

Undergraduate Topics in Computer Science

Kevin Lano
Sobhan Yassipour Tehrani

Introduction to Software Architecture

Innovative Design using
Clean Architecture
and Model-Driven Engineering



Springer

Undergraduate Topics in Computer Science

Series Editor

Ian Mackie, University of Sussex, Brighton, UK

Advisory Editors

Samson Abramsky , Department of Computer Science, University of Oxford, Oxford, UK

Chris Hankin , Department of Computing, Imperial College London, London, UK

Mike Hinchey , Lero—The Irish Software Research Centre, University of Limerick, Limerick, Ireland

Dexter C. Kozen, Department of Computer Science, Cornell University, Ithaca, NY, USA

Hanne Riis Nielson , Department of Applied Mathematics and Computer Science, Technical University of Denmark, Kongens Lyngby, Denmark

Steven S. Skiena, Department of Computer Science, Stony Brook University, Stony Brook, NY, USA

Iain Stewart , Department of Computer Science, Durham University, Durham, UK

Joseph Migga Kizza, Engineering and Computer Science, University of Tennessee at Chattanooga, Chattanooga, TN, USA

Roy Crole, School of Computing and Mathematics Sciences, University of Leicester, Leicester, UK

'Undergraduate Topics in Computer Science' (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems, many of which include fully worked solutions.

The UTiCS concept relies on high-quality, concise books in softback format, and generally a maximum of 275–300 pages. For undergraduate textbooks that are likely to be longer, more expository, Springer continues to offer the highly regarded Texts in Computer Science series, to which we refer potential authors.

Kevin Lano · Sobhan Yassipour Tehrani

Introduction to Software Architecture

Innovative Design using Clean Architecture
and Model-Driven Engineering

Kevin Lano
Department of Informatics
King's College London
London, UK

Sobhan Yassipour Tehrani
Department of Computer Science
University College London (UCL)
London, UK

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-031-44142-4 ISBN 978-3-031-44143-1 (eBook)
<https://doi.org/10.1007/978-3-031-44143-1>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Preface

Software architecture has become recognised as a key factor in the maintainability and sustainability of software systems – poor quality architectures are likely to doom a system to becoming an expensive drain on business resources, costing more and more money to maintain over time as the system structure deteriorates and hinders evolution. In contrast, the choice of a good quality and appropriate architecture can ensure that a system retains its usefulness for many years and remains open to extension and enhancement.

This book provides a comprehensive introduction to software architecture concepts and techniques, with a focus upon the ‘clean architecture’ principles (Chapter 2), which have been shown to be effective for a wide range of software systems.

We introduce the key concepts of software architectures in Chapter 1, and provide a ‘toolkit’ of 30 architectural styles and patterns which can be used for constructing mobile apps, enterprise and web applications, machine learning systems, and safety-critical systems (Chapters 5 to 11).

We describe how architectural design can be applied within agile or model-driven software development processes (Chapter 3), and consider in detail the specialised issues of architecture for specific kinds of application: mobile apps (Chapter 6), web applications and enterprise systems (Chapters 7 and 8), service-oriented systems (Chapter 9), machine learning systems (Chapter 11) and safety-critical systems (Chapter 10). We also look at architecture in the context of software re-engineering and examine to what extent an improved architecture can be ‘retro-fitted’ into a legacy system (Chapter 12).

This book is based on the module “Software architecture and design” which has been taught at final year undergraduate and masters level at King’s College London from 2010–2023. The material has been class-tested and supporting tools, slides and complete case study examples are provided on the book website¹.

¹ <https://agilemde.co.uk/arch>

The book covers:

- The key concepts of architectural design.
- The concepts of the clean architecture approach.
- Architectural styles and patterns.
- Architectures for particular domains: mobile apps; enterprise systems; service-based systems; machine learning systems; safety-critical systems.
- Software architecture in the context of model-driven development and agile development approaches.
- Software architecture in the context of legacy system renewal.

The core topics of a one-semester undergraduate module in software architecture are covered in Chapters 1 to 9. Chapters 10 to 12 are more advanced topics suitable for a more specialised software architecture module at masters level. The book is also relevant to software practitioners and researchers.

Acknowledgements

Lynn Alwakeel contributed to the definition of the mobile app architectural approach described in Chapter 6. Holistic Risk Solutions and Dr Howard Haughton contributed the financial re-engineering case of Chapter 12. Kunxiang Jin contributed to the model-based testing approach of Chapter 12. Many researchers have contributed to the development of the AgileUML tools used here, in particular Shekoufeh Kolaoudou-Rahimi and Hessa Alfraihi. Over many years, feedback from students on the “Software architecture and design” course has helped to refine and clarify the presentation of the material included here.

London, UK
August 2023

Kevin Lano
Sobhan Yassipour Tehrani

Contents

1	Introduction to Software Architecture Concepts	11
1.1	What is software architecture?	11
1.1.1	Why is software architecture important?	13
1.1.2	The software architect role	13
1.1.3	Development methods and software architecture	14
1.1.4	Representing architectures	14
1.2	Key architectural concepts	15
1.2.1	Components	15
1.2.2	Connectors	17
1.2.3	Configurations	18
1.3	UML component diagram notation	18
1.3.1	Provided and required interfaces	20
1.3.2	Assembly connectors	23
1.4	Properties of components and architectures	24
1.4.1	Reusability	25
1.4.2	Substitutability	26
1.4.3	Sustainability	27
1.4.4	Maintainability	28
1.4.5	Extensibility	29
1.5	Quality of service (QoS) properties	29
1.6	Architectural design approaches	30
2	Introduction to Clean Architecture Concepts	35
2.1	Motivation for the clean architecture	35
2.2	The clean architecture: SOLID principles	37
2.2.1	SRP: Single responsibility principle	37
2.2.2	OCP: Open-closed principle	39
2.2.3	LSP: Liskov substitution principle	39
2.2.4	ISP: Interface segregation principle	42
2.2.5	DIP: Dependency inversion principle	43
2.2.6	Architecture design principles	45

2.3	Component boundaries	46
2.4	Technical debt at the architectural level	46
3	Development Methods	51
3.1	Model-driven engineering	51
3.2	Agile development and architectural design	52
3.2.1	Architectural refactoring	54
3.3	Derivation of architectures from requirements models	56
3.3.1	Three-tier architectures	56
3.3.2	Requirements models	57
3.3.3	Derivation of architectures from requirements	60
3.4	Analysis of clean architecture principles at the specification stage	63
3.5	Generation of architectures from specification models	68
4	Compound Components and Complex Connectors	73
4.1	Composite components	73
4.2	Advanced connectors	74
5	Architectural Styles	81
5.1	Adapter architectural style	81
5.2	Pipes and filters architectural style	82
5.2.1	Pipes and filters example: Order processing system . . .	84
5.3	Two-tiered (Client/Server) architectural style	85
5.4	N-tiered architectural style	85
5.4.1	Example N-tier architectures: enterprise and mobile systems	87
5.4.2	N-tiered style: advantages and disadvantages	87
5.5	Layered architectural style	88
5.6	Blackboard architectural style	89
5.6.1	Blackboard example: IDE	91
5.6.2	Blackboard example: Holiday reservation system	91
5.6.3	Blackboard example: Realtime cloud datastore	93
5.6.4	Advantages and disadvantages of the blackboard style	93
5.7	Model-View-Controller (MVC) architectural style	94
5.7.1	Model-View-Controller advantages and disadvantages	96
5.8	Heterogeneous styles	96
6	Mobile Application Architectures	99
6.1	Mobile applications	99
6.1.1	Similarities between mobile applications and enterprise systems (EIS)	100
6.1.2	Distinctive aspects of mobile applications	100
6.1.3	Mobile app example	102
6.2	Mobile architecture patterns	105
6.2.1	Model-View-Controller and alternative versions	106

6.2.2	Value Object/Data Transfer Object pattern	112
6.2.3	Model Facade/Session Facade pattern	113
6.2.4	Service Activator pattern	114
6.2.5	Data Access Object (DAO) pattern	114
6.3	The Android platform and applications	117
6.4	The iOS platform and applications	125
6.4.1	UIKit	125
6.4.2	SwiftUI	127
6.5	Comparison of iOS versus Android	130
6.6	Mobile app development approaches	131
6.7	Mobile backend as a service (MBaaS)	132
7	Enterprise Information Systems and Application Servers ..	137
7.1	Enterprise systems	137
7.1.1	Application servers	137
7.2	.NET enterprise services	139
7.3	Transactions service	140
7.4	Object pooling service	143
7.5	Just-in-time activation service	145
7.6	Loosely-coupled events service	146
7.7	Queued components service	147
7.8	COM+ services as basic enterprise styles	148
8	Web Application and Enterprise System Architectures ...	151
8.1	Enterprise system architectures	151
8.2	Java Enterprise Edition	152
8.2.1	Java EE Components: Servlets	153
8.2.2	Java EE Components: Java Server Pages (JSPs)	153
8.2.3	Java EE Components: Enterprise JavaBeans (EJBs) ..	155
8.3	Web application architectures	155
8.3.1	Client tier	158
8.3.2	Presentation tier	160
8.3.3	Business tier	161
8.3.4	Web application frameworks	162
8.4	View Helper style	163
8.5	Front Controller style	164
8.6	Intercepting Filter style	166
8.7	Composite View style	166
8.8	Session Facade style	168
8.9	Service Locator style	169
8.10	Extract-transform-load (ETL) style	169
8.11	Using multiple styles	170

9	Service-oriented Architectures	177
9.1	The cloud and cloud services	177
9.1.1	Infrastructure-as-a-Service (IaaS)	178
9.1.2	Platform-as-a-Service (PaaS)	178
9.1.3	Software-as-a-Service (SaaS)	178
9.1.4	Virtualisation servers	179
9.2	Service-oriented architecture (SOA)	179
9.3	Web services	180
9.3.1	SOAP web services	182
9.3.2	REST and RESTful web services	182
9.3.3	Web service patterns	185
9.4	Microservices	187
10	Safety-critical and Embedded Systems Architectures	191
10.1	Safety-critical systems	191
10.1.1	Technological risk	191
10.1.2	Fault tolerance versus fault avoidance	193
10.2	Fault tolerance	194
10.2.1	Static redundancy: Triple Modular Redundancy architecture	195
10.2.2	Dynamic redundancy: Standby spare architecture	197
10.2.3	Software fault tolerance	197
10.3	Safe states and safety shutdowns	198
10.4	Design approaches for SCS components	200
10.4.1	Programming languages for SCS	200
10.5	Embedded system architectures	202
10.6	Reactive control systems	203
10.7	Examples of SCS architectures	204
10.8	Software safety standards	206
11	Architectural Design for Machine Learning Systems	211
11.1	Machine learning (ML)	211
11.1.1	Machine learning techniques	211
11.1.2	ML system construction and operation	216
11.2	Machine learning example: requirements classification	218
11.3	Machine learning systems architectural issues	220
11.3.1	Language-specific issues: Python	221
11.4	Trustworthiness of ML systems	222
12	Software Architectures and Re-engineering	225
12.1	Legacy software systems	225
12.2	Challenges of legacy code abstraction	227
12.2.1	BASIC and variants	227
12.2.2	COBOL	228
12.3	Program abstraction using ANTLR and <i>CSTL</i>	229

12.3.1	VB6/VBA	231
12.3.2	Cobol85	232
12.4	Quality analysis and restructuring	235
12.4.1	Quality analysis	235
12.4.2	Restructuring	236
12.4.3	Rearchitecting	237
12.5	Code synthesis and test case generation	240
12.6	Re-engineering examples	241
12.6.1	VB6/VBA re-engineering	241
12.6.2	Cobol85 re-engineering	244
References		255
A User guide for architecture diagram tool		259
B User guide for AgileUML		261
Index		263



Chapter 1

Introduction to Software Architecture Concepts

Abstract This chapter covers:

- The definition of software architecture and its key concepts.
- Description of the UML component notation for architectures.
- Definition of key properties of components and architectures, such as reusability, sustainability, substitutability and maintainability.
- Quality of service properties and their influence on architectural choices.

1.1 What is software architecture?

The term *software architecture* refers to a global and high-level view of a software system, which describes aspects such as:

- The software components and component dependencies and interconnections within the system.
- Decisions about which technologies and platforms will be used.
- The distribution structure, for systems deployed on multiple computing nodes.
- Links and interfaces to any external systems.

In a software architecture model, the system structure is considered as a first class artefact, and details of processing and data are not shown, instead the focus is on the separate parts of the system and how they are connected and communicate. A software architecture can be sketched before any code is written.

The term *software design* can refer more broadly to the construction and organisation (design) of individual components, modules, classes and even operations and code segments. *Architectural design* refers to the design of architecture-level structures of the software system.

An example software architecture, of a simple audio player mobile app, is shown in Figure 1.1.

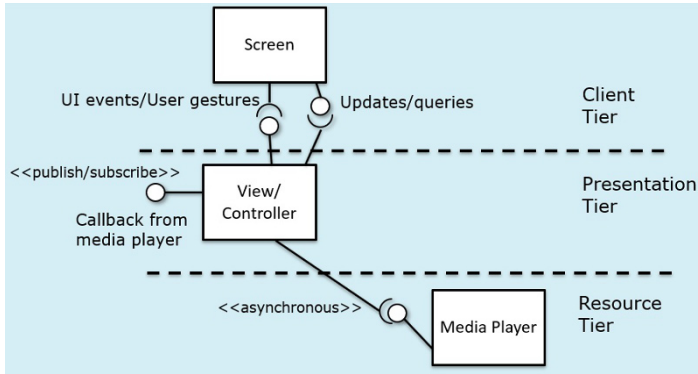


Fig. 1.1 Architecture example

The idea of software architecture became part of the mainstream of software engineering from the mid 90's onwards, with the landmark book by Shaw and Garlan [83] establishing the core definition of the term:

Software architecture involves the description of

- Elements from which systems are built.
- Interactions among those elements.
- Patterns that guide their composition.
- Constraints on these patterns.

Subsequently, research and practice in software architecture extended the concept to:

1. Definition of languages (architecture description languages, ADLs) for specifying architectures.
2. Establishing catalogues of architectural styles or patterns.
3. Analysis and evaluation of architectures.
4. Principles and processes for constructing architectures, i.e., for architectural design (Section 1.6).

In this book we will focus primarily on the use of architectural styles, and on the *clean architecture* approach for architectural design [63]. The clean architecture approach considers that:

“The goal of software architecture is to minimize the human resources required to build and maintain the required system.”

We will also aim for this goal, and extend it to consider also the minimization of the environmental impact of building, maintaining and operating the required system.

1.1.1 Why is software architecture important?

Architectural designs can help to plan, manage and document a software development, to:

- Aid project management.
- Organise top-down development.
- Assign different component developments to different teams.
- Centralise coordination of how the system components should fit together.
- Document decisions about frameworks, platforms, structure, etc.
- Document how non-functional properties of systems are achieved, such as performance, reliability, security, etc.
- Document and modularise the system to aid in future maintenance and evolution.

Many software developments fail because architectural aspects are ignored due to time pressures to deliver a working system quickly. Thus developers build systems with arbitrary ad-hoc structures which are adequate to get the system working, but become increasingly difficult to maintain and change. Costs build up and cause serious impacts on the business operating the system [63].

1.1.2 The software architect role

Therefore software architects can be considered some of the most important people in software projects, as their work will have significant impact on the long-term project success. The software architect role in a project should be explicitly defined, whether a traditional plan-based development approach or an agile development approach (such as the Scrum method [80]) is used. Requirements management is closely linked to architecture management, and the software architect typically has the responsibility to decide how a new requirement should be addressed within an existing system architecture, or if the architecture needs to be significantly changed in order that the new requirement can be handled. They will be involved with the system development at potentially all development stages from requirements analysis to detailed design. In a staged development this means they participate in the requirements analysis and design phases, and in an agile/iterative development they would be involved in the project initiation phase, and in the requirements and design activities within each iteration.

The software architect builds and maintains architectural models, e.g., as UML component diagrams (Section 1.3). They need to be able to explain these models to the development team(s), and work with teams to ensure that a consistent architectural vision is used across a system. Team members should be able to read and understand the models, but do not write/modify

the models. Feedback from the developers to the architect may also result in architectural change. Because of the major impact of significant architectural changes, these need to be managed carefully, with the involvement of relevant stakeholders, although the software architect takes the leading role in the technical aspects of selecting one approach versus another (e.g., using an MVC architecture instead of a Blackboard architecture, as described in Chapter 5).

Generally, there is no magic recipe or formula for producing ‘good’ architectures, and a software architect needs to gain experience of working with different architectures and to develop an understanding of how they apply in practice. We can express some architectural principles and approaches which should be followed and which have had a good track record of success over many different systems and over many years, of which the *clean architecture* principles (Chapter 2), and the concepts of *architectural patterns and styles* (Chapters 5 to 11), evolved by numerous practitioners, are particularly important.

1.1.3 Development methods and software architecture

Apart from principles and techniques, the development approach adopted for a particular system can have a substantial impact on the architectural practices employed: *model-driven engineering (MDE)* has a natural synergy with architectural design, enabling (in principle) the enforcement of architectural choices through the automated construction of software conforming to the choices. In particular it can assist in ensuring the clean architecture principles without some of the costs involved in manual software construction.

On the other hand, *Agile methods* have sometimes been viewed as antagonistic to architectural design, as these methods emphasise the production of executable code, and typically try to minimise the production of documentation. They tend to focus on functional requirements, whilst non-functional requirements, such as efficiency and responsiveness, are particularly significant for architectural choices. Chapter 3 considers these issues in detail.

1.1.4 Representing architectures

Architectures should be documented in diagrams (informal or formal) and possibly in specialised textual notations. This helps for:

- Clarifying ideas and decisions.
- Analysing properties of the architectures.

- Sharing architecture information among members of a development team, or between different teams.
- Stakeholder communication and understanding.

Documentation of architectures can use an architecture description language (ADL). In this book we will use UML component diagram notation, but there are other ADLs. At the most basic level a sketch using boxes and arrows can be used, but such sketches ignore important detail about the component connections and boundaries in particular.

1.2 Key architectural concepts

Software architectures consist of three main parts:

- *Components* – representing computational/data storage nodes.
- *Connectors* – for information interchange between components.
- *Configurations* – which instantiate components and connectors in particular arrangements.

1.2.1 Components

Components are not just random assemblies of code/data, but should be cohesive encapsulated units of a system, which usually play a clearly defined role in the system, e.g., as:

- Clients or servers.
- Databases.
- Filters.
- Layers.

In terms of a programming language implementation, components are usually represented as packages or classes. We will use the term ‘module’ interchangeably with ‘component’.

Components may be simple or *composite* – composite components are essentially sub-systems with their own internal structure, built out of smaller components (Chapter 4). Components *encapsulate functionality* – component *interfaces* expose what clients need to know to use it, but otherwise the internal details are either completely or partially hidden (*black-box* versus *grey-box* components).

Definitions of the component concept include:

“A software package that can be independently replaced. It both provides and requires services based on specified interfaces.” [18]

“... a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.” [88]

We will largely follow these definitions, but distinguish between:

- **Abstract Component:**

- Conceptual representation of a component.
- Shown in ADLs.
- Could potentially be implemented as a concrete component in different languages/platforms.

- **Concrete Component:**

- An actual component file in a specific implementation format (e.g., C# code).
- Ready to be deployed.
- May be specific to particular platform/software infrastructure (e.g., .NET).

Associated with a component are its *provided* and *required* interfaces, which specify, respectively:

- The services it provides to its clients.
- The services it requires from other components (its suppliers).

There can be several (different) required interfaces and several different provided interfaces. A component may provide the same interface that it requires – for example if it is acting as a delegate, data transmitter or filter between other components. We will use the UML component notation for provided and required interfaces: a ball symbol linked to the component for a provided interface, and a cup/socket symbol linked to the component, for a required interface.

An example component could be a hotel reservation component (Figure 1.2). It is used as a supplier through a provided interface *IHotelRes* and is itself a client using two other components via interfaces *IBilling* and *ILoyaltyProgram*.

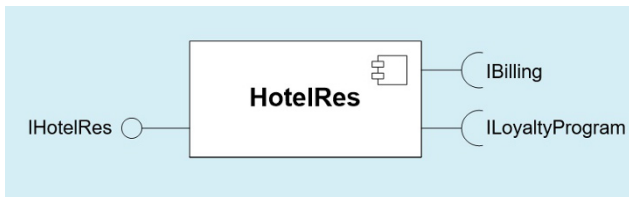


Fig. 1.2 Hotel reservation component

This is an abstract component because there is no commitment to a particular implementation platform.

1.2.2 Connectors

Connectors model the dependencies and communication routes between components, the communication protocols that govern their communications, and any restrictions imposed by an architectural style, for example that the connection should be read-only: i.e., that the client can only query the server but not update its data.

Types of component interactions include:

- Simple (synchronous and local) procedure calls – this is the default type of connection.
- Asynchronous procedure calls (Chapter 4).
- Publish-subscribe interaction (Chapter 4).
- Remote procedure calls.
- Asynchronous event multicast (Chapter 7).
- Realtime data streams.

In Figure 1.2 the *HotelRes* component should check credit card validity and take a booking charge. This is a synchronous communication (and transactional) via the *IBilling* interface. The *HotelRes* component also notifies a *LoyaltyProgram* component via the *ILoyaltyProgram* interface if a regular customer books – this can be a loosely-coupled publish/subscribe communication, because synchronisation of the two components is not necessary in this case.

The simplest form of connector is an *assembly* connector which links one client component to one supplier via their respective required and provided interfaces (Figure 1.3).

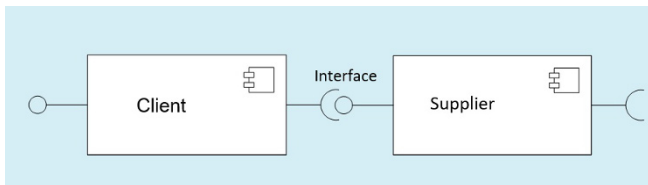


Fig. 1.3 One client using one supplier

Assembly connectors can also link multiple clients to one server (Figures 1.4 and 1.5).

Fig. 1.4 Multiple clients using one server (1)

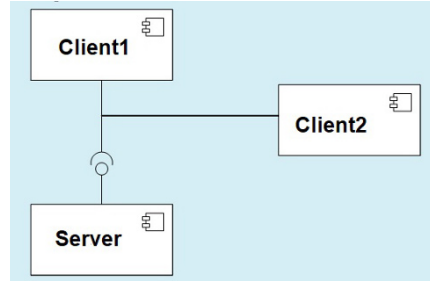
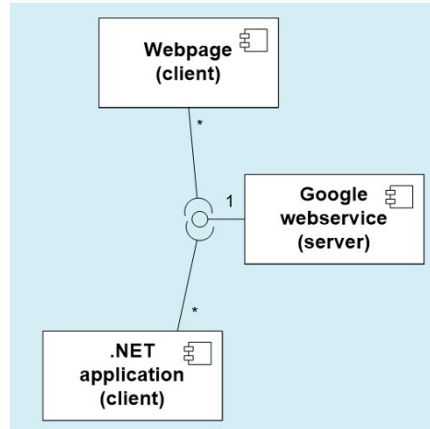


Fig. 1.5 Multiple clients using one server (2)



1.2.3 Configurations

Configurations are a connected graph of components and connectors that describe the structure of an architecture. Various properties can be specified in terms of configurations – for example, it is considered desirable that the graph has no cycles (i.e., that a component cannot be indirectly a client of itself via other components).

Various architectural styles have characteristic configurations, for example the ‘Pipes and filters’ and ‘Intercepting filter’ styles are characterised by linear unidirectional chains of components of the same kind.

1.3 UML component diagram notation

The Unified Modelling Language (UML) [27] was established in the 1990’s as an international standard for software modelling. UML has many different diagram notations, of which class diagrams, use case diagrams and sequence diagrams are among the most-often used. We will use simple examples of

these notations in this book, and use UML *component diagrams* to document architectures. These are related to class diagrams, but show components as boxes with a ‘package’ icon in the top right corner (Figure 1.6). An alternative is to write the stereotype `<<component>>` in the box (Figure 1.7). Instead of writing the list of functions/operations provided by the component inside its box, one or more *provided* interfaces of the component are defined, represented by the ball symbol, connected to the component by a line. The provided interfaces are usually named, and can be defined as UML classes with the `<<interface>>` stereotype, corresponding to the usual object-oriented idea of an interface: a class with operation declarations but no implementations. A novel aspect of component diagrams is that they also make explicit the *required* interfaces of a component: those external services which it needs in order to operate. One or more required interfaces of a component can be defined, represented by the cup symbol, connected to the component by a line. The required interfaces are usually named, and can be defined as UML classes with the `<<interface>>` stereotype.

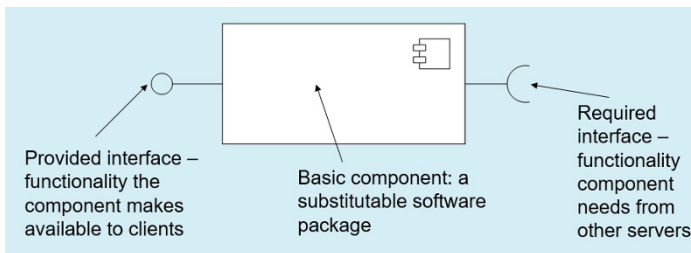


Fig. 1.6 UML component diagram notation

For this book we will use the component diagram editor of the AgileUML tool¹ to draw component diagrams. This tool links together interface specifications drawn in the class diagram editor (top pane of Figure 1.7) with the names of these interfaces in the component diagram (bottom pane of Figure 1.7).

We will mainly use Java to illustrate architectural concepts at the code level, however component diagrams can be used to define architectures for systems written in other object-oriented and non-OO languages, such as C#, Swift, C and Ada. Table 1.1 shows the relation between component diagram concepts and programming language constructs for different languages.

Some languages (such as Python) omit an explicit interface construct, which means that the software architecture cannot be expressed explicitly in the code, only by means of comments and external documentation. Most object-oriented languages do have some form of interface construct, which enables the *provided* interfaces of a component to be expressed as the imple-

¹ github.com/eclipse/agileuml

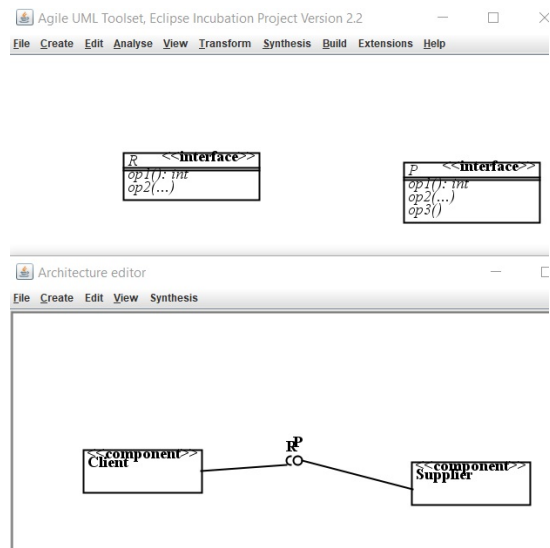


Fig. 1.7 Architecture example in architecture editor

Table 1.1 Architectural elements in different programming languages

Language	Components	Interfaces	Connectors
Ada	Packages	Package specifications	with, use
C	Code (.c) files	Header (.h) files	#include
C++	Class implementations	Class declarations	#include
C#	Classes; Namespaces	Interfaces	using
Go	Files; Packages	interface type	import
Java	Classes; Packages	Interfaces	import
Python	Classes; Files	—	import
Swift	Classes; Files	Protocols	import

mented interfaces of its main (entry point) class. However, there is usually no clear expression of the *required* interfaces of a component. Ada, C and C++ separate module specifications from module bodies/implementations, whilst Java, C#, Go, Python and Swift do not. The Go language is unusual in that there is no explicit *implements* relation between a class and an interface.

1.3.1 Provided and required interfaces

An interface in UML is essentially the same concept as in Java: A list of methods with input/output types. Classes in both Java and UML can implement one or more interfaces using the “implements” relation. If an interface *I* is implemented by class *C*, this means that instances of *C* will have im-

plementations of all methods listed in *I*. E.g., a Java class *C* implementing the *Cloneable* interface means that objects *obj* of *C* can be cloned by calling *obj.clone()*, but the interface does not specify how this happens.

Interfaces separate implementation from the declaration of functionality. This allows implementations to be changed without breaking the system. The provided interfaces provide an abstracted representation of the capabilities of the component. Clients only know the information published in the interface, and cannot assume any particular implementation. This means that replacing one component or implementation by another should not affect the clients if the same interfaces are satisfied. For example, a data persistence facility for a banking system could have published operations:

- *createAccount(accountId : String, name : String, customerId : String)*
- *getBalance(accountId : String) : int*
- *addBalance(accountId : String, amount : int)*
- *deleteAccount(accountId : String)*

These are examples of *create-read-update-delete* (CRUD) operations. The operations would be used by the business tier of the system². Any data persistence facility that supports these operations should be acceptable to the business tier.

A component interface consists of:

- A signature: a list of (public) functions with input and output types.
- A semantics: Some documentation describing what these functions are meant to do and how they should be used.
- The semantics can be given in textual comments, or in the formal Object Constraint Language (OCL) language of UML [71].

One component may have several provided interfaces, and one interface may have several implementations.

Interfaces are represented in two ways in UML models:

1. Interface names next to ball and cup/socket symbols in component diagrams.
2. As interface specifications, drawn as rectangles like UML classes, but with the `<<interface>>` keyword at the top. These contain the interface name and a list of operation signatures provided by the interface.

An interface specification needs to be defined (in text form, or in a class diagram) if the interface name is used in a component diagram.

The UML class diagram syntax for an operation signature in a class or interface is:

visibility operation-name(parameter-list) : return-type

For example:

² We use the term ‘tier’ synonymously with ‘layer’.

```
+ checkAvailability(date1 : Date,
                  date2 : Date, roomt : RoomType) : boolean
```

The visibility is one of: public (+), protected (#), or private (-). Visibility is always public for operations in interfaces, so can be omitted in this case. The return type can be 'void' to indicate that the operation does not return a value.

We will usually write interface specifications in textual UML notation. For example, a simple interface for manipulating sequences of strings could be:

```
interface StringSequence
{ operation addElement(s : String) : void;
  operation removeFirst(s : String) : void;
  operation removeAll(s : String) : void;
  operation hasElement(s : String) : boolean;
  operation indexOf(s : String) : int;
}
```

This uses the same conventions for operation declarations as the class diagram, but the keyword **operation** is added before the declaration.

In Java, the provided interfaces of a class are explicit: these are the interfaces listed after the *implements* keyword. But required interfaces are implicit, they are defined by whatever supplier operations are used by the class (and these can be buried deep within the code).

An example UML component diagram of a complete system is shown in Figure 1.8. This is a holiday reservation system with main component *HolidayReservationSession*. The component uses three different supplier components for hotel reservation (room booking), car reservation (car hire), and air reservation (flight booking). There are therefore three required interfaces of *HolidayReservationSession*, one for each required component.

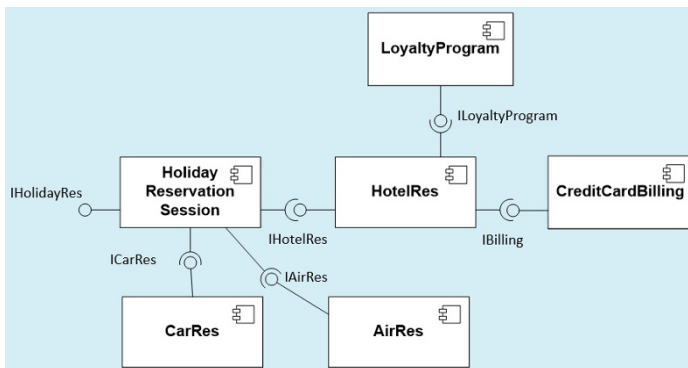


Fig. 1.8 Architecture example: Holiday reservation system

We could implement *HolidayReservationSession* as a Java class of the form:

```
public class HolidayReservationSession
    implements IHolidayRes {
    IHotelRes myHotelRes; /* Instance of required interface */
    ICarRes myCarRes;
    IAirRes myAirRes;

    public HolidayReservationSession(IHotelRes aHotelRes,
                                     ICarRes aCarRes,
                                     IAirRes aAirRes) {

        this.myHotelRes = aHotelRes;
        this.myCarRes = aCarRes;
        this.myAirRes = aAirRes;
    } /* Instantiate suppliers */

    ...
}
```

In this class, the provided interface *IHolidayRes* of *HolidayReservationSession* is listed as an implemented interface, whilst the required interfaces are used as the declared types of supplier objects. The constructor of the class also takes parameters of these required interface types, in order to instantiate the supplier objects. This style of coding helps to make explicit what interfaces the component requires. Notice that no concrete supplier classes are named, only the interfaces which they are expected to implement.

1.3.2 Assembly connectors

UML component diagram notation provides two kinds of simple connectors: Delegation connectors (considered in Chapter 4), and assembly connectors. Assembly connectors connect two components, one provides services required by the other. The connector defines communication between the components, by default this is synchronous and local. The providing component is termed a *server/supplier* to the *requiring/client* component. The requiring component calls methods of the supplier's provided interface.

Assembly connectors are depicted by placing the provided (ball) interface adjacent to the required (socket/cup) interface.

Figure 1.9 shows an extended version of the holiday reservation system with assembly connectors between each of the components.

The required and provided interfaces on an assembly connection may be the same, in which case we only need to write the interface name once. However, in general they could be different, for example *P* and *R* in Figure 1.7.

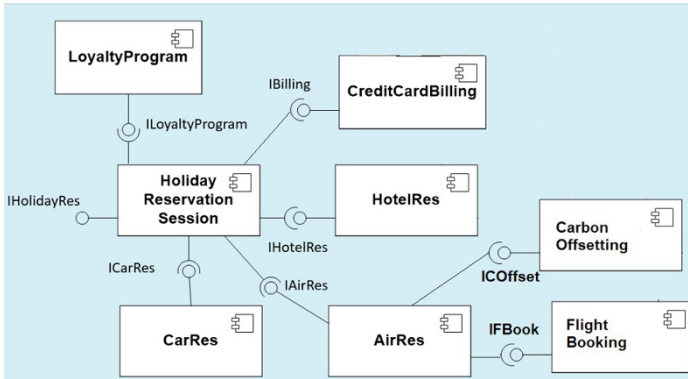


Fig. 1.9 Extended holiday reservation system

In this situation, P must provide all the operations (with the same operation names and signatures) that R does, in other words P is effectively an extension or subtype of R .

1.4 Properties of components and architectures

Components are software modules which should be:

- Key units of composition.
- Cohesive, encapsulating a coherent set of related functionalities.
- Reusable.
- Substitutable.
- Independently deployable.
- Composable into large systems.

In turn, the assemblies of components into architectures should satisfy quality properties such as maintainability and extensibility.

Cohesion of a component means that its functions and data are semantically related (they relate to a common purpose), instead of being arbitrary collections of elements.

Encapsulation means that the details of how components are implemented are hidden from clients – clients do not need to look at the component code to use the component, they only need its provided interface specifications.

Reusability for a component means that the component can be used in different systems without change.

For example, *HotelRes* is a modular and cohesive component – all of its functionalities are concerned with finding hotel rooms and booking stays in a hotel. It would have operations such as *checkAvailability*, *makeReservation*,

changeReservation and *cancelReservation*. It could be used either as a stand-alone component (Figure 1.2), or as part of a larger holiday booking system (Figure 1.8).

Client programmers do not need (and usually cannot see) the detailed code of the component, instead they should be able to use it with no problems via its provided interface.

1.4.1 Reusability

Reuse is an important property for components because by reusing an existing component it should be possible to:

- Reduce development time and cost.
- Improve reliability and quality.

Reuse has been a long-standing goal within software engineering, and there have been some success stories, beginning with libraries of mathematical functions such as *sin*, *cos*, *log*, etc (with common names and functionality in C, C++, Java and other languages), and now extending to complete components via open-source projects such as github and QuantLib.

There remain several problems and obstacles to reusability, such as the following technical difficulties:

- Component functionality may not be reusable because components encapsulate business logic specific to an organisation, or rely upon organisation-specific supplier components.
- The component granularity is too coarse or too fine.
- Components do not provide exactly what is needed, so that wrappers/adapters need to be used, complicating the reuse process.

Problems with reusability also include economic and organisational obstacles:

- Designing for reuse requires a higher up-front investment – and the benefits may not be clear.
- Reuse requires a long-term vision and buy-in from the management.
- The benefits of reuse must outweigh the risks/costs.
- “Not invented here” syndrome – developer mistrust of components of uncertain provenance, that use different coding practices and standards.

A successful example of reuse is the QuantLib (<https://www.quantlib.org/>) large-scale open source library for quantitative finance, which has been developed over 20 years. The library has been widely used in academia and industry.

However there have been many failures, for example (at the time of writing) no recognised benchmark solution has been established for using SQLite from Swift, despite many candidate solutions being proposed³.

Ideally, components should be defined to support reuse in a desirable range of contexts and systems. This means they should be flexible enough to support interconnections/adaptions that might occur in the future, without impacting on their key functionality.

At a larger scale, it is sometimes possible to reuse a core set of components as part of a *software product line* – a family of products all based on a common set of functionalities and business entities [34]. The common code base is parameterised to enable a wide range of variations on the common functionalities to be provided. Effectively the reusable parts become a custom software platform for a dedicated range of applications. This form of reuse embodies substantial specific knowledge, and hence tends to be used within a single company or for highly-specialised application areas. QuantLib can be regarded as an example of this kind of reuse, for the domain of quantitative finance applications.

1.4.2 Substitutability

Components are substitutable units within their containing system, in principle they can be replaced at design or run time by any component that offers (at least) equivalent functionality, based on compatibility of its interfaces.

We can replace a component with an alternative or updated version, without breaking the system in which the component is used, if the following restrictions are satisfied.

Component *B* can be substituted for component *A* if:

1. *A*'s provided interfaces are preserved by *B* (*B* has the same or more provided interfaces than *A*).
2. The signature and expected behaviour of all operations of all provided interfaces of *A* are preserved by *B*.
3. *B*'s required interfaces are the same as or fewer than *A*'s – *B* cannot require more functionalities than *A* did, because the existing system context (of *A*) may not provide these.

As an example of substitutability, the original version of a hotel reservation component (Figure 1.10) can be replaced by an alternative version which requires less than the original (Figure 1.11). It would also be valid to replace *HotelRes* by a version with an additional provided interface (Figure 1.12).

Substitutability is a transitive relation: if *B* is substitutable for *A*, and *C* is substitutable for *B*, then *C* is substitutable for *A*. It is analogous to the subclass relation between classes: if *B* is substitutable for *A* then it is

³ stackoverflow.com/questions/24102775/accessing-an-sqlite-database-in-swift

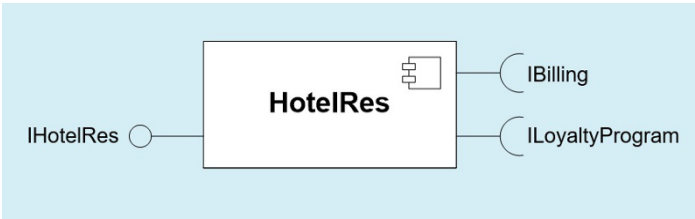


Fig. 1.10 Original component: Hotel reservation component



Fig. 1.11 Alternative hotel reservation component



Fig. 1.12 Hotel reservation with extra provided interface

effectively a specialisation of *A* (it can be used in all systems where *A* could be used).

1.4.3 Sustainability

An architecture is *sustainable* if (i) it does not lead to excessive resource usage by operation of the software system; (ii) the system can be created and maintained without the use of excessive resources.

Sustainable operation requires consideration of:

- The choice of components used in the system, so that they have low resource usage.
- Localisation of connections where possible, minimisation of non-local connections, and efficient use of non-local connections and remote data re-

sources. For example, external data which is highly stable could be cached locally to avoid unnecessary network accesses.

- Reduction in the number of external components/libraries, to remove unnecessary supplier components/imports, and hence to reduce the size of deployed executables.

Sustainable creation and maintenance applies especially to machine learning (ML) systems, where very high computational resources may be required to train or retrain ML models or to perform inference from models (Chapter 11). In general, reuse of existing components or development artefacts for a new system can help to reduce the development effort and resources needed for this system. Model-driven engineering (Chapter 3) can contribute to sustainable software development in this respect, by facilitating reuse of software specifications and designs. MDE can also reduce development effort via increased automation of development processes.

We can also refer to *business sustainability* – the degree to which a system can be supported by a business in the long term. This concerns the long-term availability of staff with the knowledge needed to operate and maintain the system. If a business depends on an external vendor for system support, then it needs to consider the long-term existence of that vendor and their commitment to the particular system.

1.4.4 Maintainability

Software maintenance involves:

- Identification and correction of errors (bug-fixing).
- Preventative maintenance – refactoring or restructuring of code to improve its quality and reduce costs for future changes.
- Evolution and enhancement – such as replacing one data storage technology with another, or adding a new service.

Software maintenance costs and expended resources can become much higher than the original software development costs/resources, for long-lived software, thus any improvements to the software architecture which could reduce these costs are of high significance.

A well-designed architecture can reduce maintenance effort by minimising the impact of changes to one component upon other components.

If a component C is changed in such a way that it still satisfies all its existing provided interfaces $IC_1, IC_2, \dots$, then no client of C should be affected by the changes within C . In addition, if the changes do not lead to new required functionalities of C , then its relations with supplier components can also remain unchanged, and the suppliers are not affected by internal changes to C .

Replacing a supplier component by another (e.g., replacing a local relational database by a remote cloud datastore) should not impact on clients if the replacement satisfies the same required interfaces that the original version satisfied.

1.4.5 Extensibility

A system is extensible if it can be enhanced with further functionalities and capabilities without excessive effort. Individual component extension means that additional provided interfaces are added, or existing provided interfaces extended with new operations. In turn, this can lead to new required interfaces/operations of the component, and hence potentially the addition of new supplier components.

Apart from these extensions, the component should otherwise preserve its existing behaviour, so that clients of the previous version do not need to be aware that the enhancement has taken place – ie., there is *backwards compatibility*.

1.5 Quality of service (QoS) properties

The properties of extensibility, sustainability, etc, given above are examples of *Quality-of-service (QoS)* properties. QoS properties are non-functional requirements on how a system provides its services, such as the system availability, responsiveness, fault-tolerance, efficiency, performance, scalability, security, transactionality, trustworthiness, etc. These properties often have a decisive effect on the choice of an architecture for a system. For example, in cases where high performance is a priority, techniques for reducing the time cost of inter-component communication must be used, such as value objects in mobile apps and enterprise information systems (EIS) (Chapters 6–8). Similarly, in safety-critical systems (SCS), a required level of fault tolerance determines the choice of an architecture based on redundancy (Chapter 10).

Table 1.2 summarises some quality of service properties and what kinds of system they are particularly relevant for.

One QoS property may conflict with another, for example increasing the modularity of a system to enhance maintainability may impair performance. Some fault-tolerance architectures such as ‘hot standby spares’ impact on efficiency (Chapter 10). In general a tradeoff between different properties may be necessary. Any such tradeoff should be documented with a rationale to justify the choices made.

Table 1.2 Quality-of-service properties

<i>QoS property</i>	<i>Description</i>	<i>Relevant for</i>
<i>Availability</i>	System functionalities are always/almost always available to clients	EIS; protection/shutdown systems (SCS)
<i>Efficiency</i>	System makes efficient use of computational resources	Mobile apps; ML systems; embedded systems
<i>Evolvability</i>	Low effort is required to evolve the system to meet new requirements	All systems
<i>Fault tolerance</i>	System can continue to function despite one or more failures	EIS; SCS; microservices
<i>Maintainability</i>	Low effort is required to maintain the system	All systems
<i>Responsiveness</i>	System responds in short time to input events/client request	Mobile and web apps; reactive systems
<i>Performance</i>	The system meets time and memory bounds for its operation	All systems
<i>Scalability</i>	The system can successfully serve large numbers of simultaneous users	EIS; web apps; web services
<i>Security</i>	The system is protected from intrusions and security attacks	All systems
<i>Transactionality</i>	Key processes in the system execute as indivisible units	EIS; SCS
<i>Trustworthiness</i>	System can be trusted to produce accurate results for critical purposes	SCS; ML systems
<i>Usability</i>	Interaction with system is supported in a way that is natural for users and does not require excessive user effort	Web apps; mobile apps; EIS

1.6 Architectural design approaches

In this book we do not prescribe any particular approach for creating an architectural design, instead we provide a ‘toolkit’ of architectural styles and patterns which can be used with different approaches and for systems in a variety of domains. We generally follow the clean architecture principles (Chapter 2) and recommend that components are defined so that they have a definite role (such as Presenter, Controller, Interactor, etc) in the clean architecture terms, and so that they belong to exactly one of the clean architecture levels.

A leading architectural design approach is *Attribute-driven design* (Bass et al, “Software Architecture in Practice”). This involves an iterative process by which the architectural choices for parts of a system are resolved based on the *architecturally significant requirements* for the selected part. These are all the quality of service attributes and business goals that relate to the selected part.

Model-driven development approaches (Brambilla et al, “Model-Driven Software Engineering in Practice”; Lano, “Agile Model-Based Development Using UML-RSDS”) tend to favour a top-down decomposition approach for architectural construction, whereby major subsystems and their interfaces are determined first, and then decomposed into smaller subsystems until indivisible modules appear at the lowest level.

Constraint/Invariant-based design uses the logical invariants of the system to guide the choice of an implementation which maintains the invariants [6], [32], [48]. This approach needs specialised tool support when it is used to generate complete executable code, however the idea of invariants between parts of a system also underlies various design patterns, such as the Observer pattern (which ensures that the state of data views are a function of the observed data) and architectural patterns such as Blackboard (Chapter 5) and MVVM (Chapter 6). Invariants between UI data and business data express how business data should be presented by the UI, whilst invariants on the business data are integrity/consistency constraints which may require checking by several parts of the system, and lead to the definition of components such as validation beans (Chapter 8). Transactions (Chapter 7) are often designed in order to ensure that critical business invariants are maintained. Service Broker (Chapter 9) maintains a constraint that all services provided to the client meet the client specification. Reactive control systems are usually intended to maintain control invariants by means of a monitor-decision-command cycle (Chapter 10). The Liskov substitution principle (Chapter 2) concerns the satisfaction of superclass properties by a subclass via an invariant (the refinement relation) which relates their states.

Summary

This chapter has introduced the key elements of software architectures: components, connectors and configurations.

UML component diagrams have been introduced, consisting of components, connectors and interface specifications.

Desirable properties of components and architectures have been described, including encapsulation, reusability, substitutability, sustainability, maintainability and extensibility. Quality of service concepts have been introduced.

Alternative architectural design approaches have been described.

Further Reading

A brief introduction to UML is given by:

- Martin Fowler, UML Distilled (3rd Edition), Addison Wesley, 2003.

Other books on software architecture include:

- Len Bass, Paul Clements, and Rick Kazman: Software Architecture in Practice, Addison-Wesley, 2021.
- Humberto Cervantes and Rick Kazman, Designing Software Architectures: A Practical Approach, Addison-Wesley, 2016.
- Martin Fowler – Patterns of Enterprise Application Architecture, Addison-Wesley, 2003.
- John Cheesman and John Daniels: UML Components, Addison-Wesley, 2000.
- Robert Martin – Clean Architecture, Prentice Hall, 2018.

Books on model-driven engineering include:

- Marco Brambilla, Jordi Cabot, Manuel Wimmer – Model-Driven Software Engineering in Practice, Morgan & Claypool, 2012.
- Kevin Lano – Agile Model-Based Development Using UML-RSDS, CRC Press, 2016.

Exercises

1

Draw an interface specification for a mailing component which provides operations to *send* an input string text message, *setMailHost* to set a mail host to an input string value, and *setMailPort* to set a mail port to an input integer value.

2

Draw the component diagram of a component *C* which requires interfaces *IA* and *IB*, and provides an interface *IC* defined to have operations *op1*(*x* : *int*) : *String*, *op2*(*s* : *String*, *i* : *int*), *op3*(*s1* : *String*) : *String*. Include an interface specification of *IC*.

3

Draw a component diagram and the interface specifications for a software component *TradeManagement* in a finance system.

The component provides a single interface *TradeI* with update operations *setTrader*(*traderId* : *String*), *setAccount*(*accountId* : *String*), *setAsset*(*asset* : *String*), *setTradeKind*(*tr* : *int*), *setQuantity*(*x* : *int*) and *executeTrade*() : *String*.

The component requires an interface *ExchangeI* with operations *submitTrade(tradeType : int, asset : String, quantity : int) : String* and *cancelTrade(id : String)*.

4

If component *M* has provided interface *P* and required interface *Q* defined as:

```
interface P {
    operation op1(x : int) : int;
}

interface Q {
    operation opr1(y : double) : double;
    operation opr2(z : String) : int;
}
```

can *M* be substituted by a component *N* which has the following provided interface *R* and required interfaces *T* and *W*? Explain your answer.

```
interface R {
    operation op1(x : int) : int;
    operation op2(x : int) : void;
}

interface T {
    operation opr1(y : double) : double;
}

interface W {
    operation opr2(z : String) : int;
}
```

5

A system involves two related component interfaces which support file processing and iteration over collections (Figure 1.13). Define revised interface specifications which facilitate the use of either files or iterators for iteration processing. Draw a component diagram showing the revised interface specifications.

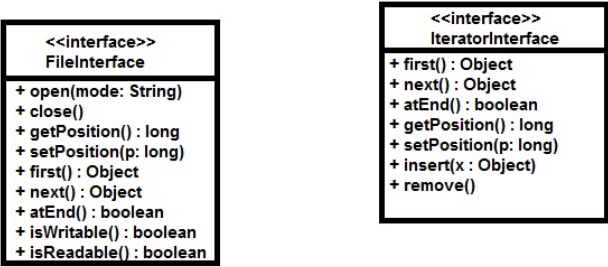


Fig. 1.13 File and Iterator components

6

Can the component *C1* in Figure 1.14 be substituted for component *C2*? Give a reasoned argument to explain your answer.

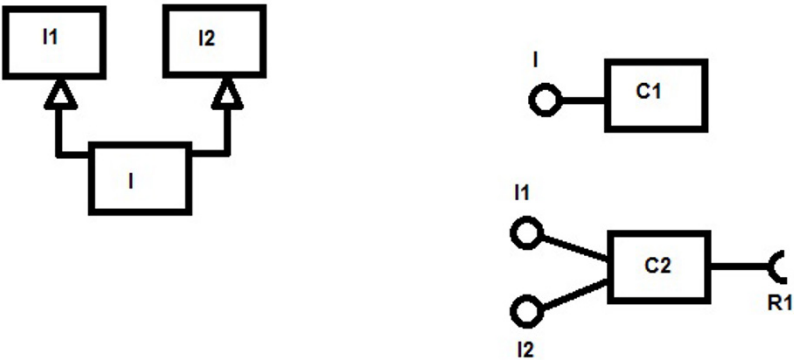


Fig. 1.14 Components and interfaces

7

Explain why the architecture of Figure 1.9 is likely to be more appropriate in general for a holiday reservation system than the original version of Figure 1.8.



Chapter 2

Introduction to Clean Architecture Concepts

Abstract This chapter covers the key concepts of the clean architecture approach to architectural design, and defines some quality measures and quality aspects of architectures with respect to this approach.

2.1 Motivation for the clean architecture

As we discussed in the previous chapter, architectural decisions are often driven by the quality of service requirements of a software system, the non-functional requirements such as performance, security, reliability, maintainability, evolvability, etc. which concern how the system provides its services. Different requirements and priorities on requirements lead to different architectures. The clean architecture is oriented towards improving maintainability and evolvability, and does this by restricting the *dependencies* between components, where a component X depends on component Y if X knows the name of component Y (or the name of a class or interface contained in Y).

At the implementation level, this means that if X is independent of Y , then X does not need to *import/include* Y , and changes to Y do not require recompilation of X . (e.g., the components can be in separate .jar files).

This dependency relation is distinct from the calling relationship between components: component X may call component Y via an interface RX *defined in* X , and so could be independent of Y . In this situation, Y depends on X because it needs to implement and hence refer to interface RX .

The clean architecture uses this idea of *dependency inversion* to enforce the core principle of the *dependency rule* [63]:

Platform-specific components can depend on platform-independent components but not conversely.

The dependency rule helps to insulate the core business components of a system from changes to more peripheral aspects such as data persistence tech-

nologies and UI technologies. Thus the system becomes more maintainable and evolvable as the effort involved in technology changes is reduced.

In general there may be a range of degrees of dependency between components X and Y :

- Independence: no assumptions or knowledge about Y are present in X .
- Syntactic dependence: some names of classes/interfaces of Y are used in X , or Y itself is referred to in X .
- Semantic dependence: in addition, X depends on the precise semantics of an element of Y , such as a specific operation.
- Code dependence: X depends upon a specific coding of an element of Y .

The stronger the degree of dependency of component X on component Y , the more likely it is that changes to Y will affect X and hence also lead to changes in X .

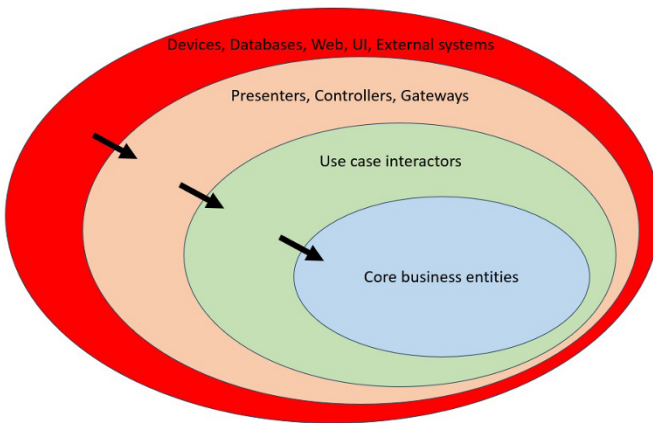


Fig. 2.1 Clean architecture system organisation (From [63])

Figure 2.1 shows the clean architecture organisation of systems as layers or circles of components at the same level of abstraction from specific technologies. The arrows denote the admissible dependencies between layers. At the highest level of abstraction are components representing the core business entities, which are independent of technologies and even independent of the use cases of a particular application. For example, in any banking application we could have classes *Account*, *Customer*, *Transaction* representing these domain concepts. The next layer consists of components for the business operations – components which implement the system functionality of the use cases of the particular application. These depend on the core business entities, but are independent of particular UI/data storage technologies. For example, a banking application could have use cases to create a new account for a customer, or to close an account. The algorithms for these procedures

reside in components termed the use case ‘interactors’ for such banking operations. Usually, there is a different interactor for each different category of user of the system, i.e., for different use case actors (cf., the Single Responsibility Principle below).

The next layer consists of intermediary components which link the business-specific parts of the system to general-purpose UI and data storage technologies. Typically there are data-access objects (DAOs), otherwise known as *gateways* or database interfaces, to manage a specific data storage technology such as a relational database or cloud datastore. There are also components which manage UI interaction: *controllers*, which control the UI navigation sequence and UI logic, and *presenters*, which convert business data into formats (such as web page content) for display via particular UI technologies.

Finally at the outermost level of the system there are the concrete technologies and specific platform components such as relational databases, web pages and device drivers.

2.2 The clean architecture: SOLID principles

The clean architecture defines several principles of good component design and architecture design, which encompass many different kinds of system. The acronym SOLID is used for the following five principles of component design.

2.2.1 SRP: Single responsibility principle

This principle states that *each component should have only one reason to change*. A more explicit statement is that *each component should provide services for a single actor*. SRP aims to avoid the situation where a component has overloaded responsibilities, trying to perform too wide a range of capabilities, for different clients, and hence becomes excessively large and complex and expensive to maintain. When a component is used by different clients for substantially different purposes, then conflicts can arise, and changes made to satisfy one client may break functionality or behaviour expected by another client.

For example, we could have a component *OclIterator* which performs iteration operations *next*, *previous*, *hasNext*, etc over data aggregates, and we could overload *OclIterator* to also iterate over database result sets, representing sets of database table rows produced by a database query (the *columns* attribute and the final five operations in Figure 2.2). Although these func-

tionalities are conceptually related by the idea of iteration, they are also liable to change for unrelated reasons:

- Standard iteration processing may need to be extended to a new kind of data structure, e.g., to iterate over graphs or trees.
- Database result set processing may need to be extended to handle more sophisticated database processing, e.g., to modify the database via the result set.

Ultimately we could end up with a highly incohesive component, where many operations have entirely separate functionality for the two different kinds of processing (standard iteration and database result set processing).

OclIterator <component>
markedPosition: int = 0 position: int = 0 elements: Sequence = Sequence{} generatorFunction: Function(int, OclAny) = lambda _x0 : int in 0 columns: Sequence(OclAttribute) = Sequence{}
hasNext(): boolean isAfterLast(): boolean isBeforeFirst(): boolean hasPrevious(): boolean nextIndex(): int previousIndex(): int getPosition(): int markPosition(): void moveForward(): void moveBackward(): void moveTo(...): void setPosition(...): void movePosition(...): void moveToMarkedPosition(): void moveToFirst(): void moveToLast(): void moveToStart(): void moveToEnd(): void <u>newOclIterator Sequence(...): OclIterator</u> <u>newOclIterator Set(...): OclIterator</u> <u>newOclIterator String(...): OclIterator</u> <u>newOclIterator String String(...): OclIterator</u> <u>newOclIterator Function(...): OclIterator</u> getCurrent(): OclAny set(...): void insert(...): void remove(): void next(): OclAny nextResult(): OclIteratorResult previous(): OclAny length(): int at(...): OclAny close() getCurrentFieldByIndex(...): OclAny getColumnName(...): String getColumnClassName(...): String getColumnCount(): int setCurrentFieldByIndex(...)

