ChatGPT

Dependency Graph of Modules

- **Core Engine (`core/`)** – Contains the validation logic and data structures:
- `flatten.ts`: Flattens nested token JSON into flat tokens and builds a directed graph of **dependency edges** (only token reference dependencies) [1] [2].
- `engine.ts` / `index.ts`: Defines the `Engine` class (stores token values, reference graph, and constraint plugins) [3] [4]. The engine is kept minimal – it doesn't itself know about specific constraint types.
- `constraints/`: Each **plugin** implements one constraint category (e.g. `monotonic.ts`, `wcag.ts`, `threshold.ts`, `cross-axis.ts`). They conform to a common interface and **register via** `engine.use()` [5]. Plugins are mostly independent functions, though **monotonic plugins reuse parsing helpers** (for px, number, lightness) and share types (importing an `Order` tuple type from `poset.ts`) [6]. No plugin imports the CLI or other high-level modules.
- `color.ts`: Utility for CSS color parsing, luminance, and contrast math (used by WCAG and cross-axis plugins) [7] [8].
- `poset.ts`: Utilities to build and manipulate a partial-order graph for constraint relationships (used for exporting Hasse diagrams and highlighting violations) [9] [10]. This is separate from the Engine's runtime graph.
- `breakpoints.ts`: Helpers to load breakpoint-specific token override files and constraint files. Contains logic to pick the appropriate JSON files (e.g. `themes/color.md.order.json` if present) [11] [12].
- `cross-axis-config.ts`: Reads a **rules JSON** and produces a combined cross-axis plugin. It parses the JSON structure into `CrossAxisRule` objects and logs debug warnings for unknown token IDs [13] [14].

- **Internal coupling:** Core modules mostly reference each other in a one-way fashion. For example, the monotonic plugin imports a type from `engine.js` and parsing from `color.js` [15]. The Engine itself remains decoupled from specific plugin implementations (it just stores any plugins added). There are **no cyclical imports** observed in core.

- **CLI Layer (`cli/`)** – Handles command parsing (via Yargs) and orchestrates the core engine:

- `dcv.ts`: CLI entry with Yargs definitions for commands **validate**, **graph**, **build**, **why**, **set**, **patch**, etc. [16] [17]. It wires each command to a handler function in `cli/commands/*`.
- `commands/*.ts`: Each command implementation loads inputs, calls core logic, and formats output. For example, `validate.ts` loads tokens, builds an engine, runs all constraints, and prints a summary [18] [19].
- `engine-helpers.ts`: Bridges CLI and core engine. This module is a bit of a **kitchen sink**: it flattens tokens and instantiates the Engine with all applicable constraint plugins. It reads multiple files directly (e.g. loads any `themes/*.order.json` files for monotonic rules) and attaches the corresponding plugins [20] [21]. It also injects default WCAG contrast checks here (more on that below).
- `config.ts` & `config-schema`: Load `dcv.config.json` or package.json settings (e.g. user-specified constraint rules) and validate the schema.

- `json-output.ts`: Assembles JSON output objects (for the `--summary json` or machine-readable results).

- **CLI ↔ Core coupling:** The CLI depends heavily on core internals (imports `flattenTokens`, all plugins, etc.), but core does **not** import CLI. This one-way coupling keeps the engine library usage possible. However, some CLI modules like `engine-helpers` know a lot about core plugin details (file names, plugin constructors), making it a mini-"god module" for assembling the engine. There is minor duplication (e.g. both `engine-helpers` and `validateCommand` load constraint files). Overall, separation of concerns is intact (core logic vs. I/O and presentation), with some tight connections in the CLI for expedience.

- **Adapters (`adapters/`)** – Format converters:

- `css.js`, `json.js`, `js.js`: Functions to emit the final tokens in CSS variables, JSON, or JS module format. These are used by the `build` command to output tokens in multiple formats [22]. They apply an optional mapping manifest for custom naming but otherwise just transform the flat `{ id: value }` map.
- *Input adapters:* The design mentions support for Style Dictionary or Tokens Studio formats, but this is handled implicitly. In practice, the code's `flattenTokens` expects an internal normalized shape (using `$value`). A separate adapter system for input isn't fully implemented (the `adapters/README.md` outlines goals, but e.g. there's no function automatically converting a Style Dictionary JSON to `$value` format at runtime). This suggests either the input JSON must already use `$value` keys or the adapter idea is not yet realized in code (potential drift from plans).

Import Relationships & Coupling: In summary, the module dependencies form a directed acyclic graph: **CLI → Core → (no further)**. The core engine and plugins are self-contained (no references to CLI or external frameworks). The CLI layer imports core functions and Node modules (`fs`, etc.) freely. There are no cyclical imports or obvious "spaghetti" usage; however, **the CLI does hardcode knowledge of core file names and plugin internals**, which is a form of tight coupling (e.g. assuming constraint JSON file locations and always adding certain plugins in code). No single module completely dominates the codebase (the Engine is small, and logic is spread across plugins and commands), so we don't see a classic giant "god object." The largest functions are the CLI command handlers like `validateCommand` (~150 lines) which coordinate many tasks in one place. These could be refactored for clarity, but the overall separation of concerns is logical.

Core Execution Paths from Entry Points

1. `dcv validate`: End-to-End Flow – The validate command is the heart of the tool's workflow: - **Token Loading:** If no custom path is given, it loads the default tokens file (currently `tokens/tokens.example.json`) and merges any override layers (local or breakpoint-specific) [18] [23]. It flattens this JSON into a flat map and dependency edges. - **Engine Setup:** The CLI creates an Engine via `createValidationEngine()`, which adds all relevant constraint plugins. This includes monotonic order plugins for any `themes/*.order.json` found (typography, spacing, layout, color) and any WCAG rules defined in config [20] [24]. Notably, in this path the code only adds WCAG rules if provided via config (it does **not** always add defaults here – see drift section). - **Cross-Axis & Threshold Plugins:** After engine creation, `validateCommand` attaches the cross-axis plugin and threshold checks *conditionally*: it always loads the global cross-axis rules file (`themes/cross-axis.rules.json`) and a breakpoint-specific one if running per breakpoint [5]. It also unconditionally adds a **ThresholdPlugin for touch target size** (hardcoded rule

