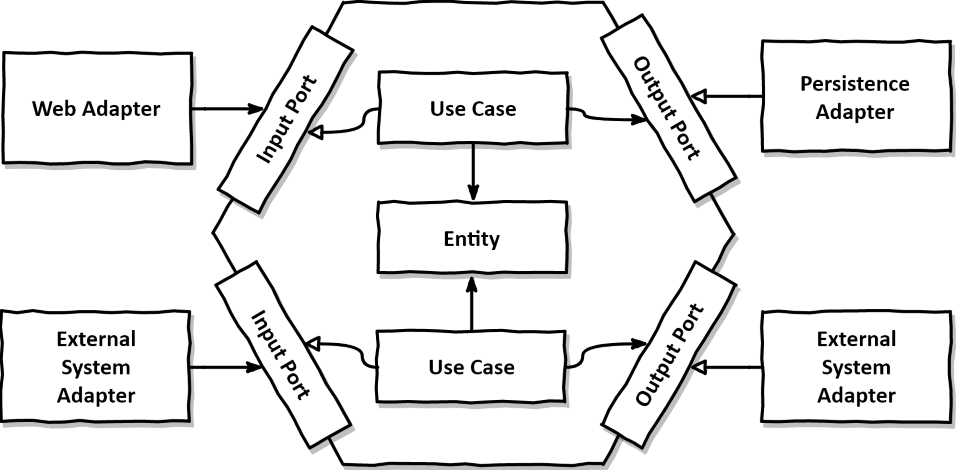
Hexagonal Architecture with Java and Spring

<https://reflectoring.io/spring-hexagonal/>

“六边形架构”的主要特征，与常见的分层架构风格相反，是我们组件之间的依赖关系指向“内向”，指向我们的领域对象:



六边形只是描述应用程序核心的一种奇特的方式，该核心由域对象、在其上操作的用例，以及向外部世界提供接口的输入和输出端口组成。

让我们看看这个体系结构风格中的每个原型。

Domain Objects

在具有丰富业务规则的领域中，域对象是应用程序的生命线。域对象可以包含状态和行为。行为越接近状态，代码就越容易理解、推理和维护。

域对象没有任何外部依赖。它们是纯Java，并为用例提供了操作它们的API。

因为域对象不依赖于应用程序的其他层，所以其他层的更改不会影响它们。它们可以在没有依赖关系的情况下进化。这是单一责任原则(“SOLID”中的“S”)的一个主要例子，该原则声明组件应该只有一个更改的理由。对于我们的域对象，这个原因是业务需求的变化。

拥有单一的职责可以让我们演进我们的领域对象，而不必考虑外部依赖关系。这种可演变性使六边形架构风格非常适合于实践领域驱动设计。在开发时，我们只是遵循依赖关系的自然流程:我们在域对象中开始编码，并从那里向外扩展。如果这不是领域驱动的，那么我不知道什么才是。

Use Cases

我们知道用例是对用户用我们的软件做什么的抽象描述。在六角形架构风格中，将用例推广到我们代码库的一级公民是有意义的。

从这个意义上说，用例是处理与某个用例有关的所有事情的类。作为一个例子，让我们考虑一个银行应用程序中的用例“将钱从一个帐户发送到另一个帐户”。我们将创建一个类SendMoneyUseCase，它具有一个独特的API，允许用户转账。代码包含特定于用例的所有业务规则验证和逻辑，因此不能在域对象中实现。其他一切都委托给域对象(例如，可能有一个域对象帐户)。

与域对象类似，用例类不依赖于外部组件。当它需要六边形之外的东西时，我们创建一个输出端口。

Input and Output Ports

域对象和用例位于六边形内，即在应用程序的核心内。每次与外部的通信都是通过专用的“端口”进行的。

输入端口是一个简单的接口，可以由外部组件调用，并由用例实现。调用这种输入端口的组件称为输入适配器或“驱动”适配器。

输出端口也是一个简单的接口，如果我们的用例需要从外部(例如数据库访问)调用它。这个接口是为满足用例的需要而设计的，但它是由称为输出或“驱动”适配器的外部组件实现的。如果您熟悉SOLID原则，那么这是一个依赖倒置原则(SOLID中的“D”)的应用程序，因为我们正在使用一个接口将依赖关系从用例倒置到输出适配器。

