White Paper

AGILITY & MODULARITY, TWO SIDES OF THE SAME COIN

www.paremus.com

May 2013

Version 1.0

Contents

1 INTRODUCTION

Agile development methodologies are increasing popular. Yet most '*Agile*' experts and analysts discuss agility in isolation; i.e. without reference to '*Structure*'[1]. Perhaps as a result of this, many organizations attempt to invest in '*Agile*' processes without ever considering the structure of their applications. Yet, these oversights are surprising given that '*Agility*' is an emergent characteristic, or property, of the underlying entity; and for an entity to be '*Agile*' it must have a high degree of structural modularity[2].

Hence the question, '*How might one realize an Agile system?'* should be recast to *'How might one build systems with high degrees of structural modularity?'.*

For this reason, we start by exploring the relationship between structural modularity and agility.

[1] The exception to the rule being Kirk Knoernschild
[2] Diversity & Complexity - Scott Page. ISBN-13**:** 978-0691137674

2 STRUCTURE, MODULARITY & AGILITY

Business Managers and Application Developers face many of the same fundamental challenges. Whether a business, or a software application serving a business, the entity must be cost effective to create and maintain. If the entity is to endure, it must also be able to rapidly adapt to unforeseen changes in a cost effective manner.

If we hope to effectively manage a System, we must first *understand* the System. Once we *understand* a System, manageable *Change* and directed *Evolution* are possible.

Yet we do not need to *understand* all of the fundamental constituents of the System; we only need to understand the *relevant* attributes and behaviors for the level of the *hierarchy* we are responsible for managing.

SERVICES SHOULD BE OPAQUE

From an external perspective, we are interested in the exposed behavior; the type of *Service* provided, and the properties of that *Service*. For example is the *Service* reliable? Is it competitively priced relative to alternative options?

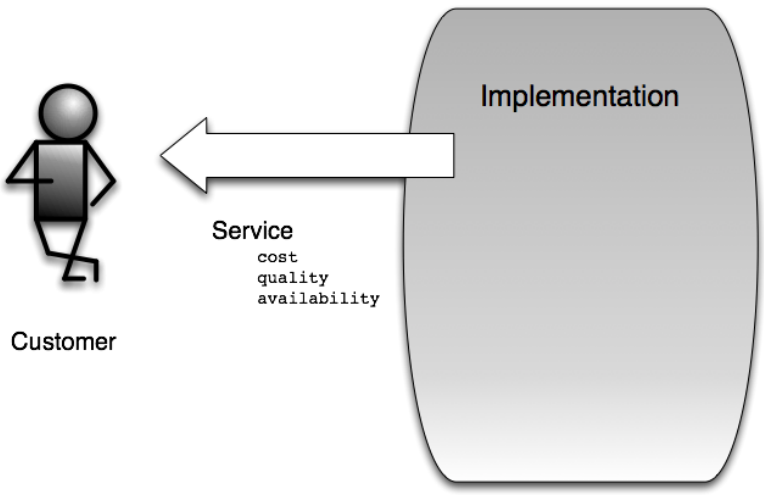


Figure 1: A consumer of a Service.

As a consumer of the *Service* I have no interest in how these characteristics are achieved. I am only interested in the advertised *Capabilities,* which may or may not meet my *Requirements*.

TO MANAGE I NEED TO UNDERSTAND STRUCTURE

Unlike the consumer, the implementation of the *Service* is of fundamental importance to the *Service* provider. To achieve an understanding, we create a conceptual model by breaking the *System* responsible for providing the *Service* into a set of smaller interconnected pieces. This graph of components may represent an '*Organization Chart',* if the entity is a business, or a mapping of the components used, if the entity is a software application.

A first simple attempt to understand our abstract *System* is shown below.

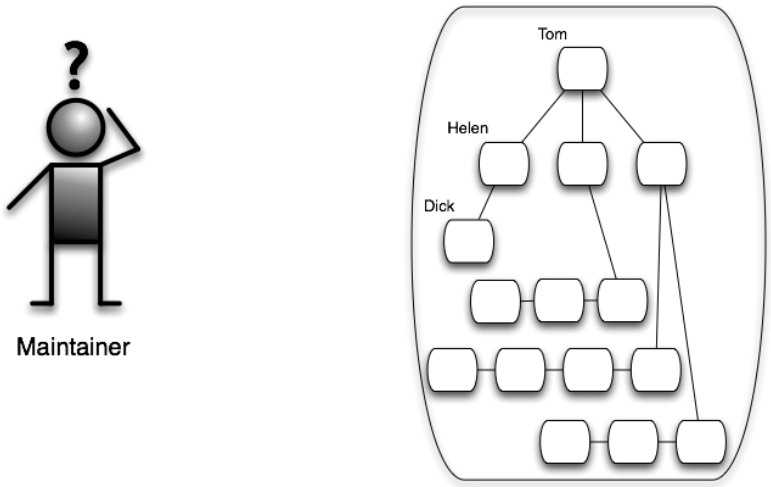


Figure 2: The Service Provider / System Maintainer

From this simple representation we immediately know the following:

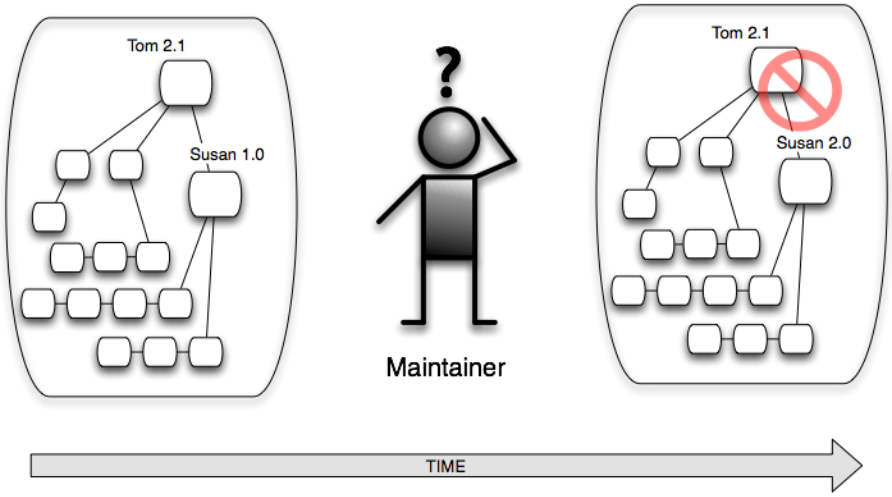
- The *System* is composed of 15 *Components*.
- The names of the *Components*.
- The dependencies that exist between these *Components;* though we don't know why those dependencies exist.
- While we do not know the responsibilities of the individual *Components*; from the degree on inter-connectedness, we can infer that component '*Tom*' is probably more important than '*Dick*'.

