ARM yourself for Enterprise Application Development

If you've been involved in large scale software development for any length of time, you'll surely recognise the following symptoms of application architecture and design decay:

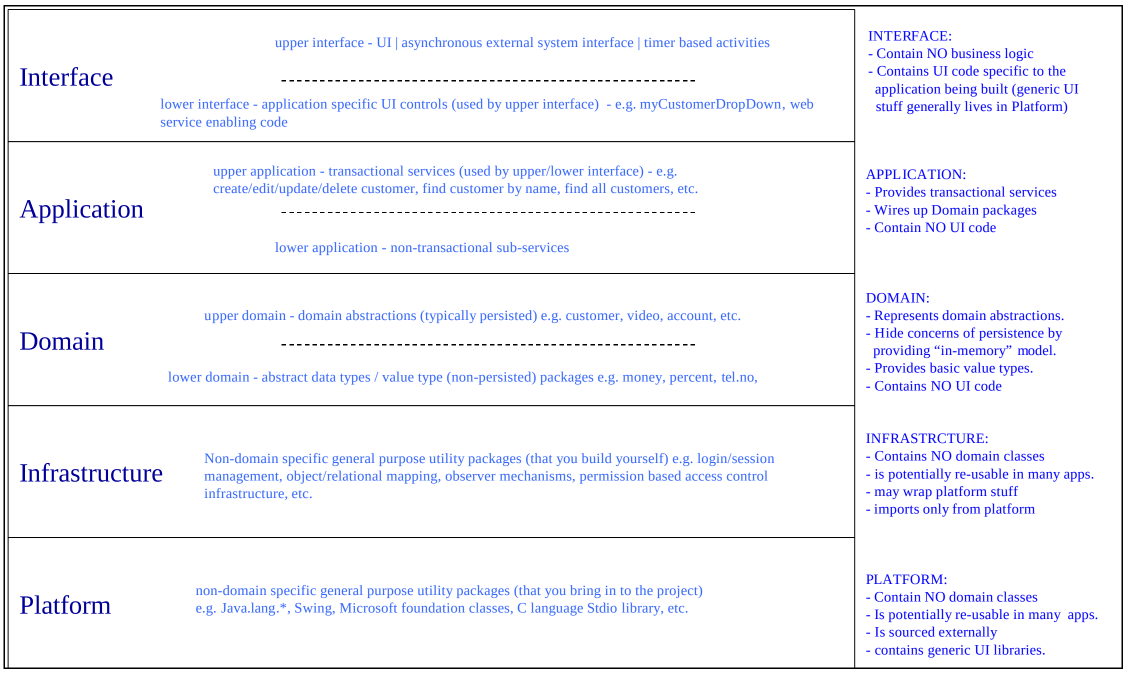
- you keep having to *cut'n'paste repetative code* (along, of course, with any associated bugs), as proper *re-use* of code isn't a viable option. Poor code factoring makes this impossible;
- this in turn leads to *duplication* existing all over the place – making a functional single simple functional change then requires amendments to be made all over the place;
- the *choatic package structure* of the application means there is a prohibitively expensive learning curve for new staff – there's no clarity of responsibility in what package does what;
- everyone keeps *treading on everyone else's toes* when making changes to the system – lack of any consistent packaging rules make intelligent work scheduling difficult;
- *automated testing is difficult*, it's impossible to test any package in isolation due to poor dependency management;
- *bug fixing is diffiult*, spaghetti-like class dependencies make it difficult to track the source of an error,
- all in all, the whole application seems to be *unstable*, simple functional changes require large code rewords, and everything breaks all the time;

In this article I discuss an architectural reference model (ARM) for large scale applications, that I've successfully employed on three large enterprise application developments in the last few years (two in the finance sector, and one in on-line auctioning).

The ARM is made up of of five architectural stata (Interface, Application, Domain, Infrastructure and Platform). The overriding purpose of the ARM is to provide a clear set of rules for large scale application decomposition, that encourage seperation of concerns, maximises re-use of code (primarily) within the application and (secondarily) across applications, that leads to good code factoring without duplication, and increases application stability in the face of changing requirements.

To achieve this end, the ARM has an associated, and fairly easy to apply, set of rules which force a coherency of structure on your application, leading to: greater clarity of responsibility – as to which code does what, improved code factoring – reducing duplication within the code, increased consistency in packaging rules, more manageable dependencies between packages, thus mitigating to some degree the problems outlined above.

