Software engineering disposable packages

Michael Tan and Dwayne Read explore why building add-ons to software packages benefit users and developers

Michael Tan is a Principal Consultant at Acumen Alliance and has more than 20 years' experience in information systems. He has a strong technical background in systems development, application architectures and project management. Tan works as a consultant providing IT strategic planning skills to Commonwealth departments, statutory agencies and private sector businesses overseas. E-mail: michael.tan@acumen.com.au

Dwayne Read is an Engineering Process Consultant at Strategic Systems and is a process mentor and trainer with over 14 years' experience in software engineering. Read specialises in object-oriented techniques, extreme programming practices and a range of proven development and technical management activities. E-mail: dwayne@ss.com.au

Car analogy

I knew a yuppy who spent $20,000 on top-of-the-range car hi-fi equipment for his $15,000 car. Crazy, it seems. My wise acquaintance said not. "When I change my car (like, every six months), I simply bolt my hi-fi gear onto my new car. What's the problem?"

It raised the question: why can't the same be done for the software industry? If the car represents a package and the hi-fi gear is the 'add-on' made to the package, why can't the add-on stuff simply be 'bolted on' to the new version of the package?

If you are a package developer, you want to provide a package that offers ease of add-on functions like car manufacturers allowing users to add a hi-fi system. There is a need for such a type of package and the software engineering techniques that enable these flexibilities.

Package versus build

The business case between commercial off-the-shelf packages and build-your-own has been widely discussed. Most organisations prefer the package approach if suitable packages are available. Additionally, many prefer to change the business processes to fit a package rather than to modify the package to fit a process. But in the highly competitive world of today, what competitive advantage can one glean from the package if everybody simply adapts to the package? How can one differentiate oneself from the rest? If an organisation has a superior business strategy, it must be able to implement these strategies through some unique business rules in a package to differentiate itself from those who use the same package. But how does one do that?

The key concept here is add-on versus modifications.

Current inhibitors

In the current software environment, it is not a simple job to take all the modifications one has made in an older version and reuse these in a new version of a package. The norm is to reapply the modifications through some forms of linkages or compile and link edit. This is a tedious task because the package has been modified. The more one modifies a package to meet one's needs, the higher the risk that the modifications won't fit or that it takes a lot more effort to fit the modifications to the new version of the package.

Add-on versus modification

It is, therefore, important that users should not modify a package. The maintenance and evolution of a package should remain the realm of the vendor. To provide flexibility to meet specific user needs, the package should provide interfaces where users can implement their own business rules. These implementations must not modify the package itself. They merely add to the package using the interfaces.

A case in point

A Commonwealth department administers some 60-plus grants in its portfolio. It will be an outrageous investment building a program management system for each program. The investment will be huge and time consuming. On the other hand, all the programs share about the same processes at the high level, as depicted in Table 1. Only minor differences in terms of specific business rules exist in some of the lower processes. Examples of the differences are data collection and the criteria used in the evaluation during the review process.

Table 1 also lists typical variations to the key functions and the type of technology required to support the processes.

The department went to the market for a package that provides support to the main processes at the top level. The package must also provide capability to satisfy individual program differences as depicted in the variations row of Table 1. If you were a package developer, how would you design your system? In the fol-

	APPLY	REVIEW	APPROVAL	DISBURSEMENT	ACQUITTAL	ADMINISTRATION
KEY FUNCTIONS	• Apply • Query status • Maintain accounts	• Review applications • Recommended	• Review recommendations • Approve for payments • Pass payment details to FMS for payments	• Validate and pay • Return payments receipts • Payment receipts updated in database	• Enter audit reports • Process audit reports	• Add users • Query status • Check payments • Who has what
VARIATIONS	• Data collections • Validation rules	• Business rules to qualify an application				• Operational queries
TECHNOLOGY	• Browsers • Oracle • E-mail and attachments • EDMS	• Browsers • Oracle	• Browsers • FTP • XML • Oracle	• SAP • FTP • XML • Oracle	• Browser • Oracle	

Table 1 **High-level processes and functions of a grant management system**

lowing sections, some software engineering techniques are offered that can be used to provide the capabilities.

Handling variations

For a system to be able to handle the variations as add-ons, it has to be designed to do so. This is at the heart of software architecture. The realities are that requirements change, technologies change, the market/customer wants more flexibility and the development organisation wants to gain better reuse to help reduce costs, time to market and so on. This doesn't just happen by chance – it has to be designed to handle it.

