We hope you enjoy this sample PDF of our special 20th anniversary book. The full book is 78 pages and features a curated collection of 20 articles, a visual timeline of important milestones, and an introduction from thoughtbot Founder and CEO, Chad Pytel.

Learn more and purchase the full version at https://thoughtbot.com/resources/20-for-20.

thoughtbot

Sample

Table of Contents

Intro ii

Timeline 02

Blog Posts 06

TESTING AND GOOD CODE PRACTICES

Let's Not - Joe Ferris 06
The Case for WET Tests - Amanda Beiner 12
Testing from the Outside-In - Joël Quenneville 17
Write Readable Code - Edward Loveall 20
Avoid the Three-state Boolean Problem - Gabe Berke-Williams 23

DEI AND TEAMWORK

Pay the cost up front. Don't distribute it to others. - German Velasco 25
DEI: Speaking Up About Our Values - Christopher Kuttruff, Mina Slater & Sally Hall 28
Working in my native language requires empathy - Matheus Richard 30
Private Messages are not Inclusive - Stephen Lindberg 32
Pairing is Caring - Valeria Graffeo 34

DESIGN

Start with the problem - Christian Reuter 36
Best practices for designing apps people actually use - Devin Jameson 38
Tailwind and the Femininity of CSS - Elaina Natario 42
It's Only Color - Mike Borsare 44

PROCESS

Working Iteratively - Joël Quenneville 52
How to Estimate Feature Development Time: Maybe Don't - Amanda Beiner 57
Investment Time - Ian Zabel and Dan Croak 61
Moving Beyond, Not Getting Over, Imposter Syndrome - Stephanie Minn 65
Choosing processes that build trust - Amanda Beiner 67
Say no to more process, say yes to trust - German Velasco 71

Credits 74

Introduction

by Chad Pytel

Founder & CEO

I started thoughtbot in 2003 with four friends from university. We didn't have a grand vision for what we hoped to achieve. We mostly wanted to keep working together, creating cool things.

It wasn't until after two fairly mediocre years that thoughtbot truly became thoughtbot. That was when three of the original founders decided to go and get regular jobs.

Being faced with the choice of sticking with it, or giving up - and then choosing to stick with it, put us in a completely different mindset.

We realized that it wasn't worth continuing if we weren't happy and fulfilled in our work.

So we started to be much more clear about what we believed about the way products should be designed and built, and how companies should be run.

That was September of 2005, and on November 6, 2005, the first two articles were posted to our blog, Giant Robots Smashing into other Giant Robots: The Correct Amount of Planning and Try Ruby!.

Since that time, as we've continued to try to find better ways of working, our blog has been an important part of sharing what we learn along the way.

We're excited to share this curated list of 20 blog posts in celebration of our 20th anniversary, along with a visual timeline of important milestones.

To compile these articles we reviewed the more than 2,100 blog posts we've made in the history of the blog and narrowed it down to these 20 gems.

thoughtbot hasn't lasted this long, and accomplished what we've done solely through my efforts.

It has also been through the efforts of hundreds of team members over the years. My heartfelt thanks go out to everyone who has contributed to both thoughtbot, and our blog, over the years.

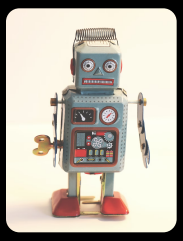
And thank *you* for reading.

Timeline

A 20-year journey

20 years is a long time, and so much has happened over our history. We couldn't have predicted where thoughtbot would be today, but it's been an exciting journey, packed with learning and experiences. Every step is an adventure and inspiration for the next 20 years to come.

Here are some of our major milestones so far.



JUNE 2, 2003

thoughtbot founded

A partnership between five WPI graduates and friends.

SEPTEMBER 9, 2005

A turning point

3 of the original founders leave and we reincorporate as thoughtbot, inc.

NOVEMBER 6, 2005

Our Blog Starts

The first two articles are posted to our blog: The Correct Amount of Planning and Try Ruby!. The blog looked something like this: 2005 blog.