1. Introducing the ARM



Interface	upper interface - UI | asynchronous external system interface | timer based activities --- lower interface - application specific UI controls (used by upper interface) - e.g. myCustomerDropDown, web service enabling code	INTERFACE: - Contain NO business logic - Contains UI code specific to the application being built (generic UI stuff generally lives in Platform)
Application	upper application - transactional services (used by upper/lower interface) - e.g. create/edit/update/delete customer, find customer by name, find all customers, etc. --- lower application - non-transactional sub-services	APPLICATION: - Provides transactional services - Wires up Domain packages - Contain NO UI code
Domain	upper domain - domain abstractions (typically persisted) e.g. customer, video, account, etc. --- lower domain - abstract data types / value type (non-persisted) packages e.g. money, percent, tel.no,	DOMAIN: - Represents domain abstractions. - Hide concerns of persistence by providing "in-memory" model. - Provides basic value types. - Contains NO UI code
Infrastructure	Non-domain specific general purpose utility packages (that you build yourself) e.g. login/session management, object/relational mapping, observer mechanisms, permission based access control infrastructure, etc.	INFRASTRCTURE: - Contains NO domain classes - is potentially re-usable in many apps. - may wrap platform stuff - imports only from platform
Platform	non-domain specific general purpose utility packages (that you bring in to the project) e.g. Java.lang.*, Swing, Microsoft foundation classes, C language Stdio library, etc.	PLATFORM: - Contain NO domain classes - Is potentially re-usable in many apps. - Is sourced externally - contains generic UI libraries.

Figure 1 – The five strata of the architectural reference model – each of which acts as a placeholder for one or more packages. The stratum in which a package "lives" is determined by what the classes within the package do, and what other packages the classes depend on.

1.1. Application sub-division

The ARM – see figure 1 - is based on five fundamental architectural strata – each of which I'll describe in some detail in the coming sections. Each stratum acts as a *placeholder* for the packages that, combined together, will make up the source code to your application. It's important to note that the stratum are *not* themselves packages, but – as per the rules in the following section – help to determine what should and shouldn't be in any individual package.

The ARM has two sets of associated rules: general rules – which apply across the whole model, and stratum specific rules, which apply to each stratum individually:

1.2. The rules (general)

- *strata are divided according to functionality and 'natural' dependencies.*
  Each stratum has an associated set of responsibility guidelines (a.k.a. concerns) which determine what the packages and classes within it are allowed to do. Each stratum is

'naturally' dependent on strata below it – in the sense that the functionality of the classes within it will be built using the facilities provided by lower stratum classes and packages.

- *depend downwards.*

  Packages in a given stratum are only allowed to depend on - import from - packages in the same or lower stratum. Dependencies to packages in the same stratum should be avoided where this does not unduly increase complexity.

- *packages can only live in one stratum.*

  A package that contains some classes that belong in one stratum, and some that belong in another, is designated to live in the higher stratum. Well factored applications tend to have a greater number of classes within the lower strata – indeed one objective of the ARM is to put a focus on this type of factoring – so there's a clear implication here that if a package cross stata boundaries you should consider restructuring it. Applications with all or most of the packages in the higher strata – as per this rule – tend to be poorly factored – and often exhibit the problems discussed at the beginning of this article.

1.3. The rules (stratum specific)

- *Platform underpins the application development.*

  Platform contains packages or utilities that are acquired externally to support the development. Typical examples include Jakarta Struts, java.lang.*, Swing, Microsoft Foundation Classes, .NET GUI libraries, etc. Although not discussed greatly in this article choice of Platform technology is a key to ensuring project success –hence its inclusion in this model.

- *Infrastructure neither contains nor depends on domain-specific code*

  Packages in *Infrastructure* are those which contain general purpose (non-domain specific) classes that provide utility functionality applicable to many different types of application. Infrastructure may only depend on (import from) platform. Typical examples include: general purpose object/relational mapping code (persistence); general purpose observer mechanisms; general purpose group based security mechanisms; and thin wrappers imposing a restricted API on Platform functionality.

- *Domain contains domain specific classes (often called entities)*

  Packages in *Domain* contain domain specific abstractions that you'd typically find in an entity relationship or domain model. User and/or external system interface/presentation code is specifically prohibited. In the context of enterprise applications, Domain should provide the illusion (abstraction) that all domain classes are in memory, hiding the details of any persistence mechanism. Domain may also hide other Infrastructure concerns, where this does not introduce needless complexity.

The upper domain stratum contains packages made up of “reference” objects – those typically persisted in their own right (with primary key, etc.) in a relational database. Typical examples include: a customers package (providing access/update functionality to the set of customers), an accounts package, etc. The lower domain stratum often contains packages that provide “value” objects – those that are typically contained by value as attributes of upper domain classes. Examples include: telephone number, date, money, etc.

- *Application provides a service oriented architecture*

  Packages in *Application* provide a set of application specific transactional services that the Interface stratum uses to query and/or update the application state. Application packages typically “wire-up” or provide the “linking glue” for decoupled Domain packages (see figure 4). The upper Application stratum will provide transaction services like: createCustomer, getAllCustomers, createAccount, getAccountsByCriteria, etc. Lower Application packages, if present, will contain utility sub-services that are non-transactional, but which are used to construct the transactional services provided by the upper Application.

- *Interface packages make the application do something!*

  At its most fundamental, the purpose of the Interface stratum is to interact with the outside world, and at its request, call the Application stratum to change the application’s internal state. Most commonly the Interface holds packages that provide application specific user interface – that is - user interface classes that are particular to the application in question (*not* general purpose UI toolkit classes). Interface may also contain code to parse application specific file-interchange formats (e.g. a custom XML format), deal with requests from external systems (web-enabling code), or application specific timer related activities.

