SAMPLE

Building Tomorrow's Software Engineers

The Contract's Keeper in the Machine Era

Aiman JoeHanz

Free sample: Introduction & Chapter 1

putrabooks.github.io

Buy the full book:

Carousell carousell.sg/p/building-tomorrow-s-software-engineers-ebook-pdf-1471387165

Ko-fi

Payhip

Contents

Introduction

Chapter 1 — The Contract

Chapter 2 — The Two Folklores

Chapter 3 — The Way In

Chapter 4 — The Working Day

Chapter 5 — The Craft

Chapter 6 — The Expiry

Chapter 7 — The Conversation

Chapter 8 — The Instruments

Chapter 9 — The Year

Chapter 10 — The Long Arc

Introduction

A piece of software is a collection of promises, and most of the trade's trouble begins when a promise is kept in somebody's head instead of in the work.

This book describes how that work is actually done: the daily practice of the people who author those promises and keep them, the territory the work covers, the rewards the career actually pays, and the differences the machine era is already making to a trade whose ordinary code can now be drafted for it. It is aimed at people weighing the role, at practitioners of neighbouring trades moving across into it, and at the managers whose budgets pay for systems that strangers not yet hired will one day maintain.

The claim is narrow. Software engineering is not the production of code, and it is not the pursuit of the elegant solution. It is the maintenance of contracts: the stated promises that let one part of a system depend on another without reading it. A function's signature is a contract. An endpoint's behaviour is a contract. A queue's ordering, a timeout's meaning, an error's shape, a field's nullability: contracts, all of them, and the system is exactly as trustworthy as the smallest of them. The trade's real work is deciding what to promise, writing the promise where a reader can find it, and keeping it when the pressure arrives.

Two folklores get in the way of seeing it. The first is the wizard, for which an engineer's value is the code only they can understand. It is attractive because it feels like mastery and it pays, briefly, like scarcity. It fails because an organisation that cannot read its own system cannot change it, and the wizard's value evaporates at the first holiday. The second is the feature factory, for which software is a throughput business: tickets in, features out, velocity on the wall. It is attractive because it is measurable. It fails because it optimises the production of code nobody promised to maintain.

The chapters proceed as the collection proceeds: first the contract itself, then the folklores that misdescribe the trade, the routes by which people enter it, the shape of its ordinary day, the craft at the bench, the strain, the conversations that surround the code, the instruments that keep it honest, one practitioner's composite year, and the arc of a career.

One caution belongs at the front. This is the trade the machine era reached first, and its practitioners are the first generation to work above a tool that drafts their ordinary output for them. The strain the trade carries is accordingly not workload, which is every trade's burden, but expiry: the

knowledge beneath a career loses validity while the career is being lived, and the machine has shortened the half-life. The chapter on that strain answers with a distinction, not a syllabus: between what this trade rents and what it owns.

The book's method is the collection's method, and the reader who has travelled this far with the series will recognise its furniture. Each volume takes one occupation and finds the single idea that, held honestly, organises the whole of its practice; each then examines that idea from the trade's four fixed angles, the way in, the day, the strain and the arc, and closes on one constructed year in one constructed working life, assembled from the ordinary weather of the real one. The composite is announced as composite, the claims are kept within what the trade's public record can support, and where the record is thin the book says so rather than filling the gap with invented statistics, which are the industry's favourite genre and this collection's standing refusal.

A word about the trade this volume describes, because its title names the trade's largest profession and its chapters treat it at the collection's usual altitude. Software engineering, in this book, is not merely the occupation of the people who build the visible applications, and the chapters' examples are drawn as deliberately from the quiet systems as the loud ones: the order service and the reporting warehouse, the payment path and the log line, the seams where one team's promise meets another team's schedule. The trade's territory is wherever behaviour is produced by code that somebody will one day read, and the volume's claim is that the territory is recognisably one place, held together by the single practice the first chapter names.

