دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

پایان‌نامه کارشناسی ارشد

گرایش نرم‌افزار

معماری‌های تطبیق‌پذیر: روشی برای پویایی رفتار

توسط

سالار مصداقی‌نیا

زیرنظر

دکتر سیدحسن میریان‌حسین آبادی

دی‌ماه ۱۳۸۳

قدردانی

سپاس و ستایش خدای منعم را سزاست که انسان را آفرید، امکان رشد و کمال به او داد، و به او چیزهایی آموخت که نمی‌دانست. اگر لطف و یاریش نبود امید و امکان انجام این کار را نداشتم. همچنین، سلام خدا بر پیامبر گرامی اسلام و خاندان پاکش باد که آن‌ها را اصل و معدن دانش و مربی بشر قرار داد.

در انجام این کار و به‌پایان بردن تحصیل در دورهٔ کارشناسی ارشد مدیون زحمات و یاری افراد بسیاری هستم که قدردانی از تمام آن‌ها در این‌جا ممکن نیست. از مادر و پدرم که امکان تحصیل و رفاه لازم را جهت فراغت از سایر مسائل، برایم به‌وجود آوردند و همواره مشوقم بودند، قدردانی می‌نمایم. از همسرم که در مدت نگارش این پایان‌نامه گرفتاری مرا درک و تحمل کرده و با علاقه، روند کار را دنبال می‌نمود، از صمیم قلب متشکرم. همچنین، از خانوادهٔ او که در مدتی از نگارش پایان‌نامه مهمان آن‌ها بودم، تشکر می‌نمایم.

از استاد راهنمای ارجمندم جناب آقای دکتر میریان که در مقاطع مختلف، راهنمایی‌های حیاتی و ارزشمندی در اختیارم قرار دادند، سپاسگزاری می‌نمایم. همچنین، از دوست عزیزم آقای محمدصادق مکارم که با وجود گرفتاری‌های خود، مدیریت پیاده‌سازی آزمایشی ایده‌های این پایان‌نامه را به‌عهده گرفت و از دوست صمیمی و استاد گرامیم آقای رامتین خسروی به‌دلیل راهمنایی در مورد تعریف صورت مسئلهٔ این پروژه صمیمانه قدردانی می‌نمایم.

از زحمات استاد مشاورم جناب آقای دکتر جلیلی و همچنین، از استاد ممتحن خارجی جناب آقای دکتر شمس که قبول زحمت کرده و در جلسهٔ دفاع حاضر شدند و راهنمایی‌های مفیدی برای بهبود پایان‌نامه‌ارائه دادند، قدردانی می‌نمایم.

از اساتید و مسئولین دانشگاه صنعتی شریف و دانشکدهٔ کامپیوتر سپاسگزارم. در پایان ، از مربیان و اساتیدی که در طول زندگی مطالبی از ایشان آموخته‌ام، تشکر می‌نمایم.

معماری‌های تطبیق‌پذیر: روشی برای پویایی رفتار

چکیده

در بسیاری از محیط‌ها، منطق حرفه مدام در حال تغییر است و سیستم‌های اطلاعاتی مربوط به این محیط‌ها باید به‌سرعت با نیازهای جدید تطبیق داده شود. برخی از این سیستم‌ها نیازمند قابلیت تطبیق منطق حرفه توسط کاربر انسانی (متخصص دامنه) می‌باشند. قابلیت تطبیق در بیشتر موارد یک موضوع مربوط به سطح معماری نرم‌افزار است. درنتیجه، معماری باید این ویژگی کیفی را فراهم کند. در سیستم‌های شیءگرا، منطق حرفه به دو بخش ساختار و رفتار موجودیت‌های (اشیاء) حرفه تقسیم می‌شود. سیستم‌های شیءگرایی که به کاربر اجازهٔ تطبیق موجودیت‌های حرفه را می‌دهند، از سبک معماری مدل قابل تطبیق شیء استفاده می‌کنند. در حال حاضر، ضعف‌هایی در پشتیبانی از تطبیق رفتاری در این سیستم‌ها دیده می‌شود. به‌عبارت‌دیگر، در این سیستم‌ها تطبیق رفتاری محدود به رفتار بسیار ساده می‌باشد. همچنین، امکان پشتیبانی از اعمال جدید وجود ندارد. در این پایان‌نامه روشی برای پشتیبانی از تطبیق ساختاری و رفتاری، توسط معماری سیستم‌های نیازمند تطبیق توسط انسان، ارائه می‌گردد. این روش امکان پیاده‌سازی عملیات عمومی، که برای موجودیت‌های آینده قابل استفاده هستند و همچنین تعریف پویای عملیات ناموجود در سیستم را به‌وجود می‌آورد. این روش برای ایجاد عملیات عمومی شبیه روش خودنگری در زبان‌های برنامه‌نویسی انعکاسی عمل می‌کند و برای تعریف عملیات جدید قابلیت ترکیب عملیات ابتدایی را به متخصص دامنه می‌دهد. در ضمن، با این روش می‌توان عملیات را به‌صورت پویا به موجودیت‌ها منتسب کرد. به‌علاوه، این روش از وراثت ساختار و رفتار موجودیت‌ها پشتیبانی می‌کند.

واژه‌های کلیدی: قابلیت تطبیق نرم‌افزار، معماری قابل تطبیق، معماری انعکاسی، سبک معماری مدل قابل تطبیق شیء، و تطبیق رفتاری

Keywords: *Software Adaptability, Adaptable Architecture, Reflective Architecture, Adaptive Object Model Architecture Style, and Behavioral Adaptability*

فهرست

۱ مقدمه ۷

۱—۱ ویژگی‌های کیفی، تغییر، و استفادهٔ مجدد ۷

۱—۲ نیاز به انعطاف‌پذیری در نرم‌افزار ۹

۱—۳ ساختار پایان‌نامه ۱۰

۲ قابلیت تطبیق نرم‌افزار ۱۱

۲—۱ مروری بر مسائل تغییر و تطبیق در نرم‌افزار ۱۱

۲—۲ محل تغییر ۱۵

۲—۳ شرایط تغییر ۱۶

۲—۴ تعریف مسئله ۱۷

۲—۵ کارهای مرتبط ۱۷

۲—۵—۱ سبک معماری مدل قابل تطبیق شیء ۱۸

۲—۵—۲ سیستم‌های مدیریت پایگاه‌داده‌ها ۱۹

۲—۵—۳ سیستم‌های جریان‌کار قابل تطبیق ۲۰
۲—۵—۴ مدل‌های سطح بالای قابل اجرای نمایشی ۲۲
۲—۵—۵ سیستم‌های انعکاسی ۲۲

۲—۶ جمع‌بندی ۲۵

۳ معماری نرم‌افزار: امکانی برای تحقق قابلیت تطبیق ۲۶

۳—۱ تعریف معماری نرم‌افزار ۲۶

۳—۲ استفادهٔ مجدد از راه‌حل‌های معماری نرم‌افزار ۲۷

۳—۳ ایجاد نرم‌افزار برای خانواده‌ای از محصولات ۳۰
۳—۳—۱ دامنه و مهندسی دامنه ۳۲
۳—۳—۲ زبان‌های مخصوص دامنه ۳۳

۳—۴ سایر روش‌ها ۳۴

۳—۵ جمع‌بندی ۳۴

۴ مبانی ایجاد سیستم‌های قابل تطبیق توسط انسان ۳۶

۴—۱ تحلیل صورت مسئله ۳۶
۴—۱—۱ مفهوم ویژگی و ویژگی قابل تطبیق ۳۷
۴—۱—۲ ارتباط تطبیق و تعریف پویا ۳۸
۴—۱—۳ تعریف ساختار و رفتار ۳۹
۴—۱—۴ نیاز توأم به ساختار و رفتار قابل تطبیق ۴۰
۴—۱—۵ سایر مسائل مربوط به تطبیق ۴۰
۴—۱—۶ یک ابزار زیربنایی برای سیستم‌های قابل تطبیق ۴۱

۴—۲ اهداف یک موتور تطبیق ۴۱

۴—۳ سبک معماری سیستم‌های قابل تطبیق ۴۲

۴—۴ چرخهٔ حیات سیستم‌های قابل تطبیق ۴۳

۴—۵ سلسله‌مراتب‌های تکاملی: مانع قابلیت تطبیق ۴۵

۴—۶ تاکتیک‌ها و تکنیک‌هایی برای دست‌یابی به قابلیت تطبیق ۴۷
۴—۶—۱ تاکتیک داده به‌جای برنامه ۴۷
۴—۶—۲ تاکتیک عمومی کردن برنامه‌ها از طریق پارامتری کردن ۴۷
۴—۶—۳ تاکتیک ترکیب عملیات در زمان اجرا برای ساختن عملیات پیچیده‌تر ۴۸
۴—۶—۴ تکنیک‌های شیءگرا برای تحقق تاکتیک‌ها ۴۸

۴—۷ روش تطبیق در سبک معماری مدل قابل تطبیق شیء ۴۹
۴—۷—۱ سیستم فروشگاه ۵۰
۴—۷—۲ ساختار قابل تطبیق برای سیستم فروشگاه ۵۱
۴—۷—۳ رفتار قابل تطبیق برای سیستم فروشگاه ۵۵

۴—۸ دشواری تطبیق رفتاری در مقایسه با تطبیق ساختاری ۶۱

۴—۹ جمع‌بندی ۶۲

۵ راه‌حلی برای معماری موتورهای تطبیق ۶۳

۵—۱ حالت کلی یک سیستم شیءگرا نیازمند تطبیق ۶۳
۵—۱—۱ انواع سلسله‌مراتب ۶۳
۵—۱—۲ انواع کلاس ۶۵

۵—۱—۳ انواع وراثت ۶۶
۵—۱—۴ انواع مشخصه ۶۶
۵—۱—۵ انواع انتساب ماندگار ۶۷
۵—۱—۶ انواع عمل ۶۸

۵—۲ راه‌حلی عمومی برای تطبیق ساختاری و رفتاری ۶۹
۵—۲—۱ ادغام سلسله‌مراتب‌ها ۷۰
۵—۲—۲ عملیات عمومی ۷۴
۵—۲—۳ ترکیب عملیات ابتدایی ۷۹

۵—۳ بحث ۸۳

۵—۴ جمع‌بندی ۸۵

۶ مورد مطالعه: پیاده‌سازی یک خط تولید قابل تطبیق ۸۶

۶—۱ سیستم خط‌تطبیق برای دامنهٔ سیستم‌های داده‌گرا ۸۶

۶—۲ امکانات تعریف و به‌روزرسانی مدل ۸۷

۶—۳ تفسیر مدل شیء محصول ۹۱

۶—۴ نتیجه‌گیری ۹۲

۷ نتیجه‌گیری و کارهای آینده ۹۵

A واژه‌نامه‌ی فارسی به انگلیسی ۹۸

B واژه‌نامه‌ی انگلیسی به فارسی ۱۰۵

فهرست شکل‌ها

۲—۱ لایه‌بندی سیستم‌های جریان‌کار قابل تطبیق. ۲۱

۲—۲ لایه‌بندی یک سیستم جریان‌کار کاملاً قابل تطبیق. ۲۲

۲—۳ معماری سیستم نیترو (منبع: [۲۱]، با کمی تغییر). ۲۴

۴—۱ شمای کلی سبک معماری سیستم‌های قابل تطبیق ۴۳

۴—۲ چرخهٔ حیات سیستم‌های قابل تطبیق ۴۴

۴—۳ سلسله‌مراتب به‌دست آمده از مدل‌سازی معمول محصولات سیستم فروشگاه ۴۹

۴—۴ به‌کارگیری الگوی شیء قاعده برای تبدیل سلسله‌مراتب به یک مدل قابل تطبیق‌تر . . . ۵۱

۴—۵ به‌کارگیری الگوی مشخصه برای رفع مشکل تطبیق مشخصه‌ها ۵۱

۴—۶ مربع نوع: دو بار استفاده از هریک از الگوهای شیء نوع و مشخصه ۵۲

۴—۷ اضافه کردن وراثت به مربع نوع ۵۳

۴—۸ دو بار به‌کارگیری الگوی مرکب برای مدل‌سازی محصولات ترکیبی ۵۴

۴—۹ مدل عمومی برای انتساب بین محصولات ۵۴

۴—۱۰فاکتورگیری از مدل مشخصه و انتساب ۵۵

۴—۱۱پیاده‌سازی عمل sell با استفاده از الگوی راهبرد ۵۶

۴—۱۲اضافه کردن عمل create با استفاده از الگوی راهبرد ۵۷

۴—۱۳قرار دادن یک نسخه از عمل create در مدل ۵۸

۴—۱۴پیاده‌سازی عمل create با استفاده از یک برنامهٔ عمومی ۵۸

۴—۱۵سلسله‌مراتب شامل محصولات قابل بازگشت ۶۰

۴—۱۶مدل قابل تطبیق سلسله‌مراتب شامل محصولات قابل بازگشت ۶۰

۵—۱ حالت کلی سلسله‌مراتب‌های یک سیستم شیءگرا ۶۴

۵—۲ مدل قابل تطبیق برای شکل ۵—۱، با تبدیل مستقل سلسله‌مراتب‌های تکاملی ۷۱

۵–۳ ادغام مدل قابل تطبیق سلسله‌مراتب‌های تکاملی ۷۳
۵–۴ یک مدل کاملاً قابل تطبیق ۷۴
۵–۵ انتساب اعمال و موجودیت‌ها ۷۷
۵–٦ جداسازی کلّی اعمال و موجودیت‌ها ۷۸
۵–۷ مدل کاملاً قابل تطبیق کاهش‌یافته ۷۹
۵–۸ جداسازی عملیات ابتدایی و عملیات دامنه ۸۰
۵–۹ ترکیب عملیات ابتدایی برای ایجاد دو عمل الف) پس‌دادن محصول و ب) جایزه . . . ۸۲
۵–۱۰ مدلی برای تعریف عملیات پیچیده با ترکیب عملیات ابتدایی ۸۳

٦–۱ نمودار کلاس کامل مثال فروشگاه ۸۸
٦–۲ مدل ماندگاری سیستم خط‌تطبیق ۸۸
٦–۳ فرم تعریف موجودیت Purchase ۸۹
٦–۴ نمودار فعالیت عمل ثبت‌نام مشتری ۹۰
٦–۵ تعریف عمل ثبت‌نام مشتری با استفاده از زبان XML ۹۰
٦–٦ تعریف عمل جایزه با استفاده از زبان XML ۹۱
٦–۷ کلاس‌های سیستم خط‌تطبیق برای تفسیر مدل ۹۲
٦–۸ اجرای عمل ثبت‌نام مشتری ۹۳
٦–۹ فرم ایجاد مشتری ۹۳

فصل ۱

مقدمه

قابلیت تطبیق یکی از ویژگی‌های کیفی نرم‌افزار است که برخی سیستم‌ها به‌دلیل شرایط ویژه‌ای که محیط به آن‌ها تحمیل می‌کند، نیازمند آن هستند. این فصل به معرفی این ویژگی کیفی و علت نیاز به آن، اختصاص دارد. در قسمت ۱—۱، ویژگی‌های کیفی نرم‌افزار و به‌طور خاص ویژگی‌های کیفی [۱] مرتبط با قابلیت تطبیق مرور می‌شود. سپس، در قسمت ۱—۲ دلایل و نمونه‌هایی از نیاز به انعطاف و قابلیت تطبیق در سیستم‌های نرم‌افزاری ارائه می‌گردد. در پایان، قسمت ۱—۳ بقیهٔ فصل‌های پایان‌نامه را معرفی می‌کند.

۱—۱ ویژگی‌های کیفی، تغییر، و استفادهٔ مجدد

در یک طبقه‌بندی سنتی نیازمندی‌های نرم‌افزار به دو دستهٔ کارکردی [۲] و غیرکارکردی [۳] (یا ویژگی‌های کیفی) تقسیم می‌شوند. مؤلفین اهمیت ویژگی‌های کیفی را یادآور شده‌اند و تأکید کرده‌اند که این ویژگی‌ها را نمی‌توان پس از تولید براساس نیازمندی‌های کارکردی، به نرم‌افزار اضافه کرد [۱]. بنابراین، نرم‌افزار در صورتی دارای یک ویژگی کیفی خواهد بود که از ابتدای فرآیند تولید برای دست‌یابی به آن برنامه‌ریزی شود و نمی‌توان انتظار داشت یک ویژگی به‌صورت خودبه‌خود در نرم‌افزار اتفاق بیفتد. به‌عنوان مثال در صورت نیاز به تغییر، استفادهٔ مجدد، یا امنیت باید طراحی برای تغییر [۴]، طراحی برای استفادهٔ مجدد [۵]، یا طراحی برای امنیت [۶] انجام شود.

بس [۷] و دیگران ویژگی‌های کیفی نرم‌افزار را در شش عنوان طبقه‌بندی می‌کنند [۱]: در دسترس

---

[۱] Quality Attributes

[۲] Functional

[۳] Non-Functional

[۴] Design for Change

[۵] Design for Reuse

[۶] Design for Security

[۷] Bass

بودن[۸] (که شامل قابلیت اطمینان نیز می‌شود)، قابلیت تغییر[۹] (که شامل قابلیت حمل[۱۰]، قابلیت استفادهٔ مجدد[۱۱]، و برخی ویژگی‌های دیگر نیز می‌شود)، کارایی[۱۲]، امنیت[۱۳]، قابلیت آزمون[۱۴]، و قابلیت استفاده[۱۵]. در اوایل پیدایش نرم‌افزار، بیشتر تلاش‌ها و توجهات مربوط به کیفیت، به کارایی محدود می‌شد. اما با پیچیده‌تر شدن محصولات نرم‌افزاری و افزایش هزینهٔ تولید نرم‌افزار (گران‌شدن نرم‌افزار نسبت به سخت‌افزار)، ویژگی‌های کیفی محصول که به زمان تولید نرم‌افزار مربوط می‌شوند و در زمان اجرا قابل مشاهده نیستند، مانند قابلیت تغییر نرم‌افزار و سایر ویژگی‌های مرتبط با آن، نیز به‌میزان چشم‌گیری مورد توجه قرار گرفته‌اند. برخی مؤلفین به این تغییر تمرکز در مهندسی نرم‌افزار اشاره کرده‌اند [۲]. می‌توان ادعا کرد از اواسط دههٔ ۸۰ قابلیت تغییر و استفادهٔ مجدد موضوع اصلی بسیاری از تحقیقات دانشگاهی و تلاش‌های صنعتی در زمینهٔ مهندسی نرم‌افزار بوده است. البته این بدین معنا نیست که در سال‌های قبل از دههٔ ۸۰ کاری در این زمینه صورت نگرفته است، بلکه برعکس بسیاری از روش‌های امروز مدیون مفاهیم مطرح شده در آن دوران در مانند واحدمندی[۱۶]، جداسازی دغدغه‌ها[۱۷]، پنهان‌سازی اطلاعات[۱۸]، تولید نرم‌افزار بر مبنای مؤلفه‌ها[۱۹]، و تولید نرم‌افزار برای خانواده‌ای از سیستم‌ها، می‌باشد (برای مثال، واحدمندی و تولید نرم‌افزار برای خانواده‌ای از سیستم‌ها در دهٔ ۷۰، در مقالاتی مثل [۳] و [۴] مورد بحث بوده‌اند.)

حاصل تلاش‌های انجام‌شده در زمینهٔ تولید نرم‌افزار قابل تغییر و قابل استفادهٔ مجدد، پیدایش روش‌هایی همچون شیءگرایی[۲۰]، روش‌های مبتنی بر مؤلفه در تولید نرم‌افزار، چارچوب‌های شیءگرا[۲۱]، خط تولید نرم‌افزار[۲۲] و روش‌های تولیدی[۲۳] مثل برنامه‌نویسی جنبه‌گرا[۲۴] بوده است. همچنین، بسیاری از مباحث معماری نرم‌افزار[۲۵] از جمله برخی الگوهای تحلیل[۲۶]، معماری[۲۷] و طراحی[۲۸] به‌عنوان راه‌حلی برای برآورده کردن نیازمندی‌هایی از این دست، ارائه شده‌اند.

به‌طورکلی، بسیاری از ویژگی‌های غیرقابل مشاهده در زمان اجرا به‌هم وابسته می‌باشند، به‌این‌معنی که

---

[۸] Availability
[۹] Modifiability
[۱۰] Protability
[۱۱] Reusability
[۱۲] Performance
[۱۳] Security
[۱۴] Testability
[۱۵] Usability
[۱۶] Modularity
[۱۷] Separation of Concerns
[۱۸] Information Hiding
[۱۹] Component-Based Software Development
[۲۰] Object Orientation
[۲۱] Object Oriented Frameworks
[۲۲] Software Product-Line
[۲۳] Generative
[۲۴] Aspect-Oriented Programming (AOP)
[۲۵] Software Architecture
[۲۶] Analysis Patterns
[۲۷] Architectural Patterns
[۲۸] Design Patterns

بهبود یکی به بهبود دیگری می‌انجامد. برای نمونه تکنیک‌هایی که ایجاد نرم‌افزار را آسان می‌کند (مثل برنامه‌نویسی تولیدی[۲۹] [۵])، تغییر و استفادهٔ مجدد نرم‌افزار را نیز ساده‌تر می‌کند.

۱—۲ نیاز به انعطاف‌پذیری در نرم‌افزار

ریشهٔ بسیاری از مشکلات دنیای نرم‌افزار این است که هیچ چیز در محیط یک نرم‌افزار ثابت نمی‌ماند. تنها مورد ثابت خود تغییر است. بنابراین، یک سیستم نرم‌افزاری باید انعطاف‌پذیر باشد. برخی مثال‌های نیاز به انعطاف‌پذیری در نرم‌افزار عبارتند از:

- مدیریت نیازمندی‌ها[۳۰]: نیازمندی‌های نرم‌افزار مدام در حال تغییر هستند. بنابراین، در زمان تولید و به‌کارگیری نرم‌افزار بسیاری از مفروضاتی که در مورد وظیفهٔ نرم‌افزار وجود دارد عوض می‌شوند و لازم است این تغییرات در نرم‌افزار منعکس شوند. به‌همین‌دلیل، فرآیندهای جدید تولید نرم‌افزار توجه ویژه‌ای به مدیریت تغییرات[۳۱] دارند (مثلاً یکی از بهترین تجاربی[۳۲] که RUP[۳۳] براساس آن‌ها بنا شده، مدیریت تغیرات است [٦]).
- تطبیق با کاربر[۳۴]: تمام سیستم‌های نرم‌افزاری برای مجموعه‌ای از کاربران مختلف ایجاد می‌شوند و مجموعه‌ای از سرویس‌ها را ارائه می‌دهند. به‌همین دلیل حالت عمومی دارند. ممکن است لازم باشد نرم‌افزار برای کاربران مختلف بر حسب سطح مهارت، علایق، یا روش استفاده از سیستم تطبیق‌داده شود، یا خود تطبیق پیدا کند [۷].
- جبران خطا[۳۵]: گاهی در محیط نرم‌افزار اتفاقاتی می‌افتد که نرم‌افزار باید نسبت به آن عکس‌العمل نشان دهد. مثلاً ممکن است در یک سیستم ناگهان یک قطعهٔ سخت‌افزاری دچار اشکال شود و سیستم نرم‌افزاری مجبور شود سخت‌افزار پشتیبان را راه‌اندازی و استفاده کند [۸].
- باز بودن[۳٦] و قابلیت اتصال مؤلفه‌های جدید[۳۷]: یک ویژگی مناسب در سیستم‌های مبتنی بر مؤلفه این است که امکان اتصال مؤلفه‌های جدید به سیستم، بدون تغییر سایر قسمت‌های سیستم [۹] و در مواردی بدون نیاز به شروع مجدد سیستم [۸] وجود داشته باشد.

ادامهٔ بحث به نوع خاصی از انعطاف که در برخی از سیستم‌های نرم‌افزاری مورد نیاز است، معطوف می‌شود.

[۲۹] Generative Programming

[۳۰] Requirements Management

[۳۱] Change Management

[۳۲] Best Practices

[۳۳] Rational Unified Process: فرآیندی است برای تولید نرم‌افزار به‌روش شیءگرا که در اواخر دههٔ ۹۰ توسط شرکت رَشنال ارائه شده است.

[۳۴] User Adaptation

[۳۵] Failure Recovery

[۳٦] Openness

[۳۷] Component Plugability

نیاز به قابلیت تطبیق در نرم‌افزار

از جمله مشکلاتی که تولیدکنندگان نرم‌افزار، با وجود پیشرفت‌های قابل توجهی که در روش‌های تولید نرم‌افزارهای قابل تغییر صورت گرفته، با آن مواجه هستند، تولید نرم‌افزار برای دامنه‌هایی[38] است که با تناوب زیادی نیازمند تغییرات هستند. برای مثال، می‌توان به دامنه‌هایی همچون صنعت بیمه و خدمات درمانی اشاره کرد که بصورت مداوم نیازمند تطبیق با شرایط جدید (در اثر تغییر در قوانین حرفهٔ[39] در شرکت‌های بیمه‌ای یا تغییر شکل اطلاعات ثبت شده در مورد مشاهدات پزشکی) در سیستم نرم‌افزاری می‌باشند [۱۱، ۱۰]. در این قبیل سیستم‌ها منطق حرفه مدام در حال تغییر است و توقف کار سیستم به‌منظور ایجاد تغییرات در متن برنامه مشکلات زیر را به‌وجود می‌آورد:

۱) سرویس‌دهی سیستم با تناوب بالایی مختل می‌گردد که این قابل تحمل نیست.

۲) ازآن‌جاکه این تغییرات نیازمند متخصصین تولید نرم‌افزار می‌باشند، هزینهٔ بالایی دارند.

۳) همچنین در بعضی از این سیستم‌ها موجودیت‌های خیلی زیادی وجود دارند که شناسایی آن‌ها در مرحلهٔ توصیف نیازمندی‌ها بسیار دشوار است. گذشته از این، در صورتی‌که تولیدکنندگان بخواهند هر یک از این موجودیت‌ها را با یک کلاس مدل کنند، عملی نخواهد بود. به‌عنوان مثال در یکی از این سیستم‌ها تعداد کلاس‌های مورد نیاز، ۱۰۰۰۰ عدد تخمین زده شده است [۱۰].

چنین محیط‌هایی نیازمند سیستم‌هایی هستند که از تغییرات پویا در منطق حرفه[40] پشتیبانی کنند. اگر روش معمول ایجاد نرم‌افزارهای شیء گرا در مورد این سیستم‌ها به‌کار گرفته شود، موجودیت‌هایی که منطق حرفه را مدل‌سازی می‌کنند، یک یا چند سلسله‌مراتب بزرگ را به‌وجود می‌آورند که به‌دلایلی که در قسمت ۴—۵ ارائه می‌گردد، امکان تغییرات پویا ندارد.

موضوع این پایان‌نامه در راستای بهبود روش‌های تولید این‌گونه سیستم‌ها تعریف شده است. در صورتی‌که نیازمندی پویایی در چنین سیستمی برآورده شود، به آن سیستم، قابل تطبیق گفته می‌شود. البته انواع دیگری از قابلیت تطبیق نیز وجود دارند که در قسمت ۲—۱ معرفی می‌شوند، ولی در این پایان‌نامه مورد نظر نیستند. در قسمت ۲—۴، به‌طور دقیق توضیح داده می‌شود که این تحقیق قرار است چه مشکلی را حل کند.

۱—۳ ساختار پایان‌نامه

در فصل ۲ مسئلهٔ قابلیت تطبیق به‌صورت دقیق‌تری بررسی می‌شود. فصل ۳ به‌مرور مباحث معماری نرم‌افزار می‌پردازد. فصل ۴ مبانی ایجاد سیستم‌های قابل تطبیق توسط انسان را بیان می‌کند. در فصل ۵ راه‌حل ارائه شده در این پایان‌نامه برای تطبیق ساختاری و رفتاری مطرح می‌گردد. فصل ٦ پیاده‌سازی یک سیستم نمونه را، به‌نام خط‌تطبیق، با استفاده از راه‌حل ارائه‌شده، نشان می‌دهد. فصل ۷ به نتیجه‌گیری و کارهای آینده اختصاص دارد.

---

[38] Domains

[39] Business Rules

[40] Business Logic

فصل ۲

قابلیت تطبیق نرم‌افزار

عبارت قابلیت تطبیق توسط مؤلفین مختلف برای رساندن معانی مختلفی به‌کار گرفته شده است. به‌همین‌دلیل، لازم است به‌دقت مشخص شود که در این پایان‌نامه این عبارت، چه معنایی دارد. در این فصل علاوه بر روشن شدن مفهوم قابلیت تطبیق، ابعاد مختلف این مفهوم، مورد بررسی قرار می‌گیرد. قسمت ۲—۱ یک طبقه‌بندی از تغییرات نرم‌افزار ارائه می‌دهد. قسمت ۲—۲ در مورد محل تغییر بحث می‌کند. در قسمت ۲—۳ شرایط تغییر نرم‌افزار مورد بحث قرار می‌گیرد. قسمت ۲—۴ صورت مسئلهٔ این پایان‌نامه را به‌دقت تعریف می‌کند. و در پایان، قسمت ۲—۵ به بررسی کارهای انجام شده در این زمینه می‌پردازد. قسمت ۲—٦ جمع‌بندی مطالب این فصل را ارائه می‌دهد. قسمت این فصل را جمع‌بندی می‌کند.

۲—۱ مروری بر مسائل تغییر و تطبیق در نرم‌افزار

هر یک از اصطلاحات قابل تطبیق[۱] یا تطبیق‌یاب[۲] در متون به‌معانی مختلفی استفاده شده‌اند و مفهوم نزدیکی دارند. برای توضیح این مفاهیم ابتدا لازم است طبقه‌بندی مناسبی برای تغییرات نرم‌افزار ارائه گردد. عبارت قابل تغییر[۳] مفهوم گسترده‌ای دارد و استفاده از آن در مورد یک سیستم، با ابهاماتی همراه است. هدف از این قسمت تشریح این مفهوم و مشخص کردن نوع خاصی از آن به‌عنوان قابلیت تطبیق است. تغییرات نرم‌افزار را می‌توان از نظر نحوهٔ اعمال تغییر به دو گروه کلی تقسیم کرد:

۱) **ایستا**[۴]: اگر نتوان تغییرات را در زمان اجرا به سیستم اعمال کرد، تغییرات ایستا هستند. اینگونه تغییرات در نرم‌افزار نیازمند توقف اجرای سیستم، اصلاح متن برنامه، ترجمهٔ[۵] مجدد، آزمون مجدد، استقرار[٦] مجدد، و راه‌اندازی مجدد[۷] سیستم می‌باشند. روش‌های تغییرات ایستا را بر حسب ابزار مورد استفاده

[۱] Adaptable

[۲] Adaptive

[۳] Modifiable یا Changeable

[۴] Static

[۵] Compilation

[٦] Deployment

[۷] Restart

برای اعمال تغییرات و تولید مجدد سیستم، می‌توان به دو دسته تقسیم کرد:

(a) روش دستی: این روش عبارت است از تغییر متن برنامه توسط برنامه‌نویس. در این روش، به‌جز محیط مجتمع تولید[۸] (شامل محیط برنامه‌نویسی و مترجم) ابزار دیگری مورد استفاده قرار نمی‌گیرد. سیستم‌هایی که در تولید آن‌ها از روش دستی استفاده شده است، تغییر آن‌ها نیز نیازمند روش دستی است. علاوه بر این سیستم‌هایی که تولید آن‌ها همراه با پالایش خودکار[۹] از یک مدل با سطح تجرید[۱۰] بالاتر به یک مدل با سطح تجرید پایین‌تر بوده است نیز در مواردی نیازمند روش دستی برای اعمال تغییرات هستند. علت این مطلب این است که در تولید این‌گونه سیستم‌ها به‌دلایلی پس از انجام پالایش خودکار، مدل با سطح تجرید بالاتر کنار گذاشته شده است و اصلاحات مورد نیاز، در مدل با سطح تجرید پایین‌تر اعمال شده است. دلیل این امر ممکن است کافی نبودن جزئیات در مدل حاصل از پالایش خودکار باشد. مثال این گونه روش‌ها تولید کد از مدل‌هایی مثل مدل UML[۱۱] است که در آن چارچوب سیستم (مثلاً واسط‌های کلاس‌ها) تولید می‌شود ولی محتوای آن‌را باید با برنامه‌نویسی ایجاد کرد.

(b) روش‌های تولیدی: ممکن است تغییردهندهٔ سیستم به‌طور مستقیم متن نهایی برنامه را تغییر ندهد، بلکه با تغییر مدل‌ها یا توصیف[۱۲] سطح بالاتر سیستم یا حتی یک متن برنامهٔ میانی در طی یک یا چند مرحله پالایش، تبدیل[۱۳] یا تولید[۱۴] خودکار به سیستم قابل اجرای جدید دست پیدا کند. معمولاً، در این روش‌ها در تولید یا نگه‌داری سیستم برنامه‌نویس متن برنامهٔ نهایی را تغییر نمی‌دهد، درغیراین‌صورت، با تولید مجدد متن برنامه، تغییرات از بین می‌روند. روش‌هایی مثل برنامه‌نویسی جنبه‌گرا [۱۲]، برنامه‌نویسی ویژگی‌گرا[۱۵] [۱۳]، و متامدلسازی[۱۶] [۱۴] مثال‌هایی از روش‌های تولیدی هستند. بنابراین، خاصیت اصلی تغییر به‌روش ایستا، توقف اجرا و اصلاح متن برنامه و ترجمهٔ مجدد می‌باشد و تکنیک اصلاح و ایجاد سیستم جدید اهمیتی ندارد.

۲) **پویا**[۱۷]: اینگونه تغییرات در زمان اجرا صورت می‌گیرد و در خود سیستم امکاناتی برای تغییر تعبیه شده است، درنتیجه نیازی به تغییر متن برنامه نیست. به این سیستم‌ها قابل تطبیق یا انعکاسی[۱۸] گفته می‌شود. روش‌هایی که امکان این‌گونه تغییرات را فراهم می‌کنند خود بر اساس عامل تغییر دهنده به دو دسته تقسیم می‌شوند:

---

[۸] Integrated Development Environment (IDE)

[۹] Automated Refinement

[۱۰] Abstraction Level

[۱۱] Unified Modeling Language: زبانی برای مدل‌سازی تصویری است که در سال ۱۹۹۶ توسط شرکت رَشنال عرضه شده است.

[۱۲] Specification

[۱۳] Transformation

[۱۴] Generagtion

[۱۵] Feature-Oriented Programming

[۱۶] Meta-Modeling

[۱۷] Dynamic

[۱۸] Reflective

(a) روش‌های خود-تطبیقی[۱۹] یا تطبیق‌یابی[۲۰] : سیستم با تشخیص خود، به‌منظور سازگاری با شرایط (کاربر یا محیط اطراف) در رفتار یا پیکربندی[۲۱] خود تغییراتی اعمال می‌کند. از این روش‌ها در سیستم‌هایی از قبیل سیستم‌های با واسط کاربر هوشمند، سیستم‌های شبکه‌ای، توزیع‌شده، محاسبات سیار[۲۲]، محاسبات فراگیر[۲۳]، مأموریت-بحرانی[۲۴]، و ایمنی-بحرانی[۲۵] برای اهداف متفاوتی استفاده می‌شود. به‌طورکلی، می‌توان این قابلیت را نوعی هوشمندی قلمداد کرد. مثال‌هایی از کاربرد این روش‌ها عبارتند از:

- تطبیق پویای رفتار: برای مثال، اگر در حین یک محاسبهٔ توزیع‌شده، به‌طور ناگهانی، پهنای باند از میزان مشخصی کمتر شود، ممکن است استراتژی محاسبه به‌طور خودکار تغییر کند.
- بازپیکربندی پویا[۲۶] [۸، ۱۵]: یک سناریوی نمونه این است که در صورت زیاد شدن درخواست سرویس از یک مؤلفهٔ سرویس‌دهنده، به‌طور خودکار نمونهٔ[۲۷] جدیدی از مؤلفه در میزبان[۲۸] دیگری راه‌اندازی شده و بار کاری بین دو نمونه تقسیم گردد. یا در صورت نیاز به نسخهٔ جدیدی از یک مؤلفه که در حال حاضر در میزبانی وجود ندارد، مؤلفه از جایی دریافت شده[۲۹]، به‌طور خودکار جایگزین نسخهٔ قدیمی گشته، اجرا گردد. از جمله دلایل بازپیکربندی پویا جبران خطا می‌باشد. همچنین، یک عامل متحرک[۳۰] که برای انجام کاری از یک میزبان به میزبان دیگری مهاجرت می‌کند مثالی از این نوع تطبیق است.
- تطبیق با کاربر: برخی سیستم‌ها می‌توانند با کاربر خود تطبیق پیدا کنند و تمایلات کاربر خود را به خاطر بسپارند و روش تعامل با کاربر را طوری تغییر دهند که کارایی کاربر[۳۱] بالاتر رود [۷]. برای مثال ممکن است سیستم غلط‌های کاربر را اصلاح کند یا انتخاب‌های او را حدس بزند.

(b) روش‌های تطبیق توسط انسان[۳۲]: در این روش‌ها، سیستم امکاناتی در اختیار کاربر انسانی، شامل تولیدکننده[۳۳]، کاربر، یا متخصص دامنه[۳۴] قرار می‌دهد، تا بتواند ساختار و رفتار سیستم را با نیازمندی‌های جدید تطبیق دهد[۳۵]. به‌عبارت‌دیگر، با استفاده از این روش‌ها سیستم بدون

---

[۱۹] Self-Adaptivity
[۲۰] Adaptivity
[۲۱] Configuration
[۲۲] Mobile Computing
[۲۳] Pervasive Computing
[۲۴] Mission-Critical
[۲۵] Safety-Critical
[۲۶] Dynamic Reconfiguration
[۲۷] Instance
[۲۸] Host
[۲۹] Download
[۳۰] Mobile Agent
[۳۱] Efficiency
[۳۲] Adaptability
[۳۳] Developer
[۳۴] Domain Expert
[۳۵] برخی از مؤلفین معتقدند که مدل تحلیل سیستم می‌تواند توسط متخصصین دامنه‌ای که در زمینهٔ تحلیل (مثلاً به‌روش شیءگرا) آموزش دیده‌اند ایجاد شود [۱۶].

برنامه‌نویسی تکامل[۳۶] پیدا می‌کند. مثال‌هایی از کاربرد این روش‌ها عبارتند از:

- تطبیق جریان کار[۳۷] در سیستم‌های مدیریت جریان کار پویا[۳۸]: سیستم‌های جریان کار روند انجام یک فرآیند حرفه در سازمان را مدل می‌کنند. در صورت تغییر فرآیند حرفه، لازم است بتوان به‌سرعت و بدون تغییر متن برنامه، این تغییر را در سیستم اعمال کرد. برای رسیدن به این هدف سیستم‌های مدیریت جریان کار پویا به‌وجود آمده‌اند که مدلی پویا از فرآیندهای سیستم دارند و اجازه می‌دهند متخصصین دامنه با استفاده از یک سیستم نمادگذاری[۳۹] سطح بالا (یک زبان متنی یا گرافیکی سطح بالا) جریان کار را تطبیق دهند. در قسمت ۲—۵—۳ مباحثی در مورد این سیستم‌ها مطرح می‌گردد.
- تعریف موجودیت جدید: تعریف سیاست‌های جدید در یک سیستم بیمه، نمونه‌ای از این تغییرات است. لازم است سیستمی که برای خودکار کردن فرآیندهای بیمه‌ای نوشته می‌شود، امکان تعریف و تغییر سیاست‌های بیمه‌ای ساده و مرکب را بدهد [۱۷، ۱۰]. برای این منظور بهتر است یک مدل پویا از سیاست‌های بیمه‌ای و زبانی سطح بالا برای تغییر این مدل استفاده شود.

یک سیستم قابل تطبیق، نیازمند یک معماری قابل تطبیق[۴۰] (معماری پشتیبانی کننده قابلیت تطبیق) است [۱۳] به‌عبارت‌دیگر، قابلیت تطبیق از ویژگی‌هایی است که معماری نرم‌افزار باید از آن پشتیبانی کند و درغیراین‌صورت، عملی نخواهد شد (معماری نرم‌افزار موضوع فصل ۳ می‌باشد.) معمولاً، معماری این سیستم‌ها این‌گونه است که سیستم مدلی از خود (متامدل) دارد و رابطهٔ علّی بین رفتار خود و آن مدل را حفظ می‌کند. این مدل انعکاسی از سیستم است و تغییرات آن در سیستم منعکس می‌شود. در معماری این سیستم‌ها لایه‌ها یا مؤلفه‌هایی وجود دارد (متا—لایه[۴۱] [۲۰، ۱۹، ۱۸]) که مکانیزمی برای اعمال تغییرات پویا در مدل سیستم به‌وجود می‌آورند.

منظور از طبقه‌بندی بالا تنها معرفی راه‌های مختلف تغییرات در سیستم‌ها و عامل اعمال‌کنندهٔ آن‌ها می‌باشد، نه طبقه‌بندی سیستم‌ها. به عبارت دیگر، نمی‌توان یک سیستم نرم‌افزاری را در یکی از طبقات پویا یا ایستا قرار داد زیرا ممکن است قسمت‌های مختلف یک سیستم را به‌طرق مختلفی بتوان تغییر داد. برای مثال، ممکن است یک سیستم با استفاده از امکان بازپیکربندی پویای مؤلفه‌ها بتواند به‌صورت خودکار سرویس‌های جدیدی را بارگذاری کند، درحالی‌که برای تغییر دادن منطق حرفه لازم باشد یک برنامه‌نویس متن برنامهٔ یک مؤلفه را عوض کند. گذشته از این، ممکن است در بعضی موارد برای اعمال یک تغییر در نرم‌افزار چند روش تغییر به‌طور هم‌زمان به‌کار گرفته شود. به‌عنوان مثال، یک سیستم مبتنی بر مؤلفهٔ باز[۴۲] را تصور کنید که اجازه می‌دهد یک مؤلفه به‌صورت وصل‌کن—اجراکن[۴۳] در سیستم به‌روز شود، درحالی‌که خود این مؤلفه از طریق برنامه‌نویسی به‌روز شده است. بنابراین، بر حسب این‌که چه چیزی قرار است تغییر کند، نوع تغییر در سیستم متفاوت می‌شود. قسمت ۲—۲ در مورد محل تغییر بحث می‌کند.

---

[۳۶] Evolution
[۳۷] Workflow
[۳۸] Adaptive Workflow Management System
[۳۹] Notation
[۴۰] Adaptable Architecture
[۴۱] Meta-Layer
[۴۲] Open Component-Based System
[۴۳] Plug and Play

۲—۲ محل تغییر

اگر دقیق‌تر به موضوع قابلیت تغییر نظر شود، این سؤال مطرح می‌گردد که چه سیستمی را می‌توان قابل تغییر دانست. آیا در چنین سیستمی باید هر تغییری ممکن باشد؟ این‌طور نیست. زیرا می‌توان نشان داد هیچ سیستمی وجود ندارد که هر نوع تغییری در آن امکان‌پذیر باشد. از این گذشته، پشتیبانی از بسیاری از تغییرات برای یک سیستم ضروری نیست و این تغییرات در طول حیات سیستم هرگز اتفاق نمی‌افتند. به‌عنوان مثال، فرض کنید سیستمی ۴ ویژگی A، B، C، و D دارد. از بین این ویژگی‌ها A و B نیازمند هیچ تغییری نیستند، درحالی‌که C و D ممکن است تغییر کنند. حال اگر در این سیستم، تنها تغییر A و C ممکن باشد، آیا می‌توان گفت این سیستم قابل تغییر است؟ اگر جواب مثبت باشد می‌توان پرسید چرا تغییر B در آن ممکن نیست و اگر جواب منفی باشد می‌توان گفت چرا تغییر A در آن ممکن است. بنابراین، در حالت کلی نمی‌توان گفت یک سیستم نرم‌افزاری قابل تغییر است یا خیر. به‌عبارت دیگر، در هر سیستم برخی از تغییرات به‌سادگی و برخی با دشواری اعمال می‌گردد. پس قابلیت تغییر (همچنین، انواع آن مثل قابلیت تطبیق)، همان‌طورکه برخی مؤلفین اذعان دارند [۱]، موضوعی نسبی است (این مطلب در مورد سایر ویژگی‌های کیفی نیز صحت دارد [۱]).

با پذیرش این مطلب، مسائل جدیدی مطرح می‌شود. برای مثال، چگونه می‌توان یک سیستم قابل تغییر ایجاد کرد، درحالی‌که عبارت قابلیت تغییر ابهام دارد؟ یا چگونه می‌توان این مطلب را در مورد یک سیستم ارزیابی کرد؟ یا چگونه کاربر سیستم می‌تواند از تغییرپذیری سیستم اطمینان حاصل کند؟ برای پاسخ به این مسائل باید راهی پیدا کرد که قابلیت تغییر به‌دقت در مورد یک سیستم تعریف شود. در ضمن، همان‌طورکه قبلاً اشاره شد، نمی‌توان انتظار داشت یک سیستم خودبه‌خود دارای یک سری ویژگی‌های کیفی باشد و اگر قرار است قابلیت تغییر داشته باشد، باید در زمان تولید سیستم، طراحی برای تغییر انجام شود. یک راه مناسب برای حل این مسائل، استفاده از سناریو-نویسی برای مشخص کردن ویژگی‌هایی است که ممکن است تغییر کند [۱]. در این روش، قابلیت تغییر با استفاده از یک مجموعه مشخص از سناریوهای تغییرات، مشخص می‌گردد. درنتیجه، کافی است تغییرات مشخص شده در سناریوها، به‌راحتی، قابل اعمال باشند تا به سیستم قابل تغییر گفته شود و لازم نیست سایر انواع تغییرات ساده باشند. این سناریوها را می‌توان به‌عنوان نیازمندی‌های غیرکارکردی سیستم در نظر گرفت و پس از توافق مشتری و تولیدکننده مبنایی برای ارزیابی سیستم قرار داد.

مسئلهٔ محل تغییر در مورد قابلیت تطبیق نیز مطرح است. در سیستم‌های مختلفی که به‌نحوی قابل تطبیق هستند، قسمت‌های قابل تطبیق متفاوتی به چشم می‌خورند. تطبیق‌های ممکن در یک سیستم نرم‌افزاری، انواع مختلفی دارند، مانند:

- تطبیق در ارتباط کاربر با کامپیوتر: در برخی سیستم‌ها امکان برخی تطبیق‌های رفتاری از طریق پرسیدن تمایلات کاربر[۴۴] یا فایل تنظیمات[۴۵] وجود دارد. همچنین، بعضی سیستم‌ها خود هوشمندانه روش ارتباط با کاربر را به‌مرور زمان تطبیق می‌دهند. این نوع تطبیق را می‌توان تطبیق با کاربر یا تطبیق در ارتباط کاربر با کامپیوتر نامید. همان‌طورکه گفته شد، این نوع تطبیق هم می‌تواند توسط انسان اعمال شود و هم می‌تواند خودکار باشد [۷].
- تطبیق در منطق حرفه: برخی سیستم‌ها به کاربر امکان تطبیق ساختاری می‌دهند (منظور از ساختار

[۴۴] User Prefrences
[۴۵] Configuration File

موجودیت‌های حرفه و ارتباطات آن‌ها می‌باشد.) به‌عنوان مثال، کاربر می‌تواند موجودیت‌های جدید تعریف کند. همچنین، ممکن است رفتار (شامل عملیات بر روی موجودیت‌های سیستم و جریان‌کار) قابل تعریف و تطبیق باشد. این نوع تطبیق که تطبیق در منطق حرفه می‌باشد، در دامنه‌هایی مورد نیاز است که فرآیند حرفه به‌طور دائمی در حال تغییر است. خود این نوع را می‌توان به دو دستهٔ تطبیق ساختاری و تطبیق رفتاری تقسیم کرد. این نوع تطبیق توسط انسان اعمال می‌شود. در قسمت ۴—۱—۳ تعریف ساختار و رفتار ارائه خواهد شد.