Fig. 2.2 OclIterator component

The clean architecture refers to this principle as the common closure principle (CCP) when considering the allocation of classes to components: “a component should have only one reason to change” [63]. This means that a component should only contain classes that are likely to change for the same reasons.

2.2.2 OCP: Open-closed principle

This principle states that components should be changed by adding new capabilities, and not by modifying existing capabilities. This provides a strong assurance of backwards compatibility for existing clients of the component (they can continue to use it without even being aware that it has been updated), whilst retaining some flexibility to enhance and extend the component for new services and clients. A test suite for a component can be reused without change for the extended component, and additional tests are only needed for the new functions.

The Swift language provides a facility for this concept by enabling existing classes to be extended with new features, by an `extension C` declaration separate from the original definition of class `C`.

This property is especially useful during development of an application by different developers each contributing code to a codebase (e.g., in a github repository). Provided that each developer only adds code and does not modify existing code, then their independent changes can usually be merged without conflicts.

The OCP is aligned with the concept of component substitutability, because the latter allows components to be extended with new capabilities either as new provided interfaces, or as extensions of existing provided interfaces. But existing provided functionality cannot be deleted, nor modified in ways which break the semantics of the existing provided interfaces. In addition, new required interfaces or operations are not permitted. Thus components are open for extension but not for changes that break existing semantic assurances.

2.2.3 LSP: Liskov substitution principle

This principle is a strong constraint on how superclasses and subclasses can be related [57]. Effectively it says that all subclass behaviour should satisfy the superclass specification.

Liskov substitutability means that if superclass S and subclass C are related by an *extends* or *implements* dependency from C to S , then for each operation op of S , there should be an operation op of C with the same name and signature and such that:

1. The version $C::op$ of op defined in C should accept any input accepted by $S::op$, i.e., it should have defined behaviour for any data where $S::op$ has defined behaviour.
2. On input data where $S::op$ is defined, the results and effects of $S::op$ and $C::op$ should be indistinguishable by a caller of the operation.

The LSP applies also to an interface S and implementation C . In this case, the LSP means that clients of a component with a provided interface defined by specification S should not be surprised by any functionality exhibited by an implementation C of S . The semantics of S could be defined using OCL [71], giving a precise mathematical specification of the intended functionality of its operations. The LSP states that any implementation C of S should be entirely consistent with the specification, so far as any client could distinguish. I.e., no functional test could detect any inconsistency between C and S .

Likewise, S and C could both be interfaces, with C an extension of S . This situation arises in dependency inversion, where S is a client required interface, and C a supplier provided interface. In Figure 2.5 this means that *ProvI* should be Liskov-substitutable for *ReqI*. The LSP enables the client to be semantically independent of the supplier: all necessary semantic assumptions are made upon S (owned by the client), not upon C .

As an example of LSP, an interface of a general *Collection* with operations *addElement* and *hasElement* could be defined by the UML/OCL specification:

```
interface Collection
{ attribute elems : Set(OclAny) := Set{};

  operation addElement(x : OclAny)
  pre: true post: true
  activity:
    elems := elems->including(x);

  operation hasElement(x : OclAny) : boolean
  pre: true
  post:
    result =
      if elems->includes(x) then true else false endif;
}
```

OclAny is the OCL ‘universal type’, analogous to *Object* in Java.

Users of *Collection* can call the operations *addElement* and *hasElement*. The specification tells users that *col.hasElement(x)* should return true after *col.addElement(x)* has been executed at least once.

An extension *OrderedCollection* of *Collection* could be defined, which enhances *Collection* by defining an ordering of elements:

```
interface OrderedCollection extends Collection
{ attribute seq : Sequence(OclAny) := Sequence{};

  operation addElement(x : OclAny)
  pre: true post: true
  activity:
    seq := seq->append(x);
```

```

operation hasElement(x : OclAny) : boolean
pre: true
post:
  result =
    if seq->includes(x) then true else false endif;
}

```

This version is Liskov-substitutable with *Collection*: a user cannot distinguish the behaviour of the two interfaces by executing any sequence of calls of the two operations on corresponding instances of the interfaces. This should be intuitively clear, but could be formally proved by checking that the *refinement relation*

$$elems = seq \rightarrow asSet()$$

is valid between the states of the interfaces (i.e., this invariant should be established by the joint initialisation of the classes, and preserved by the joint execution of the respective *addElement* versions). The $\rightarrow asSet()$ operator removes duplicates from its argument, e.g.:

$$Sequence\{1, 3, 1, 2, 3\} \rightarrow asSet() = Set\{1, 3, 2\}$$

However, if an operation *size()* : *int* was added to both interfaces, Liskov-substitutability would no longer hold: *elems* of *Collection* may be strictly smaller than the *seq* of *OrderedCollection*, after the same series of operations is applied to new instances of each. Thus the operations defined as

```

operation size() : int
pre: true
post: result = elems->size();

```

in *Collection* and as

```

operation size() : int
pre: true
post: result = seq->size();

```

in *OrderedCollection* would give different results after a sequence of calls such as *addElement*(1), *addElement*(1). In order to achieve semantic compatibility, the *size* definition in *OrderedCollection* would need to be:

```

operation size() : int
pre: true
post: result = seq->asSet()->size();

```

The Liskov subtype relation needs to be satisfied in two places in an architecture:

1. Between the required interface on one side of an assembly connector, and the provided interface on the other side.

2. Between the provided interface of a component and its internal implementation of the interface.

Because Liskov substitutability is transitive, these conditions also mean that the component implementation satisfies the expected semantics of each required interface linked to a provided interface of the component.

Component substitutability requires that a provided interface $P1$ of the replaced component must be preserved or replaced by a Liskov-substitutable interface $P2$ of the replacing component. For required interfaces the direction is reversed: any replaced required interface $R1$ must be Liskov-substitutable for the replacing required interface $R2$ (Figure 2.3).

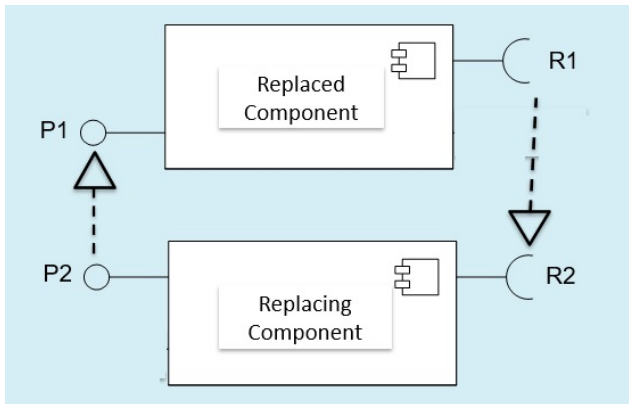


Fig. 2.3 Component substitutability with LSP

A more restrictive version of the LSP requires additionally that any new operations in the subclass C should be expressible as combinations of existing S operations. The reason is that clients should not be surprised by any changes of state which can be observed in an object obj of C , if they only have available a specification S of the superclass (obj could be shared with another client who can call the additional operations of C). It also means that invariants of S remain invariants of C . However, this stronger version of LSP clearly limits the OCP and prevents any genuinely new operations being introduced (for example, a *remove* operation could not be added to a subclass of *Collection* above). For most purposes the more general version of the LSP is satisfactory.

2.2.4 ISP: Interface segregation principle

This principle states that components should not depend on components or interfaces that they don't use: each such dependency causes wasted effort and

resources when maintaining and deploying the system, and complicates the build and deployment processes.

For example, a component could declare that it requires the services in interfaces *R1* and *R2*, but its code only in fact uses *R1*. This situation may result from evolution of the system, e.g., to provide functionality within the component which previously was provided externally, but the developer forgot to remove the redundant dependency. A more subtle case is when two supplier modules are partly duplicates of each other, and the client could satisfy all its requirements by using only one of them. This situation requires careful inspection and refactoring of code in order to remove the excess dependency. Finally, there may be cases where a relatively simple external service is used, which could instead be implemented within the module (e.g., the notorious ‘padleft’ function case, where a simple function to pad a string to the left was used from an open-source library by a large number of applications either directly or indirectly, causing havoc when the creator of the function withdrew it from the library [21]).

As a general principle, making components more autonomous is a good policy, however this may create redundancy (several components each implementing the same functionality) and lead to code bloat and excessively large and complex modules. Thus some balance needs to be achieved between complete autonomy and excessive component dependencies.

From the viewpoint of the developer of a supplier module, the ISP indicates that distinct groups of operations of the module, that are likely to be used by different clients, should be separated into different provided interfaces. Otherwise, if all the operations were declared in a single large provided interface, all clients would be forced to depend on all the operations.

The clean architecture also refers to this principle as the common reuse principle (CRP) at the level of components: “Don’t force users of a component to depend on things they don’t need”. The guideline in this case is only to combine classes in one component if the classes are likely to be used together, i.e., because they are closely related.

2.2.5 DIP: Dependency inversion principle

This principle is more accurately referred to as a technique, a means by which the direction of a conventional calling dependency is reversed. This technique is used to enforce the dependency rule that high-level components should not depend on low-level components, instead low-level components should depend on high-level.

Figure 2.4 shows a poor-quality architecture where a client module directly calls operations defined in a supplier module. It is therefore directly dependent on the specific supplier and is impacted by any change in the supplier.

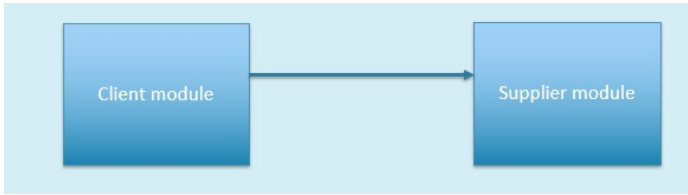


Fig. 2.4 Direct client-supplier dependency

Instead, provided and required interfaces can be introduced, so that the dependency is now indirect: (i) from the client to its required interface *ReqI*, (ii) from the implementation of the supplier upon *ProvI*, and (iii) from *ProvI* to *ReqI*. The final dependency occurs because *ProvI* needs to provide all the operations required by *ReqI*, in order that the assembly connection is valid. Figure 2.5 shows this version.

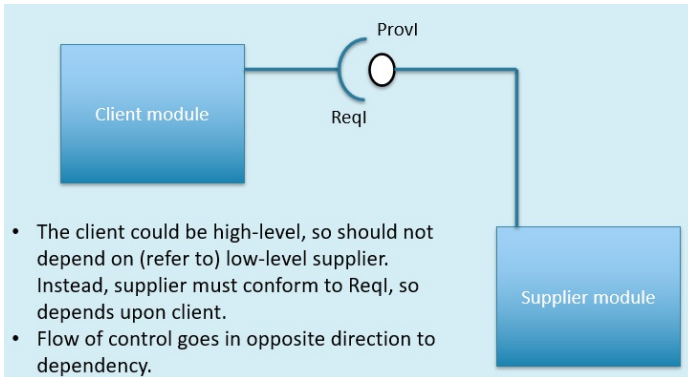


Fig. 2.5 Client-supplier relationship with assembly connector

Dependency inversion therefore occurs if we place the required interface *ReqI* into the client module, and the provided interface *ProvI* into the supplier: the supplier module then has a dependency to the client (Figure 2.6). The dependency runs in the opposite direction to the calling relation: even though the client calls the supplier, no name of any supplier element occurs in the client.

This mechanism does not need to be used between every pair of client/supplier modules, only in situations where the client is at a higher abstraction level than the supplier, for example, when the client is a use case interactor and the supplier is a gateway or presenter in the terms of Figure 2.1.

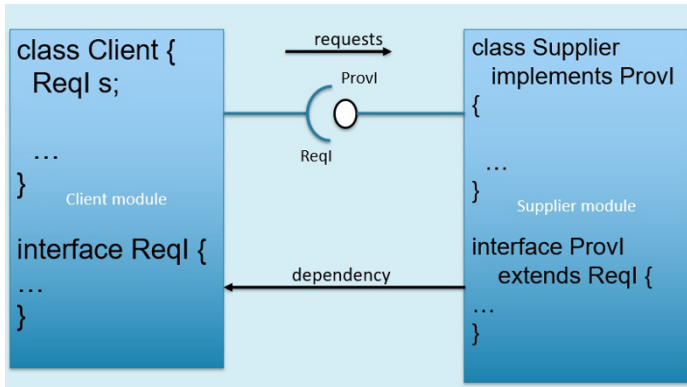


Fig. 2.6 Client-supplier relationship with dependency inversion (Java)

2.2.6 Architecture design principles

The clean architecture states the following principles for the overall structure of an architecture:

- **ADP: Acyclic dependencies principle:** There should be no cycles in the component dependency graph.
- **The Dependency Rule:** Dependencies should point from low-level components to high-level, not from high-level to low-level.

The motivation for the ADP is that cyclic dependencies cause additional costs and delays during development and maintenance: the developer/team responsible for one component is affected by updates to the modules they depend upon, and their work also affects the developers of those modules! Cyclic dependencies may also cause infinite recursion at runtime: an operation *opA* of module *A* can invoke operation *opB* of module *B*, which invokes *opA*.

The OCP may help to reduce the problems caused by cyclic dependencies: if it is possible to reach agreement that existing operations will not be changed (at least as regards to their signatures and semantics), then the danger of existing code in one module being invalidated by changes to another module are reduced. However this ‘freezing’ of interfaces can be too restrictive in practice, e.g., if it is realised that an operation needs to have a different signature to the originally defined signature.

The dependency rule is motivated by the consideration that components that are relatively stable should not depend on components more liable to change/that need to change more frequently. E.g.: a business tier entity bean should not depend on a database, instead the dependency should run in the opposite direction.

2.3 Component boundaries

In addition to controlling dependencies between components, the clean architecture is concerned with the definition of *boundaries* between components.

Component boundaries consist of:

- The input boundaries, defined by provided interfaces of the component and the data types used by these interfaces (i.e., as parameter types of their operations).
- The output boundaries, defined by required interfaces and their data types.

Normally, the provided interface of a component belongs to the component, but a required interface may belong to either the client or supplier (if the required and provided interfaces of an assembly connection are the same) or to neither (for example, an interface used by several subcomponents of a composite component would belong to the composite).

The types of data transferred into or out of a component may be basic types such as numbers, strings or booleans, but restricting to such types leads to large numbers of parameters when business data (such as customer or product data in an e-commerce site) is being used. Instead, the data for business entities can be packaged up into *value objects* or *data transfer objects* (Chapter 6), which are simple class types which enable all data values of a given business object (such as an individual customer or product) to be stored in a single object value. Because any part of the application may refer to such a type, say *EValueObject* for business entity *E*, the type must itself belong to the business entities tier.

2.4 Technical debt at the architectural level

The concept of *technical debt* was originally introduced for source code, and refers to the long-term cost impacts of poor coding practices such as ‘copy-and-paste’ duplication of code blocks. The concept is analogous to financial debt: someone takes out a loan (a developer rapidly writes code without regard to code quality) which ultimately has to be paid back. It can either be paid back in a lump sum (the code is substantially refactored to remove the flaws – an expensive process) or interest payments are made over a long period (there are added costs of maintaining the code due to its embedded quality flaws or technical debt).

Quality flaws are also termed ‘code smells’, because they indicate that something is not quite right with the code. They are indicators that technical debt exists in the code.

Examples of quality flaws associated with technical debt at the code level are:

- DC – duplicated (cloned) code, which leads to duplicated maintenance actions and effort.
- CC – excessive code complexity, in terms of number of control flow branches, which leads to high effort being expended to understand, change and test the code.
- ENO – excessive number of operations in a class.
- EPL – excessive number of parameters in an operation, including local variables.
- MGN – ‘magic numbers’ or explicit literal values written in procedural statements instead of being declared as constants, such as 3.14 being written for the value of π in a statement

```
area = 2*r*3.14;
```

- EFO – excessive *fan-out* of a function: the function calls too many other functions.
- EFI – excessive *fan-in* of a function: the function is called by too many other functions.

Such code-level technical debt indicators apply to the implementations of components within an architecture. In addition there are architectural flaws which contribute to technical debt at the architecture level [11]:

- Embedded platform-specific code, such as direct calls to a database connection within a business entity component – this is a violation of the clean architecture dependency rule.
- Platform-specific data being passed between components, such as database result sets or HTTP request packets – this also violates the dependency rule because any user of the data becomes dependent upon the specific platform.
- Excessive interface size, with too many operations (e.g., more than 10) in one interface – this indicates potential violation of the ISP or SRP.
- Excessive numbers of provided interfaces – this indicates a module which is trying to perform too many functions, and can indicate violation of the SRP.
- Excessive numbers of required interfaces – this can indicate a module which has poorly-managed requirements, and violation of the ISP.
- Cyclic dependency or calling relationships between modules – this represents a violation of the ADP.

EFI and EFO also apply to the connections into and out of components. Components which have both EFI and EFO are potentially the cause of problems because they are likely to be excessively complex – analogous to the ‘God class’ code smell at the code level [37].

More generally, there may be incorrect partitioning of components, with different forms of functionality combined in a single component, such as UI coding and business operation coding. Technical debt may arise due to the

failure to apply relevant architectural patterns, such as Value object (Chapter 6) or Front controller (Chapter 8) to reduce complexity and duplication within an architecture.

If a quality flaw is identified, a strategy for reducing or removing this flaw should be determined. Dependency rule flaws can be addressed by proper segregation of the platform-specific and platform-independent code into separate components. Likewise, partitioning flaws can be addressed by appropriate code segregation and splitting of components. Cyclic dependencies require careful refactoring of functionality and potentially the creation of a new component, in order to remove the circularity. SRP issues can be addressed by splitting the excessively complex component. Relevant architectural patterns should be selected and applied to address specific flaws such as multiple entry points to an application (Front controller) or inefficient transfer of data (Value object).

Further Reading

- <https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles>, 2023.
- Clean Architecture, R. Martin, Prentice Hall, 2018.
- Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern, 2004.
<http://www.martinfowler.com/articles/injection.html>

Exercises

1

Explain why there may be conflicts between ISP and SRP.

2

Which of the following restrictions on classes/components are necessary for the *dependency rule*?

1. Concrete classes should not be inherited from.
2. Required interfaces of components should be made explicit.
3. Provided interfaces of components should be made explicit.
4. Business components should depend only on data repositories.
5. Object creation should be managed via Abstract Factories.

3

If component A has a single provided interface AI with operations $setx(v : int)$ and $getx() : int$, and a single required interface $ANotify$ with operation $xchanged(v : int)$, can A be replaced in all contexts by a revised version which has an extended AI offering also an operation $lock()$, and an extended required interface with an additional operation $locked()$? Explain your answer.

4

Explain the reason for the relation between the required interfaces in Figure 2.3.

5

If class S and subclass C are considered as components, does Liskov substitutability of S by C also ensure component substitutability between the classes as components? Explain your answer.



Chapter 3

Development Methods

Abstract This chapter describes the use of architectural design within model-driven engineering (MDE) and agile development processes. As an example of MDE, we give an MDE-based approach for the architectural design of simple business systems. We also discuss the issues involved in using architectural design in an agile development approach.

3.1 Model-driven engineering

Model-driven engineering (MDE) is a software development approach which focusses upon the use of *models* as key software artefacts: models which describe the software requirements, or which describe the specification and design at different levels of abstraction and detail. Typical software modelling notations are UML class diagrams, or UML use case models. In this book we have been using UML component diagrams to model software architectures.

Models have several purposes:

- To facilitate communication and understanding about the system within the development team, and between the team and stakeholders.
- To explore alternative designs.
- To perform formal analysis, e.g., to check that clean architecture principles are being followed.
- For automated test case generation – *model-based testing* (MBT).
- As a starting point for automated code generation.

An advantage of software modelling is that issues and alternatives for building the system can be explored before detailed coding begins. For architectural design, component diagrams provide a view of the possible ways in which a system can be decomposed into components, and these components then assembled into the system as a whole. Issues such as the need for de-

dependency inversion between two components can be recognised and resolved at this level of abstraction, prior to coding the components (Section 3.4).

Another use of models for software architecture is to *derive* an outline architecture from early-stage models such as business concept models and use cases, using these high-level models to infer what components, interfaces and connections are needed in the architecture of the system (Section 3.3).

Finally, from detailed models, complete code of a system can be generated, including interfaces, components and dependencies conforming to a particular architectural style and to the clean architecture principles (Section 3.5). Test cases can be generated based on the interface specifications of a component, these tests can then be applied to any candidate implementation of the interfaces, in order to check that the implemented behaviour conforms to the specification.

3.2 Agile development and architectural design

Agile development has become one of the most widely-used software development approaches in practice [38]. The principles of agile development include:

- Incremental work on requirements, coding, integration in short cycles (termed *sprints* in the Scrum¹ agile method).
- Responsiveness to changing requirements.
- Close collaboration between the development team and the customer.

Figure 3.1 shows a typical agile development process, based on the Scrum method [80].

The key events in a Scrum-like agile process are:

- *Initiation*: identification of team membership and stakeholders, and of relevant background information and resources for the project. This phase aims to understand the problem that needs to be solved. In this stage conceptual models could be created, such as outline class diagrams and top-level component diagrams.
- *Sprint Planning*: performed by the project team before a sprint, the team agrees the tasks to be worked on in the sprint and in which order.
- *Daily stand-up meeting*: a brief meeting at the start of each day, this organises the activities of the team, reviews sprint progress and identifies issues.
- *Sprint Review*: this presents completed work to stakeholders at the end of the sprint.

¹ <https://www.scrum.org>

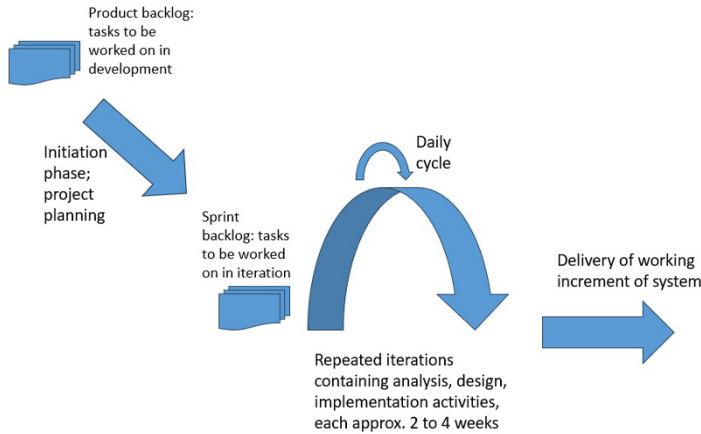


Fig. 3.1 Agile development process

- *Sprint Retrospective*: after the sprint review, and before the next sprint planning. Analyses the achievements of the sprint, and discusses ideas for improvement of the development process.

Each team includes a team master or leader, responsible for organising the agile development process, and a Product Owner (customer representative), who can give direct feedback on the correctness of the developed solutions.

Key artefacts of an agile method include:

- *User story*: a unit of requirements, with an identified user, user requirement for the system, and a rationale for the requirement.
- *Product backlog*: an ordered (in terms of priority) list of user stories which the project is responsible for implementing.
- *Sprint backlog*: an ordered list of user stories to be completed in the next sprint.

Within an iteration, the development team can use a *task board* showing the tasks to do, tasks in progress and completed (Figure 3.2).

Agile methods emphasise the production of code over documentation, with documentation being minimised where possible. Architectural models could be sketched at the initiation stage of an agile development process, and be continually refined during the iterations of the development process. The models would identify what components/subsystems are expected to exist in the system, and therefore the models would help to coordinate the work of the team members. A specific team member could be designated as the project architect, or the architect may be external to the team and responsible for ensuring that its work is consistent with an existing system architecture. The architect would be the principal author and maintainer of the architecture models.

Product Backlog	Iteration Backlog	Coding	Under Test	Completed
<i>task9</i> <i>task10</i> <i>task12</i> <i>task13</i> <i>task14</i> <i>task15</i>	<i>task6</i> <i>task7</i> <i>task8</i> <i>task11</i>	<i>task4</i> <i>task5</i>	<i>task2</i>	<i>task1</i> <i>task3</i>

Fig. 3.2 Task board example

Adoption of the clean architecture principles should increase agility. In particular, the dependency rule ensures that changes to low-level components do not impact high-level components. For example, a new requirement for a mobile app to allow multiple users of the app on different devices to share data could lead to a change in database technology – from a local SQLite database to a cloud datastore – but this change should not impact the core functional components of the system, which would use the database via a DAO (gateway) with dependency inversion, and hence would be independent of the specific database technology.

3.2.1 Architectural refactoring

In the context of architectural design an important agile practice is *refactoring*, the regular restructuring of software to improve it, to remove quality flaws such as redundancy, etc. [26, 28]. Code-level refactoring does not normally change the functionality of a software component, but instead only rewrites its implementation. Thus the refactoring of the internal code of a component should not affect clients of the component.

The software architecture itself may be refactored, e.g., to break an excessively large interface *IX* into smaller separate and more cohesive interfaces *IX1*, *IX2*, etc. In accordance with the clean architecture ISP, this would enable clients to be more selective about which operations they depend upon (Figures 3.3, 3.4).

A direct calling connection between classes (Figure 2.4) can evolve into (i) a calling connection from the client to an extracted interface of the supplier; (ii) an assembly connection between the classes packaged into components (Figure 2.5); (iii) a dependency inversion assembly connection (Figure 2.6).

An initial architecture may evolve into a more elaborate form as development proceeds, eg., to transition from a Blackboard to a Model-view-

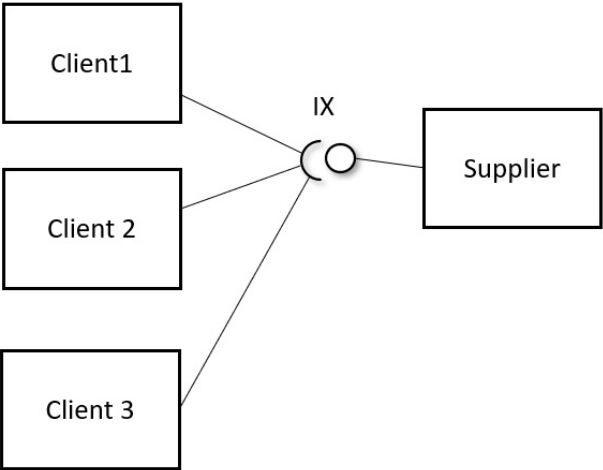


Fig. 3.3 Interface refactoring example (before refactoring)

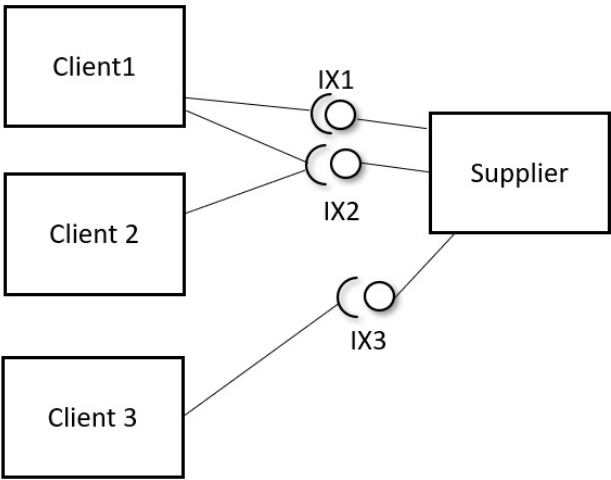


Fig. 3.4 Interface refactoring example (after refactoring)

controller architecture (Chapter 5). This type of change may involve the introduction of new components and some changes to component responsibilities, while leaving the overall organisation of the architecture unchanged. Radical change and reorganisation of an architecture should only be undertaken if it is clear that a mistaken choice has been made, that could not be sustained in the long term.

The focus of agile development upon coding rather than documentation, and upon rapid software production for immediate needs, can create challenges for the application of software architecture principles [5], [75]. Among these challenges are the difficulty of maintaining architectural models, especially of keeping them up-to-date with code. In an agile context, frequent requirements change can lead to many changes to architectures based on the requirements. We consider that one approach to address this conflict is to adopt an *agile MDE* approach [3], whereby development effort is focussed at the system specification level, with code artefacts being automatically synthesised from the specification. This enables rapid change to both specifications and code, and ensures that consistency is maintained between the specification and code, and between different code artefacts.

For example, in an agile MDE approach, the transition from a local to a cloud datastore for a mobile app can be accomplished by changing one stereotype in the app specification and regenerating the code [51].

We have included a toolset for agile MDE, *AgileUML*, on the supporting website. This toolset provides lightweight modelling capabilities, software quality analysis, refactoring operations, code generation for multiple target platforms, and reverse-engineering tools. A user guide for AgileUML is included in the appendix.

3.3 Derivation of architectures from requirements models

Instead of architectures being manually constructed, for certain kinds of system, such as simple business systems, it is possible to derive an appropriate architecture, semi-automatically, from requirements models. In particular, from outline class diagrams of the main business concepts of a system (business concept models) and the system use case models, the components and interfaces in the business tier of the system can be identified via a systematic process (Section 3.3.3).

3.3.1 Three-tier architectures

A common style of architectural design is the *three tier architecture* (Figure 3.5).

In this style the components of the system are divided into three layers or tiers:

1. The UI/Presentation layer, containing components to perform interaction with users.

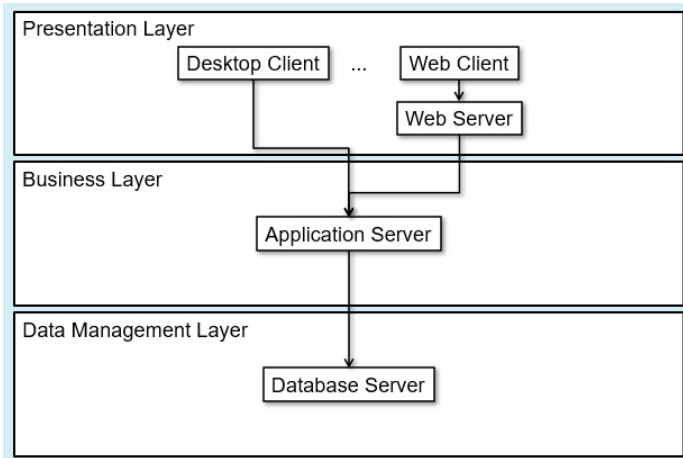


Fig. 3.5 Three-tier architecture system organisation

2. The business tier or functional core, containing the main business processing components.
3. The data management tier, containing components for managing persistent datastores such as relational databases, and also for interacting with remote services, such as business-to-business (B2B) services.

Notice that both the UI and data management tiers are platform-specific, whereas the business tier is (or should be) platform-independent.

Here we will focus on the organisation of the business tier and how this can be derived. The business tier is typically organised into two main sub-tiers (Figure 3.6):

1. Business operation components/system services, which are use case interactors in the sense of the clean architecture.
2. Business objects/entities, which are entity components in the clean architecture terminology.

The use case interactor components will have interfaces which represent the business use cases of the application, termed *system interfaces*, whilst the business entity components have interfaces for business operations independent of specific applications, termed *business interfaces* (Figure 3.7).

3.3.2 Requirements models

Requirements-level models are initial specifications of the data and operations which a system is responsible for. These models omit detailed data and

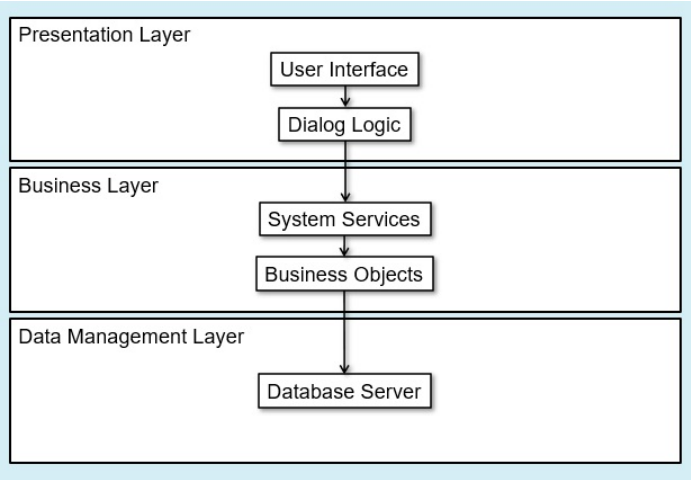


Fig. 3.6 Business tier organisation

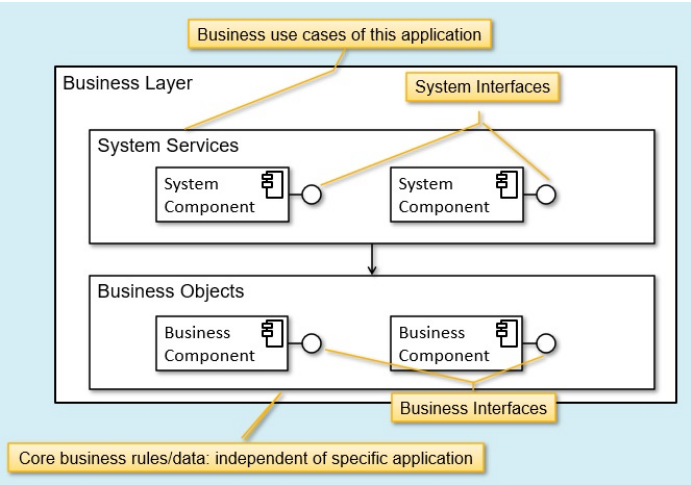


Fig. 3.7 Business tier components and interfaces

functional semantics, and should be comprehensible by stakeholders familiar with the domain.

3.3.2.1 Business concept models

System data is described in *business concept models*. These are high-level class diagrams which model all relevant concepts in the domain of the system and their relations. Concepts are represented as classes and relations are

represented as associations between classes. Associations are marked with their role multiplicities (* for many, 1 for one, 0..1 for optional, etc). The concept model may contain classes that are relevant in the domain but which may not be implemented in the application, e.g., because there is an existing system that already covers the processing involved with these classes. Attributes and operations of classes are omitted. Figure 3.8 gives a concept model for a room booking system.

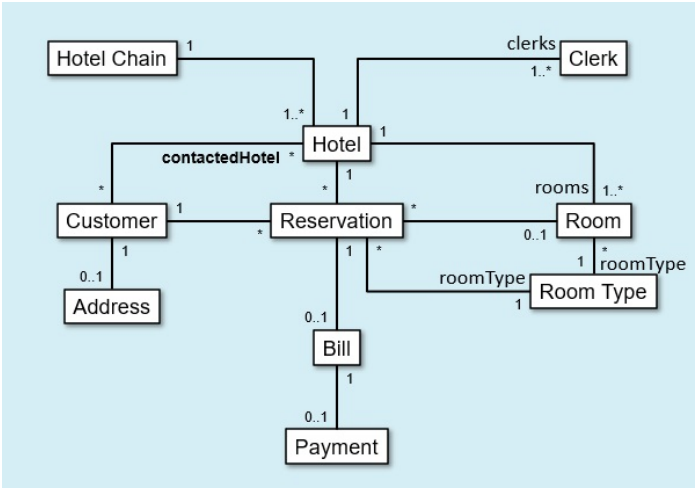


Fig. 3.8 Room booking concept model

In these graphical class diagrams, classes are shown as rectangles containing the class name, and associations as lines between the related classes, with optional role names at one or both ends, and multiplicities at one or both ends. Association multiplicities have the meanings described in Table 3.1.

Table 3.1 Association multiplicities

Multiplicity	Meaning
*	Each object at the other end can be linked to any finite number of objects at this end.
0..1	Each object at the other end is linked to at most one object at this end.
1	Each object at the other end is linked to exactly one object at this end.
1..*	Each object at the other end is linked to at least one object at this end.

In other words, the * annotation means that the role at that end is a set or sequence of objects of the class at the end, of unbounded size. The 0..1 annotation means that the role is a set/sequence of size ≤ 1 – essentially an optional object of the class. The 1 annotation means the role is a specific (non-null) object of the end class.

3.3.2.2 Use case models

Use case models define the high level functionalities or services offered by a system, represented as ovals linked to one or more ‘actors’ or participants in the use case. Actors represent a category of possible users/participants of the use case. Figure 3.9 shows three use cases of the room booking system, each of these is used by the Guest actor, representing a customer.

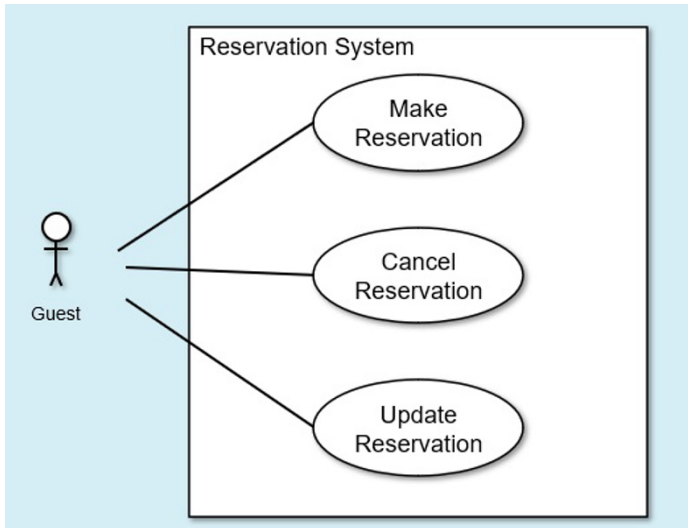


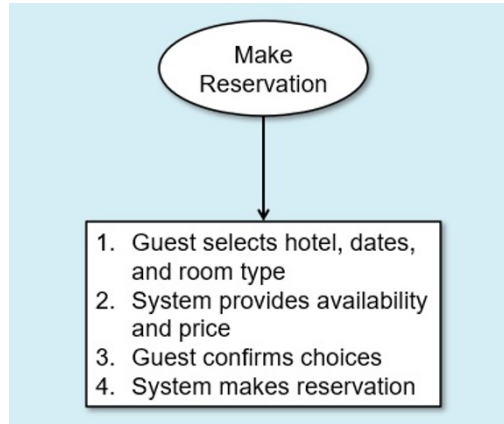
Fig. 3.9 Room booking system use cases

For each use case, there may be one or more *scenarios* of use, defining alternative sequences of actions which may occur when the use case is executed. Typically, a scenario consists of alternating user and system steps, involving a request by the user to the system, a response action by the system, followed by potentially further interactions between the user and the system (Figure 3.10).

3.3.3 Derivation of architectures from requirements

In this section we describe an approach to derive an initial architecture of a system from the system requirements models, for simple business systems. This approach is due to Cheesman and Daniels, “UML Components”, Addison-Wesley, 2001.

Fig. 3.10 Use case scenario



3.3.3.1 Derivation of business operation components

The use case interactor component (business operation component) that implements a given use case needs to provide operations to implement all the system steps of the use case. We define a separate interface (one per use case) which offers the system step operations of the use case, and we define these interfaces as provided interfaces of an interactor component which implements all the use cases for a given actor. For the room booking system, this means that there is a single *ReservationSystem* component handling all *Guest* use cases, each of which has a separate provided interface (Figure 3.13).

This style of architectural design satisfies the SRP of the clean architecture: each business operations component only has a single actor as its client. The ISP is also satisfied. However there may be duplication between interfaces, if two or more use cases share common system steps, and if there are a large number of use cases for a given actor, then the number of provided interfaces for their business operations component may be excessive. These issues may be addressed by rationalisation and merging of the provided interfaces of the use case interactor.

3.3.3.2 Derivation of business entity components

The business concept model is used to identify the business entities and their interfaces. The business concept model is analysed and refined to remove irrelevant concepts (i.e., concepts which do not need to be represented in computational form). Aspects which are already implemented in an existing system (e.g., billing services) are not included. *Core* and *dependent* types are identified:

- *Core types* directly correspond to primary business information, and will form the basis for entity component identification.
- *Dependent types* represent subordinate information of core types – instances of dependent types only make sense in relation to some core type instance.

In our example, the core types are *Hotel* and *Customer*, and *Room* and *Reservation* are dependent types.

There can also be *category types* (such as the room type in the room booking example). These are essentially enumerated types that classify other types. Figure 3.11 shows the refined business type model.

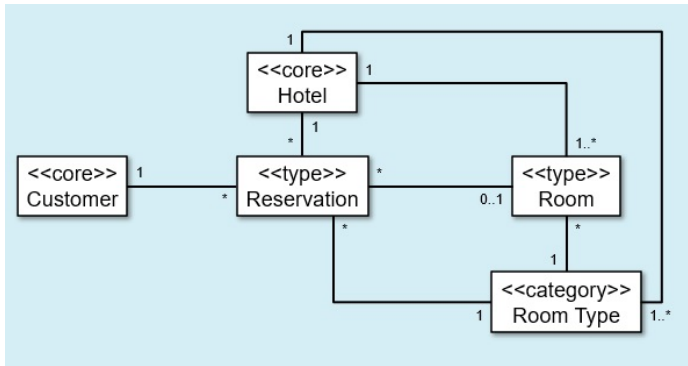


Fig. 3.11 Core and dependent business types

Core types typically have some unique business identifier. They do not have any mandatory associations (except to category types). Mandatory means there is a 1 multiplicity at the other end of the association to the core type. In contrast, there is usually a 1 to * or 1 to 1..* relation (one-many association) between core types and dependent types. For example, the order items of an e-commerce order, or rooms of a hotel.

To identify business interfaces we define one business component for each core type. This component manages instances of the core type and of all closely-related dependent types of the core type. We allocate one business interface per business component (this may be divided into smaller interfaces in a refinement, following the ISP). There may be ambiguous cases where it is not clear if a dependent entity belongs to one core entity or another. In general, if two entities need to evolve together, they should be in one component (by the SRP). In our example, it is not clear from the data model if *Reservation* belongs with *Customer* or *Hotel*. However, by looking at the use cases performed by a *Guest* actor (Figure 3.9), we see that they all involve *Reservation* and *Hotel*. In contrast, use cases to add, remove and edit *Customer* details are performed by an *Administrator* actor. Thus *Customer* and *Reservation* are involved in the use cases of different actors and should

not (by the SRP) be in the same component. Instead, *Reservation* belongs with *Hotel* in one business entity component (Figure 3.12).

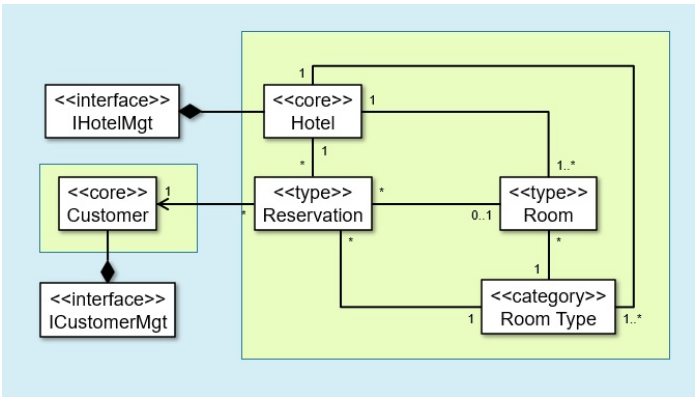


Fig. 3.12 Business entities and interfaces

Figure 3.13 shows the derived business tier architecture for the room booking system.

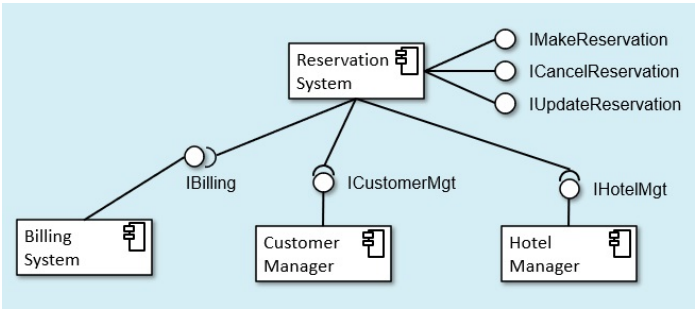


Fig. 3.13 Business tier architecture for room booking system

3.4 Analysis of clean architecture principles at the specification stage

The creation of system specification and architectural models enables the early detection of architectural issues and problems, such as violations of the clean architecture principles. Figure 3.14 shows one form of analysis which can be performed by AgileUML, using an example of a simple banking system: the

usage relations between use cases and the data of class diagram classes can be automatically identified and graphically presented (green arrows denote read access and red arrows denote write access).

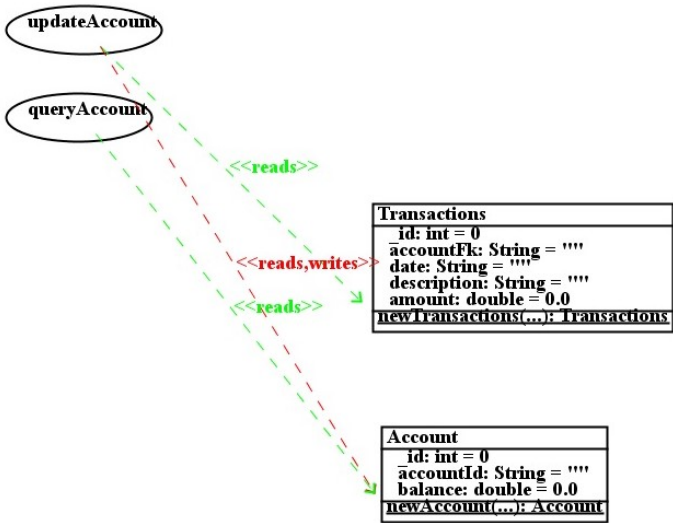


Fig. 3.14 Use case read and write dependencies to class diagram

This form of analysis can detect violations of SRP, because these occur when two different use cases with different actors both access the same class (Figure 3.15). In this example, the *Staff* actor performs the *updateAccount* use case, whilst the *Customer* actor performs the *queryAccount* use case.

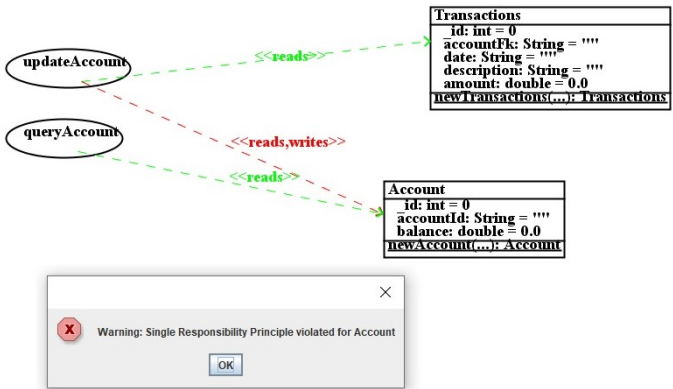


Fig. 3.15 SRP violation example

To detect violations of OCP, warnings would be issued when the code of any operation of a class is changed, if there is any call to the operation in the system.

To check LSP, informal verification can be carried out based on the superclass and subclass operation specifications, or formal verification can be performed using a translation of the specification to a formal mathematical representation such as the B notation.

For example, the following specification can be verified by checking that the subclass invariant Inv_{Sub} is true in the joint initial state $st = Set\{\}$ and $sq = Sequence\{\}$, and is preserved by joint execution of $Sup::add(x)$ and $Sub::add(x)$:

```
abstract class Sup {
    attribute st : Set(String);

    operation add(x : String ) : void
    pre: true
    post: st = st@pre->including(x);
}

class Sub extends Sup {
    invariant st = sq->asSet();

    attribute sq : Sequence(String);

    operation add(x : String) : void
    pre: true
    post: sq = sq@pre->append(x);
}
```

To check ISP, the quality analysis checks of AgileUML can be applied, these issue warnings if there are excessive numbers of operations in an interface (Figure 3.16).

ISP violations can also be detected by checking if any operations of supplier objects are called: if no operation of a referenced class S is called within client class C , then S can be removed as a supplier of C (Figure 3.17).

With regard to the dependency inversion principle, component diagrams can be used to generate class diagram elements which have the appropriate interfaces and relationships to support this. Figure 3.18 shows the generation of a class diagram (in the top pane) from a component diagram (lower pane).

Violation of ADP is detected by identifying calling cycles between operations of different classes, and identification of reference cycles between classes.

To check the dependency rule, the AgileUML tool identifies platform-specific and independent components, and checks if platform-independent components refer to platform-specific components (Figure 3.19). A class is considered platform-specific if it directly utilises files, processes, Excel func-

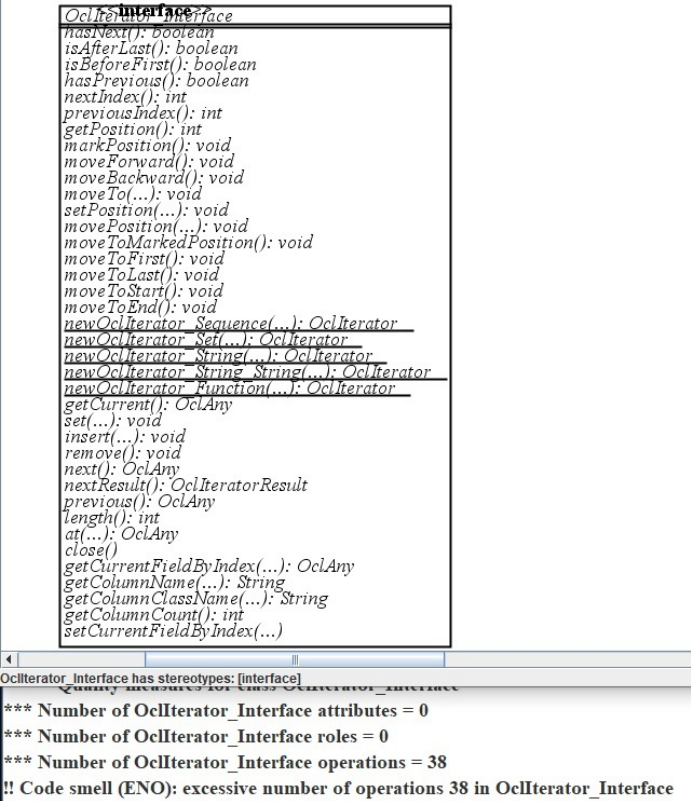


Fig. 3.16 ISP violation example (1)

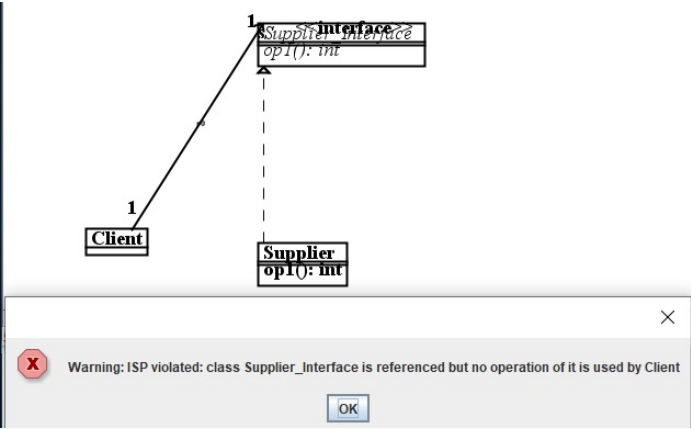


Fig. 3.17 ISP violation example (2)

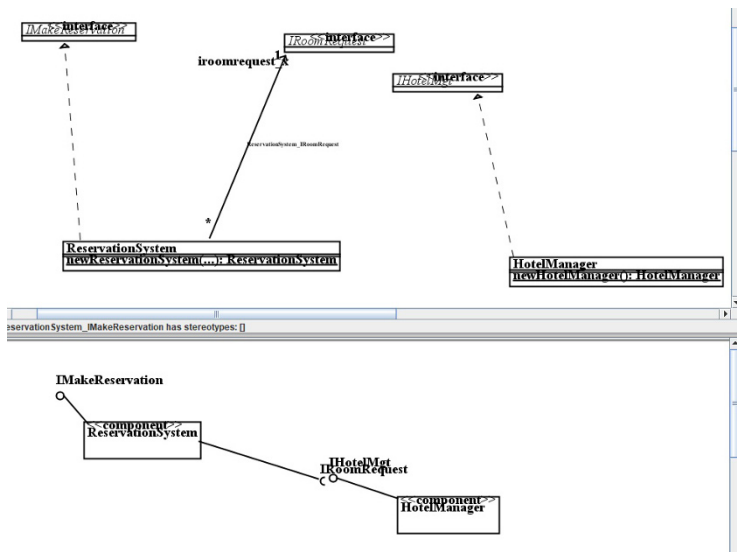


Fig. 3.18 Class diagram synthesis from component diagram

tions, databases, sockets or internet requests (OCL library classes *OclFile*, *OclProcess*, *Excel*, *OclDatasource* and *SQLStatement*).

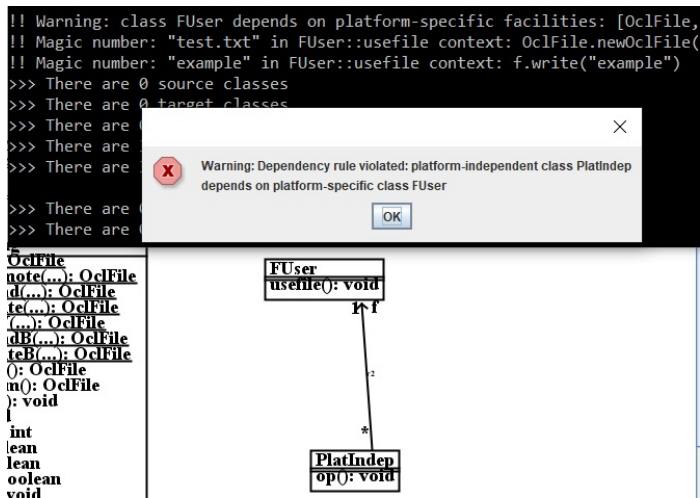


Fig. 3.19 Dependency rule violation example

3.5 Generation of architectures from specification models

Explicit specification models can also be used to generate complete executable code for applications, together with components, interfaces and connections conforming to particular architectural styles or principles. This can substantially reduce the manual effort required to write code and organise it into appropriate architectures. In particular, there may be some necessary duplication of code within an architecture:

- In the UI tier, validation checks on the input data for a use case are typically based on the *business rules* for the use case, which are also represented in the business tier. E.g., that someone applying for a bank account must be aged 16 or more. The checks could be performed by *validation bean* objects in the UI tier.
- Passing data between components in value objects – the value objects often have data based on that of the core business entities.

By using automated code generation from specification models, the necessary validation beans and value objects can be generated from the same model entity/operation definitions as the core business components. This ensures consistency between the different artefacts, and reduces coding effort.

The specification model serves as the ‘single source of truth’ for the application code, and should be maintained during system evolution.

As an example, consider the *Reservation* entity from the room booking system of Section 3.3.3 above. A detailed specification for this entity could be:

```
class Reservation
{ attribute reservationId : String;
  attribute startDate : String;
  attribute endDate : String;
  invariant startDate <= endDate;

  reference hotel : Hotel;
  reference customer : Customer;
  ....
}
```

The invariant states that the reservation interval must be non-empty. Dates would be represented in the format “YYYY:MM:DD”, so that string comparison corresponds to chronological order (e.g., “2023:12:31” < “2024:01:01”). This invariant should be checked for the input data of the *makeReservation* and *updateReservation* use cases, and this could be done by a validation bean *ValidateReservation*, called from a UI tier component such as a Controller (Figure 3.20). In addition, the invariant needs to be checked in the database

interface or Gateway for data storage of *Reservation* instances. Thus the semantic rule appears in widely-scattered places in the application. This would be difficult to manage manually, and it would be easy for mistakes to be made in coding and inconsistencies to be introduced. Instead, automated code generation could be used to produce the necessary code elements. If the constraint is changed in the model, then modified code can be regenerated.

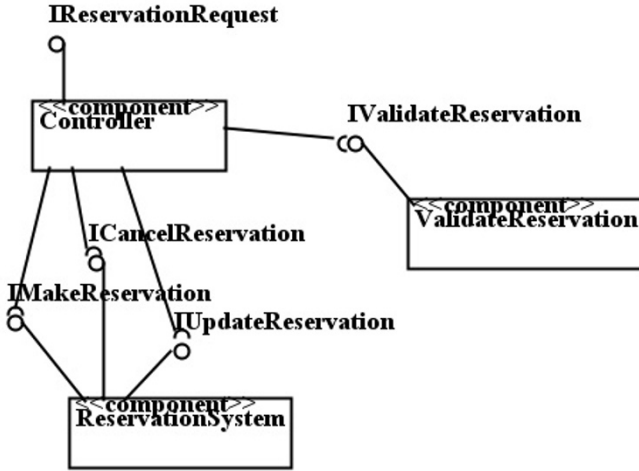


Fig. 3.20 Room booking UI tier/business tier

AgileUML can generate web-based applications, desktop applications and iOS and Android mobile apps from UML/OCL specifications [49]. Each of these implementations may involve the production of multiple inter-related components at different levels in the clean architecture hierarchy.

In the case of SwiftUI mobile app synthesis [51], we use the model-view-view model (MVVM) app architecture (Chapter 6), with the following specific components:

- Business entity classes *E.swift* are generated for each business entity *E*.
- The use case interactor component *ModelFacade.swift* implements all application use cases, and is generated from the specifications of these cases.
- SwiftUI views for each use case are generated from the use case specification.
- Data access objects/gateway components *DAO_E.swift* are generated for each business entity *E*.
- Value objects/data transfer objects *eVO.swift*, *ucVO.swift* are generated for each business entity and each use case.
- A database interface for local SQLite database storage is generated for the local persistent business entities.

- A cloud database interface for remote data storage is generated for remote persistent business entities.