有了输入和输出端口，我们就有了数据输入和离开系统的截然不同的地方，这使得对架构的推理变得很容易。

Adapters

适配器构成六边形架构的外层。他们不是核心的一部分，而是与核心互动。

输入适配器或“驱动”适配器调用输入端口来完成某些工作。例如，输入适配器可以是一个web接口。当用户单击浏览器中的按钮时，web适配器调用某个输入端口来调用相应的用例。

输出适配器或“驱动”适配器由我们的用例调用，并且可能，例如，提供来自数据库的数据。输出适配器实现一组输出端口接口。请注意，接口是由用例指定的，而不是由用例指定的。（Note that the interfaces are dictated by the use cases and not the other way around.）

适配器使交换应用程序的某一层变得很容易。如果应用程序应该可以从胖客户机使用到web，那么我们添加一个胖客户机输入适配器。如果应用程序需要不同的数据库，则添加一个新的持久性适配器，实现与旧适配器相同的输出端口接口。

Show Me Some Code!

在简要介绍了上面的六边形架构风格之后，让我们最后看一看一些代码。将架构风格的概念转换成代码总是取决于解释和风格，所以请不要参考下面给出的代码示例，而是作为创建自己风格的灵感。

代码示例都来自我在GitHub上的“BuckPal”示例应用程序，围绕着从一个账户向另一个账户转账的用例。为了本文的目的，一些代码片段做了轻微的修改，所以看看原始代码的回购。

Building a Domain Object

我们首先构建一个为我们的用例服务的域对象。我们创建了一个Account类，用于管理账户的取款和存款:

@AllArgsConstructor(access = AccessLevel.PRIVATE)

public class Account {

@Getter private final AccountId id;

@Getter private final Money baselineBalance;

@Getter private final ActivityWindow activityWindow;

public static Account account(

AccountId accountId,

Money baselineBalance,

ActivityWindow activityWindow) {

return new Account(accountId, baselineBalance, activityWindow);

}

public Optional<AccountId> getId(){

return Optional.ofNullable(this.id);

}

public Money calculateBalance() {

return Money.add(

this.baselineBalance,

this.activityWindow.calculateBalance(this.id));

}

public boolean withdraw(Money money, AccountId targetAccountId) {

if (!mayWithdraw(money)) {

return false;

}

Activity withdrawal = new Activity(

this.id,

this.id,

targetAccountId,

LocalDateTime.now(),

money);

this.activityWindow.addActivity(withdrawal);

return true;

}

private boolean mayWithdraw(Money money) {

return Money.add(

this.calculateBalance(),

money.negate())

.isPositiveOrZero();

}

public boolean deposit(Money money, AccountId sourceAccountId) {

Activity deposit = new Activity(

this.id,

sourceAccountId,

this.id,

LocalDateTime.now(),

money);

this.activityWindow.addActivity(deposit);

return true;

}

@Value

public static class AccountId {

private Long value;

}

}

一个帐户可以有许多相关联的活动，每个活动都表示对该帐户的取款或存款。因为我们并不总是想加载给定帐户的所有活动，所以我们将其限制在特定的活动窗口。为了仍然能够计算帐户的总余额，帐户类具有baselineBalance属性，其中包含活动窗口开始时帐户的余额。

正如您在上面的代码中所看到的，我们构建域对象时完全不依赖于架构的其他层。我们可以自由地按照我们认为合适的方式对代码进行建模，在这种情况下，创建一个非常接近模型状态的“丰富”行为，使其更容易理解。

如果我们愿意，我们可以在我们的领域模型中使用外部库，但是这些依赖关系应该相对稳定，以防止强制更改我们的代码。例如，在上面的例子中，我们包含了Lombok注释。

Account类现在允许我们在单个帐户上取款和存款，但是我们希望在两个帐户之间转账。因此，我们创建一个用例类来为我们编排这个。

Building an Input Port

然而，在我们实际实现用例之前，我们创建了该用例的外部API，它将成为我们六边形架构中的输入端口:

public interface SendMoneyUseCase {

boolean sendMoney(SendMoneyCommand command);

@Value

@EqualsAndHashCode(callSuper = false)

class SendMoneyCommand extends SelfValidating<SendMoneyCommand> {

@NotNull

private final AccountId sourceAccountId;

@NotNull

private final AccountId targetAccountId;

@NotNull

private final Money money;

public SendMoneyCommand(

AccountId sourceAccountId,

AccountId targetAccountId,

Money money) {

this.sourceAccountId = sourceAccountId;

this.targetAccountId = targetAccountId;

this.money = money;

this.validateSelf();

