عنوان: **اصل معکوس سازی وابستگی‌ها**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۲۵ ۹:۵
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, Dependency Injection, IoC

پیش از شروع این سری نیاز است با تعدادی از واژه‌های بکار رفته در آن به اختصار آشنا شویم؛ از این واژه‌ها به کرات استفاده شده و در طول دوره به بررسی جزئیات آن‌ها خواهیم پرداخت:

1) Dependency inversion principle یا DIP (اصل معکوس سازی وابستگی‌ها)
DIP یکی از اصول طراحی نرم افزار است و D آن همان D معروف SOLID است (اصول پذیرفته شده شیءگرایی).

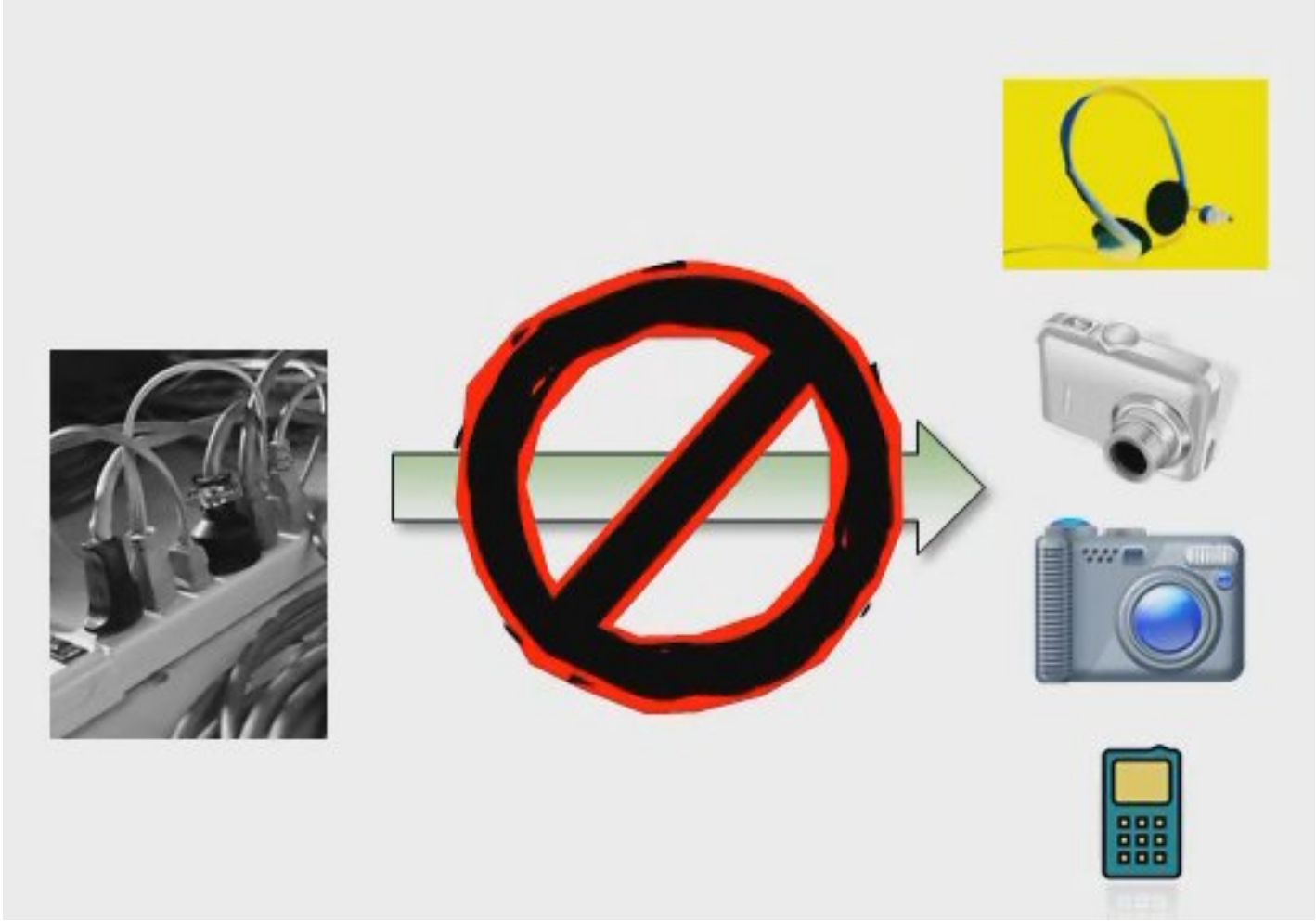
2) Inversion of Control یا IOC (معکوس سازی کنترل)
الگویی است که نحوه پیاده سازی DIP را بیان می‌کند.

