**Evaluating Polymorphism in Compiler Design**

Samuel Caldwell

Champlain College

CSCI-430: Compilers

Dr. Kenneth Revett

11 November 2025

# Evaluating Polymorphism in Compiler Design

Polymorphism in C++ offers a principled way to trade centralized branching for type-directed behavior that mirrors the compiler's domain model. At a measurement level, replacing large `switch`/`if` cascades with virtual or statically bound overrides tends to lower cyclomatic complexity at call sites, shrinking the number of explicit decision edges a reader must track (McCabe, 1976; Fowler, 2018). At a cognitive level, it improves chunking by letting developers reason in terms of named abstractions—`Expr`, `Stmt`, `Instruction`, `Pass`—whose responsibilities are localized and whose invariants can be read where they are enforced, thereby reducing working-memory load (Sweller, 1988; Martin, 2008). This alignment supports the Open–Closed Principle: new language features or target idiosyncrasies are added by introducing new classes that satisfy existing interfaces rather than editing a central dispatcher, which curbs modification ripple and confines review to semantically relevant modules (Gamma, Helm, Johnson, & Vlissides, 1994; Martin, 2008).

However, polymorphism redistributes, rather than eliminates, complexity. Local simplicity can increase global dispersion: behavior becomes scattered across many overrides, and the "at-a-glance" exhaustiveness of a dense conditional or pattern match is lost, complicating audits and cross-cutting policy enforcement such as diagnostic consistency or feature gating (Martin, 2008; Fowler, 2018). The performance envelope also shifts. While modern compilers devirtualize many calls, dynamic dispatch can still inhibit inlining on hot paths, whereas table-driven or pattern-matching approaches over closed sets often yield tighter code and clearer global invariants, particularly in parsing, instruction selection, and fast IR rewrites (Aho, Lam, Sethi, & Ullman, 2006; Stroustrup, 2013). Consequently, effective designs balance forms of polymorphism: use dynamic dispatch where the variation set is open and semantic clarity dominates; prefer static polymorphism (e.g., CRTP) or declarative tables where throughput and exhaustiveness are paramount; and centralize cross-cutting policies so that localized overrides do not drift (Sutter & Alexandrescu, 2004; Lattner & Adve, 2004).

A qualified case can be made that test-driven development (TDD) partially mitigates the readability trade-offs introduced by polymorphism, but it does not eliminate them. On the plus side, writing tests first for each override or strategy creates a "living specification" that localizes intent next to behavior; fine-grained, example-driven tests document contracts and edge cases for dispersed polymorphic classes, improving discoverability and enabling safe refactoring (Beck, 2002; Fowler, 2018; Martin, 2008). Test names and fixtures can serve as navigational affordances—developers can jump from a failing test to the precise subtype under scrutiny—and characterization tests help maintain consistent diagnostics and policies across subclasses (Feathers, 2004; Meszaros, 2007). However, TDD cannot substitute for architectural clarity: high volumes of tests may still mirror underlying indirection, and heavy use of doubles or mocks can obscure real integration semantics, increasing cognitive load when tracing end-to-end behavior (Meszaros, 2007; Fowler, 2018). Moreover, strong test coverage does not guarantee "at-a-glance" exhaustiveness that centralized conditionals or declarative tables provide. In practice, TDD complements polymorphism by tightening feedback loops and documentation, while readability at scale still depends on disciplined interfaces, limited hierarchy depth, and centralized enforcement of cross-cutting policies (Fowler, 2018; Martin, 2008).

## Parsing

**Benefits.** Replacing monolithic `switch`/`if` dispatch with polymorphic nonterminal handlers (e.g., `Nonterminal::parse()` with concrete `IfStmt`, `CallExpr`, `Block`) lowers cyclomatic complexity at the call site and aligns code structure with the grammar's nonterminals (McCabe, 1976; Aho et al., 2006). This alignment aids traceability: each syntactic construct has a single class where recognition, local error messages, and minimal recovery live, which reduces working-memory demands because readers reason about one cohesive unit rather than a control-flow tangle (Sweller, 1988; Fowler, 2018). In hand-written recursive-descent and Pratt parsers, polymorphic precedence/affix handlers (prefix/infix/postfix) map cleanly to operator-binding rules, yielding interfaces such as

`nud()`/`led()` or `parse_infix()` that encode grammar intent and simplify incremental extension (Aho et al., 2006; Fowler, 2018). Polymorphism also helps encapsulate context-sensitive quirks—indentation blocks, semicolon insertion, or language modes—behind construct-specific contracts, reducing incidental coupling in the driver.

