Software Architect's Handbook

Become a successful software architect by implementing effective architecture concepts

Packt

www.packt.com

By Joseph Ingeno

Software Architect's Handbook

Become a successful software architect by implementing effective architecture concepts

Joseph Ingeno

Packt

BIRMINGHAM - MUMBAI

Software Architect's Handbook

Commissioning Editor: Merint Mathew
Acquisition Editor: Denim Pinto
Content Development Editor: Priyanka Sawant
Technical Editor: Ketan Kamble
Copy Editor: Safis Editing
Project Coordinator: Vaidehi Sawant
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Graphics: Jason Monteiro
Production Coordinator: Aparna Bhagat

First published: August 2018

Production reference: 1290818

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78862-406-0

`www.packtpub.com`

To my children, Adriana and Alexander,
who make the world a better place.

Contributors

About the author

Joseph Ingeno is a software architect who oversees a number of enterprise software applications. During his career, he has designed and developed software for a variety of different industries. He has extensive experience working on web, mobile, and desktop applications using different technologies and frameworks.

Joseph graduated from the University of Miami a with Master of Science and a Bachelor of Business Administration degrees in Computer Information Systems, and followed that with a Master of Software Engineering degree from Brandeis University.

He holds several certifications, including the Microsoft Certified Solutions Developer and the Professional Software Engineering Master Certification from the IEEE Computer Society.

A special thanks to my wife, Sally, and the rest of my family for their understanding and patience while I devoted the time necessary to write this book.

I would like to thank Priyanka Sawant, Ketan Kamble, Ruvika Rao, Gaurav Aroraa, Anand Pillai, Denim Pinto, and everyone at Packt Publishing who provided input and support during the writing of this book.

About the reviewers

Gaurav Aroraa has done an MPhil in computer science. He is a Microsoft MVP, a lifetime member of **Computer Society of India** (**CSI**), an advisory member of IndiaMentor, certified as a Scrum trainer/coach, XEN for ITIL-F, and APMG for PRINCE-F and PRINCE-P. He is an open source developer, a contributor to TechNet Wiki, and the founder of Ovatic Systems Private Limited. In his career of over 20 years, he has mentored thousands of students and industry professionals. Apart from that, he's written over 100 white papers for research scholars and various universities across the globe.

> *I'd like to thank my wife, Shuby Arora, and my angel daughter, Aarchi Arora, as well as the team at PACKT.*

Anand B Pillai is a technophile by profession with 20 years' of experience in software development, design, and architecture. Over the years, he has worked with numerous companies in fields ranging from security, search engines, large-scale web portals and big data. He is a founder of the Bangalore Python Users' Group and is the author of *Software Architecture with Python* (PacktPub, April 2017). Anand is currently a VP of an engineering at the early-stage legal technology startup, Klarity Law. He happily resides with his family in Bangalore, India.

Table of Contents

Preface 1

Chapter 1: The Meaning of Software Architecture 8
- **What is software architecture?** 8
  - ISO/IEC/IEEE 42010 standard definition 9
  - What makes up a software architecture? 10
  - Software architecture is an abstraction 10
  - Software architecture is about the important stuff 10
- **Why is software architecture important?** 11
  - Defining a solution to meet requirements 11
  - Enabling and inhibiting quality attributes 12
  - Giving you the ability to predict software system qualities 12
  - Easing communication among stakeholders 13
  - Managing change 13
  - Providing a reusable model 14
  - Imposing implementation constraints 14
  - Improving cost and effort estimates 14
  - Serves as training for team members 15
  - Software architecture is not a silver bullet 16
- **Who are the consumers of software architectures?** 16
- **What is the software architect role?** 17
  - Software architects are technical leaders 17
  - Software architects perform a number of duties 18
  - Ivory tower software architects 18
  - What are software architects expected to know? 19
  - Don't be overwhelmed 20
  - Is the software architect role right for you? 21
- **Summary** 22

Chapter 2: Software Architecture in an Organization 23
- **Types of software architects** 24
  - Enterprise architect 24
  - Solution architect 25
  - Application architect 25
  - Data architect/information architect 26
  - Infrastructure architect 27
  - Information security architect 28
  - Cloud architect 28
- **Software development methodologies** 29
  - The Waterfall model 30

Phases of the Waterfall methodology 31
Issues with the Waterfall methodology 32
Agile software development methodologies 32
Agile values and principles 33
An iterative methodology 34
Adaptive rather than predictive 34
Daily stand-up meetings 35
Project management 36
The importance of software project estimation 36
Putting effort into the estimates 36
Being a realist (or even a pessimist) 37
Team and situational factors to consider 37
Project schedule changes 37
Getting a project back on schedule 38
Working overtime 38
Reducing scope 38
Adding resources 39
Reallocating resources 39
Identifying problem areas 40
Acting as early as possible 40
Office politics 40
Understanding your organization's goals 42
Addressing the concerns of others 42
Assisting people with their goals 42
Knowing when to compromise 43
Being aware of cultural differences 43
Software risk management 43
Risk avoidance 45
Transferring the risk to another party 45
Risk mitigation 46
Risk acceptance 46
Configuration management 46
Changing management 47
Software product lines 49
Benefits of a software product line 50
Core assets of an organization 50
Risks of product line engineering 51
Summary 51
Chapter 3: Understanding the Domain 53
Developing business acumen 53
Familiarity with general business topics 54
Understanding your organization's business 55
Domain-driven design 56
Encourages and improves communication 56
What is a ubiquitous language? 57

Entities, value objects, and aggregates 59
Entities 59
Value objects 59
Aggregates and root entities 60
Separating the domain into subdomains 60
What are bounded contexts? 61
Requirements engineering 63
Types of software requirements 63
Business requirements 64
Functional requirements 64
Non-functional requirements 65
Constraints 65
The importance of requirements engineering 66
Software requirements must be measurable and testable 67
Software requirements that affect architecture 67
Requirements elicitation 68
Techniques to elicit requirements 69
Interviews 69
Requirements workshops 69
Brainstorming 70
Observation 71
Focus groups 72
Surveys 72
Document analysis 73
Prototyping 73
Reverse engineering 74
Get access to the proper stakeholders 74
Summary 75
Chapter 4: Software Quality Attributes 76
Quality attributes 76
External or internal 77
Quality attributes and the SDLC 78
Testing quality attributes 78
Maintainability 79
Types of software maintenance 80
Corrective maintenance 80
Perfective maintenance 80
Adaptive maintenance 81
Preventive maintenance 81
Modifiability 81
Extensibility and flexibility 82
Scope of modifications 82
Designing for maintainability 82
Reducing size 83
Increasing cohesion 83
Reducing coupling 83
Measuring maintainability 83

Lines of code (LOC) 84
Cyclomatic complexity 84
Depth of inheritance tree (DIT) 86
Usability 87
Allowing users to complete their tasks efficiently 87
Learnability 87
Providing useful feedback 88
Accessibility 88
Usability needs to be considered during requirements 89
Usability testing 89
Appealing visual design 89
Providing a good help system 91
Software must be useful, and not just usable 91
Availability 91
Calculating availability based on time 92
Calculating availability based on request success rate 93
Faults, errors, and failures 93
Detecting faults 94
Ping/echo reply 94
Heartbeat 94
Timestamp 94
Voting 94
Sanity test/sanity checking 95
Condition monitoring 95
Self-tests 95
Recovering from faults 95
Exception handling 95
Retry strategy 96
Varying levels of redundancy 96
Rollback 97
Graceful degradation 97
Ignoring faulty behavior 97
Preventing faults 98
Removal from service 98
Transactions 98
Increasing competence sets 98
Exception prevention 98
Portability 99
Adaptability 99
Installability 99
Replaceability 100
Internationalization and localization 100
Maintaining portability 102
Interoperability 102
Challenges with interoperability 102
Locating and exchanging information with another system 103
Interoperability standards 104

Interoperability testing 104
Testability 105
Controllability 105
Observability 106
Isolability 106
Automatability 106
Complexity of the software 107
Importance of test documentation 108
What makes a good tester? 109
Summary 110
Chapter 5: Designing Software Architectures 111
Software architecture design 112
Making design decisions 112
Software architecture design terms 113
Structure 113
Element 114
System 114
Subsystem 114
Module 114
Component 115
The importance of software architecture design 115
Making key decisions 115
Avoiding design decisions can incur technical debt 116
Communicating the architecture to others 116
Providing guidance to developers 117
Influencing non-technical parts of the project 117
Top-down versus bottom-up design approaches 117
Top-down approach 117
Advantages of the top-down approach 118
Disadvantages of the top-down approach 118
Bottom-up approach 119
Advantages of the bottom-up approach 120
Disadvantages of the bottom-up approach 120
Which approach should I use? 121
Greenfield versus brownfield software systems 123
Greenfield systems 123
Brownfield systems 124
Architectural drivers 124
Design objectives 125
Primary functional requirements 126
Quality attribute scenarios 126
Prioritizing quality attribute scenarios 127
Constraints 127
Architectural concerns 128
Leveraging design principles and existing solutions 129

