

NO NONSENSE: PROGRAMMING CONCEPTS

A JARGON FREE INTRODUCTION TO
WRITING SOFTWARE SYSTEMS

BY LEON BROWN



FEATURING SMART CONTENT:
USE YOUR SMARTPHONE TO ACCESS
INTERACTIVE EXERCISES AND VIDEO

No Nonsense: Programming Concepts

Copyright © 2018 Nextpoint Solutions Ltd.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Nextpoint, its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Nextpoint has endeavoured to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Nextpoint cannot guarantee the accuracy of this information.

First published: April 2018

Nextpoint Solutions Ltd.
Bulloch House
10 Rumford Place
Liverpool
L3 9DG

Table of Contents

Software Development Introduction.....	4
1 Overview.....	4
.....	5
1.1 No Nonsense Principles.....	6
1.2 Focus on Immediate Requirements.....	6
1.3 Keep It Simple.....	7
1.4 Keep It Real.....	9
1.5 Don't Repeat Yourself.....	10
1.6 Elegance Is Always Secondary to Functionality.....	12
1.7 Short Term Laziness Is Bad.....	12
1.8 Strategic Laziness Is Good.....	13
Programming.....	13
1.9 Programming With Code.....	14
1.10 Programming Languages.....	15
1.11 Language Variations.....	19
1.12 Choosing the Right Language.....	20
1.13 Future Programming.....	23
1.14 Programming Today.....	26
1.15 Later this Week.....	27
Types of Software.....	28
1.16 Hardware.....	29
1.17 Operating System.....	31
1.18 Applications.....	32
Formatting vs Programming.....	37
Conclusion.....	43
Exercise.....	44
2 2	45
Input / Output Concepts.....	45
1 The Link.....	45
2 Analogue to Digital.....	46
3 Speech.....	49
3.1 Output: Spoken.....	49
3.2 Input: Listening.....	51
4 Data Storage and Access.....	64
5 Display.....	69
6 Conclusion.....	71
7 Exercises.....	71
Data.....	72
1 Purpose.....	72
2 Primitive Data.....	73
2.1 Boolean: True/False.....	74
2.2 Numbers.....	75

2.3 Char.....	77
3 Advanced Data Types.....	78
3.1 Strings.....	79
3.2 Colours.....	79
3.3 Images.....	82
3.4 Animation.....	91
3.5 Audio.....	94
4 Detection of Advanced Data Types.....	98
5 Making Sense.....	99
5.1 Raw Data.....	99
5.2 Formatted Data.....	101
5.3 Structured Data.....	105
6 Compressed Data.....	118
6.1 Dictionary Compression.....	119
6.2 Maths Compression.....	122
7 Encryption.....	134
7.1 One Way Encryption.....	135
7.2 Two Way Encryption.....	138
8 Conclusion.....	140
9 Exercises.....	140
Processing.....	142
1 Concepts.....	143
1.1 Abstraction.....	144
1.2 Condition.....	144
1.3 Data.....	144
1.4 Function.....	145
1.5 Loop.....	146
1.6 Object.....	147
2 Validation of input and output. XSS.....	149
2.1 Processing advantages for speech to text.....	149
3 Decompression.....	149
3.1 Data Dictionary.....	150
4 Encrypted Data.....	150
Testing.....	152
1 Data validation.....	152
1.1 Automated testing.....	153
Being Agile.....	156
Real Projects and Jobs.....	185

1

Software Development Introduction

Useful systems exist for one purpose; to provide output of interest to its users. From simple web pages to complex computer games and decision systems, the role of a software system is defined through how it treats information to provide useful output. This chapter provides an introduction to the main concepts of software systems that lead to delivery of output for the end user.

1 Overview

Software systems are made from components that consist of the following processes:

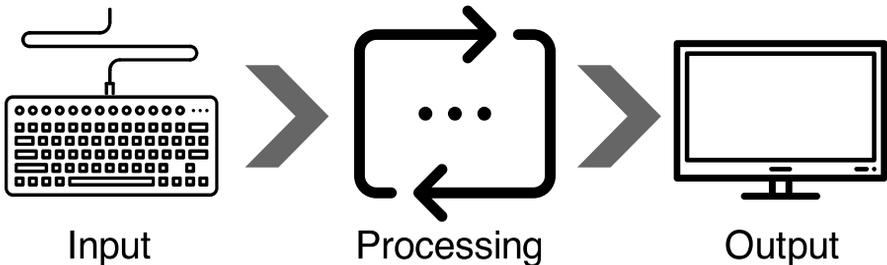


Figure 1.1: Process of each component inside a software system.

From the simplest systems to the most complicated, this model is consistent for all software. All complexity emerges from simplicity – that is, software systems become more complicated as the number of their components increase.

“All complexity emerges from simplicity – that is, software systems become more complicated as the number of their components increase.”

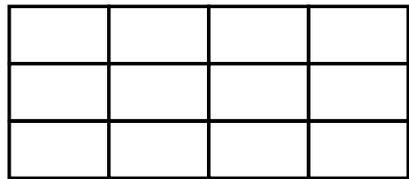
Control of complexity is an important consideration for good software design. While the primary concern is to guarantee that the software produces the intended output, increasing complexity creates the following problems:

- Increased testing requirements.
- More scope for faults to exist.
- Higher risk of faults being introduced during maintenance.
- Increased difficulty to add new functionality.
- More difficulty to change existing functionality.
- Higher hardware requirements.

Awareness of these problems allows for code to be proactively developed in ways that reduce complexity and increase maintainability.



Low Complexity



High Complexity

Figure 1.2: *High complexity is made from individual units of low complexity.*

1.1 No Nonsense Principles

Programming is as much an art as it is a science. Like film critics debate whether “Star Wars: The Last Jedi” was worth watching, the same applies to software code. There will always be multiple ways to describe software functionality through code, with programmers developing their own programming style and technique preferences; all being subject to opinion.

With so much choice for programmers to choose from to write code, it’s important to remember that not all code is equal. Let’s look at principles that will help you to create more good quality code, with none of the bad and less of the ugly...

1.2 Focus on Immediate Requirements

Firstly, the most important concern for code is to make sure that whatever functionality and output it delivers matches the requirements. While academic courses and “elite” programmers often imply that elegant programming techniques are the primary focus in software development, this alone will never guarantee that the end result will match the project’s specifications.

The most important concern for code is to make sure that the resulting functionality matches the project’s requirements.

Secondly, the other part of this principle is the focus on what’s “immediately” required. While all code results in new features appearing in the software, some of these features are required more urgently than others. Focusing on immediate requirements will allow you to get a functioning version of the software into the hands of users more quickly so that their benefits can be delivered sooner rather than later. A focus on immediate requirements also allows more achievable deadlines to be negotiated and avoids time being wasted on features that turn out not to be required.

It’s great to have high quality code using object oriented principles and the like, but all this elegance becomes pointless if the code doesn’t do the job it’s intended to do. Elegant code also requires more time to develop, increasing the risk of incomplete or missing functionality when time is a factor. In this type of

scenario, ugly “spaghetti” code that delivers the full specification criteria will always be the better code, despite its lack of elegance.

1.3 Keep It Simple

Good communication gets the message across in a way that’s accurate and easily understood; nothing more and nothing less. Let’s look at the two components of this:

- Accuracy.
Must describe the correct information that’s intended to be communicated.
- Easily understood.
Presenting the communication in a format that requires minimal effort and vocabulary to make sense of the wording.

Effective communication allows the widest range of people to understand its message with the minimum amount of effort. The more complexity that’s added to the message, the less likely the message will be understood by recipients due to limitations of their vocabulary and attention span.

