GEBZE TEKNİK ÜNİVERSİTESİ

ELEKTRONİK MÜHENDİSLİĞİ BÖLÜMÜ

ELEC 334

2020 – 2021 BAHAR DÖNEMİ

HW1

Öğrencinin

Numarası : 1801022071

Adı Soyadı : Alperen Arslan

Applying Test Driven Development to Embedded Software

Test Driven Development (TDD) is increasing in information technology applications and product development; however, it has not been widely applied in embedded software development. Embedded developers face many challenges. TDD can help overcome some of these challenges, but TDD has to be adapted for embedded systems development.

Refactoring is the activity whereby the structure of the code being worked on is changed without changing its behavior. This is done to clean up any messes that are made in the heat of battle trying to get something to work. Prior to executing this core cycle of TDD, the developer establishes the context for the work being done. When the developer decides to work on a specific module, that module has a set of responsibilities and behaviors that allow the module to provide what is needed to the embedded system.

Unit Test

TDD can be applied as unit tests and as acceptance tests. Unit tests are tests that provide feedback to the developer about whether the code written does what it is expected to do. Unit tests are behavioral tests and attempt to fully exercise the module under test. Ideally, unit tests are run every few minutes with every code change. This can be a significant challenge when we have long compile and download times. Each module has a suite of unit tests. They also provide regression tests and relieve the programmer from having to test the same code repeatedly manually.

The Target Hardware Bottleneck

Use of evaluation hardware, sometimes called eval boards, is common in the industry. An eval board is a development board with the same processor configuration as the target system and ideally some of the same I/O. Sometimes eval boards and target systems may be too expensive to have one dedicated to each developer, or they may have inferior or expensive debug tools.

Tailoring the Cycle

Keep in mind that the incremental code and test cycle needs to have a short cycle time to keep the developer in the rhythm. If the build and test time starts to get longer than a minute or two, partial builds should be used to keep the code and test cycle short.

Hardware Is Not Available

Testing in the development system is an important accelerator for projects because the development system is a proven and more stable execution environment. It often has a richer debugging environment than the target, and each developer has one or can get one tomorrow. The development system is the first place to run any hardware-independent test.

Testable Design

To make embedded software testable, you must isolate the hardware dependencies by designing the embedded software for this purpose. Software development starts from the inside with the solving of the application problem and then works its way to the outside, where application code meets the hardware. Interfaces should be designed to describe how the core application interacts with hardware-provided services. The isolation approach is not new; it simply comprises the application of the well-known practices of abstraction and encapsulation and the separation of concerns that are essential for TDD. This technique is independent of any computer language or tool; it is simply a matter of applying good modular design practices.

Trends in Embedded Software Engineering

The increasing complexity of functional and extrafunctional requirements for embedded systems calls for new software development approaches. Over the last 20 years, software's impact on embedded system functionality, as well as on the innovation and differentiation potential of new products, has grown rapidly. This has led to an enormous increase in software complexity, shorter innovation cycle times, and an ever-growing demand for extrafunctional requirements—software safety, reliability, and timeliness, for example—at affordable costs.

Embedded Software

In addition, embedded systems developers require extensive domain knowledge. For example, developing a vehicle stability control system is impossible if you don't understand the physics of vehicle dynamics. Consequently, embedded software development has been restricted mainly to control engineers and mechanical engineers, who have the necessary domain expertise. With the rapidly growing complexity of embedded software systems, however, many companies have run into software quality problems. Supporting seamless cooperation between domain and software development experts to combine their complementary expertise remains a core challenge in embedded software development.

From Programming to Model Driven Engineering

Managing the rapidly increasing complexity of embedded software development is one of the most important challenges for increasing product quality, reducing time to market, and reducing development cost. MDD is one of the promising approaches that have emerged over the last decade. Instead of directly coding software using programming languages, developers model software systems using intuitive, more expressive, graphical notations, which provide a higher level of abstraction than native programming languages. In this approach, generators automatically create the code implementing the system functionalities. To manage embedded systems' growing complexity, modeling will likely replace manual coding for application-level development—just as high-level programming languages have almost completely replaced assembly language. Following the shift from assembly language to high-level programming languages, and from programming to model-based design, comes the shift from MDD to domain-specific development10—a shift already on the horizon. Some industry case studies have already successfully applied domain specific development. MDD is based on general purpose languages and code generators for different application domains. This claim for generality contradicts the optimization of the language and code generators to a given application context, and to the hardware platform used. Domain-specific modeling lets developers use domain-specific concepts, so it provides even more-intuitive modeling languages and integrates more software developer know-how and application-specific optimizations into the code generators.

Quality Assurance of Safety Related Systems

Before code generation, developers can apply static and formal verification techniques to ensure software quality; after code generation, they can use dynamic-testing approaches. Regarding quality assurance, dynamic testing is still a widespread approach for determining the correct functioning of software and systems. However, only formal verification techniques can achieve complete correctness proofs. Still, every formal technique has disadvantages. Applying formal techniques to a complete, complex software system isn't possible. Moreover, formally proven software might still be unsafe, and safe software might not be completely correct. Correctness clearly supports safety, but it doesn't substitute for safety analysis.