به نام خدا

تمرین دوم

انجام دهنده:

محمد سعید کودک پور

810196540

دانشکده مهندسی برق و کامپیوتر دانشگاه تهران

پاسخ سوال اول

برای این سوال، از الگوهای طراحی Strategy و Template Method بهره خواهیم گرفت. همچنین در صورت بهره گیری از یکی از روش های خاص موجود در بلاگ مارتین فاولر، می توانیم از Factory Method هم استفاده کنیم.

در ادامه دلایل انتخاب و جزئیات بیشتر را برای هر کدام از تصمیمات و طراحی پیشنهادی خود می گوییم.

اولاً می دانیم Strategy یک الگوی طراحی رفتاری است که به ما این امکان را می دهد که خانواده ای از الگوریتم ها را تعریف کنیم، هر یک از آن ها را در یک کلاس جداگانه قرار دهیم و object های آنها را interchangeable کنیم.

همچنین می دانیم Template Method یک الگوی طراحی رفتاری است که اسکلت یک الگوریتم را در superclass تعریف می کند اما به subclass ها اجازه می دهد بدون تغییر ساختار الگوریتم، استپ های خاصی از الگوریتم را override کنند.

و 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.

همانطور که در تمرین قبل دیدیم، مسئله ما این بود که کدمان را به نحو مناسب Refactor کنیم که بتوانیم به سادگی فرمت های جدیدی مثل HTML یا در آینده فرمت های دیگری مثل JSON یا .... را علاوه بر فرمت تکست برای خروجی اضافه کنیم.

دو روش اصلی ای که ما در تمرین قبل به بررسی آن ها پرداختیم، روش های Parameter-Dispatch و استفاده از آبجکت ها و کلاس ها بود.

در روش استفاده از کلاس ها، ما توابع محاسبه thisAmount و سایر متغیرها را اکسترکت کردیم. همچنین توابع ایجاد خروجی های مختلف را هم جدا کردیم. حال در این روش، مسئله ما تولید انواع خروجی بود. اما هم قسمت تولید انواع خروجی و هم محاسبه متغیرها متفاوت می شد. در واقع ما الگوریتم های تولید هر نوع خروجی را طبق فرمت خروجی و همچنین محاسبات متغیر ها را داشتیم و به بیانی یک خانواده از الگوریتم ها داشتیم که هر کدام را در یک کلاس جدا گذاشته بودیم و طبق ورودی، یکی از آن ها کاربرد پیدا می کرد. خب این دقیقاً استفاده از الگوی طراحی Strategy بود که استفاده شد.

یک نوع طراحی خوب دیگری که قابل انجام است، این است که ما فقط می خواهیم انواع مختلف خروجی را داشته باشیم. پس تمامی قسمت های کد به جز مرحله تولید خروجی یکسان است. در واقع، ما در یک superclass می توانیم بدنه اصلی تولید خروجی را بنویسیم و سپس در یک سری subclass مرتبط با هر نوع خروجی، تغییرات لازم را به وجود بیاوریم تا خروجی به شکل مد نظر تبدیل شود که این یعنی همان استفاده از الگوی طراحی Template Method. اما به نظر من همان Strategy می تواند الگوی مناسب تری باشد.

در روش Parameter-Dispatch، ایده این بود که یک آرگومان format هم به تابع اصلی اضافه می کردیم و سپس توابع ایجاد خروجی برای هر یک از حالت ها را هم استخراج می کردیم. در واقع تابع اصلی طبق پارامتر format تشخیص می داد که چه نوع خروجی ای باید تولید شود و تابع مدنظرش را صدا میزد. ما این روش را با استفاده از کلاس ها ترکیب کرده بودیم و به یک طراحی بهینه رسیده بودیم. در این روش، بعد از اینکه تابع اصلی format را می گرفت، طبق اینکه چه نوع خروجی ای میخواست، کلاس مد نظر را می ساخت. این دقیقاً همان مدلی است که ما از طراحی Factory Method داریم. پس در این روش، از طراحی Factory Method استفاده می شد.

برای هر روش استفاده شده، الگوی به کار رفته را شرح دادیم.

پاسخ سوال دوم

برای پیاده سازی الگوریتم کد هافمن برای فشرده سازی اطلاعات، از الگوی طراحی Composite استفاده می کنیم. در ادامه دلایل انتخاب و طراحی را شرح می دهیم.

