JAVA APPLICATION ARCHITECTURE MODULARITY PATTERNS WITH EXAMPLES USING OSGI pdf

1: java - OSGi: should I have a Dao bundle? - Stack Overflow

*Java Application Architecture: Modularity Patterns with Examples Using OSGi (Robert C. Martin Series) [Kirk Knoernschild] on www.enganchecubano.com *FREE* shipping on qualifying offers. "I'm dancing! By god I'm dancing on the walls.*

Reading List Java Application Architecture: The first part of the book argues for the case for modularity, by talking about the runtime and development support for modules. It then builds on this showing how modularity can be used at design-time to facilitate architecture, and how this can integrate with service oriented architecture. At the end, there is a reference implementation that shows how the pieces fit together. Base Patterns Manage Relationships â€" Design module relationships Module Reuse â€" Emphasize reusability at the module level Cohesive Modules â€" Module behavior should serve a singular purpose Dependency Patterns Acyclic Relationships â€" Module relationships must be acyclic. Levelize Modules â€" Module relationships should be levelized. Physical Layers â€" Module relationships should not violate the conceptual layers. Container Independence â€" Modules should be independent of the runtime container. Independent Deployment â€" Modules should be independently deployable units. External Configuration â€" Modules should be externally configurable. Default Implementation â€" Provide modules with a default implementation. Extensibility Patterns Abstract Modules â€" Depend upon the abstract elements of a module. Separate Abstractions â€" Place abstractions and the classes that implement them in separate modules. Utility Patterns Collocate Exceptions â€" Exceptions should be close to the classes that throw them. Levelized Build â€" Execute the build in accordance with module levelization. Although this section is OSGi specific, the patterns in the rest of the book apply to any module system in Java, including the up-coming Jigsaw project, and as a guide for those wanting to build modular software without a runtime module system. InfoQ caught up with the Kirk, and asked for his thoughts on software and modularity, and started off by asking how software complexity has changed over the last few decades: More powerful hardware along with more expressive programming languages makes it easier to create more powerful software systems. The study also claims that the number of lines of code doubles every seven years. People who use software naturally demand more from it, and so we churn out new versions and upgrades to support our users. But more code means more maintenance. And maintaining large systems is inherently more difficult than smaller systems. What effect has that had with how software is designed and architected? In many ways, we still design software the same way today that we did 15 or 20 years ago. SOA and services are a step in the right direction, but that only addresses part of the challenge. One of the most perplexing aspects of service design is determining service granularity. Too coarse-grained and the service does too much to be useful across a variety of different contexts i. As I discuss in the book, maximizing reuse complicates use. A single level of abstraction cannot solve this problem. Additional levels are necessary, and modularity is that next level. Modularity is the next step that will help you architect "all the way down. Modules, which are at a different level of granularity than services, help you architect all the way down. What is the future of modularity on the JVM? There is a lot going on with modularity on the JVM. OSGi is being baked into just about every major vendor product and is available for use within many organizations today. For those of you familiar with Frederick Brooks, the problem centers around the essential complexity of software development. Architecture and design is very difficult. Oracle is also baking Jigsaw into Java SE, which will be available sometime in Modularity is coming to the Java platform, and it is going to change the way we design software systems. What made you decide to write Java Application Architecture? About 10 years ago, I recognized a significant problem with software I was designing. We put all this effort into designing really good class structures and then we packaged everything into this huge single deployable unit, like an EAR or WAR file. About that same time, I inked my second book deal and planned to write about software patterns. The perplexing issues I faced with my designs carried over into my writing; something was missing that made it impossible to realize the advantages I sought. I also

began studying the works of Clemens Szyperski who is the author of " Component Software: Over the next couple of years, I honed this approach. By using the JAR file as the unit of modularity, I was suddenly able to realize many of the advantages I sought. The patterns you see in the book are the result of what began about 10 years ago. So I took some time away from the book, continued to prove the patterns, and about two or three years ago, reached out to a very patient publisher to let them know I was ready. Fortunately, they gave me another chance and the result is the modularity patterns. Maybe that explains more about how the book came to be rather than why I wrote it. Ultimately, I wrote it because the patterns made a big difference in improving the systems I developed and I wanted to share them with the world. Is the book OSGi specific? The patterns, and the entire book actually, are designed to be used with everything available in standard Java today. My discovery of OSGi was actually somewhat of an epiphany. I found a community of like-minded folks and a framework that gave runtime support to many of the patterns. The book provides some examples that demonstrate OSGi, but primarily to demonstrate the strengths of OSGi and give people a glimpse into what it takes to move a system from standard Java to OSGi. In Chapter 2, I talk about the two facets of modularity â€" the runtime model and the development model. The development model can be subdivided into the programming model and the design paradigm. A module framework is going to give you a programming model and a runtime model. But no framework is going to help you design a good set of modules. I like to draw an analogy with the object-oriented paradigm. Design patterns help you with that. The same is true with modularity. This way, you can start applying the concepts in the book right now without adopting any new frameworks or making any additional infrastructure investment. Most of the code samples use standard Java to demonstrate the concepts. I added several additional code samples built using OSGi to illustrate the additional benefits you get from runtime support for modularity. Is modularity only about reuse? Reuse is it interesting because reuse sells. By that I mean just about major technology trend of the past couple of decades touts reuse as its primary advantage. Shortly thereafter it was component-based development. Then we moved onto service oriented architecture. Ironically, each of them has failed to live up to expectations, and many development teams continue to struggle with reuse. But modularity has several other advantages, too. These include better maintainability, dependency management, and simply increasing your ability to understand very complex software systems. Several chapters in Part 1 of the book discuss these ideas in detail. Can modularity be applied to existing systems, or does it need to be designed in up-front? Can systems be refactored to become more modular? The result is a highly modular system. For instance, just about every team tries to layer their software systems. But these are logical layers. That is, they exist conceptually in the class structure but not physically among deployable units. When you attempt to layer your software into modules, you start focusing on deployable units. I talk more about this in the Physical Layers patterns. What do you see as the future of modularity for Java? Modularity is going to change how we develop and deliver software systems on the Java platform. From a development perspective, it fills a gap left by other major design paradigms. For delivering software, it marks the end of the monolithic and static platform. Perhaps, given the level of excitement surrounding application storefronts in the mobile ecosystem, this repository will resemble something like an "enterprise module store. At some point, perhaps the environment will assemble itself based on the infrastructure requirements of your application. A sample of the book is available, and the code for the pattern samples is available on GitHub. The book is available for purchase from Amazon in printed form , as well as in Kindle and iBooks electronic format. About the Book Author Kirk Knoernschild is a software developer with a passion for building great software. He takes a keen interest in design, architecture, application development platforms, agile development, and the IT industry in general, especially as it relates to software development. His recent book, Java Application Architecture: Modularity Patterns with Examples Using OSGi, was published in , and introduces 18 patterns that help you design modular software. You can visit his website here. Despite having previously been an editor for EclipseZone and a nominee for Eclipse Ambassador in , his day-to-day role involves neither Eclipse nor Java.

