Software Architecture for Real Time Systems

Chris Gerth - FRC1736 Robot Casserole

October 29, 2016

Abstract

A general approach to structuring software is presented. This approach is common in industry for the organization of real-time, complex mechanical systems. It suits itself well to controls-algoithm based computation and state machine implementation. Commonly used software tools are discussed.

Part I
What Software Architecture Accomplishes

1 The Need for Good Architecture

1.1 Basic Needs

The writing of software can be analyzed on many levels. Aside from broad implications of programming language and target execution environment, "functionally correct" software can be implemented in many ways. To say a piece of software is "functionally correct" implies only that it meets the minimum standards for today's requirements. There is no thought to what sort of changes would be required in the future, or what parts of software need to be touched for future changes. This is the primary gap software architecture attempts to solve - amongst the miriad of acceptable "functionally correct" implementations, adding further architectural constraints is key to creating software which is easy to edit, and robust when altered.

1.2 Added Benifits

A properly architected software system bestows many benifits on the maintainers of the codebase. Aside from being pleasing to look at and simple to

understand, software changed can be made with great confidence. Good architecture reduces the number of unexpected side-effects any given change will have. This leads to simpler testing, and faster development.

2 Discovering What Architecture Is Useful

Under the reasonalbe presumtion that changing software will be inevitable, it would behoove the software's creator to spend extra time up front to analyze the sorts of change possible.

2.1 Immutable Requirements

Some things are relatively immutable - target architecture, major functional components, or number of active users are often fixed for real-time systems. Constraints for code size and execution time are unlikely to be lifted. Considering these constraints as never-changing is largely acceptable because changing them would have broader implications to the system as a whole. If a major component is added or removed, there would be enough other changes occurring to necessitate time-intensive activity, which could easily include a re-write of the codebase. Re-writes imply large architecture tear-up, and extensive re-testing. Due to this larger effort, it is less likely to be undertaken. As will be discussed later on, carefully deciding what requirements will be immutable can help reduce the complexity of the implementation.

2.2 Mutable Requirements

However, finer details are often highly mutable. For example, the exact RPM of a rotating member needed to launch a ball is likely to be tweaked frequently. So frequently, in fact, that having a mechanism for altering it without chaning the software source code would be desireable (this is referred to as a "calibration" and will be discussed later). Less extreme examples of mutable requirements might be the type of sensor used to detect a particular condition - the input style changes, but the information acquired is the same.

2.3 Velocity versus Architecture

Architecture is often viewed as a extra layer of complexity to deal with while writing software. Indeed, it can slow development at first, but a more holistic view is more approprate.

2.3.1 Extereme Velocity

The fastest way to develop new software is to have extremely smart people type code as fast as they can, with as few distractions as possible. Psychologists refer to this state as "flow[1]," where the brain is actively making design decisions both

[1] https://en.wikipedia.org/wiki/Flow_(psychology)

conciously and unconciously. The whole of the thought process is devoted to both design and implementation simultaneously. Large portions of the relevant functionality are considered at the same time. Little thought is given to organization or architecture - when something is deemed as "needed", it is simply inserted wherever most convienent. All efforts are on actually writing as much code as possible - architecture and testing are cast aside as much as possible.

For many single-developer projects, this is the way to work. When unimpeeded by external constraints, and the functionality is small enough to be considered at the same time, this is indeed an exteremly efficent way to write software. As one develops his or her own side projects, or works on homework assignments, one will likely have great success with this method.

This method of exterme velocity starts to break down when the scope of the project expands. Codebases get larger and more monstrous, more developers start working on the project, or requirements change and require tearup of code that hasn't been touched in a while.

Many fast-paced companies get around this through the "pivot" - an industry term for completely tearing up what you have and quickly writing something completely different. However, for real-time, safety crucial systems, we cannot simply rely upon the wits of individual devepers to make the right choices all the time in a velocity-crucial situation.

2.3.2 Extreme Architecture

When safety becomes of paramount importance, individual developers can no longer be trusted 100% to guarntee their work is correct. Many approaches can be taken to guarnteeing correctness. Especially in certain industries (like aerospace or medical equipment), software structure, validation, and design become far more important than actually writing the code itself. Many more hours are put into architecting software such that safety standards are obeyed, and critical performance criteria are met. Changing even a single line of code is a massive effort, requiring vast documentation, possibly months of approvals and testing. Despite the tedium of such efforts, they are industry-standard accepted best practices when software is used to control systems that could potentially kill people. Various safety regulation bodies (ISO, for example) are tightning the restrictions on how software can be architected for these systems.

There are many cases where this sort of extreme architecture is required. However, it does come with a large cost in development time. Code bases simply cannot be changed rapidly. Due to this, inapproprately applying this level of architecture to systems which do not need it will only impeed progress.