A note on tone is owed, because the machine era's noise has changed how books like this are read. This is not a book of forecasts, and it does not argue that the trade is ending, ascending, or entering any other weather condition. It is a description of a craft under pressure, written for practitioners who will spend their working lives inside that pressure, and its sentences are meant to be checked against the reader's own experience rather than believed. The reader who finds a chapter describing a room they have worked in has read the book correctly. The reader who finds a chapter describing a room they have never seen should treat it as a map of terrain they may cross, and the reader who finds no chapter describing their own trade's trouble should write that one themselves, because the trade's literature is the estate the last section described, and it is always short of its next holder.

Nothing here assumes a particular language, framework, vendor or company. The mechanisms described hold wherever behaviour is produced by code that somebody else will one day read, and where the mechanisms differ across languages and eras, the differences are named as they appear.

Chapter 1 — The Contract

What a Contract Is

A contract is a promise with a producer and a consumer, stated somewhere both can read it. In this trade it takes unglamorous forms: a function's signature, an endpoint's documented behaviour, a queue's delivery guarantee, a field's meaning, an error's shape, the pause a timeout implies. What makes any of them a contract is not the form but the asymmetry underneath: the consumer builds on the promise without reading the implementation, so the promise had better be true.

The whole of the craft follows from that asymmetry. If consumers had to read everything they depend on, software could not be built at the scale it is built. A modern service rests on thousands of promises made by strangers, most of them dead, some of them wrong, all of them load-bearing somewhere. The contract is the mechanism that lets a stranger use a component in confidence, and the trade's practitioners are, above everything else, the people who make that confidence justified.

It is worth pausing on how strange this is as a way of working. A carpenter can inspect a joint. A mason can see a wall. A software engineer ships a promise into a system whose total behaviour nobody has ever observed, and the promise's truth will be tested, over years, by callers the producer will never meet, in configurations that did not exist when the promise was made. The trade's dignity and its anxiety come from the same source: the work's consequences are larger than its author's view.

Where the Promises Live

The promises live in more places than the obvious ones. The signature is the visible promise: what is accepted, what is returned, what the types permit. Around it sit the quieter ones. The name of a thing is a promise about what it does. A default value is a promise about the empty case. An ordering is a promise about sequence, and so is the absence of one. A timeout is a promise about how long the consumer will wait, and its silent change is the classic broken contract: nothing fails in the change, and everything fails after it.

Beyond the code sit the promises written in the materials that surround it. The migration script's idempotency is a promise to whoever runs it twice. The configuration schema is a promise about what the system believes the world looks like. The example in the documentation is a promise that behaviour will match it, and documentation that has drifted from the code is a contract

forged rather than broken, because the consumer trusts the more readable of the two. Even the deployment script's order of operations is a promise, and one that has quietly decided more outages than most algorithms.

Then there are the promises the system makes outside itself, which are the ones that end careers. The page that says the item was saved is a promise to the person who believed it. The email that says the booking was cancelled is a promise with legal weight in some jurisdictions. The invoice line is a promise with a price. The trade sometimes talks as though its obligations end at the process boundary, and the ordinary user's experience disagrees, at length, in writing, on the company's public review pages.

The Unstated Promise: A Worked Case

The cheapest way to see the contract is to watch one break. Consider a composite case assembled from the trade's ordinary materials. A service returns a list of recommendations. Its first consumer, the website, shows the first three and ignores the rest. A second consumer, built eighteen months later for the mobile application, notices that the field is occasionally empty and writes code that handles emptiness by showing the most popular items instead.

Neither team has made a promise about emptiness. The website does not know the field can be empty because it has never seen it empty. The mobile team knows it can be empty but not why, or how often, or whether the emptiness means nothing to show, a failure to fetch, or a deliberate suppression. The service's authors, meanwhile, have no idea either consumer exists, because the component was shared by a link in a chat thread two reorganisations ago.

The system now contains three unstated positions on one question, and the fourth position belongs to the future: an engineer on the service, unaware of all of this, decides that emptiness is an error condition and starts returning a failure status in that case. The mobile application's careful fallback now silently masks a real failure. The website starts showing error text nobody designed. A support ticket arrives, is routed to the wrong team twice, and is eventually resolved by a revert whose commit message says only that it restored previous behaviour, which is true and useless.