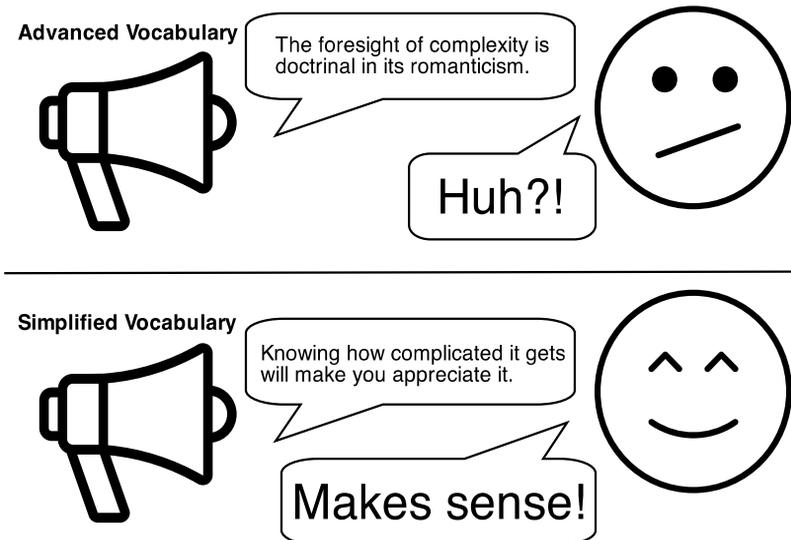


Figure 1.3: Using language to demonstrate how advanced vocabulary is understood by fewer people.

Software Development Introduction

Writing code is the same as communication; after all, it's a communication of rules for the computer to follow. While computers don't make mistakes, people who write their code do!

People are more likely to make mistakes in writing and understanding code when dealing with higher levels of complexity. It's in your best interest to make sure that code is kept simple so that its current and future development avoids inheriting new faults through misunderstandings of programmers.

Let's look at some of the features of low complexity code:

- **Easy to read.**
Like with any form of communication, good code should be presented in a way that is easy to understand. This allows yourself and other programmers to analyse what's happening when it comes to bug fixes and adding future updates without the risk of introducing new faults due to misinterpreting the meaning of the code.
- **Short and sweet.**
Shorter code is easier to manage compared to longwinded code, especially where the entire code set can be viewed at a glance without scrolling. Wherever the same functionality can be defined with reduced code, less is almost always more.
- **Modular.**
Functionality that requires higher complexity can and should be broken down into smaller and easier to manage components. A modular approach to creating functionality allows for easier to understand code that can be fixed and adapted without breaking the bigger picture.
- **Literally defined.**
Advanced code concepts such as object oriented programming are the equivalent to advanced language vocabulary. While there are advantages to using these concepts in your code, you should always remember that they increase the skill requirement for other people to make use of your code. Avoid using advanced concepts that don't provide enough strategic value to justify their use. While most academics and "elite" programmers will balk at this suggestion, it's important to remember that people who need to adapt your code may not have advanced programming skills. Like with language

Software Development Introduction

vocabulary, keeping your code concepts simple means that more people will have the skills to work with it in the future.

- **Easy to follow.**
Code created with too many dependencies becomes difficult to understand. Even where the code is written in an easy to understand format, it becomes difficult to track components and their requirements when they are unnecessarily spread across different locations within the code's architecture. Make code easy to follow by keeping everything self contained wherever possible.

There is no set rule for creating low complexity code, meaning that some of these features contradict each other. Good code is all about finding a balance between the benefits offered by each feature and deciding which options are best suited to the situation you are writing your code to address.

1.4 Keep It Real

Ambition can lead to positive outcomes for a software project, but is also a factor that needs to be managed properly to avoid disappointment. Optimism is the main culprit behind this risk, especially when expectations of other people in the project need to be managed. Being too optimistic establishes a situation where expectations exceed what's achievable with the available skills, time, budget and technology.

Let's look at the main factors for keeping realistic expectations:

- **Skills**
Your ability to write and manage the software components is the first hurdle that defines whether expectations can be met. Risk of failing expectations increase whenever there are requirements you have no prior experience of developing. While it's often necessary to learn new skills on the job, you need to consider the implication of "learning time", along with the possibility of overestimating your ability to learn the new skills. Being too optimistic will result in failing to meet deadlines or missing features from the project specification. Similarly, being too conservative in your judgement to learn new skills will limit your ability to deliver the project's requirements.
- **Time**

Software Development Introduction

Available budget, your availability and business opportunity incentives are the three factors that dictate the amount of time available for the project. It's easy to be too optimistic about what can be achieved in the available time, so make sure to reserve a margin in your estimations for unexpected difficulties.

- **Budget**

The majority of software development costs come from time investment, so the total cost is mostly proportionate to the amount of time required. While there may not be a specific time target for the project's delivery, the software must be completed within the amount of time that the budget covers. Wherever the budget is funded on a fixed price basis, being overly optimistic can lead to the software developers losing money on the time they invest into the project. Meanwhile, teams pitching for software development contracts will find it difficult to win anything if they significantly overestimate budget requirements.

- **Technology**

Aside from skills, technology is the only other factor dictating what is technically possible. With enough time, budget and skills, anything can be achieved if the required technology is available. Limitations of this may come in the form of whether the project can access the technology, such as having the required budget to cover its costs, along with the necessary skills to make use of it. Optimism on the ability to access required technology, whether through budget or its existence, is a risk that could backfire. Similarly, a conservative mindset that dismisses the possibility of accessing technology can lead to wasted time "reinventing the wheel" or lost opportunities for the project to deliver beneficial results.

Being realistic isn't about eliminating optimism from requirements estimation; it's about defining realistic ambition with consideration to any scope for negative consequences.

1.5 Don't Repeat Yourself

New programmers and so called "cowboys" are often tempted to replicate software features through copy and pasting from a previous version of the functionality. Stop right there if you find yourself doing this!

Unlike coursework and small personal projects, commercial projects are typically subject to changing requirements. All code is a reflection of the project's requirements, so when these change, the project's code also needs to change. This introduces problems for managing and testing your code if it has been copied and pasted “tens, hundreds or even thousands” of times throughout your project.

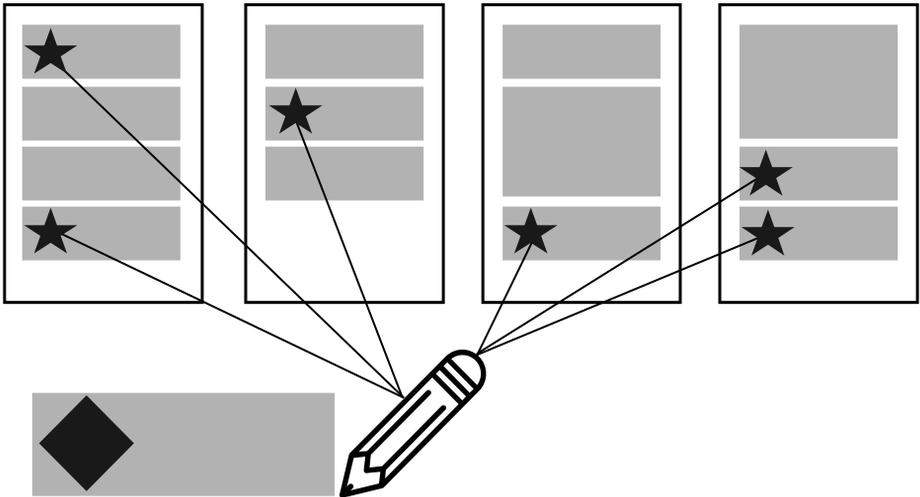


Figure 1.4: *Manually duplicated code requires intensive effort to update and test.*

The answer to duplication of software features without the duplication of code is the use of elegant programming patterns and techniques. Examples of this include object oriented programming and the use of n-tier software patterns. These concepts are explained in more thorough detail in chapter 4, Processing.