Testing from the outside in

Joël Quenneville - June 9, 2014 | Updated March 6, 2019

A few months ago my colleague Josh Steiner wrote a comprehensive post on How We Test Rails Applications, detailing the different types tests we write and the various technologies that go with them. In this follow up, we will take a closer look at thoughtbot's testing workflow.

We use a process known as "Outside-in testing", driving our development from high-level tests and working our way down to lower-level concerns. Say we are working on an e-commerce site and want to implement the following story:

> *As a guest, I can add items to my shopping cart so that I can keep on shopping*

Before we start thinking about models, controllers, or other architectural concerns we write a high-level RSpec feature test that describes the behavior from the user's perspective.

```
# spec/features/guest_adds items_to_shopping_cart_spec.rb
feature 'Guest adds items to shopping cart' do
  scenario 'via search' do
    item = create(:item)

    visit root_path
    fill_in 'Search', with: item.name
    click_on 'Search Catalogue'

    click_on item.name
    click_on 'Add to Cart'
    click_on 'Shopping Cart'

    expect(page).to have_content(item.name)
    expect(page).to have_content("Subtotal: #{item.price}")
  end
end
```

Testing from the outside in

cont'd

Depending on how much of the application is implemented, this test could break in multiple places. If this were a newly-generated application we might need to implement a home page. Once we have a home page we would probably get an error while attempting to use the search bar saying that 'No such route exists'. This leads us to implement a `/items` route.

```
# config/routes.rb
# ...
resources :items, only: [:index]
# ...
```

The next few errors walk us through creating an `ItemsController`, with an empty `index` action and corresponding view. Now that we can successfully click on "Search Catalogue", we get an error saying that there the desired item does not appear in the search results so we expose some items in the controller and display them in the view.

```
# app/controllers/items_controller.rb
#...
def index
  @items = Item.search(params[:search_query])
end
#...
```

```
# app/views/items/index.html.erb
<% @items.each do |item| %>
  <%= link_to item.name, item %>
<% end %>
```

This gives us a new error saying that there is no method `search` defined `Item`. At this point, we drop down a level of abstraction and write a unit test for `Item`.

Testing from the outside in

cont'd

```
# spec/models/item_spec.rb
describe Item, '.search' do
  it 'filters items by the search term' do
    desired_item = create(:item)
    other_item = create(:item)

    expect(Item.search(desired_item.name)).to eq [desired_item]
  end
end
```

This test leads us to correctly implement `Item.search`:

```
# app/models/item.rb
#...
def self.search(term)
  where(name: term)
end
#...
```

Now the unit test passes so we go back up to our feature test. We can successfully click on the item's name in the search results!

We keep following this pattern for the remaining test failures, dropping down to the unit test level when necessary, until we have a green test suite. Now our story has been successfully implemented!

Mocking and Stubbing

The goal of a feature test is to test the real system from end-to-end from the user's perspective. To do this, we use real database records and don't mock or stub any of our objects. We do stub calls to external websites (via webmock or a fake) since the network can be unreliable. Our tests should run without an internet connection.

When dropping down to the unit test level, we aggressively mock/stub out dependencies and collaborators. The goal of a unit test is to prove the functionality of the object being tested, not the functionality of its collaborators. Difficulty in testing two objects in isolation from each other often points to too tight coupling between them.

Further Reading

For some more great articles on testing, check out:

- How we Test Rails Applications
- Don't Stub the System Under Test
- Feature tests with Capybara

Write Readable Code

Edward Loveall - March 18, 2022

Code is read more than it's written. We write it once, and then read it back. It goes through review if we're on a team. It is read again when someone else needs to understand, add to, or modify that code. This includes ourselves weeks or months later.

Despite this, we tend to focus on the writing as the main "action". Writing is very important, but before we write we need to understand the context. We must read before we write. It's much easier to understand code if it's written well. Even in write-heavy situations like a new codebase, we eventually have to come back and read our first steps. We should optimize code to be read.