3) Dependency injection یا DI (تزریق وابستگی‌ها)
یکی از روش‌های پیاده سازی IOC است.

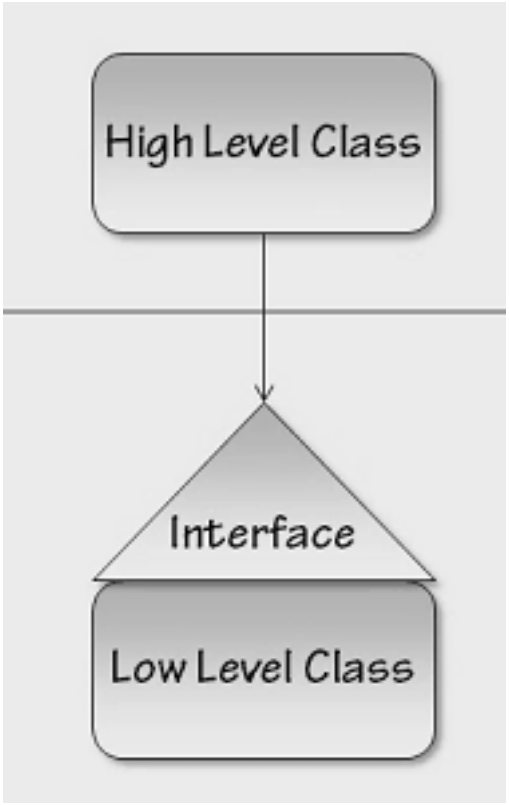
4) IOC container
به فریم ورک‌هایی که کار DI را انجام می‌دهند گفته می‌شود.

### Dependency inversion principle چیست؟

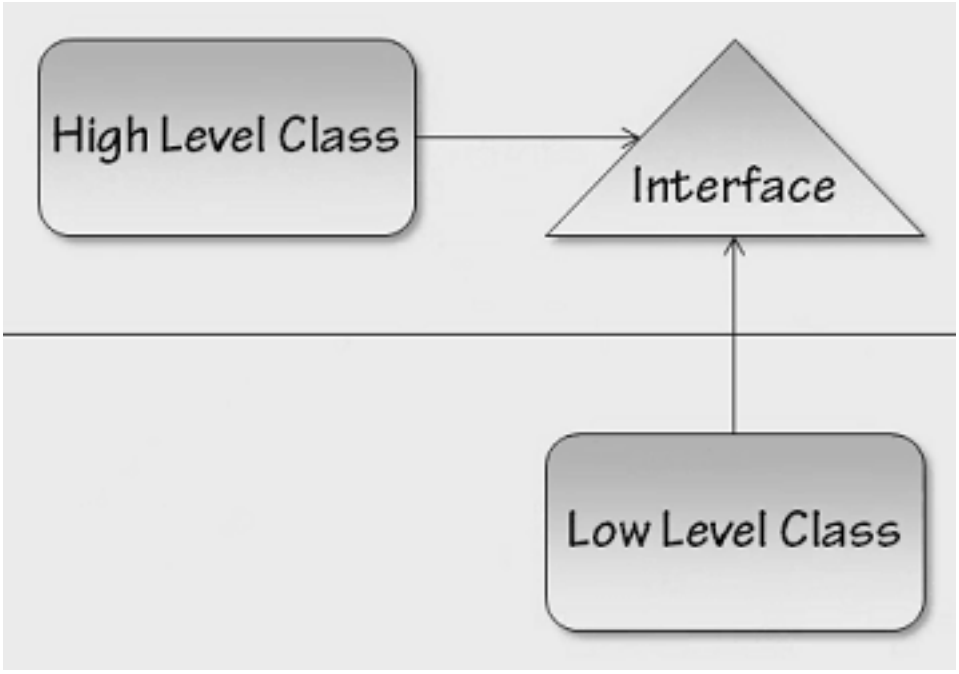
اصل معکوس سازی وابستگی‌ها به این معنا است که بجای اینکه ماژول‌های سطح پایین سیستم، رابط‌های قابل استفاده‌ای از خود را در اختیار سطوح بالاتر سیستم قرار دهند، ماژول‌های قرار گرفته در سطوحی بالاتر، اینترفیس‌هایی را تعریف می‌کنند که توسط ماژول‌های سطح پایین پیاده سازی خواهند شد.
همانطور که ملاحظه می‌کنید به این ترتیب وابستگی‌های سیستم معکوس خواهند شد. نمونه‌ای از عدم استفاده از این طراحی را در دنیای واقعی به صورت رومزه با آن‌ها سر و کار داریم؛ مانند وسایل الکترونیکی قابل حملی که نیاز به شارژ مجدد دارند. برای مثال تلفن‌های همراه، دوربین‌های عکاسی دیجیتال و امثال آن.



