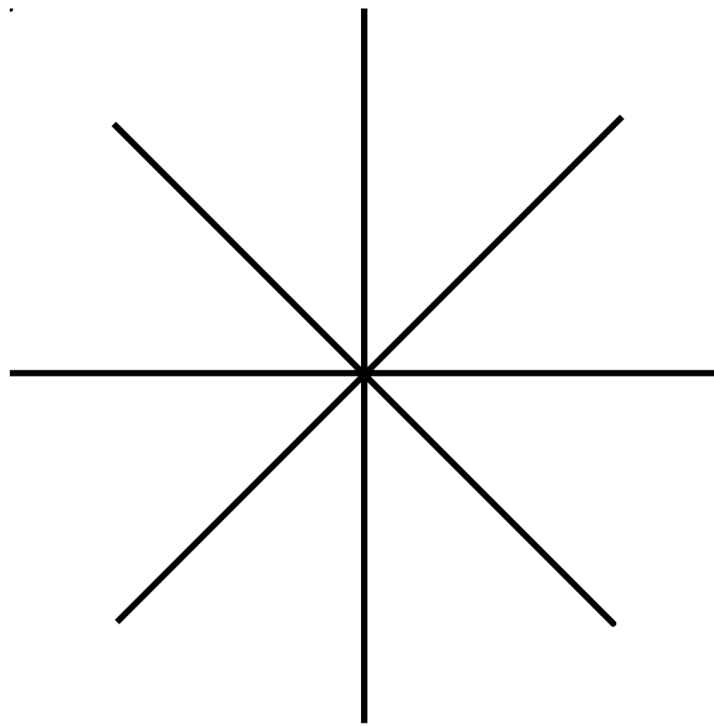


On Design Systems

Theory, analysis and practice



Gregory

Introduction

In recent years large scale, multiplatform, highly interactive applications changed the design and development landscape. Design systems are part of this technological and cultural¹ paradigm shift and they are here to stay in the foreseeable future.

While new design systems and component libraries pop up every week this discipline is still young enough to lack comprehensive literature. Practice is ahead of theory in this particular field.

At the moment of writing this guide Amazon lists four books on design systems. None of them aims for a big picture built on software development theory and on thorough competitive analysis.

To fill this gap, *On Design Systems* presents a theoretical foundation based on existing software design patterns, programming paradigms and principles. It analyses current implementations to reveal common knowledge then puts theory and analysis in practice by implementing an example design system.

The findings should empower non-technical audiences with a framework to rely on when managing the creation of design systems.

For tech leads and project managers it offers more: a complete theory, an analytical framework for further exploration, and a working example with source code open for betterment and customisation.

About the author

Csongor Bartus has a degree in Computer Science. By interest, he is a self-taught designer having works featured in online galleries². He created his first design system in 2015. Since then he works at the intersection of design and code.

Version

v1.0.0 - January 2021

Copyright

© 2021 Bartus Csongor. All rights reserved.

¹ [Design systems are a cultural challenge — Beat](#)

² [Metamn](#)

Introduction	2
About the author	2
Version	2
Copyright	2
History and context	5
Prior art	5
The component model	5
Design systems vs. component libraries	6
A big picture	6
Theory	8
Minimal API Surface Area	8
Functional Composition	9
Tokens	10
Single Source of Truth	10
Theme Specification	10
Type System	11
Components	11
The Base / Variant Pattern	12
Single-responsibility Principle	13
Rule of Three	13
Props	14
The Open / Closed Principle	14
The Deno Style Guide	15
A big picture	16
Analysis	17
Workflow	17
Purpose and audience	17
Features	18
Deliverables	19
Code	19
Documentation	20
Design guidelines	21
Live playground	21
Example applications	22
Tests	22
Packaging	22
A big picture	24
Practice	25

Edo—An example design system for marketing websites	25
Glossary	26
Resources	28
Feedback	28

History and context

Design systems have been around since 2013-2014.

Everything started with Lonely Planet's Rizzo in 2013 then followed by Google's Material Design in 2014 when the concept hit the mainstream.

2020 was the year when design systems popped up every week. Adele³ UXPin lists over a hundred systems and libraries—yet the list is incomplete.

Prior art

Before design systems and component libraries, templates and CSS frameworks were the standard ways creating web sites and applications.

This prior art lets developers prepare styles and animations and assign them to HTML elements via a primitive mechanism: using class names, ids or data attributes.

