approach.

The DOSAEE environment provides a domain framework that supports the development of systems based on an architectural view of the systems. The DOSAEE framework supports the development of systems from requirements definition through system implementation. Our research has been sponsored by NASA Goddard and US Army CECOM through the SBIR program. This research has involved investigating the relationship between frameworks and patterns, and more specifically, how patterns can be used in the development of robust frameworks for a particular domain. The DOSAEE program is transitioning our laboratory research to industry with the actual application of frameworks to three domains.

The first domain is a NASA program that is developing a standard way to manage telemetry devices. The second framework involves the re- engineering of a commercial modeling program. In the third program, we're creating a framework to assist the Army in developing communication planning tools.

We will begin each of the applications within the next six months, and use a domain-oriented software architecture engineering framework as its foundation. We believe that the research we've conducted in the area of domain-based frameworks, as well as the use of patterns in the development of frameworks, will greatly benefit the workshop. Since we will begin applying our R&D results to the first framework in August, the ability to understand areas where other have been successful is also of great interest to us. I have an extensive background in software development for large programs, and our approach has focused on solving real-world problems that fit well with the theme of the workshop.

1.2 Contents of this Paper

The remainder of this paper describes our research effort into Domain-Oriented Software Architecture Engineering Frameworks, programs to which the framework will be applied, a brief write-up of the programs, and research I have conducted for the last 15 years.

*Section 2.0 provides a detailed description of the domain architecture framework. In this section, we highlight the steps - and the rationale for each step - required to build a domain architecture framework. Our approach is based on an analysis of current research, our expertise in the area of domain-specific modeling and architectures, and our real-world experience in building large, complex, military software systems. SPS has identified a domain-directed approach to building models, architectures, and implementations of complex software systems.

*Section 3.0 provides a detailed description of one of the domain architecture frameworks under development. In this section, we highlight the development of the JD-MAT framework being developed for the US Army CECOM under a phase II SBIR. This section also lists the other frameworks SPS is developing using the domain-oriented approach.

*Section 4.0 provides an overview of the research and projects that I have been involved with for the last 15 years. Since most of my experience has been in the development of large programs that span the entire software lifecycle, I have gained tremendous insight into the approach needed to develop our framework to meet real-world needs.

2.0 Domain-Oriented Software Architecture Engineering Framework

Recent reuse initiatives, by DARPA for example, have focused on both domain-specific and architectural approaches. Over the past several years, with funding from multiple sources, SPS has been developing a domain-oriented approach to understanding, representing, and analyzing software architectures.

Software architectures are recognized as an important aspect of a software system's design and documentation in that they provide: a high-level design description of the overall system. A framework for reuse of large-scale components. An automatable representation that is suitable for various forms of analysis.

A domain-oriented approach is being followed because it:

- Will integrate with domain modeling efforts that are precursors to system development activities.
- Acknowledges the similarities in construction of applications within a given domain (and the differences in construction of applications across different domains) resulting in an improved opportunity for reuse over other reuse approaches.
- Leverages the expertise that developers accumulate when working on a product line of systems that fall within a given domain.

Figure 2-1. Domain-Oriented Software Architecture Engineering Framework Figure 2-1 depicts the Domain-Oriented Software Architecture Engineering Framework. Our framework is based on three separate but highly related engineering processes: Domain Analysis & Modeling, Architecture Development, and System Implementation. Each of these engineering processes is composed of one or more activities. The Domain Analysis & Modeling process defines an organization's overall understanding of a problem domain area. Domain Analysis identifies the information, functions, relations, constraints, etc., that are relevant to the selected domain.

By creating an object-oriented representation of the domain, Domain Modeling serves as a focal point for agreement among various participants, stimulates further analysis, documents the domain, and is an input to the Architecture Development process.

Architecture Development is aimed at creating architectures that guide current and future system development efforts. Architectures define the overall structure and organization of a software system. Two distinct kinds of architectures are created. The first is a domain architecture that represents a template for building multiple, similar systems within the given domain. The second is a system-specific architecture that is to be used as the initial design of a system to be implemented. This second system-specific architecture is a specialization of the template architecture, capturing additional functions and constraints for the actual system to be built. Development of either a domain or system-specific architecture identifies architectural abstractions in the form of:

- Architectural styles and patterns used to define organizational structures within the architecture.
- Architectural components identifying the functions that are to be provided within the system.
- Architectural constraints which place limitations on allowed component interactions or define non-functional properties.

Our current representation of architectural constraints is in the form of protocols that define the detailed interactions between architectural components. System-specific architectures are used as inputs to the System Implementation process, with the intent of building a working software system within the given domain. The System Implementation process emphasizes reuse. Reuse is supported directly for components that are small enough, common enough, or built within the boundaries of the current architecture, to fit directly into the system being constructed. Reuse through adaptation of COTS and GOTS software is also supported.

Our adaptation-based reuse acknowledges that many existing components will have been built prior to this system, as well as prior to the existence of the domain and specialized architectures from which this system is being built. These components may not fit directly into the current system but can often be adapted to allow their reuse. We believe that working from an architectural framework greatly enhances the opportunities for reuse. As multiple applications are built from a given domain architecture, more direct reuse will occur (as opposed to adapted reuse).

The benefits to organizations building product lines or delivering customer-customized systems that have a common foundation will be significant cost and time savings in development, enhanced quality, and easier maintenance and enhancement of the delivered systems. The following sub-sections provide an overview of the processes and activities in the framework. Note that the activities within the three engineering processes are not applied sequentially because the engineering processes themselves are performed with a high degree of overlap and iteration.

2.1 Domain Analysis and Modeling

This function allows us - with the help of domain experts - obtain knowledge about the domain. Domain experts are individuals considered to have a thorough understanding of the domain including the lexicon and ontology. Experts may include the users and builders of any legacy systems that may exist. Through a knowledge acquisition process, the domain experts describe the domain, often by focusing on functional requirements, to the knowledge engineer. The engineer begins to capture the information, probing for further detail or clarification, in a series of representations ranging from textual descriptions to box-and-line diagrams. Several views of the domain are created during domain analysis.

These different representations capture specific aspects of the problem under study. Taken together, these representations form the domain model. Any model is an abstraction of the real world. During the development of a domain model, the engineers will decide that some information about the real world will not be represented in the model. The choice of what information to include and what to abstract away is based on the intended purpose for the model.

The domain model presents an unambiguous representation of the domain and is used to understand the domain. It is also used to communicate this understanding to non-domain experts who will develop an architecture from which automated systems relevant to the domain can be built. Because the domain model is developed independent of any design or implementation concerns or constraints, it represents the problem domain and not the solution domain. The resulting domain model is input to the Architecture Development process.

2.2 Architecture Development

The philosophy of the Architecture Development process is to support engineers working at a variety of abstraction levels in defining software architectures. The actual construction of an architecture may be driven from either a top-down or bottom-up approach. The result is an architecture composed of components, abiding by constraints and conforming to accepted design patterns and architectural styles. The specific activities of this process allow engineers to define the styles, patterns, components, and constraints, and use this information to formulate and represent architectures.

