# The Art of Software Development

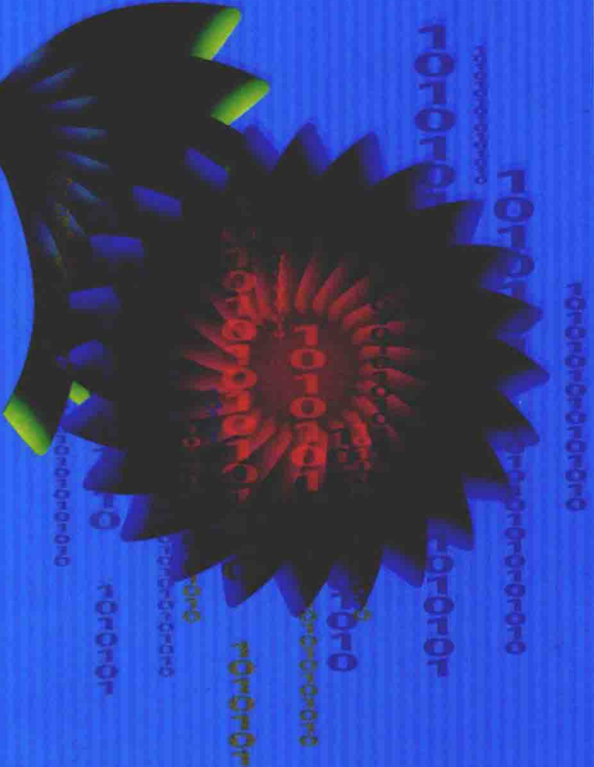
# 软件开发的艺术

Lizhi Xin

辛立志 著

University of Science and Technology of China Press

中国科学技术大学出版社

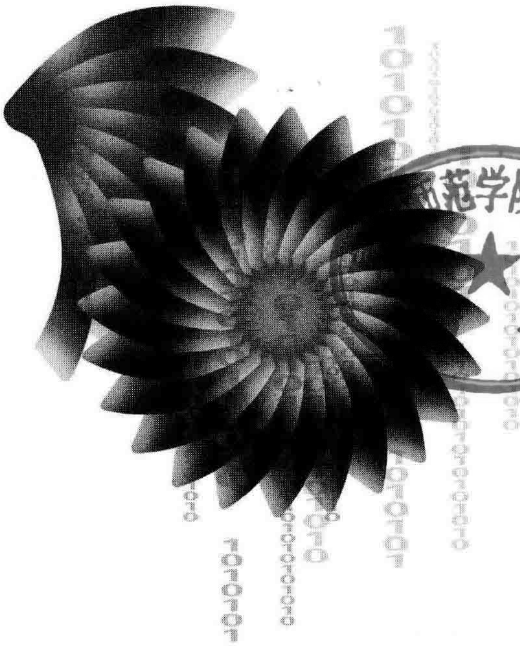


# The Art of Software Development

# 软件开发的艺术

Lizhi Xin

辛立志 著



University of Science and
Technology of China Press

中国科学技术大学出版社

## 内 容 简 介

本书论述了如何运用常见的设计原理来简化软件开发，以创建 CRM(客户关系管理)软件系统为范例，详细阐述了 metadata-driven(元数据驱动)过程的设计理念和步骤。本书共分为七章：第一章为导论，第二章为原理，第三章到第六章分别阐述了软件开发的四个阶段(即需求、分析、设计和构造阶段)，第七章为结论。全书图文并茂，叙述清晰，实例丰富，是一部来自于作者大量经验的总结性论著。

本书可作为学习软件开发的学生的参考书，特别适用于对软件开发技术有一定了解但希望进一步提高开发水平的应用开发人员。

**图书在版编目(CIP)数据**

软件开发的艺术＝The Art of Software Development：英文/辛立志著．—合肥：中国科学技术大学出版社，2016．8
ISBN 978-7-312-03961-4

