Agility and Modularity: Two Sides of the Same Coin

Technical Whitepaper

May 2014

Introduction

Agile development methodologies are increasingly popular. Yet while the *social* and *process* aspects of "agile" are discussed at length and practiced by an increasing number of organizations, there is much less appreciation of the fundamental importance of a modular code base[1]. Many organizations invest heavily in agile processes without ever considering the structure of their applications. For this reason, many agile initiatives fail to fully deliver the expected business benefits.

This oversight is especially surprising if one considers that, from the scientific perspective,[2] "agility" is an emergent characteristic: meaning a property that results from underlying structure of the entity. For an entity to be "agile" it must have *a high degree of structural modularity*.[3]

Hence the agility question needs be recast from "How do I build agile business systems?" to "How do I build highly modular business systems?"

This paper investigates the relationship between structural modularity and agility and explains how OSGi™, the open industry standard for Java modularity, provides the necessary foundation upon which the next generation of highly agile business system should be built.

[1] The exception to the rule being Kirk Knoernschild

[2] The study of Complex Adaptive System

[3] Diversity & Complexity - Scott Page. ISBN-13: 978-0691137674

Abstraction

Business managers and application developers face many of the same fundamental challenges. Whether the entity is a business unit, or a software application serving that business unit, the entity must be cost-effective to create and to subsequently maintain. If the entity is to endure, it must be able to adapt to unforeseen changes in a cost-effective manner.

The Customer

From an external perspective, as a consumer we are interested in the service(s) offered by the entity and, if several equivalent services are available, how their properties compare. Is a service reliable? Is it a green/ethical service? Is the service geographically local to me, and/or in the correct jurisdiction? Is it competitively priced?

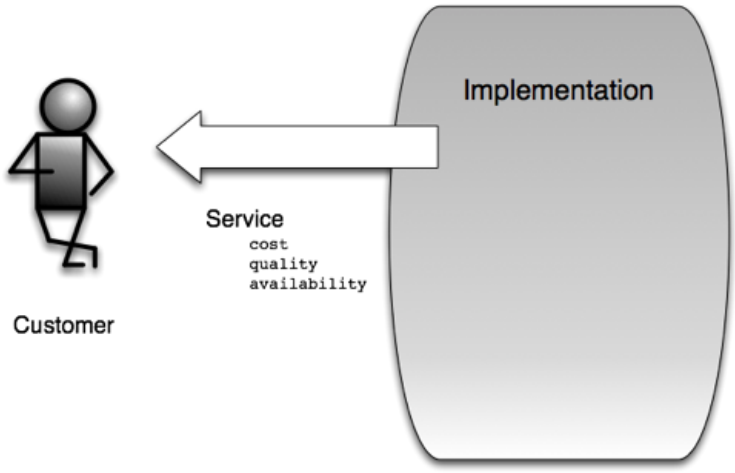

Figure 1: A consumer of a service.

As a consumer, as long as the selected service honors its description, I have no interest in the internal implementation details.

> As the consumer of a service I am interested in the advertised *capabilities* of a service. Do these capabilities meet my *requirements*?

The Service Provider

As the service provider (a business unit or an application), the internal implementation details are fundamental.

Let us consider a modern application. Today, an application is no longer a single block of monolithic code deployed to a single large computer, but rather a set of small, interconnected software components that may span many physical or virtual computing resources, (the reason for this shift will become increasingly apparent as the reader continues). To meet service availability targets, this structure must be maintained. To meet new business objectives, the structure needs to be adapted. To achieve each of these, the structure must be understood.

This graph of interconnected software components is similar to the classic business organizational chart.

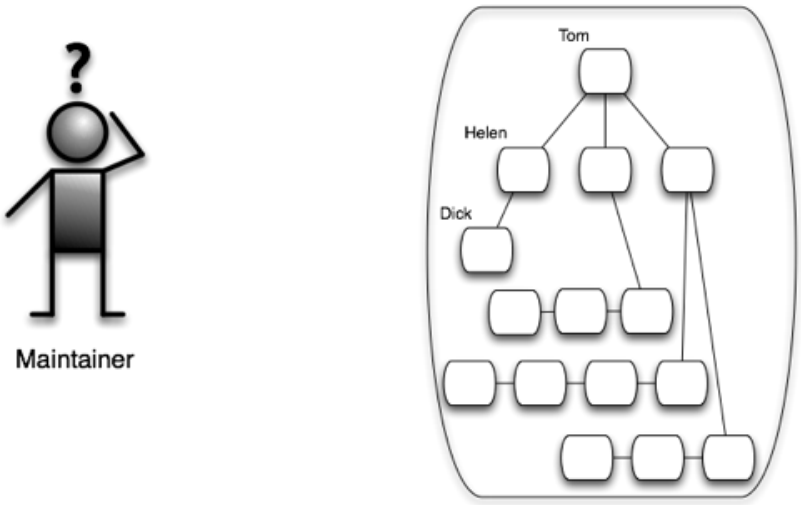

Figure 2: The service provider/system maintainer

Org charts allow us to quickly gain a basic understanding of the structure of a business unit. For example, from Figure 2 we can deduce that:

- The *entity* is composed of 15 components.
- We know the names of the components.

- We also know 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 of inter-connectedness, we can infer relative importance. For example, component *Tom* is probably more critical than *Dick*.

Note that while we are responsible for the management and maintenance of this entity (the business unit or the application), it is unlikely that we created many, or indeed any, of the individual components used. Just as our customer is primarily interested in the capabilities of the service we offer, with no understanding of the implementation, we likewise have little or no understanding of the internal construction of the components we use. We simply require their capabilities.

Requirements & Capabilities

While we know dependencies exist, we have no idea as to why those specific dependencies exist. Also, if nodes are changed, how might this change affect those dependencies, the overall structure, and the resultant service to our customers?