Interactive:

Discover the advantages of being able to duplicate functionality without duplicating code.

- Problems of duplicated code.
- Concept of reusable code.

1.6 Elegance Is Always Secondary to Functionality

While elegant code protects against maintainability nightmares emerging from repeated code and the like, taking this to the extreme causes other problems that are equally as damaging to the project. A software project's primary focus should always be to make use of the best approach to deliver the required features.

While strategically useful, elegant code is often more costly to develop in the short term due to the extra thinking and planning requirements. This approach also requires people to have a higher level of skill and experience with the applied elegant programming techniques; highly skilled people are also more expensive than their beginner and junior level counterparts.

The aim of elegant code is to reduce long term production costs by allowing future updates and testing to be performed with minimal effort. However, code that is too elegant requires more effort and skill to work with. Problems that emerge from this include:

- Difficulty in understanding how the code works leads to increased scope for mistakes to be made.
- Elimination of task delegation to lower skilled (and hence lower paid) programmers leads to increased maintenance costs by limiting deployment options to more experienced senior level developers – who are more expensive to hire.
- Increased learning and analysis requirements – executed at least once for each new developer introduced to the project. This may also need to be repeated if developers have taken a break, such as being deployed to other projects.

Where code is too elegant, it can become as difficult, if not more, to maintain the software than poorly written “spaghetti” code. The main difference with overly elegant code is in how maintenance options are restricted to using higher skilled senior level developers who are significantly more costly to hire.

1.7 Short Term Laziness Is Bad

Taking upfront shortcuts usually leads to more work requirements further into the project. Despite these shortcuts initially allow for rapid progress, the

problems they produce later in the project's lifecycle lead to an eventual decrease in productivity when:

- More time is spent on fixing broken code.
- Change requirements become problematic to implement.

The engagement of change requirement modifications leads to “blowback”, where updates not only take extra time to implement, but are more likely to produce new faults that require additional time to fix. Meanwhile, the longer the project is extended by these problems, the more likely additional change requests will emerge.

1.8 Strategic Laziness Is Good

While short term laziness produces long term problems, there are still ways to use shortcuts to reduce your overall workload. Good programming is all about being able to do more with less time, so it makes sense for strategies to have a focus on reducing the need to write new code – i.e. efficiency.

Write less code to do more of the work...

The main difference with this principle is the focus on producing code and development processes to be highly flexible, reusable and automatic wherever possible. This is not about an ability to copy and paste code across the project, but allowing an individual set of code to deliver multiple functionality and to be easily adapted for future requirements without major updates. The time benefits delivered by this principle can mean the difference between investing several hours versus just a minute or two to implement and test each change request.

Programming

The process of defining how software systems should work is called programming; a skill requiring the ability to imagine, design and describe processes in a logical order. In the same way that anyone can draw a picture, anyone can program a software system, but only experts have the skill to design with accurate detail and quality.

Anyone can draw a picture, but only an expert has the skill to design with accurate detail and quality.

The way in which software is defined is always a secondary concern to what is being programmed. No matter how good the quality of a program's definition, it will always be of little or no use if not defined to solve the right problem in the correct way. Therefore, learning to define software correctly is the primary skill that is transferrable to all forms of programming. Learning how to represent a program's design in a format such as code is always a secondary skill of a good programmer.

1.9 Programming With Code

The most flexible approach to programming is to define the program design though code created as a series of step by step instructions that tell a computer exactly what to do. Although coding and programming are often used interchangeably, they are two different concepts. Let's take a look at their differences:

- **Programming.**
The process of defining rules for a system to follow.
- **Coding.**
The act of creating written (or otherwise) notation representing the descriptions of rules, information and concepts.

Equivalent concepts are found in story writing, poetry and other creative literature. While coding is the equivalent of the written descriptions, the programming is the structure of the story being described. In the same way that computers can support programming languages, anyone who has the ability to make sense of English will be able to understand and imagine the events of a story written in this language.. For an author, the English language is merely a format to express their ideas and imagination – in the same way that a programmer uses code to express their thinking processes.



Think About...

Writing a novel is similar to writing a program.

While JK Rowling may have had the creative skills to generate the idea for the story behind Harry Potter, she also required the technical writing skills to allow her ideas to be expressed in a way that her book's readers would understand. These writing skills include the ability to describe the scenery, characters and events in a format that helps readers to use their imagination to full effect. The literature's writing is "programming" the experience of its readers through the use of detailed descriptions and grammatical "syntax" that helps people to avoid misunderstanding any of the details being described.

Another factor that will have been considered in the "design" of Harry Potter is how the stories can be evolved in future releases – i.e. books. This design may be influenced by story strategy for entertaining fans, as well as accommodating currently unknown influences such as embracing future commercial money making opportunities and brand appeal. In this sense, the story needs to be designed with flexibility for current and future plots to be defined in ways that meet whatever requirements may emerge.

Like with the written language used, this design of the story also has parallels with programming. While stories are designed to have plots for entertaining the reader, software systems are designed to have features for solving problems. Both are required to be written in a way that is clearly understood, and flexible to accommodate new requirements that may emerge in the future.

1.10 Programming Languages

The only form of instruction that computers are able to follow is called machine code; a language whose "alphabet" consists of just two characters – 0 and 1. While this approach is easy for computers to understand, it is very difficult for

Software Development Introduction

us mere humans to make sense of. This causes problems wherever there is a need to write software:

- Takes longer to learn.
- Fewer people able to learn.
- Longer to write software.
- Faulty code is more likely to be created.
- More difficulty in correcting the faults.

Language is all about defining a meaning that makes sense to the receiver. The critical elements of language are vocabulary, followed by grammar. Let's compare what machine code vocabulary may look like compared to its English counterpart:

Machine Code	English Description
00000000	Stop Program.
0000000 <u>1</u>	Turn bulb fully on.
000000 <u>1</u> 0	Turn bulb fully off.
00000 <u>1</u> 00	Dim bulb by 10%.
0000 <u>1</u> 000	Brighten bulb by 10%.
000 <u>1</u> 0000	If bulb is fully on, skip over next instruction.
00 <u>1</u> 00000	If bulb is fully off, skip over next instruction.
o <u>1</u> 000000	Go to start of program (address 0).

Instructing the computer to gradually increase the light until it reaches full brightness would look like:

```
00001000
00010000
```

Software Development Introduction

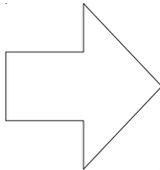
```
01000000  
00000000
```

While this makes perfect sense to a computer, it doesn't make much sense to humans. Let's look at this code using the English equivalent descriptions:

```
Brighten bulb by 10%.  
If bulb is fully on, skip the next instruction.  
Go to start of program (first instruction).  
Stop Program.
```

This version of the code is now easy for humans to make sense of, but is unusable by computers. The complexity of modern computer systems means that it's unfeasible to expect programmers to learn and write advanced machine code to meet the level of requirements and time constraints of typical software projects. There is a need for people to have the ability to write code in ways that are productive and maintainable, while at the same time allowing the code to be understood by computer systems.

Fortunately, it's possible to use "middle layer" software to translate human written code into machine language. This type of software is not yet advanced enough to accept plain English as the instruction language, but it does allow instructions to be provided using a cut down version of everyday language. Let's look at how this could look:

English Notation	Translation	Machine Code
<pre>Brighten(bulb,10%) If bulb.bright < 100% Then Go start End</pre>		<pre>00001000 00010000 01000000 00000000</pre>

Like English, this language has a "syntax"; a set format that words and grammar must be used with in order for the instructions to be valid. Let's look at the how each part of the example code has been constructed:

Instruction	Type	Description
Brighten	Function	Will attempt to brighten the provided

		target by the specified amount. In this case, the target is “bulb” and the amount is “10%”.
If	Keyword	Checks to see if a condition is true. In this case, it is checking to see the bulb’s ‘bright’ property is “less than” (<) 100%. Any bright value of bulb that is less than 100 will produce a result of “true”.
Then	Keyword	Used in combination with the previous “If” command, the “Then” command will execute the following instruction whenever a the previous condition returned true. In this case, the “Go” command is executed.
Go	Keyword	Sends the program’s execution to a specific point in the code. In this case, start is the beginning of the code – i.e. the first instruction.
End	Keyword	Instructs the computer to stop executing any further code in the program.

