CLIPCODE™

Thoughts On Software Design

By Eamon O'Tuathail

www.clipcode.net

Before we look at how we should go about building software and other knowledge-intensive products, we should first briefly explore the ideal architectural goals for delivering useful products. Simply put, what are we really trying to achieve? When we finish creating a product - how should third parties decide whether it is mediocre, pretty good or absolutely amazing?

In other areas of life, picking what is "best" is relatively easy – the best car (Pagani, of course), the best red wine (St. Emillion), the best place to go on holidays (Naples in Florida) or the best painter (Vermeer, from the lovely Dutch town of Delft) all stand out as obvious selections. But for software, how do we pick "the best"? Since software is so varied, a more practical approach might be to consider which characteristics are we after in selecting the best software? Let's see what we appreciate ...

In one (marketecturishly long) sentence, software should be simple, low-cost, secure, standards-based, many-core-aware, resource-identified, cloud-suitable, user-focused, event-driven, agent-delegating, attention-valuing, semantically-arranged, highly-available, model organized, pattern-following, team-oriented, community-building, multi-everything, ecosystem-establishing, reseller-/integrator-/ISV-friendly, product-line-delineated, feature-delivering, design-led, loosely coupled, globalized, accessible, policy-driven, iteration enhancing, directory-using, graph-publishing, multi-factor authenticating, transactional, personalizable, highly-innovative, composable, auto-updating with release channels, data-structuring, experience-sharing, object-exposing, shell-discoverable and administrable.

Let's briefly explore each of these & see how they bring real benefits to customers. We'll do this from the viewpoint of software platforms, but note much of this is highly relevant to all knowledge-intensive products.

Simple

Software is too complex. Too many software engineers with too much time on their hands have devised ever more complicated platforms and the bewildered end-users and IT professionals are having a difficult time keeping up. Indeed, this is one of the key reasons they don't, and instead stick with 5-, 7- and even 10-year old software platform versions that they are familiar with (but yet lack modern feature sets). Radically simpler platform design is a key goal.

Low-cost

Software is too expensive. Especially the software that you want. Typically vendors strip anyway any significant value from their low-end offerings, in the hope of enticing you to upgrade or hiding the true cost until it is too late and you and your team have committed many months of work to the platform. Often there is a requirement for other products, in addition to the one you really want. So one needs to consider the entire stack that is required, and all costs surrounding that stack. The entire software platform should be offered at a significantly lower cost.

Secure

A software platform must be secure – in areas such as mandatory two-factor authentication, always-on encryption, access control, provisioning and role-based security. Security must be embedded in the platform architecture from the start and not some later optional add-on.

Standards-based

Open-protocols, open schemas and open APIs – the goal is to be based on open standards as much as possible. Every API we use, every network protocol we communicate over and every data schema we use to store knowledge should be based on open standards.

Many-core-aware

The era of many cored computing has arrived. Moving beyond two core and four core processors, we are likely to see far more cores per processor (CPU, GPU and hybrid) and software platforms will need to be designed to properly exploit the massive additional availability of compute power. Asynchronous programming and parallel programming (not the same) need to become commonplace.

Resource-identified

Things that interest us – documents, concepts, pages, business contracts, purchase orders, etc. are all known as resources. Resources are identifiable, securable, versionable and locatable.

Semantically-arranged

The semantic web is the next great step for software platforms. By exposing resources as semantic concepts opens the way for very interesting features.

Cloud-suitable

The cloud provides off-premises compute/storage/networking/services and software must be designed to exploit the specific features of cloud computing with ease.

Questions about “how” / “when” / “where” / “by whom” related to subsystems within a platform in the cloud around the globe need to be part of early decision making.

User-focused

The user should be at the heart of the design process.

It would be great to have a real end user on the development team, but at least regularly availability to them is needed so that prompt/accurate user feedback is available. The development team needs to have close contact with actual packagings of the platform. How user interact with the platform needs to be given serious consideration. When users have problems and contact the helpdesk, this critically important stream of data about where real world problems exist with the platform should flow back to the development team.