- تطبیق در پیکربندی سیستم (بازپیکربندی پویا[46]): در مقیاسی بزرگ‌تر بعضی سیستم‌های توزیع شده امکان تطبیق توپولوژی مؤلفه‌ها را در زمان اجرا دارند. به این نوع تطبیق پیکربندی مجدد پویا گفته می‌شود. این نوع تطبیق هم می‌تواند توسط انسان اعمال شود و هم می‌تواند خودکار باشد.

در این پایان‌نامه به قسمتی از یک سیستم که ممکن است توسط متخصص دامنه مورد تطبیق واقع شود یک ویژگی قابل تطبیق[47] از سیستم گفته می‌شود[48] به‌عنوان مثال، ممکن است یک مشخصه از یک نوع موجودیت حرفه، یک ویژگی قابل تطبیق باشد که امکان اضافه، حذف، یا به‌روز رسانی آن وجود دارد.

واضح است که مفاهیم محل تغییر و نوع‌تغییر مستقل از هم یا اصطلاحاً عمود بر هم[49] هستند.

۲—۳ شرایط تغییر

مطلب قابل بحث دیگری در مورد قابلیت تغییر، این است که با چه شرایطی می‌توان گفت که یک ویژگی قابلیت تغییر دارد؟ در در زمان بینهایت و با منابع نامحدود هر نوع تغییری را می‌توان در سیستم اعمال کرد. بنابراین، در تئوری تمام ویژگی‌های هر سیستمی قابل تغییر هستند. اما قطعاً نمی‌توان به چنین خاصیتی قابلیت تغییر گفت. راه‌حلی که می‌توان برای رفع این ابهام ارائه داد، چنین است: برای این‌که یک ویژگی در یک سیستم قابل تغییر باشد باید زمان و منابع (تخصص‌های لازم، هزینه، و ...) مورد نیاز برای این کار کمتر از یک مقدار مشخص باشد. حد بالای منابع مورد نیاز و سایر شرایط اعمال تغییر را می‌توان در سناریوی مربوط به هر تغییر مستند کرد [۱]. از جمله منابع مورد نیاز برای یک تغییر منابع انسانی است. به‌عبارت‌دیگر، باید مشخص کرد برای اعمال یک تغییر به چه تخصصی احتیاج است (مثلاً معمار نرم‌افزار یا برنامه‌نویس).

در مورد قابلیت تطبیق نیز باید شرایط به‌دقت مشخص شود. حداقل شرط تطبیق، همان‌طورکه گفته شد، این است که تغییر در زمان اجرا اعمال شود. به‌عبارت‌دیگر، نیاز به‌تغییر متن برنامه‌های سیستم، ترجمه، آزمون، استقرار، و راه‌اندازی مجدد نباشد. اما این‌که یک نوع تطبیق چه منابعی احتیاج دارد، توسط چه‌کسی انجام می‌شود، و اعمال تغییر چه تأثیری بر عملکرد کاربران مشغول کار دارد، و مسائلی از این قبیل، باید برای هر نوع تطبیق مشخص شود.

---

[46] Dynamic Reconfiguration

[47] Adaptable Feature

[48] از کلمهٔ ویژگی در مقالات به‌معانی دیگری استفاده شده است که ممکن است به این تعریف نزدیک باشند یا با این تعریف سازگار نباشند (مثل تعریف ارائه شده در [۲۲]). این تعریف از جایی گرفته نشده است و به‌دلیل یافت نشدن کلمهٔ دیگر از کلمهٔ ویژگی استفاده شد.

[49] Orthogonal

۲—۴ تعریف مسئله

هدف از این پایان‌نامه ارائهٔ روشی برای ایجاد نرم‌افزارهای با قابلیت تطبیق عملیات بر روی موجودیت‌های حرفه (رفتار)، توسط انسان می‌باشد.

در نتیجه، تغییرات ایستا و روش‌های بهبود آن‌ها خارج از بحث هستند. همچنین، روش‌های خود-تطبیقی نیز با موضوع مورد نظر ارتباطی ندارد. محل تغییر (ویژگی‌های مورد تطبیق) را نمی‌توان بدون در نظر گرفتن یک دامنه و یک سیستم کاربردی برای آن دامنه مشخص کرد اما از سه نوع تطبیق نام‌برده شده، روش ارائه شده در این پایان‌نامه به تطبیق منطق حرفه اختصاص دارد. بنابراین، بحثی از تطبیق ارتباط انسان و کامپیوتر و بازپیکربندی پویا و سایر انواع قابل تصور از تطبیق نمی‌شود[۵۰]. در مورد شرایط تغییر نیز باید در مورد هر ویژگی مورد تطبیقی که در سیستم مورد نیاز است در هنگام ایجاد سیستم تصمیم‌گیری شود. اما به‌طورکلی، می‌توان گفت تطبیق مورد نظر باید توسط یک متخصص دامنه در زمانی از مرتبهٔ یک ساعت قابل اعمال باشد.

به‌طور خلاصه مسئله مورد نظر این پایان‌نامه ارائهٔ روشی برای ایجاد سیستم‌های قابل تطبیق است که:

۱) امکان تطبیق رفتاری در منطق حرفه (در سطح عملیات بر روی موجودیت‌های حرفه و نه جریان کار) را به‌وجود آورد.

۲) تطبیق توسط متخصص دامنه قابل اعمال باشد.

۳) تطبیق در زمانی از مرتبهٔ یک ساعت قابل انجام باشد.

تعریف ارائه شده در این قسمت کلی است و لازم است برای برخی کلمات به‌کار رفته در آن، توضیح بیشتری ارائه گردد. در قسمت ۴—۱—۳ تعریف ساختار و رفتار ارائه خواهد شد. همچنین، در مورد دامنه در قسمت ۳—۳—۱، و تطبیق در سطح جریان کار و جداسازی آن از تطبیق مورد نظر این پایان‌نامه در قسمت ۲—۵—۳ بحث می‌شود.

ازاین‌پس، هرگاه نوع تطبیق ذکر نشود، منظور تطبیق توسط متخصص دامنه است.

۲—۵ کارهای مرتبط

در این قسمت، کارهای مرتبط با قابلیت تطبیق توسط انسان، مرور می‌شود. این کارها یا به‌طور مستقیم در مورد سیستم‌های قابل تطبیق هستند، یا مفاهیم و مکانیزم‌های مشترکی را بیان می‌کنند. قسمت ۲—۵—۱ کارهای انجام شده در مورد سبک معماری مدل قابل تطبیق شیء که مبنای کار انجام شده در این پایان‌نامه می‌باشد را معرفی می‌کند. در چهار قسمت بعد، به تحقیقات انجام شده در چهار زمینهٔ مرتبط اشاره می‌شود: متاداده در سیستم‌های مدیریت پایگاه‌داده‌ها، تطبیق در جریان کار در سیستم‌های مدیریت جریان‌کار قابل تطبیق، مدل‌های سطح بالای قابل اجرای نمایشی، سیستم‌های دارای انعکاس. علت اهمیت این زمینه‌ها در بحث قابلیت تطبیق، وجود مکانیزم‌های مشترک برای دست‌یابی به تغییرات در زمان اجراست.

---

[۵۰] البته بااین‌که سایر انواع تغییر و تطبیق مورد نظر نیستند بسیاری از مفاهیم و روش‌ها به‌صورت مشترک بین این موارد و تطبیق منطق حرفه وجود دارد و مطالعهٔ این روش‌ها در روش ارائه شده و نگارش این پایان‌نامه بی‌تأثیر نبوده است.

۲–۵–۱ سبک معماری مدل قابل تطبیق شیء

در سال ۱۹۹۸ جانسون متوجه یک سبک معماری شد که در سیستم‌های متعددی مورد استفاده قرار گرفته بود. او این روش را مدل پویای شیء[۵۱] نامید [۲۳]. این سبک به نام‌های مدل فعال شیء[۵۲] و چارچوب محصول تعریف شده توسط کاربر[۵۳] نیز در مقالات ظاهر شد تا این‌که نام سبک معماری مدل قابل تطبیق شیء[۵۴] به آن داده شد[۱۱]. این روش در برخی مقالات و کارگاه‌های آموزشی از جمله [۲۷، ۱۱، ۲٦، ۲۵، ۲۴]، مورد بحث قرار گرفت.

این سبک، روش قدرتمندی برای مدل‌سازی دامنه‌های فوق‌العاده پویا به‌شمار می‌رود. ایدهٔ این سبک معماری این است که برای پویا بودن موجودیت‌های سیستم در تعریف آن‌ها به‌جای کلاس‌ها از اشیاء استفاده گردد. در این روش قسمتی از منطق حرفه به جای متن برنامه در متادادهٔ سیستم قرار می‌گیرد و در پایگاه‌داده‌ها ذخیره می‌شود. در نتیجه، چون به‌روزرسانی متاداده در زمان اجرا ممکن می‌باشد، پویایی مورد نظر حاصل می‌شود. این روش برای ایجاد ساختار پویا از الگوهای طراحی نوع‌شیء[۵۵] و خصوصیت[۵٦] [۱۱] و برای ایجاد رفتار پویا از الگوهای راهبرد[۵۷] [۲۸]، مفسر[۵۸]، و شیء قاعده[۵۹] [۱۷] استفاده می‌کند. همچنین، در این روش بر حسب نیاز از الگوهای دیگری مثل مرکب[٦۰] نیز استفاده می‌شود. مدل قابل تطبیق شیء در چندین سیستم واقعی به‌کارگرفته شده و موفقیت‌آمیز بوده است. در ادامه برخی از این سیستم‌ها بررسی معرفی می‌شوند.

چارچوب محصول تعریف شده توسط کاربر [۱۰] با هدف فراهم‌کردن امکان تعریف پویای محصولات مرکب (یک محصول مرکب از محصولات ساده یا مرکب دیگر تشکیل می‌شود)، مثل یک سیاست بیمه‌ای، برای کاربران، و همچنین، تعریف عملیات ارزیابی ساده بر روی این محصولات، مثل برآورد کردن هزینهٔ سیاست بیمه‌ای از روی اجزای آن بدون نیاز به برنامه‌نویسی، ایجاد شده است. منظور از عملیات ارزیابی محاسبهٔ کمیت‌هایی است که مقدار آن برای یک شیء تابعی از مقدار آن برای اجزای تشکیل‌دهندهٔ آن باشد. این سیستم برای مدل‌سازی سیاست‌های بیمه‌ای به‌وجود آمد. در تولید این سیستم تخمین زده شد که اگر قرار باشد تمام ترکیب‌های ممکن سیاست‌های بیمه‌ای از طریق وراثت در کلاس‌ها پیاده‌سازی شوند، ۱۰۰۰۰ کلاس مورد نیاز خواهد بود. بنابراین، ترجیح داده شد به‌جای وراثت از ترکیب استفاده شود. با وجود قدرت زیاد در مدل‌سازی ساختار در این سیستم امکان توصیف رفتارهای پیچیده‌تر از آنچه گفته شد بدون استفاده از برنامه‌نویسی وجود ندارد.

چارچوب دامنهٔ پزشکی IPDH[٦۱] [۱۱] برای تعریف و ثبت اطلاعاتی در مورد بیماران، و افراد مرتبط با آن‌ها مثل اولیا و پزشکان به‌وجود آمده است. این سیستم شامل چند برنامهٔ کاربردی متفاوت است که

---

[۵۱] Dynamic Object Model

[۵۲] Active Object Model

[۵۳] User Defined Product Framework

[۵۴] Adaptive Object Model Architecture Style

[۵۵] Type-Object Pattern

[۵٦] Property Pattern

[۵۷] Strategy Pattern

[۵۸] Interpreter Pattern

[۵۹] Rule-Object Pattern

[٦۰] Composite

[٦۱] Illinois Department of Public Health

اطلاعاتی به‌صورت مشترک بین آن‌ها وجود دارد. از جملهٔ این اطلاعات، مشاهدات پزشکی در مورد افراد و ارتباط افراد و سازمان‌ها می‌باشد. یک مشاهده برای ثبت داده‌ای در مورد بیمار در یک دورهٔ زمانی مشخص، مانند فشار خون، به‌کار می‌رود. یک مشاهده ممکن است مرکب (ترکیبی از مشاهدات دیگر) باشد. هر مرکز پزشکی انواع مشاهدات خاص خود را تعریف می‌کند که تعداد آن‌ها بسیار زیاد است. همچنین، به‌صورت مداوم انواع مشاهدات جدیدی تعریف می‌شوند. بنابراین، بازهم مشکلاتی مشابه آن‌چه قبلاً گفته شد، وجود دارد. چارچوب IPDH نیز یک مدل قابل تطبیق شیء است و براساس الگوی مشاهده [۱۶] ایجاد شده است. این سیستم نیز تنها از عملیات پویای ساده مثل اعتبارسنجی داده‌های ورودی پشتیبانی می‌کند.

چارچوب گردش‌کار سند آرگو[۶۲] [۲۹] برای سازمان آرگو که چندصد مدرسه را مدیریت می‌کند به‌وجود آمده است. این سیستم را می‌توان یک خط تولید نرم‌افزار برای ایجاد سیستم‌هایی بایک مدل حرفهٔ مشترک که نیازمند پایگاه‌داده‌ها، اسناد الکترونیکی، جریان کار، و عملکرد اینترنتی می‌باشند، به‌حساب آورد. این چارچوب فوق‌العاده قدرتمند است و به‌خوبی می‌توان ساختار و رفتار را در آن مدلسازی کرد و تطبیق داد. در این سیستم قواعد رفتاری بوسیلهٔ نوشته‌های[۶۳] زبان Smalltalk بیان می‌شوند.

[۳۰] یک الگو برای استفاده از برنامه‌نویسی جنبه‌گرا در ایجاد برنامه‌های قابل تطبیق ارائه می‌دهد و در [۳۱] نحوهٔ به‌کارگیری این الگو برای مدل‌های قابل تطبیق شیء توضیح داده می‌شود. این الگو خود مکانیزم تطبیق نیست، بلکه به ایجاد و نگه‌داری برنامه‌های قابل تطبیق از جمله مدل‌های قابل تطبیق شیء کمک می‌کند.

یک مشکل در تطبیق رفتاری در سیستم‌های مبتنی بر مدل قابل تطبیق شیء، این است که این سیستم‌ها نمی‌توانند مدل‌سازی رفتاری را که از قبل پیش‌بینی نشده، با یک زبان ساده پشتیبانی کنند.

۲—۵—۲ سیستم‌های مدیریت پایگاه‌داده‌ها

سیستم‌های مدیریت پایگاه‌داده‌ها[۶۴] یک نمونهٔ کلاسیک از نگه‌داری توصیف سیستم به‌صورت متاداده هستند. شمای[۶۵] هر سیستم پایگاه‌داده‌ها به‌صورت بخشی از کاتالوگ یا متاداده در سیستم مدیریت پایگاه‌داده‌ها تعریف می‌شود. یک سیستم مدیریت پایگاه‌داده‌ها را می‌توان به یک موتور تطبیق (موتور تطبیق در قسمت ۳—۳ تعریف می‌گردد) برای دامنهٔ سیستم‌های پایگاه‌داده‌ها تشبیه کرد. هر سیستم پایگاه‌داده‌ها یک محصول خاص از این دامنه حساب می‌شود. زبان مخصوص دامنه‌ای که در این سیستم‌ها برای تعریف و تطبیق یک سیستم پایگاه‌داده‌ها به‌کار می‌رود زبان تعریف داده[۶۶] (مانند SQL) است. یک سیستم پایگاه‌داده را می‌توان با تغییر متادادهٔ آن در زمان اجرا تغییر داد.

حال این سؤال مطرح می‌شود که با وجود سیستم‌های مدیریت پایگاه‌داده‌ها، چه نیازی به سیستم‌های قابل تطبیق وجود دارد و اساساً سیستم‌های قابل تطبیق چه تفاوتی با سیستم‌های پایگاه‌داده‌ها دارند. شباهت این دو سیستم از نظر کاربرد این است که در هر دو می‌توان موجودیت‌های یک محیط را مدل‌سازی کرد و روی آن‌ها عملیات انجام داد و حتی چنان که در بالا گفته شد مکانیزم کار نیز شبیه است. پاسخ سؤال این

[۶۲] Argo Document Workflow Framework

[۶۳] Script

[۶۴] Database Management Systems

[۶۵] Schema

[۶۶] Data Definition Language (DDL)

است که با وجود شباهت‌های ظاهری گفته شده در عملکرد این دو نوع سیستم در حقیقت این دو سیستم با اهداف مختلفی به‌وجود آمده‌اند و به‌همین‌دلیل، در دو سطح تجرید متفاوت هستند و کاربرد متفاوتی دارند. برخی دلایل این مطلب از این قرارند:

- یک سیستم پایگاه‌داده‌ها (محصول سیستم‌های مدیریت پایگاه‌داده‌ها) در سطح تجرید بالاتری از یک سیستم نرم‌افزاری تمام عیار است زیرا باوجوداین‌که در تئوری تمام انواع داده‌ها و عملیات روی آن‌ها در این سیستم‌ها ممکن است، این داده‌ها و عملیات، ابتدایی و سطح پایین هستند و کار کردن با این سیستم‌ها برای یک کاربر غیربرنامه‌نویس به‌راحتی امکان‌پذیر نیست. به‌عبارت‌دیگر، سیستم واسطی مورد نیاز است که بین کاربر و سیستم پایگاه‌داده ارتباط برقرار کند.
- سیستم‌های قابل تطبیق خود از سیستم‌های مدیریت پایگاه‌داده برای ذخیره داده و انجام عملیات روی آن، استفاده می‌کنند و خود اقدام به عملیات سطح پایین پردازش داده نمی‌کنند.
- معمولاً، یک سیستم قابل تطبیق برای یک دامنهٔ خاص طراحی می‌شود و موجودیت‌ها و عملیات آن دامنه را پشتیبانی می‌کند. بنابراین، چنین سیستمی در سطح تجرید پایین‌تری نسبت به یک سیستم مدیریت پایگاه‌داده‌ها، که برای تمام دامنه‌ها قابل استفاده است، قرار دارد.

۲—۵—۳ سیستم‌های جریان‌کار قابل تطبیق

دامنهٔ دیگری که نیاز شدیدی به قابلیت تطبیق دارد دامنهٔ سیستم‌های جریان کار است[۶۷]. از این سیستم‌ها برای خودکارسازی فرآیند حرفه[۶۸] (که شامل جریان کار، گردش اسناد، کار گروهی، و این قبیل موارد می‌شود) استفاده می‌گردد. برای رفع نیاز قابلیت تطبیق در سیستم‌های جریان کار، سیستم‌های جریان کار قابل تطبیق[۶۹] بوجود آمده‌اند. مشابه با آنچه در قسمت قبل در مورد سیستم‌های مدیریت پایگاه‌داده‌ها گفته شد، یک سیستم مدیریت جریان کار را می‌توان یک موتور تطبیق برای ایجاد سیستم‌های جریان‌کار دانست. این سیستم‌ها باید از دو جهت قابلیت تطبیق داشته باشند [۳۴، ۳۳، ۳۲]:

۱) تعریف و تغییر پویای فرآیند (جریان کنترل[۷۰]): در این سیستم‌ها لازم است امکان توصیف جریان کار با یک زبان سطح بالا (معمولاً تصویری) وجود دشته باشد. همچنین، باید امکان تغییر تعریف فرایند (تکامل[۷۱] جریان کار) در زمان اجرا وجود داشته باشد.

۲) قابلیت پشتیبانی از استثناها[۷۲] (تشخیص و رسیدگی به آن‌ها) در اجرای یک فرآیند و پشتیبانی از فرآیندهای ویژه[۷۳] (دیمی)، که از قبل قابل تعریف نیستند: علت نیاز به فرایندهای ویژه ممکن است یکی از این دلایل باشد: عدم امکان تعریف جریان‌کار در زمان ایجاد سیستم بدلیل حجم زیاد یا نامشخص بودن بعضی مسائل، نیاز به تصمیم‌گیری کاربر در زمان مواجه شدن با بعضی مسائل، اتفاقات غیرقابل پیش‌بینی، و بروز خطا.

[۶۷] Workflow Systems

[۶۸] Business Process

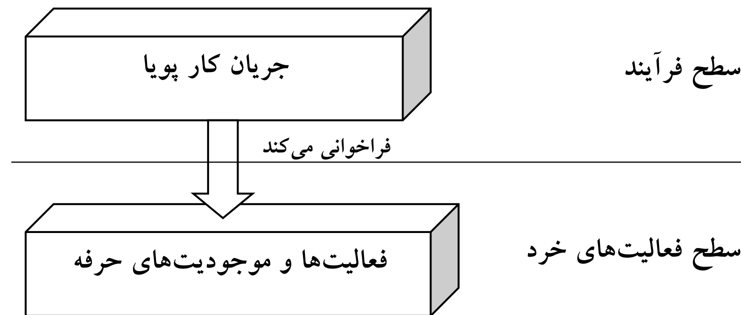
[۶۹] Adaptive Workflow Systems

[۷۰] Control Flow

[۷۱] Evolution

[۷۲] Exception

[۷۳] Ad hoc



شکل ۲—۱: لایه‌بندی سیستم‌های جریان‌کار قابل تطبیق.

امکان تطبیق تعریف فرآیند خود مسائل جدیدی به‌دنبال دارد، از جمله:

- باید تکلیف نمونه‌های ناتمام جریان کار[۷۴] مشخص شود. در برخی موارد باید این نمونه‌ها به نمونه‌های جریان کار جدید تبدیل شوند (انتقال نمونه[۷۵]) [۳۴] در برخی موارد ممکن است با این نمونه‌ها بصورت استثناء برخورد شود.
- سابقهٔ[۷۶] نمونه‌های پایان یافته جزئی از داده‌های سیستم است که باید حفظ شود. بنابراین، باید تضمین شود تغییر تعریف فرآیندها لطمه‌ای به بازیابی نمونه‌های قدیمی نمی‌زند.
- در برخی موارد لازم است سابقهٔ تغییرات در تعریف فرآیند نیز نگه‌داری شود [۳۵].

حال این سؤال مطرح است که با وجود سیستم‌های جریان کار قابل تطبیق، آیا مشکل سیستم‌های قابل تطبیق حل شده است؟ جواب به‌دلایل زیر منفی است:

۱) در سیستم‌های جریان کار جداسازی دغدغه‌ها به این شکل صورت گرفته است که سطح جریان کنترل بین فعالیت‌ها[۷۷] (یا کارها[۷۸]) از خود فعالیت‌ها جدا شده است. شکل ۲—۱ این مطلب را نشان می‌دهد. هدف در مباحث سیستم‌های جریان کار قابل تطبیق، سطح جریان کنترل است، نه سطح فعالیت، که مورد بحث در سیستم‌های قابل تطبیق عمومی می‌باشد. می‌توان پویایی را به سطح فعالیت نیز تعمیم دارد. برای این منظور، لازم است موجودیت‌های حرفه و عملیات روی آن‌ها به‌صورت پویا قابل تعریف باشند. شکل ۲—۲ لایه‌بندی یک سیستم جریان‌کار کاملاً قابل تطبیق را نشان می‌دهد.

۲) تمام سیستم‌هایی که نیازمند قابلیت تطبیق هستند لزوماً از دامنهٔ سیستم‌های جریان کار نیستند.

۳) سیستم‌های مدیریت جریان کار در حال حاضر با مشکلاتی در پشتیبانی از تغییرات مواجه هستند [۳۲] و موضوع قابلیت تطبیق در سیستم‌های جریان کار یک مبحث باز است که تحقیقات در مورد آن ادامه دارد (برای نمونه می‌توان به برخی کارها که اخیراً انجام شده است اشاره کرد، مانند [۳۵].)

---

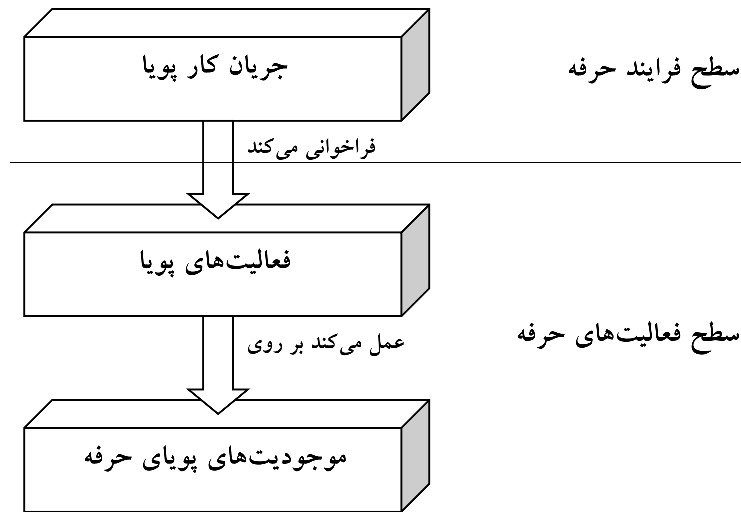
[۷۴] Unfinished Workflow Instances

[۷۵] Instance Migration

[۷۶] History

[۷۷] Activities

[۷۸] Tasks



شکل ۲–۲: لایه‌بندی یک سیستم جریان‌کار کاملاً قابل تطبیق.

به هر حال، این مباحث شباهت بسیار زیادی به مباحث مطرح در مورد قابلیت تطبیق دارند و راه‌حل‌های هر یک از این دو حوزه در دیگری قابل استفاده است. به‌عنوان مثال، در [۳۵، ۳۶] با استفاده از سبک معماری مدل قابل تطبیق شیء یک معماری برای سیستم‌های جریان کار قابل تطبیق، با نام جریان کار خُرد[۷۹] ارائه شده است.

۲–۵–۴ مدل‌های سطح بالای قابل اجرای نمایشی

متحرک‌سازی[۸۰] یا شبیه‌سازی اجرای مدل‌های سطح بالا (مثل مدل معماری و مدل طراحی) عملی است که برای اهدافی نظیر نمایش به ذی‌نفعان و وارسی[۸۱] مدل‌های سطح بالا قابل استفاده است. در این روش‌ها بر خلاف روش‌های متکی بر پالایش برنامه‌ای تولید نمی‌شود و یک ماشین مجازی[۸۲] مدلی را به‌عنوان ورودی دریافت می‌کند و آن‌را تفسیر می‌کند. برای این منظور مدل سیستم، متامدلی که زبان توصیف مدل را در اختیار کاربر قرار می‌دهد. به‌عنوان مثال، در [۳۷] یک ماشین مجازی برای تفسیر و اجرای مدلهای UML طراحی شده است. این سیستم از سلسله‌مراتب متامدل ۴ لایهٔ UML [۳۸] پشتیبانی می‌کند. برای ایجاد این سیستم UML توسعه داده شده است. در این سیستم رفتار با نمودار ترتیب و نمودار حالت مدل می‌شود.

۲–۵–۵ سیستم‌های انعکاسی

در [۳۹]، یک سیستم انعکاسی[۸۳] این طور تعریف می‌شود:

[۷۹] Micro-Workflow
[۸۰] Animation
[۸۱] Verification
[۸۲] Virtual Machine
[۸۳] Reflective System

یک سیستم انعکاسی، سیستمی است که از ساختارهایی [مدل‌هایی] استفاده می‌کند که نمایندهٔ [مدلی از] خود سیستم هستند. به مجموعهٔ این ساختارها نمایش-از-خود[۸۴] سیستم گفته می‌شود. نمایش-از-خود این امکان را به‌وجود می‌آورد که سیستم بتواند در مورد خود به سؤالاتی پاسخ دهد و از عملیاتی بر روی خود پشتیبانی کند. از آنجاکه این نمایش-از-خود با سیستم رابطهٔ علّی دارد[۸۵]، می‌توان گفت:

۱) سیستم همواره نمایش دقیقی از خود دارد.
۲) وضعیت[۸۶] و محاسبهٔ[۸۷] آن همواره با این نمایش-از-خود سازگار است. درنتیجه، یک سیستم انعکاسی می‌تواند با استفاده از محاسبات خود، خود را تغییر دهد.

منظور از این‌که سیستم و نمایش-از-خود آن رابطهٔ علّی دارند، این است که در صورت تغییر سیستم نمایش-از-خود سیستم نیز تغییر می‌کند و برعکس [۳۹].

انعکاس ابتدا در حوزهٔ هوش مصنوعی مطرح شده است [۱۹]. مقالهٔ [۳۹] که در سال ۱۹۸۷ نوشته شده است، در مورد استفاده از انعکاس در زبان‌های برنامه‌نویسی بحث می‌کند. انعکاس به حوزه‌های دیگری از قبیل معماری برنامه‌های کاربردی [۱۱]، سیستم‌های عامل [۱۹]، و میان‌افزارها[۸۸] [۴۰] نیز وارد شده است. این تعریف با کمی تغییر برای تمام سیستم‌های انعکاسی قابل استفاده است.

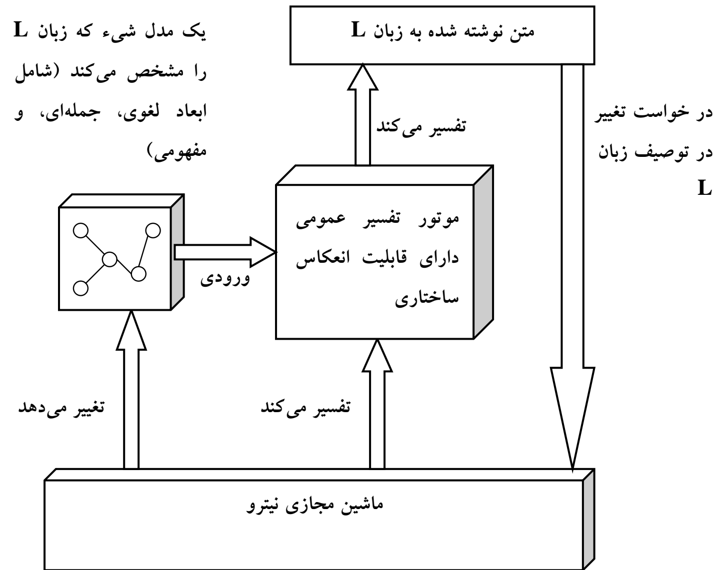
از جمله سیستم‌هایی که از تغییرات زمان اجرا پشتیبانی می‌کنند زبان‌های برنامه‌نویسی دارای قابلیت انعکاس هستند. در این زبان‌ها تطبیق توسط انسان صورت نمی‌گیرد، اما باتوجه‌به شباهت معماری برخی از این زبان‌ها به معماری سیستم‌های قابل تطبیق توسط انسان، آشنایی با این مباحث برای تحقیق در زمینهٔ قابلیت تطبیق مفید است. همچنین، می‌توان در ایجاد سیستم‌های قابل تطبیق از انعکاس زبان‌های برنامه‌نویسی استفاده کرد.

در [۲۱ ،۲۰]، انعکاس پشتیبانی شده توسط زبان‌های برنامه‌نویسی به سطوح زیر تقسیم می‌شود:

۱) **خودنگری**[۸۹]: امکان دسترسی به ساختار برنامه وجود دارد ولی امکان تغییر آن وجود ندارد. مثال این سطح انعکاس موجود در زبان جاوا می‌باشد (کتاب‌خانهٔ java.lang.reflection) [۲۱] که با استفاده از آن یک برنامه می‌تواند به اطلاعات مربوط کلاس‌ها، اشیاء، عملیات[۹۰]، و مشخصه‌ها[۹۱]، مانند نام مشخصه‌های یک کلاس، دسترسی داشته باشد. طبق تعریف ارائه شده در [۳۹] زبانی که از خودنگری پشتیبانی می‌کند یک زبان انعکاسی است. یک برنامه با استفاده از این اطلاعات می‌تواند رفتار پویا داشته باشد. مثلاً یک عمل که شیئی را به‌عنوان پارامتر ورودی دریافت می‌کند، می‌تواند کلاس آن را تشخیص دهد و بر این اساس رفتار متفاوتی برای اشیاء از کلاس‌های متفاوت نشان دهد.

---

[۸۴] Self-Representation
[۸۵] Causally-Connected
[۸۶] Status
[۸۷] Computation
[۸۸] Middleware
[۸۹] Introspection
[۹۰] Methods
[۹۱] Attributes



شکل ۲—۳: معماری سیستم نیترو (منبع: [۲۱]، با کمی تغییر).

۲) **انعکاس ساختاری**[۹۲]: امکان تغییر پویای ساختار برنامه وجود دارد. به‌عنوان مثال اضافه کردن یک مشخصه به یک کلاس نمونه‌ای از انعکاس ساختاری است.

۳) **انعکاس رفتاری**[۹۳] **یا انعکاس محاسباتی**[۹۴]: امکان تغییر پویای رفتار برنامه وجود دارد. به‌عنوان مثال در زبان جاوا، کتاب‌خانهٔ java.l ang.reflection.Proxy [۲۱] چنین مکانیزمی را فراهم می‌آورد.

۴) **انعکاس زبانی**[۹۵]: برخلاف سه سطح قبلی که مربوط به برنامه بودند، این سطح مربوط به خود زبان برنامه‌نویسی است. برای مثال ممکن است به این وسیله بتوان معنی[۹۶] دستورات زبان را تغییر داد.

[۲۱، ۲۰] سیستمی به‌نام نیترو[۹۷] برای انعکاس نامحدود[۹۸] پیشنهاد می‌دهند. با استفاده از این سیستم می‌توان تمام خصوصیات یک زبان برنامه‌نویسی را در زمان اجرا تغییر داد. همانطور که شکل ۲—۳ نشان می‌دهد، در سیستم نیترو روش کار به‌طور خلاصه به این شکل است که در یک موتور تفسیر عمومی (موتور تفسیر عمومی دارای قابلیت انعکاس ساختاری[۹۹]) توصیف زبان L را به‌صورت یک مدل شیء دریافت می‌کند. این موتور تفسیر متن برنامهٔ نوشته شده به زبان L را بر اساس توصیف این زبان اجرا می‌کند. برنامهٔ نوشته شده به این زبان می‌تواند دستوراتی داشته باشد که خود زبان L را تغییر دهد.

[۹۲] Structural Reflection
[۹۳] Behavioral Reflection
[۹۴] Computational Reflection
[۹۵] Linguistic Reflection
[۹۶] Semantic
[۹۷] nitrO
[۹۸] Non-Restrictive Reflection
[۹۹] Structural-Reflective Generic Interpreter Engine

در برخی مقالات از جمله [۴۱] اصطلاح دیگری به‌نام انعکاس معماری[۱۰۰] مورد بحث قرارگرفته‌است که با وجود تشابه نام با انعکاس زبان‌های برنامه‌نویسی تفاوت دارد و مربوط به بازپیکربندی معماری در سیستم‌های محاسبات سیار و محاسبات فراگیر است، که در قسمت ۲—۱ مورد بحث قرار گرفت. سیستم‌های دارای انعکاس معماری نیز، معمولاً، مدلی از معماری خود دارند و رابطهٔ علّی بین مدل و معماری سیستم را حفظ می‌کنند.

همان‌طورکه گفته شد، انعکاس زبان‌های برنامه‌نویسی و معماری از مصادیق خود-تطبیقی هستند نه تطبیق توسط کاربر، اما مکانیزم‌ها و مفاهیم مشابهی بین این روش‌ها وجود دارد. به‌عنوان نمونه، سیستم نیترو یک مثال تمام عیار از کاربرد ماشین مجازی برای تفسیر مدل و تغییر آن در زمان اجرا می‌باشد. مفهوم انعکاس قابل تعمیم به سیستم‌های قابل تطبیق توسط انسان می‌باشد. تنها تفاوت در این است که نمایش-از-خود این سیستم‌ها توسط انسان تغییر داده می‌شود.

۲—۶ جمع‌بندی

در این فصل، در یک طبقه‌بندی تغییرات نرم‌افزار به دو دستهٔ ایستا و پویا (تغییر) تقسیم شد. سپس هدف این پایان‌نامه که به‌طور خلاصه قابلیت تطبیق رفتاری در سیستم‌های قابل تطبیق توسط انسان می‌باشد، بیان گردید، و کارهای مرتبط با این کار مورد بررسی قرار گرفت. در فصل بعد مقدماتی در مورد معماری نرم‌افزار ارائه می‌گردد. و از فصل ۴ حرکت به‌سمت راه‌حل شروع می‌شود. در این تحقیق سبک معماری مدل قابل تطبیق شیء، برای پشتیبانی از تطبیق رفتاری گسترش پیدا می‌کند.

[۱۰۰] Architectural Reflection

فصل ۳

معماری نرم‌افزار: امکانی برای تحقق قابلیت تطبیق

همان‌طورکه قبلاً اشاره شد معماری نرم‌افزار در تحقق ویژگی‌های کیفی نرم‌افزار نقش اساسی دارد. در این پایان‌نامه با استفاده از برخی روش‌های مطرح در معماری سیستم‌های نرم‌افزاری، یک معماری برای سیستم‌های قابل تطبیق عرضه می‌گردد. این فصل به‌طور خلاصه به‌معرفی این مباحث می‌پردازد. نکتهٔ قابل توجه این است که در مورد هیچ یک از این مباحث و به‌طورکلی مباحث علمی، در بین نویسندگان اتفاق نظر وجود ندارد. بااین‌حال، سعی شده است در مباحث این فصل از منابعی که در بین محققین مهندسی نرم‌افزار معتبر محسوب می‌شوند، استفاده شود.

در قسمت ۳—۱ تعریف معماری نرم‌افزار ارائه می‌گردد. در قسمت ۳—۲ مفاهیم تاکتیک، تکنیک، الگو، و سبک معماری معرفی می‌گردند. قسمت ۳—۳ در مورد استفادهٔ مجدد و ایجاد نرم‌افزار برای خانواده‌ای از محصولات بحث می‌کند. قسمت ۳—۴ سایر روش‌های مطرح در زمینهٔ معماری نرم‌افزار برای تولید نرم‌افزارهای واحدمند را بررسی می‌کند. قسمت ۳—۵ مباحث این فصل را جمع‌بندی می‌کند.

۳—۱ تعریف معماری نرم‌افزار

مفهوم معماری نرم‌افزار، از ابتدای تولید نرم‌افزار وجود داشته، اما در حدود پانزده سال است که این نام برای آن متداول شده است. به‌طور غیررسمی معماری یک سیستم نرم‌افزاری عبارت است از ساختار کلی آن، اما تعاریف دقیق‌تری نیز از این مفهوم وجود دارد. در [۱]، معماری نرم‌افزار را اینگونه تعریف شده است:

> معماری نرم‌افزار یک برنامه یا یک سیستم کامپیوتری عبارت است از ساختار[۱] یا ساختارهایی از سیستم که از اجزاء[۲] سیستم، خصوصیاتی از این اجزا که از خارج آن‌ها قابل مشاهده است، و ارتباطات آن‌ها تشکیل می‌شود.

---

[۱] Structure

[۲] Elements

ساختار (یا دید[۳]) که در این تعریف آمده را می‌توان با یک نمودار که جنبه‌ای مشخص (مثلاً نحوهٔ استقرار مؤلفه‌های نرم‌افزاری بر روی پردازنده‌های سخت‌افزاری) از سیستم را نشان می‌دهد، نمایش داد. منظور از خصوصیت قابل مشاهده از خارج، خصوصیتی از یک جزء است که از نظر سایر اجزاء سیستم (که در بیرون آن قرار دارند و با آن در ارتباط هستند) اهمیت دارد، مانند این‌که جزء چه سرویسی ارائه می‌دهد و واسط آن چیست. از این تعریف نکات زیر نتیجه می‌شوند:

- هر سیستمی که نرم‌افزار در آن وجود دارد، حتماً، معماری نرم‌افزار دارد چه مستند شده باشد و چه نشده باشد؛ چه در زمان تولید سیستم فعالیت مشخصی به‌نام طراحی معماری وجود داشته باشد و چه وجود نداشته باشد. دلیل این مطلب این است که هر سیستم نرم‌افزاری از اجزاء مرتبط تشکیل می‌شود (در ابتدایی‌ترین حالت یک سیستم از یک مؤلفه تشکیل می‌گردد.) بنابراین، نحوهٔ ارتباط و سرویس‌دهی این اجزاء که در یک یا چند ساختار قابل نمایش است، معماری نرم‌افزار می‌باشد.
- معماری یک سیستم ممکن است از چندین ساختار تشکیل شده باشد.
- معماری بر یک دید تجریدی از سیستم تأکید دارد. به‌عبارت‌دیگر، بر حسب معماری مسائل سیستم را می‌توان به دو دسته تقسیم کرد:

۱) مسائل مهم در سطح معماری[۴]: مسائلی که در سطح تجرید معماری توجه به آن‌ها ضروری است، مثل ویژگی‌های کیفی قابلیت تغییر و قابلیت تطبیق.

۲) مسائل غیرمهم در سطح معماری[۵]: مسائلی که در سطح تجرید معماری، لازم نیست و بلکه نباید به آن‌ها توجه کرد مثل الگوریتم داخل یک جزء برای ارائهٔ یک سرویس.

رفتار یک جزء تاجایی مهم در سطح معماری است که از دید سایر اجزاء اهمیت داشته باشد.

توجه به معماری نرم‌افزار برای دست‌یابی به یک نرم‌افزار با کیفیت اهمیت کلیدی دارد [۱]. همان‌طورکه گفته شد، نمی‌توان ویژگی‌های کیفی نرم‌افزار را پس از ایجاد کارکردهای[۶] سیستم به آن تزریق کرد، بلکه باید در مراحل اولیهٔ تولید نرم‌افزار برای آن برنامه‌ریزی کرد. معماری نرم‌افزار جایی است که طراحان سیستم باید در مورد چگونگی برآورده کردن این ویژگی‌ها تصمیم‌گیری کنند. به‌عبارت‌دیگر، اغلب ویژگی‌های کیفی از جمله قابلیت تغییر و قابلیت تطبیق از مسائل مهم در سطح معماری هستند. در فرآیندهای جدید تولید نرم‌افزار (مثل RUP)، بخشی از فرآیند به طراحی و ارزیابی معماری اختصاص دارد. همان‌طورکه در قسمت ۳—۲ توضیح داده می‌شود، برخی از سبک‌ها و الگوهای معماری، به‌عنوان راه‌حلی برای ایجاد سیستم‌های قابل تطبیق به‌وجود آمده‌اند.

۳—۲ استفادهٔ مجدد از راه‌حل‌های معماری نرم‌افزار

بخشی از تلاش‌ها در حوزهٔ معماری نرم‌افزار، معطوف به مستندسازی و استفادهٔ مجدد از راه‌حل‌هایی می‌شود که مهندسین نرم‌افزار در معماری برخی سیستم‌ها با موفقیت استفاده کرده‌اند. این راه‌حل‌ها در سطوح

---

[۳] View

[۴] Architecturaly Significant

[۵] Architecturaly Insignificant

[۶] Functionalities

مختلفی از تجرید قرار دارند. برخی راه‌حل‌ها، یک راهنمای[۷] کلی برای طراحی گروهی از سیستم‌ها هستند، درحالی‌که برخی دیگر ممکن است راهنمایی برای استفاده از ساختارهای موجود در یک روش برنامه‌نویسی، مثل شیءگرا، باشند. بعضی دیگر از این راه‌حل‌ها یک مسئلهٔ مشخص طراحی یا معماری را حل می‌کنند. در این قسمت تعاریفی در این رابطه ارائه می‌گردد.

تاکتیک: عبارت است از یک تصمیم طراحی که در ایجاد یک ویژگی کیفی در نرم‌افزار مؤثر است [۱]. به‌عنوان مثال، تکرار مؤلفه‌ها تاکتیکی است که برای بالا بردن ویژگی در دسترس بودن سیستم قابل استفاده است.

تکنیک: عبارت است از یک تصمیم طراحی مبتنی بر مکانیزم‌های یک روش تولید نرم‌افزار مثل شیءگرایی. به‌عنوان مثال، نمایندگی[۸] تکنیکی است که با استفاده از آن ترکیب[۹]، از نظر استفادهٔ مجدد به‌قدرت‌مندی وراثت می‌شود [۲۸]. تکنیک‌ها در سطح تجرید پایین‌تری نسبت به تاکتیک‌ها هستند و می‌توان آن‌ها را برای پیاده‌سازی تاکتیک‌ها به‌کار برد.

نوشته‌های متعددی به معرفی تاکتیک‌ها و تکنیک‌های مناسب برای پیاده‌سازی سیستم‌های مختلف پرداخته‌اند. در قسمت ۴–٦ تاکتیک‌ها و تکنیک‌هایی برای طراحی سیستم‌های قابل تطبیق ارائه می‌گردد.

الگو: مفهوم دیگری که در سطح وسیع‌تری برای مستند کردن راه‌حل‌های طراحی سیستم‌های نرم‌افزاری به‌کارگرفته شده است، مفهوم الگو می‌باشد. حدود ده سال است که مفهوم الگو[۱۰] به مهندسی نرم‌افزار وارد شده است[۱۱]. یک الگو عبارت است از یک راه‌حل مشخص با خصوصیات شناخته شده برای یک مسئلهٔ مشخص که در جاهای مختلف تکرار می‌شود. الگوها در سطوح مختلف و برای کاربردهای مختلفی تعریف شده‌اند. از جمله:

- الگوهای حرفه[۱۲] [۴۲]
- الگوهای فرآیند[۱۳] [۴۶، ۴۵، ۴۴، ۴۳]
- الگوهای تحلیل[۱۴] [۱٦]
- الگوهای معماری[۱۵] [۱۸]
- الگوهای طراحی[۱٦] [۲۸]
- الگوهای پیاده‌سازی[۱۷] [۱۸]

[۷] Guideline
[۸] Delegation
[۹] Composition
[۱۰] Pattern
[۱۱] این مفهوم اولین بار در معماری ساختمان مطرح شده است و از آنجا به مهندسی نرم‌افزار آمده است [۲۸].
[۱۲] Business Patterns
[۱۳] Process Patterns
[۱۴] Analysis Patterns
[۱۵] Architecural Patterns
[۱٦] Design Patterns
[۱۷] Idioms

الگوهای تحلیل، معماری، و طراحی برای استفاده توسط معماران نرم‌افزار مناسب هستند و در ادامه، بحث بر روی این الگوها متمرکز می‌شود. فرق الگو با تاکتیک این است که کاربرد آن مشخص‌تر است و در نتیجه دامنهٔ مسائلی که به آن می‌پردازد کوچک‌تر است. ممکن است یک الگو چند تاکتیک را محقق کند و یک تاکتیک ممکن است توسط الگوهای مختلفی محقق شود. معمولاً در الگوهای طراحی از تکنیک‌ها استفاده می‌شود. به‌عبارت‌دیگر، الگوی طراحی کاربرد خاصی از یک سری تکنیک برای حل یک مسئله می‌باشد. الگوها ممکن است عمومی یا وابسته به یک دامنهٔ خاص باشند. کتاب‌های [۴۶، ۴۵، ۴۴، ۴۳] برخی الگوهای مربوط به دامنه‌های مختلف، الگوهای وابسته به زبان‌های برنامه‌نویسی، و الگوهای مربوط به جنبهٔ خاصی از نرم‌افزار مثل ماندگاری[۱۸] و توزیع[۱۹] را مستند کرده‌اند. اکثر الگوها در جامعهٔ مهندسی نرم‌افزار شیءگرا مطرح و در نتیجه با استفاده از مفاهیم شیءگرا مستند شده‌اند. همان‌طورکه در قسمت ۲—۵—۱ گفته شد، در طراحی نرم‌افزارهای مبتنی بر مدل قابل تطبیق شیء، از الگوهای مختلفی استفاده می‌شود، مانند:

- الگوی طراحی شیء نوع[۲۰] [۱۱]
- الگوی طراحی مشخصه[۲۱] [۱۱]
- الگوی طراحی راهبرد[۲۲] [۲۸]
- الگوی طراحی مرکّب[۲۳] [۲۸]
- الگوی طراحی شیء قاعده[۲۴] [۱۷]
- الگوی طراحی مفسّر[۲۵] [۲۸]
- الگوی طراحی سازنده[۲۶] [۲۸]
- الگوی تحلیل مشاهده[۲۷] [۱۶]
- الگوی معماری انعکاس[۲۸] [۱۸]

سبک معماری: در مورد سبک معماری و یکی بودن یا نبودن آن با الگو مؤلفین مختلف نظرات مختلفی دارند. باتوجه‌به‌این‌که عقیدهٔ [۴۷] در این مورد صحیح‌تر به‌نظر می‌رسد، همین نظر در این‌جا بیان می‌شود. طبق این نظر، استفاده از یک سبک معماری برای بهبود یک یا چند ویژگی کیفی در نرم‌افزار مؤثر است و کل معماری نرم‌افزار را تحت تأثیر قرار می‌دهد، درحالی‌که یک الگوی معماری یا طراحی بخشی از معماری نرم‌افزار را تحت تأثیر قرار می‌دهند. همچنین، می‌توان به‌طور هم‌زمان در یک سبک از الگوهای مختلفی استفاده کرد. طبق [۱]، یک سبک معماری این ویژگی‌ها را مشخص می‌کند:

[۱۸] Persistence
[۱۹] Distribution
[۲۰] Type-Object Design Pattern
[۲۱] Property Design Pattern
[۲۲] Strategy Design Pattern
[۲۳] Composite Design Pattern
[۲۴] Rule-Object Design Pattern
[۲۵] Interpreter Design Pattern
[۲۶] Builder Design Pattern
[۲۷] Observation Analysis Pattern
[۲۸] Reflection Architectural Pattern

۱) مجموعه‌ای از انواع اجزاء

۲) یک طرح توپولوژیکی [باتوجه‌به نظر مورد قبول نویسنده این مورد لازم نیست در سبک معماری مشخص و ثابت باشد]

۳) مجموعه‌ای محدودیت‌های معنایی[۲۹]

۴) مجموعه‌ای از مکانیزم‌های ارتباطی بین این اجزاء

کار این پایان‌نامه براساس سبک معماری مدل قابل تطبیق شیء بنا شده است.