requiring `control.size.min >= 44px` [25]. These are added for each validation run. - **Execution & Results:** The Engine's `evaluate()` is then called on **all token IDs** (or an affected subset in incremental mode) to collect violations [26]. The code separates errors vs warnings, and prints a summary. By default, output is human-readable text; if `--format json` or `--summary json` is used, it aggregates all issues into a JSON structure (including counts per breakpoint) [27] [19]. The exit code logic uses the `--fail-on` flag: e.g. `--fail-on error` causes a nonzero exit if any errors were found. - **Multi-Breakpoint Loop:** If `--all-breakpoints` is set, the above process repeats for each breakpoint (`sm`, `md`, `lg`), effectively treating each override set as a separate scope [28] [18]. Results can be summarized per breakpoint and totaled. The code measures timing per scope and (intended to) compare against any performance budget flags (though this feature isn't fully realized – see drift). - **Key Functions in Flow:** *Token flattening* provides the base data (`flat` tokens and edges). *Engine.commit* is not used in the bulk validate (only in interactive `set`), instead `engine.evaluate(allIds)` checks everything at once [26]. Each plugin's `evaluate()` runs over relevant tokens/pairs and produces issues. The cross-axis plugin, for instance, iterates its rules and only emits an issue if the rule's token(s) were in the "candidates" set (ensuring we only flag changes or, in full validate, everything) [29] [30].

2. `dcv graph`**: Generating Dependency/Constraint Graphs** – This command has two modes: - **Dependency Graph (default):** It flattens the tokens and takes the list of dependency edges (token references) and formats them in the requested output (JSON, Mermaid, or Graphviz DOT) [31] [32]. In this mode, it does not run the Engine or constraints – it's purely the token reference graph (useful for debugging token relationships). The CLI uses a helper to format nodes and edges; e.g., JSON output is an object with `nodes` and `edges` lists. - **Constraint Poset (`--hasse` flag):** If the user specifies `--hasse <name>`, the tool will read a constraint file (e.g. `themes/typography.order.json` if name is "typography") and build a **poset graph** of the order relations [33]. It computes the **transitive reduction** (to get a minimal Hasse diagram) [34] [35]. Options allow filtering by prefix or focusing on a subset of tokens. If `--highlight-violations` is used, the code will instantiate an Engine, load the same tokens and constraints, and evaluate any violations on those order relations to mark edges/nodes in red [36] [37]. (It manually applies the Monotonic plugin or MonotonicLightness depending on if the poset is color-related, and also the Threshold plugin for the touch target rule, to incorporate that into the diagram [38].) This is a complex path that mixes data gathering with on-the-fly validation to produce an annotated diagram. The output can be Mermaid or DOT (and even rendered to SVG/PNG by shelling out to Graphviz if requested).

In summary, `dcv graph` either dumps the raw token dependency graph or a constraint hierarchy graph. It doesn't produce violation text – it's more for visualization. The two modes use different code paths and underlying data (token references vs. static constraint definitions).

3. `dcv build`**: Token Build/Export** – This command does **no constraint checking**; it's purely to output tokens in various formats: - It loads the base tokens (and optional local and breakpoint overrides similarly to validate). If a `--theme <name>` is specified, it will additionally load `tokens/themes/<name>.json` and merge those values into the output set [39]. This allows building theme-specific token files (multi-theme support in a basic form). - After flattening, it collects all final token values in a dictionary `allValues` [40]. Then, depending on `--format`, it uses the adapter functions to emit CSS, JSON, or JS. By default, it writes to `dist/tokens.<format>` (or all three if `--all-formats` is set) [41] [42]. The build command also supports a `--mapper` (JSON manifest to map token IDs to custom output names), which it will load and apply for aliasing purposes [43]. If `--dry-run` is given, it prints the output to console instead of writing files [44]. - Essentially, `build` is like a mini style dictionary: take internal tokens and produce consumable

artifacts. It doesn't use the Engine at all. It does assume certain file locations (tokens file, theme files) much like validate does.

4. `dcv set` **: Interactive Token Editing** – This command leverages the Engine's **incremental update** capability: - It loads tokens (and config) similar to validate, and calls `createEngine()` (note: this path uses `createEngine` which *does* include some default plugins like WCAG, as opposed to `createValidationEngine`) [45] [24]. All constraints are thus in place on the Engine. - If `--theme <name>` is provided, it will load that theme's tokens and call `engine.commit()` on each token from the theme JSON to apply those overrides to the base Engine [46] [47]. This effectively "pre-applies" a theme before setting additional values. - The user can specify token assignments in two ways: as positional args (`tokenId=value`) or via a JSON/YAML file (`--json` flag) for batch updates. The code parses each expression or JSON structure into a list of `{id, value}` changes (with support for setting to null = unset) [48] [49]. - For each change, it calls `engine.commit(id, newValue)`. The Engine will update that one token's value and compute the **affected set** of tokens (all dependents in the reference graph) and immediately re-evaluate constraints on that small subset [50]. The result includes: - `result.patch`: a map of the changed token (and any *derived* tokens, if there were computed ones) to their new value [51]. - `result.affected`: list of other tokens that were impacted by this change (e.g. tokens referencing the changed token) [52]. - `result.issues`: any constraint violations detected among the changed token and its dependents [53]. - The CLI accumulates all patch changes into a `finalResult` object to output at the end (so if multiple tokens are set, it shows the net changes) [54] [55]. It prints each change and, if not in `--quiet` mode, reports any *immediate issues* ("Issues: ...") for that change [55]. For example, if you set a token to a value that breaks a monotonic order, you'll see an issue warning right away. - If `--write` is used, it will also persist the changes to the `tokens/overrides/local.json` file so they become a permanent override [56] [57]. (This uses helper functions `setDeep` / `deleteDeep` to update the nested JSON structure and save it.) - **Batch mode:** If a JSON of multiple values is provided, the code can apply them in one go and output a combined patch. There is logic to handle `$unset` markers in the JSON, etc., merging all changes before optionally writing to the override file [58] [59]. - The `set` command's use of `engine.commit` showcases the incremental validation: only the relevant constraints are checked (the plugin `evaluate()` methods internally skip tokens not in the candidate set). This makes interactive use efficient. The CLI even prints which other token IDs were affected by the change [55], leveraging the Engine's graph.