Coupling structure (HTML), presentation (CSS) and behaviour (Javascript) with such a fragile construct never scaled⁴ well. Deliverables become large affecting page performance.

Aside performance problems, template-based UI frameworks—of Rails, Wordpress, Laravel fame—and CSS frameworks—utility / atomic / functional CSS libraries like Tailwind, Bootstrap, Bulma, Tachyons and dozens more—led to unmanageable⁵ source code in the long run.

A characteristic of this prior methodology is to let developers add an arbitrary number of styles to any HTML element. When styling is such a way open-ended consistency and design uniformity is lost, or hard to maintain over time. As the number of styles added to class names increases the maintainability of the source code decreases.

The component model

Google and Facebook, by their scale, seized first the need for a paradigm shift. Their response was Polymer, Angular, Material Design, and React—respectively.

Component-based UI frameworks represented by React, Vue, Svelte merge HTML, CSS and Javascript into standalone units—components—and find ways to scale them up.

This novel separation of concerns shifts focus from individual building blocks—structure, presentation, behaviour—to a single component encapsulating them all. Scaling now has a single focus versus three.

³ [Adele – Design Systems and Pattern Libraries Repository](#)

⁴ [The problems of CSS at scale](#)

⁵ [jxnblk.com Two Steps Forward. One Step Back](#)

To assure the source code is maintainable on long term the new methodology promotes a constraints-based approach when styling HTML elements. It limits the possibilities for adding new styles to a well defined level, the token layer.

Design systems vs. component libraries

Design systems and component libraries represent the component model. While similar in most aspects they differ in developer experience.

The distinction is well articulated by Mark Dalgleish, creator of the Braid design system.

The difference between a component library and a design system is whether or not your components have 'className' and 'style' props — Mark Dalgleish⁶

Or, to use the terms coined by Brent Jackson⁷, a pioneer in the field, design systems embrace the constraints-based approach while component libraries are open-ended.

Design systems are strict. They form a complete, closed system. They don't allow on-the-fly customisation. They are more expensive to create, and easier to use later.

Component libraries are loose. They offer the basics and let customisation happen at any point, any time.