As indicated in Figure 1, the first step to designing software architecture (also known as being 'architecturally centric') is that the variations of the system are identified and scoped – known as the architectural requirements. This is actually harder than it appears. It is undesirable to just say that the entire system is end-user configurable (or similar), as this is too broad. To go down this track will more likely than not result in significant expense and delays as one attempts to develop the ultimate generic solution – too ideological. Energies need to be focused on the parts that are variable, and these should be clearly scoped.

As with any good requirements specification, the architectural requirements must be testable. This gives the architecture a clear goal of what (and where) effort needs to be applied to ensure that a new or replacement add-on can be incorporated.

The 'business rules to qualify an application' variation from Table 1 (above) is a fairly common situation where the business rules are different for each customer/department or they change over time.

The variation 'business rules to qualify an application' would typically be scoped as follows:

- **Configure business rules to qualify an application (ARCH-007) –** The Department Manager can alter the settings of the business rules that indicate the approval or disapproval of an application. The configurable parameters (for the default business rules) are:
  - Maximum revenue threshold (e.g. $20 million)
  - Minimum revenue threshold (e.g. $10,000)
  - Maximum total portfolio value threshold (e.g. $5 million)
  - The roles that can manually approve (i.e. override) an application (e.g. director).
- **Replace business rules to qualify an application (ARCH-008) –** The Systems Integrator can install alternative business rules logic to be applied in the qualification of an application. Note: any configurable parameters for the alternative business logic will need to be specified.

So what has been achieved here? Hopefully, the

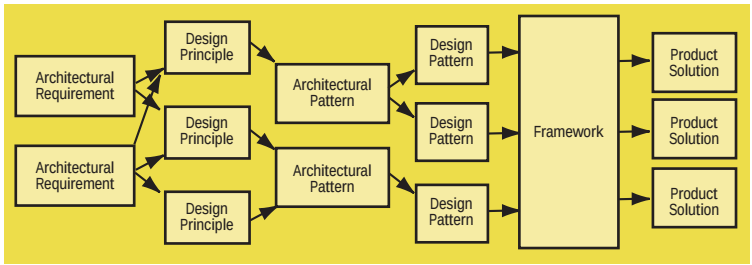

Figure 1 **Key steps for framework development**

For a system to be able to handle the variations as add-ons it has to be designed to do so. This is at the heart of software architecture

extent of the 'variation' that is required to be handled by the system has been clarified. The second architectural requirement (ARCH-008) is the critical one, as it is highlighting that a mechanism needs to be built to enable replacement of the business rules – this would be an add-on. The first architectural requirement (ARCH-007) is actually just an instance of one of these, probably the default one for the framework, showing that it actually has some configurable parameters (that will need user interface, etc.). Ivar Jacobson's[1] use case analysis approach lends itself to scoping these two architectural requirements in the form of 'verb the noun' ('configure' being the verb and 'business rules' being the noun). Why? Because it helps to ensure that they are testable. The roles (à la 'actors') that can undertake these tasks have been identified (i.e. the department manager and the systems integrator). This gives more context and indicates who has the privileges to do these tasks.

Now for the design work. The system architecture needs to be captured in three ways: the logical and the physical architecture and a set of design principles.

1. The logical architecture is typically captured in the form of a package diagram, possibly with some key interface classes represented. This should provide the roadmap of how the software is modularised, typically into layers (e.g. presentation, security, business layer, transaction, persistence, legacy interface) and partitions (e.g. contact management, timesheets, payroll), plus where any critical third-party and/or legacy components logically reside (e.g. an object request broker, a database gateway.).

2. The physical architecture is typically captured as a combination of the deployment and component diagram. There should be one for each network topology that needs to be supported (e.g. local area network, LAN, plus fibre optic WAN to headquarters, and etc.). These show the physical hardware, the executable software and pertinent data stores.

3. It's the design principles that are the interesting/challenging part of architectural design. This stems from some work by Richard Helm, one of the Design Patterns authors.[2] The architecture's design principles are the rules of why the design is the way it is, capturing the motivation and impact of adhering to such rules. For the 'business rules to qualify an application' example, some design principles would probably be along the lines of:

- **All business logic in domain objects:** All validation and processing of business logic is encapsulated in the domain objects within the business layer. Note: only base-type data entry validation (e.g. integer, date, etc.) is permitted in the presentation layer.
  Motivation: Consolidation of business logic in one area; easier to extend and/or replace; thinner and 'dumber' clients.
  Impact: Two-step validation of data (field and domain object), simpler GUI code.
- **All domain objects to call 'extendLogic':** Before applying the 'core' business logic, each domain object will call the 'extendLogic' method that will execute any customised business logic and return true/false to the calling object. The 'core' business logic will then run/not-run according to the Boolean return.
  Motivation: Common mechanism for extending business logic without the need to alter existing code; isolate the customer unique extensions.
  Impact: Isolates extensions without code changes; call overhead for domain objects with no extended business logic.