JAVA APPLICATION ARCHITECTURE MODULARITY PATTERNS WITH EXAMPLES USING OSGI pdf

2: The Future of Modularity: OSGi : Java Application Architecture

Java Application Architecture MODULARITY PATTERNS WITH EXAMPLES USING OSGI Kirk Knoernschild Upper Saddle River, NJ â€¢ Boston â€¢ Indianapolis â€¢ San Francisco New York â€¢ Toronto â€¢ Montreal â€¢ London â€¢ Munich â€¢ Paris â€¢ Madrid.

It fills a gap that has existed since we began develop enterprise software systems in Java. But within each lies a common theme, and some key phrases. Here are a few of the definitions. An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural elements and behavioral elements into progressively larger subsystems, and the architecture style that guides this organization â€" these elements and their interfaces, their collaborations, and their composition. The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. A formal description of a system, or a detailed plan of the system at component level to guide its implementation 2. The structure of components, their inter-relationships, and the principles and guidelines governing their design and evolution over time. There exist important underlying currents embodied by these keywords. But these keywords also lead to some important questions that must be answered to more fully understand architecture. What makes a decision architecturally significant? What are the elements of composition? How do we accommodate evolution of architecture? And what does this have to do with modularity? He described how the earth orbits around the sun and how the sun, in turn, orbits around the center of a vast collection of stars called our galaxy. At the end of the lecture, a little old lady at the back of the room got up and said: The world is really a flat plate supported on the back of a giant tortoise. In dysfunctional organizations, architects and developers fail to communicate effectively. The result is a lack of transparency and a lack of understanding by both sides. As shown in Figure 1, architects tend to bestow their wisdom upon developers who fail to grasp the high level concepts in terms that make sense to them. The failure often occurs though I recognize there are other causes because architecture is about breadth and development is about depth. Each group has disparate views of software architecture, and while both are warranted, a gap between these views exists. The architect might focus on applications and services while the developer focuses on the code. Sadly, there is a lot in between that nobody is focused on. It is this gap between breadth and depth that contributes to ivory tower architecture. The Ivory Tower 4. So assuming good intent on the part of the architect and the developer, how can we bridge the gap between breadth and depth? How can we communicate more effectively? How do we increase understanding and transparency? In most successful software projects, the expert developers working on that project have a shared understanding of the system design. These components are usually composed of smaller components, but the architecture only includes the components and interfaces that are understood by all the developersâ€¦Architecture is about the important stuff. We must have a shared understanding of how the system is divided into components, and how they interact. And it is through this social aspect of architecture that we can break down the divide between architects and developers. There is a huge middle ground that each must also focus on, as illustrated by the diagram in Figure 2. Architecture all the way Down Focusing exclusively on top level abstractions is not enough. Emphasizing only code quality is not enough either. We must bridge the gap through other means, including module and package design. Often times, when I speak, I ask the audience to raise their hands if they spend time on service design. I also ask them to raise their hand if they spend time on class design and code quality. But then I ask if they also spend time on package and module design. Usually, only a small percentage leave their hands raised. This is unfortunate, because module and package design are equally as important as service and class design. Within each application or service awaits a rotting design, and atop even the most flexible code sits a suite of applications or services riddled with duplication and lack of