  The Upper Interface typically contains complete user interface dialogs such as the “create a new customer” screen, or the “find a video” screen. Lower Interface, if present, may contain application specific user-interface “widgets” such as a “find video by name” pane (one used on multiple screens), or an account-type drop-down listbox (containing values such as ‘current’, ‘savings’, etc).. Lower Interface widgets are thus used to build upper Interface user interfaces.

2. Video stores – a case study

That's enough theory for the moment. I'll now show you an example based on a video store application. Note that the apparently simple set of requirements – see figures 2 and 3 - are sufficient to pose some architectural complexity.

- customers can register for email or text message alerts that tell them when a video is available to rent
- customers can reserve a video by replying to a text message alert
- customers can search for videos by partial title, and reserve a video (using the GUI)

Figure 2 – requirements fragment for iteration 3 of the video store application. The application has two type of user interface – one GUI based and one mobile phone based. Figure 3 shows the associated domain model for this application.

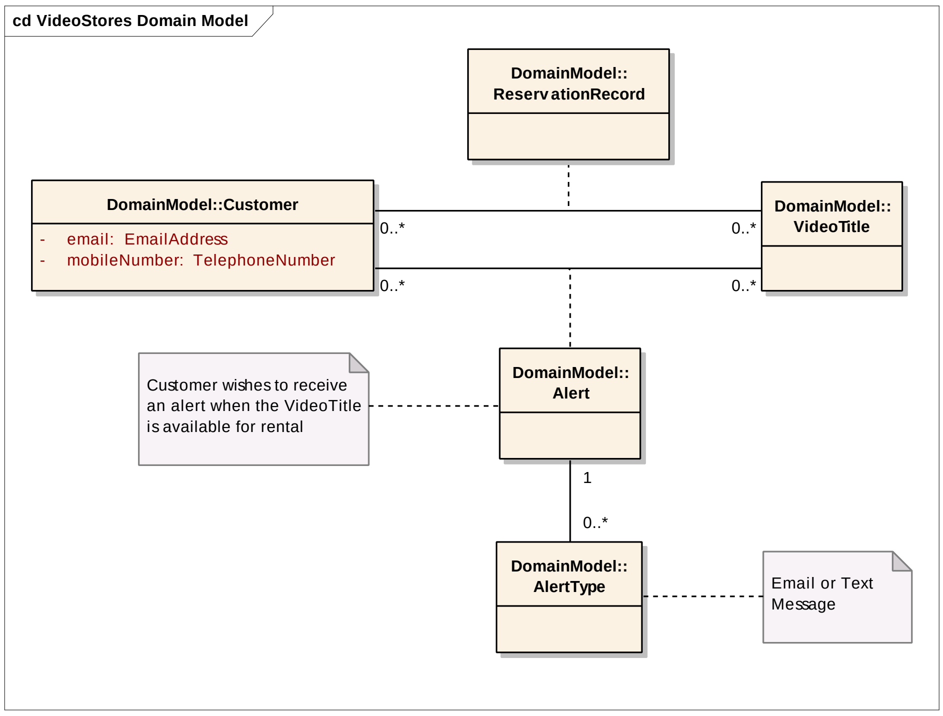

cd VideoStores Domain Model
DomainModel::ReservationRecord
DomainModel::Customer
- email: EmailAddress
- mobileNumber: TelephoneNumber
DomainModel::VideoTitle
0..*
0..*
0..*
0..*
Customer wishes to receive an alert when the VideoTitle is available for rental
DomainModel::Alert
1
0..*
DomainModel::AlertType
Email or Text Message

Figure 3 – domain model for the video store application.

Figure 4 shows the architecture resultant from iteration 3 of the video stores project.

