Methodologies and Artifacts in Software Design

Ken Kennedy, Tacksoo Im, David Beck, D.E. Stevenson

18 May 2006

1 Introduction

This paper presents various paradigms of software design. Each of these paradigms consist of a methodology, method(s), and tools. The definitions of methodology and method differ even among disciplines in computer science. In this paper, we define methodology as a world-view. For example, the goal of a formal methodology is to formalize specifications such that a mathematical proof can be given, which states that the specifications are correct or incorrect with regards to the intentions of the designer. The methodology describes the world-view but does not detail how a tool could be built for the methodology.

Methods are built upon a methodology. The definition of method that we use comes from "Software Engineering: A Practitioner’s Approach": which defines a method (their term is methodology) as "a mechanism translating the physical problem to its design representation, a notation for representing functional components and their interfaces, heuristics for refinement and partitioning, and guidelines for quality assess-
ment”. For example, the formal methodology has the following methods: axiomatic, denotational, and operational semantics.

A tool is an implementation of a method. This can either be a software program, such as a theorem prover like Isabelle, or it can be a language, such as Z. While we define three different components of a paradigm, this is primarily for consistency in the paper. It may be possible to describe some paradigms best with only a methodology and tools while other paradigms might have additional components between methodology and method(s) and/or between method(s) and tools.

Traditionally, there are four major areas of concern in software design: data, architecture, interfaces, and components [18]. We identify the various paradigms that influence the ways in which each concern is designed. Each design paradigm can be separated into methodology, method and tools. We identified the design paradigms and argue that each can be separated into methodology, method and tools. But some tools are not unique to a methodology and cannot be categorized easily. We call these tools, “design artifacts” and they can be anything is used to represent design. A comprehensive list of design artifacts is listed and its historic origins are identified.

2 Timeline

diagram of the timeline

3 Design Methodologies

different methodologies
3.1 Functional Programming

3.1.1 Functional Programming Methodology

Functional programming represents a different approach to programming. A functional programming language will have as its foundation lambda calculus rather than a Turing machine. We know from Turing that the two are equivalent, so both are capable of computing the same problems, but a language built upon one will differ from a language built upon another. Lambda calculus was originally developed by Alonzo Church to provide a foundation for the field of mathematics [4]. It was shown that lambda calculus could be used for computable functions; however, determining the equivalence of two lambda calculus expressions was found to be an undecidable problem.

3.1.2 Functional Programming Method

The functional programming languages introduced many new important ideas. A few of these ideas include lexical closures, garbage collection, trees, dynamic typing, and macros. Many of these have been incorporated into other languages and have changed how computer programs are developed.

3.1.3 Functional Programming Tools

The first language for functional programming was LISP (LISt Processing). LISP was created by John McCarthy as a reaction against FORTRAN [15]. The first version of the language was developed for an IBM 704 by Steve Russell. LISP was quickly picked up by those in the field of artificial intelligence, and it is the second oldest programming language that is still currently being used. Of course, the LISP of today is very different
LISP implemented several revolutionary features. One of these was garbage collection. However, on the computers of the day, LISP was not an efficient language due to features like garbage collection and it being interpreted. This eventually resulted in a new type of computer known as a LISP machine. A LISP machine used LISP as its primary language.

Over the years, there were a large number of forks from the original and subsequent LISP versions. One of the most popular is known as Scheme, created in the 1970s by Guy L. Steele and Gerald Jay Sussman. This resulted in programs written for one version of LISP, such as Scheme, not always working properly for another version, such as Emacs LISP. The problem was even made worse by programs not working on different implementations of the same version. To alleviate this problem, the ANSI Common LISP standard was adopted in 1994. The standard tried to take the best parts of all of the major versions of LISP and combine them into one language.

3.2 Formalization

3.2.1 Formal Methodology

The purpose of a formal methodology is to provide a mathematical basis for a specification such that aspects about the specification can be proven to determine if the specification is what was intended by the designer. In this respect, formal methodologies have been applied to both the construction of hardware and software systems.
3.2.2 Formal Methods

Formal methods are often separated into three categories: axiomatic, denotational, and operational. An overview of each of these methods is given below.

