Software: architecture, engineering: Survey

vishvAs vAsuki

September 17, 2012

Contents

1 Themes and techniques

For distributed computing, computer networks, software engineering, see elsewhere. Programming language concerns (including techniques like object oriented programming, functional programming etc..) are considered elsewhere. Architecture considers how code may be organized.

1.1 Engineering Process

How to come up with better processes to get teams to churn out more reliable software with fewer resources?

Part I

Art of Engineering software

2 The overall picture

2.1 Analogies

Programming is a matter of unambiguously telling the computer what to do, in a language it understands.
Programming is analogous to wiring a complicated electronic circuit: Joining wires of various modules is the right way.
Programming is analogous to designing a pipeline or circuit on the computer (as in chip design or in designers like rational rose, or in rapidminer or in lego robotics).

2.2 Understanding specifications

In the industry, this involves surveying customers, rapid prototyping, talking to end-users etc.. In research, the specifications come from the research problem.

3 Programming paradigms

3.1 Chapter Scope

Programming language features enabling different programming paradigms and architectures are considered elsewhere. Eg: Higher order functions underlies functional programming paradigm.

3.2 Imperative vs declarative programming

3.2.1 Imperative programming

Views computation in terms of altering state.

3.2.1.1 Structured programming

In this programming paradigm, state changes are restricted locally. This helps organize code and state information. Structured programming code is more easily understood and maintainable than common procedural language code. This includes object oriented programming.

3.2.2 Declarative programming

Describes what needs to be done, rather than how to do it Eg: saying list.map(fn) instead of a C-like for-loop. Includes functional programming, logical programming.

The ideal of declarative programming is for the programmer to be able to express computation in terms of computation of a mathematical function - despite internally using imperative programming and dealing with memory at a lower level.

3.3 Modularity With Object oriented programming

3.3.1 Encapsulation/ locality

One collates within objects various values describing object-state. Often nouns correspond to objects, verbs to method calls.
For the sake of clarity, one may separate member variables from methods, creating two separate classes : ClassA and ClassABehavior.

3.3.1.1 Member addition test

Not every entity manipulated by an object need be its member - one may for example merely pass it as an argument to the object's methods.
To decide if an object should have a certain member, one way could be to ask the question: when a deep-copy of an object is made, is it proper that a copy of the member is made?

3.3.2 Class relationships

3.3.2.1 Identifying shared code and state

The fundamental operation in deciding the mechanism for code and state sharing between two objects is intersection - one determines precisely the state and methods that are to be shared.

3.3.2.2 Sharing code and state

Sharing state and associated methods between between two objects can be accomplished in two ways: class inheritence, aggregation/ containment (and sharing) of objects. The difference is the one between 'is-a' and 'has-a' relationships.

3.3.2.3 Inheritence vs aggregation

If objectA contains all of objectB, inheritence may be appropriate. In case multiple inheritence is necessary but the programming language won't allow it, one may use aggregation.
Sharing state and associated methods between between two objects can be accomplished in two ways: class inheritence, containment (and sharing) of objects.

3.3.3 Add function interface to classes

Called Type-class pattern because generic/ type-parametrized classes are used to solve the below problem.

3.3.3.1 Problem

Suppose that one wants to define a function f whose arguments include objects of classes C and D.
Suppose further that we want to define generic function g, which accepts objects of all classes with f - without need of alteration as new classes with f are defined.
Further, we want calls to g to be type checked at compile-time.

3.3.3.2 Inheritence

A common way of implementing this is to add methods $C.f$ and $D.f$; and perhaps having them extend an interface $hasF$.
But, this may not be possible (because you don't have access to the source code of C or D). Also, for various aesthetic reasons, you may not want C and D to be 'coupled' through an interface.

3.3.3.3 Generic function use

One can alternately define a generic function: $f[T]$, which can behave differently based on the class of its input - whether it is C or D. This feat is often implemented using 'pattern matching' in case of algebraic data types.
The disadvantage of this is that type-checking usually happens at run-time, rather than at compile-time.

3.3.3.4 Creating concrete fn objects with generic interfaces

One can create a trait/ interface $hasF[T]$ where the function f is defined, and create objects $C1$ and $D1$ which have the traits $hasF[C1]$ and $hasF[D1]$.
Then, $g[T]$ can be defined to accept $hasF[T]$.
If the programming language defines an 'implicit scope' whence arguments to functions may be drawn, one does not even need to pass these function objects explicitly.