It is important to note that we may not have created these *Components* and/or we may have no understanding of their internal construction. Just as the consumers of our *Service* are interested in the *Capabilities* offered, we, as a *consumer* of these components, simply *Require* their *Capabilities*.

REQUIREMENTS & CAPABILITIES

At present, we have no idea why the dependencies exist between the *Components* just that those dependencies exist. Also, this is a time independent view. What about change over time?

One might initially resort to using *versions* or *version ranges* with the named entities; changes in the structure indicated by version changes on the constituents. However, as shown in figure 3, *versioned names,* while indicating change, fail to explain why `Susan 1.0` can work with `Tom 2.1`, but `Susan 2.0` cannot!



The earlier System functioned correctly, the later System, with an upgraded Component fails.

Why is this?

Figure 3: How do we track structural change over time?

It is only when we look at the *Capabilities* and *Requirements* of the entities involved that we understand the issue. `Tom 2.1` *Requires* a `Manager` *Capability,* a capability that can be provided by `Susan 1.0`. However, at the later point in time `Susan 2.0`, having reflected upon her career, decided the retrain. `Susan 2.0` now no longer advertises a `Manager` *Capability*, but instead advertises a `Plumber 1.0` *Capability*.

This simple illustration demonstrates that dependencies need to be expressed in-terms of *Requirements* and *Capabilities* of the participating entities and not their names[3]. These descriptions should also be intrinsic to the entities; i.e. components should be *self-describing*[4].

[3] The Apache Maven project has recently (2013) discussed adoption of version ranges - http://maven.apache.org/enforcer/enforcer-rules/versionRanges.html - for artifact names. While an improvement the concept is still flawed as the dependencies are still described in terms of the entities names.

[4] The *Requirements*, *Capabilities* and dependencies might be documented somewhere, but overtime these explanations will become dated; i.e. the System has changed since the original documentation was produced and this documentation not updated.

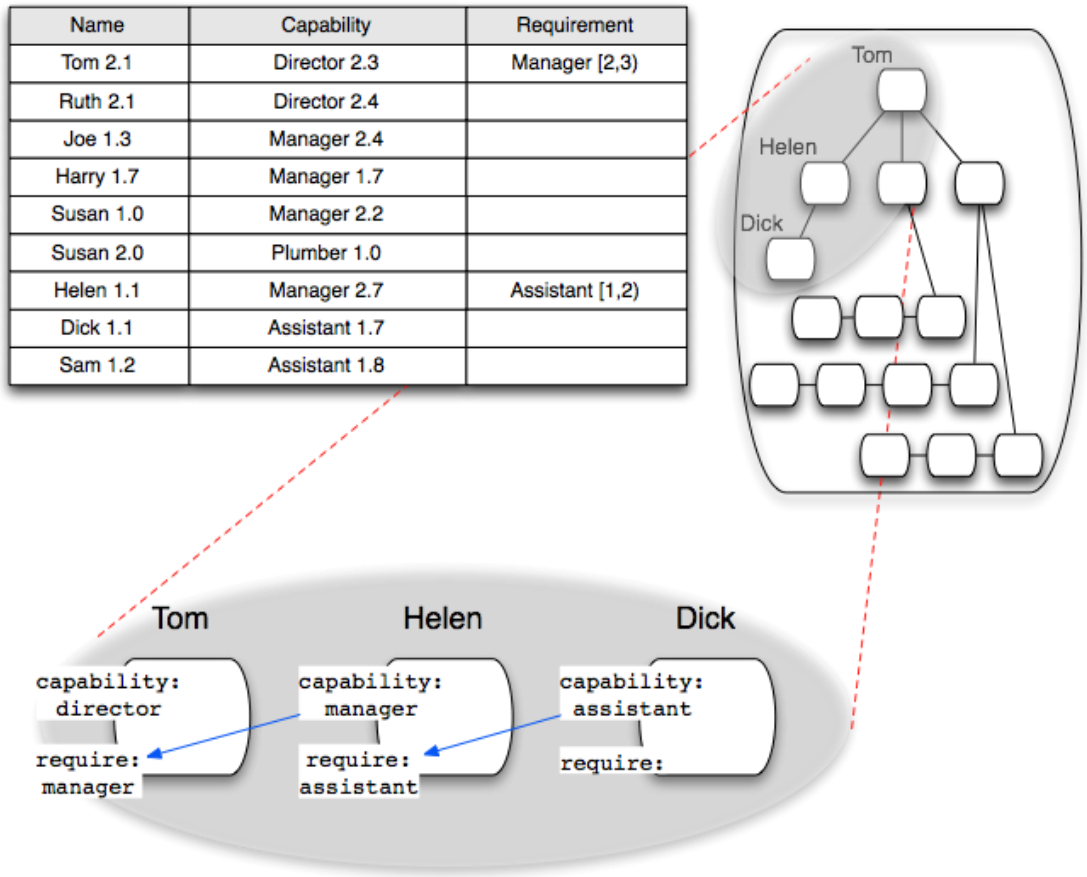


Figure 4: An Organizational Structure: Effectiveness of versioned Names, Capabilities / Requirements and the use of Semantic versioning.

As shown, we can completely describe the System in terms of *Requirements* and *Capabilities*, without referencing specific named entities.