understanding. A resilient package structure and corresponding software modules help bridge the divide between services and code. Modularity is an important intermediate technology that helps us architect all the way down, and is the conduit that fills the gap between breadth and depth. We need to start focusing on modularity to ensure a consistent architecture story is told. It is the glue that binds. That is, a system with a resilient, adaptable, and maintainable architecture. Examining the earlier definitions of architecture leads us to the goal of architecture. All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change. We attempt to make something architecturally insignificant by creating flexible solutions that can be changed easily, as illustrated in Figure 3. But herein lies a paradox. Figure 3 â€" The Goal of Architecture 4. And the way to eliminate architecture by minimizing the impact of cost and change is through flexibility. The more flexible the system, the more likely that the system can adapt and evolve as necessary. And complexity is the beast we are trying to tame because complex things are more difficult to deal with than simple things. But what if we were able to tame complexity while increasing flexibility, as illustrated in Figure 4? Taming the best we call complexity is the emphasis of Chapter 5. Is it even possible? In other words, how do we eliminate architecture? Figure 4 â€" Maximize Flexibility, Manage Complexity 4. In other words, we should try to defer commitment to specific architectural decisions that would lock us to a specific solution until we have the requisite knowledge that will allow us to make the most informed decision. Identifying the seams in a system involves identifying clear lines of demarcation in your architecture. Where Booch talks about components, today we talk about modules. Modularity combined with design patterns and SOLID principles, represent our best hope to minimize the impact and cost of change, thereby eliminating the architectural significance of change. The joints of a system are discussed in Chapter 5, Section 5. We devoted Chapter 2 to defining module. Developing a system with an adaptive, flexible, and maintainable architecture requires modularity because we must be able to design a flexible system that allows us to make temporal decisions based on shifts that occur throughout development. Modularity has been a missing piece that allows us to more easily accommodate these shifts, as well as focus on specific areas of the system that demand the most flexibility, as illustrated in Figure 5. Figure 5 â€" Encapsulating Design Modularity, in conjunction with design patterns and SOLID principles, represent our best hope to minimize the impact and cost of change. Through explanation, we answered each of these questions. A decision is architecturally significant if the impact and cost of change is significant. The elements of composition include, classes, modules, and services. Evolution is realized by designing flexible solutions that can adapt to change. But flexibility breeds complexity, and we must be careful to build flexibility in the right areas of the system. Modularity is an important intermediate component that helps increase architectural agility. It fills a gap that exists between architects and developers. It allows us to create a software architecture that can accommodate shifts. Modularity helps us architect all the way down.

JAVA APPLICATION ARCHITECTURE MODULARITY PATTERNS WITH EXAMPLES USING OSGI pdf

3: Chapter 4 â€“ Architecture and Modularity : Java Application Architecture

Java Application Architecture: Modularity Patterns with Examples Using OSGi By Kirk Knoernschild Published Mar 15, by Prentice Hall.

Today, I find the exact opposite. Patterns have become commonplace, and most developers use patterns on a daily basis without giving it much thought. Patterns are no longer fashionable. But, the role design patterns have played over the past decade should not be diminished. They were a catalyst that propelled object-oriented development into the mainstream. They helped legions of developers understand the real value of inheritance and how to use it effectively. Patterns provided insight into how to construct flexible and resilient software systems. Patterns are still widely used today, but for many developers, they are instinctive. No longer do developers debate the merits of using the Strategy pattern. Nor must they constantly reference the GOF book to identify which pattern might best fit their current need. Instead, good developers now instinctively design object-oriented software systems. Many patterns are also timeless. That is, they are not tied to a specific platform, programming language, nor era of programming. With some slight modification and attention to detail, a pattern is molded to a form appropriate given the context. As we learn more about patterns, we offer samples that show how to use patterns in a specific language. We call these idioms. They are not tied to a specific platform or language. But most important, I hope that with your help, these patterns will evolve and morph into a pattern language that will help us design better softwareâ€”software that realizes the advantages of modularity. Many of these design principles are embodied within design patterns. Further analysis of the GOF patterns reveals that many of them adhere to these principles. For all the knowledge shared, and advancements made, that help guide object-oriented development, creating very large software systems is still inherently difficult. These large systems are still difficult to maintain, extend, and manage. The current principles and patterns of object-oriented development fail in helping manage the complexity of large software systems because they address a different problem. They help address problems related to logical design but do not help address the challenges of physical design.

JAVA APPLICATION ARCHITECTURE MODULARITY PATTERNS WITH EXAMPLES USING OSGI pdf

4: Knoernschild, Java Application Architecture: Modularity Patterns with Examples Using OSGi | Pearson

Welcome to the homepage of Java Application Architecture: Modularity Patterns with Examples Using www.enganchecubano.com book is now available on Amazon in print and Kindle versions, iBooks, and other sites.

Kirk Knoernschild is one of the leading experts when it comes to the OSGi community. DZone recently had a chance ot interview him about his thoughts on modularity in and his new book is "Java Application Architecture: Who is your target audience for the book? Is it more than just developers using OSGi? The book is for any software developer or architect who wants to build more architecturally resilient software. The first example of this appears at the end of Chapter 3. Then, in Part 3 toward the end of the book, I take several of the pattern examples and "OSGi-ify" them. Because the samples used the patterns, they are already highly modular and introducing OSGi into the mix is very easy. And platform support for modularity is coming soon, be it OSGi or Jigsaw, so this book helps your prepare for that support and design better software in the process. What are the benefits of modularity and why is it so important? Well, there are several subtle benefits, and this is really the focus of the first seven chapters of the book where I talk about why modularity is so important. Modular software is much easier to extend, reuse, maintain, and adapt. These are real benefits that you experience almost immediately. The days of designing these monolithic applications must come to and end. In fact, it will come to and end and we see it happening right now. Instead, we have to start delivering Web APIs that allow you to support many different clients. As I like to say, within each service awaits a rotting design and without modularity, your services become these huge monoliths too. So you have to architect all the way down, and modularity helps you do this. Modularity is a cornerstone of the future architectural model and we have to start designing modular software. Major platform vendors are recognizing this. The OpenJDK recognizes this. And we, the enterprise developers, have to recognize this too. The book shows you how to do this today. Then, once your platform actually treats modularity as a first class construct, several other amazing things begin to happen. The environment itself responds to these capabilities and does what it needs to enable them. Again, I talk about these ideas in the first seven chapters and then show you how to do it in the remainder of the book. There are several examples of this taking place already today, though in a more limited fashion. The most obvious is the Eclipse platform, of which the plugin model is based entirely on modularity. In this latter example, the local environment responds to the requirements of the application and provides just the right set of capabilities. Just about every platform vendor is incorporating ideas like this into their platforms, and most are using OSGi to do it. Other vendors, like Paremus , are leading the change by introducing new platforms. And as we look to move applications to the cloud, this type of dynamic platform is a perfect fit. Kind of makes the this concept of Profiles that were introduced as part of Java EE6 look pretty antiquated because they are still so static. A module framework like OSGi is one way to design modular software for Java. Are there any other ways for Java? When it comes to other languages, the ideas in the book translate quite well. And the patterns are largely language agnostic. For instance, in Part 3, I show how easy it is to use a language like Groovy and Scala. There is certainly disagreement over some key items, but the reality is that both OSGi and Jigsaw are going to be major components on the Java platform and developers are going to have to choose one based on their needs. So if you start designing software this way, you are effectively future proofing the architecture so that you can adopt the module framework you want when you want to. How widespread is OSGi in the Java community and what are your predictions for how adoption will change based on the introduction of Jigsaw? Some people might complain that OSGi is too complex. That is, if the platform vendors are using OSGi as the cornerstone of their platforms, then maybe we should think about using it for the applications we build. OSGi is mature and proven, but Jigsaw is going to be the standard. The next couple of years will be an interesting ride for the Java platform, and a lot of things are going to change. My intent with the book is to help you incorporate modular design thinking into your development initiatives today. Quickly and easily gain access to the tools and information you need! Explore, test and

