

Floating Icebergs

Improving the Application Development Process with Codefree Executable Models

By Paul Warren (BSc Hons)

What is it about software applications that make them so difficult to build and maintain? This white paper sets out to answer this question, and proposes an alternative strategy that could yield significantly better results.

Problems with existing strategies

A number of theories have been put forward as the reason for software projects failing to deliver the required functionality, overrunning on time, and exceeding their budget. The following section explores some of these issues.

Why do developments overrun?

Collaboration

Wouldn't it be nice if a software application could be built in a quicker, more collaborative way? Most developments are a mix of business knowledge and programming skill. Such ingredients tend to come in separate packets. Significant re-alignment of disparate mental models is required to marry these worlds together.

The division of labour means that teams cannot be formed with flexibility of role. Distinct visions and solutions are imagined, with risky prospects for convergence.

Is it not surprising then that application development is frequently a protracted process, extending through a number of phases over an unquantifiable period of time?

What if the number of threats to any project were exponentially proportional to the length of time between identifying the need for an application, and that application achieving business benefit?

Missed Opportunities and Moving Targets

The development process would be more predictable if the problem seeking a solution were to stand still. Few business activities are void of uncertainty. Change happens. Herein lies a dilemma. If the target has moved, should you re-align your sights, or should you shoot anyway and hope that you were close enough to cause useful damage?

So-called 'Scope Creep' is cited as the cause of a number of project failures, but hindsight is a wonderful thing and we cannot know if the development would have been successful had it followed a more rigid path.

Blank Sheet Syndrome

It is easier to criticise than it is to design. Specifying cohesive requirements, without seeing them materialise in a real system is difficult. Regular feedback is vital to confirm understanding; indeed this is a fundamental part of the process. It is well documented that late changes are more costly to implement.

The Leaning Tower

Software systems are an amalgam of inter-dependent components.

Consider a typical arrangement. A database is used to store the data: queries and stored procedures interact with the database to provide the business logic: the user interface accesses the middle tier. This layered approach is favoured for its modularity and ability to change, however, the consequence of stacking components in this way is that the whole structure becomes unstable. Changes to the foundation layers can have a detrimental impact on other layers.

String and Sealing Wax

Software resembles an iceberg. The visible bit may well: behave as expected, pass the required tests, and be exactly what you ordered, but the bit below the surface that you can't see?

Software Engineering is also unique. Unlike other engineering disciplines where it is obvious when a product fails to meet its design brief, a computer program can function despite being built from dysfunctional code.

If you could see this bit of the iceberg, the bit below the surface, you would have much more confidence that the whole design meets its specification.

Resistance to Change

Having overcome all of the hurdles, and created an application that is delivering business benefit, there will inevitably be requests for change. This is why software is soft! You may have resisted the temptation to change the scope of the project until now, but cannot continue to ignore the pressure for change. What challenges lie ahead?

The Fear of ‘breaking it’

Unless the application has been designed to accommodate such changes, there is a significant danger of breaking a working system in pursuit of new functionality. This is the concept of software entropy, where starting again is advocated rather than risk modifying a working solution. Assuming you do attempt change, how much testing of the original features should you do when writing new code?

Different dynamics come into play when a live system is being modified. Different skills, a different mindset, and a different attitude to risk.

Live data distinguishes a production system from a development prototype. Data needs to be protected from the intrusion of new functionality, it must be modified to accommodate new requirements and manipulated to fit new structures. This is potentially a traumatic exercise.

Environmental Changes

Requests to run the application on a different server, in a different location, on a different platform or shared by a different group of users can all equally scupper a working application. Server and platform portability may not be an issue if this has been considered at the outset, but most applications do not migrate easily from desktop to web, for example.

The Toolkit

The pitfalls surrounding the development and maintenance of software are well documented, so what are the best techniques for avoiding them?

The Mythical Man-Month

Consider the project management triangle of Quality – Cost – Time. This suggests that the project can be accelerated, and still deliver the same quality, by increasing the resources available. This comes at a cost, but may still be justified in the overall business case.