Ⅰ．软…　Ⅱ．辛…　Ⅲ．软件开发—英文　Ⅳ．TP311．52

中国版本图书馆 CIP 数据核字(2016)第 123997 号

**出版**　中国科学技术大学出版社
安徽省合肥市金寨路 96 号，230026
http://press.ustc.edu.cn
**印刷**　安徽国文彩印有限公司
**发行**　中国科学技术大学出版社
**经销**　全国新华书店
**开本**　787 mm×1092 mm　1/16
**印张**　16.25
**字数**　501 千
**版次**　2016 年 8 月第 1 版
**印次**　2016 年 8 月第 1 次印刷
**定价**　49.00 元

**To Mom, Dad, and Sons**

# About the Author

```
<author name= "Lizhi Xin" chinesename= "辛立志">
  <job title= "Consultant">
    <myclients>
      <client name= "Bank of America" />
      <client name= "JP Morgan Chase Bank" />
      <client name= "CME Group Inc." />
      <client name= "Deloitte and Touche" />
      <client name= "Allstate Insurance" />
      <client name= "BlueCross BlueShield" />
      <client name= "US Department of Argiculture" />
      <client name= "John Lang LaSalle" />
    </myclients>
    <myproducts url= "http://www.softworkbench.com">
      <product name= "Softworkbench Studio" />
      <product name= "Stored Procedures Generator" />
    </myproducts>
  </job>
  <education>
    <graduateschool>
      <name> University of Wisconsin - Milwaukee</name>
      <chinesename> 威斯康星大学密尔沃基分校</chinesename>
    </graduateschool>
    <undergraduateschool>
      <name> University of Science and Technology of China</name>
      <chinesename> 中国科学技术大学 </chinesename>
    </undergraduateschool>
    <highschool>
      <name> High School Affiliated to Renmin University of China</name>
      <chinesename> 人大附中</chinesename>
    </highschool>
  </education>
</author>
```

# Preface

Software development is more art than science, more craft than engineering. Programming is easy because programming is writing code from well-defined specifications. Software development is hard because software development has to define the problem in the first place, the difficult part of software development is to figure out what to build, not how to build it. Software development is not just writing code, software development is about problem solving and decision making.

Programming languages come and go. Methodologies may change. Yet the principles of software development, and the art of applying it, have remained constant for a long period of time.

Ability to master the principles of software development: in my opinion, your ability to understand and apply basic principles to software developments is the ability that separates good developers from great developers.

This book shows how I applied the basic principles of software development to solving problems and making design decisions for a simplified CRM (Customer Relationship Management) application. Knowledge and experiences play a major role in developing software products. Knowledge can be acquired by reading books; however, experiences can only be gained by hands-on activities. I hope you can learn some useful knowledge from this book and apply the knowledge you learned to the projects you are working on and gain the experiences yourself.

In Chinese swordsman fiction, three levels of swordsmanship are used to describe the road to mastery of the art. The first level, swordsman, is good at using swords; the second level, swordsmaster, is good at using any object as swords, for example, a dead branch; the third level, swordsgod, is the level where the sword and the swordsman become one: no sword needed at all, swordsgod defeats their enemies using Tao force.

I found that these three levels can be applied easily to the art of software development as follows:

The first level, developers are good at one programming language and understand the fundamentals of software design. They never really learn the art of software development, they merely know how to program.

The second level, developers are good at not only different programming languages but also different development methodologies; also they know design patterns and application frameworks. At the second level, developers realize that programming language is just syntax, a tool to solve problems from the users.

The third level, developers realize that at the heart of software development is philosophies of thinking and learning; they begin to realize that developing software is just an extension of the way they think in real life. They will not care about particular programming languages or methodologies, and can switch between them easily. They know how to deal with complexities and changes using basic principles of software development.

Let's walk together in the software development journey from the swordsman to the swordsgod!

## Acknowledgements

I have been working on this book, in one form or another, for more than three years, and lots of people have helped and supported me in lots of ways.

I'd like to take this page to thank all those who made this book possible. First, I thank my Mom and Dad for the inspiration, love and support they give me. I'd never make it far without them. I also want to thank my two wonderful boys for the help when I am writing; actually a lot of ideas and thinking regarding this book happened at those cold ice rinks while my two boys have fun playing ice hockey and figure skating.