۳–۳ ایجاد نرم‌افزار برای خانواده‌ای از محصولات

در قسمت قبل استفادهٔ مجدد در سطوح مفهومی (مثل طراحی) مورد بحث قرار گرفت. این قسمت به استفادهٔ مجدد در سطح فیزیکی (محصولات نرم‌افزاری) می‌پردازد. تابه‌حال، بیشتر نرم‌افزارها به‌صورت محصولات منفرد[۳۰] تولید شده‌اند. در این مشی استفادهٔ مجدد بسیار دشوار است و مهندسین نرم‌افزار یک سازمان یا سازمان‌های مختلف همواره در حال اختراع مجدد چرخ هستند [۴۸]. برای رفع این مشکل، صنعت نرم‌افزار کم‌کم به‌روش‌های تولید نرم‌افزار برای خانواده‌ای (یا دامنه‌ای) از محصولات روی آورده است. البته ایدهٔ این موضوع در دههٔ ۷۰ مطرح شده بود [۴]، اما این ایده در سال‌های اخیر با معرفی مفاهیمی چون چارچوب‌های شئ‌گرا[۳۱] [۴۹، ۲۲] و خطوط تولید نرم‌افزار[۳۲] [۵۲، ۵۱، ۴۷، ۵۰، ۴۸] محقق شد.

از ابتدای پیدایش صنعت نرم‌افزار تلاش برای پیدا کردن روش‌هایی برای بهبود استفادهٔ مجدد شروع شد. در اواخر دههٔ ۶۰ ایدهٔ ساختن سیستم با ترکیب مؤلفه‌ها مطرح شد. در دههٔ ۷۰ این تلاش‌ها به پیدایش روش‌های برنامه‌نویسی واحدمند[۳۳] انجامید. در این روش برای استفادهٔ مجدد از یک واحد[۳۴] باید آن‌را به‌تمامی استفاده کرد یا در متن برنامهٔ آن تغییر ایجاد کرد. در دههٔ ۸۰ روش شیء گرا به‌وجود آمد که در آن کلاس‌ها واحد استفادهٔ مجدد، و وراثت و ترکیب مکانیزم‌های آن هستند. بااین‌حال، هنوز استفادهٔ مجدد در سطح فردی و اغلب از طریق مؤلفه‌های کوچک—مقیاسی[۳۵]، که اجزای سیستم‌های بزرگ‌تر را به‌وجود می‌آورند، صورت می‌گرفت، مانند انواع داده—ساختارها. در این روش‌ها مشکل اصلی استفادهٔ مجدد که استفاده از مؤلفه‌های بزرگ—مقیاسی[۳۶] که قسمت عمدهٔ سیستم را می‌سازند و جنبه‌های زیادی از آن‌ها قابل تطبیق باشد مورد توجه نبود. در اواخر دههٔ ۸۰ چارچوب‌های شیءگرا برای رفع این مشکل معرفی شدند [۴۹]. همچنین، به‌طور موازی روش‌های مبتنی بر مؤلفه[۳۷] در حال عرضه شدن بودند [۴۷]. در دههٔ ۹۰ مفهوم خط تولید نرم‌افزار، که حاصل ترکیب معماری نرم‌افزار و روش‌های مبتنی بر مؤلفه می‌باشد [۴۷] و

[۲۹] Constraint Semantics
[۳۰] Stand Alone Product or One of a Kind Product
[۳۱] Object Oriented Frameworks
[۳۲] Software Product-Lines
[۳۳] Modular Programming
[۳۴] Module
[۳۵] Small-scale Components
[۳۶] Large-scale Component
[۳۷] Component-based

می‌توان آن‌را حالت عمومی چارچوب‌های شیءگرا در نظر گرفت، به‌کارگرفته‌شد و مورد تحقیقات وسیعی قرار گرفت.

در طراحی سیستم‌هایی که قابلیت ایجاد خانواده‌ای از محصولات را دارند پیش از مهندسی محصول[۳۸] مهندسی دامنه[۳۹] صورت می‌گیرد. مهندسی دامنه عبارت است از شناسایی نیازمندی‌های تمام اعضای دامنه و طراحی یک سیستم زیربنایی که برای ایجاد هر یک از اعضا مناسب باشد. این فرآیند، معمولاً، در مورد دامنه‌هایی امکان پذیر است که به بلوغ رسیده‌اند.

خط تولید نرم‌افزار سیستمی است که برای ایجاد[۴۰] و راه‌اندازی محصولات یک خانواده (یک دامنهٔ مشخص) از نرم‌افزارها به‌کار گرفته می‌شود. سیستم‌های خط تولید از نظر بلوغ در سطوح مختلفی قرار دارند [۵۱]. بعضی تنها زیرمجموعه‌ای از مؤلفه‌های مورد نیاز در اعضای خانواده را دارند و باید بقیهٔ قسمت‌ها را با برنامه‌نویسی ایجاد کرد. درحالی‌که بعضی دیگر از مناسب‌سازی[۴۱] تمام مؤلفه‌های تمام اعضای خانواده پشتیبانی می‌کنند و به‌هیچ‌وجه نیاز به برنامه‌نویسی برای ایجاد یک محصول ندارند.

یک چارچوب شیءگرا مجموعه‌ای است از کلاس‌ها (واسط‌ها[۴۲]، کلاس‌های مجرد[۴۳]، و کلاس‌های واقعی[۴۴]) که راه‌حل مجردی برای خانواده‌ای از محصولات نرم‌افزاری را در خود دارد [۴۹، ۲۲]. بنابراین، چارچوب بخشی از طراحی و پیاده‌سازی یک برنامهٔ کاربردی را ارائه می‌دهد و از طریق برنامه‌نویسی (وراثت، ترکیب، نمایندگی و ...) در یک محصول مورد استفاده قرار می‌گیرد [۴۹]. یک چارچوب ممکن است جعبه‌سیاه[۴۵] باشد (که در آن چگونگی عملکرد کلاس‌ها برای برنامه‌نویس مشخص نیست و او فقط اشیائی از کلاس‌های موجود ایجاد و استفاده می‌کند) یا جعبه‌سفید[۴۶] (که در آن برنامه‌نویس از پیاده‌سازی کلاس‌ها مطلع است و با استفاده از وراثت کلاس‌های خود را بر اساس کلاس‌های چارچوب ایجاد می‌کند) یا ترکیبی از این دو [۴۹]، اما به‌هرحال، لازمهٔ استفاده از آن‌ها برنامه‌نویسی است. یک چارچوب ممکن است تنها به جنبهٔ خاصی (مثلاً واسط کاربر) از یک سیستم بپردازد [۲۲]، بنابراین، در بسیاری موارد در ایجاد یک محصول از چند چارچوب استفاده می‌شود. چارچوب‌ها را می‌توان روشی برای پیاده‌سازی خط تولید نرم‌افزار در نظر گرفت هر چند روش‌های دیگری نیز وجود دارد.

سیستم‌های قابل تطبیق توسط انسان برای ایجاد محصولاتی از یک خانواده قابل استفاده هستند، مانند سیستم‌های آرگو [۲۹] و آبجکتیوا [۱۱]، و طبق بعضی از تعاریف موجود برای خط تولید می‌توان آن‌ها را خط تولید دانست. بااین‌حال، این سیستم‌ها از جهاتی با تصور متداول از خط تولید متفاوتند:

- در ایجاد این سیستم‌ها شناخت فراگیر دامنه صورت نمی‌گیرد و تنها کافی است الگوی کلی موجودیت‌ها و عملیات (متا مدل منطق حرفه) شناسایی شده و بر اساس آن یک سیستم کلی به‌وجود آید که امکان تعریف تکاملی مدل واقعی محصول را فراهم می‌کند. به‌عبارت‌دیگر، در مدل سیستم موجودیت‌های مخصوص کاربرد کمتر دیده می‌شوند و به‌جای آن‌ها موجودیت‌های مجرد دامنه مدل‌سازی می‌شوند.

[۳۸] Product Engineering
[۳۹] Domain Engineering
[۴۰] Product Instantiation
[۴۱] Customization
[۴۲] Interfaces
[۴۳] Abstract Classes
[۴۴] Concrete Class
[۴۵] Blackbox
[۴۶] Whitebox

- تنظیم سیستم برای یک محیط خاص، تکاملی[۴۷] است و از طریق تطبیق توسط کاربر صورت می‌گیرد و نه آنی.
- مکانیزم ایجاد یک محصول مشخص در این روش بیشتر بر اساس تغییر متادادهٔ سیستم است نه تولید[۴۸] سیستم یا برنامه‌نویسی قسمت‌هایی که در خط تولید پشتیبانی نمی‌شوند. به‌عبارت‌دیگر، در این روش محصول و خط تولید دو سیستم جدا از هم نیستند و خط تولید مفسر مدل محصول است.

بااین‌که برخی سیستم‌های قابل تطبیق چارچوب نامیده شده‌اند [۱۰، ۲۹]، طبق تعریف بالا نمی‌توان یک سیستم قابل تطبیق ایده‌آل را یک چارچوب قلمداد کرد زیرا نکتهٔ اساسی در این سیستم‌ها تکامل بدون برنامه‌نویسی است. اما در عمل به‌دلیل پیچیدگی‌های موجود در مدل‌سازی، مخصوصاً در بعد رفتاری، این سیستم‌ها به‌صورت ترکیبی پیاده‌سازی شده‌اند. به‌عبارت‌دیگر، بخش‌هایی از سیستم قابل تطبیق هستند، درحالی‌که تغییر بخش‌های دیگری از سیستم نیازمند برنامه‌نویسی است. هر چه این سیستم‌ها به برنامه‌نویسی نزدیک‌تر شوند از هدف خود دور شده‌اند.

ازاین‌پس، به خطوط تولید نرم‌افزار قابل تطبیق، موتور تطبیق می‌گوییم. در این پایان‌نامه راه‌حلی برای معماری موتورهای تطبیق دارای تطبیق ساختاری و رفتاری، توضیح داده می‌شود.

۳—۳—۱ دامنه و مهندسی دامنه

مفهوم کلمهٔ دامنه شبیه مفهوم آن در ریاضیات است و معنای مجموعه را در خود دارد. پیش از تعریف دامنه باید مفهوم محیط[۴۹] معرفی گردد.

محیط: محل مشخصی (مثل شرکت بیمهٔ الف یا آموزش دانشگاه ب) که سیستم نرم‌افزاری برای آن نوشته می‌شود، محیط آن نرم‌افزار نامیده می‌شود.

دامنه: به مجموعه‌ای از محیط‌های شبیه به هم، دامنه گفته می‌شود.

در بعضی نوشته‌ها به‌جای کلمهٔ محیط از کلمات سیستم یا سازمان استفاده می‌شود. در برخی موارد مفهوم دامنه از روی تسامح به جای محیط استفاده می‌شود که صحیح نیست. همچنین، گاهی به مجموعه‌ای از سیستم‌های نرم‌افزاری دامنه گفته می‌شود (مانند دامنهٔ سیستم‌های مبتنی بر وب).

بنابر این تعریف، هر دامنه یک مجموعه است و خصوصیات آن شبیه خصوصیات یک مجموعه ریاضی است. به‌عنوان مثال، ممکن است یک دامنه زیرمجموعهٔ یک دامنهٔ دیگر باشد. ملاک‌های متفاوتی برای شباهت بین محیط‌ها وجود دارد، به‌همین‌دلیل، به‌شکلهای مختلفی می‌توان محیط‌ها را در دامنه‌ها طبقه‌بندی کرد. یک محیط با ملاک‌های مختلف به دامنه‌های مختلفی تعلق می‌گیرد. دو نوع روش طبقه‌بندی محیط‌هادر دامنه‌ها عبارتند از:

- طبقه‌بندی براساس حرفه: مثل دامنهٔ سیستم‌های حساب‌داری، دامنهٔ سیستم‌های پزشکی، و دامنهٔ سیستم‌های آموزش الکترونیکی

---

[۴۷] Evolutionary

[۴۸] Generation

[۴۹] Environment

- طبقه‌بندی براساس نوع سیستم نرم‌افزاری: دامنهٔ سیستم‌های جریان‌کار، دامنهٔ سیستم‌های مبتنی بروب، دامنهٔ سیستم‌های توزیع شده

همان‌طورکه قبلاً اشاره شد در فرآیند ایجاد سیستم‌هایی برای خانواده‌ای از محصولات، مرحله‌ای به‌نام مهندسی دامنه[۵۰] وجود دارد. هدف از این مرحله درک نیازمندی‌های سیستم‌های مناسب برای محیط‌های آن دامنه و ایجاد یک ابزار زیربنایی (مثل موتور تطبیق) می‌باشد. این ابزار، که با استفاده از تحلیل اشتراک‌ها و افتراق‌های بین محیط‌های دامنه ایجاد می‌شود، در سطح تجرید بالایی قرار دارد و ایجاد محصولات مورد نیاز در محیط‌های دامنه را تسهیل می‌کند.

۳–۳–۲ زبان‌های مخصوص دامنه

در سیستم‌های نرم‌افزاری بسیاری از کارها از طریق زبان‌ها انجام می‌شود. در این‌جا منظور از زبان، تنها زبان‌های طبیعی یا زبان‌های برنامه‌نویسی همه‌منظوره نیست. بلکه، هر نظام نمادگذاری‌ای که نحو[۵۱] و معانی[۵۲] آن مشخص باشد یک زبان است، حتی اگر گرامر آن به‌طور صریح مستند نشده باشد. برای مثال، مجموعهٔ ورودی‌ها یا خروجی‌های یک برنامهٔ کامپیوتری را می‌توان یک زبان دانست. لازم نیست یک زبان حتماً متنی باشد (مثلاً UML یک زبان مدل‌سازی تصویری[۵۳] است) یا متن نوشته شده در آن زبان حتماً در یک فایل متنی ذخیره شود (ورودی ممکن است از طریق یک فرم به سیستم داده شود). بنابراین، می‌توان ادعا کرد هر چیزی که در دنیای نرم‌افزار وجود دارد، حتی دادهٔ یک برنامه، یک گرامر دارد و در نتیجه یک زبان است. اما گستردگی دامنهٔ همهٔ زبان‌ها یکسان نیست. برخی همه منظوره هستند (مثل UML که یک زبان مدلسازی همه منظوره است [۵۳]) و برخی کاربرد بسیار محدودی دارند که به آن‌ها زبان مخصوص دامنه[۵۴] گفته می‌شود.

یک زبان مخصوص دامنه این طور تعریف می‌شود [۵۴]: یک زبان برنامه‌نویسی یا یک زبان توصیفی قابل اجرا، که از طریق نمادگذاری و تجریدهای مناسب، قدرت بیانی متمرکز بر یک دامنهٔ خاص و معمولاً محدود به آن دامنه فراهم می‌آورد.

از این زبان‌ها می‌توان برای برنامه‌نویسی توسط کاربر نهایی[۵۵] استفاده کرد. در سیستم‌های قابل تطبیق می‌توان منطق حرفه را با زبان‌های مخصوص دامنه توصیف کرد. بنابراین، مدل قابل تطبیقی که پیش از این در مورد آن صحبت شد با یک زبان مخصوص دامنه بیان می‌شود و سیستم از طریق تفسیر این مدل، رفتار مورد نظر را نشان می‌دهد. از جمله اهداف در این سیستم‌ها، داشتن زبانی قابل فهم برای متخصصین دامنه است. ازآنجاکه زبان‌های مخصوص دامنه در سطح تجرید نزدیک‌تری نسبت به دامنه قرار دارند، و گستردگی کمتری دارند، کار کردن با آن‌ها ساده‌تر است. همچنین، اگر بتوان یک زبان را تصویری[۵۶] نمایش داد، درک آن ساده‌تر است[۵۷] [۵۴، ۵۵].

---

[۵۰] Domain Engineering

[۵۱] Syntax

[۵۲] Semantics

[۵۳] Visual Modeling Language

[۵۴] Domain-Specific Language

[۵۵] End-User Programming

[۵۶] Visual

[۵۷] البته این جمله با کمی تسامح در بیان مطرح شده است، زیرا متن خود نوعی نمایش تصویری است که عناصر تشکیل دهندهٔ آن

۳—۴ سایر روش‌ها

در سال‌های اخیر بسیاری از روش‌ها و جهان‌بینی‌های [۵۸] جدید در تولید نرم‌افزار مطرح شده‌اند. به‌وجودآورندگان این روش‌ها انگیزه‌های مختلفی برای طراحی این روش‌ها برمی‌شمارند، مثل: جبران نقص‌های جهان‌بینی شیءگرا [۱۲، ۵۶]، بهبود تولید نرم‌افزار برای خانواده‌ای از سیستم‌ها [۴۸]، یا بهبود واحدمندی [۵۹] [۱۳] که به تسهیل ایجاد، تغییرات، و استفادهٔ مجدد نرم‌افزار کمک می‌کند. برخی از این روش‌ها عبارتند از برنامه‌نویسی جنبه‌گرا [۱۲]، برنامه‌نویسی موضوع‌گرا [۶۰] [۵۶]، برنامه‌نویسی ویژگی‌گرا [۱۳]، متامدل‌سازی [۲۷، ۱۴]، برنامه‌نویسی تطبیقی [۶۱] [۵۷]، مؤلفه‌های تطبیقی اتصالی [۶۲] [۵۸]، و جداسازی چندبعدی دغدغه‌ها [۶۳] [۱۳].

این روش‌ها برای پشتیبانی از تغییرات در زمان اجرا به‌وجود نیامده‌اند و از قابلیت تطبیق توسط انسان پشتیبانی نمی‌کنند. شاهد بر این مطلب این است که برخی از این روش‌ها ماهیت زبانی [۶۴] دارند و تغییرات و توسعه‌هایی در زبان‌های برنامه‌نویسی پیشنهاد می‌کنند (مثل برنامه‌نویسی جنبه‌گرا)، برخی دیگر ماهیت تولیدی دارند (مثل متامدلسازی) و ابزاری برای تسهیل ایجاد و اصلاح برنامه‌ها به‌شمار می‌روند، بعضی راه‌حلی برای طراحی بهتر برنامه‌ها (نحوهٔ تجرید و تقسیم به واحدها، و جداسازی دغدغه‌ها) ارائه می‌دهند، گروهی نیز ترکیبی از این موارد را در خود دارند. اما به‌هرحال، در تمام این روش‌ها تغییرات از طریق برنامه‌نویسی در سیستم اعمال می‌گردد. حتی در مواردی که اتصال مؤلفه‌های جدید بدون تغییر سایر قسمت‌های سیستم مورد پشتیبانی قرار می‌گیرد، مؤلفه‌های جدید خود باید از طریق برنامه‌نویسی ایجاد شوند. بنابراین، نمی‌توان از این روش‌ها به‌تنهایی برای ایجاد نرم‌افزارهای قابل تطبیق استفاده کرد اما می‌توان از آن‌ها در کنار روش‌های مختص نرم‌افزارهای قابل تطبیق مثل مدل قابل تطبیق شیء (قسمت ۲—۵) استفاده کرد. دراین‌صورت، این روش‌ها می‌توانند ایجاد و نگه‌داری خود موتور تطبیق را ساده‌تر کنند. باتوجه‌به نامربوط بودن این مباحث، این پایان‌نامه بیش از این به آن‌ها نمی‌پردازد.

۳—۵ جمع‌بندی

در این فصل برخی مفاهیم مرتبط با معماری نرم‌افزار مورد بررسی قرار گرفت. برای حل مسئلهٔ تطبیق و به‌طور خاص تطبیق رفتاری، معمولاً لازم است دامنهٔ خاصی در نظرگرفته شود و یک موتور تطبیق برای آن دامنه ایجاد گردد. موتور تطبیق یک زبان مخصوص دامنه دارد که متخصص دامنه با استفاده از آن ویژگی‌های سیستم را تعریف می‌کند و تطبیق می‌دهد. برای مستند کردن راه‌حل ارائه شده در این پایان‌نامه از

حروف الفبا هستند. این ادعا را این‌گونه می‌توان اصلاح کرد: کار کردن با نمادگذاری‌هایی که ما به‌نام تصویری می‌شناسیم به این دلیل که در سطح تجرید پایین‌تری قرار دارند (به‌عبارت‌دیگر، عناصر تشکیل دهندهٔ آن‌ها پیچیده‌تر از حروف سادهٔ الفبا هستند) ساده‌تر است. این همان دلیلی است که برای ترجیح زبان‌های مخصوص دامنه بر زبان‌های همه منظوره بیان شد.

[۵۸] Paradigms

[۵۹] Modularity

[۶۰] Subject Oriented Programming

[۶۱] Adaptive Programming

[۶۲] Adaptive Plug and Play Components (APPC)

[۶۳] Multi Dimentional Separation of Concerns

[۶۴] Linguistic

مفاهیم تاکتیک، تکنیک، و الگو، که در این فصل مورد بحث قرار گرفتند، استفاده خواهد شد.

فصل ۴

مبانی ایجاد سیستم‌های قابل تطبیق توسط انسان

در این فصل با شروع از صورت مسئله، تعاریف و نظرات نویسنده در مورد مسائل مختلف مرتبط با سیستم‌های قابل تطبیق مطرح می‌شود. همچنین، اهداف سیستم‌های قابل تطبیق مورد تحلیل قرار می‌گیرد تا یک راهبرد مناسب برای ایجاد سیستم‌های قابل تطبیق به‌دست آید. قسمت ۴–۱ به تحلیل صورت مسئله، تفکیک مسائل مرتبط، و ارائهٔ برخی تعاریف می‌پردازد و نشان می‌دهد چرا داشتن یک موتور تطبیق برای ایجاد یک سیستم قابل تطبیق ضروری است. قسمت ۴–۲ اهداف یک موتور تطبیق را ارائه داده و سبک-سنگین می‌کند. قسمت ۴–۳ سبک معماری سیستم‌های قابل تطبیق توسط انسان را توضیح می‌دهد. در قسمت ۴–۴ چرخهٔ حیاتی برای سیستم‌های قابل تطبیق ارائه می‌گردد. روش معمول در مدل‌سازی شیءگرا به‌دلایلی که در ادامهٔ فصل مورد بحث قرار می‌گیرد، نمی‌تواند پاسخگوی نیازهای یک موتور تطبیق باشد. بنابراین، در مدل‌سازی سیستم‌های قابل تطبیق توسط انسان نحوهٔ به‌کارگیری مفاهیم شیءگرا کمی متفاوت با روش معمول است. به‌عبارت دیگر، مدل‌سازی یک موتور تطبیق نیازمند تاکتیک‌های جدیدی است که به‌شکل جدیدی از طراحی شیءگرا می‌انجامد. در قسمت ۴–۵ از این فصل در مورد علت قابل تطبیق نبودن سیستم‌های که با روش متداول طراحی شیءگرا، ایجاد شده‌اند، توضیحاتی ارائه می‌گردد. سپس، در قسمت ۴–٦ تاکتیک‌هایی برای دست‌یابی به قابلیت تطبیق و تکنیک‌هایی برای تحقق آن‌ها در سیستم‌های شیءگرا عرضه می‌شود. قسمت ۴–۷ در یک مثال، استفاده از الگوها را برای پیاده‌سازی تاکتیک‌ها و تکنیک‌ها، در سبک معماری مدل قابل تطبیق شیء نشان می‌دهد. قسمت ۴–۸ دلایل دشواری تطبیق رفتاری نسبت به تطبیق ساختاری را بررسی می‌کند. در قسمت ۴–۹، مباحث این فصل جمع‌بندی می‌شود.

۴–۱ تحلیل صورت مسئله

در قسمت ۲–۴ گفته شد که صورت مسئلهٔ این پایان‌نامه، ارائهٔ روشی برای ایجاد سیستم‌های قابل تطبیق است که:

۱) امکان تطبیق رفتاری در منطق حرفه (در سطح عملیات بر روی موجودیت‌های حرفه و نه جریان کار را به‌وجود آورد.

۲) تطبیق توسط متخصص دامنه قابل اعمال باشد.

۳) تطبیق در زمانی از مرتبهٔ یک ساعت قابل انجام باشد.

پیش از دست‌یابی به یک راه‌حل مناسب، لازم است مقدماتی ارائه گردد.

۴—۱—۱ مفهوم ویژگی و ویژگی قابل تطبیق

پیش ازاین، در قسمت ۲—۲ به مفهوم ویژگی قابل تطبیق اشاره شد. در این قسمت چند تعریف مرتبط با این مفهوم ارائه می‌گردد.

ویژگی[۱]: هر جزء یا خاصیت کارکردی از سیستم یک ویژگی از سیستم است.

برای مثال یک موجودیت، یک مشخصهٔ یک موجودیت، نامی که در واسط کاربر برای یک موجودیت به‌کار می‌رود، یا دسترسی‌های یک کاربر خاص را می‌توان ویژگی‌های سیستم دانست. در این تعریف عمداً سعی شده است حد و مرز سخت‌گیرانه‌ای برای ویژگی مشخص نشود. یک ویژگی ممکن است ساده باشد یا مرکب.

ویژگی‌ساده[۲]: یک ویژگی که از یک دید بیرونی به سیستم (دید کاربر)، ویژگی دیگری در خود نداشته باشد[۳].

ویژگی‌مرکب[۴]: یک ویژگی که از دید کاربر، از ویژگی‌های دیگری تشکیل شده باشد.

به‌عنوان مثال نوع دادهٔ دومین مشخصهٔ موجودیت a از سیستم یک ویژگی ساده است. اما خود موجودیت a یک ویژگی مرکب است که مشخصه‌ها و اعمال آن اجزای آن هستند.

ویژگی ساده قابل تطبیق[۵]: یک ویژگی ساده که در زمان اجرا قابل تغییر باشد[۶].

ویژگی کاملاً قابل تطبیق[۷]: یک ویژگی سادهٔ قابل تطبیق، یک ویژگی کاملاً قابل تطبیق است. همچنین، یک ویژگی مرکب که تمام ویژگی‌های تشکیل دهندهٔ آن کاملاً قابل تطبیق باشند، یک ویژگی کاملاً قابل تطبیق از سیستم است[۸].

---

[۱] Feature

[۲] Simple Feature

[۳] اگر از دید یک دانشمند کامپیوتر به سیستم نگاه شود بیت‌های برنامه ویژگی‌های ساده را تشکیل می‌دهند. دید یک مهندس سخت‌افزار، الکترونیک، و یا یک فیزیک‌دان، از این سطح هم پایین‌تر می‌رود. بنابراین، در مورد ویژگی‌هایی که قرار است توسط کاربر مشاهده و تطبیق داده شوند، مناسب است که دید کاربر (یا متخصص دامنه) ملاک قرار داده شود

[۴] Composite Feature

[۵] Simple Adaptable Feature

[۶] این تعریف را می‌توان از تعریف قابلیت تطبیق نتیجه گرفت

[۷] Totally Adaptable Feature

[۸] تعاریف ویژگی کاملاً قابل تطبیق و ویژگی کاملاً غیرقابل تطبیق، بازگشتی هستند.

ویژگی کاملاً غیرقابل تطبیق[۹]: یک ویژگی سادهٔ غیرقابل تطبیق یک ویژگی کاملاً غیرقابل تطبیق است. همچنین، یک ویژگی مرکب که تمام ویژگی‌های تشکیل دهندهٔ آن کاملاً غیرقابل تطبیق باشند، یک ویژگی کاملاً غیرقابل تطبیق از سیستم است.

ساده یا مرکب بودن ویژگی‌های قابل تطبیق تأثیری در مسائل و روش‌های تطبیق ندارد. در حقیقت، اگر مکانیزمی برای تطبیق ویژگی‌های ساده ارائه شود، برای ویژگی‌های مرکب نیز قابل استفاده است زیرا اگر ویژگی‌های تشکیل دهندهٔ یک ویژگی مرکب تطبیق پیدا کنند، خود ویژگی مرکب نیز تطبیق پیدا کرده است. در مورد ویژگی‌های مرکب، ممکن است بخشی از یک ویژگی قابل تطبیق باشد و بخشی از آن قابل تطبیق نباشد.

در هنگام استخراج نیازمندی‌های سیستم، باید ویژگی‌های قابل تطبیق را در سناریوها مستند کرد. در قسمت ۲—۲ اندکی در مورد سناریوها توضیح داده شد. در ادامه، بیش از این از سناریوها صحبت نخواهد شد و فرض بر این است که ویژگی‌های قابل تطبیق سیستم برای معماران سیستم مشخص شده‌اند.

۴—۱—۲ ارتباط تطبیق و تعریف پویا

همان‌طورکه در فصل ۲ به‌طور مفصل بحث شد، قابلیت تطبیق نوعی قابلیت تغییر است. بااین‌حال، قابلیت تطبیق منطق حرفه تنها به‌معنی تغییر پویای ویژگی‌های تطبیقی سیستم نیست، بلکه شامل تعریف[۱۰] اولیهٔ آن‌ها نیز می‌شود. به‌عنوان مثال، قابلیت تعریف و تغییر پویای ساختار یک موجودیت حرفه هر دو از مصادیق قابلیت تطبیق ساختاری منطق حرفه هستند. در اینجا منظور از تعریف پویای یک ویژگی، تعریف آن با استفاده از یک سیستم نمادگذاری سطح بالا می‌باشد. واضح است که ابزار لازم برای این قابلیت، خود با استفاده از برنامه‌نویسی ایجاد شده است.

قضیهٔ تناظر تطبیق و تعریف پویا: اگر یک ویژگی موجود در سیستم کاملاً قابل تطبیق باشد، این ویژگی در هنگام ایجاد در سیستم نیز کاملاً به‌صورت پویا قابل تعریف بوده است و بر عکس. (به‌عنوان مثال اگر در سیستمی متخصص دامنه بتواند ساختار یک موجودیت را تطبیق دهد، حتماً تعریف ساختار موجودیت نیز به‌صورت پویا امکان‌پذیر بوده است و برعکس.)

اثبات: برای اثبات قسمت مستقیم قضیه می‌توان از برهان خلف استفاده کرد: فرض کنید تغییر پویای یک ویژگی A از سیستم به‌صورت کامل ممکن باشد، درحالی‌که تعریف پویای ویژگی A ممکن نبوده و بخشی از این ویژگی به‌نام a (که خود یک ویژگی است) از طریق برنامه‌نویسی ایجاد شده است. به‌عبارت‌دیگر، قسمتی از برنامه مثل P، که شامل یک یا چند ساختار برنامه‌نویسی است، a را تعریف می‌کند. دراین‌صورت، یکی از این دو ادعا باید درست باشد: یا تغییر a نیازمند تغییر برنامه است که این خلاف فرض است؛ یا در تغییر a برنامه دست نخورده باقی می‌ماند. دراین‌صورت، دیگر a به‌شکلی که در P تعریف شده وجود نخواهد داشت، درحالی‌که P وجود دارد و این خلاف فرض تعریف a در P می‌باشد. بنابراین، فرض تغییر پویا و تعریف غیرپویای A به تناقض می‌انجامد و قضیهٔ مستقیم اثبات می‌شود.

برای اثبات قسمت معکوس قضیه کافی است توجه شود که در صورت تعریف پویای یک ویژگی، اصلاً برنامه‌ای وجود ندارد که بتوان آنرا تغییر داد. در نتیجه تغییرات نیز پویا خواهند بود.

[۹] Totally Not-Adaptable Feature

[۱۰] Definition

ممکن است بخشی از یک ویژگی مرکب سیستم از طریق برنامه‌نویسی ایجاد شده باشد و بخشی به‌صورت پویا (با استفاده از یک سیستم نمادگذاری سطح بالاتر) تعریف شده باشد. دراین‌صورت، بنابر قضیهٔ بالا تنها بخشی که به‌صورت پویا تعریف شده است قابلیت تطبیق پویا را دارد.

از این قضیه به این نتیجه می‌رسیم: کافی است سیستم امکان تعریف پویای یک ویژگی را داشته باشد تا آن ویژگی در سیستم قابل تطبیق قلمداد شود و بر عکس. بنابراین، یک سیستم قابل تطبیق سیستمی است که امکان تعریف ویژگی‌های قابل تطبیق را با استفاده از یک سیستم نمادگذاری سطح بالا فراهم می‌کند. چنین سیستمی، حتی اگر در ظاهر واسطی برای تغییر این تعاریف در اختیار کاربر نگذارد، معماری آن از تغییر تعاریف پشتیبانی می‌کند و به‌راحتی می‌توان واسط کاربر مورد نیاز برای تطبیق را، به سیستم اضافه کرد.

۴—۱—۳ تعریف ساختار و رفتار

در قسمت ۲—۴ گفته شد که هدف این پایان‌نامه ارائهٔ راه‌حلی برای تطبیق رفتاری می‌باشد. قبل از ارائهٔ راه‌حل لازم است مفهوم رفتار دقیق‌تر بیان شود. در شیءگرایی برای هر شیء دو جنبه قابل تصور است. جنبهٔ ساختاری که با مشخصه‌های یک شیء مشخص می‌شود و جنبهٔ رفتاری که با اعمال شیء مشخص می‌گردد. به‌عبارت‌دیگر، توجه جنبه ساختاری معطوف به داده می‌باشد، بنابراین، با متغیرهای داده‌ای یا مشخصه‌ها و رابطه‌های پایگاه داده محقق می‌شود، درحالی‌که توجه جنبهٔ رفتاری به پردازش می‌باشد و با جملات اجرایی برنامه محقق می‌شود.

این مطب، در سطحی بالاتر، در مورد مجموعه‌ای از اشیاء نیز قابل طرح می‌باشد. ساختار یک مجموعه از اشیا عبارت است از مجموعهٔ ساختار هریک از آن‌ها به‌همراه ارتباطات ایستای آن‌ها با یکدیگر. منظور از ارتباط ایستا، نسبتی ماندگار[۱۱] (شبیه مفهوم ارتباط[۱۲] بین موجودیت‌ها در نمودار موجودیت—ارتباط[۱۳]) است که در هر لحظه از زمان بین دو شیء وجود دارد. به‌عنوان مثال نشانهٔ S5 از علائم بیماری D10 می‌باشد. رفتار یک مجموعه از اشیا نیز مجموعه‌ای است از رفتار هر یک از آن‌ها به‌همراه ارتباطات پویای آن‌ها. منظور از ارتباط پویا، تعاملی[۱۴] است که برای انجام عملیات بین آن‌ها انجام می‌شود. مثلاً برای ثبت یک نمونهٔ بیماری، یک شیء کنترلی عمل ثبت شیء بیماری را فراخوانی می‌کند. در این سطح نیز توجه جنبهٔ ساختاری به داده‌ها معطوف می‌شود و پیاده‌سازی آن با استفاده از مفاهیم داده‌ای (انواع داده، متغیر ارجاعی[۱۵] یا اشاره‌گر[۱۶] در زبان‌های برنامه‌نویسی و مفاهیم مشخصه، رابطه، و کلید خارجی[۱۷] در سیستم‌های پایگاه داده‌های رابطه‌ای[۱۸]) انجام می‌شود و توجه جنبهٔ رفتاری به پردازش است و با استفاده از جملات اجرایی برنامه (مثل جملات محاسباتی یا فراخوانی[۱۹] و ارسال پیام[۲۰]) پیاده‌سازی می‌شود.

---

[۱۱] Persistent Association

[۱۲] Relationship

[۱۳] Entity-Relationship Diagram

[۱۴] Collaborations or Interactions

[۱۵] Reference Variable

[۱۶] Pointer

[۱۷] Foreign Key

[۱۸] Relational Database Systems

[۱۹] Invocation یا Call

[۲۰] Message Passing

انتساب‌های غیرماندگار بین اشیاء، باتوجه‌به این‌که تنها در پردازش مورد استفاده قرار می‌گیرند و از دید کاربر قابل مشاهده نیستند، در مسئلهٔ قابلیت تطبیق اهمیتی ندارند و در تعریف ساختار نیامده‌اند.

همان‌طورکه در قسمت ۲—۵—۳ گفته شد، رفتار سطح بالاتری نیز وجود دارد که در سطح جریان کار دیده می‌شود. چنان‌که قبلاً نیز مورد تأکید قرار گرفت، این مبحث خارج از تمرکز این پایان‌نامه است، زیرا عملیات مورد نظر ما، ریزدانه‌تر هستند.

پس در این پایان‌نامه منظور از قابلیت تطبیق ساختاری امکان تغییر پویای تعریف موجودیت‌های حرفه و ارتباطات ایستای آن‌ها و منظور از تطبیق رفتاری تغییر پویای عملیات روی موجودیت‌های حرفه و تعاملات آن‌ها می‌باشد.

۴—۱—۴ نیاز توأم به ساختار و رفتار قابل تطبیق

در سیستم‌های مبتنی بر سبک معماری مدل قابل تطبیق شیء، دستیابی به ساختار پویا بسیار ساده‌تر از دست‌یابی به رفتار پویا می‌باشد. مثال‌هایی از این گونه سیستم‌ها در قسمت ۲—۵—۱ مورد بررسی قرار گرفتند. به‌همین‌دلیل، معمولاً، این سیستم‌ها از پویایی ساختار، بهتر از رفتار پشتیبانی می‌کنند. حال این سؤال مطرح می‌شود که سیستمی که از اهداف آن رفتار قابل تطبیق است، نسبت به پویایی ساختار چه سیاستی می‌تواند داشته باشد. در این رابطه، توجه به این نکته ضروری است که بسیار بعید است سیستمی نیازمند رفتار پویا باشد ولی نیازمند ساختار پویا نباشد. دلایلی که در قسمت ۱—۲ برای نیاز به قابلیت تطبیق ارائه شد، هم برای تطبیق رفتاری صادق هستند و هم برای تطبیق ساختاری. بنابراین، روشی که در این پایان‌نامه ارائه می‌گردد، هم از قابلیت تطبیق رفتاری پشتیبانی می‌کند و هم از قابلیت تطبیق ساختاری.

ازسوی‌دیگر، در صورتی که سیستمی مورد نیاز باشد که ساختار ثابت و رفتار پویایی دارد نیز می‌توان از روشی که ارائه می‌شود استفاده کرد. حتی در حالتی که این اصرار بر ایجاد ساختار با استفاده از برنامه‌نویسی وجود داشته باشد (مثلاً به‌دلیل ملاحظات کارایی) نیز می‌توان از ترکیبی از این روش و انعکاس موجود در زبان‌های برنامه‌نویسی برای دست‌یابی به رفتار پویا بر روی یک ساختار ثابت استفاده کرد.

۴—۱—۵ سایر مسائل مربوط به تطبیق

مسائل دیگری نیز وجود دارند که به مسئلهٔ تطبیق ارتباط پیدا می‌کنند. به‌عنوان مثال، می‌توان به موارد زیر اشاره کرد:

- حالات نامعتبر[۲۱] یا ناسازگار[۲۲] و انتقال نمونه‌ها[۲۳]: در صورت تغییر ساختار یک موجودیت، امکان دارد نمونه‌های موجود آن به یک حالت نامعتبر وارد شوند. دراین‌صورت باید نمونه‌های قبلی موجودیت به نمونه‌های جدید تبدیل شوند.
- سابقه[۲۴] تغییرات: باید سابقهٔ تغییرات ساختاری و رفتاری (مثلاً با شمارهٔ نسخه) نگه‌داری شود.

[۲۱] Invalid
[۲۲] Inconsistent
[۲۳] Instance Migration
[۲۴] History

- عقب‌گرد[۲۵]: باید در صورت نیاز امکان عقب‌گرد به تعاریف قبلی ویژگی‌های ساختاری و رفتاری امکان‌پذیر باشد.
- بررسی نوع[۲۶] و مدیریت خطاهای زمان اجرا: باید تا حد امکان تعاریف داده شده به سیستم بررسی شوند تا ناسازگاری در آن‌ها وجود نداشته باشد. به‌عنوان مثال، تعریف یک عمل در سیستم نباید به مشخصه‌ای از یک موجودیت اشاره کند که وجود نداشته باشد. در صورت ممکن نبودن کشف بعضی انواع خطا در زمان تعریف، سیستم باید در مقابل خطاهای زمان اجرا عکس‌العملی منطقی داشته باشد.

یک سیستم قابل تطبیق باید مکانیزم‌هایی برای این موارد ارائه کند. اما این مکانیزم‌ها مستقل از خود مکانیزم تطبیق است و می‌توان به‌هر روشی که برای تطبیق استفاده می‌شود این مکانیزم‌ها را اضافه کرد. بنابراین، لازم نیست این پایان‌نامه این مشکلات را حل کند.

۴—۱—۶ یک ابزار زیربنایی برای سیستم‌های قابل تطبیق

امروزه برای تولید و نگه‌داری یک سیستم نرم‌افزاری طیف گسترده‌ای از روش‌ها وجود دارند. در یک طرف این طیف برنامه‌نویسی محض قرار دارد که در این حالت جز نوشتن متن برنامه و ترجمهٔ آن به یک برنامه قابل اجرا هیچ نوع امکانات دیگری مورد استفاده قرار نمی‌گیرد. روش‌های تبدیلی یا تولیدی که در قسمت ۲—۱ در مورد آن توضیح داده شد، بهبودی در این روش ابتدایی ایجاد می‌کنند. در سوی دیگر طیف، روشی (هر چند فرضی) قرار دارد که تولید کنندهٔ سیستم برای تولید یا تطبیق سیستم حتی یک خط هم برنامه نمی‌نویسد (شبیه آنچه در [۳۷] وجود دارد). حال این سؤال مطرح است که برای یک سیستم قابل تطبیق چه روشی مناسب است.

از اهداف این کار نتیجه می‌شود که روشی مورد نظر است که با استفاده از آن متخصص دامنه بتواند با یک سیستم نمادگذاری سطح بالا ویژگی‌های سیستم را به‌صورت پویا تعریف کند یا تغییر دهد. واضح است که برای این کار یک سیستم یا ابزار زیربنایی لازم است که مفسر این زبان سطح بالا باشد و واسط‌های کاربری مناسب را در اختیار متخصص دامنه قرار دهد. این سیستم از این طریق امکان تعریف، تطبیق و اجرای سیستم کاربردی مورد نیاز کاربر را فراهم می‌کند. چنین سیستمی پیش از این موتور تطبیق نامیده شد. در ادامه در مورد اهداف سطح بالای یک موتور تطبیق بحث می‌شود تا یک راهبرد مناسب برای ایجاد چنین سیستمی به‌دست آید.

۴—۲ اهداف یک موتور تطبیق

دست‌یابی به یک راهبرد مناسب برای یک موتور تطبیق نیازمند تحلیل اهداف آن می‌باشد. با یک دید ایده‌آل‌گرایانه، چهار هدف زیر را می‌توان برای چنین سیستمی در نظر گرفت:

۱) تغییر در زمان اجرا: تطبیق به‌طور کامل در زمان اجرا (بدون برنامه‌نویسی و در زمانی از مرتبهٔ یک ساعت) صورت گیرد.

---

[۲۵] Rollback

[۲۶] Type Checking

۲) عمومیت دامنه: برای تمام سیستم‌هایی که نیازمند قابلیت تطبیق در زمان‌اجرا توسط کاربر هستند، قابل استفاده باشد.

۳) عمومیت تعریف و تطبیق: هر نوع ویژگی‌ای را بتوان در این سیستم به‌صورت پویا تعریف کرد و تغییر داد.

۴) سادگی: یک متخصص دامنه که با برنامه‌نویسی آشنایی ندارد، بتواند سیستم کاربردی را با استفاده از این سیستم تولید و نگه‌داری کند.

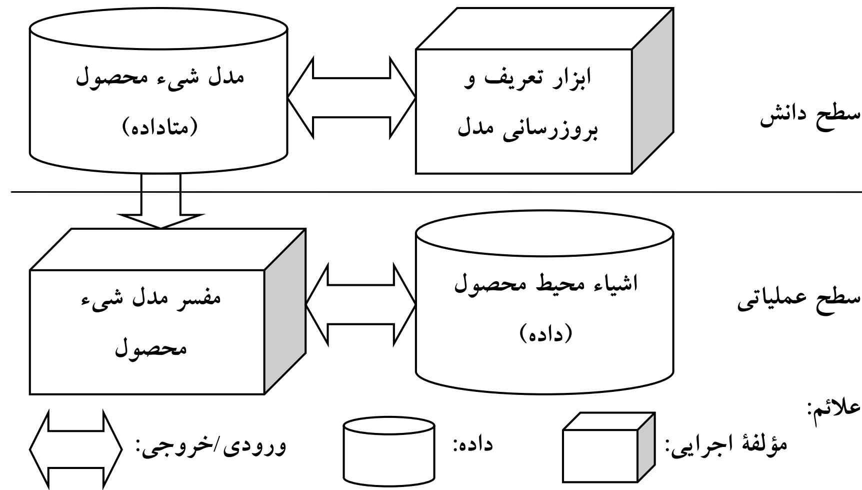
در ذکر اهداف سیستم از سایر نیازمندی‌ها مثل کارایی، به‌دلیل بدیهی بودن، صرف‌نظر شده است. مجموع اهداف ۲ و ۳ قدرت موتور تطبیق در مدل‌سازی را می‌رسانند و اهداف ۱ و ۴ به قابلیت استفاده از سیستم به‌وجود آمده توسط موتور تطبیق، برمی‌گردند. در ادامه نشان داده می‌شود که دستیابی به این اهداف به‌صورت هم‌زمان امکان‌ناپذیر است.

اگر سیستمی حداکثر قدرت مدل‌سازی را داشته باشد، چنان‌که در اهداف ۲ و ۳ بیان شده، زبان این سیستم به پیچیدگی یک زبان برنامه‌نویسی همه‌منظوره خواهد شد. زیرا اولاً، باید عمومیت دامنه داشته باشد؛ در نتیجه، باید در سطح تجرید بسیار پایینی باشد که تمام دامنه‌ها را پوشش دهد (مثل زبان‌های همه‌منظوره). ثانیاً، در صورت عمومیت تعریف باید این زبان توانایی توصیف تمام ویژگی‌های ساختاری و رفتاری را داشته باشد که این مطلب نیز باعث می‌شود این زبان به‌سمت عمومیت پیش رود و در سطح تجرید پایینی قرار گیرد. ازسوی‌دیگر، پیچیده شدن زبان با قابلیت استفاده از این سیستم سازگار نیست زیرا اگر زبان پیچیده باشد، سادگی از بین می‌رود و متخصص دامنه نمی‌تواند با این زبان کار کند. علاوه‌براین، بااین‌که در ظاهر تطبیق در زمان اجرا و بدون برنامه‌نویسی صورت می‌گیرد، در عمل مزایای تطبیق زمان اجرا از بین می‌رود زیرا اعمال تغییرات با چنین زبانی، زمانی بیشتر از مرتبهٔ زمانی یک ساعت نیاز دارد.