3.4 Functional programming paradigm

3.4.1 Introduction

The main themes in functional programming languages are higher order and partially applied functions and functions without side effects.

3.4.1.1 Facilitating parallel programming

Absence of side effects (ie use of immutable data structures) eliminates problems such as race conditions arising out of shared memory.
Higher order functions - particularly those defined on lists - makes parallel programs easier to express.

3.4.1.2 Computation as a vector function

Let D be the set of all scalars. Any computation can be specified as a vector function $f : D^{k_{in}} \rightarrow D^{k_{out}}$. We name the arguments to f as $x_1, x_2..x_{k_{in}}$.
This function can in-turn be defined in terms of other simpler functions - using currying for example.

3.4.2 Monad

A monad is a generic object/ value which contains within it another value v of type t. Apart from monad construction and ability to get the contained value, it contains the following basic functions:
map(f), which applies f on v and returns the resulting monad.

3.4.2.1 None monad: error handling

This construct is very useful in error handling: in case of an exception one can return a 'None' monad, for which map(f) has no effect.

3.5 Exception handling

Exception handling features which require special language features are explained in the programming languages survey.

3.5.1 Error codes

As in case of C, one may cause a function to exit abnormally in case of an exception, notifying the caller of the case by returning a special value, and setting a particular flag/ message in a memory location.

3.5.2 Monads vs exception throwing

Using monads is cleaner conceptually in its ability to write the program as a series of function operations. The resultant code is also more concise compared to the exception throwing paradigm.

3.5.3 Advantages

3.5.3.1 Conciseness

Very concise: fewer lines of code.

Fewer bugs.

3.5.3.2 Highly parallelizable

For functions without dependencies, order can be reversed, plus there are no side-effects: Thread-safe.
So likely to become popular.

4 Correct coding

4.1 High level strategy

4.1.1 Modularity

This is an excellent technique for dealing with complexity in software construction. Its benefits are described elsewhere. Use it to break up code into small easily codeable, provable and testable pieces.

4.1.1.1 Separation of concerns

Each module may provide certain services and may inturn require services provided by other modules for their realization.
However, the internal workings of any module is, to a great extant opaque to other modules.

4.1.1.2 Benefits

The greatest benefit of modularity is the ability to reduce complexity in designing, analyzing and implementing complex pieces of software.
Also, it facilitates code reuse, and Fewer lines of code imply fewer bugs.
The abstraction/ hiding of inner working of modules also enables a high degree of mutability: Modules may be rewritten, tuned for different conditions freely as long as they provide the agreed-upon services.

4.1.1.3 Service Layers

In case module A depends on the services provided by module B, but not vice-versa, module B can be considered to occupy a lower level/ layer in a software/ service stack.
Examples of use of this powerful simplifying technique: designing computer networks' protocols, operating system design, compilers etc..

4.1.1.4 Organizing code

Clearly understand what functions are available, where they are likely to be used etc..

You can organize code using OOP ideas even if you use functional programming: use namespaces.
If you have state info in various objects, use access specifiers well: don't provide public access unless you have to.
Also use nested functions where possible!
Refactor often.

4.1.1.5 In implementing a monolithic function

Also, make use of anonymous/ nested functions to neatly chop up the logic which goes into a function!

4.1.2 Code reuse

4.1.2.1 Beware of redoing others' work

Try hard not to write code when others have already done that. Debugging takes time. Eg: Tempted by P's suggestions that it is easy to implement, I wrote matlab version of block coordinate descent algorithm described in the van-greer paper; but debugging it took much time. Eg2: wrote cross validation code, only to discover that it is part of the matlab statistics toolbox.

4.1.3 Conciseness

Absence of boilerplate code is desirable - less clutter. The important logic of the code should stand-out, parts of code dealing with trivialities like null-checks and exception handling should be very concise and non-dominating.

4.2 Module-level strategy overview

Write down an implementation suitable for the language being used.

4.2.1 Preconditions

Don't write code in computer language until you are clear about the logic. You should be surprised if the program fails.

4.2.2 Activities

Do this cyclically: coding attempt, systematic code review, run tests.

4.2.3 'Correct conception' vs 'Debugging until correctness'

Usual case: It is difficult to write perfectly correct programs at the first attempt. Often, most of the time is spent in debugging code, rather than in conceiving its logic.

