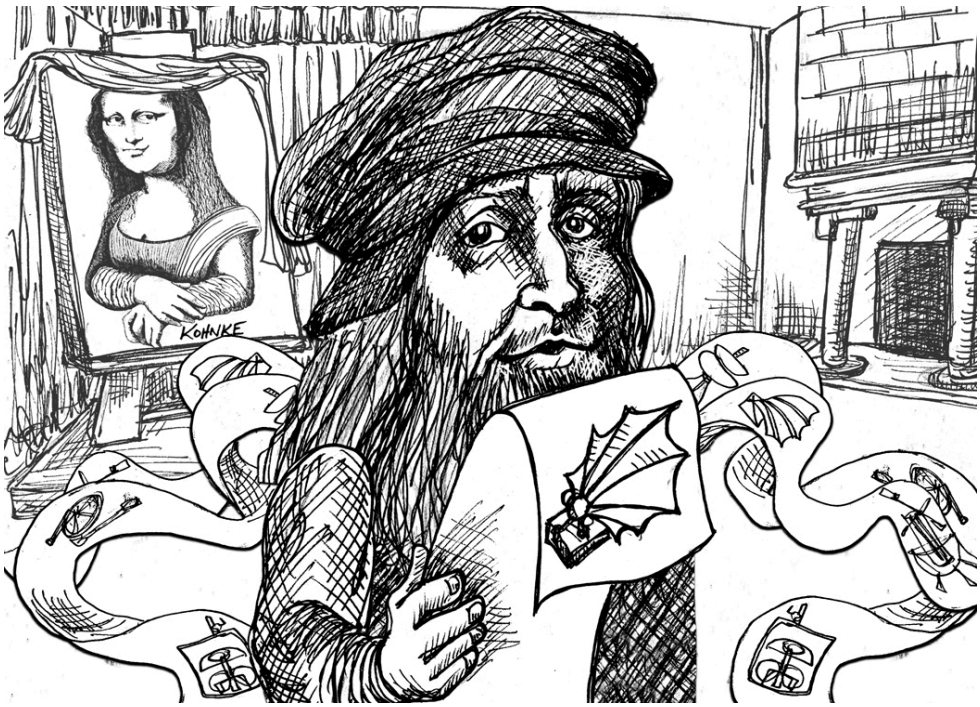
21

Continuous Design

By Jeff Langr



Product owners (and others responsible for helping to define a software product) determine the set of end goals that a system must accomplish for its users. Developers, in turn, build support for each of these goals in the system, in the form of code and configuration. As Jack W. Reeves taught us over three decades ago, this code (and configuration[1]) is the ultimate and definitive representation of "the design."[2]

---

1. For example, [PPP02], http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, and https://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (or just google SOLID).

2. https://www.developerdotstar.com/mag/articles/reeves_design.html. Republished in [PPP02].

The word *design* classically has two primary meanings:

- The plan for building something
- The form and functionality exuded by the current state of that something

Continuous design, also abbreviated as … Wait, that abbreviation is already taken. We could just utter the full phrase "continuous design" throughout this chapter. Ultimately, we'll just call it "design," because that's how we best build software systems: one conscious design decision after another.

As we were saying: Continuous design means that we continually plan how to accommodate new behaviors and that our system has a new design each time we add a new behavior. Most of that planning occurs as we begin writing the code that will provide the new behaviors.

Adding new code or updating existing code always changes the design, often only in a very small way.

Design is not a nebulous phase of development that occurs at the outset of a project. It is not solely the activity of drawing high-level sketches about what the system needs to look like.

Design is a continuous activity. Our system's design should command our constant attention, because it changes continuously, by definition. Sounds daunting, but in reality, it's easier than the alternative of generating an initial set of design models, then pretending we can ignore design after that.

Continuous Change

While software is made up of code, it's more useful to say that software is made of thousands of small behaviors (sometimes called *units*), each implemented in the form of code. Our job as programmers is to create and orchestrate these behaviors in a way that supports meeting the end goals of a system's users.

We must first translate end goals into a number of smaller software behaviors. Given the goal “present a random list of flash cards for learning Czech nouns,” we might implement the following sequence of behaviors.

1. Retrieve a random subset of all nouns.
2. For each noun:
    a. Present a question: A fill-in-the-blank sentence along with a multiple-choice listing of the four possible noun forms.
    b. Capture the user’s choice.
    c. Show the user the correct answer if their choice is incorrect.
3. Repeat steps 2a through 2c for each question that the user chose incorrectly.

There are usually many ways to skin a cat, so we might instead support an alternate set of steps. For example, the user might be able to continue selecting an answer for a single question until they get it correct. We might present hints with each failed answer. We might replace multiple-choice questions with fill-in-the blank questions as the user improves in skill.

If there are many design choices for an application itself, there are orders of magnitude more choices for how we write the underlying code. No, that’s not right. We have infinite design choices. Most of those choices will be suboptimal.

Our product managers and owners shape our software products to meet the perceived needs of the marketplace. Most software products continue to demand constant change in the form of new needs, and enhancements to existing needs.

In turn, we must constantly be able to meet the demand by shaping and enhancing existing software. A software system is a unique kind of product: It must not only continuously present a design that supports all current needs to date, but also accept that this design will need to be different tomorrow.

We might be able to build a slick, useful flash card system in a short period of time, perhaps a month or two. After an initial release, users will demand new behaviors. We’re selling new features—great! But we must also continuously return to the current system’s inherent design—its codebase.

In order to add new features, we're forced to figure out how to accommodate new concepts into the existing design, things that might have never been considered before.

Continuous Design

When it comes to making choices in software, we have only a few kinds of primary building blocks: modules (classes), functions (methods), data references (variables), and statements/expressions. The first three of these constructs can be named, to give us humans a better chance of understanding what concepts exist in the codebase, and also a better chance of finding the code we need to change or extend.

How we choose to manage and organize these building blocks is the activity of software design. Our system's design is the collection of choices we've made to this point in time.

We can easily make many bad choices when assembling these constructs, and in a very short time too. After making changes to our flash card app, it might still work, but our changes might have unnecessarily increased the cost for any future modifications.

If changing our flash card app results in unorganized, convoluted, and unclear code, accommodating new needs will be slow. We'll expend too much time navigating code to find all the places it must change. Once we find those locations (usually plural), we'll expend too much time understanding what the current code does. We'll then expend too much time carefully inserting our changes and ensuring that all the existing behaviors remain unbroken.

Each design choice we make impacts the cost for future choices. We succeed only if we minimize the negative design impact of each choice. This is the critical concept known as *continuous design*.