2.2.1 Identify Architectural Components

Architectural components are the building blocks of architectures, patterns, and styles. Different abstraction levels apply to components, depending on whether they are part of an architecture, a pattern, or a style. In an architecture, components identify actual processing capabilities to be provided by the system. In a pattern, components are "non-functional," meaning their purpose is not important to understanding the pattern or to describing the pattern design. Components in patterns may represent fine- or course-grained functions that are grouped as part of the description of the pattern. At the style level, components typically represent large subsystems or are entire systems themselves interacting according to the specified style. In this activity, architectural components are identified and then placed into a library so that architects can combine them into architectures, patterns, and styles. SoftArch supports the Identify Architectural Components activity at an architectural level.

2.2.2 Develop Architectural Constraints

Architectural constraints represent rules that define the ways components can be combined in architectures, patterns, and styles. Our current direction is to use the interactions that exist between components as the basis of our architectural constraints. At all levels of abstraction within our architectural framework - whether it is style, pattern, or architecture - components interact in a specific way. These interactions form the basis of the constraints governing the way components are grouped into these styles, patterns, and components. In this activity, architectural constraints are identified. The constraints are provided in a library so that architects can use them in their groupings of components which are combined to form architectures, patterns, and styles. SoftArch supports the Develop Architectural Constraints activity at an architectural level.

2.2.3 Identify Architectural Styles & Design Patterns

Architectural styles are high-level descriptions of the way systems are put together. Styles such as pipe-and-filter, client-server, or layered are commonly used to describe the gross organization of a system. This level of description is useful for understanding how the subsystems of a given system communicate with each other, but tells little about the underlying implementation or design of the individual subsystems. Design patterns are recognizable configurations of software components that define the way those components interact to achieve some functionality. Common examples of patterns are model-view-controller, observer, and iterator.

Patterns represent the first layer of true design information, in that software developers can use this information as a guideline for building a system or understanding how a system was constructed. The objective of the Identify Architectural Styles & Design Patterns activity is to build a library of architectural styles and design patterns that are known to be useful within the given domain. This library will contain abstract descriptions of the styles and patterns that can be used to

- Drive the organization of domain and system-specific architectures.
- Analyze an existing architectural representation for conformance to known styles and patterns.

Styles and patterns are represented as constrained groupings of components. The constraints are defined protocols, and the components are at an appropriate abstraction level for the style or pattern being described. While the actual set of patterns and styles found in this library may be fairly generic, including those commonly identified in research systems and guides for practicing designers, this library contains information regarding the application of these patterns and styles to the particular domain.

In addition, it is expected that unique styles and patterns, identified by analyzing existing systems within the selected domain, will be captured in this library. Relations between architectures, patterns, and styles exist. An architecture, or some subset of it, is an implementation of a particular pattern. Furthermore, architectures follow given styles. We track these relationships so that specific uses of patterns and styles can be identified and so that the architecture can be viewed at a variety of abstraction levels.

2.2.4 Develop Domain Architecture

The major objective of our Domain-Oriented Software Architecture Engineering Framework is to produce architectural representations. In this activity the focus is on building an architecture that is suited for driving the implementation of a family of related systems. Subsequent activities will specialize this template architecture into one suited for building a specific system. A domain architecture is defined by bringing together components, abiding by the stated constraints, with the result being a representation of the system to be built.

Our architecture is represented - in the same fashion as styles and patterns - as groupings of architectural components that abide by the stated architectural constraints. At this architectural level, the components and constraints are at a low level of abstraction, describing functionality and interactions that are to be provided in a resulting system. Subparts of the architecture can be mapped to design patterns and styles found in the architectural library. This mapping is used to validate the architecture's structure as well as to provide abstract representations of the architecture for review, discussion, and documentation purposes

2.2.5 Specialize Domain Architecture

A domain architecture describes the high-level design of a family of systems. It is also the starting point for developing a specific system. While a specific system will embody the requirements and constraints represented by the domain architecture, it expands upon these by adding new requirements and constraints that will be included in the actual system to be built. The evolution of a domain architecture into a specific system architecture is a process of specialization and refinement. Specialization and refinement of the domain architecture uses the same techniques defined in the previous discussion of the Architecture Development process. Additional architectural components, architectural constraints, design patterns, and architectural styles may be defined. Using these as well as previously defined architectural entities, the domain architecture extends to embody additional capabilities to be provided in the specific system. (Capabilities found in the domain architecture but not needed in the specific system may also be removed as needed.)

2.3 System Implementation

The System Implementation process focuses on building an end system. Our research concentrates on maximizing reuse of legacy, COTS, and GOTS components during this effort. The following sub-sections provide an overview of the activities in the System Implementation process.

2.3.1 Generate Design Specifications

The first step in any implementation activity is to produce detailed design specifications from which the system will be built. From an architectural perspective, this requires refining the detailed architecture representations and transforming them to a design representation suitable for the development team's need. This activity is outside the scope of our domain- oriented architectural research program.

2.3.2 Insert Reusable Components

Direct reuse, meaning the insertion of existing software components into a system's implementation without modification, is supported in two ways by our approach. First, the domain and system-specific architectures may have been built with explicit knowledge of existing components that apply within the domain. As a result, these components can be directly reused in the system being developed. Second, as systems are implemented from the domain architecture, any manually developed components are ideally suited for use in future systems development. An organization building a product line of systems can - over a period of time - develop a large library of functional components that directly fit into the architecture.

2.3.3 Integrate Adapted Components

A second level of reuse attempts to use legacy, GOTS, or COTS software components that were built prior to the domain and system-specific architectures and that do not fit within the architectural constraints of those architectures. These components might still be reused through the development of adaptors, which make these components work within the architecture of the system under development. Often, the investment to develop a new capability would be far greater than the investment needed to adapt the component to the existing architecture. In these cases, a reuse approach based on component and protocol adaptation can be followed. SPS has an on-going NASA-funded research effort focused on developing techniques that support the identification of these opportunities for reuse and that support automated adaptation of the components.

2.3.4 Implement Remaining System

Segments of the application that are not implemented through the reuse of existing components are programmed using traditional implementation techniques. Through a successful application of our domain-oriented architecture approach, the amount of manually programmed code should decrease significantly with each new application built for a given domain. In addition to the expected cost and time savings, this step will yield benefits in the areas of quality, maintenance, and system evolution.

2.4 Methodology To Support The Development Of Domain Architectures

Figure 2-2 Development of Domain Architecture

Our company researched the use of domain architectures to solve the JTF network management's goal of building plug-and-play network management systems using GOTS and COTS components. This effort revealed the need to define a development methodology for domain architectures. Our initial methodology extends existing software development methodologies to support the idea of components and patterns that define known practices of combining components to achieve a goal. Figure 2-2 defines this methodology by identifying the views that need to be supported in the development of a domain architecture. The methodology defined in Figure 2-2 is a refinement of the engineering process. It details the specific views that need to be developed to create an architecture and develop a product line foundation.

The creation of the architecture is defined by generating a first cut view of how the major subsystems interact in the system. The Architectural View provides a standard set of symbols to convey an abstract view of the system. The Architectural View will continue to evolve as system design takes place. Its design will comprise known architectural patterns that have been documented and proven to be successful in previous systems.

