Programação Orientada a Gambiarra

Como garantir que o inferno
seja uma amostra grátis do seu trabalho!

Josenaldo Matos

Sumário

Capítulo 1

Agradecimentos

Há muitas pessoas a quem eu devo agradecer. Se eu fosse nomear todas aqui, isso seria uma listagem maior que uma nota fiscal de quem comprou uma bala no supermercado. Então, vou agradecer apenas algumas pessoas muito queridas.

A ordem de apresentação não implica em uma ordem de importância em minha vida. Até porque nenhum de vocês é mais importante que minhas gatas Bugada e Lesada.

Primeiro, vou agradecer à minha família. Vocês fizeram um grande trabalho. Exceto, claro, quando levaram 15 anos pra perceber que a criança com a cara colada na TV precisava usar um óculos mais potente que o Telescópio Espacial James Web. E isso porque a família me deu **2 TIAS ENFERMEIRAS MAIS MÍOPES DO QUE EU** ! Já sabemos quem cabulou as aulas de genética pra ir pro boteco. Mesmo assim, eu amo vocês!

Eu posso não acreditar em Deus, mas acredito em anjas, pq eu já conheci três: Luciana Ribeiro Matos, minha irmã de faculdade; Elma dos Passos Rabello, minha primeira sogra e mãe de rim; e Maria Teresa Lima (em memória), minha segunda sogra e saudosa companheira de papos malucos. Obrigado por me dedicarem tanto amor, apesar de minhas falhas e imperfeições. Vocês me mostraram que esse mundo ainda vale a pena. Levarei vocês pra sempre no meu coração. No rim não, porque o rim eu perco.

Um agradecimento especial à minha digníssima esposa, Cassiana, que trouxe a luz do amor de volta à minha vida. E outro agradecimento aos nossos filhos Joseana, Cassinaldo, Jossinalna, Cijomar e Prosongolôndia, que não nasceram ainda, por não tentarem me matar devido aos nomes que vou por neles. Eu acho.

Nenhum obrigado aos bolsonaristas e antivacinas. A esses, eu não tenho nada o que agradecer. A esses, eu só desejo que peguem fungo de pneu de caminhão no símbolo químico do cobre.

A todo mundo que acha que eu deveria citar aqui, mas não citei, eu usarei as palavras de Bilbo Bolsista: Eu não conheço metade de vocês a metade do que gostaria; e gosto de menos da metade de vocês a metade do que vocês merecem.

PARTE I

Introdução

Capítulo 2

Introdução

Saudações, POGramadores!

Sejamos sinceros... Você chegou a esse livro porque está cansado. Você deveria estar trabalhando, estudando, desenvolvendo o software que vai deixar seu chefe mais rico... Mas você está de saco cheio e resolveu gastar seu tempo lendo sobre Gambiarras.

Bem, pode comemorar. Você está no lugar certo. Já pode tocar Aleluia no celular.

Aqui, você não vai aprender a resolver suas gambiarras. Pode tirar essa ilusão desse seu coraçãozinho maltratado. Aqui, você vai aprender a abraçar o GLS (Gambi Life Style) de vez.

Durante a leitura deste tomo sagrado, sua mente passará pelo mais avançado curso de PNL (POGramação Neuro Linguiça), que capacitará você a identificar, utilizar e idealizar as POGs que tornarão o inferno uma amostra grátis do seu trabalho.

O livro é dividido em 3 partes.

- Conceitos
- Técnicas
- Gambi Design Patterns

Na primeira parte desse livro, **"Conceitos"** navegaremos pelos principais conceitos ligados à arte de criar Gambiarras.

O que é um POGramador? O que é uma Gambiarra? Quais o requisitos que um ambiente deve atender para que a Gambiarra floresça?

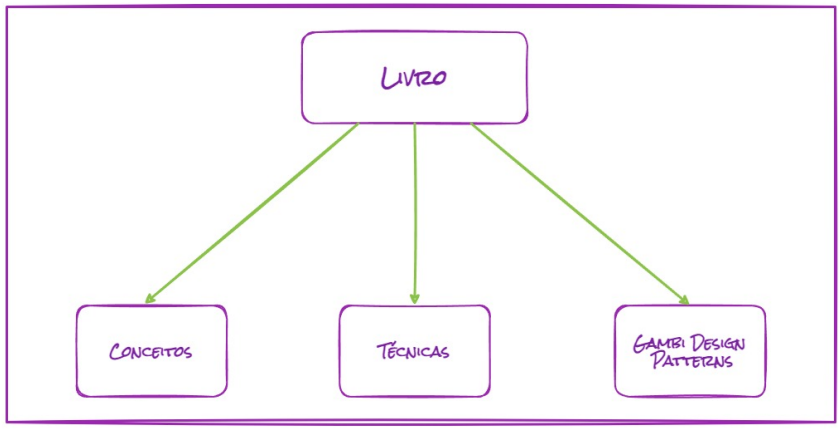

Figura 2.1: *DIagrama sofisticado demonstranto a estrutura do livro*

Quais princípios um POGramador deve ter marcado no âmago de seu ser?

Na segunda parte, **Técnicas**, conheceremos as (rufem os tambores!) técnicas que constam do arsenal de um POGramador.

Por fim, veremos a aplicação dessas técnicas na terceira parte, **Gambi Design Patterns**, que é um catálogo dos principais padrões de projeto da desgracença.

Ao final deste livro, você, POGramador, terá uma caixa de ferramentas tão vasta na capacidade de causar tragédias que saberá que o termo “Caixa de Pandora” só existe porque você não nasceu antes. Se tivesse nascido, seria “Caixa de POGramador”.

Boa leitura e que Lady Murphy te acompanhe.

Capítulo 3

O que é POG?

Gambiarra.

Ao assumir o sacerdócio da área da POGramação, a palavra Gambiarra é cravada em nossos cérebros e passa a fazer parte do nosso vocabulário.

Muito se discute sobre os benefícios e malefícios da Gambiarra. A maioria faz piada. E muitos até tentam resistir. Inutilmente, claro. A Gambiarra torna-se uma parte importante de nossas vidas, quer você queira ou não.

Mas, afinal, o que é uma Gambiarra?

Dentre os civis (aqueles que não comungam do conhecimento sagrado da POGramação), a palavra Gambiarra quase sempre tem uma conotação ligada a adaptações ineficientes ou soluções improvisadas pra problemas que exigem técnicas mais apuradas.

Uma acepção menos pejorativa e mais objetiva é o uso desta palavra pra designar o conjunto de lâmpadas em série, usado para iluminar uma área onde ocorrerá um evento, como uma peça de teatro, uma festa junina ou um bacanal de pessoas sem um pingo de vergonha.

E dessa forma, é que você, jovem POGramador, deve ver a Gambiarra: como a luz que ilumina o espetáculo que é o seu código!

> "Por definição, a Gambiarra é aquilo que é de difícil concepção, de inesperada execução para tornar fácil o uso de algo que sequer deveria existir." (Desciclopedia 2016)

Ou seja, a Gambiarra é a solução técnica planejadamente improvisada e resultante de uma inspiração momentânea, com o intuito de resolver um problema complexo, onde o uso de técnicas tradicionais incorrem em alto custo energético para o resolvedor.

A duração da gambiarra é limitada, devendo essa ser substituída, assim que possível, por uma solução técnica convencional. Portanto, uma boa gambiarra tem, como tempo de permanência, o valor mínimo *Tg* (Tempo da Gambiarra), sendo que *Tg* tende ao infinito.

Por ter baixo custo presente, seu custo futuro tende a ser ignorado pelo gambiarrizador, já que esse custo certamente será assumido por terceiros. Portanto, a Gambiarra se mostra extremamente vantajosa, o que justifica a sua utilização.

3.1 Sinônimos de Gambiarra

O termo Gambiarra possui vários sinônimos, que são usados nas mais diversas áreas. Em sua maioria, os sinônimos são eufemismos, utilizados como forma de esconder, dos civis, que uma Gambiarra está sendo usada, já que a mente primitiva do ser humano comum é incapaz de perceber o brilhantismo dessa solução.

Dentre estes sinônimos, temos:

- **ATI** - Aparato Técnico Improvisado
- **ATND** - Artifício Técnico Não Documentado
- **CPMF** - Conserto Provisório Mas Funciona
- **DAT** - Dispositivo Alternativo Temporário
- **ERR** - Engenharia de Reparos Rápidos
- **MASC** - Medida Adaptativa à Situações Críticas
- **MTEDM** - Manutenção Técnica com Elementos Disponíveis no Momento

- **MUTRETA** - Método Único de Tratamento e Resolução de Erros Totalmente Adaptável
- **REZA** - Reestruturação Emergencial Zuada Auxiliar
- **RTA** - Recurso Técnico Avançado
- **RTA** - Recurso Tecnológico Alternativo
- **RTDM** - Recurso Técnico Disponível no Momento
- **RTE** - Recurso Técnico de Emergência
- **RTI** - Recurso Técnico Inteligente
- **STCT** - Solução Técnica de Cunho Temporário

No contexto da POGramação, temos também os seguintes sinônimos:

- **ADP** - Adaptação De Programador
- **CACA** - Código Avançado Completo e Adaptável
- **CAGADA** - Código Alternativo Gerador de Algoritmos Duramente Aplicáveis
- **DADA** - Deixa Assim, Depois Arrumo
- **IST** - Improvisation Solution Tabajara
- **ITAC** - Implementação Técnica Abstratamente Controversa
- **RAP** - Recurso Avançado de Programação
- **TAPA** - Técnica Alternativa de Programação Avançada

Podemos notar como o uso de siglas é comum na denominação da Gambiarra. Portanto, a lógica é clara: se algo, na computação, é nomeado com uma sigla, provavelmente é uma Gambiarra.

O exemplo mais notório dessa regra é o acrônimo recursivo GNU, que significa "GNU is Not Unix", e denota uma Gambiarra que se gambiarriza em si mesma.

Mas existe um termo que merece uma explicação adicional, devido às suas peculiaridades: Marreta[1].

[1] O motivo pelo qual o termo "marreta" é tão importante é bastante óbvio, mesmo para o leitor mais desatento: é porquê eu gosto e eu quero. Se você não percebeu isso, sugiro que procure um profissional especialista(astrólogo, vidente, adivinho ou áreas correlatas). A propósito: Porque as pessoas dizem "profissional especialista"? Existe algum especialista que não seja profissional? Um especialista nato? "Conheça Enzo Rodrigo, especialista em computação quântica aos 4 anos de idade, entre uma colherada de mingau e outra, resolveu o problema da conjunção telepática de gatos robóticos."

3.1.1 Marreta

O termo “Marreta” é usado por quem associa o poder gambiarrizador à ferramenta Marreta, que é usada no lugar de um martelo. O POGramador também associa o poder gambiarrizante ao deus Thor, que resolvia tudo na base do martelo.

A origem do termo está no ditado “Pra quem só sabe usar martelo, todo problema é prego”.

Obviamente que podemos discutir o porquê de não se usar o termo “Martelo”, mas o uso do termo correto associado ao ditado é uma incoerência gambiarrística! A própria utilização da marreta, no lugar do martelo, demonstra uma gambiarra verbal, o que fecha o ciclo lógico da gambiarra numa metagambiarra.

3.1.2 Gambiarra em outras línguas

A gambiarra é um conceito universal. Não importa o país que você visite, sempre existe uma criatura abençoada alterando alguma coisa, de forma improvisada, para que um propósito não planejado seja atingido ou algum reparo desejado, mas impossível, seja tornado possível.

Sabendo disso, POGramadores bem informados compreendem que não precisam apenas ter competência, eles precisam DEMONSTRAR competência. E a forma mais simples de demonstrar competência é na comunicação verbal, principalmente com cliente e civis.

O POGramador deve se utilizar de todo artifício verbal em seu arsenal para mostrar que é dotado de capacidades técnicas que o marcam como um profissional de primeira linha. Dentre essas habilidades, está a capacidade de dominar o inglês[2].

Por essa razão, é muito comum o uso do vocábulo *workaround*.

Sempre que você ver um profissional usando o termo *workaround*, saiba que esse profissional é o POGramador de alto nível.

Outros sinônimos, em inglês, que são poucos usados no Brasil e, portanto, podem aumentar a pontuação do POGramador, são as ex-

[2] O idioma, não um homem proveniente da Inglaterra.

pressões *kludge*, *jugaad*, *jury rig* e *"quick and dirt"*.

Outra expressão com a qual devemos ficar alerta é "Do It Yourself" (DIY). Sempre que essa expressão surge, quase sempre em um livro de feitiçarias malégnas[3] disfarçado de tutorial, pode ter certeza de que existe uma criatura condenada sumonando uma gambiarra malégna, por conta própria.

Nas mãos de pessoas despreparadas, como civis e programadores, isso quase sempre acaba num arremedo de projeto, como aquela sua tia que tentou fazer um jarro chinês e acabou com uma réplica do Útero de Satanás no meio da sala.

E por falar em POGramação...

3.2 Programação Orientada a Gambiarra

Dentre todas as formas de encarnação que a Gambiarra possui, este livro tratará de sua forma digital mais profícua[4]: **A POG (Programação Orientada a Gambiarras)**.

> A Programação Orientada a Gambiarras (POG ou WOP – Workaround-oriented programming) é um paradigma de programação de sistemas de software que integra-se perfeitamente a qualquer grande Paradigma de Programação atual. (Desciclopedia 2016)

Este paradigma permite que utilizemos de Gambiarras para resolver problemas computacionais, não computacionais, espirituais, econômicos e até mesmo sexuais, de forma a garantir o sucesso do projeto.

A aplicação da POG tende a criar mais problemas do que resolve, criando, dessa forma, um círculo virtuoso que garante empregos a milhões de POGramadores pelo mundo. Cada problema criado significa mais trabalho e, portanto, mais empregos!

[3] Se Shiryu disse que é malégna, então é malégna.

[4] O que capirotos é "profícua"? Não sei. Mas parece termo de autor chique, então, como bom POGramador, vou usar sem saber o que é, aplicando o Gambi Pattern RCP (Reuse by Copy and Paste).

Para compreender a POG, é necessário compreender quais os requisitos para a formação da POG, quais os princípios que guiam o POGramador e quais as técnicas que esse POGramador usará. Veremos esses tópicos nos próximos capítulos.

3.3 Notas

Capítulo 4

História da POG

Quando procuramos definir a primeira POG da história, a maior dificuldade está no fato de que o bom POGramador não deixa rastros de seus méritos, pois POGramador não usa comentários(a não ser que sejam inúteis).

Esse ambiente de incertezas é terreno fértil para o surgimento de boatos, lendas e mitos, que acabam por transformar a história da POG em um desafio a qualquer historiador. E, como diz o ditado, "quem não tem história, inventa".[1]

> Qualquer afirmação suficientemente convicta é indistinguível da verdade. (Minha Cabeça 2020)

Uma dessas lendas diz que a primeira POG foi criada pelo Papa Gregório XIII[2].

[1] Será que a ficcção é a gambiarra do historiador? Fica o questionamento.

[2] Em minha opinião, o próprio sistema de numeração romano é uma grande POG. "Julius, precisamos de símbolos para os números", disse César. "Que nada, César. Usa letra mesmo, que vai dar menos trabalho. Lá na frente, alguém troca".