Sailing on the Four Cs of Continuous Design

An Unclean Code system makes it harder to

- Find code you're looking for.

- Understand where you need to make changes.
- Make changes without creating defects.
- Write tests to verify its behaviors.

You might recognize these characteristics from Chapter 13, "Clean Classes."

We have many design perspectives to help guide us toward a Clean Code system. As we change our system, we continue to reflect on one or more of these perspectives—SOLID, code smells, simple design, design patterns, and others, including various one-off principles such as the Law of Demeter or Tell-Don't-Ask.

For the concept of continuous design, we'll try—as many have before—to codify it all in a simple, hopefully memorable set of rules. Consider this YADP: Yet Another Design Perspective.

Our four Cs design considerations are

- Clarity: The programmer's intents in the system are all clearly stated.
- Conciseness: The programmer's intents in the system are implemented in a minimal amount of code.
- Confirmability: All unit behaviors in the system can be easily tested, in a way that also provides "living documentation" of the behavioral choices that were made.
- Cohesion: Each module in the system has a high level of cohesion—all its elements maximally relate to one another.

These considerations don't live in isolation. We'll talk about them one by one, but as we discuss, think about how they can both support and oppose each other.

Clarity

Getting code working is always our first step. We're challenged to add a small, new piece of behavior to the code. We think about how we want to solve the problem for a small amount of time, then quickly splash some code into our editor. We whittle the code a bit until we finally manage to get it working, then move on to the next slice of logic needed.

If we never had to read that code again, we might consider ourselves "done." Code complete! It ain't broke, and so we ain't gonna try 'n fix what's already working. Why risk breaking things?

But we will have to read that code again, when we're later required to change nuances of the logic! Or when we're asked "What does that code do in this case?" Or when we're required to add new related behaviors. Or in a half hour when we have to return to the code because it's not quite done yet.

For each of these needs, we want the code to be as clearly stated as possible. Anything unclear wastes time.

A challenge: Decipher all the nuances of the `retrieveWords` function. It might take you a half minute or two.

```
export const retrieveWords = async words => {
  const wordPrefix = words.length === 1 ? 'word' : 'w
  let wordList = "";
  for (let i = 0; i < words.length; i++) {
    const currentWord = words[i];
    const quotedWord = '"' + currentWord + '"';
    wordList += quotedWord;
    if (i !== words.length - 1) {
      wordList += ",";
    }
  }

  const finalPrompt = `Given a list of nouns separate
    `provide appropriate Czech language information f
    `${wordPrefix} ${wordList}`
  const response = await sendPrompt(`${format} ${fina
  const startIndex = response.indexOf('[')
  const endIndex = response.lastIndexOf(']') + 1
  const jsonText = response.slice(startIndex, endInde
  return JSON.parse(jsonText)
}
```

This is not necessarily bad code, but it could easily be better. It took more time for you to decipher than necessary, so we'll call it unclean code.

A continual focus on code clarity tells us to produce code that we can all consume quickly. In the following refactored code, `retrieveWords` appears as the bottommost function. Start your reading there.

```
const sliceJSONArrayFrom = response => {
  const startIndex = response.indexOf('[')
  const endIndex = response.lastIndexOf(']') + 1
  return response.slice(startIndex, endIndex)
}

const joinQuoted = words =>
  words.map(word => `"${word}"`).join(',')

const pluralizeIfMany = (word, list) =>
  list.length === 1 ? word : `${word}s`

export const retrieveWords = async words => {
  const finalPrompt = `Given a list of nouns separate
    `provide appropriate Czech language information f
    `${(pluralizeIfMany('word', words))} ${(joinQuote

  const response = await sendPrompt(`${format} ${fina

  return JSON.parse(sliceJSONArrayFrom(response))
}
```

If we're tasked with making changes to `retrieveWords`, understanding its overall intent and flow now occurs more quickly. The function initializes a `finalPrompt` as a long text string with a couple of embedded concepts:

- Either the text "word" or "words," depending on the size of the `words` array
- A joined string of the quoted words in that array

Those two concepts are now calls to tiny, isolated functions.

We then send a longer prompt string to an LLM, receiving a text response. Finally, we slice out the JSON text from the response string (which exists because we sent the LLM a desired format as part of the prompt), then parse it.

We can understand such a function in about fifteen seconds; it is largely a statement of policy. It uses extracted functions with names that capture their intent. Each extracted function is implemented in one to three lines of code. If we needn't change any of those implementations, that's one to three lines of hidden detail we can ignore for now and maybe forever.

If we need to change one of the helper functions, the streamlined policy declaration allows us to find it quickly and focus on a tiny bit of implementation detail. We can make our changes without considering any other code.

Editing Our Code

Crafting code with high clarity demonstrates that we care enough about our teammates (and ourselves) to spend a few moments editing our code before moving on.

As developers, we're good about getting things to work—our #1 job. We need to also remind ourselves that we're writers as well, and that others must consume what we write. Once we capture our ideas on paper, good writing demands that we revisit our spewage and edit it to emphasize clarity.

The sad fact is that we're not typically taught to edit our code. The vast majority of code out there shows it, and wastes copious amounts of your time as a result. WTFs per minute are high.

In fact, we're often told expressly not to edit code. "If it ain't broke, don't fix it." There's that lame, misapplied mantra again.

If other people can't make quick sense of your code, it *is* broken.

If you picked up a poorly written book, one that required you to reread sentences and paragraphs before they made sense, you'd consider it a bad, broken book (we bet you had at least one of these at university).

The mantra exists because we fear breaking things. Yes, code is a brittle material. It's pretty easy to make a dumb coding mistake in just about any programming language, yet not even spot it. As a result, we're inclined to avoid cleaning things up, despite how easy it usually is.

Fearing making changes to our code is a quality smell. There are simple paths for creating the controls that tell us, with every tiny change, whether or not we've broken already-working logic. Visit Chapter 14, "Testing Disciplines," to learn how.

Quick Steps to Clarity

- Extract implementation detail from multipurpose functions to create new functions with concise names.
- Move functions as appropriate to other modules. This results in increased cohesion, which helps readers find code where they might expect it.
- Replace comments with clear declarations.
- Write tests that double as documentation on all behavioral intents coded into the system.

Objections to Clarity

Seasoned programmers are able to immediately digest short code phrases. An experienced JavaScript developer can quickly comprehend the following couple of examples:

```
words.map(word => `"${word}"`).join(',')
list.length === 1 ? word : `${word}s`
```

Such phrases still demand careful reading, though. We must mentally assemble the tokens involved into the behavioral concept they represent. A small potential for misinterpretation exists, as does a small potential for a defect. The following line of seemingly idiomatic code should give us pause:

```
for (let i = 0; i <= words.length; i++)
```

Yet for many, it does not. That's not the idiom; this is:

```
for (let i = 0; i < words.length; i++)
```

The first `for` loop is likely a defect.

