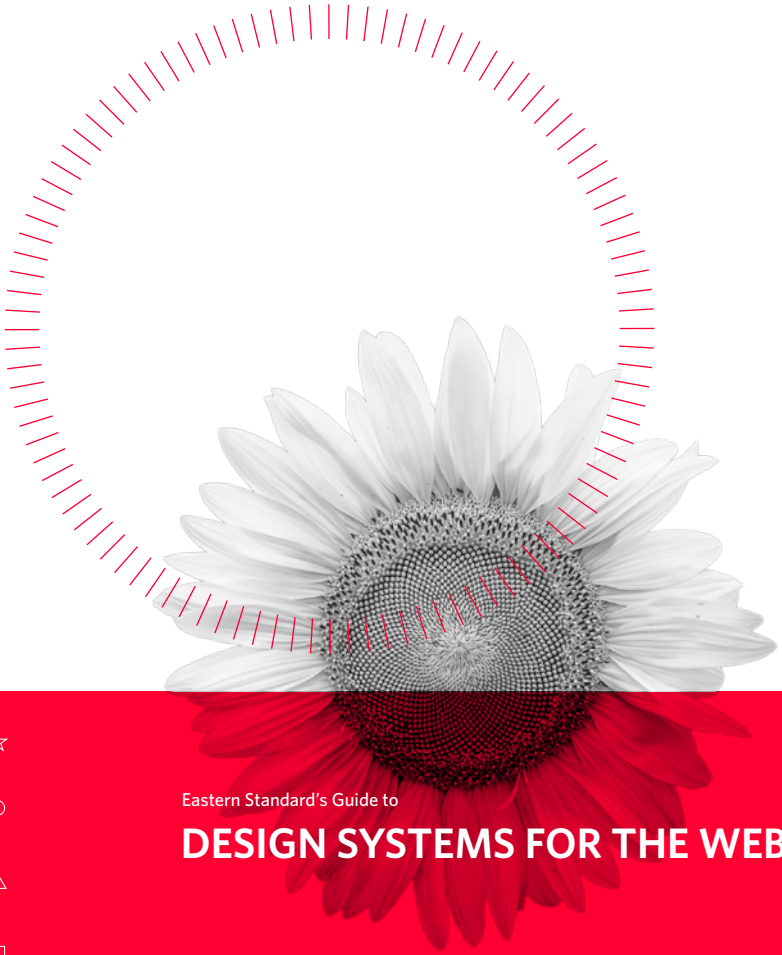
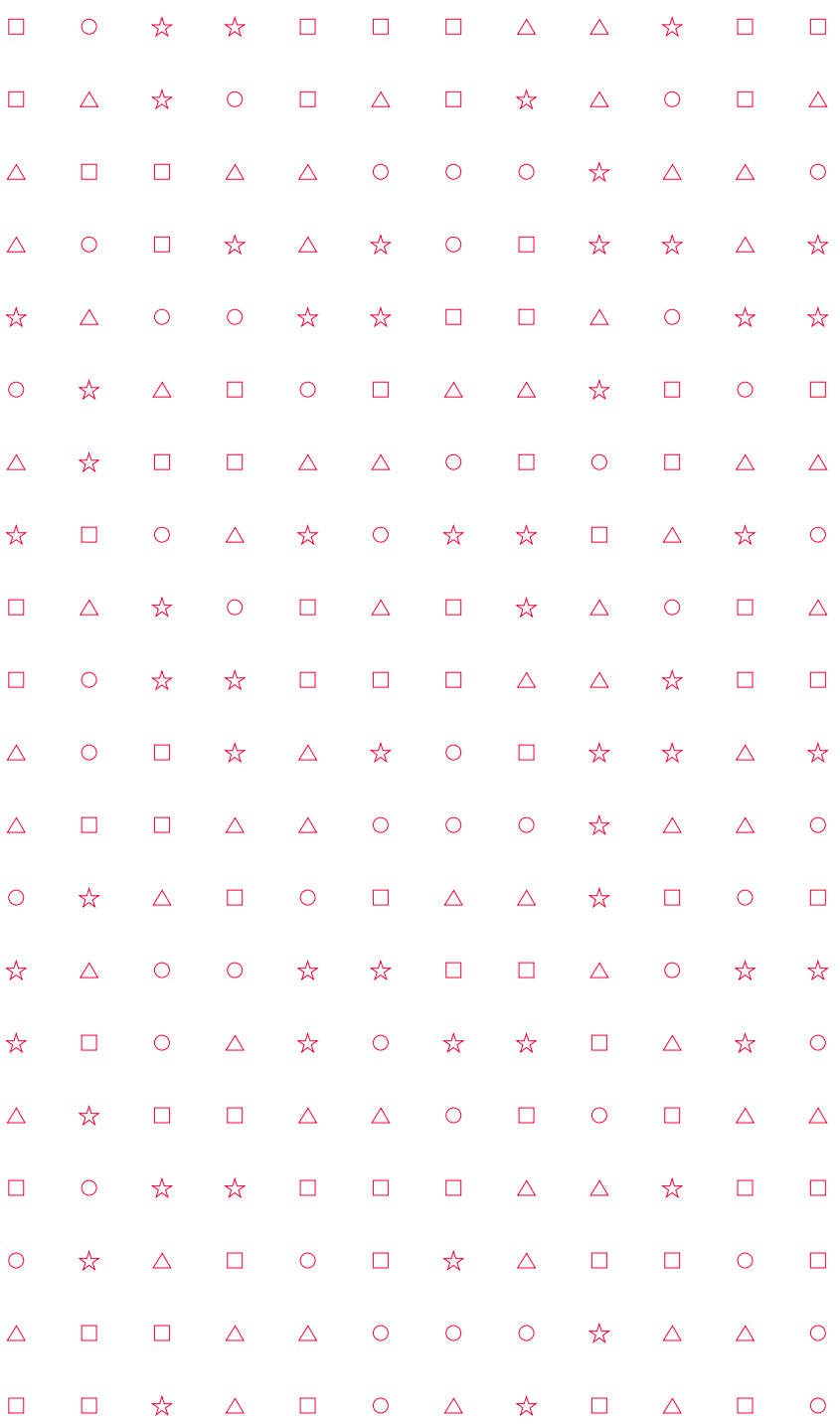


Eastern Standard's Guide to

DESIGN SYSTEMS FOR THE WEB

Jim Keller



Eastern Standard's Guide to

design systems for the web

Jim Keller

Author: Jim Keller

Contributors: Mark Gisi, Brian Huddleston
Editor: Lisa Picozzi
Design: Brian Huddleston

Eastern Standard
1218 Chestnut St., 4th Floor
Philadelphia, PA 19107
easternstandard.com

contents

8–9 introduction

PART 1
10–21 design approach for component systems

PART 2
22–33 deriving components: figuring out what you'll need to design

PART 3
34–41 the end goals of your system + scaling into the future

About This Guide

THIS DOCUMENT REPRESENTS thoughts, experiences, and best practice recommendations for systems-based, scalable website design. We've spent several years refining our approach to the design and implementation of content-rich websites, transitioning away from "templates" and toward modular components that can be combined, re-used, and evolved as needs change. Especially among large institutions or enterprises, single-use websites — e.g., for a particular department or service line — leave clients in a perpetual state of needing a ground-up redesign. When a site is designed using a series of rigid templates, it represents an organizational "snapshot in time" that is often at risk of being outdated by the time the site launches. Without systems-based thinking behind the design and implementation, the common solution is to wait until the next redesign to meet the evolving needs of the organization.

