2.9.2

While the case could be made an architecture-centric style is more consistent with either the waterfall model or Boehme's spiral model of software development, those familiar with the backgrounds of each may be more likely to liken this style to the latter model. Such an observation is made based on the difference in the general timeline according to which design decisions are made. As in a traditional, waterfall-based approach, design decisions are made during a specific 'phase' and using a spiral approach, design decisions made be made concurrent with documentation of the design decisions, according to the first of the 6 tenets of Boehme's model's invariant.[1]

2.10.3

The architecture of the Google Search web app/program was developed and refined by two students, who contributed ideas while at Stanford Univ., as discussed in a paper google divulged to the general public ~1998 according to its Corporate 'history in depth' webpage. An overview of the research and architectural key components underlying or contributing to the search engine's development includes synopses of prototypes and the nature of search engines in the context of corporate and academic settings, dating from 1994 up until 1998. Early design prototypes listed, namely Yahoo,Lycos, and WWW-Worm to include a few, were created – as the authors describe – to handle a relatively exhaustible number of documents and search queries, as opposed to the scalable architecture decisions upon which google based its design. In the introductory paper, referenced herein[2], the authors describe requirements analyses that involved distinguishing among key benefits brought forth by utilizing a design that builds upon common parts but in unique ways, for building an architecture that makes their service fast to the end user, efficient in terms of memory, scalable as the web evolved, adaptable as improving algorithms would be applied and evaluated, and progressive in terms of processes that were improved through real-time and further complicating new factors. Concurrent processing was used in a database-oriented design that uses an ISAM indexer that can be reverse-indexed to build a context out from a search key or vice versa, based on hashing, database table merges, and serializeable structures for manipulating data stored in any number of ways as seen on the web. Aside from a novel pagerank technique, BigFiles for VFS implementation, and data structures including tuples, compressed data indexable across clusters, and an advanced indexing process, google was unprecedented in their discovery and application of user feedback and in ability to further increase the efficiency of their app alongside development and implementation of new ideas.

2.10.8

The architectural/design decisions and tradeoffs associating with identifying selection of using a parser generator such as GNU's BISON or building a parser manually, by-hand in an object-oriented language are somewhat complex and are likely dependent upon the nature of the requirements for which the parser should fulfill. As there are algorithmic differences in the

[1] Source: http://en.wikipedia.org/wiki/Spiral_model

[2] http://infolab.stanford.edu/~backrub/google.html

lexicographic analyzing capabilities of various parsing techniques and implementations w/r/t timing and efficiency, this could be of large concern to some users. However, while certain implementations may be more time-efficient, on some levels ie: speed, especially for large inputs to the parser, the efficiency gains in speed may be offset by the memory overhead losses that result from their usage. For example, using an automatic parser generator for purposes such as utilizing in a free-form language context or relatively architecturally-primitive usage-scenario such as the Reverse Polish Notation calculator might be entirely suitable. In this case, C/C++ or other languages offer direct library support to manipulate the object types required to carry out the tasks needed for parsing the input into suitable groupings based on defined semantics. However for more complex lexicographic analysis, it may be the case that other programming languages have more conscientiously devised functions for handling textual cues in parsing that are still built upon a low-level language like 'C,' which may offer both straightforward manually-programmable grammar-defining but also offer some degree of object-orientedness, simultaneously. While at a more extreme end, still, Java may offer both efficient parsing on Order-of-Magnitude based inputs (in large), but requires the additional overhead of a virtual-machine's necessary memory (->RAM ..ie: virtual memory?) to run. In turn, it would be more efficient to code a parser that can utilize its built-in reflection capabilities than to have to hand-code this type of capability using C code afresh.

2.10.9

Implementing a parser using a parser generator, described above, would require defining rules for the grammar(/syntax) of the input used to identify various constructs that signify bits of cohesion in the program that may be adjoined by various distinctive parts of the input supplied by the user. In addition, the sum of the individual parts described may form bigger parts, such as statements, which comprise even bigger parts, such as functions, and so on until the first piece of code (the start) is identified, and the parser can continually run through the entirety of the input supplied. The architecture of the generated parser, in the case of Bison, came essentially from a set of grammar-rules that a user provides to the parser generator (Bison) so it can produce a parser that analyzes the user-described programming language. Essentially, the parser-generator *interprets* the supplied programming-language rules and syntactical descriptions in order to generate a parser-implementation file which can interpret source code written in the described-languages code format. The generated parser implementation file is generated as a .C file that includes the yyparse function and any other (presumably, C-coded-) functions defined in the grammar, as available methods that can be called from within the implementation file, which it is supplied to Bison as an input-source parameter alongside the actual programming-language code being parsed by the generated-parser. The architecture for the parser-generator came from token-identification symbols (which may include identifiers such as terminal and non-terminal symbols) supplied by the user's grammar and are used by the lexical analyzer[3] when parser-generation is occurring.

[3] Source: http://www.gnu.org/software/bison/manual/html_node/Bison-Parser.html#Bison-Parser