Nothing in this story involves bad code. Every engineer acted reasonably on the knowledge available to them, which is the unsettling part: the failure was manufactured entirely by unstated promises. The repair is equally ordinary,

and it is the repair this chapter is arguing for. Somebody writes the promise down: emptiness means this, occurs under these conditions, is stable as a behaviour, and changes through this channel. One paragraph, placed where consumers look, retires the entire class of failure, and the trade's seniors know that most of their hardest afternoons were caused by the paragraph nobody wrote.

Three Properties

A reliable contract has three properties. It is stated, in a place the consumer can find without asking the producer. It is small, promising less than the implementation could do, because the wider the promise the harder the keeping. And it is stable under revision, meaning that when it must change, the change is announced, versioned, and priced, rather than discovered.

Each property has a failure twin. The unstated contract breaks the consumer who read the code and the consumer who did not alike, in different directions: the reader over-fits to today's implementation and shatters on an internal refactor; the non-reader assumes a generality that never existed and shatters on the first edge case. The oversized contract is kept honestly for a year and then becomes the reason the component cannot be improved without breaking the world, and its authors learn that every capability exposed is a capability owed. The silently revised contract is the costliest artefact a software organisation can contain, because its breaking is discovered at the moment of use, by a stranger, in production, when the producer is on holiday.

The three properties also explain the trade's most misunderstood habit: refusing to build the feature that is clearly easy. The senior who declines to expose a capability that the implementation already supports is not being lazy or territorial. They are pricing a promise, because exposure is not a one-time act but a maintenance obligation measured in decades of other people's build decisions, and the cheapest promise to keep is the one declined on the day it was cheap to decline.

Who May Change What

A promise without an owner is a promise with a countdown. The mature practice therefore attaches to each contract a small apparatus of governance, and none of it is bureaucracy, whatever the folklore says. The version number is a promise about change. The deprecation notice is a promise that the change will be announced before it arrives, and the sunset date is a promise that the announcement was serious. The changelog is the contract's public

record, and the practice of writing it before the code is written, stating the promise the change will make, is one of the trade's quiet competitive advantages.

Ownership here means answerability rather than property. The question who may change this promise has a legitimate answer in every healthy team: a person or a named group, reachable through a channel, who will answer a question about the promise within a day. The teams that struggle are those where the honest answer is whoever has the credentials, and the consumer who needs to ask a question has no way to ask it that does not involve becoming a nuisance.

The Contract Across Scales

The contract scales. At the smallest scale it is a function's signature, and the discipline is the same at every scale above it: state the promise, keep it, announce the changes. At the service scale the contract is the interface the network carries, and the consumers are other teams with their own deadlines. At the organisation scale the contract is the platform's promise to its internal customers, and the consumers are every product the company ships. At the public scale the contract is the interface strangers build businesses on, and the discipline finally becomes what it always was, visible in the way the great public interfaces are revised: slowly, loudly, and with years of notice.

The trade's seniority is largely the ability to work at larger scales without losing the small one. The engineer who can hold a function's signature and an organisation's platform promise in the same argument is the one the room defers to, because most practitioners can do one or the other: the ones who over-index on the small scale become purists whose elegance is irrelevant to the roadmap, and the ones who over-index on the large scale become slidemakers whose platforms are made of promises nobody below them can keep.

The Machine Era's First Effect

Generated code raises the volume of code and lowers the cost of writing it, and both forces press on the contract from opposite sides. More code means more promises, made faster, in more places, by people with less context on each one. Cheaper code means the value of any individual implementation falls, and the value of the promise it upholds rises, because the promise is the part that cannot be regenerated. A component can be rewritten in an afternoon. The trust of every consumer of it cannot.

The practitioner who works above a generating tool is therefore not doing less engineering. They are doing the part of engineering that was always load-bearing: deciding what to promise, to whom, under which conditions, and standing behind the answer. The tool drafts the implementation. The contract is still authored by a person, and the failures of the next decade will belong, overwhelmingly, to the authored promises.