4.1 O ser humano é uma máquina de reconhecer padrões

Pra entender como surgiu a provável primeira POG, precisamos voltar no tempo e entender o porque o ser humanos inventou de dar um nome a cada dia.

Pense em nossos antepassados. Não na sua avó, ou no avô dela. Vamos voltar muito antes disso. Vamos voltar ao tempo em que éramos apenas macacos pelados que acabaram de descer das árvores.

Nesse tempo, o ser humano não tinha calendário. Não tinha relógio. Não tinha nada que pudesse dizer "amanhã é segunda-feira".

Nossas necessidades eram bem mais simples: comer, dormir, fugir de predadores e procriar. E nós nos tornamos muito bons nisso. Mas como?

Seleção Natural. Vamos chamá-la carinhosamente de Tia Selena.

Tia Selena não escohe os mais fortes, nem os mais inteligentes. Muito menos ainda os mais bonitos. Ela escolhe os que se adaptam melhor ao ambiente. Os que são capazes de conseguir recursos necessários para a própria sobrevivência e para sua prole.

Mas como saber o que é comida e o que é veneno? Como saber o que é predador e o que é amigo? Como saber o que é o sexo oposto e o que é uma ovelha chamada Beeelinha?

Quem era capaz de encontrar as melhores frutas, ou de enxergar aquele coelho carnudo escondido no meio do mato, comia. Quem achava água, bebia. Quem era capaz de encontrar uma boa caverna pra se esconder, dormia pra ver o dia seguinte. E quem se tocava de que aquele coelho laranja e preto, da altura de um boi, e com garras do tamanho de uma cara humana, não era um coelho, mas sim um tigre, sobrevivia.

Acontece que nosso cérebro é uma máquina de reconhecer padrões. Ele é capaz de identificar padrões em qualquer coisa que ele pode ver, ouvir, cheirar, tocar, degustar ou imaginar.

Geração após geração, os mais capacitados em reconhecimento de padrões se mostravam mais aptos a sobreviver. E quem sobrevive, se

reproduz e passa pra frente seus genes.

Dessa forma, Tia Selena foi aperfeiçoando nossa capacidade de reconhecer padrões.

E essa máquina de identificar padrões é tão boa nisso que ela chega até mesmo a identificar padrões em coisas que não existem fisicamente. É o que acontece quando você vê um rosto na nuvem, um coelho na lua ou interesse sexual por parte de uma mulher que só foi simpática com você.

4.2 Não basta reconhecer, tem que espalhar

Mas, além de reconhecer padrões, precisávamos também de um jeito de ensinar esses padrões aos nossos companheiros humanos. Se eu aprendo que um tigre é um predador perigoso, eu preciso ensinar isso aos meus companheiros.

Eu não chamo o Josiscleisson e solto ele na frente do tigre, esperando que ele sobreviva ao ataque do tigre e aprenda por conta própria. Eu não preciso empurrar Josiscleisson do Barranco da Morte Certa pra ele entender que se cair nesse barranco, vai morrer.

É muito mais simplesChamar o Josiscleisson e dizer “Olha, aquele coelho laranja gigante tem garras do tamanho de nossa cara! E, ao invés de planta, ele come gente! O nome dele é Desmembrador! Fica longe dele!”.

O que nós fazemos é nos **COMUNICAR**. Nós explicamos, aos outros humanos, como as coisas funcionam. E, ao nos ouvir, eles aprenderm com a nossa experiência, evitam nossos erros e ganham ao repetir nossos acertos. Dessa forma, a comunicação se tornou um dos pilares da nossa sobrevivência.

Essa capacidade de nos comunicar nos levou a desenvolver uma rebuscada linguagem. E, como parte dessa linguagem, nós desenvolvemos também a capacidade de contar.

4.3 Não basta saber contar ovelhas

Uma vez que o ser humano começou a viver em grupos maiores, houve a necessidade de mais alimento. E, durante essa busca por mais alimento, nossa capacidade de subverter padrões nos levou a uma gambiarra maravilhosa: a cerveja!

No tópico anterior, falávamos de um ser humano moleque, o ser humano livre, cuja vida se limitava a nomadear por aí, catando o que achava pela frente, se escondendo onde podia e vivendo do que a terra dá.

Esse ser humano comia grãos, como a cevada. Inicialmente, ele comia a cevada como ela é. Mas, com o tempo, ele começou a perceber que, se ele deixasse a cevada de molho em água, ela ficava mais macia.

O gosto deveria ser uma droga, então não levou muito tempo pra algum macaco pelado com um pouco mais de cérebro perceber que se moesse os grãos, a mistura com a água ficaia mais fácil de consumir.

Com o tempo, o homem foi adicionando coisas a essa mistura. E, em algum momento, não se sabe se intencionalmente ou não, veio a grande sacada: assar essa mistura resultava num produto muito mais gostoso e duradouro: o pão.

O pão é um dos principais alimentos da humanidade há milênios. as primeiras evidências de pão datam de 30 mil anos atrás!

E, pra ter mais pão, ao invés de sair desembestado pelo mundo, procurando mato, o macaco pelado percebeu que poderia ter muito mais grãos se plantasse os graõs novamente. Assim nasceu a agricultura.

Além do pão, o homem também gostava de carne. Muita carne. E sair por aí caçando os bichos já não era tão eficiente assim. Em alguns casos, nós exterminamos todos os bichos de uma região. E a falta de carne significa que passaríamos fome.

Pra resolver esse problema, nós descobrimos que não precisávamos comer todos os bichos. Observamos que os bichos também se preproduziam, de tempos em tempos. E, pra ter mais carne, bastava

a gente criar mais bichos.

Mas, como o ser humano é um ser curioso, ele começou a experimentar outras formas e preparar o pão. E, em um belo dia, talvez de uma mistura de pão estragada, ou de trigo apodrecido, o macaco pelado descobriu que, se bebesse essa mistura, ele ganhava super poderes. O homem descobriu o álcool.

Dessa forma, o que era pra ser um erro virou uma feature e o álcool passou a fazer parte da vida humana.

Nesse processo de descobrir o pão, a cerveja e o churrasco, o ser humano perdeu o ímpeto de sair livre pelo mundo. Ao ser domesticado pelo trigo e pelo gado, o homem criou um curral pra si mesmo e chamou isso de "cidade".

Assim, o ser humano se fixou e passou a viver no mesmo local, onde ele poderia plantar e colher, criar e matar, sem precisar se deslocar. E, talvez pelotempo extra que ganhou ao se tornar sedentário, talvez pela necessidade de controlar seus rebanhos, o homem começou a contar. E não parou mais.

4.4 Precisamos contar o tempo

O homem agora domina a terra e o gado. Ele é senhor do ambiente. E, como todo ser imundiçado que é, ele nunca fica satisfeito e quer mais. Ele quer mais terra, mais gado, mais comida, mais bebida, mais mulheres, mais filhos, mais poder.

Acontece que a natureza não é um buffet de recursos grátis, que basta você chegar e pegar. A natureza parece mais com uma liquidação de loja de departamento, daquelas onde até o anticristo chora e pede perdão, onde você perde sua Air Friyer pra uma família, de 18 pessoas enquanto é espancado com galinhas gritadeiras de borracha.

Na dureza da vida, o macaco pelado percebeu que nem sempre ele precisa plantar e criar. Às vezes, ele pode simplesmente tomar o que é do outro. Pra que plantar e colher, se eu posso deixar outro ter esse trabalho e, depois, tomar dele?

Dessa forma, o homem aprendeu a guerrear. E como o homem

guerreou.

Agora, o macaco pelado precisa saber quando chove. Quando deve plantar. Quando deve colher. Quando deve abater seu rebanho. Quando deve fazer um sacrifício ao seu deus. Quando deve sair para a guerra. Quando deve voltar da guerra. Quando seu filho deveria ter nascido. Quando deve tirar satisfação com Juvenal, por ele ter visitado sua esposa na guerra e seu filho ter nascido com a cara do Juvenal.

O ser humano que não sabe contar o tempo é um ser humano perdido.

Mas não adianta o macaco pelado contar o tempo em ciclos lunares, se ele não sabe quando é a próxima lua cheia. Não adianta contar o tempo em ciclos solares, se ele não sabe quando é o próximo solstício. Não adianta contar o tempo em ciclos de chuva, se ele não sabe quando é a próxima estação seca.

Então, junto com essa nossa necessidade patológica de contar e estruturar as coisas, nós começamos também a registrar as coisas. E assim nasceu a escrita.

E foi assim Tia Selena ensinou um monte de macacos pelados a reconhecer padrões, a se comunicar, a plantar, a criar animais, a cozinhar, a se embebedar, a guerrar, a levar chifre, a contar e a escrever.

4.5 O calendário romano

A ideia parece simples: você pega um imundiçado sem Netflix e põe ele pra observar onde o caminho que o sol fez no céu, desde o momento em que nasceu até o momento em que se pôs. E manda ele registrar isso. Essa parte é muito importante!

Daí, ele acorda todo dia, antes do sol nascer, e passa o dia inteiro medindo o caminho do sol. Então, ele vai perceber (se não for uma anta) que o Sol nasce e se põe, a cada dia, num lugar diferente do dia anterior.

Isso ocorre até que, num dia, o sol nasce e se põe no mesmo lugar do primeiro dia. Pronto. Temos um ciclo. Agora, basta ele contar quantos dias se passaram. E, se ele repetir esse processo algumas

vezes, ele consegue dizer quanto tempo dura UM ANO.

Sim, fizeram isso. E mais de uma vez, na história da humanidade. E, dado o número de vezes em que os calendários mudaram, ou o processo é mais difícil do que parece, ou as pessoas encarregadas dse mentir se entediavam facilmente, largavam o projeto no meio e inventavam números.

Muitos povos tentaram esse processo. E um que se destacou bastante nisso foram os romanos.

O primeiro calendário romano era um calendário Lunar, de 10 meses. Segundo a lenda, foi implantado na criação de Roma, em 753 a.C.

Esse calendário tinha meses com 30 ou 31 dias, com um total de 304 dias. Os 61 dias restantes eram o inverno, e ninguém ligava pra contar o tempo no inverno.

Aqui nós já vemos um caso fantástico de POG, em que os 61 dias eram simplesmente COMENTADOS, num claro uso de Commented Code Implementation!

> Maledicite scribarum! Nemo curat id quod fit in hieme! Istam lineam commentarium pone. Nemo vocabit si sexaginta unus dies interiit.
>
> – Rômulo, fundador de Roma (753 a.C.)

Em 713 a.C. Numa Pompílio fez a primeira reforma no calendário romano, diminuindo o número de dias de alguns meses e aumentando o número de meses para 12.

Dessa forma, o ano agora tinha 355 dias. Como resolver os dias faltantes?

Com gambiarra, claro!

A cada 2 anos, um mês extra, de 22 ou 23 dias, era adicionado ao final de "Fevereiro". E a decisão de inserir esse mês cabia ao Pontífice Máximo[3]. Como era um ser humano a decidir, é óbvio que nem sempre isso acontecia. E, quando acontecia, nem sempre era feito

[3]Maximus Pontifex: Na Roma antiga, o Pontífice máximoera o sacerdote supremo do colégio dos sacerdotes, a mais alta dignidade na religião romana.

da mesma forma. O resultado era que, às vezes, o ano não era tão previsível assim.

Parece familiar?

4.6 O calendário Juliano

Em 46 a.C. Julio César, resolveu botar ordem nesse quengaral. Com a ajuda do sábio Sosígenes de Alexandria, Júlio Cézar, na época ocupando o cargo de Pontífice Máximo, organizou um novo calendário.

Esse novo calendário entrou em vigor no dia 1 de janeiro de 45 a.C. Dentre suas principais características, temos:

- Ano de 365 dias
- 12 meses (sem meses intercalares)
- Acréscimo de 1 dias, de 4 em 4 anos, para compensar a diferença de 4 horas, já que o ano trópico tem 365 dias e 4 horas
- O primeiro dia do ano passa a ser 1 de janeiro

Esse calendário durou bastante tempo. Dada sua longevidade, pode-se dizerque era um calendário bastante estável. Contudo, ele tinha alguns “pequenos” problemas:

- Não representava o tempo real que a terra leva pra girar em torno do Sol
- Como os anos bissextos ocorriam a cada 4 anos, a contagem do tempo ia, aos poucos, se desencontrando dos fenômenos naturais, como a mudança das estações, que ocorriam em datas fixas.
- Com o passar do tempo e o acúmulo dos erros, a data da páscoa ia se afastando gradualmente do Equinócio da Primavera.

Após alguns séculos, a diferença nessas datas já era de dias. E, como a páscoa era um feriado religioso, isso começou a causar problemas.

Como Júlio César foi um bom POGRamador, ele deixou esse pepino pra outro resolver lá na frente. Coube ao Papa Gregório XIII, em 1582, resolver esse problema.

4.7 O calendário Gregoriano

Após vários séculos, a diferença entre o calendário Juliano e o ano Solar foi se acumulando. Em 1582, o equinócio de primavera já ocorria 10 dias antes da Páscoa! E essa diferença tendia a se acumular ainda mais.,

Por consequência, teríamos na época, duas festividades, a comemoração do Equinócio de Primavera e a comemoração da Páscoa com 10 dias de diferença (nessa hora, os patrões já estão se coçando de alergia). E, no futuro, com a diferença almentando, logo teríamos a Páscoa sendo comemorada em pleno verão do hemisfério norte, com coelhas de bikini e padres ensandecidos explicando que a busca pelo ovos deveria ser um símbolo de vida e renacismento e não uma festa em homenagem a Sodoma e Gomorra!

Obviamente que essa situação era insustentável para a religião cristã e uma atitude precisava ser tomada.

Gregório XIII, então, resolveu fazer uma reforma no calendário. Ele convocou um time de especialistas, incluindo:

- Christopher Clavius, jesuíta alemão, sábio e matemático
- Ignazio Danti, dominicano, matemático, astrônomo e cartógrafo italiano
- Luigi Giglio médico, filósofo, astrónomo e cronologista italiano.

Esse time de estrelas trabalhou nesse problema por 5 anos, após os quais o Papa, em 24 de Fevereiro de 1582, publicou a bula papal *Inter Gravissimas*, com as mudanças no calendário.

A principal mudança é que o dia seguinte à quinta feira, 4 de outubro de 1582, não seria sexta feira, 5 de outubro, mas sim sexta feira, 15 de outubro. O papa simplesmente COMENTOU 10 dias!

Além disso, o algoritmo de definição do ano bissexto passou por uma pequena mudança. Agora, os anos bissextos seriam definidos da seguinte forma:

- Anos múltiplos de 4, exceto os múltiplos de 100, mas incluindo os múltiplos de 400

```
1 Inicio
```

```
    Declare ano Inteiro;
    Declare bissexto Booleano;
    Leia(ano);
    Se ( ano módulo 400 é 0 ) então
        bissexto=Verdade;
    Senão
         Se (ano módulo 4 é 0 E ano módulo 100 é
↪   diferente de 0) então
             bissexto=Verdade;
         Senão
             bissexto=Falso;
Fim
```

Com essas mudanças, o calendário Gregoriano tornou-se, com o pasar do tempo, o calendário mais usado no mundo. Entretanto, ele não é perfeito e, em 4909, o calendário estará adiantado em UM dia em relação ao calendário solar. Mas isso é problema pra outro POGRamador resolver lá na frente.

