DZone Refcardz

Patterns of Modular Architecture

By Kirk Knoernschild

ABOUT THE MODULARITY PATTERNS

Module frameworks are gaining traction on the Java platform. Though modularity isn't a new concept, it promises to change the way we develop software applications. You'll only be able to realize the benefits of modularity if you understand how to design more modular software systems.

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: Modularity Patterns with Examples Using OSGi.*

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, except for base patterns, includes a diagram description, and implementation guidance.

Base Patterns: Fundamental modular design concepts upon which several other patterns exist.

Dependency Patterns: Used to help you manage dependencies between modules.

Usability Patterns: Used to help you design modules that are easy to use.

Extensibility Patterns: Used to help you design flexible modules that you can extend with new functionality.

Utility Patterns: Used as tools to aid modular development.

Logical vs. 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.

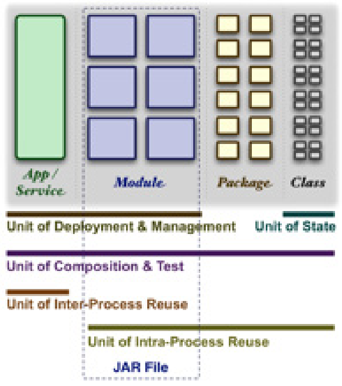
The runtime model focuses on how to manage software systems at runtime. A module system, such as OSGi, is required to take advantage of the runtime model.

The development model deals with how developers create modular software. The development model can be broken down into two sub-categories. 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. But a module framework won't help you design great software modules. 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.


App / Service
Module
Package
Class
Unit of Deployment & Management
Unit of State
Unit of Composition & Test
Unit of Inter-Process Reuse
Unit of Intra-Process Reuse
JAR File

AnswerHub | Social Q&A for the Enterprise

½ of the Top 10 StackExchange 1.0 Sites Now Run on AnswerHub

Discover Why Now!

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. In other words:

Hot Tip 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.

Hot Tip 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. Carefully consider each.
- 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

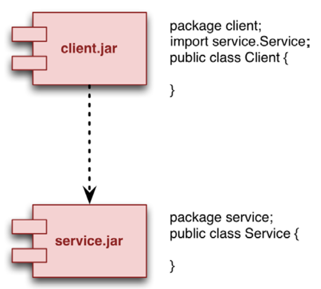
- Pay careful attention to how you allocate classes to their respective modules.
- Classes changing at the same rate and typically reused together belong in the same module.
- Classes changing at different rates and typically not reused together belong in separate modules.

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.



```
package client;
import service.Service;
public class Client {

}
```

```
package service;
public class Service {

}
```

Description
Cyclic relationships complicate the module structure. Apply the following rule to identify cyclic relationships.

Hot Tip 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.

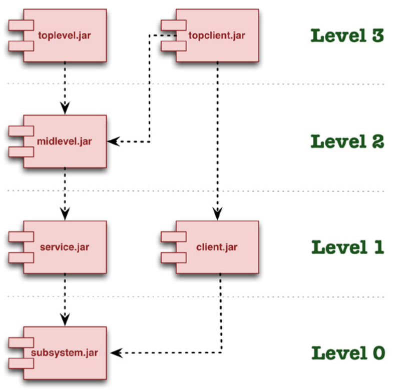
You should avoid cyclic dependencies.

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:

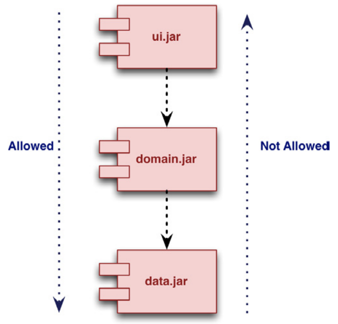
Assign external modules level 0. Modules dependent only on level 0 modules are assigned level 1. Modules dependent on level 1 are assigned level 2. Modules dependent on level n are assigned level n + 1.

