Extension of the mediator-wrapper architecture for heterogeneous data source integration by adding a mask

Dončević, Juraj

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Juraj Dončević

EXTENSION OF THE MEDIATOR–WRAPPER ARCHITECTURE FOR HETEROGENEOUS DATA SOURCE INTEGRATION BY ADDING A MASK

DOCTORAL THESIS

Zagreb, 2024

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Juraj Dončević

EXTENSION OF THE MEDIATOR–WRAPPER ARCHITECTURE FOR HETEROGENEOUS DATA SOURCE INTEGRATION BY ADDING A MASK

DOCTORAL THESIS

Supervisor: Professor Krešimir Fertalj, PhD

Zagreb, 2024

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Juraj Dončević

PROŠIRENJE ARHITEKTURE MIRITELJ–OMOTAČ ZA INTEGRACIJU HETEROGENIH IZVORA PODATAKA DODAVANJEM MASKE

DOKTORSKA DISERTACIJA

Mentor: prof. dr. sc. Krešimir Fertalj

Zagreb, 2024.

The DOCTORAL THESIS was created at the University of Zagreb, Faculty of Electrical Enigneering and Computing, at the Department for Applied Computing

Supervisor: Professor Krešimir Fertalj, PhD

DOCTORAL THESIS has: 231 pages

DOCTORAL THESIS br.: _____

About the Supervisor

Krešimir Fertalj is a full professor at the Department of Applied Computing at the Faculty of Electrical Engineering and Computing (FER), University of Zagreb, where he lectures a couple of computing courses on graduate, specialist and doctoral studies. His professional and scientific interest is in automated software engineering, complex information systems, project management and in software security. He led several scientific and research projects and a few dozen of development projects. He was a mentor to students for over 300 bachelor and graduate theses, 9 MSc and 12 PhD theses. He has published nearly 200 scientific and technical papers. Prof. Fertalj is the founder of the Laboratory for Special Purpose Information Systems and of Postgraduate Specialist Study "Project Management" at FER. He is a senior member of IEEE and a full member of the Croatian Academy of Engineering (HATZ). He served as a Head of Department at FER, a Secretary of the Department of Information Systems of HATZ and was one of the founders and a member of the management board of the PMI chapter in Croatia.

O mentoru

Krešimir Fertalj redoviti je profesor na Zavodu za primijenjeno računarstvo Fakulteta elektrotehnike i računarstva (FER) Sveučilišta u Zagrebu, gdje predaje nekoliko računarskih kolegija na diplomskim, specijalističkim i doktorskim studijima. Njegov stručni i znanstveni interes je automatizirano programsko inženjerstvo, složeni informacijski sustavi, upravljanje projektima i sigurnost softvera. Vodio je nekoliko znanstveno-istraživačkih projekata te nekoliko desetaka razvojnih projekata. Bio je mentor studentima na više od 300 preddiplomskih i diplomskih radova, 9 magistarskih i 12 doktorskih radova. Objavio je gotovo 200 znanstvenih i stručnih radova. Prof. Fertalj osnivač je Laboratorija za informacijske sustave posebne namjene i poslijediplomskog specijalističkog studija "Upravljanje projektima" na FER-u. Viši je član IEEE i redoviti član Hrvatske akademije tehničkih znanosti (HATZ). Obnašao je dužnost voditelja zavoda na FER-u, tajnika Odjela informacijskih sustava HATZ-a te je bio jedan od osnivača i član upravnog odbora PMI ogranka u Hrvatskoj.

Abstract

This doctoral research deals with the extension of the mediator–wrapper architecture for heterogeneous data source integration. Scenarios that require the addition of diverse system access interfaces where the mediator–wrapper architecture underperforms are identified. Therefore, an extension of the mediator–wrapper architecture is proposed by the addition of a new component type called a mask. The mask presents a new layer in the mediator–wrapper architecture, effecting the creation of a mask–mediator–wrapper architecture. The proposed architecture is observed in modern technical aspects by including the consideration of architectural quanta. The proposed architecture is examined through qualitative and quantitative analyses, and case studies emulating a legacy data store preserving system and a data mesh. The mask is observed as a generic component that can be implemented in various mask kinds. This observation is facilitated by an analysis of the mask's inner components and data flows. Masks are proposed to be uniformly developed components by using a mask framework. The masks are proposed to be used in architecture topologies as prefabricated and configurable components. Although the translation of schemas and queries in the mask are determined to be one-way transformations, the translation of data is bidirectional. The use of bidirectionalisation for data translation is examined to reduce the effort of implementing a mask and enable reasoning about the correctness of the implemented two-way transformation. A simple symmetric lens is determined as the method of choice for data transformations and treated as a design pattern. A lens is implemented for use in a mask kind. A testing framework is created for determining the behavedness of the implemented lenses. A mask–mediator–wrapper system, called Janus, is presented as a prototype to satisfy the need for empirical proof and experimentation. Janus supports implementing masks through a mask framework as a proof-of-concept. The Janus mask framework is shown to enable the implementation of a mask kind in nine general steps. Three implementations of mask kinds in Janus are presented. The qualitative analysis and case studies are confirmed by the deployment of Janus components in different architectural topologies.

Keywords: software architecture, data integration, data management, mediator–wrapper, bidirectionalisation, functional programming, category theory

Proširenje arhitekture miritelj–omotač za integraciju heterogenih izvora podataka dodavanjem maske

Uvod

Integracija podataka uvijek je imala ključnu ulogu u upravljanju podacima; očekivano je da sustav koji rukuje s više izvora podataka u nekom trenutku izvrši njihovu integraciju. Iako se podaci mogu integrirati na razini pojedinačnog slučaja korištenja, sustavi obično zahtijevaju sveobuhvatnu integraciju podataka iz više izvora. Ostvarivanje generičke integracije podataka iz više skupova podataka ključno je za postizanje jedinstvenog prikaza izvorišnih podataka i smislenog uvida u podatke. Sustav za integraciju izvora podataka je sustav za upravljanje podacima koji ima zadatak integriranja dva ili više izvora podataka [1].

Istraživanje o integraciji podataka polje je bilo predmet sveobuhvatnih studija tijekom 1990-ih [2, 3, 4, 5, 6, 7]. Napredak u istraživanju je označilo ostvarivanje dva značajna sustava za integraciju heterogenih izvora podataka - GARLIC [8] i TSIMMIS [9]. Oba sustava su implementirana arhitekturom miritelj–omotač. Nažalost, ti su sustavi bili kratkog vijeka i nisu bili javno dostupni.

U području integracije podataka došlo je do značajnih noviteta pojavom NoSQL sustava [10]. NoSQL sustavi su ojačali značaj paradigmatske heterogenosti izvora podataka [1]. Podaci se više nisu samo pohranjivali u relacijskim bazama podataka, već i u formatima bez sheme u specijaliziranim bazama podataka, pa čak i u datotekama. Proteklo desetljeće označeno je povećanjem količine podataka i popularizacijom jezera podataka koja omogućavaju pohranu velikih količina distribuiranih, nestrukturiranih i heterogenih podataka [11, 12]. Sustavi poput jezera podataka pridonijela su povećanju raznolikosti podataka, naglašavajući važnost ostvarivanja integracije heterogenih podataka. Napredak integracije podataka kaska za uvođenjem novih sustava za upravljanje podacima i novim izvorima podataka, a tome pridodaje nedostatak sveobuhvatnih i otvorenih sustava za eksperimentiranje [13].

Dugogodišnji trendovi u upravljanju podacima ukazuju na sve veću važnost reprezentacije

podataka. Može se očekivati kako će se sljedeći pomak na području integracije podataka ticati reprezentacije podataka, te kako će integracijski sustavi morati omogućiti ne samo heterogenost izvora podataka, nego i reprezentacije podataka [14]. Cilj istraživanja predstavljenog ovom disertacijom je uvođenje nove arhitekture čija bi se fleksibilnost mogla nositi s budućim izazovima u integraciji podataka po pitanju reprezentacije podataka.

Istraživanjem u sklopu ove disertacije uvodi se arhitektura maska–miritelj–omotač kao proširenje arhitekture miritelj–omotač. Proširenje se ostvaruje dodavanjem nove komponente zvane maska. Komponenta maske se konceptualno razrađuje do razine principa rada, te se za nju predlaže mogućnost stvaranja radnog okvira koji omogućava standardizirani razvoj maski. Predloženi koncepti dokazani su razvojem prototipnog sustava Janus, zasnovanog na arhitekturi maska–miritelj–omotač. Janus sadrži prototipe tri vrste maski i radni okvir za njihov razvoj. Dvosmjerne transformacije za podatke u različitim formatima ostvarene su korištenjem jednostavne simetrične leće kao oblikovnog obrasca, u svrhu smanjenja napora implementacije pojedine vrste maske. Jednostavne simetrične leće za dvosmjerne transformacije implementirane su u sklopu sustava Janus, te je uz njih implementiran i jednostavan radni okvir za testiranje implementiranih leća kojim je moguće dokazati razinu ispravnosti transformacija, odnosno ponašanja leće.

Kako bi se pokazalo poboljšanje arhitekture maska–miritelj–omotač, predstavlja se sposobnost sustava maska–miritelj–omotač za emuliranje drugih sustava za upravljanje podacima - sustav SOS i mreže podataka (eng. *data mesh*). Emulacija se empirijski dokazuje postavljanjem i konfiguracijom komponenti sustava Janus.

Ova disertacija predstavlja sljedeće znanstvene doprinose ostvarene tijekom doktorskog istraživanja:

1. Proširenje arhitekture miritelj-omota č za reprezentaciju podataka, sheme podataka i upita nad podacima komponentom nazvanom maska, kojom će se omogućiti implementacija raznovrsnih pristupnih sučelja sustava za integraciju heterogenih izvora podataka;
2. Metoda za stvaranje dvosmjernih transformacija podataka u maskama korištenjem bidirekcionalizacije, kako bi se smanjio implementacijski napor po vrsti maske;
3. Radni okvir kojim se omogu ćuje ugradnja prethodno definirane maske i predložene metode, verificiran prototipom ugradnje nad reprezentativnim primjerom hipotetičkog sustava za integraciju heterogenih izvora podataka.

Preliminarni pojmovi iz softverskih arhitektura

Poglavlje 2 daje pregled pojmova koji su preliminarno potrebni za razumijevanje sadržaja ove disertacije u vidu softverskih arhitektura. Potpoglavlje 2.1 predstavlja pojmove modula, komponenti i slojeva, te ih povezuje uz pojmove metrike sprezanja i kohezije kôda [15, 16]. Isto

potpoglavlje predstavlja pojam arhitekturalnog kvanta i evolucijske arhitekture [17], te prepoznate arhitekturalne karakteristike [15]. Predstavlja se i tijek procesa identifikacije komponenti sustava [15] koji se koristio kao vodilja pri razradi arhitekture iz doprinosa. Ranije spomenuti pojmovi karakteristika arhitektura povezuju se s uvjetima za dobro oblikovane komponente koje je prethodno uveo Meyer [18]. Uvjeti koje je predstavio Meyer korišteni su pri razradi komponenti za arhitekturu iz doprinosa.

Potpoglavlje 2.2 daje kratki uvod u sustave za upravljanje podacima, poglavito sustave za integraciju izvora podataka. Potpoglavlje 2.3 predstavlja arhitekturu miritelj–omotač u kontekstu sustava za integraciju izvora podataka, te predstavlja ustaljena pravila za komponente miritelja [3] i omotača [5, 8]. Oba tipa komponenti su sažeto predstavljena definicijama [14]. Daljnja diskusija vodi u strategije raspodjele shema po komponentama miritelja i omotača [19], te se predstavljaju dvije konfiguracije arhitekture miritelj–omotač - s jednim slojem miritelja i s dva sloja miritelja. Ove strategije su važna točka diskusije u kvalitativnoj analizi arhitekture iz doprinosa.

Potpoglavlje 2.4 predstavlja najsuvremeniju arhitekturu za raspodijeljeno upravljanje podacima - mrežu podataka (eng. *data mesh*) [20]. Arhitektura mreže podataka koristi se u studijama slučaja, gdje se demonstrira kako arhitektura iz doprinosa može emulirati mrežu podataka.

Potpoglavlje 2.5 opisuje postupak kvantitativne analize fleksibilnosti softvera [21, 22] kojom je moguće uspoređivati softver. Analiza koristi kvantizirane evaluacije broja promjena generičkih modula softvera u scenarijima promjene zahtjeva. Prilagodba predstavljene kvantitativne analize fleksibilnosti softvera koristi se u ovoj disertaciji za usporedbu relevantnih arhitektura.

Preliminarni pojmovi iz teorije kategorija, funkcijskog programiranja i bidirekcionalizacije

Poglavlje 3 daje pregled pojmova koji su preliminarno potrebni za razumijevanje sadržaja ove disertacije u vidu teorije kategorija, funkcijskog programiranja i bidirekcionalizacije. Ovo poglavlje služi kako bi se predstavio osnovni skup znanja koji je potreban za razumijevanje pojmova iz područja bidirekcionalizacije. Potpoglavlje 3.1 uvodi osnovne pojmove teorije kategorija i veže ih uz programiranje korištenjem kategorije tipova kao referentne kategorije. Potpoglavlje 3.2 nadalje razrađuje pojmove srodne teoriji kategorija kroz okvir funkcijskog programiranja. Uvode se pojmovi parcijalne aplikacije, funkcija višeg reda, funktora i monada.

Potpoglavlje 3.3 predstavlja područje bidirekcionalizacije korištenjem prethodno uvedenih mehanizama funkcijskog programiranja. Bidirekcionalizacija se predstavlja kroz tri istaknutije metode bidirekcionalizacije: semantička bidirekcionalizacija, sintaksna bidirekcionalizacija i bidirekcionalni kombinatori. Semantička bidirekcionalizacija se poglavito predstavlja kroz

rad Voigtländera [23]. Sintaksna bidirekcionalizacija se poglavito predstavlja kroz rad Matsude [24]. Uvode se osnovna pravila za kružne transformacije, odnosno provjeru ponašanja transformacija. Ta pravila se nadalje razrađuju prema potrebi pojedinih tehnika bidirekcionalizacije, kao što su korištenje komplemenata i monada. Nadalje se predstavljaju leće kao nositelji bidirekcionalnih transformacija, te se opisuju asimetrične, simetrične [25] i jednostavne simetrične leće [26]. Jednostavna simetrična leća koristi se kao oblikovni obrazac u implementaciji dvosmjernih transformacija komponente iz doprinosa.

Arhitektura maska–miritelj–omotač

Poglavlje 4 uvodi arhitekturu iz doprinosa - maska–miritelj–omotač. Ovo poglavlje je preuzeto iz znanstvenog rada objavljenog u sklopu doktorskog istraživanja [14], uz manje prilagodbe.

Potpoglavlje 4.1 predstavlja uočene probleme postojeće arhitekture miritelj–omotač. To se ostvaruje kroz kvalitativnu analizu raspodjela shema po komponentama, kojom se uočava kako pojedine komponente arhitekture miritelj–omotač moraju upravljati s dvije ili više shema različitih slučaja uporabe. Zaključuje se kako to krši ispravnu raspodjelu odgovornosti između komponenata, pogotovo u miriteljskim komponentama u najvišem sloju arhitekture. Zaključuje se kako je miriteljima dodijeljeno upravljanje i reprezentacija podataka, te da to nije povoljno u slučaju potrebe za više različitih reprezentacija podataka sustava. Potreba za istim se naglašava pregledom radova iz područja koji ukazuju na raznolikost potrebnih sučelja sustava za upravljanje podacima. Kao rješenje u okviru arhitekture miritelj–omotač predlaže se dodavanje trećeg pravila za miritelje koje propisuje da miritelji služe mirenju (modela), a ne za reprezentaciju.

Potpoglavlje 4.2 predstavlja rješenje uočenih problema u vidu proširenja postojeće arhitekture dodatnom komponentom za potrebe reprezentacije. Nova komponenta se uvodi pod nazivom *maska*. Za masku se propisuju tri pravila koja su u skladu s prethodno predstavljenim pravilima za miritelje i omotače, kao i s pravilima za dobro oblikovane komponente. Time se predlaže stvaranje nove arhitekture, zvane maska–miritelj–omotač, koja čini znanstveni doprinos ovog doktorskog istraživanja. Arhitektura maska–miritelj–omotač se kvalitativno analizira kroz raspodjelu shema po komponentama, te se dobivena raspodjela uspoređuje s raspodjelom u arhitekturi miritelj–omotač. Utvrđuje se kako je arhitektura maska–miritelj–omotač poboljšanje naspram arhitekture miritelj–omotač, jer sve komponente nove arhitekture posjeduju samo jednu shemu; čime su odgovornosti komponenti ispravno granulirane i raspodijeljene.

Potpoglavlje 4.3 koristi prilagođenu kvantitativnu analizu fleksibilnosti softvera kako bi se dokazalo poboljšanje postigunuto proširenjem postojeće arhitekture. Analiza se provodi na razini komponenata sustava kao generičkih modula. Analiza pokazuje kako predložena arhitektura ima poboljšanu fleksibilnost naspram postojećih u scenarijima dodavanja novih

reprezentacija podataka i dodavanja miritelja, te da u slučaju dodavanja novih izvora podataka ne dolazi do nazadovanja.

Komponenta maske

Poglavlje 5 predstavlja daljnju razradu komponente maske izvan okvira crne kutije. Ovo poglavlje je preuzeto iz znanstvenog rada objavljenog u sklopu doktorskog istraživanja [14], uz manje prilagodbe. U potpoglavlju 5.1 izvode se funkcijski zahtjevi na masku korištenjem pravila za maske.

U potpoglavlju 5.2 nadalje se razrađuju unutarnje komponente maske putem funkcijskih zahtjeva. Model tokova podataka, predstavljen dijagramom, služi dodatnoj razradi unutarnjih komponenti. Ova sistemska analiza rezultira konceptualnim modelom dizajna komponenti unutar maske. Teoretski se zaključuje kako maska može biti generički implementirana, izuzev komponenata s funkcionalnostima translacije shema, upita (uključujući naredbe) i podataka.

Potpoglavlje 5.3 predstavlja koncept radnog okvira za razvoj maska. Pretpostavka generičnosti komponente maske povlači zaključak kako bi se maske mogle implementirati putem radnog okvira. Time bi se njihov postupak implementacije pojednostavio i standardizirao. Ovo potpoglavlje uvodi pojmove vrste i tipova maski. Vrstama maske smatraju se biblioteke koje podupiru određeni oblik reprezentacije, dok se tipovima maske smatraju oblici implementacije konkretnih izvršivih komponenti maske.

Potpoglavlje 5.4 postavlja pretpostavke o tome kako je moguće maske ostvariti u modalitetima virtualizirajuće i materijalizirajuće maske, čime bi se omogućilo u ostvarivanje integracijskog sustava koji je ujedno i virtualizirajući i materijalizirajući.

Prototipni sustav

Iako propisani doprinosi ovog doktorskog istraživanja upućuju na mogućnost korištenja hipotetskog sustava, poglavlje 6 predstavlja ostvareni prototip sustava, nazvanog Janus, kao dokaz koncepta arhitekture maska–miritelj–omotač [27]. Ovime se pretpostavljeni doprinosi ojačavaju.

Potpoglavlje 6.1 predstavlja općeniti dizajn Janusa po pitanju implementacije pojedinih komponenti, konfiguracije mirenja shema, te modela za upite, naredbe, sheme i podatke.

Potpoglavlje 6.2 predstavlja ostvarenje prototipa maske. Potpoglavlje predstavlja implementirani radni okvir za implementaciju maski kao dokaz koncepta. Korištenjem radnog okvira pokazano je kako je implementacija vrste maske svedena na devet općenitih koraka (deseti korak uključuje implementaciju same izvršne komponente maske). U potpoglavlju se predstavljaju i implementacije prototipa maske: virtualizirajuća REST Web API maska, materijalizirajuća SQLite maska i materijalizirajuća LiteDB maska.

Metoda za bidirekcionalne transformacije podataka

Poglavlje 7 predstavlja korištenje jednostavne simetrične leće kao odabrane metode bidirekcionalizacije za implementaciju dvosmjernih transformacija podataka u maskama. Analiza i diskusija o maski je pokazala postojanje jednosmjernih i dvosmjerne translacije u maskama. Osiguravanje ispravnosti dvosmjernih transformacija i minimizacija implementacijskog troška osigurani su korištenjem jednostavne simetrične leće kao oblikovnog obrazaca. Samo korištenje oblikovnog obrasca implicira smanjenje implementacijskog troška, jer se implementacija tipizira. Korištenje jednostavne simetrične leće obrazloženo je u potpoglavlju 7.1.

Potpoglavlje 7.2 predstavlja implementaciju leća u programskom jeziku C#. Potpoglavlje 7.3 predstavlja implementirane leće koje su korištene u implementaciji REST Web API maske. Potpoglavlje 7.4 predstavlja jednostavni radni okvir za testiranje ponašanja implementiranih leća, te raspravlja do koje razine je moguće zaključiti da je ponašanje dokazano testom. Smanjenje implementacijskog troška očituje se i standardizacijom pisanja testova putem radnog okvira.

Studije slučaja maska–miritelj–omotača

Poglavlje 8 predstavlja studije slučaja emulacije drugih sustava od strane hipotetskog sustava maska–miritelj–omotača. Ovo poglavlje je preuzeto iz znanstvenog rada objavljenog u sklopu doktorskog istraživanja [14], te neobjavljenog znanstvenog rada [28] stvorenog u sklopu doktorskog istraživanja. Ovim poglavljem proširuje se kvalitativna analiza arhitekture maska–miritelj–omotač.

U potpoglavlju 8.1 predstavlja se mogućnost emulacije sustava za očuvanje naslijeđenih sustava pohrane podataka *Save Our Systems* (SOS), putem hipotetskog sustava maska–miritelj–omotača. Hipotetski se pokazuje kako je sustav SOS moguće i nadograđivati korištenjem arhitekture maska–miritelj–omotač, čime ga se može pretvoriti u integracijski sustav ili mu uvesti nove oblike reprezentacije.

U potpoglavlju 8.2 predstavlja se mogućnost emulacije mreže podataka putem hipotetskog sustava maska–miritelj–omotača. Nad generičkim slučajevima demonstrira se hipotetsko emuliranje platforme podatkovne infrastrukture i kontejnera podatkovnog proizvoda. Kontejner podatkovnog proizvoda emulira se u dva oblika, bez lokalnog spremišta domenskih podataka i s lokalnim spremištem domenskih podataka. Predlažu se pogodnosti hipotetske emulacije u vidu niskorizičnih proba uvođenja, brzog prototipiranja, garantiranja evolucijske karakteristike i standardizacije sustava mreže podataka.

Potpoglavlje 8.3 obuhvaća prototipiranje studija slučaja korištenjem sustava Janus kao reprezentativnog sustava maska–miritelj–omotača. Prototipiranja se provode postavljanjem komponenti u kontejnerizirano okruženje korištenjem alata Docker Compose.

Potpoglavlje 8.3.1 predstavlja postavke i postavljanje komponenti sustava Janus u topologiju heterogenog sustava za integraciju podataka. Potpoglavlje 8.3.2 predstavlja postavke i postavljanje komponenti sustava Janus u topologiju kojom se emulira sustav SOS. Potpoglavlje 8.3.3 predstavlja postavke i postavljanje komponenti sustava Janus u topologiju mreže podataka.

Dodaci

Poglavlje 10 je skup dodataka vezanih uz ovaj. Poglavlje dodataka sadrži popis kratica korištenih u radu, prefikse potrebne za izvršavanje programskih odsječaka, upute za pokretanje prototipova studija slučaja, konfiguracijske tablice prototipova studija slučaja. Potpoglavlje 10.5 sadrži detaljan pregled translacije shema, upita i podataka u prototipu sustava za integraciju izvora podataka.

Zaključak

Ova disertacija donosi unaprijeđenje postojeće arhitekture miritelj–omotač za integraciju heterogenih izvora podataka. Unaprijeđenje je postignuto proširenjem postojeće arhitekture novim tipom komponente zvanim maska, čime je stvorena nova arhitektura maska–miritelj–omotač.

Kroz kvalitativnu analizu pokazano je kako je nova arhitektura unaprijeđenje naspram postojeće u vidu raspodijele sheme po komponentama sustava. Nova arhitektura omogućava pojedinoj komponenti zaduženost za upravljanje isključivo jedne sheme. Nova arhitektura je prilagođena podršci raznolikih reprezentacija podataka, što se pregledom područja istraživanja pokazalo kao trend. Kvantitativna analiza fleksibilnosti softvera pokazala je bolju fleksibilnost arhitekture maska–miritelj–omotač naspram miritelj–omotača u slučajevima kada se dodaju novi tipovi reprezentacija i novo mirenje, te da ne postoji nazadovanje pri dodavanju novog izvora podataka u topologiju arhitekture sustava.

Komponenta maske je razrađena do principa rada. Za masku su propisana pravila u skladu s postojećim pravilima za komponente miritelja i omotača, kao i prilagodba pravila miritelja. Preko propisanih pravila za masku razrađeni su funkcijski zahtjevi na masku. Korištenjem funkcijskih zahtjeva napravljena je sistemska analiza u obliku modeliranja tokova podataka i procesa. Iz sistemske analize proizašao je konceptualni dizajn komponente maske. Dizajn maske sugerira mogućnost stvaranja radnog okvira za standardizaciju i smanjanje troška implementacije pojedine vrste maske. Za dvosmjerne transformacije podataka, koje su otkrivene analizom, predložena je mogućnost korištenja metoda bidirekcionalizacije radi daljnjeg smanjenja troška implementacije maske.

Predložena arhitektura i komponenta maske ostvareni su implementacijom sustava Janus kao dokaza koncepta. Janus omogućava implementaciju maski putem ostvarenog radnog okvira.

Tri vrste maski implementirane su korištenjem radnog okvira u Janusu. Korištenje radnog okvira za implementiranje maski pokazano je kako je implementacija maske svedena na devet općenitih koraka. Također, Janus omogućava implementaciju jednostavnih simetričnih leća, kao oblikovnog obrasca za dvosmjerne transformacije. Korištenje leća omogućava provjeru ispravnosti implementiranih dvosmjernih transformacija, što je do određene razine omogućeno jednostavnim radnim okvirom za testiranje leća u Janusu.

Sustav Janus korišten je kao prototip studije slučaja čime je dokazano da je sustav za integraciju heterogenih izvora podataka ostvariv putem arhitekture maska–miritelj–omotač. Nadalje, pretpostavljena kvalitativna svojstva arhitekture maska–miritelj–omotač dokazana su prototipima studija slučaja sustava SOS i mreže podataka. Ovi prototipi studija slučaja pokazali su kako generičnost komponenti maska–miritelj–omotača omogućuje emuliranje više različitih sustava, te da predstavljena arhitektura može služiti i široj uporabi u upravljanju podacima.

Ključne riječi: softverske arhitekture, integracija podataka, upravljanje podacima, miritelj–omotač, bidirekcionalizacija, funkcijsko programiranje, teorija kategorija

Contents

1. Introduction 1
1.1. Motivation 1
1.2. Contributions 2
1.3. Research methodology 3
1.4. Thesis structure 4

2. Software architecture preliminaries 7
2.1. Fundamentals of software architectures 7
2.2. Data integration and management systems 15
2.2.1. A note on metamodelling 17
2.2.2. Data source integration systems 18
2.3. Mediator–wrapper architecture 21
2.3.1. On the roles of mediator–wrapper components 24
2.3.2. On schema hierarchies in the mediator–wrapper architecture 25
2.4. Data mesh 28
2.5. Quantitative shift-cost analysis 31

3. Category theory, functional programming, and bidirectionalisation preliminaries 36
3.1. Basic category theory 36
3.2. Functional programming 42
3.2.1. Functors 47
3.2.2. Monads 49
3.3. Bidirectionalisation 52
3.3.1. Syntactic BX 57
3.3.2. Semantic BX 58
3.3.3. BX combinators 60
3.3.4. Other notable approaches 63
3.3.5. Lenses 63

4. Mask–mediator–wrapper architecture 69
4.1. Problems with the mediator–wrapper architecture 69
4.2. Extending the mediator–wrapper architecture 71
4.2.1. The mask's effect on the system schema hierarchy 74
4.3. Quantitative shift-cost analysis of the mediator–wrapper architecture 75

5. Mask component 83
5.1. Mask component functional requirements 83
5.2. Mask inner components 84
5.2.1. Data translation 90
5.3. Mask framework 90
5.4. Mask modalities 92

6. Prototype system 93
6.1. Janus system 93
6.1.1. Janus schema model 97
6.1.2. Janus data model 99
6.1.3. Janus query model 100
6.1.4. Janus command model 101
6.1.5. Janus communication 104
6.1.6. Janus components 106
6.1.7. Janus mediator component 108
6.1.8. Janus wrapper component 113
6.2. Proof of concept mask in the Janus system 116
6.2.1. Mask framework 116
6.2.2. Web REST API mask 119
6.2.3. LiteDB mask 139
6.2.4. SQLite mask 144

7. Method for bidirectional data transformations 149
7.1. Lenses for data transformation 149
7.2. Lens implementation in C# 151
7.3. Web API mask lenses 153
7.3.1. `RowDataDtoLens` 153
7.3.2. `TabularDataDtoLens` 157
7.4. Behavedness of Janus lenses 160

8. Mask–mediator–wrapper case studies 165
8.1. SOS system emulation case study 165
8.2. Data mesh emulation case study 168
8.2.1. Emulating a data mesh 171
8.2.2. Expected benefits 174
8.3. Case study prototypes 176
8.3.1. Janus as a data source integration system 176
8.3.2. Janus for SOS system emulation 179
8.3.3. Janus for data mesh emulation 180

9. Conclusion 186

10. Appendix 189
10.1. List of abbreviations 189
10.2. Code example prefix 190
10.3. Running the case studies' prototypes using Docker compose 191
10.4. Configuration tables for the case studies' prototypes 194
10.4.1. Data source integration system case study prototype configuration . . .194
10.4.2. SOS system case study prototype configuration 194
10.4.3. Data mesh case study prototype configuration 195
10.5. Additional details on the data source integration system case study prototype .197
10.5.1. Schema translation 197
10.5.2. Query translation and execution activities 207
10.5.3. Data translation 210

Bibliography 216

Biography 230

Životopis 231

Chapter 1

Introduction

1.1 Motivation

Data integration has always played a critical role in data management; a system handling multiple data sources is expected to integrate them at some point. Although a subset of data can be integrated by implementing integration for a specific use case, systems usually require the integration of data from multiple data sources. Finding a generic way of facilitating data integration of multiple data sets is paramount to achieving a unified view of the underlying data and deriving meaningful insights over large quantities of data.

Research on data integration has been ongoing for at least since the introduction of the first database systems. The field was subject to comprehensive studies in the 1990s [2, 3, 4, 5, 6, 7]. This led to the construction of two notable heterogeneous data source integration systems - GARLIC [8] and TSIMMIS [9]. Both systems were implemented with the mediator–wrapper architecture. Sadly, these systems were short-lived and weren't openly available.

This time period was also the advent of changes in data models for data storage. Researchers tried to pull away from the idea of relational databases and experiment with new paradigms. Researchers were led in part by hypotheses that the relational model is expressively restrictive, outdated, or unable to support large amounts of data. The result of this was the appearance of NoSQL systems [10]. Data was no longer just being stored in relational databases but also in schemaless formats in specialized database management systems and even files, and this opened new research questions as well as created room for innovation in data integration. Researchers started focusing more on the abstract term data source than the database.

Research into extract–transform–load processes [29] also tackled the integration of data from multiple data sources. But this topic primarily concerns data warehousing and data analysis [30, 31], and entails explicitly programming data integration through mechanisms such as map/reduce.

Research in the last decade has tried to deal with the growing amount of data, especially

with how to store, manage and handle it. Data lakes were introduced as a solution to store large quantities of distributed, unstructured, and heterogeneous data [11, 12]. This solution opened the possibility for the introduction of heterogeneous data storage at a larger scale, adding to the stress on data integration research to follow up. Notably, *Apache Linkis* has started tackling data integration [32], especially for *Hadoop*-based products.

Data integration research is yet to fully catch up and the landscape is once again shifting; toward graph-formatted data [33, 34, 35, 36], Web-based data [37, 38, 39], and new kinds of specialized data sources [37, 40, 41, 42, 43, 44]. The referenced research also points out a trend that a data management system is not only concerned with how to acquire and transform data but also with the representation of that data.

The exasperations and aspirations of data integration research were expressed by Golshan et al. [13]:

> "...it is time for data integration operators to break free of end-to-end data integration systems and be available in the open source to speed up adoption and progress."
>
> "The first challenge [...] is that progress of data integration and its application in practice are hindered by the fact that there are very few quality tools with which practitioners and researchers can freely experiment."

Summarily, the research has yet to produce a concrete, freely usable, and open-source data source integration system.

These ideas should be the guiding principles for developing a complete and robust data integration strategy. The end goal should be a comprehensive, evolvable, extendable, and open system for heterogeneous data source integration. The research described in this thesis examines, proposes, proves, and postulates the architectural framework and implementational design with which the long-desired heterogeneous data source integration system can be built.

1.2 Contributions

This thesis makes the following scientific contributions:

1. Extension of the mediator–wrapper architecture for data, schema and query representation via a component named mask, enabling the implementation of different kinds of heterogeneous data source integration system access interfaces;
2. Method for creating data transformations in masks using bidirectionalisation to reduce the effort of mask implementation;
3. Framework enabling the implementation of the previously defined mask and propositioned method, verified by an implementation prototype over a representative example of a hypothetical heterogeneous data integration system.

1.3 Research methodology

The research of state-of-the-art and seminal publications has shown that data source integration systems lack the flexible representational capabilities required in a modern technological setting. Previous research on data source integration systems had primarily focused on data acquisition, translation and integration, while mostly overlooking the importance of data representation. This is the primary research problem of this thesis - expanding the representational capabilities of a data integration system.

The doctoral research has revealed that the mediator–wrapper architecture remains the most suitable choice for data source integration in a modern technological setting. Therefore, any effort to expand representational capabilities should focus on this architecture. The suitability of the mediator–wrapper architecture stems from its components' adherence to concrete rules and the capability to support schema hierarchies at the component level.

The component rules and schema hierarchy examples provided the opportunity to qualitatively analyse and discuss the existing architecture, as well as reason about the state-of-the-art architecture proposed by this research. The mask–mediator–wrapper architecture is the proposed architecture providing the expansion of representational capabilities through the extension of the mediator–wrapper architecture. The expansion of representational capabilities is facilitated by the ability of the proposed architecture to provide multiple representations of system queries, commands, schema, and data. The proposed architecture is qualitatively analysed and compared with the mediator–wrapper architecture. The proposed architecture is also analysed through case studies' prototypes. The case studies' prototypes include not just a data source integration system, but also two separate data management systems: the legacy-preserving Save Our Systems (SOS) system, and a data mesh.

The mask–mediator–wrapper architecture implied the introduction of a new component type, called a mask. The mask component presents another research problem, in terms of its feasibility, design, and uniformity. As with the existing mediator and wrapper component types, rules for the mask were proposed in accordance with the planned capabilities of the proposed architecture. The mask was further detailed through functional requirements stemming from the inferred rules. The functional requirements enabled further examination of the mask's inner components and data flows. This examination showed that the mask could hypothetically be implemented generically, requiring only the implementation of a set of translators and a masked application interface per mask kind. This hypothesis was expanded on to propose that mask kinds can be constructed using a generic framework to simplify mask development and contribute to the uniformity of mask implementations.

The mask is proposed to contain one-way translations for schemas and queries (including commands). On the other hand, data translation requires a two-way transformation. The cor-

rectness of such transformations is in question if written ad-hoc, as well as the effort required for their implementation. This presents another research problem of this thesis. These problems were proposed to be solved through the use of bidirectionalisation methods. Concretely, the overview of the field of bidirectionalisation pointed in the direction of lenses as a design pattern to facilitate the two-way transformations. The simple symmetric lens was chosen as the lens to facilitate such transformations in the mask, due to its simplicity and symmetrical character. Examples of lenses were implemented in C#, as well as a generic testing framework for testing the behavedness of lenses.

A prototype mask–mediator–wrapper system was implemented as a substitute for the hypothetical data source integration system of the third contribution, making the contributions of this thesis more tangible. The implemented system is called Janus. Janus has served as a prime example of a mask–mediator–wrapper system during the research. Janus was implemented to contain a mask framework for constructing masks. To exemplify the development of masks by using the framework, three mask kinds were created as prototypes. Additionally, the development of a mask has been distilled into 10 steps. An implemented lens was used to facilitate data transformations in the REST Web API mask, proving that lenses with a determined level of behavedness can be used in masks.

Concrete case studies' prototypes were configured to prove the statements made about the mask–mediator–wrapper architecture in the qualitative analysis and case studies. The first case study prototype demonstrates the Janus system's capability of being deployed as a data source integration system; as primarily hypothesised in the qualitative analysis. The second case study prototype demonstrates the capability of the Janus system to emulate the SOS system. The third case study prototype demonstrates the capability of the Janus system to emulate a data mesh. The second and third case study prototypes are used to show an originally unanticipated versatility of the mask–mediator–wrapper architecture.

1.4 Thesis structure

Chapter 2 introduces the preliminary terms in software architectures. Section 2.1 provides the essentials for observing architectural characteristics and component properties, and includes the modern perspective through evolutionary architectures. Section 2.2 concerns data management and integration systems, where the mediator–wrapper architecture is introduced as a data integration system (Section 2.3), and the data mesh is introduced as a data management system (Section 2.4). Section 4.3 introduces the quantitative shift cost analysis as a way of quantifying software flexibility.

Chapter 3 introduces preliminaries regarding category theory and functional programming. Sections 3.1 and 3.2 introduce definitions and explanations of the terms and mechanisms exten-

sively used in bidirectionalisation. Bidirectionalisation is introduced through a set of methods in Section 3.3. Section 3.3.5 examines bidirectional lenses.

Chapter 4 introduces the mask–mediator–wrapper architecture; discussing the first contribution of this thesis. The deficiencies of the mediator–wrapper architecture are discussed from a qualitative perspective in Section 4.1. The mask–mediator–wrapper architecture is introduced in Section 4.2 as an extension of the mediator–wrapper architecture to resolve the examined deficiencies. A quantitative shift-cost analysis is performed in Section 4.3 as further proof that the mask–mediator–wrapper architecture is an improvement over the mediator-wrapper architecture.

Chapter 5 is concerned with the state-of-the-art mask component type with which the mediator–wrapper architecture was extended. The chapter advances the first contribution theoretically and sets the theoretical background for the third contribution. Sections 5.1 and 5.2 theoretically examine the mask component itself. Section 5.3 discusses the strategy for constructing masks via a framework. Section 5.4 examines the modalities of masks in terms of previously observed integration systems' modes of use.

Chapter 6 introduces the system Janus implemented during this doctoral research. Janus is a prototype built using the proposed architecture. The design, models and components of the Janus system are discussed in Section 6.1. Section 6.2 presents the implementation of a framework enabling the implementation of masks in the Janus system, the process of implementing a mask, and three concrete mask implementation prototypes; finalizing the third thesis contribution.

Chapter 7 is concerned with the use of lenses as the bidirectional method of choice for data transformations in a mask. The choice of the appropriate lens type is discussed in Section 7.1. The implementation of a lens using the C# programming language is discussed in Section 7.2. Lenses used in an implemented mask (in the Janus system) are presented in Section 7.3. Section 7.4 discusses the behavedness provability of the implemented lenses.

Chapter 8 examines case studies to analyse the capabilities of the mask–mediator–wrapper architecture. Two case studies present the capability of the proposed architecture to emulate two additional data management architectures, beyond serving just as a data source integration system. Sections 8.1 and 8.2 present the two case studies. Section 8.3 introduces prototypes of the three case studies as proof of the proposed capabilities of the mask–mediator–wrapper. Section 8.3.1 proves the primarily proposed data source integration system capabilities by deploying the Janus system components as such a system. This section finalizes the first contribution of this thesis. Section 8.3.2 proves the capabilities of emulating the SOS data management system by appropriate deployment of the Janus system components. Section 8.3.3 proves the capabilities of emulating a data mesh by appropriate deployment of the Janus system components.

Chapter 9 presents the conclusions of this doctoral research.

The appendix (Chapter 10) contains the list of all abbreviations used in the text; they are not introduced directly in the text for the sake of readability (Section 10.1). The appendix contains the common code prefix for the Haskell code examples (Section 10.2), the instructions for running the Janus system components as the case studies' prototypes by using Docker Compose (Section 10.3), tables for noting the configuration of the components deployed in the case studies' prototypes (Section 10.4), and additional details about the Janus system's schema and query translation through an established case study prototype (Section 10.5).

Chapter 2

Software architecture preliminaries

In this chapter, some basics of observing software systems in terms of architecture are presented. An introduction to the basic and modern concepts of software architecture is given. In addition, an overview of data management systems in the context of data integration is given based on the conference paper on taxonomy [1]. The chapter includes a section on the mediator–wrapper architecture, which is extended by the contributing architecture of this thesis. The overview of the mediator–wrapper architecture is from a published journal paper [14] written during this doctoral research. A section on the data mesh for data management is given; this is summarised from a preprint [28]. The data mesh is used as a part of a case study in Chapter 8. The chapter concludes with an overview of the quantitative analysis that was adapted to determine certain properties of the architecture used in this thesis.

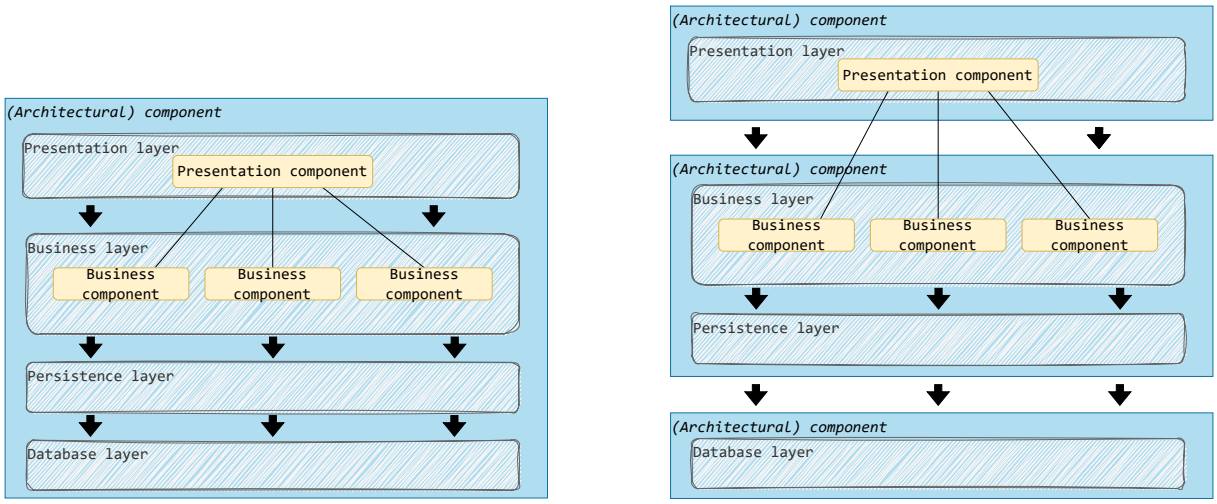
2.1 Fundamentals of software architectures

On modules, components, and layers

Software architectures are usually observed in terms of modules and components, and the placement of these in a computer environment. The most atomic observed element of software architecture is the *module* - a logical grouping of related code [15, 16, 17]. Multiple modules can be part of a component. A *component* is a physical packaging of modules [17]. Components are physically present in computer systems as files. Modules and components are sometimes used interchangeably by some authors, but Richards and Ford [15] and Ford et al. [16, 17] do differentiate between the two. Because the compilation of its code creates a physical file, the component can theoretically be placed on different machines. In modern development environments that enable portability, components contain some form of pre-compiled intermediate code (e.g. OpenJDK Java Bytecode or C# .NET Common Intermediate Language). The term component is also used interchangeably to denote architectural components. An architectural component represents a functional physical element of an architecture and can be comprised

of one or more components. Components within architectural components are also referred to as "inner components" in the remainder of this thesis to make the distinction clearer in certain parts of the text.

Two examples of architectural components and inner components of a hypothetical multi-layer information system are given in Figure 2.1. A system can be logically layered, where each layer contains a set of components. These layers can become components themselves, depending on the build configuration (in Java this is referred to as an uber or slim JAR). In the example of Figure 2.1a all components are kept as a single architecture component; presumably on a single machine. Figure 2.1b gives an example where layers and components are also physically separated; on different machines.



(a) Single architectural component

(b) Multiple architectural components

Figure 2.1: Relation example of inner and architectural components

Layering, componentization and modularization are used to facilitate the separation of concerns (responsibilities). The reasoning behind this is that each layer, component, and module concerns a certain specific functionality of a software system. To this end, a system can be functionally granulated into finely-grained components. Such components are concerned with one specific functionality each. This reasoning can be run into the extreme end where even the smallest function is separated into its own inner component. Ford et al. [17] state that granulation produces positive outcomes in the code until it causes the loss of code cohesion. Analogously, architectural component granulation can cause latency if applied excessively [15].

Coupling and cohesion

Coupling and cohesion are fundamental properties observed in software. *Cohesion* is the degree to which the elements inside a module belong together [45]. Cohesion can be defined as a range from positive to negative [15]:

- *Functional cohesion* - Every part of the module is related to the other, and the module contains everything essential to function.
- *Sequential cohesion* - Two modules interact, where one outputs data that becomes the input for the other.
- *Communicational cohesion* - Two modules form a communication chain, where each operates on information and/or contributes to some output.
- *Procedural cohesion* - Two modules must execute code in a particular order.
- *Temporal cohesion* - Modules are related based on timing dependencies.
- *Logical cohesion* - The data within modules is related logically but not functionally.
- *Coincidental cohesion* - Elements in a module are not related other than being in the same source file.

Coupling is the degree of interdependence between modules [45]. Yourdon and Constantine defined *afferent* and *efferent* coupling as code metrics. Afferent coupling measures the number of incoming connections to a code artefact. Efferent coupling measures the outgoing connections to other code artefacts. These metrics allowed Martin [46, 47] to define his own metrics of *instability* and *abstractness*.

Definition 1. *For afferent coupling C^a and efferent coupling C^e, I is the measure of code instability calculated by:*

$$I = \frac{C^e}{C^e + C^a} \qquad \text{(INSTABILITY)}$$

Definition 2. *For a number of abstract classes (including interfaces) in a module m^a and a number of concrete classes in a module m^c, A is the measure of code abstractness calculated by:*

$$A = \frac{\sum m^a}{\sum m^c} \qquad \text{(ABSTRACTNESS)}$$

Richards and Ford [15] state that these metrics can be used on other code elements (e.g. lines of code), not just classes.

Building on these coupling metrics, Martin [47] proposed the *distance from the main sequence* metric (Figure 2.2). The distance from the main sequence is the perpendicular distance from a line idealizing the main sequence: $A + I = 1$. Balanced code in terms of instability and abstractness is positioned on the main sequence line.

Definition 3. *The distance from the main sequence is calculated by:*

$$D = |A + I - 1| \qquad \text{(DFMS)}$$

where A is the measure of abstractness and I the measure of instability.

Coupling and cohesion are intertwined concepts because attempting to divide a cohesive module would only result in increased coupling and decreased readability [45].

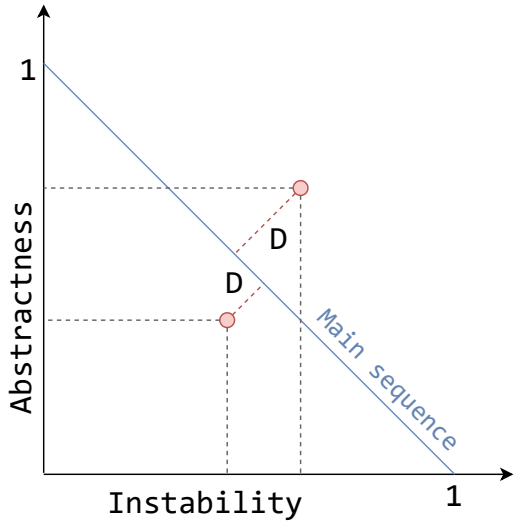


Figure 2.2: Illustration of the distance from the main sequence

Additionally, Page-Jones [48] recoined coupling as *connasence*; by ignoring coupling direction and concentrating on the reason for the coupling. Two components are connascent if a change in one would require the other to be modified in order to maintain the overall correctness of the system. Most importantly, connasence is recognized having two types: *static* and *dynamic*. Static connasence is coupling created in code and can be detected by static analysis of the code. Dynamic connasence is coupling that appears at runtime (e.g. remote service calls, or library references).

Coupling can be transferred into architectural reasoning by observing architectural components instead of modules. This is sensible because components are physical manifestations of modules [17]. Coupling between components is created through compile-time or runtime dependencies [17]. Compile-time dependencies are usually created by references to library components. Runtime dependencies are usually manifested through references to remote services. Architectural components also manifest cohesion and the statement that the division of a cohesive component causes increased coupling (by Yourdon and Constantine [45]) still holds.

Architectural quantum and evolutionary architecture

Ford et al. [17] observed architectural coupling to facilitate the term - *architectural quantum*. An architectural quantum is an independently deployable component with high functional cohesion, which includes all structural elements required for the system to function properly. The expression "all structural elements" should be stressed in this definition, because unlike an inner or architectural component, an architectural quantum contains multiple elements of a system. An architectural quantum may include databases, search engines, reporting tools [17], and any such service that might not be directly tied to the codebase.

An architectural quantum is typically exemplified with the comparative example of a mono-

lithic and microservice architecture (Figure 2.3 and Figure 2.4). A monolithic architecture contains just one architectural component (the monolith) and its dependent parts. A microservice architecture consists of multiple services called microservices, each responsible for functionalities within a separate bounded context [49]. In the monolith architecture, the monolith itself is the architectural quantum along with its dependent parts. In the microservices architecture, each microservice with its dependent parts is an architectural quantum.

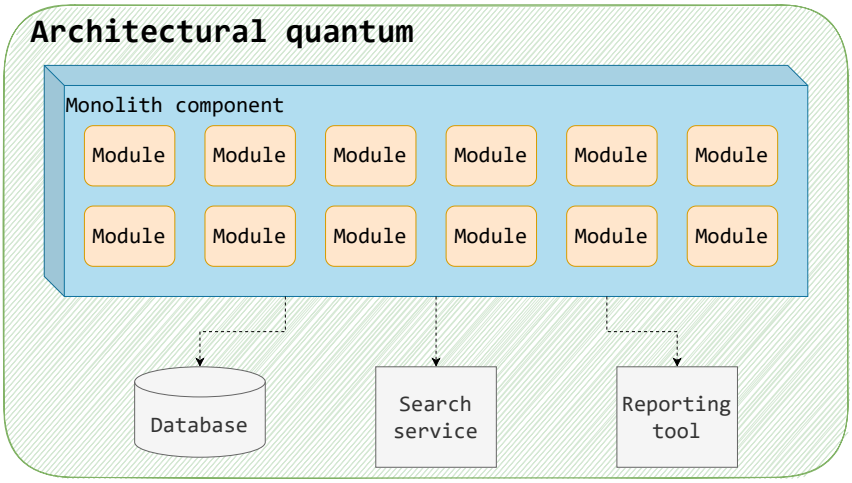


Figure 2.3: Monolith architecture with denoted architecture quantum

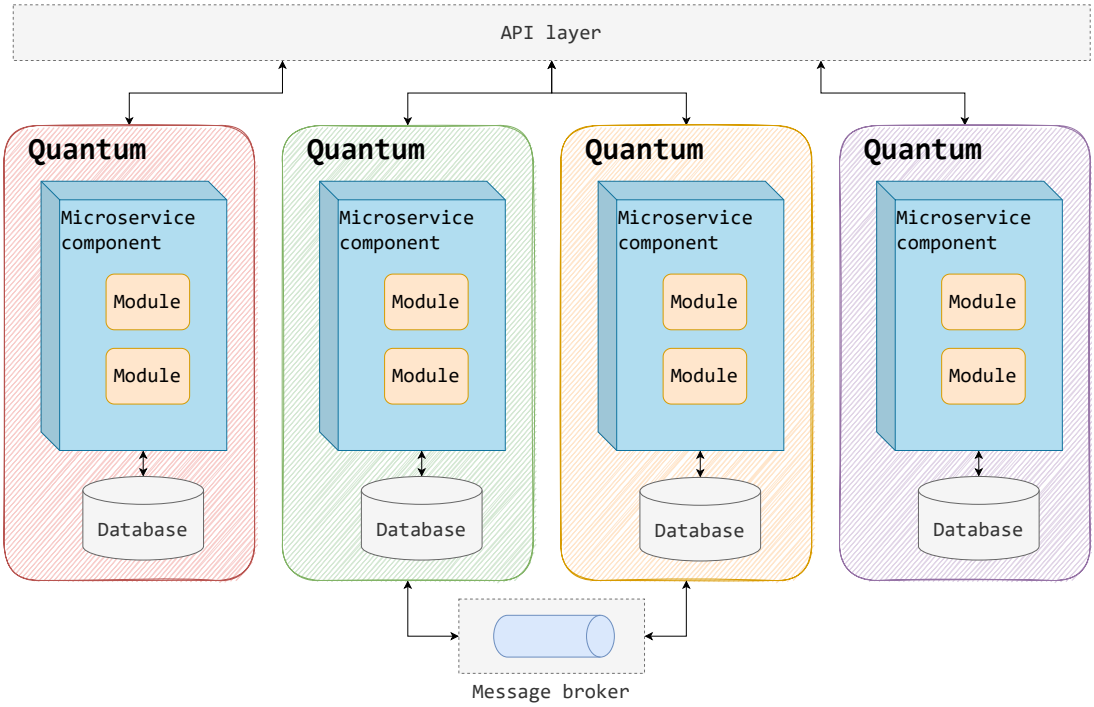


Figure 2.4: Microservices architecture with denoted architecture quanta

Evolutionary architectures use the principle of the architectural quantum to support incremental changes across multiple dimensions [17]. Dimensions of changes generally include functional changes and technological changes. This means that not only requirement shifts

need to be accommodated in the architecture, but technological shifts as well. Technological shifts entail changes in the technologies and third-party components used in the system. If such elements are abstracted to reasonable levels, then the entire system is able to "evolve" over a long period of time. Additionally, evolutionary architecture principles introduce *fitness functions*. Architectural fitness functions provide an objective integrity assessment of some architectural characteristics [17]. Fitness functions use unit tests, integration tests, architecture metrics, contract tests, process metrics, and monitors as tools to assess the fitness of an architecture implementation. Fitness functions can be run continually during a system's runtime or can be triggered at certain points of development (e.g. during DevOps deployment).

Evolvability is the defining characteristic of evolutionary architectures. Evolvability is recognized as being impacted by the following architectural dimensions [17]:

- *Technical* - The implementation parts of the architecture: the frameworks, dependent libraries, and the implementation language(s).
- *Data* - Database schemas, table layouts, optimization planning, etc. The database administrator generally handles this type of architecture.
- *Security* - Defines security policies, guidelines, and specifies tools to help uncover deficiencies.
- *Operational/System* - Concerns how the architecture maps to existing physical and/or virtual infrastructure: servers, machine clusters, switches, cloud resources, and so on.

Architectural characteristics

Richards and Ford [15] assessed multiple architectural styles through a set of architectural characteristics. Out of many alternative architectural characteristics sets, even including the ISO25010 standard [50], the authors [15] drew the following set of architectural characteristics:

- *Deployability* - Degree of ease with which the system is deployed.
- *Elasticity* - Ability of the system to handle sudden increases in the amount of data or users.
- *Evolvability* - Ability of the system to evolve through its life-cycle (adherence to the evolutionary architecture principles).
- *Fault tolerance* - Ability to operate as intended despite hardware or software faults.
- *Modularity* - Degree of cohesion and decoupling in the system modules.
- *Overall cost* - Monetary, time, and effort cost to build and deploy the system.
- *Performance* - Performance of a system in measurable technical terms (e.g. processing time, response time).
- *Reliability* - Degree to which a system functions under specified conditions for a specified period of time.
- *Scalability* - Ability to handle increasing amounts of data and users over time.

- *Simplicity* - Degree to which the system is simple to build, maintain, and deploy over.
- *Testability* - Degree to which the system is able to be tested.

These characteristics are not a finite set; the set can be changed (and is expected to change) according to the requirements at hand. While Richards and Ford [15] do evaluate multiple architecture styles through a common quantified grading system (from 1 to 5 stars), their evaluations are inherently qualitative. This is because they don't set a concrete quantitative analysis framework for their evaluations.

Component identification workflow

Richards and Ford [15] propose a generic component identification workflow for the inference of components (Figure 2.5).

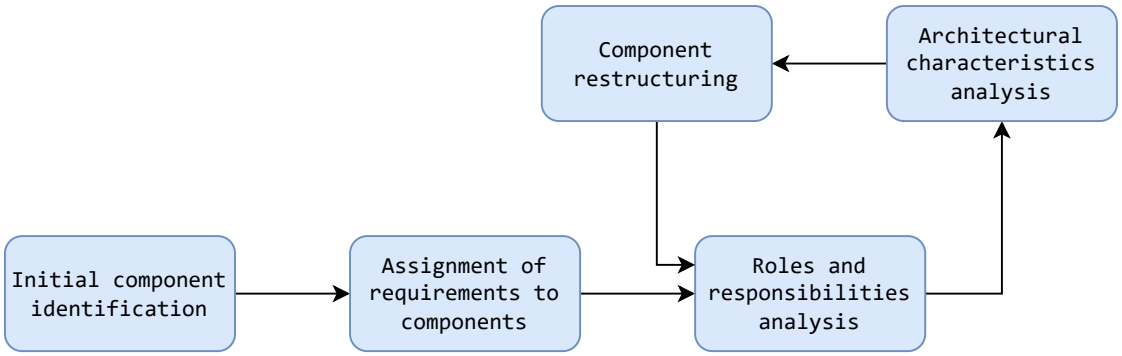


Figure 2.5: Component identification workflow [15]

As illustrated in Figure 2.5, the workflow contains the following steps:

- *Initial component identification* - The architect must determine top-level components of an architecture. The point of this initial step is simply to start the process of component identification, not to produce a perfect architecture on the first attempt.
- *Assignment of requirements to components* - The architect must align requirements with the inferred components. This can include creating new components, consolidating existing ones, or decomposing components.
- *Roles and responsibilities analysis* - The architect refines the architecture even further by taking into account the roles and responsibilities of components. Roles and responsibilities are inferred from the requirements.
- *Architectural characteristics analysis* - In this step, the architect also takes into account architectural characteristics and further refines the architecture accordingly.
- *Component restructuring* - The architect should cooperate with the developers to determine if there are any previously unforeseen technological obstacles to developing the system in the proposed architecture. The workflow cycles back to the roles and responsibilities analysis and continues until the architecture is adequately refined.

This workflow is effectively used throughout the research presented in this thesis. Initial component identification, assignment of requirements to components, roles and responsibilities analysis, and architectural characteristics analysis are manifested in Chapter 4 and 5. The result of component restructuring and the outcome of the cycle iterations through development is evident in Section 6.1.

Well-formed components

Meyer [18] proposed architectural reasoning in terms of architectural components in a general sense; describing the properties of what can be considered *well-formed components*. A system component should satisfy the following conditions [18]:

RC1 It can be used by other software elements, its “clients”.

RC2 It possesses an official usage description, which is sufficient for a client author to use it.

RC3 It is not tied to any fixed set of clients.

The conditions set by Meyer predate those by Richards and Ford, but they are not at odds. They simply exhibit the same notions in different terms. **RC1** stresses the importance of what Richards and Ford would relate as modularity and deployability. **RC2** would be akin to simplicity. **RC3** would be relatable to scalability and modularity.

The mediator component

This section is included to clarify some ambiguities on a component called a "mediator". This component name will be mentioned extensively in terms of data integration and management systems in the subsequent sections. Unfortunately, the term "mediator" is a homonym for another component and architectural pattern used in the general field of software architectures. The use of the term in data integration and management systems predates its use in the general field of software architectures.

In modern software architectures, the mediator is a component of the event-driven architectural style. It is used for process orchestration between multiple services. Although the component’s name is the event mediator, it is colloquially referred to as just the mediator.

The event-driven architecture is based on event handling through message queue (broker) components. Components subscribe to certain message topics defined in the message queues. When a message on a certain topic appears, the subscribers are notified and are able to pull the message from the queue for a certain time. Components can also publish messages with a topic to the message queues. This focuses the system design on messages being relayed across the topology. Events precipitate the creation of messages in the system, so messages are a reflection of events occurring in the system’s surrounding environment.

Event-driven architecture with a mediator topology (also referred to as orchestration) utilizes a mediator component to receive an initial event and coordinate operations spanning mul-

tiple components regarding the initial event (Figure 2.6). The mediator generates corresponding processing events for dedicated event channels (queues) handled by the components required for each operation. These operational components are referred to as event processors. In essence. The mediators can be tasked with handling complex action paths, so they usually support specifying entire business processes via the Business Process Execution Language (BPEL).

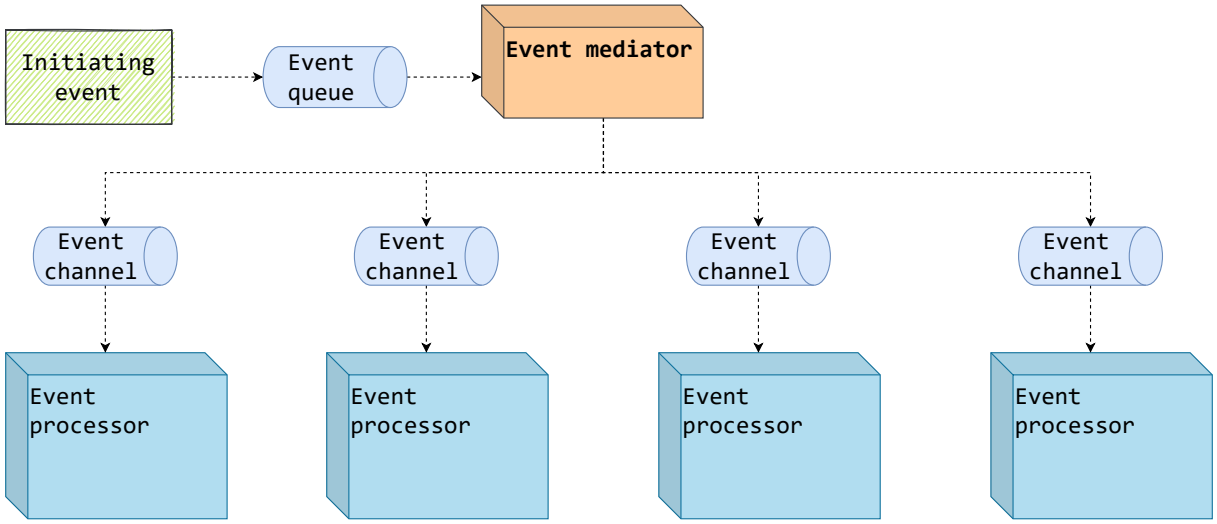


Figure 2.6: Mediator in the event-driven architecture [15]

The alternative to the mediator topology is the broker topology. It necessitates the use of multiple message brokers with multiple topics to which event processors are independently subscribed. It is also referred to as choreography.

The referred "mediator" in the subsequent sections isn't the event mediator but a different, although similar, component.

2.2 Data integration and management systems

Numerous authors usually preface their work in the field of data management with a statement that the significance of data in technological fields is increasing. This is illustrated by Statista's incrementally updated report on the volume of data worldwide in Figure 2.7 [51]. With each yearly report, the projected years in the graph form a steeper curve than in the previous report year. This trend places emphasis on the management of data and imposes some general research questions - How can data be stored? How can data be exchanged? How can data be viewed? How can data be analyzed? How can data be utilized?

The primordial efforts of tackling data management can be found in database management systems, leveraging relational algebra to store and manipulate structured data. The work by Edgar Codd [52] is usually presented as the beginning of major efforts in empowering relational database systems. These efforts were fruitful and relational database management systems be-

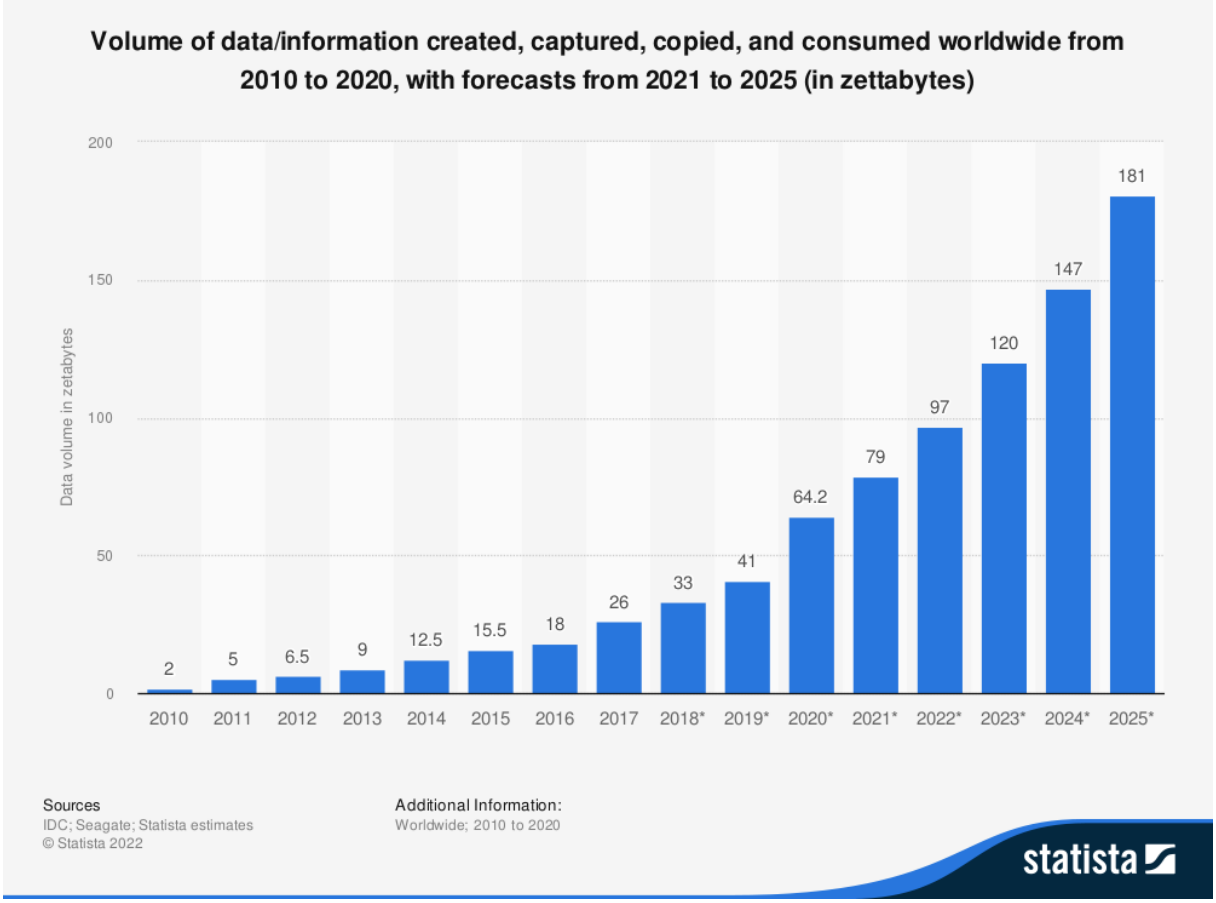


Figure 2.7: Volume of data from 2010 to 2025, as presented by Statista [51]

came the norm by the 1990s. The acceptance of relational systems in software systems led to their proliferation by many vendors; some systems had even become unsupported by their authors and turned into legacy systems. This precipitated the problem of managing legacy data stores and presenting them with a modern application interface. In parallel, the possibility of integrating multiple data stores became an attractive research topic. The work of Sheth and Larson [2] presents an overview of the scientific and technological efforts and reasoning of the day. Sheth and Larson [2] concerned themselves with, what they called, federated multi-database systems. Data integration, schema integration, and querying were predominantly studied on relational databases.

During this period, the management of non-structured data became a prominent topic, caused by the advent of NoSQL systems by Carlo Strozzi [10]. The absence of schemas in these systems enables the structure of data to vary, so the term *schemaless* is used to denote this property. This only added complexity to the data integration challenges, as traditional relational databases and structured data storage systems could not easily handle non-structured data. However, NoSQL databases provided a flexible and scalable solution for handling unstructured or semi-structured data, such as text, images, videos, and social media data. As a result, organizations started to adopt NoSQL databases to store and process large amounts of

unstructured data. Overall, the rise of NoSQL databases and the management of non-structured data have brought new challenges and opportunities to the data management field.

An alternative idea to data integration was data exchange/interchange between data sources [4]. This solution became impractical because it required direct connections between different data sources acting as peers. In parallel, ETL processes [29] over multiple data sources for data warehousing also appeared and are to this day an active research topic [30, 31].

Research has currently been exploring the idea of data lakes and how to process such large quantities of distributed, unstructured, and heterogeneous data [12, 53], but even these have recently been found wanting and inciting the concept of the data mesh [54]. Research is currently shifting toward graph data [33, 34, 35, 36] and Web-centered data [37, 38, 39].

It is clear that the research of data integration is very unlikely to abate, especially as new kinds of potential data sources continue to be created [37, 40, 41, 42, 43, 44], and as new consolidating solutions appear [32].