Our clients include a national base of major universities, health systems, and corporations, many of whom have dozens or even hundreds of websites in their ecosystem. We have worked closely with them on the design, implementation, and rollout of web design systems and content management platforms that introduce consistency and governance to their organizations, while still allowing content flexibility and unique aesthetic personality for various departments, schools, or service lines.

One of these clients recently asked us to give their internal design team a presentation about the work that we'd done, and the preparation for that presentation made it clear that we have a lot to say on this topic.

We've organized our thoughts and experiences here in major points and valuable takeaways to help you create modular designs that translate into scalable content management tools.

Before We Begin

Before we get into specifics, it's important to review some reasons why we feel that there's still more to say on the topic

of modular web design, given that it's a common topic of conversation that has been codified in approaches like Brad Frost's *Atomic Design* and Google's Material Design.

Put simply, even among those who are familiar with atomic design, we still see — and were ourselves subject to — mistakes and missed opportunities in how the approach actually gets implemented. A few of these missteps are discussed below.

Visual Consistency vs. Design Systems

Any self-respecting designer will tell you that they are consistent in their design approach: that there's a reason and a rationale behind decisions, and that they stay close to the established brand guidelines and UX best practices.

There is, however, an important distinction between maintaining visual language (the kind of consistency we see most often) and working within a thoughtfully constructed design system. Our hope is that this document will help you understand that distinction, and help you identify areas where you may not be taking the idea of "consistency" far enough.

Atomic Design

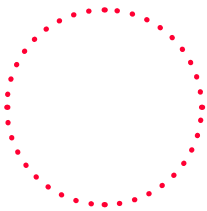
Brad Frost's *Atomic Design*[1] is perhaps the most popular reference point for modular design systems on the web. It discusses many of the same concepts contained in this document, but we hope to add a few helpful, practical layers to help you on your way to executing a modular design system.

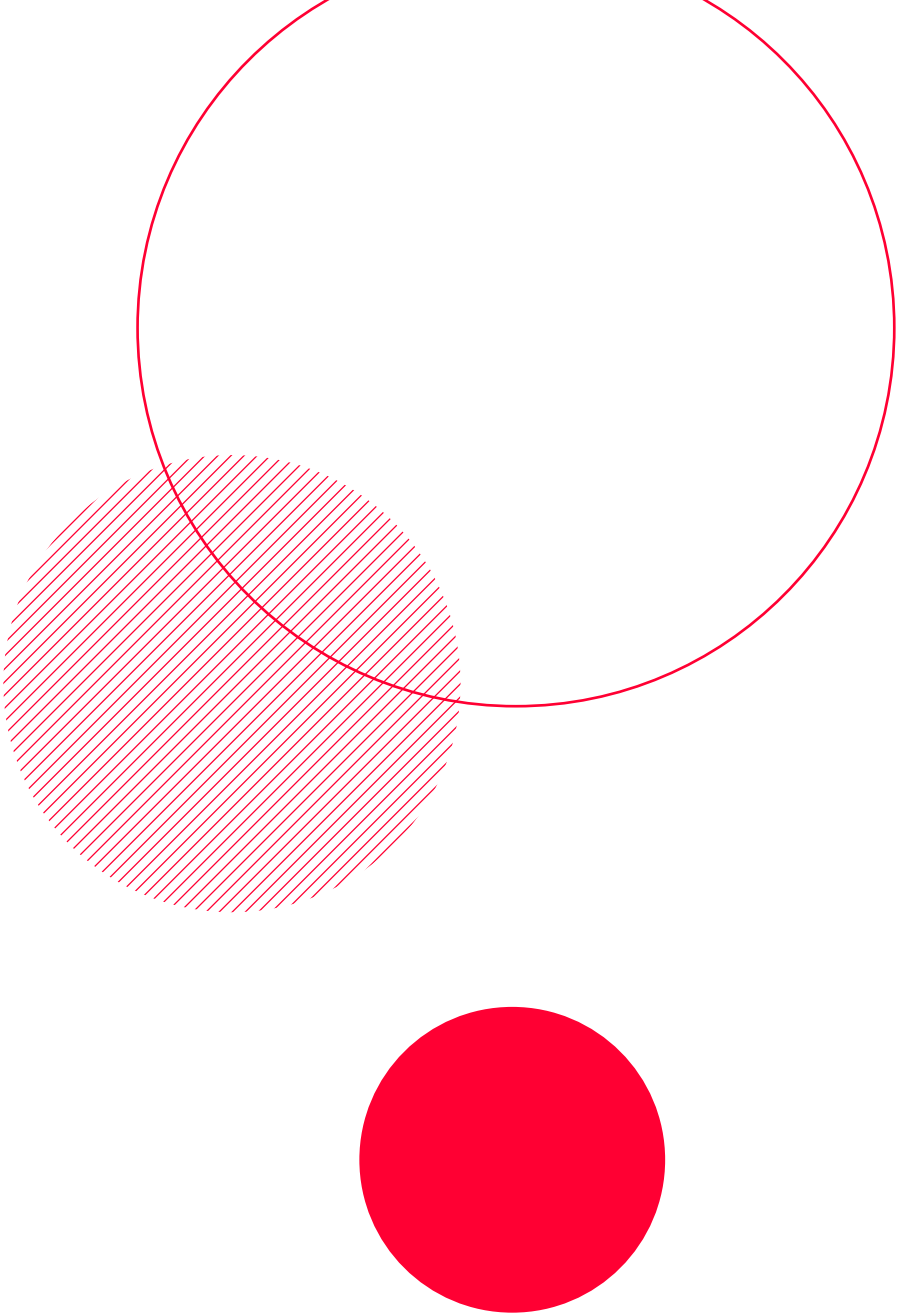
We've also chosen to abandon use of the atomic design nomenclature (atoms, molecules, organisms, etc.) because we've found that it doesn't resonate with clients. We prefer to use the terms "components," "layouts," and "page elements" when educating clients and stakeholders, as you'll see throughout this guide.

1 atomicdesign.bradfrost.com

PART 1

design approach for component systems





THIS SECTION IS INTENDED to highlight the distinction between *maintaining a consistent design language* and *building a truly modular, component-based design.* Understanding this difference is crucial to collaboration between designers and developers.

Developers often find gaps in consistency when they attempt to implement a set of design comps. In other words, a page element that the designer considers to be "the same on both pages" is interpreted and implemented by the developer as two completely different elements. That kind of discrepancy happens throughout implementation, and the problems it creates are felt all the way through to the quality assurance process. You might start finding that, for example, fixing the leading or padding on one page doesn't affect the "same" element on the other page; that other page needs to be fixed separately. Why does this happen?

The answer ultimately has to do with how CSS rules are applied. The developer is under the following constraints when implementing page elements:

- Everything must have a name. If it's on the page and it has style applied to it, the developer had to give it a name in order to apply the styling.
- Every single aspect of the styling needs to be spelled out to the browser. From padding to line height, font size, color, etc., each element is individually addressed in the styling rules.
- If anything within the page element changes from page to page — no matter how minor — that name has to change in some way. If you've got a "callout-block" on page one, and a similar-but-not-exactly-the-same callout block on page two, you just introduced "callout-block-alternate" or "callout-block-1" and "callout-block-2."