اولاً می دانیم Composite یک الگوی طراحی ساختاری است که به ما این امکان را می دهد که آبجکت ها را در ساختارهای درختی ترکیب کنیم و سپس مانند این که این ساختار ها، آبجکت های individual هستند با آن ها کار کنیم.

خب حال به سراغ الگوریتم هافمن برای فشرده سازی اطلاعات برویم. در لینک اول این الگوریتم به طور کامل و واضحی توضیح داده شده است و لینک دوم که کلیات این روش است:

https://www.techiedelight.com/huffman-coding/

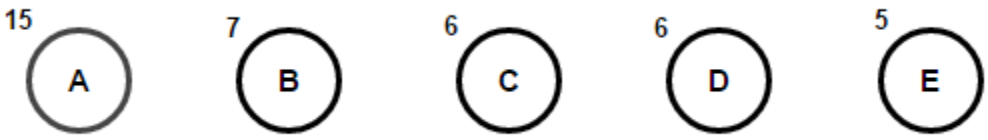
https://en.wikipedia.org/wiki/Huffman_coding

تکنیک هافمن با ساخت یک Binary Tree از نودها کار می کند. یک نود می تواند نود داخلی باشد یا برگ باشد. در ابتدا تمامی نودها، برگ هستند که هر کدام از آن ها، یک کاراکتر را شامل می شوند. همچنین وزن یا فرکانس تکرار هر کاراکتر هم در نود ان موجود است. نودهای داخلی، شامل وزن کاراکتر و لینک به دو نود فرزند خواهند بود. و ساخت درخت اینطور صورت می گیرد که بیت 0، فرزند سمت چپ را نشان می دهد و بیت 1، فرزند سمت راست را نشان می دهد. در نهایت درخت نهایی شامل n تا نود برگ و n-1 تا نود داخلی است. همچنین برای ساخت درخت هافمن، از priority queue استفاده می کنیم که نود با کمترین تعداد تکرار یا کمترین وزن، بیشترین اولویت را دارد. در نهایت الگوریتم کلی به شکل زیر خواهد بود:

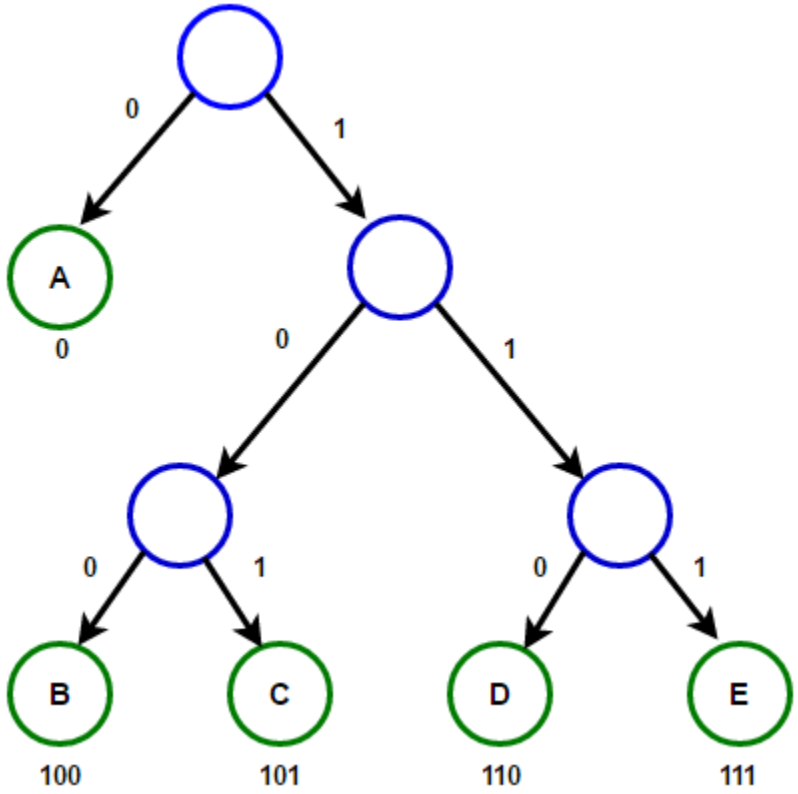
```
1. Create a leaf node for each character and add them to the priority queue.

2. While there is more than one node in the queue:

    • Remove the two nodes of the highest priority (the lowest frequency) from the queue.

    • Create a new internal node with these two nodes as children and a frequency equal to the sum of both
      nodes' frequencies.

    • Add the new node to the priority queue.

3. The remaining node is the root node and the tree is complete.
```

