

ISP Is a Conditional Corollary of DIP Applied Per Client

An Asymmetric Relationship Gated by Interface Evolution Origin

Ing. Yannick Loth, M.Sc. (Management)

Software Architect & Independent Researcher · Messancy, Belgium
yl@infolead.eu · ORCID: 0009-0003-5754-827X

2026-05-23

DOI: 10.5281/zenodo.20350293

Abstract. The Interface Segregation Principle (ISP) and the Dependency Inversion Principle (DIP) are often presented as independent in textbooks and [1], but applying DIP’s ownership clause once per client–provider edge yields the same interfaces that ISP prescribes when DIP is interpreted causally coherently.

This paper considers only class-level structural relationships. Under this hypothesis, applying DIP’s ownership clause per client always produces ISP-compliant segregation: DIP applied per client implies ISP universally.

The converse—ISP implies DIP applied per client—is **conditional** and holds only under client-driven interface evolution. When interfaces are provider-driven (third-party APIs) or governed by shared negotiation (B2B contracts), ISP can be satisfied without satisfying DIP’s ownership clause.

Therefore, the relationship is asymmetric: DIP applied per client implies ISP universally at the class level, but ISP implies DIP applied per client only under client-driven evolution. Under causal interpretation of DIP’s ownership clause, ISP makes explicit the granularity constraint that is already logically implicit in DIP: an interface owned by a client cannot contain functionality beyond what that client requires, otherwise changes would lack causal grounding in client requirements.

The proof requires no formal machinery beyond the definitions of DIP and ISP restated in this paper.

Several prior authors have argued that SOLID’s five heuristics are not all independent, but, to the author’s knowledge, the specific derivation—that DIP’s ownership clause applied per client is ISP—has not previously been identified in the literature. This paper examines why the connection went unnoticed for nearly thirty years; the most fundamental factor is that both heuristics were stated informally, making logical derivation impossible until they are stated precisely.

Contents

1. Introduction	3
2. Structural Formulations of DIP and ISP	5
2.1 Dependency Inversion Principle	5
2.2 Interface Segregation Principle	7
2.3 Scope of the Two Heuristics	7
2.4 Martin's Hierarchy of Heuristics	7
3. ISP Is a Conditional Corollary of DIP Applied Per Client	9
3.1 Interface Evolution Origins	9
3.1.1 Client-Driven Evolution	9
3.1.2 Provider-Driven Evolution	10
3.1.3 Shared Governance	10
3.2 Forward Direction: DIP Applied Per Client Implies ISP (Universal)	10
3.3 Reverse Direction: ISP Implies DIP Applied Per Client (Client-Driven Case)	11
3.4 Summary Table	12
4. Worked Example Across Three Evolution Origins	14
4.1 Setup	14
4.2 Client-Driven Evolution	14
4.3 Provider-Driven Evolution	15
4.4 Shared Governance	16
4.5 Summary	17
5. Why the Connection Went Unnoticed	18
5.1 The Heuristics Were Never Stated Formally	18
5.2 DIP Was Taken for Granted	18
5.3 ISP Was Never the Strongest Heuristic	19
5.4 DIP's Paper Uses a Single Client	19
5.5 Different Symptoms, Different Problems	20
5.6 Three Different Causal Contexts Complicated Identification	20
5.7 The Community Already Knew the Heuristics Were Not Independent	21
6. Implications for SOLID	22
6.1 Causal Coherence and the Absurd-Literal Distinction	22
6.2 ISP Makes Explicit What DIP Implicitly Requires	22
6.3 The Reduction of SOLID	23
6.4 Pedagogical Consequence	24
7. Conclusion	25

References

Chapter 1

Introduction

The SOLID heuristics — Single Responsibility (SRP), Open/Closed (OCP), Liskov Substitution (LSP), Interface Segregation (ISP), and Dependency Inversion (DIP) — are widely taught as five distinct and independent design directives [1], and practitioners are expected to check each independently and apply each separately.

- Note 1.1 Martin named these design guidelines “Principles” (the P in SRP, OCP, LSP, ISP, DIP), and this paper retains those original names for reader recognition. They are, however, **heuristics** rather than formal principles: empirically grounded rules of thumb, not theorems derived from an axiomatic system. Throughout this paper we qualify them as heuristics while preserving their original names.◊
- This view requires revision for ISP and DIP. The title of this paper uses “conditional corollary” to indicate that the full ISP–DIP equivalence is conditional on evolution origin: the forward direction (DIP applied per client implies ISP) holds universally, but the reverse direction requires client-driven evolution.
- This paper focuses on class-level structural relationships between clients and interfaces. By examining how DIP’s ownership clause operates across the three distinct causal structures of interface evolution—client-driven, provider-driven, and shared governance—we show that ISP makes explicit the granularity constraint already implicit in DIP’s ownership clause under causal interpretation. The forward direction—DIP applied per client implies ISP—holds universally at the class level: applying DIP’s ownership clause once per client–provider edge always produces client-specific interfaces. This is not just structural coincidence; under causal interpretation, DIP’s ownership clause logically implies granularity, so ISP is an explication of what DIP already requires. (Throughout this paper, “DIP→ISP” is shorthand for “DIP applied per client implies ISP,” not “DIP applied once implies ISP”—the per-client qualification is load-bearing.)
- The converse—ISP implies DIP applied per client—holds only under client-driven evolution. When interfaces evolve according to provider strategy or require joint negotiation, ISP can be satisfied without satisfying DIP’s ownership clause. Under these conditions, ISP retains independent structural value: it prescribes client-specific interfaces even when the causal structure that grounds DIP’s ownership clause is absent.
- This paper makes three contributions:
1. A formal statement of the *ownership clause* for DIP, making explicit what Martin’s examples illustrate but his DIP statement does not (Heuristic 2.1.1).