There is a specific mechanism worth naming, because it will be the era's characteristic incident. A generated implementation does not merely satisfy a stated contract; it implies a hundred unstated ones, in its defaults, its ordering, its error shapes and its rounding choices, all of them plausible, none of them read. The consumer builds on the implied promises exactly as the worked case's consumers did, and the first revision of the generated component, also generated, changes them without ceremony. The discipline that prevents this is the chapter's whole argument applied to machine output: state the promises the implementation must make, review the draft for promises it made that nobody asked for, and refuse to ship the ones that were merely convenient.

The Promise and the Person

Because a contract is a social object, it has a psychology, and the trade is honest about it earlier than most. An engineer who states a promise in public, with their name attached, has done something that changes their own behaviour: the stated promise is harder to abandon than the private intention, and the shame of breaking it quietly is a genuine design material. The trade's older cultures knew this and used it, in code review rituals and in the sign-off lines of interface documents, and the modern equivalents, the pull request and the changelog, are the same technology with friendlier clothes.

The same psychology explains the folklore of the bus factor, which the industry misdiagnoses as a knowledge problem. The team that collapses when one member is hit by a bus was not carrying too little documentation. It was carrying too few promises. Knowledge in a person's head is unstated by definition, and the organisation's exposure is not the size of the head but the count of the promises it alone holds. The repair is not a wiki drive, which decays in months. It is the conversion of each private promise into a stated one, with an owner, a place and a revision channel, which is precisely why the repair feels like more work than a wiki drive: it is.

Reading as Contract Work

Half of the trade's working time is spent reading, and the folklore treats this as a cost of writing. It is closer to the opposite: reading is the trade's method of receiving promises, and the reader's skill determines the resolution at which the system can be seen. The junior reads a file. The mid-career engineer reads a component and its neighbours. The senior reads the seams, and can enter an unfamiliar system and return, within an hour, with the list of promises it makes that most of its maintainers could not name, which is the list that matters when something must change.

The craft of reading is therefore worth describing rather than assuming. It is the habit of asking, of every block of code, what does this promise, to whom, and what happens if it is false. It treats names as claims to be tested, defaults as decisions to be found, and error paths as the system's confession. And it writes down what it finds, because the reader's notes are the raw material of the contract's repair, and the reader who says nothing has merely been a tourist in the failure's future.

The Contract and the Clock

Some of the trade's most consequential promises are about time, and they need a paragraph of their own, because prose handles them worst. A latency figure is a promise about patience. An availability figure is a promise about absence, which is a strange thing to promise and can only be promised as a proportion, never as an absolute. A durability figure is a promise about survival, and the honesty of its measurement depends on the honest answer to survival against what. These numbers are contracts even when nobody writes a paragraph around them, because consumers build budgets on them, and budgets, once built, are promises too.

The mature form of this is the service level with an objective, and the craft point is not the arithmetic but the authorship. An objective is a promise the team writes about itself, and the team that writes it honestly, publishes it, and treats its own breaches as contract events, has converted the foggiest part of the system into an auditable promise. The teams that resent the practice are usually not resenting the measurement. They are resenting the first occasion on which their private optimism was required to become a public commitment.

When Contracts Meet

The final complication is that promises compose, and composition is where honest contracts produce dishonest systems. Two components may each keep every promise made about them and still compose into a failure: the retry

policy that is correct for each service becomes a storm when both policies are held at once, and the timeout that is generous at each layer becomes a denial of service when stacked three deep. No single promise is broken, and no single author is at fault, which is why composition failures are the trade's hardest postmortems and its best arguments for senior judgement.

The repair is not a cleverer component. It is a contract at the layer above, where the composition itself is treated as a component with its own promises, its own owner, and its own review. This is the quiet definition of what the industry calls architecture, stripped of its glamour: architecture is the practice of making promises about the space between components, which is the space where, unmanaged, the failures of the next decade are already queued.

Contracts and Money

A promise is also a financial object, and the trade is more useful to its organisations when it says so plainly. Every contract a system carries has a carrying cost, paid in review time, test time and the quiet narrowing of what the system can become, and every contract has a breach cost, paid by whichever party discovers the difference between what was believed and what was true. The engineering argument for stating promises is ultimately an argument about who pays the breach cost: the organisation that states its contracts pays it in planning, at a discount, and the organisation that does not pays it in production, with interest, in public.