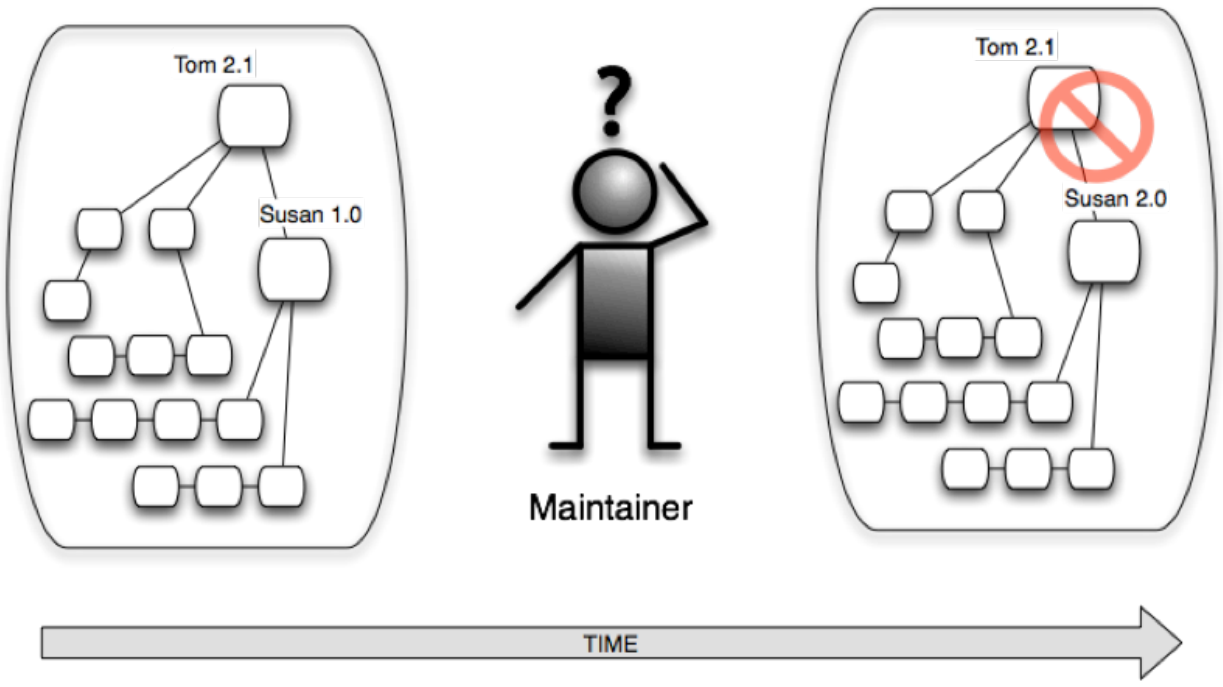

Figure 3: How do we track structural change over time? The earlier system functioned correctly; the later system - with an upgraded component - fails. Why is this?

With respect to managing change, one might initially resort to *versioning* the node names. Changes in the structure would be indicated by version change on the affected nodes.

However as shown in Figure 3, *versioned names,* while indicating change, fail to communicate the impact of change, to explain why `Susan 1.0` can work with `Tom 2.1`, but `Susan 2.0` cannot!

It is only when we look at the capabilities and requirements of the nodes participating in the graph that we understand the problem.

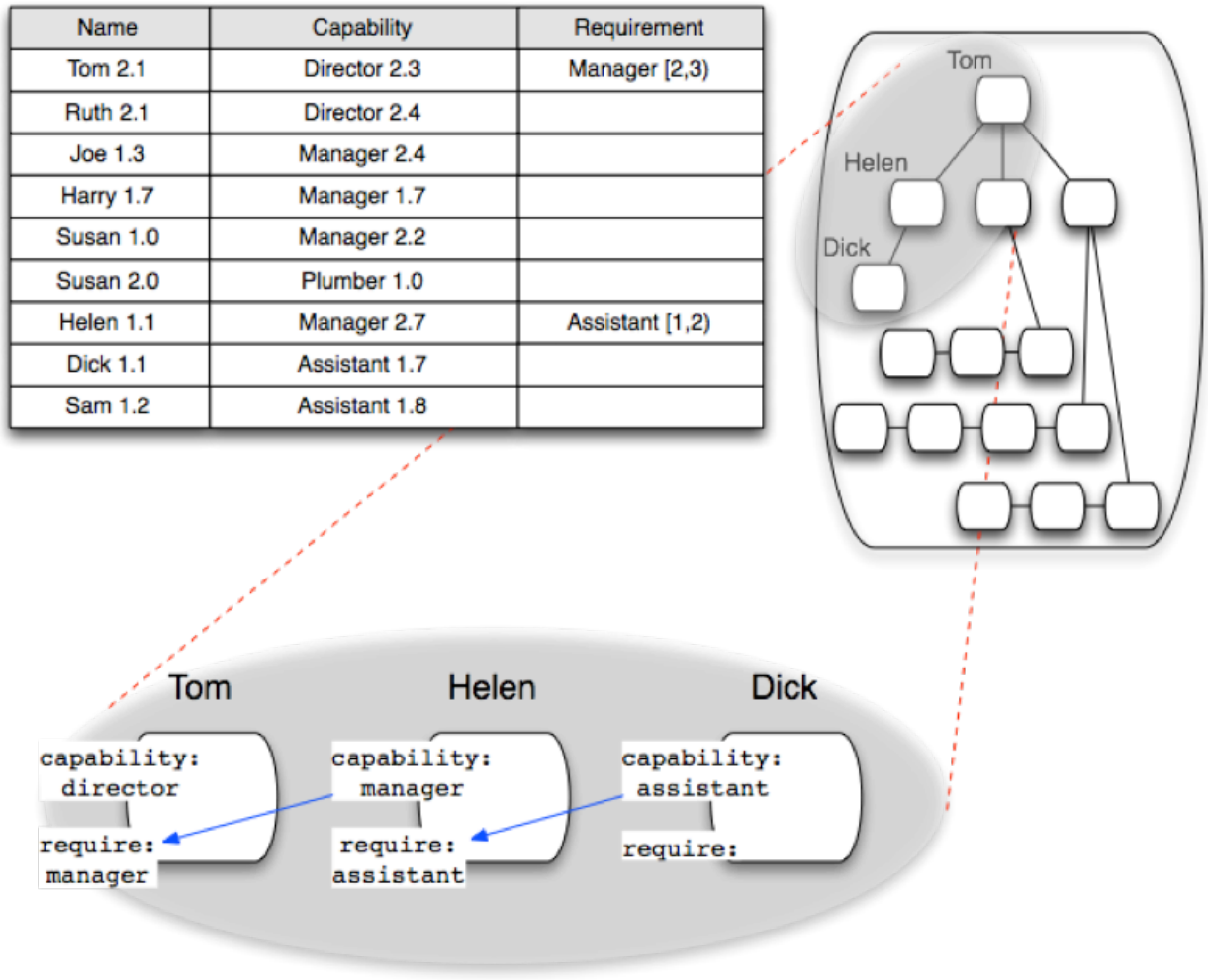

Figure 4: An organizational structure: effectiveness of versioned names, capabilities/ requirements and the use of semantic versioning.

