Report: IFIP TC2 Summer School

Gordon's Bay, 15-20 Jan 2006

Andrew Boake, SystemicLogic and Espresso Research Group, University of Pretoria

Introduction

I must tell you about a recent series of lectures that I attended in Gordon's Bay, near Cape Town, in mid-January 2006. Arranged under the auspices of IFIP (International Federation for Information Processing) TC2 (Technical Committee 2: Software Theory and Practice), the event was billed as a "Summer School". This was something that mainly university postgraduate students, but also industry practitioners, in the general area of software engineering, and more specifically software architecture, could attend to brush up on their awareness of important aspects in the field before embarking on a tough year of research / industrial software delivery.

The idea was to get world experts on various relevant topics to bring delegates up to date on the state of the art. In this, in my opinion, it certainly succeeded. Approximately 100 attendees from various universities (mainly South African, but also including a sizable contingent from the rest of Africa), the local software development industry, and a smattering of overseas visitors enjoyed a week of being connected to the mother ship. And if that was not enough, we found ourselves at an idyllic sea-side resort with numerous ice-cream shops, and miles of beach to walk along early in the mornings and after lectures (to ponder the day's information). Among the things that

Professor Judith Bishop is really good at is choosing conference venues (and then of course organizing them with military precision).

In my usual obsessive-compulsive style of writing down every gesture of the speakers, I took copious notes, including ideas that came to me as I listened. There's nothing quite like sitting listening to world experts to get the creative juices flowing. As you probably won't be able to decipher the notes as they stand (those who know me know I can write directly onto microfiche), this is an attempt to paraphrase what I feel were the important ideas expressed by the speakers, and some of the ideas I had (mainly as side bars).

For this report, I have chosen to highlight five of the speakers whose topics I found closest to my areas of interest. That is not to say anything bad about the others – I had to stop somewhere. I hope that this report will serve two purposes: I hope that it gives you some idea of what seems to important in the different fields right now, and I also hope it encourages you to join us in two years' time for the next one. I'll certainly be there if its humanly possible.

Speakers

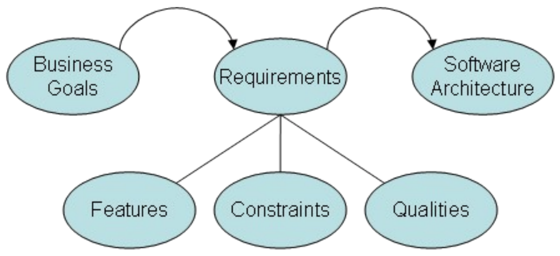
Len Bass (Software Engineering Institute, Pittsburgh, USA)

Design Principles of Software Architecture

Len was one of three guys from the Software Engineering Institute, which is affiliated to the Carnegie-Mellon University in Pittsburgh, USA. With Paul Clements and Rick Kazman, he has written many excellent books on Software Architecture, a couple of which I use extensively in my Honours course in Software Architecture at Tuks.

Len spoke around aspects of design in software architecture, and gave me many things to think about.

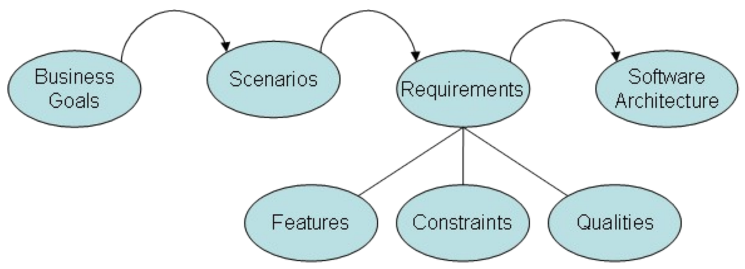
First, he gave us an overview of where architectures come from.



Business goals give rise to requirements, which result in architectures. Behind this is the process of *design*, which you can define as making decisions among alternatives. Features are the major functional requirements. Constraints give you fewer alternatives to choose between, being decisions you don't have to make. Qualities are so important to software architectures that some people say they are the reason for software architecture. For example , a required quality of a system may be that it should be long-lived. This in turn means it should be modifiable, and therefore should be designed, for example, in layers. Another example is, if you want a system to be robust, you should design it with redundancy in mind. What struck me here was the fact that software development in industry places a large emphasis on the required features, and expends very little effort on expressing the required qualities, which often have more impact on the design approach than the features.

It is important to specify qualities precisely: for example, if you want a system to be modifiable, you should specify the types of expected changes. In a similar vein, if you want a system to be secure, you should specify the types of threats. Qualities should be measurable. I also wondered here how exactly one should express these desired qualities, especially for large real-world systems.

How does one get to these requirements? Stories (scenarios) are important. One should walk through things that could happen to the system, and how it would respond. A functional requirement could then be expressed as: a stimulus (specifying its source), the environment in which the system operates (where and when), the response (with *measures*), and artifacts produced. This gives information to the designers of the system. An interesting fact that Len pointed out is that there are always quality requirements for every functional requirement (eg performance, security), and there are always functional requirements that must implement quality requirements. So, one can expand each by considering these scenarios.



