

**پایان‌نامه دوره کارشناسی**

**مهندسی کامپیوتر**

عنوان پروژه:

**طراحی و پیاده‌سازی چارچوب ذخیره و بازیابی اطلاعات با الگوها**

دانشجو:

محسن الهی‌فرد

استاد راهنما:

دكتر محمد کلانتری

**اسفند 1403**



**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 به‌کارگیری 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 حافظۀ نهان نیز در آن لحاظ شده است تا عملکرد آن در پردازش داده‌ها بهبود یابد. اگرچه هدف از ارائۀ 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.

**کلمات کلیدی**

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

**فهرست مطالب**

[فصل 1 1](#_Toc191082872)

[مقدمه 1](#_Toc191082873)

[1-1 مقدمه 1](#_Toc191082874)

[1-2 ضرورت پروژه 1](#_Toc191082875)

[1-3 اهداف پروژه 2](#_Toc191082876)

[1-4 روش و ساختار پروژه 2](#_Toc191082877)

[فصل 2 3](#_Toc191082878)

[پیشینه: طراحی چارچوب پایداری با الگوها 3](#_Toc191082879)

[2-1 مقدمه 3](#_Toc191082880)

[2-2 مشکل اشیای پایدار 4](#_Toc191082881)

[2-3 یک سرویس پایداری از چارچوب پایداری 5](#_Toc191082882)

[2-4 چارچوب‌ها 5](#_Toc191082883)

[2-5 الزامات سرویس و چارچوب پایداری 6](#_Toc191082884)

[2-6 مفاهیم کلیدی 7](#_Toc191082885)

[2-7 الگو: نمایش اشیا به‌عنوان جداول 8](#_Toc191082886)

[2-8 پروفایل مدل‌سازی داده در UML 8](#_Toc191082887)

[2-9 الگو: شناسۀ شی 9](#_Toc191082888)

[2-10 دسترسی به سرویس پایداری با استفاده از یک Facade 10](#_Toc191082889)

[2-11 نگاشت اشیا: الگوی Database Mapper یا Database Broker 11](#_Toc191082890)

[2-12 طراحی چارچوب با الگوی Template Method 14](#_Toc191082891)

[2-13 مادی‌سازی با الگوی Template Method 15](#_Toc191082892)

[2-14 پیکربندی نگاشت‌گرها به‌کمک MapperFactory 20](#_Toc191082893)

[2-15 الگو: مدیریت حافظۀ نهان 21](#_Toc191082894)

[2-16 یکپارچه‌سازی و پنهان‌سازی دستورات SQL در یک کلاس 22](#_Toc191082895)

[2-17 حالت‌های تراکنشی و الگوی حالت 23](#_Toc191082896)

[2-18 طراحی یک تراکنش با الگوی دستور 27](#_Toc191082897)

[2-19 مادی‌سازی تنبل با پروکسی مجازی 30](#_Toc191082898)

[2-20 چگونه روابط جداول را نمایش دهیم؟ 32](#_Toc191082899)

[2-21 ابرکلاس شی پایدار و جداسازی دغدغه‌ها 33](#_Toc191082900)

[2-22 مسائل حل‌نشده 34](#_Toc191082901)

[فصل 3 35](#_Toc191082902)

[روش: پیاده‌سازی چارچوب پایداری با الگوها 35](#_Toc191082903)

[3-1 مقدمه 35](#_Toc191082904)

[3-2 ساخت طرح‌واره و درج رکوردهای پایگاه دادۀ مورد استفاده 35](#_Toc191082905)

[3-3 پیاده‌سازی چارچوب: بخش پایگاه داده 36](#_Toc191082906)

[3-4 پیاده‌سازی چارچوب: بخش مدل‌ها 41](#_Toc191082907)

[3-5 پیاده‌سازی چارچوب: بخش پایداری 43](#_Toc191082908)

[3-6 پیاده‌سازی چارچوب: بخش تراکنش‌ها 48](#_Toc191082909)

[3-7 پیاده‌سازی چارچوب: بخش اصلی 49](#_Toc191082910)

[فصل 4 51](#_Toc191082911)

[جمع‌بندی 51](#_Toc191082912)

[4-1 مقدمه 51](#_Toc191082913)

[4-2 ارزیابی عملکرد چارچوب پیشنهادی: چالش‌ها و راهکارها 51](#_Toc191082914)

[4-3 نتیجه‌گیری 52](#_Toc191082915)

[مراجع 53](#_Toc191082916)

[واژه نامه انگلیسی به فارسی 54](#_Toc191082917)

فهرست شکل‌ها

[شکل 1-2: نگاشت اشیا و جداول 8](#_Toc190362749)

[شکل 2-2: مثال پروفایل مدل‌سازی داده در UML 9](#_Toc190362750)

[شکل 3-2: ارتباط اشیا و رکوردها به‌واسطۀ شناسه‌های شی 10](#_Toc190362751)

[شکل 4-2: یک PersistenceFacade 11](#_Toc190362752)

[شکل 5-2: نگاشت‌گرهای پایگاه داده 13](#_Toc190362753)

[شکل 6-2: الگوی Template Method در یک چارچوب رابط کاربری گرافیکی 15](#_Toc190362754)

[شکل 7-2: Template Method برای اشیای نگاشت‌گر 16](#_Toc190362755)

[شکل 8-2: بازنویسی Hook Method 17](#_Toc190362756)

[شکل 9-2: بهینه‌سازی کد با استفادۀ مجدد از Template Method 18](#_Toc190362757)

[شکل 10-2: ساختار برخی از بسته‌ها و کلاس‌های چارچوب 19](#_Toc190362758)

[شکل 11-2: متدهای محافظت‌شده در UML 20](#_Toc190362759)

[شکل 12-2: نمودار حالت شی پایدار 23](#_Toc190362760)

[شکل 13-2: ارث‌بری کلاس‌های اشیای پایدار از PersistentObject 24](#_Toc190362761)

[شکل 14-2: متدهای commit و rollback 25](#_Toc190362762)

[شکل 15-2: اعمال الگوی حالت 26](#_Toc190362763)

[شکل 16-2: دستورات عملیات پایگاه داده 29](#_Toc190362764)

[شکل 17-2: پروکسی مجازی با Manufacturer 31](#_Toc190362765)

[شکل 18-2: ابرکلاس شی پایدار 33](#_Toc190362766)

[شکل 1-3: بخشی از اسکریپت SQL پایگاه دادۀ مورد استفاده جهت به‌کارگیری در میزکار MySQL 36](#_Toc190362767)

[شکل 2-3: پیاده‌سازی قسمت اتصال بخش پایگاه دادۀ چارچوب 37](#_Toc190362768)

[شکل 3-3: پیاده‌سازی قسمت عملیات بخش پایگاه دادۀ چارچوب 39](#_Toc190362769)

[شکل 4-3: پیاده‌سازی کلاس QueryBuilder چارچوب 40](#_Toc190362770)

[شکل 5-3: پیاده‌سازی کلاس PersistentObject چارچوب 41](#_Toc190362771)

[شکل 6-3: پیاده‌سازی کلاس‌های City و Country چارچوب 42](#_Toc190362772)

[شکل 7-3: پیاده‌سازی کلاس PersistenceFacade چارچوب 43](#_Toc190362773)

[شکل 8-3: پیاده‌سازی قسمت نمایشگر بخش پایداری چارچوب 45](#_Toc190362774)

[شکل 9-3: پیاده‌سازی قسمت حالت‌های بخش پایداری چارچوب 47](#_Toc190362775)

[شکل 10-3: پیاده‌سازی قسمت دستورات بخش تراکنش‌های چارچوب 48](#_Toc190362776)

[شکل 11-3: پیاده‌سازی قسمت تراکنش بخش تراکنش‌های چارچوب 49](#_Toc190362777)

[شکل 12-3: پیاده‌سازی بخش اصلی چارچوب 50](#_Toc190362778)

# فصل 1

# مقدمه

## 1-1 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 ارائۀ ساختار این چارچوب، مزایا، معایب، چالش‌ها و پیاده‌سازی آن با استفاده از فناوری‌های موجود می‌پردازند.

## 1-2 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 ارائۀ راهکارهایی برای حل این مشکلات، می‌تواند به عنوان یک راهکار مؤثر در سیستم‌های نرم‌افزاری مورد استفاده قرار گیرد.

## 1-3 اهداف پروژه

این پروژه به‌دنبال 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 مقایسۀ چارچوب پیشنهادی با روش‌های موجود از نظر کارایی و سازگاری است.

## 1-4 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، به‌کمک 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 2

# پیشینه: طراحی چارچوب پایداری با الگوها

## 2-1 مقدمه

برنامۀ NextGen[[1]](#footnote-1) مانند اکثر برنامه‌ها نیاز به ذخیره و بازیابی اطلاعات در یک مکانیزم ذخیره‌سازی پایدار، مانند پایگاه دادۀ رابطه‌ای[[2]](#footnote-2) دارد ]1[. این فصل به طراحی یک چارچوب برای ذخیره‌سازی اشیای پایدار می‌پردازد.

به‌طور کلی، بهتر است به‌جای توسعۀ یک چارچوب پایداری از ابتدا، از یک محصول آماده استفاده کرد یا خریداری نمود. این چارچوب می‌تواند به‌صورت یک محصول مستقل یا به‌عنوان بخشی از مدیریت پایداری درون‌محفظه‌ای برای Entity Bean‌ها[[3]](#footnote-3) در فناوری‌هایی مانند EJB[[4]](#footnote-4) و سایر فناوری‌های جاوا مورد استفاده قرار گیرد. توسعۀ یک سرویس پایداری O-R در سطح صنعتی ممکن است سال‌ها زمان ببرد و شامل چالش‌های پیچیده‌ای باشد که نیاز به تخصص ویژه دارند. علاوه‌بر این، فناوری‌هایی مانند JDO[[5]](#footnote-5) راهکارهای جزئی برای این مسئله ارائه می‌دهند.

بنابراین، هدف این فصل ارائۀ یک چارچوب پایداری در سطح صنعتی یا جایگزینی برای فناوری‌هایی مانند JDO نیست، بلکه از این چارچوب به‌عنوان ابزاری برای توضیح طراحی چارچوب‌ها با استفاده از الگوها استفاده می‌شود، زیرا این موضوع به‌عنوان یک مطالعۀ موردی مناسب مطرح است. همچنین، این فصل نمونۀ 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 به‌عنوان یک روش پیشنهادی برای طراحی سرویس پایداری در سطح صنعتی تلقی شود.

## 2-2 مشکل اشیای پایدار

فرض کنید در برنامۀ NextGen، داده‌های مربوط به ProductSpecification در یک پایگاه دادۀ رابطه‌ای ذخیره شده‌اند. هنگام اجرای برنامه، این داده‌ها باید به حافظۀ محلی بارگذاری شوند. اشیای پایدار به اشیایی گفته می‌شود که نیاز به ذخیره‌سازی پایدار دارند، مانند نمونه‌های ProductSpecification.

مکانیزم‌های ذخیره‌سازی و اشیای پایدار به‌صورت پایگاه داده‌های شی‌گرا، پایگاه‌ داده‌های رابطه‌ای و... هستند. در صورتی که از پایگاه دادۀ 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 گستردگی استفاده از پایگاه داده‌های رابطه‌ای، اغلب به‌جای پایگاه داده‌های شی‌گرا از این روش استفاده می‌شود. اما در این صورت، مشکلاتی به‌دلیل عدم تطابق بین مدل داده‌ای رابطه‌ای[[6]](#footnote-6) و مدل شی‌گرا[[7]](#footnote-7) به وجود می‌آید که در ادامه بررسی خواهند شد. برای حل این مشکل، به یک سرویس نگاشت شی-رابطه‌ای[[8]](#footnote-8) ویژه نیاز است. علاوه‌بر پایگاه داده‌های رابطه‌ای، گاهی ذخیره‌سازی اشیا در سایر فرمت‌ها و مکانیزم‌های ذخیره‌سازی مانند فایل‌های Flat[[9]](#footnote-9)، ساختارهای XML[[10]](#footnote-10)، فایل‌های PDB در سیستم‌عامل Palm[[11]](#footnote-11)، پایگاه داده‌های سلسله‌مراتبی[[12]](#footnote-12) و... مطلوب است. همانند پایگاه داده‌های رابطه‌ای، در این موارد نیز عدم تطابق بین اشیا و این فرمت‌های غیرشی‌گرا وجود دارد و برای کار با این فرمت‌ها، به سرویس‌های ویژه‌ای مشابه سرویس نگاشت شی-رابطه‌ای نیاز خواهد بود.

## 2-3 یک سرویس پایداری از چارچوب پایداری

یک چارچوب پایداری، مجموعه‌ای عمومی، قابل ‌استفادۀ مجدد و توسعه‌پذیر از انواع داده‌ها است که قابلیت‌هایی را برای پشتیبانی از اشیای پایدار فراهم می‌کند. یک سرویس پایدار (یا زیرسیستم پایداری[[13]](#footnote-13)) در واقع این خدمات را ارائه می‌دهد و با استفاده از یک چارچوب پایداری ایجاد می‌شود.

یک سرویس پایداری معمولاً برای کار با پایگاه دادۀ رابطه‌ای طراحی می‌شود، که در این صورت به آن سرویس نگاشت شی-رابطه‌ای 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 برنامۀ NextGen، سرویس پایداری به‌عنوان یک زیرسیستم در لایۀ خدمات فنی[[14]](#footnote-14) قرار می‌گیرد.

## 2-4 چارچوب‌ها

اگرچه ممکن است بیش‌ازحد ساده‌سازی شود، اما چارچوب را می‌توان مجموعه‌ای قابل توسعه از اشیا برای انجام عملکردهای مرتبط دانست. نمونۀ بارز آن، چارچوب‌های رابط کاربری گرافیکی[[15]](#footnote-15) مثل AWT یا Swing در جاوا است.

ویژگی شاخص یک چارچوب این است که پیاده‌سازی هستۀ 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.

به‌عنوان مثال، چارچوب رابط کاربری گرافیکی Swing در جاوا شامل کلاس‌ها و رابط‌هایی برای عملکردهای اصلی رابط کاربری گرافیکی است. توسعه‌دهندگان می‌توانند ابزارک‌ها[[16]](#footnote-16)ی تخصصی را با ارث‌بری (زیرکلاس‌گیری)[[17]](#footnote-17) از کلاس‌های Swing و بازنویسی[[18]](#footnote-18) برخی متدها ایجاد کنند. همچنین، آن‌ها می‌توانند با استفاده از الگوی ناظر[[19]](#footnote-19)، رفتار پاسخ به رویدادهای مختلف را برای ابزارک‌های از پیش تعریف‌شده (مانند JButton) تغییر دهند. این همان مفهوم چارچوب است.

به‌طور کلی، یک چارچوب، یک مجموعۀ منسجم از رابط‌ها[[20]](#footnote-20) و کلاس‌هاست که برای ارائۀ خدمات به بخش هسته‌ای و ثابت یک زیرسیستم منطقی با یکدیگر همکاری می‌کنند. چارچوب، شامل کلاس‌های انتزاعی[[21]](#footnote-21) و مشخص[[22]](#footnote-22) است که رابط‌های موردنیاز، نحوۀ تعامل اشیا و سایر قواعد را تعریف می‌کنند. چارچوب معمولاً (و نه همیشه) از کاربر چارچوب[[23]](#footnote-23) 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.»[[24]](#footnote-24) این بدان معناست که کلاس‌های تعریف‌شده توسط کاربر (مانند زیرکلاس‌های جدید) پیام‌هایی را از کلاس‌های از پیش تعریف‌شدۀ 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 استفادۀ مجدد[[25]](#footnote-25) را ارائه می‌دهند (بسیار بیشتر از کلاس‌های مستقل). بنابراین، اگر یک سازمان به افزایش قابلیت استفادۀ مجدد نرم‌افزارها علاقه‌مند باشد (و چه سازمانی نیست؟!)، باید بر توسعۀ چارچوب‌ها تأکید کند.

## 2-5 الزامات سرویس و چارچوب پایداری

برای نرم‌افزار NextGen POS[[26]](#footnote-26)، 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 را PFW[[27]](#footnote-27) می‌نامیم. PFW یک چارچوب ساده‌شده است[[28]](#footnote-28). این چارچوب باید قابلیت‌های ذخیره و بازیابی اشیا در یک مکانیزم ذخیرۀ 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.

## 2-6 مفاهیم کلیدی

برای بررسی بخش‌های بعدی به تعدادی مفهوم کلیدی نیاز است.

در این گزارش، می‌خواهیم نگاشتی بین کلاس و مخزن پایدار[[29]](#footnote-29) 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[[30]](#footnote-30)‌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 شناسۀ منحصربه‌فرد[[31]](#footnote-31) باشد.

