```
#%% md
---
title: "  The Hidden Geometry of Software Coupling (Part 1)"
date: 2026-03-09 00:11:50 -0500
categories:
  - sdlc
  - architecture
  - coupling
  - metrics
author: steven
---

#    The Hidden Geometry of Software Coupling
### Part 1 — The Metrics That Predict Architectural Failure

#%%
# ============================================================
# SETUP — Run this first!
# ============================================================

# Install plotly if needed (uncomment):
```

```
# !pip install  plotly pandas

import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd
import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots
import numpy as np

print("   Plotly version:", go.__version__ if hasattr(go, '__version__') else 'installed')
print("   matplotlib version:", mpl.__version__ if hasattr(mpl, '__version__') else 'installed')
print("   Setup complete!")

# ============================================================
#    CONFIGURATION
# ============================================================

CONFIG = {
```

```
    # Background colors
    'paper_bg': '#FCF5E5',
    'plot_bg': 'rgba(252,245,229,0.8)',

    # Instability colorscale (list of [position, color])
    # Position: 0.0 to 1.0, Color: any CSS color
    'instability_colorscale': [
        [0.00, '#FFFFFF'],   # White at I=0
        [0.05, '#6AD8D8'],   # Light teal
        [0.20, '#0D3D3D'],   # Dark teal
        [0.30, '#E0B060'],   # Light bronze
        [0.45, '#5A3A12'],   # Dark bronze
        [0.50, '#3A0808'],   # Darkest red (center)
        [0.55, '#E89048'],   # Light copper
        [0.70, '#6A3210'],   # Dark copper
        [0.80, '#8A4A9A'],   # Light purple
        [0.95, '#2A0A3A'],   # Dark purple
        [1.00, '#000000'],   # Black at I=1
    ],

    # Distance colorscale (green=good, red=bad)
```

```
    'distance_colorscale': [
        [0.0, '#2E7D32'],    # Dark green (D=0, ideal)
        [0.2, '#4CAF50'],    # Green
        [0.4, '#8BC34A'],    # Light green
        [0.5, '#FFEB3B'],    # Yellow
        [0.7, '#FF9800'],    # Orange
        [0.85, '#F44336'],   # Red
        [1.0, '#B71C1C'],    # Dark red (D=1, worst)
    ],

    # Zone colors for Main Sequence
    'zone_pain': 'rgba(232,180,180,0.6)',
    'zone_useless': 'rgba(180,180,232,0.6)',
    'main_seq': '#4CAF50',

    # Default camera for 3D plots
    'camera': dict(
        eye=dict(x=1.5, y=1.5, z=1.2),
        up=dict(x=0, y=0, z=1)
    ),

```

```
    # Mesh resolution
    'mesh_points': 100,  # Lower than matplotlib for performance
}

print("   Configuration loaded")
#%% md
##   Introduction

Software engineers love to talk about architecture in
qualitative terms.

- *“This module feels tightly coupled.”*
- *“That dependency seems risky.”*
- *“This design is flexible.”*

But beneath those instincts lies something far more concrete.

The **structure of software systems can be measured**.

And those measurements often predict architectural problems **
long before production failures reveal them**.
```

```
102
103 The formulas behind these metrics have been around since the
    1990s.
104 They require no machine learning.
105
106 Just counting.
107
108 And occasionally a little division.
109
110 #%% md
111 ##   The Architecture That Was “Good Enough”… Until It Opened a
    Hellmouth
112
113 _For the first six months, everything felt fast.  Our Ruby-on-
    Rails application was humming along:_
114
115 * Features shipped quickly.
116 * Bug fixes took hours, not days.
117 * Engineers felt productive.
118
119 _Then, something strange started happening; it began to shift:_
```

```
120 * A simple change began taking longer.
121 * A feature that should have touched one component suddenly
    required edits across:
122   - seemingly-unrelated models and controllers
123   - background jobs
124   - serializers
125   - service classes
126   - many tests scattered across the codebase, some that seem
    unrelated.
127
128 _Then the real symptoms appeared:_
129 * New engineers joined the team and couldn't make heads or tails
     of the system.
130 * Bug fixes triggered unrelated failures.
131 * A “small refactor” broke three features nobody expected to be
    connected.
132 * Every change started to feel dangerous.
133
134 Eventually they bring in an architect like YT.  I've seen this
    so much, I spend a bit of time assembling the tools I need to
    report on the appropriate metrics and report on them (over time
```

```
134 , assuming use of SCM, git, svn, etc.)
135
136 They are told:
137 > “Your problem isn’t Rails. Your problem is **coupling**."
138 > Here are your danger areas from worst to least.