بنابراین، دستیابی به سیستمی که این اهداف را با هم برآورده کند، ناممکن است و لازم است اهداف ساده‌تر شوند. ازآن‌جاکه هدف اصلی در سیستم‌های مورد بحث در این پایان‌نامه، قابلیت تطبیق توسط متخصص دامنه می‌باشد، اهداف ۱ و ۴، اهداف اصلی یک موتور تطبیق به‌حساب می‌آیند. بنابراین، از اهداف ۲ و ۳ به این شکل عمومی صرف‌نظر می‌شود. به‌عبارت‌دیگر، این قابل قبول است که یک موتور تطبیق محدود به یک دامنهٔ خاص باشد. همچنین، لازم نیست با چنین سیستمی بتوان بدون برنامه‌نویسی هر تغییری در سیستم کاربردی قابل اعمال باشد. اما باید در طراحی یک موتور تطبیق سعی شود که تا حد امکان و بدون خدشه وارد شدن به قابلیت استفاده از سیستم، قدرت مدل‌سازی بالا باشد.

۴—۳ سبک معماری سیستم‌های قابل تطبیق

شکل ۴—۱ ایدهٔ اصلی سبک معماری سیستم‌های قابل تطبیق را نشان می‌دهد. این مدل حالت تطبیق داده شده‌ای از مدل‌های ارائه شده در [۱۱] برای ذخیره سازی متاداده و معماری‌های انعکاسی در [۱۹، ۱۸] می‌باشد. نحوهٔ کار به‌صورت کلی چنین است که متخصص دامنه به‌وسیلهٔ ابزار تعریف و به‌روزرسانی مدل، مدل شیء سیستم خود (محصول) را تعریف می‌کند. این مدل که به‌شکل داده نگه‌داری می‌شود، در حقیقت متادادهٔ سیستم است. یک مفسّر، این مدل را تفسیر کرده و امکان عملیات مختلف بر روی اشیائی از انواع موجودیت‌های تعریف شده در مدل را در اختیار کاربر قرار می‌دهد. نکتهٔ قابل توجه در این شکل تقسیم



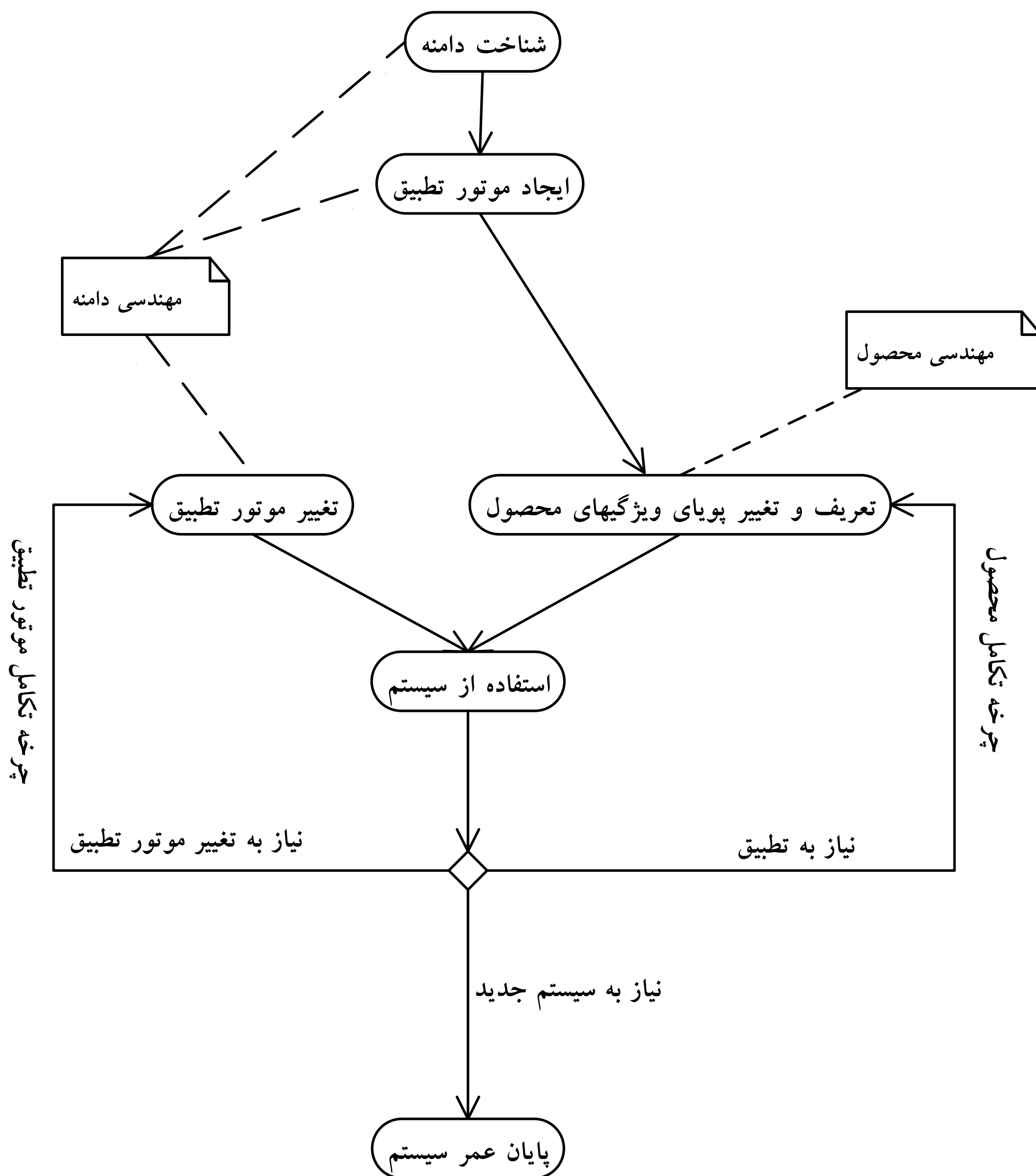
شکل ۴—۱: شمای کلی سبک معماری سیستم‌های قابل تطبیق

مفاهیم به دو سطح دانش[۲۷] (متا) و سطح عملیاتی[۲۸] (پایه[۲۹]) می‌باشد. لازمهٔ دست‌یابی به چنین سیستم پویایی این است که به‌جای مدل‌سازی محصول از سطح تجرید[۳۰] بالاتری استفاده شود و یک متامدل برای خانواده‌ای از محصولات ایجاد گردد (متن برنامه‌های سطح دانش یک متا مدل از سیستم را پیاده‌سازی می‌کند.) دادهٔ سطح دانش، مدلی از یک محصول مشخص است. در سطح عملیاتی این مدل اجرا می‌شود و دادهٔ این سطح اشیاء واقعی مربوط به محصول هستند. مجموع مؤلفه‌های ابزار تعریف و به‌روزرسانی مدل و مفسر مدل شیء محصول، موتور تطبیق را تشکیل می‌دهند. خود محصول به‌صورت یک سیستم جدا وجود ندارد و به‌صورت دادهٔ موتور تطبیق محقق می‌شود.

۴—۴ چرخهٔ حیات سیستم‌های قابل تطبیق

شکل ۴—۲ چرخهٔ حیات سیستم‌های قابل تطبیق مبتنی بر یک موتور تطبیق را با استفاده از نمودار فعالیت UML[۳۱] نشان می‌دهد. در مورد شناخت دامنه قبلاً صحبت شد. در اینجا لازم است یادآوری شود که برخلاف سایر روش‌های تولید خانواده‌ای از محصولات لازم نیست تمام جزئیات نیازمندی‌ها (شامل ساختار و رفتار کلیهٔ موجودیت‌های حرفه) در شناخت دامنه استخراج شود. بلکه کافی است یک الگوی کلّی از ساختار و رفتار سیستم به‌دست آید. بر اساس این الگوی کلی در مرحلهٔ بعد موتور تطبیق به‌وجود می‌آید. بعد از این مرحله می‌توان سیستم را در محیط واقعی راه‌اندازی کرد و متخصصین دامنه می‌توانند ویژگی‌های محیط را به‌صورت پویا تعریف کنند و تغییر دهند. همان‌طور که در شکل مشخص است در این فرآیند دو چرخهٔ تکامل وجود دارد:

[۲۷] Knowledge Level
[۲۸] Operational Level
[۲۹] Base Level
[۳۰] Abstraction Level
[۳۱] UML Activity Diagram



شکل ۴–۲: چرخهٔ حیات سیستم‌های قابل تطبیق

۱) چرخهٔ تکامل محصول: در صورتی که تغییرات نیازمندی‌ها با استفاده از قابلیت تطبیق موجود در سیستم قابل اعمال باشند متخصص دامنه آن‌ها را اعمال می‌کند.

۲) چرخهٔ تکامل موتور تطبیق: در صورتی که تغییرات نیازمندی‌ها با استفاده از قابلیت تطبیق موجود در سیستم قابل اعمال نباشند، به‌عبارت‌دیگر، قدرت مدل‌سازی موتور تطبیق برای نیازمندی‌های جدید کافی نباشد، لازم است موتور تطبیق با استفاده از برنامه‌نویسی تغییر کند.

واضح است که هرچه نیاز به تغییر موتور تطبیق کمتر باشد، هزینه کمتر خواهد بود. بنابراین، معماران باید سعی کنند متامدل سیستم نیازمندی‌های احتمالی بیشتری را پوشش دهد یا به‌عبارت‌دیگر قدرت مدل‌سازی موتور تطبیق بیشتر باشد. ازسوی‌دیگر، این مطلب باعث کلی‌تر شدن متامدل (پوشش تعداد بیشتری از محصولات و دامنه‌ها) خواهد شد و در نتیجه پیچیدگی زبان مدل‌سازی بیشتر می‌شود. و این به‌معنی کاهش قابلیت استفادهٔ سیستم خواهد بود. به این دوگانگی در اهداف قبلاً نیز اشاره شد. در هر حال سبک-سنگین کردن این اهداف بخشی از وظایف معماران سیستم است و نمی‌توان یک راه‌کار کلی برای آن ارائه کرد.

۴—۵ سلسله‌مراتب‌های تکاملی: مانع قابلیت تطبیق

روش متداول در طراحی سیستم‌ها به‌روش شیءگرا شامل ایجاد یک کلاس برای هر نوع موجودیت حرفه می‌باشد [۲۵]. این روش، معمولاً، به پیدایش یک یا چند سلسله‌مراتب بزرگ از موجودیت‌های حرفه می‌انجامد. به‌دلایل متعددی استفاده از سلسله‌مراتب وراثت در سیستم‌ها، به‌طورکلی، و در سیستم‌های با نیازمندی قابلیت تطبیق، به‌طور خاص، مشکل آفرین است. برخی مشکلات کلی وراثت عبارتند:

- پیچیدگی وراثت [۱۰، ۲۲]: این مطلب را با مثالی از [۱۰] می‌توان نشان داد. فرض کنید یک شرکت بیمه انواعی از سیاست‌های بیمه‌ای دارد. سیاست بیمه‌ای توصیفی است از مواردی که بیمه آن‌ها را پوشش می‌دهد، مانند منزل، و هزینه‌ای که شرکت بیمه در صورت صدمه دیدن آن‌ها به مشتری می‌پردازد. یک سیاست بیمه‌ای می‌تواند شامل سیاست‌های دیگر باشد. در روش معمول مدل‌سازی، یک سیاست مرکب را می‌توان از طریق ارث‌بری از سیاست‌هایی که اجزای آن را مشخص می‌کنند، مثل سیاست بیمهٔ سیل برای خانه، سیاست بیمهٔ اتومبیل، و سیاست بیمهٔ دارایی، تعریف کرد. دراین‌صورت وراثت چندگانه [۳۲] مورد نیاز است که خود پیچیدگی‌ها وابهاماتی در مدل‌سازی و پیاده‌سازی دارد. از آن گذشته برای ایجاد سیاستی که دو اتومبیل را بیمه می‌کند، لازم است دو بار از کلاس سیاست بیمهٔ اتومبیل ارث‌بری انجام شود که پیاده‌سازی چنین وراثتی، معمولاً، در زبان‌های برنامه‌نویسی شیء گرا ممکن نیست.
- ناسازگاری با محصورسازی [۳۳]: باتوجه‌به این‌که وراثت یک روش استفاده مجدد جعبه‌سفید [۳۴] است و کلاس فرزند از جزئیات پیاده‌سازی کلاس پدر مطلع است، محصورسازی خدشه‌دار می‌شود [۲۸].

[۳۲] Multiple Inheritance

[۳۳] Encapsulation

[۳۴] Whitebox Reuse

- وابستگی فرزند به پدر: برنامه‌نویس باید از پیاده‌سازی کلاس(های) پدر آگاهی کامل داشته باشد که خود این مطلب مانعی برای استفادهٔ مجدد و تغییر سیستم به‌شمار می‌رود. درضمن، بخشی از کارکرد کلاس فرزند در کلاس(های) پدر پیاده‌سازی شده است که باعث وابستگی فرزند به پدر می‌شود. برای مثال، در مواقع استفادهٔ مجدد فرزند گاهی لازم می‌شود کلاس پدر تغییر کند تا کلاس فرزند در دامنهٔ جدیدی قابل استفاده باشد [۲۸].

موارد زیر مشکلات سلسله‌مراتب وراثت در طراحی زیرمجموعه‌ای از سیستم‌ها را نشان می‌دهد که در نتیجهٔ این مشکلات نیازمند قابلیت تطبیق خواهند بود:

- بزرگ شدن سلسله‌مراتب وراثت [۲۵، ۱۰]: گفته شد که در سیستم‌های با موجودیت‌های دامنهٔ متعدد، مدل‌سازی هر موجودیت با یک کلاس، به یک سلسله‌مراتب بزرگ (شامل صدها یا هزاران کلاس) ختم خواهد شد و در نتیجه برنامه غیرقابل کنترل می‌شود. محاسبه‌ای که در [۱۰] برای یک شرکت بیمه انجام شده، نشان می‌دهد پیاده‌سازی چنین سلسله‌مراتبی حتی اگر ممکن باشد شامل ۱۰۰۰۰ کلاس می‌شود.
- نیاز به برنامه‌نویسی: صرف نظر از حجم و پیچیدگی که در موارد قبل مطرح شد، در این روش ایجاد و تغییر موجودیت‌های حرفه تنها از طریق برنامه‌نویسی ممکن است. این مطلب دقیقاً با هدفی که معماران نرم‌افزارهای قابل تطبیق به‌دنبال آن هستند، در تضاد است.

مواردی که بیان شد، به‌هیچ‌وجه به‌معنی بی‌فایده بودن وراثت، به‌عنوان یکی از امکانات طراحی شیء گرا، نیست. به‌عبارت‌دیگر، هر روشی مزایا و معایبی دارد و منظور از بیان این موارد این است که اگر وراثت در جای نامناسب مورد استفاده قرار گیرد، ضرر آن بیشتر از نفع آن است. سلسله‌مراتب‌ها را می‌توان به دو نوع زیر تقسیم کرد:

۱) سلسله‌مراتب‌های غیرتکاملی (ایستا): در عملیات معمولی سیستم کلاسی به این سلسله‌مراتب‌ها اضافه نمی‌شود و نیازی به تغییر در رفتار آن‌ها وجود ندارد. البته، این سلسله‌مراتب‌ها در مواردی نیازمند تغییراتی هستند که با تغییر متن برنامه‌ها اعمال می‌شوند، ولی این تغییرات باتوجه به تناوب کم، کار سیستم را مختل نمی‌کند.

۲) سلسله‌مراتب‌های تکاملی (پویا): بخشی از عملیات معمول سیستم نیازمند اضافه شدن یا تغییر کلاس‌های این سلسله‌مراتب‌ها می‌باشد.

اگر بخشی از یک سیستم سلسله‌مراتبی تکاملی باشد، مثلاً در مورد سیستم‌هایی با ویژگی‌های گفته شده در قسمت ۱—۲، سیستم نیازمند یک مدل قابل تطبیق به‌جای یک سلسله‌مراتب است. در بقیهٔ قسمت‌های سیستم، همان‌طور که به‌زودی نشان داده می‌شود، وراثت یک مکانیزم مؤثر قلمداد می‌شود.

پیدایش سلسله‌مراتب وراثت برای موجودیت‌های حرفه دو دلیل دارد: تفاوت در ساختار موجودیت‌ها و تفاوت در رفتار آن‌ها [۲۵]. برای از بین بردن سلسله‌مراتب باید هر یک از این دو دلیل را در نظر گرفت. راه‌حل‌های جداگانه‌ای برای رفع این دو دلیل قابل استفاده هستند. در قسمت بعد، به چند تاکتیک عمومی برای طراحی نرم‌افزارهای قابل تطبیق اشاره می‌گردد.

۴—۶ تاکتیک‌ها و تکنیک‌هایی برای دست‌یابی به قابلیت تطبیق

مفاهیم تاکتیک و تکنیک در قسمت ۳—۲ تعریف شد. در این قسمت تاکتیک‌هایی برای ایجاد قابلیت تطبیق در نرم‌افزار و تکنیک‌هایی برای تحقق این تاکتیک‌ها در سیستم‌های شیءگرا معرفی می‌گردد.

۴—۶—۱ تاکتیک داده به‌جای برنامه

یک نرم‌افزار را می‌توان به دو بخش تقسیم کرد: برنامه و داده[۳۵]. تغییر داده یک امر پذیرفته شده در عملیات سیستم نرم‌افزاری است و جزئی از عملکرد سیستم است. حتی بهتر است گفته شود که عملیات سیستم از راه تغییر داده صورت می‌گیرد. ازسوی‌دیگر، تغییر برنامه امری دشوار است که برای ذی‌نفعان سیستم کاری تحمیلی قلمداد می‌شود و تحقیقات زیادی در جهت ساده کردن آن یا فرار از آن صورت گرفته است و می‌گیرد. یک فرق اساسی بین داده و برنامه این است که کاربر می‌تواند داده را در زمان اجرا تغییر دهد، اما نمی‌تواند برنامه را در زمان اجرا تغییر دهد.

با این مقدمه، این مطلب به ذهن خطور می‌کند که اگر بتوان راهی پیدا کرد که ویژگی‌های سیستم به‌جای برنامه با داده مدل‌سازی شوند مشکل قابلیت تطبیق حل شده است. به داده‌ای که قسمتی از خود سیستم را توصیف می‌کند متاداده گفته می‌شود. همان‌طور که در قسمت ۴—۸ توضیح داده می‌شود، استفاده از این تاکتیک در مورد رفتار به‌اندازهٔ ساختار کارایی ندارد. تاکتیک‌های بعدی راه‌حلی برای رفتار قابل تطبیق ارائه می‌دهد.

لازم به ذکر است که این تاکتیک بسیار شبیه به‌کارگیری نمایش-از-خود است که در سیستم‌های انعکاسی به‌کار می‌رود و در قسمت ۲—۵—۵ مورد بحث قرارگرفت.

۴—۶—۲ تاکتیک عمومی کردن برنامه‌ها از طریق پارامتری کردن

ایدهٔ این تاکتیک این است که اگر تبدیل برخی ویژگی‌های برنامه به داده ناممکن یا نامناسب باشد (مثلاً ویژگی‌های رفتاری)، که قطعاً همین‌طور است، این ویژگی‌ها طوری ایجاد شوند که در شرایط مختلف و بر روی انواع مختلف ورودی‌ها قابل اجرا باشند. به‌عبارت‌دیگر، این ویژگی‌ها باید عمومی[۳۶] نوشته شوند تا درصورت لزوم برای کاربردهای مختلفی مورد استفاده قرار گیرند. مکانیزم کار در این روش پارامتری کردن[۳۷] آن قسمت از برنامه است. مثالی از این روش مفهوم قالب[۳۸] در زبان C++ است [۵۹]. با استفاده از قالب‌ها می‌توان کلاس‌هایی ایجاد کرد که بتوانند با انواع دادهٔ مختلفی کار کنند. مثلاً می‌توان صفی[۳۹] ایجاد کرد که

---

[۳۵] در حقیقت نمی‌توان مرز دقیقی بین داده و برنامه مشخص کرد زیرا موجودیت‌های نرم‌افزاری می‌توانند هم داده باشند و هم برنامه. به‌عبارت‌دیگر، هر برنامه خود داده‌ای است برای کامپایلر؛ هر برنامهٔ کاربردی قابل اجرا داده‌ای است برای سیستم عامل؛ برنامه‌های سیستم‌عامل خود دادهٔ سخت‌افزار هستند. ازسوی‌دیگر، هر داده‌ای نیز گرامری دارد و از این جهت، بی‌شباهت به یک برنامه نیست. بنابراین، وقتی می‌خواهیم داده و برنامه را از هم جدا کنیم، باید سطح تجرید بحث را مشخص کنیم. در این‌جا سطح تجرید بحث، سطح برنامهٔ کاربردی است.

[۳۶] Generic

[۳۷] Parametrization

[۳۸] Template

[۳۹] Queue

قابلیت ذخیره کردن انواع دادهٔ مختلف را داشته باشد. در هنگام ساختن نمونه‌ای از این صف، باید نوع داده‌ای که قرار است در آن ذخیره شود به‌عنوان پارامتر به کلاس صف داده شود. البته مفهوم قالبی که در زبان‌های برنامه‌نویسی وجود دارد برای تطبیق توسط کاربر مناسب نیست، اما اگر بتوان روش مشابهی پیدا کرد که در آن پارامتر در زمان اجرا توسط متخصص دامنه مشخص شود، برای هدف مورد نظر قابل استفاده است.

۴—٦—۳ تاکتیک ترکیب عملیات در زمان اجرا برای ساختن عملیات پیچیده‌تر

تاکتیک عمومی کردن برنامه‌ها از طریق پارامتری کردن بااین‌که روش قدرتمندی برای پشتیبانی از رفتار قابل تطبیق می‌باشد، از عملیات محدودی پشتیبانی می‌کند. دلیل این مطلب این است که در استفاده از این تاکتیک برای تعریف یک عمل جدید، لازم است قالب عمل در سیستم موجود باشد. اما برخی اعمال هستند که از قبل قابل پیش‌بینی نیستند تا قالب آن‌ها در سیستم قرار داده شود. تاکتیک ترکیب عملیات در زمان اجرا برای ساختن عملیات پیچیده‌تر با هدف پشتیبانی از این عملیات مورد استفاده قرار می‌گیرد. ایدهٔ این تاکتیک ارائه روشی است که برای ایجاد عملیات پیچیده‌تری که از قبل پیش‌بینی نشده‌اند عملیات ساده را به‌صورت پویا با هم ترکیب کند. منظور از عملیات ساده عملیاتی درشت دانه‌تر از دستورات یک زبان برنامه‌نویسی است، زیرا درغیراین‌صورت، پیچیدگی، بیش از توان متخصصین دامنه خواهد بود. این تاکتیک نیازمند تاکتیک عمومی کردن برنامه‌ها از طریق پارامتری کردن است. به‌عبارت‌دیگر عملیات ساده‌ای که در اینجا مطرح هستند باید به‌اندازهٔ کافی عمومی باشند تا بتوان در کاربردهای مختلف آن‌ها را با هم ترکیب کرد.

۴—٦—۴ تکنیک‌های شیءگرا برای تحقق تاکتیک‌ها

سه تاکتیک بالا کلی هستند و مستقل از یک روش طراحی خاص مثل شیء گرایی. این تاکتیک‌ها، خود توسط یک سری تکنیک طراحی شیءگرا قابل پیاده‌سازی هستند. در ادامه برخی از این تکنیک‌ها بررسی می‌شوند:

۱) شیء به‌جای کلاس: در برنامه‌نویسی شیء گرا اشیاء ماهیت داده‌ای و پویا دارند زیرا هر شیء در زمان اجرا مجموعه‌ای از داده‌ها می‌باشد و کلاس‌ها ماهیت برنامه‌ای و ایستا. بنابراین، اگر در جایی بتوان از اشیاء به‌جای کلاس‌ها استفاده کرد، تاکتیک داده به‌جای برنامه تحقق می‌یابد. در استفاده از این تکنیک انتساب [۴۰] بین اشیاء جایگزین رابطهٔ نمونه [۴۱] بودن بین اشیاء و کلاس‌ها می‌شود.

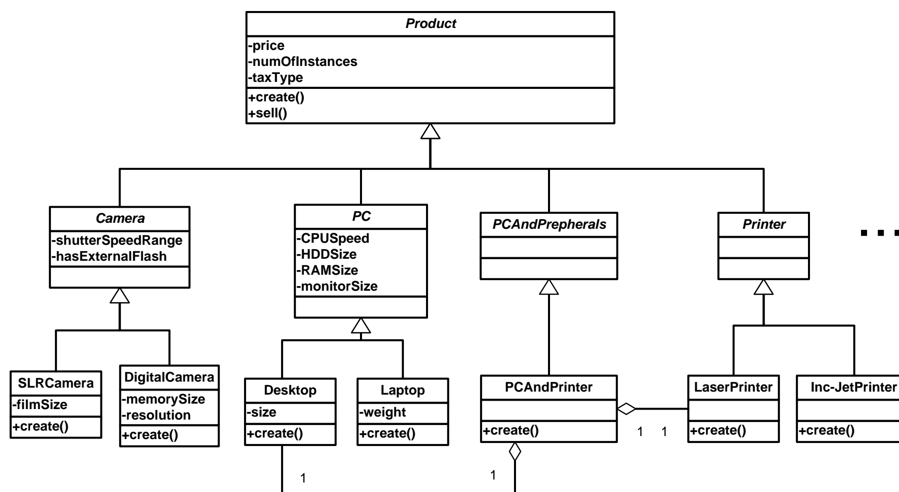
۲) ترکیب به‌جای وراثت: یک راه‌حل برای خلاصی از سلسله‌مراتب این است که به‌جای استفاده از ارث‌بری به‌عنوان مکانیزم استفادهٔ مجدد، از ترکیب استفاده شود [۲۸]. در این روش اشیاء جدید و احتمالاً پیچیده‌تر از ترکیب اشیاء موجود به‌وجود می‌آیند. یک مزیت این روش استفادهٔ مجدد جعبه سیاه [۴۲] است. مزیت دیگر این روش پویایی بیشتر می‌باشد زیرا ترکیب با استفاده از اشیاء انجام می‌شود و نه کلاس‌ها و مکانیزم پیاده‌سازی آن، به‌دست آوردن یک مرجع [۴۳] در زمان اجرا است. نقطه ضعف ترکیب در مقابل وراثت، استفادهٔ مجدد کمتر است. زیرا معمولاً ترکیب اشیاء موجود برای ایجاد

[۴۰] Association

[۴۱] Instance

[۴۲] Blackbox Reuse

[۴۳] Reference



شکل ۴—۳: سلسله‌مراتب به‌دست آمده از مدل‌سازی معمول محصولات سیستم فروشگاه

کارکردهای جدید کافی نیست و تعریف کلاس‌های جدیدی ضروری است و ازآن‌جاکه تعریف و تغییر کلاس‌های جدید با استفاده از وراثت ساده‌تر است، استفادهٔ مجدد در این روش بهتر انجام می‌شود. از این تکنیک می‌توان برای تحقق تاکتیک داده به‌جای برنامه استفاده کرد.

۳) نمایندگی: راهی است برای این‌که ترکیب از نظر استفادهٔ مجدد به‌قدرت‌مندی وراثت شود [۲۸]. در این عمل یک شیء به‌یک شیء دیگر نمایندگی می‌دهد که عملی را روی آن اجرا کند. معمولاً شیء اول مرجعی از خود در اختیار نماینده قرار می‌دهد تا نماینده از طریق آن مرجع به‌عملیات یا داده‌های شیء دسترسی داشته باشد. این تکنیک را می‌توان در پیاده‌سازی عمومی کردن برنامه‌ها از طریق پارامتری کردن به‌کار برد.

باتوجه‌به نقایص و توانایی‌های وراثت، ترکیب، و نمایندگی، در عمل ترکیبی از این روش‌ها راه‌حل مناسب را به‌وجود می آورد.

هر یک از این تاکتیک‌ها و تکنیک‌ها در برخی سیستم‌های قابل تطبیق مورد استفاده قرار گرفته‌اند. در حقیقت تاکتیک‌های معرفی شده در این قسمت باید باهم ترکیب شوند تا قابلیت تطبیق ساختاری و رفتاری را به‌وجود آورند. همان‌طور که در قسمت ۳—۲ گفته شد، یک راه برای به‌کارگیری تاکتیک‌ها و [تکنیک‌ها] در معماری نرم‌افزار، استفاده از الگوهایی است که آن‌ها را محقق می‌کند [۱]. در قسمت ۴—۷ با استفاده از یک مثال به‌کارگیری این تاکتیک‌ها و تکنیک‌ها در سیستم‌های طراحی شده با استفاده از سبک معماری مدل قابل تطبیق شیء نشان داده می‌شود.

۴—۷ روش تطبیق در سبک معماری مدل قابل تطبیق شیء

در این قسمت، با استفاده از یک مثال ساده ایدهٔ کلی الگوهایی که برای تطبیق ساختاری و رفتاری در سبک معماری مدل قابل تطبیق شیء به‌کار گرفته می‌شوند، ارائه می‌گردد.

۴—۷—۱ سیستم فروشگاه

یک فروشگاه بزرگ که محصولات متعددی را به مشتریان عرضه می‌کند، در صدد ایجاد یک سیستم مبتنی بر وب برای نمایش اطلاعات و فروش محصولات خود می‌باشد. شکل ۴—۳ بخشی از سلسله‌مراتب محصولات این فروشگاه را، در روش مدل‌سازی محصولات از طریق کلاس‌ها، نشان می‌دهد. هریک از اشیاء کلاس‌های این سلسله‌مراتب یک نوع محصول مشخص (مثل تلفن‌همراه Nokia-8219 یا چاپگر HP-LaserJet-1300) را مدل می‌کند. بنابراین، یک شیء از یکی از کلاس‌ها یک مدل محصول را مشخص می‌کند و نه یک نمونهٔ خاص از یک مدل محصول را. باتوجه‌به این‌که به‌طور روزانه محصولاتی به این فروشگاه اضافه می‌شوند، این سلسله‌مراتب یک سلسله‌مراتب تکاملی است. نکات زیر در مورد این سلسله‌مراتب قابل توجه هستند:

- مشخصه‌های price و numOfInstances در تمام انواع محصول مشترک هستند.
- هر یک از انواع محصول موجود در این فروشگاه می‌تواند مشخصه‌های منحصر به‌فرد خود را داشته باشد.
- بعضی از کلاس‌ها، مثل Product و Camera مجرد[۴۴] هستند و تنها برای فاکتورگیری از مشخصه‌ها در سیستم قرار داده شده‌اند. نام کلاس‌های مجرد به‌صورت ایتالیک نشان داده می‌شوند. سایر کلاس‌ها واقعی[۴۵] هستند و نام آن‌ها معمولی نوشته می‌شود.
- یک محصول قابل فروش ممکن است از محصولات دیگری تشکیل شود (مثل کلاس PCAndPrinter که شامل یک دستگاه کامپیوتر و یک چاپگر لیزری می‌باشد و اجزای آن به‌تنهایی نیز قابل فروش هستند).
- اعمال sell و create، برای تمام محصولات وجود دارند، اما برای هر نوع محصول متفاوت هستند.

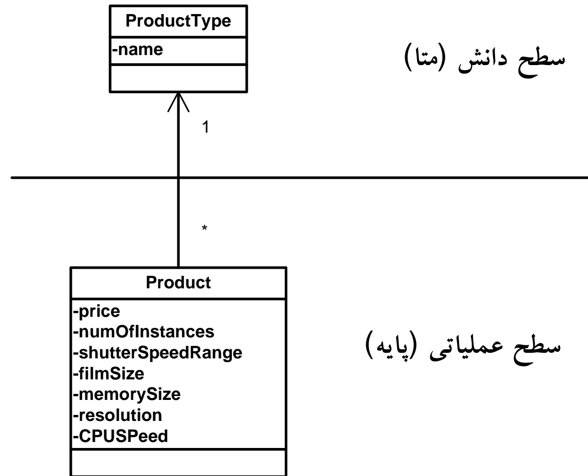
عمل create عبارت است از تعریف یک محصول جدید. این عمل برای محصولات مختلف ماهیت یکسانی دارد ولی چون مشخصه‌ها و قواعد اعتبارسنجی محصولات مختلف متفاوت است، در هریک از کلاس‌های واقعی باید این عمل به‌صورت متفاوتی پیاده‌سازی شود. عمل sell برای محصولات مختلف یکسان است. اما، یک فروشگاه ممکن است چند سیاست محاسبهٔ هزینهٔ محصول داشته باشد: بی‌مالیات، بامالیات شهروندان، و بامالیات غیرشهروندان. اما تعداد این سیاست‌ها و روش محاسبهٔ مالیات در هر سیاست ثابت می‌باشد. مشخصهٔ taxFree با مالیات یا بی‌مالیات بودن محصول را مشخص می‌کند. شهروند بودن یا نبودن خریدار به‌عنوان ورودی عمل فروش به آن داده می‌شود.

این سلسله‌مراتب یک سلسله‌مراتب تکاملی است و به‌دلایلی که در قسمت ۴—۵ گفته شد، کارایی لازم را برای سیستم فروشگاه ندارد (مهم‌ترین دلایل عدم پشتیبانی از تعریف بدون برنامه‌نویسی محصولات جدید و زیاد شدن تعداد کلاس‌ها می‌باشد.) در ادامه با استفاده از سبک معماری مدل قابل تطبیق شیء مدلی ارائه می‌گردد که مسائل گفته شده در آن وجود نداشته باشد.

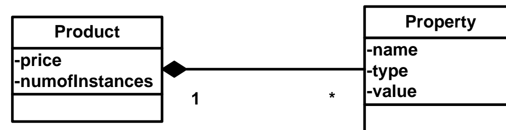
---

[۴۴] Abstract

[۴۵] Concrete



شکل ۴—۴: به‌کارگیری الگوی شیء قاعده برای تبدیل سلسله‌مراتب به یک مدل قابل تطبیق‌تر



شکل ۴—۵: به‌کارگیری الگوی مشخصه برای رفع مشکل تطبیق مشخصه‌ها

۴—۷—۲ ساختار قابل تطبیق برای سیستم فروشگاه

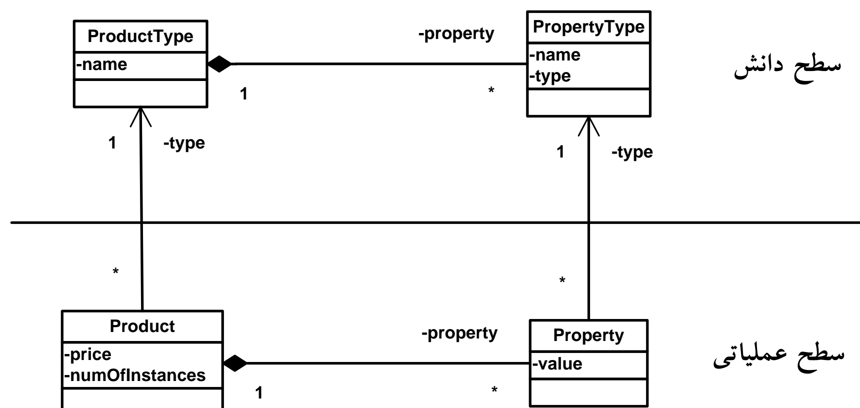
یک راه برای پیاده‌سازی تاکتیک داده به‌جای برنامه، استفاده از الگوهای طراحی شیء‌نوع[۴۶] و مشخصه[۴۷] برای غلبه بر تفاوت در ساختار موجودیت‌های سلسله‌مراتب است. با این روش هر نوع موجودیت با یک شیء (داده) مدل‌سازی می‌شود (تکنیک شیء به‌جای کلاس) که با انواع مشخصه‌های خود رابطهٔ ترکیب دارد (تکنیک ترکیب به‌جای وراثت). همچنین، نمونه‌های این انواع موجودیت نیز با مشخصه‌های خود رابطهٔ ترکیب خواهند داشت.

شکل ۴—۴ نشان می‌دهد که چگونه با استفاده از الگوی شیء نوع می‌توان مدل قابل تطبیقی برای این سلسله‌مراتب ارائه کرد. در این مدل هر نوع محصول با شیئی از کلاس ProductType و هر محصول با شیئی از کلاس Product در سیستم ثبت می‌شوند و وابستگی یک محصول به یک نوع محصول از طریق انتساب اشیاء آن‌ها نشان داده می‌شود. بنابراین، این الگو، تاکتیک داده به‌جای برنامه را محقق می‌کند زیرا کلاس‌هایی که انواع محصول را پیاده‌سازی می‌کردند، جای خود را به اشیاء داده‌اند و نمونه بودن جای خود را به انتساب داده است. یک نکتهٔ قابل توجه در این شکل وجود سطوح دانش و عملیاتی (شبیه شکل ۴—۱) می‌باشد. این دو سطح با نام متا و پایه[۴۸] نیز شناخته می‌شوند.

[۴۶] Type Object Design Pattern

[۴۷] Property Design Pattern

[۴۸] Base



شکل ۴—۶: مربع نوع: دو بار استفاده از هریک از الگوهای شیء نوع و مشخصه

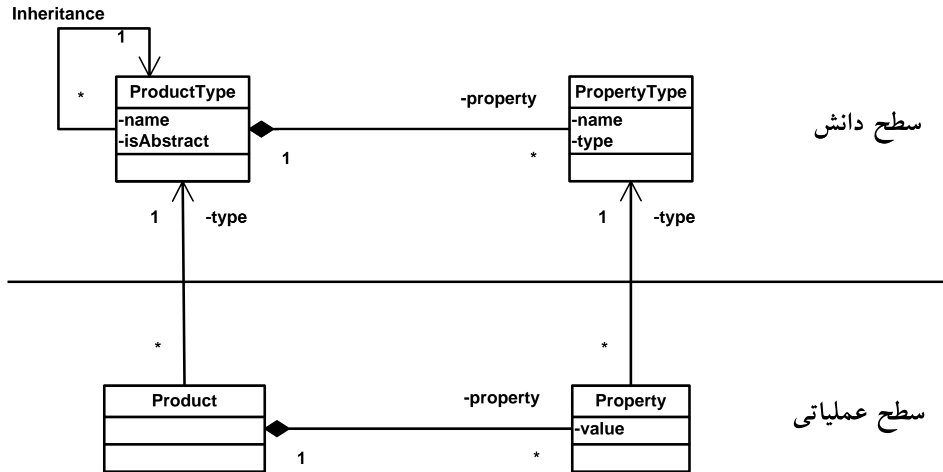
در این مدل نیز هنوز اشکالاتی در زمینهٔ قابلیت تطبیق وجود دارد، زیرا همان‌طورکه در شکل مشخص است، کلاس Product باید از مشخصه‌های انواع مختلف محصول پشتیبانی کند و در تعریف محصولات جدید که مشخصه‌های جدید دارند، لازم است این مشخصه‌ها از طریق برنامه‌نویسی به کلاس Product اضافه شوند. برای حل این مشکل می‌توان از الگوی دیگری به‌نام مشخصه[۴۹] استفاده کرد. شکل ۴—۵ کاربرد این الگو برای مثال مورد بحث نشان می‌دهد. این الگو با استفاده از تکنیک ترکیب به‌جای وراثت، تاکتیک داده به‌جای برنامه را محقق می‌کند.

از آنجا که لازم است نوع مشخصه‌ها را نیز بتوان به‌صورت پویا تعریف کرد، می‌توان یک بار دیگر از الگوی مشخصه استفاده کرد. حاصل شکل ۴—۶ است که مربع نوع[۵۰] نامیده می‌شود. در این مدل ارتباط هر مشخصه با نوع آن از طریق انتساب نشان داده می‌شود (الگوی شیء نوع). البته همان‌طورکه در شکل دیده می‌شود، می‌توان مشخصه‌هایی که در اشیاء مختلف مشترک هستند را، مثل name برای انواع محصول و price و numOfInstances برای محصولات، در خود کلاس‌ها قرار داد.

مدل به‌دست آمده توانایی مدل‌سازی موجودیت‌های واقعی (و نه مجرد)، که با کلاس‌های برگ از شکل ۴—۳ نشان داده شده‌اند، را دارد. به عبارت دیگر، این مدل از وراثت موجودیت‌ها پشتیبانی نمی‌کند، و درنتیجه امکان استفادهٔ مجدد از تعریف موجودیت‌ها و اعمال را به متخصص دامنه نمی‌دهد. شکل ۴—۷ وراثت را به مربع نوع اضافه می‌کند. در این مدل، مجرد یا واقعی بودن انواع موجودیت با مشخصهٔ بولی isAbstract نشان داده می‌شود. در این مدل تغییر دیگری نیز در مربع نوع اعمال شده است. مشخصه‌های price و numOfInstances از کلاس Product حذف شده است. دلیل این مطلب این است که در شکل ۴—۶ با مشخصه‌ها به دو شکل متفاوت برخورد می‌شود. برخی مشخصه‌ها با استفاده از اشیاء کلاس PropertyType تعریف می‌شوند و برخی به‌صورت مشخصه‌های ایستا در کلاس Product حک می‌شوند. هدف از این کار فاکتورگیری از مشخصه‌های مشترک در تمام انواع محصول است. این روش یک اشکال اساسی دارد: در تمام برنامه‌ها باید با مشخصه‌ها به دو شکل متفاوت برخورد کرد که باعث پیچیدگی برنامه می‌شود. به‌علاوه، ممکن است لازم باشد یکی از این مشخصه‌های ایستا نیاز به تغییر داشته باشد که دراین‌صورت متن برنامه باید تغییر کند. حال که امکان فاکتورگیری از مشخصه‌ها از طریق وراثت مهیا شده

[۴۹] Property Pattern

[۵۰] Type Square



شکل ۴—۷: اضافه کردن وراثت به مربع نوع

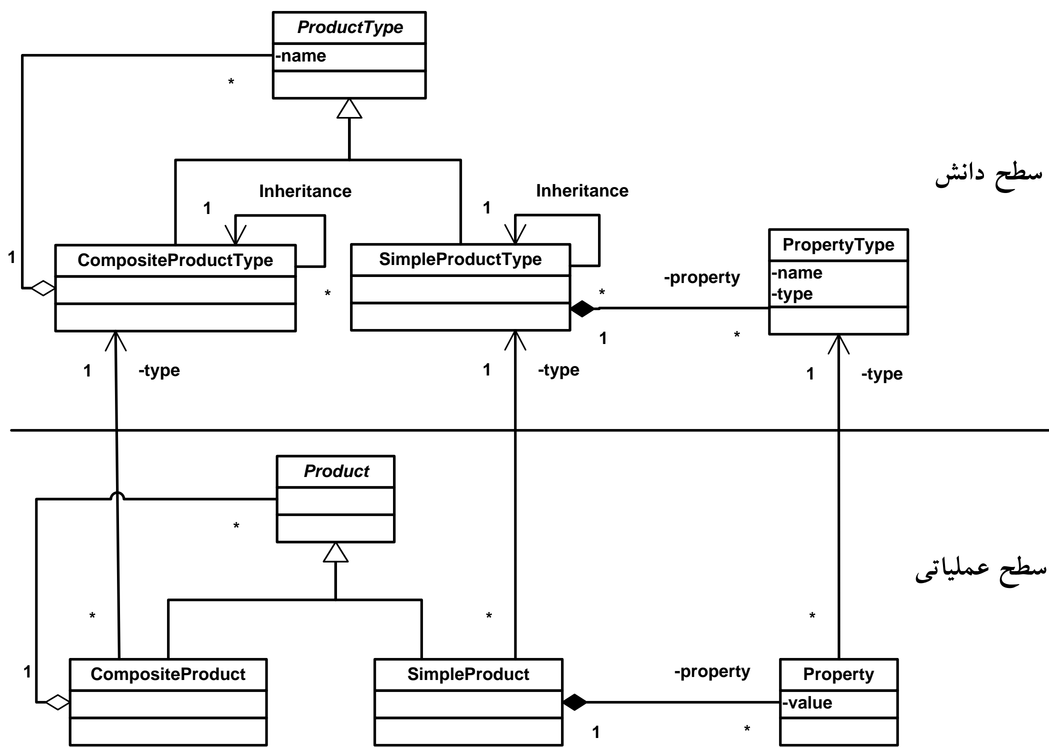
است، نیازی به تعریف مشخصه‌ها به‌شکل ایستا نیست. برای مدل‌سازی سیستم فروشگاه، متخصص دامنه می‌تواند ابتدا یک نوع محصول به‌نام Product با مشخصه‌های price و numOfInstances تعریف کند و بقیهٔ انواع محصول را به‌عنوان بچه‌های آن به سیستم وارد نماید. البته کارایی این روش به‌دلیل نیاز به دسترسی به مشخصه‌های غیرایستا پایین‌تر است. بااین‌حال، کاهش کارایی برای دست‌یابی به قابلیت تطبیق قابل قبول است.

مدل ارائه شده هنوز به‌طورکامل پاسخ‌گوی نیاز فروشگاه نیست، زیرا نمی‌توان موجودیت‌های مرکب ایجاد کرد. راه‌حل این مشکل استفاده از الگوی طراحی مرکب[۵۱] می‌باشد. شکل ۴—۸ به‌کارگیری این الگو را نشان می‌دهد. در این مدل، الگوی مرکب هم در مورد نوع موجودیت‌ها به‌کارگرفته شده است و هم در مورد خود موجودیت‌ها. در این مدل، یک نوع محصول می‌تواند ترکیبی از انواع محصولات ساده یا مرکب دیگر باشد. همچنین، یک محصول نیز می‌تواند، به‌شکلی که در نوع آن مشخص شده، ترکیبی از محصولات ساده یا مرکب دیگر باشد. این اطلاعات، هیچ‌کدام، در سیستم برنامه‌نویسی نمی‌شوند و همه به‌صورت داده نگه‌داری می‌شوند تا قابل تغییر باشند.

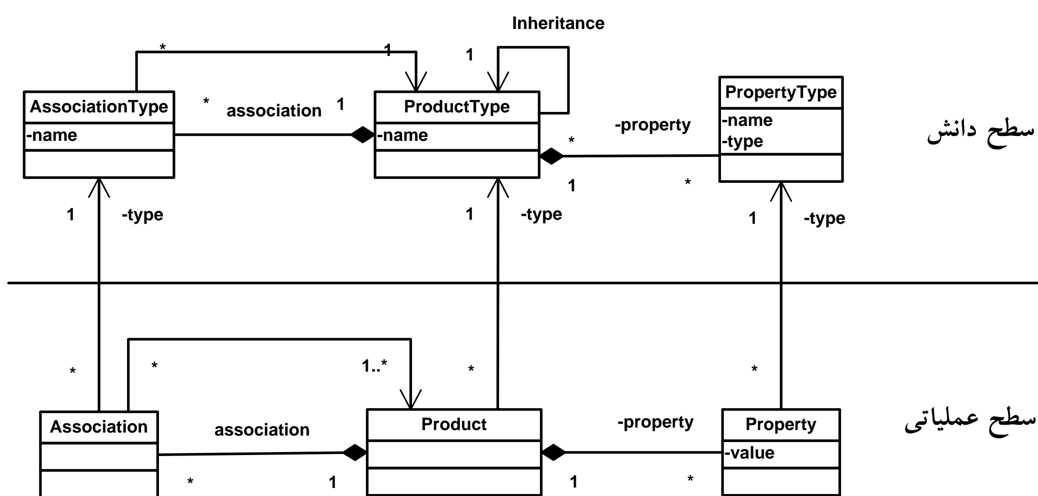
ترکیب محصولات حالت خاصی از انتساب بین محصولات می‌باشد. در مدل شکل ۴—۸ یک محصول مرکب نمی‌تواند مشخصه داشته باشد. دراین‌صورت، اگر محصول مرکبی باشد که علاوه بر اجزاء مشخصه‌های خاصی نیز داشته باشد با این مدل قابل تعریف نیست. شکل ۴—۹ یک مدل عمومی برای تعریف انتساب بین محصولات را نشان می‌دهد. در این مدل، یک انتساب ممکن است تک‌مقداری یا چندمقداری باشد. این مدل حالت کلی مدل شکل ۴—۸ است.