Selecting a design concept 129
Software architecture patterns 130
Reference architectures 130
Benefits of reference architectures 130
Refactoring a reference architecture for your needs 131
Creating your own reference architecture 131
Tactics 132
Externally developed software 132
Buy or build? 133
Advantages/disadvantages of building 133
Advantages/disadvantages of buying 134
Researching external software 134
Should I use open source software (OSS)? 135
Advantages of using open source software 136
Disadvantages of using open source software 136
Documenting the software architecture design 137
Sketching the architecture design 137
Documenting the design rationale 138
Design rationale for design evaluation 139
Design rationale for design verification 139
Design rationale for design knowledge transfer 139
Design rationale for design communication 140
Design rationale for design maintenance 140
Design rationale for design documentation 140
Design rationale for design reuse 140
Using a systematic approach to software architecture design 141
A general model of software architecture design 142
Architectural analysis 142
Architectural synthesis 143
Architectural evaluation 143
Architecture design is an iterative process 143
Selecting an architecture design process 144
Attribute-driven design (ADD) 145
Step 1 – Reviewing inputs 147
Step 2 – Establishing the iteration goal and selecting inputs to be considered in the iteration 147
Step 3 – Choosing one or more elements of the system to refine 148
Step 4 – Choosing one or more design concepts that satisfy the inputs considered in the iteration 148
Step 5 – Instantiating architectural elements, allocating responsibilities, and defining interfaces 148
Step 6 – Sketching views and recording design decisions 149
Step 7 – Performing analysis of current design and reviewing the iteration goal and design objectives 149
Step 8 – Iterating if necessary 149
Microsoft's technique for architecture and design 150
Step 1 – Identifying architecture objectives 150

Step 2 – Identifying key scenarios 151
Step 3 – Creating application overview 151
Determining your application type 152
Identifying your deployment constraints 152
Identifying important architecture design styles 152
Determining relevant technologies 152
Step 4 – Identifying key issues 153
Step 5 – Defining candidate solutions 153
Architecture-centric design method (ACDM) 153
Step 1 – Discovering architectural drivers 155
Step 2 – Establishing project scope 155
Step 3 – Creating notional architecture 155
Step 4 – Architectural review 155
Step 5 – Production go/no-go 156
Step 6 – Experiment planning 156
Step 7 – Experimenting with and refining the architecture 156
Production planning and production 157
Architecture development method (ADM) 157
The Open Group Architecture Framework (TOGAF) 157
TOGAF architecture domains 158
TOGAF documentation 158
Phases of the ADM 159
Phase A – Architecture vision 160
Phase B – Business architecture 160
Phase C – Information systems architectures 161
Phase D – Technology architecture 161
Phase E – Opportunities and solutions 161
Phase F – Migration planning 162
Phase G – Implementation governance 162
Phase H – Architecture change management 162
Tracking the progress of the software architecture's design 162
Using a backlog to track the architecture design progress 163
Prioritizing the backlog 163
DIVE criteria 164
Dependencies 164
Insure against risks 164
Business value 164
Estimated effort 164
Active and dynamic architecture backlogs 164
Summary 165
Chapter 6: Software Development Principles and Practices 167
Designing orthogonal software systems 168
Loose coupling 168
Types of coupling 169
Content coupling 169
Common coupling 169
External coupling 170

Control coupling 170
Stamp coupling (data-structured coupling) 170
Data coupling 170
Message coupling 171
No coupling 171
The Law of Demeter (LoD) / principle of least knowledge 171
Designing for loose coupling 171
High cohesion 172
Types of cohesion 172
Coincidental cohesion 172
Logical cohesion 173
Temporal cohesion 173
Procedural cohesion 173
Communicational cohesion 174
Sequential cohesion 174
Functional cohesion 174
Designing for high cohesion 175
Minimizing complexity 175
KISS principle – "Keep It Simple, Stupid" 176
Origin of KISS 176
Applying KISS to software 177
Don't make it overly simple 177
DRY – "Don't Repeat Yourself" 178
Copy-and-paste programming 178
Magic strings 178
How to avoid duplication 180
Don't make things overly DRY 180
Information hiding 180
Reasons for information hiding 181
What needs to be exposed/hidden? 181
YAGNI – "You Aren't Gonna Need It" 182
Avoid over-engineering a solution 182
Situations where YAGNI doesn't apply 182
Separation of Concerns (SoC) 183
Following SOLID design principles 184
Single Responsibility Principle (SRP) 184
Open/Closed Principle (OCP) 187
Liskov Substitution Principle (LSP) 189
Interface Segregation Principle (ISP) 191
Dependency Inversion Principle (DIP) 194
Inversion of Control (IoC) 195
Dependency Injection (DI) 196
Benefits of DI 196
DI patterns 197
DI containers 200
Helping your team succeed 201
Unit testing 201
What are unit tests? 201
Benefits of unit testing 202
Properties of a good unit test 202

Atomic 202
Deterministic 203
Automated and repeatable 203
Isolated and independent 203
Easy to set up and implement 203
Fast 203
The AAA pattern 204
Arrange 204
Act 204
Assert 204
Naming conventions for unit tests 204
Unit test class names 205
Unit test method names 205
Code coverage for unit tests 205
Keeping unit tests up to date 206
Setting up development environments 206
Providing a README file 207
Pair programming 208
Benefits of pair programming 208
Using pair programming when it is needed 209
Reviewing deliverables 209
Code reviews 209
Formal inspections 210
Roles for a formal inspection 211
Inspection meeting and follow-up 211
Walkthroughs 212
Summary 212

Chapter 7: Software Architecture Patterns 213
Software architecture patterns 214
Using software architecture patterns 214
Overusing architecture patterns 215
Understanding the difference between architecture styles and architecture patterns 215
Layered architecture 216
Open versus closed layers 216
Tiers versus layers 217
Advantages of layered architectures 218
Disadvantages of layered architectures 219
Client-server architecture (two-tier architecture) 220
Using stored procedures for application logic 221
N-tier architecture 222
Presentation tier 223
Business tier 223
Data tier 224
Event-driven architecture 224
Event channels 224
Message queues 225
The point-to-point channel pattern 225
Message topics 225

The publish-subscribe pattern 225
Event-driven architecture topologies 226
The mediator topology 226
Event mediator implementations 227
The broker topology 227
Event processing styles 228
Simple event processing (SEP) 229
Event stream processing (ESP) 229
Complex event processing (CEP) 229
Types of event-driven functionality 230
Event notification 230
Event-carried state transfer 230
Event-sourcing 231
The Model-View-Controller pattern 232
Model 233
View 233
Controller 234
Advantages of the MVC pattern 234
Disadvantages of the MVC pattern 234
The Model-View-Presenter pattern 235
Model 235
View 235
Presenter 236
The Model-View-ViewModel pattern 236
Model 237
View 237
ViewModel 237
The Command Query Responsibility Segregation pattern 238
The query model and the command model 239
Using event-sourcing with CQRS 240
Advantages of CQRS 240
Disadvantages of CQRS 241
Service-oriented architecture 241
What makes SOA different from other distributed solutions? 242
Benefits of using a SOA 242
Increases alignment between business and technology 243
Promotes federation within an organization 243
Allows for vendor diversity 243
Increases intrinsic interoperability 244
Works well with agile development methodologies 244
Cost-benefit analysis of SOA 244
Challenges with SOA 245
Key principles for service orientation 246
Standardized service contract 246
Service loose coupling 246
Service abstraction 247
Service reusability 247

Service autonomy 247
Service statelessness 248
Service discoverability 248
Service composability 248
SOA delivery strategies 248
The top-down strategy 249
The bottom-up strategy 249
The agile strategy 250
Service-oriented analysis 250
Defining business automation requirements 250
Identifying existing automation systems 251
Modeling candidate services 251
Service layers and service models 251
Task service 253
Entity service 253
Utility service 253
Service-oriented design 253
Service interface design 254
Service interface granularity 255
Service registries 255
Service descriptions 256
Structuring namespaces 257
Orchestration and choreography 258
Summary 259

Chapter 8: Architecting Modern Applications 260
Monolithic architecture 260
Benefits of a monolithic architecture 261
Drawbacks of a monolithic architecture 262
Microservice architecture 263
SOA done right 263
Characteristics of microservice architecture 264
Small, focused services 264
Well-defined service interfaces 265
Autonomous and independently deployable services 265
Independent data storage 265
Better fault isolation 266
Communicating with lightweight message protocols 266
Designing polyglot microservices 267
Polyglot programming 268
Polyglot persistence 268
Using too many technologies 269
Considering service granularity 269
Nanoservices 269
Sharing dependencies between microservices 270
Stateless versus stateful microservices 271
Service discovery 271
Using a service registry 271
Self-registration pattern 272