5. `dcv patch` **and** `patch:apply` **: Diffing and Applying Changes** – These two work together: - `patch`: This command computes the differences between an override set and base tokens. It can take a `--overrides` JSON (either a file path or inline JSON of flat overrides). Internally, it loads the base tokens, applies the override values to a clone, and flattens both to compare [60] [61]. It then produces a **PatchDocument** listing all changes: added tokens, removed tokens, and modified tokens [62] [63]. Each change has an `id`, old value, new value, and a type (`add`, `remove`, `modify`). The patch also includes a hash of the base tokens state, to detect drift later [64] [65]. The output is a JSON containing all these details. - `patch:apply`: This takes a patch (file or JSON string) and applies it to a tokens file. It loads the current tokens, verifies the base hash (warns if mismatch) [66], then for each change in the patch it either sets a token's `$value` or removes it (deletes the `$value` field) [67] [68]. The result is either printed (`--dry-run`) or saved to a file. Essentially this automates applying a diff. It ensures that removal means the token key stays in the JSON (but with no value) so that references to it don't break JSON structure [69]. - These commands don't run validation; they are utility for managing token files. They rely on the same flatten/merge logic but not on Engine or plugins.

Overall, the core execution flows follow the intended architecture: **parse inputs → (build Engine + attach plugins) → run validation → output results** for the validate path, and similar structured steps for other commands. The CLI commands are somewhat lengthy and do multiple tasks (parsing args, file I/O, invoking core, formatting output), but each major phase is visible in the code.

Architecture vs Implementation: Notable Drifts

Despite generally following the planned multi-phase architecture (normalization → graph → plugins → report), the codebase reveals several places where **implementation diverges from the documented architecture or expectations:**

- **Constraint Graph vs Engine Graph:** The architecture description says DCV builds a unified dependency graph with *both* token references and constraint relationships (edges for order rules, etc.) [70] [71]. In implementation, the Engine's `graph` only contains reference edges (built during token flattening). **Constraint relations are *not* added to Engine.graph**; they are enforced separately by plugins. For example, monotonic constraints don't create edges between tokens in the Engine – the plugin simply iterates configured pairs each run. The Hasse diagram generation for constraints is done outside the Engine using `poset.ts`. This means the code doesn't explicitly detect cycles in constraints via the Engine graph (only reference cycles are caught by flattening's resolution loop). The unified reasoning over a single "graph of everything" is therefore a bit split: references live in Engine.graph, constraint orders live in separate structures (posets in the graph command, or implicit in plugin logic). This is an **architectural drift** from the idea of a single DAG for all dependencies, but it doesn't break functionality – it just means the Engine doesn't natively know that, say, `typography.size.h1 >= typography.size.h2` is a dependency (so Engine's `.affected()` won't propagate along constraint edges, only reference edges).

- **Default Token Paths and Config:** Documentation (e.g. README FAQ) says that by default the tool will find `tokens.json` and `themes/*.json` in your project [72]. In code, however, the CLI defaults to `tokens/tokens.example.json` for most commands [73] [74]. This is a subtle drift – likely the intention was `tokens.json`, but the implementation uses a provided example path (perhaps left from development). It could confuse users following the docs. Similarly, the `--policy` flag shown in the README quick start (`npx dcv validate tokens.json --policy themes/policies/aa.json` [75]) does not exist in the CLI code – instead, the code expects you to use `--config` or just place the files in `themes/`. The configuration system can load `package.json`'s `dcv` field or a `dcv.config.json`, but the one-off `--policy` CLI shortcut isn't implemented, indicating a documentation-or-code mismatch.

- **WCAG Defaults & Config Inconsistency:** The design overview emphasizes built-in accessibility checks. In implementation, there's a discrepancy in how WCAG contrast rules are handled:

- In `engine-helpers.createEngine()` (used by `dcv set` and possibly programmatic API), **default WCAG contrast pairs are always added** (three default foreground/background combos are registered regardless of user config) [76].

- In `createValidationEngine()` (used by `dcv validate`), only WCAG rules from the user config are added – no default pairs are included [77] [78]. This means a plain `dcv validate` might

not actually check contrast at all unless the user has a `themes/wcag.json` or config entry, whereas interactive `dcv set` will warn about contrast on certain hardcoded roles. This divergence is likely unintentional. The architecture expectation was probably that some baseline contrast rules always apply (the README even implies default AA rules). The code implementation is inconsistent here, indicating an architectural intent that hasn't been cleanly realized in code.

- **Threshold Constraint Level:** DCV supports "Thresholds & Policies" (min/max values). The only implemented threshold is the touch target size (44px) rule. The architecture didn't explicitly state whether thresholds are errors or warnings, but the README example shows it as a warning ("WARN threshold control.size.min...should be >= 44px, got 30px" [79]). In code, the ThresholdPlugin defaults to `level: "error"` if not specified [80]. The hardcoded rule added in validateCommand does not set a level, so it becomes an **error** [25]. This is a drift either in documentation or implementation: likely they intended threshold violations to be non-fatal warnings (since 44px tap target is important but maybe not a strict error for CI). Currently, running `dcv validate` on the example tokens triggers an error for the 30px control size (which caused the CI to fail in strict mode until they used `--fail-on off`). This is a minor mismatch that could be clarified or adjusted.

- **Incremental Validation "Cache":** The docs tout *"incremental validation automatically detects changed tokens and only validates those tokens plus dependents... built-in – no configuration needed."* [81]. In practice, the CLI always reloads and re-flattens tokens fresh on each run of `dcv validate` (no persistent cache across runs). The **Engine.commit/affected mechanism** is used within a single session (e.g. `set` command or potentially a watch mode) but there's no caching between separate CLI invocations. So the "incremental" feature is only true in the context of an in-memory Engine that you reuse. The advertised behavior isn't actually implemented for the standard CLI validate (each run is full). If users interpret the docs as DCV magically speeding up repeated runs by skipping unchanged tokens, that's not happening in code currently. This is a gap between architecture promise and implementation – possibly something planned but not yet done (e.g. they could dump a state file to `.dcvcache` to persist results, but no such logic exists now).