The research community expresses data integration in terms of data sources or data stores, not just databases. These terms are often used interchangeably, but this thesis differentiates between the terms data source and data store with the following definitions:

- A *data store* is a system or mechanism by which data is stored.
- A *data source* is any system or mechanism that can source data.

Any data store can be a data source, but not every data source is a data store. Most notably, a data stream is not a data store but a data source.

2.2.1 A note on metamodelling

Data management systems typically try to encompass a greater variety of data and its use cases, so it is practical to think of data as an abstract construct. This thesis delves into terms like metadata, models and metamodels, so it is appropriate to introduce such terms at this point. The following definitions are adapted in their succinct form from Gonzalez-Perez and Henderson-Sellers [55, 56].

Definition 4. *A **model** is a representation of a system under study (SUS).*

Definition 5. *A **metamodel** is a description of a model.*

Definition 6. *A **metametamodel** is a description of a metamodel.*

Gonzalez-Perez and Henderson-Sellers [55, 56] also point out that the SUS is abstracted by a cognitive model of the observer before being put down as a specific model. Despite this, it is practical to consider just the direct mapping between these models, ignoring the possible intermediate models.

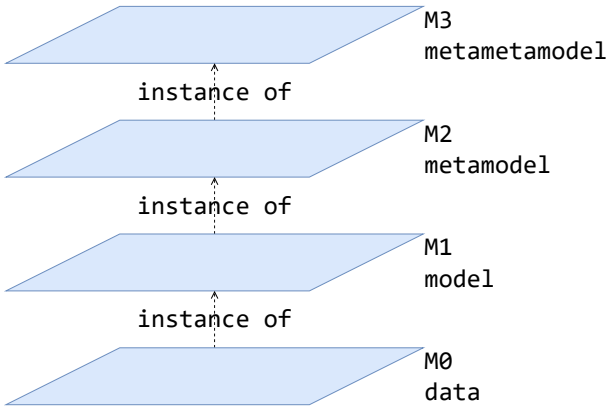


Figure 2.8: Object Management Group's four-layer hierarchy

Definitions 4, 5, and 6 can be understood in their reverse, as usually illustrated by the Object Management Group (Figure 2.8). The nature of their specific research doesn't concern an abstract system under study but concrete data, which is more aligned with the topic of this thesis. The observation of metamodelling in the domain of data allows the introduction of a *instance of* relationship between the data, model, metamodel and metametamodel. The most common observation of such relationships is found in the Unified Modelling Language.

Additionally, the term *metadata* is also used throughout the thesis. This term denotes a specific instance of data describing some other data. E.g. this can be a schema describing the structure of the data. In turn, the model used to describe the schema is a metamodel.

2.2.2 Data source integration systems

The taxonomy of data source integration systems has suffered greatly from undefinedness and incompleteness, so when reading through past research one can stumble upon different naming conventions (especially those promptly made for the material at hand). To illustrate a few examples, one author considers multi-database systems a category [19], while another author considers them a subcategory [2]. One author names a concept as a federated system, another author names the same concept as a multi-database system [57], and a third author considers them separate concepts [7]. The solution to these taxonomical qualms was proposed by Dončević and Fertalj [1] with a revised taxonomy. The authors presented the taxonomy in terms of database integration systems. Accordingly, this subsection presents this taxonomical revision in the generalized term of data sources where applicable.

A *data source integration system* is a data management system with the task of integrating data from two or more data sources (Figure 2.9).

Data source integration systems integrate three basic components that can hypothetically be inferred from data sources: *schemas*, *queries*, and *data*. Data in these systems is usually acquired as query results.

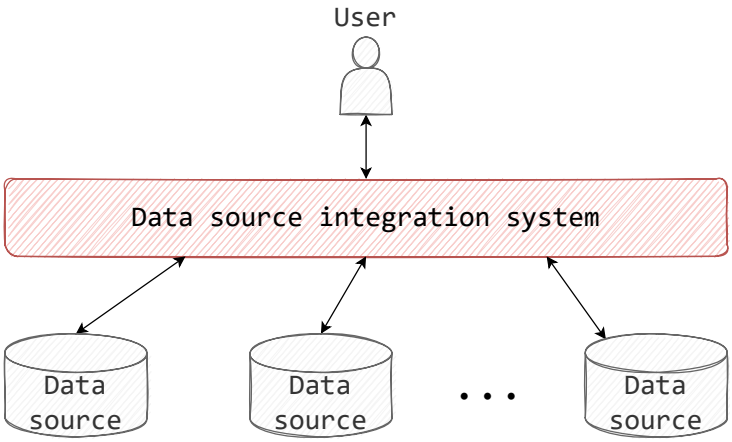


Figure 2.9: Conceptual illustration of a data source integration system

Data source integration systems can be divided into two groups based on their mode of use: *virtualising* and *materialising*. Virtualising systems create unified views (virtualisations) of multiple data sources but keep the data in the connected data sources. Materializing systems produce (materialise) unified data stores as a product of their integration, hence the data is kept in the produced data store.

Data source integration systems, according to Özsu and Valduriez [19], are determined by three dimensions (Figure 2.10): *heterogeneity*, *autonomy*, and *distribution*.

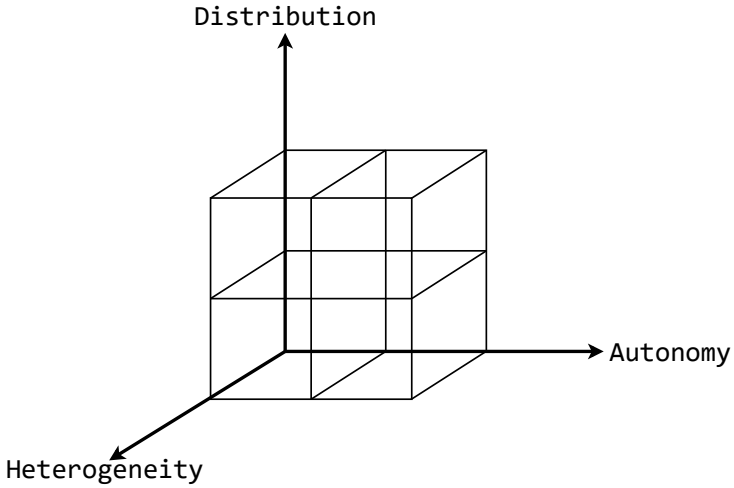


Figure 2.10: Dimensions of data source integrations systems [19]

The kind of data source heterogeneity an integration system supports can be divided into five categories: *technical* [2, 58], *structural* [6, 57], *semantic* [2], *syntactic* [58], and *paradigmatic* [1]. Technical heterogeneity indicates that a system integrates data sources with different technical implementations; this is the precondition for the system to be considered heterogeneous. Structural heterogeneity indicates that the models of the connected data sources are different. Semantic heterogeneity implies different modelling of semantically equivalent data in the connected data sources. Syntactic heterogeneity implies a difference in the connected data sources'

query languages. Paradigmatic heterogeneity was introduced as an additional marker to indicate that an integration system functions over paradigmatically different sources [1]. Paradigmatic heterogeneity implies that the integration system also belongs to the other categories. Dončević and Fertalj [1] proposed these be called *hybrid* data source integration systems.

Three categories are regarded in the autonomy dimension of data source integration systems [2, 57]: *design autonomy*, *communication autonomy*, and *execution autonomy*. Design autonomy implies that connected data sources don't require model corrections to be included in the integration system. Communication autonomy implies that connected data sources continue to have the same level of openness to other clients besides the integration system. Execution autonomy implies the ability of connected data sources to continue executing tasks from other clients besides the integration system without hindrance. The degree of autonomy is determined by behaviour alternatives [19]: *tight integration*, *semi-autonomy*, and *total isolation*. Tight integration disables the connected data sources' ability to control their own data and access to it. Semi-autonomy implies that connected data sources relinquish control of only a part of their data to the integration system. Total isolation implies that the connected data sources can still act as completely stand-alone systems; they aren't aware that they are part of an integration system.

The distribution dimension concerns the distribution of data in the data source integration system. This dimension infers three implementation alternatives: *undistributed system*, *client-server system*, *peer-to-peer system*. The undistributed alternative is not of any particular interest, as the existence of multiple data sources requires the latter alternatives. The client-server alternative involves just one system component for the purpose of integration, but the component can be instantiated multiple times (Figure 2.11). These instances have no means of intercommunication. The peer-to-peer alternative involves multiple integration system components that communicate to form a unified system (Figure 2.12). In the peer-to-peer alternative, each system component is dedicated to a single data source. In terms of database integration systems, the client-server alternative produces the multi-database integration system, and the peer-to-peer alternative produces the federated database integration system. These can be taxonomically brought into the abstract definition of a data source as *multi-data-source integration system* and *federated data source integration system*, respectively.

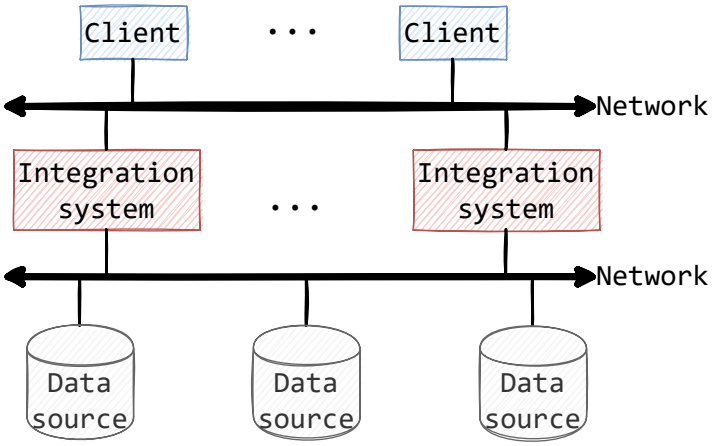


Figure 2.11: Client-server alternative

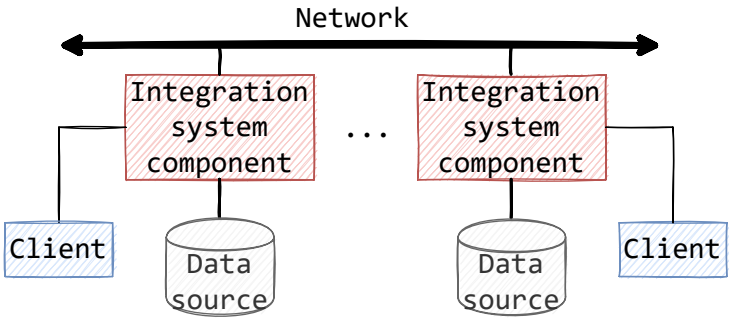


Figure 2.12: Peer-to-peer alternative

2.3 Mediator–wrapper architecture

The mediator–wrapper (MW) architecture doesn't originate from the aforementioned dimensioning of data source integration systems. Rather, it is a product of lateral reasoning about the functionalities of an integration system.

The MW architecture was first envisioned as an information system architecture [3], allowing a modular architecture for sub-tasking when numerous data sources are imposed, in opposition to monolithic architectures. This was specifically intended for information and knowledge management systems for informed decision-making.

Expanding on the idea of the MW architecture's applicability, Papakonstantinou et al. [4] observed its usage for the exchange of data across heterogeneous information sources. Roth and Schwarz [5] also observed the MW architecture to uniformly access legacy stores through the GARLIC system [8]. Similarly, the MW architecture was used as a basis for the TSIMMIS project [9]. Garcia-Molina et al. [59] recognized the MW architecture primarily for the purpose of data source integration systems.

The MW architecture in the most general sense is an architectural pattern, consisting of mediator and wrapper components, used to query and acquire data from multiple data sources. The

wrapper component is directly connected to a data source and acts as a standardized interface to that data source. The wrapper wraps (or encapsulates) the data source for further use throughout the rest of the system, effectively making it the only component in immediate contact with the data source. To guarantee such functionalities, the wrapper must be able to translate queries, data and metadata coming to and from the data source, as well as the layers above.

The mediator component is architecturally situated above the wrappers. The mediator's task is to connect multiple wrappers and integrate their data and metadata. Because data, metadata and queries are logically intertwined, the mediator also must have the ability to decompose and allocate queries to its connected wrappers. Both the wrapper and mediator components come with an implicit consideration that they are generically implemented to enable multiple configurable deployments.

Certain aspects of the MW architecture can be clarified by following the top-to-bottom flow of data as shown in the conceptual illustration of the MW pattern in Figure 2.13. The mediator receives a query which is then propagated accordingly to its connected wrappers. Not all wrappers need to be included in a query, as all the data required by the query might not be in all the data sources. The queried wrappers then translate the queries according to their data source's schema and querying language. The returned result is then translated back into the system's standardized result format and propagated to the mediator and above.

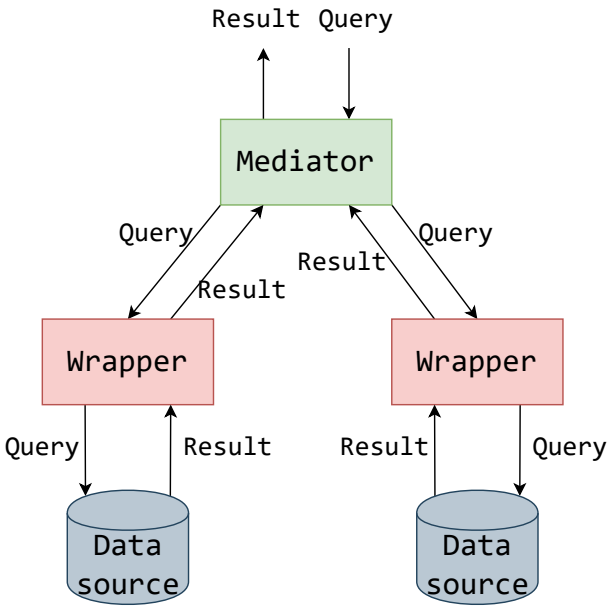


Figure 2.13: MW pattern [59]

This pattern of interaction is the basis for the complete MW architecture. A more global view is shown in Figure 2.14, illustrating the layering of mediator and wrapper components. The data sources to be integrated are at the lowest layer. Each data source is directly covered by a single wrapper. As an example, a system where each wrapper operates over a single data source is displayed (Figure 2.14), although Özsu and Valduriez [19] display a possibility of

a wrapper operating over multiple data sources. It can be observed that the "one wrapper – one data source" setting gives more agility for appending new data sources to the integration system, as it allocates the responsibility of overseeing data sources to each wrapper separately and thus balances the workload. It is also interesting to remark that this component setting is better suited for systems being built bottom-up [7, 19], where data sources are expected to be appended and the global data overview is expected to change.

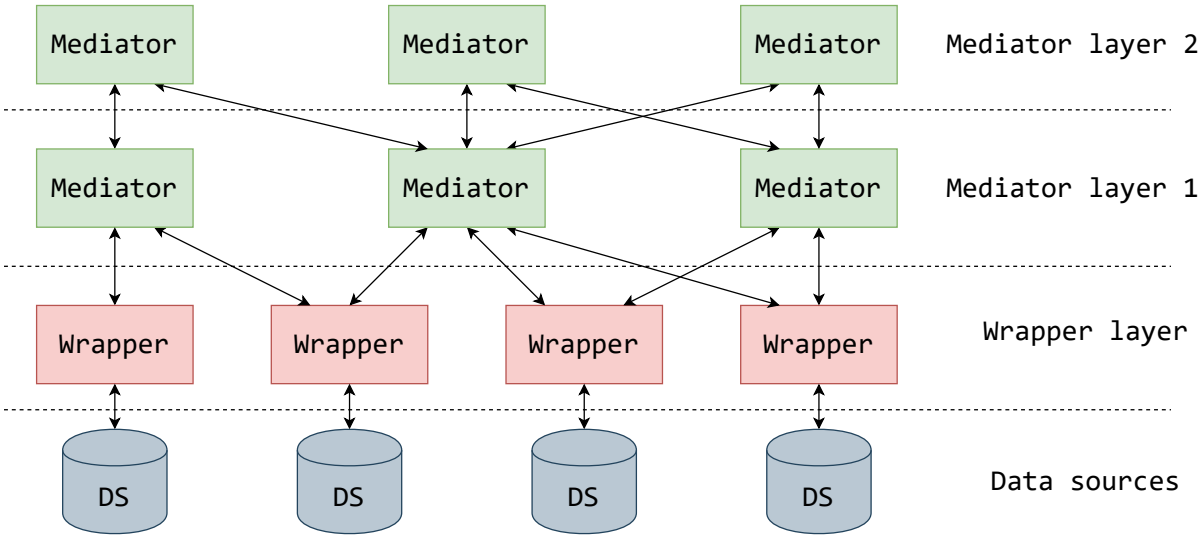


Figure 2.14: MW architecture with layered mediators [19]

The first layer of mediators is located directly above the wrapper layer. Figure 2.14 displays their relationship in a form where each mediator in this layer can be connected to multiple wrappers, and multiple wrappers can be connected to a single mediator. This is in line with the MW architecture displayed by Özsu and Valduriez [19]. Papakonstantinou et al. [4] and Jurczyk et al. [60] displayed an architecture in which each mediator of the first mediator layer is connected to just one wrapper and vice versa, showing that this is also a feasible solution in cases where mediators are only needed for transformation. The first mediator layer can be used to mediate between wrappers over paradigmatically similar data sources or data sources that have an overlapping or connected domain.

The second and upper layers of mediators can be used to raise the level of abstraction. The mediators of the upper layers are used to mediate between mediators of the lower layers, thus possibly encompassing multiple different data sources. Such a layering strategy is used by Moura et al.[61] in a form of special and central mediators to organize and distribute processing load, which can also be a beneficial effect if components are run on different machines. On the other hand, Chawathe et al. [9] used layering to enable localized logical management of data sources. This layering strategy was also proposed by Özsu and Valduriez [19].

Systems featuring a single monolithic mediator have been both proposed and implemented [5, 62, 63, 64, 65, 66]. This can be found to be an ample, quick, and expedient solution if the

number of connected data sources is not large or expected to rise. If the number of connected data sources rises, then the processing load on that single mediator is increased and this can easily lead to increased latency when querying any of the connected data sources through the integration system. These types of systems can be covariantly classified as multi-data source integration systems [1], taking note that the data source component translation is distributed (assigned to the wrappers). It should also be noted that in these cases, the mediator component is not really a component but rather a software module.

Recently, Sethi et al. [67] also focused on creating and maintaining a concern-oriented architecture system. Their workers each have a Data Source API in similarity to wrappers. The mediator's functionalities are assigned to a single coordinator acting as the query entry point and planner, and another worker to join the query results (akin to a reducer node in MapReduce).

2.3.1 On the roles of mediator–wrapper components

Considering the previously mentioned roles and interactions of wrappers and mediators in the MW architecture, it can be determined what kind of properties these components should have, and to which rules they should adhere. General conditions for well-formed components by Meyer [18] have already been presented in Section 2.1. Meyer's conditions can be expanded to determine what conditions a wrapper or mediator should specifically meet. For wrappers in a data source integration system, by following the example of [5, 8] (in their case the GARLIC system), the following rules (goals) are set:

RW1 The start-up cost to write a specific wrapper should be small. The wrapper itself can be constructed quickly with little need for prior knowledge of the data source integration system internal structure. There is a basic service upon which a specific wrapper is built upon.

RW2 Wrappers should be able to evolve. Incremental upgrades to the wrapper should be possible.

RW3 Wrappers should be modular and independent. Wrappers for new data sources can be integrated into the existing data source integration system without disturbing user applications, and other wrappers or components.

RW4 Wrappers should be participants in query planning. The wrapper may use whatever knowledge it has about a repository's query and specialized search facilities to dynamically determine how much of a query the repository is capable of handling.

The wrapper component type is succinctly defined in Definition 7 [14].

Definition 7. *The wrapper is a component that allows uniform access to a data source by wrapping the data source in terms of schema, queries, and data.*

For mediators in a data source integration system, following the ideas of Wiederhold [3],

the following rules are set:

RMe1 Structuring mediators into hierarchies should not lead to problems.

RMe2 Mediators should drive transformations. Mediators are there to accommodate the need for data and metadata restructuring. Queries are also affected by this restructuring.

The mediator component type is succinctly defined in Definition 8 [14].

Definition 8. *The mediator is a component used to manage transformations involving multiple unified schemas, the data they represent, and queries used to acquire the data.*

2.3.2 On schema hierarchies in the mediator–wrapper architecture

One of the advantages of using the MW architecture is the ability to modularly translate schemas by using the architecture's components themselves. To better understand these specifics, a generic example of a schema-type hierarchy is displayed in Figure 2.15, which shows all the possible schema types and their possible relationships. This is, in multiple forms, explained by [19].

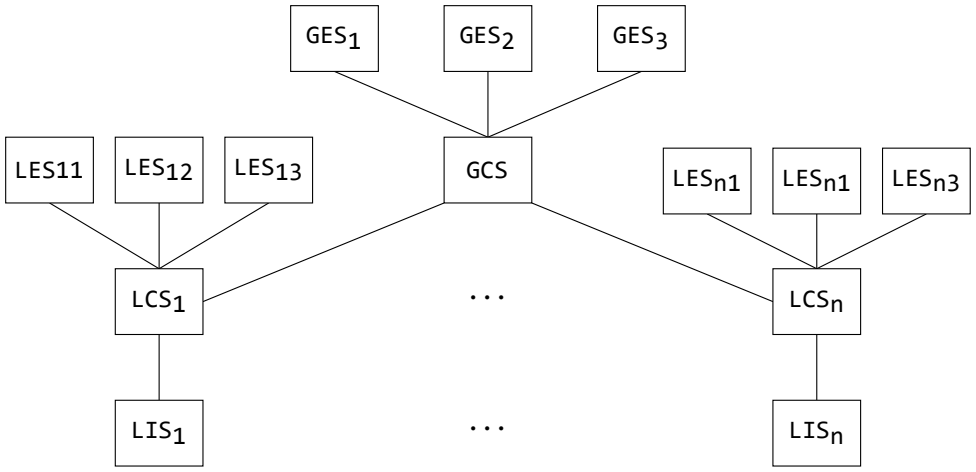


Figure 2.15: A schema hierarchy [19].

Starting bottom-up in Figure 2.15, the first type of schema is a local internal schema (LIS). The LIS is the schema found in the connected data source itself, defined in the data source's native form. For the data source integration system to be able to work on the connected data source, it must translate the LIS to a more generic and adaptable form that is used system-wide - this is the local conceptual schema (LCS). The LCS can then be translated into a local exported schema (LES). The LES is, for all intents and purposes, a partial or transformed schematic view of the LCS. As the data source integration system does not use the LES for its internal functioning, the LES can be described in an entirely different form and presented to the user. The global conceptual schema (GCS) is created by integrating the local conceptual schemas. In turn, the GCS can also be exported to the user in multiple forms, just like the LES. Such an exported schema is called a global exported schema (GES).

In a multi-database integration system, these schemas are all found in the same integration component in the form of metadata and are generated by modules. On the other hand, in an MW architecture data source integration system, these schemas are worked on gradually. This is done through the system's wrapper and mediator layers, each layer creating a more encompassing global schema or creating new forms of exported schemas.

As an example, a certain system-wide schema hierarchy is presented in Figure 2.16 with the schemas' relationships in accordance with the former explanation. The schema relationships (mergers or extractions) are presented by connecting lines, while the arrows on top of the schemas demonstrate which of them can be accessed by a user.

To provide a tangible illustration, an example is provided to demonstrate the appropriate allocation of schemas to MW components. Setting this example for the continuation of this paper, the schemas from the exemplified hierarchy (Figure 2.16) can now be assigned to MW architecture components. In Figures 2.17 and 2.18, the components are displayed with their assigned schemas (illustrated by a white rectangle) adjacent to them.

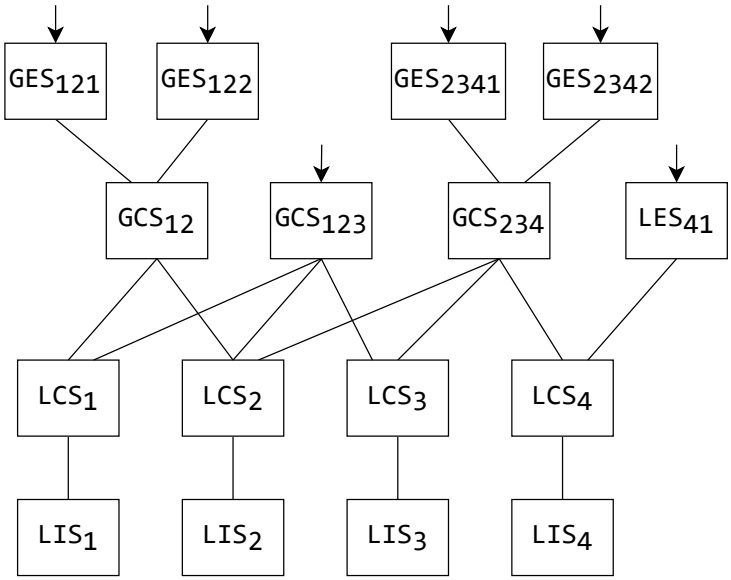


Figure 2.16: An example of a system-wide schema hierarchy.

The first possible assignment of schemas is to a system with a single mediator layer, as displayed in Figure 2.17. The LISs are positioned in the connected data sources. The wrappers then form their individual LCSs based on their connected data source's LIS. The wrappers' LCSs are then used by the mediators to create their GCSs. In this example, an aforementioned case of a mediator connected to a single wrapper is also displayed. This mediator generates an LES, and thus this mediator is only used for translation. The other mediators, along with their GCSs, generate GESs. A mediator can be used to create GESs to remove the need for another architectural layer of the mediators above. Of course, this might decrease system latency but will increase the complexity of mediator components, as they now must manage multiple user

role access.

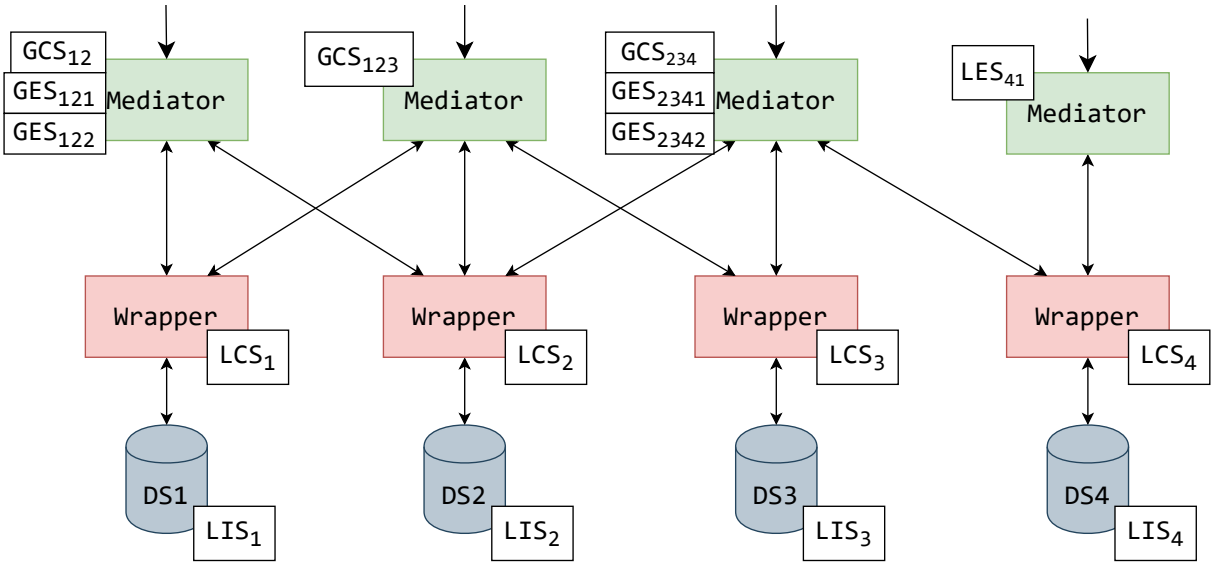


Figure 2.17: An exemplified assignment of schemas to an MW system with a single mediator layer.

Another example of schemas' assignments is displayed in Figure 2.18. In this example, there is another mediator layer on top of the architectural hierarchy. These mediators are used exclusively for exporting schemas, similar to the translating mediator. In this alternative, each mediator exports just one form of GES, thus reducing their task base and reducing the required complexity for user role management. In other words, each mediator could have just one form of a data-accessing user.

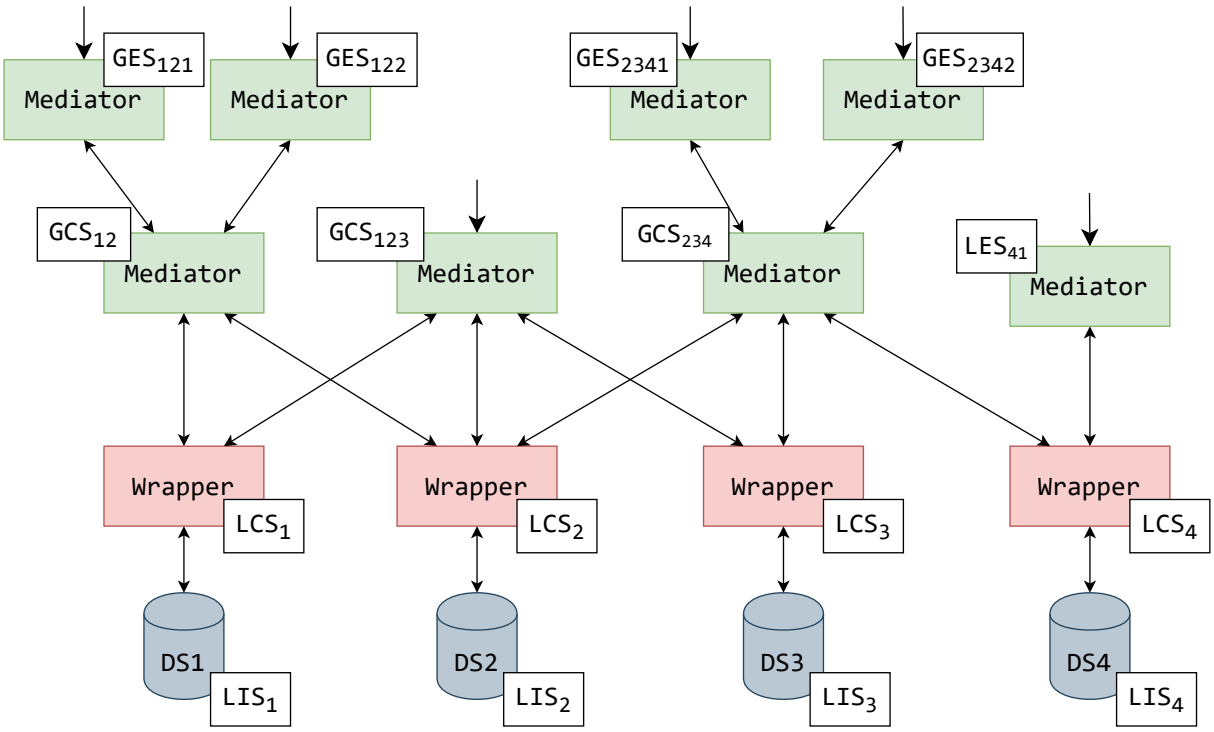


Figure 2.18: An exemplified assignment of schemas to an MW system with an exporting mediator layer.

2.4 Data mesh

The data mesh is a decentralized data management architecture that converges the ideas of [54]:

- distributed domain-driven architecture;
- self-serve platform design;
- product thinking with data.

It was proposed by Dehghani [20, 54, 68] as an alternative to the centralized data platform approach. The centralized data platform approach centrally manages and serves data through coupled ETL processes (Figure 2.19). The centralized data platform requires technologically specialized teams to support and develop ETL processes to serve data for analytical systems. Dehghani proposed a shift of technology-oriented data platforms towards those centred around domains and bounded contexts. This proposal brings the ideas of domain-driven design, as presented by Evans [49], from operational systems into the field of data management and analytical systems.

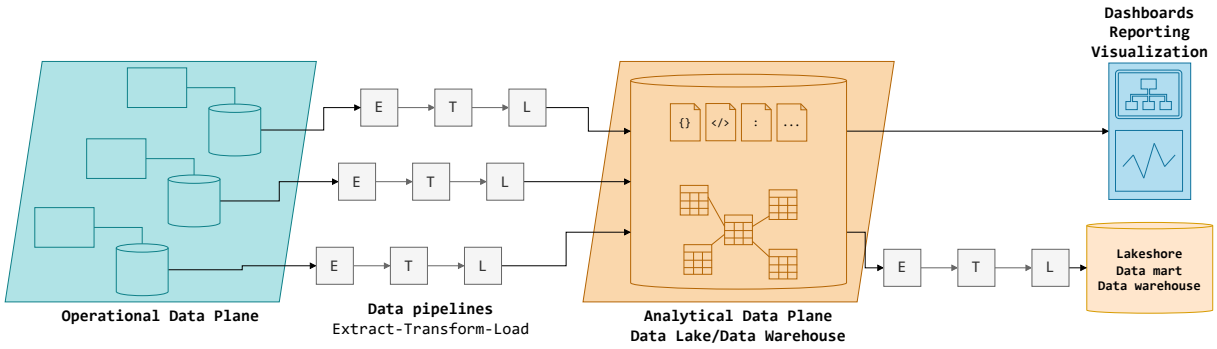


Figure 2.19: Illustration of centralized data management architectures

In a data mesh, an organization's data is arranged into bounded contexts. This paradigm leverages Conway's law [69], which postulates that *organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations*. By federating data in such a way, teams can be separated along the lines of domains (bounded contexts), not technologies or functionalities. Such teams acquire domain-specific knowledge faster and handle data and new tasks with greater expertise [20]. There is a trade-off inferred by adopting the data mesh, as postulated by Richards and Ford [15] in terms of architectures, which requires teams to be cross-functional. Concretely, instead of delegating the task of constructing a data pipeline to a specialized data technician team, the domain teams are tasked with building it for themselves. Paired with Conway's law, the data mesh is not only an architecture but also an organizational template.

Because teams in a data mesh are specialized, they can deliver a greater focus on the quality of data their domains provide. This enables teams to think of the data they provide in their domains as products, hence the name *data products*. The primary data source location for the

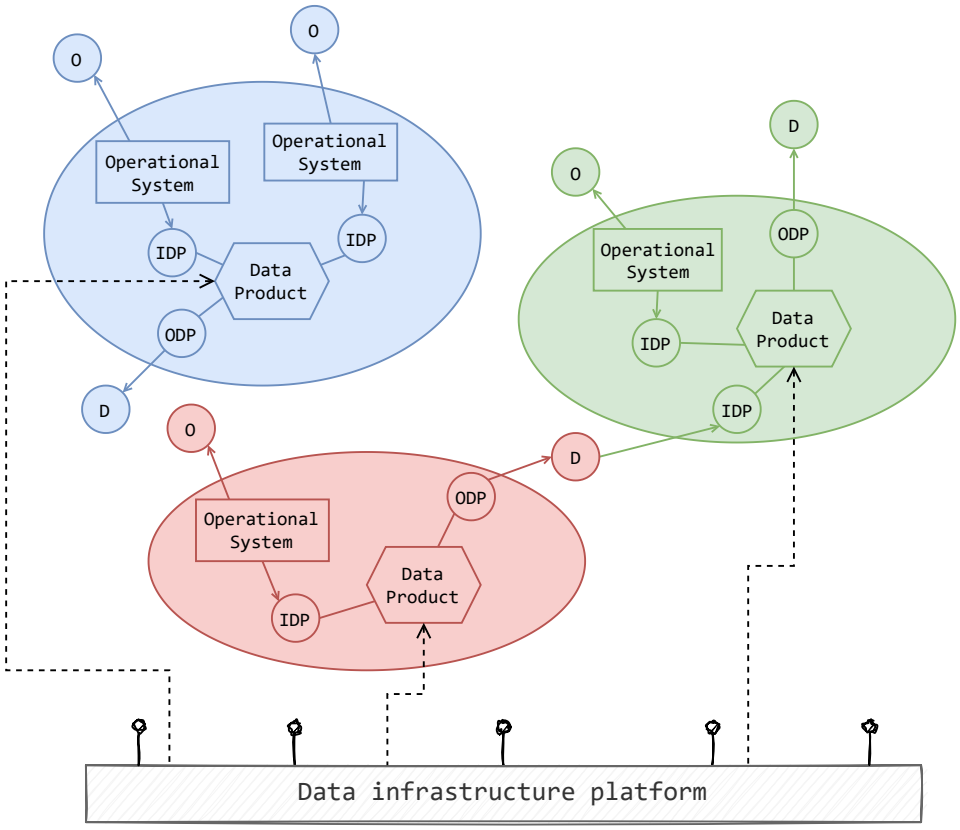


Figure 2.20: Multiple domains serving domain-oriented data (O - operational data; D - analytical data; IDP - input data port; ODP - output data port)

data mesh is the *data infrastructure platform* (DIP) [20]. The DIP contains (or refers to) existing data warehouses, data lakes or operational data stores. Operational systems tied to domains also contribute to the data product; the idea is that operational systems are also designed as bounded contexts. Data products can also be created by acquiring data from other domains through their own data products, which guarantees data quality from both the providing and receiving domains' end. A domain data product shouldn't acquire data from other domains' operational systems, because this would circumvent the data quality guarantee. Data products are consumed by users (for reporting, visualisation, dashboards etc.), operational systems, other analytical systems, or other data products. A hypothetical data mesh is illustrated in Figure 2.20.

Data products are conceptually illustrated as having input and output data ports, as in Figure 2.20. This usual imaging, provided by Dehghani [20, 54, 68], remarkedly evokes the ports-and-adapters pattern of the hexagonal architecture by Cockburn [70]. In the hexagonal architecture, the domain implementation is completely decoupled from driving and driven actors. This is done by prescribing a set of behaviours that the domain will recognize or enact (e.g. interfaces) called ports, and a set of adapters for those ports as a means to interact with actors (e.g. interface implementations). Reflecting ports and adapters back onto data products, they should be independent of the technological mechanisms by which they are created. To provide the data product with the aforementioned properties, an architectural component must be constructed

accordingly. Data product provisioning is accomplished by the *data product container* (DPC; Figure 2.21), which is an architectural quantum [20]. This infers that the data mesh itself has evolvable characteristics and is an evolutionary architecture.

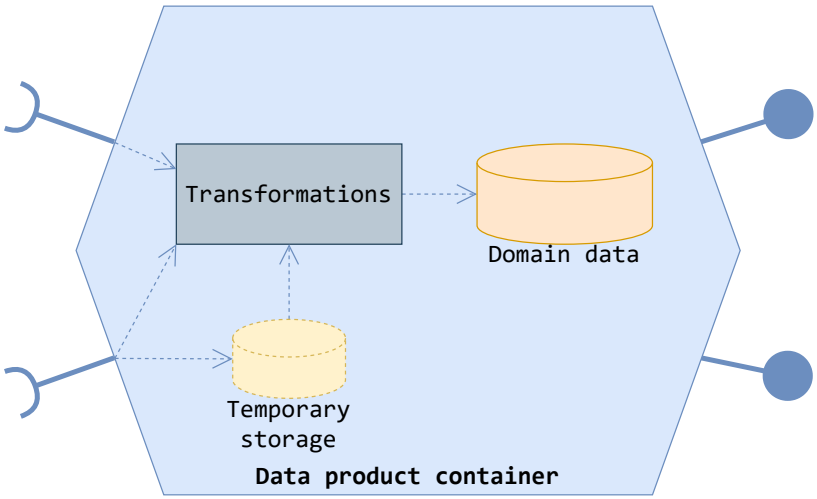


Figure 2.21: A data product container having a local domain data storage - data prepared for serving (adapted from [20])

The DPC has to consume data from the input ports, transform the data in adherence to global quality standards [20], and offer a means of serving the data as a ready product. A more distilled view of the DPC is that might not be implemented to handle data as a stream and might be required to keep data in temporary storage before being transformed into a viable product [28]. Therefore, the data needn't be served as a stream, so a data product store (domain data in Figure 2.21) can be used to store the data product.

The data mesh is proposed by Dehghani [54] to enable:

•Scalable polyglot big data storage;
•Encryption for data at rest and in motion;
•Data product versioning;
•Data product schema;
•Data product de-identification;
•Unified data access control and logging;
•Data pipeline implementation and orchestration;
•Data product discovery, catalogue registration and publishing;
•Data governance and standardization;
•Data product lineage;
•Data product monitoring/alerting/log;
•Data product quality metrics (collection and sharing);
•In memory data caching;
•Federated identity management;

•Compute and data locality.

A point of particular importance for this thesis is the statement by Dehghani [68] on what kind of data could the data mesh be tasked to serve: *Depending on the nature of the domain data and its consumption models, data can be served as events, batch files, relational tables, graphs, etc., while maintaining the same semantic*.

2.5 Quantitative shift-cost analysis

Richards [15] used a 5-level grading system to evaluate characteristics and compare the covered architecture styles. Despite the analysis results being quantified it is by its nature a qualitative analysis. This evaluation is sufficient for a fundamental architecture overview and to remark that software architectures are often differentiated by their trade-offs, but qualitative analysis results are difficult to clearly compare. Conversely, quantitative analysis results are clearly comparable if they were acquired from the same experiment.

Mens and Eden introduced a way to quantitatively measure and analyse the flexibility of software in terms of requirements changes in their two linked papers [21, 22]. This analysis will be referred to as a *quantitative shift-cost analysis*. Their definition of flexibility is assumed from IEEE's standard glossary of software engineering terminology (Definition 9) [71]. This same definition of flexibility is accepted in this thesis.

Definition 9. *Flexibility is the ease with which a system or a component can be modified for use in applications or environments other than those for which it was specifically designed.*

Mens and Eden [21, 22] observed a generic environment where arising requirements shift the problems within a specific domain. Requirement *shifts* necessitate changes in the existing software implementation called *adjustments*. Each pairing of a shift and adjustment is considered an evolution step (not directly tied with the terms evolvability and evolutionary from Section 2.1). This is illustrated in Figure 2.22.

The aforementioned terms are defined as follows [21, 22]:

Definition 10. *A requirement $\boldsymbol{r}$ is a well-defined specification of the program's expected behaviour, expressed in terms of the problem domain $\mathscr{D}$.*

Definition 11. *An implementation $\boldsymbol{i}$ is the subject of the evolution effort.*

Definition 12. *A shift is a specific change to a given set of requirements. A shift is represented as a function $\sigma_\delta(r,D,d)$, where $D \in \mathscr{D}$ and δ is an adjustment operation over D which changes the element $d \in D$.*

Definition 13. *An adjustment is a specific change to a given implementation i creating i', represented as $\alpha = (i,i')$.*

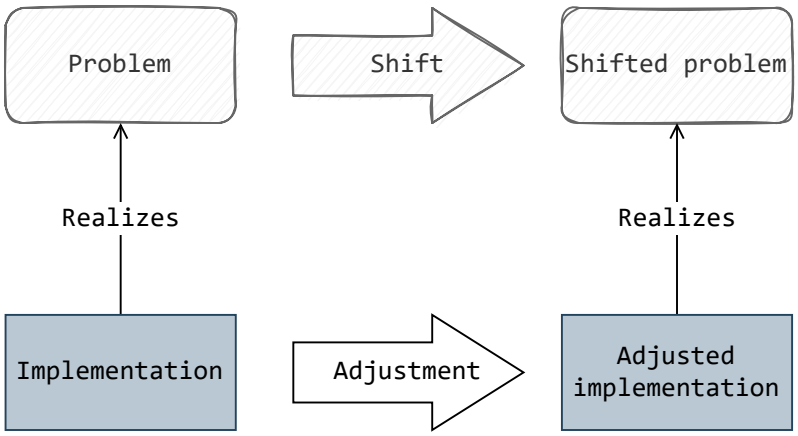


Figure 2.22: Conceptual illustration of an evolution step

Definition 14. *An evolution step is a pair of a shift and adjustment* $\varepsilon = (\sigma, \alpha)$.

While shifts are regarded in terms of requirements, adjustments on an implementation can be observed with different levels of granularity regarding the implementation's elements. Mens and Eden [21, 22] generically define this element as a *module*. This module is not necessarily the module from Section 2.1 by definition, but a theoretical placeholder for implementation elements like methods, functions, structures, classes, etc. A module in this terminology can represent a concrete code module. This generality allows the cost of an evolution step to be parametrized according to the implementation elements of interest. Hence, this analysis can be used for evaluating design patterns and, as will be shown in Section 4.3, architectures.

Definition 15. *The evolution cost metric is calculable as:*

$$\mathscr{C}_{module}(\varepsilon) = |modules(i) \,\Delta\, modules(i')| \qquad \text{(SHIFTCOST)}$$

where Δ *is a symmetric set difference such that* $A \Delta B = (A \setminus B) \cup (B \setminus A)$*, and* $modules(i)$ *are all modules in an implementation i.*

To simplify the statements of Definition 15, the evolution cost is the sum of the number of all modules added or removed during an adjustment effected by a shift.

Example 1 exemplifies the process of acquiring an evolution cost of a class hierarchy and is provided to demonstrate the nature and potential of this analysis.

Example 1. An object-oriented environment is observed, where the domain consists of an interface `IGameObject` denoting an object in a hypothetical two-dimensional game, and implementing classes `Rectangle`, `Circle`, `Triangle`, and `Square` representing two-dimensional objects in the game. The domain diagram is presented by Figure 2.23. Evaluations of evolution costs are made for two separate shifts on methods and classes as modules.

Shift scenario 1: addition of a `rotateCW` method to the `IGameObject` interface

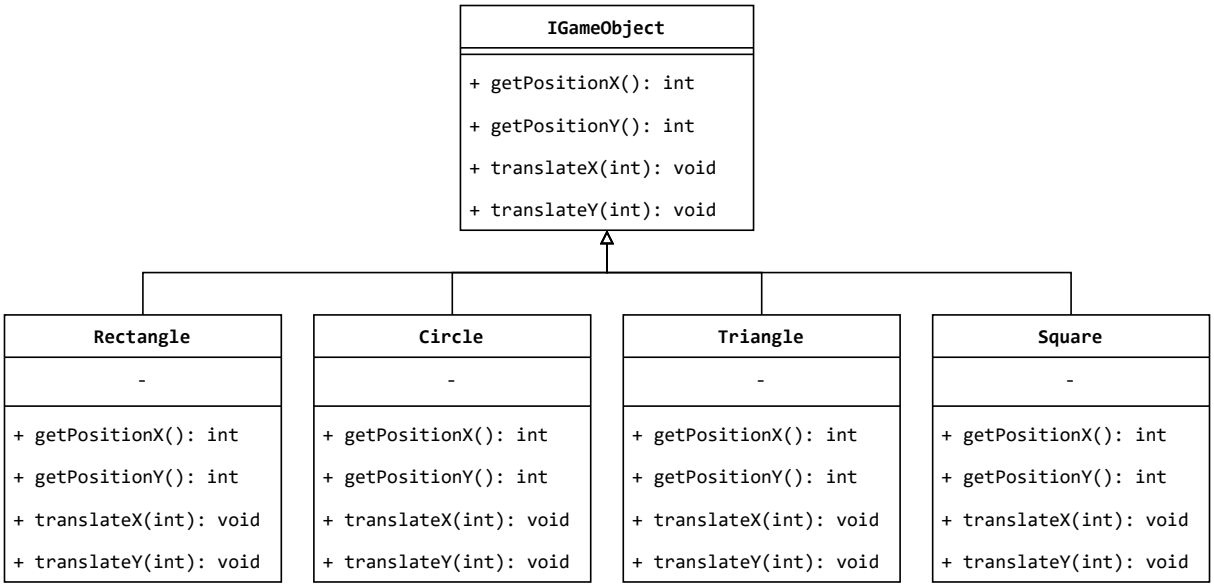


Figure 2.23: Game object domain diagram

The addition of a method to an interface necessitates implementing the method in the implementation classes (Figure 2.24). The `rotateCW` method must be implemented in four classes, so the method-moduled evolution cost is 4. This can be concretely specified as the following calculation:

$\alpha = (i_{before}, i_{after})$

$d = methods$

$methods(i_{before}) = \{\texttt{Rectangle.getPositionX}, ..., \texttt{Square.translateY}\}$

$methods(i_{after}) = \{\texttt{Rectangle.getPositionX}, ..., \texttt{Square.rotateCW}\}$

$methods(i_{before}) \, \Delta \, methods(i_{after}) =$

$\{\texttt{Rectangle.rotateCW}, \texttt{Circle.rotateCW}, \texttt{Triangle.rotateCW}, \texttt{Square.rotateCW}\}$

$\mathscr{C}_{method}(\varepsilon) = |methods(i_{before}) \, \Delta \, methods(i_{after})| = 4$

Since there were no adjustments made to classes, it follows that:

$\mathscr{C}_{class}(\varepsilon) = |classes(i_{before}) \, \Delta \, classes(i_{after})| = 0$

It can be concluded that in the general case of adding methods to the interface, the evolution cost on classes will always be 0, but the evolution cost on methods will depend on the number of implementation classes. It can be stated that for this type of shift:

$\mathscr{C}_{method}(\varepsilon) = |classes(i_{before/after})|$;

$\mathscr{C}_{class}(\varepsilon) = 0$.

Mens and Eden [21, 22] represent this as being akin to big-O notation.

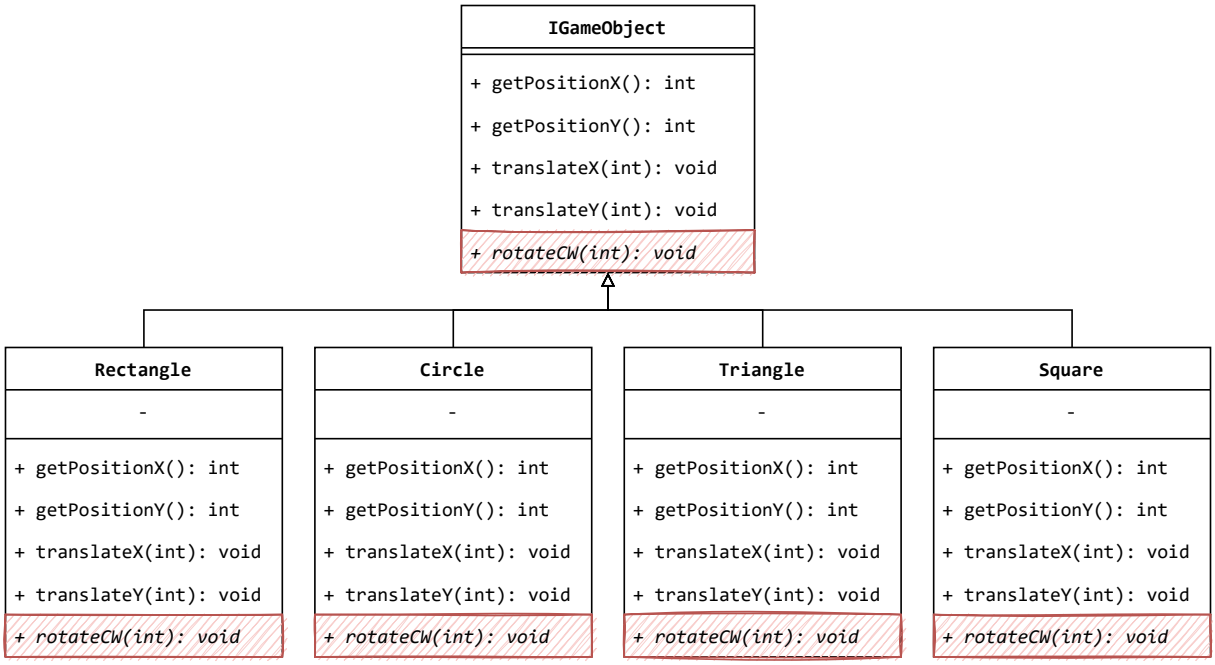


Figure 2.24: Addition of the `rotateCW` method

Shift scenario 2: addition of a `Rhombus` class implementing the `IGameObject` interface

The addition of an implementation class necessitates implementing the interface's methods in the implementing class (Figure 2.25). The interface contains four methods, so the evolution cost in terms of methods is 4. The evolution cost in terms of classes is trivially 1. This can be concretely specified as the following calculation:

$\alpha = (i_{before}, i_{after})$

$d = methods$

$methods(i_{before}) = \{\texttt{Rectangle.getPositionX}, ..., \texttt{Square.translateY}\}$

$methods(i_{after}) = \{\texttt{Rectangle.getPositionX}, ..., \texttt{Rhombus.translateY}\}$

$methods(i_{before})\ \Delta\ methods(i_{after}) =$

$\{\texttt{Rhombus.getPositionX}, \texttt{Rhombus.getPositionY}, \texttt{Rhombus.translateX}, \texttt{Rhombus.translateY}\}$

$\mathscr{C}_{method}(\varepsilon) = |methods(i_{before})\ \Delta\ methods(i_{after})| = 4$

Adjustments made to classes involve adding one class, so it follows that:

$\mathscr{C}_{class}(\varepsilon) = |classes(i_{before})\ \Delta\ classes(i_{after})| = 1$

In a general case of adding implementation classes to the interface, the evolution cost on classes will always be 1, but the evolution cost on methods will depend on the number of methods the interface has. It can be stated that for this type of shift:

$\mathscr{C}_{method}(\varepsilon) = |methods(i_{IGameObject})| = constant;$

$\mathscr{C}_{class}(\varepsilon) = 1.$

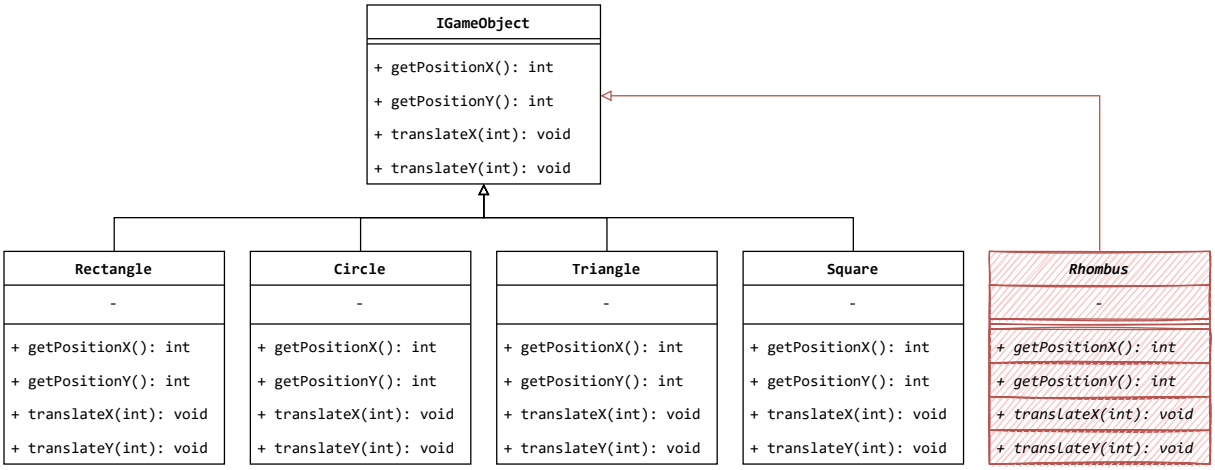


Figure 2.25: Addition of the `Rhombus` class

It can be concluded that with a growing number of implementation classes, the domain is more flexible to implementation class additions than to interface method additions.

In the remainder of this thesis, the evolution cost is referred to as the *shift cost*, as costs are expressed over shifts. This is to prevent the mixing of concepts regarding evolvability and to keep in line with the contributing journal article [14]. The nomenclature is also adjusted later on to simplify the expressions made while running a quantitative analysis, but the core concept is kept.

Chapter 3

Category theory, functional programming, and bidirectionalisation preliminaries

This chapter is provided to introduce the minimal preliminary information required for the understanding of concepts tied to category theory and functional programming. First and foremost, the concept of lenses is a construct from functional programming that has its basis in category theory. The Janus system's design, which is presented later (see Chapter 6), relies heavily on functional programming and category theory. To illustrate concepts tied to category theory applied in functional programming, examples in Haskell-like code snippets will be provided.

3.1 Basic category theory

Category theory is a branch of pure mathematics that stems from the field of algebraic topology [72]. Category theory has greatly influenced computer science and has been used extensively in programming language design and implementational techniques, software design and implementation, and type theory. The most basic term in category theory is that of the *category*.

Definition 16. *A* **category C** *is comprised of:*

1.a collection of ***objects****;*

2.a collection of arrows called ***morphism****;*

3.operations assigning each arrow f an object $dom f$, its domain, and an object $cod f$, its codomain (we write $f : A \to B$ to signify that $dom f = A$ and $cod f = B$);

4.a ***composition operator*** *assigning to each pair of arrows f and g with $cod f = dom g$, a composite arrow $g \circ f$ satisfying the following associativity law:*

for any arrows $f : A \to B$, $g : B \to C$, and $h : C \to D$ (with A, B, C, and D not necessarily distinct),

$$h \circ (g \circ f) = (h \circ g) \circ f;$$

5.for each object A, an identity arrow $id_A : A \to A$ *satisfying the following identity law:*
for any arrow $f : A \to B$,
$id_B \circ f = f$ *and* $f \circ id_A = f$.

Constructions in category theory can be graphically represented by a commutative diagram, which is a directed graph where:

•vertices are objects;

•morphisms are directed edges such that for $f : A \to B$ there is an edge pointing from vertex A to B.

A commutative diagram is shown for category C from Definition 16 in Figure 3.1, where a dashed directed edge is added to represent the composition of morphisms f, g, and h.

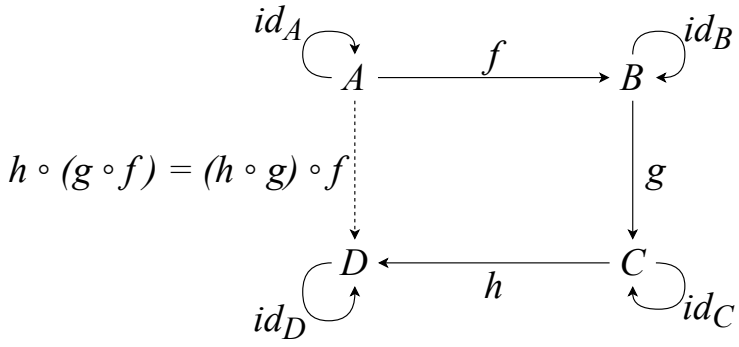


Figure 3.1: A commutative diagram representing category *C* from Definition 16

Each possible composition of morphisms also implies the existence of an equivalent morphism by substitution [73]. By the example of Figure 3.2, morphisms f and g can compose, since $cod\, f = dom\, g$, hence there exists a morphism that signifies $g \circ f$ - the h morphism. Identity morphism compositions are usually omitted.

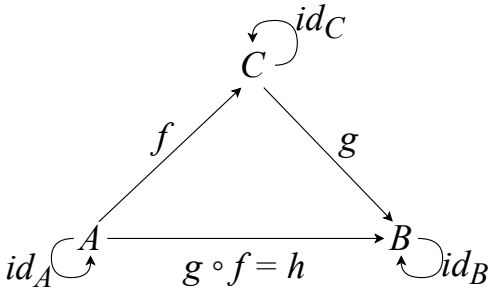


Figure 3.2: Morphism composition example

The essence of category theory is to try and omit the contents of a category's objects as much as possible, so they can be reasoned about in terms of their morphisms. Morphisms can be affected by the nature of the objects they pair. The abstract nature of Definition 16 allows the replacement of a category's objects and morphisms with other corresponding mathematical and logical constructs. To show a simplified example, authors [72, 73] usually present a case

where objects can represent sets and morphisms can represent total functions between sets. This creates a category called *Set*. This *Set* category can represent a category of sets of numbers, similarly presented as *NumSet* in Example 2.

Monomorphism, epimorphism, isomorphism

Morphisms can be distinguished by their properties, especially if they are considered mappings of quantifiable objects. In category theory, these properties are reasoned about without looking into the objects the morphisms map, but by taking into account the relations between the morphisms themselves.

Definition 17. *A morphism $f : B \to C$ in a category C is a **monomorphism** if, for any pair of C-arrows $g : A \to B$ and $h : A \to B$, the equality $f \circ g = f \circ h$ implies that $g = h$.*

Definition 18. *A morphism $f : A \to B$ is an **epimorphism** if, for any pair of arrows $g : B \to C$ and $h : B \to C$, the equality $g \circ f = h \circ f$ implies that $g = h$.*

Definition 19. *A morphism $f : A \to B$ is an **isomorphism** if there is an arrow $f^{-1} : B \to A$, called the inverse of f, such that $f^{-1} \circ f = id_A$ and $f \circ f^{-1} = id_B$. The objects A and B are said to be **isomorphic** if there is an isomorphism between them.*

Definition 20. *Two objects that are isomorphic are often said to be identical **up to isomorphism** or **within an isomorphism**.*

There is a correspondence between the terms monomorphism, epimorphism, and isomorphism when discussing categories with set-like objects. Monomorphism corresponds to injectivity, epimorphism corresponds to surjectivity, and isomorphism corresponds to bijectivity. Example 2 illustrates this by approaching a category of number sets.

Example 2. A *NumSet* category contains objects (Figure 3.3):

- $\mathbb{N}_0$ - a set of natural numbers including *zero*;
- $\mathbb{Z}$ - a set of integers;

and total functions as morphisms:

- $id_{\mathbb{N}_0}$, such that $id_{\mathbb{N}_0}(x) = x$;
- $id_{\mathbb{Z}}$, such that $id_{\mathbb{Z}}(x) = x$;
- $f : \mathbb{N}_0 \to \mathbb{Z}$, such that $f(x) = -x$;
- $g : \mathbb{Z} \to \mathbb{N}_0$, such that $g(x) = |x|$.

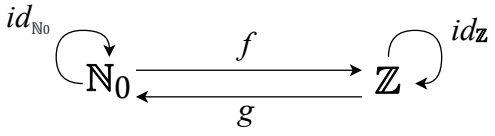


Figure 3.3: *NumSet* category diagram

f maps all elements of $\mathbb{N}_0$ into $\mathbb{Z}$ in such a way that $f(x_1) = f(x_2)$ implies $x_1 = x_2$, making it a injective function. f is non-surjective as its codomain is not the entirety of $\mathbb{Z}$. In terms of category theory, f is a monomorphism. g is a non-injective function on $\mathbb{Z}$, as can be proven by example of $g(-1) = 1$ and $g(1) = 1$. On the other hand, g is a surjective function as its codomain covers the entirety of $\mathbb{N}_0$. This makes g an epimorphism. The composition $g \circ f$ is bijective, therefore it is correspondingly an isomorphism.

Terminal and initial objects

Objects in category theory can be classified by forming *universal constructions* - objects and accompanying morphisms that share a common property[72]. The simplest of these are the initial and terminal objects, as introduced by Definitions 21 and 22 [72] (Figure 3.4).

Definition 21. *An object 0 is called an **initial object** if, for every object X, there is exactly one arrow from 0 to X.*

Definition 22. *An object 1 is called a **terminal object** if, for every object X, there is exactly one arrow from X to 1.*

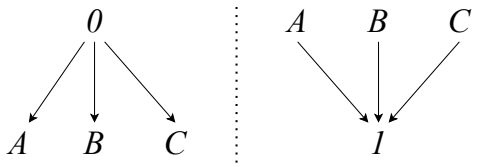


Figure 3.4: Diagrams of an initial and terminal object

In terms of the *Set* category, an initial object is the singular empty set $\{\}$ of the category. For every set S in *Set*, there is a unique empty function from $\{\}$ to S. A terminal object is any singleton set $\{x\}$ of *Set* where $x \in S$, since for every set S there is a function from S to a singleton set $\{x\}$.

Products

A product is a universal construction in category theory (Figure 3.5). It corresponds to the Cartesian products in set theory. A product is constructed by recognizing that products have projection functions; for a product $A \times B$ there exist projection functions $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$. There is also an initial set, which must contain two functions which connect it to the projected product sets: $f : C \to A$ and $g : C \to B$. To virtually construct a product, the functions f and g are formed into a product function $\langle f, g \rangle$, such that $\langle f, g \rangle : C \to A \times B$ and $\langle f, g \rangle(x) = (f(x), g(x))$. These functions can be recovered by setting $f = \pi_1 \circ \langle f, g \rangle$ and $g = \pi_2 \circ \langle f, g \rangle$.

Definition 23. *A product of two objects A and B is an object $A \times B$, together with two projection morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$, such that for any object C and pair of arrows $f : C \to A$ and $g : C \to B$ there is exactly one mediating arrow $\langle f, g \rangle : C \to A \times B$, with $f = \pi_1 \circ \langle f, g \rangle$ and $g = \pi_2 \circ \langle f, g \rangle$ making the diagram commute [72].*

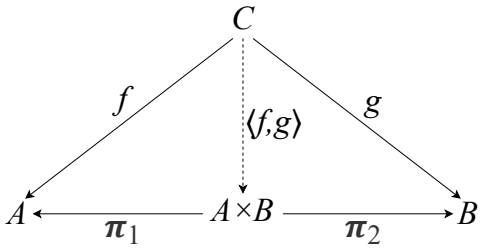


Figure 3.5: Commutative diagram of a product

Coproducts

Due to the notion of duality, that for every category C there exists an opposite category C^{op} in which morphisms are reversed, it is possible to construct an opposite construction to the product - the coproduct (Figure 3.6). The coproduct corresponds to the discriminate (disjoint) union in set theory.

Definition 24. *A coproduct of two objects A and B is an object $A + B$, together with two injection arrows $\iota_1 : A \to A + B$ and $\iota_2 : B \to A + B$, such that for any object C and pair of arrows $f : A \to C$ and $g : B \to C$ there is exactly one arrow $[f, g] : A + B \to C$ making the diagram commute [72].*

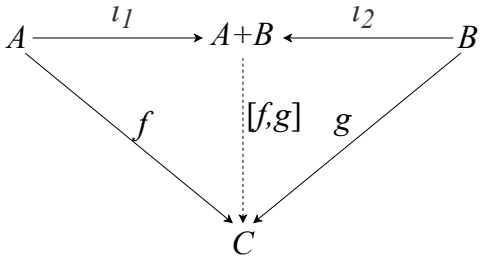


Figure 3.6: Commutative diagram of a coproduct

Relationship to types and computer programs

Objects and morphisms in category theory can be used to denote various ideas. The given examples over vague sets or sets of natural numbers and integers can be transposed onto a more concrete environment of computer programs. A category can be observed where objects are types, and morphisms are total functions. This is exemplified by a simple type category in Example 3 adapted from Pierce [72].

Example 3. A category *ST* has the following objects representing types:

Int - integer values;

Real - real number values (discrete);

Bool - boolean values;

Unit - type with just one possible value;

Void - empty type;

The *ST* category has the following morphisms representing total functions between types:

$true : Unit \to Bool$;

$false : Unit \to Bool$;

$zero : Unit \to Int$;

$not : Bool \to Bool$;

$succInt : Int \to Int$;

$succReal : Real \to Real$;

$isZero : Int \to Bool$;

$toReal : Int \to Real$;

$\forall \alpha \in (Int \cup Real \cup Bool \cup Unit)\ unit : \alpha \to Unit$

The *ST* category's diagram is shown in Figure 3.7. The *ST* category can be assessed in terms

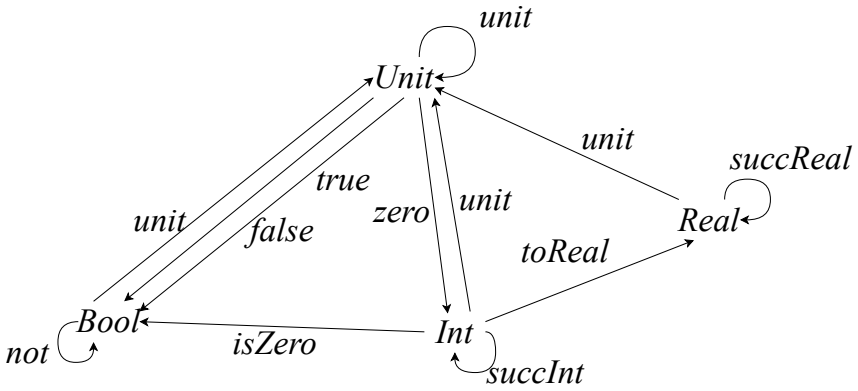


Figure 3.7: Diagram of the *ST* category

of a computer program. The category is equipped with a means to create constants via functions: $true$, $false$, and $zero$. *Real* is not covered by constant creation, whose constant can be created by an implied $toReal \circ zero$ composition. *toReal* and *isZero* represent a function mapping integers to real and boolean values respectively. The category contains a terminal object *Unit*, which is accessed from all types via their respective *unit* functions. The *ST* category does contain a *Void* as an initial object, but the example omits it further for the sake of clarity. In a type system such as this, no actual constants could be created if morphisms were declared to stem from *Void* because such a function couldn't have any *Void* value passed to it. Many type systems, like in Haskell, substitute this by having constant-creating functions stem from *Unit*. This doesn't prevent the *Unit* from being a valid terminal object [73]. Haskell does additionally

provide a polymorphic function from *Void* to all types called *absurd*, but this function can never be invoked. It is important to note that morphisms *not*, *succInt*, and *succReal* aren't identity functions. On the other hand, $unit : Unit \rightarrow Unit$ is an identity function. These formulations make the hypothetical computer program declarative.

3.2 Functional programming

The declarative nature of a computer program doesn't impact its ability to be run on a genuine computer. A subset of declarative programming is functional programming (alt. functional paradigm). Tying in functional programming with category theory, data types are objects and pure total functions are morphisms. Pure functions are those functions that have the following properties [73]:

•repeatedly return the same result with the same set of arguments;

•have no side effects (no mutation of static variables, references or input/output streams).