Design-led

A pleasant visual experience is required. Let's face it–most developers are not the most talented in graphic design. So hire in an experienced design professional for this.

Event-driven

Events happen all the time and software platforms should be designed to identify, distribute and react to events, in other words, to be event-driven.

Rather than using a timer to check if something happened, or waiting for a client system to send a request to a server, event producers should be able to immediately have interested parties informed of what happened and then they can react if needed, independently of the event producer.

Agent-delegating

Allow automated semantic agents (pieces of software) to work on behave of users and hosts in order to automatically perform tasks based on events that occur.

Attention-valuing

There needs to be an appreciation that the attention span of humans is highly valuable, and interruptions are therefore a major cost factor. It is clear knowledge professionals while doing important work are being excessively interrupted during their work day by issues that can either be delayed without causing a problem, automatically responded to, or automatically delegated to another knowledge professional.

Loosely-coupled

Constant change is a characteristic of modern life, and of modern software platforms. By creating platforms that are loosely coupled, changes in one area can be isolated from having ripple effects in other areas.

Highly-available

If users are to entrust their knowledge networks to a platform, it must be resilient to issues and highly available.

Model-structured

Potential models include Directory Model, Domain Model, Feature Model, Behavior Model, Learning Model, and many more - all of which can be expressed and integrated in a Semantic Model. For software products, the code-related models all have a common fabric that weaves itself through all the models – and that is the source code.

The models can be considered as viewports into the codebase, from different angles. What each shows is specific to what the model needs to represent, but importantly the viewports do not conflict with each other.

Pattern-following

Design patterns show tried and trusted ways of solving problems. A platform should use these where appropriate and create new ones where necessary.

Team-oriented

People work in teams. To achieve any result of significance, a highly collaborative team needs to work together, and the software platform needs to assist.

Community-building

The marketing folks use terms such as "network effects" and "adoption" to indicate the value of communities for software platforms. The long-term viability (or lack thereof) of many platform hinges on building an active community.

Multi-everything

There are many ways of performing tasks. The platform should be multi-protocol, Multi-view, Multi-channel, Multi-Tenant, Multi-Instance and Multi-paradigm (mobile app, web, desktop (they have not gone away), speech, TV, wearables, car).

Ecosystem-establishing

Partners power modern software platforms. A platform needs to offer partnership opportunities. ISVs, system integrators, hosting providers, consultants and trainers all are part of an ecosystem surrounding successful platform.

Reseller-/Integrator-/ISV- friendly

Resellers, system integrators and ISVs play a key part in the sales process for software platforms. This needs to be an important consideration in designing a platform's competitive offerings.

Product-line-delineated

Where multiple products are part of a family, a product line strategy can help

Internationalized

People from around the world should be able to use the platform.

Policy-driven

Any decisions should be exposes to power-users and IT professionals as policy. This surfaces as configuration, templates, workflow, rules & lifecycle.

Directory-using

Identity knowledge should not be scattered across the network. Nether should it is stored in app-specific silos. Instead should be managed in a controlled environment, known as a directory with carefully managed access permissions.

Graph-publishing

If identity knowledge needs to be taken out of app-specific silos, then why not all knowledge. Imagine for an enterprise if all its knowledge were published in a distributed stored knowledge graph, and all apps and all knowledge professionals could access all knowledge relevant to their requirements, within security restrictions.

Accessible

People with disabilities should be able to easily use the system. Often features added for this have more wide-spread usage (e.g. the API for accessibility is widely used for automated UI testing).

Transactional

Each transaction is identified as being carried out by a named user, when and where it occurred is recorded, and is atomic.

Personalizable

People are different! They want the platform to work the way they want.

Multi-factor authenticating

Security experts have long realized there are three ways of authentication – what you know (password or passphrase), what you are (biometrics) and what you have (smart card or your mobile phone). The best foundation for distributed authentication is a combination of two or more of these.

Highly-innovative

Software platforms have not changed much over the past few years. A new software platform should offer significant innovation. If it is just a “me-to” product, why bother?

