**فصل دوم Clean Architecture - داستان دو ارزش!!!**

**1. مقدمه:**

هر سیستم نرم افزاری دو ارزش متفاوت **یعنی ساختار و عملکرد** را برای ذینفعان به ارمغان می‌آورد. وظیفه مهندسین نرم‌افزار این است که به شکلی کار کنند تا هر دو ارزش در بالاترین سطح خود باقی بمانند. متاسفانه در اغلب موارد مهندسین نرم‌افزار تمرکز خود را روی یکی از این ارزش‌ها می‌گذارند و از ارزش دیگر غافل می‌شوند. موضوعی که بیش از پیش ناراحت کننده است، این واقعیت است که معمولا روی مورد کم‌ارزش‌تر تمرکز می‌شود و این عدم تناسب بین سطوح این ارزش‌ها در نهایت منجر به بی‌ارزش شدن و عدم کارایی نرم‌افزار می‌شود.

**2. عملکرد:**

به طور یقین مهم‌ترین ارزش یک نرم‌افزار عملکردی است که ارائه می‌دهد. اگر دقیق‌تر به ماجرا نگاه کنیم، توسعه دهندگان و مهندسین نرم‌افزار استخدام می‌شوند تا سیستم‌هایی را تولید کنند که با عملکرد خود، موجب درآمدزایی یا کاهش‌ هزینه‌ها برای ذینفعان پروژه‌ شوند. به همین دلیل در اولین گام‌ها تیم توسعه به کمک ذینفعان می‌روند تا کمک کنند ویژگی‌ها و عملکرد‌های صحیحی را برای نرم‌افزار خود در نظر بگیرند. بعد از این مرحله است که توسعه دهندگان نرم‌افزاری را مطابق با این ویژگی‌ها و عملکرد‌های مورد انتظار طراحی کرده و توسعه می‌دهند.

اگر برنامه آنطوری که مورد انتظار است کار نکند، تیم توسعه شروع به دیباگ برنامه می‌کند تا خطای موجود در سیستم را پیدا نموده و آن را مطابق با نیازمندی مورد انتظار اصلاح می‌کند.

اغلب مهندسین نرم‌افزار به این مرحله از کار به عنوان کل کار خود نگاه می‌کنند. آن‌ها گمان می‌کنند اینکه کدی بنویسند که ویژگی‌های مورد انتظار را پیاده کند و در صورت وجود مشکل آن‌را دیباگ کنند کل کاری است که باید یک توسعه دهنده انجام دهد. اما متاسفانه اشتباه می‌کنند.

**3. معماری:**

دومین ویژگی‌ با ارزش یک سیستم نرم‌افزاری را باید در کلمه نرم‌افرار جستجو کنیم. در حقیقت این کلمه از ترکیب دو کلمه "نرم" و "افزار" یا "ابزار" تشکیل شده است. تکلیف قسمت دوم یعنی "افزار" که معلوم است و اشاره به یک محصول می‌کند. اما ارزش دوم در دل کلمه "نرم" نهفته است.

اگر در لغت‌نامه دهخدا یا عمید جستجو کنیم معانی "رقیق، سست، آرام، آهسته، لطیف" را برای کلمه "نرم" خواهیم یافت. خوب این لطافت و نرم بودن باید در همه جنبه‌های این ابزار وجود داشته باشد و مهم ترین جنبه‌ای که باید با مفهوم نرم بودن هم‌خوانی داشته باشد، قابلیت تغییر عملکرد سیستم به سادگی و لطافت و نرمی است. مسلما اگر قرار بود تغییرات در عملکرد سیستم به سختی اتفاق بیافتد به جای نام نرم‌افزار از نام سخت‌افزار استفاده می‌کردیم.

تغییرات در سیستم‌های نرم‌افزاری باید به سادگی قابل اعمال باشد. اگر به مرور زمان در اثر شناخت دقیق‌تر و یا تغییر شرایط کسب و کار ذینفعان پروژه ویژگی‌ها و شرایط سیستم را تغییر دادند، این تغییر عقیده باید بسیار ساده و آسان در سیستم قابل اعمال باشد. هنگامی که نیاز به تغییرات داریم، پیچیدگی انجام کار باید مربوط به محدوده و اندازه تغییرات باشد نه به شکل خود تغییرات.

این تفاوت بین شکل و اندازه تغییرات است که معمولا هزینه تولید نرم‌افزار را به مرور افزایش می‌دهد. به همین دلیل است که بعضا تغییرات بدون در نظر گرفتن اندازه آن‌ها هزینه تولید نرم‌افزار را افزایش می‌دهند. به همین دلیل است که هزینه تولید نرم‌افزار در اوایل عمر آن کمتر از سالهای بعدی آن است.

اگر از منظر ذینفعان پروژه به تغییرات و درخواست‌ها نگاه کنیم، کارها ساده است، آن‌ها یک حجم ثابت از کار و تغییرات را در طول سالیان مختلف ارسال کرده اند و توقع دارند که کار مثل قبل و در مدت زمان قبل انجام شود. اما از نظر توسعه‌دهنده بی‌نوا ما یک پازل بسیار بزرگ و پیچیده و در حال رشد داریم که هر بار قطعه‌ای جدید به آن اضافه می‌شود یا قطعه‌ای از آن تغییر می‌کند، سرهم کردن آن بسیار پیچیده‌تر از قبل می‌شود و یافتن محل مناسبی برای قطعه جدید سخت‌تر و پیچیده‌تر از قبل است.

شاید استفاده از کلمه "شکل" در این مطلب کمی دور از انتظار باشد، اما در نهایت نتیجه خوبی از این استفاده خواهیم گرفت. اگر دقت کنید اغلب توسعه دهندگان این تصور را دارند که کاری نامتعارف انجام می‌دهند. مثلا قرار است یک تیرآهن مستطیل شکل را در فضای یک میل‌گرد جای دهند.

مسئله اساسی معماری سیستم‌ها است. هنگامی که معماری‌ سیستم، شکل خاصی را بر دیگری ترجیح می‌دهد، تغییر در شکل سیستم و نیازمندی‌ها سخت‌تر می‌شود. در نتیجه معماری سیستم باید جدای از شکل انجام کار‌ها باشد.