This implies that a pure function is idempotent and stateless in regard to its surrounding program environment. However, these properties don't limit programming capabilities in functional programming. Side effects are handled by a mechanism introduced in Section 3.2.2. Exclusive use of pure functions allows the use of the substitution model to run proofs on programs, where each expression can be substituted by a sub-expression or concrete value as a result of expression evaluation [74].

Functional programming has been implemented in many programming languages, with Haskell being popular in the programming language, category theory, and type theory community. As mentioned at the beginning of Chapter 3, Haskell-like notation will be used for illustrative examples of category theory in software design. Type notation in Haskell relies on the *Curry-Howard* (alt. *arrow*) notation. This notation is adopted from lambda-calculi and implies a vital notion of functions in functional programming - all functions have just one parameter. This much is already evident in set theory. E.g. a function $f : (A \times B) \rightarrow C$ doesn't individually map elements of A and B into C, but takes as an argument a single tuple of elements from A and B denoted as $(A \times B)$. This is not evident in formulaic mathematical expressions for functions, e.g. $f(x,y) = x + y$. In the Curry-Howard notation, such a function would be annotated as $Int \rightarrow Int \rightarrow Int$. The Curry-Howard arrow is an operator of a type constructor [73] that indicates the types of parameters the function accepts and the type of its return value. The arrow operator is left-associative, so the last argument is the return type of the function. This notation also allows the usage of brackets. A function defined in this manner is called a *curried* function.

Example 4. A function *add* is defined in over an *Int* type as follows:
$add : (Int \times Int \times Int) \rightarrow Int$ such that $add(x,y,z) = x + y + z$

In Curry-Howard notation, *add* is defined as follows:
$add : Int \to Int \to Int$, with a definition: $add(x) = \lambda y.\lambda z.x + y + z$.
This can be translated into Haskell as:

```
--curriedfuntiontosum3integers
add::Integer->Integer->Integer->Integer
add x =
    \y ->
        \z -> x + y + z
--calladdandassertresult
main =
    letresult = add 3 7 11
    in
    print$assert (== 21) (result)
```

Listing 3.1: Explicitly curried function *add* in Haskell

```
--non-curriedfunctiontosum3integers
add:: (Integer,Integer,Integer) ->Integer
add (x, y, z) = x + y + z
--calladdandassertresult
main =
    letresult = add (3, 7, 11)
    in
    print$assert (== 21) (result)
```

Listing 3.2: Non-curried function *add* in Haskell

It is important to note that Haskell implicitly supports currying, but Listing 3.1 in Example 4 explicitly curries the `add` function to illustrate the case of curried functions.

In lambda-calculi, as is consequently recognized in functional programming, a function receives individual arguments by returning a new function that receives the successive argument. In functional programming, each argument is said to be *applied* to the function. This is the basis for partial application. The consequence of this mechanic is the ability to treat functions as objects in object-oriented programming. Hence, partially applied functions can be referenced with variables. In a bare type system, akin to the one provided by Scheme/Lisp, this enables the construction of more complex data types. The most notable of these examples was given by Abelson et al. [74], showing how a tuple data type can be constructed by the use of captured variables, closures, and partial application. A closure is a data structure containing a lambda expression [75]. A captured variable is a variable declared outside of the closure's scope, but referenced inside the closure. In Listing 3.3 a similar example to the one by Abelson et al. [74] is given; a tuple of strings with projection functions $first$ and $second$. This implementation is akin to the definition of a categorical product from Section 3.1.

```
--declareafunctiontypetoaliasastringtuple
typeStringTuple =Integer-> [Char]
--constructsastringtupleasalambda
consStringTuple:: [Char] -> [Char] -> StringTuple
consStringTuple x y =
    \idx ->ifidx == 0thenxelsey
--returnsthefirstelementofatuple
first:: StringTuple -> [Char]
first t = t 0
--returnsthesecondelementofatuple
second:: StringTuple -> [Char]
second t = t 1
--createastringtuple
tuple = consStringTuple"42""JohnDoe"
--mainfunction
main =
    letfirstElement = first tuple
        secondElement = second tuple
    indo
    print$assert (=="42") firstElement
    print$assert (=="JohnDoe") secondElement
```

Listing 3.3: Construction of a tuple by using lambda functions, closures, and partial application

Functional programming allows the composition of functions; this is analogous to the composition of morphisms in category theory. Consequently, this enables the construction of more complex functions from simpler functions. The composition of morphisms in a functional programming environment is exemplified in 5

Example 5. For a collection of functions:

$zero : Unit \to Int$, where $zero(x) = 0$

$addOne : Int \to Int$, where $addOne(x) = x + 1$

$divTwo : Int \to Int$, where $divTwo(x) = x/2$

$mulThree : Int \to Int$, where $mulThree(x) = x * 3$

$$gtThree : Int \to Bool, \text{ where } gtThree(x) = \begin{cases} true, & x > 3 \\ false, & \text{otherwise} \end{cases}$$

A composite function can be defined by:

$composite : Unit \to Bool$,

where $composite(x) = gtThree \circ mulThree \circ divTwo \circ addOne \circ zero(x)$.

This can be defined and declared in terms of functional programming as:

```
--createsa0integer
zero:: () ->Integer
```

```
zero_ = 0
--addsonetoanumber
addOne::Integer->Integer
addOne x = x + 1
--dividesnumberbytwo
divTwo::Integer->Integer
divTwo x = x `div` 2
--multipliesnumberbythree
mulThree::Integer->Integer
mulThree x = x * 3
--determinesifnumberisgreaterthan3
gtThree::Integer->Bool
gtThree x = x > 3
--composefunctionswiththe`.`operator
composed:: () ->Bool
composed = gtThree . mulThree . divTwo . addOne .zero
--runandtest
main =print$assert (not) (composed ())
```

Listing 3.4: Composition of functions

Example 5 also raises the point in functional programming that functions can be treated as objects. The `composed` function doesn't call the composition of the functions on its right-hand side, this is done in the `main` function. The `composed` function is just a reference to the composition of functions (a function itself), which is treated as an object of type `Unit -> Bool`. In functional programming, referenced objects (including value objects) should not be mutated in the program. Copying objects with the applied mutations is preferred as an alternative, so objects are considered immutable by default.

Currying functions enables arguments to be passed individually as parameters of functions. With each argument passed, a new function accepting the next parameter is returned. The returned function is equivalent to the underlying closure, with the first argument ending up as a captured variable in the closure. This piecemeal passing of arguments to parameters of functions is called partial application. Each argument application returns a function that uses it as a captured variable. Due to this, every function can be also considered a constructor for a family of functions. Example 6 illustrates the basic functioning of partial application.

Example 6. A collection of partially applied functions is given, where `sumThree` sums three given integers, `addToThree` adds two integers to the number 3 by applying it to `sumThree`, and `addToTen` adds an integer to the number 10 by partially applying the number 7 to `addToThree`.

```
---addthreeintegers
sumThree::Integer->Integer->Integer->Integer
sumThree x y z = x + y + z
```

```
--addtwointegersto3
addToThree::Integer->Integer->Integer
addToThree = sumThree 3--alsovalidaddToThreexy=sumThree3xy
--addintegerto10
addToTen::Integer->Integer
addToTen = addToThree 7--alsovalidaddToTenx=addToThree7x
--runandtest
main =
    letresult = 19
    indo
    print$assert (== result) (sumThree 3 7 9)
    print$assert (== result) (addToThree 7 9)
    print$assert (== result) (addToTen 9)
```

Listing 3.5: Collection of partially applied functions

Since functions are viewed as objects in functional programming, they inherit their basic nature. As with value objects, function objects can be passed as arguments to a function if a function is parametrized as such. A function that accepts a function as a parameter is called a high-order function. High-order functions further enable the management of abstraction in a program. A high-order function might define the general way in which a certain operation is to be executed, but the operation itself is not detailed. When the operation function is provided, the high-order function constructs a concrete function. This allows the construction of a strategy pattern similar to the one in object-oriented programming [49, 76] where the high-order function acts as an interface or abstract class. This mechanism is presented in Example 7.

Example 7. A high-order function `createStartingScreen` is used to parametrize the printing of a starting screen for a hypothetical command shell by a welcome message that names the user. Functions `welcomeEng`, `welcomeCro`, and `welcomeDeu` create welcome messages in their respective languages depending on the user name they are given. Functions `createStartingScreenCro`, `createStartingScreenEng`, and `createStartingScreenDeu` are concretizations of `createStartingScreen`.

```
--createsenglishwelcome
welcomeEng:: [Char] -> [Char]
welcomeEng name ="Welcome,"++ name ++"!"
--createscroatianwelcome
welcomeCro:: [Char] -> [Char]
welcomeCro name ="Dobrodosli,"++ name ++"!"
--createsgermanwelcome
welcomeDeu:: [Char] -> [Char]
welcomeDeu name ="Willkommen,"++ name ++"!"
--startingscreenfunction
```

```
createStaringScreen:: ([Char] -> [Char]) -> [Char] -> [Char]
createStaringScreen welcomeFun userName =
    "***************"++"\n"++
    (welcomeFun userName) ++"\n"++
    "~/HomeDir>"
--croatianwelcomescreen
createStartingScreenCro:: [Char] -> [Char]
createStartingScreenCro userName =
    createStaringScreen welcomeCro userName
--englishwelcomescreen
createStartingScreenEng:: [Char] -> [Char]
createStartingScreenEng userName =
    createStaringScreen welcomeEng userName
--germanwelcomescreen
createStartingScreenDeu:: [Char] -> [Char]
createStartingScreenDeu userName =
    createStaringScreen welcomeDeu userName
--runandprint
main::IO()
main =do
    putStrLn$createStartingScreenCro"John"
    putStrLn$createStartingScreenEng"John"
    putStrLn$createStartingScreenDeu"John"
```

Listing 3.6: High-order functions to print a welcome screen

3.2.1 Functors

Functors arise as a necessity to map structures between two categories, but in such a way that object-morphism structures are preserved. A definition of a functor according to Pierce [72] is given in Definition 25. To preserve structures in other categories, functors must guarantee that they preserve identity morphisms and compositions of morphisms.

Definition 25. *Let C and D be categories. A* ***functor*** $F : C \to D$ *is a map taking each C-object A to a D-object F(A) and each C-morphism* $f : A \to B$ *to a D-morphism* $F(f) : F(A) \to F(B)$*, such that for all C-objects A and composable C-morphisms* f *and* g

- $F(id_a) = id_{F(A)}$;
- $F(g \circ f) = F(g) \circ F(f)$.

In functional programming, functors are used to control transformations over data. This can be practical when data has to abide by certain rules. Example 8 is derived from a similar age-type example made by Buonanno [77], where data regarding a person's age has to respect certain limits.

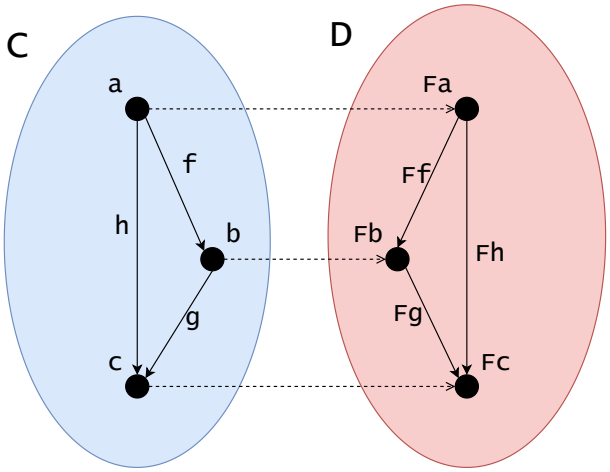


Figure 3.8: Conceptual diagram of a functor

Example 8. The `Age` functor limits age values to $[0, 99]$. To modify the data inside the `Age` type, only the `ageMap` function can be used. The `ageMap` function calls the given argument method and then calls the appropriate method to create an age in a valid range. The argument method `f` may return a value from an invalid range, but the subsequent call of `makeAge` sets the value in the valid range.

```
--Agefunctortype
dataAge = AgeInt
--EqdefinitionforAge
instanceEqAgewhere
  (Age a) == (Age b) = a == b
--functorreturnfunction
makeAge ::Int-> Age
makeAge n = Age (max0 (min99 n))
--showimplementation
instanceShowAgewhere
  show(Age n) =shown
--functormapforage
ageMap:: (Int->Int) -> Age -> Age
ageMap f (Age n) = makeAge(f n)
--runandtest
main =
        --createanAgevaluewithavalueof102(really99)
    letmadeAge = makeAge 102
        --add5yearstotheage
        incAge = ageMap (+ 5) madeAge
        --remove105yearsfromtheage
        decAge = ageMap (subtract105) madeAge
        --addanother18years
        finalAge = ageMap (+ 18) decAge
    indo
```

```
26    print$assert (== Age 99) madeAge
27    print$assert (== Age 99) incAge
28    print$assert (== Age 0) decAge
29    print$assert (== Age 18) finalAge
```

Listing 3.7: Primitive `Age` functor

According to Buonanno [77], a functor `F` must have the following functions available over its type (kind):

- `map: F a -> (a -> b) -> F b`;
- `return: a -> F a`.

Buonanno [77] states that a functor doesn't need a `return` function, because it is defined by the properties a `map` function should observe. The existence of `return` is just a technicality since the type `a` has to have a way of being lifted into the functor type `F a`.

It is interesting to note a subtle difference between a functor, as defined in category theory and one defined in a program. Functors in a programming environment are *endofunctors* (Figure 3.9); these map categories onto themselves, instead of mapping categories onto other categories. All functors defined in a program are endofunctors since they map types from the existing type system onto the very same type system; they can't map to types indescribable by the existing type system.

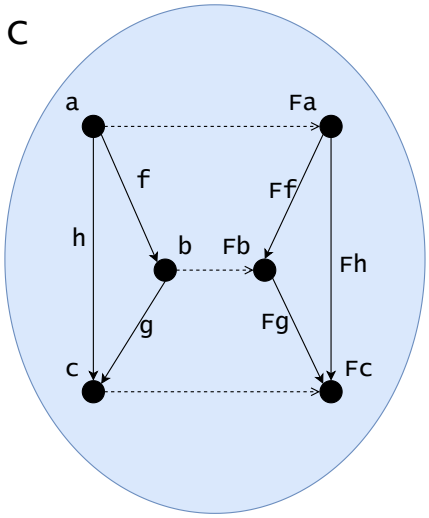
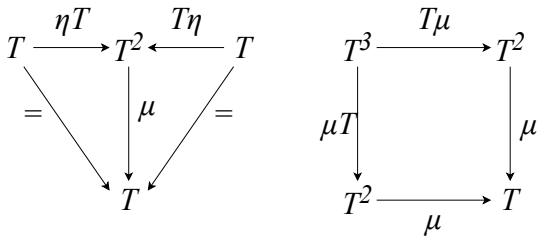


Figure 3.9: Conceptual diagram of an endofunctor

3.2.2 Monads

Monads are an important tool in functional programming used to manage side effects (exceptional behaviour in a program) or to box behaviour for types to certain rules. In Haskell, even printing data onto the standard output is managed by the `IO` monad. Definition 26 gives a formal definition of a monad in category theory according to Barr and Wells [78].

Definition 26. *A monad $M = (M, \eta, \mu)$ on a category C consists of a functor $M : C \to C$, together with two natural transformations $\eta : id \to M$ and $\mu : M^2 \to M$ for which the following diagrams commute.*



Buonanno [77] states that a monad `M` must contain the following functions over its type (kind):

- `bind: M a -> (a -> M b) -> M b;`
- `return: a -> M a.`

A monad must adhere to the *monad laws* [77, 78]:

- *right identity* - `return a >>= f ≡ f a;`
- *left identity* - `m >>= return ≡ m;`
- *associativity* - `(m >>= g) >>= h ≡ m >>= (\x -> g x >>= h).`

Where `x ≡ y` denotes that `x` is substitutable by `y`.

The `return` method, just like in the case of functors, lifts the enclosed type into a monad type. The `bind` function enables the management of side effects when calling functions that also produce a monad of the same kind. The `bind` function allows chained calling and composition of functions that return the same kind of monad (see Example 9); this is the basis of the railway pattern [77, 79]. All monads are by definition functors, and this can be emphasised by the fact that a `map` function can be expressed in terms of `bind` and `return` functions [77]:

```
1 map::MonadM=>M a -> ( a -> b ) -> M b
2 mapx f = bind x (return. f )
```

Listing 3.8: A sketch of `map` in terms of `bind` and `return`

A more notable monad type, which is universally used in many languages, is `Option`*. `Option` is used to encase data in such a way that it indicates whether there is *some* data or *none*. This enables the elimination of nullable types from use in codespaces that require explicitly defined behaviour [79], and primitively managing outcomes of operations.

Example 9. The monadic `Option a` type is defined as a discriminate union of the `Some a` type and `None`. `Some` indicates the presence of data, and `None` indicates the absence of data. the `map` function relating to `Option` is defined in terms of `bind` and `return` functions. Option-returning

*Also named `Maybe` or `Optional` in other languages and environments. The name `Option` is used to stay in accordance with the name of the analogous type used in the Janus system in Section 6.1.

functions are chained by the bind operator >>=, which is in Haskell an infix redeclaration of `bind`. The code below demonstrates the `Option` monad working over an integer type:

```
--shouldinclude:importPreludehiding((>>=),map,return)
--Optiondefinition
dataOption a = Some a | None
--EqinstanceforOption
instanceEqa=>Eq(Option a)where
    Some x == Some y = x == y
    None == None     =True
    _    == _        =False
--returnfunction
return:: a -> Option a
returnx = Some x
--bindfunction
bind:: Option a -> (a -> Option b) -> Option b
bind (Some a) f = f a
bind None f = None
--bindoperator
(>>=):: Option a -> (a -> Option b) -> Option b
(>>=) = bind
--showinstance
instanceShowa=>Show(Option a)where
    show(Some x) =showx
    showNone ="None"
--mapintermsofbindandreturn
map:: Option a -> (a -> b) -> Option b
mapx f = bind x (return. f)
--incrementsvalue
inc::Int-> OptionInt
inc x = Some (x + 1)
--decrementsvalue
dec::Int-> OptionInt
dec x = Some (x - 1)
--returnsNone
retNone::Int-> OptionInt
retNone _ = None
--runandtest
main =
  letsimpleMap =map(return4) (+ 1)
      bindResult1 = (return4) >>= inc >>= dec >>= inc >>= inc
      bindResult2 = (return4) >>= inc >>= dec >>= retNone >>= inc
  indo
  print$assert (== Some 5) simpleMap
  print$assert (== Some 6) bindResult1
```

```
43  print$assert (None) bindResult2
```

Listing 3.9: Primitive `Option a` monad

3.3 Bidirectionalisation

Bidirectionalisation (or bidirectional transformations; *BX*) is a collection of approaches for creating two-way data transformations resulting in consistent data. BX observes the relationship between forward and backward transformations on data. Such transformations should ideally be inverses and enable round-trip transformation of data. The roots of BX can be found in Bancilhon and Spyrators [80], who proposed a translator-defining method for determining updates on an underlying relational database from updates on relational views. Updateable views are now a common feature in modern relational database management systems. BX has also been proposed for use in data model management and synchronization [81, 82, 83, 84], data source integration [85, 86], triple graph grammars [87], and syntax parsing with a possibility of program transpilation [88].

Three primary BX approaches stand out, as recognized by Foster et al. [89]: *syntactic BX*, *semantic BX*, and *BX combinators*. Syntactic BX observes the function definition of the forward transformations and infers a backward transformation function. Semantic BX infers a backward transformation from a forward transformation without inspecting it, but by observing the changes made to the data. BX combinators allow the description of the forward and backward transformation simultaneously by combining lenses, usually via domain-specific languages. These approaches are complementary and can be utilised together, as in the case presented by Voigtländer et al. [90]

All BX approaches recognize the pairing of a *get* and *put* function as the standard basis for two-way invertible transformations. The *get* function represents a forward transformation, and the *put* function represents a backward transformation. The definition of this pairing of functions, named *lens*, is given in Definition 27 in the most general terms [23, 89].

Definition 27. *For a set of source structures S and a set of view structures V, the **get** and **put** functions*

$$get \in S \to V; \tag{GET}$$

$$put \in V \to S \to S; \tag{PUT}$$

*form a **well-behaved lens** such that the following laws apply for $s \in S$ and $v \in V$:*

$$get\ (put\ v\ s) = v; \tag{PUTGET}$$

$$put\ (get\ s)\ s = s \tag{GETPUT}$$

The PutGet and GetPut laws guarantee basic round-tripping of the data transformations. PutGet proscribes that the *put* function should fully reflect any changes made to the view [89], and GetPut stipulates that the *put* function must not change the source if the view is not modified at all.

The general environment discussed in BX literature can be visualized as in Figure 3.10, where a source structure is transformed into a view structure, which is updated, and finally returned to the source structure. This chain of operations results in a perceived update of the source structure (a fictitious transformation $source \rightarrow source'$).

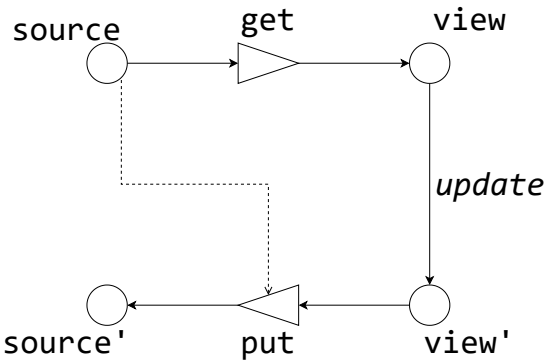


Figure 3.10: Representation of the view-update problem

There is a higher level of strictness for lenses above being *well-behaved*, allowing them to be sounder [23, 89] in terms of their *put* functions not creating side-effects. Definition 28 implies that chaining the *put* function should only lead to changes in the source only if the view had changed at some point.

Definition 28. *A well-behaved lens that obeys the following law*

$$put\ v'\ (put\ v\ s) = put\ v'\ s \qquad \text{(PUTPUT)}$$

is considered a ***very well-behaved lens***

If the GetPut and PutPut both hold for a lens, then updates to the view can always be undone, since the following equation is implied:

$$put\ (get\ s)(put\ v\ s) = s \qquad (3.1)$$

As a consequence, any update to the source can be undone by simply recalling the *put* function with the non-updated view.

Example 10. Two functions, `head` and `tail`, are used in `get` and `put` functions of a hypothetical lens. The following code exemplifies the case when PutGet, GetPut, and PutPut laws hold.

```
--returnsthefirstelementofalist
head:: [a] -> a
head(x:xs) = x
--omitsthefirstelementofalist
tail:: [a] -> [a]
tail(x:xs) = xs
--getfunction
get:: [a] -> a
get = Main.head
--putfunction
put:: a -> [a] -> [a]
put view source = [view] ++ (Main.tailsource)
--mainfunction
main =
    letsource = ['a', 'b', 'c', 'd', 'e']
        view = 'a'
        updatedView = 'x'
        updatedSource = ['x', 'b', 'c', 'd', 'e']
    indo
    --PUTGETlaw
    print$assert (== updatedView) (get(put updatedView source))
    --GETPUTlaw
    print$assert (== source) (put (get source) source)
    --PUTPUTlaw
    print$assert (== updatedSource) (put updatedView (put view
    ↪ source))
```

Listing 3.10: A `get` and `put` pair implemented with list `tail` and `head`

The laws in Example 10 wouldn't hold if function signatures were changed so that `head: [a] -> [a]` and `tail: [a] -> [a] -> [a]`, since the Haskell list type allows empty lists. Additionally, these laws wouldn't hold even if elements were added to the view list. Haskell doesn't enable empty characters (e.g. `''`) or storing multiple characters in a single character variable, so an item can't be deleted from or added to the view. The `get` and `put` functions given in this example represent a very well-behaved lens, although a very limited one. The precise problem which prevents the well-behavedness of the lens is related to the change of the shape of the view on updating. To mitigate these problems Voigtländer [23] proposes type specializations (see Section 3.3.2), while Matsuda [24] proposes a more general solution of weakening the PutGet and PutPut. This weakening is presented in Definition 29.

Definition 29. *A get-put pair satisfying*

$$\frac{(put\ v\ s)\downarrow}{get\ (put\ v\ s) = v} \qquad \text{(PARTIAL-PUTGET)}$$

and

$$\frac{(put\ v\ s)\downarrow}{put\ v'\ put(v\ s) = put\ v'\ s} \qquad \text{(PARTIAL-PUTPUT)}$$

is called a ***partial*** *very well-behaved lens.*

The weakening conveys that the *put* function is not injective. Non-injectivity is enforced so a view is returned to the source in an expanded domain which also contains an error state. Listing 3.11 demonstrates such a modification. The *put* from Listing 3.11 is a simplified example and not a pure function, as it creates an exception. A pure weakened *put* can be implemented to return a monad [91, 92], with the monadic *bind* function serving as a composition operator. This implies that the *get* function also returns the monad.

```
1  --returnsthefirstelementofalist
2  head:: [a] -> [a]
3  head(x:xs) = [x]
4  --omitsthefirstelementofalist
5  tail:: [a] -> [a]
6  tail(x:xs) = xs
7  --getfunction
8  get:: [a] -> [a]
9  get = Main.head
10 --weakenedputfunction
11 put :: [a] -> [a] -> [a]
12 put view source =
13     letupdatedViewLen =lengthview
14         viewLen = (length. Main.head) source
15     in
16         ifupdatedViewLen == viewLen
17         thenview ++ (Main.tailsource)
18         elseerror"Wrongviewshapeafterupdate"
19 --mainfunction
20 main =
21     letsource = ['a', 'b', 'c', 'd', 'e']
22         view = ['a']
23         updatedView = ['x']
24         updatedSource = ['x', 'b', 'c', 'd', 'e']
25         illegalView = ['x', 'y']
26         illegalUpdatedSource = ['x', 'y', 'b', 'c', 'd', 'e']
27     indo
28     print$assert (== updatedView) (get(put updatedView source))--
   ↪ PUTGET
29     print$assert (== source) (put (get source) source)--GETPUT
30     print$assert (== updatedSource) (put updatedView (put view
   ↪ source))--PUTPUT
31     print$assert (== illegalUpdatedSource) (get(put illegalView
```

```
    ↪ source))--raiseserror
```

Listing 3.11: Code changes for a lens with a weakened *put*

Complements

In the `put` function of Listing 3.11, the expression `view ++ (Main.tail source)` implied that an "opposite" function to `head` is known. It is important to note that this is not an inverse function, but a *complement* function to `head` (and reflexively `get`). Such a function might not always be implicitly known. A complement function is a function that preserves the data lost in the forward transformation (Definition 30) [89]. The complement function is also referred to as a *residual function* and the complement as a *residue*, hence the abbreviated naming *res* in Definition 30. The complement created by the complement function remains untouched by the *put* function; this is the reason why authors usually specify the complement as *constant-complement* [93].

Definition 30. *Let $get \in S \to V$ be a total function from S to V. A total function $res \in S \to C$ computes a complement for get if and only if the tupled function $\langle get, res \rangle \in S \to (V, C)$ is injective.*

Changes required to support a non-computed complement in Listing 3.11 is given in Listing 3.12.

```
1  --complementfunction
2  res:: [a] -> [a]
3  res = Main.tail
4  --weakenedputfunction
5  put :: [a] -> [a] -> [a]
6  put view source =--canalsobesubstitutedas:putviewcompl
7      letupdatedViewLen =lengthview
8          viewLen = (length. Main.head) source
9      in
10         ifupdatedViewLen == viewLen
11         thenview ++ (Main.res source)
12         elseerror"Wrongviewshapeafterupdate"
```

Listing 3.12: Code changes for a lens with a complement function

Complements allow further mechanisms for bidirectionalisation. Given a forward transformation and a computable complement, a tupled function of the two can be inverted to obtain a very well-behaved lens [80]. Abou-Saleh et al. [93] state that a very well-behaved lens even induces an isomorphism $S \cong V \times C$. This is shown in Defintion 31 by Foster et al. [89].

Definition 31. *Let $get \in S \to V$ be a forward transformation function and let $res \in S \to C$ be a complement function for it. The function $put_{(get,res)}$ defined by*

$$put_{(get,res)}\ v\ s = inv(v,\ res\ s), \text{where } inv = \langle get, set\rangle^{-1} \tag{UPD}$$

is a suitable backward transformation function. When combined with a get, it yields a very well-behaved lens.

Foster et al. [89] present two cases for the *inv* definition:

1. when the $\langle get, res\rangle$ function is injective and surjective, *inv* is its (full) inverse $\forall s, v, c.s \in S, v \in V, c \in C$:

$$inv(\langle get, res\rangle\ s) = s \tag{LEFTINV}$$

$$\langle get, res\rangle\ (inv\ (v,c)) = (v,c) \tag{RIGHTINV}$$

2. when the $\langle get, res\rangle$ function is not surjective, *inv* is a left inverse for it but only a partial right-inverse $\forall s, v, c.s \in S, v \in V, c \in C$:

$$\frac{(inv\ (v,c))\downarrow}{\langle get, res\rangle\ (inv\ (v,c)) = (v,c)} \tag{PARTIAL-RIGHTINV}$$

Create

BX can further be enriched by the addition of the *create* function. The *create* function is used to create a source structure from a view structure, where authors [94, 95, 96] usually express the view as an abstract representation of structures due to the idea being expressed in terms of language design. The *create* function is a special case of *put* where no former source is provided, and default values are used to fill in the missing data that was eliminated by *get*. The *create* function is defined in Defintion 32 [94, 95, 96].

Definition 32. *$\forall s, v.s \in S, v \in V$ the **create** function is such that: $create : V \to S$, where non-existing values of s in v are substituted by default values of their type set, and such that the following rule applies on the lens containing create:*

$$get\ (create\ v) = v \tag{CREATEGET}$$

3.3.1 Syntactic BX

Syntactic BX is an approach by which backward transformation functions are derived from a forward transformation by analyzing the program definition (syntax) of the forward transformation. Matsuda et al. [24], who first proposed the approach, base syntactic BX on the compositionally of primitive view functions (or primitive *get* functions) via several combinators. These

combinators are familiar constructs from functional programming languages:

•composition: $g \circ f$ defined by: $(g \circ f) = g(f\ x)$;

•mapping: $map\ f$ defined by: $map\ f\ [x_1,...,x_n] = [f\ x_1,...,f\ x_n]$;

•product: $f \times g$ defined by: $(f \times g)(x,y) = (f\ x, g\ y)$;

•conditional: if p then f else g defined by:

$$(\text{if } p \text{ then } f \text{ else } g)\ x = \begin{cases} f\ x, & \text{if } p\ x \\ g\ x, & \text{otherwise} \end{cases}.$$

This collection of combinators is based on the work of Foster et al. [95], which has shown that if primitive backward transformations can be prepared for primitive view functions, then complex backward transformations for complex forward transformations can be created from the primitive backward transformations. Matsuda et al. [24] further proposed that backward transformations can be derived automatically with this approach. The syntactic BX approach is implemented in three steps:

1. *Derviation of a complement function*. For a given *get* function, a *res* function is syntactically derived.
2. *Tupling and program inversion*. The *get* and *res* functions are tupled and inverted so they form a partial inverse $\langle get, res \rangle^{-1}$. The inverse satisfies LeftInv and Partial-RightInv.
3. *Construction of a backward transformation*. The partial inverse $\langle get, res \rangle^{-1}$, complement function *res*, and UPD are used to derive a backward transformation. The transformation can be further optimized by deforestation [97] or partial evaluation.

The syntactic BX approach enables the use of a standard functional language to implement a two-way transformation. The downside is that the expressiveness of the language is limited to primitive view functions.

3.3.2 Semantic BX

Semantic BX approach is based on deriving backward transformations by analyzing the effects of forward transformations. Voigtländer [23] proposed this approach by basing it on the notion of free theorems [98]. The backward transformation can be derived from a forward transformation through a high-order function. The high-order bff (name abbreviated from *bidirectionalisation for free*) function tracks the differences between the source and view structures made by *get* and accordingly constructs a *put* function reflecting those differences. This approach relies on the premise that the *get* function is polymorphic.

In the most general terms, the bff function is declared as:

$$bff : (S \to V) \to (V \to S \to S) \tag{BFF}$$

; where the first parameter (in parenthesis) represents the targeted *get* function, with the return

type corresponding to a *put* function.

Voigtländer [23] discusses the levels of correctness that this approach produces by iteratively increasing the specification of the S and V types, starting from a simple list structure that has no type limitation (kind):

$$bff : (\forall a.[a] \to [a]) \to (\forall a.[a] \to [a] \to [a]) \tag{3.2}$$

Consequently, the PutGet and GetPut laws are representable as:

- $bff\ get\ (get\ s)\ s \equiv s$ and
- $get\ (bff\ v\ (get\ s)) \equiv v$,

respectively.

The bff definition is susceptible to data duplication on updates in the view, so an equality check is required. The kind of the parameters is then specified to adhere to an equality operator:

$$bff_{Eq} : (\mathbf{Eq}\ a.[a] \to [a]) \to (\mathbf{Eq}\ a.[a] \to [a] \to [a]) \tag{3.3}$$

This is demonstrated to create a partial *put*, noticeable when originally equal values in a view are updated to different values [23]. The parameter kind can be specified to adhere to orderable data, and the type can be expanded to hold orderable indicators of elements in the list. This produces the high-order bff_{Ord} function:

$$bff_{Ord} : (\mathbf{Ord}\ a.[a] \to [a]) \to (\mathbf{Ord}\ a.[a] \to [a] \to [a]) \tag{3.4}$$

These principles can be transposed to more advanced data structures adhering to the `Zippable`, `Traversable`, and `Foldable` kinds. An implementation by Voigtlander [23] demonstrates bff working over the recursive data structure of a tree:

```
data Tree a = Node (Tree a) (Tree a) | Leaf a.
```

The complement is derived inside the bff function with an enclosed set of expressions covered by the *inv* function:

$$inv : \forall a\ .\ \mathbf{Eq}\ a \Rightarrow ([a], (Int, IntMap\ a)) \to [a] \tag{3.5}$$

The semantic BX approach hypothetically enables the derivation of backward functions from already compiled forward functions, because it doesn't rely on a function's definition. The limitation of this is that the forward function must be polymorphic.

3.3.3 BX combinators

In contrast to the other BX approaches BX combinators do not automatically derive a *put* function from a *get* function. BX combinators take into account that a *get* function might have multiple *put* functions. The choice of the concrete *put* function is left to the user. This means that BX combinators already have a prepared set of *get* and *set* functions at the user's disposal, which, together with a developed type system, can be used to guarantee certain levels of behavedness. To assure this guarantee, BX combinators are built as separate programming languages. Boomerang is a general-purpose bidirectional language for processing textual data [94, 99]. BiGUL [100] is a core language serving as a foundation for higher-level languages. Asano et al. [85] proposed a BX language for decentralized data integration. The HOBiT language [101] was proposed to remove the combinator-based language limitation of using a point-free style of programming (where no function arguments can be explicitly identified). A more specifically purposed bidirectional language based on BX combinators is Augeas, which is used to globally edit Linux configuration files in a tree format [102, 103]. These languages rely on the work of Foster et al. [95, 104] who proposed the use of combinators over lenses to manage the forward and backward transformations. The combinators can be composed together to form more complex combinators with properties reflected by the combinators in their composition.

The BX combinator approach also acknowledges the use of complements, and a lens is recognized as containing the following functions:

$$get \in S \to V; \tag{3.6}$$

$$res \in S \to C; \tag{3.7}$$

$$put \in V \to C \to S. \tag{3.8}$$

Barbosa et al. [105] also note the *create* function:

$$create \in V \to S. \tag{3.9}$$

Foster et al. [89] additionally enrich the *put* function with a `Maybe` monad[†] (akin to `Option` in Example 3.9) to facilitate totality:

$$put \in V \to \texttt{Maybe}\, C \to S. \tag{3.10}$$

Modified round-tripping laws PutGet, CreateGet and GetPut are presented as [105]:

[†]The authors originally presented it with an uncurried type $put \in V \times \texttt{Maybe}\, C \to S$ to simplify some of their definitions.

$$get\ (put\ v\ c) = v; \tag{3.11}$$

$$get\ (create\ v) = v; \tag{3.12}$$

$$put\ (get\ s)\ (res\ s) = s; \tag{3.13}$$

including the `Maybe` monad [89]:

$$\frac{get\ s = v \quad res\ s = c}{put\ (v, \texttt{Just}\ c) = s}; \tag{3.14}$$

$$\frac{put\ (v, \texttt{m}\ c) = s}{get\ s = v}; \tag{3.15}$$

with a modified PutPut:

$$\frac{}{put\ (v', \texttt{Just}\ (res\ (put\ (v, \texttt{m}\ c)))) = put\ (v', \texttt{m}\ c)}. \tag{3.16}$$

The BX combinator approach also focuses on alignment problems. Alignment problems appear when the view data is modified with insertions, removals, or significant updates. Example 11 illustrates these problems through a transformation round-trip.

Example 11. Let source data `s` be:

```
s = [("Charlemagne", "Holy Roman Emperor"),
     ("Theodoric", "King of the Ostrogoths"),
     ("William I", "King of England"),
     ("Robert I", "King of Scotland")].
```

For a `get` that projects the first element of the tuple in the list over `s` the following `v` is generated:

```
v = ["Charlemagne", "Theodoric", "William I", "Robert I"];
```

together with a complement `c`:

```
c = ["Holy Roman Emperor",
     "King of the Ostrogoths",
     "King of England",
     "King of Scotland"].
```

The update of `v` involves the removal of `"Theodoric"`, the appending of `"Baldwin IV"`, and the update of `"William I"` to `"William the Conqueror"`, making the updated view `v'`:

```
v' = ["Charlemagne", "William the Conqueror", "Robert I", "Baldwin IV"].
```

Reflecting the update backwards onto the source `s'`, with setting the value for the second element to "Baldwin IV" to a default string `""`, a satisfactory result would be:

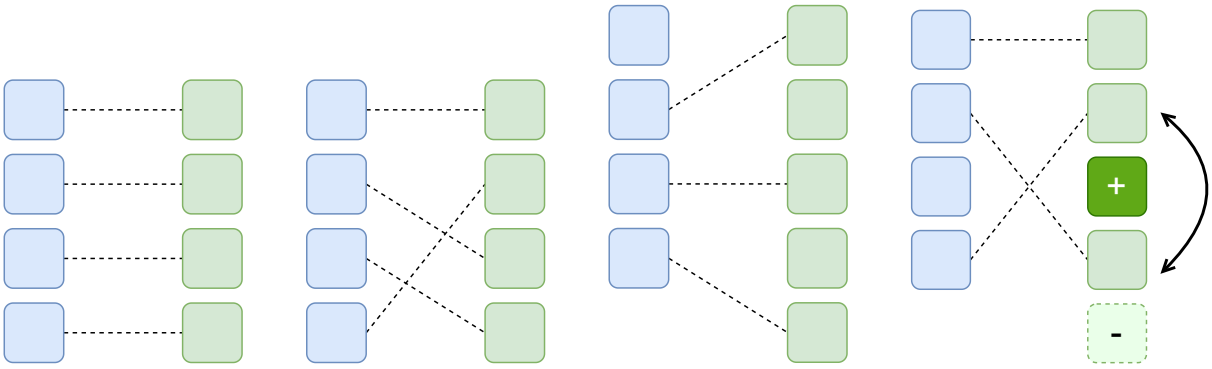
```
s' = [("Charlemagne", "Holy Roman Emperor"),
     ("William the Conqueror", "King of England"),
```

```
    ("Robert I", "King of Scotland"),
    ("Baldwin IV", "")].
```

Achieving such a `s'` is not trivial. A lens matching the values by starting position with the complement would produce a misaligned result:

```
s' = [("Charlemagne", "Holy Roman Emperor"),
    ("William the Conqueror", "King of the Ostrogoths"),
    ("Robert I", "King of England"),
    ("Baldwin IV", "King of Scotland")].
```

Alignment strategies can be used to remedy the alignment problems. Alignment strategies observe the source and view as reordable chunks [89] (items of a list in Example 11). This viewpoint allows complex data structures to be aligned at the generic level. Barbosa et al. [94, 105] recognized four alignment strategies as illustrated in Figure 3.11. The *positional* strategy compares the positions of the source and view chunks. The *best match (minimizing edit distance)* strategy minimizes the sum of total edit distances between pairs of matched chunks and the lengths of unmatched chunks [105]. The *best non-crossing* considers the minimizing edit distance but produces alignments that don't cross positions (edges). The *actual operations* strategy considers operations performed over the view and calculates the corresponding alignment.



(a) positional (b) best match (c) best non-crossing (d) actual operations

Figure 3.11: Alignment strategies

Combinators and lenses that use alignment strategies are called *matching* combinators and lenses [89, 105], respectively.

BX combinators offer several benefits over other approaches [89]:

- development of type systems with strong behavioural properties is easier;
- allows flexibility in terms of choosing an appropriate *put* function for a *get* function;
- they are easier to extend with additional features, like alignment strategies [94, 105] or additional lens types [106].

A drawback of combinators is that they aren't reliably implementable in existing languages

[89], and are implemented in languages that are syntactically similar to existing languages. Language design is proposed by Anjorin and Ko [107] to go so far as to enable visual editing of combinators, mimicking circuit diagrams.

3.3.4 Other notable approaches

Other approaches to BX exist along with syntactic, semantic, and combinator-based BX. These approaches could hypothetically also prove to be complementary to the aforementioned ones, akin to the combining of syntactic and semantic BX [90].

Putback lenses are an approach that inverts the transformation derivation process; a *get* function is derived from a *put* function. This approach still considers that defining a single transformation function can lead to the derivation of a transformation function in the opposite direction, but with a postulate that it is easier to derive such transformations by observing a *put*. Fischer et al. [108] state that this is so because the *put* function has the potential to describe all intentions of the transformations. The round-tripping laws still apply and are observed in the putback approach [108], as well as the use of monads to control side effects [91]. Proof of concept languages BiFluX [109] and BiGul [100] were created using the putback approach, as well as a library for updateable views called BRUL [110]. The BIRDS language [85, 86] was used to facilitate a data management architecture and framework [85, 86, 111, 112]. A key indication of the complementarity of the putback approach with other approaches is that the aforementioned languages use lens combinators as proposed by the authors mentioned in Section 3.3.3; the difference being just the opposite direction of transformation function derivation.

A relational approach for a bidirectional language for XML, called biXid, was introduced by Kawanaka and Hosoya[113]. The proposed relational approach tackled some deficiencies of the combinator approach at the time regarding inconsistencies found in deep structures by expressing the relations between models with regular expression patterns.

3.3.5 Lenses

The term *lens* has been already used in this thesis to denote a pairing of a *get* and *put* function and was formally defined in Definition 27 through a well-behaved lens. A lens can be understood as a design pattern or data structure in terms of software engineering. Haskell already recognizes lenses as design patterns or at least structures. This is illustrated through the intuition of many authors [25, 104, 114] to represent lenses as records (invoking functions with $l.get$ where l is a lens record). The lens pattern, in its simplest form, contains a *get* and a *put* function which offer certain guarantees for bidirectional transformations (Figure 3.12).

Foster [104] additionally includes a $create$ function in the lens definition (Definition 33).

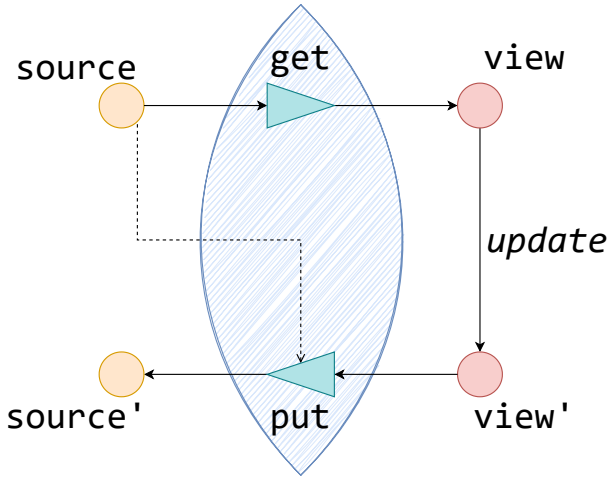


Figure 3.12: Basic lens as a structure

Definition 33. *Let S and V be sets of structures, where $s \in S$ and $v \in V$. A basic lens l, from S to V comprises of three total functions:*

$$l.get \in S \to V;$$

$$l.put \in V \to S \to S;$$

$$l.create \in V \to S;$$

obeying the following PutGet, CreateGet and GetPut rules:

$$l.get\ (l.put\ v\ s) = v;$$

$$l.get\ (l.create\ v) = v;$$

$$l.put\ (l.get\ s)\ s = s;$$

is $l \in S \Leftrightarrow V$.

Matsuda et al.[115] also annotate the lens as a generic type $L\ a\ b$ where a is the type of the source structure and b is the type of the view structure. They use L and *Lens* interchangeably between term expressions and code examples. In essence, it can be stated that $l \in (L\langle S,V\rangle \equiv S \Leftrightarrow V)$.

Foster [104] described a basic lens in general terms:

1. Lenses implement robust abstractions. Users can make arbitrary modifications to the view without having to consider whether their changes are consistent with the underlying source.
2. Lenses propagate view updates "exactly" to the source.
3. When possible, lenses preserve any source information that is not reflected in the view.

Lenses can be constructed compositionally through function composition defined in Definition 34 by Matsuda and Wang [115].

Definition 34. *Basic lenses Lens a b and Lens b c can be composed into Lens a c following:*
$(\hat{\circ}) : Lens\ b\ c \to Lens\ a\ b \to Lens\ a\ c$
$(Lens\ get_2\ put_2) \hat{\circ} (Lens\ get_1\ put_1) =$
$\quad Lens\ (get_2 \circ get_1)\ (\lambda v\ s \to put_1\ (put_2\ v\ (get_1\ s))\ s)$
where the "$\hat{\circ}$" represents the lens composition operator.

Additionally, lens composition from Definition 34 can be trivially extended to include *create* (Definition 35).

Definition 35. *Basic lenses Lens a b and Lens b c can be composed into Lens a c following:*
$(\hat{\circ}) : Lens\ b\ c \to Lens\ a\ b \to Lens\ a\ c$
$(Lens\ get_2\ put_2\ create_2) \hat{\circ} (Lens\ get_1\ put_1\ create_1) =$
$\quad Lens\ (get_2 \circ get_1)\ (\lambda v\ s \to put_1\ (put_2\ v\ (get_1\ s))\ s)\ (create_1 \circ create_2)$
where the "$\hat{\circ}$" represents the lens composition operator.

The complement can also be generally included in the composition definition (Definition 36).

Definition 36. *Basic lenses Lens a b and Lens b c can be composed into Lens a c following:*
$(\hat{\circ}) : Lens\ b\ c \to Lens\ a\ b \to Lens\ a\ c$
$(Lens\ get_2\ res_2\ put_2\ create_2) \hat{\circ} (Lens\ get_1\ res_1\ put_1\ create_1) =$
$\quad Lens\ (get_2 \circ get_1)$
$\qquad (res_2 \circ get_1)$
$\qquad (\lambda v\ s \to put_1\ (put_2\ v\ (res_2 \circ get_1\ s))\ (res_1\ s))$
$\qquad (create_1 \circ create_2)$
where the "$\hat{\circ}$" represents the lens composition operator.

The "$\hat{\circ}$" operator is associative, and has an identity lens id_L as its unit;
$id_L :\ Lens\ a\ a$
$id_L = Lens\ id (\lambda\ \_\ v \to v)$.
This as a consequence means that a set of lenses can form a category, where objects are types and morphisms are lenses; a lens of type *Lens a b* is a morphism from *a* to *b*[115].

Matching lenses were also shown to be composable with lens combinators in sequence, as products, as iterations, and as unions [89, 95].

Asymmetric lenses

Lenses mentioned so far are considered *asymmetric*; the lens pictured in Figure 3.12 as a visual example. The term "asymmetric" is used because the source *S* is primary and determines the

view V, but not the other way around [25, 93, 106, 114]. It can be stated that an asymmetric lens is a lens that contains a *get* and *put* function, keeping with the semantics of these functions explained in this section. An asymmetric lens can also contain a *create* and *res* function to guarantee levels of well-behavedness.

Symmetric lenses

Symmetric lenses were introduced by Hofmann et al. [25], where the source nor the view determine one another (Figure 3.13). This proposal postulates that a lens does not contain *get* and *put* functions, but rather contains opposite *put* functions. The complement is considered for both sides of the lens, so a complement C is a pairing of a complement from X and Y that can be noted as $(c_X, c_Y) : C$; such that $X \times C$ defines Y and $Y \times C$ defines X. These complements must be stored and managed, adding complexity to symmetric lenses not found in asymmetric lenses [114]. It is also interesting to note the subtle change in the notation, whereas S and V were used, now X and Y are used as sets of structures to signify the symmetry. A symmetric lens can be defined as shown in Definition 37 [25], and conditioned for very well-behavedness by Definition 38 [93].

Definition 37. *A **symmetric lens** l from X to Y (written $l \in X \Leftrightarrow Y$) has three parts: a set of complements C, a distinguished element $\texttt{missing} \in C$, and two functions:*

$$putr \in X \times C \to Y \times C;$$

$$putl \in Y \times C \to X \times C;$$

satisfying the following round-tripping laws:

$$\frac{putr(x,c) = (y,c')}{putl(y,c') = (x,c')}; \tag{PUTRL}$$

$$\frac{putl(y,c) = (x,c')}{putr(x,c') = (y,c')}. \tag{PUTLR}$$

*A symmetric lens satisfying PutRL and PutLR is considered **well-behaved**.*

Definition 38. *A symmetric lens l from X to Y is considered a **very well-behaved** lens if it satisfies the following laws:*

$$\frac{putr(x,c) = (y,c')}{putr(x',c') = putr(x',c)}; \tag{PUTPUTR}$$

$$\frac{putl(y,c) = (x,c')}{putl(y',c') = (y',c)}. \tag{PUTPUTL}$$

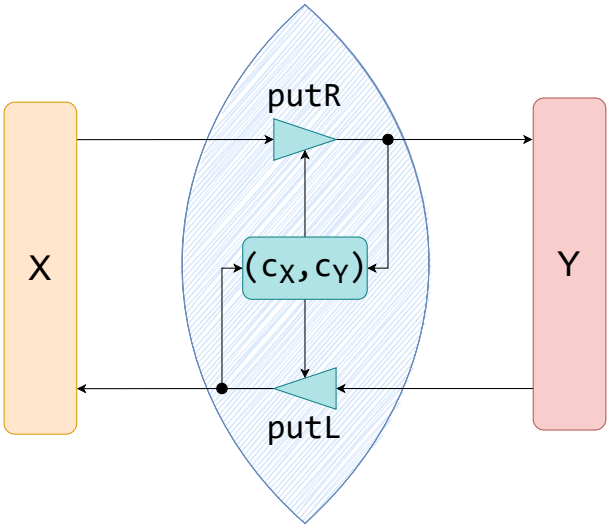


Figure 3.13: Structure of a symmetric lens

Simple symmetric lens

Miltner et al. [106] proposed a subset of symmetric lenses that don't contain a complement - *simple symmetric* lenses (Figure 3.14). On the other hand, simple symmetric lenses contain *create* functions used to create default values when introducing new data. *createR* and *createL* functions are created for each side of the lens. A simple symmetric lens is presented by Definition 39.

Definition 39. *A simple symmetric lens $l \in X \Leftrightarrow Y$ contains the following four functions:*

$$createR : X \to Y$$

$$createL : Y \to X$$

$$putR : X \to Y \to Y$$

$$putL : Y \to X \to X$$

subject to four round tripping laws:

$$putL\ (createR\ x)\ x = x \qquad \text{(CREATEPUTRL)}$$

$$putR\ (createL\ y)\ y = y \qquad \text{(CREATEPUTLR)}$$

$$putL\ (putR\ x\ y)\ x = x \qquad \text{(PUTRL)}$$

$$putR\ (putL\ y\ x)\ y = y \qquad \text{(PUTLR)}$$

Since simple symmetric lenses don't define complements, they don't require mechanisms for storing them. Consequently, simple symmetric lenses are symmetric lenses that satisfy the

property of "forgetfulness" [114].

Miltner et al. [106, 114] showed that simple symmetric lenses are strictly more expressive than classical asymmetric lenses. The authors also provide that an asymmetric lens is representable through a simple symmetric lens (Definition 40).

Definition 40. *Let* l *be an asymmetric lens.* l *is also a simple symmetric lens, where:*

$$l.createL\ y = l.create\ y \qquad l.createR\ x = l.get\ x$$

$$l.putL\ y\ x = l.put\ y\ x \qquad l.putR\ x\ y = l.get\ x$$

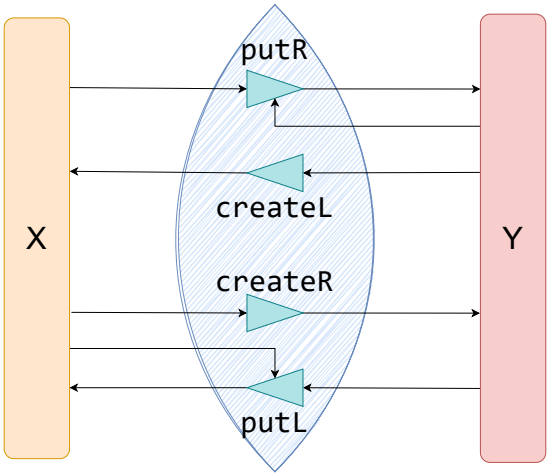


Figure 3.14: Structure of a simple symmetric lens

Chapter 4

Mask–mediator–wrapper architecture

This chapter presents the first contribution of this thesis regarding the MW architecture extension. This chapter is taken directly from the contributing paper [14] with some minor revisions. The shortcomings of the MW architecture are presented as the *raison d'etre* for the architectural extension in Section 4.1. The architectural extension is proposed in Section 4.2 as an addition of a new component type, supported by component rules realignment and addition, to rectify some detected problems in the separation of concerns of the MW architecture. An adapted quantitative shift-cost analysis is presented in Section 4.3 to prove that the extension is an improvement in terms of flexibility.

4.1 Problems with the mediator–wrapper architecture

Up to this point, the ways in which users connect to and use data management systems have been omitted. The way in which a user uses a system is a key point of a system's usability. Multiple authors have opted for connecting users to the system via specialized applications which are system specific [61, 63, 64]. Such applications connect directly to the highest mediator layer (or the integration layer in case of an alternative architecture).

This puts additional responsibilities on the mediators in the higher layers. These mediators not only have to mediate schemas from lower layers, but also manage their GESs, as shown in the example of Figure 2.17. Opposed to this, the exporting mediators, shown in Figure 2.18, seem like a better solution due to the functional responsibility being shared among multiple mediator components. This also has its issues. **RMe1** prohibits the mediator from exporting data in a format that is not internally used by the system itself, meaning that data translation is going to have to be done in a user application. This breaches the system's separation of concerns, leading to client applications having to perform the translations. It seems that this responsibility cannot be shared among components of the mediator type. Therefore, a third rule for mediators must be added:

RMe3 Mediators should be used to mediate, not to represent.

This problem is further exacerbated when one takes notice of the user applications usually implemented alongside these systems. Although user applications generally display just one format of data, it is interesting to notice the variety of data formats that have been used as presentational in different systems - from JSON collections [58, 116] and XML documents [117], to tabular data [118, 119]. The way of access can also be varying - a JDBC API [119], web applications built onto the system [58] and even a web API [116].

This is also the case with state-of-the-art databases and frameworks designed with specific representations of data sources in mind. Some authors still show a preference for an SQL interface [41, 42, 67, 120], while others prefer a key-value [40, 43, 44], graph [34, 35, 36], semantic web [38, 39], an XML [121], flattened data [122], or a plain-text interface [123]. Pang et al. [53] also showed a system with three types of data representations: object storage via a REST API [124], file storage, and a NoSQL table store service. Benedikt et al. [125] and Qin et al. [126] also showed that data representation (views) is becoming a key factor in data handling. Dehghani [68] stated these same aspirations for the data mesh, concretely mentioning the need for events, batch files, relational tables, and graphs as models of served data.

With the increase in data format variety (illustrated by Table 4.1), it is becoming more apparent that a data source integration system, as a singular data source, will itself have to support data representation in different formats. It is important to note that data, schemas, and queries face this same issue equally.

A more general point is that the MW architecture in its current state diverges from the concepts of architectural layering, separation of concerns, managing dependencies, control flow and testability. These concepts pave the way for a flexible and largely scalable system [46, 127]. Such a system is an expected requirement for gathering and managing large amounts of data from multiple sources.

Table 4.1: Overview of existing data management concepts and projects in regards to their data representation.

Reference (Project)	Data Representation and Access
[4][9][8]	specialized desktop application
[116][58][53][124]	JSON (+ web API)
[117][121]	XML
[34][36][35]	graph
[119]	JDBC
[58]	Web application
[118][119][67][41][120][42][122]	tabular data
[44][43][40]	key-value
[38][39]	semantic web
[123]	plain text

4.2 Extending the mediator–wrapper architecture

It is evident that currently, in the MW architectural pattern, the responsibility of representing data, schemas, and queries cannot be assigned to any of the existing component types without assigning too much responsibility to them. For this reason, the system designer is forced to decide whether to assign this responsibility to the highest mediator layer or a user application.

Due to the nature of the problem being the assignment of a system functionality to a component type, and all existing component types being finely utilized via their given rules, it has become obvious that there is a component gap in the upper layer of the MW architecture. In other words, due to **RMe3**, there is a task that no component type is adequate to additionally handle. Hence, there is a requirement for another type of system component that could take on the responsibility of representing system data.

A new theoretical component is introduced to the existing MW architecture, which is named a *mask*. A mask masks the system at a certain point in the schema hierarchy into a representational form that can be easily handled by users, effectively taking on the responsibility of representing the system. The mask should be placed at the top of the architectural hierarchy,

positioned between the users and the highest mediator layer. Placing the masks on top of the architectural hierarchy effectively creates a mask layer. Consequently, this extended variant of the MW architecture is called the *mask–mediator–wrapper* (MMW) architecture. Figure 4.1 displays the positioning and relationship of the mask components and layer with other components in the architecture.

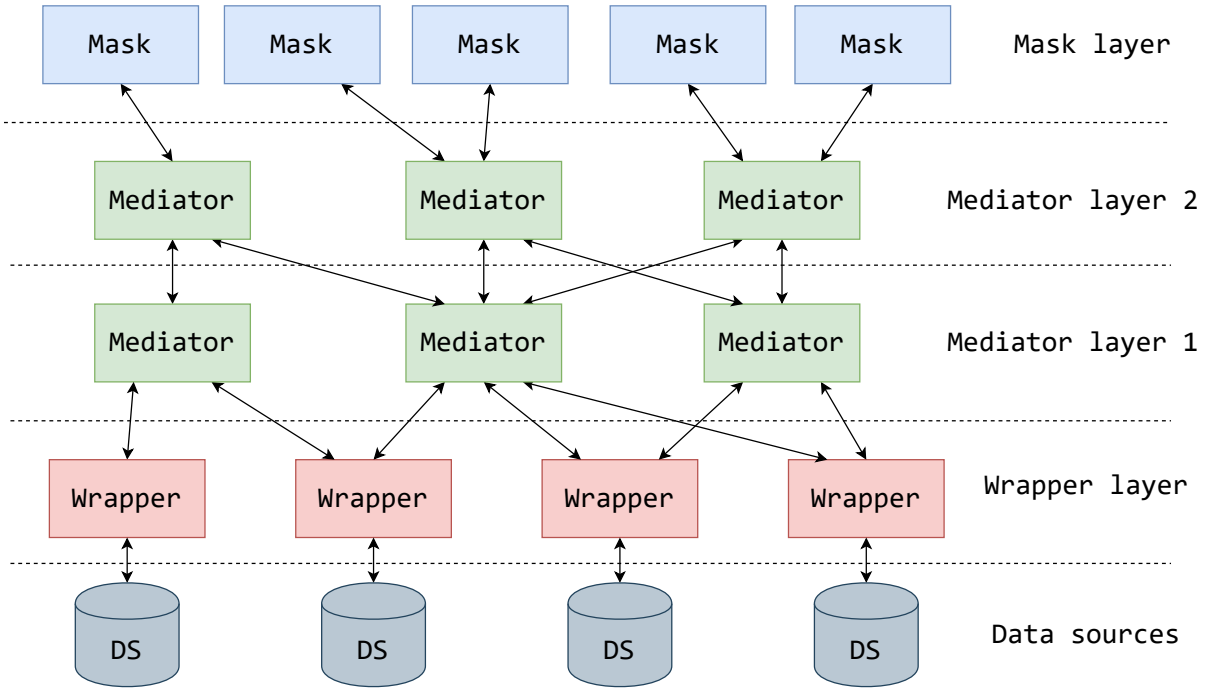


Figure 4.1: The MMW architecture with layered mediators

Using the mask, the system's representational logic is decoupled from the system's mediation logic and the user's application logic. Furthermore, by adding the mask as a system component type, the system has a finer separation of responsibility and gains benefits that help expand and simplify its usability. If a mask supporting a form of standardized technological access to data is implemented, then access to the system becomes available to a wide variety of applications implemented over that standard of access.

Observing the mask with an implementation example, one could implement a mask in the form of a REST service with requests over URLs returning resources in JSON, akin to the system access shown in [116]. In this way, any application built to send requests to a REST service and receive its responses can now be used as a client application.

Another interesting way to look at a mask component is to imagine it as an inversed wrapper, as illustrated by the flow and dissemination of data in Figure 4.2. While the wrappers concern themselves with adapting the source data from the outside world to accommodate the data source integration system's standard, the masks concern themselves with adapting the standardized data to accommodate the outside world. Additionally, wrappers import data from multiple sources, while masks export data to multiple destinations. Hence, the data source integration

system can now be seen as a single logical point of collecting, transforming and providing data in various formats. This is in accordance with the modern notion that data management systems consume, transform, and serve data.

As was the case with wrappers and mediators, the rules for masks are set as follows:

RMa1 A mask should be positioned at the top of the architecture.

RMa2 A mask only connects to a single mediator.

RMa3 A mask is used for representational purposes, representing a schema, querying data, and representing the result data.

RMa1 follows from the consensus that the presentation layer in system architectures is positioned at the top (furthest on the user side). The mask, its use being representation, is the system's presentation layer.

RMa2 follows from the reversal of its statement. The mask could connect to multiple mediators, then it would need to also apply mediation - breaking the separation of responsibilities among the components. Hence, a mask is allowed to connect to just one mediator, and all the mediation is left to the the mediators.

RMa3 states a set of basic functional requirements that are expected of most data access systems. This rule articulates that the mask component does not in any way diminish the system's functionality.

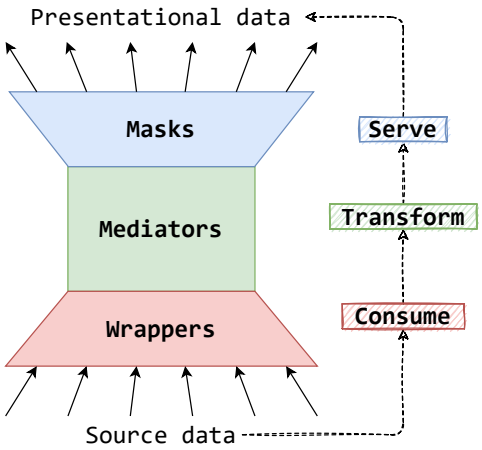


Figure 4.2: Stylistic view of the MMW architecture

The mask component type is succinctly defined in Definition 41.

Definition 41. *The mask is a component used to manage the representation of uniform schemas, queries, and data.*

The presented rules for masks, mediators, and wrappers imply that they are architectural quanta - *independently deployable components with high functional cohesion, which include all structural elements required for the system to function properly* [17]. This surprising compat-

ibility with modern architectural ideas is the primary reason why the MW architecture and its extensions can be considered a relevant research topic even today.

4.2.1 The mask's effect on the system schema hierarchy

To show that the addition of masks affects only the mediators in the higher layers and decreases these mediators' responsibilities, in Figure 4.3, the assignment of schemas from the system-wide schema hierarchy from Figure 2.16 is shown. As in Figures 2.17 and 2.18, Figure 4.3 shows components and their assigned schemas adjacent to them in white rectangles.

The wrappers themselves and their schemas have remained unchanged, but there is a significant difference above the first mediator layer. It is important to note that the placement of prior existing components has not been changed—all the mediators still connect to the same wrappers, and the mediators all operate over the same GCSs. Analogous to the examples shown in Figures 2.17 and 2.18, the mediator components of the (now only existing) mediator layer operate over their respective GCSs. The mask components have been assigned all the GESs.

There is a noteworthy schema rename in the example of Figure 4.3, for what was originally GCS_{123}. As the GCS_{123} itself was an exported schema in prior examples, in this example, the schema might be in a fundamentally different format. Hence, the schema operated over in the mask cannot be named the same as the schema in the mediator. To mark this change, what was once GCS_{123} used for exporting is now GES_{123} - a fully-fledged exported schema.

A similar effect can be seen in the case of the mediator that in Figures 2.17 and 2.18 operates over the LES_{41}. As this mediator's schema is not directly exported, it is renamed GCS_{41}, although it currently only incorporates LCS_4. The mask component above this mediator has taken the responsibility of representation and is consequently assigned LES_{41}.

There is also an interesting case in Figure 4.3 concerning the translation of schema LCS_4 to the upper layers (via LCS_4, GCS_{41}, LES_{41}) and the components used for this task. The previously exporting mediator of LES_4 from Figures 2.17 and 2.18 is preserved. This mediator's schema is also renamed to GCS_{41}, as stated earlier. For the moment ignoring the **RMa2**, it can be questioned whether this purely translational mediator is even required - rightly so, if it is additionally considered that the system probably uses standardized interfaces for inter-component communication. It could be concluded that the mask with LES_{41} could be connected directly to the wrapper with LCS_4.

However, this is not the case, as the mediator with GCS_4 (formerly LES_4) must be preserved. The reason for this statement is two-fold from the angle of system design. Firstly, the mediator is not only used for translation but also enables transformations within the schema itself (as is stated by **RMe2**). Connecting the mask directly to the wrapper, although feasible, would disable the system to apply further transformations on schema LCS_4. Secondly, the benefit of using an MW architecture, and by extension our own, is the ability to append data sources after

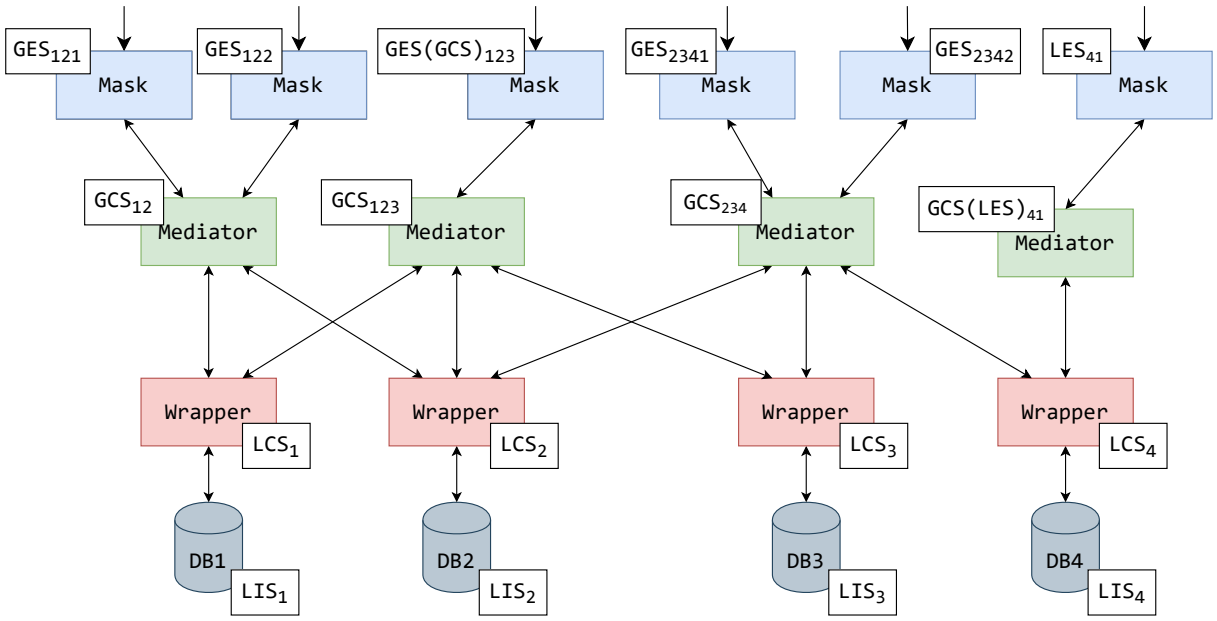


Figure 4.3: An exemplified assignment of schemas to a MMW system

the system has been set up. Connecting the mask directly to the wrapper leaves the system without a mediator to mediate between the wrapper with LCS_4 and any additional to-be-connected wrapper. Because of this, the system would lose the beneficial property of being (completely) appendable.

This is an example of how the **RMa2** preserves not only the component hierarchy of the architecture but also the properties of the system itself.

4.3 Quantitative shift-cost analysis of the mediator–wrapper architecture

To prove that the MMW architecture simplifies an MW-based data source integration system's maintenance and change management (flexibility), a levelled quantitative analysis to compare the MMW and MW alternatives is needed. The following analysis is based on an evolution-cost quantitative analysis for measuring software flexibility described by Eden and Mens [22], as described in Section 4.3.