A description of the commercial products that make up the software layers is provided in the Implementation Technology View. This view defines any requirements for the infrastructure of the system being developed to include specific COTS/GOTS products. For military product lines, the DII COE technical reference model will play a major role in developing this view.

The Physical View takes into account the physical constraints on the system related to the allocation of processes to processors and their effect on the system. This view contains information associated with bandwidth requirements and the event load that each of the platforms will need to support. If the system is going to be deployed on multiple platforms, multiple physical views will exist, with each view placing additional constraints on the architecture.

The Architectural Principle &, Rules provide a description of the "must have" constraints on the system. Because the Architectural Principles are treated as the constitution of the product line, they provide high-level guidance for product line development. Architectural Principles should apply to all products within the domain, and, like our Constitution, should change very little over the life of the product line. Rules, on the other hand, convey more detailed constraints that can be placed on one or more products that exist in the product line. An example of a principle might be "the product line shall be based on COTS/GOTS products to the maximum extent possible."

Once all architectural constraints are defined, system development can move into the design phase. The design phase consists of moving from the information in the domain model to the system model being developed. The Design View is generated from the continued decomposition of sub-system and objects of the domain model to the point where architectural and design patterns can be identified. The architectural constraints defined in the architectural stage guide the decomposition.

The Component Implementation View is the transition from conceptual design to implementation. This stage translates the abstract representation of components and patterns to the physical existence of patterns consisting of domain components or of components mapped to COTS/GOTS products used in the system development.

The Thread View provides a validation of the system by building threads through the components to ensure the system meets the users' needs. This view allows the architecture to be validated while taking into account non-functional properties that affect the system development.

3.0 JD-MAT framework that will be developed starting in mid-December 1997

The overriding objective of our effort was to develop a methodology and framework that allows the creation of a family of network management products to support the JTF mission. The ability to build the network management platforms through the use of existing assets and the ability to migrate to new technologies are important aspects of our approach. SPS has conducted significant research in the development of domain architectures which were leveraged in the creation of the JD-MAT framework. Figure 3-1 presents the concepts supported by JD-MAT.

Figure 3-1. JD-MAT framework Specifically, JD-MAT provides the following capabilities.

A domain modeling tool specifically tailored to the information and visual modeling needs of network communications planning and management. This tool will support the definition of service-independent domain models and service-specific specializations of these models. The domain models produced by this tool will describe the entities and functions found in the communications planning and management domain. This model will ensure that the domain requirements are being addressed and that all overlap among the services is explicitly documented.

A domain architecture tool to rigorously define and capture the high level system architecture that all systems built from the JCPMS product line must follow. This domain architecture will capture the high level system building blocks from which a product line instance can be composed. Specializations of the base architecture will be supported, allowing specific system architectures to be defined. By rigorous definition we mean an architectural definition that allows tailored views of the architecture to be defined (e.g., data model, interface model, etc.) and allows tools to perform analysis of the architectural model (e.g., consistency, completeness, non-functional properties).

3.2 Other programs being developed at SPS using the domain-oriented software architecture engineering framework approach

The following frameworks will be developed starting August 1, 1997: … Commercial modeling tool for fire simulation will be re-engineered starting August 1, 1997 using our SAFE system architecture. … Gateway telemetry framework for NASA this program is being used as the pilot program for the develop of the DOSAEE framework. 4.0

Biography LEE S. KRAUSE SENIOR SYSTEM ENGINEER Summary of Technical Expertise Research Areas: Development of component-based software architectures frameworks, Software Architectures, Joint Network Management * Real-time Software Design, Coding, Testing and System Integration *   Software: C, C++, Java, Fortran, Ada, Pascal *   Software Tools: Open Interface Elements, Xrunner, NetMetrixs, Cadre, Rational Rose SUMMARY OF PROFESSIONAL EXPERIENCE April 1994 to Present: Software Productivity Solutions, Inc. Mr. Krause's most recent assignment is the Principal Investigator on the JCPMS Domain Modeling and Architectural Toolset (JD-MAT) Phase I SBIR. JD-MAT is researching ways to model the JCPMS domain to allow a "plug-and-play" domain architecture to be developed. This technology will significantly increase the number of GOTS and COTS used in the JCPMS product line.   Mr. Krause also is technical lead and program manager on the Domain Oriented Software Analysis and Engineering Environment (DOSAEE) Phase II SBIR program, and uses the InSight technology to develop the model and create the actual DOSAEE tool based on the model. Mr. Krause's primary task was to develop a software architecture paradigm that could be applied to real programs and provide significant reuse of existing components through the use of adaptor technology. Mr. Krause has also played a significant role in the design, development, and integration of InSight's user interface. His primary task involved developing the tools that define, display , and manipulate the symbolic representation of InSight models. Dec 93 to April 94: Harris Corporation, RSA Program Mr. Krause worked as an engineering consultant, evaluating software test tools and integrating test tools to be incorporated into the test suite. Mr. Krause integrated COTS products into the engineering model for handling capture/playback and operator history file capabilities. Aug 86 to March 93: Grumman Melbourne Systems Mr. Krause worked as an engineering consultant for the Joint-STARS Program. He was a major contributor to the distributed architecture design. This included database management, transaction management, and distributed processing used on J-STARS. He designed, developed, and integrated the Local Data Manager (LDM) process. The J-STARS system was developed as a multi-processing distributed system. The LDM process resided on the remote nodes to process distributed data for tabular and graphic displays. Mr. Krause designed the Area Service Processing (ASP) process, which provides the radar management operator with information about the operation and control of the radar system. He presented the preliminary and detailed design of ASP at Government PDR and CDR. Mr. Krause performed critical thread timing analysis on system threads within the Operation and Control subsystem to verify that system timelines were being met. Mr. Krause developed initial B5 specification for the Radar Operation and Management function. This entailed the grouping of A-Spec requirements into the B5 function. The development of algorithms and a requirements traceability matrix were part of this activity. EDUCATION B.S. Computer Engineering, Rochester Institute of Technology, 1984 A.A. Science, Rockland Community College, June, 1981

Static and Dynammic Assertion Mechanisms for Communicating Framework Concepts

Boris Bokowski & Lutz Kirchner

Abstract

According to our experience, typical problems in framework development are caused by misunderstandings between users and developers regarding key concepts and mechanisms of the framework. We identify three classes of such problems, which we call the interaction, execution phases and object composition problem classes. For each of the problem classes, we propose lightweight formal description techniques similar to assertions for communicating the underlying concepts and mechanisms and for enabling automatic checks whether the framework is used properly.

1. Introduction

It is widely accepted that developing good application frameworks is hard. Most of the problems in framework development have their roots in the difficulty of understanding and communicating the core concepts and mechanisms of a framework. For three key activities in the development process of a successful framework, overcoming this difficulty is particularly important.

First, it is necessary to communicate the framework concepts between the framework developers. It is desirable that experiences of many developers are leveraged, in order to ensure that the framework abstractions work for all intended uses of the framework. Thus, larger development teams should result in more comprehensive and more general frameworks. It can be observed, however, that current successful frameworks have often been engineered by very small teams, and that it is hard to avoid misunderstandings even in small teams. If there would be a way to describe the concepts and mechanisms of a framework more succinctly, especially for the tedious process of testing and refining a framework, larger development teams could be set up.