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

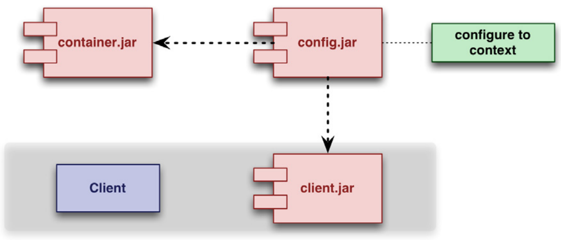
Layering a system helps ease maintenance and testability of the application. Common layers include presentation (i.e., user interface), domain (i.e., business), and data access. Any conceptually layered software system can be broken down into modules that correspond to these conceptual layers. Physical 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.
- It's fine if modules within a layer have relationships between them. These modules will be at different levels.

Container Independence

Modules should be independent of the runtime container.



Description

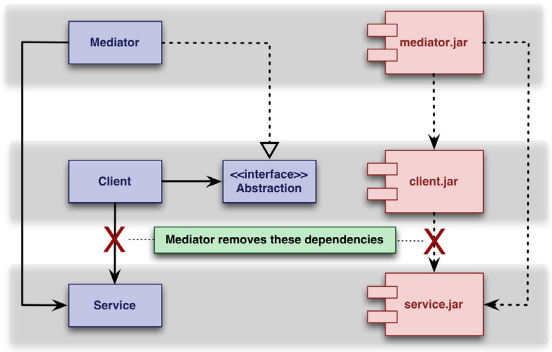
Heavyweight modules are dependent upon a specific runtime environment and are difficult to reuse across contexts. Environmental dependencies also negatively affect your ability to test modules. Modules independent of the runtime container are more likely reused, and are more easily maintained and testable.

Implementation Guidance

- Avoid importing container-dependent packages in your module's code.
- Use External Configuration to configure a module so that it can operate in different runtime environments.
- Use dependency injection to abstract container dependencies.

Independent Deployment

Modules should be independently deployable units.


Mediator
mediator.jar
Client
<<interface>> Abstraction
client.jar
Mediator removes these dependencies
Service
service.jar

Description

The less outgoing dependencies a module has, the easier the module is to reuse. A module with no outgoing dependencies is independently deployable and can be reused without the worry of identifying which additional modules might be necessary. Lower-level modules inherently have fewer outgoing dependencies and increase the opportunity for reuse.

Implementation Guidance

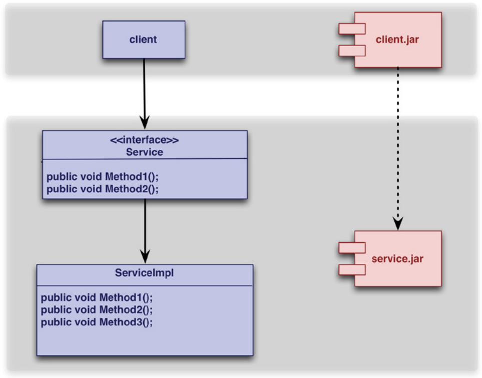
- Not all modules can be independently deployable. Some module dependencies are always necessary.
- In addition to reducing outgoing dependencies, container dependencies must also be minimized for those modules that are independently deployable.
- Highly cohesive modules are easier to make independently deployable units.

USABILITY PATTERNS

We want modules that other developers find easy to interact with. The usability patterns help design modules that are easy to understand and use.

Published Interface

Make a module's published interface well known.



Description

Modules should encapsulate implementation details so that other modules don't need to understand the implementation to use the module. A module's published interface exposes the capabilities you want to make available to other developers.

Hot Tip

A published interface consists of the public methods within the public classes within the "exported" packages that other modules are able to invoke.

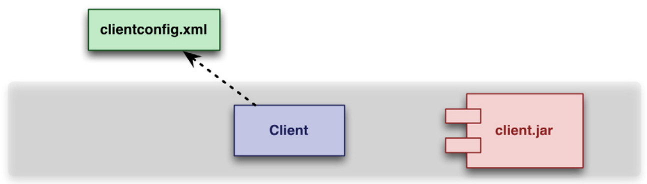
In standard Java, there is no way to explicitly state which packages a module exports, so it's difficult to enforce a published interface. Module frameworks, such as OSGi, shine in this situation.