In any case, extracting small implementation details into intention-revealing functions streamlines code in `retrieveWords`:

```
const joinQuoted = words =>
  words.map(word => `"${word}"`).join(',')

const pluralizeIfMany = (word, list) =>
  list.length === 1 ? word : `${word}s`

export const retrieveWords = async words => {
  const finalPrompt =
    `Given a list of English nouns, separated by comm
    `provide appropriate Czech language information f
    `${(pluralizeIfMany('word', words))} ${(joinQuote
    // …
```

Declare Intent, Don't Ooze Details

We use functional pipelines partly because they allow us to replace details with declarations:

```
words.map(word => `"${word}"`).join(',')
```

That's loads easier to follow than the old-school procedural equivalent:

```
let wordList = "";
for (let i = 0; i < words.length; i++) {
  const currentWord = words[i];
  const quotedWord = '"' + currentWord + '"';
  wordList += quotedWord;
  if (i !== words.length - 1) {
    wordList += ",";
  }
}
```

The old-school code also has a dramatically increased possibility of hiding defects (hearken back to the `for` loop example).

Keeping Pipelines Streamlined

While it's reasonable to code short in-line functions within a pipeline:

```
export const mostExpensiveHighlyRatedBookInEachCategc
  categories.flatMap(category =>
    category.books
      .filter(book => !!book.title)
      .filter(book => book.rating > 4.0)
      .sort((a, b) => b.price - a.price)
      .map((book) =>
        ({ category: book.category, title: book.title
      .slice(0, 1))
```

they break the flow and require careful stepwise reading.

Replacing all the in-line lambdas—even the very short functions—with named functions does wonders for our ability to quickly follow the entire flow:

```
export const
mostExpensiveHighlyRatedBookInEachCategory = (categor
  const byPrice = (a, b) => b.price - a.price
  const highlyRated = book => book.rating > 4.0
  const hasTitle = book => !!book.title
  const categoryAndTitle = (book) =>
    ({ category: book.category, title: book.title })

  return categories.flatMap(category =>
    category.books
      .filter(hasTitle)
      .filter(highlyRated)
      .sort(byPrice)
      .map(categoryAndTitle)
      .slice(0, 1))
}
```

When reading the larger function, we scan past the one-liner function declarations and focus on the `return` statement. We're able to read and

mentally assemble what's going on more quickly now, because the pipeline is abstracted and not full of detail.

Hmm. The updated solution contains an idiomatic expression: `.slice(0, 1)`. It returns the first element of an array (less idiomatic) or an empty array if the array is empty (yes idiomatic). We don't eliminate our idioms unless they're causing readers to come to a grinding halt. If so, we find a way to abstract them. (Here, we might create a `firstOrDefault` function to supplant the slice call.)

Note that as we extract functions, we find insufficient cohesion in the form of misplaced bits of logic. Typical. A couple of functions extracted from this example's pipeline might find more appropriate homes in a book module.

Conciseness

A functional pipeline can provide code that is not only very clear, but also concise. The solution that requires the fewest possible tokens is maximally concise—but this is almost never what we really want.

Code that is maximally concise without consideration for clarity is obfuscated. The only legitimate time when we might deliberately obscure code is as part of a post-development deployment process (perhaps for compression or security interests), or as challenge/entertainment (see https://www.ioccc.org).

The absolute need for clarity means we must find a balance between conciseness and clarity. Finding the balance isn't tough, though—just ask your teammates. They'll let you know the moment something doesn't make sense. Watch them read through your code, and it'll usually be obvious when they're struggling with your ultra-optimized logic.

Let's start with an opposite example, however—code that's clear but not concise. The following `generateCard` function, along with a couple of helper functions, assembles a text-based flash card (complete with the answer on it—how curious!).

```
// generate integer from 0 to n-1
export const randomInt = n => {
  const randomValue = Math.random(); // Generate a ra
  const scaledValue = randomValue * n; // scale to de
  const flooredValue = Math.floor(scaledValue); // fl
  return flooredValue;
}

// return a word in its plural form if needed
const pluralizeIf = (word, isPlural) => {
  if (isPlural) {
      return `${word}s`; // Append 's' to make the wo
  } else {
      return word; // Return the original word if not
  }
}

// generates prompt text for a language card game
const generateCard = (adjectives, nouns) => {
  // Select a random adjective and noun from the prov
  const randomAdjectiveIndex = randomInt(adjectives.l
  const randomNounIndex = randomInt(nouns.length);
  const adjective = adjectives[randomAdjectiveIndex];
  const noun = nouns[randomNounIndex];

  // Determine the case and number randomly
  const nominativeOrAccusative =
    randomInt(2) === 0 ? 'nominative' : 'accusative';
  const isPlural = randomInt(2) === 0;
  const singularOrPlural = isPlural ? 'plural' : 'sir

  // Get correct forms of the noun & adjective based
  const correctNoun = noun[singularOrPlural][nominati
  const nounGender = noun.gender.toLowerCase();
  const correctAdjective =
    adjective[nominativeOrAccusative][singularOrPlura

  // Construct the prompt text for the game card
  const promptText = `Adjective: ${adjective.word}
Noun: ${noun.word}
Determine the ${singularOrPlural} ${nominativeOrAccus
Correct answer: ${correctAdjective} ${correctNoun}`;

  // Return the constructed prompt
```

```
  return promptText;
}
```

The code appears typical to us, and that's not a good thing. The core function, `generateCard`, is a top-to-bottom affair. It's not terribly long, but it includes comments to guide readers through what's happening in the next few statements. Each of the function's statements uses clear variable names and is readily comprehensible. We can carefully read through the code statement by statement, mentally assembling groups of statements into unit behaviors. What's not to like?

Indeed, the above example is how many developers happily continue coding for most of their career. But it's suboptimal for numerous reasons.

A reworked version results in additional small helper functions.