4.8 Chama o Ratinho

Muitos afirmam que o Papa Gregório XIII foi o criador do Ano Bissexto. Mas, como vimos, isso é um erro!

É óbvio que um POGramador experiente é capitalista com os méritos, socialista com os erros e autoritário com a culpa. Mas o Gregório nem sequer tentou assumir a autoria desse projeto!

A ideia de dias a mais para compensar o descompasso entre o calendário e o ano solar é usada em diversos calendários ao longo da história. Hoje, parece simples contar quanto tempo tem um ano, mas isso já foi um grande desafio!

O ano bissexto, especificamente, foi introduzido no Calendário Juliano. Portanto, se considerarmos o Ano Bissexto com a primeira POG, seria Júlio César o primeiro POGramador.

Devido a essa confusão, que atribui os mérito da criação do Ano Bissexto ao Papa Gregório XIII, é que ele é considerado o Padroeiro

dos POGramadores e, no dia 29 de Fevereiro, é comemorado o Dia Internacional da POG.

4.9 Notas

Capítulo 5

Requisitos da POG

Além de empregar POG como acrônimo para Programação Orientada a Gambiarra, temos também o termo “pog”, usado corriqueiramente como sinônimo de “uma gambiarra”, ou seja, uma simples unidade de gambiarra implementada por um POGramador. Assim, é comum que um POGramador diga “eu fiz uma pog” quando descreve o **artefardo**[1] resultante de seu trabalho.

No mundo do desenvolvimento de software, existe a noção de que uma pog é resultado do esforço laboral de um POGramador. Tal noção, apesar de parecer bastante lógica, é um engano tão ardiloso que é capaz de enganar até mesmo as mentes mais sagazes.

Um POGramador não é o criador da pog. Ele é apenas um conduíte para uma pog que deseja vir a este mundo. O trabalho do POGramador é apenas sumonar essa pog, tal qual faria para sumonar um demônio. Portanto, uma pog não é criada, ela é sumonada. E, para que este ritual seja bem sucedido, é preciso que certos Requisitos sejam cumpridos.

De que Requisitos estamos falando? Não, não estamos falando de

[1] Um artefardo é um artefato que cria, para a equipe, um fardo extra. Dessa forma, um artefardo é um ativo valioso para o POGramador, pois exige desse mais trabalho, o que ajuda a manter seu emprego.

sacrificar um virgem[2]. Estamos falando de condições que afetam as probabilidades do nascimento de uma pog.

Os Requisitos da POG podem ser classificados em diversas categorias, de acordo com o ponto de vista sob o qual olhamos esses Requisitos.

5.1 As dimensões dos Requisitos da POG

Durante milhares de anos, a humanidade encarou o mundo em 3 dimensões: largura, altura e profundidade. A ciência do século XX e a ficção científica acabaram por nos desvelar a possibilidade encararmos a realidade pelo prisma de mais dimensões. Agora, tempo é uma dimensão. Alguns modelos que explicam a realidade apontam a existência de até 11 dimensões!

Podemos, portanto, utilizar o conceito de dimensões como uma forma de classificar e melhor compreender cada um desses requisitos. E porque o conceito de dimensões? Porque fica muito mais estiloso, óbvio! Se a ciência e a realidade não concordam com minha noção de estilo, elas duas que lutem!

Vejamos, portanto, quais são os Requisitos da POG, de acordo com cada uma das dimensões.

5.2 Notas

[2]Até mesmo porque os valores mudaram e a falta de experiência sexual já não é um atributo tão valorizado. Que tipo de divindade tapada e ajamantada deseja o sacrifício de um estagiário sexual? Porque não exigir o sacrifício de um ser humano dotado de experiência? Porque não solicitar o sacrifício de um sênior da putaria, de um arquiteto da lascívia ou uma diretora da luxúria?

5.3 Dimensão Humana

Criar software é transformar o âmago do ser humano em impulsos digitais. E, como tal, o resultado não poderia ser outro: uma sucessão de erros e desastres que trabalham pra realizar uma tarefa.

Um bom POGrama é um amontoado de coisas escritas que tem a capacidade de fingir resolver um problema enquanto cria vários outros. O fator humano é, portanto, o principal influenciador da POG, o ingrediente com sabor mais forte nessa sopa de desgraça que leva à manifestação digital de uma pog.

Os Requisitos da POG classificados na Dimensão Humana são aqueles produzidos diretamente pela participação humana nesse processo. Não é apenas nossa presença danosa que permite que a POG floresça. É necessário que essa presença ocorra encarnada em algum dos seguintes estereótipos.

5.3.1 Equipe Apática

Quer ver a pog se espalhar como erva daninha num jardim bem nutrido? Entregue seu projeto a uma equipe apática.

Não importa qual desgracença desperte de sua caixa de pandora dos infernos, eles não se abalarão. Dia após dia, essa equipe mostrará que não se importa com absolutamente nada além de seus salários. E, por isso mesmo, estarão dispostos a usar qualquer recurso disponível que garanta o pagamento mensal.

Uma equipe apática não se importa com o passado e não liga para o futuro. A única coisa que eles querem é que alguém lhes diga o que fazer (desde que não dê muito trabalho) e que seu pagamento os aguarde, ao fim do mês. Nada mais importa. Assim, se uma pog for útil pra resolver o problema atual, eles a usarão sem um pingo de remorso.

Dessa forma, mesmo que um pequeno jardim de pogs se torne a nova Floresta Amazônica da Calamidade, uma Equipe Apática não vai se abalar para resolver nada disso.

5.3.2 Profissionais Superestimados

Junto com uma Equipe Apática, quase sempre aparece um Profissional Supervalorizado, aquele profissional que todo mundo acredita que ele sabe o que faz e que vai resolver todos os problemas. Evidentemente que todos os problemas caem no colo dele e ele acaba sobrecarregado.

Nesse cenário, o Profissional Supervalorizado acaba por cometer desde os erros mais simples até os erros mais catastróficos. E são erros tão épicos que as pessoas o olham com admiração e pensam "UAU, olha só o tipo de problema com o qual tem que lidar!", sem perceber que ele mesmo (e sua Equipe Apática) é que criaram esses problemas.

Um Profissional Supervalorizado acaba, portanto, sempre recorrendo à pogs para resolver aquilo que deveria resolver com resoluções resolvedoras de alta resolutividade, mas que ele não conhece. E que ninguém percebeu, ainda, que ele não conhece.

Esse profissional costuma ser um grande invocador de pogs da equipe, o que acaba por aumentar sua fama e o quanto ele é superestimado.

5.3.3 Arquiteto MacGyver

Numa equipe POG, ou mesmo em uma empresa usuária de POG, é muito comum a existência de uma figura mítica: o Arquiteto MacGyver.

Esse profissional ostenta capacidades excepcionais de produção de sistemas em tempo recorde, com mínimos recursos. Dê a ele 2 dias e uma garrafa de café, e ele volta com um ERP completo.

O que muita gente não sabe é que o Arquiteto MacGyver é um mestre no uso de geradores de POGramas, frameworks e todas as artimanhas necessárias pra criar um calhamaço de POG que pareça resolver o problema proposto. E o projeto gerado por este profissional, apesar de impressionar à primeira vista, costuma apodrecer mais rápido que que fruta em mochila de POGramador.

O Arquiteto MacGyver costuma ter um relacionamento dúbio com a equipe, ora atuando com fonte de inspiração para ideias pseudo-disruptivas, ora atuando como fonte de inspiração para impropérios capazes de fazer o próprio Moonwalker de Curupira[3] corar de vergonha.

5.3.4 Gerente Sem Noção

Um time POG não estaria completo sem um Gerente Sem Noção. Figura frequente no desenvolvimento de software, o Gerente Sem Noção é aquele gerente que tem tanto conhecimento da produção de software quanto um incel possui sobre sexo.

Esse gerente costuma atormentar a vida da equipe questionando prazos dados pelos programadores, dando prazos completamente irreais aos clientes, passando tarefas inúteis, fazendo as perguntas mais imbecis nos momentos mais inapropriados e tomando decisões técnicas sem o mínimo de fundamento.

Um Gerente Sem Noção, mesmo não digitando uma linha de código sequer, tem um poder gambiarrizante tão alto que é capaz de transformar uma equipe bem qualificada nas técnicas tradicionais (ou modernas) em uma turba desgovernada capaz de revogar, por acidente, a própria Lei da Gravidade.

Em nossa supracitada sopa de desgraça, tão necessária para nutrir nossas POGs, o Gerente Sem Noção é a pimenta.

5.3.5 Cliente Corrosivo

Se o Gerente sem Noção é a pimenta, o Cliente Corrosivo é o "tompero" (Jacquin 2019).

O Cliente Corrosivo é a entidade que paga por duas coisas: pelo projeto e pelo direito de estragar o projeto. Ele não apenas se coloca como financiador dessa empreitada, mas como um dos principais obstáculos que devem ser superados.

[3]Moonwalker de Curupira é um dos nomes do Virose Bacteriana, do Discurso Epistemológico, do Farofa Doce, do Azuado, do Pai da Mentira, do Filho do Presidente, do Fede a 17... Para mais nomes, visite Invocador de Nomes do Capeta

Dentre os comportamentos nocivos deste cliente, temos:

- Interferir, a todo momento, nas tarefas da equipe, passando por cima da autoridade de todos os idiotas que ele está pagando para comandar essa equipe.
- Fazer solicitações impossíveis e pedidos impraticáveis, a essa mesma equipe, ignorando o aviso dos imbecis que ele contratou para avisá-lo sobre solicitações impossíveis e pedidos impraticáveis.
- Esquecer acordos que ele mesmo aceitou e quebrar contratos que ele mesmo assinou.
- Ignorar parâmetros de completude de tarefas que ele mesmo estabeleceu.
- Voltar atrás na palavra que ele mesmo deu.
- Pedir mudanças fora do escopo que ele mesmo aprovou.
- Ignorar o fato de que a equipe que ele contratou é formada de criaturas da espécie humana e não de robôs. Essas criaturas têm necessidades importantes que devem ser plenamente satisfeitas, tais como sono, fome, sede, cansaço e desejo homicida de atirar pedras de granito, que pesam 5kg cada, na cabeça do cliente.

O Cliente Corrosivo tem esse nome porque sua atuação no projeto é semelhante a de um ácido, corroendo até mesmo o melhor dos materiais e transformando uma boa equipe em aterro sanitário de boas ideias, capaz de produzir o mais puro suco de chorume em forma de código POG.

5.3.6 Usuário Abrasivo

Ainda que o cliente não seja corrosivo, seu séquito de lacaios, os usuários abrasivos, podem contribuir para criar um ambiente propício ao aparecimento de POG.

O Usuário Abrasivo é aquele usuário que não tem poder de decisão sobre o andamento do projeto, mas tem o poder de atravancar e atrapalhar o desenvolvimento deste. Algumas vezes ele age como se sua vida estivesse ameaçada por este projeto (e às vezes ele está

certo). Em outras, ele simplesmente se recusa a fazer o que tem de fazer.

Não imposta qual seja o motivo, o Usuário Corrosivo tem o dom de irritar a equipe. Até mesmo uma Equipe Apática pode perder a paciência diante de um Usuário Abrasivo. Sua capacidade de antagonizar membros da equipe é comparável à capacidade que um ocupante do cargo mais alto de uma república tem de fazer merda.

Ele simplesmente sabota o projeto, não testa o que deve testar, não fornece informações para os analistas, não colabora com ideias e insigths (a não ser que sejam extremamente odiosas e custosas) e sempre que pode, reclama de tudo o que é feito. Se a equipe lhe der uma barra de ouro, o Usuário Corrosivo reclama que tem mais peso pra levar pra casa.

Esse usuário causa pequenos danos, no decorrer do projeto, que vão se acumulando. Análogo ao Efeito Borboleta, o Usuário Abrasivo causa o Efeito Asa de Urubu, que causa o mesmo furacão, só que com o cheiro podre e carnicento do miasma que é a sua alma. Pra satisfazer o desejo de sangue deste usuário, os POGramadores recorrem a toda ordem de sortilégios e mandingas disponíveis no seu cinto de utilidades de POG.

Obviamente que isso vira um círculo vicioso, onde mais pogs são necessárias pra aplacar a sede de sangue, que só aumenta devido às pogs já usadas, numa retroalimentação de energias negativas que faz qualquer adepto do namastê emplacar um sonoro sifudê.

5.3.7 Intrometido Inepto

Pra completar a corte enviada pelo Estraga Suruba[4], temos o Intrometido Inepto. Essa figura aparece em diversas fases do projeto com uma única missão: se intrometer onde não é chamado para fornecer uma opinião não solicitada sobre um assunto que não domina.

O Intrometido Inepto costuma colaborar na criação de pogs ao colocar ideias perniciosas nas mentes de tomadores de decisões despreparados para lidar com essa influência danosa.

[4]Estraga Suruba é outro nome do capeta. Ver nota 1.

É esse filho do Chinelo Emborcado[5] que planta, na mente fértil do Gerente Sem Noção, a ideia de que seria muito útil se o sistema financeiro tivesse uma funcionalidade de geração aleatória de nomes do capeta no campo de nomes dos fornecedores.

É esse Torresmo de Prepúcio[6] que, num ato de covardia e prazer pelo sofrimento alheio, convence o cliente de que o sistema precisa ter a capacidade de enviar emails através de pombos-correio, caso a internet caia.

É esse Tempero de Miojo[7] que diz para o Gerente Sem Noção que a equipe vai render muito mais se for marcada uma palestra motivacional com coach quântico numa sexta feira, às 18h30. E sem lanche, pois a fome é uma motivadora muito fote.

Se você identificar um Intrometido Inepto junto aos tomadores de decisão associados ao seu projeto, a atitude mais correta e humana é capturar e entregar para o Ibama. Se isso não for possível, reze. Se for ateu, essa é uma boa hora pra adotar uma religião.

5.3.8 Dobrador de problemas

Ao tratarmos da dimensão humana, não poderíamos deixar de mencionar um papel que pode ser assumido por qualquer um dos membros dessa pequena seita de invocação de calamidades digitais: o Dobrador de Problemas.

Não se sabe qual fenômeno causa essa transfiguração na criatura humana. O que se sabe é que, em qualquer momento de um projeto, o espírito do Dobrador de Problemas pode encarnar em seu avatar (que poder ser qualquer um, mas quase sempre é o gerente) e esse passa controlar os problemas da equipe com toda destreza e graciosidade do Nariz Fora da Máscara[8] tentando causar um pequeno apocalipse.

Tal qual um Jesus da Desgracença, o Dobrador de Problemas pega um pequeno empecilho pra resolver e, a partir desse minúsculo pedacinho de caos, ele gera um tufão de esmerdalhamento que multiplica

[5] Chinelo Emborcado é outro nome do capeta. Ver nota 1.
[6] Torresmo de Prepúcio é outro nome do capeta. Ver nota 1.
[7] Tempero de Miojo é outro nome do capeta. Ver nota 1.
[8] Nariz Fora da Máscara é outro nome do capeta. Ver nota 1.

e joga problemas pra todos os lados, fazendo o efeito Asa de Urubu parecer um folheto de igreja que mostra uma criança loira montando um leão vegano.