Second, in order to explain to a framework's user how to use a framework correctly, again the core concepts of the framework need to be communicated. Because the usability of an application framework depends on the understandability of the solutions and concepts it provides, it is of utmost importance to have ways to describe those solutions and concepts in a way that is easy to understand. For example, if just one user-defined component is not implemented according to those concepts, it might affect the function of the whole framework.

Third, it is important to have ways to clearly identify mismatches between how the framework is actually used and how the framework was intended to be used. This information is needed for determining whether more development iterations are needed for a framework, and it can be used as the input to the next phase of framework refinement, such that more general solutions can be found and included in the framework.

It has been recognized that for supporting these activities, better description techniques than those of classical object-oriented design and implementation are needed. The success of design patterns shows the need to communicate concepts and mechanisms that cannot be associated with single classes. We claim that it would significantly help in the design and implementation process as well as for the usage of a framework, if lightweight formal description techniques were available to communicate those concepts and mechanisms succinctly, making it possible to employ automatic checks at compile-time or run-time. We still think that design patterns are very important, but in addition to using design patterns, we propose introducing assertion-like constructs that can be integrated into the source code of a framework, enabling tools that can check whether the framework's and the framework user's code is consistent with the concepts and mechanisms described.

These checks may be performed either at compile-time, analogous to a static type checker, or at run-time, analogous to enabling checking of assertions. Hard-to-be-found errors could be avoided and bad designs or workarounds could be identified at an early stage. In the main part of this paper, we identify three classes of problems that, according to our experience, occur when during framework development, key concepts and mechanisms of the framework are not understood by a user or a co- developer of the framework. We call these the interaction, execution phases and object composition problem classes. In addition to an abstract description of each problem class, we describe a concrete example together with problems that may occur, and we give an idea as to how formal description techniques could tackle the problems. The paper is organized as follows.

In section 2, we describe our experiences with framework development, followed by an introduction of the three problem classes. Section 3 concludes and presents topics for discussion in the workshop.

2. Problem Classes

The problems we describe in this section stem from six years of experience in building and using object-oriented frameworks. We were part of the development team of SEPIA [Streitz et al. 92] and DOLPHIN [Streitz et al. 94], systems for cooperative hypertext authoring and cooperative meeting-support, respectively. Not only did we make use of the frameworks of VisualWorks\Smalltalk extensively, but also during the numerous design iterations from the first single- user SEPIA to the fully cooperative DOLPHIN, several own frameworks emerged within the systems. Based on SEPIA and DOLPHIN, we helped designing and implementing COAST [Schuckmann et al. 96], a framework for building syn- chronous groupware applications. We also assisted in the design of PIDGETS++ [Scholz, Bokowski 96], a C++ framework for building non-standard graphical user interfaces.

2.1. Interaction

In frameworks, non-trivial protocols usually govern the interactions between objects. Information about these protocols - if represented at all - is spread over several classes, making it hard to understand and thus to adhere to them.

Consider, for example, the UNIDRAW framework [Vlissides 95] for building domain-specific drawing editors. In this framework, an abstract class Manipulator is provided for flexible interpretation of mouse events, depending, for example, on the currently selected drawing tool. The interface of Manipulator consists of three methods: grasp(), manipulate(), and effect(). Assume that the user presses the mouse button over a UNIDRAW object. For interpreting this mouse event and all following events until the mouse button is released, a new object of type Manipulator is created. The view object that received the mouse press event then calls grasp() of the newly created manipulator, passing the event as a parameter. All following mouse events are passed to the manipulator as parameters of successive calls of manipulate(), leading to incremental graphical updates, until FALSE is returned as result of an invocation of manipulate(). After that, the view object calls effect() once, giving the manipulator the chance to perform any final actions. The intended behavior of the framework depends on the correct ordering of calls to manipulator objects - first, one invocation of grasp(), then multiple invocations of manipulate(), and finally one invocation of effect(). Since manipulators can have children, and user-defined manipulators can be integrated, there are several places in the framework where this ordering must be obeyed.

A violation of this ordering requirement could lead to one or more of the following serious problems:

1. Unexpected behavior of the user-interface.
2. Problems regarding display update may occur.
3. Because necessary initialization or finalization actions are not performed, hard-to-find errors may occur at arbitrary later phases of program execution.

In general, the problem arises when objects in a framework interact closely, and when this interaction follows a certain protocol. To communicate those protocols to framework developers and users, we envision a protocol description language, which specifies the correct order of invocations for groups of objects.

Several description techniques for inter-object protocols have been proposed, the most prominent being CONTRACTS [Holland 92]. Contracts are no type-level construct, however. They are used to generate code that by construction obeys the specified protocols. This approach leads to severe difficulties when objects need to participate in more than one interaction at a time. Thus, we propose that the envisioned description techniques should allow to check at compile-time whether concrete classes implement the protocols correctly. One proposal for such a description technique - called interaction protocols - can be found in [Bokowski 97].

2.2. Execution Phases

Control flow in frameworks is often distributed over a large number of objects of different classes. It is common that this control flow is structured by means of global execution phases. Of all methods belonging to a set of classes, some methods may belong to only one of the execution phases, and therefore should not be called during other execution phases.

An example can be found in PIDGETS++ [Scholz, Bokowski 96], a framework for building non-standard user interfaces. Central to PIDGETS++ is a constraint system that maintains lazy one-way constraints between objects of the abstract class Expression. Expression objects have a current value, which may depend on other expressions' values. An expression's value is cached, and it can be queried using a method val(). In case the cache is not set, the default implementation of val() calls the expression's method calculate(). For a special kind of expressions, objects of class Var, their current value may be set, which causes a method invalidate() on all dependent expressions to be called. Dependency relationships are realized by an instantiation of the Observer pattern. The method invalidate(), which has a default implementation in class Expression, invalidates the expression's cached value and then calls invalidate() on all dependent expressions recursively. The Pidgets++ framework may be extended by writing concrete subclasses of Expression. Typically, such a class implements its own version of calculate(), and, optionally, of invalidate(). The framework requires that in implementations of calculate(), invalidate() may not be called on any expression object, and that in implementations of invalidate(), calculate() may not be called. Note that the requirement should be enforced even if method implementations call other methods, in which case the transitive closure of all methods called from calculate() may not include invalidate() and vice versa.

A violation of this requirement could lead to one or more of the following serious problems:

1. The lazy evaluation property of expressions is no longer ensured.
2. Hidden performance problems.
3. Occasionally, infinite loops may occur.
4. Inconsistent expression values may result.

In general, the problem arises when different phases of execution in a graph of objects should not interfere with each other. To communicate those execution phases to framework developers and users, we propose that methods may be annotated such that conflicting sets of methods may be specified. At compile- time, it could be ensured, that for all methods of one set, their call graphs do not include any call to a method of a conflicting set. Note that unlike in synchronization specifications for concurrent object-oriented languages, conflicting methods in the sense used here need not be methods of the same object.

2.3.Object Compositions

Since frameworks strive for flexibility, flexible ways of composing objects are very important. Allowed compositions, however, often cannot be derived from the types of object references alone. It is very common that additional constraints regarding the object structure have to be fulfilled, which sometimes makes it hard for framework users to set up correct object compositions.