Formal methods started with axiomatic semantics which originated from papers by Robert Floyd and C. A. R. Hoare which each provided a different approach to formalizing a software program. The more common is to use pre and postconditions to specify what the state of the program should be before a command is run and to specify what the state of the program should be after the command has been run.

Denotational semantics were created by Dana Scott and Christopher Strachey who based the semantics upon domains in which domains had a set of properties. Since then, denotational semantics have been extended using power domains to make the semantics even more general and able to handle concurrency.

Operational semantics are used to describe the behaviour of the computational steps of a program by comparing a computation to a simpler one.

3.2.3 Formal Methods Tools

A large number of formal methods tools and languages have been developed. Two of the most popular tools are automated theorem provers and model checkers. These tools are used to verify the specification of hardware and software systems. Using these tools, a system can be proven correct, provided that there are not a large number of states in the system.

Many formal specification languages have been developed. Perhaps the most popular of these languages is Z. Z has been used in the creation of several high-profile
systems where security and stability was required.

3.3 Logic Programming

3.3.1 Logic Programming Methodology

The idea of logic programming originated from John McCarthy. The purpose of the logic programming methodology is to base computer programming languages on mathematical logic. By basing a language on mathematical logic, a user can then get either a true or false answer from a problem.

3.3.2 Logic Programming Method

There are several logics that logic programming systems are based upon. The logic used most commonly is first-order predicate calculus which is an extension to propositional calculus and includes existential and universal quantifiers. In general, first-order predicate calculus is undecidable.

3.3.3 Logic Programming Tools

Prolog (programmation en logique) is the best known logic programming tool. Alan Colmerauer and Robert Kowalski created Prolog for use in the area of natural language processing. It is a declarative language in which the user tells the language what to do, rather than how to do it.
3.4 Parallel Programming

3.4.1 Parallel Programming Methodology

The parallel methodology is the idea of splitting a program into small sections that can be run in multiple processors. Broadly, there is hardware level parallelism where parallelism is achieved through the use of hardware constructs and software level parallelism where language level constructs are used to delegate tasks to processors. Flynn’s taxonomy divides parallelism in terms of the number of streams of data and instruction. This classification can further be divided into data level parallelism and thread level parallelism. There exist many different types of programming models for parallel programming because there are many different types of parallel architectures [Jordan.03].

3.4.2 Parallel Programming Method

Designing parallel programs requires the matching of the problem to the architecture of a parallel system. Some problems such as ray-tracing can be easily parallelized but many problems require the designer to make decisions according to the architecture that the problem will run on.

3.4.3 Parallel Programming Tools

Tools Despite the difficulty programming parallel applications in imperative languages, various API’s(application program interface) such as MPI and OpenMP are used to program parallel applications in languages such as C and C++. Languages such as Linda and Occam provide an example of how ordinary languages can be extended to include parallel features. One such example is JavaSpaces, where Java was extended to
include features from Linda. There is also a theoretical parallel language called Unity [Chandy, Misra 88] that is used widely in distributed systems. A typical Unity program is composed of statements that run in parallel until no change can occur. This state of no change is called a fixed point and a correct Unity program always converges to a fixed point. Besides these languages there are many choices for parallel languages but because of portability and cost issues, many parallel languages are not used [Pfister]. The parallel computing environment requires a stable platform for its mostly scientific users and thus any tool that is not expected to last has not been adopted widely. [Simon]

3.5 Structured Programming

3.5.1 Structured Programming Methodology

It is hard to say that the structured design method has a specific unit of analysis like an object or a function. Instead, the structured design method tries to organize the program into manageable units suitable for maintenance and understanding. But it can be argued that the fundamental unit of analysis in structured design is a single line of computer ”instruction” and that structured design tries to organize programs into functions, objects or modules for a better design.

