



دانشگاه تربیت دیر شهید رجایی
دانشکده مهندسی کامپیوتر

پایان نامه دوره کارشناسی مهندسی کامپیوتر

عنوان پژوهش:

طراحی و پیاده‌سازی چارچوب ذخیره و بازیابی اطلاعات با الگوها

دانشجو:

محسن الهی فرد

استاد راهنما:

دکتر محمد کلانتری

الله رب العالمين

چکیده

در دنیای امروز، مدیریت و ذخیره‌سازی داده‌ها اهمیت بسیاری دارد، بهویژه در سیستم‌های پایدار که نیازمند طراحی چارچوب‌هایی کارآمد و سازگار هستند. این پروژه به بررسی و طراحی یک چارچوب پایداری برای ذخیره‌سازی و بازیابی داده‌ها می‌پردازد که ضمن حفظ یکپارچگی و سازگاری داده‌ها، عملکرد و انعطاف‌پذیری بالایی را ارائه دهد. با بهره‌گیری از روش‌های نوین طراحی و به کارگیری الگوهای مناسب، چارچوبی ارائه شده است که می‌تواند مشکلات ناسازگاری داده‌ها را کاهش داده، کارایی سامانه‌های مدیریت داده را افزایش دهد و در محیط‌های نرم‌افزاری مختلف ارائه شود. همچنین، روش‌های بهینه‌سازی، مدیریت موثر تراکنش‌ها و همزمانی و مدیریت حافظه نهان نیز در آن لحاظ شده است تا عملکرد آن در پردازش داده‌ها بهبود یابد. اگرچه هدف از ارائه این چارچوب، رقابت با روش‌های سنتی مدیریت داده نبوده است، اما این چارچوب می‌تواند برای مدیریت داده‌ها در سیستم‌های نرم‌افزاری مقیاس‌پذیر و پیشرفته به کار رود.

كلمات کلیدی

چارچوب پایداری، بازیابی و ذخیره‌سازی داده‌ها، نگاشت اشیا به پایگاه داده، مدیریت تراکنش و همزمانی داده‌ها، مدیریت داده‌ها با الگوها

فهرست مطالب

۱	فصل ۱
۱	مقدمه
۱	۱-۱ مقدمه
۱	۱-۲ ضرورت پژوهش
۲	۱-۳ اهداف پژوهش
۲	۱-۴ روش و ساختار پژوهش
۳	فصل ۲
۳	پیشینه: طراحی چارچوب پایداری با الگوها
۳	۲-۱ مقدمه
۴	۲-۲ مشکل اشیای پایدار
۵	۲-۳ یک سرویس پایداری از چارچوب پایداری
۵	۲-۴ چارچوبها
۶	۲-۵ الزامات سرویس و چارچوب پایداری
۷	۲-۶ مفاهیم کلیدی
۸	۲-۷ الگو: نمایش اشیا به عنوان جداول
۸	۲-۸ پروفایل مدل‌سازی داده در UML
۹	۲-۹ الگو: شناسه شی
۱۰	۲-۱۰ دسترسی به سرویس پایداری با استفاده از یک Facade
۱۱	۲-۱۱ نگاشت اشیا: الگوی Database Broker یا Database Mapper
۱۴	۲-۱۲ طراحی چارچوب با الگوی Template Method
۱۵	۲-۱۳ مادی‌سازی با الگوی Template Method
۲۰	۲-۱۴ پیکربندی نگاشت‌گرها به کمک MapperFactory
۲۱	۲-۱۵ الگو: مدیریت حافظه نهان
۲۲	۲-۱۶ یکپارچه‌سازی و پنهان‌سازی دستورات SQL در یک کلاس
۲۳	۲-۱۷ حالت‌های تراکنشی و الگوی حالت
۲۷	۲-۱۸ طراحی یک تراکنش با الگوی دستور
۳۰	۲-۱۹ مادی‌سازی تنبیل با پروکسی مجازی
۳۲	۲-۲۰ چگونه روابط جداول را نمایش دهیم؟
۳۳	۲-۲۱ ابرکلاس شی پایدار و جداسازی دغدغه‌ها

۳۴ ۲-۲۲ مسائل حل نشده

۳۵

فصل ۳

۳۵

روش: پیاده‌سازی چارچوب پایداری با الگوهای

۳۵	۳-۱ مقدمه
۳۵	۳-۲ ساخت طرح‌واره و درج رکوردهای پایگاه داده مورد استفاده
۳۶	۳-۳ پیاده‌سازی چارچوب: بخش پایگاه داده
۴۱	۳-۴ پیاده‌سازی چارچوب: بخش مدل‌ها
۴۳	۳-۵ پیاده‌سازی چارچوب: بخش پایداری
۴۸	۳-۶ پیاده‌سازی چارچوب: بخش تراکنش‌ها
۴۹	۳-۷ پیاده‌سازی چارچوب: بخش اصلی

۵۱

فصل ۴

۵۱

جمع‌بندی

۵۱	۴-۱ مقدمه
۵۱	۴-۲ ارزیابی عملکرد چارچوب پیشنهادی: چالش‌ها و راهکارها
۵۲	۴-۳ نتیجه‌گیری

۵۳

مراجع

۵۴

واژه نامه انگلیسی به فارسی

فهرست شکل‌ها

۸	شکل ۲-۱: نگاشت اشیا و جداول
۹	شکل ۲-۲: مثال پروفایل مدل‌سازی داده در UML
۱۰	شکل ۲-۳: ارتباط اشیا و رکوردها به‌واسطهٔ شناسه‌های شی
۱۱	شکل ۲-۴: یک PersistenceFacade
۱۳	شکل ۲-۵: نگاشت‌گرهای پایگاه داده
۱۵	شکل ۲-۶: الگوی Template Method در یک چارچوب رابط کاربری گرافیکی
۱۶	شکل ۲-۷: Template Method برای اشیای نگاشت‌گر
۱۷	شکل ۲-۸: بازنویسی Hook Method
۱۸	شکل ۲-۹: بهینه‌سازی کد با استفادهٔ مجدد از Template Method
۱۹	شکل ۲-۱۰: ساختار برخی از بسته‌ها و کلاس‌های چارچوب
۲۰	شکل ۲-۱۱: متدهای محافظت‌شده در UML
۲۳	شکل ۲-۱۲: نمودار حالت شی پایدار
۲۴	شکل ۲-۱۳: ارثبری کلاس‌های اشیای پایدار از PersistentObject
۲۵	شکل ۲-۱۴: متدهای commit و rollback
۲۶	شکل ۲-۱۵: اعمال الگوی حالت
۲۹	شکل ۲-۱۶: دستورات عملیات پایگاه داده
۳۱	شکل ۲-۱۷: پروکسی مجازی با Manufacturer
۳۳	شکل ۲-۱۸: ابرکلاس شی پایدار
۳۶	شکل ۳-۱: بخشی از اسکریپت SQL پایگاه داده مورد استفاده جهت به کارگیری در میزکار MySQL
۳۷	شکل ۳-۲: پیاده‌سازی قسمت اتصال بخش پایگاه داده چارچوب
۳۹	شکل ۳-۳: پیاده‌سازی قسمت عملیات بخش پایگاه داده چارچوب
۴۰	شکل ۳-۴: پیاده‌سازی کلاس QueryBuilder چارچوب
۴۱	شکل ۳-۵: پیاده‌سازی کلاس PersistentObject چارچوب
۴۲	شکل ۳-۶: پیاده‌سازی کلاس‌های City و Country چارچوب
۴۳	شکل ۳-۷: پیاده‌سازی کلاس PersistenceFacade چارچوب
۴۵	شکل ۳-۸: پیاده‌سازی قسمت نمایشگر بخش پایداری چارچوب
۴۷	شکل ۳-۹: پیاده‌سازی قسمت حالت‌های بخش پایداری چارچوب
۴۸	شکل ۳-۱۰: پیاده‌سازی قسمت دستورات بخش تراکنش‌های چارچوب

شکل ۳-۱۱: پیاده‌سازی قسمت تراکنش بخش تراکنش‌های چارچوب ۴۹

شکل ۳-۱۲: پیاده‌سازی بخش اصلی چارچوب ۵۰

فصل ۱

مقدمه

۱-۱ مقدمه

در دنیای فناوری اطلاعات، مدیریت داده‌ها و حفظ پایداری آن‌ها از اهمیت ویژه‌ای برخوردار است. سیستم‌های پایدار ذخیره‌سازی اطلاعات، نقش اساسی در تسهیل توسعه و اجرای نرم‌افزارهای پیچیده ایفا می‌کنند. در این راستا، طراحی چارچوب‌های پایدار، نیازمند استفاده از الگوهای بهینه و فناوری‌های نوین است تا ضمن افزایش کارایی، از چالش‌های مرتبط با سازگاری داده‌ها جلوگیری شود. در این گزارش، به بررسی روش‌های طراحی یک چارچوب پایدار پرداخته شده است که بتواند ضمن حفظ سازگاری داده‌ها، فرایندهای ذخیره و بازیابی را بهینه‌سازی کند. فصل‌های بعدی به ارائه ساختار این چارچوب، مزايا، معایب، چالش‌ها و پیاده‌سازی آن با استفاده از فناوری‌های موجود می‌پردازند.

۱-۲ ضرورت پروژه

با افزایش حجم داده‌ها و پیچیدگی سامانه‌های اطلاعاتی، نیاز به یک چارچوب پایدار برای مدیریت داده‌ها بیش از پیش احساس می‌شود. چالش‌های اصلی در این زمینه شامل زمانبری فرایندهای ذخیره و بازیابی، نیاز به مدیریت تراکنش‌ها، کاهش خطاهای ناشی از ناسازگاری داده‌ها و افزایش کارایی سامانه‌های پایگاه داده

است. چارچوب پیشنهادی در این ، با ارائه راهکارهایی برای حل این مشکلات، می‌تواند به عنوان یک راهکار مؤثر در سیستم‌های نرمافزاری مورد استفاده قرار گیرد.

۱-۳ اهداف پروژه

این پروژه به دنبال طراحی یک چارچوب پایدار جهت ذخیره و بازیابی داده‌ها، استفاده از الگوهای طراحی مناسب برای عملکرد و انعطاف‌پذیری چارچوب، بررسی چالش‌های مرتبط با پیاده‌سازی سیستم‌های پایدار و مقایسه چارچوب پیشنهادی با روش‌های موجود از نظر کارایی و سازگاری است.

۱-۴ روش و ساختار پروژه

در این پروژه ضمن بررسی تحقیقات مرتبط با طراحی چارچوب‌های پایدار و الگوهای مورد استفاده و همچنین، مدل‌های مفهومی برای چارچوب پیشنهادی، به کمک پایتون، پیاده‌سازی این چارچوب انجام شده است. در فصل دوم به طراحی چارچوب و در فصل سوم به پیاده‌سازی چارچوب خواهیم پرداخت. این فصل، کلیات پروژه را معرفی می‌کند و مسیر را مشخص می‌سازد. در ادامه، الگوهای طراحی مرتبط با چارچوب پایدار بررسی خواهند شد.

فصل ۲

پیشینه: طراحی چارچوب پایداری با الگوها

۱-۲ مقدمه

برنامه NextGen^۱ مانند اکثر برنامه‌ها نیاز به ذخیره و بازیابی اطلاعات در یک مکانیزم ذخیره‌سازی پایدار، مانند پایگاه داده رابطه‌ای^۲ دارد [۱]. این فصل به طراحی یک چارچوب برای ذخیره‌سازی اشیای پایدار می‌پردازد.

به‌طور کلی، بهتر است به جای توسعه یک چارچوب پایداری از ابتدا، از یک محصول آماده استفاده کرد یا خریداری نمود. این چارچوب می‌تواند به صورت یک محصول مستقل یا به عنوان بخشی از مدیریت پایداری درون‌محفظه‌ای برای Entity Bean‌ها^۳ در فناوری‌هایی مانند EJB^۴ و سایر فناوری‌های جاوا مورد استفاده قرار گیرد. توسعه یک سرویس پایداری O-R در سطح صنعتی ممکن است سال‌ها زمان ببرد و شامل چالش‌های پیچیده‌ای باشد که نیاز به تخصص ویژه دارد. علاوه‌بر این، فناوری‌هایی مانند JDO^۵ راهکارهای جزئی برای این مسئله ارائه می‌دهند.

بنابراین، هدف این فصل ارائه یک چارچوب پایداری در سطح صنعتی یا جایگزینی برای فناوری‌هایی مانند JDO نیست، بلکه از این چارچوب به عنوان ابزاری برای توضیح طراحی چارچوب‌ها با استفاده از الگوها استفاده

^۱ یک برنامه نمونه که اغلب در متون آموزشی و پژوهشی برای نمایش مفاهیم طراحی نرم‌افزار و الگوهای معماری استفاده می‌شود.

^۲ Relational Database

^۳ روشی در EJB که در آن، چرخه حیات و عملیات پایداری Entity Bean‌ها به صورت خودکار توسط محفظه اجرایی EJB مدیریت می‌شود، بدون نیاز به پیاده‌سازی مستقیم عملیات پایگاه داده توسط توسعه‌دهنده.

^۴ Enterprise JavaBeans که یک معماری سرور محور در جاوا برای توسعه برنامه‌های سازمانی است و شامل کlassen‌های تحت سرور جهت مدیریت تراکشن‌ها، امنیت، هم‌روندی و پایداری داده‌ها است.

^۵ Java Data Objects یک API استاندارد در جاوا که برای مدیریت پایداری اشیا و ارتباط آن‌ها با منابع داده‌ای (مانند پایگاه داده‌های رابطه‌ای) استفاده می‌شود. JDO مستقل از فناوری خاصی بوده و می‌تواند در محیط‌های مختلف پیاده‌سازی شود.

می‌شود، زیرا این موضوع به عنوان یک مطالعهٔ موردی مناسب مطرح است. همچنین، این فصل نمونهٔ دیگری از استفاده از UML برای انتقال مفاهیم طراحی نرم‌افزار را ارائه می‌دهد.

این چارچوب صرفاً برای معرفی طراحی چارچوب‌ها ارائه شده است و نباید به عنوان یک روش پیشنهادی برای طراحی سرویس پایداری در سطح صنعتی تلقی شود.

۲-۲ مشکل اشیای پایدار

فرض کنید در برنامهٔ NextGen، داده‌های مربوط به ProductSpecification در یک پایگاه دادهٔ رابطه‌ای ذخیره شده‌اند. هنگام اجرای برنامه، این داده‌ها باید به حافظهٔ محلی بارگذاری شوند. اشیای پایدار به اشیایی گفته می‌شود که نیاز به ذخیره‌سازی پایدار دارند، مانند نمونه‌های ProductSpecification مکانیزم‌های ذخیره‌سازی و اشیای پایدار به صورت پایگاه داده‌ای شی‌گرا، پایگاه داده‌های رابطه‌ای و... هستند. در صورتی که از پایگاه داده شی‌گرا برای ذخیره و بازیابی اشیا استفاده شود، نیازی به سرویس‌های پایداری سفارشی یا شخص ثالث نخواهد بود. این ویژگی یکی از مزایای استفاده از این نوع پایگاه داده‌ها است. با توجه به گستردنگی استفاده از پایگاه داده‌های رابطه‌ای، اغلب به جای پایگاه داده‌های شی‌گرا از این روش استفاده می‌شود. اما در این صورت، مشکلاتی به دلیل عدم تطابق بین مدل داده‌ای رابطه‌ای^۱ و مدل شی‌گرا^۲ به وجود می‌آید که در ادامه بررسی خواهند شد. برای حل این مشکل، به یک سرویس نگاشت شی‌رابطه‌ای^۳ ویژه نیاز است. علاوه‌بر پایگاه داده‌های رابطه‌ای، گاهی ذخیره‌سازی اشیا در سایر فرمات‌ها و مکانیزم‌های ذخیره‌سازی مانند فایل‌های Flat^۴، ساختارهای XML^۵، فایل‌های PDB در سیستم‌عامل Palm^۶، پایگاه داده‌های سلسله‌مراتبی^۷ و... مطلوب است. همانند پایگاه داده‌های رابطه‌ای، در این موارد نیز عدم تطابق بین اشیا و این

^۱ Record-Oriented

^۲ Object-Oriented

^۳ O-R Mapping

^۴ فایل‌های متنی ساده که داده‌ها را بدون ساختار پیچیده ذخیره می‌کنند. داده‌ها معمولاً در قالب متن ساده (TXT) یا جداول جداسده با کاما (CSV) ذخیره می‌شوند، بدون اینکه شامل ساختار سلسله‌مراتبی یا روابط بین داده‌ها باشند. این نوع فایل‌ها در مقایسه با پایگاه داده‌های رابطه‌ای ساده‌تر، اما از نظر جست‌وجو و مدیریت داده‌ها، محدود‌تر هستند.

^۵ یک نوع فایل متنی ساختاریافته که داده‌ها را در قالب برچسب‌های (Tags) سلسله‌مراتبی ذخیره می‌کنند.

^۶ فایل‌هایی که اطلاعات را به صورت رکوردهای باینری ذخیره می‌کنند و برای برنامه‌های مختلف مانند کتاب‌های الکترونیکی، دفترچه‌های یادداشت و پایگاه داده‌ها کاربرد دارند.

^۷ نوعی ساختار ذخیره‌سازی درختی؛ در این مدل، هر رکورد والد می‌تواند چندین رکورد فرزند داشته باشد، اما هر فرزند فقط یک والد دارد.

فرمتهای غیرشی‌گرا وجود دارد و برای کار با این فرمتهای سرویس‌های ویژه‌ای مشابه سرویس نگاشت شی‌رابطه‌ای نیاز خواهد بود.

۲-۳ یک سرویس پایداری از چارچوب پایداری

یک چارچوب پایداری، مجموعه‌ای عمومی، قابل استفاده مجدد و توسعه‌پذیر از انواع داده‌ها است که قابلیت‌هایی را برای پشتیبانی از اشیای پایدار فراهم می‌کند. یک سرویس پایدار (یا زیرسیستم پایداری^۱) در واقع این خدمات را ارائه می‌دهد و با استفاده از یک چارچوب پایداری ایجاد می‌شود.

یک سرویس پایداری معمولاً برای کار با پایگاه داده رابطه‌ای طراحی می‌شود، که در این صورت به آن سرویس نگاشت شی‌رابطه‌ای نیز گفته می‌شود. معمولاً، یک سرویس پایداری باید اشیا را به رکوردها (یا سایر انواع داده‌های ساختاریافته مانند XML) تبدیل کرده و در پایگاه داده ذخیره کند و هنگام بازیابی داده‌ها از پایگاه داده، رکوردها را به اشیا تبدیل نماید.

از منظر معماری لایه‌ای در برنامه NextGen، سرویس پایداری به عنوان یک زیرسیستم در لایه خدمات فنی^۲ قرار می‌گیرد.

۲-۴ چارچوب‌ها

اگرچه ممکن است بیش از حد ساده‌سازی شود، اما چارچوب را می‌توان مجموعه‌ای قابل توسعه از اشیا برای انجام عملکردهای مرتبط دانست. نمونه بارز آن، چارچوب‌های رابط کاربری گرافیکی^۳ مثل AWT یا Swing در جاوا است.

ویژگی شاخص یک چارچوب این است که پیاده‌سازی هسته اصلی و عملکردهای ثابت را ارائه می‌دهد و در عین حال، مکانیزمی برای توسعه‌دهندگان فراهم می‌کند تا عملکردهای متغیر را در آن جای‌گذاری کرده یا آن‌ها را گسترش دهند.

به عنوان مثال، چارچوب رابط کاربری گرافیکی Swing در جاوا شامل کلاس‌ها و رابطه‌ای برای عملکردهای اصلی رابط کاربری گرافیکی است. توسعه‌دهندگان می‌توانند ابزارک‌های تخصصی را با ارثبری

¹ Persistence Subsystem

² Technical Services Layer

³ GUI Frameworks

⁴ Widgets

(زیرکلاس‌گیری)^۱ از کلاس‌های Swing و بازنویسی^۲ برخی متدها ایجاد کنند. همچنین، آن‌ها می‌توانند با استفاده از الگوی ناظر^۳، رفتار پاسخ به رویدادهای مختلف را برای ابزارک‌های از پیش تعریف شده (مانند JButton) تغییر دهند. این همان مفهوم چارچوب است.