Maybe this does not have to be the case. If more time and care is spent in correctly conceiving the program, less time is spent in debugging.

4.2.3.1 Choice

The former activity is slightly slow (to the point of being boring) while latter is frustrating. However, the former is the lesser of two evils, and must be chosen. Don't stray too far from the pen and paper. Increases clarity, increases enthusiasm.

4.2.4 Changes to old code

When ever you make a change to old code, follow the 'correct coding' protocol again. Be very sure to systematically review the altered function to ensure that it works as intended. If possible rerun test.

4.3 Specify behaviour perfectly

Specify the domain and range of the function perfectly.
Specifying the behaviour of a function on various inputs and contingencies can drastically reduce negligence.

4.3.1 Check input validity

Include checks for calls with invalid input. This is facilitated by programming language constructs called 'assertions'.

4.3.2 Module comments

Comment on various state variables, where they are altered etc..

4.3.3 Function comments

Include the following parts: Description, Syntax, Input, Output, Examples, References, 'see also'.

4.4 Systematic code review

Critique the implementation to check faithfulness to the logic.

4.4.1 Very important!

Don't run/ test the code until you are sure that the logic is perfect, if not the syntactic expression.
Don't be sure that the implementation is correct until you have systematically reviewed the functioning! This is because, often, it is not possible to design a complete set of test cases.

4.4.2 Over-Confidence bias

This is best done with a fresh mind, rather than immediately after the coding attempt.

4.4.3 Tracking confidence

One's confidence in each function's implementation should be carefully tracked - perhaps through comments of the form 'Confidence in correctness: High. Reason: Tested multiple times'.

4.4.4 Prove correctness

Then write down the logic in detail. Use safety and progress properties to reason about algorithms: particularly the former. You should be surprised if the program fails.

4.4.4.1 Write statements to prove

Simply writing down the statements to be proved and arguing at a high level why they are correct yields major results.
Demands of correctness often correspond to invariant propositions involving certain variables.
[**Incomplete**]

4.4.4.2 Axioms for external functions calls

State your expectations of the functions you are calling.

4.4.4.3 Thoroughness vs Time required

This can take a long time : especially when recursion, distributed computing or other intricate logic is involved.
This is not necessarily a trivial task, even when the logic, at a high level, seems simple. Eg: Find the path in a tree to a leaf with a certain property, while implementing pruning to avoid visiting leaves unnecessarily.

4.5 Test

4.6 Desiderata

Ideal traits of tests are that they should be fast, independent (failure of one should not affect another), failures should be informative.
A test consists of various assertions. Several tests are grouped under test programs.
A test framework should provide simple syntax to specify tests.

4.6.1 Isolating modules

Modules may need to be isolated for testing.

4.6.1.1 Fake

Fake objects have working implementations, but usually take some shortcut (perhaps to make the operations less expensive), which makes them not suitable for production. Eg: In memory database.

4.6.1.2 Mock

Mocks are pre-programmed with expectations of the calls they are expected to receive.

4.7 Debug

Be a detective! Understand the bug, solve it in minimum time. Use tools such as debuggers which allow 'stepping into the code', loggers etc..

4.7.1 Syntactic errors

These are the easiest to resolve. But, in case of poorly documented / new language features or in case of complicated instructions, these can be time consuming.

4.7.2 Semantic errors

4.7.2.1 Verify input validity

If some function call does not go through, first verify that arguments which are being passed are not weird.

4.7.2.2 Take a step back

It is best to return to 'design and prove correctness' stage of program implementation and do everything that follows. It is important to know when to step back - do not rush with this!

4.7.2.3 Logging

Include logger statements at critical points in functions, ye should be able to control the logging level.

4.7.3 Errors in memory/ process management

Make good use of debuggers.

In case of memory leaks, the computer may end up allocating an ever increasing amount of memory to the program until it runs out of resources.

5 Speedy coding

Coding speed, in most applications, should not come at the cost of errors.

5.1 Syntax references

Make and use surveys / examples to easily lookup language syntax and functions.

5.2 Choosing programming languages

One must use the language most appropriate language to solve the problem at hand quickly and with computational efficiency.
Sometimes, a tradeoff may be involved: tradeoff between expressiveness and efficiency of a language. One may even use different programming languages to specify different parts of the logic to the computer.

