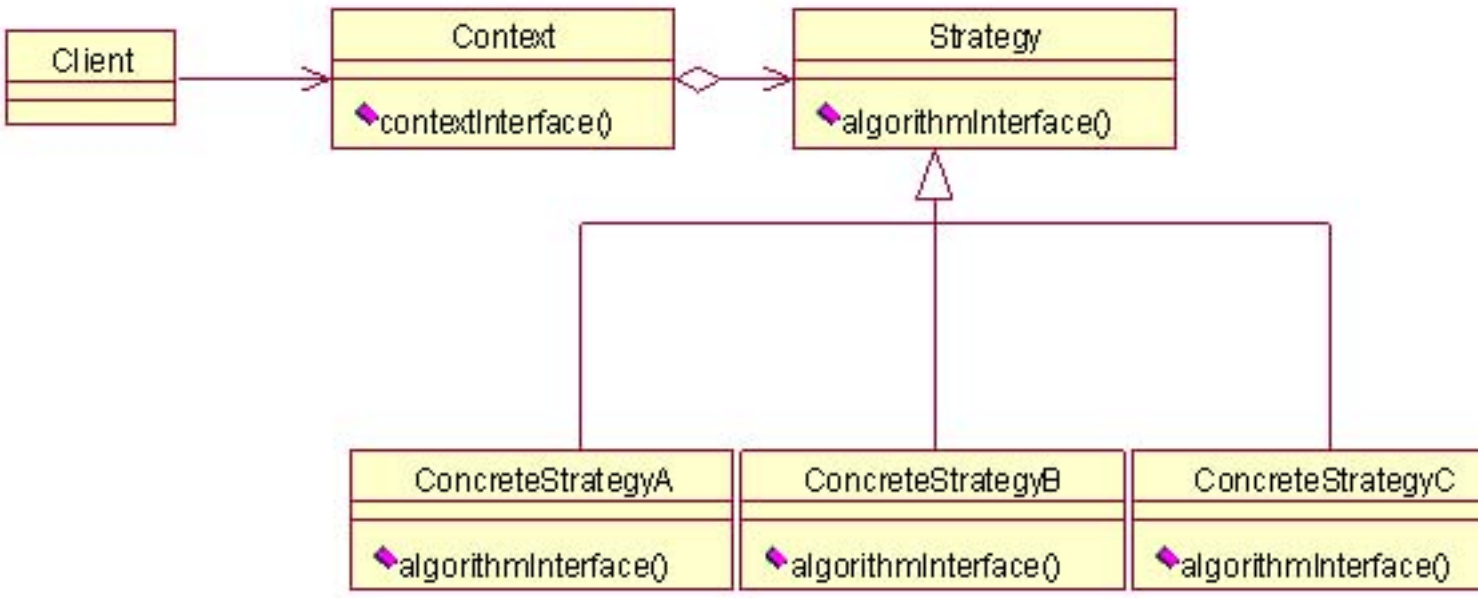
عنوان: **الگوی استراتژی - Strategy Pattern**
نویسنده: مجتبی شاطری
تاریخ: ۱۳۹۲/۰۶/۲۰ ۰:۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, Dependency Injection, SOLID Principals, Inheritance Strategies, SOLID, Dependency Inversion

الگوی استراتژی (Strategy) اجازه می-دهد که یک الگوریتم در یک کلاس بسته بندی شود و در زمان اجرا برای تغییر رفتار یک شیئ تعویض شود.
برای مثال فرض کنید که ما در حال طراحی یک برنامه مسیریابی برای یک شبکه هستیم. همانطوریکه می‌دانیم برای مسیر یابی الگوریتم‌های مختلفی وجود دارد که هر کدام دارای مزایا و معایبی هستند. و با توجه به وضعیت موجود شبکه یا عملی که قرار است انجام پذیرد باید الگوریتمی را که دارای بالاترین کارائی است انتخاب کنیم. همچنین این برنامه باید امکانی را به کاربر بدهد که کارائی الگوریتم‌های مختلف را در یک شبکه فرضی بررسی کنید. حالا طراحی پیشنهادی شما برای این مسئله چیست؟
دوباره فرض کنید که در مثال بالا در بعضی از الگوریتم‌ها نیاز داریم که گره‌های شبکه را بر اساس فاصله‌ی آنها از گره مبداء مرتب کنیم. دوباره برای مرتب سازی الگوریتم‌های مختلف وجود دارد و هر کدام در شرایط خاص، کارائی بهتری نسبت به الگوریتم‌های دیگر دارد. مسئله دقیقا شبیه مسئله بالا است و این مسله می‌توانند دارای طراحی شبیه مسله بالا باشد. پس اگر ما بتوانیم یک طراحی خوب برای این مسئله ارائه دهیم می‌توانیم این طراحی را برای مسائل مشابه به کار ببریم.

هر کدام از ما می‌توانیم نسبت به درک خود از مسئله و سلیقه کاری، طراح‌های مختلفی برای این مسئله ارائه دهیم. اما یک طراحی که می‌تواند یک جواب خوب و عالی باشد، الگوی استراتژی است که توانسته است بارها و بارها به این مسئله پاسخ بدهد.

الگوی استراتژی گزینه مناسبی برای مسائلی است که می‌توانند از چندین الگوریتم مختلف به مقصود خود برسند.

نمودار UML الگوی استراتژی به صورت زیر است :



اجازه بدهید، شیوه کار این الگو را با مثال مربوط به مرتب سازی بررسی کنیم. فرض کنید که ما تصمیم گرفتیم که از سه الگویتم زیر برای مرتب سازی استفاده کنیم.

1 - الگوریتم مرتب سازی Shell Sort 2 - الگوریتم مرتب سازی Quick Sort
3 - الگوریتم مرتب سازی Merge Sort

ما برای مرتب سازی در این برنامه دارای سه استراتژی هستیم. که هر کدام را به عنوان یک کلاس جداگانه در نظر می‌گیریم (همان کلاس‌های ConcreteStrategy). برای اینکه کلاس Client بتواند به سادگی یک از استراتژی‌ها را انتخاب کنید بهتر است که تمام کلاس‌های استراتژی دارای اینترفیس مشترک باشند. برای این کار می‌توانیم یک کلاس abstract تعریف کنیم و ویژگیهای مشترک کلاس‌های استراتژی را در آن قرار دهیم و کلاس‌های استراتژی آنها را به ارث ببرند(همان کلاس Strategy) و پیاده سازی کنند.

