به نام خدا

آزمون پایان ترم

انجام دهنده:

محمد سعید کودک پور

810196540

دانشکده مهندسی برق و کامپیوتر دانشگاه تهران

پاسخ سوال اول

تعریف لفافه بندی یا Encapsulation

Encapsulation یعنی اینکه مقادیر یا state داخلی یک ساختار داده یا یک شی داخل یک کلاس را مخفی بکنیم و از اینکه افراد، کدها، فانکشن ها و یوزرهای دیگر آن آبجکت، دسترسی مستقیم به آن ها داشته باشند، جلوگیری بکنیم. مزیت لفافه بندی این است که رفتار آبجکت مورد نظر است که تعیین می کند استیت آن چگونه تغییر بکند. این مورد باعث می شود که هم استیت آبجکت به صورت غیر قابل انتظار تغییر نکند و هم با تغییر استیت داخلی یک آبجکت یا یک کلاس، افراد در حال استفاده از آن آبجکت، تحت تاثیر آن تغییر قرار نمی گیرند و این دو مورد باعث مدیریت بهتر وابستگی ها می شوند.

تعریف وراثت یا Inheritance

Inheritance یعنی اینکه یک کلاس می تواند رفتار داخلی و عملیات را از یک کلاس دیگر ارث ببرد، این رفتارها و عملیات را تغییر دهد و یا رفتارها و عملیات جدیدی اضافه بکند؛ اما نمیتواند رفتارهای قبلی را از بین ببرد. این مورد به دو شکل زیر معنی می شود:

1-رابطه ISA و sub-classing یا sub-typing

2-وراثت به صورت Implementation Inheritance

تعریف چند ریختی یا Polymorphism

Polymorhism یعنی قابلیتی که یک آبجکت، سمبل، نام یا علامت بتواند در برنامه اشکال یا رفتارهای مختلفی را داشته باشد. در واقع توانایی اینکه یک آبجکت بتواند جند فرم مختلف را به خود بگیرد. همچنین معمول ترین استفاده چندریختی در برنامه نویسی شی گرا، زمانی اتفاق می افتد که رفرنس کلاس والد، برای اشاره به آبجکتی از کلاس فرزند مورد استفاده قرار می گیرد (Dynamic Binding or Late Binding). مدل های دیگر چندریختی شامل Template Class ها یا انواع مختلف Overloading می باشند.

پاسخ سوال دوم

الگوی طراحی Strategy

الگوی طراحی Strategy یک الگوی طراحی رفتاری است که به ما این امکان را می دهد که یک خانواده از الگوریتم ها تعریف کنیم، هر کدام از آن ها را در یک کلاس جداگانه قرار دهیم و آن ها را encapsulate کنیم و آبجکت های آن ها را قابل تعویض یا قابل مبادله بکنیم.

الگوی Strategy این امکان را فراهم می کند که الگوریتم مستقل از کلاینت هایی که از آن استفاده می کنند، متغیر باشد.

با بیان دو مسئله، دلیل وجود این الگوی طراحی و نحوه کار آن را توضیح می دهیم:

مسئله اول

فرض کنید می خواهیم یک اپلیکیشن navigation برای مسافران بسازیم. این اپلیکیشن با داشتن یک نقشه به مسافران کمک می کند که به سادگی به هر شهری می خواهند جهت گیری کنند. فرض کنید پس از اولین ورژن اپ، ویژگی برنامه ریزی خودکار مسیر بیشترین درخواست را داشته باشد که یوزر بتواند با وارد کردن آدرس، سریع ترین مسیر به مقصد را روی نقشه ببیند. در ورژن بعدی، اپلیکیشن می تواند مسیرهای گذرنده از جاده را پیدا کند، این تنها برای رانندگان مناسب بود و سایر افراد که با سایر وسیله ها یا حتی پیاده دنبال مسیر بهینه بودند، نمی توانستند استفاده خوبی از اپلیکیشن داشته باشند. پس در آپدیت بعدی، این ویژگی را برای مسیرهای پیاده روی اضافه می کنیم. در آپدیت بعدتر، این مورد را برای وسایل نقلیه عمومی اضافه می کنیم. این روند همینطور ادامه دارد و باید برای مسیرهای دوچرخه سواری و تمامی وسایل نقلیه دیگر هم اضافه شود.

حال از منظر کسب و کار، اضافه کردن این موارد، یک اپلیکشن موفق را وعده می دهد. اما از نظر فنی، هر بار که الگوریتم مسیریابی جدید اضافه می شود، کلاس اصلی مسیریاب، ار نظر سایزی دو برابر می شود و از یک جایی به بعد، انقدر بزرگ می شود که maintain آن دیگر امکان پذیر نیست.

علاوه بر آن هر تغییر در یکی از الگوریتم ها، شامل رفع باگ یا هرگونه تغییر دیگر، تمامی کلاس را تحت تاثیر قرار می دهد و امکان ایجاد خطا در کدی که قبلاً کار می کرده را بالا می برد. همچنی کار تیمی دیگر موثر نخواهد بود، زیرا اعضایی از تیم که بلافاصله پس از یک ریلیز موفقیت آمیز استخدام شده اند، شکایت می کنند که وقت زیادی را برای حل تعارضات مرج، صرف می کنند. در واقع، پیاده سازی یک ویژگی جدید نیاز به تغییر همان کلاس عظیم دارد که با کد تولید شده توسط افراد دیگر مغایرت دارد.

مسئله دوم

فرض کنید می خواهیم یک جریان تکستی را به خطوط بشکنیم. برای این کار، الگوریتم های زیاد وجود دارد. Hard-wire کردن تمامی این الگوریتم ها در کلاس های مختلف، به دلایل زیر، مشکل زا خواهد بود:

1-کلاینت هایی که نیاز به شکستن خطوط دارند، اگر شامل کد شکستن خطوط باشند، بسیار پیچیده می شوند و این کلاینت ها را بزرگ تر می کند و نگهداری آن ها سخت تر می شود، مخصوصاً اگر چندین الگوریتم شکستن خطوط را داشته باشند.

2-الگوریتم های مختلف در زمان های مختلف، مناسب خواهند بود. پس اگر نیاز به استفاده از همه آن ها نداریم، نمی خواهیم آن ها را ساپورت کنیم.

3- اضافه کردن الگوریتم های جدید و تغییر الگوریتم های موجود هنگامی که شکننده خطوط بخشی جدایی ناپذیر از کلاینت است، دشوار است.

