

THE COST OF CUSTOMISATION

Mike Evans; Incremental Ltd., UK

SUMMARY

Shipyards often seek to improve operations through adoption of management information systems. The envisaged benefits include time savings, cost reductions and better strategic information. However when they go to market they are offered a dichotomy - either buy a commoditised 'shrink-wrapped' product or a heavy-weight ERP (Enterprise Resource Planning) suite. The former are cheap but generic, the latter powerful but expensive. Neither can satisfy both the requirements and budgets of many yards.

This paper will demonstrate how the increasing ability of agile software development to rapidly and cheaply produce software, combined with the growing range of ever-more powerful open source technologies, can bridge this gap. We will show how not only initial development but crucially on-going support benefit from these innovations, and how every yard can now afford its own uniquely customised management information system.

NOMENCLATURE

AOP	: Aspect Orientated Programming [1]
CAD	: Computer Aided Design
ERP	: Enterprise Resource Planning
MIS	: Management Information Systems
OSGi	: Open Services Gateway initiative www.osgi.org
RCP	: Rich Client Platform www.eclipse.org/rcp
SOA	: Service Orientated Architecture www.opengroup.org/projects/soa

1. INTRODUCTION

We can divide computer applications for the marine industry into two broad categories – functional software and procedural software.

Functional software is targeted at specific technical tasks that are generic across the industry. Examples are CAD and hydrodynamic analysis.

Procedural software also aims to solve problems encountered across the industry, but which are generally solved on a company-to-company basis. Examples are estimating and planned maintenance systems.

Some procedural software is so constrained by external factors such as legislation that, for our purposes, they can be categorised as functional; e.g. pure accounting packages.

Functional software is essentially used 'as-is'. Selection is based on the capabilities and cost-effectiveness of the initially supplied product. Once installed, it is then operated within these constraints.

However with procedural software the initial selection is only the first step. Customers then expect to tailor the software to their unique requirements. The subjects of this paper are the issues and implications of this

customisation process, with an emphasis on how technical innovation can bring down costs.

2. MANAGEMENT INFORMATION SYSTEMS

For this paper we will use management information systems (MIS) as the example procedural computer application. MIS are not the only possible choice. However their functionality and application offer an ideal base with which to illustrate our ideas.

MIS cover a wide range of functionalities designed to aid the planning, monitoring and control of operations within shipyards. Under this very broad definition, we would expect the market to provide an equally broad range of software products. Instead we find either commoditised 'shrink-wrapped' products such as Microsoft Project or heavy-weight ERP suites from companies such as Oracle and SAP.

The former are relatively cheap (\$100's - \$1000's) but offer only generic functionality that cannot be tailored to yards' specific business. They can be considered functional software by our definition.

The latter can be integrated with a yard's business processes. However the costs of purchasing and then customising these procedural systems are in the \$10,000's-\$100,000's.

3. LIFETIME COST

The reason for this dichotomy is that pricing reflects the lifetime cost per deployment for the software supplier. This can straight-forwardly be calculated as:

$$C_{\text{lifetime}} = (C_{\text{development}} + C_{\text{support}}) / n$$

where:

C_{lifetime}	: lifetime cost per deployment
$C_{\text{development}}$: total development cost
C_{support}	: total support cost
n	: number of deployments

All things being equal there is no significant difference between the development costs of functional and procedural software. Moreover any improvements in software development practices equally benefit any development effort. Thus ‘C_{development}’ is not a differentiator. What makes the difference is the customisation inherent in procedural software deployments and its effect on ‘n’ and ‘C_{support}’.

4. THE COST OF CUSTOMISATION

The customisation of procedural software inflicts two cost penalties when compared to functional software. The first is its effect on the divisor ‘n’ - the number of deployments. The second is its direct impact on C_{support}.

Bespoke developments where ‘n’ equals one inevitably cost more than functional software where ‘n’ can be in the 100’s, 1000’s or more. The delicate art of cost-effective customisation is to increase the value of ‘n’ whilst still offering the unique functionality required by each customer. This feat requires an enhancement to the formula given previously:

$$C_{\text{lifetime}} = \Sigma (C_{\text{development}} + C_{\text{support}}) / n$$

We now consider the software not as a single entity but as a system made up of multiple components. Our cost calculation is applied in a similar way, only now the variables can be different for each component. Our aim is thus to increase the proportion of components where ‘n’ is high (and conversely decrease the number where ‘n’ is low or one) whilst simultaneously reducing ‘C_{support}’ for the components. It is this last consideration that merits further investigation, and that prevents this paper being a simple argument for code re-use.

5. ENCAPSULATION

The support cost ‘C_{support}’ of a specific component is a function of its quality and its complexity. Code complexity is whole field in itself, but in most modern development methodologies and technologies, a good proxy for code complexity is encapsulation.

Encapsulation hides internal representation and implementation details, thereby protecting parts of a system and its data from unwanted access or alteration by other parts. Encapsulation enables change in components without compromising the stability or functionality of other components.