Third-party registration pattern 272
Types of service discovery 273
Client-side discovery pattern 274
Server-side discovery pattern 274
Using microservices is not for everyone 275
Serverless architecture 276
Function as a Service (FaaS) 277
Backend as a Service (BaaS) 278
Advantages of serverless architectures 279
Cost savings 279
Scalable and flexible 279
Focus on building your core products 279
Polyglot development 280
Disadvantages of serverless architectures 280
Difficulties with debugging and monitoring 280
Multitenancy issues 280
Vendor lock-in 281
Complexity of designing many functions 281
Not as many runtime optimizations 281
Still immature 281
Taking a hybrid approach to serverless 282
Function deployment 282
Function invocation 283
Synchronous request 283
Asynchronous request (message queue) 284
Message stream 284
Batch job 285
Cloud-native applications 285
Reasons to move to the cloud 286
Reducing costs 286
Greater flexibility and scalability 286
Automatic updates 286
Disaster recovery 287
What are cloud-native applications? 287
Containerized 287
Dynamically orchestrated 288
Microservices-oriented 289
No downtime 289
Continuous delivery 289
Support for a variety of devices 290
Twelve-factor apps 290
Codebase 291
Dependencies 291
Configuration 292
Backing services 292
Build/Release/Run 292
Processes 293
Port binding 293
Concurrency 294

Disposability 294
Development/production parity 294
Logs 295
Administrative processes 295
Summary 296

Chapter 9: Cross-Cutting Concerns 297
Cross-cutting concerns 298
General guidelines for cross-cutting concerns 299
Identifying cross-cutting concerns 299
Using open-source and third-party solutions 299
Maintaining consistency 299
Avoiding scattered solutions 300
Avoiding tangled solutions 300
Implementing cross-cutting concerns 301
Using dependency injection (DI) 301
Using the decorator pattern 302
Aspect-oriented programming 306
Types of advice 307
Weaving 308
Compile-time weaving 309
Runtime weaving 309
Types of cross-cutting concerns 310
Caching 310
Configuration management 311
Auditing 312
Security 312
Exception management 312
Logging 313
Understanding log levels 313
Routing log entries 314
Using Elastic Stack 315
Elasticsearch 315
Logstash 316
Kibana 316
Beats 317
Cross-cutting concerns for microservices 317
Leveraging a microservice chassis 317
Using the sidecar pattern 318
Summary 320

Chapter 10: Performance Considerations 321
The importance of performance 321
Performance affects user experience 322
Bounce rate 322
Conversion rate 322
Performance is a requirement 323

Page speed affects search rankings 323
Defining performance terminology 323
Latency 324
Throughput 324
Bandwidth 324
Processing time 325
Response time 325
Workload 325
Utilization 325
Taking a systematic approach to performance improvement 326
Profiling an application 326
Instrumentation 327
Statistical profilers 327
Analyzing the results 328
Implementing changes 328
Monitoring results 329
Server-side caching 329
Caching data in distributed applications 330
Using a private caching strategy 330
Using a shared caching strategy 330
Priming the cache 331
Invalidating cached data 331
Expiring data 331
Evicting data 331
Cache usage patterns 332
Cache-aside pattern 332
Read-through pattern 332
Write-through pattern 333
Write-behind pattern 333
Improving web application performance 333
Leveraging HTTP caching 333
Using a validation token 334
Specifying cache-control directives 334
Taking advantage of compression 335
File compression 335
Lossless compression 336
Lossy compression 336
Content-encoding (end-to-end) compression 337
Minifying resources 337
Bundling resources 338
Using HTTP/2 341
Multiplexing 342
Server push 342
Header compression 343
Implementing HTTP/2 343
Using content delivery networks (CDNs) 344
Optimizing web fonts 344

Optimizing the critical rendering path 346
Database performance 347
Designing an efficient database schema 347
Normalizing a database 347
Denormalizing a database 348
Identifying primary and foreign keys 348
Selecting the most appropriate data types 349
Using database indexes 349
Primary/clustered indexes 349
Secondary/non-clustered indexes 350
Having too many indexes 350
Scaling up and out 351
Database concurrency 352
Database transactions 352
Optimistic versus pessimistic concurrency control 352
CAP theorem 353
ACID model 354
Atomicity 354
Consistency 354
Isolation 354
Durability 355
BASE model 355
Basic availability 355
Soft state 355
Eventual consistency 355
Summary 356
Chapter 11: Security Considerations 357
Securing software systems 358
The three states of information 358
The CIA triad 359
Confidentiality 359
Integrity 359
Availability 360
Threat modeling 360
Decomposing an application 361
Identifying and categorizing potential threats 361
STRIDE threat model 362
Spoofing identity 362
Tampering with data 362
Repudiation 362
Information disclosure 363
Denial-of-service 363
Elevation of Privilege 363
Prioritizing potential threats 363
DREAD risk assessment model 364
Damage potential 364
Reproducibility 364
Exploitability 365
Affected users 365

Discoverability 365
Responses to threats 365
Avoiding the risk 366
Transferring the risk 366
Accepting the risk 366
Mitigating the risk 367
Types of security control 367
Physical security controls 367
Administrative controls 368
Technical security controls 369
Prevention 369
Detection 370
Response 370
Secure by design 370
Minimizing the attack surface 371
Defense in depth 371
Principle of least privilege (PoLP) 371
Avoiding security by obscurity 372
Keep software designs simple 372
Secure by default 372
Default deny 373
Validating input 373
Secure the weakest link 373
Security must be usable 373
Fail securely 374
Cryptography 374
Encryption 374
Symmetric (secret key) encryption 375
Asymmetric (public key) encryption 375
Cryptographic hash functions 375
Identity and access management (IAM) 377
Authentication 377
What is multi-factor authentication (MFA)? 378
Authorization 378
Storing plaintext passwords 379
Storing encrypted passwords 379
Storing hashed passwords 379
Using domain authentication 380
Implementing a centralized identity provider (IdP) 380
OAuth 2/OpenID Connect (OIDC) 381
OAuth 2 roles 381
JSON web token (JWT) 382
Header 383
Payload 383
Signature 384
Authorizing with the authorization server 384
Most common web application security risks 385
Injection 385

Broken authentication 385
Sensitive data exposure 386
XML external entity (XXE) attack 386
Broken access control 388
Security misconfiguration 388
Cross-site scripting (XSS) 389
Insecure deserialization 389
Using components with known vulnerable components 390
Insufficient logging and monitoring 390
Unvalidated redirects and forwards 391
Summary 391
Chapter 12: Documenting and Reviewing Software Architectures 393
Uses of software architecture documentation 394
Communicating your architecture to others 394
Assisting the development team 395
Educates team members 395
Providing input for software architecture reviews 395
Allowing for the reuse of architectural knowledge 396
Help the software architect 396
Creating architecture descriptions (ADs) 397
Software architecture views 397
Software architecture notations 398
Informal software architecture notations 398
Semiformal software architecture notations 399
Formal software architecture notations 399
Including design rationales 400
Overview of the Unified Modeling Language (UML) 401
Types of modeling 401
Class diagrams 401
Visibility 402
Association 402
Aggregation 403
Composition 404
Multiplicity 404
Dependency 405
Generalization/specialization 405
Realization 406
Component diagrams 407
Package diagrams 409
Deployment diagrams 410
Use case diagrams 411
Sequence diagrams 413
Lifeline 414
Activation boxes 414
Messages 414
Loops 415
Optional flows 416
Alternative flows 416
Activity diagrams 417

Start/end nodes 418
Actions/Control flow 419
Decision/merge nodes 419
Fork/join nodes 420
Reviewing software architectures 420
Software architecture analysis method (SAAM) 421
Scenario-based analysis of software architecture 421
SAAM steps 422
Step 1 – Develop scenarios 422
Step 2 – Describe the architecture 422
Step 3 – Classify and prioritize scenarios 422
Step 4 – Evaluate scenarios 423
Step 5 – Assess scenario interaction 423
Step 6 – Create an overall evaluation 423
Architecture tradeoff analysis method (ATAM) 424
ATAM participant roles 424
ATAM phases 424
Phase 0 – Partnership and preparation 425
Phase 1 – Evaluation 425
Phase 2 – Evaluation (continued) 428
Phase 3 – Follow-up 429
Active design review (ADR) 429
ADR steps 430
Step 1 – Prepare the documentation for review 430
Step 2 – Identify the specialized reviews 431
Step 3 – Identify the reviewers needed 431
Step 4 – Design the questionnaires 431
Step 5 – Conduct the review 431
Active reviews of intermediate designs (ARID) 432
ARID participant roles 432
ARID phases 432
Phase 1 – Pre-meeting 433
Phase 2 – Review meeting 434
Summary 435
Chapter 13: DevOps and Software Architecture 436
DevOps 437
CALMS 437
Culture 438
Automation 439
Lean 440
Measurement 441
Sharing 441
Why DevOps? 441
DevOps toolchain 442
DevOps practices 444
Continuous integration (CI) 445
Automated builds 445
Software versioning 446
Automated testing 447
Continuous delivery (CD) 448