139
140 Some people remember this from _Software Engineering Principles_
     (or whateverz the course/book/article you read about this).  It
     maybe have come up a few times since, but never _exactly
    quantified_.  Now here it is in the real world.  The application
     had quietly evolved into something **infamous**:
141
142 ```text
143 A Tightly Coupled Monolith
144 ```
145 >  It's a simple complex system.  Because it's simple, it's
    prone to cascades, and because it's complex, you can't predict
    what's going to fail. Or how. -- _"The Expanse"_
146
147 * A change almost anywhere could trigger side-effects somewhere
    else.
```

```
148 * Features that should have touched one module required edits
    across five.
149 * Bug fixes became archaeology.
150 * And the surprising part?
151 * These structural problems weren't mysterious.
152 * They were **measurable**.
153 ___
154 #%% md
155 ##    The Two Numbers That Explain Most Architecture
156
157 Nearly every structural coupling metric derives from two simple
    counts.
158
159 ```text
160         Who depends on me?
161                 ↑
162                Ca
163                 │
164                 ● GIVEN MODULE/PACKAGE
165                 │
166                Ce
```

```
167                 ↓
168         Upon whom do I depend?
169 ```
170
171 ### Afferent Coupling (Ca)
172
173 ```text
174 Ca = number of external modules that depend on this one
175 ```
176
177 Afferent coupling measures **responsibility**.
178
179 If many modules depend on you, your stability matters.
180
181 Break this module, and others break too.
182
183 Modules with high Ca become **structural anchors**.
184
185 ### Efferent Coupling (Ce)
186
187 ```text
```

```
188 Ce = number of external modules this module depends on
189 ```
190
191 Efferent coupling measures **vulnerability**.
192
193 The more dependencies you have, the more ways your code can
    break.
194
195 Every dependency introduces:
196
197 - version risk
198 - semantic assumptions
199 - upgrade friction
200
201 Dependencies are powerful.
202
203 But they are never free.
204
205 ###    A Simple Analogy
206
207 These metrics behave like a financial balance sheet.
```

```
208
209 | Metric                          | Analogy                                |
210 |---------------------------------|----------------------------------------|
211 | `Ca` *(Afferent Coupling)* | Creditors (who depends on you) |
212 | `Ce` *(Efferent Coupling)* | Debts (who you depend on)      |
213
214 * Modules with many creditors must be **_stable_**.
215 * Modules with many debts are inherently **_fragile_**.
216 ---
217 #%% md
218 ##    The Instability Index (I)
219 <img src="../assets/images/metrics-instability-fig1-overview.png
    ">
220 From Ca and Ce we derive a powerful ratio.
221
222 ```text
223 I = Ce / (Ce + Ca)
224 ```
225
226 Instability ranges from **0 to 1**.
227
```

```
228 | I Range | Stability | Meaning | Change Strategy |
229 |---|---|---|---|
230 | 0.0 ≤ I < 0.25 | Stable | Many dependents, few dependencies |
    Change with care |
231 | 0.25 ≤ I < 0.50 | Balanced | Healthy structural position |
    Normal dev pace |
232 | 0.50 ≤ I < 0.75 | Borderline | Dependency heavy | Monitor
    closely |
233 | 0.75 ≤ I ≤ 1.0 | Unstable | Few dependents, many dependencies
     | Refactor freely |
234
235 <img src="../assets/images/metrics-instability-fig2-curves.png">
236
237 Modules with **low instability** are depended on by many others.
238
239 Modules with **high instability** depend on many others but are
    depended upon by few.
240
241 This leads to one of the most important architectural principles
    .
242
```

```
### Stable Dependencies Principle

Dependencies should flow **toward stability**.

```text
unstable modules  →  stable modules
```

When stable modules depend on unstable ones, architectural fragility appears quickly.
___
#%%

==
MAIN SEQUENCE 2D PLOT
==