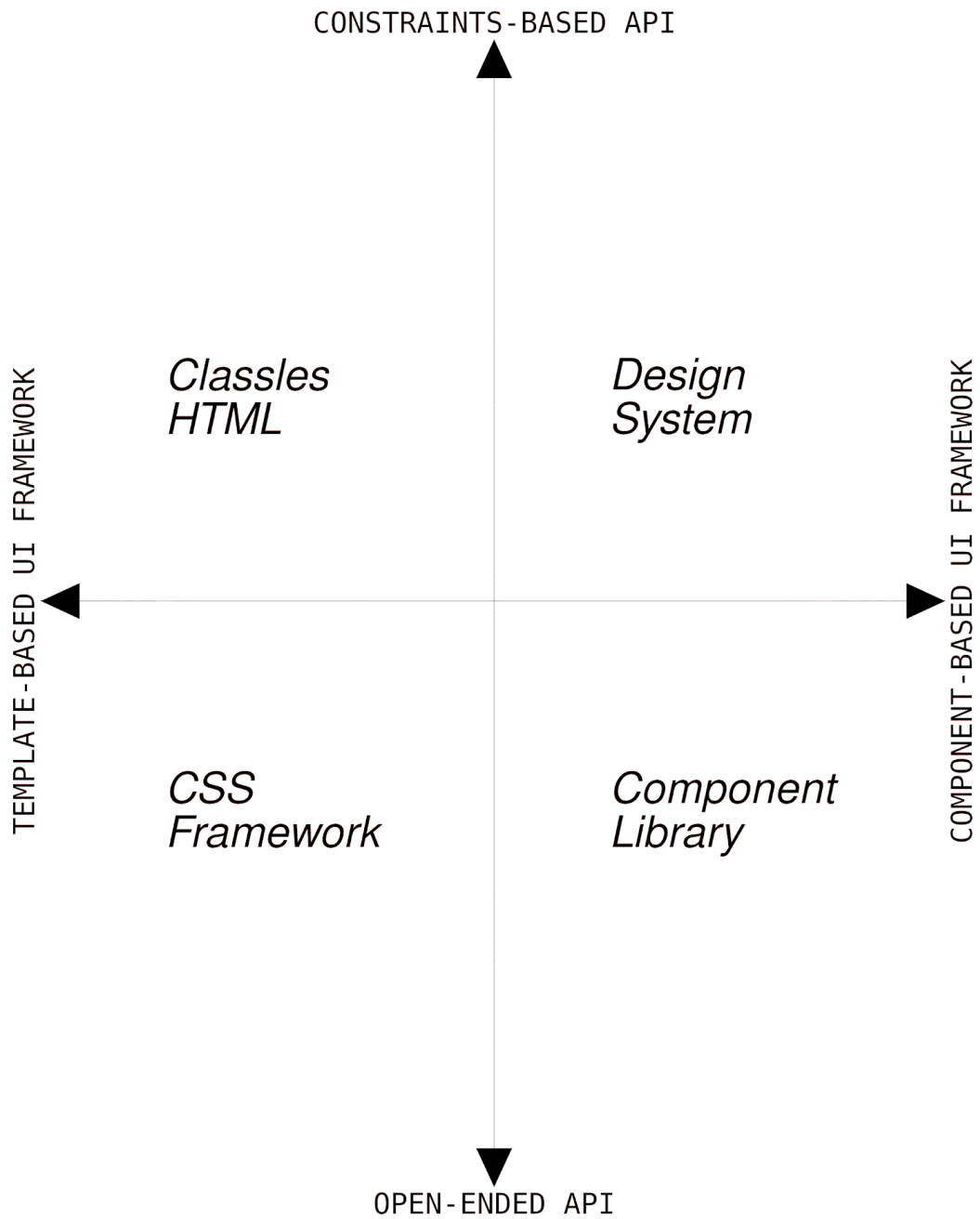
A big picture

The prior art and the new paradigm both fit on a simple map. After all, they represent solutions for the same problem—performance and scalability—from different angles.

Time hasn't yet decided which approach works better. The latter, in contrast, has enterprise support unseen before on this scene.

⁶ [Mark Dalgleish](#)

⁷ [Jxnblk](#)



Theory

It's good to build long-lasting software on theory. It drives design decisions and makes the result scalable and interoperable.

Design systems are long-lasting software and young enough to have no theoretical model defined yet. A single attempt⁸ exists to standardize tokens and there is a consensus, a common practice on workflow and deliverables. Nothing more.

To create a complete theory, existing software design patterns and the pioneering groundwork done in the field is to rely on.

The current work of Tae Kim, Maja Wichrowska⁹, Mark Dalgleish and Brent Jackson define a theoretical foundation for design systems. They identify the building blocks—tokens, components and props—and pinpoint where standardization is missing: in naming, structuring (scope and hierarchy) and composing.

To render existing software design patterns useful, design systems first have to find their place on the software development map. When defined as software with a public interface (API) they do become part of this family.

APIs are well known artifacts where standard practice—design patterns, programming paradigms and principles—apply.

Tokens, components and their attributes—props—together are enough to define an API for a design system. Leaving theory making to apply general API theory to this new context.

Minimal API Surface Area

The API Design Patterns¹⁰ book defines the main characteristics of an API as operational, expressive, simple and predictable.

Facebook, the creator of React gives hints on how to achieve them when building APIs.

React is moving towards a minimal API surface area. Instead of providing many framework features, React is trying to utilize patterns, paradigms and JavaScript language features to accomplish the same tasks that other frameworks have dedicated APIs for — Sebastian Markbage: Minimal API Surface Area | JSConf EU 2014¹¹

⁸ [System UI](#)

⁹ [Building \(and Re-Building\) the Airbnb Design System Maja Wichrowska](#)

¹⁰ [Manning | API Design Patterns](#)

¹¹ [Sebastian Markbage: Minimal API Surface Area | JSConf EU 2014](#)

The strategy to reach simplicity and predictability is to rely on existing work instead of adding new abstractions. No abstraction is better than wrong abstraction.

History shows this approach is right. The attempt to organize web-specific code around abstract concepts following a set of rigid rules failed spectacularly. OOCSS, SMACSS, Atomic CSS and co, once a silver bullet, are all gone now¹².

Forcing external concepts—from biology, promoted by Atomic CSS—on web development doesn't scale. In web development the abstraction is already present and it's natural.

Typography, colors, navigation, layout, form, card, hero are common, well understood concepts. They don't require further abstraction. UI Guideline's component standardization attempt¹³ lead to the same results as the simple, common sense, natural naming conventions put together by CSS Layout¹⁴.

