Closing the Usability Gap in Naked Objects

A Composable UI Primitive Approach

Anonymous Author(s)

Abstract

The naked objects pattern proposed that user interfaces could be generated directly from domain models, eliminating the cost of bespoke UI development while guaranteeing consistency between interface and business logic. Twenty-five years later, this promise remains largely unfulfilled—not because automatic generation failed, but because the generated interfaces lacked the spatial organization and navigational fluency that users expect. This paper argues that the design space for presenting structured information is far narrower than the software industry assumes, and that a small set of composable UI primitives—tiles, tile stacks, and recursively nested master-detail views—can cover it. We present Strvct, an open-source JavaScript framework implementing this approach, and show that centralizing the model-to-view pipeline produces emergent capabilities—transparent internationalization, annotation-driven persistence with cloud sync, and automatic schema generation—that would require significant per-component effort in conventional frameworks.

CCS Concepts: • **Software and its engineering → Object oriented frameworks**; *Model-driven software engineering*; • **Human-centered computing** → *Interactive systems and tools.*

Keywords: naked objects, automatic UI generation, model-driven development, composable primitives, Miller columns

1 Introduction

Most application frameworks treat the user interface as a separate engineering problem from the domain model. Each screen must be designed, implemented, and maintained independently. Every new model class or schema change propagates into view code, form layouts, navigation logic, and responsive design. This duplication is not incidental—it is structural, and its cost grows linearly with the number of domain objects.

Naked objects [6] proposed a radical alternative: expose domain objects directly to users and generate the interface automatically. Developers write only the domain model; the UI follows from it. This also guarantees structural consistency—the interface always reflects the actual state and shape of the model, because there is no separate representation to fall out of sync.

The pattern was described by Pawson and Matthews in 2000 and has been implemented in several frameworks, most notably Apache Isis (now Apache Causeway) [1] for Java. These implementations demonstrated the core thesis: automatic UI generation from domain models is feasible and produces functionally complete interfaces. Yet adoption has remained confined to internal tools, administrative interfaces, and prototypes. The most significant production deployment—Ireland's Department of Social Protection, which used naked objects on over 4,500 desktops to administer social welfare benefits—is itself an internal administrative system, reinforcing rather than contradicting this pattern.

The reason is not technical but perceptual. The generated interfaces—generic forms for objects, tables for collections, menus for navigation—are correct and complete but *feel wrong*. They lack the spatial hierarchy, navigational depth, and responsive behavior that users have come to expect. They resemble database administration tools more than the applications people use daily. This usability gap, rather than any limitation of the underlying pattern, has prevented naked objects from reaching its potential.

This paper describes an approach to closing that gap. We argue that the design space for presenting structured information is narrower than it appears, identify a small set of composable primitives that covers it, and present a framework—Strvct—that demonstrates the approach in a production application.

2 The Usability Gap

Prior naked objects implementations typically present each object as a form with fields for its properties, and collections as tables or lists. Navigation is handled through menus, links, or search. This strategy is functionally sufficient but creates four specific problems:

Lack of spatial hierarchy. Users expect spatial relationships to convey meaning: hierarchy expressed top-to-bottom, navigation depth expressed left-to-right, containment for ownership. Generic forms and tables flatten these relationships, requiring users to navigate through menus rather than perceiving structure visually.

No viewport adaptation. Modern applications invest heavily in responsive design—collapsing navigation, stacking layouts, hiding secondary content. Earlier naked objects implementations were desktop Java applications with no responsive design concept at all. Later web-based implementations such as Apache Causeway adopted CSS grid frameworks like Bootstrap, but their responsive breakpoints must be configured per-object via layout files or annotations—the responsive behavior does not generalize automatically across the domain model.

Inconsistent navigation depth. As users navigate deeper into an object hierarchy, form-based interfaces either replace the current view entirely (losing context) or open new windows (fragmenting context). Neither gives users a sense of where they are within the larger structure.

No visual continuity. Without a consistent spatial model, users cannot build a mental map of the application. Each navigation action feels like arriving at a new, disconnected screen rather than moving within a coherent space.