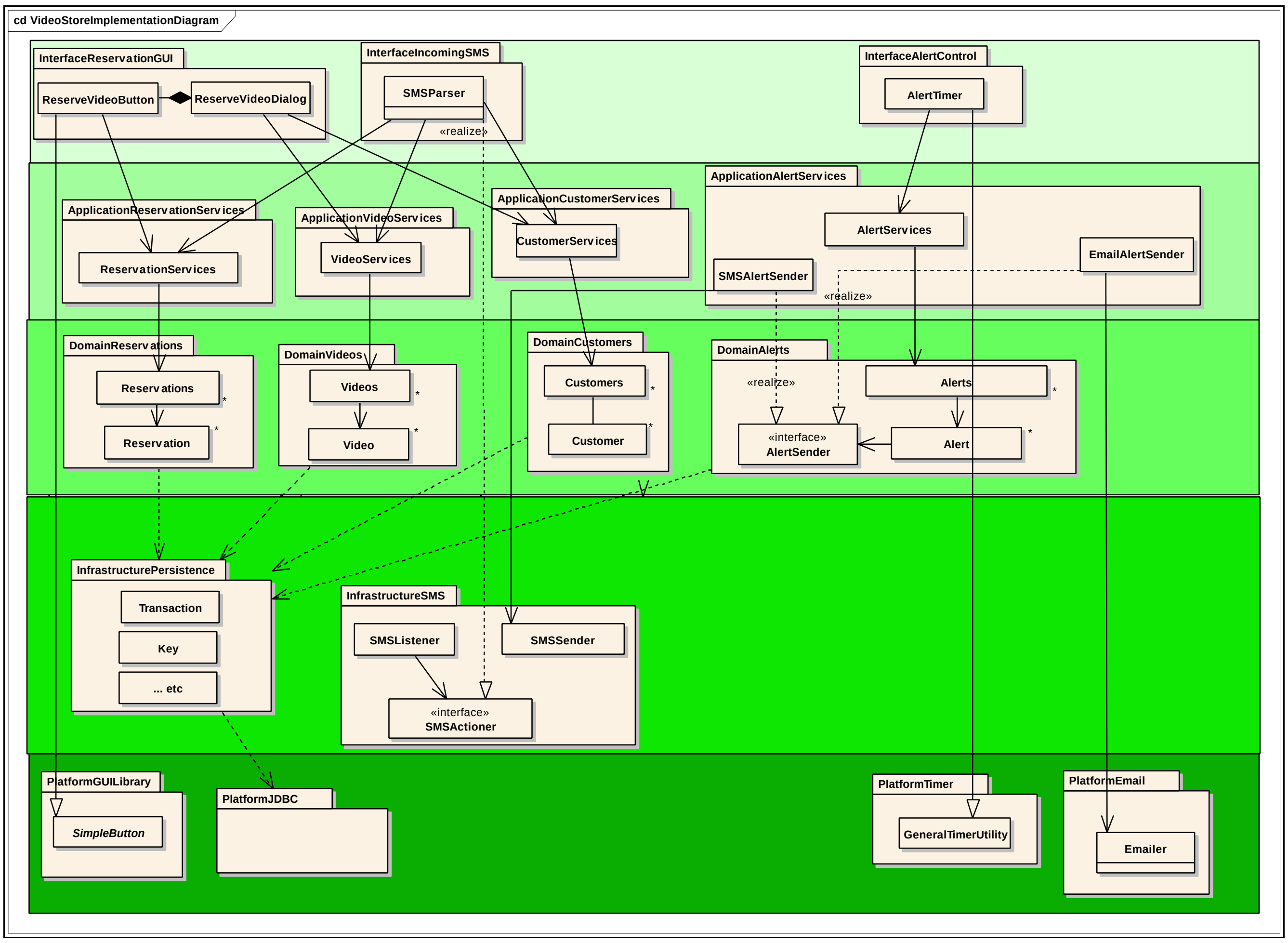


Figure 4 – detailed view of the architecture of the video store system, with strata superimposed. Package names also indicate the stratum of a package. One infrastructure package (SMS) has been culled from a previous project. The Persistence package (which manages the object/relational mapping in a domain independent fashion) is shown only in outline.

2.1. Interface

Interface contains three packages, each of which is responsible for kicking the application into life:

- InterfaceReservationGUI – responsible for the UI for reserving a video,
- InterfaceIncomingSMS – responsible for reserving videos in the event a customer responds to an SMS alert, parsing the SMS text and creating a reservation for the video.
- InterfaceAlertControl – responsible for sending alerts periodically.

Notice that all three of these packages depend - due to either inheriting from a class or implementing an interface - either on Infrastructure or Platform packages. In all three cases control is passed to Interface code from Infrastructure or Platform code via a call-back (using a technique called Inversion of Control or Dependency Injection). This is not an uncommon situation, and occurs when code is factored to remove application and domain specific dependencies – leading to better code factoring and making the SMS package, in particular, usable in multiple contexts. The ARM gives context to this type of code factoring.

2.2. Application

Application contains four decoupled packages, each of which provides a set of services that can be used by Interface code. Note that ReservationServices code in figure 4 is used by the both the SMSParser and the InterfaceReservationGUI packages – each packages provides a different user interface to the same underlying functionality. This type of code factoring is a key motivation in the separating the concerns of Interface and Application.

2.3. Domain

Domain contains four packages – one for each entity identified in the domain model – each of which follows a common project pattern (see figure 5).

What is particularly noteworthy here is that each package in Domain is completely decoupled from the others. None of the Domain packages depend on each other: Videos don't know about Customers, Customers don't know about Videos, and perhaps most surprisingly Reservations (see figure 6) don't know or depend directly on either.