به طور کلی، یک چارچوب، یک مجموعه منسجم از رابطه‌ها^۴ و کلاس‌های است که برای ارائه خدمات به بخش هسته‌ای و ثابت یک زیرسیستم منطقی با یکدیگر همکاری می‌کنند. چارچوب، شامل کلاس‌های انتزاعی^۵ و مشخص^۶ است که رابطه‌ای موردنیاز، نحوه تعامل اشیا و سایر قواعد را تعریف می‌کنند. چارچوب معمولاً (و نه همیشه) از کاربر چارچوب^۷ انتظار دارد که برای استفاده، سفارشی‌سازی و گسترش خدمات چارچوب، از کلاس‌های موجود زیرکلاس‌گیری کند. چارچوب دارای کلاس‌های انتزاعی است که می‌توانند هم شامل متدهای انتزاعی و هم متدهای مشخص باشند. چارچوب از اصل هالیوود پیروی می‌کند: «ما با شما تماس می‌گیریم، نه شما با ما».^۸ این بدان معناست که کلاس‌های تعریف شده توسط کاربر (مانند زیرکلاس‌های جدید) پیام‌هایی را از کلاس‌های از پیش تعریف شده چارچوب دریافت می‌کنند. این پیام‌ها معمولاً با پیاده‌سازی متدهای انتزاعی کلاس والد مدیریت می‌شوند. مثال چارچوب پایداری که در ادامه ارائه خواهد شد، این اصول را نشان می‌دهد. همچنین، چارچوب‌ها سطح بسیار بالایی از قابلیت استفاده مجدد^۹ را ارائه می‌دهند (بسیار بیشتر از کلاس‌های مستقل). بنابراین، اگر یک سازمان به افزایش قابلیت استفاده مجدد نرم‌افزارها علاقه‌مند باشد (و چه سازمانی نیست؟)، باید بر توسعه چارچوب‌ها تأکید کند.

۲-۵ الزامات سرویس و چارچوب پایداری

برای نرم‌افزار NextGen POS^{۱۰}، به یک سرویس پایداری نیاز داریم که با استفاده از یک چارچوب پایداری ساخته شود (که همچنین بتوان از آن برای ایجاد سایر سرویس‌های پایداری نیز استفاده کرد). این چارچوب

¹ Subclassing

² Overriding

³ Observer Pattern

⁴ Interfaces

⁵ Abstract

⁶ Concrete

⁷ Interface User

⁸ Hollywood Principle: "Don't call us, we'll call you."

⁹ Reusability

¹⁰ یک سامانه نقطه فروش (Point of Sale) نسل جدید که برای مدیریت عملیات فروش، پردازش تراکنش‌ها و ذخیره اطلاعات مربوط به خرید در محیط‌های تجاری استفاده می‌شود.

را PFW^۱ می‌نامیم. PFW یک چارچوب ساده‌شده است^۲. این چارچوب باید قابلیت‌های ذخیره و بازیابی اشیا در یک مکانیزم ذخیره پایدار و انجام عملیات commit و rollback روی تراکنش‌ها را فراهم کند. طراحی این چارچوب باید به‌گونه‌ای باشد که امکان توسعه و پشتیبانی از مکانیزم‌ها و فرمتهای ذخیره‌سازی مختلف، مانند پایگاه داده‌های رابطه‌ای، رکوردها در فایل‌های Flat، یا فایل‌های XML را فراهم کند.

۲-۶ مفاهیم کلیدی

برای بررسی بخش‌های بعدی به تعدادی مفهوم کلیدی نیاز است. در این گزارش، می‌خواهیم نگاشتی بین کلاس و مخزن پایدار^۳ آن (برای مثال، یک جدول در پایگاه داده) و همچنین بین ویژگی‌های شی و فیلد‌های یک رکورد وجود داشته باشد. به عبارت دیگر، باید یک نگاشت طرح‌واره^۴ ای بین این دو طرح‌واره تعریف شود.

برای ارتباط آسان بین رکوردها و اشیا و همچنین جلوگیری از ایجاد تکرارهای نامناسب، هر رکورد و شی باید دارای هویت و شناسه منحصر به‌فرد^۵ باشد.

یک نگاشت‌گر پایگاه داده از نوع ساختگی محض^۶ مسئول مادی‌سازی^۷ و غیرمادی‌سازی^۸ داده‌ها است. مادی‌سازی فرایند تبدیل یک نمایش غیرشی‌گرای داده‌ها (مثل‌آر رکوردها در یک مخزن پایدار) به اشیا است. غیرمادی‌سازی عملیات معکوس آن است که با نام غیرفعال‌سازی^۹ نیز شناخته می‌شود.

سرویس‌های پایداری برای افزایش عملکرد، اشیای مادی‌شده را در حافظه نهان^{۱۰} ذخیره می‌کنند. آگاهی از حالت اشیا در ارتباط با تراکنش جاری مفید است. برای مثال، تشخیص این که کدام اشیا تغییر یافته‌اند^{۱۱} کمک می‌کند تا مشخص شود آیا نیاز به ذخیره مجدد در مخزن پایدار دارند یا نه.

¹ Persistence Framework

² یک چارچوب پایداری صنعتی و کامل خارج از حوزه این مقدمه است.

³ Persistent Store

⁴ Schema Mapping

⁵ Object Identity and Identifier

⁶ Pure Fabrication به معنای عدم وجود در دنیای خارجی

⁷ Materialization

⁸ Dematerialization

⁹ Passivation

¹⁰ Cache

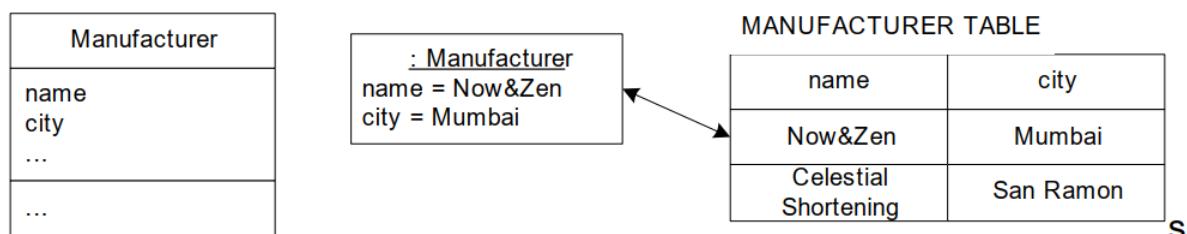
¹¹ Dirty Objects

همچنین، از مفهوم مادی‌سازی تنبل^۱ در این گزارش استفاده می‌شود؛ بدین معنا که همه اشیا به‌طور هم‌زمان مادی نمی‌شوند؛ بلکه یک نمونه خاص فقط در صورت نیاز و به‌صورت درخواستی مادی می‌شود. مادی‌سازی تنبل را می‌توان با استفاده از یک مرجع هوشمند به نام پروکسی مجازی^۲ پیاده‌سازی کرد.

۲-۷ الگو: نمایش اشیا به‌عنوان جداول

چگونه یک شی را به یک رکورد یا طرح‌واره پایگاه داده رابطه‌ای نگاشت کنیم؟ الگوی نمایش اشیا به‌عنوان جداول پیشنهاد می‌کند که برای هر کلاس شی پایدار^۳، یک جدول در پایگاه داده رابطه‌ای تعریف شود. ویژگی‌های شی که شامل نوع داده‌های ابتدایی (مانند عدد، رشته و...) هستند، به ستون‌های جدول نگاشت می‌شوند.

اگر یک شی فقط شامل ویژگی‌هایی از نوع داده‌های ابتدایی باشد، فرایند نگاشت ساده خواهد بود. اما همان‌طور که خواهیم دید، این مسئله همیشه به این سادگی نیست، زیرا اشیا ممکن است دارای ویژگی‌هایی باشند که به اشیای پیچیده‌تر ارجاع دهنند، در حالی که مدل رابطه‌ای نیازمند این است که مقادیر اتمی باشند (مطابق فرم 1NF). به شکل ۲-۱ توجه کنید.



شکل ۲-۲: نگاشت اشیا و جداول

۲-۸ پروفایل مدل‌سازی داده در UML

در ارتباط با پایگاه داده‌های رابطه‌ای، جای تعجب ندارد که UML به یک نشانه‌گذاری محبوب برای مدل‌های داده‌ای تبدیل شده است. لازم به ذکر است که یکی از مصنوعات رسمی در فرایند یکپارچه^۴، مدل داده است

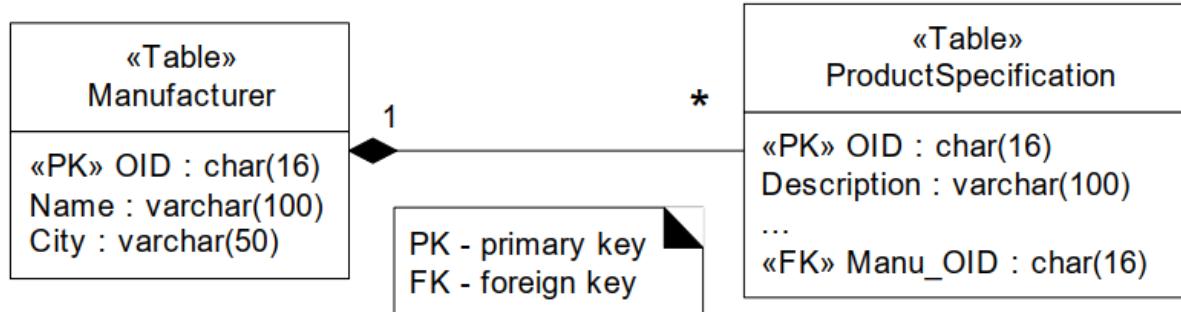
¹ Lazy Materialization

² Virtual Proxy

³ منظور از شی پایدار، شی‌ای است که می‌تواند از فرایند یا رشته‌ای که آن را ایجاد کرده است زنده بماند. یک شی پایدار وجود دارد تا زمانی که به صراحت حذف شود.

⁴ UP یا یک چارچوب مهندسی نرم‌افزار برای توسعه سیستم‌های شی‌گرا

که بخشی از رشته طراحی^۱ محسوب می‌شود. شکل ۲-۲ برخی از نشانه‌گذاری‌های UML را برای مدل‌سازی داده نشان می‌دهد. در این شکل، یک محدودیت ارجاعی^۲ را مشاهده می‌کنیم، به این معنا که یک ردیف ProductSpecification نمی‌تواند بدون یک ردیف مرتبط Manufacturer وجود داشته باشد.



شکل ۲-۲: مثال پروفایل مدل‌سازی داده در UML

این استریوتایپ‌ها^۳ بخشی از هسته UML نیستند، بلکه یک افزونه محسوب می‌شوند. به‌طور کلی، دارای مفهومی به نام پروفایل UML است: یک مجموعه منسجم از استریوتایپ‌های UML، مقادیر برچسب‌گذاری شده و محدودیت‌ها برای یک هدف خاص. برخی موارد رایج در پروفایل UML که مثالی از آن در شکل ۲-۲ مشاهده می‌شود به OMG^۴ برای تأیید رسمی ارائه شده بودند.

۲-۹ الگو: شناسه شی

داشتن روشی سازگار برای ارتباط بین اشیا و رکوردها مطلوب است و باید اطمینان حاصل شود که بازتولید مکرر یک رکورد منجر به ایجاد اشیای تکراری نشود. الگوی شناسه شی پیشنهاد می‌کند که یک شناسه شی (OID) به هر رکورد و شی (یا نماینده‌ای از یک شی) اختصاص داده شود.

یک OID معمولاً یک مقدار الفبایی عددی است که برای هر شی منحصر به فرد است. روش‌های مختلفی برای تولید شناسه‌های منحصر به فرد برای OID‌ها وجود دارد که از شناسه‌های منحصر به فرد در یک پایگاه داده خاص تا شناسه‌های منحصر به فرد در سطح جهانی را شامل می‌شود، مانند تولید کننده‌های توالی پایگاه داده و راهبرد تولید کلید High-Low.

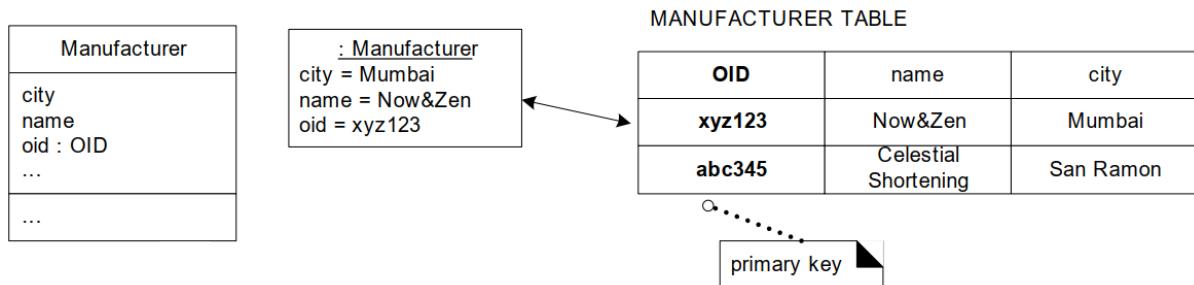
¹ Design Discipline

² Referential Constraint

³ Stereotypes

⁴ سازمان استاندارد سازی فعال در حوزه توسعه و حفظ استانداردهای مربوط به فناوری‌های شی‌گرا و معماری‌های نرم‌افزاری

در دنیای اشیا، یک OID توسط یک رابط یا کلاس OID نمایش داده می‌شود که مقدار واقعی و نحوه نمایش آن را در بر می‌گیرد. در پایگاه داده‌های رابطه‌ای، معمولاً به صورت یک مقدار متنی با طول ثابت ذخیره می‌شود. هر جدول یک OID به عنوان کلید اصلی دارد و هر شی نیز (به طور مستقیم یا غیرمستقیم) دارای یک OID خواهد بود. اگر هر شی با یک OID مرتبط باشد و هر جدول دارای کلید اصلی OID باشد، هر شی می‌تواند به طور منحصر به فرد به یک سطر در یک جدول خاص نگاشت شود. به شکل ۲-۳ نگاه کنید.



شکل ۲-۳: ارتباط اشیا و رکوردها به واسطه شناسه‌های شی

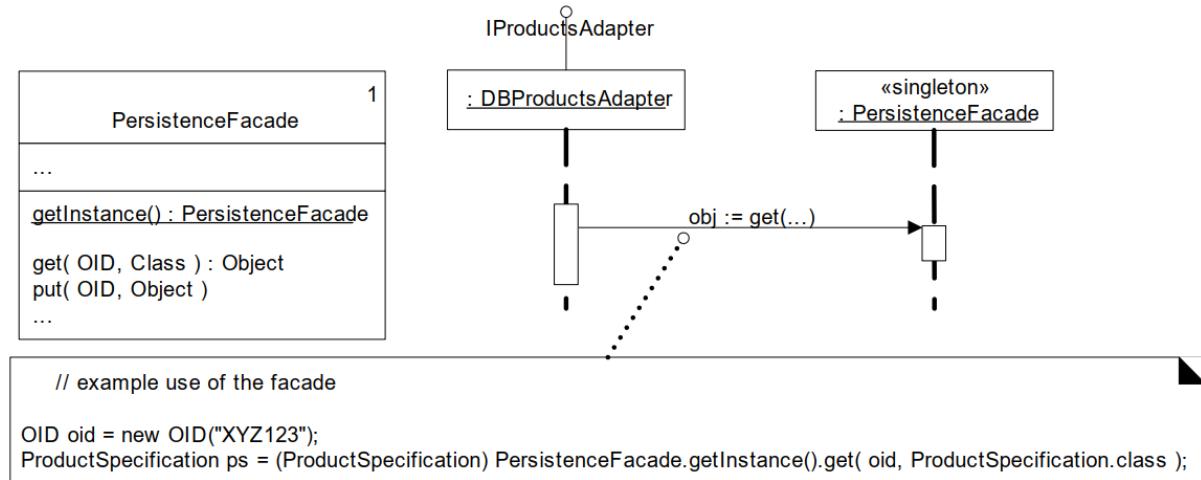
این یک نمای ساده‌شده از طراحی است. در واقع، ممکن است OID مستقیماً در شی پایدار قرار نگیرد، هرچند این کار امکان‌پذیر است. در عوض، ممکن است در یک شی پروکسی قرار داده شود که شی پایدار را در بر می‌گیرد. طراحی به انتخاب زبان برنامه‌نویسی بستگی دارد. همچنین، OID یک نوع کلید سازگار برای استفاده در رابط سرویس پایداری فراهم می‌کند.

۲-۱۰ دسترسی به سرویس پایداری با استفاده از یک Facade

گام اول در طراحی این زیرسیستم، تعریف یک Facade برای سرویس‌های آن است. می‌دانیم که یک الگوی رایج برای ارائه یک رابط یکپارچه به یک زیرسیستم است.

در ابتدا، عملیاتی برای بازیابی یک شی با استفاده از OID موردنیاز است. اما علاوه بر OID، زیرسیستم باید بداند که چه نوع شی‌ای را باید باز تولید کند؛ بنابراین، نوع کلاس نیز ارائه خواهد شد.

شکل ۲-۴ برخی از عملیات‌های Facade و نحوه استفاده از آن را در همکاری با یکی از آداتورهای سرویس نشان می‌دهد. در این شکل، PersistenceFacade به عنوان یک شی Singleton عمل کرده و NextGen عملیات ذخیره‌سازی و بازیابی اشیا را مدیریت می‌کند. متدهای PersistenceFacade getINSTANCE و getInstance یک نمونه از را باز می‌گردانند. متدهای get و getByName یک شناسه شی و نوع کلاس موردنظر، شی مربوطه را بازیابی می‌کنند. در این ساختار، DBProductsAdapter به عنوان یک آداتور برای تعامل با پایگاه داده عمل می‌کند. کدی که در این شکل قرار گرفته است نشان می‌دهد که با دریافت OID یک شی ProductSpecification از طریق PersistenceFacade بازیابی می‌شود.



شکل ۲-۴: یک PersistenceFacade

۲-۱۱ نگاشت اشیا: الگوی Database Broker یا Database Mapper

مانند همهی Facade، خود مستقیماً کار را انجام نمی‌دهد، بلکه درخواست‌ها را به اشیای زیرسیستم واگذار می‌کند. اما چه کسی مسئول مادی‌سازی و غیرمادی‌سازی اشیا از یک ذخیره‌ساز پایدار است؟

الگوی Information Expert پیشنهاد می‌کند که کلاس شی پایدار (ProductSpecification) خود یک نامزد مناسب است، زیرا شامل بخشی از داده‌هایی است که باید ذخیره شوند.

اگر یک کلاس شی پایدار کدی را برای ذخیره‌سازی خود در پایگاه داده تعریف کند، این روش طراحی نگاشت مستقیم^۱ نمی‌شود. این روش در صورتی عملی است که کدهای مرتبط با پایگاه داده بهصورت خودکار تولید شده و توسط یک کامپایلر پس‌پردازش به کلاس تزریق شوند، به‌گونه‌ای که توسعه‌دهنده نیازی به مشاهده یا نگهداری آن نداشته باشد.

اما اگر نگاشت مستقیم بهصورت دستی اضافه و نگهداری شود، معايب زیادی دارد و از نظر برنامه‌نویسی و نگهداری، بخوبی مقیاس‌پذیر نیست. این روش می‌تواند موجب اتصال^۲ قوی بین کلاس شی پایدار و دانش مریبوط به ذخیره‌سازی پایدار شود که اصل کاهش اتصال را نقض می‌کند. از طرفی، مسئولیت‌های پیچیده در

¹ Direct Mapping Design

² Coupling

یک حوزه نامرتبط نسبت به وظایف اصلی کلاس، که اصل انسجام^۱ بالا و جداسازی دغدغه‌ها^۲ را نقض می‌کند. در این حالت، مسائل مربوط به سرویس‌های فنی با منطق برنامه ترکیب می‌شوند. ما یک رویکرد کلاسیک نگاشت غیرمستقیم^۳ را بررسی خواهیم کرد که در آن از اشیای دیگری برای نگاشت اشیای پایدار استفاده می‌شود.