These problems are not inherent to the naked objects pattern. They are artifacts of a UI strategy—generic forms and tables—that prior implementations chose because it was simple and sufficient for their target audience. The question is whether a different strategy can retain the benefits of automatic generation while producing interfaces that meet the expectations set by modern hand-crafted applications.

3 A Narrow Design Space

Before proposing a specific solution, it is worth examining what those expectations actually entail.

When you survey informational interfaces across applications, websites, and operating systems, the same spatial conventions appear repeatedly: hierarchy expressed top-to-bottom, navigation depth expressed left-to-right, containment for ownership, and lists for collections. Consider three examples:

- **macOS Finder** (Miller Columns): a horizontal chain of list panels, each showing the contents of the item selected in the panel to its left. Selecting a folder reveals its contents in the next column. The spatial metaphor is depth as horizontal position.
- **iOS Settings**: a vertical list of categories. Selecting one pushes a new list onto a navigation stack, with a back button to return. The spatial metaphor is depth as screen replacement with a linear path.
- **Slack**: a vertical list of channels on the left, message content on the right, thread detail in a panel further right. The spatial metaphor is the same master-detail pattern at two levels of nesting.

These are three different applications built by three different teams for three different purposes, yet they use the same underlying spatial logic. This is not coincidence. These conventions are inherited from how we organize written information—the same top-to-bottom, left-to-right reading order found in Western text. They are so deeply embedded in interface culture that deviating from them creates confusion rather than innovation.

Bespoke UI developers are already converging on these patterns—unconsciously and inconsistently. The variation between hand-crafted interfaces is largely superficial: different visual styling, different spacing, different component libraries, but the same underlying spatial logic. Most of what distinguishes one application's navigation from another's, at the structural level, is accidental rather than essential.

This observation has a practical consequence: if the design space is narrow, a framework that applies these conventions uniformly may produce interfaces that are not merely acceptable but *preferable* to a patchwork of bespoke screens, because the user can rely on consistent navigation throughout the application. The framework's limitation—it cannot produce arbitrary layouts—is actually an advantage, because arbitrary layouts are precisely what creates inconsistency.

There are, of course, interfaces that fall outside this narrow space: data visualizations, design canvases, game renderers, media editors. These require domain-specific rendering that cannot be derived from model structure alone. But these are a minority of the screens in most applications. The majority—settings, lists, forms, inspectors, hierarchical browsers—are well within the space that a small set of composable primitives can cover.

4 Approach: Composable UI Primitives

Our approach is to define a small set of composable UI primitives that embody the spatial conventions identified above. Each primitive handles one aspect of presentation; composed together, they cover the navigational and layout patterns found in typical informational applications.

Note: the following diagrams illustrate view layouts, not their actual appearance.

4.1 Tiles

The fundamental unit of presentation is the **tile**: a view that presents a single domain object or a single property of a domain object.

Summary tiles present domain objects with a title, subtitle, and optional sidebars. They serve as the primary navigation element: selecting a summary tile reveals the object's contents in an adjacent detail area.

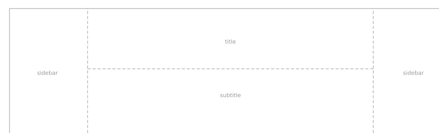

Figure 1. A summary tile presents a domain object with title, subtitle, and optional sidebars.

Property tiles present individual properties as key-value pairs, with optional notes and validation errors. Specialized property tiles handle common types—strings, numbers, dates, images, booleans—with type-appropriate editing interactions.

Tiles support gestures for direct manipulation: slide-to-delete, long-press reordering, and drag-and-drop between

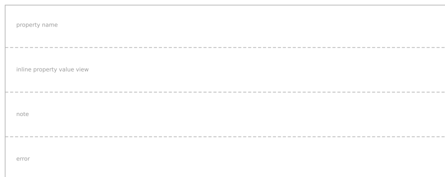

Figure 2. A property tile presents a single property as a key-value pair with optional note and error.

tile stacks or across browser windows. Domain objects register which MIME types they accept, enabling type-safe import and export through standard drag interactions.

Tiles can be subclassed for domain-specific presentation, but the default tiles are designed to be sufficient for the majority of cases. The goal is to make custom tiles the exception, not the rule.