**Costs.** The dispersion of recognition logic across many classes weakens at-a-glance exhaustiveness: a centralized `switch` over token kinds or a table-driven recognizer immediately shows complete coverage and default cases, which is valuable for audits and for enforcing uniform diagnostics. In hot front ends, virtual dispatch on every token boundary can inhibit inlining and degrade branch prediction; dense `switch`es over enums often compile to jump tables with excellent locality (Aho et al., 2006). Furthermore, sophisticated error recovery—panic-mode sync sets, phrase-level repair—benefits from global visibility into follow sets and FIRST/FOLLOW relations; scattering recovery across overrides risks inconsistent messaging and resynchronization unless a shared policy layer is retained (Aho et al., 2006; Martin, 2008).

**LL/LR and declarative engines.** For LL-style, hand-written parsers, polymorphism fits the top-down, by-nonterminal mental model and can mirror the grammar closely. For LR/GLR or PEG-based approaches, the natural representation is data-driven: parse tables or compiled pattern matchers centralize decisions and deliver predictable performance and error states; replacing these with deep subtype hierarchies often fights the algorithm's grain (Aho et al., 2006; Cooper & Torczon, 2011; Ford, 2004). Even in Pratt/recursive-descent designs, declarative operator tables (precedence/associativity) frequently outperform a forest of virtual overrides for operators, while remaining readable and exhaustively checkable (Cooper & Torczon, 2011).

**Modern C++ alternatives.** C++ variants offer a middle path: represent tokens or AST node sums with `std::variant` and use `std::visit` to obtain localized handlers with compile-time exhaustiveness checks, preserving a single site of dispatch per concern without virtual costs. This

pattern keeps recognition decisions explicit and auditable while still distributing semantic actions into small, testable callables (Stroustrup, 2013; Fowler, 2018).

**Heuristics.** Use dynamic polymorphism where the set of constructs is open and evolving, where per-construct diagnostics matter, and where readability trumps raw throughput. Prefer table-driven/pattern-based parsing for closed grammars, performance-critical lexeme-to-structure recognition, and uniform recovery. Centralize cross-cutting policies—error taxonomy, feature gating, recovery strategy—and let polymorphic nonterminals focus on local recognition and messages (Aho et al., 2006; Martin, 2008; Fowler, 2018; Cooper & Torczon, 2011).

## Semantic Analysis

**Benefits.** Polymorphism localizes language rules and invariants at the point where they conceptually belong. AST classes exposing virtual operations such as `Expr::type_check()`, `Decl::bind()`, `Stmt::control_effects()`, or `Node::attributes()` let each construct encode its own constraints—name-lookup context, mutability rules, purity or effect annotations, temporal lifetime checks—reducing the need to cross-reference a monolithic dispatcher (Aho et al., 2006; Fowler, 2018; Martin, 2008). Overload resolution, implicit conversions, and generics/instantiation can be factored into polymorphic strategies (e.g., `Conversion::rank()`, `OverloadSet::select()`), turning intricate multi-branch control into replaceable components and enabling per-language-mode variants (Muchnick, 1997; Cooper & Torczon, 2011). For compilers that carry attributes (types, effects, nullability, ownership) on nodes, polymorphic getters and checkers align with synthesized/inherited attribute flow, making the mapping from static semantics to code explicit and testable at class granularity (Aho et al., 2006). This organization also aids incremental evolution—adding a new construct or attribute typically requires adding one class plus focused tests rather than touching a broad central `switch`—lowering modification risk and supporting parallel development (Fowler, 2018; Martin, 2008).