یک نگاشت‌گر پایگاه داده از نوع ساختگی محض[[32]](#footnote-32) مسئول مادی‌سازی[[33]](#footnote-33) و غیرمادی‌سازی[[34]](#footnote-34) داده‌ها است. مادی‌سازی فرایند تبدیل یک نمایش غیرشی‌گرای داده‌ها (مثلاً رکوردها در یک مخزن پایدار) به اشیا است. غیرمادی‌سازی عملیات معکوس آن است که با نام غیرفعال‌سازی[[35]](#footnote-35) نیز شناخته می‌شود.

سرویس‌های پایداری برای افزایش عملکرد، اشیای مادی‌شده را در حافظۀ نهان[[36]](#footnote-36) ذخیره می‌کنند. آگاهی از حالت اشیا در ارتباط با تراکنش جاری مفید است. برای مثال، تشخیص این که کدام اشیا تغییر یافته‌اند[[37]](#footnote-37) کمک می‌کند تا مشخص شود آیا نیاز به ذخیرۀ مجدد در مخزن پایدار دارند یا نه.

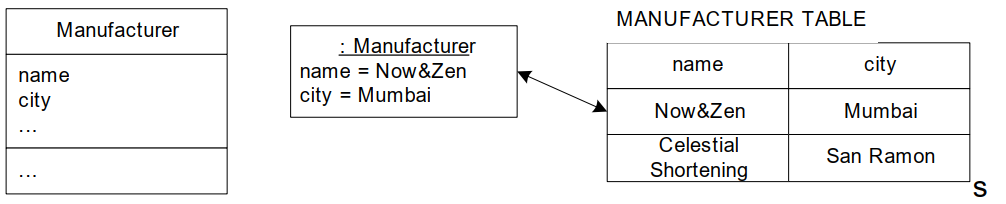
همچنین، از مفهوم مادی‌سازی تنبل[[38]](#footnote-38) در این گزارش استفاده می‌شود؛ بدین معنا که همۀ اشیا به‌طور هم‌زمان مادی نمی‌شوند؛ بلکه یک نمونۀ خاص فقط در صورت نیاز و به‌صورت درخواستی مادی می‌شود. مادی‌سازی تنبل را می‌توان با استفاده از یک مرجع هوشمند به نام پروکسی مجازی[[39]](#footnote-39) پیاده‌سازی کرد.

## 2-7 الگو: نمایش اشیا به‌عنوان جداول

چگونه یک شی را به یک رکورد یا طرح‌وارۀ پایگاه دادۀ رابطه‌ای نگاشت کنیم؟

الگوی نمایش اشیا به‌عنوان جداول پیشنهاد می‌کند که برای هر کلاس شی پایدار[[40]](#footnote-40)، یک جدول در پایگاه دادۀ رابطه‌ای تعریف شود. ویژگی‌های شی که شامل نوع داده‌های ابتدایی (مانند عدد، رشته و...) هستند، به ستون‌های جدول نگاشت می‌شوند.

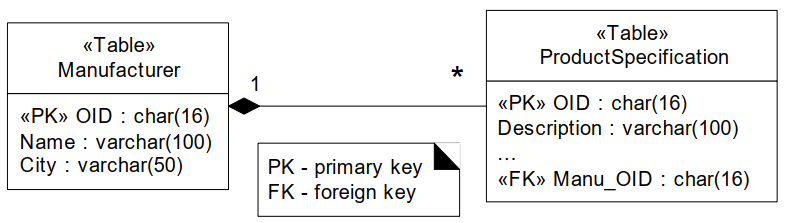
اگر یک شی فقط شامل ویژگی‌هایی از نوع داده‌های ابتدایی باشد، فرایند نگاشت ساده خواهد بود. اما همان‌طور که خواهیم دید، این مسئله همیشه به این سادگی نیست، زیرا اشیا ممکن است دارای ویژگی‌هایی باشند که به اشیای پیچیدۀ دیگر ارجاع دهند، درحالی‌که مدل رابطه‌ای نیازمند این است که مقادیر اتمی باشند (مطابق فرم 1NF). به شکل 1-2 توجه کنید.



شکل 1-2: نگاشت اشیا و جداول

## 2-8 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[[41]](#footnote-41)، مدل داده است که بخشی از رشتۀ طراحی[[42]](#footnote-42) محسوب می‌شود. شکل 2-2 برخی از نشانه‌گذاری‌های UML را برای مدل‌سازی داده نشان می‌دهد. در این شکل، یک محدودیت ارجاعی[[43]](#footnote-43) را مشاهده می‌کنیم، به این معنا که یک ردیف ProductSpecification نمی‌تواند بدون یک ردیف مرتبط Manufacturer وجود داشته باشد.



شکل 2-2: مثال پروفایل مدل‌سازی داده در UML

این استریوتایپ‌ها[[44]](#footnote-44) بخشی از هستۀ UML نیستند، بلکه یک افزونه محسوب می‌شوند. به‌طور کلی، UML دارای مفهومی به نام پروفایل UML است: یک مجموعۀ منسجم از استریوتایپ‌های UML، مقادیر برچسب‌گذاری‌شده و محدودیت‌ها برای یک هدف خاص. برخی موارد رایج در پروفایل UML که مثالی از آن در شکل 2-2 مشاهده می‌شود به OMG[[45]](#footnote-45) برای تأیید رسمی ارائه شده بودند.

## 2-9 الگو: شناسۀ شی

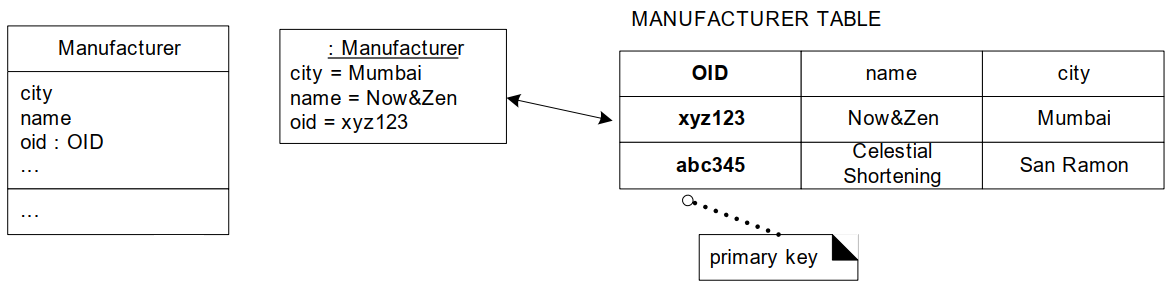
داشتن روشی سازگار برای ارتباط بین اشیا و رکوردها مطلوب است و باید اطمینان حاصل شود که بازتولید مکرر یک رکورد منجر به ایجاد اشیای تکراری نشود.

الگوی شناسۀ شی پیشنهاد می‌کند که یک شناسۀ 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 منحصربه‌فرد است. روش‌های مختلفی برای تولید شناسه‌های منحصربه‌فرد برای OIDها وجود دارد که از شناسه‌های منحصربه‌فرد در یک پایگاه دادۀ خاص تا شناسه‌های منحصربه‌فرد در سطح جهانی را شامل می‌شود، مانند تولیدکننده‌های توالی پایگاه داده و راهبرد تولید کلید High-Low.

در دنیای اشیا، یک OID توسط یک رابط یا کلاس OID نمایش داده می‌شود که مقدار واقعی و نحوۀ نمایش آن را در بر می‌گیرد. در پایگاه داده‌های رابطه‌ای، معمولاً به‌صورت یک مقدار متنی با طول ثابت ذخیره می‌شود.

هر جدول یک OID به‌عنوان کلید اصلی دارد و هر شی نیز (به‌طور 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 به‌طور منحصربه‌فرد به یک سطر در یک جدول خاص نگاشت شود. به شکل 3-2 نگاه کنید.



شکل 3-2: ارتباط اشیا و رکوردها به‌واسطۀ 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.

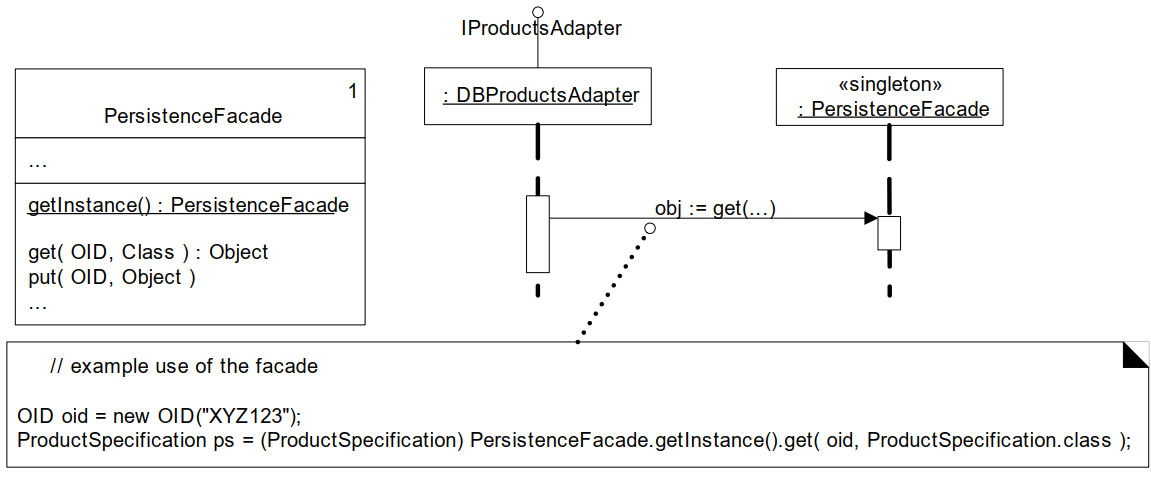
همچنین، OID یک نوع کلید سازگار برای استفاده در رابط سرویس پایداری فراهم می‌کند.

## 2-10 دسترسی به سرویس پایداری با استفاده از یک Facade

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

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

شکل 4-2 برخی از عملیات‌های Facade و نحوۀ استفاده از آن را در همکاری با یکی از آداپتورهای سرویس NextGen نشان می‌دهد. در این شکل، PersistenceFacade به‌عنوان یک شی Singleton عمل کرده و عملیات ذخیره‌سازی و بازیابی اشیا را مدیریت می‌کند. متد getInstance یک نمونه از PersistenceFacade را بازمی‌گرداند. متد get با دریافت یک شناسۀ شی و نوع کلاس موردنظر، شی مربوطه را بازیابی می‌کند. در این ساختار، DBProductsAdapter به‌عنوان یک آداپتور 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 4-2: یک PersistenceFacade

## 2-11 نگاشت اشیا: الگوی Database Mapper یا Database Broker

PersistenceFacade، مانند همه‌ی Facade‌ها، 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[[46]](#footnote-46) نامیده می‌شود. 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، معایب زیادی دارد و از نظر برنامه‌نویسی و نگهداری، به‌خوبی مقیاس‌پذیر نیست. این روش می‌تواند موجب اتصال[[47]](#footnote-47) قوی بین کلاس شی پایدار و دانش مربوط به ذخیره‌سازی پایدار شود که اصل کاهش اتصال را نقض می‌کند. از طرفی، مسئولیت‌های پیچیده در یک حوزۀ نامرتبط نسبت به وظایف اصلی کلاس، که اصل انسجام[[48]](#footnote-48) بالا و جداسازی دغدغه‌ها[[49]](#footnote-49) را نقض می‌کند. در این حالت، مسائل مربوط به سرویس‌های فنی با منطق برنامه ترکیب می‌شوند.

ما یک رویکرد کلاسیک نگاشت غیرمستقیم[[50]](#footnote-50) 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 حافظۀ 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 Broker در طراحی سیستم‌های توزیع‌شده معنای متفاوت و دیرینه‌ای دارد.[[51]](#footnote-51)

برای هر کلاس شی پایدار، یک کلاس نگاشت‌گر جداگانه تعریف می‌شود. شکل 5-2 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.

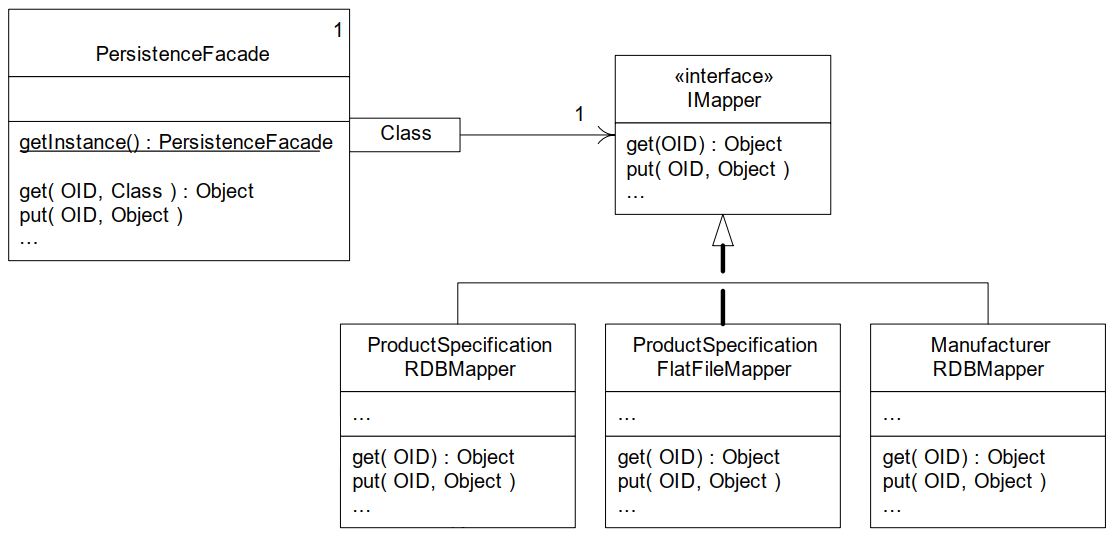
class PersistenceFacade{// ...public Object get(OID oid, Class persistenceClass){// an IMapper is keyed by the Class of the persistent object IMapper

mapper = (IMapper) mappers.get(persistenceClass);// delegatereturn mapper.get(oid);

}// ...

}

در شکل 5-2، یک ارتباط واجد شرایط[[52]](#footnote-52) 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[[53]](#footnote-53)). توجه کنید که در این نسخه از get، دیگر نیازی به ارسال Class به‌عنوان یک پارامتر نیست، زیرا کلاس برای یک نوع پایدار خاص به‌صورت ثابت[[54]](#footnote-54) تعیین شده است. هر نگاشت‌گر اشیا را به روش منحصربه‌فرد خود دریافت (get) و ذخیره (put) می‌کند، بسته به نوع و فرمت ذخیره‌سازی داده‌ها.



شکل 5-2: نگاشت‌گرهای پایگاه داده

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

رویکرد نگاشت‌گرهای مبتنی بر متاداده[[55]](#footnote-55) انعطاف‌پذیرتر اما پیچیده‌تر است. برخلاف طراحی دستی نگاشت‌گرهای جداگانه برای انواع مختلف اشیای پایدار، در این روش، نگاشت بین طرح‌وارۀ شی و طرح‌وارۀ دیگر (مثلاً رابطه‌ای) به‌صورت پویا[[56]](#footnote-56) و براساس متاداده‌ای که نگاشت را توصیف می‌کند، تولید می‌شود. به‌عنوان 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[[57]](#footnote-57) پشتیبانی می‌کنند، مانند C#، Java و Smalltalk امکان‌پذیر است، اما در زبان‌هایی مانند C++ که فاقد این قابلیت هستند، دشوار خواهد بود.

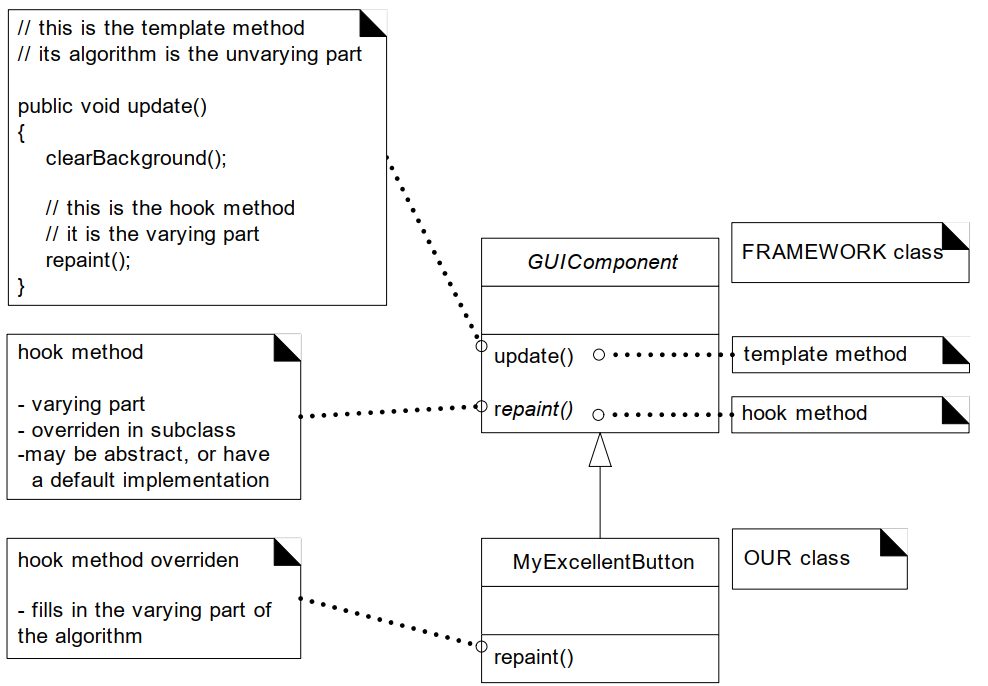
با استفاده از نگاشت‌گرهای مبتنی بر متاداده، می‌توان نگاشت طرح‌واره را بدون تغییر در کد منبع و تنها با تغییر در یک ذخیره‌ساز خارجی اعمال کرد. این امر، اصل تغییرات محافظت‌شده[[58]](#footnote-58) را در برابر تغییرات طرح‌واره تضمین می‌کند.