- **"Why" Explanations:** The architecture highlights rich explanations ("why a constraint failed, highlighting the chain of values or references leading to the violation"). The CLI `dcv why` command is implemented, but its output is fairly minimal. It shows the provenance of a single token's value, its direct references, and immediate dependents [82] [83]. It does **not** list which constraints that token participates in or a chain of reasoning for a violation – contrary to what the System Overview suggests. The API docs even define a structure for `why()` including a list of constraints affecting the token and whether they are satisfied [84], but the current code does not produce that. For example, if a token failed a monotonic rule, `dcv why token` will show its value and references, but not explicitly say "it should be $\leq$ token X but is not." That part of the explanation system appears to be an **unfulfilled plan** in the implementation. (The needed data is partially there – e.g. violations carry `nodes` and `edges` context – but there's no code to integrate that into the why output yet.)

- **Plugin Extensibility:** The architecture calls the engine "plugin-based" and extensible, and a future plugin API is on the roadmap. In code, all constraint types are indeed implemented as plugins conforming to a simple interface, which is good. However, there is no dynamic plugin loading or plugin registration system beyond editing the source. For instance, you cannot drop in a new .js file and have DCV pick it up as a custom rule – you'd have to modify `engine-helpers.ts` or your own script to call `engine.use()`. This isn't exactly a drift (the feature may simply not be implemented

yet), but it's a gap between the *vision* of extensibility and the current state. The core is ready for it, but the hooks for users are missing.

- **Missing Config Hooks for Some Constraints:** Relatedly, some constraint categories are hardcoded in logic rather than driven by config files:

- Monotonic constraints *can* be provided via JSON files (the `themes/*.order.json`), which is as designed.
- WCAG constraints can be provided via a config JSON (and defaults were supposed to always run).

- **Thresholds, Lightness, Cross-axis:** Lightness constraints reuse the same `order` JSON mechanism (e.g. `color.order.json` is treated as a monotonic lightness rule by code when the id paths appear color-like) [85]. But there's no general "thresholds.json" or policy file ingestion. The one threshold rule is embedded in code. Cross-axis rules do come from JSON files (`cross-axis.rules.json`), which is good, but the config system does not expose an easy way to include threshold rules or additional simple rules outside writing a plugin or altering code. This indicates a bit of **architecture drift** – the system was meant to allow various constraint definitions via data, but currently it's a mix of data-driven (order, cross-axis JSON) and code-driven (threshold, default WCAG, etc.) enforcement.

- **Documentation vs Behavior Edge Cases:** A few smaller mismatches:

- The README suggests using `--strict` for validate (to fail on any warning) and `--fail-on` for CI gates. The code's handling of `--fail-on` is implemented, but `--strict` is parsed as an option without a clear effect (possibly intended to treat warnings as errors, but the implementation uses `failOn` for that). It's a bit confusing.
- The support for **multiple token files** (e.g. passing an array of tokensPath or themesPath in the API) is hinted in docs [86], but the CLI doesn't support multiple inputs at once. This might be an API-only feature (merging multiple token sources) that isn't clearly documented on the CLI side.
- **Breakpoints naming:** The code expects override files in `tokens/overrides/{sm,md,lg}.json` exactly [87]. If someone had a different breakpoint naming (say `xl`), there's no easy config for that – a mild deviation from a more flexible architecture where "any breakpoints are allowed via config." The docs don't mention ability to change those names, so perhaps not a drift but a limitation.

In summary, these drifts mostly concern configuration and extensibility: the core engine is doing what the architecture says (enforcing monotonic, contrast, etc.), but *how* those rules are supplied or toggled isn't as generalized as one might expect. Some defaults are hardcoded (contrary to the expectation of everything being data-driven), and some documented features either aren't implemented yet or behave differently (incremental caching, certain explanation details, default file locations). None of these break the fundamental operation, but they are areas where **the code deviates from the "ideal" architecture or the documentation**.

Hidden Invariants & Implicit Assumptions in the Code

Beyond the explicit constraints, the implementation has some **built-in rules and assumptions** that aren't immediately obvious from the high-level docs:

- **Default WCAG Checks on Specific Tokens:** As noted, the CLI engine setup unconditionally adds a few contrast checks if using `createEngine()` (currently those target common design token names like `color.role.text.default` on `color.role.bg.surface`, etc.) [88]. This acts as a safety net for users who don't specify any contrast rules. However, it's not documented that *"DCV will always check contrast for text vs surface, accent vs surface, etc."* If a user's design tokens use different naming, those default checks might do nothing (or could misfire if by coincidence a different token uses that name). It's a hidden invariant in code: DCV assumes those token IDs exist and represent the main text/background colors.

- **Always Enforcing 44px Touch Target:** Similarly, the single Threshold rule is effectively always on. Every `validate` run invokes a ThresholdPlugin for `control.size.min >= 44px` [25]. If the design tokens have `control.size.min`, it *will* be checked against 44px (and produce an error or warning if smaller). This happens even if the user provided their own policy JSON – there is currently no mechanism to turn off the built-in threshold. The docs do mention 44px as an example, but not that it's permanently hardcoded. It's a hidden assumption that 44px is a universal minimum (coming from accessibility guidelines). Organizations might want to tweak this (e.g. use 48px or remove it), which currently would require editing the code.

- **Unparseable Values Are Skipped Silently (Except WCAG):** The plugins have internal logic to handle unexpected token values:

- Monotonic plugins (`MonotonicPlugin` and `MonotonicLightness`) **ignore tokens that can't be parsed to numbers/lightness**. For example, if a token's value is `"none"` or some string that `parseSize` can't convert, the plugin will `continue` without flagging an error [89]. This means DCV will quietly skip rules if the values aren't suitable types. The idea is that a non-numeric value simply isn't subject to a numeric order constraint. This is reasonable, but not obvious to users – e.g., if a token meant to be a number is a string and thus unparsed, DCV won't warn about it, it will just not enforce the order for that pair.

- The WCAG plugin, by contrast, **does flag unparseable colors**. If a color value can't be parsed or if a referenced token is missing, it returns a warning issue saying "Unparseable color(s): …" [90]. So color constraints notify the user of bad data, but monotonic and threshold constraints do not – they fail open. This is an implicit design choice. It ensures DCV doesn't spam errors for tokens that aren't meant to be numeric (skipping gracefully), but it could also hide misconfigurations. This isn't documented in user docs; it's a code-level invariant about error handling philosophy (warn on color parse issues, skip on numeric parse issues).