**Costs.** Dispersed overrides can obscure global invariants. Consistency requirements—diagnostic wording and codes, deprecation gates, capability checks, feature flags, or phase-ordering constraints—are easy to diverge when replicated across many `type_check()` implementations (Martin, 2008). Complex cross-cutting rules such as variance checking, linear/affine ownership, definite assignment, or flow-sensitive type refinement often need a program-wide view; encoding them solely as per-node methods risks duplicated logic and blind spots at control-flow joins (Muchnick, 1997; Cooper & Torczon, 2011). Debugging becomes nontrivial when the active rule depends on dynamic subtype plus contextual state (e.g., scope, generics environment, conditional compilation), increasing navigation effort and stack depth—especially problematic in security reviews where auditors prefer a single auditable locus for policy (Fowler, 2018). Performance-wise, virtual dispatch in tight semantic passes (e.g., flow typing, data-flow-based nullness) can inhibit inlining; an enum-plus-`switch` or data-driven rule table sometimes compiles to more predictable code with clearer exhaustiveness (Aho et al., 2006; Stroustrup, 2013).

**Patterns and alternatives.** Combine dynamic and static polymorphism. Use runtime polymorphism for open sets—user-definable declarations, extensible attributes, plugin checks—and static polymorphism (CRTP, templates) for hot, closed-world analyses such as lattice transfer functions, constraint propagation, and dominator-aware checks, which benefit from inlining and compile-time specialization (Sutter & Alexandrescu, 2004; Stroustrup, 2013). The Visitor pattern remains effective for traversals that must compose multiple semantic concerns (scope building, type checking, effect checking) while keeping cross-cutting policies centralized in the visitor, not scattered across nodes (Gamma et al., 1994; Fowler, 2018). Data-driven formulations—attribute-grammar engines, constraint systems, or tables of typing rules compiled into evaluators—offer a declarative single source of truth with predictable evaluation order; this improves auditability and often performance, while node methods act as thin adapters (Aho et al., 2006; Muchnick, 1997). For languages with algebraic data

types, representing AST variants via `std::variant` and using `std::visit` yields localized handlers with compile-time exhaustiveness checks, blending clarity with performance and avoiding virtual dispatch (Stroustrup, 2013).

**Heuristics.** Favor per-node polymorphism for rules that are (a) local to the construct, (b) unlikely to require whole-program reasoning, and (c) expected to evolve—e.g., construct-specific error recovery, context-free attribute defaults, or surface-syntax desugaring. Prefer centralized visitors or data-flow frameworks for rules that are (a) global or order-sensitive (e.g., definite assignment, liveness-aware capture), (b) require uniform diagnostics/policies, or (c) benefit from exhaustiveness guarantees (Aho et al., 2006; Muchnick, 1997; Cooper & Torczon, 2011). Enforce consistency with shared utility layers (diagnostic emitters, capability/feature guards) invoked by both polymorphic nodes and centralized passes, and backstop dispersion with characterization tests that assert identical outcomes across language modes and node families (Fowler, 2018; Martin, 2008).

## Optimization and IR Construction

**Benefits.** Polymorphism helps decompose transformation logic along stable IR concepts—values, instructions, regions, and passes—so each unit exposes well-scoped behaviors like `Instruction::can_fold()`, `Instruction::rewrite_uses()`, or `Pass::run(Function&)`. This separation improves locality and testability: foldability, aliasing characteristics, side-effect summaries, and dominance preconditions live with the constructs they govern, enabling focused proofs and micro-tests (Lattner & Adve, 2004; Fowler, 2018). In SSA-based IRs, polymorphic `Value`/`Instruction` hierarchies make def–use reasoning explicit, allowing analyses (e.g., constant propagation, GVN, LICM) to query uniform capabilities—“isPure,” “isIdempotent,” “hasNoNaNs”—without sprawling `switch` ladders (Muchnick, 1997; Cooper & Torczon, 2011). Pass-factory and analysis-provider interfaces further reduce cognitive load: a small, stable set of abstract base classes (“analysis result,” “preserved set,” “pass adaptor”) clarifies contracts between analyses and transforms, decoupling worklists and

invalidation policies from transform specifics (Lattner & Adve, 2004). When combined with strategy/state objects (e.g., target-specific cost models, alias-analysis strategies), polymorphism isolates variability behind names that communicate intent, improving readability in large optimization pipelines (Gamma et al., 1994; Martin, 2008).