Special thanks are due to University of Science and Technology of China Press.

# Contents

# 1 Introduction

*If you know both yourself and your enemy, you can win numerous battles without jeopardy.*

— Sun Wu (544 BC)

## 1.1 History of Computing

Computing is intimately tied to the representation of numbers. Eventually, the concept of numbers became concrete, and advances in the numeral system and mathematical notation eventually led to the discovery of mathematic operations such as addition, subtraction, multiplication, division, squaring, square root, and so forth.

The earliest known tool for use in computation was abacus shown in Figure 1. 1. The Chinese invented a more sophisticated abacus from around the 2nd century BC known as Suan Pan. Usually, Suan Pan is about 20 cm tall and it comes in various widths depending on the application. It usually has more than seven rods. There are two beads on each rod in the upper deck and five beads on each rod in the bottom deck. This configuration is used for both decimal and hexadecimal computing. The beads are counted by moving

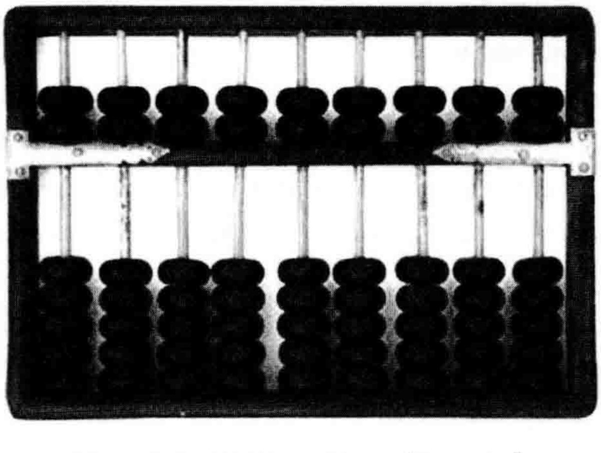


**Figure 1.1 A Chinese Abacus (Suan Pan)**

them up or down towards the beam. Suan Pan can be used for functions other than counting; there are very efficient Suan Pan techniques have been developed to do multiplication, division, addition, subtraction, square root operations at high speed.

The Antikythera mechanism is believed to be the earliest known mechanical analog computer. It was designed to predict astronomical positions and eclipses. It was discovered in 1901 in the Antikythera wreck off the Greek island of Antikythera, between Kythera and Crete, and has been dated back to *circa* 100 BC.

In 1837 Charles Babbage first described his Analytical Engine which is accepted as the first design for a modern computer. The analytical engine had expandable memory, an arithmetic unit, and logic processing capabilities able to interpret a programming language with loops and conditional branching. The Analytical Engine was never built because it was too complicated to be implemented by the technologies of the 19th century. It had to wait for the inventions of transistors and integrated circuits to build the analytical engine, a century later. Ada Lovelace who devised algorithms for Charles Babbage's Analytical Engine was an English mathematician. She is often regarded as the first computer programmer.

### 1.1.1 Information Theory

In 1703, Gottfried Wilhelm Leibnitz developed logic in a formal, mathematical sense with his writings on the binary numeral system. In his system, the ones and zeros also represent true and false or on and off states. But it took more than a century before George Boole published his Boolean algebra in 1854 with a complete system that allowed computational processes to be mathematically modeled.

In 1948, Claude Shannon published his classic paper "A Mathematical Theory of Communication" in the *Bell Systems Technical Journal*, a landmark that established the discipline of information theory.

Shannon emphasized that "The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point." Shannon also realized that the content of the message was irrelevant to its transmission and bits can be used as a new way of representing the most fundamental unit of information.

It is mainly because of Shannon that the binary standard has been adopted for sending and receiving information, earning him the title "The Father of the Digital Age".

Any real-world object is seen to be able to be reduced to bits. At the smallest unit in the computer, information is stored as bits and bytes. Let's look at how that works.

**Bit**

- A bit stores just a 0 or 1.
- Like an atom, the smallest unit of storage.
- Everything in a computer is 0's and 1's.