Now we understand that `Tom 2.1` *requires* a `Manager` *capability.* This capability was initially provided by `Susan 1.0`. However, at the later point in time, `Susan 2.0`, having reflected upon her career, decided to re-train. `Susan 2.0` no longer advertises a `Manager` capability, but instead advertises a `Plumber 1.0` capability.

Once dependencies are expressed in terms of requirements and capabilities, then flexible substitution is possible. A node in the graph may be replaced by any other node whose capabilities satisfy the requirements of its neighbors.

Communicating Change - The Role of Semantic Versioning

Hence capabilities and requirements provide the mechanism via which we understand structural dependencies. However, we are still left with the problem of understanding the level of impact caused by a degree of change; i.e. the `Tom, Susan` scenario just discussed. Via simple versioning we can see that changes have occurred; however, we do not understand the consequences of these changes.

- If an employee is promoted and/or re-trained (capabilities enhanced), are the dependencies shown in the org chart still valid?
- If we re-factor a software component (changing/not changing, internal implementation and/or a public interface), to what degree are the dependencies still valid?

If, however, semantic versioning is used (see http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf), the impact of a change can be communicated; the importance of this is increasingly being realized by the industry. For example, the Apache Maven project has recently (2013) discussed adoption of version ranges for artifact names.[4] (While a welcome improvement, the concept is still flawed as the dependencies are still described in terms of the entities' names.)

Semantic versioning achieves this in a completely generic manner via the following mechanism:

- Advertised capabilities are versioned with a `major.minor.micro` versioning scheme.

[4] http://maven.apache.org/enforcer/enforcer-rules/versionRanges.html

- We then collectively agree that - `minor` or `micro` version changes represent non-breaking changes for third parties; 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.
- The interpretation of `micro, minor` and `major` are domain and context specific.
- *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`*.

If these semantic versioning rules are now used with the previous organization chart, we can immediately deduce that:

- If *Joe* is substituted for *Helen, Tom's* R*equirements* 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 specified by Tom i.e. `[2,3)`.

Semantic versioning used with requirements and capabilities provides sufficient information to enable substitution of components while ensuring that structural dependencies continue to be met: our simple system, whether a business unit or an application is agile, maintainable and evolvable!

Turtles - All the Way Down

What happens as the size and sophistication of our business unit or service provider increases? The number of constituent components and the inter-dependencies between these components also increases. Usually, if left unchecked, the number of inter-dependencies increases much more rapidly than the number of components. The structure becomes increasingly *complex*.

Those of you who have already noticed the degree of self-similarity[5] arising in the previous example may already have guessed the appropriate response to this situation.

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

To make the larger system manageable we introduce a new level of structural abstraction.

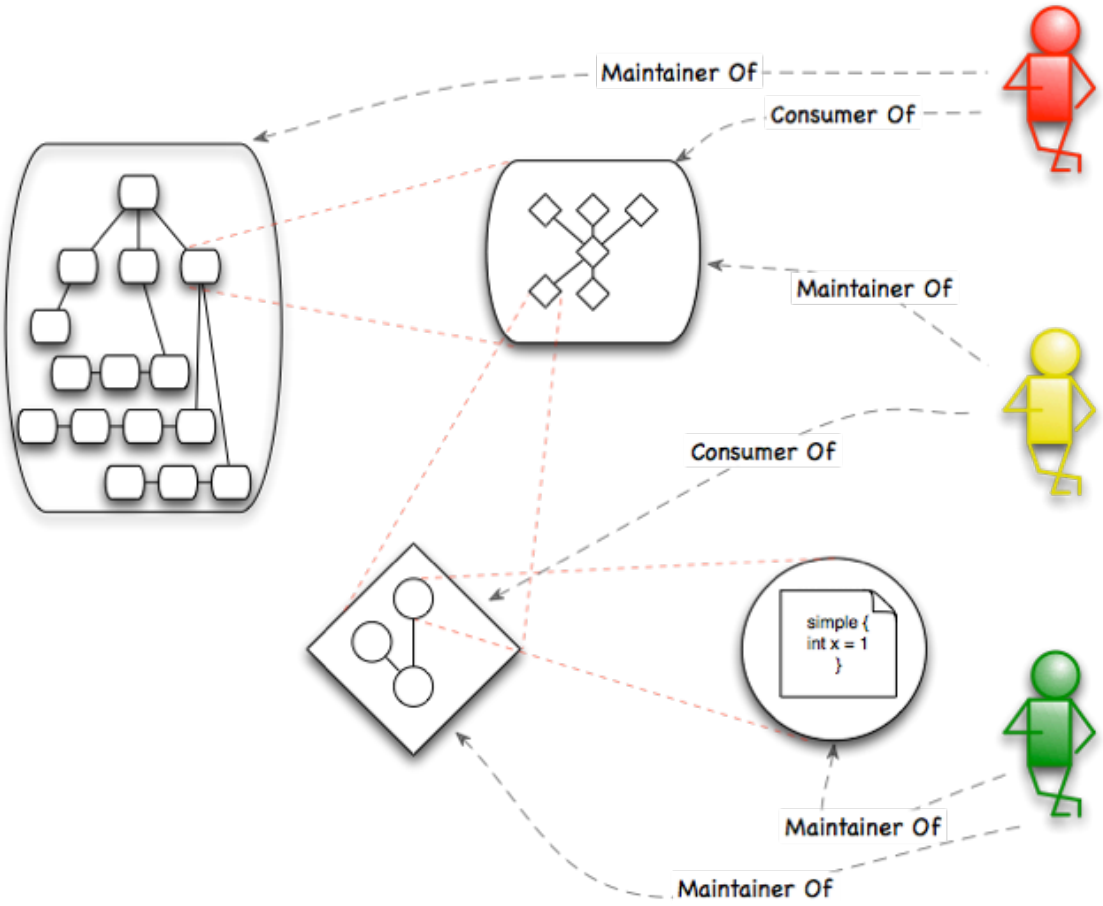

Figure 5: An agile hierarchy: Each layer only exposes necessary information. Each layer is composite with the dependencies between the participating components expressed in terms of their requirements and capabilities.

As shown in figure 5, it is possible for an individual to be a consumer at one level of a structural hierarchy while being a maintainer at the next logical layer above.

For enterprise software, this process started in the mid to late 1990s as organizations started to adopt coarse-grain modularity as embodied by Service Oriented Architectures (SOA) and Enterprise Service Buses (ESBs). These approaches allowed legacy 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, the core applications never actually changed; the existing application interfaces simply exposed as *SOA Services,* and/or message *Consumers*/*Publishers*. As the degree of modularity introduced was only one level deep, each post-SOA application was as internally inflexible as its pre-SOA predecessor.