برای یک مثال هم، مثال زیر را ببینیم:

در ابتدا یک متن داریم که کاراکتر های A، B، C، D و E به ترتیب 15، 7، 6، 6 و 5 بار تکرار شده اند. در واقع در ابتدا نودها به شکل زیر خواهند بود:



و پس از اجرای الگوریتم، درخت به صورت زیر می شود:



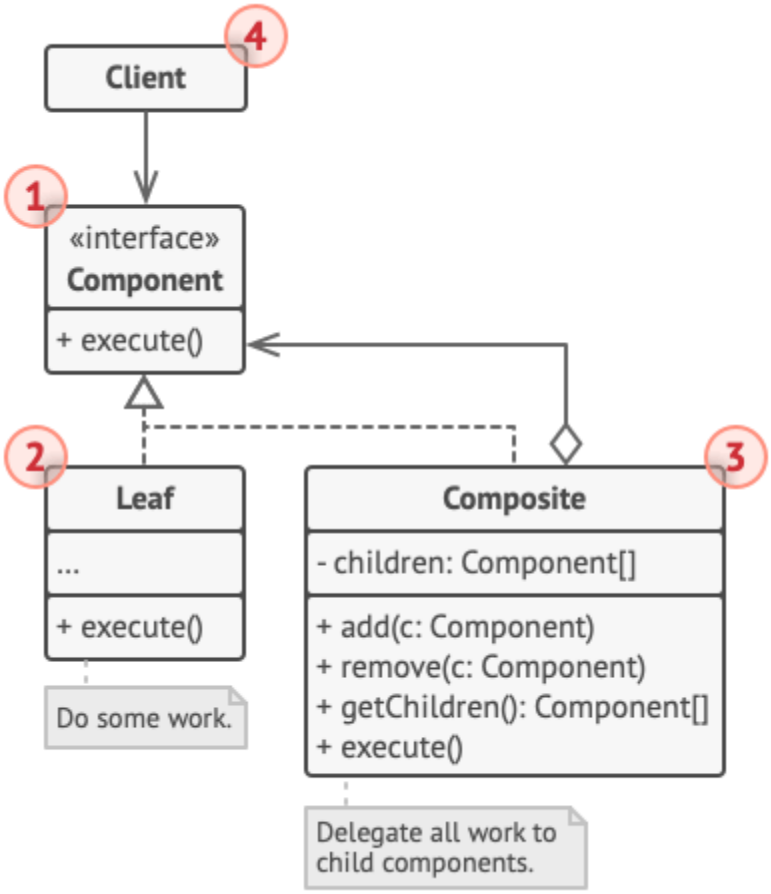
به طور کلی با الگوریتم آشنا شدیم. همانطور که می بینیم، در الگوریتم هافمن، بحث تبدیل مسئله به یک درخت باینری را داریم. در واقع کاراکترهای موجود در متن را به همراه تعداد تکرارشان، به یک نود در درخت مسئله تبدیل می کنیم. خب وقتی حرف از درخت به میان می آید، اولین و مناسب ترین الگوی طراحی ای که به ذهنمان می رسد، الگوی طراحی Composite است. در واقع چیزی که ما در این مسئله می خواهیم، این است که آبجکت ها را که همان کاراکترهای متن هستند، در ساختارهای درختی ترکیب کنیم و سپس مانند این که این ساختار ها، آبجکت های individual هستند با آن ها کار کنیم. خب این یعنی ما باید از Composite استفاده کنیم.

در نهایت برای طراحی نهایی، کلاس های Leaf و Container خواهیم داشت که Leaf ها همان المنت های پایه ای یا همان کاراکترها هستند که هیچ sub element ای ندارند. اما Container یک المنت یا نود را مشخص می کند که sub element هم دارد. در واقع نودهای داخلی را می توان از این نوع گرفت. این Container ها، Concrete Class های فرزندانشان را نمی دانند و با یک اینترفیس خاص که در ادامه توضیح می دهیم با sub element ها کار می کنند. در واقع وقتی یک Container یک درخواست را دریافت می کند،

کار مورد نظر را به sub element ها می دهد و نتایج میانی را پردازش می کند و در نهایت نتیجه نهایی را به کلاینت بر می گرداند.

همچنین همانطور که گفتیم یک اینترفیس Component داریم که عملیات هایی که برای المنت های مختلف درخت یکسان هستند را توصیف می کند و Container به وسیله آن با sub class هایش کار می کند. همچنین کلاینت هم با همین اینترفیس با تمامی element ها کار می کند.