در زیل کلاس Abstract که کل کلاس‌های استراتژی از آن ارث می‌برند را مشاهده می‌کنید :

```
abstract class SortStrategy
{
        public abstract void Sort(ArrayList list);
}
```

کلاس مربوط به QuickSort

```
class QuickSort : SortStrategy
{
        public override void Sort(ArrayList list)
        {
          // الگوریتم مربوطه
        }


}
```

کلاس مربوط به ShellSort

```
class ShellSort : SortStrategy
{
        public override void Sort(ArrayList list)
        {
          // الگوریتم مربوطه
        }
}
```

کلاس مربوط به MergeSort

```
class MergeSort : SortStrategy
{
        public override void Sort(ArrayList list)
        {
          // الگوریتم مربوطه
        }
}
```

و در آخر کلاس Context که یکی از استراتژی‌ها را برای مرتب کردن به کار می‌برد :

```
class SortedList
{
        private ArrayList list = new ArrayList();
        private SortStrategy sortstrategy;

        public void SetSortStrategy(SortStrategy sortstrategy)
        {
            this.sortstrategy = sortstrategy;
        }
        public void Add(string name)
```

```
        {
            list.Add(name);
        }
        public void Sort()
        {
            sortstrategy.Sort(list);
        }
}
```

نظرات خوانندگان

نویسنده: علي
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۲:۴۶

با سلام؛ لطفا کلاس آخری را بیشتر توضیح دهید.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۲:۵۵

کلاس آخری با یک پیاده سازی عمومی کار می‌کنه. دیگه نمی‌دونه نحوه مرتب سازی چطور پیاده سازی شده. فقط می‌دونه یک متد Sort هست که دراختیارش قرار داده شده. حالا شما راحت می‌تونی الگوریتم مورد استفاده رو عوض کنی، بدون اینکه نیاز داشته باشی کلاس آخری رو تغییر بدی. باز هست برای توسعه. بسته است برای تغییر. به این نوع طراحی رعایت open closed principle هم می‌گن.

نویسنده: SB
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۴:۲۳

بنظر شما متد Sort کلاس اولیه، نباید از نوع Virtual باشد؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۴:۴۸

نوع کلاسش abstract هست.

نویسنده: مجتبی شاطری
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۶:۴۷

در صورتی از virtual استفاده می‌کنیم که یک پیاده سازی از متد Sort در SortStrategy داشته باشیم ، اما در اینجا طبق فرموده دوستمون کلاس ما فقط انتزاعی (Abstract) هست.

نویسنده: سيد ايوب كوكبي
تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۱:۲۲

چرا استراتژی توسط Abstract پیاده سازی شده و از اینترفیس استفاده نشده؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۲:۵۱

تفاوت مهمی نداره ؛ فقط اینترفیس ورژن پذیر نیست. یعنی اگر در این بین متدی رو به تعاریف اینترفیس خودتون اضافه کردید، تمام استفاده کننده‌ها مجبور هستند اون رو پیاده سازی کنند. اما کلاس Abstract می‌تونه شامل یک پیاده سازی پیش فرض متد خاصی هم باشه و به همین جهت ورژن پذیری بهتری داره.
بنابراین کلاس Abstact یک اینترفیس است که می‌تواند پیاده سازی هم داشته باشد.
همین مساله خاص نگارش پذیری، در طراحی ASP.NET MVC به کار گرفته شده: (^)
برای من نوعی شاید این مساله اهمیتی نداشته باشه. اگر من قرارداد اینترفیس کتابخانه خودم را تغییر دادم، بالاخره شما با یک حداقل نق زدن مجبور به به روز رسانی کار خودتان خواهید شد. اما اگر مایکروسافت چنین کاری را انجام دهد، هزاران نفر شروع خواهند کرد به بد گفتن از نحوه مدیریت پروژه تیم‌های مایکروسافت و اینکه چرا پروژه جدید آن‌ها با یک نگارش جدید MVC کامپایل نمی‌شود. بنابراین انتخاب بین این دو بستگی دارد به تعداد کاربر پروژه شما و استراتژی ورژن پذیری قرار دادهای کتابخانه‌ای که ارائه می‌دهید.

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۳:۲۷

اطلاعات خوبی بود، ممنون، ولی با توجه به تجربه تون، در پروژه‌های متن باز فعلی تحت بستر دات نت بیشتر از کدام مورد استفاده میشه؟ اینترفیس روحیه نظامی خاصی به کلاس‌های مصرف کننده اش میده، یه همین دلیل من زیاد رقبت به استفاده از اون ندارم، آیا مواردی هست که چاره ای نباشه حتما از یکی از این دو نوع استفاده بشه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۳:۴۹

- اگر پروژه خودتون هست، از اینترفیس استفاده کنید. تغییرات آن و نگارش‌های بعدی آن تحت کنترل خودتان است و build دیگران را تحت تاثیر قرار نمی‌دهد.
- در پروژه‌های سورس باز دات نت، عموما از ترکیب این دو استفاده می‌شود. مواردی که قرار است در اختیار عموم باشند حتی دو لایه هم می‌شوند. مثلا در MVC یک اینترفیس IController هست و بعد یک کلاس Abstract به نام Controller، که این اینترفیس را پیاده سازی کرده برای ورژن پذیری بعدی و کنترلرهای پروژه‌های عمومی MVC از این کلاس Abstract مشتق می‌شوند یا در پروژه RavenDB از کلاس‌های Abstract زیاد استفاده شده، مانند AbstractIndexCreationTask و AbstractMultiMapIndexCreationTask و غیره.

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۷/۰۱ ۱۶:۱۱

توابع abstract بطور ضمنی virtual هستند.

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۷/۰۱ ۱۸:۳۸

در کلاس abstract نیز می‌توان از پیاده سازی پیشفرض استفاده کرد . یکی از تفاوتهای کلاس abstract با Interface همین ویژگی است که سبب ورژن پذیری آن شده است.