5.2.1 Expressiveness/ conciseness

Expressiveness is important in rapid prototyping/ programming.
Writing linear algebra logic in Java or C, for example, is far more tedious and cumbersome than using Matlab for the purpose, even in the presence of some good library functions: You end up worrying about low-level things like data representation and casting, rather than the linear algebra.

5.2.1.1 Iteration

Much of programming involves iteration.
Common operations and transformations on collections like Lists, Maps etc are much more efficient in functional languages like Scala than in C or Java. Idioms like map, filter, find, groupBy, indexWhere, zip, foreach reduce programming time significantly.
Freed from having to keep 'book-keeping' specifics in mind while programming, the programmer is happier to write code which considers all corner cases (rather than take shortcuts and state known bugs). The programmer can focus on algorithm-level optimizations and be more creative in general.

5.3 Complexity

More complex code is harder to implement : sometimes, there one can make choices which lead to simpler code - by simplifying requirements or interface languages for example.

5.3.1 Cyclostomic complexity

THis is the number of edges in a representation of a program as a flow chart. The more conditional and looping statements there are, the greater the cyclostomic complexity.

5.4 Maintainability

Constructs may need to be changed or extended in the future. Code written in the past should be easy to understand.

5.4.1 Comments for enabling understandability

So make comments and documentation copious.
When declaring (class) variables, write comments indicating purpose, desired properties.
When a particular design choice is implemented, comments may be included to justify the choice.

5.4.2 Naming conventions

Use good naming conventions for all identifiers. Use meaningful names; make it easy to read with capital letters at word beginnings: listIterator.
Macros and constants in CAPS.
Classes starting with capital letter. methods, objects, packagesMemoir start with small case. Maybe variable names prefixed to indicate data type.
In case of collections/ namespaces of multiple objects of type Str, the Java convention is to use the name str, rather than strs.

5.4.3 Version control

Use version control or just dropbox. [**Incomplete**]

5.4.4 Coding conventions

Include sequence: Be alphabetical. In case of C++: mainHeader.h, c system, c++ system, other.
Terminate namespaces with // namespace video

5.5 Automate tests

In case of code to be shipped: test every function; make test scripts.

5.5.1 Externalizing messages

Message strings should, in most cases, not be hard-coded inside the code. Instead, message strings should be externalized; perhaps be returned by a function when provided with an appropriate key.
This separates the program logic from the messages cleanly. One advantage of doing this is that these messages can be changed without touching the program logic.
This is also convenient when the program needs to be made available in several different languages.

5.6 For research

5.6.1 Have a good library

Don't rewrite the same logic multiple times: put it away in functions.

5.6.2 Experiment functions

Make a function out of each experiment.
Clearly make program variables for all parameters involved in the experiment. You should not have to comment or uncomment code to run different variations of the experiment; you should just set the experiment parameters as necessary.

5.6.3 Automatic logging

Corresponding to each experiment, create a log file with a time stamp automatically, perhaps of the same name.

6 Writing efficient code

Put theoretical analysis from algorithms and data structures to good use.

6.1 Optimizing code for speed

Use a profiler to detect parts of the code which are slow. Use libraries optimized for speed.
Avoid repeated memory allocation.
Avoid writing loops in interpreted languages without 'just in time compiling' facilities like Matlab: use a compiled language like C instead: otherwise inter-

pretation costs for executing the code piles up.

7 Group programming strategy

7.1 Processes

Hold code reviews.
Use a bug/ task management system.
Have major and minor code-releases.

7.2 Pair programming

Advantage is that you learn from your partner. Programming is more social, enjoyable; enthusiasm is maintained.

Part II

Special software

8 Computer networks

Considered in a separate survey.

9 Operating systems

Act as an interface between the various software jobs and the hardware.
Tasks include scheduling, memory management, disk management, handling i/o etc.

9.1 Scheduling

Swapping: must remember location of current instruction, memory contents etc..

9.2 Threads

There are threads/ processes. States: ready to be executed, executing, blocked: waiting for resource or a signal, terminated.

9.2.1 Coordinating processes

See distributed computing ref.

10 User interfaces

10.1 GUI Metaphors

The most common graphical user interfaces use the desktop/ workspace and the windows metaphor.

10.1.1 Desktop

A desktop/ workspace corresponds to a big area in the available screen space whence applications are launched, and within which statuses, widgets and windows are displayed.
There may be multiple workspaces on a device - especially in case of devices with limited screen size.