**4. ارزش بیشتر:**

بودن یا نبودن مسئله این است. معماری یا عملکرد؟ کدام یک برای یک سیستم نرم‌افزاری مهم‌تر است؟ ارائه عملکرد درست برای یک سیستم نرم‌افزاری مهم‌تر است یا قابلیت تغییر ساده‌تر؟

قاعدتا اگر این سوال را از مدیران کسب و کار بپرسید بی درنگ به شما پاسخ می‌دهند مهم این است که سیستم باید به درستی کار کند. اما توسعه دهندگان معمولا با این نظریه مشکل دارند و مخالف این نظر هستند. بیایید با یک مثال دقیق‌تر این موضوع را بررسی کنیم.

اگر شما برنامه‌ای در اختیار من قراردهید که به بهترین و کاملترین شکل ممکن کار می‌کند اما غیرقابل تغییر است، شاید در کوتاه مدت مشکلی ایجاد نکند. اما در طولانی مدت و با تغییر نیازمندی‌های من، عملکرد سیستم غیرقابل استفاده شده و نرم افزار بلا استفاده می‌شود.

حال اگر شما برنامه‌ای به من بدهید که عملکرد صحیحی ندارد اما به سادگی قابل تغییر است، من می‌توانم به سادگی ایرادات سیستم را رفع کنم و آن‌را تبدیل به سیستمی کارا و کامل کنم و در ادامه‌ هم با تغییر نیازمندی‌ها به سادگی نرم‌افزار را تغییر می‌دهم.

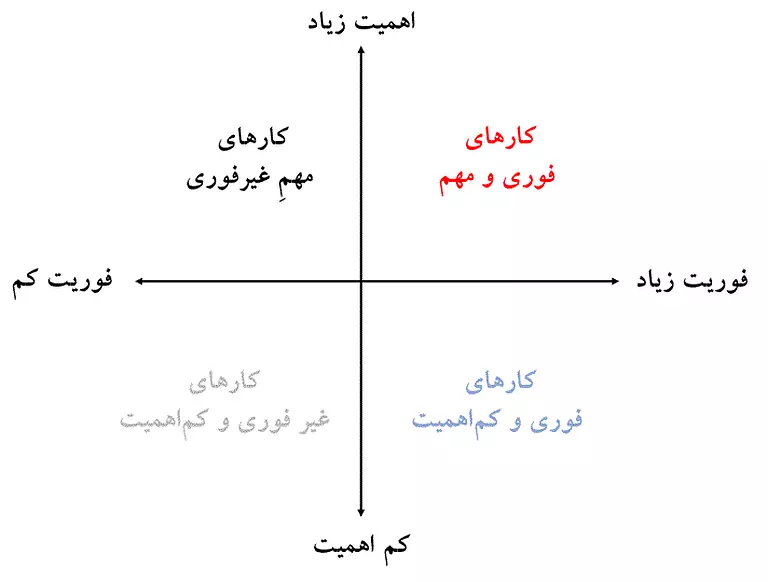
ممکن است با این دو مثال متقاعد نشده باشید و با این نظریه مخالف باشید، چون به هرحال اینکه سیستمی غیرقابل تغییر باشد به دور از واقعیت است، اما اگر کمی دقت کنید سیستم‌ها ممکن است قابل تغییر باشند ولی هزینه تغییرات اینقدر زیاد باشد که صرفه اقتصادی در پیاده سازی ویژگی‌جدید یا تغییر ویژگی‌های موجود وجود نداشته باشد.

اگر از مدیران کسب و کار بپرسید " آیا می‌خواهید بعدا در سیستم تغییر ایجاد کنید؟" به شما پاسخ خواهند داد: "بله قطعا تغییرات خواهیم داشت". اما اگر کمی با آن‌ها کلنجار بروید به شما خواهند گفت که عملکرد حال حاضر مهم‌تر از انعطاف سیستم برای پذیرش تغییرات آینده است. در همین شرایط اگر آن‌ها از شما بخواهند تغییری در سیستم اعمال کنید و شما تخمین زمان و هزینه زیادی برای سیستم داشته باشید آن‌ها عصبانی خواهند شد و به شما خواهند گفت که چرا اجازه دادید کار سیستم به جایی برسد که انجام یک تغییر ساده اینقدر هزینه‌بر باشد.

**5. ماتریس آیزنهاور:**

در سال ۱۹۵۴ (۱۳۳۳ هجری شمسی) رئیس‌جمهور سابق ایالات متحده آمریکا، جناب دوایت دی. آیزنهاور، در جریان یک سخنرانی بین‌المللی حرف جالبی زد. مضمون سخنان آیزنهاور این بود: «مشکلات من دو نوعند: فوری‌ها و مهم‌‌ها. فوری‌ها مهم نیستند، و مهم‌ها هم هیچگاه فوری نیستند.»

برای درک بهتر این صحبت ماتریسی مانند شکل زیر در نظر می‌گیریم:



با کمی دقت در این سخن قدیمی، به نکات ارزشمندی دست می‌یابیم. کارهایی که فوری هستند معمولا مهم نسیتند و کارهایی که مهم هستند به ندرت فوری هستند. قطعا در این بین کارهایی هم وجود دارند که شرایط متفاوتی دارند یعنی کارهایی که فوری و مهم هستند و کارهایی که غیرفوری و بی اهمیت هستند.

اگر بخواهیم این 4 حالت را در یک لیست مرتب قرار دهیم، این لیست به شرح زیر خواهد بود:

* مهم و فوری
* مهم و غیرفوری
* غیرمهم و فوری
* غیرمهم و غیر‌فوری

ارزش اولی که برای یک سیستم‌ نرم‌افزاری بیان کردیم یا همان عملکرد آن‌ها یک نیاز فوری است و ارزش دوم آن یعنی معماری و قابلیت تغییر آن بسیار مهم است. در لیست بالا کارهای مهم رتبه 1 و 2 را به خود اختصاص داده‌اند و کارهای فوری رتبه‌های 1 و 3.

یکی از بزرگترین ایراداتی که مدیران و ذینفعان کسب و کارها انجام می‌دهند این است که کارهایی که در رده سوم قراردارد یعنی غیرمهم و فوری را به مرتبه یک یعنی مهم و فوری ارتقا می‌دهند. این جابجایی باعث می‌شود کارهایی که مربوط به معماری در رتبه دوم می‌شود یعنی مهم و غیرفوری دائما به تاخیر بیافتد و کارهایی که اهمیت انجام کمی دارد دائم به پروژه اضافه بشود.

نداشتن مدیرانی که اهمیت معماری خوب را بدانند و حاضر باشند کارهای غیرمهم خود را فدای معماری خوب کنند چالشی بزرگ در راه توسعه دهندگان نرم‌افزار است. این دلیلی است که به عنوان مهندس نرم‌افزار استخدام می‌شویم، یعنی اثبات اینکه در شرایطی هزینه کردن برای داشتن معماری خوب با ارزش‌تر از ارائه چند ویژگی جدید و تغییر چند ویژگی قدیمی است.