Unfortunately, it has also been shown that increasing the number of people working on a project can actually reduce productivity, due to the increased amount of communication necessary.

Nothing More Nothing Less

There are some who would argue that keeping an iron grip on the specification, and delivering exactly what is asked for is the secret of success. This approach may be preferable to a situation where the goalposts keep on moving, but does it really deliver what is required? Rapid Application Development and similar agile methodologies have their supporters, and they do support the vision of collaborative development without significant approved documentation beforehand. However exciting to read about, they are just too risky for most projects.

Re-use

The notion of re-using code or modules that have been developed for a previous project is commendable. The evidence of success is somewhat harder to find. Frequently parts do not fit, extensive co-ordination is required to engender re-usability, and there is a natural tendency for programmers to be suspicious of foreign code. Finally, re-use only provides returns when you have something to re-use.

Test, Test and Test Again

No one would argue with the principle of rigorous testing to confirm expected behaviour, however, there are issues that testing cannot resolve. Fear of failure is offered as justification for failure to test, whereas test failures should be embraced. Like customer complaints, they should be welcomed as a valuable resource, providing there aren't too many!

Much has been written about the cost of errors being greatest the later in the development cycle they are discovered, but there is also the psychological impact of testing blocking deployment.

Some teams or developers may also advocate releasing software that is not fully tested, and have users discover the bugs. The compulsion to do this can be well understood if your project is running late, and no further functionality can be added.

Whatever approach is taken to testing, testing alone cannot speed up the development cycle.

Object Orientation – Alignment with the Real World

Heralded as the Silver Bullet that would revolutionise software development, Object Orientation has had a significant impact. The most popular development frameworks support the use of classes, objects and methods. Certain aspects of OO do provide better visibility of the lower levels of the iceberg, with OO developments being largely self-documenting, but has OO contributed to faster developments?

Inheritance, encapsulation and polymorphism on their own do not provide significant gains in terms of development speed, but may be responsible for reducing errors and rework.

The greatest contribution that OO has to make is in the alignment of software objects with the real-world entities being modelled. The majority of software applications can now be considered in terms of a set of objects that are created, modified, and then destroyed. Even without a guaranteed methodology for creating appropriate classes, this concept is much more widely accessible. Disparate mental models are starting to align. The iceberg is emerging from under the water.

The Unified Modelling Language

The Unified Modelling Language (UML) is a major advance in terms of standardising the way systems are diagrammed. The problem is the term unified. 'Unified' is a relative concept, referring to the unification of various object-modelling dialects. What is needed is absolute unification, unification across the business and technical spheres.

UML may be able to resolve a number of conflicts amongst builders, arising from the misinterpretation of a system design, but on its own it does nothing for speeding up the development cycle.

Code Generation

If writing code is so problematic, why not get a machine to write it for you? This sounds promising, and you would expect a computer to be able to generate flawless code, but the code generator is a software application that suffers from the same pressures that we have discussed.

Combine modelling with code generation and you surely have a winning combination? There can only be one reason to *generate* code, and the reason is to enable it to be modified post-generation. There are many reasons why you might want to do this. The code-generator might not be all that good, or you might need to optimise parts of the code for performance. As soon as you edit generated code, you have broken the link with the model, and any gains will be lost.

Summary

There are clearly contributions that each of the above-mentioned initiatives can make to enhance the development cycle, but none of them on their own look capable of providing the step change in development productivity we are seeking.

An Alternative Proposition

The Achilles' heel of software development is the software itself, the code, the hidden part of the iceberg. If you could remove the code completely from a development, you can avoid a number of the pitfalls that beset present day developments, and significantly speed up the development process.

How can you have software development without code? Code is only a means to an end – the end being an application that enables users to enact the functionality they require. Program code is currently the most common way of achieving this outcome, but is it the only way?

To be accurate, we are talking about a development system with generic code but without user-written or machine-generated code.

