Top 10 Concepts That Every Software Engineer Should Know

Alex Iskold Jul 23, 2008

The future of software development is about good craftsmen. With infrastructure like Amazon Web Services and an abundance of basic libraries, it no longer takes a village to build a good piece of software.

These days, a couple of engineers who know what they are doing can deliver complete systems. In this post, we discuss the top 10 concepts software engineers should know to achieve that.

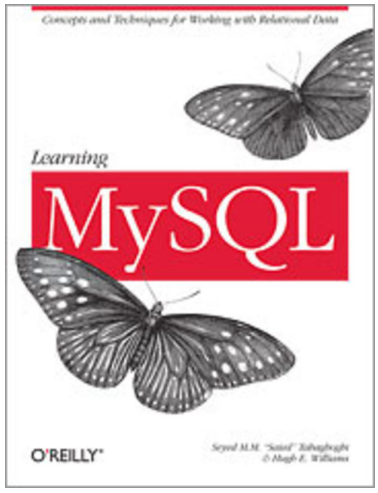
A successful software engineer knows and uses design patterns, actively refactors code, writes unit tests and religiously seeks simplicity. Beyond the basic methods, there are concepts that good software engineers know about. These transcend programming languages and projects - they are not design patterns, but rather broad areas that you need to be familiar with. The top 10 concepts are:

1. Interfaces
2. Conventions and Templates
3. Layering
4. Algorithmic Complexity
5. Hashing
6. Caching
7. Concurrency
8. Cloud Computing
9. Security
10. Relational Databases

10. Relational Databases

Relational Databases have recently been getting a bad name because they cannot scale well to support massive web services. Yet this was one of the most fundamental achievements in computing that has carried us for two decades and will remain for a long time. Relational databases are excellent for order management systems, corporate databases and P&L data.

At the core of the relational database is the concept of representing information in records. Each record is added to a table, which defines the type of information. The database offers a way to search the records using a query language, nowadays SQL. The database offers a way to correlate information from multiple tables.



The technique of data normalization is about correct ways of partitioning the data among tables to minimize data redundancy and maximize the speed of retrieval.

9. Security

With the rise of hacking and data sensitivity, the security is paramount. Security is a broad topic that includes authentication, authorization, and information transmission.

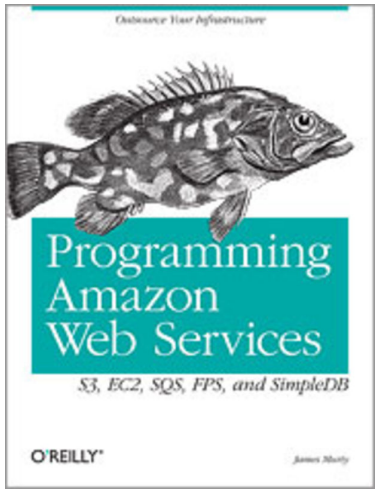
Authentication is about verifying user identity. A typical website prompts for a password. The authentication typically happens over SSL (secure socket layer), a way to transmit encrypted information over HTTP. Authorization is about permissions and is important in corporate systems, particularly those that define workflows. The recently developed OAuth protocol helps web services to enable users to open access to their private information. This is how Flickr permits access to individual photos or data sets.

Another security area is network protection. This concerns operating systems, configuration and monitoring to thwart hackers. Not only network is vulnerable, any piece of software is. Firefox browser, marketed as the most secure, has to patch the code continuously. To write secure code for your system requires understanding specifics and potential problems.

8. Cloud Computing

In our recent post Reaching For The Sky Through Compute Clouds we talked about how commodity cloud computing is changing the way we deliver large-scale web applications. Massively parallel, cheap cloud computing reduces both costs and time to market.

Cloud computing grew out of parallel computing, a concept that many problems can be solved faster by running the computations in parallel.



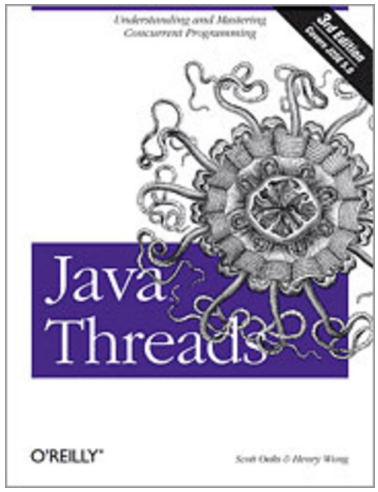
After parallel algorithms came grid computing, which ran parallel computations on idle desktops. One of the first examples was SETI@home project out of Berkley, which used spare CPU cycles to crunch data coming from space. Grid computing is widely adopted by financial companies, which run massive risk calculations. The concept of under-utilized resources, together with the rise of J2EE platform, gave rise to the precursor of cloud computing: application server virtualization. The idea was to run applications on demand and change what is available depending on the time of day and user activity.