Hence, with hindsight it should not be surprising that SOA - by itself - failed to deliver the promised cost savings and business agility.[6] Yet, while not an end in itself, traditional SOA is a valuable step on the journey towards modular systems.

To be Agile?

Agile systems need to exhibit the following characteristics:

- A *hierarchical* structure: Each layer in the hierarchy composed from components from the underlying layer.
- *Abstraction*: For each layer, the behavior of participating components is exposed via stated requirements and capabilities relevant to that layer.
- *Isolation*: Strong isolation ensures that the internal composition of each participating component is masked at each layer.
- *Self-Describing*: Within each layer the relationship between the participating components is self-describing; i.e. dependencies are described 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 remain localized to the affected components; the boundaries created by strong structural modularity shield 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 conclusion is as simple as it is profound. Systems achieve agility through structural modularity.

[6] http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html

OSGi and Modular Systems

Most developers appreciate that applications should be modular. Indeed logical modularity in the form of “Design Patterns” was rapidly embraced in the early years of object-oriented programming (see http://en.wikipedia.org/wiki/Design_Patterns). However, it has taken the industry much longer to appreciate the importance of *structural modularity* and its fundamental relationship to application maintainability and agility.

Today it is not unusual for an agile Java development team to break large Java artifacts (JARs) into a number of smaller ones. As these, in turn, grow in size, they are again broken down into yet smaller units. The development team understands that its agile objectives cannot be achieved with a large, monolithic code-base.

While the dependencies and the impact of change are understood by the development team that created the application, this structural information is not explicitly associated with the software artifacts. Should members from that development team leave, knowledge concerning application structure is lost. This exposes the organization to significant long-term governance problems, increased operational risk and spiraling maintenance costs. 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 since the components can only be understood by cracking open each Java JAR and reading code. All the components must be analyzed this way as the dependencies between them cannot be inferred in any other way!

The use of Maven helps the situation somewhat. Maven expresses dependencies between Java JARs in terms of component names (*Project Object Model - POM*). This enables a Maven based application to be simply assembled by a third party that didn’t create the original code. However, as we already know from the initial examples, 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. As no metadata exists that adequately describes the inter-relationship between the components, the resultant business system remains intrinsically *fragile*.

Aware of these issues, Kirk Knoernschild explores the relationship between agility and structural modularity in his book "Java Application Architecture"[7] and identifies the problem of the "missing middle." Knoernschild concludes that essential structural layers are missing in the architecture of traditional Java applications. At the coarse-grained end of the modularity spectrum we have traditional services, whereas at the fine-grained end of the modularity spectrum we have Java *packages* and *classes*. However, there is a void in the center.

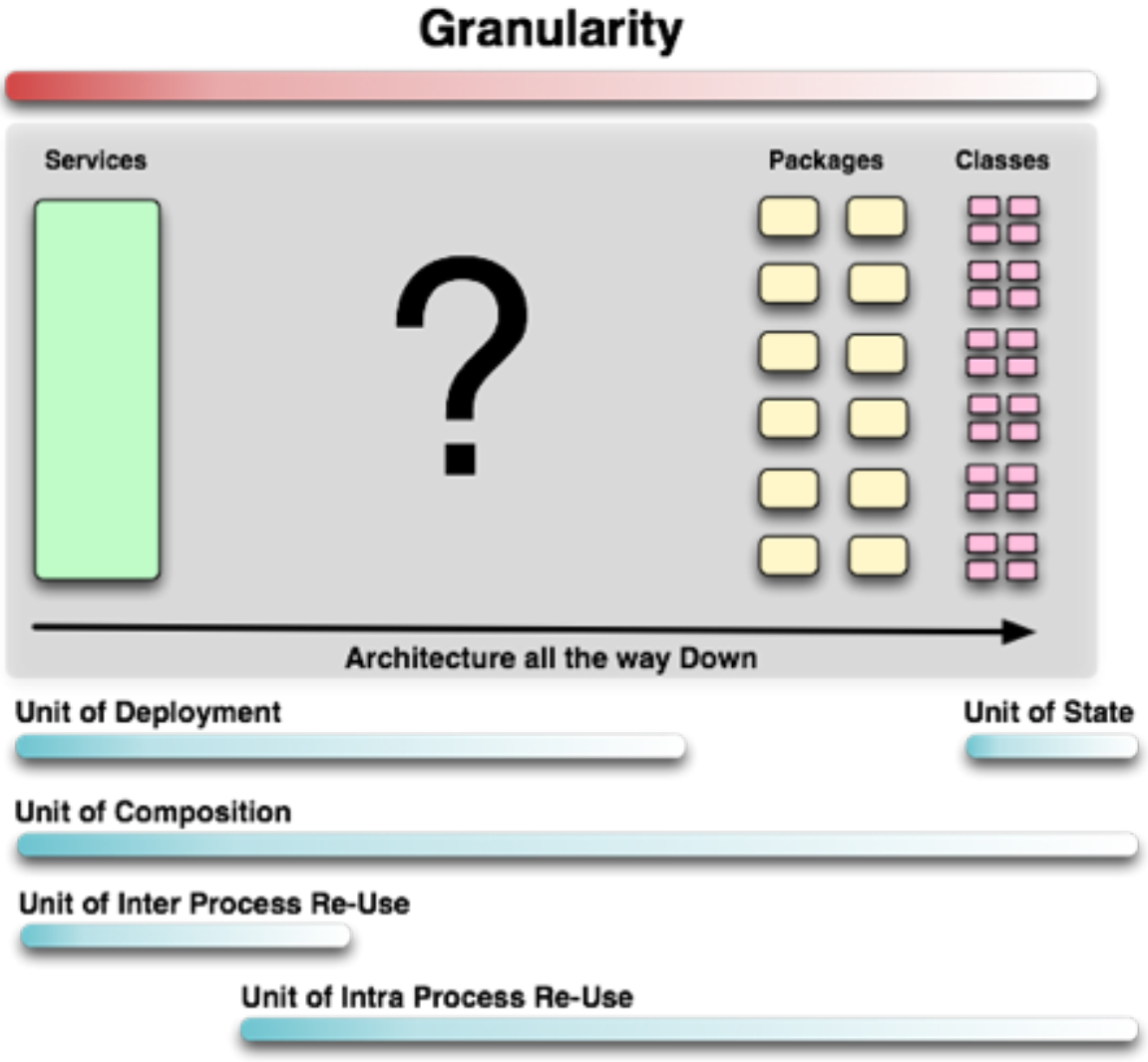

Figure 6: Structural hierarchy: The Missing Middle

[7] http://techdistrict.kirkk.com

Based on open industry standards created by the OSGi Alliance, OSGi directly address this issue by providing Java with the required modularity framework.