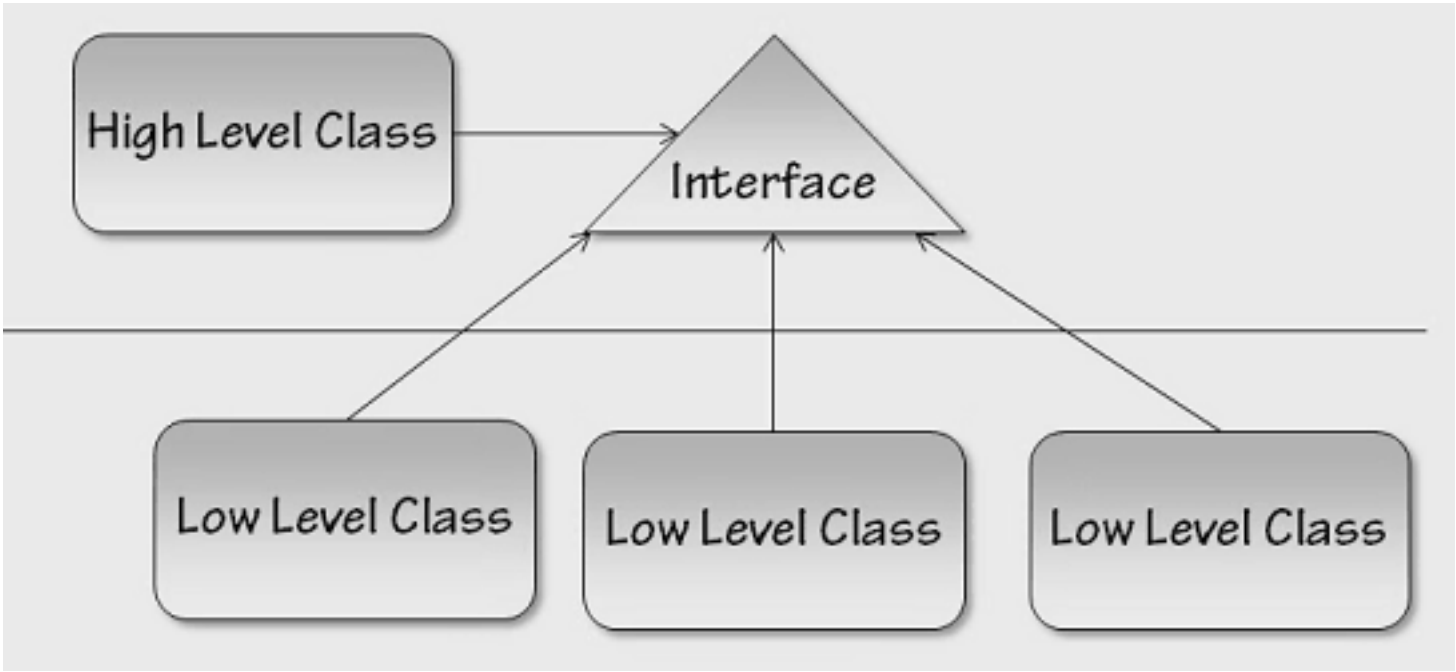
هر کدام از این‌ها، رابط‌های اتصالی متفاوتی دارند. یکی USB2، یکی USB3 دیگری Mini USB و بعضی‌ها هم از پورت‌های دیگری استفاده می‌کنند. چون هر کدام از لایه‌های زیرین سیستم (در اینجا وسایل قابل شارژ) رابط‌های اتصالی مختلفی را ارائه داده‌اند، برای اتصال آن‌ها به منبع قدرت که در سطحی بالاتر قرار دارد، نیاز به تبدیلگرها و درگاه‌های مختلفی خواهد بود.
اگر در این نوع طراحی‌ها، اصل معکوس سازی وابستگی‌ها رعایت می‌شد، درگاه و رابط اتصال به منبع قدرت باید تعیین کننده نحوه طراحی اینترفیس‌های لایه‌های زیرین می‌بود تا با این آشفتگی نیاز به انواع و اقسام تبدیلگرها، روبرو نمی‌شدیم.