2. A proof that DIP's ownership clause applied per client implies ISP universally at the class level, with the converse holding only under client-driven evolution (Theorem 3.2.1, Corollary 3.3.1).
3. The identification of three distinct *interface evolution origins*—client-driven, provider-driven, and shared governance—as the conceptual framework that explains why the ISP-DIP connection was historically hard to see and why ISP retains independent value outside client-driven contexts.

Chapter 2

Structural Formulations of DIP and ISP

To establish the ISP-DIP relationship, I first restate both heuristics precisely enough to reason about their relationship.

Notation: I use standard set notation throughout this paper:

- $A \subseteq B$ means “A is a subset of or equal to B”
- $A \cup B$ means “the union of sets A and B”
- Structural diagrams use UML notation: \rightarrow (depends on), \dashrightarrow (realizes)

2.1 Dependency Inversion Principle

The classical DIP [2] states: (a) high-level modules should not depend on low-level modules; both should depend on abstractions; and (b) abstractions should not depend on details; details should depend on abstractions.

The two clauses (a) and (b) are stated at the level of dependency direction, not interface ownership. Directional dependency structure alone does not resolve the ownership question.

In Martin’s own examples and books [1], [3], he does address ownership informally: he shows clients defining interfaces and cautions against letting providers drive interface evolution. However, this guidance is presented through worked examples and prose rather than as an explicit clause in DIP’s statement. The gap between Martin’s concrete examples (which show client ownership) and his formal DIP statement (which focuses on directionality) is examined in detail in Section 5.2.

Without specifying that the client—rather than the provider—should define and control the interface, DIP’s prescriptive power is incomplete: one can satisfy the directionality clause (depending on abstractions) while violating what I call the ownership clause (interface owned and evolved by provider).

I therefore add the *ownership clause*—client defines and controls the interface it depends upon—to make DIP’s prescriptive power complete for distinguishing client-driven from provider-driven interfaces. This is a formalization of what Martin’s examples illustrate but his DIP statement does not explicitly state.

The interface abstracts away the details of how the client’s needs are implemented, but only the client knows what it needs. The provider cannot influence what the client requires, so the interface should reflect only the client’s knowledge, not the provider’s preferences.

Example 2.1.1 ▶ Consider a `ReportingService` that must produce external messages. It could directly depend on an `EmailNotificationDispatcher` service, which would mean the client knows that the notification mechanism is implemented by email.

Domain knowledge is silent about how messages are transmitted, requiring only that the system can send them. The architect's choice of email as the delivery mechanism is an implementation decision grounded in organizational infrastructure rather than domain requirements. Because this choice may change as infrastructure evolves, the client should not depend upon it.

Applying DIP allows the client's code to embody its domain requirements precisely: the client needs only the ability to send messages, and this need is captured by the interface on which it depends; the interface itself ignores implementation details. The implementation detail that messages are sent by email is abstracted away from the client, as it lies entirely outside the client's domain of concern.

When the client interface fails to abstract away such implementation details, the client's code becomes contaminated by irrelevant knowledge and the purpose of introducing an interface is defeated. ◇

This client-only influence on interface specification follows from the client's epistemic position: only the client knows what it needs from the interface, so only the client can validly drive interface evolution. DIP's prescriptive power follows from this: the client, not the provider, should drive interface design and evolution. **The heuristic's value lies in enabling faithful representation of domain knowledge without contaminating the client with irrelevant implementation details.**

This ownership requirement does not constrain the provider's implementation: the provider must support at least the functionality required by the client, but the provider may offer additional capabilities beyond the client's current needs. Ownership governs who drives interface change (causality), not what capabilities the provider can implement.

Heuristic 2.1.1 ~ **Dependency Inversion Principle (DIP).**

For every client–provider pair (B, A) , where B depends on services provided by A :

1. *Directionality*: B should depend on an abstraction (an interface) rather than on A directly.
2. *Ownership*: B defines the interface it depends upon; changes to that interface are caused only by changes in B 's requirements.

◇

Note that DIP's ownership clause does not explicitly specify interface size—it requires the client to define and own the interface but says nothing explicitly about how many methods it should include.

The ownership clause admits two interpretations:

1. *Literal interpretation*: the client defines the interface; changes are “caused by changes in the client's requirements” in name only. Under this reading, a client could define a monolithic interface containing unused methods and still satisfy the clause—a degenerate case discussed in Section 6.1.
2. *Causal interpretation*: changes to the interface are *causally grounded* in client requirements—every method in the interface exists because some client requirement caused it to be added. Under this reading, methods without causal

grounding in client requirements cannot appear in the interface. This paper adopts the causal interpretation throughout.

2.2 Interface Segregation Principle

The classical ISP [4] states that many client-specific interfaces are better than one general-purpose interface. This is closely related to Fowler’s concept of *role interfaces* [5]: interfaces defined for the specific role a client plays, rather than for the full capability set of the provider.

ISP addresses a granularity blind spot in DIP’s ownership clause: DIP ensures the client owns the interface but does not explicitly prescribe its size or shape, whereas ISP explicitly prescribes interface granularity.

Heuristic 2.2.2

~ **Interface Segregation Principle (ISP).**

For every provider class A and clients B_1, \dots, B_n :

1. *Granularity*: each client B_i depends on a client-specific interface I_{B_i} declaring only the services B_i actually uses.
2. *Segregation*: no client is forced to depend on a general-purpose interface that contains services it does not need.

◇

ISP forbids a general-purpose interface that serves multiple clients: every client’s dependency set is confined to its own client-specific slice.

ISP also ensures that client code is not contaminated by irrelevant implementation details. When clients depend only on the services they actually use, their code reflects domain-relevant knowledge without extraneous information.

Like DIP, **ISP’s value lies in enabling faithful representation of domain knowledge**. Where the two heuristics differ is in what they prescribe: ISP focuses on interface size and specificity, whereas DIP focuses on dependency directionality and ownership. DIP requires client ownership to be meaningful—without it, the heuristic collapses—while ISP remains coherent under provider ownership (Definition 3.1.3) because who owns the interface itself reflects domain knowledge about that decision.