Summing up, naming and structuring in design systems should follow the Minimal API Surface Area principle promoting re-use of existing practices and avoiding further abstractions.

Functional Composition

Composition again should follow a pattern. When the pattern comes from the same theoretical line as naming and structuring a consistent theory forms.

According to Facebook and React¹⁵, composing a system from smaller parts is best possible when the underlying components behave predictably. They provide clear and stable interfaces—input parameters and return values.

This approach is specific to functional programming, a paradigm replacing imperative programming as the component model replaces templates and CSS frameworks.

Both paradigms solve the same problem—to build modular, interactive applications—in different ways. The imperative way implements modularity through tight-coupling and interaction synchronously. The functional way implements modularity with loose-coupling and interaction asynchronously.

Applying functional composition to design systems completes the theoretical framework. The result is consistent and backed by a large company and today's most popular component-based UI framework.

¹² [Google Trends on CSS methodologies](#)

¹³ [UI Guideline - Component Standardization](#)

¹⁴ [CSS Layout](#)

¹⁵ [The reactive, functional nature of React](#)

While a general theoretical framework exists, a good practice is to apply the theory on every specific building block. Beside validating the theory—new, block-level design patterns might emerge.

Tokens

What we know about tokens is that they represent the layer where all settings of a component-based UI framework go. And there is a standardization attempt for making them interoperable.

Tokens are the first approach collecting settings into a common place. Prior art spread them across the parts of the system requiring a single modification to involve multiple places and files. This took a cognitive toll on developer attention and led to reduced experience.

Single Source of Truth

In information systems design and theory, single source of truth (SSOT) is the practice of structuring information models and associated data schema such that every data element is mastered (or edited) in only one place — Wikipedia¹⁶

By definition tokens follow this pattern. They structure settings into a separate layer. Change takes place in this single location and spreads automatically across the other parts of the system.

In design systems this layer is well-guarded, accessible through constraints. Component libraries, in contrast, allow a more liberal usage of tokens following the open-ended principle.

Theme Specification

Theme Specification¹⁷, an open source organization and initiative, is the attempt to standardize tokens to enable interoperable UI components.

Github, Artsy, Gatsby and others embrace this theory and build their design systems upon.

The specification is not yet complete—it lacks full documentation. It doesn't fully follow existing API naming and structuring conventions—The Minimal API Surface Area principle. And it's highly opinionated. Its purpose is tied to a specific company's specific needs.

For naming and structuring Theme Specification can serve as inspiration not as a recipe to follow.

¹⁶ [Single source of truth](#)

¹⁷ [Theme Spec – Theme UI](#)

Further, it turns out the specification doesn't fully support arbitrary responsive styles. Fonts are responsive, headings sizes are not. This is due to the recursive composition pattern they use to compose tokens.

Functional composition turns out to be a better choice. By nature it is open to changes and variations while recursion requires updates to the algorithm when edge cases occur.

Type System

What Theme Specification lacks most is a type system.

The main purpose of a type system is to reduce possibilities for bugs in computer programs by defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way — Wikipedia¹⁸

The root of the problem is their language of choice, Javascript, a dynamically typed language without type system support. This leads to bugs—otherwise discoverable during development time with a statically typed language—to slip into production.

If Typescript, a similar language to Javascript with type system support, were the language of choice, Theme Specification could be enhanced—the naming and composition bugs fixed—to serve as a standard for tokens.

Components

Components, the basic artifacts of the component model, merge structure (HTML), presentation (CSS) and behaviour (Javascript) into a standalone unit. This novel separation of concerns reduces the scalability issues to a single item, raising the possibility of a better performance.

In design systems components define the layer above tokens. It contains visual and functional elements like navigation, buttons, forms, articles, carousels and more.

While the number of tokens is finite, the number of components is ever expanding. A design system may contain hundreds of components and their variations. This leads to difficulties in naming, structuring and composing them.

On large scale locating, finding components in the source code, extending them takes a toll on developer attention. To reduce cognitive load, lift developer experience, and make or break an API—good naming conventions are to rely on.

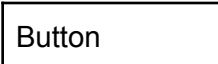



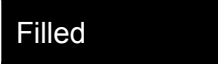















¹⁸ [Type system](#)

The Base / Variant Pattern

Naming high level components—header, button, menu, list—is easy. Consensus is always possible.

Over time high level components tend to decompose into smaller parts and different states.