در مدلی که در شکل ۴—٦ برای مشخصه‌ها ارائه شد، و هنوز در شکل ۴—۹ نیز دیده می‌شود، تمام انواع مشخصه‌های ممکن یک موجودیت در مشخصهٔ value از اشیاء کلاس Property نگه‌داری می‌شوند. این مدل بااین‌که عملی است (برای مثال value می‌تواند تمام انواع مشخصه‌را به‌صورت رشته در خود نگه‌داری کند)، ممکن است مناسب نباشد زیرا پوشش مشخصه‌های چندمقداری در این روش دشوار است و همچنین، به‌دلیل تبدیل بین انواع کارایی پایینی دارد. یک راه دیگر این است که برای هرنوع دادهٔ ابتدایی یک مشخصهٔ ساده و یک مشخصهٔ آرایه‌ای در کلاس Property قرار داده شود. پیاده‌سازی این راه نیز

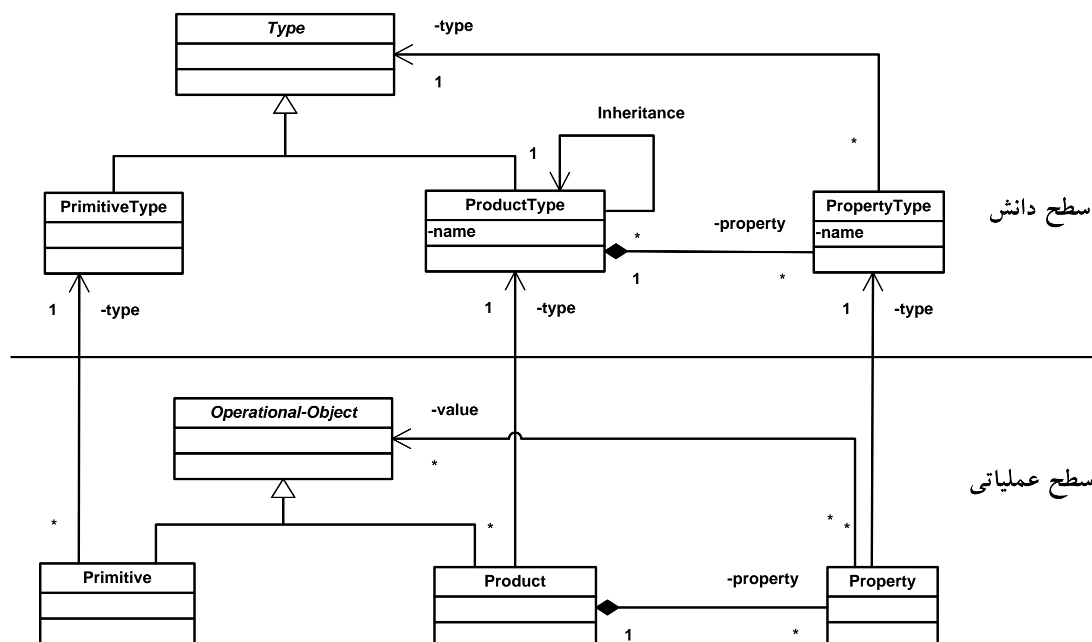
[۵۱] Composite Design Pattern



شکل ۴–۸: دو بار به‌کارگیری الگوی مرکب برای مدل‌سازی محصولات ترکیبی



شکل ۴–۹: مدل عمومی برای انتساب بین محصولات



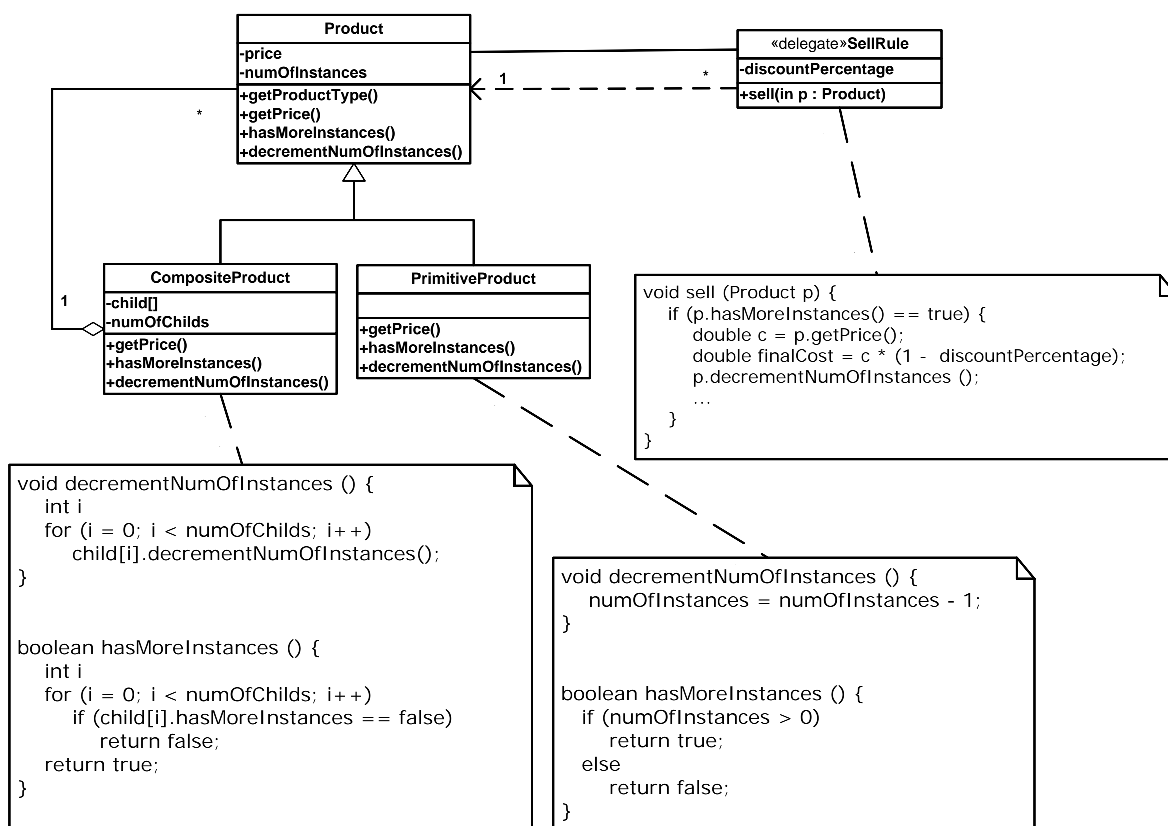
شکل ۴—۱۰: فاکتورگیری از مدل مشخصه و انتساب

مشکلاتی دارد. به‌عنوان مثال، هر عملی‌که قرار است روی مشخصه‌ها کار کند باید جملات شرطی متعددی PropertyType را بررسی کند تا تصمیم بگیرد کدام یک از مشخصه‌های مربوط به نوع ابتدایی را استفاده کند. یک راه دیگر برای حل این مشکل قرار دادن مشخصه‌های ابتدایی در اشیاء جداگانه است. این راه‌حل شبیه راه‌حل شکل ۴—۹ برای انتساب است. برای بالابردن استفادهٔ مجدد و راحت‌تر شدن ایجاد سیستم می‌توان انتساب و مشخصه را به‌صورت نشان داده شده در شکل ۴—۱۰، ترکیب کرد. این مدل بی‌شباهت به پشتیبانی زبان‌های برنامه‌نویسی شیءگرا از انتساب نیست. در این زبان‌ها، معمولاً، انتساب با استفاده از متغیرهای مرجع مدل می‌شود.

نمی‌توان گفت این مدل بهترین و کامل‌ترین مدل برای مدل‌سازی ساختار یک سلسله‌مراتب تکاملی است، زیرا در کاربردهای مختلف مدل‌های مختلفی مناسب هستند.

۴—۷—۳ رفتار قابل تطبیق برای سیستم فروشگاه

در بالا گفته شد که پیدایش یک سلسله‌مراتب دو ریشه دارد: تفاوت در ساختار و تفاوت در رفتار. در مدلی که تابه‌حال به آن رسیده‌ایم، مشکل تفاوت در ساختار حل شده است ولی از رفتار صحبتی به‌میان نیامده است. یکی از اعمال تعریف شده بر روی سلسله‌مراتب شکل ۴—۳ عمل sell می‌باشد. این عمل برای محصولات مختلف سلسله‌مراتب متفاوت است ولی تفاوت آن به سه گروه بی‌مالیات، بامالیات شهروندان، و بامالیات غیرشهروندان محدود است. بنابراین، می‌توان آن را طوری ایجاد کرد که تمام محصولات فعلی و آینده را پوشش دهد. یک راه برای این کار این است که عمل sell از کلاس Product تمام حالات ممکن را پوشش دهد (تاکتیک عمومی‌کردن برنامه‌ها از طریق پارامتری کردن). این روش مشکل را حل می‌کند، اما به یک



شکل ۴—۱۱: پیاده‌سازی عمل sell با استفاده از الگوی راهبرد

برنامهٔ مفصل که به‌روزرسانی آن دشوار است می‌انجامد. یک طراحی بهتر استفاده از الگوی راهبرد[۵۲] برای جداسازی سیاست‌های مختلف محاسبهٔ مالیات می‌باشد. با استفاده از این الگو، می‌توان یک عمل از یک کلاس را به کلاس دیگری منتقل کرد. اشیاء کلاس اول با استفاده از اشیاء کلاس دوم، عمل مورد نظر را اجرا می‌کنند. شکل ۴—۱۱ استفاده از الگوی راهبرد برای عمل sell را نشان می‌دهد. در مورد اعمال پیچیده‌تری که مثل عمل sell تغییرات آن‌ها محدود است می‌توان این روش را با استفاده از الگوهای مفسر و شیء قاعده کامل‌تر کرد.

راه‌حل‌هایی که در [۲۵، ۱۰] برای پویایی رفتار پیشنهاد شده، از چنین اعمالی پشتیبانی می‌کند. در [۲۵] رفتار شامل ارزیابی قواعدی است که با استفاده از الگوهای طراحی شیء قاعده[۵۳] و راهبرد تعریف و اجرا می‌شوند. برای هر مشاهدهٔ پزشکی ساده می‌توان یک شئ قاعده تعریف کرد که برای اعتبارسنجی[۵۴] مقدار مشاهدهٔ ساده (مثلاً محدودهٔ مشاهدات کمّی مانند دمای بدن) استفاده شود. در [۱۰] با استفاده از الگوهای راهبرد و مفسّر[۵۵] روی یک مؤلفهٔ مرکب می‌توان محاسبه‌ای انجام داد که مقدار آن تابع مشخصی از مقدار اجزای آن است.

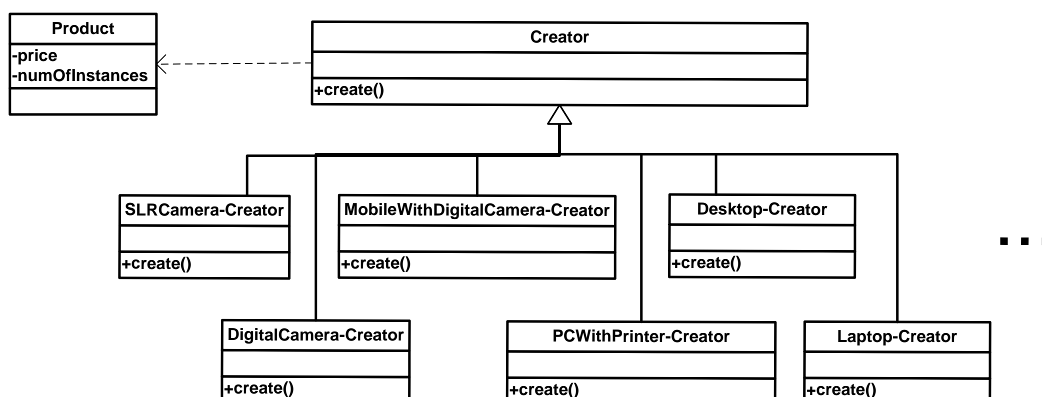
---

[۵۲] Strategy

[۵۳] Rule Object Design Pattern

[۵۴] Validation

[۵۵] Interpreter Design Pattern

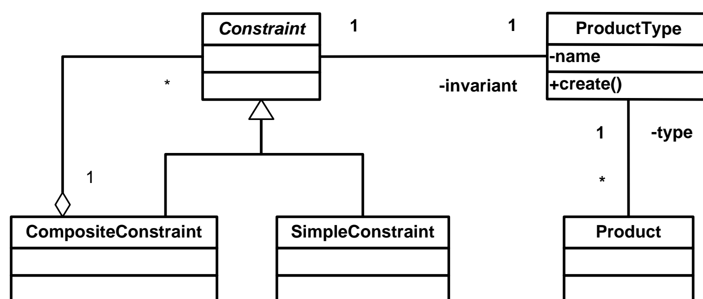


شکل ۴—۱۲: اضافه کردن عمل create با استفاده از الگوی راهبرد

اما در مورد عمل create وضعیت متفاوت است، زیرا مشخصه‌های انواع مختلف محصول با یکدیگر متفاوت هستند و در ایجاد یک نمونه از هریک، کاربر باید مشخصات جداگانه‌ای را وارد کند. فرض کنید در مورد عمل create از روشی که در مورد sell برای غلبه بر تفاوت‌ها به‌کارگرفته شد، استفاده شود. باتوجه‌به این‌که در مورد یک سلسله‌مراتب با خانواده‌ای از اعمال مواجه هستیم با استخراج این اعمال از سلسله‌مراتب، به سلسله‌مراتبی از کلاس‌های عمل می‌رسیم که در بدترین حالت (حالتی که آن عمل در تمام موجودیت‌ها متفاوت است) با سلسله‌مراتب قبلی هم‌ریخت[۵۶] است و تعداد کلاس‌های آن‌ها با هم برابر است. در سایر موارد، کلاس‌های راهبرد کمتر از کلاس‌های اولیهٔ موجودیت هستند. شکل ۴—۱۲ این حالت را برای مثال فروشگاه نشان می‌دهد. از آنجا که بسیاری از کلاس‌ها در مثال فروشگاه مجرد[۵۷] هستند، لازم نیست کلاس راهبرد برای آن‌ها پیاده سازی شود. بنابراین، در شکل به‌ازای هر کلاس واقعی[۵۸] از سلسله‌مراتب شکل ۴—۳ یک کلاس راهبرد وجود دارد. بااین‌وجود، به‌دلیل این‌که لازم است برای هر عمل مثل create، چنین سلسله‌مراتبی تعریف شود، باز هم احتمالاً تعداد کلاس‌ها بیش از سلسله‌مراتب اولیه می‌شود. حداکثر تعداد کلاس‌های راهبرد برابر است با حاصل‌ضرب تعداد اعمال و تعداد کلاس‌های واقعی نوع محصول. بنابراین، همان‌طور که در [۱۰] نیز بیان شده است، مشکل از این طریق حل نشده است و تنها سلسله‌مراتب به‌جا(ها)ی دیگری منتقل شده است.

بنابراین، یک مدل ساده مشابه آن‌چه در مورد sell نشان داده شد، نمی‌تواند مشکل را حل کند. برای حل این مشکل می‌توان سطح خودنگری از انعکاس موجود در زبان‌های برنامه‌نویسی شیءگرا را شبیه‌سازی کرد (انعکاس زبان‌های برنامه‌نویسی و خودنگری در قسمت ۲—۵—۵ توضیح داده شد.) به‌عبارت‌دیگر، می‌توان برنامه‌ای نوشت که با داشتن اطلاعات مربوط به یک موجودیت (مشخصه‌ها، محدودیت‌ها و ...) رفتار خود را با آن موجودیت تطبیق دهد. نکتهٔ قابل توجه این است که ماهیت عمل create برای انواع مختلف محصول یکسان است و تنها مشخصه‌های محصولات و قواعد حاکم بر آن‌ها متفاوت می‌باشد. بنابراین، می‌توان از رفتار حاکم بر سلسله‌مراتب انواع محصول فاکتورگیری کرد و برای اجرای یک عمل عمومی بر روی یک شیء، شیء نوع را به‌عنوان پارامتر عمل در نظر گرفت (تاکتیک عمومی کردن برنامه‌ها از طریق پارامتری کردن). شکل ۴—۱۳ محل قرار گیری عمل create و شکل ۴—۱۴ متن برنامهٔ نوشته

[۵۶] Homomorphic
[۵۷] Abstract
[۵۸] Concrete



شکل ۴–۱۳: قرار دادن یک نسخه از عمل create در مدل

```
void create () {
   for each PropertyType of this ProductType or its SuperTypes{
      insert an appropriate input box on a user interface form;
      show the form to user;

      when user clicks 'save' {
         create product as an instance of Product;

         if (constraint.assess(product) == true)
            save product to database;
         else
            show appropriate error message to user;
      }
   }
   ...
}
```

شکل ۴–۱۴: پیاده‌سازی عمل create با استفاده از یک برنامهٔ عمومی

شده برای این عمل را نشان می‌دهد. از آن جاکه در مدل فروشگاه کلاس‌های سلسله‌مراتب جای خود را به اشیاء داده‌اند نیازی به استفاده از انعکاس واقعی (با استفاده از کلاس‌ها) نیست و کافی است اشیاء نوع مشخصه (PropertyType) از یک نوع محصول (ProductType) بررسی شوند و در زمان اجرا یک واسط کاربر مناسب برای ورود اطلاعات یک محصول ایجاد شود. در پایان عمل نیز پیش از ثبت محصول بررسی می‌شود که قوانین حاکم بر مشخصه‌های محصول رعایت شده باشند. از جمله مواردی که در مدل جدید اضافه شده‌اند قوانین ناظر بر نوع موجودیت هستند که با کلاس Constraint مدل شده‌اند. هر شیء از کلاس SimpleConstraint یک عبارت سادهٔ بولی[۵۹] را مدل می‌کند (مثل $x < ۱۰$) و اشیاء کلاس CompositeConstraint این عبارات ساده را با عملگرهای بولی (مثل AND) ترکیب می‌کنند (الگوی مرکّب). در نتیجه، این محدودیت‌ها، که در شکل ؟؟ در متن برنامه قرار داشتند، اکنون به‌صورت داده مدل‌سازی شده‌اند. (به‌کارگیری مجدد تاکتیک داده به‌جای برنامه می‌باشد.) بنابراین، علاوه بر انواع مشخصه (که در شکل نشان داده نشده‌اند) یک نوع محصول با یک شیء مشخص‌کنندهٔ محدودیت نیز در ارتباط می‌باشد که نقش یک ثابت[۶۰] را برای نوع محصول ایفا می‌کند.

حال سناریوهای تغییر زیر را در نظر بگیرید:

۱) یک محصول جدید، مثل پیراهن، به محصولات فروشگاه اضافه شود.

۲) امکان پس‌دادن برای برخی انواع محصول به سیستم اضافه شود.

۳) عمل جایزه به سیستم اضافه شود: به یک مشتری‌ای که بیشترین خرید را انجام داده، در صورتی‌که، میزان خرید او بیش از ۲۰۰۰۰۰۰ ریال بوده است جایزه تعلق بگیرد و این مطلب از طریق پست الکترونیکی به او اطلاع داده شود.

سناریوی اول به‌طور کامل توسط این سیتم پوشش داده می‌شود زیرا با انواع جدید محصول قابل تعریف هستند و اعمال sell و create طوری نوشته شده‌اند که می‌توانند برای هر نوع محصول جدیدی اجرا شوند. اما واضح است که مدل ارائه شده توانایی پشتیبانی از سناریوهای ۲ و ۳ را ندارد. راه‌حلی که در قسمت ۵—۲ ارائه می‌شود، می‌تواند از این سناریوها نیز پشتیبانی کند.

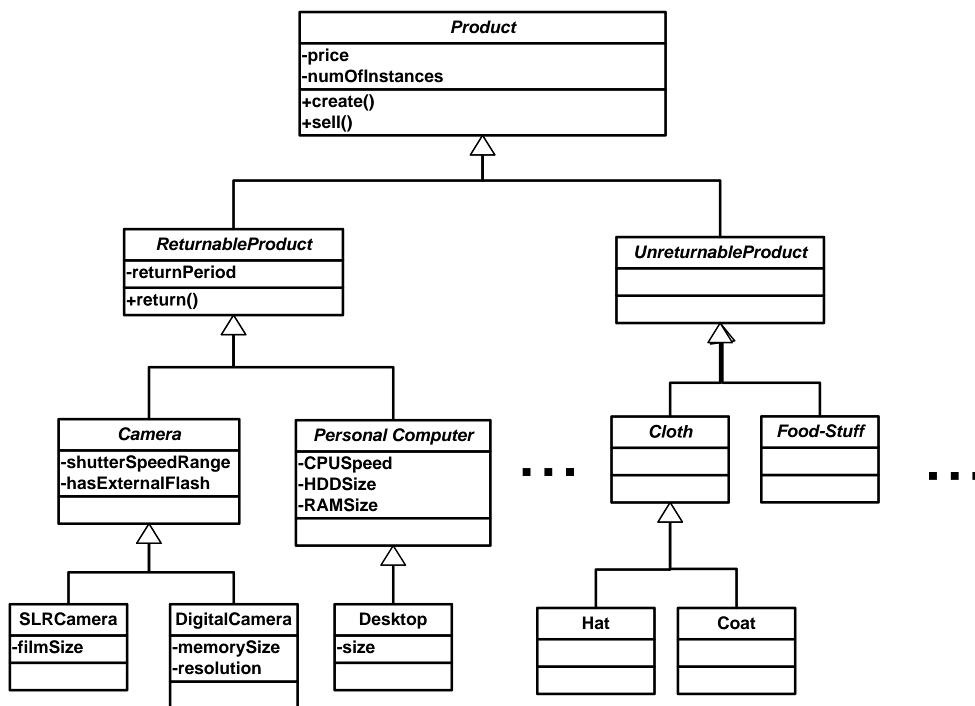
نکتهٔ دیگری که در مورد ارائهٔ یک مدل قابل تطبیق برای سلسله‌مراتب‌ها قابل توجه است، این است که همیشه یک سلسله‌مراتب به یک مربع‌نوع ساده تبدیل نمی‌شود. برعکس، ممکن است نیاز باشد یک طبقه‌بندی موجود تا چند سطح در مدل قابل تطبیق نیز وجود داشته باشد. برای مثال اگر وجود محصولات قابل بازگشت در زمان مهندسی دامنه تشخیص داده شده بود، سلسله‌مراتب سیستم به‌صورت شکل ۴—۱۵ درمی‌آمد تا پیاده‌سازی عمل پس‌دادن را برای گروهی از محصولات پشتیبانی کند. همان‌طورکه مشخص است، این عمل مخصوص یک زیردرخت خاص از سلسله‌مراتب می‌باشد.

شکل ۴—۱۶ یک مدل قابل تطبیق برای این سلسله‌مراتب را نشان می‌دهد. جزئیات مربوط به مشخصه و انتساب که در این مدل نشان داده نشده است، مانند شکل ۴—۱۰ می‌باشد. عمل پس‌دادن و محصولات قابل بازگشت را با استفاده از یک مشخصهٔ بولی که نشان‌دهندهٔ قابل بازگشت بودن محصول است، نیز می‌توان مدل کرد. اما در مورد اعمال پیچیده‌تر ممکن است لازم باشد سلسله‌مراتب تا سطوح پایین‌تر از ریشه، در مدل قابل تطبیق ظاهر شود. بنابراین، مدل قابل تطبیق معادل کاهش سلسله‌مراتب است نه حذف آن.

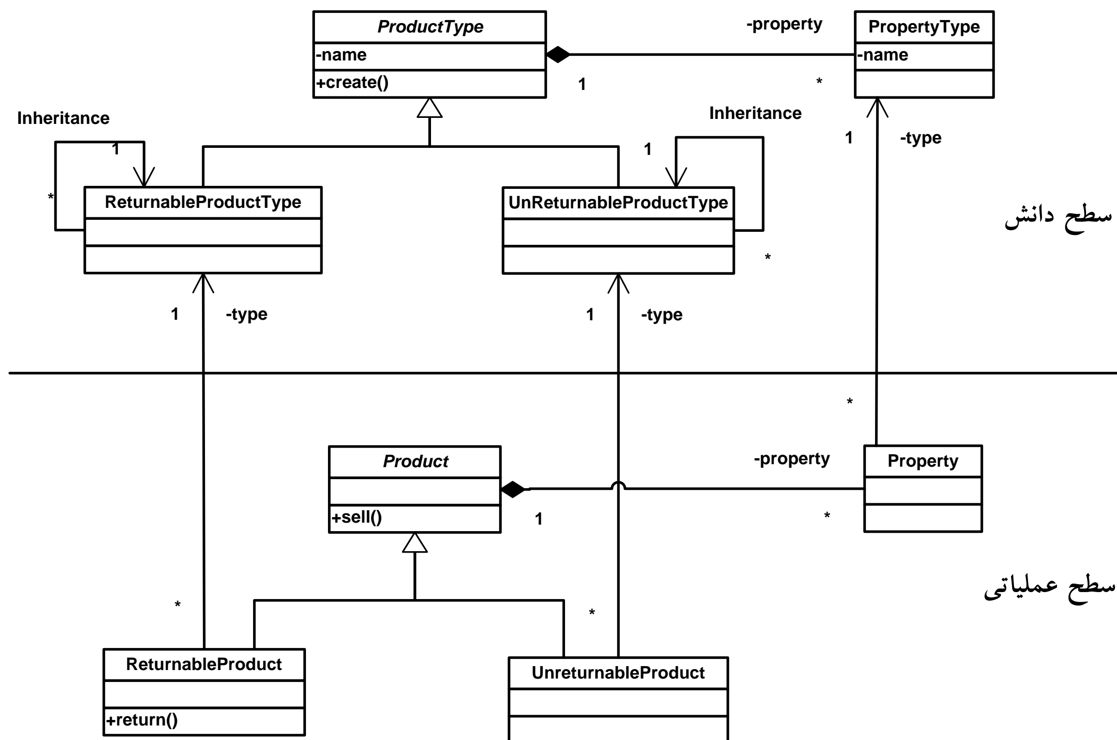
---

[۵۹] Boolean

[۶۰] Invariant



شکل ۴–۱۵: سلسله‌مراتب شامل محصولات قابل بازگشت



شکل ۴–۱۶: مدل قابل تطبیق سلسله‌مراتب شامل محصولات قابل بازگشت

در این قسمت نشان داده شد که چگونه می‌توان با استفاده از چند الگوی ساده، تاکتیک داده به‌جای برنامه را در مورد ساختار پیاده کرد و یک سلسله‌مراتب بزرگ و پیچیده را به‌صورتی قابل تطبیق درآورد. این روش به‌قدری کلی و قدرتمند است که می‌تواند مشکل تطبیق ساختاری هر سلسله‌مراتب تکاملی را حل کند. البته، لازم به‌تذکر است که این مدل تنها یک شکل مدل‌سازی با استفاده از الگوهای نامبرده را نشان می‌دهد. بر حسب نیاز به‌طرق مختلفی می‌توان یک مدل قابل تطبیق ایجاد کرد. به‌عنوان مثال، ممکن است لازم باشد یک نوع موجودیت زیر نوعی‌از یک نوع موجودیت دیگر باشد، تا بتوان از تعریف موجودیت‌ها استفادهٔ مجدد کرد. اما به‌هرحال، این مدل ایدهٔ اصلی روش محقق کردن تاکتیک‌های قسمت ۴—٦ را نشان می‌دهد. این روش از دو جهت ناقص می‌باشد.

۱) همان‌طور که با استفاده از سناریوهای اعمال جدید نشان داده شد، این روش از پشتیبانی از رفتارهای پیش‌بینی نشده ناتوان است. در قسمت ۴—۸، توضیحاتی در مورد مشکلات تطبیق رفتاری ارائه می‌گردد.

۲) مثال فروشگاه حالت خاصی از سیستم‌های شیءگرا را نشان می‌دهد که در آن تنها یک سلسله‌مراتب وجود دارد. قسمت ۵—۱ یک مدل کلی برای سیستم‌های شیءگرا عرضه می‌کند.

۳) طبق نظر مؤلفین این سبک معماری، وقتی استفاده از این روش در مورد یک سلسله‌مراتب مناسب است که رفتار کلاس‌های سلسله‌مراتب تفاوت اندکی داشته باشد [۲۵]. این فرض نیز به کلیت روش لطمه وارد می‌کند.

در قسمت ۵—۲ با گسترش راه‌حل تطبیق یک سلسله‌مراتب، یک راه‌حل کلی برای تطبیق ساختاری و رفتاری سیستم‌های شیءگرا، ارائه می‌گردد.

۴—۸ دشواری تطبیق رفتاری در مقایسه با تطبیق ساختاری

ساختار یک سیستم، اساساً، پیچیدگی کمتری نسبت به رفتار آن دارد، زیرا اولاً حجم اطلاعاتی که ساختار یک نوع موجودیت در خود دارد، معمولاً، بسیار کمتر از رفتار آن نوع موجودیت است[٦١]. ثانیاً در مورد موجودیت‌های حرفه (برخلاف کلاس‌هایی که برای توصیف داده‌ساختارهای پیچیده به‌کار می‌روند) تنوع در ترکیب‌های ساختار بسیار کمتر از ترکیب‌های رفتاری است زیرا ساختارهایی از یک زبان برنامه‌نویسی را که ساختار برنامه را توصیف می‌کنند، (مثل انواع داده[٦٢]) به‌شکل‌های کمتری می‌توان در یک جملهٔ برنامه باهم ترکیب کرد. به‌عنوان مثال در زبان C++ کلمات مشخص‌کنندهٔ انواع داده مثل int و char را نمی‌توان باهم ترکیب کرد و بااین‌که می‌توان با استفاده از کلماتی مثل آرایه[٦٣] و اشاره‌گر[٦٤] با همان انواع ساختارهای جدیدی تعریف کرد، اما به‌طور معمول، ساختارهای بسیار پیچیده در سیستم‌های مدل‌سازی حرفه کمتر مورد نیاز هستند. در ضمن، از ترتیب مختلف جملاتی که ساختار را توصیف می‌کنند، ساختار جدیدی به‌وجود نمی‌آید و انتخاب و تکرار نیز برای جملات ساختاری معنی ندارد. از این مطالب می‌توان نتیجه گرفت که

[٦١] این مطلب را می‌توان به‌طرق مختلفی نشان داد: شمارش تعداد خطوط برنامه یا تعداد کلمات تخصیص داده شده به‌بعد ساختاری و رفتاری یک نوع موجودیت، راه‌هایی ساده برای این کار هستند.

[٦٢] Data Types

[٦٣] Array

[٦٤] Pointer

درک ساختار یک سیستم برای انسان (برنامه‌نویس یا غیربرنامه‌نویس) بسیار ساده‌تر از درک رفتار سیستم است. به‌عبارت‌دیگر، حتی اگر ساختار پویا با زبانی به‌پیچیدگی خود زبان برنامه‌نویسی نمایش داده شود، باز هم آموزش آن به متخصص دامنه عملی است. بنابراین، در مدلی که در بالا ارائه شد در تعریف ساختار یک نوع موجودیت نه حجم زیادی از اطلاعات وجود خواهد داشت و نه پیچیدگی خاصی دیده می‌شود.

همان‌طور که در قسمت ۳—۳—۲ گفته شد، ورودی یک برنامه، خود یک گرامر دارد. بنابراین، می‌توان گفت سیستمی که در بالا برای تعریف ساختار به‌کارگرفته شد، خود شامل یک زبان می‌باشد. در سیستم‌های قابل تطبیق مختلف این زبان ممکن است جزئیات بیشتر یا کمتری داشته باشد. واضح است که هر چه این زبان جزئیات بیشتری داشته باشد، قدرت آن بیشتر و در مقابل درک و کار کردن با آن دشوارتر است. در پیچیده‌ترین حالت، این زبان جزئیاتی به‌اندازهٔ یک زبان برنامه‌نویسی همه‌منظوره دارد. طراحی چنین زبانی برای توصیف رفتار سیستم نیز عملی است (ساده‌ترین راه این است که از ساختارهای یک زبان برنامه‌نویسی همه‌منظوره استفاده شود) اما دراین‌صورت، اهداف گفته شده برای موتور تطبیق (قسمت ۴—۲) نقض شده‌اند. دلیل این مطلب این است که این راهبرد شبیه ایجاد مفسر یک زبان برنامه‌نویسی همه‌منظوره با استفاده از یک زبان برنامه‌نویسی دیگر است و همانطور که در بالا اشاره شد به‌دلیل پیچیدگی و حجم رفتار در برابر ساختار، کارکردن با چنین زبانی دشوار است و هدف سادگی استفاده از موتور تطبیق، برآورده نمی‌شود. بنابراین، تاکتیک داده به‌جای برنامه به‌تنهایی برای محقق کردن قابلیت تطبیق رفتاری کافی نیست.

در قسمت ۴—۷—۳ به‌طور خلاصه بیان شد که می‌توان با شبیه‌سازی انعکاس زبان‌های برنامه‌نویسی اعمالی عمومی ایجاد کرد. این روش با این‌که مناسب به‌نظر می‌رسد، کافی نیست. زیرا، ازآن‌جاکه موجودیت‌های سیستم متغیر هستند، نمی‌توان تمام اعمال را از قبل پیش‌بینی کرد. در نتیجه تنها اعمال شناخته شده را می‌توان با این روش ایجاد کرد.

در جمع‌بندی می‌توان گفت که مشکل قابلیت تطبیق رفتار دو جنبه دارد:

۱) پیچیدگی بیش از حد که امکان مدل‌سازی توسط متخصص دامنه را کاهش می‌دهد و به دشوار بودن اعمال تاکتیک داده به‌جای برنامه می‌انجامد.

۲) غیرقابل پیش‌بینی بودن اعمالی که در آینده مورد نیاز هستند که به دشوار بودن اعمال تاکتیک عمومی کردن برنامه‌ها از طریق پارامتری کردن می‌انجامد.

۴—۹ جمع‌بندی

در این فصل نشان داده شد که سیستم‌های قابل تطبیق با روش‌های معمول معماری و پیاده‌سازی قابل ایجاد نیستند و نیازمند یک نرم‌افزار زیربنایی به‌نام موتور تطبیق هستند. همچنین، چرخهٔ حیات ایجاد و بهره‌برداری از سیستم‌های قابل تطبیق بیان شد. سپس وجود سلسله‌مراتب‌های تکاملی به‌عنوان مشکلی در روش معمول طراحی شیءگرا معرفی گردید و سه تاکتیک «داده به‌جای برنامه»، «عمومی کردن برنامه‌ها از طریق پارامتری کردن»، و «ترکیب عملیات در زمان اجرا برای ساختن عملیات پیچیده‌تر» برای حذف این مشکل ارائه شد. در یک مثال ایده‌های کلی سبک معماری مدل قابل تطبیق شیء نشان داده شد، سپس در مورد ضعف‌های معماری ارائه شده برای این مثال و همچنین، دشواری تطبیق رفتاری در مقابل ساختاری توضیحاتی ارائه شد. در فصل بعد، یک روش پیشنهادی برای معماری موتورهای تطبیق، که این ضعف‌ها را برطرف می‌کند، ارائه می‌شود.

فصل ۵

راه‌حلی برای معماری موتورهای تطبیق

در فصل قبل یک مثال ساده از سیستم‌های قابل تطبیق مورد بررسی قرار گرفت و مشکلات راه‌حل از جمله کلی نبودن آن و عدم پشتیبانی از اعمال جدید نشان داده شد. در این فصل، یک راه‌حل عمومی برای ایجاد نرم‌افزارهای قابل تطبیق ارائه می‌گردد، که این مشکلات را نداشته باشد.

قسمت ۵–۱ یک مدل کلی از سیستم‌های شیءگرایی که نیازمند قابلیت تطبیق هستند ارائه می‌دهد و در قسمت ۵–۲ راه‌حلی عمومی برای تطبیق ساختاری و رفتاری، که مدل کلی ارائه شده برای سیستم‌های شیءگرا قابل تطبیق را پوشش می‌دهد، عرضه می‌گردد. قسمت ۵–۳ به بحث در مورد میزان استفاده از راه‌حل ارائه شده اختصاص یافته است و قسمت ۵–۴ به جمع‌بندی این فصل می‌پردازد.

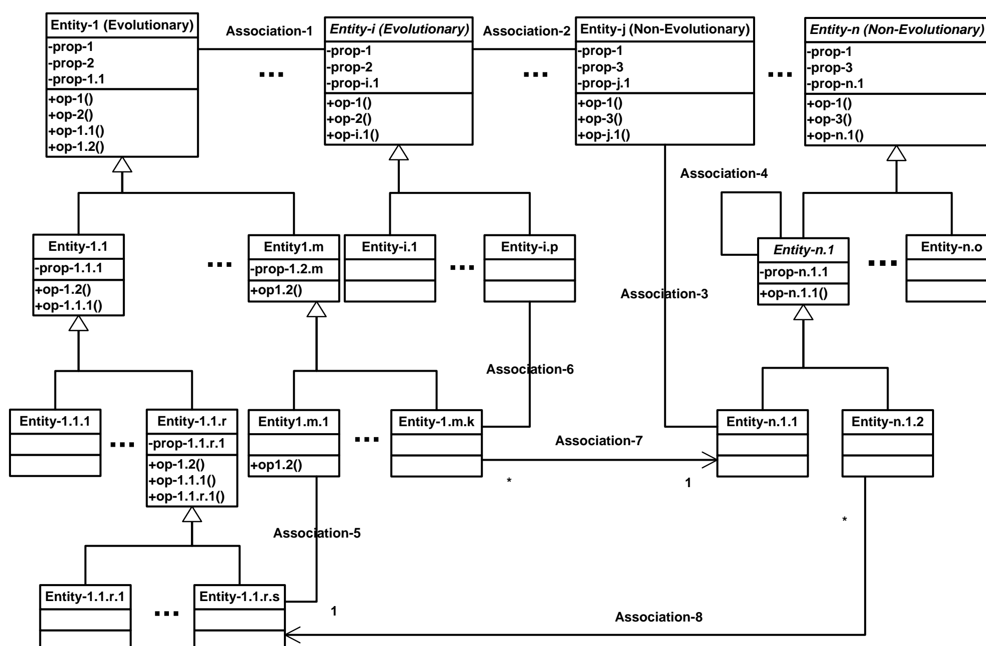
۵–۱ حالت کلی یک سیستم شیءگرا نیازمند تطبیق

مثال فروشگاه که در قسمت ۴–۷ ارائه شد، یک حالت خاص از سیستم‌های شیءگرا نیازمند تطبیق را نشان می‌دهد که در آن تنها یک سلسله‌مراتب تکاملی موجود است. در این قسمت یک حالت کلی برای سیستم‌های شیءگرا نیازمند تطبیق، از دید متخصص دامنه ارائه می‌گردد. منظور از دید متخصص دامنه، ابعادی از سیستم است که در هنگام تعریف و تغییر مدل محصول، برای یک متخصص دامنه قابل‌مشاهده هستند.

شکل ۵–۱ حالت کلی یک سیستم شیءگرا را نشان می‌دهد. در ادامه با استفاده از این نمودار، حالات مختلف سلسله‌مراتب، کلاس، وراثت، انتساب، مشخصه، و عمل یک سیستم شیءگرا قابل تطبیق، بررسی می‌شود.

۵–۱–۱ انواع سلسله‌مراتب

در حالت کلی یک سیستم شئ گرا قابل تطبیق ممکن است از چندین سلسله‌مراتب تشکیل شود. ازاین‌به‌بعد، از سلسله‌مراتب‌ها با اسم کلاس رأس سلسله‌مراتب نام برده‌می‌شود. به‌عنوان مثال، سلسله‌مراتب‌های شکل ۵–۱ عبارتند از: Entity-1 تا Entity-n. ازآن‌جاکه یک کلاس منفرد، مثل Entity-j، حالت خاص یک



شکل ۵—۱: حالت کلی سلسله‌مراتب‌های یک سیستم شیءگرا

سلسله‌مراتب غیرتکاملی است، بدون لطمه وارد شدن به کلیت بحث، درادامه، چنین کلاسی یک سلسله‌مراتب غیرتکاملی، خوانده می‌شود. همان‌طورکه قبلاً گفته شد، دو نوع سلسله‌مراتب در سیستم‌های شیءگرا وجود دارد:

۱) تکاملی: مانند سلسله‌مراتب‌های Entity-1 تا Entity-i

۲) غیرتکاملی: مانند سلسله‌مراتب‌های Entity-j تا Entity-n

ازآن‌جاکه سلسله‌مراتب‌های تکاملی و غیرتکاملی، از نظر مفاهیم شیءگرا، باهم تفاوتی ندارند، یک سلسله‌مراتب غیرتکاملی را نیز می‌توان به‌روش نشان داده شده در قسمت ۴—۷ به‌صورت قابل تطبیق درآورد. این روش، در حالت کلی، به کاهش کارایی سیستم می‌انجامد اما ممکن است در بعضی موارد سودمند باشد. در قسمت ۵—۳ در مورد کارایی سیستم‌های مبتنی بر مدل قابل تطبیق شیء، توضیح داده می‌شود. قسمت ۵—۲ فایدهٔ یک مدل قابل تطبیق برای سلسله‌مراتب‌های غیرتکاملی، را بیان می‌کند.

نتیجه: در یک سیستم قابل تطبیق، درصورت قابل قبول بودن افت کارایی، می‌توان برای سلسله‌مراتب‌های غیرتکاملی، یک پیاده‌سازی قابل تطبیق ارائه داد.

ازسوی‌دیگر، همان‌طور که گفته شد در روش‌های مرسوم مدل‌سازی، یک سلسله‌مراتب تکاملی به‌صورت غیرقابل تطبیق مدل می‌شود. بنابراین، صرف‌نظراز نوع سلسله‌مراتب، دو نوع پیاده‌سازی برای هر سلسله‌مراتبی قابل تصور است:

۱) پیاده‌سازی (یا مدل) قابل تطبیق

۲) پیاده‌سازی (یا مدل) غیرقابل تطبیق

باتوجه‌به هدف قابلیت تطبیق، یک نقطهٔ شروع مناسب برای طراحی سیستم این است که تمام سلسله‌مراتب‌های تکاملی به‌صورت قابل تطبیق مدل شوند و تمام سلسله‌مراتب‌های غیرتکاملی به‌صورت غیرقابل تطبیق مدل شوند. بنابراین، این قسمت با همین فرض نگاشته می‌شود. در قسمت ۵—۲ نشان داده می‌شود که چرا این روش همواره مناسب نیست.

همان‌طورکه در مثال فروشگاه نشان داده شد، در مدل‌های قابل تطبیق بخشی از کلاس‌های سلسله‌مراتب از مدل حذف شده، با اشیاء مدل می‌شود و بخشی باقی می‌ماند. به سطحی (عمقی) از سلسله‌مراتب که از آن به پایین کلاس‌های سلسله‌مراتب حذف شده‌اند سطح حذف می‌گوییم (خود سطح حذف نیز حذف شده است). بنابر آن‌چه گفته شد، در پیاده‌سازی شکل ۵—۱ از یک مدل غیرقابل تطبیق برای سلسله‌مراتب‌های Entity-j و Entity-n و از یک مدل قابل تطبیق برای سلسله‌مراتب‌های Entity-1 و Entity-i استفاده می‌شود. فرض کنید سطح حذف سلسله‌مراتب‌های Entity-1 و Entity-i سطح ۲ باشد (به عبارت دیگر، فقط کلاس رأس این سلسله‌مراتب‌ها باقی می‌ماند.) انتخاب سطح حذف برای هر سلسله‌مراتب، به‌عهدهٔ معماران سیستم است.

۵—۱—۲ انواع کلاس

در یک سیستم شیءگرا، دو نوع کلاس وجود دارد:

۱) کلاس واقعی: قرار است از آن اشیائی به‌وجود آید. در شکل به‌جز کلاس‌های Entity-i، Entity-n و Entity-n.1، بقیهٔ کلاس‌ها واقعی هستند.

۲) کلاس‌های مجرد: که قرار نیست از آن اشیائی به‌وجود آید، ولی به‌منظور استفادهٔ مجدد از طریق وراثت تعریف می‌شود. یک کلاس مجرد بخشی از اعمال زیرکلاس‌های خود را پیاده‌سازی می‌کند و زیرکلاس‌ها باید بقیهٔ اعمال را پیاده‌سازی کنند. کلاس‌های Entity-i، Entity-n و Entity-n.1 مجرد هستند.

همان‌طور که در قسمت ۴—۷ نشان داده شد، در سبک معماری مدل قابل تطبیق شیء، کلاس‌های واقعی با اشیاء مدل می‌شوند. یک موتور تطبیق که تنها امکان تعریف کلاس‌های واقعی را به متخصص دامنه می‌دهد (مثل مدلی که برای سیستم فروشگاه ارائه شد)، از نظر قدرت مدل‌سازی (توصیف انواع مختلف ساختار و رفتار موجودیت‌ها) تفاوتی با یک موتور تطبیق مشابه دارای قابلیت وراثت، ندارد. زیرا تمام سیستم‌های دارای وراثت را می‌توان بدون استفاده از این مفهوم پیاده‌سازی کرد. از دید برنامه‌نویسی شیءگرا، وراثت مکانیزمی است که استفادهٔ مجدد را میسر می‌کند و در نتیجه باعث سادگی در برنامه‌نویسی می‌شود. به‌طور مشابه، وراثت در یک مدل قابل تطبیق باعث سادگی کار متخصص دامنه می‌شود. بنابراین، در حالت کلی، پشتیبانی از وراثت موجودیت‌های تعریف‌شده توسط کاربر برای یک موتور تطبیق یک امتیاز قلمداد می‌شود.

باتوجه‌به این‌که در استفاده از یک موتور تطبیق مفهوم پیاده‌سازی وجود ندارد، تنها جایی‌که می‌توان امکان استفادهٔ مجدد به‌وجود آورد، تعاریفی است که متخصص دامنه به‌عنوان ورودی، به موتور تطبیق می‌دهد. بنابراین، پیاده‌سازی بخشی از اعمال در کلاس پدر، که در توضیح کلاس‌های مجرد به آن اشاره شد، در موتور تطبیق مفهومی ندارد. بااین‌حال، در این سیستم‌ها داشتن مفهوم موجودیت مجرد برای استفادهٔ مجدد از تعاریف، مورد نیاز است. نوع موجودیتی که کلاس Camera از مثال فروشگاه را مدل می‌کند، در صورت

وجود، باید مجرد باشد. همچنین، به‌همین‌دلیل، مفاهیم واسط و کلاس پیاده‌سازی که در برخی زبان‌های شیءگرا، مثل جاوا، وجود دارند، در مورد موتورهای تطبیق بی‌معنی هستند. و در شکل نیز نشان داده نشده‌اند.

۵–۱–۳ انواع وراثت

دو نوع وراثت در سیستم‌های شیءگرا وجود دارد:

۱) وراثت تک‌گانه: یک کلاس فقط یک پدر دارد.

۲) وراثت چندگانه: یک کلاس می‌تواند بیش از یک پدر داشته باشد.

پیاده‌سازی وراثت چندگانه در زبان‌های برنامه‌نویسی شیءگرا با ابهاماتی همراه است. به‌عنوان مثال در صورتی‌که دو پدر از یک شیء هر دو یک عمل را پیاده‌سازی کرده باشند، مترجم یا مفسر نمی‌تواند تشخیص دهد پیاده‌سازی مورد نظر در کلاس فرزند کدام است.

۵–۱–۴ انواع مشخصه

مشخصه‌های هر کلاس سلسله‌مراتب، به بچه‌های آن به‌ارث می‌رسد. به سطحی از سلسله‌مراتب که یک مشخصه در آن تعریف می‌شود، سطح مشخصه می‌گوییم. باتوجه‌به تقسیم‌بندی سلسله‌مراتب‌ها به تکاملی و غیرتکاملی، یک مشخصه از یک کلاس می‌تواند از یکی از انواع زیر باشد:

۱) متعلق به یک یا چند سلسله‌مراتب غیرتکاملی: فرقی نمی‌کند این مشخصه در چه سطحی از سلسله‌مراتب تعریف شود. مشخصه‌های prop-n.1 و prop-n.1.1 از سلسله‌مراتب Entity-n و مشخصه prop-3 که بین سلسله‌مراتب‌های Entity-j و Entity-n مشترک است، از این نوع می‌باشند.