بخشی از این رویکرد، استفاده از الگوی Database Broker است. این الگو پیشنهاد می‌کند که یک کلاس خاص برای مادی‌سازی، غیرمادی‌سازی اشیا و ذخیره کردن آن‌ها در حافظه نهان مسئول باشد. این الگو همچنین با عنوان الگوی Database Mapper شناخته می‌شود، که نام بهتری نسبت به Database Broker محسوب می‌شود، زیرا مسئولیت آن را بهتر توصیف می‌کند. علاوه‌بر این، اصطلاح Database Broker در طراحی سیستم‌های توزیع‌شده معنای متفاوت و دیرینه‌ای دارد.^۴ برای هر کلاس شی پایدار، یک کلاس نگاشت‌گر جداگانه تعریف می‌شود. شکل ۲-۵ نشان می‌دهد که هر شی پایدار ممکن است کلاس نگاشت‌گر مخصوص به خود را داشته باشد و همچنین ممکن است نگاشت‌گرهای متفاوتی برای مکانیزم‌های ذخیره‌سازی مختلف وجود داشته باشد. قطعه‌ای از کد مرتبط با این طراحی در ادامه ارائه شده است.

```
class PersistenceFacade
{
// ...
public Object get(OID oid, Class persistenceClass)
{
// an IMapper is keyed by the Class of the persistent object IMapper
mapper = (IMapper) mappers.get(persistenceClass);
// delegate
return mapper.get(oid);
}
// ...
}
```

در شکل ۲-۵، یک ارتباط واجد شرایط^۵ می‌بینیم که به این معناست که PersistenceFacade و اشیای IMapper وجود دارد. با استفاده از یک کلید از نوع Class، یک IMapper پیدا می‌شود (مثلاً از طریق جست‌وجو در HashMap^۶). توجه کنید که در این نسخه از get، دیگر نیازی به ارسال Class به عنوان یک

¹ Cohesion

² Separation of Concerns

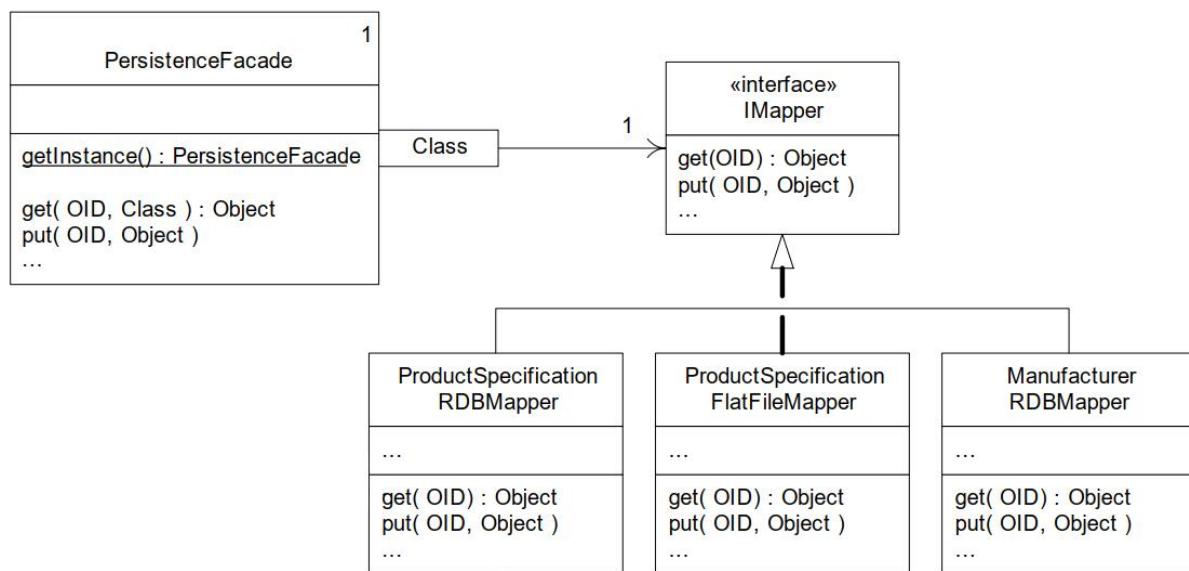
³ Indirect Mapping

⁴ در سیستم‌های توزیع‌شده، یک فرایند سرور Front-End Broker است که وظایف را به فرایندهای سرور Back-End واگذار می‌کند.

⁵ Qualified Association

⁶ در HashMap، جست‌وجو با استفاده از کلید انجام شده و به کمک تابع هش، مقدار مربوطه در زمان تقریباً ثابت بازیابی می‌شود.

پارامتر نیست، زیرا کلاس برای یک نوع پایدار خاص به صورت ثابت^۱ تعیین شده است. هر نگاشت‌گر اشیا را به روش منحصر به فرد خود دریافت (get) و ذخیره (put) می‌کند، بسته به نوع و فرمت ذخیره‌سازی داده‌ها.



شکل ۲-۵: نگاشت گرهای پایگاه داده

اگرچه این نمودار دو نگاشتگر ProductSpecification را نشان می‌دهد، اما تنها یکی از آن‌ها در سرویس پایداری در حال اجرا فعال خواهد بود.

رویکرد نگاشت‌گرهای مبتنی بر متاداده^۲ انعطاف‌پذیرتر اما پیچیده‌تر است. برخلاف طراحی دستی نگاشت‌گرهای جداگانه برای انواع مختلف اشیای پایدار، در این روش، نگاشت بین طرح‌واره شی و طرح‌واره دیگر (مثلاً رابطه‌ای) به صورت پویا^۳ و براساس متاداده‌ای که نگاشت را توصیف می‌کند، تولید می‌شود. به عنوان مثال: «جدول X به کلاس Y نگاشت می‌شود؛ ستون Z به ویژگی P در شی نگاشت می‌شود» (در عمل، این نگاشت بسیار پیچیده‌تر است). این روش در زبان‌هایی که از برنامه‌نویسی بازتابی^۴ پشتیبانی می‌کنند، مانند Java، C# و Smalltalk امکان‌پذیر است، اما در زبان‌هایی مانند C++ که فاقد این قابلیت هستند، دشوار خواهد بود.

1 Hardwired

2 Metadata

3 Dynamic

4 Reflective Programming

با استفاده از نگاشت‌گرهای مبتنی بر متاداده، می‌توان نگاشت طرح‌واره را بدون تغییر در کد منبع و تنها با تغییر در یک ذخیره‌ساز خارجی اعمال کرد. این امر، اصل تغییرات محافظت‌شده^۱ را در برابر تغییرات طرح‌واره تضمین می‌کند.

باین حال، یک ویژگی مهم در این چارچوب این است که می‌توان هم از نگاشت‌گرهای کدنویسی‌شده دستی و هم از نگاشت‌گرهای مبتنی بر متاداده استفاده کرد، بدون اینکه تغییری در کد کلاینت‌ها ایجاد شود (که این خود نوعی کپسوله‌سازی پیاده‌سازی^۲ محسوب می‌شود).

۲-۱۲ طراحی چارچوب با الگوی Template Method

بخش بعدی، برخی از ویژگی‌های طراحی Database Mapper‌ها را توضیح می‌دهد که جزء اصلی چارچوب پایداری هستند. این ویژگی‌های طراحی، مبتنی بر الگوی طراحی Template Method از GoF^۳ می‌باشند.^۴ این الگو در قلب طراحی چارچوب‌ها قرار دارد^۵ و اکثر برنامه‌نویسان شی‌گرا، حتی اگر نام آن را ندانند، به‌طور عملی با آن آشنا هستند.

ایده‌اصلی این الگو، تعریف یک Template Method در یک ابرکلاس^۶ است که اسکلت کلی یک الگوریتم را مشخص می‌کند، همراه با بخش‌های ثابت و متغیر آن. Template Method متدهای دیگری را فراخوانی می‌کند که برخی از آن‌ها را می‌توان در زیرکلاس‌ها^۷ بازنویسی کرد. به‌این ترتیب، زیرکلاس‌ها می‌توانند متدهای متغیر را بازنویسی کنند تا رفتار خاص خود را در نقاط تغییرپذیر الگوریتم اضافه نمایند (به شکل ۲-۶ نگاه کنید).

^۱ Protected Variations

^۲ Encapsulation of Implementation

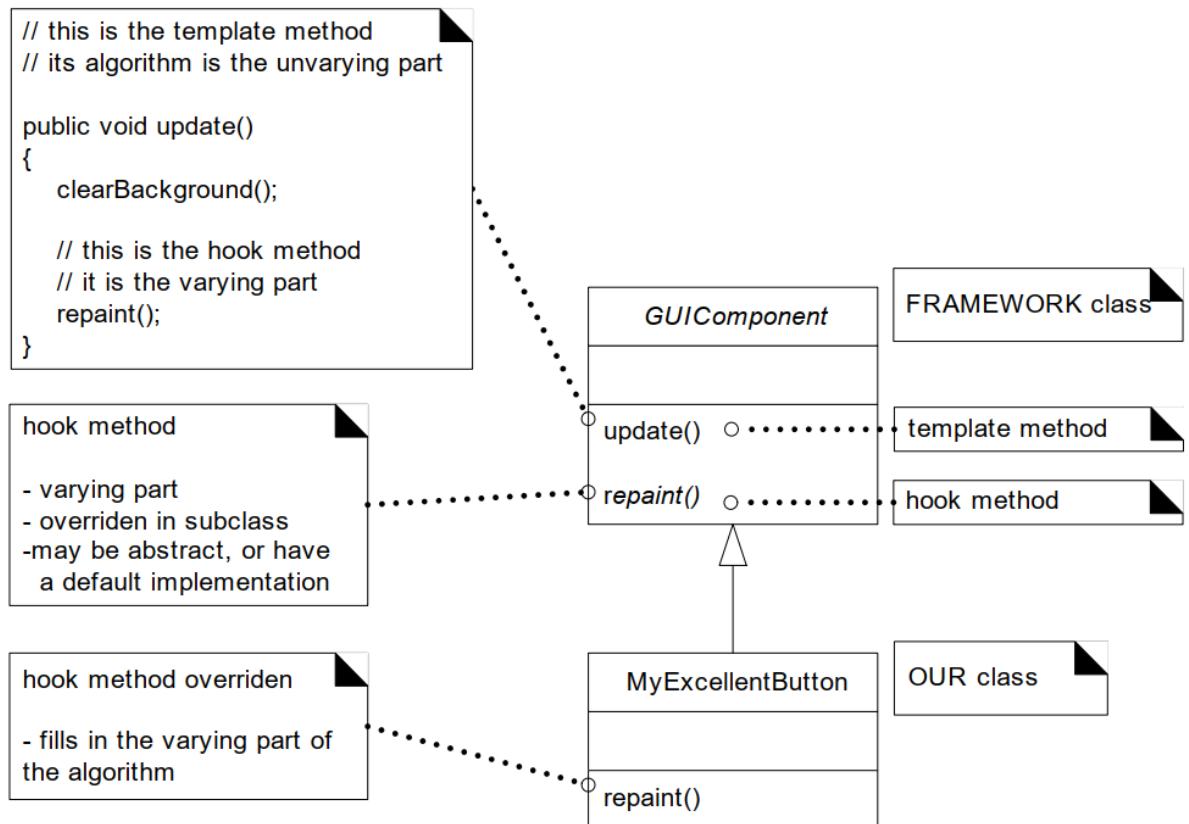
^۳ GoF (Gang of Four) به چهار نویسنده کتاب Design Patterns: Elements of Reusable Object-Oriented Software اشاره دارد که الگوهای طراحی را معرفی کردند.

^۴ این الگو هیچ ارتباطی با Template های C++ ندارد، بلکه قالب کلی یک الگوریتم را توصیف می‌کند.

^۵ به‌طور خاص‌تر، این الگو در چارچوب‌های Whitebox به کار می‌رود. این چارچوب‌ها معمولاً مبتنی بر سلسله‌مراتب کلاس‌ها و ارث‌بری هستند و کاربر باید تا حدی از طراحی و ساختار آن‌ها آگاه باشد؛ به همین دلیل به آن‌ها Whitebox گفته می‌شود.

^۶ Superclass

^۷ Subclass



شکل ۶-۲: الگوی Template Method در یک چارچوب رابط کاربری گرافیکی

در این شکل، توجه کنید که متد `repaint` در کلاس `MyExcellentButton` از متد `update` که از ابرکلاس به ارث برده شده است، فراخوانی می‌شود. این رویکرد در اتصال به یک کلاس چارچوبی معمول است.

۲-۱۳ مادی‌سازی با الگوی Template Method

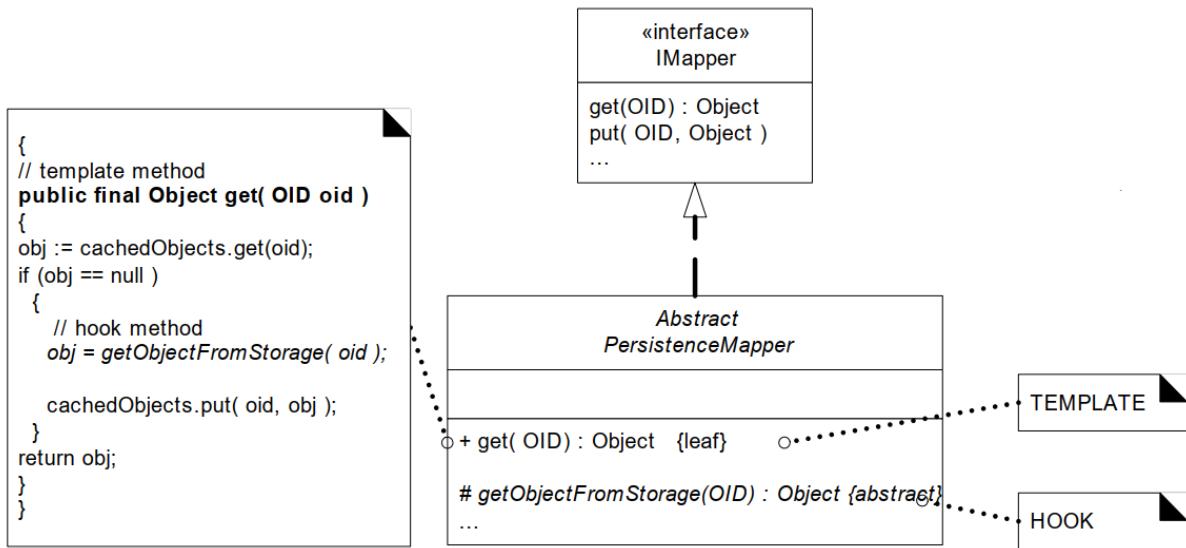
اگر بخواهیم دو یا سه کلاس نگاشت‌گر پیاده‌سازی کنیم، متوجه شباهت‌هایی در کد خواهیم شد. ساختار کلی الگوریتم تکراری برای مادی‌سازی یک شی به صورت زیر است:

```
if (object in cache)
    return it
else
    create the object from its representation in storage
    save object in cache
    return it
```

نقطه تغییرپذیر این الگوریتم در چگونگی ایجاد شی از داده‌های ذخیره شده است.

برای پیاده‌سازی این الگو، متد `get` را به عنوان `Template Method` در یک ابرکلاس انتزاعی (`AbstractPersistenceMapper`) تعریف می‌کنیم که اسکلت الگوریتم را مشخص کند. سپس، در

زیرکلاس‌ها، از یک Hook Method برای پیاده‌سازی بخش متغیر استفاده خواهیم کرد. شکل ۲-۷ طراحی اصلی این ساختار را نشان می‌دهد.



شکل ۲-۷: برای اشیای نگاشت‌گر

در شکل ۲-۷، نمادگذاری leaf استفاده شده است که برای عملیات و کلاس‌های نهایی^۱ یا برگ^۲ استفاده می‌شود. در نمادگذاری UML، # به معنای محافظت‌شده است؛ یعنی فقط برای زیرکلاس‌ها قابل مشاهده است.

همان‌طور که در این مثال نشان داده شده است، رایج است که Template Method عمومی^۳ باشد، در حالی که Hook Method به صورت محافظت‌شده تعریف شود. IMapper و AbstractPersistenceMapper بخشی از چارچوب پایداری هستند. حالا، یک برنامه‌نویس می‌تواند با افزودن یک زیرکلاس به این چارچوب متصل شده و متد getObjectFromStorage را بازنویسی یا پیاده‌سازی کند. شکل ۲-۸ یک نمونه از این رویکرد را نشان می‌دهد.

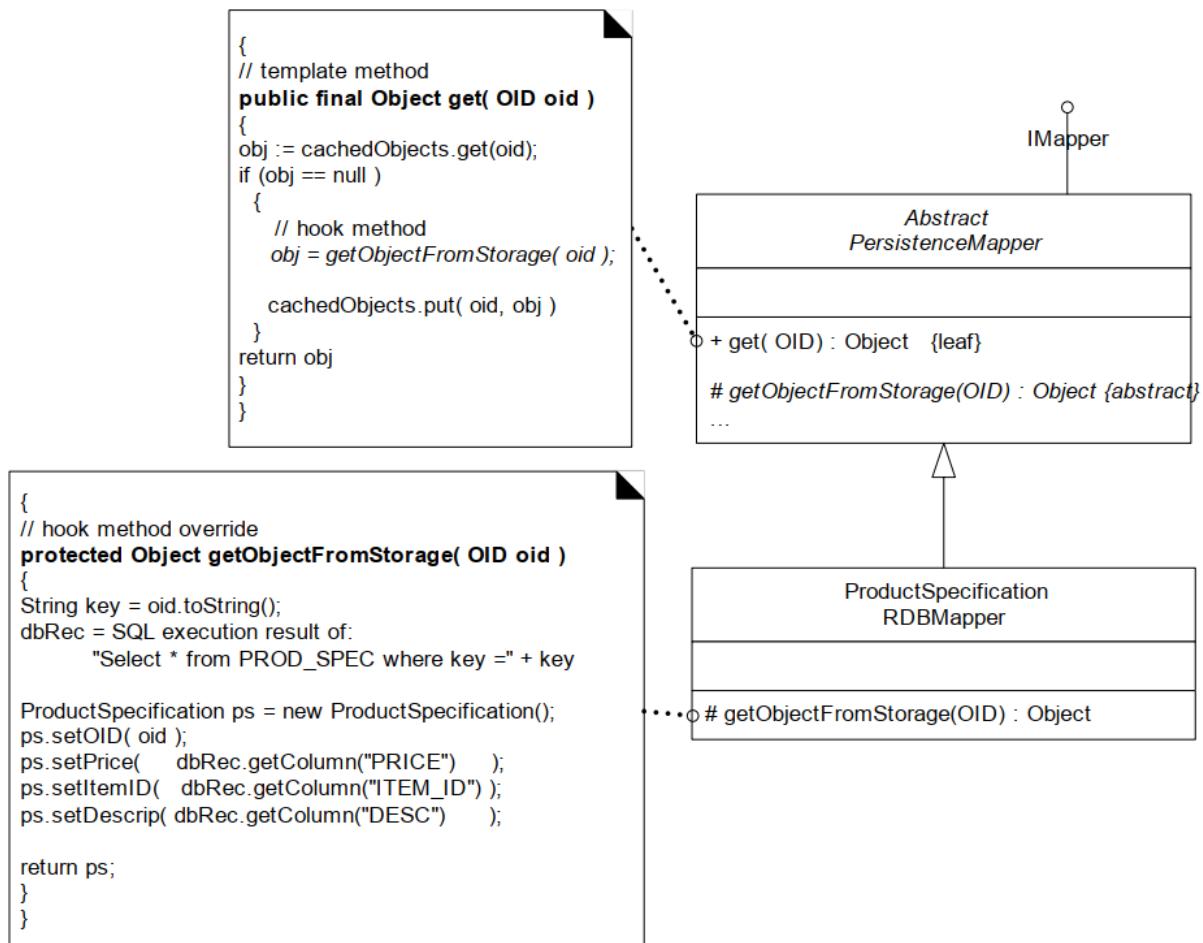
اگر در پیاده‌سازی Hook Method در شکل ۲-۸ فرض کنیم که بخش اولیه الگوریتم (اجرای دستور SELECT در SQL) برای همه‌ی اشیا یکسان است و تنها نام جدول پایگاه داده تغییر می‌کند، پس باز دیگر می‌توان از الگوی Template Method برای جداسازی بخش‌های متغیر و ثابت الگوریتم استفاده کرد. در شکل ۲-۹ نکته پیچیده این است که متد getObjectFromStorage در AbstractRDBMapper یک

¹ Final

² Leaf

³ Public

نسبت به متدهای `get` در `AbstractPersistenceMapper` است، اما همین متدها یک `Hook Method` نسبت به `getObjectFromRecord` جدید `Hook Method` `Template Method` محسوب می‌شود.