(Again, start your reading at `generateCard` at the bottom.)

```
export const randomInt = n => Math.floor(Math.random(

const pluralizeIf = (word, isPlural) => isPlural ? `$

const randomElement = (array) => array[randomInt(arra

const cases = ["nominative", "accusative"]
const numbers = ["singular", "plural"]
const generateRandomCardData = (adjectives, nouns) =>
  adjective: randomElement(adjectives),
  noun: randomElement(nouns),
  grammaticalCase: randomElement(cases),
  number: randomElement(numbers)
})

const formatCard =
  (adjective, noun, number, grammaticalCase,
   correctAdjective, correctNoun) =>
  `${adjective.word} ${pluralizeIf(noun.word, number
  Provide the ${number} ${grammaticalCase} case.
  Correct answer: ${correctAdjective} ${correctNoun}`

const generateCard = (adjectives, nouns) => {
  const { adjective, noun, grammaticalCase, number }
        generateRandomCardData(adjectives, nouns)
```

```
  const correctNoun = noun[number][grammaticalCase]
  const correctAdjective = adjective[grammaticalCase]
                                    [number]
                                    [noun.gender.toLc

  return formatCard(adjective, noun, number, grammati
                    correctAdjective, correctNoun)
}
```

Clarity is increased in a few ways. Distilling `generateCard` into three top-level chunks allows readers to quickly identify the overall policy.

- Generate random flash card elements using the adjectives and nouns passed in.
- Determine the correct answers to the challenge on the flash card by dereferencing the noun and adjective structures passed in.
- Return a formatted card containing all the relevant information.

We can immediately understand and trust each supporting function or piece of data in this solution. We might not even need to look at any supporting code or data.

The separate functions enhance clarity by allowing us to focus on small, understandable chunks. And while it would help to see the data structures involved (individual noun and adjective objects specifically), every statement otherwise stands on its own. No statements demand a comment. Comments would only increase the development, comprehension, and maintenance costs for this code.

As for understanding the data structures, that's what unit tests are for. Still, we might make a case for extracting a couple of functions to get just an ounce more isolation from implementation specifics in `generateCard`:

```
const generateCard = (adjectives, nouns) => {
  const { adjective, noun, grammaticalCase, number }
    generateRandomCardData(adjectives, nouns)

  const correctNoun = selectNoun(noun, number, gramma
  const correctAdjective =
    selectAdjective(adjective, grammaticalCase, numbe
```

```
  return formatCard(
    adjective,
    noun,
    number,
    grammaticalCase,
    correctAdjective,
    correctNoun)
}
```

We also spotted opportunities to cohere seemingly disparate elements, by introducing a new abstraction—the notion of an object that captures the underlying data elements of a random card. Yet there's still something amiss with our function. The excessive number of parameters returned and passed about not only muddles the solution, but also decreases its concision.

Let's make a final cleanup pass.

```
const generateRandom = (adjectives, nouns) => ({
  adjective: randomElement(adjectives),
  noun: randomElement(nouns),
})

const formatCard =
(phrase, caseAgreement, correctAdjective, correctNoun
  `${phrase.adjective.word} ${pluralizeIf(
                               phrase.noun.word,
                               caseAgreement.number

  Provide the ${caseAgreement.number} ` +
              `${caseAgreement.grammaticalCase} case.

  Correct answer: ${correctAdjective} ${correctNoun}`

const selectNoun = (noun, caseAgreement) =>
  noun[caseAgreement.number][caseAgreement.grammatica

const selectAdjective = (adjective, gender, caseAgree
  adjective[caseAgreement.grammaticalCase]
           [caseAgreement.number]
           [gender.toLowerCase()]
```

```
const cases = ["nominative", "accusative"]
const numbers = ["singular", "plural"]

const generateCaseAgreementChallenge = () => ({
  grammaticalCase: randomElement(cases),
  number: randomElement(numbers)
})

const generateCard = (adjectives, nouns) => {
  const phrase = generateRandom(adjectives, nouns)
  const caseAgreement = generateCaseAgreementChalleng

  const correctAdjective = selectAdjective(
    phrase.adjective, phrase.noun.gender, caseAgreeme
  const correctNoun = selectNoun(phrase.noun, caseAgr

  return formatCard(
    phrase, caseAgreement, correctAdjective, correctN
}
```

Slightly more-thoughtful data groupings make `generateCard` seem less of a haphazard bustle of wayward local variables. Each of its handful of steps emphasizes the key elements involved. We also reworked `selectAdjective` to accommodate a gender parameter instead of a noun. Our choice emphasizes that a noun's gender, specifically, is key to determining the proper adjective.

Code Duplication

Being concise in code means implementing concepts once and only once. We often find small chunks of implementation detail replicated through a module or codebase—usually from a couple to a handful of lines that are oft repeated. Here's a commonly found example—repetitive error handling:

```
export const postItem = (request, response) => {
  const itemDetails = retrieveItem(request.body.upc)
  if (!itemDetails) {
    response.status = 400
    const errorMessage = 'unrecognized UPC code';
    logError(errorMessage)
    return response.send({error: errorMessage})
  }
```

```
  const checkout = Checkouts.retrieve(request.params.
  if (!checkout) {
    response.status = 400
    const errorMessage = 'nonexistent checkout';
    logError(errorMessage)
    return response.send({error: errorMessage})
  }

  const newCheckoutItem = Checkouts.addItem(checkout,

  sendResponse(response, newCheckoutItem, 201)
}
```

Repetition increases costs for a number of reasons. The essence of the controller function `postItem` is obscured by repetitious error handling code. This duplication also decreases clarity, by mixing implementation details into policy flow.

```
const sendRequestError = (response, message) => {
  response.status = 400
  logError(errorMessage)
  response.send({error: message})
}

export const postItem = (request, response) => {
  const itemDetails = retrieveItem(request.body.upc)
  if (!itemDetails)
    return sendRequestError(response, 'unrecognized U

  const checkout = Checkouts.retrieve(request.params.
  if (!checkout)
    return sendRequestError(response, 'nonexistent ch

  const newCheckoutItem = Checkouts.addItem(checkout,
  sendResponse(response, newCheckoutItem, 201)
}
```

Often, eliminating duplication involves replacing concrete details with abstractions. It's generally a win-win-win-win situation (increased conciseness, clarity, and ease of confirmability, plus a springboard to increased cohesion), but we don't go overboard by obsessing over two lines of code that happen to look the same. We seek to find and eradicate

duplicate implementations of concepts, not incidentally common lines of code.

Duplication fosters many increases in cost/effort:

- Time to read and comprehend cost
- Time to search code in general (we're always wading through increased amounts of irrelevant results)
- Time to test
- Effort to find all points of change for a replicated concept
- Risk of not finding all necessary points of change
- Time to analyze whether variances found in duplicated code are deliberate or if they represent defects
- Effort to change duplicated code
- Cost to refactor code
- Propagation of defect costs, when common code is defective

We've worked on many systems that were two-to-three times as large as they might have been due to rampant code duplication. Oh, the days of getting paid by the line of code! (Just kidding. It never happened for any of us.)