**6. جنگ برای معماری:**

به انجام رساندن وظیفه یک مهندس نرم‌افزار یعنی اجبار سازمان به پذیرفتن هزینه کردن برای معماری خوب به جای ارائه ویژگی جدید یعنی شروع یک جنگ و کشمکش دائمی. تیم توسعه باید مثل یک پزشگ متعهد دارو‌هایی را برای بیمار خود که همان سازمان است تجویز کند و به سازمان اثبات کند که همیشه خودرن خوراکی‌های خوشمزه و پرچرب کار خوبی نیست و در مقابل سازمان مثل یک بچه لجوج اصرار به نخوردن دارو و استفاده از غذاهای همیشگی دارد.

یک تیم توسعه موفق به خوبی از پس این کشمکش برخواهد آمد. نکته ای که به عنوان یک مهندس نرم‌افزار باید همیشه در ذهن خود داشته باشیم این است که ما کارمند ذینفعان پروژه نیستیم، بلکه ما خودمان یکی از ذینفعان پروژه هستیم. ما نیز در پروژه‌ها سهیم هستیم و باید از ارزش سهام خودمان در پروژه‌ها دفاع کنیم. این بخشی از وظیفه و نقش ما به عنوان مهندس نرم‌افزار است.

در پایان به این نکته دقت داشته باشید که اگر ابتدا سیستم را توسعه دهید و بعد از مدتی به سراغ ایجاد یک معماری برای آن بروید، احتمالا توسعه و تغییر سیستم بسیار پرهزینه خواهد بود. اگر در سیستمی که به این شکل توسعه داده شده است کار می‌کنید به این معنا است که آنچنان که شایسته و لازم بوده است برای حفظ حقوق خود در پروژه مبارزه نکرده اید.

**فصل سوم Clean Architecture - مروری بر پارادایم‌ها**

**1. مقدمه:**

سه پارادایم برنامه نویسی که در این قسمت تعریف خواهیم کرد عبارتند از:

* برنامه نویسی شی گرا
* برنامه نویسی ساخت‌یافته
* برنامه نویسی تابعی

سه پارادایم برنامه نویسی

**2. برنامه نویسی ساخت‌یافته:**

اولین روش برنامه نویسی که مورد اقبال عمومی قرار گرفت، برنامه‌نویسی ساخت‌یافته بود که البته اولین روش ابداع شده نبود. این روش توسط دانشمند هلندی آقای ادسخر ویبه دِیکسترا در سال 1968 به جامعه توسعه دهندگان معرفی گردید. آقای دیکسترا نشان داد وجود نامحدود و بدون برنامه دستورات Jump یا همان gotoها در برنامه لطمه بسیار زیادی به ساختار برنامه‌ها وارد می‌کند. در فصل‌ بعدی جزئیات بیشتری در مورد برنامه‌نویسی ساخت‌یافته خواهیم دید اما در اینجا به بیان این نکته اکتفا می‌کنیم که بسیاری از دستورات کنترلی و شرطی و تکرار مانند if, then, else, do, while, until که این روزها بسیار پر استفاده هستند حاصل تلاش‌های این دانشمند فقید برای حذف goto و پیاده سازی ساختاری صحیح در برنامه نویسی است. اگر بخواهیم این روش برنامه نویسی را در یک جمله خلاصه کنیم، می‌توانیم بگوییم:

برنامه‌نویسی ساخت‌یافته نظم و انضباط را در انتقال مستقیم کنترل در برنامه‌ها به ارمغان می‌آورد.

**3. برنامه‌نویسی شی‌گرا:**

دومین روش برنامه‌نویسی که فراگیر شد، دست بر قضا 2 سال زودتر از روش قبل یعنی در سال 1966 معرفی شده بود. این روش برنامه‌نویسی توسط آقایان اوله-یوهان دال و کریستین نیگارد به جامعه برنامه نویسی معرفی گردید. این برنامه‌نویسان متوجه شدند function call stack frame در زبان ALGOL این قابلیت را دارد که به Heap منتقل شود. با توجه به این شرایط متغیر‌های داخل توابع این اجازه را دارند که بعد از پایان فراخوانی توابع نیز باقی بمانند. در این شرایط توابع تبدیل به سازنده‌های کلاس شدند. متغیر‌های داخل توابع به عنوان Instance variable در نظر گرفته شدند و توابعی که به صورت تو در تو در تابع سازنده تعریف می‌شدند نیز تبدیل به متد‌های کلاس‌ها شدند. با مرتب سازی و دسته بندی توابع به این شکل، رفته رفته مفهوم چند ریختی نیز شکل گرفت. در فصل پنجم به طور کامل این روش توسعه را بررسی خواهیم کرد اما اگر بخواهیم به طور خلاصه پارادایم شی‌گرا را در یک جمله تعریف کنیم به جمله زیر می‌رسیم:

برنامه نویسی شی گرا نظم و انظباط را در انتقال غیر مستقیم کنترل به ارمغان می‌آورد.

**4. برنامه‌نویسی تابعی:**

سومین روش برنامه‌نویسی که اخیر بسیار مورد توجه قرار گرفته است، اولین روشی بود که به دنیای برنامه‌نویسی معرفی شده است. در واقع ابداع این روش به قبل از دوران برنامه‌نویسی باز می‌گردد. برنامه‌نویسی تابعی حاصل مستقیم تلاش‌های آقای آلونزو چرچ است که در سال 1936 محاسبات لاندا را ارائه کرد. محاسبات لاندا پایه زبان برنامه نویسی LISP که در سال 1958 توسط آقای جان مک کارتی ابداع شد را بنا نهاد. مفهوم بنیادی در محاسبات لاندا تغییرناپذیری (Immutability) است. در این روش گفته می‌شود که مقدار یک نماد تغییر نخواهد کرد. با توجه به این مفهوم می‌توانیم بگوییم "تخصیص دادن" در این روش برنامه‌نویسی بی معنا است. البته اغلب زبان‌های برنامه‌نویسی تابعی دستوراتی را برای تغییر مقدار متغیر‌ها در شرایط خاص در اختیار توسعه دهندگان قرار می‌دهند.

برنامه نویسی تابعی نظم و انضباط را در تخصیص مقدار به ارمغان می‌آورد.