- OSGi bundles express dependencies in terms of Java package requirements and capabilities. Hence it is immediately apparent to a third party, whether a particular OSGi bundle can be substituted with a potential alternative.
- OSGi bundles use semantic versioning. Hence it is immediately apparent to a third party whether a change to an OSGi bundle is potentially a breaking change to those using it.

With enforced isolation, dependencies expressed via requirements, capabilities and semantic versioning, OSGi bundles' are the natural choice for enabling Java code re-use. OSGi bundles are also essential elements in an application's composition hierarchy (unit of composition) and provide the basis for a natural unit of deployment, update and patch (unit of deployment).

One structural layer up, OSGi also provides µServices. These are lightweight services that are able to dynamically find and bind to each other at runtime. OSGi services may be co-located within the same JVM, or via use of an implementation of OSGi's remote service specification, distributed across JVMs separated by an IP network. Coarse-grained business applications may then be composed from a number of finer grained, co-located or distributed OSGi µServices (*Unit of Intra/Inter Process Re-Use*), so completing OSGi's "Agile - All the Way Down!" story.

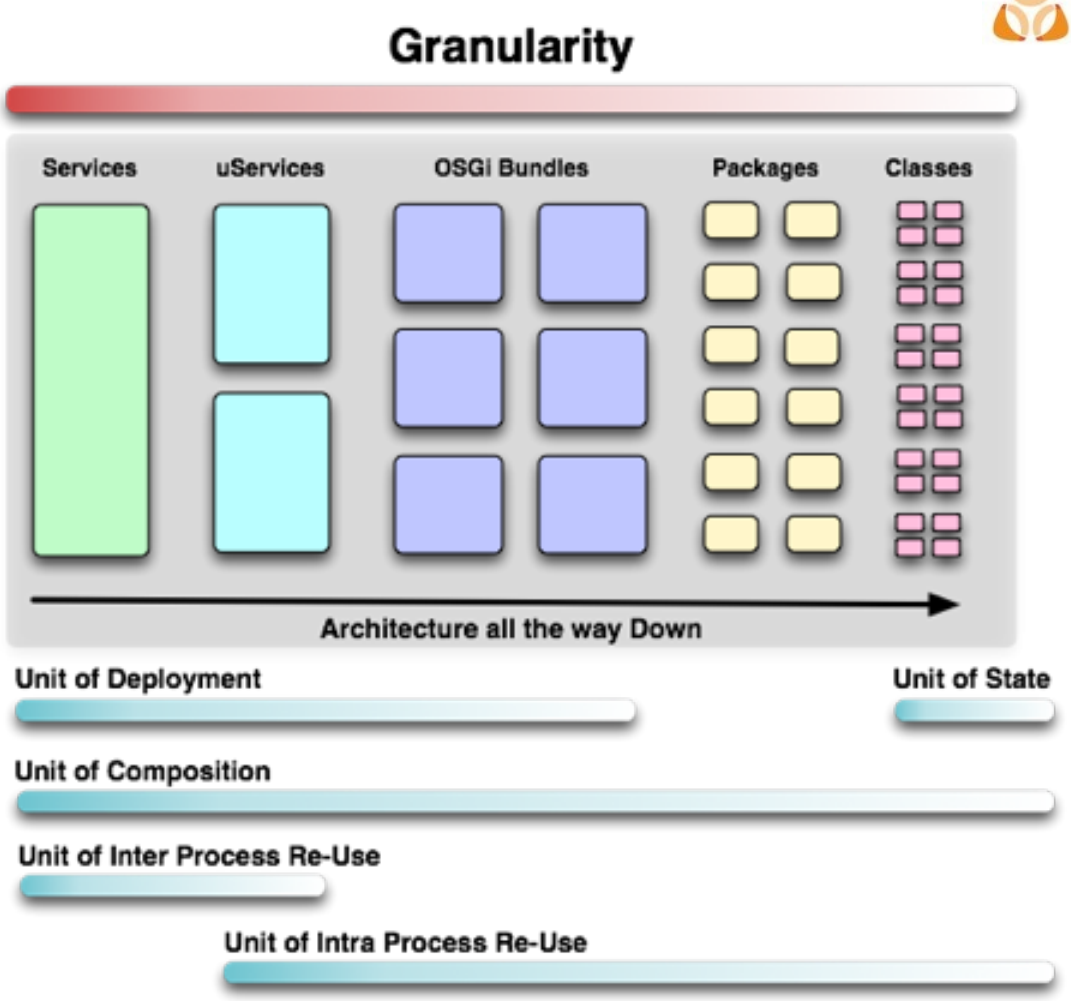

Figure 7: Structural hierarchy: OSGi services and bundles

Knoernschild concludes that:

"Not only does OSGi help us enforce structural modularity, it provides the necessary metadata to ensure that the Modular Structures we create are also Agile Structures."[8]

[8] k Knoernschild -Java Application Architecture

OSGi Enabling Agile Processes

As explained by Taleb in AntiFragile,"[9] *predictive* and *prescriptive* processes stand a high chance of failure as the large number of rigid interdependencies mean that Black Swan events, which are by their nature unpredictable, are much more likely.

The "Agile Movement"[10] is fundamentally an acknowledgement of, and response to, this issue; and focuses upon the organizational processes required to achieve agile development. A diverse spectrum of complementary lean and agile methodologies now exist; the two most popular are known as *Scrum* and *Kanban*.[11]

However, as is hopefully now apparent, the very same arguments apply to rigid monolithic software! Structural modularity is the foundation upon which application agility is realized, and the value of OSGi in enabling popular agile processes will be the subject of this section.

Scrum

"Customers frequently change their minds"

At its core, Scrum is a pragmatic methodology that acknowledges that *things* change in unforeseen and unforeseeable ways. Scrum acknowledges the existence of requirement churn, and adopts an empirical[12] approach to software delivery, accepting that the problem cannot be fully understood or even 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"[13] effort, meaning that 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. The tasks and estimated commitment for a sprint

---

[9] AntiFragile: How to live in a World we don't Understand - Nassim Taleb. ISBN-13: 978-1846141560

[10] http://www.agilemanifesto.org

[11] http://en.wikipedia.org/wiki/Lean_software_development

[12] http://en.wikipedia.org/wiki/Empirical

[13] http://en.wikipedia.org/wiki/Timeboxing

are identified in a planning meeting. A review, or retrospective, meeting follows the sprint to review progress and identify lessons for the next sprint.