If design is the process of making decisions, the decisions you make first should be those that have profound, far-reaching implications. This is his basis of defining where architecture is important. He defines *architectural* to be anything with system-wide impact. He says that design is too hard to do all at once, so one has to decide which requirements would have more impact than others, call these the *architecturally significant* ones, and use these as drivers of the early design decisions (otherwise known as the architecture).

Here, I wondered how this related to the 'simple design' (no big-up-front-design) approach of the agile community, and especially how design evolution would fit into such an approach. What kinds of design decisions need to be made up front (architecture), versus what kinds of design decisions can be grown / emerge / be refactored into a system?

Architectural tactics are approaches to improve qualities: given a desired quality, it seems you should be able to know / look up (?) a particular tactic that explains how to design a system to achieve that quality. For this, you need a model of the system that will allow you to understand it, and analyse it. There was a hint here of the need for an automated design assistant that would help you to find the right tactic to employ.

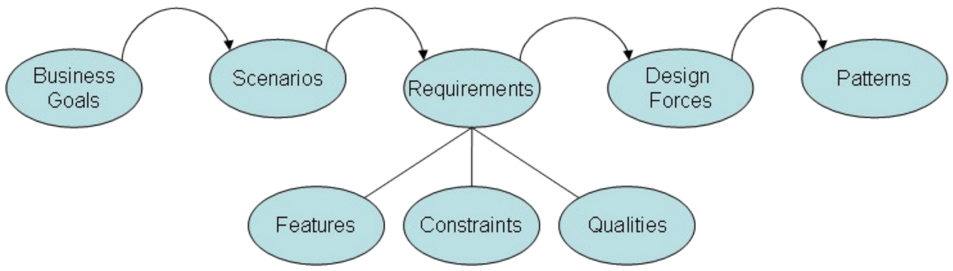
Kinds of architectural design decisions include:
- coordination model and communication mechanisms (eg synch vs asynch),
- data model (abstractions, especially those relevant to inter-element communication),
- allocation of functionality (major categories, modes, divide and assign),
- management of resources (time, process model, resource management, limits),
- binding-time decisions (dependencies, state, variants),
- architectural mapping (different views, elements, dependencies).

Here, I thought that perhaps a really good MSc topic would be to develop a required quality – architectural decision model for J2EE or .Net, a framework that would seek to understand *why* one would use design patterns in a particular way, at a particular time.

Len spoke about the need for *views* of an architecture – looking at a system from different vantage points. These include a static, structural point of view (including looking at things like coupling, cohesion and ability to handle change (cf David Parnas' reason for encapsulation was to hide those things that would most likely change)). Another view is that showing dynamic aspects like concurrency and state. This reminded me of Phillipe Kruchten's 4 + 1 View model of software architecture, where he says very similar things. Yet another view is a mapping view, where you show things like deployment, how to handle mobility, and implications of failure.

We then went on to discuss a particularly interesting typical (usually late) requirement addition, and the far-reaching effect it could have on system design. This was the specification that the system should provide an 'undo' button (which has to reverse changes), or even worse, a 'cancel' button (which must stop any further changes, and undo those already done). If one looks at a 'model-view-controller' pattern (often used in interactive systems), this requirement can impact all three levels. The view should acknowledge the 'cancel' request and show progress status of the 'cancel' operation happening in a separate thread, giving control back to the user. The controller should know that the previous process is stopped and a process undoing all of the completed work is under way. The model should free resources and perhaps roll back to the last transaction boundary. Of course, these considerations would be different in different development environments, for example mainframe vs web.

The really important thing would be to be able to reason about the costs and benefits of fulfilling a requirement, especially one as complex as 'cancel'. This would mean being able to understand the 'design forces' that a requirement would exert on a system, which patterns should be used to resolve these design forces, and the effort and benefit of undertaking this effort.



In closing, Len spoke about the need for guidance to practicing architects in these tasks, so that they should have access to, for example, checklists of things to consider, when undertaking a design task to fulfill a particular requirement. This was important so that they should be able to learn from previous wisdom, and not forget important, perhaps less obvious, aspects. This applied not only to estimation up front, but also in actually implementing the required changes.

I wondered how many practicing software engineers:

- know this detail of the design implications of seemingly simple requirements,
- read, learn from, and apply the patterns literature in their day to day tasks,
- take the effort to understand the complexity and effort required to, and explicitly state the benefits of, implementing stated requirements in their planning before implementation,
- understand and use modeling notations like UML to express, analyse and communicate their designs.

Paul Clements (Software Engineering Institute, Pittsburgh, USA)

View-Oriented Representation of Software Architecture

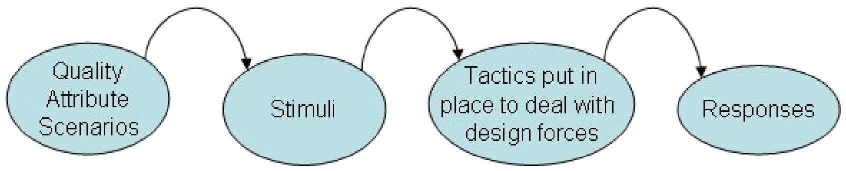
Paul Clements was the second SEI guy to talk. He turned out to be an excellent enthusiast (some would say evangelist) of the benefits of doing software architecture.