بااین‌حال، یک ویژگی مهم در این چارچوب این است که می‌توان هم از نگاشت‌گرهای کدنویسی‌شدۀ دستی و هم از نگاشت‌گرهای مبتنی بر متاداده استفاده کرد، بدون اینکه تغییری در کد کلاینت‌ها ایجاد شود (که این خود نوعی کپسوله‌سازی پیاده‌سازی[[59]](#footnote-59) محسوب می‌شود).

## 2-12 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[[60]](#footnote-60) می‌باشند.[[61]](#footnote-61) این الگو در قلب طراحی چارچوب‌ها قرار دارد[[62]](#footnote-62) و اکثر برنامه‌نویسان شی‌گرا، حتی اگر نام آن را ندانند، به‌طور عملی با آن آشنا هستند.

ایدۀ اصلی این الگو، تعریف یک Template Method در یک ابرکلاس[[63]](#footnote-63) 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[[64]](#footnote-64) بازنویسی کرد. به‌این‌ترتیب، زیرکلاس‌ها می‌توانند متدهای متغیر را بازنویسی کنند تا رفتار خاص خود را در نقاط تغییرپذیر الگوریتم اضافه نمایند (به شکل 6-2 نگاه کنید).



شکل 6-2: 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.

## 2-13 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 به‌صورت زیر است:

if (object in cache)

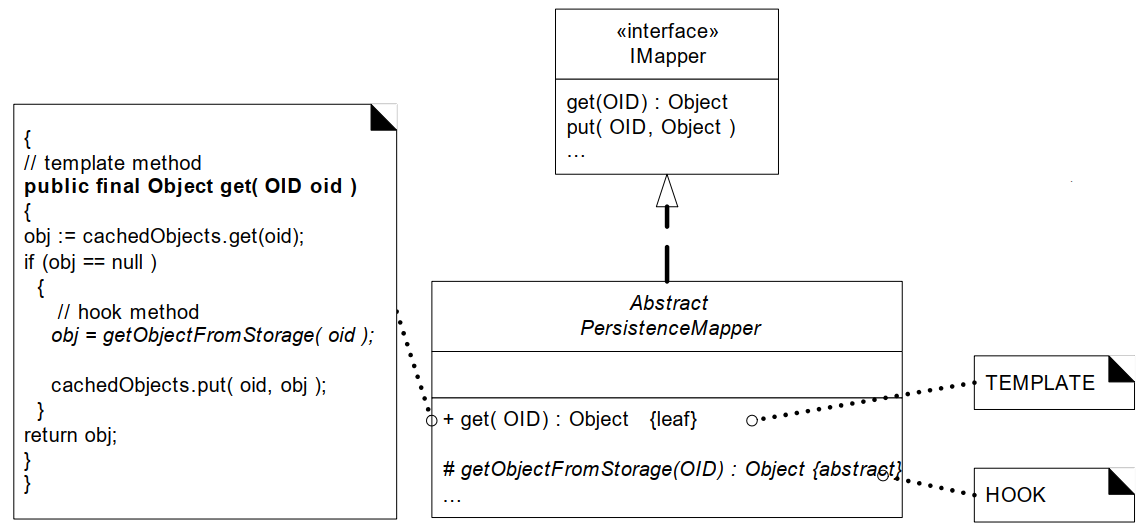
return it

else

create the object from its representation in storage  
save object in cache  
return it

نقطۀ تغییرپذیر این الگوریتم در چگونگی ایجاد شی از داده‌های ذخیره‌شده است.

برای پیاده‌سازی این الگو، متد get را به‌عنوان 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 7-2 طراحی اصلی این ساختار را نشان می‌دهد.

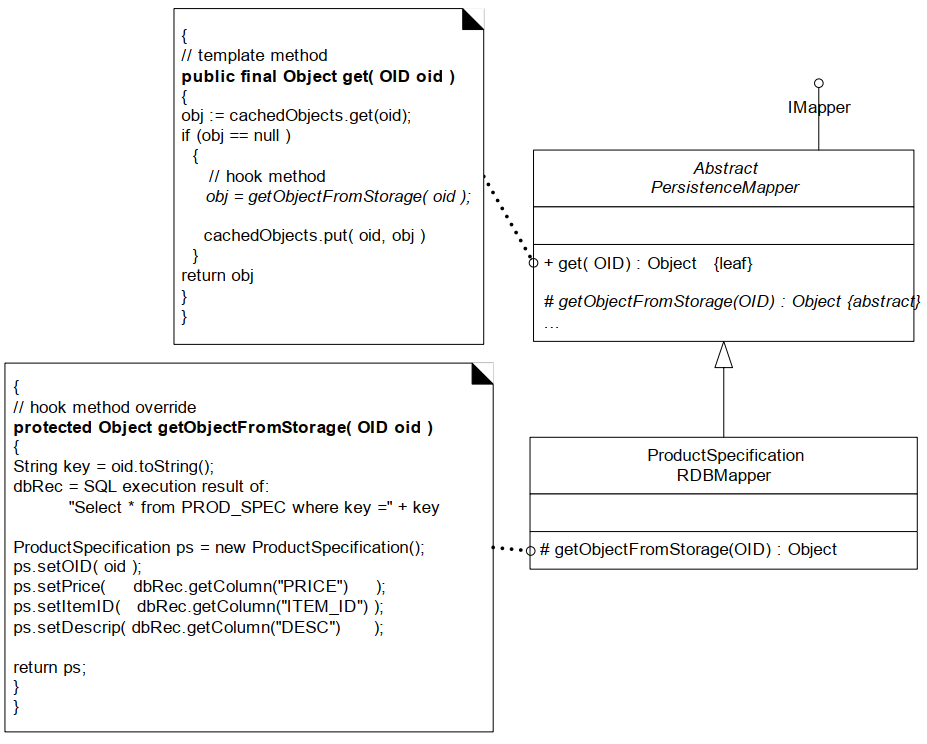


شکل 7-2: Template Method برای اشیای نگاشت‌گر

در شکل 7-2، نمادگذاری leaf استفاده شده است که برای عملیات و کلاس‌های نهایی[[65]](#footnote-65) یا برگ[[66]](#footnote-66) استفاده می‌شود. در نمادگذاری UML، # به‌معنای محافظت‌شده است؛ یعنی فقط برای زیرکلاس‌ها قابل مشاهده است.

همان‌طور که در این مثال نشان داده شده است، رایج است که Template Method عمومی[[67]](#footnote-67) باشد، درحالی‌که Hook Method به‌صورت محافظت‌شده تعریف شود. AbstractPersistenceMapper و IMapper 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 8-2 یک نمونه از این رویکرد را نشان می‌دهد.

اگر در پیاده‌سازی Hook Method در شکل 8-2 فرض کنیم که بخش اولیۀ الگوریتم (اجرای دستور SELECT در SQL) برای همه‌ی اشیا یکسان است و تنها نام جدول پایگاه داده تغییر می‌کند، پس بار دیگر می‌توان از الگوی Template Method برای جداسازی بخش‌های متغیر و ثابت الگوریتم استفاده کرد. در شکل 9-2 نکتۀ پیچیده این است که متد getObjectFromStorage در AbstractRDBMapper یک  
Hook Method نسبت به متد get در AbstractPersistenceMapper است، اما همین متد، یک Template Method نسبت به Hook Method جدید getObjectFromRecord محسوب می‌شود.

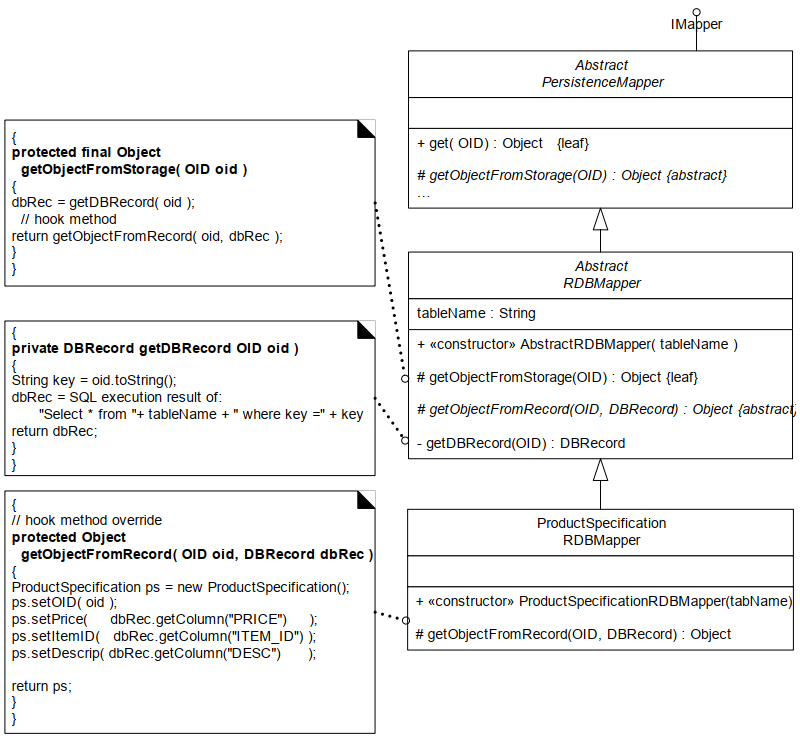


شکل 8-2: بازنویسی Hook Method

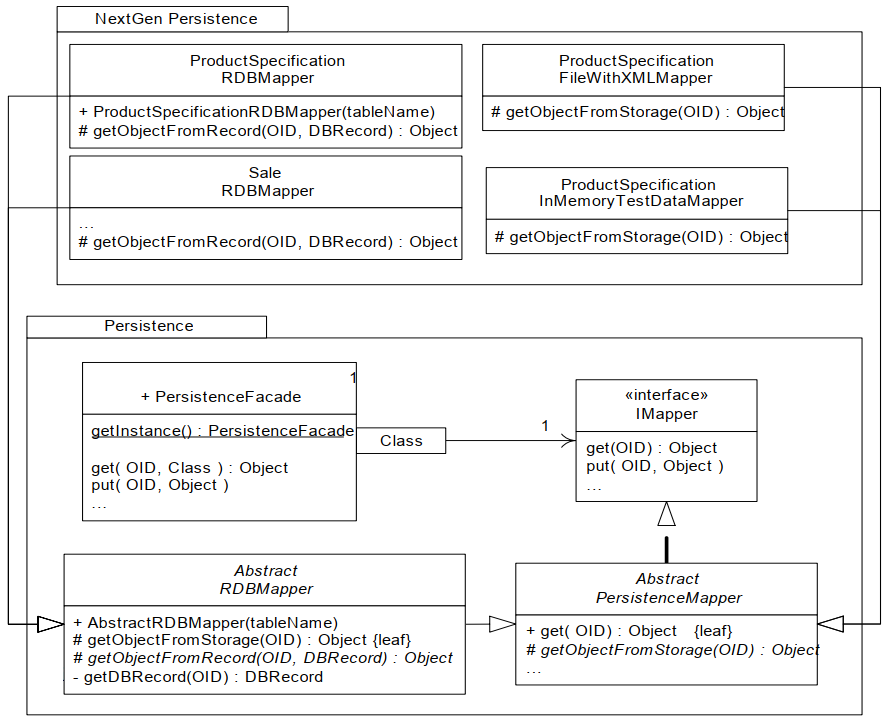
در شکل 9-2 توجه کنید که سازنده‌ها[[68]](#footnote-68) چگونه می‌توانند در UML تعریف شوند. استفاده از استریوتایپ اختیاری است و اگر نام سازنده با نام کلاس یکسان باشد، احتمالاً نیازی به آن نخواهد بود.

اکنون، کلاس‌های IMapper، AbstractPersistenceMapper و AbstractRDBMapper بخشی از چارچوب هستند. برنامه‌نویس 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 زنجیرۀ سازنده‌ها[[69]](#footnote-69) به AbstractRDBMapper ارسال شود).

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[[70]](#footnote-70) یا برای جداول و فایل‌های خاص جدید در یک مکانیزم ذخیره‌سازی موجود، سفارشی‌سازی کند. شکل 10-2 ساختار برخی از بسته‌ها[[71]](#footnote-71) و کلاس‌ها را نشان می‌دهد. نکتۀ قابل توجه این است که کلاس‌های خاص NextGen نباید در بستۀ عمومی سرویس‌های فنی مربوط به پایداری قرار بگیرند. این نمودار، همراه با شکل 9-2، ارزش یک زبان بصری مانند UML را برای توصیف بخش‌های نرم‌افزار نشان می‌دهد، زیرا اطلاعات زیادی را به‌صورت خلاصه منتقل می‌کند.



شکل 9-2: بهینه‌سازی کد با استفادۀ مجدد از Template Method



شکل 10-2: ساختار برخی از بسته‌ها و کلاس‌های چارچوب

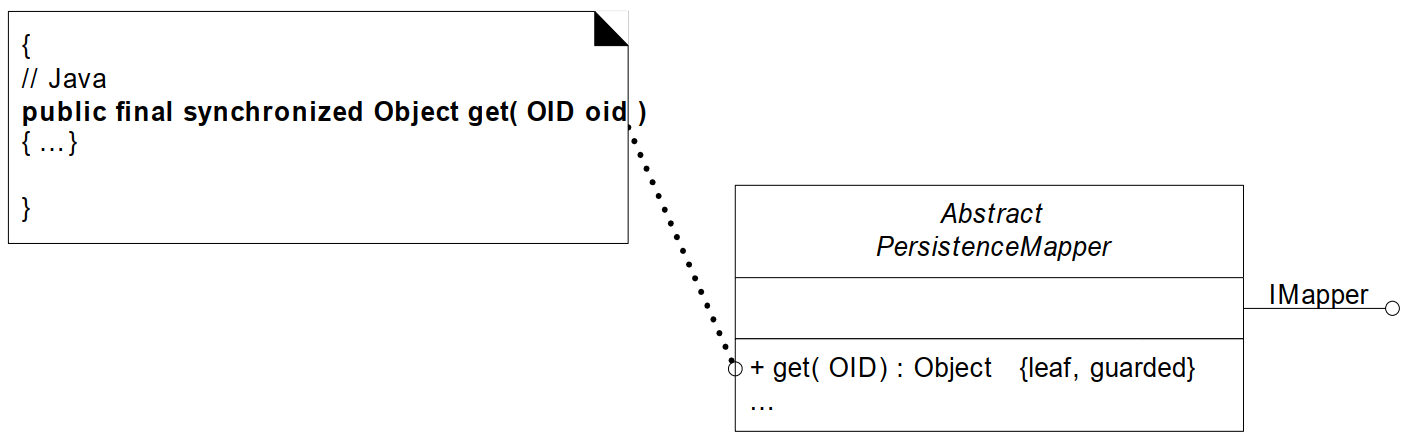
کلاس ProductSpecificationInMemoryTestDataMapper نمونه‌ای از نگاشت‌گرهای دادۀ آزمایشی در حافظه است. این‌گونه کلاس‌ها برای تست سیستم بدون نیاز به پایگاه دادۀ خارجی به‌کار می‌روند و اشیای از پیش تعیین‌شده[[72]](#footnote-72) را فراهم می‌کنند.

در فرایند یکپارچه، سند معماری نرم‌افزار[[73]](#footnote-73) به‌عنوان یک راهنمای یادگیری برای توسعه‌دهندگان آینده عمل می‌کند و دیدگاه‌های معماری کلیدی را ارائه می‌دهد. گنجاندن نمودارهایی مانند اشکال 9-2 و 10-2 در سند معماری نرم‌افزار پروژۀ NextGen، با هدف این سند که ارائۀ اطلاعات مهم معماری است، همسو می‌باشد.

