یک روش جدید برای مدلسازی ادراکی در برنامه نویسی شیء گرا

ولید الاحمد
دانشگاه شارجه
واحد سیستم های اطلاعاتی مدیریت
walahmad@sharjeh.ac.ae

ترجمه:
کوروش یزدانی
دانشجوی مهندسی صنایع دانشکده فنی دانشگاه تهران
Kourosh@yazdani.id.ir

چکیده

ساختارها و مکانیسم های چند زبانه ای که مدلسازی ادراکی در برنامه نویسی شیء گرا را پشتیبانی می کند، وجود دارد. به عنوان مثال، اصل جایگزینی لیسکف، برنامه نویسی پیمانکاری می یر، ساختار درونی بتا، واسطه های C# و جاوا اسکریپت و جداسازی زیرنوع های سلسله مراتبی از زیرکلاس های سلسله مراتبی تایمر و سادر (Timor and Sather) را می توان نام برد. تمام این مکانیسم ها و ساختارها، ابزارهای مفید و قدرتمندی هستند جهت اطمینان از سازگاری معنایی زیرکلاس ها با سوپرکلاس ها. وقتی به صورت مستقل به کار می روند، به یک سلسله مراتب موروثی صحیح منجر می شوند که به راحتی فهمیده و تمدید می شود. این مقاله مطالعات و مکانیسم های متفاوتی را بحث می کند که با هم در تعامل هستند و بررسی می کند که آیا مدلسازی ادراکی با استفاده از چنین ابزارهایی بدست می آید یا خیر. این مقاله نشان خواهد داد که ارتباط بین این ابزار امکان دارد به مغایرت بیانجامد و امکان دارد از سلسله مراتب موروثی مشروع جلوگیری کند. این مقاله یک روش و راهبرد جدید جهت مدلسازی ادراکی که بر پایه ترکیب این مکانیسم هاست، پیشنهاد می کند. پیشنهاد جدید، مدلسازی ادراکی را در سطح برنامه بهتر پشتیبانی می کند.

1- مقدمه

برنامه نویسی شیءگرا می تواند به عنوان یک راهبرد مدلسازی برای ساخت نرم افزار توصیف گردد. در برنامه نویسی شیء گرا، اجرای یک برنامه یک مدل فیزیکی و شبیه سازی بخشی از دنیای واقعی یا تصوری در نظر گرفته می شود. نرم افزاری که در قالب مفاهیمی که با ذهن انسان ارتباط دارد ساخته می شود، آسان تر ساخته، فهمیده و نگهداری می شود. در برنامه نویسی شیءگرا، مفاهیم سیستم (چیزها، فعالیت ها، اجماع ها و غیره) و پدیده واقعی (مثال های ویژه ی هواپیماها، پروازها، رزرو صندلی ها و غیره) به ترتیب به عنوان کلاس ها و شیء ها در سیستم مدل کامپیوتری نمایش داده می شوند. شکل 1 ارتباط بین برنامه شیءگرا و واقعیتی که آن مدلسازی می کند را مشخص می کند.

در OOP، مفاهیم کلیدی به عنوان کلاس ها، که به صورت سلسله مراتبی سازماندهی می شوند، مدل می شوند که در نتیجه یک کلاس جدید به طور طبیعی به عنوان زیر کلاسی از یک کلاس موجود

تعریف می شود. زیرکلاس، متغیرها و روش ها را از سرکلاس (سوپرکلاس) خود به ارث می برد و ممکن است متغیرها و روش های جدید خودش را اضافه کند و نیز ممکن است روش هایی که به ارث برده پایمال کند. روش های به ارث رسیده به دو دلیل امکان دارد باطل شوند: روش در زیرکلاس، رفتار به ارث برده را باقی نگه دارد اما فعالیت های اضافی را بیافزاید (بازتعریفی برای تکمیل). روش در زیرکلاس، رفتار به ارث برده را با رفتار کاملا جدید جایگزین می کند (بازتعریفی برای اثربخشی).