To start, he gave a really interesting definition of software engineering, apparently used by Davd Parnas. Software Engineering is about creating *high quality*, *multi-version*, *multi-person* software. I understand this to refer to the need for mission-critical software that must survive several changes beyond its first implementation, and that is complex because it has to do with meeting the requirements of several stakeholders. For this kind of software, you need software engineering – for lesser software, you can survive without it.

After a brief glance of the origins of software architecture (including references to Dijkstra and Parnas's early contributions), we had an interesting question asked: What do you think about agile methodologies? Paul said he was suspicious of agile methodologies: they seemed to be good to get projects out of the door, but were not proven to support *maintainable* software.

Paul says that large scale design decisions cannot be made by programmers. Architecture is an abstraction of a system, a mission statement for the programmer. It permits achievement of the required quality attributes: if you care about X, examine Y in the architecture. It also facilitates communication with the stakeholders. In fact, a while ago, Paul asked someone what they thought about software engineering, and the response was: "It has turned out to involve a lot of talking".

Paul also talked about architectural tactics, being a catalog of design approaches that mapped well-defined quality attributes to recommended architectural patterns. He mentioned Len Bass' Attribute-Driven Design work, which seeks to choose appropriate styles, patterns and tactics in given design tasks. The rationale behind this work is that there is a discernable correlation between design decisions and quality attributes.


Quality Attribute Scenarios
Stimuli
Tactics put in place to deal with design forces
Responses

Generally it seems that there is a lot of interest in trying to bridge the gap between getting requirements and coming up with a good design to satisfy those. Somehow, design always seems like a black art to me. People look at problems, and somehow come up with ideas of how to solve those problems. Can one really systematize design into a set of rules?

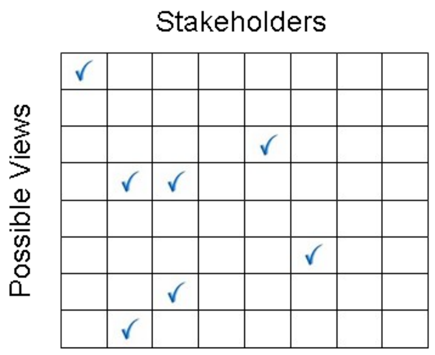
We then covered ATAM (Architecture Tradeoff Analysis Method), a method to analyse architectures. Essentially, it seems to consist of the following activities:

1. Extract required quality attributes from business goals.
2. Analyse the architecture and the approaches taken, to extract actual quality attributes.
3. These should both be broken down to detailed, measurable attributes, along with scenarios that illustrate them.
4. The required quality attributes are positioned in two dimensions (how important they are to the business (High / Medium / Low) vs how difficult they are to achieve (High / Medium / Low)). These are then prioritized, starting with those classified High – High.
5. Record risks by evaluating the architectural decisions, and deciding whether they meet these required quality attributes.
6. Stakeholders give scenarios that reflect their role and concerns about the system. These are prioritized, and compared to those expressed by the business and the architect.
7. Write a report on findings:
   - Strengths / weaknesses
   - Tradeoffs between attributes
   - Sensitivity points
   - Risks
8. Categorize themes behind the risks, and map these to business drivers and the architecture.

This seemed to me to be a common-sense way of evaluating any designs (not just architectures) at a point in the development process – getting a neutral party to evaluate whether or not the designs meet their stated quality objectives. It would really be a good idea to add this step into industry software design processes.

We also explored what it means to document architectures, in order to communicate them. Paul spoke about a set of documents, each conforming to a standard template for ease of understanding, and organized for ease of reference. He recommended: *"Document the relevant views, then add information that applies to more than one view, thus tying the views together".* The view types he spoke about were be *modules* (units of implementation), *components and connectors* (run-time, execution), and *allocation* (deployment, development environment). He said that Phillipe Kruchten's 4 + 1 View Model was an Object Oriented variant of this approach.

Paul discussed an approach to select views to document, essentially a matrix listing possible views, and marking their relevance to stakeholders. These should then be combined to reduce their number, and prioritized based on need. One should provide a documentation roadmap, providing context and navigation to the reader.

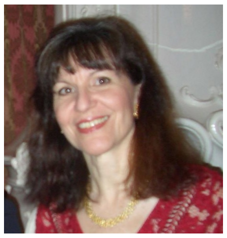


I thought that it would be a really good exercise to actually do one of these documentation exercises, to be able to see what this involved, what the views looked like, and whether they were actually useful.

Apart from describing what a architecture repository should look like, this also enabled what Paul called a *document-based culture*. Instead of having information in architects' heads, open to misinterpretation, and taking up most of the their valuable time in communicating the same information over and over, the documentation should be seen as the authority, the arbitrator in debates.

In question time, Paul said:

- Architecture Description Languages were enjoying much research attention, but were not used much in practice.
- Class diagrams (from UML) were actually a mixture of structure and run-time aspects, and so potentially mixed information which could be shown in different views.
- In general, he was less than enthusiastic about UML for architectural views, even considering the additions to UML 2.0, which were intended to support this. For example, he said that UML 2.0 connectors were not architectural connectors.