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.

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.

It is hopefully self-evident that applying Scrum to a large monolithic code base is difficult. Conversely, as will be shown, Scrum concepts map well to a highly modular code base comprised of many self-describing, strongly isolated OSGi bundles.

Kanban

Kanban originates from the Japanese word "signboard" and traces back to Toyota, the Japanese automobile manufacturer, in the late 1940s.[14] Kanban encourages teams to have a shared understanding of work, workflow, process, and risk, enabling the team to build a shared comprehension of problems and suggest improvements, which can be agreed by consensus.

Kanban places an emphasis on “work in progress:”

1. Work-In-Progress (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 stage of the workflow 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.

Kanban encourages small, continuous, incremental and evolutionary changes.

Kanban concepts map naturally to a highly modular code base comprised of many self-describing, strongly isolated artifacts; specifically, the Kanban WIP process idea maps directly to the subset of OSGi bundles that are being actively worked upon. Hence, Kanban pull-based flow rates can be mapped to OSGi bundles’ change/release

[14] http://en.wikipedia.org/wiki/Kanban

rates. And as the degree of structural modularity increases, as the OSGi bundles become more fine-grain, the Kanban pull -based flow rates naturally increase, with each smaller OSGi bundle spending correspondingly less time in a WIP state.

The Agile Maturity Model

Fashioned after the Capability Maturity Model,[15] which allows organizations or projects to measure their improvements on a software development process, the Modularity Maturity Model[16] is an attempt to describe how far along the modularity path an organization or project might be. As is now argued, this, in turn, will directly dictate the level of agility that might be reasonably expected.

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

Level 1: **Ad Hoc** - No formal modularity exists. Dependencies are unknown. Applications have no, or limited, structure. Agile processes are likely to fail as application code bases are monolithic and highly coupled. Testing is challenging as changes propagate unchecked, causing unintentional side effects. Governance and change management are costly and acknowledged to be high-level operational risks.

Level 2: **Modules** - 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. Artifact repositories are used; however, their value is compromised as the artifacts are not self-describing. Agile processes are possible and do deliver some business benefit. However, the ability to realize Continuous Integration (CI) is limited by ill-defined dependencies. Governance and change management are not addressed. Testing is still failure-prone. Indeed, the testing process is now the dominant bottleneck in the agile release process. Governance and change management remain costly and continue to be high-level operational risks.

Level 3: **Modularity** - Module dependencies are now expressed via contracts (i.e. capabilities and requirements). Dependency resolution becomes the basis of software construction. Dependencies are semantically versioned, enabling the impact of change to be communicated. By enforcing strong isolation and defining dependencies in terms of capabilities and requirements, modularity enables many small development teams to

[15] http://en.wikipedia.org/wiki/Capability_Maturity_Model

[16] An initial version of a *Modularity Maturity Model*; this proposed by Dr Graham Charters at the OSGi Community Event 2011. The version in this paper has been adapted to emphasize Agility aspects.

efficiently work independently and in parallel. The efficiency of Scrum and Kanban management processes correspondingly increases. Sprints are now associated with one or more well-defined structural entities; i.e. the development or refactoring of OSGi bundles. Semantic versioning enables the impact of refactoring to be contained and efficiently communicated across team boundaries. Via strong modularity and isolation, parallel teams can safely sprint on different structural areas of the same application. Strong isolation and semantic versioning enable efficient/robust unit testing. Governance and change management are now demonstrably much lower operational risks.

Level 4: **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. Services lay the foundation for runtime loose coupling. The dynamic find and bind behaviors in the OSGi service model directly enable loose coupling by enabling the dynamic formation of composite applications. All local and distributed service dependencies are automatically managed. The change of perspective from code to OSGi µServices increases developer and business agility yet further: new business systems being rapidly composed from the appropriate set of pre-existing OSGi µServices.

Level 5: **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. From a business perspective, devolution enables significant 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 increases the effectiveness of offshoring/near shoring or the use of third-party, OSS or crowd-

sourced software components. This, in turn, directly leads to significant and sustained reductions in operational cost.

Level 6: **Dynamism** - This final level builds upon Modularity, Services and Devolution, and is the culmination of the organization's modularity/agility journey.

Business applications are rapidly assembled from modular components. As strong structural modularity is enforced (i.e. isolation enforced by the OSGi bundle boundary), components may be efficiently and effectively created and maintained by a number of alternate providers (onshore, nearshore, offshore, OSS, third party).

- 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 and change control processes.
- Software fixes may be rapidly deployed into production.
- The capabilities of existing applications may be rapidly extended.
- 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 data center environments.

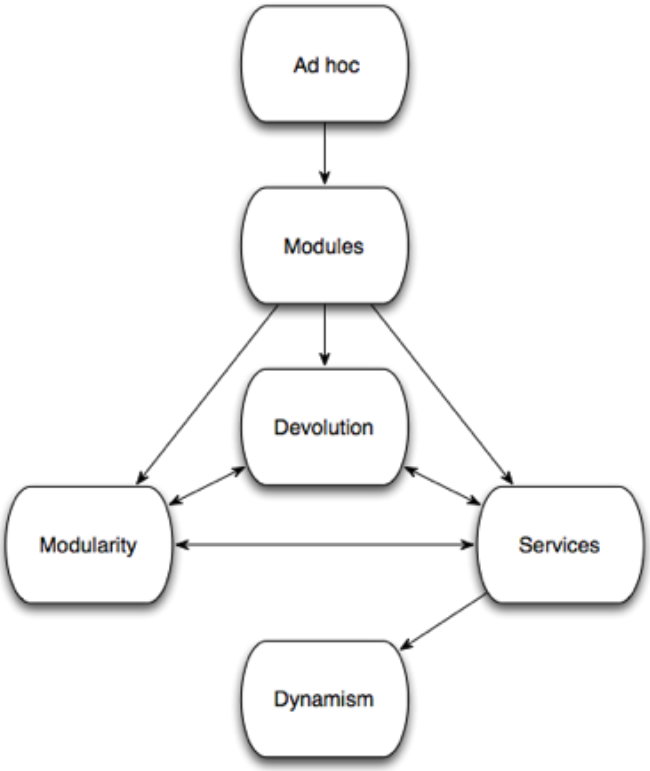

Figure 8: Modularity Maturity Model

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 `Dynamism` endpoint as soon as possible.