EVOLUTION AND THE ROLE OF SEMANTIC VERSIONING

Capabilities and *Requirements* are now the primary means via which we understand the structure of our *System*. However we are still left with the problem of understanding change over time.

- In an organization chart; to what degree are the dependencies still valid if an employee is promoted (*Capabilities* enhanced)?
- In a graph of interconnected software components; to what degree are the dependencies still valid if we refactor one of the components (changing / not changing a public interface)?

By applying simple versioning we can see that changes have occurred; however we do not understand the impact of these changes. However, if instead of simple versioning, semantic versioning is used (see http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf), the potential impact of a change can be communicated.

This is achieved in the following manner:

- *Capabilities* are versioned with a `major.minor.micro` versioning scheme. In addition, we collectively agree that - `minor` or `micro` version changes represent non-breaking changes; e.g. `2.7.1` ➞ `2.8.7`. In contrast `major` version changes; e.g. `2.7.1.` ➞ `3.0.0.` represent breaking changes which may affect the users of our component.
- *Requirements* are now specified in terms of a range of acceptable *Capabilities*. Square brackets (`'['` and `']'`) are used to indicate inclusive and parentheses (`'('` and `')'`) to indicate exclusive. Hence a range `[2.7.1, 3.0.0)` means any *Capability* with version at or above `2.7.1` is acceptable up to, but not including `3.0.0`.

Using this approach we can see that if *Joe* is substituted for *Helen, Tom's Requirements* are still met. However *Harry*, while having a *Manager Capability*, cannot meet *Tom's Requirements* as *Harry's* `1.7` skill set is outside of the acceptable range for Tom i.e. `[2,3)`.

Via the use of semantic versioning the impact of change can be communicated. Used in conjunction with *Requirements* and *Capabilities* we now have sufficient information to be able to substitute components while ensuring that all the structural dependencies continue to be met.

Our job is almost done. Our simple System is *Agile* & *Maintainable*!

AGILE - ALL THE WAY DOWN

The final challenge concerns complexity. What happens when:

a) the size and sophistication of the System increases?

b) there are an increased number of components and a large increase in inter-dependencies?

The reader having already noticed a degree of self-similarity [5] arising in the previous examples may have already guessed the answer.

[5] http://en.wikipedia.org/wiki/Self-similarity

The *Consumer* of our *Service* selected our *Service* because the advertised *Capabilities* met the consumers *Requirements* (see Figure 1). The implementation of the *System* which provided this *Service* is masked from the consumer. This pattern is once again repeated one layer down. The *System's* structure is itself described in-terms of the *Capabilities* and *Requirements* of the participating components (see Figure 4). This time, the internal structure of the components are masked from the *System*, as shown in Figure 5; this pattern may be repeated at many logical layers.

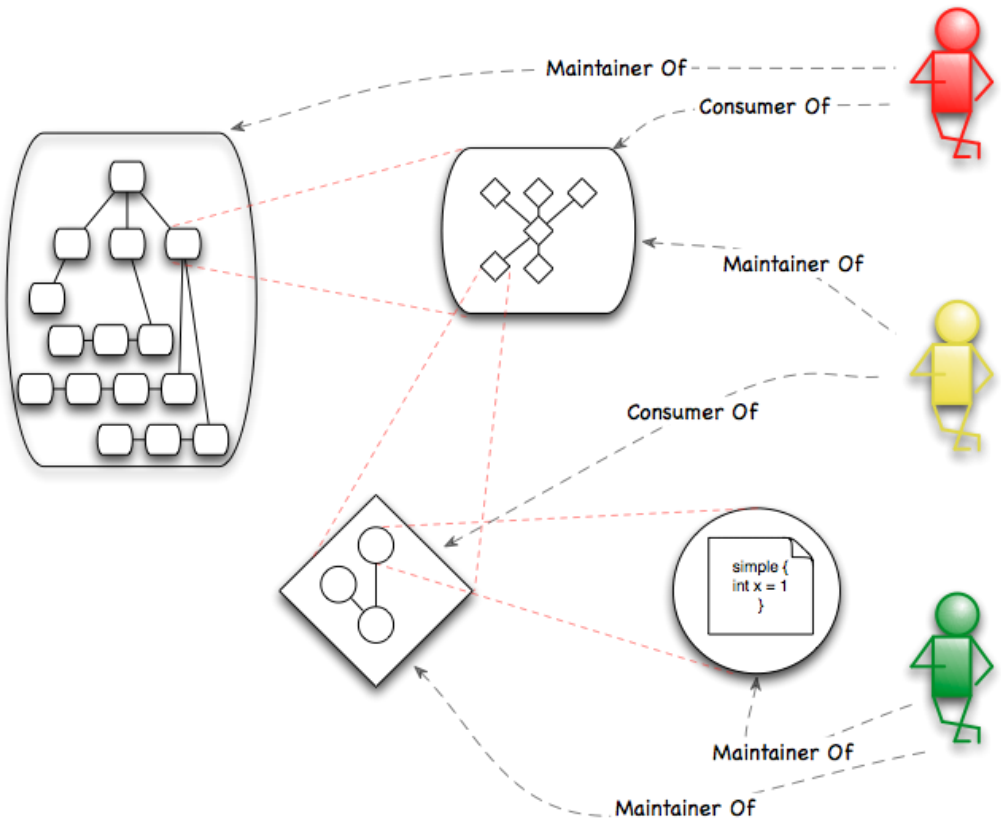


Figure 5: An Agile Hierarchy: Each layer only exposes necessary information. Each layer is composite with the dependencies between the participating components expressed in-terms of their *Requirements* and *Capabilities*.

All truly *agile* systems are built this way, consisting of a hierarchy of structural layers. Within each structural layer the components are self-describing: self-describing in terms of information relevant to that layer, with unnecessary detail from the lower layers masked.