2.3 Scope of the Two Heuristics

DIP addresses two distinct failure modes: (a) depending on a concrete class with no abstraction at all (directionality violation), and (b) depending on an interface whose specification is not governed by the client’s needs (ownership violation). ISP addresses only the second: it presupposes that interfaces exist and asks whether many client-specific interfaces are used instead of one general-purpose interface. Accordingly, every ISP violation (using one general-purpose interface instead of many client-specific interfaces) is a DIP ownership-clause violation (the interface is governed by the provider, not the client) for some client, but not every DIP violation is an ISP violation — a DIP directionality violation (depending directly on a concrete class with no abstraction) lies outside ISP’s scope.

2.4 Martin’s Hierarchy of Heuristics

Martin himself recognized relationships among his heuristics. In his own words, DIP is “an out-growth of OCP and LSP” [2]:

- OCP (Open/Closed Principle) establishes the goal: modules should be open for extension but closed for modification
- LSP (Liskov Substitution Principle) ensures that derived classes maintain behavioral contracts with their bases
- DIP emerges as the structural mechanism to achieve OCP’s goals by inverting dependencies, which relies on LSP’s substitution guarantees

DIP itself has two components: a directional clause (depend on abstractions, not concretions) and an ownership clause (client defines and controls the interface). The directional clause prevents cascading changes by inverting dependency direction. The ownership clause ensures that the abstraction is governed by the client’s requirements, not the provider’s internal roadmap.

ISP complements this hierarchy by making explicit what DIP’s ownership clause already implies. DIP’s ownership clause states that the client defines the interface, but does not explicitly specify what size or shape that interface should be. However, under causal interpretation, DIP’s ownership clause logically implies granularity: if the client defines the interface and only client requirements cause changes, then the interface cannot contain functionality beyond what the client requires.

ISP serves as the explicit granularity mechanism for DIP’s ownership clause: it states in concrete terms what DIP already requires implicitly through its causal structure. A single general-purpose interface that serves multiple clients violates DIP’s ownership clause under causal interpretation, because changes to methods not used by a given client cannot be causally grounded in that client’s requirements.

Viewed together, the SOLID heuristics form a dependency-management system:

- SRP identifies when elements should be separated (they have multiple reasons to change)
- OCP states the design goal (extension without modification)
- LSP ensures substitution works as expected
- DIP provides the structural mechanism (depend on abstractions, with client ownership)
- ISP refines DIP’s ownership granularity (client-specific interfaces)

This paper focuses on the ISP–DIP relationship because these two heuristics address the same underlying constraint from different perspectives: preventing coupling between elements that require changes for different reasons. DIP’s ownership clause enforces this at the abstraction level; ISP enforces this at the interface granularity level. The formal proof in Chapter 3 shows that DIP applied per client implies ISP at the class level under all interface evolution conditions.

Chapter 3

ISP Is a Conditional Corollary of DIP Applied Per Client

The proof uses only the definitions of DIP and ISP restated in Chapter 2—no additional formal machinery is required.

3.1 Interface Evolution Origins

Before proving the ISP–DIP relationship, I must distinguish three distinct causal structures for interface evolution. The causal structure determines when the ISP–DIP relationship holds or fails.

Remark 3.1.1 *Interface evolution origin* refers to the causal source of changes to an interface’s specification (added methods, removed methods, modified contracts). ◇

Definition 3.1.1

◆ **Interface Ownership.**

An interface is *owned* by the party that unilaterally controls changes to its specification: only that party’s requirements or strategy can cause additions, removals, or modifications to the interface’s methods and contracts. ◇

Three distinct origins of interface evolution exist:

3.1.1 Client-Driven Evolution

Definition 3.1.2

◆ **Client-Driven Evolution.**

An interface exhibits *client-driven evolution* when all changes to its specification originate in the requirements of the client modules that depend on it. The provider implements interface changes in response to client requirements. ◇

Examples:

- Internal service boundaries where provider is a subcomponent of client’s module
- Contract-first API design where client dictates service terms
- Domain-driven design with upstream domains defining downstream contracts

Manifestation: Interface changes originate in business requirements flowing from client to provider. Provider has no independent roadmap for this interface.

Worked Example: An e-commerce system defines a `PaymentProcessor` interface in its checkout module. When checkout needs fraud detection, client adds

`detectFraud()` method to `PaymentProcessor`. Payment service implements this change immediately because interface exists to serve checkout requirements.

3.1.2 Provider-Driven Evolution

Definition 3.1.3

◆ **Provider-Driven Evolution.**

An interface exhibits *provider-driven evolution* when all changes to its specification originate in the provider's independent product strategy or technical roadmap. Clients adapt their usage to accommodate provider-initiated changes. ◇

Examples:

- Third-party APIs (Stripe, AWS SDKs, database drivers)
- Platform-as-a-Service offerings
- Libraries with multi-tenant consumers

Manifestation: Interface changes originate in provider's independent product roadmap. Clients are consumers of provider-driven evolution.

3.1.3 Shared Governance

A third origin exists when neither client nor provider exclusively controls interface evolution.

Definition 3.1.4

◆ **Shared Governance Evolution.**

An interface exhibits *shared governance evolution* when changes to its specification can originate from either client requirements or provider strategy, and such changes require explicit negotiation and coordination between both parties. ◇

Examples:

- B2B integration contracts between independent organizations
- API version management with backward compatibility commitments
- Industry-standard specifications with multiple implementors

Manifestation: Interface evolution is jointly controlled. Neither party unilaterally dictates interface changes.

3.2 Forward Direction: DIP Applied Per Client Implies ISP (Universal)

Having distinguished three evolution origins, I now prove the ISP–DIP relationship. The forward direction holds universally, while the reverse direction requires an additional condition.

Theorem 3.2.1

◇ **ISP Follows from Per-Client DIP.**

Let A be a provider class with services S_A and clients B_1, \dots, B_n ($n \geq 2$) using $S_{B_1}, \dots, S_{B_n} \subseteq S_A$ respectively, where the clients collectively exhaust A 's total services ($S_{B_1} \cup \dots \cup S_{B_n} = S_A$).