Eden and Mens [22] proposed that a software's flexibility can be measured and compared to other designs by approximating the cost of implementing anticipated changes - shifts. The cost of shifts is defined as the quantity of software units that need to be changed, added, or removed. These software units are called modules in a general sense but are exemplified with classes and methods.

To adjust this analysis for the level of architecture design in this paper, the modules are viewed as architectural components. The analysis compares an isomorphic example (shown

in Figure 4.4) of an MW architecture with one mediator layer (1LMW), an MW architecture with two mediator layers (2LMW), and an MMW architecture. In the cases of 1LMW and 2LMW, mediators are considered to have functionalities of both mediation and representation. The 1LMW and 2LMW architectures were chosen for this analysis because they generally represent the solutions of the GARLIC and TSIMMIS system architectures. The GARLIC has been presented as both a 1LMW and 2LMW system, while the TSIMMIS has been presented as a 2LMW system. In Figure 4.4, red rectangles represent individual wrappers, green rectangles represent both mediators with representational functionality and mediators without (tick marks representing names of mediators without representational functionalities), and blue-green rectangles represent masks in the MMW architecture and mediators in the 2LMW architecture.

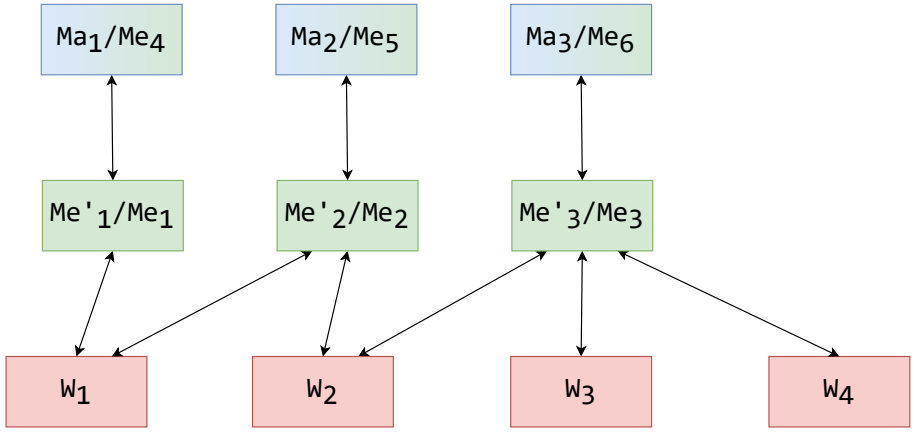


Figure 4.4: Architecture used in the analysis

The analysis is conducted over four scenarios: adding a new representation type, adding a new representation, adding a new mediator, and adding a new wrapper to a mediation. The symbolic nomenclature for this analysis is defined as follows:

> For a set of components S_{comp} of possible types $S_{comptypes} = \{Ma, Me, Me', W\}$ representing a mask, a mediator with representational functionality, a mediator without representational functionality, and a wrapper respectively, and a set of possible actions over those components $S_{act} = \{impl, depl\}$ representing implementation and deployment respectively, C_X^Y us the cost of performing an action $Y \in S_{act}$ over component $X \in S_{comp}$, with the addition of C_{Conn}^{set} signifying the cost of setting up a connection between a pair of components $\{(c_1, c_2) \mid c_1, c_2 \in S_{comp}\}$

Since a mediator with representational functionality is more complicated to implement than a mediator without representational functionality, the cost of implementing the former is greater than the latter:

$$C_{Me}^{impl} > C_{Me'}^{impl} \tag{4.1}$$

Due to a greater number of functionalities that need to be supported by the surrounding system to which the component is being deployed, the deployment of a mediator with representational functionality is also more costly than that of a mediator without representational functionalities. This is because their deployment includes the tasks of setting up system resource access permissions, component settings, and firewall rules, all of which are either increased in quantity or complexity in the case of a representational mediator. Therefore, the following expression is concluded:

$$C_{Me}^{depl} > C_{Me'}^{depl} \tag{4.2}$$

As in the former statement, for the same reasons, the deployment of a mask component is considered less costly than a mediator with representational functionalities. In addition, the mediator has a communication node intended for access to multiple sources. This is considered bloat, as the representational components connect to only one component in the lower layer. The mask, on the other hand, has a communication node inherently allowing just one connection to the lower layer (as per **RMa2**), making the connection configuration simpler. Therefore, the following expression is concluded:

$$C_{Me}^{depl} > C_{Ma}^{depl} \tag{4.3}$$

Scenario 1: Adding a new representation type

In this scenario, a requirement for a new representation type on top of the combined schemas of wrappers W_2, W_3, and W_4 is added. Since a new type of representation is required, in a 1LMW, an entirely new mediator must be implemented. This new mediator also must be deployed and connected to wrappers W_2, W_3, and W_4. The outcome of the shift on 1LMW is displayed in Figure 4.5 (added elements are marked with dashed lines), with the addition of Me_4 and its connections to the required wrappers.

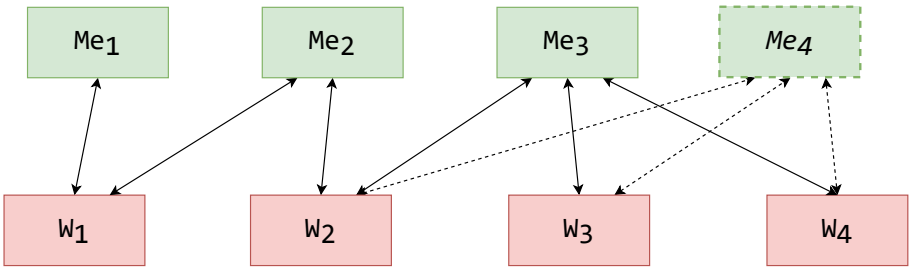


Figure 4.5: Scenario 1 outcome on a one-layer mediator MW architecture

Thus, the cost of the shift is

$$C_{1LMW}^{1} = C_{Me}^{impl} + C_{Me}^{depl} + 3 \times C_{Conn}^{set}$$

In a general case, with the number of connected wrappers being N, the cost is

$$C^{1}_{1LMW} = C^{impl}_{Me} + C^{depl}_{Me} + N \times C^{set}_{Conn}$$

It can be noticed that this architecture forms redundant connections between wrappers and mediators, adding to the shift cost.

Again, in the case of a 2LMW, a new mediator must be implemented and deployed (Me_7). In this case, the mediator is stacked on top of a mediator on the lower layer of mediators (Me_3). Hence, mediator Me_3 is reused for combining wrappers W_2, W_3, and W_4, and only one connection is set up. The outcome of the shift on 2LMW is displayed in Figure 4.6, where the added mediator component and connection are illustrated with dashed lines.

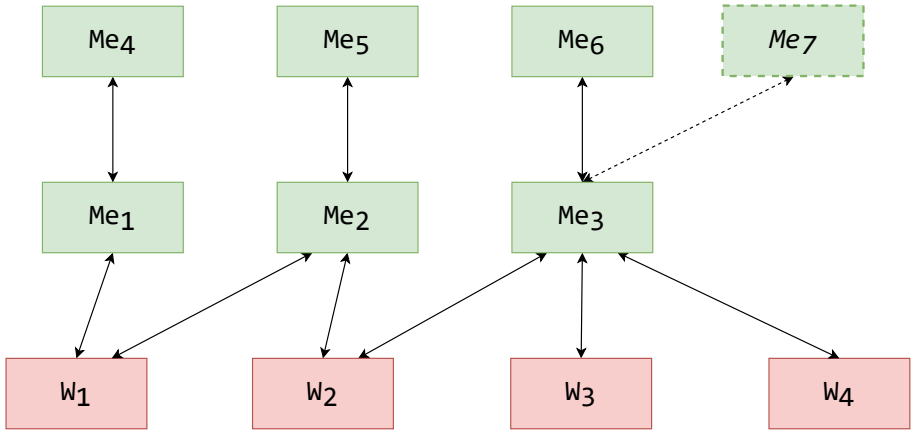


Figure 4.6: Scenario 1 outcome on a two-layer mediator MW architecture

The cost for this shift is

$$C^{1}_{2LMW} = C^{impl}_{Me} + C^{depl}_{Me} + C^{set}_{Conn},$$

which remains true for any general case.

In the case of an MMW architecture, to create a new type of representation, a new mask (Ma_4) is required to be implemented and deployed. Only one connection setup is required, as the new mask only connects to one mediator in the mediator layer (Me'_3). The mediators in this architecture do not serve a representational purpose, so they do not have representational functionality. The outcome of the shift on MMW is displayed in Figure 4.7, with the added mask component and connection illustrated with dashed lines.

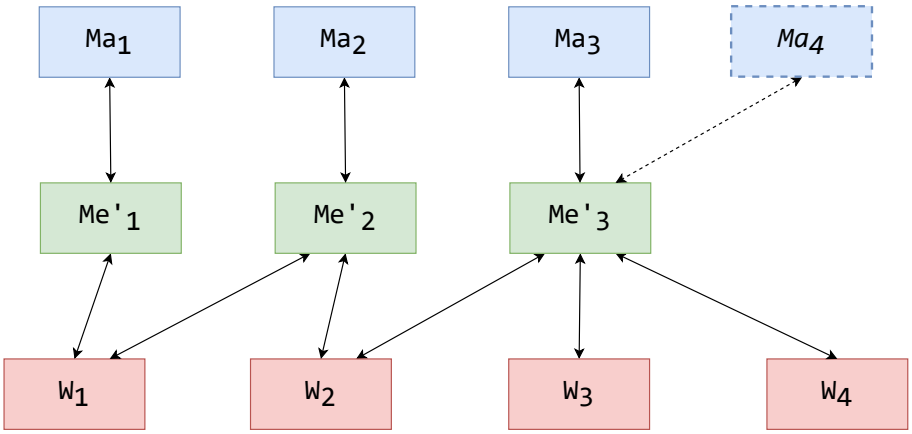


Figure 4.7: Scenario 1 outcome on a MMW architecture

The shift cost in this and general cases is:

$$C^1_{MMW} = C^{impl}_{Ma} + C^{depl}_{Ma} + C^{set}_{Conn}$$

Scenario 2: Adding a new representation

In this scenario, a requirement for a new representation on top of combined schemas of wrappers W_1 and W_2 is added. The representational component is already implemented, so none of the cases will have an implementation cost, just the deployment cost and the cost of connection setup.

In a 1LMW, the new mediator is deployed, and two connections to the two wrappers W_1 and W_2 are set up. The shift cost is

$$C^2_{1LMW} = C^{depl}_{Me} + 2 \times C^{set}_{Conn}$$

In a general case, the shift cost is determined by the number of required redundant connections to wrappers N:

$$C^2_{1LMW} = C^{depl}_{Me} + N \times C^{set}_{Conn}$$

In a 2LMW, a new mediator is deployed, and one connection to its underlying mediator (Me_2) is set up. The shift cost is

$$C^2_{2LMW} = C^{depl}_{Me} + C^{set}_{Conn}$$

In an MMW architecture, a new mask is deployed, and one connection to its underlying mediator (Me'_2) is set up. The shift cost is

$$C^2_{MMW} = C^{depl}_{Ma} + C^{set}_{Conn}$$

Scenario 3: Adding a new mediator

In this scenario, a requirement for a new mediator over wrappers W_2 and W_3 is added. It is assumed that this type of mediator already exists, so there is no cost of implementation.

In a 1LMW, a mediator is deployed and connected to the two wrappers. The shift cost is:

$$C^3_{1LMW} = C^{depl}_{Me} + 2 \times C^{set}_{Conn}$$

Again, for a general case where N is the number of connected wrappers, the shift cost is:

$$C^3_{1LMW} = C^{depl}_{Me} + N \times C^{set}_{Conn}$$

In a 2LMW, a mediator must be deployed to the lower mediator layer to combine the wrappers and a mediator in the upper mediator layer to provide the representation. The set-up connections also must be considered, as two connections are set up toward the wrappers and a single connection between the mediators. The shift cost is:

$$C^3_{2LMW} = 2 \times C^{depl}_{Me} + 3 \times C^{set}_{Conn}$$

In a general case, where N is the number of connected wrappers, the shift cost is

$$C^3_{2LMW} = 2 \times C^{depl}_{Me} + (N+1) \times C^{set}_{Conn}$$

In an MMW architecture, together with deploying a mediator, a mask must be provided. The mask type is considered as already implemented (analogous to the cases of MW architectures), so it has to only be deployed. Two connections are set up toward the wrappers and a single connection between the mediator and mask. The shift cost is:

$$C^3_{MMW} = C^{depl}_{Me'} + C^{depl}_{Ma} + 3 \times C^{set}_{Conn}$$

In a general case, where N is the number of connected wrappers, the shift cost is

$$C^3_{MMW} = C^{depl}_{Me'} + C^{depl}_{Ma} + (N+1) \times C^{set}_{Conn}$$

Scenario 4: Adding a new wrapper to an existing mediation

Additionally, to demonstrate that these architectures are sound (the MMW architecture first and foremost), a scenario of adding a new wrapper can be analyzed. The appending of wrappers to an existing mediator does not impact the rest of the components, as the wrapper is deployed and a single connection to the required mediator is set up. Thus, the shift cost for all architectures

is:

$$C^4_{1LMW} = C^4_{2LMW} = C^4_{MMW} = C^{depl}_W + C^{set}_{Conn}$$

Analysis of the Shift Costs

With the shift costs evaluated, a more concise comparison of architectures can be made. Table 4.2 displays all the shift costs for each scenario and architecture.

Table 4.2: Shift costs for all scenarios and architectures

Sc.	1LMW	2LMW	MMW
1	$C^{impl}_{Me} + C^{depl}_{Me} + N \times C^{set}_{Conn}$	$C^{impl}_{Me} + C^{depl}_{Me} + C^{set}_{Conn}$	$C^{impl}_{Ma} + C^{depl}_{Ma} + C^{set}_{Conn}$
2	$C^{depl}_{Me} + N \times C^{set}_{Conn}$	$C^{depl}_{Me} + C^{set}_{Conn}$	$C^{depl}_{Ma} + C^{set}_{Conn}$
3	$C^{depl}_{Me} + N \times C^{set}_{Conn}$	$2 \times C^{depl}_{Me} + (N+1) \times C^{set}_{Conn}$	$C^{depl}_{Me'} + C^{depl}_{Ma} + (N+1) \times C^{set}_{Conn}$
4	$C^{depl}_W + C^{set}_{Conn}$	$C^{depl}_W + C^{set}_{Conn}$	$C^{depl}_W + C^{set}_{Conn}$

The first scenario demonstrates that in the MMW architecture, the addition of a new type of representation is only dependent on the implementation and deployment of a mask component. The other two architectures depend on mediator components. The 1LMW shift cost noticeably depends on the number of connected wrappers - to emphasize, for adding a representation type. The 2LMW and MMW architectures are not at such a disadvantage, their difference being the type of component added to the system. Since a mask is less costly to implement and deploy than a mediator, the overall shift cost in scenario 1 is the lowest in the MMW case.

The second scenario also shows that the 1LMW shift cost is dependent on the number of wrappers. The cases of 2LMW and MMW are again analogous, but a mask is less costly to deploy than a representational mediator. This makes the shift cost of the MMW case the lowest again. As it was discussed earlier in the text, using a mediator just for representation, without using its mediation functionalities, is akin to killing a fly with a cannonball.

The third scenario shows the shift cost overhead that 2LMW and MMW have as opposed to 1LMW when setting up mediation. There is an obvious trade-off in these architectures between the shift cost of adding mediation or representations. To maintain a less costly (and qualitatively simpler) representation addition, the overhead cost of adding mediation is increased. This overhead can be quantified for 2LMW as

$$C^{overhead}_{2LMW} = C^3_{2LMW} - C^3_{1LMW} = C^{depl}_{Me} + C^{Set}_{Conn},$$

and for the MMW,

$$C_{MMW}^{overhead} = C_{MMW}^{3} - C_{1LMW}^{3} = (C_{Me'}^{depl} - C_{Me}^{depl}) + C_{Ma}^{depl} + C_{Conn}^{set}.$$

Considering that the MMW mask and mediator are less costly to deploy than the 2LMW mediator, the overhead cost is reduced in favour of the MMW.

The fourth scenario shows that the addition of a new wrapper to the MMW system has no effect on the rest of the system hierarchy, as it is also expected of the other MW architectures. The MMW finds itself in no detrimental opposition to the other architectures.

Chapter 5

Mask component

This chapter details the mask component implementation up to the point of prescribing the paradigm or technology in which the mask should be implemented. Sections of this chapter are taken from the contributing paper [14] with some minor revisions. This chapter presents the closing details of the first thesis contribution regarding the concretisation of the mask component and sets the groundwork for the third thesis contribution regarding the outlines of the mask framework.

5.1 Mask component functional requirements

The general practice up to this point was to analyze the mask as a generic black box component and explain how it would work in synthesis with other system components. To expand the idea of the mask even further, it can no longer be observed just as a black box. The possible inner workings of a mask provide the ability to distil this architectural component even further in terms of design and development. As with most software systems, the mask, a miniature system itself, can be internally elaborated by following some functional requirements.

Using the mask's properties that have been introduced via its defined rules, relations to other components, and effect on the architectural layout, we introduce some basic functional requirements:

F1 The mask must interface with the system via mediators. The mask connects to just one mediator, but it should in general be able to connect to and communicate with any system mediator interchangeably. A connection with a wrapper is feasible, but it is inadvisable and thus not of primary concern.

F2 The mask must provide a user access interface. The user access interface is the point of user system access. This interface can take any implementational form, provided that the chosen form has presentational abilities for data storage concerns. This interface is interchangeable and does not have an effect on the general way in which data source component translations

take place.

F3 The mask must translate schemas from the system format to the user access (masked) format. The mask ascertains the system schema provided by its connected mediator and adapts the schema to a defined mask format.

F4 The mask must translate queries from the user access (masked) format to the system format. The queries are given by the user through the user access interface in a masked format and are translated into the system format. To determine mask-to-system element mappings, the query translation can use the results of schema translations.

F5 The mask must translate results from the system format to the user access (masked) format. The results received through the system must be adapted to the defined mask format. To determine certain metadata aspects (e.g., the naming of attributes) of the data results, the results of the schema translations can be used.

The requirement **F1** follows from **RMa1**, **F2** from **RMa2**, and the requirements **F3**, **F4** and **F5** from **RMa3**.

5.2 Mask inner components

Following the functional requirements from Section 5.1, a conceptual depiction of the mask's inner components is devised. This is displayed in Figure 5.1 as a conceptual model of functional components and the types of data they are expected to handle. These components present a generalized idea of what kind of functionalities a mask should have and what their relationships should be in terms of data exchange and dependency. These components do not present real-world components, but rather a possible grouping of some real-world components providing a functionality.

This sketch allows the mask's functionalities to be put into context. The schema, query, and result translators are recognized as components with the task of translating data source components. The central role in translation is given to the schema translator as queries and query results are translated by using schemas generated by the schema translator. The system access interface is used to connect to the system via a mediator in the layer below. The outer access interface is a generic component, able to accommodate an adequate form of an access interface.

A noticeable trademark of this model is that there is a focus on the flow of data and its conversion by the components. A masked query is translated and sent (down) into the system. Reciprocally, the result of such a query is translated into a masked format to be sent (up) to the user. Similar is the case of schema translation; the system specified schema is translated into a masked schema for presentation to the user.

Such data transformations can only be achieved through processes, so in a general sense, it

is more sensical to discuss the mask in terms of processes and the data that flow between them. To achieve a more detailed elaboration of the mask, building upon the model from Figure 5.1, a data flow diagram is constructed as displayed in Figure 5.2. Figure 5.2 displays the recognized processes as circles, outer entities as rectangles (users and mediator), data storage as open rectangles, and flowing data as named arrows.

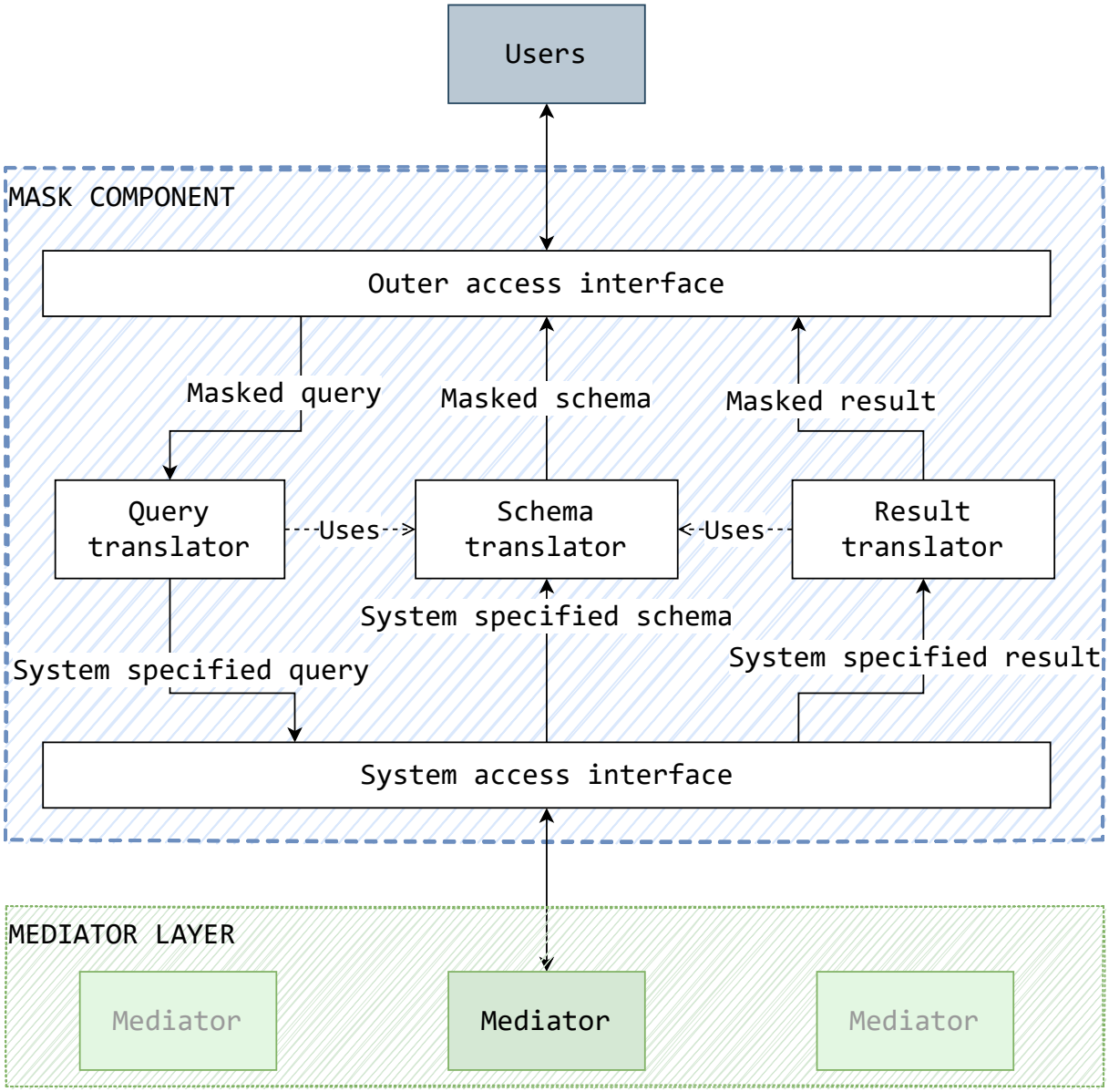


Figure 5.1: A conceptual model of the mask's functional components

On the right-hand side of Figure 5.2, the schema translator of Figure 5.1 is decomposed as two processes: schema loading and schema translation. The schema loading process is concerned with the acquisition of a system schema from a connected mediator. This schema is also stored for other usages, besides schema translation, but it should be reacquired frequently to maintain an up-to-date schema. For this reason, schema loading is considered a separate and independent process. The schema translation process uses the currently acquired system

schema and schema mapping rules from a separate storage to create a masked schema. This masked schema is presented to the user.

The querying process in Figure 5.2 is a complex process that concerns itself with querying over a mediator. It is closely tied to processes of query translation and result translation. These processes are effectively subprocesses of querying but have been extracted due to their importance and correlation with components in Figure 5.1.

The query translation process translates a masked query into a system-formatted query. It requires data about the current system schema and schema mapping rules to determine the way in which they are reflected in the current query. This must be considered, as the schema translation might change the resources' names or change their schematic, so it becomes important to reverse those translations when constructing a system query. The query translation process also requires query mapping rules. As a general example, and for the moment setting a simplified generic model for a query - these rules might explain how a projection or selection in a query is to be translated.

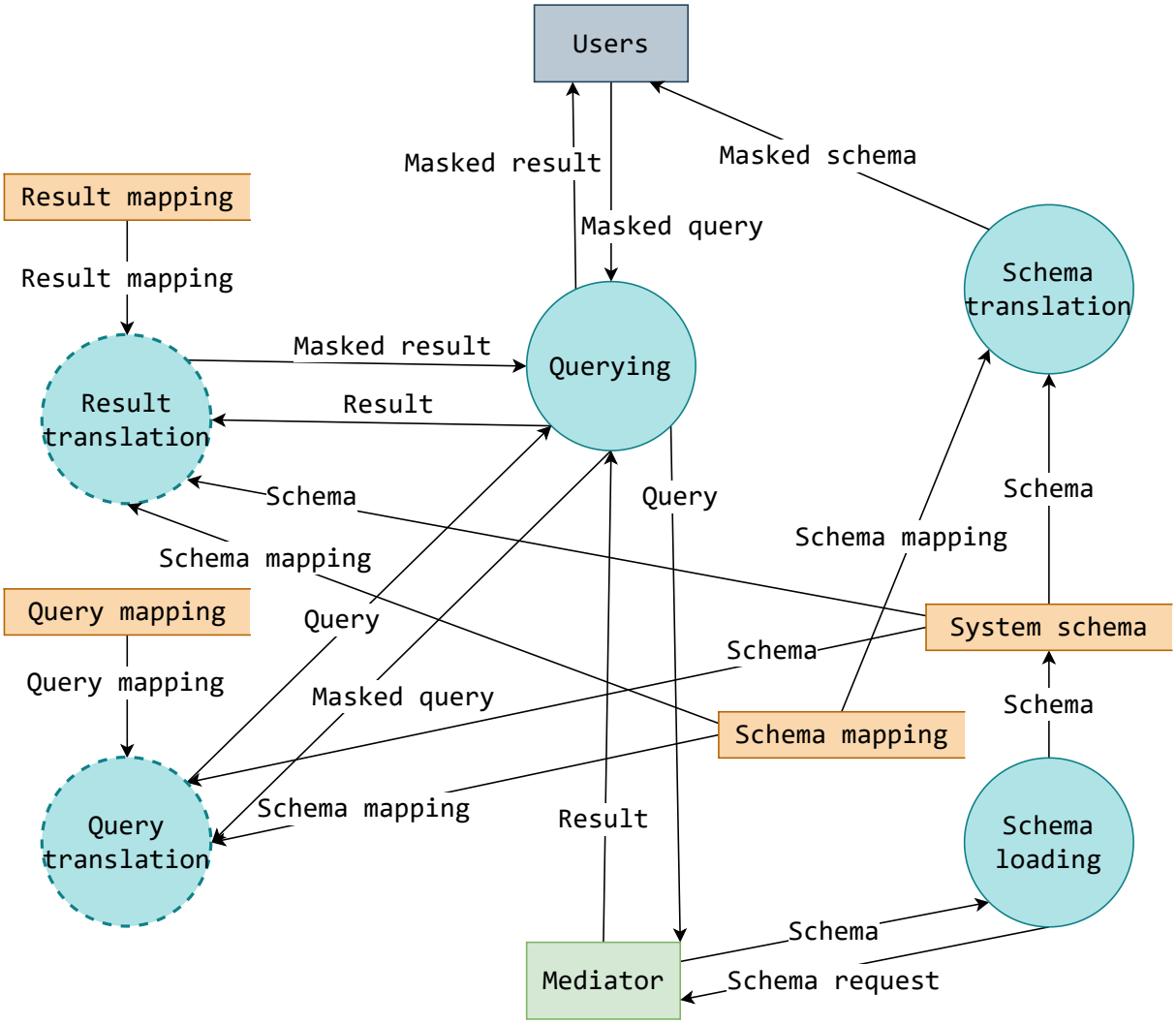


Figure 5.2: Dataflow diagram following the mask's functionalities

The result translation process also requires data about the current system schema and schema mapping rules, as it is also concerned with translating a small view-like portion of the schema with the addition of holding result data that can also go through some masking transformations. Just like in other translational cases, result data translation also requires some mapping rules for data results.

There is also a very interesting feature of the diagram in Figure 5.2 regarding all the mapping (rules) data storage. The result mapping, schema mapping, and query mapping data storage do not have any data inflows. These mappings are, in the context of this diagram, then clearly provided by some other undefined source. In fact, these mappings can only be provided by the developers of a certain mask component. These mappings are the exact point at which the system can no longer be designed as generic or abstract, and some concrete implementation or empirical data describing the masking of the system is required.

Considering the mentioned findings, a high-level design for a mask is proposed and shown in the diagram of Figure 5.3. The goal is also to think of the mask's component design without reducing generality to avoid prescribing any concrete programming paradigms, languages, or specific design patterns. Figure 5.3 illustrates the recognized components of a mask as white rectangles (cylinder in the case of a metadata database), required implementations for a mask kind as grey rectangles (implementation indicated by arrows with empty arrow-heads), and data flow as arrows with black arrow-heads.

Following the process inference of the data flow diagram of Figure 5.2, in Figure 5.3, the schema manager and schema translator (interface) are recognized as components - analogous to the schema loading and schema translation processes, respectively. The schema manager observes the system schema and updates the stored system schema appropriately.

The inference of a general querying process in the data flow diagram leads to the recognition of the query manager component. This component manages the underlying translations and query execution sequencing. In general, its purpose is to produce a masked result for a masked query. This is achieved through the processes of query and data result translation, which themselves in turn lead to the recognition of the query translator (interface) and data result translator (interface) components.

Regarding the mask's communication abilities with the rest of the system, the mask sends system format queries and results, just like mediators and wrappers. For this reason, the mask can use a standardized communication node used in all other component types. Due to the communication restrictions of the mask (allowing for a single mediator connection), an extension of a basic communication node should also be implemented. This also allows the addition of new message types if such a requirement should arise later.

To store all the data inferred for storage in Figure 5.2, a metadata store should also be introduced to the component. Such a store would be used by all components that require at least

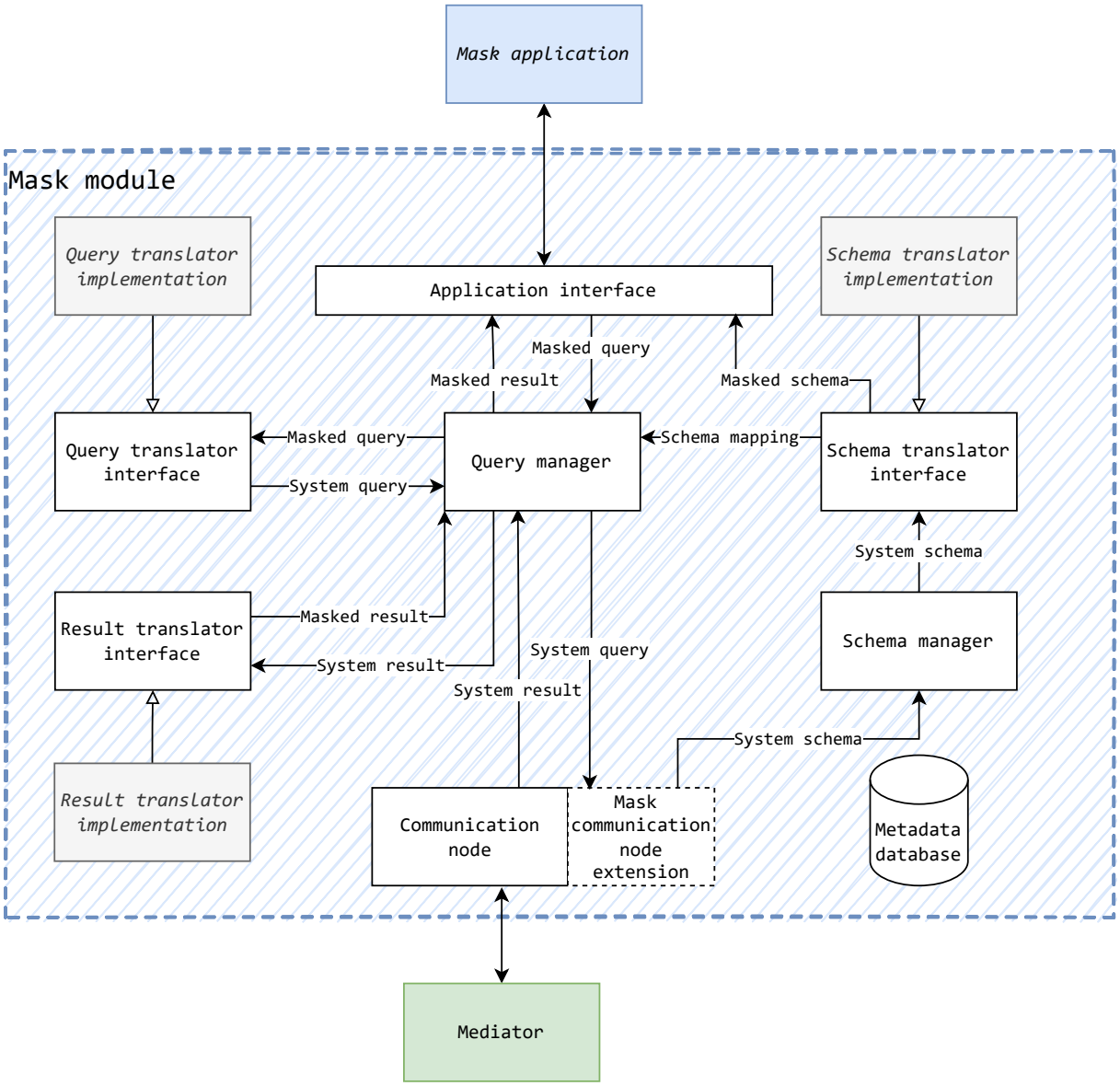


Figure 5.3: High-level design of the mask component

some schematic information or technical information. In Figure 5.3, this store is displayed, but the connections to other components are omitted for the sake of clarity.

On the other hand, the mappings (marked as data storages in Figure 5.2) are not stored in the metadata database. They are presented by the component implementations (marked grey) in Figure 5.3. For the mask to remain as generic as possible, such mapping rules are explicitly described by the schema, query, and result translator implementations. The aforementioned interfaces are used to keep a level of abstraction toward the other inner components. The implementations are case-specific and created by the mask's developers according to their respective interfaces. This allows the development of a mask kind to be done without the need for extensive coding, as only the missing implementation pieces need to be filled in. This by consequence, not only simplifies the development process but also decreases the time required to develop a

certain kind of mask component.

It is important to note that the term “interface” is used in its broadest form here, not excluding the development of the mask component in a non-OOP language. Along with these interfaces being implemented as standard OOP interfaces, they can also be implemented as high-order functions in a functional paradigm or as separate implementations of function prototypes defined in library headers (to be linked before compilation) in a structural-procedural paradigm. This is one of the beneficial results of generic and abstract reasoning about mask components.

The inner components that have been elaborated up to this point are part of a mask module or rather a library. This is best understood from the point of another component marked solidly blue in Figure 5.3 - the mask application. The mask application is the execution entry point of the mask component. In the continuation of previous possible use-case examples, this component could be a web API or a TCP server listening for JDBC. Whichever the exemplified case, it would use the mask module as a library to connect to the integration system. The interfacing of the mask application and the module is achieved through the mask application interface that provides a universal interface for data storage. In essence, the mask application interface provides the following:

•The acquisition of a mask schema;

•Querying via a masked query;

•Receiving masked results.

A well-designed mask module allows developers to treat it as a simple native data provider without the need for additional transformations. Of course, the achievement of such a property is also dependent on the developer’s ability to provide schema, query, and result translator implementations fitting well with the implemented mask application.

If such design generalizations were not considered, the development of each kind of mask component would create a lot of excess repeated work and increase the overhead workload, as all aspects of a mask would need to be re-implemented and retested. Such development would also have an impact on the management of multiple mask-kind codebases, as none would conform to any design standard.

Keeping in form with the proposed design, the development of a mask-type component is narrowed down to the implementation of just four components:

•Schema translator implementation;

•Query translator implementation;

•Result translator implementation;

•Mask application.

This obviously reduces the workload and time required to implement a mask component, removing the need for the re-implementation of core components. The development following

the proposed design allows logical layering of the mask component in the segment of the mask application, as the mask module can be treated as a provider or service. Such standardization allows the mask components to be potentially built, tested, and maintained by a community of developers in the form of an open-source software initiative.

These observations implicitly consider the mask to be used as a prefabricated and configurable component - generically developed once for a required format, with the ability to be reused in multiple (MMW) systems.

5.2.1 Data translation

A data source integration system's functionalities don't explicitly end with querying. A data source integration system can additionally facilitate the mutation of data in the data sources. Consequently, a data source integration system doesn't just support querying but also commanding. The *commanding* nomenclature is taken from the command-query responsibility segregation principle [128].

With the addition of commanding, the system's data model is supposed to serve as both results and data inputs. This change trivially affects the mask component by producing the requirement for a command translator. A more consequential effect of this is the arising requirement for a translator to enable data to be transformed to and from the masked and system format with a certain level of correctness. The data translator has to output masked data that is equivalent to the inputted system data, and also be able to output system data that is equivalent to the inputted mask data. The conclusion is that a data translator has to facilitate two-way transformations if the data source integration system is to correctly provide both commanding and querying. Such transformations are the topic of research in the field of bidirectionalisation.

5.3 Mask framework

An implementation framework is the most feasible way to facilitate the uniform and cognitively simplest implementation of masks. Restrictions via typing of components can be proscribed to prevent developers from constructing ill-formed components as much as possible and to guide the development process itself. The development process, as will be shown in Section 6.1, can be described in general guiding steps.

Figure 5.4 illustrates the relationship between elements involved in the mask development process. A framework with generic typing and pre-implemented components is used to create a mask library. The mask library is a library of a specific mask kind. It is created as a physical packaging of the mask module (from Figure 5.3) with concrete implementations assigned to the proposed interfaces. The mask library contains the implementations of the concrete command,

query and schema managers, as well as the command, query, schema and data translator implementations as classes. The library also prescribes the use of a communication node for masks. The mask type is determined by the mask application which is built using the library. The mask application is an instantiable mask component that uses a specific mask library. Multiple mask types can be created from a single mask (kind) library. The mask application, with its enclosed mask library, effectively forms a mask component. The mask component is used by end-users to access the system in the format of the implemented kind. A point to note is that end-users here are considered as persons or systems that consume the masked data through a masked schema using masked queries (or commands for data mutation).

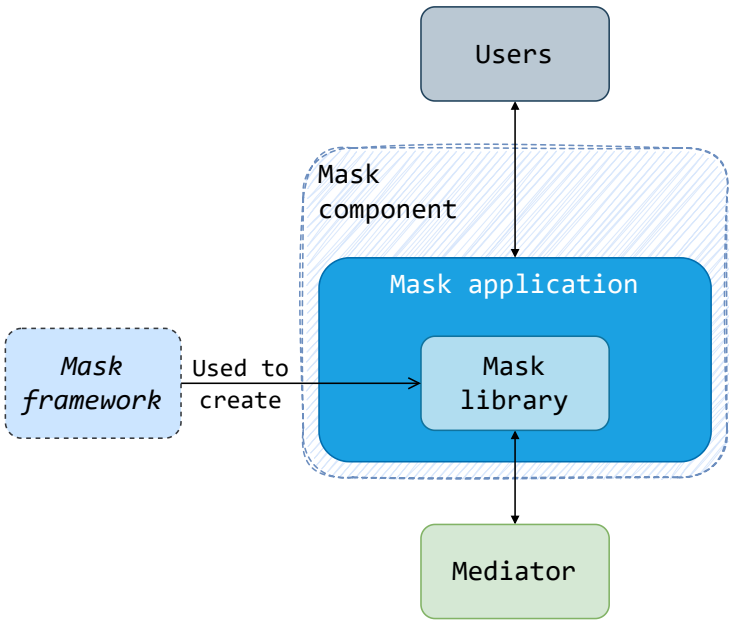
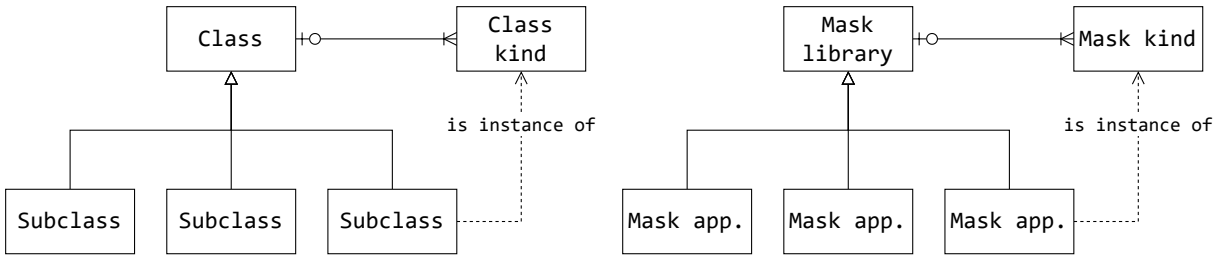


Figure 5.4: Mask development concept

It is important to note that once a mask kind and its corresponding type are implemented to create a mask component, they do not need to be reimplemented for use in other projects. The mask components are introduced to systems as prefabricated components that only need to be configured to function properly within the MMW topology at hand. This brings into light users who will be tasked with configuring and managing the deployed mask components - these are to be considered as a separate group from the end-users.

The difference between a mask kind and type here is most apparent. The nomenclature of kind and type is taken from Gonzalez-Perez and Henderson-Sellers[55, 56] and their review of kind, types, and powertypes in metamodelling (Figure 5.5). A mask kind is defined by the representation its library supports (e.g. Web API, JDBC, graph data), while a mask type is defined by the application which encases the library (e.g. web server, CLI daemon, desktop application). Implementing a mask kind creates a powertype, whose concretisation as an application infers a type.



(a) Generic powertype example (b) Mask powertype

Figure 5.5: Powertype examples for a generic case and masks

5.4 Mask modalities

Virtualising and materialising data integration systems were mentioned in Section 2.2.2 as being two groups based on the integration system's mode of use. The mask framework and component, as generically described in this chapter, support the construction of both *virtualising and materialising masks*. This capability will be presented in Section 6.1 with prototypal masks from both groups. Since a hypothetical MMW system facilitates multiple masks, an MMW system can be a virtualising and materialising system at the same time. To put these statements succinctly:

- A *virtualising mask* manages a virtualised representation of the system in terms of a unified view of a certain mask kind.
- A *materialising mask* produces a materialised representation of the system in terms of a generated data store of a certain mask kind*.

*It will be demonstrated in the case studies and their respective prototypes (Chapter 8) how the materialising of masks can be used to create temporary localised stores or to create a physical export of the data.

Chapter 6

Prototype system

This chapter concerns the implementation of an MMW system prototype, called *Janus*. Specifically, Section 6.2 introduces the third contribution of the research covered by this thesis - a framework for implementing masks. The contribution is enhanced by substituting the hypothetical system with the prototype Janus system. Section 6.2 also includes the steps taken to implement a Web REST API mask by using the mask framework and a succinct overview of two materializing masks that were implemented for further experimentation and prototyping.

6.1 Janus system

Research cited in this thesis has yet to produce a concrete, comprehensive, freely usable and open-ended data source integration system. This problem is best summarized by Golshan et al. [13]:

> *"...it is time for data integration operators to break free of end-to-end data integration systems and be available in the open source to speed up adoption and progress."*
>
> *"The first challenge [...] is that progress of data integration and its application in practice are hindered by the fact that there are very few quality tools with which practitioners and researchers can freely experiment."*

To provide a solution for future scientific contributions and to concretise this thesis' contribution of a mask framework implementation, the framework is not only developed for use in a hypothetical system but a genuine (although prototype) system. The development of a prototype system also allows reasoning about concrete system elements which are translated, making the mask framework implementation more candid.

The implemented prototype of a heterogeneous data source integration system is called *Janus* [27]. The system is named after the ancient Roman god Janus Bifrons, the god of

beginnings, gates, transitions, time, duality, doorways, passages, frames, and endings. This symbolises the intentions of this and future research in terms of data management and bidirectionalisation.

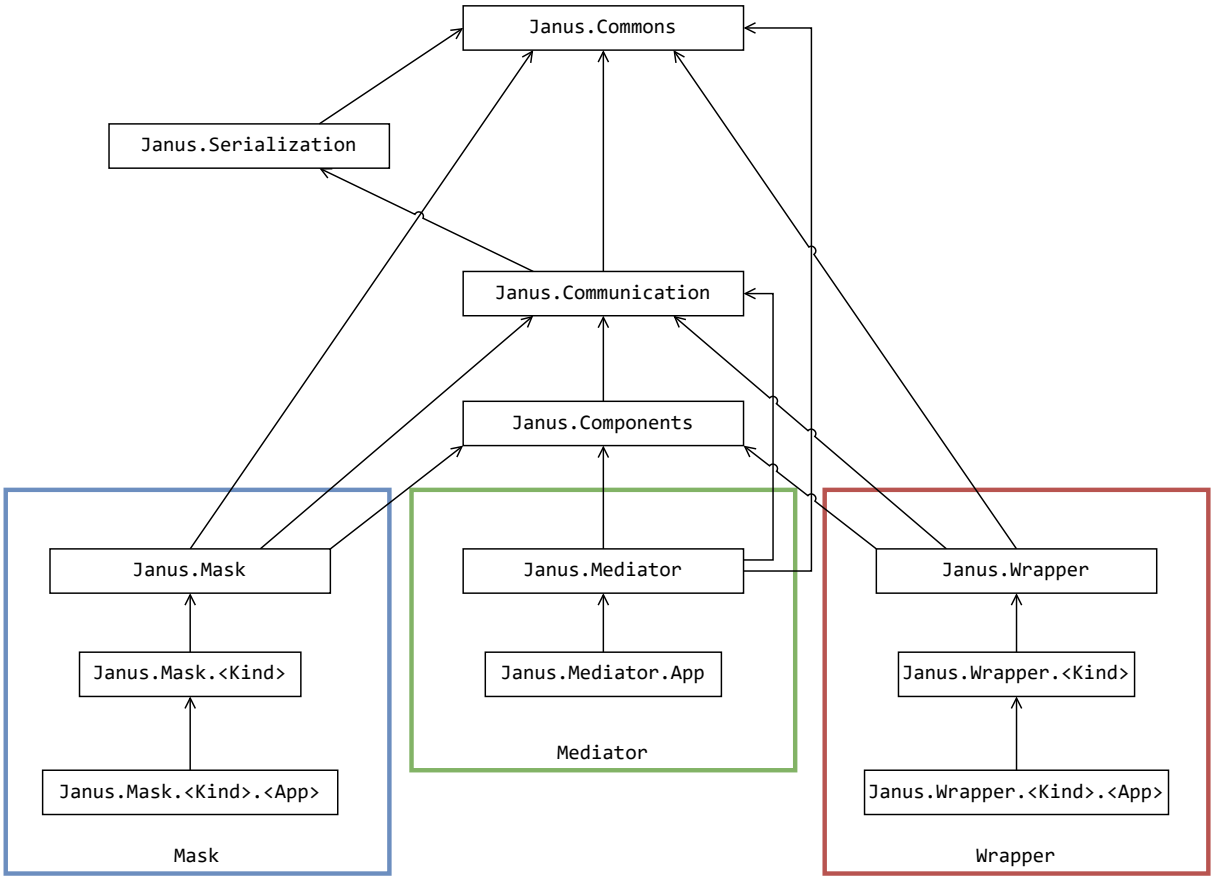


Figure 6.1: Solution structure of the Janus system with key project dependencies (coloured rectangles outline the projects used exclusively by individual component types; red - wrapper, green - mediator; blue - mask)

Janus is implemented using C# in .NET 6. The conceptual solution structure is shown in Figure 6.1, where each rectangle represents a C# project in a .NET solution. The `Janus.Commons` project defines the common Janus system models regarding schema, data, queries, and commands; these definitions are in the system format. The schema model is presented in Section 6.1.1, the data model in Section 6.1.2, the query model in Section 6.1.3, and the command model in Section 6.1.4. The `Janus.Commons` project also contains definitions of communication messages for communication between system components (Section 6.1.5).

`Janus.Serialization` contains all the serialization interfaces required for implementing serialization functionalities for data formats in Janus. Each supported data format is implemented in a separate project. `Janus.Communication` contains definitions of the communication capabilities of Janus (Section 6.1.5) as communication nodes and network adapters.

`Janus.Components` contains the core interfaces that all Janus MMW components should adhere to. `Janus.Mask`, `Janus.Mediator`, and `Janus.Wrapper` contain core component def-

initions implemented from interfaces of `Janus.Components`. The `Janus.Mediator` is directily used by the (in Figure 6.1 hypothetical) `Janus.Mediator.App` to construct a deployable and runnable component. This is because the mediator has no additional definitions of kinds. `Janus.Wrapper` is the core framework for the wrapper component type, and it is used to define specific wrapper kinds for each type of data source. The wrapper kind library projects are then used to construct deployable and runnable wrapper components.

The mask framework is defined as a collection of interfaces in the `Janus.Mask` project. The definitions in `Janus.Mask` are used to construct separate mask kind library projects for specific representations. Each implemented mask kind library is named with the following pattern: `Janus.Mask.<Kind>`. The mask kind libraries are then used to implement concrete mask applications in their own projects. The creation of a mask application implies that a mask type has been created. The mask application projects are named with the following pattern: `Janus.Mask.<Kind>.<Application>`; where the "Application" placeholder is used to indicate what application is implemented in the project (e.g. Web application, Web API, desktop application, CLI application). The application project for a mask kind must produce an executable component that allows the mask component to be deployed and run.

It is interesting to note that, in accordance with the concept presented by Figure 4.2, the mask's project structure mirrors that of the wrapper.

The compilation outputs of the individual projects are physical components in the form of DLL files. The component application projects output executable components, while the library projects output library components. The components created on a solution-wide build of the current Janus codebase and their dependencies are shown in Figure 6.2.

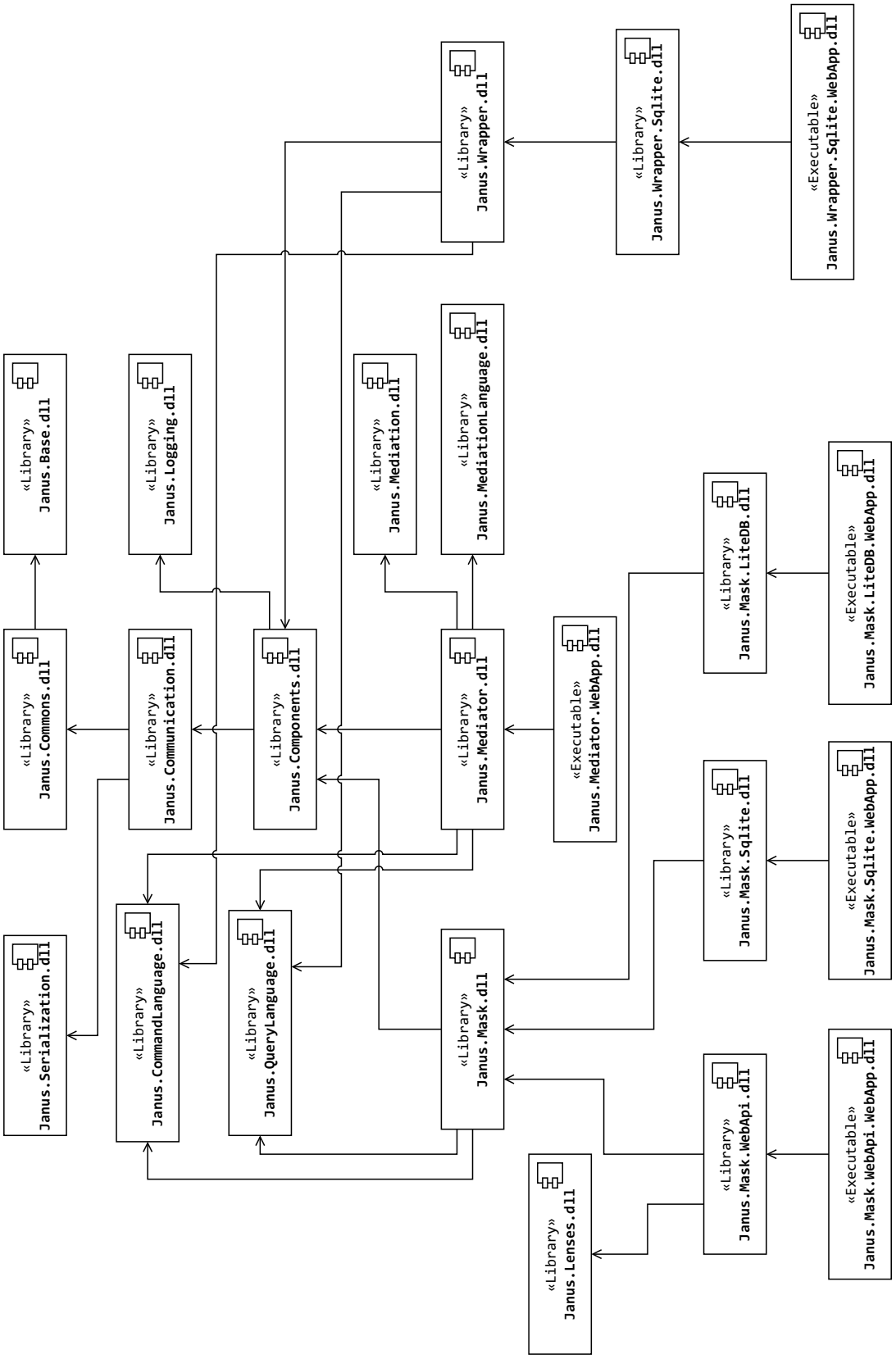


Figure 6.2: Janus system components and their dependencies (dependencies are depicted transitively; individual serialization libraries are omitted)

6.1.1 Janus schema model

The schema model in Janus takes inspiration from a statement by Kleppman [129] that:

> *"...relational databases turned out to generalize very well, beyond their original scope of business data processing, to a broad variety of use cases."*

The Janus system's schema model is illustrated by a class diagram in Figure 6.3.

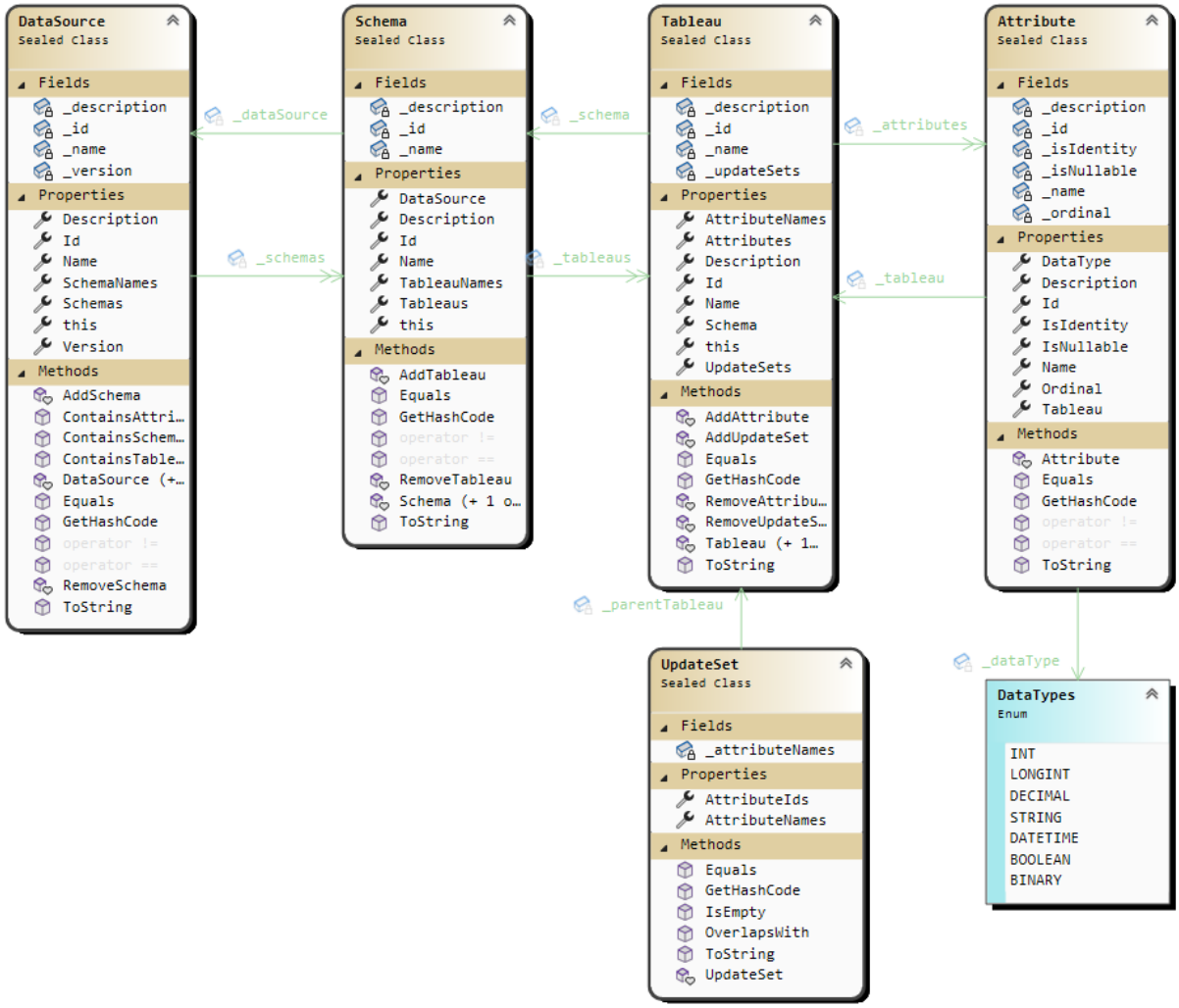


Figure 6.3: Janus schema model

The schema model specifies a singular root element - the *data source*, defined by the `DataSource` class. The `DataSource` class contains metadata about the data source it represents - the data source name, version, textual description and identifier. The `DataSource` doesn't just represent the schema of a wrapped data source but can also represent a virtualized view; e.g. a schema generated in a mediator or loaded in a mask. The data source can contain schemas. The *schema* construct is defined by the `Schema` class and is related to collections of data storage units in a data source; e.g. schema in a relational database. The schema contains an identifier, name, and textual description. The schema can contain tableaus. A *tableau*, as defined by the

`Tableau` class, is a tabular data descriptor related to data unit structures; such as relational tables or document collections for example. The name "tableau" is chosen to denote the tabular nature of the structure, but not to cause confusion when dealing with tables. Additionally, a notable difference from tables is that tableaus are not marked as being in relationships with other tableaus. In the Janus schema model, relationships are ignored. A tableau contains an identifier, name, and textual description. A tableau consists of attributes. *Attributes*, represented by the `Attribute` class, relate to relational attributes or document keys. An attribute contains its identifier, name, textual description, and annotations on whether it represents nullable values, identity values, its preferred ordinality in the attribute set, and data type. Janus currently supports the data types represented in Table 6.1

Table 6.1: Janus supported data types and their mappings to C# .NET types

Janus data type	**C# .NET type**
INT	`int`
LONGINT	`long`
DECIMAL	`double`
STRING	`string`
DATETIME	`DateTime`
BOOLEAN	`bool`
BINARY	`byte[]`

The identifiers for the schema model elements are generated from the elements' names and their parent element's identifier. A data source doesn't have a parent element, as it is the root element, and its identifier is equivalent to its name. A schema's identifier is formed from its parent data source identifier and its own name. A tableaus identifier is formed from its parent schema identifier and its own name. An attributes identifier is formed from its parent tableau and its own name. This helps to determine an element's affiliation with other elements. A dot-delimited representation is used for identifiers, as exemplified in Table 6.2.

Table 6.2: Identifier examples for schema model elements

Element type	**Element name**	**Identifier**
Data source	`ShopDb`	`ShopDb`
Schema	`main`	`ShopDb.main`
Tableau	`Items`	`ShopDb.main.Items`
Attribute	`Price`	`ShopDb.main.Items.Price`

Since some data sources might not represent a physical data source, but rather a virtual one, the update set mechanism is used to denote which attributes can be used in a command (see Section 6.1.4 for commands). Attributes in an update set are expected to come from a single physical data source. More on update sets is given in Section 6.1.7.

The construction of a valid schema model instance is supported by the `SchemaModelBuilder` class, which provides a series of construction declarations.

6.1.2 Janus data model

The Janus data model (Figure 6.4) reflects that of the schema model, so tabular data is described as if it originated from a tableau. The `TabularData` class describes this data in terms of a possible name and data types of columns. The name of the tabular data is usually used to relate it to a query as a result. The tabular data's schema is simply seen as containing columns, where only the expected data type is noted. To ease the manipulation of individual data instances, tabular data is defined as containing rows. Rows are defined by a `DataRow` class, where values are kept for each column defined in a `TabularData` instance.

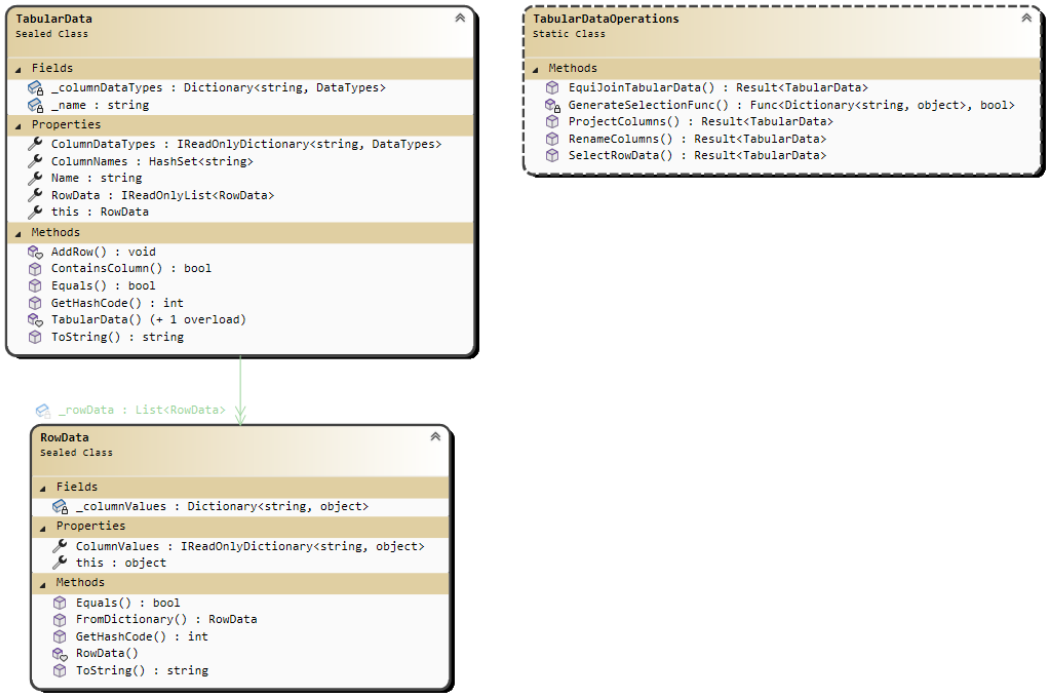


Figure 6.4: Janus data model

Along with the data model, the Janus codebase offers a collection of operations over tabular data: selection, projection, equi-join, and column rename. These can be found as extension methods in the `TabularDataOperations` class.

Construction of a valid `TabularData` instance is enabled through the `TabularDataBuilder` class, which provides a series of construction declarations.

6.1.3 Janus query model

The Janus query model is defined as having three clauses: joining, selection, and projection. None of these clauses is obligatory in a query, but a query must define an initial tableau. In the absence of the three clauses, The initial tableau's data is fully selected and projected onto the query results. The query model supports only natural equi-join. The query model is illustrated in Figure 6.5 as a class diagram.

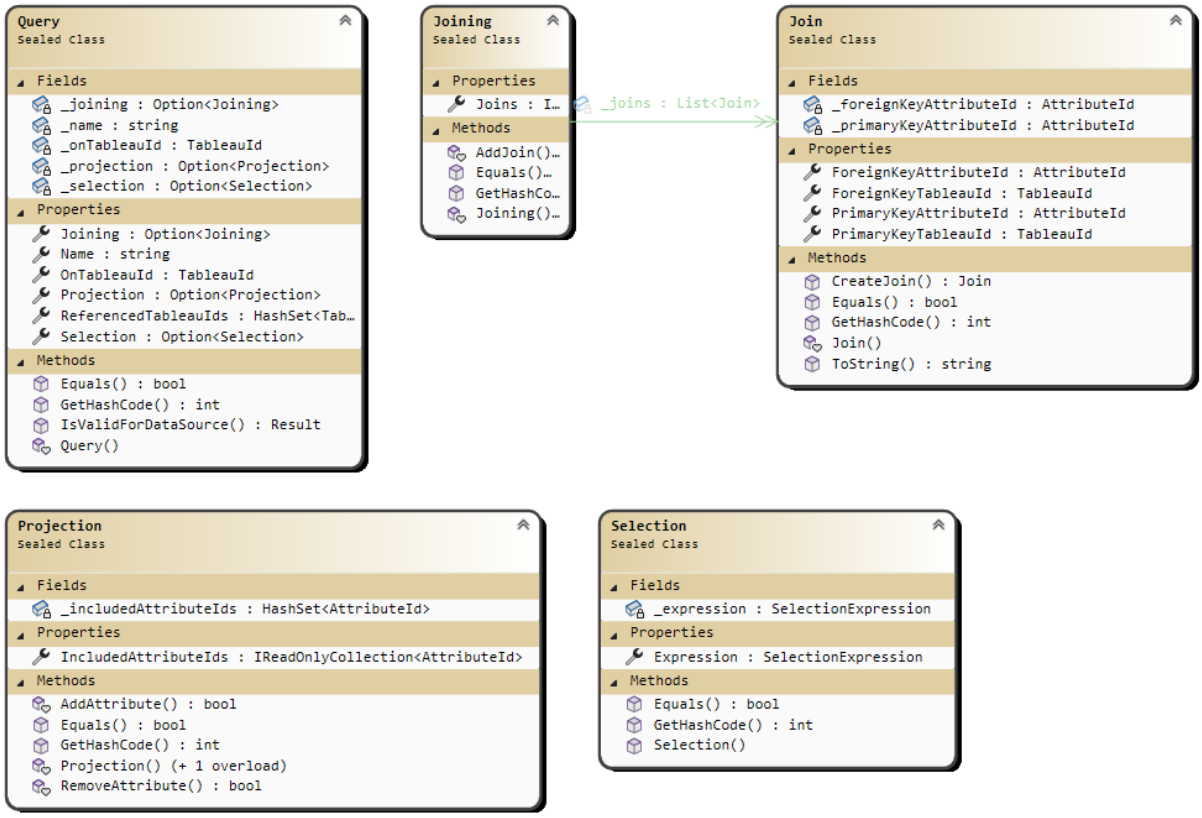


Figure 6.5: Janus query model

A Janus system query is defined by the `Query` class. `Query` contains the optional clause definitions, the initial tableau identifier, and a method to check a query's validity over a data source's schema model. Projection is defined by the `Projection` class, which contains a set of all the projected attributes' identifiers. The joining clause is defined by the `Joining` class, which contains a list of `Join` objects representing joins. A join is defined to contain a primary and foreign key attribute identifier. The foreign key is considered the attribute from the left operand tableau, and the primary key is considered the attribute from the right operand tableau. The selection clause is defined by the `Selection` class, which holds the selection expression. Selection expressions are defined by the selection expression model shown in Figure 6.6.

Query creation is supported by the `QueryModelBuilder` and `QueryModelOpenBuilder` classes which provide a declarative construction. The regular query builder enables the construction of a valid query over a given data source, while the open builder provides the building

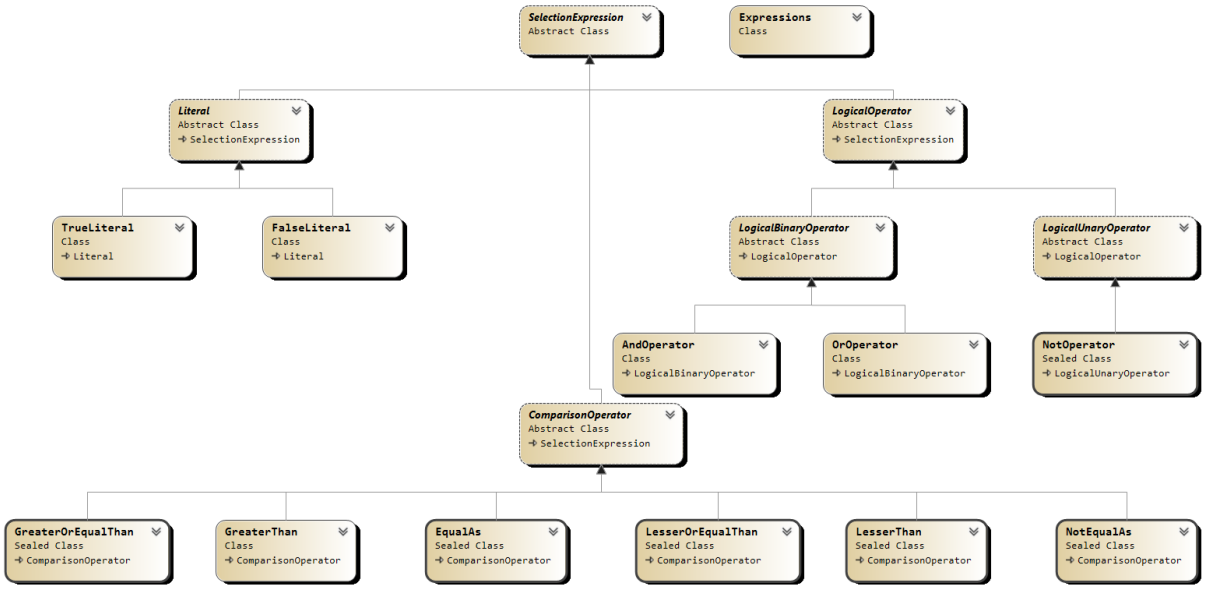


Figure 6.6: Janus selection expression model

of a valid query without a target data source. Queries built using the `QueryModelOpenBuilder` must be additionally validated over their intended target data sources; the `IsValidForDataSource` method is used in that case.