متد get در AbstractPersistenceMapper شامل بخشی از کد است که ایمنی در برابر اجرای همزمان[[74]](#footnote-74) ندارد؛ زیرا ممکن است شی واحدی در چندین نخ به‌طور همزمان مادی‌سازی شود. از آنجا که سرویس پایداری به‌عنوان یک زیرسیستم خدمات فنی باید ایمن 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، به‌گونه‌ای که PersistenceFacade به یک شی سرور راه دور[[75]](#footnote-75) تبدیل شود و در آن چندین نخ به‌طور همزمان اجرا شده و به چندین مشتری سرویس دهند.

بنابراین، این متد باید کنترل هم‌زمانی[[76]](#footnote-76) داشته باشد. اگر از جاوا استفاده شود، اضافه کردن کلیدواژۀ synchronized می‌تواند این مشکل را حل کند.

شکل 11-2 در UML یک متد همگام‌سازی‌شده[[77]](#footnote-77) را نشان می‌دهد.



شکل 11-2: 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 مجموعۀ متدهای محافظت‌شدۀ یک شی اجرا شود.

## 2-14 پیکربندی نگاشت‌گرها به‌کمک MapperFactory

مشابه نمونه‌های قبلی از Factoryها در این مطالعۀ موردی، پیکربندی PersistenceFacade با مجموعه‌ای از IMapperها می‌تواند از طریق یک شی کارخانه‌ای (MapperFactory) انجام شود. بااین‌حال، 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() { … }

}

زیرا این روش از اصل تغییرات محافظت‌شده 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 پیاده‌سازی‌شده با HashMap) به‌عنوان کلید، از کلاس مربوط به انواع پایدار استفاده می‌کند و مقدار هر کلید یک IMapper متناظر است.

سپس، Facade می‌تواند مجموعه IMapperهای خود را به‌صورت زیر مقداردهی کند:

class PersistenceFacade {

private java.util.Map mappers = MapperFactory.getInstance().getAllMappers();

}

MapperFactory می‌تواند با استفاده از یک طراحی داده‌محور[[78]](#footnote-78) مجموعه‌ای از IMapperها را مقداردهی کند، به این معنا که می‌تواند با خواندن ویژگی‌های سیستم[[79]](#footnote-79)، کلاس‌های IMapper موردنظر را شناسایی و نمونه‌سازی کند. اگر زبان از قابلیت‌های برنامه‌نویسی بازتابی پشتیبانی کند، می‌توان نام کلاس‌ها را به‌عنوان رشته خواند و از متدی مانند Class.newInstance برای نمونه‌سازی استفاده کرد. بنابراین، مجموعۀ نگاشت‌گرها می‌تواند بدون تغییر در کد منبع مجدداً پیکربندی شود.

## 2-15 الگو: مدیریت حافظۀ نهان

نگه‌داری اشیای مادی‌شده در یک حافظۀ نهان محلی[[80]](#footnote-80) به‌منظور بهبود عملکرد و پشتیبانی از عملیات مدیریت تراکنش (مانند commit) مطلوب است، زیرا فرایند مادی‌سازی نسبتاً کند است.

الگوی مدیریت حافظۀ نهان پیشنهاد می‌کند که مسئولیت مدیریت حافظۀ نهان به‌عهدۀ نگاشت‌گرهای پایگاه‌ داده باشد. درصورتی‌که برای هر کلاس از اشیای پایدار یک نگاشت‌گر جداگانه استفاده شود، هر نگاشت‌گر می‌تواند حافظۀ نهان مخصوص خود را مدیریت کند.

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

## 2-16 یکپارچه‌سازی و پنهان‌سازی دستورات SQL در یک کلاس

قرار دادن[[81]](#footnote-81) دستورات SQL در کلاس‌های مختلف نگاشت‌گر پایگاه دادۀ رابطه‌ای اشکال بزرگی محسوب نمی‌شود، اما می‌توان آن را بهبود بخشید. فرض کنید به‌جای آن، یک کلاس ساختگی محض واحد (و به‌‌‌صورت Singleton) به نام RDBOperations وجود داشته باشد که تمام عملیات SQL (مانند SELECT، INSERT و...) در آن یکپارچه شده باشند. کلاس‌های نگاشت‌گر پایگاه دادۀ رابطه‌ای برای دریافت یک رکورد یا مجموعه‌ای از رکوردهای پایگاه داده (مثلاً 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) { ... }

}

به‌این‌ترتیب، 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 متمرکز شده و کلاس‌های مختلف می‌توانند به‌صورت ماژولار و بدون کدنویسی مجدد از آن استفاده کنند.

در نتیجۀ 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 متمرکز کردن تمام دستورات SQL در یک کلاس، یافتن و بهینه‌سازی آن‌ها برای یک متخصص SQL بسیار آسان‌تر می‌شود.

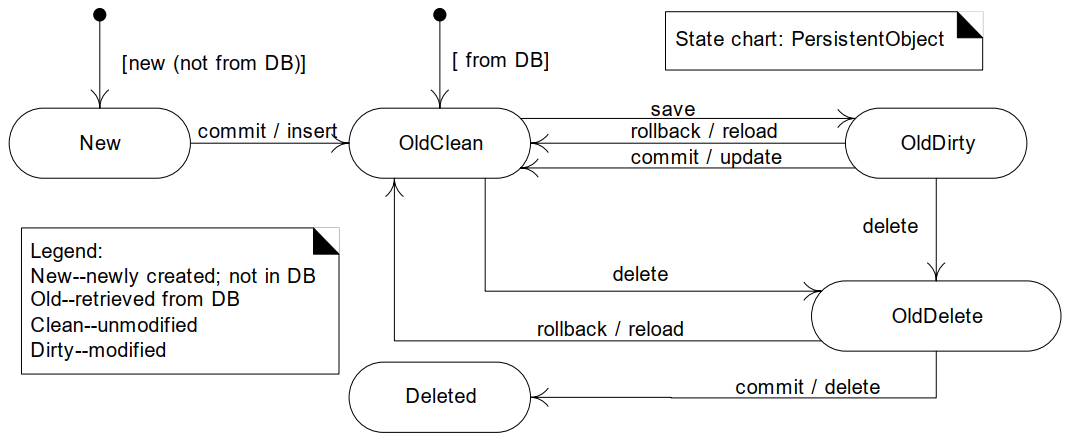
همچنین، با این کار، روش و جزئیات دسترسی کپسوله‌سازی می‌شود؛ برای مثال، به‌جای قرار دادن SQL، می‌توان آن را با یک Stored Procedure در پایگاه داده جایگزین کرد. همچنین، می‌توان از رویکردی پیشرفته‌تر مبتنی بر متاداده استفاده کرد که در آن SQL به‌صورت پویا از یک توصیف متادادۀ خارجی تولید شود.

به‌عنوان یک معمار نرم‌افزار، نکتۀ جالب این تصمیم طراحی این است که تحت‌تأثیر مهارت‌های توسعه‌دهنده قرار گرفته است. در اینجا، یک معامله بین انسجام بالا و راحتی برای یک متخصص انجام شده است. همۀ تصمیمات طراحی صرفاً به دلایل مهندسی نرم‌افزار مانند اتصال و انسجام گرفته نمی‌شوند، بلکه عوامل دیگری نیز در آن تأثیرگذار هستند.

## 2-17 حالت‌های تراکنشی و الگوی حالت[[82]](#footnote-82)

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 (مثلاً 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. به‌عنوان مثال، واکنش‌ها ممکن است مشابه حالاتی باشند که در نمودار حالت[[83]](#footnote-83) شکل 12-2 نشان داده شده است.



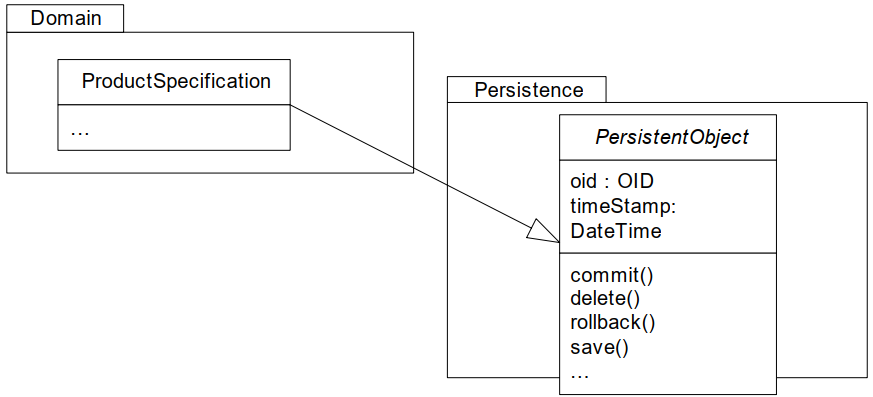
شکل 12-2: نمودار حالت شی پایدار

به‌عنوان 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.

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

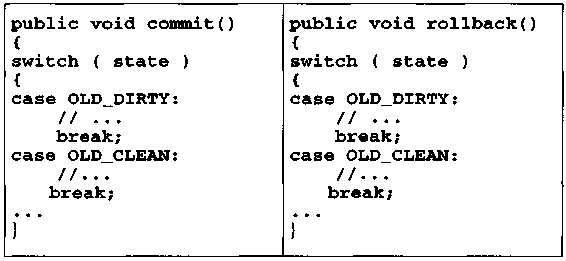
از نظر UML، این یک مثال عالی است که نشان می‌دهد چگونه یک نمودار حالت می‌تواند اطلاعاتی را که در قالب‌های دیگر بیان آن دشوار است، به‌طور خلاصه و مؤثر منتقل کند.

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.[[84]](#footnote-84) برای نمونه، شکل 13-2 را ببینید.



شکل 13-2: 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[[85]](#footnote-85) براساس کد حالت تراکنشی نیاز دارند. متدهای commit و rollback در هر حالت، عملیات متفاوتی انجام می‌دهند، اما ساختارهای منطقی مشابهی دارند.



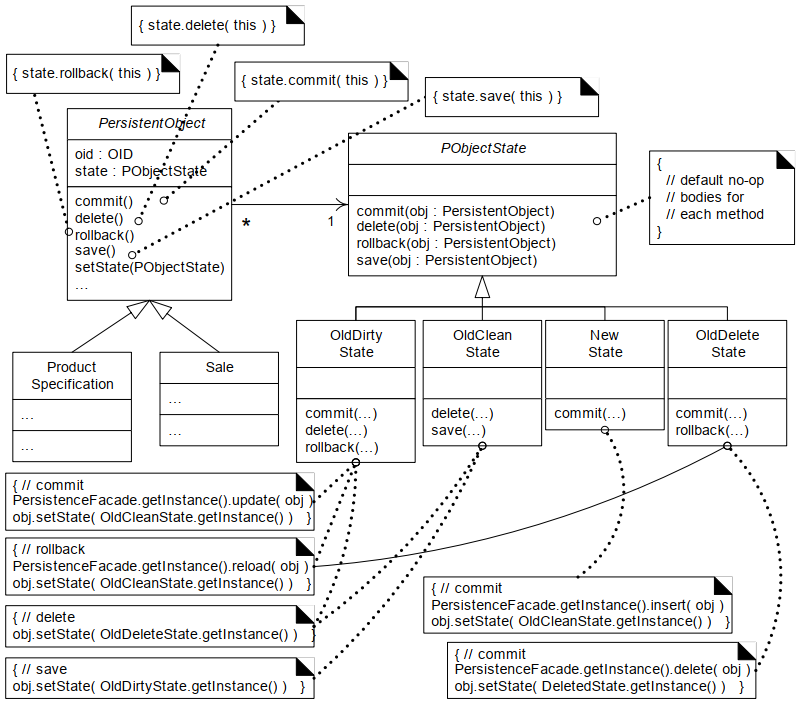
شکل 14-2: متدهای commit و rollback

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.

راه‌حل 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[[86]](#footnote-86) 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 15-2 کاربرد این الگو را در زیرسیستم پایداری نشان می‌دهد.



شکل 15-2: اعمال الگوی حالت[[87]](#footnote-87)

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 به‌صورت ارث‌بری‌شده هیچ‌کاری انجام نمی‌دهد اجرا می‌شود، چراکه شی تغییری نکرده و نیازی به ذخیره‌سازی مجدد ندارد.

توجه کنید که در شکل 15-2، کلاس‌های حالت و رفتار آن‌ها مطابق با نمودار حالت در شکل 12-2 هستند. الگوی حالت یکی از مکانیزم‌های پیاده‌سازی یک مدل انتقال حالت[[88]](#footnote-88) در نرم‌افزار است[[89]](#footnote-89) و باعث می‌شود که یک شی در پاسخ به رویدادها بین حالات مختلف جابجا شود.

به‌عنوان یک نکتۀ عملکردی، جالب است که این اشیای حالت در واقع بدون حالت[[90]](#footnote-90) هستند، یعنی هیچ ویژگی ندارند. بنابراین، نیازی به ایجاد چندین نمونه از یک کلاس حالت وجود ندارد؛ هر کلاس حالت می‌تواند Singleton باشد. به‌عنوان مثال، هزاران شی پایدار می‌توانند به یک نمونه واحد از OldDirtyState ارجاع دهند.

## 2-18 طراحی یک تراکنش با الگوی دستور[[91]](#footnote-91)

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[[92]](#footnote-92) است، مجموعه‌ای از وظایف که باید همگی با موفقیت انجام شوند، یا هیچ‌کدام انجام نشوند. به عبارت دیگر، تراکنش باید اتمی[[93]](#footnote-93) باشد.

در چارچوب سرویس پایداری، وظایف یک تراکنش شامل درج، به‌روزرسانی و حذف اشیا می‌شود. به‌عنوان 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.

حال فرض کنید یک تراکنش شامل دو وظیفه[[94]](#footnote-94) باشد، عملیات درج برای افزودن یک رکورد به جدول B و عملیات به‌روزرسانی برای به‌روزرسانی رکوردی در جدول A. اگر عملیات به‌روزرسانی زودتر از درج اجرا شود، یک خطای یکپارچگی ارجاعی رخ خواهد داد.

راه‌حل این است که وظایف پایگاه داده را مرتب کنیم. برخی از مشکلات ترتیب اجرا وابسته به طرح‌وارۀ پایگاه داده هستند، اما یک استراتژی کلی این است که ابتدا درج‌ها، سپس به‌روزرسانی‌ها 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)، صف‌بندی، تأخیر در اجرا، ثبت گزارش[[95]](#footnote-95) یا لغو[[96]](#footnote-96) نیاز دارند؟

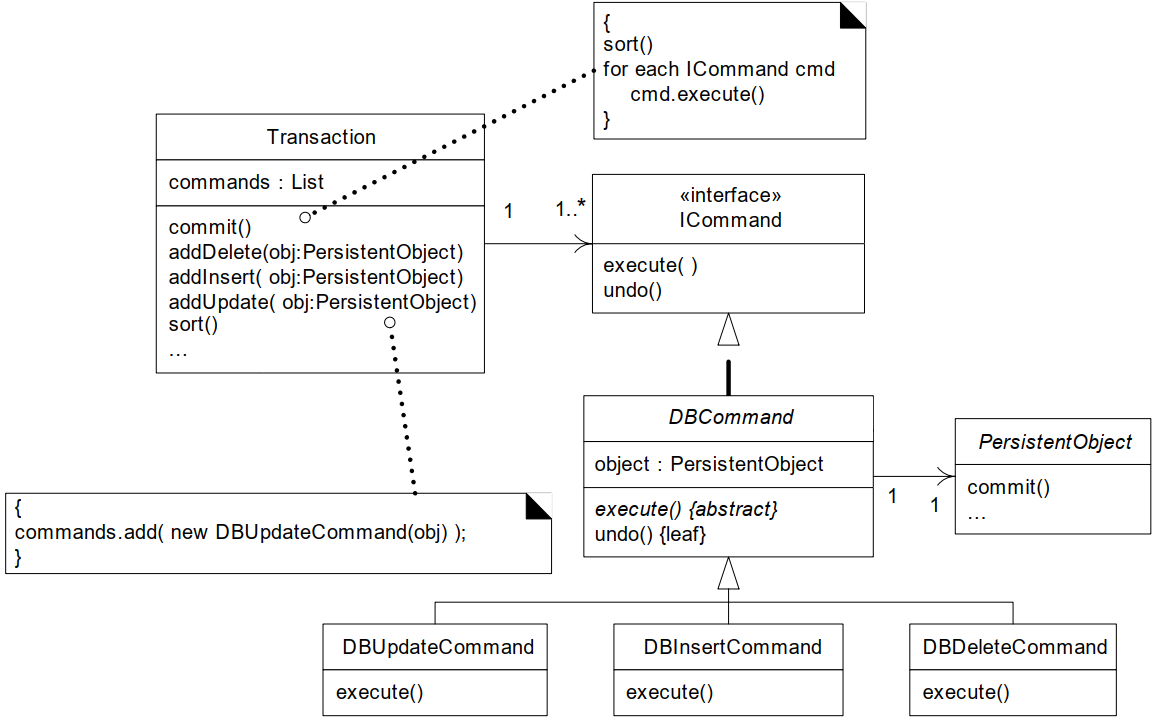
راه‌حل این است که هر وظیفه را به یک کلاس تبدیل کنیم که یک رابط مشترک را پیاده‌سازی می‌کند.

