The Formal Vocabulary of Layout Failure

A layout bug is rarely just a layout bug. It is usually a small visible crack in a larger chain of meaning, measurement, policy, and trust.

The original trigger was simple: a table header looked different on two consecutive pages. On one page the header appeared clipped or incomplete. On the next page the repeated header appeared fuller and taller. The source content had not changed. The renderer had changed the way it represented the same content.

That difference matters because readers do not debug documents. They believe them. A PDF either teaches the eye where things begin and end, or it makes the reader repair the document mentally. Good typography is not decorative correctness. It is cognitive honesty.

This document names the failure modes we observed and the principles they point toward. The point of naming is practical: a named bug can become a trace contract, an audit rule, an API object, and eventually an invariant.

1. Visual Symptom

The lowest level is the visual symptom. This is what the reader sees before anyone opens a trace file or reads a source span.

In the table-header case, the visible symptom was that the same table header looked different across fragments. Page one of the split table showed shorter or clipped labels. The continuation page showed fuller labels with different wrapping. The human report was not “the parser lost data” or “row height is wrong.” The human report was simpler and more reliable:

The repeated header does not look the same.

Formal names at this level include:

- inconsistent repeated table header
- header text clipping
- header truncation
- continuation header mismatch
- non-identical table header rendering

The important discipline is not to jump too early. A screenshot tells us what happened to the artifact, not yet why it happened. The text may still exist in the source. It may even exist in the PDF text stream. It may be painted outside a clip box, hidden

by a background, pushed below a row boundary, or wrapped under a row height that cannot contain it.

A good visual diagnosis records three facts:

1. What is visible.

2. Where it is visible.

3. What stable expectation was violated.

For the header example:

What is visible: repeated table headers differ.

Where it is visible: between the first table fragment and a continuation fragment.

Expected behavior: the same semantic header, under the same table grid and page target, should render through the same resolved header layout unless an explicit continuation policy says otherwise.

The visual layer does not solve the bug. It preserves the bug honestly enough that the system can investigate it.

2. Typography And Paged Media

At the typography level, the failure is a table-header repetition failure.

In paged media, long tables often split across pages. A continuation page usually repeats the header so the reader can keep the column meanings in working memory. That repeated header is sometimes called a running table header, continued table head, or repeated table head.

Repeating a header is not enough. The repeated header must preserve the table's schema. If the first fragment and continuation fragment appear to have different headers, the table starts lying about its own structure.

Formal names at this level include:

- broken running table header
- repeated table head mismatch
- table continuation header inconsistency
- unstable header row across page breaks
- continuation context failure

A table header is a local contract with the reader. It says: this column means this. If the labels are clipped, shortened, or differently wrapped in a way that changes comprehension, the

reader may think the schema changed. That is not a cosmetic failure; it is a meaning failure.

The right distinction is target-level versus fragment-level variation.

It is fine for the same table to wrap differently on A4 and trade paperback. The target changed.

It is not fine for the same table to wrap differently between page 4 and page 5 of the same target. The target did not change. The semantic object did not change. The fragment should not invent a new interpretation.

The typographic contract is simple:

The table may split. The schema must not.

3. Layout Engine

At the layout-engine level, this is fragmentation non-idempotence.

Idempotence means that the same operation, given the same relevant inputs, produces the same result. Here the relevant inputs are the table id, semantic header row, final column grid, font context, preset, and page target. If fragment zero and fragment one resolve different header geometry from those same facts, the operation is not stable.

A split table is one semantic object expressed as multiple page fragments:

- fragment 0: the first visible piece
- fragment 1: the next visible piece
- fragment 2: a further continuation

The bug appears when the fragments do not replay the same resolved header. One path may measure the original header. Another path may synthesize a repeated header. They look related in the code, but they are not the same authority.

Formal names at this level include:

- fragmentation non-idempotence
- continuation-fragment reflow divergence
- split-table header measurement drift

- geometry/content disagreement
- fragment-local recomposition bug
- first-fragment versus continuation-fragment policy split

A typical broken shape looks like this:

```
measure_original_header(table)
measure_repeated_header(table)
```

The two functions are almost the same. That is the trap. One includes padding. One does not. One uses bold metrics. One uses body metrics. One allows slash breaks. One treats slash-separated labels as unbreakable. One clips to reserved height. One clips to natural height.