**5. زنگ تفکر:**

با کمی تفکر متوجه خواهید شد که هریک از این روش‌های برنامه‌نویسی توانایی‌هایی را از دسترس برنامه‌نویسان حذف کرده است. هیچ کدام از این روش‌های برنامه‌نویسی ویژگی جدیدی به داشته‌های توسعه دهندگان اضافه نکرده است. هرکدام از این روش‌ها قواعدی را به سیستم تحمیل کرده تا ویژگی‌هایی که مخالف اهداف آن‌ها است را از دسترس توسعه دهندگان خارج کند. این روش‌های برنامه نویسی بیشتر از اینکه به ما بگویند چه کارهایی انجام دهیم، میگویند چه کارهایی نباید انجام دهیم.

اگر به گونه ای دیگر نگاه کنیم، هر کدام از این روش‌های برنامه نویسی توانایی‌های ما را در توسعه کم کرده اند. این سه روش در مجموع goto، function pointer و assignment را از ما گرفته‌اند. با این اوصاف با پیشرفت علم و اضافه شدن روش‌های دیگر آیا چیز دیگری برای ادامه این راه باقی می‌ماند؟

به لطف خدا با بررسی شرایط متوجه می‌شویم که در طی دهه 1958 تا 1968 هر سه این روش‌ها به دنیای برنامه‌نویسی معرفی گردید و هرآنچه نیاز بود از برنامه نویسان گرفته شود، گرفته شد. اکنون با گذشت سال‌های بسیار زیاد از آن زمان روش جدیدی به دنیای برنامه نویسی معرفی نشده است.

**6. خلاصه:**

شاید با خود فکر کنید معرفی روش‌های برنامه نویسی چه ارتباطی به معماری نرم‌افزار دارد. باید بگویم بسیار مرتبط است. ما از چندریختی به عنوان مکانیزمی جهت عبور از مرز‌های معماری استفاده می‌کنیم. از قواعد برنامه‌نویسی تابعی جهت تعیین محل و نحوه دسترسی به داده‌ها استفاده می‌کنیم و از برنامه نویسی ساخت یافته برای طراحی و پیاده سازی الگوریتم‌های خود استفاده می‌کنیم.

دقت کنید که چطور این سه مفهوم و روش برنامه‌نویسی پاسخی به سه نگرانی بزرگ هنگام معماری می‌دهند: چگونگی عملکرد، جدا بودن اجزا و مدیریت داده‌ها.

در سه فصل آینده به شرح و بسط کاملتری از این سه روش برنامه‌نویسی خواهیم پرداخت.

**فصل چهارم Clean Architecture - برنامه‌نویسی ساخت‌یافته**

**2. اثبات:**

اولین مشکلی که توجه دیکسترا را به خود جلب کرد این بود که برنامه‌نویسی بسیار سخت بود و برنامه نویسان نمی‌توانستند خیلی خوب برنامه‌نویسی کنند. حتی برای نوشتن ساده‌ترین برنامه‌ها نیز توسعه دهنده‌ها نیاز داشتند به جزئیات خیلی زیادی توجه کنند و مطالب زیادی را به خاطر بسپارند که برای مغز انسان‌ها بیش از حد پیچیده بود و در راه انجام این کار هیچ راهنمایی وجود نداشت. فراموش کردن یک نکته جزئی باعث می‌شد برنامه‌ای که در ظاهر تصور می‌شد باید به خوبی کار کند، در عمل دچار مشکل باشد.

راهکار دیکسترا برای حل این مشکلات استفاده از روش اثبات ریاضیات بود. او می‌خواست یک سلسله مراتب اقلیدسی از قضایای فرضیه ها ، نتایج و لمها را ایجاد کند. او فکر می‌کرد برنامه نویسیان می‌توانند از این سلسله مراتب مانند ریاضی‌دانان استفاده کنند. در حقیقت او تصویر می‌کرد برنامه نویسان می‌توانند تعداد حقیقت اثبات شده از قبل را به کمک کدنویسی به هم متصل کنند و در نهایت نتیجه ای درست بگیرند.

دیکسترا متوجه شد برای انجام این کار باید بتواند راهکار‌هایی برای نوشتن اثبات‌های پایه‌ای برای الگوریتم‌های ساده ارائه کند. در این مرحله او متوجه چالشی بودن کاری که در پیش گرفته شد.

او در طول تحقیقاتش متوجه شد وجود برخی goto در برنامه‌ها باعث می‌شود که نتوان برنامه‌ها را به ماژول‌های کوچک شکاند. در نتیجه اگر شما نتوانید برنامه‌ها را به واحد‌های کاری کوچکی بشکانید، امکان استفاده از روش تقسیم و غلبه نیز وجود ندارد. این اولین مشکل بود. برای داشتن اثبات‌های‌ پایه برای الگوریتم‌های ساده باید بتوان از روش تقسیم و غلبه استفاده کرد.

در طرف مقابل دستورات goto قرار داشتند که این مشکل را ایجاد نمی‌کردند. با بررسی دقیق‌تری روی دستورات goto که ساختار برنامه را دچار مشکل نمی‌کردند او متوجه شد این دستورات شامل دستوراتی برای انتخاب و تکرار می‌باشند. این دستورات پایه ای برای دستورات امروز مثل if/then/else و do/while بودند. بخش‌هایی از برنامه که فقط از این دسته دستورات goto استفاده می‌کردند به سادگی قابل شکستن به ماژول‌های کوچک‌تری بودند.

دیکسترا می‌دانست این دستورات کنترلی هنگامی که در ساختار اجرای متوالی قرار بگیرند، کارایی ویژه‌ای پیدا می‌کنند. البته این مورد 2 سال قبل توسط بوهم و ژاکوپینی به اثبات رسیده بود. آن‌ها اثبات کرده بودند که هر برنامه‌ای می‌تواند از ترکیب سه ساختار ایجاد شود که این سه ساختار عبارتند از :

* توالی
* انتخاب
* تکرار

این دستاوردی مهم بود: ساختاری‌های بسیار کنترلی که باعث ایجاد ماژول‌های کوچک قابل اثبات می‌شد می‌توانستند ساختار هر برنامه‌ای را ایجاد کنند. از اینجا بود که برنامه نویسی ساخت‌یافته متولد شد.

دیکسترا نشان داد که دستورات متوالی با یک بار بررسی پشت‌سرم هم قابل تایید و اثبات هستند. در این روش برای بازبینی تعدادی ورودی دریافت شده و بر اساس ترتیبی خاص پردازش و بازبینی شده و خروجی‌‌هایی تولید می‌شود. این کار تفاوت چندانی با اثبات‌های معمول ریاضی ندارد.