If you've been designing or building websites for a while, it may seem that this misalignment on the topic of consistency is just an unfortunate part of the process — that designers and developers just see things differently. In reality, the problem can be eliminated entirely by introducing the right process and using the right language when talking about design.

Naming Conventions

Naming conventions are probably the most important aspect of building a component-based design system. If you stop reading this guide at the end of this paragraph, the one thing you should take away is to assign every component in your design a unique name. Not only will this introduce a common language to your project, but it will immediately highlight when your system is growing out of control. If you've got "call to action style 1," "call to action style 2," "call to action style 2 with big heading," "apply now call to action," and "donation call to action," your system is almost certainly getting away from you. Think about how you can combine the use case into fewer total components. Those components are probably doing almost the same thing, so they can probably share a unified design.

The names for your components should actually be as abstract as possible. In other words, don't call the three top news stories on the homepage "Homepage Featured News." Each word in that name traps your component to an incredibly specific use case, which is exactly what you're trying to steer clear of:

- "Homepage" assumes that this component will always live in only one place on the site.
- "Featured" have you defined what "featured" actually means? Are you later going to find that you have other "featured" content that is designed differently, or content that shares this design but isn't classified as "featured?" In other words, have you made the word "featured" completely meaningless in your system?
- "News" assumes that this component will only ever show news. Can it also show events? Latest blog posts?

These types of naming considerations are why, in Google's Material Design, you'll find component names like "cards," "chips," and "snackbars." The design for these components can be applied to many difference use cases, so they are named as abstractly as possible.

Doing It Wrong: Context Over Function

Below is an example of designing by context, not by function (FIG. 1-3). Note how we have three distinct components (with their own font style, image placement, and sizing) that all serve essentially the same purpose.

The below is an example of why your developer mutters to herself and keeps a flask tucked in her coat. The calls to action below share a common design language, but they are entirely different in the eyes of a developer because they each will require their own unique name and unique set of rules in the CSS code.

FIG. 1

FIG. 2

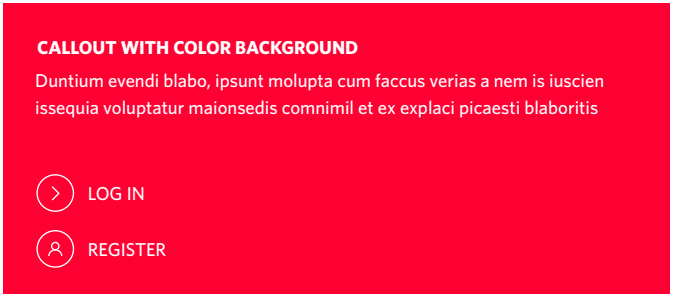


FIG. 3

You can avoid this situation by asking yourself the following three questions as you begin to design a component. I've also noted answers that are applicable to the example above:

1 **What is this thing, fundamentally?**
It's not the "market overview" call to action — that's way too specific and designed for a particular use case. Rather, this component is "an eye-catching block with an image, title, supporting text, and up to two links."

2 **What purpose does it serve?**
It directs users to important content elsewhere on the site.

3 **Is this same need likely to come up later?**
Almost certainly.

If you consistently ask these questions as your design system takes shape, you'll start to notice something very convenient and powerful: You'll see that you've already designed components that share the same answer to "what is this thing, fundamentally?" In that instance, rather than designing something new, you can re-use an existing component or create a slight variant of that component (discussed in the next section) to suit the use case.

If you've got a truly one-off use case that only appears once on the site, and there's good reason for it to only appear once, we refer to this as a "page element." The name is used to distinguish these one-offs from the set of reusable components in the system.

Introducing Variation

One of the most difficult aspects of component design is creating visual interest and excitement while still maintaining rigorous consistency. We don't actually expect that you'll use the exact same teaser or call to action for everything on your site. However, any newly introduced component should have a clear justification for existing. If you've got a use case that could be satisfied by a minor adjustment to an existing component, consider introducing a variant to that component rather than creating an entirely new one.

The examples to the right (FIG. 4-7) show several variations on a call to action. Some have background images, some have supporting text, some are double-wide. The reason these are different from the previous "Doing It Wrong" example is that each variation can inherit a majority of its styles from the base call to action. In other words, the changes are additive; they don't take the same property (text size, padding, etc.) and alter it.

How do you know when something crosses the border from "variation" to "new component?" It's not always clear. If you've answered the three questions from the previous section and still aren't sure, consider the following:

Does the proposed variation potentially upset the original purpose the component was designed for? As an example: Adding or removing the supporting text from our call to action component doesn't upset the original purpose; it still directs users to important content elsewhere on the site.

Conversely, let's say you wanted to use a similar design treatment for a component that simply breaks up the page with an image, title, and text. It might look similar to your call to action, so you might be tempted to call it "call to action without link." However, this is a scenario where you'd want to introduce a new component, as a call to action without a link doesn't satisfy the original purpose at all.

There's no perfect rule for defining the boundaries between variations and components, but the fact that you're thinking about it means that you're probably doing OK!

FIG. 4

FIG. 5

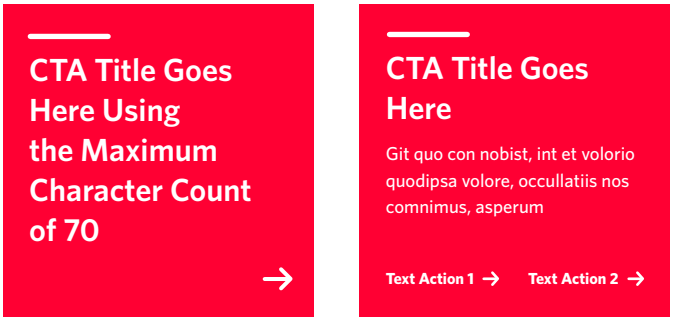
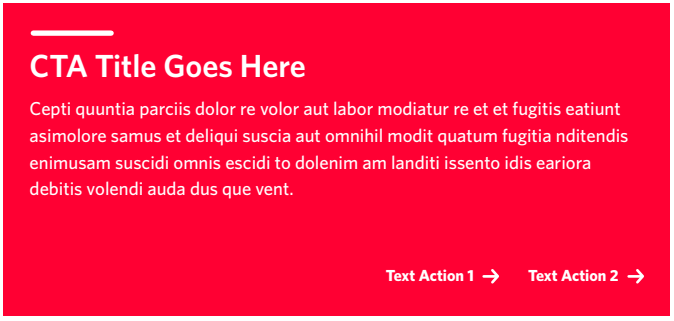


FIG. 6

FIG. 7



Layouts: Putting It All Together

We make it a point to avoid calling our pages "templates" because they're not templated in the traditional sense. The components can be arranged on the page in almost any order, giving the content managers considerable flexibility without allowing poor content decisions to upset the integrity of the design.

Here is one of the most critical things to remember in putting together layouts: Your layouts should be built exclusively from components in your system, with no exceptions.

