ChatGPT

Architecture of the Design Constraint Validator (DCV)

High-Level Architecture Narrative

Design Constraint Validator (DCV) is a plugin-based engine that validates design tokens (structured values like colors, spacings, etc.) against a set of mathematical and accessibility constraints. At a high level, DCV ingests one or more token definition files (JSON format, possibly nested or in various token schema formats) and optional constraint configuration files, then produces outputs such as validation reports (violations of constraints) or processed token files (e.g. CSS variables). The system is organized into distinct phases and subsystems [1] [2]:

- **Token Parsing & Normalization:** Raw token inputs (potentially in different schemas like Style Dictionary, Figma Tokens Studio, etc.) are first parsed and normalized by adapter utilities. This step flattens nested token structures into a flat list of token entries, each with a stable dot-notated ID, and resolves any inter-token references (e.g. tokens whose value is defined as a reference to another token) [3] [4]. The output is a uniform internal token list with unique IDs and concrete values, with all `{token.ref}` placeholders replaced by actual values (or flagged if unresolved) [5].

- **Dependency Graph Construction:** From the flattened tokens, DCV builds a directed acyclic graph (DAG) of token dependencies [2] [6]. In this graph, each node represents a token, and an edge represents a dependency (e.g. token A's value references token B, so B → A is a graph edge). This graph is used to understand propagation of changes and to detect any **cycles** (circular references are disallowed). A cycle in token definitions is treated as a correctness error; DCV will detect it and report an error (e.g. "Circular reference detected: a → b → a") [7] [8]. By design, the token dependency graph must remain acyclic for the engine to function correctly.

- **Constraint Validation Engine:** DCV's core is a **constraint validation engine** that applies a series of constraint plugins to the token data [9]. The Engine maintains the current set of token values and the dependency graph in memory, and it allows plugins to be registered. Each **constraint plugin** implements a specific rule or invariant (for example, "typography sizes must be monotonic" or "foreground/background color contrast must meet WCAG ratios"). The engine invokes each plugin's check on the tokens and collects any violations produced [10] [11]. Notably, the engine's plugin system is extensible – built-in plugins cover common rules (WCAG contrast, monotonic order, etc.), and developers can add custom plugins. The plugin interface is simple: it takes the Engine (access to all token values) and returns a list of **violations**, each describing a rule that is broken (severity level, rule name, token(s) involved, and message) [12] [13]. The engine itself is stateless with regard to validation logic; all rule logic resides in plugins, making the engine a generic driver that can evaluate any set of constraints.

- **Violation Reporting & CLI:** After running validations, DCV reports any **constraint violations** found. The output can be machine-readable (JSON) or human-readable text. DCV includes a Command-Line

Interface (CLI) tool (`dcv` command) which wraps the core engine to provide user-facing commands [14]. Major CLI commands include `validate` (to run the full validation on a token set and report errors/warnings), `why` (to explain a token's provenance and which dependencies or rules affect it), `graph` (to output the token dependency graph or constraint graph in visual formats like Mermaid or DOT), and `build` (to generate artifacts like a CSS file or JSON of tokens). The CLI orchestrates the end-to-end flow: it locates and loads token files and constraint files, invokes the engine and plugins, and then formats the results for output. In essence, the CLI is a thin layer on top of the core engine that handles I/O (file system reads/writes, formatting, and user options), whereas the engine and plugins perform the core computation.

Overall, the architecture cleanly separates the concerns: **parsing** and **graph building** (data ingestion) happen before validation; the **Engine+Plugins** perform pure validation logic on an in-memory model; and the **CLI/Reporting** handles user interaction and output formatting. This design allows DCV to support different input formats and output formats without changing the core validation logic, and to extend new types of constraints via plugins without modifying the engine's code [10] [11].

Module Map (Folders & Responsibilities)

The repository is organized into directories that reflect these subsystems:

- **Core Engine (`core/`):** The `core` directory contains the fundamental logic of DCV. This includes the `Engine` class (in `core/engine.ts`) which manages token values, the dependency graph, and plugin evaluation [15] [16]. The Engine provides methods to get/set token values, register plugins, compute the set of *affected* tokens given a change (traversing the graph of dependents) [17], and evaluate all plugins against a set of candidate tokens [18]. The core also defines shared types like `TokenId`, `TokenValue` (primitive types for values), and `ConstraintIssue` (structure for a validation violation) [19]. Another key part of core is the **token flattener** (`core/flatten.ts`), which contains functions to "flatten" nested token JSON structures into a flat map and to resolve `{references}` in token values [20] [21]. This module produces two critical outputs: a dictionary of flat tokens (token ID → token object with value and metadata) and a list of dependency edges (pairs of `[fromId, toId]` meaning `fromId` is referenced by `toId`) [22] [23]. The core also includes a `patch` utility (to apply or generate diffs of token sets) and a `why` utility (`core/why.ts`) that helps trace a token's provenance. For example, `core/why.ts` can produce a report of a token's immediate dependencies ("depends on") and dependents, the source file ("provenance") indicating whether it came from base tokens, a theme override, etc., and even a simple reference chain if the token is an alias chain [24] [25]. This is used by the `why` CLI command to explain how a token's value is derived [26] [27]. Finally, `core` holds utility modules like `core/color.ts` (color space conversion, luminance and contrast calculations used by the WCAG plugin) and `core/poset.ts` (partial order utilities for monotonic relationships).

- **Constraint Plugins (`core/constraints/`):** Within core, the `constraints` subfolder contains implementations of built-in constraint plugins. Each plugin is typically exposed as a factory function returning a `ConstraintPlugin` object that the Engine can use. For example, `MonotonicPlugin` in `core/constraints/monotonic.ts` returns a plugin that enforces numeric ordering between token pairs [28] [29]. It takes a list of ordering rules (each a triple like [TokenA, ">=", TokenB]) and a parser function to convert token values to numbers, then its `evaluate` method checks each rule

and yields an issue if a rule is violated (e.g., TokenA < TokenB when it should be ≥) [29]. Crucially, these plugins are designed to integrate with DCV's incremental validation: they receive from the Engine a set of "candidate" tokens that changed, and they only report issues involving those tokens to avoid flagging the same static issue repeatedly [30]. Other core plugins include `WcagContrastPlugin` (`wcag.ts`) for color contrast rules, which ensures the contrast ratio between a foreground color token and background color token meets a minimum (e.g. 4.5:1) [31] [32], `ThresholdPlugin` (`threshold.ts`) for simple min/max threshold rules on numeric values (e.g. a token must be ≥ 44px) [33] [34], and `CrossAxisPlugin` (`cross-axis.ts`) which handles more complex multi-factor rules (e.g. if token X has a certain value, token Y must satisfy some condition relative to X) [35] [36]. The cross-axis plugin is quite sophisticated, allowing conditional requirements and even custom contrast checks across tokens [37] [38]. All plugins conform to the `ConstraintPlugin` interface (with an `id` and an `evaluate(engine, candidates)` method) and produce `ConstraintIssue` objects when rules are violated.