The language has two types of command – functions and keywords. The difference between these types of command is that keywords are instructions of the language, while functions are instructions created by the programmer. In short, a function acts as a container for executing several keywords in a specific order defined by you; the application programmer.

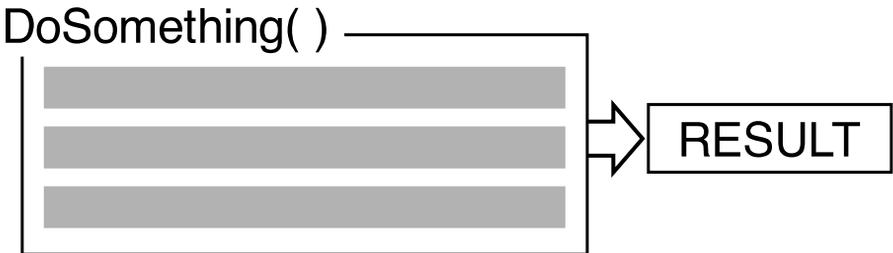


Figure 1.5: A function executes a collection of instructions in order to return a specified result.

Functions are mainly useful for allowing functionality to be reused without the need to repeat code. These functions are made more reusable by their ability to accept parameters that allow for customised reactions to different information provided. In this case, the “Brighten” function accepts parameters for target and amount, with each separated by a comma.

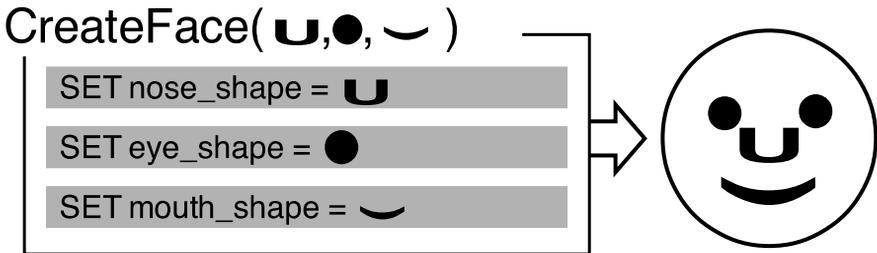


Figure 1.6: A function using parameters to affect the returned result.



Interactive:

Use animation concepts to understand the relationship between keywords and functions.

- Select keyframes to use.
- Create individual animations from keyframes.

qrin.uk/cs/3

1.11 Language Variations

While all programming languages beyond machine code are designed to make programming easier to some degree, each language is designed with an emphasis that makes them ideal for different purposes. Examples include:

- Easiness to write code.
For creating code quickly to put something together without needing advanced programming skills. Useful for developing prototypes of an idea, or for use by people who need to customise software functionality for a more broader job role.
- Enforcing high quality code.

Software Development Introduction

Makes life easier in software maintenance for people with advanced programming skills.

- **Easiness to learn.**
Allows people to learn foundation concepts of programming without being confused or distracted by more advanced concepts.
- **Execution speed.**
For types of software that require fast processing in order to be usable or useful – such as AAA computer games and scientific modelling system.
- **Domain specific purposes.**
Artificial intelligence, data management, web page content and multimedia content are examples where specialist software systems are used.

All languages share support for foundation logic principles and grammar – otherwise known as syntax. It's not uncommon to see languages share the same names for commands and grammatical syntax, hence making it significantly easier to learn additional programming languages.

While all programming languages share foundation logic principles, some are more focused on supporting advanced concepts such as object oriented programming, while others keep their focus on providing a language that's easy to use. There are also languages that provide a mixture of easier programming with features for advanced concepts.

1.12 Choosing the Right Language

In addition to some programming languages being designed for a specific purpose, complexity is another factor of suitability. While advanced software developers require advanced features to create advanced software systems, the complexity of this technically “ideal” programming language is generally overwhelming for people with a lower level of skill.

Advanced software systems require advanced programming features to express advanced concepts...

Software Development Introduction

For beginners, programming languages that focus on simplicity make it easier to learn foundation principles without unnecessary distraction. This has a significant advantage for building a solid understanding of programming concepts that can be expanded upon, as well as an understanding of strategies to navigate and read code. While keeping code concepts simple, learners are able to develop awareness that leads to an ability to be self sufficient and confident in their abilities to independently solve problems.

Beginners need a focus on simplicity to avoid unnecessary distraction.

From a commercial perspective, many software projects don't require the use of advanced programming concepts. In these situations, advanced programming increases the skills required for future maintenance updates, resulting in unnecessary cost increases. This is because people with advanced programming skills are harder to find, and hence more expensive to hire. It makes better sense to develop simple software functionality using tools and code that regular IT staff can learn quickly or already use.

However, not all software project requirements can be developed with simplified code. The use of advanced programming concepts is a critical element for retaining control of complicated software projects. While these concepts are more difficult to learn, their payoff is delivered many times over through saved time, ability to adapt the software and increased quality. For these projects, it makes sense to use a programming language that offers advanced features to allow programmers with advanced knowledge to produce code that can fully benefit from the advantages of using advanced programming concepts.

Language	Purpose
SQL	A low complexity and easy to understand query language created specifically for interactions with database systems. Allows analysts and software applications to easily request and manage details about data in order to find answers to questions. www.mysql.com
Visual Basic	Easy to learn, allowing new programmers and non specialists to

Software Development Introduction

	<p>create desktop software applications and extended functionality for Microsoft Office.</p>
Python	<p>English like language that allows functionality to be created for desktop and server applications. It is also used by some software applications to allow new functionality to be added.</p> <p>www.python.org</p>
Monkey	<p>Created specifically for the development of multimedia applications such as games. Allows for advanced programming concepts to be expressed in a highly English like language.</p> <p>www.monkeycoder.co.nz</p>
Javascript	<p>Originally and primarily used for allowing interactivity in web pages. As the web has evolved beyond merely providing online “brochure” style content, Javascript has become the standard for programming application functionality into web based content.</p> <p>In more recent times, Javascript has been used outside of the web browser, such as within sensor based devices and web services. This language is the one to watch for future programming standards.</p>
PHP	<p>A scripted language primarily used for controlling the server side operations of web applications. It is designed with intermediate complexity that allows code to combine advanced and simplified programming styles. Unlike many languages, PHP provides programmers with choice on how to define code – for good or bad.</p> <p>www.php.net</p>
Java	<p>A very strict language that enforces code to be written to high standards. This is a significant advantage for commercial applications where code quality has a direct impact on flexibility for change control and other project issues. The same factors become a significant disadvantage for beginner programmers</p>

	<p>due to a steep learning curve being imposed; not only to learn about programming, but also the need to learn how Java will accept code.</p> <p>www.java.com</p>
Assembler	<p>Created as a direct comparison to machine code instructions. Unlike all of the other languages mentioned, there is no focus on making code related to everyday English beyond presenting individual machine code commands as shortened words. For example, using JMP as the command for “jump to memory address”. This language requires high skill and knowledge of how to use the specific hardware that’s being programmed – without any support of higher level concepts provided by programming languages that make code easier to develop.</p> <p>While this language is no longer used for the majority of commercial software projects, it is still useful for developing custom code for low specification hardware. These are typically miniature low cost computer components found inside electronic devices such as toasters and DVD players. These systems provide highly specific features such as timer and sensor input functionality that are relatively simple to code and require minimal memory and hardware resources to execute.</p>

1.13 Future Programming

Programming technologies have consistently progressed in reducing the difficulty of writing code. While programmers originally had to manually program with machine code, the introduction of Assembler made expert programmers more productive by allowing them to write human readable instructions. Since then, new programming languages have emerged to eliminate the expert knowledge required for directly programming hardware, replacing the emphasis on using code to define logical rules with an increasingly more “English” style format.