3.5.2 Structured Programming Method

Design Method The structured design methods were introduced in the mid-seventies by a variety of people involved in software engineering. Broadly, structured design can be separated into top-down and bottom-up strategies. Top-down strategy asks us to look at the big picture first and then elaborate the details whereas bottom-up strat-
egy asks us to elaborate the details before we consider the big picture. In other words, top-down is a decomposition strategy and bottom-up is a composition strategy. Both strategies need not be competing and can be used simultaneously in the design process. The top-down process will hide the details and help improve modularity and the bottom-up process will result in utility functions that will make the system more compact. Software designers recommend the use of both strategies during the iterations of the design process. This is referred to as stepwise refinement a technique that is widely used in structured design. Structured programming, another structured design methodology, is the splitting of a program into subsections that have a single entry and exit point. Structured programming can be considered to have its foundation in Djikstra’s "GOTOs Considered Harmful" paper in 1968. Structured programming emphasizes modularity and a top-down system design approach emphasizing high level specification and subsequent addition of detail. Programs are divided into logical elements that interact with one another via parameters and values, and it reduces the need for GOTO statements.

3.5.3 Structured Programming Methods Tools

Tools Languages such as Pascal and Ada were specifically designed to promote structured programming and many procedural languages such as C were used to do structured programming in the 70’s. Flow charts, Jackson systems diagrams, data flow diagrams, state diagrams are some of the artifacts that are used in structured design.
4 Design Artifacts

previous subsections

4.1 Flowcharts

Flowcharts are used to specify the steps of a process. They have been used for such a long time that their creation can not be attributed to any particular person. Flowcharts were extensively used in the 1950’s and 1960’s to model the behavior of computer programs.

4.2 Types

The theory of types [20] arises from Russell’s attempts to remove the paradox of the set of all sets that became associated with set theory in the early 1900s. Types add a hierarchy to set theory with the lowest level consisting of individuals, the next level consisting of sets of the individuals, and so forth.

4.3 Lambda Calculus

Lambda calculus was originally developed by Alonzo Church to provide a foundation for the field of mathematics [4]. It was shown that lambda calculus could be used for computable functions; however, determining the equivalence of two lambda calculus expressions was found to be an undecidable problem.
4.4 Turing Machines

The Turing machine is an abstract computing machine developed by Alan Turing in order to solve the Entscheidung problem; however, he discovered that Alonzo Church had already solved this problem with lambda calculus. During Turing’s time at Princeton with Church, Turing proved that his Turing machine was equivalent to lambda calculus.

4.5 Finite State Machines

The finite state machine (FSM) is similar to a Turing machine with the main difference being that an FSM has a finite number of states while a Turing machine has no such restriction. The FSM was developed by several people with the most significant contributions from Mealy and Moore.

4.6 FORTRAN

Between 1954 and 1957, John Backus of IBM lead the FORTRAN (FORmula TRANslator) team in developing the first high-level computer programming language. The syntax and semantics of the language were based off of the way algorithms were written in numerical methods texts.

4.7 LISP

LISP (LISt Processing) was created by John McCarthy and initially written for an IBM 704 by Steve Russell. LISP was a reaction against FORTRAN. McCarthy
believed that programming languages should be based off of lambda calculus rather than Turing machines.

### 4.8 COBOL

The Department of Defense develops COBOL (COmmon Business Oriented Language) as a language for business applications. COBOL was based off of Grace Hopper’s FLOW-MATIC and Bob Bemer’s COMTRAN.

### 4.9 SNOBOL

David J. Farber, Ralph E. Griswold, and Ivan P. Polonsky of Bell Labs develop SNOBOL (String-Oriented Symbolic Language). SNOBOL used a virtual machine to handle different platforms.

### 4.10 Petri Nets

Petri nets have been used extensively in modeling and design. They were developed by C. A. Petri in his dissertation [17]. Several variations of Petri nets exist with the most popular version being Place/Transition nets. Place/Transition nets became Turing-complete through the use of Hack’s inhibitor arcs [12]. Thus a Petri net is suitable for modeling any system that is computable.

### 4.11 Coloured Petri Nets

Coloured Petri nets (CPN) were developed by Kurt Jensen [13] as an extension to Petri nets. The main difference between a CPN and a Petri net is that the tokens in a CPN can
be distinguished from one another. Since a Petri net is Turing-complete, the advantage of a coloured Petri net is not that it is more powerful than a Petri net, but rather that it is more efficient in the number of places and transitions required for a model.