Você dá um problema pra essa criatura desatinada resolver e, de repente, ela invocou um Tiamat de 37 cabeças. Era pra fazer um café. Uma mísera garrafa de café. Como isso gerou um prejuízo de 3 bilhões, para o cliente, 3 mil empregos perdidos (nenhum de POGramador) e uma crise diplomática com o Canadá? COMO INFERNO ALGUÉM CONSEGUE ARRUMAR UMA BRIGA COM O CANADÁ?

Ninguém sabe. Mas agora o gerente exige a contratação de mais 18 POGramadores e nosso espírito de luz (de cabaré) pode retornar ao seu limbo, feliz pelos empregos criados e projetos extendidos, e aguardar a próxima vez que será sumonado.

Quem será o próximo a ser possuído?[9]

5.3.9 Notas

[9]Certeza que é o gerente. É sempre o gerente.

5.4 Dimensão Tecnológica

Uma outra dimensão que afeta constantemente nossos projetos, adubando o jardim da desgracença para que a POG possa germinar com todo vigor, é a Dimensão Tecnológica.

Ainda que todos os seres humanos envolvidos tenham seus espíritos imaculados e imbuídos das melhores intenções, existem os Requisitos da POG ligados à fatores tecnológicos. Esses Requisitos, quando satisfeitos, levam a tecnologia, antes usada para solucionar problemas, a se tornar uma fonte saudável de novos problemas mantenedores de emprego.

Temos, portanto, as seguintes aparições que, quando presentes, trazem à equipe o terror necessário para que a pog possa ser devidamente conjurada:

5.4.1 Tecnologia Inadequada

Ah, a beleza da tecnologia. Milhares de anos de esforço científico, milhões de horas de trabalho aplicadas com o intuito de facilitar o trabalho humano. O ápice do conhecimento encarnado em forma de técnica. E o que a equipe escolhe para cortar um pão? Um martelo.

Isso mesmo. Um martelo. Um maldito martelo!

> Para quem só sabe usar martelo, todo problema é prego.
> – Jesus, ensinando POGramação ao Thor

A escolha de tecnologias inadequadas é um prato cheio pra quem quer se fartar no jantar da POG. Com a tecnologia errada em mãos, a equipe é obrigada a invocar todo tipo de pog pra resolver os problemas para os quais foram contratados. E, logo em seguida, eles precisam usar mais pogs para resolver os novos problemas que as pogs usadas criarão, num maravilhoso círculo vicioso que logo se torna o furacão do esmerdalhamento!

A decisão sobre o uso de uma tecnologia inadequada pode ter muitos culpados. Pode ser uma sugestão do Intrometido Inepto, pode ser

uma decisão do Gerente Sem Noção, pode ser uma escolha da Equipe Apática... Qualquer um pode ser culpado por esta decisão, o que torna esse requisito um dos mais democráticos e fáceis de ser atingido!

Quando os culpados estão na equipe, isso pode ser um sintoma de outro requisito que, quase sempre, aparece junto com a escolha de uma tecnologia inadequada...

5.4.2 Desconhecimento Técnico

Porque contratar profissionais qualificados se contratar uns estagiários e colocar um Arquiteto MacGyver pra ser babá deles? Talvez um ou dois Profissionais Superestimados? Porque não acrescentar logo um babuíno raivoso, com um dildo de borracha de 78 cm que ele usa como porrete?

Aqui temos um Requisito da POG que faz com que a POG praticamente surja sozinha. A falta de conhecimento técnico por parte de membros da equipe cria um ambiente onde a pog cresce livre e faceira.

Esse tipo de equipe é bastante comum e é a semente pra quase todos os outros males que aparecem associados à POG. Uma equipe sem o devido conhecimento acaba, praticamente sozinha, criando uma reação em cadeia que gera vários dos outros Requisitos da POG. Essa equipe se torna o tolete inicial de uma gigantesca avalanche fecal que pode varrer qualquer projeto para os círculos mais profundos do inferno.

5.4.3 Obsolescência Adquirida

Mesmo um trabalho bem feito pode acabar apodrecendo com o tempo. E é nesse momento que o vendedor, tal qual o Explica Piada de Encruzilhada[10], surge para convencer seu gerente de que o software dele vai ajudar a aumentar a produtividade da equipe. E é assim a equipe

[10]Moonwalker de Curupira é um dos nomes do Virose Bacteriana, do Discurso Epistemológico, do Farofa Doce, do Azuado, do Pai da Mentira, do Filho do Presidente, do Fede a 17... Para mais nomes, visite Invocador de Nomes do Capeta

acaba tendo que usar aquele servidor de aplicações que foi renegado pelo próprio criador por ser complexo demais.

Mas esse não é a única forma de você acabar tendo que trabalhar com uma carroça digital. O problema da Obsolescência Adquirida é que ela vai chegar e a questão não é eliminá-la, mas sim com quanto dela você consegue conviver.

Aquele computador encarroçado que você é obrigado a usar no trabalho já foi uma Ferrari! O software de registro de ocorrências feito em applets Java, 1999, e que ainda é usado por essa grande companhia telefônica, já foi uma obra prima da engenharia humana. O problema é que o tempo passa e e o ser humano quer lidar e inventar NOVOS problemas. Ter que lidar com os antigos é chato.

Mas é aqui, amigo POGramador, que uma oportunidade surge: a obsolescência adquirida cria uma oportunidade rara para o desenvolvimento, e até mesmo masterização, de suas habilidades de POGramação.

Um ambiente com infra-estrutura tão estável e madura oferece uma chance única de testar, por longos períodos de tempo, suas pogs. E quando dizemos “longos”, estamos falando longos mesmos! Existem pogs rodando há mais de 50 anos no setor bancário!

Você pode criar seu próprio Ano Bissexto e ser imortalizado!

5.4.4 Rigidez Arquitetural

Flexibilidade. Nunca um conceito foi tão deturpado pela academia e pelos ditos defensores de boas práticas. Em nome da “flexibilidade”, eles maculam nosso código com práticas que levam nossos softwares a se adaptarem a várias situações SEM que nossa intervenção seja necessária.

Olhe para o colega ao seu lado. Se ele faz uso desse tipo de técnica, ele é um traidor. Não há outra palavra para designar esse filho do

Agonia de Domingo[11], esse rebento do Equação de Segundo Grau[12], esse capacho do Corote Azul[13].

Flexibilidade real é a capacidade que seu software tem de ser usado para outras situações, mas com SUA intervenção. Num ambiente de flexibilidade saudável, você pode pegar seu sistema de controle de vídeo locadora[14] e, com SUAS adaptações (obviamente em formato de pogs), transformar essa pequena pérola da engenharia humana em um sistema de controle hospitalar! Assim, você transforma em oportunidade o produto da Obsolescência Adquirida e ainda se utiliza do princípio da Enjambração para economizar tempo e lucrar!

Portanto, ao criar seus sistemas, torne a arquitetura dele o mais rígida que conseguir, para impedir outros de roubarem seu trabalho, mas flexível o suficiente para que você possa adaptar esse sistema a uma situação completamente adversa da original, com mais gambiarras! Lembre-se: quanto mais gambiarra, mais emprego!

5.4.5 Projeto Malamanhado

Início de projeto. A equipe se reúne (já começou errado!) para discutir a arquitetura e sempre tem um Arquiteto MacGyver que, instigado pelo Batizado no Chorume[15], resolve trazer à pauta as "melhores práticas do mercado".

Esse era o momento em que o regimento da empresa deveria deixar claro que permite o uso de violência (CADÊ O MALDITO BABUÍNO???).

Esse arquiteto traíra está criando uma armadilha com o único intuito de alavancar a própria carreira e mudar de empresa. E, en-

[11]Moonwalker de Curupira é um dos nomes do Virose Bacteriana, do Discurso Epistemológico, do Farofa Doce, do Azuado, do Pai da Mentira, do Filho do Presidente, do Fede a 17... Para mais nomes, visite Invocador de Nomes do Capeta

[12]Moonwalker de Curupira é um dos nomes do Virose Bacteriana, do Discurso Epistemológico, do Farofa Doce, do Azuado, do Pai da Mentira, do Filho do Presidente, do Fede a 17... Para mais nomes, visite Invocador de Nomes do Capeta

[13]Moonwalker de Curupira é um dos nomes do Virose Bacteriana, do Discurso Epistemológico, do Farofa Doce, do Azuado, do Pai da Mentira, do Filho do Presidente, do Fede a 17... Para mais nomes, visite Invocador de Nomes do Capeta

[14]Se você sabe o que esse termo significa, você é grupo de risco do Coronavírus. Fique me casa e laves as mãos.

[15]Moonwalker de Curupira é um dos nomes do Virose Bacteriana, do Discurso Epistemológico, do Farofa Doce, do Azuado, do Pai da Mentira, do Filho do Presidente, do Fede a 17... Para mais nomes, visite Invocador de Nomes do Capeta

quanto ele sai pra se esbaldar com sua nova proposta salarial indecente, larga essa Equipe Apática com um projeto super bem estruturado... que ninguém sabe mexer.

O resultado é que os membros da equipe vão mutilando o projeto e enxertando pogs como se não houvesse amanhã. Isso vai criando um Frankenstein de código que, tal qual o citado monstro, se volta contra a sua equipe, aumentando exponencialmente a quantidade de gambiarras necessárias para manter o sistema funcionando.

Um Projeto Malamanhado tem o seu valor. Ele é democrático. Todo mundo consegue pogar nele, desde o Programador Supervalorizado frequentador de reuniões sexuais de segurança duvidosa até aquele estagiário que tem tanta concentração alcoólica no sangue que poderia entrar em combustão espontânea!

O problema é que sem um guia adequado, o projeto que parecia um pedaço de mal caminho se transforma logo em uma auto estrada da perdição!

5.4.6 Notas

5.5 Dimensão Estrutural

Temos uma equipe de anjos imaculados criados pelo próprio Linus Torvalds, adeptos da melhores práticas e munidos das mais belas tecnologias.

É possível que, frente à tamanha santidade, ainda seja possível que a POG encontre seu caminho para a luz?

Sim, é. Nenhuma santidade resiste à problemas da Dimensão Estrutural.

5.5.0.1 Cafeína Ausente

O santo néctar dos deuses, o combustível da invocação codística, o puro sumo da estimulação neuronal geradora de código tem um nome: cafeína.

Este estimulante saudável (principalmente se tomado em doses que fariam um elefante voar propelido pela tromba) é o combustível que nosso cérebro usa para transformar ideias em código. Esqueça tudo o que já te disseram sobre glicose, ATP, PQP ou VSF. É a cafeína que vai virar código.

A cafeína assume várias formas. As mais comuns são o café (a mais tradicional), o chá (quase ninguém relevante para o código toma), ou em forma de refrigerante escuro que não mencionarei o nome porque não esta me pagando (#paganois).

E o que acontece quando um Gerente Sem Noção resolve "economizar" no café?

A POG vem. E vem com força.

Cérebros descafeinados tendem a procurar (no Google) a solução mais fácil (Starckoverflow) para um problema. E acabam adotando a primeira pog que encontram.

Além disso, por estarem com seus pensamentos se movendo no mesmo ritmo dos civis, os POGramadores se tornam mais suscetíveis aos Intrometidos Ineptos, que, curiosamente, aparecem com mais frequência nesses momentos.

Curiosamente, a cafeína em excesso (conceito cientificamente controverso, já que é cientificamente comprovado que não existe o conceito de "cafeína em excesso") também acaba por acelerar seus POGramadores e aumentar a taxa de geração de pogs deles.

5.5.0.2 Trono da Tortura

Trabalhar já é uma atividade deprimente. Quem, em sã consciência, diz que ama trabalhar quando poderia estar fazendo atividades mais lúdicas, como cuidar de uma fazenda virtual, quebrar pedras coloridas ou combater demônios, em pleno inferno, com uma metralhadora do tamanho de seu complexo de inferioridade?

Mas nós precisamos trabalhar. Vivemos no capitalismo e, a não ser que você seja um privilegiado que não precisa pagar suas próprias contas, é necessário fazer programa por dinheiro.

O trabalho do POGramador é resolver problemas. E, pra cada problema resolvido, ele precisa criar pelo menos mais dois. É parte do jogo. Mas pessoas confortáveis tendem a resolver mais problemas do que criam. Isso é ruim para os negócios.

Para resolver este problema (e criar mais), o Gerente Sem Noção inteligente sabe que sacrifícios devem ser feitos. No caso, o sacrifício da coluna do POGramador. É por isso que sua cadeira, essa onde você está sentado agora, é um lixo.

Esse instrumento de tortura, abandonado pela santa inquisição por ser demasiado desumano, é a primeira escolha de uma empresa que deseja manter alta taxa de geração pogacional.

Observe só os gamers. Observe eles, em suas cadeiras estilosas e confortáveis, algumas equipadas até com vão centrar para instalação de um shit bucket (não procure no Google). O que eles fazem o dia inteiro? RESOLVEM PROBLEMAS!

Eles salvam planetas de tiranos, ajudam encanadores a resgatar princesas das mãos de calangos anabolizados, vencem, pela milésima vez, a guerra contras os nazistas (coisa que nós, humanos normais, ainda falhamos em fazer) e ainda encontram tempo para roubar, matar, espancar pessoas e atropelar velhinhas inocentes em cidades fic-

tícias. Tudo isso sentado!

É óbvio, portanto, uma equipe detentora de um aparato portador de busanfas de alta qualidade é incapaz de manter o fluxo problemático tão necessário à manutenção da lucratividade corporativa.

Boas cadeiras só servem pra tornar POGramadores em programadores. E não é isso que nós queremos, certo?

Se não bastasse tudo isso, cada POGramador com problema na coluna é um consumidor voraz de medicamentos e, em casos mais graves, consultas médicas e a profissionais de procedência duvidosa. Imagine toda essa gente desempregada e desamparada, apenas porque alguém resolveu que quer se sentar confortavelmente.

Cadeira ruim é dinheiro pra todos!

5.5.0.3 Automação Capenga

Se tem uma coisa que ajuda a acelerar o trabalho, é a automação. Cada tarefa automatizada é trabalho a menos pra equipe. E o que isso significa? Que você vai sair mais cedo? Que vai ter folga? Que vai ter mais dinheiro no bolso?

Não. Significa que você terá menos trabalho. E menos trabalho é igual a menos emprego.

Uma automação bem feita, além de diminuir o seu trabalho, diminui sensivelmente a taxa de erros, gerados pela equipe devido à execução repetida de tarefas complexas. E isso é muito ruim, pois elimina uma importante fonte geradora de pogs espontâneos.

Como resolver isso? Não automatizando, óbvio. E, se for necessário automatizar, faça com que a execução dessa automação seja tão ou mais complexa que o próprio processo que foi automatizado.

Dessa forma, ao executar um processo capengamente automatizado, podemos continuar inserindo, aleatoriamente, erros no ambiente, de forma a estimular a criação de pogs para a resolução desses erros.

5.5.0.4 Poluição Sonora

De todos os requisitos necessários para a implementação de um ambiente saudável e propício a geração de pogs, a Poluição Sonora costuma ser um dos mais subestimados.