This pattern is repeated again and again throughout natural and man-made systems. Natural ecosystems build massive structures from nested hierarchies of modular components. For example:

- The Organism
- The Organ
- The Tissue
- The Cell

For good reason, commercial organizations attempt the same structures:

- The Organization
- The Division
- The Team
- The Individual

Hence we might expect a complex *Agile* software system to also mirror these best practices:

- The Business Service
- Coarse grained business components
- Fine grained micro-Services
- Code level modularity

This process started in the mid/late 1990's as organizations started to adopt coarse grain modularity as embodied by Service Oriented Architectures (SOA) and Enterprise Service Buses (ESB's). These approaches allowed business applications to be loosely coupled; interacting via well-defined service interfaces or message types. SOA advocates promised more '*Agile*' IT environments as business systems would be easier to upgrade and/or replace.

However, in many cases the core applications never actually changed. Rather the existing application interfaces were simply exposed as *SOA Services*. When viewed in this light it is not surprising that SOA failed to deliver the promised cost savings and business agility: http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html.

Because of the lack of internal modularity, each post-SOA application was as inflexible as its pre-SOA predecessor.

TO BE AGILE?

We conclude this section with a brief summary of the arguments developed so far.

To be 'Agile' a *System* will exhibit the following characteristics:

- A *Hierarchical* Structure: The System will be hierarchical. Each layer composed from components from the next lower layer.
- *Isolation*: For each structural layer; strong isolation will ensure that the internal composition of each participating component will be masked.
- *Abstraction*: For each layer; the behavior of participating components is exposed via stated *Requirements* and *Capabilities*.
- *Self-Describing*: Within each layer the relationship between the participating components will be self-describing; i.e. dependencies will be defined in terms of published *Requirements* and *Capabilities*.
- *Impact of Change:* Via semantic versioning the impact of a change on dependencies can be expressed.

Systems built upon these principles are:

- Understandable: The System's structure may be understood at each layer in the structural hierarchy.
- Changeable: At each layer in the hierarchy, structural modularity ensures that changes remains localized to the affect components; the boundaries created by strong structural modularity shielding the rest of the System from these changes.
- Evolvable: Within each layer components may be substituted, therefore the system supports diversity and is evolvable.

The System achieves Agility through structural modularity.

3 BUT WE ARE ALREADY MODULAR!

Most developers appreciate that applications should be modular. However, whereas the need for *logical* modularity was rapidly embraced in the early years of Object Orientated programming (see http://en.wikipedia.org/wiki/Design_Patterns), it has taken significantly longer for the software industry to appreciate the importance of structural modularity; especially the fundamental importance of structural modularity with respect to increasing application maintainability & agility and controlling / reducing environmental complexity.

JUST A BUNCH OF JARS

In *Java Application Architecture (see http://techdistrict.kirkk.com)* Kirk Knoernschild explores structural modularity and develops a set of best practice structural design patterns. As Knoernschild explains, no modularity framework is required to develop in a modular fashion; for Java the JAR is sufficient.

Indeed, it is not uncommon for '*Agile'* development teams to break an application into a number of smaller JAR's as the code-base grows. As JAR artifacts increase in size, they are broken down into collections of smaller JAR's. From a code perspective, especially if Knoernschild's structural design patterns have been followed, one would correctly conclude that - at one structural layer - the application is modular.

BUT IS IT '*AGILE*' ?

From the perspective of the team that created the application, and who are subsequently responsible for its on-going maintenance, the application is more *Agile*. The team understand the dependencies and the impact of change. However, this knowledge is not explicitly associated with the components. Should team members leave the company, the application and the business are immediately compromised. Also, for a third party (e.g. a different team within the same organization), the application may as well have remained a monolithic code-base.

While the application has one layer of structural modularity - it is not self-describing. The metadata that describes the inter-relationship between the components is absent; the resultant business system is intrinsically *fragile*.

WHAT ABOUT MAVEN?

Maven artifacts (*Project Object Model - POM*) also express dependencies between components. These dependencies are expressed in-terms of the component names. For this reason, a Maven based modular application can be simply assembled by any third party.

However, as we already know, the value of name based dependencies is limited. As the dependencies between the components are not expressed in terms of *Requirements* and *Capabilities,* third parties are unable to deduce why the dependencies exist and what might be substitutable.

The application can be assembled, but it cannot be changed. It is debatable whether the application is more '*Agile*' than the previous '*Bunch of JAR's'* approach

THE NEED FOR OSGI

As Knoernschild demonstrates in his book *Java Application Architecture,* once structural modularity is achieved it is trivially easy to move to OSGi - *the modularity standard for Java.*

OSGi expresses dependencies in terms of *Requirements* and *Capabilities.* It is therefore immediately apparent to a third party which components may be interchanged. As OSGi also uses semantic versioning, it is immediately apparent to a third party whether a change to a component is potentially a breaking change.

> Kirk Knoernschild – Java Application Architecture
>
> Not only does OSGi help us to enforce structural modularity, it provides the necessary metadata to ensure that the modular Structures we create are also Agile structures.

OSGi also has a key part to play with respect to structural hierarchy.

At one end of the modularity spectrum we have *Service Oriented Architectures,* at the other end of the spectrum we have Java *Packages* and *Classes*. However, as explained by Knoernschild, essential layers are missing between these two extremes.

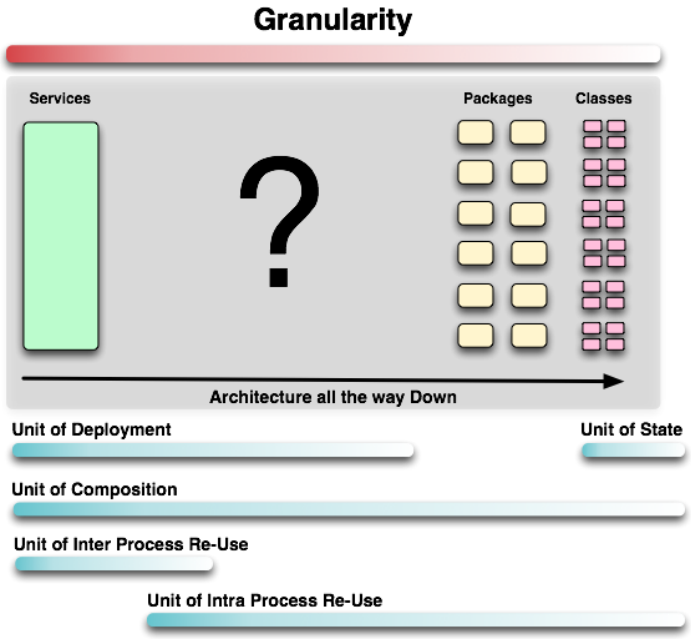


Figure 6: Structural Hierarchy: The Missing Middle

The problem, this missing middle, is directly addressed by OSGi.