در این روش هنگامی که با یک ساختار انتخابی سر و کار داشته باشیم، چندین بار مراحل را طی خواهیم کرد و هر بار یکی از مسیر‌های قابل انتخاب را دنبال خواهیم کرد. اگر در پایان هر بار مسیریابی و ادامه راه به نتیجه درستی برسیم کار به پایان می‌رسد.

ساختار‌های تکراری اما کمی متفاوت است. برای اثبات یک تکرار صحیح باید از استقرا استفاده کنیم. او ابتدا برای ساختار‌های تکرار حال 1 را بررسی می‌کردو سپس حالت N و N +1 را بررسی و اثبات می‌کرد در نهایت شرط‌های ابتدایی و انتهایی را نیز اثبات می‌کرد.

هر چند این روش اثبات بسیار پرزحمت و زمانگیر بود، اما به هر حال نتیجه‌ای حاصل می‌شد و این دستاوردی بسیار بزرگ بود.

**3. یک اعلامیه دردناک:**

در سال 1968 دیکسترا نامه‌ای را برای سردبیر CACM که در مارچ منتشر شده بود ارسال کرد و عنوان این نامه این بود " استفاده از عبارات goto مضر است" و در این مقاله او نظریه ساختار‌های سه‌گانه خود را شرح داد.

در این زمان در جامعه برنامه نویسی آشوبی به پا شد. در آن زمان هنوز اینترنت و شبکه‌های اجتماعی وجود نداشت و جامعه برنامه نویسی نتوانست آنطور که باید و شاید از خجالت اقای دیسکترا در بیاید. ( امکان پیوند اقوام و بستگان ایشان و ابلاغ سلام خدمت برخی دیگر از بستگان ایشان توسط جامعه برنامه نویسان میسر نشد). البته آن‌ها نامه‌ها و مطالبی را برای بسیاری از مجلات آن زمان ارسال کردند که لزوما خیلی مودبانه نیز نبودند. در این بین البته با اینکه مطالب منفی زیادی ارسال شد، اما برخی نیز به شدت از این نظریه استقبال کرده و آن را مثبت ارزیابی کردند. اینجا بود که جنگی برای نزدیک به 10 سال بین این دو گروه به راه افتاد.

در نهایت دیکسترا این جنگ را با پیروزی به پایان رسانید و مجادله‌ها به پایان رسید. زمانی‌که زبان‌های برنامه‌نویسی جدید پا به جهان گذاشتند اهمیت کمتری برای دستور goto قائل شدند و کم کم این دستور از چرخه برنامه‌نویسی حذف شد. اغلب زبان‌های برنامه‌نویسی مدرن یا دستور goto ندارند یا به صورت بسیار محدود قابل استفاده است و اگر توسعه دهنده‌ای این دستور را در کد توسعه دهنده دیگری مشاهده کند قطعا اولین چیزی به ذهنش می‌رسد ناتوانی ذهنی نویسنده است و بعد به بررسی دقیق شرایط خواهد پرداخت.

این روز‌ها همه ما برنامه‌های ساخت یافته می‌نویسیم. نه به خاطر اینکه به این کار علاقه داریم و این روش را انتخاب کرده‌ایم بلکه به خاطر اینکه اصلا زبان‌های برنامه‌نویسی حالت دیگری را در اختیار ما قرار نداده‌اند و نمیتوانیم کنترل برنامه را بدون در نظر گرفتن شرایط از هر جایی به جای دیگر منتقل کنیم.

البته برخی توسعه دهندگان دستورات break و exceptionها را معادل با توجه به شرایطی که دارند و به هم ریختگی‌که در جریان اجرای ترتیبی برنامه ایجاد می‌کنند معادل goto میدانند که البته خیلی هم دور از واقعیت نیست اما این دستورات غالبا تشکیل دهنده ساختار کلی برنامه‌ نبوده و به صورت پیش‌فرض نیز مثل گذشته به عنوان روالی عادی برای برنامه در نظر گرفته نمی‌شوند.

**4. تجزیه به عملکرد‌ها:**

برنامه نویسی ساختیافته این امکان را فراهم می‌کند که دائما برنامه را به ماژول‌های عملیاتی کوچکتری تقسیم کنیم. و این به این معناست که در نهایت برنامه ‌می‌تواند به توابع کوچکی تقسیم شود. در نتیجه شما می‌توانید یک مسئله بسیار بزرگ را دریافت کنید و برای حل آن تعدادی تابع مفهومی در حوزه مسئله استفاده کنید و آن توابع نیز در دل خود از تعدادی تابع کوچک‌تر استفاده کنند و این کار تا به پایان رسیدن حل مسئله ادامه پیدا کند.

بر پایه این روش توسعه، تحلیل و تجزیه ساخت یافته نیز در دهه 1970 رواج بسیاری پیدا کرد. افرادی مانند اد یوردن، لری کنستانتین، تام دی‌مارکو و میلر پیجونز از پیشگامان و معرفی کنندگان این روش بودند. با این روش‌ها برنامه نویسان می‌توانستند مسئله‌های بسیار بزرگ را به واحد‌های کوچکی شکسته و در نهایت تعداد زیادی ماژول‌های کوچک قابل بررسی و اثبات بسازند.

**5. اثبات بی اثبات:**

هر چند این روش بسیار مورد قبول جامعه برنامه نویسی قرار گرفت، اما هیچگاه آن دستاورد اولیه مورد نظر دیکسترا برای ساخت تعدادی نظریه و اثبات رسمی که برای هر کاری قابل قبول باشد، محقق نشد. هیچ گاه راهکار‌های ثابتی که که همیشه قابل اثبات باشند به جامعه ارائه نگردید. در نهایت رویای دیکسترا برای انجام این کار کمرنگ و کمرنگ‌تر شد و در نهایت از بین رفت. استفاده از روش‌های رسمی این روز‌ها طرفدارانی دارد و برخی هنوز معتقدند که استفاده از اثبات‌های رسمی راهکار دستیابی به نرم‌افزار‌های با کیفیت است.