Micaela Serra (University of Victoria, Victoria, Canada)

Hardware/Software Co-Design of Embedded Systems

Micaela Serra is an animated Italian speaker, very passionate about her field. This made for rather entertaining listening. Micaela works, researches and lectures in the interesting area between hardware and software, especially in embedded computing. Unfortunately, this is not really my field, so I am really only able to give a brief overview of what I heard in the talk.

She told us about the need to design software and hardware together. In many cases, such as for medical devices, avionics, digital cameras, microwave ovens, elevators, mechatronic systems (like a car's brake and stability controllers) and cellphones, it is important to take a step back and understand both hardware and software. These systems often have difficult characteristics, like having to be real-time, mission critical, robust, distributed, having to have long-lived power sources, and being manufactured in serious volumes. Such design decisions should be the product of the collaboration between many stakeholders: marketers, software designers and developers, hardware designers, manufacturers and technicians, but would need to be seen as an integrated whole.

Often the hardware / software split is itself a design decision. Generally, the trend is to develop stuff in software, for programmability. Then, one may move to hardware from software for performance reasons, or from hardware to software for configurability.

This made me think of some work we are doing in the Espresso research group, at the University of Pretoria, with wireless sensor networks. The specific operating system we are using, TinyOS, is designed exactly in this way. The whole operating system to be deployed on a particular wireless sensor is composed from only those components you require. These may be thin wrappers over hardware, or wholly software. This choice often depends on the hardware you have available (or have room for), and the performance you require.

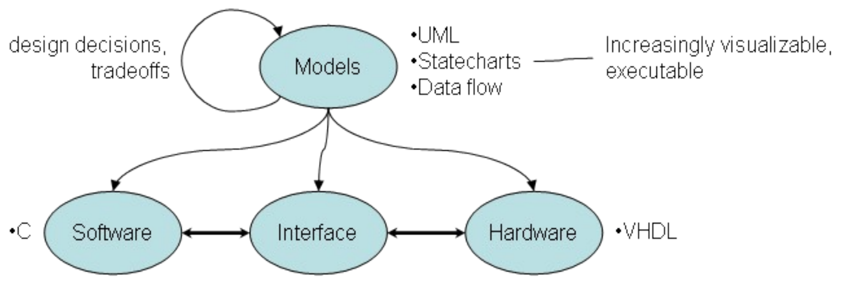
One of the problems in this area is integrating the design methods typically used in hardware projects and software projects. Design forces span both domains, and must be considered as a whole. A specific example from the medical field is implanted devices that should be increasingly remotely configurable in order to prevent unnecessary operations. This may even require handling unanticipated changes, over a number of years. This requirement often affects both the deployed hardware and software.

Another example is the use of Field Programmable Gate Arrays (FPGA's), a type of hardware that is composed from configurable processing 'cells', able to be changed to various specialized processing configurations by downloading configuration data.

Still another example is optimizing compilers for power consumption. Micaela showed how programming constructs and styles could affect power consumption, by showing us how to label state machines so that the least number of bit flips are required to change state (eg 1 -> 3 (01 -> 11) uses half the power of 1 -> 2 (01 -> 10)). This made me think of work Bruce Watson and his team are doing in optimizing large finite state machine performance in the Fastar research group at the University of Pretoria and Eindhoven in the Netherlands, as well as work on wireless sensor network power consumption being done by the Electronic Engineering Department at Tuks.

Another one of the problems in the area is the lack of sufficiently powerful tools. There is an increasing need for powerful simulation and visualization tools, to allow designers to understand the ramifications of design decisions, before manufacturing begins. There is also a need for 'executable specifications', something I took to mean: being able to specify both software and hardware models in an expressive enough language that would allow these tools to 'execute' them (a concept close to simulation?) to show designers what their decisions meant.

The diagram below is my interpretation of what she was talking about: a unified set of models that allows understanding and simulation of both hardware and software, and compilers that generate the appropriate hardware specification and software code. These compilers apparently need quite a bit more work to become industrial tools.


design decisions, tradeoffs
Models
•UML
•Statecharts
•Data flow
Increasingly visualizable, executable
•C
Software
Interface
Hardware
•VHDL

The third problem in the area that Micaela pointed out is the development of interfaces between hardware and software (an example is device drivers). This typically involves many months of painstaking manual work, which often has to be redone when the partition between hardware and software shifts, a new piece of hardware comes out, or a new piece of software must interface to it.

In all, this was a interesting lecture, showing some problems that we business system developers hardly ever have to consider, given that we accept the hardware platform we are given, and program on top of many layers of software that shield us from its details.

Bertrand Meyer wanted to know why the embedded world still persisted with languages like C, when there were languages that supported, for example, better safety, reliability and maintainability. Micaela said that the main reason was that the industry players in the embedded world were behind, and liked, C. They would change only when tools that supported other approaches, and languages, were better.