این یک الگوی ساده اما بسیار کاربردی است؛ زیرا عملیات به اشیا تبدیل می‌شوند و در نتیجه قابل مرتب‌سازی، ثبت، صف‌بندی و ... خواهند بود. به‌عنوان مثال، در چارچوب شی‌گرا برای پایداری، شکل 16-2 کلاس‌های دستورات مربوط به عملیات پایگاه داده را نشان می‌دهد.

اجرای کامل یک راهکار برای تراکنش‌ها شامل جزئیات بیشتری است، اما ایدۀ اصلی این بخش این است که هر وظیفه یا عملیات در تراکنش را به‌عنوان یک شی با یک متد چندریختی[[97]](#footnote-97) به نام execute نمایش دهیم. این رویکرد انعطاف‌پذیری زیادی ایجاد می‌کند، زیرا با تبدیل درخواست‌ها به اشیا، امکان مدیریت بهتر آن‌ها فراهم می‌شود.

یک نمونۀ کلاسیک از الگوی دستور، مدیریت عملیات در رابط کاربری گرافیکی مانند برش[[98]](#footnote-98) و چسباندن[[99]](#footnote-99) است؛ مثلاً متد execute در CutCommand، عملیات برش را انجام می‌دهد یا متد undo در CutCommand، عملیات برش را بازگشت می‌دهد. CutCommand همچنین داده‌های لازم برای انجام عملیات بازگشت را حفظ می‌کند. تمامی فرمان‌های رابط کاربری گرافیکی در یک پشتۀ تاریخچه[[100]](#footnote-100) 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 تحویل داده می‌شود، که می‌تواند دستورات را صف‌بندی، ثبت، اولویت‌بندی و اجرا کند.



شکل 16-2: دستورات عملیات پایگاه داده

در این مثال، undo هیچ عملی انجام نمی‌دهد (no-op)، اما یک راهکار پیچیده‌تر می‌تواند یک متد چندریختی undo به هر زیرکلاس اضافه کند، که به‌طور منحصر‌به‌فرد 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 16-2، شاید execute به‌سادگی با فراخوانی object.commit انجام شود، اما هر شی دستور می‌تواند اقدامات منحصربه‌فرد خود را اجرا کند.

## 2-19 مادی‌سازی تنبل با پروکسی مجازی

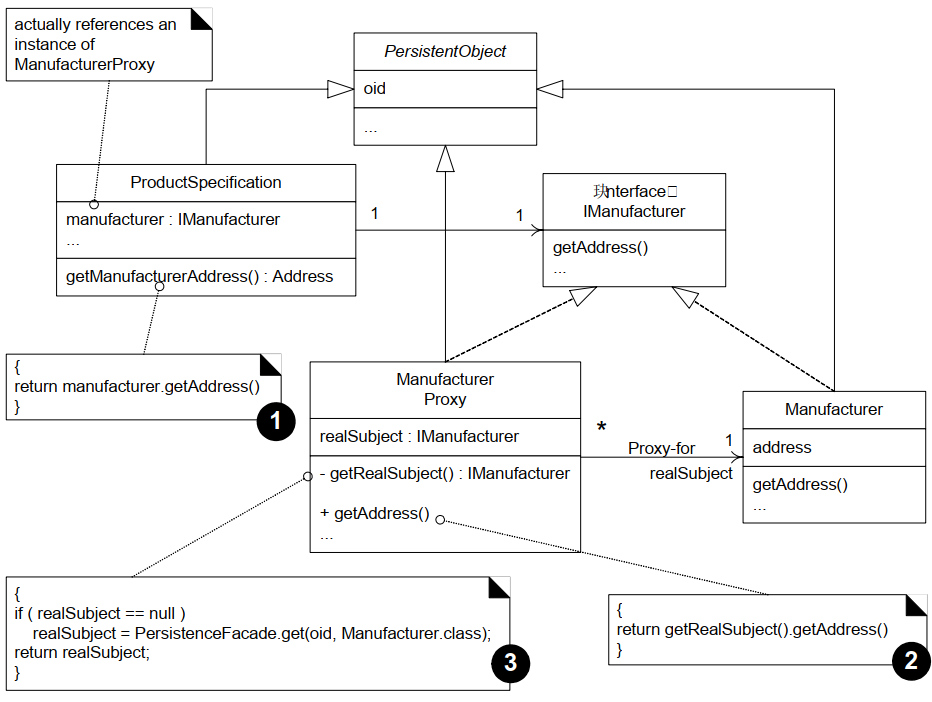
گاهی اوقات، به‌تعویق انداختن[[101]](#footnote-101) مادی‌سازی یک شی تا زمانی که واقعاً لازم باشد، مطلوب است، که معمولاً برای بهبود عملکرد انجام می‌شود. مثلاً فرض کنید اشیای ProductSpecification به یک شی Manufacturer ارجاع دارند، اما در بیشتر موارد نیازی به بازیابی Manufacturer از پایگاه داده وجود ندارد. تنها در سناریوهای خاصی مانند تخفیف‌های تولیدکننده[[102]](#footnote-102)، اطلاعات سازنده مانند نام و آدرس موردنیاز خواهد بود.

به‌تعویق انداختن بازیابی اشیای فرزند تا زمانی که واقعاً موردنیاز باشند، مادی‌سازی تنبل 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 تنبل را پیاده‌سازی می‌کند. پروکسی، یک شی سبک‌وزن[[103]](#footnote-103) است که به‌جای شی واقعی قرار می‌گیرد، تا زمانی که واقعاً لازم شود.

مثالی مشخص از الگوی پروکسی مجازی با ProductSpecification و Manufacturer در شکل 17-2 نشان داده شده است. این طراحی بر این فرض استوار است که پروکسی‌ها OID (شناسۀ 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 پیامی با نام getAddress به ManufacturerProxy 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.



شکل 17-2: پروکسی مجازی با Manufacturer

پیاده‌سازی پروکسی مجازی بسته به زبان برنامه‌نویسی متفاوت است. جزئیات این پیاده‌سازی خارج از محدودۀ این فصل است، اما به خلاصه‌ای از آن در ادامه می‌پردازیم. در C++، یک کلاس اشاره‌گر هوشمند[[104]](#footnote-104) به‌صورت 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 به‌صورت دستی انجام نمی‌شود. در عوض، یک تولیدکنندۀ کد ایجاد می‌شود که کلاس‌های مربوط به موضوع (مانند Manufacturer) را تحلیل کرده و سپس IManufacturer و ProxyManufacturer را تولید می‌کند. یک جایگزین دیگر در جاوا استفاده از Dynamic Proxy API است. در Smalltalk، یک Virtual Morphing Proxy (یا Ghost Proxy) تعریف می‌شود که از متدهایی برای تبدیل به موضوع واقعی استفاده می‌کند. در این روش نیز نیازی به تعریف IManufacturer نیست.

در شکل 17-2 مشاهده می‌شود که ManufacturerProxy برای مادی‌سازی موضوع واقعی خود با PersistenceFacade همکاری می‌کند. اما چه کسی ManufacturerProxy را ایجاد می‌کند؟ پاسخ، کلاس Database Mapper برای ProductSpecification است. کلاس نگاشت‌گر وظیفه دارد هنگام مادی‌سازی یک شی تصمیم بگیرد که کدام‌ یک از اشیای فرزند آن باید به‌صورت مادی‌سازی پیش‌دستانه[[105]](#footnote-105) و کدام باید به‌صورت مادی‌سازی تنبل با پروکسی بارگذاری شوند.

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

مثالی از مادی‌سازی پیش‌دستانه به‌صورت 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 تنبل به این‌صورت 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;

}

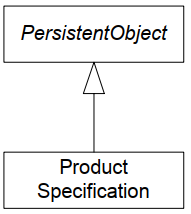
}

## 2-20 چگونه روابط جداول را نمایش دهیم؟

کد در بخش قبلی به یک کلید خارجی به نام MANU\_OID در جدول PRODUCT\_SPEC متکی است تا به رکوردی در جدول Manufacturer پیوند بخورد. این موضوع سؤالی را برجسته می‌کند: چگونه روابط اشیا در مدل رابطه‌ای نمایش داده می‌شوند؟ پاسخ این سوال در الگوی نمایش روابط اشیا به‌صورت 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 رابطه را ثبت کند. برای ارتباطات یک‌به‌چند، مانند یک مجموعه، یک جدول واسط که OIDهای هر شی در رابطه را ثبت می‌کند ایجاد کنیم. برای ارتباطات چندبه‌چند نیز همین کار را انجام دهیم. این روش‌ها به ما امکان می‌دهند تا روابط بین اشیا را در مدل رابطه‌ای به‌طور مؤثر نمایش دهیم.

## 2-21 ابرکلاس شی پایدار و جداسازی دغدغه‌ها

در طراحی نرم‌افزار، یک راه‌حل متداول برای ارائۀ قابلیت پایداری در اشیا، ایجاد یک ابرکلاس خدمات فنی انتزاعی به نام PersistentObject است که تمام اشیای پایدار از آن ارث‌بری می‌کنند (شکل 18-2 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 18-2: ابرکلاس شی پایدار

طراحی شکل 18-2 امکان‌پذیر است، اما از نظر اتصال و ترکیب نگرانی فنی مربوط به پایداری با منطق کاربردی یک شی دامنه، مشکل‌ساز است.

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

از سوی دیگر، جداسازی دغدغه‌ها یک فضیلت مطلق نیست که باید به هر قیمتی رعایت شود. همان‌طور که در مقدمۀ تغییرات محافظت‌شده بحث شد، طراحان باید روی نقاطی تمرکز کنند که به‌طور واقعی باعث بی‌ثباتی پرهزینه می‌شوند. اگر در یک برنامۀ خاص، گسترش کلاس‌ها از PersistentObject به یک راه‌حل 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.

## 2-22 مسائل حل‌نشده

این یک مقدمۀ بسیار مختصر به مشکلات و راه‌حل‌های طراحی در یک چارچوب و سرویس پایداری بود. بسیاری از مسائل مهم به‌طور سطحی بیان شده‌اند، ازجمله غیرمادی‌سازی اشیا[[106]](#footnote-106)، مادی‌سازی و غیرمادی‌سازی مجموعه‌ها، اجرای پرس‌وجو[[107]](#footnote-107) 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).

# فصل 3

# روش: پیاده‌سازی چارچوب پایداری با الگوها

## 3-1 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 مدیریت‌کنندۀ پایگاه‌داده، حالت برای مدیریت حالت اشیا و دستور برای اجرای عملیات به‌صورت 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 ایجادکنندۀ پرس‌وجو[[108]](#footnote-108) برای ساخت پویای پرس‌وجو‌های SQL استفاده شده است که امکان فیلتر کردن، مرتب‌سازی و محدود کردن نتایج را فراهم می‌کند. این چارچوب به‌گونه‌ای طراحی شده است که قابلیت اتصال به چندین نوع پایگاه‌داده را داشته باشد و به‌راحتی قابل توسعه برای موجودیت‌های دیگر باشد.

## 3-2 ساخت طرح‌واره و درج رکوردهای پایگاه دادۀ مورد استفاده

در این پروژه به‌منظور آزمایش نگاشت شی-رابطه‌ای و عملیات درج، به‌روزرسانی و حذف، نیاز به یک پایگاه داده داریم. پایگاه دادۀ مورد استفاده، بخشی از پایگاه دادۀ sakila (فقط جداول city و country) است که کدهای ساخت طرح‌واره و درج رکوردهای جداول آن‌ها برداشته شده اند. شکل 1-3 نمایی از بخشی از اسکریپت SQL مربوط به این پایگاه داده جهت استفاده در میزکار[[109]](#footnote-109) MySQL نشان می‌دهد.



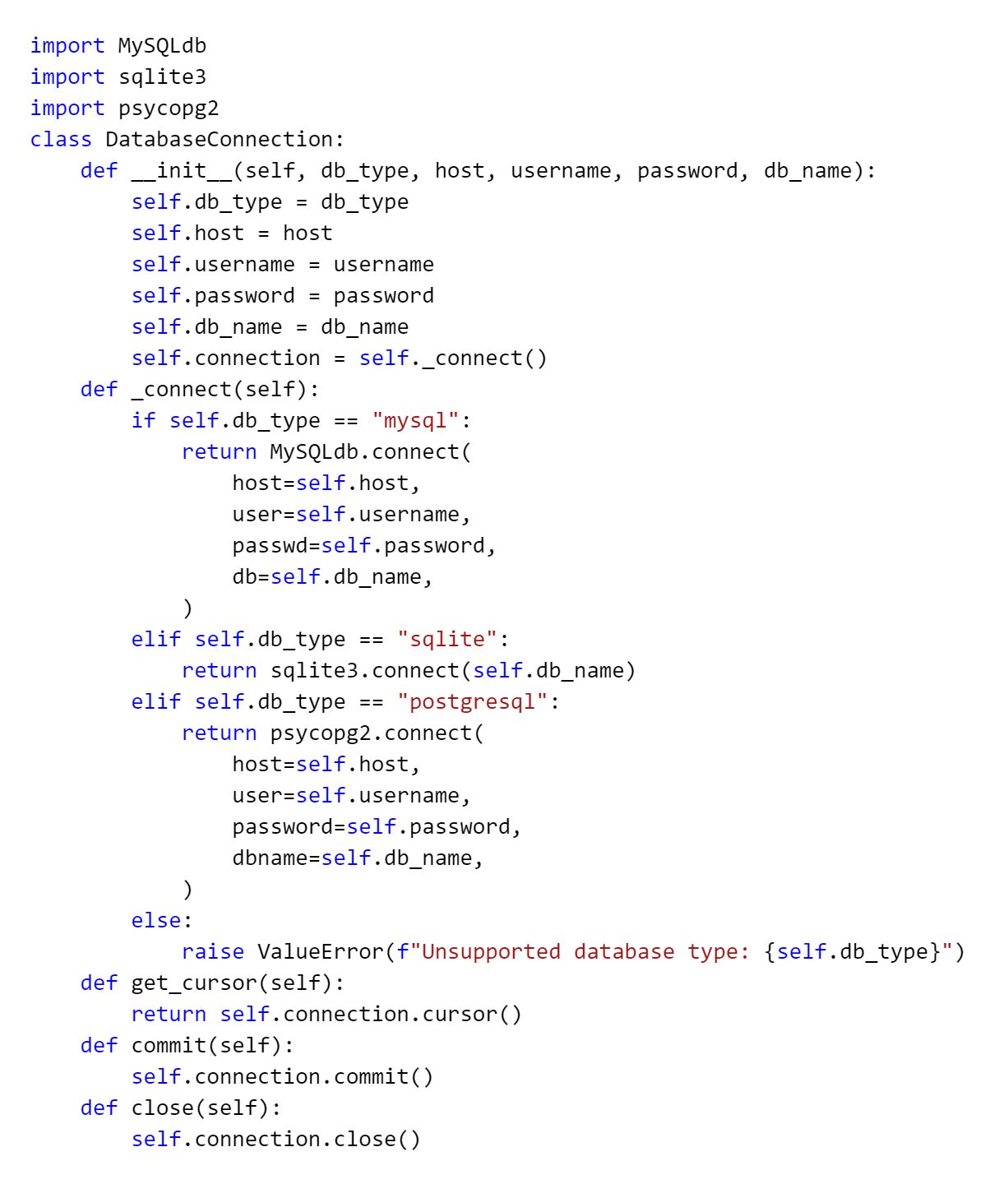
شکل 1-3: بخشی از اسکریپت SQL پایگاه دادۀ مورد استفاده جهت به‌کارگیری در میزکار MySQL

همان‌طور که مشاهده می‌شود، جدول city دارای ستون‌های city\_id (کلید اصلی)، city، country\_id (کلید خارجی 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، 600 رکورد و به جدول country، 109 رکورد اضافه کردیم. حال، این دو جدول آمادۀ انجام نگاشت شی-رابطه‌ای و عملیات درج، به‌روزرسانی و حذف هستند.