شکل1- مدل کردن شیءگرای نرم افزار

مدلسازی ادراکی (که غالباً در ادبیاتش تقسیم می شود) در OOP دو معنی دارد: مفاهیم کلیدی در دامنه مشکل که در یک ساختار درختی از طریق مکانیسم موروثی سازماندهی می شود. زیرکلاس ها در سلسله مراتب موروثی باید از لحاظ معنا از سوپرکلاس ها پیروی کنند. این بدین معناست که یک زیرکلاس فقط می تواند جهت دلایل تکمیل شدن، روش های جدیدی را اضافه کند یا با روش های موروثی جایگزین کند. به عبارت دیگر، توقیف متغیرها و روشهای موروثی وپایمال کردن روش های موروثی جهت دلایل اثربخشی مجاز نیست. این چیزها می تواند با استفاده مجدد قانونی کد انجام شود.
دو دسته تفکر از نظر اهداف ارث وجود دارد. به عبارت دیگر، موروثی بودن فقط برای مدلسازی مفاهیم دامنه ی کاربردی می باید به کار رود یا باید برای استفاده ی مجدد کد استفاده شود که در نتیجه کابرد می تواند به طور سریع و البته ضعیف گسترش یابد. این موضوع قابل بحث بوده و دو دسته ی کلی را ایجاد می کند؛ طرفداران مدلسازی موروثی که معتقدند که ساختارهای موروثی ای که مدلسازی ادراکی را منعکس می کنند، برای فهم؛ نگهداری و تعمیم (گسترش) آسان هستند و مهم اینکه نوعی ایمنی تضمین می شود، زیرا واسطه ی زیر کلاس از واسطه ی سوپرکلاس پیروی می کند. در دست دیگر، طرفداران دورنمای استفاده مجدد کد موروثی، از استفاده از ارث به عنوان وسیله ای جهت استفاده مجدد کد دفاع می کنند، زیرا نوشتن یک کلاس که از کد موجود استفاده می کند، به جای نوشتن آن از ابتدا، سریع تر و احتمالاً اثربخش تر به گسترش کاربد کمک خواهد کرد.
تا زمانیکه تمام زبان های برنامه نویسی شیءگرا دومین هدف ارثی بودن را تامین می کنند، فقط تعداد کمی زبان پشتیبانی سطح را برای هدف اول تامین می کنند. این مقاله پشتیبانی زبان برنامه نویسی برای هدف اول (مدلسازی ادراکی) را بررسی و ممیزی می کند و یک روش جدید برای مدلسازی ادراکی در زبان های برنامه نویسی شیءگرا پیشنهاد می کند. به طور اخص، مروری بر پشتیبانی برای مدلسازی ادراکی در بتا، ایفل،جاوا و C# و نیز تایمر و سادر دارد. مکانیسم بتا یک تالاش خوب برای تاکید بر دیدگاه مدلسازی موروثی از طریق ساختار درونی است. ایفل مدلسازی ادراکی را از طریق استفاده ی

مکانیسم تاکید و اثبات، پیش می برد. C# و جاوا مدلسازی ادراکی را از طریق ساختارهای واسطه ای، ترقی می دهند. نهایتاً، در تایمر و سادر دیدگاه مدلسازی ادراکی از طریق جداسازی آن از دیدگاه بازاستفاده ای کد، پشتیبانی می شود. از طرف دیگر، اصل جایگزینی لیسکف (LSP) توسط ساختارهای زبانی پشتیبانی نمی شود و نیازهای اضافی را روی زیرنوع بندی یعنی پیروی، تحمیل می کند. پشتیبانی هایی برای مداسازی ادراکی باید توسط قبول اصل جایگزینی لیسکف تشخیص داده شود.

تعامل مکانیسم ها با یکدیگر و با مکانیسم های موروثی مثل پلیمورفیسم (Polymorphism) برای پشتیبانی مدلسازی ادراکی، چند کاستی و مشکل را آشکار می کند. به طور اخص، تعامل بین LSP و دیگر مکانیسم ها، مشکلات زیرکانه و ماهرانه ای را ناشی می شود و در نتیجه هدف اصلی مدلسازی ادراکی از بین می رود.

بقیه ی مقاله به این صورت سازمان یافته است؛ در بخش 2، مدل سازی ادراکی و دیدگاه های موروثی استفاده ی مجدد کد ارائه می گردند. سپس بخش 3، یک دید کلی از پشتیبانی زبان برای مدلسازی ادراکی می دهد. سپس بخش 4، تعامل بین تکنیک های مختلف مدلسازی ادراکی با هم و با مکانیسم های موروثی را می آزماید. بخش 5 یک روش جدید برای مدلسازی ادراکی معرفی می کند تا کاستی های مکانیسم های موجود را بکاهد. نهایتاً در بخش 6، چند نتیجه گیری و نگاه اجمالی از کار آینده به دنبال می آید.