}

}

}

通过调用sendMoney()，我们的应用程序核心之外的适配器现在可以调用这个用例。

我们将所需的所有参数聚合到SendMoneyCommand值对象中。这允许我们在value对象的构造函数中执行输入验证。在上面的示例中，我们甚至使用了Bean验证注释@NotNull，它是在validatself()方法中验证的。这样，实际的用例代码就不会被嘈杂的验证代码污染了。

现在我们需要这个接口的实现。

Building a Use Case and Output Ports

在用例实现中，我们使用域模型从源帐户取款，并向目标帐户存款:

@RequiredArgsConstructor

@Component

@Transactional

public class SendMoneyService implements SendMoneyUseCase {

private final LoadAccountPort loadAccountPort;

private final AccountLock accountLock;

private final UpdateAccountStatePort updateAccountStatePort;

@Override

public boolean sendMoney(SendMoneyCommand command) {

LocalDateTime baselineDate = LocalDateTime.now().minusDays(10);

Account sourceAccount = loadAccountPort.loadAccount(

command.getSourceAccountId(),

baselineDate);

Account targetAccount = loadAccountPort.loadAccount(

command.getTargetAccountId(),

baselineDate);

accountLock.lockAccount(sourceAccountId);

if (!sourceAccount.withdraw(command.getMoney(), targetAccountId)) {

accountLock.releaseAccount(sourceAccountId);

return false;

}

accountLock.lockAccount(targetAccountId);

if (!targetAccount.deposit(command.getMoney(), sourceAccountId)) {

accountLock.releaseAccount(sourceAccountId);

accountLock.releaseAccount(targetAccountId);

return false;

}

updateAccountStatePort.updateActivities(sourceAccount);

updateAccountStatePort.updateActivities(targetAccount);

accountLock.releaseAccount(sourceAccountId);

accountLock.releaseAccount(targetAccountId);

return true;

}

}

基本上，用例实现从数据库装载源帐户和目标帐户，锁定帐户，使其他事务不能同时发生，进行取款和存款，最后将帐户的新状态写回数据库。

此外，通过使用@Component，我们将该服务变成一个Spring bean，可以注入到任何需要访问SendMoneyUseCase输入端口的组件中，而不需要依赖于实际的实现。

为了从数据库加载和存储帐户，该实现依赖于输出端口LoadAccountPort和UpdateAccountStatePort，这是我们稍后将在持久性适配器中实现的接口。

输出端口接口的形状由用例决定。在编写用例时，我们可能会发现需要从数据库中加载某些数据，因此我们为它创建了一个输出端口接口。当然，这些端口可能在其他用例中被重用。在我们的例子中，输出端口如下所示:

public interface LoadAccountPort {

Account loadAccount(AccountId accountId, LocalDateTime baselineDate);

}

public interface UpdateAccountStatePort {

void updateActivities(Account account);

}

Building a Web Adapter

有了域模型、用例以及输入和输出端口，我们现在已经完成了应用程序的核心(即六边形内的所有内容)。然而，如果我们不把它与外部世界连接起来，这个核心就无法帮助我们。因此，我们构建一个适配器，通过一个REST API公开我们的应用核心:

@RestController

@RequiredArgsConstructor

public class SendMoneyController {

private final SendMoneyUseCase sendMoneyUseCase;

@PostMapping(path = "/accounts/send/{sourceAccountId}/{targetAccountId}/{amount}")

void sendMoney(

@PathVariable("sourceAccountId") Long sourceAccountId,

@PathVariable("targetAccountId") Long targetAccountId,

@PathVariable("amount") Long amount) {

SendMoneyCommand command = new SendMoneyCommand(

new AccountId(sourceAccountId),

new AccountId(targetAccountId),

Money.of(amount));

sendMoneyUseCase.sendMoney(command);

}

}

如果您熟悉Spring MVC，您会发现这是一个非常无聊的web控制器。它只是从请求路径读取所需的参数，将它们放入SendMoneyCommand中，然后调用用例。在更复杂的场景中，web控制器还可能检查身份验证和授权，并执行更复杂的JSON输入映射。

上面的控制器通过将HTTP请求映射到用例的输入端口向世界公开我们的用例。现在让我们看看如何通过连接输出端口将应用程序连接到数据库。

Building a Persistence Adapter

输入端口由用例服务实现，而输出端口由持久性适配器实现。假设我们使用Spring Data JPA作为管理代码库持久性的首选工具。实现输出端口LoadAccountPort和UpdateAccountStatePort的持久性适配器可能会像这样:

@RequiredArgsConstructor

@Component

class AccountPersistenceAdapter implements

LoadAccountPort,

UpdateAccountStatePort {

private final AccountRepository accountRepository;

private final ActivityRepository activityRepository;

private final AccountMapper accountMapper;

@Override

public Account loadAccount(

AccountId accountId,

LocalDateTime baselineDate) {

AccountJpaEntity account =

accountRepository.findById(accountId.getValue())

.orElseThrow(EntityNotFoundException::new);

List<ActivityJpaEntity> activities =

activityRepository.findByOwnerSince(

accountId.getValue(),

baselineDate);

Long withdrawalBalance = orZero(activityRepository

.getWithdrawalBalanceUntil(

accountId.getValue(),

baselineDate));

Long depositBalance = orZero(activityRepository

.getDepositBalanceUntil(

accountId.getValue(),

baselineDate));

return accountMapper.mapToDomainEntity(

account,

activities,

withdrawalBalance,

depositBalance);

}

private Long orZero(Long value){

return value == null ? 0L : value;

}

@Override

public void updateActivities(Account account) {

for (Activity activity : account.getActivityWindow().getActivities()) {

if (activity.getId() == null) {

activityRepository.save(accountMapper.mapToJpaEntity(activity));

}

}

}

}

适配器实现了所实现的输出端口所需的loadAccount()和updateActivities()方法。它使用Spring数据存储库从数据库加载数据并将数据保存到数据库中，并使用一个accountapper将帐户域对象映射到AccountJpaEntity对象，AccountJpaEntity对象表示数据库中的帐户。

同样，我们使用@Component将它变成一个可以注入到上面用例服务中的Spring bean。

这样做值得吗?

人们经常问自己，这样的架构是否值得付出努力(包括我自己)。毕竟，我们必须创建端口接口，并且我们有x在域模型的多种表示之间进行映射。web适配器中可能有一个域模型表示，持久性适配器中可能有另一个域模型表示。

那么，这种努力值得吗?

作为一名专业顾问，我的回答当然是“视情况而定”。

如果我们正在构建一个仅仅存储和保存数据的CRUD应用程序，那么像这样的架构可能会带来开销。如果我们正在构建一个具有丰富业务规则的应用程序，这些业务规则可以用结合状态和行为的丰富域模型来表达，那么这个体系结构就会非常出色，因为它将域模型置于事物的中心。

的深入了解

上面只是给出了一个六边形架构在实际代码中可能是什么样子的概念。还有其他方法可以做到这一点，所以请随意尝试并找到最适合您需要的方法。此外，web和持久性适配器只是外部适配器的示例。可能会有到其他第三方系统或其他面向用户的前端的适配器。

如果您想深入了解这个主题，可以看看我的书，它更详细地介绍了测试、映射策略和快捷方式等内容。

关注我的推特，了解更多如何成为一名更好的软件开发人员的建议。

<https://reflectoring.io/spring-boot-gradle-multi-module/#spring-boot-application-build-file>

实现代码

<https://github.com/jainakshat072/work>

<https://github.com/venustuspl/hexagon-architecture-example>

<https://github.com/lmartinezn/buckpal>

├── adapters

| ├── buckpal-persistence

| | ├── src

| | └── build.gradle

| └── buckpal-web

| ├── src

| └── build.gradle

├── buckpal-application

| ├── src

| └── build.gradle

├── common

| ├── src

| └── build.gradle

├── buckpal-configuration

| ├── src

| └── build.gradle

├── build.gradle

└── settings.gradle

顶级的构建。gradle file配置了所有子模块之间共享的构建行为，这样我们就不必在子模块中复制东西。

buckpal-configuration模块包含实际的Spring引导应用程序和将Spring应用程序上下文组合在一起的任何Spring Java配置。要创建应用程序上下文，它需要访问其他模块，每个模块都提供应用程序的某些部分。我还在其他上下文中看到过这个称为基础设施的模块。

公共模块提供了可以被所有其他模块访问的某些类。

buckpal-application模块包含组成“应用程序层”的类:实现查询和修改域模型的用例的服务。

适配器/buckpal-web模块实现了应用程序的web层，该层可以调用应用程序模块中实现的用例。

adapters/buckpal-persistence模块实现了我们应用程序的持久化层。

settings.gradle file in the parent folder:

include 'common'

include 'adapters:buckpal-web'

include 'adapters:buckpal-persistence'

include 'buckpal-configuration'

include 'buckpal-application'

例如，公共模块将在所有其他模块之前构建，因为所有其他模块都依赖于它。

在父构建中。gradle文件，我们现在定义了基本配置，所有子模块共享:

plugins {

id "io.spring.dependency-management" version "1.0.8.RELEASE"