### 4.12 Semantic Networks

A semantic network is a design methodology derived from psychology [23] and brought over to computer science via research in artificial intelligence [19]. The structure of the network is supposed to mimic how an individual stores knowledge. Semantic networks are directed acyclic graphs (DAGs) that consist of nodes, links, and labels with the label of a link describing the relation of one node to another.

### 4.13 Frames

Frames, created by Marvin Minsky [16], are an extension to semantic networks. While both frames and semantic networks can represent the same information, frames impose more of a structure on the network thus speeding up the process of creation and searching. The frames system consists of frames, slots, and slot values with arcs from slots to frames. Unlike semantic networks, the arcs do not contain labels. The relation the arc defines is dependent upon that which the arc links together.

### 4.14 Formalization

When mathematical rigor is applied to software analysis and design, it is called formal methods. One of the more popular methods of applying formal methods is to apply it during the specification step. Predicate calculus, set theory and logic notation are
used to make precise statements about the specification of a software system. Any inconsistencies in the system will be discovered by analyzing its mathematical specifications. What makes this possible is the use of the concepts called data invariant, state, operation, precondition and postcondition in the specification. These concepts are represented using mathematical notations and the problems of contradictions, ambiguities, vagueness and incompleteness can be identified and remedied using mathematical analysis [18]. The detractors of formal method say that using mathematical rigor to specify a system is too difficult and expensive but the advocates counter that formal methods has its place in safety-critical systems where the consequences of software failure is very costly. But applying mathematical rigor in software verification has been less successful. In [7], De Millo, Lipton, and Perlis argue that program proofs is not practical and likely to remain unpractical in the future.

4.15 Relational Modeling

Relational Modeling was created by Ted Codd in 1970 to describe data in terms of mathematical relations [5]. Relational modeling gives a logical description about data and it allows the optimal organization of data through normalization. Normalization is done to get rid of data redundancy and update anomalies found in database systems. E-R Modeling usually precedes relational modeling in database design [6].

4.16 Entity-Relational Modeling

In simple terms, Entity-Relation (ER) modeling is used to model what we refer to as “data”. This type of modeling is called data modeling and it is used in software anal-
ysis, database analysis and anywhere that requires data to be modeled. The premise of this technique is the view that the world is composed of data elements called entities and the relationships between these entities. One of the main advantages of ER modeling is its ease of understanding. Many designers find that communicating with ER diagrams presents little difficulty. Peter Chen, the inventor of ER modeling [2], explains this by citing the direct correspondence between ER diagrams and languages such as Chinese and English. Chen also suggests that the ER-modeling of concepts is perhaps how humans in general model the world [3]. ER-modeling is widespread in database analysis because it maps well with the relational model (arranging data in tables) and it is the basis of object-oriented analysis where the relationship is hierarchical instead of a more general relationship defined in ER-modeling. ER-modeling is now indispensable in acquiring the data model that directly leads to the requirements of a software system. A good design can be said to begin with an accurate data model described by ER diagrams.

4.17 Abstract Data Types

Abstract Data types were first introduced in the 1975 as a way defining a data type by its operations instead of its implementation [11]. For example, a stack is defined by its first-in-last-out operation not its implementation. The use of abstract data types separates the implementation of a data type from its actual use thereby freeing the users of a data type from worrying with its implementation. It also provides a mechanism for communicating about data more effectively. In object-oriented languages, each class is an abstract data type.
4.18 Parallel Computing

Parallel computing is the idea of splitting a program into small sections that can be run in multiple processors. Broadly, there is hardware level parallelism where parallelism is achieved through the use of hardware constructs and software level parallelism where language level constructs are used to delegate tasks to processors. There exist many different types of programming models for parallel programming because there are many different types of parallel architectures [14]. Despite the difficulty programming parallel applications in imperative languages, various API’s(application program interface) such as MPI and OpenMP are used to program parallel applications in languages such as C and C++. Languages such as Linda and Occam provide an example of how ordinary languages can be extended to include parallel features. One such example is JavaSpaces, where Java was extended to include features from Linda. Designing parallel programs requires the matching of the problem to the architecture of a parallel system. Some problems such as ray-tracing can be easily parallelized but many problems require the designer to make decisions according to the architecture that the problem will run on.