2- مدلسازی ادراکی در برابر استفاده مجدد کد

جهت روشن کردن تفاوت بین مدلسازی ادراکی و استفاده مجدد کد، یک مثال سلسله مراتبی موروثی برای هر نوع فرض می شود. سلسله مراتب های موروثی با استفاده از UML مدل می شوند. ویژگی های اصلی هریک در ادامه توصیف خواهدشد.

شکل2 – انعکاس سلسله مراتب موروثس (الف) مدلسازی ادراکی (ب) استفاده مجدد کد.

در شکل 2(الف)، در هر سطح پائینی سلسله مراتب موروثی، زیرکلاس های جدید ویژگی ها و رفتار جدیدی را تعریف می کنند. رفتار بازتعریف شده در راهی انجام می شود که به طور معنایی از رفتار موروثی پیروی می کند. برای مثال، حقوق یک مدیر طوری حساب می شود تا حقوق (همان طور که برای یک کارمند حقوق بگیر حساب می شود) به علاوه ی مزایا بازگردانده شود. ویژگی ها یا روش های موروثی مجاز نیستند در سطح زیرکلاس دور انداخته شوند. در طرف دیگر، شکل 2(ب) یک سلسله مراتب موروثی را که رفتار در چند زیرکلاس لزوماً از سوپرکلاس پیروی نمی کند، نشان می دهد. برای

مثال، روش ScaleSide1 برای اشیاء چهارگوش به کار نمی رود و بایستی دور انداخته شود. همچنین، چند ویژگی موروثی در چند سطح لازم نیست؛ زاویه و وجه2 (Side2) برای یک شیء چهارگوش نیاز نیست. علاوه بر این، پیاده سازی موثرتر روش ()area در سطح مربع مستطیل وجود دارد.
مهمترین مزیت سلسله مراتبی بودن کلاس که طبقه بندی ادراکی در راستای کلاس ها را منعکس می کند این است که چنین سلسله مراتب هایی برای فهمیدن، نگهداری، استفاده، گسترش و استدلال آسان تر می باشند. هیچ اختلاف نظری در مورد اهمیت و فایده ی این نوع از سلسله مراتبی بودن کلاس وجود ندارد. در حقیقت، موضوعات مرتبط با گسترش چنین سلسله مراتبی دغدغه ی توسعه دهندگان نرم افزار و محققان جامعه ی شیءگرا بوده و هنوز هم می باشد. به هرحال، مفاهیم دنیای واقعی، نمی بایست لزوماً در قالب رده بندی هایی مثل گیاه شناسی و جانورشناسی توصیف شوند. ساختاربندی ادراکی با اینکه مدلسازی می تواند با استفاده مجدد کد تداخل یابد، منطبق است. برای مثال، مربع ها موارد خاصی از مربع-مستطیل ها هستند.

3- مکانیسم های زبان

تلاش های زیادی جهت بهبود و فرم دهی ارتباط بین یک زیرکلاس و سوپرکلاسش برای ایجاد سلسله مراتبی درست که مدلسازی ادراکی را منعکس کند، وجود داشته اند. تلاش ها شکل های متفاوتی داشته و از تراک های متفاوتی پیروی می کرده اند. در یک طرف، محققانی هستند که از جداسازی زیرنوع بندی از زیرکلاس بندی جهت اجتناب از تداخل هایی که امکان دارد زمانیکه دو مکانیسم یکپارچه می شوند، رخ دهد، حمایت می کنند. در طرف دیگر، برخی تلاش ها که از ابتکارات کلی و خطوط راهنمایی مثل LSP نتیجه می شوند. همچنین، تلاش های دیگری در غالب ساختار ها و مکانیسم های زبانی آمده اند مثل مکانیسم MPC و ساختار درونی بتا. هریک از این روش ها به موضوعات قطعی مربوط به موروثی بودن و سازگاری معنایی بین زیرکلاس ها و سوپرکلاس هایشان اشاره و تاکید دارند. زبان های برنامه نویسی شیءگرای حاضر درجات مختلفی از پشتیبانی را برای مدلسازی ادراکی تامین می کنند. اکنون، تازمانیکه جداسازی زیرنوع بندی از زیرکلاس بندی، BIC، واسطه ها و MPC توسط نقش ها و مکانیسم های زبان ها تحت فشار هستند، LSP نمی تواند تحت فشار باشد. به عبارت دیگر، کامپایلری که تقسیم بندی کلاس ها را رد کند، وجد ندارد زیرا این با LSP قابل قبول نیست. ما معتقدیم که خطوط راهنما و ابتکارات نیاز دارند تحت فشار زبانی باشند که به طور کامل از مکانیسم موروثی سود می برد. به هرحال، هنوز جهت پشتیبانی کامل LSP در سطح زبان کارهایی برای انجام وجود دارند. ما به تعدادی از آنها که در این موضوع به خوبی بیانیه ی یک کار هنری انجام دادیم، اشاره می نمائیم.