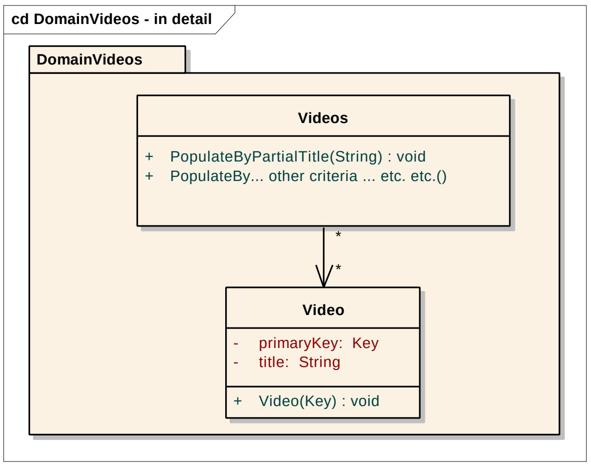

cd DomainVideos - in detail
DomainVideos
Videos
+ PopulateByPartialTitle(String) : void
+ PopulateBy... other criteria ... etc. etc.()
*
*
Video
- primaryKey: Key
- title: String
+ Video(Key) : void

Figure 5 – detailed view of the contents of the DomainVideos package. Videos provides the mechanism by which collections of video Keys can be returned to Application code. Individual Keys are then used to instantiated individual Videos. The other domain packages in this example follow this pattern.

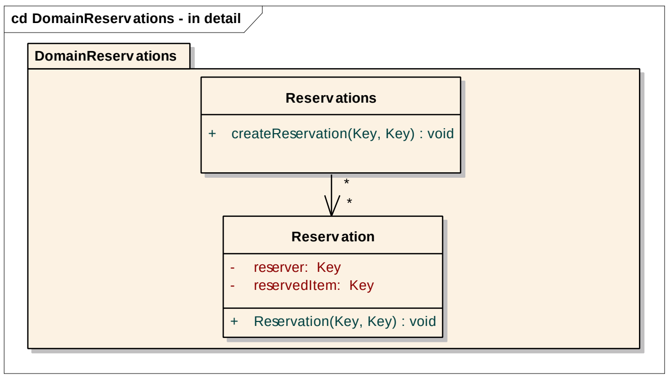

cd DomainReservations - in detail
DomainReservations
Reservations
+ createReservation(Key, Key) : void
*
*
Reservation
- reserver: Key
- reservedItem: Key
+ Reservation(Key, Key) : void

Figure 6 – The DomainReservation package is completely decoupled from DomainVideo or DomainCustomer packages. This is achieved using a simple technique, namely by the Reservation class storing only the Keys of the reservedItem and the reserver. In this case it is up to Application code determine what it does with these Keys. A variety of other techniques can be used to achieve Domain package decoupling. For example, Application code may often wire up Domain packages using Adapters. Note that the Key class shown here is imported from the Persistence package (see figure 4).

The DomainAlerts package both keeps track of alerts and sends them when asked to do so. In figure 4, it doesn't actually know the mechanism(s) by which an alert can be sent, so it provides an interface (of the Java type) which is implemented by ApplicationAlertServices. If only one alert mechanism were needed, this might be considered needless complexity. However, Alerts can be sent by email or SMS, so this design results in less code being required overall, and has the additional benefit of making automated testing an easier proposition (see figure 7). The design is now also inherently extensible.

It is not atypical to see examples of Domain packages having their behaviour customised, in this fashion, by Application packages – indeed, this is one of the drivers for the seperation of Domain and Application.

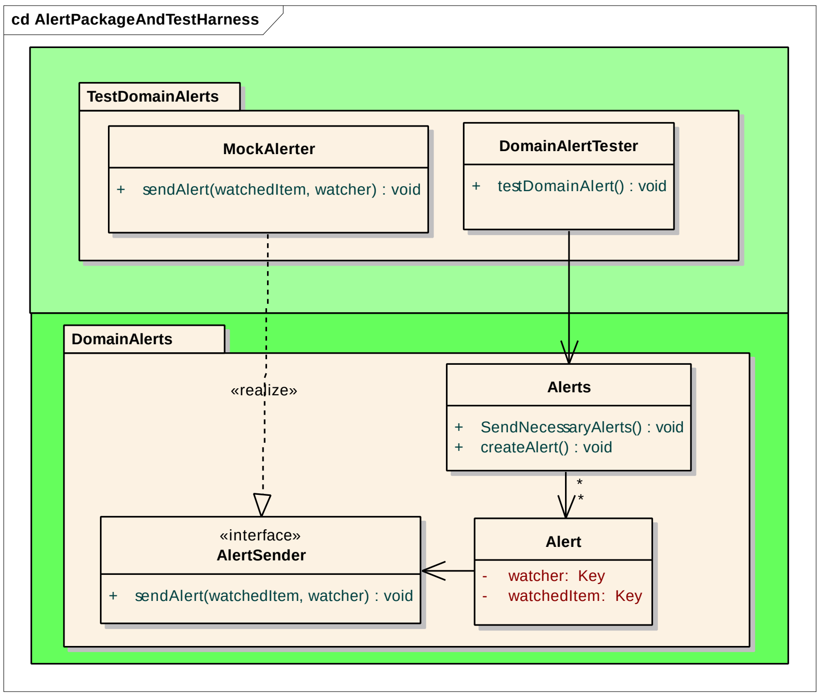

cd AlertPackageAndTestHarness
TestDomainAlerts
MockAlerter
+ sendAlert(watchedItem, watcher) : void
DomainAlertTester
+ testDomainAlert() : void
DomainAlerts
«realize»
Alerts
+ SendNecessaryAlerts() : void
+ createAlert() : void
*
*
«interface»
AlertSender
+ sendAlert(watchedItem, watcher) : void
Alert
- watcher: Key
- watchedItem: Key

Figure 7 – TestDomainAlerts masquerading as an Application package – adopting a client code perspective. To test DomainAlerts in isolation, TestDomainAlerts provides a MockAlerter, which enables it to verify that alerts are being invoked correctly. This is an example of dependency injection in action.