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۸/۲۱ ۹:۱۵

بهتر نیست در کلاس SortedList برای مشخص کردن استراتژی مرتب سازی، از روش تزریق وابستگی - Dependency Injection - استفاده بشه؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۸/۲۱ ۹:۲۵

خوب، الان هم وابستگی کلاس یاد شده از طریق سازنده آن در اختیار آن قرار گرفته و داخل خود کلاس وهله سازی نشده. (در این مطلب طراحی بیشتر مدنظر هست تا اینکه حالا این وابستگی به چه صورتی و کجا قرار هست وهله سازی بشه و در اختیار کلاس قرار بگیره؛ این مساله ثانویه است)

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۸/۲۱ ۱۱:۴۳

از طریق سازنده کلاس SortedList؟ بنظر نمیاداز طریق سازنده انجام شده باشه. ولی ظاهرا این امکان هست که کلاس بالادستی که می‌خواهد از SortedList استفاده کند، بتواند از طریق تابع SetSortStrategy کلاس مورد نظر رادر اختیار SortedList قراردهد. به نظر شبیه Setter Injection می‌شود.

عنوان: **چرا توسعه چابک (Agile Development)؟**
نویسنده: توسعه چابک
تاریخ: ۱۳۹۲/۰۹/۱۱ ۱۲:۲۵
آدرس: www.dotnettips.info
گروه‌ها: Scrum, چابک, SOLID, Agile Developement, شی گرایی

خیلی از ما با کابوس پروژه ای که هیچ تجربه ای در انجام آن نداریم روبرو شده ایم. نبودن تجربه موثر منجر به خطاهای تکراری و غیر قابل پیش بینی شده و تلاش و وقت ما را به هدر می-دهد. مشتریان از کیفیت پایین، هزینه بالا و تحویل دیر هنگام محصول ناراضی هستند و توسعه دهندگان از اضافه کارهای بیشتر که منجر به نرم افزار ضعیت-تر می-گردد، ناخشنود.

همین که با شکستی مواجه می-شویم از تکرار چنین پروژه هایی اجتناب می-کنیم. ترس ما باعث می-شود تا فرآیندی بسازیم که فعالیت-های ما را محدود نموده و ایجاد آرتیفکت-ها [۱] را الزامی کند. در پروژه- جدید از چیزهایی که در پروژه‌های قبلی به خوبی کار کرده-اند، استفاده می-کنیم. انتظار ما این است که آنها برای پروژه جدید نیز به همان خوبی کار کند.

اما پروژه-ها آنقدر ساده نیستند که تعدادی محدودیت و آرتیفکت- ما را از خطاها ایمن سازند. با بروز خطاهای جدید ما آنها را شناسایی و رفع می-کنیم. برای اینکه در آینده با این خطاها روبرو نشویم آنها را در محدودیت-ها و آرتیفکت-های جدیدی قرار می-دهیم. بعد از انجام پروژه‌های زیاد با فرآیندهای حجیم و پر زحمتی روبرو هستیم که توانایی تیم را کم کرده و باعث کاهش کیفیت تولید می-شوند.

فرآیندهای بزرگ و حجیم می-تواند مشکلات زیادی را ایجاد کند. متاسفانه این مشکلات باعث می-شود که خیلی از افراد فکر کنند که علت مشکلات، نبود فرآیندهای کافی است. بنابراین فرآیندها را حجیم-تر و پیچیده-تر می-کنند. این مسئله منجر به تورم فرآیندها می-گردد که در محدوده سال ۲۰۰۰ گریبان بسیاری از شرکت-های نرم افزاری را گرفت.

اتحاد چابک

در وضعیتی که تیم-های نرم افزاری در بسیاری از شرکت-ها خود را در مردابی از فرآیندهای زیاد شونده می-دیدند، تعدادی از خبره-های این صنعت که خود را اتحاد چابک [۲] نامیدند در اویل سال ۲۰۰۱ یکدیگر را ملاقات کرده و ارزش هایی را معرفی کردند تا تیم-های نرم افزاری سریعتر نرم افزار را توسعه داده و زودتر به تغییرات پاسخ دهند. چند ماه بعد، این گروه ارزش-هایی تعریف شده را تحت مانیفست اتحاد چابک در سایت http://agilemanifesto.org منتشر کردند.

مانیفست اتحاد چابک
ما با توسعه نرم افزار و کمک به دیگران در انجام آن، در حال کشف راه‌های بهتری برای توسعه نرم افزار هستیم. از این کار به ارزش‌های زیر می-رسیم :

۱- افراد و تعاملات بالاتر از فرآیندها و ابزارها

۲- نرم افزار کار کننده بالاتر از مستندات جامع

۳- مشارکت مشتری بالاتر از قرارداد کاری

۴- پاسخگویی به تغییرات بالاتر از پیروی از یک برنامه

با آنکه موارد سمت چپ ارزشمند هستند ولی ما برای موارد سمت راست ارزش بیشتری قائل هستیم.

افراد و تعاملات بالاتر از فرآیندها و ابزارها
افراد مهمترین نقش را در پیروزی یک پروژه دارند. یک فرآیند عالی بدون نیروی مناسب منجر به شکست می-گردد و بر عکس

افراد قوی تحت فرآیند ضعیت ناکارآمد خواهند بود.

یک نیروی قوی لازم نیست که برنامه نویسی عالی باشد، بلکه کافیست که یک برنامه نویسی معمولی با قابلیت همکاری مناسب با سایر اعضای تیم باشد. کار کردن با دیگران، تعامل درست و سازنده با سایر اعضای تیم خیلی مهمتر از این که یک برنامه نویس با هوش باشد. برنامه نویسان معمولی که تعامل درستی با یکدیگر دارند به مراتب موفقتر هستند از تعداد برنامه نویسی عالی که قدرت تعامل مناسب با یکدیگر را ندارند.

در انتخاب ابزارها آنقدر وقت نگذارید که کار اصلی و تیم را فراموش کنید. به عنوان مثال می-توانید در شروع به جای بانک اطلاعاتی از فایل استفاده کنید، به جای ابزار کنترل کد گرانقیمت از برنامه رایگان کد باز استفاده کنید. باید به هیچ ابزاری عادت نکنید و صرفا به آنها به عنوان امکانی جهت تسهیل فرآیندها نگاه کنید.

نرم افزار کار کننده بالاتر از مستندات جامع

نرم افزار بدون مستندات، فاجعه است. کد برنامه ابزار مناسبی برای تشریح سیستم نرم افزاری نیست. تیم باید مستندات قابل فهم مشتری بسازد تا ابعاد سیستم از تجزیه تحلیل تا طراحی و پیاده سازی آن را تشریح نماید.