4.19 Prolog

Prolog (programmation en logique) was created by Alain Colmerauer and Robert Kowalski in 1970. It was originally designed to solve problems in the field of computational linguistics but has found application in AI and database systems. In domains such as expert systems and natural language processing, problems can be stated in terms of knowledge and the inferences that can be derived from the knowledge [21]. This
paradigm is called logic programming and Prolog is the most widely used logic programming language. Prolog supports declarative specifications and unlike most programming languages that detail the instruction to execute it, Prolog does a theorem proof on the declared specifications that are expressed in first-order predicate calculus.

4.20 APL

Dr. Ken Iverson created APL (A Programming Language) in 1962 to introduce a set of notations that could be used to teach computer science and write concise programs. APL was inspired by mathematical notation and it is heavily used in statistics and other areas of mathematics where array processing occurs many times [8]. For this reason, APL is also known as Array Processing Language. APL's notation relies on non-ASCII characters such as Greek alphabets and it is processed from right to left. APL's mathematics inspired notation yields programs that are concise and sometimes hard to understand. But proponents of APL claim that APL is an elegant language where programs need not be obfuscated.

4.21 Ada

Ada was created by Jean Ichbiah and his team at CII Honeywell Bull in 1977. It was designed to solve the "language problem" that the department of defense had. Hundreds of languages were being used in the embedded systems of the US department of defense and there was a growing concern about the safety of each language. The department of defense set out to solve the problem by designing a common language that could replace all the existing languages. Ada meets the needs of the department of de-
fense by supporting exception handling, generics and other features that promote safer programming. Ada is widely used in military systems and aviation software where program safety is a top concern [22].

### 4.22 Smalltalk

Smalltalk was designed by Alan Keys in 1969. Smalltalk was inspired by Simula and Sketchpad. It is a pure object-oriented language that uses message passing for communication. There were many versions of Smalltalk but Smalltalk-80 has become the version that closely resembles modern day Smalltalk. Smalltalk is expressive and open and it is used in GUI development, prototyping and in telecommunication software. Smalltalk has also influenced other languages such as Ruby and Java [22].

### 4.23 Scheme

Scheme was designed by Guy Steele and Gerald Jay Sussman in 1975. It is a dialect of Lisp and has minimalism as a design philosophy. Scheme’s syntax is simply the most basic constructs that are needed to write a program, everything else is implemented as a library [1]. Scheme’s minimalist approach allows the programmer to choose the style of programming, whether that is object-oriented, imperative or functional. Scheme is used in various universities as a first language for learning computer science.

### 4.24 Snobol

Developed by a team including David J. Farber, Ralph E. Griswold and Ivan P. Polonsky in Bell Labs, Snobol stands for String-Oriented Symbolic Language [10]. It was
developed in 1962 and it become popular as a string manipulation language because of
it’s rich set of operations for string manipulation. Snobol is also a dynamically typed
language where any data type can be manipulated by other data types. Snobol is now
superceded by a variety of other languages that has string handling but it is a good
example of a special purpose language.

4.25 Basic

Basic (Beginner’s All-purpose Symbolic Instruction Code) was created in 1963 by John
G. Kemeny and Thomas E. Kurtz of Dartmouth College. It was designed for beginners
in computing and therefore features a simple syntax and it is easy to learn [9]. Basic
was designed as a language where the user did not have to know how the machine
worked to use the computer. This makes in easier for the user to learn the language and
use it but comes at a cost of speed. Currently, there are hundreds of implementations
of Basic and it has helped novices learn computer languages effectively.

