عنکبوت، یک اسب نیست

یحیی پورسلطانی

در مهندسی نرم‌افزار و در برنامه‌نویسی با زبان‌های برنامه‌نویسی شیءگرا، ارث‌بری یک قابلیت مهم برای جلوگیری از تکرار کدهای برنامه‌نویسی است و استفاده‌ی درست از آن، می‌تواند به افزایش قابلیت استفاده‌ی مجدد و مراقبت و نگهداری بهتر از کدهای نرم‌افزار کمک کند. عده‌ی زیادی از برنامه‌نویسان، از این قابلیت استفاده نمی‌کنند و برخی، به اشتباه از آن استفاده می‌کنند. استفاده‌ی نادرست از این قابلیت نه تنها باعث راحتی کار نشده، بلکه باعث انتشار خطا در سطوح مختلف برنامه نیز می‌شود. در این نوشتار، در قالب یک مثال به توضیح ارث‌بری پرداخته‌ایم و با اقتباس از این مثال شیرین، که توسط جناب آقای دکتر رامان رامسین (عضو هیئت علمی دانشکده‌ی مهندسی کامپیوتر دانشگاه صنعتی شریف) ارائه شده‌است، سعی کرده‌ایم نحوه‌ی استفاده‌ی درست از این قابلیت را برای شما شرح دهیم.

رفتارها (توابع) انتزاعی

در برنامه‌نویسی، توابع انتزاعی [1] توابعی هستند که بدنه‌ی آن‌ها تعریف نشده و صرفاً نام آن‌ها، نوع خروجی و پارامترهای ورودی آن‌ها تعریف می‌شوند. پیاده‌سازی بدنه‌ی این توابع، به کلاس‌هایی واگذار می‌شود که از کلاس حاوی این توابع، ارث‌بری داشته باشند. توجه کنید که از یک کلاسی که حاوی حداقل یک تابع انتزاعی است، نمی‌توان نمونه‌سازی کرد. (یعنی نمی‌توانید از این کلاس‌ها، آبجکت بسازید.)

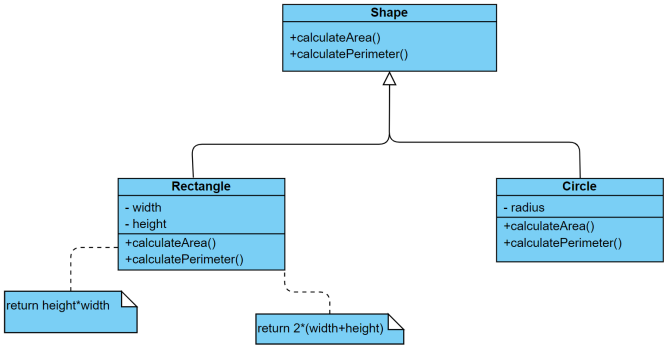
ارث‌بری چیست؟

در برنامه‌نویسی، ارث‌بری [2] کمک می‌کند که بتوانیم رفتارهای یک کلاس را در کلاس دیگری، به ارث برسانیم. در ادامه، با یک مثال خیلی ساده، این مفهوم را برایتان توضیح می‌دهیم.