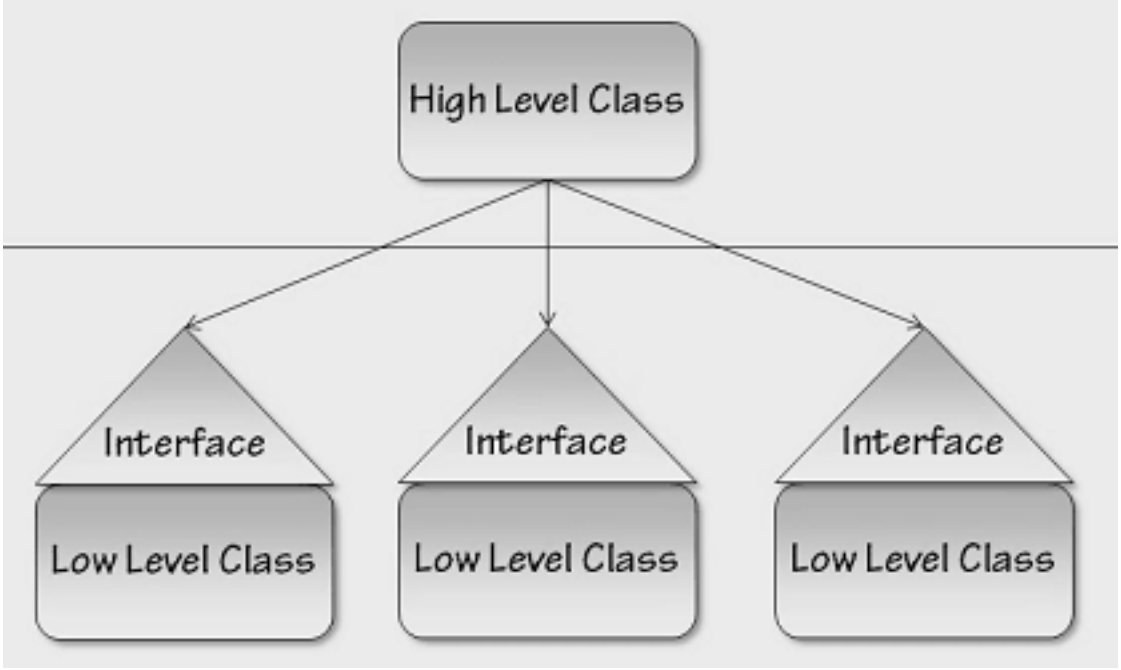

اگر وابستگی‌ها معکوس نشوند مطابق تصویر فوق، کلاس سطح بالایی را خواهیم داشت که به اینترفیس کلاس‌های سطح پایین وابسته است. البته در اینجا اینترفیس یک کلمه عمومی است و بیشتر نحوه در معرض دید و استفاده قرار دادن اعضای یک کلاس مد نظر بوده است تا اینکه مثلا الزاما اینترفیس‌های زبان خاصی مدنظر باشند.
مشکلی که در این حالت به زودی بروز خواهد کرد، افزایش کلاس‌های سطح پایین و بیشتر شدن وابستگی کلاس‌های سطح بالا به آن‌ها است. به این ترتیب قابلیت استفاده مجدد خود را از دست خواهند داد.