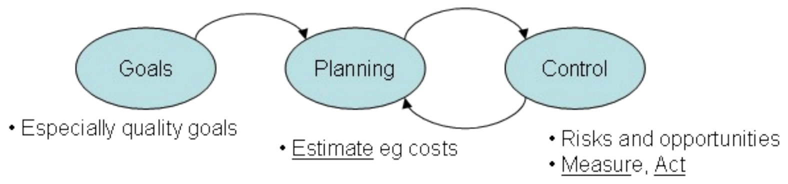
Rick Kazman (Software Engineering Institute, Pittsburgh, USA)

Strategic Software Engineering

Rick Kazman was the third SEI guy to speak. Whereas Len Bass told us about design issues in Software Architecture, and Paul Clements told us about why we should use an architecture, how we should evaluate one and how to document one, Rick focused more on the strategic issues surrounding software engineering (or, in other words, how to manage software engineering efforts from a strategic point of view – understanding the measurable value that certain activities contribute).

He started off with the usual dismal picture of Software Engineering painted by the 1995 Standish CHAOS Report, which almost everyone quotes to show just how far the terms "software" and "engineering" are from one another. (As an aside, I recently read David Parnas' view that software engineering is an unconsummated marriage!) At this point, Bertrand Meyer stood up and asked if anyone had read more recent versions of that report (apparently, these reports are ongoing measures of the state of software development). Later reports show that the situation is improving. Not many people say that!

The point that Rick wanted to get to was, by analyzing how and why projects fail, one can see that the majority of problems in software projects are caused by poor planning and control, where goals are not defined up front, not checked for feasibility, and progress towards them is not measured. The model I had in mind as he spoke was something like this:



This looks like the usual engineering feedback loop that people talk about when they explain how, if you want to manage something, you need to measure it. He used this as a basis to explain that his view of Strategic Software Engineering is to get software people to estimate, plan and control like a proper engineering project. His view is that these techniques (which look like ordinary management / operations research techniques) are not taught to software people, and are not applied widely in software projects.

He explained that risk and opportunity are quantifiable, and you can use this to manage projects better. On the risk side, if you know (or can estimate) the probability of something happening, and the loss that will occur if it does happen, you can calculate the risk exposure. On the opportunity side, if you know (or can estimate) the probability of benefiting from an opportunity, and the value that will accrue if you do benefit, you can calculate the opportunity potential. Then, you can plan your activities so that you minimize risks (if you know something is risky, don't do it), and maximize opportunities (if you know its risky not to do something, then do it.) Strategic software engineering methods explicitly optimize expected value in this way.

I wondered whether design forces, which are the things you must balance when designing software, are probability-based, as in the above discussion.

In the next series of slides, Rick went through a typical system integration scenario, showing how one could quantify the risks and benefits of different alternatives, and so come up with a reasoned choice. This was interesting, because it partially answered my question – several of the factors considered were concerned with how difficult the design and development effort would be in the different alternatives, and the chances of succeeding. These factors were also used to substantiate spending time and money prototyping uncertain areas, and in choosing a package that had less functionality, but that eased the integration job considerably. Using this method, the customer and management were shown the facts, and could make decisions based on how much risk (in the form of additional time and money) they were willing to take, for certain benefits. The prioritized risks also helped to show what should be done next in the development life cycle to gain the most benefit, as well as showing when over-engineering could be averted (this much is enough to manage an acceptable level of risk).

A couple of interesting little adages:

"An engineer must know:
- How much it will cost.
- How long it will take.
- How much of the problem it will solve."

"Excel – the software engineer's friend."

We then went through quite a detailed example of developing a project strategy. We made a list of important attributes of the project (for example, robustness, performance, development time), and the probabilities and severity of problems if we didn't meet acceptable levels. We also made a list of different techniques that we could use to assess these attributes (testing, analyzing a model, reviews, simulation), and a table of which technique could be applied to which attribute, the cost to do so, and the extent to which the application of that technique would fix the problems. The problem is to find which techniques to use, on which attributes, in which order, that most improve the project.

Different strategies were compared, for example choosing the lowest cost, the maximum benefit, or some cost-benefit tradeoff. We could compare which strategy would give the best results and choose that one. I thought that this was quite a nice decision framework, based on balancing risks, which could also be applied to other areas, such as choosing an architectural route from an as-is situation towards an architectural vision.

I wondered if this was a way of characterizing design forces and design decisions, making them more quantitative, and less 'rule of thumb'. I resolved to look further into these concepts in my research on design in software engineering methods.

Bertrand Meyer (ETH Zurich, Switzerland)

Design by Contract – Four Easy Pieces

Bertrand Meyer is a most entertaining lecturer. He runs around with the microphone asking questions of the audience members, cracks very dry jokes quietly, and takes you very carefully and clearly through the subject matter, leaving you feeling as if this was the answer all along.

Let me see if I can give you an idea of the impact he made on me. I have a list in my head of imminent computer people I have had the privilege of hearing in person, and that have had a tremendous shaping effect on my career. That list includes Niklaus Wirth, Christopher Alexander (OK, so he would be amazed to be called a computer person, but he certainly ranks up there), Grady Booch, Jim Rumbaugh, Ivar Jacobsen, James Cockburn, Kent Beck, Richard Gabriel, and now - Bertrand Meyer.

Falsifiable Statements and Software Engineering

He started his talk with the statement that he always tries to make falsifiable statements – statements that can be disproved. Basically, says Bertrand, in hard topics like maths and physics, the statements that one makes can be disproved either by argument, or by experiment. Even in soft topics (I assume he was alluding to Software Engineering), one should also make these kinds of statements.