If you come across a need or use case that isn't supported, don't ever change one particular layout. Revisit your system and add or update your components.

FIG. 8

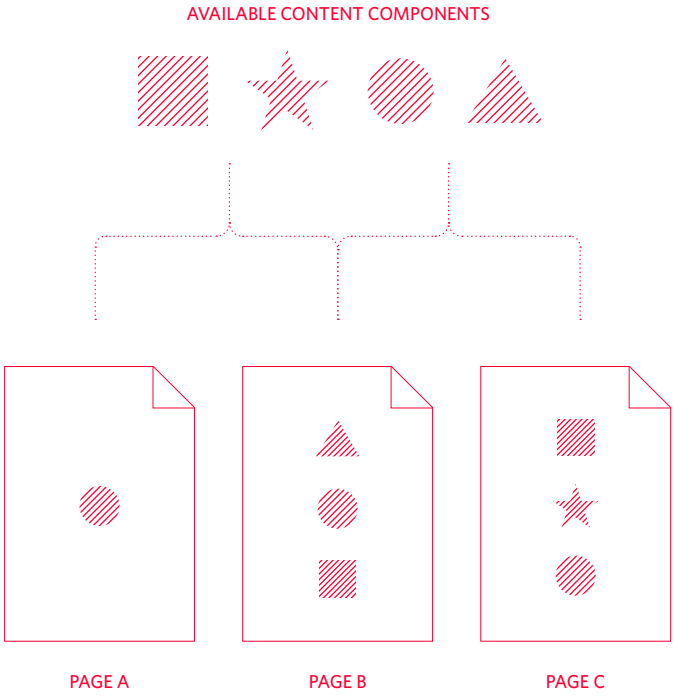
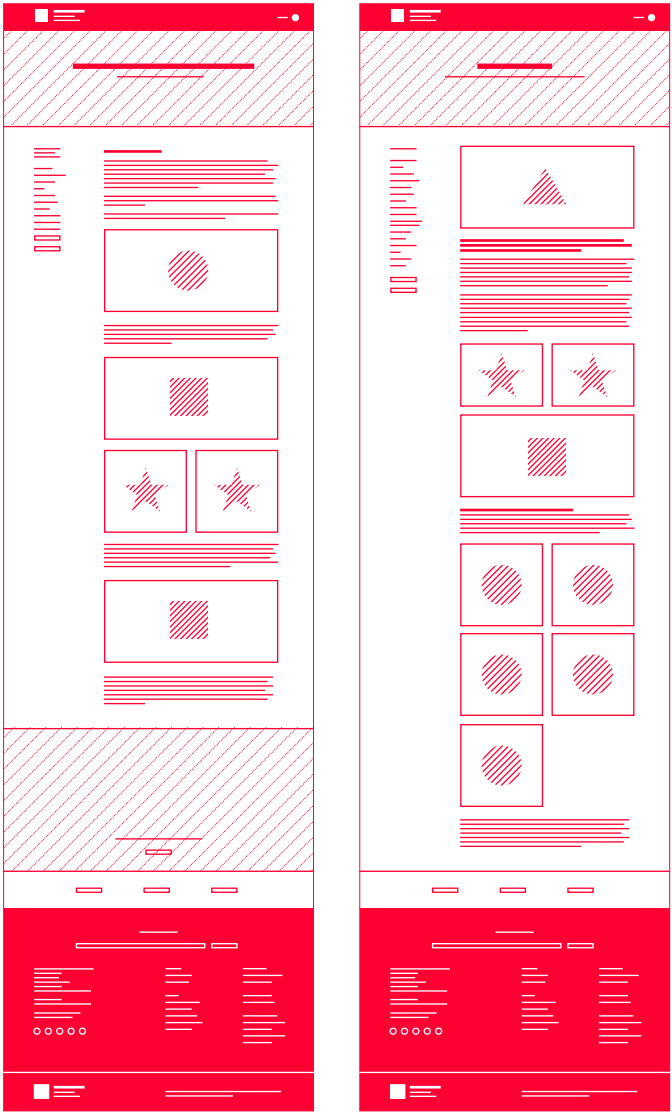


FIG. 9



Component Ordering

One of the biggest challenges you will face in building your layouts is in maintaining the integrity of the design when the order of the components changes (FIG. 8-9). Especially when your components will be built into a content management system, you can't assume that Component A will always come before Component B on the

page. They all have to fit together regardless of order. This is something that you'll need to keep in mind as you combine your components into layouts. Explore different component ordering, and avoid the temptation to make adjustments to just one layout. Also try to avoid creating rules that apply at the layout level (e.g., "this component must always come after that one"); it's preferable to revise the component to make it more flexible.

Maintaining an Aesthetic You Can Be Happy With

One of the most challenging aspects of component-based design is that your mockups may come out feeling bland and blocky. This is a trap that's easy to fall into, and designers often feel limited by systems-based design.

Try not to think of consistency and systems as limitations being imposed on creativity. Instead, focus on embracing them. Remember that working within constraints is nothing new, since it's rare that designers have the luxury of creating a visual language entirely from scratch. The majority of web projects are influenced by existing websites, branding from print collateral, or identity guidelines. In the same way, you need to find clever ways to work within those constraints, you'll need to exercise creativity to find solutions within a design system (even one you've constructed yourself). As you craft your design system, always keep in mind that you should be working from reasons, not rules. The goal is not to sacrifice aesthetics or visual impact for the sake of consistency; it's to achieve a successful balance of both.

When in the beginning phases of constructing a systems-based design, it's critical that you order the steps in your design process in a sensible way. Design pieces for the system in the order of most used to least used. Establish the smallest, most common pieces first, such as body text and buttons, then extend your focus to the rest of the design. This approach allows you to see as early as possible when a basic element needs to be modified or altered to fit within the larger design system, and allows for the minimum amount of rework as you move forward.

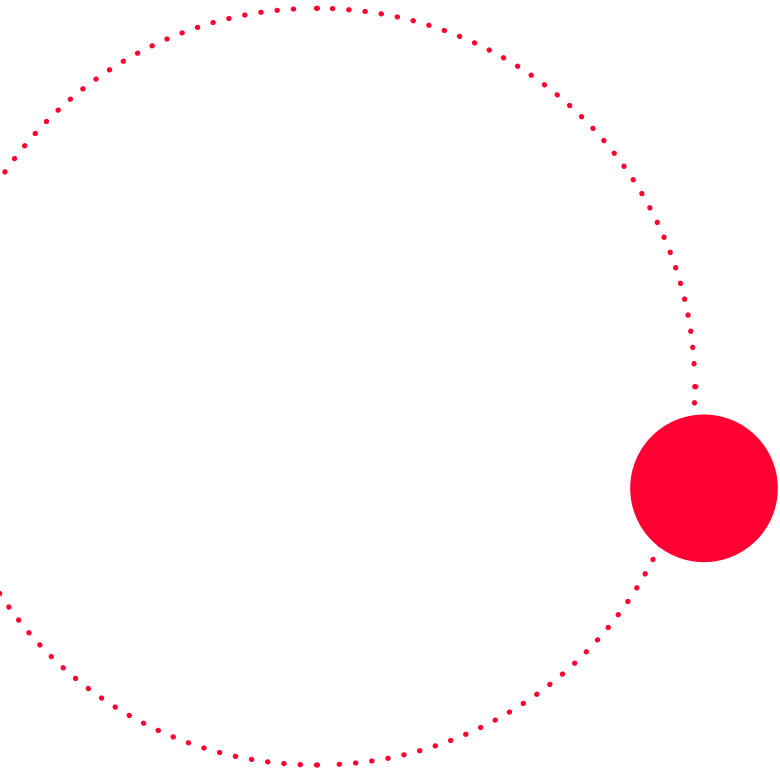
Avoid designing your components in isolation. Although they are often presented this way in style guides, it's easier to design your components within the context of other components they'll frequently be paired with. As discussed previously, it's also very important to consider that these pieces will be rearranged in varied order and in many different combinations. As you're iterating on each component, test them thoroughly by juxtaposing them with others from the system. Just remember not to make layout-exclusive decisions: If you're going to change something to suit a layout, change the underlying components themselves.