Hypothesis: Consider only the class-level structure — which classes, interfaces, and dependency edges exist — without any assumption about packaging or modularization.

Applying DIP's ownership clause independently to each client-provider edge produces interfaces $I_{B_i} = S_{B_i}$ with: (i) $I_{B_1} \cup \dots \cup I_{B_n} = S_A$ (coverage follows from client completeness), and (ii) no client depends on any service it does not use. That is, the design satisfies ISP (Heuristic 2.2.2). \square

Proof. (direct from the DIP construction) Each application of DIP's ownership clause produces an interface containing exactly the services the client uses: $I_{B_i} = S_{B_i}$. Client B_i depends only on I_{B_i} and sees S_{B_i} . No client is exposed to services it does not use — this is exactly the ISP condition of Heuristic 2.2.2.

(i) **Coverage:** By the completeness hypothesis, $S_{B_1} \cup \dots \cup S_{B_n} = S_A$, so $I_{B_1} \cup \dots \cup I_{B_n} = S_A$.

(ii) **Segregation:** Each client sees only its own slice. If the S_{B_i} are pairwise disjoint, the interfaces are disjoint as well. If they overlap, the shared services appear in multiple interfaces — but each client still depends only on its own interface, so no client is forced to depend on services it does not use. ISP is satisfied in either case. ■

This forward direction holds universally. Applying DIP's ownership clause per client—client defines the interface it depends upon—produces client-specific interfaces regardless of who actually drives interface evolution. The construction is purely structural: the resulting interfaces declare exactly what each client uses, which is precisely ISP's requirement.

The forward direction's universality follows from DIP's prescriptive nature: applying DIP's ownership clause per client specifies client-specific interfaces as the design outcome. The prescriptive structure holds regardless of causal context.

Why this is universal:

DIP's ownership clause states: “client defines the interface it depends upon.” Applying this clause per client-provider edge means:

- Each client B_i defines an interface $I_{\{B_i\}}$ declaring exactly what B_i uses
- Provider A implements all interfaces $I_{\{B_i\}}$
- No client depends on methods it does not use

This construction does not reference who drives evolution. The clause is purely prescriptive about design structure. Whether in practice the provider independently evolves interfaces (provider-driven) or clients drive changes (client-driven), applying DIP per client always produces the same structural result: client-specific interfaces satisfying ISP.

3.3 Reverse Direction: ISP Implies DIP Applied Per Client (Client-Driven Case)

Corollary 3.3.1

⇒ **ISP ⇔ Per-Client DIP When Client Has Sole Influence.**

ISP and per-client DIP are structurally equivalent if and only if the client has sole influence over interface evolution. Equivalence fails whenever the provider influences the interface (through ownership, shared governance, or any form of participation). \square

Proof. (the forward direction is Theorem 3.2.1; here I prove when equivalence holds)
Converse (ISP ⇒ per-client DIP):

Step 1: Provider influence breaks equivalence. By definition of ownership (Definition 3.1.1), ownership is exclusive to a single party that unilaterally controls changes. If the provider influences interface evolution (whether through full ownership, shared governance, or any form of participation), then DIP’s ownership clause (Heuristic 2.1.1) fails by definition: the ownership clause requires client ownership, and provider influence contradicts this requirement. Therefore DIP is false. Under this condition, DIP is false while ISP may be true (the provider can still segregate interfaces by client). Since DIP and ISP can have different truth values, they are not equivalent. Equivalence is impossible in any provider-influence case.

Step 2: Client-only influence yields equivalence. What remains is the case where the client has sole influence over interface evolution. Under this condition, interface changes originate entirely from client requirements. Since ISP holds (the interface has been segregated into client-specific slices I_{B_1}, \dots, I_{B_n}), each client B_i depends on exactly the methods it uses via I_{B_i} , satisfying DIP’s directionality clause (Heuristic 2.1.1). Moreover, because B_i drives all evolution of I_{B_i} , ownership of I_{B_i} belongs to B_i by the definition of interface evolution origin (Definition 3.1.2). This satisfies DIP’s ownership clause (Heuristic 2.1.1), which requires that the client defines and controls the interface it depends upon. Therefore both ISP and DIP are true, and the equivalence holds. ■

The forward direction holds universally regardless of causal context. The reverse direction holds under the client-driven evolution condition defined in Definition 3.1.2.

Why client-driven assumption is necessary:

Under provider-driven evolution, ISP can be satisfied without satisfying DIP’s ownership clause. For example, a third-party API provider may expose client-specific interfaces (I_{B_1}, \dots, I_{B_n}) that segregate methods by client use, satisfying ISP, yet the provider—not clients—owns and evolves these interfaces. The causal source of interface changes is the provider’s product strategy, not client requirements, so DIP’s ownership clause fails even though ISP holds.

Under shared governance, the interface is governed by negotiation between parties. Neither heuristic’s ownership model applies: the client does not unilaterally define the interface, and the provider does not unilaterally control it either. The interface becomes an independent coordination artifact.

3.4 Summary Table

Table 3.1

The forward direction $DIP \rightarrow ISP$ holds universally across all evolution origins, but $ISP \rightarrow DIP$ holds only under client-driven evolution.

Evolution origin	Forward direction ($DIP \rightarrow ISP$)	Reverse direction ($ISP \rightarrow DIP$)
Client-driven	Universal	Holds Corollary 3.3.1
Provider-driven	Universal	Fails: ISP without client ownership possible
Shared governance	Universal	Fails: neither ownership model applies

The forward direction’s universality reflects that DIP applied per client prescribes client-specific interfaces regardless of causal context. The reverse direction’s context-dependence reflects that ISP’s structural prescription does not entail client ownership unless the causal structure makes client ownership the natural outcome.