Continuous deployment 448
Architecting for DevOps 449
Important quality attributes for DevOps 449
Some architecture patterns complement DevOps 451
Deploying to the cloud 451
Cloud types 452
Public cloud 452
Private cloud 452
Hybrid cloud 453
Cloud models 454
Infrastructure as a service (IaaS) 455
Containers as a Service (CaaS) 456
Platform as a Service (PaaS) 458
Serverless/Function as a Service (FaaS) 459
Software as a Service (SaaS) 459
Summary 460
Chapter 14: Architecting Legacy Applications 461
Legacy applications 461
Issues with legacy applications 462
Why are legacy applications used? 463
More than just code 464
Refactoring legacy applications 464
Making legacy code testable 465
Benefits of unit testing 465
Refactoring for unit tests 466
Where to start writing tests? 466
Removing redundant code 467
Unreachable code 467
Dead code 468
Commented-out code 468
Duplicate code 468
Using tools to refactor 469
Making small, incremental changes 469
Transforming monoliths to microservices 469
Migrating to the cloud 470
The 6 R's 471
Remove (or retire) 471
Retain 471
Replatform 472
Rehost 472
Repurchase 472
Refactor (or re-architect) 472
Moving to an agile approach 473
Modernizing build and deployment processes 473
Automating the build and deployment processes 474
Practicing continuous integration (CI) 474

Practicing continuous delivery (CD) 475
Updating the build tools 475
Integrating with legacy applications 475
Confirming the need for integration 476
Determining the type of integration 476
Sharing functionality between systems 477
Performing data integration 477
Summary 477

Chapter 15: The Soft Skills of Software Architects 479
Soft skills 479
Communication 480
Communicating the architecture 480
Communicating about quality attributes 481
Communicating expectations 481
The 7 Cs of communication 481
Clarity 482
Conciseness 482
Concreteness 483
Courteousness 483
Consideration 483
Correctness 484
Completeness 484
Listening skills 485
Hearing is not listening 485
Showing empathy 485
Tips for effective listening 485
Giving presentations 486
The 4 Ps of presentations 487
Plan 487
Prepare 487
Practice 488
Present 489
Leadership 489
Getting others to follow you 489
Dealing with challenges 490
Being a technical leader 490
Taking responsibility 490
Focusing on others 491
Delegating tasks 491
Driving change 492
Communication and leadership 492
Mentoring others 493
Leading by example 493
Depending on others 494
Negotiation 494
How negotiation skills may be used 495

Informal/formal negotiations 495
Working with remote resources 497
Benefits of using remote resources 497
Challenges when using remote resources 497
Communication 498
Cultural differences 498
Impromptu meetings 498
New employee onboarding 498
Work quality 499
Confidential company data 499
Summary 500
Chapter 16: Evolutionary Architecture 501
Change is inevitable 501
Reasons for change 502
Expecting change 502
Lehman's laws of software evolution 503
Lehman's software categories 503
S-type systems 503
P-type systems 504
E-type systems 504
The laws 504
Law I – Continuing change 505
Law II – Increasing complexity 505
Law III – Self-regulation 505
Law IV – Conservation of organizational stability 506
Law V – Conservation of familiarity 506
Law VI – Continuing growth 506
Law VII – Declining quality 506
Law VIII – Feedback system 507
Designing evolutionary architectures 507
Making guided architectural changes 508
Fitness functions 508
Categories of fitness functions 509
Examples of fitness functions 510
Making incremental changes 511
Architectural changes across multiple dimensions 512
Loosely coupled architectures 512
Designing evolvable APIs 513
Applying Postel's Law to APIs 514
Using standards in your software system 514
Last responsible moment (LRM) 515
Summary 515
Chapter 17: Becoming a Better Software Architect 517
Practicing continuous learning 518
Improving the breadth and depth of your knowledge 518
Avoiding the law of the instrument 519

Finding the time for learning 519
Ways to keep your skills sharp 520
Participating in open source projects 521
Creating your own open source project 521
Writing your own blog 522
Increasing your visibility 523
Starting your own blog 523
Things to avoid 524
Spending time teaching others 524
Finding opportunities to teach 525
Being a mentor 525
Trying new technologies 526
Continuing to write code 527
Assigning yourself coding tasks 527
Working on your own project 527
Reading code 528
Attending user groups and conferences 528
Presenting at a user group or conference 529
Meeting new people 530
Taking responsibility for your work 530
Attending to your well-being 531
Being proud of your work 531
Summary 531

Other Books You May Enjoy 533

Index 536

Preface

Modern software systems are complex, and the software architect role is a challenging one. This book was written to help software developers transition into the role of a software architect, and to assist existing software architects to be successful in their role. It helps readers understand how being a software architect is different than being a developer and what it takes to be an effective software architect.

This comprehensive guide to software architecture begins by explaining what software architecture entails, the responsibilities of the software architect position, and what you will be expected to know. Software architects must have technical and non-technical skills, and they must have both breadth and depth of knowledge.

The book progress to covering non-technical topics such as the importance of understanding your organization's business, working in the context of an organization, and gathering requirements for software systems. It then takes a deep dive into technical topics such as software quality attributes, software architecture design, software development best practices, architecture patterns, how to improve performance, and security considerations.

After reading this book, you should have a familiarity with the many topics related to software architecture and understand how to be a software architect. Technologies and practices may change over time, but the book lays a strong foundation on which you can build a successful career as a software architect.

Who this book is for

This book is aimed at senior developers and software architects who want to learn how to be a successful software architect. Readers should be experienced software development professionals who want to advance in their career and become a software architect. It covers a wide range of topics that will help readers learn what it takes to be effective in the software architect role.

What this book covers

`Chapter 1`, *The Meaning of Software Architecture*, begins the book by providing a definition of software architecture. The book establishes what makes up a software architecture and the reasons why it is important to a software system. It also details the software architect role, including the responsibilities of software architects and what they are expected to know.

`Chapter 2`, *Software Architecture in an Organization*, focuses on software architecture in the context of an organization. It covers the different types of software architect roles and software development methodologies that you may encounter. Non-technical topics such as project management, office politics, and risk management are explained. The development of software product lines and the creation of architectural core assets are also covered.

`Chapter 3`, *Understanding the Domain*, discusses the business aspects of being a software architect. It covers topics such as familiarizing yourself with your organization's business, **domain-driven design** (**DDD**), and how to effectively elicit requirements for the software system from stakeholders.

`Chapter 4`, *Software Quality Attributes*, covers software quality attributes and their importance to a software architecture. Some common software quality attributes are presented, including maintainability, usability, availability, portability, interoperability, and testability.

`Chapter 5`, *Designing Software Architectures*, concentrates on the important topic of software architecture design. It details what is involved with architecture design and its importance to a software system. The chapter discusses different approaches to architecture design, the drivers for it, and the design principles that can be leveraged during the process.

The chapter presents the use of various systematic approaches to software architecture design, including **attribute-driven design** (**ADD**), Microsoft's technique for architecture and design, the **architecture-centric design method** (**ACDM**), and the **architecture development method** (**ADM**).

`Chapter 6`, *Software Development Principles and Practices*, describes proven software development principles and practices that can be used to build high-quality software systems. Concepts such as loose coupling and high cohesion are covered, as well as principles such as KISS, DRY, information hiding, YAGNI, and the **Separation of Concerns** (**SoC**).

The chapter includes a discussion of the SOLID principles, which include the single responsibility, open/closed, Liskov substitution, interface segregation, and dependency inversion principles. The chapter closes with topics related to helping your team succeed, including unit testing, setting up development environments, pair programming, and reviewing deliverables.

`Chapter 7`, *Software Architecture Patterns*, discusses one of the most useful software architecture design concepts. Learning the architecture patterns that are available to you and when to properly apply them is a key skill for software architects. The chapter details a number of software architecture patterns, including layered architecture, **event-driven architecture** (**EDA**), **Model-View-Controller** (**MVC**), **Model-View-Presenter** (**MVP**), **Model-View-ViewModel** (**MVVM**), **Command Query Responsibility Segregation** (**CQRS**), and **Service-Oriented Architecture** (**SOA**).