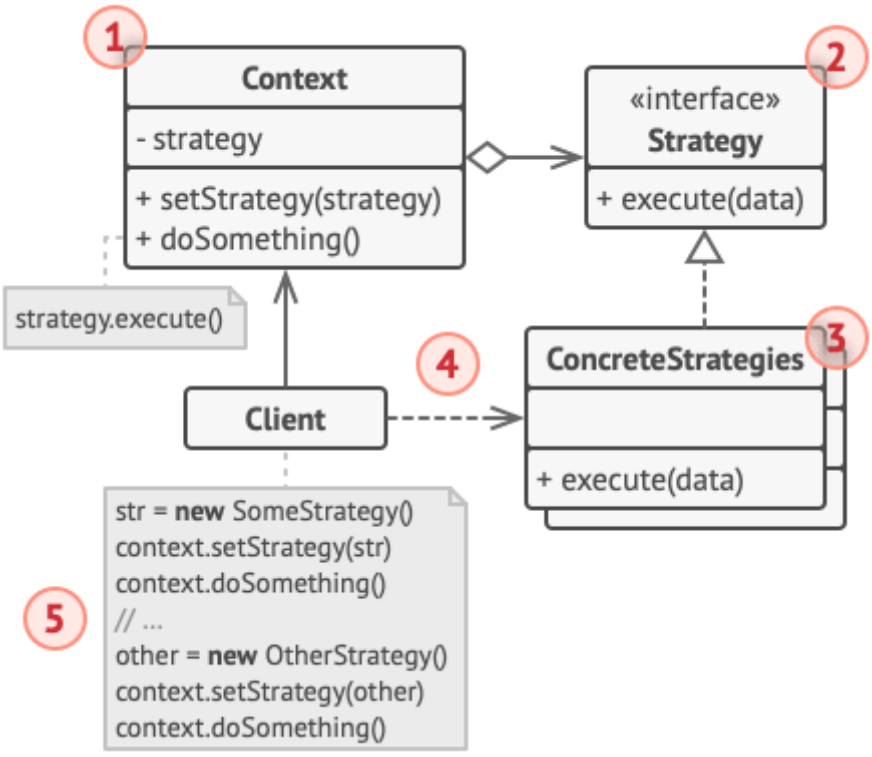
راه حل با الگوی Strategy

الگوی Strategy، پیشنهاد می دهد که که کلاسی را که کار خاصی را به طرق مختلف انجام دهد، داشته باشیم و همه این الگوریتم ها را در کلاس های جداگانه ای بنام استراتژی استخراج کنیم.

کلاس اصلی، Context نامیده می شود و باید دارای فیلدی برای ذخیره سازی ارجاع به یکی از استراتژی ها باشد. Context کار مورد نظر را به جای اینکه خودش اجرا کند، به یکی از استراتژی های لینک شده ارجاع می دهد.

همچنین Context مسئول انتخاب الگوریتم مناسب برای کار مورد نظر نیست. در عوض، کلاینت استراتژی مورد نظر را به Context انتقال می دهد. در واقع، Context چیز زیادی در مورد استراتژی ها نمی داند. Context با تمام استراتژی ها از طریق یک اینترفیس عمومی یکسان کار می کند، که فقط یک متد واحد را برای تریگر کردن الگوریتم encapsulate شده در استراتژی انتخاب شده نشان می دهد. به این ترتیب Context از استراتژی های مشخص مستقل می شود، بنابراین می توانیم الگوریتم های جدیدی اضافه کنیم یا الگوریتم های موجود را بدون تغییر کد Context یا سایر استراتژی ها اصلاح کنیم.

در واقع ساختار کلی این الگو به شکل زیر است:



در ساختار بالا، هر قسمت به شرح زیر است:

1- Context یک رفرنس به یکی از استراتژی های مشخص را نگهداری می کند و فقط از طریق اینترفیس استراتژی با این آبجکت ارتباط برقرار می کند.

2- Strategy interface در همه استراتژی های مشخص مشترک است. این اینترفیس، متدی را تعریف می کند که Context از آن برای اجرای یک استراتژی استفاده می کند.

3- Concrete Strategies تغییرات مختلف الگوریتمی را که Context از آن استفاده می کند، پیاده سازی می کنند.

4- Context هر زمان که نیاز به اجرای الگوریتم داشته باشد، متد اجرایی در آبجکت استراتژی لینک شده را صدا می زدند. Context نمی داند که با چه نوع استراتژی ای کار می کند و یا الگوریتم چگونه اجرا می شود.

5- Client یک آبجکت استراتژی خاص ایجاد می کند و آن را به Context انتقال می دهد. Context ستری را نمایش می دهد که به کلاینت این امکان را می دهد که استراتژی مرتبط با Context را در زمان اجرا جایگزین کنند.

در نهایت، از این الگو در شرایط زیر استفاده می شود:

- وقتی که بسیاری از کلاس های مرتبط، تنها در رفتارشان متفاوت اند. Strategy یک راه برای پیکربندی یک کلاس با یکی از چند رفتار را فراهم می کند.
- وقتی که به تعداد زیادی الگوریتم متفاوت نیاز داریم.
- وقتی که یک الگوریتم از داده هایی استفاده می کند که کلاینت ها نباید از آنها چیزی بدانند. برای جلوگیری از افشای ساختارهای داده پیچیده و خاص الگوریتم، از الگوی استراتژی استفاده می کنیم.
- وقتی که یک کلاس بسیاری از رفتارها را تعریف می کند، و این ها به عنوان چندین جمله شرطی در عملکردهای آن ظاهر می شوند. به جای استفاده از بسیاری از شرط ها، شاخه های شرطی مرتبط را به کلاس استراتژی خود منتقل می کنیم.