- **Partial Order Utilities (`core/poset.ts`):** DCV uses partial-order (poset) representations for monotonic constraints. The `poset` module can construct a directed graph from a list of order rules, then compute the **transitive reduction** to produce a Hasse diagram (a minimal graph of the partial order) [39] [40]. It provides functions to output these graphs in formats suitable for visualization. For example, `toMermaidHasseStyled` produces a Mermaid.js flowchart text with nodes and edges, highlighting certain nodes/edges and labeling them if they correspond to constraint violations [41] [42]. Similarly, `toDotHasseStyled` produces Graphviz DOT format output with optional highlighting [43] [44]. These utilities are used by the `graph` CLI command when exporting constraint graphs (e.g. typography scale) to help users visualize monotonic relationships and pinpoint violations (violating edges can be colored differently in the output) [45] [46].

- **Adapters (`adapters/`):** The adapters folder contains code to handle various **input and output formats**. DCV is not tied to one token file schema; it aims to accept common design token JSON formats. The adapters define how to normalize these into DCV's internal model. The project's documentation states that adapters strive to be **pure mappers** from external format to internal, preserving all meaningful data (IDs, types, descriptions) and avoiding side-effects [47] [48]. For instance, if using Style Dictionary format (which uses a `"value"` field and nested objects), an adapter would map each token to a flat `{id, value, type, meta}` structure, possibly renaming `"value"` to DCV's expected `$value`. In practice, the core flatten function looks for the `$value` key to identify token definitions [20], so Style Dictionary's plain `"value"` might be converted to `$value` before flattening. The `adapters/json.ts` and `adapters/js.ts` modules provide output functions: e.g. `emitJSON()` serializes the final tokens and their values to a JSON file (with an optional mapping of token IDs to alternative keys) [49] [50], and `emitJS()` exports the tokens as a JavaScript module (essentially embedding the JSON in an `export default {...}` format). The `adapters/css.ts` module handles generation of a CSS Custom Properties file from token values, mapping token IDs to CSS variable names. It includes a default mapper (which turns a token id like `dimension.spacing.scale.4` into a CSS var `--dimension-spacing-scale-4` by replacing dots with hyphens) [51], and supports a **manifest** of aliases so that multiple CSS variable names can point to the same token (for backward compatibility of CSS vars) [52] [53]. Using these, it can output a `:root { --token-name: value; ... }` block for all tokens [54], or a smaller patch block for just changed tokens [55]. In summary, adapters isolate the differences in file formats and naming conventions, allowing the rest of the engine to work on a consistent data shape.

- **CLI Commands (`cli/`):** The `cli` directory contains the implementation of the various CLI commands and supporting utilities. It uses the core modules to accomplish user-level tasks. For example, the `validate` command (`cli/commands/validate.ts`) loads the configuration and token files, sets up an Engine with all relevant constraints, runs the validation, and outputs a summary [56] [57]. Internally, it uses an **engine helper** (`cli/engine-helpers.ts`) to create and initialize the Engine. This helper reads default constraint files (like `themes/typography.order.json` for monotonic typography rules, etc.) and registers the corresponding plugins on the engine [58] [59]. It also reads the user's config (if any) for additional WCAG contrast rules to apply [60]. The validate command loops through breakpoints if multi-breakpoint mode is requested (treating each breakpoint as a separate token set by merging override files) [56] [61]. It then collects all issues and prints errors/warnings, or in JSON mode produces a structured report including stats [62] [63]. Another important CLI module is `cli/commands/graph.ts`, which handles the `graph` command. This has two modes: if asked for a **Hasse diagram** (monotonic order graph), it will load an order definition (e.g. `themes/typography.order.json`), build a poset via `core/poset.ts`, optionally highlight violations by actually running the engine on those tokens, and then output the diagram in the desired format (Mermaid, DOT, etc.) [45] [46]. If instead a raw dependency graph of all tokens is requested, it flattens the tokens and outputs the reference graph (all token reference edges) in Mermaid, DOT or JSON form [64] [65]. The CLI also includes the `why` command implementation (`cli/commands/why.ts`), which uses `flattenTokens` and the `core/why.explain` function to produce a provenance report for a specific token [66] [67], then formats it either as JSON or a human-readable table of dependencies/dependents [68] [69]. The `build` command (`cli/commands/build.ts`) uses the flatteners and adapters to output the design tokens in various formats (CSS, JSON, JS) for use in applications – essentially acting like a "compiler" of the design tokens into consumable artifacts [70] [71]. The CLI modules share some helper code (in `cli/commands/utils.ts` and others) for tasks like loading JSON files safely, writing output to files, pretty-printing tables, etc.

In summary, the **module structure** cleanly separates core logic (engine and plugins) from CLI and I/O concerns. The **Core Engine** and **constraints** modules focus on in-memory operations and algorithms, **Adapters** handle format transformations for input/output, **Poset/Graph** utilities handle specialized graph logic for visualization, and the **CLI** ties everything together into user-visible commands.

Data Flow Through the System

1. Input Ingestion: The data flow begins with reading the design token definitions. The CLI (or the library user, if using the API) specifies a path to a tokens JSON file or directory. DCV can handle a single flat JSON or a set of JSON files (e.g., a base tokens file plus optional override files for responsive breakpoints). The `loadTokens` utility simply reads the JSON into a JavaScript object [72]. If breakpoint-specific validation is requested (e.g. `--breakpoint md` or `--all-breakpoints`), the system will load the base tokens and then merge in a secondary JSON specific to that breakpoint [73] [74]. For example, for `md` (medium) breakpoint, DCV will load the base `tokens.json`, then merge overrides from `tokens/overrides/md.json` (if present), with overrides taking precedence [73] [74]. This merging is handled by a utility that deep-merges JSON objects (see `core/breakpoints.ts` `mergeTokens` function) to ensure the final token set for that breakpoint includes all base tokens plus any overridden values for that context [75] [76]. The result after this step is a single unified token object (potentially nested) representing the design tokens for the chosen scope (either global or a specific breakpoint).