The correct shape is:

```
canonical_header = resolve_header_layout(table,
 final_column_grid)

for fragment in table_fragments:
    fragment.header = replay(canonical_header)
```

Replay does not mean blind copy. The y-coordinate changes. The fragment phase changes. The page number changes. But the internal header facts must remain stable:

- cell text sequence

- cell count
- column x positions
- column widths
- padding
- line breaks
- row height
- baseline offsets
- clip bounds
- border policy

A useful trace should make the failure obvious:

```
1 table=node-89 header natural=78pt reserved=
  42pt fragment=0
2 table=node-89 header natural=78pt reserved=
  78pt fragment=1
```

That is not a vague “looks bad” finding. It tells us that content and width were stable, but first-fragment reservation was wrong.

The rule is: measurement, fragmentation, placement, and paint must share the same geometry. When they do not, the artifact becomes an argument between stages.

4. Data Model

At the data-model level, the failure is duplicated derived state.

The source Markdown contained one header row. But a renderer does not keep content in one form. It lowers the document through a chain:

- source text
- Markdown events
- semantic book model
- publication IR
- component IR
- measurement records
- fragmentation records
- placement records
- paint commands

- PDF text and glyph output

Derived state is necessary. The danger is not derivation. The danger is ungoverned derivation.

Formal names include:

- duplicated derived state
- stale derived view
- multi-source-of-truth layout state
- derived representation drift
- denormalized layout metadata mismatch
- semantic identity loss

The fix is not “never copy data.” A renderer must copy, lower, cache, and transform. The fix is to make authority explicit.

For a table:

- semantic content authority: the source or semantic IR
- column grid authority: the table layout solver
- header geometry authority: the resolved header layout

- fragmentation authority: the page-breaking solver
- paint authority: the resolved display list

The painter should not rediscover table-header wrapping from text. It should consume a resolved display list with stable source identity and stable geometry. The audit should then compare emitted instances against that authority.

A better record might look like:

```
ResolvedTableHeader {
    table_id,
    column_grid_id,
    cells,
    row_height,
    display_items,
    source_span,
    derivation_trace
}
```

Fragments can translate this record. They should not reinterpret it.

Semantic identity is not enough. Decision identity matters too. We need to know not only where the glyph came from, but why it has that width, that line break, that row height, that clip box, and that page.

5. Software Design And Connascence

At the software-design level, this is a connascence problem.

Connascence is coupling that forces two parts of the system to change together. Some coupling is inevitable. Hidden coupling is the problem.

In the table-header case, multiple places may "know" the same rule:

- how to measure the header
- how to wrap header text
- how to choose row height
- how to repeat a header
- how to clip a header
- how much padding a header cell has
- where rules and backgrounds are painted

If those rules live in multiple places, one will eventually drift.

Formal names include:

- connascence of algorithm

- connascence of execution
- connascence of timing
- connascence of position
- DRY violation in layout policy
- split authority
- parallel implementation of the same rule

Connascence of algorithm happens when two code paths implement the same logic independently.

Connascence of execution happens when one stage assumes another stage has already run and produced a usable result.

Connascence of timing happens when header layout is cached before final width exists, and the cache is not invalidated after width changes.

Connascence of position happens when code assumes row index 0 is always the header, but a fragmented table shifts or synthesizes rows.

The least-connascence design is to turn hidden rules into explicit data:

```
header resolver creates canonical header layout
fragmenter reserves canonical header height
painter replays canonical header display items
auditor compares emitted header instances
against canonical id
```

Then a change to header padding changes the style policy or resolver, not three separate fragments of near-duplicate code.

The practical test is:

“If I change this rule, how many places must change?”

If the answer is unknown, the system does not have architecture yet. It has folklore.

6. Invariants And Contracts

An invariant is a rule that must be true for every valid output.

For repeated table headers, the invariant is:

Given a table id, final column grid, font context, and page target, every repeated header instance must be geometrically and textually equivalent to the canonical header layout, except for explicit continuation markers.

Formal names include:

- fragment-stable header contract
- repeatable artifact invariant
- canonical header replay invariant
- measurement-paint consistency contract
- table schema continuity invariant

The invariant must be precise enough for machines:

- same text sequence
- same cell count
- same column positions
- same column widths

- same row height
- same padding
- same line breaks
- same baseline offsets
- same clip bounds
- same background and rule positions

There may be exceptions, but exceptions must be explicit:

- add a “continued” marker
- omit a repeated caption if the preset says so
- use a continuation seam instead of a terminal rule

Silent divergence is not an exception. It is a bug.