**Byte**

- One byte = grouping of 8 bits (e.g. 1101011).
- 8 bits can make 256 different patterns.
- One byte can store one letter (e.g. 'c').

### 1.1.2 Turing Machine

1936 was a key year for computer science. Alan Turing and Alonzo Church independently, and also together, introduced the formalization of an algorithm, with limits on what can be computed, and a "purely mechanical" model for computing.

These topics are covered by what is now called the Church-Turing thesis, a hypothesis about the nature of mechanical calculation devices, such as electronic computers. The thesis claims that any calculation that is possible can be performed by an algorithm running on a computer, provided that sufficient time and storage space are available.

In his classic paper "On Computable Numbers, with an Application to the Entscheidungs Problem" in 1937, Turing described that:

*All details sets of instructions that can be carried out by a human calculator can also be carried out by a suitably defined simple machine.*

It is known then as the Turing Machine.

Turing machines are not physical objects, but mathematical ones. They show if and how any given algorithm can be computed. Turing machines are state machines, where a state represents a position in a graph. State machines use various states, or graph positions, to determine the outcome of the algorithm.

The Turing machine mathematically models a machine that mechanically operates on a tape. There are symbols on this tape, which the machine can read and write, one at a time, using a tape head. Operation is fully determined by a finite set of elementary instructions such as "in state 42, if the symbol seen is 0, write 1; if the symbol seen is 1, change into state 17; in state 17, if the symbol seen is 0, write a 1 and change to state 6", etc.

More precisely, a Turing machine consists of:

1. A tape divided into cells, one next to the other. Each cell contains a symbol from some finite alphabets.

2. A head that can read and write symbols on the tape and move the tape left and right (and only one) cell at a time.
3. A finite-state control unit that stores the state of the Turing machine at a finite table of instructions (occasionally called an action table).

### 1.1.3 Von Neumann Machine

In 1946, a model for computer architecture was introduced and became known as von Neumann architecture. The Von Neumann model derives from a 1945 computer architecture description by the mathematician and physicist John von Neumann and others, First Draft of a Report on the EDVAC. This describes a design architecture for an electronic digital computer with subdivisions of a processing unit consisting of an arithmetic logic unit and processor registers, a control unit containing and instruction register and program counter, a memory to store both data and instructions, external mass storage, and input and output mechanisms shown in Figure 1. 2.

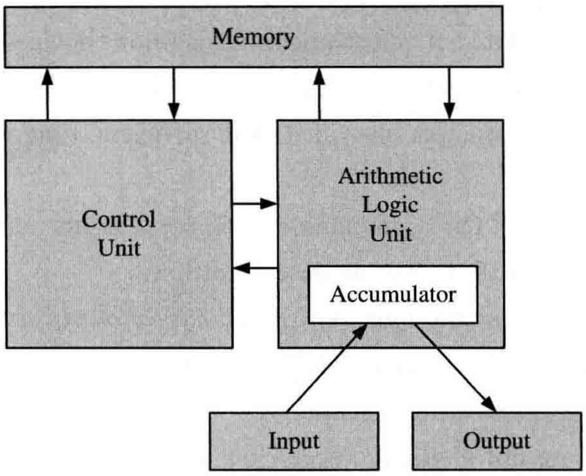


**Figure 1. 2 Von Neumann Machine Architecture**

Von Neumann architecture has evolved to mean a stored-program computer. A stored-program digital computer is one that keeps its programmed instructions, as well as its data, in read-write, random-access memory (RAM). Stored-program computers were advancement over the program-controlled computers.

- A stored-program computer includes by design an instruction set and can store in memory a set of instructions (a program) that details the computation.
- A stored-program design allows for self-modifying code.

On a large scale, the ability to treat instructions as data is what makes assemblers, compilers and other automated programming tools possible. One can "write programs which write programs".

## 1.2 Software Development

This book is about software development; let's first take a look at the definition of software and software development at *Wikipedia*.