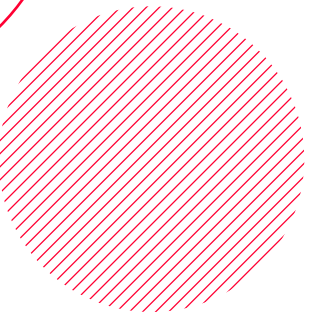
Often, the needs of a large design system are many: informational, promotional, eye-catching, educational, etc. Each of these can be achieved by varying the combination of components used to represent that content. A good system can dial up or dial back the expressiveness of a given page in order to support its purpose. An information-heavy section could be made easier to absorb for the user by breaking apart that page into clearly defined headers and accordions, allowing the user to skim the information at a glance and take in only what they need. Alternatively, a landing page could use components featuring hero images and larger typography to focus on setting a tone for the rest of the site.

A design system for the web is similar to the rhythm when turning the pages of a book: Pieces are revealed gradually while browsing, and each user is seeing these pieces for the first time in different orders and combinations, depending upon where they enter into the experience. Avoid worrying over the individual instances, and instead focus on the system as a whole. Try to capitalize on this rhythm where appropriate. Some components by nature can be much more expressive, whereas others are meant to be more informative, and can be treated as such. The system itself is only as useful as its weakest piece, and each part should be measured and assessed, not only against the other components, but against the overall tone of the system itself. When working correctly, an effective system will support itself. Instead of feeling isolated and weak, the cumulative effect of the components will add up to a whole greater than the sum of its parts.

PART 2

deriving components: figuring out what you'll need to design





THE PREVIOUS CHAPTER DISCUSSED a component-based approach to creating design mockups. Understanding the visual design approach is important for setting a baseline for our discussion, but in practice, the first step in the process — before you begin any visual design — is to understand the content and the interactions that your design will need to represent.

As discussed in the previous chapter, the design of the components themselves must be tied directly to an understanding of purpose — i.e., what any given component is supposed to do — and design decisions must be measured against how effectively they serve that purpose.

This chapter discusses the process of actually defining your requirements as a first step toward establishing your component library. The steps outlined in this chapter are often undertaken by a user experience strategist or information architect working in collaboration with a visual designer.

Yesterday's Design Process

Hopefully the following description sounds unfamiliar to you, or at least like ancient history:

After some brief conversations, a designer would sketch some possible visual design options — often up to three — and present them to the client or stakeholders. The client would respond to the aesthetic of each design, often mixing and matching elements from each, ultimately arriving at a hybrid design that they were satisfied with. Then, only after choosing the visual approach, would the design and/or development teams begin trying to fit the content and functional requirements into a design the client had already approved. Frustrations mounted between and among design, developer, and client, and constant deficits were found and rework was required to make the design actually fit the needs of the content or functionality.

The preceding paragraph describes what the steps in a web design process look like when performed in the wrong order, but this was an extremely common approach before content-first and user-first thinking began to take hold. Hopefully everyone reading this has moved away from the design process I just described, but even if you have, it's important to outline and understand why it was such a poor approach:

1. **Form before function** — It's tempting to start sketching and designing without having built a deep understanding of the functional and content requirements. You might be excited about ideas you have, the visual brand, or imagery, but you need to start at the right place or you'll end up with a "great" design that doesn't actually work. It will probably end up ruining your design, as you're forced to make concessions later in the project. Remember: Wireframes aren't just useful sketches to help you brainstorm. They are a way to separate the visual concerns from the architecture and content priority of the site, allowing you to focus on the fundamental needs of what you're designing, before you begin adding the visual layer.

2. **Making clients happy in the short term and unhappy in the long term** — If you're a good designer, it should be fairly easy for you to design something the client likes from an aesthetic standpoint. Some clients are going to be savvy and detail-oriented enough to poke functional holes in the design, but most will trust that you've considered the needs of the website and give you a thumbs-up without recognizing all of the hidden deficits in what you've presented.

3. **Presenting multiple design options** — Almost never a good idea. You should take your pre-design IA and UX work very seriously, especially if you're performing research, user testing, analytics review, a content audit, and other qualitative and quantitative steps to establish the needs and success metrics for your project. If you present multiple design options just for the sake of presenting three different options, what does that say about your confidence in your requirements gathering?

Do all three options actually reflect equally successful ways of solving the design challenges you identified? You should identify the best, most effective solution to each requirement and craft them all into a system. Iterate internally, but present only your best work.

A Better Way: Designing From Purpose, Needs, and Priority

While a detailed overview of the content/UX discovery process is outside the scope of this guide, the requirements gathering process must leave the visual designer with an understanding of three key points with respect to each component: purpose, needs, and priority. In this section, we'll help you understand how to define and provide these aspects of each component.

If you're familiar with agile software development, there are many similarities between this process and the agile development process: The design requirements are broken into manageable, isolated pieces, articulated at a high level, then tied to a series of more specific requirements.

Establishing Purpose

Before you begin sketching your first wireframe, review and isolate the requirements that have emerged from your discovery process. Do you need to show a teaser for an event? A call to action button? A hero banner? A search interface? For each requirement, ask yourself "what am I trying to accomplish, and what do I need to put on the screen in order to accomplish it?" Put another way: "What problem — from a user's perspective — will this solve?" Continuing to ask yourself these questions will help you keep the purpose of your component at top of mind. You need to be able to justify why it exists in the first place; otherwise, it's just cluttering up your system. Remember: Nothing goes on the screen without a reason. Nothing goes on the screen if you can't explain its complete, end-to-end interaction.

The preceding paragraph ended with "nothing goes on the screen if you can't explain its complete, end-to-end interaction." It's worth highlighting this point and providing an example of what not to do. Remember that design can inherently suggest functionality, and that fact can be dangerous if you haven't established what the functionality actually is. For example, consider a basic search interface, but with a tiny "insignificant" link in the design that says "advanced search," indicating that the user can click to reveal additional options. Depending on the requirements, an "advanced search" option could add days or even weeks of effort to a project. If you've added something to the design for which the end-to-end functionality (which fields it searches, how it displays result counts, etc.) is not entirely vetted, leave it out of your mockups.

As an example, consider the Google.com search interface. This may be the most well-known, purpose-driven interface on the planet.