- **Ordering of Constraint Loading:** The `engine-helpers.createEngine()` function reads constraint files in a fixed order: typography, spacing, layout, color – and then adds plugins in that sequence [20]. While probably not an issue, it means if there were any interactions or naming overlaps, that's the order of precedence. Also, it reads those files from a hardcoded relative path (`themes/…`) with no fallback. The code assumes your constraint files live in a `themes` directory

next to where you run the command. This convention isn't enforced or explained externally (except a hint in the FAQ), but it's an implicit expectation in the code.

- **Dependency on Token Naming Conventions:** A subtle invariant is that DCV treats token *IDs* as meaningful in some cases:

- The default WCAG and threshold rules target specific ID substrings (as mentioned).

- The color monotonic plugin assumes that if you supply "color" order rules, it should use the lightness comparator. The code decides to use `MonotonicLightness` for `colorOrders` specifically [91]. This implies that the constraint JSON for colors is expected to be named `color.order.json` and contain color token ids – then it switches parsing logic. If one had a numeric scale in a file named color.order.json, DCV would still treat it as lightness and perhaps mis-evaluate. This is an implicit binding between the file name (or the variable used) and the plugin behavior. It's not externally documented, but important internally.

- **Unknown Tokens in Rules Don't Fail Validation:** If a constraint JSON (especially cross-axis rules) references a token ID that doesn't exist in the tokens, DCV will not throw an error. The cross-axis loader logs a debug message about unknown IDs and even suggests similar IDs (using Levenshtein distance) [92] [93], but by default this output is only shown if you run with a debug flag. In normal runs, it will silently skip rules involving missing tokens. So an invariant is that *constraint definitions are optional and permissive* – DCV assumes missing tokens simply mean the rule is not applicable. This prevents crashes if, say, a design system doesn't have a token that a generic policy expected, but it might also hide a mistake (typo in a token name inside a rules file). The user guide doesn't mention how unknown IDs in constraints are handled, but the code chooses resilience over strictness here.

- **Provenance and Overrides:** The `why` command merges an optional `themes/theme.json` and `tokens/overrides/local.json` into the explanation context [94]. This means DCV has a concept of a "theme layer" and an "override layer" for provenance, even if you didn't explicitly provide them. It will categorize a token's source as "override", "theme", or "base" by checking if its id appears in those JSONs. This is a hidden feature – the user might be unaware that placing values in `tokens/overrides/local.json` automatically flags them as overrides in DCV's eyes. It's not harmful, but it's an implicit contract about file locations if you want the provenance tracking to work.

- **Engine Commit Behavior:** When using the Engine incrementally (e.g. via `commit` in `setCommand`), note that **only reference dependencies propagate**. If you change a token that has no tokens referencing it, `Engine.affected()` will return an empty set [52], so only that token's constraints are checked. If that token participates in a monotonic rule with another token (but there's no direct reference between them), the other token won't be in the affected set. The monotonic plugin handles this by considering the changed token as a candidate and still checking the pair (since it checks if either token in the pair is in the candidate set) [95]. This is a subtle invariant: **the plugin logic compensates for the Engine's graph being reference-only.** Users don't see this directly, but it's why incremental checks still catch order violations even though the Engine's own dependency graph didn't link those tokens. It's a clever internal invariant that makes the incremental validation work for constraints. However, if a future plugin isn't careful with candidate handling, it could miss things in commit mode.

In short, the code contains **built-in rules (contrast pairs, min size) and behaviors (skip unknowns, skip unparseable)** that aren't immediately toggled or visible in user configs. These reflect design decisions to make the validator robust out-of-the-box (e.g. always enforce a few critical rules, don't crash on incomplete data). They're not necessarily "bad," but they **aren't obvious from the architecture docs**. Users might only discover them by reading the source or noticing the behavior (e.g. "oh, it's checking a 44px rule I didn't configure"). Documenting these defaults and assumptions would improve transparency.

Risk Areas for Future Extensions

Looking ahead to planned extensions (multi-theme, breakpoint overlays, DTS integration), a few parts of the current design could become pain points:

- **Multi-Theme Support:** Currently, DCV can handle different themes by running separately or by using the `--theme` option in `build` to output theme-specific tokens. But there's **no mechanism to validate multiple themes together** or ensure consistency across themes. Each run of `validate` considers one token set (plus optional breakpoint overrides). If in the future we want to validate a design system that has, say, Light and Dark theme tokens simultaneously (ensuring both meet constraints, or even that Light and Dark have certain relationships), the current architecture doesn't directly support that. You would need to run `dcv validate` twice and manually compare results. Possible risks/needed changes:
- The **config and file conventions** assume a single active theme. E.g. `themes/*.json` holds constraints presumably for one theme. Multi-theme might require namespacing constraints or tokens (like theme A vs theme B). The code doesn't have a concept of "current theme" beyond the overrides merge in `build`.
- If Decision Themes Studio (DTS) wants to use DCV to validate all themes in one go or ensure parity, DCV might need a feature to load multiple token files at once. The API has `tokensPath: string[]` in its types [86], hinting at multi-file support, but this isn't fleshed out in CLI. Without careful design, trying to merge two themes' tokens could cause false violations (since constraints might fire comparing tokens across themes unintentionally).

- **Trade-off:** The simplest approach is to keep validating themes independently (which is what DCV does now). That avoids complex interactions. This is fine for now, but if "multi-theme coherence" becomes important (for example, ensuring dark and light themes both have monotonic scales independently), DCV might just delegate that to multiple runs or require the user to script it. There's a risk of attempting to shoehorn multi-theme into the current single-Engine model and getting it wrong. Probably, treating each theme separately (one Engine per theme) is the intended route – which the current design can handle, but automation around that is minimal.