COAST [Schuckmann et al. 95] is a framework for building synchronous document-based groupware applications. In COAST, it is a central concept that document structures referenced by session objects have to be displayable, because the framework will open windows on these document structures for all users which participate in the session. COAST includes abstract classes for document components as well as document composites. The application programmer might want to implement a new subclass of composite, which stores components in a non-standard way. There, he may forget that according to COAST's design, it is the composite's job to set a component's container attribute if necessary. Since components may exist without knowing a container, it is not necessary to set a component's container variable in all cases. However, if a composite gets referenced by a session object, all components of that composite will be displayed automatically by the framework. In that situation, it is required that each component references the correct container, since the components' appearance partly depends on their container. Errors will occur when components of the framework are not composed according to such requirements. In this example, a component will either know no container or a false container. This may lead to runtime errors or hard-to-analyze false display contents.

In general, the problem arises when object structures form general graphs, possibly containing cycles. Note that if object structures are restricted to trees, a static type system together with constructor functions usually already ensures that object compositions are correct, which is not the case for graph-like object structures. To communicate allowed object compositions to framework developers and users, we propose that constraints regarding object compositions may be specified. One possibility to specify those constraints would be class invariants known from EIFFEL [Meyer 88]. However, class invariants usually specify consistent states within one object as opposed to inter-object relationship constraints. A better description technique would be the invariants for CONTRACTS [Holland 92], because they constrain relationships between several objects of different classes. But it is often the case that such invariants should not be enforced at all times, especially when in different execution phases of a framework, different consistencies over object graphs are required. Therefore, we think it would be good to associate such invariants with execution phases.

Conclusions, Topics for Discussion

We proposed lightweight formal description techniques for communicating important concepts and mechanisms of object-oriented frameworks. We argued that being able to perform compile-time and run-time checks is a good motivation for description techniques at the type level, and that several problems that typically occur in object-oriented frameworks can be tackled by such techniques. At the workshop, we would be interested in discussing the following topics: How do our experiences compare to those of other framework developers regarding the problem classes we described? What are other candidate problems in framework development that could be ameliorated by formal description techniques?

About the authors Boris Bokowski is a PhD student at Freie Universität Berlin, Germany, and he is working on concepts for representing object interactions in frameworks. Currently, he is working on a tool that implements compile-time and run-time checks for frameworks written in Java. Lutz Kirchner is one of the designers of the COAST-framework [Schuckmann et al. 96], a framework for building synchronous, document-based groupware applications. He is a research assistant at Fakultät Informatik, Technische Universität Dresden, Germany. 5.

References

[Bokowski 97] B. Bokowski: Interaction Protocols - Typing of Object Interactions in Frameworks. Technical Report B-96-10, Freie Universitaet Berlin, Institut fuer Informatik, 1996.

[Holland 92] I.M. Holland: Specifying Reusable Components Using Contracts, Proceedings of ECOOP'92, Springer 1992

[Lewis 95] T. Lewis (ed.): Object-Oriented Application Frameworks, Manning Publications, 1995

[Meyer 88]B. Meyer: Object-Oriented Software Construction, Prentice- Hall 1988

[Scholz, Bokowski 96]E. Scholz, B. Bokowski: Pidgets++ - A C++ Framework Unifying Postscript Pictures, Graphical User Interface Objects, and Lazy One-Way ConstraintsProceedings TOOLS USA 1996, Prentice-Hall

[Schuckmann et al. 96] C. Schuckmann, L. Kirchner, J. Schuemmer, J.M. Haake: Designing object-oriented synchronous groupware with COAST, Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96), Boston, Massachusetts, 1996

[Streitz et al. 92] N. Streitz, J.M. Haake, J. Hannemann, A. Lemke, W. Schuler, H. Schuett, M. Thuering: SEPIA: A cooperativehypermedia authoring environment. Proceedings of the 4th ACM European Conference on Hypertext (ECHT '92), Milan, Italy, 1992

[Streitz et al. 94]N. Streitz, J. Geissler, J.M. Haake, J. Hol: DOLPHIN: Integrated meeting support across local and remote desktop environments and Liveboards. Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94), Chapel Hill, N.C., USA, 1994

[Vlissides 95]J. Vlissides: Unidraw - A Framework for Building Domain- Specific Graphical Editors, in [Lewis 95]

Patterns for Developing Successful Object-Oriented Frameworks

Joseph W. Yoder

August 27, 1997

1 Overview

The work described here extends last years OOPSLA framework workshop paper [Yoder 1996] describing a "Business Modeling" application framework that was developed and deployed at Caterpillar, Inc. Caterpillar, Inc. joined the National Center for Supercomputing Applications at The University of Illinois as an Industrial Partner in December 1989 to support the educational function of the University and to use the University environment to research new and interesting technologies. During the partnership Caterpillar has initiated various projects, including the evaluation of supercomputers for analysis and the investigation of virtual reality as a design tool.

As described in last year's workshop paper, the Business Modeling project aims to provide managers with a tool for making decisions about such aspects of the business as: financial decision making, market speculation, exchange rates prediction, engineering process modeling, and manufacturing methodologies.

This tool must be flexible, dynamic, and be able to evolve along with business needs. Therefore, it must be constructed in such a way so as to facilitate change. It must also be able to coexist and dynamically cope with a variety of other applications, systems, and services [Foote & Yoder 1996].

One of the problems that we had to deal with was that there were many different business units within Caterpillar in which the system needed to be deployed. Each of these business units had a different business model and different requirements for viewing and searching their data.

The system that was developed and deployed was a Financial Modeling framework [Yoder 1997] that generated reports, answered questions such as why costs were too high, allowed for error corrections, and included a security model. The framework allowed for quickly creating applications that examine financial data stored in a database. The applications that are generated produce profit and loss statements, balance sheets, detailed analysis of departments, sales regions and business lines, with the ability to drill down to individual transactions.

The framework was developed using Smalltalk but it can create a complete system for examining financial data without any Smalltalk programming. This is primarily because the business model is stored in a database. The framework interprets the descriptions of the business model along with GUI descriptions for the reports and dynamically builds an application based upon this information.

The primary principle discussed in this paper is to not program in a general-purpose language, rather program in a higher-level domain-specific language. You can do more with a general-purpose language, but with a higher-level language you can more quickly program, within a limited domain, in fewer lines of code. This work provides a visual-object-oriented language [Burnett, Goldberg, and Lewis 1995] to allow one to program without feeling like they are programming. They are designing the application in terms of domain specifics that they are familiar with.

2 Patterns for Developing Frameworks

The framework development process follows the model presented in the Evolving Frameworks paper [Roberts & Johnson 1997]. We started out with three business units that appeared to be completely different. After working on these for a few months, we saw many similarities. This lead us to develop supporting white-box frameworks for reusing code within and across these applications. It required the developer to write many subclasses for specifying the differences between business units, however there were some key functional areas that could be abstracted out and reused. We only fully implemented one system for Caterpillar.