A header decomposes into logo, title, subtitle and description. A button might be simple or filled and should reflect states like green—primary, blue—secondary or grey—disabled. A button can or cannot have an icon associated.

Button	Default	Disabled	Primary	Secondary
Simple				
Filled				
As link				
With icon				
Just icon				

In this context difficulties arise and an easy-to-use naming convention is necessary.

Finding such a naming convention took four years for Airbnb¹⁹. Their solution is the Base / Variant pattern (B/V)²⁰ which resembles a similar methodology used in CSS frameworks called Block-Element-Modifier (BEM)²¹. The common root of these patterns is the Template method pattern²², one of the classic design patterns in software development.

¹⁹ [Building \(and Re-Building\) the Airbnb Design System Maja Wichrowska](#)

²⁰ [Design Systems: React Buttons with the Base + Variant Pattern](#)

²¹ [CSS / BEM / Methodology](#)

²² [Template method pattern](#)

The Airbnb example underlines the necessity of a complete design systems theory.

Single-responsibility Principle

Applying the Base / Variant pattern for buttons yields the following structure:

```
# Base (in B/V)
# - Block (in BEM)
Button
# Variants (in B/V)
# - Modifiers (in BEM)
ButtonFilled
ButtonPrimary
ButtonSecondary
ButtonDisabled
# - Elements (in BEM)
/Icon
/Text
```

The pattern advocates a small base and simple rules for expansion. Defines a scalable naming convention encouraging splitting components into small parts.

In a system where decomposition results in independent, self-contained components each with a unique responsibility the Single-responsibility Principle²³ kicks in.

Part of SOLID²⁴—a set of well-known design principles—the Single-responsibility Principle is a necessary and sufficient condition for good code.

It makes sure code is extensible, maintainable, scalable and composable while offering an enjoyable developer experience.

When code segments are unique, self-contained modifications and extensions happen in a singular well defined place—nowhere else—reducing developer cognitive load and speeding up application development time.

Independent, atomic components display a modest interface. The less their inner workings, the less arguments and props to influence these workings, the better they compose up. Composition scaling is about simple, predictable interfaces.

²³ [Single-responsibility principle](#)

²⁴ [SOLID](#)

Rule of Three

As a downside, the Single-responsibility Principle tends to increase the number of components which leads to increased cost of maintenance.

To the rescue, the Rule of Three²⁵ pattern balances this act. It allows refactoring—applying the Single-responsibility Principle—at the third strike, ignoring the first two attempts, needs to create sub-components and folders.

After all patterns applied, the button structure becomes:

```
# Base
# - Includes Icon and Text according to the Rule of Three
Button
# Variants
# - Variations go into independent components according to the Rule of
Three
ButtonFilled
ButtonPrimary
ButtonSecondary
ButtonDisabled
```

Props

Props represent the attributes of a design system. Red for colors, Helvetica for fonts, centered for layout.

They span tokens and components without defining another layer. Every token and component has props and when they sum up they lead to naming, structuring and composing problems.

For naming props there is no standard design pattern. Applying the Minimal API Surface Area principle helps by forcing prop names to match existing API (HTML, CSS, JavaScript) naming conventions.

When CSS provides the `font-family` prop design tokens should adapt it versus coming up with a new term like `font-name`. The same is true for Javascript, which provides an `onclick` event handler to drive components to adopt it.

²⁵ [Rule of three \(computer programming\)](#)

The Open / Closed Principle

Structuring props makes or breaks component and token composition. Short list of props scales better than a long list—as seen in the open-ended versus constraints-based API approach, or at the Single-responsibility Principle.

Fewer props represent fewer coupling points when parts compose up. When a part changes fewer props update faster. Known as loose coupling²⁶, this principle makes code extensible, and extensibility leads to maintainability.

This concept of being able to extend the application without modifying existing code is called the Open/Closed Principle. It is impossible to get to a situation where 100% of your code will always be open for extensibility, but closed for modification. Still, with loose coupling we get closer, and it gets easier to add new features and requirements to our system — Wikipedia²⁷

The problem of structuring props doesn't stop here.

The Deno Style Guide

In this particular context—how to structure props—software development theory offers no exact recipe on implementation. It took until recent days to surface quantitative rules for the Open / Closed principle.

Deno, in the same way a cornerstone in web technology as Facebook's React, in its Style Guide²⁸ promotes the idea of functions (tokens and components) to have max two props. The rest, if any, should go into an options object.

Reducing the number of props to max three offers a promise for composition scalability.