Implementation Guidance

- In standard Java, document the published interface you expect other clients to invoke.
- In standard Java, expose the published interface via abstractions and discourage other developers from using the concrete classes.
- Module frameworks, such as OSGi, allow you to export packages and allow you to more easily enforce a published interface.

External Configuration

Modules should be externally configurable.



Description

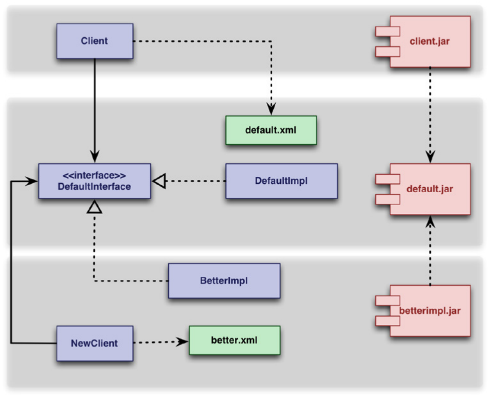
Module initialization typically requires configuring the module to its environmental context. Externalizing the configuration decreases context dependencies and allows you to use the module across a wider array of environments. External configuration increases a module's reuse, but makes it more difficult to use because developers must understand how to configure the module.

Implementation Guidance

- Use External Configuration to eliminate a module's environmental dependencies.
- Include a configuration file within the module that defines a default configuration, making the module easier to use.
- Remain cognizant of the tradeoff between increased reuse and decreased usability. In other words, maximizing reuse complicates use.

Default Implementation

Provide modules with a default implementation.



Description

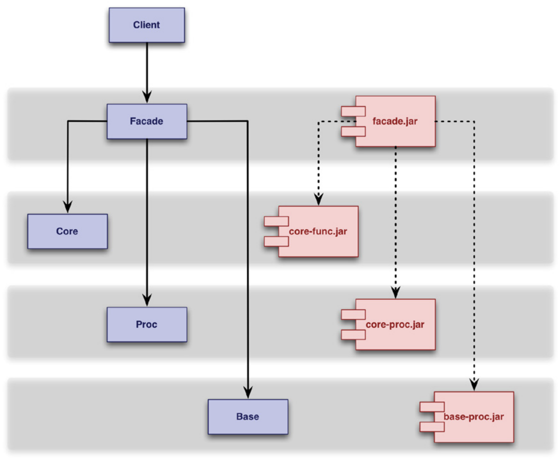
To maximize reuse, a module must be flexible enough so that it can function in a variety of different operating environments. Yet, making a module easier to use leads us to incorporate more functionality into a module so developers are required to do less when using the module. A default implementation with well-defined extension points helps address this tension.

Implementation Guidance

- When defining a Default Implementation, depend upon the abstract elements of a module (see Abstract Modules or Separate Abstractions).
- Include a default configuration in the module, but make the module externally configurable, as well.
- Always create a Test Module to test the default implementation.

Module Façade

Create a façade serving as a coarse-grained entry point to another fine-grained module's underlying implementation.



Description

Fine-grained and lightweight modules are inherently more reusable. But fine-grained modules are also typically dependent on several other modules. A Module Façade defines a higher level API that coordinates the work of a set of fine-grained modules. The façade emphasizes ease of use while the finer-grained modules emphasize reuse.

Implementation Guidance

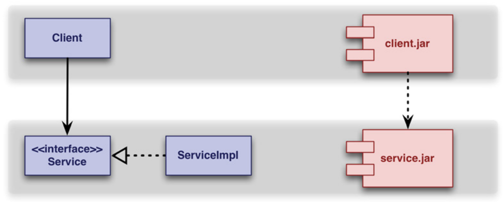
- Don't emphasize reuse of the façade. Use it to wire together and configure multiple fine-grained modules.
- Place context and environmental dependencies in the façade.
- Use the façade as an entry point for your integration tests.