**6. علوم پایه نجات‌بخش:**

علوم پایه بعضا تفاوت‌های اساسی با ریاضیات دارند، برای مثال من نمی‌توانم به شما قانون دوم نیوتون را اثبات کنم. بلکه من تنها می‌توانم قانون را به شما بگویم و سپس با آزمایش و اندازه گیری نتایج شما را متقاعد کنم که قانونی که بیان شده است درست است. هرچقدر هم قانون ارائه شده صحیح باشد راهکار ریاضی که آن را اثبات کند وجود ندارد. همه چیز وابسته به آزمایش و اندازه گیری است. اصلا مهم نیست که یک قانون چند بار امتحان می‌شود و چند سال مورد قبول همکان است، هر لحظه ممکن است یک آزمایش انجام شود و متوجه شویم قانونی که تا امروز تصور می‌کردیم صحیح است دیگر کاربردی ندارد. طبیعت قوانی علوم پایه این است که آن‌ها قابل رد کردن هستند اما قابل اثبات نیستند.

با این حال هنوز که هنوزه ما زندگی خود را بر مبنای این قوانین بنا می‌کنیم. هر بار که سوار ماشین می‌شود روی زندگی خود و اثبا F=m.a شرط بندی می‌کنید.

علوم با اثبات صحت یک مطلب کار نمی‌کنند بلکه با رد صحت آن‌ها عمل می‌کنند. هر قاعده‌ای را که با چندین و چندبار آزمایش قابل رد کردن نباشد را می‌پذیریم تا روزی که دانش ما به حدی برسد که بتوانیم آن قانون را نقض کنیم.

البته بعضی قواعد هم نه قابل تایید است و نه رد. برای مثال جمله " این یک دروغ است" نه صحیح است و نه غلط. این یک مثال ساده از مسائل غیر قابل اثبات است.

در نهایت اینگونه جمع بندی می‌کنیم که با ریاضیات عبارات قابل اثبات صحیح را در اختیار ما قرار میدهد و به کمک علوم پایه عبارات ناصحیح را شناسایی می‌کنیم.

**7. تست‌ها:**

دیکسترا می‌گوید:"تست ها وجود باگ‌ها را اثبات می‌کند نه عدم وجود آن‌ها را". به بیان دیگر به کمک تست‌ها می‌توانیم ثابت کنیم که یک برنامه به درستی کار نمی‌کند ولی اگر همه تست‌ها هم به درستی پاس شود نمی‌توانیم به قطعیت بگوییم که برنامه به درستی کار می‌کند. نهایتا بعد از تعداد قابل قبولی تست می‌توانیم بگوییم برنامه آنقدر قابل اعتماد است که برای مدتی تا اثبات اشتباه کار کردن آن مورد استفاده قرار بگیرد.

با پذیرش این جمله میپذیریم که هرچند توسعه نرم‌افزار در ظاهر بسیار وابسته و برگرفته از ریاضیات است اما در نهایت از جمله علوم پایه‌ای است.

برنامه‌نویسی ساخت یافته ما را به سمت ساخت ماژول‌های کوچک و قابل اثبات سوق می‌دهند. سپس از تست ‌ها برای اثبات عدم صحت عملکرد این ماژول‌ها استفاده می‌کنیم و اگر هیچ تستی موفق به رد آن ماژول نشد می‌گوییم این ماژول به اندازه کفایت قابل اعتماد است و می‌توانیم از آن استفاده کنیم.

**8. نتیجه‌گیری:**

در هر سطحی که بخواهیم تصور کنیم، از کوچترین توابع تا بزرگترین ماژول‌های نرم‌افزاری، دنیای نرم‌افزار برگرفته از علوم پایه است. کار ما به عنوان معمار و توسعه دهنده نرم‌افزار این است که توابع و ماژول‌ها و سرویس‌هایی را ایجاد کنیم که به سادگی قابل تست باشد و به کمک این تست‌ها قابلیت اطمینان سازه‌ها قابل اندازه گیری باشد. به همین دلیل است که برنامه نویسی و تفکر ساخت‌یافته از کوچکترین توابع تا بزگترین بخش‌‌های نرم افزار باید جاری باشد.

**پی‌نوشت:** وقتی به عملکرد افرادی مثل دیکسترا فکر می‌کنم که از چه دنیای اولیه‌ای چه دانش و دستاورد‌هایی ایجاد کردن و به سادگی و صادقانه در اختیاردنیا قرار دادن و دنیایی رو از تاریکی نجات دادن، از گفتن واژه "میدانم" در مورد خودم خجالت می‌کشم.

**فصل پنجم Clean Architecture - برنامه‌نویسی شی‌گرا**

در آینده نه چندان دور خواهیم دید که پایه و اصل یک معماری خوب فهم درست و استفاده صحیح از اصول برنامه‌نویسی شی‌گرا است. اما OO چیست؟

شاید ساده‌ترین پاسخ به این سوال این باشد "ترکیب عملکرد و داده‌ها با هم". هر چند این تعریف به دلایل زیادی نمی‌تواند صحیح باشد و دلالت بر این دارد تکه کد‌های زیر با هم متفاوت است.

این یک تعریف ناقص است، برنامه نویسان بسیار قبل‌تر از سال 1966 داده‌ها را به توابع ارسال می‌کردند، یعنی پیش از اینکه دال و نیگارد function call stack frame را به heap انتقال دهند و شی‌گرایی را ابدا کنند.

یکی دیگر از جواب‌هایی که به این پرسش داده می‌شود این است "روشی برای مدل‌سازی دنیای واقعی". قطعا یکی از بهترین راه‌های فرار برای پاسخ به این سوال همین است. اما واقعا مدل دنیای حقیقی چه معنایی دارد؟ اصلا چرا باید تمایل داشته باشیم همچین کاری انجام دهیم؟ شاید منظور اصلی از این جمله این باشد که با توجه به شباهت برنامه‌نویسی به این روش با دنیای واقعی درک آن ساده‌تر است. اما با این حال باز هم این تعریف بسیار ساده‌ انگارانه و به دور از واقعیت است و هیچ کمکی به درک بهتر برنامه‌نویسی شی‌گرا نمی‌کند.

برخی دیگر برای تعریف این روش برنامه‌نویسی دست به دامان سه کلمه جادویی در این حوزه می‌شوند. کپسوله‌سازی، چند ریختی و ارث بری. پیامد این تعریف هم این است که برنامه‌نویسی شی‌گرا یعنی ترکیبی از این سه کلمه کلیدی یا اینکه زبان برنامه‌نویسی شی‌گرا باید این سه ویژگی را داشته باشد. بیایید این سه کلمه جادویی را دقیق‌تر بشناسیم.