}

subprojects {

group = 'io.reflectoring.reviewapp'

version = '0.0.1-SNAPSHOT'

apply plugin: 'java'

apply plugin: 'io.spring.dependency-management'

apply plugin: 'java-library'

repositories {

jcenter()

}

dependencyManagement {

imports {

mavenBom("org.springframework.boot:spring-boot-dependencies:2.1.7.RELEASE")

}

}

}

首先，我们包含了Spring Dependency Management插件，它为我们提供了以后会用到的dependencyManagement闭包。

然后，我们在subprojects闭包中定义一个共享配置。子项目中的所有内容都将应用于所有子模块。

子项目中最重要的部分是依赖性管理闭包。在这里，我们可以在某个版本中定义对Maven工件的任何依赖关系。如果我们需要子模块中的某个依赖项，我们可以在子模块中指定它，而不需要提供版本号，因为版本号将从dependencyManagement闭包中加载。

我们在示例中添加的唯一依赖项是对Spring Boot的Maven BOM(物料清单)的依赖项。这个BOM包含了与给定的Spring Boot版本(2.1.7)兼容的确切版本中Spring Boot应用程序可能需要的所有依赖项。在这种情况下释放)。因此，我们不需要列出我们自己的每一个依赖项，从而可能得到错误的版本。

另外，请注意，我们将java库插件应用于所有子模块。这使我们能够使用允许我们定义更细粒度依赖范围的api和实现配置。

模块构建文件

在模块构建文件中，我们现在只需添加模块所需的依赖项。

文件适配器/ buckpal-persistence /构建。gradle是这样的:

dependencies {

implementation project(':common')

implementation project(':buckpal-application')

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

// ... more dependencies

}

持久性模块依赖于公共模块和应用程序模块。所有模块都使用通用模块，因此这种依赖性是很自然的。对应用程序模块的依赖源于我们遵循的六边形架构风格，在这种风格中，持久化模块实现了位于应用程序层的接口，因此充当了应用程序层的持久化“插件”。

然而，更重要的是，我们添加了Spring - Boot -starter- Data - JPA的依赖关系，它为Spring Boot应用程序提供了Spring Data JPA支持。注意，我们没有添加版本号，因为版本是自动从父构建文件中的spring-boot-dependencies BOM解析的。在本例中，我们将获得与Spring Boot 2.1.7 release兼容的版本。

注意，我们在实现配置中添加了spring-boot-starter-data-jpa依赖项。这意味着此依赖关系不会泄漏到包含持久化模块作为依赖关系的模块的编译时。这可以防止我们在不需要JPA的模块中意外地使用JPA类。

web层的构建文件，适配器/buckpal-web/build。gradle看起来很相似，只是依赖于spring-boot-starter-web:

dependencies {

implementation project(':common')

implementation project(':application')

implementation 'org.springframework.boot:spring-boot-starter-web'

// ... more dependencies

}

我们的模块可以访问它们为Spring Boot应用程序构建web或持久层所需的所有类，而没有不必要的依赖。

web模块对持久性一无所知，反之亦然。作为一名开发人员，我们不可能在无意中向web层添加持久性代码，或者在不有意识地向构建添加依赖性的情况下向持久性层添加web代码。gradle文件。这有助于避免可怕的大泥球。

Spring Boot Application Build File

现在，我们所要做的就是将这些模块聚合到单个Spring Boot应用程序中。我们在巴克帕尔构型模块中做这个。

在buckpal-configuration /构建。在gradle构建文件中，我们将依赖关系添加到所有模块中:

plugins {

id "org.springframework.boot" version "2.1.7.RELEASE"

}

dependencies {

implementation project(':common')

implementation project(':buckpal-application')

implementation project(':adapters:buckpal-persistence')

implementation project(':adapters:buckpal-web')

implementation 'org.springframework.boot:spring-boot-starter'

// ... more dependencies

}

我们还添加了Spring Boot Gradle插件，它给我们提供了bootRun Gradle任务。现在我们可以使用。/gradlew bootRun启动Gradle应用程序。

同样，我们将必须的@ springbootapplication注释类添加到buckpal-configuration模块的源文件夹中:

@SpringBootApplication

public class BuckPalApplication {

public static void main(String[] args) {

SpringApplication.run(BuckPalApplication.class, args);

}

}

这个类需要访问spring-boot-starter依赖项赋予我们访问的SpringBootApplication类和SpringApplication类。