2.4. Infrastructure

Infrastructure contains two packages. The InfrastructurePersistence package is worthy of an article in its own right – but for our purpose here let's just say that it manages the interface to a relational database, exports the Key class (and probably a Versioned Key class) and Transaction (unit of work) class, and makes sure things are stored in the database. As shown, both Application and Domain classes rely on Persistence facilities directly. In particular each service exported by Application packages will use Transaction class facilities.

Of more interest to us here is the InfrastructrueSMS package. A package providing general purpose SMS facilities is an obvious candidate for inclusion in Infrastructure– and could be genuinely re-used across many projects. In the example shown, an SMSListener class is instantiated to listen on an appropriate number (i.e. the number on which we're expecting to receive incoming text messages). Incoming messages are forwarded to the SMSActioner interface, which in this example is realised by the SMSParser class in the InterfaceIncomingSMS package. See figure 8.

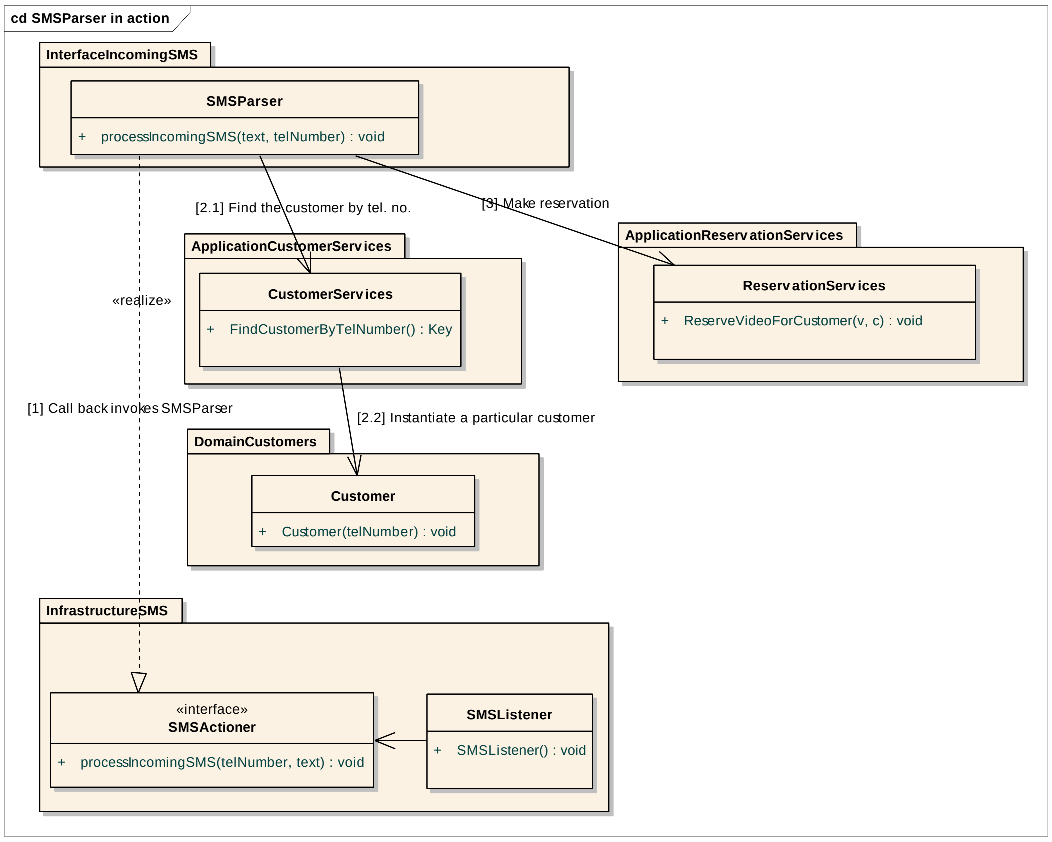

cd SMSParser in action
InterfaceIncomingSMS
SMSParser
+ processIncomingSMS(text, telNumber) : void
[2.1] Find the customer by tel. no.
[3] Make reservation
ApplicationCustomerServices
CustomerServices
+ FindCustomerByTelNumber() : Key
«realize»
ApplicationReservationServices
ReservationServices
+ ReserveVideoForCustomer(v, c) : void
[1] Call back invokes SMSParser
[2.2] Instantiate a particular customer
DomainCustomers
Customer
+ Customer(telNumber) : void
InfrastructureSMS
«interface» SMSActioner
+ processIncomingSMS(telNumber, text) : void
SMSListener
+ SMSListener() : void

Figure 8 – [1] An incoming SMS invokes a call-back to the InterfaceIncomingSMS package. [2.1] SMSParser.processIncomingSMS parses the incoming message to extract the Video ID to be reserved (contained in the message) and [2.2] looks up the Customer who sent the SMS by their telephone number, before finally [3] making a Reservation.