Because Derrick Kourie and I have a Masters student (Mandy Northover) who is busy in the area of applying Carl Popper's philosophy to software development methods, this immediately caught my interest. According to my rather naïve understanding, Carl Popper said that a scientific theory is only the current set of statements that have not been disproved (falsified), and one should see scientific theories that way, not as an inviolate explanation of life the universe and everything. In short, Mandy's work explains that applying this philosophy to developing software means something closer to the approach taken by agile methods (lets start with something simple and refactor it when we need to, or when one of the design statements has been "disproved"), and further away from the approach taken by those methods that develop a huge architecture and try to make it accommodate any change that comes along (trying to make the design "disprove"-proof).

The One Notation Principle

Bertrand is of course the inventor of the Eiffel programming language, which is renowned to be very supportive of more formal software engineering approaches. His overall approach was to talk about different aspects of software engineering, and illustrate them in Eiffel. If I had to sum up what I thought his talks were about, I would suggest he is saying:

> "We have too many notations, and too many disparate steps in developing software. These include natural language for requirements, modeling languages for architecture and design, and programming languages for development. Much of software engineering, and even much of

software engineering research, concentrates on bridging those gaps. Why don't we collapse all of these into one notation that supports the fundamentally necessary concepts for developing large, complex systems? In this notation, we would fill in the information at the abstraction level where we currently are, leave the rest until we get to it, and hide the rest when we don't want to see it. In this one notation, we can then use tools that check correctness and enforce component contracts far better than before."

Having been thinking about topics like Model Driven Architecture and software development tools for a while now, I knew that these often were about exactly that, and tried to help developers to overcome complexity. So, I went with that assumption, to see where it would lead to, while keeping in the back of my head the thought that one-language approaches are always easier than the integration of packages, legacy, new components and middleware glue that we contend with regularly.

More about this theme as we progress through Bertrand's talks…

Patterns

He first addressed some of the trends in software engineering today. Patterns, he says, are a step back to craftsmanship – they are rules of thumb as craftsmen used to use, rather than more fundamental concepts, like Donald Knuth's algorithms. He considers them a half-good idea, the best in two decades.

He went through a well-reasoned step-by-step consideration of what it takes to implement a pattern like Observer in programming language code, and some of the challenges faced by implementers, given the mechanisms in programming languages today (examples are Java listeners and .Net delegates). He showed how complex GUI "callbacks" (a form of observer mechanism – what to do when a button gets pressed) can get. The reasons had quite a lot to do with the corner that object orientation paints us into: procedures are not first-class citizens, and so we need separate objects to encapsulate events and their handlers. The code to handle this is often generated by tools, which encourages myopia (you are only allowed to see what you are allowed to change, so you have difficulty knowing how it fits in). If you're doing it yourself, a common tactic is to "copy and paste" (this is too complex to remember how to do, so I'll just use my previous stuff that worked). The approach Bertrand showed in Eiffel used the concept of an agent (an "operation object" that seems to have its roots in the lambda calculus, where I first learnt about functions with bound and unbound arguments).

A well-documented API is always better than a design, he says emphatically. For example, we don't program quicksort anymore – we call a library function. For this reason, he said, most patterns could and should be available as a set of components, with well-defined APIs. 65% of patterns he says are componentisable, 26% partially, and the rest not. Bertrand says that the most important task in architecture today is to componentize useful patterns. The above "agent" and other patterns in Eiffel were being turned into such components in the work of a student of his, Karine Arnout. He says that the patterns people do not like this (turning beauty into "mere code" is his interpretation), and have even rejected a paper of his that suggests this as a way to go.

Interestingly, he says that using patterns as designs (that map to a whole collection of language elements) shows a failure on the part of the language designer. He also says that the only problems in software engineering today are language problems. I take that to mean that when the languages don't allow us to concisely express our problems and solutions, and don't help us to ensure that those are correct, this leads to complexity and its consequent problems. And, that means language matters more than we care to admit. Food for thought (or at least to fan the flames of the language wars currently raging on the Espresso forum…).

I see Bertrand as one of those people that carefully goes over ground that others have rushed over – these issues are not really even debated in patterns circles anymore. Although at first some people suggested expressing combinations of patterns in a "language", it seems that the prevailing thought is that you can't express patterns in language form - it is assumed that patterns are just design-level mechanisms. Grady Booch says the amazing thing about patterns is that when applied correctly, they disappear. In other words, they are a conceptual tool for thinking about a system's design, but at the code level, they are merely seen as vague design constraints. Here Bertrand says you should be able to use them directly in your code, as components drawn from a library. I look forward to reading the work of his student.

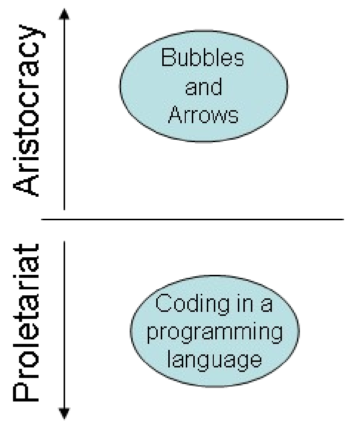
I asked him if he thought that Model Driven Architecture was also a symptom of failure on the part of language designers. He said he considered it bad that model and code were assumed to be different, and therefore needed translation from one to the other. They should go towards expressing model and code ("detailed model") in one language.