The same arithmetic explains the trade's most reliable negotiating instrument. When a product manager asks for a change, the engineer who can name the promises the change would break, the consumers that depend on them, and the price of their revision, has converted a disagreement into a quotation. The room that argues about effort is arguing about feelings. The room that argues about contracts is arguing about the system, and the second room, whatever it decides, decides with the cost visible.

The Promise Audit

Because promises decay, they need an audit, and the trade's mature teams run one without calling it that. The audit is a periodic reading of the system's stated contracts with three questions in hand. Which promises are still true. Which are false but unbroken, the most dangerous class, because their falsity is undiscovered capital. And which are false and known, the debts, each of which needs an owner and a date or an honest retirement.

The audit's findings are usually humbling and always cheap relative to the

alternative, which is the incident that performs the audit without consent. The practitioner who leads one will typically find that a fifth of the documented promises describe a system that no longer exists, that a handful of undocumented ones have quietly become the company's critical path, and that the two or three sentences most worth writing were, once again, the ones nobody wrote. The audit is the chapter's three habits, notice, state, announce, applied in a loop, and the organisations that run the loop find that their systems age like instruments rather than like terrain: maintained, tuned, and played, rather than eroded and abandoned.

The Worked Contract Page

Because this chapter has argued for a paragraph nobody wrote, it owes the reader the paragraph's shape. The contract page, in its mature form, is four short blocks. The promise: what the component does, in one paragraph, in the consumer's vocabulary. The refusals: what it does not do, stated as happily as the promises, because the refusals are what the consumer builds against in the dark. The conditions: ordering, timing, volume, the empty case and the failure case, each named with its meaning rather than its default. And the revision channel: who owns the promise, where its changes are announced, and how far in advance. The whole page is shorter than the emails its absence generates, which is the argument of this chapter reduced to a length, and the practitioner who writes one this week, for the component they hold today, has performed the entire first chapter of this book in an afternoon.

The Promise's Reader, Named

The chapter's last refinement is a change of focus the whole book will inherit: the promise's reader is not an abstraction, and the craft improves the moment the practitioner can name the reader of each promise they hold. The signature's reader is the next engineer to call the function, in a year, with a broken assumption. The error message's reader is the colleague at three in the morning, described in the day chapter. The commit message's reader is the debugger following a regression's ancestry. The contract page's reader is the successor, who is the author's only real immortality, and the documentation's reader is the organisation's next decision, which will be made by someone who has never met the author and will trust the page exactly as far as the page's honesty has earned.

This naming is not a rhetorical device; it is the trade's working method of design under uncertainty, because the named reader converts vague quality into specific obligations. The practitioner who cannot name a promise's reader

has found, in the same moment, both the promise's weakness and the reason it was hard to write. The practitioner who can name the reader, and their state of mind, and what they will be trying to accomplish when they arrive, has the paragraph's outline already drafted, and the chapter's closing observation is that this is the craft's smallest and most transferable act: to write, deliberately, for the one person who will actually have to live inside the promise, at the hour of its breaking.

What This Chapter Asks

The rest of the book assumes the contract as its centre of gravity, and asks three things of the reader in return. Notice promises, including the ones nobody wrote down. Write the ones you make, smaller than you could. And when one must change, treat the change as an event with witnesses, not a private edit.

A reader who holds those three habits holds the trade's core, and the chapters that follow are mostly the application of them to the places software engineers actually spend their lives: the territories, the day, the conversations, the instruments, the year, and the arc. The application is the substance, because knowing that promises matter is the work of an afternoon, and keeping them for a decade is the profession.

ABOUT THE AUTHOR

The Building Tomorrow's People series gives one role per volume the same treatment: the honest definition, the failure modes, the instruments, and the discipline a working professional can run. Aiman JoeHanz wrote this installment for software engineers, whose trade he defines by the contracts it keeps rather than the languages it uses, where a contract is an interface, a promise about failure, a review that means something, and a piece of code the next person can change without fear. He is less interested in what a system does on a good day than in what its authors agreed it would do on a bad one.