Janus supports an SQL-like query language defined in ANTLR4 [130] which enables the construction of textual queries in user interfaces for component management. Queries can reference one data source at a time. Queries expressed in this way are created in the system format. An example of a textual query is given in Listing 6.1 (the query can be run on the `TracksData` data source from Section 8.3.1).

```
SELECT*
FROMTracksData.main.tracks
JOINTracksData.main.genres
    ONTracksData.main.tracks.GenreId == TracksData.main.genres.
    ↪ GenreId
JOINTracksData.main.media_types
    ONTracksData.main.tracks.MediaTypeId == TracksData.main.
    ↪ media_types.MediaTypeId
WHERETracksData.main.genres.Name ="Soundtrack";
```

Listing 6.1: Query language example

6.1.4 Janus command model

The Janus system has a command model defined for source data manipulation. Three command types are supported: *delete*, *insert*, and *update*. The command model classes are shown in Figure 6.7.

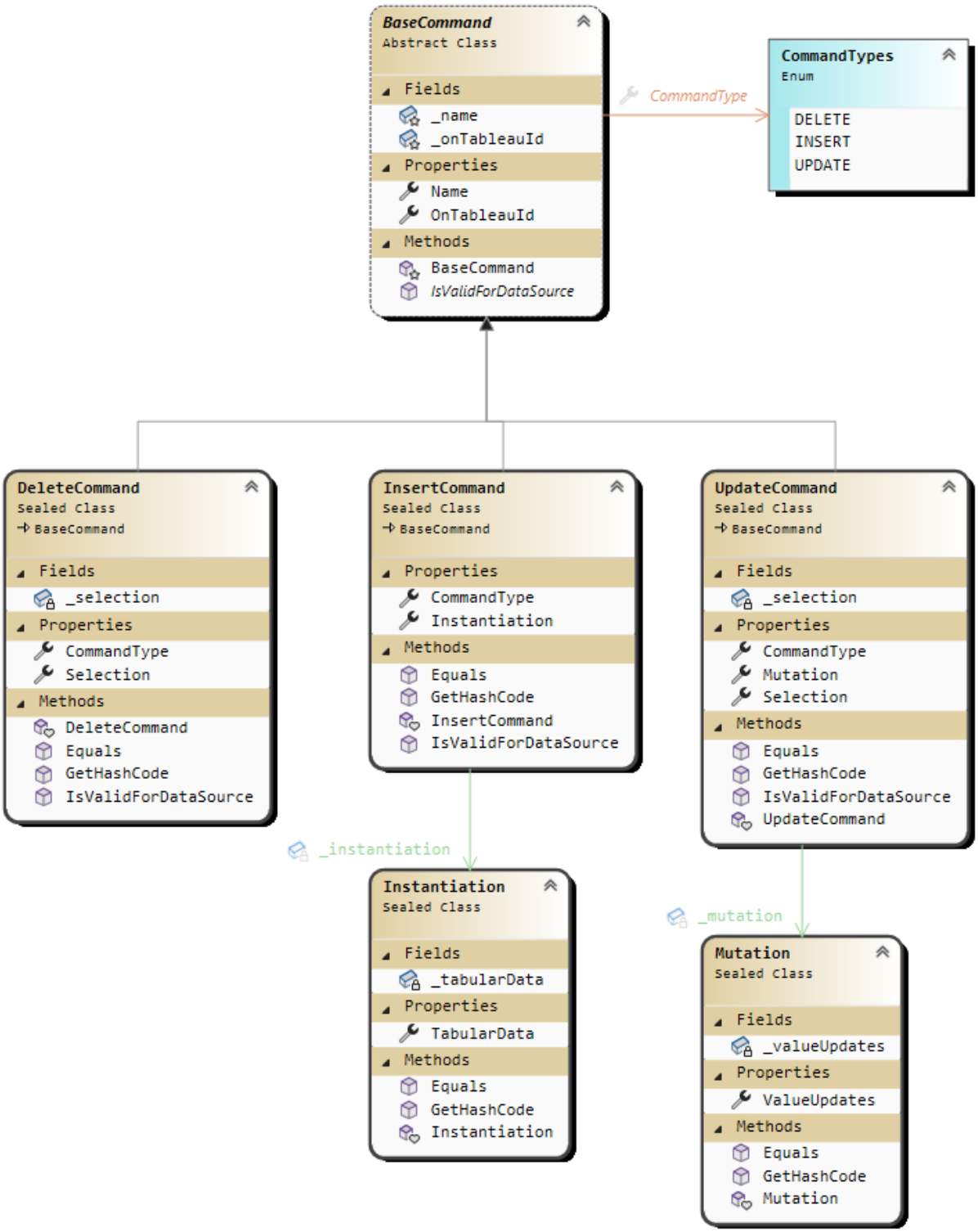


Figure 6.7: Janus command model

The delete command is used to delete data from a data source. The delete command contains a tableau targeting clause and an optional selection clause. The optionality of the selection clause is modelled so that the `None` is effectively a logical `false`. This is to prevent unintentional deletion of tableau data when the selection clause is accidentally omitted. The delete command is valid and can be run over a tableau that contains an update set containing all of the

tableau's attributes.

The insert command is used to insert data into a data source. The insert command contains a tableau targeting clause and an instantiation clause. The instantiation clause supports declaring multiple tuple values for insertion. The insert command is valid and can be run over a tableau that contains an update set containing all of the tableau's attributes.

The update command is used to update data in a data source. The update command contains a tableau targeting clause, a mutation clause, and an optional selection clause. The update command is valid and can be run over attributes from a single update set; this includes the attributes referenced in the selection clause.

Command creation is supported by the `DeleteCommandBuilder`, `InsertCommandBuilder`, and `UpdateCommandBuilder` classes which provide a declarative construction. These builders construct commands over a specified target data source. Each contains an open builder counterpart: `DeleteCommandOpenBuilder`, `InsertCommandOpenBuilder`, and `UpdateCommandOpenBuilder`. The open builders don't guarantee validity over a data source, hence the commands built by them must be additionally validated over their intended target data sources; the `IsValidForDataSource` method is used in those cases.

Janus supports an SQL-like command language defined in ANTLR4 [130] which enables the construction of textual commands in user interfaces for component management. Commands expressed in this way are created in the system format. Examples of textual commands are given in Listing 6.2 (these commands can be run on the data sources created in the prototype of Section 8.3.1).

```
/*DELETEcommand*/
DELETE
FROMUsersInvoices.main.customers
WHEREUsersInvoices.main.customers.Country =="USA"ANDUsersInvoices
    ↪ .main.customers.Company =="TradesInc.";
/*INSERTcommand*/
INSERT
INTOInvoicingData.Main.Users(UserFirstName, UserLastName, UserEmail)
VALUES("John","Doe","johnnyd@gmail.com");
/*UPDATEcommand*/
UPDATEUsersInvoices.main.employees
WITHTitle <-"Seniorprogrammer", RepotsTo <- 6L, Phone <-"+1(403)
    ↪  246-9882"
WHERE(UsersInvoices.main.employees.LastName =="King"AND
    ↪ UsersInvoices.main.employees.FirstName =="Robert")ORFALSE;
```

Listing 6.2: Command language examples for delete, insert, and update commands

6.1.5 Janus communication

The Janus system's communication is defined by communication nodes, network adapters, and messages. The Janus message types are listed in Table 6.3. The request message type names are denoted with the `_REQ` suffix, and the analogous response type names are denoted with the `_RES` suffix. All response messages contain information pertaining to the outcome of the operation initiated by their request; be it a successful or failing outcome.

To support evolvability, communication in Janus is implemented by separating the concerns of logical communication behaviour from the technological way in which communication is achieved. The logical aspect of messaging is defined in the classes inheriting the `CommunicationNode` class (Figure 6.8). A concrete communication node class is defined per MMW component type.

Table 6.3: Message types available in the Janus system

Message type	Description
`HELLO_REQ`	Used to test a connection to a remote node or to establish a connection with a remote node where both nodes are expected to register one another.
`HELLO_RES`	Used to confirm the acceptance of a connection request by the remote node. If the request was also used to request a connection registration, then the response also signifies that the registration on the remote node is successful.
`BYE_REQ`	Used to request deregistration of a (requesting node's) connection on a remote node.
`SCHEMA_REQ`	Used to request a schema from a component.
`SCHEMA_RES`	Used to respond to a schema request and transfer the schema of a remote component to the requesting component.
`QUERY_REQ`	Used to request a query run from a remote component.
`QUERY_RES`	Used to respond to a query request with an outcome of a query run; an outcome message and an optional query result (as tabular data).
`COMMAND_REQ`	Used to request a command run from a remote component.
`COMMAND_RES`	Used to respond to a command request with an outcome of a command run.

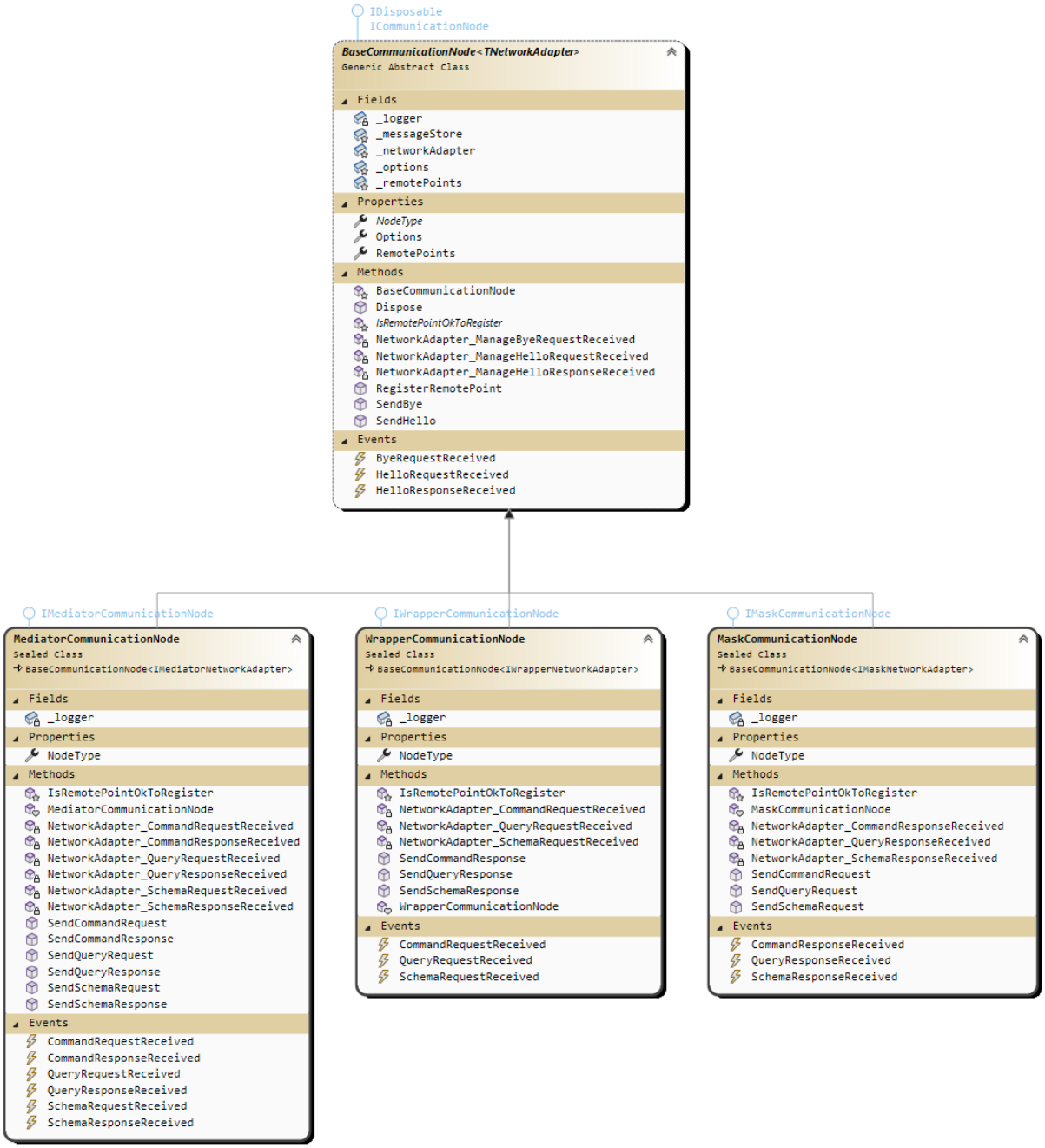


Figure 6.8: Janus communication node classes

The communication node type interfaces are parametrized by the network adapter type they support. Network adapters classes contain technological concerns of communication. Currently, only network adapters supporting TCP are implemented. An adapter is implemented per MMW component type. The available network adapter implementations are shown in Figure 6.9. Additional network adapters can be implemented, e.g. using HTTP or message queues.

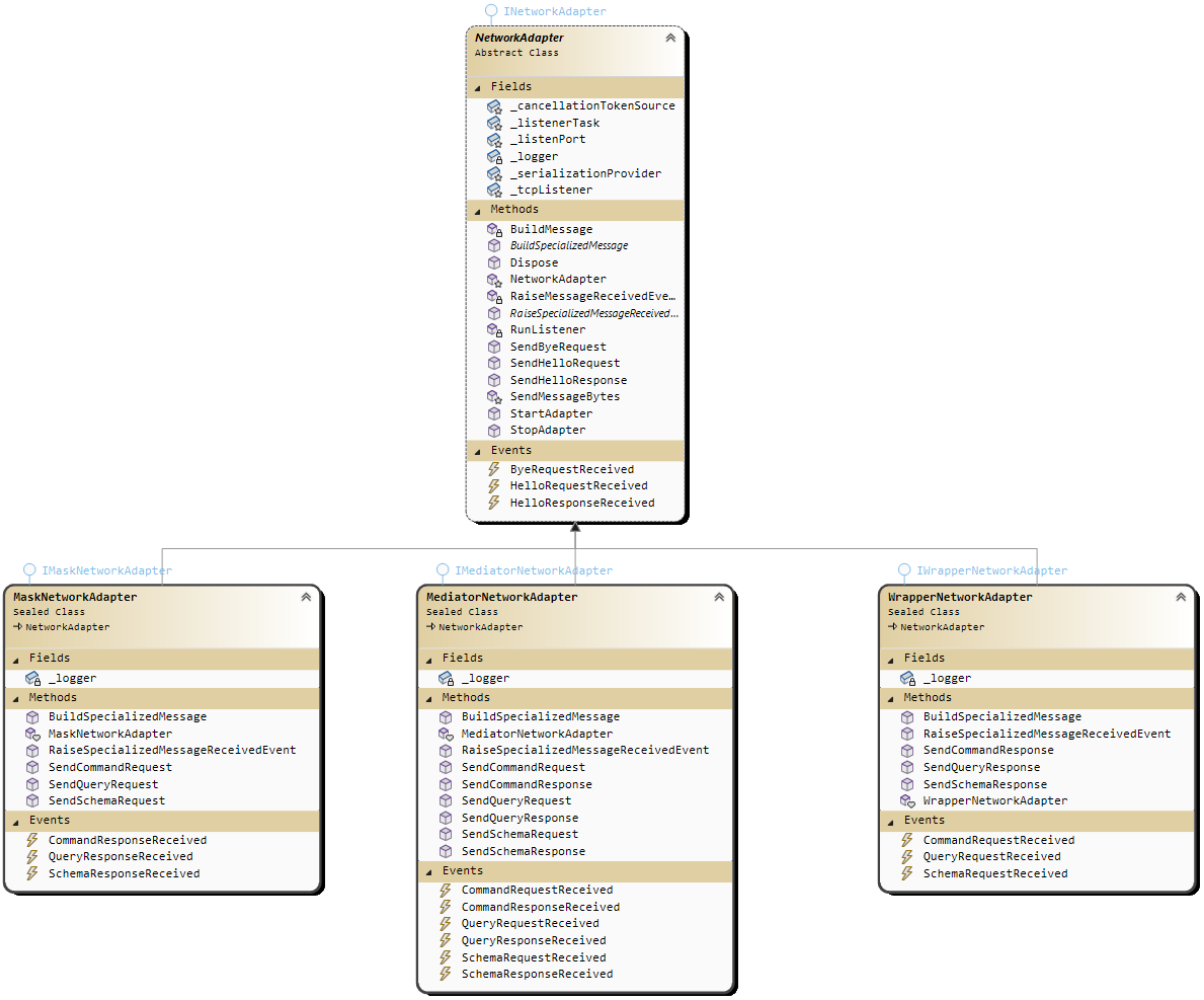


Figure 6.9: Janus TCP adapter classes

Additional network adapters could hypothetically work with a variety of data formats, so a flexible data serialization codebase is required. In the case of the TCP network adapters, the messages are expected to be transferred in binary formats. To facilitate the serialization of messages into binary data and demonstrate the flexibility of implementing new data format types, the Janus codebase contains serialization operation interfaces. These interfaces are implemented for the following binary formats: BSON, MongoBSON, Avro, and Protocol Buffers (Protobufs). These formats can be used interchangeably with the TCP network adapter. Additionally, the textual JSON format is supported, but it is currently not used for communication purposes.

6.1.6 Janus components

The Janus system contains three types of components: mask, mediator, and wrapper. Each of these component types needs to implement interfaces from the `Janus.Components` namespace. The interfaces provide a framework upon which system components are implemented. The

common component interfaces are:

- `IComponentOptions` describes the minimum required options for a system component. This includes the node identifier, listening port, timeout in milliseconds, communication format, network adapter type, startup remote points, and a persistence connection string.
- Schema management interfaces:
  - `IComponentSchemaManager` describes the base functionality of a schema manager, such as loading a schema and providing it to outer components. **Schema loading** means that a schema is going to be actively used in a component for querying, commanding, or mediation.
  - `IDelegatingSchemaManager` describes the base functionality of a delegating schema manager, which can work over other components' schemas.
  - `IMediatingSchemaManager` describes the base functionality of a mediating schema manager, which performs schema mediation.
- Query management interfaces:
  - `IComponentQueryManager` describes a component query manager that runs queries on its own schema.
  - `IDelegatingQueryManager` describes query manager that can delegate query runs to other components.
- Command management interfaces:
  - `IComponentCommandManager` describes a component command manager that runs commands on its own schema.
  - `IDelegatingCommandManager` describes command manager that can delegate command runs to other components.
- `IComponentManager` describes the base functionalities of a component, such as connecting with other components, acquiring schemas, and running commands and queries. This is the main interface for use in component applications. An implemented component manager is expected to have a schema, query, and command manager, as well as a communication node.
- Persistence interfaces are used to persist known remote points and schemas, as well as other metadata. These interfaces are akin to the repository pattern interfaces. The current common interfaces are:
  - `IPersistence<TId, TModel>` describes a persistence interface generically. All persistence interfaces should inherit from this interface and state their concrete generic types.
  - `IRemotePointPersistence` describes persistence for remote points.
  - Currently, persistence implementations can be found under `Janus.<ComponentType>.Persistence.LiteDB`. Persistence is implemented over

a JSON store file database called *LiteDB*.

•Translation interfaces are used to describe translators for data, schemas, queries, and commands. Since they proscribe a significant number of generics, they should be given façade implementations.

- `ISchemaTranslator<TSchemaSource, TSchemaDestination>` describes a schema translator.
- `IQueryTranslator<*>` describes a query translator. This interface exhaustively requires source and destination types for query clauses.
- `ICommandTranslator<*>` describes a command translator. This interface exhaustively requires source and destination types for command clauses and command types. This interface covers all three command types.
- `IDataTranslator<TSource, TDestination>` describes a data translator. For components specific component types, the data translator usually has its `TSource` or `TDestination` set to `TabularData`. Data translators can be implemented using *lenses* to provide sound bidirectional data transformations.

6.1.7 Janus mediator component

The core mediator components can be found in the `Janus.Mediator` namespace (Figure 6.10). Classes from this namespace rely on interfaces from `Janus.Components`. The core components of a mediator implementation are:

- `MediatorOptions` represents the options of a mediator component. It implements the `IComponentOptions` interface, without additions.
- `MediatorSchemaManager` manages the schema in the mediator, enables schema acquisition from other components, and schema mediation through the `Janus.Mediation` namespace. The `MediatorSchemaManager` implements the `IComponentSchemaManager`, `IDelegatingSchemaManager`, and `IMediatingSchemaManager` interfaces.
- `MediatorQueryManager` manages queries in the mediator, allowing queries over a mediated schema or other components. It implements the `IDelegatingQueryManager` interface. The mediator query manager additionally defines a special method to run queries on a mediated schema: `RunQuery(Query query, MediatorSchemaManager schemaManager)`
- `MediatorCommandManager` manages commands in the mediator, allowing command over a mediated schema or other components. It implements the `IDelegatingCommandManager` interface. The mediator command manager additionally defines a special method to run commands on a mediated schema: `RunCommand(BaseCommand command, MediatorSchemaManager schemaManager)`
- `MediatorManager` is the mediator component manager. It implements the `IComponent-`

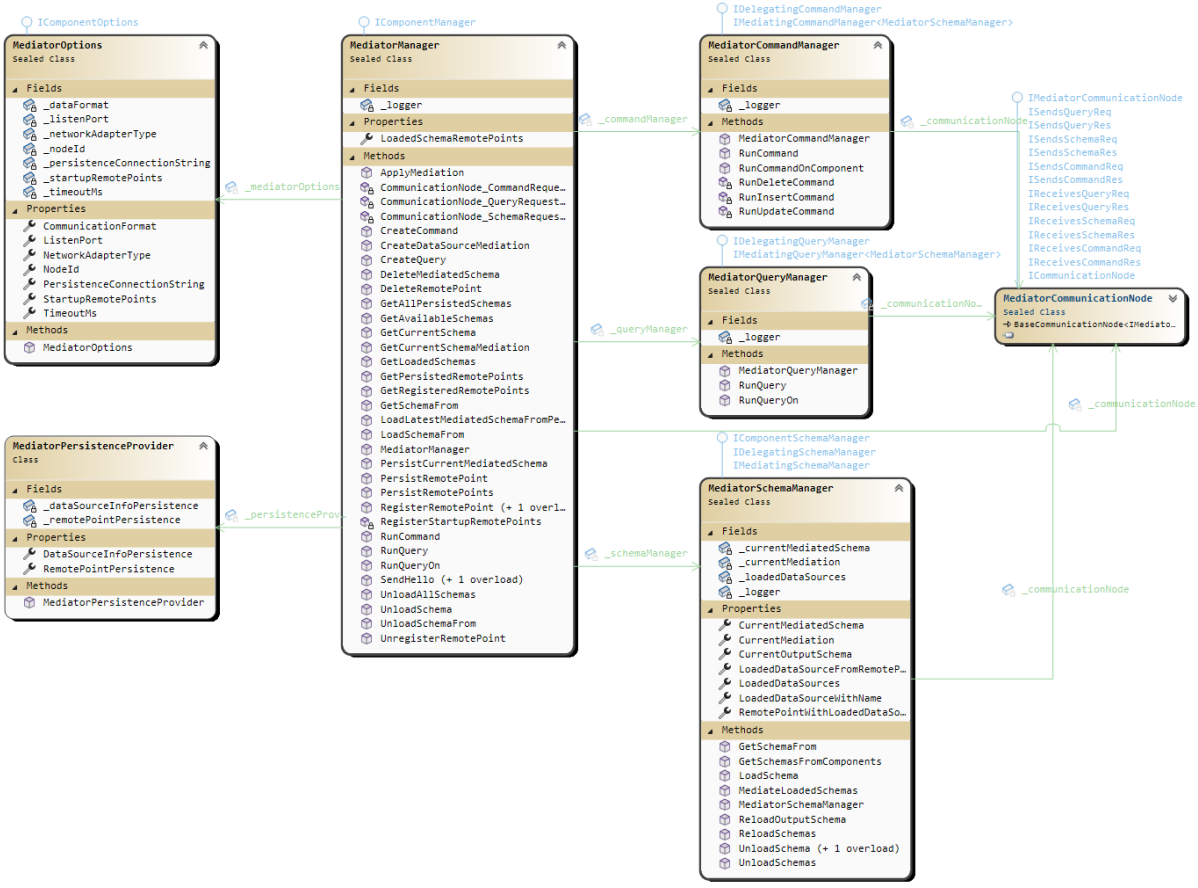


Figure 6.10: Janus mediator classes

`Manager` interface.

- The `MediatorPersistenceProvider` provides the mediator with persistence objects. It can be found under the `Janus.Mediator.Persistence` namespace. Currently, the provider provides persistence for data source information (mediation, loaded schemas for the mediation, and mediated schema) and remote points.

Mediation

The mediator relies on the `Janus.Mediation` namespace for concrete mediation algorithms and data structures that enable it to mediate schemas, queries, commands, and data. All mediation is determined through the specification of the schema model mediation. The `MediationBuilder` class is used as an entry point for schema model mediation construction. It provides the means to declaratively define a schema model mediation. To simplify the declaration of mediation for a management interface, a mediation language has been implemented using ANTLR4. An exemplified mediation script (used to specify mediation in Section 8.3.1) is given in Listing 6.3. The mediation language takes inspiration from the work of Asano et al. [85], where a view is declared, followed by a description of how to populate the view.

A mediation script begins with hierarchical clauses describing the mediated data source,

schemas, and tableaus with their attributes. After each individual tableau's attribute declaration, a description of how to populate the tableau's attributes is given by a source query. A description can be given for each element, enclosed by the '#' symbol, next to their declaration. If attribute description propagation is enabled (as exemplified in line 3 of Listing 6.3), then all attributes without explicitly provided descriptions inherit the descriptions provided by their source attribute. An attribute's data type, identity, and nullability annotations are automatically propagated from the underlying data source. An attribute's identifier is inferred from its declared name and the declared names of its parent elements. The ordinality of an attribute is determined by the position of its declaration in the "WITH ATTRIBUTES" clause.

```
1  SETTING
2      PROPAGATE UPDATE SETS
3      PROPAGATE ATTRIBUTE DESCRIPTIONS
4  DATASOURCEMusicData VERSION "1.0" #Mediated data about music#
5      WITHSCHEMAMain #Default schema#
6          WITHTABLEAUAlbums #Data about albums#
7              WITHATTRIBUTES
8                  AlbumId,
9                  AlbumTitle,
10                 ArtistName
11             BEING
12                 SELECT
13                     AlbumsArtistsData.main.albums.AlbumId,
14                     AlbumsArtistsData.main.albums.Title,
15                     AlbumsArtistsData.main.artists.Name
16                 FROMAlbumsArtistsData.main.albums
17                 JOINAlbumsArtistsData.main.artists
18                     ONAlbumsArtistsData.main.albums.ArtistId ==
   ↪ AlbumsArtistsData.main.artists.ArtistId
19         WITHTABLEAUTracks #Data about tracks and albums#
20             WITHATTRIBUTES
21                 TrackId,
22                 TrackName,
23                 GenreName,
24                 MediaType, #Can be AAC, MP3, FLAC, etc.#
25                 AlbumTitle,
26                 DurationMs,
27                 Composer
28             BEING
29                 SELECT
30                     TracksData.main.tracks.TrackId,
31                     TracksData.main.tracks.Name,
32                     TracksData.main.genres.Name,
33                     TracksData.main.media_types.Name,
```

```
34                  AlbumsArtistsData.main.albums.Title,
35                  TracksData.main.tracks.Milliseconds,
36                  TracksData.main.tracks.Composer
37              FROMTracksData.main.tracks
38              JOINTracksData.main.genres
39                  ONTracksData.main.tracks.GenreId == TracksData.
    ↪ main.genres.GenreId
40              JOINTracksData.main.media_types
41                  ONTracksData.main.tracks.MediaTypeId ==
    ↪ TracksData.main.media_types.MediaTypeId
42              JOINAlbumsArtistsData.main.albums
43                  ONTracksData.main.tracks.AlbumId ==
    ↪ AlbumsArtistsData.main.albums.AlbumId
44      WITHTABLEAUUsers #Data about users/customers#
45          WITHATTRIBUTES
46              UserId,
47              UserEmail,
48              UserFirstName,
49              UserLastName,
50              UserCountry
51          SELECT
52              PlaylistsUsersData.main.customers.CustomerId,
53              PlaylistsUsersData.main.customers.Email,
54              PlaylistsUsersData.main.customers.FirstName,
55              PlaylistsUsersData.main.customers.LastName,
56              PlaylistsUsersData.main.customers.Country
57          FROMPlaylistsUsersData.main.customers
58      WITHTABLEAUPlaylists #Data about playlists#
59          WITHATTRIBUTES
60              PlaylistId,
61              PlaylistName,
62              CreatorEmail
63          BEING
64              SELECT
65                  PlaylistsUsersData.main.playlists.PlaylistId,
66                  PlaylistsUsersData.main.playlists.Name,
67                  PlaylistsUsersData.main.customers.Email
68              FROMPlaylistsUsersData.main.playlists
69              JOINPlaylistsUsersData.main.customers
70                  ONPlaylistsUsersData.main.playlists.CreatorId ==
    ↪  PlaylistsUsersData.main.customers.CustomerId
71      WITHTABLEAUPlaylistTracks #Tracks in playlists#
72          WITHATTRIBUTES
73              TrackId,
74              TrackName,
```

```
75                  TrackGenre,
76                  PlaylistId,
77                  PlaylistName
78              BEING
79                  SELECT
80                      PlaylistsUsersData.main.playlist_track.TrackId,
81                      TracksData.main.tracks.Name,
82                      TracksData.main.genres.Name,
83                      PlaylistsUsersData.main.playlist_track.PlaylistId
    ↪ ,
84                      PlaylistsUsersData.main.playlists.Name
85                  FROMPlaylistsUsersData.main.playlist_track
86                  JOINTracksData.main.tracks
87                      ONPlaylistsUsersData.main.playlist_track.TrackId
    ↪  == TracksData.main.tracks.TrackId
88                  JOINPlaylistsUsersData.main.playlists
89                      ONPlaylistsUsersData.main.playlist_track.
    ↪ PlaylistId == PlaylistsUsersData.main.playlists.PlaylistId
90                  JOINTracksData.main.genres
91                      ONTracksData.main.tracks.GenreId == TracksData.
    ↪ main.genres.GenreId
```

Listing 6.3: Janus mediation language example

The mediation process can also be set up to propagate update sets (Listing 6.3, line 2) so that they remain preserved in the higher levels of the schema hierarchy. Propagation of update sets allows data to be manipulated from mediated schemas. Figure 6.11 demonstrates some situations involving update set propagation to a mediated tableau. The blue, purple, and orange update set columns are partially projected in the mediated tableau, but they remain open to data manipulation through the projected columns. The data sets remain, but they are represented strictly along the lines of their projected columns. A concrete example of update set propagation is given in Section 10.5.1.

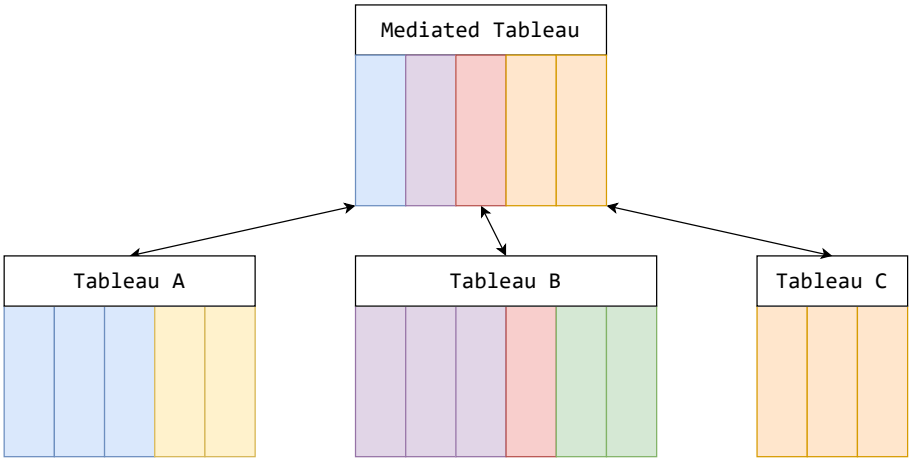


Figure 6.11: Update set mediation

6.1.8 Janus wrapper component

The core wrapper components can be found in the `Janus.Wrapper` namespace (Figure 6.12). Classes from this namespace rely on interfaces from `Janus.Components`. The wrapper components in this namespace are still generic, abstract, and partially implemented. The rest of the implementation depends on a specific data source type - the wrapper kind. The core components of a wrapper at this level are:

- The `WrapperOptions` is an implementation of `IComponentOptions`, with the addition of a data source name specification, command permission, and a data source connection string.
- The `WrapperSchemaManager` is an implementation of `IComponentSchemaManager`. The wrapper's schema manager contains a `SchemaInferrer`, which is used for schema loading in a wrapper.
  - The `SchemaInferrer` is a mechanism by which a schema is inferred from a wrapper's underlying data source. The schema inferrer itself uses a `schemaModelProvider` which is an implementation of the `ISchemaModelProvider` interface.
  - The `ISchemaModelProvider` is used to describe a provider for a specific schema model. The implementations of this interface determine what will be considered a data source, schema, tableau, attribute, and update sets, as well as determining data type mappings in the underlying data source. The provider uses special schema model classes to provide universal information; this can be found under `Janus.Wrapper.SchemaInference.Model`. Each specific data source type - wrapper kind - has to implement its own schema model provider following this interface.
- The `WrapperCommandManager<*>` represents a command manager for a wrapper. At this stage, the command manager is still a generic class with some methods implemented.

To use it for a concrete implementation of a wrapper kind, upon assigning the generic types, create a façade class. This eases type specifications during dependency injection. The command manager uses some generic types that are at this stage only defined as interfaces.

- The `WrapperQueryManager<*>` represents a query manager for a wrapper. At this stage, the query manager is still a generic class with some methods implemented. To use it for a concrete implementation of a wrapper kind, upon assigning the generic types, create a façade class. This eases type specifications during dependency injection. The query manager uses some generic types that are at this stage only defined as interfaces.
- These are the interfaces and types describing local execution, translation and model types:
  - The `IWrapperQueryTranslator` describes a local query translator that translates queries from a system format to a local data source query format. This interface inherits from the `IQueryTranslator<*>` interface from `Janus.Components`. The interface fixes the types on the input side of the translation to the system format.
  - The `IWrapperCommandTranslator` describes a local command translator that translates a command from a system format to a local data source command format. This interface inherits from the `ICommandTranslator` interface from `Janus.Components`. The interfaces fix the types on the input side of the translation to the system format.
  - The `IWrapperDataTranslator` describes a local data translator that translates a local data format to the `TabularData` format. This interface inherits from the `IDataTranslator<*>` from `Janus.Components`. The interface fixes the types on the output side of the translation to the system format.
  - The `IQueryExecutor` describes a local query executor. This is a generic interface, requiring types of local clauses, local data, and local queries. An implementation is recommended to be a façade, to ease dependency injection.
  - The `ICommandExecutor` describes a local command executor. This is a generic interface, requiring types of local clauses, local data, and local delete, insert, and update commands. An implementation is recommended to be a façade, to ease dependency injection.
  - Abstract classes for queries and commands can be found in the `LocalCommanding` and `LocalQuerying` namespaces.
- The `WrapperManager` represents a wrapper component manager. At this stage, the wrapper manager is still a generic class with some methods implemented. To use it for a concrete implementation of a wrapper kind, upon assigning the generic types, create a façade class. This eases type specifications during dependency injection.
- The `WrapperPersistenceProvider` contains persistence functionalities for the wrapper. This includes data source and remote point persistence.

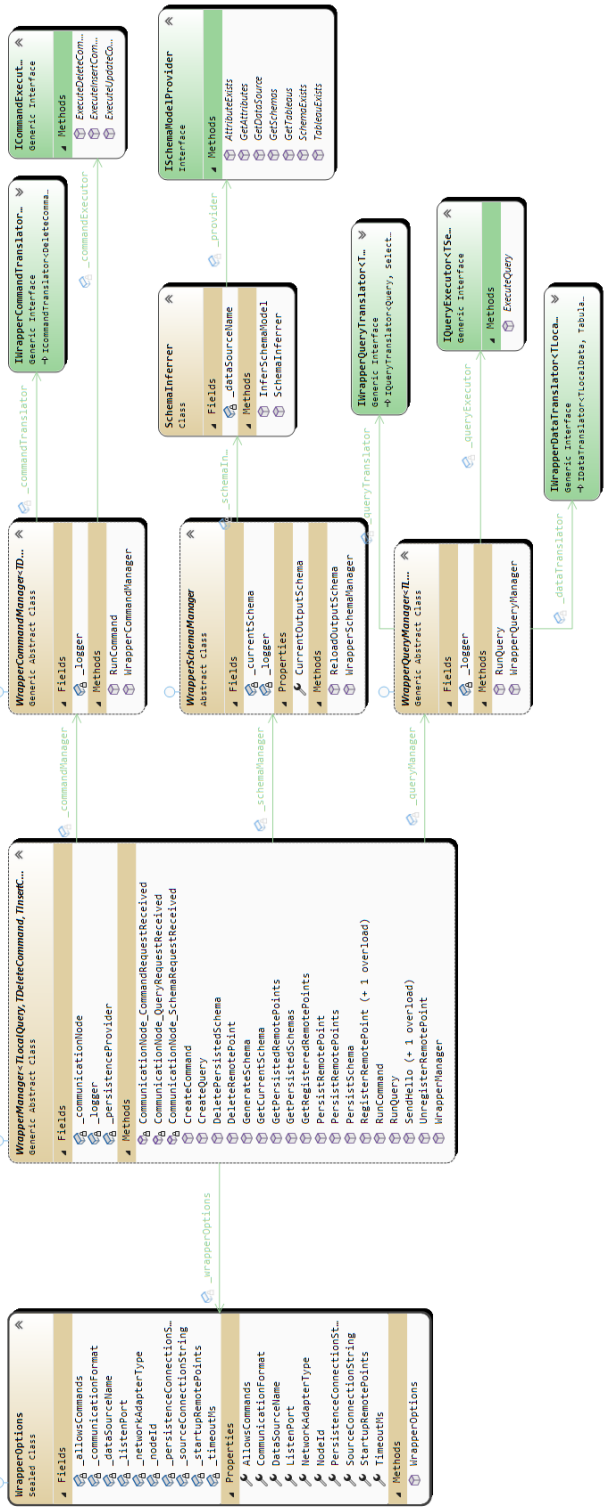


Figure 6.12: Janus wrapper classes

6.2 Proof of concept mask in the Janus system

6.2.1 Mask framework

The mask framework is implemented as the `Janus.Mask` library in the Janus project. The mask framework (Figure 6.13) uses the core Janus component definitions of `Janus.Components` by implementing its interfaces generically. These generic interfaces are declared so that they guide the development of a mask kind; making the developer implement the inner components of a mask in a sequence determined by their type dependencies. For the system-end interactions, generic type specifications are prescribed as the system models for schemas, data, queries and commands. The generic type specifications for masked representation models in those generic classes only prescribe that they should inherit the base masked models. Hence, concrete managers can only be typified if their types are constructed using classes specified as certain masked models. Since the manager and translator inner components are declared as interfaces, they are expected to be implemented as classes in the Janus system.

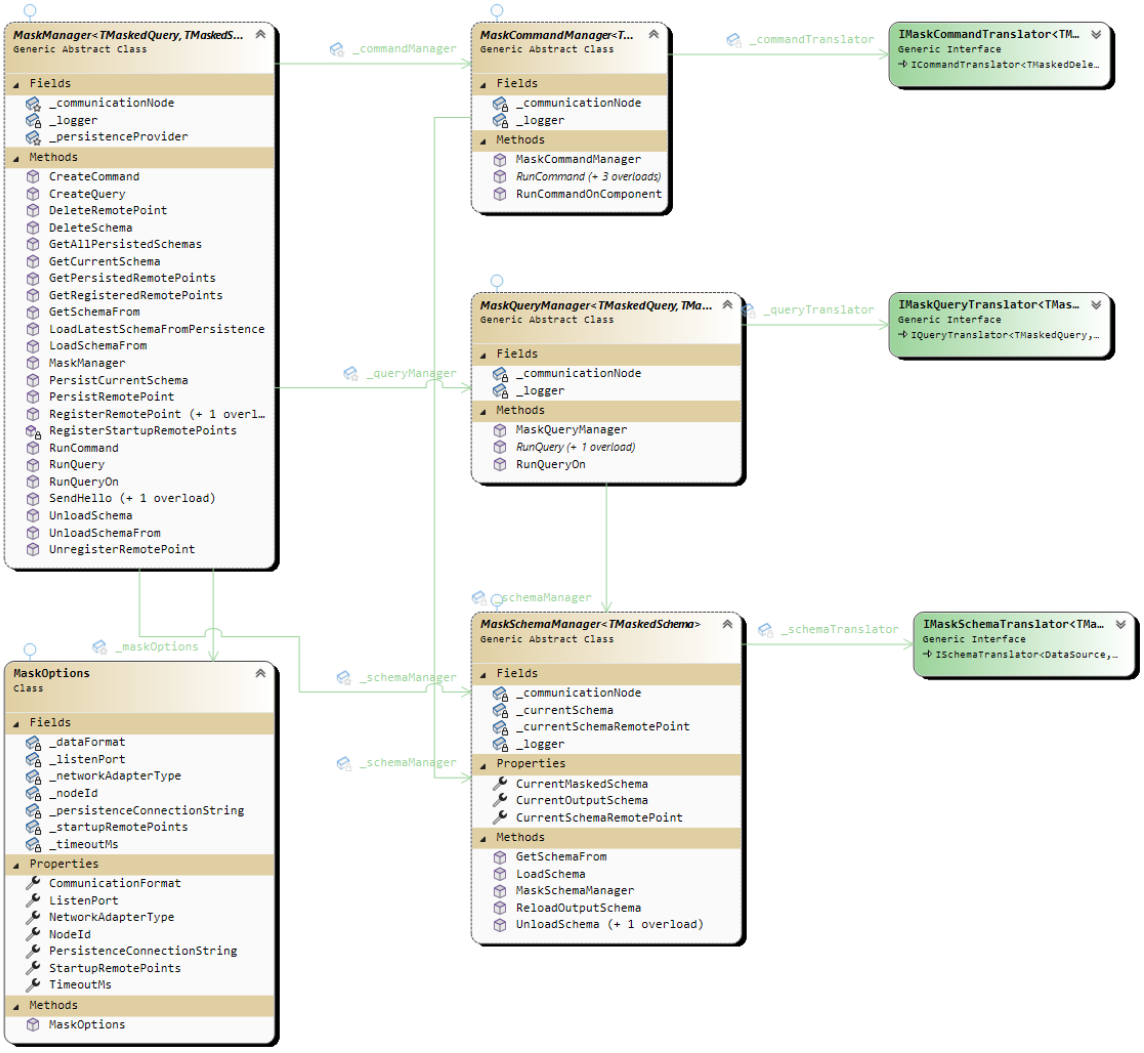


Figure 6.13: Mask framework classes

The mask framework component definitions are as follows:

- `MaskCommandManager<*>` describes a mask command manager. It implements the `IDelegatingCommandManager` interface. The `MaskCommandManager<*>` contains implemented methods to run common model commands on the loaded schema, but it also contains abstract methods that represent the functionality of running commands in the localized command model. This class contains multiple generic type parameters, hence its derivatives should present façades with concrete types.
- `MaskQueryManager<*>` describes a mask query manager. It implements the `IDelegatingQueryManager` interface. The `MaskQueryManager<*>` contains implemented methods to run common model queries on the loaded schema, but it also contains abstract methods that represent the functionality of running queries in the localized query model. This class contains multiple generic type parameters, hence its derivatives should present façades with concrete types.
- `MaskSchemaManager<*>` describes a mask schema manager. It implements the `IDelegatingSchemaManager` interface. This class contains a generic type parameter `TMaskSchema` which represents the type of a masked schema model that a specific mask kind will use. Its derivatives should be implemented as façades.
- `MaskOptions` represents the options of a mask component. It implements the `IComponentOptions` interface, without additions.
- `MaskManager<*>` is the mask component manager. It implements the `IComponentManager` interface. This class contains multiple generic type parameters, hence its derivatives should present façades with concrete types.
- •Translation interfaces can be found in the `Janus.Mask.Translation` namespace. They have fixed source generic type parameters to system models. Specific destination types are to be specified for a mask kind. These are the interfaces for the local (mask) translators that are required for a concrete mask kind implementation:
  - `IMaskCommandTranslator`
  - `IMaskQueryTranslator`
  - `IMaskSchemaTranslator`
  - `IMaskDataTranslator`
- •Abstract local query, command, and data model can be found in the `MaskedQueryModel`, `MaskedCommandModel`, `MaskedSchemaModel`, and `MaskedDataModel` namespaces respectively.
- •Persistence functionality can be found in the `Janus.Mask.Persistence` namespace.

Listing 6.4 shows the type declaration of the generic `MaskQueryManager<*>` class. It is specified that the `TMaskedData` type parameter is required to inherit from the generic `MaskedData<*>` class, the `TMaskedSchema` from `MaskedSchema`, and in a more complex specification,

the `TMaskedQuery` must inherit from a generic `MaskedQuery<*>` class with concrete clause types. The mask manager component is even more complex in its type specification than the query manager (Listing 6.5), because it requires concrete specifications of all masked models.

```
publicabstractclass
MaskQueryManager<TMaskedQuery,
                 TMaskedStartingWith,TMaskedSelection,
                 TMaskedJoining,TMaskedProjection,
                 TMaskedSchema,
                 TMaskedData,TMaskedDataItem>
    :IDelegatingQueryManager
    whereTMaskedQuery:MaskedQuery<TMaskedStartingWith,
                                  TMaskedSelection,
                                  TMaskedJoining,
                                  TMaskedProjection>
    whereTMaskedSchema:MaskedDataSource
    whereTMaskedData:MaskedData<TMaskedDataItem>
```

Listing 6.4: Janus mask framework query manager definition

```
publicabstractclass
MaskManager<TMaskedQuery,TMaskedStartingWith,TMaskedSelection,
            TMaskedJoining,TMaskedProjection,
            TMaskedDeleteCommand,TMaskedInsertCommand,
            TMaskedUpdateCommand,
            TMaskedMutation,TMaskedInstantiation,
            TMaskedSchema,TMaskedData,TMaskedDataItem>
    :IComponentManager
    whereTMaskedQuery:MaskedQuery<TMaskedStartingWith,
                                  TMaskedSelection,
                                  TMaskedJoining,
                                  TMaskedProjection>
    whereTMaskedDeleteCommand:MaskedDelete<TMaskedSelection>
    whereTMaskedInsertCommand:MaskedInsert<TMaskedInstantiation>
    whereTMaskedUpdateCommand:MaskedUpdate<TMaskedSelection,
                                           TMaskedMutation>
    whereTMaskedSchema:MaskedDataSource
    whereTMaskedData:MaskedData<TMaskedDataItem>
```

Listing 6.5: Janus mask framework mask manager definition

The typifications facilitate a standard for constructing mask kinds, enabling the construction of uniformly implemented masks. In essence, this facilitates a ubiquitous language for mask kind development. Additionally, these complex typifications are created so they can guide the development of a mask kind. In the Janus system, the development of a mask kind library using the mask framework is distilled into these general steps:

1.Create a mask kind library project, naming it: `Janus.Mask.<kind_name>`.

2.Implement the `MaskedDataModel`, `MaskedQueryModel`, `MaskedCommandModel`, and `MaskedSchemaModel` namespaces; respective to the mask kind being implemented. Implement them using the abstract classes found in the corresponding namespaces of the `Janus.Mask` project.

3.Implement the translator interfaces from the `Janus.Mask.Translation` namespace. Place them in the mask kind's `Translation` namespace. Set the `TDestination` generic type parameters according to the declared mask kind's local models from the previous step.

4.Implement the mask kind's schema manager by extending the `MaskSchemaManager<*>`. Place the masked schema model as the generic type parameter.

5.Implement the mask kind's query manager by extending the `MaskQueryManager<*>`. Place the masked model types in the generic type parameters accordingly.

6.Implement the mask kind's command manager by extending the `MaskCommandManager<*>`. Place the masked model types in the generic type parameters accordingly.

7.Implement the mask kind's component options class; even if nothing new is added to the class.

8.Representation-instance implementations should be placed in additional underlying namespaces.

9.Implement the mask kind's component manager by extending the `MaskManager<*>` class. Place the masked model types in the generic type parameters accordingly.

To develop the mask type (a runnable mask component), an additional step is required:

10.Create a mask kind application project, naming it: `Janus.Mask.<kind_name>.<app_type>`. Reference the mask kind library. Implement the application.

6.2.2 Web REST API mask

As a proof-of-concept for the mask component and the mask framework, a *virtualising Web REST* API mask* library was developed (referred to as the Web API mask). The use of the mask framework enabled the development of the Web API mask according to the steps provided in Section 6.2.1.

Before the actual development took place, design decisions were made concerning the mappings of the system format models to the representational format models.

The Web API must be constructed using the model-view-controller pattern provided by ASP.NET. The schema model elements must map to a REST API in such a way that:

- the `DataSource` is represented by the entire API starting at the root path '/', the;
- each `Schema` is represented by a URI subpath marked by the schema's name;

*The REST API is implemented at maturity level 2.

- each `Tableau` is represented by a controller with appropriate actions marking an explicit resource path;
- all `Attributes` are represented as fields of DTOs served or received at the resource paths.

Following the aforementioned decisions, each tableau is mapped to a controller so that it supports querying via a GET method request. The selection clause is generally supported through a query string in the URL. Joins and projection are not supported, as they don't have natural analogues in the REST API format. The acquisition of a singular item by its identifier is implicitly supported. The DELETE and INSERT methods are enabled on a resource if the underlying tableau contains an update set with all attributes included. These methods are supported by Janus insert and delete commands. Appropriate DTO types are prepared to facilitate INSERT on each resource. A PUT method is created for each update set on a tableau, with an accompanying DTO type.

The Web API is constructed at runtime, without the need for the entire mask application to restart if the underlying system schema changes.

The following presents the actual steps taken during the development of the Web API mask kind and its Web application type.

1. Creation of the library project

A C# .NET class library project is created to facilitate the code for the Web API mask library. The project is named `Janus.Mask.WebApi`.

2. Implementation of the masked models

The masked query model is implemented as classes describing the typing of the controller and DTO classes that must be created and instantiated during the mask's runtime. These classes are placed under the `MaskedSchemaModel` namespace for consistency with the framework's namespaces. The `WebApiTyping` class is declared to inherit from the framework's `MaskedDataSource` class. This marks the `WebApiTyping` class as the root of the masked model that is analogous to the `DataSource`. Consequently, this indicates that this model is considered a masked schema model for this mask kind. The `ControllerTyping` class contains the specifications for constructing a controller; its route, route prefix, DTO types, and supported operations. The `DtoTyping` represents the specification for individual DTOs served by controller actions. This specification will be used to construct concrete DTO classes and their instances at runtime. The Web API masked schema model is presented in Figure 6.14.

Figure 6.14: Web API masked schema model implementation

The masked data model is represented by the `WebApiDtoData` class, inheriting from the mask framework's `MaskedData<TItem>` class. The generic parameter is set as an `object` since the concrete DTO type will be determined only at runtime. This is a masked analogue for the `TabularData`. The masked data model is placed in `MaskedDataModel` namespace to be consistent with the mask framework namespaces.

The masked query is implemented by the `WebApiQuery` in the `MaskedQueryModel` namespace. The `WebApiQuery` is an analogue for the system `Query` as driven by an HTTP GET method. This is marked by setting the `WebApiQuery` to inherit from the mask framework's `MaskedQuery<*>`. The provided generic parameters are specified such that they reflect the aforementioned design decisions about querying in the Web API mask. The query's starting (or target) element is determined through a tableau identifier. The target tableau will be acquired through the controller supporting that specific query. Selection is represented as a nullable string, facilitating a URL query string. Since the support of joins and projection is not intended, the related types are provided as a `Unit` to truncate the `WebApiQuery` in those terms. The specified type is given in Listing 6.6.

```
1 publicsealedclassWebApiQuery
2     : MaskedQuery<TableauId,string?, Unit, Unit>
```

Listing 6.6: Web API mask masked query class type

The Web API masked command model contains the `WebApiDelete`, `WebApiInsert` and

`WebApiUpdate` commands, inheriting from their respective mask framework classes. `WebApiDelete` is a masked delete command, setting its selection type as a nullable string to enable the use of URL query strings. The `WebApiDelete` represents a `DeleteCommand` driven by an HTTP DELETE method. `WebApiInsert` is a masked insert command, with its instantiation being set as an `object` to facilitate the various DTO types that will be generated in the Web API. `WebApiInsert` represents an `InsertCommand` driven by an HTTP POST method. `WebApiUpdate` is a masked update command, with the selection type set as a nullable string to support URL query strings, and the mutation type set as an object enabling the use of various DTO types generated by the Web API. `WebApiUpdate` represents an `UpdateCommand` driven by an HTTP PUT method. The typification of the command model classes is shown in Listing 6.7.

```
1 publicsealedclassWebApiDelete : MaskedDelete<string?>
2
3 publicsealedclassWebApiInsert : MaskedInsert<object>
4
5 publicsealedclassWebApiUpdate : MaskedUpdate<string?,object>
```

Listing 6.7: Web API mask masked command model class types

The implemented masked models provide the basis for the forthcoming implementation of the Web API mask's inner components.

3. Implementation of translators

The next mask development step is the implementation of translators. The required translator interfaces can be found in the mask framework's project (`Janus.Mask`) in the `Translation` namespace. Each of the masked models requires an implementation of its translator. The direction of the translations in the translators is also important to note.

Schema translator

To remain consistent with the (non-obligatory) sequence of development in the masked model, the translator for the schema model is observed first. The `WebApiSchemaTranslator` is implemented by adhering to the mask framework's `IMaskSchemaTranslator<TMaskedSchema>` interface. The `TMaskedSchema` generic type parameter specifies the type to which the schema will be translated (destination type), and must inherit from the `MaskedDataSource`. In the case of the Web API mask, this type is the `WebApiTyping` class. The source type for the translation is already set beforehand as the `DataSource`. The interface enforces the implementation of a `Translate` method accepting a source `DataSource` and returning the result of a translated `WebApiTyping`. The type specifications of the implemented schema translator and the consequent method signature are shown in Listing 6.8.

```
1 publicclassWebApiSchemaTranslator :
2     IMaskSchemaTranslator<WebApiTyping>
```

```
{
    public Result<WebApiTyping> Translate(DataSource dataSource)
        => ...
}
```

Listing 6.8: Web API mask schema translator

The `Translate` method iterates over the `Tableau` elements of a `DataSource`, noting their schemas. For each tableau a `ControllerTyping` is created as follows:

- a controller route prefix is determined by its parent schema name;
- the type of the property representing the identity attribute is determined;
- a POST method DTO is specified with a `DtoTyping` if the tableau contains an update set containing all the tableau's attributes (default update set);
- a DELETE method is specified if the tableau contains a default update set;
- PUT method DTOs are specified as `DtoTyping` objects for each update set on the tableau.

The controller typings are gathered into a `WebApiTyping` object and returned as a result.

Data translator

The data translator in the Web API mask kind library is implemented as the `WebApiDataTranslator`. The code for the translator is shown in Listing 6.9. Unlike the other translators in the mask, the data translator enables two-way transformations for data. Consequently, the translator contains two `Translate` methods, each for one transformation direction of the translation. The mask framework's translator interface is already fixed on the `TabularData` as the source, while the destination is at this point set to indicate the `WebApiDtoData` and `object` as the data and the data item type respectively. This destination typification is limited to models inheriting the aforementioned `MaskedData` by the framework's interface. To ensure the correctness of the transformations in both ways, a lens pattern is used in the form of a `TabularDataDtoLens` to facilitate them (see Chapter 7). The translator enables the persistence of column names from the `TabularData` through the constructor parameter `columnNamePrefix`, as well as an alternative to providing a DTO type over a generic parameter via the `originalType` constructor parameter.

```
public class WebApiDataTranslator :
    IMaskDataTranslator<WebApiDtoData, object>
{
    private readonly TabularDataDtoLens<object> _dataLens;
    private readonly string _columnNamePrefix;

    public WebApiDataTranslator(
        string? columnNamePrefix = null,
        Type? originalType = null)
    {
        _dataLens =
```

```
            SymmetricTabularDataDtoLenses
                .Construct<object>(
                    columnNamePrefix,
                    originalType);
        _columnNamePrefix = columnNamePrefix ??string.Empty;
    }

    publicResult<TabularData> Translate(WebApiDtoData source)
        => Results.AsResult(
            () => _dataLens.PutLeft(
                source.Data,
                Option<TabularData>.None));

    publicResult<WebApiDtoData> Translate(TabularData destination)
        => Results.AsResult(
            () => _dataLens.PutRight(
                destination,
                Option<IEnumerable<object>>.None)
                .Map(data =>newWebApiDtoData(data)));
}
```

Listing 6.9: Web API mask data translator

Query translator

The Web API mask's query translator is implemented as the `WebApiQueryTranslator` (Listing 6.10), implementing the mask framework's `IMaskQueryTranslator<*>` interface. The framework's interface enforces the specification of generic types such that only a class inheriting the `MaskedQuery` can be provided. Additionally, proper clause types relating to the masked model are also enforced. In contrast to the schema translator, the system model here is positioned on the destination end of the translator, although this is not clearly visible in the mask framework interface, it is visible in the interface provided by the `Janus.Components` library.

Each query clause is translated by its own translation method. These methods are individually used in the `Translate` method translating the entire query. Since the masked query model doesn't contain joins and projection, their translation methods receive the `Unit` type and return a logical error result if they were to be called by accident.

The masked query is translated in the `Translate` method by first initiating a translation of the URL query string through string parsing. The result of the selection translation is mapped to the result of an operation initialising the open query builder and assigning the translated selection to the system query via the builder.

```
publicsealedclassWebApiQueryTranslator :
    IMaskQueryTranslator<WebApiQuery, TableauId,string?, Unit, Unit>
{
```

```
    public Result<Query>
    Translate(WebApiQuery query)
        => Results.AsResult(
            () => TranslateSelection(
            Option<string?>.Some(query.Selection?.TrimStart('?')),
            $"{query.StartingWith}.")
            .Map(selectionExpression =>
                QueryModelOpenBuilder.InitOpenQuery(query.
    ↪ StartingWith)
                    .WithSelection(conf => conf.WithExpression(
    ↪ selectionExpression))
                    .Build()));

    public Result<Joining>
    TranslateJoining(
        Option<Unit> joining,
        TableauId? startingWith =null)
        => ...

    public Result<Projection>
    TranslateProjection(Option<Unit> projection)
        => ...

    public Result<SelectionExpression>
    TranslateSelection(Option<string?> selection)
        => ...
}
```

Listing 6.10: Web API mask query translator

Command translator

The command translator for the Web API mask is implemented as the `WebApiCommandTranslator` class. This translator contains the translation methods required for all three command types. The generic type parameters concern the Web API masked command model by which the source model is specified. As in the case of the query translator, the command translator must implement translation methods for clauses that appear in commands. The notable difference from the other translators is that it requires the `WebApiDataTranslator` (visible as a field in Listing 6.11) to translate the data used in the instantiation and mutation clauses.

```
publicsealedclassWebApiCommandTranslator :
IMaskCommandTranslator<WebApiDelete, WebApiInsert, WebApiUpdate,
    ↪ string?,object,object>
{
    privatereadonlyWebApiDataTranslator _dataTranslator;

```

```
6       public WebApiCommandTranslator()
7       {
8           _dataTranslator = new WebApiDataTranslator();
9       }
10
11      public Result<DeleteCommand>
12      TranslateDelete(WebApiDelete delete)
13          => ...
14
15      public Result<InsertCommand>
16      TranslateInsert(WebApiInsert insert)
17          => ...
18
19      public Result<Instantiation>
20      TranslateInstantiation(Option<object> instantiation)
21          => ...
22
23      public Result<Mutation>
24      TranslateMutation(Option<object> mutation)
25          => ...
26
27      public Result<SelectionExpression> TranslateSelection(Option<
    ↪ string> selection)
28          => ...
29  }
```

Listing 6.11: Web API mask query translator

At this point, the type dependencies set by the mask framework can be noted. The translators can't be correctly implemented without the concrete masked model types and the manager components can't be correctly implemented without the concrete translator types.

4. Implementation of the `MaskSchemaManager`

The Web API schema manager, named `WebApiMaskSchemaManager` (Listing 6.12), is implemented by inheriting the mask framework's `MaskedSchemaManager` abstract class. The `MaskedSchemaManager` already contains the required methods implemented generically. These methods concern the loading, unloading and reloading of the schema into the manager, as well as providing the loaded schema to the components using the schema manager. Only the masked data source generic type parameter is required for the concrete implementation of a schema manager. In this case, the `WebApiTyping` is provided. The schema manager requires a mask communication node instance and a schema translator function, as prescribed by the constructor parameters. The mask communication node is used to acquire schemas from other system com-

ponents. This is visible in Listing 6.12 in the `LoadSchema` method, where the communication node is used to send a schema request (`SCHEMA_REQ`). The schema translator is used to translate the acquired schemas into the masked format (visible in the `CurrentMaskedSchema` property in Listing 6.12). When referencing other inner components, it is best to use the concrete types (their façades) instead of the exhaustively typed generic interfaces used in the base classes.

```
publicsealedclassWebApiMaskSchemaManager :
    MaskSchemaManager<WebApiTyping>
{
    privatereadonlyWebApiSchemaTranslator _schemaTranslator;

    publicWebApiMaskSchemaManager(
        MaskCommunicationNode communicationNode,
        WebApiSchemaTranslator schemaTranslator,
        ILogger? logger =null)
        :base(communicationNode, schemaTranslator, logger)
    {
        _schemaTranslator = schemaTranslator;
    }

    //belowaremethodstakenoverfromthebaseclass
    publicOption<WebApiTyping> CurrentMaskedSchema
    publicOption<TMaskedSchema> CurrentMaskedSchema
        => _currentSchema.Map(_schemaTranslator.Translate)
            .Bind(translation => translation
                ? Option<TMaskedSchema>.Some(translation.Data) :
    ↪ Option<TMaskedSchema>.None);

    publicOption<DataSource> CurrentOutputSchema
        => ...

    publicasync Task<Result<DataSource>>
    GetSchemaFrom(RemotePoint remotePoint)
        => ...

    publicasync Task<Result<DataSource>>
    LoadSchema(RemotePoint remotePoint)
        => (await Results.AsResult(async () =>
        {
            ...
            var result =
                await _communicationNode
                .SendSchemaRequest(remotePoint);
            ...
        }))
```

```
39
40      publicasync Task<Result<DataSource>>
41      ReloadOutputSchema()
42          => ...
43
44      publicResult UnloadSchema()
45          => ...
46
47      publicResult
48      UnloadSchema(RemotePoint remotePoint)
49          => ...
50  }
```

Listing 6.12: Web API mask schema manager

5. Implementation of the `MaskQueryManager`

The Web API mask query manager is implemented as a class called the `WebApiMaskQueryManager` (part of implementation given in Listing 6.13). The query manager inherits the abstract `MaskQueryManager` class that already contains most methods implemented generically. The generic type parameters for the `MaskQueryManager` reflect on the masked query, schema and data model. The query manager requires a mask communication node to send queries to other system components (line 41 in Listing 6.13), a schema manager to acquire information about the currently loaded schema and its source, and a query translator to translate the masked queries to the system format.

The `MaskQueryManager` requires the implementation of the `RunQuery` method accepting a masked query as a parameter and returning the masked data as a result. The `RunQuery` is shown in Listing 6.13. The method first prepares a temporary data translator. The method then translates the masked query into the system format, passing it to the `RunQuery` accepting the system format. The query result data is finally translated by the temporary data translator into the masked format and returned as a result.

```
1   publicsealedclassWebApiMaskQueryManager :
2       MaskQueryManager<WebApiQuery, TableauId,string?,
3                        Unit, Unit, WebApiTyping,
4                        WebApiDtoData,object>
5   {
6       privatereadonlyWebApiQueryTranslator _queryTranslator;
7       privatereadonlyILogger<WebApiMaskQueryManager>? _logger;
8
9       publicWebApiMaskQueryManager(
10          MaskCommunicationNode communicationNode,
11          WebApiMaskSchemaManager schemaManager,
```

```
12          WebApiQueryTranslator queryTranslator,
13          ILogger? logger =null)
14          :base(communicationNode, schemaManager, queryTranslator,
    ↪ logger)
15      {
16          _queryTranslator = queryTranslator;
17          logger?.ResolveLogger<WebApiMaskQueryManager>();
18      }
19
20      publicoverrideasync Task<Result<WebApiDtoData>>
21      RunQuery(WebApiQuery query)
22          => await Results.AsResult(() =>
23          {
24              var dataTranslator =
25                  newWebApiDataTranslator(query.StartingWith.ToString
    ↪ () +".", query.ExpectingReturnDtoType);
26
27              var queryResult =
28                  Task.FromResult(_queryTranslator.Translate(query))
29                  .Bind(query => RunQuery(query))
30                  .Bind(async data => dataTranslator.Translate(data))
31                  .Map(data => data);//boxit
32
33              returnqueryResult;
34          });
35
36      //methodtakenoverfromthebaseclass
37      publicasync Task<Result<TabularData>>
38      RunQuery(Query query)
39          => (await Results.AsResult<TabularData>(async () =>
40              ...
41                  var result =
42                      await _communicationNode
43                      .SendQueryRequest(
44                          query,
45                          _schemaManager.CurrentSchemaRemotePoint.Value
46                          );
47              ...
48              );
49  }
```

Listing 6.13: Web API mask query manager

6. Implementation of the `MaskCommandManager`

The Web API mask command manager is implemented as a class called the `WebApiMaskCommandManager`. The command manager supports operations regarding all three types of commands. The generic type parameters concern the masked command, schema and data models. The command manager requires a mask communication node in order to send commands to other system components, a schema manager to acquire the latest loaded schema, and a command translator to translate the commands from the masked to the system format.

Listing 6.14 contains a partial implementation of the command manager. The implementation of the `RunCommand` method for the update command is given starting at line 28. The method translates the command to the system format using the command translator and then proceeds to run the command in the system format. The `RunCommand` method (line 41) from the base class uses the communication node to send a command request to a remote system component. There is no reverse translation since the result is only a logical indication of the outcome of the operation.

```
1  publicsealedclassWebApiMaskCommandManager :
2      MaskCommandManager<WebApiDelete, WebApiInsert, WebApiUpdate,
3                         string?,object,
4                         object, WebApiTyping>
5  {
6      privatereadonlyWebApiCommandTranslator _commandTranslator;
7      privatereadonlyILogger<WebApiMaskCommandManager>? _logger;
8
9      publicWebApiMaskCommandManager(
10         MaskCommunicationNode communicationNode,
11         WebApiMaskSchemaManager schemaManager,
12         WebApiCommandTranslator commandTranslator,
13         ILogger? logger =null)
14         :base(communicationNode, schemaManager, commandTranslator,
       ↪ logger)
15     {
16         _commandTranslator = commandTranslator;
17         _logger = logger?.ResolveLogger<WebApiMaskCommandManager>();
18     }
19
20     publicoverrideasync Task<Result>
21     RunCommand(WebApiDelete command)
22         => ...
23
24     publicoverrideasync Task<Result>
25     RunCommand(WebApiInsert command)
26         => ...
```

```
27
28      publicoverrideasync Task<Result>
29      RunCommand(WebApiUpdate command)
30          => await Results.AsResult(() =>
31          {
32              var commandResult =
33                  Task.FromResult(
34                      _commandTranslator.TranslateUpdate(command))
35                  .Bind(cmd => RunCommand(cmd));
36
37              returncommandResult;
38          });
39
40      //frombaseclass
41          publicasync Task<Result>
42          RunCommand(BaseCommand command)
43          => (await Results.AsResult(async () =>
44          {
45              ...
46
47              var result =
48                  await _communicationNode
49                  .SendCommandRequest(
50                      command,
51                      _schemaManager.CurrentSchemaRemotePoint.Value);
52
53              returnresult;
54          }));
55  }
```

Listing 6.14: Web API mask command manager

7. Implementation of the mask's options class

The Web API mask doesn't contain any additional options regarding the component itself, except for the options regarding the Web API instance. Because of this, the `WebApiMaskOptions` inherits from the mask framework's `MaskOptions` and extends it with fields regarding the specifics of the Web API instance.

