Timeless Software

Summary

Large-scale, enterprise software, like SAP ERP, must be able to leverage and adapt to change while delivering continuity and reliability at low cost. SAP software is architected on principles that enable fast integration of, and the ability to reliably manage and exploit, changing and emerging technologies

Author(s): Dr. Vishal Sikka, Chief Technology Officer

Company: SAP AG

Created on: 31 October 2008

Author Bio

Dr. Vishal Sikka, Chief Technology Officer (CTO) of SAP, leads the company's technology and innovation strategy across its product portfolio. Vishal is responsible for ensuring a clear and harmonized road map for SAP products to deliver ongoing innovation and long-term value to customers worldwide. Furthermore, Vishal leads the company's forward-thinking efforts around emerging technologies using design thinking to build SAP next generation products. He is responsible for SAP's global research efforts and is chartered with SAP's architecture governance and standards.

Prior to his role as CTO, Vishal was the senior vice president of architecture and chief software architect at SAP, responsible for the road map and the direction for the architecture of SAP's products and infrastructure. Before that, he was head of the advanced technology group responsible for strategic innovative projects.

Before joining SAP, Vishal was area vice president for platform technologies at Peregrine Systems (Remedy), responsible for application development and integration technologies and architecture. He joined Peregrine following the acquisition of Bodha, Inc., where he was founder and chairman/CEO. Bodha developed technology for non-invasive, service-based integration of enterprise applications and for semantic information integration.

Vishal holds a doctoral degree in computer science from Stanford University in California, and his experience includes research in automatic programming, information and application integration, and artificial intelligence at Stanford, at Xerox Labs in Palo Alto, and at two startup companies.

Vishal reports to SAP Co-CEO Léo Apotheker and is based in SAP Labs Palo Alto, United States.

Table of Contents

Timeless Software – Part 1

Change and Renovation

I'm sitting on a Lufthansa flight in the newly refurbished cabin of a Boeing 747, and I cannot help but marvel at the evolution of this beautiful plane. The 747 was first released in the 1960s, before I was born. Today, the machine is so fundamentally different from its beginnings. It has an advanced cockpit, new communication and navigation systems, quieter, more efficient engines, and an updated, modern cabin, fitted with comfortable seats and new amenities. And yet, essentially, the Boeing 747 is the same as when it was first born: it possesses the same principles of flight, the same levels of safety and reliability, the same optimizations around the essentials of travel requirements, fuel consumption, and maintenance.

At its core, the 747 is engineered around safety, reliability, and maintainability. And like a rudder, these core attributes have guided and stabilized the evolution of the plane through a long period of changing customer expectations, advancing technologies, new market requirements, and shifting government regulations.

36 years after delivering our first packaged application, we've learned at SAP that successful large-scale enterprise software follows essentially the same trajectory and lineage. Enterprise software solves fundamental problems that businesses face every day, over generations of business change and of technological change and, in doing so, it continuously evolves in a constant cycle of renovation. I call this Timeless Software, and want to write here about what some of its fundamental characteristics are, and how it will help define our software for the next several generations of changes to come.

Long-Lived Relationships

Today, SAP software covers a massive breadth of business activities. The functionality of the SAP Business Suite deals with a large spectrum of business processes, from finance and human resources to sales and service, from planning and procurement to manufacturing and logistics, from managing supply chains to managing business strategy, from decision-making to governance and compliance. In addition, the suite comprises variations on these processes across 100+ countries and 25+ industries. Despite this massive reach, customers expect a fundamental degree of coherence, stability, reliability, and integration across their entire software landscape.

The demand for stability, given the mission critical nature of many of these business processes, coupled with the fundamental ways in which customers deploy the software to mirror the uniqueness of their business, means that our software and our relationships with customers, are very long-lived and often last decades. Over this long lifespan, customers always expect the software to contribute to two fundamental metrics:

- **Control costs**, by ensuring that the software is integrated, comprehensive and reliable, easy to operate and cheaply administer
- **Allow for growth**, by ensuring that the software addresses differentiated areas and is easy to evolve, change, and integrate as necessary

So this is the essential duality that our customers expect from their IT landscape: Deliver **operational efficiency** via coherence and stability, while **enabling business growth** and managing change necessary to survive and grow. And our prime requirement then becomes: Enable evolution of our software without disruption; provide a large breadth of stable functionality, over generations of change. And it is around this requirement that we seek to design and architect the evolution of our software.