Machine Code (1940s - 1970s)	Assembler (1970s - 1990s)	Javascript (1990s - present)
<pre>00001000 01010001 01000110 00000110</pre>	<pre>LD a,10 LD b,a add a,b sub 6</pre>	<pre>Var a = 10; var b = a; a = a + b a = a - 6;</pre>

Figure 1.7: Comparison of how programming languages have become easier to write and understand.

This pattern of increasingly easier options to create software points towards a future where programming is defined through easier methods than is currently the norm. While there will always be a need for people with skills to write code, industry demand is and will continue to move away from the need to manually write code and move towards more “natural” methods to design the logic behind software systems. Technology advancements are likely to give rise to new methods of defining code, such as through pictures and diagrams that don’t rely so heavily on skills in maths and writing.

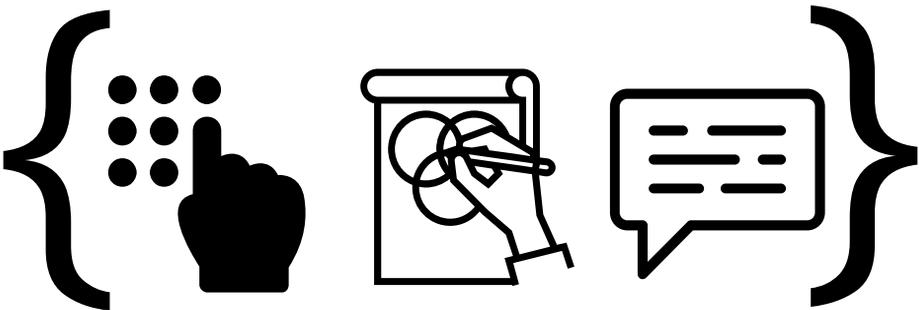


Figure 1.8: Future programming will likely take many more forms than just written code.

Although it sounds like the technology for this is far away, the reality is that the foundations are already in place. Let’s look at how modern approaches to human computer interaction are likely to influence the evolution of programming:

User Interfaces

The use of visual interaction concepts as seen on typical computers including Windows PC and Mac are now the most common method of defining instructions for computers. These visual interfaces have meant that the average non technical person can easily figure out how to use computer systems without the need to learn any command line programming instructions that were previously a mandatory requirement before the introduction of systems such as Microsoft's Windows and Apple's Mac OS.

Similarly, the use of visual user interfaces have already allowed software systems to be defined without the need to write any code. This visual approach to creating software often isn't perfect by any means, but still allows people who have little or no code writing skills to create databases, web pages and simple apps.

Automated Code

Through the use of tools based around visual and spoken interactions, the code for software features can be defined without being directly written by a human. Not only does this approach eliminate high skill requirements, but also opens scope to increase efficiency for software development.

Natural Language

Advancements in natural language recognition has led to the emergence of digital assistants such as Amazon's Alexa and Siri from Apple, allowing people to provide spoken instructions and requests to their smart devices such as smartphones and computers using spoken language. Not only do these technologies allow speech to be used as a form of human/computer communication, but also using regular spoken language – meaning no prior knowledge is required to learn specialised instructions or language syntax.

Written and visual communications are also forms of natural language that technology has made advancements to make use of. The ability for computer systems to recognise handwriting, pictures, video and photography already exist, meaning that these are already candidates for future programming methods.

Artificial Intelligence

Combining the use of user interface interactions with natural language recognition and the ability for software systems to write their own code can be further supported with the use of artificial intelligence (AI) technologies. The application of this allows computer systems to work with human programmers to create better code, design more effective features and identify better ways to solve problems.

An example of the application of artificial intelligence to software programming is genetic algorithms. Based on evolutionary theory's concept of natural selection, these algorithms have the ability to monitor and rewrite their own rules. Through a process of experimentation and performance analysis, genetic algorithms deploy multiple versions of "experimental" revisions in order to identify and choose the best performing updates. It is this process that allows genetic algorithms to become better at whatever they have been created to do as they gain more exposure to new experiences.

1.14 Programming Today

While the foundations are already in place for a new revolution in the process of defining software systems, coding skills are still going to be in demand for the majority of software projects in the foreseeable future – at least the next decade or two. Serious software development still requires functionality to be defined efficiently through the use of code, which is currently the only option if you want complete control of defining the software's functionality.

Although the demand for coding skills isn't going to disappear anytime soon, requirements for coding is and will continue to evolve in response to industry demands. The primary influence of businesses expenditure is the need to control costs, and as the saying goes – time is money. Any type of programming resource that reduces costs through faster production and lowering skill requirements are good candidates to be adopted by commercial projects. A resource becomes a "trend" when adopted by many industry projects. When this happens, skills and knowledge of using these resources become sought after by employers. Let's look at some of the trends that are emerging in industry:

- **The rise of Javascript everywhere.**
From humble beginnings as a simple scripting language for basic web page interactivity, Javascript has evolved for use in all types of software application. The ability to use the language for extending

Software Development Introduction

functionality in a diverse range of applications from Adobe Photoshop to Microsoft Office, as well as for creating standalone apps for mobile, desktop, server and sensor devices means that Javascript code is highly portable and reusable. As well as Javascript being a much easier language to develop with compared to counterparts such as Java and C++, this write once to use anywhere ability provides industry organisations with an incentive to choose Javascript for reducing/eliminating time and costs to rewrite software functionality for different platforms.

- **Frameworks.**
Pre-made code created to extend the default ability of a programming language. These frameworks are designed to allow projects to quickly get started with the benefits of a robust foundation that can withstand future change requirements. These frameworks reduce costs associated with upfront startup time, documentation, ongoing maintenance and skill requirements.
- **Legacy languages.**
Systems developed in the past still exist, so although new projects don't tend to use languages such as Perl, there is still a need to extend and adapt software developed with these older technologies. Eventually, these systems will be replaced when they are no longer required or where their maintenance becomes unfeasible.

Through learning to code, you also learn transferrable knowledge required for any type of software development.

1.15 Later this Week

The evolution of software development is gradually moving to embrace forms of programming that aren't so heavily dependent on manually writing code. One of these emerging patterns is through the use of specialised software development tools that allow instructions to be created through user interface interactions. These tools also allow features to be supported by so called "scripts", which are manually written code to support the features created via user interface interactions.

- Microsoft Office / Open Office
- Unity

- Adobe Captivate
- Filemaker / Microsoft Access

Anyone who has knowledge and experience of writing code will always be in a better position to embrace new programming technologies. People with coding skills can quickly write scripts to extend functionality and define software more efficiently without any reliance on user interface features – which can be slow and confusing, if available at all. Knowledge and experience gained through code based programming is also transferrable to other forms of programming, hence allowing you to adapt more quickly to different development tools using your knowledge of their underlying principles.



Watch and Learn:

Identify how visual authoring tools are making it easier to create interactive content.

- Visual programming.
- Minimal coding.

Types of Software

Code written using a programming language forms an end product known as a program – or software. There are three primary categories that software programs can be created under; hardware, operating systems and applications. Let's look at what these are.

