

The Bellew-Norrie Method

Nathan Bellew

10/25/2026

From \$2.5 billion to \$7.6 billion: the estimated cost of the Federal Aviation Administration's Advanced Automation System more than tripled before the program was restructured in 1994, with key components facing delays of eight years or more (U.S. General Accounting Office, 1996). Although these figures encompass a major modernization program rather than an isolated software rewrite, they illustrate the stakes of changing complex operational systems. Small companies face a related design question much earlier: how can they build an initial product quickly while preserving their ability to expand and modify it? Research shows that startups adapt engineering practices to their constraints and face different challenges as their products mature (Paternoster et al., 2014; Wang et al., 2016). The proposed Bellew-Norrie Method addresses this challenge through two goals: designing software that can accommodate new features, integrations, and developers, and establishing a development approach that can mature alongside it. Combining established object-oriented and structured-design practices, the method concentrates design rigor on core components while keeping supporting work proportional to its significance. Clear responsibilities, replaceable components, and explicit rules provide a foundation for extending the product and strengthening development practices as demands increase. A grocery store developing an online ordering system will serve as a recurring example to make these decisions concrete. Six qualities guide the Bellew-Norrie Method: traceability, readability, cohesion, low coupling, explicit contracts, and invariant protection.

Traceability

NASA's software engineering guidance identifies a consequence of weak traceability: "implementing unnecessary features unrelated to mission requirements, adding cost and complexity" (NASA, n.d., "Why This Matters in NASA Context" section). For a small company developing its first product, that warning raises a practical concern: how much development effort is building capabilities without a clearly established purpose? Traceability connects a requirement to its origin and to the design, implementation, and verification that address it (NASA, n.d.). These connections work in both directions: developers can investigate how a requirement is implemented or determine why an existing component is necessary. Maintaining these relationships helps identify missing functionality and assess the effects of changing requirements (NASA, n.d.). The use-case-driven approach associated with Object-Oriented Software Engineering (OOSE) provides a practical starting point for establishing these connections around clearly defined user goals.

OOSE organizes software development around use cases, which describe how an actor interacts with a system to accomplish a goal, including successful and unsuccessful

outcomes (Jacobson & Cockburn, n.d.). For a small grocery store developing an online ordering system, one such goal is allowing customers to remove unwanted items before checkout. The use case establishes the expected behavior: the selected item disappears from the cart, the total is recalculated, and the customer can confirm the change. Linking that description to the responsible components and acceptance tests establishes traceability between the customer's need and its implementation. If promotional discounts are introduced later, those links help the team identify where removing an item could affect discount eligibility. OOSE is therefore the preferred approach to traceability in early product development: its use cases provide a clear reference point for connecting what customers need to what developers build and verify.

Readability

Approximately 58% of developers' time was spent on program comprehension activities in a field study (Xia et al., 2018). These activities included understanding code and consulting supporting materials, demonstrating the substantial effort involved in determining how software works (Xia et al., 2018). Code readability concerns how easily developers can read and understand source code (Oliveira et al., 2021). For a small company developing an early product, readability deserves particular attention in the core behaviors identified through use cases, because those behaviors will need to remain understandable as features and developers are added. A practical priority is domain readability: making the purpose and rules of the business recognizable in those core features. Domain-Driven Design (DDD) is the preferred approach for establishing this connection between what a business does and how its software expresses it.

DDD centers software design on a model of the domain, developed through collaboration between developers and domain experts (Evans, 2015). Its ubiquitous language establishes consistent terminology within a defined context and carries that terminology into discussions, models, and implementation (Evans, 2015). For the grocery store's online ordering system, familiar terms such as shopping cart, order number, and substitution could appear directly in class, variable, and operation names as **shoppingCart**, **order_number**, and **request_substitution**. Their meanings would reflect how customers and store employees describe the ordering process: a substitution, for example, means replacing an unavailable item with an alternative. Operations such as adding an item, placing an order, and requesting a substitution would express those business activities directly. If the store later allows customers to approve substitutions, developers could identify the relevant order behavior through the same terms used to describe the new requirement. This shared vocabulary can help existing developers modify the product and

new developers understand its rules as the team grows. DDD is therefore the preferred approach to readability within the core domain: it gives developers a consistent business meaning to follow when understanding and extending the code.

Cohesion

“The effectiveness of a ‘modularization’ is dependent upon the criteria used in dividing the system into modules” (Parnas, 1972, p. 1053). Dividing a growing application into smaller pieces is therefore insufficient unless those pieces group responsibilities meaningfully. Cohesion describes how closely the processing elements within a module relate to one another and contribute to its purpose (Yourdon & Constantine, 1979). At the architectural level, this means examining whether a component’s responsibilities form a coherent capability rather than a collection of unrelated operations. For small companies developing early products, cohesion deserves particular attention within core domain components, where new capabilities must fit alongside existing behavior. Keeping these components focused provides a basis for extending the product without repeatedly reorganizing its foundation. Yourdon–Constantine structured design (YCSD) offers a practical approach to establishing these boundaries while allowing temporary supporting utilities to remain lightweight.