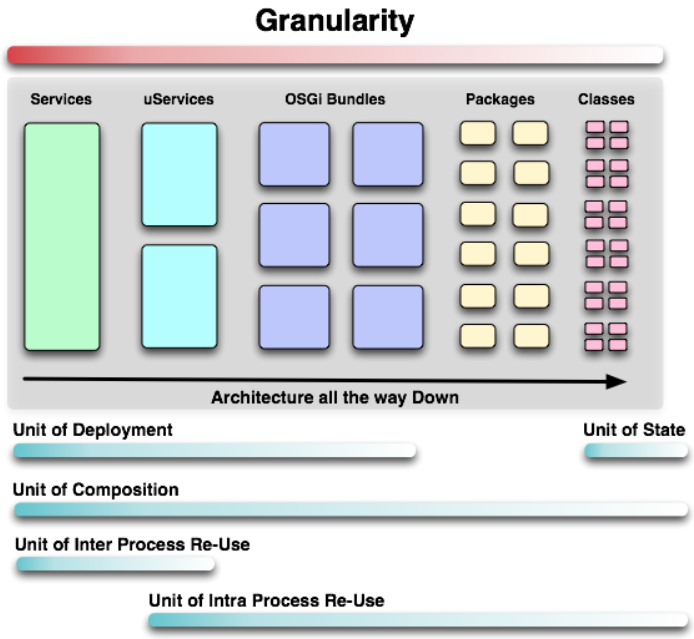


Figure 7: Structural Hierarchy: OSGi Services and Bundles

As explained by Knoernschild the modularity layers provided by OSGi address a number of critical considerations:

- *Code Re-Use:* Via the concept of the OSGi Bundle, OSGi enables code re-use.
- *Unit of Intra / Inter Process Re-Use:* OSGi Services are light-weight Services that are able to dynamically find and bind to each other. OSGi Services may be collocated within the same JVM, or via use of an implementation of OSGi's remote service specification, distributed across JVM's separated by a network. Coarse grained business applications may be composed from a number of finer grained OSGi Services.
- *Unit of Deployment:* OSGi bundles provide the basis for a natural unit of deployment, update & patch.
- *Unit of Composition:* OSGi bundles and Services are essential elements in the composition hierarchy.

Hence OSGi *bundles* and *services,* backed by the OSGi Alliance's open specifications, provide Java with essential - and previously missing - layers of structural modularity. In principle, OSGi technologies enable Java based business systems to be '*Agile - All the Way Down*!'

As we will now see, the OSGi structures (*bundles and services*) map well to, and help enable, popular *Agile Methodologies*.

4 EMBRACING *AGILE*

The Agile Movement focuses on the '*Processes'* required to achieve Agile product development and delivery. While a spectrum of *Lean & Agile* methodologies exist, each tends to be a variant of, a blend of, or an extension to, the two best known methodologies; namely *Scrum* and *Kanban* - http://en.wikipedia.org/wiki/Lean_software_development.

As will be shown, to be effective, each of these approaches requires some degree of structural modularity.

SCRUM

Customers change their minds. Scrum acknowledges the existence of '*requirement churn'* and adopts an empirical (http://en.wikipedia.org/wiki/Empirical) approach to software delivery. Accepting that the problem cannot be fully understood or defined up front. Scrum's focus is instead on maximizing the team's ability to deliver quickly and respond to emerging requirements.

Scrum is an iterative and incremental process, with the '*Sprint*' being the basic unit of development. Each *Sprint* is a "time-boxed" (http://en.wikipedia.org/wiki/Timeboxing) effort, i.e. it is restricted to a specific duration. The duration is fixed in advance for each Sprint and is normally between one week and one month. A Sprint is preceded by a planning meeting, where the tasks for the Sprint are identified and an estimated commitment for the Sprint goal is made. This is followed by a review or retrospective meeting, where the progress is reviewed and lessons for the next Sprint are identified.

During each Sprint, the team creates finished portions of a product. The set of features that go into a Sprint come from the product *backlog*, which is an ordered list of requirements (http://en.wikipedia.org/wiki/Requirement).

Scrum attempts to encourage the creation of self-organizing teams, typically by co-location of all team members, and verbal communication between all team members.

KANBAN

'Kanban' originates from the Japanese word "signboard" and its roots trace back to Toyota, the Japanese automobile manufacturer in the late 1940's (see http://en.wikipedia.org/wiki/Kanban).

Kanban encourages teams to have a shared understanding of work, workflow, process, and risk; so enabling the team to build a shared comprehension of a problems and suggest improvements which can be agreed by consensus.

From the perspective of structural modularity, Kanban's focus on *Work-In-Progress (WIP)*, *Limited Pull* and *Feedback* are probably the most interesting aspects of the methodology:

1. Work-In-Process (WIP) should be limited at each step of a multi-stage workflow. Work items are "pulled" to the next stage only when there is sufficient capacity within the local WIP limit.
2. The flow of work through each workflow stage is monitored, measured and reported. By actively managing '*flow*', the positive or negative impact of continuous, incremental and evolutionary changes to a System can be evaluated.

Hence Kanban encourages small continuous, incremental and evolutionary changes. As the degree of structural modularity increases, pull based flow rates also increase while each smaller artifact spends correspondingly less time in a WIP state.

THE AGILE MATURITY MODEL

Both Scrum and Kanban's objectives become easier to realize as the level of structural modularity increases.

