

Risk-Centric Software Architecture

George Fairbanks

RhinoResearch.com

Copyright © 2007-2009 George Fairbanks. All rights reserved. This is a pre-production draft for review only. Please do not distribute it. If you know someone who would like to be a reviewer, please email: george.fairbanks@rhinoresearch.com.

No warranty of any kind is expressed or implied. The author assumes no responsibility for errors or omissions. The author assumes no liability for incidental or consequential damages connected with or arising out of the use of the content of this book.

Errata and updates to this book are available at <http://RhinoResearch.com/book>.

This book is under contract to be published in 2010 by Taylor and Francis.

Contents

Contents	ii
Preface	vii
1 Introduction	1
1.1 Partitioning, knowledge, abstractions	2
1.2 Three examples of software architecture	3
1.3 Reflections	4
1.4 Perspective shift	6
1.5 Architects architecting architectures	7
1.6 Risk-centric architecture	8
1.7 Architecture for agile developers	9
1.8 About this book	9
I Risk-Centric Software Architecture	12
2 Software Architecture & Modeling	13
2.1 Abstraction and modeling	13
2.2 Architects architecting architectures	17
2.3 What is software architecture?	18
2.4 Why is software architecture important?	20
2.5 When is architecture important?	24
2.6 How should software architecture be used?	25
2.7 Architecture-indifferent design	25
2.8 Architecture-centric design	26
2.9 Architecture hoisting	27
2.10 Architecture in large organizations	29
2.11 Conclusion	30

2.12	Further reading	31
3	Risk-Centric Model	33
3.1	What is the risk-centric model?	34
3.2	Are you risk-centric now?	35
3.3	Risks	36
3.4	Techniques	38
3.5	Guidance on choosing techniques	41
3.6	When to stop	43
3.7	Planned and evolutionary design	43
3.8	Risk and process	46
3.9	Sketch: Applying the risk-centric model to agile processes	50
3.10	Alternatives to the risk-centric model	51
3.11	Conclusion	52
3.12	Further reading	53
4	Example: Home Media Player	56
4.1	Team communication	57
4.2	Integration of COTS components	65
4.3	Music metadata format	71
4.4	Conclusion	75
5	* Modeling Advice	77
5.1	Understand your architecture	77
5.2	Use models to solve problems	78
5.3	Make rational architecture choices	79
5.4	Choosing the abstraction level	80
5.5	Avoid Big Design Up Front	81
5.6	Focus on risks	82
5.7	Avoid top-down design	83
5.8	Distribute architecture skills	83
5.9	Remaining challenges	84
5.10	Solve problems, not just model them	87
II	Architecture Modeling	88
6	** A Mental Model of Software Architecture	89
6.1	* Canonical model structure	91
6.2	** Inter-model relationships (overview)	93
6.3	Views of a master model	93
6.4	Other ways to organize models	94
6.5	Conclusion	96
6.6	Further reading	96

7	* The Domain Model	99
7.1	Concept model	100
7.2	Navigation and invariants	101
7.3	Snapshots	102
7.4	Functionality scenarios	103
7.5	* Business model	104
7.6	Understanding the domain	104
7.7	* Further reading	105
8	* The Design Model	106
8.1	Design model	106
8.2	Boundary model	107
8.3	Internals model	116
8.4	Viewtypes	121
8.5	* Dynamic architecture models	125
8.6	Architecture description languages	125
8.7	* Conclusion	126
8.8	* Further reading	127
9	The Code Model	128
9.1	Model-code gap	128
9.2	Managing model - code consistency	131
9.3	Architecturally-evident coding style	133
9.4	Modeling existing systems	150
9.5	Summary	151
9.6	* Further reading	152
10	Encapsulation and Partitioning	153
10.1	Story at many levels	153
10.2	Hierarchy and partitioning	155
10.3	Partitioning strategies	157
10.4	Effective encapsulation	160
10.5	Building an encapsulated interface	163
10.6	Conclusion	166
10.7	Further reading	167
11	* Design Model Elements	169
11.1	Allocation elements	169
11.2	* Components	171
11.3	Connectors	179
11.4	Design decisions	189
11.5	Functionality scenarios	190
11.6	Invariants (constraints)	197