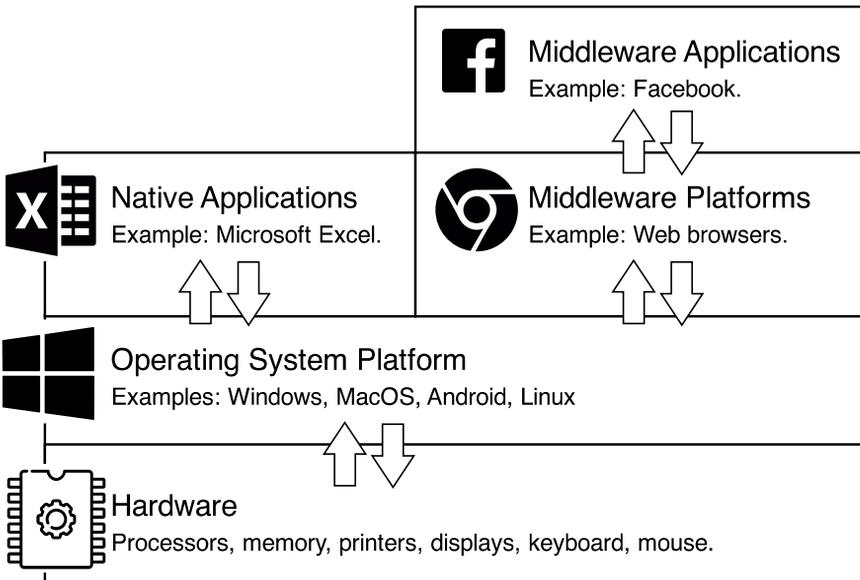


Figure 1.9: How software layers allow applications to run across different types of hardware and operating systems.

1.16 Hardware

Hardware level software is primarily focused on the logistical operations required to run the computer hardware. In short, hardware layer software is developed to provide computer systems with the ability to quickly get started and perform the task it was created to do.

Operations of hardware level software include managing input and output from devices such as the keyboard, screen display and data storage resources. Hardware level software is critical for allowing the computer system to access other types of software that are not embedded directly within the hardware, as well as allowing for the most basic of interactions to occur between the computer and its user.

Microcontrollers

It's uncommon for modern commercial projects to result in code written for the hardware level; primarily because of the high skill and time requirements increasing costs, as well as this type of code not being reusable for other types of

hardware. Software projects written at this level tend to be very small, simple and for a specific purpose.

A common motivation for writing software at this level is to reduce costs incurred in hardware production. For example, a toaster requires a very simple timer function to “pop the toast” according to the user’s preference. Using a fully featured computer system to control this would be overkill, and increase hardware costs to more than the value of the toaster. A more suitable option would be to use a microcontroller; a minimalistic low specification computer contained on a single chip.

Unlike more advanced computer systems, microcontrollers have just a few bytes of memory and operate at a very low speed. Their low specification are just enough to perform a highly specialised task, such as counting time to “pop the toast”. This low specification is also reflected by their price; often at just a few pence – making them ideal for controlling specific functionality within appliances and sensors.

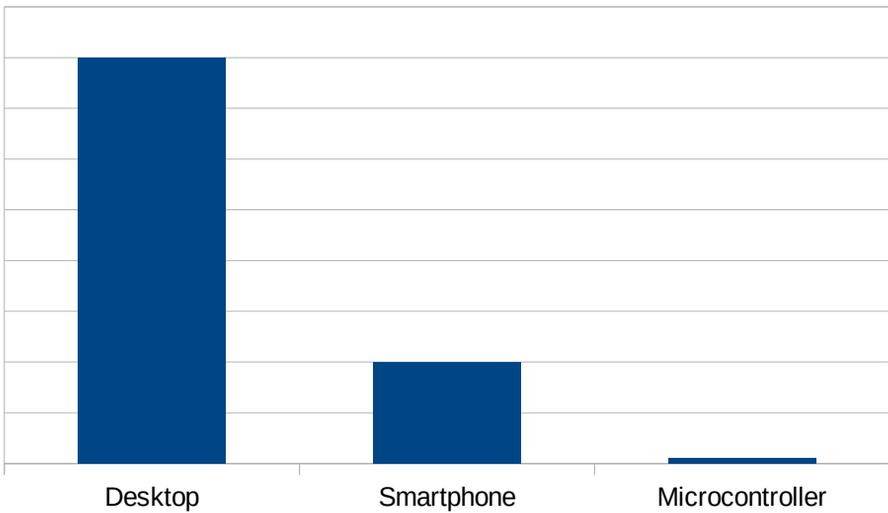


Figure 1.10: Comparison of microcontroller memory to smartphone and desktop computers.

1.17 Operating System

The next level up from hardware level software is the operating system. This layer exists to allow applications software to be created without concern for specific hardware. Several advantages exist for writing software applications for an operating system instead of for specific hardware:

- Applications can run on any type of hardware supported by the operating system.
- No need for applications to have code for managing the hardware – these are available as services from the operating system.
- Applications automatically benefit from operating system upgrades – such as gaining support for more hardware, security and efficiency.
- Services provided by the operating system significantly reduce requirements for development and testing of applications software.

Responsibilities of this layer are focused around the management of hardware resources. The operating system acts as a middleman, providing “services” to allow hardware and software applications to interact. Examples of these services include:

- Send/receive data via the Internet.
- Update/read data in memory.
- Store/read data in file storage.
- Update a screen display.
- Send a document to a printer.
- Automatic management of resources – such as:
 - *Garbage collection – the deletion of memory data that is no longer in use.*
 - *Multitasking – the management of resources provided to allow multiple software programs to run simultaneously..*
- Detect user interactions from input devices such as touchscreen, keyboard, joystick or mouse.

Just like programming languages allow more programmers to create advanced functionality without requiring advanced skills, the operating system also allows for this. By reducing the skills and time needed to directly manage hardware, application programmers are able to be more productive by investing their effort into creating “useful” features that define the purpose of their application.

In short, while hardware level programming requires the ability to define “how” to do an operation, operating system level programming reduces this to merely defining “what” operation to do.

	<p style="text-align: center;">Interactive:</p> <p>An exercise to demonstrate the role of an operating system.</p> <ul style="list-style-type: none">• Play the role of the operating system.• Translate instructions to match their target.
--	--

1.18 Applications

For general purpose computers, the applications layer is the area of software most noticeable to end users. Where the previous layers are focused primarily on the logistics of computing operations, this layer's primary focus is on providing useful functionality. Computer systems are unable to serve a useful purpose without applications.

Unlike software created at the hardware level, applications are designed to be loaded into memory as and when required by the user. These applications mostly, if not entirely, rely on controlling the computer's hardware via services provided by the operating system. The relationship is most easily described as software applications being created to control the operating system, while the operating system is designed to control the hardware.



Figure 1.11: Applications control the operating system to control the hardware.

Without a primary concern for computing logistics, the design and development of this type of software is mostly focused around solving sociological issues – i.e. people problems. From people’s need to be entertained and educated, to becoming more productive and communicate, software applications make general purpose computer systems such as desktops and smartphones become useful tools.

Applications have a prerequisite requirement for users to use the same operating system that the application was coded for. As a result of operating systems tending to be written for a limited range of hardware configurations, an additional hardware requirement is also inherited in order to run the operating system.

Middleware

The problem with applications written for a specific operating system is the inheritance of hardware compatibility. This is problematic for software applications that need to support users across different devices and markets. Although writing software applications for a specific operating system may provide instant compatibility with a range of hardware, there are still a couple of concerning issues:

- Some forms of hardware may not be supported by the chosen operating system.
- Not all users of the desired hardware will use the chosen operating system.

One strategy for software developers is to choose an operating system used by the largest number people/hardware. For desktop and laptop users, the obvious choice would be Windows from Microsoft, while applications for use with ultra portable device such as smartphones would target Google’s Android.