Architects, he also says, should not confine themselves to only the design level – they should be able to go up and down the levels of abstraction, but should always know where they are. He also says that software architecture is high level programming, and if you're bad at programming, you'll be worse at architecture. (Bad news for some architects I know who never programmed very much, and now do none at all).

On the broad subject of UML as the preferred notation for designs, as opposed to expressing them in Eiffel as he suggests, he had this to say: Textual languages have the characteristics of exactness and detail, which are not generally present in graphical notations. He also says there's "no reckoning in bubbles and arrows" (they don't make falsifiable statements). Again, the theme of saying things exactly, so that people can disprove them. It appears he doesn't care much for the stereotypical architect that wanders around with Powerpoint and a pocket full of whiteboard pens, explaining away to all who would hear.

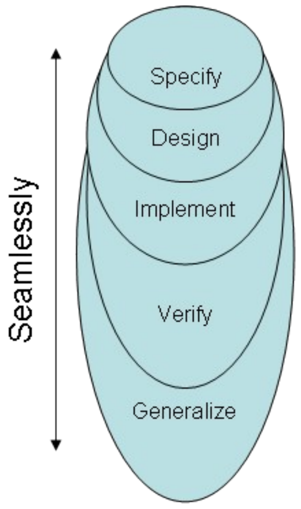
Seamless Software Engineering

Next, Bertrand turned his attention to software engineering methods. He spoke about the division of labour in typical software projects as:



The aristocratic architects and designers make broad decrees, and then tell the mere coders to "make it so". This should not be so, as there was a lot of overlap and commonality between the phases. Analysis was looking at the problem space, design was looking at the solution space at a high level of abstraction, and implementation was looking at it in detail. A good method needed to cover everything from analysis to design, to implementation, to verification (testing), and should then take some time after a project was finished to generalize what was done, to produce reusable assets.

Everyone needs a diagram of their methodology. Bertrand was no different, and gave us this one:



He said that Eiffel aimed to be the common notation to express the information necessary at each phase, to facilitate doing this seamlessly. You should use Eiffel to reason about your problems, and to specify your solution in increasing detail. He called this the single model principle, which he said you should try very hard to get as close as possible to.

At this point he made what we thought was a most interesting statement:

> *You don’t need to be imprecise when you abstract.*

He said it was very difficult to be both precise and abstract, but very easy to be just one. Too often, to be precise was to say everything, and to be abstract was to say nothing.

Again on the topic of graphical architectural notations, he said that architecture needn't be pictures. Pictures were good for visualizing and communicating, but were not precise.

I thought it was interesting that the UML protagonists are currently working on a textual variant, I assume for a very similar reason. XMI (XML Model Interchange language) is a textual notation for exchanging designs between different tools - textual because the essence of the design is expressed in a universal language for expressing data structures, XML. OCL (Object Constraint Language) is a textual design constraint language, needed because only very few design constraints can be visualized, or specified accurately enough, in graphical form. Interestingly, Eiffel has elements of all of these, in one precisely defined whole.

Of course some people think better graphically. If you need a graphical visualization of a design, EiffelStudio, Eiffel's IDE, provides a graphical view. Bertrand believes that most of the documentation should be in the software itself, with tools that extract views as needed. This graphical modeling view is an example of this. Here, I thought about XP and its insistence on the code as the ground truth, and models as illustrative, but throw-away. There seems to be an interesting convergence of views here.

One bit of advice that Bertrand gave about documentation was this:

> Documentation should be one level of abstraction higher than that which it documents.

This seems at first to be a strangely obvious thing to say. On reflection, though, how many comments in code merely express what the code says, in English? The comments should explain *why* the code does things a certain way.

I wondered a bit about whether Eiffel could in fact do everything UML does. UML has several different models, some containing base information (class and state) and some illustrative (sequence, collaboration, use case). Would the programming model in Eiffel be sufficient to show the base models? Also, how would one show some of the illustrative aspects? I also wondered about some of the concepts at a higher level than objects and classes: what about components (generally defined as separately deployable collections of objects and other resources), connectors (as in some Architecture Definition Languages), and architectural qualities (such as performance, robustness, security and so on).

Bertrand also criticized UML for enforcing a too early distinction between function and attribute in class diagrams. As in some languages' idea of component properties, one should be able to access or modify what is externally visible as an attribute, but which could internally be implemented as a stored value, or a calculated value.

He stressed the need for a distinction between an 'ask' function and a 'do' function when designing object interfaces. The first should not have any side effects – "asking a question should not change the answer". For this reason, Eiffel had very strict design principles, for example to enforce this distinction. Even in such operations as accessing a result set with a cursor, there was no '.next' (which moved the cursor on one position), rather there was a '.item' (ask), and a '.forth' (do).

Design by Contract

Next, he went on to the concept of a contract, and the topic of "design by contract". A contract is an explicit statement of the purpose of a software element – it should be specified precisely, and should be part of the software element itself. A system is a structured collection of cooperating software elements. They cooperate on the basis of these precise, explicit contracts.