Names

Names describe what variables, methods, classes are or what they do. They outline the system we're working with. It's much easier to write terse names. They keep our lines short and make it easy to type those names again and again. For example `cc = CreditCard.find` instead of `primary_card,` or `def set_attr` instead of `set_user_profile_attribute`.

The problem is that non-descriptive names like `cc` or `set_attr` require further investigation to discover what they are and how they should be used. These examples favor easy writing, not reading.

Consider the concepts you learned to write this code, and try to capture that in names. Consider the *why* or *how* something is used instead of *what* it is. `initial_sign_up_profile` says a lot more than `profile`, and `lock_stats_table_for_data_export` says more than `lock_db`.

Readability is the goal here, not name length. You can absolutely make unreadable code with long names, especially lots of long names that are too similar. Go for readability, not some arbitrary length metric.

Abstracting Procedural Logic

The code we write to manipulate a system is different from the way we describe that manipulation. Imagine the process of "showing a modal dialog". That's how we'd describe it, even to code-proficient colleagues. We don't often describe this as "find the appropriate related DOM element and set CSS classes to be visible" but that's the level that code thinks on. It's our job to translate between those levels of abstraction.

When you have a long method, the classic fix is extract method. Extract method works by breaking up our unrefined code into named abstractions representing the underlying logic. Again, we're back to naming, but with a slightly different goal. A good name allows you to describe the functionality in a way that doesn't require the user to know every internal piece of the system. It allows them to learn (or re-learn) the deeper details as needed.

Write Readable Code

cont'd

Here's an example of showing a modal with JavaScript:

```
function async showModal(event) {
  const target = event.target;
  const modal = document.querySelector(
    event.target.dataset.relatedModalSelector
  );
  if (!modal || !modal.classList.contains("modal")) {
    return;
  }

  for (const element of document.querySelectorAll("modal")) {
    element.classList.add("hidden");
  }

  const data = modal.dataset;
  const modalTitle = JSON.parse(data.display)["title"];
  const modalContent = await fetchModalData(data.remoteUrl);
  modal.innerHtml = modalContent;
  modal.classList.remove("hidden");
}
```

If you already know how the modal system works, this is reasonable to read. But most people don't keep that information in their heads at all times. Abstracting this procedural logic will help anyone looking at this code with fresh eyes understand where they need to make changes:

```
function async showModal(event) {
  const modal = this.findPossibleModal(event);
  if (!this.isValidModal(modal)) {
    return;
  }

  await this.setModalContent(modal);
  this.hideEveryModal();
  this.revealModal(modal)
}
```

Write Readable Code

cont'd

The refactor makes the necessary steps for displaying modals clear and easily understood. If we need, we can find specific implementation details in extracted methods, and it's immediately clear what each method is doing. All the pieces exist on a similar level of abstraction; in this case manipulating related DOM elements. The encapsulating method `showModal` is an abstraction, too, that exists with abstractions on a similar level. It's easy to imagine other nearby interactions like `submitForm`, `syncUserProgress`, or `enableFocusMode`.

Testing

When testing, it's relatively common to isolate the setup phase from the rest of the test using abstractions like `let` or `before`. Many tests in the same file require similar (or the same) pieces of context to run, so consolidating that setup feels like a natural way to DRY up a test. Grouping related code can also feel similar to abstraction.

But this makes tests harder to read. That setup code defines the state of the system. More often than not we haven't seen these tests recently or ever. These pieces of setup are critical to understanding how to fix existing tests or add more. A test separated from its context forces us to memorize that context which distracts from our problem solving skills. A good test tells a story.

Most tests also test a system in multiple states; no single setup can speak for all scenarios. At best, shared setup will have to be redefined for individual tests, scatting that context. At worst, setup is entirely wasted as global setup goes unused. When we put shared setup at the top, we are assuming that all future tests need this particular setup. Write a few more tests and that assumption will likely prove false, causing us to reorganize the whole file or just live with the waste.