One example included an operating room scheduling system, which by definition needed to do a lot of work with dates and timestamps. Among many other amusements, we found a common five-line piece of (old-school) Java date handling logic. These five lines were repeated in over fifty places throughout the codebase. Our costs increased dramatically when we sought to replace the old-school logic with a modern library.

Extract-and-move is once again our trustworthy workhorse for tackling duplication problems.

- Identify a clump of implementation detail that represents a singular concept.
- Extract it to a function.
- Focus on the new function in isolation. Move it to another module if appropriate.

As awareness of the existence of a new module increases, we start to spot more opportunities to move additional related behaviors into it. The cohesion of our modules increases as a result.

Conciseness Nits

Every unnecessary token adds to the clutter of our code. We can and should often use things like idiomatic constructs and syntactic sugar to reduce the amount of unnecessary code that we must scan and mentally assemble. A few pet nits of ours:

Return Boolean Expressions

Learning a language well means we must not only learn its syntax, but also learn the most concise way of representing common concepts. Some of these streamlined constructs might go so far as to be classified as idiomatic. (Yes, we've used that word a number of times; we guess we should finally define it: Idiomatic code isn't obvious the first time you see it.)

A ternary operator is idiomatic: In an isolated context, it doesn't inherently provide all the information you need to immediately decipher it. The second or third time you see it, however, you'll know how to interpret it—it has become obvious to you. It's like riding a bicycle; you won't likely ever need to be reminded after that.

Sometimes we see anti-idioms:

```
const hasTitle = book => {
  if (book.title !== null) {
    return true
  } else {
    return false
  }
}
```

Anti-idiom? Despite the fact that this construct demands five precious vertical source lines, we usually digest it as a single chunk when we see it (because unfortunately, we see it a lot). Sometimes we even notice when a chucklehead has sneakily reversed the order of the `true` and `false` returns.

We rarely say that things are flat-out wrong in programming, but this is one of those cases.

Simply (and we don't use that word lightly) return the boolean expression:

```
const hasTitle = book => book.title !== null
```

Unnecessary `Else`

Maybe we needn't embrace the ternary operator—worse things than `if-else` statements pollute our coding world—but if we're going to code a conditional return:

```
const pluralizeIf = (word, isPlural) => {
  if (isPlural) {
      return `${word}s`
  } else {
      return word
  }
}
```

we should at least have the decency to not include the unnecessary `else` keyword. An early return suffices:

```
const pluralizeIf = (word, isPlural) => {
  if (isPlural) {
    return `${word}s`;
  }
  return word
}
```

No, we really don't need the braces, though odds are good our team has adopted a coding standard that insists on them. But it's really just overprotective clutter. It's hard to mess up if you start with a short, single-line `if` statement:

```
const pluralizeIf = (word, isPlural) => {
  if (isPlural) return `${word}s`
```

```
    return word
  }
```

We write tests if we remain worried. In any case, there's honestly not much good reason to eschew a ternary for this example:

```
const pluralizeIf = (word, isPlural) => isPlural ? `$
```

We get it, though, if you object on aesthetic grounds.

And So On

We could dedicate an entire book to clutter in code. Most of it, however, you'll figure out on your own. We don't seek clever code; we instead seek elegant code. Elegant code says exactly what it needs, without too few or too many words. It is both clear and concise.

Confirmability

Our continuous design journey requires that we know what our code is intended to do. Without that knowledge, any change is unsafe. A system may contain many thousands of behavioral units. An unnoticed change in behavior to any one of them can allow us to unleash a costly defect in production.

Fortunately, we can write thousands of small, fast unit tests that verify whether we've created any regressions. These tests can tell us within seconds the moment we broke something. We'll need other kinds of tests at higher levels, of course, such as end-to-end functional tests, performance tests, load tests, and contract tests. But if we want to move fast, we need to be able to rapidly create and manage tests around the unit implementations.

Fear Degrades Design

In contrast, the lack of such tests gradually slows us down. As the amount of (not unit-tested) code increases, our costs to verify it increase. We can manually step through only so many test cases.

We can write integration tests, but they're generally costlier to build and maintain. They also create a slower feedback loop. Finally, it's unrealistic to expect that you'll be able to cover all the thousands of intentional decisions and logic variants that went into the codebase by virtue of integration tests (which also can increase the challenge of pinpointing the defect when they do fail).

Without fast feedback in the form of unit tests, we relegate ourselves to that "it ain't broke, don't fix it" mentality. Our codebase degrades by definition. We don't have the confidence to ensure that the code remains cohesive, clear, and concise. Instead, we habitually slap out "first drafts": We get our code working, then immediately move on to the next thing without making any edits.

It seems fast at first, yet the results of fear-driven coding quickly become apparent in the codebase. Developers learn to do the worst possible thing to a system, rather than what they know to be the best thing for the system.

A real example from a high-performance, historically famous large system: Developers needed to create variants of behaviors existing within typically long (100+ lines) C++ member functions. Shipping defective versions of these existing behaviors could cost the company millions of dollars per minute.

As a result, developers habitually replicated (i.e., copied/pasted) entire member functions, then made their changes within these copies. They didn't change the original functions—no one wanted to be the one who broke code that was already working. ("Why did you even touch that code?")

Fear significantly increases code duplication and costs.