The user-interface frameworks primarily consisted of summary reports, detailed transaction reports, and graph reports. Some basic components were built with some abstract superclasses. Every time you needed a new report for a business unit, you wrote a new subclass for the GUI that also implemented much of the business logic. These white-box frameworks followed what you might think of business objects as our framework had classes for income, cost, inventories and the like. All of the similarities were abstracted out into a common ReportModel superclass which was a subclass of the ApplicationModel class in VisualWorks.

Early in the development stage, we noticed that the business logic needed a lot of queries to get values from a relational. These queries often had to trigger updates of other queries or values whenever some selection criteria changed. For example, maybe the user was looking at the income for product “X" and wanted to change the view to show the income for product “Y". In order to get all of the views to update correctly, every view and every query that had values coming from the database representing income for product “X" had to be updated to represent income for product “Y". We used Smalltalk's dependency mechanism to handle these updates. Needless to say, the maintenance of these updates became a nightmare. Every time we wanted to change queries or write a query based upon the values of other queries, we had to either change or write the appropriate update method.

These were the early “hot-spots" for us. This lead us to develop a Query- Object class hierarchy that allowed to quickly build queries for getting values from the database and also allowed us to use the ValueModel provided by Smalltalk for automatically updating values. These because components for building the business logic and plugging them into views for being displayed.

Queries could be built that were dependent upon a ValueModel. These QueryObjects were automatically updated whenever the ValueModel that they were dependent upon changed. You could also build QueryObjects based upon other queries that automatically updated themselves whenever any of the queries that they were dependent upon changed. This reduced our maintenance nightmare by a large factor as we no longer had to worry about writing or updating dependency code. [Brant & Yoder 1996] talks about these issues in more detail in their Reports patterns.

We were also able to build a black-box framework for graphs and detailed reports early on as each of them only took a collection of values and displayed a report for them. Thus, we could parameterize these reports to “openOn:" a query or collection of values that then dynamically built what was needed for viewing the data. This was done by reusing what was provided by the VisualWorks graphing and database frameworks.

Most reports still required customization and subclassing. A major improvement happened when we separated the business logic from the application models. We ended up with a logically three-tiered system where the domain objects knew how to get all of their values from the database and the GUIs knew how to display the values and update them. We had two class hierarchies that needed to be maintained, one for the GUIs and one for the domain objects. Whenever a new report needed to be created, the developer would subclass one of the GUI objects and overwrite what was being displayed and also wrote a subclass of a domain object that described the business logic for what was being displayed.

We then noticed that all of the GUIs were very similar and were really just the same view on different data elements. This led us to build a component library for the GUIs which were primarily pluggable objects. This became the beginnings of a black-box framework as the GUIs were built from descriptions that were decoded into how to build the GUI and what domain elements should be plugged into the view.

As we saw more and more similarities, we continued to either extend our components or make new components. We developed more pluggable objects and they became more and more fine-grained. This was an iterative process where we continued to rethink our design and refactor our code.

Ultimately we saw where we could also describe the business logic in a database and build the domain objects on the way. This allowed for the business logic to dynamically change without rewriting code. We ended up with a restrictive domain-specific language. It allowed the developer of the system to program at a high-level within the limits of the domain, thus quickly producing an application for analyzing the finances of a business. The set of frameworks had evolved to be completely black-box. We provide ways to describe the GUIs and the business logic and store the values in a database. The values are read in on demand and through the use of the Interpreter and Builder patterns [Gamma et. al 1995] dynamically build an application as needed.

The next step was to create visual builders for describing the GUIs and the business logic. We took a top-down approach for a proof of concept. This approach took the way you viewed reports from the top-level down to the summary, detailed and graph reports and allowed you to step through and describe the reports. We also allowed for the business logic to be described in this manner as it starts from the high level reports and you let describe the values that get plugged into the GUIs down to the queries that ultimately get mapped into SQL code for getting the values from a relational database.

We ended up with two databases. One database is the database that models the real business data. It has the values of the actual transaction of your business that may include for example costs, income, and so forth. This database can evolve with your business needs which might not only take new types of data, but might also have a new structure defining the database. Even though our framework maps into a relational database, since QueryObjects are a layer above the real business database, our framework can easily be changed to map into any type of database for getting your business values.

The second database which holds the values describing the GUIs and the business logic (the meta-data). The structure of this database should never change unless you change the framework. However the values that are kept in this database does change with your business needs. Whenever a new report or a new way to slice your data is needed, the modeler simply describes the new look of the new report along with the business logic needed for presenting the values in the report. This data is used for dynamically building the view onto your business data.

The surprising part of this was that instead of ending up with normal business objects such as Income and Cost, our framework consisted of Formula, Query, and GUI objects that really just supported our domain-specific language.

Ten years ago, people would have thought that our first version of the system separated the business logic from the GUI. After all, we didn't really write any view classes. We only wrote application models. However, when you have a well-developed GUI framework, the GUI work is how you build application models. Now you want to separate application models from the "real" model. And if you want it to be dynamic, thus adaptable to the changing needs of your business, it is important to incorporate reflective techniques into your framework.

3 Conclusion

This workshop paper has brie y described the evolutionary process that was undertaken during the development of the Financial Modeling Framework. Framework development was an interactive process for us as we first had to develop some working applications in order to see the common abstractions. Early on as we were learning the domain we developed white-box frameworks. We were then able to generalize from the concrete examples. As we iterated through the process, the abstractions revealed ways to generalize into components. These components became more black-box and fine-grained which required much less programming.

Ultimately we ended up with a domain-specific language for building these frameworks without writing any Smalltalk code. This then lead to the development of a visual-language for building your application into some- thing that the domain-experts will be familiar with. The only thing left is to extend the language with debugging tools, profilers, version control, and optimizers.

This work was supported in part by Caterpillar/NCSA at the University of Illinois. The author is grateful for the support provided by the above mentioned entities but in no way holds them responsible for any possible inaccuracies or claims.

4 References

[Goldberg & Robson 1983] Adele Goldberg and David Robson; Smalltalk- 80: The Language and its Implementation, Addison-Wesley, Reading, MA, 1983.

[Brant &Yoder 1996] John Brant and Joseph Yoder; Reports Collected papers from the PLoP '96 and EuroPLoP '96 Conference, Technical Report #wucs-97-07, Dept. of Computer Science, Washington University Department of Computer Science, February 1997, URL: http://www.cs.wustl.edu/~schmidt/PLoP-96/yoder.ps.gz.

[Foote & Yoder 1996] Brian Foote and Joseph Yoder; Architecture, Evolution, and Metamorphosis, Second Conference on Pattern Languages of Pro- grams (PLoP '95) Monticello, Illinois, September 1995 Pattern Languages of Program Design 2 edited by John Vlissides, James O. Coplein, and Norman L. Kerth. Addison-Wesley, 1996.

[Gamma et. al 1995] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[Roberts & Johnson 1997] Don Roberts and Ralph Johnson; Evolving Frame- works: A Pattern-Language for Developing Object-Oriented Frameworks, Third Conference on Pattern Languages of Programs (PLoP '96) Monticello, Illinois, September 1995 Pattern Languages of Program Design 3 edited by John Vlissides, James O. Coplein, and Norman L. Kerth. Addison-Wesley, 1997.