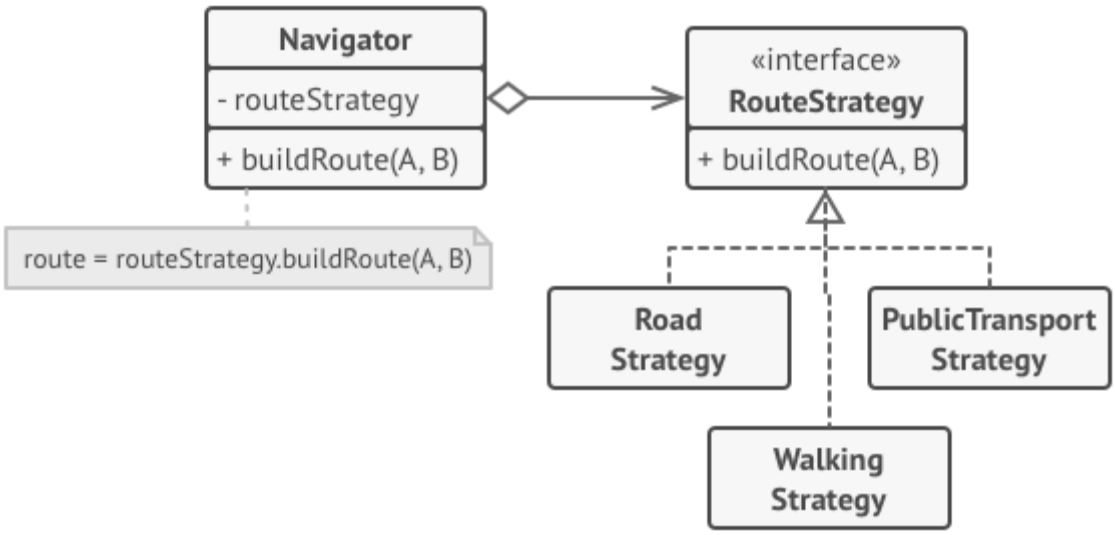
در نهایت راه حل های دو مسئله گفته شده را با الگوی استراتژی بیان می کنیم:

راه حل مسئله اول

در این مسئله، هر الگوریتم مسیریابی را می توان با یک متد buildRoute به کلاس خودش استخراج کرد. این متد، مبدا و مقصد را می پذیرد و مجموعه ای از ایست های بازرسی مسیر را برمی گرداند.

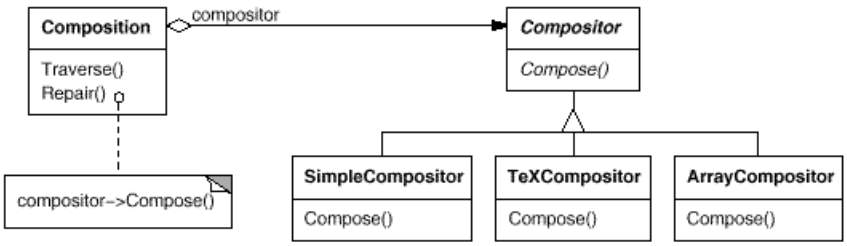
اگرچه که آرگومان ها یکسان هستند، هر کلاس مسیریابی ممکن است مسیری متفاوت ایجاد کند، کلاس اصلی برایش مهم نیست که کدام الگوریتم انتخاب شده است زیرا کار اصلی آن ارائه مجموعه ای از ایست های بازرسی بر روی نقشه است. این کلاس متدی برای تغییر استراتژی مسیریابی فعال دارد، بنابراین سرویس گیرندگان آن مانند دکمه های موجود در رابط کاربری می توانند رفتار مسیریابی فعلی انتخاب شده را با روش دیگری جایگزین کنند.

در نهایت شمای کلی راه حل به صورت زیر می شود:



راه حل مسئله دوم

با تعریف کلاس هایی که الگوریتم های مختلف شکستن خطوط را در خود encapsulate می کنند، مشکل را حل می کنیم که این کلاس ها هر کدام یک Strategy هستند. با گفته های بالا، شمای کلی راه حل به صورت زیر می شود:



پاسخ سوال سوم

بخش اول:

یک نمونه از روش های مورد تاکید agile، TDD یا همان Test Driven Development است که برای اجرای آن نیاز به طراحی انعطاف‌پذیر داریم که در طراحی نرم افزار تاثیر مثبت می گذارد. همچنین TDD ارتباط تنگاتنگی با تست خودکار نرم افزار دارد و ما در این قسمت مجموعه ای از تاثیرات مثبت TDD و تست خودکار را در طراحی نرم افزار بیان می کنیم. در واقع TDD و تست خودکار تاثیرات مثبت زیر را دارند:

- طراحی بهتر برنامه و کیفیت کد بالاتر

هنگام نوشتن آزمون ها ، برنامه نویسان ابتدا باید هدفی را که با قطعه کد به دست می آورند تعریف کنند. توسعه دهندگان تجربه ای را که ارائه می دهد و نحوه مطابقت آن با سایر قطعات کل کد را تخمین می زنند. علاوه بر این ، TDD به شدت با اصل DRY ارتباط دارد. در واقع می گوید برنامه نویسان باید از تکرار کد در قسمت های مختلف سیستم خودداری کنند. این اصل، توسعه دهندگان را به استفاده از کلاس ها و توابع کوچک برای نیازهای خاص دعوت می کند. این به تعریف واضح اشیای سیستم کمک می کند. همچنین وقتی توسعه دهندگان کدی را با استفاده از TDD می نویسند ، کیفیت بالاتر می رودزیرا تمام اشتباهات و خطاهای احتمالی قبلاً در نظر گرفته شده اند. در اینجا ، توسعه دهندگان برای جلوگیری از همه خرابی ها ، تست های لازم و کد را می نویسند. در نتیجه ، کد نتایج بهتری را می دهد. به طور خلاصه ، یک مزیت بزرگ در تمام جنبه های TDD وجود دارد. اصلاح، گسترش، تست و نگهداری یک کد با ساختار زیبا آسان تر است. هر چه کد تمیزتر و ساده تر باشد، تلاش شما برای اصلاح یا به روزرسانی کمتر می شود. و این به طور مستقیم به موفقیت پروژه شما کمک می کند.

- مستندسازی دقیق پروژه

هنگام نوشتن تست برای نیازهای خاص، برنامه نویسان بلافاصله یک specification دقیق و با جزئیات ایجاد می کنند. با ادامه این روند، مستندسازی کامل و دقیقی خواهیم داشت.

- TDD زمان مورد نیاز برای توسعه پروژه را کاهش می دهد

طبق مطالعه ای که توسط اریک الیوت، بنیانگذار Parallel Drive انجام شده، نگهداری از یک برنامه ساخته شده بدون اجرای Test Driven Development ممکن است دو برابر بیشتر از برنامه ایجاد شده با TDD باشد.

در ادامه یک سری دیگر از مزیت ها را بدون توضیح اضافه بیان می کنیم.