**Costs.** Excessive runtime polymorphism in hot inner loops—scanning use lists, walking dominator trees, iterating worklists—can inhibit inlining and introduce unpredictable branch behavior, harming steady-state throughput (Stroustrup, 2013). Dispersing rewrite rules into many overrides obscures global invariants such as preservation of SSA dominance, memory-SSA coherence, or canonicalization order; mis-ordered local rewrites can oscillate or fight each other without a centralized normalization policy (Cooper & Torczon, 2011; Muchnick, 1997). Readability also suffers when key decisions (e.g., "what counts as side-effect free?") are scattered across instruction subclasses rather than captured in one auditable table or traits registry; auditors then chase dynamic dispatch across the hierarchy (Martin, 2008; Fowler, 2018). Finally, open-ended subclassing can complicate analysis-invalidation reasoning: if each transform decides locally which analyses it preserves, subtle bugs arise unless a common preservation protocol is enforced at the pass-manager boundary (Lattner & Adve, 2004).

**Patterns and alternatives.** Use dynamic polymorphism to define stable extension points—abstract `Pass`, `Analysis`, and `Instruction` contracts—so new IR constructs or optimizations plug in without editing central `switch`es (Gamma et al., 1994; Lattner & Adve, 2004). Implement hot-path mechanics (worklists, lattice operations, transfer functions) with static polymorphism (CRTP, templates) to enable inlining and specialization while keeping intent clear (Sutter & Alexandrescu, 2004; Stroustrup, 2013). Centralize canonicalization through declarative, data-driven engines (rewrite tables, pattern DAGs) that map recognizable IR shapes to normalized forms; this regains at-a-glance exhaustiveness and predictable pass behavior while letting instruction classes expose small trait queries consumed by the

matcher (Aho et al., 2006; Cooper & Torczon, 2011). For analysis frameworks, prefer traits registries (e.g., tables declaring purity, memory effects, and `nsw`/`nuw` semantics) over per-subclass overrides for facts that are global invariants; per-node polymorphism should compute or refine those traits, not redefine policy (Martin, 2008; Fowler, 2018). Finally, enforce consistency via a pass manager that mediates analysis lifetimes and invalidation rules through a single protocol, keeping cross-cutting correctness visible despite localized transforms (Lattner & Adve, 2004).

**Heuristics.** Reserve runtime polymorphism for open extension surfaces (adding instructions, passes, or target cost models). Prefer static polymorphism or declarative tables for closed, performance-critical transformations (constant folding, peepholes, canonicalization). Keep global normalization, analysis invalidation, and diagnostic policy centralized to avoid drift across overrides. Write characterization tests per instruction/transform and integration tests that assert global invariants (dominance, memory-SSA coherence), so local polymorphic decisions remain readable within a disciplined, auditable whole (Fowler, 2018; Martin, 2008; Cooper & Torczon, 2011; Muchnick, 1997; Lattner & Adve, 2004).

## IR Generation

**Benefits.** Polymorphism clarifies the mapping from surface syntax to intermediate representation by letting each AST construct own its lowering rules (e.g., `Expr::lower(IRBuilder&)`, `Stmt::lower(CFG&)`). This improves locality: evaluation order, short-circuit points, exception/cleanup edges, and temporary lifetimes are expressed next to the construct that dictates them rather than diffused through a large dispatcher (Lattner & Adve, 2004; Muchnick, 1997). For expressions, polymorphic overrides can encapsulate side-effect staging (e.g., sequencing for function calls vs. arithmetic), value categories (r- vs. l-value), and overflow/undefined-behavior flags that must be reflected in IR (`nuw`/`nsw`, trapping ops), yielding predictable SSA formation and def–use chains (Lattner & Adve, 2004). For statements, dedicated lowerers can build canonical control-flow shapes—structured

blocks, early exits, landing pads—producing CFGs that are easier for downstream optimizations to reason about (Cooper & Torczon, 2011). Target-aware lowering hooks (addressing modes, calling conventions, varargs) can be isolated in polymorphic strategy objects, keeping source-level constructs clean while still emitting ABI-correct IR (Muchnick, 1997; Aho et al., 2006). This structure improves testability: each override can be exercised with small surface programs and checked against expected IR patterns, reducing cognitive load during review (Fowler, 2018; Martin, 2008).