Fashioned after the *Capability Maturity Model* (see http://en.wikipedia.org/wiki/Capability_Maturity_Model), which allows organizations or projects to measure their improvements on a software development process; the *Modularity Maturity Model* is an attempt to describe how far along the modularity path an organization or project might be[6]. We extend this analysis further calling out the impact of an organization's level of *Modularity Maturity* on an organization's *Agility.*

Keeping in step with the *Modularity Maturity Model we* refer to the following six levels.

Ad Hoc - No formal modularity exists. Dependencies are unknown. Java applications have no, or limited, structure. In such environments it is likely that *Agile Management Processes* will fail to realize business objectives.

Modules - Instead of classes (or JARs of classes), named modules are used with explicit versioning. Dependencies are expressed in terms of module identity (including version). Maven, Ivy and RPM are examples of modularity solutions where dependencies are managed by versioned identities. Organizations will usually have some form of artifact repository; however the value is compromised by the fact that the artifacts are not self-describing in terms of their *Capabilities* and *Requirements.*

[6] The *Modularity Maturity Model*; this proposed by Dr Graham Charters at the OSGi Community Event 2011 (see http://slidesha.re/ZzyZ3H)

This level of modularity is perhaps typical for many of today's in-house development teams. Agile processes such are Scrum are possible, and do deliver some business benefit. However ultimately the effectiveness & scalability of the Scrum management processes remain limited by deficiencies in structural modularity; for example *Requirements* and *Capabilities* between the Modules usually being verbally communicated. The ability to realize *Continuous Integration* (CI) is again limited by ill-defined structural dependencies.

Modularity - Module identity is not the same as true modularity. As we've seen Module dependencies should be expressed via contracts (i.e. *Capabilities* and *Requirements*), not via artifact names. At this point, dependency resolution of *Capabilities* and *Requirements* becomes the basis of a dynamic software construction mechanism. At this level of structural modularity dependencies will also be semantically versioned.

With the adoption of a modularity framework like OSGi the scalability issues associated with the Scrum process are addressed. By enforcing encapsulation and defining dependencies in terms of *Capabilities* and *Requirements*, OSGi enables many small development teams to efficiently work independently and in parallel. The efficiency of Scrum management processes correspondingly increases. *Sprints* can be clearly associated with one or more well defined structural entities i.e. development or refactoring of OSGi bundles. Meanwhile Semantic versioning enables the impact of refactoring to be efficiently communicated across team boundaries. As the OSGi bundle provides strong modularity and isolation, parallel teams can safely *Sprint* on different structural areas of the same application.

Services - Services-based collaboration hides the construction details of services from the users of those services; so allowing clients to be decoupled from the implementations of the providers. Hence, *Services* encourage loose-coupling. OSGi Services dynamically *find* and *bind* behaviors directly enable loose-coupling, and enable the dynamic formation, or assembly of, composite applications. Perhaps of greater importance, *Services* are the basis upon which runtime *Agility* may be realized; including rapid enhancements to business functionality, or automatic adaption to environmental changes.

Having achieved this level of structural modularity an organization may simply and naturally apply *Kanban* principles and achieve the objective of *Continuous Integration*.

Devolution - Artifact ownership is devolved to modularity aware repositories which encourage collaboration and enable governance. Assets may be selected on their stated *Capabilities*. Advantages include:

- Greater awareness of existing modules
- Reduced duplication and increased quality
- Collaboration and empowerment
- Quality and operational control

As software artifacts are described in terms of a coherent set of *Requirements* and *Capabilities*, developers can communicate changes (breaking and non-breaking) to third parties through the use of semantic versioning. Devolution allows development teams to rapidly find third-party artifacts that meet their *Requirements*. Hence *Devolution* enables significantly flexibility with respect to how artifacts are created, allowing distributed parties to interact in a more effective and efficient manner. Artifacts may be produced by other teams within the same organization, or consumed from external third parties. The *Devolution* stage promotes code re-use and efficient, low risk, out-sourcing, crowd-sourcing and in-sourcing of the artifact creation process.

Dynamism - This level builds upon Modularity, Services & Devolution and is the culmination of our Agile journey.

- Business applications are rapidly assembled from modular components.
- As strong structural modularity is enforced (isolation by the OSGi bundle boundary), components may be efficiently and effectively created and maintained by a number of small - *on-shore*, *near-shore* or *off-shore* development teams.
- As each application is self-describing, even the most sophisticated of business systems is simple to understand, to maintain, and to enhance.
- As semantic versioning is used; the impact of change is efficiently communicated to all interested parties, including Governance & Change Control processes.
- Software fixes may be hot-deployed into production - without the need to restart the business system.
- Application capabilities may be rapidly extended and applied, also without needing to restart the business system.

Finally, as the dynamic assembly process is aware of the *Capabilities* of the hosting runtime environment, application structure and behavior may automatically adapt to location; allowing transparent deployment and optimization for public Cloud or traditional private datacenter environments.