The key implementation details of the `WebApiMaskOptions` are shown in Listing 6.15. The `_startupWebApi` field is introduced to allow the indication of whether the component should automatically start up the Web API instance. This is included for practical purposes regarding automatised deployment. The `WebApiOptions` class contains specific options regarding the instance of the Web API: the HTTP port number, HTTPS port number and whether to even use

HTTPS.

```
publicsealedclassWebApiMaskOptions : MaskOptions
{
    privatereadonlyWebApiOptions _webApiOptions;
    privatereadonlybool_startupWebApi;
    ...
}
...
publicclassWebApiOptions
{
    publicintListenPort { get; init; }
    publicint? ListenPortSecure { get; init; }
    publicboolUseSSL { get; init; }
}
```

Listing 6.15: Web API mask options

8. Implementation of the instance functionalities

All code regarding the actual Web API instance is placed in the `InstanceManagement` namespace. The primary task of the code in this namespace is to facilitate the repeated creation of an ASP.NET Web API according to the typing specifications gathered as a masked schema. The concretely implemented classes enabling the running of such a feature are shown in Figure 6.15.

The `WebApiInstance` contains the Web API instance at runtime as an ASP.NET `WebApplication`. The `TypeFactory` is used to create new controllers and their DTO types according to the typings given as `ControllerTyping` and `DtoTyping`. The types are created through the use of abstract partially implemented templates and their concrete implementation using the IL generator. The DTOs are only templated as inheriting the `BaseDto` class. The controllers are templated through the generic abstract `GenericController` class. This base controller class only provides support for GET methods, so additional generic interfaces `IDeleteController`, `IPutController` and`IPostController` are assigned and implemented using the IL generator to enable the required DELETE, PUT and POST methods. These new types are placed in a dynamic assembly.

Since the Web API instance controllers require access to querying and commanding, they must have such functionalities provided. This is achieved by packing the mask query and command managers into their respective query and command provider; the `QueryProvider` and `CommandProvider`. Because of this, the controllers unanticipatedly also adhere to the command query segregation principle. To simplify the injection of the provider dependencies, the providers are injected via the `ProviderFactory`, which is then used to acquire the concrete `QueryProvider` and `CommandProvider` instances as needed.

Figure 6.15: Key classes used for the Web API instance management

The `StartApplication` method in the `WebApiInstance` prepares and starts the Web API instance as follows:

•the mask's current masked schema is acquired through the schema manager;

- the `WebApiOptions` are applied to the Web API configuration;
- the elements of the masked schema are used together with the `TypeFactory` in a `GenericControllerFeatureProvider` to populate a dynamic assembly with the new DTO and controller types;
- the newly created controller types are added to the ASP.NET `WebApplication` through the use of the `GenericControllerFeatureProvider`;
- the `QueryProvider` and `CommandProvider` are registered as services through the `ProviderFactory`;
- the Swashbuckle Swagger graphical interface is added to the configuration to enable a visual overview of the Web API;
- the `WebApplication` object is built, ran asynchronously, and referenced for future management purposes.

9. Implementation of the `MaskManager`

The Web API mask manager is implemented as the `WebApiMaskManager` class, inheriting from the mask framework's `MaskManager`. The generic type parameters concern all of the aforementioned Web API masked models. The `WebApiMaskManager` inherits all the basic component management methods; this includes establishing connections with other system components, managing schemas, and sending queries and commands. The mask manager is the central command component that abstracts away the functionalities of the other management components. Because of this encompassing scope, the mask manager requires all of the other manager components. This is visible in Listing 6.16 with the constructor requiring the query, command and schema managers, as well as the communication node.

The Web API mask manager is also tasked with creating and managing the Web API instance. The management of the Web API instance in terms of the Web API mask manager interface implies the ability to start and stop the Web API instance. This functionality is shown in Listing 6.16 in the `StartWebApi` and `StopWebApi` methods.

```
1  publicsealedclassWebApiMaskManager
2      : MaskManager<WebApiQuery, TableauId,string?, Unit, Unit,
3                    WebApiDelete, WebApiInsert, WebApiUpdate,
4                    object,object, WebApiTyping,
5                    WebApiDtoData,object>
6  {
7      ...
8      publicWebApiMaskManager(
9          MaskCommunicationNode communicationNode,
10         WebApiMaskQueryManager queryManager,
11         WebApiMaskCommandManager commandManager,
12         WebApiMaskSchemaManager schemaManager,
```

```
13          ...
14          WebApiMaskOptions maskOptions,
15          ...)
16          {
17              ...
18              _webApiInstance =
19                  newWebApiInstance(
20                  maskOptions.WebApiOptions, commandManager,
21                  queryManager, schemaManager, logger);
22              ...
23          }
24
25      publicResult StartWebApi()
26          => Results.AsResult(
27              () => _webApiInstance
28                      .StartApplication(GetCurrentSchema()));
29
30      publicResult StopWebApi()
31          => Results.AsResult(
32              () => _webApiInstance.StopApplication());
33
34
35  }
```

Listing 6.16: Web API mask manager

10. Creation of the Web application type

A separate ASP.NET MVC Web Application project is created with the name `Janus.Mask.WebApi.WebApp`. This project contains the code for the Web application type of the Web API mask kind. The project references the `Janus.Mask.WebApi` project. The `Program.cs` file contains the configuration code for the Web application, as well as the dependency injection configuration for all of the mask's inner components. The dependency injection configuration includes all of the required components for instantiating a `WebApiMaskManager`. This crucially includes:

- the mask component options;
- the TCP network adapter;
- the communication data serializer;
- the mask communication node;
- the mask query, command, schema, and query translators;
- the query, command, and schema managers;
- the mask manager.

The final class diagram overview of the implemented inner components of the Web API

mask is given in Figure 6.16

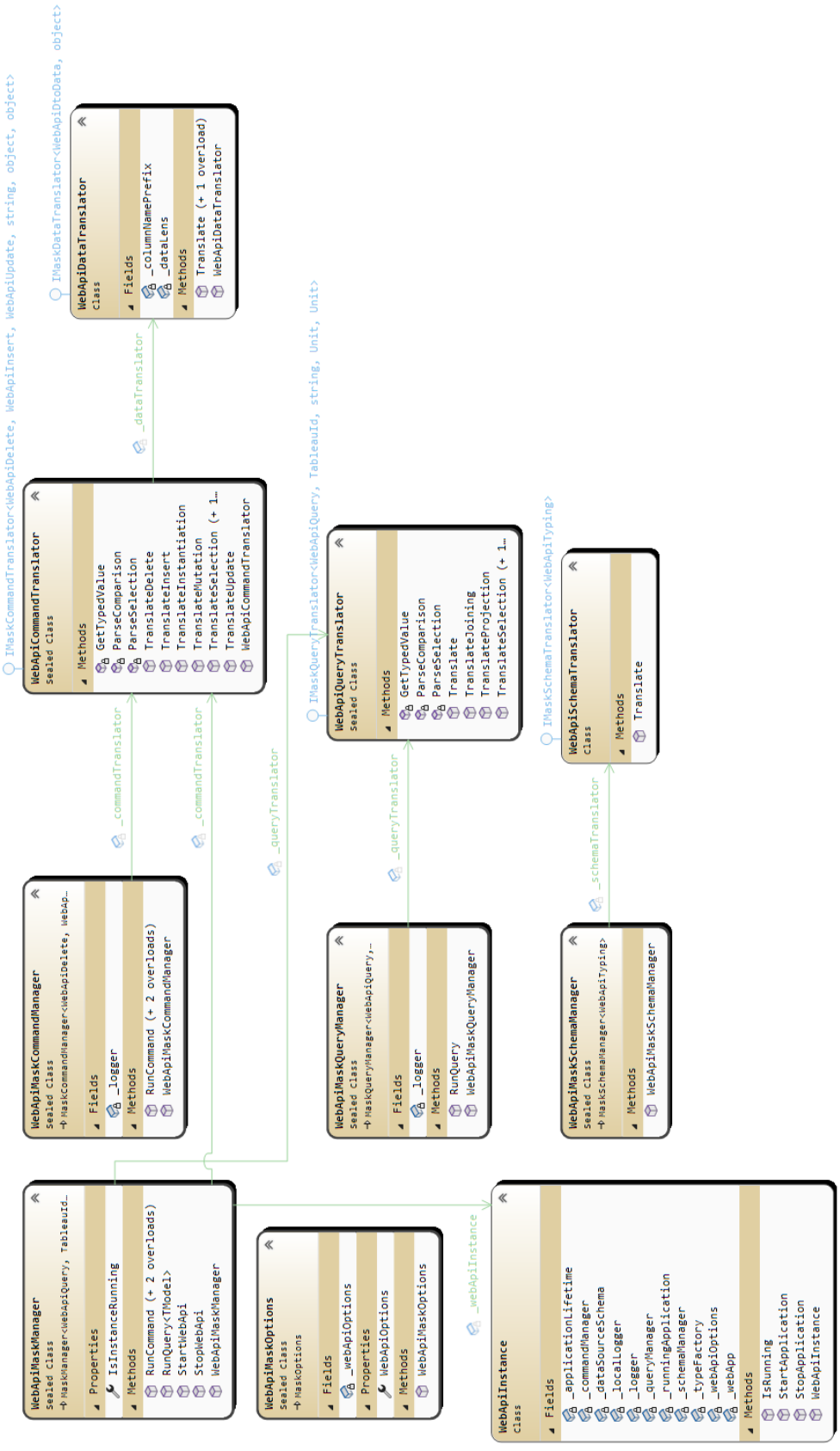


Figure 6.16: Management classes of the Janus Web API mask library

Figure 6.17 is provided in order to compare the established implementation of the Web API mask with the proposed usage of the mask framework in Section 5.3, specifically illustrated in Figure 5.4. The mask framework (itself implemented as the `Janus.Mask` library project) was used to construct a Web API mask library. The Web API mask library is used primarily through the Web API mask manager since the manager provides a façade of the Web API mask's functionalities. Foremost, the Web API mask manager controls the Web API instance and shares its `QueryManager` and `CommandManager` with it. The Web API instance is an instance of ASP.NET Web API application recreated and restarted at runtime depending on the current schema available to the mask manager. The Web API instance provides the masked data, schema, queries, and commands to end-users. These components make up the Web API mask kind.

In order for the Web API mask manager to be run, it must be positioned within an executable component. The mask management Web application is used to this end, as well as enabling the Web API mask to be managed through a Web graphical user interface. The management user interface is not intended for end-users but for administrators managing the deployed component. The creation of a mask management Web application effectively determines the type of this deployable and executable mask component - a Web application type of a Web API kind.

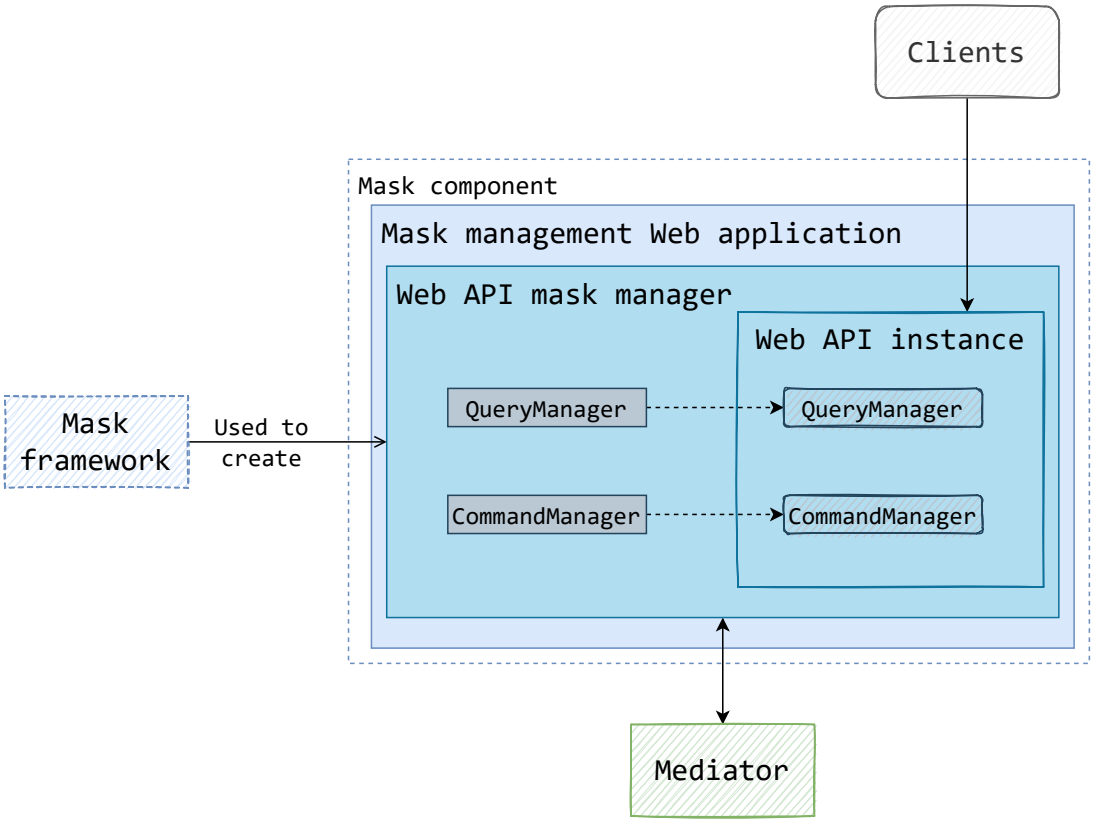


Figure 6.17: Web API mask creation facilitated by the mask framework

The Web API mask component (as proposed in Section 5.3) is a prefabricated component that only requires adjustments to its configuration to be deployed in various MMW topologies.

The configuration is adjusted through a JSON configuration file placed alongside the executable component. An example of the relevant parts of a configuration file is given in Listing 6.17 for a Web API mask from the case study prototype of Section 8.3.1.

```
{
  "MaskConfiguration": {
    "NodeId": "MusicWebApiMask",
    "ListenPort": 30001,
    "TimeoutMs": 5000,
    "CommunicationFormat": "AVRO",
    "NetworkAdapterType": "TCP",
    "EagerStartup": true,
    "StartupRemotePoints": [{"Address": "172.24.2.1", "ListenPort":
    ↪ 20001}],
    "StartupNodeSchemaLoad": "MusicMediator",
    "StartupWebApi": true,
    "PersistenceConnectionString": "./mask_database.db",
    "WebApiConfiguration": {
      "ListenPort":8801,
      "UseSSL": false,
      "ListenSecurePort":8802
    }
  },
  "WebConfiguration": {
    "Port": 8301,
    "AllowedHttpHost": "http://*",
    "AllowedHttpsHost": "https://*"
  },
  ...
}
```

Listing 6.17: Example of a configuration file of a Web API mask

To provide a concrete example of a running Web API mask component, Figure 6.18 displays an instance of a generated Web API schema which can be found in the case study prototype of Section 8.3.1. The Swagger interface is included in the Web API mask to provide end-users with a concise overview of the masked schema. A more detailed overview of the generated Web API REST schema can be found in Section 10.5.1.

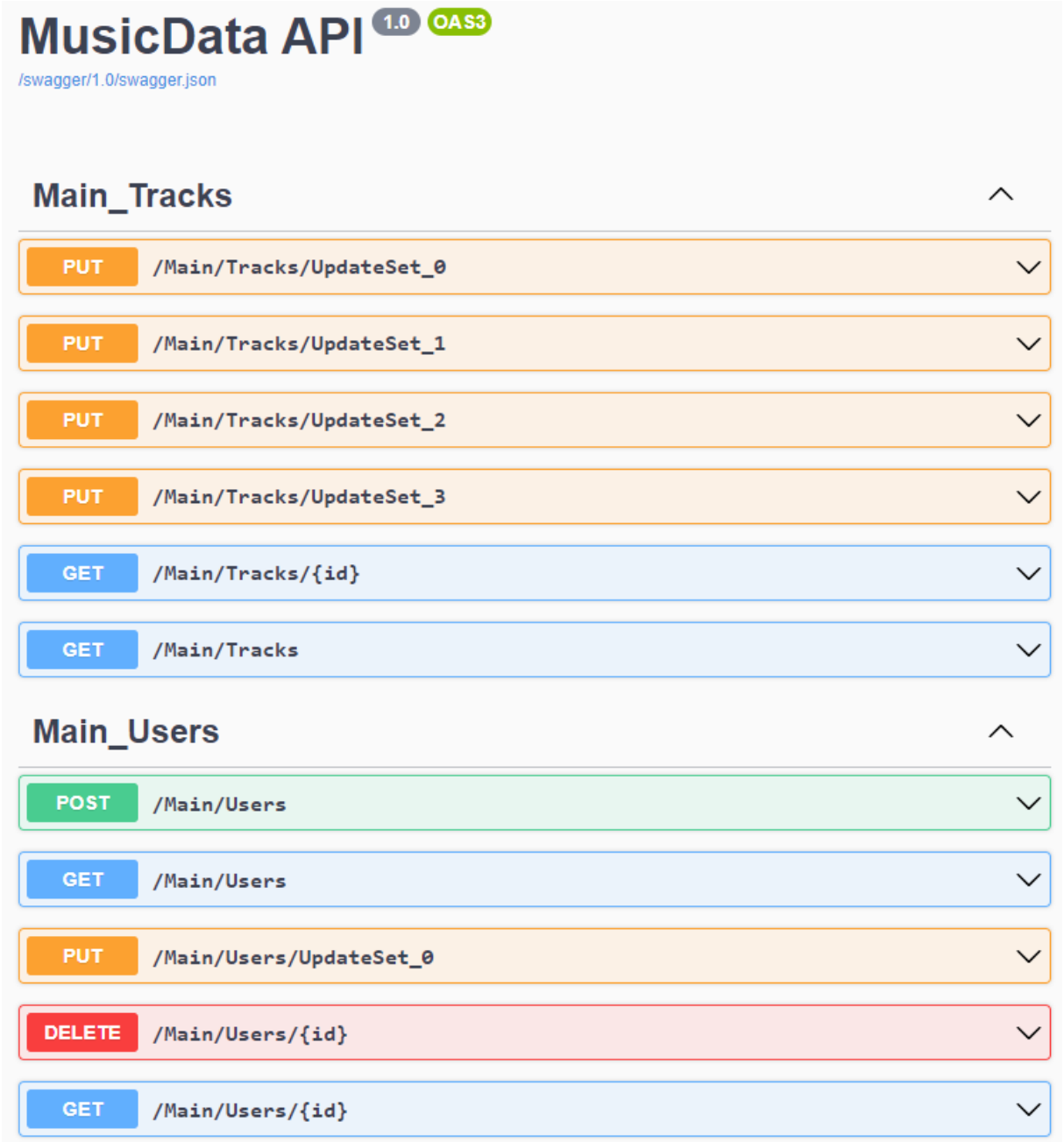


Figure 6.18: A Swagger interface for a Web REST API generated by a mask at runtime

Figure 6.18 shows the specific case of mask representing a system format data source `MusicData`, consisting of a single `Main` schema having two tableaus `Tracks` and `Users`, as a Web API.

6.2.3 LiteDB mask

As a prototype of a materialising mask discussed in Section 5.4, the *materialising LiteDB mask* was implemented. For a given data source, the LiteDB mask must generate a LiteDB database and populate it with the available data. The LiteDB mask library was implemented follow-

ing the steps for implementing a mask kind via the mask framework from Section 6.2.1. The implementation is illustrated in Figure 6.19.

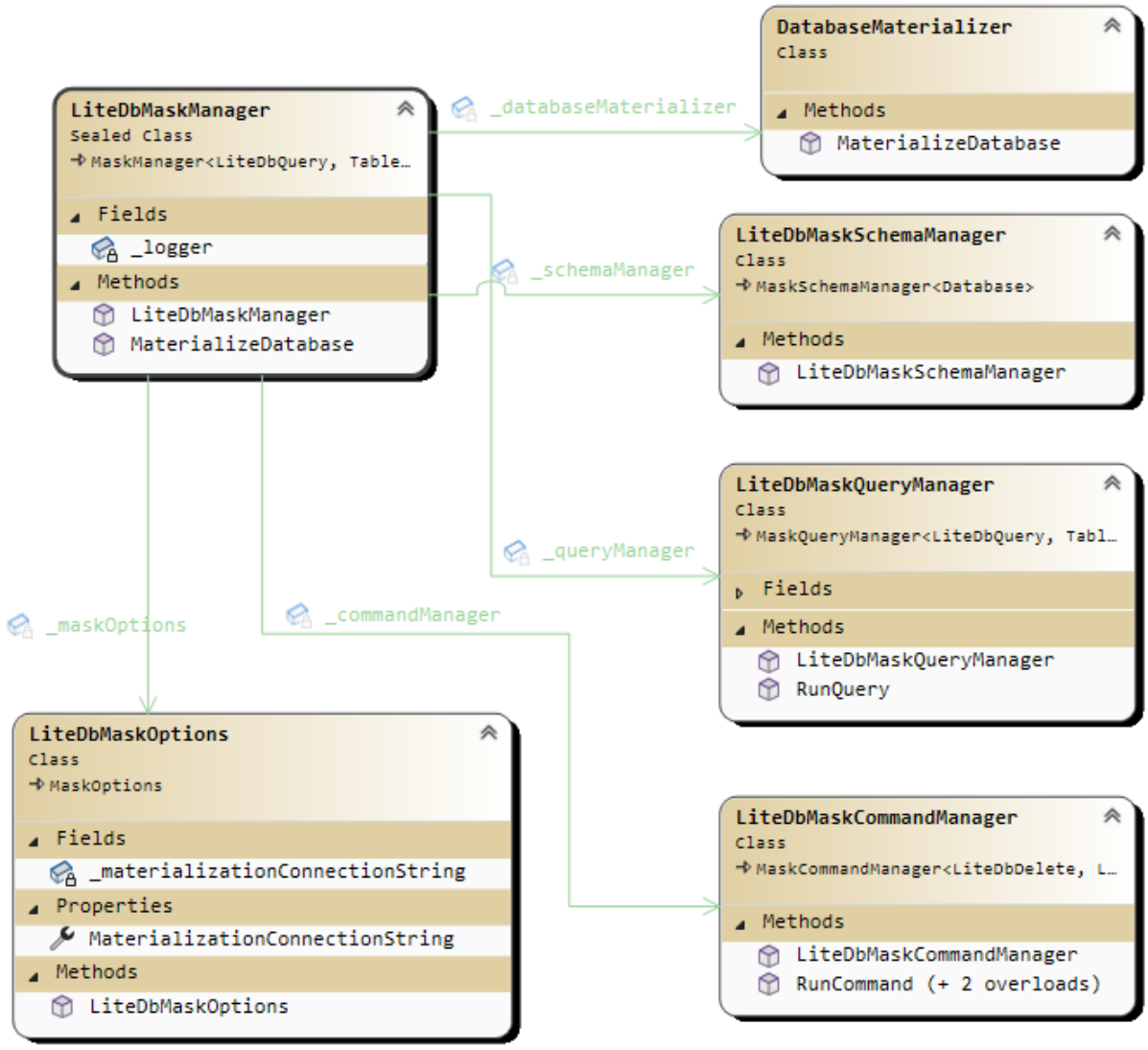


Figure 6.19: Management classes of the Janus LiteDB mask library

The LiteDB mask doesn't require representations for querying or commanding, since it is not used for virtualisation, and querying and commanding operations are expected to be run on the materialised database itself. Hence, the command and query models are not required in the LiteDB mask, but the mask framework requires these models to exist. Consequently, the command and query models are truncated by using the `Unit` type for clause type specifications (Listing 6.18).

```
/*litedbmaskedquery*/
publicabstractclassMaskedQuery<TStartingWith,TSelection,
                               TJoining,TProjection>
publicsealedclassLiteDbQuery
    :MaskedQuery<TableauId,Unit,Unit,Unit>
```

```
6
7  /*litedbmaskeddeletecommand*/
8  publicabstractclassMaskedDelete<TSelection> :MaskedCommand
9  publicsealedclassLiteDbDelete:MaskedDelete<Unit>
10
11 /*litedbmaskedinsertcommand*/
12 publicabstractclassMaskedInsert<TInstantiation>
13     :MaskedCommand
14 publicsealedclassLiteDbInsert:MaskedInsert<Unit>
15
16 /*litedbmaskedupdatecommand*/
17 publicabstractclassMaskedUpdate<TSelection,TMutation>
18     :MaskedCommand
19 publicsealedclassLiteDbUpdate:MaskedUpdate<Unit,Unit>
```

Listing 6.18: Janus LiteDB mask query and command model type definitions

The materialisation is facilitated by the `DatabaseMaterializer`. This is analogous to the Web API mask's instance (Section 6.2.2). To materialise the database on request, the LiteDB mask manager acquires the current masked schema from the schema manager, which then translates its currently loaded schema to the masked format (Figure 6.20) before returning it to the mask manager. The masked schema is then used by the `DatabaseMaterializer` to create a proper mapping strategy between the system and database schema. System queries are created and run, each to acquire data from a tableau. The resulting data is then translated into `LiteDbData` containing a collection of BSON documents appropriate for storage in a LiteDB database. The BSON documents are finally stored in the database, effectively materialising the database.

The current mapping strategy determines the representation of the system format schema. The data source is represented by a database file. The schema is represented by a prefix to collection names, concatenated by the underscore symbol. Each tableau is represented by a collection (of BSON documents). Individual attributes are represented as primitive value fields as implemented in LiteDB. Update sets are not represented in the materialised database since they are only related to the source data.

Reflecting on Figure 5.4 from Section 5.3, the LiteDB mask's creation via a mask framework is depicted in Figure 6.21. The LiteDB mask manager component represents the LiteDB mask library as a provider of all functionalities of a LiteDB mask. The LiteDB mask manager contains a database materializer component which is tasked with creating a new LiteDB database and making the required queries to populate it. The database materializer only requires the `QueryManager` from the LiteDB mask manager, since it only loads the data into the database. In contrast to the Web API mask, the LiteDB materializer is not a separately controlled instance of an application, just an object with the required methods to materialize the database.

The LiteDB mask manager component is placed within an ASP.NET Web application to

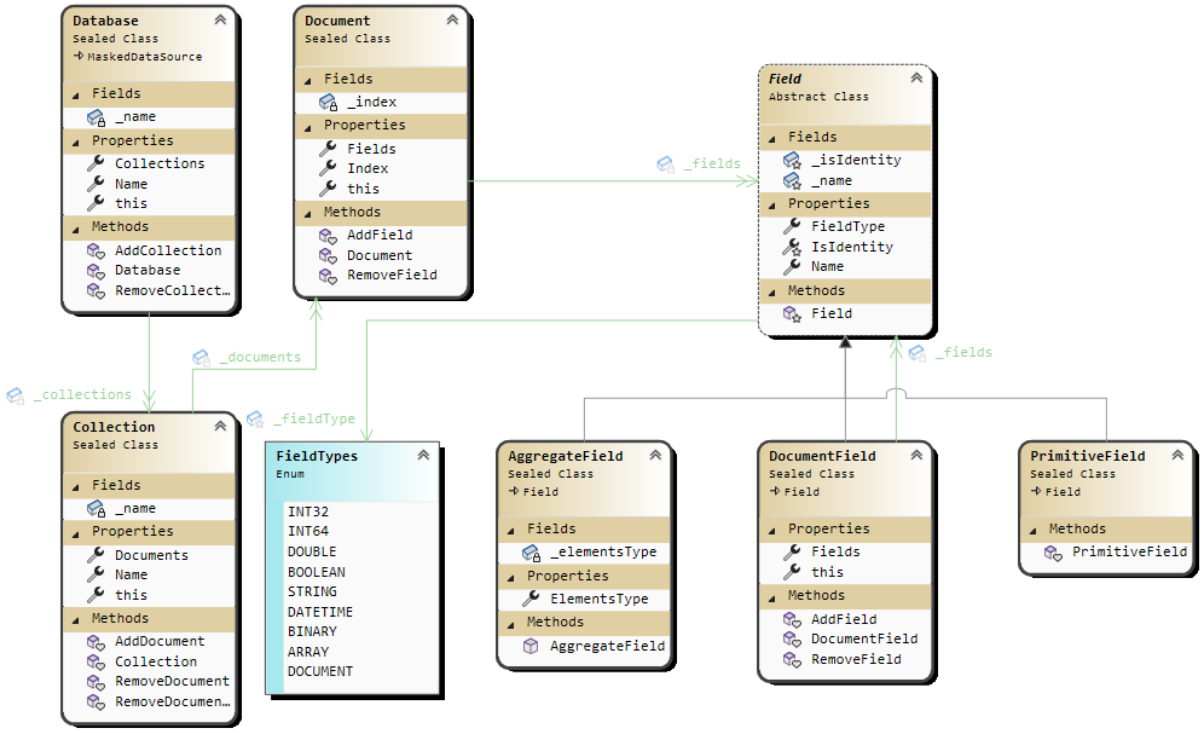


Figure 6.20: Janus LiteDB masked schema model

make it an executable component. The Web application enables the management of the mask component through a graphical user interface. The end-users are only expected to access the materialised database, and not the mask itself in this case. This mask is effectively of the Web application type of the LiteDB kind.

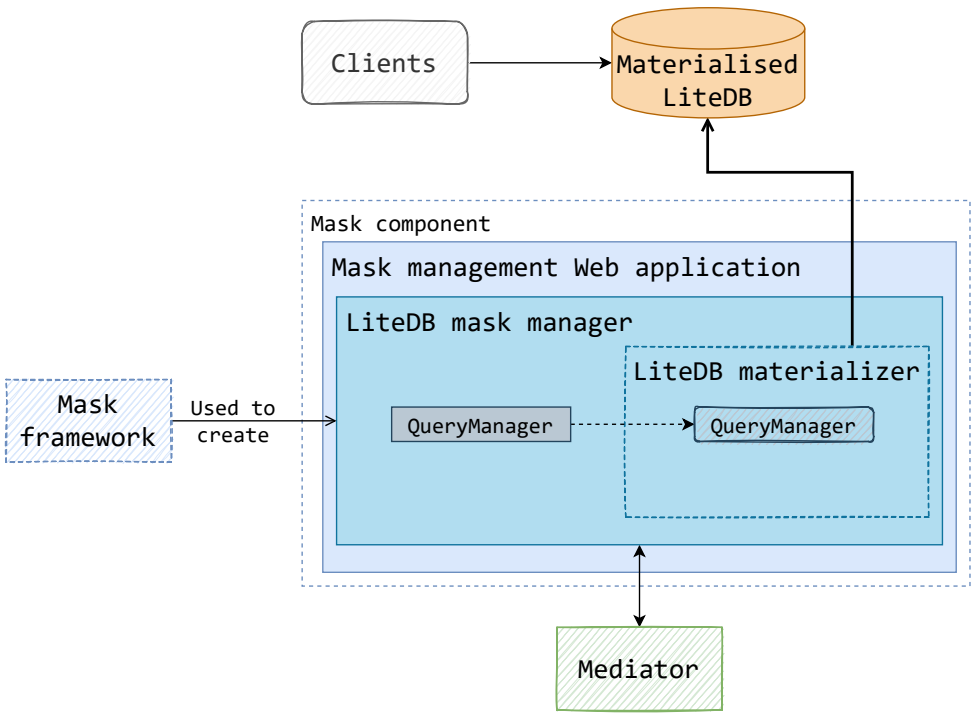


Figure 6.21: LiteDB mask creation facilitated by the mask framework

As was proposed in Section 5.3, the implemented LiteDB mask presented in this section is a prefabricated and configurable component that can be deployed in different MMW topologies. The configuration of the LiteDB component comes in the form of a JSON configuration file positioned next to its executable. An example configuration file used in the case study prototype of Section 8.3.1 is shown in Listing 6.19.

```
{
  "MaskConfiguration": {
    "NodeId": "InvoicingLiteDBMask",
    "ListenPort": 30004,
    "TimeoutMs": 5000,
    "CommunicationFormat": "AVRO",
    "NetworkAdapterType": "TCP",
    "EagerStartup": true,
    "StartupRemotePoints": [{"Address": "172.24.2.2", "ListenPort":
    ↪ 20002}],
    "StartupNodeSchemaLoad": "InvoicingMediator",
    "StartupMaterializeDatabase": true,
    "MaterializationConfiguration": {
      "ConnectionString": "invoicing.db"
    },
    "PersistenceConnectionString": "./mask_database.db",
  },
  "WebConfiguration": {
    "Port": 8307,
    "AllowedHttpHost": "http://*",
    "AllowedHttpsHost": "https://*"
  }
  ...
}
```

Listing 6.19: Example of a configuration file for a LiteDB mask

Taking the example of the case study prototype of Section 8.3.1, namely the `InvoicingLiteDBMask`, Listing 6.20 shows the content of the first documents from the three collections generated in the prototype. The database was materialised from the `InvoicingData` with a single `Main` schema containing the `Users`, `UserListenedTracks`, and `UserInvoices` tableaus.

```
/* Main_Users */
{
  "_id": {"$oid": "650c6ee87992cf0a3a8350ea"},
  "UserId": {"$numberLong": "1"},
  "UserFirstName": "Luis",
  "UserLastName": "Goncalves",
```

```
  "UserEmail": "luisg@embraer.com.br"
}, ...
/* Main_UserListenedTracks */
{
  "_id": {"$oid": "650c6eea7992cf0a3a8352c1"},
  "InvoiceItemId": {"$numberLong": "1"},
  "UserEmail": "leonekohler@surfeu.de",
  "Quantity": {"$numberLong": "1"},
  "TrackName": "Balls to the Wall"
}, ...
/* Main_UserInvoices */
{
  "_id": {"$oid": "650c6ee87992cf0a3a835125"},
  "InvoiceId": {"$numberLong": "1"},
  "UserId": {"$numberLong": "2"},
  "UserEmail": "leonekohler@surfeu.de",
  "InvoiceDate": {"$date": "2023-09-21T16:27:20.6910000Z"},
  "InvoiceTotal": 1.98
}, ...
```

Listing 6.20: Examples of the first documents from three collections generated by the LiteDB mask

The LiteDB mask can be used to physically export data or to create localised or temporary storage. A demonstration of this usability is given in Section 8.3.1, where the LiteDB mask is used to physically export mediated data into a LiteDB database.

6.2.4 SQLite mask

Another prototype of a materialising mask was implemented - the *materialising SQLite mask*. For a given data source, the SQLite mask must generate a SQLite database and populate it with the available data. The SQLite mask library was implemented following the steps for implementing a mask kind via the mask framework from Section 6.2.1. The implementation is illustrated in Figure 6.22.

The SQLite mask is analogous to the one presented for LiteDB. Consequently, its querying and commanding capabilities are also truncated in the way described in Section 6.2.3.

The materialisation of an SQLite database is facilitated by the `DatabaseMaterializer`. To materialise the database, the SQLite mask manager acquires the current masked schema from the schema manager, which then translates its currently loaded schema to the masked format (Figure 6.23) before returning it to the mask manager. The masked schema model is allowed to be modified by a user in terms of adding relationships between tables. The finalised masked schema is then used by the `DatabaseMaterializer` to create a proper mapping strategy between the system and database schema. System queries are created and run, each to acquire

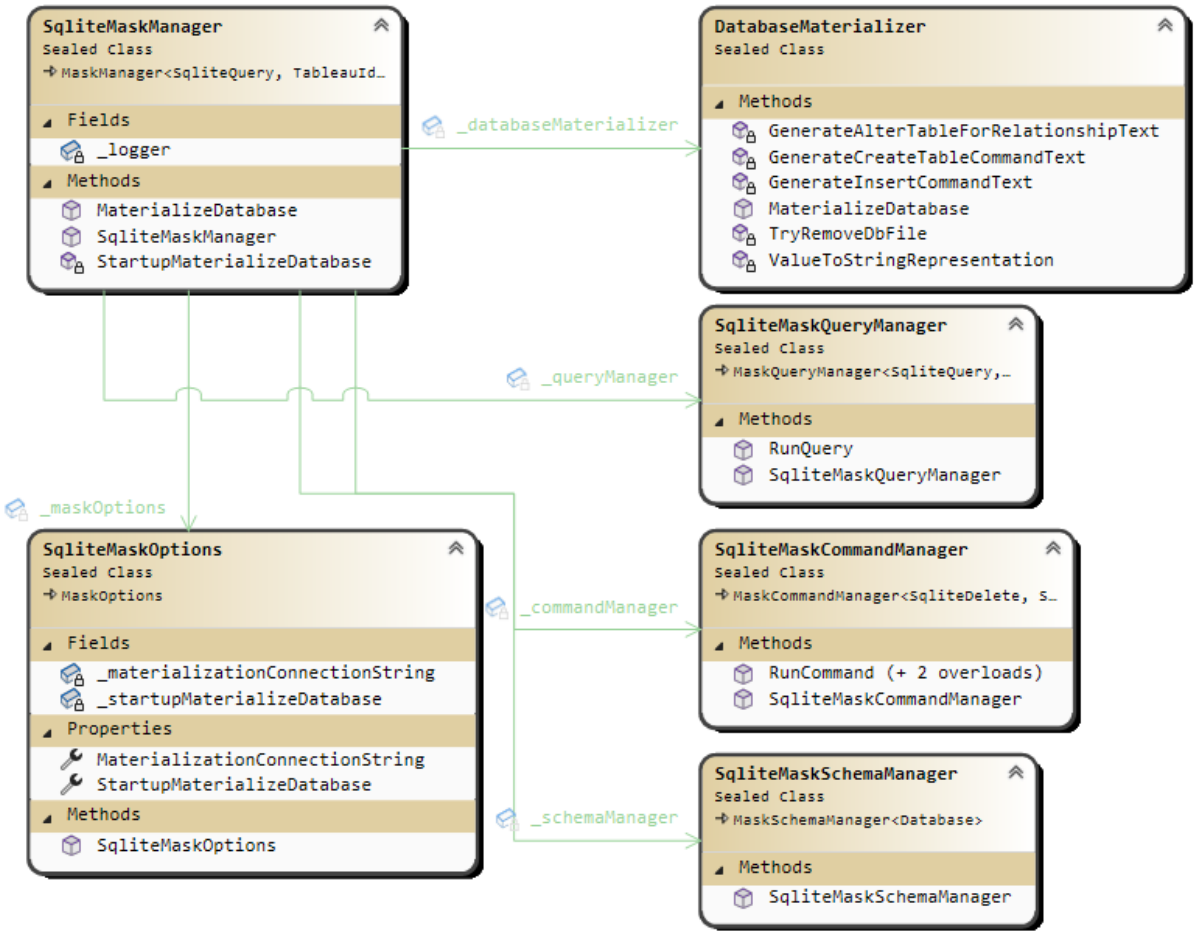


Figure 6.22: Management classes of the Janus LiteDB mask library

data from a tableau. The resulting data is then translated into `SqliteTabularData` containing a collection of `SqliteDataRow` which are appropriate for use in INSERT commands of an SQLite database.

The current schema model mapping strategy is as follows. The data source is represented by a database file. The schema is represented by a prefix to table names, concatenated by the underscore symbol. Each tableau is represented by a table. Individual attributes are represented as columns. Update sets are not represented in the materialised database since they are only related to the source data.

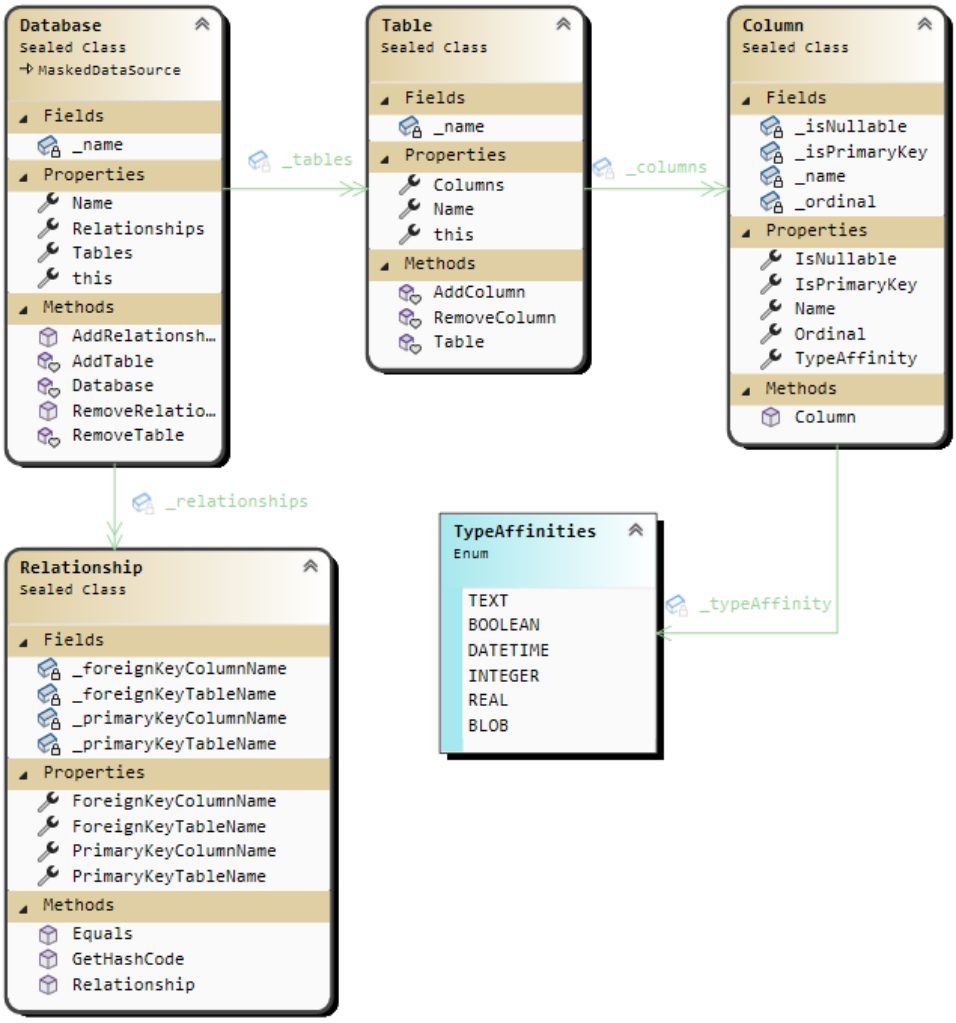


Figure 6.23: Janus SQLite masked schema model

Reflecting on Figure 5.4 from Section 5.3, the SQLite mask's creation via a mask framework is depicted in Figure 6.24. The SQLite mask manager component represents the SQLite mask library as a provider of all functionalities of a LiteDB mask. The SQLite mask manager contains a database materializer component which is tasked with creating a new SQLite database and making the required queries to populate it. The database materializer only requires the `QueryManager` from the SQLite mask manager, since it only loads the data into the database. The SQLite materializer is analogous to the one presented in Section 6.2.3.

The SQLite mask manager component is placed within an ASP.NET Web application to make it an executable component. The Web application enables the management of the mask component through a graphical user interface. The end-users are only expected to access the materialised database, and not the mask itself in this case. This mask is effectively of the Web application type of the SQLite kind.

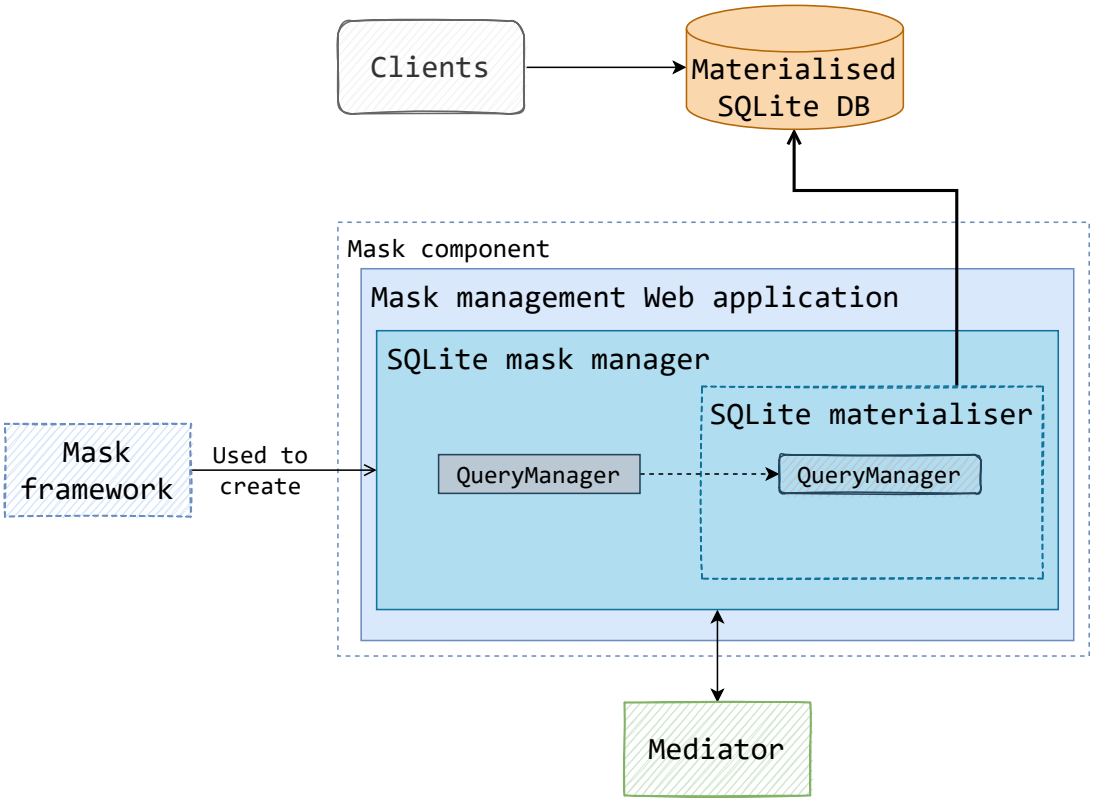


Figure 6.24: SQLite mask creation facilitated by the mask framework

Keeping with the proposal in Section 5.3, the implemented SQLite mask is a prefabricated and configurable component that can be deployed in different MMW topologies. The configuration of the SQLite component comes in the form of a JSON configuration file positioned next to its executable. An example configuration file used in the case study prototype of Section 8.3.1 is shown in Listing 6.21.

```
{
  "MaskConfiguration": {
    "NodeId": "MusicSqliteMask",
    "ListenPort": 30003,
    "TimeoutMs": 5000,
    "CommunicationFormat": "AVRO",
    "NetworkAdapterType": "TCP",
    "EagerStartup": true,
    "StartupRemotePoints": [{"Address": "172.24.2.1", "ListenPort":
    ↪ 20001}],
    "StartupNodeSchemaLoad": "MusicMediator",
    "StartupMaterializeDatabase": true,
    "MaterializationConfiguration": {
      "ConnectionString": "Data Source = ./music.db"
    },
    "PersistenceConnectionString": "./mask_database.db",
  },
```

```
  "WebConfiguration": {
    "Port": 8305,
    "AllowedHttpHost": "http://*",
    "AllowedHttpsHost": "https://*"
  },
  ...
}
```

Listing 6.21: Example of a configuration file for an SQLite mask

The case study prototype of Section 8.3.1, provides an example for a deployed SQLite mask component - the `MusicSqliteMask`. The prototype exemplifies the materialisation of an SQLite database from a data source named `MusicData` with a single schema `Main` having multiple tables; of which Figure 6.25 shows the physical model of the materialised SQLite database. The materialised database contains no concrete relationships, since they are not specified in the materialisation.

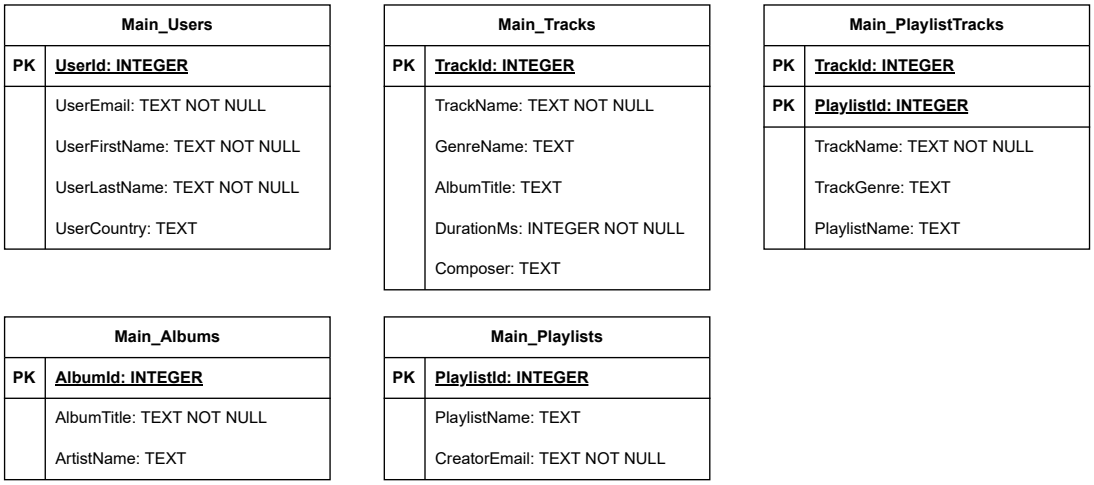


Figure 6.25: Example of an SQLite database materialised by the SQLite mask component

The SQLite mask is used to create physical exports of data in Section 8.3.1 and to create local data storages (in Section 8.3.3).

Chapter 7

Method for bidirectional data transformations

This chapter deals with the second contribution of this thesis - *a method for creating bidirectional data transformations in masks using bidirectionalisation to reduce the effort of mask implementation*. The chapter discusses the chosen method of *lenses* for two-way data transformations in masks, and the form in which they are theoretically adapted to fit into the requirements of the Janus system. The chapter covers the chosen method's implementation in the Janus system to transform masked and system data bidirectionally, as well as the method for providing proof for the level of behavedness of implemented lenses.

7.1 Lenses for data transformation

As was mentioned in Section 5.1, masks need to be capable of transforming data from a system format to a representative format, and vice-versa. Bidirectionalisation methods, as presented in Sections 3.3 and 3.3.5, can be utilized to facilitate a means of constructing two-way data transformations. Additionally, the use of bidirectionalisation methods provides the means of ascertaining the level of correctness of the created program.

The semantic and syntactic methods of bidirectionalisation provide the means to automatically generate inverse functions, and this alone could reduce implementation effort. Implementation effort can also be reduced by using design patterns [76]. The lens can be observed as a design pattern containing bidirectional transformations. This pattern is also composable, as lenses have the ability to be reused in complex lenses via combinators or simply composed. The use of the lens method doesn't rule out the possibility of using semantic and syntactic bidirectional methods to reduce the effort required of implementing elementary lenses. Rather, the use of lenses facilitates the hypothetical use of other bidirectionalisation methods. Consequently, the research for this thesis focuses on the way in which lenses can be utilised in a tangible

software system.

The software system in question for lens utilization in this thesis is Janus because it's a system that was identified to have the requirement of bidirectional transformations through the aforementioned requirements on masks.

Data in the Janus system is used as query results, mutation or instantiation specifications on equal terms in both the system and representational format. Therefore, changes to data can be expected on both sides of the transformations. A symmetric lens is appropriate for such a scenario. Additionally, the use of symmetric lenses allows for stateful data transformations on both sides to be introduced if needed later in the system's lifetime. The symmetric lens allows the position of the data formats (there is no strict source and view) to be ignored when composing lenses. A symmetric lens ubiquitously marks the transformations between formats, with the left or right orientation being inconsequential when selecting a lens for further composition. To cut down the complexity of a lens implementation even further, the need for complement storage is removed by choosing a *simple symmetric lens* as the lens to drive the bidirectional transformations in the Janus system.

The simple symmetric lens should also be able to accordingly manage errors and side-effects, if such arise. The existence of such cases is best exemplified in the Web API mask (Section 6.2.2), where data must be transformed through the use of reflection (see Section 7.3). Reflection introduces volatile operations to any programming language and can lead to unforeseen side-effects. To take such cases into account, a monad is introduced as the carrier of the result of a transformation. Expanding on Definition 39, a simple symmetric lens in the Janus system is declared as in Defitinition 42.

Definition 42. *A simple symmetric lens $l \in X \Leftrightarrow Y$ in the Janus system contains the following four functions:*

$$createR : X \to Result\ Y$$

$$createL : Y \to Result\ X$$

$$putR : X \to Y \to Result\ Y$$

$$putL : Y \to X \to Result\ X$$

subject to four round-tripping laws expressed by:

```
createR x >>= \y -> putL y x = return x          (CreatePutRLTest)
createL y >>= \x -> putR x y = return y          (CreatePutLRTest)
putR x y >>= \y -> putL y x = return x             (PutRLTest)
putL y x >>= \x -> putR x y = return y             (PutLRTest)
```

where Result is a monad having the associated `bind` *(denoted with the ">>=" operator) and* `return` *functions.*

7.2 Lens implementation in C#

Concrete lens implementations were introduced into the Janus system as a means to provide two-way data transformations for masks, guaranteeing a level of correctness for the implemented transformations. The level of correctness depends on the behavedness property of the lens and will be reasoned through lens behavedness when discussing these transformations. This is because behavedness was introduced as an established metric for bidirectional transformations in Section 3.3.

The code related to lenses and their implementations can be found under the `Janus.Lenses` namespace, or in the `Lenses` sub-namespace of a mask library when the implemented lens uses a specific masked data model tied to the mask kind. The `Janus.Lenses` contains the base definition of a simple symmetric lens in the form of a generic abstract class `SymmetricLens<*>`. The generic types of this class, `TLeft` and `TRight`, represent left and right types for which the simple symmetric lens provides bidirectionalisation. This abstract class is presented in its entirety in Listing 7.1. The lens definition, as prescribed by the abstract class, requires that all transformation methods be implemented. These methods are not directly exposed as public members but are exposed via their respective `Func<*>` properties. This enables referencing methods of a specific lens instance, simplifying function composition implemented in Janus' base library.

```
1 publicabstractclassSymmetricLens<TLeft, TRight>
2 {
3     //putL:Y->X?->X
4     publicFunc<TRight, Option<TLeft>, Result<TLeft>> PutLeft
5         => _PutLeft;
6
7     //putR:X->Y?->Y
8     publicFunc<TLeft, Option<TRight>, Result<TRight>> PutRight
```

```
        => _PutRight;

    //createR:X?->Y
    publicFunc<Option<TLeft>, Result<TRight>> CreateRight
        => _CreateRight;

    //createL:Y?->X
    publicFunc<Option<TRight>, Result<TLeft>> CreateLeft
        => _CreateLeft;

    protectedSymmetricLens() { }

    protectedabstractResult<TLeft>
    _PutLeft(TRight right, Option<TLeft> left);

    protectedabstractResult<TRight>
    _PutRight(TLeft left, Option<TRight> right);

    protectedabstractResult<TRight>
    _CreateRight(Option<TLeft> left);

    protectedabstractResult<TLeft>
    _CreateLeft(Option<TRight> right);
}
```

Listing 7.1: Simple symmetric lens as defined in the Janus system

The simple symmetric lens definition also enables the omission of a current state object through the `Option<T>` monad. This can be introduced because the data transformations in the mask might not have to be stateful - data is completely recreated during a transformation.

A trivial example of a symmetric lens implementation in the Janus system is exemplified by the `IntStringLens`, as shown in Listing 7.2. This lens isn't used in the Janus system, rather it is used as a illustrative implementation.

```
publicsealedclassIntStringLens : SymmetricLens<int,string>
{
    protectedoverrideResult<int> _CreateLeft(Option<string> right)
        => Results.AsResult(
            () => right.Match(
                    r => Convert.ToInt32(r),
                    () =>default)
                    );

    protectedoverrideResult<string> _CreateRight(Option<int> left)
        => Results.AsResult(
```

```
12              () => left.Match(
13                      l => l.ToString(),
14                      () =>string.Empty)
15                      );
16
17      protectedoverrideResult<int>
18      _PutLeft(stringright, Option<int> left)
19          => Results.AsResult(() => Convert.ToInt32(right));
20
21      protectedoverrideResult<string>
22          _PutRight(intleft, Option<string> right)
23          => Results.AsResult(() => left.ToString());
24  }
```

Listing 7.2: Simple symmetric lens as defined in the Janus system

7.3 Web API mask lenses

The `RowDataDtoLens` and `TabularDataDtoLens` are concretely used in the Janus system. They are specifically used for providing two-way data transformations for the Web API mask's data translator.

7.3.1 `RowDataDtoLens`

The `RowDataDtoLens` is an example of an elementary lens, which provides bidirectional transformations between the Janus data row model (described by the `RowData` class; see Section 6.1.2) and a singular DTO which is generically typed. This lens can be described with the expression: `RowDataDtoLens` $\in$ `RowData` $\Leftrightarrow$ `TDto`. The `TDto` type remains generic since DTO types are generated at runtime, and consequently, the type of the lens.

Listing 7.3 shows the expected equivalent left and right-side data as transformed by a `RowDataDtoLens` $\in$ `DataRow` $\Leftrightarrow$ `PersonDto` lens. The left-side data contains a `RowData` object having column values typed as `long`, `double`, `string` and `DateTime`. The right-side data contains the equivalent data representation in the form of a `PersonDto` object*. The `RowDataDtoLens` is expected to map the `RowData` column values to the fields of the `PersonDto` according to their respective names. The `PersonDto` fields in Listing 7.3 are hidden behind the equivalently named properties. The `PersonDto` object is expected to be represented by the ASP.NET Web framework as a JSON object shown in Listing 7.4.

```
1   RowData left =
```

*The exemplified `PersonDto` and its associated `TabularData` structure is used in the tests for lenses used in the Janus system. These structures are representative of the data types used in the system

```
    RowData.FromDictionary(newDictionary<string,object?>
    {
            {"Id", 1L },
            {"Coefficient", 1.2 },
            {"FirstName","John"},
            {"LastName","Doe"},
            {"DateOfBirth",newDateTime(1965, 1, 1)}
    });

PersonDto right =
    newPersonDto
    {
        Id = 1L,
        Coefficient = 1.2,
        FirstName ="John",
        LastName ="Doe",
        DateOfBirth =newDateTime(1965, 1, 1)
    };
```

Listing 7.3: The equivalent left and right data transformed by a `RowDataDtoLens` $\in$ `DataRow` $\Leftrightarrow$ `PersonDto`

```
{
    "Id": 1,
    "Coefficient": 1.2,
    "FirstName": "John",
    "LastName": "Doe",
    "DateOfBirth": "1965-01-01T00:00:00"
}
```

Listing 7.4: The json representation of the `PersonDto` object

Listing 7.5 contains the implementation of a *putR* function as the `_PutRight` method in the `RowDataDtoLens` class. The entire `_PutRight` implementation is wrapped in the `AsResult` high-order function to facilitate failures by exceptions risen through the use of reflection. Since the type environment of the lens is generated at runtime, the method tries to determine the DTO type either through the specified generic parameter of the lens or through a field containing the specific type (the latter can be specified through a constructor). The method then creates a default instance of the right-typed output object by using runtime instantiation. The output object is optionally populated with the field values specified by the `right` parameter through reflection. The output object is finally populated through reflection by column values of the `left` value.

```
protectedoverrideResult<TDto>
_PutRight(RowData left, Option<TDto> right)
```

```
    => Results.AsResult(() =>
    {
        //determinetypeofDTO
        Type rightType = _dtoType.Value ??typeof(TDto);
        //createadefaultrightitemobject
        var rightItem = Activator.CreateInstance(rightType);
         //populatetherightitemwithrespecttogivenright
        if(right)
        {
            foreach(var fieldinrightType.GetRuntimeFields())
             {
                var rightFieldValue = field.GetValue(right.Value);
                var targetField = rightType.GetField(field.Name,
↪ BindingFlags.Instance | BindingFlags.NonPublic);
                targetField?.SetValue(rightItem, rightFieldValue);
             }
         }
        //mapvaluesfromrightfieldsintorightcolumnvalues
        foreach(var (colName, value)inleft?.ColumnValues.Map(t =>
↪ (t.Key.Split('.').Last(), t.Value)) ?? Enumerable.Empty<(string
↪ ,object?)>())
        {
            stringfieldName =$"_{colName}";

            var targetField = rightType.GetField(fieldName,
↪ BindingFlags.Instance | BindingFlags.NonPublic);
            targetField?.SetValue(rightItem, value);
        }

        return(TDto)rightItem;
    });
```

Listing 7.5: *putR* function implemented by a `_PutRight` method of the `RowDataDtoLens`

Listing 7.6 contains the implementation of a *putL* function as the `_PutLeft` method in the `RowDataDtoLens` class. The `_PutLeft` method's principles of error management and DTO type inference is equivalent to the `_PutRight` method. The `_PutLeft` method then determines the column name prefixes for the resulting `RowData`. This is either taken from a specification from a field set by a constructor beforehand or through the `left` parameter's longest common column name prefix. The `right` type's property names and types are determined to specify the column names, system types of the column values, and their corresponding property names. The column values dictionary is used to populate the data of the resulting `RowData` by acquiring the DTO object's property values by using reflection. The column values dictionary is then used to create the resulting `RowData`.

```
protectedoverrideResult<RowData>
_PutLeft(TDto right, Option<RowData> left)
    => Results.AsResult(() =>
    {
        //determineDTOtype
        var dtoType = right?.GetType() ?? _dtoType.Value ??typeof(
    ↪ TDto);
        //determineRowDatacolumnnameprefix
        stringcolumnNamePrefix =
            _columnNamePrefix
            ? _columnNamePrefix.Value
            : FindLongestCommonPrefix(
                (left.Value ?? CreateLeft(Option<TDto>.None).Match(
                    l => l,
                    msg => RowData.FromDictionary(newDictionary<
    ↪ string,object?>()))
                ).ColumnValues.Keys
            );
        //determineDTOfieldnamesandtypesforpopulation
        var columnInfos =
            dtoType.GetRuntimeProperties()
            .Map(property => (name: property.Name, type: property.
    ↪ PropertyType))
            .Map(t => (propertyName: t.name, propertyType: t.type,
    ↪ columnName:$"{columnNamePrefix}{t.name}"))
            .ToDictionary(t => t.propertyName, t => t);
        //createcolumnvaluedictionarytoinstantiateRowData
        var rowData =
            newDictionary<string,object?>(
            (left.Value ?? CreateLeft(Option<TDto>.None).Match(
                    l => l,
                    msg => RowData.FromDictionary(newDictionary<
    ↪ string,object?>())
                    )
                ).ColumnValues
            );
        //populaterowdatadictionaryfromDTOpropertyvalues
        foreach(var propertyindtoType.GetRuntimeProperties())
        {
            var value = property.GetValue(right);
            var columnName = columnInfos[property.Name].columnName;
            rowData[columnName] = value;
        }

```

```
40          returnRowData.FromDictionary(rowData);
41      });
```

Listing 7.6: *putL* function implemented by a `_PutLeft` method of the `RowDataDtoLens`

7.3.2 `TabularDataDtoLens`

The `TabularDataDtoLens` provides bidirectional transformations between the Janus tabular data model (described by the `TabularData` class; see Section 6.1.2) and an enumerable of DTOs. The `TabularDataDtoLens` can be considered a complex lens since it uses the `RowDataDtoLens` to bidirectionally transform each DTO item and row data. The `TabularDataDtoLens` inherits the behavedness of the `RowDataDtoLens` in terms of individual row data and DTO items, but the transformation of tabular data and enumerable of DTOs depends on its own qualities. The `TabularDataDtoLens` can be described with the expression: `TabularDataDtolens` $\in$ `TabularData` $\Leftrightarrow$ `IEnumerable<TDto>`.

Listing 7.7 shows the expected equivalent left and right-side data as transformed by a `TabularDataDtoLens` $\in$ `TabularData` $\Leftrightarrow$ `IEnumerable<PersonDto>`. The left-side data contains a `TabularData` object with two rows having column value types as `LONGINT`, `DECIMAL`, `STRING` and `DATETIME`. The right-side data is a `IEnumerable<PersonDto>`, where the `PersonDto` is the same as the one used in Section 7.3.1. The right-side data is expected to be represented by the ASP.NET Web framework as a JSON object shown in Listing 7.8.

```
1  TabularData left =
2      TabularDataBuilder.InitTabularData(newDictionary<string, Commons
   ↪  .SchemaModels.DataTypes>
3      {
4          {"Id", DataTypes.LONGINT },
5          {"Coefficient", DataTypes.DECIMAL },
6          {"FirstName", DataTypes.STRING },
7          {"LastName", DataTypes.STRING },
8          {"DateOfBirth", DataTypes.DATETIME }
9      })
10     .AddRow(conf => conf.WithRowData(newDictionary<string,object?>()
11     {
12         {"Id", 1001L },
13         {"Coefficient", 1.0 },
14         {"FirstName","John"},
15         {"LastName","Smith"},
16         {"DateOfBirth",newDateTime(1985, 7, 14)}
17     }))
18     .AddRow(conf => conf.WithRowData(newDictionary<string,object?>()
19     {
```

```
        {"Id", 1002L },
        {"Coefficient", 1.1 },
        {"FirstName","Emily"},
        {"LastName","Johnson"},
        {"DateOfBirth",newDateTime(1992, 2, 19)}
    })).Build();

IEnumerable<PersonDto> right =newList<PersonDto>
{
    newPersonDto
    {
        Id = 1001L,
        Coefficient = 1.0,
        FirstName ="John",
        LastName ="Smith",
        DateOfBirth =newDateTime(1985, 7, 14)
    },
    newPersonDto
    {
        Id = 1002L,
        Coefficient = 1.1,
        FirstName ="Emily",
        LastName ="Johnson",
        DateOfBirth =newDateTime(1992, 2, 19)
    }
}
```

Listing 7.7: The equivalent left and right data transformed by a `TabularDataDtoLens` $\in$ `TabularData` $\Leftrightarrow$ `IEnumerable<PersonDto>`

```
[
    {
      "Id": 1001,
      "Coefficient": 1.0,
      "FirstName": "John",
      "LastName": "Smith",
      "DateOfBirth": "1985-07-14T00:00:00"
    },
    {
      "Id": 1002,
      "Coefficient": 1.1,
      "FirstName": "Emily",
      "LastName": "Johnson",
      "DateOfBirth": "1992-02-19T00:00:00"
    }
]
```

Listing 7.8: The json representation of the `IEnumerable<PersonDto>` object

Listing 7.9 shows the implementation of the *putR* function as the `_PutRight` method in the `TabularDataDtoLens` class. The method body is wrapped by the `AsResult` high-order function, to ensure the control of the program flow. The `left` parameter's `RowData` is mapped to individual DTO instances by using the `RowDataLens` (`_rowDataLens` field in the class). The mapping doesn't directly produce the DTO objects, but rather their encasing results. A fold is initiated to check the result outcomes and reach the DTO object. The fold returns an aggregated result containing the `IEnumerable<TDto>` with all of the DTOs transformed from individual `RowData` of the left-side `TabularData`.

```
protectedoverrideResult<IEnumerable<TDto>>
_PutRight(TabularData left, Option<IEnumerable<TDto>> right)
    => Results.AsResult(() =>
        left.RowData
        //mapitemlensresultsoverrowdata
        .Map(rd => _rowDataLens.PutRight(rd, Option<TDto>.None))
        //foldresultsintosingularresult
        .Fold(Results.OnSuccess<IEnumerable<TDto>>(Enumerable.Empty<
    ↪ TDto>()),
                (dtoRes, results) => results.Bind(r => dtoRes.Map(dto
    ↪  => r.Append(dto))))
    );
```

Listing 7.9: *putR* function implemented by a `_PutRight` method of the `TabularDataDtoLens`

Listing 7.10 shows the implementation of the *putL* function as the `_PutLeft` method in the `TabularDataDtoLens` class. The method body is wrapped by the `AsResult` high-order function, to ensure the control of the program flow. The `right` parameter DTO items are mapped to individual results containing column values for each row of the future `TabularData`. The collection of column values is then folded over an initialized `TabularDataBuilder`. The name specification of the resulting `TabularData` is given to the builder, and the `Build` method is called. The resulting `TabularData` object is returned wrapped in a result type.

```
protectedoverrideResult<TabularData>
_PutLeft(IEnumerable<TDto> right, Option<TabularData> left)
    => Results.AsResult(() =>
        right.Map(rightItem => _rowDataLens.CreateLeft(Option<TDto>.
    ↪ Some(rightItem))
                                .Bind(createdLeft => _rowDataLens.
    ↪ PutLeft(rightItem, Option<RowData>.Some(createdLeft)))
                                .Match(l => l.ColumnValues, msg =>
    ↪ newDictionary<string,object?>()))
```

```
7             .Fold(TabularDataBuilder.InitTabularData(newDictionary<
  ↪ string, DataTypes>((left.Value ?? CreateLeft(Option<IEnumerable
  ↪ <TDto>>.Some(right)).Data).ColumnDataTypes)),
8                 (values, tabularBuilder) => tabularBuilder.AddRow(
  ↪ conf => conf.WithRowData(newDictionary<string,object?>(values
  ↪ ))))
9                 .WithName(left.Value?.Name ?? CreateLeft(Option<
  ↪ IEnumerable<TDto>>.None).Data.Name)
10                .Build()
11        );
```

Listing 7.10: *putL* function implemented by a `_PutLeft` method of the `TabularDataDtoLens`

7.4 Behavedness of Janus lenses

One of the reasons for proposing bidirectional methods for two-way data transformations in masks was the ability to determine the correctness of those transformations. The behavedness of the implemented lenses can be determined by implementing tests which check their adherence to the round-tripping rules presented in Definition 42. The round-tripping laws' tests for all lenses are declared as a generic lens testing framework, described and partially implemented by the `SymmetricLensTestingFramework<TLeft, TRight>` class. Listing 7.11 shows that the framework is practically a C# reendition of the Haskell-like laws from Definition 42. The developer is only required to specify the test data and the lens under test when implementing the concrete behavedness tests.