As shown, 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` positions the organization for subsequent migration to the `Modularity` and `Services` levels.

It should be noted that the ability to pursue multiple alternative options is in itself a key indicator of an increasingly agile environment!

As the level of structural modularity increases through adoption of OSGi, both Scrum and Kanban processes become correspondingly more efficient and effective.

Agility & CI - An OSGi Use Case

The following real-world use case demonstrates a successful realization of these principles.

Siemens Corporate Technology Research group is comprised of a number of engineers with diverse skills spanning computer science, mathematics, physics, mechanical and electrical engineering. The group provides solutions to Siemens business units based on neural network 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. The resultant business solutions are composed from the repository of re-usable components.

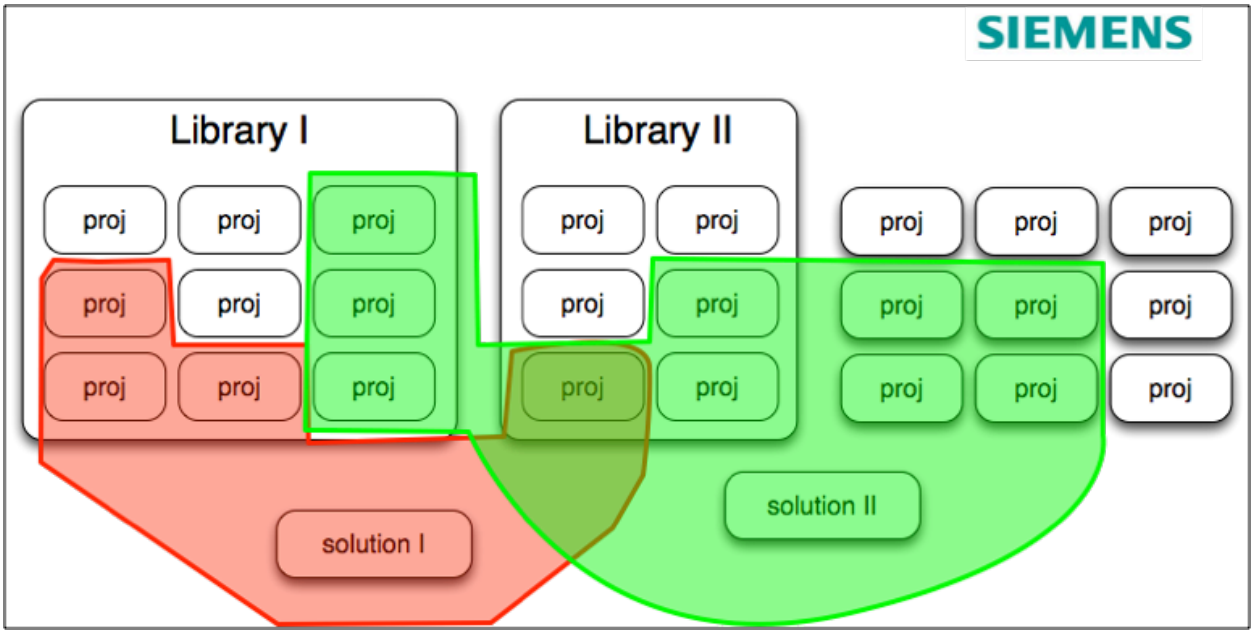

Figure 9: Siemens' Product Repository

In a presentation entitled "Workflow for Development, Release and Versioning with OSGi/Bndtools: Real World Challenges,"[17] the Siemens team presented both the business drivers and the resultant solution that met these objects via a highly agile OSGi based continuous integration environment.

[17] The 2012 OSGi community event (http://www.osgi.org/CommunityEvent2012/Schedule)

The Siemens' team stated the following business objectives:

1. *Build Repeatability*: It must be possible 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*: Siemens required the ability to 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 this assembly would work.
3. *Full Traceability I:* It must be possible to ensure that the released software artifacts are always exactly the same artifacts that were tested by QA. Specifically, the solution must avoid the necessity to rebuild in order to advance from the testing state into the released state.
4. *Full Traceability II*: It must be possible to trace the artifact's heritage back to its original sources and dependencies.

To achieve rapid prototyping, 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 based upon the maturity of OSGi technology, the open industry specifications that underpin OSGi implementations, and the technology governance provided by the OSGI Alliance. Also, OSGi semantic versioning, fully leveraged within semantic version aware development and build processes, would be critical for achieving the stated business objectives.

The solution 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 Bndtools project was selected (see http://bndtools.org).