The Dynamics of Change

Business Change

What are the dynamics of constant change? We know that business requirements change all the time; markets evolve, businesses are bought and sold, regulations change, and the circumstances that govern how customers purchase products evolve constantly. Even the ordinary, day-to-day challenges of competition require IT organizations to constantly shift and evolve their landscapes. None of this is for the faint of heart.

New Technology Layers

The technological layers are also drivers of change and can transform the way people interact. This year, we estimate that nearly a billion business users worldwide will use various mobile devices to conduct business. Every year we see roughly two new major UI paradigms. Just in the last 3 years, we have seen the iPhone, Google's work on Google maps, and highly interactive web applications enabled by AJAX, Adobe's work with AIR, Microsoft's work on Silverlight, and others.

Emerging Programming Languages

Even programming languages, and programming models around them, continuously evolve. Roughly every 10 years a major new language emerges, and minor ones emerge every 3 years or so, well within the lifecycle of large-scale applications. New programming models and robust developer communities can quickly emerge around new languages. The language Ruby, for example, is thought to have reached a million programmers faster than any other language ever.

Evolving Infrastructure

The three key infrastructural building blocks: processors, network and memory, evolve continuously and often non-linearly. And this evolution sometimes enables, or requires, new architectural paradigms. For instance, cheap main-memory and elastic farms of simple servers have enabled fundamentally new ways of analyzing large amounts of data. Similarly, multi-core processors require rethinking application programming to better utilize their parallelism or risk slowing down.

So large scale software, over its lifetime, is subjected to change continuously, business change, as well as change across all the technology layers that it inhabits.

As I look to the future evolution of our products for the next generation, this becomes our essential challenge:

- How do we build applications to serve the needs of every user, and every activity, in every business around the world?
- How do we do so effectively, efficiently, and with maximum coherence?
- How do we evolve these applications, their ongoing change, consumption, delivery and integration, as well as their connection to the present, across generations of change?

I believe the answer to this essential question is Timeless Software.

Understanding Timeless Software

Enterprise applications are built using a collection of programming models and languages that describe their content, are executed using a set of corresponding containers or run-time, and continuously change over their lifetime. My sense is that these three constructs form the essence around which we need to understand Timeless Software, its characteristics, and how we achieve it:

- **Content**, i.e. the application content, the UI content, the integration content, to represent and serve the activities of users
- **Containers**, i.e. the runtime(s) that this content inhabits, and
- **Change**, i.e. the ongoing operation and evolution of both the content and the containers over the lifecycle of a solution while maintaining a continuous link with the past

There are other aspects, to be sure, but these are the three basic ones and I want to share my view on their evolution in my next blog in this series.

Timeless Software – Part 2

The Evolution of Content Creation

Enterprise systems cannot rely long-term on any one programming language. Alan Kay once observed that there is a major new language every ~10 yrs and several minor ones in the interim. So over its life span, a major enterprise system sees adoption curves of several languages. Just in the last several years we have seen very rapid adoption of .Net languages, Ruby, Python/Perl/Php, Javascript, and others. Perhaps even more interestingly, programming models emerge around these languages, and often the success of a programming model, e.g. JEE or Ruby-on-Rails, brings with it a large community of programmers, drives the adoption of the language, and an explosion of software artifacts around it.

Domain-Specific Languages (DSL)

But lots of languages and dialects also exist for other reasons: There are many different domains and problem characteristics within enterprise systems and for each domain, unique combinations of syntax, tooling conveniences and programming models emerge over time. From Jon Bentley's "little languages" to the modern-day notion of "domain specific languages", there are many variations in essentially the same exercise: expressing meaning in convenient, specialized ways.