2.5. Platform

Platform contains the building blocks that underpin the whole development. If you assume the video stores application is written in Java, it will most likely use standard Java libraries to build the GUI, to interface to the database (JDBC), for the basic timing mechanism, and to send email.

3. The ARM revisited

3.1. A conceptual *and* practical model

As you can see from the above example, the reference model has been of conceptual - in terms of assisting in how to think about application structure - and a practical - in terms of concrete package sub-division visible in source code - assistance in getting us to a well structured, well factored application with a coherent and manageable set of dependencies.

Summarising:

The ARM (including its rules) prescribes how to horizontally sub-divide a large application into packages based on the responsibilities assigned to classes, and the natural dependencies that will exist between these classes.

3.2. A deeper look at the strata

So what exactly does it mean that one stratum is "on top of" another in this model? The ARM pulls together three threads of reasoning about good application structure:

- *Dependencies.* Most simply understood is the issue of compile-time dependencies. Packages in a higher stratum import from packages in the same or lower stratum. Put another way, the dependencies always point downwards. Understanding and managing your dependencies is a key feature of flexible application architecture and the ARM is there to help you do this.

- *Functional specificity/neutrality.* The higher the stratum, the more application specific and functionally powerful the operations provided, and the nearer you get to having a complete application. Put another way, the higher you go, the easier it should be to build your specific application with the facilities provided. Conversely, the lower you go, the more work you will have to do to build a specific application, but the more open the range of possible application you could write is.

- *Stability.* The higher the stratum, the less stable (in the face of changing customer requirements) packages become. It's far more likely that an Interface package will change than a Infrastructure package – assuming you've factored your packages correctly – and the whole point of factoring out domain specific functionality from Infrastructure is make it stable against change. There is a *but* here, however. Given the current state of programming language technology, changes in non-functional requirements (say, changing a single user in-memory application to a multi-user database application) can still pull our foundations out from under us. Hence the desire to tie-down non-functional issues as early as possible in a project lifecycle.

Dependency management and functional power/specificity are integrally related concepts, for the pure and simple reason that we build higher level functionality out of lower level functionality - and hence depend on it. Stability is a by-product of factoring out "higher" level functionality that is more likely to be subject to change. In combination, these factors make the ARM a powerful weapon in the enterprise architects armoury.

3.3. Why five strata?

As you’ve seen, Platform is the home of the technology base that underpins your application development. Getting your Platform components right can be a project in its own right – hence its inclusion in the ARM - even though it has a lot of similarities with Infrastructure. The distinction between Platform and Infrastructure is, however, very clear: If it is externally sourced and non-domain specific – it’s Platform. It’s written internally, it’s Infrastructure - again, assuming no domain dependencies have crept it.

The distinction between Interface and Application is also fairly clear cut – if a class is directly related to application specific user interface – then it’s Interface code; if some code is there to deal with external system integration e.g. web-services – then it’s Interface; if the code is dealing with parsing an application specific file format, e.g. an application specific XML format, then it’s Interface. But note here, in the same way that user interface code splits into general purpose (Platform) and application specific (Interface), so most XML parsers are Platform. XML itself is domain-neutral – but specific DTDs will require custom Interface code to be written to interpret domain-specific concepts.

The distinction between Application and Domain is sometimes a source of questions. As I said earlier, Application code is there to provide a set of transactional services for Interface code to use. Domain code is there to provide decoupled domain-level abstractions like Book, Account, and so on, which are then combined by Application code to provide application functionality. Without Application packages, decoupled Domain packages would remain forever decoupled – and there would be no application!

The Application/Domain divide is also important from a re-use persepctive – and here I’m talking about re-using Domain code within the single application discusssed. In figure 4 we saw how the DomainAlerts package was customised by the Application layer in two different ways – one to provide text message alerts, the other to provide email alerts. Pushing the common alerts code down into Domain assists achieves re-use through improved code factoring, and also localises most alert related code into the DomainAlert package. This, in turn, will reduce the cost of functional change in the way in which alerts are supposed to operate.

One final motivation for the Application/Domain divide is testability – an example of this is shown in figure 7. Whilst is it both possible and advisable to undertake automated “functional” testing using the service oriented interface provided by the Application stratum, it will difficult, using this approach, to get a high degree of test coverage. Testing domain packages in isolation, as per figure 7, enables you to greatly improve the overall test coverage and hence overall application reliability.

4. Any questions?

- *We do agile development, this seems like big up front design…*

  Design guidance and development process are orthogonal concepts. You might arrive at an architecture by months of forethought, or you can let the ARM assist you in evolving your architecture in an incremental fashion. For the purposes of this article at least, that's up to you!

- *What about the cost (in code terms) of all these strata?*

  The ARM is about packaging. Every line of code you write using the ARM should be absolutely necessary to meet the needs of your customers, the needs of automated testing, and the needs of delivering a high-quality well-factored application that shows intent at a code level. You don't have to write any code *because* of the ARM – you just have to put the right code into the right package!