The best contracts appear at several layers:

```
before fragmentation:
    assert table.column_grid.resolved
    assert table.header_layout.resolved

during fragmentation:
    assert fragment.reserved_header_height ==
table.header_layout.row_height

during painting:
    assert paint_clip.height ==
fragment.reserved_header_height

```

```
after PDF emission:
    assert extracted_header_text(fragment_n) =
= canonical_header_text
    assert glyph_boxes_within_header_clip
```

Earlier failures are cheaper. Final artifact checks are still necessary because PDF rendering is not merely data transformation. It is a visual promise.

7. Testing And Audit

At the testing level, the table-header problem is a metamorphic visual regression.

A metamorphic test does not always need a single golden image. It can assert a relationship:

The same table header, under the same resolved grid, must be equivalent across all fragments.

Formal names include:

- metamorphic property failure
- visual regression
- snapshot instability
- cross-fragment consistency failure
- PDF extraction invariant failure
- layout oracle gap

A serious audit system needs several signals:

Trace audit:

- table id
- fragment id
- header layout id

- column grid id
- reserved height
- natural height
- fallback path
- canonical replay versus recomposition

Geometry audit:

- boxes stay inside bounds
- body starts below header
- repeated header height matches canonical height
- clip rectangles match reserved geometry

Text audit:

- source labels are present
- extracted header text matches expected labels
- body continuation text is not confused with header text

Pixel audit:

- overlaps
- clipped text
- false closure

- table overflow
- footnote collisions
- stranded headings
- avoidable underfull pages

Pixel audit alone says something is wrong. Trace audit says why. The useful system combines both.

A good finding is not:

```
page 4 header bad
```

A good finding is:

```
table=node-89
fragment=0
problem=header_reserved_height_less_than_natural_height
canonical_header_height=78pt
reserved_header_height=42pt
likely_stage=fragmentation
```

That is the difference between looking at a PDF and debugging a renderer.

8. False Closure At A Split Table

A table is not only a grid. A table is a promise: these facts belong together.

When a table crosses a page boundary, that promise has to cross with it. If the first fragment paints a strong bottom rule, then leaves a large white gap, the eye reads completion. The table may continue on the next page, but the page has already told the reader that the object ended.

That is false closure.

Formal names include:

- false closure
- premature closure
- missing continuation cue
- continued table ambiguity
- split-table continuation failure
- fragment boundary decoration mismatch
- continuation-aware decoration failure
- non-terminal fragment painted as terminal

This lives near CSS and paged-media ideas such as box fragmentation and box decoration across breaks. A fragmented block can either paint each fragment as a complete box, or paint non-final fragments as open continuations.

For tables, a complete box is often the wrong metaphor. A non-final table fragment should not look terminal.

Possible policies:

- extend table background to the bottom of the usable content area
- suppress or soften terminal bottom rules on non-final fragments
- use a continuation seam instead of a closing rule
- add an explicit “continued” marker when appropriate
- keep split-row backgrounds visually continuous
- avoid row splitting unless continuity is visually clear

This is not decoration for decoration’s sake. It is visual honesty. The page break is physical. The table is semantic. The renderer must respect both.

The trace contract should know the fragment phase:

```
table=node-143
fragment=0
phase=first
continues=true
```

```
terminal_decoration=false
continuation_background=
extends_to_region_bottom
split_row=true
```

The audit rule should follow:

```
if table_fragment.continues:
    assert not paints_terminal_bottom_rule
    assert continuation_cue_present
    assert visual_background_does_not_imply_fin
al_closure
```

A false ending is still an ending to the reader. Do not create one unless the object is actually done.

9. Avoidable Underfull Pages

The next failure is the large page-bottom gap that appears even though the beginning of the next text block could fit.

This is an avoidable underfull page.

Not every blank area is a bug. Books need air. Chapter openings may deliberately breathe. Footnotes may reserve space. Images and tables may be atomic. A heading must not be stranded. One orphan line of prose can be worse than whitespace.

The issue is narrower:

If the remaining main-flow space can fit a legal prefix of the next splittable block, and no stronger rule forbids it, the engine should not eject the whole block.