It would be very difficult to manage this coding and to maintain consistency between all these components manually. Instead, any changes to the system requirements can be carried out by manual specification changes, which are then automatically propagated to consistent changes to the code of all components.

One problem with the use of MDE is the integration of auto-generated code with manually-written and maintained code. This integration can be facilitated by separating MDE-generated and manual code into separate components, which only interact via interfaces. For example, in an embedded system, the low-level device drivers could be manually coded and maintained, whilst the main control component would be auto-generated from specification models. When a change is needed to the strategy for code-generation, this change should be made to the MDE code generators, not by manual modification of the generated code.

Summary

In this chapter we have described the application of MDE techniques for architectural design. A process for the derivation of outline architectures from requirements models has been given, following the Cheesman and Daniels approach. We have also described how MDE can be used to enforce clean architecture principles by the automated generation of components, interfaces and connections conforming to the principles.

We have discussed architectural design within agile methods, including the practice of architectural refactoring.

Further Reading

- Cheesman and Daniels: UML Components. Addison Wesley 2001.
- Kevin Lano – Agile Model-Based Development Using UML-RSDS, CRC Press, 2016.
- Robert Martin: Clean Architecture, Prentice Hall, 2018.

Exercises

1

Draw a business concept model for a system in the banking domain, which enables staff of a bank to (i) enter new customer details in order to set up a new customer record for the customer; (ii) to create a new bank account for a customer. The steps of these use cases are shown in Figure 3.21.

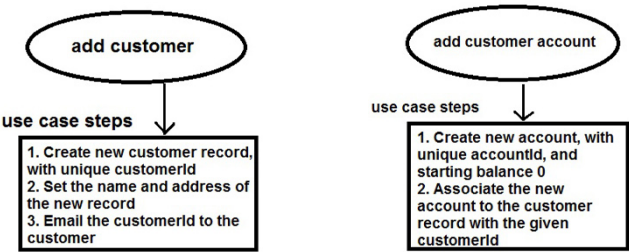


Fig. 3.21 Bank use case model

2

Use the use cases of Figure 3.21 and your answer to exercise 1 to define system interfaces for the banking system. Draw these using UML component notation.

3

Explain why the LSP fails to hold between *Square* and *Rectangle* in Figure 3.22.

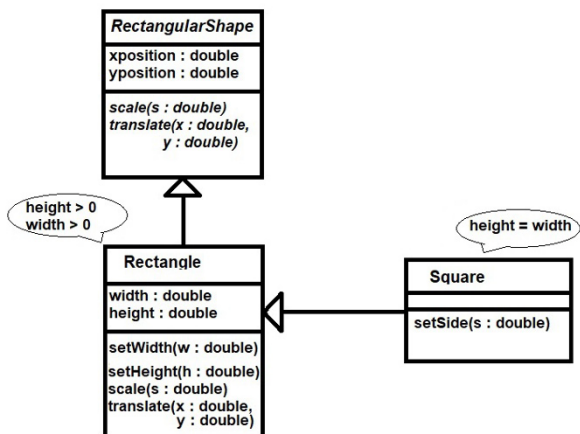


Fig. 3.22 Class diagram of shapes



Chapter 4

Compound Components and Complex Connectors

Abstract This chapter extends the idea of a software component by defining subsystems as compound components with an internal architectural structure. Such constructions need new kinds of connectors: *delegation connectors*, which link external provided/required interfaces of a subsystem to corresponding interfaces within the subsystem. We also consider different communication models for connectors: publish-subscribe and asynchronous connectors.

4.1 Composite components

Instead of organising systems as a large collection of basic components, it usually makes sense to structure complex architectures into subsystems, where each subsystem is a compound component containing several simpler components. This hierarchical organisation restricts potential component interaction, provides greater encapsulation, and enables the reuse of entire sections of an architecture, not just its basic components.

This approach also modularises the architecture and aids project management and maintenance of the system's code.

In all ADLs, there are two kinds of component:

1. Basic components – viewed from the outside as a black box, with no internal composition specified.
2. Composite or compound components – white box or grey box, with a visible internal structure but not necessarily with details of the inner components.

UML has several notations for compound components, we will use composite structure diagrams, which specify a composite component as a small architecture within a boundary. Figure 4.1 shows an example composite component for the room booking system.

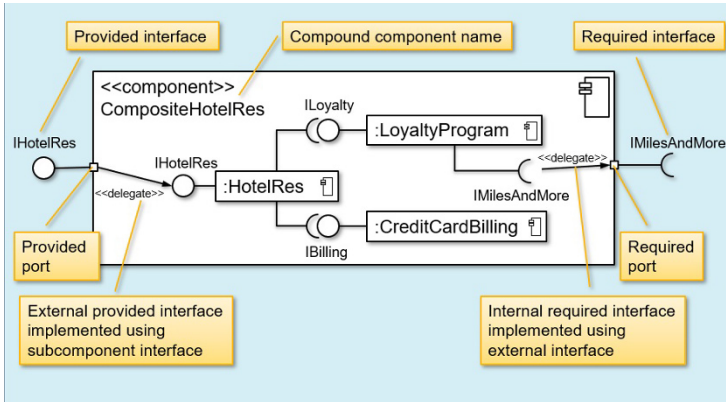


Fig. 4.1 Composite component example

In this diagram, termed a *composite structure diagram*, the internal components are represented as component instances, `: C`, because the composite component is regarded as an assembly of *instances* of the basic components, linked together by connectors.

Another new aspect here is the use of *delegation connectors*, marked with the `<<delegate>>` stereotype. These link external and internal interfaces of the same kind (provided to provided and required to required). A call to an external provided interface is delegated to the corresponding internal provided interface, and likewise a request on an internal required interface is delegated to a request on the corresponding external required interface. This is in contrast to assembly connectors, which link required interfaces to provided.

The boundary of the composite component is crossed via *ports*, which represent access points to/from the composite components. Delegate connectors actually join ports to internal interfaces. In general ports can have quite complex semantics, but here we consider them simply to be part of the composite component boundary.

In terms of Java code, composite components can be defined by packages which include all the inner components as classes, and these inner components are instantiated by an *assembly class* that implements the composite component (Figures 4.2, 4.3).

4.2 Advanced connectors

The `<<delegate>>` connector stereotype illustrates the use of UML stereotypes to specialise the kind of a connector within an architecture. Other specialisations which are marked by stereotypes include:

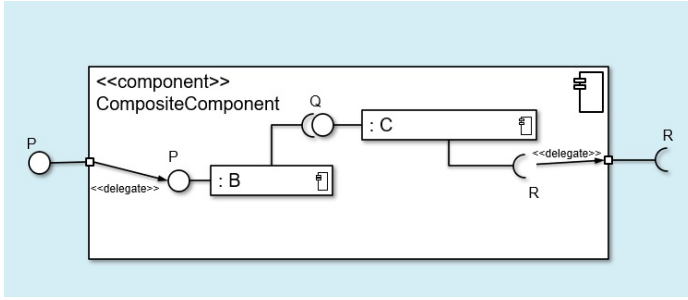


Fig. 4.2 Composite component example 2

```

interface P {
    public int op(int x);
}
interface R {
    public void opr(int x);
}
interface Q {
    public void opl();
}

class B implements P
{ Q qx;

    B(Q x)
    { this.qx = x; }

    public int op(int x)
    { qx.opl();
      return x;
    }
}

class C implements Q
{ R rx;

    C(R x)
    { this.rx = x; }

    public void opl()
    { rx.opr(1); } /* delegation */
}

class CompositeComponent implements P
{ R rx;
  private B bx;
  private C cx;

    public CompositeComponent(R x)
    { this.rx = x;
      this.cx = new C(rx);
      this.bx = new B(cx);
    } /* assembly of subcomponents */

    public int op(int x)
    { return bx.op(x); } /* delegation */
}
  
```

Fig. 4.3 Composite component example 2 (Java code)

- Read-only connectors, where the client only queries the state of a server: *<<query>>* or *<<readOnly>>* stereotypes.
- Remote procedure call (RPC) connectors, marked by *<<remote>>* or *<<RPC>>* stereotypes.

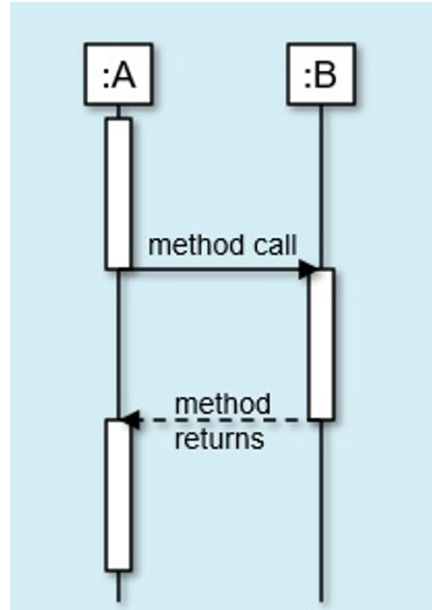
Some architectural styles use more advanced forms of connectors, particularly:

- Publish-Subscribe connectors.
- Asynchronous connectors.

The default assembly connection between components uses *synchronous* communication, as in Java method calls between objects. An object implementing a client component *A* communicates with an object of supplier component *B* by calling methods on *B*'s provided interface. With synchronous communication, the caller *A* instance must wait for invoked method calls to

finish before continuing its own thread of control. Figure 4.4 shows this as a UML sequence diagram, where objects are denoted as vertical ‘lifelines’ and communications are shown as arrows between objects. Time increases from the top to the bottom of the lifeline.

Fig. 4.4 Synchronous communication



This behaviour is not shown directly on UML component diagrams, but we can define stereotypes to specify connector semantics. We will use the `<<synchronous>>` stereotype in this case (Figure 4.5). This is the default stereotype and is assumed if no stereotype is explicitly written on a connector.

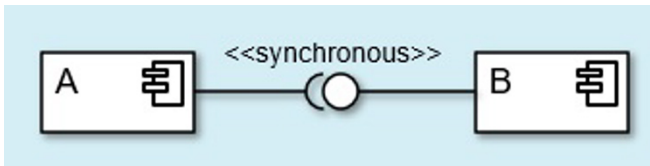


Fig. 4.5 Synchronous connection

In contrast, for *asynchronous* communication, a caller *A* instance can continue with its own computation after sending a call to a *B* instance, and does not wait for a response. This technique is useful for decoupling components, reducing communication delays, and increasing the responsiveness of *A*. But

it needs more complex management of the interaction – for example by enabling the called component to ‘callback’ to its caller *A* instance when the request result is available (Figure 4.6). We will use the `<<asynchronous>>` stereotype on connectors to mark that they involve asynchronous invocation (Figure 4.7).

Fig. 4.6 Asynchronous communication

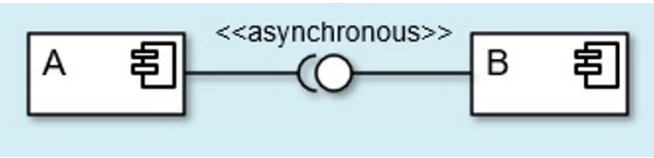
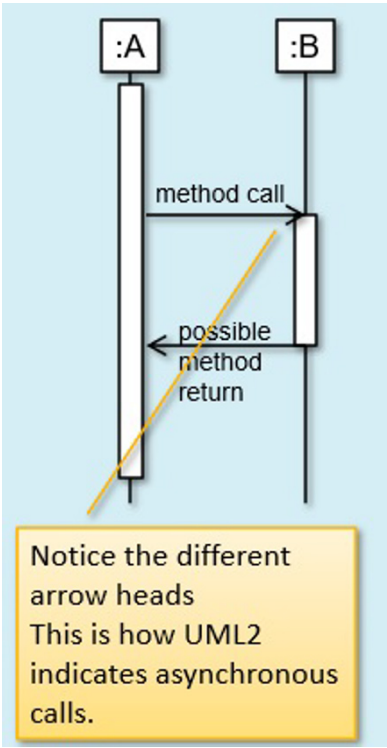


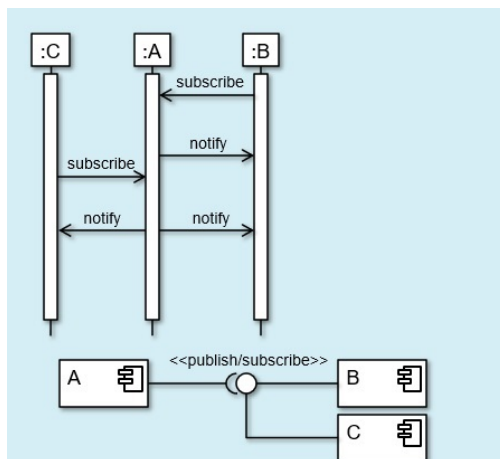
Fig. 4.7 Asynchronous connection

- A more sophisticated form of connection is a *publish/subscribe* connector:
- Components subscribe to events from a publisher, and can unsubscribe.
 - The publisher does not directly connect to individual subscribers.
 - Instead it broadcasts events to a list of all current subscribers.

Examples of this kind of connection occur in many UI frameworks, such as the event producer/event listener model of Java Swing. Event consumers (such as an object that responds to the selection of a menu item) register with the menu item to receive such selection events. The consumer interface can be very general (such as *ActionListener* in Java Swing), so that the producer does not depend on the concrete consumer class.

To represent this kind of connection we will use the $\ll publish/subscribe \gg$ stereotype (Figure 4.8).

Fig. 4.8 Publish-subscribe communication and connector



It is possible to have N publishers ($N \geq 1$) and M subscribers ($M \geq 1$) connected together by one publish-subscribe connector. This means that an event published by any publisher is broadcast to all subscribers.

Summary

In this chapter we have introduced the concept of composite components, and considered alternative forms of connector: delegation, asynchronous and publish-subscribe connectors.

Exercises

1

Express the architecture of Figure 4.1 as an outline Java package with classes for the components and for the complete subsystem.

2

Which form of connector would be the most appropriate for the following situations?

- A bank logging functionality, which logs all transactions in a file, for audit purposes.
- A credit card billing operation, which debits a given amount from a specified card.
- A stock exchange trade monitoring system, which monitors the trades conducted by each trader on the exchange in order to detect suspicious patterns of trades.

3

In the situation where a UI visual element (as event publisher) is connected to UI logic components via a publish-subscribe connection, should this connection be synchronous or asynchronous? Explain your answer.



Chapter 5

Architectural Styles

Abstract Architectural styles express recurring patterns of architectural design. They are the analogue at the architecture level of design patterns at the code design level, and help to guide the construction of architectures for new systems based on what architectures have been effective for similar types of system.

An architectural style consists of: (i) A vocabulary of architectural elements, such as ‘Filter’ or ‘Controller’; (ii) A set of configuration rules for connecting components in characteristic ways; (iii) A semantic interpretation explaining the role and meaning of elements within the architecture.

In this chapter we focus on the following domain-independent styles: Adapter; Pipes and filters; 2-tiered (Client-Server); N-tiered; Layered; Blackboard; Model-View-Controller (MVC).

5.1 Adapter architectural style

This style is based on the Adapter design pattern from the original patterns collection of [31]. It is used when a client requires an interface $I1$, but there is only available a supplier with a mismatching provided interface $I2$. To bridge the gap between these components, an intermediary component is introduced, which implements $I1$ using the $I2$ operations (Figure 5.1).

This situation arises when some external library is used, whose API does not exactly match the required services of our application. The external component cannot be changed, so one option is to introduce an adapter. Similarly, if a legacy code application needs to be reused in a new application, the legacy system can be ‘wrapped’ by an adapter which translates the new application requests into calls to the legacy API.

In the context of the clean architecture, if dependency inversion needs to be applied where the client is a business-tier component such as an interactor, and the supplier is an external component at a lower level (such as

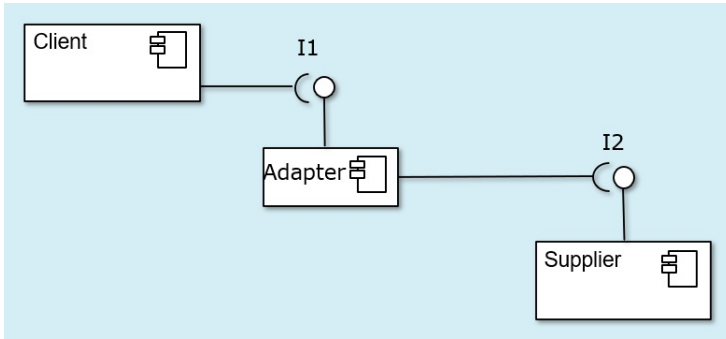


Fig. 5.1 Adapter architectural style

a cloud datastore), then it is not possible to define a dependency from the supplier to the client, and instead an intermediary adapter is needed, in the role of a gateway (Figure 5.2). Dependency inversion operates between the interactor and gateway. The Data Access Object (DAO) pattern for mobile and enterprise systems is an example of this situation (Chapters 6, 8).

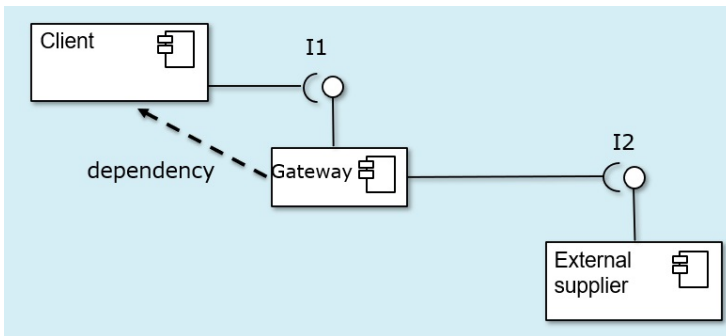


Fig. 5.2 Gateway Adapter

5.2 Pipes and filters architectural style

This style involves one kind of component: Filter. This represents a component which has a single provided interface (e.g., *IFilter*, or *ProcessRequest*) and this is also the single required interface. Filters receive requests and decide whether to forward them to the next component in the chain, to reject them, or to perform some local processing on the requests (Figure 5.3). This style maintains the semantic invariant that all requests emerg-

ing from the output end of the chain must satisfy all forwarding conditions F_1 and ... and F_n of the filters within the chain. In addition, the order of output requests is normally the same as the order of input requests, i.e., the output sequence is a subsequence of the input sequence.

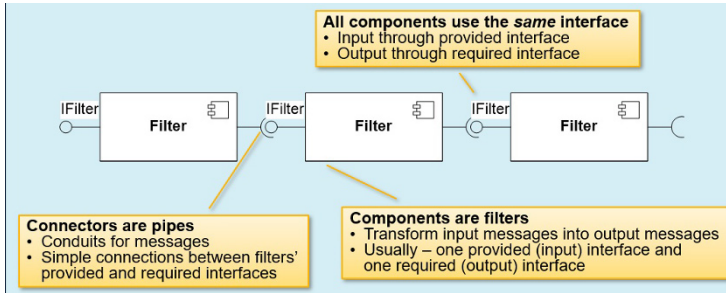


Fig. 5.3 Pipes and filters architectural style

The advantages of this style are that it is simple, and the system behaviour is the sum of component behaviour. It also facilitates the adding, replacement, and reuse of filters in the chain. Notice that the semantic invariant is independent of the order of the filters, so a given chain may be reordered as necessary to achieve QoS properties such as efficiency, whilst still preserving the chain semantics. In addition to linear chains of filters, it is also possible to use the style to define more general directed acyclic networks of filters, for example, a filter may produce two separate sequences of output requests from a given input sequence, formed from the requests which (i) pass or (ii) fail the filter condition. These disjoint outputs are then handled by separate target filters or subsystems. A filter could alternatively receive requests from multiple input streams and produce a single output stream from the combined inputs.

Disadvantages of the Pipes and filters style include that the lowest common denominator of data formats needs to be used in the filter interface, i.e., very general types such as Object or String need to be used for the parameter types of interface operations, because the same interface is used throughout the architecture.

Examples of Pipes and filters include UNIX/Linux shells (piping data between processes), telecommunications signal processing pipelines, and natural language processing (text processing pipelines). It also has wide application in enterprise systems and business systems.

5.2.1 Pipes and filters example: Order processing system

This is an online e-commerce system, where new orders arrive from clients in the form of request messages. These messages are encrypted to prevent third parties from spying on a customer’s order, messages also contain authentication information (a digital certificate) to ensure orders are placed only by trusted customers. Duplicate messages could be sent and these must be eliminated (E.g., warnings on shopping sites to click the ‘Order Now’ button only once).

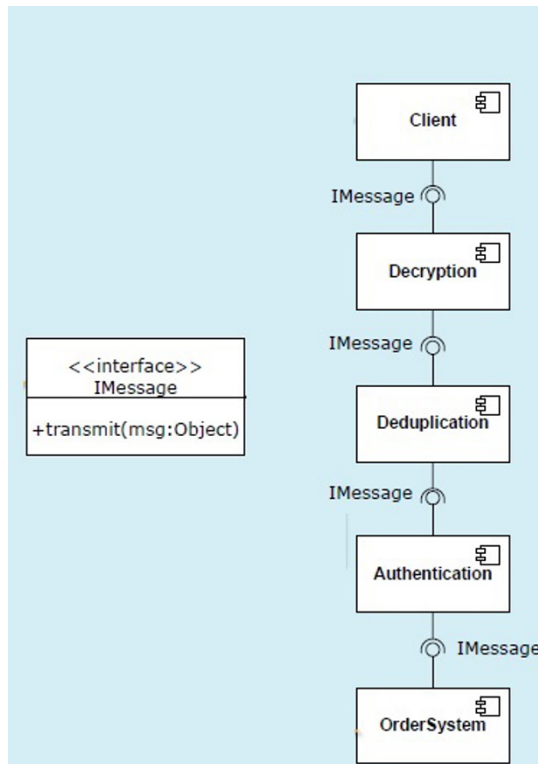


Fig. 5.4 Order system filter chain

Pipes and filters can therefore be used on the server side to transform the stream of possibly duplicated, encrypted messages containing extra authentication data into a stream of unique, simple plain-text order messages (Figure 5.4).

5.3 Two-tiered (Client/Server) architectural style

This is one of the most common styles in distributed systems. It is characterised by division of components into either *clients* or *servers*: Clients request services from servers, and servers provide services to clients, but servers do not require services from clients. In general there are a larger number of clients than servers. Either or both clients/servers may be composite components (Figure 5.5).

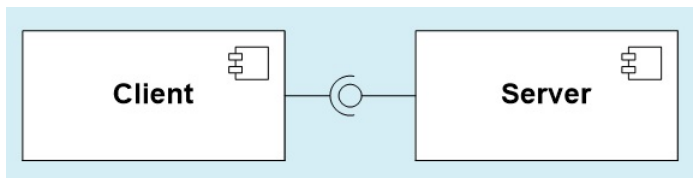


Fig. 5.5 Two-tier architecture system organisation

Servers do not know the number of or identities of their clients, whilst clients know the server's identity. Clients and servers are often distributed, with remote procedure call (RPC) used to implement connections. The stereotype `«remote»` can be used on the connectors to make this situation explicit.

This architectural style is useful whenever there is a need to separate application logic into two tiers, or where a local application needs to communicate with a remote application, possibly owned by a different organisation.

For example (Figure 5.6): Google offers a search webservice, and a range of clients use the webservice – web clients could use the search service as part of their web pages, and a .NET-based business-2-customer application could use the webservice programmatically to search for best buys of products.

5.4 N-tiered architectural style

This style generalises the client/server style by allowing servers to also be clients of other servers (Figure 5.7). Application logic is divided between the chain of components/layers. As with 2-tier, clients and servers may be composite components, and servers provide services to their clients, but do not require services from them, clients require services of the server. No cycles are permitted in the dependency graph (ADP).

In an N-tiered architecture, servers do not know the number or identities of their clients, whilst clients know the server's identity. Components may be distributed (with RPC used to implement connections). This style is useful to separate application logic into separate software packages at different levels

5.4.1 Example N-tier architectures: enterprise and mobile systems

Enterprise Information Systems (EIS) are typically organised using N-tier architectures with up to 5 tiers (Figure 5.8):

- *Client tier*: UI components for direct interaction with clients.
- *Presentation tier*: UI logic components such as presenters, controllers and helpers.
- *Business tier*: business functions and logical representations of business data. Usually this tier is internally divided between use case interactors and business entity components.
- *Integration tier*: components (gateways) mediating between the business tier and data sources.
- *Resource tier*: databases, webservices, etc.

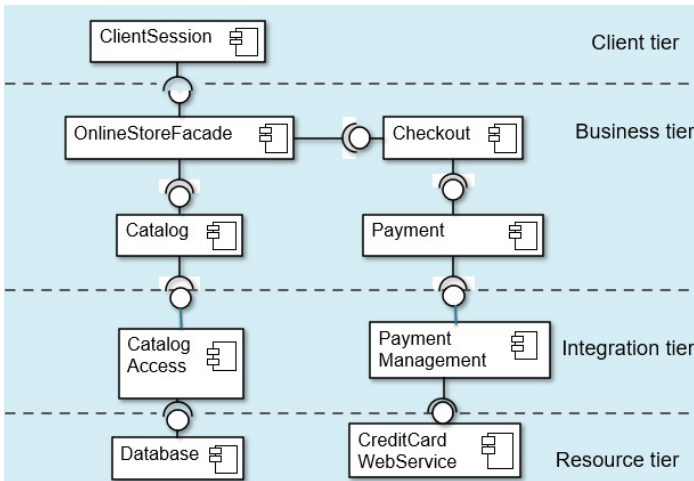


Fig. 5.8 Four-tier EIS architecture

N-tier architectures can also be used for mobile apps. Figure 5.9 shows the architecture of a simple m-health app which records some information about a person and calculates some measures such as a body-mass-index (BMI). Mobile app architectures are discussed in Chapter 6.

5.4.2 N-tiered style: advantages and disadvantages

The N-tiered architectural style has the advantages of enforcing separation of concerns between different aspects of the system, permitting a divide and

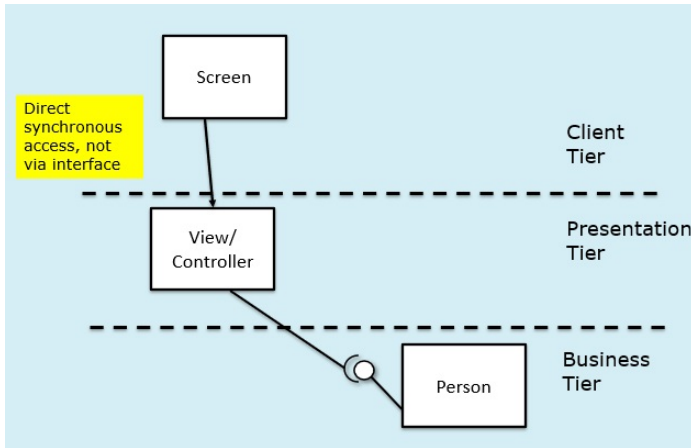


Fig. 5.9 Mobile architecture example (3-tier)

conquer approach to solving large scale problems: different teams can work on developing and maintaining the UI, the business tier, the data resources, etc. However the separation into multiple tiers/layers also has disadvantages with regard to performance (especially with distributed components): to address this, data needs to be transferred in logical groups (value objects), and caching can be introduced, or other techniques used to limit network usage. We discuss these issues in Chapters 7 and 8.

5.5 Layered architectural style

This style also adopts a hierarchical system organisation, with a multi-level client/server structure. Components are classified by layers, and each layer exports a provided interface to be used by higher layers. The key idea is that each layer represents a different abstraction level, whereby the functionality of one layer is implemented by using the API of a layer at a lower level of abstraction (i.e., more hardware-oriented).

There are stricter constraints for this style, compared to the N-tier style:

- There is a 1-1 relation between each layer – each layer contains a single component.
- Each layer communicates only with its immediate neighbour above or below it (no jumping over layers).

This architecture applies where a system needs to be organised as a technology stack, with layers at different levels of abstraction (higher levels more user-oriented, lower levels more machine/device oriented). Connectors are based on the protocols of the layers: the APIs of the layers may be pre-

defined (e.g., the HTTP API) and there would then be no choice about the provided interfaces.

Examples include operating systems, where layers correspond to virtual machines running on other machines (virtual or real). Here the lower layers are fully opaque, and a user of a virtual machine has no access to the layers below it. Network communications and embedded systems are also examples of this style of organisation. An embedded system is typically divided into three layers for software, firmware and hardware (Figure 5.10).

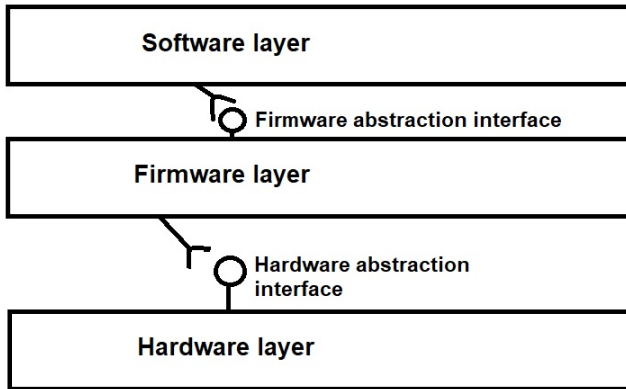


Fig. 5.10 Embedded system architecture

Advantages of layered architectures include:

- Increased abstraction from low-level coding and specific devices.
- Improved maintainability.
- A clear role is defined for each layer.
- Improved reuse and portability.

Disadvantages include that some functionality may be distributed around different layers, and in some cases layer bridging may be required – a layer might need to communicate directly with a non-neighbour, in order to improve performance. Determining the correct abstraction level for layers may need significant analysis, if the layers are not already fixed by existing protocols/technologies.

5.6 Blackboard architectural style

This is a significantly more complex style than those we've seen so far, in particular because it involves dependencies in two directions (violating the ADP).

In this style there are two kinds of components: (1) the Blackboard – acting as a central data structure or repository; (2) Clients/Views – which update and receive information from the blackboard. System control is driven by the blackboard state, which serves as the ‘single point of truth’ for the logical system data.

The constraints of this style are that there is one blackboard component and one or more clients. The clients call the blackboard via assembly connectors, and the clients receive notifications from the blackboard via a publish/subscribe connector. In order to avoid the danger of infinite recursion (endless sequences of responses and updates caused by a single initiating event), the client operations that handle notifications from the blackboard should not themselves trigger updates to the blackboard.

Blackboard is similar in structure to the Observer design pattern [31], with the blackboard playing the role of the Observable/Subject, and clients in the role of Observers/Views (Figure 5.11). But in Observer, the Observers do not update the Subject, whilst in the Blackboard style the blackboard can be updated by clients (although some may have read-only access), and the updates could be synchronous or asynchronous (Figure 5.12). As with the Subject in Observer, the blackboard can notify all clients when it is updated, so that these clients will all have a consistent view of the underlying subject (blackboard) state. The blackboard is acting as a ‘single source of truth’ about the logical data of the system.

Each client state will typically be some (view-specific) function of the blackboard state, in the same manner as with the Observer pattern views/subject. However, the blackboard itself may be a function of the states of the clients, such as a composition of their states.

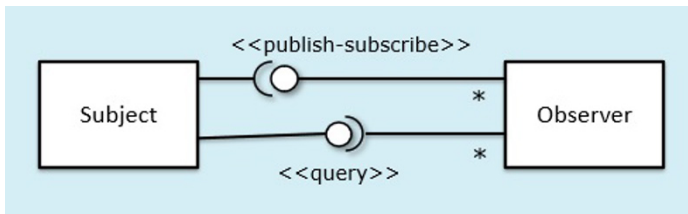


Fig. 5.11 Observer pattern

Clients encapsulate individual pieces of application functionality, such as a particular screen/editable panel in a programming IDE. The clients do not directly interact with each other, but only through their effects on the blackboard. The clients make changes to the blackboard, and these changes lead consequentially to further changes to one or more clients.

The blackboard is an active component, a state change to the blackboard triggers a selection of clients to notify – those clients whose states need to change as a result of the blackboard state change – and these clients then ex-

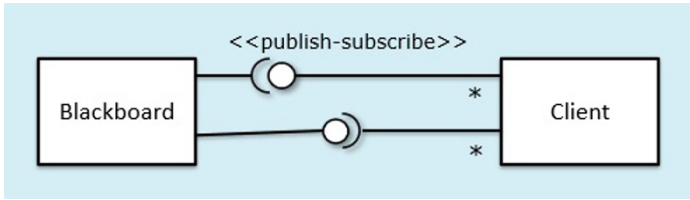


Fig. 5.12 Blackboard architecture

ecute locally their own individual responses to the notification. For example, an edit to an application’s code in one IDE window could lead to a change in the IDE repository state, and hence to corresponding changes to other views of the application in other IDE windows.

The publish/subscribe style of connection is used for clients to be informed of blackboard changes. Separate (a)synchronous connectors from the clients to the blackboard are used for changing the blackboard state. Because the blackboard component is at a higher level of platform independence compared to the views, the views should depend on the blackboard, but not vice-versa. Thus dependency inversion should be used for the publish/subscribe connection between these components.

Some typical uses of the Blackboard style are IDEs such as Visual Studio or Eclipse, and e-commerce systems where stateless client components are driven by stateful data components.

5.6.1 Blackboard example: IDE

This example is an integrated software development environment (IDE), where a stateful repository maintains the definitive version of the software code being manipulated (Figure 5.13).

Three components present views of the code, such as a UML class diagram, a class hierarchy view, and the text editor view. A change made in any one view can change the code in the repository. For example, removing an inheritance between two classes in the class hierarchy view will update the repository data about these classes and hence cause updates to the other views which show them.

5.6.2 Blackboard example: Holiday reservation system

This system stores all holiday reservation data in a database, there are three components for different aspects of holiday reservation logic: booking hotels, flights and car hire. Each component could connect to the database separately

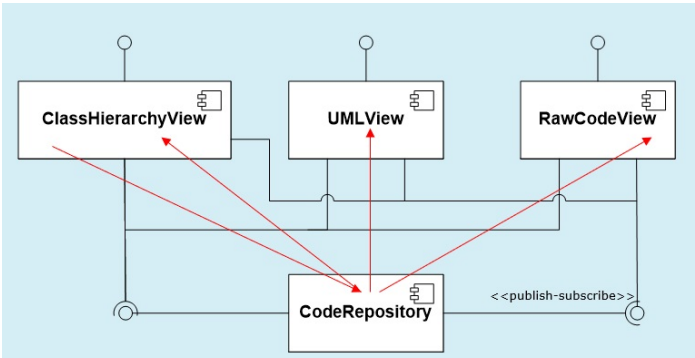


Fig. 5.13 Blackboard architecture for IDE

– however this would cause performance and integrity problems because the three aspects are inter-dependent: a change in one aspect (such as the dates of flights) affects another aspect (the valid dates to book a hotel). The solution is to use an intermediate single blackboard component, which handles the connection to the database, and provides a centralised ‘single source of truth’, combining the information about each user’s reservations (Figure 5.14).

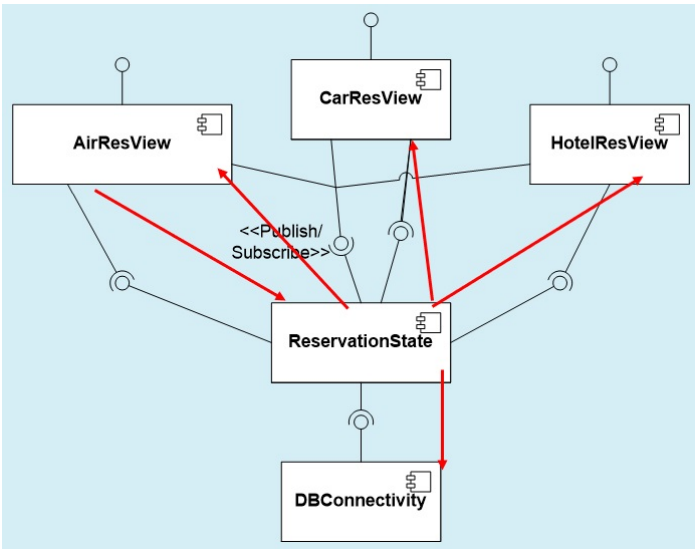


Fig. 5.14 Blackboard architecture for travel booking system

The three reservation logic components can change the blackboard, and also subscribe to be notified of important changes to the blackboard data. Thus if the flight dates are changed, automatically the hotel and car bookings

are marked as needing to be updated. The blackboard can be configured to only persist valid reservation data (with all required aspects being consistent) to the database.

5.6.3 Blackboard example: Realtime cloud datastore

In this type of system (e.g., mobile apps using the Firebase datastore¹), a stateful repository on the cloud holds data accessed and modified by multiple clients, such as apps running on different mobile devices (Figure 5.15). Each client should have the same view of data as others (subject to delays in change propagation and connectivity issues).

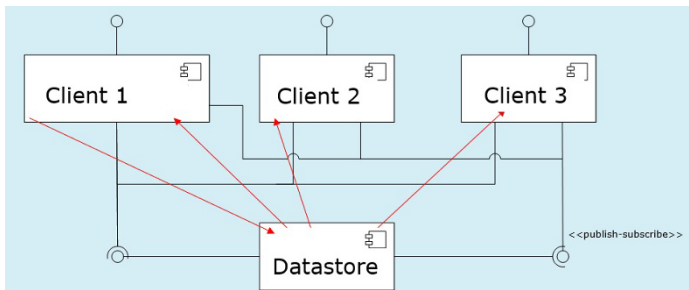


Fig. 5.15 Blackboard architecture for cloud datastore

This also enables federated identity and sharing of app data with social media sites and other app users.

5.6.4 Advantages and disadvantages of the blackboard style

Advantages of the style are:

- Clients are relatively independent of each other and can be changed independently.
- The data store is independent of individual clients, instead it publishes notifications on a general required interface, which individual clients implement via listener provided interfaces.
- Scalable (new clients can be added easily).
- Data integration/consistency is achieved through the blackboard.

Disadvantages include:

¹ firebase.google.com

- The interaction is complex and involves logical constraints relating clients and blackboard – this means that the client and blackboard parts should be considered as a composite component and developed by the same team/developer.
- It is not appropriate if clients need to make complex updates to the blackboard, i.e., to execute an algorithm on the blackboard, instead of making simple updates.
- Efficiency issues can arise due to the extensive communication, especially if components are distributed.

A simplified form of Blackboard can be used if it is only necessary for one client at a time to be up-to-date with the blackboard state, e.g., in a situation where only one UI screen can be visible at a time. In this ‘Shared data with manual refresh’ style, there are no updates from the blackboard to the views, instead views write to the blackboard, and read its state in order to present this only in the case when they are initialised or redisplayed (Figure 5.16).

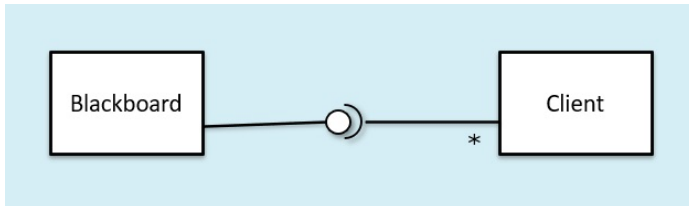


Fig. 5.16 Simplified Blackboard architecture

5.7 Model-View-Controller (MVC) architectural style

This is an elaboration of the blackboard style, specifically designed to enable complex interactions between the clients and the repository, termed the *Model* in MVC.

MVC involves three different kinds of component (Figure 5.17):

1. Views – responsible for displaying/receiving information to/from users.
2. Controller – responsible for managing interaction with users and implementing the intent of user actions by updating the model.
3. Model – responsible for maintaining a consistent single internal representation of application knowledge.

The model representation typically consists of business entity data. The model does not depend on any individual view or on the controller.

It propagates changes in its state to the views via a publish/subscribe connector.

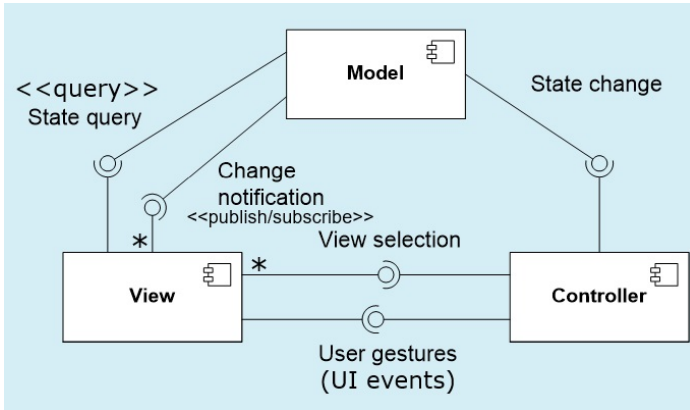


Fig. 5.17 MVC architecture

The semantics of MVC involves one model, one controller and multiple views, with the model encapsulating application state and core business functionality. It acts as an Observable in relation to the views as Observers, and notifies views of changes in its state. Each view state is a view-specific function of the model state. Views register for updates from the model, maintain a consistent presentation of model state, and communicate user gestures (such as button clicks) to the controller. The controller maps user actions to model updates, and also manages the view selection, i.e., deciding what should be the current view in focus.

In contrast to Blackboard:

- The Controller can implement coarse-grain operations in terms of fine-grain Model (blackboard) operations.
- In Blackboard, clients directly use the fine-grained blackboard operations.
- The Controller in MVC has the responsibility to maintain integrity constraints for the Model data, in Blackboard this would need to be done within the blackboard or the clients.
- Dynamic changes of views are supported by means of the controller. Blackboard has a static set of views.

As with Blackboard, the view operations which handle updates from the Model should not themselves trigger calls to the Controller.

MVC generalises Observer because the Views present up-to-date views of the model state: the model notifies views when its state changes. The same interaction is used as in the Observer pattern: Views have the Observer role, and the Model has the Subject role. Views do not directly update the model. As with Blackboard, to avoid dependency of the business tier component (Blackboard or Model) upon the presentation tier Views, dependency inversion can be used between these components.

5.7.1 Model-View-Controller advantages and disadvantages

Advantages of MVC are that the views are relatively independent of each other, and that the model is independent of individual clients. It is scalable in the sense that new clients can be added easily. Data integration is achieved through the Model, and complex updates are handled by the Controller.

Disadvantages are that there are semantic dependences between all components, so that their interaction needs careful management. As with the Blackboard style, there is a significant amount of communication, so there can be performance issues especially if the system is distributed.

5.8 Heterogeneous styles

Heterogeneous styles combine different architectural styles, this is necessary in large systems which involve a combination of functionalities and aspects. The challenge in such a case is to ensure that the individual styles are consistently combined, and do not conflict with each other.

One approach is to apply styles in a compositional manner, with the main system decomposed into subsystems, and one style applied per subsystem (i.e., in a composite component). For example, an N-tiered style could be used to structure an overall architecture, and a Blackboard style used for a specific kind of interaction, usually involving the UI/Presentation tier (Figure 5.18).

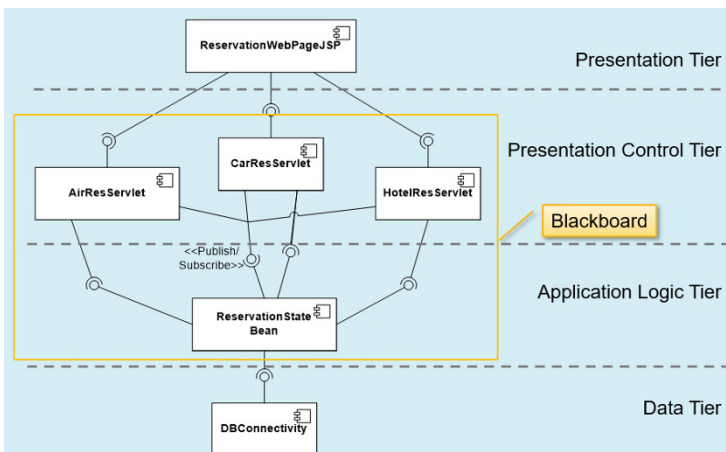


Fig. 5.18 Integrating different styles

Summary

In this chapter we have described seven important architectural styles: Adapter, Pipes and filters, Client-server, N-tier, Layered, Blackboard and MVC. Examples of the styles have been given, and their advantages and disadvantages explained.

Exercises

1

Design and draw the UML architecture diagram of the following system, using a suitable architectural style. Justify the choice of this style.

The system is an HTML editor which has three views of an HTML document:

- A Word-style view and editor (visual editor).
- An HTML source view and editor with colouring of text used to highlight the scope of HTML elements (code editor).
- A read-only preview of the document in Firefox.

A user should be able to work in either of the editors and see their changes propagated to the other two views.

2

Explain what architectural style you would use for the following system, and why:

The system is an IDE to support program development. It has a graphical interface with two main views: (i) an editable view where the programmer writes code; (ii) a help/advice view which provides information about the code constructs and recommendations on good programming techniques. This is not editable but is updated based on changes to (i). The system also highlights or underlines code in (i) to identify problematic code segments.

3

Draw a UML architecture diagram for your chosen architecture for the system of question 2. Interfaces should be named, but interface specifications are not required.

4

Identify appropriate architectural styles for the following systems, and explain your choice of styles:

- A real-time system simulator, which simulates the execution of embedded system components.
- A cloud datastore, providing data storage facilities to registered users. Changes to one dataset are notified to all users registered to read it.

5

Draw a 5-tier EIS architecture for an estate agency system which allows users to register with their details and preferences, to login, once registered, and to search for matching properties and to logout, once they are logged in.

6

Explain how the *layered* architecture differs from the *N-tier* architecture.

Which would be most appropriate for an online bookstore with separate business components to manage stock, to manage purchases/delivery and to manage customers?

7

Express the semantic invariants of a Pipes and filters network which has a tree structure with multiple output branches produced from a single input stream of requests.

Express the semantic invariants of a network consisting of a filter which produces a single output stream from two or more input streams.



Chapter 6

Mobile Application Architectures

Abstract This chapter describes the concepts of mobile application architectures, including the specific features of mobile platforms which lead to specialised forms of architecture. We consider Android and iOS app architectures in detail, and specific architectural patterns for the mobile domain.

6.1 Mobile applications

Mobile applications are ‘apps’ that are intended to execute on mobile phones and tablets. This includes a wide range of systems across business, entertainment, social and health domains. The special characteristics of mobile devices affect all aspects of mobile apps:

- Mobile devices typically have restricted interfaces (display size), processing power, memory and connectivity.
- Apps can use a wide range of data inputs, including on-device sensors such as cameras, accelerometers, etc, and utilise a wide range of interaction modes with users.
- Usability, security, and responsiveness are important quality-of-service (QoS) properties for mobile apps.
- The possibly intermittent nature of connectivity means that apps must manage offline/online status.

Several mobile device platforms exist, of which Android and iOS have become the predominant platforms. A mobile app platform acts as an *application server* or higher-level operating system to support and manage software components and device resources on behalf of individual apps.

6.1.1 Similarities between mobile applications and enterprise systems (EIS)

There are several similarities between mobile apps, web applications and enterprise systems, and hence a number of architectural styles/patterns can be used for all of these kinds of system.

The overall architectural design for these systems is usually some form of N-tier architecture, and some version of the MVC pattern is commonly used for the presentation tier and for presentation tier/business tier interaction. Each kind of system utilises data resources, which can be local resources or remote resources. In a mobile app these resources are termed ‘assets’ or ‘content providers’. A tier (termed the *integration tier* in an N-tier architecture) manages the interaction of the business tier with resources/content providers, using *Data Access Object* components as adapters.

Therefore, patterns used in both mobile and business/enterprise systems include:

- N-tier.
- MVC and variations (Section 6.2.1).
- Value Object/Data Transfer Object (Section 6.2.2).
- Session Facade/Model Facade (Section 6.2.3).
- Service Activator (Section 6.2.4).
- Data Access Object (Section 6.2.5).

6.1.2 Distinctive aspects of mobile applications

Because of the resource and other constraints of mobile devices, we need to use the following principles for the architectural design of mobile apps:

- Components should be relatively small and lightweight – with low resource utilisation.
- UI components should be relatively simple due to UI restrictions – with only a few controls/UI elements per screen.
- User interactions should be relatively short (e.g., no more than five steps to complete a task).
- Only linear interactions should be used, without requiring multi-tasking or switching from screen to screen – because usually only one screen will be visible at a given time.
- Any heavyweight computation (such as training a machine learning model) should be done off-device on a remote server.

Other factors are that the mobile platforms themselves evolve and change quite frequently, and new programming languages have been introduced specifically for this domain (e.g., Kotlin, Swift), which are also evolving.

There are regular changes to platform services and libraries, including third-party software and services from Google, Amazon, etc. Thus the mobile domain is characterised by rapid changes to technology. In contrast, business-oriented .NET and Java EE platforms for EIS have been relatively stable over the last 20 years. Therefore, app architectural designs should be organised to be effective regardless of such changes, and to be as independent, as far as possible, of specific technologies and versions.

UI behaviour and navigation is a critical aspect of many apps, affecting the ‘user experience’ or UX which is a key factor determining if an app is successful or not.

Two main forms of navigation are:

1. *Lateral* navigation between alternative options at the same level (e.g., tab options in a tabbed screen, Figure 6.1).
2. *Forward/reverse* navigation between different steps in a sequential process (e.g., a login/signup process, Figure 6.2).

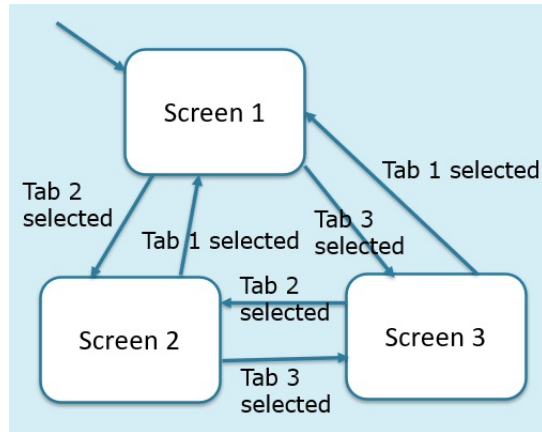
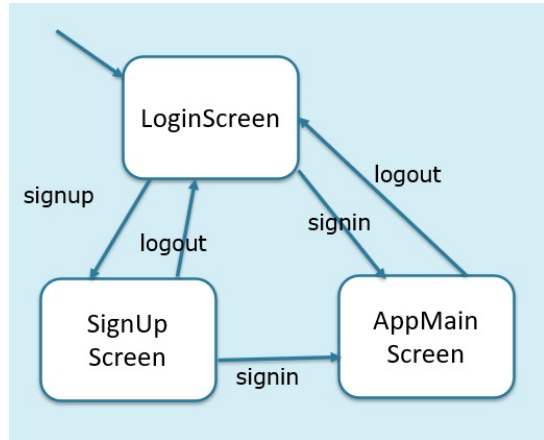


Fig. 6.1 Lateral navigation, represented as a state machine

Lateral navigation is often provided by built-in UI elements (so there is no need for additional programming by the app writer). Forward/reverse navigation may need specific coding (e.g., by defining *segues* between screens in iOS, or invocation of *activities* via *intents* in Android).

Other alternative navigation options are *context/pop-up menus* which appear when prompted by a click/touch on the current screen and show options specifically relevant to that screen/screen location, possibly involving entering a nested/subordinate UI interaction (e.g., selecting an attachment file for a text message), and *hamburger menus*, with an icon of three horizontal lines which expands into a menu when selected. These techniques however have the disadvantage that the menu options are normally hidden and users therefore require some familiarity with the app in order to use these.

Fig. 6.2 Forward/reverse navigation



6.1.3 Mobile app example

The following example, app4, is a simple mobile health (m-health) app, which enables the user to calculate their body-mass index (BMI) and to estimate the calories consumed by different exercise sessions.

The app specification in UML consists of two enumerated types *Gender*¹ and *Exercise*, and two use cases, *computeBMI* and *calorieCount*:

```

package app4 {
    enumeration Gender {
        literal male;
        literal female;
    }

    enumeration Exercise {
        literal walking;
        literal jogging;
        literal running;
        literal swimming;
        literal weights;
    }

    usecase computeBMI : double {
        parameter height : double;
        parameter weight : double;

        precondition height > 0;
        precondition weight > 0;

        activity:
        result := weight / ( height * height ) ;
    }
}

```

¹ This type is introduced for use in a more sophisticated subsequent version of the app.

```

usecase calorieCount : double {
    extendedBy computeBMI;

    parameter gender : Gender;
    parameter exercise : Exercise;
    parameter time : double;

    activity:
        var factor : double ;
        factor := 1 ;
        if exercise = walking
        then factor := 100
        else (
            if exercise = running
            then factor := 300
            else (
                if exercise = jogging
                then factor := 200
                else factor := 250
            )
        ) ;
        result := factor * ( time / 60 ) ;
    }
}

```

The use cases define the main functionalities of the app. Each use case has input parameters, the values of these will be input on an app screen for the use case, and a returned value, which will be displayed on the use case screen as the result of executing the use case. The *precondition* clauses describe validity conditions on input values. The use case actions are defined by the *activity* : clause, the actions are triggered by a button press or other selection of the functionality on the use case screen. Normally, each use case would be used via a separate screen, but in this case, because the use cases are closely related, they can both be presented on a single screen. The *extendedBy* clause expresses that the *computeBMI* use case is a use case extension of *calorieCount*, i.e., it is intended to be used as an optional addition to the *calorieCount* use case, and therefore that they should share a screen. Figure 6.3 shows the specification as a UML class diagram. Figure 6.4 shows the app being developed using Android Studio, the IDE for Android apps.

Figure 6.5 shows the app being developed for SwiftUI in the Xcode IDE for iOS apps.

In the supplementary materials on the book website² we provide the complete code of app4 in Android and iOS versions.

² agilemde.co.uk/arch

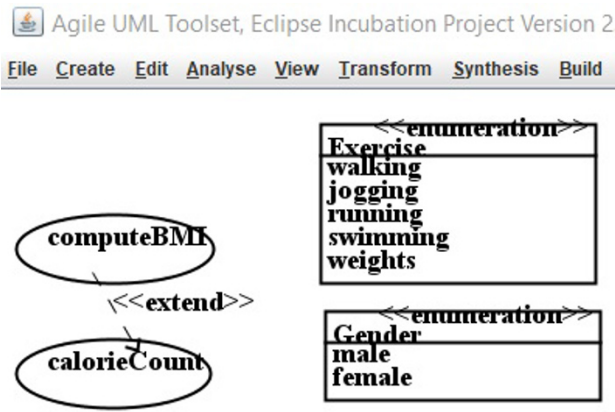


Fig. 6.3 M-health app specification

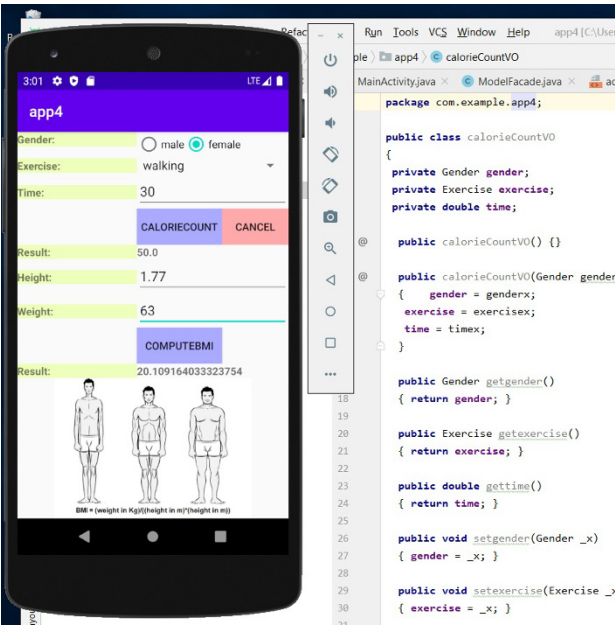


Fig. 6.4 Android version of m-health app

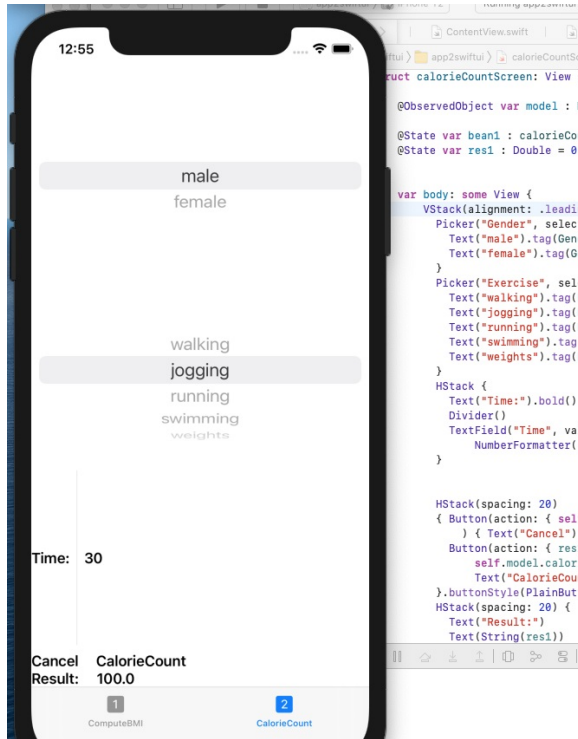


Fig. 6.5 SwiftUI version of m-health app

6.2 Mobile architecture patterns

The following mobile architecture patterns can be applied for both Android and iOS, but with some minor differences across these platforms:

- *Model-view-controller (MVC)*: concerned with management of the UI, controlling transitions between screens, updating screens based on business data, reading data input and translating user actions into business operations, etc.
- *Data access object (DAO)* as a gateway to access datastores/resources on behalf of business components.
- *Value object* to pass data between components as groups of platform-independent data values of entity attributes.
- *Service activator*: managing asynchronous interaction with remote synchronous services, in order to enable the main thread of the app to continue with user interaction while the service request/response is being carried out.
- *View helper*: input data validation and other local processing defined in separate auxiliary helper components, not within main view components.