There are lots of programming models and domain-specific languages around user interfaces, for instance. Data has lots of variations too. Modeling business data, languages for querying, reporting and analytics, for search (as Google showed with their map/reduce programming model), for managing XML based or other hierarchical data, and others. Describing flows, events, rules, software lifecycle, and other aspects each bring their own variations, and the same thing happens in specific application areas and in particular industries. Over time, with successful adoption, these abstractions and conveniences increase.

Our own ABAP, for instance, has, over time, integrated several programming models within a general purpose language: abstractions and extensions for data access, for reporting, for UI, even object-oriented programming within ABAP, in the form of ABAP objects. Java, similarly, grew over the years in lots of domains and ultimately the JSR institution served to systematize the inclusion of extensions and programming models within the language.

And there are similar examples in other domains, in hardware design for instance. Even cliques of teenagers invent their unique DSLs for texting.

Programmer Segmentation

Another key source of diversity in programming stems from the nature of the programmers. Programmers bring different degrees of training/understanding in computer science concepts, in business, and in particular domains. So languages and language constructs, as well as specific abstractions emerge for different programmer segments, be it system programmers, business analysts, administrators, or others.

This diversity is great, insofar as it enables useful abstractions and separation of concerns, so different classes of problems can be dealt with uniquely. After all, the world does not speak one language, as any visit to the UN would demonstrate.

But the challenge is the resulting complexity that these isolations create. The various abstractions and specializations lead to islands of diverse, non-interoperable languages, slower run-times and more complicated software lifecycle management. Like barnacles attaching themselves to a host, these variations often lead to increased landscape complexity and higher costs of operation.

Requirements for an Enterprise Programming Model

My sense is that we need an enterprise programming model that is deeply heterogeneous yet integrated. One that enables expression of meaning in a wide variety of simple and convenient ways, including ways yet to be invented, without losing coherence. In my view the next enterprise programming model needs to:

- Enable developers across lots of domains and specializations to use their native abstractions and conveniences
- Support a family of integrated domain-specific languages and tooling conveniences to build software artifacts with maximum efficiency and productivity
- Use a powerful glue to bind these diverse elements together
- Allow itself to be extended by communities and developers of various sorts in lots of different ways
- Be able to integrate the next great languages, including languages yet to be invented, and even allow itself be renovated and embedded in other programming models

Glues that Bind

Some advanced development work we've done in our labs indicates that such an integrated design-time environment is indeed possible and can bridge a heretofore uncrossed divide between families of highly specialized DSLs that are nevertheless integrated into a coherent whole. A key piece of this puzzle is a glue that binds the various DSLs together. The glue in this case, is a mechanism that takes a base language, such as Ruby, and uses capabilities such as reflection, to extend the base language with the grammar of new DSLs in a seamless way. The timelessness comes from being able to add new DSLs dynamically to the base language, completely incrementally, without knowing about these in advance.

We have experimented with several DSLs that plug into a glue and the glue in turn integrates seamlessly into a base language such as Ruby or Javascript. In a promising effort named BlueRuby conducted by our SAP Research team, we have demonstrated how standard Ruby code can be run natively inside the ABAP language run-time, thereby achieving the benefits of both flexibility in Ruby programming and the enterprise-grade and robust ABAP environment. I see several exciting developments ahead along these lines that will lead us to new paradigms in extremely efficient content creation without losing coherence.

The Evolution of Containers: Next Runtimes

Enterprise run-times are faced with a significant challenge of optimizing the execution of the diverse and heterogeneous language landscapes described above. So if the content is to be built with maximum efficiency of expression and flexibility, then the containers need to enable maximum efficiency in execution. Our key challenge then is to bridge this divide between flexibility and optimization. In layered architectures, and with the first several years of service-oriented architectures behind us, we often take it as a maxim that the benefits of flexibility and abstraction come at the expense of optimization. We take it as understood that layers of abstraction, by creating an indirection, usually cost in performance. But I believe this is a false divide. Run-times need to separate meaning from optimization, and diversity in design-times need not lead to heterogeneity in run-times.