4.2 Tile Stacks

A **tile stack** is a scrollable, ordered sequence of tiles presenting the subnodes of a domain object. Tile stacks support flexible orientation (vertical or horizontal) and gestures for adding, removing, and reordering items.

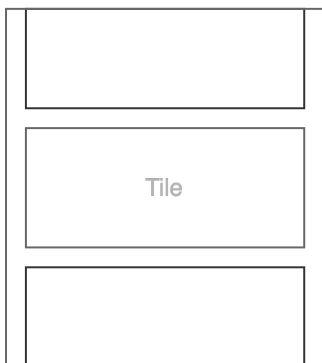

Figure 3. A tile stack: a scrollable sequence of tiles presenting the subnodes of a domain object.

4.3 Master-Detail Views

A **master-detail view** pairs a tile stack (the master) with a detail area that presents the currently selected item. The detail area itself may contain another master-detail view, enabling arbitrarily deep navigation through recursive composition.

Three features make this composition practical:

Flexible orientation. The detail area can be positioned to the right of or below the master, as specified by the domain object or overridden by the interface. This allows the same primitive to express both horizontal navigation (like a file manager) and vertical drill-down (like a settings panel).

Automatic collapsing. When the viewport is too narrow to display the full chain of master-detail views, earlier

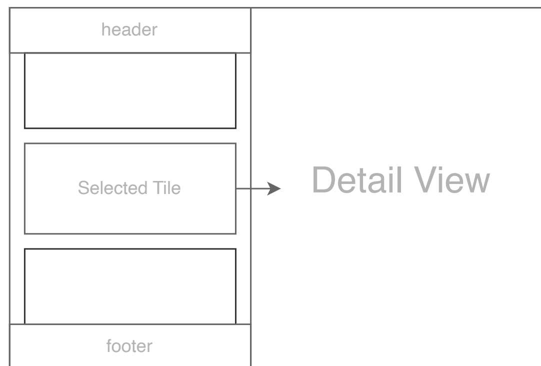

Figure 4. A master-detail view pairs a tile stack with a detail area for the selected item.

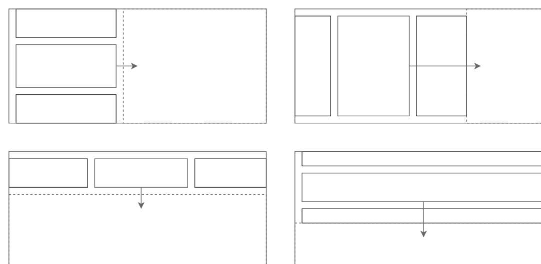

Figure 5. Master-detail views support horizontal (left) and vertical (right) orientations.

columns automatically collapse. A breadcrumb bar tracks the navigation path and provides back-navigation. The same structure works on a wide desktop monitor and a narrow mobile screen without any per-object responsive design.

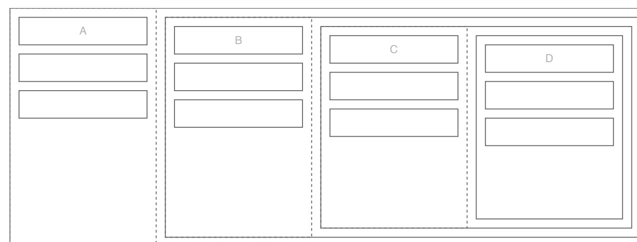

Figure 6. Expanded: all columns visible when viewport width permits.

Header and footer areas. The master section supports optional header and footer views for features like search, message input, or group actions, allowing common interaction patterns to be expressed within the same compositional framework.

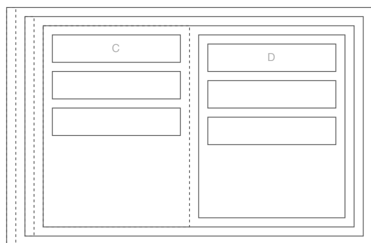

Figure 7. Collapsed: earlier columns automatically hidden on narrow viewports, with breadcrumb navigation.

4.4 Composition

Nesting master-detail views with varying orientations produces navigation structures that match many common application patterns: Miller column file browsers, settings hierarchies, email clients, chat applications, inspector panels. These are not special cases implemented individually—they are natural compositions of the same three primitives.