The purpose of this component is extremely straightforward: As a user, I can type a series of search keywords and be taken to a list of relevant results. (If you're familiar with Agile terminology, the purpose of a component is quite similar to a "user story" for a given feature. Without including much detail, it explains the component's most fundamental reason for existing.)

Note how the purpose, as outlined above, doesn't actually inform visual design. There are many different ways the purpose above could be realized, and that's part of the point. As we'll discuss later, one of the benefits of a design system is that the design itself can evolve without changing the underlying functionality.

If you're unable to clearly and definitively state the purpose of something you're about to add to your design, you should stop to question yourself. Is the purpose unclear, and you need to confirm the requirements you're designing to? Or are you adding elements to the page that don't actually bring value? Remember what Antoine de Saint-Exupéry, author of *The Little Prince*, said of design: "A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."

Establishing Needs

Defining the specific needs of your component is where you begin to establish its functional requirements, and it's also where you begin to indirectly inform design: You are establishing certain criteria and constraints that the designed component must adhere to in order to maintain its purpose. (If you're still thinking in terms of Agile, this would be your "acceptance criteria.")

Following the Google example above, the "needs" overview would look something like this:

- Single-line text area that can support a word or long phrase
- A search button to submit the search form
- An "I'm feeling lucky" button to automatically take a user to a result for their search

Let's actually talk about that last "need," since it may have occurred to you that it is debatable. Do we really need an "I'm feeling lucky" button to suit the needs of this component? Personally, I would argue that "I'm feeling lucky" is superfluous from a UX perspective, and I wouldn't include it in my list of needs. However, needs can come from many places. "I'm feeling lucky" is essentially part of Google's brand at this point, or maybe someone in a very high position really likes the nostalgia factor. Needs can come from many places, which is why it's important to extract them before you begin — and then have to rework — your design.

The search button as a need for this component is also worth talking about. It seems obvious that a search button to submit your keywords would be a requirement, but you may recall that "instant search" was a Google feature at one point; results would begin appearing while you were still typing. That feature eliminated the need to actually click the search button, which changed the needs of the component. Later, the feature was removed[2] because so many searches were happening on

2 searchengineland.com/google-dropped-google-instant-search-279674

mobile, which had "very different input and interaction and screen constraints."

The main takeaway from the discussion about the search button is that needs can evolve, change, and change back. This is one of the reasons why it's so critical to isolate purpose, needs, and priority, and to think in terms of components when approaching your design.

Defining Priority

The final piece of this component puzzle is priority. Priority has to do with the visual importance given to different components within a layout. Following with our Google search example, the priority of that component may be expressed as "Before the user searches, the search component must be the most prominent thing on their screen. After searching, the results themselves are most prominent, but the search component must remain a close second in priority, and must be the most obvious point of interaction outside of the results themselves".

Notice how we're continuing to establish constraints around the design, which, again, should be taken as a good thing. In fact, you should try to think of these priority definitions not as limitations, but rather as benchmarks that will confirm the success of your design solution. Problem-solving is what separates graphic design from fine art, and when designing websites, you should definitely be problem-solving at every stage. Establishing purpose, needs, and priority will give you a clear understanding of which problem you're actually trying to solve.

Purpose and Needs: A Real-World Example

Consider, as an example, the typical "homepage hero" that many designers have come to accept as the anchor tenant of the homepage. You're familiar with this component: A majority of the screen is taken up with a series of large, rotating photos that are (ostensibly) used to drive users to important, timely content on the site.

There's just one problem: If you're trying to drive users to important and timely content on the site, a rotating image carousel is a measurably poor way of accomplishing that goal. Users rarely wait for the images to scroll by, and almost never interact with the images that appear later in the carousel. This overused page component is misguided because the purpose, needs, and priority were never established.

Generally speaking, we've come to avoid designing rotating hero carousels. There are certainly instances where there's value in having a large, establishing photo and short, meaningful messaging. In those instances, we recommend a single photo (non-rotating) and a single statement, usually tied to one or more calls to action. But where possible, we try to deconstruct this trope altogether.

For example, in designing the website for Penn State's Student Affairs Division, it became clear from research and user testing that the website was used as a critical, go-to information repository. Visitors wanted to immediately drill down to specific information, and only rarely were visits introductory or exploratory.

So we thought about the purpose, needs, and priority of the homepage hero area:

The *purpose* of the hero area is to offer visitors a first step toward the information they are looking for. It *needs* to feature clear, distinct links to primary topics and a way to search. Its *priority* is that it should be the most prominent component on the homepage at every screen size.

The site still features a large, tone-setting image in the main hero area, but it is connected to six primary topic buttons that drive users deeper into the site. There was simply no justifiable reason to use this screen real estate for a scrolling banner image because it wouldn't have satisfied the purpose, needs, and priority we were actually designing to.

Dealing With Single-Use Page Elements

There are times when you will be faced with the following reality: There's a need for a component that will only ever exist once throughout your site or application. Because it's not going to repeat elsewhere, it's a true outlier. For example, maybe it's a homepage feature or a map interface that has no compelling reason to exist outside of a particular context.

This situation is sometimes completely valid, and you should go ahead and design a single-use "page element" (which is the terminology we've adopted for these one-offs). Before you resign yourself to a single-use page element, though, make sure that it is truly single use. Is there really no other practical reason it would be used elsewhere? Would a few minor tweaks allow it to merge with another page element and therefore become a reusable component?

Remember, we're thinking in terms of reasons, not rules. There are real-world scenarios for single-use page elements — just make sure you're actually in one of those scenarios before you design a "one-off." Also, you should try to build the page element using other components you've already designed. If it needs a button, don't design a new kind of button if your system already has one. The single-use page element itself should be built from smaller components in your system, insofar as that's possible.

Crafting Layouts From Your Components

While purpose and needs should be defined in isolation on a component-by-component basis, there comes a point where you need to begin thinking about how those components will actually work together in a layout.

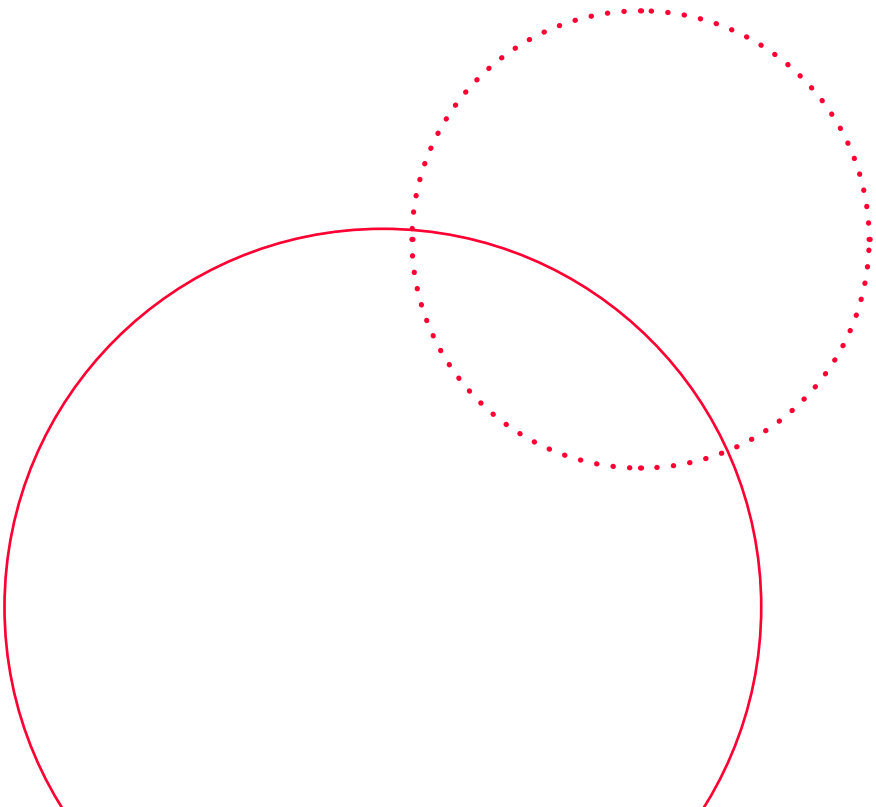
The point in your process when you start applying components to layouts is based on individual preferences and experience. I know designers who prefer to start putting together layouts very early in the process, even while still finalizing their components. This is a reasonable approach as long as they are