For the aforementioned `IntStringLens`, the test implementation can be as exemplified in Listing 7.12. Since the `IntStringLens` lens passes these tests it can be stated that it is a very well-behaved lens, but only for the values -42 and "-42". To ascertain further proof of very well-behavedness in a general case, the lens transformation methods themselves would have to be evaluated, e.g. by using the substitution model. In this simple lens case, a test might simply be run over all possible integer values in C#, with the addition of invalid string representations of integers. This would prove that the lens is very well-behaved, but would require an impractical amount of computational resources and time for each test run. Additionally, type domains can be infinite, as is the case with the `RowDataDtoLens` and `TabularDataDtoLens`. Since their transformations depend on reflection, their tests would have to cover all possible classes that contain the system types mapped to the Janus `DataTypes`. The obvious solution is to demand that a lens be written in such a form that enables the use of the substitution model to prove its level of behavedness. In a primarily object-oriented language, even in C# which supports some functional paradigm aspects, this is at times difficult to achieve without an underlying functional paradigm framework and can hinder the timely development of a lens. The use of

reflections adds to this problem since it's questionable how such mechanisms could be expressed functionally.

```
publicabstractclassSymmetricLensTestingFramework<TLeft, TRight> :
    ↪ SymmetricLensTesting
{
    protectedabstractTLeft _x { get; }
    protectedabstractTRight _y { get; }

    protectedabstractSymmetricLens<TLeft, TRight> _lens { get; }

    publicoverridevoidCreatePutLRTest()
    {
        var result =
        _lens.CreateLeft(Option<TRight>.Some(_y))
            .Bind(x => _lens.PutRight(x, Option<TRight>.Some(_y)));

        Assert.True(result);
        Assert.Equal(_y, result.Data);
    }

    publicoverridevoidCreatePutRLTest()
    {
        var result =
        _lens.CreateRight(Option<TLeft>.Some(_x))
            .Bind(y => _lens.PutLeft(y, Option<TLeft>.Some(_x)));

        Assert.True(result);
        Assert.Equal(_x, result.Data);
    }

    publicoverridevoidPutLRTest()
    {
        var result =
        _lens.PutLeft(_y, Option<TLeft>.Some(_x))
            .Bind(x => _lens.PutRight(x, Option<TRight>.Some(_y)));

        Assert.True(result);
        Assert.Equal(_y, result.Data);
    }

    publicoverridevoidPutRLTest()
    {
        var result =
        _lens.PutRight(_x, Option<TRight>.Some(_y))
```

```
42              .Bind(y => _lens.PutLeft(y, Option<TLeft>.Some(_x)));
43
44          Assert.True(result);
45          Assert.Equal(_x, result.Data);
46      }
47  }
```

Listing 7.11: Simple symmetric lens testing framework class for round-tripping laws

```
1  publicsealedclassIntStringLensTests :
       ↪ SymmetricLensTestingFramework<int,string>
2  {
3      protectedoverrideint_x => -42;
4
5      protectedoverridestring_y =>"-42";
6
7      protectedoverrideSymmetricLens<int,string> _lens =>
       ↪ IntStringLenses.Construct();
8  }
```

Listing 7.12: Round-tripping laws' tests for `IntStringLens`

The practical view of the situation is that the lens framework tests should be used to verify the behavedness of a lens over sound instance representatives of the data types that will be used in the transformations. For the `RowDataDtoLens` and `TabularDataDtoLens`, this can be a DTO class and `TabularData` instance that contains properties of all possible types represented by the Janus `DataTypes` model (of significance to the lense's use case).

The `RowDataDtoLens` and `TabularDataDtoLens` are tested using the testing framework in `Janus.Lenses.Tests` project as follows in Listing 7.13 and 7.14. The `PersonDto` class is the same one that was used to demonstrate the expected data equivalence after transformation applications in Section 7.3.

```
1  publicsealedclassRowDataDtoLensTests :
2  SymmetricLensTestingFramework<RowData, PersonDto>
3  {
4      protectedoverrideRowData _x =>
5          RowData.FromDictionary(newDictionary<string,object?>
6          {
7                  {"Id", 1L },
8                  {"Coefficient", 1.2 },
9                  {"FirstName","John"},
10                 {"LastName","Doe"},
11                 {"DateOfBirth",newDateTime(1965, 1, 1) }
12         });
13     protectedoverridePersonDto _y =>newPersonDto
```

```
    {
        Id = 1L,
        Coefficient = 1.2,
        FirstName ="John",
        LastName ="Doe",
        DateOfBirth =newDateTime(1965, 1, 1)
    };

    protectedoverrideSymmetricLens<RowData, PersonDto> _lens
        => SymmetricRowDataDtoLenses.Construct<PersonDto>();
```

Listing 7.13: Round-tripping laws' tests for `RowDataDtoLens`

```
publicsealedclassTabularDataDtoLensTests :
SymmetricLensTestingFramework<TabularData, IEnumerable<PersonDto>>
{
    protectedoverrideTabularData _x =>
        TabularDataBuilder.InitTabularData(newDictionary<string,
    ↪ Commons.SchemaModels.DataTypes>
        {
            {"Id", DataTypes.LONGINT },
            {"Coefficient", DataTypes.DECIMAL },
            {"FirstName", DataTypes.STRING },
            {"LastName", DataTypes.STRING },
            {"DateOfBirth", DataTypes.DATETIME }
        })
        .AddRow(conf => conf.WithRowData(newDictionary<string,
    ↪ object?>()
        {
            {"Id", 1001L },
            {"Coefficient", 1.0 },
            {"FirstName","John"},
            {"LastName","Smith"},
            {"DateOfBirth",newDateTime(1985, 7, 14) }
        }))
        .AddRow(conf => conf.WithRowData(newDictionary<string,
    ↪ object?>()
        {
            {"Id", 1002L },
            {"Coefficient", 1.1 },
            {"FirstName","Emily"},
            {"LastName","Johnson"},
            {"DateOfBirth",newDateTime(1992, 2, 19) }
        }))
        ...//morerows
        .WithName("PeopleData")
```

```
31          .Build();
32
33      protectedoverrideIEnumerable<PersonDto> _y =>newList<
    ↪ PersonDto>
34      {
35          newPersonDto
36          {
37              Id = 1001L,
38              Coefficient = 1.0,
39              FirstName ="John",
40              LastName ="Smith",
41              DateOfBirth =newDateTime(1985, 7, 14)
42          },
43          newPersonDto
44          {
45              Id = 1002L,
46              Coefficient = 1.1,
47              FirstName ="Emily",
48              LastName ="Johnson",
49              DateOfBirth =newDateTime(1992, 2, 19)
50          },
51          ...//moreitems
52      };
53
54      protectedoverrideSymmetricLens<TabularData, IEnumerable<
    ↪ PersonDto>> _lens =>
55          SymmetricTabularDataDtoLenses.Construct<PersonDto>();
56  }
```

Listing 7.14: Round-tripping laws' tests for `TabularDataDtoLens`

These tests confirm that the implemented `RowDataDtoLens` and `TabularDataDtoLens` are well-behaved for sound representatives of the transformed data; and that they are adequate to facilitate transformations in Web API mask data translator.

Chapter 8

Mask–mediator–wrapper case studies

This chapter provides additional qualitative observations on the mask–mediator–wrapper architecture in the form of case studies. The provided case studies present the capability of the mask–mediator–wrapper architecture to emulate other data management systems. The mask–mediator–wrapper architecture is shown to be versatile through the hypothetical emulation of the *SOS system* in Section 8.1 and the *data mesh* in Section 8.2. These sections take the material from the contributing paper [14] and the tied preprint paper [28], with some minor revisions.

The chapter introduces prototypes for each case study, including the data source integration system as a case study, to prove the correctness of the previously made assumptions (Section 8.3). The case studies are an integral part of the qualitative analysis of the MMW architecture, hence their proof is paramount. The case studies' prototypes introduce target architecture topologies that the MMW architecture was proposed to be capable of emulating and supporting. The Janus system was utilized as the demonstrating system. The case studies' prototypes demonstrate the emulation capabilities of the MMW architecture by deploying the Janus system as a *data source integration system*, the *SOS system*, and a *data mesh*.

8.1 SOS system emulation case study

As referenced before, Atzeni et al. [116] presented SOS (Save Our Systems) as a system for uniform operations over non-relational stores. The representational format of this system is a Web REST API with URI-like resource identification, serving result data as JSON objects.

The SOS system is organized into two main modules of the *common data model* and the *common interface*. The common data model provides access to the underlying data stores and is supported by a *mapper* implementation which is created per data store type. The model mappers are constructed via a `NonRelationalMapper` interface. The common interface provides system access features in terms of a Web REST API, where a schema is denoted by URIs and data is served as JSON objects. Data access and manipulation are modelled over HTTP GET,

DELETE and PUT methods. The common interface is supported by *handlers*, which are implementations of the `NonRelationHandler` interface. The handlers are responsible for providing operations for the system access methods. The modules and components are conceptually shown in Figure 8.1.

The authors of the system don't make it clear whether the system is intended to be deployed as a monolith (hinted in [131]), or if each handler and mapper pair is deployed as a separate service (hinted in [116]). This thesis optimistically takes the latter as the perceived best case for the system in terms of architectural flexibility.

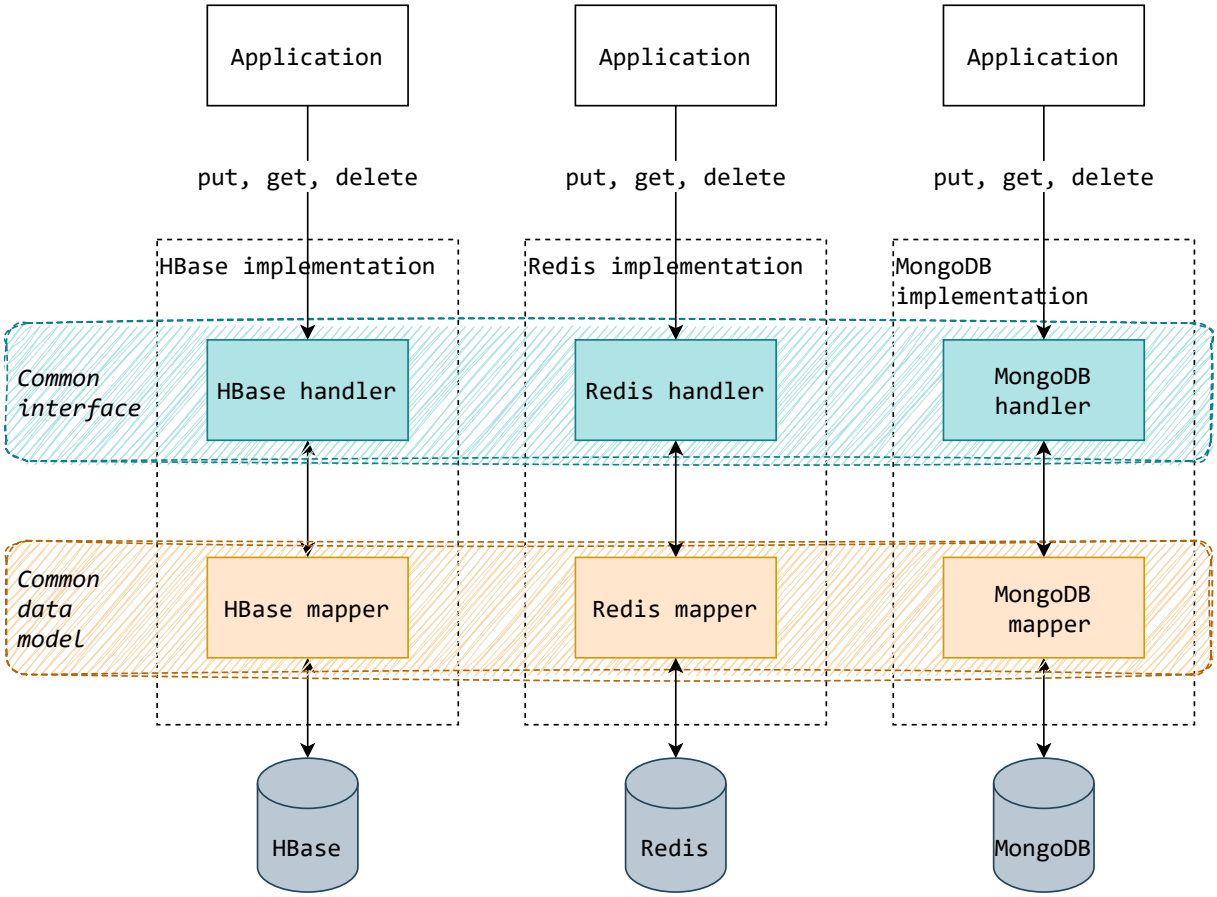


Figure 8.1: SOS system (adapted from [116])

The authors of the SOS system have proposed the possibility of it being a supporting element for data integration [131]. Although not a data source integration system, the SOS system could keep its raison d'être and be conveniently extendable to a data source integration system if it were reimagined following the MMW architecture. Additionally, using different mask kinds, the representational form of these data storages could be expanded on.

A hypothetical example of such a system use case is presented in Figure 8.2, where the revised SOS system provides uniform access over a single HBase, Redis, and MongoDB database. At the top of the component hierarchy, each database must be converted to the SOS interface for application access, as presented by [116]. This interface can be presented via mask components

of the SOS mask kind.

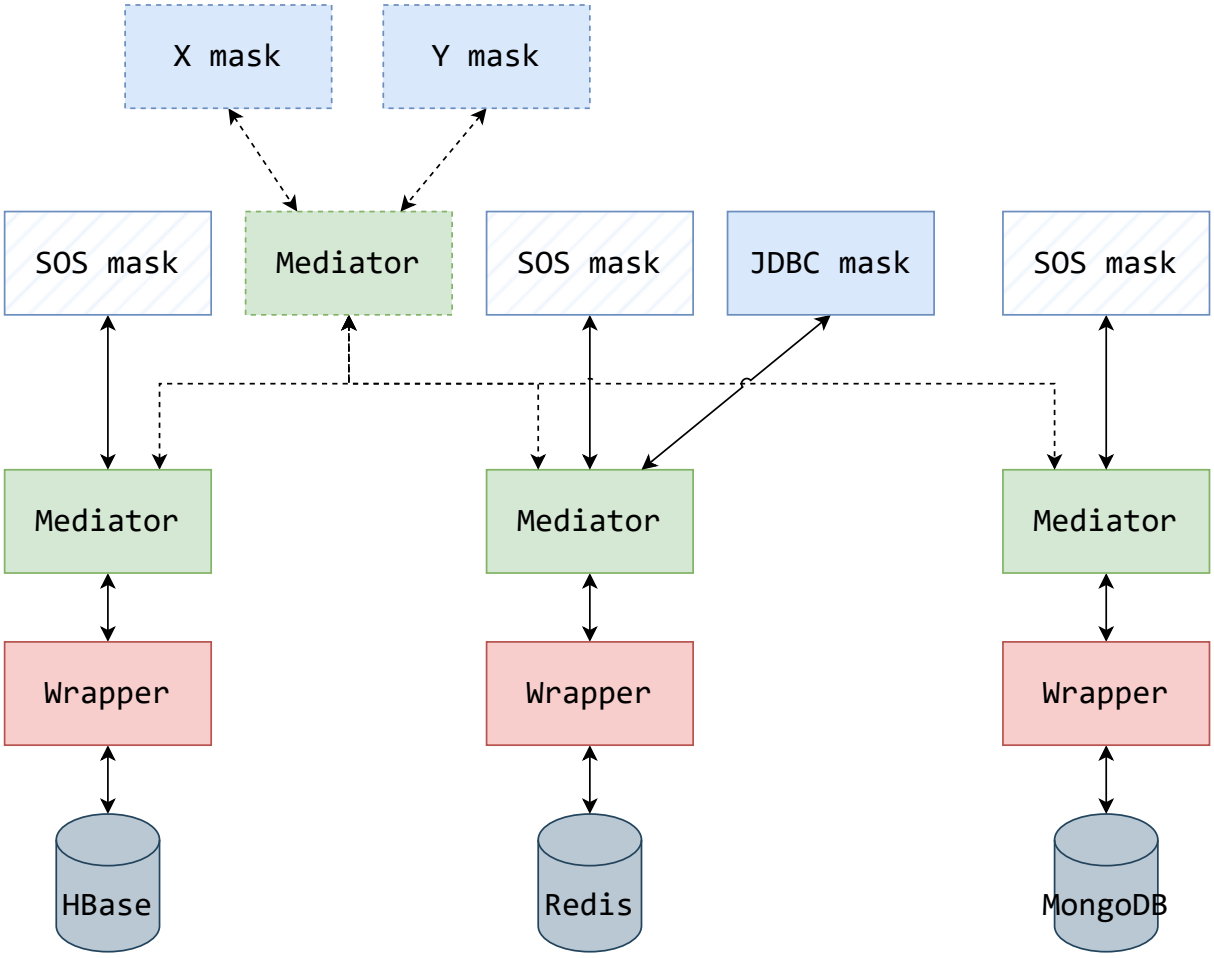


Figure 8.2: The SOS system emulated by the MMW architecture

Starting from the bottom of Figure 8.2 hierarchy, observing only solidly outlined elements, a wrapper is connected to each data source. Since it is not advisable to connect masks directly to wrappers and following **RMa2** stating that masks should only connect to mediators - a single mediator is connected to each wrapper. An SOS mask is connected to each of these mediators, thus encompassing the original SOS system's functionalities. These settings do not restrict the system to just using an SOS mask. Additional mask kinds can be connected to these singular mediators to offer an alternative presentational form and keep in line with the SOS use case of uniform system access. Such an example is given in Figure 8.2 with a JDBC mask representing the Redis database.

The singular mediators can be used to adapt the wrapped schemas, but also enable the system topology to be expanded. If by the example of the mediator shown with a dashed outline in Figure 8.2, a mediator connecting to each of the singular mediators is provided, then that part of the system becomes a data source integration system. Consequently, the integrating mediator allows connections coming from different masks.

Not only does the MMW architecture allow the emulation of a system, such as the SOS,

but it also expands on it. If it were not for the mask components taking on the responsibility of system representation, all mediators would have to be adapted (redesigned and reimplemented) to the SOS interface standard to work as both the SOS system and a data source integration system, which itself does not explicitly require adherence to the SOS interface standard. In effect, the use of the MMW architecture enables standardized implementation of mask kinds dedicated to data representations from Table 4.1 (illustrated in Figure 8.2 as masks X and Y).

8.2 Data mesh emulation case study

The data mesh has been exhaustively described in general terms of its capabilities and properties in Section 2.4. To show that a data mesh can be emulated with the MMW architecture, it must be shown that these properties can be satisfied through the MMW architecture. The essence of this case study is to show that the data mesh and MMW are compatible. These two concepts originate as solutions for distinct sets of problems, but they do close with the same conclusive ideas about how the systems should be built. It can be said that the data mesh and MMW are orthogonal concepts, but prove to be complementary; as one enriches the other.

The compatibility of the data mesh and MMW can be shown through four system aspects: adherence to the consume-transform-serve methodology [20], data models, evolvability, and coverage of capabilities. Through these aspects, the MMW is set to be shown as capable of satisfying them, since these aspects are already satisfied by the data mesh.

Consume–transform–serve

A data product is expected to autonomously consume, transform and serve data [20]. This was illustrated in Figure 2.21, where the input ports consume data, transformations are done inside the container, and the data is served at the output ports. In the MMW architecture, as illustrated conceptually by Figure 4.2, data is consumed by wrappers, transformed by mediators, and served by masks. By providing the general consume-transform-serve capabilities, the MMW architecture is capable of acting as a data product in some component settings.

Data model

Domain-driven design models are instinctively thought of as classes or structs. This is because domain-driven design is usually observed in operational systems*, which here is not the particular case. In analytical systems, this line of thinking might prove to be a trap, because

*Disambiguation on the operational and analytical system terms mentioned here:
An operational system is considered a system tasked with supporting the day-to-day business of an organization; primarily implemented via online transaction processing. An analytical system is considered a system tasked with supporting analytical requirements of an organization; primarily implemented via online analytical processing

the consumed data might be defined by different metamodels and it's expected that the data is served in polyglot form. Dehghani [68] also recognizes this by naming multiple possible forms of data the data mesh might consume and serve, as was cited in Section 2.4. Therefore, domain-driven design should be considered abstractly when it comes to modelling - decoupling it from a predetermined metamodel. Kleppmann [129] has noted the surprising lifetime of relational systems in the software industry: *relational databases turned out to generalize very well, beyond their original scope of business data processing, to a broad variety of use cases.* The relational model should fit the data mesh well. Nothing prevents a domain or a bounded context from being described in a relational model; this is done regularly in operational systems that use relational databases. A quasi-relational model could be used as a system format model in the MMW architecture. Asano et al. [85] already demonstrated a tabular model in mediators. Support for polyglot access can be significantly easier to research and implement, as there are already numerous implementations of relational mapping to other models. This is the key point for the MMW architecture's capability to provide domain-agnostic modelling, processing and data sharing across the organization, as proposed by Dehghani[20].

Evolvability

Technological capriciousness is one of the driving values of evolutionary architectures [17]. Dehghani [20] derives from this the suggestion that: "it's only appropriate to [...] leave the specific implementation details and technology to be refined and built over time". A data mesh should be evolvable, and it would be favourable if the MMW architecture followed this. The MMW architecture respects the dimensions of evolvability, mentioned in Section 2.1, in the following ways:

Technical

The inner components of the MMW component types are finely grained and their core functionalities can be decoupled from technologically changeable inner components by using the ports-and-adapters pattern [70]. Hence, the MMW components can be adapted to different technologies. This allows rapid development of support for newer or previously unsupported technologies.

Data

Both data and metadata are shared in an MMW system and are described in a generalized manner. Although alterations to the core metamodels are anticipated to be rare, their impact can be reduced by storing their definitions in a common library.

Security

Authentication, authorization, confidentiality and data integrity in communication can be decoupled along with communication protocols and data formats. Security in terms of system-concerned operations (e.g. sharing confidential system information or sending

unsafe data) can be assured through a common library describing behaviours for general exchanges and each component type.

Operational/System

Since each MMW component is an architectural quantum, it can be easily mapped onto existing infrastructure, providing great flexibility. The evolvability of the MMW architecture enables it to adhere to the data mesh's prescribed norms, which promote the interoperability of diverse technologies and ensure its long-term compatibility with the data mesh paradigm.

Capability coverage

The question of capability coverage boils down to the MWW architecture's ability to replicate the capabilities of the data mesh. Section 2.4 listed the data mesh capabilities, and these can be argued as covered as presented in Table 8.1.

Table 8.1: Coverage of data mesh capabilities by the MMW architecture

Scalable polyglot big data storage	The MMW architecture enables the utilization of heterogeneous data sources and various representations.
Encryption for data at rest and in motion	MMW component communication can be extended with encryption protocols. Encryption of static data essentially involves encrypting the local data storage.
Data product versioning	Data products can be versioned through a metamodel that supports versioning. Entire data schemas can be marked with a version indicator.
Data product schema	Since queries are defined over some form of schema, a data product schema is inherently included in each component.
Data product de-identification	De-identification can be supported by data transformation in mediators as data hiding, or by settings in wrappers.
Unified data access control and logging	Data access can be controlled at the component level. A mask can additionally provide access control through its application.
Data pipeline implementation and orchestration	Data pipelines are created by connecting (architecturally composing) the mask, mediator and wrapper components for data consumption, transformation and serving.

Data product discovery, catalogue registration and publishing	Data product discovery is provided by schema metadata (sources used to create the schema). Common interfaces simplify this process.
Data governance and standardization	Data governance is distributed because MMW components are quanta and can be grouped. Standardization is guaranteed by the MMW components' standard interfaces.
Data product lineage	Lineage can be examined by looking into the sources used for mediator transformations, and mask and wrapper translations.
Data product monitoring/alerting/log	Each MMW component can be deployed along with a monitoring application. As standard engineering practice, logging is expected to be component-level. Alerting mechanisms can be created as a part of a monitoring or management application.
Data product quality metrics (collection and sharing)	Schema metadata can carry quality metrics. This allows quality metric declaration per data product version.
In-memory data caching	There is no limitation to an in-memory data caching implementation to optimize request response times.
Federated identity management	Identity management is expected to be supported by the infrastructure.
Compute and data locality	Data can be placed in local data stores using materializing masks. The local data stores then must be consumed by a wrapper on request (this is illustrated in Section 8.2.1).

8.2.1 Emulating a data mesh

Figure 8.3 represents an example of an arbitrary data mesh which is emulated by the MMW components. The data mesh is constructed from a DIP and three domains that are generically named X, Y, and Z. Each of these domains contains an operational system and a DPC. The data products serve some exemplified analytical data. The operational systems are not emulated through the MMW components, since they are not primarily concerned with data management or analytical aspects of the data mesh. Hence, the focus of emulation is on the DPCs and DIP.

A DIP should enable uniform data access, resolving the need for duplicate data pipelines and storage [20]. Hence, it is fitting that the DIP is emulated by deploying wrappers over sources

the platform should cover. The data pipelines are emulated by connections to the wrappers, where the arduous data pipeline setup is substituted by a connection setup to a wrapper which wraps the data of interest. Data acquisition and transfer are handled by the protocols MMW components use for communication.

The simplest pattern of DPC emulation is illustrated in domain Z. A mediator is used to acquire and mediate the data from multiple DIP sources by connecting to the appropriate wrappers. DPCs access the DIP exclusively through the use of their mediator. A DPC wrapper connecting to a DIP source would be considered an anti-pattern, as it would infringe on the separation of concerns between the DIP and the DPC. A DPC wrapper is used to consume data coming from the domain's operational system. This wrapper is also mediated by the DPC mediator. The DPC mask is used to represent the mediated schema and data as a standardized API chosen for the data mesh. The data served by the mask is the domain's data product and can be used by the domain's operational system or served outside of the domain.

The domains X and Y reiterate the aforementioned pattern but with the addition of serving and consuming a data product from domain Y (annotated as D'_y in Figure 8.3). The DPC in a standard (non-MMW) data mesh implementation is expected to always access data products through a standardized API, the same as any other data product consumer. With the MMW, some corners can be cut. The mediator drives the data transformation to create a data product, hence the data exiting the mediator is the finished data product but in an MMW system format. This doesn't affect the value or cleanliness of the data product, simply its format; the mask only facilitates the translation. This claim is supported by rules **RMe3** and **RMa3**. A data product format translation is not required for it to be consumed by another MMW component - a receiving mediator from a remote domain. Conversely, another data mesh API wrapper would have to be deployed in the consuming DPC to acquire the data product. A mask and a wrapper would have to intercede the data product between the serving and consuming mediators. Additionally, this pattern would result in a requirement to deploy a separate wrapper for each consumed data product.

The example of Figure 8.3 illustrates a data mesh emulation where there are no locally stored data products. When a request for a data product is given, this triggers a set of queries throughout the architecture to acquire the data, construct the data product, and serve it. As was discussed in Section 2.4, some prepared domain data can be kept in a local data store to expedite data acquisition. The same functionality can be emulated by the MMW architecture. To facilitate this, the emulation of a DPC must be realigned with the requirement for a local domain data store.

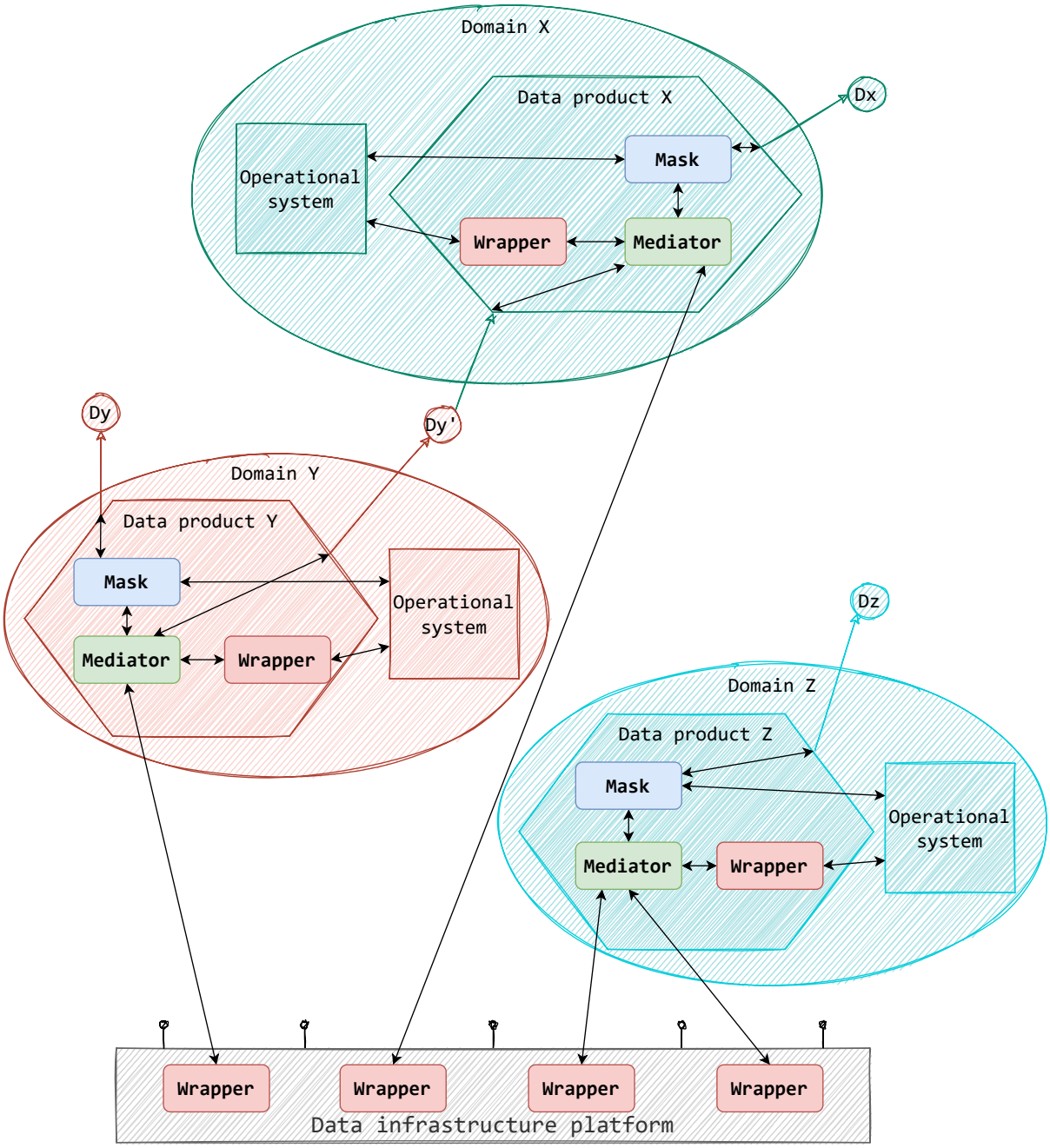


Figure 8.3: A data mesh emulated by MMW components

Figure 8.4 illustrates the case where a local domain data store is used by an MMW-emulated DPC. The DPC in this case is segregated into two component groups. The data ingress group is tasked with data acquisition and preparation. The ingress group consists of. The data egress component group is tasked with finalizing and serving the data. MMW components illustrated as being on the left-hand side of the DPC are the ingress group components, and the components on the right-hand side are the data egress group. As in the previous example, the DPC consumes data via its ingress mediator either from DIP wrappers, an operational system wrapper, or another DPC's egress mediator. The data is not immediately served by an egress component, rather a materializing ingress mask is used to create and populate a domain data store. The

domain data store can hold finalized data products or data that has been prepared for the final transformation into a data product by the egress components. On a request to the data mesh API mask (the egress mask), the request is interceded by the egress mediator and delegated to the egress wrapper. The egress wrapper consumes the prepared data, and the egress mediator either finalizes the data product creation or forwards the data to the mask. The egress mediator's responsibilities in a DPC are determined by the nature of the domain data store. The alternative situation is a request from another DPC, which is then received by the egress mediator itself and the aforementioned operations are executed analogously.

The ingress and egress component groups can work independently and asynchronously from one another. The ingress components can consume and store data in various modes, e.g. at time intervals, or on administrator requests. The egress components execute their tasks on request. Additionally, the egress mediator can include the ingress mediator in the data acquisition. This enables the acquisition of the newest data product possible, but requires increased computation. This is because the data product is at least partially prepared in the domain data store and should require lesser amounts of preparation for serving. This setting allows the MMW-emulated DPC to be more elastic than in a case when no intermediate domain data store is used.

This setting is revertable to the simple DPC pattern by completely bypassing the domain data store. This is achieved by configuring the egress mediator to send an exhaustive data request to the ingress mediator.

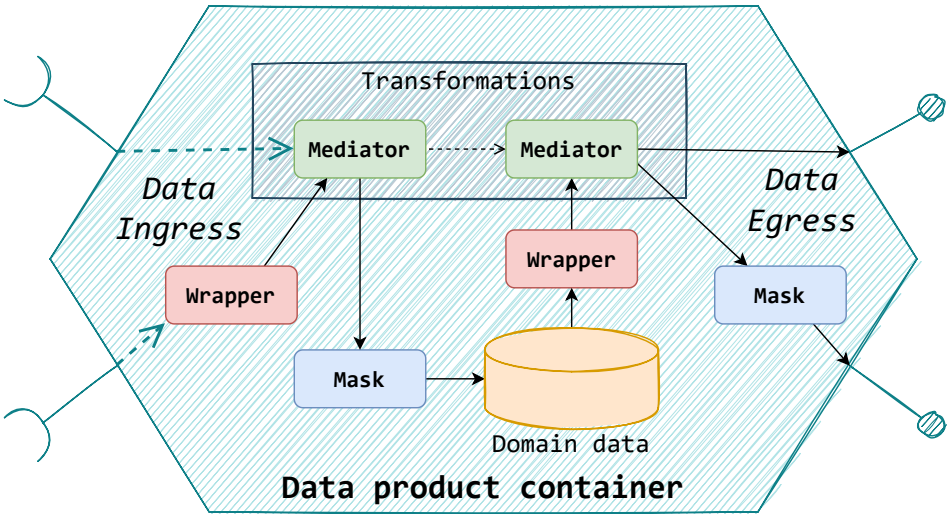


Figure 8.4: Data product with localized domain data emulated by MMW components (arrows denote flow of data as results of queries)

8.2.2 Expected benefits

The data mesh is a state-of-the-art concept and even Dehghani [20] refuses to set any implementational technology in stone. This is justified since the data mesh is observed through the

lens of evolvability, and such systems must be adaptable to changing technologies (see Section 2.1). Consequently, a lot of technological and organizational trial-and-error is expected from the first data mesh adopters. This is especially the case if the organization adopting the data mesh might not possess what would be data *at scale* [20] or might possibly not have the organizational prowess to implement and use it. Using the MMW architecture to emulate the data mesh is beneficial to organizations which are at risk but keen on using it to manage their data. The MMW-emulated data mesh enables:

Low-risk adoption trials

Organizations can set up a trial run of a data mesh in parallel with an existing analytical or as an initial solution. The risk is lowered in terms of:

Development failure

Development is required only if specialized components are required. The MMW-emulate data mesh construction boils down to deploying prebuilt components and editing their configurations to work as a data mesh.

Loss of large resource investment

No extensive coding is required to construct an MMW-emulated data mesh. A small technical team can prepare a demonstration in a short time since all components are prebuilt. The investment in terms of finance is reduced to the cost of additional infrastructure and work time required.

Deteriorated business usability

The MMW-emulated data mesh can run in parallel with an existing analytical and data management system. Deterioration of business usability is bounded by the duration of the trial run.

Rapid prototyping

Since the construction of an MMW-emulated data mesh involves deploying and configuring prebuilt components, the time to set the system up is minuscule. Once the MMW-emulated data mesh is deployed, its DPCs can be replaced piece by piece with permanently developed data mesh components. The DIP can also be gradually converted into a more permanent infrastructure since it is emulated by a collection of wrappers.

Evolvability

The MMW architecture guarantees evolvability, so any system emulated with it will inherently be evolvable. Accordingly, MMW-based emulation can be used to guarantee evolvability.

Standardization

The MMW architecture can be used to standardize the architectural quanta of the data mesh, which lowers the cognitive load of developers by using known experience, languages, and APIs [20]. Standardization also contributes to the composability of software

systems.

8.3 Case study prototypes

8.3.1 Janus as a data source integration system

The MMW architecture was primarily envisioned as a data source integration architecture. To prove that the MMW architecture is capable of producing an architectural topology acting as a data source integration system, the Janus system's components were containerised and deployed as illustrated in Figure 8.5. The prototyped data source integration system integrates the data represented by the logical model in Figure 8.6.

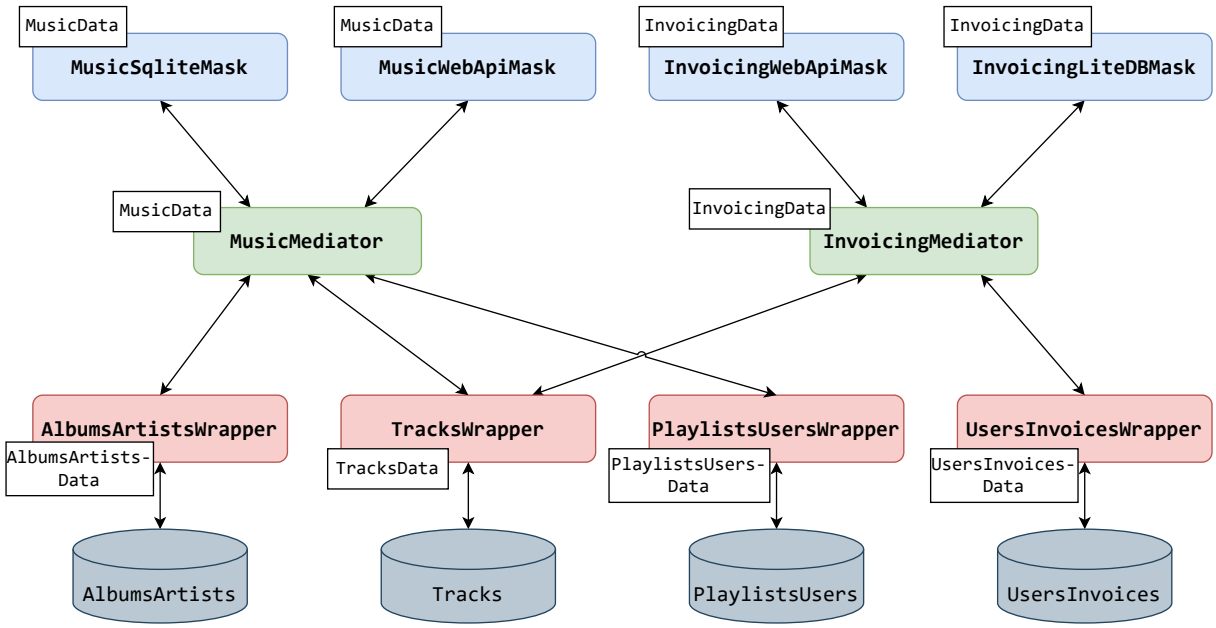


Figure 8.5: Architectural topology of an example data source integration system driven by the MMW architecture

An SQLite sample database, called *chinook*, was taken as the primary data source for the case study prototype [132]. The sample database was separated into four thematic databases. The *AlbumsArtists* database contains data about music albums and artists. The *Tracks* database contains data about music tracks. The *PlaylistsUsers* database contains data about users' music playlists. The *UsersInvoices* contains billing data for users according to their played music tracks. The *customers* table can be found with replicated data in both the *PlaylistsUsers* and *UsersInvoices* databases. This is to show that the entire schema encompassed by the integration system might not be normalised. The four databases are represented by the logical model diagram in Figure 8.6. Figure 8.6 has the tables of the *AlbumsArtists* databases coloured blue, tables of the *Tracks* coloured orange, tables of the *PlaylistsUsers* database coloured red,

and the tables of the *UsersInvoices* database coloured green. The *customers* table is replicated in the *PlaylistsUsers* and the *UsersInvoices* database. Wrapper components are deployed and configured for each database and are named according to the database they wrap: *AlbumsArtistsWrapper*, *TracksWrapper*, *PlaylistsUsersWrapper*, and *UsersInvoicesWrapper*. Mediators are named according to the topics of their mediated schemas: *MusicMediator* and *InvoicingMediator*. Mask components are named according to their kind and the topic they represent: *MusicSqliteMask*, *MusicWebApiMask*, *InvoicingWebApiMask*, and *InvoicingLiteDBMask*. The component naming is established by setting the component communication node identifiers in their respective configurations. All of the MMW components are typified as Web applications.

Reflecting on the schema hierarchies presented as a tool for the qualitative analysis in Sections 2.3.2 and 4.2.1, the topology of this case study prototype can be discussed in those terms. The schemas of the underlying databases are LISs, in their native format. The schemas in the wrappers (*AlbumsArtistsData*, *TracksData*, *PlaylistsUsersData* and *UsersInvoices*) are LCSs translated to the system format. The mediators join their respective connected LCSs into GCSs (*MusicData* and *InvoicingData*). Masks translate the GCSs to LESs (*MusicData* and *InvoicingData*) by translating the schemas to the appropriate representational formats.

Following the illustrations of schema assignments in the qualitative analysis of the MMW architecture and its predecessor (Sections 2.3 and 4.2.1), Figure 8.5 accordingly conveys the placement of system schemas as white rectangles next to their respective component.

To achieve the topology depicted in Figure 8.5 a Docker compose was specified with the suitable source connection strings, mediation scripts, component startup options and communication settings. The SQLite databases intended for integration were placed inside their respective wrapper containers. The system component containers were placed in a virtual Docker network running in bridge mode. Individual component access points exposed by the Docker compose are detailed in Table 10.2. The Docker compose specification and accompanying files for containerisation are provided in the Janus system's git repository [27].

This case study prototype shows that an MMW system can be used as a data source integration system. The topology demonstrates the capabilities provided by adding a mask component, where each mediated schema is represented by masks of different kinds. The case study prototype also demonstrates that virtualising and materialising masks can drive the integration system in different modalities. An additional benefit provided by the Janus system is that the schema elements have been translated to a singular naming convention, whereas the source databases used *snake case* to name tables, and *Pascal case* to name columns.

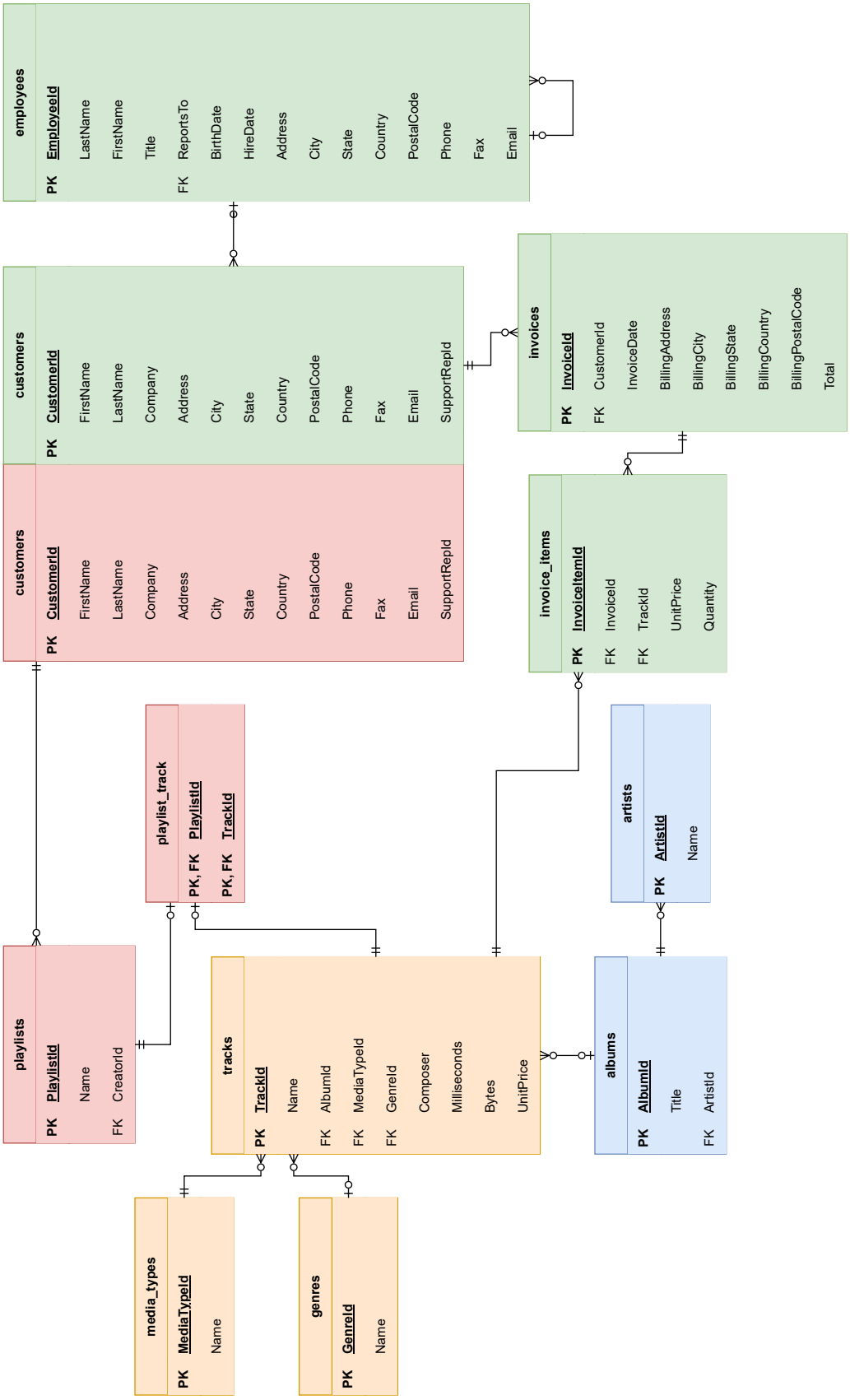


Figure 8.6: Unified logical model diagram of the sample databases used in the case study prototype (colouring: *UsersAlbums* - blue; *Tracks* - orange; *PlaylistsUsers* - red; *UsersInvoices* - green)

A detailed description of the schema translations in this case study prototype, providing additional details on the schema translation mechanisms, can be found in the Appendix in Section 10.5.1. Activities undertaken by the Janus system components while executing queries (sending requests to the masked Web API) are detailed in the Appendix Section 10.5.2.

8.3.2 Janus for SOS system emulation

It was discussed during the qualitative analysis in Section 8.1 that the MMW architecture could emulate the SOS system to preserve legacy data stores by representing them as REST Web APIs. To prove that the MMW architecture is capable of producing an architectural topology emulating the SOS system, the Janus system's components were containerised and deployed as illustrated in Figure 8.7. The databases from the data source integration system case study prototype (Figure 8.6) were used as surrogates of legacy data stores.

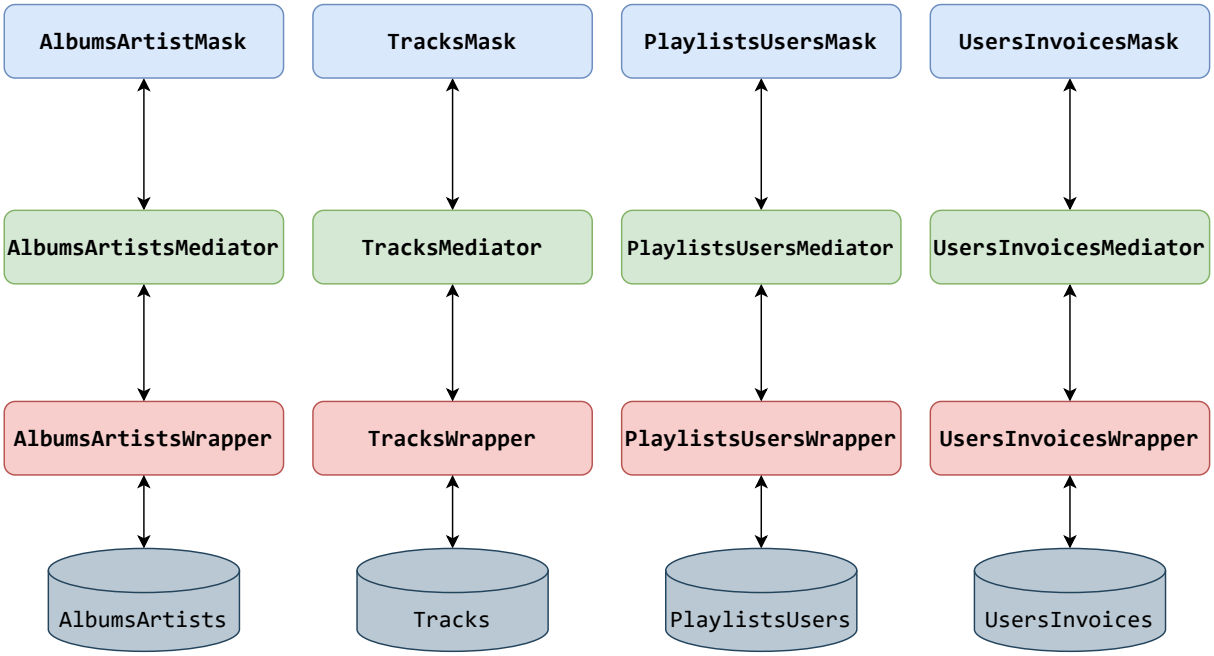


Figure 8.7: Architectural topology of the SOS system emulated by the MMW architecture

As illustrated in Figure 8.7, each database is assigned a silo of MMW components to translate its schema to the REST Web API format. This topology follows the idea discussed in Section 8.1, with the omission of additionally proposed components. Each component is named according to its component type and the database it is assigned to. For example, the *AlbumsArtistsWrapper* wraps the *AlbumsArtists* database, the *AlbumsArtistsMediator* adapts the schema, and the *AlbumsArtistsMask* represents the adapted schema as a REST Web API. Adaptation in this case is meant in terms of standardising naming conventions for schema elements. The schema model data sources were named after their originating databases. All MMW components in this case study prototype are typed as Web applications.

To achieve the topology depicted in Figure 8.7 a Docker compose was specified with the suitable data source connection strings, mediation scripts, component startup options and communication settings. The SQLite databases mimicking legacy data stores were placed inside their respective wrapper containers. The system component containers were placed in a virtual Docker network running in bridge mode. Individual component access points exposed by the Docker compose are detailed in Table 10.3. The Docker compose specification and accompanying files for containerisation are provided in the Janus system's git repository [27].

The case study prototype shows that an MMW architecture system, namely Janus, can emulate the SOS system. Thus, the case study prototype provides empirical proof for the assumptions made in the qualitative analysis regarding the SOS system's emulation via an MMW architecture system. Janus enables the creation of a standardized SOS mask, which is familiarly represented as a Web API mask kind in this case study prototype. The case study prototype's topology can also be expanded to a data source integration system with other representation formats by adding mediator and mask components, as already exemplified in the case study prototype of Section 8.3.1.

8.3.3 Janus for data mesh emulation

This case study prototype proves the theorized capability of the MMW architecture to emulate a data mesh. The operational goal of this case study prototype is to emulate a data mesh with the topology presented in Figure 8.8. The topic of the domain for this case study prototype is a music streaming platform. The case study prototype data mesh has a DIP with data sources containing data about music, countries, customers, listening, media types of track instances, and customer subscription plans. The data is separated into bounded contexts presented by DPCs concerning music, customers, customer listening history, and customer subscription plans. The DIP data sources are illustrated with a logical model diagram in Figure 8.9 The DPCs in this case study prototype are not tied to any operational systems' data for the sake of clarity, because wrappers over operational data stores mirror the requirement for wrappers emulating the DIP.

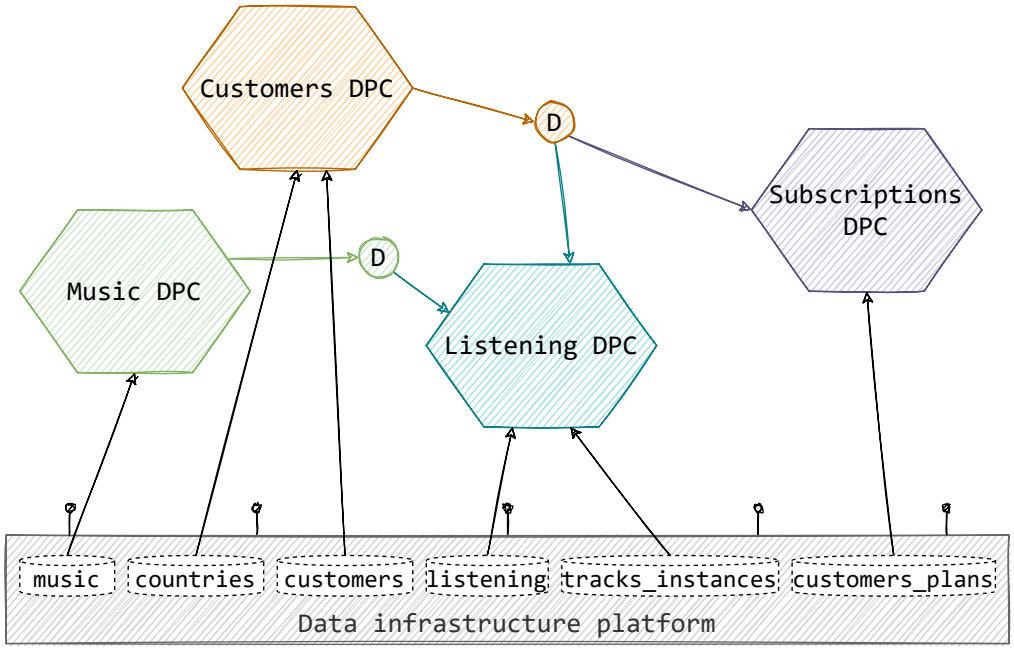


Figure 8.8: Topology of the data mesh to be prototypically emulated

SQLite databases were created to act as DIP data sources. The databases were created using Python scripts with surrogate data generated by the *Faker* library. The Janus system was used as a representative MMW architecture system. Web API masks were used for serving data to entities outside of the data mesh, while SQLite wrappers were used to uniformly represent the SQLite databases in the DIP. Since the Listening DPC contains the most complexity in terms of required data sources it is presented in Figure 8.10 as an example of an emulated DPC via Janus components.

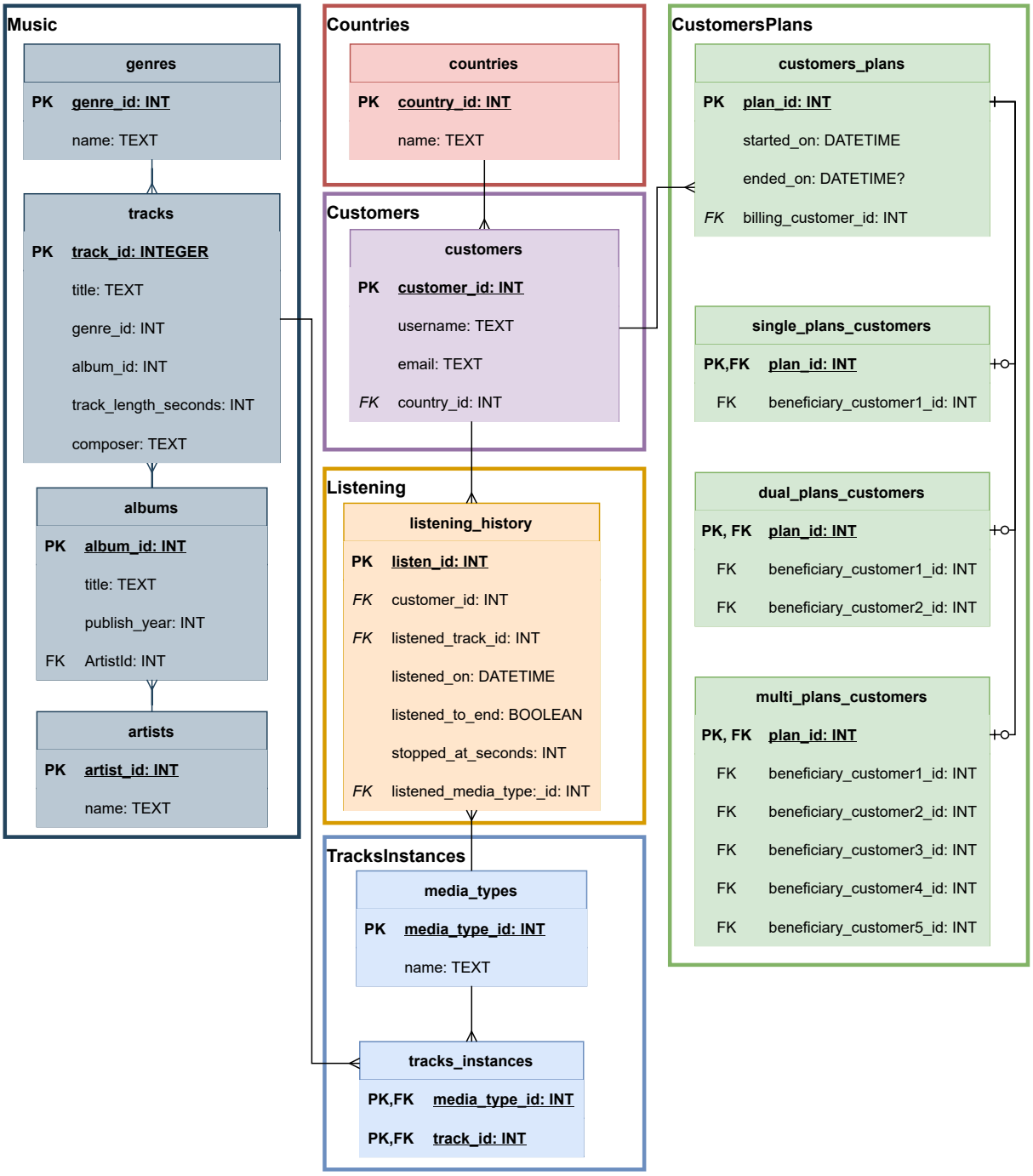


Figure 8.9: Logical model diagram of the databases used as DIP data sources in the data mesh emulation case study prototype

The DIP *listening* and *tracks_instances* data sources used by the *Listening* DPC are wrapped by their respective wrappers, inferring the *ListeningData* and *TracksInstancesData* schemas. The DIP wrappers are of the SQLite kind and are configured not to allow commands on their data sources - they are read-only. The *Music* and *Customers* DPCs provide their data products to the *Listening* DPC from their egress mediators; their schemas are accordingly named *MusicData* and *CustomersData*. The *Listening* DPCs ingress mediator mediates the aforementioned schemas and creates the *ListeningIngressData* mediated schema. The *ListeningIngressData*

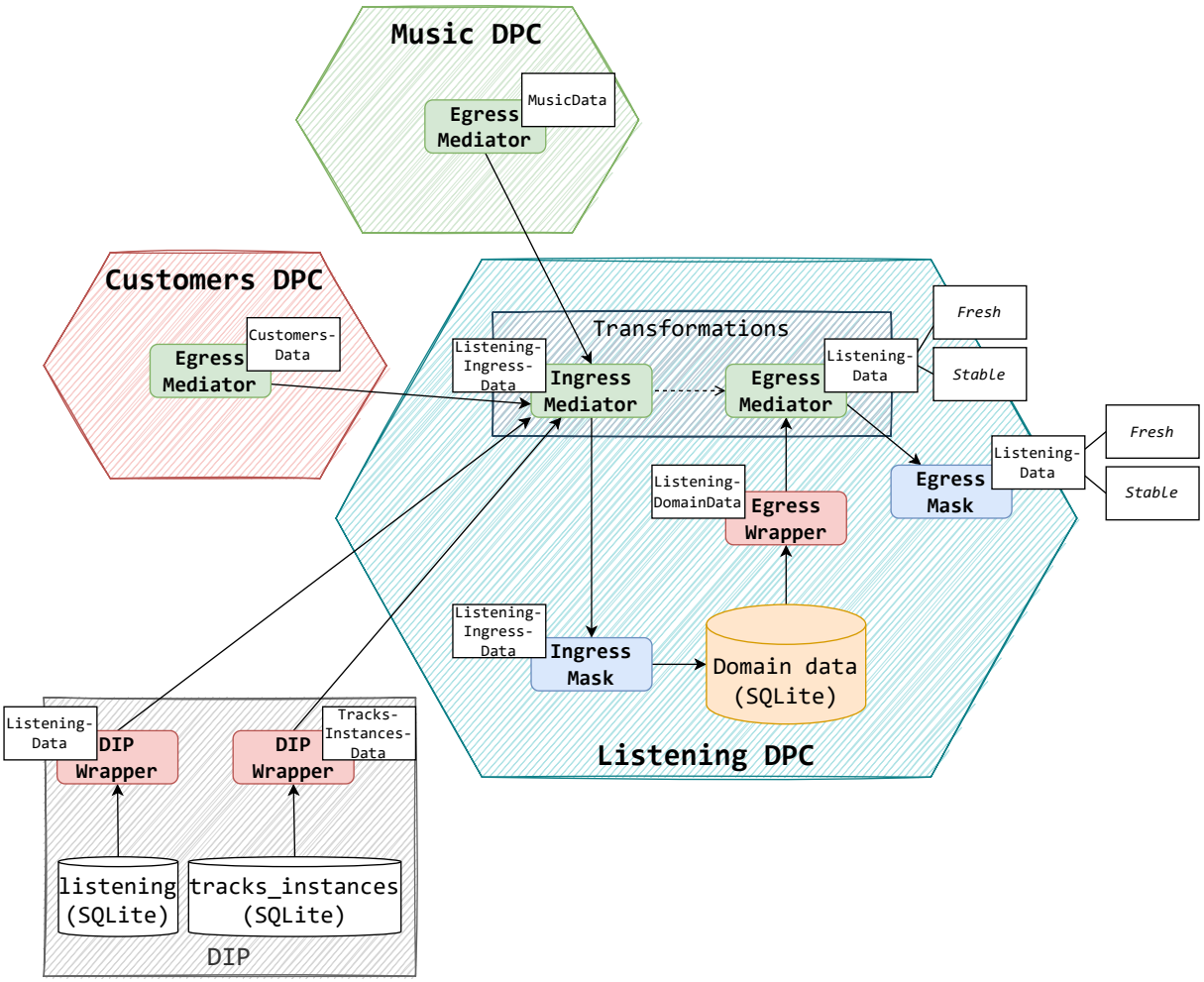


Figure 8.10: Listening DPC emulated by the Janus system with its adjacent emulated DPCs and DIP

schema is loaded by the *Listening* DPC's ingress mask, which is an SQLite materialising mask. The ingress mask materialises the domain data store (data product) as an SQLite database. To serve the data product, the SQLite egress wrapper wraps the domain data store and infers from it the *ListeningDomainData* schema. The *Listening* DPC's egress mediator loads the schemas from the egress wrapper and the ingress mediator. The two loaded schemas are then mediated into the *ListeningData* schema, which can be used for serving the data product. The *ListeningData* schema is instantiated as a `DataSource` schema model in the Janus system. The `DataSource` is declared through mediation to have two `Schema` instances; one under the name *Stable* representing the data acquired from the domain data store, the other under the name *Fresh* representing the data acquired directly from the ingress mediator. The *Stable* and *Fresh* schemas contain the same tableaus and attributes, but their sources differ. The *Fresh* schema provides the most recent data, and the *Stable* schema provides a preprocessed data snapshot which can be acquired more quickly and with less computational load. The *Listening* DPC's egress mask is a Web API kind mask, and it represents as such the *ListeningData* schema. The *Fresh* and *Stable* schemas are represented as separate URL route prefixes `/Fresh` and

`/Stable`. For example, to acquire the preprocessed listening history of a customer with the user name "johnsmith" the URL for a GET request, sans the domain, would be: `/Stable/ListeningHistory?UserName=johnsmith`. The concrete Web API routes of the Listening DPC provided by the egress mask are shown in Table 8.2, as well as an example of the served data in the JSON format is shown in Listing 8.1

Table 8.2: Routes found in the Listening DPC's egress mask Web API

Route	**HTTP method**
`/Fresh/ListeningHistory/{id}`	`GET`
`/Fresh/ListeningHistory?{query_string}`	`GET`
`/Stable/ListeningHistory/{id}`	`GET`
`/Stable/ListeningHistory?{query_string}`	`GET`

```
{
    "ListenId": 1,
    "CustomerUserName": "wendyjennings",
    "ListenedTrackId": 499,
    "ListenedTrackName": "Au Privave",
    "ListenedOn": "2023-09-21T12:56:24"
    "ListenedToEnd": true,
    "StoppedAtSeconds": 16,
    "ListenedMediaType": "AAC"
}
```

Listing 8.1: Example of a JSON representing a listening history item provided by the Egress mask of the Listening DPC

To achieve the topology depicted in Figure 8.8 a Docker compose was specified with the suitable data source connection strings, mediation scripts, component startup options and communication settings. The SQLite databases mimicking DIP data sources were placed inside their respective wrapper containers. The system component containers were placed in a virtual Docker network running in bridge mode. Materialised SQLite databases posing as domain data were placed in virtual volumes, each specifically used by their respective DPC services. Individual component access points exposed by the Docker compose are detailed in Tables 10.4, 10.5, 10.6, 10.7, and 10.8, along with the names of the data sources the components infer, mediate, or represent. The Docker compose specification and accompanying files for containerisation are provided in the Janus system's git repository [27].

The case study prototype shows that an MMW architecture system, namely Janus, can emulate a data mesh. Therefore, the case study prototype provides empirical proof for the assumptions made in the qualitative analysis regarding the data mesh's emulation via an MMW architecture system. Janus components were demonstrated to simultaneously emulate multiple DPCs and a DIP by using its configurable MMW components. It was demonstrated that Janus can emulate a DPC containing a local domain store. The case study prototype also presented the naming convention for schemas and MMW components when emulating a data mesh.

Chapter 9

Conclusion

This thesis presented research involving the extension of the MW architecture to facilitate the addition of various representational system access interfaces. The motivation for this research was to increase the flexibility and capability of heterogeneous data source integration systems and to provide a framework for the development of a comprehensive future system of this type. The MW architecture has been shown to underperform through a qualitative analysis when requirements for various data access interfaces arise. The qualitative analysis was facilitated by a discussion and study on schema hierarchy deployments, where schemas from an exemplified schema hierarchy were deployed to architectural components. The individual MW architecture components were exhibited as having to manage multiple schemas, clearly not adhering to the concept of responsibility separation. The outlook for future exacerbation of the problem is discussed through the trend of increasing variety in data representation. This presents the primary research problem.

To solve the problem, the research theoretically introduced a new component type specifically intended for representational purposes, called a *mask*. Due to the introduction of a new component type, the extended architecture proposed in the presented research was named the *mask–mediator–wrapper* architecture.

The research produced a set of rules for the mask component in accordance with the existing rules for mediators and wrappers, additionally providing a new rule for mediators to further distil their roles in the architecture. Just like the mediator and wrapper components, the mask is also considered a generic component - allowing prefabrication and configurable deployment in multiple topologies. The mask's functionalities were analysed and its design was discussed through an analysis of the mask's rules, functional requirements inferred from those rules, expected data flow and processes implied by the functional requirements, and the inner component design determined from the previous analyses. The findings of this analysis suggested that the implementation of a mask kind varied only in terms of the implementations of translator components, while the remaining inner components can be implemented generically. Since the mask's

implementation is parametrized by these implementations, it was proposed that masks could be uniformly developed through a mask framework. The framework was proposed to be able to create a library for a mask kind, which would then be used to develop an application representing a mask type. The ability to provide such a framework has posed another research problem. The MMW architecture and mask component were only discussed theoretically at this point.

A quantitative shift-cost analysis was conducted to prove that the theoretic architectural extension doesn't negatively impact a data source integration system and is beneficial when introducing new representations. The quantitative analysis showed that the MMW architecture is more flexible than the existing modes of the MW architecture in scenarios requiring the addition of new representations (existing or entirely new), and scenarios requiring the addition of another mediation. The scenario of including a new data source is determined to have a shift-cost on par with the existing MW architecture modes.

Case studies were developed to hypothetically demonstrate the flexibility and versatility of the proposed architecture. The case studies were concerned with the emulation of two additional data management systems other than a data source integration system. The SOS system, intended for persevering legacy data stores, was shown as emulated by MMW components. Such an emulation enabled additional flexibility in terms of adding representation capabilities and adding mediation to the existing architectural topology. The state-of-the-art data mesh architecture was the focus of the second emulation case study. The MMW components were theoretically shown as being capable of emulating the DIP and DPC components of a data mesh. The MMW components were also shown as being capable of emulating a DPC containing a local domain data store.

The translations in a mask were theorised to include one-way transformations of queries (including commands) and schemas. On the other hand, the translation of data included in mutations, insertions, and query results requires two-way transformations that guarantee some level of correctness. This raised the research question of the ability to minimize the effort required of implementing two-way transformations in a mask while maintaining the ability to reason about the correctness of the implemented transformations. The examination of the field of bidirectionalisation resulted in the choice of lenses to facilitate these transformations. A simple symmetric lens was concretely chosen and introduced as a design pattern through an implementation in C#. Implementing two-way transformations in the form of simple symmetric lenses enabled the transformations to be reused, even for constructing complex lenses. The reusability of design patterns, in general, reduces implementational effort. An additional reduction in the implementational effort was introduced by providing a behavedness-level testing framework for the implemented simple symmetric lenses.