- **Breakpoint Overlays and Responsive Design:** DCV already supports breakpoint-specific token overrides and can validate each breakpoint in turn. However, there is **no cross-breakpoint constraint checking**. In responsive design, one might have rules like *"the mobile value of X should be* $\leq$ *the desktop value of X"* or ensure scales progress across breakpoints (often called "fluid" or "ramp" consistency). The current plugin set doesn't cover that. If in the future we want to enforce consistency across breakpoints, it would require new logic:

- Possibly new cross-axis rules that reference tokens with breakpoint qualifiers (DCV would need to treat `token[sm]` vs `token[lg]` as separate IDs or have a way to compare them). The current

flatten merges overrides into one flat set rather than keeping multiple versions of a token in memory. So to compare breakpoints, the architecture might need to change to represent multiple states of tokens simultaneously.

- The risk here is complexity: extending the model to multi-dimensional tokens (theme dimension, breakpoint dimension) could complicate the graph and plugins significantly. It might be out of scope for DCV (and instead handled by a higher-level tool like DTS). For now, DCV assumes each validation scope is one theme at one breakpoint. If the user tries `--all-breakpoints`, DCV just validates each independently and summarizes – it doesn't link the results or enforce relations between them.

- Another minor risk: the code's merging of overrides doesn't deep-clone the token objects, it merges in place and re-flattens. If the same Engine were reused across breakpoints (it isn't in CLI, but hypothetically), leftovers from one merge could carry to another. The current CLI avoids that by making a fresh Engine per breakpoint, so it's fine. But if someone misused the API to validate multiple breakpoints sequentially with one Engine, they might get incorrect results. The design isn't meant for that, so it's just something to be aware of in extension.

- **Integration with Decision Themes Studio (DTS):** DTS will provide a UI and a "5-axis framework" (Tone, Emphasis, Size, Density, Shape). Likely it will use DCV under the hood for constraint validation. Potential risk areas in integration:

- **File System Dependence:** DCV's CLI and even some core parts (cross-axis loader, engine-helpers) read JSON files from disk directly [21] [96]. In a studio environment, tokens and rules may be managed in-memory or via an API, not as local files. To integrate cleanly, DCV's core might need refactoring to accept already-loaded data. Some parts already do (you can call `flattenTokens` on an object, or use the programmatic API passing objects). But others, like `loadCrossAxisPlugin(path)` strictly takes a file path. This will need to be adapted to avoid file I/O when used in a browser or a cloud service. It's a manageable change (they could expose a function to directly load rules from an object), but currently it's a tight coupling to the filesystem.
- **Real-time Feedback and Performance:** In DTS, as designers tweak values, DCV might be running continuously. The Engine is fairly efficient for moderate token sets, and the commit/affected mechanism is designed for this. A risk is if certain plugins don't scale or do redundant work. For instance, the cross-axis plugin currently checks *all* its rules every time `evaluate()` is called, gating by candidates internally [29]. If there are hundreds of cross-axis rules, that might be a bit slow on every keystroke. There may need to be optimization or selective execution (e.g. if a rule doesn't involve any changed tokens, skip it). The groundwork is there (the candidates set), but one must ensure each plugin uses it properly. Monotonic and cross-axis do check candidates; WCAG does too (it skips if neither fg nor bg changed) [97]. So performance should scale linearly with number of rules and changed tokens, which is good. The risk is more on the DTS side ensuring it calls the right incremental methods (not reloading everything each time).
- **DTS-specific Rules:** DTS's 5-axis framework might introduce new composite rules (e.g., tying tone to contrast, or density to spacing). DCV's cross-axis plugin is quite flexible (it can express conditional rules relating different tokens). It might handle many of those cases if configured. However, if DTS requires more complex logic (say involving more than two tokens at once, or non-numeric conditions beyond the current operators), DCV might need enhancements. The risk is that the current CrossAxisRule schema is somewhat limited (one "when" and one "require" clause, or a contrast ratio clause) [98]. If we need to enforce something like "if X and Y, then Z", that would either require

multiple rules or extending the JSON format. It's doable but adds complexity. The plugin system can handle custom code, but the goal would be to keep things data-driven for DTS to configure easily.

- **Parallel or Batch Processing:** If DTS wants to validate many themes or scenarios concurrently (e.g., on a server for all combinations), spinning up multiple Engine instances might be needed. The Engine is lightweight, but loading large JSON and computing graphs for each could be heavy. There's no known thread-safety issue (everything is single-threaded JS), but just performance wise it's something to test.

- **Feedback loop and Overrides:** DTS likely will use the `set` / `commit` approach to adjust tokens and get immediate feedback on violations (much like the CLI `dcv set`). One consideration: the current implementation's reliance on global file paths for saving overrides won't apply in DTS. They will manage state differently. But that's fine – DCV core doesn't require writing files; that's purely CLI. The risk is minimal here, just ensuring the integration uses DCV's API rather than CLI.

- **Complex Constraint Combinations:** As new use-cases emerge (multi-axis, multi-theme), interactions between constraints could become tricky. For example, if a token fails a monotonic constraint and a cross-axis constraint simultaneously, DCV currently reports both separately. That's okay, but if in DTS they want to present a more unified diagnosis, they might need to correlate issues. The current code doesn't attempt to correlate or prioritize violations (no concept of "this is the root cause vs secondary"). It simply lists all. This is philosophically fine (a violation is a violation), but in a UI context, it might overwhelm users. Addressing that would be more of an enhancement than a refactor – e.g., grouping related violations or suppressing cascading failures – but it's something to consider as a risk to usability rather than code correctness.

In summary, **the architecture holds up for the current scope**, but to extend it: - Handling multiple themes in one go will likely require running separate validations and comparing outcomes (outside DCV's current responsibility) or expanding config to differentiate theme-specific constraints. - Breakpoint overlay relations are not handled and would need new features (which could complicate the model). - The tight coupling to filesystem in a few spots is an easy technical fix to enable broader integration. - Performance and clarity in a real-time environment seem okay, but should be validated with large rule sets. - The good news is the core is quite modular (plugins) and stateless (you can create many Engines). The risks are more about **missing features or conventions** that might be needed, rather than problematic existing code that would break.