> *Computer software, or just software, is any set of machine-readable instructions that directs a computer's processor to perform specific operations.*
>
> *Software development is the development of a software product.*

### 1.2.1 Complexity

Ask a programmer, an architect, a project manager, a business analyst or almost anyone who knows anything about software development and you will get various answers along the lines of:

- Poorly document requirements.
- Changing requirements.
- Insufficient analysis.
- Poor communication.
- Poor estimates.
- Unrealistic deadlines.

If you look at all these answers, you can see that the root cause is simply due to complexity. Coping with change just magnifies the complexity. The real problem of software development is dealing with complexity and change. Over the last thirty years, many different paradigms have been created in attempt to solve the software crisis, still there is no single solution to the crisis. The software industry acts as if it were a fashion industry by chasing new methodologies, techniques and tools.

Complexity and uncertainty are two biggest enemies of software development. Unlike enemies in a real war, complexity and uncertainty have no obvious weaknesses, and cannot be deceived. Instead, software projects have a tendency of deceiving us. More often than not, our schedules are optimistic, and we usually under-estimate the complexity and effort required for developments and testing tasks.

Unlike a real enemy, complexity and uncertainty will find all your weaknesses. The only way to defeat complexity is through understanding the requirement as well as simplifying the solution. The big difference between a great general and a good general is knowledge, and great generals are always thinking and trying to understand their enemy.

You can say the same thing about programmers; the great programmers are always trying to understand the requirement.

Complexity was recognized as a significant challenge in software development. The complexity of software can be analyzed from the aspects of structure complexity, state complexity and behavioral complexity. For example, an e-commerce application as shown in Figure 1.3 includes customer, order, product objects as well as order entry process, order fulfillment process and payment process, also an order can either be approved or be rejected, or be in pending state.

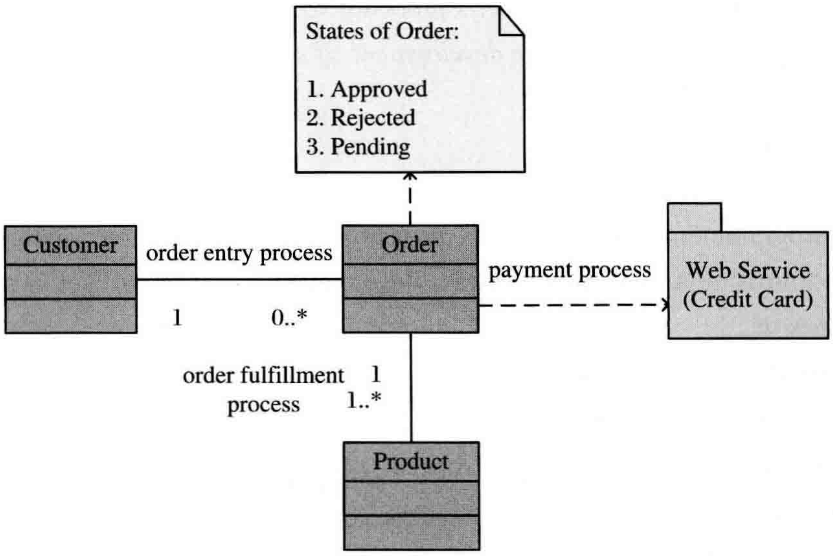


**Figure 1.3 E-Commerce Software Applications**

Software is a discrete system which can be described as objects, relations, processes, constraints and interactions with external system. The set theory can be used to describe software system as below:

$$\text{Software System} = (O, R, P, C, I)$$

where

O is a set of objects of the system, $O = \{o_1, o_2, \cdots, o_n\}$;

R is a set of relations between pairs of the objects in the system, $R = \{r_1, r_2, \cdots, r_n\}$;

P is a set of processes (or functions), $P = \{p_1, p_2, \cdots, p_n\}$;

C is a set of constraints on the objects and processes, $C = \{c_1, c_2, \cdots, c_n\}$;

I is a set of interactions with external systems, $I = \{i_1, i_2, \cdots, i_n\}$.

A software system, $S = (O, R, P, C, I)$ can be illustrated as shown in Figure 1.4.

Now let's take a look at structure complexity first. Structure complexity is determined not only by the number of objects, but also by the relations among the objects. The structure of software system is a set of objects and relations, if we define,