2. Parsing and Flattening: Next, DCV normalizes this token object. The `flattenTokens` function is applied to the merged token data structure [77]. This function traverses the nested object recursively to produce a flat list of tokens. In this traversal: - Every nested path (like `color.brand.primary`) becomes a dot-delimited **TokenId** (`"color.brand.primary"`). - If a token entry has a `$value` field, that indicates a concrete token definition. The flattener records its ID and value, and also captures metadata: the `$type` (type of token, if provided) and any references within the value string [78] [23]. For example, a token defined as `{"primary": { "$value": "{color.base}", "$type": "color"}}` will be flattened into an entry with `id: "color.primary"`, `value: "{color.base}"` (initially), `type: "color"`, and a list of `refs: ["color.base"]` because the value references another token [79] [80]. - The flattener also immediately adds a dependency edge for each reference found: in the above case, it adds an edge (`"color.base" → "color.primary"`) to indicate `color.primary` depends on `color.base` [23]. - After collecting all tokens, a second pass of the flattener **resolves references**. It iterates through tokens replacing any `{token.path}` placeholders with the actual values of those referenced tokens [21] [81]. This resolution is done iteratively (repeated until no unresolved references remain or a maximum iteration count is reached) [82] [83]. If it cannot resolve a reference (e.g., a token references an undefined token) or if a cyclical chain prevents full resolution, the flattener throws an error to indicate an invalid token configuration [84] [85]. By the end of this phase, we have: - A flat map of **TokenId → Token data** (each with final `value` resolved, original `raw` value preserved, and metadata like type). - A list of **Graph Edges** (each an ordered pair `[from, to]` meaning `to` token depends on `from` token's value).

These outputs represent the internal model of the design tokens, independent of how they were originally formatted. The adapter's goal of preserving provenance is partially reflected here: for instance, DCV tags each token with a provenance label (base/theme/override) by checking which source file it came from (the `why` report uses this, labeling a token as `"base"` vs `"override"` origin by seeing if its ID existed in the override layer) [26] [86]. This way, later reporting can indicate if a value came from a base token file or an override.

3. Building the Engine & Graph: With the flattened token map and edges, DCV initializes the validation engine. The `Engine` is constructed by passing in the initial values (a record of token IDs to primitive values) and the list of edges [87]. Internally, the Engine stores the token values in a map and builds an adjacency list representation of the dependency graph [88]. Each edge `[A, B]` (meaning B depends on A) is registered such that in the Engine's graph, `A`'s set of dependents includes B [88]. The engine at this point has a complete picture of the token dependency DAG (direct references only; transitive closure can be computed when needed). It may also perform a sanity check for cycles here or rely on the flattener having not thrown an error if a cycle was present. The engine is now ready for constraint evaluation.

4. Loading Constraints: Before validation, DCV determines what constraints (plugins) to apply. There are multiple sources for constraint rules: - **Built-in defaults:** DCV has some default constraint rules that are applied if relevant files are present. For example, it looks for files like `themes/typography.order.json`, `themes/spacing.order.json`, `themes/color.order.json`, etc. These define monotonic ordering rules for various token categories (e.g., that larger heading tokens ≥ smaller heading tokens). If such a file exists, DCV loads the list of order pairs and instantiates a MonotonicPlugin for it [58]. The "color" order file triggers a special `MonotonicLightness` plugin (ensuring a set of colors have monotonically increasing/decreasing lightness) [89]. This mechanism means that simply dropping an `.order.json` file in the `themes/` directory automatically activates the corresponding plugin, which is convenient but also a bit implicit. - **Configuration file (`dcv.config.json`):** Users can supply a config that, among other things,

lists custom WCAG contrast rules. For instance, a config might specify certain token pairs and required contrast ratios. If the config provides a `constraints.wcag` array, DCV will create a WcagContrastPlugin with those rules [60] [90]. (No custom monotonic rules are configured via `dcv.config.json` currently – those come from the separate order files, as described – and cross-axis rules are handled in their own JSON, below.) - **Policy/Theme files:** Some constraints might come from what the documentation calls "policy" or theme JSON files. For example, the user could have a file for an AA contrast policy or an internal design policy. In the current architecture, a "policy" JSON for contrast might actually just be fed into `dcv.config.json` or the WCAG plugin. Another type of policy is the **cross-axis rules** file: by default DCV expects a file `themes/cross-axis.rules.json` defining any cross-axis constraints (complex conditional rules). The `validate` command always attempts to load `themes/cross-axis.rules.json` (and breakpoint-specific versions like `cross-axis.sm.rules.json`) and, if found, registers a CrossAxisPlugin with those rules [91]. So, much like monotonic orders, the presence of a file in the `themes/` directory controls whether a certain plugin runs. - **Always-on rules:** A small number of constraints are hard-coded. Notably, DCV always enforces a **minimum touch target size** of 44px (this is a common accessibility guideline) by injecting a ThresholdPlugin for token `control.size.min >= 44px` [92] [93]. This rule is added in multiple places (during `validate` and also during `graph --highlight-violations` when generating diagrams) to ensure that even if the user hasn't specified it, the check is performed. This is a design decision – it provides a sensible default but currently cannot be configured off except by using `--fail-on off` to ignore warnings.

Once the relevant plugins are identified and created, the Engine registers them via `engine.use(plugin)`. At this point, the **Engine is fully configured** with the current token graph and all applicable constraint checks.