Operating Across Layers of Abstraction

More than a decade ago, I examined one aspect of this issue in my own Ph.D. work, in looking at how meaning, specified in highly generic logic-based languages, could be executed optimally using specialized procedures that could cut the layers of abstraction to achieve significant optimization compared to a generic logical reasoning engine. The principle underneath this is the same one – by separating meaning from optimization, a system can provide both:

- the efficiency and generality of specification in a wide variety of specialized dialects interoperating over a common glue,
- a very efficient implementation of that glue down to the lowest layer possible in the stack, across the layers of abstraction

There are examples of this principle at work in other areas in our industry. The OSI stack implements seven very clean layers of abstraction in the network, and yet a particular switch or a router optimizes across these layers for extreme runtime efficiency. Hardware designers, similarly, use a variety of languages to specify various hardware functions, e.g. electrical behavior, logical behavior or layout, and yet when a chip is assembled out of this, it is an extremely lean, optimized implementation, baked into silicon. Purpose-built systems often can dictate their requirements to the platform layers below, whereas general-purpose systems often do not know in advance how they will be utilized, and can often be suboptimal compared to purpose-built systems, but more widely applicable.

Managing State Across Boundaries

But beyond crossing the layers of abstraction, run-times have an additional burden to overcome. In enterprise systems, we are often faced with tradeoffs in managing state across boundaries of processes and machines.

There are three key building blocks in computing: networks, i.e. moving data around, processors, i.e. transforming data, and state, i.e. holding data, in memory or on a disk, etc. And different types of applications lend themselves to differing optimizations along these three dimensions.

Several years ago, when dealing with some difficult challenges in advanced planning and optimization, our engineers did some pioneering work in bringing applications close together with main-memory based data management in our LiveCache technology. The result was successfully implemented in the SAP Advanced Planner and Optimizer. This key component of SAP Supply-Chain Management, demonstrates how locality coupled with a highly purpose-built run-time offers a unique optimization on network, state, and processing.

Main-Memory Data Structures

More recent work in business intelligence demonstrates that when it comes to analytics, a great way to achieve performance improvements and lower costs, is to organize data by columns in memory, instead of rows in a disk-based RDBMS. Now we can perform aggregation and other analytical operations on the fly within these main-memory structures. Working together with engineers from Intel, our Trex and BI teams achieved massive performance and cost improvements in our highly successful BIA product. We are now taking this work a lot further; in looking at ways to bring processing and state close together elastically, and on the fly, and by looking at ways that the application design can be altered so that we can manage transactional state safely, and yet achieve real-time up-to-date analytics without expensive and time-consuming movement of data into data warehouses via ETL operations.

Hasso's New Architecture

In an experiment we dubbed Hana, for Hasso's new architecture (and also a beautiful place in Hawaii), our teams working together with the Hasso-Plattner-Institut and Stanford demonstrated how an entirely new application architecture is possible, one that enables real-time complex analytics and aggregation, up to date with every transaction, in a way never thought possible in financial applications. By embedding language runtimes inside data management engines, we can elastically bring processing to the data, as well as vice-versa, depending on the nature of the application.

Optimizing for the Enterprise

Enterprise systems with broad functionality, such as the Business Suite or Business byDesign, often need several types of these optimizations. One can think of these as elastic bands across network, state, and processing. Large enterprises need transactional resiliency for core processes such as financials, manufacturing and logistics. They need analytical optimizations, ala BIA, for large-scale analytics over data. They also need LiveCache style optimization for complex billing and pricing operations. They need to support long-running transactions to support business-to-business processes that work across time zones, they need collaborative infrastructure for activities such as product design, and others. Each of these patterns consumes the underlying infrastructure, memory, network and processing, in fundamentally different ways.

Limitations of Cloud Platforms

This breadth is one key aspect that the existing SaaS offerings are extremely narrow in scope. Serving broad enterprise functionality off the cloud is a fundamentally different architectural challenge, than taking a niche edge application, such as sales force automation or talent management, and running it off what is essentially a large-scale client-server implementation. My sense is that enterprise ready cloud platforms will enable extremely low costs of running cloud services that have a broad footprint: transactional, analytical, long-running and others, with extreme ease of development and extensibility. We have some early promising results in these areas, but neither the current SaaS offerings, nor any other cloud platform I am aware of, can address this challenge for the foreseeable future.