در تصویر فوق حالتی را مشاهده می‌کنید که وابستگی‌ها معکوس شده‌اند. تغییر مهمی که در اینجا نسبت به حالت قبل رخ داده است، بالا بردن اینترفیس، به بالای خط میانی است که در تصویر مشخص گردیده است. این خط، معرف تعریف لایه‌های مختلف سیستم است. به عبارتی کلاس‌های سطح بالا در لایه دیگری نسبت به کلاس‌های سطح پایین قرار دارند. در اینجا اجازه داده‌ایم تا کلاس لایه بالایی اینترفیس مورد نیاز خود را تعریف کند. این نوع اینترفیس‌ها در زبان سی شارپ می‌توانند یک کلاس `Abstract` و یا حتی یک `Interface` متداول باشند.
با معکوس شدن وابستگی‌ها، لایه سطح بالا است که به لایه زیرین عنوان می‌کند: تو باید این امکانات را در اختیار من قرار دهی تا بتوانم کارم را انجام دهم.




اکنون اگر در یک سیستم واقعی تعداد کلاس‌های سطح پایین افزایش پیدا کنند، نیازی نیست تا کلاس سطح بالا تغییری کند.
کلاس‌های سطح پایین تنها باید عملکردهای تعیین شده در اینترفیس را پیاده سازی کنند. و این برخلاف حالتی است که وابستگی‌ها معکوس نشده‌اند:



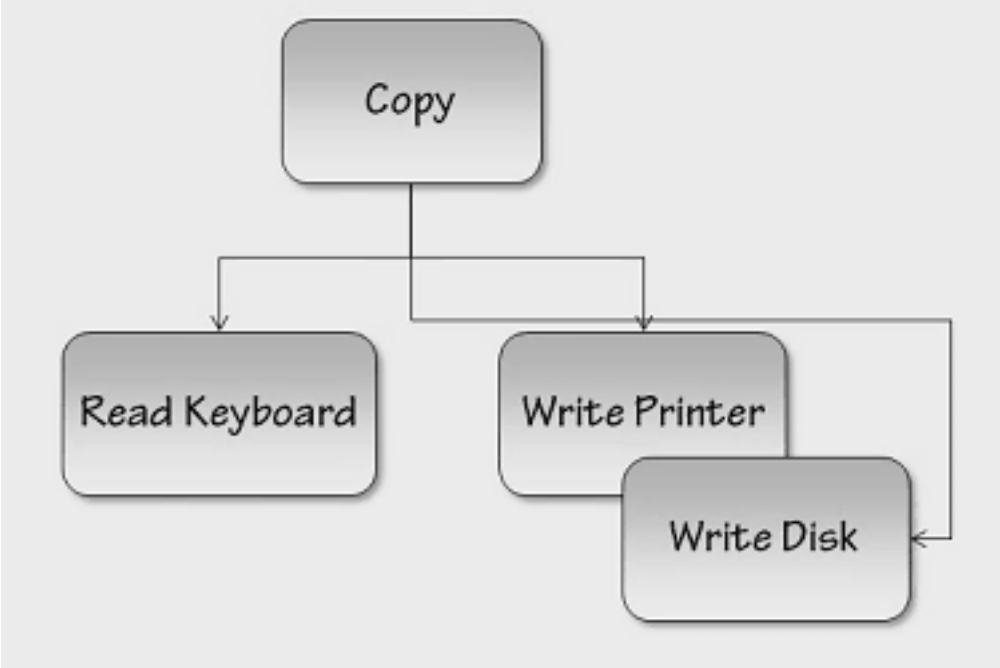

**تاریخچه اصل معکوس سازی وابستگی‌ها**

اصل معکوس سازی وابستگی‌ها در نشریه C++ Report سال 1996 توسط شخصی به نام Bob Martin (معروف به Uncle Bob!) برای اولین بار مطرح گردید. ایشان همچنین یکی از آغاز کنندگان گروهی بود که مباحث Agile را ارائه کردند. به علاوه ایشان برای اولین بار مباحث SOLID را در دنیای شیءگرایی معرفی کردند (همان مباحث معروف هر کلاس باید تک مسئولیتی باشد، باز باشد برای توسعه، بسته برای تغییر و امثال آن که ما در این سری مباحث قسمت D آن‌را در حالت بررسی هستیم).