Recommendations & Proposed Improvements

To align the codebase with its architecture and future needs, I suggest the following:

1. Refactor Constraint Loading for Clarity and Extensibility: Instead of scattering logic across `engine-helpers` and `validateCommand`, provide a single cohesive way to load constraints: - Introduce a **"ConstraintLoader" module** that reads all constraint JSON files from a directory (or multiple dirs) and returns an object with all applicable rules (orders, cross-axis, thresholds, etc.). This could be driven by a config file or naming conventions. For example, it could pick up any `*.order.json` for monotonic, `*.rules.json` for cross-axis, and perhaps a `thresholds.json` if introduced. This central loader can replace the multiple `loadOrders` calls [99] and the hardcoded file reads [21] in `engine-helpers`. - Use this to make `createEngine` and `createValidationEngine` share code or even merge into one. The distinction between them (adding default WCAG or not) can be controlled by flags or config rather than separate functions, which reduces inconsistency. This way, **all entry points use the same constraint**

initialization process, avoiding scenarios where one path forgets to add a plugin. - This refactor will reduce the coupling in `engine-helpers` (which currently knows file paths) and pave the way for user-extensible constraints: e.g., if tomorrow we add a plugin for "color harmony", the ConstraintLoader could detect `color.harmony.json` and load it, rather than baking that into engine-helpers.

2. Expose Configuration for Built-in Rules: The default WCAG pairs and threshold rule should be configurable or at least documented. Ideally: - Allow users to disable the built-in threshold or change its value without editing code. For instance, support a `constraints.thresholds` array in `dcv.config.json` (similar to how WCAG rules are passed) so one can override or add thresholds. If the config is present, use those; if not, maybe keep the default. Or simply document that "control.size.min 44px" is always enforced and consider making it always a warning to reduce disruption. - The default WCAG checks (text vs surface, etc.) could be moved to a sample config file (e.g., `themes/policies/aa.json` contains them). In CLI, we could automatically include `themes/policies/aa.json` constraints if found and no other policy is provided. This makes the behavior explicit (the user sees the file with those rules) rather than hidden in code. It also aligns with the docs that mention example policy JSONs. - By making these defaults data-driven, we also solve the inconsistency between `createEngine` and `createValidationEngine`. Both would simply load the same default policy if none specified. - **Trade-off:** Making things configurable adds complexity for the user who doesn't care – but we can ship with sane defaults (like an included `aa.json` policy) so that out-of-the-box behavior remains the same. It just becomes more transparent and flexible.

3. Align Documentation with Implementation (or vice versa): It's important to reconcile the discrepancies: - Update the README/Docs to reflect the real default behavior (mention the `tokens/tokens.example.json` if that remains, or better yet change the default to `tokens.json` to match expectations). If `--policy` isn't going to be implemented, remove that from examples and instead explain how to use `dcv.config.json` or place files in `themes/`. - Document the 44px touch target rule explicitly as a built-in check and clarify its severity (if it remains an error in code, docs should show it as an error, or change code to make it a warning as docs showed). - Clarify the incremental validation note – perhaps rephrase it to say *"DCV supports incremental validation in watch mode or via the API (Engine.commit), but each CLI run validates the entire set by default."* Alternatively, implement an actual caching between runs (e.g., store a hash of last validated tokens and skip unchanged, though that could be overkill and brittle). It might be safer to tone down the claim in docs for now. - Provide a section on "Default Constraints and Assumptions" in the docs – listing the default WCAG checks, threshold rule, and the expected token naming (so users know to have `color.role.text.default` etc., or else those checks won't find their tokens). - Align the API docs with the real output. For example, the `why()` output structure in docs suggests listing related constraints – either implement that or remove it from docs to avoid confusion. Given effort, it'd be great to implement: e.g., after collecting `issues` in validate, we could annotate each token with which rules it broke or passed. But that's non-trivial to do quickly. So probably update docs to reflect current behavior, and keep the richer explanation as a future feature.

4. Enhance Testing in Key Areas: Add or improve tests for scenarios that are likely in the extension areas: - **Cross-axis rules:** Create a test where a `cross-axis.rules.json` is loaded and ensure that violations are caught and that unknown token references log a suggestion (perhaps capture the debug output). This will give confidence that the complex rule parsing works as intended. Currently, there's no explicit test covering cross-axis logic. - **Threshold violation behavior:** Write a test for a token that violates the 44px rule and confirm it appears in results as expected (and with correct level). This ensures that if we decide it should be a warning, the test will lock that behavior. - **Multiple breakpoints:** There is a test covering `--all-`

`breakpoints` summary output, but we could add one that verifies per-breakpoint results differ when overrides differ, etc. Also test that `--breakpoint sm` (single) works. - **Engine API usage:** If not already, test the direct `validate()` API with tokensObject/policyObject as the docs show, to ensure no regressions. This can catch if, say, the default policy isn't being applied in that pathway. - **Integration test for multi-theme (if feasible):** Perhaps simulate two themes by calling validate twice and comparing outputs, or test that using `--theme` in build includes correct values. This isn't directly DCV's responsibility to compare themes, but at least ensure that using `--theme` doesn't break validation (currently, `validate` has no `--theme` concept – maybe we should add one?). If supporting `--theme` for validate is desired (to auto-merge a theme overrides file before validating), we should implement and test that. That would be a quick win for multi-theme: e.g., `dcv validate --theme dark` could internally do what build does (merge tokens/themes/dark.json then validate). Right now, users would have to swap token files or use overrides mechanism manually.

- **(Optional) Implement Plugin API Hooks:** As a longer-term improvement, define a way for custom constraints to be plugged in. This might involve:
- Defining a stable interface for plugins (we have `ConstraintPlugin` type already) and allowing users to register them either via the config file (pointing to a module path) or by extending the CLI. This is likely low priority until someone asks for it, but it aligns with the open architecture concept. One simple step: provide an official way to turn off certain built-in plugins via config (like `{"plugins": {"threshold": false}}` to disable the threshold check, etc.).
- Considering the roadmap mentions a plugin API, this refactor would make DCV more flexible for unknown future rules without editing core. Risk is complexity in loading arbitrary code, but even supporting user-supplied JSON-based rules (beyond what cross-axis can express) could be valuable.