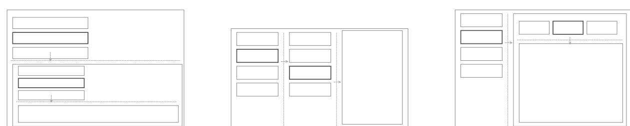

Figure 8. Three nesting patterns from the same primitives: vertical (left), horizontal (center), and hybrid (right).

This composability is the key insight. Rather than implementing a fixed set of application templates, the framework provides building blocks that compose to produce appropriate layouts for each part of the domain model. The Miller Column pattern [5] has been used since NeXTSTEP for file browsing; our contribution is making it recursive, orientation-flexible, and self-composing based on model annotations.

5 From Model to Interface

To make the "write the model, get the UI" claim concrete, consider a minimal domain class in Strvct:

```
(class Character extends SvStorableNode {

    initPrototypeSlots () {
        {
          const slot = this.newSlot("name", "");
           slot.setSlotType("String");
           slot.setShouldStoreSlot(true);
           slot.setSyncsToView(true);
           slot.setCanEditInspection(true);
        }
        {
          const slot = this.newSlot("level", 1);
           slot.setSlotType("Number");
           slot.setShouldStoreSlot(true);
           slot.setSyncsToView(true);
        }
        {
        const slot = this.newSlot("inventory", null);
           slot.setFinalInitProto(Inventory);
           slot.setIsSubnodeField(true);
        }
    }

    initPrototype () {
        this.setShouldStore(true);
    }

    subtitle () {
        return "Level " + this.level();
    }

}.initThisClass());
```

This definition contains no UI code, no form layouts, no navigation logic, and no serialization code. Yet it produces:

- A **summary tile** displaying the character's name as a title and "Level 3" as a subtitle
- **Property tiles** for `name` (editable string field) and `level` (editable number field), with appropriate input types
- A **navigable field** for `inventory` that, when selected, opens a new master-detail view showing the inventory's contents
- **Automatic persistence** to IndexedDB, with dirty tracking and transactional commits
- **Bidirectional synchronization** between model and view—editing a field updates the model; programmatic model changes update the view
- **Automatic translation** of field labels and values when internationalization is active

The slot annotations—`setShouldStoreSlot`, `setSyncsToView`, `setCanEditInspection`, `setIsSubnodeField`—are the bridge between the domain model and the framework's automatic behaviors. Each annotation controls one aspect of the object's lifecycle; together, they provide enough information for the UI, storage, and synchronization layers to operate without additional code.

6 Architecture

Strvct is implemented as a client-side JavaScript framework. Applications run as single-page apps in the browser, making heavy use of client-side persistent storage—both for caching code and resources via a content-addressable build system, and for maintaining a persistent object database of application state in IndexedDB.