²⁶ [Writing Maintainable, Loosely-Coupled Code - Manning](#)

²⁷ [Open-closed principle](#)

²⁸ [The Deno Manual](#)

A big picture

When perceived as software with a public interface, design systems become equipped with a complete theory.

The API aspect defines the general guidelines—Minimal API Surface Area, Functional compositions—and sticks each layer to it. Later layers, individually, can define their own design patterns.

	Naming	Structuring	Composing
API	Minimal API Surface Area		Functional Composition
Tokens	Single Source of Truth		
	Type System		
Components	Base / Variant Pattern		
	Single-responsibility Principle		
	Rule of Three		
Props		The Open / Closed Principle	
		Deno Style Guide	

Analysis

While theory shows how to write and organize code it tells nothing about the workflow, the deliverables, the usability, the technology stack of a design system, and the common practices behind.

Analysing existing systems, studying, understanding, classifying and comparing them reveals this knowledge. Exploring the ways how others roll their own shows the limits and possibilities of this new software medium and the surrounding ecosystem.

The analytical framework²⁹ underpinning this guide employs over 50 aspects and criterias to understand the big picture. It's open ended, ready to accept new, interesting libraries anytime they pop up.

The framework includes design systems and component libraries built on React. The filter is necessary to reduce the large number of available systems to manageable dozens. This bound to a specific technology narrows down the findings on the tech stack but leaves the other aspects—workflow, deliverables, usability, common practices—intact.

Workflow

The process of creating similar systems defines a workflow. A procedure with steps and outcomes often following a standard.

Analysis shows in design systems the workflow is standard. Creating a new system always starts with defining purpose and audience then features.

Purpose and audience

Purpose and audience define why a company builds a design system. Reasons vary but resemble a pattern:

- Internal: To build apps for their brand.
- External: To build general purpose apps.
- Foundational: To help build other design systems and component libraries.

Majority builds internal systems—no wonder—design systems shine when used to create uniform looking products across a portfolio.

²⁹ [Design Systems Analytical Framework](#)

A handful of companies create general purpose systems. More precisely they create an internal system and share it with the public. This generosity links to the company size: Google³⁰, IBM³¹, Ant³², Adobe³³ can afford to open source code worth millions:

This represents millions of dollars of investment for each company to duplicate work that many other companies are also doing. — Adobe Spectrum³⁴

Foundational systems represent a new business model. Modulz³⁵ helps teams create design systems without writing code. Institutions³⁶ share their work for the common good.

Defining purpose and audience is free-form thinking. No rules apply here. The result is a sentence, or two, capturing the essence of a new endeavour.

Features

Features define the main characteristics of the system: composable, accessible, beautiful, performant, adaptive, natural.

Analysis shows the best practice is to choose measurable features instead of pursuing subjective goals. Accessibility is verifiable, beautiful is not. The measurable aspect gives credibility to a design system. It shows its creators' ability to execute a plan from start to finish.

Certain systems go for a full set of features while others cherry-pick parts of it. Some systems share a common set of features while others come up with a unique set.

Material Design goes full circle. From top-bottom connects designers and their tools with developers of all platforms and technologies. From Figma to web and native apps everything is in a system.

Radix from Modulz is bare bones to that level of not offering any styling.

Spectrum from Adobe offers server side rendering, virtual lists, state management and accessibility support for anybody building design systems.

³⁰ [Homepage - Material Design](#)

³¹ [Carbon Design System](#)

³² [Ant Design - The world's second most popular React UI framework](#)

³³ [Spectrum](#)

³⁴ [Why React Aria? – React Aria](#)

³⁵ [Modulz](#)

³⁶ [Bold Design System](#)

Shopify Polaris³⁷, IBM Carbon shares a common set of characteristics. Workday's Canvas³⁸ and University of Santa Catarina's Bold³⁹ do share, but another set of common features.

It might happen to the purpose and audience, the features of a new system to match an existing implementation.

A complete match renders the new system unjustifiable to build. A partial match enables code reuse: instead of creating from scratch, certain parts of the new system should borrow ready-made solutions from other systems.

Deliverables

A set of products define the deliverables of a system. Their quantity and quality defines user and developer experience, and the learning curve. All aspects combined drive or reject the adoption, the success of the system.

Unlike the workflow, analysis shows deliverables aren't uniform. Some systems deliver less, others deliver at a better quality.