Software Development Introduction

While this strategy makes sense, there are a few flaws in this approach:

- **The largest audience isn't always the most suitable.**
 - *Where the aim is to make a profit from sales of the app, it's well known that users of Apple's iPhone and iPad devices are much more likely to purchase apps than user's of Google's Android.*
 - *Creative professionals generally have a preference to using Apple Macs.*
 - *Many technical types have a preference to using one of the variants of Linux.*
- **There is often a need for the application to be used on different types of device.**

While Windows is popular on desktop and laptop devices, it's very uncommon on ultra portables such as smartphones.
- **Situations often dictate the most suitable type of device.**

While a desktop computer has a lot of processing power, it's not suitable for any situation involving remote locations away from a power supply. It's for this reason that smartphones and tablets have become more popular than traditional home computers.
- **There is always scope for the technology landscape to change.**

Once upon a time, not so long ago, the desktop computer was the most common form of home computer – with Microsoft's Windows being the operating system of choice. In an unexpected twist to the story, ultraportable devices started to appear that gave an equivalent experience for using the web and entertainment applications, resulting in users switching their primary choice of computer from desktop to smartphone. Today, as of 2018, Google's Android is the most popular operating system used on consumer devices.

While an option to solve this problem would be to write separate versions of the software application for each operating system, this is time consuming and costly. Many software projects have a limited time and budget, so spending more on this to repeat work isn't ideal. This option may not be possible where there is a limitation on the available budget or time.

Enter middleware – a solution for allowing software applications to run on operating systems they were not specifically written for. The use of middleware means that software applications can be developed without concern for the

hardware or operating system. Wherever the middleware is supported, the software application is also supported; hence no need for duplicated software development.

Software developers have several option categories to choose from when making use of middleware to create applications:

- **Virtual machines**

A simulation of a computer that doesn't exist. The idea here is that software applications are written and compiled to the virtual machine's specification as though it were a real computer system; the virtual machine manages any translation required to make code work on the real computer's operating system.

Examples:

- *Java – a platform created to allow software applications to run on top of any operating system and over the web.*
- *Web – an open standard that has emerged, allowing software applications to run on any computer system that has a web browser with support for web standards.*

- **Emulation**

A type of virtual machine designed to simulate a type of computer system that exists in the real world. Emulation differs in its focus from virtual machines by allowing a computer to become compatible with existing software of another. This is mostly used for allowing modern computers to run software designed for older systems – whether it be old business applications or games created for retro game consoles. To a lesser extent, emulation is also used to allow modern computers to run the software of other modern computers.

Examples:

- *Wine – used to allow computers running versions of the Linux operating system to run software applications created for Microsoft Windows.*
- *DOSBox – allows modern Windows based computers to run software applications created for Microsoft's old operating system, Microsoft DOS.*

Software Development Introduction

- *SNES9x – created to allow games created for the Super Nintendo games console to run on computers using Windows, Linux, Mac or Android operating system.*

- **Libraries**

While most approaches to middleware focus on managing the entire functionality of the software application, libraries focus on just part of the functionality. Options are available for these resources to be installed as an extension of the user's operating system or to be packaged as part of the software application.

Example:

- *OpenGL – a graphics library that allows the same code for visual software applications to run without modification on different computer platforms.*

- **Scripting**

While the previous examples rely on using some form of compiled machine code to deliver application functionality, scripting has a big focus on storing instructions in human readable form. There are options for scripts to be embedded within a parent software application, as well as the script being the software application. Scripts are translated into machine code as they are executed – meaning they are technically slower for the computer to run, but are much easier to change.

While technically being slower for the computer to execute, many scripting languages make use of techniques such as “just in time” (JIT) compilation. This allows for scripts to be efficiently compiled before they are needed, avoiding the need to compile during the program's execution.

Examples:

- *Javascript – the language most popular for its use by web browsers to define web page interactivity.*
- *PHP – mostly used on servers to manage “behind the scenes” work required for web systems such as Facebook.*

Software Development Introduction

- *Python – Another scripting language that can be used to create server functionality, but is also used for desktop and sensor device based applications.*
- *Microsoft Office - has support for macros, which are code written in Javascript or VBA (Visual Basic for Applications) to allow new features to be added to the Office applications.*

The use of middleware changes the approach for software development strategy. Application development is no longer restricted to focusing on fixed hardware or operating system platforms. Where software has been developed for a fixed platform, middleware opens opportunities for these applications to be used elsewhere.

Formatting vs Programming

Programming and formatting are two methods of defining instructions that are often confused to be the same. While they share many similarities, both have a significantly different focus and purpose. Their most notable differences can be identified as:

- Formatting is for presentation.
- Programming is for functionality.

Understanding these differences allows software to be constructed in ways that formatting and programming code can compliment each other without compromise. The ability to define content presentation rules separately to the main functionality code is a useful ability for projects where designers and programmers need to collaborate.

While programmers may have skills to define both programming and formatting code, a designer's skillset is likely to be limited to content formatting; and not necessarily to an advanced standard. A good programming strategy would use code as a resource to provide more control to designers by responding to information contained within any formatting provided. Not only does this allow designers to take more responsibility for their work, it also helps to avoid the project becoming overly dependent on the programmer(s). More details about these issues are covered in chapter 6, Being Agile.



Think About...

Software projects – more than just code.

Many software projects require at least a small element of input from designers, content writers, musicians and other creatives. While still a software project, the resulting software is no longer the creation of programmers. All creatives seek to control the presentation of their content, whether it's for visual presentation, readability or audio.

Formatting becomes a useful tool in software development for providing creatives with the ability to control how their content is presented without the need to understand the underlying code behind the software. This provision of control also relieves programmers from being directly responsible for implementing creative changes – resulting in reduced workload and more time to focus on functionality. Even better, software developers can progress to develop functionality before the creative presentation requirements have been finalised.

Let's look at some specific examples of how this approach to software development works:

Graphic Design

People involved in creating visual content for use in software applications will be highly focused appearances; whether it be based on their vision or to match a design specification they've been provided to work from. Like with software development, visual design is a concern affected by ongoing change requests. Scope for workflow bottlenecks and ongoing technical problems emerge wherever there is a reliance on programmers to convert and integrate design content into the software system.

The use of formatting within the software system solves these issues by providing designers with the ability to control their design content. Where possible, embracing a suitable formatting standard would benefit the project by allowing designers to create visual content using graphics software they are already familiar with. Not only does this avoid the need

for designers to learn how to manually create formatting, it will also allow them to maximise their productivity and output quality.

Vector formats such as SVG (Scalable Vector Graphic) are ideal for this type of graphic integration into software projects. Unlike regular images, these formatting standards allow graphics to be defined in ways that software code can interact with. For example, the graphics designer can create an image of a car in which the code can apply reactions to components such as when the wheels are clicked on. In this case, the designer can make as many changes as they want to their visuals without needing the programmer to adapt any code.

Content Writing

Projects with a heavy element of written content require the services of a content writer. It could be a journalist for a news website, an education professional for e-learning content, or a story writer for an interactive game. Formatting can be used to describe the context of the written content in a way that code created by a programmer can decide how to make use of.

Like with graphic design content, the use of a standardised format would allow authors to use their preferred text authoring tools to write content without any need to invest effort into learning new skills. The project would also benefit from any authoring features available from these tools that make writers more productive and produce higher quality writing.

Popular formatting standards supported by writing tools include XML, ePub and HTML. All of these formats are easy for programmers to write code that can read and respond to this type of content. In many cases such as with the use of HTML, content can be styled directly by the graphic designers – further helping to avoid the programmer being the cause of workflow bottlenecks.

Language Translation

An extension of content writing that's specific to project requirements needing to support more than one language. Many strategies are available to allow content to be formatted in ways that allow code to detect the most suitable content to display. In some cases, it is possible to control the presentation of multiple languages from presentation formats

such as CSS.