`Chapter 8`, *Architecting Modern Applications*, explains the software architecture patterns and paradigms that are used with modern applications deployed to the cloud. After describing a monolithic architecture, the chapter details microservices architecture, serverless architecture, and cloud-native applications.

`Chapter 9`, *Cross-Cutting Concerns*, places its focus on functionality that is used in multiple areas of the system. It explains how to handle cross-cutting concerns in your applications. Topics covered include using **Dependency Injection** (**DI**), the decorator pattern, and **aspect-oriented programming** (**AOP**) to implement cross-cutting concerns. The chapter also provides a look at different cross-cutting concerns, including caching, configuration management, auditing, security, exception management, and logging.

`Chapter 10`, *Performance Considerations*, takes a close look at performance. It describes the importance of performance and techniques to improve it. Topics such as server-side caching and database performance are discussed. An examination of web application performance is included in the chapter, including coverage of HTTP caching, compression, minimizing and bundling of resources, HTTP/2, **content delivery networks** (**CDNs**), optimizing web fonts, and the critical rendering path.

`Chapter 11`, *Security Considerations*, covers the critical topic of software application security. Security concepts such as the **confidentiality**, **integrity**, and **availability** (**CIA**) triad and threat modeling are presented. The chapter provides readers with various principles and practices for creating software that is secure by design.

`Chapter 12`, *Documenting and Reviewing Software Architectures*, places its focus on software architecture documentation and reviewing software architectures. It describes the various uses for software architecture documentation and explains how to use UML to document a software architecture. The chapter discusses various software architecture review methods, including the **software architecture analysis method** (**SAAM**), **architecture tradeoff analysis method** (**ATAM**), **active design review** (**ADM**), and **active reviews of intermediate designs** (**ARID**).

`Chapter 13`, *DevOps and Software Architecture*, provides coverage of the culture, practices, tools, and culture of DevOps. The chapter explains key DevOps practices such as **continuous integration** (**CI**), **continuous delivery** (**CD**), and **continuous deployment**.

`Chapter 14`, *Architecting Legacy Applications*, provides readers with an understanding of how to work with legacy applications. The widespread use of legacy applications makes this topic important for software architects. The chapter covers refactoring legacy applications and how to migrate them to the cloud. It discusses modernizing build and deployment processes for legacy applications as well as how to integrate with them.

`Chapter 15`, *The Soft Skills of Software Architects*, is all about the soft skills that software architects should possess to be an effective software architect. After describing what soft skills are, the chapter proceeds to topics such as communication, leadership, negotiation, and working with remote resources.

`Chapter 16`, *Evolutionary Architecture*, teaches how to design software systems so that they have the ability to adapt to change. It explains that change is inevitable, so software architects should design software architectures that can evolve over time. The chapter explains some of the ways that change can be handled and it introduces the use of fitness functions to ensure that an architecture continues to meet its desired architectural characteristics as it undergoes change.

`Chapter 17`, *Becoming a Better Software Architect*, stresses to readers that the process of career development is an ongoing one. After becoming a software architect, one must seek to continuously gain new knowledge and improve their skills. The chapter details ways that a software architect can practice self-improvement, including continuous learning, participating in open source projects, writing your own blog, spending time teaching others, trying new technologies, continuing to write code, attending user groups and conferences, taking responsibility for your work, and attending to your general well-being.

To get the most out of this book

Although readers should have experience of software development, no specific prerequisites are required to begin reading this book. All of the information that you need is contained in the various chapters. The book does not require knowledge of any particular programming language, framework, or tool. The code snippets in the book that illustrate various concepts are written in C#, but they are simple enough that prior C# experience is not necessary.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/SoftwareArchitectsHandbook_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Now we can use that constant in our `GetFilePath` method."