5. Validation Execution: DCV now evaluates the constraints. If running the full `validate` command (as opposed to an incremental single change), it will consider *all tokens* as candidates for checking. In practice, the validate CLI does: `issues = engine.evaluate(allTokenIds)` [57], passing all token IDs to the Engine's evaluate method. The engine in turn invokes each plugin's `evaluate` or `check` function. Each plugin scans through its rules. Because each plugin knows which tokens or token pairs it cares about, it can limit its work to those. Many plugins also utilize the candidates set to skip work on tokens that haven't changed – in the full validation case, the candidate set is effectively all tokens, so no skipping occurs. The plugins return lists of violations (if any). These violation objects include details like severity (`"error"` or `"warn"`), a rule identifier (e.g. `"monotonic"` or `"wcag-contrast"`), the token(s) involved, an optional context ("where") for additional explanation, and a message describing the problem [94] [95]. For example, the MonotonicPlugin might produce a violation: `{level: "error", rule: "monotonic", id: "typography.size.h1|typography.size.h2", message: "… violated: 32 < 34"}` if a larger heading token was found to be smaller than a subordinate heading [96]. All these violations from all plugins are gathered into a final list.

6. Reporting & Output: The final set of violations is then reported according to the user's specified format. In CLI text mode, DCV will print a summary and each violation line by line, tagging them as ERROR or WARN and indicating the rule and token(s) [62] [97]. In JSON mode, DCV will output a JSON object containing an array of violations and some statistics [63]. The statistics typically include the number of tokens checked, number of errors, number of warnings, and duration, among other possible data [98]. If the user provided the `--fail-on` option, DCV uses that to decide the process exit code (for example, fail on warn means treat any warning as a failing condition). After reporting, the validate command is done (the Engine and data are just discarded, since it's a one-time run).

Other commands have variations on this flow: - The `why` **command** essentially stops after flattening. It does not run all constraints, but rather uses the flattened data and graph to trace one token. Using the `explain()` function, it collects that token's current value, what other tokens it directly depends on (parents in the graph) and what tokens depend on it (children), and identifies the provenance (base or override) [25] [26]. It also tries to follow a simple chain of references – if token A references B, which references C, it will produce a chain A→B→C for clarity [99] [100]. The output is either a JSON with these fields or a formatted table in the console listing the dependencies and dependents with their values. - The `graph` **command** for a dependency graph takes the flattened edges and formats them without involving the Engine or plugins (unless the user requests highlighting of violations). Essentially, it treats the `edges` list from flatten as a directed graph and outputs it. For example, in Mermaid format it will list each token as a node and draw arrows from each source token to the tokens that reference it [101] [65]. If filtering by a regex is requested, it will filter out edges not matching the pattern [65]. This gives a visualization of *all token interrelationships*. - The `graph --hasse` **(poset graph)** command takes a monotonic constraint file (order rules) and uses the poset utilities. The data flow in that case: it reads the monotonic rules from a JSON file (independent of the main tokens; these files list design decisions like color or typography hierarchy). It builds a poset graph and reduces it to a minimal Hasse diagram [45] [102]. If the user wants to highlight violations on that graph, DCV will *actually instantiate an Engine with the tokens and run the relevant monotonic plugin on those tokens* to find which relations are violated [103] [46]. It then marks those specific edges or nodes in the graph output (for example, coloring an edge red and labeling it with the violation message) [46] [92]. This is a clever reuse of the Engine: even for graph visualization, the engine's output is used to annotate the graph with real data about rule satisfaction. - The `build` **command** doesn't run constraints at all. Its data flow is: flatten the tokens (and possibly merge in an additional "theme" tokens file if `--theme` option is used to overlay theme-specific token values) [104], then take the flat token values and feed them to the appropriate adapter for output. It can output to CSS (via `valuesToCss` which produces a `:root { --var: value; }` stylesheet [70] [71]), to JSON (via `emitJSON`, which creates a JSON of token id → value [105] [106]), or to a JS module (via `emitJS`). If `--all-formats` is specified, it writes out all three formats in one go [107]. The build process ensures the output directory exists and writes the files, logging the paths. Essentially, `build` is taking the internal token representation and emitting it in various external representations for consumption by other tools (like generating a tokens.css that can be used in an app). No validation occurs in the build flow (the assumption is you run `validate` separately).

Throughout these flows, a few **important data integrity checks** occur: - **Circular Reference Check:** The flatten phase ensures there are no infinite loops in references; a max iteration count is set to break and error out if a cycle is suspected [82] [85]. - **Missing Token Reference:** If during resolution a `{reference}` cannot be resolved (no such token), the flattener throws an error [84]. This would typically abort validation with a fatal error, as it indicates the input tokens are inconsistent. - **Unknown Constraint Targets:** The cross-axis plugin loader checks that any token IDs referenced in rules actually exist in the known token set; it logs warnings if a rule refers to an unknown token (helping users catch typos in the constraint files) [108] [109]. - **Candidate Set Limiting:** When using the Engine's `commit()` method (which applies a single token change), the engine computes the set of *affected* tokens (all tokens that depend on the changed token) [110] [111]. It then evaluates only those tokens' constraints, not touching unrelated parts of the graph. This yields an output patch containing the changed token (and potentially any recomputed dependent token values, if that were implemented) and any new issues. This design guarantees that making a change in one area doesn't trigger re-validation of distant, unaffected tokens, improving performance. The commit operation returns the list of affected token IDs, the list of constraint issues found, and a patch (the delta of changed values) [111]. In the current DCV CLI, this fine-grained incremental use is not directly exposed, but the internals support it for potential future real-time editing scenarios.

After any command finishes its run, the outputs (violations, graphs, or files) are available to the user. Importantly, DCV's processing is deterministic – given the same input tokens and constraints, it will always produce the same set of violations and the same output files. There are no random or time-based variants in the engine; it's purely based on the input data and rules. This reproducibility is crucial for CI/CD usage: if a design token change introduces a violation, it will consistently be caught by DCV in any environment.

Key Invariants and Correctness Conditions

DCV's architecture enforces several **correctness invariants** to ensure the validity of the token set and the reliability of validation results:

- **Acyclic Token Graph:** The dependency graph of tokens must remain a Directed Acyclic Graph (DAG) [2] [6]. Cycles in token definitions (even indirect ones) are considered errors. The system will detect a cycle during flattening/resolution and report it, rather than entering an infinite loop. For example, if token A references B and B references A, DCV will flag a "Circular reference" error [7] [8]. This invariant ensures that tokens have well-defined values (no endless recursion) and that the Engine's graph algorithms (like traversing dependents) terminate.