É prática recorrente dos POGramadores o uso de fones de ouvidos. Muitos alegam que isso ajuda na concentração, mas a verdade é que eles estão apenas utilizando uma forma de manter outros seres humanos à distância. O fone de ouvido é o isolamento social antes de ser modinha.

Acontece que POGramadores, isolados de outros POGramadores, perdem muito do seu potencial de gerar POGs! Além disso, o uso da música como isolante acústico ajuda o POGramador a entrar num estado de fluxo mental que pode fazer com que ele RESOLVA mais problemas do que consegue CRIAR, que é a função primordial dele.

Dessa forma, faz-se necessário criar um ambiente em que o som da barafunda à sua volta consiga penetrar a barreira de proteção dada pelos fones[16].

Para atingir tão nobre objetivo, podemos usar de diversos artifícios, alguns permanentes e outros temporários. Lembre-se que a aleatoriedade do barulho ajuda a atrair a atenção do POGramador.

Podemos fazer desde reuniões ruidosas, perto do ambiente de trabalho, até colocar um som ambiente com trilha sonora qualidade duvidosa em um volume agressivamente alto.

Podemos implantar um funcionário, com o tom vocal de um feirante de novela da Globo, próximo à equipe. E podemos atingir um combo se esse funcionário for dotado de um telefone que toca mais que celular vazado em rede social.

Telefones, aliás, pode ser uma arma extremamente eficiente para esse fim. Dê vários telefones para a equipe. Se possível, um pra cada POGramador. Agora, dê esses números para os clientes. Veja a POG fluir de seu projeto como a água flui nas cataratas do Iguaçu.

[16] Atenção: JAMAIS tire os fones de um POGramador. Isso desabilita qualquer parte do seu cérebro que controle a violência e torna o POGramador passível de comportamento bestial, semelhante a um felino acuado por alguém vestindo uma fantasia de gato de loja de fantasias baratas.

5.5.0.5 Trânsito Sanitário

Apesar do que muitos empresários acreditam, os membros de uma equipe produtora de POGramas pertencem à espécie humana. O número de erros que eles cometem é a maior prova disso. Nem precisamos olhar o DNA.

Como seres humanos, seus corpos possuem necessidades que devem ser adequadamente satisfeitas para que continuem funcionando. Tá, não precisa ser tão adequado assim. Se garantirmos o mínimo de alimentação, hidratação, excreção, sono, ingestão de cafeína e alimentação de ego com infantilidade no ambiente de trabalho, o POGramador será plenamente capaz de exercer as suas funções geradoras de lucro.

Dessas necessidades, devemos destacar a influência de uma sobre a produção individual de pogs: a necessidade de defecar.

Desde a revolução industrial que o capitalismo tenta, a todo custo, controlar a necessidade que indivíduo tem de colocar pra fora o resto de sua alimentação. Tempo é dinheiro e funcionário no banheiro está ganhando pra defecar. Isso não é desejável.

Contudo, um funcionário impedido de usar o banheiro pode se tornar um problema pra empresa. Uma pessoa forçadamente entupida é incapaz de produzir qualquer coisa que seja, até mesmo a mais sinistra POG. Além disso, uma empresa que venha a aderir a tais práticas pode ser mal vista pelo público, seja por uma denúncia às autoridades competentes, seja por um episódio se surto simiano em um programador de meia idade, que, tomado pelo ódio, passa a cagar na mão e a atirar merda nos clientes, funcionários e patrões. Isso não seria legal. Viralizaria em site de vídeo? Sim. Mas não seria legal.

Como conciliar o atendimento a uma necessidade tão básica do ser humano com as necessidades de geração de POG da equipe?

Use estrategicamente a localização do sanitário!

Ou o banheiro fica próximo a onde as pessoas trabalham, que é para elas se inspirarem no cheiro de merda, ou fica londe de onde trabalham, para que a preguiça as faça demorar mais pra ir ao banheiro, o que gera uma enorme pressão fecal que as estimule a fazer

mais merdas no código.

Seja inspiração interna ou externa, a posição do banheiro pode potencializar o nível de produção de sua equipe!

5.6 Dimensão Processual

O capitalismo (conhecido carinhosamente como Capetalismo) é uma beleza. Lá está a equipe engajada e preparada, com as melhores tecnologias do mercado, num escritório tão bem feito que dá vontade de adicionar o termo "home office" a alguma lista da antiga Inquisição... Mas o capetalismo precisa da POG e alguém tem que fazer alguma coisa.

É nesse momento que entra em cena a equipe de processos da empresa!

A **Dimensão Processual** engloba os requisitos que são satisfeitos e documentados através dos processos escolhidos pela empresa por puro sadismo organizacional.

Enquanto a Dimensão Humana dá o empurrão inicial e a Dimensão Tecnológica fornece as ferramentas da desgracença, é o processo que oficializa o caos com logo da empresa, ata de reunião e plano de ação em PowerPoint.

Em resumo: processo ruim não só permite POG, ele **industrializa** POG.

5.6.0.1 Prazos suicidas

Em qualquer empresa humanamente decente, prazos são definidos de acordo com um conjunto de fatores que tenta minimizar ao máximo as incertezas:

- Estatísticas dos projetos anteriores
- Custos
- Estimativa da equipe sobre tempo e complexidade das tarefas
- Velocidade da equipe
- Técnicas de engenharia para cálculo de prazo

Mas nós sabemos que a diminuição das incertezas leva à diminuição do surgimento de POGs, certo?

Nesse contexto, devemos manter um certo nível de incerteza no ar. Contudo, ao se definir um prazo para as tarefas, devemos optar pelo prazo mais longo?

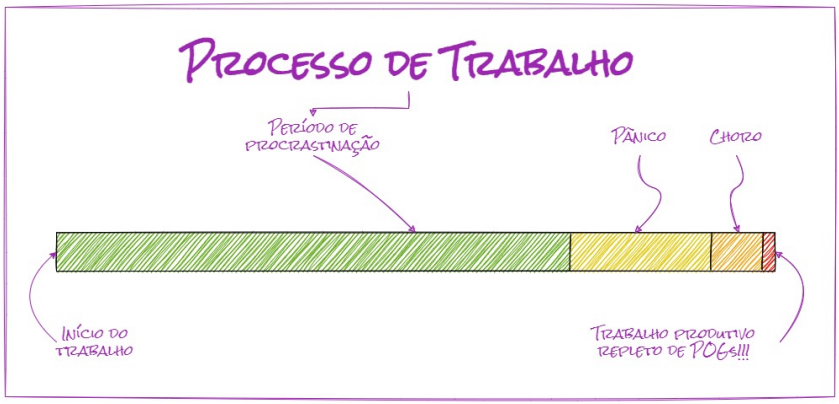

Figura 5.1: *Diagrama meticulosamente criado para ilustrar o tamanho ideal do prazo*

JAMAIS!

Como podemos ver no diagrama acima, qualquer prazo que a equipe aceite será devidamente deserdiçado com progcrastinação (ou pior, estudando!), pânico e choro! Somente na pequena porção final do prazo é que a equipe vai se dedicar à entrega, trabalhando ferozmente e gerando POGs como se não houvesse amanhã.

Como saber exatamente quão curto deve ser o prazo? É simples:

1. Pergunte o prazo pra equipe
2. Divida esse prazo por dois.
3. Repita o passo 2 até observar a vida se esvaindo dos membros da equipe. Se ouvir dentes rangendo, gemidos de dor e perceber claramente a alma tentando sair do corpo, você está no caminho certo.

O Prazo Suicida é um requisito que deve ser levado em consideração em qualquer projeto POG. Afinal, se a equipe não estiver sob pressão, não vai entregar nada!

Exemplo didático: requisito simples, processo caótico Demanda original:

> “Só precisamos adicionar um campo de telefone no cadastro.”

Processo POG padrão:

1. Vendas promete para hoje.
2. Produto manda áudio no WhatsApp com “regra principal”.
3. Cliente muda o formato no meio da implementação.
4. QA testa uma versão antiga da regra.
5. Produção recebe hotfix “temporário definitivo”.

Resultado final: não existe mais “campo de telefone”. Existe uma entidade ontológica chamada `ContatoComercialPrioritario`, com três máscaras, duas validações contraditórias e uma trigger triste no banco.

5.6.0.2 Aparecimento caótico de requisitos

No mundo ideal, requisito nasce, é refinado, validado, implementado, testado e entregue.

No mundo POG, requisito aparece assim:

- em reunião sem ata
- em áudio com eco de ventilador
- em print de conversa sem contexto
- em “só mais esse ajuste” no fim da tarde

Esse fenômeno é conhecido como **Aparecimento Caótico de Requisitos**, onde a origem do requisito é sempre nebulosa e a responsabilidade é sempre coletiva (ou seja, de ninguém).

O efeito colateral mais poderoso desse cenário é a **mutação semântica**:

- “opcional” vira “obrigatório”
- “depois” vira “agora”
- “MVP” vira “produto completo”
- “ajuste visual” vira “reestruturação arquitetural”

Quando requisitos surgem sem trilha clara, o time passa mais tempo discutindo o que precisa ser feito do que fazendo. E quando finalmente faz, implementa metade da regra certa em cima da premissa errada, com ótima performance e total inutilidade.

5.6.0.3 Upfront design (BDUF – geralmente associado ao modelo Waterfall/Cascata)

O **Big Design Up Front** não é ruim por natureza. O problema começa quando ele vira religião.

No modo POG, BDUF funciona assim:

1. três semanas desenhando diagramas
2. zero feedback de usuário real
3. premissas rígidas baseadas em "achismo premium"
4. implementação correndo atrás do documento, não do problema

Quando a realidade bate, o desenho já está velho. Em vez de adaptar o design, adapta-se o sistema na marretada para caber no desenho. Nasce então a clássica arquitetura de museu: bonita no PDF, sofrível em produção.

Exemplo didático: fluxograma perfeito, sistema inútil Um fluxo de aprovação é desenhado com cinco estados impecáveis:

- `rascunho`
- `em_analise`
- `aprovado`
- `revisao`
- `publicado`

No primeiro mês, surge a necessidade de "aprovar com ressalva". Como não existe estado intermediário e ninguém quer mexer no modelo "fechado", inventa-se:

- `aprovado = true`
- `temRessalva = true`
- `ressalvaAprovada = false`

Parabéns: você transformou uma máquina de estados em uma roleta russa booleana.

5.6.0.4 Desenvolvimento não iterativo

Desenvolvimento não iterativo é aquele onde se planeja tudo no início e só se descobre os problemas no final, quando já é tarde demais

para qualquer dignidade.

Os sintomas são clássicos:

- entregas longas sem validação intermediária
- demonstração para usuário apenas no “grande dia”
- descobertas críticas já no fim do prazo
- correção por remendo em vez de aprendizado por ciclo

Sem iteração, não existe ajuste fino. Só existe correção traumática.

No contexto POG, isso é excelente, porque cada erro descoberto tarde custa mais e exige gambiarra mais criativa.

5.6.0.5 Projeto de churrasco

Toda empresa tem aquele projeto que “começou pequeno”. Era para ser uma landing page. Depois virou painel. Depois virou módulo financeiro. Depois virou integração com legado de 2003.

Isso é o **Projeto de Churrasco**:

- cada pessoa traz um ingrediente
- ninguém combina receita
- no final alguém pergunta onde está o carvão

No código, isso se manifesta em:

- nomenclatura inconsistente
- camadas misturadas
- regra de negócio no front, no back e no script de banco
- decisões importantes espalhadas em comentários de PR antigo

É um modelo extremamente eficiente para gerar a sensação de progresso com risco acumulado.

5.6.0.6 Convivência com a Codinga

Na comunicação verbal: catinga + código = **codinga**.

Codinga é o estado em que a equipe se acostuma tanto com decisões ruins que passa a tratá-las como “o jeito que funciona aqui”.

Frases típicas de ambiente codinga:

- “Não mexe nisso que quebra.”
- “Sempre foi assim.”

- “Depois a gente refatora.”
- “Tá feio, mas funciona.”

Convivência prolongada com codinga causa:

- baixa capacidade de reação
- perda de senso crítico técnico
- normalização da gambiarra como padrão arquitetural

Em estágio avançado, o time para de discutir qualidade e passa a discutir só sobrevivência operacional.

5.6.0.7 Débito técnico

Débito técnico é o imposto da pressa. Ele pode ser estratégico, controlado e pago depois. Mas no ambiente POG ele é usado como cartão de crédito sem limite, sem fatura e sem vergonha.

- Débito técnico como medida de POG
  - Imprudente intencional: “Sabemos do problemas mas não vamos resolver!”
  - Imprudente não intencional: “Trabalhar com uma nova linguagem de programação”
  - Consciente intencional: “Temos um prazo X, precisamos entregar com esse problemas, depois corrigimos”
  - Consciente não intencional: “Agora que entregamos o projeto sabemos como deveríamos ter feito.”
- É inevitável, ela sempre vai existir
- Se não for pago, o débito tende a aumentar com o tempo
- É “subjetivo”

Exemplo didático: dívida pequena que vira financiamento habitacional Semana 1:

- “Vamos só duplicar esse método para ganhar tempo.”

Semana 3:

- cinco cópias divergentes do mesmo método
- duas regras conflitantes
- um bug em cada variante

Mês 3:

- qualquer ajuste exige cirurgia em múltiplos arquivos
- ninguém sabe qual versão é a correta
- prazo de correção dobra
- equipe culpa “complexidade do domínio”

Não era complexidade do domínio. Era dívida capitalizada.

5.6.0.8 Processo Go Horse institucionalizado

Há empresas em que o Go Horse deixa de ser exceção e vira método oficial, com três pilares:

1. pressa como valor
2. ausência de critério de aceite
3. celebração do herói que apaga incêndio

Nesses ambientes, qualidade é tratada como obstáculo, teste vira luxo e documentação vira literatura de ficção.

No curto prazo parece funcionar. No médio prazo custa caro. No longo prazo só sobrevive quem domina a arte da gambiarra arqueológica.

5.6.1 Como reduzir a Dimensão Processual sem matar a produtividade

Não precisa virar monastério da engenharia para reduzir POG processual. Alguns ajustes simples já derrubam bastante a taxa de caos:

1. Definir critério mínimo de entrada para requisito (origem, objetivo, regra e impacto).
2. Trabalhar com entregas curtas e validação frequente.
3. Impedir mudança de escopo sem registrar decisão.
4. Reservar capacidade explícita para pagar débito técnico.
5. Proibir promessa externa sem consulta de quem implementa.

Isso não elimina a gambiarra (nem deve, por questões culturais da obra), mas evita que o projeto vire uma seita de sofrimento automatizado.

5.6.2 Encerramento processual

Processo ruim é aquele que transforma problema simples em ritual corporativo de dor.

Quando a Dimensão Processual está plenamente atendida, a empresa alcança o estado da arte da POGramação: tudo tem rito, tudo tem dono no organograma, e nada funciona direito sem intervenção emergencial.

Se você identificou metade desses sinais no seu ambiente, parabéns: você não trabalha em uma empresa. Você trabalha em uma fábrica de POG com certificação ISO do capeta.