با این حال، مستندات زیاد از مستندات کم بدتر است. ساخت مستندات زیاد نیاز به وقت زیادی دارد و وقت بیشتری را می‌گیرد تا آن را با کد برنامه به روز نمایید. اگر آنها با یکدیگر به روز نباشند باعث درک اشتباه از سیستم می‌شوند.

بهتر است که همیشه مستندات کم حجمی از منطق و ساختار برنامه داشته باشید و آن را به روز نماید. البته آنها باید کوتاه و برجسته باشند. کوتاه به این معنی که ۱۰ تا ۲۰ صفحه بیشتر نباشد و برجسته به این معنی که طراحی کلی و ساختار سطح بالای سیستم را بیان نماید.

اگر فقط مستندات کوتاه از ساختار و منطق سیستم داشته باشیم چگونه می‌توانیم اعضای جدید تیم را آموزش دهیم؟ پاسخ کار نزدیک شدن به آنها است. ما دانش خود را با نشستن در کنار آنها و کمک کردن به آنها انتقال می-دهیم. ما آنها را بخشی از تیم می-کنیم و با تعامل نزدیک و رو در رو به آنها آموزش می-دهیم.

مشارکت مشتری بالاتر از قرارداد کاری

نرم افزار نمی-تواند مثل یک جنس سفارش داده شود. شما نمی‌توانید یک توصیف از نرم افزاری که می‌خواهید را بنویسید و آنگاه فردی آن را بسازد و در یک زمان معین با قیمت مشخص به شما تحویل دهد. بارها و بارها این شیوه با شکست مواجه شده است.

این قابل تصور است که مدیران شرکت به اعضای تیم توسعه بگویند که نیازهای آنها چیست، سپس اعضای تیم بروند و بعد از مدتی برگردند و یک سیستمی که نیازهای آنها را برآورده می‌کند، بسازند. اما این تعامل به کیفیت پایین نرم افزار و در نهایت شکست آن می‌انجامد. پروژه‌های موفق بر اساس دریافت بازخورد مشتری در بازه‌های زمانی کوتاه و مداوم است. به جای وابستگی به قرارداد یا دستور کار، مشتری به طور تنگاتنگ با تیم توسعه کار کرده و مرتبا اعمال نظر می-کند.

قراردادی که مشخص کننده نیازمندیها، زمانبندی و قیمت پروژه است، اساسا نقص دارد. بهترین قرارداد این است که تیم توسعه و مشتری با یکدیگر کار کنند.

پاسخگویی به تغییرات بالاتر از پیروی از یک برنامه

توانایی پاسخ به تغییرات اغلب تعیین کننده موفقیت یا شکست یک پروژه نرم افزاری است. وقتی که طرحی را می-ریزیم باید مطمئن شویم که به اندازه کافی انعطاف پذیر است و آمادگی پذیرش تغییرات در سطح بیزنس و تکنولوژی را دارد.

مسیر یک پروژه نرم افزاری نمی-تواند برای بازه زمانی طولانی برنامه ریزی شود. اولا احتمالا محیط تغییر می-کند و باعث تغییر در نیازمندی‌ها می-شود. ثانیا همین که سیستم شروع به کار کند مشتریان نیازمندی-های خود را تغییر می‌دهند. بنابراین اگر بدانیم که نیازها چیست و مطمئن شویم که تغییر نمی-کنند، قادر به برآورد مناسب خواهیم بود، که این شرایط بعید است.

یک استراتژی خوب برای برنامه ریزی این است که یک برنامه ریزی دقیق برای یک هفته بعد داشته باشیم و یک برنامه ریزی کلی برای سه ماه بعد.

اصول چابک

-۱ بالاترین اولویت ما عبارت است از راضی کردن مشتری با تحویل سریع و مداوم نرم افزار با ارزش. تحویل نرم افزار با کارکردهای کم در زود هنگام بسیار مهم است چون هم مشتری چشم اندازی از محصول نهایی خواهد داشت و هم مسیر کمتر به بیراهه می‌رود.

-۲ خوش آمدگویی به تغییرات حتی در انتهای توسعه. اعضای تیم چابک، تغییرات را چیز خوبی می‌بینند زیرا تغییرات به این معنی است که تیم بیشتر یاد گرفته است که چه چیزی مشتری را راضی می‌کند.

-۳ تحویل نرم افزار قابل استفاده از چند هفته تا چند ماه با تقدم بر تحویل در دوره زمانی کوتاهتر . ما مجموعه از مستندات و طرحها را به مشتری نمی‌دهیم.

-۴ افراد مسلط به بیزنس و توسعه دهندگان باید روزانه با یکدیگر روی پروژه کار کنند . یک پروژه نرم افزاری نیاز به هدایت مداوم دارد.

-۵ ساخت پروژه را بر توان افراد با انگیزه بگذارید و به آنها محیط و ابزار را داده و اعتماد کنید . مهمترین فاکتور موفقیت افراد هستند، هر چیز دیگر مانند فرآیند، محیط و مدیریت فاکتورهای بعدی محسوب می-شوند که اگر تاثیر بدی روی افراد می-گذارند، باید تغییر کنند.

-۶ بهترین و موثرترین روش کسب اطلاعات در تیم توسعه، ارتباط چهره به چهره است. در تیم چابک افراد با یکدیگر صحبت می‌کنند. نامه نگاری و مستند سازی فقط زمانی که نیاز است باید صورت گیرد.

-۷ نرم افزار کار کننده معیار اصلی پیشرفت است . پروژه‌های چابک با نرم افزاری که در حال حاضر نیازهای مشتری را پاسخ می‌دهد، سنجیده می‌شوند. میزان مستندات، حجم کدهای زیر ساخت و هر چیز دیگری غیره از نرم افزار کار کننده معیار پیشرفت نرم افزار نیستند.

-۸ فرآیندهای چابک توسعه با آهنگ ثابت را ترویج می‌دهد. حامیان، توسعه دهندگان و کاربران باید یک آهنگ توسعه ثابت را حفظ کنند که بیشتر شبیه به دو ماراتون است یا دوی ۱۰۰ متر. آنها با سرعتی کار می‌کنند که بالاترین کیفیت را ارائه دهند.