Composable

There is a need to build up solutions from composable parts, possibly with different combinations for different packagings. Composability needs to be added from the start. It makes sense to have a single composability story, so that internal parts and third-party parts are composed using the same architecture.

Feature-delivering

A platform needs to deliver features which bring tangible benefits to customers, users and other stakeholders. Thinking in terms of features allows us to look at a very important part of platform architecture – what to leave out. If any piece of functionality does not impact of features that are of benefits to stakeholders, then serious consideration should be given to not implementing that functionality.

Development teams (even very large ones) cannot create all the functionality they want, so choices have to be made. The obvious basis for choice is the impact functionality will have on key features of the platform.

Iteration-enhancing

A big system that runs well inevitably starts from a small system that runs well, and it has been shown that an iterative agile approach to software development is the optimal use of engineering resources.

We need to decide on features that are useful overall, grading then according to benefit that they deliver to customers, but also grouping them into buckets that make sense to be created together, and then deciding on an iteration plan. Each iteration needs to deliver enhancements that are tangible to customers.

Everygreen (auto-updating with release channels)

Software needs to be regularly updated and it is desirable that updates happen automatically, so that, without human intervention, the deployed software is always up-to-date. What "up-to-date" means varies depending on the intended use of the software. There is always a tradeoff in software between maturity of the software and the latest features. A business team working on a production environment will require a stable release, whereas an integration team planning to launch a new system may wish to use a beta release, and software developers may wish to get early access to a developer release to help them build against what is coming in future, whereas the really early adopters may wish to have a nightly "canary" build. Consider these differing releases to be different release channels, that allow users to specify the level of maturity they require in the software that is to be auto-deployed to their systems.

A nice succinct way of putting all this is to say software installations should be "everygreen".

Data-structuring

Some folks talk about structured data (they mean SQL), semi-structured data (they mean XML) and un-structured data (they mean blobs such as audiovisual files), which this is rediculous terminology, since all data in some way is structured. There are patterns in all forms of data and such patterns form structures. Pieces of data may well be structured differently, but there is some form to each piece and there are numerous benefits to externalizing the structure so that a range of applications (not just the one that created it) can interact with the data. A text file containing a snippet of programming code has structure – it can be lexically scanned and parsed to build an abstract syntax tree. A binary file containing a bitmap has structure – it typically has numerous blocks, with a metadata block (containing descriptive information about the document), it usually has a header block containing offsets to data blocks, it will have a color table block and so on.

Experience-sharing (browser, app, server, cloud, Internet of things)

Software developers and IT professionals are swamped with an ever-increasing flow of different technologies. They have difficulty keeping up, spend a large amount of time learning and often work on important projects with a sub-optimal level of experience of the technologies involved. This has become a major problem for technology providers and users. If would be of benefit if technologies used in one scenario can be used elsewhere (browser, app, server, cloud).

Object-exposing

Modern software applications consist of large Framework Models and it can be quite advantageous to expose a certain amount of this to third parties. Even if the main use of the application is expected to be via the provided user interface, some (often important) customers may wish to write their own apps and integrate with your app via your Framework Model.

An Framework Model may be exposed for direct programmatic access and/or as a messaging API (it is often a good idea to provide both).

Shell-discoverable

The command shell or command line interface is extremely useful. For interactive command entry, and scripts and in-app macros, a scripting shell is a valuable tool. The shell is faster in the hands of an experienced user compared to navigating possibly multiple levels of menus or ribbons, though it does mean the user needs to know the shell syntax and the command name. Application developers need to consider how commands from their applications can be discoverable from the shell.

Administrable

When writing software, we need to strive to ensure it can be administered. For any non-trivial system that is targetting a large user base, there will be characteristics that a company's system administrator will wish to configure for multiple groups of users. Rather than perform the configuration manually at each user's device, it is obviously better to use some central administrative console to perform a single action or a small number of actions.

Software developers need to think of the administrator as a very important stakeholder whose needs should be considered. For many types of software applications, enterprises are a large market and with a little extra thought on how the app can be administered at scale, significant additional sales can be made.