Formal names include:

- avoidable page-fill gap
- underfull page
- loose page break
- missed partial-fragment opportunity
- unnecessary block deferral
- legal-prefix placement failure

- page badness not charged strongly enough

The phrase legal prefix is the key.

For prose, a legal prefix might be two lines if the preset forbids a one-line orphan:

```
remaining_space ⩾ height_of(min_first_fragment_lines)
```

For headings, the legal prefix is not the heading alone. It is heading plus enough following payload:

```
remaining_space ⩾ heading_height + keep_with_next_payload
```

For code, one line is usually too weak. A legal prefix may be two or three lines:

```
remaining_space ⩾ height_of(min_code_fragment_lines)
```

For tables and images, the rules differ because the object may be atomic, row-splittable, cell-splittable, or float-like.

The useful audit finding is not “large whitespace.” It is:

```
page=8
remaining_main_flow=58pt
next_block=paragraph node-812
legal_prefix=2 lines
legal_prefix_height=34pt
```

```
decision=deferred_entire_block
finding=avoidable_underfull_page
```

Pagination is a negotiation between evils: widows, orphans, stranded headings, footnote collisions, ugly table splits, ugly code splits, and whitespace. Whitespace is not always wrong. Avoidable whitespace is wrong.

The human version is simple:

Do not make the reader turn the page before the page is done speaking.

10. Semantic Structure Must Be Authoritative

The screenshot with “Chapter 2” beginning mid-page exposed a different problem. The heading looked like a chapter title to the human reader. But to the renderer it was only a heading, because the ingest rule treated H1 as the chapter boundary and the generated manuscript used H2 headings for chapter titles.

That distinction is fundamental:

Heading is visual structure.

Chapter is publication structure.

A heading is something the reader sees. A chapter is something the book is. A chapter may contain a heading, but it is not reducible to one.

Formal names for this principle include:

- separation of content structure from presentation
- semantic structure versus visual structure
- semantic-first design
- semantic source of truth
- structural markup over presentational markup

- make illegal states unrepresentable
- lowering with preservation of intent
- avoid connascence of meaning

The core rule is:

Semantic structure must be authoritative; visual form must be derived.

If a block is a chapter, the system should not rely on every downstream stage rediscovering that fact from text like "Chapter 2." That creates connascence of meaning. The parser guesses. The running head generator guesses. The page solver guesses. The audit guesses. Eventually one guess differs.

The better model is:

```
source syntax → semantic classification →
publication IR → layout policy
```

Once ingest classifies a block as `Chapter`, downstream systems consume that semantic fact:

- page sequence policy starts it on a new page
- opener policy decides title treatment and drop-cap eligibility
- running matter uses its title

- bookmarks use its outline level
- counters may reset or increment
- footnote policy may reset if configured
- audit checks chapter-start invariants

This is not only cleaner. It is safer. It prevents a visible heading from pretending to be enough structure.

The important architectural boundary is between recognition and rendering. Recognition belongs near ingest or semantic classification. Rendering should not parse prose to recover intent.

In short: name the thing once, then carry the name forward.

11. A Structure Policy API

Markdown is flexible. That is useful for writers and annoying for renderers. One manuscript uses # for chapters. Another uses ## because # is the book title. Another writes `# Part I` and `## Chapter 1`. Another uses lines like `Chapter 7` with no Markdown heading at all.

We cannot be deterministic about author convention unless we make the convention explicit.

The existing system already has the beginning of this idea: `ChapterBoundary` says which heading level starts a chapter. That is good, but it is too small for the full problem.

The object I would want is a `DocumentStructurePolicy`.

Its job is to classify source blocks into publication objects before layout:

```
structure:
  title: first-h1

  divisions:
    - kind: chapter
      match:
        block: heading
        level: 2
        text_regex: '^(Chapter|Appendix)\b'
      page_start: next-page
      opener: chapter-opener
      running_matter: use-title
      counters:
        increment_chapter: true
```

```
        reset_footnotes: false

    - kind: part
      match:
        block: heading
        level: 1
        text_regex: '^Part\s+'
      page_start: recto
      opener: part-opener
```

The CLI can provide convenient sugar:

```
faux-press render book.md \
  --chapter-boundary h2 \
  --chapter-title-regex '^(Chapter|Appendix)\b' \
  --chapter-start next-page
```

But regex should be only one matcher inside a typed rule. Raw line regex is useful as a fallback. AST-level matching is safer because it sees headings, paragraphs, thematic breaks, front-matter, and directives as structured things.