# Example modules to plot (modify this!)
EXAMPLE_MODULES = [
    {'name': 'Database Schema', 'I': 0.1, 'A': 0.05, 'color': 'red'},
```

```
    {'name': 'Unused Interfaces', 'I': 0.7, 'A': 0.9, 'color': 'blue'},
    {'name': 'Domain Logic', 'I': 0.3, 'A': 0.7, 'color': 'green'},
    {'name': 'Service Layer', 'I': 0.7, 'A': 0.3, 'color': 'green'},
    {'name': 'Perfect Balance', 'I': 0.5, 'A': 0.5, 'color': 'darkgreen'},
    {'name': 'API Gateway', 'I': 0.8, 'A': 0.4, 'color': 'orange'},
]

# Calculate D for each
for m in EXAMPLE_MODULES:
    m['D'] = abs(m['A'] + m['I'] - 1)

fig_main_seq = go.Figure()

# Zone of Pain (lower-left triangle)
fig_main_seq.add_trace(go.Scatter(
    x=[0, 0, 0.5, 0], y=[0, 0.5, 0, 0],
```

```
278     fill='toself',
279     fillcolor=CONFIG['zone_pain'],
280     line=dict(width=0),
281     name='Zone of Pain',
282     hoverinfo='name',
283 ))
284 
285 # Zone of Uselessness (upper-right triangle)
286 fig_main_seq.add_trace(go.Scatter(
287     x=[1, 1, 0.5, 1], y=[1, 0.5, 1, 1],
288     fill='toself',
289     fillcolor=CONFIG['zone_useless'],
290     line=dict(width=0),
291     name='Zone of Uselessness',
292     hoverinfo='name',
293 ))
294 
295 # Main Sequence line
296 fig_main_seq.add_trace(go.Scatter(
297     x=[0, 1], y=[1, 0],
298     mode='lines',
```

```
299     line=dict(color=CONFIG['main_seq'], width=4),
300     name='Main Sequence (A+I=1)',
301 ))
302 
303 # Distance contour lines
304 for d in [0.2, 0.4, 0.6, 0.8]:
305     # Above main sequence
306     x_above = [max(0, d), min(1, 1)]
307     y_above = [min(1, 1-x_above[0]+d), max(0, 1-x_above[1]+d)]
308     fig_main_seq.add_trace(go.Scatter(
309         x=x_above, y=y_above,
310         mode='lines',
311         line=dict(color='gray', width=1, dash='dash'),
312         showlegend=False,
313         hoverinfo='skip',
314     ))
315     # Below main sequence
316     x_below = [0, min(1, 1-d)]
317     y_below = [max(0, 1-d), 0]
318     fig_main_seq.add_trace(go.Scatter(
319         x=x_below, y=y_below,
```

```
320         mode='lines',
321         line=dict(color='gray', width=1, dash='dash'),
322         showlegend=False,
323         hoverinfo='skip',
324     ))
325 
326 # Plot example modules
327 for m in EXAMPLE_MODULES:
328     fig_main_seq.add_trace(go.Scatter(
329         x=[m['I']], y=[m['A']],
330         mode='markers+text',
331         marker=dict(size=15, color=m['color'], line=dict(width=2
    , color='white')),
332         text=[m['name']],
333         textposition='top right',
334         textfont=dict(size=11),
335         name=m['name'],
336         hovertemplate=(
337             f"<b>{m['name']}</b><br>"
338             f"I: {m['I']:.2f}<br>"
339             f"A: {m['A']:.2f}<br>"
```

```
340                 f"D: {m['D']:.2f}<extra></extra>"
341             ),
342         ))
343 
344 # Zone labels
345 fig_main_seq.add_annotation(
346     x=0.15, y=0.15, text='<b>ZONE OF PAIN</b><br>(Stable + Concrete)',
347     showarrow=False, font=dict(size=12, color='#8B0000'),
348 )
349 fig_main_seq.add_annotation(
350     x=0.85, y=0.85, text='<b>ZONE OF USELESSNESS</b><br>(Unstable + Abstract)',
351     showarrow=False, font=dict(size=12, color='#00008B'),
352 )
353 
354 fig_main_seq.update_layout(
355     title=dict(
356         text='<b>Main Sequence: A + I = 1</b>',
357         font=dict(size=18),
358         x=0.5,
```

```
    ),
    xaxis=dict(
        title='I (Instability) →',
        range=[-0.05, 1.05],
        dtick=0.1,
        gridcolor='rgba(0,0,0,0.1)',
    ),
    yaxis=dict(
        title='A (Abstractness) →',
        range=[-0.05, 1.05],
        dtick=0.1,
        scaleanchor='x',
        gridcolor='rgba(0,0,0,0.1)',
    ),
    paper_bgcolor=CONFIG['paper_bg'],
    plot_bgcolor=CONFIG['paper_bg'],
    height=700,
    showlegend=True,
    legend=dict(x=1.02, y=1),
)
fig_main_seq.show()
```

```
380
381
382 # Uncomment to save:
383 # fig_main_seq.write_html('main-sequence-interactive.html',
    include_plotlyjs='cdn')
384 #%% md
385 ---
386 ##    Abstractness (A)
387
388 This metric differentiates types as **concrete** or **abstract
    ** (`interface`/`protocol`/`port`).
389
390
391 ```text
392 A = Na / Nc
393 ```
394
395 ### Where
396
397 * `Na =` number of abstract types
398 * `Nc =` total number of types
```

````
399
400
401 ### Interpretation
402 * `A = 1` → completely abstract
403 * `A = 0` → completely concrete
404
405 ### Conclusion
406 * Abstraction provides flexibility
407 * Concrete code provides behavior
408 * Good architecture balances both
409 ---
410 #%% md
411 ##    The Main Sequence
412
413 When plotting **Abstractness (A)** against **Instability (I)**,
    something interesting appears.