Among deliverables code and documentation is mandatory. Without code there is no design system, without documentation nobody will understand and use it. The rest—design guidelines, live playground, example applications, tests and packaging—are optional.

The amount and quality of deliverables depends on company size and resources. State-of-the-art comes from big brands, innovation constrained by resources is specific for small companies.

The analytical framework underpinning this research tries to cover all ends. Identifies a minimum viable set of features with minimum viable quality requirements, then shows the high-end of the spectrum—how an ideal design system looks like.

Comparing different systems and approaches helps to better understand the problem domain. Scoring and classifying them is subjective and often wrong. It only serves to spot the viable requirements and the nice-to-have extremes.

Code

Code forms the base of all deliverables—everything else builds around—making it the most important delivery.

Code can or can not follow theory. But in all cases it should stay usable. Theory and usability don't necessarily overlap, but when do they reveal a perfect execution.

³⁷ [Shopify Polaris](#)

³⁸ [Workday Canvas Design System](#)

³⁹ [Bold Design System](#)

When a theoretical framework built on design patterns is in place, analysing code quality reduces to checking if the patterns apply.

The analysis shows a binary picture. Systems use either all, or the majority of design patterns, or none of them. Half of the analysed systems belong to the first group.

The extent of how patterns interconnect to form a whole defines the developer experience, the usability of the source code. The picture is more nuanced here. All analysed systems try to come up with an integrated experience to moderate the binary picture..

Leaders in implementing all design patterns do not necessarily lead in integration. Bold alone stands out in both categories. They've managed to combine design patterns in an uniform, complete, pleasant way.

Leaders who didn't score well in integration don't deserve blame. Their purpose and goals might not require to cover all aspects of code usability.

Documentation

The documentation site must present all building blocks of a design system—tokens and components together with their props—with an enjoyable experience.

The list of tokens and components reveal the structuring and naming practices and whether or not a theory is in place.

Concise and clear prop descriptions are of paramount importance. They define the bulk of the user experience of a design system. Easy to understand props, and their combinations showcased with live, editable examples result in faster application development. Poor props documentation might lead devs to abandon the system.

Shopify Polaris, Material UI, IBM Carbon stand out in presenting props. The other end—Canvas—is discouraging, even when it shines in other aspects like code quality and usability.

Integrated experience—when there are no separate apps to drive parts of the documentation—reduces cognitive load. When a feature takes the visitor to a visually different new site it takes a toll on developer attention.

IBM's Carbon offers the live playground via third party services lowering the user experience. The rest of the analysed systems offer their services integrated.

Search, another documentation feature, is necessary on systems showcasing a large number of tokens and components. Easy finds enhance user experience, no search breaks it. More than half of the systems offer this capability.

Generated documentation is an important but undervalued aspect where the majority of analysed systems fail.

Hand-written documentation might not represent the complete truth. It might go outdated, irrelevant, or missing. This approach requires manual replication of code changes in documentation. The process involves people, tools, time and might run out of sync.

When documentation is generated from code it always represents the code—sticks to the single source of truth. As code changes the documentation changes automatically. A practice a few of the analysed systems embrace in spite technology exists for automation.

Design guidelines

Design guidelines⁴⁰ capture design decisions during development. No matter the team and the project—there are always decisions to make.

A well-documented collection of such decisions is priceless when developers meet a new system. Later it can serve as a reference point.

Clear directions encapsulated in design guidelines reduce the learning curve—the time to understand the motivations of the creators and to grasp the big picture and reduce application development time.

Every analysed system implements design guidelines, the majority with success.

Live playground

As seen at props—live, editable examples are great help.

Live playgrounds go further. They offer a build-on-the-fly feature without requiring developers to leave the documentation site. Devs can test the capabilities of the system outside of an integrated development environment.

This aspect is important. First it makes sure the design system works as advertised. Bugs might appear in the development environment then disappear in the live playground. This gives precious hints for developers where to look for the fix. And confidence the original system works.

Second, when done well, live playgrounds can replace programming environments and enable non-developers to build apps without coding knowledge.

⁴⁰ [Human interface guidelines](#)

Around half of the examined systems provide a live playground. Braid goes far with its Playroom by “*empowering designers and developers to iterate together in the same medium using the same components, reducing the need for high fidelity mockups before development starts.*”

We want to allow you to spend less time polishing mockups and more time polishing the product — Playroom⁴¹

Example applications

After tweakable props and live playground the next step to showcase the features of a design system is best possible via example applications.