Today's most vivid example of cloud computing is Amazon Web Services, a package available via API. Amazon's offering includes a cloud service (EC2), a database for storing and serving large media files (S3), an indexing service (SimpleDB), and the Queue service (SQS). These first blocks already empower an unprecedented way of doing large-scale computing, and

surely the best is yet to come.

7. Concurrency

Concurrency is one topic engineers notoriously get wrong, and understandibly so, because the brain does juggle many things at a time and in schools linear thinking is emphasized. Yet concurrency is important in any modern system.



Concurrency is about parallelism, but inside the application. Most modern languages have an in-built concept of concurrency; in Java, it's implemented using Threads.

A classic concurrency example is the producer/consumer, where the producer generates data or tasks, and places it for worker threads to consume and execute. The complexity in concurrency programming stems from the fact Threads often needs to operate on the common data. Each Thread has its own sequence of execution, but accesses common data. One of the most sophisticated concurrency libraries has been developed by Doug Lea and is now part of core Java.

6. Caching

No modern web system runs without a cache, which is an in-memory store that holds a subset of information typically stored in the database. The need for cache comes from the fact that generating results based on the database is costly. For example, if you have a website that lists books that were popular last week, you'd want to compute this information once and place it into cache. User requests fetch data from the cache instead of hitting the database and regenerating the same information.

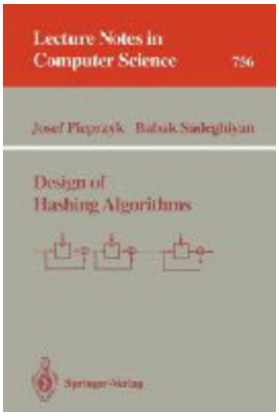
Caching comes with a cost. Only some subsets of information can be stored in memory. The most common data pruning strategy is to evict items that are least recently used (LRU). The prunning needs to be efficient, not to slow down the application.

A lot of modern web applications, including Facebook, rely on a distributed caching system called Memcached, developed by Brad Firzpatrick when working on LiveJournal. The idea was to create a caching system that utilises spare memory capacity on the network. Today, there are Memcached libraries for many popular languages, including Java and PHP.

5. Hashing

The idea behind hashing is fast access to data. If the data is stored sequentially, the time to find the item is proportional to the size of the list. For each element, a hash function calculates a number, which is used as an index into the table. Given a good hash function that uniformly spreads data along the table, the look-up time is constant. Perfecting hashing is difficult and to deal with that hashtable implementations support collision resolution.

Beyond the basic storage of data, hashes are also important in distributed systems. The so-called uniform hash is used to evenly allocate data among computers in a cloud database. A flavor of this technique is part of Google's indexing service; each URL is hashed to particular computer. Memcached similarly uses a hash function.



Hash functions can be complex and sophisticated, but modern libraries have good defaults. The important thing is how hashes work and how to tune them for maximum performance benefit.

4. Algorithmic Complexity

There are just a handful of things engineers must know about algorithmic complexity. First is big O notation. If something takes O(n) it's linear in the size of data. O(n^2) is quadratic. Using this notation, you should know that search through a list is O(n) and binary search (through a sorted list) is log(n). And sorting of n items would take n*log(n) time.

Your code should (almost) never have multiple nested loops (a loop inside a loop inside a loop). Most of the code written today should use Hashtables, simple lists and singly nested loops.

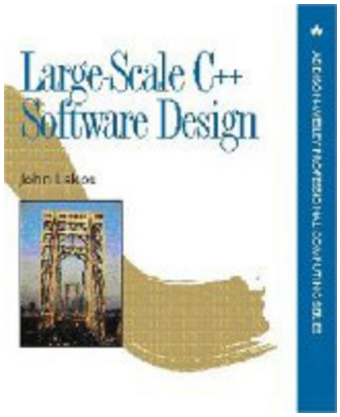
Due to abundance of excellent libraries, we are not as focused on efficiency these days. That's fine, as tuning can happen later on, after you get the design right.