11.7	Modules	198
11.8	* Patterns and styles	199
11.9	Ports	209
11.10	Quality attributes	216
11.11	Quality attribute scenarios	217
11.12	Responsibilities	219
11.13	Tradeoffs	221
11.14	Conclusion	221
12	Inter-model relationships	222
12.1	Projection (view)	222
12.2	Partition	227
12.3	Composition	227
12.4	Classification	227
12.5	Generalization	228
12.6	Designation	229
12.7	Refinement	230
12.8	Binding	232
12.9	Dependency	233
12.10	Using the relationships	233
12.11	Conclusion	234
12.12	Further reading	235
13	* How to Use the Models	236
13.1	** Analysis	236
13.2	Desirable model traits	239
13.3	Achieving model quality	244
13.4	Working with views (consistency, specialization)	247
13.5	Architectural mismatch	249
13.6	* Effective diagrams	250
13.7	* Plan for the user interface	251
13.8	* Prescriptive vs descriptive abstractions	252
13.9	* Conclusion	252
13.10	Further reading	253
14	* Conclusion	254
14.1	* Challenges	255
14.2	Use standard architectural abstractions	259
14.3	Focus on quality attributes	260
14.4	Conclusion: Knowledge, partitioning, and abstraction	261
	Definitions	262
	Bibliography	263

Contents

vi

Index

272

Preface

Software systems are larger and more complex than ever, and helpful software architecture techniques have been developed, yet software architecture is shunned by many software developers. How did this happen? I believe there are two primary reasons. First, software architecture is associated with big, high-bureaucracy projects while most developers are seeking lighter-weight software development approaches. Second, software architecture is associated with the annoying guy in the corner office, who has the title of *architect*, draws pictures, never codes, and is a general drag on the project.

This book seeks to change these perceptions and to democratize software architecture techniques for the benefit of all software developers. This book describes a way for you to identify engineering risks and combat them with deliberately chosen engineering techniques. It avoids the “one size fits all” tarpit with advice on how to identify the risks in building your system. Any particular technique mitigates some risks but not others, so you must learn which techniques mitigate which risks, or else you will waste your time on the wrong techniques. Most techniques have a range of rigor from quick-and-dirty to meticulous. There is no need for meticulous designs when risks are small, nor any excuse for sloppy designs when risks threaten your success.

This is the book I wish I had when I started developing software. At the time, there were books on languages, and books on object-oriented programming, but few books on design. Knowing the features of the C++ language does not mean you can design a good object-oriented system, nor does knowing the Unified Modeling Language (UML) imply you can design a good system architecture.

There is a difference between being able to hit a tennis ball and knowing why you are able to hit it, what psychologists refer to as *procedural knowledge* and *declarative knowledge*. Some readers of this book will already be proficient at designing and building systems and will have employed many of the techniques found here. Reading this book should make them more aware of what they have been doing and provide names for the concepts. By augmenting their procedural knowledge with declarative knowledge they will improve their ability to mentor younger developers.

There is a gap between where university training ends and what current software de-

velopment practice demands. New developers spend years learning how to work with complexity and scale. They learn mostly through inference and often invent idiosyncratic notations and concepts. For example, every developer has some informal concept of “big chunk of code” but may not distinguish between that code at run-time versus compile-time, may only consider the interfaces the code provides and not what interfaces it requires of others, and may only reason about simple connectors like method calls. This is not a criticism of any developer’s skills but instead reflects how new the field of software engineering is and how we are still developing our standards. This book primarily uses UML notation because it is prevalent in the field, and it is important that you know how to express your ideas using a standard language so that others can interpret the brilliance of your squiggles.