-۹ توجه مداوم به برتری تکنیکی و طراحی خوب منجر به چابکی می-گردد . کیفیت بالاتر کلیدی برای سرعت بالا است. راه سریعتر رفتن این است که نرم افزار تا جایی که ممکن است پاک و قوی نگهداریم. بنابراین همه اعضای تیم چابک تلاش می-کنند که با کیفیت-ترین کار ممکن را انجام دهند. آنها هر آشفتگی را به محض ایجاد برطرف می‌کنند.

-۱۰ سادگی هنر بیشینه کردن مقدار کاری که لازم نیست انجام شود، است. تیم چابک همیشه ساده‌ترین مسیر که با هدف آنها سازگار است را در پیش می-گیرند. آنها وقت زیادی روی مشکلاتی که ممکن است فردا رخ دهد، نمی-گذارند. آنها کار امروز را با کیفیت انجام داده و مطمئن می-شوند که تغییر آن در صورت بروز مشکلات در فردا، آسان خواهد بود.

-۱۱ بهترین معماری و طراحی از تیم‌های خود سازمان ده بیرون می‌آید. مدیران، مسئولیت‌ها را به یک فردی خاصی در تیم نمی‌دهند بلکه بر عکس با تیم به صورت یک نیروی واحد برخورد می-کنند. خود تیم تصمیم می-گیرد که هر مسئولیت را چه کسی انجام دهد. تیم چابک با هم روی کل جنبه‌های پروژه کار می-کنند. یعنی یک فرد خاص مسئول معماری، برنامه نویسی، تست و غیره نیستند. تیم، مسئولیتها را به اشتراک گذاشته و هر فرد بر کل کار تاثیر دارد.

-۱۲ در بازهای زمانی مناسب تیم در می-یابد که چگونه می-تواند کاراتر باشد و رفتار خود را متناسب با آن تغییر دهد. تیم

می-داند که محیط دائما در حال تغییر است، بنابراین خود را با محیط تغییر می-دهد تا چابک بماند.

ضرورت توسعه چابک

امروزه صنعت نرم افزار دارای سابقه بدی در تحویل به موقع و با کیفیت نرم افزار است. گزارشات بسیاری تایید می-کنند که بیش از ۸۰ درصد از پروژه‌های نرم افزاری با شکست مواجه می-شوند؛ در سال ۲۰۰۵ موسسه IEEE برآورد زده است که بیش از ۶۰ بیلیون دلار صرف پروژه‌های نرم افزاری شکست خورده شده است. عجب فاجعه-ای؟

شش دلیل اصلی شکست پروژه‌های نرم افزاری

وقتی که از مدیران و کارکنان سوال می-شود که چرا پروژه‌های نرم افزاری با شکست مواجه می-شوند، آنها به موضوعات گسترده ای اشاره می-کنند. اما شش دلیل زیر بارها و بارها تکرار شده است که به عنوان دلایل اصلی شکست نرم افزار معرفی می-شوند:

-۱ درگیر نشدن مشتری

-۲ عدم درک درست نیازمندها

-۳ زمان بندی غیر واقعی

-۴ عدم پذیریش و مدیریت تغییرات

-۵ کمبود تست نرم افزار

-۶ فرآیندهای غیر منعطف و باد دار

چگونه چابکی این مشکلات را رفع می-کند؟

با آنکه Agile برای هر مشکلی راه حل ندارد ولی برای مسائل فوق بدین صورت کمک می-کند:

مشتری پادشاه است!

برای رفع مشکل عدم همکاری کاربر نهایی یا مشتری، Agile مشتری را عضوی از تیم توسعه می-کند. به عنوان عضوی از تیم، مشتری با تیم توسعه کار می-کند تا مطمئن شود که نیازمندها به درستی برآورده می-شوند. مشتری همکاری می-کند در شناسایی نیازمندی-ها، تایید می-کند نتیجه نهایی را و حرف آخر را در اینکه کدام ویژگی به نرم افزار اضافه شود، حذف شود و یا تغییر کند، را می-زند.

نیازمندی‌ها به صورت تست-های پذیرش [۳] نوشته می-شوند

برای مقابله با مشکل عدم درک درست نیازمندی-ها، Agile تاکید دارد که نیازمندیهای کسب شده باید به صورت ویژگی-هایی تعریف شوند که بر اساس معیارهای مشخصی قابل پذیرش باشند. این معیارهای پذیرش برای نوشتن تست-های پذیرش به کار می-روند. به این ترتیب قبل از اینکه کدی نوشته شود، ابتدا تست پذیرش نوشته می-شود. این بدین معنی است که هر کسی باید اول فکر کند که چه می-خواهد، قبل از اینکه از کسی بخواهد آن را انجام دهد. این راهکار فرایند کسب نیازمندی-ها را از بنیاد تغییر می-دهد و به صورت چشم گیری کیفیت برآورد و زمان بندی را بهبود می-دهد.

زمانبندی با مذاکره بین تیم توسعه و سفارش دهنده تنظیم می-شود

برای حل مشکل زمان بندی غیر واقعی، Agile زمان بندی را به صورت یک فرآیند مشترک بین تیم توسعه و سفارش دهنده تعریف می-کند. در شروع هر نسخه از نرم افزار، سفارش دهنده ویژگی‌های مورد انتظار را به تیم توسعه می-گوید. تیم توسعه تاریخ تحویل را بر اساس ویژگی-ها برآورد می-زد و در اختیار سفارش دهنده قرار می-دهد. این تعامل تا رسیدن به یک دیدگاه مشترک ادامه می-یابد.

هیچ چیزی روی سنگ حک نشده است، مگر تاریخ تحویل

برای رفع مشکل ضعف در مدیریت تغییرات، Agile اصرار دارد که هر کسی باید تغییرات را بپذیرد و نسبت به آنها واقع بین باشد. یک اصل مهم Agile می-گوید که هر چیزی می-تواند تغییر کند مگر تاریخ تحویل! به عبارت دیگر همین که محصول به سمت تولید شدن حرکت می-کند، مشتری (در تیم محصول) می-تواند بر اساس اولویت-ها و ارزش-های خود ویژگی-های محصول را کم یا زیاد کرده و یا تغییر دهد. به هر حال او باید واقع بین باشد. اگر او یک ویژگی جدید اضافه کنید، باید تاریخ تحویل را تغییر دهد. به این ترتیب همیشه تاریخ تحویل رعایت می-گردد.