- *Online/offline DAO*: an integration-tier DAO that manages offline behaviour, eg., using a local cache to store data when there is no network connection. It handles the synchronisation of local and remote data.

6.2.1 Model-View-Controller and alternative versions

A number of versions and elaborations of MVC have been used for mobile apps:

- MVP – Model View Presenter.
- MVVM – Model-View-ViewModel, which is particularly appropriate for SwiftUI (Section 6.4.2).
- VIPER – View, Interactor (use case interactor, managing model data), Presenter, Entity (model), Router/Wireframe (navigation router).

In contrast to the usual three-way interaction of conventional MVC (Figure 6.6), mobile apps tend to use the (view) controller as a bridge between views and model (Figure 6.7). Moreover, for mobile apps there is typically one controller per view, rather than one controller managing multiple views. One reason for the bridge approach is that views in mobile platforms (such as Android and iOS UIKit screens) are passive components with no programming capabilities, thus the responsibility for updating them resides with their view controller. This restriction does not apply to SwiftUI views, which are programmatic components.

The bridge approach also avoids direct connection between the view and model, however it tends to make the controller complex, because it handles both UI logic and business (use case) logic, leading to the ‘Massive View Controller’ problem.

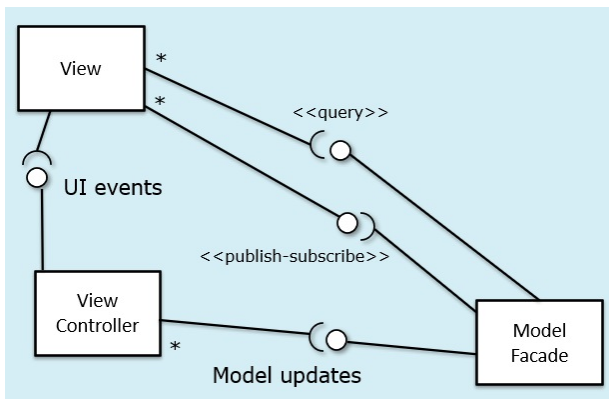


Fig. 6.6 Conventional MVC structure for mobile apps

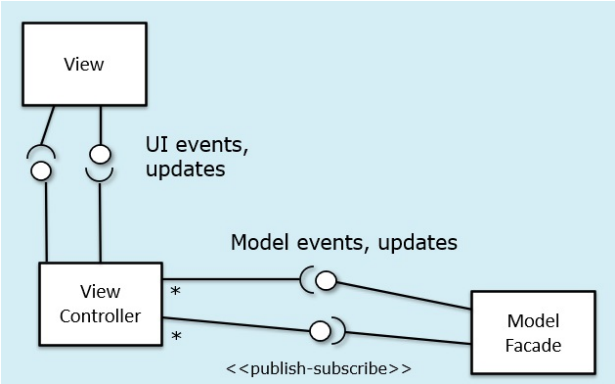


Fig. 6.7 Bridge MVC structure for mobile apps

The VIPER pattern tries to resolve this issue by separating out the MVC components into more fine-grained components which are aligned to the clean architecture levels (Figure 6.8). The Presenters are responsible for providing model data content to their associated views, and requesting model updates/queries via the interactor, based on user actions. The interactor manages model data, including the persistence of this data (via DAOs), and requests to external services.

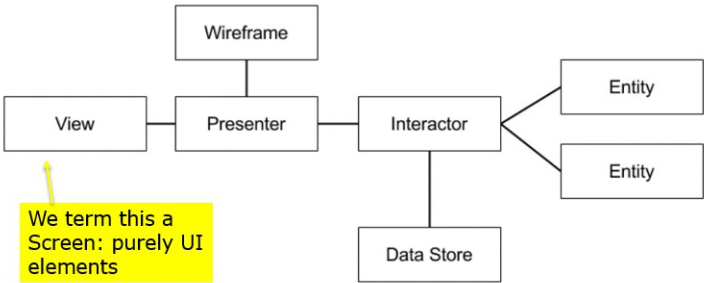


Fig. 6.8 VIPER structure (From [76])

VIPER follows the single responsibility principle, with each component having a single role, and with no mixing of functional responsibilities within components (although the VIPER Presenter involves both presenter and controller aspects in terms of the clean architecture). However this approach can result in fragmentation and many components and connections. In simple apps some of the components may be omitted.

- We can adapt VIPER as follows (Figure 6.9):
- View Controllers for individual screens. These manage local updates of the screen in response to UI events. They query and update model data

via a Model Facade. They may receive notifications of model/environment changes, which they then present on the screen they are responsible for. Screens may directly invoke operations of their view controllers, instead of via an interface, because of the close coupling of these components (e.g., Figure 5.9).

Auxiliary presenter and view helper components may be used, such as validation beans.

- Navigation Controllers are responsible for managing navigation between screens (transitions such as segues) – they are either informed of transition trigger events directly by UI elements or by delegation from view controllers.
- Model Facade has the role of an interactor in the clean architecture, implementing use cases by interaction with business entities, and interacting with local and remote resources/services via DAOs and Service Activators.
- Remote services are managed via Service Activators when the service is synchronous.
- DAOs provide a simple API for resources (e.g., CRUD operations for business entities).

Service Activators and DAOs have the role of gateways in terms of the clean architecture.

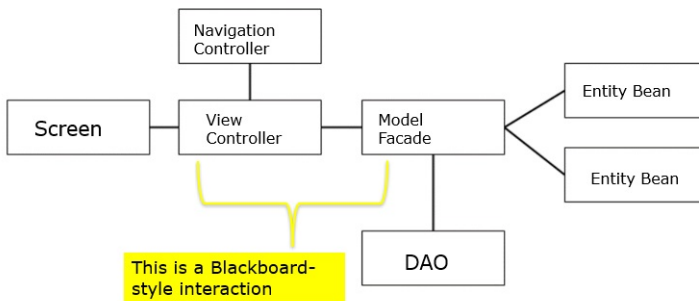


Fig. 6.9 Generalised VIPER structure

An elementary mobile architecture using only the screen and view controller elements of the adapted VIPER architecture, together with a view helper (backing bean) is shown in Figure 6.10. This could be a simple monitor for GPS (location service) events from the device, e.g., giving the current location in latitude and longitude of the device.

Figure 6.11 shows an app which plays audio files by invoking a media player service from the app platform. This has a similar type of interaction to the Service Activator pattern, with an asynchronous call to the service and a callback notification when the track is ready to play.

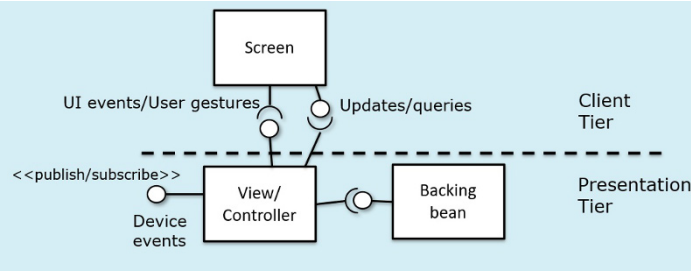


Fig. 6.10 Basic app

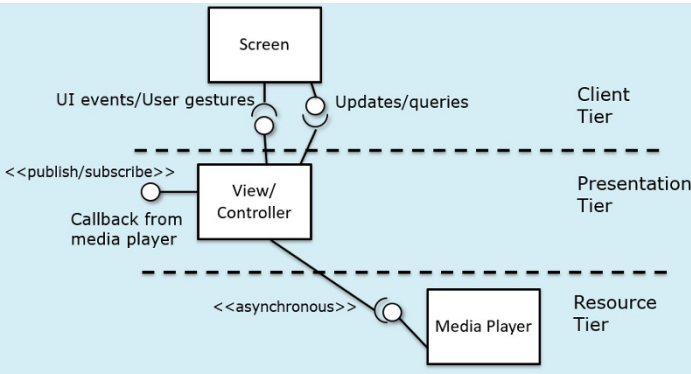


Fig. 6.11 Audio file player app

A similar example, but using Service Activator to access a remote service, is given in Figure 6.12.

Figure 6.13 shows an app with a local database, where value objects are used to transfer business data between the database DAO and other components. The DAO acts as an adapter which translates CRUD operations such as *createItem(...)* into invocations of the specific database technology API commands, such as invocation of SQL statements using the SQLite API³.

Figure 6.14 shows the architecture of an m-health app with three screens in a tabbed navigation, e.g., to compute the calories consumed by a particular exercise, for a particular person. Screen 1 could be for entry of personal information such as age, weight, gender, screen 2 for information about the type of exercise and its duration and intensity, and the final screen shows the results. The first two screens are just used for data entry, and are not updated by the app. This app would be a more elaborate version of the app4 described in Section 6.1.3. In general, single-screen apps are considered more usable than multi-screen apps, so a multi-screen UI should only be adopted where a single-screen UI is not possible.

³ sqlite.org

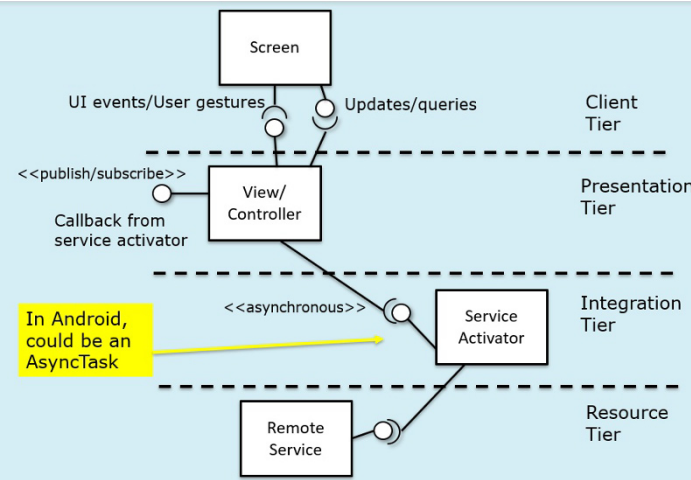


Fig. 6.12 Accessing remote service example

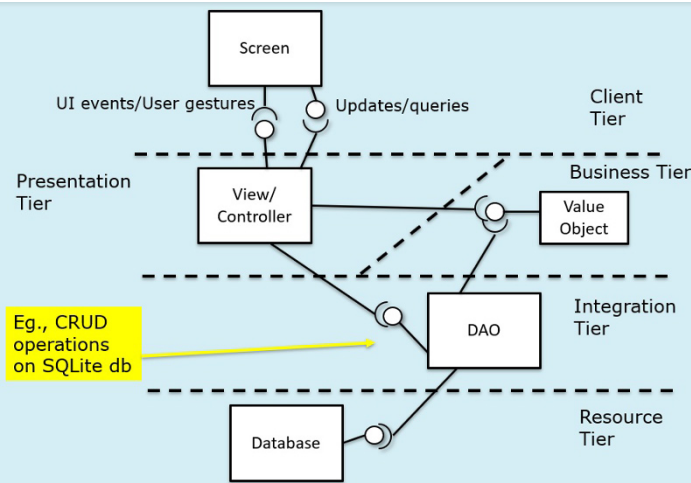


Fig. 6.13 Simple database app

Figure 6.15 shows a multiple screen app where there is a model facade performing business operations, on some ‘item’ class, e.g., products, journeys, meals or any other kind of persistent data.

In the supplementary materials on the book website we provide the complete code of a bond analysis app, app5 (Figures 6.16, 6.17) following the above VIPER-based mobile architecture organisation.

This adapted VIPER architectural style is one possible approach for building mobile app architectures. The advantages are that there is a clear separation of concerns, with different components and tiers having different pur-

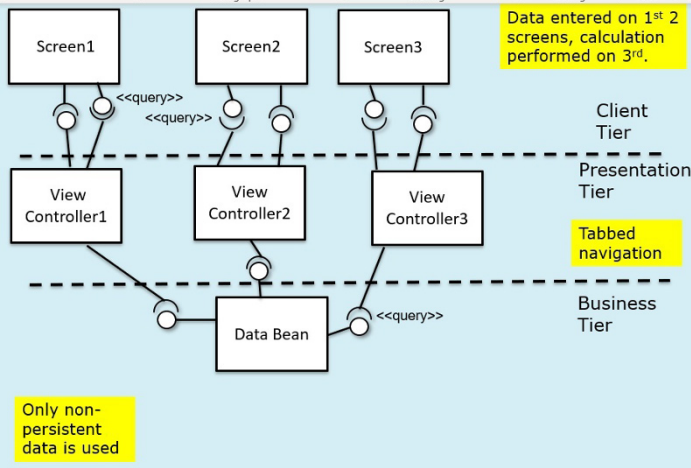


Fig. 6.14 Multiple screens example

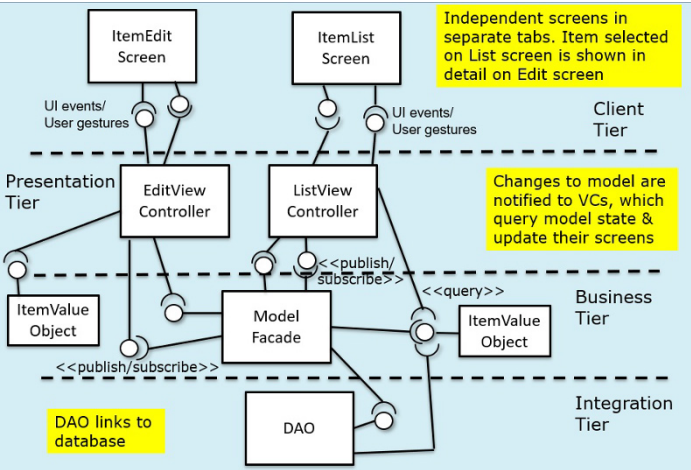


Fig. 6.15 App with multiple screens and model facade

poses. Security and portability are facilitated by a multiple tier architecture. However there is also the disadvantage that app performance can be impacted by the need to transfer data across multiple tiers/between several components. Software management of a large number of components is also time consuming for the developer(s).

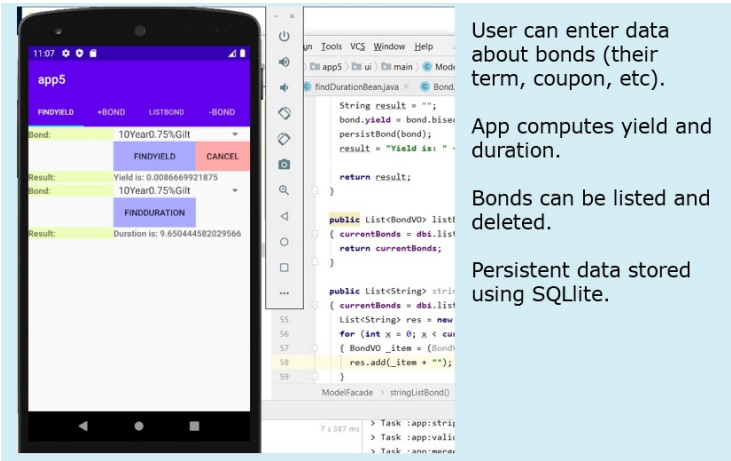


Fig. 6.16 Bond analysis app, computation screen and Android ModelFacade

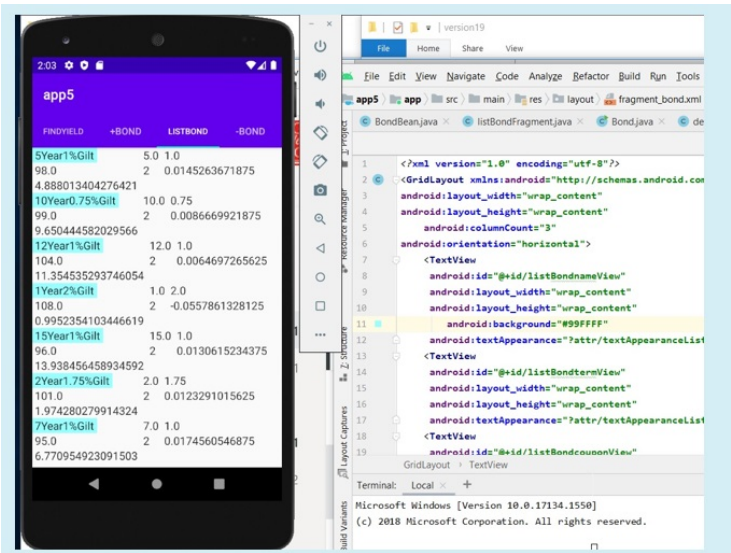


Fig. 6.17 Bond analysis app, list screen and Android UI

6.2.2 Value Object/Data Transfer Object pattern

This pattern is used to define data types for transferring business entity data between components and tiers within an architecture. The principle is that it is simpler and more efficient to transfer entity data as a single value object instead of as several individual attribute values. The value object is a passive receptacle of data, typically created as a result of a database query, or by a

UI request (e.g., the parameters of a web request originating in the fields of a client web page).

If we have a business entity class of the form

```
class E
{ attribute att1 : T1;
  ...
  attribute attn : Tn;

  operation op1(...)
  ...;

  ...
}
```

then its derived value object could look like:

```
class EValueObject
{ attribute att1 : T1;
  ...
  attribute attn : Tn;

  static operation newEValueObject(obj : E) : EValueObject
  pre: true
  post: EValueObject->exists( x |
    x.att1 = obj.att1 & ... & x.attn = obj.attn &
    result = x );

  ...
}
```

newEValueObject instantiates a value object based on the data of a business entity instance *obj*. If there are references (associations) between business entities, then these can be represented by a value object for the referred entity or by a collection of value objects of the referred entity. Alternatively, foreign key values could be used to represent references.

Additionally, value objects can be defined to hold all the parameters of a use case, so that constraints and validity conditions on these parameters can be conveniently checked.

6.2.3 Model Facade/Session Facade pattern

These components are use case interactors, which implement the use cases of an application by using business entity data. Typically, each model facade handles the use cases linked to a single actor (because of the SRP), and there

can be a provided interface for each use case (as with the process described in Section 3.3.3).

The model facade is also termed a business operations component or *session facade* because it provides a set of operations which may be executed in a single logically-connected ‘session’ of use of a system (such as adding products to a shopping cart, selecting a payment option, and completing payment). The model facade may also process callbacks from service activators, or notifications from device services such as location services.

The model facade is a central functional component of an application, and typically interacts with a presentation tier controller, and with gateway components in the integration tier to access persistent data or to invoke services. The connections to integration tier components should use dependency inversion. Figure 6.15 shows the general use of a Model facade.

6.2.4 Service Activator pattern

In this pattern, the Service Activator component is called asynchronously from the app (usually from the Model Facade interactor, or from a Controller), but invokes a synchronous task such as an HTTP connection request (Figure 6.18). The motivation for this pattern is to avoid blocking the main UI thread (indeed, Android does not permit such blocking to take place). Thus the pattern benefits the *responsiveness* quality-of-service property. Again, dependency inversion is needed for the model facade/service activator connection.

The service activator may have additional operations, e.g., so it can be queried on task progress, or halted if the task requires excessive time/resources. Typically it will execute a callback operation to inform the app model facade or controller of task success/failure.

In iOS the caller of the service activator can provide a Swift *closure* (anonymous function or lambda expression) to be used as the callback operation. In Android, an *AsyncTask* or *Service* can notify a *Receiver* to carry out a callback.

An example of a service activator app is app7 (an alternative version is app16) in the supplementary materials. This app uses the service activator pattern to download share price data from yahoo.finance. Analysis and graphing of share price information is then carried out (Figure 6.19).

6.2.5 Data Access Object (DAO) pattern

A data access object is a gateway component which serves as an intermediary between the business tier of an application and specific data storage tech-

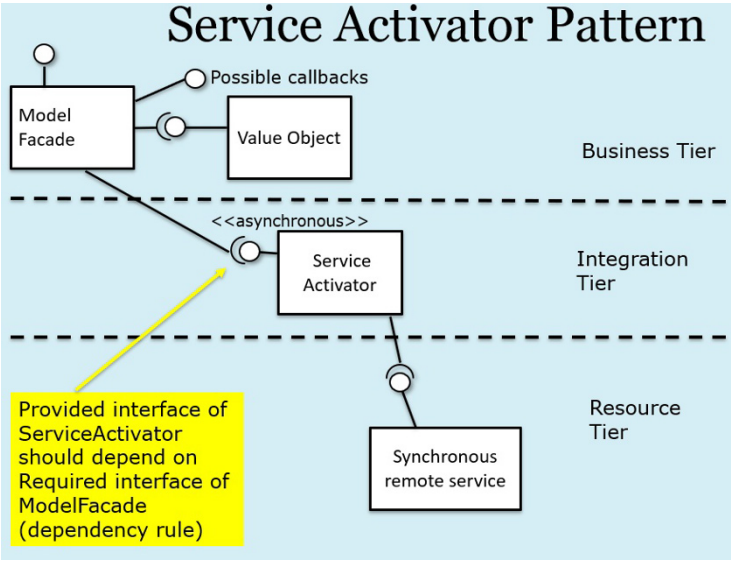


Fig. 6.18 Service Activator pattern

App loads financial data from yahoo finance, in CSV format for range of dates.

Service activator used to carry out web request asynchronously.

Calls back to model façade when data ready – uses a DAO to convert CSV to objects & stores as local (non-persistent) data.

MF computes data moving average. Graph screen component draws original data (blue line) and moving average (orange line).

Fig. 6.19 Share analysis app

nologies or other resources. Essentially it serves as an adapter (Figure 5.2), implementing business tier requests to create, read, update or delete business entity instances in terms of the specific resource API. Thus a request *createAccount(acctnId)* in a banking application could be implemented in terms of an SQL *INSERT* statement, if the resource is a relational database. Different DAOs are usually defined for different resources, and the DAO components are typically used by an interactor business tier component such as a model facade. Apart from adapting the resource API to the requirements of the business tier operations, a DAO may implement different interaction strategies for accessing and updating resources. External data which is known to be constant may be locally cached after being retrieved for the first time, to avoid unnecessary network requests. Updates to the resource may be batched, to avoid multiple network requests.

A particular strategy which is relevant to mobile apps is the *Online/offline DAO* (Figure 6.20). This DAO specialisation is concerned with writing or reading a remote data resource, handling the situation of loss of connectivity.

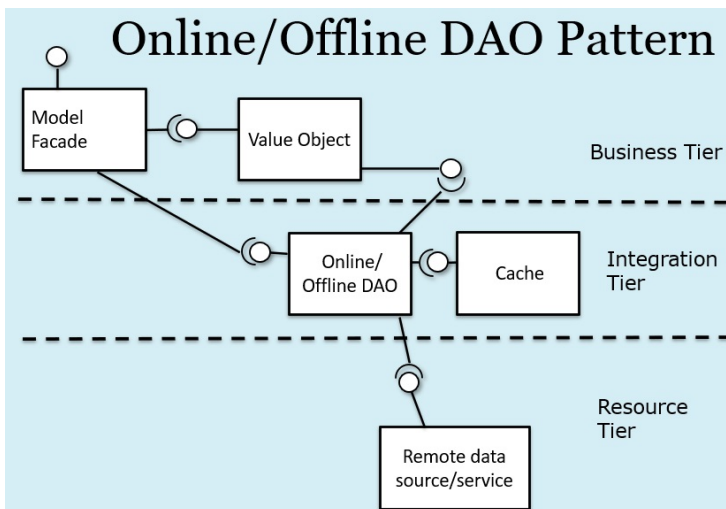


Fig. 6.20 Online/Offline DAO

Typically the online/offline DAO component carries out the following processing:

- When reading, it stores retrieved data in a local cache.
- When offline, the DAO accesses cached data.
- A time limit can be set for retaining cached data.
- Cache overflow should be handled, e.g., by discarding the oldest data.

- For writing the remote data store, the DAO maintains a queue of write instructions when offline, and executes them and removes them from the queue when the connection is restored.
- The written data is also added to the read cache.

This strategy enables some app functionality to be available even in cases where connectivity fails. It ensures that the most recent available external data is used by the app, and that update requests are executed on the external resource as soon as possible.

6.3 The Android platform and applications

Android provides a native app platform for a wide range of mobile devices, primarily phones and tablets. Android apps are usually built using the Android Studio IDE. For Android apps the client tier consists of screens defined in XML layout files. These screens are managed by Activities or Fragments in the role of View Controllers/Presenters, written in Java or Kotlin. Some navigation is provided by platform services, e.g., a tabbed interface. Navigation can also be explicitly programmed using Intents (generalised service requests). Navigation should be centralised where possible in Navigation Controllers. Embedding screen transitions within more general code can be error-prone and difficult to maintain.

For apps on the Android platform:

- Backing beans can be coded for view controllers, e.g., to carry out input data validation.
- Android adapters bind screen/view data and model data.
- In the business tier Model Facades can be coded as plain Java/Kotlin classes, that provide operations for use cases/use case steps. These are independent of specific UI and resource tier technologies.
- In the integration tier, data access objects perform database management (e.g., for SQLite, <https://sqlite.org/>). They provide interfaces with CRUD-level operations on business entities.
- Service activator objects communicate with remote resources – in Android all network communication must be launched from a separate thread, not by the main UI thread.
- Management of online/offline status may be needed.
- The resource tier contains local data sources or ‘assets’ (files, images, databases); remote services (web services); remote files (data, audio, video); user management services (e.g., on the cloud); cloud database services, and social media services.

The layouts and contents of UI screens are defined in XML, using custom tags (for example, as shown on the right side of Figure 6.17). Activities are used as ViewController components for managing specific screens/composite

views, and Fragments as components for managing parts of a composite view. Intents are requests for functionality from the platform services or application classes. Intents can be directed at specific supplier components or broadcast to be handled by any suitable supplier. E.g., an intent to open an image could be answered by any available image tool on the device.

Android adapters bind data sources to UI screen elements. E.g., usually a list adapter is used to present a list of items. Services are functional components without associated views, they can be initiated asynchronously and terminated when no longer needed. Receivers are functional components handling a single request, e.g., callbacks from services.

Android app UI logic involves NavController(s) implementing transitions from one view to another, and different ViewControllers (Activities or Fragments) managing parts of an overall interaction. E.g., there would be one view controller for the login screen, one for the main screen, and others for specific options. Fragments are used to permit alternative view arrangements for different devices (multiple fragments may be shown simultaneously on larger devices). Fragments are also used to manage screens in a screen group (e.g., a tabbed screen group).

Figures 6.21 and 6.22 show schematic architectures which are appropriate for Android single-screen and multiple-screen apps.

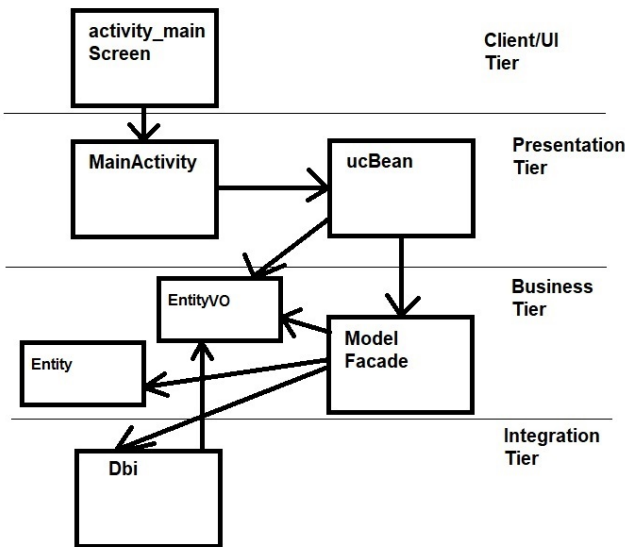


Fig. 6.21 Architecture for single-screen Android apps

The m-health app of Section 6.1.3 is an example of a simple single-screen app. It has the Android architecture shown in Figure 6.23.

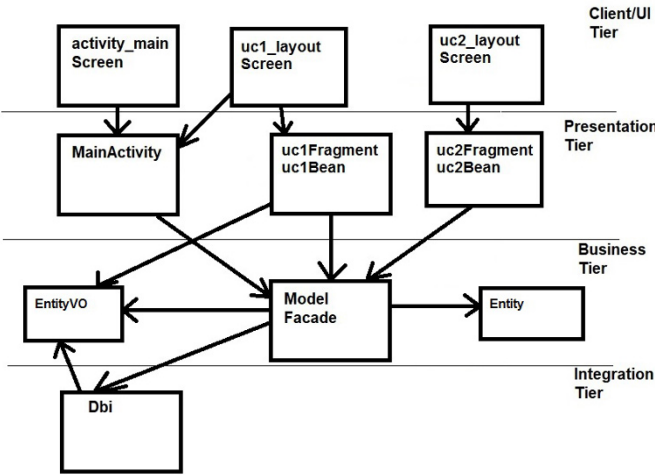


Fig. 6.22 Architecture for multiple-screen Android apps

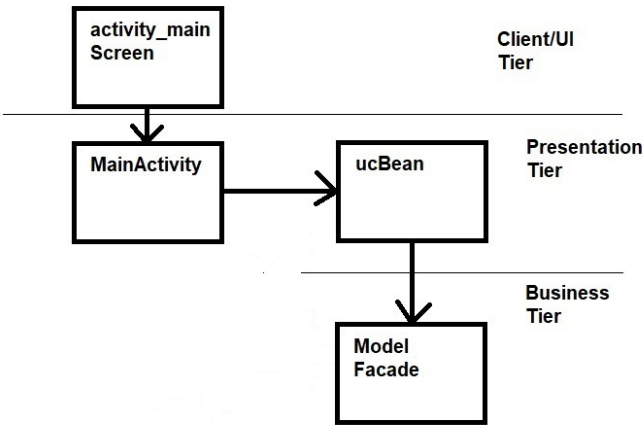


Fig. 6.23 Android architecture for app4

An example of a multi-screen Android app could be a restaurant finder app, defined with the following behaviour (Figure 6.24).

- Users can register, and set preferences.
- Price and type of cuisine can be selected (vegetarian; French, etc).
- Users can search for a restaurant matching preferences; near to a given location.
- Users can open a link to a booking site for the selected restaurant.

- Other possibilities would include placing a review on a review site, or sharing on social media.

A possible architecture of this app is shown in Figures 6.25, 6.26.

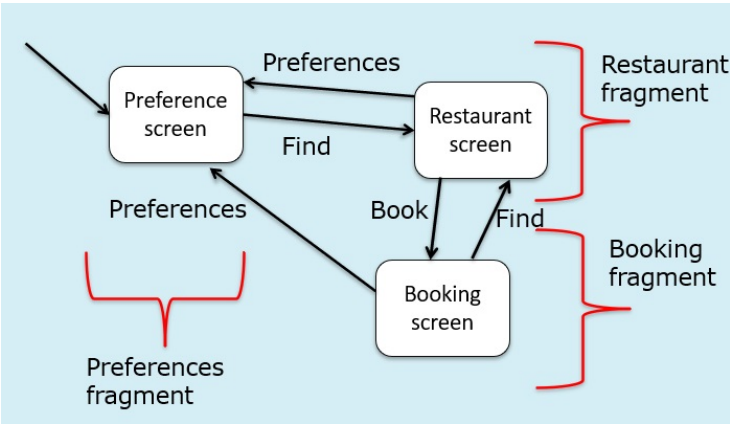


Fig. 6.24 Restaurant finder UI logic

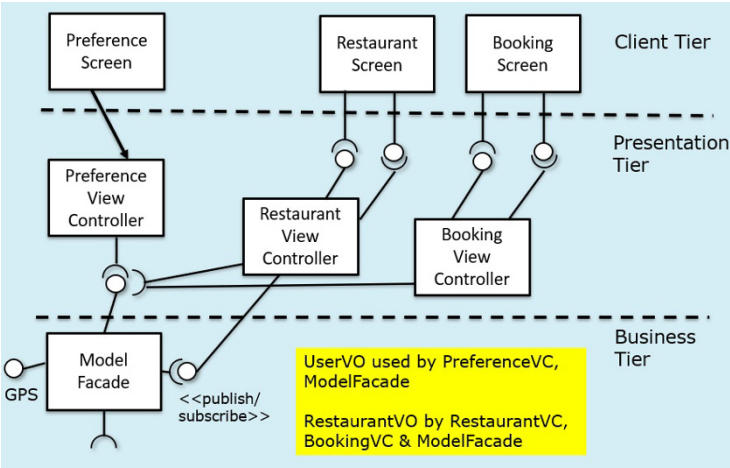


Fig. 6.25 Restaurant finder app (1)

For the m-health app, app4, the view controller of the single app screen is coded as an Android activity with the following data:

```
public class MainActivity extends AppCompatActivity
    implements AdapterView.OnItemClickListener
{ calorieCountBean caloriecountbean;
```

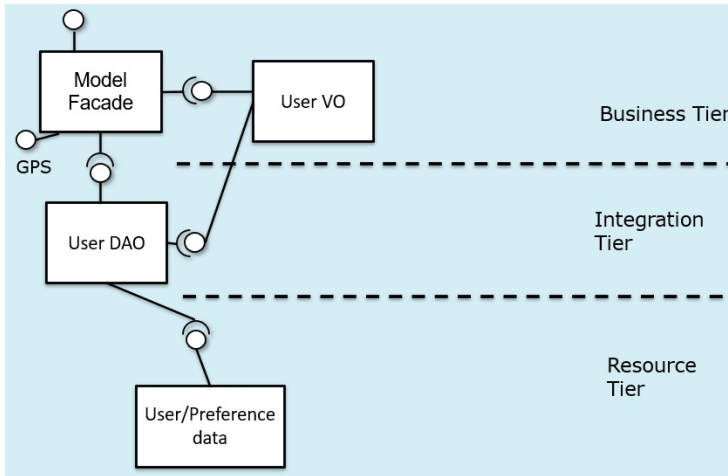


Fig. 6.26 Restaurant finder app (2)

```

computeBMIBean computebmibean;

RadioGroup genderGroup;
String genderData = "";
Spinner exerciseSpinner;
String[] exerciseListItems =
    {"walking", "jogging", "running", "swimming", "weights"};
String exerciseData = "";
EditText timeTextField;
String timeData = "";
TextView calorieCountResult;

EditText heightTextField;
String heightData = "";
EditText weightTextField;
String weightData = "";
TextView computeBMIResult;

```

calorieCountBean and *computeBMIBean* are validation beans for the respective use cases. Otherwise, the fields of the activity class correspond to the UI elements shown on the app screen. In the case of *calorieCount* these are:

- A *RadioGroup* for the gender radio buttons, together with the currently-selected gender option in *genderData*.
- A *Spinner* or selection list for the *exercise* options, identified in *exerciseListItems*, together with the currently-selected exercise in *exerciseData*.
- An *EditText* text entry field and value for *time*.
- A *TextView* field for the result of the use case.

Similarly, for *computeBMI* there are *EditText* fields for data entry and a *TextView* for the result.

Android activity classes have several standard methods, such as *onCreate*, which defines the initialisation actions of the activity and its associated screen:

```
@Override
protected void onCreate(Bundle bundle)
{ super.onCreate(bundle);
  setContentView(R.layout.activity_main);
  genderGroup = (RadioGroup) findViewById(R.id.calorieCountgenderGroup);
  exerciseSpinner =
    (Spinner) findViewById(R.id.calorieCountexerciseSpinner);

  ArrayAdapter<String> exerciseAdapter =
    new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item,
      exerciseListItems);
  exerciseAdapter.setDropDownViewResource(
    android.R.layout.simple_spinner_dropdown_item);

  exerciseSpinner.setAdapter(exerciseAdapter);
  exerciseSpinner.setOnItemSelectedListener(this);

  timeTextField = (EditText) findViewById(R.id.calorieCounttimeField);
  calorieCountResult = (TextView) findViewById(R.id.calorieCountResult);
  caloriecountbean = new calorieCountBean(this);
  heightTextField = (EditText) findViewById(R.id.computeBMIheightField);
  computeBMIResult = (TextView) findViewById(R.id.computeBMIResult);
  weightTextField = (EditText) findViewById(R.id.computeBMIweightField);
  computeBMIResult = (TextView) findViewById(R.id.computeBMIResult);
  computebmibean = new computeBMIBean(this);
}
```

In this example, *onCreate* links the activity to the app screen *R.layout.activity_main* and links each activity field to the corresponding UI widget from this screen. For the exercise list, this involves an adapter component to enable dynamic responses to selection of list items on the UI.

The responses to list item selection events are defined by the two methods (of *OnItemSelectedListener*):

```
public void onItemSelected(AdapterView<?> _parent, View _v,
  int _position, long _id) {
  if (_parent == exerciseSpinner)
  { exerciseData = exerciseListItems[_position]; }
}

public void onNothingSelected(AdapterView<?> _parent) {
  exerciseData = exerciseListItems[0];
}
```

The responses to button presses selecting or cancelling the use case functionalities are as follows:

```
public void calorieCountOK(View _v)
{ InputMethodManager _imm =
  (InputMethodManager) getSystemService(INPUT_METHOD_SERVICE);
  try {
```

```

        _imm.hideSoftInputFromWindow(_v.getWindowToken(), 0);
    } catch (Exception _e) { }

    if (genderGroup.getCheckedRadioButtonId() == R.id.calorieCountgendermale)
    { genderData = "male"; }
    else if (genderGroup.getCheckedRadioButtonId() ==
        R.id.calorieCountgenderfemale)
    { genderData = "female"; }
    caloriecountbean.setgender(genderData);
    caloriecountbean.setexercise(exerciseData);
    timeData = timeTextField.getText() + "";
    caloriecountbean.settime(timeData);
    if (caloriecountbean.iscalorieCountererror())
    { Log.w(getClass().getName(), caloriecountbean.errors());
        Toast.makeText(this,
            "Errors: " + caloriecountbean.errors(), Toast.LENGTH_LONG).show();
    }
    else
    { calorieCountResult.setText(caloriecountbean.calorieCount() + ""); }
}

public void calorieCountCancel(View _v)
{ caloriecountbean.resetData();
    timeTextField.setText("");
    calorieCountResult.setText("");
}

public void computeBMIOK(View _v)
{ InputMethodManager _imm =
    (InputMethodManager) getSystemService(INPUT_METHOD_SERVICE);
    try {
        _imm.hideSoftInputFromWindow(_v.getWindowToken(), 0);
    } catch (Exception _e) { }
    heightData = heightTextField.getText() + "";
    computebmibean.setheight(heightData);
    weightData = weightTextField.getText() + "";
    computebmibean.setweight(weightData);
    if (computebmibean.iscomputeBMIerror())
    { Log.w(getClass().getName(), computebmibean.errors());
        Toast.makeText(this,
            "Errors: " + computebmibean.errors(), Toast.LENGTH_LONG).show();
    }
    else
    { computeBMIResult.setText(computebmibean.computeBMI() + ""); }
}
}

```

These operations involve (i) the transfer of UI screen data to local data of the activity, and storage of the data in a use case validation bean (*caloriecountbean* or *computebmibean*) which also plays the role of a data transfer object; (ii) performing the validation check of the input data – this includes type checks and checks of preconditions; (iii) if invalid, displaying an error message; (iv) if valid, invoking the use case function via the bean,

and displaying the result. In this respect, the bean is acting as a *Business Delegate* [4], i.e., as a proxy for a possibly remote business service.

The activity class combines the roles of a controller and presenter in terms of the clean architecture, and is heavily platform-dependent. In contrast, the validation beans and the core business functional component *ModelFacade* are plain Java classes. The *ModelFacade* has the role of an interactor in the clean architecture terminology. For this app, it simply performs the use case computations defined in the app specification (Section 6.1.3):

```
public class ModelFacade
{
    static ModelFacade instance = null;

    public static ModelFacade getInstance(Context context)
    { if (instance == null)
      { instance = new ModelFacade(context); }
      return instance;
    }

    private ModelFacade(Context context) {}

    public double computeBMI(double height, double weight)
    { double result = 0;
      result = weight / (height * height);
      return result;
    }

    public double calorieCount(Gender gender, Exercise exercise, double time)
    { double result = 0;
      double factor = 0;
      factor = 1;
      if (exercise == Exercise.walking)
      {
          factor = 100;
      }
      else {
          if (exercise == Exercise.running)
          {
              factor = 300;
          }
          else {
              if (exercise == Exercise.jogging)
              {
                  factor = 200;
              }
              else {
                  factor = 250;
              }
          }
      }
      result = factor * (time / 60);
      return result;
    }
}
```

ModelFacade is defined as a Singleton class, so that the unique instance can be globally referenced as *ModelFacade.getInstance(cntxt)*.

6.4 The iOS platform and applications

IOS is the major competitor to Android as a native app platform. Unlike Android, which can run on the devices produced by many different phone manufacturers, the same company (Apple) produces iOS and the devices that it runs on, thus ensuring tighter integration (in principle) of devices and the OS. Apps are designed using the Xcode IDE.

6.4.1 UIKit

UIKit was the main iOS platform framework for apps, prior to SwiftUI. In UIKit:

- UI screens are defined in .xib and .storyboard files for individual screens and interconnected screens – the screen layout and elements are designed using the Xcode visual editor.
- ViewControllers are defined as UIViewController Swift or Objective-C files.
- Transitions between screens are defined as named segues.
- VIPER and related MVC patterns are suitable.

Using the ‘bridge MVC’ pattern, the app4 m-health app can be coded in UIKit as follows:

- One screen and associated view controller *ucViewController* for each use case *uc*. The view controller combines controller and presenter roles in terms of the clean architecture.
- For each use case *uc*, a validation bean *ucValidationBean* is used to check the parameters and preconditions of *uc*.
- A *ModelFacade* component contains the implementation of all the use case activities. This is an interactor in terms of the clean architecture.

As an example, the view controller for *computeBMI* has the form:

```
import UIKit

class computeBMIViewController : UIViewController {
    var modelFacade : ModelFacade = ModelFacade.getInstance()
    var computeBMIValidator : computeBMIValidationBean =
        computeBMIValidationBean()

    @IBOutlet weak var heightInput : UITextField!
```

```

@IBOutlet weak var weightInput : UITextField!
@IBOutlet weak var resultOutput: UILabel!

var userId : String = "0"

override func viewDidLoad()
{ super.viewDidLoad() }

@IBAction func computeBMI(_ sender: Any) {
    guard
        let height = Double(heightInput.text),
        let weight = Double(weightInput.text)
    else { return }

    if computeBMIValidator.iscomputeBMIErrror(height, weight)
    { return }

    var result : Double = modelFacade.computeBMI(height, weight)
    resultOutput.text = String(result)
}

override func didReceiveMemoryWarning()
{ super.didReceiveMemoryWarning() }
}

```

The @IBOutlet annotations indicate that these variables are bound to corresponding UI widgets. @IBAction indicates that the function is invoked by a UI gesture such as a button press.

The guard construct in *computeBMI* attempts to convert the text input field values to *Double* values, and only succeeds if this is possible, otherwise the else clause is executed.

As in the case of Android apps, the model facade is a plain Singleton class, without platform dependencies:

```

import Foundation
import Darwin

class ModelFacade
{ static var instance : ModelFacade? = nil

    static func getInstance() -> ModelFacade
    { if (instance == nil)
        { instance = ModelFacade() }
        return instance!
    }

    init() { }

    func computeBMI(height : Double, weight : Double) -> Double
    { var result : Double
        result = weight / (height * height)

        return result
    }
}

```

```

func calorieCount(gender : Gender, exercise : Exercise, time : Double)
    -> Double
{
    var result : Double
    var factor : Double = 0
    factor = 1
    if exercise == Exercise.walking
    {
        factor = 100
    }
    else {
        if exercise == Exercise.running
        {
            factor = 300
        }
        else {
            if exercise == Exercise.jogging
            {
                factor = 200
            }
            else {
                factor = 250
            }
        }
    }
    result = factor * (time / Double(60))

    return result
}
}

```

6.4.2 SwiftUI

SwiftUI adopts both a different UI definition approach and a different conceptual execution model to UIKit. Whereas both Android and UIKit are event-driven (UI content is the result of a series of events), SwiftUI is data-driven (the UI content is a function of the model data).

UI screens are defined as structs (simplified classes), and also perform the role of view controllers, with a simplified interaction between the model and UI. Certain model data is published as available for UI screens to observe. The layout and content of screens is defined as a function of this data. Local data of a SwiftUI view is declared as `@State` variables, whilst observed data is declared as `@ObservedObject` data. For example:

```

struct calorieCountScreen : View {
    @State var bean : calorieCountVO = calorieCountVO()
    @ObservedObject var model : ModelFacade
    ...
}

```

Transitions between screens are defined via navigation views. Programming is in Swift. UIKit components can be used via wrappers. A suitable architectural pattern is Model-View-ViewModel (MVVM), shown schematically in Figure 6.27. In this pattern the views and view controllers are combined, and the model facade acts as a ViewModel, selectively publishing some model data for the views to access to determine their structure and content. The model facade also updates and queries model data and interacts with the integration tier for data persistence and other services.

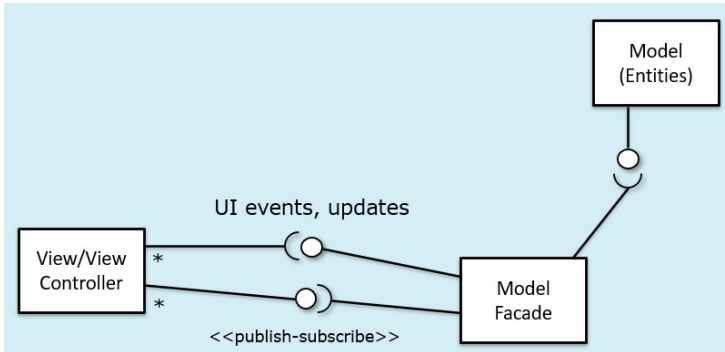


Fig. 6.27 Model-View-ViewModel

In the case of app4, each use case has a screen defined as a SwiftUI View, and a model facade component acts as the source of observable model data. As with Android and UIKit, the model facade is an interactor component which implements each use case.

The complete screen definition for *computeBMI* is:

```

import SwiftUI

struct computeBMIScreen: View {
    @State var bean : ComputeBMIVO = ComputeBMIVO()
    @ObservedObject var model : ModelFacade
    @State var res : Double = 0.0

    var body: some View {
        VStack(alignment: .leading) {
            HStack {
                Text("Height:").bold()
                Divider()
                TextField("Height", value: $bean.height,
                    formatter: NumberFormatter()).keyboardType(.decimalPad)
            }

            HStack {
                Text("Weight:").bold()
                Divider()
                TextField("Weight", value: $bean.weight,

```

```

        formatter: NumberFormatter()).keyboardType(.decimalPad)
    }

    HStack(spacing: 20)
    { Button(action: { self.model.cancelcomputeBMI() } )
      { Text("Cancel") }
      Button(action: { res = self.model.computeBMI(_x: bean) } )
      { Text("ComputeBMI") }
    }.buttonStyle(PlainButtonStyle())

    HStack(spacing: 20) {
        Text("Result:")
        Text(String(res))
    }
    }.padding(.top)
}
}

```

VStack vertically aligns the contained subelements, whilst *HStack* horizontally aligns the subelements.

ComputeBMIVO acts as a data transfer object to group together all input data of the *computeBMI* use case for transmission to the model facade. The model facade is very similar to the UIKit version:

```

import Foundation

import Darwin
import SwiftUI

class ModelFacade : ObservableObject
{ static var instance : ModelFacade? = nil

    static func getInstance() -> ModelFacade
    { if instance == nil
      { instance = ModelFacade() }
      return instance!
    }

    init() { }

    func cancelcomputeBMI() { }

    func computeBMI(_x : ComputeBMIVO) -> Double
    { var result : Double = 0.0
      if _x.iscomputeBMIerror()
      { return result }
      let height : Double = _x.getheight()
      let weight : Double = _x.getweight()

      result = weight / ( height*height)

      _x.setresult(_x: result)
      return result
    }
}

```

```

func cancelcalorieCount() { }

func calorieCount(_x : calorieCountV0) -> Double
{ var result : Double = 0.0
  if _x.iscalorieCountererror()
  { return result }
  let gender : Gender = _x.getgender()
  let exercise : Exercise = _x.getexercise()
  let time : Double = _x.gettime()

  var factor : Double = 0
  factor = 1
  if exercise == Exercise.walking
  {
    factor = 100
  }
  else {
    if exercise == Exercise.running
    {
      factor = 300
    }
    else {
      if exercise == Exercise.jogging
      {
        factor = 200
      }
      else {
        factor = 250
      }
    }
  }
  result = factor*(time/Double(60))

  _x.setresult(_x: result)
  return result
}
}

```

6.5 Comparison of iOS versus Android

The two platforms have many similarities at the level of concepts (Figure 6.28), however the details of technologies and programming languages used differ considerably. With the advent of SwiftUI, the architectural approaches relevant for the app platforms have also diverged.

	Android	iOS/UIKit	iOS/SwiftUI
Screens	XML files with layout definitions	UIKit view, defined via Xcode	SwiftUI View struct
View Controller	Activity or Fragment class	UIViewController class	Combined with the screen
Navigation	Tabs or Activity/Fragment invocation via Intents	Screen transitions via segues	Tab and navigation views
UI/Model interaction	UI events trigger model changes; UI components query model, or asynchronously updated	As for Android	UI/Model bindings. UI is a function (Observer) of observable model data

Fig. 6.28 Comparison of Android and iOS

6.6 Mobile app development approaches

The division of mobile app technologies between Android and iOS means that app developers need to target both platforms if they want to achieve the widest market for their products. This results in substantial duplication of effort if fully manual development is used.

Cross-platform app development approaches aim to produce app implementations for both Android and iOS, without the need to manually code separate implementations. Cross-platform approaches include web apps, interpreted apps, cross-compiled apps, and model-driven app development:

- *Web apps*: this involves creating an app using web technologies such as HTML, CSS, and JavaScript, and deploying this to a URL which enables mobile users to access the app via device-based browsers. Thus there is no need to install the app to access it. However this approach depends upon the availability of an Internet connection, and web apps have restricted access to device services and sensors.
- *Interpreted apps*: this means that an app is written using a cross-platform interpreted language such as JavaScript. At runtime, a language interpreter executes the app code, with native UI components being used. React Native from Facebook⁴ and Titanium⁵ are examples of this concept. However the approach generally suffers from poor performance and high resource usage due to the inefficiency of runtime code interpretation compared to compiled execution.

⁴ <https://reactnative.dev>

⁵ <https://titaniumsdk.com>

- *Cross-compiled apps*: in this approach, an app is coded in a general programming language such as C# in Xamarin from Microsoft⁶ and Dart in Flutter from Google⁷. The codes are compiled to native byte code and this executes directly on the Android and iOS platforms.
- *Model-driven app development*: in this approach a single app specification is written in a language such as UML, or in a domain-specific modelling language (DSL), and native codes for Android and iOS are automatically generated from the specification [51]. This combines the benefit of a purely native implementation with the use of a single app specification.

Each of these approaches/frameworks have appropriate architectures that are typically used with the approach. For web apps, a web application architecture can be used (Chapter 9), taking into account the restrictions of mobile user interfaces: web pages should be relatively simple, without excessive content, and interaction sequences should not span more than 4 separate pages.

React works naturally with the Redux architecture based on Elm⁸ and the classical Command design pattern [31]. The distinctive aspect of this architecture is that the *Model* part is treated as immutable. UI and other external events are converted into actions (command instances), and are applied by an interactor component in the business tier to produce new model instances (instead of updates to the model). This new instance is then rendered to the UI via Views. The use of immutable state avoids concurrent execution issues such as race conditions and deadlocks. In terms of the Command pattern, the Redux Model or Store is the Command Receiver, whilst the Redux Action(s) are Command(s) (Figure 6.29).

The inefficiency of JavaScript interpretation has been addressed in recent versions of React by the use of compiled C++ libraries (Figure 6.30).

Flutter uses a MVVM approach similar to SwiftUI, in which the screen layout and content is a function of the app state. Xamarin is integrated into the .NET platform, so event-based processing architectures such as MVC would be more natural in this case.

Model-driven approaches can utilise any appropriate architecture, the approach of [51] uses the adapted VIPER architecture described in Section 6.2.1 above.

6.7 Mobile backend as a service (MBaaS)

MBaaS provides off-device services for mobile apps, such as cloud-based app user authentication/management, push notification services, data storage on

⁶ <https://dotnet.microsoft.com/en-us/apps/xamarin>

⁷ <https://flutter.dev>

⁸ <https://guide.elm-lang.org/architecture>

Fig. 6.29 Redux architecture

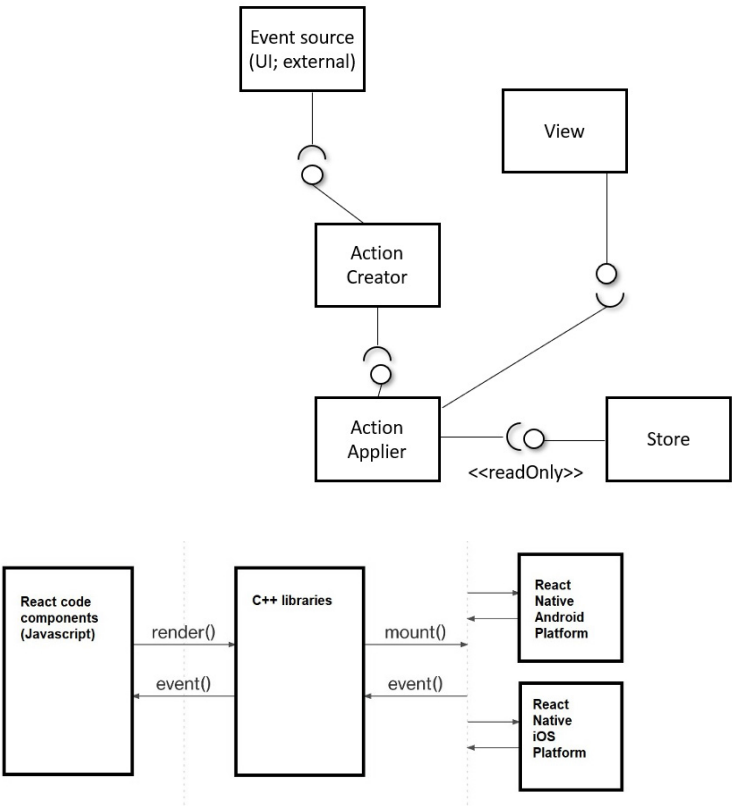


Fig. 6.30 React implementation architecture (From react.native.dev)

the cloud, application analytics and social media integration into apps via federated identity.

The idea is similar to software/infrastructure as a service (Chapter 9), but oriented to mobile app clients. MBaaS services often use a *freemium* monetisation model: service use is free up to a certain level of use, then charged per usage. A widely-used MBaaS is Firebase⁹, a Google cloud database providing:

- App user authentication/management, email/password managed by Firebase or 3rd party accounts from Google, Facebook, Twitter, GitHub.
- Data storage using JSON tree-structured data.
- Apps can listen for data-change and authentication-change events in their Firebase data store.
- Application analytics and crash reporting.

⁹ firebase.google.com

- Deployment of functions to the cloud.
- Testing facilities for Android apps.
- Freemium monetisation model: free use up to a quota.
- API for use in either iOS or Android apps.

Summary

In this chapter we have described mobile app architectures and patterns, including MVC and variations on this, Service Activator and Online/Offline DAO. The Android and iOS platforms have been described, together with development approaches for these platforms, and the concept of MBaaS.

Further Reading

- Clean architecture for SwiftUI, Medium, 2023.
<https://medium.com/swlh/clean-architecture-for-swiftui-6d6c4eb1cf6a>
- R. Nunkesser, *Choosing a global architecture for mobile applications*, TechRxiv, <https://doi.org/10.36227/techrxiv.14212571.v1>, 2021.
- D. Sanchez, A. Rojas, H. Florez, *Towards a clean architecture for Android apps using model transformations*, IJCS, vol. 49, no. 1, 2022.

Exercises

1

Design and draw an MVVM app architecture for a SwiftUI app which monitors share prices provided by a real-time data feed. The app alerts the user if the price of a tracked share moves outside a preset range. The app user can add/remove shares for monitoring, and edit the alert range for shares.

2

Discuss the advantages and disadvantages, for an app developer, of using *MBaaS*.

Explain whether MBaaS is appropriate for the following systems:

- A travel messageboard app, where users can post accounts of visits to different tourist destinations, and comments on these accounts.

- An internal hospital app which enables hospital staff to manage their appointments. This can involve confidential patient data.

3

Design an appropriate architecture for an Android app which enables users to see the carbon footprint of different proposed journey routes and modes of transport. There is a single screen showing a map, with two text fields for journey start and end locations (these locations can alternatively be specified by clicking on the map), and a selection list for transport mode. The resulting carbon footprint is then displayed. How could the app itself be made more environmentally-friendly?

4

In the chapter we suggested that in a master-detail situation, where one screen shows a list of items, and another shows a selected item, that the blackboard architecture could be used. How could this be simplified, if we know that only one screen can be visible to the user at any one time?

5

Design a mobile app architecture (suitable for either Android or iOS implementation) for a travel journal app. Draw a diagram showing all the components that you propose to use, and explain which patterns you are using, with a justification for their suitability.

The app requires user registration/login, and enables registered users to maintain a list of travel journals describing their holidays. Journals consist of a text file and one or more image files, and the app enables these to be created/edited/deleted and stored persistently in a cloud database. You can assume that user authentication/management is also handled by the cloud platform.

6

Draw an architecture for a mobile app which classifies images stored in jpeg files, and extracts text content. The image classification is performed on-device by Google MLKit using a local ML model, and text extraction is performed using the MLKit OCR library.