We too often lump together two distinct ideas when we discuss software process. The first idea involves the management of a team of engineers so that they produce a system. This kind of process deals with schedules, resource commitments, and stakeholder needs. If you are building a bridge, you need to arrange financing, get permits, negotiate with government, and hire engineers — these things deal with project management. The second idea of software process is technical, and deals with what the engineers do to ensure the system works. Bridge engineers, for example, study existing bridge styles, use models to calculate stresses and strains, and build prototypes.

Quite a few books cover the management process idea for software architecture. Other books have expanded our understanding of the language and concepts needed to describe systems at large scale. Too few books, however, help with the second idea: the engineering tasks themselves. To help fill this gap, this book focuses its attention on building and analyzing architecture and design models. It describes the basic techniques used to reason about medium to large sized problems and provides pointers to where to learn specialized techniques in more detail.

Compared to other books that draw sharp distinctions between architecture and design, this book embraces both as part of a gradient. In my experience across companies of various sizes and domains, I have never seen a successful compartmentalization of architecture separate from design. Design decisions influence the architecture and vice versa. Engineers responsible for the system drill into an obstacle in detail, understand it, then pop-up to relate the nature of that obstacle to the architecture as a whole. The approach in this book embraces this drill-down/pop-up behavior by relating models at varying levels of abstraction, from architecture to data structure design.

About me

I have supported agile techniques since their early days: In 1996 I successfully encouraged my department to switch to from a 6-month to a two-week development cycle, and in 1998 I started doing test-first development. I have been a software developer on projects including the Nortel DMS-100 central office telephone switch, statistical analysis for a driving simulator, an IT application at Time Warner Telecommunications, plug-ins for the Eclipse IDE to support static analysis, and code covering soup to nuts for my own web startup

company. Like many of you, I tinker with Linux boxes as an amateur system administrator, and have a closet lit by blinking lights and warmed by power supplies.

I was lucky to meet Desmond D’Souza early in my career and later worked at his company with the group of bright folks he collected. I have taught object-oriented design classes and software architecture classes to industrial clients since the late 1990’s, often as part of a larger consulting arrangement. This has allowed me to dig into additional domains, including banking, finance, and enterprise content management. Working with Desmond and his group, I realized that careful engineering has its place, and that architecture and design are compatible with agile software development.

My career has been a quest to learn how to build software systems. That quest has led me to interleave academic study with industrial software development. Over the years I have gotten a bachelors and masters degree in computer science from the University of Virginia and the University of Colorado at Boulder, and a Ph.D. in software engineering from Carnegie Mellon University, but as a nontraditional student I worked for years in industry between each degree. Software frameworks were increasingly common and complex, yet programmers had difficulty learning them and gaining full confidence in their plug-ins, so for my dissertation I developed a new kind of specification, called a design fragment, to describe how to use a framework, and built an Eclipse-based tool that could validate correct use of the framework. I was enormously fortunate to be advised by David Garlan and Bill Scherlis, and to have Jonathan Aldrich and Ralph Johnson on my committee.

During my academic tour, I realized that one of my “secret weapons” compared to most others was my industrial experience. Being in the trenches and struggling to build systems had given me an intuition as to what the hard problems were and what kinds of techniques might help. I later realized that few practitioners had spent so long in academics, and few academics had spent so long in industry. This book has its roots in both fields.

Audience

Readers should already know basic software development ideas, things like object-oriented software development, the Unified Modeling Language (UML), use cases, and design patterns. Some experience with how real software development proceeds is exceedingly helpful, as most of this book’s basic arguments are predicated on common experiences. If you have seen a project build too much documentation, or do too little thinking before coding, you will know how software development can go wrong, and therefore be looking for remedies like those offered in this book.

Too often I see developers from one field criticizing the engineering techniques in another field, or worse, simply assuming that their standard engineering techniques will work elsewhere. This book identifies the risks in various domains and explains how you should use different techniques in different fields. Consequently, experienced developers will likely be familiar with some of the ideas in this book, though perhaps only the ones from their field.

This book is also suitable as a textbook in an advanced undergraduate or graduate level

course. Students should have had experience with software development, at least on internships, and ideally on team projects, or else they may look on the techniques here as yet more bureaucracy without purpose.