## 3-3 پیاده‌سازی چارچوب: بخش پایگاه داده

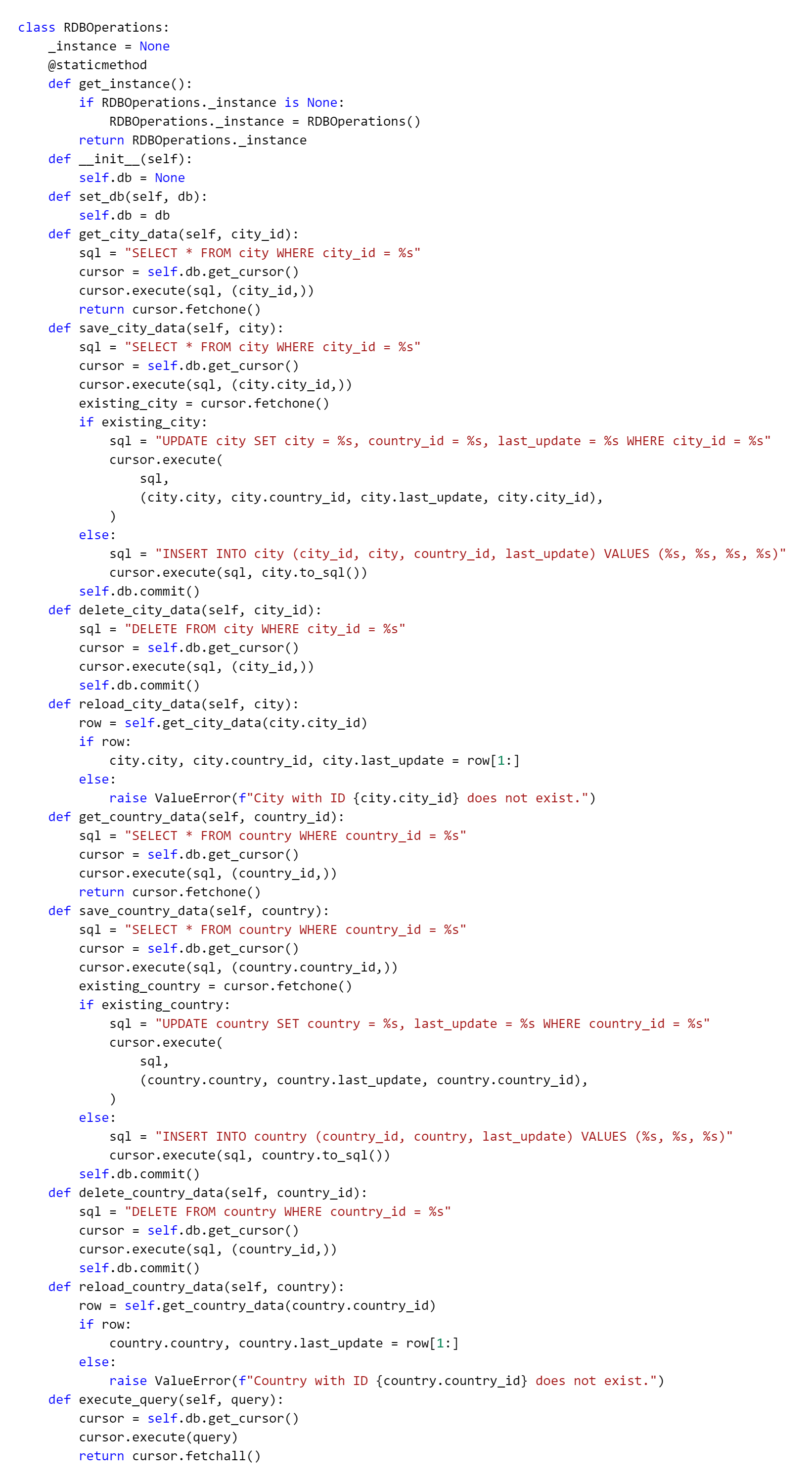
در این قسمت به پیاده‌سازی بخشی از چارچوب که به پایگاه داده مرتبط است می‌پردازیم.

در ابتدا در قسمت اتصال، کلاسی به نام DatabaseConnection تعریف می‌کنیم که امکان اتصال به سه نوع پایگاه دادۀ مختلف (MySQL، SQLite و PostgreSQL) را فراهم می‌کند. برای این کار، نیاز به کتابخانه‌های MySQLdb، sqlite3 و psycopg2 (به‌ترتیب برای MySQL، SQLite و PostgreSQL) داریم. کلاس تعریف‌‌شده یک متد سازنده دارد که هنگام ایجاد نمونه‌ای از کلاس فراخوانی می‌شود. این متد، ورودی‌های db\_type (نوع پایگاه داده که می‌تواند mysql، sqlite و یا postgresql باشد، host (آدرس سرور پایگاه داده)، username (نام کاربری برای احراز هویت)، password (رمز عبور برای احراز هویت) و db\_name (نام پایگاه داده) را دریافت می‌کند.[[110]](#footnote-110) سپس، متد \_connect به‌کمک ویژگی connection برای برقراری اتصال به پایگاه داده فراخوانی می‌شود. این متد، بسته به نوع پایگاه داده، اتصال مناسب را ایجاد می‌کند. 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 ValueError ایجاد می‌کنیم. متد get\_cursor، یک cursor برمی‌گرداند که برای اجرای دستورات SQL در پایگاه داده استفاده می‌شود. متد commit، جهت ذخیرۀ تغییرات داده‌ها (درج، به‌روزرسانی و حذف) استفاده می‌شود. متد close برای بستن اتصال به پایگاه داده به کار گرفته می‌شود تا منابع آزاد شوند. به پیاده‌سازی انجام‌شده در شکل 2-3 توجه کنید.



شکل 2-3: پیاده‌سازی قسمت اتصال بخش پایگاه دادۀ چارچوب

در ادامه به بررسی قسمت عملیات می‌پردازیم. در این قسمت، کلاس RDBOperations تعریف شده‌است 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 نمونۀ این کلاس را ذخیره کرده باشیم. متد get\_instance، یک متد ایستا[[111]](#footnote-111)ست که تضمین می‌کند همیشه فقط یک نمونه از کلاس RDBOperations داشته باشیم. این متد، اگر نمونه‌ای قبلاً ایجاد نشده باشد، آن را می‌سازد و در دفعات بعدی، همان نمونۀ ایجادشده را برمی‌گرداند. متد سازندۀ این کلاس، مقدار اولیۀ db را به None تنظیم می‌کند. db قرار است یک نمونه از کلاس DatabaseConnection باشد که قبلاً تعریف شده است. متد set\_db به کلاس اجازه می‌دهد اتصال به پایگاه داده را دریافت کند و در متغیر self.db نگه دارد. متد get\_city\_data اطلاعات یک شهر خاص را از جدول city جست‌وجو کرده و برمی‌گرداند. از fetchone استفاده شده تا فقط یک رکورد برگردانده شود. متد save\_city\_data ابتدا همان عملیات جست‌وجو را انجام می‌دهد. اگر شهر موردنظر از قبل در پایگاه داده وجود داشته باشد، رکورد آن به‌روزرسانی می‌شود؛ وگرنه رکورد درج می‌شود. در هر دو حالت، تغییرات در پایگاه داده commit می‌شوند. متد delete\_city\_data شهر مشخص‌شده را از جدول city حذف می‌کند. متد reload\_city\_data اطلاعات جدید را از پایگاه داده خوانده و در شی city به‌روزرسانی می‌کند. اگر اطلاعاتی برای این city\_id وجود نداشته باشد، یک خطای ValueError برمی‌گرداند. متدهای دیگر یعنی get\_country\_data، save\_country\_data، delete\_country\_data و reload\_country\_data نیز به‌صورت مشابه برای country عمل می‌کنند. متد execute\_query برای اجرای یک پرس‌وجوی سفارشی در پایگاه داده استفاده می‌شود. این متد، نتیجۀ اجرای پرس‌وجو را برمی‌گرداند. به شکل 3-3 توجه کنید.



شکل 3-3: پیاده‌سازی قسمت عملیات بخش پایگاه دادۀ چارچوب

در ادامه، کلاس QueryBuilder برای ساختن دستورات SQL به‌صورت پویا طراحی شده است. این کلاس به کاربران اجازه می‌دهد شرایط فیلتر، ترتیب، محدودیت و اتصال جداول را به‌صورت 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 ذخیرۀ شرایط WHERE)، order\_by (ذخیرۀ دستور  
ORDER BY برای مرتب‌سازی)، limit (مقدار محدودیت تعداد نتایج LIMIT) و joins (لیستی برای ذخیرۀ اتصال جداول) است که در متد سازنده، مقادیر اولیه تنظیم شده اند. این کلاس دارای متد filter است که یک شرط فیلتر به لیست filters اضافه می‌کند. این متد، ورودی‌های column (ستون موردنظر)، operator (عملگر مقایسه‌ای مثل IN، LIKE و...) و value (مقدار موردنظر برای مقایسه) است و به‌عنوان خروجی، شی self را برمی‌گرداند تا امکان زنجیره‌ای فراخوانی کردن متدها فراهم شود. متد order\_by\_column مقدار order\_by را تنظیم می‌کند. این متد به‌عنوان ورودی، column (نام ستونی که باید مرتب شود) و ascending (صعودی یا نزولی بودن مرتب‌سازی) را دریافت می‌کند. متد limit\_results، تعداد نتایج بازگشتی را محدود می‌کند. متد join، یک JOIN به دستور SQL اضافه می‌کند. این متد به‌عنوان ورودی، table (نام جدول دیگر که باید متصل شود) و on\_clause (شرطی که دو جدول را به هم متصل می‌کند) دریافت می‌کند. متد build رشتۀ SQL نهایی را تولید و برمی‌گرداند. به شکل 4-3 توجه کنید.

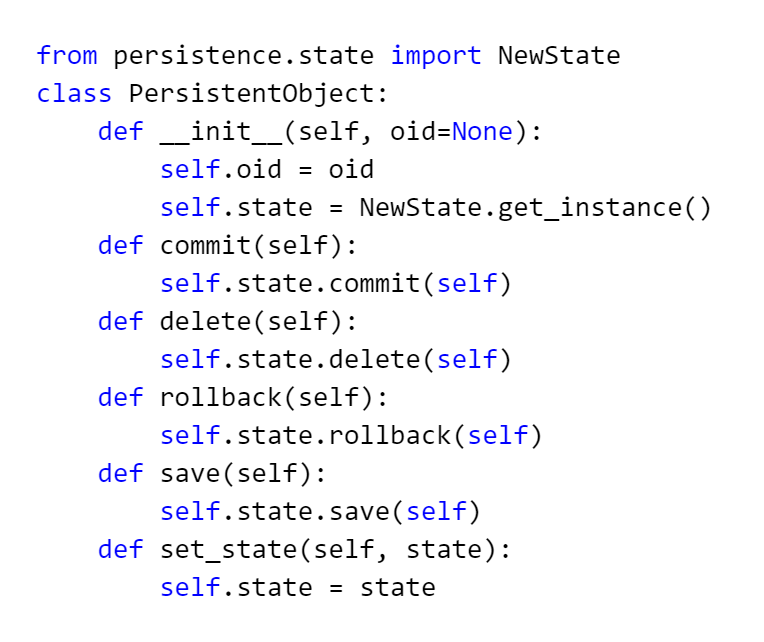


شکل 4-3: پیاده‌سازی کلاس QueryBuilder چارچوب

## 3-4 پیاده‌سازی چارچوب: بخش مدل‌ها

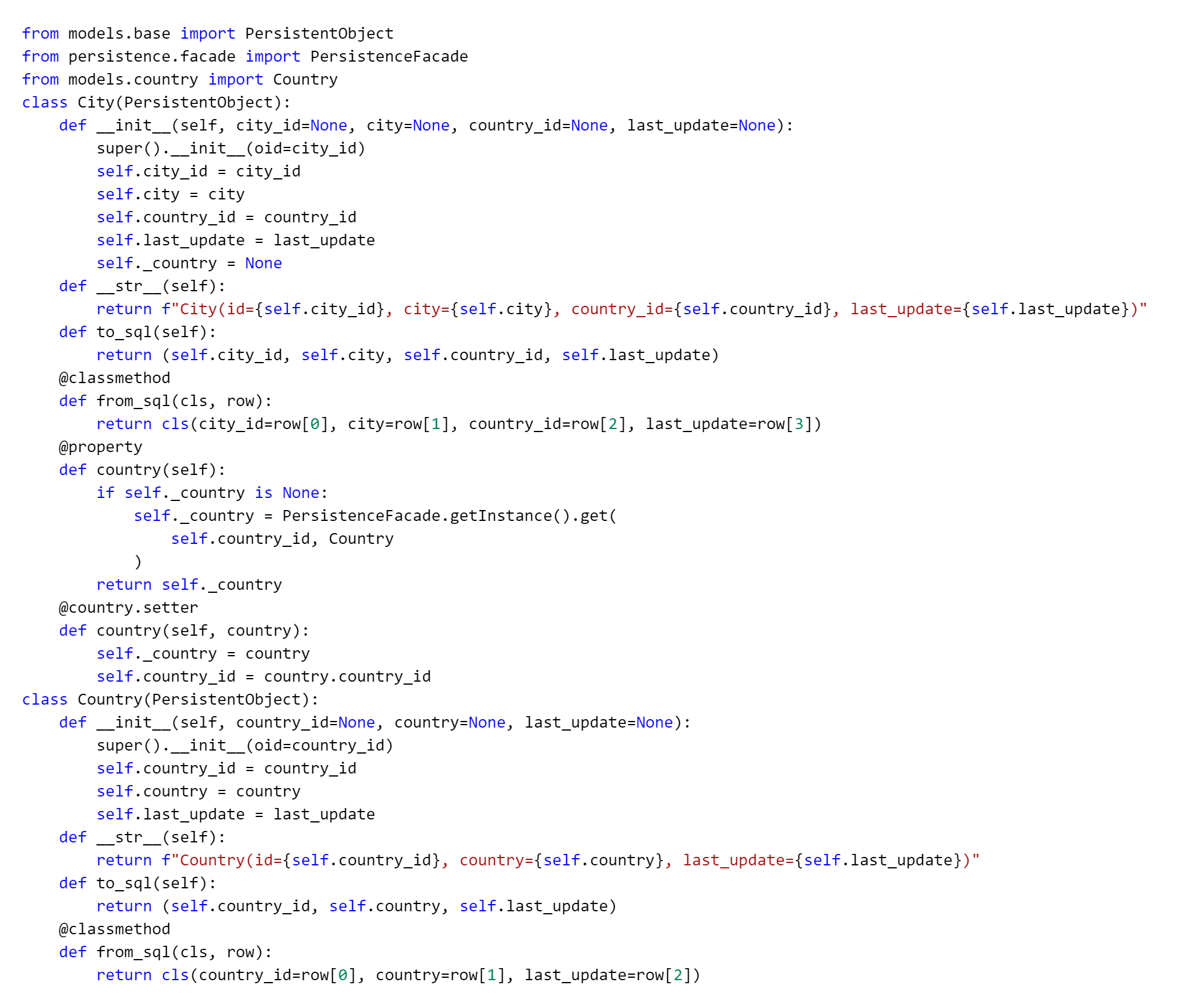
در این بخش، مدل‌های چارچوب را در چند قسمت پیاده‌سازی می‌کنیم.

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 متد get\_instance دریافت می‌شود که نشان‌دهندۀ 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 ذخیرۀ 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 5-3 توجه کنید.



شکل 5-3: 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. متد \_\_str\_\_ نمایش متنی و خوانا از شی را ارائه می‌دهد و هنگام دیباگ کردن داده‌ها و چاپ اطلاعات شهر استفاده می‌شود. متد to\_sql داده‌های شی را به فرمت مناسب برای ذخیره‌سازی در پایگاه داده تبدیل می‌کند. این متد هنگام درج در پایگاه داده کاربرد دارد. متد 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[[112]](#footnote-112) class method استفاده کرده و در خود به کلاس دسترسی داشته است. متد country که با دکوراتور property تعریف شده (به این معنا که هنگام فراخوانی آن، نیازی به گذاشتن پرانتزها نداریم)، رابطۀ 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.[[113]](#footnote-113) از الگوی طراحی Singleton برای دریافت داده‌های کشور استفاده شده است. متد setter جهت تنظیم \_country و country\_id نوشته شده است. کلاس Country نیز با تعدادی متد که شبیه متدهای کلاس City هستند تعریف می‌شود. پیاده‌سازی کلاس‌های City و Country در شکل 6-3 قابل مشاهده است.