So to summarize, I believe the next great run-times will implement the glue at lowest levels possible in the stack, cutting across the layers of abstractions that make developers' lives easy at design-time but are not needed at run-time. These runtimes will flexibly enable various different application-oriented optimizations across network, state, and processing and will enable execution in specialized containers or consolidated containers, in elastic, dynamically reconfigurable ways. This deployment elasticity will take virtualization several layers higher in the stack, and will open new ways for customers to combine flexibility and optimization under one unified lifecycle management, the final piece of the puzzle.

The Evolution of Change: Lifecycle Management

We've had a look at the evolution of Content and the evolution of Containers, but perhaps most important of all is the evolution of Change, managing the lifecycle of a system over the continuous change in its contents and containers. Enterprise software lives a very long time, and changes continuously over this time.

Developers often do not think beyond the delivery of their software. For some, lifecycle management is only an afterthought. But lifecycle management is essential to ensure continuity in what is usually the very long life of an enterprise system. It is the embodiment of the relationship that the system maintains with the customer over several generations. Lifecycle management encompasses several aspects:

- change in functionality
- change in deployment
- integrating a new system with an existing one
- ongoing administration and monitoring

Working with Legacy Systems

One of the fundamental pre-requisites of lifecycle management is the ability to precisely describe and document existing or legacy systems. This documentation, whether it describes code, or system deployment, is a critical link across a system's life. ABAP systems have well-defined constructs for change management, software logistics, versioning, archiving, etc., as well as metadata for describing code artifacts that makes it easier to manage change.

Consuming legacy software often means understanding what is on the "inside". Well-defined wrappers, or descriptors, of software can help with this. But it is also often necessary to carve well-defined boundaries, or interfaces, in legacy code. Such firelaning, which has long been a practice in operating systems to evolve code non-disruptively, is essential to manage code's evolution over the long haul.

Service oriented architectures are a step in this direction, but having legacy code function side-by-side with "new" code often requires going far beyond what the SOA institution has considered so far. It requires having data– especially master data – interoperability, enabling projections, joins and other complex operations on legacy code. It requires having lifecycle, identity, security, and versioning related information about the legacy code. It means having policies in place to manage run-time behavior, and other aspects.

Most of these steps today are manual, and enterprises pay significant integration costs over a system's lifetime to manage them. Over time I see this getting significantly better. But it starts with provisioning, or enabling, existing code to behave in this manner, carving nature at her joints, as Alan Kay once told me the Greeks would say. I also see incumbents with an existing enterprise footprint, as having a significant advantage in getting here. It is often far easier to carve a lane out of existing code, than it is to replace it.