If any one, overall trend can be discerned in the evolution of software engineering, it is the trend towards greater encapsulation at the finer and the broader scale. Examples range from the macro concerns of Service Orientated Architectures (SOA) to the intricacies of join points for orthogonal concerns in Aspect Orientated Programming (AOP).

For our purposes, the effect of encapsulation on support costs can be explained by stereotyping the architectures of customisable, procedural systems into one of :

- Big Ball of Mud;
- Hub and Spoke;
- Lego Brick.

The **Big Ball of Mud** architecture can be described as:

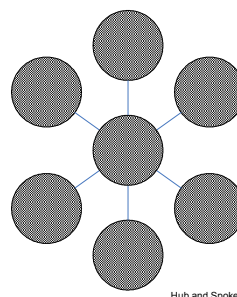
“... a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.” [2]

This is the default architecture for complex, procedural software systems, particularly those with an extended life. Big Balls of Mud have high ‘C_{support}’ and low ‘n’ (frequently 1 as they become, in effect, bespoke systems for each customer). It is difficult to customise them beyond

Big Ball of Mud

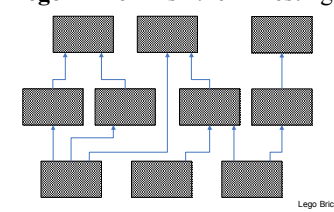
pre-planned limits, and which usually relies on a process of customisation by obfuscated configuration & undocumented procedures.

Hub and Spoke architectures have a common core into which optional spokes of functionality can be plugged. Customisation is a matter of selecting the required spokes and then tailoring these. A Hub and Spoke architecture offers a coarse grained component model, which risk becoming interdependent Small Balls of Mud. Alternatively the hub tends to expand over time at the expense of the spokes, becoming itself a Big Ball of Mud.



Hub and Spoke

Lego Brick is the finest grain component model. Functionality is broken down to as low a level as possible. Each component is linked only to its direct dependencies. Lego Bricks offer the lowest ‘C_{support}’ and highest ‘n’ of these architectures.



Lego Brick

Note this terminology should be applied strictly within the context of this paper and not confused with similar terms used elsewhere. The architectural category of procedural software is often revealed by the terminology used and procedures necessary to customise it. If features can be ‘unlocked’ or ‘configured’ then you are probably looking at a Big Ball of Mud. If you have ‘modules’ that can be incorporated in a one-off manner by the supplier’s technical staff you probably have a Hub and Spoke. If you can pick and choose the functionality you require, and apply it on a per-user, per-machine, per-day basis then you have true Lego Bricks.

6. WHY LEGO BRICKS?

It is tempting to view the three architectures described as a continuum, describing in reverse order the evolution of many computer products. They start out as cleanly designed Lego Brick version 1’s. Then the contingencies of production life, working to customers’ timescales, gradually erode the initial design principles and structure. Customisation upon customisation causes components to bloat and merge. Coupling makes Lego bricks into spokes, and eventually a Big Ball of Mud emerges.

However this view does not account for fundamental differences between Hub and Spoke, and Lego Brick architectures. These can be listed as:

- non arbitrary granularity;
- functional specialisation;
- general applicability;
- minimal state;
- additive, not just complimentary;
- active dependency management;
- refactorable code.

The **granularity** of Hub and Spoke architectures is frequently defined by the product’s marketing, which naturally targets customers’ business concerns; e.g. an MIS product may offer separate payroll and cost monitoring modules. These modules may integrate but are maintained as discrete components. In fact the processing is more or less the same so the code is either pushed into the Hub or duplicated across the Spokes. Reflecting the division of responsibilities within a shipyard does not lead to good technical design.

In contrast Lego Brick architectures offers a much finer level of granularity where the components can be scoped on purely technical merits, and then grouped as necessary to provide the business functionality.

Finer granularity makes for simpler components. They are **specialised** for their particular functionality, but **generalised** in their applicability. For instance, continuing the payroll/cost monitoring example, both functions may use a common component that collects job card data (how long who worked on what) and aggregates it. The component need not concern itself

with payroll or cost matters, needing only to know in what way to aggregate the data. Moreover if a new requirement arises to, e.g., monitor men’s exposure to VOC emissions, the component is immediately reusable.

A particularly powerful technique is to abstract the business domain model as much as possible; e.g. if a cost monitoring component deals with ‘cost centers’ rather than ‘projects’, then it can easily be used re-used in a planned maintenance function.

Generalisation also allows for **extensibility**. It removes constraints on how systems can be customised. It changes conversations with customers from ‘here is the list of options available’ to ‘what do you want’.

Finer grained, simpler components also have less internal **state**. This makes them more testable, which increases confidence when introducing changes.

These arguments point to an ever-finer degree of componentisation. This will almost inevitably degrade unless rigorous engineering discipline is applied throughout the system’s production life. Two fundamental disciplines must be followed if this is not to happen. Both are to do with the relationships between components.