3-1- LSP (اصل جایگزینی لیسکف)

در شکل ساده، LSP بیان می کند که یک شیء از یک زیرکلاس می تواند جایگزین یک شیء از یک سوپرکلاس در هر وضعیتی شود. (LSP) یک تعریف از زیرنوع بندی در قالب جانشینی می باشد.

(LSP) می گوید که شما می توانید انواع را با زیرانواع بدون خدشه ای به برنامه هایی که از آنها استفاده می کنند، جایگزین (عوض) کنید. برای نمونه؛

Let q(x) be a property provable about objects x of type T. Then q(y) should *be true for objects y* of *type S where S is a subtype of T.*

به عبارت دیگر، اگر نوعی از T باشد، پس اشیاء نوع T دریک برنامه را می توان با موارد نوع S تعویض کرد، بدون تغییرهریک از ویژگیهای مطلوب آن برنامه. یک نوع یعنی اگر یک نوع S معمول با نوع T سازگار است پس نوعی از T است درحالیکه همیشه برعکس آن صادق نیست .

LSPرا می توان با مثالهای زیادی از زندگی نشان داد در حالیکه نظامها باید از قراردادهای اصلی پیروی کنند. در برنامه نویسی مبتنی بر گزینه ها، این یعنی گزینه هایی از یک نوع مشابه به پیام های مشابه واکنش نشان میدهند. مشتقهای نوع زیر توسط LSP برگشت داده می شوند. یک مربع خواص خود را از دست خواهد داد زمانی که ارزش یکی از اظلاع آن تغییر می کند. اگرچه مربع از نظر ریاضی و مفهومی مستطیل است، با این حال LSP این مشتق را اجازه نمی دهد. یک صف داده ها عناصررا دریک انتها اضافه می کند و عناصر را از انتهای دیگر پاک می کند؛ عناصر را نمی توان از هر دو طرف اضافه یا حذف کرد.

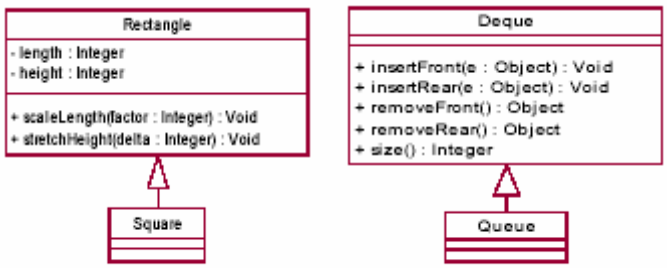

Figure 3: *Class Hierarchies that violate LSP.*

MPC -2-3

برنامه نویسی درقرارداد، یک روش اصلاح نرم افزاری است. این از پیش شرطها، شرایط برای سندسازی تغییر در وضعیت ایجاد شده توسط یک برنامه استفاده می کند. ایده اصلیMCP ، این است که طبقه ها باید غیرمتغیرهایشان را مشخص کنند. روشها باید پیش شرطهای آنها را مشخص کنند و اینکه پس از اجرای آنها چه چیزی باید صحیح باشد. پیش شرطها را می توان ضعیف کرد و غیر متغیرها را می توان تقویت کرد .

MPCمجموعه ای از قوانین زبانی است که می تواند برای بررسی مطابقت قراردادهای فرعی با توافقهای ایجاد شده با طبقه های مافوق مورد استفاده قرار گیرد.. بعضی از مشتقهای طبقه ممکن است با این قوانین تناقض داشته باشند MCP .کمترا ز LPS محدود کننده است. بخش قرارداد آن، بخشی از رفتار را که ما انتظار داریم تغییر نکند نشان می دهد .