A block of code is set as follows:

```
public string GetFilePath()
{
    string result = _cache.Get(FilePathCacheKey);

    if (string.IsNullOrEmpty(result))
    {
        _cache.Put(FilePathCacheKey, DetermineFilePath());
        result = _cache.Get(FilePathCacheKey);
    }

    return result;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public string GetFilePath()
{
    string result = _cache.Get(FilePathCacheKey);

    if (string.IsNullOrEmpty(result))
    {
        _cache.Put(FilePathCacheKey, DetermineFilePath());
        result = _cache.Get(FilePathCacheKey);
    }

    return result;
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "In a direct dependency graph, at compile-time, **Class A** references **Class B**, which references **Class C**"

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
The Meaning of Software Architecture

A comprehensive look at software architecture must first begin with its definition. This chapter provides reasons as to why software architecture plays an important role in a software project, and the benefits of having a good architectural design.

It is also important to understand the stakeholders and team members who are affected by the software architecture of a system. The chapter will go into detail about the software architect's role, what software architects are supposed to know, and whether the role is right for you.

In this chapter, we will cover the following topics:

- What is software architecture?
- Why is software architecture important?
- Who are the consumers of software architectures?
- What is the software architect role?

What is software architecture?

What exactly is software architecture? You probably have your own ideas about what it is, based on your knowledge and experiences. Certainly, there are plenty of definitions out there. If you do an online search or ask various friends and colleagues, you will get varying answers. The definition is somewhat subjective and influenced by the viewpoints and perceptions of the individual who is providing the definition. However, there are some core concepts that are essential to software architecture, and before we delve into deeper topics, establishing a common understanding of what software architecture entails is imperative.

Using the word *architecture* for software originated from similarities with the construction industry. When the term was first used, the Waterfall software development methodology was common and it dictated that large, up-front designs needed to be completed before any code was written. Similar to the architecture of a building, which necessitates a lot of planning before construction takes place, so it was with software as well.

In modern software design, the relationship between the construction and software industries is no longer as close. Software methodologies now focus on developing software applications that are highly adaptable and can be changed easily over time, resulting in less of a need for rigid, upfront planning. However, software architecture still consists of early design decisions that can be difficult to change later.

ISO/IEC/IEEE 42010 standard definition

There is a standard definition for software architecture, which resulted from a joint effort between the **International Organization for Standardization** (**ISO**) and the **Institute of Electrical and Electronics Engineers** (**IEEE**). ISO/IEC/IEEE 42010 systems and software engineering's architecture description is an international standard that defines software architecture as:

> *"Fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution."*

The standard makes the following main points:

- A software architecture is a fundamental part of a software system
- A software system is situated in an environment, and its software architecture takes into consideration the environment in which it must operate
- An architecture description documents the architecture and communicates to stakeholders how the architecture meets the system's needs
- Architecture views are created from the architecture description, and each view covers one or more architecture concerns of the stakeholders

What makes up a software architecture?

In the book, *Software Architecture in Practice, 2nd Edition*, a definition of software architecture is given as:

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."*

A software system contains structures, and this definition notes that a software system is made up of one or more of them. It is the combination of these that forms the overall software architecture. A large software project may have multiple teams working on it, each responsible for a particular structure.

Software architecture is an abstraction

Software architecture is an abstraction of a software system. The structures of a software system consist of its elements. Software architecture concerns itself with defining and detailing the structures, their elements, and the relationships of those elements with each other.

Software architecture focuses on the public aspects of the elements, and how they interact with each other. For elements, this may take the form of their public interfaces. It does not deal with the private implementation details of the elements. While the behavior of the elements does not have to be exhaustively documented, care should be taken in understanding how elements have to be designed and written so that they can properly interact with each other.

Software architecture is about the important stuff

Computer scientist Ralph Johnson, who co-authored *Design Patterns: Elements of Reusable Object-Oriented Software*, once said:

> *"Architecture is about the important stuff. Whatever that is."*

Software projects vary, and the amount of design effort, time, focus, and documentation devoted to particular aspects of a software architecture differ. Ultimately, software architecture consists of important design decisions that shape the system. It is made up of the structures and components that are significant to the quality, longevity, and usefulness of the system.

Software architecture consists of some of the earliest decisions that are made for a software system and some of the hardest to change. In modern software development, the architecture should anticipate change, and be designed in such a way as to maximize the potential of adapting and evolving to this change. We will be discussing evolutionary architecture in `Chapter 16`, *Evolutionary Architecture*.

Why is software architecture important?

Why should we care about software architecture anyway? Sometimes a developer just wants to jump right in and start coding.

Software architecture is the foundation of a software system. Like other types of engineering, the foundation has a profound effect on the quality of what is built on top of it. As such, it holds a great deal of importance in terms of the successful development, and eventual maintenance, of the system.

Software architecture is a series of decisions. Some of the earliest decisions come from designing the architecture, and these carry a high degree of importance because they affect the decisions that come after it.

Another reason software architecture is important is because all software systems have an architecture. Even if it comprised just one structure with one element, there is an architecture. There are software systems that don't have a formal design and others that don't formally document the architecture, but even these systems still have an architecture.

The greater the size and complexity of a software system, the more you will need a well thought-out architecture in order to succeed. Software architecture provides a number of benefits when done properly, which greatly increase the chances that the software system will succeed.

A proper foundation laid down by a software system's architecture yields a number of benefits. Let's take a deeper look at those benefits.

Defining a solution to meet requirements

Software strives to meet all functional, non-functional, technical, and operational requirements. Working closely with stakeholders, such as domain experts, business analysts, product owners, and end users, allows requirements to be identified and understood. A software architecture defines a solution that will meet those requirements.

Software architecture is the foundation for software, so software systems that lack a solid architecture make it more difficult to meet all of the requirements. Poor architectures will lead to implementations that fail to meet the measurable goals of quality attributes, and they are typically difficult to maintain, deploy, and manage.

Enabling and inhibiting quality attributes

Software architecture either enables quality attributes or inhibits them. Quality attributes are measurable and testable properties of a system. Some examples of quality attributes include maintainability, interoperability, security, and performance.

They are *non-functional* requirements of a software system as opposed to its features, which are *functional* requirements. Quality attributes and how they satisfy the stakeholders of the system are critical, and software architecture plays a large role in ensuring that quality attributes are satisfied. The design of a software architecture can be made to focus on certain quality attributes at the cost of others. Quality attributes may be in conflict with each other. A software architecture, when designed properly, sets out to achieve agreed-upon and validated requirements related to quality attributes.

Giving you the ability to predict software system qualities

When you look at a software architecture and its documentation, you can predict the software system's qualities. Making architecture decisions based on quality attributes makes it easier to fulfill those requirements. You want to start thinking about quality attributes as early as possible in the software development process as it is much more difficult (and costly) to make changes to fulfill them later. By thinking about them up front, and using modeling and analysis techniques, we can ensure that the software architecture can meet its non-functional requirements.

If you are not able to predict if a software system will fulfill quality attributes until it is implemented and tested, then costly and time-consuming rework may be necessary. A software architecture allows you to predict a software system's qualities and avoid costly rework.

Easing communication among stakeholders

Software architecture and its documentation allow you to communicate the software architecture and explain it to others. It can form the basis for discussions related to aspects of the project, such as costs and duration. We will discuss this topic further when we go into detail about software architecture in an organization.

A software architecture is abstract enough that many stakeholders, with little or no guidance, should be able to reason about the software system. Although different stakeholders will have different concerns and priorities in terms of what they want to know about the architecture, providing a common language and architecture design artifacts allows them to understand the software system. It is particularly useful for large, complex systems that would otherwise be too difficult to fully understand. As requirements and other early decisions are made for the software system, a formal software architecture plays an important role and facilitates negotiations and discussions.

Managing change

Changes to a software system are inevitable. The catalyst for change can come from the market, new requirements, changes to business processes, technology advances, and bug fixes, among other things.

Some view software architecture as inhibiting agility and would prefer to just let it emerge without up-front design. However, a good software architecture helps with both implementing and managing changes. Changes fall into one of the following categories:

- Limited to a single element
- Involve a combination of elements, but do not require any architectural changes
- Require an architectural change

Software architecture allows you to manage and understand what it would take to make a particular change. Furthermore, a good architecture reduces complexity so that most of the changes that need to be made can be limited to a single element or just a few elements, without having to make architectural changes.

Providing a reusable model

An established architecture might be used again within an organization for other products in a product line, particularly if the products have similar requirements. We'll discuss an organization's product lines, reuse of architecture, and the benefits in the next chapter. For now, simply recognize that, once a software architecture is completed, documented, understood, and used in a successful implementation, it can be reused.

When code is reused, resources, such as time and money, are saved. More importantly, the quality of software that takes advantage of reuse is increased because the code has already been tested and proven. The increase in quality alone translates to savings in resources.

When a software architecture is reused, it is not just code that is reused. All of the early decisions that shaped the original architecture are leveraged as well. The thought and effort that went into the requirements necessary for the architecture, particularly non-functional requirements, may be applicable to other products. The effort that went into making those decisions does not necessarily have to be repeated. The experience gained from the original architectural design can be leveraged for other software systems.

When a software architecture is reused, it is the architecture itself, and not just the software product, that becomes an asset to the organization.

Imposing implementation constraints

A software architecture introduces constraints on implementation and restricts design choices. This reduces the complexity of a software system and prevents developers from making incorrect decisions.

If the implementation of an element conforms to the designed architecture, then it is abiding by the design decisions made by the architecture. Software architecture, when done properly, enables developers to accomplish their objectives and prevents them from implementing things incorrectly.

Improving cost and effort estimates

Project managers ask questions such as: When is it going to be done? How long is it going to take? How much is it going to cost? They need this type of information to properly plan resources and monitor progress. One of the many duties of a software architect is to assist project management by providing this type of information and assisting with determining the necessary tasks and estimates for those tasks.

The design of the software architecture itself affects what types of task will be necessary for implementation. As a result, work-breakdown of tasks is dependent on the software architecture and the software architect can assist project management with the creation of the tasks.

Two major approaches to project management estimation are as follows:

- **Top-down approach**: This starts with the final deliverables and goals and breaks them down into smaller packages of work
- **Bottom-up approach**: This starts with specific tasks first, and groups them together into packages of work

For some projects, a project manager may take a more top-down approach, while developers who are going to be working on specific tasks may take a bottom-up perspective. With the experience and knowledge that most software architects possess, they can potentially assist with either approach. A combination of these approaches, where tasks are looked at from both viewpoints, can lead to the best estimates.