414
415 Healthy modules tend to cluster along a line defined by:
416
417 ```text
418 A + I = 1
````

```
419 ```
420
421 This line is called the **Main Sequence**.
422
423 Modules drifting far from this line often indicate structural
    issues.
424 ---
425
426 #%%
427 module_names = ['Core Domain', 'API Contracts', 'Shared Utils',
    'Infra Adapter', 'Feature X', 'Plugin Y', 'Data Model', 'Proto
    Layer']
428 I = np.array([0.10, 0.20, 0.75, 0.85, 0.55, 0.90, 0.15, 0.35])
429 A = np.array([0.85, 0.70, 0.10, 0.05, 0.40, 0.05, 0.20, 0.55])
430
431 fig, ax = plt.subplots(figsize=(7, 6))
432 x = np.linspace(0, 1, 200)
433 ax.plot(x, 1 - x, linewidth=2, label='Main Sequence: A + I = 1')
434 ax.scatter(I, A)
435
436 for name, x0, y0 in zip(module_names, I, A):
```

```
437     ax.annotate(name, (x0, y0), xytext=(5, 5), textcoords='
    offset points', fontsize=9)
438 
439 ax.set_title('Abstractness vs Instability')
440 ax.set_xlabel('Instability (I)')
441 ax.set_ylabel('Abstractness (A)')
442 ax.set_xlim(0, 1)
443 ax.set_ylim(0, 1)
444 ax.legend()
445 plt.tight_layout()
446 plt.show()
447 
448 #%% md
449 ##    Architectural Danger Zones
450 
451 ### Zone of Pain
452 
453 ```text
454 low A
455 low I
456 ```
```

````
457
458 Meaning:
459
460 ```text
461 concrete AND stable
462 ```
463
464 These modules are depended on by many other modules but contain
    little abstraction.
465
466 Examples often include:
467
468 - database schemas
469 - configuration systems
470 - foundational libraries
471
472 Changing them causes cascading impact.
473
474 Hence the name.
475
476 ### Zone of Uselessness
````

````
477
478 ```text
479 high A
480 high I
481 ```
482
483 Meaning:
484
485 ```text
486 abstract AND unstable
487 ```
488
489 These modules contain abstractions nobody uses.
490
491 Example:
492
493 ```text
494 12 interfaces
495 1 implementation
496 0 dependents
497 ```
````

````
498
499 Beautiful architecture.
500
501 No real purpose.
502
503 ##    Distance From the Main Sequence
504
505 We quantify architectural imbalance using:
506
507 ```text
508 D = |A + I - 1|
509 ```
510
511 Values near **0** lie close to the Main Sequence.
512
513 Higher values indicate potential architectural issues.
514
515 Large values may indicate:
516
517 - rigid dependency hubs
518 - unused abstractions
````

```
519 - architectural layering problems
520
521 #%%
522 distance = np.abs(A + I - 1)
523
524 fig, ax = plt.subplots(figsize=(10, 4.8))
525 ax.bar(module_names, distance)
526 ax.set_title('Distance From Main Sequence')
527 ax.set_ylabel('D = |A + I - 1|')
528 ax.axhline(0.3, linestyle='--', linewidth=1, label='Review
    threshold')
529 ax.legend()
530 plt.xticks(rotation=30, ha='right')
531 plt.tight_layout()
532 plt.show()
533
534 #%% md
535 ##   ⍰ Where These Metrics Apply
536
537 These metrics apply to many software systems:
538
```

```
539 - large monoliths
540 - modular applications
541 - plugin architectures
542 - libraries
543 - microservices
544
545 Microservice systems especially benefit from coupling analysis
    because dependencies often hide behind **network calls rather
    than imports**.
546
547 A service with high efferent coupling may rely on many
    downstream services.
548
549 Each dependency increases operational risk.
550
551 Understanding coupling helps prevent systems from drifting
    toward the dreaded:
552
553 ```text
554 distributed monolith
555 ```
```

```
556
557 ##    The Takeaway
558
559 Architecture is often treated as an art.
560
561 But beneath the diagrams lies something more mechanical.
562
563 Software systems obey structural forces.
564
565 Coupling is one of them.
566
567 And like gravity…
568
569 you can ignore it.
570
571 But you cannot escape it.
572
573 ##    References
574
575 - Martin, R. C. (1994). *OO Design Quality Metrics: An Analysis
    of Dependencies.*
```

```
576 - Martin, R. C. (2017). *Clean Architecture.*
577 - Lakos, J. (1996). *Large-Scale C++ Software Design.*
578 - Ford, N., Parsons, R., & Kua, P. (2017). *Building
    Evolutionary Architectures.*
579 #%%
580
```