**2. کپسوله‌سازی:**

دلیل اینکه کپسوله‌سازی به عنوان یکی از اصول شی‌گرایی شناخته می‌شود این است که زبان‌های شی‌گرا روشی ساده و کارآمد جهت کپسوله‌سازی داده‌ها و عملکرد‌ها ایجاد می‌کنند. در نتیجه می‌توان مرزی برای داده‌ها و عمکرد‌های مرتبط با هم تعیین کرد. بیرون این مرز هیچ اثری از داده‌ها دیده نمی‌شود و تنها می‌توان از عملکرد‌ها عمومی مطلع شد و از آن‌ها استفاده کرد. این عملکرد را احتمالا در تعریف اعضای داده‌ای خصوصی و توابع عمومی در کلاس‌ها مشاهده کرده‌اید.

اگر زبان‌های شی‌گرا کپسوله سازی بهتری از پیشینیان خود در اختیار ما قرار نمی‌دهند، قطعا ارث بری را در اختیار ما می‌گذارند.

با کمی دقت در می‌یابیم ارث بری نحوه جدیدی از گروه‌بندی داده‌ها و توابع در محدوده‌ای قابل دسترس است. این کاری است که برنامه نویسان C بسیار قبل از تولد زبان‌های شی گرا توانایی انجام آن را داشتند. تکه کد زیر را در نظر بگیرید:

کمی منصفانه اگر به قضیه نگاه کنیم می‌توانیم بگوییم هرچند روش‌هایی برای پیاد‌ه‌سازی ارث بری قبل از معرفی شی‌گرایی وجود داشته، اما ظهور و بروز شی‌گرایی و زبان‌های شی‌گرا موجب تسهیل پیاده سازی این کار شده است.

صرفا جهت یادآوری: تا اینجا فهمیدیم که کپسوله سازی دستاودری برای شی‌گرایی نیست و ارث‌بری هم خیلی ابداع و اختراع جدیدی محسوب نمی‌شود، صرفا کمی ساده سازی شرایط اتفاق افتاده است. اما هنوز یک کلمه جادویی دیگر در کیسه خود داریم که باید به سراغ آن برویم:

**4. چند ریختی:**

آیا ما قبل از شی‌گرایی عملکردی شبیه به چندریختی در اختیار داشتیم؟ جواب این سوال به طور قطع بله است. تکه کد C زیر برای پیاده سازی copy را مشاهده کنید.

کلام آخر اینکه چندریختی استفاده‌ای از اشاره‌گر‌ها به توابع هستند. برنامه نویسان از سال‌‌ها قبل از معرفی شی‌گرایی و از این تکنیک و اشاره‌گر‌های به توابع برای به دست آوردن چندریختی استفاده می‌کردند. دقیق‌تر اگر بخواهیم صحبت کنیم، در این زمینه هم شی‌گرایی حرف جدیدی برای گفتن ندارد.

در حقیقت زبان‌های شی‌گرا امکان جدیدی در اختیار ما قرار نداند، بلکه یک ویژگی و روش قدیمی را به روشی امن و آسان در اختیار ما قرار دادند. همه ما میدانیم استفاده از اشاره‌گرها به توابع چقدر می‌تواند خطرناک باشد. مدیریت این اشاره‌گرها به یاد داشتن همه شرایط کاری بسیار دشوار و خطرناک است. اگر به هر دلیلی خطایی اتفاق بیوفتد رهگیری و رفع خطا بسیار سخت است.

زبان‌های شی گرا کاری که انجام دادند حذف فرایند‌های دستی مدیریت چند ریختی و توکار کردن فرایند ایجاد و مدیریت این اشاره‌گر‌ها بود که در نتیجه خطرات استفاده از آن‌ها نیز از بین رفت. سادگی و امنیتی که برای برنامه نویسان تا قبل از شی‌گرایی آرزویی دست نیافتنی بود.

بر این مبنا ما می‌توانیم نتیجه گیری کنیم که OO در جهت معکوس‌سازی کنترل جریان برنامه قواعدی را ایجاد کرده است.

**4.1. قدرت چند ریختی:**

واقعا چه چیز فوق‌العاده‌ای در مورد چند ریختی وجود دارد؟ برای درک بهتر جذابیت‌ها، بیایید مجددا به سراغ مثال کپی برویم. اگر دستگاه ورودی و خروجی جدیدی ابداع شود چه اتفاقی برای آن تکه کد می‌افتد؟ فرض کنید که می‌خواهیم داده‌هایی را از ورودی که توانایی تحلیل دست‌خط انسان را دارد دریافت کنیم و برای خروجی که توانایی خواندن مطالب را دارد ارسال کنیم. چه تغییری باید در پیاده سازی تابع copy ایجاد کنیم تا با این ورودی و خروجی‌های جدید کار کند؟

ما نیاز نداریم برنامه خود را تغییر دهیم. دقیقا نکته کار همینجاست. حتی نیاز به کامپایل مجدد برنامه‌خود نیز نداریم. اما چرا؟ چون در برنامه Copy هیچ وابستگی به نحوه پیاده سازی ورودی و خروجی وجود ندارد. تا زمانی که دستگاه‌های ورود و خروجی ما آن پنج تابع مورد نظر برای File را دارا باشند برنامه copy ما می‌تواند از آن‌ها استفاده کند.

در حقیقت دستگاه‌های ورودی و خروجی به عنوان پلاگین‌هایی به برنامه copy متصل می‌شوند.

اما چرا سیستم‌های UNIX تمامی ورودی و خروجی‌ها را به عنوان پلاگین در نظر می‌گیرند؟ به خاطر اینکه از اوخر دهه 1950 یادگرفتیم که برنامه‌های ما نباید به دستگاه‌های ورودی و خروجی وابسته باشند. به خاطر اینکه برنامه‌های زیادی وابسته به ورودی و خروجی نوشته شد تا در نهایت فهمیدیم که نیاز داریم یک برنامه باید بتواند با ورودی و خروجی‌های متفاوتی کار کند.

در اوایل ورودی ها کارت‌های پانچ شده بودند و برای انتشار خروجی نیز کارت‌های جدیدی پانچ می‌شد. ناگهان استفاده از کارت‌ها متوقف شد و استفاده کنندگان شروع به استفاده از نوار‌ها کردند. این کار بسیار سختی بود، چون باید بخش زیادی از برنامه‌ها باز نویسی می‌شد و از همان زمان یاد گرفتیم که برنامه‌ها باید از دستگاه‌های ورودی و خروجی جدا و بی اطلاع باشند.