It can be helpful when project managers, the software architect, and the developers work together to provide estimates. The most accurate estimates can be obtained by mutual discussions between team members until a consensus is achieved. Sometimes during the consensus building, someone on the team will provide an insight that others had not previously considered, allowing everyone to rethink their position and possibly revise their estimates.

A software system with accurate requirements that are reflected in the software architecture can avoid costly rework that would be necessary if key requirements were missed. In addition, a well-thought-out architecture reduces complexity, allowing it to be easily reasoned about and understood. Reduced complexity can result in more accurate cost and effort estimates.

Serves as training for team members

The system's architecture and its documentation serve as training for the developers on the team. By learning the various structures and elements of the system, and how they are supposed to interact, they learn the proper way in which the functionality is to be implemented.

A software development team may experience change, such as having new team members join or existing ones leave. The introduction and orientation of new members to a team often takes time. A well-thought-out architecture can make it easier for developers to transition to the team.

The maintenance phase of a software system can be one of the longest and costliest phases of a software project. Like new team members introduced during development, it is common for different developers to work on the system over time, including those introduced to maintain it. Having a solid architecture available to teach and bring aboard new developers can provide an important advantage.

Software architecture is not a silver bullet

The Mythical Man-Month by Frederick P. Brooks is one of the seminal texts in software project management. It contains various essays on software engineering. Although this book was written some time ago, and some of the references are now outdated, it provides thought-provoking advice about software development that is timeless and still applicable today:

> *"There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity."*

Fred Brooks 1986 essay, *No Silver Bullet – Essence and Accident in Software Engineering*, which is included in the twentieth anniversary edition of the book, begins with this quote. It essentially conveys the idea that there is no silver bullet in software development.

Software architecture, as well, is not a silver bullet. Although we have covered a number of reasons why software architecture is important, there is no specific architecture or combination of components that will serve as a silver bullet. It can't be thought of as a magical solution that will solve all problems. As we will learn in more detail later, software architectures are about compromises between different and sometimes conflicting requirements. Each architectural approach has pros and cons that must be weighed and evaluated. No one approach should be viewed as a silver bullet.

Who are the consumers of software architectures?

When we create a software architecture, who is it for? There are a variety of stakeholders in a software system, such as the end users of the system, business analysts, domain experts, quality assurance personnel, managers, those who may integrate with the system, and operations staff members. Each of these stakeholders is affected by the software architecture to some degree. While certain stakeholders will have access to, and be interested in, examining the software architecture and its documentation, others will not.

Some of these stakeholders are indirect consumers of the architecture in that they care about the software, and because the software architecture is the foundation of the system, they become indirect consumers of the architecture. As a software architect, you are serving these types of consumers in addition to the direct consumers. For example, end users are perhaps one of the most important stakeholders and should be a major focus. The software architecture must allow the implementation to satisfy the requirements of the end users.

When we discuss the consumers of a software architecture, we can't omit the developers who work on that software. As a software architect, you need to be thinking about your developers, whose work is directly affected by the software architecture. They are the ones who will be working on the software on a daily basis.

What is the software architect role?

Now that we know what software architecture is, the importance and benefits of it, and have an understanding that there are a variety of stakeholders who are affected by it, let's examine the software architect role. What makes someone a software architect? What does it mean to be a software architect?

Certainly, software systems can be developed without a software architect. You may have worked on a project in which no one was playing the software architect role. In some of those cases, the project may have succeeded despite that, or it may have failed because of it.

When no one is specifically given the software architect title, someone on the team may end up making architectural decisions. Such an individual is sometimes called an **accidental architect**. They haven't been given the title of software architect, but they are performing some of the same duties and making the same types of decision. Occasionally, when there is no software architect, the architectural design results from a collaboration between multiple developers.

The smaller and less complex the software system is, the more you may be able to succeed without a software architect. However, if a project is large in size and/or complexity, you are more likely to need someone to play the formal role of software architect.

Software architects are technical leaders

Software architects are technical leaders of a software project and should be committed to the project no matter what challenges arise. They provide technical guidance to management, customers, and developers. As such, they are often a liaison between technical and non-technical resources.

Although software architects have many responsibilities, foremost among them is being responsible for the technical aspects of software systems. While the software architect collaborates with others, as the technical leader the software architect is ultimately responsible for the software architecture, its design, and the architecture documentation for a software system.

Software architects perform a number of duties

Software architects are required to undertake different types of duties, not all of which are technical. Software architects combine their experience, knowledge, and skills, both technical and non-technical, to fulfill such duties. Software architects will be expected to have a firm grasp of designing software architectures, architecture patterns, and best practices.

Software architects should have the ability to foresee possible issues and design architectures to overcome them. They should be able to mitigate risks and evaluate solutions such that they can select the proper one to resolve a particular problem. While some of the skills and duties of a software architect are similar to what a senior developer might do, it is a very different role. Software architects shoulder a greater amount of responsibility, and there is a larger expectation of what a software architect brings to a project.

Senior developers have a great *depth* of knowledge regarding the technologies that they use on a project. They are highly proficient in the languages, tools, frameworks, and databases that are used in their software systems. While software architects are expected to have this depth of knowledge as well, they must also possess a wide *breadth* of knowledge. They need to be familiar with technologies that are not currently being used in the organization so that they can make informed decisions about the design of the architecture.

Ideally, software architects have the breadth of knowledge to be aware of multiple solutions to a problem and understand the trade-offs between them. It can be just as important for a software architect to understand why a particular solution will *not* work as it is to understand why one will.

Ivory tower software architects

If you find yourself in the role of a software architect, you are going to want to avoid being an **ivory tower architect**. A software architect who is in an *ivory tower* refers to one who, either by how they approach their position or because of how an organization works, is isolated from others.

If a software architect is working from an ivory tower, they may be creating an architecture based on a *perfect-world* environment that really doesn't reflect real scenarios. In addition, they may not be working closely with the developers who will be creating implementations based on the architecture.

The more that a software architect works on their own, isolated from stakeholders and other developers, the more likely they are to be out of touch with the needs of those individuals. As a result, they may be designing software architectures that do not meet the varying needs and requirements of a diverse group of stakeholders.

Software architects should take a more hands-on approach. A software architect's duties should already include involvement in a number of phases in a software life cycle, but being hands-on helps avoid being out of touch. For example, a software architect may do some of the coding with the team in order to stay more involved. Leading by example, such as using your own code to serve as references for others, is one way to take a hands-on approach while also keeping your skills sharpened.

An involved approach will help you keep abreast of what issues and difficulties developers may be facing, and what the architecture may be lacking. Leading from the trenches can be much more effective than leading from an ivory tower, and you are more likely to gain the trust and respect of your teammates. If a software architect is out of touch or misinformed, even if the perception is inaccurate, their effectiveness as a leader will be diminished.

An ivory tower architect might be someone who is viewed as commanding from above. A software architect should use their experience and knowledge to teach others, and not preach. Take opportunities to make your teammates better by teaching, but also look forward to learning from others. Teammates can and will provide valuable and insightful feedback regarding your designs.

An organization should not have processes and/or an organizational hierarchy in place that separate the architect from stakeholders. They should not be separated from the technical implementation because doing so will take the architect away from the technology and skills that made them a good candidate for being a software architect in the first place.

What are software architects expected to know?

Software architects are expected to have skills and knowledge on a variety of topics. This book focuses on many of those topics. They include non-technical duties, such as:

- Providing leadership
- Assisting project management, including cost and effort estimation
- Mentoring team members

- Helping to select team members
- Understanding the business domain
- Participating in gathering and analyzing requirements
- Communicating with a variety of technical and non-technical stakeholders
- Having a vision for future products

Technical topics that software architects should be familiar with include:

- Understanding non-functional requirements and quality attributes
- Being able to effectively design software architectures
- Understanding patterns and best practices for software development
- Having a deep knowledge of software architecture patterns, their pros and cons, and knowing when to choose one over another
- Knowing how to handle cross-cutting concerns
- Ensuring performance and security requirements are met
- Being able to document and review software architectures
- Having an understanding of DevOps and the deployment process
- Knowing how to integrate and work with legacy applications
- Being able to design software architectures that adapt to change and evolve over time

Don't be overwhelmed

If you find yourself in the software architect role for the first time, or if you are joining a team that has been working on an existing software system for some time, it can be natural to feel overwhelmed by all that you do not know. It will take time to wrap your head around everything that you will eventually need to know.

As your experience grows, you'll feel more comfortable when you start on a new project. Just like anything, experience in different situations will make you more comfortable with taking on new challenges. You'll also understand that it will take some time to become acquainted with the business domain, people, processes, technology, details, and intricacies that come with each software system.

Is the software architect role right for you?

If you care about the software that you are working on and all of its stakeholders, including the software's end users and developers, then you care about the important design decisions that go into building the software. Ultimately, that means you care about its architecture. Concerning yourself with the most important decisions can be challenging, but it can be enjoyable and rewarding for that very reason.

Software architects need to communicate with a variety of stakeholders and sometimes serve as a bridge between management, technical staff, and non-technical staff. If this is not something you want to get involved with, being a software architect may not be the best fit for you.

Software architects are passionate about technology. They have a deep understanding of the technologies they are working with and keep those skills fresh by practicing their craft and being involved with projects. They must have a large breadth of knowledge and have a familiarity with technologies that they may not be currently using on a project. It is necessary to keep up with the fast pace of change in areas such as languages, tools, and frameworks. Being aware of a range of technologies will allow you to recommend the best solution to a particular problem.

Software architects should love to learn and play with new technologies because being a software architect requires continuous learning. As someone with a lot of wisdom to share, and who will be a leader on a team, you should enjoy mentoring and teaching others. Making those who work around you better at their jobs is a part of your job.

All software applications have a purpose. Good software architects make every effort to ensure that the software applications they work on serve their purpose as best that they can. If this is something you care about, the software architect role may be right for you.

Summary

Software architecture is the structure or structures of a system, their elements, and the relationships between those elements. It is an abstraction of a software system. Software architecture is important because all software systems have an architecture, and that architecture is the foundation for the software system.

Software architecture provides a number of benefits, such as enabling and inhibiting quality attributes, allowing you to predict software system qualities, easing communication with stakeholders, and allowing you to more easily make changes. It also provides a reusable model that could be used in multiple software products, imposes implementation constraints that reduce complexity and minimizes developer errors, improves cost/effort estimates, and serves as training for new team members.