Conflated Units Imply Tough Tests

If we write code like the following `postCheckoutTotal` function, writing tests for it will be daunting:

```
export const postCheckoutTotal = (request, response)
  const checkoutId = request.params.id
```

```
const checkout = Checkouts.retrieve(checkoutId)
if (!checkout) {
  response.status = 400
  response.send({error: 'nonexistent checkout'})
  return
}

const messages = []
const discount = checkout.member ? checkout.discour

let totalOfDiscountedItems = 0
let total = 0
let totalSaved = 0

checkout.items.forEach(item => {
  let price = item.price
  const isExempt = item.exempt
  if (!isExempt && discount > 0) {
    const discountAmount = discount * price
    const discountedPrice = price * (1.0 - discount

    // add into total
    totalOfDiscountedItems += discountedPrice

    let text = item.description
    // format percent
    const amount = parseFloat(
      (Math.round(price * 100) / 100).toString()).t
    const amountWidth = amount.length

    let textWidth = LineWidth – amountWidth
    messages.push(pad(text, textWidth) + amount)

    total += discountedPrice

    // discount line
    const discountFormatted = '-' + parseFloat(
      (Math.round(discountAmount * 100)/100).toStri
    textWidth = LineWidth - discountFormatted.lengt
    text = `   ${discount * 100}% mbr disc`
    messages.push(`${pad(text, textWidth)}${discour

    totalSaved += discountAmount
  }
  else {
```

```
      total += price
      const text = item.description
      const amount = parseFloat(
        (Math.round(price * 100) / 100).toString()).t
      const amountWidth = amount.length

      const textWidth = LineWidth – amountWidth
      messages.push(pad(text, textWidth) + amount)
    }
  })

  total = Math.round(total * 100) / 100

  // append total line
  const formattedTotal = parseFloat(
    (Math.round(total * 100) / 100).toString()).toFix
  const formattedTotalWidth = formattedTotal.length
  const textWidth = LineWidth – formattedTotalWidth
  messages.push(pad('TOTAL', textWidth) + formattedTo

  if (totalSaved > 0) {
    const formattedTotal = parseFloat(
      (Math.round(totalSaved * 100) / 100).toString()
    console.log(`formattedTotal: ${formattedTotal}`)
    const formattedTotalWidth = formattedTotal.length
    const textWidth = LineWidth – formattedTotalWidth
    messages.push(pad('*** You saved:', textWidth) +
  }

  totalOfDiscountedItems =
    Math.round(totalOfDiscountedItems * 100) / 100

  totalSaved = Math.round(totalSaved * 100) / 100

  response.status = 200
  // send total saved instead
  response.send(
    { id: checkoutId,
      total,
      totalOfDiscountedItems,
      messages,
      totalSaved })
}
```

Capturing all the nuances and writing tests for all relevant cases might take a few hours. Also, there's a good possibility that we don't cover all the cases, which increases our risk of shipping defects.

Small, Isolated Units Imply Simple Tests

It is much easier to write small, simple tests as we craft the logic for assembling a checkout receipt. No doubt you can spot a few of the bits of duplication in the midst of `postCheckoutTotal`, as well as the number of opportunities for extracting conceptual units to new functions.

Here's an improved solution, a work in progress.

```
const sendSuccessResponse = (response, body) => {
  response.status = 200
  response.send(body)
}

const sendErrorResponse = (response, message) => {
  response.status = 400
  response.send({ error: message })
}

export const postCheckoutTotal = (request, response)
  const checkoutId = request.params.id
  const checkout = Checkouts.retrieve(checkoutId)
  if (!checkout)
    return sendErrorResponse(response, 'nonexistent c

  const { totals, messages } = createReceipt(checkout

  sendSuccessResponse(response, {
    id: checkoutId,
    messages,
    total: round2(totals.total),
    totalOfDiscountedItems: round2(totals.totalOfDisc
    totalSaved: round2(totals.totalSaved)
  })
}
```

The function `postCheckoutTotal`, now a declaration of policy, can be covered with a couple of "end-to-end" tests.

The module `receipt.js` exposes code that can be directly and easily unit tested:

```
import { pad } from '../../util/stringutil'
import { calculateTotal,
         calculateTotalOfDiscountedItems,
         calculateTotalSaved } from './checkout-model

const LineWidth = 45
const Indent = `    `

export const round2 = amount => Math.round(amount * 1

const formatAmount = amount => parseFloat(round2(amou

const shouldApplyDiscount = (checkout, item) =>
  !item.exempt && memberDiscountPct(checkout) > 0

const calculateDiscountAmount = (checkout, item) =>
  memberDiscountPct(checkout) * item.price

const memberDiscountPct =
  checkout => checkout.member ? checkout.discount : 0

const lineItem = (text, lineAmount) => {
  const amount = formatAmount(lineAmount)
  const amountWidth = amount.length
  const textWidth = LineWidth - amountWidth
  return pad(text, textWidth) + amount
}

export const createReceiptMessages = (checkout, total
  const messages = []
  checkout.items.forEach(item => {
    messages.push(lineItem(item.description, item.pri
    if (shouldApplyDiscount(checkout, item)) {
      messages.push(lineItem(
        `${Indent}${memberDiscountPct(checkout) * 100
        -calculateDiscountAmount(checkout, item)))
    }
  })
```

```
  messages.push(lineItem('TOTAL', round2(totals.total
  if (totals.totalSaved > 0)
    messages.push(lineItem('*** You saved:', totals.t
  return messages
}

export const createReceipt = checkout => {
  const totals = ({
    total: calculateTotal(checkout),
    totalSaved: calculateTotalSaved(checkout),
    totalOfDiscountedItems: calculateTotalOfDiscounte
  })
  return {
    totals,
    messages: createReceiptMessages(checkout, totals)
  }
}
```

By separating out `receipt.js`, we've also increased cohesion. The `postCheckoutTotal` function is primarily concerned with orchestrating the creation of a response to a request. It delegates the calculation specifics of assembling a receipt and totals to the `receipt.js` module.

The `receipt.js` still includes a few utility/helper functions that could be again moved, then tested in isolation (notably `round2` and `formatAmount`).

Here's the final cohesive module, `checkout-model.js`:

```
export const calculateTotal = checkout => {
  let total = 0
  checkout.items.forEach(item => {
    if (!item.exempt && (checkout.member ? checkout.d
      total += item.price * 1.0 -
                  (checkout.member ? checkout.discount
    else
      total += item.price
  })
  return total
}

export const calculateTotalSaved = checkout => {
  let totalSaved = 0
```

```
  checkout.items.forEach(item => {
    if (!item.exempt && (checkout.member ? checkout.d
      totalSaved +=
        (checkout.member ? checkout.discount : 0) * i
  })
  return totalSaved
}

export const calculateTotalOfDiscountedItems = checkc
  checkout.items
    .filter(item => !item.exempt &&
                    (checkout.member ? checkout.discc
    .reduce((total, item) => total + item.price *
       (1.0 - (checkout.member ? checkout.discount :
```

Now that it is isolated, we can focus on improving clarity and conciseness within these calculate functions. We can put tests in place in short order. It took ten minutes to create these tests for `calculateTotal` :

```
import {calculateTotal} from "./checkout-model";

describe('calculate total', () => {
  it('is zero when no items exist', () => {
    const checkout = {
      items: []
    }
    const result = calculateTotal(checkout)

    expect(result).toBe(0)
  })

  it('is price of single undiscounted item', () => {
    const checkout = {
      items: [{ price: 42 }]
    }

    const result = calculateTotal(checkout)

    expect(result).toBe(42)
  })

  it('discounts item when member attached & item not
    const checkout = {
      member: {},
```

```
    discount: 0.1,
    items: [{
      exempt: false,
      price: 100
    }]
  }

  const result = calculateTotal(checkout)

  expect(result).toBe(90)
})

it('does not discount item when exempt', () => {
  const checkout = {
    member: {},
    discount: 0.1,
    items: [{
      exempt: true,
      price: 100
    }]
  }

  const result = calculateTotal(checkout)

  expect(result).toBe(100)
})

it('does not discount item when no member attached'
  const checkout = {
    discount: 0.1,
    items: [{
      exempt: true,
      price: 100
    }]
  }

  const result = calculateTotal(checkout)

  expect(result).toBe(100)
})

it('totals discounted and non-discounted items', ()
  const checkout = {
    discount: 0.1,
    member: {},
```

```
      items: [
        { exempt: true, price: 5 },
        { exempt: true, price: 10 },
        { exempt: false, price: 100 },
        { exempt: false, price: 200 },
      ]
    }

    const result = calculateTotal(checkout)

    expect(result).toBe(5 + 10 + 90 + 180)
  })
})
```