5. Address Pragmatic Shortcuts that are okay for now but worth keeping an eye on: - The use of synchronous file reads in the CLI (like `require('node:fs').readFileSync` in engine-helpers [21]) is fine in a CLI context (token sets are small, and it simplifies logic). However, as DCV might be used in a watch mode or integrated into other tools, these blocking calls could be replaced with non-blocking or moved out of hot paths. It's not urgent since performance is likely dominated by actual validation, but for polish, converting these to asynchronous (or at least clarifying that CLI is I/O bound anyway) could be done. - The Engine is currently recreated on each validate. If incremental caching were desired, one could reuse the Engine across runs if the token file hasn't changed (keeping the graph in memory). This would complicate the CLI (statefulness in what is now a stateless run). Probably not worth it unless performance proves to be an issue with thousands of tokens. So this shortcut (recompute every time) is fine now – just be aware if performance tuning is needed later, caching the flattened graph could be an avenue. - The **monotonic plugin skip logic** (not flagging unparseable values) might be too lenient. If a token meant to be numeric isn't, perhaps the user should know. We could consider adding a warning for that scenario (similar to how WCAG does) in the future. This is a quality suggestion: it would catch errors in token data (like a typo "1..5rem" that gets skipped). It's a trade-off between strictness and flexibility. Currently the shortcut is to silently ignore – safe but possibly hiding issues. We should gauge user feedback on whether that's desirable. A middle-ground: when `--strict` mode is on, treat unparseables as warnings or errors. This would align with the notion of strict validation. - **Unknown IDs in rules:** Similarly, currently only shown in debug logs. It might be better to surface a non-fatal warning if a constraint file lists a token that doesn't exist. Otherwise a user might think a rule is enforced when it's actually not doing anything because of a typo. Perhaps when loading cross-axis rules (or orders), collect unknown IDs and print a one-line warning "(X rules skipped due to unknown tokens: …)". This keeps validation non-breaking but informs the user. It's an enhancement to consider so that the "silent skip" invariant becomes an explicit notification. It's a fairly

small change in `loadCrossAxisPlugin` to gather unknowns and return a message. - **Improve `why` output:** While perhaps beyond the immediate scope, adding constraint info to `dcv why` (as the docs envisioned) would be great. For example, if a token failed a monotonic rule, the `WhyReport` could include an entry like *constraint "typography.order" violated against token X*. We have the data (violations list) from the last run, but since `why` is its own command, it doesn't currently run a full validation. One idea: when running `why <token>`, we could quietly run `engine.evaluate` for just that token's affected set to see if any constraints involving it fail. This could populate a "constraints" field. This would make `why` more informative (and closer to architecture intent). It's a bit complex but doable. At minimum, update docs or help text to set correct expectations if we can't implement it yet.

6. Multi-Theme and Breakpoint Future-Proofing: In design discussions, consider: - Implementing a `--theme` option for `validate` as mentioned, which simply loads `tokens/themes/<name>.json` on top of base tokens (just like build does) before validating. This low-effort addition would allow users to validate theme variants easily (e.g. `dcv validate --theme dark`). Under the hood, it can reuse the same logic from build (a couple of lines to merge in theme tokens) [100]. This doesn't solve cross-theme constraints, but it at least streamlines per-theme runs. - If cross-theme consistency rules are needed (like ensuring two themes have the same set of token IDs or certain relative differences), that might be outside DCV's scope (could be a separate linter or a feature in DTS). I'd recommend keeping DCV focused on single-set validation for now, to avoid over-complicating the engine logic. - For responsive cross-breakpoint rules, maybe wait for demand. If needed, one could extend the cross-axis rule format to allow specifying a `bp` context or referencing tokens in different breakpoints. The current code actually has a notion of `bp` in CrossAxisRule (it filters rules by a `bp` property so you can have rules that only apply to a specific breakpoint) [101]. This is already partially implemented (they do `if (r.bp && bp && r.bp !== bp) continue`), meaning one can include breakpoint-specific rules in the cross-axis file. That's useful (e.g. a rule only for mobile). It still doesn't compare across breakpoints, but it shows the system is aware of per-breakpoint rule targeting. So at least document that (power users might use it). - Monitor performance with breakpoints: currently for `--all-breakpoints`, DCV re-flattens tokens 3 times (for sm, md, lg). If token count grows, this triple work could be optimized by flattening once and then just overlaying differences. But given typical token set sizes, it's likely fine. Just keep it in mind if someone complains about speed.

By implementing these recommendations, we'd achieve: - **Consistency and transparency** – The behavior across CLI commands and config will be more predictable and easier to understand (no surprise built-ins that aren't documented). - **Extensibility** – New rules or plugins can be added via config or easily slotted in, which helps future-proof for things like custom org rules or new types (e.g., shape constraints). - **Maintainability** – A single code path for loading constraints reduces duplication (currently engine-helpers vs validate logic). Clearer separation of pure logic vs I/O will also make integration with other tools cleaner. - **User confidence** – When the tool clearly reports all it's doing (including skipped rules or default assumptions), users can trust it more in CI (fewer "why didn't it catch this?" mysteries).

Most of these changes are moderate in effort but high in payoff for aligning the implementation with the intended architecture and use cases. The core engine is strong; a bit of polish and restructuring around it will make DCV truly robust as it scales to multi-theme, multi-breakpoint design systems in the future.

---

1 2 flatten.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/flatten.ts

3 4 51 52 53 index.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/index.ts

5 18 19 23 25 26 27 28 validate.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/commands/validate.ts

6 15 89 95 monotonic.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/constraints/monotonic.ts

7 8 90 97 wcag.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/constraints/wcag.ts

9 10 34 35 poset.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/poset.ts

11 12 breakpoints.js
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/breakpoints.js

13 14 92 93 96 101 cross-axis-config.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/cross-axis-config.ts

16 17 73 74 dcv.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/dcv.ts

20 21 24 76 77 78 85 88 91 99 engine-helpers.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/engine-helpers.ts

22 39 40 41 42 43 44 100 build.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/commands/build.ts

29 30 98 cross-axis.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/constraints/cross-axis.ts

31 32 33 36 37 38 graph.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/commands/graph.ts

45 46 47 48 49 50 54 55 56 57 58 59 set.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/commands/set.ts

60 61 62 63 64 65 patch.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/core/patch.ts

66 67 68 69 patch-apply.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/commands/patch-apply.ts

70 71 Design Constraint Validator (DCV) – System Overview.pdf
file://file-UAb4mBCNrsUZUnxiDWzXG6

72 75 79 81 87 README.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/README.md

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

82 83 94 why.ts
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/cli/commands/why.ts

84 86 API.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/API.md