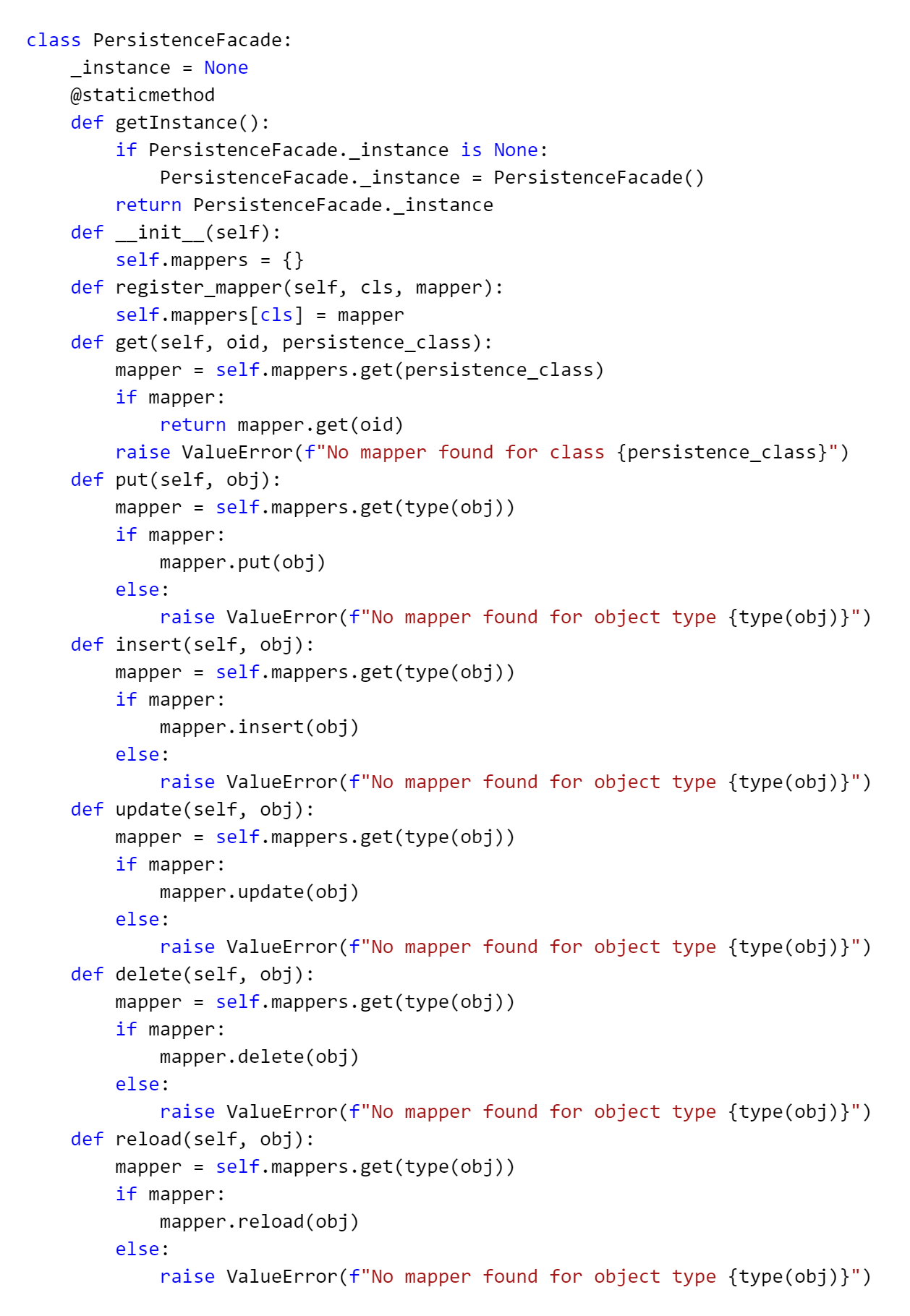


شکل 6-3: پیاده‌سازی کلاس‌های City و Country چارچوب

## 3-5 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 واسط 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. متد get ابتدا بررسی می‌کند 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 این‌صورت، خطا ایجاد می‌شود. به‌طور مشابه، متدهای put، insert، update، delete و reload پیاده‌سازی شدند. به‌طور کلی، این کلاس نقش یک لایۀ میانی بین مدل‌ها و پایگاه داده را ایفا می‌کند و کار با داده‌ها را تسهیل می‌کند. به شکل 7-3 توجه کنید.



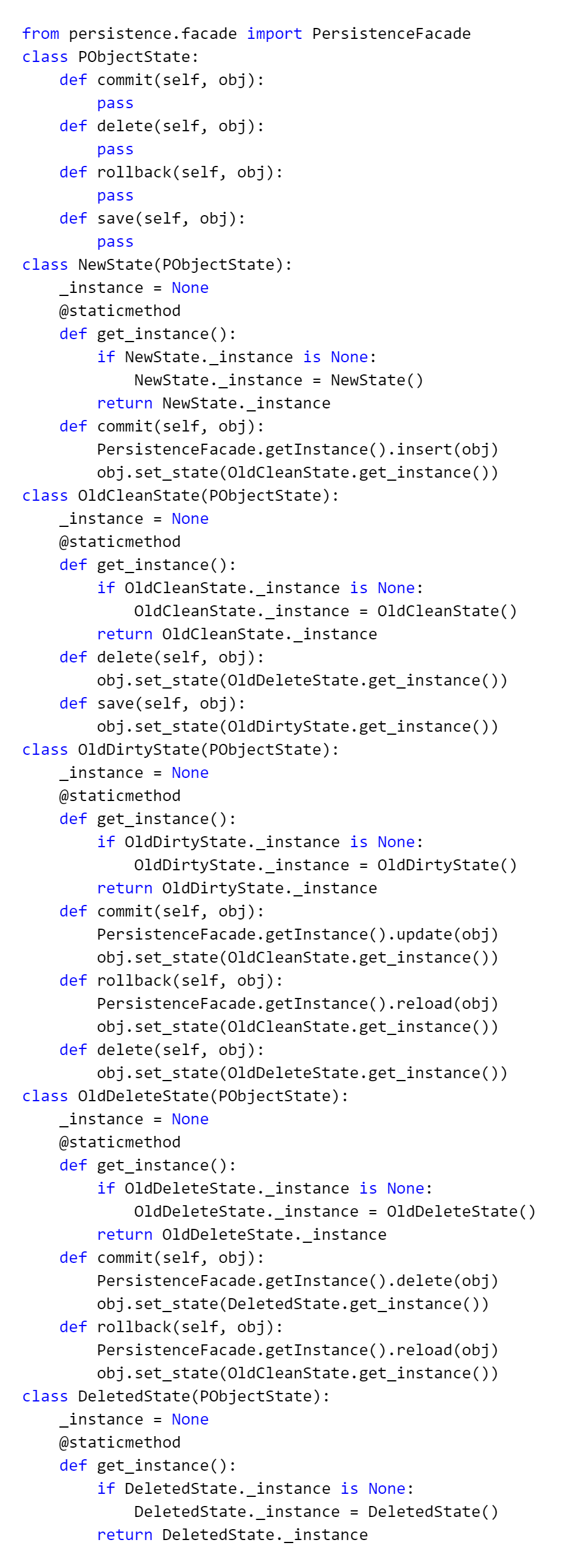
شکل 7-3: پیاده‌سازی کلاس PersistenceFacade چارچوب

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 متد get و put را تعریف می‌کند. این متدها در کلاس‌های فرزند باید پیاده‌سازی شوند. کلاس AbstractPersistenceMapper یک کلاس انتزاعی است که از IMapper ارث‌بری کرده و یک حافظۀ نهان به همراه قفل[[114]](#footnote-114) برای مدیریت همزمانی دارد. این کلاس دو ویژگی cached\_objects (یک دیکشنری برای نگهداری اشیای واقع در حافظۀ نهان جهت جلوگیری از درخواست‌های اضافه به پایگاه داده) و lock (برای جلوگیری از مشکل هم‌روندی[[115]](#footnote-115) در مواقعی که چندین نخ به داده‌ها دسترسی دارند) دارد. مقدار اولیۀ lock در متد سازنده برابر خروجی threading.Lock قرار می‌گیرد. متد get ابتدا بررسی می‌کند که آیا شی مورد نظر در حافظۀ نهان موجود است یا خیر. اگر نباشد، از متدی که در کلاس‌های فرزند پیاده‌سازی خواهد شد داده را از پایگاه داده دریافت می‌کند. اگر داده از پایگاه داده دریافت شود، آن را در حافظۀ نهان ذخیره می‌کند. متد put در کلاس‌های فرزند باید پیاده‌سازی شود. کلاس AbstractRDBMapper از AbstractPersistenceMapper ارث‌بری کرده و عملیات مربوط به پایگاه داده رابطه‌ای را مدیریت می‌کند. این کلاس دارای ویژگی‌ rdb\_operations (شی از RDBOperations برای مدیریت ارتباط با پایگاه داده) است. کلاس CityMapper مسئول ارتباط با پایگاه داده برای اشیای City است. این کلاس متدی دارد که داده‌های مربوط به شهر را از پایگاه داده دریافت می‌کند. اگر داده یافت شود، متد from\_sql از کلاس City را برای تبدیل داده به شی City فراخوانی می‌کند. متدهای CRUD نیز پیاده‌سازی می‌شوند. کلاس CountryRDBMapper متدهایی به‌صورت مشابه دارد. به شکل 8-3 توجه کنید.



شکل 8-3: پیاده‌سازی قسمت نمایشگر بخش پایداری چارچوب

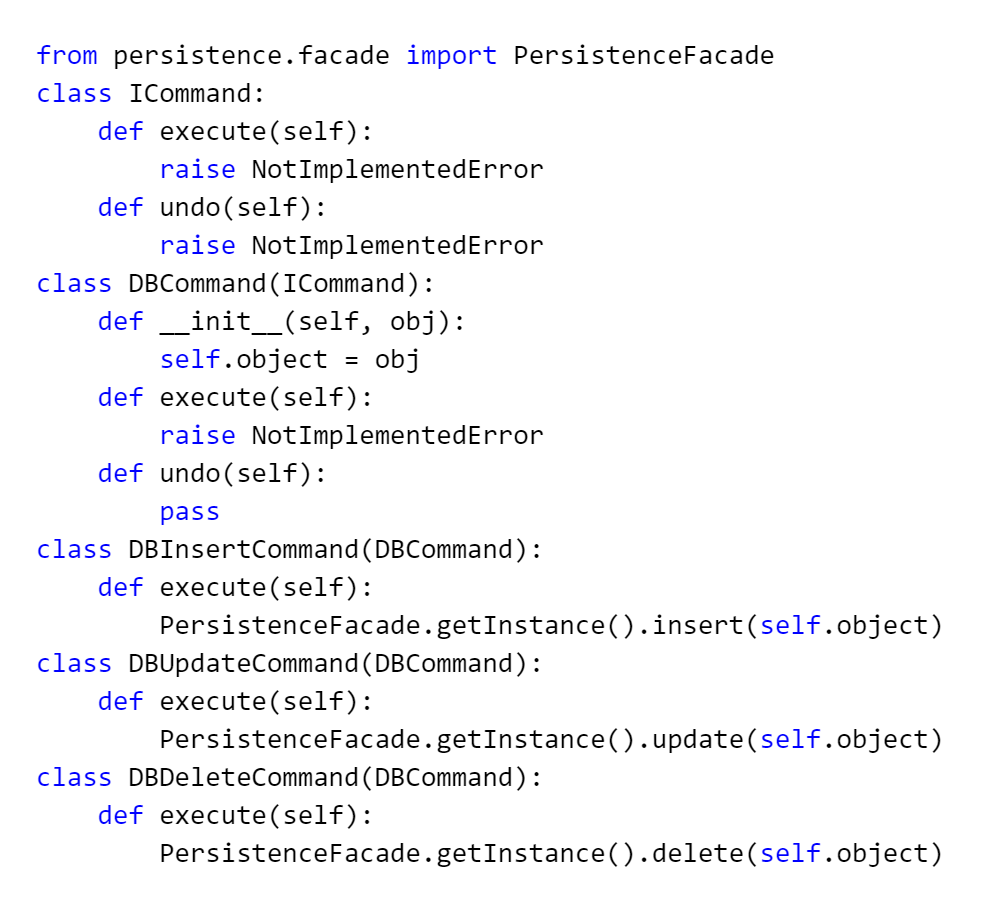
در ادامه، الگوی حالت را برای مدیریت حالت‌های مختلف یک شی در پایگاه داده بررسی می‌کنیم. این الگو باعث می‌شود که رفتار یک شی بسته به حالت جاری آن تغییر کند. کلاس PObjectState یک کلاس پایه است که حالت‌های مختلف یک شی را تعریف می‌کند. این کلاس چهار متد commit، delete، rollback و save دارد. کلاس NewState 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 ذخیره‌شده و بدون تغییر است. متد delete در این کلاس، شی را به حالت حذف‌شده (OldDeleteState) تغییر می‌دهد. متد save 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. متد commit در این کلاس، تغییرات شی در پایگاه داده را ذخیره کرده و حالت آن را به OldCleanState تغییر می‌دهد. متد rollback در این کلاس، لغو تغییرات و بارگذاری مقدار قبل از پایگاه داده را انجام می‌دهد. متد delete، شی را به حالت OldDeleteState تغییر می‌دهد. کلاس OldDeleteState نمایانگر شی‌ای است که قرار است حذف شود، اما هنوز commit نشده است. متد در commit در این کلاس، حذف را تأیید می‌کند و حالت شی را به DeletedState تغییر می‌دهد. متد rollback در این کلاس، تغییرات را لغو کرده و حالت را به OldCleanState برمی‌گرداند. کلاس DeletedState نشان‌دهندۀ 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 9-3 توجه کنید.



شکل 9-3: پیاده‌سازی قسمت حالت‌های بخش پایداری چارچوب

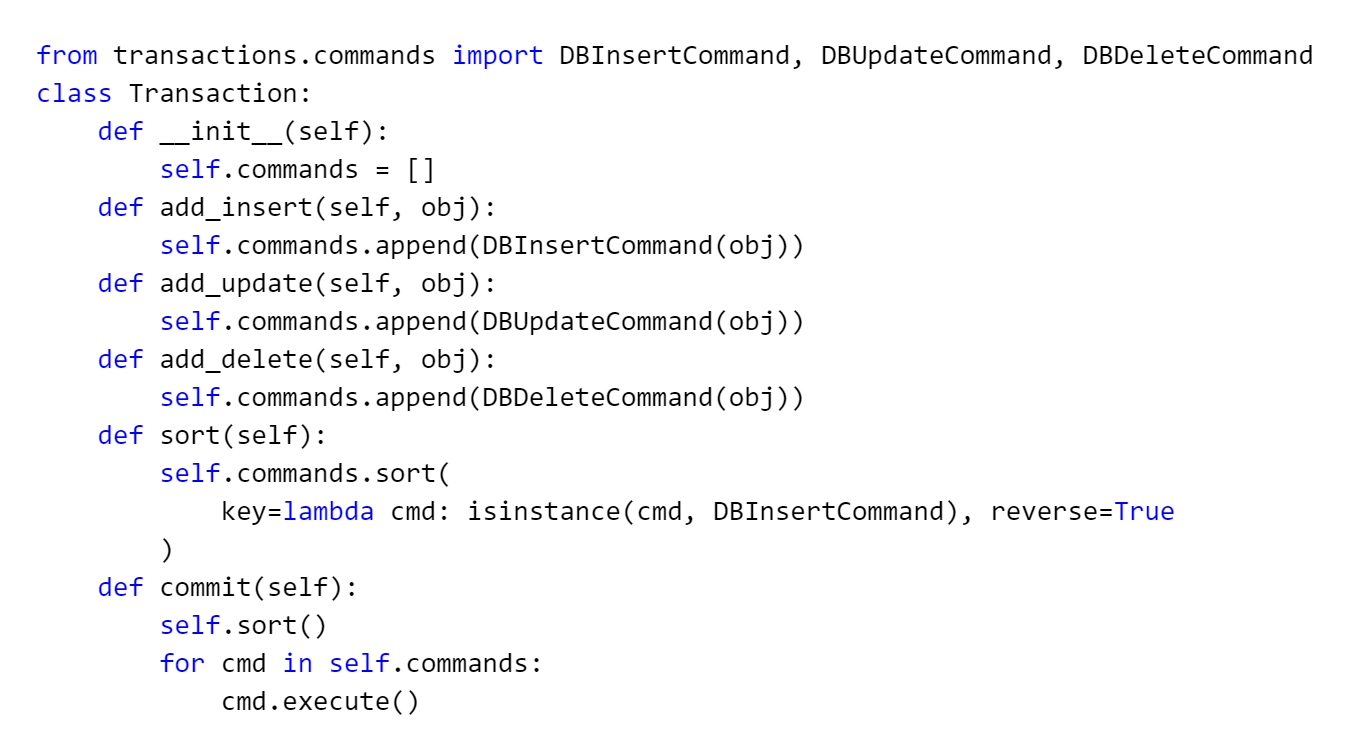
## 3-6 پیاده‌سازی چارچوب: بخش تراکنش‌ها

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 DBCommand از ICommand ارث‌بری می‌کند و کلاس پایه برای همۀ دستورات پایگاه داده است. این کلاس دارای یک ویژگی شی است که عملیات روی آن انجام می‌شود. متدها در این کلاس نیز پیاده‌سازی نشده اند. کلاس DBInsertCommand عملیات درج در پایگاه داده را انجام می‌دهد. متد execute از طریق PersistenceFacade شی را در پایگاه داده ذخیره می‌کند. کلاس‌های DBUpdateCommand و DBDeleteCommand به‌صورت مشابه پیاده‌سازی می‌شوند. به شکل 10-3 توجه کنید.