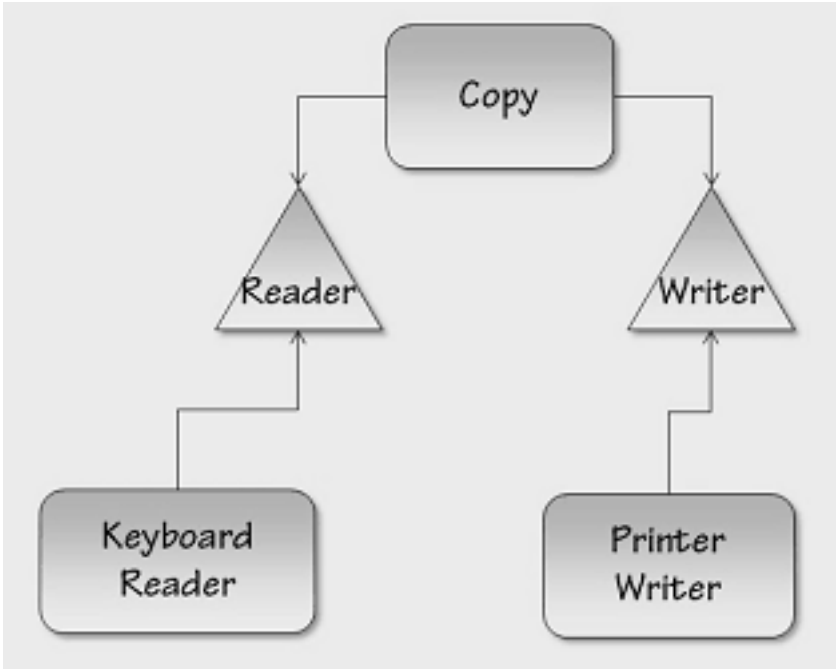
مطابق تعاریف Uncle Bob:
الف) ماژول‌های سطح بالا نباید به ماژول‌های سطح پایین وابسته باشند. هر دوی این‌ها باید به Abstraction وابسته باشند.
ب) Abstraction نباید وابسته به جزئیات باشد. جزئیات (پیاده سازی‌ها) باید وابسته به Abstraction باشند.

**مثال برنامه کپی**

اگر به مقاله Uncle Bob مراجعه کنید، یکی از مواردی را که عنوان کرده‌اند، یک برنامه کپی است که می‌تواند اطلاعات را از صفحه کلید دریافت و در یک چاپگر، چاپ کند.




حال اگر به این مجموعه، ذخیره سازی اطلاعات بر روی دیسک سخت را اضافه کنیم چطور؟ به این ترتیب سیستم با افزایش وابستگی‌ها، پیچیدگی و `if` و `else`های بیشتری را خواهد یافت؛ از این جهت که سطح بالایی سیستم به صورت مستقیم وابسته خواهد بود به ماژول‌های سطح پایین آن.
روشی را که ایشان برای حل این مشکل ارائه داده‌اند، معکوس کردن وابستگی‌ها است:

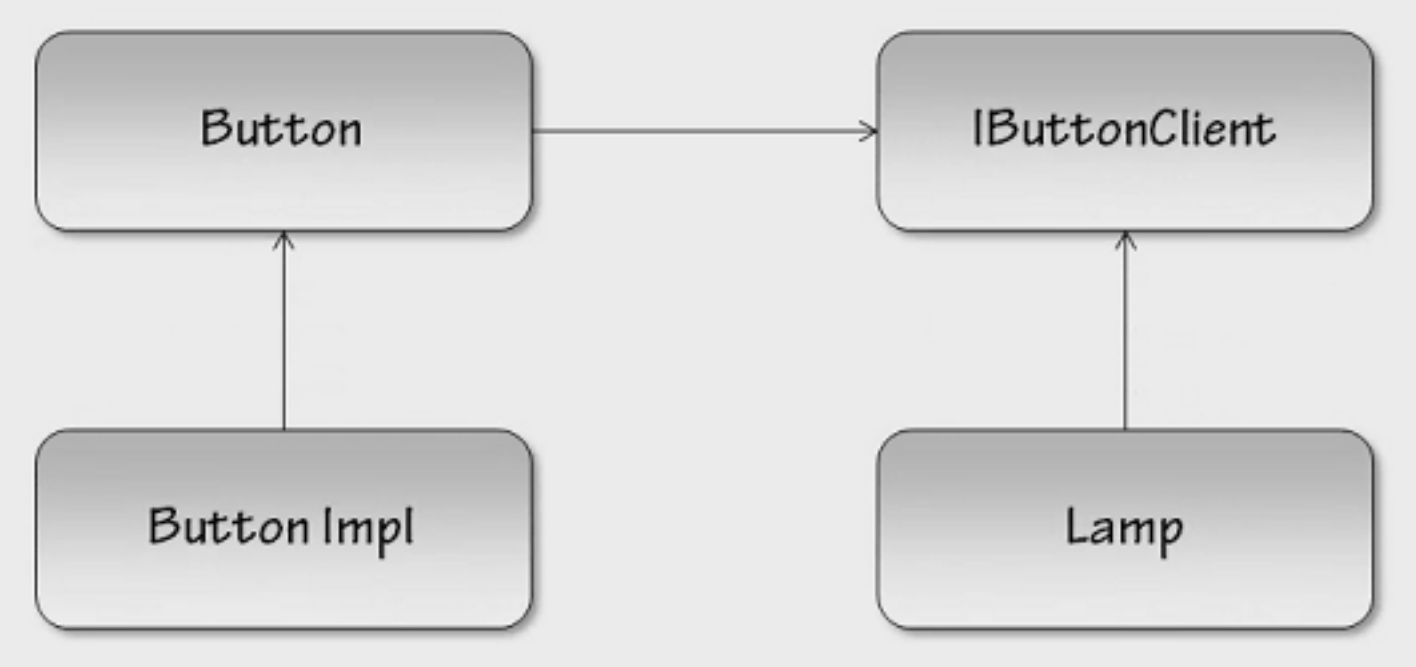



در اینجا سطح بالایی سیستم وابسته است به یک سری تعاریف `Abstract` خواندن و یا نوشتن؛ بجای وابستگی مستقیم به پیاده سازی‌های سطح پایین آن‌ها.
در این حالت اگر تعداد `Readers` و یا `Writers` افزایش یابند، باز هم سطح بالایی سیستم نیازی نیست تغییر کند زیرا وابسته است

به یک اینترفیس و نه پیاده سازی آن که محول شده است به لایه‌های زیرین سیستم.
این مساله بر روی لایه بندی سیستم نیز تاثیرگذار است. در روش متداول برنامه نویسی، لایه بالایی به صورت مستقیم متدهای لایه‌های زیرین را صدا زده و مورد استفاده قرار می‌دهد. به این ترتیب هر تغییری در لایه‌های مختلف، بر روی سایر لایه‌ها به شدت تاثیرگذار خواهد بود. اما در حالت معکوس سازی وابستگی‌ها، هر کدام از لایه‌های بالاتر، از طریق اینترفیس از لایه زیرین خود استفاده خواهد کرد. در این حالت هرگونه تغییری در لایه‌های زیرین برنامه تا زمانیکه اینترفیس تعریف شده را پیاده سازی کنند، اهمیتی نخواهد داشت.

**مثال برنامه دکمه و لامپ**

مثال دیگری که در مقاله Uncle Bob ارائه شده، مثال برنامه دکمه و لامپ است. در حالت متداول، یک دکمه داریم که وابسته است به لامپ. برای مثال وهله‌ای از لامپ به دکمه ارسال شده و سپس دکمه آن‌را کنترل خواهد کرد (خاموش یا روشن). مشکلی که در اینجا وجود دارد وابستگی دکمه به نوعی خاص از لامپ است و تعویض یا استفاده مجدد از آن به سادگی میسر نیست.
راه حلی که برای این مساله ارائه شده، ارائه یک اینترفیس بین دکمه و لامپ است که خاموش و روشن کردن در آن تعریف شده‌اند. اکنون هر لامپی (یا هر وسیله الکتریکی دیگری) که بتواند این متدها را ارائه دهد، در سیستم قابل استفاده خواهد بود.



## نظرات خوانندگان

نویسنده: سیروس
تاریخ: ۱۳۹۲/۰۱/۲۵ ۹:۴۹

مثل همیشه عالی بود، واقعا جامعه برنامه نویسان به این مطالب زیاد نیاز دارند.
خیلی اوقات فکر می‌کنیم همین که از اینترفیس استفاد کنیم کار تمومه؛ غافل از اینکه این اینترفیس‌ها پایین‌تر از اون خطه!