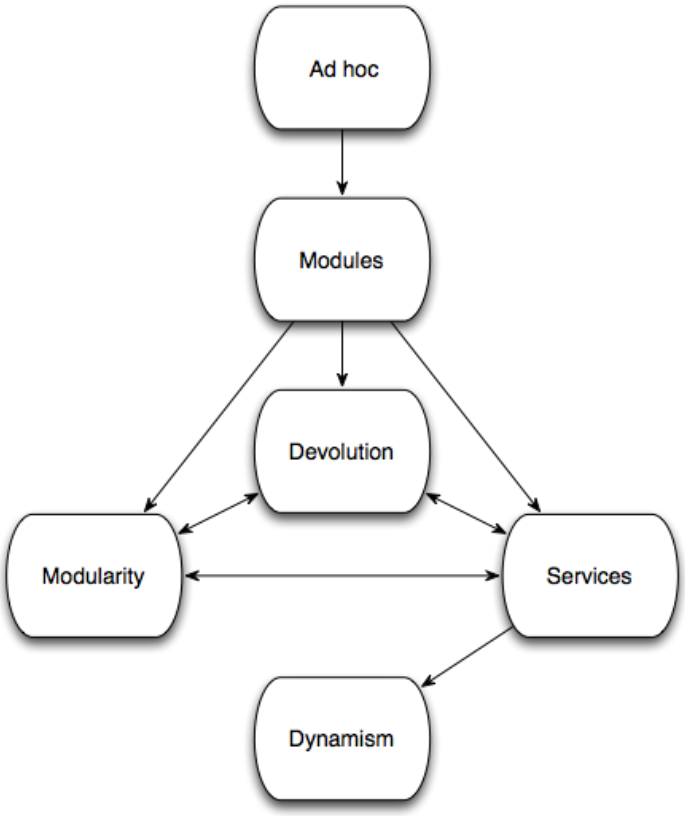


Figure 8: Modularity Maturity Model

An organization's *Modularization Migration* strategy will be defined by the approach taken to traversing these Modularity levels. Most organizations will have already moved from an initial Ad Hoc phase to Modules. Meanwhile organizations that value a high degree of *Agility* will wish to reach the endpoint; i.e. Dynamism. Each organization may traverse from Modules to Dynamism via several paths, adapting migration strategy as necessary.

- To achieve maximum benefit as soon as possible; an organization may choose to move directly to Modularity by refactoring the existing code base into OSGi bundles. The benefits of Devolution and Services naturally follow. This is also the obvious strategy for new greenfield applications.
- For legacy applications an alternative may be to pursue a Services first approach; first expressing coarse grained software components as *OSGi Services*; then driving code level modularity (i.e. *OSGi bundles*) on a *Service* by *Service* basis. This approach may be easier to initiate within large organizations with extensive legacy environments.

- Finally, one might move first to limited Devolution by adopting OSGi metadata for existing artifacts. Adoption of *Requirements* and *Capabilities,* and the use of semantic versioning, will clarify the existing structure and impact of change to third parties. While structural modularity has not increased the move to Devolution it positions the organization for subsequent migration to the Modularity and Services levels.

A diverse set of choices and the ability to pursue these choices as appropriate, is exactly what one would hope for, and expect from, an increasingly *Agile* environment!

5 AGILITY & CONTINUOUS INTEGRATION - AN OSGI USE CASE

Siemens Corporate Technology Research group is comprised of a number of engineers with diverse skills spanning computer science, mathematics, physics, mechanical engineering and electrical engineering. The group provides solutions to Siemens business units based on neural network technologies and other machine learning algorithms. As Siemens' business units require working examples rather than paper concepts, Siemens Corporate Technology Research are required to rapidly prototype potential solutions for their business units.

In a presentation entitled '*Workflow for Development, Release and Versioning with OSGi / Bndtools: Real World Challenges'* at the 2012 OSGi Community Event (http://www.osgi.org/CommunityEvent2012/Schedule), the Siemens team presented these business drivers and the solution undertaken to achieve a highly Agile OSGi based *Continuous Integration* environment.

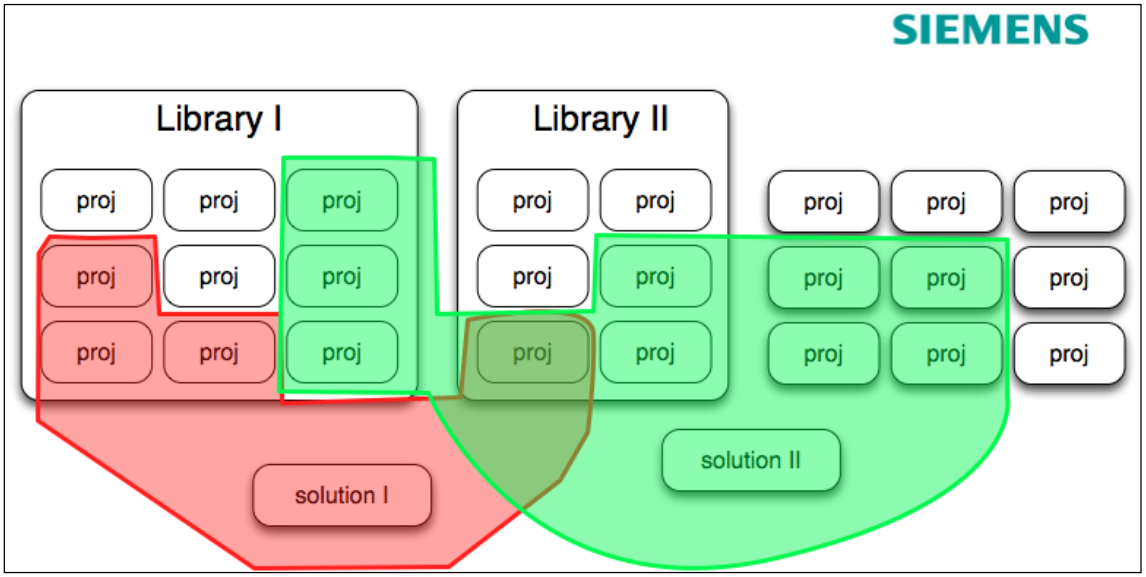


Figure 9: Siemens' Product Repository

The Siemens' team had the following objectives:

1. *Build Repeatability*: To ensure that old versions of products can always be rebuilt from exactly the same set of sources and dependencies, even many years in the future. This would allow Siemens to continue supporting multiple versions of released software that have gone out to different customers.
2. *Reliable Versioning*: By using OSGi Semantic Versioning, in conjunction with a build process that assures the versions in released artifacts are always correct with respect to those semantics, Siemens could quickly and reliably assemble a set of components (including their own software along with third party and open source) and have a high degree of confidence that they will all work together.
3. *Full Traceability*: the software artifacts that are released are always exactly the same artifacts that were tested by QA, and can be traced back to their original sources and dependencies. There is no necessity to rebuild in order to advance from the testing state into the released state.

To achieve rapid prototyping the Siemens' required a repository of software components, including a generic framework and an *algorithm toolbox*. OSGi was chosen as the enabling modularity framework, this decision was based upon the maturity of OSGi technology, the open industry specifications which underpin OSGi implementations, and the technology governance provided by the OSGI Alliance.