BIC -3-3

بتا، طبقه ها و روشها، تابع ها و انواع را در یک مکانیسم چکیده گیری به نام الگو یکپارچه می کند.

آشکار بودن در بتا فقط برای حمایت از دیدگاه مفهومی آشکاری مورد استفاده قرار می گیرد. در بتا، طبقه، طبقه فرعی یا طبقه مافوق عبارتند از الگو، الگوی فرعی و الگوی برتر می باشند .

درواقع، بتا ساختارهای زبانی خاصی دارد که مدلسازی مفهومی را حمایت میکند. یکی از این ساختارها، INNER، که مکانیسمی است که برای بسط عملکردها فراهم می شود .

این برای انعکاس این واقعیت معرفی می شود که تاثیر یک الگوی روشی در یک الگوی برتر همیشه در یک الگوی فرعی لازم است . به عبارت دیگر، بخش عملکرد الگوی برتر با بخش عملکرد الگوی فرعی ترکیب می شود. فهرست 1 مثالی از چگونگی استفاده از داخلی را نشان می دهد.

```
PERSON : (#  name : @ Text; age : @ integer;
  Initialize :< (# n : @ Text; a : @ integer  enter (n, a)
        do n->Name; a->age; INNER
   #);
  Print :<  (#  do  "Person's name is "->PutText; name->PutText;
     "Person's age is "->PutText; age->PutInt; INNER
   #);
#);
EMPLOYEE : PERSON (#  salary : @real;
   Initialize ::<  (# s : @ real       enter s do s->salary;  INNER #);
    Print ::< (# do  "Employee's salary is "->PutText;
                  salary->PutReal; INNER #);
#);
```

Listing 1: *Inner Construct in Beta.*

زمانی که الگوی روش Print مثلاً برای یک کارمند استفاده می شود، بخش عملکرد الگوی پرینت در شخص اولین بار پس از الگوی پرینت در کارمند دنبال می شود. اگر INNER در جایی در الگوی پرینت شخص قرار نداشت، بخش عملکرد پرینت در کارمند هرگز اجرا نمی شود. بدین ترتیب ساختار INNERدر بتا به منظور ترکیب تاثیر الگوی برتر و الگوی فرعی بکار می رود تا سازگاری معنایی بین الگوهای فرعی و الگوهای برتر را تضمین کند چیز دیگری که در بتا مهم است این است که وقتی یک الگو بسط داده می شود. بخش ورود و بخش خروج الگوی فرعی مجموعه ای از آن دو است. بخش ورود برای روش شروع در کارمند s)، a، (nاست. در [22]، گفته شد که INNER نیازمند آگاهی قبلی درباره مشتقهای آینده یک الگو است. که همیشه در عمل امکان پذیر است. به علاوه ، این برعکس کپسول سازی است ، که اصل مهمی در توسعه نرم افزاری است .

3-4- واسطه ها

یک سطح میانی، مجموعه ای از رفتارهای انتزاعی است. در جاوا و#C ، سطحی شبیه به یک طبقه چکیده شبیه است از این نظر که اعضا به کار نمی روند. با سطوح میانی، می توان سلسله مراتب مدلسازی مفهومی و سلسله مراتب اجرا را تفکیک نمود. سطح میانی، پروتکلی ایجاد می کند که طبقه ها ممکن است مانند فهرست 2 اجرا شوند .

```
public interface Shape {
        public void draw();
        public double area();
        public double perimeter();
}
public class Rectangle implements Shape{
        private Point topLeft, bottomRight;
        public void draw() { //code }
        public double area() { //code }
        public double perimeter() { //code }
}
```

Listing 2: *Interface-Class Relationship in Java.*

سطوح میانی را می توان به گونه ای بسط داد که تمام روشهای سطح مبدا آشکار شوند و روشهای اضافه را بتوان مشخص کرد. مثلاً سطحی برای مدلسازی رفتار شکلهای متحرک می تواند شکل سطح میانی را بسط دهد و روشهای جدیدی برای حرکت دادن شکلها نظیر چرخاندن، تفسیر و ... بسط دهد .
طبقه ای در جاوا یا C# می تواند انتخاب کند که هر تعداد سطح میانی را اجرا کند. لازم نیست که طبقه ، رفتار عمومی اش را با استفاد از ساختار سطح میانی تعریف کند. مثلاً طبقه " مربع " را می توان بصورت زیرمجموعه ای از مستطیل درنظر گرفت. سطوح میان در واقع برای تفکیک طبقه های فرعی از انواع فرعی معرفی نمی شوند و آنها بخشی از سلسله مراتب آشکاری نیستند.