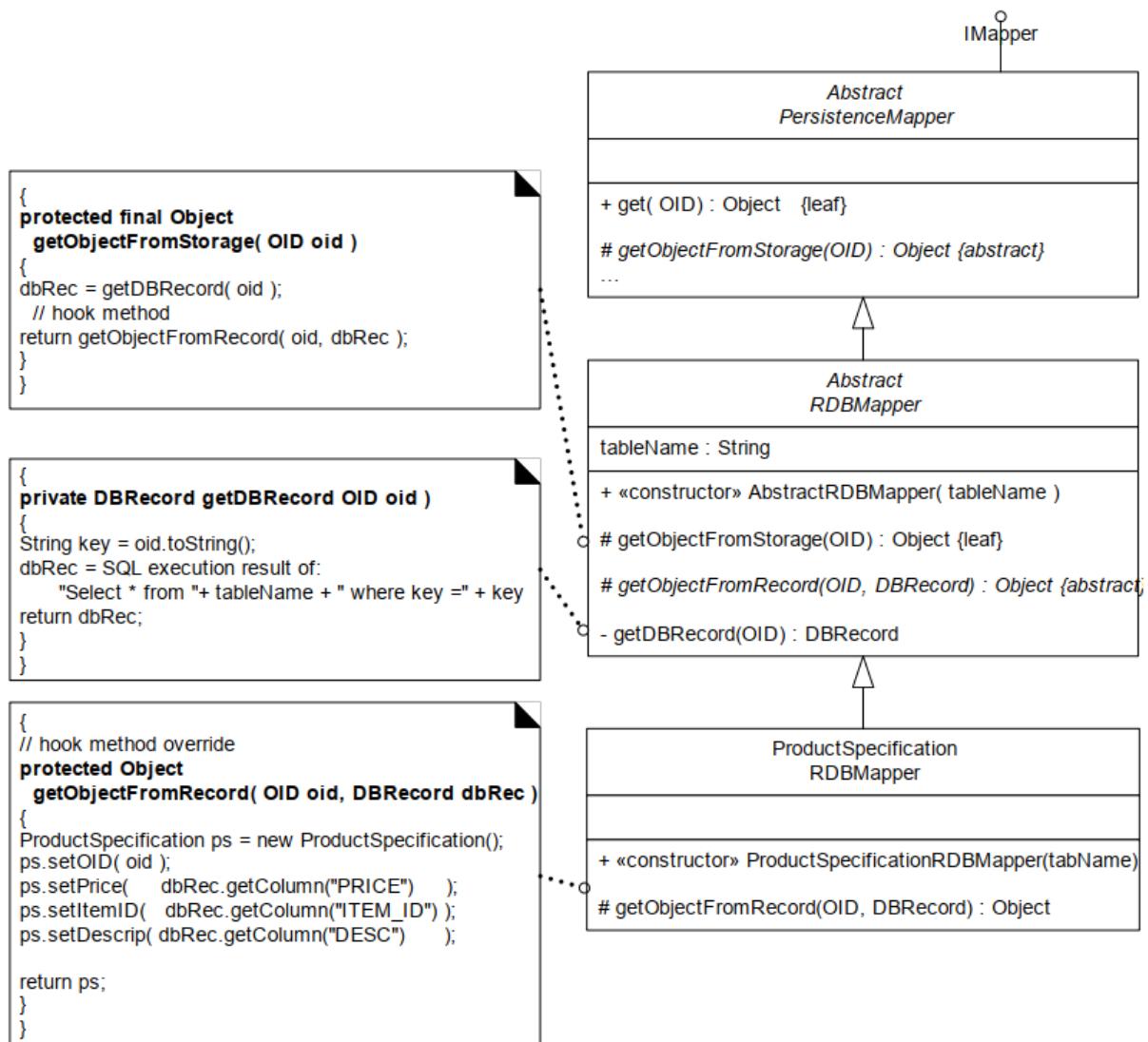
شکل ۲-۸: بازنویسی Hook Method

در شکل ۲-۹ توجه کنید که سازنده‌ها^۱ چگونه می‌توانند در UML تعریف شوند. استفاده از استریوتایپ اختیاری است و اگر نام سازنده با نام کلاس یکسان باشد، احتمالاً نیازی به آن نخواهد بود. اکنون، کلاس‌های `AbstractRDBMapper`, `AbstractPersistenceMapper` و `IMapper` بخشی از چارچوب هستند. برنامه‌های کاربردی تنها نیاز دارد که زیرکلاس خود (مانند `ProductSpecificationRDBMapper`) را اضافه کند و اطمینان حاصل نماید که این کلاس با نام جدول ایجاد می‌شود (تا از طریق زنجیره سازنده‌ها^۲ به `AbstractRDBMapper` ارسال شود).

¹ Constructors

² Constructor Chaining

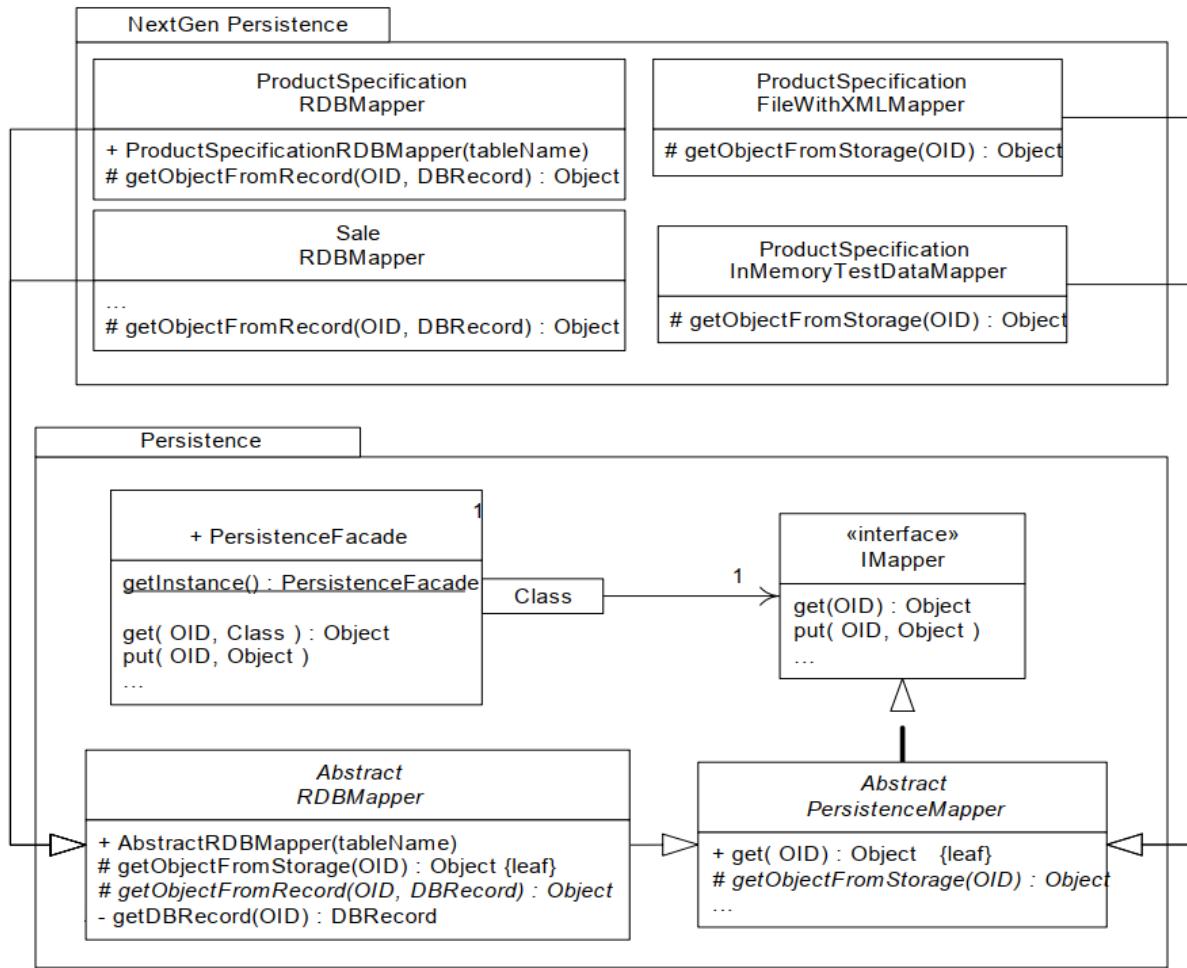
سلسله‌مراتب کلاس‌های Database Mapper یک بخش اساسی از چارچوب است. برنامه‌نویس می‌تواند زیرکلاس‌های جدیدی اضافه کند تا این چارچوب را برای مکانیزم‌های جدید ذخیره‌سازی پایدار^۱ یا برای جداول و فایل‌های خاص جدید در یک مکانیزم ذخیره‌سازی موجود، سفارشی‌سازی کند. شکل ۲-۱۰ ساختار برخی از بسته‌ها^۲ و کلاس‌ها را نشان می‌دهد. نکته قابل توجه این است که کلاس‌های خاص NextGen در بسته عمومی سرویس‌های فنی مربوط به پایداری قرار بگیرند. این نمودار، همراه با شکل ۲-۹، ارزش یک زبان بصری مانند UML را برای توصیف بخش‌های نرمافزار نشان می‌دهد، زیرا اطلاعات زیادی را به صورت خلاصه منتقل می‌کند.



شکل ۲-۹: بهینه‌سازی کد با استفاده مجدد از Template Method

¹ Persistence Storage Mechanisms

² Packages



شکل ۲-۱۰: ساختار برخی از بسته‌ها و کلاس‌های چارچوب

کلاس ProductSpecificationInMemoryTestDataMapper نمونه‌ای از نگاشت‌گرهای داده آزمایشی در حافظه است. این گونه کلاس‌ها برای تست سیستم بدون نیاز به پایگاه داده خارجی به کار می‌روند و اشیای از پیش تعیین شده^۱ را فراهم می‌کنند.

در فرایند یکپارچه، سند معماری نرم‌افزار^۲ به عنوان یک راهنمای یادگیری برای توسعه‌دهندگان آینده عمل می‌کند و دیدگاه‌های معماری کلیدی را ارائه می‌دهد. گنجاندن نموذارهایی مانند اشکال ۲-۹ و ۲-۱۰ در سند معماری نرم‌افزار پروژه NextGen، با هدف این سند که ارائه اطلاعات مهم معماری است، همسو می‌باشد. متدهای get در AbstractPersistenceMapper شامل بخشی از کد است که اینمی در برابر اجرای همزمان^۳ ندارد؛ زیرا ممکن است شی واحدی در چندین نخ به‌طور همزمان مادی‌سازی شود. از آنجا که سرویس

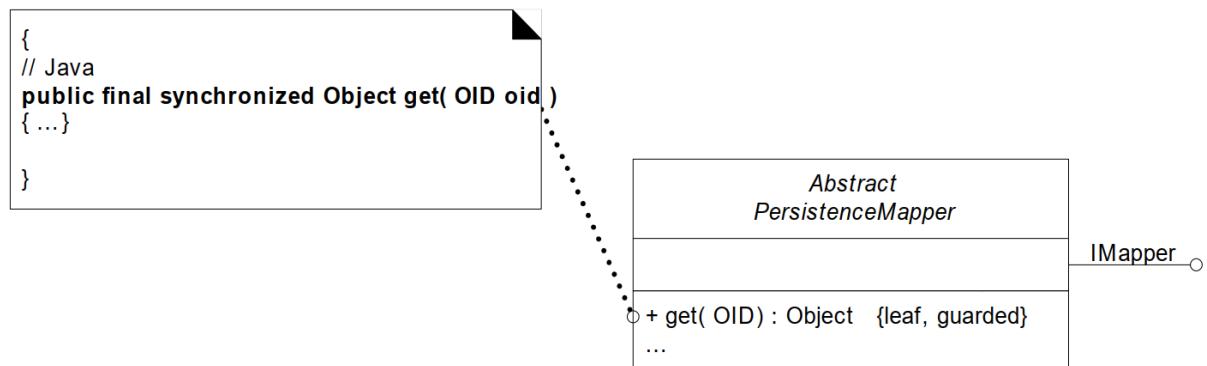
¹ Hard-Coded Objects

² SAD که یک مستند فنی توضیح‌دهنده دیدگاه‌های کلیدی معماری یک سیستم نرم‌افزاری است.

³ Thread Safety

پایداری به عنوان یک زیرسیستم خدمات فنی باید این در برابر اجرای همزمان طراحی شود، این موضوع اهمیت ویژه‌ای دارد. در واقع، کل زیرسیستم ممکن است در یک پردازش جداگانه روی یک کامپیوتر دیگر توزیع شود، به گونه‌ای که PersistenceFacade به یک شی سرور راه دور^۱ تبدیل شود و در آن چندین نخ به طور همزمان اجرا شده و به چندین مشتری سرویس دهند. بنابراین، این متد باید کنترل همزمانی^۲ داشته باشد. اگر از جاوا استفاده شود، اضافه کردن کلیدواژه synchronized می‌تواند این مشکل را حل کند.

شکل ۲-۱۱ در UML یک متد همگام‌سازی‌شده^۳ را نشان می‌دهد.



شکل ۲-۱۱: متدهای محافظت‌شده در UML

در این شکل، `guarded` به این معناست که متد محافظت‌شده است؛ یعنی در هر لحظه، فقط یک نخ می‌تواند در میان مجموعهٔ متدهای محافظت‌شده یک شی اجرا شود.

۲-۱۴ پیکربندی نگاشت‌گرها به کمک MapperFactory

مشابه نمونه‌های قبلی از Factory‌ها در این مطالعه موردی، پیکربندی PersistenceFacade با مجموعه‌ای از IMapper‌ها می‌تواند از طریق یک شی کارخانه‌ای (MapperFactory) انجام شود. با این حال، یک تغییر جزئی مطلوب این است که برای هر نگاشت‌گر یک متد جداگانه در Factory تعریف نشود. برای مثال، این روش مطلوب نیست:

```

class MapperFactory {
    public IMapper getProductSpecificationMapper() { ... }
    public IMapper getSaleMapper() { ... }
}
  
```

¹ Remote Server Object

² Thread Concurrency Control

³ Synchronized Method

زیرا این روش از اصل تغییرات محافظت شده در برابر افزایش تعداد نگاشت‌گرها پشتیبانی نمی‌کند (که این تعداد در آینده نیز افزایش خواهد یافت). در نتیجه، روش بهتری به‌شکل زیر پیشنهاد می‌شود:

```
class MapperFactory {  
    public Map getAllMappers() { ... }  
}
```

در این روش، کلاس `java.util.Map` (احتمالاً پیاده‌سازی شده با `HashMap`) به عنوان کلید، از کلاس مربوط به انواع پایدار استفاده می‌کند و مقدار هر کلید یک `IMapper` متناظر است.

سپس، `Facade` می‌تواند مجموعه `IMapper`‌های خود را به صورت زیر مقداردهی کند:

```
class PersistenceFacade {  
    private java.util.Map mappers = MapperFactory.getInstance().getAllMappers();  
}
```

می‌تواند با استفاده از یک طراحی داده‌محور^۱ مجموعه‌ای از `IMapper`‌ها را مقداردهی کند، به این معنا که می‌تواند با خواندن ویژگی‌های سیستم^۲، کلاس‌های `IMapper` موردنظر را شناسایی و نمونه‌سازی کند. اگر زبان از قابلیت‌های برنامه‌نویسی بازتابی پشتیبانی کند، می‌توان نام کلاس‌ها را به عنوان رشته خواند و از متدهای `Class.newInstance` برای نمونه‌سازی استفاده کرد. بنابراین، مجموعه نگاشت‌گرها می‌تواند بدون تغییر در کد منبع مجدداً پیکربندی شود.

۲-۱۵ الگو: مدیریت حافظه نهان

نگهداری اشیای مادی شده در یک حافظه نهان محلی^۳ به منظور بهبود عملکرد و پشتیبانی از عملیات مدیریت تراکنش (`commit`) مطلوب است، زیرا فرایند مادی‌سازی نسبتاً کند است.

الگوی مدیریت حافظه نهان پیشنهاد می‌کند که مسئولیت مدیریت حافظه نهان به‌عهده نگاشت‌گرهاشان باشد. در صورتی که برای هر کلاس از اشیای پایدار یک نگاشت‌گر جداگانه استفاده شود، هر نگاشت‌گر می‌تواند حافظه نهان مخصوص خود را مدیریت کند.

هنگامی که اشیا مادی‌سازی می‌شوند، در حافظه نهان قرار می‌گیرند و `OID` آن‌ها به عنوان کلید ثبت می‌شود. در نتیجه، در درخواست‌های بعدی، نگاشت‌گر ابتدا حافظه نهان را جست‌وجو می‌کند و در صورت یافتن شی، از مادی‌سازی مجدد و غیرضروری جلوگیری خواهد شد.

¹ Data-Driven Design

² System Properties

³ Local Cache

۲-۱۶ یکپارچه‌سازی و پنهان‌سازی دستورات SQL در یک کلاس

قرار دادن^۱ دستورات SQL در کلاس‌های مختلف نگاشت‌گر پایگاه داده رابطه‌ای اشکال بزرگی محسوب نمی‌شود، اما می‌توان آن را بهبود بخشد. فرض کنید بهجای آن، یک کلاس ساختگی محض واحد (و بهصورت SELECT SQL) به نام RDBOperations وجود داشته باشد که تمام عملیات (مانند INSERT...) در آن یکپارچه شده باشند. کلاس‌های نگاشت‌گر پایگاه داده رابطه‌ای برای دریافت یک رکورد یا مجموعه‌ای از رکوردهای پایگاه داده (مثلًا ResultSet) با این کلاس همکاری کنند. در این صورت، رابط آن به این شکل خواهد بود:

```
class RDBOperations {  
    public ResultSet getProductSpecificationData(OID oid) { ... }  
    public ResultSet getSaleData(OID oid) { ... }  
}
```

بهاین ترتیب، برای مثال، یک نگاشت‌گر می‌تواند کدی مشابه زیر داشته باشد:

```
class ProductSpecificationRDBMapper extends AbstractPersistenceMapper {  
    protected Object getObjectFromStorage(OID oid) {  
        ResultSet rs = RDBOperations.getInstance().getProductSpecificationData(oid);  
        ProductSpecification ps = new ProductSpecification();  
        ps.setPrice(rs.getDouble("PRICE"));  
        ps.setOID(oid);  
        return ps;  
    }  
}
```

در این روش، تمام عملیات پایگاه داده در یک کلاس مرکز شده و کلاس‌های مختلف می‌توانند بهصورت مازوچی و بدون کدنویسی مجدد از آن استفاده کنند.

در نتیجه این ساختگی محض به سهولت در نگهداری و بهینه‌سازی عملکرد توسط یک متخصص می‌رسیم. بهینه‌سازی SQL نیاز به فردی متخصص در این حوزه دارد (و نه یک برنامه‌نویس شی‌گرا). با مرکز کردن تمام دستورات SQL در یک کلاس، یافتن و بهینه‌سازی آن‌ها برای یک متخصص SQL بسیار آسان‌تر می‌شود. همچنین، با این کار، روش و جزئیات دسترسی کپسوله‌سازی می‌شود؛ برای مثال، بهجای قرار دادن SQL می‌توان آن را با یک Stored Procedure در پایگاه داده جایگزین کرد. همچنین، می‌توان از رویکردی پیشرفته‌تر مبتنی بر متاداده استفاده کرد که در آن SQL بهصورت پویا از یک توصیف متاداده خارجی تولید شود.

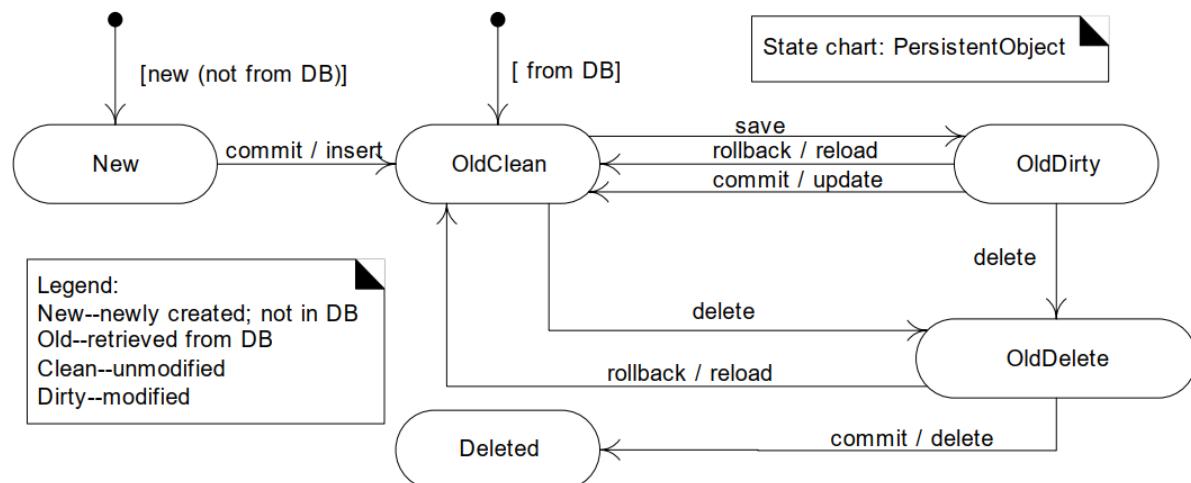
به عنوان یک معمار نرم‌افزار، نکته جالب این تصمیم طراحی این است که تحت تأثیر مهارت‌های توسعه‌دهنده قرار گرفته است. در اینجا، یک معامله بین انسجام بالا و راحتی برای یک متخصص انجام شده است. همه

^۱ Hard-coding

تصمیمات طراحی صرفاً به دلایل مهندسی نرم‌افزار مانند اتصال و انسجام گرفته نمی‌شوند، بلکه عوامل دیگری نیز در آن تأثیرگذار هستند.^۱

۲-۱۷ حالت‌های تراکنشی و الگوی حالت^۲

مسائل مربوط به پشتیبانی از تراکنش‌ها می‌توانند پیچیده شوند، اما برای ساده نگهداشتن موضوع و تمرکز بر الگوی حالت (از الگوهای GoF)، فرضیاتی را در نظر می‌گیریم. در این فرضیات، اشیای پایدار می‌توانند درج، حذف یا ویرایش شوند. انجام عملیات روی یک شی پایدار (مثلًاً ویرایش آن) بلافاصله باعث بهروزرسانی پایگاه داده نمی‌شود؛ بلکه این تغییرات تنها پس از اجرای یک عملیات commit صریح اعمال خواهند شد. علاوه‌بر این، پاسخ به یک عملیات وابسته به حالت تراکنشی شی است. به عنوان مثال، واکنش‌ها ممکن است مشابه حالاتی باشند که در نمودار حالت^۳ شکل ۲-۱۲ نشان داده شده است.