EXTENSIBILITY PATTERNS

We want software that is easy to extend without modifying the existing codebase. We also want to deploy this new functionality without affecting other areas of the system. The extensibility patterns help us achieve this goal.

Abstract Modules

Depend upon the abstract elements of a module.



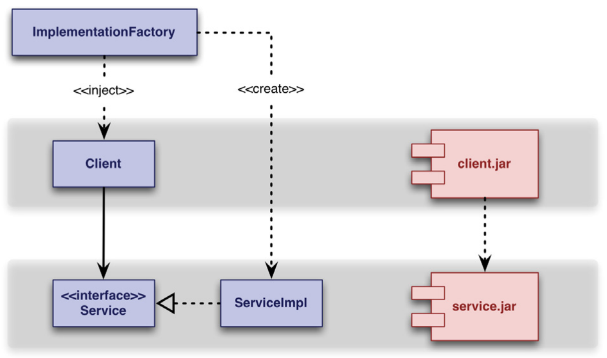
Description

Depending on the abstract elements of a module gives you greater opportunities to extend the system by defining new modules with classes that implement or extend the abstraction. Any clients of the module also have the ability to define their own implementations and to plug them into the module.

Implementation Guidance

- Use an Implementation Factory to create a module's underlying implementation.
- Use Abstract Modules when you have many incoming module dependencies and you want the flexibility to swap out underlying implementations.
- Strive to make the abstraction within a module as stable as possible. That is, avoid changes since it will have many many other modules that are dependent upon it.

Implementation Factory
Use factories to create a module's implementation classes.



Description
Any module whose classes depend upon the abstract elements of another module should avoid referencing any implementation classes. Doing so will compromise your module design. Consider the following rule.

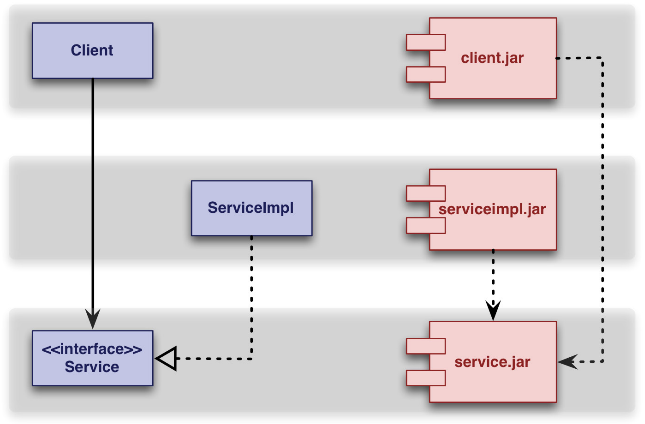
Hot Tip If a class depending on an abstraction must be changed in order to instantiate a new implementation of the abstraction, the design is flawed.

Furthermore, if a module must be changed to accommodate the instantiation, the module design is compromised.

Implementation Guidance

- The factory must be separated from the module containing the class instances it creates as well as the module dependent upon the abstractions.
- Externalize the creation of a module's implementation to a configuration file.
- Consider using a dependency injection framework like Spring, OSGi Blueprint, or OSGi Declarative Services. These will serve as your factory to wire together the appropriate implementations at runtime.

Separate Abstractions
Place abstractions and the classes that implement them in separate modules.



Description
Separating abstractions from their implementation offers the greatest flexibility to provide new implementations that completely replace existing implementations. With Separate Abstractions, you can define new behavior and plug it into your system without affecting existing system modules. Separate Abstractions can be used to develop a plug-in architecture. As a general guideline, apply the following rule.

Hot Tip Keep the abstraction closer to the classes that depend upon it and further from the classes that extend or implement it.