3-5- تفکیک سلسله مراتب طبقه فرعی و نوع فرعی

چندین OOPL نظیر تیمور و ساتر، تفکیک سلسله مراتب نوع را از سلسله مراتب طیقه اجرا میکند. آنها معمولا دو ساختار زبانی متفاوت را برای مدلسازی آنها معرفی می کنند. یک نوع تیمور، شبیه به سطح میانی در جاوا یا C# است. این با نوع لغت اصلی مانند فهرست 3 تعریف می شود. این نوع را می توان به روشهای مختلف اجرا کرد. یک اجراء مبتنی بر مجموعه صف داده ها در فهرست 3 تعریف می شود.

```
type Queue {
        maker init(int maxSize);
        op void insertAtBack(ELEMENT e);
        op ELEMENT removeAtFront();
        enq ELEMENT front();
        enq int length();
}
impl ArrayQueue of Queue {
        // proper representation
        maker init(int maxSize){ //code }
        op void insertAtBack(ELEMENT e) { //code }
        op ELEMENT removeAtFront(){ //code }
        enq ELEMENT front(){ //code }
        enq int length() { //code }
}
```

Listing 3: *Type-impl Relationship in Timor.*

اجراهای ممکن دیگری از صف داده ها وجود دارد. نکته مهم در اینجا این است که رفتار اعضاء را نمی توان دوباره مشخص کرد .
بعضی از مشکلات این روش، عبارتند از: آشکاری را پیچیده می کند و حد زیادی را به اجرا کننده طبقه تحمیل می کند تا نوع مناسب را انتخاب کند. همچنین ترکیب انواع مختلف، سلسله مراتب را پیچیده می

کند. به علاوه، بسیاری از زبانهای مدلسازی، نظیر UML بین نوع فرعی و طبقه فرعی تمایز قائل نمی شوند. نقشه برداری مدلهای طراحی، کارساده ای است.

4- تعامل مکانیسمهای مدلسازی مفهومی

پس از بحث درباره ابزارهای حمایتی مختلف برای مدلسازی مفهومی، به این مسئله توجه می کنیم که چگونه این ابزارها با یکدیگر تعامل می کنند. این فقط زمانی است که یک مکانیسم با مکانیسمهای دیگری تعامل می کند که متناقض است و مشکل ایجاد می شود. مواردی وجود دارند که در آنها یک گزینه از یک طبقه فرعی را نمی توان با طبقه برتر جایگزین کرد. این مکانیسم با LSP تناقض دارد چون روشهایی را که در گزینه های طبق برتر قابل اجرا هستند برتری دارد. در واقع همه تکنیکهای مدلسازی مفهومی رابطه مهمی با پلی مورفیزم دارند. پلی مورفیزم آن چیزی است که به مشتریان یک سطح میانی امکان می دهد اهمیت ندهند کدام اجرا در هر نقطه استفاده می شود .

4-1- LSP و MPC

این بخش تعامل LSP با MPC را بررسی می کند. تعریف مجدد این روش دریک طبقه فرعی ممکن است مشکل ساز باشد و آسیب به LSP و MPC ممکن است زمانی رخ دهد که زبان ، قوانین و محدودیتهایی را به این مکانیسم تحمیل نکند. اساساً، روش تعریف مجدد، می تواند بسیاری از چیزهای روش تعریف شده را تغییر دهد، از جمله پیش شرطها.

مثالهای بکار رفته دراینجا برای نشان دادن این تعامل ، درباره یک طبقه عمومی است که ماشینها را نشان می دهد. این بخش بر تعامل LSP با پیش شرطها و غیر متغیرهای طبقه تاکید دارد .

4-1-1- غیر متغیرهای طبقه

یک غیر متغیر طبقه الزام آن است که باید در ایجاد هریک از مثالهای طبقه برآورد شود. اگر گزینه ای ازیک طبقه فرعی، را بتوان با گزینه ای از کلاس برتر درهر محیط جایگزین کرد، پس گزینه های طبقه فرعی باید حداقل غیرمتغیرهای طبقه برتر را برآورده کند. به عبارت دیگر، توافقهای انجام شده در یک طبقه اصلی باید توسط تمام زیرمجموعه های پتانسیل درنظر گرفته شود .