شکل ۲-۱۲: نمودار حالت شی پایدار

به عنوان مثال، یک شی old dirty (کهنه و تغییریافته) شی‌ای است که از پایگاه داده بازیابی شده و سپس تغییر کرده است. هنگام اجرای عملیات commit، این شی باید در پایگاه داده بهروزرسانی شود. در مقابل، شی‌ای که در حالت old clean (کهنه و بدون تغییر) قرار دارد، نباید هیچ عملی انجام دهد، زیرا تغییری نکرده است.

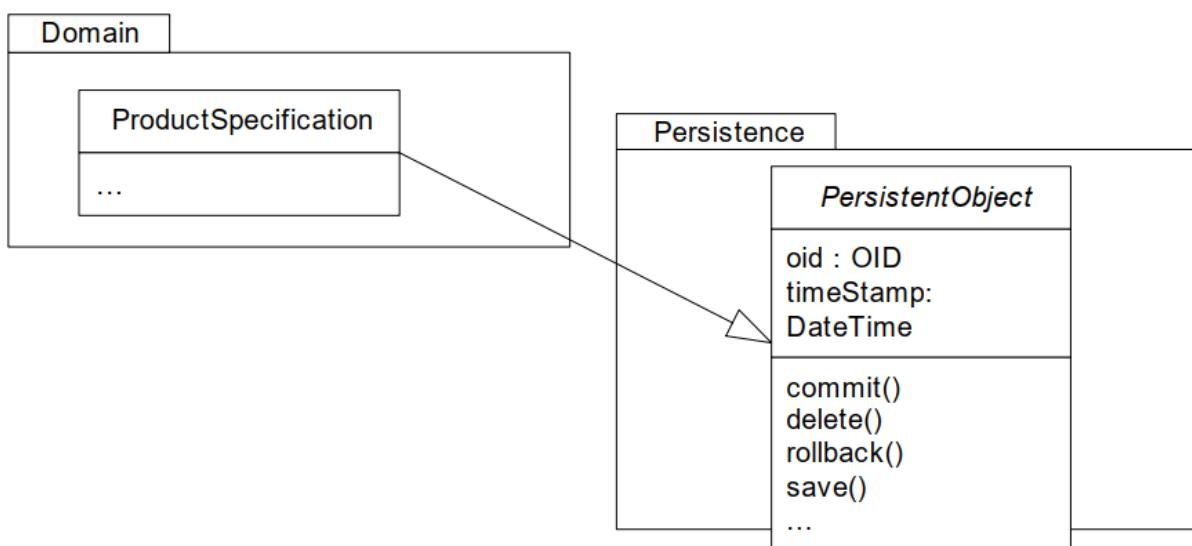
¹ State Pattern

² Statechart

در چارچوب شی‌گرا، وقتی عملیات حذف یا ذخیره انجام می‌شود، این عملیات بلافصله باعث حذف یا ذخیره rollback در پایگاه داده نمی‌شود؛ بلکه شی پایدار به حالت مناسب انتقال می‌یابد و منتظر اجرای commit می‌ماند تا واقعاً عملی انجام شود.

از نظر UML، این یک مثال عالی است که نشان می‌دهد چگونه یک نمودار حالت می‌تواند اطلاعاتی را که در قالب‌های دیگر بیان آن دشوار است، به‌طور خلاصه و مؤثر منتقل کند.

در این طراحی، فرض می‌کنیم که تمام کلاس‌های اشیای پایدار از یک ابرکلاس به نام PersistentObject ارثبری می‌کنند. این کلاس خدمات فنی مشترکی برای پایدارسازی ارائه می‌دهد.^۱ برای نمونه، شکل ۲-۱۳ را ببینید.



شکل ۲-۱۳: ارثبری کلاس‌های اشیای پایدار از PersistentObject

این همان مسئله‌ای است که با استفاده از الگوی حالت حل خواهد شد. توجه کنید که متدهای commit و rollback به ساختارهای مشابهی از منطق شرطی^۲ براساس کد حالت تراکنشی نیاز دارند. متدهای commit و rollback در هر حالت، عملیات متفاوتی انجام می‌دهند، اما ساختارهای منطقی مشابهی دارند.

^۱ برخی از چالش‌های مربوط به ارثبری از کلاس PersistentObject در ادامه مورد بحث قرار خواهند گرفت. به‌طور کلی، هر زمان که یک کلاس دامنه (Domain Object) از یک کلاس مربوط به خدمات فنی ارثبری کند، باید با دقت بررسی شود، زیرا این کار نگرانی‌های معماری (مانند جداسازی پایداری داده‌ها و منطق برنامه) را در هم می‌آمیزد.

² Case Logic

<pre> public void commit() { switch (state) { case OLD_DIRTY: ... break; case OLD_CLEAN: ... break; ... } </pre>	<pre> public void rollback() { switch (state) { case OLD_DIRTY: ... break; case OLD_CLEAN: ... break; ... } </pre>
--	--

شکل ۲-۱۴: متدهای rollback و commit

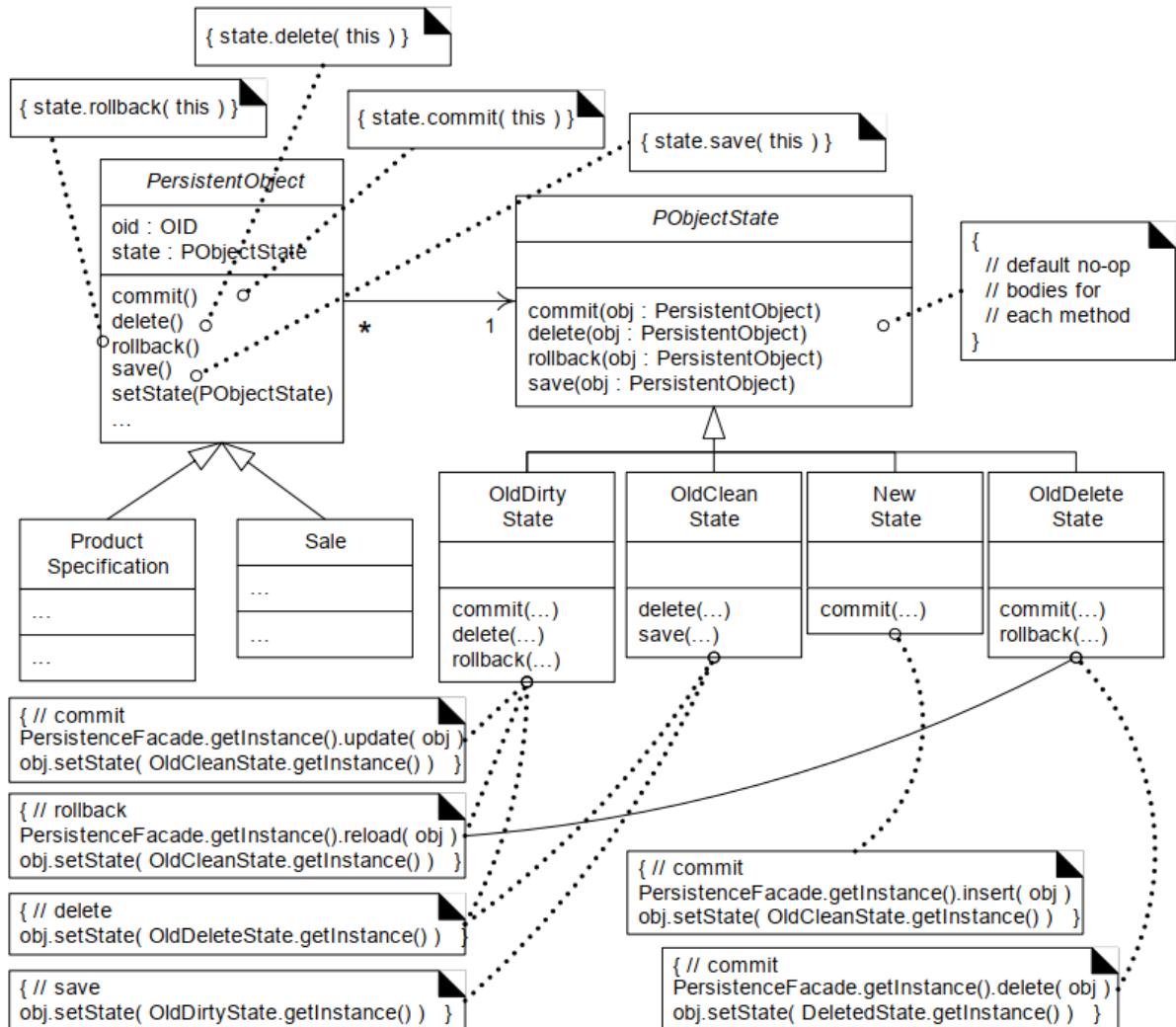
الگوی حالت، جایگزینی برای ساختار تکراری منطق شرطی است.

مشکل این است که رفتار یک شی وابسته به حالت آن است و متدهای آن شامل منطق شرطی برای انجام عملیات وابسته به حالت می‌باشند.

راه حل این است که برای هر حالت، یک کلاس مجزا ایجاد شود که یک رابط مشترک را پیاده‌سازی کند. عملیات وابسته به حالت از شی زمینه^۱ به شی حالت فعلی واگذار می‌شود. اطمینان حاصل می‌کنیم که شی زمینه همیشه به شی حالتی که حالت فعلی را منعکس می‌کند، اشاره دارد.

شکل ۲-۱۵ کاربرد این الگو را در زیرسیستم پایداری نشان می‌دهد.

¹ Context Object



شکل ۲-۱۵: اعمال الگوی حالت^۱

در این طراحی، متدهای وابسته به حالت در کلاس **PersistentObject** اجرای خود را به شی حالت مرتبط واگذار می‌کنند. اگر شی زمینه به **OldDirtyState** اشاره کند، اجرای متد `commit` باعث بهروزرسانی پایگاه داده می‌شود. سپس شی زمینه به **OldCleanState** تغییر حالت می‌دهد. در مقابل، اگر شی زمینه به **OldCleanState** اشاره کند، متد `commit` که به صورت ارثبری شده هیچ کاری انجام نمی‌دهد اجرا می‌شود. چراکه شی تغییری نکرده و نیازی به ذخیره‌سازی مجدد ندارد.

^۱ کلاس Deleted به دلیل محدودیت فضا در نمودار حذف شده است.

توجه کنید که در شکل ۲-۱۵، کلاس‌های حالت و رفتار آن‌ها مطابق با نمودار حالت در شکل ۲-۱۲ هستند. الگوی حالت یکی از مکانیزم‌های پیاده‌سازی یک مدل انتقال حالت^۱ در نرم‌افزار است^۲ و باعث می‌شود که یک شی در پاسخ به رویدادها بین حالات مختلف جابجا شود.

به عنوان یک نکته عملکردی، جالب است که این اشیای حالت در واقع بدون حالت^۳ هستند، یعنی هیچ ویژگی ندارند. بنابراین، نیازی به ایجاد چندین نمونه از یک کلاس حالت وجود ندارد؛ هر کلاس حالت می‌تواند ارجاع OldDirtyState باشد. به عنوان مثال، هزاران شی پایدار می‌توانند به یک نمونه واحد از Singleton دهند.

۲-۱۸ طراحی یک تراکنش با الگوی دستور^۴

در بخش قبلی، نگاهی ساده‌شده به مفهوم تراکنش‌ها داشتیم. در این بخش، بحث گسترش می‌یابد، اما تمامی مسائل مربوط به طراحی تراکنش را پوشش نمی‌دهد. به‌طور غیررسمی، تراکنش یک واحد کاری^۵ است، مجموعه‌ای از وظایف که باید همگی با موفقیت انجام شوند، یا هیچ‌کدام انجام نشوند. به عبارت دیگر، تراکنش باید اتمی^۶ باشد.

در چارچوب سرویس پایداری، وظایف یک تراکنش شامل درج، به‌روزرسانی و حذف اشیا می‌شود. به عنوان مثال، یک تراکنش می‌تواند شامل دو عملیات درج، یک عملیات به‌روزرسانی و سه عملیات حذف باشد. برای نمایش این مفهوم، یک کلاس Transaction اضافه می‌شود. ترتیب اجرای عملیات پایگاه داده در یک تراکنش می‌تواند بر موفقیت و عملکرد آن تأثیر بگذارد.

مثل‌اً فرض کنید پایگاه داده دارای یک محدودیت یکپارچگی ارجاعی باشد، به این صورت که هنگام به‌روزرسانی یک رکورد در جدول A که شامل یک کلید خارجی به جدول B است، پایگاه داده نیاز دارد که رکورد مربوطه در جدول B از قبل وجود داشته باشد.

^۱ State Transmission Model

^۲ راهکارهای دیگری نیز برای مدیریت وضعیت تراکنش‌ها وجود دارند، از جمله منطق شرطی قرار داده شده، مفسرهای ماشین حالت (State Machine Interpreters) و تولیدکننده‌های کد که توسط جداول حالت هدایت می‌شوند.

^۳ Stateless

^۴ Command

^۵ Unit of Work

^۶ Atomic

حال فرض کنید یک تراکنش شامل دو وظیفه^۱ باشد، عملیات درج برای افزودن یک رکورد به جدول B و عملیات بهروزرسانی برای بهروزرسانی رکوردهای در جدول A. اگر عملیات بهروزرسانی زودتر از درج اجرا شود، یک خطای یکپارچگی ارجاعی رخ خواهد داد.

راه حل این است که وظایف پایگاه داده را مرتب کنیم. برخی از مشکلات ترتیب اجرا وابسته به طرح واره پایگاه داده هستند، اما یک استراتژی کلی این است که ابتدا درجهای سپس بهروزرسانی ها و در نهایت، حذفها اجرا شوند.

با این حال، ترتیبی که یک برنامه وظایف را به تراکنش اضافه می‌کند، لزوماً بهترین ترتیب اجرا نیست. بنابراین، وظایف باید درست قبل از اجرا مرتب‌سازی شوند.

این مسئله ما را به سمت یک الگوی دیگر از GoF به نام الگوی دستور هدایت می‌کند. چگونه می‌توان درخواست‌ها یا وظایفی را مدیریت کرد که به قابلیت‌هایی مانند مرتب‌سازی (اولویت‌بندی)، صفت‌بندی، تأخیر در اجرا، ثبت گزارش^۲ یا لغو^۳ نیاز دارند؟

راه حل این است که هر وظیفه را به یک کلاس تبدیل کنیم که یک رابط مشترک را پیاده‌سازی می‌کند. این یک الگوی ساده اما بسیار کاربردی است؛ زیرا عملیات به اشیا تبدیل می‌شوند و در نتیجه قابل مرتب‌سازی، ثبت، صفت‌بندی و ... خواهند بود. به عنوان مثال، در چارچوب شی‌گرا برای پایداری، شکل ۲-۱۶ کلاس‌های دستورات مربوط به عملیات پایگاه داده را نشان می‌دهد.

اجرای کامل یک راهکار برای تراکنش‌ها شامل جزئیات بیشتری است، اما ایده اصلی این بخش این است که هر وظیفه یا عملیات در تراکنش را به عنوان یک شی با یک متدهای چند ریختی^۴ به نام execute نمایش دهیم. این رویکرد انعطاف‌پذیری زیادی ایجاد می‌کند، زیرا با تبدیل درخواست‌ها به اشیا، امكان مدیریت بهتر آن‌ها فراهم می‌شود.

یک نمونه کلاسیک از الگوی دستور، مدیریت عملیات در رابط کاربری گرافیکی مانند برش^۵ و چسباندن^۶ است؛ CutCommand، عملیات برش را انجام می‌دهد یا متدهای undo و execute در CutCommand. عملیات برش را بازگشت می‌دهد. CutCommand همچنین داده‌های لازم برای انجام عملیات بازگشت را

¹ Task

² Logging

³ Undoing

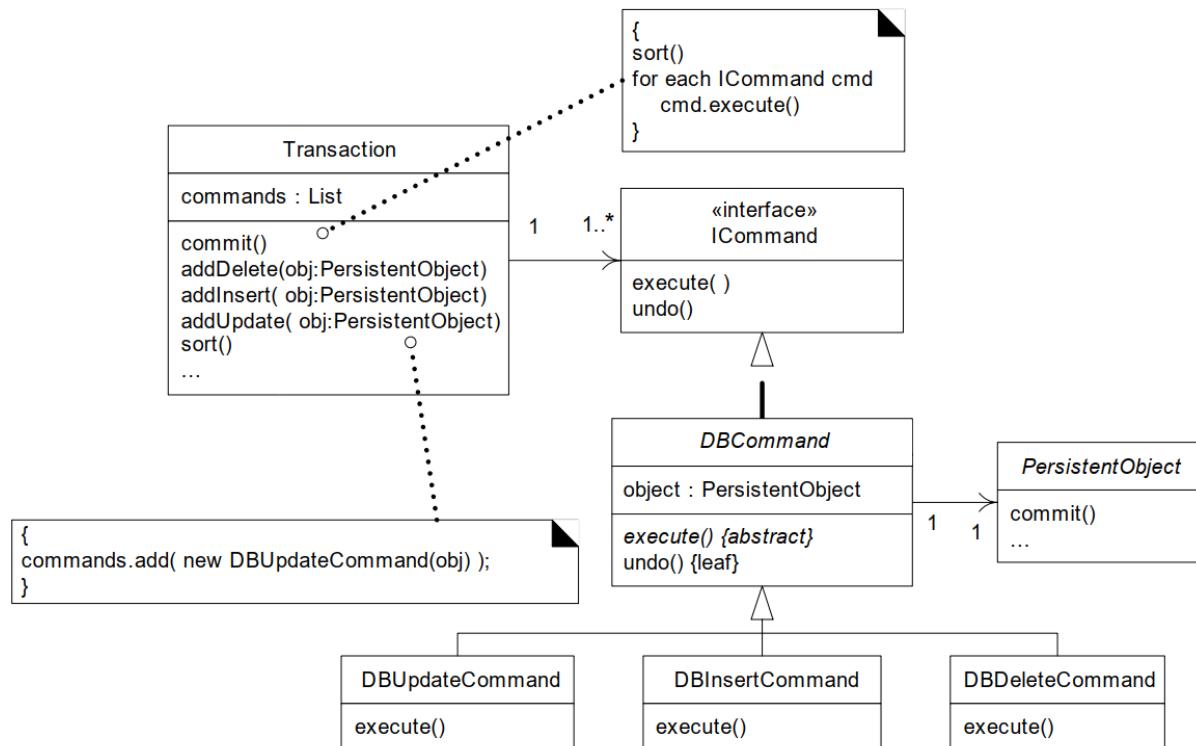
⁴ Polymorphic

⁵ Cut

⁶ Paste

حفظ می‌کند. تمامی فرمان‌های رابط کاربری گرافیکی در یک پشته تاریخچه^۱ ذخیره می‌شوند تا در صورت نیاز، یکی پس از دیگری از پشته خارج شده و لغو شوند.

یکی دیگر از کاربردهای رایج الگوی دستور، مدیریت درخواست‌های سمت سرور است. هنگامی که یک شی سرور یک پیام (درخواست راه دور) دریافت می‌کند، یک شی دستور برای آن درخواست ایجاد می‌شود. سپس این شی دستور به یک CommandProcessor تحويل داده می‌شود، که می‌تواند دستورات را صفت‌بندی، ثبت، اولویت‌بندی و اجرا کند.



شکل ۲-۱۶: دستورات عملیات پایگاه داده

در این مثال، **undo** هیچ عملی انجام نمی‌دهد (**no-op**)، اما یک راهکار پیچیده‌تر می‌تواند یک متدهای **commit** و **execute** را از زیرکلاس اضافه کند، که به طور منحصر به فرد می‌داند چگونه یک عملیات را لغو کند. در این مثال، از استراتژی مرتب‌سازی استفاده شده بود تا الگوریتم‌های مرتب‌سازی مختلف بتوانند ترتیب اجرای دستورات را مشخص کنند. در شکل ۲-۱۶، شاید **execute** به سادگی با فراخوانی **object.commit** انجام شود، اما هر شی دستور می‌تواند اقدامات منحصر به فرد خود را اجرا کند.