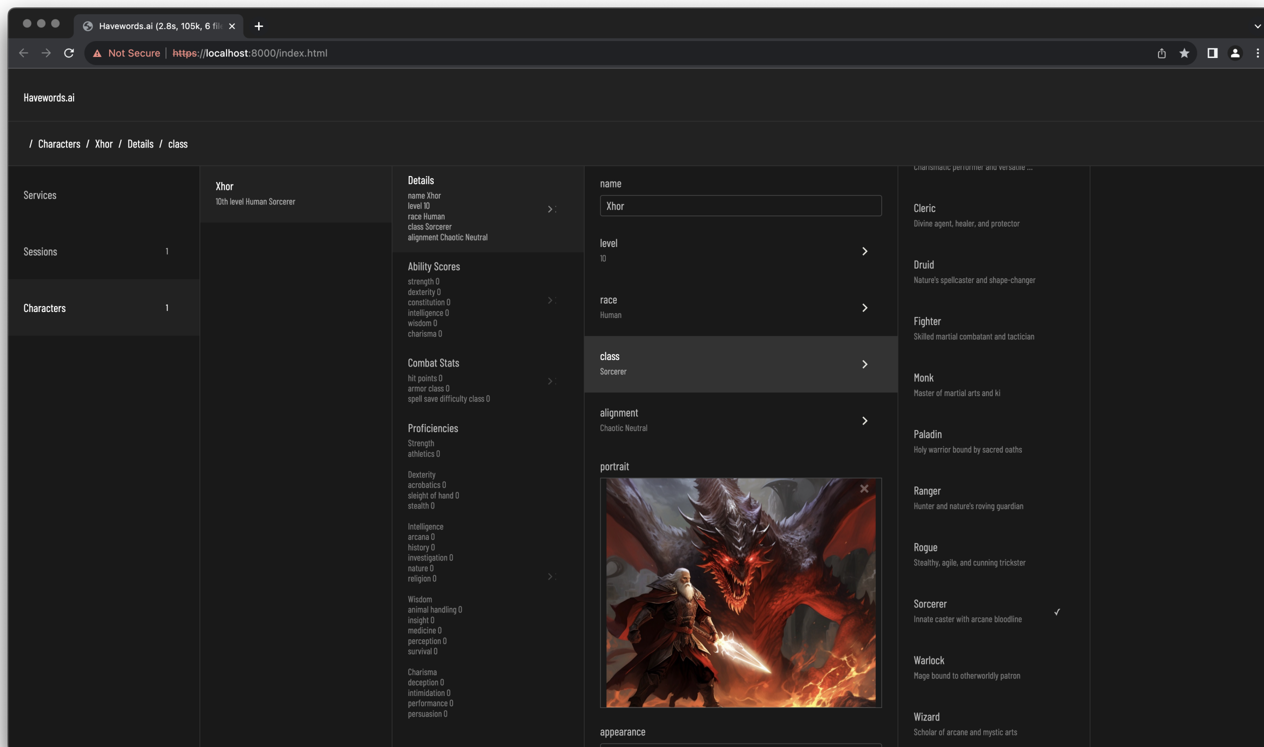

Figure 9. The Strvct framework as used in undreamedof.ai, an AI-powered virtual tabletop for tabletop roleplaying games. The interface—including character sheets, campaign hierarchies, session management, and settings panels—is generated entirely from domain model annotations.

6.1 Domain Model

The domain model is a graph of objects inheriting from a common base class. Each object has properties declared as *slots* with annotations, actions exposed as methods, a `subnodes` array of child objects, a `parentNode` reference, and a unique persistent identifier.

The model is fully independent of the UI layer. Model objects hold no references to views and communicate outward solely by posting notifications. This allows the same model code to run headlessly in Node.js for testing or server-side processing.

6.2 The Annotation Bridge

The slot system is what makes automatic UI and storage possible. Rather than using raw instance variables, properties carry metadata annotations that each framework layer consults independently:

- **Type**—selects the appropriate property tile and enables on-write type checking: the framework validates values when setters are called, catching type errors at the point of mutation rather than at storage or rendering time
- **Persistence**—includes the slot in storage records
- **View synchronization**—triggers view updates when the value changes
- **Subnode relationship**—controls whether the value appears in the object's navigable hierarchy
- **Editability**—determines whether the property can be modified through the UI
- **Auto-initialization**—specifies a class to instantiate if no value was loaded from storage
- **Translation context**—provides semantic context for AI-powered translation

Crucially, not every property in the model is exposed in the UI. Only slots annotated for view synchronization or marked as subnode fields appear in the generated interface; the rest remain internal to the model. This means the domain model can be richer than its presentation—implementation details, caches, and intermediate state stay hidden without requiring a separate view model or DTO layer.

No single annotation knows about the others. The UI layer reads type and editability; the storage layer reads persistence; the synchronization layer reads sync flags. This separation means new layers can be added—internationalization, cloud

sync, schema generation—without modifying existing annotations or the domain model itself.

6.3 Storage

Persistence is annotation-driven. The persistence layer monitors slot mutations, batches dirty objects at the end of each event loop into atomic transactions, and commits them to IndexedDB. On load, stored records are deserialized back into live object instances with relationships re-established.

A separate content-addressable blob store handles large binary data using SHA-256 hashes as keys, providing automatic deduplication. Objects store hash references rather than blob data directly.

Automatic garbage collection walks the stored object graph from the root; unreachable objects are removed.