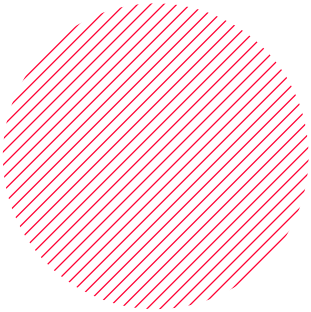
always looking at the design as a system of building blocks, and as long as the design is still in an early, iterative phase.

If you are tackling your first component-based design, you may want to wait until later in the process to begin crafting layouts. You may be inclined to start with layouts because it's comfortable, but you should first create a detailed component guide as an authoritative point of reference. Once you're able to see and think in components, you can work on layouts earlier in the process, iterating over your component design as the layouts come together. The same way a composer can isolate the violins, flutes, and other instruments while listening to a piece of music, the designer should see their design as a series of smaller, purpose-driven pieces, not as a single, indivisible mesh.

PART 3

the end goals of your system + scaling into the future





THERE'S QUITE A BIT OF UP-FRONT WORK INVOLVED in crafting a design system, but your total outlay of effort over the course of a project should be dramatically reduced if you are thoughtful in your decisions and diligent in your documentation.

Ultimately, there are four primary end goals of a component-based system:

Adaptability
As the needs of your site or application evolve, your system should prove to be adaptable to changing requirements. You should find it easy to add new components to the system or to enhance existing ones without the need to deconstruct your design approach from the ground up. Design components should be "hot swappable" in this way: Individual components can be changed without any major shakeup of the overall design. This flexibility allows you and your product owners to evolve the design over time because your system is tolerant to changing requirements. It should be entirely possible to iterate over the design system for a long time, making incremental changes to keep up with updated brand guidelines, new requirements, and other changes that will stress test the objectivity of your approach.

We have a number of enterprise-level clients for whom fast, organizational changes are not always practical. It can be difficult for these types of organizations to keep up with changing technology and customer expectations, and many have felt the pain of being in a constant state of re-designing their website, seemingly with no end in sight. Putting these kinds of clients on a component-based system — something thoughtfully and thoroughly architected with concern for scalability — has allowed them to focus on actually exercising their website as a tool, rather than being forever in "tinkering" mode. We have at least one client who is currently working on the third iteration of their "same" website, though it bears no resemblance to the original. It has been redesigned in manageable pieces, as necessary, to the point where there's

nothing left of the original front end. But there was never a need for a massive re-design process after the initial rollout, because the system was self-governing. The normal problems that creep into a site were kept to a manageable point, so a complete tear-down hasn't been necessary.

Controlled Flexibility/Maintenance of Design Integrity

In the case of websites built on a content management system (CMS), your component system will be used by editors who are likely not well-versed in design. Your goal here is to provide content editors flexibility without giving them the ability to degrade the integrity of the design. The editors should have some control over the order of elements on a page, but they shouldn't be able to make fundamental changes to color, typography, or sizing. The days of providing editors a fully manageable WYSIWYG (what-you-see-is-what-you-get) editor should be considered over at this point. Using a full-featured WYSIWYG editor for content management not only allows editors to apply styles that may be incompatible with the design system, but it goes against principles of semantic, portable content.

Rather than providing a large WYSIWYG field for editors to use, the content management system should allow them to apply various components to a page, filling in the fields for heading, text, image, etc., but without requiring them to do any actual "designing." Basic styles such as bold, italic, underline, and links can be maintained, but giving full control over the HTML would prove fatal to the long-term viability of the content. You need the CMS to hold clean, semantic data with as little markup as possible, so that the content remains completely separate from the presentation. Remember: The presentation layer (i.e., your front-end design) is likely to change, whether through component redesign or a CMS migration. If the content structure and styling are intertwined (as is the case with arbitrarily-styled HTML blobs), you can't strategically alter the front end without also editing the content itself.

When you're implementing a component system for a CMS-based website, a new kind of education is required for content managers. They need to understand the reasons to use one component over another (e.g., where should I use an accordion vs. a tab panel?), and be given some guidance on how components should be used (e.g., don't use three large callout blocks in a row if they weren't designed for that). Your content, UX, and design teams should be prepared to offer this additional level of training when unveiling the content management system.

Usability

Ultimately, designing for the web is about providing an experience to the users who actually use the website or app. Thinking about your design as a system — as an architecture for your user experience — infuses the whole process with a user-first approach that will be reflected in the end product.

If you've done the up-front work to create true consistency in your design, to think through the purpose of each component and establish reasons for your design choices, and to ensure that you're not adding confusing or unnecessary elements to the screen, your users will be immediately better off. You will have intrinsically eliminated as much confusion as possible from your interface, leaving only what's necessary for the experience you wish to create.

A Note About Affordance

In my view, one of the most important aspects of interface design is what's known as "perceived affordance." Coined by Don Norman in his book *Psychology of Everyday Things,* "affordance" refers to the actions that a user perceives to be possible when looking at an object. Applied to UX design, it can be thought of in terms of "does this thing look like it will actually do what I, as the user, expect it to do?" For example, underlined text in a web browser suggests that the text will be clickable as a link.

In a systems approach to design, the affordance of each component and element in your system should be clear to the user. Something that looks clickable should be clickable, and something that looks scrollable should be scrollable, always. You should use established principles of browser or device UI where available*, but it's not always a mortal sin to require users to "learn" something about your interface as long as their gained knowledge is applicable across the entire site or application. An example of what not to do: If you "teach" the user that a right-facing arrow means a link to a new page or section, don't later use a right-facing arrow as a way to expand an accordion component.

Scalable Implementation

Your design system will ultimately be translated into CSS, HTML, and the requisite back-end and behavioral code to make the whole thing come to life. Developers are forced to build their systems within constraints, ranging from unfortunate limitations of a particular platform to the fact that programming is fundamentally based on strict rules and syntax.

One of the most common reasons that the front-end implementation of a website or app begins to become burdensome is that the programmer is forced to work within constraints that the designer wasn't mindful of. As a result, the total amount of code — and its complexity — increases as the programmer accounts for inconsistencies in the design, or the constant appearance of "one-offs," where special code has to be written to deal with a use case that only appears in one place.