- **Unique Stable Token IDs:** Each token is identified by a unique dot-notation ID string. These IDs act as stable identifiers across the system – they serve as keys in the Engine's value map, node labels in the graph, and appear in violation reports. Adapters are designed to preserve or generate stable IDs from the input structures [48]. The correctness condition here is that no two distinct tokens share the same ID (which is guaranteed by how flattening works: two different paths in the JSON would lead to different dot paths, and if there's a duplicate key, the JSON itself wouldn't parse or the later one would override). Stable IDs are crucial because DCV uses them to track provenance and to map violations to specific tokens. Once established, a token's ID does not change during a run of the engine; if the underlying value is updated (via commit or override), it's still the same token ID in reports. This immutability of identity allows consistent tracking of issues (e.g., you wouldn't want an ID to appear in one part of the process and a different label in another).

- **No Mutation of Input Structures:** The token normalization process and adapters follow a functional approach – they create new normalized objects rather than in-place modification. The adapter guidelines explicitly state "avoid mutation: adapters are pure mappings" [47]. This means the original token JSON remains untouched; DCV produces a new flat representation for internal use. This purity helps maintain a clear separation between input data and DCV's output, and avoids side effects that could confuse multi-pass processing or watchers. It also means one can reuse the same input object to run DCV multiple times (for different breakpoints, for instance) without it being altered by a previous run.

- **Deterministic Constraint Evaluation (Purity of Constraints):** Constraint plugins behave like pure functions: they inspect token values via the Engine's `get` interface and output a list of violations, without altering any Engine state. The plugin interface does not provide any way to modify tokens – they only read the Engine's data [12]. This is by design: constraints should be checks, not transformations. An invariant here is that running the same plugin on the same engine state will always yield the same violations. There are no random elements in the checks (e.g., the WCAG plugin computes a contrast ratio formula, which is deterministic given the inputs [112]). Plugins also should not have side effects like caching or modifying Engine values. This ensures that the order of plugin

execution doesn't matter for the final result (and indeed, DCV treats plugins as independent checks that can be run in any sequence or even in parallel). It also ensures **idempotence**: running the full set of validations twice yields the same set of violations; running it once after a commit yields just the new violations relevant to that commit, with everything else unchanged.

- **Violation Consistency and Referential Integrity:** Every reported violation refers to real token IDs that exist in the token set (and often to real relationships in the graph). The system ensures that a violation object's `token` or `nodes` field corresponds to valid tokens. For example, a WCAG contrast violation will list the foreground and background token IDs that failed – those come directly from the rules provided and are validated against the known token IDs when the plugin is loaded [113] [114]. The cross-axis loader even keeps track of "unknownIds" and logs suggestions if a rule references a token that wasn't found, effectively enforcing that you can't silently have a constraint on a non-existent token [109] [113]. This invariant prevents the system from reporting nonsense errors. Additionally, the *graph edges* included in some violation reports (the `edges` array in a Violation) are always actual edges from the dependency graph, giving further guarantee that the violation context is consistent with the token relationships. For instance, a custom plugin might include `nodes: ['tokenA','tokenB']` and an edge `[tokenA, tokenB]` to illustrate that the relationship between A and B is part of the violation context – DCV expects those to be actual nodes and edges known to the Engine.

- **No False Positives from Unaffected Tokens (Incremental Validity):** When running in incremental mode (via the Engine's `commit` or by providing a limited candidate set to `evaluate`), DCV observes an invariant that *only tokens that have changed or depend on a change should be revalidated*. The Engine's `affected()` function computes all transitive dependents of a starting token [110] [115], and only that subset is passed to plugins. Furthermore, each built-in plugin reinforces this by internally filtering on the candidates set (for example, MonotonicPlugin only pushes an issue if either of the tokens in the pair was in the candidates set) [30]. The result is that a constraint that is technically violated but by tokens that have remained unchanged will not be reported during an incremental update – it would presumably have been reported earlier and still holds, so re-reporting it on every change is avoided. This invariant (no checking unchanged subgraphs) helps maintain **output stability**: making an unrelated change should not suddenly surface an old violation again and again. It also improves performance by not re-computing known state. (If a full re-validation is needed, the user would run the `validate` command fresh, which checks everything.)

- **Reproducible Outputs and Idempotence:** If no tokens change and no constraints change, running DCV should produce the exact same output each time. This is a subtle invariant but important for CI usage – DCV will not nondeterministically flip an outcome. Because the engine logic and plugins are pure and functional, and the input parsing is deterministic, the JSON report or build artifacts should be byte-for-byte the same given identical inputs. The only caveat could be the ordering of how violations appear (since plugins are iterated in an array and tokens in an object might be iterated in insertion order, it's stable as long as code isn't changed). The design avoids things like using current timestamps in the output or random sampling, so the outputs are suitable for diffing between runs.

- **Constraint Soundness (no spurious passes or fails):** Each built-in constraint is designed to accurately reflect the intended design rule. For instance, the WCAG contrast calculation converts colors to a linear luminance and computes the contrast ratio formula as defined by the standard [116] [117]. It even handles alpha compositing of semi-transparent colors over a background [112]. These

implementations maintain the invariant that a passing token set truly meets the specified guidelines. If DCV says colors have contrast $\geq$ 4.5, it's because by formula they do. Similarly, monotonic rules use a numerical parser (like `parseSize`) to ensure they're comparing apples to apples (converting all values to, say, pixels or numeric scale) [118] [119]. This prevents false passes (e.g., strings compared lexicographically would be wrong, so they parse to numbers). The integrity of these rule checks is part of DCV's correctness – it would be a bug if a constraint plugin reported a violation erroneously or missed a real violation. Hence, a lot of attention is given in code to parsing and comparing correctly, as well as skipping cases that can't be evaluated (e.g., if a token's value can't be parsed to a number, monotonic plugin will skip it rather than flag it incorrectly) [29] [120].

In summary, DCV's invariants ensure that the token dataset it processes is well-formed (no cycles, valid references), that the evaluation of constraints is consistent and isolated (no unintended interactions or state leaks between checks), and that the output accurately and deterministically reflects the state of the design system's compliance with the defined constraints.