**Costs.** Excessive distribution of lowering across many subclasses can obscure global invariants that must hold uniformly, such as dominance of definitions, exception-edge completeness, debug-info attachment, or naming/canonicalization rules for temporaries. When these policies live only "by convention" inside overrides, inconsistency and code drift are likely (Martin, 2008). Performance can suffer if every micro-construct crosses a virtual boundary, preventing inlining of common builder patterns or constant-folding opportunities that a centralized routine might expose to the optimizer (Stroustrup, 2013). Readability also suffers when reconstructing the full lowering of a compound construct demands jumping through numerous overrides; a single, auditable table stating "construct X lowers to shape Y with edges Z" can communicate intent more directly (Cooper & Torczon, 2011; Muchnick, 1997). Finally, advanced features—short-circuit boolean lowering, structured concurrency, destructors/RAII cleanups—interact across constructs; scattering their orchestration into per-node methods risks subtle control-flow bugs without a centralized cleanup/region manager (Aho et al., 2006).

**Patterns and alternatives.** Use runtime polymorphism to expose stable lowering extension points on AST nodes and on backend strategies (calling convention, exception model), but implement hot builders (basic-block creation, phi insertion, temporary materialization) with static polymorphism or inlineable helpers to preserve performance (Sutter & Alexandrescu, 2004; Stroustrup, 2013). Centralize cross-cutting invariants in a thin framework layer—an `IRLoweringContext` that enforces dominance, attaches debug locations, manages cleanup stacks, and funnels all edge creation through a single API—

so per-node overrides cannot bypass policy (Martin, 2008; Fowler, 2018). For regular patterns, prefer declarative or table-driven lowerings (e.g., opcode/operand-shape to IR templates) that per-node methods invoke; this regains at-a-glance exhaustiveness and enables mechanical auditing (Cooper & Torczon, 2011; Aho et al., 2006). Where the language has sum-type ASTs, a `std::variant` plus `std::visit` for localized lowering decisions can provide compile-time exhaustiveness without virtual dispatch, while still delegating complex sub-cases to named helpers (Stroustrup, 2013).

**Heuristics.** (1) Let polymorphic node methods handle local sequencing, temporary formation, and shape selection; (2) keep global guarantees (dominance, EH edges, debug info, canonical naming) in a centralized context; (3) use declarative templates or tables whenever many constructs share a lowering schema; and (4) reserve virtuals for genuinely open sets (user-defined constructs, plugin dialects), relying on static polymorphism or data-driven templates for closed, performance-critical cases (Lattner & Adve, 2004; Cooper & Torczon, 2011; Muchnick, 1997; Fowler, 2018).

## Code Generation (Target Lowering, Selection, Emission)

**Benefits.** Polymorphism maps naturally onto backend boundaries—sub-targets, calling conventions, register files, instruction formats—so each concern is encapsulated behind an interface whose overrides live in target-specific subclasses. Abstract services such as `TargetInfo`, `FrameLowering`, `RegisterInfo`, and `CallingConvention` concentrate ABI rules, prologue/epilogue construction, spill semantics, and callee/caller-save policies where they belong, reducing the cognitive load of global “if target == X” conditionals throughout the pipeline (Lattner & Adve, 2004; Muchnick, 1997). In instruction selection, polymorphic hooks like `legalize()`, `select()`, and `combine()` let targets express addressing modes, immediates, and predicate rules close to the machine model while keeping the generic selector simple. This improves testability (per-instruction-family unit tests) and traceability (a reviewer can open the target subclass to see the precise lowering of, for example, shifts with carry or predicated moves) (Cooper & Torczon, 2011; Muchnick, 1997).

Schedulers and register allocators benefit from pluggable, polymorphic cost models that encapsulate pipeline latencies, hazard recognizers, and allocation preferences; these isolations enable cross-target reuse of the same infrastructure without readability-eroding conditionals (Lattner & Adve, 2004; Aho et al., 2006). Finally, polymorphic emitters clarify the final mile—from MC-layer instruction encodings to fixups and relaxations—so relocation and bundling rules are not scattered across ad hoc code paths (Muchnick, 1997).