Keeping all of that setup inline makes that test much more readable. It's staggeringly not DRY, but DRY isn't a useful goal for tests. We do not need tests to be built on reusable abstractions and have a short line count. We need tests to give us predictable confidence in our system and help us refactor.

Broader Goals

It's worth remembering that specific metrics like code complexity, test coverage, and "DRY" aren't goals by themselves. The goal is code that we can easily understand and confidently change to give users the best possible software. Although "readable" is harder to measure, having it as a guiding principle can help us know when to bend or break these quantitative rules and build better software.

Avoid the Three-state Boolean Problem

Gabe Berke-Williams - February 24, 2014 | Updated March 6, 2019

Quick, what's wrong with this Rails migration?

```
add_column :users, :admin, :boolean
```

Yep - it can be null. Your Boolean column is supposed to be only `true` or `false`. But now you're in the madness of three-state Booleans: it can be `true`, `false`, or `NULL`.

Why to avoid NULL in Boolean columns

Boolean logic breaks down when dealing with NULLs. This StackOverflow answer goes into detail.

For example:

- `true AND NULL` is `NULL` (not `false`)
- `true AND NULL OR false` is `NULL`

Fortunately, it's easy to fix.

NOT NULL

Adding a `NOT NULL` constraint means that you'll never wonder whether a `NULL` value means that the user is not an admin, or whether it was never set. Let's add the constraint:

```
add_column :users, :admin, :boolean, null: false
```

But now the migration doesn't run.

Set a default value

The `NOT NULL` constraint means that this migration will fail if we have existing users, because Postgres doesn't know what to set the column values to other than `NULL`. We get around this by adding a default value:

```
add_column :users, :admin, :boolean, null: false, default: false
```

Avoid the Three-state Boolean Problem

cont'd

Now our migration runs, setting all users to be not admins, which is the safest option. Later, we can set specific users to be admins. Now we're safe and our data is normalized.

What's next

For more on the danger of null values, read If You Gaze Into nil, nil Gazes Also Into You. You can also browse our Postgres-related blog posts.

Pay the cost up front.

Don't distribute it to others.

German Velasco - October 2, 2020

In our world of rapid communication, it's easy to forget how disruptive our communication practices can be. We seldom stop to think about how we interrupt others. I'm no exception.

Lately, I've been trying to consider the golden rule for time: treat others' time as you like yours to be treated. In practice, I find that means paying the cost up front with my time instead of distributing it to others.

What do I mean? Let's look at these cases:

Meetings

Can you still remember the times when someone would tap you on the shoulder and ask to chat "really quick"? Nothing like an impromptu meeting to stop you from finishing the task you were working on. What was the meeting about? Who knows. But we were there anyway, so why not grab a room? Now we do "quick calls".

Those meetings weren't always bad. Neither are the quick calls. But too often, they are our first reaction to uncertainty. Our day is soon consumed with "quick calls" and "brief meetings", and our real work becomes the glue between them. Or maybe something worse happens — we get your actual work done outside of work hours when calls and meetings don't get in the way.

What if we paid the cost up front and planned the meeting ahead of time?

No meeting should be scheduled without a clear goal — a way of knowing when the participants meet the meeting's purpose. Who knows, you might realize you can send an email with questions and avoid the meeting altogether — what a victory!

And if you still need a meeting, have a goal for the meeting, a plan, and a hard stop. All the preparation will likely shorten the time it takes to meet.

Pay the cost up front by planning. Don't spread that cost to the people you invite to the meeting and compound the cost to your company.