Architectural Smells and Potential Issues

While the overall design of DCV is sound, a few architectural weaknesses and code smells emerge on closer inspection of the implementation:

- **Scattered Configuration & Duplicate Logic:** The handling of constraint rules is spread across multiple mechanisms in a way that could be unified. For example, monotonic constraints are loaded from dedicated JSON files (one per axis like typography, spacing, etc.), WCAG contrast rules can come from a `dcv.config.json` or default presets, cross-axis rules come from another JSON, and a threshold rule is hardcoded in the code [92]. This scattering means a developer needs to look in several places to understand all enforced constraints. Additionally, some logic is duplicated: the code that loads order files and registers multiple monotonic plugins appears in two variants (one in `createEngine` and one in `createValidationEngine` for breakpoint-specific loading) [58] [121]. They both do similar loops for typography, spacing, layout, color with nearly identical code. If a new category of order were added, it would have to be inserted in multiple places. This duplication is an opportunity for mistakes (one path might register a plugin that another path forgets). It also slightly complicates maintenance – the monotonic plugin essentially does the same thing regardless of breakpoint (just on a subset of tokens), so conceptually one could factor that better.

- **Core/CLI Boundary Blur – File I/O in Core Layer:** Ideally, core logic (the engine and constraints) should be independent of I/O and file paths, but there are places where this separation is violated. The `core/breakpoints.ts` and `core/cross-axis-config.ts` modules directly read files from disk (`fs.readFileSync` etc.) [122] [123]. They assume a particular project structure (looking for `tokens/tokens.example.json`, `themes/*.json` files, etc.) inside core code. This is more of a CLI concern or at least an integration concern – the core engine should theoretically just consume data given to it. By hardcoding file paths and reading them, the core is somewhat coupled to a file-based project layout. This makes it less flexible if someone wanted to use DCV as a library with tokens provided programmatically (bypassing the filesystem). It also duplicates knowledge of "where things are" between CLI and core. For instance, `loadTokensWithBreakpoint()` in core assumes a default file name `tokens.example.json` for base tokens [124], which might not match what the user passes via CLI. In fact, there's a slight inconsistency: the CLI `ValidateOptions` has a

`tokens` path option [125] [126], but the validate command implementation ignores it and always calls `loadTokensWithBreakpoint()` with its internal defaults. This suggests a misalignment – the core is off doing its own thing, even if the user requested a specific file. Such tight coupling of core code to a specific file structure is an architectural smell; it limits reuse and can lead to confusion (a user might specify a tokens path, but DCV might still load a different default file).

- **Incremental Filtering Responsibility:** The incremental validation design, while efficient, relies on each plugin to manually filter issues based on the `candidates` set. This is not enforced by an interface – it's a convention the built-in plugins follow (e.g., MonotonicPlugin only pushes an issue if one of the involved tokens is in the `candidates` set) [30]. If a custom plugin author doesn't adhere to this pattern, their plugin might report violations even for untouched tokens during incremental runs, breaking the intended "only changed parts" behavior. In fact, the official docs for custom plugins show an example plugin that checks all tokens of a certain pattern without mention of the candidates filtering at all [127] [128] (likely because the docs describe an older `check(engine)` interface). This discrepancy points to an **architectural smell in the plugin API**: it's not clearly defined how incremental updates should be handled. Either the Engine should handle filtering which issues to surface or the plugin interface should explicitly include the candidate set in a documented way. As it stands, the burden is on each plugin to do the right thing, which is error-prone and not obvious from an outside perspective (a new plugin might inadvertently flood the output with repeated issues). In short, the abstraction of a "ConstraintPlugin" is a bit leaky – the plugin needs knowledge of DCV's incremental strategy to function ideally.

- **Mix of JS and TS Artifacts:** In the repository, one can see both `.ts` and compiled `.js` files checked in side by side (e.g., `core/flatten.ts` and `core/flatten.js` both exist). This isn't an architecture flaw per se, but it's a maintenance smell. It raises the risk of discrepancies between source and output if not managed, and clutters the repository structure. Ideally, build artifacts wouldn't be committed or the source structure would be separate from the distribution structure. Having them together can confuse readers and contributors – one might accidentally read the older compiled .js instead of the authoritative .ts source. It's a minor point, but it does reflect on the project's structure.

- **Implicit Conventions and Hidden Coupling:** Some parts of the design rely on convention rather than explicit wiring, which could be seen as brittle. For example, the Engine helper automatically loads certain plugins if corresponding files are found, as mentioned. This "magic" is convenient, but if someone wasn't aware, it might be surprising that dropping a file in a directory changes the program behavior. Similarly, the naming convention of files (`*.order.json`, `cross-axis.rules.json`) is effectively an API, but it's not surfaced except in documentation. The core code doesn't log "loading constraints from typography.order.json" unless debug is on; it just silently does it [58]. This can make debugging harder – if a user misnames a file, DCV will simply not load it with no obvious indication except perhaps a debug flag. A more explicit configuration might improve clarity. There's also hidden coupling in how token override layers are identified: the `why` logic assumes if a token ID exists in the overrides JSON it loaded, that token's provenance is "override" [26]. This works given how `loadTokensWithBreakpoint` merges layers, but if that process changes, the why logic would break. These are somewhat subtle and minor, but they show places where responsibilities could be more cleanly delineated or signaled.

- **Minor API vs Implementation Mismatch:** The documentation and types sometimes refer to concepts not perfectly mirrored in code, which can be seen as a structural/documentation smell. For instance, docs refer to an `Engine.validate()` method and plugin `check()` methods [12] [129], whereas the implementation uses `Engine.evaluate()` and plugin `evaluate()` methods [16] [130]. It's clear the design evolved (perhaps from a simpler API to one supporting incremental evaluation), but these inconsistencies, if not updated, can confuse users or new contributors about how things are actually structured. Another example: the output violation type in docs has `kind` and `nodes` fields [131], but in code `ConstraintIssue` uses `rule` and some plugins use `id` to combine nodes, etc. While not catastrophic, it hints at some *technical debt* in aligning naming and structure between the conceptual model and actual implementation.