**Costs.** Dynamic dispatch on every candidate match or combine in the inner loops of selection and scheduling can inhibit inlining and increase branch mispredictions, especially when operating on large DAGs/CDFGs; dense tables or generated decision trees often outperform virtual chains while offering at-a-glance coverage (Cooper & Torczon, 2011; Stroustrup, 2013). Distributing legality and selection rules across many overrides risks policy drift: two closely related opcodes may diverge subtly in flags, clobbers, or trap behavior if not governed by a centralized traits registry. The same applies to calling conventions and frame layouts—when each override sprinkles small special cases, reviewers lose a single auditable locus for ABI invariants (Martin, 2008; Fowler, 2018). Debugging can also become harder: which `select()` fired depends on dynamic types and sub-target features, stretching stack traces across many classes; table-driven matchers and explicit patterns make the decision path more transparent for diagnostics (Aho et al., 2006; Cooper & Torczon, 2011). Finally, open-ended subclassing complicates global reasoning about codegen correctness properties—e.g., guaranteeing that all expansions preserve liveness and maintain def–use ordering—unless a centralized verification pass asserts these invariants (Muchnick, 1997).

**Patterns and alternatives.** High-performance backends often pair dynamic polymorphism at coarse extension points with declarative or static mechanisms in hot paths. Declarative tree- or DAG-rewrite systems (BURS/BURG-style or DSL-generated matchers) compile patterns into compact decision structures and jump tables, regaining performance and exhaustiveness while keeping target-specific

knowledge readable as data rather than scattered overrides (Fraser & Hanson, 1995; Proebsting, 1992; Cooper & Torczon, 2011). Targets can still contribute polymorphic traits—e.g., "addressing mode accepts scaled index," "immediate fits N bits"—that the table-driven selector consumes, preserving clarity without per-node virtual calls (Muchnick, 1997). For scheduling and register allocation, use polymorphic strategy objects to select among algorithms or feed cost models, but implement the inner kernels (ready queues, priority updates, interference checks) with static polymorphism (CRTP/templates) to enable inlining and specialization (Sutter & Alexandrescu, 2004; Stroustrup, 2013). In emission, keep a polymorphic façade (`Emitter::encode(const Instr&)`) while driving the encoding via generated tables (opcodes, operand kinds, fixup recipes) so reviewers can audit encodings centrally and tools can validate completeness (Aho et al., 2006; Muchnick, 1997). Across all stages, enforce global invariants—clobber sets, implicit defs/uses, flag preservation, barrier semantics—through a shared verification layer that runs after selection/scheduling/emission; this counterbalances the dispersion introduced by polymorphism (Cooper & Torczon, 2011; Lattner & Adve, 2004).

**Heuristics.** (1) Use dynamic polymorphism to plug in target families, subtargets, and ABI variants; use declarative tables or generated matchers to decide specific instruction forms and combines (Fraser & Hanson, 1995; Proebsting, 1992; Cooper & Torczon, 2011). (2) Keep hot loops table- or template-driven; reserve virtual calls for coarse-grained choices (legalization strategy, scheduler policy, RA mode) (Stroustrup, 2013; Sutter & Alexandrescu, 2004). (3) Centralize non-negotiable policies—calling conventions, prologue/epilogue structure, implicit clobbers, side-effect flags—in auditable registries verified by backend-specific checkers (Martin, 2008; Fowler, 2018). (4) Prefer polymorphic traits and cost models over polymorphic actions in tight paths: compute facts once, then let generated decision code consume them (Muchnick, 1997; Lattner & Adve, 2004). In short, polymorphism provides clean seams for target diversity and architectural clarity, while declarative tables and static polymorphism safeguard throughput and auditability in selection, scheduling, and emission (Aho et al.,

2006; Cooper & Torczon, 2011; Lattner & Adve, 2004; Fraser & Hanson, 1995; Proebsting, 1992; Stroustrup, 2013).