۲) متعلق به یک سلسله‌مراتب تکاملی و با سطحی بالاتر از سطح حذف (به عبارت دیگر، کلاسی که مشخصه را تعریف کرده بود، هنوز در سلسله‌مراتب دیده می‌شود.) مشخصهٔ و prop-1.1 از سلسله‌مراتب Entity1 از این نوع است.

۳) متعلق به یک سلسله‌مراتب تکاملی و باسطحی پایین‌تر از سطح حذف (به عبارت دیگر، کلاسی که مشخصه را تعریف کرده بود دیگر در سلسله‌مراتب دیده نمی‌شود.) مشخصه‌های prop-1.1.1 و prop-1.1.r.1 از سلسله‌مراتب Entity1 از این نوع هستند.

۴) مشترک در چند سلسله‌مراتب تکاملی (و منحصر به این سلسله‌مراتب‌ها): مشخصهٔ prop-2 بین دو سلسله‌مراتب Entity-1 و Entity-i مشترک است، از این نوع می‌باشد.

۵) مشترک در چند سلسله‌مراتب تکاملی و غیرتکاملی: عمل prop-1 که بین سلسله‌مراتب‌های Entity-1، Entity-i، Entity-j، و Entity-n مشترک است، از این نوع می‌باشد.

موارد این دسته‌بندی برحسب دشواری در ارائهٔ یک راه‌حل عمومی دربرگیرندهٔ استفادهٔ مجدد، به‌صورت صعودی مرتب شده است. پشتیبانی از استفادهٔ‌مجدد در این حالات می‌تواند به‌سادگی سیستم کمک کند ولی ممکن است مشکلاتی نیز به‌دنبال داشته باشد.

۵-۱-۵ انواع انتساب ماندگار

بین اشیاء یک سیستم شیءگرا ارتباطاتی از قبیل وابستگی [۱] و انتساب [۲] وجود دارد. وابستگی حالت کلی ارتباطات بین اشیاء را نشان می‌دهد. هر نوع ارتباط، از جمله فراخوانی، ارسال پیام، و ارتباط ماندگار بین اشیاء، یک وابستگی قلمداد می‌شود. انتساب که یک حالت خاص از وابستگی است به‌معنی وجود مشخصه‌ای از نوع یک شیء، در شیء دیگر است. انتساب بین اشیاء در حالت کلی چندبه‌چند است. باوجوداین‌که در یک موتور تطبیق، وابستگی‌ها و انتساب‌های متعددی وجود دارد، فقط یک حالت خاص از انتساب که نشان‌دهندهٔ ارتباط ماندگار بین اشیاء است توسط متخصص دامنه قابل مشاهده است و اهمیت دارد زیرا سایر انواع وابستگی و انتساب که به‌طور معمول در سیستم‌های شیءگرا به‌کار می‌روند، جنبهٔ پیاده‌سازی دارند و ربطی به مدل حرفه، که از دید متخصص دامنه مهم است، ندارند. ازآن‌جاکه انتساب در مورد تمام اشیاء یک کلاس یکسان تعریف می‌شود، در ادامه به انتساب بین اشیاء دو کلاس، انتساب بین آن دو کلاس می‌گوییم. حالات زیر برای چندی [۳] این انتساب‌ها قابل تصور است:

۱) یک‌به‌یک

۲) یک‌به‌چند

۳) چندبه‌چند

با تعریف یک موجودیت اضافه می‌توان یک انتساب چندبه‌چند را با دو انتساب یک‌به‌چند جایگزین کرد. بنابراین، فرض می‌کنید تنها انتساب‌های یک‌به‌چند و یک‌به‌یک در سیستم وجود دارند. بدون لطمه وارد شدن به کلیت بحث می‌توان فرض کرد در این انتساب‌ها چندی طرف اول بزرگ‌تر یا مساوی چندی طرف دوم است و بنابراین، مرجع (یا کلید خارجی) اشیاء کلاس دوم در اشیاء کلاس اول ذخیره می‌شود. باتوجه‌به این‌که برخی کلاس‌ها در پیاده‌سازی قابل تطبیق سلسله‌مراتب‌ها حذف می‌شوند، انواع مختلف انتساب ماندگار در یک سیستم قابل تطبیق عبارتند از:

۱) انتساب ماندگار بین دو کلاس حذف نشده (این کلاس‌ها ممکن است متعلق به سلسله‌مراتب‌های تکاملی یا غیرتکاملی باشند.): انتساب‌های Association-1، Association-2، Association-3، و Association-4 در شکل، از این نوع هستند.

۲) انتساب ماندگار بین دو کلاس حذف شده از یک سلسله‌مراتب تکاملی: انتساب Association-5 از این نوع است.

۳) انتساب ماندگار بین دو کلاس حذف شده از دو سلسله‌مراتب تکاملی: انتساب Association-6 از این نوع است.

۴) انتساب ماندگار بین یک کلاس حذف شده و یک کلاس حذف نشده (طرف دوم ممکن است متعلق به یک سلسله‌مراتب تکاملی یا غیرتکاملی باشد.): انتساب Association-7 از این نوع است.

۵) انتساب ماندگار بین یک کلاس حذف نشده و یک کلاس حذف شده (طرف اول ممکن است متعلق به یک سلسله‌مراتب تکاملی یا غیرتکاملی باشد.): انتساب Association-8 از این نوع است.

---

[۱] Dependency

[۲] Association

[۳] Multiplicity

در انواع ۱، ۲، و ۳ ممکن است یک کلاس در هر دو طرف یک انتساب ظاهر شود.

۵–۱–۶ انواع عمل

در هر سلسله‌مراتب یک سری اعمال برای کلاس‌های مختلف تعریف شده است. بدون لطمه وارد شدن به کلّیت مطلب، می‌توان فرض کرد اعمال هم‌نام عمل‌کرد شبیهی دارند و در نتیجه می‌توان آن‌ها را هم‌نوع نامید. بنابراین، نوع عمل[۴] با نام آن ارتباط دارد. به یک پیاده‌سازی از نوع عمل (که مخصوص یک یا چند کلاس از سلسله‌مراتب است) نمونهٔ عمل می‌گوییم. در یک سلسله‌مراتب، یک نوع عمل برای زیر درختی از کلاس‌ها تعریف می‌شود و برای بقیهٔ کلاس‌ها بی‌معنی است. به بالاترین سطحی که یک عمل در سلسله‌مراتب تعریف می‌شود سطح عمل می‌گوییم. به‌عنوان مثال عمل op-1.1.1 در سلسله‌مراتب Entity-1 در سطح ۲ قرار دارد. هر نوع عمل از نظر تعدّد نمونه‌ها در یک سلسله‌مراتب، ممکن است به یکی از دو دستهٔ زیر تعلق داشته باشد:

۱) تک نمونه‌ای: تنها یک پیاده‌سازی از عمل وجود دارد که برای تمام زیردرخت کلاس پیاده‌سازی کننده، استفاده می‌شود. اعمال op-1 و op-1.1 از سلسله‌مراتب Entity-1 تک‌نمونه‌ای هستند. البته عمل op-1 در سلسله‌مراتب‌های دیگر نیز پیاده‌سازی شده است اما در سلسله‌مراتب Entity-1 تک‌نمونه‌ای می‌باشد.

۲) چند نمونه‌ای: پیاده‌سازی‌های متعددی از عمل در کلاس‌های مختلف سلسله‌مراتب وجود دارد. عمل op-1.2 از سلسله‌مراتب Entity-1 چند نمونه‌ای است.

همان‌طور که در قسمت ۴–۷ نشان داده شد، برخی از اعمال در یک سیستم قابل تطبیق به‌صورت عمومی نوشته می‌شوند. به‌عبارت‌دیگر، یک عمل عمومی یک فاکتورگیری از نمونه‌های مختلف عمل است.

باتوجه‌به تقسیم‌بندی سلسله‌مراتب‌ها به تکاملی و غیرتکاملی انواع یک عمل عبارتند از:

۱) متعلق به یک یا چند سلسله‌مراتب غیرتکاملی: فرقی نمی‌کند این عمل در چه سطحی از سلسله‌مراتب پیاده‌سازی شود. اعمال op-n.1 و op-n.1.1 از سلسله‌مراتب Entity-n و عمل op-3 که بین سلسله‌مراتب‌های Entity-j و Entity-n مشترک است، از این نوع می‌باشند.

۲) متعلق به یک سلسله‌مراتب تکاملی و با سطحی بالاتر از سطح حذف (به عبارت دیگر، بالاترین کلاسی که عمل را تعریف کرده بود، هنوز در سلسله‌مراتب دیده می‌شود.) اعمال op-1.1 و op.1-2 از سلسله‌مراتب Entity1 از این نوع هستند.

۳) متعلق به یک سلسله‌مراتب تکاملی و با سطحی پایین‌تر از سطح حذف (به عبارت دیگر، بالاترین کلاسی که عمل را تعریف کرده بود، دیگر در سلسله‌مراتب دیده نمی‌شود.) اعمال op-1.1.1 و op-1.1.r.1 از سلسله‌مراتب Entity1 از این نوع هستند.

۴) مشترک در چند سلسله‌مراتب تکاملی (و منحصر به این سلسله‌مراتب‌ها): عمل op-2 که بین دو سلسله‌مراتب Entity-1 و Entity-i مشترک است، از این نوع می‌باشد.

---

[۴] Operation Type

۵) مشترک در چند سلسله‌مراتب تکاملی و غیرتکاملی: عمل op-1 که بین سلسله‌مراتب‌های Entity-1، Entity-i، Entity-j، و Entity-n مشترک است، از این نوع می‌باشد.

موارد این دو دسته‌بندی برحسب دشواری در ارائهٔ یک راه‌حل عمومی دربرگیرندهٔ استفادهٔ مجدد، به‌صورت صعودی مرتب شده است. پشتیبانی از استفادهٔ‌مجدد در این حالات می‌تواند به سادگی استفاده از سیستم کمک کند ولی ممکن است مشکلاتی نیز به‌دنبال داشته باشد.

اجزای این نمودار کلاس، برای توضیح حالات مختلف ممکن، در این شکل قرار داده شده‌اند. نمودار کلاس یک سیستم شیءگرا، هر شکل دیگری می‌تواند داشته باشد. اما، انواع سلسله‌مراتب، کلاس، وراثت، مشخصه، انتساب، و عمل آن، خارج از انواع بررسی شده در بالا نخواهد.

بخشی از طبقه‌بندی بالا ارتباطی به این کار ندارد و از طریق برنامه‌نویسی پشتیبانی می‌شود. این بخش شامل کلاس‌های مجرد و واقعی، انواع وراثت در طراحی غیرقابل تطبیق سلسله‌مراتب‌ها (طراحی متداول سیستم‌های شیءگرا)، و مورد ۱ از انواع ممکن برای مشخصه، انتساب ماندگار، و و عمل می‌شود. دلیل این مطلب، عدم حذف کلاس‌ها و امکان پشتیبانی از موارد نام‌برده از طریق آن‌ها می‌باشد.

مدلی که در قسمت ۴—۷ برای یک سلسله‌مراتب تکاملی نشان داده شد، بخشی دیگری از این طبقه‌بندی را پشتیبانی می‌کند. این بخش شامل کلاس‌های مجرد و واقعی و وراثت تک‌گانه در پیاده‌سازی قابل تطبیق سلسله‌مراتب‌ها، موارد ۲ و ۳ از انواع مشخصه، مورد ۲ از انواع انتساب ماندگار، و مورد ۲ از انواع عمل می‌شود. روش معمول پیاده‌سازی سلسله‌مراتب در یک سیستم شیءگرا، امکان فاکتورگیری از مشخصه‌ها و عملیات مشترک در چند سلسله‌مراتب (موارد ۴ و ۵ از انواع مشخصه و عمل) را از بین می‌برد. البته در حالت کلی نیازی به این فاکتورگیری نیست. در مدل قابل تطبیق ارائه شده در قسمت ۴—۷، باتوجه‌به سادگی تعریف مشخصه‌ها و امکان استفادهٔ مجدد مشخصه‌ها از طریق وراثت، نیازی به فاکتورگیری از مشخصه‌ها در بین چند سلسله‌مراتب نیست. اقدام به این کار سیستم را پیچیده‌تر خواهد کرد. ضمن این‌که معمولاً مشخصه‌های چندانی به‌صورت مشترک بین سلسله‌مراتب‌ها وجود ندارد. بنابراین، این پایان‌نامه تلاشی در جهت پوشش دادن مشخصه‌های مشترک بین سلسله‌مراتب‌ها (موارد ۴ و ۵ انواع مشخصه) نمی‌کند. اما وضعیت در مورد عملیات متفاوت است. برخی عملیات بین تمام سلسله‌مراتب‌های سیستم مشترک هستند. به‌عنوان مثال عمل create برای موجودیت‌های هر سلسله‌مراتبی وجود دارد. پوشش این اشتراک عملیات نیازمند این است که استفادهٔ مجدد عملیات، در سطحی فراتر از یک سلسله‌مراتب انجام شود.

پوشش انواع مختلف انتساب نیز یک ضرورت است، هرچند برخی از این انواع را با برنامه‌نویسی نیز می‌توان اعمال کرد. در قسمت ۵—۲ یک مدل عمومی برای پوشش حالات مختلف عمل و انتساب ارائه می‌گردد.

۵—۲ راه‌حلی عمومی برای تطبیق ساختاری و رفتاری

همان طور که قبلاً در تحلیل اهداف یک موتور تطبیق گفته شد، نمی‌توان روشی پیدا کرد که به‌طور هم‌زمان قدرت و سادگی نامحدود داشته باشد. همچنین، گفته شد که در این سیستم‌ها کمتر شدن قدرت در مقابل بیشتر شدن سادگی قابل قبول است. بااین‌حال، باید تلاش کرد قدرت مدل‌سازی، تا آن‌جا که به سادگی لطمه وارد نشود، حداکثر گردد.

گفته شد که تاکتیک داده به‌جای برنامه، به تنهایی پاسخ‌گوی نیاز تطبیق رفتاری نیست. در این قسمت نشان داده می‌شود که چگونه می‌توان از ترکیب سه تاکتیک داده به‌جای برنامه، عمومی کردن برنامه‌ها از طریق پارامتری کردن، و ترکیب عملیات در زمان اجرا برای ساختن عملیات پیچیده‌تر، به تطبیق رفتاری دست پیدا کرد. راه‌حلی که در این قسمت ارائه می‌شود نه تنها حالت عمومی راه‌حلی است که در قسمت ۴—۷—۳ برای عمل create مورد استفاده قرار گرفت، بلکه اعمال جدیدی که ممکن است در آینده تعریف شوند را پوشش می‌دهد.

راه‌حل پیشنهادی شامل سه ایدهٔ اصلی زیر است:

۱) ادغام سلسله‌مراتب‌ها برای بهبود استفاده مجدد و افزایش قدرت مدل‌سازی

۲) عملیات عمومی که برای موجودیت‌های آینده نیز قابل استفاده هستند

۳) ترکیب عملیات ابتدایی عمومی برای دستیابی به عملیات پیچیده‌تر

۵—۲—۱ ادغام سلسله‌مراتب‌ها

راه‌حلی که در ابتدا برای تبدیل سلسله‌مراتب‌های شکل ۵—۱ به یک مدل قابل تطبیق، به ذهن می‌رسد، عبارت است از تبدیل هریک از سلسله‌مراتب‌های تکاملی به یک مدل قابل تطبیق و حفظ سلسله‌مراتب‌های غیرتکاملی به‌صورت فعلی. شکل ۵—۲ مدل حاصل از این روش تبدیل را نشان می‌دهد. در این شکل، تنها کلاس‌هایی از سلسله‌مراتب‌های غیرتکاملی Entity-j و Entity-n نشان داده شده است که با موجودیت‌های سلسله‌مراتب‌های تکاملی Entity-1 و Entity-i انتساب دارند، ولی فرض بر این است که سلسله‌مراتب‌های غیرتکاملی به‌طور کامل در این مدل وجود دارند. در این مدل Association-1، Association-2، Association-3، Association-4، بدون اشکال، موجود هستند. Association-5 با استفاده از مدل قابل تطبیق سلسله‌مراتب Entity-1، به‌صورت متاداده مدل می‌شود. در کلاس‌های Entity-1 و Entity-i اعمال مربوط به کل سلسله‌مراتب‌های این دو کلاس دیده می‌شود. بنابراین، در این مدل برای هر نوع عمل از یک سلسله‌مراتب تکاملی یک نمونه پیاده‌سازی می‌شود. توضیح کاملتر این مطلب، در قسمت ۵—۲—۲ ارائه می‌گردد.

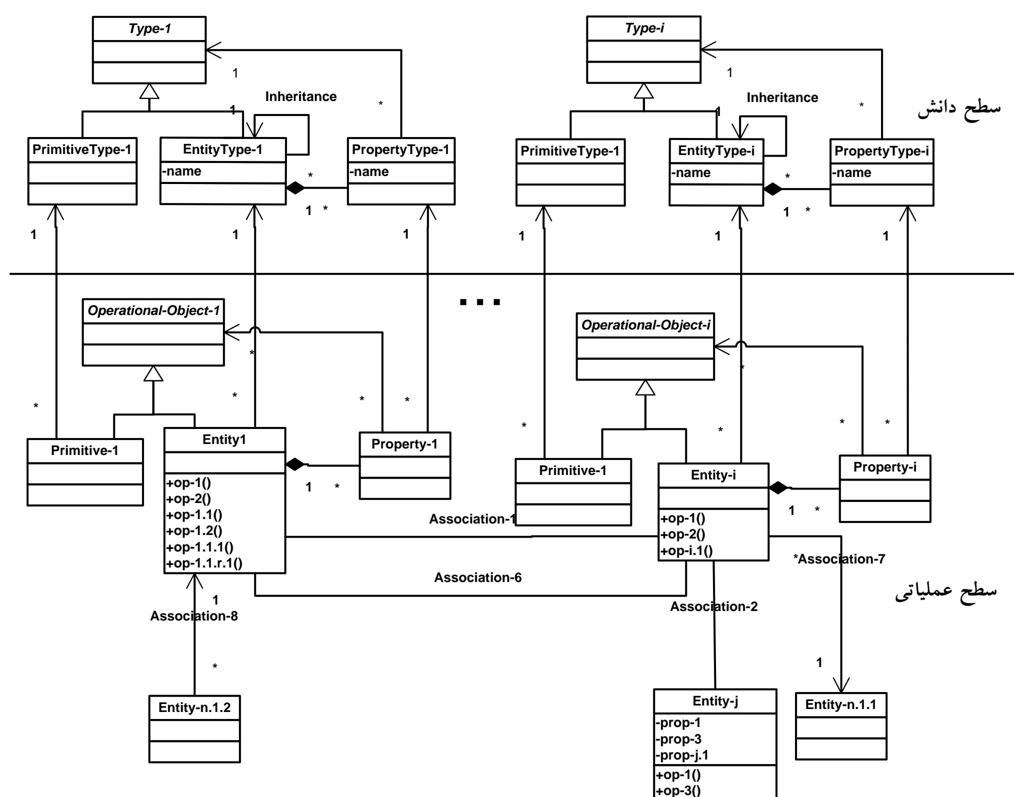
این مدل با مشکلات زیر مواجه است:

۱) عدم استفادهٔ مجدد در پیاده‌سازی مدل قابل تطبیق سلسله‌مراتب‌های تکاملی و در نتیجه نیاز به پیاده‌سازی‌های تکراری در مورد مشخصه‌ها، انتساب‌ها، وراثت و سایر امکاناتی که در مدل قابل تطبیق یک سلسله‌مراتب وجود دارد

۲) اشکال در مدل‌سازی انتساب Association-6 (انتساب ماندگار نوع ۳): این انتساب بین دو زیرکلاس از Entity-1 و Entity-i تعریف شده بود، ولی اکنون، بین کل موجودیت‌های این دو سلسله‌مراتب قرار دارد.

۳) اشکال در مدل‌سازی انتساب‌های Association-7 (انتساب ماندگار نوع ۴): با این مدل تمام موجودیت‌های سلسله‌مراتب Entity-1 می‌توانند با نمونه‌های Entity-n.1.1 رابطه داشته باشند.

۴) اشکال در مدل‌سازی انتساب‌های Association-8 (انتساب ماندگار نوع ۵): با این مدل نمونه‌های Entity-n.1.2 می‌توانند با تمام موجودیت‌های سلسله‌مراتب Entity-1 رابطه داشته باشند. اما باتوجه‌به



شکل ۵–۲: مدل قابل تطبیق برای شکل ۵–۱، با تبدیل مستقل سلسله‌مراتب‌های تکاملی

پیاده‌سازی Entity-n.1.1 از طریق برنامه‌نویسی، می‌توان این مشکل را با بررسی نوع زیرموجودیت Entity-1 در متن برنامه حل کرد.

۵) برخورد دوگانه با انتساب نوع ۲ و بقیهٔ انواع انتساب: به‌عبارت‌دیگر در این مدل یک انتساب نوع ۲ به‌صورت متاداده مدل می‌شود، وسایر انتساب‌ها در متن برنامه حک می‌شوند. این روش مشکلاتی مشابه آن‌چه در مورد برخورد دوگانه با مشخصه‌ها، در قسمت ۴—۷ گفته شد، دارد.

٦) عدم استفادهٔ مجدد در عملیات مشترک بین سلسله‌مراتب‌های تکاملی، مانند op-2 (عمل نوع ۴)

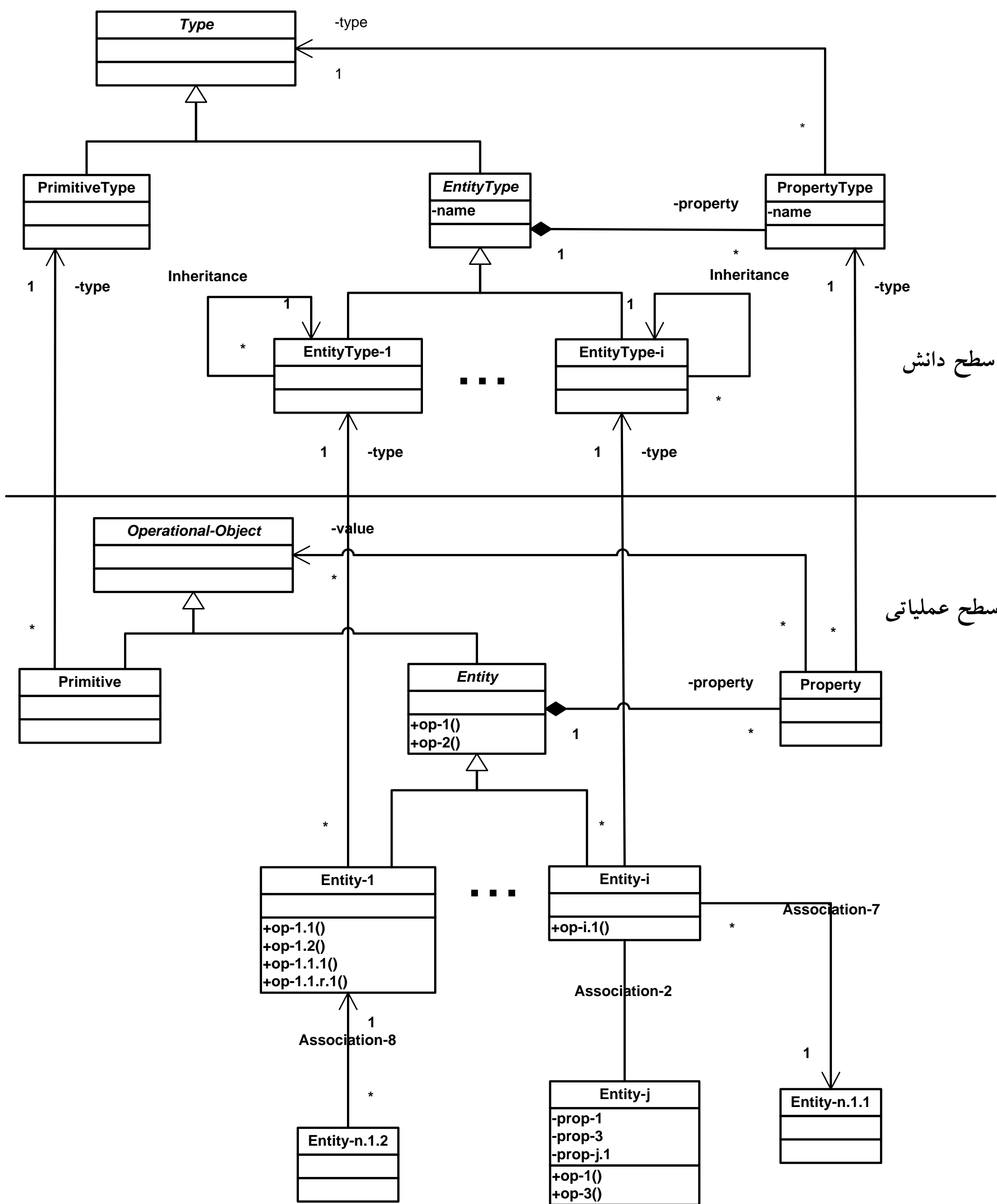
۷) عدم استفادهٔ مجدد در عملیات مشترک بین سلسله‌مراتب‌های تکاملی و غیرتکاملی، مانند op-1 (عمل نوع ۵)

۸) اعمال op-1.1.1 و op-1.1.r.1 که در کلاس Entity-1 قرار داده شده‌اند، تنها برای برخی زیرموجودیت‌های Entity-1 با معنی هستند، اما برای همه تعریف شده‌اند. این مطلب علاوه بر مشکل کردن درک مدل برای تولیدکنندگان و تغییردهندگان سیستم، امکان اجرای این اعمال بر روی موجودیت‌های نادرست را به‌وجود می آورد.

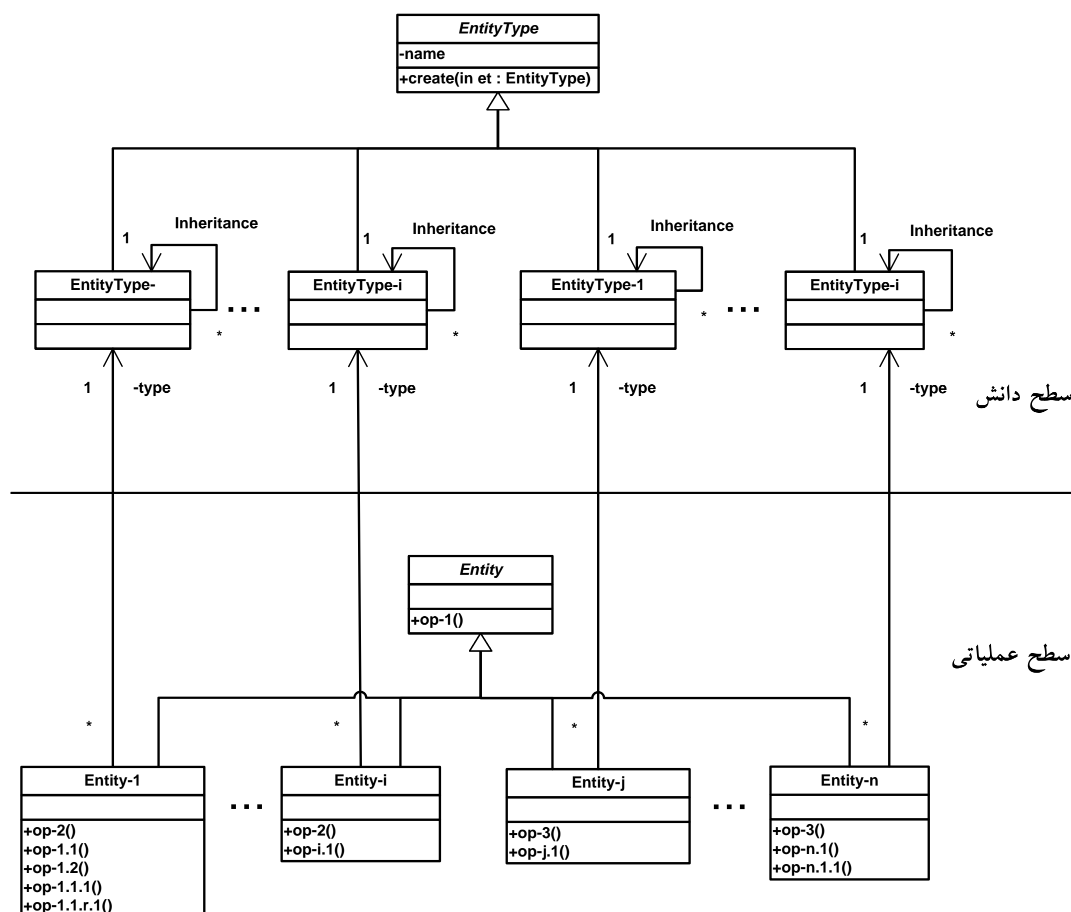
راه‌حل برخی از مشکلات گفته شده این است که مدل‌های مربوط به سلسله‌مراتب‌های تکاملی ادغام شوند. شکل ۵—۳ این راه‌حل را نشان می‌دهد. در این شکل کلاس Entity به‌عنوان پدر سلسله‌مراتب‌های تکاملی و کلاس EntityType به‌عنوان پدر کلاس‌های نوع این سلسله‌مراتب‌ها اضافه شده‌اند و از این طریق از مشترکات مدل‌های قابل تطبیق شکل ۵—۲ فاکتورگیری شده است. در این مدل، تنها یک مربع نوع بین کلاس‌های EntityType، Entity، Property، و PropertyType وجود دارد، بنابراین، مورد ۱ از مشکلات بالا حل شده است. همچنین، با این مدل، انتساب Association-6 به‌دقت به‌صورت متاداده قابل تعریف است و در نتیجه مشکل ۲ نیز حل می‌شود. انتساب Association-1 و Association-6 را می‌توان به‌صورت متاداده تعریف کرد، بنابراین، بخشی از مشکل ۵ حل شده است. عمل op-2 نیز در این مدل یک‌بار پیاده‌سازی می‌شود، در نتیجه، مشکل ٦ نیز حل شده است. در سلسله‌مراتب‌های تکاملی از عمل op-1 فاکتورگیری شده است، اما سلسله‌مراتب‌های غیرتکاملی این عمل را جداگانه پیاده‌سازی می‌کنند. در مجموع، مشکلات ۳، ۴، ۸ و قسمتی از مشکلات ۵ و ۷ هنوز باقی هستند.

یک راه‌حل برای رفع مشکلات گفته شده، این است که سلسله‌مراتب‌های غیرتکاملی نیز به‌صورت تکاملی پیاده‌سازی شوند. به این مدل یک مدل کاملاً قابل تطبیق می‌گوییم. شکل ۵—۴ بخشی از یک مدل کاملاً قابل تطبیق را برای مدل شکل ۵—۱ نشان می‌دهد. در این مدل کلاس سلسله‌مراتب‌های غیرتکاملی نیز زیر کلاس‌های Entity هستند. در این مدل تمام انتساب‌ها به‌دقت با متاداده قابل تعریف هستند بنابراین، مشکلات ۳ و ۴ و ۵ حل می‌شوند. همچنین، عمل op-1 که بین سلسله‌مراتب‌ها مشترک است، در کلاس Entity به‌صورت عمومی پیاده‌سازی می‌شود، بنابراین، مشکل ۷ نیز حل می‌شود. اما کلاس op-2 دیگر بین تمام زیرکلاس‌های Entity مشترک نیست، بنابراین، نمی‌توان آن‌را در کلاس Entity فاکتورگیری کرد. در نتیجه، مشکل ٦ دوباره به مدل برگشته است. همچنین، همین مشکل برای op-3 نیز قابل طرح است. مشکل ۸ و مشکل مشابه آن برای op-n.1.1 در مدل وجود دارند. راه‌حل این مشکلات در قسمت ۵—۲—۲ ارائه می‌شود.

این راه‌حل به‌دلیل کاهش کارایی، در تمام سیستم‌ها قابل استفاده نخواهد بود. دراین‌صورت، می‌توان از مدل شکل ۵—۳ استفاده کرد و مشکلات آن مدل را از طریق بررسی‌های اضافه در برنامه‌نویسی حل کرد.



شکل ۵—۳: ادغام مدل قابل تطبیق سلسله‌مراتب‌های تکاملی



شکل ۵–۴: یک مدل کاملاً قابل تطبیق

یک راه‌حل دیگر که ممکن است مناسب باشد، استفاده از انعکاس زبان‌های برنامه‌نویسی برای دست‌یابی به یک راه‌حل بین دو راه‌حل ارائه شده است. به این مطلب در کارهای آینده (فصل ۷) اشاره می‌گردد.

در ایدهٔ ادغام سلسله‌مراتب‌ها از تاکتیک «داده به‌جای برنامه» استفاده شده است. البته این ایده به‌طور کامل از دست‌آوردهای این پایان‌نامه نیست و پیش از این در مدل‌های ارائه شده در [۱۱، ۲۹] نیز نشان داده شده است.

۵–۲–۲ عملیات عمومی

در شکل‌های ۵–۲، ۵–۳، و ۵–۴، از هر نوع عمل موجود در یک سلسله‌مراتب، تنها یک نمونه در مدل قابل تطبیق سلسله‌مراتب‌های تکاملی قرار داده شد. در بعضی موارد این فاکتورگیری به چند سلسله‌مراتب نیز تعمیم داده شد. به عبارت دیگر، تک نمونهٔ این اعمال باید طوری پیاده‌سازی شود که تمام نمونه‌های عمل را پوشش دهد. علاوه بر این، اعمال باید به‌قدری عمومی باشند که بتوانند موجودیت‌های جدید را نیز پشتیبانی کنند. همان‌طورکه در قسمت ۴–۷ در مورد عمل create از مثال فروشگاه گفته شد، درحالت کلی در پیاده‌سازی چنین اعمالی نمی‌توان فرض را بر وجود یک سری مشخصهٔ خاص در موجودیت گذاشت،

اگرچه در بعضی موارد مثل عمل sell این فرض صحیح است. راه‌حل پیشنهادی برای پیاده‌سازی این گونه اعمال، این است که برخی اطلاعات سطح دانش، مثل انواع مشخصهٔ یک موجودیت، در اختیار این اعمال قرار گیرد. با داشتن این اطلاعات می‌توان به اعمال عمومیت بخشید. همان‌طور،که قبلاً گفته شد این روش نوعی انعکاس به‌حساب می‌آید. شکل ؟؟ این روش را برای عمل create نشان داد. پیاده‌سازی انعکاسی اعمال هیچ محدودیتی ندارد و هر عملی را می‌تواند پشتیبانی کند. اما لازمهٔ این روش این است که در هنگام مهندسی دامنه، تمام حالات ممکن برای نمونه‌های یک نوع عمل پیش‌بینی شود. درغیراین‌صورت، با پیدایش نیاز به یک نمونهٔ جدید از عمل، باید متن برنامهٔ عمل عمومی تغییر داده شود. این روش کاربردی از تاکتیک عمومی‌کردن برنامه‌ها از طریق پارامتری کردن می‌باشد.

در ادامه در مورد مشکلات باقی‌مانده از قسمت بحث می‌شود. این مشکلات عبارتند از:

۱) عدم امکان فاکتورگیری از برخی اعمال مثل op-2

۲) قرار گرفتن برخی اعمال مثل op-1.1.1 در جای نادرست

یک راه‌حل برای این مشکلات این است که چنین اعمالی، هر یک، در کلاسی جداگانه تعریف شوند. دراین‌صورت، برای اجرای این اعمال، اشیاء مناسب (و در مواردی اشیاء نوع) به‌عنوان پارامتر به عمل داده می‌شوند و عمل بر روی اشیاء اجرا می‌شود (تکنیک نمایندگی). حال این سوال مطرح می‌شود که چگونه می‌توان کنترل کرد که اعمالی که به این شکل تعریف می‌شوند، تنها روی اشیاء مناسبی اجرا گردند. از آنجا که این اعمال در حالت کلی مختص یک نوع موجودیت نیستند، نباید در متن برنامهٔ آن‌ها نام یک نوع موجودیت را حک[۵] کرد. دو روش زیر برای جلوگیری از اجرای اعمال روی موجودیت‌های نامربوط قابل تصور می‌باشد:

۱) بررسی توسط خود عمل: در این روش عمل از طریق بررسی خصوصیات نوع موجودیت ارسال شده، از اجرای خود بر روی موجودیت‌های نامربوط جلوگیری می‌کند. به‌عنوان مثال، این روش را این طور می‌توان پیاده‌سازی کرد که عمل عمومی پیش از اجرا وجود یک سری مشخصه را در موجودیت ارسال شده بررسی کند و در صورت عدم وجود این مشخصه‌ها، از اجرای عمل اجتناب کند. ممکن است این روش نتواند از بعضی اجراهای نامربوط جلوگیری کند. به‌عنوان مثال، فرض کنید موجودیت نامربوطی نیز دارای مشخصه‌های موردنظر عمل می‌باشد و آن‌ها را به‌منظور دیگری استفاده می‌کند؛ دراین‌صورت، عمل در جلوگیری از اجرای نامربوط ناموفق خواهد بود. این روش با برنامه‌نویسی تدافعی[۶] [۶۰] متناظر است. در برنامه‌نویسی تدافعی یک برنامه هیچ فرضی در مورد پارامترهای ورودی ارسالی توسط سرویس‌گیرندگان نمی‌کند، و وظیفهٔ بررسی صحت آن‌ها را خود به‌عهده می‌گیرد. این روش ممکن است به کارایی لطمه وارد کند.

۲) انتساب اعمال به موجودیت‌ها توسط متخصص دامنه: در این روش سیستم امکان انتساب اعمال به موجودیت‌ها را به متخصص دامنه می‌دهد. این روش نیز تضمین نمی‌کند که خطاهای زمان اجرا رخ ندهد زیرا ممکن است متخصص دامنه در انتساب دچار اشتباه شود. این روش با طراحی با قرارداد[۷] متناظر است [۶۱]. در برنامه‌نویسی مبتنی بر قرارداد ورودی‌ها و خروجی‌های یک عمل، در قالب یک قرارداد بیان می‌شوند و وظیفهٔ استفادهٔ درست از آن به‌عهدهٔ سرویس‌گیرندگان است.

---

[۵] Hardcode

[۶] Defensive Programming

[۷] Design by Contract

می‌توان از ترکیب این دو روش نیز برای به حداقل رساندن خطاهای زمان اجرا استفاده کرد. البته، دراین‌صورت نیز تضمینی برای عدم اجراهای نامربوط اعمال وجود ندارد، اما این درصد خطا پذیرفتنی است، و می‌توان آن‌را هزینهٔ پذیرش تطبیق توسط متخصص دامنه قلمداد کرد. روش بررسی توسط خود عمل را باید برنامه‌نویس در هنگام ایجاد برنامهٔ عمل، پیاده‌سازی کند و احتیاج به طراحی خاصی ندارد. اما روش انتساب اعمال به موجودیت‌ها توسط متخصص دامنه تاحدی نیاز به طراحی دارد، که در ادامه روشی برای آن عرضه می‌گردد.

نکتهٔ دیگری که در مورد اجرای اعمال عمومی باید در نظر گرفته شود این است که حتی با فرض متناسب بودن عمل و نوع موجودیت، به‌دلایل زیر نمی‌توان از درستی یک اجرای خاص آن اطمینان حاصل کرد:

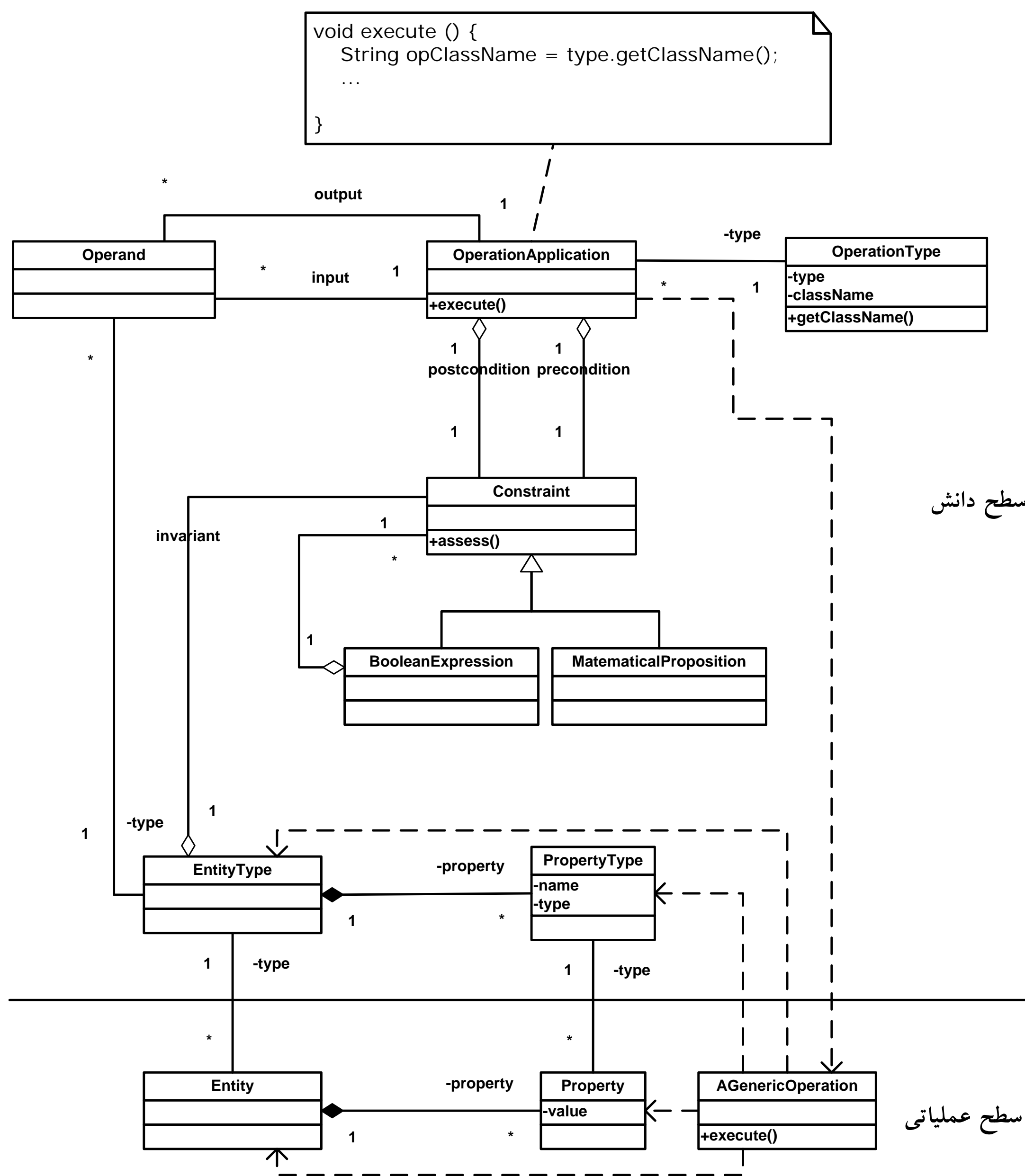
۱) وجود پیش‌شرط [۸] برای اجرای عمل بر روی موجودیت: ممکن است بخشی از این پیش شرط مربوط به عمل باشد نه به یک موجودیت خاص. به‌عنوان مثال، ممکن است عمل نسبت به مقادیر مشخصه‌های موجودیت ورودی انتظار خاصی داشته باشد، که دراین‌صورت می‌توان پیش‌شرط را توسط روش بررسی توسط خود عمل پیاده‌سازی کرد. درغیراین‌صورت، پیش‌شرط مربوط به خود موجودیت خواهد بود. به‌عبارت‌دیگر، از نظر منطق حرفه اجرای عمل بر روی یک نوع موجودیت در شرایط خاصی نادرست می‌باشد. روش پیاده‌سازی این نوع پیش‌شرط به‌زودی توضیح داده می‌شود.

۲) وجود پس‌شرط [۹] برای اجرای عمل بر روی موجودیت: انواع پس‌شرط و روش پیاده‌سازی آن‌ها شبیه پیش‌شرط می‌باشد.

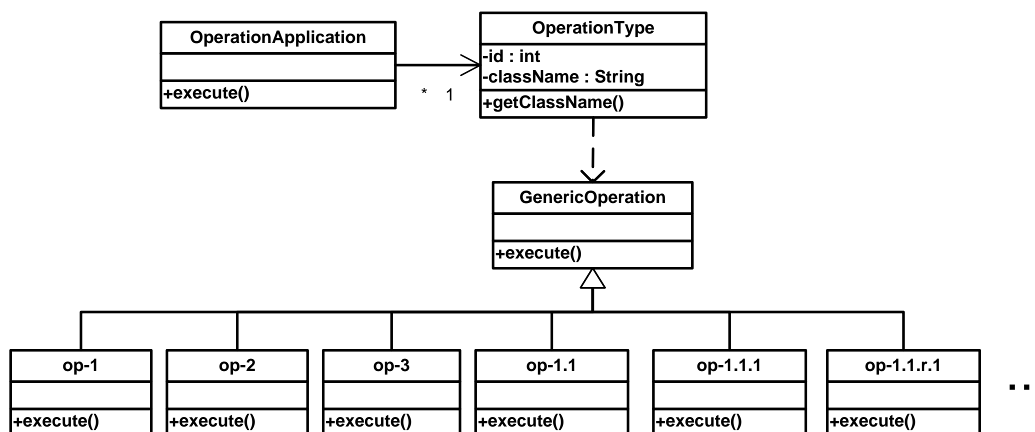
۳) وجود ثابت [۱۰] در مورد یک موجودیت: ثابت شرطی است که همواره باید در مورد یک موجودیت برقرار باشد و به یک عمل خاص مربوط نمی‌شود.

به این شرط‌ها محدودیت می‌گوییم. یک عمل ممکن است بر روی بیش از یک موجودیت اجرا شود. دراین‌صورت، پیش‌شرط‌ها و پس‌شرط‌های ممکن است ناظر به تمام این موجودیت‌ها تعریف شوند. در اجرای عمل نیز ثابت‌های تمام موجودیت‌ها باید برقرار باقی بمانند. شکل ۵–۵ راه‌حلی برای مسائل بالا ارائه می‌دهد. در این راه‌حل کلاس AGenericOperation یک عمل عمومی را پیاده‌سازی می‌کند. اشیاء کلاس OperationApplication کاربردی از این عمل را برای مجموعه از موجودیت‌ها تعریف می‌کنند. نوع عمل در اشیاء کلاس OperationType تعریف شده است و انتساب بین این اشیاء و اشیاء کلاس OperationApplication کاربردی از الگوی شیء نوع می‌باشد. انواع موجودیت‌های ورودی و خروجی برای این کاربرد از طریق انتساب‌های input و output مشخص می‌شود. پیش‌شرط‌ها، پس‌شرط‌ها، و ثابت‌ها توسط کلاس Constraint و بچه‌های آن پیاده‌سازی شده و با انتساب‌های precondition، postcondition، و invariant به اشیاء OperationApplication و EntityType مرتبط می‌گردند. این انتساب‌ها حالت اشتراکی دارند، بنابراین، محدودیت‌های تعریف شده با استفاده از Constraint قابل استفادهٔ مجدد هستند. یک شیء از کلاس MatematicalProposition، یک عبارت ریاضی را در خود نگه‌می‌دارد که نتیجهٔ بولی دارد، مانند $entity.x > entity.y$. یک شیء از کلاس BooleanExpression یک عبارت بولی را در خود نگه‌می‌دارند که ترکیبی بولی از اشیاء MatematicalPredicate می‌باشد. محدودیت‌ها را از راه‌های مختلفی می‌توان

[۸] Precondition
[۹] Postcondition
[۱۰] Invariant



شکل ۵–۵: انتساب اعمال و موجودیت‌ها



شکل ۵—٦: جداسازی کلّی اعمال و موجودیت‌ها

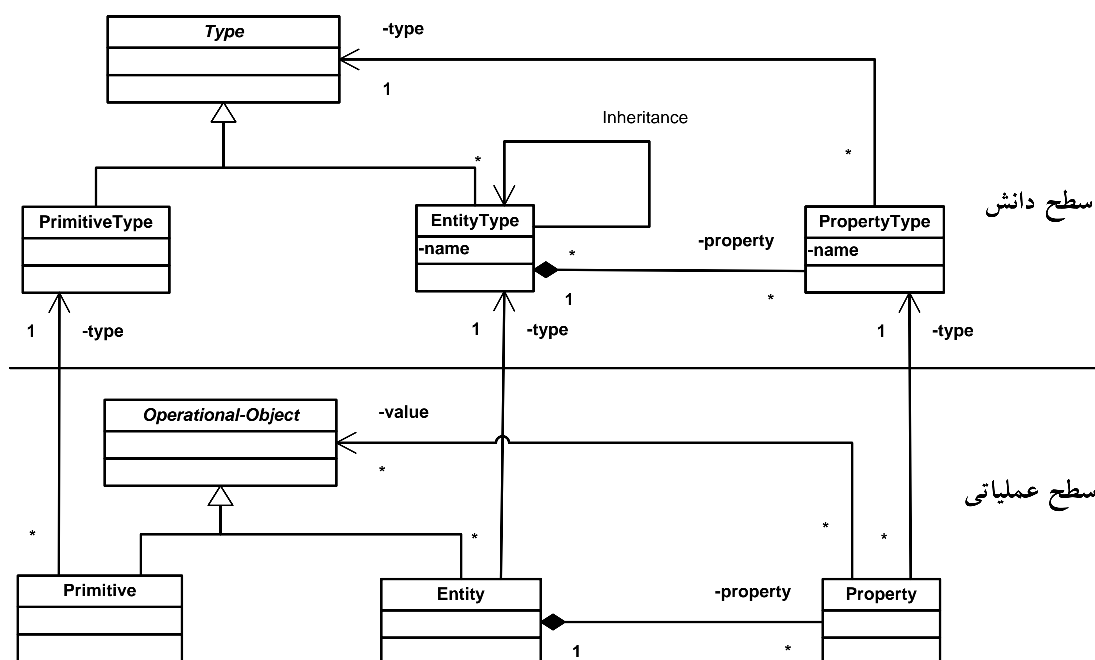
مدل‌سازی کرد (به‌عنوان مثال با استفاده از منطق‌های ریاضی مختلف) که شکل ۵—۵ تنها یک نمونه از این راه‌ها را نشان می‌دهد. در این مدل در انتساب اعمال به انواع موجودیت ارث‌بری وجود دارد.