6.4 Synchronization

Model and view layers communicate through a deferred, deduplicated notification system. When a model property changes, a notification is posted; observing views schedule a sync pass. Multiple changes within a single event loop are coalesced. Bidirectional sync stops automatically when values converge, preventing infinite loops. Observations use weak references, so garbage collection of either party automatically cleans up subscriptions.

7 Emergent Capabilities

When the framework controls the entire pipeline from model annotation to rendered view, capabilities that would normally require per-component effort become emergent properties of the architecture. Each capability described below exists because of the same structural fact: the framework has complete knowledge of the domain model and controls the single point where model data flows to the UI.

7.1 Transparent Internationalization

Because all UI text flows from model slot annotations through a single rendering pipeline, translation can be injected at the model-to-view boundary transparently. No per-component `t()` calls or translation key files are needed. New model classes are translatable by default.

This centralization also makes AI-powered translation practical. The framework discovers every translatable string by walking model class prototypes—an introspection that is only possible because all UI-visible text originates from annotated slots. Slot-level context annotations (e.g., "game mechanic" vs. "ui label") travel with the model, giving the AI translator domain-appropriate terminology without developer effort. Adding support for a new language becomes a configuration change rather than a translation project.

In a conventional framework, achieving automatic translation would require either instrumenting every component with translation calls, or building an extraction tool that parses component templates to find translatable strings. Both approaches scale linearly with the number of components. In Strvct, the cost is zero per component—the framework handles it structurally.

7.2 Transparent Persistence and Cloud Sync

Because the framework owns the complete object graph and understands the structure of every node through annotations, it can split persistence into two strategies transparently. The object pool—the graph of domain objects and their properties—is loaded synchronously, ensuring the model is immediately available for UI rendering. Large binary resources such as media and document files are stored separately in a content-addressable blob store and loaded asynchronously on demand, so they never block the UI.

This same structural knowledge enables transparent cloud synchronization. Subgraphs of the local object database can be lazily synced to cloud storage without the developer writing any sync logic—the framework knows which objects have changed, which blobs they reference, and how to reconcile local and remote state. The developer annotates what should persist; the framework decides how and when to load, store, and sync it.

7.3 JSON Schema Generation

Domain objects can automatically generate JSON Schema descriptions of themselves based on their slot annotations. This is particularly valuable for integration with large language models, which can use the schema to understand the structure of domain objects, validate their output, and generate patches or new instances that conform to the model's constraints. In Strvct's production use, AI assistants modify game state by generating JSON Patch operations validated against auto-generated schemas—a workflow that requires no hand-written schema definitions.

7.4 Integrated Debugging

Because the UI is a direct projection of the model, the same navigable interface serves as its own debugging tool. A developer mode reveals the full object graph—including properties normally hidden from end users—using the same tile and master-detail views as the application itself, and any visible element can be inspected to navigate directly to its backing model object. This is particularly valuable for objects that have no end-user-facing interface at all—networking pipelines, request chains, caches, and other infrastructure. Because the framework generates a navigable interface for *every* object in the graph, developers can explore these internal structures visually, edit their properties through type-appropriate controls, and invoke actions—effectively a custom debugging interface for every object in the system, generated without writing any debugging code. If the interface is structurally identical to the model, then inspecting the model *is* inspecting the interface.

7.5 Content-Addressable Resource Loading

The framework's build system produces a content-addressable bundle where source files are keyed by SHA-256 hash. The client caches resources locally in IndexedDB and uses hash comparison to determine what has changed between deployments. Unchanged resources are never re-downloaded, and identical content across different file paths is stored only once. This is a consequence of the framework controlling the full resource pipeline—analogous to how it controls the model-to-view pipeline—and provides caching granularity that standard bundlers cannot achieve.

8 Case Study: undreamedof.ai

Strvct has been used to build undreamedof.ai, an AI-powered virtual tabletop for Dungeons & Dragons and other tabletop roleplaying games. The application includes:

- **Character system**: hierarchical character sheets with ability scores, inventory, spellcasting, and equipment—approximately 30 domain classes
- **Campaign system**: adventure management with nested locations, NPCs, creatures, treasures, and narrative structure—approximately 20 domain classes
- **Session system**: real-time multiplayer game sessions with AI game master, dice rolling, voice narration, and peer-to-peer networking—approximately 25 domain classes
- **AI integration**: multiple AI service providers, tool calling, streaming responses, and prompt composition—approximately 15 domain classes

The application comprises roughly 90 domain model classes. Of these, fewer than 10 required custom view classes. The remainder—including character sheets, campaign hierarchies, settings panels, and administrative interfaces—use the framework's automatically generated views exclusively.