طبقLSP ، غیرمتغیرهای طبقه باید در سطح طبقه فرعی تقویت شوند. هم LSP و هم MPC نیازمند یک چیز هستند. غیرمتغیرهای طبقه نشان می دهند که یک ماشین باید بتواند به سرعت حداقل 100 کیلومتر بر ساعت برسد و ماشین حداقل باید 3 چرخ داشته باشد. حالا مشتقهای ماشین طبقه را مانند فهرست 4 درنظر بگیرید .

طبقه فرعیLigiers ، غیر متغیر طبقه آشکار را با یک غیر متغیر ضعیف تر جاگزین می کند: یک ماشین Ligier نمیتواند به سرعت بیش از 80 کیلومتر برسد. غیر متغیر طبقه درسطح طبقه فرعی به حداکثر سرعت بیشتر از 100 و کمتر از 80 می رسد که با هم تناقض دارند. بهرحال ، اگر به طور اتفاقی، ماشین Ligier باشد، طبقه فرعی Ligier را نمی توان با یک گزینه عمومی تر جایگزین کرد .

طبقه فرعی Harley ، غیرمتغیری را تعریف می کند به گونه ای که بخش مربوط به سرعت را تقویت می کند و بخش مربوط به تعداد چرخها را تضعیف می کند. غیرمتغیر طبقه سراسری منجر به تناقض خواهد شد: این طبقه فرعی LSP یا MPC را ارضاء نمی کند .

```
class Car feature
    -- attributes, constructors, and features
    invariant
        maxSpeed >=100; nrWheels >= 3
end
class Ligiers inherit Car feature
    -- attributes, constructors, and features
    invariant
        maxSpeed <= 80
end
class Harley inherit Car feature
    -- attributes, constructors, and features
    invariant
        maxSpeed >=160; nrWheels = 2
end
```

Listing 4: *Class Invariants in Eiffel.*

4-1-2- پیش شرطها

یک پیش شرط یک مدل، الزامی است که باید انجام شود. این محدودیتهایی را نشان میدهد که در غالب آن، روش به طور مناسب عمل خواهد کرد. به این صورت، یک روش تعریف مجدد، میتواند به تاثیرات روش تعریف شده دست پیدا کند. مثلاً تعریف مجدد یک روش را درنظر بگیرید که باعث می شود با سرعت خاصی حرکت کند. پیش شرط این روش در سطح طبقه عمومی نشان می دهد که سرعت ماشین نباید از 120 km/h فراتر رود و نباید منفی باشد. حالا مشتقهای طبقه زیررا همانگونه که در فهرست 5 نشان داده شده درنظر بگیرید .

- طبقه فرعی ترابانت، پیش شرط آشکاری را تعریف میکند تا آنرا تقویت کند: سرعت از 80 کیلومتر بیشتر نمیشود. پیش شرط در سطح طبقه فرعی منجر به الزام می شود. این LSP را برطرف نمی کند. موردی را درنظر بگیرید که درآن راننده از نوع خاص ماشین آگاه نیست. اگر ماشین ترابانت باشد، سرعت ممکن است امکان پذیر نباشد. قانون MPC درباره پیش شرطها نیز نقض می شود .
- طبقه فرعی کابریولت پیش شرط آشکار را به گونه ای تعریف می کند که یک شرایط اضافه را اضافه می کند. شخصی که از نوع خاص خودرو آگاه نیست، انتظار دارد که درسرعت لازم حرکت کند. حتی اگر سقف ماشین بازباشد. نتایج پیش شرط تعریف شده در یک الزام مرکب. این یعنی اینکه یک کابریولت قادر خواهد بود با سرعت بین 0 و 120 حرکت کند حتی اگر سقف باز باشد.

```
class CAR feature
    drive(speed : Integer) is
    require  speed >= 0; speed <= 120 end
    -- other features and attributes
```

```
end
class TRABANT inherit CAR
    redefine drive end
feature
    drive(speed : Integer) is
    require else speed >= 0; speed <= 80  end
end
class CABRIOLET inherit CAR
    redefine drive end
feature
    drive(speed : Integer) is
    require else
        speed >= 0; speed <= 120
        roofClosed
end
```