Observation 3.4.1

⊙ **Why “Per Client” and Not “Once”.**

If DIP is applied only once — creating a single shared abstraction I_A declaring all of S_A — both clients depend on the full interface through I_A . The dependency direction is inverted (both depend on an abstraction, not a concretion), but B still sees S_C ’s methods and C still sees S_B ’s methods. ISP is still violated.

The segregation emerges precisely because each dependency edge is inverted independently. Each inversion asks “what does *this* client need?” — and the answer is a client-specific slice of S_A .

- DIP applied once (shared abstraction): one fat interface \rightarrow ISP violated.
- DIP applied per client: segregated interfaces \rightarrow ISP satisfied.

◇

Chapter 4

Worked Example Across Three Evolution Origins

The formal proof in Chapter 3 established the relationship between ISP and DIP across the three evolution origins. This chapter illustrates that relationship concretely using a notification system comprising one provider class (*A*) and two client classes (*B* and *C*).

By examining how ISP and DIP interact under each evolution origin, we can see why the forward direction (DIP applied per client implies ISP) holds universally while the reverse direction (ISP implies DIP applied per client) holds only under client-driven evolution.

The worked example demonstrates that the formal result is not merely theoretical—it reflects structural patterns that appear in real systems and explains why Martin’s ISP examples (often provider-driven) do not obviously derive from DIP.

4.1 Setup

Three classes:

- *A* = `NotificationDispatcher`, provides:
 - `sendConfirmationEmail`,
 - `sendShippingSMS`,
 - `sendAlertEmail`.
- *B* = `OrderService`: uses
 - `A.sendConfirmationEmail`,
 - `A.sendShippingSMS`.
- *C* = `ReportingService`: uses
 - `A.sendAlertEmail`.

B and *C* need different services from *A*.

4.2 Client-Driven Evolution

Context: `OrderService` and `ReportingService` are internal modules within the same product. When business requirements change, these modules drive interface evolution. `NotificationDispatcher` is a subcomponent implementing interfaces defined by its clients.

Fat-Interface Design (DIP Applied Once):

In the naive design, a single shared abstraction I_A declares all three methods, and both B and C depend on it:

$$\begin{aligned} B &\rightarrow I_A \leftarrow A \\ C &\rightarrow I_A \end{aligned}$$

The dependency direction is inverted (both clients depend on an abstraction rather than a concretion), satisfying DIP's directionality clause—but no single client owns I_A , violating DIP's ownership clause. Furthermore, B still sees `sendAlertEmail` (which it does not use) and C still sees `sendConfirmationEmail` and `sendShippingSMS` (which it does not use). ISP is violated.

DIP Applied Per Client:

Step 1: invert the $B \rightarrow A$ edge. B defines its own interface, declaring the services it actually uses:

$$I_B = \{\text{sendConfirmationEmail}, \text{sendShippingSMS}\}.$$

B depends on I_B ; A implements I_B .

Step 2: invert the $C \rightarrow A$ edge. C defines its own interface:

$$I_C = \{\text{sendAlertEmail}\}.$$

C depends on I_C ; A implements I_C .

Step 3: observe the result.

$$B \rightarrow I_B \leftarrow A \twoheadrightarrow I_C \leftarrow C$$

Table 4.1

Applying DIP per client eliminates interface bloat so no client depends on methods it does not use.

Client	Depends on	Sees
B	I_B	<code>sendConfirmationEmail</code> , <code>sendShippingSMS</code>
C	I_C	<code>sendAlertEmail</code>

No client depends on methods it does not use. The fat interface has been segregated into two client-specific interfaces — this is ISP, obtained by applying DIP once per client-provider edge.

ISP Implies DIP Applied Per Client:

Starting from ISP compliance (the structure above), interfaces I_B and I_C exist and are client-specific. Under client-driven evolution, each interface's owner is determined by the causal source of changes: when `OrderService` needs a new notification feature, it adds the method to I_B and `NotificationDispatcher` (A) implements the change. The causal structure satisfies DIP's ownership clause. Under client-driven evolution, ISP and per-client DIP are equivalent.

4.3 Provider-Driven Evolution

Context: `NotificationDispatcher` (A) is a third-party notification service (an external API like Twilio or AWS SNS). The provider independently evolves its API (I_{provider}) based on its product strategy. `OrderService` and `ReportingService` are consumers adapting to provider-driven changes.

Fat-Interface Design (Single Provider API):

The provider exposes a single `NotificationService` interface declaring all methods. Both clients depend on this provider interface.

$$\begin{aligned} B &\rightarrow I_{\text{provider}} \leftarrow A \\ C &\rightarrow I_{\text{provider}} \end{aligned}$$

The fat interface violates ISP, but the provider may later offer client-specific interfaces as part of its multi-client strategy.

Provider Offers Client-Specific Interfaces:

The provider introduces segregated interfaces:

$$\begin{aligned} I_{\text{order}} &= \{\text{sendConfirmationEmail}, \text{sendShippingSMS}\} \\ I_{\text{report}} &= \{\text{sendAlertEmail}\} \\ B &\rightarrow I_{\text{order}} \leftarrow A \twoheadrightarrow I_{\text{report}} \leftarrow C \end{aligned}$$

From a structural perspective, this is identical to the client-driven case: each client depends only on methods it uses. ISP is satisfied.

ISP Does NOT Imply DIP Applied Per Client:

Although ISP holds, DIP's ownership clause fails. The interfaces are owned by the provider, not by the clients. If the provider adds a `sendWebhook` method to I_{report} because it suits its platform strategy, `ReportingService` must adapt to the change it did not initiate. The client does not define or own the interface. ISP is satisfied, but per-client DIP is not.

DIP Applied Per Client Still Implies ISP:

If we apply DIP's ownership clause per client (hypothetically—clients define their own interfaces), the result is structurally identical to the provider's segregated interfaces. The forward direction remains universal: prescribing client-specific interfaces always satisfies ISP, regardless of who actually drives evolution.

4.4 Shared Governance

Context: `NotificationDispatcher` and its clients belong to different organizations negotiating an integration contract. Interface changes require explicit coordination and version negotiation between parties.