This ratio—approximately 90% auto-generated views—is the practical test of the composable primitive approach. The domain model is non-trivial: character sheets have deeply nested hierarchies (character → ability scores → individual scores → modifiers), campaigns contain recursive location trees, and the session system manages real-time state across multiple connected clients. Despite this complexity, the default tiles and master-detail views produce navigable, usable interfaces throughout.

The custom views that were needed fall into the category identified in Section 3 as outside the narrow design space: a chat interface for AI conversation, a 3D dice roller, and a map view. These are inherently graphical, domain-specific components that cannot be derived from model annotations. Their existence does not undermine the approach—it confirms that the boundary between auto-generated and bespoke views falls where predicted.

9 Related Work

Naked objects implementations. The pattern was introduced by Pawson and Matthews [7] and formalized in Pawson's doctoral thesis [6]. Apache Causeway [1] (formerly Apache Isis) is the most mature naked objects framework, providing automatic UI generation for Java domain models with both a web UI (Wicket viewer) and a REST API. JMatter [2] implemented naked objects for Java Swing. Both use form-and-table UI strategies and target enterprise or administrative use cases. Constantine [4] offered an early critique of the pattern's usability, arguing that automatically generated object-oriented interfaces lack the task-oriented structure users need. Pawson acknowledged this tension, noting that naked objects are best suited to "sovereign" applications with frequent users, and less appropriate for transient or occasional use [6]. Strvct differs from prior implementations in its UI strategy (composable spatial primitives rather than forms and tables) and its target (end-user applications rather than internal tools).

Model-driven UI generation. The broader field of model-driven development has produced approaches like IFML [3] (Interaction Flow Modeling Language) and UsiXML, which use abstract UI models to generate interfaces. These typically require separate UI models in addition to domain models—a layer of specification that naked objects explicitly eliminates. Strvct's approach is closer to naked objects in that the domain model itself, annotated with metadata, is the only specification needed.

Miller Columns. The column-based navigation pattern was introduced in NeXTSTEP and popularized by macOS Finder [5]. It provides spatial continuity when browsing hierarchical data. Strvct extends this pattern by making it recursive (columns can nest vertically or horizontally), orientation-flexible (each level can choose its own orientation), and self-composing (the layout is determined by model annotations rather than application code).

Low-code and no-code platforms. Modern low-code platforms (Retool, Appsmith, OutSystems) aim to reduce UI development cost through visual builders and pre-built components. They approach the same problem as naked objects—reducing the cost of UI development—but from the opposite direction: rather than eliminating bespoke UI, they make bespoke UI faster to produce. The result is still a collection of individually designed screens that must be maintained as the data model evolves. Naked objects eliminates this maintenance cost entirely.

Declarative UI frameworks. Modern frameworks such as SwiftUI and Jetpack Compose reduce per-screen boilerplate by letting developers declare UI structure in terms of data bindings and composable view primitives. This is a significant improvement over imperative UI code, and the composable view primitives bear a surface resemblance to Strvct's tiles and tile stacks. However, the developer still

writes view code for each screen—the binding between model and view is explicit and per-component. Adding a new model property requires adding a corresponding view declaration. Naked objects eliminates this coupling entirely: the model *is* the specification, and the view follows from it without any per-component view code. Declarative frameworks make bespoke UI easier to write; naked objects makes it unnecessary.

AI-generated UI. Large language models can now generate UI code from natural language descriptions. This automates the *creation* of bespoke interfaces but does not address their *maintenance*—each generated screen is still a separate artifact that must be updated when the model changes. Naked objects is a fundamentally different approach: rather than automating the production of bespoke UIs, it eliminates the need for them.

10 Discussion

10.1 The Crossover Point

Hand-crafted interfaces may appear more polished early in an application's life, when the number of screens is small and each can receive individual design attention. But as the domain model grows, the cost of maintaining bespoke screens grows with it, while inconsistencies accumulate. At some point—the crossover point—a consistent, automatically generated interface produces a better user experience than a patchwork of hand-crafted screens, because the user can rely on uniform navigation throughout the application.