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۱/۲۵ ۱۳:۱۵

این بحث با کیفیت مطلوب و دقت نظر خاصی مطرح شده است که البته از آقای نصیری هم جز این انتظار نمی‌ره.
ابتدا شرایط موجود بیان و بررسی می‌شود و برای بهبود آن راهکاری را مفید می‌بینیم و می‌پذیریم به نام اصل وارونگی وابستگی یا همان معکوس‌سازی وابستگی که در این بخش به آن پرداخته شده است.
موارد ۲ و ۳ و ۴ که به آن اشاره شده است در حقیقت روند طبیعی محقق شدن این اصل است که در بخش‌های بعدی به آن‌ها پرداخته خواهد شد. توجه به این روند سبب می‌شود این مفاهیم به جای هم به کار برده نشوند.
پس از قبول اصل یاد شده باید آن‌را پیاده سازی نمود. الگوی وارونگی کنترل برای پیاده سازی آن تدوین می‌گردد. برای اجرای این الگو نیاز به روشی پیدا می‌کنیم که وابستگی را به طور کامل بیرون شیء وابسته نگه داریم. پس چاره ای جز تزریق آن در زمانی که لازم است نداریم. بسیار خوب نام آن‌را روش تزریق وابستگی می‌گذاریم. در نهایت تزریق‌هایی که ممکن است پی در پی لازم شود را گردن کس دیگری می‌اندازیم که همان برنامه‌جانبی یا کتابخانه یا فریم‌ورک‌هایی هستند که به آن‌ها می‌گوییم چه وقت چی تزریق کنند. (آن‌ها در بردارنده اطلاعاتی هستند که مثلاً شیء ۱ برای انجام کار شیء ۲ باید به آن داده شود)
حال اگر تازه با این مفاهیم آشنا شده اید توصیه می‌کنم یک بار دیگر این بخش را مطالعه کنید و منتظر روند پیاده سازی این اصل در بخش‌های بعد باشید.

تشکر از مهندس نصیری برای اجرای این دوره آموزشی.

نویسنده: مسعود2
تاریخ: ۱۳۹۲/۰۵/۲۰ ۱۲:۲۲

تعریف دقیق ماژولهای سطح بالا و سطح پایین چیست و چطوری میشه اونها رو در نرم افزار پیداکرد؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۲۰ ۱۲:۳۰

مراجعه کنید به مطلب « مراحل Refactoring یک قطعه کد برای اعمال تزریق وابستگی‌ها » جهت نحوه یافتن وابستگی‌ها و معکوس کردن آن‌ها.

نویسنده: مسعود2
تاریخ: ۱۳۹۲/۰۵/۲۱ ۱۴:۳۶

فرمودید:" پس چاره ای جز تزریق آن در زمانی که لازم است نداریم ". آیا نمیتوان به جای تزریق وابستگی از الگوهایی مثل Service Locator و Factory استفاده نمود؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۲۱ ۱۴:۵۵

اول یکبار دوره را کامل مطالعه کنید. در طی این سری مباحث به « بایدها و نبایدهای تزریق وابستگی‌ها » مفصل پرداخته شده.

نویسنده: Programmer

تاریخ: ۱۳۹۲/۰۶/۰۵ ۱۴:۴۷

با سلام
اینکه با مثال مفهوم رو توضیح دادید خیلی خوبه!
با توجه به انتزاعی بودن برنامه نویسی، ارائه یک مثال عینی کار رو خیلی راحت‌تر میکنه و خواننده راحت‌تر تصویر سازی میکنه و متوجه امر میشه.
بازم ممنون
راستی این وهله سازی یعنی چی؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۰۵ ۱۴:۵۵

وهله سازی معادل instantiation است. یک instance یا یک وهله.

نویسنده: وحید م
تاریخ: ۱۳۹۲/۰۷/۱۳ ۱۳:۵۷

با تشکر از مطالب مفیدتون .
ببخشید منظور از لایه بالاتر همان ui و پایین همان service است . ممنون از راهنمایی شما

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۷/۱۳ ۱۴:۳۰

یک مثالش می‌تونه لایه UI و لایه سرویس باشه.