یک نکتهٔ قابل توجه در مورد این مدل نحوهٔ فراخوانی عمل عمومی می‌باشد. علاوه بر نوع عمل، نام کلاسی که یک عمل عمومی را پیاده‌سازی کرده است نیز در اشیاء کلاس OperationType نگه‌داری می‌شود. بنابراین، می‌توان با استفاده از انعکاس زبان‌های برنامه‌نویسی با داشتن نام کلاس، عمل execute آن را فراخوانی کرد. نحوهٔ ارسال پارامترها به اعمال عمومی در فصل ٦ توضیح داده می‌شود. این روش پیاده‌سازی کمک می‌کند که اعمال جدیدی که با برنامه‌نویسی ایجاد می‌شوند، بدون تغییر سایر برنامه‌ها و تنها با تعریف یک شیء جدید از کلاس OperationType به سیستم اضافه شوند. این خاصیت که مصداقی از قابلیت اتصال مؤلفه‌های جدید به سیستم می‌باشد را قابلیت اتصال اعمال جدید به سیستم[11] می‌نامیم.

برای یکنواختی و استفادهٔ مجدد از این مدل، می‌توان تمام اعمال عمومی را حتی در صورت موجود بودن کلاس آن‌ها در مدل، از کلاس‌های انواع موجودیت و موجودیت استخراج کرد و در کلاس‌های جداگانه قرار داد. شکل ۵—٦ این مطلب را نشان می‌دهد. به‌ازای هر یک از اعمال تعریف شده در زیرکلاس‌های GenericOperation یک شیء از کلاس OperationType نیز در سیستم وجود دارد. با استفاده از اشیاء کلاس OperationApplication این عملیات به انواع موجودیت‌های مناسب منتسب می‌شوند. حال که تمام عملیات و مشخصه‌های موجودیت‌های مختلف از آن‌ها جدا شده است، می‌توان کلاس رأس سلسله‌مراتب‌ها را نیز حذف کرد و تمام موجودیت‌ها را با استفاده از کلاس Entity پیاده‌سازی کرد. دراین‌صورت، کلاس‌های نوع موجودیت‌های مختلف نیز حذف می‌شوند و تمام انواع موجودیت تنها با اشیاء کلاس EntityType تعریف می‌گردند. شکل ۵—۷ این مدل را نشان می‌دهد. این مدل بسیار شبیه مدل شکل ۴—۱۰ می‌باشد که برای یک سلسله‌مراتب استفاده شد. در حقیقت در این مدل بسیاری از اطلاعات ساختار دامنه حذف شده است و به متاداده منتقل شده است.این مدل هر دو مشکل رفتاری مطرح شده در ابتدای این قسمت را حل می‌کند. بااین‌حال، هنوز موتور تطبیق نمی‌تواند از عملیاتی که شکل عمومی آن‌ها در سیستم تعریف نشده، پشتیبانی کند. در قسمت ۵—۲—۳ راه‌حلی برای این مشکل ارائه می‌گردد.

تغییر دیگری که نسبت به مدل‌های قبلی در این مدل دیده می‌شود، پشتیبانی از وراثت چندگانه است. باتوجه‌به این‌که موجودیت‌های این مدل، خود هیچ عملی را پیاده‌سازی نمی‌کنند، این موجودیت‌ها شبیه

[11] Operation Plugability



شکل ۵–۷: مدل کاملاً قابل تطبیق کاهش‌یافته

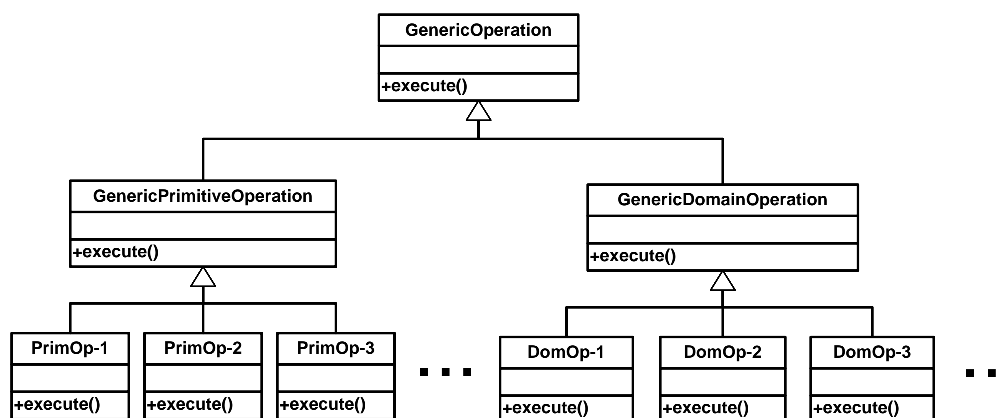
واسط‌ها در زبان‌های برنامه‌نویسی شیءگرا هستند. در این زبان‌ها، واسط‌ها ابهامات وراثت چندگانه را حل می‌کنند. نقش وراثت در این مدل، ارث‌بری مشخصه‌ها، انتساب‌ها، و کاربردهای اعمال عمومی است. باتوجه‌به این‌که هر کاربرد عمل عمومی، مخصوص به یک سری موجودیت خاص تعریف می‌شود، برخوردی بین اعمال پدرهای مختلف یک موجودیت به‌وجود نمی‌آید. در بدترین حالت، یک موجودیت ممکن است با چند نقش متفاوت در یک عمل ظاهر شود.

در ایدهٔ عملیات عمومی از دو تاکتیک «داده به‌جای برنامه»، «عمومی کردن برنامه‌ها از طریق پارامتری کردن» استفاده شده است.

۵–۲–۳ ترکیب عملیات ابتدایی

مدلی که تابه‌حال ارائه گردیده است، باوجود قابلیت‌های زیاد مانند امکان تعریف انواع جدید و به‌کارگیری عملیات موجود بر روی این انواع، نقایصی دارد. مهم‌ترین نقص این مدل در زمینهٔ تطبیق رفتاری این است که تنها عملیاتی را پشتیبانی می‌کند که شکل عمومی آن‌ها در سیستم وجود دارد. بنابراین، اگر یک نوع موجودیت، عملی را نیاز داشته باشد که حالت کلی آن در سیستم موجود نباشد، باید آن‌را از طریق برنامه‌نویسی ایجاد و به سیستم اضافه کرد. ازآن‌جاکه نمی‌توان در زمان مهندسی دامنه تمام عملیات ممکن سیستم را پیش‌بینی کرد، قدرت تطبیق رفتاری سیستم محدود می‌شود. برای مثال، در قسمت ۴–۷ دو سناریوی پس دادن محصول و جایزه مطرح شدند که بدون برنامه‌نویسی توسط سیستم فروشگاه قابل پیاده‌سازی نبودند.

البته در مدل ارائه شده عملیات عمومی ایجاد شده توسط برنامه‌نویسی را می‌توان بدون توقف سیستم و تغییر سایر قسمت‌ها به سیستم اضافه کرد. اما بازهم ممکن است نیاز به برنامه‌نویسی باعث کندی کار شود.



شکل ۵—۸: جداسازی عملیات ابتدایی و عملیات دامنه

در نتیجه، بهتر است سیستم امکاناتی داشته باشد که متخصص دامنه بتواند عملیاتی را که به‌صورت فوری نیازمند آن‌ها می‌باشد، تعریف کند.

راه‌حلی که می‌توان برای رفع این نقص به‌کار گرفت، استفاده از تاکتیک ترکیب عملیات در زمان اجرا برای ساختن عملیات پیچیده‌تر، می‌باشد. روش پیشنهادی عبارت است از قرار دادن یک سری عملیات ابتدایی عمومی[۱۲] در سیستم است که بتوان از ترکیب پویای آن‌ها اعمال جدید را تولید کرد. حال، این سئوال مطرح است که این عملیات را چگونه باید انتخاب کرد. اگر این عملیات به‌ریزدانگی ساختارهای یک زبان برنامه‌نویسی باشند، قدرت زبان مدل‌سازی زیاد می‌شود، اما پیچیدگی زبان نیز بیش از توان متخصصین دامنه خواهد بود. علت این مطلب این است که ساختارهای زبان برنامه‌نویسی در سطح تجرید بسیار بالایی قرار دارند و برای این کاربرد، بیش از حد عمومی هستند. ازسوی‌دیگر، اگر این عملیات به‌درشت‌دانگی عملیات عمومی مربوط به یک موجودیت خاص باشند، کاربرد خاص و محدودی خواهند داشت و در نتیجه نمی‌توان آن‌ها را با هم ترکیب کرد و عملیات جدیدی به‌دست آورد. بنابراین، می‌توان نتیجه گرفت که این عملیات باید مستقل از موجودیت[۱۳] (یا مشترک در بین موجودیت‌ها[۱۴]) باشند ولی درشت‌دانه‌تر از ساختارهای یک زبان برنامه‌نویسی باشند. برای نمونه، عمل create در مثال فروشگاه از قسمت ۴—۷ انتخاب خوبی به‌نظر می‌رسد زیرا نه‌تنها برای تمام موجودیت‌ها قابل استفاده است، در سطح تجرید پایین‌تری نسبت به ساختارهای یک زبان برنامه‌نویسی نیز قرار دارد. در بیشتر موارد، اعمال ابتدایی مستقل از دامنه[۱۵] نیز می‌باشند. عمل op-1 از شکل ۵—۱، که بین تمام سلسله‌مراتب‌ها مشترک است، نیز یک عمل ابتدایی می‌باشد.

اعمال ابتدایی را بر حسب نیاز به‌شکل‌های مختلفی می‌توان تعریف کرد. شکل ۵—۸ تقسیم اعمال به اعمال دامنه و یک مجموعهٔ پیشنهادی از اعمال ابتدایی را نشان می‌دهد. سرویس ارائه‌شده توسط اعمال Create، DisplayeOne، Update، Delete به‌ترتیب عبارت است از ایجاد، نمایش، به‌روزرسانی و حذف یک موجودیت دلخواه. Search برای جست‌وجوی یک یا چند موجودیت به‌کار می‌رود و DisplayList فهرستی از موجودیت‌ها را نمایش می‌دهد. DisplaySelectableList فهرستی از موجودیت‌ها را نشان می‌دهد و به کاربر امکان می‌دهد از بین آن‌ها تعدادی را برای ارسال به اعمال بعدی انتخاب کند. تمام اعمال ابتدایی نشان

[۱۲] Generic Primitive Operations
[۱۳] Entity-Inependent
[۱۴] Entity Cross-Cutting Operation
[۱۵] Domain-Inependent

داده شده، بین انواع مختلف موجودیت مشترک هستند. در برخی کاربردها ممکن است یک سری اعمال ابتدایی مورد نیاز باشد که برای تمام موجودیت‌ها قابل استفاده نیست. برای مثال عمل Email را می‌توان یک عمل ابتدایی فرض کرد که در عملیات پیچیده‌تری قابل به‌کارگیری است ولی تنها برای موجودیت‌هایی قابل استفاده است که مشخصهٔ email دارند. با سیستم‌های نمادگذاری مختلفی می‌توان عملیات ابتدایی را با هم ترکیب کرد.

انتساب هر یک از اعمال شکل ۵–۸ به انواع موجودیت مطابق روش ارائه شده در ۵–۲–۲ انجام می‌شود. علاوه بر انتساب عملیات ابتدایی به موجودیت‌ها باید روشی برای ترکیب عملیات نیز در اختیار متخصصین دامنه قرار داده شود. در این جا منظور از ترکیب، اجرای عملیات با ترتیبی مشخص است. این اجرا باید نتیجه‌ای معادل عمل پیچیدهٔ مورد نظر متخصص دامنه داشته باشد. ترکیب عملیات در سیستم را می‌توان به دو بخش تقسیم کرد:

۱) تعریف عملیات مرکب با استفاده از عملیات ابتدایی

۲) اجرای عملیات مرکب

اجرای عملیات مرکب در قسمت ٦–۳ بررسی می‌گردد. در ادامه مدلی ارائه می‌شود که امکان تعریف عملیات مرکب را فراهم کند.

منظور از ترکیب عملیات ابتدایی، تعریف ترتیبی از آن‌هاست که در صورت اجرا نتیجه‌ای معادل عمل مرکب مورد نظر خواهد داشت. بنابراین، ترکیب عملیات نیازمند موارد زیر است:

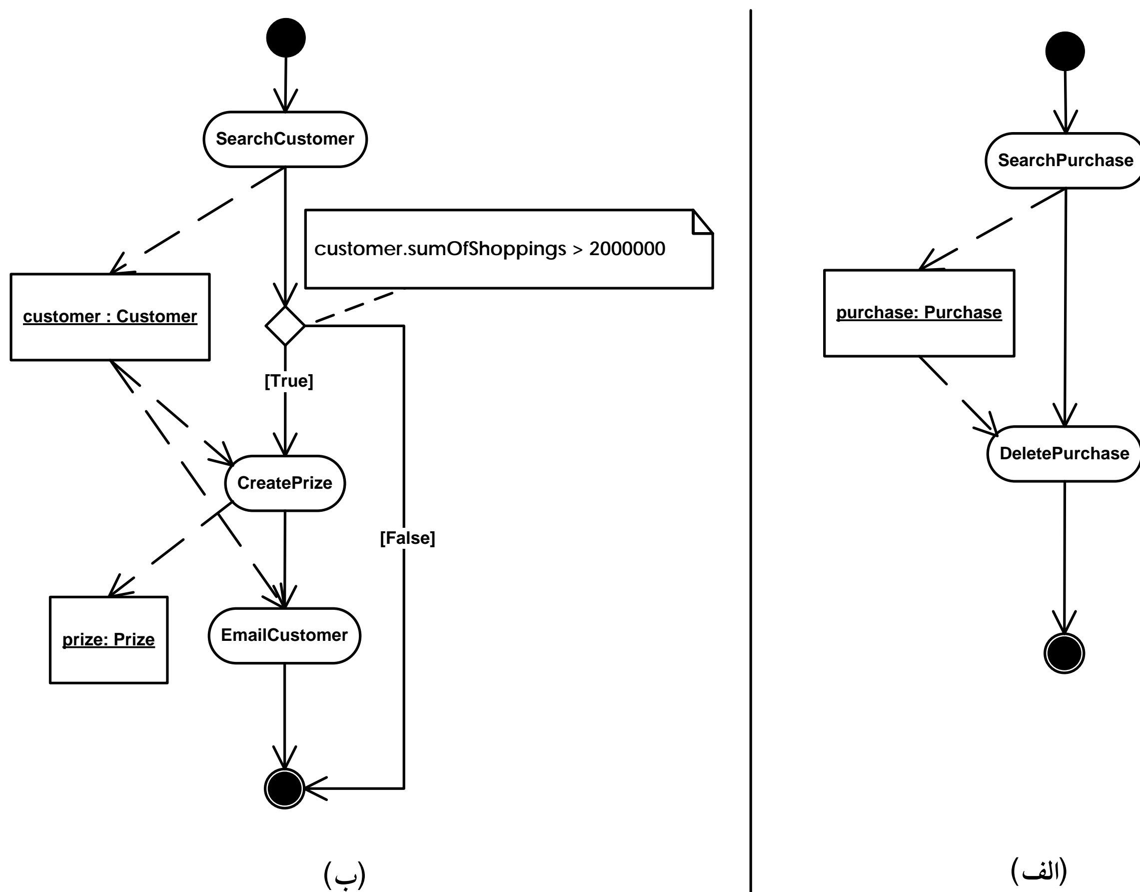
۱) یک سیستم نمادگذاری برای توصیف جریان کنترل بین عملیات ابتدایی: چنین سیستمی باید از ساختارهای متداول جریان کنترل مانند ترتیب[16]، انتخاب[17]، و تکرار[18] پشتیبانی کند.

۲) روشی برای انتقال خروجی‌های اعمال به اعمال بعدی

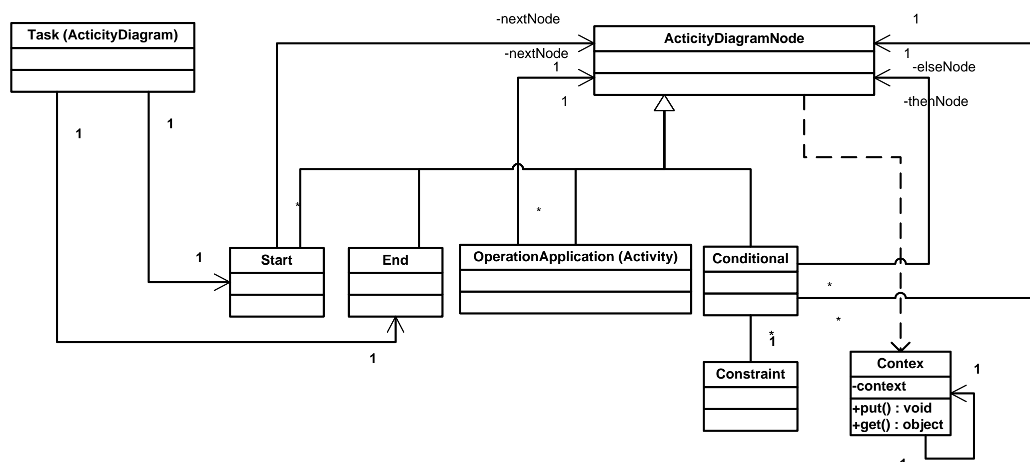
در روش ارائه شده در این پایان‌نامه برای توصیف جریان کنترل بین عملیات ابتدایی، از نمودار فعالیت UML استفاده می‌شود. البته هر نمودار دیگری که ساختارهای گفته شده را داشته باشد نیز برای این کار مناسب است. شکل ۵–۹ ترکیب عملیات ابتدایی برای ایجاد دو عمل بازگشت محصول و جایزه از مثال فروشگاه را نشان می‌دهد. در این دو مدل فرض براین است که انواع موجودیت Customer، Prize، و Purchase در سیستم تعریف شده‌اند.

مدل‌های نشان داده شده در سطح بالایی از تجرید قرار دارند. نمودار فعالیت تنها ترتیب اجرای عملیات را مشخص می‌کند. برای مناسب‌سازی[19] عملیات ابتدایی باید اطلاعات دیگری نیز دراختیار سیستم قرار داد. بخشی از این اطلاعات که مربوط به ورودی‌ها و خروجی‌های اعمال می‌باشد، توسط مدل شکل ۵–۵ قابل تعریف است. اما در یک موتور تطبیق معمولاً جزئیات بیشتری برای مناسب‌سازی اعمال لازم است. میزان این جزئیات بستگی به میزان عمومیت عمل در پشتیبانی از حالات مختلف ممکن دارد. برای مثال عمل Search را می‌توان طوری تعریف کرد که تنها توانایی جست‌وجو بر روی یک نوع موجودیت را

[16] Sequence
[17] Selection
[18] Iteration
[19] Customization



شکل ۵–۹: ترکیب عملیات ابتدایی برای ایجاد دو عمل الف) پس‌دادن محصول و ب) جایزه



شکل ۵—۱۰: مدلی برای تعریف عملیات پیچیده با ترکیب عملیات ابتدایی

داشته باشد یا هر نوع جست‌وجوی پیچیده‌ای که موجودیت‌های متعددی را شامل می‌شود، به‌وسیلهٔ آن ممکن باشد. در صورت دوم مناسب‌سازی عمل جست‌وجو به‌قدری پیچیده می‌شود که خود نیازمند زبانی به‌پیچیدگی یک زبان تعریف پرس‌وجو[۲۰] (مثل SQL) است. دراین‌صورت سادگی سیستم از بین می‌رود. بنابراین، در حالت کلی بهتر است از پیچیده‌کردن عملیات ابتدایی برای پوشش تمام حالات ممکن اجتناب شود. پذیرش این مطلب، به‌معنی کم شدن قدرت موتور تطبیق در ایجاد عملیات پیچیده از ترکیب عملیات ابتدایی است.

شکل ۵—۱۰ مدلی برای تعریف عملیات مرکب با استفاده از نمودار فعالیت ارائه می‌دهد. یک شیء از کلاس Task، یک عمل مرکب ناموجود در سیستم را تعریف می‌کند و یک نمودار فعالیت به آن نسبت می‌دهد. عملیات ابتدایی که با استفاده از OperationApplication به موجودیت‌ها منتسب می‌شوند، درحقیقت، فعالیت‌های این نمودار فعالیت هستند. همان‌طور که در شکل مشخص است شرط‌ها نیز با استفاده از Constraint قابل پیاده‌سازی هستند. برای انتقال موجودیت‌های ورودی و خروجی به اعمال از یک شیء از کلاس Context استفاده می‌شود.

نکته‌ای که در از در کنار هم قرار دادن نمودار شکل‌های ۵—۵، ۵—۸، و ۵—۱۰ به‌دست می‌آید این است که در مدل فعلی می‌توان اعمال دامنه را نیز در ترکیب عملیات شرکت داد. باتوجه‌به این‌که ممکن است در مواردی این ویژگی مفید باشد، نیازی به جلوگیری از آن وجود ندارد.

در ایدهٔ ترکیب عملیات از سه تاکتیک «داده به‌جای برنامه»، «عمومی کردن برنامه‌ها از طریق پارامتری کردن»، و «ترکیب عملیات در زمان اجرا برای ساختن عملیات پیچیده‌تر»

۵—۳ بحث

یک سئوال که قابل طرح است این است که در طراحی یک موتور تطبیق تا چه حد باید از این روش‌ها استفاده کرد. به‌طور مشخص:

[۲۰] Query

- تا چه حد باید سلسله‌مراتب‌ها را ادغام کرد و کاهش داد؟ آیا باید سلسله‌مراتب‌ها را کلاً حذف کرد و تمام انواع موجودیت‌ها را تنها با یک کلاس EntityType مدل کرد؟
- عملیات عمومی (ابتدایی و دامنه) تا چه حد کلی باشند و آینده را پیش‌بینی کنند؟ آیا باید یک عمل عمومی دامنه طوری نوشته شود که از حالاتی که در مهندسی دامنه دیده نمی‌شوند نیز پشتیبانی کند؟ آیا اعمال ابتدایی باید طوری نوشته شوند که از ترکیب آن‌ها هر نوع عملی را بتوان ساخت؟
- چه میزان از ترکیب عملیات استفاده شود؟ آیا باید تمام عمیات عمومی دامنه را با استفاده از عملیات ابتدایی ساخت؟

استفادهٔ بیشتر از این روش‌ها به خصوصیات زیر برای موتور تطبیق می‌انجامد:

- قابلیت تطبیق بیشتر: کارهای بیشتری را می‌توان بدون برنامه‌نویسی انجام داد.
- عمومیت بیشتر دامنه: در حرکت به‌سمت کلی شدن اطلاعات مربوط به دامنه کم‌کم از مدل حذف می‌شوند و درنتیجه محصولات دامنه‌های بیشتری را می‌توان با موتور تطبیق ایجاد کرد.
- کارایی کمتر: به‌دلایل زیر کارایی کمتر می‌شود:

۱) عملیات عمومی برای انتخاب از بین حالات مختلف پشتیبانی شده باید پارامترهای ورودی و مدل محصول را تفسیر کنند. این کار زمان‌گیرتر از اجرای یک برنامهٔ غیرقابل تطبیق است که انواع موجودیت و عملیات روی آن‌ها، در آن حک شده‌اند.

۲) ازآن‌جاکه مشخصه‌های موجودیت‌ها در اشیاء جداگانه‌ای قرار دارند، دسترسی به آن‌ها نیازمند پیمایش زنجیرهای فراخوانی بزرگ‌تری است و درنتیجه زمان‌گیرتر است.

۳) در اثر فشرده بودن مدل داده‌ای (تمام موجودیت‌ها در یک یا تعداد بسیار کمی جدول از پایگاه داده‌ها ذخیره می‌شوند) پرس‌وجوها پیچیده‌تر و درنتیجه زمان‌گیرتر می‌شوند.

- سادگی کمتر: هرچه قابلیت‌های یک موتور تطبیق بیشتر شود و عمومی‌تر باشد، زبان تعریف مدل محصول آن، مفصل‌تر شده و از نظر پیچیدگی به یک زبان برنامه‌نویسی نزدیک‌تر می‌شود.
- دشواری بیشتر در بررسی نوع: باتوجه‌به آزادی عمل بیشتر متخصص دامنه، بررسی ورودی‌های او برای پیدا کردن اشکالات دشوارتر می‌شود و در نتیجه خطاهای بیشتری در زمان اجرا اتفاق می‌افتد.
- دشواری بیشتر در مهندسی دامنه: درک فراگیر نیازمندی‌های آینده کاری بسیار دشوار است.
- دشواری بیشتر در پیاده‌سازی سیستم: هرچه قابلیت تطبیق بیشتر می‌شود سیستم حجیم‌تر و پیچیده‌تر می‌شود و درنتیجه پیاده‌سازی آن سخت‌تر است.

سبک-سنگین کردن این مزایا و معایب و تصمیم‌گیری در مورد میزان استفاده از روش‌های مطرح شده، کاری است که باید در طراحی هر سیستم قابل تطبیق، توسط معماران نرم‌افزار انجام شود. به‌عبارت‌دیگر، ازآن‌جاکه سیستم‌های قابل تطبیق از نظر حجم داده، تعداد انواع موجودیت، عملیات، و سایر ابعاد باهم متفاوت هستند نمی‌توان برای استفاده از سه روش گفته شده، یک قاعدهٔ عمومی استخراج کرد. بااین‌حال، راهنمایی‌های زیر مؤثر هستند:

- موجودیت‌هایی که ایستا یا پویا بودن آن‌ها تأثیری در سیستم ندارد، در صورت امکان، بهتر است به‌صورت ایستا ایجاد شوند.
- استفاده از عملیات عمومی دامنه در مجموع سریع‌تر و ساده‌تر از ترکیب عملیات است. در نتیجه باید بنا را بر استفاده از این‌گونه عملیات گذاشت. به‌عبارت‌دیگر، بهتر است از اعمال ترکیبی تنها به‌صورت یک راه‌حل موقتی استفاده شود و در اولین زمان این عملیات با عملیات دامنه که از طریق برنامه‌نویسی ایجاد می‌شوند، جایگزین گردند.
- در زمان مهندسی دامنه نباید در پیش‌بینی حالات آینده افراط کرد. زیرا شناسایی تمام حالات، حتی اگر ممکن باشد، بسیار زمان‌گیر است.

۵—۴ جمع‌بندی

در این فصل راه‌حلی کلی برای مسئله قابلیت تطبیق نرم‌افزار ارائه گردید. موتورهای تطبیقی که این راه‌حل را به‌طور کامل پیاده‌سازی کنند خواص زیر را خواهند داشت:

۱) امکان تطبیق ساختاری برای سیستم‌های شیءگرا با هر مدل و هر تعداد سلسله‌مراتب

۲) امکان استفاده از عملیات تعبیه شده در سیستم برای موجودیت‌های جدید

۳) امکان تعریف عملیات جدید با ترکیب عملیات ابتدایی

۴) وراثت ساختار و رفتار موجودیت‌ها

بنابراین، با در نظر گرفتن مباحث مطرح در قسمت ۵—۳، اهداف «تغییر در زمان اجرا» و «سادگی» موتورهای تطبیق که در قسمت ۴—۲ به‌عنوان اهداف اصلی معرفی شد، توسط این راه‌حل قابل برآورده شدن می‌باشد. ضمن این‌که ایدهٔ ترکیب عملیات می‌تواند اهداف فرعی «عمومیت دامنه» و «عمومیت تعریف و تطبیق» را برآورده کند. اما لازم است معمار نرم‌افزار در نحوهٔ استفاده از این روش‌ها دقت کند.

همان‌طور که قبلاً گفته شد راه‌حل ارائه شده برای قابلیت تطبیق، برای تمام قسمت‌های هر سیستمی مناسب نیست، بلکه برای سیستم‌ها یا قسمت‌هایی از سیستم مناسب است که به دلایل نام‌برده شده در ۱—۲ نیازمند قابلیت تطبیق هستند. در فصل بعد یک مورد مطالعه برای نشان دادن قابلیت پیاده‌سازی این ایده‌ها ارائه می‌گردد و نشان داده می‌شود که صورت مسئلهٔ این پایان‌نامه توسط این روش حل شده است.

فصل ٦

مورد مطالعه: پیاده‌سازی یک خط تولید قابل تطبیق

به منظور ارزیابی قابلیت پیاده‌سازی مدل ارائه شده در فصل ۵ بخشی از آن در قالب سیستمی به‌نام خط‌تطبیق پیاده‌سازی شده است. ایده‌های زیر در این سیستم پیاده‌سازی شد:

۱) اعمال عمومی پارامتری

۲) ترکیب عملیات ابتدایی برای ساختن اعمال پیچیده‌تر

البته باتوجه‌به آزمایشی بودن این سیستم، هدف پیاده‌سازی کامل روش ارائه شده و امکان بهره‌برداری کامل از این خط تولید نبوده است. البته توجه به این نکتهٔ ضروری است که پیاده‌سازی موفقیت‌آمیز این روش در سیستم خط‌تطبیق، به‌معنی قابل استفاده بودن آن در تمام سیستم‌ها نیست و تشخیص مناسب بودن این روش برای یک مسئله نیازمند نظر معماران نرم‌افزار است. در این فصل این پیاده‌سازی مورد بررسی قرار می‌گیرد.

در قسمت ٦—١ توضیح داده می‌شود که معماری سیستم خط‌تطبیق شامل چه مواردی می‌شود. قسمت ٦—٢ امکانات سیستم خط‌تطبیق را برای تعریف و به‌روزرسانی مدل محصول توضیح می‌دهد. در قسمت ٦—٣ نحوهٔ تفسیر مدل محصول در سیستم خط‌تطبیق توضیح داده می‌شود. قسمت ٦—۴ نتایج این مورد مطالعه را ارائه می‌دهد.

٦—١ سیستم خط‌تطبیق برای دامنهٔ سیستم‌های داده‌گرا

این سیستم برای ایجاد نرم‌افزارهایی از خانوادهٔ سیستم‌های داده‌گرا[1] فوق‌العاده پویا قابل استفاده است. سیستم‌های داده‌گرا سیستم‌هایی هستند که وظیفهٔ اصلی آن‌ها عملیات غیرپیچیده (مثل ذخیره‌سازی، به‌روزرسانی، و جست‌وجو) بر روی موجودیت‌های محیط می‌باشد. در سیستم‌های داده‌گرا فوق‌العاده پویا، موجودیت‌های دامنه و عملیات روی آن‌ها به‌شدت متغیر می‌باشد. همانطور که قبلاً ذکر شد، بیمه و خدمات درمانی دو نمونه از این محیط‌ها هستند. خصوصیات سیستم خط‌تطبیق عبارت است از:

[1] Data-Oriented

- امکان تعریف موجودیت‌ها، مشخصه‌ها، و انتساب‌های بین موجودیت‌ها
- انواع دادهٔ ابتدایی: صحیح[۲]، رشته[۳]، و بولی[۴]
- اعمال ابتدایی عمومی: create، display، search (بر روی یک موجودیت)، و email
- ترکیب عملیات ابتدایی با استفاده از نمودار فعالیت
- تعریف پیش‌شرط و پس‌شرط برای هر یک از اعمال موجود در یک نمودار فعالیت

در این سیستم وراثت پیاده‌سازی نشده است زیرا برای نشان دادن قابلیت پیاده‌سازی ایده‌ها نیازی به پیاده‌سازی وراثت نیست. برای توضیح بیشتر کافی است توجه شود که وراثت کارکردی جز تعمیم مشخصه‌ها و اعمال یک موجودیت به بچه‌های آن ندارد. برای ارث‌بری مشخصه‌ها کافی است قسمت‌هایی از برنامه که مشخصه‌های یک نوع موجودیت را پیمایش می‌کنند به‌صورت بازگشتی[۵] روی پدران موجودیت نیز اجرا شود. در مورد اعمال عمومی و اعمال مرکب نیز اگر این اعمال بر روی موجودیتی قابل اجرا باشند ارث‌بری آن‌ها کار بسیار ساده‌ای است. درحقیقت، بازهم نوشتن برنامه‌ها به‌صورت بازگشتی مشکل را حل می‌کند. تنها مشکلی که ممکن است وراثت ایجاد کند، افت بیشتر کارایی است.

اعمال ابتدایی عمومی، نمونه‌ای از اعمال عمومی هستند. پیاده‌سازی اعمال ابتدایی، باتوجه‌به این‌که روی موجودیت‌های بیشتری قابل اجرا هستند، دشوارتر از پیاده‌سازی اعمال دامنه می‌باشد. بنابراین، پیاده‌سازی و اجرای این اعمال نشان می‌دهد که ایدهٔ عمومی کردن برنامه‌ها از طریق پارامتری کردن، قابل پیاده‌سازی است.

این سیستم از زبان جاوا برای برنامه‌نویسی و از Ms SQL Server به‌عنوان سیستم مدیریت پایگاه‌داده‌ها استفاده می‌کند. همچنین، از فایل‌های XML برای تعریف اعمال مرکب استفاده می‌شود.

پیش از ادامهٔ بحث، به شکل ٦—١ که مدل کامل‌تری از مثال فروشگاه را نشان می‌دهد، توجه کنید. در این مثال به‌جز محصولات، کلاس‌های دیگری نیز وجود دارد.

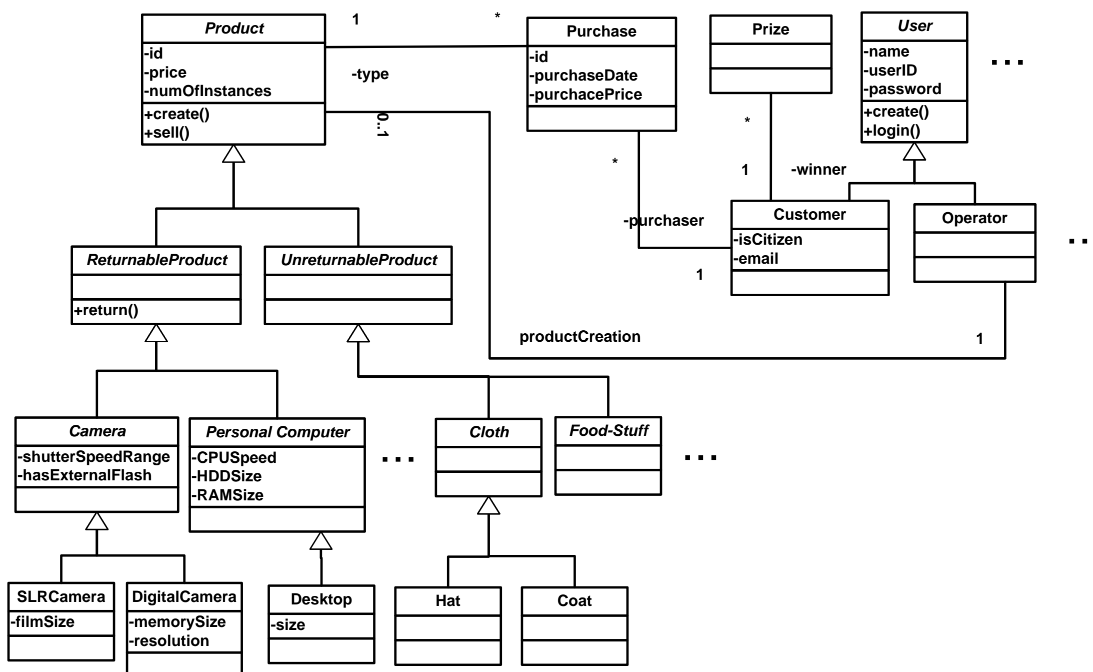
٦—٢ امکانات تعریف و به‌روزرسانی مدل

مدلی که در فصل ۵ ارائه شد، بخشی از معماری موتور تطبیق را نشان می‌دهد که برای ذخیره‌سازی مدل محصول قابل استفاده است. شکل ٦—٢ مدل ماندگاری[٦] سیستم خط‌تطبیق را نشان می‌دهد. علاوه بر این موارد، سیستم نیازمند ابزاری برای تعریف پویای مدل محصول می‌باشد. این ابزار امکانات زیر را به متخصصین دامنه ارائه می‌دهد:

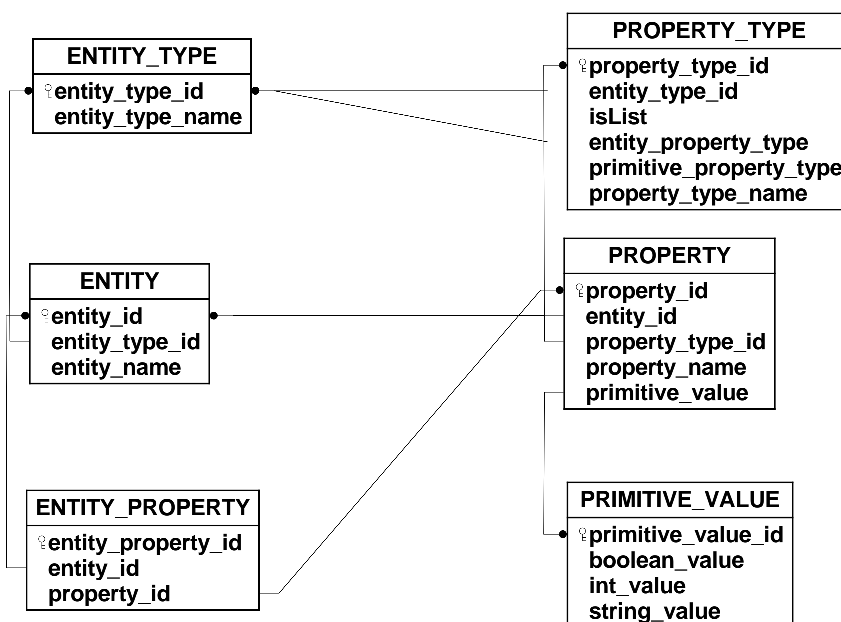
- تعریف و تغییر انواع موجودیت‌های قابل تطبیق، مانند ProductType در مثال فروشگاه

---

[۲] Integer
[۳] String
[۴] Boolean
[۵] Recursive
[٦] Persistence



شکل ٦–١: نمودار کلاس کامل مثال فروشگاه



شکل ٦–٢: مدل ماندگاری سیستم خط‌تطبیق

Add New Entity Type

Add New Entity Type And it's Property Types

Entity Name Attribute

Entity Name: Purchase

Add Entity Type

PropertyType

Property Type Name

Property Type Value is a List

Primitive Value: int

Entity Type Value: Customer

Add New Property Type

Property TYpe Name	Property Type is a List	Value Type
id	FALSE	int
purchaseDate	FALSE	String
price	FALSE	int
customer	FALSE	Customer
product	FALSE	SLRCamera

Delete Selected Property Types

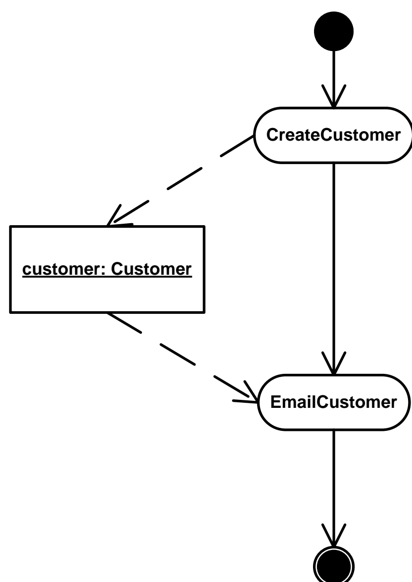
شکل ٦—۳: فرم تعریف موجودیت Purchase

- تعریف و تغییر انتساب عملیات عمومی به انواع موجودیت‌ها
- تعریف و تغییر عملیات مرکب بر اساس عملیات ابتدایی
- تعریف و تغییر محدودیت‌ها و انتساب آن‌ها به موجودیت‌ها و عملیات

در یک موتور تطبیق برای ورود مدل به سیستم از دو روش می‌توان استفاده کرد: فایل‌های متنی (مانند فایل‌های XML) یا واسط کاربر (شامل فرم‌هایی برای تعریف موجودیت‌ها، نمودار فعالیت برای تعریف عملیات مرکب و ...). برای تغییر مدل نیز می‌توان از همین دو روش استفاده کرد با این تفاوت که سیستم پس از اطلاع از تغییر، باید عملیات لازم را برای مهاجرت نمونه‌های موجود انجام دهد. مدل تعریف شده را می‌توان در فایل یا پایگاه‌داده‌ها ذخیره کرد. انتخاب این روش‌ها تأثیر چندانی در معماری موتور تطبیق ندارد. در سیستم خط‌تطبیق از واسط کاربر برای ورود انواع موجودیت‌ها و از فایل‌های XML برای تعریف عملیات مرکب استفاده می‌شود. تعریف موجودیت‌ها و مشخصه‌های آن‌ها در پایگاه‌داده‌ها ذخیره‌سازی می‌شود.

شکل ٦—۳ فرم تعریف یک نوع موجودیت به‌نام Purchase را از مثال فروشگاه نشان می‌دهد، که خود با دو نوع موجودیت دیگر در ارتباط است.

همان‌طورکه گفته شد، یکی از ایدهٔهای عرضه شده در این پایان‌نامه تعریف عملیات مرکب بااستفاده از اعمال ابتدایی می‌باشد. یک عمل مرکب قابل تعریف در مثال فروشگاه عبارت است از ثبت‌نام مشتری: ابتدا مشتری اطلاعات خود را وارد می‌کند و سپس یک نامهٔالکترونیکی به او فرستاده می‌شود تا موفقیت‌آمیز بودن ثبت‌نام به او اطلاع داده شود. شکل ٦—۴ نمودار فعالیت و شکل ٦—۵ توصیف این عمل به‌زبان XML را نشان می‌دهد.



شکل ٦—۴: نمودار فعالیت عمل ثبت‌نام مشتری

```
<?xml version="1.0" encoding="utf-8" ?>

<Task name="Register-Customer">
   <node type="start">
      <nextnode>CreateCustomer</nextnode>
   </node>

   <node type="operation" name="CreateCustomer" genericOperation="Create">
      <output type="customer" multiplicity="single" name="c"></output>
      <nextnode>EmailCustomer</nextnode>
   </node>

   <node type="operation" name="EmailCustomer" genericOperation="Email">
      <input type="customer" multiplicity="single" name="c"></input>
      <parameter type="emailInfoFile">create-customer.txt</parameter>
      <nextnode>end</nextnode>
   </node>

   <node type="end"></node>
</Task>
```

شکل ٦—۵: تعریف عمل ثبت‌نام مشتری با استفاده از زبان XML

```xml
<?xml version="1.0" encoding="utf-8" ?>

<Task name="Prize">
   <node type="start">
      <nextnode>SearchCustomer</nextnode>
   </node>

   <node type="operation" name="SearchCustomer" genericOperation="Search">
      <output type="customer" multiplicity="single" name="prizeNominee">
         <rule fieldName="sumOfShoppings" type="MAX"></rule>
      </output>
      <nextnode>LeastShoppingCondition</nextnode>
   </node>

   <node name="LeastShoppingCondition" type="conditional">
      <input type="customer" multiplicity="single" name="prizeNominee"></input>
      <constraint>prizeNominee.sumOfShoppings>2000000</constraint>
      <thennode>CreatePrize</thennode>
      <elsenode>end</elsenode>
   </node>

   <node type="operation" name="CreatePrize" genericOperation="Create">
      <input type="customer" multiplicity="single" name="prizeNominee"></input>
      <output type="prize" multiplicity="single" name="aw">
         <rule fieldName="winnerId" type="equal">prizeNominee.id</rule>
      </output>
      <nextnode>EmailCustomer</nextnode>
   </node>

   <node type="operation" name="EmailCustomer" genericOperation="Email">
      <input type="customer" multiplicity="single" name="prizeNominee"></input>
      <parameter type="emailInfoFile">prize.txt</parameter>
      <nextnode>end</nextnode>
   </node>

   <node type="end">
   </node>
</Task>
```

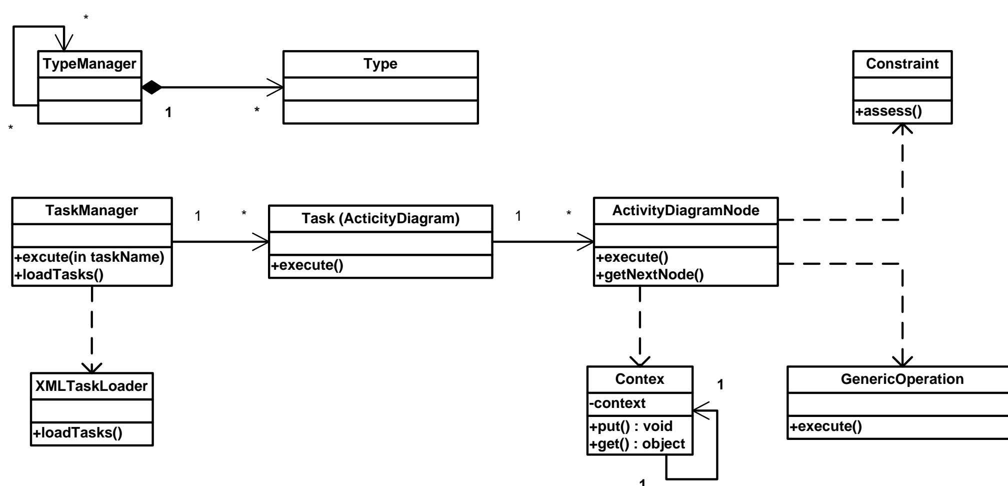
شکل ٦—٦: تعریف عمل جایزه با استفاده از زبان XML

نمودار فعالیت عمل مرکب جایزه در شکل ۵—۹ نشان داده شد. تعریف این عمل به‌زبان XML در ٦—٦ نشان داده شده است.

٦—۳ تفسیر مدل شیء محصول

تا این جا نحوهٔ تعریف، ذخیره‌سازی، و تغییر یک مدل قابل تطبیق بیان گردید، اما در مورد تفسیر و اجرای این مدل بحث چندانی صورت نگرفت. تفسیر مدل شامل بخش‌های زیر می‌شود:

- خواندن مدل از فایل یا پایگاه داده‌ها و تبدیل آن به اشیاء سطح دانش: این عمل ممکن است یک‌باره انجام شود یا به‌علت ملاحظات کارایی هرقسمت برحسب نیاز خوانده شود.
- فراخوانی عملیات دامنه با ارسال پارامترهای مناسب: باتوجه‌به این‌که امکان دارد عملی، محدودیتی را نقض کند و یا به‌دلایل دیگر مثل بروز خطا، ناتمام بماند، لازم است عملیات دامنه درون تراکنش‌ها اجرا شوند.