## Synthesis: Conclusions and Cross-Cutting Design Guidelines

Polymorphism in C++ is most effective in compiler design when it is treated as a mechanism for aligning code structure with stable domain concepts while preserving auditability and throughput. The central conclusion is not that polymorphism reduces complexity outright, but that it redistributes complexity into named, testable seams—AST nodes, IR instructions, passes, target traits—thereby lowering local decision load and improving evolvability (McCabe, 1976; Fowler, 2018; Gamma et al., 1994). This redistribution pays off when the variation set is open (new language constructs, subtargets, or passes) and when clarity and modular tests outweigh raw dispatch speed (Martin, 2008; Lattner & Adve, 2004). Because dispersion can obscure global invariants and hinder at-a-glance exhaustiveness, successful designs counterbalance polymorphism with central, declarative sources of truth and performance-friendly dispatch in hot paths (Aho et al., 2006; Cooper & Torczon, 2011).

From these observations follow four defensible guidelines. First, separate open from closed variation sets. Employ dynamic polymorphism where extension is essential (e.g., per-node `type_check()`, per-target calling conventions), and prefer static polymorphism or tables where the case space is fixed and exhaustiveness matters (e.g., canonicalization schemas, peephole rewrites) (Stroustrup, 2013; Sutter & Alexandrescu, 2004; Aho et al., 2006). Second, centralize cross-cutting policies—diagnostic taxonomy, feature gates, dominance and exception-edge guarantees, ABI invariants—in narrow framework layers or declarative registries consumed by polymorphic components (Martin, 2008; Fowler, 2018). Empirically, infrastructures like LLVM demonstrate this pattern: rich class hierarchies at the edges, coupled with central pass management, traits registries, and verifiers that assert global invariants (Lattner & Adve, 2004). Third, optimize hot paths with declarative or compile-time mechanisms. Inner loops of parsing, optimization, and selection benefit from jump-table

`switch`es, generated decision trees, or template-based kernels that the compiler can inline and specialize (Aho et al., 2006; Cooper & Torczon, 2011; Stroustrup, 2013). Fourth, use tests as specification, not as a substitute for design. TDD can document dispersed behaviors and stabilize refactoring, but it cannot restore the at-a-glance exhaustiveness lost to over-dispersion; therefore, tests must complement—rather than replace—centralized policies, verifiers, and declarative summaries (Beck, 2002; Meszaros, 2007; Fowler, 2018). Characterization tests for each override should be paired with integration checks for global properties such as dominance, memory-SSA coherence, and ABI correctness (Muchnick, 1997; Cooper & Torczon, 2011). In sum, a hybrid architecture—dynamic polymorphism at semantic seams, static polymorphism and declarative tables in closed, performance-critical cores, and centralized policy/verification—maximizes local readability and extensibility without sacrificing auditability and speed (McCabe, 1976; Aho et al., 2006; Fowler, 2018; Martin, 2008; Lattner & Adve, 2004; Cooper & Torczon, 2011; Stroustrup, 2013).

# References


Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, techniques, and tools* (2nd ed.). Pearson.

Beck, K. (2002). *Test-driven development: By example*. Addison-Wesley.

Cooper, K., & Torczon, L. (2011). *Engineering a compiler* (2nd ed.). Morgan Kaufmann.

Feathers, M. (2004). *Working effectively with legacy code*. Prentice Hall.

Ford, B. (2004). Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)* (pp. 111–122). ACM.

Fowler, M. (2018). *Refactoring: Improving the design of existing code* (2nd ed.). Addison-Wesley.

Fraser, C. W., & Hanson, D. R. (1995). *A retargetable C compiler: Design and implementation*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. *Proceedings of the International Symposium on Code Generation and Optimization*, 75–88.

Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308–320.

Meszaros, G. (2007). *xUnit test patterns: Refactoring test code*. Addison-Wesley.

Muchnick, S. S. (1997). *Advanced compiler design and implementation*. Morgan Kaufmann.

Proebsting, T. A. (1992). BURS automata and dynamic programming. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM.

Stroustrup, B. (2013). *The C++ programming language* (4th ed.). Addison-Wesley.

Sutter, H., & Alexandrescu, A. (2004). *C++ coding standards: 101 rules, guidelines, and best practices*. Addison-Wesley.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12*(2), 257–285.