Elegant algorithms and performance is something you shouldn't ignore. Writing compact and readable code helps ensure your algorithms are clean and simple.

3. Layering

Layering is probably the simplest way to discuss software architecture. It first got serious attention when John Lakos published his book about Large-scale C++ systems. Lakos argued that software consists of layers. The book introduced the concept of layering. The method is this. For each software component, count the number of other components it relies on. That is the metric of how complex the component is.

Lakos contended a good software follows the shape of a pyramid; i.e., there's a progressive increase in the cummulative complexity of each component, but not in the immediate complexity. Put differently, a good software system consists of small, reusable building blocks, each carrying its own responsibility. In a good system, no cyclic dependencies between components are present and the whole system is a stack of layers of functionality, forming a pyramid.

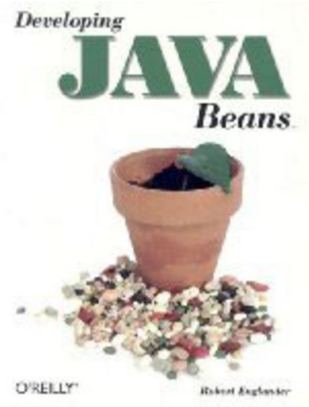


Lakos's work was a precursor to many developments in software engineering, most notably Refactoring. The idea behind refactoring is continuously sculpting the software to ensure it'is structurally sound and flexible. Another major contribution was by Dr Robert Martin from Object Mentor, who wrote about dependecies and acyclic architectures

Among tools that help engineers deal with system architecture are Structure 101 developed by Headway software, and SA4J developed by my former company, Information Laboratory, and now available from IBM.

2. Conventions and Templates

Naming conventions and basic templates are the most overlooked software patterns, yet probably the most powerful.



Naming conventions enable software automation. For example, Java Beans framework is based on a simple naming convention for getters and setters. And canonical URLs in del.icio.us: http://del.icio.us/tag/software take the user to the page that has all items tagged *software*.

Many social software utilise naming conventions in a similar way. For example, if your user name is *johnsmith* then likely your avatar is *johnsmith.jpg* and your rss feed is *johnsmith.xml*.

Naming conventions are also used in testing, for example JUnit automatically recognizes all the methods in the class that start with prefix *test*.

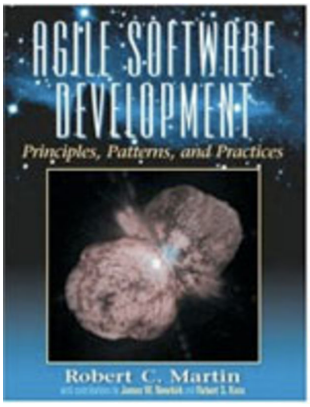
The templates are not C++ or Java language constructs. We're talking about template files that contain variables and then allow binding of objects, resolution, and rendering the result for the client.

Cold Fusion was one of the first to popularize templates for web applications. Java followed with JSPs, and recently Apache developed handy general purpose templating for Java called Velocity. PHP can be used as its own templating engine because it supports *eval* function (be careful with security). For XML programming it is standard to use XSL language to do templates.

From generation of HTML pages to sending standardized support emails, templates are an essential helper in any modern software system.

1. Interfaces

The most important concept in software is interface. Any good software is a model of a real (or imaginary) system. Understanding how to model the problem in terms of correct and simple interfaces is crucial. Lots of systems suffer from the extremes: clumped, lengthy code with little abstractions, or an overly designed system with unnecessary complexity and unused code.



Among the many books, Agile Programming by Dr Robert Martin stands out because of focus on modeling correct interfaces.

In modeling, there are ways you can iterate towards the right solution. *Firstly*, never add methods that might be useful in the future. Be minimalist, get away with as little as possible. *Secondly*, don't be afraid to recognize today that what you did yesterday wasn't right. Be willing to change things. *Thirdly*, be patient and enjoy the process. Ultimately you will arrive at a system that feels right. Until then, keep iterating and don't settle.

Conclusion

Modern software engineering is sophisticated and powerful, with decades of experience, millions of lines of supporting code and unprecidented access to cloud computing. Today, just a couple of smart people can create software that previously required the efforts of dozens of people. But a good craftsman still needs to know what tools to use, when and why.

In this post we discussed concepts that are indispensible for software engineers. And now tell us please what you

would add to this list. Share with us what concepts you find indispensible in your daily software engineering journeys.

Image credit: cbtplanet.com