The first is that functionality must always be **added** as components are built on top of one another. If component A depends on component B, A *must not change* B’s behaviour. In the object orientated world, it is acceptable for subclasses to override super class behaviour. For customising a system of inter-related components, this is almost invariably a bad idea. It introduces dependencies, risks unintended consequences and drives up ‘C_{support}’.

The second concept is that component **dependencies** must be explicitly declared and **managed**. Your Lego Bricks must be bound by a framework that accommodates changes to individual components and can deal with any mismatches or inconsistencies. Crucially it must handle versioning and be capable of backing-out changes.

If these disciplines are followed, a Lego Brick architecture produces **refactorable code**. This is code that can be modified, customised and extended beyond its initial scope. Maintenance and customisation work can be focussed and safe. In our cost equation, ‘n’ remains high and ‘C_{support}’ low.

7. THE EFFECT OF OPEN SOURCE

The Lego Brick architecture requires both suitable technologies and suitable development methodologies. Both are now available and both have been heavily influenced by the Open Source movement.

Open source development involves disparate groups of developers remotely co-operating, sharing ideas and code. This style of development more or less mandates a Lego brick architecture. Thus the patterns involved are repeatedly exercised and improved in an open forum. A classic example of the shift from Big Balls of Mud to Lego Bricks is the move away from the heavyweight, monolithic application servers to lightweight inversion-of-control frameworks.

Open source has heavily influenced the growth of ‘agile’ development methodologies. These demand an iterative development approach with many and frequent releases. Effectively they are customising their products every day. Naturally open source provides much technology that can be used within Lego Brick architectures. Third party code can be wrapped as a component and immediately used.

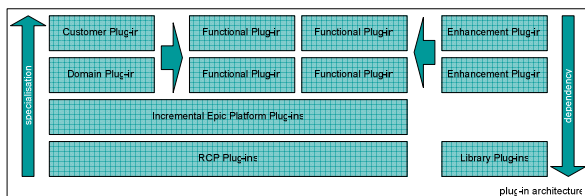
Open source offers a selection of frameworks to manage your Lego Bricks. This reduces the absolute scope of work, and permits the inverted efficiencies of scale encountered in software development (a 30 person development team does not produce 6 times the output of a 5 person development team).

Open source also provides the technologies to support Lego Brick development, such as source code repositories for complex version control, and continuous integration tools with automated regression testing.

Thus open source provides and proves the tools necessary to develop systems using the Lego Brick architecture. We shall now take a look at such a system.

8. AN EXAMPLE : INCREMENTAL EPIC

Incremental Epic is a management information system for ship repair and small-scale ship building. It is a classic Lego Brick architecture based on the open source Eclipse RCP project. The Lego Bricks in this case are OSGi bundles - within Eclipse these are called **plug-ins**.



To the base RCP platform, we add our own platform plug-ins providing abstract implementations of common elements, a security model, and most significantly a generic business entity model.

Upon this lie functional plug-ins, each representing a particular area within contract control – estimating, planning, purchasing, personnel, etc. Some, such as invoicing, rely on other functional plug-ins; others require only the platform.

The functional plug-ins are then tailored through a combination of enhancement and specialisation plug-ins. Enhancement plug-ins host functionality orthogonal to primary business logic such as reporting and Microsoft Office™ integration. The specialisation plug-ins are at domain (i.e. industry) and customer level. A noticeable feature of the design is though Incremental Epic was written specifically for ship repair, marine specific specialisation is only introduced via one of these top-level plug-ins.

Customisation can be introduced through more than 30 ‘extension points’. Through careful use of type and defensive programming, they are capable of specialising the business model and its presentation without compromising the stability or processing of the core functionality.

Customisation need not be a one-off process. Incremental Epic is designed to be continually customised, with each individual installation running its own set of plug-ins. These can be added and removed at any time. The system can even self test itself for any particular upgrade, customisation, or change of plug-ins.

Incremental Epic leverages all the advantages of its architecture. It is developed and maintained by a small team, yet is customised for each customer. The plug-in model permits a large number of variations in the core offering whilst preventing an explosion in the number of development streams.

9. CONCLUSIONS

The cost of customisation is directly related to the differences between different deployments of a system. To reduce lifetime costs, we need to reduce the differences. This requires the use of encapsulation and a fine grained component model. Modern technologies and methodologies, influenced and supplied by the open source movement, provide the means to inexpensively produce customised systems. Affordable, tailored computer systems are now within the reach of every ship yard.

10. REFERENCES

1. Kiczales et al, ‘Aspect Orientated Programming’, *European Conference on Object-Oriented Programming*, 1997
2. Foote, Yoder, ‘Big Ball of Mud’, *Fourth Conference on Patterns Languages of Programs*, 1999

11. AUTHOR'S BIOGRAPHY

Mike Evans is Technical Director at Incremental Ltd, a company specialising in software for the marine industry. Mike started his career in ship repair and is a trained naval architect. Mike is a CEng and a CITP of the British Computer Society.