The Next Generation of Instrumentation

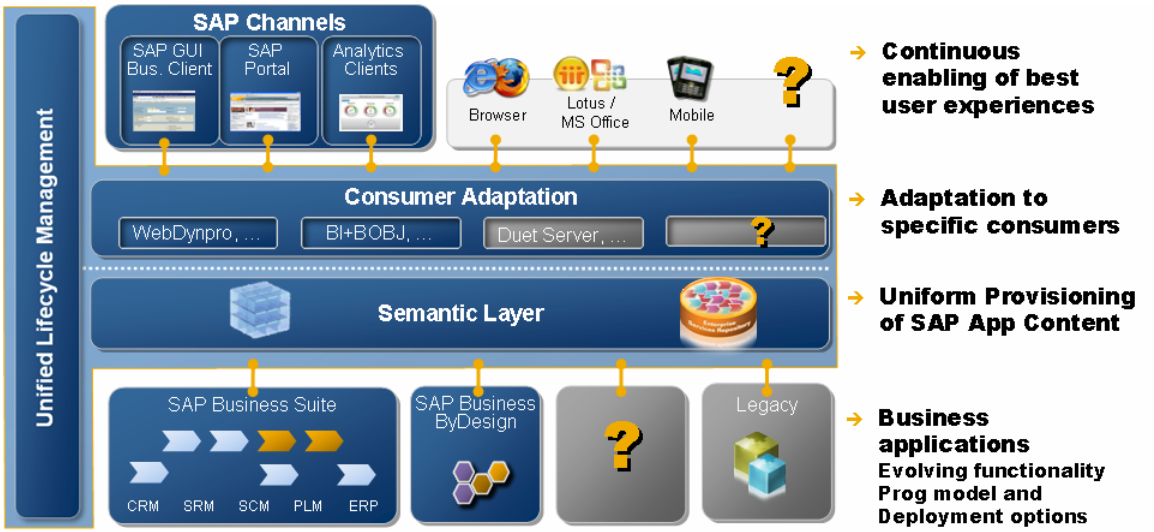
Great lifecycle management is the essential change management mechanism. My sense is, next generation lifecycle management will enable systems that can easily be tried, consumed, extended, added to, removed from, projected on, integrated with, etc. This will be achieved by enabling every artifact in a system to be measured, managed, and tested. We will see existing and legacy code being instrumented for administration, for documentation as well as for integration. This will require us to provide precise mechanizable specification and documentation of all important aspects of the system as a key ingredient. The specification of a system's behavior, its usage, service-levels and policies describing its operation, especially for security, access and change, will be fundamental to this. We already see efforts in this direction towards precise, mechanized specifications of system behavior and we will see more of this. SAP has already taken some steps in this direction with our enhanced enterprise support offering, that enables a business to lifecycle manage system landscape across their entire business from one console.

Deep interoperability between design-times, run-times and lifecycle management, will enable us to combine deployment options in ways that were not possible before. For the foreseeable future we see customers employing some parts of their processes as on-demand services, but deploying most of their processes on-premise. Our lifecycle management frames will ensure that customers can make such deployment choices flexibly.

Timeless Software – Part 3

The Timeless Software Evolution of Our Products

We've seen in the previous two blogs how the dynamics of change lead to the evolution of content, containers, and the evolution in the way change itself is managed. The illustration below shows how our systems will evolve and adapt to new consumers, new technologies, even new languages, how it will be provisioned for a run-time architecture that can optimize across network, state, and processing, and how unified lifecycle management provides a secure foundation for executing change and adopting to new



requirements.

Our portfolio of products will continually evolve along these principles. As the picture above illustrates, we will continue to enhance our massive yet coherent breadth of functionality, to reflect ever increasing business activities across industries, geographies, and roles. This functionality will be built using an evolving programming model, often in languages that have not yet been invented. And will be deployed in new ways, in the cloud, as appliances, on-premise, and all of the above. This functionality will be exposed for wide varieties of consumption, across consumers, business user workplaces, and devices, rendered via our client-side technologies or those of others. Even technologies that have yet to be invented. Enterprise SOA was just the beginning of enabling such decoupling, for process-flexibility. There will be much more. And yet all of this functionality will be under the same lifecycle frame, the backbone that will support the constant evolution, and constant optimization of our landscape at our customers. Our products will therefore reflect these principles. We will continually carve new lanes, and deliver new functionality, even deep new technologies. The applications will evolve continuously, and piecewise, as nature does: bringing new things, renovating others, adding here and retiring there, and doing so without breaking its essential qualities: reliability, integrity, integration, seamless administration, change and lifecycle management.

Just as every few years we humans shed most of our cells, acquire new memories and lessons, decisions and beliefs, evolve and yet stay essentially who we are, I believe it is possible for software to renovate itself completely, and yet continuously.

So as excited as I am looking ahead to innovations on the horizon and beyond, that there is tons of new technologies, new capabilities, and new functionality to be delivered in our software, it is perhaps most reassuring that none of these will break the essential promises at the heart of timelessness, of reliability, integrity, coherence and continuous evolution.

On that reassuring thought, it is time to press the bed button on my seat and try out the fancy new lie-flat bed to end a day that began already 3 timezones away, 20 hours ago. And as I browse thru the 80 movies onboard, and notice the flight monitor displaying the plane's airspeed of 567 miles/hr, things that passengers 40 years ago couldn't have imagined, I find myself thankful for being in the comfort of a well engineered timeless system.

Copyright