The tests gave us the confidence to fearlessly refactor `calculateTotal` to a clearer solution:

```
const memberDiscountPct = checkout =>
  checkout.member ? checkout.discount : 0

const shouldApplyDiscount = (checkout, item) =>
  !item.exempt && memberDiscountPct(checkout) > 0

const calculateDiscountedPrice = (checkout, item) =>
  item.price * (1.0 - memberDiscountPct(checkout))

export const calculateTotal = checkout => {
  const totalDiscountable = checkout.items
    .filter(item => shouldApplyDiscount(checkout, ite
    .reduce((total, item) => total +
              calculateDiscountedPrice(checkout, item
  const totalNonDiscountable = checkout.items
    .filter(item => !shouldApplyDiscount(checkout, it
    .reduce((total, item) => total + item.price, 0);
  return totalDiscountable + totalNonDiscountable;
}
```

It looks like our refactoring increased the lines of code, for `calculateTotal` at least. However, the overall number of lines in the module decreases by a handful after we made similar changes to the other calculation functions as well.

When Should We Ensure Confirmability?

Many developers prefer to write a significant chunk of code, then later come back and meet what they view as their obligation to achieve code coverage mandates. The mandate for code coverage—the measure of what percent of code is exercised by tests—becomes a self-fulfilling prophecy: We get what we ask for. If a development team is mandated a minimum coverage metric of 75%, we'll get that and no more.

Seventy-five percent coverage doesn't sound bad until you invert it: One-quarter of our system has insufficient unit tests. They're also often the more complex parts of our system: They're harder to test, and when meeting a metric we don't really believe in, we'll do it in the most expeditious way we can imagine.

The reason so much of our code is harder to test is because testing it wasn't a paramount concern while we were coding it. Our primary interest was to just get the damn thing working. As a result, the complex parts and even the not-so-complex parts are tougher to test. Our code is not easily confirmable.

Writing tests after the fact is harder because the existing code

- Lacks clarity. It is overly complex, because it wasn't edited. It's tougher to figure out the basic question of what the functions do.
- Lacks cohesion. It becomes increasingly difficult to discern all the actual developer intents, because they aren't all in one place, and they aren't decoupled from other intents.
- Lacks conciseness. It contains excessive code, as well as excessively repeated constructs that must be retested in multiple contexts.
- Contains deep dependency chains. As a result, tests demand increased effort to set up the appropriate data/context. The existence of what are typically private dependencies means that we must reshape the code to allow tests to inject the dependencies.

One more problem with that 75% or 80% or whatever code metric: The remaining portion of our code is a 20%+ blob of "who really knows what this stuff does." Good tests not only provide us with the ability to edit code without fear, but they also document all the developer intents.

If writing tests later creates so many problems (or, more positively stated, doesn't help us sufficiently enough), the obvious answer is to start with tests. With each new behavior we must add, we write a test that captures that behavior. Our test's name summarizes that behavior (which helps us later find the behaviors we seek). Our test's code provides an example that pins down the design for how we effect that behavior.

Our code coverage becomes a self-fulfilling prophecy: We attain complete coverage on all the behaviors we needed and designed into the system. We get high confidence to continuously shape its design to be clear, cohesive, and concise. Our system is eminently confirmable.

You might be wondering about what that coverage metric mandate should be. The right answer: (a) Mandates are a bad idea, and (b) the coverage number is never what's important. What's important is that developers are sold on the value of delivering what we know to be working software.

Cohesion

It was infeasible to get to this last C criterion for design without previously mentioning cohesion. Our continuous design criteria necessarily intertwine to support each other, and at times they can be at odds with one another. Increasing clarity might require diminishing conciseness, for example.

But here we are, and cohesion deserves its own focused discussion.

We're building `Czecher`, a flash card application designed to help us ingrain challenging aspects of the Czech language. As a first step, we've created the module `words.js` to allow us to add new nouns to our collection of words to practice:

```
import { retrieveWord } from '../prompts/languageClie

const definitions = {}

export const clearWords = () =>
  Object.keys(definitions).forEach(key => delete defi

export const loadDefinition = (word, definition) =>
  definitions[word] = { …definition, word }
```

```
export const addWord = async word => {
  if (definitions[word]) return

  const definition = await retrieveWord(word)
  definitions[word] = { …definition, word }
}

export const definition = word => definitions[word]

export const allDefinitions = () =>
  Object.values(definitions)
```

The `addWord` function takes an English word as an argument (e.g., "dog"). If a definition has already been loaded for the word, the function does nothing. Otherwise, `addWord` calls `retrieveWord` to look up information using a third-party resource (OpenAI in this case). The resulting definition is added to a local store. Calling the definition function with a word returns the detailed information.

Some tests might add a little color:

```
import { when } from 'jest-when'
import { addWord, clearWords, definition,
         allDefinitions, loadDefinition }
  from './words'
import { retrieveWord } from '../prompts/languageClie

jest.mock('../prompts/languageClient')

describe('addWord', ()  => {
  const orangeDefinition = {
    word: 'orange',
    gender: 'MI',
    singular: { nominative: 'pomeranč', accusative: '
    plural: { nominative: 'pomeranče', accusative: 'p
  }
  const dogDefinition = {
    word: 'dog',
    gender: 'MA',
    singular: { nominative: 'pes', accusative: 'psa'
    plural: { nominative: 'psi', accusative: 'psy' }
  }
```

```
  beforeEach(() => {
    clearWords()
    jest.resetAllMocks()
  })
  it('retrieves and saves definition on add', async (
    when(retrieveWord).calledWith('orange')
      .mockResolvedValueOnce(orangeDefinition)

    await addWord('orange')

    expect(definition('orange')).toEqual(orangeDefini
  })

  it('does nothing if already added', async () => {
    loadDefinition('orange', orangeDefinition)

    const word = definition('orange')

    expect(word).toEqual(orangeDefinition)
    expect(retrieveWord).not.toHaveBeenCalled()
  })

  const add = async (word, definition) => {
    when(retrieveWord).calledWith(word)
        .mockResolvedValueOnce(definition)
    await addWord(word)
  }

  it('persists multiple words', async () => {
    await add('orange', orangeDefinition)
    await add('dog', dogDefinition)

    const definitions = allDefinitions()

    expect(definitions).toEqual([orangeDefinition, dc
  })
})
```

The `words` module is our first incremental step toward an MVP.[3] As a quick measure, we chose to persist the words in an in-memory JavaScript object (`{}`), where the keys are the words and the values provide the language detail we need. The constant `orangeDefinition` provides one example.