2.3.3 A Happy Medium

For most forms of software development, a middle ground is desired. Software architecture must be constraining enough to aid development, but not so constaining as to impeed progress. How this is achieved is discussed somewhat in this document. There are even degrees you can get in Software Engineering

which teach you how to do it. Even still, defining an excellent architecture is a tallent, perhaps even a form of art. The redeeming aspect is that even average architectures still provide benifit to developers.

2.3.4 Playing for the Long Game

When first starting off a large software project, maintaing a consistant architecture will often seem tedious. New files, extra interfaces, moving content around - all will seem as a impediment to getting to a goal when a more straightforward solution is possible. However, by properly constraining development activities from the start, the project will grow in a much more managable way. There is a definite inflection point where the pain of a single change is huge for the poorly architected project, but much easier on the well architected project. Stay strong through the first couple hundred lines of code development, and your efforts will pay off during the next thousand lines of development.

Part II
What is Architecture

Software Architecture takes on many forms. For the purposes of this paper (FRC robotics, and similar real-time systems), the focus will be on dividing up functionality into subsystems, and defining the interfaces between those subsystes.

3 Division of Functionality

3.1 Definition and Purpose

A fundamental step of defining a software architecture is being able to break up the different pieces of required functionality. The robot is not a monolithic machine, so neither should the codebase be without divisions and boundaries. This is done so that each descrete piece of functionality can be (as much as reasonably possible) be thought about on its own, without considering the whole system.

3.2 Functional Subsystems

The first step in a good software architecture is sub-dividing the total content to be implemented into subsystems. If the whole robot is considered a "system", then its subsystems would be the physical coponents that make it up. Usually these are the natural divisions that arise from unique pieces of functionality. For example, on a usual robot, one will have different subsystems for the drivetrain,

the manipulator arm (or arms), the attachements to that arm, the intake mechanism, etc. Since this high-level division of functional content is present on the physical robot, it would logically make sense to group the software similaraly.

3.2.1 State

Each functional subsystem will need to internally maintain a "state." State is the academic term for any relevant, time-varying information about the subsystem. For example, the state of a drivetrain could be described by the voltage applied to the motors, the motor rotational speed, and the current draw of the motors. There are mathematical relationships between state variables, sometimes impacted by external factors. An external factor (like a human driver) might command a desired state, and the logic controling the subsystem would have to push the subsystem toward that desired state. Sometimes doing this requires more advanced control algorithms (like PID), other times it just means mapping one relevant variable to another. Regardless, all controlled subsystems can be characterized by this state model.

Sometimes, it is useful to explicitly use this state model to name variables and describe interactions within the subsystem. Other times it just makes the code more complex. It is a judgement call of the developer to use this model or not. However, for complex subsystems, analyzing and organizing it in terms of its state is the most accepted way of tackling the problem of controling it.

3.3 Platform-Specific Vs. Pure Logic

In the state model of the world, we will often talk about things like "motor voltage" or "wheel speed". These are useful concepts for humans who have gone to school to use for conceptualizing what is occuring in their subsystems. However, for computers, it's all just a bunch of 1's and 0's. Where this frequently comes to a head is in measuring or actually outputting these physical quantities. Reading wheel speed actually requires a lot of wiring, circuitry, physical devices, complex computer hardware, and convoluted interface software. However, to the end user of the speed, they don't actually care about that.

This brings us to our next important division of subsystes - separating platform specific details from pure logic. Any easy-to-understand software will take human-readable signals as input (physically meaningful things, like speed or voltage), and output similarly human-readable values. Separately, there will be functional subsystems dealing with the conversion between human-meaningful value and the bits and bytes needed by the computing platform hardware to actually measure or realize a physical effect.

The FRC libraries do a great job of this already - Datalink and FPGA details are hidden behind very nice methods such as "getVoltage()". For this reason we don't often have to be too concerned about this sort of architectural decisions. What we do have to watch out for is the fact that these hardware-access methods take a long time to execute. Many times longer than pure logic. Therefor, we

should be careful to only read physical values just as frequently as needed, never more.

There are similar conversations to be had when it comes to interacting with operating systems, or other execution-environment specific details. That, however, is beyond the scope of this document.

3.4 Software Specifc Subsystems

In addition to the physical subsystems present in the robot that the software must control, other software-specific subsystems are available and commonly used. For FRC robots, these include:

- Driver Feedback (website or smart Dashboard)
- Calibration interfaces
- Evnet and signal logging
- Many Others

While these will not be discussed specifically in detail, their architecure may be treated similarly to physical subsystems.

4 Interfaces

4.1 Definiton and Purpose

Once a codebase has been divided into unique subsystems, one must further consider how those subsystems will interact with eachother. Defining and constraining this interaction is the step of defining the software interfaces.