Shared Interface Specification:

A joint `NotificationContract` interface is defined through negotiation. Both parties must agree on additions like `sendConfirmationEmail`:

- Provider must implement it.
- Clients must be prepared to use it.
- Breaking changes require contract amendments.

$$\begin{aligned} B &\rightarrow I_{\text{contract}}^{\text{negotiated}} \leftarrow A \\ C &\rightarrow I_{\text{contract}}^{\text{negotiated}} \end{aligned}$$

Shared Governance with Segregation:

The negotiated contract can specify client-specific slices:

$$I_{\text{order}}^{\text{negotiated}} = \{\text{sendConfirmationEmail}, \text{sendShippingSMS}\}$$

$$I_{\text{report}}^{\text{negotiated}} = \{\text{sendAlertEmail}\}$$

$$B \rightarrow I_{\text{order}}^{\text{negotiated}} \leftarrow A \leftrightarrow I_{\text{report}}^{\text{negotiated}} \leftarrow C$$

Structurally, this again satisfies ISP.

ISP Does NOT Imply DIP Applied Per Client: Neither pure DIP (client owns interface) nor pure provider-driven ISP (provider owns interface) captures the causal reality. The interface is an independent coordination artifact governed by negotiation, not by unilateral ownership from either party. ISP holds structurally (because ISP doesn't make assumptions about who owns the client interfaces), but DIP's ownership assumption is too strong for this context.

Practical Guidance:

Under shared governance, version management and breaking-change policies become load-bearing concerns. When a new feature requires interface change, both parties must negotiate, among others:

- Major version bump for breaking changes.
- Minor version for backward-compatible additions.
- Deprecation period before removal.

Neither DIP nor ISP alone provides such coordination guidance.

4.5 Summary

Table 4.2
DIP universally implies ISP across all evolution origins, but ISP only implies DIP under client-driven evolution.

Evolution origin	ISP structure	ISP → DIP applied per client	DIP applied per client → ISP
Client-driven	Client-specific interfaces	✓ Biconditional holds Corollary 3.3.1	✓ Universal
Provider-driven	Client-specific interfaces	✗ Provider owns interfaces	✓ Universal
Shared governance	Client-specific interfaces	✗ Neither ownership model applies	✓ Universal

The forward direction—DIP applied per client implies ISP—holds universally across all three contexts. The reverse direction—ISP implies DIP applied per client—holds only under client-driven evolution. This explains why Martin's ISP examples (provider-driven contexts) do not obviously derive from DIP, while his ISP origin story (client-driven Xerox system) does exhibit the causal connection.

Chapter 5

Why the Connection Went Unnoticed

Martin published DIP in May 1996 [2] and ISP in August 1996 [4], yet they have been taught as independent heuristics ever since. Given the close structural relationship established in Chapter 3, why was this connection not identified for nearly thirty years?

This chapter examines the historical and conceptual factors that obscured the derivation. The formal result in Chapter 3 stands on its own merits regardless of this historical explanation. The factors below are offered as context, not as argument for the result.

5.1 The Heuristics Were Never Stated Formally

DIP and ISP are informal design directives—English sentences illustrated by examples—rather than formal propositions. Without precise definitions, logical derivation between design heuristics cannot be attempted. The connection becomes visible only when both are stated in formal terms, as this paper does in Chapter 2.

Martin’s reasoning in both papers relies on concrete examples and design intuition. This pedagogical approach does not raise the question of whether one heuristic follows from the other—that question has no meaning until the heuristics are stated in a form that admits logical inference. As Henney [6] observed, SOLID’s heuristics are better described as patterns than as formal principles, and patterns are not the kind of object one derives from another.

Both heuristics appear simple: DIP states “depend on abstractions,” and ISP states “don’t depend on what you don’t use.” Neither invites deeper questioning, and the casual wording obscures that “the client defines the interface” and “the interface declares only what the client needs” express the same constraint—as the proof in Chapter 3 shows.

5.2 DIP Was Taken for Granted

DIP’s formulation is broad and its examples are simple. Most practitioners and textbooks treat DIP’s core directive — depend on abstractions — as self-evident. The ownership concept appears in Martin’s examples and informal guidance,

but is not stated as an explicit clause alongside (a) and (b) in DIP's formal statement. Because directionality is the headline, ownership received less scrutiny. The ownership clause was underemphasized relative to the directionality clause, obscuring ISP's derivability.

Martin never formulated ownership as a separate, explicit clause alongside (a) and (b) in his DIP paper. His examples operate within a layered architecture convention, where interfaces conventionally live within providers (the lower layer). He introduces the layering section of his DIP paper [2] by citing Booch's definition [7] and structures his examples within that convention. Our formalization separated directionality (depend on abstractions) from ownership (client defines the interface), and this separation revealed DIP→ISP.

The same separation surfaces a related issue at the package level: where should client-specific interfaces live in the module structure? That question is outside the scope of this paper, which focuses solely on class-level structure. Martin's layered-architecture examples consistently place interfaces in the provider package, but when dependency direction is formalized independently of placement assumptions, the question of where interfaces belong becomes unavoidable. The class-level derivation and the package-level placement conflict are thus two manifestations of the same phenomenon: layered architecture's implicit placement assumption masked design questions that only appear when the formulation makes the assumption explicit.

5.3 ISP Was Never the Strongest Heuristic

Among the five SOLID heuristics, ISP has always attracted the least independent attention. Henney [6] argued that ISP becomes redundant if SRP is properly applied. Miller [8] questioned whether ISP was included primarily to complete the acronym. Kaminski [9] noted confusion about ISP's vagueness and identified multiple incompatible interpretations. ISP is frequently nominated for elimination from SOLID, but the debate has consistently been ISP versus SRP, not ISP versus DIP.

Because ISP was regarded as the weakest of the five, it did not receive the kind of formal analysis that would have revealed its derivability from DIP.