10.1.1.1 Layering

Various windows, widgets and the desktop conceptually form a stack of various displays available to the user. This concept is useful when understanding widgets.

10.1.2 Taskbar, statusbar

A narrow portion of the screen is sometimes dedicated to displaying the statuses of various running applications, outputs of widgets displaying info like time, weather etc..; and to provide place for menus to launch applications.

10.1.3 Windows

A window is an area of the screen which is dedicated as an input/ output interface. A window may be overlaid with another, or 'minimized' to have 0 screen area - and the corresponding area available for interaction with the window is correspondingly reduced. A window may be fully closed, possibly resulting in the termination of the corresponding application.
In case of small desktop workspaces (as in pocket computers), the active window occupies all of the available area on the workspace.

10.1.3.1 Task switching

Switching between windows is done using a program called 'task manager', which is generally invoked using the taskbar or special key combinations.

10.1.4 Views and perspectives

The Window is designed to be a collection of various views. Eg: Eclipse, RapidMiner. Perspectives are particular arrangements of certain views. UI's often allow (re)definition of perspectives by moving or adding views.

10.1.5 Widget

A widget is a window which is always on the layer immediately above the desktop. So, whenever one views the desktop, the widget is necessarily seen on top of it. Unlike windows, they usually don't have simple means to close them.

10.2 GUI Software Stack

Graphical user interfaces are implemented using a stack of software. The software stack, with examples is as follows:

- Windowing system: provides abstraction from hardware, provides graphics primitives, basic window drawing abilities. Eg: xorg server.
- Login session manager. Enables selection of desktop environment. Eg: gdm, kdm, lxdm, simpledm.
- Desktop environment. Eg: gnome, kde (programmed with widgeting toolkits gtk+ and qt respectively).
  application ivocation: mouse driven or keyboard driven (possibly with autocompletion).
  transferring copied content, drag/ drop.
  Status panel showing useful updates like time, weather, input language.
- Window managers. Eg: gnome-shell, compiz.
  Responsible for: Working with multiple windows, switching between them, positioning them (possibly in different workspaces).

10.2.1 Comparison

Keyboard shortcuts for the above are jointly provided.
Various GUI software stacks differ in the facilities they provide. The penalty for greater facility comes in terms of memory and processing power used to merely provide the desktop environment.

10.2.1.1 Look vs utility

There is additional cost due to fashion - newer, heavier window managers are provided not just to provide greater facility, but to provide alternative looks favored by the hip crowd which is often not dense with power-users. Also, newer window managers could be experimental or unstable in many regards - eg: Ubuntu Unity in 2011.

10.3 Special input hardware

10.3.1 Buttons

Buttons are often provided, dedicated for a simgle (possibly broad) purpose. These include buttons to power off, adjust volume, reset the computer etc.. They provide a useful, cheap ever-present way of accepting input from the user, which is partly why they are not located on the screen.

10.3.2 Dials

Dials - simulated or otherwise - are very useful for accepting numeric input; and are therefore provided in case of clocks, timers, etc..

10.4 Special indicators

10.4.1 Lighted icons

Sometimes lighted icons are provided to provided binary information about something. Eg: is the caps-lock on, is there a waiting notification.

11 Logging systems

Examples include various java logging frameworks including log4j.

11.1 Requirements

One is to be able to log at various levels, usually named debug, info, error etc..
Log messages should be informative and customizable: printing things like the name of the function where it originates, the class-name, the line number, the time, the log-message level etc..
Logging is to have a simple, yet expressive syntax for constructing messages.
One is to be able to set logging levels for various modules at various levels, so as to filter out relatively unimportant messages out of the log.
One is to be able to direct the log messages of various modules to arbitrary files, web-services or consoles.

12 Domain specific languages (DSL)

12.1 Objectives

The objective is often to define a special, simple language for a certain domain so as to provide a linguistic user interface usable by non-programmer domain-experts. This could also be an intermediate step for programmers ultimately providing other interfaces - a paradigm called 'Language oriented programming'.

12.2 Compiler implementation

These are often written using high level functional languages like Scala, Groovy which let one define binary operators/ relations, easy type conversions, ways to dynamically handle arbitrary class members, syntax for symbols etc.. So, one actually writes DSL code which also makes sense in that high level language; and this code is interpreted suitably due to exploitation of the aforementioned features.
Binding symbols is often accomplished using hashmaps.

13 Compilers

[**Incomplete**]