¹ History Stack

۲-۱۹ مادی‌سازی تنبیل با پروکسی مجازی

گاهی اوقات، به تعویق انداختن^۱ مادی‌سازی یک شی تا زمانی که واقعاً لازم باشد، مطلوب است، که معمولاً برای بهبود عملکرد انجام می‌شود. مثلاً فرض کنید اشیای ProductSpecification به یک شی Manufacturer ارجاع دارند، اما در بیشتر موارد نیازی به بازیابی Manufacturer از پایگاه داده وجود ندارد. تنها در سناریوهای خاصی مانند تخفیف‌های تولیدکننده^۲، اطلاعات سازنده مانند نام و آدرس موردنیاز خواهد بود.

به تعویق انداختن بازیابی اشیای فرزند تا زمانی که واقعاً موردنیاز باشند، مادی‌سازی تنبیل نامیده می‌شود. این مفهوم را می‌توان با استفاده از الگوی پروکسی مجازی (از الگوهای GoF) پیاده‌سازی کرد، که یکی از انواع مختلف الگوی پروکسی است.

یک پروکسی برای یک شی دیگر (شی واقعی) است که تنها در اولین ارجاع به آن، آن را از پایگاه داده بازیابی می‌کند. بنابراین، پروکسی، مکانیزم مادی‌سازی تنبیل را پیاده‌سازی می‌کند. پروکسی، یک شی سبک وزن^۳ است که به جای شی واقعی قرار می‌گیرد، تا زمانی که واقعاً لازم شود.

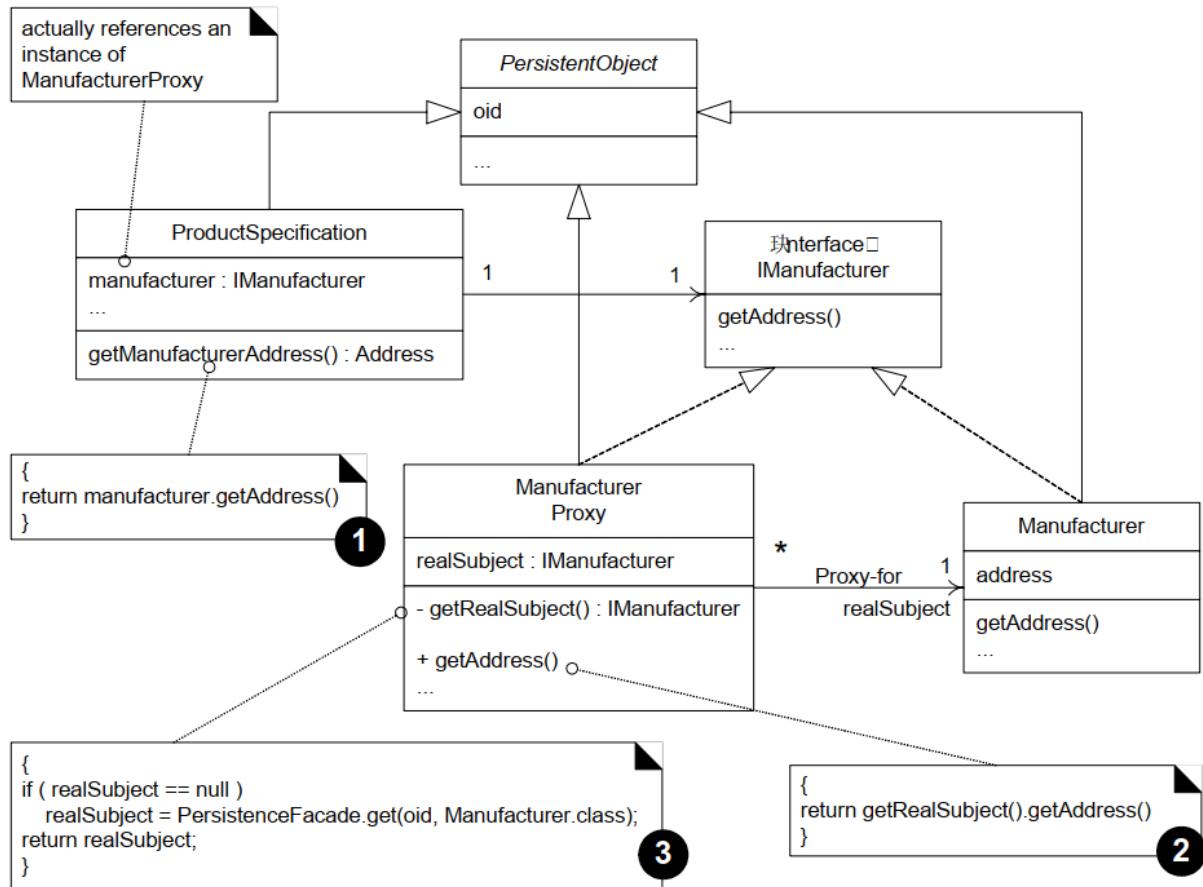
مثالی مشخص از الگوی پروکسی مجازی با ProductSpecification و Manufacturer در شکل ۲-۱۷ نشان داده شده است. این طراحی بر این فرض استوار است که پروکسی‌ها OID (شناسه شی) مربوط به موضوع واقعی خود را می‌دانند و هنگامی که به مادی‌سازی نیاز باشد، از OID برای شناسایی و بازیابی موضوع واقعی استفاده می‌شود.

توجه داشته باشید که ProductSpecification دسترسی به یک نمونه از IManufacturer دارد. ممکن است Manufacturer مربوط به این ProductSpecification هنوز در حافظه مادی‌سازی نشده باشد. هنگامی که ProductSpecification getAddress پیامی با نام ManufacturerProxy به ارسال می‌کند (گویی که یک شی Manufacturer واقعی است)، پروکسی Manufacturer واقعی را مادی‌سازی کرده و با استفاده از OID آن را بازیابی و در حافظه مادی‌سازی می‌کند.

¹ Deferring

² Manufacturer Rebate

³ Lightweight



شکل ۲-۱۷: پروکسی مجازی با Manufacturer

پیاده‌سازی پروکسی مجازی بسته به زبان برنامه‌نویسی متفاوت است. جزئیات این پیاده‌سازی خارج از محدوده این فصل است، اما به خلاصه‌ای از آن در ادامه می‌پردازیم. در C++, یک کلاس اشاره‌گر هوشمند^۱ به صورت template تعریف می‌شود. در این روش، نیازی به تعریف رابط IManufacturer نیست. در Java کلاس ManufacturerProxy پیاده‌سازی می‌شود و رابط IManufacturer تعریف می‌گردد. با این حال، این پیاده‌سازی معمولاً به صورت دستی انجام نمی‌شود. در عوض، یک تولیدکننده که ایجاد می‌شود که کلاس‌های مربوط به موضوع (مانند Manufacturer) را تحلیل کرده و سپس IManufacturer و Dynamic Proxy API را تولید می‌کند. یک جایگزین دیگر در جاوا استفاده از ProxyManufacturer برای تبدیل به موضوع واقعی استفاده می‌کند. در این روش نیز نیازی به تعریف IManufacturer نیست.

در شکل ۲-۱۷ مشاهده می‌شود که ManufacturerProxy برای مادی‌سازی موضوع واقعی خود با همکاری می‌کند. اما چه کسی PersistenceFacade را ایجاد می‌کند؟ پاسخ، کلاس

^۱ Smart Pointer

برای Database Mapper ProductSpecification کلاس نگاشت‌گر وظیفه دارد هنگام مادی‌سازی یک شی تصمیم بگیرد که کدام یک از اشیای فرزند آن باید به صورت مادی‌سازی پیش‌دستانه^۱ و کدام باید به صورت مادی‌سازی تنبیل با پروکسی بارگذاری شوند.

دو راهکار جایگزین مادی‌سازی پیش‌دستانه و مادی‌سازی تنبیل برای این موضوع وجود دارد.

مثالی از مادی‌سازی پیش‌دستانه به صورت زیر است:

```
class ProductSpecificationRDBMapper extends AbstractPersistenceMapper {
    protected Object getObjectFromStorage(OID oid) {
        ResultSet rs = RDBOperations.getInstance().getProductSpecificationData(oid);
        ProductSpecification ps = new ProductSpecification();
        ps.setPrice(rs.getDouble("PRICE"));
        ...
    }
    ...
}
```

در اینجا اصل موضوع آمده است:

```
String manufacturerForeignKey = rs.getString("MANU_OID");
OID manuOID = new OID(manufacturerForeignKey);
ps.setManufacturer((Manufacturer) PersistenceFacade.getInstance().get(manuOID, Manufacturer.class));
```

در اینجا، مادی‌سازی تنبیل به این صورت پیاده‌سازی می‌شود:

```
class ProductSpecificationRDBMapper extends AbstractPersistenceMapper {
    protected Object getObjectFromStorage(OID oid) {
        ResultSet rs = RDBOperations.getInstance().getProductSpecificationData(oid);
        ProductSpecification ps = new ProductSpecification();
        ps.setPrice(rs.getDouble("PRICE"));
        String manufacturerForeignKey = rs.getString("MANU_OID");
        OID manuOID = new OID(manufacturerForeignKey);
        ps.setManufacturer(new ManufacturerProxy(manuOID));
        return ps;
    }
}
```

۲-۲۰ چگونه روابط جداول را نمایش دهیم؟

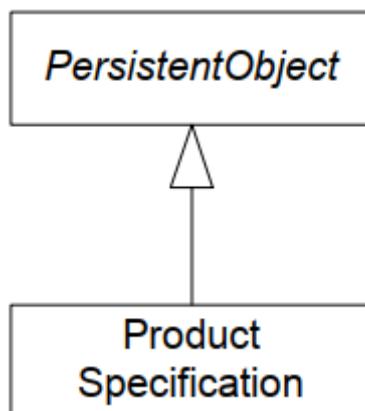
کد در بخش قبلی به یک کلید خارجی به نام PRODUCT_SPEC در جدول MANU_OID متکی است تا به رکوردی در جدول Manufacturer پیوند بخورد. این موضوع سؤالی را بر جسته می‌کند: چگونه روابط اشیا در مدل رابطه‌ای نمایش داده می‌شوند؟ پاسخ این سوال در الگوی نمایش روابط اشیا به صورت جداول ارائه شده است که پیشنهاد می‌کند: برای ارتباطات یک‌به‌یک، یک کلید خارجی OID در یکی یا هر دو جدولی که اشیا در رابطه را نمایش می‌دهند قرار دهیم، یا یک جدول واسط ایجاد کنیم که OID‌های هر شی در

^۱ Eager Materialization

رابطه را ثبت کند. برای ارتباطات یک به چند، مانند یک مجموعه، یک جدول واسط که OID‌های هر شی در رابطه را ثبت می‌کند ایجاد کنیم. برای ارتباطات چند به چند نیز همین کار را انجام دهیم. این روش‌ها به ما امکان می‌دهند تا روابط بین اشیا را در مدل رابطه‌ای به‌طور مؤثر نمایش دهیم.

۲-۲۱ ابرکلاس شی پایدار و جداسازی دغدغه‌ها

در طراحی نرم‌افزار، یک راه حل متداول برای ارائه قابلیت پایداری در اشیا، ایجاد یک ابرکلاس خدمات فنی انتزاعی به نام PersistentObject است که تمام اشیای پایدار از آن ارثبری می‌کنند (شکل ۲-۱۸ را ببینید). این کلاس معمولاً ویژگی‌هایی مانند OID و متدهایی برای ذخیره‌سازی در پایگاه داده را تعریف می‌کند. اگرچه این روش نادرست نیست، اما به دلیل اتصال کلاس‌های دامنه به کلاس PersistentObject، دغدغه‌هایی را به همراه دارد؛ به‌طوری‌که کلاس‌های دامنه به یک کلاس خدمات فنی وابسته می‌شوند.



شکل ۲-۱۸: ابرکلاس شی پایدار

طراحی شکل ۲-۱۸ امکان‌پذیر است، اما از نظر اتصال و ترکیب نگرانی فنی مربوط به پایداری با منطق کاربردی یک شی دامنه، مشکل‌ساز است.

این طراحی نشان‌دهنده جداسازی واضح دغدغه‌ها نیست، بلکه با این ارثبری، دغدغه‌های خدمات فنی با منطق کسب‌وکار لایه دامنه ترکیب می‌شوند.

از سوی دیگر، جداسازی دغدغه‌ها یک فضیلت مطلق نیست که باید به هر قیمتی رعایت شود. همان‌طور که در مقدمه تغییرات محافظت‌شده بحث شد، طراحان باید روی نقاطی تمرکز کنند که به‌طور واقعی باعث بی‌ثباتی پرهزینه می‌شوند. اگر در یک برنامه خاص، گسترش کلاس‌ها از PersistentObject به یک راه حل تمیز و ساده منجر شود و مشکلاتی در طراحی یا نگهداری در بلندمدت ایجاد نکند، چرا از آن استفاده نشود؟

پاسخ در درک تکامل نیازمندی‌ها و طراحی برنامه نهفته است. همچنین، زبان برنامه‌نویسی نیز تأثیرگذار است: زبان‌هایی با وراثت تک‌گانه (مانند جاوا) در این رویکرد تنها ابرکلاس ارزشمند خود را مصرف کرده‌اند.

۲-۲۲ مسائل حل نشده

این یک مقدمه بسیار مختصر به مشکلات و راه‌حل‌های طراحی در یک چارچوب و سرویس پایداری بود. بسیاری از مسائل مهم به‌طور سطحی بیان شده‌اند، از جمله غیرمادی‌سازی اشیا^۱، مادی‌سازی و غیرمادی‌سازی مجموعه‌ها، اجرای پرس‌وجو^۲ برای گروهی از اشیا، مدیریت کامل تراکنش‌ها، مدیریت خطا هنگام شکست عملیات پایگاه داده، دسترسی چندکاربره و استراتژی‌های قفل‌گذاری و امنیت (کنترل دسترسی به پایگاه داده).

^۱ به‌طور خلاصه، نگاشت‌گرها باید متدهای `putObjectToStorage` را تعریف کنند. غیرمادی‌سازی سلسله‌مراتب ترکیب نیاز به همکاری بین چندین نگاشت‌گر و نگهداری جداول ارتباطی دارد (در صورتی که از یک پایگاه داده رابطه‌ای استفاده شود).

² Query

فصل ۳

روش: پیاده‌سازی چارچوب پایداری با الگوها

۳-۱ مقدمه

در این فصل، چارچوب نرمافزاری برای مدیریت پایگاه داده‌های رابطه‌ای با استفاده از الگوهای طراحی که در فصل قبل مورد بررسی قرار گرفت پیاده‌سازی شده است. هدف اصلی این چارچوب، تسهیل انجام عملیات پایه‌ای مانند درج، بهروزرسانی، حذف و بازیابی داده‌ها از جداول پایگاهداده است. این چارچوب از الگوهای طراحی مانند **Singleton** برای اطمینان از وجود تنها یک نمونه از کلاس‌های مدیریت‌کننده پایگاهداده، حالت برای مدیریت حالت اشیا و دستور برای اجرای عملیات به صورت تراکنش‌های گروهی استفاده می‌کند.

در این چارچوب، جداول شهر و کشور به عنوان نمونه‌هایی از موجودیت‌های پایگاهداده در نظر گرفته شده اند و عملیات **CRUD** روی آن‌ها انجام می‌شود. همچنین، از یک ایجاد‌کننده پرس‌و‌جو¹ برای ساخت پویای پرس‌و‌جوهای SQL استفاده شده است که امکان فیلتر کردن، مرتب‌سازی و محدود کردن نتایج را فراهم می‌کند. این چارچوب به گونه‌ای طراحی شده است که قابلیت اتصال به چندین نوع پایگاهداده را داشته باشد و به راحتی قابل توسعه برای موجودیت‌های دیگر باشد.

۳-۲ ساخت طرح‌واره و درج رکوردهای پایگاه داده مورد استفاده

در این پروژه به منظور آزمایش نگاشت شی-رابطه‌ای و عملیات درج، بهروزرسانی و حذف، نیاز به یک پایگاه داده داریم. پایگاه داده مورد استفاده، بخشی از پایگاه داده **sakila** (فقط جداول city و country) است که کدهای ساخت طرح‌واره و درج رکوردهای جداول آن‌ها برداشته شده اند. شکل ۳-۱ نمایی از بخشی از اسکریپت SQL مربوط به این پایگاه داده جهت استفاده در میزکار² MySQL نشان می‌دهد.

¹ Query Builder

² Workbench

```

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
DROP SCHEMA IF EXISTS cc;
CREATE SCHEMA cc;
USE cc;
CREATE TABLE city (
    city_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    city VARCHAR(50) NOT NULL,
    country_id INT UNSIGNED NOT NULL,
    last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (city_id),
    KEY idx_fk_country_id (country_id),
    CONSTRAINT `fk_city_country` FOREIGN KEY (country_id) REFERENCES country (country_id) ON DELETE RESTRICT ON UPDATE CASCADE
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
CREATE TABLE country (
    country_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    country VARCHAR(50) NOT NULL,
    last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (country_id)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
Insert into country
(`country_id`, `country`, `last_update`)
Values
('1','Afghanistan','2006-02-15 04:44:00.000')
;
Insert into country

```

شکل ۱-۳: بخشی از اسکریپت SQL پایگاه داده مورد استفاده جهت به کارگیری در میزکار MySQL

همان‌طور که مشاهده می‌شود، جدول city دارای ستون‌های city_id (کلید اصلی)، city (کلید اصلی)، country_id اشاره‌کننده به کلید اصلی جدول country و last_update است. جدول country نیز دارای ستون‌های country_id (کلید اصلی)، country و last_update است. به جدول city، ۶۰۰ رکورد و به جدول country، ۱۰۹ رکورد اضافه کردیم. حال، این دو جدول آماده انجام نگاشت شی-رابطه‌ای و عملیات درج، بهروزرسانی و حذف هستند.

۳-۳ پیاده‌سازی چارچوب: بخش پایگاه داده

در این قسمت به پیاده‌سازی بخشی از چارچوب که به پایگاه داده مرتبط است می‌پردازیم. در ابتدا در قسمت اتصال، کلاسی به نام DatabaseConnection تعریف می‌کنیم که امکان اتصال به سه نوع پایگاه داده مختلف (PostgreSQL، MySQL و SQLite) را فراهم می‌کند. برای این کار، نیاز به کتابخانه‌های PostgreSQL، MySQL و SQLite (به ترتیب برای psycopg2، sqlite3 و MySQLdb) داریم. کلاس تعریف شده یک متده‌سازنده دارد که هنگام ایجاد نمونه‌ای از کلاس فراخوانی می‌شود. این متده، ورودی‌های db_type (نوع پایگاه داده که می‌تواند sqlite، mysql و ya postgresql باشد، host (آدرس سرور پایگاه داده)، username (نام کاربری برای احراز هویت)، password (رمز عبور برای احراز هویت) و

(نام پایگاه داده) را دریافت می‌کند.^۱ سپس، متدهای `_connect` به کمک ویژگی `db_name` برقراری اتصال به پایگاه داده فراخوانی می‌شود. این متدها بسته به نوع پایگاه داده، اتصال مناسب را ایجاد می‌کنند. اگر قصد اتصال به MySQL را داشته باشیم، از `MySQL.connect` برای ایجاد این اتصال استفاده می‌کنیم. برای پایگاه داده‌های دیگر نیز چنین می‌کنیم. اگر مقدار `db_type` معتبر نباشد، یک خطای SQL ایجاد می‌کنیم. متدهای `cursor` و `get_cursor` برای اجرای دستورات در پایگاه داده استفاده می‌شود. متدهای `commit` و `close` برای بستن اتصال به پایگاه داده به کار گرفته می‌شود تا منابع آزاد شوند. به پیاده‌سازی انجام شده در شکل ۳-۲ توجه کنید.

```

import MySQLdb
import sqlite3
import psycopg2
class DatabaseConnection:
    def __init__(self, db_type, host, username, password, db_name):
        self.db_type = db_type
        self.host = host
        self.username = username
        self.password = password
        self.db_name = db_name
        self.connection = self._connect()
    def _connect(self):
        if self.db_type == "mysql":
            return MySQLdb.connect(
                host=self.host,
                user=self.username,
                passwd=self.password,
                db=self.db_name,
            )
        elif self.db_type == "sqlite":
            return sqlite3.connect(self.db_name)
        elif self.db_type == "postgresql":
            return psycopg2.connect(
                host=self.host,
                user=self.username,
                password=self.password,
                dbname=self.db_name,
            )
        else:
            raise ValueError(f"Unsupported database type: {self.db_type}")
    def get_cursor(self):
        return self.connection.cursor()
    def commit(self):
        self.connection.commit()
    def close(self):
        self.connection.close()

```

شکل ۳-۲: پیاده‌سازی قسمت اتصال بخش پایگاه داده چارچوب

^۱ در SQLite، نیازی به `host` و `password` نیست.