Local data is stored, which records the image and text elements identified in particular images. This data can be persisted to files in the local app filespace.

The GUI is a tabbed interface with two tabs, one for image selection and display, and the other to display the data extracted from the image. You can assume that navigation between tabs is handled by the app platform and does not need to be additionally programmed.

Explain which mobile app patterns you use and why they are relevant for this system.

7

Identify how the Redux architecture relates to the classical MVC architecture, i.e., what elements of Redux correspond to the MVC elements.

Projects

1

Develop the Android trip-planner system described in Exercise 3 above.

2

Develop the travel journal app described in Exercise 5 above, using a cross-platform approach to support both iOS and Android implementations.

3

Develop the image classifier app described in Exercise 6 above, for the Android platform. You can choose any image-classification task, such as classifying meals as vegetarian or non-vegetarian, classifying pet images, etc.



Chapter 7

Enterprise Information Systems and Application Servers

Abstract This chapter covers architectural styles and patterns for enterprise information systems (EIS) and the concepts of application servers, and the motivations for using them. We look specifically at the .NET EIS platform and .NET enterprise services.

7.1 Enterprise systems

Enterprise systems are business systems for large scale organisations, these systems typically have many users of different kinds, such as staff, customers, other businesses, and they integrate a wide range of different workflows concerning the same core business data.

EIS can be characterised as:

- Large-scale business-critical systems, performing significant business functionality, central to the services and operations of an organisation.
- System failure can result in substantial financial loss and potentially reputation loss.
- The systems need to satisfy multiple quality-of-service (QoS) properties such as fault-tolerance, efficiency, scalability, security, trustworthiness, etc.

7.1.1 Application servers

The need to consider QoS properties adds complexity to applications, because software developers cannot focus on business logic alone, and ensuring the properties may require specific additional expertise.

The concept of an *application server* or *middleware* was introduced to address this problem. An application server platform such as Java EE or .NET

provides standard QoS support for the business-logic layer of applications running on the platform. This support is incorporated into applications via annotations and configuration of the applications, and does not need additional implementation work, so that developers can focus on business logic.

Application servers typically assume an N-tier architectural style, and house business-logic components within a middle tier (Figure 7.1). Other tiers may also need additional middleware, which we do not consider here.

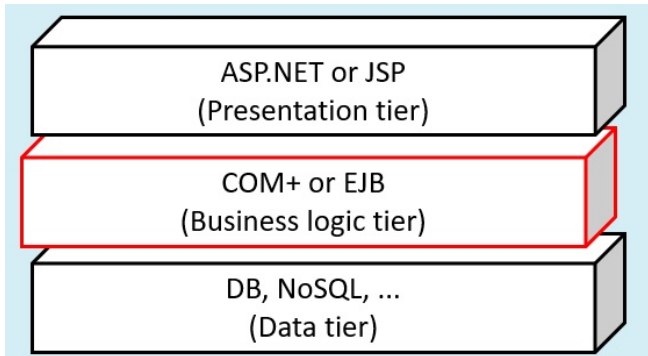


Fig. 7.1 Application server in N-tier architecture

The characteristics of an application server platform are that:

- Components are *managed* by the platform.
- Instantiation, connection, and garbage collection of component instances are the responsibility of the middleware, not the application programmer.
- All component interaction can be intercepted by middleware, to achieve transparency of location, lifetime, etc.

The management of components is typically achieved by a Proxy-style arrangement or *Stub-Skeleton Pattern* between components (Figure 7.2).

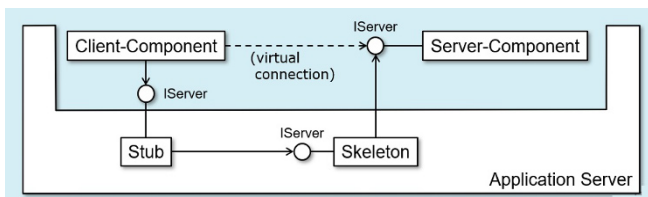


Fig. 7.2 Stub-Skeleton pattern

This pattern provides *language transparency* between components, i.e., they can communicate and interact even if they are written in different pro-

gramming languages. It also provides *location transparency* – independence from where a component runs (locally or remotely), and *lifetime transparency* – independence from whether a component is currently running or needs to be instantiated, and more generally it conceals how many instances of a component there are.

7.2 .NET enterprise services

.NET is a concrete technology for EIS, providing enterprise services for managed business tier components termed COM+ components. .NET enterprise services are a suite of pre-programmed functionalities available to COM+ components, and are designed to solve many performance, scalability, and fault-tolerance requirements of an enterprise. However, it should always be remembered that the adoption of any framework or platform for your application also has a disadvantage, in that the application becomes dependent on the services offered by the platform, and therefore it will be difficult to migrate it to a new environment.

.NET enterprise services include:

- Transactions.
- Object pooling.
- Just in time (JiT) activation.
- Loosely coupled messaging.
- Queued components.

The specification of the services needed by a component can be given by a declarative design-time configuration whereby developers apply service-related attributes/annotations at design time. .NET attributes are pieces of metadata attached to class declarations – program-level equivalents of UML stereotypes, which give extra information about the annotated class/element.

We will use UML component diagrams to model COM+ architectures, using customised stereotypes and tagged values for the platform. Recall that stereotypes are UML annotations *«stereotype»* indicating the specialisation of a UML element. For example *«BusinessCase»* to indicate a specialised form of use case. Each stereotype can also be associated with one or more *tagged values* – additional (meta) attributes used to specify properties of the specialised concept. For example, *actor = “CEO”* could be a tagged value assignment for the stereotype *«BusinessCase»* (Figure 7.3).

Figure 7.4 shows an example of a C# COM+ component and its equivalent representation in a UML component diagram. The C# class implements *ServicedComponent*, which marks it as a .NET managed class. For managed classes a no-argument public constructor is necessary, so that the platform can instantiate the class as needed.

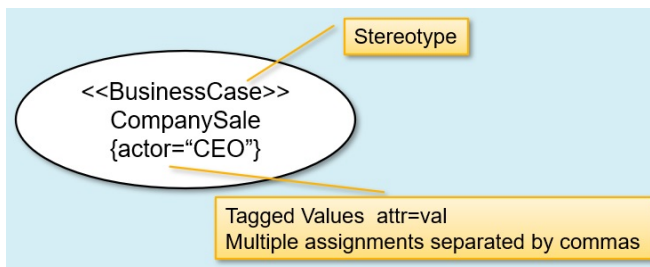


Fig. 7.3 Stereotypes and tagged values

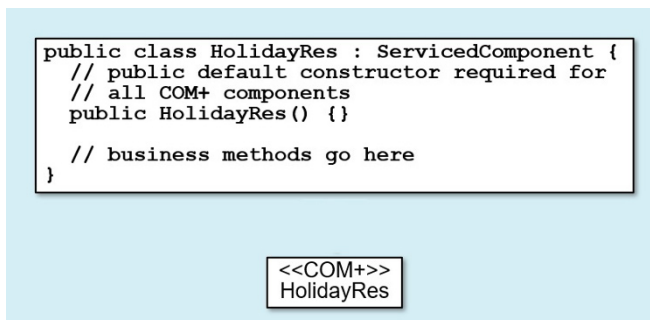


Fig. 7.4 COM+ component in C# and UML

7.3 Transactions service

Transactions concern the *atomicity* of actions, i.e., to what extent a sequence/collection of actions should be processed as an indivisible whole. We say that an interaction is *transactional* if failure of individual steps leads to failure of the entire interaction, and the undoing of the effects of steps that have completed. Transactions are often designed in order to ensure that critical business invariants are maintained: if the invariant would be violated by the execution of an incomplete set of steps within the transaction, then the entire transaction is cancelled and no update to the state takes place. Only correct complete executions are permitted.

Figure 7.5 shows the series of execution steps (a UML *interaction model*) for a version of our holiday reservation system – the system takes a reservation request, and then:

1. Checks that the requested flight and hotel are available and that the credit card is valid.
2. If all checks pass, charges the card, then confirms the flight and hotel bookings.

A problem arises if a component such as *HolidayRes* crashes, or if the card becomes invalid, during processing. Or if someone else booked the last seat on

the plane during the booking processing, but the card was charged already. Without transactions, the interaction permits a room to be reserved, but not paid for, or that the card is charged, but no holiday reserved. This is a case where transactionality is essential. The maintained business invariant in this case is that all paid-for services have been booked and confirmed, and that all booked and confirmed services have been paid for. Moreover, that for each trip, both a flight and hotel have been booked, or neither.

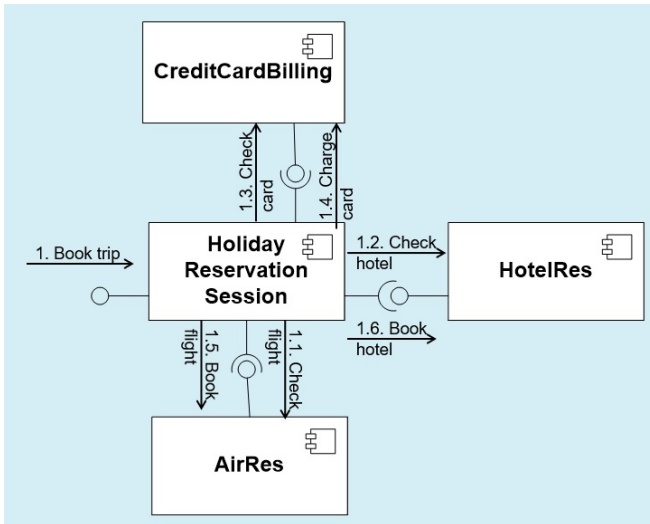


Fig. 7.5 Holiday booking scenario

Transactions provide an “all-or-nothing” semantics for a group of operations or components: either all operations in the transaction group succeed or none do.

If the holiday reservation interaction is made transactional, then we can ensure that if the card has been charged, then we have booked a flight and a hotel. Alternatively, if we have no hotel, then the card has not been charged and we have not booked a flight.

Transactions are difficult to implement manually, so this is a case where an application server is relevant. .NET transactions provide transactional semantics as a service.

Transactions were originally a concept from databases. Database transactions group a set of database operations and ensure the ACID properties:

- Atomicity – all or nothing behaviour for the group.
- Consistency – ensuring consistent data at the end of the interaction.
- Isolation – concurrent transactions do not affect each other.
- Durability – once changes have been made, they are not lost except through new changes from a following transaction.

In COM+, transactions have the following properties:

- A group of components are associated with a transaction.
- When an operation of the group is invoked from outside the group, the COM+ transaction saves the state of all components in the group.
- If a step of the operation fails, the state of the group is returned to what it was just before the call – this is termed a *rollback*.
- Otherwise, all changes are *committed*.

COM+ transactions ensure the atomicity and consistency properties, and can ensure different levels of isolation. Transactions can be nested for more fine-grain control.

Transactions in COM+ are indicated by a Transaction annotation, with attributes:

- *Transaction level*, which can have the values:
 - Required – meaning that the component must be run within a transaction: if a transaction is already running, the component joins that transaction, otherwise, COM+ will create a new transaction for the component.
 - Supported – if a transaction is already running, the component joins that transaction.
 - Requires new – COM+ will always create a new transaction for the component.
 - Not supported – the component will never run within a transaction.
- *Isolation level* – this defaults to Serializable, the attribute affects the extent to which transactions can see changes made by concurrent transactions.

Figure 7.6 shows an example of COM+ annotations for a transactional version of *HolidayRes*. In component diagrams, transaction attributes can be shown as corresponding tagged values (Figure 7.7).

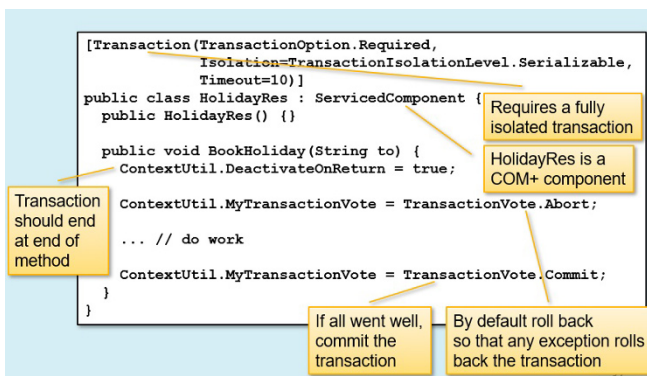


Fig. 7.6 Transaction service example code

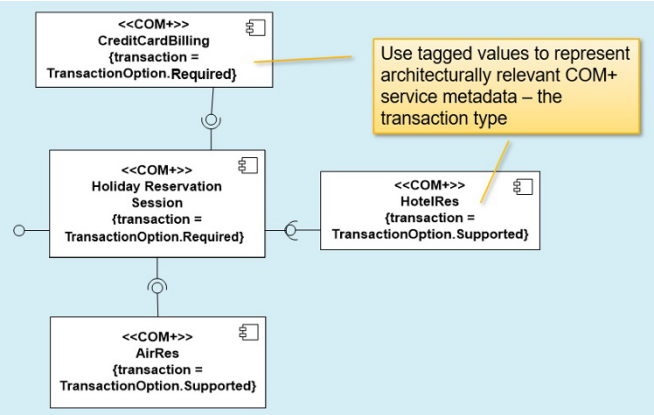


Fig. 7.7 Transaction service specification in UML

7.4 Object pooling service

The aim of the object pooling service is to improve performance by pre-initialising component instances which require significant resources to create. The service keeps a set of component instances active in a pool shared by clients. When a client instantiates a component, the reference is actually linked to an existing instance in the pool, so that no new instance is actually created. If a client calls an operation on the component, the call is delegated to an arbitrary available member of the pool (Figure 7.8). If all pool members are currently serving calls, the call is queued until a supplier instance becomes available.

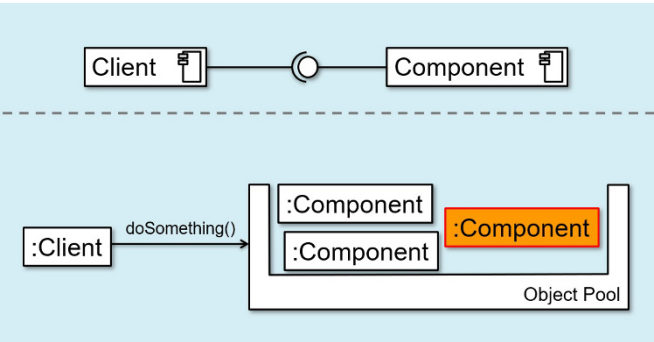


Fig. 7.8 Object Pooling service

An example could be a bank, where many bank clerks all need to access a database or remote connection to access customer details. Only a certain number of connections may be available, and initialising a connection could be a resource-expensive task.

In the object pooling service, COM+ manages the pool and handles details of object activation, selection, reuse, and deactivation. Developers specify configuration parameters, such as the minimum and maximum pool size, and creation request time outs. An admin console allows for monitoring the pool status.

Advantages of the object pooling service include:

- Significant performance and scaling benefits.
- Reduces object use time for each client.
- Factors out time-consuming initialisation and resource acquisition.
- Shares the cost of acquiring expensive resources across all clients by pre-allocating objects when the application starts, before any client requests come in.
- Governs resource use, e.g., setting maximum pool level to keep open only as many database connections as there are licenses.
- Takes best advantage of available hardware resources – the pool configuration can be adjusted as hardware resources change.

Object pooling requires that pooled objects cannot retain client state, or be bound to a specific thread, because the objects must be interchangeable and equivalent from the client perspective.

UML component notation for pooling uses tags based on .NET attributes (Figure 7.9).

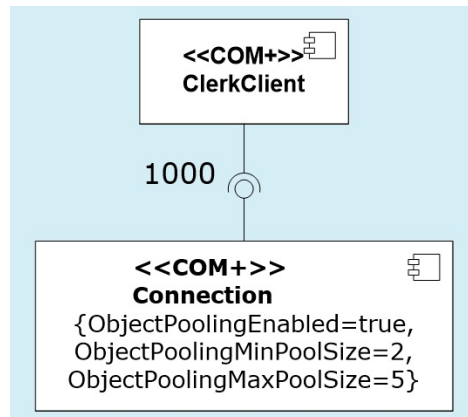


Fig. 7.9 Object Pooling service in UML

Object pooling is related to the Object Pool design pattern [33].

7.5 Just-in-time activation service

This service addresses the situation where components are resource-expensive to maintain in a running state, but relatively cheap to instantiate. For example, a share trading system could use real-time analytics in an *Information* component, which interacts with an external data source via a specialised finance communication protocol. This consumes significant memory resources in operation, so performance would improve if *Information* objects were only created when needed.

The COM+ Just-in-Time (JIT) Activation service deactivates an unused component object, but enables clients to still maintain references to it. On the next call to the component, the object is reactivated transparently, just in time. This gives life cycle transparency, and allows resources to be used more productively.

Figure 7.10 shows the .NET notation for the JIT service, and Figure 7.11 shows the component diagram notation.



Fig. 7.10 Just-in-time service in COM+

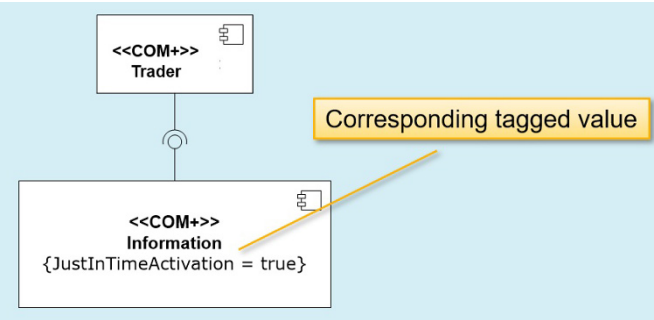


Fig. 7.11 Just-in-time service in UML

7.6 Loosely-coupled events service

In Chapter 4 we described the publish/subscribe asynchronous style of message passing, this is a platform-neutral abstraction of specific mechanisms such as Java UI event listening. The COM+ loosely-coupled event service supports publish/subscribe through late-bound asynchronous method calls, and generalises it to include an additional intermediary event forwarding/broadcast component to further decouple the subscriber and publisher.

With a publish/subscribe connector, subscriber components subscribe to events from the publisher, and the publisher sends events to all subscribers. In COM+ additional sink components called *event classes* are introduced as intermediaries, these act purely as sinks and do not contain any implementation of their own. Components may subscribe to an event sink, the publisher calls methods on the event sink and COM+ (not the event class) delegates these calls to all subscribers (Figure 7.12).

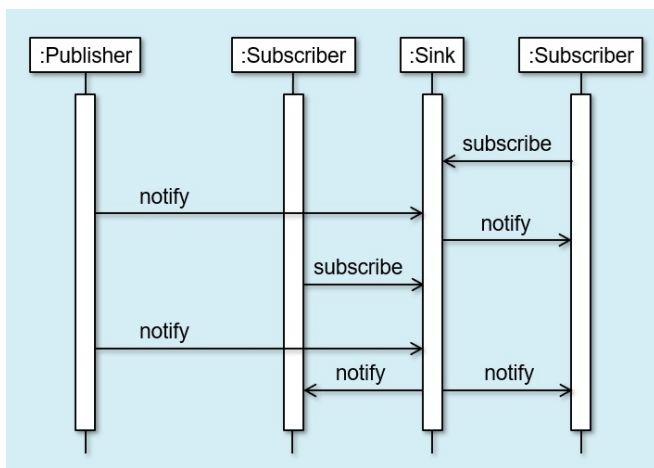


Fig. 7.12 COM+ loosely-coupled events communication model

The event class is a means of ensuring greater anonymity and flexibility: subscribers cannot see the publisher, and the publisher does not know its subscribers. The arrangement enables different publishers to be substituted without subscribers being aware, e.g., alternative news feeds being used as a source of financial events.

In UML component diagrams we make EventClass components explicit, and use the `{EventClass = true}` tagged value to express that a component plays this role. We can represent the delegation of messages by the `<<delegate>>` stereotype (Figure 7.13). Multiple sinks may be associated with each publisher, to achieve scalability to large numbers of subscribers.

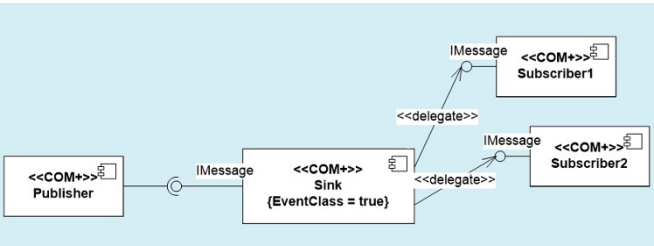


Fig. 7.13 Loosely-coupled events architecture

7.7 Queued components service

This service addresses the problem of distributed enterprise systems where nodes of the network may sometimes be disconnected (e.g., because they are hosted on mobile devices), but some overall business process still needs to be carried out by their interaction. The problem is related to the connectivity management problem for individual mobile devices, addressed by the online/offline DAO pattern of Chapter 6, but is more general, because it involves the management of multiple potentially two-way communications.

In COM+ a component can be annotated as “Queued”, which means that COM+ provides a queue to buffer messages. Queues may be necessary on both the client (sender) and server (receiver) side of a communication. On the client side of a distributed communication there is a *Recorder* component which enqueues request messages in an outgoing queue and acts as a proxy for the server (Figure 7.14). Queued messages are sent on when a connection to the server is available. On the server side a *Listener* component receives the messages and passes these to a *Player* component which allocates requests to the correct server.

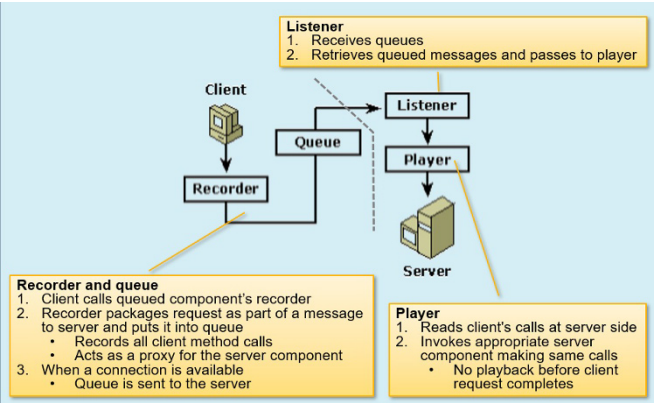


Fig. 7.14 Queued components in COM+

In UML component notation we abstract the *Recorder*, *Listener* and *Player*, and instead indicate a queued connection via a delegation connector and specialised tags that indicate queued messaging (Figure 7.15).

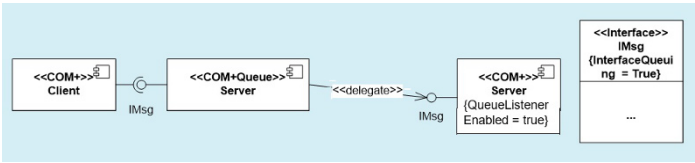


Fig. 7.15 Queued components in UML notation

This architecture handles the case where messages may be delayed because of poor network connections, but does not ensure that lost messages are detected, nor that messages are delivered in the order they were sent. More sophisticated communication management is required to achieve these QoS requirements.

7.8 COM+ services as basic enterprise styles

Each COM+ service can be regarded as a platform-specific architectural style or pattern, consisting of component kinds, constraints and uses (Figure 7.16).

Service	Component kinds, constraints	Use
Transactions	Business logic components	Recovering from errors Ensuring business processes complete successfully as a group
Object pooling	• Client components • Server (pooled) components • No state or thread affinity	Whenever server component is fat and expensive to create for clients
Just in time activation	Client and server (JIT activated) components	Whenever server is thin but often idle
Queued Components	Client, server components. Queued interface cannot return values nor take input by reference	Sometimes disconnected client components
Loosely coupled events	Publisher, subscribers, event component Event component does not implement its interface, shares the same provided interface as server	Anonymous communication Decoupling Need to ensure open scalability to new subscribers

Fig. 7.16 Application services as architectural styles

Typically a complex system will involve a combination of enterprise services to achieve a range of goals. The styles are built for integration specifically within an N-tiered architecture, and particularly the business tier. We have to take into account that combinations or configurations of services may

negatively impact performance, reliability, etc. For example, transactional behaviour carries additional processing overhead, as does loosely-coupled messaging. Object pooling cannot be used with stateful components such as shopping carts, which retain client state between calls (we do not want to share our shopping cart with random other customers). Most complex enterprise styles also involve interaction with other tiers of the architecture

Summary

This chapter has looked at the concept of application servers, and in particular the enterprise services provided by the .NET platform. Services for transactions, object pooling, just-in-time activation, queued components and loosely-coupled messaging have been described.

Exercises

1

Consider the investment management system shown in Figure 7.17. The system manages investments by an investor in potentially three separate financial products: shares, bank savings and bullion (gold or silver). The *InvestMan* interface includes an operation *withdraw* which permits the investor to withdraw completely their investments in all three products in one step.

Identify a suitable COM+ service to implement this system and explain the rationale for using the service.

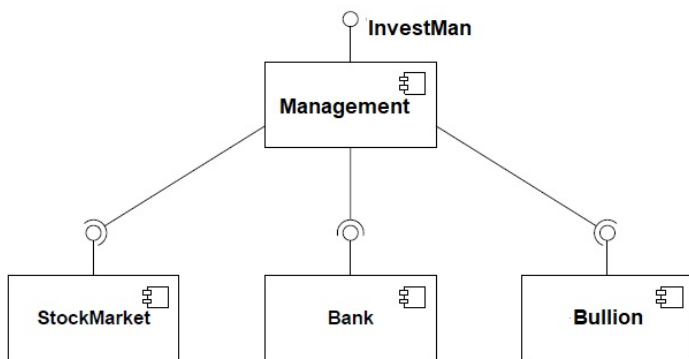


Fig. 7.17 Investment management system

2

Redraw the architecture of Figure 7.17 using the appropriate COM+ service which you identified in question 1.

3

Explain which of the following components can be pooled in an object pool:

- A customer *Wishlist* holding a set of items which a customer would like to purchase when they become available on a shopping site.
- A connection to a remote datasource.
- A stateless validation bean whose only operation validates a group of data items supplied as the operation parameters.

4

Select a suitable COM+ service for the following system, and draw the architecture diagram of the system.

The system is a sales/order system for a home improvements service, where salespersons may confirm or change orders using remote devices at customer sites or in transit. The main datastore and business processing is carried out at the company office.

5

A financial analysis system needs to receive asynchronous notifications from possibly different market data sources, regarding critical market events. The system does not need to otherwise interact with the data sources, and should not depend on the specific source system(s).

Identify a suitable .NET service for this system, and draw the architecture for the system using this service.



Chapter 8

Web Application and Enterprise System Architectures

Abstract This chapter will explain some fundamental enterprise architecture styles and patterns, and show how these can be applied to organise web application and EIS architectures. We will also explain the key concepts of Java EE/Jakarta, the main alternative EIS platform to .NET.

8.1 Enterprise system architectures

EIS architectures typically have up to five tiers (Figure 8.1):

- Client tier: UI components for direct interaction with clients.
- Presentation tier: UI logic components, presenters, controllers and helpers.
- Business tier: business functions and logical representations of business data – interactors, entities and auxiliary elements such as value objects.
- Integration tier: gateways and other components mediating between the business tier and data sources.
- Resource tier: databases, web services, etc.

An example is shown in Figure 8.2.

Each enterprise style or pattern typically addresses concerns specific to a particular tier of an enterprise system. Enterprise styles try to maintain a clear separation of concerns and responsibilities between the tiers. This improves modularity, maintainability and testability. Some styles also concern the efficiency of communication between tiers and the reduction of resource usage.

We will look at the following EIS styles, based on the design patterns of (Deepak Alur et al., Core J2EE Patterns, [4]):

Presentation tier styles: View helper; Front controller; Intercepting filter; Composite view.

Business logic tier styles: Session facade.

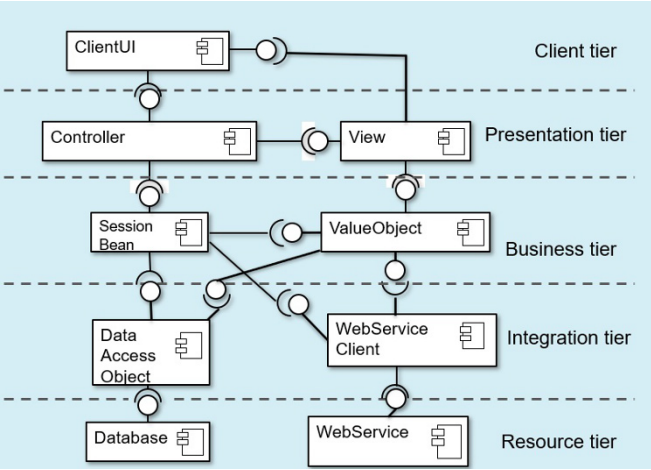


Fig. 8.1 Schematic five-tier EIS architecture

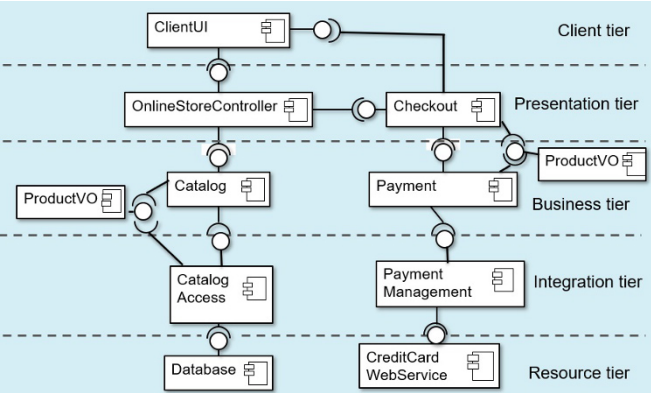


Fig. 8.2 Example five-tier EIS architecture

Integration tier styles: Service locator; Extract-transform-load (ETL).

8.2 Java Enterprise Edition

Java EE extends Java standard edition (Java SE) in order to define facilities for distributed enterprise applications. The front end of an EIS can be defined by Java EE web application facilities including Servlets, Java Server Pages (JSPs), and Java Server Faces (JSF). Enterprise Java Beans (EJBs) express the business logic of the application – these are annotated specialised Java classes, analogous to managed COM+ components using annotated C# for

.NET. The Java Persistence API manages the persistence of business data, and there are web service facilities for SOAP-based or RESTful web services. Figure 8.3 compares the Java EE and .NET elements.

.NET	Java EE
COM+ components	EJB – stateless, entity and stateful beans
Transactions	JTS
Loosely coupled events	JMS
Queued components	Can be implemented through JMS
Object pooling	Instance pooling
JIT	Activation/Passivate (on stateful beans)
ASP.NET	JSP
.NET web services	Java web services

Fig. 8.3 Java EE versus .NET for EIS

8.2.1 Java EE Components: Servlets

Servlets are specialised Java classes which process HTTP GET, PUT, POST and DELETE requests via corresponding methods *doGet*, *doPost*, *doPut*, *doDelete*. They also have initiation and destruction operations *init* and *destroy* for lifecycle management. Servlets are typically used as presentation tier Controller components in an MVC architecture, e.g., as a Front Controller (Section 8.5). Servlets can delegate requests to other servlets or to JSPs, via the *dispatch* service. In fact, JSPs are compiled to servlets by the EE platform. Figure 8.4 shows an example servlet.

8.2.2 Java EE Components: Java Server Pages (JSPs)

JSPs generate web page content, including complete web pages, for display on the user’s browser, in response to requests. They play the same role as ASPs in .NET. JSPs combine literal (static) HTML text and dynamically-produced content (e.g., using data from a database or business tier component) using JSP commands and Java code. Figure 8.5 shows an example of a JSP. Text in red shows the specialised JSP tags for transferring request data to a *welcomeHelper* bean (the first line) and retrieving data from it (the *jsp: getProperty* tag). Blue text, within the `<%` and `%>` delimiters, is Java code, and other text is static HTML.

```

public class PropertySystemController extends HttpServlet
{ public void init(ServletConfig cf) throws ServletException
  { super.init(cf); }

  public void doGet(HttpServletRequest rq, HttpServletResponse rs)
  throws ServletException, IOException
  { // carry out any common security/authentication checks

    String regC = rq.getParameter("Register");
    if (regC != null)
    { // pass request to register servlet
      dispatch(rq, rs, "RegisterUserServlet");
      return;
    }
    String editC = rq.getParameter("Edit");
    if (editC != null)
    { // pass request to edit servlet
      dispatch(rq, rs, "EditUserServlet");
      return;
    }
    ...
  }
}

```

Example of online estate agent system, enabling users to register & to edit their details

Fig. 8.4 Servlet example

```

<jsp:useBean id="welcomeHelper" scope="request"
class="beans.WelcomeHelper"/>

<html>
<head><title>Welcome page</title></head>
<body> <h1>Welcome page</h1>

<% if (welcomeHelper.nameExists()) { %>
<center><h3>Welcome
<jsp:getProperty name="welcomeHelper" property="name"/>
</h3></center>
<% } %>

<center><h4>Glad that you are visiting our site!</h4></center>
</body>
</html>

```

Fig. 8.5 Example of a JSP

Typically, JSPs are used as View presentation-tier components in an MVC architecture. In terms of the clean architecture they are presenters, and they convert model (business) data into a viewable form as HTML for the web UI of the system. Controllers in an MVC structure select which JSP to activate, depending on the UI logic and the current state of the user interaction session. JSPs can utilise auxiliary beans (plain Java classes) to perform computational tasks/store local data. JSPs can also be pagelets (generating partial web content to be assembled into complete pages by other views) or can act as controllers themselves.

8.2.3 Java EE Components: Enterprise JavaBeans (EJBs)

These are the core business components of a Java EE system. They correspond to managed COM+ components in the .NET platform. They reside in the business tier of an EIS.

The different forms of EJB are:

- *Session Beans* – business functionality components, which can be Stateless, Statefull or Singleton session beans. They can have both Local or Remote interfaces for use by clients on the same or a different JVM.
- *Message-driven Beans* – business objects driven by events (messages).
- *Entity Beans* – Java classes for business entities, these provide OO abstractions of database data.

Stateless session beans are business components dedicated to a single client, and do not retain state between method calls. They are suitable for one-off operations, such as sending a confirmation email. They can be pooled in object pools.

Statefull session beans are also dedicated to a single client, but retain state related to that client, so cannot be pooled. They are suitable for user sessions involving a sequence of operations as part of a workflow/interaction (e.g., a shopping cart in an e-commerce site).

A key principle of EJB programming is that these business-tier components should operate on business entity data as if these entity instances were local objects of plain Java classes, transparently of their actual implementation in a possibly remote persistent data storage technology. The Java EE platform manages this transparent access via the persistence API.

8.3 Web application architectures

Web applications can be regarded as simplified EIS which have a restricted set of functionalities focussed on a relatively few closely-related use cases, and which have few actors and low QoS requirements compared to general EIS. They are simple business systems which utilise web pages executing in a client browser as their UI. For such systems the emphasis is upon the client and presentation tiers: the client tier consists of static or dynamic web pages together with scripts, style sheets and other resources to control the appearance and behaviour of the pages. The presentation tier is usually organised with the MVC pattern, and contains a single controller (or multiple controllers – one for each actor) together with views for each web page (Figure 8.6). Each view may have a helper bean to provide supporting functionality.

There are two main choices for the architecture of the web application client and presentation tiers:

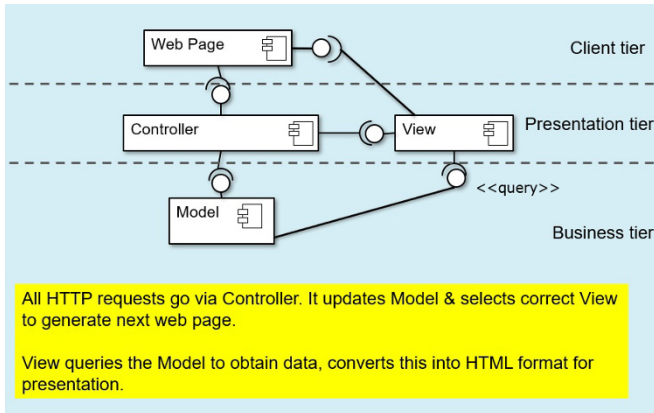


Fig. 8.6 Conventional web application architecture

1. *Traditional web interaction:* entire pages are loaded into the client browser as a result of application processing in response to user actions. The controller receives client requests and activates the appropriate view to generate the response web page.

An example of this style for an online shopping site for books is shown in Figure 8.7.

2. *Interactive pages using asynchronous JavaScript and XML (AJAX):* each page can be divided into parts, with the parts being updated individually by the server, independently of user interaction with other parts of the page.

In this case individual views manage individual pages, including asynchronous updates of the pages (Figure 8.8). A classic example of AJAX processing is the autocomplete form field, where typing in a field triggers queries to the server side to find relevant items whose names/identifiers extend the partially-typed field data. Because this only involves queries of model data, the interaction can be handled by a View in Figure 8.8, without the need to invoke a controller.

This approach is appropriate for increasing the responsiveness of the application.

In either case, the usability of the web application is an important QoS property, which means that the web app designer should try to simplify the user interaction where possible by reducing the number of steps needed to access required functionality, and by providing appropriate feedback in the case of invalid actions or errors. The layout of web pages and appropriate use of graphics and menus are also significant for usability. In general, the commands and actions available on a page should be clearly visible as commands, immediately distinguishable from other text or graphics. A consistent style of presenting commands should be used (e.g., as buttons or menu options), and

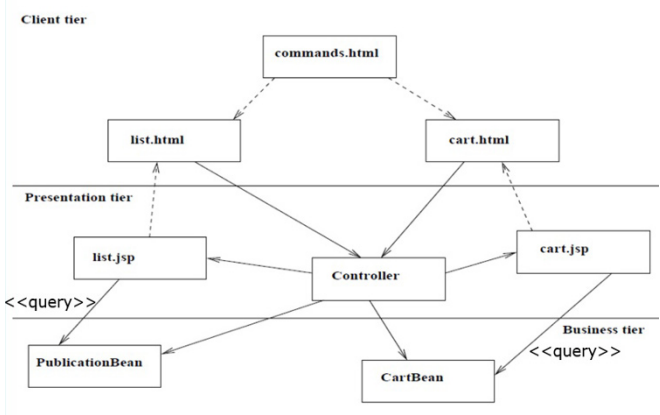


Fig. 8.7 Online bookstore architecture

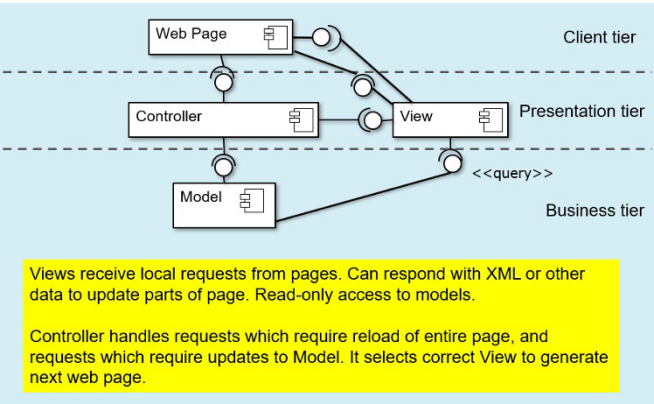


Fig. 8.8 AJAX web application architecture

valid alternative commands should be equally visible. Deliberately making some alternatives difficult to access compared to others has negative impact on usability (such as the common practice of making rejection of cookies an obscured option or elaborate many-step process compared to selection of a single “Accept all cookies” button). Similarly, navigation options should be clearly visible as such, and navigation simplified and rationalised where possible.

8.3.1 Client tier

The client tier of a web application consists of screens (web pages in HTML format) together with auxiliary files such as style sheets (CSS files) and JavaScript script files.

HTML and CSS files are not processing components, they are simply used to display server-generated information and to initiate requests to the server side. In this respect they are similar to Android XML-specified mobile app screens (Chapter 6). Some simple UI effects, such as tabbed screens, can be defined using HTML and CSS, but not any computations¹. A client tier without client side processing is referred to as a *thin client*.

Alternatively, client-side processing can be defined, using JavaScript scripts. JavaScript can be used to check the consistency of web form data before submission to the server, and to perform local computations, e.g., to calculate the sum of a series of numeric form fields. The client tier components are then known as *fat clients*.

An example of client-side processing using JavaScript is shown in Figure 8.9.

Quantify impact measures for outcomes

Number of families reporting increase in income:

Total number of families in client set:

Benefit ratio:

Fig. 8.9 Client-side processing example

This web page is produced by the following HTML, which uses an embedded JavaScript function *compute* to perform the computation. The script reads the first two form fields, converts their values to numbers, and writes the ratio of the numbers to the third form field:

```
<!DOCTYPE html>
<html><head>
<title>Computation example</title>

<script language = "JavaScript">
function compute()
{ var x1 = Number(document.form4.field1.value);
```

¹ For more details on the use of CSS, see our course <https://agilemde.co.uk/cllweblong.pdf>

```

    var x2 = Number(document.form4.field2.value);
    document.getElementById("field3").innerHTML = x1/x2;
}
</script>
</head>

<body id="main_body">
<div id="main_container">

<h1>Quantify impact measures for outcomes</h1>

<form name = "form4" action = ""><p>

<table title="SRQI computation">
<tr><th width="200px"> </th>
    <th width="400px"> </th></tr>

<tr><td> <strong>Number of families reporting increase in income:
</strong></td><td>
<input name = "field1" type = "text" onchange = "compute()"></td></tr>

<tr><td> <strong>Total number of families in client set:</strong></td><td>
<input name = "field2" type = "text" onchange = "compute()"></td></tr>

<tr><td> <strong>Benefit ratio:</strong></td><td id = "field3">0</td></tr>
<p>

<tr><td> </td><td> <input type = "button"
    value = "Calculate" onclick = "compute()"></td></tr>

</table>
</form>
</div></body></html>

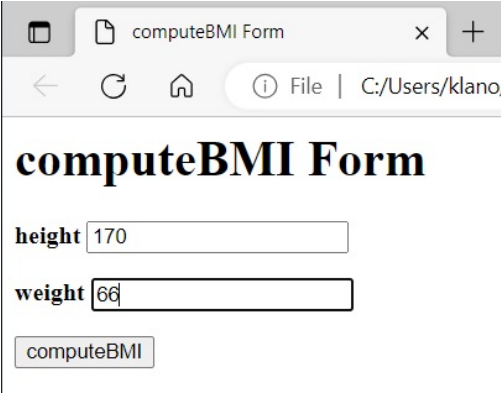
```

The function is triggered by a change to either input field, or by an explicit button press of the *Calculate* button.

Using client-side processing improves application responsiveness and potentially reduces server load and the amount of information transferred between client and server. However, different browsers may execute scripts in different ways, or scripts may be blocked, thus there is not a guarantee that this functionality will be executed. AJAX processing may involve frequent exchanges of information between client and server (as in the autocomplete form field example), increasing network load.

Another example of a web application could be a web-based implementation of the health app from Section 6.1.3. This could have a web form for each use case, as shown in Figure 8.10.

Fig. 8.10 Health web app example



8.3.2 Presentation tier

On the server side, controller components are normally implemented as servlets in a Java EE system, whilst views are written in JSP. However it is possible for both to be written as JSPs. Table 8.1 lists some of the possible options for server-side web application components, depending on the technological platform being used.

Table 8.1 Server-side web application components

<i>Platform</i>	<i>Controller</i>	<i>View</i>	<i>Helper</i>
JavaEE	Servlet or JSP	JSP	Java class
.NET	ASP or C# IHttpHandler	ASP	C# class
Python	Python program	Python program producing HTML	Python program

As an example, the controller and view/presenter for the app4 *computeBMI* use case could be combined in a single JSP:

```
<jsp:useBean id="bean" scope="request"
  class="app4.computeBMIBean"/>

<jsp:setProperty name="bean" property="height" param="height"/>
<jsp:setProperty name="bean" property="weight" param="weight"/>

<html>
<head><title>computeBMI</title></head>
<body>

<h1>computeBMI</h1>
<% if (bean.iscomputeBMIErrror())
{ %> <h2>Error in data: <%= bean.errors() %></h2>
<h2>Press Back to re-enter</h2> <% }
else { %> <p><strong> Result = </strong> <%= bean.computeBMI()%> </p>
```

```

<h2>computeBMI performed</h2> <% } %>
<hr>

<%@ include file="index.html" %>
</body>
</html>

```

The *computeBMIBean* class serves as a validation bean to check the *computeBMI* parameters, and also as a business delegate to invoke the business tier operation that implements the use case *computeMBI*. The result is displayed on the web page generated by the JSP.

8.3.3 Business tier

The business tier for a web app would typically use session beans, although plain Java classes can be sufficient for simple applications.

For app4, a plain Java class is used. As with the mobile app implementations of this system, the interactor implementing the use cases is a singleton class called *ModelFacade*:

```

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ModelFacade
{ static ModelFacade instance = null;

    public static ModelFacade getInstance()
    { if (instance == null)
      { instance = new ModelFacade(); }
      return instance;
    }

    private ModelFacade() { }

    public double computeBMI(double height, double weight)
    { double result ;
      result = weight / (height * height);
      return result;
    }

    public double calorieCount(Gender gender, Exercise exercise, double time)
    { double result ;

      double factor = 0.0;
      factor = 1;
      if (exercise == Exercise.walking)
      {
        factor = 100;

```

```
    }
    else {
        if (exercise == Exercise.running)
        {
            factor = 300;
        }
        else {
            if (exercise == Exercise.jogging)
            {
                factor = 200;
            }
            else {
                factor = 250;
            }
        }
        result = factor * (time / 60);
        return result;
    }
}
```

8.3.4 Web application frameworks

A number of web application frameworks have been defined to simplify the development of web applications. Examples based on Java EE include Java Server Faces (JSF), Spring² and Struts³. For Python there are Flask⁴ and Django⁵. These all use some variation on an MVC-style architecture, for example Django defines a *Model-View-Template (MVT)* architecture:

- Model: manages persistent data and database interaction.
- View: request handling and coordinating the processing of requests – actually a Controller role in the conventional MVC pattern.
- Template: static and dynamic content of a web page – actually the View role in terms of MVC.

As with the use of mobile frameworks and MBaaS, the application developer has to weigh the convenience of using a framework against the potential costs of system migration if the framework becomes obsolete or proves to be inappropriate in the long term.

² spring.io/projects/spring-framework

³ struts.apache.org

⁴ palletsprojects.com/p/flask

⁵ www.djangoproject.com

8.4 View Helper style

This style addresses the problem of duplication of code between different view components, such as common authentication and presentation processing between views which have similar functionalities. The pattern factors out the common elements into an auxiliary component which is called by the original views. It does not itself directly interact with users.

Figure 8.11 shows the general structure of the style. The view helper is a presentation tier component, and produces common content which is then used by specific presenters.

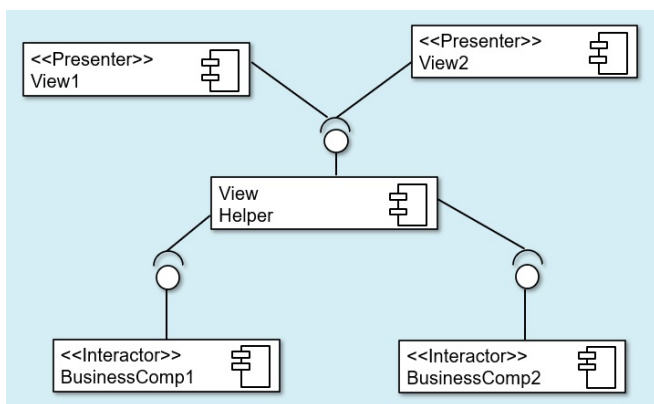


Fig. 8.11 View Helper architectural style

In .NET a view helper could be defined as an ASP view, and in Java EE defined as a JSP view. In either case it would be a ‘pagelet’, a presenter which produces partial content which will be used by other views to assemble UI screens.

For example, a web-based college time-tabling system could have different but similar views for lecturers and students. Each uses a business component (such as a COM+ component) for authentication, and each uses another business component to access time-tabling information. Each view includes its own code to invoke authentication actions and then create a presentation (Figure 8.12). Common system logic is therefore duplicated over multiple views in this system, which can cause maintenance costs and bugs.

The system can be refactored using View helper, with both lecturer and student views sharing the same helper. The view helper can retrieve the model states (timetable data) and produce content from it for presentation to different users by user-specific views (Figure 8.13).

An alternative to using pagelet views is to use ‘bean’ classes in conventional programming languages to supply common services to presentation tier components. E.g.: an email validator `ValidatorBean.java` used by both a

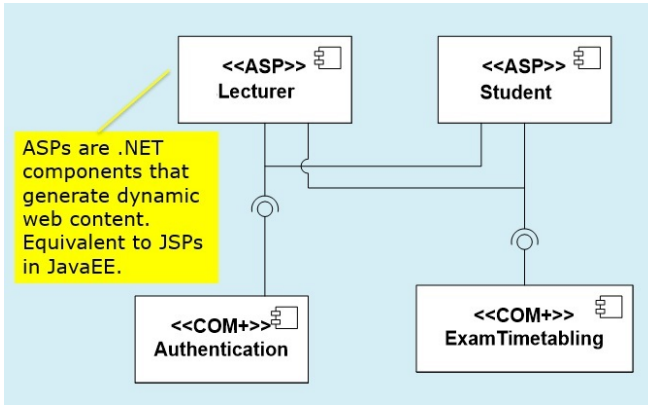


Fig. 8.12 View Helper example – problematic structure

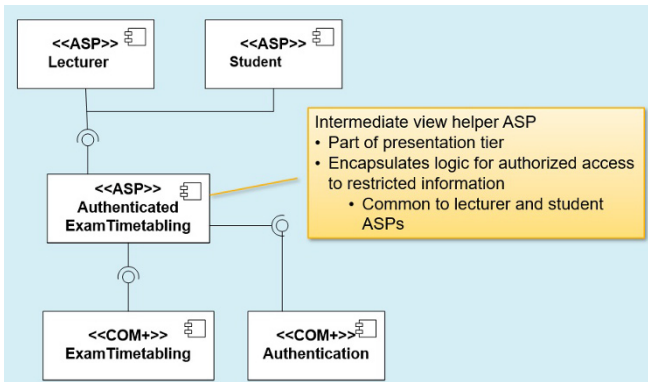


Fig. 8.13 View Helper solution

Login.jsp and Signup.jsp. This approach can be used if there is common logic which does not involve producing UI content.

8.5 Front Controller style

This style defines a controller component as a unique entry point to a system, with this controller handling all requests to the system. No other component can receive user requests. The front controller centralises common authentication processing which otherwise would need to be duplicated in every component that could receive user requests. It may delegate requests to suitable subcontrollers, and choose an appropriate view to display request responses. It can handle errors and manage selection of content creation strategies (e.g.,

dependent on the target device). It can globally manage the navigation structure of the UI (i.e., to perform as a navigation controller).

For example, an on-line shop could be organised so that each view is itself directly accessible via a URL to receive user requests (Figure 8.14). All these views will usually share some behaviour, such as logging of the user progress through the shop, but there is no centralised point for handling of this common functionality. Navigation logic is also scattered across all views, so it is difficult to update the global navigation structure and maintain consistency.

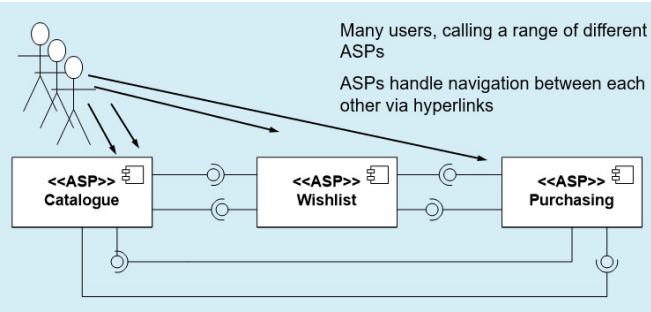


Fig. 8.14 Front Controller – example problem

A front controller solution to this case is shown in Figure 8.15. Here the user logging and navigation logic is centralised in the front controller, reducing overall code size and duplication.

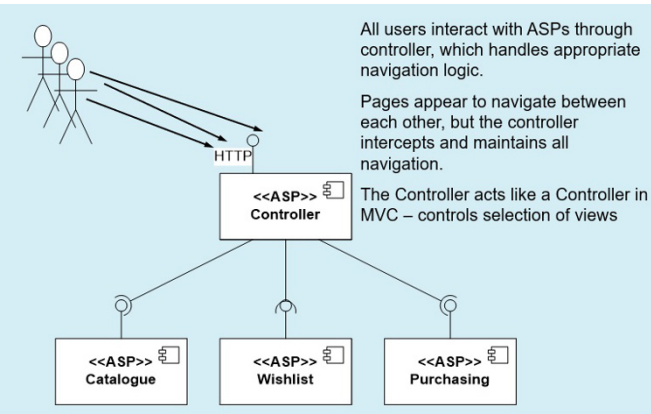


Fig. 8.15 Front Controller – example solution

8.6 Intercepting Filter style

Intercepting Filter is a version of the Pipes-and-filters style (Chapter 5), which defines pluggable filters for pre and post-processing of requests or responses, e.g.: for security checking, authentication, session management, etc. The style provides a flexible and configurable means to add filtering, and pre and post processing to request handling (Figure 8.16). Client requests to an EIS are typically checked before being processed, for example to check if the client's IP address is from a trusted network, or if the client has a valid session in the system, or if the client's browser is supported by the application. The pattern uses a chain of filters to carry out successive tests. Each individual check is the responsibility of an individual module in the chain. The pattern is related to the classic Chain of Responsibility object-oriented design pattern [31].

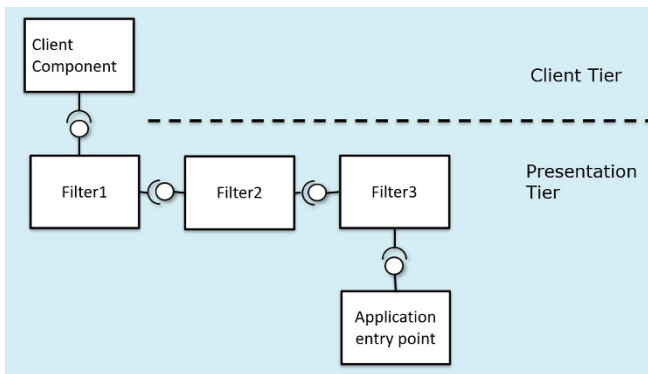


Fig. 8.16 Intercepting Filter structure

Figure 8.17 shows an example of the style.

8.7 Composite View style

This style involves the composition of a view out of parts (subviews). Complex web pages are often built out of multiple parts, such as a navigation section, news section, etc. Some parts may be dynamically updated using a technique such as AJAX, whilst other parts are static. Managing such composite page layout and content in a single global view provides poor flexibility. Instead, the pattern allows views to be flexibly composed as recursive structures of objects (Figure 8.18).

The elements of the pattern are:

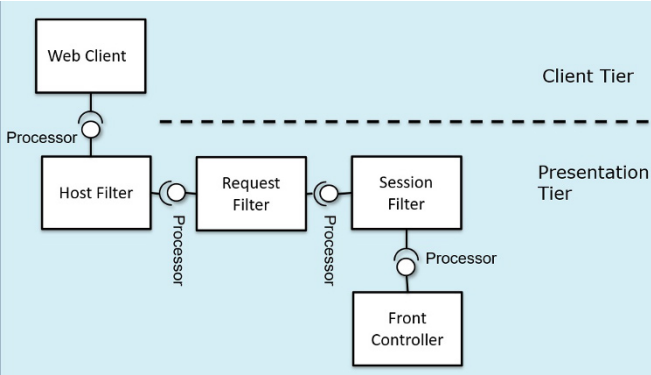


Fig. 8.17 Intercepting Filter example

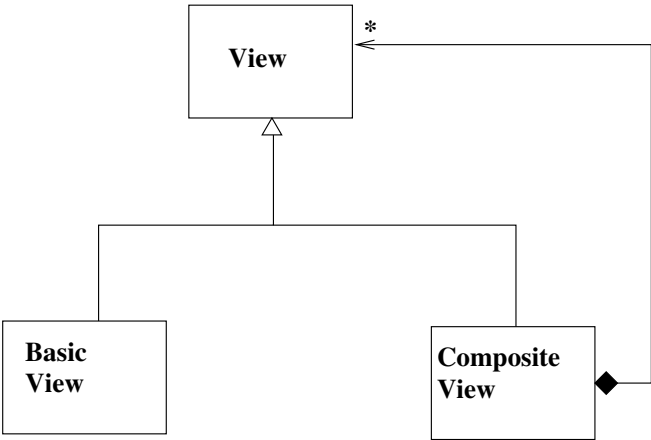


Fig. 8.18 Composite view structure

- *View*: a general view, either basic or composite.
- *Basic View*: a self-contained view, such as a web page division or section, which is not further sub-divided.
- *Composite View*: a view that is an aggregate of multiple views. Its parts can themselves be composite.

An example of this pattern in Java could be the `<jsp:include page = "subview.jsp">` tag in JSP, used to include subviews within a composite JSP page. Other approaches include custom JSP tags and XSLT (if content data is stored as XML).

8.8 Session Facade style

This style is used when there is complex interaction between the presentation tier and business tier, in particular if a presentation-tier component requires access to multiple business objects (Figure 8.19). Such a situation should be corrected because it involves tight coupling of presentation and business-logic tier, and means that the presentation tier contains detailed knowledge of the business tier structure – which makes it costly to change business tier organisation as this will also impact the presentation tier. It also means that the presentation tier is partly implementing business logic, which is an incorrect separation of concerns. In terms of performance it could cause a high network load if the two tiers are remote from each other, as will often be the case. The network load arises because there are too many method invocations between the tiers, caused by the poor allocation of responsibilities.