combine our data quality APIs at Melissa Developer Portal â€" home to tools that save time and boost revenue. Read More From DZone.

JAVA APPLICATION ARCHITECTURE MODULARITY PATTERNS WITH EXAMPLES USING OSGI pdf

5: Java Application Architecture: Modularity Patterns with Examples Using OSGi | InformIT

Start by marking "Java Application Architecture: Modularity Patterns with Examples Using OSGi (Software Patterns Series)" as Want to Read: Want to Read saving Want to Read.

Modularity involves breaking a large system into separate physical entities that ultimately makes the system easier to understand. For instance, software modules with few incoming dependencies are easier to change than software modules with many incoming dependencies. Likewise, software modules with few outgoing dependencies are much easier to reuse than software modules with many outgoing dependencies. Reuse and maintainability are important factors to consider when designing software modules, and dependencies play an important factor. Module cohesion also plays an important role in designing high-quality software modules. Contrarily, a module that does too much is difficult to reuse because it provides more behavior than other modules desire. When designing modules, identifying the right level of granularity is important. Modules that are too fine-grained provide minimal value and may also require other modules to be useful. Modules that are too coarse-grained are difficult to reuse. The principles in this book provide guidance on designing modular software. Many of these principles would not be possible without the principles and patterns of object-oriented design. On the Java platform, the unit of modularity is the JAR file. Although these principles can be applied to any other unit, such as packages, they shine when using them to design JAR files. In OSGi parlance, a module is known as a bundle. OSGi provides a framework for managing bundles that are packaged as regular Java JAR files with an accompanying manifest. The manifest contains important metadata that describes the bundles and its dependencies to the OSGi framework. However, OSGi is not a prerequisite for using the modularity patterns. OSGi simply provides a runtime environment that enables and enforces modularity on the Java platform. OSGi offers the following capabilities: Enables and enforces a modular approach to architecture on the Java platform. Permits modules to be deployed and updated within a running system without restarting the application or the JVM. Allows modules to hide their implementation details from consuming modules. Encourages service-oriented design principles in a more granular level within the JVM. Requires explicit declaration of dependencies between modules.

JAVA APPLICATION ARCHITECTURE MODULARITY PATTERNS WITH EXAMPLES USING OSGI pdf

6: Java Application Architecture

Java Application Architecture: Modularity Patterns with Examples using OSGi is Kirk Knoernschild's seminal book on a pattern catalogue for modular systems design. Starting with an overview of the.

There are a couple of reason I chose not to. Among other things, designing modular software requires that we understand the weight and granularity of individual modules discussed in Chapter 6 , and use the right techniques to manage the coupling between modules dependencies are discussed in Chapter 5. In other words, designing good software is our job. Tools and technologies may help, but make no guarantee. For more on the differences between the runtime and development model, see Chapter 3. I want to use the same tools and techniques that we can leverage in the enterprise right now. Instead, I want to focus on pure modularity. Instead, OSGi allows us to take advantage of the runtime benefits of modularity, such as hot deployments. In general, I want to focus this chapter on the design paradigm, not the tools and technologies. Hopefully, that resonates with you. Instead, shifts typically occur, and as things unfold, the modules become more apparent as development progresses. So to start, while the system is small, I favor coarser-grained and heavier-weight modules. Remember, the material in the first six chapters describe the essence of my motivation here. Prior to paying the bill, the system should apply a discount to the bill in an amount that has been negotiated with the payee we call this the process of auditing the bill. Applying this discount is a fairly complex process, and a 3rd party vendor has been commissioned that will apply this discount. Additionally, we must integrate with a legacy financials system that must be fed payment information for reconciliation. The initial class diagram can be seen in Figure 1. The AuditFacade integrates with the 3rd party vendor software that applies the discount, and the Payment class integrates with a legacy financial system. Of particular interest, note the bi-directional relationship between Bill and the AuditFacade and Payment classes â€" a sure sign of a problem that will haunt us later. This way, you can experiment with the system without running any scripts to create the database. If you are compelled to do so, feel free to add a real database backend. That repository can be found at http: In most systems we develop, we try to design layers that encapsulate specific behaviors and isolate certain types of change. Typical layers include a UI layer, a business or domain object layer, and a data access layer. In this system, we have these three layers. The UI layer is represented by the red classes in Figure 1. These classes are shown in blue in Figure 1. If I truly have a layered system, then I should be able to break out each layer into a separate module where modules in the upper layers depend on modules in lower layers, but not vice versa. Anyway, the end result is relatively simple to understand. Only a build script change so that certain classes are allocated to specific modules. The structure is shown in Figure 2. This refactoring was an example of applying the Physical Layers pattern. Listing 1 illustrates a portion of the initial build script where all of the classes were bundles into the WAR file. Here, we only show the business object layer. We created a new build target where we create the module and then added a new line to the dist target where that module is now included in the WAR file. Build Script with Physical Layers 8. Foremost, it proves that my class level architecture was decent. I was able to break the system out into modules for the various layers without having to change a bunch of code. Had there been violations in the layered structure, it would have been significantly more difficult pulling off this refactoring because I would have been forced to remove the unwanted dependencies. And as we progress, the amazing transformation of a system lacking modularity to a highly modularized version will unfold. This design has two fundamental flaws. The Bill is tightly coupled to the concrete AuditFacade class and the relationship is bi-directional. Audit method of the Bill Class Notice that the audit method actually creates the AuditFacade, calls the audit method, and passes a reference to Bill. While there are obvious technology reasons why we need to clean this up, there is also a motivating business force. The contract expires in 6 months, but we deliver the initial version of the system in 3 months. This solves the first half of our problem â€" the tight coupling between the Bill and AuditFacade implementation. The result is the class diagram shown in Figure 4. The modified audit method is shown in Listing 4. If we take