[Yoder 1996] Object-Oriented FrameWorks Tutorial for OOPSLA '96, URL: http://www-cat.ncsa.uiuc.edu/ ~yoder/papers/oopsla96/ooframewks/ooframewks.html.

[Yoder 1997] A FrameWork for Building Financial Models, URL: http://www-cat.ncsa.uiuc.edu/~yoder/financial framework/.

Appendix – Summary of the 1996 OOPSLA Conference

Development of Object-Oriented Frameworks

Organizers

Todd Hansen—Northern Telecom, USA
Craig Hilsenrath —Greenwich Capital Markets, USA
Bill Opdyke—Lucent Technology, USA
Steven Fraser —Northern Telecom, USA
Honna Segel —Northern Telecom, Canada
Erich Gamma— IFA Consulting, Switzerland

Overview

The purpose of this workshop was to give attendees a better understanding of how to avoid pitfalls and develop successful frameworks. We posed three questions:

- How can we achieve the right level of generality in a framework given shifting requirements?
- How can we communicate the framework architecture and achieve usability?
- How can we design effective economic models to clarify what should be framework-developed?

The format of the workshop included an initial group introduction session. The goals of this session were to briefly introduce each participant and discuss their expectations, experiences, views on successful framework characteristics, and areas of interest. Subsequently, participants were divided into discussion groups based on their interests. Each group brainstormed on one of the three topics and presented their findings to the entire audience. The workshop concluded with a group discussion of potential framework topics and a list of possible next steps.

1 Workshop Specific Content

1.1 Expectations

Most participants were expecting to gain insight into framework development issues such as training, usability, and economic justifications. Others wanted to learn from the successes and failures of experienced framework developers. A few just anticipated good soul-refreshing conversation.

1.2 Areas of Interest

Participants expressed personal interest in the following areas:

- refactoring designs
- testing frameworks
- patterns and pattern languages
- effects of centralization and decentralization on framework development and adoption
- how to turn a framework into a stand-alone product, and how to support it
- how to deal with feedback between framework developers and user groups
- relationships between frameworks and design patterns/architectures

- experience with reuse of frameworks
    - adaptability of frameworks
- how to represent a framework
- how to capture the correct interface
- requirements collection

1.3 What Is A Framework?

There was no clear consensus on a single definition for a framework. However, several definitions were suggested:

- an abstraction of, or schematic for, a family of problems
- a group of objects which meet a well-developed set of functional requirements. The framework may incorporate other frameworks and objects
- a set of interacting components designed to facilitate application development for a specific domain
- a reusable design expressed in text, graphics, and code, which captures design decisions
- a collection of classes and their relationships, used to capture the reuse of design and possibly of implementation
    - structure, relationships, and roles for composing objects to implement a software application for some domain

One distinguishing characteristic of frameworks is that they impose a collaborative model to which applications must adapt. This implies that there is an inversion of flow control so that the thread of control now lies within the framework, not the application.
A framework is not an API or a subsystem.

1.4 Characteristics of Successful Frameworks

1.4.1 Organizational

Successful frameworks are cost-effective and strategic. They are usually built by a small number of highly-integrated, experienced OO designers (i.e., fewer than four designers) and delivered quickly (i.e., something usable within one year). Framework introduction is managed by the design team. To minimize excessive code churn, customers don't exploit the framework before it is reasonably mature.

Frameworks are used by choice, not by organizational or corporate mandate. Equal treatment within the organization is accorded application designers *and framework developers, and framework changes are communicated via a controlled feedback mechanism running between application teams and framework designers.*

Framework-savvy organizations recognize the importance of framework development even though this activity is not a direct source of revenue. Strategic frameworks survive cost-cutting measures.

1.4.2 Technical

Successful frameworks address relevant problems and are scoped to a single domain. The framework does not try to do too much. This scoping is based on extensive domain analysis, not preconceived solutions. Good designs are decoupled (e.g., DB and UI) and make modest—not excessive—use of inheritance and aggregation.

Successful frameworks meet customer performance, functional, and productivity requirements. They are not too general, too complex, or too flexible. Rather, they are well-documented, easy to understand, use, and extend. Upon delivery, they are both testable and tested.

Successful frameworks are distinguished by repeated reuse, typically by more than three applications. They are easily adapted to changing requirements. Although a good framework supports application delivery, it is supported and maintained as a separate product.

1.4.3 Examples of Successful Frameworks

- ObjectTime
    - Unidraw/HotDraw
- ET++
- MVC
- MacApp

1.4.4 Examples of Unsuccessful Frameworks

- Cosnos—too little return on investment
- NextStep—business mismatch (end user vs. developer)
- Motif—hard to use and to extend
- CommonPoint—too big an effort

2 General Discussion and Conclusions

2.1 Breakout Group 1: Achieving the Right Level of Generality in a Framework Given Shifting Requirements

This discussion group addressed different development strategies that can be used to minimize requirements churn in frameworks that cater to multiple dependent customers and applications. Four basic approaches were considered: Depth First, Breadth First, Reference Model, and Pearl Dive. The group defined each approach and recorded the potential benefits and drawbacks.

2.1.1 Depth First

Definition

Gather requirements from all customers but select one application around which to develop the initial framework. Add new behavior to the framework by considering each additional application individually.

Benefits

Requirements are scoped primarily around one specific application. A framework can then be produced quickly, allowing customers to provide early feedback to the framework team, and allowing low-level design issues to be identified and resolved early in development.

Drawbacks

Framework designers can expect significant design and code churn as new requirements are added. In many cases, it may be difficult to rework the framework to support additional requirements. Framework robustness may be compromised.

2.1.2 Breadth First

Definition

Gather requirements from all customers and consider all these requirements when developing the framework. Try to generalize the framework enough from the beginning to support all of the applications.

Benefits

All known requirements are considered from the beginning of the development effort so that framework design and code churn are reduced.

Drawbacks

Numerous initial requirements can be overwhelming and may lead to long development delays before anything useful is delivered to the customers. A lack of early understanding of low-level requirements, and generally slower iterations, constitute potential risks to the design.

2.1.3 Reference Model

Definition

Build a generic reference application representative of all customer requirements. This application can be used to train users and to provide a model for adding new behavior to support new applications. This is a specialized version of both the Depth First and Breadth First approaches.

Benefits

The reference model is generally simpler and can be useful in training customers who will use the framework. It can be produced quickly so that customers can verify their requirements early. Customers gain confidence in the framework by working with the reference application, and they can test the feasibility of adding new requirements to the framework.

Drawbacks

It may be difficult to build a reference framework which is representative of all customer requirements. It will also take additional time to develop the reference application, and some of this framework behavior may be non-reusable.

2.1.4 Pearl Dive

Definition

Pearl Diving is a combination of the Depth First and Breadth First approaches. Consider requirements from all customers and stratify them based on common behavior or characteristics. Select one reference application from each group and use these requirements for initial framework development.

Benefits

This approach limits the scope of the requirements while still considering the basic behavior required by all applications. The framework can be produced quickly and customers can provide early feedback to the framework teams. Low-level design issues are identified and resolved early. Pearl Diving can help identify cases in which a need for multiple frameworks is indicated by the resulting stratification.