معماری برنامه‌ها به این روش و با استفاده از پلاگین‌ها در اغلب سیستم‌های عامل برای دستگاه‌های ورودی و خروجی مورد استفاده قرار گرفت. با این حال برنامه نویسان، با توجه به نیاز به استفاده از اشاره‌گر به توابع برای پیاده سازی این معماری تمایل چندانی به استفاده از این روش در برنامه‌های عادی خود نداشتند.

شی‌گرایی امکان پیاده سازی معماری مبتنی بر پلاگین‌ها را در هر برنامه‌ای و هر چیزی فراهم کرد.

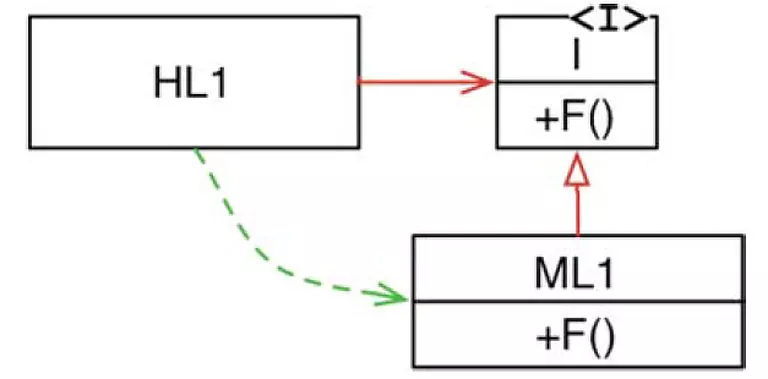
**4.2. معکوس‌سازی وابستگی:**

بیایید با هم تصور کنیم که نرم‌افزارها پیش از اینکه روش ساده‌ای برای پیاده‌سازی چندریختی ایجاد شود به چه شکلی بودند. در یک برنامه عادی تابع main یک تابع سطح بالا را صدا می‌زد،سپس تابع سطح بالا یک تابع سطح میانی‌را صدا می‌زد و در ادامه آن تابع سطح میانی یک تابع سطح پایین را صدا می‌زد. این جریان را جریان کنترل برنامه می‌نامیم. با مشاهده تصویر زیر درمیابیم که جریان کنترل برنامه و جریان وابستگی سورس کد برنامه مشابه یکدیگر بوده اند.

مقایسه جریان کنترل برنامه و وابستگی سورس کد

در تابع main برای استفاده از یک ویژگی، ماژول مورد نیاز پیاده‌سازی کننده آن ویژگی باید اسم برده می‌شود. این کار در C به کمک include انجام می‌شود. در جاوا import و در سی‌شارپ using این وظیفه را انجام می‌دهند. به همین ترتیب هر ماژولی نیاز به استفاده از ماژول دیگر داشته باشد باید آن را نام ببرد.

در این روش پیاده سازی جریان وابستگی توسط جریان کنترل برنامه به برنامه تحمیل می‌شود. اما با معرفی چند ریختی، راهکار کاملا متفاوتی را می‌توان در نظر گرفت.



همانطور که در تصویر بالا مشاهده می‌کنید، ماژول HL1 تابع F در ماژول ML1 را استفاده می‌کند و این کار را با واسطه اینترفیس **I** انجام می‌دهد. هنگام اجرای برنامه خبری از **I** نیست و HL1 مستقیما به سراغ ML1 و تابع F می‌رود. حالا این ML1 است که به **I** وابسته شده است. جریان کنترل برنامه با جریان وابستگی متفاوت شده است. به این روش اصطلاحا معکوس سازی وابستگی یا dependency inversion گفته می‌شود که تاثیرات عمیقی بر طراحی و پیاده‌سازی نرم‌افزار دارد.

در حقیقت زبان‌های شی‌گرا روشی امن و ساده برای پیاده‌سازی چند ریختی ایجاد کردند که باعث شد معکوس‌سازی وابستگی در هرجایی و هر‌شرایطی به سادگی قابل پیاده باشد.

مجددا به تصور اول مربوط به جریان کنترل‌برنامه و جریان وابستگی بازگردید و ببیند در کدام قسمت‌ها با استفاده از این تکنیک جریان کنترل برنامه و وابستگی امکان معکوس‌سازی دارند.

با استفاده از این روش، معماران نرم‌افزار کنترل کاملی بر رو جهت وابستگی سورس‌کد در سیستم دارند. دیگر اجباری برای یکسان در نظر گرفتن جهت کنترل و جهت وابستگی وجود ندارد. اصلا مهم نیست که هنگام اجرا کدام ماژول قرار است از امکانات کدام ماژول استفاده کند. جهت وابستگی سورس کد کاملا به تصمیم معمار بستگی دارد.

این یعنی قدرت. این قدرت اصلی است که OO برای ما به ارمغان آورد. حداقل از منظر یک معمار نرم‌افزار این بهترین هدیه OO است.

حال با این قدرت چه کارهایی قابل انجام است؟ حال با این قدرتی که در اختیار داریم می‌توانیم جریان وابستگی را به گونه‌ای تغییر دهیم که UI و Database به Business وابسته باشند به جای وابستگی Business به UI و Database.

با این روش دیتابیس و UI پلاگین‌هایی برای Business محسوب می‌شوند و بدون تغییر در Business توانایی عوض کردن این پلاگین‌ها را داریم.

با این روش UI و Database می‌توانند در ماژول‌های کاملا جدا توسعه داده شده و کامپایل شوند و در اختیار Business قرار بگیرند. مثلا در جاوا jar فایل‌ها و در سی شارپ dllها این کار را برای ما انجام می‌دهند. به طور خلاصه اگر سورس کد یکی از پلاگین‌ها تغییر کند، تنها نیاز به کامپایل و انتشار همان پلاگنی است و Business متوجه این تغییرات نمی‌شود.

**5. جمع بندی:**

شی‌گرایی چیست؟ جواب‌ها و نظریه‌های متفاوتی برای این سوال وجود دارد. اما از نگاه یک معمار نرم‌افزار پاسخ این است: OO توانایی استفاده از چندریختی برای کنترل کامل بر جریان کنترل و وابستگی در نرم افزار است. OO به معمار امکان می‌دهند سیستم را بر پایه پلاگین‌ها طراحی و پیاده سازی کند. در این روش ماژول با عملکرد سطح بالا و پیاده سازی Business دیگر وابستگی به ماژول‌های سطح پایین ندارد. ماژول‌های سطح پایین پلاگین‌هایی هستند که با توجه به نیاز ماژول سطح بالا عملکرد‌هایی را پیاده سازی می‌کنند و به صورت کاملا مجزا قابلیت توسعه و انتشار دارند.