پس ساختار کلی طراحی به شکل زیر خواهد شد:



پاسخ سوال سوم

برای این سوال، به طور عمده از الگوی طراحی Interpreter بهره خواهیم گرفت. همچنین در مراحلی از دو الگوی Composite و Factory Method هم بهره می گیریم.

در ابتدا، می دانیم Interpreter یک الگوی رفتاری است که با گرفتن یک زبان، نمایشی برای گرامر آن به همراه مفسری که از نمایش فوق برای تفسیر جملات در زبان استفاده می کند، تعریف می کند. در واقع، الگوی Interpreter راهی برای ارزیابی گرامر زبان یا expression های آن را فراهم می کند. این الگو شامل ایمپلیمنت کردن یک expression interface است که می تواند یک context خاص را تفسیر کند.

همچنین می دانیم Composite یک الگوی طراحی ساختاری است که به ما این امکان را می دهد که آبجکت ها را در ساختارهای درختی ترکیب کنیم و سپس مانند این که این ساختار ها، آبجکت های individual هستند با آن ها کار کنیم.

و 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.

حال به سراغ طراحی و دلایل انتخاب می رویم:

اولاً ما باید در این مسئله، متغیرهایی داشته باشیم که می توانند مقدار بگیرند. بدین منظور یک کلاس Variable خواهیم داشت که این کلاس، اینترفیس Expression که در ادامه توضیح می دهیم را implement می کند. این کلاس دو فیلد نام و مقدار را دارد که همه چیزی است که ما برای یک متغیر در یک زبان ریاضی لازم داریم. همچنین تابع مشتق گیری هم در این کلاس تعریف می شود که نتیجه طبق متغیر مشتق گیری متفاوت خواهد بود. همچنین برای تابع ارزیابی این کلاس، فقط کافی است مقدار متغیر موردنظر(this.value) را برگردانیم.

حال در این مسئله، باید مشتق گیری از یک متغیر را هم ساپورت کنیم. بدین منظور ما دو اینترفیس برای کار با متغیر ها و عبارات می خواهیم. یک اینترفیس برای مشتق گیری و دیگری برای ارزیابی Expression ها.

در نهایت یک اینترفیس Expression خواهیم داشت که برای عبارات زبان است که در واقع یک پکیج برای هر دوی اینترفیس های مشتق گیری و ارزیابی است و ما دیگر دو اینترفیس جدا برای آن ها نداریم و هر دو در همین Expression هستند و به سادگی توابع مشتق گیری و ارزیابی در کلاس های دیگر پیاده سازی خواهند شد. خب عبارات زبان هستند که باید مشتق گیری یا ارزیابی روی آن ها صورت بگیرد. این یعنی کلاس Expression ما است که مشتق گیری یا ارزیابی را در خود صدا می زند.

هچنین یک کلاس Value هم پیاده سازی می کنیم که فقط فیلد مقدار دارد که Expression را implement کند و به هر variable، مقدار آن را اختصاص می دهد. همچنین تابع مشتق گیری هم در این کلاس تعریف می شود که واضحاً نتیجه صفر خواهد بود زیرا متغیری ندارد. تابع eval هم در این کلاس، مقدار را بر میگرداند.

همچنین برای قابلیت تعریف توابع جدید زمان اجرا، به شکل عبارت لامبدا در نظر می گیریم که یک زیر کلاس برای همان Expression ها است و به سادگی پیاده سازی می شود.

در نهایت برای تعریف operator های مختلف، ما آن ها را به دو مدل operatorهای یگانه و دوگانه تقسیم می کنیم. در واقع، sin و cos را یگانه و جمع، تفریق، ضرب و تقسیم را دوگانه در نظر می گیریم.

برای یگانه ها، یک کلاس unary خواهیم داشت که Expression را implement می کند و توابع ارزیابی و مشتق گیری را به صورت abstract دارد. سپس دو کلاس جدا برای sin و cos ایجاد می کنیم که از کلاس unary ارث بری می کنند و هر دو تابع ارزیابی و مشتق گیری را برای خود پیاده سازی می کنند و توابع فوق را override می کنند. در واقع طبق این که sin هستند یا cos، توابع مد نظر پیاده سازی خواهند شد.