Listing 5: *Preconditions in Eiffel.*

4-1-3- بحث

ممکن است پیشنهاد شود که این سلسله مراتبهای طبقه ضعف را نشان می دهند و ممکن است روشهای دیگری برای ساختن آنها وجود داشته باشد.اگرچه مثالهای فوقی ممکن است در نگاه اول نامناسب به نظر برسد، قطعاً نشان می دهند که درزمان تعامل اصول LSP با MPC چه اتفاقی ممکن است اشتباه باشد. میتوان گفت که دلیل تمام مشکلات و نتاقضهای بین طبقه های ماشین و طبقه های فرعی اش به این دلیل بود که واقعیت طبقه به عنوان یک طبقه مملوس تعریف شده است و نه انتزاعی

به هرحال اگر منبع طبقه موجود نباشد چه؟ هریک ازما می خواهد که نرم افزارش محکم، قابل استفاده مجدد و قابل توسعه باشد. صحت و استحکام نرم افزار مهمتر از امکان استفاده مجدد از آن است. بنابراین ما معتقدیم که طبقه های فرعی که قراردادهای طبقه برترشان را نقض می کنند مجاز نیستند. در [23] شرایط این دو دیدگاه مختلف تعریف می شود .

5- یک روش هیبریدی

در این بخش روش جدیدی برای مدلسازی مفهومی پیشنهاد می شود. روش جدید، ویژگیهای زیر را دارد:

- طبقه فرعی و نوع فرعی متحدند: آنچه لازم است یک مکانیسم واحد است که هردو را با هم انجام دهد. ساختار طبقه مانند بسیاری ازOOPL های فعلی، برای دستیابی به نوع فرعی و طبقه فرعی بکار می رود. زبانهای مدلسازی مبتنی بر گزینه نظیر UMLبین این دو ویژگی تمایز قائل نمی شود.این دو دیدگاه مفید هستند و باید مطابق 8 مورد بحث قرارگیرند. مثلاً یک صف داده ها دو انتها می تواند صف داده ها را با اضافه کردن روشهایی بسط داده شود.
- سازگاری معنایی بین طبقه های فرعی و طبقات اصلی. این را میتوان با یک مکانیسم ترکیب نظیر INNER در بتا بدست آورد اما درجهت مخالف. به عبارت دیگر یک روش در یک طبقه فرعی ، باید همیشه از نسخه طبقه اصلی استفاده کند. این شبیه به روابط سازنده در جاوا است که از یک ساختار

برتر استقاده می کند. به منظوراستفاده از تعریف مجدد بازدهی، به مکانیسمهایی نیاز است که نیاز به اجرای مجدد را حذف کنند .

- کاهش تعداد تعریفهای مجدد روش: این را میتوان با مجبورکردن طبقات به اجرا سطح میانی از طریق استفاده از روشهای مجموعه تسهیل نمود. این روشها باید روشهای مجازی باشند تا بتوان آنها را درطبقات فرعی مجدد تعریف کرد. طبقه های فرعی، روابط بین متغیرهای نمونه را بیان میکنند. روشها از این رابطه زمانی استفاده می کنند که درطبقه های فرعی، مجدداً تعریف می شوند. این رابطه میتوند مساوی بودن طول با ارتفاع را درطبقه مربع باشد. این مکانیسم بطور کامل در [21] بیان می شود که در آن از مثالهای کاملی استفاده شده است. سطح میانی همگانی یک طبقه باید با استفاده از ساختار سطح میانی بیان شود. اجرای یک طبقه باید همیشه سطح میانی طبقه را اجرا کند. هرجا که به روشهایی از سطح میانی نیاز نباشد، این روشها باید در سطوح میانی جدید لغت modify تکرار شوند تا نشان دهند که طبقه ای که این سطح میانی را اجرا می کند، باید یک بند برتر غیرخالی داشته باشد .
- حمایت از دیدگاهها: مشخصات روشها دریک سطح میانی، باید بطور رسمی از نظر شرایط قبل/ بعد و غیر متغیرها مشخص شوند به هر حال به منظور تضمین کپسول کردن غیر متغیرها آنها را باید از نظر روشهای عمومی مانند [24] تعریف کرد. ما معتقدیم که اجرای این شرایط دریک OOPL از LSPبهتر حمایت می کند و متعاقباً حمایت بهتری از مدلسازی مفهومی خواهد بود .