this flexible class structure and continue to deploy in the single bill. So what we really must do is separate the audit functionality out into a module separate from the bill. Separating the AuditFacade interface and AuditFacade1 implementation out into separate modules results in the diagram illustrated in Figure 5. We have a cyclic dependency between our bill. Remember our discussion from Chapter 5 â€" excessive dependencies are bad and cyclic dependencies are especially bad. We need to fix this problem. Abstract Modules This change will also be reflected in the build file that packages up these modules, as shown in Listing 5, where we can see the audit. The Build Script Creating the Audit. Upon applying this little trick, we can see that we have now removed the bi-directional relationship between Bill and the AuditFacade interface, as shown in Figure 6. Acyclic Relationships between Classes Whereas previously the AuditFacade accepted a Bill to the audit method, it now accepts an Auditable type. The new AuditFacade interface can be seen in the code snippet shown in Listing 6. The key element at this point is how we allocate these classes to their respective modules. Interfaces should be closer to the classes that use them, and farther away from the classes that implement them. Allocation of classes to modules is done at build time, so after modifying our build script, we have the module structure as shown in Figure 7. The first was to separate the bill and audit functionality out into separate modules. The second was to remove the cyclic dependency between the bill and audit modules. This offers us decent flexibility. We can test the audit. And if we wanted to, we could deploy the audit functionality separately ie. So overall, some decent progress. But some problems remain. While we can easily create a new AuditFacade implementation and allocate it to the audit. Levelized Modules Levelized Build 8. We can bundle it in the existing audit. Obviously, if we deploy a new AuditFacade implementation, we need the interface. In other words, because they are in the same module, they can only be managed together. We actually applied this pattern when allocating the Auditable interface to the audit. First, we have to answer the following question. Where do we put the AuditFacade interface so that new implementations of the interface can be managed separately from the interface and other implementations? Our answer can be found by looking at the module structure shown in Figure 9. Separate Abstractions We separate the AuditFacade interface out into its own module â€" auditspec. The AuditFacade1 and AuditFacade2 implementations are also placed in their own modules. After adding this new class, we modified our build script to allocate the classes to the appropriate modules. The AuditFacade implementation classes are allocated to separate modules, allowing us to manage the two independently. We can also test each independently and reuse each independent of the other. This change also increases the resiliency of the bill. We can test the bill. In general, we have completely eliminated the dependencies between the bill. Remember, at the beginning of Section 7. The presence of OSGi, however, brings the same degree of flexibility to the runtime that we have at development time, which we discussed in Chapter 3. With OSGi, we would be able to install and uninstall the audit1. Appendix B shows us how OSGi allows us to do this. By breaking the system out into modules, we ease the maintenance effort and increase overall system flexibility. Recall that in the third refactoring in Section 7. In the next step we will examine how we can decouple the bill. In general, we need to answer the question. Where do exceptions belong? The Collocate Exceptions pattern says that exceptions should be close to the classes or interfaces that throw them. Since the AuditFacade interface throws the exception, as shown in Listing 7, we should put the AuditException in the same module as the AuditFacade interface, which is the auditspec. AuditFacade Method that throws an AuditException 8. This next refactoring is rather interesting.

JAVA APPLICATION ARCHITECTURE MODULARITY PATTERNS WITH EXAMPLES USING OSGI pdf

7: Chapter 8 â€“ Reference Implementation : Java Application Architecture

OSGi. The OSGi Service Platform is the dynamic module system for Java. In OSGi parlance, a module is known as a www.enganchecubano.com provides a framework for managing bundles that are packaged as regular Java JAR files with an accompanying manifest.