شکل ۶—۷: کلاس‌های سیستم خط‌تطبیق برای تفسیر مدل

- تفسیر جریان کنترل عملیات مرکب و فراخوانی عملیات ابتدایی با ارسال پارامترهای مناسب: شبیه عملیات عمومی در این‌جا نیز باید در اجرای یک عمل پیچیده از تراکنش استفاده شود.
- برآورد کردن[۷] محدودیت‌ها در آغاز و پایان اجرای عملیات

شکل ۶—۷ نمودار کلاس سیستم خط‌تطبیق برای تفسیر مدل را نشان می‌دهد. وظیفهٔ TaskManager مدیریت و اجرای عملیات مرکب است که در قالب اشیاء کلاس Task تعریف شده‌اند. اشیاء ورودی و خروجی هر عمل ابتدایی عضو یک عمل مرکب از طریق یک شیء Context در اختیار آن قرار داده می‌شود. TaskManager برای خواندن تعریف عملیات مرکب از کلاس XMLTaskLoader استفاده می‌کند. کلاس TypeManager انواع موجودیت‌ها و انواع ابتدایی موجود در سیستم را مدیریت می‌کند.

شکل ۶—۸ اجرای عمل ثبت‌نام مشتری را با استفاده از نمودار ترتیب UML نشان می‌دهد. شکل ۶—۹ فرم ورود اطلاعات مشتری که حاصل از تفسیر این عمل مرکب می‌باشد را نشان می‌دهد.

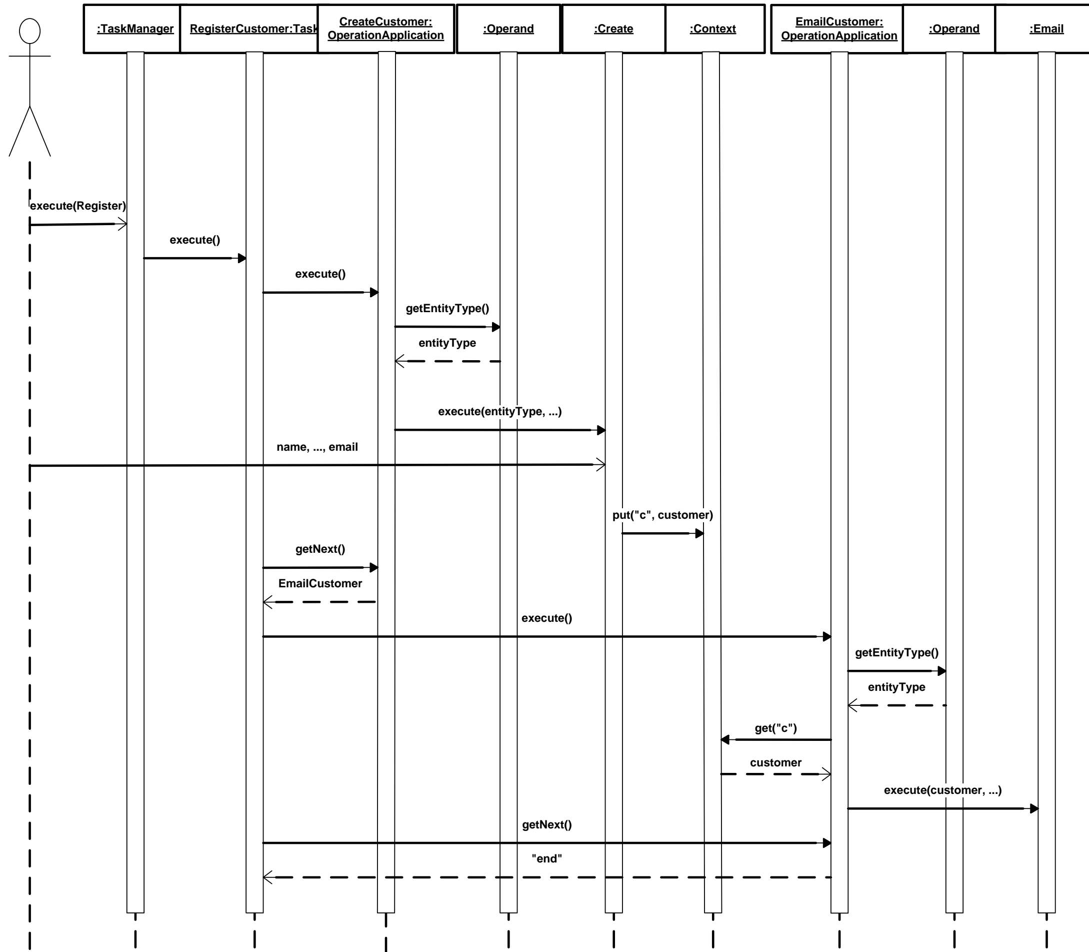
۶—۴ نتیجه‌گیری

سیستم خط‌تطبیق، باوجود محدودیت‌هایی که در مدل‌سازی سیستم‌های قابل تطبیق دارد، نشان‌دهندهٔ میزان سودمندی ایده‌های مطرح شده در این پایان‌نامه، می‌باشد. نتایج زیر از این پیاده‌سازی و تصویری که از آیندهٔ آن داریم، به‌دست می آید:

۱) ایدهٔ عملیات عمومی قابل پیاده‌سازی است، هرچند درک و پیاده‌سازی آن دشوار است.

۲) ایدهٔ ترکیب عملیات ابتدایی برای ایجاد عملیات مرکب قابل پیاده‌سازی است، اما مشکلاتی دارد که سودمندی آن‌را محدود می‌کند: اگر قدرت عملیات ابتدایی قرار داده شده در سیستم زیاد باشد (به عبارت دیگر، حالات مختلف مورد نیاز را پشتیبانی کنند) پیچیدگی زبان تعریف عملیات زیاد شده و

[۷] Assess



شکل ٦—٨: اجرای عمل ثبت‌نام مشتری

Create

Create customer

Entity Name : customer1

Property Value

id	501
name	MS Makarem
userid	82265708
password	sh6743
iscitizen	✔
email	ıakarem@sharif.edu
sumofshoppings	0

Add

شکل ٦—٩: فرم ایجاد مشتری

قابلیت استفاده از آن کاهش می‌یابد. ازسوی‌دیگر، اگر حالات پشتیبانی شده توسط این عملیات محدود باشد، نمی‌توان اعمال مورد نیاز سیستم را با این عملیات ایجاد کرد.

ایدهٔ ادغام سلسله‌مراتب‌های تکاملی، قابل پیاده‌سازی است. درحقیقت، این ایده در برخی سیستم‌های قابل تطبیق پیاده‌سازی شده است و هدف از سیستم خط‌تطبیق بررسی این ایده نبوده است. اما قابل استفاده بودن ایدهٔ ادغام سلسله‌مراتب‌های تکاملی و غیرتکاملی در سیستم‌های مختلف بستگی به تعداد موجودیت‌ها و پیچیدگی سیستم دارد. ممکن است افت کارایی اجازهٔ یک پیاده‌سازی قابل تطبیق را برای تمام سلسله‌مراتب‌ها ندهد. ارزیابی کارایی موتورهای تطبیق نیازمند یک مطالعهٔ مجزا می‌باشد.

بنابر آن‌چه گفته شد، سه هدف مطرح شده در صورت مسئلهٔ این پایان‌نامه، برآورده شده‌اند. به‌عبارت‌دیگر، جنبه‌های مختلف هدف تطبیق رفتاری با استفاده دو ایدهٔ عملیات عمومی و ترکیب عملیات برآورده شد. همچنین، زبان ارائه شده برای تعریف ویژگی‌های سیستم یک زبان سطح بالا و قابل فراگیری برای متخصصین دامنه می‌باشد و در ضمن می‌توان یک واسط کاربری گرافیکی برای تعریف و تطبیق مدل در اختیار متخصصین دامنه قرار داد. همچنین، تعریف یا تطبیق یک ویژگی از مدل محصول در زمانی کمتر از یک ساعت قابل انجام است.

باز هم تأکید می‌شود که به‌علت ملاحظاتی مثل کارایی ممکن است نتوان در برخی سیستم‌های نیازمند تطبیق منطق‌حرفه توسط انسان، از این روش استفاده کرد. همچنین، نحوهٔ و میزان به‌کارگیری این روش‌ها در موفقیت سیستم مؤثر است.

فصل ۷

نتیجه‌گیری و کارهای آینده

این فصل، به‌طور خلاصه، نتایج این پایان‌نامه را بررسی کرده و به بیان کارهای آینده می‌پردازد.

دست‌آورد اصلی

در این پایان‌نامه، راه‌حلی عمومی برای تطبیق ساختاری و رفتاری سیستم‌های شیءگرا ارائه شد. مبنای این راه‌حل سبک معماری مدل قابل تطبیق شیء می‌باشد. موتورهای تطبیقی که این راه‌حل را به‌طور کامل پیاده‌سازی کنند، خواص زیر را خواهند داشت:

۱) امکان تطبیق ساختاری برای سیستم‌های شیءگرا با هر مدل و هر تعداد سلسله‌مراتب که شامل امکان تعریف موجودیت‌ها و ارتباطات بین آن‌ها می‌شود.

۲) امکان استفاده از عملیات تعبیه شده در سیستم برای موجودیت‌های جدید

۳) امکان تعریف عملیات جدید با ترکیب عملیات ابتدایی

۴) وراثت ساختار و رفتار موجودیت‌ها

برای نشان دادن این‌که این ایده‌ها قابل پیاده‌سازی می‌باشند، بخشی از معماری ارائه شده برای موتورهای تطبیق در سیستمی به‌نام خط‌تطبیق پیاده‌سازی شده است. که نتایج آن در فصل ٦ ارائه شد.

همان‌طور که قبلاً گفته شد، راه‌حل ارائه شده برای قابلیت تطبیق، لزوماً برای هر سیستم یا تمام قسمت‌های هر سیستمی مناسب نیست، بلکه برای سیستم‌ها یا قسمت‌هایی از سیستم‌ها مناسب است که به‌دلایل نام‌برده شده در ۱ــ۲ نیازمند قابلیت تطبیق هستند. به‌دلیل افت کارایی ممکن است ایده‌های مطرح شده، قابل به‌کارگیری نباشند.

سیستم‌های مبتنی بر سبک معماری مدل قابل تطبیق شیء، از جمله سیستم‌های قابل تطبیق ایجاد شده با روش ارائه شده، معایبی هم دارند. از جمله این‌که ایجاد این‌گونه سیستم‌ها دشوارتر از سیستم‌های عادی است. همچنین، این‌طور نیست که در این سیستم‌ها هرگونه تطبیقی از طریق خود سیستم پشتیبانی شود، بلکه در بعضی موارد ممکن است تغییراتی مورد نیاز باشد که به تغییر متن برنامه منجر شود. باتوجه‌به پیچیدگی سیستم اعمال این تغییرات کار ساده‌ای نیست. می‌توان برای ساده‌کردن ایجاد و نگه‌داری

سیستم مدل قابل تطبیق شیء را با سایر روش‌ها ترکیب کرد (همانطور که قبلاً گفته شد [۳۱] با استفاده از برنامه‌نویسی جنبه‌گرا راه‌حلی برای ایجاد و نگه‌داری ساده‌تر این‌گونه سیستم‌ها ارائه می‌دهد.) علاوه بر مشکلات گفته شده، سیستم‌هایی که براساس سبک معماری مدل قابل تطبیق شیء ایجاد می‌شوند، معمولاً، کارایی پایین‌تری نسبت به سیستم‌های مشابه غیرقابل تطبیق دارند [۱۱]. از وجود این مشکلات، می‌توان نتیجه گرفت که نباید در استفاده از قابلیت تطبیق زیاده‌روی کرد. تشخیص میزان مناسب قابلیت تطبیق در هر سیستمی برعهدهٔ معماران نرم‌افزار می‌باشد.

دست‌آوردهای فرعی

به‌غیر از روشی عمومی برای ایجاد نرم‌افزارهای قابل تطبیق، این پایان‌نامه دست‌آوردهای فرعی زیر را دارد:

۱) کاهش مسئلهٔ قابلیت تطبیق به مسئلهٔ قابلیت تعریف پویا

۲) معرفی موتور تطبیق به‌عنوان یک ابزار زیربنایی برای ایجاد نرم‌افزارهای قابل تطبیق

۳) شرح چرخهٔ حیات سیستم‌های قابل تطبیق

۴) معرفی سه تاکتیک «داده به‌جای برنامه»، «عمومی کردن برنامه‌ها از طریق پارامتری کردن»، و «ترکیب عملیات در زمان اجرا برای ساختن عملیات پیچیده‌تر» برای دست‌یابی به قابلیت تطبیق

۵) معرفی تکنیک‌هایی برای پیاده‌سازی تاکتیک‌های فوق در سیستم‌های شیءگرا

٦) ارائهٔ یک حالت (مدل) کلی برای سیستم‌های شیءگرا نیازمند قابلیت تطبیق

کارهای آینده

قابلیت تطبیق باوجوداین‌که در بسیاری از سیستم‌ها به‌شکل‌های مختلف به‌کار گرفته شده، هنوز مسائل حل نشدهٔ بسیاری دارد و مسائل حل شدهٔ آن نیز به‌دلیل عدم اطلاع محققین حوزه‌های مختلف علوم کامپیوتر و مهندسی نرم‌افزار مورد استفادهٔ مجدد قرار نمی‌گیرد. به‌طورخلاصه، مطالعه و تحقیق بیشتر در زمینه‌های زیر برای ایجاد نرم‌افزارهای قابل تطبیق مفید است:

- بررسی سایر سیستم‌هایی که به‌نحوی از قابلیت تطبیق پشتیبانی می‌کنند مانند سیستم‌های جریان‌کار قابل تطبیق و زبان‌های برنامه‌نویسی دارای انعکاس
- ارزیابی و بهبود کارایی سیستم‌های مبتنی بر مدل قابل تطبیق شیء
- بررسی نوع، جلوگیری از وقوع خطاهای زمان اجرا، و عکس‌العمل مناسب در برابر این خطاها
- بهبود مدل داده‌ای سیستم
- مهاجرت نمونه‌ها
- نگه‌داری سابقهٔ تغییرات و امکان عقب‌گرد
- طراحی زبان‌هایی که برای متخصصین دامنه قابل فهم‌تر هستند

- ترکیب سبک معماری مدل قابل تطبیق شیء با انعکاس زبان‌های برنامه‌نویسی برای برقراری ارتباط بین موجودیت‌های (سلسله‌مراتب‌های) قابل تطبیق و غیرقابل تطبیق سیستم، همچنین، برای استفادهٔ روش تطبیق رفتاری در سیستم‌هایی با ساختار ایستای ایجادشده از طریق برنامه‌نویسی
- تبدیل خودکار پرس‌وجوهای مدل غیرقابل تطبیق به مدل قابل تطبیق: ازآن‌جاکه مدل داده‌ای عوض شده است و تمام نمونه موجودیت‌ها در چند رابطهٔ محدود ذخیره می‌شوند، پرس‌وجوها پیچیده‌تر می‌شوند. بنابراین، اگر بتوان پرس‌وجوهای نوشته شده برای مدل غیرقابل تطبیق را به پرس‌وجوهای مدل قابل تطبیق تبدیل کرد، گامی مهم در جهت ساده کردن پیاده‌سازی سیستم‌های مبتنی بر مدل قابل تطبیق شیء برداشته شده است.

پیوست A

واژه‌نامه‌ی فارسی به انگلیسی

ارسال پیام .... Message Passing
استثناء (در جریان کار) .... Exception
استفادهٔ مجدد جعبه سفید .... Whitebox Reuse
استفادهٔ مجدد جعبه سیاه .... Blackbox Reuse
استقرار .... Deployment
اشاره‌گر .... Pointer
اعتبارسنجی .... Validation
الگو .... Pattern
الگوهای حرفه .... Business Patterns
الگوهای پیاده‌سازی .... Idioms
الگوهای تحلیل .... Analysis Patterns
الگوهای طراحی .... Design Patterns
الگوهای فرآیند .... Process Patterns
الگوهای معماری .... Architectural Patterns
الگوی تحلیل مشاهده .... Observation Analysis Pattern
الگوی طراحی خصوصیت .... Property Design Pattern
الگوی طراحی راهبرد .... Strategy Design Pattern
الگوی طراحی سازنده .... Builder Design Pattern
الگوی طراحی شیءقاعده .... Rule-Object Design Pattern
الگوی طراحی مفسّر .... Interpreter Design Pattern
الگوی طراحی شیء نوع .... Type-Object Design Pattern
الگوی معماری انعکاس .... Reflection Architectural Pattern
امنیت .... Security
انتساب .... Association
انتقال نمونه .... Instance Migration
انعطاف‌پذیری .... Flexibility
انعکاس .... Reflection

انعکاس رفتاری .... Behavioral Reflection
انعکاس زبانی .... Linguistic Reflection
انعکاس ساختاری .... Structural Reflection
انعکاس محاسباتی .... Computational Reflection
انعکاس معماری .... Architectural Reflection
انعکاس نامحدود .... Non-Restrictive Reflection
انعکاسی .... Reflective
ایجاد یک محصول از خانواده .... Product Instantiation
ایستا .... Static
باز بودن .... Openness
بررسی نوع .... Type Checking
برنامهٔ کاربردی .... Application
برنامه‌نویسی جنبه‌گرا .... Aspect Oriented Programming (AOP)
برنامه‌نویسی خصوصیت‌گرا .... Feature Oriented Programming
برنامه‌نویسی تدافعی .... Defensive Programming
برنامه‌نویسی تطبیقی .... Adaptive Programming
برنامه‌نویسی توسط کاربر نهایی .... End-User Programming
برنامه‌نویسی تولیدی .... Generative Programming
برنامه‌نویسی مبتنی بر قرارداد .... Contract-Based Programming
برنامه‌نویسی موضوع‌گرا .... Subject Oriented Programming
پارامتری کردن .... Parametrization
پالایش .... Refinement
پالایش خودکار .... Automatic Refinement
پس‌شرط .... Postcondition
پنهان‌سازی اطلاعات .... Information Hiding
پویا .... Dynamic
پیش‌شرط .... Precondition
پیکربندی .... Configuration
تحمل خطا .... Fault Tolerance
تبدیل .... Transformation
ترکیب .... Composition
تعریف .... Definition
تعامل .... Collaboration or Interaction
تغییر ایستا .... Static Change
تغییر پویا .... Dynamic Change
تطبیق‌یاب .... Adaptive
تطبیق‌یابی .... Adaptivity
تکامل نرم‌افزار .... Software Evolution
تمایلات کاربر .... User Prefrences

تولیدکننده	Developer
تولید نرم‌افزار	Software Development
تولید نرم‌افزار بر مبنای مؤلفه‌ها	Component-Based Software Development
توزیع	Distribution
ثابت	Invariant
جبران خطا	Failure Recovery
جریان کار	Workflow
جریان کنترل	Control Flow
جداسازی دغدغه‌ها	Separation of Concerns
جداسازی چندبعدی دغدغه‌ها	Multi-Dimensional Separation of Concerns
جهان‌بینی	Paradigms
چارچوب جعبه سفید	Whitebox Framework
چارچوب جعبه سیاه	Blackbox Framework
چارچوب شی‌گرا	Object Oriented Framework
چرخهٔ تکامل محصول	Product Evolution Cycle
چرخهٔ تکامل موتور تطبیق	Adaptation Engine Evolution Cycle
چرخهٔ حیات سیستم قابل تطبیق توسط انسان	Human-Adaptable System Lifecycle
حالت ناسازگار	Inconsistent State
حالت نامعتبر	Invalid State
حک	Hardcode
خانواده‌ای از محصولات	Product Family
خط تولید نرم‌افزار	Software Product-Lines
خط‌تولید تطبیقی	Adaptable Product-Lines
خنثی	Override
خود-تطبیق	Self-Adaptive
خود-تطبیقی	Self-Adaptivity
خودنگری (انعکاس محاسباتی)	Introspection
دامنه	Domain
در دسترس بودن	Availability
درشت‌دانه	Coarse-grained
دید معماری	Architectural View
راهنما	Guideline
رفتار	Behavior
روش‌های تولیدی	Generative Methods
ریزدانه	Fine-grain
زبان مخصوص دامنه	Domain-Specific Language
زبان مدل‌سازی تصویری	Visual Modeling Language
سابقه	History
ساختار	Structure

ساختار معماری .... Architectural Structure
سبک معماری مدل قابل تطبیق شیء .... Adaptive Object Model Architecture Style
سطح پایه .... Base Level
سطح تجرید .... Abstraction Level
سطح دانش .... Knowledge Level
سطح عملیاتی .... Operational Level
سلسله‌مراتب .... Hierarchy
سلسله‌مراتب تکاملی (پویا) .... Evolutionary (Dynamic) Hierarchy
سلسله‌مراتب ثابت (ایستا) .... Non-Evolutionary (Static) Hierarchy
سلسله‌مراتب کاهش‌یافته .... Reduced Hierarchy
سیستم پایگاه داده‌ها .... Database System
سیستم پایگاه داده‌های رابطه‌ای .... Relational Database System
سیستم جریان کار .... Workflow System
سیستم جریان کار قابل تطبیق .... Adaptive Workflow System
سیستم ایمنی-بحرانی .... Safety-Critical System
سیستم عامل .... Operating System
سیستم مأموریت-بحرانی .... Mission-Critical System
سیستم مبتنی بر مؤلفه .... Component-Based System
سیستم مبتنی بر مؤلفهٔ باز .... Open Component-Based System
سیستم مدیریت پایگاه داده‌ها .... Database Management System
سیستم مدیریت جریان کار .... Workflow Management System
سیستم مدیریت جریان کار قابل تطبیق .... Adaptive Workflow Management System
سیستم نمادگذاری .... Notation
سهام‌دار .... Stakeholder
شیء برنامهٔ کاربردی .... Application Object
عامل متحرک .... Mobile Agent
عقب‌گرد .... Rollback
عمل (کلاس، شیء) .... Method
عمل ابتدایی .... Primitive Operarion
عمل دامنه .... Domain Operation
عمل عمومی .... Genreric Operarion
عمل مستقل از دامنه .... Domain-Independent Operation
عمل مشترک در موجودیت‌ها .... Entity Cross-Cutting Operation
عمود بر هم .... Orthogonal
غیر مهم در سطح معماری .... Architecturaly Insignificant
فرآیند .... Process
فرآیند حرفه .... Business Process
فرآیند ویژه (دیمی) .... Ad Hoc Process
فراخوانی .... Invocation

فعالیت (در جریان کار) .... Activity, Task
قابل تطبیق .... Adaptable
قابلیت آزمون .... Testability
قابلیت اتصال اعمال جدید به سیستم .... Operation Plugability
قابلیت اتصال مؤلفه‌های جدید .... Component Plugability
قابلیت استفاده .... Usability
قابلیت استفادهٔ مجدد .... Reusability
قابلیت اطمینان .... Reliability
قابلیت حمل .... Portability
قابلیت تطبیق .... Adaptability
قابلیت تطبیق با کاربر .... User Adaptivity
قابلیت تغییر .... Modifiability or Changeability
قابلیت درک .... Understandability
قالب .... Template
قرارداد متا—شیء .... Meta-Object Protocol
قوانین حرفه .... Business Rules
کارایی .... Performance
کارایی کاربر .... Efficiency
کارکرد .... Functionality
کاهش سلسله‌مراتب .... Hierarchy Reduction
کلاس پیاده‌سازی .... Implementation Class
کلاس مجرد .... Abstract Class
کلاس واقعی .... Concrete Class
لایه‌بندی .... Layering
کلید خارجی .... Foreign Key
ماشین مجازی .... Virtual Machine
ماندگاری .... Persistence
متاداده .... Meta-Data
متالایه .... Meta-Layer
متامدل .... Meta-Model
متامدل‌سازی .... Metamodeling
متحرک‌سازی مدل .... Model Animation
متخصص دامنه .... Domain Expert
متغیر ارجاعی .... Reference Variable
محاسبات فراگیر .... Pervasive Computing
محاسبات سیار .... Mobile Computing
محصورسازی .... Encapsulation
محیط .... Environment
محیط مجتمع تولید .... Integrated Development Endironment (IDE)

مدل شیء .... Object Model
مدل قابل تطبیق شیء .... Adaptive Object Model
مدیریت تغییرات .... Change Management
مدیریت نیازمندی‌ها .... Requirements Management
مربع نوع .... Type Square
مرتبط علّی .... Causally-Connected
مستقل از دامنه .... Domain-Inepednent
مشخصه (کلاس، شیء) .... Attribute
معماری نرم‌افزار .... Software Architecture
معانی (زبان) .... Semantics
مقیاس‌پذیری .... Scalability
میان‌افزار .... Middleware
منطق حرفه .... Business Logic
مناسب‌سازی (خط‌تولید) .... Customization
موتور تطبیق .... Adaptation Engine
موجودیت .... Entity
موجودیت حرفه .... Business Entity
مؤلفه .... Component
مؤلفهٔ بزرگ—مقیاس .... Large-scale Components
مؤلفهٔ کوچک—مقیاس .... Small-scale Components
مؤلفه‌های تطبیقی اتصالی .... Adaptive Plug and Play Components (APPC)
مهم در سطح معماری .... Architecturaly Significant
مهندسی دامنه .... Domain Engineering
مهندسی محصول .... Product Engineering
نحو (زبان) .... Syntax
نمایش-از-خود .... Self-Representation
نمایندگی .... Delegation
نمودار ترتیب UML .... UML Sequence Diagram
نمودار حالت UML .... UML State Chart
نمودار فعالیت UML .... UML Activity Diagram
نمودار کلاس UML .... UML Class Diagram
نمونه .... Instance
نوع داده .... Data Type
نیازمندی‌های غیرکارکردی .... Non-Functional Requirements
نیازمندی‌های کارکردی .... Functional Requirements
واحدمندی .... Modularity
وارسی .... Verification
واسط .... Interface
وراثت .... Inheritance

وراثت چندگانه .... Multiple Inheritance
وصل‌کن-اجراکن .... Plug and Play
ویژگی .... Feature
ویژگی ساده .... Simple Feature
ویژگی ساده قابل تطبیق .... Simple Adaptable Feature
ویژگی کاملاً قابل تطبیق .... Totally Adaptable Feature
ویژگی کاملاً غیرقابل تطبیق .... Totally Not-Adaptable Feature
ویژگی مرکب .... Composite Feature
ویژگی کیفی .... Quality Attribute
هم‌ریخت .... Homomorphic

پیوست B

واژه‌نامه‌ی انگلیسی به فارسی

Abstraction Level .... سطح تجرید
Abstract Class .... کلاس مجرد
Activity .... فعالیت (در جریان کار)
Ad Hoc Process .... فرآیند ویژه (دیمی)
Adaptable .... قابل تطبیق
Adaptable Product-Lines .... خط‌تولید تطبیقی
Adaptation Engine .... موتور تطبیق
Adaptation Engine Evolution Cycle .... چرخهٔ تکامل موتور تطبیق
Adaptability .... قابلیت تطبیق
Adaptive .... تطبیق‌یاب
Adaptive Object Model .... مدل قابل تطبیق شیء
Adaptive Object Model Architecture Style .... سبک معماری مدل قابل تطبیق شیء
Adaptive Plug and Play Components (APPC) .... مؤلفه‌های تطبیقی اتصالی
Adaptive Programming .... برنامه‌نویسی تطبیقی
Adaptive Workflow Management System .... سیستم مدیریت جریان کار قابل تطبیق
Adaptive Workflow System .... سیستم جریان کار قابل تطبیق
Adaptivity .... تطبیق‌یابی
Analysis Patterns .... الگوهای تحلیل
Application .... برنامهٔ کاربردی
Application Object .... شیء برنامهٔ کاربردی
Architectural Patterns .... الگوهای معماری
Architectural Reflection .... انعکاس معماری
Architectural Structure .... ساختار معماری
Architectural View .... دید معماری
Architecturaly Insignificant .... غیر مهم در سطح معماری
Architecturaly Significant .... مهم در سطح معماری
Aspect Oriented Programming (AOP) .... برنامه‌نویسی جنبه‌گرا

انتساب .... Association
مشخصه (کلاس، شیء) .... Attribute
پالایش خودکار .... Automatic Refinement
در دسترس بودن .... Availability
سطح پایه .... Base Level
رفتار .... Behavior
انعکاس رفتاری .... Behavioral Reflection
چارچوب جعبه سیاه .... Blackbox Framework
استفادهٔ مجدد جعبه سیاه .... Blackbox Reuse
الگوی طراحی سازنده .... Builder Design Pattern
موجودیت حرفه .... Business Entity
منطق حرفه .... Business Logic
فرآیند حرفه .... Business Process
الگوهای حرفه .... Business Patterns
قوانین حرفه .... Business Rules
مرتبط علّی .... Causally-Connected
مدیریت تغییرات .... Change Management
قابلیت تغییر .... Changeability
درشت‌دانه .... Coarse-grained
تعامل .... Collaboration
مؤلفه .... Component
قابلیت اتصال مؤلفه‌های جدید .... Component Plugability
تولید نرم‌افزار بر مبنای مؤلفه‌ها .... Component-Based Software Development
سیستم مبتنی بر مؤلفه .... Component-Based System
ویژگی مرکب .... Composite Feature
ترکیب .... Composition
انعکاس محاسباتی .... Computational Reflection
کلاس واقعی .... Concrete Class
پیکربندی .... Configuration
برنامه‌نویسی مبتنی بر قرارداد .... Contract-Based Programming
جریان کنترل .... Control Flow
مناسب‌سازی (خط‌تولید) .... Customization
سیستم مدیریت پایگاه داده‌ها .... Database Management System
سیستم پایگاه داده‌ها .... Database System
نوع داده .... Data Type
برنامه‌نویسی تدافعی .... Defensive Programming
تعریف .... Definition
نمایندگی .... Delegation
الگوهای طراحی .... Design Patterns

استقرار .... Deployment
تولیدکننده .... Developer
توزیع .... Distribution
دامنه .... Domain
مهندسی دامنه .... Domain Engineering
متخصص دامنه .... Domain Expert
مستقل از دامنه .... Domain-Inependent
عمل مستقل از دامنه .... Domain-Independent Operation
عمل دامنه .... Domain Operation
زبان مخصوص دامنه .... Domain-Specific Language
پویا .... Dynamic
تغییر پویا .... Dynamic Change
کارایی کاربر .... Efficiency
محصورسازی .... Encapsulation
برنامه‌نویسی توسط کاربر نهایی .... End-User Programming
موجودیت .... Entity
عمل مشترک در موجودیت‌ها .... Entity Cross-Cutting Operation
محیط .... Environment
استثناء (در جریان کار) .... Exception
سلسله‌مراتب تکاملی (پویا) .... Evolutionary (Dynamic) Hierarchy
جبران خطا .... Failure Recovery
تحمل خطا .... Fault Tolerance
ویژگی .... Feature
برنامه‌نویسی خصوصیت‌گرا .... Feature Oriented Programming
ریزدانه .... Fine-grain
انعطاف‌پذیری .... Flexibility
کلید خارجی .... Foreign Key
نیازمندی‌های کارکردی .... Functional Requirements
کارکرد .... Functionality
روش‌های تولیدی .... Generative Methods
برنامه‌نویسی تولیدی .... Generative Programming
عمل عمومی .... Genreric Operarion
راهنما .... Guideline
حک .... Hardcode
سلسله‌مراتب .... Hierarchy
کاهش سلسله‌مراتب .... Hierarchy Reduction
سابقه .... History
هم‌ریخت .... Homomorphic
چرخهٔ حیات سیستم قابل تطبیق توسط انسان .... Human-Adaptable System Lifecycle

الگوهای پیاده‌سازی .... Idioms
کلاس پیاده‌سازی .... Implementation Class
حالت ناسازگار .... Inconsistent State
پنهان‌سازی اطلاعات .... Information Hiding
وراثت .... Inheritance
نمونه .... Instance
انتقال نمونه .... Instance Migration
محیط مجتمع تولید .... Integrated Development Endironment (IDE)
تعامل .... Interaction
واسط .... Interface
الگوی طراحی مفسّر .... Interpreter Design Pattern
خودنگری (انعکاس محاسباتی) .... Introspection
حالت نامعتبر .... Invalid State
ثابت .... Invariant
فراخوانی .... Invocation
سطح دانش .... Knowledge Level
لایه‌بندی .... Layering
مؤلفهٔ بزرگ—مقیاس .... Large-scale Components
انعکاس زبانی .... Linguistic Reflection
ارسال پیام .... Message Passing
متاداده .... Meta-Data
متالایه .... Meta-Layer
متامدل .... Meta-Model
قرارداد متا—شیء .... Meta-Object Protocol
متامدل‌سازی .... Metamodeling
عمل (کلاس، شیء) .... Method
میان‌افزار .... Middleware
سیستم مأموریت-بحرانی .... Mission-Critical System
عامل متحرک .... Mobile Agent
محاسبات سیار .... Mobile Computing
متحرک‌سازی مدل .... Model Animation
قابلیت تغییر .... Modifiability
واحدمندی .... Modularity
جداسازی چندبعدی دغدغه‌ها .... Multi-Dimensional Separation of Concerns
وراثت چندگانه .... Multiple Inheritance
سلسله‌مراتب ثابت (ایستا) .... Non-Evolutionary (Static) Hierarchy
نیازمندی‌های غیرکارکردی .... Non-Functional Requirements
انعکاس نامحدود .... Non-Restrictive Reflection
سیستم نمادگذاری .... Notation

مدل شیء .... Object Model
چارچوب شی‌گرا .... Object Oriented Framework
الگوی تحلیل مشاهده .... Observation Analysis Pattern
سیستم مبتنی بر مؤلفهٔ باز .... Open Component-Based System
باز بودن .... Openness
سیستم عامل .... Operating System
قابلیت اتصال اعمال جدید به سیستم .... Operation Plugability
سطح عملیاتی .... Operational Level
عمود بر هم .... Orthogonal
خنثی .... Override
الگو .... Pattern
جهان‌بینی .... Paradigms
پارامتری کردن .... Parametrization
کارایی .... Performance
ماندگاری .... Persistence
محاسبات فراگیر .... Pervasive Computing
وصل‌کن—اجراکن .... Plug and Play
اشاره‌گر .... Pointer
قابلیت حمل .... Portability
پس‌شرط .... Postcondition
پیش‌شرط .... Precondition
عمل ابتدایی .... Primitive Operarion
فرآیند .... Process
الگوهای فرآیند .... Process Patterns
چرخهٔ تکامل محصول .... Product Evolution Cycle
مهندسی محصول .... Product Engineering
خانواده‌ای از محصولات .... Product Family
ایجاد یک محصول از خانواده .... Product Instantiation
الگوی طراحی خصوصیت .... Property Design Pattern
ویژگی کیفی .... Quality Attribute
سلسله‌مراتب کاهش‌یافته .... Reduced Hierarchy
متغیر ارجاعی .... Reference Variable
پالایش .... Refinement
انعکاس .... Reflection
الگوی معماری انعکاس .... Reflection Architectural Pattern
انعکاسی .... Reflective
سیستم پایگاه داده‌های رابطه‌ای .... Relational Database System
قابلیت اطمینان .... Reliability
مدیریت نیازمندی‌ها .... Requirements Management

قابلیت استفادهٔ مجدد ... Reusability

عقب‌گرد ... Rollback

الگوی طراحی شیءقاعده ... Rule-Object Design Pattern

سیستم ایمنی-بحرانی ... Safety-Critical System

مقیاس‌پذیری ... Scalability

معانی (زبان) ... Semantics

سهام‌دار ... Stakeholder

ایستا ... Static

تغییر ایستا ... Static Change

امنیت ... Security

خود-تطبیق ... Self-Adaptive

خود-تطبیقی ... Self-Adaptivity

نمایش-از-خود ... Self-Representation

جداسازی دغدغه‌ها ... Separation of Concerns

ویژگی ساده قابل تطبیق ... Simple Adaptable Feature

ویژگی ساده ... Simple Feature

مؤلفهٔ کوچک—مقیاس ... Small-scale Components

معماری نرم‌افزار ... Software Architecture

تولید نرم‌افزار ... Software Development

تکامل نرم‌افزار ... Software Evolution

خط تولید نرم‌افزار ... Software Product-Lines

الگوی طراحی راهبرد ... Strategy Design Pattern

ساختار ... Structure

انعکاس ساختاری ... Structural Reflection

برنامه‌نویسی موضوع‌گرا ... Subject Oriented Programming

نحو (زبان) ... Syntax

فعالیت (در جریان کار) ... Task

قالب ... Template

قابلیت آزمون ... Testability

ویژگی کاملاً قابل تطبیق ... Totally Adaptable Feature

ویژگی کاملاً غیرقابل تطبیق ... Totally Not-Adaptable Feature

تبدیل ... Transformation

بررسی نوع ... Type Checking

مربع نوع ... Type Square

الگوی طراحی شیء نوع ... Type-Object Design Pattern

نمودار فعالیت UML ... UML Activity Diagram

نمودار کلاس UML ... UML Class Diagram

نمودار حالت UML ... UML State Chart

نمودار ترتیب UML ... UML Sequence Diagram

قابلیت درک .... Understandability
قابلیت استفاده .... Usability
قابلیت تطبیق با کاربر .... User Adaptivity
تمایلات کاربر .... User Prefrences
اعتبارسنجی .... Validation
وارسی .... Verification
ماشین مجازی .... Virtual Machine
زبان مدل‌سازی تصویری .... Visual Modeling Language
چارچوب جعبه سفید .... Whitebox Framework
استفادهٔ مجدد جعبه سفید .... Whitebox Reuse
جریان کار .... Workflow
سیستم مدیریت جریان کار .... Workflow Management System
سیستم جریان کار .... Workflow System

کتاب‌نامه

[1] Bass L., Clements P., and Kazman R. *Software Architecture in Practice.* Addison-Wesley, 2nd edition, 2003.

[2] Foote B. and Yoder J. Evolution, architecture, and metamorphosis. In et al. J. Vlissides, editor, *Pattern Language of Program Design 2.* Addison Wesley, 1996.

[3] Parnas D. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15 (12):1053–1058, 1972.

[4] Parnas D. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2 (1), 1976.

[5] Czarnecki K. and Eisenecker U. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[6] Rational Software Corporation. *The Rational Unified Process*, 2003.

[7] Oppermann R. *Adaptive User Support: Ergonomic Design of Manually and Automatically Adaptable Software.* Lawrence Erlbaum Associates, 1994.

[8] Oreizy P., Gorlick M., Taylor R., Heimbigner D., Johnson G., Medvidovic N., Quilici A., Rosenblum D., and Wolf A. An architecture-based approach to self-adaptive software. *EEE Intelligent Systems and their Application*, 14(3):54–62, 99.

[9] Capra L., Emmerich W., and Mascolo C. Middleware for mobile computing. In *Networking Tutorials (volume 2497 of Lecture Notes in Computer Science)*, pages 20–58. Springer, 2002.

[10] Johnson R. and Oakes J. The user-defined product framework. Available at http://st-www.cs.uiuc.edu/users/johnson/papers/udp/, 1998.

[11] Yoder J. and Johnson R. The adaptive object model architectural style. In *the Working IEEE/IFIP Conference on Software Architecture (WICSA3 '02)*, pages 3–27. Kluwer, August 2002.

[12] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., and Irwin J. Aspect-oriented programming. In *the European Conference on Object-Oriented Programming (ECOOP '97)*, pages 220–242. Springer-Verlag, 1997.

[13] Devanbu P., Balzer B., Batory D., Kiczales G., Launchbury J., Parnas D., and Tarr P. Modularity in the new millenium: A panel summary. In *the 25th international conference on Software engineering (ICSE '03)*, pages 723–725, 2003.

[14] Revault N., Blanc X., and Perrot J. On meta-modeling formalisms and rule-based model transforms. ECOOP 2000, International Workshop on Model Engineering, 2000.

[15] Cheng S., Garlan D., Schmerl B., Sousa J., Spitznagel B., Steenkiste P., and Hu N. Software architecture-based adaptation for pervasive systems. In *the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing (volume 2299 of Lecture Notes in Computer Science)*, pages 67–82. Springer, 2002.

[16] Fowler M. *Analysis Patterns, Reusable Object Models.* Addison-Wesley, 1997.

[17] Arsanjani A. Rule object 2001: A pattern language for adaptive and scalable business rule construction. In *the 8th Conference on the Pattern Languages of Programs*, 2001.

[18] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M. *Pattern-Oriented Software Architecture*, volume Volume 1: A System of Patterns. John Wiley & Sons Ltd, 1996.

[19] Zimmermann C. *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, 1996.

[20] Ortin F. and Cueva J. Non-restrictive computational reflection. *Computer Standards & Interfaces*, 25 (3)(3):241–251, 2003.

[21] Ortin F. and Cueva J. Building a completely adaptable reflective system. In ECOOP 2001, Workshop on Adaptive Object-Models and Metamodeling Techniques, 2001.

[22] van Gurp J., Bosch J., and Svahnberg M. On the notion of variability in software product lines. In *the Working IEEE/IFIP Conference on Software Architecture (WICSA '01)*, 2001.

[23] Johnson R. Dynamic object model. Available at http://st-www.cs.uiuc.edu/users/johnson/papers/dom/, 1998.

[24] Yoder J. and Razavi R. Metadata and adaptive object-models. In *ECOOP'2000 Workshop Reader (vol. 1964 of Lecture Notes in Computer Science)*. Springer Verlag, 2000.

[25] Yoder J., Balaguer F., and Johnson R. Architecture and design of adaptive object models. *SIGPLAN Notices*, 36 (12)(12):50–60, 2001.

[26] Yoder J., Foote B., Balaguer F., and Johnson R. Adaptive object models for implementing business rules. OOPSLA 2001, Third Workshop on Best-Practices for Business Rules: Design and Implementation, 2001.

[27] Revault N. and Yoder J. Adaptive object-models and metamodeling techniques. In *ECOOP 2001, Workshop Reader (Lecture Notes in Computer Science)*, pages 57–71. Springer-Verlag, 2001.

[28] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[29] Devos M. and Tilman M. A repository-based framework for evolutionary software development. MetaData Pattern Mining Workshop, Urbana, IL. Available at http://www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html, 1998.

[30] Dantas A. and Borba P. Adaptability aspects: An architectural pattern for structuring adaptive applications. Third Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'2003, 2003.

[31] Dantas A., Yoder J., Borba P., and Johnson R. Using aspects to make adaptive object-models adaptable. RAM-SE'04. ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution., 2004.

[32] van der Aalst W. and van Hee K. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[33] Bernstein A., Dellarocas C., and Klein M. Towards adaptive workflow systems - cscw-98 workshop report. *SGMOD-Record and SIGGROUP-Bulletin*, 1999.

[34] Han Y., Sheth A., and Bussler K. A taxonomy of adaptive workflow management. CSCW-98 Workshop Towards Adaptive Workflow Systems, 1998.

[35] Manolescu D. Workflow enactment with continuation and future objects. In *the 17th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Conference (OOPSLA '02)*, pages 40–51, 2002.

[36] Manolescu D. and Johnson R. Dynamic object model and adaptive workflow. OOPSLA '99, Workshop on Metadata and Active Object-Model Pattern Mining, 1999.

[37] Riehle D., Fraleigh S., Bucka-Lassen D., and Omorogbe N. The architecture of a uml virtual machine. In *the 16th ACM SIGPLAN Conference on Object-Oriented Program Systems, Languages and Applications (OOPSLA '01)*, pages 327–341, 2001.

[38] Object Management Group. Unified modeling language 2 specification: Infrastructure. Available at www.omg.org, 2003.

[39] Maes P. Concepts and experiments in computational reflection. In *Proceedings of the 2nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Conference (OOPSLA '87)*, pages 147–155, 1987.

[40] Capra L., Emmerich W., and Mascolo C. Reflective middleware solutions for context-aware application. In *Reflection 2001 (volume 2192 of Lecture Notes in Computer Science)*, pages 126–133. Springer, 2001.

[41] Cazzola W., Savigni A., Sosio A., and Tisato F. Architectural reflection: Concepts, design, and evaluation. Technical report, Università degli Studi di Milano, 1999.

[42] Eriksson H. and Penker M. *Business Modeling with UML: Business Patterns at Work*. OMG Press, 2000.

[43] Coplien J. and Schmidt D. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[44] Vlissides J., Coplien J., and Kerth N. *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[45] Martin R., Riehle D., and Buschmann F. *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

[46] Harrison N., Foote B., and Rohnert H. *Pattern Languages of Program Design 4*. Addison-Wesley, 1999.

[47] Bosch J. *Design & Use of Software Architecture: Adopting and Evolving a Product-Line Approach*. Addison Wesley, 2000.

[48] Batory D. Product-line architectures. Invited Presentation, Smalltalk and Java in Industry and Practical Training, Efurt, Germany., 1998.

[49] Bosch J., Molin P., Mattson M., and Bengtsson P. Object oriented frameworks - problems & experiences. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Object-Oriented Application Frameworks*. Wiley & Sons, 1998.

[50] Batory D. and O'Malley S. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 4(1):355–398, 1992.

[51] Bosch J. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Second International Conference on Software Product Lines (volume 2379 of Lecture Notes In Computer Science)*, pages 257–271, 2002.

[52] van Gurp J. and Bosch J. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Software Practice & Experience*, 31(3):277–300, 2001.

[53] Rumbaugh J., Jacobson I., and Booch G. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[54] van Deursen A., Klint P., and Visser J. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35 (6):26–36, 2000.

[55] Esser R. and Janneck J. A framework for defining domain-specific visual languages. In OOPSLA 2001, Workshop on Domain Specific Visual Languages, 2001.

[56] Harrison W. and Ossher H. Subject-oriented programming: a critique of pure objects. *ACM SIGPLAN Notices*, 28(10):411–428, 1993.

[57] Lieberherr K., Silva-Lepe I., and Xiao C. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, 1994.

[58] Mezini M. and Lieberherr K. Adaptive plug-and-play components for evolutionary software development. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*, volume 33 (10), pages 97–116, Vancouver, October 1998. ACM.

[59] Stroustrup B. *The C++ Programming Language.* AddisonWesley, 3rd edition, 1997.

[60] McConnell S. *Code Complete.* Microsoft Press, 2nd edition, 2004.

[61] Meyer B. *Object-Oriented Software Construction.* Prentice-Hall, 2nd edition, 1988.

Adaptive Architectures: An Approach for Behavior Dynamism

Abstract

In many environments, business logic is constantly changing. So, the information systems of these environments should be adapted to the new requirements quickly. Some of these systems need adaptability of business logic by human users. Adaptability is, in most cases, an architectural issue. Therfore, software architecutre should support this quality attribute. In object-oriented systems, business logic is divided to structure and behavior of business entities (objects). Object-oriented systems that allow users to adapt business logic, use adaptive object model architecture style. At the moment, there are shortcomings in support of behavioral adaptability in these systems. In other word, in such systems behavioral adaptability is resticted to very simple behavior. Also, there is no support for defining new operations. This thesis, proposes a method for support of adaptability by software architecture of human-adaptable systems. This method provides possibility of implementing generic operations that are applicable for future entity types and also dynamic definition of new operations. This approach, simulates introspection in reflective programming languages for implementing generic operations, and provides domain experts with the possibility of composing primitive operations to define new operations dynamically. Furthermore, in this approach it is possible to assign operations to entity types dynamically. In addition, this method supports inheritance of structure and behavior of entity types.

Keywords: *Software Adaptability, Adaptable Architecture, Reflective Architecture, Adaptive Object Model Architecture Style, and Behavioral Adaptability*

Sharif University of Technology

Computer Engineering Department

Master of Science
in
Software Engineering

Adaptive Architectures: An Approach for Behavior Dynamism

by

Salar Mesdaghinia

Under supervision of

Dr. Seyed Hassan Mirian Hossein-Abadi

Jan 2005