شکل 10-3: پیاده‌سازی قسمت دستورات بخش تراکنش‌های چارچوب

در ادامه، مدیریت تراکنش‌ها براساس الگوی دستور پیاده‌سازی می‌شود. این طراحی باعث می‌شود که عملیات پایگاه داده (درج، به‌روزرسانی و حذف) به‌صورت 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 پایگاه داده شامل DBInsertCommand، DBUpdateCommand و DBDeleteCommand هستند که به‌کمک متدهای add\_insert، add\_update و add\_delete اشیا برای درج، به‌روزرسانی و حذف اضافه می‌شوند. متد sort دستورات را طوری مرتب می‌کند که ابتدا درج انجام شود، چراکه می‌خواهیم قبل از به‌روزرسانی و حذف از وجود رکورد در پایگاه داده اطمینان حاصل کنیم. متد commit تمام دستورات را اجرا می‌کند (با اولویت درج). به شکل 11-3 توجه کنید.



شکل 11-3: پیاده‌سازی قسمت تراکنش بخش تراکنش‌های چارچوب

## 3-7 پیاده‌سازی چارچوب: بخش اصلی

در این بخش، از معماری ساخته‌شده در بخش‌های قبل استفاده می‌کنیم و به‌نوعی، آزمایش چارچوب را انجام می‌دهیم. در ابتدا، ماژول‌های لازم از فایل‌های مختلف برنامه اضافه می‌شوند. سپس، تنظیمات اتصال به پایگاه داده از یک فایل config.json که محتویات آن می‌تواند مثلاً به‌صورت 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 به‌کمک RDBOperations که یک Singleton است تنظیم می‌شود؛ سپس، اتصال پایگاه داده تنظیم می‌شود تا rdb\_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 commit 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 حذف‌شده 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 country\_id=85 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 12-3 توجه کنید.



شکل 12-3: پیاده‌سازی بخش اصلی چارچوب

# فصل 4

# جمع‌بندی

## 4-1 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 پروژه بررسی شده و نقاط قوت، چالش‌ها و راهکارهای پیشنهادی برای بهبود آینده سیستم ارائه می‌شود.

## 4-2 ارزیابی عملکرد چارچوب پیشنهادی: چالش‌ها و راهکارها

همان‌طور که در فصل دوم اشاره شد، چارچوب‌های پایداری مشابهی وجود دارند که ابعاد خیلی بزرگ‌تری گسترش یافته اند. این چارچوب به‌عنوان یک نمونۀ 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 داراست.

یکی از مسائلی که کمتر به آن پرداخته شد و در صورت علاقه‌مندی می‌توانید به آن بپردازید، عملیات غیرمادی‌سازی است. همچنین، اجرای پرس‌وجوها که به‌صورت 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 به‌کمک قفل‌گذاری، این چارچوب را بهبود بخشید.

ضمناً بی‌شک مدیریت زمان و حافظه همواره در سیستم‌ها مورد اهمیت بوده است. در این پروژه با کمک گرفتن از حافظۀ نهان و ایدۀ 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 پروژه در تجارت الکترونیک، رایانش ابری، مدیریت اطلاعات سازمانی، مدیریت امور مالی و... داشته باشد، گسترش و بهبود چارچوب ارائه‌شده از جهات ذکرشده، خالی از لطف نیست.

## 4-3 نتیجه‌گیری

نتایج این پروژه نشان می‌دهد که چارچوب پیشنهادی توانسته است عملکرد مطلوبی را در مدیریت داده‌ها ارائه دهد. با وجود چالش‌های مطرح‌شده، با به‌کارگیری 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

|  |  |
| --- | --- |
| انتزاعی | Abstract |

C

|  |  |
| --- | --- |
| منطق شرطی | Case Logic |
| انسجام | Cohesion |
| الگوی دستور | Command Pattern |
| مشخص | Concrete |
| کنترل هم‌زمانی | Concurrency Control |
| زنجیرۀ سازنده‌ها | Constructor Chaining |
| شی زمینه | Context Object |
| اتصال | Coupling |

D

|  |  |
| --- | --- |
| داده‌محور | Data-Driven |
| به‌تعویق انداختن | Deferring |
| رشتۀ طراحی | Design Discipline |
| طراحی نگاشت مستقیم | Direct Mapping Design |
| اشیای تغییریافته | Dirty Objects |
| پویا | Dynamic |

E

|  |  |
| --- | --- |
| مادی‌سازی پیش‌دستانه | Eager Materialization |
| کپسوله‌سازی | Encapsulation |

G

|  |  |
| --- | --- |
| چارچوب رابط کاربری گرافیکی | GUI Framework |

H

|  |  |
| --- | --- |
| پشتۀ تاریخچه | History Stack |

L

|  |  |
| --- | --- |
| مادی‌سازی تنبل | Lazy Materialization |
| برگ | Leaf |
| شی سبک‌وزن | Lightweight Object |
| حافظۀ نهان محلی | Local Cache |
| قفل | Lock |
| گزارش | Logging |

O

|  |  |
| --- | --- |
| شناسۀ شی | Object Identifer |
| شی‌گرا | Object-Oriented |
| الگوی ناظر | Observer Pattern |
| نگاشت شی‌-رابطه‌ای | O-R Mapping |
| بازنویسی | Overriding |

P

|  |  |
| --- | --- |
| بسته | Package |
| غیرفعال‌سازی | Passivation |
| چسباندن | Pasting |
| زیرسیستم پایداری | Persistence Subsystem |
| مخزن پایدار | Persistent Store |
| نقطۀ فروش | Point of Sale |
| چندریختی | Polymorphic |
| تغییرات محافظت‌شده | Protected Variations |
| عمومی | Public |
| ساختگی محض | Pure Fabrication |

Q

|  |  |
| --- | --- |
| ارتباط واجد شرایط | Qualified Association |
| پرس‌وجو | Query |

R

|  |  |
| --- | --- |
| محدودیت ارجاعی | Referential Constraint |
| برنامه‌نویسی بازتابی | Reflective Programming |
| پایگاه دادۀ رابطه‌ای | Relational Database |
| سرور از راه دور | Remote Server |
| قابلیت استفادۀ مجدد | Reusability |

S

|  |  |
| --- | --- |
| سند معماری نرم‌افزار | SAD |
| نگاشت طرح‌واره‌ای | Schema Mapping |
| جداسازی دغدغه‌ها | Separation of Concerns |
| اشاره‌گر هوشمند | Smart Pointer |
| مفسر ماشین حالت | State Machine Interpreter |
| الگوی حالت | State Pattern |
| مدل انتقال حالت | State Transmission Model |
| نمودار حالت | Statechart |
| زیرکلاس‌گیری | Subclassing |
| همگام‌سازی‌شده | Synchronized |
| ویژگی‌های سیستم | System Properties |

T

|  |  |
| --- | --- |
| وظیفه | Task |
| لایۀ خدمات فنی | Technical Services Layer |

U

|  |  |
| --- | --- |
| لغو | Undoing |
| فرایند یکپارچه | Unified Process |
| واحد کاری | Unit of Work |

V

|  |  |
| --- | --- |
| پروکسی مجازی | Virtual Proxy |

W

|  |  |
| --- | --- |
| ابزارک | Widget |
| میزکار | Workbench |

**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

1. یک برنامۀ نمونه که اغلب در متون آموزشی و پژوهشی برای نمایش مفاهیم طراحی نرم‌افزار و الگوهای معماری استفاده می‌شود. [↑](#footnote-ref-1)
2. Relational Database [↑](#footnote-ref-2)
3. روشی در EJB که در آن، چرخۀ حیات و عملیات پایداری Entity Bean‌ها به‌صورت خودکار توسط محفظۀ اجرایی EJB مدیریت می‌شود، بدون نیاز به پیاده‌سازی مستقیم عملیات پایگاه داده توسط توسعه‌دهنده. [↑](#footnote-ref-3)
4. Enterprise JavaBeans که یک معماری سرورمحور در جاوا برای توسعۀ برنامه‌های سازمانی است و شامل **کلاس‌های تحت سرور** جهت مدیریت تراکنش‌ها، امنیت، هم‌روندی و پایداری داده‌ها است. [↑](#footnote-ref-4)
5. 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. [↑](#footnote-ref-5)
6. Record-Oriented [↑](#footnote-ref-6)
7. Object-Oriented [↑](#footnote-ref-7)
8. O-R Mapping [↑](#footnote-ref-8)
9. فایل‌های متنی ساده که داده‌ها را بدون ساختار پیچیده ذخیره می‌کنند. داده‌ها معمولاً در قالب متن ساده (TXT) یا جداول جداشده 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. [↑](#footnote-ref-9)
10. یک نوع فایل متنی ساختاریافته که داده‌ها را در قالب برچسب‌های (Tags) سلسله‌مراتبی ذخیره می‌کند. [↑](#footnote-ref-10)
11. فایل‌هایی که اطلاعات را به‌صورت رکوردهای باینری ذخیره می‌‌کنند و برای برنامه‌های مختلف مانند کتاب‌های الکترونیکی، دفترچه‌های یادداشت و پایگاه‌ داده‌ها کاربرد دارند. [↑](#footnote-ref-11)
12. نوعی ساختار ذخیره‌سازی درختی؛ در این مدل، هر رکورد والد می‌تواند چندین رکورد فرزند داشته باشد، اما هر فرزند فقط یک والد دارد. [↑](#footnote-ref-12)
13. Persistence Subsystem [↑](#footnote-ref-13)
14. Technical Services Layer [↑](#footnote-ref-14)
15. GUI Frameworks [↑](#footnote-ref-15)
16. Widgets [↑](#footnote-ref-16)
17. Subclassing [↑](#footnote-ref-17)
18. Overriding [↑](#footnote-ref-18)
19. Observer Pattern [↑](#footnote-ref-19)
20. Interfaces [↑](#footnote-ref-20)
21. Abstract [↑](#footnote-ref-21)
22. Concrete [↑](#footnote-ref-22)
23. Interface User [↑](#footnote-ref-23)
24. Hollywood Principle: "Don't call us, we'll call you." [↑](#footnote-ref-24)
25. Reusability [↑](#footnote-ref-25)
26. یک سامانۀ نقطۀ فروش (Point of Sale) نسل جدید که برای مدیریت عملیات فروش، پردازش تراکنش‌ها و ذخیرۀ اطلاعات مربوط به خرید در محیط‌های تجاری استفاده می‌شود. [↑](#footnote-ref-26)
27. Persistence Framework [↑](#footnote-ref-27)
28. یک چارچوب پایداری صنعتی و کامل خارج از حوزۀ این مقدمه است. [↑](#footnote-ref-28)
29. Persistent Store [↑](#footnote-ref-29)
30. Schema Mapping [↑](#footnote-ref-30)
31. Object Identity and Identifier [↑](#footnote-ref-31)
32. Pure Fabrication به‌معنای عدم وجود در دنیای خارجی [↑](#footnote-ref-32)
33. Materialization [↑](#footnote-ref-33)
34. Dematerialization [↑](#footnote-ref-34)
35. Passivation [↑](#footnote-ref-35)
36. Cache [↑](#footnote-ref-36)
37. Dirty Objects [↑](#footnote-ref-37)
38. Lazy Materialization [↑](#footnote-ref-38)
39. Virtual Proxy [↑](#footnote-ref-39)
40. 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 به‌صراحت حذف شود. [↑](#footnote-ref-40)
41. UP یا یک چارچوب مهندسی نرم‌افزار برای توسعۀ سیستم‌های شی‌گرا [↑](#footnote-ref-41)
42. Design Discipline [↑](#footnote-ref-42)
43. Referential Constraint [↑](#footnote-ref-43)
44. Stereotypes [↑](#footnote-ref-44)
45. سازمان استانداردسازی فعال در حوزۀ توسعه و حفظ استانداردهای مربوط به فناوری‌های شی‌گرا و معماری‌های نرم‌افزاری [↑](#footnote-ref-45)
46. Direct Mapping Design [↑](#footnote-ref-46)
47. Coupling [↑](#footnote-ref-47)
48. Cohesion [↑](#footnote-ref-48)
49. Separation of Concerns [↑](#footnote-ref-49)
50. Indirect Mapping [↑](#footnote-ref-50)
51. در سیستم‌های توزیع‌شده، Broker یک فرایند سرور Front-End است که وظایف را به فرایندهای سرور Back-End واگذار می‌کند. [↑](#footnote-ref-51)
52. Qualified Association [↑](#footnote-ref-52)
53. در HashMap، جست‌وجو با استفاده از کلید انجام شده و به‌کمک تابع هش، مقدار مربوطه در زمان تقریباً ثابت بازیابی می‌شود. [↑](#footnote-ref-53)
54. Hardwired [↑](#footnote-ref-54)
55. Metadata [↑](#footnote-ref-55)
56. Dynamic [↑](#footnote-ref-56)
57. Reflective Programming [↑](#footnote-ref-57)
58. Protected Variations [↑](#footnote-ref-58)
59. Encapsulation of Implementation [↑](#footnote-ref-59)
60. GoF (Gang of Four) به چهار نویسندۀ کتاب Design Patterns: Elements of Reusable Object-Oriented Software اشاره دارد که الگوهای طراحی را معرفی کردند. [↑](#footnote-ref-60)
61. این الگو هیچ ارتباطی با Templateهای C++ ندارد، بلکه قالب کلی یک الگوریتم را توصیف می‌کند. [↑](#footnote-ref-61)
62. به‌طور خاص‌تر، این الگو در چارچوب‌های Whitebox به‌کار 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. [↑](#footnote-ref-62)
63. Superclass [↑](#footnote-ref-63)
64. Subclass [↑](#footnote-ref-64)
65. Final [↑](#footnote-ref-65)
66. Leaf [↑](#footnote-ref-66)
67. Public [↑](#footnote-ref-67)
68. Constructors [↑](#footnote-ref-68)
69. Constructor Chaining [↑](#footnote-ref-69)
70. Persistence Storage Mechanisms [↑](#footnote-ref-70)
71. Packages [↑](#footnote-ref-71)
72. Hard-Coded Objects [↑](#footnote-ref-72)
73. SAD که یک مستند فنی توضیح‌دهندۀ دیدگاه‌های کلیدی معماری یک سیستم نرم‌افزاری است. [↑](#footnote-ref-73)
74. Thread Safety [↑](#footnote-ref-74)
75. Remote Server Object [↑](#footnote-ref-75)
76. Thread Concurrency Control [↑](#footnote-ref-76)
77. Synchronized Method [↑](#footnote-ref-77)
78. Data-Driven Design [↑](#footnote-ref-78)
79. System Properties [↑](#footnote-ref-79)
80. Local Cache [↑](#footnote-ref-80)
81. Hard-coding [↑](#footnote-ref-81)
82. State Pattern [↑](#footnote-ref-82)
83. Statechart [↑](#footnote-ref-83)
84. 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. [↑](#footnote-ref-84)
85. Case Logic [↑](#footnote-ref-85)
86. Context Object [↑](#footnote-ref-86)
87. کلاس Deleted به‌دلیل محدودیت فضا در نمودار حذف شده است. [↑](#footnote-ref-87)
88. State Transmission Model [↑](#footnote-ref-88)
89. راهکارهای دیگری نیز برای مدیریت وضعیت تراکنش‌ها وجود دارند، ازجمله منطق شرطی قرار داده شده، مفسرهای ماشین حالت  
    (State Machine Interpreters) و تولیدکننده‌های کد که توسط جداول حالت هدایت می‌شوند. [↑](#footnote-ref-89)
90. Stateless [↑](#footnote-ref-90)
91. Command [↑](#footnote-ref-91)
92. Unit of Work [↑](#footnote-ref-92)
93. Atomic [↑](#footnote-ref-93)
94. Task [↑](#footnote-ref-94)
95. Logging [↑](#footnote-ref-95)
96. Undoing [↑](#footnote-ref-96)
97. Polymorphic [↑](#footnote-ref-97)
98. Cut [↑](#footnote-ref-98)
99. Paste [↑](#footnote-ref-99)
100. History Stack [↑](#footnote-ref-100)
101. Deferring [↑](#footnote-ref-101)
102. Manufacturer Rebate [↑](#footnote-ref-102)
103. Lightweight [↑](#footnote-ref-103)
104. Smart Pointer [↑](#footnote-ref-104)
105. Eager Materialization [↑](#footnote-ref-105)
106. 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 دادۀ رابطه‌ای استفاده شود). [↑](#footnote-ref-106)
107. Query [↑](#footnote-ref-107)
108. Query Builder [↑](#footnote-ref-108)
109. Workbench [↑](#footnote-ref-109)
110. در SQLite، نیازی به host، username و password نیست. [↑](#footnote-ref-110)
111. Static [↑](#footnote-ref-111)
112. Decorator [↑](#footnote-ref-112)
113. Lazy Loading [↑](#footnote-ref-113)
114. Lock [↑](#footnote-ref-114)
115. Race Condition [↑](#footnote-ref-115)