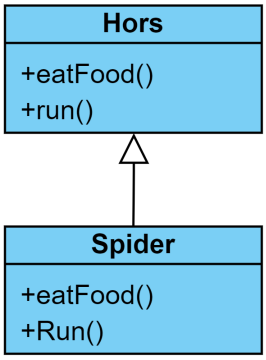
فرض کنید که در برنامه‌ی خود، دو کلاس برای مدل‌سازی دو شیء هندسی دارید: یک کلاس برای مستطیل به نام Rectangle و کلاس دیگری، برای مدل‌سازی دایره، به نام Circle. امکان محاسبه‌ی مساحت و محیط، هم برای مستطیل و هم برای دایره وجود دارد؛ اما روش محاسبه‌ی آن‌ها با هم متفاوت است؛ بنابراین، رفتار (که شما در قالب توابع آن‌ها را پیاده‌سازی می‌کنید) محاسبه‌ی محیط و مساحت، در هر دو کلاس وجود دارد. مشاهده می‌کنیم که دو رفتار تکراری در این دو کلاس وجود دارد و این دو کلاس، از یک جهت خیلی شبیه به هم هستند: هر دو کلاس، یک شکل هندسی را بازنمایی می‌کنند. از سویی، هر شکل هندسی، دارای محیط و مساحت است و دایره و مستطیل نیز، از این قاعده مستثنی نیستند. چه‌قدر خوب می‌شد که بخش‌های مشترک شکل‌های هندسی را از قبل در اختیار این دو کلاس قرار می‌دادیم تا این دو کلاس از نظر ساختار، شباهت بیشتری پیدا می‌کردند و پیاده‌سازی آن‌ها نیز راحت‌تر می‌شد. این کار به لطف امکان ارث‌بری در زبان‌های برنامه‌نویسی ممکن شده‌است. برای این منظور، می‌توانیم یک کلاس به نام Shape بسازیم و دو رفتار مشترک (بخوانید تابع!) محاسبه‌ی محیط و محاسبه‌ی مساحت را به صورت انتزاعی [3] (یعنی بدون پیاده‌سازی بدنه‌ی تابع) تعریف کنیم؛ سپس، برنامه را طوری طراحی کنیم که دو کلاس Rectangle و Shape از این کلاس، ارث‌بری داشته باشند. در این صورت، این دو کلاس نیز دارای رفتارهای محاسبه‌ی محیط و محاسبه‌ی مساحت خواهند بود. البته از آنجایی که در کلاس والد، این رفتارها به صورت انتزاعی تعریف شده‌اند و پیاده‌سازی نشده‌اند، شما باید در هر کدام از کلاس‌های دایره و مستطیل، آن‌ها را به تناسب روش محاسبه پیاده‌سازی کنید. توجه داشته باشید که رفتارهای کلاس والد، می‌توانند در کلاس فرزند پیاده‌سازی شوند و یا در کلاس والد، دارای بدنه باشند که در این صورت کلاس‌های فرزند نیازی به پیاده‌سازی آن نخواهند داشت و یا در صورت تمایل، می‌توانند آن رفتار را بازنویسی [4] کرده و تغییر دهند. بنابراین، ارث‌بری در خیلی از مواقع، باعث جلوگیری از نوشتن کد تکراری نیز می‌شود.

شکل ۱ نشان‌دهنده‌ی مدل این کلاس‌ها در زبان مدل‌سازی UML است. در زبان مدل‌سازی UML کلاس‌ها را با مستطیل نشان می‌دهیم و در بخش بالایی آن نام کلاس، در بخش میانی ویژگی‌ها و مشخصه‌های کلاس و در بخش پایانی آن، رفتارهای کلاس را می‌نویسیم. رابطه‌ی ارث‌بری را با یک فلش، از فرزند به پدر ترسیم می‌کنیم ونوک فلش را به صورت یک مثلث توخالی نشان می‌دهیم. در بسیاری از منابع معتبر، رفتارهای انتزاعی (بدون بدنه‌ی پیاده‌سازی) را به صورت ایتالیک نشان

[1] Abstract Functions
[2] Inheritance
[3] Abstract
[4] Override



شکل ۱: نمودار ساختار کلاسی کلاس Shape و زیرکلاس هایش



شکل ۲: نمودار ساختار کلاسی کلاس Horse و زیرکلاسش

می‌دهند.

رابطه‌ی is a

با دیدن مثال قبل، شاید این سوال برایمان پیش بیاید که: «کلاس‌های Rectangle و Circle دقیقا چه شباهتی با هم دارند و کدام وجه شباهت آن‌ها، باعث شد که بتوانیم رفتارهای مشترک آن‌ها را از طریق کلاس والد Shape به ارث برسانیم؟ آیا صرفا وجود دو رفتار مشترک باعث شده که بتوانیم این ارث‌بری را انجام دهیم؟» پاسخ‌گویی صحیح به این سوالات، ما را به سمت استفاده‌ی درست از ارث‌بری در طراحی نرم‌افزار و برنامه‌نویسی می‌برد.