The data model should include:

- `DocumentStructurePolicy`: the whole classification policy
- `DivisionRule`: a rule that creates a part, chapter, section, appendix, frontmatter, or backmatter object
- `BoundaryMatcher`: heading level, heading text regex, source line regex, frontmatter marker, explicit directive, or sequence pattern

- `DivisionKind`: book, part, chapter, section, scene, appendix, frontmatter, backmatter
- `PageStartPolicy`: same page, next page, recto, verso
- `OpenerPolicy`: title block, subtitle capture, epigraph capture, drop-cap eligibility
- `CounterPolicy`: reset or increment chapter, page, footnote, figure, and table counters
- `RunningMatterPolicy`: what feeds running heads
- `BookmarkPolicy`: outline level and label
- `BoundaryDecisionTrace`: why this source block became this publication object

The trace is essential:

```
source_line=42
block=heading
level=2
text="Chapter 3: Layout Engine"
matched_rule=chapter-h2
division_kind=chapter
page_start=next-page
reason=heading_level_and_regex
```

That is what makes the system debuggable. The PDF should not merely show that a chapter started on a new page. It should be able to explain why.

12. The System Shape We Want

A high-quality PDF renderer should not merely produce pages. It should produce explainable pages.

Every visible mark should have a path:

```
Source → Semantic → Publication → Component
→ Layout → Composition → Fragment →
Placement → Paint → PDF
```

Every consequential decision should have a reason:

- Why is this a chapter?
- Why did it start on this page?
- Why does this table continue?
- Why did this header repeat?
- Why does this row split?
- Why is this gap legal?
- Why was this code block split here?

- Why is this footnote on this page?

The architecture should follow a few rules.

First, classify semantics early.

Do not ask the page solver to infer chapters from text. Do not ask the painter to infer table continuation from geometry. Do not ask the auditor to infer intent from pixels alone.

Second, make derived geometry explicit.

Column grids, header layouts, row heights, code-shell metrics, legal prefixes, and footnote reservations should be named records with identities.

Third, make continuation a first-class state.

Tables, code blocks, paragraphs, and footnotes do not just “fit” or “not fit.” They fragment. Their fragments have phases: first, middle, last, only. Decoration and audit rules should know that phase.

Fourth, make page-fill decisions explainable.

If a page leaves 58pt empty, the trace should know whether the next legal prefix needed 34pt, 85pt, or 140pt. Blank space without explanation becomes suspicion.

Fifth, print worktree identity into render metadata and logs.

When multiple worktrees exist, every render should report:

- worktree path
- branch
- commit
- dirty status
- binary path
- source path
- output path
- target
- preset

This prevents a fix in one tree from being judged by a render from another.

The deeper principle is this:

The page is the final artifact, but not the final truth. The final truth is the derivation chain that can explain the page.

Appendix: Glossary

Visual symptom: The failure as it appears to the reader.

Running table header: A repeated table header shown on continuation fragments.

Fragmentation: Splitting one semantic object across pages or regions.

Idempotence: Same relevant input and context produce the same result.

Derived state: Intermediate data computed from source or earlier IR.

Connascence: Coupling where two parts must change together.

Connascence of meaning: Multiple layers independently infer the same semantic fact.

Invariant: A rule that must be true for every valid output.

Contract: A named boundary rule that stages must obey.

Metamorphic test: A test that checks a relation between outputs rather than one exact golden output.

False closure: A non-final fragment looks visually complete, so the reader thinks the object has ended.

Continuation cue: A visual signal that a fragmented object continues elsewhere.

Avoidable underfull page: A page leaves unnecessary blank space even though a legal prefix of the next splittable block could fit.

Legal prefix: The smallest typographically acceptable fragment of the next block, such as two paragraph lines, a heading plus payload, or a minimum code fragment.

Semantic structure: What the document is: chapter, section, table, footnote, appendix.

Visual structure: How the document appears: heading size, weight, spacing, rule, page position.

Structural markup: Markup that names meaning rather than only appearance.

Document structure policy: A typed classification policy that turns flexible source conventions into explicit publication objects.

Boundary decision trace: A record explaining why a source block became a chapter, section, appendix, or other division.