Drawbacks

Some requirements could be missed if stratification is performed improperly. This approach is beneficial only when the number of framework customers is significant.

2.1.5 Recommendations

The group did not reach a consensus as to the best approach in all cases. However, it seemed clear that a combination of both Breath First and Depth First approaches was desirable. This combination provides two key benefits:

- It reduces framework churn introduced by new requirements.
- It quickly provides framework behavior that customers can use and verify.

2.2 Breakout Group 2: Communicate the Framework Architecture and Achieve Usability

This discussion group focused upon how to communicate the framework intent and achieve usability. Group members identified five phases in framework adoption and usage, and the solutions differed with each phase.

2.2.1 Phase 1: Convincing Potential Users

During this phase, it is important to demonstrate architecture in practice. Overcoming “not invented here” attitudes can be a challenge. The group noted several forms of communication that can help convince potential users:

- examples — Begin with working examples and extend them in real time.
- testimony — This would preferably come from other users of the framework. Present the strengths and weaknesses of the framework.
  User testimony will enhance credibility
  and help avoid over-expectations.
- A test suite constraints document.
- An architectural overview.

2.2.2 Phase 2: Getting Started

Whereas the prior phase focused upon how the framework worked, this phase shifts attention to communicating how people should work with the framework. It is important to teach rituals that can solve simple problems simply.

Several forms of communication are important here:

- examples
- a cookbook of usage patterns, rituals, practices
- tutorial
- consulting
- a concepts manual (to present the "big picture")
- a simple interface design; visualization tools

2.2.3 Phase 3: Development

This phase begins several days, weeks, or months after developers have begun using the framework. This phase seeks to initiate the developers into some of the "mysteries" and techniques for applying and extending the framework to handle more complex applications.

Key forms of communication in this phase are:

- sharing of principles and values, perhaps at a user's conference
- more of when and why to apply the framework
- a reference manual
- consulting (in "short bits")
- a "help system," with a second, more detailed level of information
- CRC documentation

2.2.4 Phase 4: Outer Limits

In this phase, developers attempt to extend the
framework in ways that stretch the limits of the framework's capabilities.

Forms of communication that are important here include:

- intense consulting
- techniques for disciplined development
- a more comprehensive test suite
- a description of framework subtleties (algorithmic, complexity)

2.2.5 Phase 5: Usability

This phase focuses most intensely on the internal structure of the framework (rather than how the framework is used). The usability phase catalogs techniques, not only for solving nearer-term problems, but also for restructuring the framework so that it is easier to evolve and extend.

It is important to document both the incoming and outgoing messages related to a class. Naming techniques (name spaces, metaphors) should be described, as should techniques for isolating parts of the framework that change most frequently. You can then reorganize the framework so that a new user might quickly grasp a naive domain model, then motivate the deltas from the naive model. Apply aggregation where it can clarify the design.

2.2.6 Recommendations

In summary, this group described five phases in framework adoption and usage. Several forms of communication seem to recur among those phases: working examples, consulting/ support, and documentation targeted to various levels of detail. As a user community matures in its understanding and usage of a framework, the communications become more detailed and the topics more advanced. Staff members who are responsible for framework development and adoption should regularly assess which phase their customers are in and carefully focus their communications to meet user needs.

2.3 Breakout Group 3: Design Effective Economic Models to Clarify what Should Be Framework-developed

This discussion group concentrated on the framework lifecycle and the issues associated with each phase of this lifecycle.

2.3.1 Bootstrap

Create from Scratch

This phase applies for the first applications that are developed on a framework. Since the framework has not yet been written, it must in fact be developed simultaneously with these initial applications.

Issues

The initial customers must be friendly and the framework team must be “easygoing and flexible.”

The perceived value of the framework must exceed the perceived cost (adoption or replication). This often requires cooperative, integrated development of the framework and the first several applications.

Generally, the applications chosen need to be delivered as quickly as possible. The “customer” may demand that your framework be early (before it’s done), stable (before it’s been iterated through), and low-risk (before it’s been verified). Should the project fail (and most do under the above circumstances), any supporting framework will take some of the blame—even if it did not cause the failure.

Re-engineer Some Existing Applications

Here, the applications pre-date the framework. Designers must discover the framework “hidden within” the set of applications and then re-engineer the applications around the “extracted” framework.

Issues

Since the applications already exist, time-to-market pressures are reduced. However, the effort to extract the framework can interfere with ongoing feature roll-out.

2.3.2 Re-use/Emergence

Your framework exists in at least one stable version, and a couple of applications are successfully using it. Since the framework is relatively stable, learning the framework makes sense and cannot be considered a waste of time.

Issues

Customers who underestimate development costs will low-ball these costs versus reusing the already-existing framework. (They will tend not to buy the framework.) Customers who understand development costs tend to desire non-programming solutions. (They won’t buy the framework either.) Framework groups need to find organizations
that are between these two stages.

Those customers who choose to use a framework may find themselves in a conflict of interest: use
of the framework brings increased productivity and may result in “downsizing.”

2.3.3 Volume/Cash-cow

New sales produce income with little or no incremental costs. Rogue Wave and IMSL are good examples. Both of them are supported as commercial products.

Issues

Most frameworks never reach the Volume/Cash-cow phase. This may be due to selective sampling (most people never hear about successful, internal-only frameworks) or it may be due to the relative lack of success of internal frameworks.

2.3.4 Standardization Phase

Framework functionality is a commodity. For example, Rouge Wave’s original container class library was “standardized” by the STL.

A framework in the standardization phase can serve as an effective launchpad for new products which can bring in new profits. Developers can quickly build added-value products on top of the framework. For example, Rouge Wave’s network containers are built on their original container classes.

Issues

The framework is essentially frozen. Fixing known bugs may be impossible, so they become “documented known bugs.”

Some frameworks are so close to an application that this stage is never achieved (e.g., OTS - Object Transaction Service from OMG).

3 Next Steps

Workshop participants expressed interest in further exploring the question— what is a framework? This discussion has been taken on-line by the group. Other interesting topics were proposed but not considered owing to time constraints. Specifically,

- framework integration and interoperability
- testing frameworks
- manage teams with frameworks
- documenting via patterns

These are all potential candidates for an OOPSLA 97 framework workshop.

Workshop Attendees

John Artim	artimjo@oocl.com
Kent Beck	70761.1216@CompuServe.com
Terry Cherry	tccherry@nortel.ca
Rich Demers	rademers@msn.com
Steven Fraser	sdfraser@nortel.ca
John Gilbert	gilbert@acm.org
Todd Hansen	thansen@vnet.net
Steve Hayes	shayes@khatovar.com.au
Craig Hilsenrath	hilsenc@gcm.com
Wouter Joosen	wouterjoosen@cs.kuleuven.ac.be
Peter Kriens	peter.kriens@aqute.se
Mark Linton	linton@vitria.com
Jeff Oakes	ustvyzab.ibmmail.com
Bill Opdyke	william.opdyke@bell-labs.com
Arthur Reil	72360.151@compuserve.com

Jon Siegel	siegel@omg.org
Bedir Tekinerdogan	bedir@cs.utwente.nl
Larry Williams	larryw@ObjecTime.com
Joe Yoder	yoder@freedom.ncsa.uiuc.edu