در جواب، می‌گوییم آنچه که باعث شده تا بتوانیم این رابطه‌ی ارث‌بری را برقرار کنیم، ماهیت مفهومی دو کلاس Rectangle و Circle است. این دو شکل، ذاتاً اشکالی هندسی هستند و در حالت کلی، ویژگی‌های مشترک اشکال هندسی (نظیر داشتن محیط و مساحت)، در خصوص آن‌ها صادق است. بنابراین می‌توان گفت: «مستطیل و یا دایره)، از کلاس اشکال هندسی ارث می‌گیرد، چون یک شکل هندسی است» یا:

Rectangle **is a** Shape.
Circle **is a** Shape.

شما باید بتوانید بین کلاس‌های فرزند و کلاس‌های والد این عبارت را به کار ببرید. به رابطه‌ای که در حالت کلی، بین دو کلاس مستطیل و دایره و کلاس والدشان (کلاس اشکال هندسی) برقرار است، رابطه‌ی is a گفته می‌شود. استفاده از ارث‌بری، در صورتی صحیح است که بین کلاس والد و کلاس‌های فرزند، رابطه‌ی is a برقرار باشد.

میراث مردود!

همان‌طور که دیدید، ویژگی‌ها و رفتارهای کلاس والد، به کلاس‌های فرزند به ارث می‌رسد. شاید پیش خودمان بگوییم حالا که چنین است، چرا نیاییم و برای استفاده‌ی مجدد از رفتارها (شما می‌توانید بگویید توابع!) آن‌ها را به ارث نرسانیم؟ در این صورت، لازم نخواهد بود که برای کلاس‌های فرزند، آن رفتارها را مجددا پیاده‌سازی کنیم.

اجازه دهید تا با مثالی، این کار را بررسی کنیم. فرض کنید که یک کلاس دارید به نام اسب (Horse). این کلاس، رفتارهای یک اسب را مدل‌سازی می‌کند. حال تصور کنید که دو رفتار زیر، در این کلاس تعریف شده‌اند:

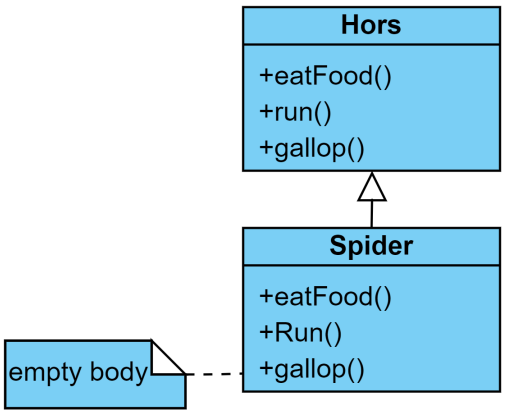
۱. غذا خوردن (eatFood).

۲. راه رفتن (run).

از سویی، کلاس دیگری برای مدل‌سازی یک عنکبوت (Spider) داشته باشیم. همه‌ی ما می‌دانیم که عنکبوت هم مثل اسب، هم راه می‌رود و هم غذا می‌خورد. ممکن است به این فکر بیفتیم که کلاس عنکبوت، از کلاس اسب ارث‌بری داشته باشد. در این صورت، رفتار راه‌رفتن و غذاخوردن، به صورت ضمنی به این کلاس نیز به ارث می‌رسد و دیگر نیازی نیست در این کلاس مجددا آن را تعریف کنیم (یک نمودار ساختار کلاسی، مطابق شکل ۲). بنابراین، به گمان خودمان، از کد استفاده‌ی مجدد [5] کرده‌ایم!