5.4 DIP's Paper Uses a Single Client

Martin's DIP paper [2] primarily illustrates DIP with single client-provider pairs; the multi-client case—one provider, several clients with different needs—is not prominently featured or explicitly analyzed. ISP emerges when DIP's construction is applied per client to a multi-client scenario, where each client-provider edge receives its own interface rather than a shared abstraction. With a single client, DIP produces a single interface and ISP is trivially satisfied. The interesting structure appears only when the same construction is repeated for each client-provider edge.

Textbook treatments of DIP typically use isolated two-class pairs, and the multi-client case rarely appears on the same page. The per-client structure becomes

visible only when all client edges of the same provider are drawn in a single diagram.

5.5 Different Symptoms, Different Problems

DIP and ISP address different problems—wrong dependency direction versus oversized interfaces—but the resulting design structure is the same when each is applied completely, as the proof in Chapter 3 shows. Because the papers address different problems, it was natural to treat them as independent heuristics. The shared mechanism was hidden behind different problem descriptions.

5.6 Three Different Causal Contexts Complicated Identification

The connection was further obscured because real-world systems exhibit three distinct causal structures for interface evolution (Definition 3.1.2, Definition 3.1.3, Definition 3.1.4).

In client-driven systems—internal modules where the client owns the interface—DIP and ISP prescribe identical structure, and the derivation is direct. If one has only seen client-driven examples (Martin’s Xerox story, internal service boundaries), the equivalence appears obvious once stated formally.

In provider-driven systems—third-party APIs and libraries—ISP can be satisfied without DIP’s ownership clause. Here, the structural similarity appears coincidental rather than causal. A designer who primarily works with external APIs sees ISP as a structural rule (segregate interfaces per client) that doesn’t connect to DIP’s ownership clause (which cannot apply—the client doesn’t own the third-party’s interface). The pattern looks “structural accident” rather than causal necessity.

In shared-governance systems—B2B integration contracts—neither heuristic fully captures the causal reality. The interface is governed by explicit negotiation, not by unilateral ownership. A B2B integration architect sees ISP as a framework for coordinating changes through version contracts, and DIP’s ownership assumption seems irrelevant (neither party owns the interface unilaterally). The pattern appears to belong to “contract design,” not dependency heuristics.

Because each context presents a different face of the same structural pattern, the underlying causal connection was hard to see. One practitioner experiences DIP→ISP equivalence as obvious (client-driven context), another experiences it as inapplicable (provider-driven context), and a third finds it irrelevant (shared-governance context). The pattern’s multi-context presence masked its single-context derivability.

Martin’s own ISP origin story [4] describes a client-driven context (the Xerox system’s internal modules), yet his ISP examples in practice papers often show provider-driven contexts (third-party libraries). The same structural pattern—each client depending on a distinct interface that declares exactly that client’s used methods—arises in all three contexts, but only in the client-driven case does it reflect DIP’s causal mechanism. This structural pattern’s apparent uni-

versality (client-specific interfaces appear everywhere interfaces are segregated, regardless of evolution origin) masked its conditional derivability from DIP (the pattern follows from DIP's ownership clause only when client-driven evolution holds).

Martin's Xerox solution created per-client interfaces (one for stapling, one for printing) by inserting an abstraction layer between each client and the provider. Martin did not describe this as "applying DIP per client" — the structural mechanism is recognizably the same, but he named the result ISP rather than identifying it as a consequence of DIP.

5.7 The Community Already Knew the Heuristics Were Not Independent

The observation that SOLID's five heuristics are not all independent is not new. Martin himself noted connections between DIP, OCP, and LSP [2]. Oldwood [10] argued that all five reduce to two concepts (separation of concerns and programming to an interface). Kaminski [9] noted that SRP, ISP, and DIP "rehash similar concepts around managing dependencies." Terhorst-North [11] proposed replacing SOLID entirely with a different framework (CUPID).

Yet, to the author's knowledge, none of these critiques identified the *specific* derivation: that DIP's ownership clause, applied once per client, *is* ISP. The community sensed overlap without locating the precise mechanism.

Chapter 6

Implications for SOLID

The result established by Theorem 3.2.1 has consequences for how SOLID is analyzed and taught.

This chapter examines three implications: the independent design guidance ISP provides, the reduction of SOLID’s five heuristics to at most four independent ideas, and the pedagogical consequences for teaching DIP.

6.1 Causal Coherence and the Absurd-Literal Distinction

The ISP-DIP relationship depends on how DIP’s ownership clause is interpreted. Under a literal but absurd interpretation, DIP’s ownership clause could be satisfied by a client defining a monolithic interface containing functionality they never use, as long as changes to that interface are “caused by changes in the client’s requirements” – a logical impossibility for methods the client doesn’t need.

Under a causally coherent interpretation, DIP’s ownership clause logically implies granularity: if the client defines the interface and only client requirements cause changes, then the interface cannot contain anything beyond what the client requires, otherwise those additions would lack causal grounding in client requirements.

This paper adopts the causally coherent interpretation throughout. The absurd-literal interpretation is mentioned only to motivate why the causal reading is necessary: without it, the ownership clause admits degenerate designs that contradict its own rationale.

6.2 ISP Makes Explicit What DIP Implicitly Requires

Under client-driven evolution (Definition 3.1.2), ISP does not add independent design guidance beyond what DIP already prescribes – it makes explicit what DIP’s ownership clause already implies.

DIP’s ownership clause states that the client defines the interface and that changes are caused only by changes in the client’s requirements. Interpreted causally coherently, this logically implies granularity: the interface cannot contain methods the client doesn’t use, because changes to such methods would lack causal grounding in client requirements. A “fat interface” owned by the client

is not a design that satisfies DIP’s ownership clause under causal interpretation – it’s a category error masquerading as compliance.

ISP serves as the explicit granularity mechanism for DIP’s ownership clause: it states in concrete terms what DIP already requires implicitly through its causal structure. The relationship is therefore one of explication, not addition: ISP operationalizes the granularity constraint that is already logically embedded in DIP’s ownership clause.