Although the expected contributions of this research permitted further investigation through a hypothetical heterogeneous data source integration system, a prototypal system was expected

to provide better solidification of the contributions. The *Janus* system acted as a prototype to substitute a hypothetical system, making the contributions tangible and provable. Janus was developed as an open-source prototype MMW system, enabling further prototyping, experimentation and proofs regarding the proposed architecture and component design. Janus provides the proof-of-concept for the hypothesised mask framework. The Janus mask framework was used to expeditiously develop three mask kinds and their respective applications: a REST Web API virtualizing mask, a LiteDB materialising mask, and an SQLite materialising mask. The process of a mask's implementation (in the Janus system) was concluded to include nine general steps. Janus also enabled the proof for the usability of lenses in masks for two-way transformations. The lens providing bidirectional transformations between DTO objects and tabular data was utilised in the Janus system's REST Web API mask.

The Janus system enabled the creation of a case study prototype for an MMW heterogeneous data source system, showing that such a system is achievable. Two additional case study prototypes over the case studies regarding the SOS system and data mesh were also conducted. The case studies' prototypes showed that the MMW architecture is capable of emulating other systems and not just acting as a data source integration system. These results, especially in the case of the data mesh, were not initially expected but have enriched the results of this research. They provide additional insight into the nature of the MMW architecture, raising a future research question on the feasibility of data management systems constructed through pre-built configurable components.

The true aspiration of this research warrants a note. The ulterior motive of this research was also the development of a pathfinding system as a stepping stone towards a comprehensive, evolvable, extendable, and open system for heterogeneous data management that can be a foundation for future research.

Chapter 10

Appendix

10.1 List of abbreviations

The thesis text uses some abbreviations that are commonly used in the covered scientific and engineering fields. The fully-qualified name of such abbreviations is omitted in the text, while abbreviations of terms introduced in the thesis are parenthesised at first appearance. Table 10.1 presents the fully-qualified names of the abbreviations.

Table 10.1: Abbreviations used in the text

Abbreviation	**Fully-qualified name**
1LMW	mediator–wrapper architecture with one mediator layer
2LMW	mediator–wrapper architecture with two mediator layers
API	application programming interface
BSON	binary javascript object notation
BX	bidirectional transformations or bidirectionalisation
CLI	command line interface
DIP	data infrastructure platform
DLL	dynamic-link library
DPC	data product container
DTO	data transfer object

HTTP	hypertext transfer protocol
HTTPS	hypertext transfer protocol secure
IL	.NET intermediate language
JDBC	java database connection
JSON	javascript object notation
MW	mediator–wrapper
MMW	mask–mediator–wrapper
NoSQL	not only SQL
OOP	object-oriented paradigm
REST	representational state transfer
SOS	save our systems
SSL	secure sockets layer
SUS	system under study
TCP	transmission control protocol
URI	uniform resource identifier
URL	uniform resource locator
XML	extensible markup language

10.2 Code example prefix

To run Haskell code examples given in Chapter 3, they should be prefixed in a programming environment with the following helper data types and functions:

```
1 --assertionresults
2 dataAssertion a = Success a | Failure a
3 --assertHoF;takesafunctiontoassertonanobjectfromparam2
4 assert :: (a ->Bool) -> a -> Assertion a
5 assert f x =iff xthen
6                     Success x
```

```
                    else
                        Failure x
--showimplementationallowssimplifiedprinting
instanceShowa=>Show(Assertion a)where
    show(Success x) ="Assertionsucceeded:"++showx
    show(Failure x) ="Assertionfailed,actualvalue:"++showx
```

Listing 10.1: Prefix functions for code listings in Haskell

`Assertion` is a data type that signifies if an assertion (provided by a truth-testing function) of an assumption was found to be true or false. The `Assertion` is created by calling the `assert` high-order function that applies a truth-testing function and the object over which the testing is to be undertaken. The instance of `Show` is provided to simplify and neaten the printing of assertion outcomes.

10.3 Running the case studies' prototypes using Docker compose

The required files for the containerisation of the Janus system mentioned in this thesis (primarily in Sections 8.3.1, 8.3.2, and 8.3.3) can be found in the Janus git repository on Github [27]. The files are located in the *experimentation* directory, which contains separate directories for each emulated system. The prerequisites for running the case study prototypes are the installations of Docker and Docker Compose. The containerisation files can be downloaded by using the `git clone` shell command. After the repository files are acquired, it is required to navigate to the desired case study's directory. The required Docker container images must be built, advisably without using cache via the `docker compose build` command. The images can be started as containers by using the `docker compose up` command. When running computationally demanding case study prototypes, it is advisable to specify only the required services of the Docker compose specification. The Docker compose services for each individual component can be configured in the specification file under the name `docker-compose.yml`. Mediation scripts used in the case study prototypes can be found in the `mediation_scripts` directory of each case study, as well as the SQLite databases under the `databases` directory. The shell commands required to run the data source integration, SOS system, and data mesh prototypes from Sections 8.3.1, 8.3.2, and 8.3.3 are provided in Listings 10.2, 10.3, and 10.4 respectively.

```
$> git clone https://github.com/JurajDoncevic/Janus.git --branch
   ↪ master
$>cd./experimentation/janus_integration
$> docker compose build --no-cache
$> docker compose up
```

Listing 10.2: Running the Docker compose to initialise the data source integration prototype from Section 8.3.1

```
$> git clone https://github.com/JurajDoncevic/Janus.git --branch
   ↪ master
$>cd./experimentation/janus_sos
$> docker compose build --no-cache
$> docker compose up
```

Listing 10.3: Running the Docker compose to initialise the SOS prototype from Section 8.3.2

```
$> git clone https://github.com/JurajDoncevic/Janus.git --branch
   ↪ master
$>cd./experimentation/janus_data_mesh
$> docker compose build --no-cache
$> docker compose up
-- or just the listening DPC
$> docker compose up -d listening_egress_mask
```

Listing 10.4: Running the Docker compose to initialise the data mesh prototype from Section 8.3.3

Depending on the computational power of the host system used for running the containers, it might be required to adjust the startup times and timeout settings of individual components. The case where an egress wrapper starts inferring a schema not yet materialised by the ingress mask is most common. The startup timing of the components' containers can be adjusted with the `SLEEP_START` argument in the `docker-compose.yml` service specification, while the timeout settings can be adjusted with the `TIMEOUT_MS` argument.

Table 10.2: Component settings for the Janus-driven data source integration system

Component name	**Component listen port**	**Web management port**	**IP address**	**Web API port / materialization path**
`AlbumsArtistsWrapper`	10001	8101	172.24.1.1	-
`TracksWrapper`	10002	8103	172.24.1.2	-
`PlaylistsUsersWrapper`	10003	8105	172.24.1.3	-
`UsersInvoicesWrapper`	10004	8107	172.24.1.4	-
`MusicMediator`	20001	8201	172.24.2.1	-
`InvoicingMediator`	20002	8203	172.24.2.2	-
`MusicWebApiMask`	30001	8301	172.24.3.1	8801
`InvoicingWebApiMask`	30002	8303	172.24.3.2	8803
`MusicSqliteMask`	30003	8305	172.24.3.3	./music.db
`InvoicingLiteDBMask`	30004	8307	172.24.3.4	./invoicing.db

10.4 Configuration tables for the case studies' prototypes

10.4.1 Data source integration system case study prototype configuration

10.4.2 SOS system case study prototype configuration

Table 10.3: Component settings for the SOS system emulation by Janus

Component name	Component listen port	Web management port	IP address	Web API port
`AlbumsArtistsWrapper`	10001	8101	172.25.1.1	-
`TracksWrapper`	10002	8103	172.25.1.2	-
`PlaylistsUsersWrapper`	10003	8105	172.25.1.3	-
`UsersInvoicesWrapper`	10004	8107	172.25.1.4	-
`AlbumsArtistsMediator`	20001	8201	172.25.2.1	-
`TracksMediator`	20002	8203	172.25.2.2	-
`PlaylistsUsersMediator`	20003	8205	172.25.2.3	-
`UsersInvoicesMediator`	20004	8207	172.25.2.4	-
`AlbumsArtistsMask`	30001	8301	172.25.3.1	8801
`TracksMask`	30002	8303	172.25.3.2	8803
`PlaylistsUsersMask`	30003	8305	172.25.3.3	8805
`UsersInvoicesMask`	30004	8307	172.25.3.4	8807

10.4.3 Data mesh case study prototype configuration

Table 10.4: Janus components settings for DIP emulation

Component name (suffixed with `_DIPWrapper`)	**Component listen port**	**Web management port**	**IP address**	**Data source name**
`Customers`	11001	8101	172.26.1.1	CustomersData
`Countries`	11003	8103	172.26.1.3	CountriesData
`CustomerPlans`	11004	8104	172.26.1.4	CustomersPlansData
`Listening`	11005	8105	172.26.1.5	ListeningData
`Music`	11007	8107	172.26.1.7	MusicData
`TracksInstance`	11008	8108	172.26.1.8	TracksInstancesData

Table 10.5: Janus components settings for Music DPC emulation

Component name (prefixed with: `Music_`)	**Component listen port**	**Web management port**	**IP address**	**Data source name**
`IngressMediator`	22001	8201	172.26.2.1	MusicIngressData
`IngressMask`	32001	8203	172.26.2.2	MusicIngressData
`EgressWrapper`	12001	8205	172.26.2.3	MusicDomainData
`EgressMediator`	22002	8207	172.26.2.4	MusicData
`EgressMask`	32002	8209	172.26.2.5	MusicData

Table 10.6: Janus components settings for Subscriptions DPC emulation

Component name (prefixed with: `Subscriptions_-`)	**Component listen port**	**Web management port**	**IP address**	**Data source name**
`IngressMediator`	23001	8301	172.26.3.1	SubscriptionsIngressData
`IngressMask`	33001	8303	172.26.3.2	SubscriptionsIngressData
`EgressWrapper`	13001	8305	172.26.3.3	SubscriptionsDomainData
`EgressMediator`	23002	8307	172.26.3.4	SubscriptionsData
`EgressMask`	33002	8309	172.26.3.5	SubscriptionsData

Table 10.7: Janus components settings for Customers DPC emulation

Component name (prefixed with: `Customers_`)	**Component listen port**	**Web management port**	**IP address**	**Data source name**
`IngressMediator`	24001	8401	172.26.4.1	CustomersIngressData
`IngressMask`	34001	8403	172.26.4.2	CustomersIngressData
`EgressWrapper`	14001	8405	172.26.4.3	CustomersDomainData
`EgressMediator`	24002	8407	172.26.4.4	CustomersData
`EgressMask`	34002	8409	172.26.4.5	CustomersData

Table 10.8: Janus components settings for Listening DPC emulation

Component name (prefixed with: `Listening_`)	**Component listen port**	**Web management port**	**IP address**	**Data source name**
`IngressMediator`	25001	8501	172.26.5.1	ListeningIngressData
`IngressMask`	35001	8503	172.26.5.2	ListeningIngressData
`EgressWrapper`	15001	8505	172.26.5.3	ListeningDomainData
`EgressMediator`	25002	8507	172.26.5.4	ListeningData
`EgressMask`	35002	8509	172.26.5.5	ListeningData

10.5 Additional details on the data source integration system case study prototype

10.5.1 Schema translation

Section 8.3.1 omitted the specifics of the schema translation and mediation implemented in the case study prototype. This appendix section delves into more detail about the schemas that were inferred in the wrappers from the data sources, the mediation that was applied, how it reflected on a mediated schema, and how the mediated schema was represented as a Web REST API. This detailed overview shows just the schemas and system components used to generate the Web API *MusicData* representation. Hence, the included components are: *AlbumsArtistsWrapper*, *TracksWrapper*, *PlaylistsUsersWrapper*, *MusicMediator*, and *MusicWebApiMask*. The inferred schema elements are coloured so that they reflect the data source they originate from in Figure 8.6.

The following tables represent the tableaus of the inferred schema from the *AlbumsArtistsWrapper*:

Tableau id	AlbumsArtistsData.main.albums	
Attribute name	**Data type**	**Is identity**
AlbumId	LONGINT	T
Title	STRING	F
ArtistId	LONGINT	F
Update set 0	AlbumId, Title, ArtistId	

Tableau id	AlbumsArtistsData.main.artists	
Attribute name	**Data type**	**Is identity**
ArtistId	LONGINT	T
Name	STRING	F
Update set 0	ArtistId, Name	

The following tables represent the tableaus of the inferred schema from the *TracksWrapper*:

Tableau id	TracksData.main.genres	
Attribute name	**Data type**	**Is identity**
GenreId	LONGINT	T
Name	STRING	F
Update set 0	GenreId, Name	

Tableau id	TracksData.main.media_types	
Attribute name	**Data type**	**Is identity**
MediaTypeId	LONGINT	T
Name	STRING	F
Update set 0	MediaTypeId, Name	

Tableau id	TracksData.main.tracks	
Attribute name	**Data type**	**Is identity**
TrackId	LONGINT	T
Name	STRING	F
AlbumId	LONGINT	F
MediaTypeId	LONGINT	F
GenreId	LONGINT	F
Composer	STRING	F
Milliseconds	LONGINT	F
Bytes	LONGINT	F
UnitPrice	DECIMAL	F
Update set 0	TrackId, Name, AlbumId, MediaTypeId, GenreId, Composer, Milliseconds, Bytes, UnitPrice	

The following tables represent the tableaus of the inferred schema from the *PlaylistsUsersWrapper*:

Tableau id	PlaylistsUsersData.main.playlists	
Attribute name	**Data type**	**Is identity**
PlaylistId	LONGINT	T
Name	STRING	F
CreatorId	LONGINT	F
Update set 0	PlaylistId, Name, CreatorId	

Tableau id	PlaylistsUsersData.main.playlist_track	
Attribute name	**Data type**	**Is identity**
PlaylistId	LONGINT	T
TrackId	LONGINT	T
CreatorId	LONGINT	F
Update set 0	PlaylistId, TrackId	

Tableau id	PlaylistsUsersData.main.customers	
Attribute name	**Data type**	**Is identity**
CustomerId	LONGINT	T
FirstName	STRING	F
LastName	STRING	F
Company	STRING	F
Address	STRING	F
City	STRING	F
State	STRING	F
Country	STRING	F
PostalCode	STRING	F
Phone	STRING	F
Fax	STRING	F
Email	STRING	F
SupportRepId	LONGINT	F
Update set 0	CustomerId, FirstName, LastName, Company, Address, City, State, Country, PostalCode, Phone, Fax, Email, SupportRepId	

The *MusicMediator* is used to mediate the aforementioned wrappers' schemas. The following is the mediation script used in the *MusicMediator* to mediate those schemas:

```
SETTING
  PROPAGATE UPDATE SETS
  PROPAGATE ATTRIBUTE DESCRIPTIONS
DATASOURCEMusicData VERSION "1.0" #Mediated data about music#
  WITHSCHEMAMain #Default schema#
    WITHTABLEAUAlbums #Data about albums#
      WITHATTRIBUTES
        AlbumId,
        AlbumTitle,
        ArtistName
      BEING
        SELECT
          AlbumsArtistsData.main.albums.AlbumId,
          AlbumsArtistsData.main.albums.Title,
          AlbumsArtistsData.main.artists.Name
        FROMAlbumsArtistsData.main.albums
        JOINAlbumsArtistsData.main.artists
          ONAlbumsArtistsData.main.albums.ArtistId ==
  ↪ AlbumsArtistsData.main.artists.ArtistId
    WITHTABLEAUTracks #Data about tracks and albums#
      WITHATTRIBUTES
        TrackId,
        TrackName,
        GenreName,
        MediaType,
        AlbumTitle,
        DurationMs,
        Composer
      BEING
        SELECT
          TracksData.main.tracks.TrackId,
          TracksData.main.tracks.Name,
          TracksData.main.genres.Name,
          TracksData.main.media_types.Name,
          AlbumsArtistsData.main.albums.Title,
          TracksData.main.tracks.Milliseconds,
          TracksData.main.tracks.Composer
        FROMTracksData.main.tracks
        JOINTracksData.main.genres
          ONTracksData.main.tracks.GenreId == TracksData.main.genres
  ↪ .GenreId
        JOINTracksData.main.media_types
```

```
41          ONTracksData.main.tracks.MediaTypeId == TracksData.main.
    ↪ media_types.MediaTypeId
42        JOINAlbumsArtistsData.main.albums
43          ONTracksData.main.tracks.AlbumId == AlbumsArtistsData.main
    ↪ .albums.AlbumId
44      WITHTABLEAUUsers #Data about users/customers#
45        WITHATTRIBUTES
46          UserId,
47          UserEmail,
48          UserFirstName,
49          UserLastName,
50          UserCountry
51        BEING
52          SELECT
53            PlaylistsUsersData.main.customers.CustomerId,
54            PlaylistsUsersData.main.customers.Email,
55            PlaylistsUsersData.main.customers.FirstName,
56            PlaylistsUsersData.main.customers.LastName,
57            PlaylistsUsersData.main.customers.Country
58          FROMPlaylistsUsersData.main.customers
59      WITHTABLEAUPlaylists #Data about playlists#
60        WITHATTRIBUTES
61          PlaylistId,
62          PlaylistName,
63          CreatorEmail
64        BEING
65          SELECT
66            PlaylistsUsersData.main.playlists.PlaylistId,
67            PlaylistsUsersData.main.playlists.Name,
68            PlaylistsUsersData.main.customers.Email
69          FROMPlaylistsUsersData.main.playlists
70          JOINPlaylistsUsersData.main.customers
71            ONPlaylistsUsersData.main.playlists.CreatorId ==
    ↪ PlaylistsUsersData.main.customers.CustomerId
72      WITHTABLEAUPlaylistTracks #Tracks in playlists#
73        WITHATTRIBUTES
74          TrackId,
75          TrackName,
76          TrackGenre,
77          PlaylistId,
78          PlaylistName
79        BEING
80          SELECT
81            PlaylistsUsersData.main.playlist_track.TrackId,
82            TracksData.main.tracks.Name,
```

```
83          TracksData.main.genres.Name,
84          PlaylistsUsersData.main.playlist_track.PlaylistId,
85          PlaylistsUsersData.main.playlists.Name
86        FROMPlaylistsUsersData.main.playlist_track
87        JOINTracksData.main.tracks
88          ONPlaylistsUsersData.main.playlist_track.TrackId ==
    ↪ TracksData.main.tracks.TrackId
89        JOINPlaylistsUsersData.main.playlists
90          ONPlaylistsUsersData.main.playlist_track.PlaylistId ==
    ↪ PlaylistsUsersData.main.playlists.PlaylistId
91        JOINTracksData.main.genres
92          ONTracksData.main.tracks.GenreId == TracksData.main.genres
    ↪ .GenreId
```

The application of the mediation script in the *MusicMediator* resulted in a schema described by the following tables (the colours marking the attributes' and update sets' origins are preserved):

Tableau id	MusicData.Main.Albums	
Attribute name	**Data type**	**Is identity**
AlbumId	LONGINT	T
AlbumTitle	STRING	F
ArtistName	STRING	F
Update set 0	AlbumId, AlbumTitle	
Update set 1	ArtistName	

Tableau id	MusicData.Main.Tracks	
Attribute name	**Data type**	**Is identity**
TrackId	LONGINT	T
TrackName	STRING	F
GenreName	STRING	F
MediaType	STRING	F
AlbumTitle	STRING	F
DurationMs	LONGINT	F
Composer	STRING	F
Update set 0	TrackId, TrackName, Composer, DurationMs	
Update set 1	GenreName	
Update set 2	MediaType	
Update set 3	AlbumTitle	

Tableau id	MusicData.Main.Users	
Attribute name	**Data type**	**Is identity**
UserId	LONGINT	T
UserEmail	STRING	F
UserFirstName	STRING	F
UserLastName	STRING	F
UserCountry	STRING	F
Update set 0	UserId, UserFirstName, UserLast-Name, UserCountry, UserEmail	

Tableau id	MusicData.Main.Playlists	
Attribute name	**Data type**	**Is identity**
PlaylistId	LONGINT	T
PlaylistName	STRING	F
CreatorEmail	STRING	F
Update set 0	PlaylistId, PlaylistName	
Update set 1	CreatorEmail	

Tableau id	MusicData.Main.PlaylistTracks	
Attribute name	**Data type**	**Is identity**
TrackId	LONGINT	T
TrackName	STRING	F
TrackGenre	STRING	F
PlaylistId	LONGINT	T
PlaylistName	STRING	F
Update set 0	PlaylistId, TrackId	
Update set 1	TrackName	
Update set 2	PlaylistName	
Update set 3	TrackGenre	

The schema loaded into the *MusicWebApiMask* component is the same as in the *MusicMediator*, but the mask provides a representation of the schema in the form of a Web REST API. The following tables contain the description of the generated API routes in the mask:

Tableau id	MusicData.Main.Albums
Route prefix	/Main

Route	**Method**
/Albums/{id}	GET
/Albums?<query_string>	GET
/Albums/UpdateSet_0?<query_string>	PUT
/Albums/UpdateSet_1?<query_string>	PUT

Tableau id	MusicData.Main.Playlists
Route prefix	/Main

Route	**Method**
/Playlists/{id}	GET
/Playlists?<query_string>	GET
/Playlists/UpdateSet_0?<query_string>	PUT
/Playlists/UpdateSet_1?<query_string>	PUT

Tableau id	MusicData.Main.PlaylistTracks
Route prefix	/Main

Route	**Method**
/PlaylistTracks/{id}	GET
/PlaylistTracks?<query_string>	GET
/PlaylistTracks/UpdateSet_0?<query_string>	PUT
/PlaylistTracks/UpdateSet_1?<query_string>	PUT
/PlaylistTracks/UpdateSet_2?<query_string>	PUT
/PlaylistTracks/UpdateSet_3?<query_string>	PUT

Tableau id	MusicData.Main.Tracks
Route prefix	/Main
Route	**Method**
/Tracks/{id}	GET
/Tracks?<query_string>	GET
/Tracks/UpdateSet_0?<query_string>	PUT
/Tracks/UpdateSet_1?<query_string>	PUT
/Tracks/UpdateSet_2?<query_string>	PUT
/Tracks/UpdateSet_3?<query_string>	PUT

Tableau id	MusicData.Main.Users
Route prefix	/Main
Route	**Method**
/Users/{id}	GET
/Users?<query_string>	GET
/Users	POST
/Users/{id}	DELETE
/Users/UpdateSet_0?<query_string>	PUT

10.5.2 Query translation and execution activities

The activity diagram in Figure 10.1 shows the sequence of activities undertaken when a GET request for the /Main/PlaylistTracks resource is initiated by an end-user on the MusicWebApiMask component. Upon receiving the request, the mask component translates the request to a system format query presented in Listing 10.5. The query is then sent as part of a QUERY_REQ message to the component from which the mask loaded its current schema - the *MusicMediator*.

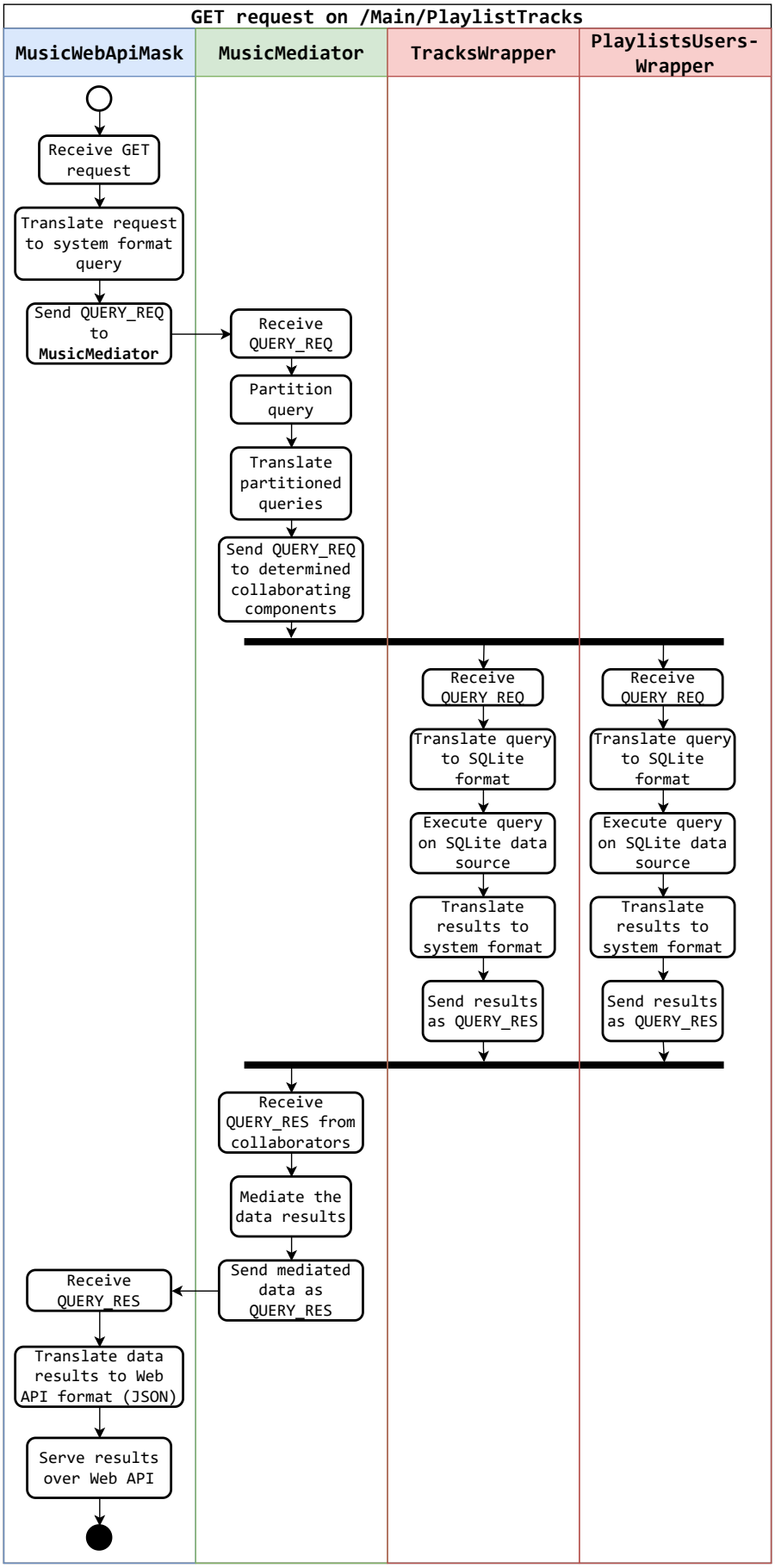


Figure 10.1: Activities undertaken by the Janus system components on sending a GET method request to `/Main/PlaylistTracks` of the *MusicWebApiMask*

The *MusicMediator* receives the message and begins the process of running the enclosed query. The query is first partitioned for sending to individual mediated schemas from remote components. The queries are translated to match the schema element names of the mediated schemas (Listings 10.6 and 10.7). The specifications of operations required to mediate the results of these queries are also created. The `QueryMediation` class is used for these specifications, and its code is available in the `Janus.Mediation` project. The translated partitioned queries are sent in parallel as `QUERY_REQ` messages to the two wrappers with data sources that are determined to contain the required data - *PlaylistsUsersWrapper* and *TracksWrapper*. The *AlbumsArtistsWrapper* is not used for this query, since it doesn't contain any required data.

```
SELECT*
FROMMusicData.Main.PlaylistTracks;
```

Listing 10.5: GET method request for `/Main/PlaylistsTracks` translated into a system format query

```
SELECT
  PlaylistsUsersData.main.playlist_track.TrackId,
  PlaylistsUsersData.main.playlist_track.PlaylistId,
  PlaylistsUsersData.main.playlists.Name
FROMPlaylistsUsersData.main.playlist_track
JOINPlaylistsUsersData.main.playlists
  ONPlaylistsUsersData.main.playlist_track.PlaylistId ==
    ↪ PlaylistsUsersData.main.playlists.PlaylistId;
```

Listing 10.6: Partitioned query as sent to *PlaylistsUsersWrapper*

```
SELECT
  TracksData.main.tracks.TrackId,
  TracksData.main.tracks.Name,
  TracksData.main.genres.GenreId,
  TracksData.main.genres.Name
FROMTracksData.main.tracks
JOINTracksData.main.genres
  ONTracksData.main.tracks.GenreId == TracksData.main.genres.GenreId
    ↪ ;
```

Listing 10.7: Partitioned query as sent to *TracksWrapper*

The wrapper components receive their `QUERY_REQ` messages and translate the enclosed queries into SQLite queries (Listing 10.8 and 10.9). The translated queries are executed and their results are translated to the system format (as `TabularData`). The results are returned to the *MusicMediator* by `QUERY_RES` messages.

```
SELECT
  playlist_track.TrackID,
```

```
3   playlist_track.PlaylistId,
4   playlists.Name
5 FROMplaylist_track
6 JOINplaylists
7   ONplaylist_track.PlaylistId == playlists.PlaylistId;
```

Listing 10.8: SQLite query executed by *PlaylistsUsersWrapper*

```
1 SELECT
2   tracks.TrackId,
3   tracks.Name,
4   genres.GenreId,
5   genres.Name
6 FROMtracks
7 JOINgenres
8   ONtracks.GenreId == genres.GenreId
```

Listing 10.9: SQLite query executed by *TracksWrapper*

The `QUERY_RES` messages containing the expected `TabularData` instances of the data results are awaited and received in the *MusicMediator*. The `TabularData` instances are mediated according to the associated mediation specification. The mediated results now consist of a singular `TabularData` object. A `QUERY_RES` message is returned to the `MusicWebApiMask`. The mask uses the `TabularDataDtoLens` to translate the data into DTOs specified by classes. The DTO class instances are turned into JSON by the ASP.NET framework. The results in the JSON format are served to the end-user who initiated the original request. The end-user might not even realise that the request used a data source integration system or that it caused a propagation of queries to multiple underlying systems; their only concern was sending a valid request to the Web API and awaiting the result in the JSON format.

10.5.3 Data translation

The example of data translation can be presented from the query activities provided in Section 10.5.2. That section dealt with the activities taken by making an HTTP GET request for the `/Main/PlaylistTracks` resource on the `MusicWebApiMask`. To observe a concrete case of data translation, this section covers the acquisition and propagation of the result data.

SQLite queries shown in Listings 10.8 and 10.9, are executed over their respective databases. The results already translated into `TabularData` are exemplified in Tables 10.9 and 10.10.

Table 10.9: Data sample of the tabular data result acquired by the `PlaylistsUsersWrapper`

PlaylistsUsersData.main.playlist_track.TrackId (LONGINT)	**PlaylistsUsersData.main.playlist_track.PlaylistId (LONGINT)**	**PlaylistsUsersData.main.playlists.Name (STRING)**
3402	1	Music
3389	1	Music
...	...	...
1063	5	90's Music
1064	5	90's Music
...	...	...
3426	15	Classical 101 - The Basics
3427	15	Classical 101 - The Basics
3367	16	Grunge
52	16	Grunge
...	...	...

Table 10.10: Data sample of the tabular data result acquired by the `TracksWrapper`

TracksData.main.tracks.TrackId (LONGINT)	**TracksData.main.tracks.Name (STRING)**	**TracksData.main.genres.GenreId (LONGINT)**	**TracksData.main.genres.Name (STRING)**
1	For Those About To Rock (We Salute You)	1	Rock
2	Fast As a Shark	1	Rock
...	...	...	...
77	Enter Sandman	3	Metal
78	Master Of Puppets	3	Metal
...	...	...	...
99	Your Time Has Come	4	Alternative & Punk
100	Out Of Exile	4	Alternative & Punk
...	...	...	...

The original mediated query referenced the mediated `MusicData.Main.PlaylistTracks` tableau (Listing 10.5). The required joins for mediating the data are determined according to the specifications given in the mediation script. This can be concretely found in lines 72-92 of Listing 10.5.1. The `JOIN` clause concerning the joining of the two data results from different wrappers can be found in lines 87-88 of the mediation script; the results must be joined according to the equality of their respective `TrackId` attributes. The joining of the data produces a `TabularData` exemplified in Table 10.11.

Table 10.11: Data sample of the tabular data results acquired by the `MusicMediator`

MusicData. Main. PlaylistTracks. TrackId (LONGINT)	**MusicData. Main. PlaylistTracks. TrackName (STRING)**	**MusicData. Main. PlaylistTracks. TrackGenre (STRING)**	**MusicData. Main. PlaylistTracks. PlaylistId (LONGINT)**	**MusicData. Main. PlaylistTracks. PlaylistName (STRING)**
3402	Band Members Discuss Tracks from "Revelations"	Alternative	1	Music
3389	Revelations	Alternative	1	Music
...	...	...	...	...
1136	Boulevard Of Broken Dreams	Alternative & Punk	1	Music
1137	Are We The Waiting	Alternative & Punk	1	Music
...	...	...	...	...
1837	Seek & Destroy	Metal	17	Heavy Metal Classic
1854	Master Of Puppets	Metal	17	Heavy Metal Classic
...	...	...	...	...

The mediated query was created on a request received from the `MusicWebApiMask` component. The mask's system query was equivalent to that of the mediated query, but it originated from an HTTP GET request on the Web API. An appropriate response is required, and it is expected that it comes in the form of an HTTP successful response containing a JSON object with the data results. To achieve this, the `TabularData` is first translated into a DTO of the expected result. The DTO class type was created at runtime, at the start of the Web API instance in the `MusicWebApiMask`. The DTO class is automatically named as `Main_PlaylistTracks_Get`.

The class is detailed in Listing 10.10 as a C# class and as Swagger schema in Listing.

```
publicclassMain_PlaylistTracks_Get
{
    publiclongTrackId { get; set; }
    publicstringTrackName { get; set; }
    publicstringTrackGenre { get; set; }
    publiclongPlaylistId { get; set; }
    publicstringPlaylistName { get; set; }
}
```

Listing 10.10: The `Main_PlaylistTracks_Get` class generated at runtime

```
{
    TrackId      integer($int64)
    TrackName    string
                 nullable: true
    TrackGenre   string
                 nullable: true
    PlaylistId   integer($int64)
    PlaylistName string
                  nullable: true
}
```

Listing 10.11: The `Main_PlaylistTracks_Get` class represented in the Swagger schema

The `TabularDataDtoLens` is then tasked with translating the `TabularData` instance into an enumerable of the `Main_PlaylistTracks_Get` class. As shown in Chapter 7, this is done by utilizing reflection. The created enumerable is then served through the middleware of the ASP.NET Web API application (mask Web API instance) where it is transformed into a JSON string.

```
[
  {
    "TrackId": 3402,
    "TrackName": "Band Members Discuss Tracks from \"Revelations\"",
    "TrackGenre": "Alternative",
    "PlaylistId": 1,
    "PlaylistName": "Music"
  },
  {
    "TrackId": 3389,
    "TrackName": "Revelations",
    "TrackGenre": "Alternative",
    "PlaylistId": 1,
    "PlaylistName": "Music"
  },
```

```
  ...
  {
    "TrackId": 1136,
    "TrackName": "Boulevard Of Broken Dreams",
    "TrackGenre": "Alternative & Punk",
    "PlaylistId": 8,
    "PlaylistName": "Music"
  },
  {
    "TrackId": 1137,
    "TrackName": "Are We The Waiting",
    "TrackGenre": "Alternative & Punk",
    "PlaylistId": 8,
    "PlaylistName": "Music"
  },
  ...
    {
    "TrackId": 1837,
    "TrackName": "Seek & Destroy",
    "TrackGenre": "Metal",
    "PlaylistId": 17,
    "PlaylistName": "Heavy Metal Classic"
  },
  {
    "TrackId": 1854,
    "TrackName": "Master Of Puppets",
    "TrackGenre": "Metal",
    "PlaylistId": 17,
    "PlaylistName": "Heavy Metal Classic"
  },
  ...
]
```

Listing 10.12: Sample of the result data served as JSON

Bibliography

[1]Don čević, J., Fertalj, K., “Database Integration Systems”, in 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO), Sep. 2020, str. 1617–1622, iSSN: 2623-8764.

[2]Sheth, A., Larson, J., “Federated Database-Systems for Managing Distributed, Heterogeneous, and Autonomous Databases”, Computing Surveys, Vol. 22, No. 3, Sep. 1990, str. 183–236, wOS:A1990GL46000002.

[3]Wiederhold, G., “Mediators in the architecture of future information systems”, Computer, Vol. 25, No. 3, Mar. 1992, str. 38–49.

[4]Papakonstantinou, Y., Garcia-Molina, H., Widom, J., “Object exchange across heterogeneous information sources”, in Proceedings of the Eleventh International Conference on Data Engineering, Mar. 1995, str. 251–260.

[5]Roth, M. T., Schwarz, P., “Don’t Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources”, in Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997, str. 10.

[6]Chiang Lee, Chia-Jung Chen, “Query optimization in multidatabase systems considering schema conflicts”, IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 6, Dec. 1997, str. 941–955, dostupno na: http://ieeexplore.ieee.org/document/649318/

[7]Busse, S., Kutsche, R.-D., Leser, U., Weber, H., “Federated Information Systems: Concepts, Terminology and Architectures”, Forschungsberichte Des Fachbereichs Informatik, 1999, str. 41.

[8]Roth, M. T., Arya, M., Haas, L., Carey, M., Cody, W., Fagin, R., Schwarz, P., Thomas, J., Wimmers, E., “The Garlic project”, in Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD ’96. Montreal, Quebec, Canada: ACM Press, 1996, str. 557, dostupno na: http://portal.acm.org/citation.cfm?doid=233269.280363

[9]Chawathe, S. S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J., "The TSIMMIS Project: Integration of Heterogeneous Information Sources", undefined, 1994, dostupno na: /paper/The-TSIMMIS-Project%3A-Integration-of-Heterogeneous-Chawathe-Garcia-Molina/14348170a14b4e2edca01521184cb2cd60b83200

[10]Leavitt, N., "Will NoSQL Databases Live Up to Their Promise?", Computer, Vol. 43, No. 2, Feb. 2010, str. 12–14, dostupno na: http://ieeexplore.ieee.org/document/5410700/

[11]Dixon, J., "Pentaho, Hadoop, and Data Lakes", dostupno na: https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/ Oct. 2010.

[12]Bogatu, A., Fernandes, A. A. A., Paton, N. W., Konstantinou, N., "Dataset Discovery in Data Lakes", in 2020 IEEE 36th International Conference on Data Engineering (ICDE), Apr. 2020, str. 709–720, iSSN: 2375-026X.

[13]Golshan, B., Halevy, A., Mihaila, G., Tan, W.-C., "Data Integration: After the Teenage Years", in Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, ser. PODS '17. New York, NY, USA: Association for Computing Machinery, May 2017, str. 101–106, dostupno na: https://doi.org/10.1145/3034786.3056124

[14]Don čević, J., Fertalj, K., Brčić, M., Krajna, A., "Mask–Mediator–Wrapper: A Revised Mediator–Wrapper Architecture for Heterogeneous Data Source Integration", Applied Sciences, Vol. 13, No. 4, Jan. 2023, str. 2471, number: 4 Publisher: Multidisciplinary Digital Publishing Institute, dostupno na: https://www.mdpi.com/2076-3417/13/4/2471

[15]Richards, M., Ford, N., Fundamentals of Software Architecture: An Engineering Approach, 1st ed. Sebastopol, CA: O'Reilly Media, Mar. 2020.

[16]Ford, N., Richards, M., Sadalage, P., Dehghani, Z., Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures, 1st ed. Sebastopol, CA: O'Reilly Media, Nov. 2021.

[17]Ford, N., Parsons, R., Kua, P., Building Evolutionary Architectures: Support Constant Change, 1st ed. Beijing: O'Reilly Media, Nov. 2017.

[18]Meyer, B., "The grand challenge of trusted components", in 25th International Conference on Software Engineering, 2003. Proceedings., May 2003, str. 660–667, iSSN: 0270-5257.

[19]Özsu, M. T., Valduriez, P., Principles of distributed database systems, 3rd ed. New York: Springer Science+Business Media, 2011, oCLC: ocn706920112.

[20]Dehghani, Z., Data Mesh: Delivering Data-Driven Value at Scale, 1st ed. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly Media, Apr. 2022.

[21]Mens, T., Eden, A. H., "On the Evolution Complexity of Design Patterns", Electronic Notes in Theoretical Computer Science, Vol. 127, No. 3, Apr. 2005, str. 147–163, dostupno na: https://www.sciencedirect.com/science/article/pii/S1571066105001465

[22]Eden, A., Mens, T., "Measuring software flexibility", IEE Proceedings - Software, Vol. 153, No. 3, 2006, str. 113, dostupno na: https://digital-library.theiet.org/content/journals/10.1049/ip-sen_20050045

[23]Voigtländer, J., "Bidirectionalization for free! (Pearl)", in POPL '09, 2009.

[24]Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M., "Bidirectionalization transformation based on automatic derivation of view complement functions", ACM SIGPLAN Notices, Vol. 42, No. 9, Oct. 2007, str. 47–58, dostupno na: https://doi.org/10.1145/1291220.1291162

[25]Hofmann, M., Pierce, B., Wagner, D., "Symmetric lenses", ACM SIGPLAN Notices, Vol. 46, No. 1, 2011, str. 371–384, dostupno na: https://doi.org/10.1145/1925844.1926428

[26]Miltner, A., Fisher, K., Pierce, B. C., Walker, D., Zdancewic, S., "Synthesizing bijective lenses", Proceedings of the ACM on Programming Languages, Vol. 2, No. POPL, 2017, str. 1:1–1:30, dostupno na: https://dl.acm.org/doi/10.1145/3158089

[27]Doncevic, J., "Janus", dostupno na: https://github.com/JurajDoncevic/Janus Original-date: 2022-03-14T13:56:20Z. Feb. 2023.

[28]Don čević, J., Fertalj, K., Brčić, M., Kovač, M., "Mask-Mediator-Wrapper architecture as a Data Mesh driver", dostupno na: http://arxiv.org/abs/2209.04661 ArXiv:2209.04661 [cs]. Sep. 2022.

[29]Kimball, R., Caserta, J., The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data, 1st ed. Indianapolis, IN: Wiley, Oct. 2004.

[30]Zhang, Y., Zhang, Y., Wang, S., Lu, J., "Fusion OLAP: Fusing the Pros of MOLAP and ROLAP Together for In-memory OLAP (Extended Abstract)", in 2019 IEEE 35th International Conference on Data Engineering (ICDE), Apr. 2019, str. 2125–2126, iSSN: 2375-026X.

[31]Forresi, C., Gallinucci, E., Golfarelli, M., Hamadou, H. B., "A dataspace-based framework for OLAP analyses in a high-variety multistore", The VLDB Journal, Vol. 30, No. 6, Nov. 2021, str. 1017–1040, dostupno na: https://link.springer.com/10.1007/s00778-021-00682-5

[32]"Apache Linkis | Apache Linkis", dostupno na: https://linkis.apache.org/

[33]Cappuzzo, R., Papotti, P., Thirumuruganathan, S., "Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks", SIGMOD Conference, 2020.

[34]da Trindade, J. M. F., Karanasos, K., Curino, C., Madden, S., Shun, J., "Kaskade: Graph Views for Efficient Graph Analytics", in 2020 IEEE 36th International Conference on Data Engineering (ICDE), Apr. 2020, str. 193–204, iSSN: 2375-026X.

[35]Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., Vaisman, A., "A model and query language for temporal graph databases", The VLDB Journal, Vol. 30, No. 5, Sep. 2021, str. 825–858, dostupno na: https://link.springer.com/10.1007/s00778-021-00675-4

[36]Chatziantoniou, D., Kantere, V., "DataMingler: A Novel Approach to Data Virtualization", in Proceedings of the 2021 International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, Jun. 2021, str. 2681–2685, dostupno na: https://doi.org/10.1145/3448016.3452752

[37]Magdy, A., Abdelhafeez, L., Kang, Y., Ong, E., Mokbel, M. F., "Microblogs data management: a survey", The VLDB Journal — The International Journal on Very Large Data Bases, Vol. 29, No. 1, Jan. 2020, str. 177–216, dostupno na: https://doi.org/10.1007/s00778-019-00569-6

[38]Arenas, M., Gottlob, G., Pieris, A., "Expressive Languages for Querying the Semantic Web", ACM Transactions on Database Systems, Vol. 43, No. 3, Nov. 2018, str. 13:1–13:45, dostupno na: https://doi.org/10.1145/3238304

[39]Krommyda, M., Kantere, V., "Visualization Systems for Linked Datasets", in 2020 IEEE 36th International Conference on Data Engineering (ICDE), Apr. 2020, str. 1790–1793, iSSN: 2375-026X.

[40]Zhou, J., Xu, M., Shraer, A., Namasivayam, B., Miller, A., Tschannen, E., Atherton, S., Beamon, A. J., Sears, R., Leach, J., Rosenthal, D., Dong, X., Wilson, W., Collins, B., Scherer, D., Grieser, A., Liu, Y., Moore, A., Muppana, B., Su, X., Yadav, V., "FoundationDB: A Distributed Unbundled Transactional Key Value Store", in Proceedings of the 2021 International Conference on Management of Data. New York,

NY, USA: Association for Computing Machinery, Jun. 2021, str. 2653–2666, dostupno na: https://doi.org/10.1145/3448016.3457559

[41]Zimányi, E., Sakr, M., Lesuisse, A., "MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS", ACM Transactions on Database Systems, Vol. 45, No. 4, Dec. 2020, str. 19:1–19:42, dostupno na: https://doi.org/10.1145/3406534

[42]Seidemann, M., Glombiewski, N., Körber, M., Seeger, B., "ChronicleDB: A High-Performance Event Store", ACM Transactions on Database Systems, Vol. 44, No. 4, Oct. 2019, str. 13:1–13:45, dostupno na: https://doi.org/10.1145/3342357

[43]Zhao, X., Jiang, S., Wu, X., "WipDB: A Write-in-place Key-value Store that Mimics Bucket Sort", in 2021 IEEE 37th International Conference on Data Engineering (ICDE), Apr. 2021, str. 1404–1415, iSSN: 2375-026X.

[44]Liang, J., Chai, Y., "CruiseDB: An LSM-Tree Key-Value Store with Both Better Tail Throughput and Tail Latency", in 2021 IEEE 37th International Conference on Data Engineering (ICDE), Apr. 2021, str. 1032–1043, iSSN: 2375-026X.

[45]Yourdon, E., Constantine, L. L., Structured design: fundamentals of a discipline of computer program and systems design. Englewood Cliffs, N.J: Prentice Hall, 1979.

[46]Martin, R., Clean Architecture: A Craftsman's Guide to Software Structure and Design, 1st ed. London, England: Pearson, Sep. 2017.

[47]Martin, R., Agile Software Development, Principles, Patterns, and Practices, 1st ed. Upper Saddle River, N.J: Pearson, Oct. 2002.

[48]Page-Jones, M., What every programmer should know about object-oriented design. New York, N.Y: Dorset House Pub, 1995.

[49]Evans, E., Domain-Driven Design: Tackling Complexity in the Heart of Software, 1st ed. Boston: Addison-Wesley Professional, Aug. 2003.

[50]"ISO 25010", dostupno na: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[51]"Data created worldwide 2005-2025 | Statistic", dostupno na: https://www.statista.com/statistics/871513/worldwide-data-created/

[52]Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", Vol. 13, No. 6, 1970.

[53]Pang, Z., Lu, Q., Chen, S., Wang, R., Xu, Y., Wu, J., "ArkDB: A Key-Value Engine for Scalable Cloud Storage Services", in Proceedings of the 2021 International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, Jun. 2021, str. 2570–2583, dostupno na: https://doi.org/10.1145/3448016.3457553

[54]Dehghani, Z., "How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh", dostupno na: https://martinfowler.com/articles/data-monolith-to-mesh.html May 2019.

[55]Gonzalez-Perez, C., Henderson-Sellers, B., Metamodelling for Software Engineering, 1st ed. Chichester, UK ; Hoboken, NJ: Wiley, Oct. 2008.

[56]Henderson-Sellers, B., On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages. Springer, Jun. 2012.

[57]Breitbart, Y., Garcia-Molina, H., Silberschatz, A., "Overview of multidatabase transaction management", 1992, str. 59.

[58]Vathy-Fogarassy, A., Hugyák, T., "Uniform data access platform for SQL and NoSQL database systems", Information Systems, Vol. 69, Sep. 2017, str. 93–105, dostupno na: https://linkinghub.elsevier.com/retrieve/pii/S0306437916303398

[59]Garcia-Molina, H., Ullman, J., Widom, J., Database Systems: The Complete Book, 2nd ed. Upper Saddle River, N.J: Pearson, Jun. 2008.

[60]Jurczyk, P., Xiong, L., Goryczka, S., "DObjects+: Enabling Privacy-Preserving Data Federation Services", in 2012 IEEE 28th International Conference on Data Engineering, Apr. 2012, str. 1325–1328, iSSN: 2375-026X.

[61]Moura, S. L. d., Coutinho, F., Siqueira, S. W. M., Melo, R. N., Nunes, S. V., "Integrating repositories of learning objects using Web-services to implement mediators and wrappers", in International Conference on Next Generation Web Services Practices (NWeSP'05), Aug. 2005, str. 6 pp.–.

[62]Hongzhi Wang, Jianzhong Li, Zhenying He, "An effective wrapper architecture to heterogeneous data source", in 17th International Conference on Advanced Information Networking and Applications, 2003. AINA 2003., Mar. 2003, str. 565–568.

[63]Chang, Y., Chang, C., Cheng, H., "Applying ontology to geographical scientific data extraction", in 2011 IEEE International Conference on Systems, Man, and Cybernetics, Oct. 2011, str. 3397–3402, iSSN: 1062-922X.

[64]Shao, Y., Di, L., Kang, L., Bai, Y., “An integrated framework for geospatial data discovering and standardized processing”, in 2013 Second International Conference on Agro-Geoinformatics (Agro-Geoinformatics), Aug. 2013, str. 334–337.

[65]Garg, B., Kaur, K., “Integration of heterogeneous databases”, in 2015 International Conference on Advances in Computer Engineering and Applications, Mar. 2015, str. 1033–1038.

[66]Schmatz, K., Berwind, K., Engel, F., Hemmje, M. L., “An Interface to Heterogeneous Data Sources Based on the Mediator/Wrapper Architecture in the Hadoop Ecosystem”, in 2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), Dec. 2018, str. 1838–1845.

[67]Sethi, R., Traverso, M., Sundstrom, D., Phillips, D., Xie, W., Sun, Y., Yegitbasi, N., Jin, H., Hwang, E., Shingte, N., Berner, C., “Presto: SQL on Everything”, in 2019 IEEE 35th International Conference on Data Engineering (ICDE), Apr. 2019, str. 1802–1813, iSSN: 2375-026X.

[68]Dehghani, Z., “Data Mesh Principles and Logical Architecture”, dostupno na: https://martinfowler.com/articles/data-mesh-principles.html Dec. 2020.

[69]Conway, M. E., “HOW DO COMMITTEES INVENT?”, Datamation magazine, F. D. Thompson Publications, Inc., 1968, str. 4.

[70]Cockburn, A., “Hexagonal architecture”, dostupno na: https://alistair.cockburn.us/hexagonal-architecture/ 2005.

[71]“IEEE Standard Glossary of Software Engineering Terminology”, IEEE Std 610.12-1990, Dec. 1990, str. 1–84, conference Name: IEEE Std 610.12-1990.

[72]Pierce, B. C., Basic Category Theory for Computer Scientists. Cambridge, Mass: The MIT Press, Aug. 1991.

[73]Milewski, B., Category Theory for Programmers. Milton Keynes: Bartosz Milewski, Jan. 2019.

[74]Abelson, H., Sussman, G. J., Sussman, J., Structure and Interpretation of Computer Programs - 2nd Edition, second edition ed. Cambridge, Mass.: The MIT Press, Sep. 1996.

[75]Sussman, G. J., Steele, G. L., “SCHEME: An Interpreter for Extended Lambda Calculus”, Dec. 1975, accepted: 2004-10-01T20:37:06Z, dostupno na: https://dspace.mit.edu/handle/1721.1/5794

[76]Gamma, E., Helm, R., Johnson, R., Vlissides, J., Booch, G., Design Patterns: Elements of Reusable Object-Oriented Software, 1st ed. Reading, Mass: Addison-Wesley Professional, Oct. 1994.

[77]Buonanno, E., Functional Programming in C#, Second Edition, 2nd ed. Shelter Island, NY: Manning, Feb. 2022.

[78]Barr, M., Wells, C., Category Theory for Computing Science, 1st ed. New York: Prentice Hall, Jul. 1990.

[79]Wlaschin, S., Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#, 1st ed. Raleigh, North Carolina: Pragmatic Bookshelf, Feb. 2018.

[80]Bancilhon, F., Spyratos, N., "Update semantics of relational views", ACM Transactions on Database Systems, Vol. 6, No. 4, Dec. 1981, str. 557–575, dostupno na: https://dl.acm.org/doi/10.1145/319628.319634

[81]Diskin, Z., "Algebraic Models for Bidirectional Model Synchronization", in Model Driven Engineering Languages and Systems, ser. Lecture Notes in Computer Science, Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M., (ur.). Berlin, Heidelberg: Springer, 2008, str. 21–36.

[82]Diskin, Z., "Compositionality of Update Propagation: Laxed PutPut", Proceedings of the Sixth International Workshop on Bidirectional Transformations (Bx 2017), Apr. 2017, str. 16.

[83]Anjorin, A., Diskin, Z., Jouault, F., Ko, H.-S., Leblebici, E., Darmstadt, T., Westfechtel, B., "Benchmarx Reloaded: A Practical Benchmark Framework for Bidirectional Transformations", Proceedings of the Sixth International Workshop on Bidirectional Transformations (Bx 2017), Apr. 2017, str. 16.

[84]Tullsen, M., "ASN.1 Encoding Schemes Done Right Using CMPCT", in Proceedings of the 8th International Workshop on Bidirectional Transformations, ser. CEUR Workshop Proceedings, Cheney, J., Ko, H.-S., (ur.), Vol. 2355. Philadelphia, PA: CEUR, Jun. 2019, str. 1–15, iSSN: 1613-0073, dostupno na: http://ceur-ws.org/Vol-2355/#paper1

[85]Asano, Y., Hidaka, S., Hu, Z., Ishihara, Y., Kato, H., Ko, H.-S., Nakano, K., Onizuka, M., Sasaki, Y., Shimizu, T., Tsushima, K., Yoshikawa, M., "A View-based Programmable Architecture for Controlling and Integrating Decentralized Data", arXiv:1803.06674 [cs], Mar. 2018, arXiv: 1803.06674, dostupno na: http://arxiv.org/abs/1803.06674

[86]Asano, Y., Cao, Y., Hidaka, S., Hu, Z., Ishihara, Y., Kato, H., Nakano, K., Onizuka, M., Sasaki, Y., Shimizu, T., Takeichi, M., Xiao, C., Yoshikawa, M., "Bidirectional Collaborative Frameworks for Decentralized Data Management", in Software Foundations for Data Interoperability, Fletcher, G., Nakano, K., Sasaki, Y., (ur.). Cham: Springer International Publishing, 2022, Vol. 1457, str. 13–51, series Title: Communications in Computer and Information Science, dostupno na: https://link.springer.com/10.1007/978-3-030-93849-9_2

[87]Weidmann, N., Anjorin, A., Fritsche, L., Varró, G., Schürr, A., Leblebici, E., "Incremental Bidirectional Model Transformation with eMoflon::IBeX", in Proceedings of the 8th International Workshop on Bidirectional Transformations, ser. CEUR Workshop Proceedings, Cheney, J., Ko, H.-S., (ur.), Vol. 2355. Philadelphia, PA: CEUR, Jun. 2019, str. 45–55, iSSN: 1613-0073, dostupno na: http://ceur-ws.org/Vol-2355/#paper4

[88]Matsuda, K., Wang, M., "FliPpr: A Prettier Invertible Printing System", in Programming Languages and Systems, ser. Lecture Notes in Computer Science, Felleisen, M., Gardner, P., (ur.). Berlin, Heidelberg: Springer, 2013, str. 101–120.

[89]Foster, N., Matsuda, K., Voigtländer, J., "Three Complementary Approaches to Bidirectional Programming", in Generic and Indexed Programming, Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Gibbons, J., (ur.). Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, Vol. 7470, str. 1–46, series Title: Lecture Notes in Computer Science, dostupno na: http://link.springer.com/10.1007/978-3-642-32202-0_1

[90]Voigtländer, J., Hu, Z., Matsuda, K., Wang, M., "Combining syntactic and semantic bidirectionalization", ACM SIGPLAN Notices, Vol. 45, No. 9, Sep. 2010, str. 181–192, dostupno na: https://dl.acm.org/doi/10.1145/1932681.1863571

[91]Pacheco, H., Hu, Z., Fischer, S., "Monadic combinators for "Putback" style bidirectional programming", in Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, ser. PEPM '14. New York, NY, USA: Association for Computing Machinery, Jan. 2014, str. 39–50, dostupno na: https://doi.org/10.1145/2543728.2543737

[92]Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P., "Reflections on Monadic Lenses", in A List of Successes That Can Change the World, Lindley, S., McBride, C., Trinder, P., Sannella, D., (ur.). Cham: Springer International Publishing,

2016, Vol. 9600, str. 1–31, series Title: Lecture Notes in Computer Science, dostupno na: http://link.springer.com/10.1007/978-3-319-30936-1_1

[93]Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P., "Introduction to Bidirectional Transformations", in Bidirectional Transformations, Gibbons, J., Stevens, P., (ur.). Cham: Springer International Publishing, 2018, Vol. 9715, str. 1–28, series Title: Lecture Notes in Computer Science, dostupno na: http://link.springer.com/10.1007/978-3-319-79108-1_1

[94]Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A., Schmitt, A., "Boomerang: Resourceful Lenses for String Data", Nov. 2007, str. 33.

[95]Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., Schmitt, A., "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem", ACM Transactions on Programming Languages and Systems, Vol. 29, No. 3, May 2007, str. 17, dostupno na: https://dl.acm.org/doi/10.1145/1232420.1232424

[96]Foster, J. N., Pilkiewicz, A., Pierce, B. C., "Quotient lenses", ACM SIGPLAN Notices, Vol. 43, No. 9, Sep. 2008, str. 383–396, dostupno na: https://doi.org/10.1145/1411203.1411257

[97]Wadler, P., "Deforestation: transforming programs to eliminate trees", Theoretical Computer Science, Vol. 73, No. 2, Jun. 1990, str. 231–248, dostupno na: https://www.sciencedirect.com/science/article/pii/030439759090147A

[98]Wadler, P., "Theorems for free!", in Proceedings of the fourth international conference on Functional programming languages and computer architecture, ser. FPCA '89. New York, NY, USA: Association for Computing Machinery, Nov. 1989, str. 347–359, dostupno na: https://doi.org/10.1145/99370.99404

[99]Foster, J. N., Pierce, B. C., "Boomerang Programmer's Manual", dostupno na: https://www.seas.upenn.edu/~harmony/manual.pdf Sep. 2009.

[100]Ko, H.-S., Zan, T., Hu, Z., "BiGUL: a formally verified core language for putback-based bidirectional programming", in Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, ser. PEPM '16. New York, NY, USA: Association for Computing Machinery, Jan. 2016, str. 61–72, dostupno na: https://doi.org/10.1145/2847538.2847544

[101]Matsuda, K., Wang, M., "HOBiT: Programming Lenses Without Using Lens Combinators", in Programming Languages and Systems, Ahmed, A., (ur.). Cham: Springer International Publishing, 2018, Vol. 10801, str. 31–59, series Title:

Lecture Notes in Computer Science, dostupno na: http://link.springer.com/10.1007/978-3-319-89884-1_2

[102]Lutterkort, D., "AUGEAS—a configuration API", Proceedings of Linux Symposium, 2008, str. 47–56.

[103]"Augeas — Main", dostupno na: https://augeas.net/

[104]Foster, J. N., "Bidirectional Programming Languages", Doktorski rad, University of Pennsylvania, 2009.

[105]Barbosa, D. M. J., Cretin, J., Foster, N., Greenberg, M., Pierce, B. C., "Matching Lenses: Alignment and View Update", 2010, str. 12.

[106]Miltner, A., Maina, S., Fisher, K., Pierce, B. C., Walker, D., Zdancewic, S., "Synthesizing symmetric lenses", Proceedings of the ACM on Programming Languages, Vol. 3, No. ICFP, Jul. 2019, str. 1–28, dostupno na: https://dl.acm.org/doi/10.1145/3341699

[107]Anjorin, A., Ko, H.-S., "Towards a visual editor for lens combinators (extended abstract)", in Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming. Nice France: ACM, Apr. 2018, str. 33–35, dostupno na: https://dl.acm.org/doi/10.1145/3191697.3191719

[108]Fischer, S., Hu, Z., Pacheco, H., "The essence of bidirectional programming", Science China Information Sciences, Vol. 58, No. 5, May 2015, str. 1–21, dostupno na: http://link.springer.com/10.1007/s11432-015-5316-8

[109]Pacheco, H., Zan, T., Hu, Z., "BiFluX: A Bidirectional Functional Update Language for XML", in Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming. Canterbury United Kingdom: ACM, Sep. 2014, str. 147–158, dostupno na: https://dl.acm.org/doi/10.1145/2643135.2643141

[110]Zan, T., Liu, L., Ko, H.-S., Hu, Z., "Brul: A Putback-Based Bidirectional Transformation Library for Updatable Views", Proceedings of the Fifth International Workshop on Bidirectional Transformations (Bx 2016), Apr. 2016.

[111]Asano, Y., Herr, D.-F., Ishihara, Y., Kato, H., Nakano, K., Onizuka, M., Sasaki, Y., "Flexible Framework for Data Integration and Update Propagation: System Aspect", in 2019 IEEE International Conference on Big Data and Smart Computing (BigComp), Feb. 2019, str. 1–5, iSSN: 2375-9356.

[112]Asano, Y., Hu, Z., Ishihara, Y., Kato, H., Onizuka, M., Yoshikawa, M., "Controlling and Sharing Distributed Data for Implementing Service Alliance", in 2019 IEEE International Conference on Big Data and Smart Computing (BigComp), Feb. 2019, str. 1–4, iSSN: 2375-9356.

[113]Kawanaka, S., Hosoya, H., "biXid: A Bidirectional Transformation Language for XML", in Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming, Portland, Oregon, USA, Sep. 2006, str. 16.

[114]Miltner, A. F., "Synthesizing Lenses", Doctoral Dissertation, Princeton University, Sep. 2020.

[115]Matsuda, K., Wang, M., "Applicative bidirectional programming: Mixing lenses and semantic bidirectionalization", Journal of Functional Programming, Vol. 28, 2018, str. e15, dostupno na: https://www.cambridge.org/core/product/identifier/S0956796818000096/type/journal_article

[116]Atzeni, P., Bugiotti, F., Rossi, L., "Uniform access to NoSQL systems", Information Systems, Vol. 43, Jul. 2014, str. 117–133, dostupno na: http://www.sciencedirect.com/science/article/pii/S0306437913000719

[117]Li, R., Lu, Z., Xiao, W., Wu, W., "XML-based integration data model and schema mapping in multidatabase systems", Journal of Systems Engineering and Electronics, Vol. 16, No. 2, 2005, str. 437–444.

[118]Kozankiewicz, H., Stencel, K., Subieta, K., "Integration of heterogeneous resources through updatable views", in 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Jun. 2004, str. 309–314, iSSN: 1524-4547.

[119]Lawrence, R., "Integration and Virtualization of Relational SQL and NoSQL Systems Including MySQL and MongoDB", in 2014 International Conference on Computational Science and Computational Intelligence, Vol. 1, Mar. 2014, str. 285–290.

[120]Abuzaid, F., Kraft, P., Suri, S., Gan, E., Xu, E., Shenoy, A., Ananthanarayan, A., Sheu, J., Meijer, E., Wu, X., Naughton, J., Bailis, P., Zaharia, M., "DIFF: a relational interface for large-scale data explanation", The VLDB Journal, Vol. 30, No. 1, Jan. 2021, str. 45–70, dostupno na: https://link.springer.com/10.1007/s00778-020-00633-6

[121]Li, Y., Cao, J., Chen, H., Ge, T., Xu, Z., Peng, Q., "FlashSchema: Achieving High Quality XML Schemas with Powerful Inference Algorithms and Large-scale Schema

Data", in 2020 IEEE 36th International Conference on Data Engineering (ICDE), Apr. 2020, str. 1962–1965, iSSN: 2375-026X.

[122]Lam, H. T., Buesser, B., Min, H., Minh, T. N., Wistuba, M., Khurana, U., Bramble, G., Salonidis, T., Wang, D., Samulowitz, H., "Automated Data Science for Relational Data", in 2021 IEEE 37th International Conference on Data Engineering (ICDE), Apr. 2021, str. 2689–2692, iSSN: 2375-026X.

[123]Gkini, O., Belmpas, T., Koutrika, G., Ioannidis, Y., "An In-Depth Benchmarking of Text-to-SQL Systems", in Proceedings of the 2021 International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, Jun. 2021, str. 632–644, dostupno na: https://doi.org/10.1145/3448016.3452836

[124]Fielding, R. T., "Architectural Styles and the Design of Network-based Software Architectures", Doctrolar dissertation, UNIVERSITY OF CALIFORNIA, IRVINE, University of California, Irvine, 2000.

[125]Benedikt, M., Bourhis, P., Jachiet, L., Tsamoura, E., "Balancing Expressiveness and Inexpressiveness in View Design", ACM Transactions on Database Systems, Vol. 46, No. 4, Nov. 2021, str. 15:1–15:40, dostupno na: https://doi.org/10.1145/3488370

[126]Qin, X., Luo, Y., Tang, N., Li, G., "Making data visualization more efficient and effective: a survey", The VLDB Journal, Vol. 29, No. 1, Jan. 2020, str. 93–117, dostupno na: http://link.springer.com/10.1007/s00778-019-00588-3

[127]Ivanics, P., "An Introduction to Clean Software Architecture", 2017, dostupno na: https://pivanics.users.cs.helsinki.fi/portfolio/docs/publications/Peter_Ivanics-Clean_Software_Architecture.pdf

[128]Young, G., "CQRS Documents by Greg Young", 2010.

[129]Kleppmann, M., Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, 1st ed. Boston: O'Reilly Media, May 2017.

[130]Parr, T., The Definitive ANTLR 4 Reference. Dallas, Texas, Raleigh, North Carolina: The Pragmatic Programmer, 2012.

[131]Atzeni, P., Bugiotti, F., Rossi, L., "Uniform Access to Non-relational Database Systems: The SOS Platform", in Active Flow and Combustion Control 2018, King, R., (ur.). Cham: Springer International Publishing, 2012, Vol. 141, str. 160–174, series Title: Notes on Numerical Fluid Mechanics and Multidisciplinary Design, dostupno na: http://link.springer.com/10.1007/978-3-642-31095-9_11

[132]“SQLite Sample Database And Its Diagram (in PDF format)”, dostupno na: https://www.sqlitetutorial.net/sqlite-sample-database/

Biography

Juraj Dončević was born on the 31st of December 1992 in Bjelovar, Croatia. He enrolled in the *Computing* undergraduate program at the University of Zagreb Faculty of Electrical Engineering and Computing (FER), where he earned his bachelor's degree in 2016. The same year he enrolled in the graduate *Software Engineering* program at FER. Upon completing his master's thesis, titled *Heterogeneous Database System*, he received his master's degree in 2018. He enrolled in postgraduate doctoral studies at FER that same year. In 2018, he also took up the position of teaching assistant at the *Department of Applied Computing* at FER. Since then, he has been involved in the following undergraduate and graduate courses: Introduction to Programming, Algorithms and Data Structures, Development of Software Applications, and Information Systems. His research involves software architecture, software design, data management, functional programming, bidirectionalisation, and category theory.

List of published work

Journal articles

1. Don čević, J.; Fertalj, K.; Brčić, M.; Krajna, A. Mask–Mediator–Wrapper: A Revised Mediator–Wrapper Architecture for Heterogeneous Data Source Integration. Appl. Sci. 2023, 13, 2471. https://doi.org/10.3390/app13042471

Conference papers

1. J. Don čević and K. Fertalj, "Database Integration Systems", 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO), Opatija, Croatia, 2020, pp. 1617-1622, doi: 10.23919/MIPRO48935.2020.9245245.

Životopis

Juraj Dončević rođen je 31. prosinca 1992. u Bjelovaru, Hrvatska. Upisao je preddiplomski studij *Računarstva* na Fakultetu elektrotehnike i računarstva (FER) Sveučilišta u Zagrebu, gdje je 2016. godine stekao zvanje prvostupnika. Iste godine upisao je diplomski studij *Programskog inženjerstva* na FER-u. Nakon obrane diplomskog rada pod nazivom *Heterogeni sustav baza podataka* diplomirao je 2018. Iste godine upisao je poslijediplomski doktorski studij na FER-u. 2018. godine preuzima i mjesto asistenta na *Zavodu za primijenjeno računarstvo* FER-a. Od tada je uključen u sljedeće preddiplomske i diplomske kolegije: Uvod u programiranje, Algoritmi i strukture podataka, Razvoj primijenjene programske potpore i Informacijski sustavi. Njegovo istraživanje uključuje arhitekturu softvera, dizajn softvera, upravljanje podacima, funkcijsko programiranje, bidirekcionalizaciju i teoriju kategorija.