These fairly specific design decisions focus purely on enabling the business logic to be replaced. There are several (maybe more) ways of meeting the same architectural requirements, but this is what has been decided. This is architecture, even though a particular method and even its return type (Boolean) is being referred to. Architecture is not just about high level design statements, it's about common mechanisms being employed across the entire system (or a large part thereof). The reason the motivation is stated is to ensure that the rules of the architecture are justifiable. The reason for stating the impact is to ensure that the pros and cons of this rule have been considered – there are always some negative side-effects, but at least this is understood upfront. Plus, these rules should be critiqued and even argued about by the development team, now and in the future (remember that requirements change) – this is healthy.

Key design principles

Now, beyond the 'business rules to qualify an application' example followed here, how should 'variations' in the large be tackled so that the system can be comprised of disposable packages?

The two fundamental design principles that come together to enable this approach are:

1. Isolate and encapsulate dependencies This is all about finding good abstractions and "information hiding – two of the founding characteristics of object-oriented (OO) software engineering itself. To enable package disposal and replacement, the elements must first be isolated. That is, 'a red circle' needs to be drawn around the features that are going to change, these are separated from the rest of the system (to reduce the direct dependencies) and then encapsulated through some form of interface mechanism, such as a simple interface or abstract 'wrapper' class, interface definition language (IDL) or XML (Extensible Markup Language) for cross-platform encapsulation, etc. Now this is even harder than coming up with the architectural requirements – as one is effectively second guessing what's going to change in the future and how (to some extent), but there is no need to worry too much about this, per se – it's a challenge, but iterative development and refactoring help in this regard.

2. **Open/closed design principle** This means to ensure that the functionality of the application can easily be extended (read 'open to extension') in such a way that the rest of the application (and several sibling applications at other sites) don't have to change their implementation (read 'closed to modification'). This can work at several levels. At the lowest/class level abstract (more generalised) classes will be created that become the 'hooks' that the rest of the application is allowed to call and, thus, be dependent upon. Sub-classes will then be created that inherit from the abstract class, enabling any one of many concrete sub-classes to be substituted at run-time 'behind the abstract interface' without the calling code knowing (this is known as polymorphism). Mechanisms can be built to further hide the construction of the concrete sub-classes through such design patterns as the Abstract Factory.[3] At the higher/component replacement level an interface class or mechanism (IDL, XML, broker, DCOM, etc.) may be used to switch components (either built in-house or from a third party) that adhere to the same interface/protocol (these components are known as the 'realisations of the interface'). Obviously one would typically have a combination of class and component level mechanisms to handle the typical variety of disposable packages and add-ons.

Building a framework (the implementation)

The architectural and design patterns have been mentioned in the context of the design principles above.

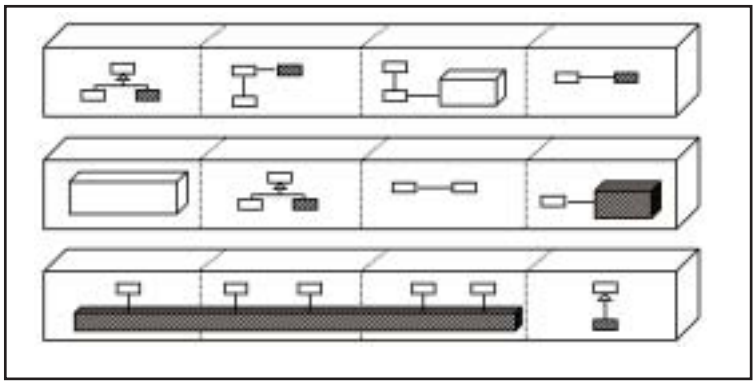

Figure 2 **Frameworks provide the placeholders for the packages to be replaced**

These patterns provide the models to enable the reuse, flexibility, extensibility and consistency required. So what are the key aspects of actually building a framework?