در ادامه به بررسی قسمت عملیات می‌پردازیم. در این قسمت، کلاس RDBOperations تعریف شده است که یک الگوی Singleton برای عملیات مربوط به پایگاه داده‌های رابطه‌ای پیاده‌سازی می‌کند و شامل متدهایی برای خواندن، ذخیره، حذف و بهروزرسانی اطلاعات مربوط به شهر (city) و کشور (country) در پایگاه داده است. برای پیاده‌سازی الگوی Singleton از متغیر `_instance` استفاده کردیم تا تنها نمونه این کلاس را ذخیره کرده باشیم. متدهای `get_instance`^۱ است که تضمین می‌کند همیشه فقط یک نمونه از کلاس RDBOperations داشته باشیم. این متدها، اگر نمونه‌ای قبلًا ایجاد نشده باشد، آن را می‌سازد و در دفعات بعدی، همان نمونه ایجاد شده را برمی‌گرداند. متدهای ساخته این کلاس، مقدار اولیه `db` را به `None` تنظیم می‌کنند. قرار است یک نمونه از کلاس `DatabaseConnection` باشد که قبلًا تعریف شده است. متدهای `set_db` به کلاس اجازه می‌دهد اتصال به پایگاه داده را دریافت کند و در متغیر `self.db` نگه دارد. متدهای `get_city_data` اطلاعات یک شهر خاص را از جدول `city` جستجو کرده و برمی‌گرداند. از استفاده شده تا فقط یک رکورد برگردانده شود. متدهای `save_city_data` ابتدا همان عملیات جستجو را انجام می‌دهند. اگر شهر مورد نظر از قبل در پایگاه داده وجود داشته باشد، رکورد آن بهروزرسانی می‌شود؛ و گرنه رکورد درج می‌شود. در هر دو حالت، تغییرات در پایگاه داده `commit` می‌شوند. متدهای `delete_city_data` شهر مشخص شده را از جدول `city` حذف می‌کنند. متدهای `reload_city_data` اطلاعات جدید را از پایگاه داده خوانده و در شی `city` بهروزرسانی می‌کنند. اگر اطلاعاتی برای این `city_id` وجود نداشته باشد، یک خطای `ValueError` برمی‌گرداند. متدهای دیگر یعنی `save_country_data`، `get_country_data` و `execute_query` نیز به صورت مشابه برای `country` عمل می‌کنند. این متدهای `reload_country_data` و `delete_country_data` برای اجرای یک پرس‌وجوی سفارشی در پایگاه داده استفاده می‌شود. این متدهای نتیجه اجرای پرس‌وجو را برمی‌گردانند. به شکل ۳-۳ توجه کنید.

^۱ Static

```

class RDBOperations:
    _instance = None
    @staticmethod
    def get_instance():
        if RDBOperations._instance is None:
            RDBOperations._instance = RDBOperations()
        return RDBOperations._instance
    def __init__(self):
        self.db = None
    def set_db(self, db):
        self.db = db
    def get_city_data(self, city_id):
        sql = "SELECT * FROM city WHERE city_id = %s"
        cursor = self.db.get_cursor()
        cursor.execute(sql, (city_id,))
        return cursor.fetchone()
    def save_city_data(self, city):
        sql = "SELECT * FROM city WHERE city_id = %s"
        cursor = self.db.get_cursor()
        cursor.execute(sql, (city.city_id,))
        existing_city = cursor.fetchone()
        if existing_city:
            sql = "UPDATE city SET city = %s, country_id = %s, last_update = %s WHERE city_id = %s"
            cursor.execute(
                sql,
                (city.city, city.country_id, city.last_update, city.city_id),
            )
        else:
            sql = "INSERT INTO city (city_id, city, country_id, last_update) VALUES (%s, %s, %s, %s)"
            cursor.execute(sql, city.to_sql())
        self.db.commit()
    def delete_city_data(self, city_id):
        sql = "DELETE FROM city WHERE city_id = %s"
        cursor = self.db.get_cursor()
        cursor.execute(sql, (city_id,))
        self.db.commit()
    def reload_city_data(self, city):
        row = self.get_city_data(city.city_id)
        if row:
            city.city, city.country_id, city.last_update = row[1:]
        else:
            raise ValueError(f"City with ID {city.city_id} does not exist.")
    def get_country_data(self, country_id):
        sql = "SELECT * FROM country WHERE country_id = %s"
        cursor = self.db.get_cursor()
        cursor.execute(sql, (country_id,))
        return cursor.fetchone()
    def save_country_data(self, country):
        sql = "SELECT * FROM country WHERE country_id = %s"
        cursor = self.db.get_cursor()
        cursor.execute(sql, (country.country_id,))
        existing_country = cursor.fetchone()
        if existing_country:
            sql = "UPDATE country SET country = %s, last_update = %s WHERE country_id = %s"
            cursor.execute(
                sql,
                (country.country, country.last_update, country.country_id),
            )
        else:
            sql = "INSERT INTO country (country_id, country, last_update) VALUES (%s, %s, %s)"
            cursor.execute(sql, country.to_sql())
        self.db.commit()
    def delete_country_data(self, country_id):
        sql = "DELETE FROM country WHERE country_id = %s"
        cursor = self.db.get_cursor()
        cursor.execute(sql, (country_id,))
        self.db.commit()
    def reload_country_data(self, country):
        row = self.get_country_data(country.country_id)
        if row:
            country.country, country.last_update = row[1:]
        else:
            raise ValueError(f"Country with ID {country.country_id} does not exist.")
    def execute_query(self, query):
        cursor = self.db.get_cursor()
        cursor.execute(query)
        return cursor.fetchall()

```

شکل ۳-۳: پیاده‌سازی قسمت عملیات بخش پایگاه داده چارچوب

در ادامه، کلاس `QueryBuilder` برای ساختن دستورات SQL به صورت پویا طراحی شده است. این کلاس به کاربران اجازه می‌دهد شرایط فیلتر، ترتیب، محدودیت و اتصال جداول را به صورت انعطاف‌پذیر اضافه کنند و در نهایت یک پرس‌وجوی SQL تولید کنند. این کلاس از الگویی استفاده می‌کند که باعث می‌شود بتوان متدها را زنجیره‌ای فراخوانی کرد. این کلاس دارای ویژگی‌های `table_name` (نام جدولی که از آن داده استخراج خواهد شد)، `filters` (لیستی برای ذخیره شرایط `WHERE`، `order_by` (ذخیره دستور ORDER BY برای مرتب‌سازی)، `limit` (مقدار محدودیت تعداد نتایج LIMIT) و `joins` (لیستی برای ذخیره اتصال جداول) است که در متدهای `filter`، `order_by` و `join` تنظیم شده اند. این کلاس دارای متدهای `filter` است که یک شرط فیلتر به لیست اضافه می‌کند. این متدهای `column` (ستون مورد نظر)، `operator` (عملگر مقایسه‌ای مثل `IN`، `LIKE` و...) و `value` (مقدار مورد نظر برای مقایسه) است و به عنوان خروجی، شی `self` را بر می‌گرداند تا امکان زنجیره‌ای فراخوانی کردن متدها فراهم شود. متدهای `order_by` و `limit_results` (صعودی یا نزولی بودن مرتب‌سازی) را دریافت می‌کنند. متدهای `join` (آنچه که از جدول دیگر که باید متصل شود) و `on_clause` (شرطی که دو جدول را به هم متصل می‌کند) دریافت می‌کنند. متدهای `build` (رشته SQL نهایی را تولید و بر می‌گردانند) و `query` (که دستور SQL را بازگشتی را محدود می‌کند) را دریافت می‌کنند.

```
class QueryBuilder:
    def __init__(self, table_name):
        self.table_name = table_name
        self.filters = []
        self.order_by = None
        self.limit = None
        self.joins = []
    def filter(self, column, operator, value):
        self.filters.append(f'{column} {operator} "{value}"')
        return self
    def order_by_column(self, column, ascending=True):
        self.order_by = f'ORDER BY {column} {"ASC" if ascending else "DESC"}'
        return self
    def limit_results(self, limit):
        self.limit = f"LIMIT {limit}"
        return self
    def join(self, table, on_clause):
        self.joins.append(f'JOIN {table} ON {on_clause}')
        return self
    def build(self):
        query = f"SELECT * FROM {self.table_name}"
        if self.joins:
            query += " " + ".join(self.joins)
        if self.filters:
            query += " WHERE " + " AND ".join(self.filters)
        if self.order_by:
            query += " " + self.order_by
        if self.limit:
            query += " " + self.limit
        return query
```

شکل ۳-۴: پیاده‌سازی کلاس `QueryBuilder` چارچوب

۴-۳ پیاده‌سازی چارچوب: بخش مدل‌ها

در این بخش، مدل‌های چارچوب را در چند قسمت پیاده‌سازی می‌کنیم.

کلاس PersistentObject یک کلاس انتزاعی برای مدیریت حالت اشیا در پایگاه داده است. این کلاس از الگوی حالت برای کنترل تغییرات یک شی در سیستم استفاده می‌کند. هر شی دارای یک OID و یک حالت است که مشخص می‌کند در چه مرحله‌ای از چرخه قرار دارد. OID معمولاً از پایگاه داده دریافت می‌شود؛ اگر مقدار آن، None باشد، یعنی این شی هنوز در پایگاه داده ذخیره نشده است. حالت، مقدار پیش‌فرض NewState دارد. این مقدار از طریق متدهای get_instance دریافت می‌شود که نشان‌دهنده الگوی Singleton برای مدیریت حالت است. این کلاس دارای متدهای commit جهت ثبت تغییرات شی در پایگاه داده و اجرای عملیات مرتبط با حالت خاص، delete جهت حذف شی از پایگاه داده، rollback جهت برگرداندن تغییرات انجام‌شده روی شی به حالت قبل و نادیده گرفتن تغییرات ثبت‌نشده، save جهت ذخیره شی و اعمال تغییرات (که به نوع حالت بسته است) و set_state است که حالت شی را تغییر می‌دهد. در کل، این کلاس، کلاسی پایه برای مدیریت اشیای پایدار است که با استفاده از الگوی حالت، هر شی می‌تواند در حالت‌های مختلف رفتارهای متفاوتی داشته باشد. به شکل ۳-۵ توجه کنید.

```
from persistence.state import NewState
class PersistentObject:
    def __init__(self, oid=None):
        self.oid = oid
        self.state = NewState.get_instance()
    def commit(self):
        self.state.commit(self)
    def delete(self):
        self.state.delete(self)
    def rollback(self):
        self.state.rollback(self)
    def save(self):
        self.state.save(self)
    def set_state(self, state):
        self.state = state
```

شکل ۳-۵: پیاده‌سازی کلاس PersistentObject چارچوب

کلاس City اطلاعات مربوط به یک شهر در پایگاه داده را مدیریت می‌کند. این کلاس از کلاس PersistentObject ارث‌بری کرده و قابلیت‌های ذخیره‌سازی، بازیابی و مدیریت حالت شی را دارد. این کلاس دارای ویژگی‌هایی متناسب با فیلدهای آن در جدول و __country است که شی کشور مربوط به شهر را در خود جای می‌دهد. مقدار OID همان city_id تنظیم می‌شود. متدهای __str__ نمایش متنی و خوانا از شی را ارائه می‌دهد و هنگام دیباگ کردن داده‌ها و چاپ اطلاعات شهر استفاده می‌شود. متدهای to_sql داده‌های شی

را به فرمت مناسب برای ذخیرهسازی در پایگاه داده تبدیل می‌کند. این متدهنگام درج در پایگاه داده کاربرد دارد. متدهنگام `from_sql`, یک شی `City` را از داده‌های دریافت‌شده از پایگاه داده می‌سازد که از آن در نگاشت اطلاعات از رکورد به شی استفاده می‌شود. همان‌طور که مشاهده می‌شود، این کلاس از دکوراتور^۱ `property` استفاده کرده و در خود به کلاس دسترسی داشته است. متدهنگام `country` که با دکوراتور `method` تعریف شده (به این معنا که هنگام فراخوانی آن، نیازی به گذاشتن پرانتزها نداریم)، رابطه بین شهر و کشور را مدیریت می‌کند. اگر مقدار `_country` مقدار نداشته باشد، اطلاعات مربوط به کشور از پایگاه داده دریافت شده و مقداردهی می‌شود.^۲ از الگوی طراحی `Singleton` برای دریافت داده‌های کشور استفاده شده است. متدهنگام `Country` نیز با تعدادی متدهنگام `setter` جهت تنظیم `country_id` و `_country` نوشته شده است. کلاس `Country` شبیه متدهای کلاس `City` هستند تعریف می‌شود. پیاده‌سازی کلاس‌های `City` و `Country` در شکل ۳-۶ قابل مشاهده است.

```
from models.base import PersistentObject
from persistence.facade import PersistenceFacade
from models.country import Country
class City(PersistentObject):
    def __init__(self, city_id=None, city=None, country_id=None, last_update=None):
        super().__init__(oid=city_id)
        self.city_id = city_id
        self.city = city
        self.country_id = country_id
        self.last_update = last_update
        self._country = None
    def __str__(self):
        return f"City(id={self.city_id}, city={self.city}, country_id={self.country_id}, last_update={self.last_update})"
    def to_sql(self):
        return (self.city_id, self.city, self.country_id, self.last_update)
    @classmethod
    def from_sql(cls, row):
        return cls(city_id=row[0], city=row[1], country_id=row[2], last_update=row[3])
    @property
    def country(self):
        if self._country is None:
            self._country = PersistenceFacade.getInstance().get(
                self.country_id, Country
            )
        return self._country
    @country.setter
    def country(self, country):
        self._country = country
        self.country_id = country.country_id
class Country(PersistentObject):
    def __init__(self, country_id=None, country=None, last_update=None):
        super().__init__(oid=country_id)
        self.country_id = country_id
        self.country = country
        self.last_update = last_update
    def __str__(self):
        return f"Country(id={self.country_id}, country={self.country}, last_update={self.last_update})"
    def to_sql(self):
        return (self.country_id, self.country, self.last_update)
    @classmethod
    def from_sql(cls, row):
        return cls(country_id=row[0], country=row[1], last_update=row[2])
```

شکل ۳-۶: پیاده‌سازی کلاس‌های `City` و `Country` چارچوب

¹ Decorator

² Lazy Loading

۳-۵ پیاده‌سازی چارچوب: بخش پایداری

در این بخش، وارد قسمت‌های اصلی این معماری خواهیم شد.

در ابتدا، قصد پیاده‌سازی کلاس PersistenceFacade را داریم. این کلاس یک الگو از نوع Facade برای مدیریت عملیات پایدارسازی داده‌ها است. این کلاس تنها یک نمونه از خود ایجاد می‌کند و از طریق نگاشت‌گرها، عملیات مختلف را روی اشیا انجام می‌دهد. این کلاس، نگاشت‌گرها را مدیریت می‌کند که واسطه بین کلاس‌های مدل و پایگاه داده هستند. از register_mapper برای ثبت نگاشت‌گرهای جدید استفاده می‌شود. هر کلاس نگاشت‌گر خاص خود را دارد. نگاشت‌گرها در یک دیکشنری به نام mappers ذخیره می‌شوند. متدهای get و register_mapper می‌کنند که آیا برای کلاس پایدار، نگاشت‌گر ثبت شده است یا خیر؛ اگر نگاشت‌گر موجود بود، شی از طریق آن نگاشت‌گر بازیابی می‌شود و در غیر این صورت، خطا ایجاد می‌شود. به‌طور مشابه، متدهای insert، update، put و delete و reload پیاده‌سازی شدند. به‌طور کلی، این کلاس نقش یک لایه میانی بین مدل‌ها و پایگاه داده را ایفا می‌کند و کار با داده‌ها را تسهیل می‌کند. به شکل ۳-۷ توجه کنید.

```
class PersistenceFacade:
    _instance = None
    @staticmethod
    def getInstance():
        if PersistenceFacade._instance is None:
            PersistenceFacade._instance = PersistenceFacade()
        return PersistenceFacade._instance
    def __init__(self):
        self.mappers = {}
    def register_mapper(self, cls, mapper):
        self.mappers[cls] = mapper
    def get(self, oid, persistence_class):
        mapper = self.mappers.get(persistence_class)
        if mapper:
            return mapper.get(oid)
        raise ValueError(f"No mapper found for class {persistence_class}")
    def put(self, obj):
        mapper = self.mappers.get(type(obj))
        if mapper:
            mapper.put(obj)
        else:
            raise ValueError(f"No mapper found for object type {type(obj)}")
    def insert(self, obj):
        mapper = self.mappers.get(type(obj))
        if mapper:
            mapper.insert(obj)
        else:
            raise ValueError(f"No mapper found for object type {type(obj)}")
    def update(self, obj):
        mapper = self.mappers.get(type(obj))
        if mapper:
            mapper.update(obj)
        else:
            raise ValueError(f"No mapper found for object type {type(obj)}")
    def delete(self, obj):
        mapper = self.mappers.get(type(obj))
        if mapper:
            mapper.delete(obj)
        else:
            raise ValueError(f"No mapper found for object type {type(obj)}")
    def reload(self, obj):
        mapper = self.mappers.get(type(obj))
        if mapper:
            mapper.reload(obj)
        else:
            raise ValueError(f"No mapper found for object type {type(obj)}")
```

شکل ۳-۷: پیاده‌سازی کلاس PersistenceFacade چارچوب

در قسمت بعدی، الگوی نگاشت‌گر برای مدیریت عملیات پایگاه داده روی اشیای مدل شهر و کشور را بررسی می‌کنیم. کلاس IMapper رابطی است که دو متدهای get و put را تعریف می‌کند. این متدها در کلاس‌های فرزند باید پیاده‌سازی شوند. کلاس AbstractPersistenceMapper یک کلاس انتزاعی است که از IMapper ارث بری کرده و یک حافظهٔ نهان به همراه قفل^۱ برای مدیریت همزمانی دارد. این کلاس دو ویژگی cached_objects (یک دیکشنری برای نگهداری اشیای واقع در حافظهٔ نهان جهت جلوگیری از درخواست‌های اضافه به پایگاه داده) و lock (برای جلوگیری از مشکل هم‌روندي^۲ در موقعی که چندین نخ به داده‌ها دسترسی دارند) دارد. مقدار اولیهٔ lock در متدهای سازنده برابر خروجی threading.Lock قرار می‌گیرد. متدهای get ابتدا بررسی می‌کند که آیا شی مورد نظر در حافظهٔ نهان موجود است یا خیر. اگر نباشد، از متدهای کلاس‌های فرزند پیاده‌سازی خواهد شد داده را از پایگاه داده دریافت می‌کند. اگر داده از پایگاه داده دریافت شود، آن را در حافظهٔ نهان ذخیره می‌کند. متدهای put در کلاس‌های فرزند باید پیاده‌سازی شود. کلاس AbstractRDBMapper از AbstractPersistenceMapper ارث بری کرده و عملیات مربوط به پایگاه داده رابطه‌ای را مدیریت می‌کند. این کلاس دارای ویژگی db_operations (شی از RDBOperations برای مدیریت ارتباط با پایگاه داده) است. کلاس CityMapper مسئول ارتباط با پایگاه داده برای اشیای City است. این کلاس متدهای دارد که داده‌های مربوط به شهر را از پایگاه داده دریافت می‌کند. اگر داده یافت شود، متدهای from_sql از کلاس City را برای تبدیل داده به شی City فراخوانی می‌کند. متدهای CRUD نیز پیاده‌سازی می‌شوند. کلاس CountryRDBMapper متدهایی به صورت مشابه دارد.

به شکل ۳-۸ توجه کنید.