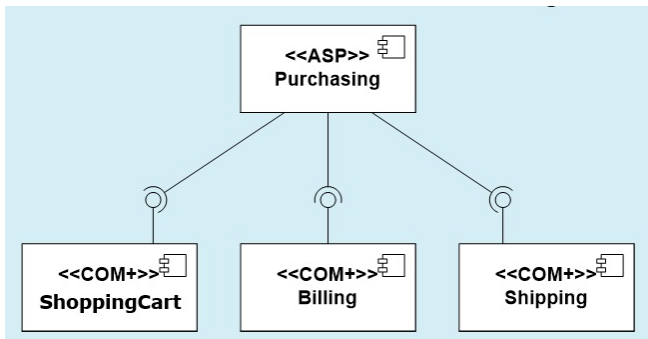


Fig. 8.19 Session Facade – example problem

The Session Facade style provides a solution to this problem by introducing an intermediate business-logic component called a *session facade* (c.f. Java EE *Session bean*) which provides a simplified interface of the business tier for clients. The session facade manages business logic and the interaction of different business objects in business workflows (Figure 8.20). Session beans correspond to system components in the Cheesman/Daniels approach of Chapter 3, they implement the system steps of use cases. They are the EIS equivalent of the Model Facade components for mobile architectures (Chapter 6). They are interactors in the clean architecture terminology.

The session facade has the advantages that it helps to separate presentation and business-logic concerns, and centralises presentation tier/business tier interactions. This can lead to improved maintainability and reduced resource usage by reducing network usage, with a reduced number of components directly accessed over the network.

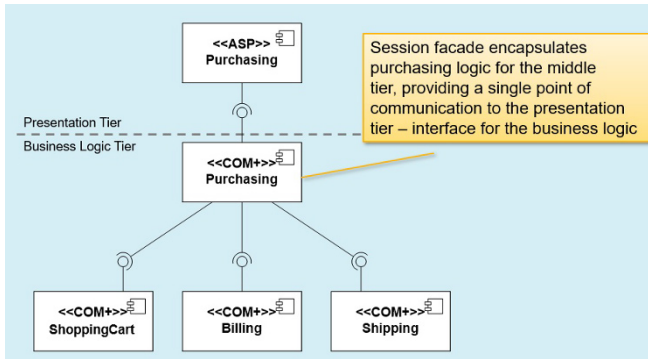


Fig. 8.20 Session Facade – example solution

8.9 Service Locator style

This pattern manages the details of service/resource lookup, in particular the lookup of external services such as web services (Chapter 9) on behalf of business components. Although (Deepak Alur et al.: Core J2EE Patterns) place this pattern in the business tier, we consider that a service locator is a gateway component, and hence belongs in the integration tier. It hides from the business tier the details of how to obtain a service, and may transfer the obtained service to a specific DAO for detailed service management. It is analogous to the Broker SOA pattern for web services (Chapter 9). The Android Intent mechanism (Chapter 6) is also related to this style, although with intents, the Android platform performs the service location on behalf of the app.

8.10 Extract-transform-load (ETL) style

This pattern concerns the batch processing of source datasets to (i) extract analytics information for business purposes, and/or (ii) to translate the data into a new target storage format. Figure 8.21 shows the schematic structure of this pattern. An example would be analysing e-commerce customer orders data in order to identify high-spending customers over a given time period. This involves scanning through one or more source tables in the e-commerce app relational database (the Extract step), performing the analytics computation (Transform step) and then populating a temporary table with the analysis results (Load step). Another example could be extracting text data of banking transactions from a CSV file, performing some simple data conversions such as mapping strings representing dates into Java *Date* objects, and storing the resulting data in a relational database. Other ETL applica-

tions are data migration projects recovering datasets from legacy data storage formats/technologies and loading them into modern ones.

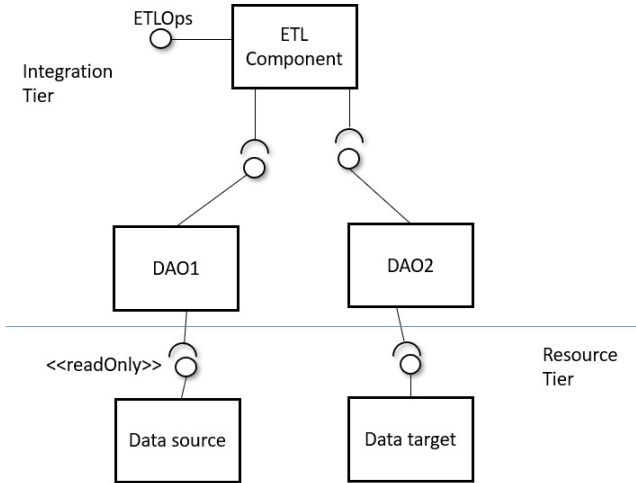


Fig. 8.21 ETL style

The pattern has the advantage of modularising the data processing steps, and providing a business interface for management of technology-specific data processing. Trustworthiness is an important QoS property which this pattern can help to ensure, in particular to ensure that there is a valid semantic relationship (semantic invariant) between the produced target data and original source data.

8.11 Using multiple styles

The above EIS styles can be naturally combined, for example an intercepting filter could have a front controller component as the final destination of its filter chain, with this controller playing the role of an MVC controller, where the model part is facaded by a session facade (Figure 8.22). The interoperation of styles is facilitated because they mainly address orthogonal concerns.

An example of combining styles could be a bank system EIS (Figure 8.23). The staff interface of this EIS supports operations to create a customer record, create accounts, and to link and unlink accounts from customers. An MVC pattern is used for the presentation tier, and the business tier has session beans for staff operations, which process business data. The presentation

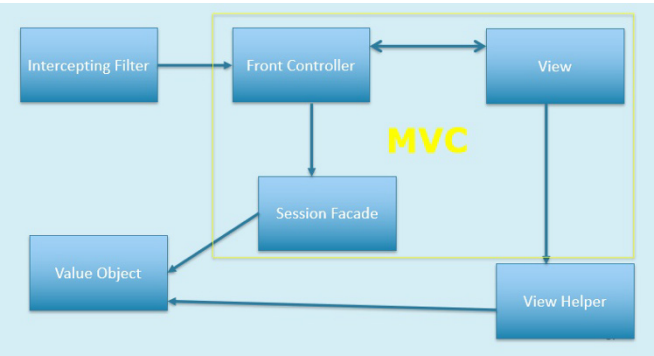


Fig. 8.22 Using multiple styles

tier view receives value objects for this business data. The integration tier populates value objects from database data, and uses these to update the database.

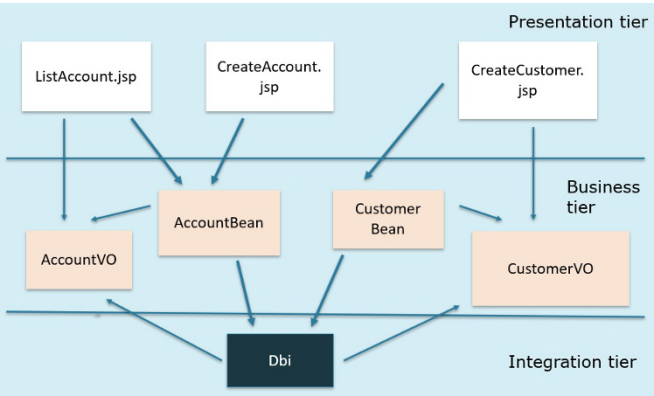


Fig. 8.23 Using multiple styles example

Summary

This chapter has considered in detail relevant architectures and architectural patterns for web applications and EIS.

Table 8.2 summarises which quality of service properties are primarily addressed by the patterns we have considered for mobile, web and enterprise applications. MVC denotes all event-based patterns based on MVC, such as VIPER and MVP.

Table 8.2 Quality-of-service properties of mobile/EIS patterns

<i>QoS property</i>	<i>Relevant patterns</i>
<i>Availability</i>	Online/offline DAO; Queued components
<i>Efficiency/</i>	Value object; Object pooling; JiT activation;
<i>Performance</i>	Model/session facade
<i>Evolvability/</i>	Model/session facade; DAO; MVC; MVVM; Service locator
<i>Maintainability</i>	
<i>Fault tolerance</i>	Transactions
<i>Responsiveness</i>	Service activator
<i>Scalability</i>	Object pooling;
	Loosely-coupled events
<i>Security</i>	Front controller; Intercepting filter;
	View helper (validation bean)
<i>Transactionality</i>	Transactions
<i>Trustworthiness</i>	ETL; MVVM;
	View helper (validation bean)
<i>Usability</i>	MVVM; MVC; Composite view;
	Service activator

Further Reading

- Martin Fowler et al.: Patterns of Enterprise Application Architecture. Addison-Wesley, 2003.
- Deepak Alur et al.: Core J2EE Patterns. 2nd edition, Prentice Hall, 2003.
- Adam Bien: Real World Java EE Patterns, press.adam-bien.com, 2012.

Exercises

1

A book wholesaler system holds data on books, orders and customers (Figure 8.24). There are two different categories of users: (i) customers, who can only view their own orders, and initiate or cancel orders for specific books, and (ii) staff, who can add/remove customers and books, and view and delete the orders of any customer.

What business tier components would be appropriate for this system?

1. A session bean which handles all use cases for all actors, and entity beans for orders, books and customers.
2. Separate session beans for customer use cases and staff use cases, and entity beans for orders, books, staff and customers.
3. A session bean which handles all use cases for all actors, and entity beans for orders, books, staff and customers.

4. Separate session beans for customer use cases and staff use cases, and entity beans for orders, books and users (combining staff and customer representations).

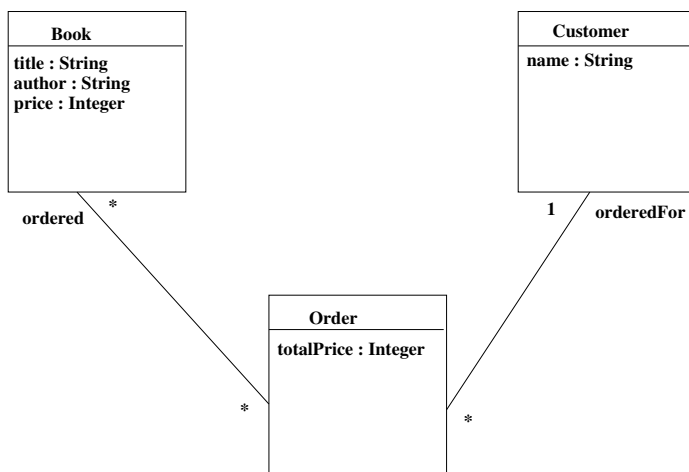


Fig. 8.24 Class diagram of book wholesaler system

2

Explain how the Value object pattern can help to assure the following non-functional quality of service properties.

1. Performance.
2. Responsiveness.
3. Maintainability.
4. Portability/platform-independence.

3

Identify suitable EIS patterns for the following situations, and explain your choice of patterns:

- A business application which can be used by company representatives on mobile devices at remote locations.
- An online banking application which may need to process over 10,000 simultaneous requests for account information.

4

Define the business, integration and resource tier architecture of a system for managing student records in a college administration system, which supports the recording of student module results and final degree awards, in addition to personal information. Management of personal and academic student data is handled by separate components, and these are also stored in separate databases.

Projects

1

Design and implement a web application for simple business budgeting: this enables the entry of *Asset* items such as investments, property, equipment, cash, pending payments, etc., and *Liability* items such as debts, invoices to be paid, due taxes, due salaries, rent, etc. The application should sum the items in each category and calculate

$$Equity = Assets - Liabilities$$

and

$$Leverage = Assets / Equity$$

2

Implement the app4 application described in Section 8.3 above as a purely client-side web app.

3

Design and implement an EIS which provides operations for bank staff to manage customer accounts. The system stores customer and account information, and provides operations to:

- Create a customer.
- Create an account for a customer.
- Add a customer to an account.
- Remove a customer from an account.
- Display the balance of a given account.
- Display the total balance of all accounts associated with a customer.

Customer data includes a unique customer identifier, and the customer name and address. Account data includes the unique account identifier, account name, and current balance of the account. A boolean flag indicates if the account is overdrawn, i.e., the balance is < 0 .



Chapter 9

Service-oriented Architectures

Abstract This chapter will look at architectural design in the context of cloud-based services, web services and microservices, which can all be considered to be forms of *service-oriented architecture* (SOA). We will describe the purpose and functionalities of cloud services, web services, microservices and SOA, and use UML component diagrams to specify SOA.

9.1 The cloud and cloud services

In Chapters 7 and 8 we described how application servers provide transparency for software components. Cloud providers enable a similar kind of transparency for hardware resources, whereby hardware resources are provided as a commodity for use by customers, on a rental basis. The cloud provider handles all the details of allocation of actual hardware to meet customer requirements, and the management of that hardware. This means that customers do not need to know what hardware their software is running on, and do not need to know where that hardware is.

From the customer viewpoint this gives increased flexibility, to only pay for the actual resources that they need, it also removes hardware maintenance costs, and the need to employ hardware support staff. From a resource consumption point of view, the centralised shared use of cloud centre resources should introduce economies of scale and reduce energy waste (i.e., compared to many individual companies running separate under-utilised machines).

Cloud service characteristics (QoS properties) include:

- Scalability – Increases in load/number of users can be handled by increasing the number of (virtual) servers running new copies of software.
- Elasticity – It is also possible to scale down resources to match reduced demand.
- Resilience – Toleration of server failures by dynamic re-allocation of servers to tasks.

Types of cloud services include Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS) and Mobile Backend-as-a-Service (MBaaS).

Cloud use by business has been progressively increasing, for example, 41% of EU businesses used cloud computing in 2021¹, up from 19% in 2016.

9.1.1 Infrastructure-as-a-Service (IaaS)

IaaS is the situation where customers rent physical machines (PMs) or virtual machines (VMs) from the cloud provider. Typically these services are purchased through an API or web front end, and there is metered consumption, so that customers pay only for what they use. This enables small companies to avoid the upfront capital cost of buying hardware, and allows them to flexibly expand their hardware provision as the company expands. Examples of IaaS include Amazon Web Services (AWS).

9.1.2 Platform-as-a-Service (PaaS)

PaaS is IaaS plus a set of standardised services, such as data management, logging, email API, payment services, etc. These platform services enable users to build software more efficiently by making use of these existing services, instead of coding their own solutions. The idea is similar to application servers for enterprise systems. An example is Google App Engine.

9.1.3 Software-as-a-Service (SaaS)

SaaS is a model of software provision where customers rent usage of a cloud-hosted software system, typically accessed through a web front end or app (Figure 9.1). The data of the system is managed transparently on cloud servers, and the execution is shared across different servers depending on loading. SaaS is an efficient delivery model from the provider's point of view, because the provider keeps control of the software and its support operations, and updates are rolled out to all clients (or to a selection of clients) by updating the server code. Examples include Microsoft Office 365, and Dropbox.

There are also advantages for SaaS customers, in that they can access the software from any device, and there are no up-front costs for software or servers. They receive immediate software updates, with effectively continuous deployment, and there are reduced software management costs.

¹ <https://ec.europa.eu/eurostat/>

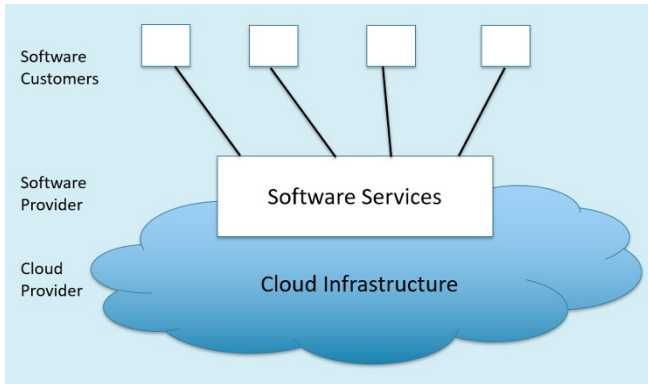


Fig. 9.1 Software as a service

However there are also disadvantages: their data is held remotely (raising privacy and security issues), and there may be service lock-in (a need to keep paying to access the software even when it is not being used).

9.1.4 Virtualisation servers

Related to cloud provision are services which enable the packaging of applications together with all software and services required by the application in a pre-initialised bundle. The principal service of this type is Docker². The application bundle acts as a ‘container’ or virtual server that hosts the application, and this bundle provides for execution of the application in its own separate environment, independent from other software running on the same device. This represents a more lightweight form of service delivery compared to using virtual machines on the cloud, and requires less resources and reduced activation time compared to virtual machines.

9.2 Service-oriented architecture (SOA)

In general, SOA views a system as a set of communicating services, the services are self-contained functional units, and have minimal shared datastores. They communicate using standard protocols and data formats (e.g., HTTP, XML, JSON). Services should be loosely-coupled, not aware of the internals of other services, and technology and location independent.

² <https://www.docker.com>

To make SOA work, service discovery mechanisms are required, e.g., via registries of services/service brokers. Services can be composed, orchestrated, etc, using different patterns for SOA. In general, SOA shifts complexity from the internal structure of an application into a network of collaborating services.

Figure 9.2 gives an example of SOA, where a travel booking service uses a local price calculator service and three external services for booking flights, car hire and hotel rooms.

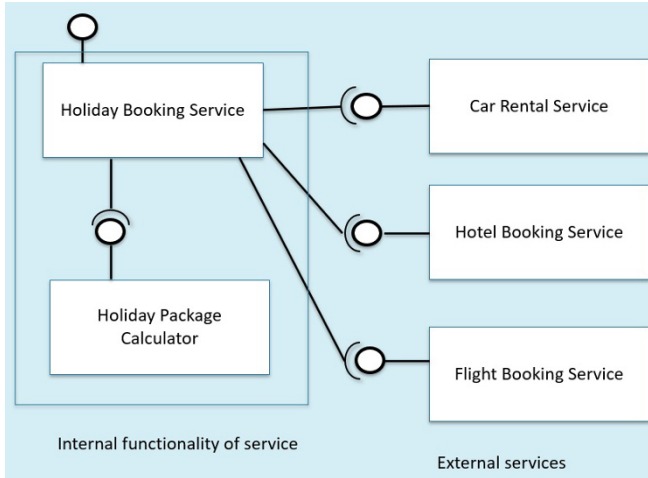


Fig. 9.2 SOA example

The main kinds of services we will consider are *web services* and *microservices*.

9.3 Web services

Web services are functional components which can be utilised via the Internet, either via a browser, or more usually, by a programmed request from a client application. Web services address the problem of how to integrate various systems written under heterogeneous platforms and different languages, distributed over a network. It is generally impractical to re-implement all the parts of such a system to operate under a single common platform or communication framework, especially if they are owned/managed by different organisations. Instead, it is possible to integrate them by implementing connection points between the systems as web services.

Figure 9.3 shows the general organisation of web services. A provider of web services publishes information about these (their purpose, what param-

eters they expect, what information they return, etc) in a directory, which can be searched by clients to find a specific service. Requests and responses then pass between client and server using either a special-purpose web service message format such as Simple object access protocol (SOAP), or as HTTP requests/responses.

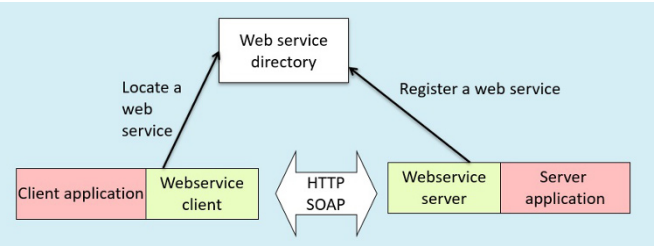


Fig. 9.3 Web service organisation

Web services are typically business-to-business (B2B) functionalities such as product ordering, shipment requests, etc, or they involve access to remote data. Web services can be a common subtask in several business processes, but they should not be performance-critical or require fine-grained interchange of data, because of the latency of network communication.

A *web service host* is the machine on which the web service executable resides. The *publisher* of a web service makes the web service available for use, whilst the *consumer* of a web service is the application that uses the web service. Figure 9.4 shows how a web service architecture could be represented in UML component diagrams.

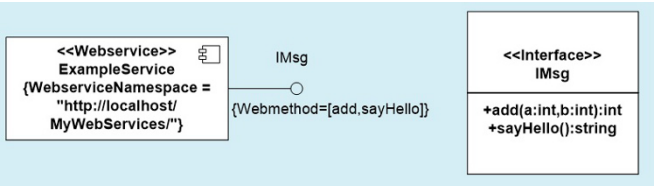


Fig. 9.4 Web service architecture in UML

We treat web services like other components, and use the `<<WebService>>` stereotype to indicate that a component is expected to be used/implemented as a web service. Specialised tagged values are introduced: *WebserviceNameSpace* to indicate the service location, and *Webmethod* to indicate the web service methods to be exposed via a provided interface.

There are two main web service approaches: SOAP and REST.

9.3.1 SOAP web services

SOAP is the *Simple object access protocol*, an XML-based protocol for exchanging messages, including descriptions of remote procedure calls (RPC). SOAP messages are XML documents (*envelopes*) describing a method call of the service. The message body can contain request or response data. Together with SOAP messages, there is also a registry language WSDL, the *Web services definition language*. This is an XML-based description of services. A client component looks up a required service in the registry, obtains a proxy reference to the web service, and invokes methods on that. The SOAP infrastructure transforms method invocations into XML messages for transmission over the network.

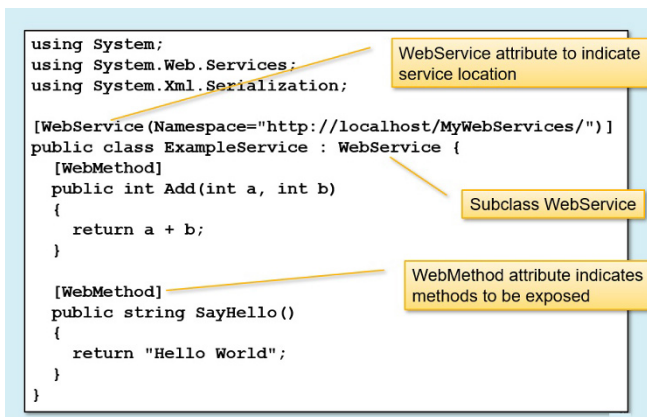


Fig. 9.5 SOAP web service in .NET

Figure 9.5 shows an example SOAP web service in .NET, with COM+ annotations used to mark a class as a web service and its methods as available web methods.

Figure 9.6 shows the corresponding annotations used in Java EE for Java SOAP web services.

9.3.2 REST and RESTful web services

Representational State Transfer (REST) is another technique for implementing web services. In this model, data and functionality is accessed using Uniform Resource Identifiers (URIs), and resources/services are identified by URIs, acting as global addresses. Unlike SOAP, REST uses only standard HTTP commands, such as:

```

import java.util.*;
import javax.xml.ws.WebService;
import javax.xml.ws.WebMethod;
import javax.xml.ws.WebParam;

@WebService( name = "ExampleService", serviceName = "ExampleService" )
public class ExampleService
{ @WebMethod( operationName = "add" )
  public int add(int a, int b)
  { return a+b; }

  @WebMethod( operationName = "sayHello" )
  public String sayHello()
  { return "Hello World"; }
}

```

WebService annotation

WebMethod annotation indicates methods to be exposed

Fig. 9.6 SOAP web service in Java EE

- GET – retrieves the current state of a resource.
- POST – transfers a new state onto a resource.

Resources are decoupled from their representation, so their content can be accessed in a variety of formats, e.g. HTML, XML, plain text, PDF, JPEG, JSON, and others.

Compared to the SOAP/WSDL approach, REST is simpler (no special languages or service registries are needed) but less flexible: clients must know the precise URL of the service that they require.

One factor with REST is that in HTTP every interaction with a resource is stateless: request messages are self-contained and cannot refer to previous states. Stateful interactions can be programmed based on the concept of *explicit state transfer*, e.g., by using URI rewriting, cookies, and hidden form fields to store information about the state of an interaction/sequence of actions. State can be embedded in response messages to point to valid future states of the interaction.

Figure 9.7 shows an example REST web service, which is a share price predictor using Monte-Carlo simulation to forecast future share prices based on the current price, growth rate and volatility of the share. All the necessary data for the service is supplied as input parameters of the service request:

```

currentPrice : double
timeDays : int
  /* length of period for prediction */
growthRate : double
variation : double
  /* volatility of the share */
runs : int
  /* number of samples to generate */

```

The number *runs* determines the accuracy of the result: increasing the value leads to greater accuracy, but also consumes more time and resources. The

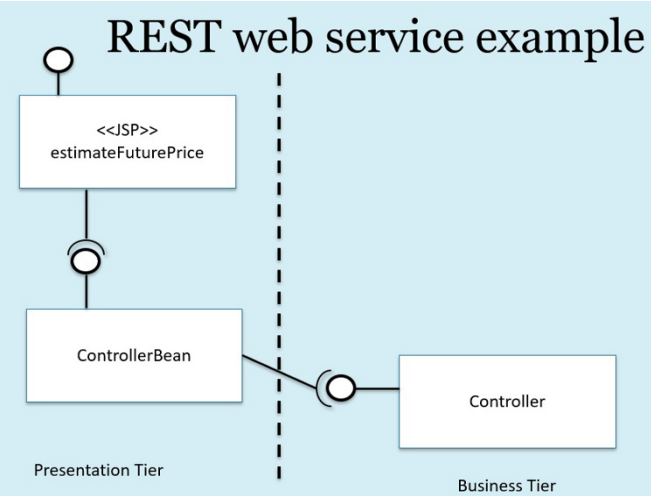


Fig. 9.7 REST web service example

result parameter gives the estimated future price. We name the service *estimateFuturePrice*.

The parameter data can be supplied programmatically or via an input web form (Figure 9.8).

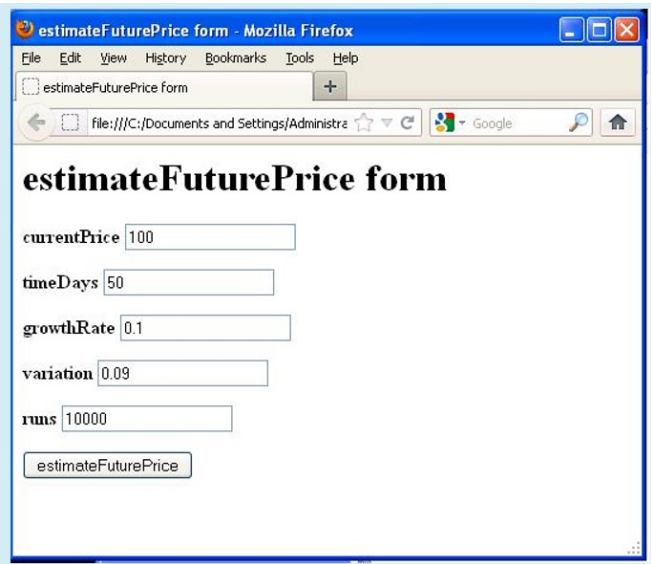


Fig. 9.8 Input form for web service

On the server side the web request issued by this form can be processed by a JSP, which also produces the result page:

```
<jsp:useBean id="bean" scope="request" class="beans.ControllerBean"/>
<jsp:setProperty name="bean" property="currentPrice" param="currentPrice"/>
<jsp:setProperty name="bean" property="timeDays" param="timeDays"/>
<jsp:setProperty name="bean" property="growthRate" param="growthRate"/>
<jsp:setProperty name="bean" property="variation" param="variation"/>
<jsp:setProperty name="bean" property="runs" param="runs"/>

<html><head><title>estimateFuturePrice</title></head>
<body>

<h1>estimateFuturePrice</h1>
<% bean.estimateFuturePrice(); %>
<h2>estimateFuturePrice performed</h2>
<strong> Result = </strong> <%= bean.getResult() %>

</body>
</html>
```

In turn, this executes the service via a *ControllerBean* object, which is a business delegate for the business tier service:

```
package beans;
import java.util.*;

public class ControllerBean
{ Controller cont;

    public ControllerBean() { cont = Controller.inst(); }

    double currentPrice, growthRate, variation;
    int timeDays, runs;
    String result;

    public void setcurrentPrice(String _s)
    { /* convert _s to a double */}

    public void estimateFuturePrice()
    { result = "" + cont.estimateFuturePrice(
        currentPrice, timeDays, growthRate, variation, runs);
    }

    public String getResult() { return result; }
}
```

9.3.3 Web service patterns

There are specific architectural patterns for web services, in particular:

- *Service Broker* – this defines a service which is designed to locate services which can meet a given request, and links such a service to the requestor.

- *Service Decomposition* – this decomposes a coarse-grained service into two or more subservices.

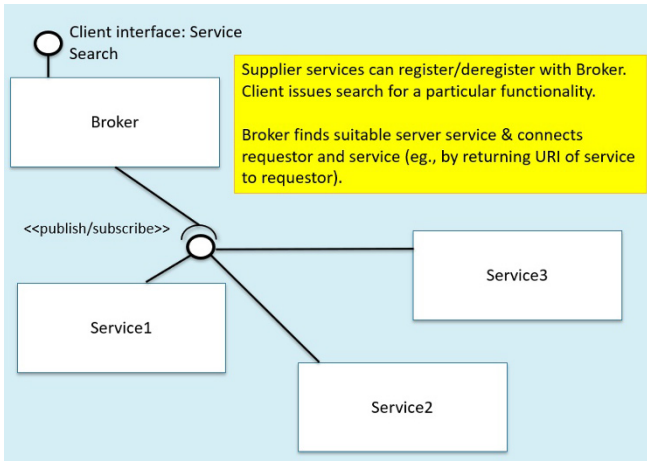


Fig. 9.9 Service Broker

Figure 9.9 shows the general schematic form of the Broker pattern for services, and Figure 9.10 a specific example of the pattern, to search for a flight booking service. The client of the broker can specify various restrictions and requirements to be met by the supplier services, e.g., price restrictions or other policies. The broker will then select services which meet these restrictions, if there are any such services available.

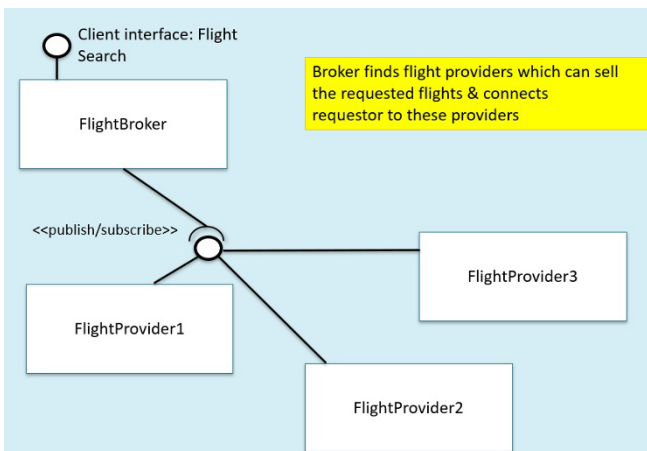


Fig. 9.10 Service broker example

Service decomposition aims to split a coarse-grained service into smaller services, which provides greater flexibility and the potential for independent use of the new services within new systems. Decomposition can be performed based on separating data or functional subunits of a service. For example, a service to translate Visual Basic (VB) programs to Python (Chapter 12) can be split into a sequential composition of a VB to UML translation, followed by a UML to Python translation. These functionalities could make sense as separate services for a wider range of purposes than the original program translation.

9.4 Microservices

Microservices are an architectural style which involves applying SOA with fine-grained services, each service represents a basic functional element or step in an overall process, and each executes in its own runtime environment. Communication is typically via web-based protocols, with no shared data.

There are two main strategies for microservice composition:

- *Orchestration* – where one master service co-ordinates other services in the correct order.
- *Choreography* – Services coordinate with each other, by generating and listening to events.

Microservice orchestration can be carried out by an API gateway, which provides an access point for clients to possibly a set of services. The gateway selects the correct service based on incoming requests, and translates service requests into calls to individual services. Effectively, it is a Front controller (Chapter 8).

For example, several related financial services could be managed in this way: Share price estimation; option valuation; trading advice, etc. (Figure 9.11).

Alternatively, Figure 9.12 shows an example of choreography, where different financial services coordinate to produce an overall result to a client.

Microservices have several advantages:

- Independent service modification.
- Incremental updates of application functionality.
- Continuous deployment in an agile development process.
- Microservices enforce cross-functional teams rather than functional silos.

However there are also disadvantages:

- Deployment is more complex as there are more ‘moving parts’.
- Higher skills and better tools are needed for development.
- For good performance there needs to be efficient cross-process communication.

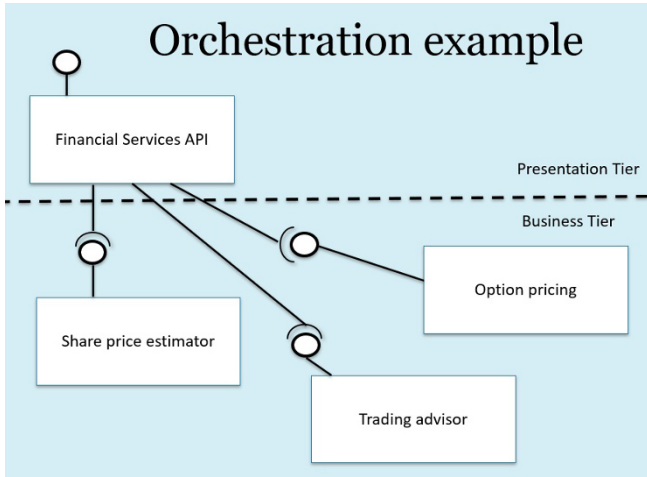


Fig. 9.11 Orchestration example

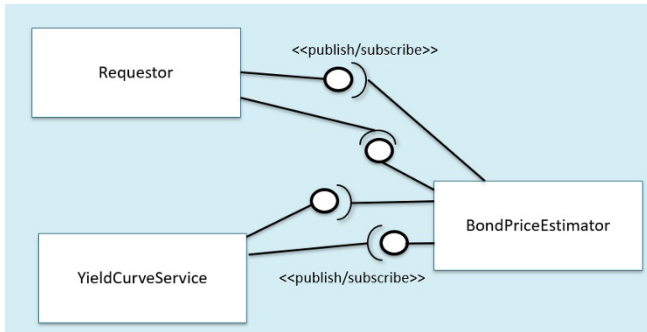


Fig. 9.12 Choreography example

- It is difficult to move component boundaries, because microservices represent a functional decomposition of the system, not an object-oriented decomposition.

A particular problem is failure management – any service can fail at any time. To handle this there needs to be extensive monitoring and defensive programming. Internal service failures should be detected by the service, and reported to the requestor, e.g., by using HTTP error codes (200 for success, 404 for failure due to an unreachable resource, etc). External service failures occur when some external cause prevents the service from operating. Actions are then needed to restart the service. There may also be service performance failure, with degradation of service due to excessive loads, etc. This can be addressed by monitoring and reconfiguration, for example, by using a *cir-*

cuit breaker which cuts an unresponsive service out of the configuration of services³.

Summary

This chapter has described some of the architectural styles and implementation issues for service-oriented architectures, in particular for web services and microservices.

Exercises

1

A publishing company has organised its book sales as a set of related services: purchase, catalog, payment, and delivery:

- *purchase*: takes a search term from the customer, uses the *catalog* to find matching products, and lets the customer select one or more, with specified quantities. It computes the total price and passes the information and customer to the *payment* service.
- *payment* interacts with an appropriate external service, such as Paypal, to complete the purchase, and when successful, passes the order information and customer to the *delivery* service.
- The *delivery* service needs to find a set of possible delivery companies which have minimal cost/time estimates for the delivery job. These are then presented to the customer for selection of their preferred delivery option.

Draw the architecture of this system. What service pattern(s) are relevant for this system?

2

For the system of Q1, if different delivery agents have the following (time, cost) estimates in (days, GBP), what options are presented to the customer?

- (5, 1.5)
- (4, 3)
- (6, 1.2)
- (2, 1.5)

³ <https://martinfowler.com/bliki/CircuitBreaker.html>

(1, 3)

3

Classify the following systems as PaaS, IaaS, SaaS, etc, and identify their business model (pay-per-use, subscription, etc).

1. Google colab⁴ – a browser-based Python environment making use of cloud-based computing resources.
2. Apache Ignite⁵ – cloud-based machine learning services.
3. Azure ML studio⁶ – cloud-based platform for building and running machine learning systems.

⁴ <https://colab.research.google.com>

⁵ <https://ignite.apache.org>

⁶ <https://azure.microsoft.com>



Chapter 10

Safety-critical and Embedded Systems Architectures

Abstract In this chapter we look at the specific architectural issues involved in *safety-critical systems* (SCS), and design approaches for SCS. Because SCS are often *embedded* systems, operating upon special-purpose hardware/devices, we also examine architectural approaches for embedded systems in this context.

10.1 Safety-critical systems

SCS are systems whose functioning or malfunctioning may produce accidents leading to loss of life or severe environmental/economic damage. Software can be safety critical if it is used to control a safety-critical system. Safety critical systems and software require additional development effort – specifically in *hazard analysis*, *risk analysis*, and the production of *verifiable designs and implementations*. There will typically be the need for *certification* or other approval of the software before the SCS is permitted to begin operation. For the highest integrity systems, techniques such as *formal methods* and *static analysis* are used to achieve the necessary degree of assurance.

10.1.1 Technological risk

Whilst occupational accidents and deaths have declined in the last 100 years, the increasing use of software in critical systems increases the potential system risks.

Risk is defined as:

$$Risk = Likelihood \text{ of accident} \times Severity \text{ of consequences}$$

Thus, both likelihood and severity factors need to be considered in assessing system risk. Likelihood factors for software-based SCS include latent errors in code, hardware failures, incorrect operation by humans, etc.

Severity factors concern the impact of accidents in domains such as nuclear power, transport, medical procedures, etc.

There have been several cases of significant SCS failures which can be traced to software causes, for example:

- *Therac-25 radiation therapy machine* (1985–87): six people killed or seriously injured by overdoses. The software system had many faults, including inadequate management of multiple processes, with access to shared variables without synchronisation, and test and set instructions not being indivisible, etc. There were no hardware protections.
- *Failure of Patriot missile system at Dhahran, Saudi Arabia* (1991): continuous operation for more than 100 hours led to a ranging error in the missile radar. As a consequence, an incoming Scud missile was not intercepted, resulting in 28 deaths and 110 injuries.
- *London Ambulance Dispatching System* (1992): failure under overload conditions caused potentially lethal delays.
- *Space Shuttle*: 12 significant software errors encountered in flights, and a further 16 severity-1 errors were discovered in the code.
- *Ariane 5* (1996): unnecessary code (reused from Ariane 4) produced a poorly-handled numeric overflow, which led to an error in the flight control and caused the loss of the unmanned flight, June 1996 (no fatalities, but over \$300 million cost).

In this case there was hardware redundancy in the form of a backup computer, but this was running the same FCS as the main computer – so both copies failed.

- *Mars Climate Orbiter* (1999): spacecraft lost during mission (cost about \$300 million). This was primarily due to a requirements fault – one software unit computed results in pound-force seconds, whilst another expected data in Newtons. Hence trajectory adjustments were out by a factor of 4.45.
- *Boeing 787 Dreamliner* (2015): experienced an integer overflow bug which could shut down all electrical generators if the aircraft systems were continuously on for more than 248 days.
- *Beresheet Moon lander* (2019): crashed on landing due to a software bug with its engine system, which prevented it from slowing down during its final descent.
- *737 MAX airliner* (2018–2020): a recurring failure in the flight control system caused two fatal crashes and 346 deaths (29th October 2018 and March 10th, 2019). The plane was grounded worldwide from March 2019 to November 2020.

Flawed information from a single external sensor fed into the flight control system (FCS) caused it to repeatedly push the plane nose down – the pilots struggled to keep their planes in the air before both crashes.

Computers introduce new risks because they enable greater complexity to be embedded and controlled, for example, self-drive cars and other autonomous systems.

The following are some definitions of SCS terms, based on [56]:

- *Reliability* – the probability that a component will perform its intended function satisfactorily for a specified duration/under specified conditions.
- *Safety* – freedom from accidents or losses.
- *Failure* – non-performance of function of component for a specified duration/under specified conditions. (i.e., an inability to fulfil operational requirements).
- *Error/fault* – a design flaw or deviation from the intended/desired state. Notice that a failure is an event, whilst an error is a state (which may lead to a failure).
- *Accident* – an undesired and unplanned event resulting in specified level of loss (financial loss, injury, loss of life or other significant loss).
- *Hazard* – a state/conditions of a system/component that with certain environmental conditions, will lead inevitably to an accident. That is, if a hazard state arises, this implies that an accident will happen in the right circumstances. For example, an electric saw creates a hazard when it is operating: it will cause an injury if a person's hand is in the path of the blade. A SCS that controls such a saw can prevent such a hazard and accident by shutting off the saw if a person is detected near the blade (alternatively, or additionally, the hazard can be prevented by placing a guard around the cutting area, which shuts off the saw if the guard is raised or removed).
- *Safety integrity level* (SIL) is a measure of criticality of a system and how safe it must be. This is based on risk analysis and classifies a system or component into SIL levels 1 to 4, where level 4 is the most critical.

10.1.2 Fault tolerance versus fault avoidance

Two main strategies are used to ensure the safety of software-based SCS: fault avoidance and fault tolerance. *Fault avoidance* means to prevent a fault occurrence or introduction by construction (e.g., by writing SCS software using formal specifications and other rigorous methods). On the other hand, *fault tolerance* means to provide a service that will deliver the required function despite faults (e.g., by means of protection systems or redundancy to counter software faults).

10.2 Fault tolerance

Fault tolerance is usually based on various forms of hardware and/or software redundancy. For example, hardware redundancy was used at the beginning of computing to cope with the high failure rates of valves (one example is the duplicated ALU's used in EDVAC (1949)). To be effective for tolerating software faults, a redundancy approach must provide protection against design faults as well as random hardware failures – it is not useful to run the same software on two computers, because the same design fault will cause both to fail (for example, as with the Ariane 5 flight loss). The SIL and failure rate required of a system guides the choice of a fault tolerant architecture – the higher the SIL, the more sophisticated the fault tolerance approach required. Analysis of the resulting design is then performed to confirm that the architecture has reduced the failure rate to the desired level.

Various categories of faults can be considered in general (Figure 10.1):

- *Random hardware faults* versus *Systematic faults*: systematic faults are due to mistakes in the specification of the system, mistakes in coding the software, or mistakes in the hardware design.
- *Permanent faults*: these persist until corrected – such as design faults, versus *Transient faults*: e.g.: α -particle interference with semiconductor memory, or *Intermittent faults*: e.g: electrical interference, poor circuit connections.
- *Localised faults*: affecting a single hardware/software module, versus *Global faults*: with effects throughout the system.

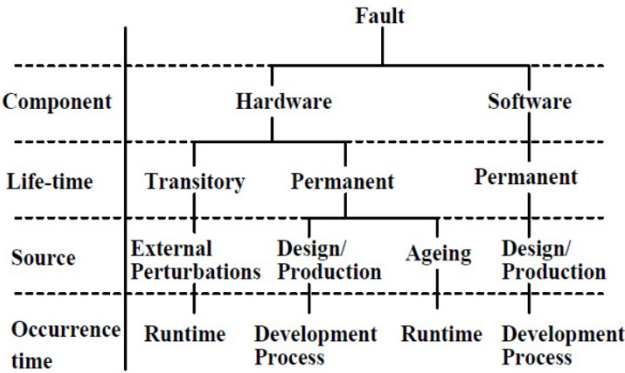


Fig. 10.1 Classification of faults

Here we are only concerned with systematic faults, and especially design and specification faults. For hardware, design faults may be detectable as a result of consequential ‘random’ failures (e.g., if the power dissipation of a

resistor has been wrongly specified, so that it fails within a short period of time).

For software, systematic faults can include:

- Software requirements and specification faults, such as specifying the incorrect units of measurement for a quantity.
- Coding faults – mistakes in the implementation of specifications.
- Logical errors within calculations, or errors due to the arithmetic imprecision of computational processing.
- Runtime errors such as stack overflows/underflows.
- Poor coding practices such as the use of uninitialised or untyped variables.

Some fault tolerance terminology is as follows:

- *Fault tolerance coverage*: the proportion of possible faults that can be tolerated.
- *Hardware redundancy*: when additional hardware is used to detect or tolerate faults.
- *Software redundancy*: when additional software is used to detect or tolerate faults.
- *Information redundancy*: the use of additional information to detect/tolerate faults. E.g., parity bits, checksums, etc.
- *Temporal redundancy*: the use of additional time. E.g.: repeating calculations and comparing the results to help remove transient fault effects.

10.2.1 Static redundancy: Triple Modular Redundancy architecture

One common fault-tolerant architecture is *triple modular redundancy* or TMR (Figure 10.2). In this approach three modules are used, together with a voting unit. If the output of one module disagrees with that of the other two, the voter takes the majority vote. This also enables the failed module to be identified. TMR protects against random hardware failures in individual modules.

TMR type redundancy doesn't however tolerate design faults if the modules are identical in design, because these will have *common mode failures* – they will all fail together on the same inputs. Instead, components of diverse design/technology can be used in the TMR arrangement. One module may use a microcomputer controller, with another module using non-programmable control, such as a sensor-triggered relay to control an actuator, or a Programmable Logic Controller (PLC). Alternatively the same hardware can be used for each module, but with *diverse software* controllers.

TMR is a form of *static redundancy*: it masks faults through redundant modules and the voting arrangement. An alternative is *dynamic redundancy*: where if faults are detected, the system is reconfigured to remove the influence

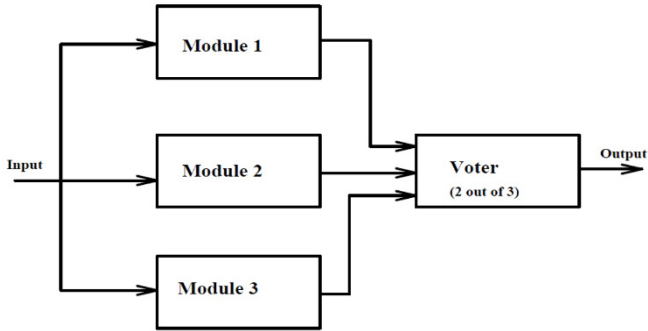


Fig. 10.2 Triple Modular Redundancy (TMR) architecture

of the faulty component (such as ‘hot’ or ‘cold’ standby spare arrangements). There can also be *hybrid* arrangements, with fault masking to prevent propagation of errors in the system, and fault detection and reconfiguration to remove faulty units.

TMR can tolerate a single fault: a single-point failure within the modules is eliminated by the voter. Other single point failures include common inputs, and the voter itself. Inputs can be taken from diverse sensors, but this may produce timing differences that produce output divergence. The voter can be built from highly dependable gates, or itself triplicated and TMR modules cascaded (Figure 10.3). Voting can be done in hardware (fast) or software (more flexible). Once a module has failed, TMR gives no further protection: it is important to notify this failure to operators, so that remedial action can be taken.

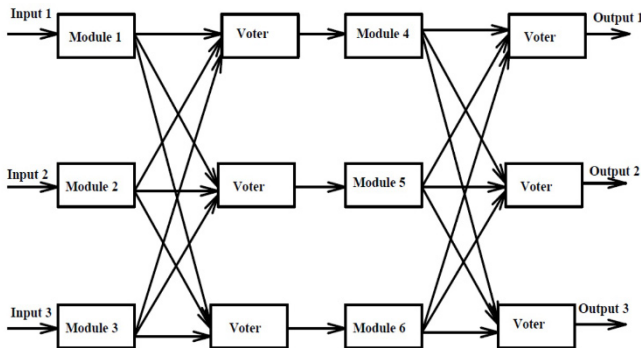


Fig. 10.3 Cascaded TMR

Notice that if *Module 1* and the first voter both fail in Figure 10.3, the other two initial voters will still give a majority output.

A TMR majority voting scheme with N modules (N odd) leads to the ability to tolerate $(N - 1)/2$ failed modules. Thus a minimum of five modules is needed to tolerate two faults. Higher values than $N = 3$ are rarely used because of the cost/power consumption, etc. The term *N-modular redundancy (NMR)* is used for $N > 3$.

10.2.2 Dynamic redundancy: Standby spare architecture

Dynamic redundancy reduces the number of modules required to achieve fault tolerance: there is usually one main operating module with one or more ‘standby spares’ (Figure 10.4). This architecture is suited to applications that can tolerate temporary loss of functionality in operation, because time is needed to detect an error and reconfigure the system. When an error is detected in the main module, the system is reconfigured to use a standby (replacement) module instead. The reconfiguration delay may be minimised by using a *hot standby* where the spare runs continuously in parallel. Effectiveness of the arrangement depends on the coverage of the fault detection method used.

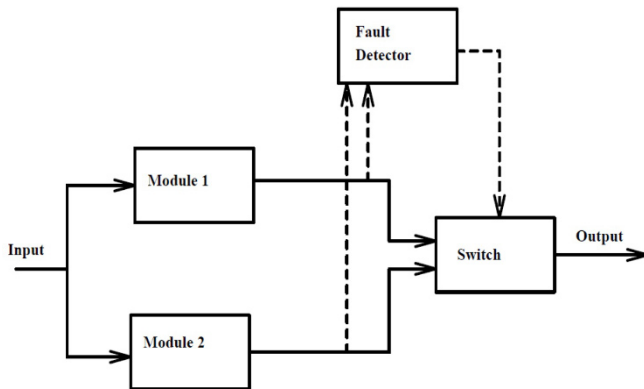


Fig. 10.4 Standby spare architecture

10.2.3 Software fault tolerance

In order to tolerate single design errors in SCS software, the concept of *N-version programming* is used, where different implementations of the same specification are employed as redundant elements. 2-version gives fault de-

tection, 3-version fault masking via a TMR architecture, etc. This concept is used in the primary flight control system of the A330/340 aircraft. The separate diverse software components should not affect each other's functioning, either directly by sharing data, or by their use of processor/operating system resources.

An alternative approach is *recovery blocks*: this is a dynamic redundancy approach where error detection is used to attempt alternative code for an operation. This is similar to the try-catch-finally exception handling construct of languages such as Java and Python. An example syntax for recovery blocks is:

```
ensure  acceptance test
by    primary module
else by alternative module 1
...
else by alternative module n
else error
```

The concept of N-version programming has been questioned, because it has been found that different developers often make similar omissions or design mistakes when implementing a common specification, and so the 'diverse' software versions produced by these developers would actually have common failures.

For example, on a 'Classify triangle' problem represented by the function

$$f : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \{\textit{scalene}, \textit{isosceles}, \textit{equilateral}\}$$

on triangle side lengths, different versions were written independently by 60 different people in 10 minutes. Tests such as $f(2, 2, 3) = \textit{isosceles}$ and $f(4, 5, 6) = \textit{scalene}$ worked fine. But 58 of the 60 evaluated (incorrectly) $f(2, 3, 10) = \textit{scalene}$. All but a few failed on $f(0, 0, 1)$ and $f(1, 3, 0)$. Common errors can also be due to missing/incomplete information in the software specification which is not corrected during development.

Diversity may reduce the overall failure rate (because two diverse versions will not always fail together on the same input) but does not guarantee independence. Increasing reliability of individual versions may actually reduce independence, because the faults that remain are more likely to be in common.

10.3 Safe states and safety shutdowns

Some systems have 'safe' (or at least safer) states:

- Railway signalling systems: all lights set to stop.
- Drilling machinery: switched off.

- Fission reactor: all damping rods fully inserted into the core.

Accidents may still happen, but the main hazards have been eliminated.

Other systems must be actively controlled to remain safe, such as self-drive cars and fly-by-wire aircraft.

Shutdown systems can put the first type of systems into their safe states when faults are detected. The shutdown systems are typically much simpler than the main control system, in order to achieve high integrity, and may be implemented in hardware or simple computational devices such as programmable logic controllers (PLCs).

The second type of systems need to utilise fault tolerance techniques, with flexible adaptation of their functioning in the case that faults occur. It may be possible to provide some active protection in certain cases – for example, ‘envelope protection’ for an airliner provides an active check and correction if the plane trajectory becomes hazardous – but this requires some active working functionality, in contrast to a simple safe state shutdown. This form of system therefore may have *protection systems* instead of shutdown systems.

A basic shutdown system architecture, suitable for SIL 1 systems, is shown in Figure 10.5. The PLC diagnostics module triggers a shutdown if the shutdown system itself develops a fault.

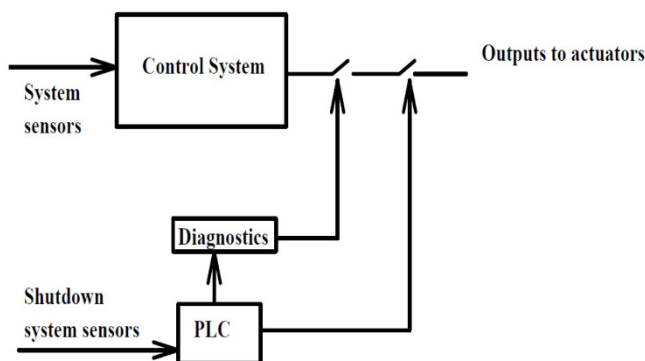


Fig. 10.5 Shutdown system for SIL 1 SCS

For higher integrity levels, some degree of fault tolerance is needed in the shutdown system itself (Figure 10.6). This helps to avoid spurious shutdowns due to failure of a single shutdown system module, but adds to the complexity of the system.

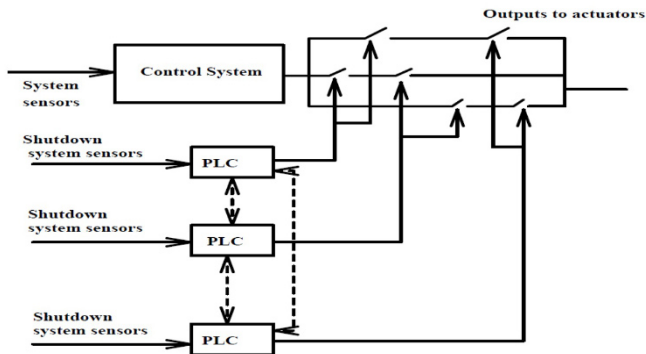


Fig. 10.6 Shutdown system for SIL 3 SCS

10.4 Design approaches for SCS components

The fault tolerant architectures described above are usually employed at the level of complete functional units or subsystems. Within each subsystem, the following principles should be employed:

- Iterative partitioning of software modules – this aids comprehension and strengthens isolation between module parts.
- Partitioning yields a hierarchical layered structure – the outer layers provide external control and command functions, implemented by calls to inner levels.
- *Localise* decisions where possible – if information about a sensor is needed for an interlock on an actuator, the information shouldn't pass up and then down several module levels (Figure 10.7) – instead make a single low-level module responsible for the interlock (Figure 10.8).
- Keep safety-critical routines as small/simple as possible.
- Isolation between modules is critical to N-version programming redundancy schemes – separate versions of a module should not influence each other's execution.

Conflicts may arise between different parts of a system if they both control the same physical actuator, for example for different functionalities – careful management of the system state/mode is needed in such a case to ensure that at most one function can be active at any time.

10.4.1 Programming languages for SCS

A controversial topic is the choice of a programming language for SCS. For very high integrity applications requiring rigorous/formal verification, a pro-

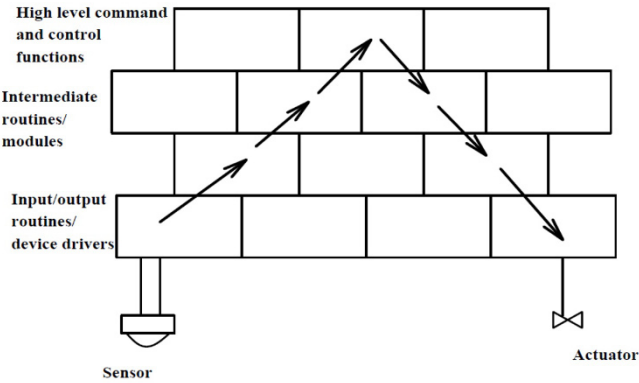


Fig. 10.7 Poor module decomposition structure

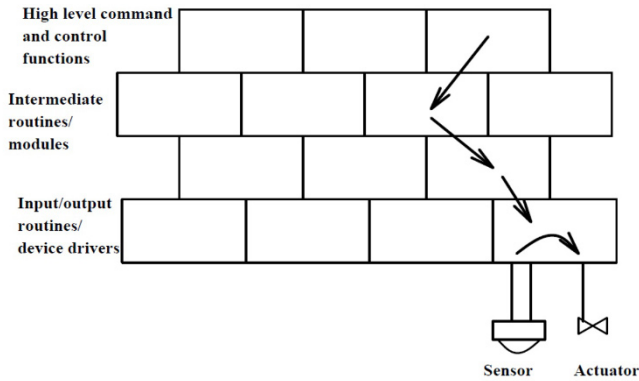


Fig. 10.8 Improved module decomposition structure

gramming language designed for verification should be used (e.g., specialised language subsets such as SPARK Ada [7]). For any SCS, programmers should avoid error-prone features such as unconstrained memory access (as with C pointers) or runtime variation in functionality (such as object-oriented dynamic binding/polymorphism). A wide range of languages are used for SCS in practice, but Ada has become a main language for SCS, especially for aerospace applications. Despite a number of problematic features, C is often used because of the simplicity of hardware interfacing using C: both firmware and software can be written in the same language. Improved C subsets, such as the MISRA C subset¹ designed for automotive applications, are also used.