حال، فرض کنید که برای کلاس Horse بخواهیم یک رفتار (در قالب یک تابع) برای مدل‌سازی چهارنعل رفتن (gallop) تعریف کنیم. با توجه به این‌که کلاس Spider نیز از این کلاس ارث‌بری می‌کند، رفتار چهارنعل رفتن به او نیز به ارث می‌رسد! از آنجایی که تصور یک عنکبوتی که چهارنعل راه برود، تا حدودی خنده‌دار (شاید هم ترسناک!)

[5] Reuse



شکل ۳: میراث مردود در مثال اسب و عنکبوت!

است، ناچار خواهیم بود که بدنه‌ی تابع gallop را در کلاس Spider خالی بگذاریم! با این کار، میراث کلاس والد را رد کرده‌ایم.

به این اتفاق، میراث مردود [۶] گفته می‌شود که عملی ناشایست است. چرا که با اعمال هر تغییر در کلاس Horse مجبوریم مطمئن شویم که آیا این تغییر، در کلاس Spider نیز قابل اعمال است یا خیر؟

راهکار چیست؟

برای این مثال، راهکار، فراهم کردن شرایطی است که رابطه‌ی is a برقرار باشد. برای این منظور، بهتر است که یک کلاسی تعریف کنیم که با کلاس اسب و کلاس عنکبوت، رابطه‌ی is a داشته باشد. می‌دانیم که هم اسب و هم عنکبوت، جانور هستند و رفتار مشترک راه‌رفتن و غذاخوردن، در همه‌ی جانوران (با کمی تساهل در خصوص گونه‌ی آن‌ها!) وجود دارد. بنابراین، تعریف یک کلاس به نام Animal با دو تابع انتزاعی eatFood و run و ارث‌بری دو کلاس اسب و عنکبوت از آن، منطقی‌تر به نظر می‌رسد؛ در شکل ۴ این اصلاح را انجام داده‌ایم.

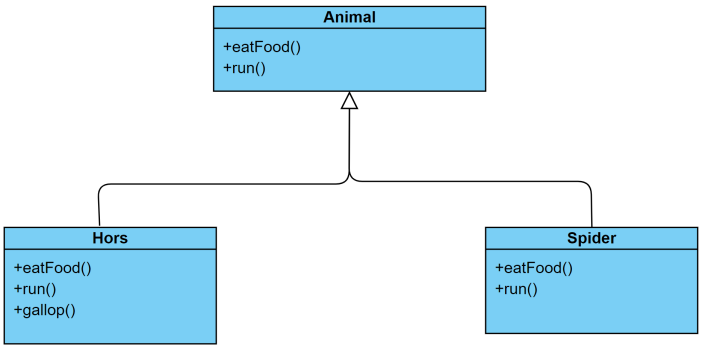
در این صورت، رابطه‌ی is a برقرار خواهد بود و خواهیم داشت:

Spider is a animal.

Hors is a animal.

همچنین، با طراحی شکل ۴ اعمال تغییر در کلاس اسب، باعث انتشار تغییر در کلاس عنکبوت نمی‌شود و با خیال آسوده، می‌توانیم کلاس اسب را گسترش دهیم.

[6] Refused Bequest



شکل ۴: اصلاح ساختار کلاسی اسب و عنکبوت

نتیجه‌ی اخلاقی!

تنها در صورتی از ارث‌بری استفاده کنید که بین کلاس والد و فرزندانش، رابطه‌ی is a وجود داشته باشد و تحت هیچ شرایطی، این رابطه را نقض نکنید. همچنین، ارث‌بری لزوما یک روش مناسب برای استفاده‌ی مجدد از کد نیست! قبل از استفاده از آن، باید از برقراری رابطه‌ی is a مطمئن شد.

تشکر و قدردانی

از استاد ارجمندم، جناب آقای دکتر رامان رامسین، که این مثال گیرا را در کلاسشان طرح کرده‌اند سپاس‌گزارم. برای آشنایی بیشتر با نکات مهم در خصوص طراحی شیءگرا، توصیه می‌کنم حتماً از محتوای درسی ایشان که در وبسایتشان منتشر شده‌است، استفاده کنید. [۷]

[7] http://sharif.edu/~ramsin/