^۱ Lock

^۲ Race Condition

```

import threading
from models.city import City
from models.country import Country
from database.operations import RDBOperations
class IMapper:
    def get(self, oid):
        raise NotImplementedError
    def put(self, obj):
        raise NotImplementedError
class AbstractPersistenceMapper(IMapper):
    def __init__(self):
        self.cached_objects = {}
        self.lock = threading.Lock()
    def get(self, oid):
        with self.lock:
            obj = self.cached_objects.get(oid)
            if obj is None:
                obj = self._get_object_from_storage(oid)
            if obj is not None:
                self.cached_objects[oid] = obj
        return obj
    def put(self, obj):
        raise NotImplementedError("Subclasses should implement 'put' method.")
    def _get_object_from_storage(self, oid):
        raise NotImplementedError(
            "Subclasses should implement '_get_object_from_storage' method."
        )
class AbstractRDBMapper(AbstractPersistenceMapper):
    def __init__(self):
        super().__init__()
        self.rdb_operations = RDBOperations.get_instance()
    def _get_object_from_storage(self, oid):
        raise NotImplementedError
class CityRDBMapper(AbstractRDBMapper):
    def _get_object_from_storage(self, oid):
        row = self.rdb_operations.get_city_data(oid)
        if row:
            return City.from_sql(row)
        return None
    def put(self, obj):
        self.rdb_operations.save_city_data(obj)
    def insert(self, obj):
        self.rdb_operations.save_city_data(obj)
    def update(self, obj):
        self.rdb_operations.save_city_data(obj)
    def delete(self, obj):
        self.rdb_operations.delete_city_data(obj.oid)
    def reload(self, obj):
        self.rdb_operations.reload_city_data(obj)
class CountryRDBMapper(AbstractRDBMapper):
    def _get_object_from_storage(self, oid):
        row = self.rdb_operations.get_country_data(oid)
        if row:
            return Country.from_sql(row)
        return None
    def put(self, obj):
        self.rdb_operations.save_country_data(obj)
    def insert(self, obj):
        self.rdb_operations.save_country_data(obj)
    def update(self, obj):
        self.rdb_operations.save_country_data(obj)
    def delete(self, obj):
        self.rdb_operations.delete_country_data(obj.oid)
    def reload(self, obj):
        self.rdb_operations.reload_country_data(obj)

```

شکل ۳-۸: پیاده‌سازی قسمت نمایشگر بخش پایداری چارچوب

در ادامه، الگوی حالت را برای مدیریت حالت‌های مختلف یک شی در پایگاه داده بررسی می‌کنیم. این الگو باعث می‌شود که رفتار یک شی بسته به حالت جاری آن تغییر کند. کلاس PObjectState یک کلاس پایه است که حالت‌های مختلف یک شی را تعریف می‌کند. این کلاس چهار متدهای commit، delete و rollback دارد. کلاس NewState نمایانگر شی جدیدی است که هنوز در پایگاه داده ذخیره نشده است. وقتی commit اجرا شود، شی در پایگاه داده ذخیره شده و حالت آن به OldCleanState تغییر می‌کند. کلاس OldCleanState نمایانگر شی ذخیره شده و بدون تغییر است. متدهای delete و commit در این کلاس، شی را به حالت حذف شده (OldDeleteState) تغییر می‌دهد. متدهای save و rollback در این کلاس، در صورت ایجاد تغییری در شی، حالت آن را به OldDirtyState تغییر می‌دهد. کلاس OldDirtyState نمایانگر شی تغییریافته‌ای است که هنوز در پایگاه داده ذخیره نشده است. متدهای commit و rollback در این کلاس، تغییرات شی در پایگاه داده را ذخیره کرده و حالت آن را به OldCleanState تغییر می‌دهد. متدهای delete و commit در این کلاس، اما هنوز commit نشده است. متدهای save و rollback در این کلاس، حذف را تأیید می‌کند و حالت شی را به DeletedState تغییر می‌دهد. متدهای commit و rollback در این کلاس، تغییرات را لغو کرده و حالت را به OldCleanState برگرداند. کلاس DeletedState نشان‌دهنده شی‌ای است که از پایگاه داده حذف شده است؛ این حالت نهایی است و پس از آن، شیء دیگر قابل بازگردانی نخواهد بود. به شکل ۳-۹ توجه کنید.

```

from persistence.facade import PersistenceFacade
class PObjectState:
    def commit(self, obj):
        pass
    def delete(self, obj):
        pass
    def rollback(self, obj):
        pass
    def save(self, obj):
        pass
class NewState(PObjectState):
    _instance = None
    @staticmethod
    def get_instance():
        if NewState._instance is None:
            NewState._instance = NewState()
        return NewState._instance
    def commit(self, obj):
        PersistenceFacade.getInstance().insert(obj)
        obj.set_state(OldCleanState.get_instance())
class OldCleanState(PObjectState):
    _instance = None
    @staticmethod
    def get_instance():
        if OldCleanState._instance is None:
            OldCleanState._instance = OldCleanState()
        return OldCleanState._instance
    def delete(self, obj):
        obj.set_state(OldDeleteState.get_instance())
    def save(self, obj):
        obj.set_state(OldDirtyState.get_instance())
class OldDirtyState(PObjectState):
    _instance = None
    @staticmethod
    def get_instance():
        if OldDirtyState._instance is None:
            OldDirtyState._instance = OldDirtyState()
        return OldDirtyState._instance
    def commit(self, obj):
        PersistenceFacade.getInstance().update(obj)
        obj.set_state(OldCleanState.get_instance())
    def rollback(self, obj):
        PersistenceFacade.getInstance().reload(obj)
        obj.set_state(OldCleanState.get_instance())
    def delete(self, obj):
        obj.set_state(OldDeleteState.get_instance())
class OldDeleteState(PObjectState):
    _instance = None
    @staticmethod
    def get_instance():
        if OldDeleteState._instance is None:
            OldDeleteState._instance = OldDeleteState()
        return OldDeleteState._instance
    def commit(self, obj):
        PersistenceFacade.getInstance().delete(obj)
        obj.set_state(DeletedState.get_instance())
    def rollback(self, obj):
        PersistenceFacade.getInstance().reload(obj)
        obj.set_state(OldCleanState.get_instance())
class DeletedState(PObjectState):
    _instance = None
    @staticmethod
    def get_instance():
        if DeletedState._instance is None:
            DeletedState._instance = DeletedState()
        return DeletedState._instance

```

شکل ۳-۹: پیاده‌سازی قسمت حالت‌های بخش پایداری چارچوب

۶-۳ پیاده‌سازی چارچوب: بخش تراکنش‌ها

در این بخش، ابتدا قسمت پیاده‌سازی الگوی دستور را خواهیم داشت. همان‌طور که قبلاً توضیح داده شد، الگوی دستور یک راهکار برای جدا کردن درخواست یک عملیات از شی انجام‌دهنده آن است. کلاس ICommand یک رابط است که حاوی متدهای execute (اجرای عملیات) و undo (لغو عملیات) است که در این پیاده‌سازی، undo بررسی نشده است. کلاس‌های دیگر این رابط را پیاده‌سازی می‌کنند و هر کدام مسئول اجرای یک عملیات خاص روی پایگاه داده هستند. کلاس ICommand از DBCommand ارث بری می‌کند و کلاس پایه برای همه دستورات پایگاه داده است. این کلاس دارای یک ویژگی شی است که عملیات روی آن انجام می‌شود. متدها در این کلاس نیز پیاده‌سازی نشده‌اند. کلاس DBInsertCommand به صورت مشابه پیاده‌سازی درج در پایگاه داده را انجام می‌دهد. متدهای execute از طریق PersistenceFacade شی را در پایگاه داده ذخیره می‌کند. کلاس‌های DBDeleteCommand و DBUpdateCommand به صورت مشابه پیاده‌سازی می‌شوند. به شکل ۳-۱۰ توجه کنید.

```
from persistence.facade import PersistenceFacade
class ICommand:
    def execute(self):
        raise NotImplementedError
    def undo(self):
        raise NotImplementedError
class DBCommand(ICommand):
    def __init__(self, obj):
        self.object = obj
    def execute(self):
        raise NotImplementedError
    def undo(self):
        pass
class DBInsertCommand(DBCommand):
    def execute(self):
        PersistenceFacade.getInstance().insert(self.object)
class DBUpdateCommand(DBCommand):
    def execute(self):
        PersistenceFacade.getInstance().update(self.object)
class DBDeleteCommand(DBCommand):
    def execute(self):
        PersistenceFacade.getInstance().delete(self.object)
```

شکل ۳-۱۰: پیاده‌سازی قسمت دستورات بخش تراکنش‌های چارچوب

در ادامه، مدیریت تراکنش‌ها براساس الگوی دستور پیاده‌سازی می‌شود. این طراحی باعث می‌شود که عملیات پایگاه داده (درج، به روزرسانی و حذف) به صورت گروهی و به ترتیب مناسب انجام شوند. کلاس Transaction یک تراکنش را مدل‌سازی می‌کند، متغیر self.commands یک لیست از دستورات پایگاه داده را ذخیره می‌کند و دستورات DBUpdateCommand، DBInsertCommand داده شامل شامل

اشیا برای DBDeleteCommand هستند که به کمک متدهای add_delete و add_update و add_insert درج، به روزرسانی و حذف اضافه می‌شوند. متدهای sort دستورات را طوری مرتب می‌کند که ابتدا درج انجام شود، چراکه می‌خواهیم قبل از به روزرسانی و حذف از وجود رکورد در پایگاه داده اطمینان حاصل کنیم. متدهای commit تمام دستورات را اجرا می‌کند (با اولویت درج). به شکل ۱۱-۳ توجه کنید.

```
from transactions.commands import DBInsertCommand, DBUpdateCommand, DBDeleteCommand
class Transaction:
    def __init__(self):
        self.commands = []
    def add_insert(self, obj):
        self.commands.append(DBInsertCommand(obj))
    def add_update(self, obj):
        self.commands.append(DBUpdateCommand(obj))
    def add_delete(self, obj):
        self.commands.append(DBDeleteCommand(obj))
    def sort(self):
        self.commands.sort(
            key=lambda cmd: isinstance(cmd, DBInsertCommand), reverse=True
        )
    def commit(self):
        self.sort()
        for cmd in self.commands:
            cmd.execute()
```

شکل ۱۱-۳: پیاده‌سازی قسمت تراکنش بخش تراکنش‌های چارچوب

۳-۷ پیاده‌سازی چارچوب: بخش اصلی

در این بخش، از معماری ساخته شده در بخش‌های قبل استفاده می‌کنیم و به نوعی، آزمایش چارچوب را انجام می‌دهیم. در ابتداء، ماثولوها لازم از فایل‌های مختلف برنامه اضافه می‌شوند. سپس، تنظیمات اتصال به پایگاه داده از یک فایل config.json که محتويات آن می‌تواند مثلاً به صورت زیر باشد، خوانده می‌شود.

```
{
    "db_type": "mysql",
    "host": "localhost",
    "username": "root",
    "password": "root",
    "db_name": "cc"
}
```

یک شی Database Connection ساخته می‌شود که با استفاده از اطلاعات config.json به پایگاه داده متصل می‌شود. عملیات پایگاه داده به کمک RDBOperations که یک Singleton است تنظیم می‌شود؛ سپس، اتصال پایگاه داده تنظیم می‌شود تا db_operations بتواند روی آن کار کند. در ادامه، PersistenceFacade برای مدیریت عملیات ذخیره‌سازی ایجاد می‌شود. دو نگاشت‌گر برای شهر و کشور ساخته شده و در PersistenceFacade ثبت می‌شوند. جهت آزمایش، شی Country جدید ساخته و در پایگاه داده ذخیره می‌شود. یک شی City جدید ساخته شده و به تراکنش اضافه می‌شود. سپس تراکنش

committ شده و درج انجام می‌شود. یک شهر از پایگاه داده بازیابی شده و چاپ می‌شود. نام شهر تغییر داده شده و یک تراکنش جدید برای بهروزرسانی آن اجرا می‌شود. شهر مجدداً از پایگاه داده دریافت شده و حذف می‌شود. شهر حذف شده دوباره از پایگاه داده خوانده می‌شود تا بررسی شود که حذف شده است. در ادامه، جهت آزمایش QueryBuilder، دو پرس‌وجوی SQL ایجاد شده است. پرس‌وجوی اول، شهرهایی را که دارند، مرتب کرده و ده عدد از آنها را دریافت می‌کند. پرس‌وجوی دوم، اتصال بین جداول city و country را انجام می‌دهد تا شهرهای ایران دریافت شوند. به شکل ۳-۱۲ توجه کنید.

```

from database.connection import DatabaseConnection
from database.operations import RDBOperations
from database.query_builder import QueryBuilder
from models.city import City
from models.country import Country
from persistence.facade import PersistenceFacade
from persistence.mapper import CityRDBMapper, CountryRDBMapper
from transactions.transaction import Transaction
import json
with open("config.json", "r") as config_file:
    config = json.load(config_file)
db = DatabaseConnection(
    db_type=config["db_type"],
    host=config["host"],
    username=config["username"],
    password=config["password"],
    db_name=config["db_name"],
)
rdb_operations = RDBOperations.get_instance()
rdb_operations.set_db(db)
persistence = PersistenceFacade.getInstance()
city_mapper = CityRDBMapper()
country_mapper = CountryRDBMapper()
persistence.register_mapper(City, city_mapper)
persistence.register_mapper(Country, country_mapper)
new_country = Country(country_id=110, country="NewCountry", last_update="2025-11-15")
persistence.insert(new_country)
new_city = City(city_id=601, city="NewCity", country_id=83, last_update="2025-11-15")
transaction = Transaction()
transaction.add_insert(new_city)
transaction.commit()
retrieved_city = persistence.get(601, City)
print("Retrieved City:", retrieved_city)
print("Retrieved City's Country:", retrieved_city.country)
retrieved_city.city = "UpdatedCity"
transaction = Transaction()
transaction.add_update(retrieved_city)
transaction.commit()
updated_city = persistence.get(601, City)
print("Updated City:", updated_city)
transaction = Transaction()
transaction.add_delete(updated_city)
transaction.commit()
deleted_city = persistence.get(601, City)
print("Deleted City:", deleted_city)
query = (
    QueryBuilder("city")
        .filter("country_id", "=", 85)
        .order_by_column("city", ascending=True)
        .limit_results(10)
)
results = rdb_operations.execute_query(query.build())
for row in results:
    print(City.from_sql(row))
query = (
    QueryBuilder("city")
        .join("country", "city.country_id = country.country_id")
        .filter("country.country", "=", "Iran")
        .order_by_column("city.city", ascending=True)
)
results = rdb_operations.execute_query(query.build())
for row in results:
    print(City.from_sql(row))

```

شکل ۳-۱۲: پیاده‌سازی بخش اصلی چارچوب

فصل ۴

جمع‌بندی

۴-۱ مقدمه

در این پژوهش، چارچوبی پایدار برای ذخیره‌سازی و بازیابی داده‌ها طراحی و پیاده‌سازی شد که هدف آن بهبود کارایی، سازگاری و مدیریت بهتر داده‌ها در سیستم‌های نرم‌افزاری است. در این فصل، نتایج به‌دست‌آمده از پژوهه بررسی شده و نقاط قوت، چالش‌ها و راهکارهای پیشنهادی برای بهبود آینده سیستم ارائه می‌شود.

۴-۲ ارزیابی عملکرد چارچوب پیشنهادی: چالش‌ها و راهکارها

همان‌طور که در فصل دوم اشاره شد، چارچوب‌های پایداری مشابهی وجود دارند که ابعاد خیلی بزرگ‌تری گسترش یافته‌اند. این چارچوب به عنوان یک نمونه آموزشی با رعایت اصول مهندسی نرم‌افزار ارائه شد و تلاش شد که بتواند با فناوری‌های مختلف سازگار باشد. یکی از اقدامات صورت‌گرفته، کاهش احتمال ناسازگاری داده‌ها با استفاده از روش‌های مدیریت تراکنش است. این چارچوب، امکان افزودن قابلیت‌های جدید بدون نیاز به تغییرات گسترده را دارد.

یکی از مسائلی که کمتر به آن پرداخته شد و در صورت علاقه‌مندی می‌توانید به آن بپردازید، عملیات غیرمادی‌سازی است. همچنین، اجرای پرس‌وجوها که به صورت زنجیره‌ای برای رکوردهای پایگاه داده پیاده‌سازی شد می‌تواند بیش از این گسترش یابد، طوری که بتوان روی گروهی از اشیا نیز این عملیات را انجام داد. علاوه‌بر این، مدیریت تراکنش‌ها می‌تواند کامل‌تر از این صورت گیرد و در صورت شکست عملیات پایگاه داده، مدیریت خطاهای لازم انجام شود. از لحاظ دسترسی امنیتی به پایگاه داده نیز می‌توان به کمک قفل‌گذاری، این چارچوب را بهبود بخشید.

ضمناً بی‌شک مدیریت زمان و حافظه همواره در سیستم‌ها مورد اهمیت بوده است. در این پژوهه با کمک گرفتن از حافظه نهان و ایده مادی‌سازی تنبل در طراحی، سعی کردیم این موضوع را محقق کنیم؛ اما ممکن است پیشنهاداتی در خصوص بهبود این مورد داشته باشید. با توجه به کاربردهای گسترده‌ای که می‌تواند این پژوهه در تجارت الکترونیک، رایانش ابری، مدیریت اطلاعات سازمانی، مدیریت امور مالی و... داشته باشد،

گسترش و بهبود چارچوب ارائه شده از جهات ذکر شده، خالی از لطف نیست.

۳-۴ نتیجه‌گیری

نتایج این پژوهش نشان می‌دهد که چارچوب پیشنهادی توانسته است عملکرد مطلوبی را در مدیریت داده‌ها ارائه دهد. با وجود چالش‌های مطرح شده، با به کارگیری راهکارهای پیشنهادی می‌توان این چارچوب را بهینه‌سازی و توسعه داد. در آینده، بررسی روش‌های پیشرفته‌تر برای پردازش داده‌ها و بهبود الگوریتم‌های مدیریت تراکنش می‌تواند موجب افزایش بهره‌وری این سیستم در کاربردهای عملی شود.

مراجع

- [1] Craig L.; *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd Edition, Pearson, 2001.

واژه نامه انگلیسی به فارسی

A

Abstract

انتزاعی

C

Case Logic

منطق شرطی

Cohesion

انسجام

Command Pattern

الگوی دستور

Concrete

مشخص

Concurrency Control

کنترل همزمانی

Constructor Chaining

زنجیره سازنده‌ها

Context Object

شی زمینه

Coupling

اتصال

D

Data-Driven

داده محور

Deferring

به تعویق انداختن

Design Discipline

رشته طراحی

Direct Mapping Design

طراحی نگاشت مستقیم

Dirty Objects

اشیای تغییریافته

Dynamic

پویا

E

Eager Materialization

مادی‌سازی پیش‌دستانه

Encapsulation

کپسوله‌سازی

G

GUI Framework

چارچوب رابط کاربری گرافیکی

H

History Stack

پشتۀ تاریخچه

L

Lazy Materialization

مادی‌سازی تنبیل

Leaf	برگ
Lightweight Object	شی سبک وزن
Local Cache	حافظه نهان محلی
Lock	قفل
Logging	گزارش
O	
Object Identifier	شناسه شی
Object-Oriented	شی گرا
Observer Pattern	الگوی ناظر
O-R Mapping	نگاشت شی-رابطه‌ای
Overriding	بازنویسی
P	
Package	بسته
Passivation	غیرفعال‌سازی
Pasting	چسباندن
Persistence Subsystem	زیرسیستم پایداری
Persistent Store	مخزن پایدار
Point of Sale	نقطه فروش
Polymorphic	چندریختی
Protected Variations	تغییرات محافظت‌شده
Public	عمومی
Pure Fabrication	ساختگی محض
Q	
Qualified Association	ارتباط واجد شرایط
Query	پرس‌وجو
R	
Referential Constraint	محدودیت ارجاعی
Reflective Programming	برنامه‌نویسی بازتابی
Relational Database	پایگاه داده رابطه‌ای

Remote Server سرور از راه دور

Reusability قابلیت استفاده مجدد

S

SAD سند معماری نرم افزار

Schema Mapping نگاشت طرح وارهای

Separation of Concerns جداسازی دغدغه ها

Smart Pointer اشاره گر هوشمند

State Machine Interpreter مفسر ماشین حالت

State Pattern الگوی حالت

State Transmission Model مدل انتقال حالت

Statechart نمودار حالت

Subclassing زیر کلاس گیری

Synchronized همگام سازی شده

System Properties ویژگی های سیستم

T

Task وظیفه

Technical Services Layer لایه خدمات فنی

U

Undoing لغو

Unified Process فرایند یکپارچه

Unit of Work واحد کاری

V

Virtual Proxy پروکسی مجازی

W

Widget ابزار ک

Workbench میز کار

Abstract

In today's world, data management and storage are of great importance, especially in persistent systems that require efficient and adaptable frameworks. This project focuses on analyzing and designing a persistence framework for data storage and retrieval that ensures data integrity and consistency while providing high performance and flexibility. By utilizing modern design methodologies and applying appropriate patterns, a framework has been developed that can reduce data inconsistency issues, enhance the efficiency of data management systems, and be implemented in various software environments. Additionally, optimization techniques, effective transaction management, concurrency control, and cache management have been incorporated to improve its performance in data processing. Although the goal of this framework is not to compete with traditional data management methods, it can be effectively used for data management in scalable and advanced software systems.

Keywords: persistence framework, data retrieval and storage, object-relational mapping, transaction and concurrency management, pattern-based data management.



Shahid Rajaee Teacher Training University
Faculty of Computer Engineering
B. Sc. Thesis

Title:

Designing and Implementing a Persistence Framework with Patterns

Supervisor:

Dr. Mohammad Kalantari

By:

Mohsen Elahifard

Winter 2025