YCSD is a function-oriented design method that decomposes software into modules and evaluates their organization through cohesion and coupling (Yourdon & Constantine, 1979). In the grocery store’s online ordering system, calculating item subtotals, applying eligible discounts, and determining the final amount could belong within an order-pricing component because these operations contribute to the same result. Assigning a delivery time serves a different purpose and would belong elsewhere, even though both activities occur during checkout. If the store introduces loyalty discounts, the pricing component provides a clear location for that behavior without absorbing delivery responsibilities. This makes YCSD particularly suitable for organizing core domain functionality: it provides a criterion for deciding which operations belong together as the product expands. A temporary script for importing a supplier’s price list can receive lighter design treatment, provided it uses established update operations rather than duplicating pricing rules. The distinction is proportional effort, not permission to undermine the core. YCSD is therefore the preferred approach to cohesion: it guides the placement of new responsibilities so that growth strengthens existing modules instead of turning them into collections of unrelated features.

Low Coupling

“With a tightly coupled architecture, small changes can result in large-scale, cascading failures” (DORA, n.d., para. 6). Tight coupling means high interdependence: a change in one component can affect others that rely on it, spreading the consequences beyond the original modification (DORA, n.d.). Coupling concerns how much one software component depends on another; low coupling limits those dependencies by allowing components to communicate through defined interfaces without needing to know each other’s internal implementation (Amazon Web Services [AWS], n.d.). This allows developers to modify a component or add features while limiting the risk to other components that depend on it (AWS, n.d.). For a small company, the practical benefit is containing the work required for a change: adding a capability should require fewer unrelated components to be modified and retested. Development effort can then remain focused on improving the product instead of repairing consequences elsewhere in the application. Establishing these boundaries between core components early is therefore a priority for supporting growth and later modification. Combining YCSD’s emphasis on limited dependencies with DDD’s defined domain boundaries is the preferred approach to achieving that separation.

YCSD evaluates the connections between modules and favors interactions that exchange necessary information without depending on internal processing details (Yourdon & Constantine, 1979). DDD complements this objective through bounded contexts that establish where a model applies and aggregates that control access to related domain objects through an aggregate root (Evans, 2015). In the grocery store’s online ordering system, the shopping cart could be modeled as an aggregate whose root manages its items and quantities. Checkout would request a summary containing the product identifiers and quantities needed to prepare an order, rather than directly accessing and modifying the cart’s internal collection. This summary would contain independent values, allowing checkout to prepare its own order representation while the cart retains control of its state. If the store later changes how cart items are stored, checkout could continue using the same summary without understanding the new storage arrangement. YCSD supplies the criterion for limiting the dependency, while DDD identifies the ownership boundary that the interaction must respect. Their combination is therefore the preferred approach to low coupling between core components: each component receives the information it needs while preserving the other’s freedom to change its implementation.

Explicit Contracts

“Contract-first means that you design the API contract (the interface) first and then write code that implements the contract” (Microsoft, 2025, “OpenAPI Initiative” section). Establishing what components promise before deciding how they deliver it provides a practical foundation for replaceable implementations. An explicit software contract states the conditions a caller must satisfy and the results an operation guarantees when those conditions are met (Meyer, 1992). Microsoft’s guidance similarly distinguishes preconditions, which apply before an operation, from post-conditions, which describe expectations afterward (Microsoft, n.d.). For a small company, these agreements give developers a defined target when extending or replacing a component. Other components can continue relying on the established behavior, provided the replacement preserves its guarantees. Explicit contracts therefore support low coupling by making the permitted interaction clear without exposing how the work is performed. Design by Contract (DbC) is the preferred method for making these expectations explicit and establishing the conditions implementations must satisfy.

DbC specifies the obligations and guarantees of software operations through preconditions, post-conditions, and invariants (Meyer, 1992). In the grocery store’s online ordering system, checkout already requests a cart summary instead of accessing the cart’s internal collection. Applying DbC would establish that the request must identify an existing cart. In return, the summary operation must supply each selected product’s identifier and quantity without changing the cart’s contents. These conditions could be expressed through executable checks and verified through tests, making the agreement something developers can evaluate rather than merely assume. If the store replaces its cart implementation, the same conditions provide a basis for checking whether checkout can continue using it unchanged. This makes DbC particularly suitable for small companies expecting their implementations to evolve: developers can revise how a component works while preserving the behavior its callers depend on.

Invariance Protection

“Many bugs occur because objects are in a state they should never have been in” (Microsoft, 2022, para. 4). An invariant is a condition that must remain true whenever a domain object is in a valid observable state, including after operations change its data (Microsoft, 2022). Protecting an invariant does not prevent values from changing; it prevents changes from leaving the object in an invalid condition. Microsoft’s guidance

explains that enforcing these rules within domain objects reduces the need for other components to repeatedly check whether those objects are valid (Microsoft, 2022). For a small company, this provides a practical foundation for adding features without requiring every new interaction to reconstruct the same business rules. Establishing these protections requires initial effort, but it creates an identifiable place to maintain rules as the product grows. Combining DDD's ownership of domain consistency with DbC's explicit conditions is the preferred approach to preserving valid state during future modifications.