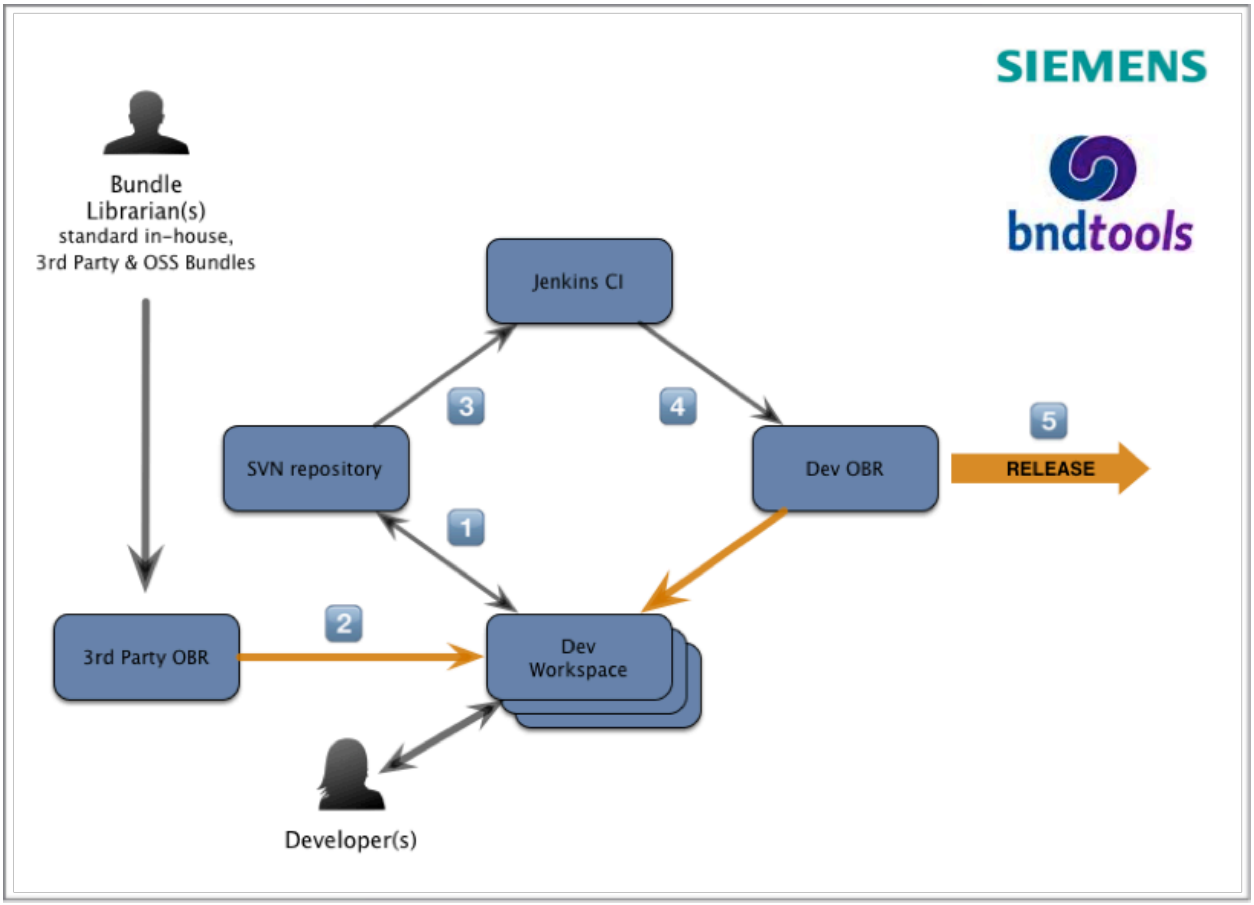

Figure 10: Siemens' OSGi/Bndtools based continuous integration environment

The resultant Bndtools based continuous integration solution is shown in Figure 10.

1. Via the Eclipse/Bndtools IDE Siemens' developers maintain their own local workspace and check results into their local SVN source repository; this SVN repository contains only work in progress (WIP).
2. Siemens' developers are able to include one or more read-only OSGi Bundle Repositories (a.k.a. OBR) in their development environment. The primary repository is the team's Read-Only Development Repository, but repositories used by other development teams, a repository containing standardized Siemens' components and approved third-party and/or OSS repositories, might be included.
3. At the point the WIP item is completed and ready for release to the local development repository, Bndtools automatically calculates the correct semantic version update based on the delta of changes in the current code base (N) with

respect to the previous released version (N-1). The Jenkins build server builds and releases the correct semantic-versioned artifact to the development repository.

4. Once released to the Read-Only Development Repository, the OSGi bundle is available for use by all developers with access to that repository. While the bundle remains unreleased, subsequent code updates can be applied, the bundle re-built with the semantic version again calculated against the same N-1 baseline.
5. At the point the artifact is released from the development repository to QA, it now defines the new baseline. All subsequent updates to the N+1 version of the OSGi bundle are now baselined by Bndtools against the newly released version.

Bndtools' sophisticated semantic versioning behaviors mean that appropriate semantic version changes are automatically calculated, so conveying the nature of the change and offloading this concern from the developer.

This strategy proved successful in delivering a highly agile development and continuous integration environment for Siemens' developers that also met the business' re-use and governance objectives.

Conclusion

Whether a senior manager responsible for building an agile business, or software developer tasked with building an agile software solution, the fundamental challenge is the same.

Agility can only be fully realized when the underlying entity - the organization or the software product - has a high degree of structural modularity. If the entity is monolithic and so change resistant, then no amount of agile process will address this. For example, there would have been little point in implementing Kanban methodologies in the pre-1900s Automobile Industry - before the innovation of the production line – which, in turn, was dependent upon a more fundamental innovation: the idea of *modular assembly.*

The relationship between structural modularity and agility is fundamental. It is also no accident that OSGi, *the modularity system for Java*, is designed the way it is. OSGi's design is a logical consequence of the challenges of building highly *modular* and therefore achieving highly *agile* Java software systems. With OSGi™, Java developers have a powerful ally; an ally that provides the foundation, based on open industry specifications, upon which their organization's IT agility goals can be fully realized.

In addition to creating open industry standards for the OSGi ecosystem, the OSGi Alliance provides a number of services to help the community accelerate its adoption of OSGi technology. Recent OSGi Alliance initiatives include the OSGi enRoute Project, which demonstrates Java modularity best practices, and a soon-be-introduced OSGi developer certification program to foster the growth of knowledge and skills to build modular agile Java based systems.

The Author

Richard Nicholson, CEO and Founder of Paremus (www.paremus.com) has been a member of the OSGi Alliance Board since 2010, and served as President between 2011 and 2013. Paremus offered the industry's first distributed OSGi Cloud runtime, the Service Fabric, in 2005, and have been actively promoting the fundamental importance of structural modularity in advanced distributed/Cloud based systems since then.

Paremus' involvement with the OSGi Alliance started in 2006, with the submission of RFP 75: RMI and Serialisation for OSGi, the foundations of what is now Remote Services/Distributed OSGi. Further Paremus contributions have included RFP 103 Class Loading Improvements, RFP 133 OSGi & Cloud Computing, RFP 158 Distributed Eventing, RFC 183 Cloud Ecosystems and OSGi, published specifications for the new R5 Resolver and most recently new specifications for Java Promises and Asynchronous Services.

In addition, the Paremus team continues its involvement in OSGi industry activities including leading the Bndtools initiative (http://bndtools.org) and contributing to the Apache Aries and Felix projects.

Paremus provides a range of OSGi training and consulting services to help organizations accelerate their adoption of OSGi and to assist with individual's preparation for the OSGi developer certification program, currently under development by the Alliance.

Richard's blog can be found at: http://blogs.paremus.com.

The OSGi Alliance

The OSGi Alliance is a worldwide consortium of technology innovators that advances a proven and mature process to enable the componentization of applications into well-defined software modules, and ensure interoperability of applications and services over a broad variety of devices.

The Alliance provides specifications, reference implementations, test suites and certification to foster a valuable cross-industry ecosystem. OSGi technology is shipping in millions of units worldwide, and is deployed by Fortune Global 500 companies in enterprise, desktop, embedded home and telematics markets. Member companies collaborate within an egalitarian, equitable and transparent environment and promote adoption of OSGi technology through business benefits, user experiences and forums.

For more information on the non-profit technology corporation, visit http://www.osgi.org or contact help@osgi.org.