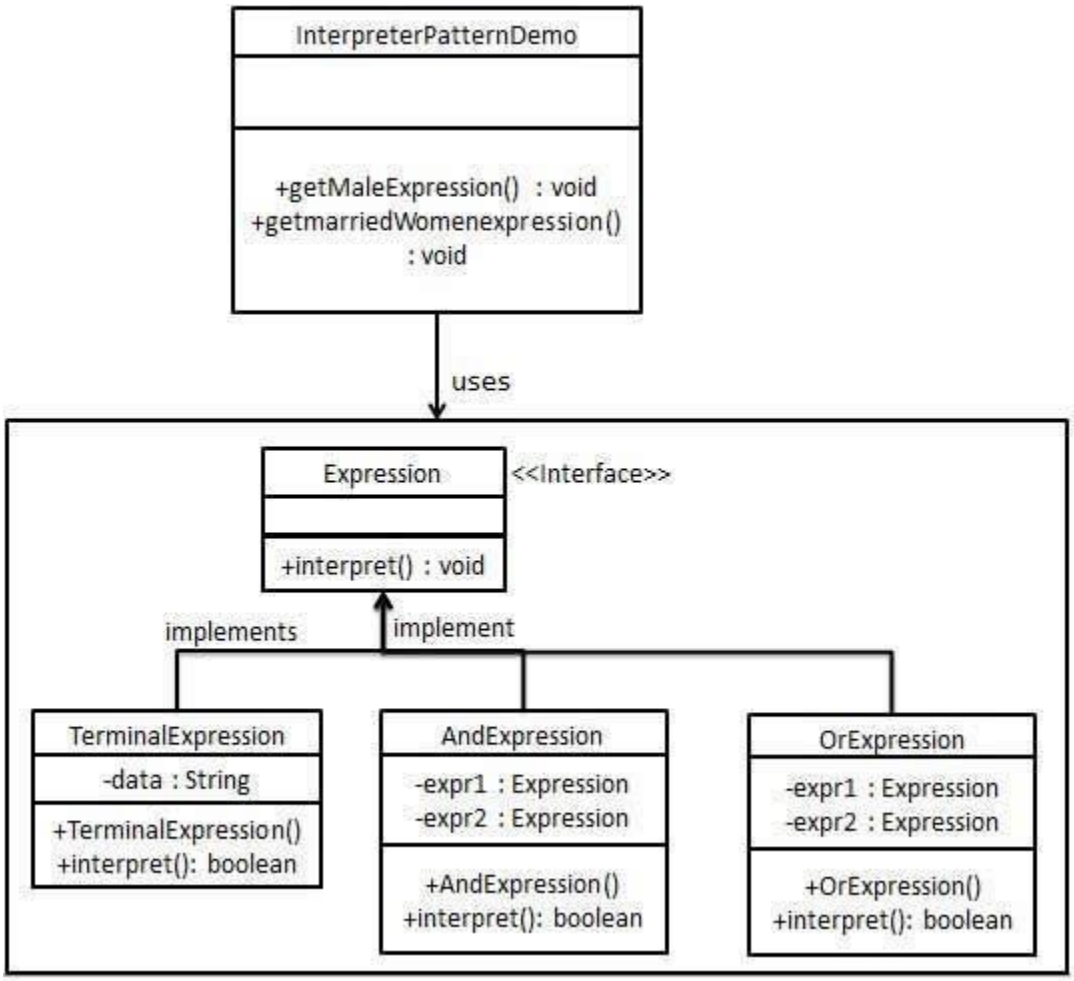
برای دوگانه ها هم دقیقاً به همین روش عمل می کنیم.