The modularity patterns lay the foundation necessary to incorporate modular design thinking into your development initiatives. No module framework is necessary to use these patterns, and you already have many of the tools you need to design modular software. This refcard provides a quick reference to the 18 modularity patterns discussed in the book Java Application Architecture: The modularity patterns are not specific to the Java platform. They can be applied on any platform by treating the unit of release and deployment as the module. Each pattern includes a diagram except for base patterns , description, and implementation guidance. Fundamental modular design concepts upon which several other patterns exist. Used to help you manage dependencies between modules. Used to help you design modules that are easy to use. Used to help you design flexible modules that you can extend with new functionality. Used as tools to aid modular development. Modular Design Almost all well-known principles and patterns that aid software design address logical design. Identifying the methods of a class, relationships between classes, and the system package structure are all logical design issues. The vast majority of development teams spend their time dealing with logical design issues. A flexible logical design eases maintenance and increases extensibility. Logical design is just one piece of the software design and architecture challenge, however. The other is modular design, which focuses on the physical entities and the relationships between them. Identifying the entities containing your logical design constructs and managing dependencies between the units of deployment are examples of modular design. Without modular design, you may not realize the benefits you expect from your logical design. The modularity patterns help you: Design software that is extensible, reusable, maintainable, and adaptable. Design modular software today, in anticipation of future platform support for modularity. Break large software systems into a flexible composite of collaborating modules. Understand where to place your architectural focus Migrate large-scale monolithic applications to applications with a modular architecture. The Two Facets of Modularity There are two facets of modularity. The runtime model focuses on how to manage software systems at runtime. A module system, such as OSGi, is required to take advantage of the runtime model. The development model deals with how developers create modular software. The development model can be broken down into two subcategories. The programming model is how you interact with a module framework to take advantage of the runtime benefits of modularity. The design paradigm is the set of patterns you apply to design great modules. A module framework gives you runtime support and a programming model for modularity. The patterns in this refcard address the design paradigm and help you design modular software. Module Defined A software module is a deployable, manageable, natively reusable, composable, stateless unit of software that provides a concise interface to consumers. On the Java platform, a module is a JAR file, as depicted in the diagram. The patterns in this refcard help you design modular software and realize the benefits of modularity. Section 2 Base Patterns The base patterns are the fundamental elements upon which the other patterns exist. They establish the conscientious thought process that goes into designing systems with a modular architecture. They focus on modules as the unit of reuse, dependency management, and cohesion. Manage Relationships Design module relationships. Description A relationship between two modules exists when a class within one module imports at least a single class within another module. If changing the contents of a module, M2, may impact the contents of another module, M1, we can say that M1 has a physical dependency on M2. Excessive dependencies will make your modules more difficult to maintain, reuse, and test. Implementation Guidance Avoid modules with excessive incoming and outgoing dependencies. Modules with many incoming dependencies should be stable. That is, they should change infrequently Use module dependencies as a system of checks and balances. For instance, enforce relationships

between software layers using modularity see Physical Layers. Module Reuse Emphasize reusability at the module level. Description An oft-cited benefit of object-oriented development is reuse. Unfortunately, objects or classes are not an adequate reuse construct. The Reuse Release Equivalence Principle explains why. The unit of reuse is the unit of release. Modules are a unit of release and are, therefore, an excellent candidate as the unit of reuse. Implementation Guidance Separate horizontal modules those that span business domains from vertical modules those specific to a business domain. Module granularity and weight play a significant role in reuse. Fine-grained modules with external configuration come with a higher likelihood of reuse. But beware, these modules may be more difficult to use. Cohesive Modules Module behavior should serve a singular purpose. Description Cohesion is a measure of how closely related and focused the various responsibilities of a module are. Modules that lack cohesion are more difficult to maintain. Implementation Guidance Pay careful attention to how you allocate classes to their respective modules. Classes changing at the same rate and typically reused together belong in the same module. Classes changing at different rates and typically not reused together belong in separate modules. Section 3 Dependency Patterns The dependency patterns focus on managing the relationships between modules. They provide guidance on managing coupling that increase the likelihood of module reuse. Acyclic Relationships Module relationships must be acyclic. Description Cyclic relationships complicate the module structure. Apply the following rule to identify cyclic relationships. If beginning with module A, you can follow the dependency relationships between the set of modules that A is directly or indirectly dependent upon and you find any dependency on module A within that set, then a cyclic dependency exists between your module structure. You should avoid cyclic dependencies. Implementation Guidance Escalation breaks cycles by moving the cause of the cyclic dependency to a managing module at a higher level. Demotion breaks cycles by moving the cause of the cyclic dependency to a lower-level module. Callbacks break a cycle by defining an abstraction that is injected into the dependent module. This implementation resembles the Observer [GOF] pattern. Levelize Modules Module relationships should be levelized. Description Levelization is similar to layering, but is a finer-grained way to manage acyclic relationships between modules. With levelization, a single layer may contain multiple module levels. To levelize modules, do the following: Assign external modules level 0. Modules dependent only on level 0 modules are assigned level 1. Modules dependent on level 1 are assigned level 2. Implementation Guidance Levels are more granular than the layers in your system. Use levels to manage relationships within layers. Levelization demands module relationships be acyclic. You cannot levelize a module structure with cycles. A strict levelization scheme, where modules are dependent only on the level directly beneath it, is conceptually ideal but pragmatically difficult. Physical Layers Module relationships should not violate the conceptual layers. Description Layering a system helps ease maintenance and testability of the application. Common layers include presentation i. Any conceptually layered software system can be broken down into modules that correspond to these conceptual layers. Physical layers helps increase reusability because each layer is a deployable unit. Implementation Guidance Begin by creating a single coarse-grained module for each layer. Enforce the layers using Levelize Build. Break out each layer into more cohesive modules and use Levelize Modules to understand and manage the relationships within the layer. These modules will be at different levels. Container Independence Modules should be independent of the runtime container. Description Heavyweight modules are dependent upon a specific runtime environment and are difficult to reuse across contexts. Environmental dependencies also negatively affect your ability to test modules. Modules independent of the runtime container are more likely reused, and are more easily maintained and testable.

JAVA APPLICATION ARCHITECTURE MODULARITY PATTERNS WITH EXAMPLES USING OSGI pdf

8: Java Application Architecture: Modularity Patterns with Examples Using OSGi by Kirk Knoernschild

Java Application Architecture lays the foundation you'll need to incorporate modular design thinking into your development initiatives. Before it walks you through eighteen patterns that will help you architect modular software, it lays a solid foundation that shows you why modularity is a critical weapon in your arsenal of design tools.