- **Tight Coupling of CLI and Engine Lifecycle:** The CLI commands often re-run flattening and engine setup even when some of that work is duplicative. For example, the validate command flattens tokens internally to build the engine, *and then flattens them again* separately to collect initial IDs for cross-axis plugin suggestions and such [56] [132]. This double computation is inefficient. It stems from the CLI not retaining the intermediate data – `createValidationEngine` reads files and flattens internally, but the validate command then does its own flatten to gather stats. This indicates the boundaries between phases aren't as modular as they could be (perhaps `createValidationEngine` could return not just the engine but also the flat tokens it built so that the CLI doesn't need to redo it). It's a small performance smell that could become an issue with very large token sets. Moreover, the CLI's approach of building everything from scratch for each command means if a user wants to both validate and then build, the tokens will be parsed and flattened twice in two separate invocations. There is no caching of the graph or reuse of computed results across commands, which could be a future improvement area (though not trivial given the CLI stateless invocation model).

Most of these smells do not cause immediate failures; rather, they affect **maintainability, extensibility, and clarity** of the system. The duplication and scattered config can make adding new constraint types or input types more error-prone. The fuzzy separation between core and CLI (with core doing file I/O) could complicate using DCV in a different context (say, integrating into a GUI app where you want to supply data directly). And the reliance on convention and plugin self-discipline (re: candidates filtering) could lead to subtle bugs if not carefully followed.

Refactor and Improvement Suggestions

To address the issues above and generally improve the structure, here are several concrete refactoring suggestions:

- **Unify Constraint Configuration:** It would be beneficial to centralize how constraints are fed into the system. Right now, the engine helper auto-loads multiple JSON files and uses both config and code defaults. A more unified approach could be to have a single configuration file (or a section in `dcv.config.json`) that lists which constraints to apply and where to get their rules. For example, instead of hardcoding file names in the engine helper, the config might have:

```
{
  "constraints": {
```

```
    "monotonic": ["themes/typography.order.json", "themes/
spacing.order.json"],
    "colorLightness": "themes/color.order.json",
    "wcag": [ ... rules ... ],
    "threshold": [ { "id": "control.size.min", "op": ">=", "value": 44 } ],
    "crossAxis": "themes/cross-axis.rules.json"
  }
}
```

This way, all constraint setup is driven from one place. The engine creation code can then simply iterate this config: load each file or rule set and register the appropriate plugin. This reduces implicit convention (no guessing file names – the config declares them) and makes it much easier for users to **enable/disable constraints**. For maintainers, adding a new constraint type is a matter of supporting it in the config and writing the plugin, rather than wiring in multiple spots. As a side effect, this would eliminate duplicated plugin-loading loops and the magic "files in themes directory" coupling, improving clarity.

- **Decouple Core from File System:** The core `Engine` and related functions should operate purely on data (already partially the case), with file system interactions moved outward. For instance, `loadTokensWithBreakpoint` and `loadCrossAxisPlugin` could be moved to the CLI layer or an intermediate "data loading" layer. The CLI would read the files (which it already largely does) and then call a core function like `mergeTokens(base, override)` (that part is fine in core as it's data manipulation) and `CrossAxisPlugin.fromRules(rulesJson)`. In other words, have the CLI pass the loaded JSON object of cross-axis rules into a CrossAxisPlugin factory, rather than having `loadCrossAxisPlugin(path)` inside core read the file. This refactor improves **separation of concerns**: core logic can be reused with any data source (e.g., in-memory), and the CLI or consumer takes responsibility for I/O. It also means the Engine doesn't assume a project structure; you could supply tokens from any source. Implementing this might involve slight redesign, for example:

  - Provide a public API to create a CrossAxisPlugin given a rules object (currently `loadCrossAxisPlugin` reads and then creates; this could be split).
  - Change `loadTokensWithBreakpoint(bp)` to something like `applyBreakpointOverrides(baseTokens, bp)`, which takes already-loaded JSON and just applies the merge (no file reading internally).
  - Remove the assumption of `tokens.example.json` in favor of requiring the caller to specify the base tokens path (the CLI already knows it from options). If a user doesn't specify, the CLI could still default to `tokens.json` or `tokens.example.json`, but in one place.

These changes would make the system more **flexible**. For instance, one could embed DCV in a web app where tokens are fetched via HTTP or generated on the fly, by calling the core flatten and engine functions with data, bypassing any filesystem dependency.

- **Streamline Engine API for Reuse and Testing:** The Engine is currently mainly used via the CLI's helpers. To facilitate reuse (and make it easier to write unit tests for constraints), the core API could be expanded. One suggestion is to add an `Engine.validateAll()` method that simply calls `evaluate()` on all tokens internally (it can easily get all keys from its map). This would match what the docs originally suggested (an `engine.validate()` for full run) and remove any confusion.

Similarly, adding `engine.getAllTokens()` or a method to retrieve the internal flat map could be useful (the code often reconstructs that map externally by re-flattening or by capturing it during Engine creation). Exposing it would avoid the redundant flatten calls. It would also let advanced users introspect the Engine's view of the world. These improvements make the Engine more self-sufficient, reducing the need for parallel structures in the CLI.

- **Eliminate Redundant Computations:** As noted, flattening occurs multiple times for the same input in some flows. A refactor could ensure that once tokens are parsed and flattened, those results are reused. For example, the `createValidationEngine` function could be modified to accept pre-flattened data (or to return the flat data along with the engine). Alternatively, the CLI validate could flatten the tokens once, then pass both the flat data and edges into Engine (there is a constructor for Engine that takes values and edges already). It could then use that same flat data for things like knownIds set or for output if needed. This change would mainly improve performance and memory use on large token sets, but it also simplifies reasoning – there's a single source of truth for the flat tokens during a run. It's a **micro-optimization** in the current scale, but it future-proofs the tool as usage grows.

- **Enhance Plugin Interface Abstraction:** To address the brittle incremental filtering, the architecture could introduce a more structured way to handle candidates. One idea is to have the Engine itself filter plugin outputs: e.g., Engine could call each plugin with the full engine state, get all violations, and then internally drop any violation whose involved tokens are not in the candidates set. However, that requires knowing which tokens a violation "involves" (which is partially given by fields like `id` or `nodes`). This could be complex if not all plugins populate those fields uniformly. Another approach is to formally document and enforce the `candidates` usage: e.g., switch the interface to something like `evaluate(engine, changedIds: Set<TokenId>): ConstraintIssue[]` and in guidelines stress that plugins should return issues related to those IDs only. Possibly even provide a base plugin class or utility that handles the common pattern (so custom plugin authors can call a helper to skip unaffected stuff). In absence of that, at least aligning documentation with implementation (rename docs to `evaluate` and mention the candidates filter) would reduce confusion. Overall, making constraint evaluation *purely functional* and consistent might mean giving up some incremental optimization or handling it in a centralized way. If performance allows, an alternative is simply to run all constraints on the full set every time and then compute a diff of new vs old violations outside (to decide what's new). That might simplify plugins (they'd always check everything), at the cost of more computation. The right balance likely is to keep the incremental approach but make it more robust by design: for example, ensure every violation includes which tokens it relates to, and let Engine do candidate filtering based on that meta-information.