- انعطاف بیشتر کد و نگهداری ساده تر
- با TDD شما یک نتیجه قابل اعتماد خواهید گرفت.
- صرفه جویی در هزینه های پروژه در طولانی مدت
- چرخه فیدبک سریع
- حذف خطای انسانی
- کاهش دوباره کاری
- افزایش قابلیت پشتیبانی
- تشخیص و رفع خطای سریع تر

بخش دوم:

استفاده از الگوهای طراحی که باعث می‌شود نسبت به تغییرات احتمالی منعطف‌تر باشیم در چابکی توسعه نرم افزار تاثیر مثبت می گذارد.

دو اصل زیر را که در بیانیه agile آمده اند را با هم ببینیم:

1. Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.

2. Continuous attention to technical excellence and good design enhances agility.

مورد دوم که واضحاً می گوید باید design خوبی داشته باشیم. برای داشتن design خوب، باید از یک design pattern مناسب برای هر مسئله استفاده کنیم. با ارضای این اصل از agile واضحاً اثر مثبتی بر چابکی توسعه نرم افزار خواهیم داشت.

اما مسئله مهم تر، تاثیر الگوی طراحی مناسب بر اصل اول یا همان پذیرا بودن تغییرات در هر مرحله از توسعه است. در واقع ما در هر مرحله از پروژه تغییرات متعددی در نیازمندی ها و چیزهای دیگر خواهیم داشت که نیازمند تغییرات در کد است. حال اگر ما از یک الگوی طراحی مناسب از ابتدای پروژه استفاده کرده باشم، واضحاً اعمال تغییرات ساده تر خواهد بود و ما می توانیم با صرف زمان، انرژی و هزینه بسیار کمتری، تغییرات خواسته

شده را اعمال کنیم. خب این واضحا تاثیر مثبتی در چابکی توسعه نرم افزار است که تصمیم طراحی درست به ما این امکان را می دهد.

پاسخ سوال چهارم

تعریف Opacity و مثالش:

Opacity در مقابل شفافیت است. به این معنی که سخت است که کد را خواند و متوجه شد که چه کاری را انجام می دهد و در واقع کد، مقصود خود را به خوبی بیان نمی کند. در واقع بحث های clean code، clean design و readable code مربوط به این مورد هستند و هدف این است که زمان زیادی برای فهم کد سپری نشود و مقصود کد به اشتباه فهمیده نشود.

یک مثال از این مورد، مسئله Misleading Comments است. مثلاً فرض کنید کدی را نوشته ایم و کامنتی را برای آن گذاشته ایم که توضیح هدف کد است. اما پس از Refactor کردن کد، کامنت های آن را تغییر نداده ایم و این مورد در Refactoring مخصوصاً به وسیله خود editor ها مثل Intellije برای Java بسیار رایج است زیرا آن ها ساختار کامنت را نمی فهمند. خب بعد از این، نفر بعدی که آن کامنت را می خواند، یک هدف دیگری را متوجه می شود که غلط است و دیگر در کد وجود ندارد. پس بهتر است به جای کامنت، کد جوری نوشته شود که کد قابل فهم باشد. در نهایت، این یک نوع Opacity بود.

تعریف Needless Repetition و مثالش:

Needless Repetition که به معنی تکرار اضافی است، یعنی اینکه طراحی، شامل ساختارهای تکراری باشد که می توانستند در قالب یک abstraction واحد، یکپارچه شوند. که وجود این مشکل، هم utiliziation و هم بهره وری را پایین می آورد که در این صورت تغییر دادن کد هم سخت می شود و هزینه های زیادی برای ما به همراه خواهد داشت.

دو نوع تکرار داریم:

1-تکرار نرم افزاری یا همان تکرار کدی که با استفاده از Abstraction قابل حل است.

2-تکرار فرایندی که با استفاده از Automation قابل حل است.

یک مثال ساده برای این مورد این است که فرض کنید یک Web Aplication داریم که در یکی از فرم های آن می خواهیم فیلدی را اضافه کنیم. مثلاً اضافه کردن فیلد ایمیل به فرم ثبت نام. در این صورت برای تغییر این مورد، باید در فرم مورد نظر، ایمیل را اضافه کنیم، در قسمت verification و validation، ایمیل را validate بکنیم، در آبجکت دیتا ترنسفر باید این مورد را اضافه بکنیم، در سرور سمت بک اند باید این مورد خوانده شود و در لاجیکمان باید وارد شود. همچنین در دیتابیس و Dao مربوط به دیتابیس هم باید اضافه شود. این فرایند، شامل تکرار شدن نام ایمیل در تمامی موارد خواهد بود که البته یک سری چیز در هر قسمت متفاوت خواهد بود اما مقدار بسیار زیادی در همه قسمت ها یکسان است که این یک نمونه از تکرار کدی و فرایندی است.

پاسخ سوال پنجم

Factory Class ها غالباً implement می شوند زیرا استفاده از آن ها این اجازه را به پروژه می دهد که از اصول SOLID بهتر و نزدیک تر پیروی کند. به طور خاص، اصول interface segregation و dependency inversion. در واقع Factory Class طبق شاخصی که در ورودی می گیرد، تصمیم می گیرد چه نمونه ای از چه کلاسی را به عنوان خروجی بدهد.

در واقع، Factoryها و رابط ها امکان انعطاف پذیری طولانی مدت را فراهم می کنند. این مورد، امکان را برای طراحی Decouple تر و قابل تست تر، فراهم می کند.

یک سری از فواید استفاده از Factory Classها را ببینیم:

1-استفاده از آن، این امکان را به ما می دهد که کانتینر Inversion of Control را معرفی کنیم که وابستگی به پیاده سازی را حذف می کند. در واقع عدم وابستگی به یک پیاده‌سازی خاص باعث می‌شود تغییرپذیری، قابلیت نگهداری و آزمون‌پذیری افزایش یابد.

2-کد را قابل تست تر می کند زیرا می توانیم اینترفیس ها را mock کنیم و برای خود Factory ها تست بنویسیم.

3- هنگامی که زمان تغییر برنامه فرا می رسد، انعطاف پذیری بیشتری به ما می دهد زیرا می توانیم بدون تغییر کد وابسته، پیاده سازی های جدیدی ایجاد کنیم.

4-Implement کردن Factory بسیار ساده است و حتی اگر در حال حاضر مورد استفاده قرار نگیرد، استفاده از آن پیشنهاد می شود.

5-عدم تکرار کد در مواقعی که تصمیم گیری برای انتخاب یک نمونه از کلاس ها نیاز است.

6-راحت تر می توان آن را گسترش داد.