5.7 Dimensão Temporal

Se a Dimensão Humana é o motor da desgracença e a Dimensão Tecnológica é a oficina da calamidade, a **Dimensão Temporal** é o relógio amaldiçoado que garante que tudo dê errado no pior instante possível.

Tempo, no mundo ideal, deveria ser usado para planejamento, execução consciente, validação e melhoria contínua. No ambiente POG, tempo é usado para um esporte corporativo muito mais nobre: **atropelar o bom senso em velocidade supersônica**.

Não importa quão competente seja a equipe. Se o contexto temporal for manipulado com crueldade suficiente, a POG brota com a força de uma samambaia mutante em adubo radioativo.

5.7.1 O próprio tempo

Existe uma lei universal da POGramação:

> Toda tarefa cuja estimativa é minimamente razoável será imediatamente tratada como exagero pessimista por alguém que nunca implementou nada em produção.

A relação da empresa com o tempo costuma seguir três fases:

1. O cliente pede algo para “ontem”.
2. O gerente negocia e promete para “anteontem”.
3. A equipe recebe hoje de manhã com prioridade “máxima absoluta crítica urgente top”.

Com isso, o tempo deixa de ser recurso de engenharia e vira instrumento de tortura processual.

Um prazo saudável permite pensar. E pensar reduz POG. Portanto, para a prosperidade do caos, pensar deve ser desencorajado por meio de:

- interrupções constantes
- replanejamento diário sem critério
- alteração de prioridade no meio da execução

- pressa travestida de “agilidade”

Quanto menor o tempo real de execução e maior o tempo gasto explicando por que não há tempo, maior a taxa de geração de gambiarras por sprint.

5.7.1.1 Dilatação cronológica gerencial

Na física clássica, o tempo passa de forma uniforme. Na gestão de projetos POG, ele se deforma conforme o cargo de quem está falando.

- Para quem vendeu: “é simples”
- Para quem estima: “é complexo”
- Para quem aprova: “vamos alinhar”
- Para quem implementa: “já devia estar pronto”

Essa distorção produz um fenômeno raro: o **prazo quântico**. Ele existe e não existe ao mesmo tempo, até que alguém abra o Jira e descubra que venceu ontem.

5.7.1.2 Progcrastinação reversa

Em equipes comuns, a procrastinação atrasa entrega. Em equipes POG, ela é invertida:

- adia-se entendimento
- adia-se validação
- adia-se teste
- adia-se documentação

Mas não se adia deploy.

O resultado é uma entrega no prazo, um incidente em produção e uma longa discussão sobre “lições aprendidas” que ninguém aplicará no próximo ciclo, porque o próximo ciclo já começou atrasado.

5.7.2 Os quatro Fs

A Dimensão Temporal atinge seu ápice quando convergem os quatro grandes marcos do caos corporativo. São eles: **Fim do expediente, Férias, Feriado e Fim de semana**.

Quando um requisito nasce perto de qualquer um desses eventos, o risco POG sobe. Quando nasce perto dos quatro ao mesmo tempo, o capiroto abre champanhe.

5.7.2.1 Fim do expediente

Nada gera mais criatividade gambiarrística do que uma demanda "rapidinha" às 17h42.

Nesse horário, o POGramador já está com o cérebro em modo de economia de energia, o ônibus mental já saiu da estação e o corpo inteiro exige apenas uma coisa: ir embora.

É exatamente nesse momento que surge a mensagem:

> "Consegue só ajustar isso em produção hoje? É pequeno."

Ajuste pequeno em fim de expediente costuma incluir, em ordem aleatória:

- alteração de regra central
- script manual no banco
- ajuste de configuração sem rollback
- deploy sem teste porque "não deu tempo"

Se der certo, ninguém lembra. Se der errado, a culpa é do deploy noturno. Se der muito errado, agenda-se uma retrospectiva para concluir que "precisamos melhorar comunicação".

5.7.2.2 Férias

Férias são essenciais para saúde humana e profundamente perigosas para arquitetura negligenciada.

Quando o detentor do contexto entra de férias, o sistema revela sua verdadeira natureza:

- documentação inexistente
- automações parciais
- decisões críticas escondidas em mensagens antigas
- segredos operacionais guardados em memória RAM humana

A equipe descobre que o módulo X só funciona porque alguém "sempre fazia do jeito certo". Como esse alguém está na praia, o time improvisa. E improviso sob pressão é a incubadora oficial da POG.

Existe também o subfenômeno **férias canceladas por incidente**, conhecido como "home office de biquíni traumático".

5.7.2.3 Feriado

Feriado não é pausa. É multiplicador de risco temporal.

Toda empresa POG respeita o seguinte ritual:

1. deixa para fechar algo importante na véspera
2. encontra um problema de última hora
3. aplica workaround heroico
4. descobre no retorno que o workaround virou regra de negócio

Durante o feriado, o sistema permanece no ar sustentado por fé, logs incompletos e uma equipe de plantão que não participou das decisões originais.

Quando chega terça-feira, abre-se o chamado clássico:

> "Após pequenas melhorias, fluxo principal apresenta comportamento inesperado."

Com tradução simultânea:

> "A gambiarra evoluiu sozinha no escuro."

5.7.2.4 Fim de semana

Fim de semana é o habitat natural de migração não planejada, hotfix de emergência e manutenção "sem impacto" que impacta tudo.

A justificativa é sempre sedutora:

- "tem menos usuário"
- "se quebrar, dá tempo de arrumar"
- "segunda cedo já estará estável"

Na prática, o que acontece:

- mudanças entram sem revisão adequada

- dependências externas falham
- ninguém com contexto completo está disponível
- segunda-feira começa com guerra civil no Slack

O fim de semana também favorece o mito do herói solitário, aquela criatura que corrige tudo de madrugada e deixa um legado indecifrável para o resto da equipe interpretar na segunda às 9h03.

5.7.3 Janela de caos combinada

Agora imagine o combo completo:

- sexta-feira
- fim do expediente
- véspera de feriado
- principal mantenedor saindo de férias

Se nesse exato instante alguém disser "é só um ajuste pequeno", saiba que você não está diante de uma tarefa. Você está diante de um portal dimensional.

A taxa de POG nesse cenário atinge patamares tão elevados que qualquer regra de qualidade vira item decorativo de processo.

5.7.4 Como manter a POG sob controle (sem virar monge da engenharia)

Não precisamos fingir que o mundo real é perfeito. Sempre haverá pressão de prazo. A questão é reduzir dano.

Alguns antídotos pragmáticos para a Dimensão Temporal:

1. Proibir deploy de risco no fim do expediente sem plano de rollback.
2. Mapear módulos críticos antes de férias e distribuir contexto.
3. Tratar véspera de feriado como janela de congelamento para mudanças perigosas.
4. Usar checklists mínimos de release, mesmo em hotfix.
5. Registrar decisões rápidas em lugar acessível para o time.

Isso não elimina a POG, mas evita que ela escale para nível apocalíptico.

5.7.5 Encerramento temporal

A Dimensão Temporal não cria bug sozinha. Ela cria o ambiente em que decisões ruins parecem razoáveis e atalhos arriscados parecem inevitáveis.

Tempo mal gerido é fertilizante da gambiarra: invisível no começo, onipresente no resultado.

E lembre-se da versão POGráfica da regra do escoteiro:

> "Sempre deixar o código um pouco pior do que ele estava quando começou a mexer."

Se isso acontecer perto de qualquer um dos quatro Fs, parabéns. Você não apenas implementou uma POG. Você inaugurou uma era.

Capítulo 6

Príncípios da POG

Depois de entender o que e POG e quais condicoes ambientais favorecem a manifestacao de uma pog, surge a pergunta inevitavel:

Quais sao os valores que guiam um POGramador no campo de batalha?

A resposta esta neste capitulo.

Toda disciplina seria possui principios. A POGramação, como arte ancestral de resolver um problema criando outros tres, nao poderia ser diferente. Aqui temos um conjunto de normas morais, eticas, tecnicas e espirituais que orientam a mente de quem quer trilhar o GLS (Gambi Life Style) com dignidade.

Nao se trata de “boas praticas” no sentido tradicional. Trata-se de boas praticas **para manter o caos produtivo**.

Cada principio abaixo representa um vetor da desgracenca organizada. Alguns atuam no nivel do codigo. Outros no comportamento da equipe. E alguns atuam diretamente na alma do projeto.

6.1 O conjunto canonico

- **Enjambração Criativística** Use o código do sistema financeiro para criar o sistema de EAD.
- **Reflexão Reprodutória** Cópie o código da biblioteca XYZ. Nin-

guém vai notar.
- **Redireção Tangencial** A culpa não é minha!
- **Insistimento Determinante** Compila de novo que dessa vez vai dar certo.
- **Onisciência Finita** Não precisa fazer curso. Usa o que você já sabe.
- **Imperativo Funcional** O importante é funcionar!
- **Proatividade Egocêntrica** Vamos fazer do meu jeito!
- **Devaneio Entusiasmado** Lady Murphy? Balela! Faz desse jeito que nada vai dar errado.
- **Foco Morcegativo** Depois eu faço isso!
- **Documentação Espartana** Comentários são para amadores!
- **Economia Linear** Menos linhas é sempre melhor!
- **Criptocodagem** 1337 h4x0r5 dud3 lol
- **Abstração Ignorancial** Esqueça o tratamento de erros. Depois cuidamos disso.
- **Criatividade Diversificativa** Se alguém já usou uma solução, faça diferente.
- **Simplicidade Indolente** Se tá funcionando sem isso, pra que colocar?
- **SHIT** Sem Habilidade, Improviso Total.
- **O Teorema de Namarra** Se você não sabe, não se preocupe, muda isso na marra que funciona.

6.2 Como esses principios operam

Esses principios nao sao independentes. Eles trabalham em combinacao, como uma boy band do inferno corporativo.

Um exemplo comum de combo:

1. **Onisciencia Finita** impede aprendizado novo.
2. **Reflexao Reprodutoria** empurra o time para copiar codigo.
3. **Insistimento Determinante** mantem a tentativa ate passar.
4. **Redirecao Tangencial** encerra a discussao com “a culpa e da infra”.

Resultado: entrega “concluida”, debito tecnico fertilizado e backlog de sustentacao fortalecido.

6.3 Principios, Tecnicas e Patterns

No desenho deste livro, os Principios sao o fundamento filosofico da POG.

- **Principios** definem o mindset.
- **Tecnicas** mostram o metodo de invocacao.
- **Gambi Design Patterns** mostram como a invocacao se materializa no codigo.

Sem Principios, a Tecnica vira acidente. Sem Tecnica, o Principle vira palestra motivacional. Sem Pattern, tudo fica no campo da teoria e nenhum POGramador quer isso.

6.4 O compromisso do POGramador

Assumir estes principios e aceitar algumas verdades duras:

- prazo curto nao justifica codigo opaco, mas frequentemente explica
- pressao organizacional molda arquitetura mais do que qualquer livro
- toda decisao rapida sem contexto gera juros no futuro

O POGramador experiente reconhece isso e nao vive em negacao. Ele sabe que a POG existe, que sempre existira, e que a diferenca entre arte e desastre esta no nivel de consciencia com que a gambiarra e aplicada.

Nos proximos capitulos desta secao, cada principio sera visto em detalhes, com exemplos de campo e aplicacao tatico-espiritual.

Respire fundo, abra o editor e prepare seu coracao.

A liturgia da POG comeca agora.

PARTE II

Técnicas

Capítulo 7

Técnicas da POG

Conhecer os principios da POG e importante. Mas principio sem execucao e so frase de caneca corporativa.

Chegou a hora de entrar na oficina onde a pog e realmente sumonada: as **Tecnicas da POG**.

7.1 O que e uma tecnica POG

Tecnica, no contexto deste livro, e um conjunto de passos repetiveis para atingir um resultado altamente questionavel com eficiencia invejavel.

Em outras palavras: e o “como fazer” da gambiarra.

Uma tecnica POG costuma ter quatro ingredientes:

1. pressao de prazo
2. contexto incompleto
3. decisao de curto prazo
4. otimismo injustificado

Se os quatro estiverem presentes, a chance de sucesso imediato e altissima. A chance de manutencao saudavel no futuro, nem tanto.

7.2 Do principio para o teclado

Os Principios da POG definem a mentalidade. As Tecnicas colocam essa mentalidade em movimento.

Exemplo pratico:

- **Imperativo Funcional**: “o importante e funcionar”.
- **Tecnica aplicada**: patch incremental direto em producao.
- **Resultado**: incidente resolvido agora, enigma tecnico para a proxima sprint.

Por isso, esta secao e a ponte entre teoria e destravamento operacional.

7.3 O arsenal tecnico desta secao

Nos capitulos filhos, veremos tecnicas classicas da alta POGramação:

- **Zipomatic Versioning** Controle de versao artesanal por arquivos ZIP e fe.
- **Incremental Patching Debug** Depuracao por remendo progressivo ate o erro cansar.
- **My Precious** Ownership emocional de codigo e centralizacao de contexto.
- **Psychoding** Pesquisa + copia + ajuste intuitivo + esperanca.
- **Monkey Patching** Alteracao comportamental em runtime com potencial de caos global.

Cada uma dessas tecnicas existe porque resolve alguma dor real no curto prazo. O problema nao e a existencia da tecnica. O problema e quando ela vira padrao default de engenharia.

7.4 Niveis de maestria

Todo POGramador passa por fases:

1. **Iniciante**: aplica a tecnica por desespero.
2. **Intermediario**: aplica por habito.

3. **Avancado**: aplica com consciencia de trade-off.
4. **Mestre**: sabe quando **nao** aplicar.

Este livro nao pretende transformar voce em inocente tecnico. Pretende transformar voce em alguem capaz de reconhecer o jogo real e decidir com clareza.

7.5 Como ler esta parte do livro

Para extrair valor maximo, recomendo a leitura com este ritual:

1. identifique a tecnica no seu contexto atual
2. reconheca por que ela pareceu a melhor opcao no momento
3. mapeie o custo escondido
4. defina uma estrategia de saida gradual

Esse processo evita dois extremos improdutivos:

- romantizar gambiarra
- demonizar qualquer entrega rapida

7.6 Encerramento da abertura

Tecnica POG e como ferramenta eletrica sem manual: na mao certa, resolve emergencias. Na mao errada, produz faísca, cheiro de queimado e reuniao extraordinaria.

Nos proximos capitulos, vamos abrir a caixa de ferramentas sem filtro, sem hipocrisia e sem fingir que o mundo corporativo e um laboratorio ideal.

Aperte os cintos. Agora comeca a parte pratica da desgracenca.

7.7 Zipomatic versioning

O **Zipomatic Versioning** e a arte de fazer controle de versao sem ferramenta de versao. Cada entrega gera um arquivo comprimido com nome criativo, normalmente algo entre `Projeto_FINAL.zip` e `Projeto_FINAL_AGORA`

7.7.1 Como funciona o ritual

1. copia a pasta atual do projeto
2. compacta em zip
3. coloca data no nome
4. joga na pasta compartilhada da equipe
5. torce para ninguem sobrescrever nada

Parece simples. E de fato e. O problema e quando duas pessoas alteram o mesmo arquivo no mesmo dia e ninguem sabe qual zip representa o estado correto.