تست-ها قبل از کد نوشته می-شوند و کاملا خودکار هستند

برای رفع مشکل کمبود تست، Agile تاکید می-کند که ابتدا باید تست-ها نوشته شوند و همواره ارزیابی گردند. هر برنامه نویس باید اول تست- را بنویسد، سپس کد لازم برای پاس شدن آن را. همین که کد تغییر می-کند باید تست-ها دوباره اجرا شوند. در این راهکار، هر برنامه نویس مسئول تست-های خود است تا درستی برنامه از ابتدا تضمین گردد.

مدیریت پروژه یک فعالیت جداگانه نیست

برای رفع مشکل فرآیندهای غیر منعطف و باددار، Agile مدیریت پروژه را درون فرآیند توسعه می-گنجاند. وظایف مدیریت پروژه بین اعضای تیم توسعه تقسیم می-شود. برای مثال هر ۷ نفر در تیم توسعه نرم افزار (متدلوژی اسکرام) زمان تحویل را با مذاکره تعیین می-کنند. همچنین کد برنامه به صورت خودکار اطلاعات وضعیت پروژه را تولید می-کند. به عنوان مثال نمودار burndown ، تست-های انجام نشده، پاس شده و رد شده به صورت خودکار تولید می-شوند.

به کار گیری توسعه چابک

یکی از مشکلات توسعه چابک این است که شما اول باید به خوبی آن را درک کنید تا قادر به پیاده سازی درست آن باشید. این درک هم باید کلی باشد (مانند Scrum و XP) و هم جزئی (مانند TDD و جلسات روازنه). اما چگونه باید به این درک برسیم؟ کتاب-ها و مقالات انگلیسی زیادی برای یادگیری توسعه چابک و پیاده سازی آن در سازمان وجود دارند، ولی متاسفانه منابع فارسی کمی در این زمینه است. هدف این کتاب رفع این کمبود و آموزش عملی توسعه چابک و ابزارهای پیاده سازی آن است.

برای این یک توسعه دهنده چابک شوید، باید به مهارت-های فردی و تیمی چابک برسید. در ادامه این مهارت-ها معرفی می-شوند.

مهارت-های فردی

قبل از هر چیز شما باید یک برنامه نویس باشید و مقدمات برنامه نویسی مانند الگوریتم و فلوچارت، دستورات برنامه نویسی، کار با متغیرها، توابع و آرایه-ها را بلد باشید. پس از تسلط به مقدمات برنامه نویسی می-توانید مهارت-های برنامه نویسی چابک را فرا بگیرید که عبارتند از:

- برنامه نویسی شیءگرا

- توسعه تست محور

- الگوهای طراحی

در ادامه نحوه کسب این مهارت-ها بیان می-شوند.

برنامه نویسی شیءگرا

اساس طراحی چابک بر تفکر شیءگرا استوار است. بنابراین تسلط به مفاهیم و طراحی شیءگرا ضروری است.

توسعه تست محور

مهمترین و انقلابی‌ترین سبک برنامه نویسی از دهه گذشته تا به امروز، توسعه یا برنامه نویسی تست محور است. این سبک بسیاری از ارزش‌های توسعه چابک را فراهم می-کند و یادگیری آن برای هر توسعه دهنده چابک ضروری است.

الگوهای طراحی

الگوهای طراحی راه حل-های انتزاعی سطح بالا هستند. این الگوها بهترین تکنیک-های [۴] طراحی نرم افزار هستند و بسیاری از مشکلاتی که در طراحی نرم افزار رخ می-دهند با استفاده از این الگوها قابل حل هستند.

مهارت-های تیمی

انجام پروژه نرم افزاری یک کار تیمی است. شما پس از یادگیری مهارت-های فردی باید خود را آماده حضور در تیم توسعه چابک کنید. برای این منظور باید با مهارت تیمی مانند آشنایی با گردشکار تولید نرم افزار، حضور موثر در جلسات، قبول مسئولیت-ها و غیره آشنا شوید.

اسکرام

تمامی مهارت-های تیمی توسعه چابک توسط اسکرام آموزش داده می-شوند. اسکرام فریم ورکی برای توسعه چابک است که با تعریف فرآیندها، نقش-ها و آرتیفکت-های مشخص به تیم-های نرم افزاری کمک می-کند تا چابک شوند.

[۱] Artifact : خروجی یک فرآیند است. مثلا خروجی طراحی شیءگرا، نمودارهای UML است.

[۲] Agile Alliance

[3] Acceptance Tests

[4] Best Practice

اطلاعات بیشتر در http://AgileDevelopment.ir

نظرات خوانندگان

نویسنده: ایمان
تاریخ: ۱۳۹۲/۱۰/۰۱ ۲۳:۵۸

شاید آدرس زیر هم به کار بیاد

Imanagement.co

فیلم‌های آموزشی رایگان راجع به مدیریت پروژه‌های نرم افزاری به شیوه اجایل

عنوان: **Design Pattern: Factory**
نویسنده: مهسا حسن کاشی
تاریخ: ۱۳۹۲/۱۱/۲۶ ۲۲:۱۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Design patterns, SOLID, Object Oriented Design, ServiceFactory, Patterns, Software architecture

الگوهای طراحی، سندها و راه حلهای از پیش تعریف شده و تست شده‌ای برای مسائل و مشکلات روزمره‌ی برنامه نویسی می‌باشند که هر روزه ما را درگیر خودشان می‌کنند. هر چقدر مقیاس پروژه وسیعتر و تعداد کلاسها و اشیاء بزرگتر باشند، درگیری برنامه نویس و چالش برای مرتب سازی و خوانایی برنامه و همچنین بالا بردن کارآیی و امنیت افزون‌تر می‌شود. از همین رو استفاده از ساختارهایی تست شده برای سناریوهای یکسان، امری واجب تلقی می‌شود.

الگوهای طراحی از لحاظ سناریو، به سه گروه عمده تقسیم می‌شوند:

-1 تکوینی: هر چقدر تعداد کلاسها در یک پروژه زیاد شود، به مراتب تعداد اشیاء ساخته شده از آن نیز افزوده شده و پیچیدگی و درگیری نیز افزایش می‌یابد. راه حل‌هایی از این دست، تمرکز بر روی مرکزیت دادن به کلاسها با استفاده از رابطها و کپسوله نمودن (پنهان سازی) اشیاء دارد.