- **Improve Responsibility Boundaries and Layering:** DCV could benefit from a clearer layering: **Data Loading Layer → Core Engine → Presentation Layer**. Right now, CLI mixes data loading and presentation, and core sometimes dips into data loading. A refactor to create a dedicated data loading module (or even just cleaner separation in CLI) would help. For example, one could have a function that given the user's GlobalOptions (paths, etc.), returns a fully prepared `Engine` plus any extra context (like the flat tokens for output). This function would call flatten, merge overrides, load config, register plugins, etc., all in one place. The CLI commands then simply invoke that and handle the results (printing or writing files). This would concentrate the orchestration logic that is currently spread out (some in validate.ts, some in engine-helpers.ts, some in breakpoints.ts) into a single coherent workflow. It would also ease implementing new frontends – e.g., if someone wanted a

VSCode extension or a GUI, they could call this “prepareEngine” function with appropriate parameters and get a ready-to-use Engine without reimplementing the loading logic.

- **Tackle Minor Tech Debt:** Aligning names and cleaning up the codebase would reduce confusion. Removing the compiled .js files from version control (if feasible) or clearly separating them into a `dist/` directory would make the repository cleaner. Updating documentation to reflect the current API and ensuring consistency (e.g., if `ConstraintIssue` uses `rule` field, the docs should use the same term) would help developers. None of these are structural changes, but they polish the architecture by making its use more straightforward and its behavior more transparent.

- **Optional: More Functional Core for Testability:** A deeper refactor (potential future consideration) could be to make the core engine even more functional. For instance, instead of building state inside Engine and then calling evaluate, one could have a pure function `validateTokens(tokens, constraints) -> ValidationResult`. Internally it would do the flatten, build engine, run plugins, and return issues, without requiring an Engine class that holds state across calls. The Engine class is currently stateful primarily to support incremental updates (it holds the current values and graph so it can apply commits). If incremental use-cases are not heavily utilized yet, one might simplify the core to a stateless validator and implement incremental caching as an optimization wrapper. However, given DCV’s intended use in design tooling, the ability to reuse an Engine for multiple commits is valuable. So rather than removing Engine, perhaps provide pure utility functions alongside it (which they partially do, like exposing `flattenTokens`, etc.). Ensuring those functions cover all needed steps means one can test each part in isolation (flattening logic, each plugin with a fake engine, etc.).

- **Better Surfacing of Constraint Sources:** To mitigate the “implicit file” issue, DCV could log or report which constraint files and config were loaded. For example, when running `validate`, if `themes/spacing.order.json` was found and applied, a verbose log or a note in the output could mention “Applied monotonic-spacing rules from spacing.order.json”. This is more of a usability improvement than a refactor, but it supports architecture transparency. It helps maintainers and users verify that the system is loading what they expect. In the code, this could be done in the engine helper – e.g., after reading each file, if found, `console.log("Loaded X rules from typography.order.json")` when verbose mode is on. This kind of instrumentation makes the dynamic aspects of the architecture more observable, which is important in a system where convention plays a role.

By implementing these suggestions, DCV’s architecture would become **clearer, more modular, and easier to extend**. We’d see all configuration in one place, core logic independent of environment, less duplicate code, and more confidence that plugins and engine are working in harmony. It would reduce the “mental overhead” for contributors who currently must track logic across core and CLI and multiple files. Moreover, these changes would position DCV well for growth – whether that’s more constraint types, integration into larger systems (like the mentioned DecisionThemes framework), or adaptation to new token formats – because the boundaries and extension points would be more well-defined. The end result would retain DCV’s strengths (its rigorous approach to validating design relationships) while streamlining its internal structure for maintainability.

Overall, DCV is a strong foundation; with some refactoring to address the above issues, it can be made even more robust and developer-friendly, ensuring it remains easy to reason about and evolve as design tokens

and constraints evolve. The goal of these improvements is to make the architecture **as elegant and reliable as the mathematical constraints it checks** – clear structure, no unnecessary repetition, and no surprises lurking in the dependency graph of the code itself.

---

1 2 3 4 5 6 7 8 10 11 12 13 73 74 116 117 Architecture.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/Architecture.md

9 14 README.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/README.md

15 16 17 18 19 87 index.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/index.ts

20 21 22 23 78 79 80 81 82 83 84 85 flatten.js
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/flatten.js

24 25 26 27 86 99 100 why.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/why.ts

28 29 30 96 118 119 120 130 monotonic.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/constraints/monotonic.ts

31 32 94 95 112 wcag.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/constraints/wcag.ts

33 34 threshold.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/constraints/threshold.ts

35 36 37 38 cross-axis.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/constraints/cross-axis.ts

39 40 41 42 43 44 poset.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/poset.ts

45 46 64 65 92 93 101 102 103 graph.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/commands/graph.ts

47 48 README.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/adapters/README.md

49 50 json.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/adapters/json.ts

51 52 53 54 55 css.ts

https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/adapters/css.ts

56 57 61 62 63 77 91 97 132 validate.ts

https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/commands/validate.ts

58 59 60 89 90 121 engine-helpers.ts

https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/engine-helpers.ts

66 67 68 69 why.ts

https://github.com/CseperkePapp/SingleLarissa/blob/3bfc775ff0de90710f5688288548088922882 6d0/cli/commands/why.ts

70 71 104 105 106 107 build.ts

https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/commands/build.ts

72 utils.ts

https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/commands/utils.ts

75 76 122 124 breakpoints.ts

https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/breakpoints.ts

88 110 111 115 engine.ts

https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/engine.ts

98 127 128 129 131 API.md

https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/API.md

108 109 113 114 123 cross-axis-config.ts

https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/cross-axis-config.ts

125 126 types.ts

https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/types.ts