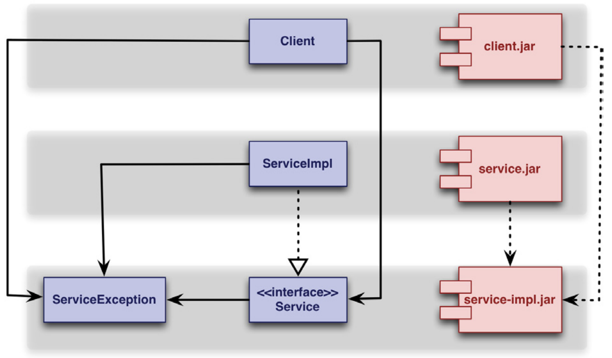
Implementation Guidance

- If all classes that depend upon an abstraction live in a single module, then place the classes and the abstraction in the same module.
- If the classes dependent upon an abstraction live in separate modules, place the abstraction in a module separate from the classes that depend upon it.
- Separating abstractions lends the ultimate flexibility to extend your system, but also increases its complexity.

UTILITY PATTERNS

The utility patterns are additional tools and techniques that aid modular development. They help you enforce your modular design and ensure quality.

Colocate Exceptions
Exceptions should be close to the class or interface that throws them.

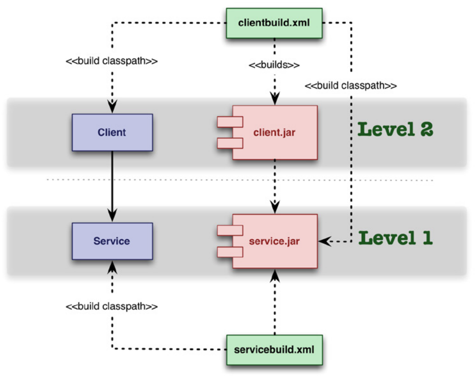


Description
Allocation of exceptions to modules has implications on module dependencies. Putting exceptions close to the classes that catch them creates a dependency from the module that throws the exception to the module containing the exception. Because invoking a method can introduce a module dependency, exceptions should be in the same module as the class containing the method that throws the exception.

Implementation Guidance

- Throw the exception on the interface or abstract class you're bound to and place the exception in the same module as the interface or abstract class.
- If abstractions across several modules throw the same exception, demote the exception to a completely separate module at a lower level.

Levelize Build
Execute the build in accordance with module levelization.



Description

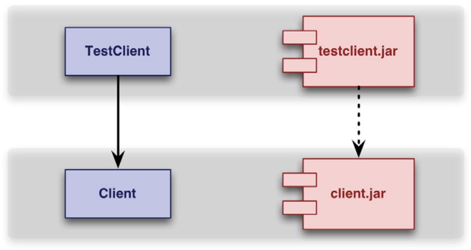
Enforcing module relationships is difficult. Though conceptually you may believe you have an acyclic module structure and fully comprehend the module relationships, a single build target with everything on the classpath allows undesirable cycles and dependencies to creep in. A levelized build helps you enforce your module dependencies. Any dependencies that violate your defined module structure will result in a build failure.

Implementation Guidance

- Avoid a full classpath build, where all classes are built using a single compile target.
- Define separate build targets for modules in different levels. Level 1 modules can be built with only external level 0 modules. At higher levels, include only the modules from lower levels that are required for a successful build.
- Defining new module dependencies will require modifying the build for that module. This is not necessarily undesirable.

Test Module

Each module should have a corresponding test module.



Description

Test modules contain all of the tests for the classes in a specific module. They allow you to test a module's underlying implementation. A Test Module may contain unit tests that test a module's classes, as well as integration tests that test the entire module's functionality.

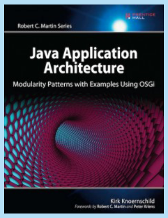
Implementation Guidance

- Depending on Abstract Modules makes it easier to define mocks and stubs for testing a module independently.
- For larger test suites, or situations where performance is paramount, separate different types of tests (i.e., unit tests, integration tests, performance tests, etc.) out into separate test modules.
- Ideally, you'll only include the test module and the module under test in the classpath when executing the tests. Pragmatically, some modules may require other dependent modules.

ABOUT THE AUTHOR

Kirk 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 2012 and presents 18 patterns that help you design modular software systems.

RECOMMENDED BOOK



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.

BUY HERE

ISBN-13: 978-1-936502-65-3
ISBN-10: 1-936502-65-8
50795
9 781936 502653
$7.95

Version 1.0