The framework repository is considered as the implementation and integration of all components (source code, binaries and data) developed in-house and from third parties that are specific to an application domain. The reason why a framework must be domain specific is that a business has a particular focus in the market, so if the framework's focus is aligned with that, then the effort and complexity of the developments will be dramatically reduced and actually achieve a higher level of reuse, requiring less application-specific development to integrate the packages and add-ons together. One cannot be all things to all people. The framework needs to be the equivalent of being able to buy any car (the framework) and any hi-fi equipment (the customer's add-on). All that is then required is to wire the two together, maybe making some kind of adaptor plate to physically join the two – this is 'systems integration'.

As portrayed in Figure 2, the framework provides the placeholders (the shaded parts) for the packages that are designed to be replaced. Often a framework will actually come with some default packages that make it a 'complete application', but it is expected that at least some of these will be replaced.

At the heart of a framework are the interfaces. These are the focal points of any architecture and should be one's own interface (even if it's based on a third party's). Why? So that it can be controlled – if it changes it can hurt … If/when a third-party component does change then if and how the update is incorporated can be controlled. This can be considered as 'glue' technology.

How far one can go with the framework depends on the nature of the domain (extent of variations required) and technologies one needs to support.

There needs to be a set of standard interfacing techniques that users can use to build add-ons to packages

However, the reality is that the framework needs to be pitched to minimise the coding activity to a little bit of integration code for each project/product variant and emphasise the building and adoption of a suite of packages and add-ons (third party or developed in-house) that can be easily integrated and reused.

Replacing packages and adding add-ons

With the framework in place, the entire process changes from one of software development to one of systems integration. Building a framework repository with the ability to add/replace packages (at varying levels) is hard work, takes several iterations, several applied projects and will/should always be an ongoing development activity (even more so than specific project development work). Both the initial and ongoing development of the framework needs to ensure healthy refactoring[4] of the software to make it easier, cleaner and more cost effective to replace the packages and integrate the add-ons.

As a 'systems integrator', the most common activity now becomes one of creating the bindings to the new package so that it complies with the framework's interfaces and comes to 'life' with the unique flavours required by the next customer. So the same features can be effectively maintained but the underlying technology (e.g. database) can be replaced, security can be added/altered, different business logic/rules can be incorporated, the look-and-feel can be changed, and etc., so long as the interfaces are adhered to. These interfaces can be custom made or one of the emerging industry 'standards'. However, one should always put their own 'wrapper' around any third-party component, albeit a standard or otherwise, as even standards change.

These bindings will typically take the form of a class or two that translate the interface or call structure of the add-on to that of the framework's defined interface or abstraction. As always, there are several options to achieving this: straight through calling code, a parallel inheritance hierarchy (such as the Bridge pattern[5]), calling a broker (typically third party) that will undertake the translation (although one needs to be compliant to their interface, not the other way around).

So now with the framework, instances can be created to produce as many product solutions as there are customers (or target markets) in a very time and cost-effective way.

Conclusion – a disposable package

In the car analogy, the car hasn't been modified in order to use the hi-fi gear. The top-of-the-range hi-fi gear is a mere add-on. They are easy to fit because they have the same interfaces as the bundled hi-fi systems. In fact, the interface is so similar in the car industry that my yuppy friend can keep changing cars and still use the same hi-fi gear.

That is what needs to be achieved in the software industry. There needs to be a set of standard interfacing techniques that users can use to build add-ons to packages, very much like hi-fi equipment for cars. Obviously the nature of the interfaces in the software world are more complex and dynamic, so protection mechanisms need to be built around these interfaces (standardised or otherwise) to enable these replacements. Users invest in the add-ons to packages to meet business needs. The add-ons can be reused easily as users upgrade to new versions of the package. The package itself is disposable. When a new version is available, the user can simply toss out the old version and apply the add-on to the new version.

With this approach, a disposable package will provide the value of a package (short implementation and low risk). A disposable package will also offer the users great capabilities to meet user-specific requirements without modifying the package. Users will continue to benefit from the package evolution as the vendor adds more function to the package to cope with advances in the industry and technology. ■

FOOTNOTES

1 Jacobson, I., et al, *The Unified Software Development Process*, Addison-Wesley, 1998.

2&5 Gamma, E., et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

3 Ibid.

4 Fowler, M., *Refactoring – Improving the Design of Existing Code*, Addison-Wesley, 2000.

FURTHER INFORMATION

Beck, K., *Extreme Programming Explained*, Addison-Wesley, 2000.

Buschmann, F., et al, *Pattern-Oriented Software Architecture*, Vol. 1, Wiley, 1996.