4.26 CPL

CPL (Combined Programming Language) was developed by Cambridge and London
Universities in 1963; however, it was (as far as I can tell) not actually implemented.
The language was a combination of ALGOL-60 and functional ideas. It was designed
for both scientific programming and general programming like theorem-proving. It
was another attempt to do what PL/1 and Ada tried to do. It also failed.
4.27 BCPL

BCPL (Basic Combined Programming Language) was developed by Martin Richards in 1969. This language was merely a simplified version of CPL with only a single types. It was designed as a systems programming language, especially for writing compilers.

4.28 B

B was developed at Bell Labs around 1969, mostly by Ken Thompson. B had only one datatype, the word. Operators treated this type either as an integer or a memory address. B also made use of libraries.

4.29 C

The C programming language was developed by Bell Labs in the early 1970s. It borrowed ideas from B and from ALGOL-68. C is a low-level programming language that operates close to hardware, although it is portable. C offers the ability to make use of structured programming techniques while still maintaining the ability to access the hardware at the assembly level. C also was a reaction against the formal and simple structure of Pascal.

4.30 C++

C++ was developed in the early 1980s by Stroustrup to provide object-oriented functionality to C programmers. C++ includes powerful libraries and templates to allow
the reusability of code. C++ also allows operator and function overloading, multiple inheritance, and odd examples of polymorphism.

4.31 Structured Programming

Structured programming can be considered to have its foundation in Dijkstra’s “GO-TOs Considered Harmful” paper in 1968. Structured programming emphasizes modularity and a top-down system design approach emphasizing high level specification and subsequent addition of detail. Programs are divided into logical elements that interact with one another via parameters and values and reduces the need for GOTO statements. Stepwise refinement is a technique within the structured programming paradigm.

4.32 Stepwise Refinement

Niklaus Wirth introduced stepwise refinement in 1971. Stepwise refinement is a technique for top-down design in which a program is first specified in a high level, abstract form, and detail is added iteratively until the program is written in a programming language.

4.33 HIPO

Hierarchy plus input-process-output (HIPO) diagrams were first introduced in the 1970s (?). A HIPO diagram is a series of drawings that begins with a general overview and ends with detailed drawings of each function.
4.34 **Pascal**

Pascal was developed by Niklaus Wirth in order to force teachers to teach structured programming. It was also developed to ease compiler construction. The first versions were interpreted; later versions were compiled. Pascal has a keyword-oriented syntax, like ALGOL.

4.35 **Eiffel**

Eiffel was developed in 1985 by Bertrand Meyer. Meyer was seeking to develop a language that combined reliability and reusability with the ability to construct complex structures. Eiffel is fully object-oriented with a keyword-oriented syntax similar to ALGOL, strongly statically typed, and implements garbage collection. Eiffel supports inheritance and operator overloading, but not method overloading. Eiffel intentionally enforces simplified structure on the programmer.

4.36 **Java**

Java was developed in 1991 by James Gosling. The language initially descended from a language called Oak, designed for use in embedded systems. Java was a reaction against the complexity and misuse of C++, and attempted to compensate for C++’s lack of good facilities for security, distributed programming, and multi-threading.

4.37 **Literate Programming**

Literate programming was developed by Donald Knuth in 1979. It is a technique designed to increase the readability and thus the maintainability of code. In the original
definition, the documentation and the source code are written into the same file and markup is combined with program organization. There have been other definitions proposed since.

4.38 Object-Oriented Modeling

Object-oriented modeling made its first debut in computational science in the early 1960s, with Sketchpad and Simula. Simula is a programming language designed to allow object-oriented modeling techniques borrowed from physics to be applied to general business simulations. Sketchpad is a drawing application that organizes the information as objects and instances.

5 Conclusions

In this paper, we have presented various design paradigms and how it can be separated into methodology, method and tools. One of the principles that is apparent in each design paradigm is how a world view or perspective determines the strengths or the weaknesses of a design methodology. Therefore, certain design problems can be better solved by using design methodologies that are suited for those problems.

In the history of software design, we can see that the invention of a design artifact usually precedes the invention of a methodology. This is evidence that methodologies will be modified and enhanced when better design artifacts are developed. Therefore, both the methodology and artifacts must be considered carefully in order to produce quality software design.
References