A contract specifies what is *expected*, what is *guaranteed*, and what will be *maintained*. A contract is a set of pre- and post-conditions, signifying what should hold before execution, and what is guaranteed to hold after execution. Each condition is a set of assertions. Each assertion is a set of clauses, which should all hold true. Each clause is a boolean expression, a statement in first order predicate logic. These are not just simple expressions specifying, for example, allowable ranges of a balance, but can specify qualities such as 'readable' or 'writeable', and advanced logic constructs such as 'implies' or 'conforms_to'. I was amazed to see the extent of the contract specification inside the Eiffel libraries – Bertrand showed us a section of the contract of a collection class. It is interesting to note that these pre- and post-conditions (minus clauses that refer to internal information) are shown to the client as the externally visible contract, and they are also used to assert that the inner workings of the class do in fact meet that contract.

Contracts vs Testing

A way of looking at programming like this says: If you don’t satisfy my pre-conditions, its *your* problem; If I don’t satisfy my post-conditions, its *my* problem. In this way, one can more easily assign responsibility for programming bugs. I wondered how this relates to test-driven development – where tests are written that exercise the code completely and consistently. Are pre- and post-conditions, or should they be, more precise ways of specifying unit test conditions? Is this only the case for unit tests? How does one extend this to integrated testing (testing systems of collaborating components)? Also, test-driven development is cited as a way of designing a class – you specify basically what its contract is before you program it. Isn't this exactly what we're talking about with pre- and post-conditions?

Contracts and Design Principles

An advantage of using pre- and post-conditions in this way is to prevent what Bertrand calls defensive programming – in other words, having to test all the way through your code for unexpected results, so that you can catch problems early. This of course has the effect of making code that much more complex, obscuring the essential logic and often decreasing the code's reliability. If the contracts are enforced, you can simplify coding, and ensure better reliability by relying on verified input and output.

There should be no hidden clauses in the contract, but they may be inherited from super-classes. I was interested in the discussion on how pre- and post-conditions can be changed in derived classes. Bertrand calls this honest subcontracting, and says that it defines the proper use of inheritance: constraining inheritance, polymorphism and dynamic binding so that they function as intended.

Apparently, when you override a method in a derived class, you may only weaken the pre-conditions, and strengthen the post-conditions. For example (thanks Derrick), a particular class method may add an element to a collection. Its pre-conditions may specify that the collection must contain each element only once, and the post-conditions that the resulting collection will be the original collection with an added element. The overwritten method in the derived class may weaken the pre-conditions (it may accept a collection with multiple copies of elements), and strengthen the post-conditions (in addition to adding an element, it will also ensure that the collection is sorted afterwards).

Is this is a more precise statement of Barbara Liskov's Substitution Rule (an object expecting an object of the super-class should be able to use an object of any sub-class)? It seems to be - the pre-conditions of the super-class satisfy those of the sub-class (and so the sub-class can accept input meant for the super-class), and the post-conditions of the sub-class satisfy those of the super-class (and so the sub-class generates output acceptable to the super-class). Therefore, the sub-class can be substituted for the super-class.

Some Thoughts about Teaching Programming

To close, Bertrand turned his attention to his teaching experience. He says he has seen many approaches, from teaching programming languages to completely formal methods. He says it is important to know that teaching programming is not about getting people to hack around in a language – you must teach *concepts* that will stay with people no matter what language they use.

Teaching programming in todays' world is becoming somewhat of a challenge – students have now been writing programs since childhood, they are interested only in the coolest applications like games, and they can get solutions for any of your projects directly from the internet.

How do you teach them what is really useful, ie the ability to see and understand the big picture? Software is increasingly becoming large scale and more complex than any other engineering discipline. A typical Boeing has about 45 million lines of code running in its systems. Some software is becoming more like human systems –adaptive, and doesn't have to work all the time (eg political systems and cities). Modern software needs both the reliability and safety of engineering, and the scale of human systems.

How does one expose students to these concepts? Bertrand gets them to work on existing large systems, starting out as customers and working their way in to become contributors. This he terms progressive opening of the black box. He uses some software developed by his department, but I thought that this principle could just as easily be applied to using an Open Source project. This also relates to Richard Gabriel's belief that the ability to see good programming in Open Source projects is an excellent way of learning to program.

Above all, says Bertrand, don’t pontificate – young people like few things less than having old people telling them what they should know. They must learn by experiencing. For example, Bertrand says they will learn the art of abstraction very soon if you throw large quantities of code at them. They will have to abstract to survive. They must also learn by imitation and exploration – encourage these.

What I enjoyed most about Bertrand Meyer is his broad knowledge of the problems currently found in industry, and the strengths and weaknesses of the languages and approaches used. He is not a closed-minded Eiffel bigot – I believe he has carefully looked at the alternatives, and has crafted what he believes is a better way. He presents his approach humbly and simply, building up to more complex solutions. I certainly will be looking into Eiffel in the not-too-distant future.

Conclusion

To end, let me express my gratitude to Professor Judith Bishop and her team for a most informative event. The speakers were enthusiastic, top class, and approachable. A sincere word of thanks to them too. We got to spend a week engaging in career-shaping conversations, to be remembered long after we went home.

I'm sure that it is such events as this one that will contribute to a growing South African software development industry.

Lets do this again soon.