در نگاه دیگر، Factory فقط برای ایجاد اشیا نیست و پیچیده تر از آن است. الگوی Factory براساس معیارهای خاصی تصمیم می گیرد که چه شی ای باید ایجاد شود، بنابراین اولاً حفظ این logic در یک مکان به جای جستجوی آن در کل سیستم آسان تر است. این منطق ایجاد نیز با Factory قابل توسعه می شود. آسان است که کلاس جدیدی را بدون لمس کد کلاینت، که از این کارخانه استفاده می کند به Factory اضافه کنیم.

در نهایت، استفاده از Factory Class ها تغییرپذیری، قابلیت نگهداری و آزمون‌پذیری را افزایش می دهد و سختی، بی تحرکی، شگنندگی و گرانروی را کاهش می دهد.

پاسخ سوال ششم

در این سوال از الگوهای طراحی Command و Factory Method استفاده خواهیم کرد.

حال به توضیح دقیق تر دلایل انتخاب این الگوهای طراحی و طراحی کلی می پردازیم.

اولاً می دانیم که Command یک الگوی طراحی رفتاری است که یک درخواست را به یک شی مستقل یا stand-alone تبدیل می کند که شامل تمام اطلاعات مربوط به درخواست است. این روش، به ما این امکان را می دهد تا درخواست ها را به عنوان آرگومان متدها منتقل کنیم، اجرای درخواست را به تأخیر بی اندازیم یا آن را در صف قرار بدهیم.

همچنین می دانیم که Factory Method یک الگوی طراحی تکاملی است که اینترفیسی برای ایجاد اشیا در یک سوپرکلاس را فراهم می کند، اما به sub classها اجازه می دهد نوع اشیا ایجاد شده را تغییر دهند.

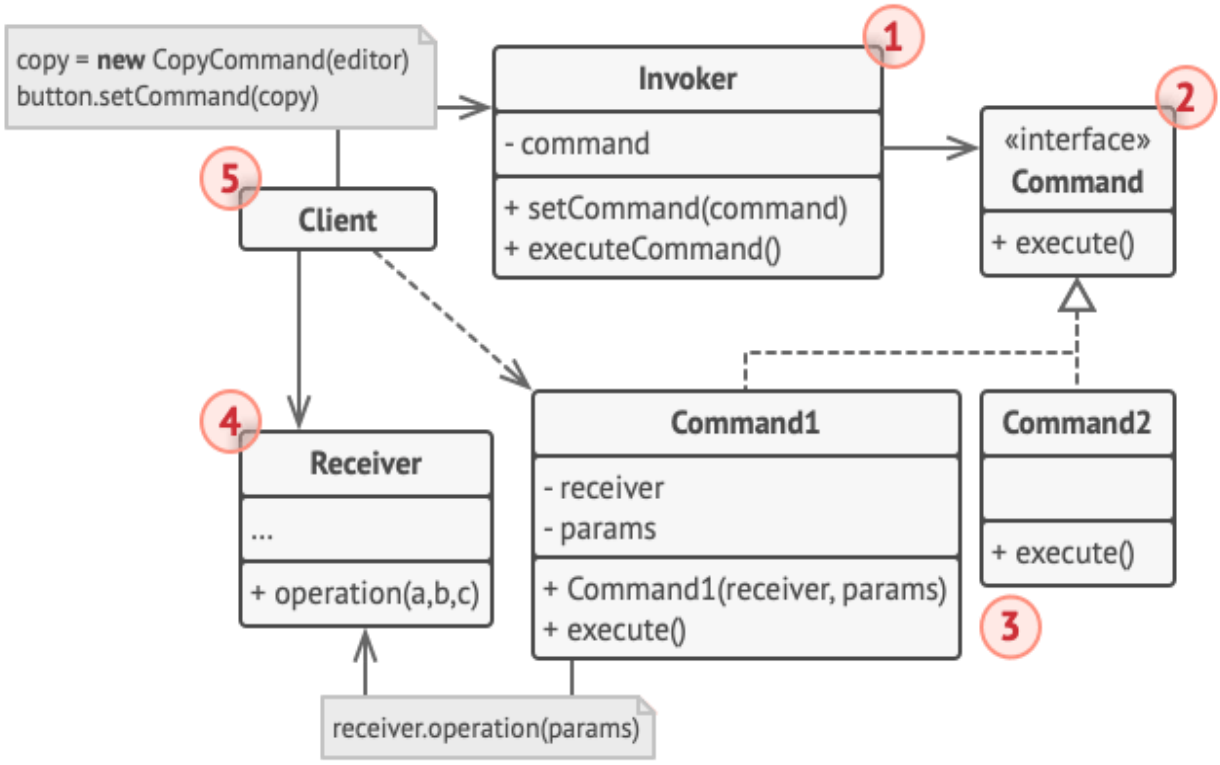
نکته نخست این است که باید به دلایل مختلف مثلاً دغدغه های performance یا Separation of Concerns نباید در همان لحظه که عملیات دریافت می شود، انجام هم بشود. در نتیجه لازم است که یک مرحله از پارس و تشخیص جزئیات را انجام بدهیم، و بعد از آن، هر عملیات یا هر تراکنش را به صورت یک Object به بخشی که باید عملیات را اجرا کند، پاس بدهیم. خب این دقیقاً همان چیزی است که در الگوی

Command به آن می رسیم و به همین دلیل، طراحی خود را برای این قسمت، مطابق الگوی طراحی Command پیش خواهیم برد. در واقع برای اینکه مرحله پارس و جزئیات را داشته باشیم، الگوی Command را به کار می گیریم که در ابتدا، آبجکت های کامند، به عنوان لینکی بین تراکنش ها و در واقع GUI و business logic هستند و در واقع این آبجکت ها، تمامی جزئیات را هندل می کنند.

بدین صورت که یک کلاس Sender خواهیم داشت که درخواست ها را initiate می کند و تنها command مربوطه را تریگر می کند و دیگر درخواست را مستقیماً به گیرنده نمی فرستد. همچنین فرستنده مسئول ساخت آبجکت کامند نیست.

سپس اینترفیس Command را خواهیم داشت که تنها یک متد برای اجرا کردن کامند مربوطه دارد. همچنین یک سری Concrete Commands داریم که ریکوئست های مختلف را که ار چه نوعی هستند implement می کنند.