But rest assured, this book is different. The way we develop Java applications is about to change, and this book explores the new way of Java application architecture. Modularity in software development is a concept that dates back to a seminal paper written by David Parnas in , and modularity as a more general concept dates back several civilizations. How is it that modularity is such a big deal in Java application architecture going forward? The answer is that modularity is coming to the Java platform, and it will change the way we develop Java applications. This book will help you Design modular software that is extensible, reusable, maintainable, and adaptable Design modular software today, in anticipation of platform support for modularity Break large software systems out into a flexible composite of collaborating modules Understand where to place your architectural focus Migrate large-scale monolithic applications to applications with a modular architecture Clearly articulate the advantages of modular software to your team Over the past several years, module frameworks have been gaining traction on the Java platform, and upcoming versions of Java will include a new module system that will allow you to leverage the power of modularity to build more resilient, malleable, extensible, and flexible software systems. Before it walks you through 18 patterns that will help you realize the benefits of modular software architecture, it lays a solid foundation that shows you why modularity is a critical weapon in your arsenal of design tools. Finally, it wraps up with several examples that demonstrate the patterns in action and the benefits. By beginning to design modular applications today, you are positioning yourself for the platform and architecture of tomorrow. I m dancing By god I m dancing on the walls. I m dancing on the ceiling. I m really, really pleased. From the Foreword by Robert C. Uncle Bob This isn t the first book on Java application architecture. No doubt it won t be the last. But rest assured, this title is different. The way we develop Java applications is about to change, and this title explores the new way of Java application architecture. Over the past several years, module frameworks have been gaining traction on the Java platform, and upcoming versions of Java will include a module system that allows you to leverage the power of modularity to build more resilient and flexible software systems. Modularity isn t a new concept. But modularity will change the way we develop Java applications, and you ll only be able to realize the benefits if you understand how to design more modular software systems. Java Application Architecture will help you Design modular software that is extensible, reusable, maintainable, and adaptable Design modular software today, in anticipation of future platform support for modularity Break large software systems into a flexible composite of collaborating modules Understand where to place your architectural focus Migrate large-scale monolithic applications to applications with a modular architecture Articulate the advantages of modular software to your team Java Application Architecture lays the foundation you ll need to incorporate modular design thinking into your development initiatives. Before it walks you through eighteen patterns that will help you architect modular software, it lays a solid foundation that shows you why modularity is a critical weapon in your arsenal of design tools. Throughout, you ll find examples that illustrate the concepts. By designing modular applications today, you are positioning yourself for the platform and architecture of tomorrow. That s why Uncle Bob is dancing. Introduces 18 new modular patterns, identified by the author - the next logical step in the evolution of object-oriented programming. These patterns help developers break up large software projects into smaller, more accessible chunks, or modules. This modular structure makes the software system less complex to develop, and easier to maintain and update. The design patterns introduced by the Gang of Four in the mid-nineties revolutionized the world of object-oriented programming, and of software development in general. Today most programmers use design patterns as a matter of course in their everyday work. More recently, folks such as Bob Martin have introduced the next level of abstraction, design principles,

which are rapidly gaining acceptance. Java Application Architecture takes this evolution of object-oriented programming to its next logical stage, introducing eighteen patterns for modular architecture, identified by the author. Kirk Knoernschild is one of the most widely respected thinkers and practitioners in the software development community. This book and these patterns are bound to be discussed - and widely used - by developers for years to come. Large software systems are inherently more complex to develop and maintain than smaller systems. Modularity involves breaking a large system into separate pieces that ultimately makes the system easier to understand. By understanding the behaviors contained within a module, and the dependencies that exist between modules, large-scale software is easier to develop, maintain, and modify.

JAVA APPLICATION ARCHITECTURE MODULARITY PATTERNS WITH EXAMPLES USING OSGI pdf

9: Book Review: Java Application Architecture

I'd like to think the modularity patterns in this book are also timeless. They are not tied to a specific platform or language. Whether you're using Java www.enganchecubano.com, OSGi, 2 or Jigsaw 3 or you want to build more modular software, the patterns in this book help you do that.

Now, I could be wrong. OSGi adoption is certainly increasing, albeit slowly. And as case studies begin to emerge that tout the cost reduction, improved responsiveness, and time-to-market advantages of OSGi, adoption will likely continue to rise. But adoption is one thing, disruption another, and I still have this nagging sensation that serves as cause for pause. What if something trendier, more fashionable surfaces, and OSGi is pushed into the backwaters? Not just evolutionary, but truly disruptive. Maybe cost reduction, improved responsiveness and time-to-market benefits will be enough. Quite possibly OSGi will flourish in the data center, as organizations seek more adaptable platforms that lend them these benefits. Because leveraging a platform built atop OSGi is separate from building modular software systems, even thoughOSGi enables both. It only means that the platform itself is adaptable. Of course, there is benefit in that. The Disruption OSGi has the potential to have have a much broader impact, affecting everyone from the developer to those working in the data center. So what might this trend be that will propel OSGi to stardom? But give me a chanceâ€¦let me explain. And these platforms span a range of markets, from mobile to social media. But each are successful in large part due to a thriving ecosystem. Apple In , Apple released their first generation iPhone. Without question, the device revolutionized the mobile phone industry. While the device offers a great user experience that has certainly played a role in its surging popularity, people flock to the iPhone today because of the wealth of applications available. With fewer preinstalled applications than the iPhone, Apple is counting on the ecosystem to drive adoption. The more consumers who flock to the device, the more developers who flock to the platform to deliver applications. As more applications become available, consumers will buy more iPads. The ecosystem fuels itself. Apple has simply provided the environment for the ecosystem to thrive. Eclipse In , the Eclipse team was thinking of ways to make Eclipse more dynamic. Their decision to use OSGi to create a rich client platform that supports plug-in architecture was the first step toward the resulting ecosystem we know today. One of the reasons developers use Eclipse is because there are an abundance of plug-ins available that allow them to do their jobs more effectively. Again, the ecosystem fuels itself. Before Hudson was CruiseControl. And while CruiseControl did help development teams get started on their path toward continuous integration, it was also unwieldy to use in many ways. Kohsuke created Hudson and gave the development community a new platform for continuous integration. With its plug-in architecture though, he also provided an environment that allows the Hudson ecosystem to thrive. Each are examples of social media tools with a strong developer community that creates extensions to the platform that users can leverage to enhance the experience. Each allows the ecosystem to thrive. And Others These are just a few examples. The ease with which WordPress themes and plug-ins can be developed and used to enhance a WordPress website is another example. In fact, many content management systems provide similar capabilities. A large reason why the Firefox web browser has emerged as the preferred web browser is the ease with which add-ons can be installed that extend the capabilities of the browser. Yet each are wildly successful because of two reasons: An environment was created that allowed an ecosystem to form and flourish. This environment includes a platform and a marketplace. A group of customers and developers converged on the marketplace and fueled growth of the platform. The result is a self-sustaining ecosystem. If you look at many of the more popular platforms that have emerged over the past decade, they tend to possess a similar characteristic â€” a community of individuals dedicated to providing great solutions leveraging the foundation of the platform. OSGi and modularity enables ecosystems on the Java platform. A strong ecosystem surrounding OSGi and modularity must leverage both. Developers would create reusable modules, implying they are designing modular software. For development teams to leverage these modules, they must be using a platform that