- *Dependencies seem to go right across the strata, shouldn't there be a rule saying one stratum can only use the stratum below it.*

  No. Consider the ReservationVideoButton in the example above – which inherits directly from the PlatformGUI package. Applying a rule like this, it would be necessary for each of the three intervening strata to provide a wrapper hiding the facilities provided by the stratum below it. To mandate this would be to mandate what is potentially needless complexity.

- *Is Persistence always Infrastructure / Platform?*

  No, not always. It takes some considerable effort and skill to build a general purpose persistence mechanism, and this may be overkill for a single project. In such circumstances, it is not uncommon to write domain-specific persistence mechanisms (sometimes in the form of brokers for various domain classes). This will tend to lead to some code duplication (it's the duplicate code you would ideally factor out into an Infrastructure package) – but may be necessary. It really does depend on circumstances.

- *Can the ARM be used in conjuction with code generation?*

  Code generation, for example generating domain-specific persistence code in the EJB style, is really an orthogonal concern – but it can be the cause of confusion. If you find you are getting confused – take a look at the code that is generated and see where it fits into the ARM.

- *What about vertical sub-division of the application – the ARM only deals with horizontal sub-division, doesn't it?*

That’s right. As I alluded to earlier, there is a 1-to-many relationship between a stratum and the packages it contains. For completeness, you need some sort of guidance in this sub-division. Help is at hand here in the form of two principles of packaging:

- The *Common Closure Principle* or CCP [RMartin], which states: “package things together that change together”. DomainAlerts contains the Alerts, Alert and AlertSender packages precisely because the are highly inter-dependent (highly coupled), and a change to any of them is likely to affect the others.
- The *Common Re-use Principle* or CRP [RMartin], which states: “package things together that are used together”. ApplicationAlertServices (see figure 4) contains three classes which are apparently (from the perspective of the relationships *within* this package) unrelated. A closer inspection, however, reveals that AlertServices.SendNecessaryAlerts method cannot be used *without* supplying one or more concrete AlertSenders – in the example shown the EmailAlertSender and SMSAlertSender classes. QED the common re-use principle applies.

- *What practical steps can I take to apply this model on my project?*

  It’s remarkably simple to use the ARM to keep track of your project’s package structure. All you need is a large whiteboard, which you divide into five horizontal sections as per the stata. Then, draw a package symbol with the package name on the board for each package in your system. Show the dependencies between packages as arrows - all going downwards, of course. This type of diagram is known as a *package map*.

5. Summary

In this article I have presented an architectural reference model that I’ve used successfully on a number of Enterprise Applications. The model and its rules are intended to assist you in improving the structure of your code, in particular to ensure greater clarity of responsibility at the package level, to improve overall code factoring, to reduce duplication within code and to enable you to manage your package dependencies effectively.

Coming full circle to the problems that introduced this article:

- the ARM encourages *repetative code* to be *pushed down* into a lower stata. Repetative Application code into Domain or Infrastructure (depending on whether it has domain dependencies or not); repetative Domain code into Infrastructure, etc.;
- this in turn leads to *less duplication* making functional changes simpler;
- *package structure* is now based on a more coherent set of rules, so it should be easier for new staff to work out what code will live where;
- keeping a package map on the wall means *intelligent work scheduling is easier*;

- *automated testing* and subsequent bug fixing is easier, as dependencies have been brought under greater control, and code factoring has been improved by putting the right code in the right statum;
- the application is now more stable – pushing non-domain specific code down into infrastructure means it won't be affected by functional changes, and pushing repetative Application code into Domain leads to greater localisation of changes in the face of varying functional requirements.

You should find out more about the ARM by looking:

- In Chapter 4 of Extreme Programming Examined (Succi & Marchesi - Addison-Wesley, 2001).
- At www.ratio.co.uk/architectural_reference_model.pdf (covers similar ground but with a different angle of attack and example).
- At www.ratio.co.uk/whitepaper_4.pdf and www.ratio.co.uk/whitepaper_7.pdf – two finance related case studies that use the ARM (the terminology used had changed, but the fundamental concepts remain the same).

6. References and credits

[RMartin] – Granularity, Robert C. Martin - C++ Report, 1996.

About the Author

Mark Collins-Cope has been working in software development for over 20 years. He is author of *Agile Software Development with Iconix Process* – Apress, 2004 (Doug Rosenberg, Matt Stephens, Mark Collins-Cope). Diagrams within this article are taken from the Ratio Group course –*Enterprise Architecture and Advanced OO Design* – see www.ratio.co.uk or email mark (at) ratio.co.uk for further information.

Thanks are due to Hubert Matthews, who worked with me on previous papers on this topic, and to Andrew Vautier and Anders Nestors of Accenture – on whose 1 Million+ line C++ banking project provided much seed material for the concepts discussed here.

Thanks also to Ilja Preuß, who reviewed earlier drafts of this article, and whose comments were invaluable.