Event Interactions

Useful for software applications that depend on intensive user interaction, formatting can be used to structure data descriptions that influence how code responds to user input. A typical strategy used for developing multimedia applications such as games and simulations, this approach provides content authors with the control they need to define how their content is presented in response to user interactions without being too dependent on support from programmers.

Like with graphic design and content writing, event descriptions can be created with the help of software applications. Although these formatted descriptions can be manually created, the use of a software application helps to:

- Guarantee consistency.
- Improve productivity.

The type of formatting required to describe event interactions is often outside the skillset of content authors, hence the use of software tools allowing them to take on responsibilities that would otherwise be left to the programmer. These software tools are often custom made applications created by the programmers for the project – although options are often available to use off the shelf tools.

The main motivation for using formatted data to define and control responses to user interactions is the need keep change requests manageable. Elements related to content presentation are most at risk of being affected by change requirements. Like with imagery and writing used for visual brochure style content, the design and delivery of event interactions has a significant influence on the perception of interactive content, so it's important for creatives to have the freedom they need to design scope for interactions to a high standard.

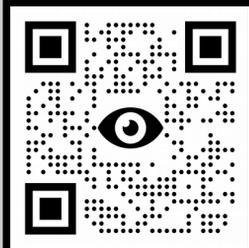
Let's take a look at the interactive story game Monkey Island. Described in its simplest form as a series of responses to the player's mouse controlled point and click interactions, it is the design of responses to these interactions that defines the quality of entertainment experience delivered. From a technical perspective, the game will use the same

functionality regardless of its story and interaction design. This foundation functionality consists of the ability to:

- Click on items.
 - Interactive items within the game’s story trigger specific responses according to “circumstances” within the game’s state when being clicked on by the user’s mouse.
- Speak to game characters.
 - Game players are able to interact with non playable game characters within the game through the use of scripted speech interactions. Like with clickable items, game players are given options to select via point and click interactions for speaking with game characters, which may change according to the current circumstances of the game’s state. Communication options chosen by game players may also affect the game status, which in turn could trigger changes to clickable items and locations.
- Change state of locations.
 - Locations in the game may display with different imagery or allow for different types of interaction based on specific states of the game. Examples of this include weather and time of day.

This functionality, especially when developed to a high standard, creates potential to deliver an exceptional entertainment experience. However, its potential can only be achieved when the creative design process has flexibility to refine the story through continuous iteration and testing; like the typical software development lifecycle.. It is this creative design process that becomes problematic for programmers – i.e. the story and game design’s continuous changes until the formula has been perfected. It is through this iteration that many changes will be required based on received feedback that only occurs after the development of the design requirements. This becomes a major problem if interactions have been “hard coded” into the software programming, leading to the project becoming too dependent on its programmers to implement every change requirement that will inevitably create an unmanageable bottleneck.

This problem can be avoided with the use of data formatting that allows creatives to make as many changes as they need to without involving programmers – providing freedom to produce better output without increasing software development requirements.



qrin.uk/cs/1

Watch and Observe:

Identify how the Monkey Island game is defined through formatted data with consideration to:

- Visual content.
- User interactions.
- Story content.

How would the game code allow each of these to be presented and influence each other?

On a technical level, there are a few differences between features of programming and formatting languages:

- Logic
- Number operations

The logic and number operation abilities of a programming language allow for less literal approaches to creating functionality and content descriptions. While formatting is mostly limited to defining outcomes to match specific conditions, code allows for advanced logic, algebra and other maths concepts to interrogate and manipulate data in ways that produce a wider range of outcomes from minimal definitions. These advanced features allow code to produce results with higher efficiency and flexibility than is possible with manual definitions of formatting.

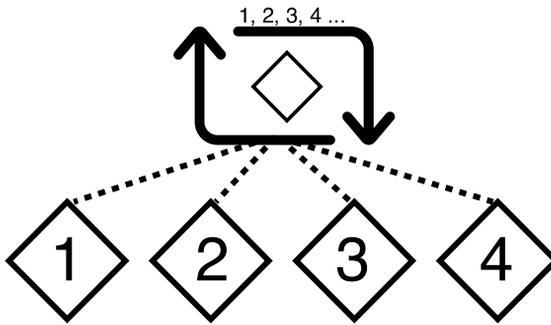
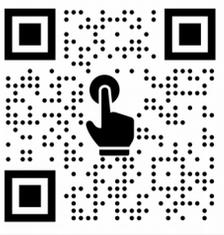


Figure 1.12: Ability of code to efficiently replicate content and data.

It's important to recognise that these differences are not an issue of deciding whether to use code "or" formatting, but more about how to use code "and" formatting to complement each other.

 <p>qrin.uk/cs/7</p>	<p>Interactive:</p> <p>Define settings for presentation and functionality using formatting and programming rules.</p> <ul style="list-style-type: none">• Set presentation with formatting controls.• Define functionality rules to affect output.
--	--

Conclusion

Creating software systems is a challenging task, so learn to avoid making the job harder than it needs to be. You can do this by following the no nonsense principles – keep it simple, keep it real, don't repeat yourself, and realise that elegance of code is always secondary to its functionality. The aim of these principles is to focus effort to be invested on achievable objectives, using the minimum amount of work required for current development and future maintenance – it's good to be strategically lazy!

Technologies available to help with the production of code have been improving ever since the early days of computing technology. From programming

languages that are easier to learn and understand, to the rise of frameworks, code editors and visual design tools, the available options for defining software systems is continuously evolving. Learn to identify which technologies are suited to specific project requirements, while also keeping your skills up to date. Programming technologies may change, but skills for analytical thinking and understanding of software architecture design are transferrable to all software development technologies.

Finally, it's important to be aware of the changing nature of commercial software projects. This type of work is very different to personal or academic coursework projects due to exposure to changing requirements and influences from different disciplines such as creative design, marketing and business management. Learn to write code that has flexibility to be easily changed, along with external controls that allow you to delegate responsibility for defining anything other than the core software functionality. The use of data formats and their associated authoring tools allow content creators to take responsibility for integrating their creations into the software system. By developing your code to provide creators with control of their work, you release yourself from being drawn into project politics and extensive involvement in supporting change requirements that become problematic.

Exercise

Identify a useful software application that you would like to create.

- Why is it useful? What problems does it solve?

What layer would the application be placed within? Why would it be created for this layer?

What data is required for the application?

What functionality would be required for the application?

How would the data interact with the functionality?

How would you use formatting to benefit your application?

What situations would you want to use code to replicate data and content?

Did you like this?

Register for the free No Nonsense
newsletter:

Tutorials

Tips

Challenges

[Click Here](#)

www.nextpoint.co.uk/register/programming-newsletter/

“ **NONSENSE** / n. Spoken or written words that have no meaning or make no sense. ”

The **NO NONSENSE** series is designed to present concepts of high complexity in an easy to understand format without the need for an academic or technical background.

PROGRAMMING CONCEPTS focus you to build a solid understanding of programming theories and their practical application. The book supports you in developing the mindset needed for progressing to learn a specific programming language.

Using a simple step by step approach that's supported with visual aides, relatable examples and interactive exercises, advanced programming concepts are presented to be simple to follow and easily remembered. Boost your confidence by learning the simplicity behind programming that is often over complicated by the pretentious language used by computer boffins.



STUDENTS

Make sense of coursework and exam topics.

LEARN FASTER | BOOST GRADES | REDUCE STRESS



TEACHERS

Easily explain advanced programming concepts.

BETTER ENGAGEMENT | EXCEED PERFORMANCE TARGETS



PROFESSIONALS

Develop knowledge for commercial projects.

COMMUNICATIONS | AWARENESS | PLANNING



HOBBYISTS

Learn concepts that will improve your projects.

BETTER CODE | AVOID PROBLEMS