3. Minimum Viable Product.

Here's our continuous design report card.

- Clarity: Pass. All the operations appear in short, JavaScript-idiomatic functions, without any C-for-Cleverness (for which we would immediately get a failing mark).
- Confirmability: Pass. All code was driven into existence by behavioral tests.
- Conciseness: Pass. The code seems about as short as it could get.
- Cohesion: Fail. Wait, fail?

Our intent is to replace the JavaScript object with a proper persistence mechanism. Unfortunately, we've conflated our little bit of definition-management logic with logic supporting storage of the definitions. When we do change, we'll have to open up the `words` module and poke through most of its methods to update implementations.

You'll likely recognize that we're describing a Single Responsibility Principle (SRP) violation. SRP and cohesion both say the same thing, minus some nuances. Cohesion is how well the elements of a module align to the same purpose.

Had we adhered to a cohesive solution, we might have found it a bit easier to build the solution in the first place. We'd also have been ready to immediately accommodate changes, rather than have to prefactor our code to support the change.

We'll refactor the code now, to demonstrate how a cohesive solution increases the clarity of the code.

```
import { retrieveWord } from '../prompts/languageClie
import * as Data from '../persistence/database'

export const clearWords = () =>
  Data.deleteAll()

export const loadDefinition = (word, definition) =>
  Data.add(word, definition)
```

```
export const addWord = async word => {
  if (Data.containsKey(word)) return

  const definition = await retrieveWord(word)
  Data.add(word, { word, …definition})
}

export const definition = word => Data.get(word)

export const allDefinitions = () => Data.allValues()
```

The implementation specifics around JavaScript objects have been replaced with abstractions: `add`, `get`, `allValues`, `containsKey`, and `deleteAll`. The underlying implementation could change from an object, to a list, to interaction with a third-party key/value store or relational database. The now-cohesive `words` module wouldn't be impacted by these changes.

The module `database.js` provides an interface that abstracts the notion for persisting data, with nothing exported that betrays the underlying data structure. Future changes to its implementation don't impact other code in the system.

```
const data = {}

export const containsKey = key => data[key]

export const deleteAll = () =>
  Object.keys(data).forEach(key => delete data[key])

export const add = (key, value) =>
  data[key] = value

export const get = word => data[word]

export const allValues = () => Object.values(data)
```

When Else Do We Design?

With a continuous design mentality, we seek to keep costs low by reviewing our code's adherence to the four Cs, a collection of code criteria

that we can continuously consult. When do we do that? Why, continuously of course, or at least continually.

But we don't only consider design when we're writing production code. It's a part of virtually every step of software development.

Up-front Design

We initiate a project by determining what new capabilities our product will bring to its users. Necessarily this requires some level of understanding about the design of the system—what it accomplishes and what its constraints are, for example.

As part of answering our analysis and design efforts, we might first produce summary diagrams that capture salient elements of the system's current design. We would then also create speculative sketches for what the design will need to look like in order to accommodate the desired new behaviors.

Stakeholders usually want to know when the project will be done at this point in the planning process. Estimates are a design consideration: Not only must we have a sense of how much time it would take to develop a new feature, but we also must know the extent to which the current design resists the new feature. Ideally, the amount of extra effort needed is "zero," which can be true if we've adhered to the four Cs well enough.

The estimates themselves might feed back into the project plan. The powers that be might decide to trim their wishlist due to unexpectedly high estimates. Given these changes to their desired scope, we reconsider the design and provide updated estimates.

Our project estimates at this time cover larger initiatives that might take weeks or months to complete. The estimates are high level as a result; we invest enough time in them to get a ballpark sense of the initiative. We might use relative sizes—small, medium, large, extra-large—to figure out which ones we can guarantee we'll deliver in the next quarter.

As we ready for development, we seek to slice these larger challenges into thinner pieces of work that we can start and deliver within shorter periods of time (perhaps hours or days). An understanding of the system's design is

once again required at this point, to ensure that our ideas around slicing up work align with the realities of the system.

Readying for Work

As we get ready to code a feature, the product folk (product owners, product managers, business analysts, UX designers, and others) are working to ensure that they've captured all the actual specifics needed for us to get started. When they meet with us to describe the new feature, we ask questions, and we dig a bit again into the system's design.

Sometimes we uncover the devil when the product people reveal the details for the feature. We discover that things aren't as ideal as we'd speculated earlier. It's also possible that the system has changed enough in the interim to make things more difficult.

Sometimes answers to our questions reveal hidden complexity. "Oh, I didn't think that bell or whistle would be a big deal," they might tell us. It might not be, from their perspective; from ours, it might be something that will be very difficult to fit into the existing design.

Starting Work

We break down our work first into smaller behavioral units. If our work is to add support for adverbs to the `Czecher` app, we might break that effort into the following smaller slices.

- Support adding a single adverb to the word list.
- Bulk-load a CSV file of adverbs.
- Add flash card examples containing adverbs.

Those slices might still be a little too large. We might split "support adding an adverb" into a few technical challenges.

- Create and incorporate LLM prompt text to specify the format for adverbs.
- Add persistence support for adverbs.
- Rework card building logic to provide examples involving adverbs.

Such decomposition necessarily involves an awareness of design, and may also spur a discussion around how the current and/or speculative design must change.

Doing Work

We write tests to drive in each unit behavior. A unit test describes the behavior and provides a “live” example demonstrating how clients interact with the system to effect that behavior. The test, as a result, is our primary mechanism for designing and documenting the interface to the system.

For each test we write, we code the behavior that we hope realizes the need specified in the test. We run our tests, correcting the code (i.e., the design) until the tests pass. We revisit the coded behavior, and we edit it for clarity, conciseness, and cohesion before moving on to the next test.

We ship when we’ve written tests for all the behaviors, as well as gotten all the tests to pass.

Design is omnipresent.