Improved Templating (“Blueprints”) – Product Brief (Draft)

This document accompanies `docs/2026 – april release x/improved-templating.requirements.md`.

It explains the idea in **non-technical terms**: what problem we’re solving, what the feature is, why it matters, and how we can deliver it safely in stages.

---

1. What We’re Trying to Achieve

We want the card creator to scale to **more card types**, more layout variants, and faster iteration—without needing to write a new bespoke “card preview component” every time we add or tweak a template.

Today, each template is effectively “hard-coded”: its layout exists as hand-written logic inside a dedicated component. That approach is fine while there are only a few templates, but it becomes a bottleneck as the app grows.

The proposed change introduces a simple concept:

> Every card template has a **blueprint**: a structured description of the card’s layers and where they go.

This blueprint becomes the “single source of truth” for layout.

---

2. The Problem (In Plain English)

Right now, card layouts live in code. That has a few predictable consequences:

- **Adding a new card type is expensive.** It usually means a new component,

new layout calculations, and new “special cases”.

- **Small layout tweaks require code changes.** Moving a box 10px is still a development task.
- **Templates drift.** Over time, each template develops slightly different rules for spacing, alignment, and edge cases.
- **Complex templates get complicated fast.** Hero/Monster-style layouts depend on content size (e.g. longer text pushes things upward), which encourages more custom math in more places.

In short: the tool can grow, but not as quickly or cleanly as we want.

---

3. The Proposal (High-Level)

Keep the existing concept of templates and template IDs.

Add a new piece of data per template: a **blueprint** that describes:

- What layers exist (background, image, icon, title, stats, description…)
- The order they are drawn (what appears on top of what)
- Where each layer goes (bounds and positioning rules)

Then introduce a **single generic renderer** that reads a blueprint and draws the card preview accordingly, using the same shared building blocks we already have (title ribbon, stats block, text block, images).

This is not “build a giant design engine”.

It’s a practical move from:

- “Layouts live in code”

to:

- “Layouts live in data (blueprints), rendered by one consistent renderer”

---

4. What Stays the Same (Important)

This approach is deliberately conservative:

- Templates still have stable IDs.
- The preview is still SVG-based, and exports still originate from the same SVG.
- We keep the existing card building blocks (title ribbons, stats blocks, text blocks).
- We are not trying to make templates fully user-editable in the first iteration.
- We are not trying to reinvent styling or theming as part of this change.

---

5. What Changes (From a User's Point of View)

In the first iteration: **almost nothing visible**. The cards should look the same.

The benefits show up as we add more templates and refine layouts:

- Faster delivery of new card types.
- More consistent spacing and layout across templates.
- Fewer "template-specific quirks".
- Better support for the "hard" layouts (like Hero/Monster) without bespoke code per template.

---

6. The Key Feature: "Layers" + "Groups"

The blueprint idea revolves around two concepts that map to how cards are already built:

6.1 Layers

A card is a stack of layers (for example):

- Background
- Artwork
- Icon
- Title
- Stats
- Description text

The blueprint simply lists these layers, in order, and describes where they go.

6.2 Groups (For the Hard Templates)

Most templates are essentially static: things are in fixed positions.

Hero and Monster cards are different because the bottom of the card is a "stack":

- Description text sits at the bottom (and can be short or long)
- Stats sit above it
- Monster icon sits above that (when present)

As the description gets longer, everything above it needs to move upward together, while the bottom stays pinned.

Blueprints support this with a "group" concept:

- A group is pinned to an edge (like the bottom).
- Its children stack upward (text, then stats, then icon).
- Optional items can appear/disappear without breaking layout.

This is important because it prevents us from scattering fragile "layout math" across multiple components.

---

7. Why This Is Worth Doing

7.1 It Scales Template Count

Adding a template becomes “define a blueprint and reuse existing parts” instead of “build a new bespoke template component”.

That makes it realistic to add the remaining HeroQuest templates and future expansions without ballooning maintenance costs.

7.2 It Improves Consistency

By introducing shared layout rules (like consistent padding and margins), templates stop drifting apart over time.

7.3 It Reduces Rework

Layout tweaks are common. When layout is data-driven, changes are localized to the blueprint rather than embedded in multiple places in code.

7.4 It Sets Up Future Features (Without Forcing Them Now)

Once templates have blueprints, future capabilities become possible:

- A “Custom card” with configurable layout options
- User-defined templates
- Sharing templates along with card exports

But we don’t need to commit to those today.

---

8. Addressing the Main Concern: “Isn’t This a Templating Engine?”

The intent is not to create an open-ended design system.

This approach is intentionally limited:

- It focuses on **layout positioning** (bounds, order, basic pinning) rather than full styling.
- It uses a small, known set of building blocks (existing card parts).
- It keeps “hard problems” contained (Hero/Monster bottom stack is handled with one group rule).

Think of it as:

> “Template blueprints” rather than “template programming”.

We want a tool that remains understandable and maintainable.

---

9. Rollout Plan (Phased, Low Risk)

Phase 1: Move existing templates onto blueprints

- Convert the existing templates to blueprint definitions.
- Start with the simplest templates (fully static layout).
- Add the group concept to cover Hero and Monster.
- Keep the old template components available as a fallback until we confirm parity.

Phase 2+: Optional future work (not required to ship Phase 1)

- Add a “Custom card” template that exposes safe configuration toggles.
- Add user-defined templates (save/share blueprints), if the project direction

supports it.

The important point is: Phase 1 provides value on its own.

---

10. Versioning and Data Portability (Why It Matters)

As the app evolves, templates may change. If we don't plan for that, old cards might render differently after an upgrade.

The blueprint approach makes versioning explicit:

- Cards record which template (and version) they were made with.
- Exports can include enough information to keep rendering stable across versions.

This becomes especially important if we ever introduce user-defined templates.

---

11. Collaboration Notes (For Multiple Developers)

This change helps align developers by separating responsibilities:

- Blueprints define layout (what goes where).
- The generic renderer handles consistent rendering rules.
- "Card parts" remain reusable building blocks.

This reduces the chance that two developers implement the same layout idea in two incompatible ways across different templates.

---

12. Summary

Blueprint-driven templates give us a path to:

- Add more card types faster
- Keep layout consistent
- Reduce ongoing maintenance costs
- Handle the complex Hero/Monster layout cleanly

...while keeping the project grounded, avoiding a full “engine”, and shipping value incrementally.