Example applications cover complete use cases thus offering the big picture. They often reveal subtle information impossible to find in the docs. And they show the way—this is how apps built with this design system will look like; this is how theory looks in practice.

Around two thirds of the analysed systems come with at least one example app. The majority manages to offer it at an enjoyable quality.

Tests

While non-mandatory deliverables, test suites assure code quality, scalability and performance. They guarantee the source code works and is extendable in the future.

Testing is hard. Tells all about the team and the company behind—if they are in for the long run; if their product is worth building on.

Analysis shows—again, like in theory—a binary picture. Either all tests—unit, integration—are in place or none of them. Young systems come without tests, stable implementations come with a complete suite.

The quality and coverage of these tests is questionable. No analysed system stands out and worth following.

Testing works when theory works. The example application accompanying this guide manages a 100% code coverage. This is possible by design—complete theory, best-in-class technology—and the relative low number of building blocks.

Packaging

Packaging makes the elements of a design system reusable. The source code splits into standalone modules and goes published into public repositories from where other projects and apps import and reuse them according to their needs.

⁴¹ [Braid Design System with Playroom](#)

Packaging and publishing to the NPM repository⁴² is mandatory. Analysis shows all systems provide this feature.

Systems designed for individual use should skip this step. Packaging is still a technical challenge in component-based UI frameworks. They also enforce a rigid folder structure reducing developer experience.

Monorepos⁴³ are a complementary technology to packaging. They support building applications with modular architecture—such as design systems and component libraries.

Less than half of the analysed systems use this feature. The reasons are the familiar—monorepos represent a technical challenge and often reduce developer experience.

⁴² [npm \(software\)](#)

⁴³ [Monorepo](#)

A big picture

Great exercise; no standalone best implementation; big brands lead; small shops innovate.

	Common practices
Workflow	
Purpose and audience	A single sentence or paragraph
Features	A list of measurable characteristics
Deliverables	
Code	Theory
	Developer experience
Documentation	Tokens, components and props
	Searchable
	User experience
Design guidelines	Reduces learning curve
Live playground	Integrated
Example apps	Worth more than thousands of words
Tests	Both unit and integration tests
	High coverage is possible
Packaging	Publish to NPM
	Monorepo

Practice

Edo—An example design system for marketing websites

Glossary

A list of terms and definitions specific to this domain. Both non-technical audience and tech leads, project managers might find it useful.

API

The programmable interface a design system is publishing for developers.

Component

See *Component-based UI framework*.

Component library

A component-based UI framework using an open-ended API.

Component-based UI framework

A web development methodology treating structure (HTML), presentation (CSS) and behaviour (Javascript) as a single concern.

Constraints-based API

Restricts the possibility to attach arbitrary numbers of styles to a HTML element.

CSS framework

A library allowing for easier, more standards-compliant web design using the Cascading Style Sheets language.

Design system

A template-based UI framework using a constraints-based API.

Developer experience

The experience developers perceive when extending a design system.

Interoperable

A part of a system is reusable across other implementations.

Open-ended API

Allows attaching arbitrary numbers of styles to a HTML element.

Principle

See *Software design pattern*.

Programming paradigm

A way to classify programming languages based on their features.

Scaling

Either refers to source code and site performance, or source code maintainability.

Software design pattern

A general, reusable solution to a commonly occurring problem within a given context in software design.

Software development theory

A collection of programming paradigms, software design patterns and principles.

Template

See *Template-based UI framework*.

Template-based UI framework

A web development methodology separating structure (HTML), presentation (CSS) and behaviour (Javascript) as different concerns.

Theory

See *Software development theory*.

User experience

The experience developers perceive when using a design system.

Resources

A selection of important resources this guide builds upon.

- [Adele – Design Systems and Pattern Libraries Repository](#)
- [The problems of CSS at scale](#)
- [jxnblk.com Two Steps Forward, One Step Back](#)
- [Building \(and Re-Building\) the Airbnb Design System Maja Wichrowska](#)
- [Sebastian Markbage: Minimal API Surface Area | JSConf EU 2014](#)
- [The reactive, functional nature of React](#)
- [Design Systems: React Buttons with the Base + Variant Pattern](#)
- [Writing Maintainable, Loosely-Coupled Code - Manning](#)
- [The Deno Manual](#)
- [Design Systems Analytical Framework](#)
- [Why React Aria? – React Aria](#)

Feedback

hi@osequi.com