The real-world implications of a poor or nonexistent design system most often shake out as seemingly infinite and ongoing problems with the implementation. Brittleness ("I fixed one thing and something else broke") is widespread, as

* In the Flash days, the most egregious example of NOT using established browser UI could be found in applications that designed and implemented their own scrollbars, often circumventing browser scrolling altogether. You should have extremely good reasons if you're going to uproot the expected UI in this manner.

are regression issues ("this thing was right before, but now it's not"). Perhaps the scariest problem is "this is fixed on one page but not on another," which indicates that the code has been implemented on a very granular basis, preventing global changes and fixes. This means there's been a complete breakdown of the systems-based approach, and you should prepare for a real grind in quality assurance.

As a developer works on a site with no underlying system, they're forced to write more and more single-purpose code, and the logic begins to branch in many directions. That's why you end up with the kinds of problems outlined in the previous paragraph. Beginning the project with a thoughtful, systems-based approach and using consistent nomenclature can prevent nearly all of these problems.

in short...

1. Establish the purpose of each component in its most fundamental form.
2. Identify specific needs for the component that are necessary for it to suit that purpose.
3. Design your components around their function, not their context — define it by what it does for the user (e.g., a callout block that directs users to another page) rather than where it happens to be utilized (e.g., "this is the featured story callout block").
4. Think globally about how the decisions you make in design will apply across the website or application, rather than designing to an individual use case.
5. Never put anything on the screen unless you can explain its purpose and detail what happens when a user interacts with it.
6. Build layouts exclusively from components in your system; never change an individual layout to incorporate something outside of the existing system. Instead, update or enhance your component library to accommodate this new use case.

your design approach should provide for

1. Long-term viability and adaptability of the system, which does not have to be re-thought as requirements change.
2. Flexibility for content managers without compromising the integrity of the design.
3. Improved user experience through consistency and clear affordance.
4. Improved experience in implementation: faster development with fewer ongoing quality assurance issues.

Above: The entryway to our Center City Philadelphia office

our company

EASTERN STANDARD is a branding and web agency created by merging a branding studio and web applications developer in order to meet our clients' demand for a truly integrated creative and digital agency. We are more than a group of strategists, designers, and developers. We understand the keys to good branding. We offer real-world experience in meeting business objectives, and we know how to leverage and present your branded content in a manner that accomplishes those objectives. We have the technical expertise to face complicated challenges or advanced functional requirements head on. Together with your team, we develop award-winning experiences through winning strategies, smart design, and modern products.

Eastern Standard — The Meaning Behind the Name
Our company name, Eastern Standard, is a reference not only to our East Coast roots, but also a nod to the Eastern philosophies of simplicity, harmony, and balance. It conveys a sense of longevity and the commitment to creativity and innovation we're known for by our clients and peers.

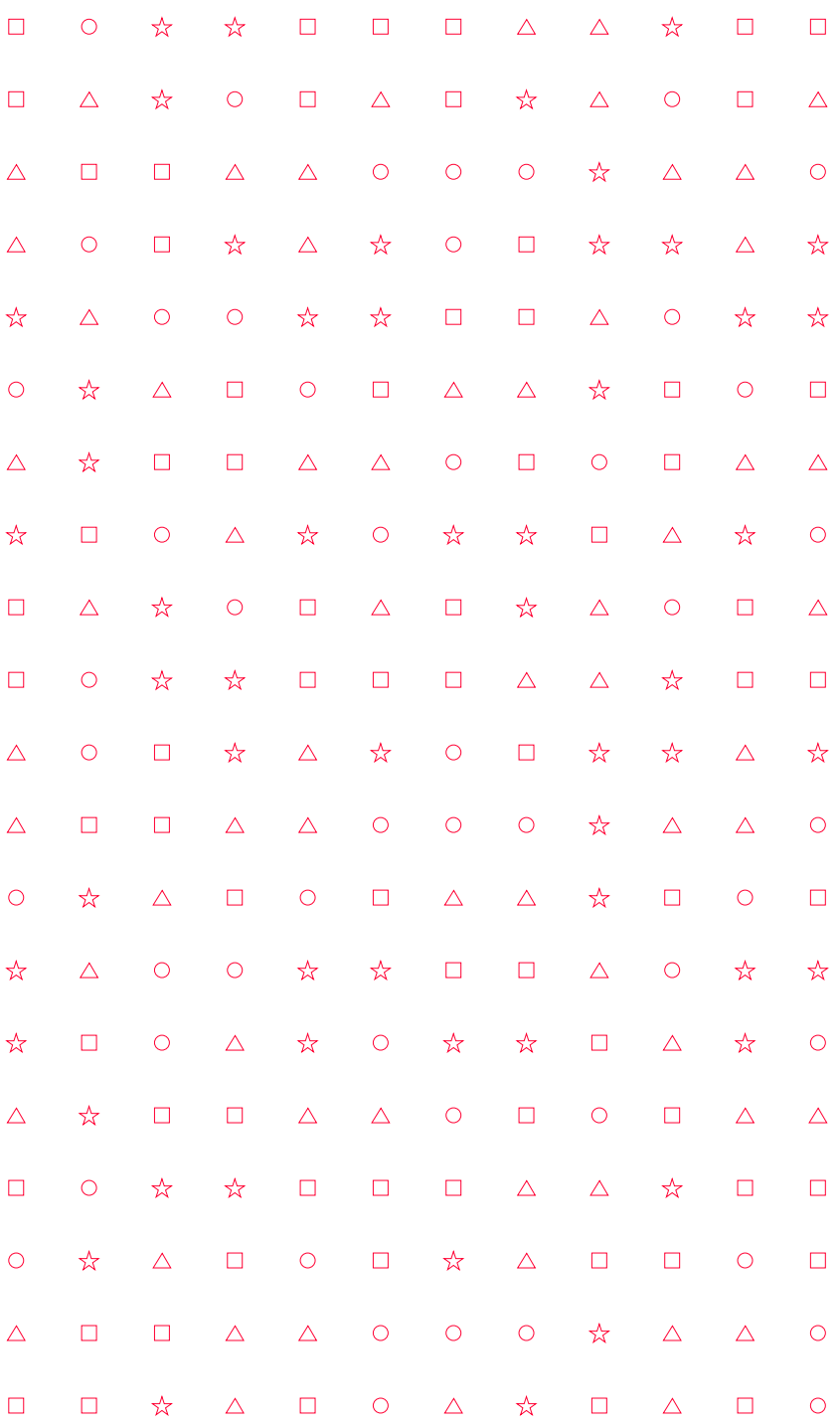
Find out more at easternstandard.com

about the author

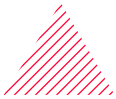
Jim Keller
Founding Partner & Technology Director at Eastern Standard

JIM KELLER IS ONE OF THE founding partners of Eastern Standard, a combined branding and web agency headquartered in Philadelphia. He has worked in nearly all aspects of web technology over his 20-year career, from extensive front-end coding to network operations support at Philadelphia's first Internet provider.

Jim's experience in UX, research, and design, and his focus on technology implementation guides our team in the creation of innovative, large-scale web projects.



notes



ESTRN
STANDARD

Eastern Standard
1218 Chestnut St., 4th Floor
Philadelphia, PA 19107
easternstandard.com