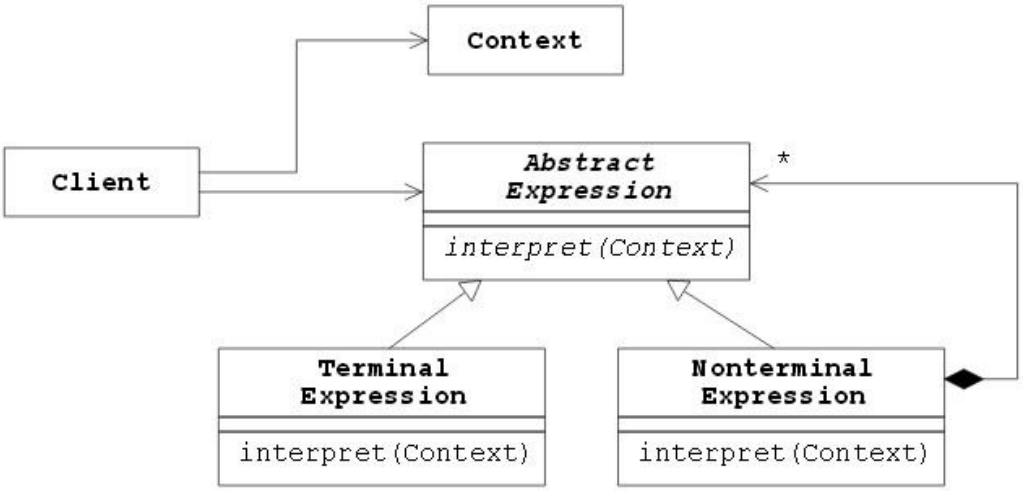
در واقع، یک کلاس dyadic خواهیم داشت که Expression را implement می کند و توابع ارزیابی و مشتق گیری را به صورت abstract دارد. همچنین این کلاس، دو فیلد از نوع Expression دارد که در واقع همان دو operand برای عملیات مورد نظر هستند که در واقع فرزندان سمت چپ و سمت راست operator مورد نظر خواهند بود. سپس چهار کلاس جدا برای جمع، تفریق، ضرب و تقسیم ایجاد می کنیم که از کلاس dyadic ارث بری می کنند و هر دو تابع ارزیابی و مشتق گیری را برای خود پیاده سازی می کنند. این کلاس ها، سپس هر دوی توابع ارزیابی و مشتق گیری را پیاده سازی می کنند و توابع فوق را override می کنند. مثلا اگر بخواهیم برای ضرب بنویسیم، چیزی شبیه زیر می شود:

```
@Override
    public double eval() {
        return this.left.eval() * this.right.eval();
    }

    @Override
    public Expression deriv(Variable var) {
        return new Plus(
                new Multiply(this.left.deriv(var), this.right),
                new Multiply(this.left, this.right.deriv(var))
        );
```

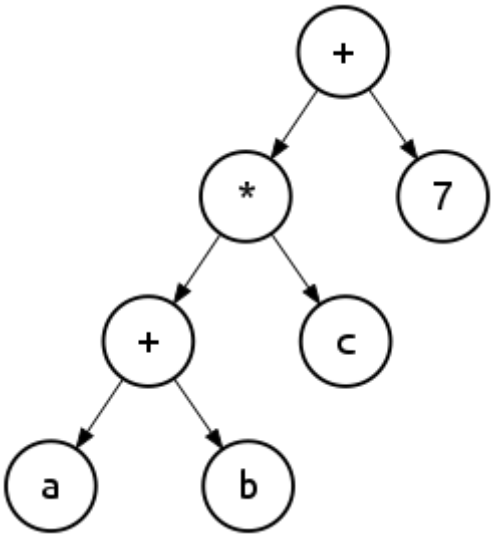
خب کلیت طراحی را با هم دیدیم.

همانطور که واضح است، ما یک گرامر برای زبان فوق تعریف کردیم و expression ها مختلف را طبق آن ساختیم. همانطور که گفتیم Interpreter یک الگوی رفتاری است که با گرفتن یک زبان، نمایشی برای گرامر آن به همراه مفسری که از نمایش فوق برای تفسیر جملات در زبان استفاده می کند، تعریف می کند. در واقع، الگوی Interpreter راهی برای ارزیابی گرامر زبان یا expression های آن را فراهم می کند. این الگو شامل ایمپلیمنت کردن یک expression interface است که می تواند یک context خاص را تفسیر کند. پس ما برای گرامر زبان و ترجمه عبارات از الگوی Interpreter استفاده کردیم. ساختار کلی به صورت زیر است و در دو شکل زیر به خوبی واضح است:





در نهایت همانطور که دیدیم، برای ارزیابی عبارت، یک ساختار درختی داریم که به شکل زیر هستند:



خب همانطور که در سوال قبل دیدیم، الگوی Composite برای طراحی این مدل مناسب است.

همچنین همانطور که می دانیم 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:

- 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.
- کد را قابل تست تر می کند زیرا می توانیم اینترفیس ها را mock کنیم و برای خود Factory ها تست بنویسیم.
- 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.
- Implement کردن Factory بسیار ساده است و حتی اگر در حال حاضر مورد استفاده قرار نگیرد، استفاده از آن پیشنهاد می شود.
- عدم تکرار کد در مواقعی که تصمیم گیری برای انتخاب یک نمونه از کلاس ها نیاز است.
- راحت تر می توان آن را گسترش داد.

طبق موارد بالا، بهتر است به جای دستی new کردن متغیر ها و expression ها از Factory Method استفاده کرد.