Software architects are technical leaders who are ultimately responsible for technical decisions, the architecture, and its documentation. They perform a number of duties and are expected to have knowledge of a variety of topics, both technical and non-technical. Although the role can be challenging, if you care about the software that you are working on and all of its stakeholders, then the software architect role can be extremely rewarding.

In the next chapter, we'll explore software architecture in an organization. Most software architects operate within the context of an organization, so it is important to understand the dynamics of developing software within one. The chapter will detail topics such as the various software architect roles you will typically find in an organization, software development methodologies that are used, working with project and configuration management, navigating office politics, and creating software product lines that leverage architectural reuse.

2 Software Architecture in an Organization

In the previous chapter, we discussed software architecture and the role of the software architect. In this chapter, we will explore those topics further, but in the context of an organization.

Software systems are developed to satisfy the business goals of an organization. Many software architects work as part of an organization. As a result, the organization's business goals, objectives, stakeholders, project management, and processes greatly affect the software architect and their work.

This chapter focuses on topics a software architect should be familiar with when working within an organization. We will take a look at the various types of software architecture roles that are commonly found in organizations, and the software development methodologies they adopt. You'll gain a good understanding of how you might be expected to work with project management and the dynamics of office politics.

Risk management and configuration management are two other aspects of working on software projects within an organization. Finally, we'll take a look at software product lines, and how architectural reuse can create core assets that can make building software products faster, more efficient, and of a higher quality.

In this chapter, we will cover the following topics:

- Types of software architects
- Software development methodologies
- Project management
- Office politics
- Software risk management
- Configuration management
- Software product lines

Types of software architects

The role of a software architect can vary from organization to organization. You may have also heard of a variety of job titles related to software architects, such as the following:

- Enterprise architect
- Solution architect
- Application architect
- Data architect/Information architect
- Solution architect
- Security architect
- Cloud architect

Some organizations have one or more architects who perform a combination of these roles. They may go by the title of software architect or by the title of one of these roles. In other organizations, different individuals play different architectural roles. Some companies organize their software architects so that they are in an architecture team. They collaborate with the team on architecture tasks but also work on other teams to design and implement software products.

This book does not focus on any one type of software architect. It deals with mostly technical topics, and so is geared toward a number of technical architect roles. Many of the technical, non-technical, and soft skills described in this book are required by more than one type of software architect. Even in organizations that have different types of architects, there is an overlap in their responsibilities and duties. Let's take a closer look at the different types of software architect roles and what they typically mean.

Enterprise architect

Enterprise architects are responsible for the technical solutions and strategic direction of an organization. They must work with a variety of stakeholders to understand an organization's market, customers, products, business domain, requirements, and technology.

The enterprise architect ensures that an organization's business and strategic goals are in sync with the technical solutions. They need to take a holistic view to ensure that their architecture designs, and the designs of other architects, are in line with the overall organization.

They should have both a deep and broad understanding of technology so that they can make the proper recommendations and architecture designs. They must also look to the future to ensure that solutions are in line with both existing needs as well as anticipated ones.

In addition to high-level architecture design documents, enterprise architects work with other architects, such as application architects, to ensure that solutions meet all of the defined requirements. Enterprise architects come up with and maintain best practices for things such as designs, implementations, and policies. For organizations that have multiple software products, they will analyze them to identify areas for architectural reuse.

Enterprise architects provide guidance, mentorship, advice, and technical leadership for other architects and developers.

Solution architect

A **solution architect** converts requirements into an architecture for a solution. They work closely with business analysts and product owners to understand the requirements so that they can design a solution that will satisfy those requirements.

Solution architects select the most appropriate technologies for the problem that needs to be solved. They may work with enterprise architects, or if such a role does not exist in the organization, take on the responsibilities of an enterprise architect, to consider an organization's overall strategic goals and enterprise architecture principles when designing their solution.

The designs created by solution architects may be reused in multiple projects. It is common in an organization to reuse architectural components and to reuse patterns across architectures in different solution areas. In large organizations that have architects playing different roles, solution architects bridge a gap between enterprise architects and application architects.

Application architect

Application architects focus on one or more applications and their architecture. They ensure that the requirements for their application are satisfied by the design of that application. They may serve as a liaison between the technical and non-technical staff working on an application.

Most of the time, application architects are involved in all the steps in the software development process. They may recommend solutions or technologies for an application, and evaluate alternative approaches to problems. Individuals in this role need to keep up with trends and new technologies. They know when to use them in order to solve a problem or take advantage of an opportunity. When appropriate, they are involved with how applications within an organization will work and integrate with each other.

Application architects ensure that the development team is following best practices and standards during implementation. They provide guidance and leadership for team members. They may be involved in reviewing designs and code. Application architects work with enterprise architects to ensure that the solutions designed for an individual application align with the overall strategy of the organization.

Data architect/information architect

Data architects are responsible for designing, deploying, and managing an organization's data architecture. They focus on data management systems, and their goal is to ensure that the appropriate consumers of an organization's data have access to the data in the right place at the right time.

Data architects are responsible for all of an organization's data sources, both internal and external. They ensure that an organization's strategic data requirements are met. They create designs and models and decide how data will be stored, consumed, and integrated into the organization's various software systems. Data architects also ensure the security of the organization's data, and define processes for data backup, data archiving, and database recovery.

They maintain database performance by monitoring environments and may be tasked with identifying and resolving various issues, including problems in production environments. Data architects may support developers by assisting with their database design and coding work.

Some organizations have the role of an **information architect**. Although the data architect and information architect roles are related, and may even be fulfilled by the same person, there is a difference between the two roles.

While data architects focus their attention on databases and data structures, information architects place their focus on users. They are concerned with user intent related to data and how data affects the user experience. They are primarily interested in how the data is going to be used and what is going to be done with it.

Information architects want to provide a positive user experience and ensure that users can easily interact with the data. They want to design solutions so that users have the ability to intuitively find the information that they need. They may conduct usability testing to gather feedback so that they can determine what changes, if any, should be made to a system. They work with UX designers and others to develop strategies that will improve the user experience.

Infrastructure architect

Infrastructure architects focus on the design and implementation of an organization's enterprise infrastructure. This type of architect is responsible for the infrastructure environment meeting the organization's business goals, and provide hardware, networking, operating system, and software solutions to satisfy them.

The infrastructure must support the business processes and software applications of the organization. These architects are involved with infrastructure components such as the following:

- **Servers**: Physical or virtual servers for either cloud or on-premises environments
- **Network elements**: Elements such as routers, switches, firewalls, cabling, and load balancers
- **Storage systems**: Data storage systems such as **storage area networks** (**SAN**) and **network-attached storage** (**NAS**)
- **Facilities**: The physical location of the infrastructure equipment, and ensuring power, cooling, and security needs are met

Infrastructure architects support the delivery of an enterprise's software applications. This includes designing and implementing infrastructure solutions and integrating new software systems with an existing or new infrastructure. Once in production, they also ensure that existing software systems continue to fulfill requirements affected by infrastructure, and run at optimal levels. Infrastructure architects may make recommendations, such as using new technologies or hardware, which will improve an organization's infrastructure.

To fulfill the demands of the enterprise, they monitor and analyze characteristics such as workload, throughput, latency, capacity, and redundancy so that a proper balance is achieved and desired performance levels are met. They use infrastructure management tools and services to assist them with the management of the infrastructure.

Information security architect

A **security architect** is responsible for an organization's computer and network security. They build, oversee, and maintain an organization's security implementations. Security architects must have a full understanding of an organization's systems and infrastructure so that they can design secure systems.

Security architects conduct security assessments and vulnerability testing to identify and evaluate potential threats. Security architects should be familiar with security standards, best practices, and techniques that can be used to combat any identified threats. They recognize security gaps in existing and proposed software architectures and recommend solutions to close those gaps.

Once security components are put into place, security architects are involved in testing them to ensure that they work as expected. When a security incident does occur, security architects are involved in their resolution and conduct a post-incident analysis. The results of the analysis are used to take proper action so that a similar incident will not occur again.

A security architect may oversee an organization's security awareness program and help to implement an organization's corporate security policies and procedures.

Cloud architect

Now that cloud deployments are the norm, having someone in an organization dedicated to cloud adoption, with the relevant expertise, has become increasingly common and necessary. A **cloud architect** is someone who is responsible for an organization's cloud computing strategy and initiatives. They are responsible for the cloud architecture used for the deployment of software systems. An organization that has someone who is focused on cloud architecture leads to increased levels of success with cloud adoption.

The responsibilities of cloud architects include selecting a cloud provider and selecting the model (for example, SaaS, PaaS, or IaaS) that is most appropriate for the organization's needs. They create cloud migration plans for existing applications not already in the cloud, including the coordination of the adoption process. They may also be involved in designing new cloud-native applications that are built from the ground up for the cloud.

Cloud architects oversee cloud management, and create policies and procedures for governance. They use tools and services to monitor and manage cloud deployments. The expertise that cloud architects possess typically means they are involved in negotiating contracts with cloud service providers and ensuring that **service-level agreements** (**SLAs**) are satisfied.

Cloud architects should have a firm understanding of security concerns, such as protecting data deployed into different types of cloud and cloud/hybrid systems. They work with security architects, or if such a role does not exist in the organization, take on the responsibilities of a security architect, to ensure that systems deployed to the cloud are secure.

For organizations that have not fully migrated to the cloud, one of the tasks for a cloud architect is to lead a cultural change within the organization for cloud adoption. Cloud strategies can fail if the organization's culture does not fully embrace them. Part of the cloud architect's job is to evangelize cloud adoption by communicating the many benefits, and influence behavior changes toward cloud adoption that will ultimately lead to cultural changes.

Software development methodologies

As a software architect working in an organization, you will typically be required to use the software development methodology that has been adopted by the organization. However, in some cases, the software architect may play a role in deciding which software development methodology is used. Either way, software architects may be able to provide input to an organization's processes, giving them the ability to make suggestions that may improve these processes.

For these reasons, it is good to have an understanding of the more common software development methodologies. There are a variety of different types that can be employed for a software project, each of which has its own strengths and weaknesses. Today, Agile methodologies are much more widely used than traditional ones, but even among the methodologies that are considered Agile, there are numerous variations.

Unfortunately, sometimes, a software project moves forward with a software development methodology that is not appropriate for the project. Prior to choosing one, you should consider which one would be the most appropriate to use. In the following sections, we'll take a look at two types of software development methodologies: the Waterfall model and Agile.