DDD assigns an aggregate responsibility for maintaining consistency among the domain objects within its boundary, while DbC specifies the conditions that operations must satisfy and preserve (Evans, 2015; Meyer, 1992). In the grocery store's online ordering system, suppose every item remaining in a shopping cart must have a positive whole-number quantity. Customers can change quantities, but entering zero should remove the item rather than leave an invalid entry, and a negative quantity should be rejected without altering the cart. DDD places responsibility for these changes with the cart aggregate; DbC makes the rule explicit as a condition that must hold after each completed operation. If the store later adds a "buy again" feature, it can use the same controlled cart operations instead of implementing its own quantity rules. The new feature still requires testing, but it does not need a separate interpretation of what makes a cart valid. Together, DDD and DbC let new features build on rules enforced at their source, allowing developers to extend the product without rediscovering what makes its data valid.

Conclusion

The Bellew-Norrie Method is intended for small teams and small companies developing new software products. It combines OOSE, DDD, YCSD, and DbC around six qualities: traceability, readability, cohesion, low coupling, explicit contracts, and invariant protection. Use cases preserve the connection between customer needs and implementation; shared business terminology makes core behavior recognizable; cohesive modules and limited dependencies provide clear locations for changes; and contracts and protected invariants establish the rules those changes must respect. Small teams can concentrate this effort on core components while keeping temporary supporting utilities lightweight. In the grocery store's online ordering system, this gives discounts, substitutions, and repeat purchases identifiable places to belong, established interactions to use, and existing rules to preserve.

For a small company introducing a new product, the practical priority is building something useful today while preserving the ability to improve it tomorrow. The Bellew-Norrie Method

addresses that priority by connecting an adaptable software foundation with a development approach that can mature alongside it. Teams can begin with focused use cases, understandable components, and explicit rules, then strengthen documentation, review, and verification as their responsibilities increase. This makes the method a practical choice for early product development, when limited time and staffing demand deliberate decisions about where design effort matters most. Small teams should establish these foundations while their products are still small, so that new features, integrations, and developers can build on the work already completed. A small company's first product should provide a foundation for its next stage of growth.

References

Amazon Web Services. (n.d.). *REL04-BP02 Implement loosely coupled dependencies*. AWS Well-Architected Framework.

https://docs.aws.amazon.com/wellarchitected/latest/framework/rel_prevent_interaction_failure_loosely_coupled_system.html

DORA. (n.d.). *Capabilities: Loosely coupled teams*. <https://dora.dev/capabilities/loosely-coupled-teams/>

Evans, E. (2015). *Domain-driven design reference: Definitions and pattern summaries*. Domain Language. https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf

Jacobson, I., & Cockburn, A. (n.d.). *Use-case foundation*. Ivar Jacobson International. <https://www.ivarjacobson.com/publications/use-case-foundation>

Meyer, B. (1992). Applying “design by contract.” *Computer*, 25(10), 40–51. <https://doi.org/10.1109/2.161279>

Microsoft. (n.d.). *Code contracts*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts>

Microsoft. (2022, September 21). *Design validations in the domain model layer*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-model-layer-validations>

Microsoft. (2025, May 8). *Web API design best practices*. Microsoft Learn.

<https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>

National Aeronautics and Space Administration. (n.d.). *SWE-052—Bidirectional traceability*. Software Engineering Handbook.

<https://swehb.nasa.gov/display/SWEHBVD/SWE-052+-+Bidirectional+Traceability>

Oliveira, D., Bruno, R., Madeiral, F., & Castor, F. (2021). *Evaluating code readability and legibility: An examination of human-centric studies*. arXiv.

<https://doi.org/10.48550/arXiv.2110.00785>

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules.

Communications of the ACM, 15(12), 1053–1058. <https://doi.org/10.1145/361598.361623>

Paternoster, N., Giardino, C., Unterkalmsteiner, M., Gorschek, T., & Abrahamsson, P. (2014). Software development in startup companies: A systematic mapping study.

Information and Software Technology, 56(10), 1200–1218.

<https://doi.org/10.1016/j.infsof.2014.04.014>

U.S. General Accounting Office. (1996). *Aviation acquisition: A comprehensive strategy is needed for cultural change at FAA* (GAO/RCED-96-159).

<https://www.gao.gov/assets/rced-96-159.pdf>

Wang, X., Edison, H., Bajwa, S. S., Giardino, C., & Abrahamsson, P. (2016). Key challenges in software startups across life cycle stages. In H. Sharp & T. Hall (Eds.), *Agile processes, in software engineering, and extreme programming* (pp. 169–182). Springer.

https://doi.org/10.1007/978-3-319-33515-5_14

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., & Li, S. (2018). Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10), 951–976.

<https://doi.org/10.1109/TSE.2017.2734091>

Yourdon, E., & Constantine, L. L. (1979). *Structured design: Fundamentals of a discipline of computer program and systems design*. Prentice-Hall.