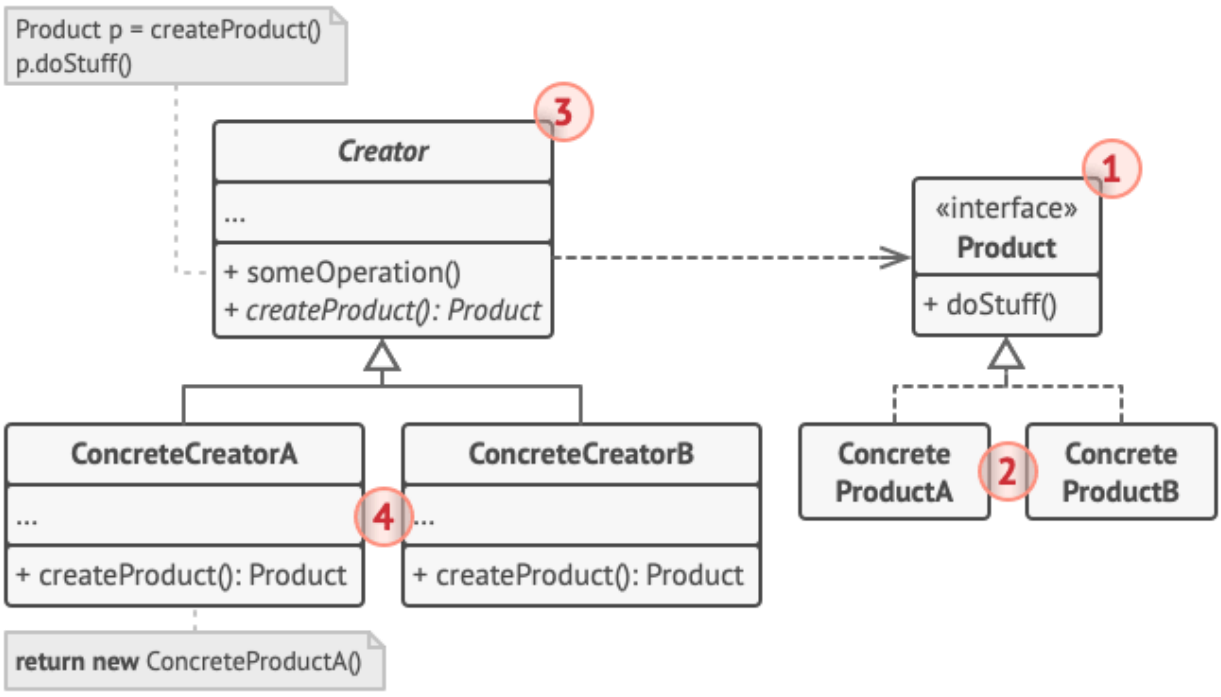
کلاس Receiver هم دارای business logic است و کارها را انجام می دهد و کلاس Client هم که وظیفه ساخت و پیکر بندی آبجکت های Concrete Command را دارد. در واقع کلاینت ما اینجا همان Factory است که در ادامه توضیح می دهیم. پس در نهایت برای هندل کردن قسمت گفته شده، طرحی چیزی شبیه شکل زیر خواهیم داشت:



پس به طور کلی، الگوی کامند برای حل مشکلی بود که همزمان با دریافت عملیات، آن انجام نشود.

حال موضوع اینکه عملیات نباید در لحظه دریافت اجرا بشود را حل کردیم. اما برای طراحی کلی، از الگوی Factory Method بهره می گیریم. زیرا همانطور که گفته شده، تراکنش ها انواع متعددی دارند. همچنین ممکن است در آینده انواع جدیدی از تراکنش ها را بخواهیم اضافه کنیم. خب بدین منظور بهتر است با استفاده از Factory Method پیش برویم که به سادگی منطق عملیاتی هر تراکنش را درون یک کلاس جدا بگذاریم که اگر در آینده بخواهیم یکی از آن ها را تغییر دهیم یا یک نوع تراکنش جدید اضافه کنیم، لازم نباشد تمامی codebase را تغییر دهیم و تنها کلاس مدنظر را تغییر می دهیم یا یک کلاس جدید اضافه می کنیم.

در نهایت با الگوی Factory Method، برای ساخت transaction موردنیاز استفاده می کنیم و برای جلوگیری از اینکه هر بار دستی new کنیم، از Factory استفاده می کنیم و با دادن رشته مربوط به تراکنش ها به آن، پس از جداسازی چند کاراکتر ابتدایی و انجام بررسی ها، ادامه عملیات و انجام منطق اصلی عملیات را به subclass ای می دهد که برای تراکنش موجود می سازد. شکل زیر را برای ساختار Factory Method در نظر بگیریم. طبق این شکل، پس از مشخص شدن نوع تراکنش، ساب کلس مربوطه به وسیله Creator ها ساخته می شود و به وسیله واسط Product، عملیات مربوطه صورت می گیرد:



پس در کل با استفاده از الگوهای Command و Factory Method، خواسته های مسئله را هندل کردیم.

پاسخ سوال هفتم

در این سوال از الگوهای طراحی State استفاده خواهیم کرد. همچنین می توانیم به جای State از Strategy هم استفاده کنیم که شاید کمی بهتر هم باشد. به هر حال من توضیحات را با State می دهم اما Strategy هم تفاوت آنچنانی ندارد.

حال به توضیح دقیق تر دلایل انتخاب این الگوهای طراحی و طراحی کلی می پردازیم.

اولاً می دانیم الگوی State یک الگوی طراحی رفتاری است که به یک آبجکت اجازه می دهد با تغییر استیت داخلی، رفتار خود را تغییر دهد. در نهایت تغییر رفتار طوری بروز پیدا می کند که انگار شی کلاس خود را تغییر داده است.