-2 ساختاری: گاهی در پروژه‌ها پیش می‌آید که می‌خواهیم ارتباط بین دو کلاس را تغییر دهیم. از این رو امکان از هم پاشی اجزایِ دیگر پروژه پیش می‌آید. راه حلهای ساختاری، سعی در حفظ انسجام پروژه در برابر این دست از تغییرات را دارند.

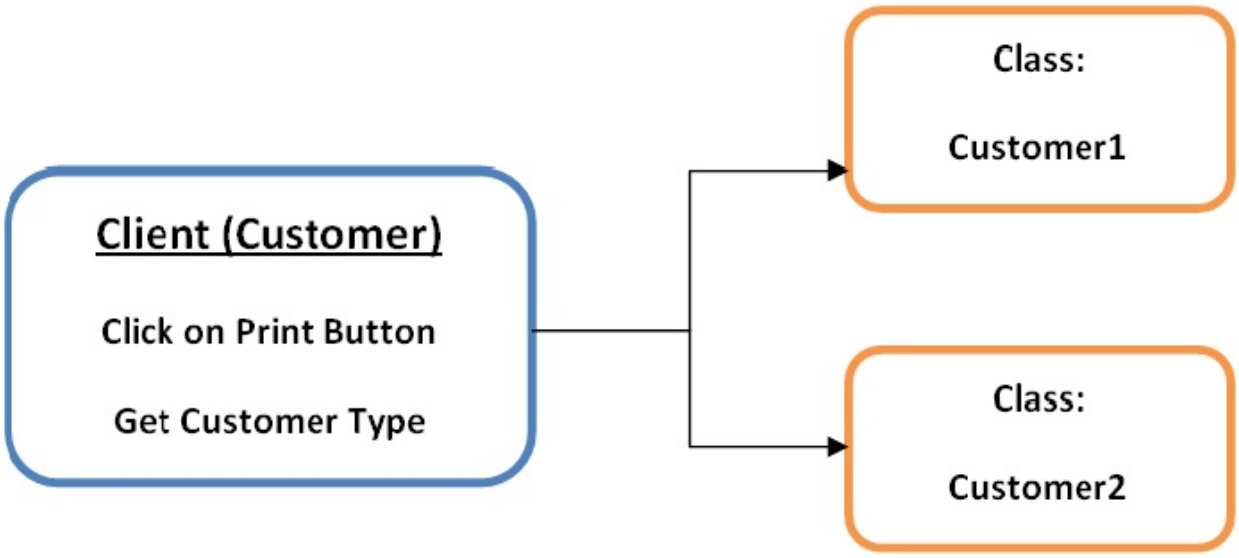
-3 رفتاری: گاهی بنا به مصلحت و نیاز مشتری، رفتار یک کلاس می‌بایستی تغییر نماید. مثلا چنانچه کلاسی برای ارائه صورتحساب داریم و در آن میزان مالیات 30% لحاظ شده است، حال این درصد باید به عددی دیگر تغییر کند و یا پایگاه داده به جای مشاهده‌ی تعدادِ معدودی گره از درخت، حال می‌بایست تمام گره‌ها را ارائه نماید.

الگوی فکتوری:

الگوی فکتوری در دستهء اول قرار می‌گیرد. من در اینجا به نمونه‌ای از مشکلاتی که این الگو حل می‌نماید، اشاره می‌کنم:

فرض کنید یک شرکت بزرگ قصد دارد تا جزییات کامل خرید هر مشتری را با زدن دکمه چاپ ارسال نماید. چنین شرکت بزرگی بر اساس سیاستهای داخلی، بر حسب میزان خرید، مشتریان را به چند گروه مشتری معمولی و مشتری ممتاز تقسیم می‌نماید. در نتیجه نمایش جزییات برای آنها با احتساب میزان تخفیف و به عنوان مثال تعداد فیلدهایی که برای آنها در نظر گرفته شده است، تفاوت دارد. بنابراین برای هر نوع مشتری یک کلاس وجود دارد.

یک راه این است که با کلیک روی دکمه‌ی چاپ، نوع مشتری تشخیص داده شود و به ازای نوع مشتری، یک شیء از کلاس مشخص شده برای همان نوع ساخته شود.