supports the runtime model. Certainly these ideas are not new. But there are also some striking differences between that which OSGi enables and the CBD fad that has come and largely gone, or whose promise was never fully realized. While some attempted to create components for the Java platform, the movement largely failed to go mainstream. IMHO, the answer is fairly simple. Even though numerous marketplaces emerged that allowed the consumer and producer to come together to buy and sell components, there was never a suitable component execution environment. That is, an environment that would support dynamic deployment, support for multiple versions, dependency management, and, in general, complete control over all components currently executing within the environment. ActiveX components did have an execution environment though did not support each of these capabilities , but Java did not. Today, in OSGi, Java has the requisite execution environment! To explain why it cannot work. Certainly there are a variety of different ways such an ecosystem could manifest itself. Possibly multiple ecosystems emerge like what we see in the mobile market today. But for a moment, imagine the world where you have the ability to easily assemble a platform from pre-built infrastructure modules that exactly meet the demands of your application. You might purchase these modules, you might choose to use open source modules, or you might build them yourself. The modules it depends upon? You develop your software modules using the sound principles and patterns of modular design to ensure loose coupling and high cohesion. As you roll out your business solution modules, you simultaneously deploy the additional infrastructure modules that are needed. In this marketplace, modules are sourced by multiple vendors. Neither the stack, nor your applications, are monolithic beasts. Instead, they are a composition of collaborating software modules. The option always exists for organizations to purchase modules from different providers, easily swapping one provider module out with another. Developers flock to sell their latest creation. Organizations seek to add amazing capabilities to their rightsized environment at a fraction of the cost compared to what they are accustomed to today. The business benefits are real. The technical advantages are real. And the resulting ecosystem is sustainable. A successful ecosystem demands both the runtime model and development model. And today, OSGi is the only standard technology that will allow this type of successful ecosystem to form on the Java platform. But will it happen? We may have a ways to go, but it sure would be cool! And it would be a shame if we lost this opportunity. OSGi is not new. Unfortunately, while OSGi is the only dynamic module system for the Java platform, adoption is not widespread. However, traction within the industry is building. Many products continue to emerge that leverage OSGi internally, and more are beginning to expose the virtues of OSGi to the developer community. Additional Notes Discuss the future of modularity and the impact OSGi will have on enabling and enforcing modularity, as well as how these patterns will be implemented and supported by OSGi. Note the increasing platform support and that development teams will be able to use it soon. Note the impact, not just on the developer and how applications are created, but also in how they are managed.

JAVA APPLICATION ARCHITECTURE MODULARITY PATTERNS WITH EXAMPLES USING OSGI pdf

Those Amazing Musical Instruments! With CD-ROM Advances in Unmanned Marine Vehicles (Iee Control Series (Iee Control Series) A Little Tour in France (Large Print Edition) Psychological factors associated with orofacial pains Text mining preprocessing techniques Animal biotechnology The Risks of Medical Innovation Little match girl Concepts and theories of memory John M. Gardiner Explorations in pragmatics: linguistic, cognitive and intercultural aspects Hot Winds from Bombay Using econometrics a practical guide by ah studenmund Fundamentals of cost accounting 5th edition solutions manual Focus on quantum mechanics Through Thick Thin (Adventures in Odyssey (Audio Numbered)) U.S. participation in African Development Fund. Counterpoint : social security is a wise investment IEEE International Symposium on Intelligent Control 1989: 25-26 September, Albany, New York Angelslayer: The Winnowing War At the Red summit Population ecology of individuals Constitution and by-laws of the Native Village of Perryville, Alaska Developing early literacy assessment and teaching The Russian House Illustrated stories from the bible Economics from the heart Korean musical instruments Curries (Essential Kitchen) Directing your interests We are all weird Story of Port Isaac, Port Quin and Port Gaverne Search for Joseph Tully Facts in a flash: Addition subtraction, grades 1-3 De fidiculis bibliographia The Amphora Pirates An Analysis of Potential Adjustments to the Veterans Equitable Resource Allocation (VERA System Beyond language and reason The Yale Editions of Horace Walpoles Correspondence, Volume 38 Dominikanie W Srodkowej Europie W XIII-XV Wieku Nine Lives Too Many