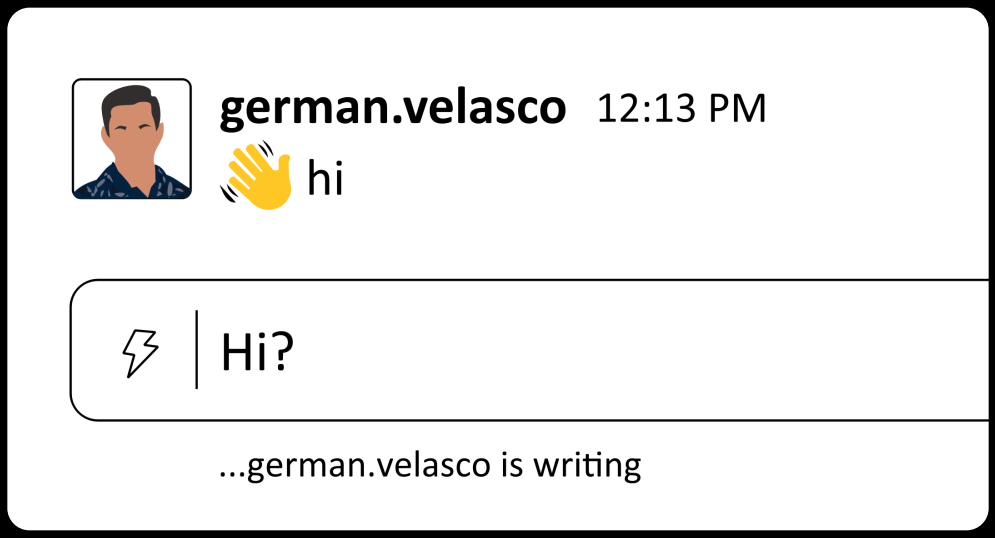
Pay the cost up front.

Don't distribute it to others.

cont'd

Messages

Have you ever been focused on solving the toughest problem in computer science, when a Slack notification pops up saying, "👋 hi"? You quickly open Slack — the solution escaping your thoughts — only to realize it was a direct message, and the person is just now typing a question.



Someone is giving your their stream of consciousness without regard for what you were doing. Oh, and that solution you had in mind, it's long gone.

It's good to be a team player, so be kind if you receive that message. But be a good team player and don't send that kind of message.

When considering sending a direct message, ask yourself, is it urgent? If so, go ahead with it. Synchronous communication directly wired to people's brains is there for that reason. But direct messages should be the exception.

If you need to ask a question, ask it on a public channel. Others might know the answer to your question (happens surprisingly often).

Pay the cost up front.

Don't distribute it to others.
cont'd

And when you write the question, take the time to state it fully. Include what you have tried and why it didn't work. When you do that, you might:

- discover you already have the answer, or
- realize haven't done enough digging on your own, or
- prepare a great question with all the context needed so that someone can help you.

Pay the cost up front by thinking through the question. Don't spread the cost to others by interrupting their concentration.

Pull requests

Have you ever been requested to review a 1000+ line pull request (PR)? Say good bye to your plans for that day, right?

The time trade-off is evident with pull requests. Time saved by the author of a large pull request directly translates to extra time spent by reviewers.

If you're the author of that work, don't open that monstrous PR. Spend some time trying to split it into independent commits and separate PRs. But be warned: it could take a significant amount of time.

Creating small, independent PRs is challenging work, and it takes a lot of *your* time. You could spend an extra 4 hours reorganizing the code! You might think it wasteful. But what if that saves each reviewer 4 hours of *their* time? Well, you just saved your team a whole lot of time and effort. Pay the cost up front. Don't distribute it to others.

Work tickets

What about work tickets that have single-line descriptions without any context? They're useful placeholders, but it's tough to start work like that.

When writing a description for a ticket, spend some time to write out what needs to happen, why it needs to happen, and potential pitfalls you've already considered. It can save the person taking the ticket hours of their time, and in this case, probably yours too.

If things are unclear, they'll probably send you a direct message and say "👋 hi". They'll then ask you what the ticket is all about, or more likely, they'll ask if you can hop on a "quick" call with them and another developer to figure out what the ticket is all about. Avoid that by paying the cost up front. Don't distribute it to others.

Credits