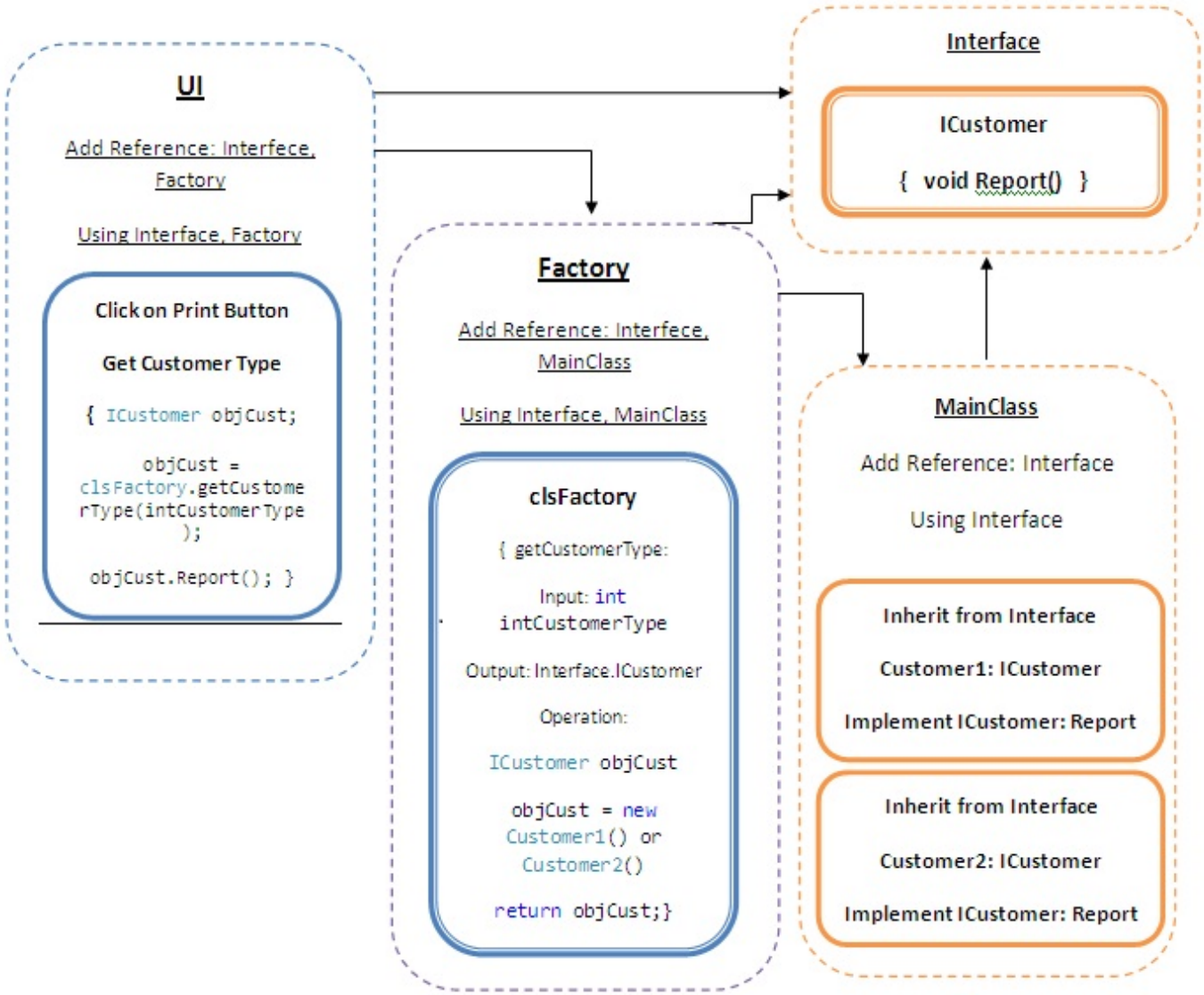

```
// Get Customer Type from Customer click on Print Button
            int customerType = 0;

            // Create Object without instantiation
            object obj;


            //Instantiate obj according to customer Type
            if (customerType == 1)
            {
                obj = new Customer1();
            }
            else if (customerType == 2)
            {
                obj = new Customer2();
            }
            // Problem:
            //          1: Scattered New Keywords
            //          2: Client side is aware of Customer Type
```

همانگونه که مشاهده می‌نمایید در این سبک کدنویسی غیرحرفه‌ای، مشکلاتی مشهود است که قابل اغماض نیستند. در ابتدا سمت کلاینت دسترسی مستقیم به کلاسها دارد و همانگونه که در شکل بالا قابل مشاهده است کلاینت مستقیما به کلاس وصل است. مشکل دوم عدم پنهان سازی کلاس از دید مشتری است.

راه حل: این مشکل با استفاده از الگوی فکتوری قابل حل است. با استناد به الگوی فکتوری، کلاینت تنها به کلاس فکتوری و یک اینترفیس دسترسی دارد و کلاسهای فکتوری و اینترفیس، حق دسترسی به کلاسهای اصلی برنامه را دارند.



گام نخست: در ابتدا یک class library به نام Interface ساخته و در آن یک کلاس با نام ICustomer می سازیم که متد Report() را معرفی می‌نماید.

Interface//

```
namespace Interface
{
    public interface ICustomer
    {
        void Report();
    }
}
```

گام دوم: یک class library به نام MainClass ساخته و با Add Reference کلاس Interface را اضافه نموده، در آن دو کلاس با نام Customer1, Customer2 می‌سازیم و using Interface را Import می‌نماییم. هر دو کلاس از ICustomer ارث می‌برند و سپس متد Report() را در هر دو کلاس Implement می‌نماییم.

```
// Customer1
using System;
```

```
using Interface;

namespace MainClass
{
    public class Customer1 : ICustomer
    {
        public void Report()
        {
            Console.WriteLine("این گزارش مخصوص مشتری نوع اول است");
        }
    }
}

//Customer2
using System;
using Interface;

namespace MainClass
{
   public class Customer2 : ICustomer
    {
        public void Report()
        {
            Console.WriteLine("این گزارش مخصوص مشتری نوع دوم است");
        }
    }
}
```

گام سوم: یک class library به نام FactoryClass ساخته و با Add Reference کلاس Interface, MainClass را اضافه نموده، در آن یک کلاس با نام clsFactory می سازیم و using Interface, using MainClass را Import می‌نماییم. پس از آن یک متد با نام getCustomerType ساخته که ورودی آن نوع مشتری از نوع int است و خروجی آن از نوع Interface-ICustomer و بر اساس کد نوع مشتری object را از کلاس Customer1 و یا Customer2 می‌سازیم و آن را return می نماییم.

```
//Factory
using System;
using Interface;
using MainClass;

namespace FactoryClass
{
    public class clsFactory
    {
        static public ICustomer getCustomerType(int intCustomerType)
        {
            ICustomer objCust;
            if (intCustomerType == 1)
            {
                objCust = new Customer1();
            }
            else if (intCustomerType == 2)
            {
                objCust = new Customer2();
            }
            else
            {
                return null;
            }
            return objCust;
        }
    }
}
```

گام چهارم (آخر): در قسمت UI Client، کد نوع مشتری را از کاربر دریافت کرده و با Add Reference کلاس Interface, FactoryClass را اضافه نموده (دقت نمایید هیچ دسترسی به کلاس‌های اصلی وجود ندارد)، و using Interface, using FactoryClass را Import می‌نماییم. از clsFactory تابع getCustomerType را فراخوانی نموده (به آن کد نوع مشتری را پاس می‌دهیم) و خروجی آن را که از نوع اینترفیس است به یک object از نوع ICustomer نسبت می‌دهیم. سپس از این object متد Report را فراخوانی می‌نماییم. همانطور که از شکل و کدها مشخص است، هیچ رابطه ای بین UI(Client) و کلاسهای اصلی برقرار نیست.

```
//UI (Client)
using System;
using FactoryClass;
using Interface;

namespace DesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            int intCustomerType = 0;
            ICustomer objCust;
            Console.WriteLine("نوع مشتری را وارد نمایید");
            intCustomerType = Convert.ToInt16(Console.ReadLine());
            objCust = clsFactory.getCustomerType(intCustomerType);
            objCust.Report();
            Console.ReadLine();
        }
    }
}
```