This equivalence breaks down under provider-driven or shared-governance evolution. When interfaces are owned by providers (third-party APIs) or governed by negotiation (B2B contracts), the causal structure that grounds DIP’s ownership clause is absent. In these contexts, ISP retains independent structural value: it still prescribes client-specific interfaces even when the causal structure prevents client ownership. ISP’s value under provider-driven and shared-governance evolution is structural, not derivational.

6.3 The Reduction of SOLID

The ISP–DIP equivalence shows that SOLID’s five heuristics contain at most four independent ideas:

Table 6.1
ISP makes explicit the granularity constraint already implicit in DIP’s ownership clause under causal interpretation; retains structural value when client ownership is impossible.

Heuristic	Status	Note
DIP	Foundational (this paper)	Client ownership of interfaces is the load-bearing clause; granularity is logically implicit under causal interpretation
ISP	Explication of DIP (client-driven case)	Makes explicit the granularity constraint already implicit in DIP’s ownership clause (Theorem 3.2.1); independent structural value under provider-driven and shared-governance evolution
SRP	Independent	Addresses cohesion within a class; not derivable from DIP alone
OCP	Independent	Addresses whether providers can be extended without breaking existing clients; a formal derivation relat-

Heuristic	Status	Note
LSP	Independent	<p>ing OCP to DIP has not yet been established</p> <p>Addresses whether subclasses can replace their base classes without breaking behavior; this is about runtime behavior, not the dependency graph</p>

6.4 Pedagogical Consequence

When teaching DIP, ownership is critical: the client defines and owns the interface it depends upon. If this clause is taught carefully—with the per-client application made explicit—ISP follows as an exercise rather than as a separate heuristic. The question “which interface is too large?” is the same as asking “for which client did I fail to apply DIP’s ownership clause?”

Chapter 7

Conclusion

This paper proves that the Interface Segregation Principle makes explicit the granularity constraint already implicit in DIP's ownership clause under causal interpretation.

This result required introducing a fundamental concept that was previously implicit in SOLID discussions but never formalized:

Interface ownership and evolution source: which party controls changes to an interface's specification and where those changes originate. Three distinct causal structures exist: client-driven evolution (changes originate in client requirements), provider-driven evolution (changes originate in provider strategy), and shared governance evolution (changes require negotiation).

The distinction among these three evolution origins is essential for understanding the ISP-DIP relationship. In client-driven systems, DIP's ownership clause logically implies granularity, making ISP an explication of what DIP already requires. Under causal interpretation, the forward direction (DIP applied per client implies ISP) holds universally, while the reverse direction holds only under client-driven evolution; the full biconditional equivalence requires client-driven evolution as an explicit condition. In provider-driven and shared-governance systems, ISP provides structural guidance independent of DIP's causal mechanism, revealing that ISP retains value even when the causal structure grounding DIP is absent.

By making this concept explicit, the apparent contradiction between DIP's ownership clause and Martin's ISP examples dissolves. Martin's ISP origin story (Xerox internal modules) describes client-driven evolution, where ISP and DIP are equivalent under causal interpretation. His ISP examples often show provider-driven contexts, where ISP provides structural guidance independent of DIP's causal mechanism.

The forward direction—DIP applied per client implies ISP—holds universally at the class level, regardless of where interface evolution originates. Applying DIP's ownership clause per client always produces client-specific interfaces because the clause is purely prescriptive about design structure, not about causal reality.

The reverse direction—ISP implies DIP applied per client—holds only under client-driven evolution (Corollary 3.3.1). When interfaces are provider-driven (third-party APIs) or governed by shared negotiation (B2B contracts), ISP can be satisfied without satisfying DIP's ownership clause.

Several prior authors — Henney [6], Oldwood [10], Kaminski [9], and others — have observed that SOLID’s five heuristics are not all independent. To the author’s knowledge, the specific derivation presented here—DIP applied per client–provider edge is ISP—has not previously been identified in the literature. The connection went unnoticed for nearly thirty years because both heuristics were stated as informal design directives rather than formal propositions.

This result settles the structural relationship between ISP and DIP but leaves open a subsequent question: in multi-module systems, where should client-specific interfaces live in the package structure? This packaging question extends the class-level analysis presented here to module-level organization and is left to future work.

Several limitations of this analysis should be noted. First, the ownership clause is this paper’s addition to DIP, not Martin’s original formulation; the result holds for the **augmented** DIP defined in Heuristic 2.1.1, and a different formalization of ownership might yield a different relationship. Second, the proof considers only flat interfaces without inheritance hierarchies; whether the derivation extends to interface extension or generic type parameters is not addressed. Third, the paper restricts itself to class-level structure and does not account for module boundaries, package visibility, or deployment units. Fourth, the “causally coherent” interpretation of DIP’s ownership clause is treated as self-evident rather than formally derived from the clause’s wording; this interpretive stance is defended in Section 6.1 but remains a philosophical choice, not a theorem.

References

- [1] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2003.
- [2] R. C. Martin, “The Dependency Inversion Principle,” *C++ Report*, May 1996.
- [3] R. C. Martin, *Clean Architecture*. Prentice Hall, 2018.
- [4] R. C. Martin, “The Interface Segregation Principle,” *C++ Report*, Aug. 1996.
- [5] M. Fowler, “Role Interface.” June 2006.
- [6] K. Henney, “SOLID Deconstruction.” Jan. 2016.
- [7] G. Booch, *Object Solutions: Managing the Object-Oriented Project*. Addison-Wesley, 1996.
- [8] J. Miller, “Putting SOLID into Perspective.” Aug. 2022.
- [9] T. Kamiński, “Deconstructing SOLID design principles.” Apr. 2019.
- [10] C. Oldwood, “KISSing SOLID Goodbye,” *ACCU Overload*, no. 122, Aug. 2014.
- [11] D. Terhorst-North, “CUPID — for joyful coding.” 2022.