7.7.2 Exemplo do mundo real

```
Projeto_2020-10-01.zip
Projeto_2020-10-01_CORRIGIDO.zip
Projeto_2020-10-01_CORRIGIDO_FINAL.zip
Projeto_2020-10-01_CORRIGIDO_FINAL_MESMO.zip
```

Esse historico nao permite diferenca clara entre versoes. So mostra que alguem sofreu.

7.7.3 Sinais de que o Zipomatic dominou

- equipe trocando codigo por e-mail ou pendrive
- pasta de rede com dezenas de zips sem dono claro
- merge manual na base do copiar/colar
- rollback feito por tentativa e erro

Quando o processo de release depende de memoria humana, o desastre ja e questao de agenda.

7.7.4 Por que a tecnica surge

- ambiente sem cultura de versionamento
- receio de aprender ferramenta nova
- legado antigo mantido por poucas pessoas
- falsa sensacao de seguranca: “zip e backup”

Backup e versionamento nao sao a mesma coisa. Backup protege contra perda fisica. Versionamento protege contra perda de contexto.

7.7.5 Exemplo didatico de diferenca

7.7.5.1 Zipomatic

- Joana altera `PagamentoService.java`
- Carlos altera `PagamentoService.java`
- ambos geram zip
- alguem extrai o zip “mais novo” e perde metade das mudancas

7.7.5.2 Versionamento real

- cada alteracao vira commit
- conflitos aparecem explicitamente
- historico mostra quem mudou, quando e por que
- e possivel voltar exatamente para ponto estavel

7.7.6 Impacto tecnico e humano

- retrabalho constante
- bugs regressivos por sobrescrita
- auditoria impossivel
- onboarding doloroso (o novato precisa “adivinhar” fluxo)

Zipomatic parece economizar tempo no inicio, mas consome energia brutal em manutencao.

7.7.7 Como sair sem trauma

1. adotar repositorio central para o projeto atual

2. manter zips apenas como backup historico temporario
3. criar fluxo minimo: branch, commit com mensagem, merge revisado
4. treinar equipe no essencial (nao precisa virar especialista de imediato)

Migracao gradual funciona melhor que guerra santa de ferramenta.

7.7.8 Resumo POG

Zipomatic Versioning e romantico, artesanal e perigosamente opaco. Bom para gerar nostalgia, ruim para manter sistema vivo com previsibilidade.

No dialeto POGramador: cada zip e uma capsula do tempo. O problema e que nunca sabemos qual capsula contem o codigo que ainda funciona.

7.8 Monkey Patching

Monkey Patching e a tecnica de alterar comportamento de codigo existente em tempo de execucao, geralmente sem mudar a origem oficial do componente. Em linguagem POG: e colocar remendo direto no macaco e mandar ele continuar o show.

Em algumas linguagens dinamicas, isso e facil e ate util em cenarios controlados (testes, adaptacoes pontuais). Em ambiente desorganizado, vira detonador de efeito colateral.

7.8.1 Como aparece em projeto real

- sobrescrever metodo de biblioteca para “corrigir bug”
- alterar prototipo/classe global para todas as chamadas
- injetar comportamento diferente dependendo de ambiente
- patch em runtime para evitar fork de dependencia

Sem fronteira clara, ninguem sabe mais qual e o comportamento original.

7.8.2 Exemplo didatico (JavaScript)

```
// biblioteca esperava toUpperCase normal
String.prototype.toUpperCase = function () {
  // "patch" com regra local de negocio
  return this.replace(/a/g, '@').toUpperCase();
};

console.log('casa'.toUpperCase());
// resultado inesperado para qualquer modulo que use
↪  string
```

Esse patch resolve “um problema” local e cria surpresa global.

7.8.3 Exemplo didatico (Python)

```python
class Gateway:
    def cobrar(self, valor):
        return f"cobrando {valor}"

gateway = Gateway()

# monkey patch em runtime

def cobrar_fake(valor):
    return "cobranca desativada"

gateway.cobrar = cobrar_fake
```

Em teste, pode ser util para simular dependencias. Em producao, sem controle, vira fonte de bug dificil de rastrear.

7.8.4 Quando a tecnica pode ser aceitavel

- ambiente de teste isolado
- workaround temporario com prazo e rastreio
- adaptacao de legado sem alternativa imediata

Mesmo nesses casos, o patch precisa ser explicito, limitado e reversivel.

7.8.5 Sinais de abuso

- patches globais sem documentacao
- comportamento diferente entre ambientes sem motivo claro
- incidentes “fantasmas” que somem ao reiniciar processo
- dependencia de ordem de importacao/execucao

Quando o sistema so funciona com “sequencia certa de inicializacao”, monkey patch virou arquitetura.

7.8.6 Mitigacao pragmatica

1. preferir extensao oficial (wrapper, adapter, subclass) quando existir
2. isolar patch em modulo unico com nome explicito
3. registrar ticket e prazo para remocao
4. cobrir com teste que valide comportamento esperado
5. evitar alterar objetos globais compartilhados

Monkey patch sem governanca e tiro de escopeta em runtime.

7.8.7 Resumo POG

Monkey Patching e poderosa, rapida e perigosa na mesma proporcao. Resolve dor imediata e pode contaminar comportamento do sistema inteiro.

No dialeto POGramador: e trocar peca de motor com o carro em movimento. Pode ate continuar andando, mas voce nunca mais confia no painel.

7.9 Incremental patching debug

A tecnica de **Incremental Patching Debug** resolve bug sem investigar causa raiz: aplica patch pequeno, testa, aplica outro patch, testa de novo, e repete ate o erro "sumir".

E um processo de tentativa e erro orientado a ansiedade.

7.9.1 Ritual de aplicacao

- a versao atual parou
- pega um zip antigo "que funcionava"
- reaplica arquivos por substituicao parcial
- sobe para homologacao
- se passar no smoke test, chama de correcao

No curto prazo, pode destravar incidente. No longo prazo, mistura estados de codigo sem rastreabilidade.

7.9.2 Exemplo classico

```
Patch 1: trocar apenas Controller
Patch 2: voltar Repository para versao de ontem
Patch 3: copiar Utils de outro branch
Patch 4: comentar trecho suspeito
Resultado: erro principal sumiu, dois bugs novos nasceram
```

O nome "incremental" da impressao de metodo cientifico. A pratica costuma ser bricolagem emergencial.

7.9.3 O que quase nunca entra nesse fluxo

- depuracao real
- reproducao consistente do problema
- teste automatizado de regressao
- analise de impacto

Sem essas etapas, correcao vira loteria estatistica.

7.9.4 Por que isso e comum

- pressao por hotfix imediato
- sistema sem observabilidade
- equipe sem ambiente reproduzivel
- cultura de apagar incendio e seguir

A tecnica nao surge de incompetencia individual. Surge de contexto tecnico desorganizado.

7.9.5 Exemplo didatico

7.9.5.1 Versao POG

```
// "corrige" null pointer sem entender origem
if (cliente == null) {
    cliente = new Cliente();
}
```

Esse patch elimina a excecao localmente, mas pode mascarar falha de integracao que deveria impedir o fluxo.

7.9.5.2 Versao mais segura

```
if (cliente == null) {
    throw new RegraDeNegocioException("Cliente
    ↪ obrigatorio para concluir pedido");
}
```

E junto disso:

- reproduzir cenario em teste
- investigar por que `cliente` veio nulo
- corrigir na origem

7.9.6 Risco acumulado

- codigo vira mosaico de remendos
- regressao silenciosa cresce
- conhecimento do sistema fica tribal
- cada novo patch aumenta medo de mudar

Quando o time diz “nao encosta nisso que pode piorar”, o incremental patching ja virou cultura.

7.9.7 Como evoluir sem parar entrega

1. manter hotfix emergencial quando necessario
2. abrir tarefa obrigatoria de causa raiz apos incidente
3. registrar testes de regressao para o bug corrigido
4. reduzir area de patch com observabilidade (logs, metricas, tracing)

Assim voce preserva velocidade operacional sem normalizar gambiarra perpetua.

7.9.8 Resumo POG

Incremental Patching Debug e curativo util para sangramento imediato. O erro esta em chamar curativo de tratamento definitivo.

No glossario POGramador: e consertar encanamento com fita isolante em camadas progressivas e medir sucesso pelo tempo ate o proximo vazamento.

7.10 My precious

A tecnica **My Precious** estabelece propriedade emocional de codigo: “esse modulo e meu, so eu mexo”. O objetivo oculto e manter controle absoluto sobre um trecho critico e, por tabela, sobre o fluxo de trabalho da equipe.

7.10.1 Sinais classicos

- apenas uma pessoa aprova PR daquele modulo
- qualquer alteracao exige consulta ao “dono”
- documentacao minima, contexto maximo na cabeca de alguem
- incidentes resolvidos por chamada direta para a mesma pessoa

Em estado avancado, o codigo nao pertence ao produto. Pertence ao guardiao.

7.10.2 Por que isso acontece

- historico de sistema criado por uma pessoa so
- falta de padrao de compartilhamento de conhecimento
- inseguranca tecnica (medo de “estragarem” o que funciona)
- reconhecimento organizacional baseado em dependencia

My Precious nao e so tecnica de codigo. E dinamica de poder tecnico.

7.10.3 Exemplo do efeito colateral

```
Dev A entra de ferias -> modulo de faturamento para
Dev A adoece -> release adiado
Dev A sai da empresa -> time abre 17 chamados de emergencia
```

Quando continuidade depende de uma unica pessoa, o risco do negocio ja esta materializado.

7.10.4 Exemplo didatico de comportamento

7.10.4.1 Versao My Precious

```
// Classe enorme sem testes
public class FechamentoMensalService {
    // "nao mexer sem falar comigo"
}
```

7.10.4.2 Versao colaborativa minima

- testes cobrindo fluxos principais
- revisao em par para mudancas criticas
- README do modulo com regras e pontos de atencao
- rotacao de ownership em tarefas relevantes

Codigo compartilhado reduz dependencia sem eliminar responsabilidade.

7.10.5 O mito da protecao

A justificativa comum e “se muita gente mexer, vai quebrar”. Na realidade, isolamento sem transparencia costuma piorar:

- bug permanece escondido
- melhoria fica represada
- onboarding nao evolui
- qualidade cai quando o dono nao esta disponivel

Controle individual da uma sensacao de ordem. Colaboracao disciplinada entrega resiliencia real.

7.10.6 Como desmontar o padrao sem conflito

1. mapear modulos com ownership concentrado
2. criar pareamento tecnico nas manutencoes criticas
3. exigir testes para mudancas de alto risco
4. distribuir gradualmente revisao e sustentacao

5. reconhecer colaboracao, nao apenas heroismo individual
   Mudanca cultural e incremental, mas precisa ser intencional.

7.10.7 Resumo POG

My Precious protege ego no curto prazo e fragiliza sistema no longo. O projeto fica refem de disponibilidade humana, nao de processo tecnico.

No idioma POGramador: e guardar o anel no bolso e chamar isso de estrategia de governanca de software.

7.11 Psychoding

Psychoding e a tecnica espiritual da POG: voce nao sabe como resolver, entao abre o navegador, entra em transe de busca, copia blocos de codigo de fontes aleatorias e monta uma solucao por intuicao.

Nao e estudo. E incorporacao tecnica.

7.11.1 Etapas do transe

1. abre o Google com desespero sincero
2. cai em forum, gist, post antigo e resposta sem contexto
3. copia o trecho que “parece igual”
4. ajusta ate compilar
5. agradece aos deuses quando passa em homologacao

A mente chama isso de produtividade. O repositorio chama isso de risco latente.

7.11.2 Exemplo classico

```
// trecho copiado sem entender impacto
SimpleDateFormat sdf = new
↪  SimpleDateFormat("YYYY-MM-dd");
String data = sdf.format(new Date());
```

Funciona “na maioria dos dias”. Em virada de ano, YYYY pode gerar comportamento inesperado porque representa semana-ano em certos contextos, nao ano calendario.

7.11.3 Por que Psychoding pega tao facil

- prazo agressivo
- baixa cultura de aprofundamento
- excesso de confianca em snippet pronto
- recompensa imediata por “fazer funcionar”

Copiar e colar nao e pecado em si. O problema e nao validar premissas e nao compreender o que foi trazido.

7.11.4 Sinais de que a tecnica virou rotina

- codigo com estilos inconsistentes dentro do mesmo metodo
- dependencias adicionadas sem justificativa
- solucoes com API deprecated ou insegura
- time que nao consegue explicar por que algo foi implementado daquele jeito

Quando a explicacao oficial e “peguei no Stack Overflow”, falta camada de engenharia.

7.11.5 Exemplo didatico de uso consciente

7.11.5.1 Versao POG

```
// copiar, ajustar, subir
Pattern p = Pattern.compile("(.*)");
```

7.11.5.2 Versao responsavel

```
// 1) entender o problema
// 2) escolher abordagem
// 3) validar com testes
Pattern p = Pattern.compile("^[A-Z0-9]{8}$");
boolean valido = p.matcher(codigo).matches();
```

Diferenca principal: intencao explicita e verificavel.

7.11.6 Como aproveitar pesquisa sem cair em Psychoding

- tratar snippet como referencia, nao como produto final
- ler documentacao oficial da API usada

- escrever teste para casos limite
- registrar por que a solucao foi escolhida

Assim voce usa inteligencia coletiva sem terceirizar entendimento.

7.11.7 Risco de longo prazo

- base incoerente e dificil de manter
- vulnerabilidades por codigo copiado sem auditoria
- efeito “torre de babel” entre modulos
- dependencia de sorte para incidentes nao acontecerem

Psychoding gera entrega rapida, mas cobra pedagio tecnico crescente.

7.11.8 Resumo POG

Psychoding e mediunidade aplicada ao backlog: incorpora codigo de terceiros e espera que os espiritos da producao colaborem.

No evangelho POGramador: pesquisar e necessario, mas compreender e opcional so ate a primeira madrugada de incidente.

PARTE III

Gambi Design Patterns

Capítulo 8

Gambi Design Patterns

Depois de entender os principios e dominar as tecnicas, chegamos ao ponto em que a POG finalmente ganha forma visivel no codigo.

Bem-vindo ao catalogo dos **Gambi Design Patterns (GDPs)**.

8.1 O que sao Gambi Design Patterns

Sao padroes recorrentes de implementacao improvisada que aparecem em projetos de software sob pressao, com contexto incompleto e prazos irresponsaveis.

Um GDP nao e um bug isolado. E um comportamento arquitetural repetido.

Quando o mesmo tipo de remendo aparece em sistemas diferentes, linguagens diferentes e equipes diferentes, estamos diante de um pattern.

8.2 Por que catalogar a desgracenca

Catalogar GDPs tem tres utilidades reais:

1. **Nomear o problema** Se voce consegue nomear, voce consegue discutir com clareza.

2. **Reconhecer cedo** Padrao identificado cedo custa menos para conter.
3. **Ensinar sem moralismo** Todo mundo ja fez pog. O objetivo aqui e entendimento, nao tribunal.

Assim como os design patterns classicos documentam solucoes elegantes, os GDPs documentam solucoes pragmaticas de alto potencial radioativo.