Consider an Object-oriented system that understands what the application needs to do by reference to a model. The creator, who may or may not be a programmer, describes the application by configuring the model. The application's data is cocooned inside the model, where it can be better managed.

This approach, the Codefree Executable Model (CEM), has a lot of benefits that are discussed below.

Benefits of the Codefree Executable Model-Driven Application

Sceptics will be saying, 'you still got code so you still got a problem', however, there are some major differences with this approach:

- Code re-use is running at an all-time high, as the same code is used in every application, therefore, testing makes sense. The code base is common to all applications, irrespective of their domain.
- The key advantage of Object-Oriented development, real-world alignment, is exploited. None of the other features of OO are really necessary, except maybe inheritance, as there is no user code to protect. There is still a need for creativity to design enough classes for a comprehensive application, but getting started is much easier with the significant entities finding themselves aligned to classes without much trouble.
- Ultra-rapid development. For the first time, it really is feasible to create an application in a single sitting. This may be the creator trying successive solutions to find the optimal or a collaborative group organically growing an application bit by bit. All of the issues associated with traditional development over extended periods of time, evaporate.

- Ultra-rapid redevelopment. Application development is characterised by the iterative cycle of build test re-design, for the reasons outlined above. There are often many ways to implement a particular requirement, and it is frequently a case of what ‘feels’ right at the time. Such decisions are justified when later parts of the design get built or data is entered. This is the point that re-factoring needs to take place, in order to preserve the integrity of the design. There are undoubtedly programmers who have a better hit rate at ‘getting it right first time’, but every programmer will have faced this dilemma. You have to get it wrong first to get it right.
- Ultra-rapid redeployment. The conflict of many competing architectures, for example web delivery versus client-server desktop, is largely eliminated, as both can co-exist and concurrently interrogate the same model.
- The fragile stack of tiers that are so unstable when changes are needed is avoided. There is only 1 tier – the model – and this is the only artefact that needs to be updated. All else flows. The iceberg remains totally visible.
- CEM-driven applications are self-documenting. The model knows what is expected, and can present this information to the user in the form of structured help pages with minimal additional effort from the designer.
- There is no separate database to design, build and manage. Many codefree tools exist, that can understand a relational design, and present its data for manipulation, but the fragile stack still exists. In addition, there is not insignificant effort required to create a valid database design in the first instance.
- CEM means platform independence. Providing an implementation of the CEM interpreter is available for a platform, then the application can be deployed without modification.

Disadvantages of the CEM-Driven Application

It would be unbalanced not to consider potential disadvantages the CEM-Driven Application approach, however, as you will see the majority of these can also be considered as advantages.

- The designer will still need to encode algorithms and business processes within the application that cannot be represented purely by the static classes and relationships of the model alone. This is obviously essential to make each application do what is required. However, it must be stressed that this customisation, performed within methods within the Object-Oriented paradigm, does not detract from the basic operations of the static application.
- CEM-driven applications all look the same. This may be considered a disadvantage when branding and differentiation are important, but could be a plus point for consistency and user familiarity. For web delivery, style-sheets can offer local customisation, but when you are processing objects there is only a limited number of things you can, or would want to do.
- Having delegated a large proportion of your application's functionality to a CEM Interpreter, you will experience some loss of control. Unlike a compiled application, which exists as executable code after the compiler is removed, the CEM interpreter will always be needed. In this sense it is analogous to a virtual machine. For some organisations this loss of control may appear unacceptable. On further investigation though, they will realise that the unique parts of the application, the model and its scripted methods, would need to be constructed had they followed a more traditional development route, but now they get to see the entire iceberg.

Conclusion

Codefree Executable Model-Driven development has the potential to revolutionise the development process. It offers exciting opportunities for non-programmers to build applications, for programmers to build applications faster, and for teams to collaborate over development projects in a previously impractical way. It may not be the answer for every software requirement, but as its activities are an essential subset of the traditional development process, there is nothing to lose from giving Codefree Executable Models a go.