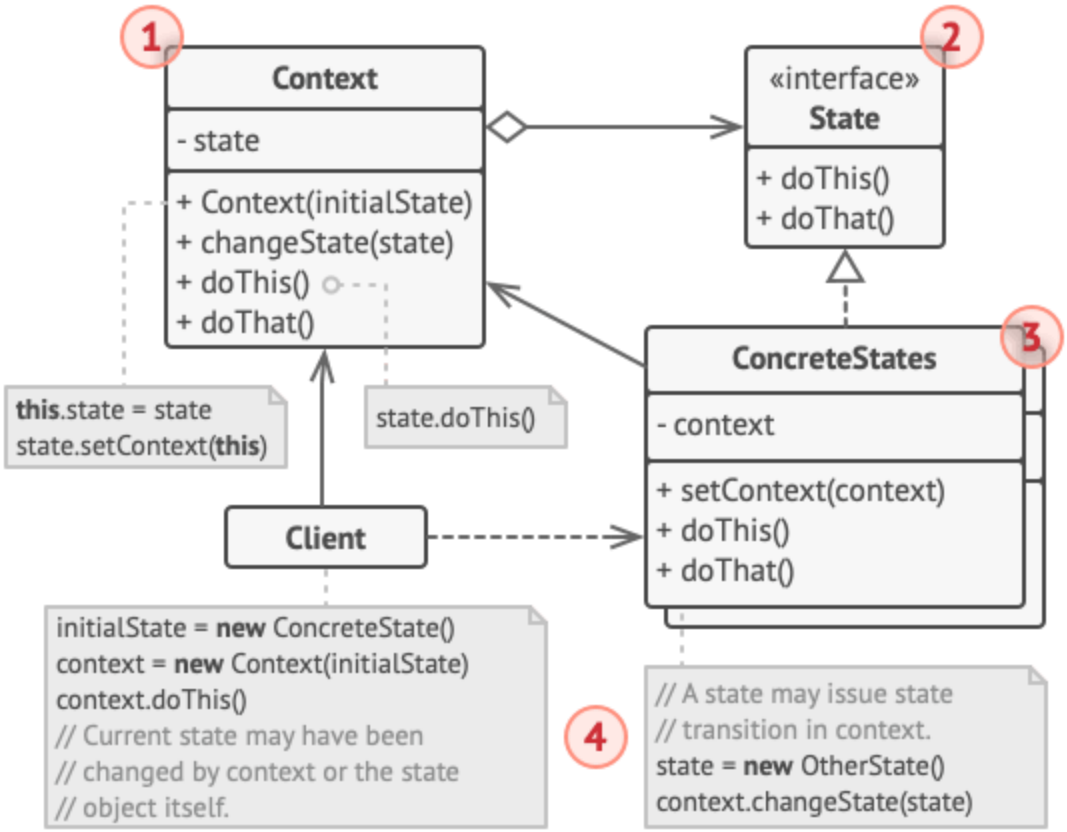
با توضیح بالا، این مسئله به طور کاملاً واضح می گوید که از State برایش استفاده شود! در واقع key ای که گرفته می شود، مشخص کننده این است که state داخلی چگونه تغییر کند و زبان مورد استفاده چه باشد. با عوض شدن key و در واقع زبان، آبجکت موردنظر باید رفتار متفاوتی بروز دهد و در واقع، با تغییر استیت، رفتارش را عوض کند و طبق زبان مد نظر خروجی بدهد. با استفاده از الگوی State از Rigidity هم به طور کامل جلوگیری می شود.

حال طبق الگوی State، کلاس های جدیدی را برای همه حالت های ممکن یک شی ایجاد می کنیم و تمام رفتارهای state-specific را در این کلاس ها استخراج می کنیم. روند کلی کار به صورت زیر است:

در ابتدا یک کلاس Context خواهیم داشت که ایم کلاس Context، دارای یک فیلد است که استیت داخلی اولیه را در آن ذخیره می کنیم. در واقع یک رفرنس به یکی از آبجکت های استیت های Concrete است. همچنین همین کلاس Context با وجود یک رابط با آبجکت های استیت کار می کند. به بیان دیگر، این کلاس، state اپلیکیشن را عوض می کند و مشخص می کند باید از کدام زبان استفاده شود. حال همانطور که گفتیم باید یک اینترفیس داشته باشیم که کلاس Context بتواند با استیت های مختلف کار کند. سپس برای هر زبان یک کلاس خواهیم داشت که همان ConcreteStates ها هستند. این کلاس ها، توابع موجود در اینترفیس state را پیاده سازی می کنند. این روش واضحاً به قدری خوب است که برای اضافه کردن زبان جدید، تنها نیاز داریم یک کلاس ConcreteStates برای آن اضافه کنیم و توابع مورد نظر اینترفیس State را برای آن هم

پیاده سازی کنیم و تغییر دیگری نخواهیم داشت. به بیان دقیق تر، در کلاس Context، توابعی برای صدا زدن توابع گتر در اینترفیس State داریم که در هر کدام از state ها هم در همان تابع، طبق کلید، متن مورد نظر را برای زبان و کلید موردنظر، باز می گردانند. پس اینطور بگوییم که اینترفیس State، یک سری تابع state-specific دارد که این توابع برای تمامی استیت های برنامه کاربرد دارند. حال هر کدام از استیت ها یا همان Concrete States ها، طبق کار خود، implementation مد نظر خود را برای آن متدها، ایجاد می کنند و در واقع برای زبان موردنظر خود، متون را تولید می کنند.

در نهایت ساختار طراحی کلی به شکل زیر خواهد شد:



استفاده از الگوی Strategy هم به همین شکل می شد. فقط در Strategy، آبجکت های مختلف که مربوط به زبان های مختلف هستند، بر خلاف State با هم کاری ندارند. به همین علت، استفاده از Strategy هم شاید انتخاب مناسب تری باشد. اما روند کلی طراحی متفاوت نیست.

پاسخ سوال هشتم

در این سوال به طور خاص از الگو طراحی Decorator استفاده خواهیم کرد. همچنین از یک سری الگوی دیگر هم استفاده خواهیم کرد که در ادامه می گوییم.

حال به توضیح دقیق تر دلایل انتخاب این الگوهای طراحی و طراحی کلی می پردازیم.

گفته شده باید همه حالت ها در هر یک از خانه ها نمایش داده شود و همچنین باید بتوان ویژگی های جدیدی که اضافه می شود را هم نمایش بدهیم. بدین منظور، از الگوی Decorator استفاده می کنیم.