The delivered solution provided a consistent approach to application launching and configuration. Specific business functionality was dynamically determined via automated dependency tree resolution and loading of the required OSGi bundles.

From a *Continuous Integration* perspective, the required environment needed to:

- work with standard developer tooling i.e. Java with Eclipse,
- have strong support for OSGi,
- support the concept of multiple repositories and
- provide a basis for *Continuous Integration* (via Jenkins) to build, test and create deployable artifacts.

For these reasons the Eclipse / Bndtools project was selected.

The solution delivered by the team enabled developers to maintain their own local workspace and check results into their local SVN source repository (see figure 10 overleaf). The SVN repository only contains work in progress (WIP). The Jenkins *Continuous Integration* server builds, tests and pushes the resultant OSGi artifacts to a shared read-only Development OBR. Meanwhile the *Bndtools* based OSGi tooling / workflow allows Siemens' developers to access one or more *OSGi Bundle Repositories* (a.k.a *OBR*); including local *Development* and / or approved Third Party repositories.

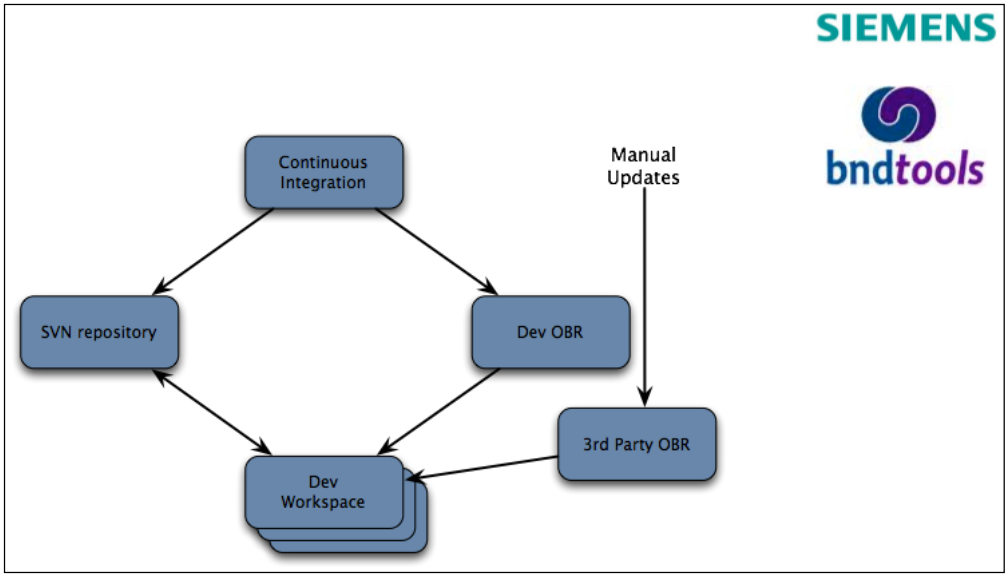


Figure 10: Siemens' OSGi / Bndtools based *Continuous Integration* Environment

The *Continuous Integration* process can collect artifacts from the Development OBR for subsequent UAT testing and deployment.

Finally, this strategy enabled non-developers to be able to rapidly assemble new solutions from the available self-describing modules - including the ability to re-assemble older projects. Hence, in addition, to achieving a highly Agile *Continuous Integration* environment for Siemens' developers, the business objectives were also met.

6 CONCLUSION

Reflect upon the following. There would be little point in implementing Kanban methodologies in the pre-1900's Automobile Industry (http://en.wikipedia.org/wiki/Ford_Model_T) as this was before the innovation of *'the production line',* which in-turn was entirely dependent upon a more fundamental innovation, the idea of modular assembly. In the same way, the Software Industry's *Agile* goals will only be fully realized if the underlying software products have a high degree of structurally modularity. If the product is structurally monolithic and change resistant, then no amount of *Agile process* will change this.

Agile methodologies already warn us against the use of rigid processes. As explained by Taleb in *AntiFragile* [7], '*Predictive*' & '*Prescriptive*' processes stand a high chance of failure as the large number of rigid interdependencies mean that 'Black Swan' events, by their nature unpredictable, are much more likely. What is frequently overlooked is that the same arguments apply to rigid monolithic software!

Key take away

The relationship between structural modularity and Agility is fundamental.

Structural modularity provides the necessary foundations upon which *Agile* systems may be built. It is no coincidence that the levels described in the *Modularity Maturity Model* are similar to the thought process followed earlier in this paper concerning the building of an *Agile* organization. It is also no accident that OSGi, *the modularity system for Java*, is the way it is. The design of OSGi is a logical consequence of the challenges of building highly *modular,* and therefore highly *Agile,* Java software systems.

With OSGi both Java developers and management have a powerful ally, an ally that provides the foundations upon which businesses can realize their *Agile* goals.

[7] AntiFragile: How to Live in a World we Don't Understand - Nassim Taleb. ISBN-13: 978-1846141560

7 FURTHER READING & ABOUT THE AUTHOR

FURTHER READING

http://www.theserverside.com/feature/Successful-modularity-depends-on-your-dependency-model

ABOUT THE AUTHOR

Richard Nicholson, as Paremus CEO and Founder, has actively driven the business and technical direction of Paremus since its formation in 2001. Paremus created the industry's first distributed OSGi Cloud runtime in 2005 - and has been actively promoting the fundamental importance of structural modularity in advanced distributed / Cloud based systems since that point. A Board member of the OSGi Alliance since 2010; Richard was OSGi Alliance President for the period from 2011 to 2013.

Richard maintains keen interest in a number of research areas including Recovery Oriented techniques and Complex Adaptive Systems and the application of such concepts to next generation distributed system design.

Prior to founding Paremus, Richard headed the European System Engineering function for Salomon Smith Barney/Citigroup. Richard graduated from Manchester University with Honors in Physics and went on to gain an Astrophysics doctorate from the Royal Greenwich Observatory.

Richard's blog can be found at: *http://adaptevolve.paremus.com and on the Paremus blogs at http://blogs.paremus.com.*