¹ <https://ldra.com/misra>

10.5 Embedded system architectures

The components of an SCS are often *embedded systems*: software-based systems operating upon special-purpose hardware devices (and possibly without an operating system). The *layered architecture* (Chapter 5) is the usual approach for embedded systems, with a clear separation between the *firmware* and *software* of the system, defined in separate layers (Figure 10.9):

- *Firmware* – code which directly interacts with hardware and is device-specific.
- *Software* – code independent of specific device/hardware, and which uses the firmware to interact with the device.

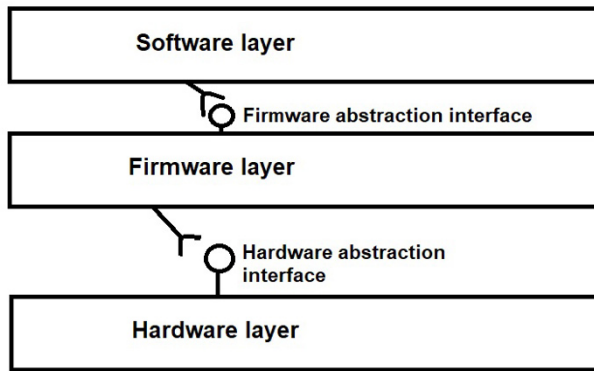


Fig. 10.9 Embedded system layered architecture

An important principle is to avoid hardware dependence within the system as far as possible, and to localise such dependencies. The parts of code which know what specific device is being controlled should be limited to firmware. This enables the code to be tested off-device, using a mock-up of the hardware (a simulator). It also avoids the need for extensive rewrite of the system code when the hardware is upgraded or changed: only the firmware needs to be changed or replaced. There may be an additional architectural layer for the operating system (if present) between the software and firmware.

An example of an embedded SCS is an airbag system for a car (Figure 10.10).

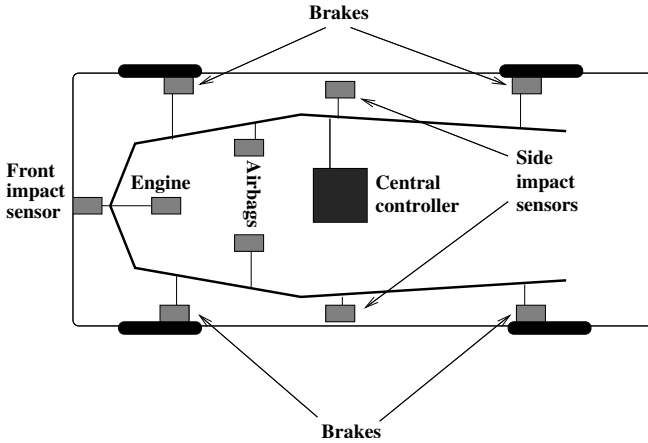


Fig. 10.10 Embedded system example

10.6 Reactive control systems

SCS and embedded systems often involve *reactive control*: a cyclic control function, which in each cycle polls sensors and user commands, and responds to detected events/commands by sending commands to actuators. It may also send alerts/warnings to operators.

The processing of such a system involves the following repeated cycle:

1. Read sensor inputs/operator commands.
2. Make decisions.
3. Send actuator commands/operator messages.

The processing is intended to maintain and enforce *control invariants*, which relate the sensor and actuator states. These invariants can express safety conditions, i.e., that no hazardous state exists.

The airbag system of Figure 10.10 could be constructed in this way: the sensors are the side and front impact sensors, and the actuators are the engine, airbags and brakes (Figure 10.11). In this case there are no user commands or messages. The control invariant maintained by the system can be expressed as:

“If any impact sensor is triggered, the airbags should be inflated, the engine should be switched off, and the brakes activated.”

A more formal version of this constraint could be written as:

```
(lsensor = On or
rsensor = On or
fsensor = On) implies
    lairbag = Activated &
    rairbag = Activated &
```

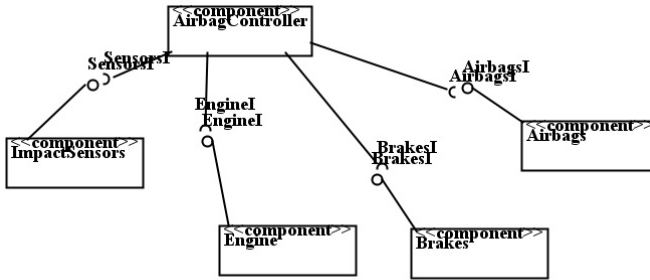


Fig. 10.11 Airbag system architecture

`engine = Off & brakes = On`

The variables *lsensor*, etc would be internal variables of the *AirbagController* which model the state of the sensor and actuator components, as known to the controller. The single *cycle* operation of *AirbagController* would poll the input sensors and send commands to the actuators to ensure that the invariant always holds at the cycle termination.

Another reactive system example is an aircraft autopilot system, which is responsible for maintaining the plane altitude and direction (heading) by polling sensors (altitude, airspeed, rate of climb, pitch angle, heading angle, bank angle, slip angle) and the pilot switches, and sending commands to actuators (elevators, ailerons, rudder – shown in Figure 10.12). The controller can be subdivided into two subcontrollers for maintaining altitude and maintaining heading (Figure 10.13): these subcontrollers control disjoint sets of actuators, so should not conflict with each other. Each subcontroller only needs a subset of the overall sensor information.

Typically a reactive control system is organised as an N-tier architecture with device-specific low-level controllers at the base level and a hierarchy of coordinating control layers above these.

In general for SCS reactive control system architectures we should separate the operational control of the system from safety shutdown or protection functions. The main control functions are typically programmed in software, whilst the shutdown/protection functions can be non-programmable or simple programmable (e.g., PLC) mechanisms, with a greater reliability/SIL than the main controller. The shutdown/protection commands to actuators should override the main control commands to the same actuators.

10.7 Examples of SCS architectures

The following are some notable examples of SCS architectures:

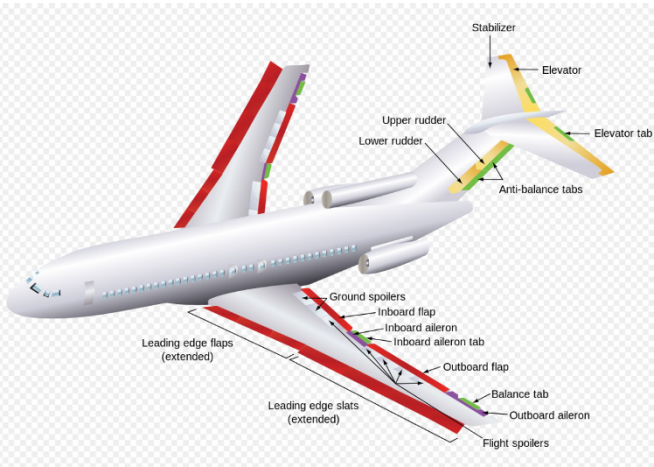


Fig. 10.12 Airliner flight control surfaces (Public domain image from www.faa.gov)

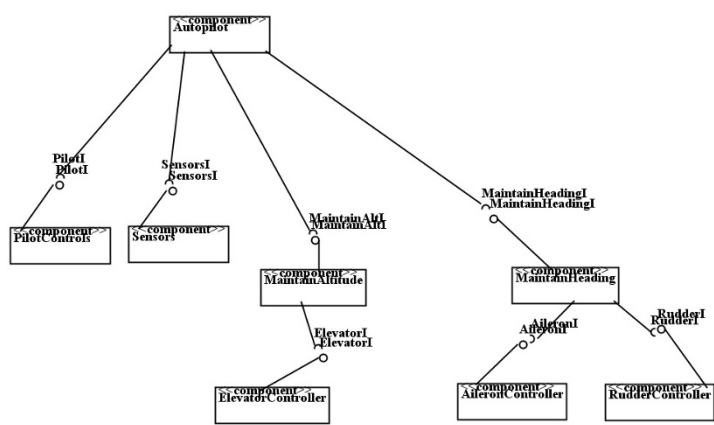


Fig. 10.13 Autopilot architecture

- *Space shuttle* – this used five identical computers with four in an NMR arrangement. The fifth could take over critical functions if all others failed, and it had a diverse version of the flight control software.
- *A330/A340 primary flight control system* – the FCS of this aircraft normally operates the control surfaces of the plane in an entirely ‘fly-by-wire’ mode. But the plane can be flown manually if the FCS fails. Multiple redundancy is used, with five computers, arranged so that failure of a single computer does not lead to loss of control of any surface. Progressive degradation of the control function is supported: three failures are needed before the system reverts to a ‘direct’ (manual operation) control law.

10.8 Software safety standards

A wide range of SCS standards have specific requirements on SCS software, across transport, health and nuclear domains:

- IEC 61508 – for safety-critical programmable systems in general.
- ISO 26262 – for automotive systems.
- IEC 62304 – medical devices.
- CENELEC EN 50128 – railway applications.
- DO-178C – for avionics systems.
- PAS 754 – software trustworthiness.

Usually these standards are expressed in terms of safety integrity levels, based on risk analysis and threshold probabilities for failure.

Summary

We have described the specialised factors and appropriate techniques for designing architectures for safety-critical software systems. Architectures for fault tolerance and fault recovery have been defined, and architectural techniques have been described for embedded and reactive control systems.

Further Reading

- N. Leveson, *Safeware: system safety and computers*, Addison-Wesley, 1995. ISBN 0-201-11972-2.
- N. Storey, *Safety-Critical Computer Systems*, Addison-Wesley, 1996. ISBN 0-201-42787-7.
- T. Stalhane et al., *The application of Safe Scrum to IEC 61508 certifiable software*, SINTEF, 2012.
- <https://www.nasa.gov/consortium/ModelBasedSystems>, 2023.
- AUTOSAR development approach for vehicle systems: www.autosar.org/, 2023.

Exercises

1

The 737 MAX disasters were apparently caused by an auxiliary flight control system (MCAS) sending the aircraft into steep dives due to a single faulty sensor reading (angle-of-attack, AoA).

What design flaws in the flight control of the aircraft does this imply? How could they be corrected?

2

In the airbag system discussed in this chapter, it is required to inflate the airbags within T milliseconds of an impact detection. What is a sufficient time bound on the cycle time of the reactive controller if this response time is to be ensured?

3

Recall the autopilot system from Figure 10.13. Define suitable interface specifications for this system, based on the following specifications of data:

Pilot controls (all boolean-valued):

- Master switch.
- Altitude switch.
- Heading switch.

Pilot setting (double-valued):

- Target altitude.
- Target heading.

Sensor data (all double-valued):

- Altitude.
- Airspeed.
- PitchAngle.
- RateOfClimb.
- BankAngle.
- HeadingDegree.
- SlipAngle.

The first four sensors plus targetAltitude are needed by the altitude controller, and the last three plus airspeed and targetHeading are needed for the heading controller.

Actuator data consists of the following double-valued parameters, representing degrees of deflection from a neutral position:

- Elevator angle.
- Aileron angle.
- Rudder angle.

4

Which kinds of failures/faults does a TMR architecture with 3 duplicated software modules for a software-based safety-critical system protect against?

1. Operator errors.
2. Random hardware failures.
3. Software design faults.
4. Software specification errors.

5

A safety-critical subsystem within a chemical processing plant consists of a holding tank for a toxic liquid (Figure 10.14). The control system for this tank opens the outlet valve $V2$ and closes the inlet valve $V1$ when a high level is detected in the tank by sensor $S1$, otherwise $V1$ is open and $V2$ is closed.

Why is this unsatisfactory as an SCS design? Suggest improvements which could be made, and explain why they increase the safety of the system.

6

Express as logical formulae the following modified control invariants for the airbag system (Figure 10.10):

- (i) If the front or left side impact sensor is triggered, then the left side airbag should inflate, the engine should be switched off and all brakes activated.
- (ii) If the front or right side impact sensor is triggered, then the right side airbag should inflate, the engine should be switched off and all brakes activated.

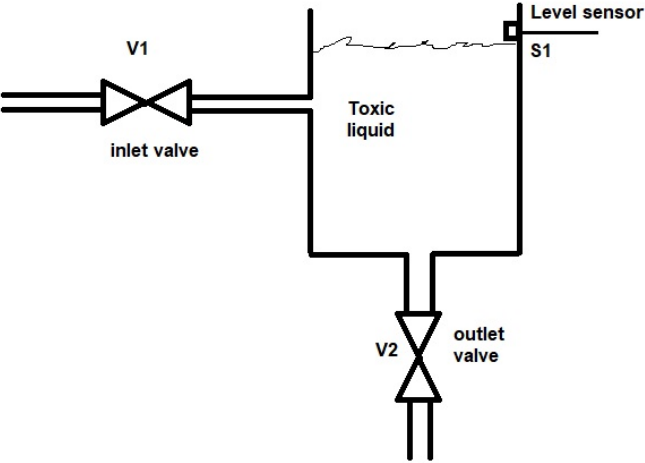


Fig. 10.14 Holding tank



Chapter 11

Architectural Design for Machine Learning Systems

Abstract This chapter considers the architectural design of machine learning (ML) applications, and identifies factors which can help to ensure the maintainability and quality of these systems.

11.1 Machine learning (ML)

ML is a particular domain of artificial intelligence (AI) systems, whereby software systems can be constructed by training a ML model on datasets of examples, using a particular ML technique/algorithm, in order to learn data classification or translation rules or other information, which can then be applied to new examples. The learnt knowledge is usually referred to as a (ML) model, but the same term is also used for the ML software structure (such as a particular form of neural net) in which the knowledge is embodied. It is clearer to use the term *ML model structure* or *ML model architecture* to denote the latter concept. Although ML systems provide unprecedented analysis and predictive power, they often depend upon very large training datasets (sometimes in the 100 GB range and upwards), and the knowledge learnt may only be implicitly represented within the ML model structure, e.g., in the connection weights between neurons in a neural net. ML systems may have a high carbon footprint, with the typical training footprint of an ML model being five times the lifetime emissions of an average car [66].

11.1.1 Machine learning techniques

Training data items are usually represented as tuples or vectors $[f_1, \dots, f_n]$ of values for a selected set of *features* which have been chosen to characterise the data items, and which provide sufficient information for the ML algorithm

to learn the ML model to produce the required results. For example, for a banking ML system to learn rules about whether to lend money to a customer, relevant features would be the customer credit score, employment status, amount to borrow, their salary/income, and age. Other features, such as the customer height and name, would be irrelevant.

ML approaches divide into two main kinds:

- *Supervised learning* (as with decision trees and simple neural nets), whereby the training dataset contains both input data and the expected label/result associated with that input. For example, a classification problem may take as input sequences $[f_1, \dots, f_n]$ of real values for each feature/attribute of the input item, together with an output label L from a finite set defining the correct classification of the item. In the case of issuing of a bank loan, the label could be a yes/no decision.
- *Unsupervised learning* only takes the input items, without labels, and aims to derive some rules about the data, e.g., to determine clusters of data or relationships between features.

In either case, there are a wide range of ML techniques that can be used, and within each technique, a range of possible ML model architectures.

Table 11.1 summarises some of the main ML techniques and their characteristics.

Table 11.1 ML techniques

<i>ML technique</i>	<i>Supervision</i>	<i>Training</i>	<i>Inference</i>
K-nearest Neighbour (KNN)	Yes	All input data is accumulated	Based on properties of the K nearest known data items to the new data.
Decision trees	Yes	Conditions on tree branches adjusted to minimise output errors for known data.	Tree applied to new data.
Random forest	Yes	Training of N trees on random subsets of data	Aggregation of results of individual trees.
Neural nets	Either	Weights within net adjusted to minimise output errors for known data.	Net applied to the new data.

Figure 11.1 illustrates the principle of KNN, in a case where there are just two input features, and hence a 2-dimensional space for input data items, and the output classification is either a triangle or a star. With $k = 3$ nearest neighbours being considered, this means the majority classification of the 3 nearest training data points to a new data item will be used as the new item classification.

Distance measures are usually some variant of Euclidian distance $\sqrt{((\Delta x_1)^2 + \dots + (\Delta x_n)^2)}$ between points in the n -dimensional input feature space.

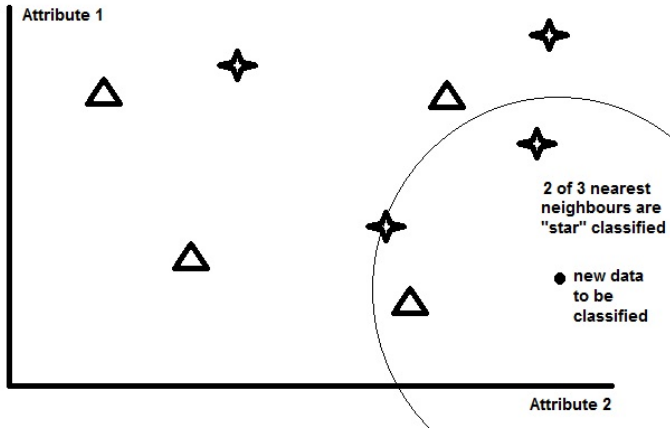


Fig. 11.1 KNN with 2 attributes and $k=3$

Decision trees build a tree of if-else conditional tests on the feature values of input data, with the leaves of the tree assigning classification results. Conditions on features may be equality tests $f = value$, or comparisons $f < value$, $f > value$, etc. The conjunction of all conditions on a branch that has classification result R represents one set of learnt criteria for assigning an input item to category R .

A *random forest* is a form of ensemble learning using a collection of decision trees (Figure 11.2), in which the different trees are intended to learn the same function, but they are trained using separate random samples of the training dataset. This produces diversity in the resulting trees, and helps to avoid the problem of *overfitting* of individual decision trees to a training dataset, i.e., the learning of rules which are too specific to the training data and that do not generalise well to other data items.

The simplest forms of neural net are *Multilayer Perceptrons (MLP)*: neural networks with an input layer, one or more hidden layers, and one output layer (Figure 11.3). Typically there is an input node for each data feature, and an output node for each possible classification or category for data items. There is a connection between each pair of nodes in adjacent layers. Connections only transmit information from the input side of the net to the output side (a *feed-forward* network). Each connection between nodes i and j has a numeric weight $w_{ij} \in [-1, 1]$. Each hidden layer and output node j computes some function f of the weighted sum of its inputs:

$$out_j = f(\sum_{i=1}^n in_i * w_{ij})$$

A typical f is the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$ or hyperbolic tangent $f(x) = \tanh(x)$.

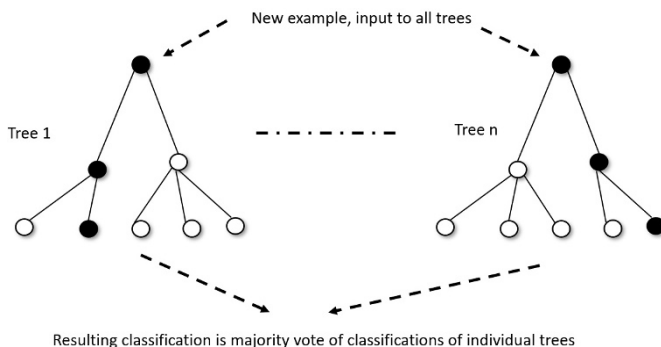


Fig. 11.2 Random forest for classification (Wikimedia Commons, author Venkata Jaganath)

Learning consists of applying training data to the inputs of the network, and adjusting the weights w_{ij} of the connections between nodes to minimise the errors in the outputs compared to the known classifications of the training data items. The term *loss* is used to denote the error between the predicted and actual classifications. Training continues until an acceptable level of loss is achieved. Once training is completed, new items are classified according to the output node activated when they are input to the network. In general, the trained network is a non-linear function of its inputs, and the precise function depends upon the connection weights and the combinator functions used at the nodes. The topology of the network, in terms of the number of hidden layers and their sizes, is an example of a ML *hyperparameter*, set by the ML designer. These are distinct from the internal model parameters which are affected by learning, such as the connection and node weights.

The original forms of neural net were feed-forward networks, with connections only in one direction from input to output. More complex forms of neural network incorporate reverse connections, to provide memory mechanisms to process ordered sequences of data items, such as words in a text sequence:

- *Recurrent Neural Networks (RNN)* additionally use the previous state of nodes as inputs. This can address sequence-prediction problems which a simple MLP cannot.
- *Long Short-Term Memory (LSTM)* can retain state over indefinite periods. LSTM has been widely applied to AI tasks including speech recognition and automated translation. It is appropriate for classifying and making predictions about time-series data.

Input data for these forms of neural net typically consist of a sequence of items, each item represented as a feature vector.

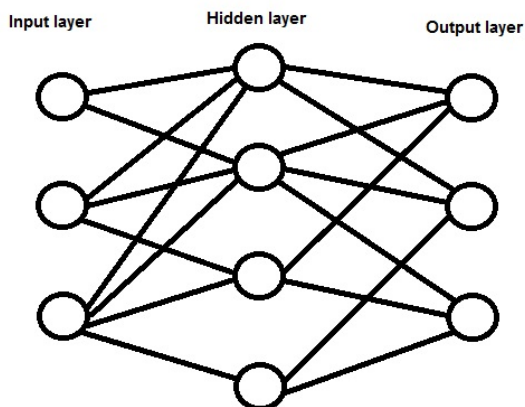


Fig. 11.3 MLP with one hidden layer

Recent advances in the power of ML systems have arisen due to the use of *deep learning* model architectures, involving multiple internal layers of neurons and encoder/decoder [19] processing of inputs/outputs. A key feature of many deep learning systems is the *attention mechanism* [60], whereby the ML system can learn which parts of an input sequence (such as an encoded natural language sentence) are most significant for generating output. The *Transformer* architecture is based primarily on attention [90], and has been used as the basis of the ‘Generative pre-trained transformer’ (GPT) and BERT [24] families of applications such as ChatGPT. The underlying models are trained using unsupervised learning on very large datasets of natural language texts, and are termed *large language models* or LLMs [93]. They can then be specialised for particular tasks by supervised retraining. The main paradigm for these applications is *translation* of information from one system of knowledge to another. For example, machine translation from English to Chinese, or summarisation of English text, or translation from molecular formulae to molecular structures. The training time/resources and dataset sizes required by LLMs may be extremely large. Figure 11.4 shows the growth in computational resources needed to train some example ML systems, and Figure 11.5 the size growth of ML models in terms of the number of internal connection weights and other parameters. Training time for LLMs has grown from hours to days and then to hundreds of days, whilst parameter size has grown to over 1000 billion, and input data size increased to over 1000 billion tokens [93].

It is important to realise that despite their considerable power, these types of systems are fundamentally *stochastic* or probabilistic in nature. Thus there will always be cases where the system gives incorrect results, and these can occur unpredictably.

Fig. 11.4 Growth in computational resources used by ML training (From openai.com/research/ai-and-compute)

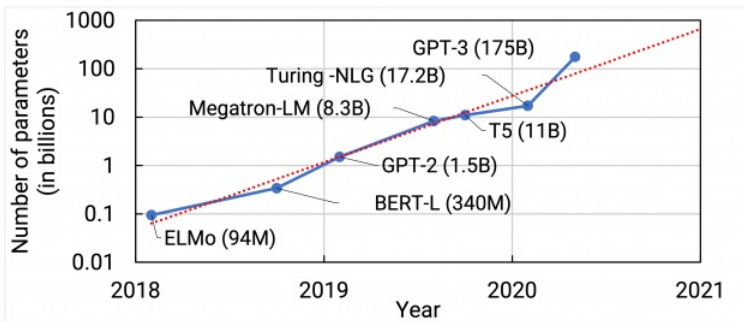
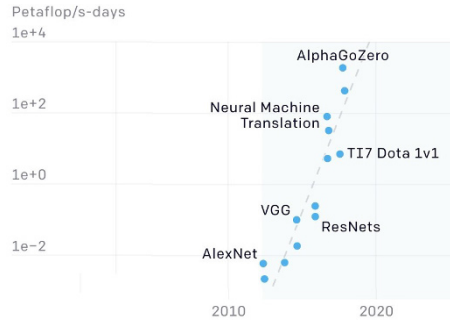


Fig. 11.5 Growth in ML model complexity (From towardsdatascience.com)

Apart from nonsymbolic ML approaches such as neural nets, there are also symbolic ML approaches, such as inductive logic programming (ILP) [65] or programming by example (PBE) [35]. These have the advantage of learning explicit knowledge in the form of symbolic rules, and usually require substantially less training data than neural net approaches. Unlike the stochastic ML approaches, they can in principle achieve 100% correctness. However they are limited to cases where precise rules do exist.

11.1.2 ML system construction and operation

There are generally several stages in the production of a ML system:

1. Data collection and preparation, which may include *normalisation* of data (e.g., ensuring that all images in an image dataset have the same dimensions), discarding or completing null information, and labelling of data in the case of supervised learning.

The Pipes-and-filters pattern (Chapter 5) is relevant for organising data pre-processing stages.

Analysis of the quality and provenance (origin) of the data is also important, as any flaws or biases in the training data will affect the quality of the trained model.

2. Feature identification and selection, and transformation of input data into vectors (or sequences of vectors) in the feature space.
3. Training, which can also involve adjusting the hyper-parameters of the learning algorithm used and the configuration of the ML technique (such as the number of hidden layers in a neural net) in order to produce an acceptable level of accuracy.
4. Validation: to test the accuracy and robustness of the trained system on new input data.
5. Deployment: creating a production version of the system for operational use.

During operation of the trained ML system it will perform inference of outputs when presented with new input data. Typically, raw input data will need to be vectorised into feature vectors or encoded in the same way that the training data was processed, using the same pre-processing pipeline, before presenting the inputs to the ML model. Likewise, outputs may need to be translated into usable forms. These processing steps can be carried out by auxiliary helper components managed by a controller component (Figure 11.6).

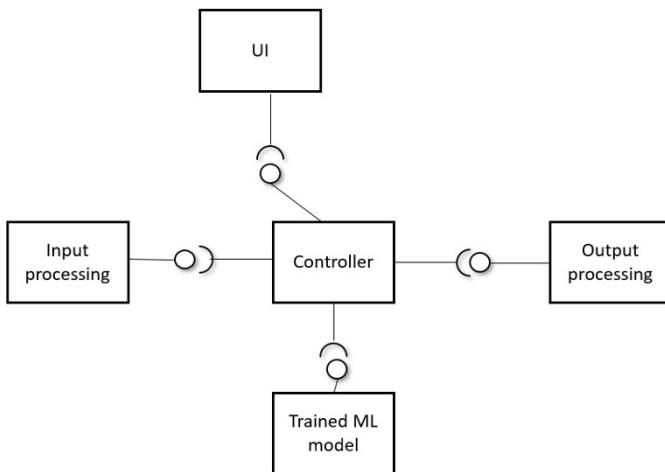


Fig. 11.6 Operational architecture of an ML system

A trained ML model can also be transferred or fine-tuned to apply to a different but related task. This is a form of reuse, and reduces in principle the cost of training for the new task.

11.2 Machine learning example: requirements classification

This ML example uses a decision tree classifier to learn rules for categorising natural language requirements statements (about software model transformation) into one of six categories, depending upon the syntax and terminology of the input statement. The training data is presented as a list X of data items represented as feature vectors, and a list Y of the expected classifications for each item:

```
from sklearn import tree
from sklearn.tree import export_text

## Classification categories:
## 0 = simple class mapping
## 1 = class/feature mapping
## 2 = conditional simple class mapping
## 3 = conditional class/feature mapping
## 4 = simple feature mapping
## 5 = other

# Training data:
X = [[1,1,0,0,1,0,0], ... ]

Y = [0, ... ]
```

There are 7 input features, each of which has a binary (0 for false, 1 for true) value:

- “sourceClass” – the name of a source model entity occurs in the statement.
- “targetClass” – the name of a target model entity occurs in the statement.
- “sourceFeature” – the name of a source model attribute or relation occurs in the statement.
- “targetFeature” – the name of a target model attribute or relation occurs in the statement.
- “mapsVerb” – a verb synonymous with ‘maps’ occurs in the statement.
- “assignsVerb” – a verb synonymous with ‘assigns’ occurs in the statement.
- “if/when” – some conditional marker occurs in the statement.

The raw training data consisted of about 70 different requirements statements together with their classifications. Data pre-processing to convert this raw data into feature vectors and expected classifications was performed manually in this case, but in general this step should be carried out by an automated procedure.

The decision tree ML algorithm from sklearn was used to induce a classifier from the pre-processed training data using the *fit* command. The maximum

depth for the induced tree (*max_depth*) is set to 6. This is an example of a hyperparameter for this ML algorithm:

```
clf = tree.DecisionTreeClassifier(random_state=0, max_depth=6)
clf = clf.fit(X, Y)
```

The learned tree can be displayed with names for the features via the commands

```
r = export_text(clf, feature_names=["sourceClass",
    "targetClass", "sourceFeature", "targetFeature",
    "mapsVerb", "assignsVerb", "if/when"])
```

```
print(r)
```

The derived decision tree is:

```
|--- assignsVerb <= 0.50
|   |--- if/when <= 0.50
|       |--- sourceClass <= 0.50
|           |--- mapsVerb <= 0.50
|               |--- class: classMapping
|               |--- mapsVerb > 0.50
|                   |--- class: unknown
|           |--- sourceClass > 0.50
|               |--- class: classMapping
|   |--- if/when > 0.50
|       |--- targetFeature <= 0.50
|           |--- class: conditionalClassMapping
|           |--- targetFeature > 0.50
|               |--- sourceFeature <= 0.50
|                   |--- class: conditionalClassMapping
|                   |--- sourceFeature > 0.50
|                       |--- class: conditionalClass/
                           FeatureMapping
|--- assignsVerb > 0.50
|   |--- if/when <= 0.50
|       |--- mapsVerb <= 0.50
|           |--- sourceFeature <= 0.50
|               |--- class: unknown
|               |--- sourceFeature > 0.50
|                   |--- class: class/FeatureMapping
|           |--- mapsVerb > 0.50
|               |--- class: class/FeatureMapping
|   |--- if/when > 0.50
|       |--- class: conditionalClass/FeatureMapping
```

It can be seen that the form of the verb is the most significant factor in sentence classification, followed by the presence/absence of a conditional indicator.

The tree can be applied to new data items (pre-processed using the same procedure as for training data) via the *predict* command:

```
clf.predict([[1,1,0,0,1,0,0]])
```

The accuracy of the classifier was evaluated on a test set of 28 new sentences. Applied to this set the classifier correctly classified 22 sentences (0.79%).

11.3 Machine learning systems architectural issues

The following architectural flaws and antipatterns are common in ML systems [13, 81]:

- *Correction cascades*: this flaw occurs when the output of a model is post-processed in an ad-hoc manner in order to correct specific errors in the ML outputs. This means that changes to the ML model to improve its outputs will necessitate further changes to the correction cascade, and may result in a worse behaviour overall.
- *Feedback loops*: these occur when an ML system influences its own behaviour through some form of direct or indirect feedback route. This can result in the ML system learning an incorrect target function, where the fed-back information overrides significant input features.
- *Glue code*: this involves dependencies of the ML system on external libraries/packages which require excessive adaptation to use. Typically the packages provide many more facilities than are actually used, violating the ISP of the clean architecture. Direct dependence of the ML application on specific libraries also violates the dependency rule.
- *Pipeline jungles*: complex data pre-processing pipelines, which cost effort to maintain and become fragile and liable to errors as more stages are added.
- *Dead experimental codepaths*: code from prototyping of the system is retained in the production system, even though it is no longer used. This is a case of dead/obsolete code, which has been the cause of several major ‘computer-aided disasters’ such as the Ariane 5 failure (Chapter 10) and the Knight Capital bankruptcy [82].
- *Garbage features*: features which do not contribute to the ML performance but have been retained from previous versions where they were significant. Detecting and removing such features can be a substantial effort.

Other code-level flaws identified by [81] are (1): basic datatypes such as doubles and integers being used without information about what they repre-

sent; (2) multiple programming languages used in one system; (3) prototyping environments being used for production systems.

The general remedy for these types of flaws is to increase the modularisation and encapsulation of parts of the ML system, in particular to separate data loading and manipulation into their own component(s), called via an interface from the main application module. Post-processing correction actions can also be separated into a dedicated component, or the ML model retrained to learn the correct mapping without the need for post-processing. Both pre- and post-processing chains can be organised using the Pipes-and-filters architectural style.

External packages should be used via interfaces and gateway adapters, with dependency inversion used to reduce dependencies.

11.3.1 Language-specific issues: Python

Python is the main language used for ML system construction, because the Python ecosystem provides established toolsets such as NLTK (for natural language processing), and PyTorch and Tensorflow for general ML.

However there are various issues with the language, which affect software quality:

- *Implicit typing*: the type of a variable is not fixed, and depends on the data that is held in the variable. Thus the programmer has to ensure that variables are used correctly, and that appropriate operations are applied to variables.
- *Implicit declarations*: variables are not explicitly declared, instead they are usually declared by the first assignment to them.
- *No interfaces*: Only classes can be defined. Again, whether a class satisfies a conceptual interface can only be determined by the programmer checking that all the expected operations are implemented in the class.
- *Multiple/non-standard inheritance*: The Python version of inheritance differs from standard OO inheritance as found in languages such as Java and C#, in particular a form of multiple inheritance is supported. This can lead to programmer mistakes and semantic errors which are difficult to detect.
- *Language evolution*: Python changed substantially from Version 2 to Version 3, and these are incompatible versions. For example, the *print*, *raise* and *except* statements have different syntax in the two versions, and integer division is denoted differently in the two versions. It is a non-trivial task to migrate large systems from Python 2 to Python 3 [79].

A consequence of implicit declarations is that it is easy to make programming errors that concern unintended confusion of local and global variables with the same name. To address this problem, Python provides a *global* key-

word enabling the programmer to explicitly assert that a local assignment does not create a new local variable, but instead updates a global variable:

```
def f(y) :  
    if y > 0 :  
        global x  
        x = 2  
    else :  
        pass  
  
f(1)  
print(x)
```

The printed result is 2, but without the *global* keyword the program is invalid. Clearly, this is unsatisfactory from a software maintenance perspective, and occurrences of the *global* and the related *nonlocal* keywords should be counted as program flaws. Various tools exist to identify quality flaws in Python programs, of which Pylint¹ is the best known.

Python *type annotations* can be used to document the (intended) type of a parameter or variable. The notation

```
v : typ = val
```

annotates a new variable *v* with a Python type name such as *bool*, *str*, *int*, *float*. Similarly, a parameter can be declared as *p : typ*. However the correctness of the type annotation is not semantically checked by Python compilers, so the annotation only serves as a comment.

11.4 Trustworthiness of ML systems

Because ML systems are increasingly being used in critical applications such as medical advice software and autonomous vehicles, and used pervasively for knowledge-related tasks, the degree to which the systems can be trusted is a significant quality of service property that should be considered. This property has several dimensions:

- Accuracy: degree to which correct results are produced by inference.
- Bias: avoidance of producing differential results for individuals or groups that are distinguished only by protected characteristics (gender, age, race, etc).
- Safety: avoidance of hazards resulting from the ML system operation and outputs.
- Explainability: the ability to explain the output of the ML system for given inputs.

¹ <https://pylint.readthedocs.io>

Various measures have been proposed to quantify these properties, such as the *average odds difference* and *statistical parity difference* measures of fairness [10].

In the case of safety, the concept of a protection system can be used (Chapter 10): the outputs of the ML system can be monitored, and a protection system activated if the outputs could result in an unsafe state, with the protection system acting to reduce or eliminate the hazard. For example, to override a clearly erroneous medical intervention recommended by an ML system. Harmful textual outputs from LLMs can be mitigated to an extent by additional fine-tuning of the models using human feedback [93].

For explainability, some ML techniques have inherently good explainability, such as decision trees and KNN – in these cases the reason for a given output can be explained in terms of the explicit details of the learnt model. However for neural nets, and particularly for complex NN architectures, the operation of the net is opaque and only general information about the degree of influence of different input features can be obtained.

It is important also to realise that the above quality properties may conflict, e.g., the most accurate algorithms tend to be the least explainable [39].

Summary

This chapter has considered the architectural issues involved in the construction of machine learning systems, and identified how clean architecture principles can be applied to improve the maintainability of such systems. Other quality aspects such as fairness and explainability have also been considered.

Exercises

1

For each of the following tasks, explain if a symbolic or nonsymbolic ML approach would be the most suitable:

1. Image classification of pet photos into categories of *cat*, *dog*, *other*.
2. Classification of meals into vegetarian or non-vegetarian based on their ingredients.
3. Classification of meals into vegetarian or non-vegetarian based on photos of the meals.
4. Translation of English text to French text.
5. Translation of Python programs to Java.

2

How many parameters does the network of Figure 11.3 have, if each connection and each non-output node has 1 parameter?

3

What are the number of parameters in a feed-forward MLP with two hidden layers of H neurons each, F input neurons and C output neurons, with connections between every node of successive layers L_i and L_{i+1} for $i = 1, 2, 3$?

4

Express a ML data pre-processing pipeline in the Pipes-and-filters architectural style, where there are successive filters for Quality, De-duplication, and Anonymisation (i.e., removal of personally-identifiable data).

5

Compare the explainability of the following ML approaches, for the learning of a binary classification problem based on 20 input features:

1. KNN with $K = 5$.
2. A single decision tree.
3. A random forest consisting of 10 trees.
4. A feed-forward MLP with one hidden layer of 10 nodes.



Chapter 12

Software Architectures and Re-engineering

Abstract In this chapter we consider the role of software architecture in the context of the re-engineering of legacy software systems, and in particular we examine how quality flaws in the legacy systems can be corrected and the system architectures improved.

12.1 Legacy software systems

Legacy software systems are usually retained by an organisation because they perform essential business functions, but their maintenance typically causes significant business costs because these systems often possess substantial technical debt and quality flaws accumulated over a long lifetime. They will have been worked upon by many different developers, and are often poorly-documented.

Starting in the mid 1980's-early 1990's, various model and specification-based approaches were proposed for reverse-engineering legacy systems, to (re)create precise documentation from their code, and to restructure legacy systems and improve system quality, as a means of prolonging the life of the system or as a step towards the production of a modernised system, potentially in a different programming language or platform [84, 14]. These concepts of model-driven modernisation (MDM) were formalised by the OMG in their Architecture-driven modernisation (ADM) approach [74]. ADM supports in principle a 'horseshoe modernisation model' whereby the starting point is the legacy system source artefacts at a low level of abstraction, these are then reverse-engineered to higher levels of abstraction, e.g., to Platform-specific, Platform-independent and Computation independent models (PSMs, PIMs, CIMs), and then forward-engineered to a target platform.

Our re-engineering approach uses an agile MDE process, based on iterations of the steps shown in Figure 12.1. This approach can be used with a Scrum-based process whereby stakeholder re-engineering requirements are

expressed as user stories and worked on by the main re-engineering team in a series of sprints. In parallel, necessary tool adaptation and extension are carried out by a supporting team.

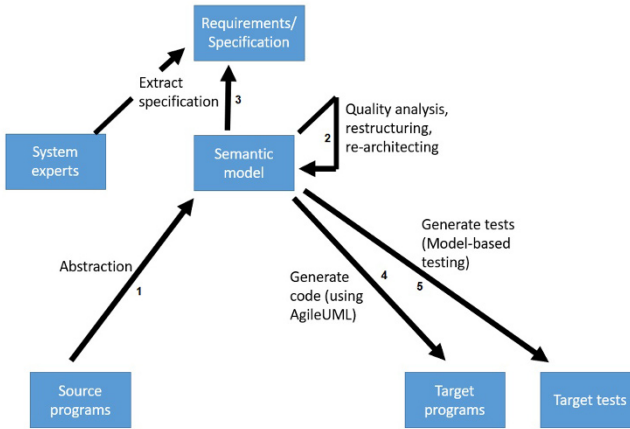


Fig. 12.1 Agile model-driven re-engineering process steps

Step 1 uses a combination of the ANTLR parser engine¹ and the *CSTL* text-to-text transformation language of [50] to translate source code to system abstractions in UML and OCL (Section 12.3).

Step 2 performs semantics-preserving refactoring operations on the abstracted specification, in order to remove or reduce quality flaws identified by quality metrics/flip indicators (Section 12.4). The system architecture can also be refactored to remove violations of clean architecture principles such as ISP and ADP (Section 12.4.3).

Step 3 involves input from system experts to identify which system requirements are implemented in which parts of the abstracted system model. A high-level system requirements specification can be produced.

Steps 2 and 3 are optional and need not be performed if the aim of re-engineering is a direct program translation of the system to a different programming language.

Steps 4 and 5 generate an executable system in a new platform, together with a test suite for the system (Section 12.5). For example, financial systems written in Visual Basic can be migrated to Python, or COBOL business systems migrated to Java or C#.

¹ <https://antlr.org>

12.2 Challenges of legacy code abstraction

Legacy code can be of recent vintage, as with the Athena system, written in Python 2 [79]. As we discussed in Chapter 11, machine learning systems in particular may have embedded technical debt which can lead to increased software operational costs and reduced effectiveness [81, 92]. The widespread use of agile methods has also led to recognition of the problems that these may cause in terms of increased technical debt and maintenance costs [9]. It is estimated that technical debt consumes over 40% of software developer effort [87].

Flaws in code can also result in wasted energy resources, due to poor efficiency or increased binary size due to dead code [23, 70, 85]. The high carbon footprint of ML systems means that improvements to their efficiency could produce significant reductions in CO₂ emissions [66].

Here we will focus on the reverse-engineering of code in old languages such as COBOL [77] and BASIC [45], which cause particular problems because of the antiquated aspects and features of these languages.

12.2.1 BASIC and variants

BASIC² was intended, as its name suggests, as a language for inexperienced programmers to use for relatively simple programming problems. It became popular with the advent of home PCs in the 1970s, and as Visual Basic (VB) and Visual Basic for Applications (VBA) became the main language for defining auxiliary code modules within MS applications such as Excel [64].

The principal challenges for software modernisation and re-engineering of VB/VBA are:

- The use of implicit typing for data items.
- GOTO statements.
- The large number of different kinds of statements (67 in VB6).
- The complexity of MS applications such as Excel, with complex spreadsheet data and hundreds of application functions, which can be called from VBA code.

As an example of VB6/VBA program analysis we consider an Excel spreadsheet and embedded VBA functions to support option pricing using both the Black-Scholes formula [12] and binomial tree estimation (Figure 12.2).

The dependence of VBA code on MS application data and functionality means that often the program code cannot be understood in isolation from

² Beginner's All Purpose Symbolic Instruction Code

Fig. 12.2 Excel/VBA example

E9						=EBinCall(B2,B3,B4,B5,B6,B7,B9)
	A	B	C	D	E	F
1	Parameters					
2	Underlying price (S)	100				
3	Strike price (K)	40		BS Call price	61.47642	
4	Volatility (Vol)	35.00%		BS Put price	0.0042	
5	Interest rate (r)	5%				
6	Dividend yield (q)	0%				
7	Time to maturity (T)	0.75				
8						
9	Number of steps	60		Binomial Call	61.47599	
10						
11	u	1.039907				
12	d	0.961625				
13	p	0.498205				
14	p'	0.517763				
15	a	18				
16						

the host application. For example, a VBA program embedded in an Excel spreadsheet may refer directly to the content of a spreadsheet cell by its coordinates *XY*:

```
Sheets("MyWorksheet").Range("XY")
```

The meaning of this value depends upon the spreadsheet context and the role of cell *XY* in the spreadsheet. In addition, VBA code can invoke any of the Excel worksheet functions, which include statistical and financial functions in addition to a substantial set of core mathematical operations. Examples of VBA functions used in the option pricing application are the following auxiliary Black-Scholes functions:

```
FUNCTION D1(S AS Double, K AS Double, Vol AS Double, R AS Double,
    Q AS Double, T AS Double) AS Double

    D1 =
        (WorksheetFunction.Ln(S / K) +
         (R - Q + Vol * Vol / 2) * T) / (Vol * Sqr(T))
END FUNCTION

FUNCTION D2(S AS Double, K AS Double, Vol AS Double, R AS Double,
    Q AS Double, T AS Double) AS Double

    D2 = D1(S, K, Vol, R, Q, T) - Vol * Sqr(T)
END FUNCTION
```

D1 uses the Excel worksheet logarithm function *Ln*.

12.2.2 COBOL

COBOL (c. 1959) predates BASIC, and was specifically intended as a language for commercial data-processing applications. The main standard versions are COBOL ‘74 and ‘85. Legacy COBOL presents a wide range of challenges for software modernisation/re-engineering:

- No general data types – instead data items are assigned a specific format at the byte level.
- Extensive use of aliasing and redefinition of data items.
- GO TO and ALTER statements.
- A primitive procedure call mechanism, PERFORM, which is vulnerable to many control flow anomalies [22].
- Multiple variants and formats of statements.

A COBOL program is structured into four main parts: (i) an *identification division*, (ii) an *environment division* (defining in particular the link between data storage devices and logical files), (iii) a *data division* (defining all program data) and (iv) a *procedure division* (defining program behaviour). In turn, each division is divided into sections, and in the procedure division, further divided into *paragraphs*, *sentences* and *statements*. Paragraphs are a key program unit, consisting of a name and a sequence of sentences. Paragraphs may be both the target of a GO TO, and invoked via a PERFORM. Thus at least two separate semantics need to be expressed for each paragraph P :

1. The semantics encountered by invoking P by *PERFORM P*.
2. The semantics encountered by a *GO TO P*, or (equivalently) by ‘falling through’ into P from a preceding paragraph/section.

An example COBOL program, with two paragraphs, each containing two sentences, is:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBOLPERFORMSOURCE.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 A PIC 999 VALUE 1.
PROCEDURE DIVISION.
P1.
    PERFORM P2.
    PERFORM P2.
P2.
    ADD A TO A.
    ADD 100 TO A.
```

This program terminates with $A = 708$.

12.3 Program abstraction using ANTLR and *CSTL*

ANTLR³ is a parser generator which takes as input a BNF-format grammar file *LParser.g4* defining the grammar of a software language L , and pro-

³ <https://antlr.org>

duces an executable parser for L . Here we utilise the VisualBasic6.g4 and Cobol85.g4 grammars from antlr.org. The output of the parsers are parse trees of the form $(tag\ t_1 \dots t_n)$ where tag is the grammar rule name (grammar non-terminal) which produced the tree, and the t_i are the immediate subterms.

$CSTL$ is a text-to-text (T2T) transformation language, based on pattern-matching and text substitution. It can be used for specifying code generators and other language-to-language mappings. Here we use $CSTL$ to abstract COBOL and VB6 programs to UML/OCL. A $CSTL$ script for processing a language L consists of a set of named rulesets, and each ruleset consists of an ordered set of rules. When a ruleset has the same name as a grammar non-terminal tag of $LParser.g4$, then it will process parse trees $(tag\ t_1 \dots t_n)$ produced by the corresponding grammar rule of $LParser.g4$ (i.e., by execution of the ANTLR parser for L). Other rulesets can be defined to carry out user-defined auxiliary functions.

Unlike template-based code generation languages such as EGL [25] and Acceleo [1], $CSTL$ uses templates for matching source elements, in addition to templates for constructing target text. This is particularly useful when the source language L contains many different elements which can be syntactically distinguished and need to be translated using different strategies. For example, Cobol85 and VB6 each have over 50 different kinds of procedural statement. $CSTL$ avoids the use of special delimiter characters to separate source and target text. Instead the LHS of $CSTL$ rules

LHS $|-->$ RHS

are expressed in the schematic concrete syntax of the source language, and the RHS is expressed in schematic concrete syntax of the target language. Metavariables $_i$ for $i : 1..99$ can be used on both the LHS and RHS, on the LHS they denote some source language item (term in an abstract syntax tree) and on the RHS the translation of that item by the current $CSTL$ script. E.g., the rule

$_1 \text{ mod } _2 |--> _1 \% _2$

defines translation of OCL modulus expressions to Java. Rules may also have conditions $\langle \text{when} \rangle\ Cond$, and actions $\langle \text{action} \rangle\ Act$.

Representation of program semantics in UML/OCL uses extensions of the OCL 2.4 standard [71], in particular a procedural extension of OCL, similar to the SOIL formalism [16], is used to represent procedural statements. Libraries *OclFile*, *OclDate*, *OclProcess*, *OclType* and *MathLib* for common programming data types and facilities have also been defined [54]. For COBOL and VB6 we require further library operations, types and components:

- Library operations for financial functions such as (net) present value, annuity, internal rate of return, etc.
- Operators $\rightarrow roundTo$ and $\rightarrow truncateTo$ generalising $\rightarrow round$ and $\rightarrow oclAsType(int)$.

- An *OrderedMap* data structure, which enables access to values both by key and by a positional index. This can be simulated by the type *Sequence(Map(String, OclAny))*.
- For VB, a library class *Excel* to represent the data and functions of Excel spreadsheets.

12.3.1 VB6/VBA

Table 12.1 summarises the abstraction strategy for VB6/VBA code.

Table 12.1 Abstraction mapping from VB6 to UML/OCL

<i>VisualBasic construct</i>	<i>Abstraction in UML/OCL</i>
module	package
record	class
record field	attribute
function, subroutine	operation
enum definitions	enumerated types
assignment, sequencing, blocks	assignment, sequencing, blocks
calls of built-in functions/subroutines	operation calls
if, while, return	if, while, return
do, for-next	while
for each in	for
select	if
error	error
exit loop	break
Integer, Byte, Long	int
LongLong	long
Single, Double	double
String	String
Array types	Sequence types
Collection	Sequence(Map(String,OclAny))

We can identify from the VB documentation [64] that the VB6 *Long* type corresponds to OCL *int*, and VB6 *Double* to OCL *double*. We can identify that numeric comparator expressions should translate directly to corresponding expressions in OCL. The operator *&* of VB6 corresponds to string concatenation *+* in OCL. The specialised operator *LIKE* of VB6 corresponds to the exact regular-expression matching operator *→isMatch* of OCL [52], whilst *IS* corresponds to reference equality *<>=*.

The VB6 grammar definition for the *valueStmt* non-terminal includes the cases:

```

valueStmt:
...
| valueStmt WS? AMPERSAND WS? valueStmt

```

```

| valueStmt WS? (EQ | NEQ |
  LT | GT | LEQ |
  GEQ | LIKE | IS) WS? valueStmt

```

Thus the corresponding abstraction ruleset *valueStmt::* in *CSTL* must have rules for each of the 9 binary operators of these cases, which express the identified semantic mapping:

```

valueStmt::
...
_1 & _2 |-->(_1 + _2)
_1 = _2 |-->_1 = _2
_1 <> _2 |-->_1 /= _2
_1 < _2 |-->_1 < _2
_1 > _2 |-->_1 > _2
_1 <= _2 |-->_1 <= _2
_1 >= _2 |-->_1 >= _2
_1 LIKE _2 |-->(_1)->isMatch(_2)
_1 IS _2 |-->_1 <=> _2

```

In general the structure of the source code is retained in the abstraction, facilitating traceability.

In the cases of typeless variable declarations

```
DIM Var
```

type inference based on the values assigned to *Var* and the processing performed on it are used where possible to deduce the specific type of *Var*. In the absence of specific typing knowledge, the ‘universal’ type *OclAny* is used for *Var*.

12.3.2 Cobol85

Table 12.2 summarises the abstraction strategy for Cobol85.

The data parts of a COBOL program are abstracted as follows:

- Data declarations in the file section, linkage section and working storage section are represented as attributes of a class *Prog_Class* representing the COBOL program.
- The OCL type of a data item is obtained from its COBOL format (PIC/PICTURE format specifier). For example, format 9(4) denotes that the item can consist of up to 4 decimal digits, corresponding to an *int*, and 9(10) corresponds to *long*, whilst numeric formats such as 99V99 which contain a decimal point position V are represented as *double*. A sign indicator S can precede the numeric picture characters. Composite items do not have a PIC/PICTURE, and are given a *String* type.

Table 12.2 Abstraction mapping from Cobol85 to UML/OCL

<i>Cobol85 construct</i>	<i>Abstraction</i>
Program suite	package
Program	class
record	class + invariants
record field/program variable	attribute
section, paragraph, subprogram	operation(s)
arithmetic statements, move, set	assignment
sequencing, in-line performs	sequencing, blocks
calls of subprograms, simple	operation calls
PERFORM, PERFORM THRU, GO TO	
IF, EVALUATE	if
PERFORM UNTIL/varying, exit	while, return
simple PERFORM TIMES	for
GO TO DEPENDING ON	if + calls
on size error	tests for overflow conditions
inspect, string, unstring	String/regex operations
stop run	OclProcess.exit(0)
Numeric data formats 9(N) for $N \leq 9$	int
Numeric data formats 9(N) for $N > 9$	long
Other numeric data formats	double
Alphabetic/alphanumeric formats	String
Array (table) types	Sequence types
Files	OclFile, sequences, maps

- Implicit or explicit aliasing relations between data items are expressed as *invariants* of *Prog_Class*. For example, the declaration

```

01 REC.
  02 FLD1 PIC XX.
  02 FLD2 PIC 9999.

```

defines three variables *REC*, *FLD1* and *FLD2*, with *REC* and *FLD1* being of string type and *FLD2* of integer type. *REC* is the concatenation of *FLD1* and *FLD2*, and *FLD1* is the first two characters of *REC*, whilst *FLD2* is the integer value of the final four characters of *REC*. Thus the semantics of the declaration is expressed as:

```

attribute REC : String;
attribute FLD1 : String;
attribute FLD2 : int;
invariant REC = "" + FLD1 + FLD2;
invariant FLD1 = REC.subrange(1,2);
invariant FLD2 = REC.subrange(3,6)->toInteger();

```

Likewise for *REDEFINES* and *RENAMES* relations on data, and 88 condition names.

The code generators of AgileUML for Java, C# and C++ enforce specification invariants $P \Rightarrow Q$, by synthesising update code so that any event

which makes P true also establishes Q [49]. Thus the invariants expressing the source code semantics of data relations can be implemented in the target platform.

- Files are abstracted as sequences together with a file position pointer. Indexed files also have an index map from file key values to file positions.

The procedure division is abstracted as follows:

- Each paragraph P is represented by two separate operations $P_Call()$, $P()$ of *Prog_Class*. $P_Call()$ represents the functionality of a *PERFORM P* statement, whilst $P()$ represents the fall-through/*GO TO P* functionality. $P()$ invokes $P_Call()$:

```
operation P()
pre: true post: true
activity: self.P_Call() ; self.Q() ;
```

where Q is the next paragraph following paragraph P , if any.

- A section is also represented by two operations in the same manner.

Additional paragraph representations may be necessary if the *PERFORM A THRU B* variant of *PERFORM* is used: each paragraph P from A up to B also has a version P_thru_B :

```
operation P_thru_B()
pre: true post: true
activity: self.P_Call() ; self.Q_thru_B() ;
```

where B_thru_B is simply B_Call .

An unusual feature of COBOL is the use of error clauses with arithmetic, string and file operations. Semantic representation of such clauses is carried out by introducing conditional tests for the error conditions. For example:

```
ADD X TO Y ON SIZE ERROR Stat.
```

is abstracted to:

```
Y := (X+Y)->truncateTo(M) ;
if MathLib.isIntegerOverflow(Y, N)
then Stat'
else skip
```

where N is the number of integer digits of Y 's format (decimal places before any decimal point) and *isIntegerOverflow(value, N)* tests if a numeric value has more integer digits than N . $Stat'$ is the translation of $Stat$. M is the number of fraction digits in Y 's format.

12.4 Quality analysis and restructuring

Once a source application has been abstracted to UML/OCL, it can be analysed for quality flaws and refactored or restructured to improve its quality. Performing restructuring at the specification level avoids the need to define multiple language-specific restructuring tools, and means that the improved structure applies to every target platform.

12.4.1 Quality analysis

AgileUML provides tools to analyse UML/OCL specifications to detect specification flaws analogous to ‘code smells’ in the sense of [28, 37]. These flaws can also be regarded as technical debt indicators [17, 62, 89]. Table 12.3 shows the code smells detected by AgileUML and the corresponding refactorings used to address these flaws. These extend the code flaws which we considered in Chapter 2.

Table 12.3 Code smells/flaws and refactorings

<i>Flaw</i>	<i>Description</i>	<i>Refactoring</i>
ENO	Excessive number of operations	Split class
EFS	Excessive operation/function size	Split operation
EPL	Excessive parameters of function/operation	Value object
CC	High cyclomatic complexity	Split operation
EFO	Excessive fan-out: high number of different functions/operations called from one function/operation.	Split operation
EFI	Excessive fan-in: high number of different functions/operations call one function/operation	Split class
CBR ₁	High number of client-supplier relations	Split class
CBR ₂	Cycles in the call graph	Replace recursion
DC	Duplicate code/expressions	Factor duplicates
UC	Unused/unreachable code	Remove code
UOP	Unused operations/functions	Remove operations
UVA	Unused local variables/parameters	Remove variables
MGN	Magic numbers: literal constant values in statements	Replace value by named constant
AIL	Average identifier length too low (≤ 2)	Define meaningful names
PMV	Same variable assigned values of different types	Define separate variables

The thresholds for these smells (e.g., a minimum size for clones) can be varied depending on the practices and internal standards of the client organisation.

Commented-out code lines can also be an indicator of maintenance problems. The percentage of such lines in the source can be measured by a simple text analysis tool. Another quality aspect is the presence of *self-declared technical debt*, i.e., code comments which indicate problems or unresolved issues in the source code, e.g., “TODO”, “FIX THIS” [61, 91].

12.4.2 Restructuring

Detected flaws can be removed or reduced by restructuring the specification, for example by splitting a large operation into parts to reduce EFS and EFO. EPL can be reduced by factoring out a group of two or more related parameters as a ‘value object’ class *VO* and replacing the group by a single *vox* : *VO* parameter. If several operations have the same *VO* parameter class after this step then they can be moved into an entity class based on the *VO* class.

The abstraction process described in Section 12.3 replaces GO TO jumps by function/operation calls, resulting in structured code which however may use recursion (introducing a CBR₂ flaw). Tail recursion can be removed by the “Replace tail recursion by iteration” code refactoring, which is applicable for tail-recursive operations defined either by OCL expressions or by statements.

Expressed in *CSTL*, the restructuring rules for this refactoring include:

```
operation _1 ( )\n  activity: _* ; self._1 ( ) ; |-->
operation _1()\n  activity: while true do ( _* );\n\n
```

In order to deal with mutually-recursive operations, it is also necessary to replace some operation calls by their definitions. This refactoring “Replace call by definition” replaces a call *self.op()* by the activity *Def_{op}* of *op*, within the activity of a different operation *op1*. It applies to operations *op* which do not return values, and which have no parameters. This refactoring converts a mutual recursion between *op1* and *op* into a self-recursive definition of *op1*.

DC can be reduced by factoring out cloned code as a new operation, and UC, UOP, UVA reduced by removing the unused elements. MGN is addressed by defining new constants with the literal value and replacing occurrences of the literal value by the constant name. Low AIL is addressed by replacing short names of one or two characters by longer semantically-meaningful names. In general, a 1-1 renaming of parameters and local variables is a semantics-preserving refactoring. ENO, EFI and CBR₁ can be reduced by splitting a class, extracting a cohesive subset of operations as a new component. We discuss this in the following section.

12.4.3 Rearchitecting

Rearchitecting attempts to produce an improved software architecture for a legacy system. This can involve segregating platform-specific code from business logic code, and by replacing direct calling dependencies between components by indirect connections via interfaces.

Typically, legacy VB6 and COBOL programs are defined in a monolithic architectural style, with a single module/program unit containing all code elements as a collection of operations/functions, together with global scope data definitions. In order to identify meaningful subcomponents of these systems, we can analyse the calling dependencies of operations on each other (the program call graph) and the dependencies of operations on global data items.

This information can be used to automatically modularise a system according to metrics of cohesion and coupling between components, using an optimisation procedure [15]. There are two main measures: of *coupling* and *cohesion*. *Coupling_{i,j}* measures how many data and calling dependencies exist between different components C_i , C_j , whilst *Cohesion_i* measures how many dependencies exist within an individual component C_i . Here we only consider calling dependencies:

$$Cohesion_i = \frac{\text{Calling dependencies in } C_i}{N_i * (N_i - 1)}$$

where $N_i > 1$ is the number of operations in C_i .

$$Coupling_{i,j} = \frac{\text{Calling dependencies between } C_i, C_j}{N_i * (N_j - 1)}$$

for $N_i > 0$, $N_j > 1$.

The optimisation process tries to maximise the quality measure

$$CRA-Index = \sum_i Cohesion_i - \sum_{i,j | i \neq j} Coupling_{i,j}$$

Modularisation and component identification to improve the *CRA* measure can be performed via a set of heuristic rules:

1. An operation f is placed in the same component as operation g if g is the only caller of f and g is itself called by some operation.
2. Operations with no caller remain in the parent/original component.
3. Operations in the same cycle of call dependencies are placed in the same component.
4. Other unallocated operations are placed in a component which has a maximal number of calling dependences on the operation.
5. Components with a common element are merged, as are mutually-dependent components.
6. Data items which are only referenced by a component C are placed in C , otherwise the item is placed in a data layer component, which is depended-

on by all components that use global data. The data layer has the responsibility to maintain data integrity constraints such as the invariants derived by COBOL abstraction (Section 12.3.2).

Cohesion is increased by these rules, however in some cases it may be preferred to group together operations which have few or no dependencies upon each other, but which are used by several common clients: this is the case for library operations, for example.

Figure 12.3 shows an example of the application of the heuristics to create components $C0$ to $C3$. An arrow $e \rightarrow f$ denotes that operation e has a calling dependency on f . Identified components are marked by red rectangles.

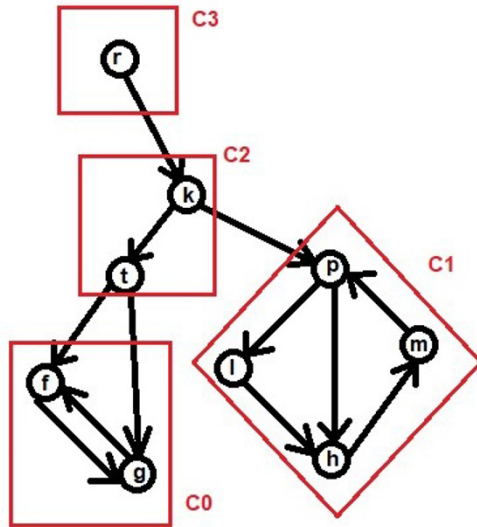


Fig. 12.3 Rearchitecting example

Each identified component C can be further transformed by extracting an interface CI which declares the externally-used operations of C . Client components then use C 's operations via CI . The operations of C_j required by C_i are gathered into a required interface $C_i_C_j_Req$ of C_i , which is extended by the provided interface C_jI of C_j (Figure 12.4).

Another hindrance to maintenance is the intermixing of business logic code together with data management code using a specific technology, e.g., direct invocation of Excel or SQL commands within business logic [63]. This design flaw leads to high maintenance effort/costs when there is technology change, such as a move from a relational to a NoSQL database. As a result of the abstraction process, file and database operations are represented by calls of the library components *OclFile* and *OclDatasource*, together with *SQLStatement*, and Excel operations are represented by calls of the *Excel* li-

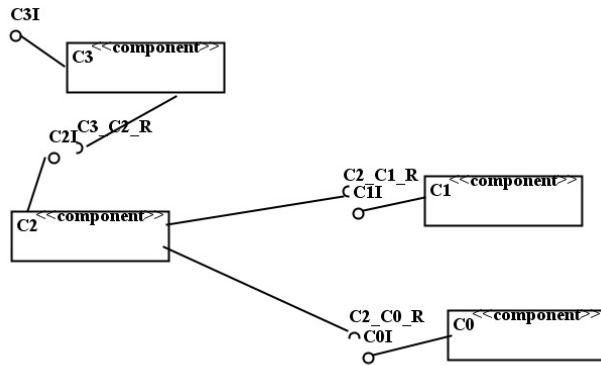


Fig. 12.4 Rearchitected example component diagram

rary operations. It is therefore direct to identify all such processing and to segregate the code by defining new *data access object* (DAO) components, which encapsulate the specific processing. The DAO operations are then called from the business logic components using dependency inversion. A *required interface* C_DAO_Req of each business logic component C is extracted, which expresses precisely which DAO operations are required by C .

The resulting architecture will normally satisfy the key principles of the clean architecture (Chapter 2):

- *No cyclic dependencies between components (ADP)*: due to heuristic rules (3) and (5).
- *Interface segregation principle “Don’t depend on things you don’t need” (ISP)*: a component C_i only depends on a component C_j if some operation of C_i requires an operation of C_j . It depends on the specific interface $C_i_C_j_R$ which contains only those operations that are actually used.
- *Dependency rule “Platform-independent components should not depend on platform-specific components” (DR)*: as described above, required interfaces R of business tier components are introduced, which the DAO components for technology-specific resource management then depend upon (their provided interfaces must satisfy/extend the R interfaces).

The *Single responsibility principle (SRP)* – that each component is only for one actor – is ensured if the original program was only used by one actor.

12.5 Code synthesis and test case generation

From the abstracted and restructured UML/OCL representation, code can be generated in multiple target languages. AgileUML provides code generators to synthesise code in ANSI C, C++, C#, Java, Go, Swift and Python. Code for Web apps, Web services, and Android and iOS mobile app platforms can be generated. The Go, Swift, Android and iOS code generators are written in *CSTL* and can be user-configured. The abstracted models could also be used for code synthesis using other MDE toolsets, such as Papyrus [72].

Testing is an important activity in re-engineering, in order to assure *semantic preservation* of the source application by the re-engineering process, i.e., that the system resulting from re-engineering still performs the same functions as the source application. To facilitate this, AgileUML automatically generates test cases and a mutation testing UI from the abstracted application specification, in order to provide prioritised testing of the synthesised code [41, 42]. The test cases are derived from the operation parameter types, and from class attribute types, so that they are ‘black box’ tests independent of the definition of operations. For each type a set of values are chosen which represent typical values and also potential boundary values. Thus for integers we use the values 0, 1, −1, 1024, −1025 and *MAXINT*. The test values can be directly configured by the end user. If an operation has preconditions, then these are also used to infer test values. For example, an operation

```
operation integerPlaces(x : int) : int
pre: x > 0
post:
    result = 1 + x->log10()->oclAsType(int)
```

would have test values 0, 1, 1024 and *MAXINT*. Similarly, test values for attributes are adjusted based on any invariants of their class. All combinations of test values for parameters and attributes are used, so that an operation with N integer parameters could have 6^N test cases. This test generation procedure can therefore result in very large test sets, and so test case selection and prioritisation based on mutation testing [43] can be used to reduce these sets to a subset of effective tests.

Mutation testing is based on creating mutated versions of operations, with a single mutation in the code or postcondition definition of an operation *op* producing a mutant version *op'*. For the above example, mutants include

```
operation integerPlaces_mut1(x : int) : int
pre: x > 0
post:
    result = 1 - x->log10()->oclAsType(int)
```

and

```

operation integerPlaces_mut2(x : int) : int
pre: x > 0
post:
  result = 1 + x->log()->oclAsType(int)

```

Tests for *op* are ranked based on what proportion of *op* mutants they detect, i.e., for which they give a different result compared to the original *op* version. For the above example, test $x = 1$ has 33% effectiveness in detecting mutants (because it does not distinguish *log* and *log10*), whilst $x = 1024$ has 100% effectiveness. Other test values have 0% effectiveness. The higher the rate of detection, in principle the more effective is the test at discovering flaws in implementations of *op*. Tests which have low scores can be eliminated from the test set. Thus we would only retain test values 1 and 1024 in the above case.

12.6 Re-engineering examples

In this section we illustrate VB6/VBA and Cobol85 re-engineering tasks with extracts from real-world applications.

12.6.1 VB6/VBA re-engineering

The option pricing example consists of 5 VBA functions, together with a spreadsheet. The VBA code can be divided into two main parts: (i) computation of put and call option values using the Black-Scholes analytic formulae [12]; (ii) a numeric estimation algorithm for call options based upon the construction of a large number of alternative price trajectories for the asset underlying the option, represented as a binomial tree.

Functions *D1* and *D2* from part (i) have been given in Section 12.2.1. The remaining functions of part (i) are as follows:

```

FUNCTION BS_Call(S AS Double, K AS Double, Vol AS Double, R AS Double,
  Q AS Double, T AS Double) AS Double

BS_Call =
  S * Exp(-Q * T) * WorksheetFunction.Norm_S_Dist(D1(
    S, K, Vol, R, Q, T), True) -
  K * Exp(-R * T) * WorksheetFunction.Norm_S_Dist(D2(
    S, K, Vol, R, Q, T), True)
END FUNCTION

FUNCTION BS_Put(S AS Double, K AS Double, Vol AS Double, R AS Double,
  Q AS Double, T AS Double) AS Double

```

```

BS_Put =
    K * Exp(-R * T) * WorksheetFunction.Norm_S_Dist(-D2(
        S, K, Vol, R, Q, T), True) -
    S * Exp(-Q * T) * WorksheetFunction.Norm_S_Dist(-D1(
        S, K, Vol, R, Q, T), True)
END FUNCTION

```

These functions implement the Black-Scholes equation solutions for put and call options, using the auxiliary functions *D1* and *D2* and the cumulative normal distribution function *Norm_S_Dist* from Excel.

It is noticeable that each of the part (i) functions have the same group *S, K, Vol, R, Q, T* of parameters. These are properly regarded as attributes of an entity type for the option being evaluated: the underlying asset price *S*, the option strike price *K*, asset volatility *Vol*, risk-free interest rate *R*, underlying asset dividend rate *Q*, and option term *T*. Therefore they should be grouped into an *OptionVO* value object in order to reduce the number of operation parameters being used.

The platform-dependence of part (i) upon Excel can be factored out as a required interface *C1_Excel_Req* which declares the *Ln* and *Norm_S_Dist* operations. AIL is too low (1.43) for *D1* and *D2*, indicating that more meaningful names should be given for the functions or for the variables in these functions.

Part (ii) consists of a single function which involves a complex numerical approximation computation:

```

FUNCTION EBinCall(S, K, Sigma, R, Q, T, N)

,
' Inputs are S = initial stock price
,           K = strike price
,           r = risk-free rate
,           sigma = volatility
,           q = dividend yield
,           T = time to maturity
,           N = number of time periods
,

DIM Dt, U, D, Pu, Pd, U2, Prob, CallV, I

Dt = T / N                                ' length of time period

U = Exp(Sigma * Sqr(Dt))                  ' size of up step

D = 1 / U                                ' size of down step

Pu = (Exp((R - Q) * Dt) - D) / (U - D)    ' probability of up step

Pd = 1 - Pu                               ' probability of down step

U2 = U * U

S = S * D ^ N                            ' stock price at bottom node at last date

```

```

Prob = Pd ^ N          ' probability of bottom node at last date

CallV = Prob * Application.Max(S - K, 0)
    ' probability weighted call value
FOR I = 1 TO N
    ' step up over nodes at last date
    S = S * U2          ' stock price

    Prob = Prob * (Pu / Pd) * (N - I + 1) / I    ' probability

    CallV = CallV + Prob * Application.Max(S - K, 0)
        ' sum weighted values
NEXT I

EBinCall = Exp(-R * T) * CallV

END FUNCTION

```

Unlike part (i), this function is provided with helpful comments, however all the parameters and local variables are untyped. The function uses the Excel *Max* function, and this dependency can be factored via a required interface *C2_Excel_Req* containing a declaration of *Max*. As with part (i), the *OptionVO* attributes are used as a parameter group, together with the *N* parameter controlling the accuracy of the approximation procedure.

Following abstraction to UML/OCL, translation of the application to Python 3.9 can be carried out, producing the code illustrated in Figure 12.5.

```

def EBinCall(self, S, K, Sigma, R, Q, T, N) :
    result = None
    EBinCall = None
    Dt = None
    U = None
    D = None
    Pu = None
    Pd = None
    U2 = None
    Prob = None
    CallV = None
    I = None
    Dt = (T/(1.0 * N))
    U = math.exp((Sigma * 0.01 * math.sqrt(Dt)))
    D = (1/(1.0 * U))
    Pu = ((math.exp(((R - Q) * Dt)) - D)/(1.0 * (U - D)))
    Pd = (1 - Pu)
    U2 = (U * U)
    S = (S * (math.pow(D, N)))
    Prob = (math.pow(Pd, N))
    CallV = (Prob * Excel.Max([(S - K),0]))
    I = 1
    while I <= N :
        S = (S * U2)
        Prob = (((Prob * (Pu/(1.0 * Pd))) * ((N - I) + 1))/(1.0
* I))
        CallV = (CallV + (Prob * Excel.Max([(S - K),0])))
        I = I + 1
    EBinCall = (math.exp((-R * T)) * CallV)
    return EBinCall

```

Fig. 12.5 Translation of option pricing app to Python

Semantic preservation was satisfied for the deterministic numeric computations of parts (i) and (ii), except that the calls of *Norm_S_Dist* produce slightly different values in the original Excel/VB compared to the translated versions of these functions in Python. The difference is less than 10^{-8} , so is unlikely to be significant in practice.

12.6.2 Cobol85 re-engineering

We examined 100 examples of COBOL code, including examples from the language manual [20] and textbook [73], together with the industrial case of [47].

GO TO statements are replaced by recursion during the abstraction process, and the ‘replace recursion by iteration’ refactoring then applied to remove this quality flaw.

An example of a real-world COBOL program is the water/sewage utilities consumption calculator from [47], which has the following structure:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CONSUMPTCALC.
DATA DIVISION.
FILE SECTION.
FD CFILE LABEL RECORDS OMITTED.
01 CREC.
02 C-DEL-TAG PIC X.
02 C-KEY.
03 C-TYPE PIC X.
03 C-TARRIF PIC 9.
03 C-CSZE PIC 99.
02 C-CONSUMPT PIC 9(7).
02 PRV-MREAD PIC 9(6).
... other fields ...
PROCEDURE DIVISION.
MAIN.
OPEN INPUT CFILE.
PERFORM CALC-CONSUMP.
CLOSE CFILE.
STOP RUN.
CALC-CONSUMP.
READ CFILE
AT END
CLOSE CFILE
STOP RUN.

IF C-TYPE = "A" AND C-TARRIF < 6
THEN
PERFORM AVERAGE-CONSUMPTION
ELSE
PERFORM METDATA-CONSM.
GO TO CALC-CONSUMP.
METDATA-CONSM.
```

```

    IF C-TYPE = "M"
    THEN
        PERFORM TARIFF5-CONSUMPTION
    ELSE
        PERFORM CALC-READING.
CALC-READING.
    IF C-TYPE = "C" AND C-TARRIF < 6
    THEN
        PERFORM AVERAGE-CONSUMPTION
    ELSE
        PERFORM P2100A.
P2100A.
    IF C-TYPE = "B"
    THEN
        PERFORM P2100B
    ELSE
        PERFORM P2105BOTH-READING.
P2100B.
    IF C-TYPE = "B" AND PRV-MREAD = ZEROS
    THEN
        PERFORM INBETWEEN-METER
    ELSE
        PERFORM AVERAGE-CONSUMPTION.
P2105BOTH-READING.
    IF C-TYPE = "D"
    THEN
        PERFORM METERED-CONSUMPTION
    ELSE
        PERFORM AVERAGE-CONSUMPTION.
AVERAGE-CONSUMPTION.
    DISPLAY "Average consumption".
TARIFF5-CONSUMPTION.
    DISPLAY "Tariff5 consumption".
INBETWEEN-METER.
    DISPLAY "Inbetween meter".
METERED-CONSUMPTION.
    DISPLAY "Metered consumption".

```

The actual code of the final four paragraphs has been replaced by display statements.

The abstraction process (step 1 of Figure 12.1) produces the following UML/OCIL specification from the above code. Note that *CFILE* is initialised via the environment division of the program, which we do not show here.

```

class CONSUMPTCALC_Class {
    invariant CREC = CFILE_Record;
    invariant C$DEL$TAG = CFILE_Record.subrange(1,1);
    invariant C$DEL$TAG = CREC.subrange(1,1);
    invariant C$KEY = CFILE_Record.subrange(2,5);
    invariant C$KEY = CREC.subrange(2,5);
    invariant C$TYPE = CFILE_Record.subrange(2,2);
    invariant C$TYPE = CREC.subrange(2,2);
    invariant C$TYPE = C$KEY.subrange(1,1);

    invariant C$TARRIF = CFILE_Record.subrange(3,3)->toInteger();

```

```

invariant C$TARRIF = CREC.subrange(3,3)->toInteger();

invariant C$TARRIF = C$KEY.subrange(2,2)->toInteger();

invariant C$CSIZE = CFILE_Record.subrange(4,5)->toInteger();

invariant C$CSIZE = CREC.subrange(4,5)->toInteger();

invariant C$CSIZE = C$KEY.subrange(3,4)->toInteger();

invariant C$CONSUMPT = CFILE_Record.subrange(6,12)->toInteger();

invariant C$CONSUMPT = CREC.subrange(6,12)->toInteger();

invariant CREC = "" + StringLib.leftAlignInto("" + C$DEL$TAG,1) +
    StringLib.leftAlignInto("" + C$KEY,4) +
    StringLib.leftAlignInto("" + C$CONSUMPT,7);

invariant C$KEY = "" + StringLib.leftAlignInto("" + C$TYPE,1) +
    StringLib.leftAlignInto("" + C$TARRIF,1) +
    StringLib.leftAlignInto("" + C$CSIZE,2);

attribute CFILE : Sequence(String) := Sequence{};
attribute CFILE_Position : int := 0;
attribute CFILE_Record : String := "";
attribute CREC : String := "";
attribute C$DEL$TAG : String := "";
attribute C$KEY : String := "";
attribute C$TYPE : String := "";
attribute C$TARRIF : int := 0;
attribute C$CSIZE : int := 0;
attribute C$CONSUMPT : int := 0;
....

operation MAIN_Call() : void
pre: true
post: true
activity: CFILE_Position := 0 ;
    self.CALC$CONSUMPT_Call() ;
    OclProcess.exit(0) ;

operation CALC$CONSUMPT_Call() : void
pre: true
post: true
activity:
    CFILE_Position := CFILE_Position + 1 ;
    if CFILE_Position > CFILE->size()
    then OclProcess.exit(0)
    else
        CFILE_Record := CFILE[CFILE_Position];
    if ( C$TYPE = "A" ) & ( C$TARRIF < 6 )
    then
        self.AVERAGE$CONSUMPTION_Call()

```

```

else
    self.METDATA$CONSM_Call();
self.CALC$CONSUMP() ;
OclProcess.exit(0);

operation METDATA$CONSM_Call() : void
pre: true
post: true
activity:
    if C$TYPE = "M"
    then self.TARIFF5$CONSUMPTION_Call()
    else
        self.CALC$READING_Call();

operation CALC$READING_Call() : void
pre: true
post: true
activity:
    if ( C$TYPE = "C" ) & ( C$TARRIF < 6 )
    then self.AVERAGE$CONSUMPTION_Call()
    else self.P2100A_Call();

operation P2100A_Call() : void
pre: true
post: true
activity:
    if C$TYPE = "B"
    then self.P2100B_Call()
    else
        self.P2105BOTH$READING_Call();

operation P2100B_Call() : void
pre: true
post: true
activity:
    if ( C$TYPE = "B" & PRV$MREAD = 0 )
    then
        self.INBETWEEN$METER_Call()
    else
        self.AVERAGE$CONSUMPTION_Call();

operation P2105BOTH$READING_Call() : void
pre: true
post: true
activity:
    if ( C$TYPE = "D" ) then
        self.METERED$CONSUMPTION_Call()
    else
        self.AVERAGE$CONSUMPTION_Call();

operation AVERAGE$CONSUMPTION_Call() : void
pre: true
post: true
activity: execute ( ( "" + "Average consumption" )->display() );

```

```

operation TARIFF5$CONSUMPTION_Call() : void
pre: true
post: true
activity: execute ( ( "" + "Tariff5 consumption" )->display() );

operation INBETWEEN$METER_Call() : void
pre: true
post: true
activity: execute ( ( "" + "Inbetween meter" )->display() );

operation METERED$CONSUMPTION_Call() : void
pre: true
post: true
activity: execute ( ( "" + "Metered consumption" )->display() );

operation CALC$CONSUMP() : void
pre: true
post: true
activity: self.CALC$CONSUMP_Call() ;
}

```

In step 2 of the process of Figure 12.1, quality analysis identifies several occurrences of ‘magic numbers’ (*MGN* code smell), i.e., the number 6 and flags “D”, “C”, etc, embedded in the program code. The call graph is also identified as being excessively complex, with a high CBR_1 measure.

By performing the refactoring “Replace call by definition” to replace the call of *CALC\$CONSUMP_Call* by its code in *CALC\$CONSUMP*, it can be seen that *CALC\$CONSUMP* is recursively-defined:

```

operation CALC$CONSUMP() : void
pre: true
post: true
activity:
  CFILE_Position := CFILE_Position + 1 ;
  if CFILE_Position > CFILE->size()
  then OclProcess.exit(0)
  else
    CFILE_Record := CFILE[CFILE_Position];
    if ( C$TYPE = "A" ) & ( C$TARRIF < 6 )
    then
      self.AVERAGE$CONSUMPTION_Call()
    else
      self.METDATA$CONSUMP_Call();
      self.CALC$CONSUMP() ;
      OclProcess.exit(0);

```

Applying the “Replace tail recursion by iteration” refactoring, this can then be restructured to:

```

operation CALC$CONSUMP() : void
  pre: true
  post: true
  activity:
    while true
    do (
      CFILE_Position := CFILE_Position + 1 ;
      if CFILE_Position > CFILE->size()
      then OclProcess.exit(0)
      else
        CFILE_Record := CFILE[CFILE_Position];
        if ( C$TYPE = "A" ) & ( C$TARRIF < 6 )
        then
          self.AVERAGE$CONSUMPTION_Call()
        else
          self.METDATA$CONSM_Call() ) ;
      OclProcess.exit(0);
    )

```

The system can be reorganised to conform to the Pipes-and-filters architecture style, whereby each of the operations *CALC\$CONSUMP*, *METDATA\$CONSUMP*, *CALC\$READING*, *P2100A*, *P2100B*, *P2105BOTH\$READING* is expressed as a filter component on the input record data, forwarding this data to either one of two output streams. The acyclic graph of filters ends in one of four alternative target business tier components corresponding to the operations *AVERAGE\$CONSUMPTION*, *TARIFF5\$CONSUMPTION*, *INBETWEEN\$METER* or *METERED\$CONSUMPTION*.

In step 3 of the process of Figure 12.1, a system expert provides analysis of the source code and abstracted system, to relate the system functionality to requirements. The requirements can be expressed as a use case model, where the use cases correspond to the four alternative business tier targets identified in step 2 (Figure 12.6). The precondition of each use case is the disjunction (over the filter graph paths which end in the use case) of the combined filter conditions on each path. For example, the precondition for *MeteredConsumption* is *C\$TYPE* = “D”.

Finally, in steps 4 and 5 of the process of Figure 12.1, target Java code can be produced for the system, together with a test suite.

Summary

This chapter has considered techniques for software re-engineering, including restructuring and re-architecting processes, and has showed how these techniques can be applied to practical cases of re-engineering tasks.

Fig. 12.6 Utilities program use case model



Exercises

1

Use the COBOL abstraction process to identify the result of the following program (example taken from [73]).

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. COBOLPERFORMSOURCE.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
77 CNT PIC 999 VALUE 0.  
PROCEDURE DIVISION.  
MAIN SECTION.  
P1.  
    MOVE ZERO TO CNT.  
    PERFORM P2 THRU P5.  
P2.  
    PERFORM P3.  
P3.  
    ADD 1 TO CNT.  
    IF CNT = 5  
        MOVE ZERO TO CNT.  
P4.  
    ADD 3 TO CNT.  
P5.  
    DISPLAY CNT.
```

2

Explain why rules 1 to 5 of Section 12.4.3 reduce or leave unchanged the CRA-Index of a module when they are applied.

3

VB also has expression forms $\text{expr1} \wedge \text{expr2}$ for power (exponentiation), $\text{expr1} / \text{expr2}$ for division, $\text{expr1} \setminus \text{expr2}$ for integer division, and $\text{expr1} \text{ MOD } \text{expr2}$ for modulus. Define appropriate *CSTL* abstraction rules for these operators.

4

Give the refactoring rule for eliminating recursive calls of the form

```
operation _1 ( )
  activity: _* ; if _2 then self._1 ( ) else _3 ;
```

5

Use the “Replace call by definition” and “Replace tail recursion by iteration” refactorings to rewrite *op1* in the following class into an iterative form:

```
class E {
  attribute x : int := 0;

  operation op1()
  pre: true post: true
  activity:
    x := x + 1;
    self.op2();

  operation op2()
  pre: true post: true
  activity:
    if x < 10 then self.op1() else skip;
}
```

Summary and Conclusions

In this book we have provided an introduction to the essential concepts of software architecture: components, connections, configurations, architectural styles, and architecture quality aspects. We have taken the ‘clean architecture’ principles as the core practices to follow when building systems that can evolve and be maintained for significant lifetimes without excessive effort. We have considered the relationship of software architecture to software requirements, particularly to quality-of-service requirements, and described how architectural design can be applied in agile and model-driven software development approaches.

We considered several application domains in detail: mobile apps, enterprise and web applications, service-oriented systems, safety-critical systems and machine learning systems. For each domain, relevant architectural issues and styles have been described, together with example case studies illustrating the architectural techniques for systems in the domain.

The field of software development undergoes rapid change, but the fundamental principles of good software design and well-organised software architectures have endured over many years and are likely to be relevant for many years to come, regardless of advances in software technologies.

References

1. Acceleo project, <https://www.eclipse.org/acceleo/>, accessed 1.5.2022.
2. M. Agarwal, K. Talamadupula, F. Martinez, S. Houde, M. Muller, J. Richards, S. Ross, J. Weisz, *Using document similarity methods to create parallel datasets for code translation*, asXiv:2110.05423v1, 2021.
3. H. Alfraihi, K. Lano, *The integration of agile development and MDE: a systematic literature review*, Modelsward 2017.
4. D. Alur, J. Crupi, D. Malks, *Core J2EE Patterns*, 2nd edition, Prentice Hall, 2003.
5. E. C. Arango, O. L. Loaiza, *SCRUM Framework Extended with Clean Architecture Practices for Software Maintainability*. In: Silhavy, R. (eds) *Software Engineering and Algorithms*. CSOC 2021. Lecture Notes in Networks and Systems, vol 230. Springer, Cham. https://doi.org/10.1007/978-3-030-77442-4_56
6. R-J. Back, *Invariant based programming*, FACS vol. 21 (3), 2008, pp. 227–244.
7. J. Barnes, *High-integrity Ada: The SPARK approach*, Addison-Wesley, 1997.
8. K. Beck et al., *Manifesto for Agile Software Development*, agilemanifesto.org, 2001.
9. W. Behutiye, P. Rodriquez, M. Oivo, A. Tosun, *Analysing the concept of technical debt in the context of agile software development*, *Information and Software Technology*, vol. 82, 2017, pp. 139–158.
10. R. Bellamy et al, *AI Fairness 360: An extensible toolkit for detecting, understanding and mitigating unwanted algorithmic bias*, arXiv preprint 1810.01943, 2018.
11. T. Besker, A. Martini, J. Bosch, *Managing architectural technical debt: a unified model and systematic literature review*, *JSS*, vol. 135, pp. 1–16.
12. F. Black, M. Scholes, *The Pricing of Options and Corporate Liabilities*, *Journal of Political Economy*, 8, 637–654.
13. J. Bogner, R. Verdecchia, I. Gerostathopoulos, *Characterising technical debt and anti-patterns in AI-based systems*, *Int. Conf. on Technical Debt.*, IEEE Press, 2021, pp. 64–73.
14. J. P. Bowen, P. Breuer, K. Lano, *Formal Specifications in Software maintenance: from code to Z++ and back again*, *Information and Software Technology*, 35 (11–12), pp. 679–690, 1993.
15. M. Bowman, L. Briand, Y. Labiche, *Solving the class-responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms*, *IEEE TSE*, 36(6), 2010, pp. 817–837.
16. F. Buttner, M. Gogolla, *On OCL-based imperative languages*, *Science of Computer Programming*, vol. 92, 2014, pp. 162–178.
17. G. Campbell, P. Papapetrou, *SonarQube in Action*, Manning Publications Co, 2013.
18. Catalysis, <http://www.catalysis.org>, 2023.
19. K. Cho et al., *Learning phrase representations using RNN encoder-decoder for statistical machine translation*, *EMNLP* 2014.

20. ClearPath Enterprise Servers, *COBOL ANSI-85 Programming Reference Manual*, April 2015.
21. K. Collins, *How one programmer broke the Internet by deleting a tiny piece of code*, Quartz magazine, March 2016.
22. M. Crawford, *Lurking within COBOL PERFORMs*, Journal of Software Maintenance, vol. 2(1), 1990, p. 33–60.
23. A. De Marco, V. Iancu, I. Asinofsky, *COBOL to Java and newspapers still get delivered*, in Proceedings IEEE International Conference on Software Maintenance and Evolution, IEEE Press, 2018, pp. 583–586.
24. J. Devlin et al., *BERT: pre-training of deep bidirectional transformers for language understanding*, ArXiv: 1810.04805, 2018.
25. Epsilon Generation Language, Epsilon project, <https://projects.eclipse.org/projects/modeling.epsilon>, accessed 1.5.2022.
26. M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, *Refactoring: improving the design of existing code*, Addison-Wesley, 1999.
27. M. Fowler, *UML Distilled*, 3rd edition, Addison Wesley, 2003.
28. M. Fowler, K. Beck, *Refactoring: improving the design of existing code*, 2nd Ed., Pearson, 2019.
29. R. Fuentes-Fernandez, J. Pavon, F. Garijo, *A model-driven process for the modernisation of component-based systems*, Science of Computer Programming, vol. 77, 2012, pp. 247–269.
30. A. Fuhr, T. Horn, V. Riediger, A. Winter, *Model-driven software migration into service-oriented architectures*, Comput. Sci. Res. Dev., vol. 28, 2013, pp. 35–84.
31. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1994.
32. S. Goldsack, K. Lano, E. Durr, *Invariants as Design Templates in Object-based Systems*, Workshop on Foundations of Component-based Systems, ESEC 97.
33. M. Grand, *Patterns in Java, Volume 1*, 2nd Edition, Wiley, 2002.
34. J. Greenfield, K. Short, S. Cook, S. Kent, *Software Factories*, Wiley, 2004.
35. S. Gulwani, *Programming by Example*, Microsoft Corp., 2016.
36. D. Guo et al., *GraphCodeBERT: Pre-training code representations with dataflow*, ICLR 2021, 2021.
37. X. He, P. Avgeriou, P. Liang, Z. Li, *Technical debt in MDE: A case study on GMF/EMF-based projects*, MODELS 2016.
38. R. Hoda, N. Salleh, J. Grundy, *The Rise and Evolution of Agile Software Development*, IEEE Software, July 2018.
39. Information Commissioners Office/Alan Turing Institute, *Explaining decisions made with AI information*, 2020.
40. J. Izquierdo, J. Molina, *Extracting models from source code in software modernisation*, SoSyM vol. 13, 2014, pp. 713–734.
41. K. Jin, K. Lano, *Design and classification of mutation operators for OCL specification*, OCL 2022, MODELS 2022.
42. K. Jin, K. Lano, *OCL-based test case prioritisation using AgileUML*, ModeVVa 2022, MODELS 2022.
43. R. Just, D. Jalali, L. Inozemtseva, M. Ernst, R. Holmes, G. Fraser, *Are mutants a valid substitute for real faults in software testing?*, proceedings 22nd ACM SIGSOFT ISFSE, 2014, pp. 654–665.
44. I. Krasteva, S. Stavru, S. Ilieva, *Agile software modernization to the service cloud*, ICIW 2013, pp. 1–9.
45. T. Kurtz, *BASIC*, ACM History of Programming Languages Conference, 1978, in SIGPLAN Notices, vol. 13, no. 8, pp. 103–118.
46. M-A. Lachaux, B. Roziere, L. Chanussot, G. Lample, *Unsupervised translation of programming languages*, arXiv:2006.03511v3, 2020.
47. K. Lano, N. Malik, *Mapping Procedural Patterns to Object-oriented Design Patterns*, Automated Software Engineering 6(3), 265–289 (1999).

48. K. Lano, *Constraint-driven development*, Information and Software Technology, Vol. 50, issue 5, pages 406–423, 2008.
49. K. Lano, *Agile model-based development using UML-RSDS*, CRC Press, 2017.
50. K. Lano, Q. Xue, S. Kolahdouz-Rahimi, *Agile specification of code generators for model-driven engineering*, ICSEA 2020.
51. K. Lano, S. Kolahdouz-Rahimi and L. Alwakeel, *Synthesis of mobile applications using AgileUML*, ISEC 2021, pp. 1–10, 2021.
52. K. Lano, *Adding regular expression operators to OCL*, OCL 2021, MODELS 2021.
53. K. Lano, *Program translation using model-driven engineering*, short paper, ICSE 2022.
54. K. Lano, S. Kolahdouz-Rahimi, K. Jin, *OCL libraries for software specification and representation*, OCL 2022, MODELS 2022.
55. T. H. Le, H. Chen, M. Babar, *Deep learning for source code modeling and generation*, arXiv:2002.05442v1, 2020.
56. N. Leveson, *Safeware: system safety and computers*, Addison-Wesley, 1995. ISBN 0-201-11972-2.
57. B. Liskov, J. Wing, *A behavioral notion of subtyping*, ACM Trans. Program. Lang. Syst. 16(6), 1994, pp. 1811–1841.
58. S. Liu, H. Li, Z. Jiang, X. Li, F. Liu, Y. Zhong, *Rigorous code review by reverse engineering*, Information and Software Technology, Vol. 133, 2021.
59. X. Liu, H. Yang, H. Zedan, *Formal methods for the re-engineering of computing systems*, Compsac ‘97, 1997.
60. M-T. Luong et al., *Effective approaches to attention-based neural machine translation*, EMNLP 2015.
61. E. da S. Maldonado, E. Shihab, N. Tsantalis, *Using natural language processing to automatically detect self-admitted technical debt*, IEEE TSE, 2017. Doi: 10.1109/TSE.2017.2654244.
62. R. Marinescu, *Assessing technical debt by identifying design flaws in software systems*, IBM Journal of Research and Development, 56(5), 2012.
63. R. Martin, *Clean Architecture*, Prentice Hall, 2018.
64. Microsoft Com, *Office VBA Reference*, <https://learn.microsoft.com/en-us/office/vba/api/overview>, Oct. 2022.
65. S. Muggleton, L. de Raedt, *Inductive logic programming: theory and methods*, Journal of Logic Programming, vols. 19–20, 1994, pp. 629–679.
66. E. Nabavi, K. Daniell, E. Williams, C. Bentley, *AI for Sustainability: A changing landcape*, in Artificial Intelligence for better or worse, Future Leaders, 2019.
67. C. Nelson, A. Siegel, *Parsimonious modelling of yield curves*, Journal of Business, vol. 60 (4), pp. 473–489, 1987.
68. A. Nguyen, T. Nguyen, T. Nguyen, *Lexical statistical machine translation for language migration*, 9th Joint meeting on Foundations of Software Engineering, 2013, pp. 651–654.
69. R. Nunkesser, *Choosing a global architecture for mobile applications*, TechRxiv, <https://doi.org/10.36227/techrxiv.14212571.v1>, 2021.
70. E. Ogheneovo, *On the relationship between software complexity and maintenance costs*, Journal of Computer and Communications, vol. 2, 2014, pp. 1–16.
71. OMG, *Object Constraint Language 2.4 Specification*, OMG document formal/2014-02-03, 2014.
72. Papyrus toolset, <https://www.eclipse.org/papyrus>, 2020.
73. A. Parkin, *COBOL for Students*, Edward Arnold publishers, 1982.
74. R. Perez-Castillo, I. Garcia-Rodriguez de Guzman, M. Piattini, *Knowledge discovery metamodel ISO/IEC 19506: A standard to modernize legacy systems*, Computer Standards and Interfaces, vol. 33, 2011, pp. 519–532.
75. F. Gomes Rocha, S. Misra, M. Soares, *Guidelines for Future Agile Methodologies and Architecture Reconciliation for Software-Intensive Systems*, Electronics 2023, 12, 1582, 2023.

76. F. Salazar, M. Brambilla, *Tailoring architecture concepts and process for mobile application development*, 3rd International Workshop on Software Development Lifecycle for Mobile, 2015, pp. 21–24.
77. J. Sammet, *The early history of COBOL*, ACM History of Programming Languages Conference, 1978, in SIGPLAN Notices, vol. 13, no. 8, pp. 121–161.
78. D. Sanchez, A. Rojas, H. Florez, *Towards a clean architecture for Android apps using model transformations*, IJCS, vol. 49, no. 1, 2022.
79. J. Sanders, <https://www.techrepublic.com/article/jpmorgans-athena-has-35-million-lines-of-python-code-and-wont-be-updated-to-python-3-in-time>, 2019.
80. K. Schwaber, M. Beedle, *Agile software development with Scrum*, Prentice Hall, 2002.
81. D. Sculley et al., *Hidden technical debt in machine learning systems*, NIPS ‘15, 2015.
82. Securities and Equities Commission, *SEC charges Knight Capital with violations of market access rule*, 2013.
83. M. Shaw, D. Garlan, *Software Architecture: Perspectives on an emerging discipline*, Prentice Hall, 1996.
84. H. Sneed, G. Jandrasics, *Inverse transformation of software from code to specification*, IEEE Conf. Soft. Maintenance, 1987.
85. H. Sneed, *Migrating from COBOL to Java: A report from the field*, IEEE Proc. of 26th ICSM, IEEE Press, 2011, pp. 1–7.
86. S. Stavru, I. Krasteva, S. Ilieva, *Challenges of Model-driven Modernization: an Agile perspective*, Modelsward 2013.
87. Stripe, *The developer coefficient: software engineering efficiency and its \$3 trillion impact on global GDP*, Technical report, 2018.
88. C. Szyperski, *Component Software – beyond object-oriented programming*, Addison-Wesley, 2002.
89. A. Tornhill, M. Borg, *Code Red: The business impact of code quality*, arXiv:2203.04374v1, 2022.
90. A. Vaswani et al., *Attention is all you need*, NIPS 2017.
91. S. Wehaibi, E. Shihab, L. Guerrouj, *Examining the impact of self-admitted technical debt on software quality*, 23rd IEEE International Conference on Software Analysis, Evolution and Re-engineering, 2016.
92. M. Zavershynskyi, *Technical debt in Machine Learning*, towardsdatascience.com/technical-debt-in-machine-learning, 2017.
93. W. Zhao et al., *A survey of large language models*, arXiv:2303.18223v10, 2023.

Appendix A

User guide for architecture diagram tool

Abstract This tutorial covers the use of the AgileUML diagram editor for UML component diagrams.

1. Download the AgileUML tools from github.com/eclipse/agileuml:

```
umlrsds.jar
```

2. Create a directory to contain *umlrsds.jar*, and create a subdirectory *output*.
3. Run the tool from the command line:

```
java -jar umlrsds.jar
```

4. This opens a class diagram editor, where you can create (*Create* menu) new classes and interfaces, and architecture diagrams (*Create* $\Rightarrow$ *Architecture*).
5. The architecture diagram editor supports the creation of components, with provided and required interfaces, and to connect these with assembly connectors.

Outline Java code can also be generated.

An applet version of the tools is available from <https://agilemde.co.uk/arch>.

1

Set up the tools as described above, and create a component *Client* and component *Supplier* in the architecture editor, together with a provided interface *P* of *Supplier*, and required interface *R* of *Client*.

Your diagram should look like Figure A.1.

2

Drag the interface *R* (or *P*) using *Edit* $\Rightarrow$ *Move* so that the two interfaces are close together, and the tool recognises that an assembly connection is

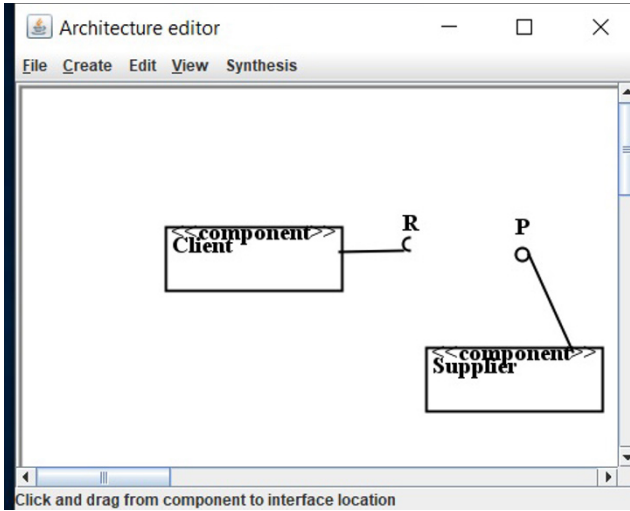


Fig. A.1 Architecture example

intended between them. This will be valid (trivially, because both interfaces are empty).

3

Go back to the class diagram and fill in more details of the interfaces, by adding an operation $op1(x : int)$ to R .

4

Check the validity of your architecture by selecting $View \Rightarrow Components$ on the architecture diagram editor. You should get the message

```
!! Warning: operation op1(int) not found in P
!! This is not a valid assembly connection!
```

5

This problem can be resolved by adding the operation $op1$ of R to P .

Appendix B

User guide for AgileUML

In addition to the architecture editor, AgileUML provides a wide range of functionalities for system specification and implementation:

- Class diagram construction, either visually via the main UI, or in text form (the “Edit KM3” option).
- Code synthesis (“Build” menu) in multiple target languages and for multiple platforms, including Java, C#, C++, Python, Swift and Go, Android and iOS mobile apps (following the VIPER and MVVM patterns described in Chapter 6), Servlet and JSP-based web applications, and REST and SOAP web service applications.

We illustrated the mobile app synthesis options with the m-health app, app4, in Chapter 6, and the synthesis of JSP-based web applications for this app in Chapter 8. The share price estimator in Chapter 9 was produced using the REST web service generator.

For Python code generation, the directory *uml2py* is needed (*uml2py.zip* on github.com/eclipse/agileuml). For C code generation, *uml2Ca* and *uml2Cb* are needed.

For Swift, Go, Android and iOS code generation the CSTL scripts directory *cg* is needed.

- Reverse-engineering from Java, COBOL, VB, Python, C and JavaScript, as described in Chapter 12. (“File” menu).

The *cg* and *libraries* directories are needed for reverse-engineering.

- Quality analysis (“Analysis” menu) for checking quality flaws/technical debt and for conformance to clean architecture principles, as described in Chapter 3.
- Requirements formalisation options to construct UML models from natural language requirements (“File” menu).

This needs supporting files *background.txt* and *verbs.txt* in the *output* directory.

A set of library classes are defined in the *libraries* subdirectory of the Github repository:

- Excel – Excel Workbook functions.
- MathLib – additional mathematical functions such as `asinh`, `factorial`, `combinatorial`, etc.
- OclDate – dates.
- OclDatasource – databases and HTTP connections.
- OclFile – for text file processing.
- OclProcess – threads and OS processes.
- OclRandom – random number generation.
- OclType – for reflection capabilities.
- StringLib – String manipulation such as *swapCase*, *format*, etc.

Each library component can be imported into a development via the Extensions menu option “import library”. Implementations in multiple target languages are provided. The libraries provide a platform for the definition of systems using AgileUML, with a wide range of built-in services that can be used by new applications.

Index

- «*Webservice*», 181
- «*asynchronous*», 77
- «*interface*», 21
- «*publish/subscribe*», 78
- «*query*», 75
- «*synchronous*», 76
- CSTL*, 230
- .NET, 138, 139

- Abstract component, 16
- ACID properties, 141
- Acyclic dependencies principle (ADP), 45
- Ada, 19, 201
- Adapter style, 81
- ADL, 12
- ADP, 239
- Agile MDE, 56
- Agile methods, 14, 52
- AgileUML, 19, 56
- AJAX, 156
- Android adapters, 117
- Android assets, 117
- Android fragments, 118
- Android Intents, 118
- Android platform, 117
- Android receivers, 118
- Android services, 118
- Android Studio, 103
- ANTLR, 226, 230
- Application server, 99, 137
- Architectural refactoring, 54
- Architectural style, 81
- Architecture description language (ADL), 15
- Architecture-driven modernisation (ADM), 225
- Assembly class, 74
- Assembly connector, 17, 23
- Asset, 100
- Association, 59
- Association multiplicities, 59
- Asynchronous communication, 76
- Attribute-driven design, 30
- Automated code generation, 68
- Availability, 30, 172

- BASIC, 227
- BERT, 215
- Bias, 222
- Blackboard style, 89
- Body-mass index (BMI), 102
- Business concept model, 58
- Business Delegate pattern, 124
- Business entity component, 36
- Business interface, 57
- Business operation component, 36, 57
- Business sustainability, 28
- Business tier, 87, 151
- Business-to-business (B2B), 181
- Business-to-business service (B2B), 57

- C, 19, 201
- C++, 19
- C#, 19
- Callback, 77
- Callback operation, 114
- Cascading style sheets (CSS), 158
- Category type, 62
- Chain of Responsibility pattern, 166
- Circuit breaker pattern, 189
- Class notation, 22
- Clean architecture, 30, 35
- Client, 23
- Client tier, 87, 151

- Client/server style, 85
- COBOL, 229
- COBOL paragraph, 229
- COBOL procedure division, 229
- Code generation, 240
- Code smell, 46, 235
- Cohesion, 24, 237
- COM+ component, 139
- Common closure principle (CCP), 38
- Common reuse principle (CRP), 43
- Component, 15
- Component boundary, 46
- Component dependency, 35
- Component interface, 21
- Composite component, 73
- Composite structure diagram, 73
- Composite View pattern, 166
- Concrete component, 16
- Configuration, 15, 18
- Connector, 15, 17
- Constraint-based design, 31
- Containerisation, 179
- Context menu, 101
- Continuous deployment, 178
- Control invariant, 203
- Controller component, 37, 94
- Core type, 62
- Correction cascade, 220
- Coupling, 237
- CRA-Index, 237
- Cross-compiled app, 132
- Cross-platform app, 131
- CRUD operation, 21
- Cyclomatic complexity (CC), 47

- Daily scrum, 52
- DAO, 239
- Data access object (DAO), 82, 108
- Data normalisation, 216
- Data Transfer Object pattern, 113
- Dead experimental codepath, 220
- Decision tree, 213
- Deep learning, 215
- Delegation connector, 74
- Dependency inversion, 35
- Dependency inversion principle (DIP), 43
- Dependency rule, 35, 45, 239
- Dependent type, 62
- Diverse software, 195
- Django, 162
- Duplicated code (DC), 47
- Dynamic redundancy, 197

- Efficiency, 30, 172

- Elasticity, 177
- Embedded system, 202
- Embedded system architecture, 89
- Encapsulation, 24
- Enterprise Information System (EIS), 87, 137
- Enterprise Java Bean (EJB), 155
- Enterprise System architectures, 87
- Entity bean, 155
- Event class, 146
- Evolvability, 30, 172
- Excel worksheet, 228
- Excessive fan-in (EFI), 47
- Excessive fan-out (EFO), 47
- Excessive number of operations (ENO), 47
- Excessive parameter list (EPL), 47
- Explainability, 222
- Explicit state transfer, 183
- Extensibility, 29
- Extract-transform-load (ETL), 169

- Fairness measure, 223
- Fat client, 158
- Fault avoidance, 193
- Fault tolerance, 30, 172, 193
- Fault tolerance coverage, 195
- Feature vector, 212
- Feed-forward network, 214
- Feedback loop, 220
- Fine-tuning, 217
- Firebase, 133
- Firebase cloud datastore, 93
- Firmware, 202
- Flight control system (FCS), 192
- Flutter, 132
- Forward/reverse navigation, 101
- Freemium monetisation, 133
- Front controller, 187
- Front Controller pattern, 165

- Garbage feature, 220
- Gateway component, 37
- github, 39
- Glue code, 220
- Go, 19
- GPT, 215

- Hamburger menu, 101
- Hardware redundancy, 195
- Hardware transparency, 177
- Hazard, 193
- Heterogeneous style, 96
- Hyperparameter, 214

- IBAction annotation, 126
- IBOutlet annotation, 126
- Implements relation, 20
- Infinite recursion, 45
- Infrastructure-as-a-Service (IaaS), 178
- Integrated development environment (IDE), 91
- Integration tier, 87, 100, 151
- Intents, 101
- Interactor component, 37
- Intercepting Filter pattern, 166
- Interface, 19
- Interface extension, 24
- Interface segregation principle (ISP), 43
- Interlock, 200
- Interpreted app, 131
- iOS, 125
- iOS UIKit, 125
- ISP, 239

- Jakarta, 151
- Java, 19
- Java EE, 151, 152
- Java Server Page (JSP), 153
- JavaScript, 132
- JSF, 162
- Just-in-time (JIT) service, 145

- K-nearest neighbour (KNN), 212
- Kotlin, 101

- Language subset, 201
- Language transparency, 139
- Large language model (LLM), 215
- Lateral navigation, 101
- Layered architecture, 202
- Layered style, 88
- Legacy software, 225
- Lifeline, 75
- Lifetime transparency, 139
- Liskov substitution principle (LSP), 39
- List adapter, 122
- Location transparency, 139
- Long Short-Term Memory (LSTM), 214
- Loosely-coupled event service, 146

- M-health, 102
- Machine learning (ML), 211
- Magic numbers (MGN), 47
- Maintainability, 28, 30
- Managed component, 138
- Mandatory association, 62
- MBaaS, 133
- Message-driven bean, 155

- Microservice, 187
- Middleware, 138
- ML carbon footprint, 211
- ML model architecture, 211
- Mobile app, 99
- Mobile app architectures, 87
- Model component, 94
- Model facade, 108
- Model facade/Session facade pattern, 114
- Model View Presenter (MVP), 106
- Model-based testing (MBT), 51, 240
- Model-driven app development, 132
- Model-driven engineering (MDE), 14, 51
- Model-driven modernisation (MDM), 225
- Model-view-controller (MVC) style, 94
- Model-View-Template pattern, 162
- Model-View-ViewModel (MVVM), 106
- ModelFacade, 124
- Module, 15
- Multilayer Perceptron (MLP), 213
- Mutation testing, 241
- MVVM architecture, 69
- MVVM pattern, 128

- N-modular redundancy (NMR), 197
- N-tier style, 85
- N-version programming, 198
- Navigation controller, 108
- Neural net, 212

- Object pooling service, 143
- Objective-C, 125
- ObservedObject annotation, 128
- Observer design pattern, 90
- OCL 2.4, 230
- OclAny, 40, 232
- onchange event, 159
- onCreate, 122
- Online/offline DAO pattern, 116
- Open-closed principle (OCP), 39
- Operation visibility, 22
- Overfitting, 213

- Pagelet, 154, 163
- Performance, 30
- Pipeline jungle, 220
- Pipes-and-Filters, 82, 216
- Platform-as-a-Service (PaaS), 178
- Port, 74
- Presentation tier, 87, 151
- Presenter component, 37
- Product backlog, 53
- Product owner, 53

- Programmable Logic Controller (PLC), 195
- Protection system, 199
- Provided interface, 16
- Publish/subscribe connection, 78, 91
- Python, 19
- Python quality issues, 221
- QoS property, 137
- Quality-of-service (QoS), 29
- Quantlib, 25
- Queued components service, 147
- Random forest, 213
- React Native, 131
- React Redux architecture, 132
- Reactive control invariant, 204
- Reactive control system, 203
- Rearchitecting, 237
- Recovery blocks, 198
- Recurrent Neural Network (RNN), 214
- Refactoring, 236
- Refinement relation, 41
- Remote procedure call (RPC), 75, 85
- Replace call by definition, 236
- Replace tail recursion by iteration, 236
- Representational State Transfer (REST), 183
- Required interface, 16
- Requirements management, 13
- Resource tier, 87, 151
- Responsiveness, 30, 172
- Reusability, 24, 25
- Risk in SCS, 191
- Rollback, 142
- Safe state, 199
- Safety, 193
- Safety integrity level (SIL), 193
- Safety-critical system (SCS), 191
- Scalability, 30, 172
- Scenario, 60
- Scrum, 52
- Scrum board, 53
- Scrum master, 53
- Security, 30, 172
- Segue, 101, 125
- Self-declared technical debt, 236
- Semantic dependence, 36
- Semantic preservation, 240
- Server, 23
- Service activator, 108
- Service activator pattern, 114
- Service Broker pattern, 186
- Service choreography, 187
- Service decomposition, 186
- Service Locator pattern, 169
- Service orchestration, 187
- Service registries, 180
- Service-oriented architecture (SOA), 177
- Servlet, 153
- Session bean, 155, 168
- Session Facade pattern, 168
- Shutdown system, 199
- Simple object access protocol (SOAP), 181
- Simplified Blackboard style, 94
- Single responsibility principle (SRP), 37
- Single-point failure, 196
- Singleton pattern, 125
- SOAP web service, 182
- Software architect, 13
- Software architecture, 12
- Software architecture model, 11
- Software design, 11
- Software fault tolerance, 198
- Software product line, 26
- Software redundancy, 195
- Software-as-a-Service (SaaS), 178
- SOLID principles, 37
- SPARK Ada, 201
- Sprint backlog, 53
- Sprint planning, 52
- Sprint retrospective, 53
- Sprint review, 52
- SQLite, 109, 117
- SRP, 239
- Standby spare architecture, 197
- State annotation, 128
- Statefull session bean, 155
- Stateless session bean, 155
- Static redundancy, 195
- Stereotype, 139
- Stub-skeleton pattern, 138
- Substitutability, 26
- Supervised learning, 212
- Supplier, 23
- Sustainability, 27
- Swift, 19, 101
- Swift class extension, 39
- Swift closure, 114
- SwiftUI, 127
- SwiftUI View, 127
- Symbolic ML, 216
- Synchronous communication, 75
- System interface, 57
- Systematic fault, 194
- Tagged value, 139

- Technical debt, 46, 235
- Thin client, 158
- Three tier architecture, 56
- Transaction service, 140
- Transactionality, 30, 141, 172
- Transformer architecture, 215
- Triple modular redundancy (TMR), 195
- Trustworthiness, 30, 172, 222
- Two-tier style, 85
- Type annotation, 222

- UML, 51
- UML component diagram, 19
- UML interaction model, 140
- UML operation, 21
- UML sequence diagram, 76
- Unsupervised learning, 212
- Unused operations/functions (UOP), 235
- Unused/unreachable code (UC), 236
- Usability, 30, 156
- Use case activity, 103
- Use case dependency, 64
- Use case extension, 103
- Use case model, 60
- Use case parameter, 103

- Use case precondition, 103
- User experience (UX), 101
- User story, 53

- Validation bean, 164
- Value object, 46, 68, 88
- Value Object pattern, 113
- VB6/VBA, 227
- View component, 94
- View controller, 108
- View Helper pattern, 163
- VIPER, 106, 107
- Virtualisation servers, 179
- void, 22

- Web app, 131
- Web application, 155
- Web service, 180
- Web service host, 181
- Web services definition language (WSDL), 182

- Xcode, 103, 125
- XSLT, 167