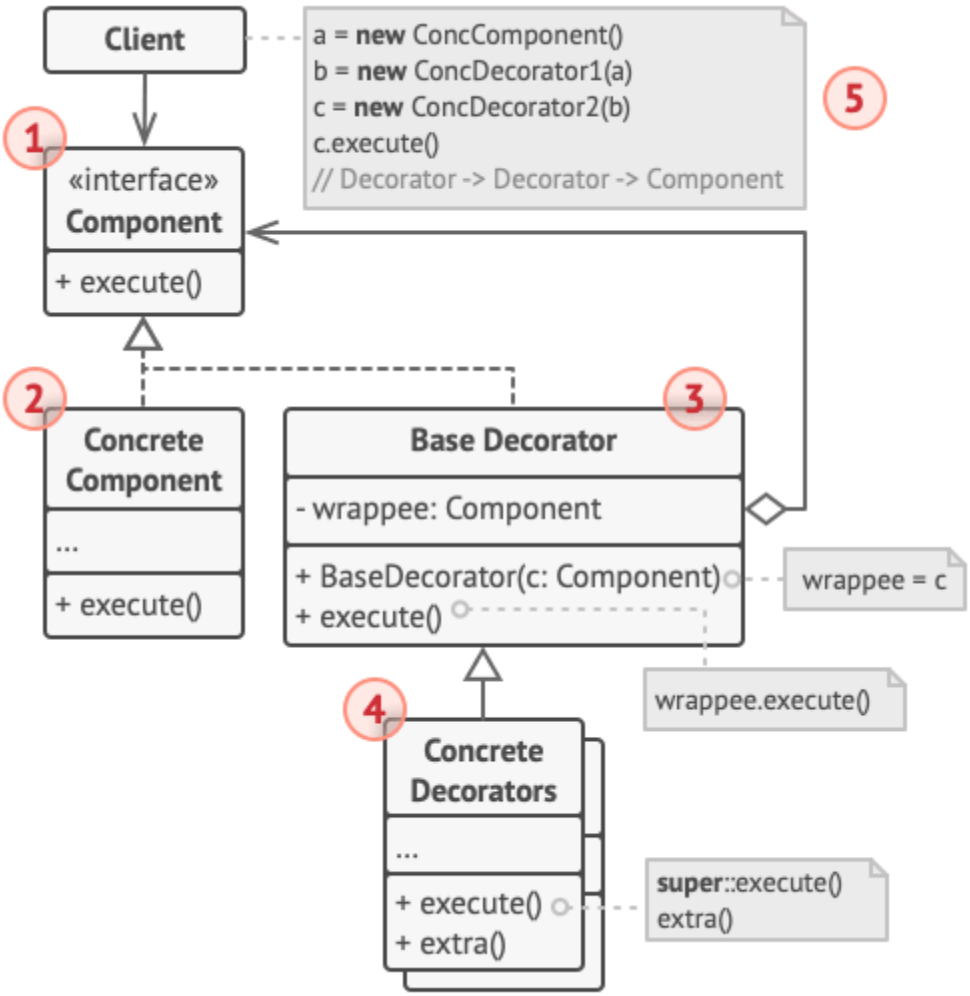
همانطور که می دانیم، Decorator یک الگوی طراحی ساختاری است که به ما این امکان را می دهد که با قرار دادن اشیا در داخل اشیا مخصوص wrapper که حاوی یک سری رفتارها هستند، رفتارهای جدیدی از رفتارهای موجود در wrapper ها را به اشیا متصل کنید.

این دقیقاً همان چیزی است که ما برای طراحی خانه ها می خواهیم. بدین صورت که برای انجام تغییرات در خانه ها و نمایش آن ها، کافی است یک دکوریتور برای هر یک ار آن ها بگذاریم. با این الگو به سادگی می توانیم ویژگی های جدید یا رفتارهای جدیدی را به هر خانه اضافه کنیم و یا کم کنیم.

طراحی ما به مانند همان ساختار الگوی Decorator است. در واقع گفتیم که wrapper خواهیم داشت. wrapper شی ای است که می تواند با برخی از شی های هدف پیوند داده شود. wrapper شامل همان متد های هدف است و تمام درخواست های دریافتی را به آن delegate می کند. با این حال، wrapper ممکن است با انجام کاری قبل یا بعد از اینکه درخواست را به هدف منتقل کرد، نتیجه را تغییر دهد. حال ما یک اینترفیس Component خواهیم داشت که در واقع اینترفیسی برای هر دوی آبجکت های wrapper و wrapped شده می باشد. سپس برای خانه ها، Concrete Component داریم که یک کلاس از آبجکت ها است که wrapped می شوند و در واقع این آبجکت ها همان خانه های بازی هستند که در ابتدا یک رفتار پایه دارند و در ادامه به سادگی می توانند به وسیله decorator ها این رفتار تغییر کنند. حال برای تغییر رفتارها و در واقع هدف اصلی یعنی اضافه و کم کردن ویژگی ها به خانه، نیازمند decorator ها هستیم. در ابتدا یک Base Decorator طراحی می کنیم که یک فیلد برای رفرنس دادن به آبجکت wrapped شده دارد و کار این decorator این است که تمامی کارها را به آبجکت wrapped شده delegate کند. و اما قسمت اصلی طراحی ما که اصلاً دلیل استفاده ما از این طراحی است، Concrete Decorators ها هستند که می توانند رفتارهای جدیدی را به صورت داینامیک به کامپوننت ها اضافه کنند. در واقه همین ها هستند که ویژگی های جدید را به خانه ها برای نمایش اضافه می کنند. این decorator ها، متدهایی که در Base Decorator بودند

را override می کنند و ویژگی های جدیدی که می خواهیم را به هر خانه اضافه می کنند. و در نهایت یک Client داریم که با اینترفیسی که گفتیم با تمامی آبجکت ها کار می کند و کامپوننت ها را در دکوریتور ها، wrap می کند.

در نهایت چیزی شبیه شکل زیر خواهیم داشت.



تا اینجا هر چیزی که گفتیم، برای ظاهر خانه ها بود. برای طراحی بازی ای به مانند Monoploy، از الگوی طراحی State برای ساختار کلی بازی و بورد بازی استفاده می کنیم که به سادگی بتوانیم با تغییر استیت برنامه، رفتار بازی و آبجکت ها را تغییر بدهیم. همچنین ترکیب این الگو با visitor هم می تواند به ایجاد رفتارهای state-dependent کمک کند که البته ضروری هم نیست.

همچنین از الگوی singleton می توانیم برای آیتم های مختلف موجود در بازی استفاده کنیم و با الگوی Strategy یا Template هم می توانیم ساختار هر state را پیاده سازی کنیم.

در این سوال، پیاده سازی بازیکنان مدنظر نبوده، اما استفاده از Factory Method می تواند برای ساخت بازیکنان و اضافه کردن آن ها به لیست بازیکنان، مناسب باشد. برای اینکه GUI نظارت دائمی بر بازی داشته باشد هم از Observer می توان استفاده کرد و برای ایجاد تعامل بین GUI و بازی می توان از الگوی Command استفاده کرد. در واقع این الگو برای فراهم کردن خدمات callback بین مدل و view است.

طراحی کلی را به صورت کلی بالا گفتیم، اما قسمت اصلی این طراحی که سوال از ما خواسته بود، با الگوی Decorator انجام شد.