The composable primitive approach shifts this crossover point earlier by improving the quality of the generated interface. The undreamedof.ai case study suggests the crossover may occur sooner than expected: at approximately 90 domain classes, auto-generated views were not merely acceptable but preferred for 90% of the interface, because they provided consistent navigation and interaction patterns that would have been difficult to maintain across hand-crafted screens.

10.2 A Familiar Dynamic

This dynamic—general methods outperforming hand-crafted solutions as scale increases—appears in other domains. In his essay *The Bitter Lesson* [8], Rich Sutton observed that across 70 years of AI research, general methods that leverage computation consistently outperformed approaches that encoded human expertise, and the gap widened with scale.

The analogy to naked objects is instructive if imperfect. In AI, the scaling dimension is computation; in naked objects, it is domain model complexity. The principle is similar: investing in a general method that improves with scale—rather than in case-by-case engineering that must be repeated for each new component—produces compounding returns. But unlike AI scaling, which depends on hardware improvements, naked objects scaling depends on a design choice: whether the framework's primitives are good enough to make hand-crafted specialization unnecessary for most screens.

10.3 Limitations

The composable primitive approach is best suited to informational and navigational interfaces—applications centered on browsing, editing, and managing structured data. Highly graphical interfaces (data visualizations, design canvases, game renderers) require domain-specific rendering that cannot be derived from model annotations. Strvct supports custom view classes for these cases, but they fall outside the automatic generation pipeline.

The spatial conventions we rely on—top-to-bottom hierarchy, left-to-right depth—reflect Western reading order. Right-to-left languages would require mirrored layouts, which the framework's flexbox-based rendering can accommodate but which have not yet been fully validated.

11 Conclusion

The naked objects pattern has offered a compelling proposition for twenty-five years: write the domain model, and the rest follows. Its limited adoption is not a failure of this proposition but of the UI strategies that prior implementations chose. Generic forms and tables were sufficient for internal tools but did not meet the expectations set by modern consumer software.

We have argued that the gap is closable because the design space is narrow. Bespoke UI developers are converging on a small set of spatial conventions; a framework can apply these conventions uniformly through composable primitives—tiles, tile stacks, and recursively nested master-detail views—that produce familiar, navigable interfaces from annotated domain models alone.

Strvct demonstrates this approach in production. A nontrivial application with approximately 90 domain classes uses auto-generated views for approximately 90% of its interface, with custom views needed only for inherently graphical components that fall outside the narrow design space. The centralized model-to-view pipeline enables emergent capabilities—transparent internationalization, annotation-driven persistence with cloud sync, and automatic schema generation—that validate the architectural decision to invest in the general method.

The challenge remains making the general method good enough that hand-crafted specialization becomes the exception rather than the rule. We believe the evidence presented here—a narrow design space, a small set of composable primitives, and a production application that confirms both—suggests this goal is within reach.

References

[1] Apache Software Foundation. [n. d.]. Apache Causeway. https://causeway.apache.org/ Formerly Apache Isis.

[2] Juan Manuel Arteaga. [n. d.]. JMatter: A Naked Objects Framework for Java Swing. http://jmatter.org/

[3] Marco Brambilla and Piero Fraternali. 2014. Interaction Flow Modeling Language. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14 Companion)*. ACM.

[4] Larry L. Constantine. 2002. The Emperor Has No Clothes: Naked Objects Meet the Interface. *Journal of Object Technology* 1, 3 (2002), 47–60. http://www.jot.fm/issues/issue_2002_07/column1/

[5] Wikipedia contributors. 2024. Miller columns. https://en.wikipedia.org/wiki/Miller_columns

[6] Richard Pawson. 2004. *Naked Objects*. Ph. D. Dissertation. Trinity College, Dublin. http://downloads.nakedobjects.net/resources/Pawson%20thesis.pdf

[7] Richard Pawson and Robert Matthews. 2002. *Naked Objects*. John Wiley & Sons.

[8] Rich Sutton. 2019. The Bitter Lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html Incomplete Ideas.