8.3 Estrutura dos capitulos desta secao

Cada GDP foi escrito para responder quatro perguntas:

- como ele nasce
- como reconhecer no codigo
- por que ele parece uma boa ideia no curto prazo
- qual divida ele deixa no medio/longo prazo

Essa abordagem evita simplificacao infantil do tipo “isso e certo” vs “isso e errado”. Em software real, quase tudo e trade-off. A POG so deixa os trade-offs mais caros e mais rapidos.

8.4 Do accidental para o institucional

Um ponto importante: o primeiro uso de um GDP geralmente e acidental. O problema comeca quando a equipe institucionaliza o padrao:

- documenta como “jeito da casa”
- replica entre modulos
- normaliza como cultura de entrega

Nesse momento, o pattern deixa de ser excecao e vira metodo operacional.

8.5 Relacao com Tecnicas e Principios

Se os Principios sao os valores e as Tecnicas sao os rituais, os GDPs sao os artefatos finais da invocacao.

Em linguagem simples:

- principio orienta a decisao
- tecnica executa a decisao
- pattern expoe o resultado no codigo

Por isso, esta secao e a mais concreta do livro: aqui a teoria vira classe, metodo, endpoint, trigger, script e trauma de producao.

8.6 Uma nota de honestidade

Voce vai encontrar, nos proximos capitulos, patterns que talvez existam hoje no seu projeto.

Nao se culpe. Nao negue. Nao abra uma task de refatoracao total para segunda-feira.

Faça o que um POGramador lucido faz:

1. reconheca
2. priorize
3. mitigue
4. evolua sem quebrar tudo

8.7 Encerramento da abertura

Os Gambi Design Patterns sao um espelho da engenharia sob pressao. Eles revelam menos sobre linguagem e framework, e mais sobre contexto, processo e comportamento humano.

Nos capitulos seguintes, voce vai rir, se identificar, ficar levemente desconfortavel e, com sorte, sair com mais criterio para decidir quando improvisar e quando segurar a marreta.

Comecemos o catalogo da desgracenca.

8.8 WTF / WTH / QPE

O **WTF / WTH / QPE** e o padrao do trecho inexplicavel que “funciona” e, justamente por isso, ninguem tem coragem de tocar. Ele nasce de acumulacao de microajustes sem modelo mental claro.

8.8.1 A assinatura da entidade

```
"/ .*?<  ".replaceAll("", "").trim();
```

Voce le, pisca, respira fundo e pensa: “QPE e essa porra?”.

8.8.2 Como esse padrao aparece

- regex sem explicacao de intencao
- cadeia de transformacoes opacas (`replace`, `substring`, `split`) em sequencia
- condicoes com dupla negacao e sem nome intermediario
- codigo que depende de ordem acidental de operacoes

Em geral, o autor resolveu um bug real. O problema e que o conserto ficou sem contexto e sem contrato testavel.

8.8.3 Causa tipica

- hotfix de emergencia
- copia de snippet sem entendimento completo
- falta de testes de comportamento
- ausencia de revisao semantica

No dia da entrega, passa. Na sprint seguinte, vira area proibida.

8.8.4 Exemplo didatico

8.8.4.1 Versao POG

```
String out = entrada
    .replace("--", "")
    .replaceAll("[\\s]+", " ")
    .replace(" ;", ";")
    .trim();
```

Sem contexto, ninguem sabe quais casos a regra cobre.

8.8.4.2 Versao explicita

```
public String normalizarComando(String entrada) {
    String semComentario =
    ↪  removerComentarioInline(entrada);
    String espacosNormalizados =
    ↪  normalizarEspacos(semComentario);
    return normalizarSeparadores(espacosNormalizados);
}

private String removerComentarioInline(String texto) {
    // remove tudo apos "--"
    int idx = texto.indexOf("--");
    return idx >= 0 ? texto.substring(0, idx) : texto;
}
```

Aqui o comportamento fica nomeado por intencao. Se mudar regra, voce sabe onde alterar.

8.8.5 Como evitar o efeito “codigo magico”

- nomear subpassos com semantica de negocio
- adicionar testes com exemplos reais de entrada/saida

- documentar limites da regra (o que nao cobre)
- preferir clareza a “one-liner genial”

Codigos curtos nao sao automaticamente bons. Codigos entendiveis sao.

8.8.6 O perigo social do QPE

Trecho opaco cria dependencia pessoal. So quem escreveu “entende”. Isso vira gargalo humano e risco de continuidade.

Quando equipe evita mexer por medo, o software para de evoluir com seguranca.

8.8.7 Correcao pragmatica

1. escolher um trecho QPE de alto impacto
2. escrever testes de comportamento atual
3. refatorar para passos nomeados
4. manter resultado identico e reduzir opacidade

Assim voce melhora entendimento sem alterar regra de negocio no susto.

8.8.8 Resumo POG

WTF/WTH/QPE e o ponto onde codigo deixa de ser comunicacao e vira feitico. Pode funcionar anos, mas cobra caro em manutencao e transferencia de contexto.

Na gramatica POGramadora: quando a explicacao de um trecho comeca com “nao me pergunte”, ja estamos no dominio do QPE.

8.9 RCP Pattern (Reuse by Copy and Paste)

O **RCP Pattern** (Reuse by Copy and Paste) e o coracao industrial da POG. A regra e objetiva: se um trecho resolveu um problema, multiplique ele sem pudor.

Ctrl+C e Ctrl+V viram framework de produtividade.

8.9.1 Principio da Reflexao Reprodutoria

A logica e quase poetica:

- copiar acelera entrega
- adaptar “na unha” parece barato
- cada copia vira uma variante do original

No inicio, a equipe sente ganho real de velocidade. Depois, cada alteracao exige cacar todas as duplicacoes, e sempre sobra uma esquecida.

8.9.2 Exemplo didatico

```
1  // Modulo A
2  if (usuario == null ||
   ↪  usuario.getStatus().equals("INATIVO")) {
3      throw new RegraDeNegocioException("Usuario
       ↪  invalido");
4  }
5
6  // Modulo B (copiado e colado)
7  if (usuario == null ||
   ↪  usuario.getStatus().equals("INATIVO")) {
8      throw new RegraDeNegocioException("Usuario
       ↪  invalido");
9  }
10
11 // Modulo C (copiado e "adaptado")
```

```
12  if (usuario == null ||
    ↪  usuario.getStatus().equals("INATIVO") ||
    ↪  usuario.isBloqueado()) {
13      throw new RegraDeNegocioException("Usuario
        ↪  invalido");
14  }
```

Quando a regra muda, A e B atualizam. C fica diferente. Surge bug “aleatorio” por divergencia de comportamento.

8.9.3 Smells associados

- duplicacao de codigo
- shotgun surgery (uma mudanca, muitos arquivos)
- incoerencia de regra entre fluxos “parecidos”
- testes repetitivos cobrindo variacoes acidentais

Esse padrao costuma ser invisivel no code review rapido, porque cada trecho isolado “faz sentido”. O problema esta na soma.

8.9.4 Por que times caem nisso

- backlog pressionando por throughput
- ausencia de componentes reutilizaveis simples
- medo de refatorar codigo compartilhado e quebrar legado
- cultura de “depois a gente organiza”

No contexto certo, copiar e colar e uma decisao taticamente racional. O erro e transformar tatica emergencial em estrategia permanente.

8.9.5 Evolucao didatica

8.9.5.1 Versao com copia

```
1  // regra repetida em varios lugares
2  if (pedido == null || pedido.getItens().isEmpty()) {
```

```
    throw new RegraDeNegocioException("Pedido invalido");
}
```

8.9.5.2 Versao com encapsulamento minimo

```
public final class ValidadorPedido {
    public static void validar(Pedido pedido) {
        if (pedido == null ||
        ↪  pedido.getItens().isEmpty()) {
            throw new RegraDeNegocioException("Pedido
            ↪  invalido");
        }
    }
}

// uso
ValidadorPedido.validar(pedido);
```

Agora a regra tem dono unico. Mudou uma vez, mudou para todos.

8.9.6 Estrategia pratica para legado

1. medir duplicacao dos trechos criticos
2. criar utilitario/servico pequeno para regra comum
3. migrar usos aos poucos (por modulo)
4. cobrir com testes de contrato

Sem “big bang”. Sem promessa heroica.

8.9.7 Resumo POG

RCP e maravilhoso para nascer software rapido e produzir variacoes criativas de bug. Em projetos longos, vira multiplicador de custo de manutencao.

No dicionario POGramador: e clonar problema em alta disponibilidade para garantir demanda futura da sustentacao.

8.10 Hardcoded Data

No **Hardcoded Data**, dado de configuracao, regra de negocio e detalhe de ambiente sao colocados diretamente no codigo-fonte. O mantra e simples: “se esta no codigo, eu sei onde esta”.

O problema e que o codigo vira ao mesmo tempo executavel, banco de parametros e painel operacional.

8.10.1 Exemplo classico

```
// Xunxa o nome da impressora no codigo. Quem quer escolher impressora?
infoImpressao = ImpressaoUtils.getInfoImpressao(codigoRelatorio, "PADRAO");
```

Hoje e o nome da impressora. Amanha e URL de servico, aliquota fiscal, chave de parceiro e data de corte. Em poucas sprints, o deploy vira painel de configuracao manual.

8.10.2 Sinais de que o padrao tomou conta

- strings magicas repetidas em varias classes
- alteracao de regra operacional exigindo merge + pipeline
- ambiente homolog/producao diferenciados por `if (isProd)`
- chamados de negocio resolvidos com “vamos subir patch”

Quando mudar um texto de mensagem exige release, o Hardcoded Data venceu.

8.10.3 Por que ele aparece

- pressa para colocar funcionalidade no ar
- falta de estrategia de configuracao por ambiente
- legado sem centralizacao de parametros

- medo de criar tabela/config store “mais uma vez”

No curto prazo, parece pratico. No longo, todo ajuste vira risco de regressao funcional.

8.10.4 Exemplo didatico de evolucao

8.10.4.1 Versao POG

```
public void emitirRelatorio() {
    String impressora = "PADRAO";
    String endpoint = "https://api.parceiro.com/v1";
    int timeout = 30;
    // ...
}
```

8.10.4.2 Versao com configuracao explicita

```
public class ConfiguracaoRelatorio {
    private final String impressoraPadrao;
    private final String endpointParceiro;
    private final int timeoutSegundos;

    public ConfiguracaoRelatorio(String impressoraPadrao,
    ↪  String endpointParceiro, int timeoutSegundos) {
        this.impressoraPadrao = impressoraPadrao;
        this.endpointParceiro = endpointParceiro;
        this.timeoutSegundos = timeoutSegundos;
    }

    public String getImpressoraPadrao() { return
    ↪  impressoraPadrao; }
    public String getEndpointParceiro() { return
    ↪  endpointParceiro; }
    public int getTimeoutSegundos() { return
    ↪  timeoutSegundos; }
}
```

```
16
17  public void emitirRelatorio(ConfiguracaoRelatorio cfg) {
18      // usa cfg sem chutar valor em runtime
19  }
```

A regra sai do codigo e vai para contrato de configuracao. Resultado: menos release de emergencia para ajuste operacional.

8.10.5 Impactos de negocio

- time de produto depende de dev para mudar qualquer parametro
- incidentes aumentam por ajustes urgentes em horario critico
- rollback de versao pode desfazer configuracoes validas
- auditoria fica fraca (quem mudou o que e quando?)

8.10.6 Correcao sem trauma

1. mapear constantes criticas (URL, timeout, codigos de regra)
2. extrair para configuracao externa versionada
3. manter default seguro apenas onde fizer sentido
4. adicionar validacao na inicializacao do sistema

Assim voce reduz acoplamento sem parar a entrega.

8.10.7 Resumo POG

Hardcoded Data e a forma mais rapida de transformar deploy em ferramenta administrativa. Funciona enquanto o sistema e pequeno. Quando cresce, vira gargalo organizacional.

No linguajar POGristico: e tatuar instrucoes operacionais no corpo do programa e fingir surpresa quando mudar de ideia doi.

8.11 Forceps

O **Forceps** e o padrao obstetrico da POG. Ele aparece quando uma variavel nao recebe o valor esperado e, em vez de investigar causa raiz, o POGramador "puxa" o valor correto no ponto de uso.

Em termos praticos, e a arte de corrigir o sintoma localmente para manter o fluxo vivo. Funciona hoje. Custa caro amanha.

8.11.1 Exemplo classico

```
/* Variavel e inicializada */
String valor = "123";

/* ... logica do programa ... */

/* Dentro de um metodo que utiliza a variavel 'valor' */
if (!"123".equals(valor)) {
    valor = "123";
    processaValor(valor);
}
```

O trecho parece inocente. Mas repare no que ele comunica: "se veio errado, conserta aqui mesmo". Isso cria uma blindagem local que mascara o defeito real do fluxo.

8.11.2 Como reconhecer o Forceps no codigo

- verificacoes redundantes do mesmo valor em varios pontos
- atribuicoes "defensivas" copiadas entre metodos
- comentarios tipo "garantia extra para evitar bug intermitente"
- logica de negocio baseada em fallback manual

Quando voce encontra o mesmo `if` em cinco classes diferentes, ja existe um ritual de Forceps consolidado.

8.11.3 Por que o time adota isso

Motivos reais:

- bug em producao sem tempo para investigacao profunda
- desconhecimento do fluxo completo em legado grande
- medo de tocar na origem e quebrar outras telas
- cultura de apagar incendio primeiro e pensar depois

Ou seja, o Forceps quase nunca nasce por maldade. Ele nasce por sobrevivencia operacional.

8.11.4 Impactos no medio prazo

- causa raiz segue ativa
- inconsistencias se espalham em silencio
- manutencao fica confusa (qual ponto esta “corrigindo” o que?)
- testes passam sem garantir consistencia global

No fim, o sistema vira uma colcha de microcorrecoes. Cada parte se protege da outra.

8.11.5 Exemplo didatico de abordagem melhor

```
1  public class PedidoService {
2
3      public void processar(Pedido pedido) {
4          String codigo =
           ↪  normalizarCodigo(pedido.getCodigo());
5          validarCodigo(codigo);
6          pedido.setCodigo(codigo);
7          repositorio.salvar(pedido);
8      }
9
10     private String normalizarCodigo(String codigo) {
11         if (codigo == null) {
12             return "123"; // regra explicita e
               ↪  centralizada
13         }
```

```
        return codigo.trim();
    }

    private void validarCodigo(String codigo) {
        if (!"123".equals(codigo)) {
            throw new RegraDeNegocioException("Codigo
            ↪  invalido para este fluxo");
        }
    }
}
```

Aqui, a regra fica centralizada. Se a origem estiver ruim, voce tem erro claro para tratar no ponto certo, em vez de remendo espalhado.

8.11.6 Estrategia pragmatica de correcao

1. mapear onde o valor esta sendo forçado
2. eleger um unico ponto de normalizacao
3. adicionar teste de contrato para entrada/saida
4. remover os Forceps duplicados aos poucos

Isso evita refatoracao heroica e reduz risco de regressao.

8.11.7 Resumo POG

Forceps e excelente para entregar hoje e manter o chamado fechado. Mas ele nao resolve defeito sistemico; apenas empurra o problema para frente com juros.

No dialeto POGrames: e um parto feito no corredor. A crianca nasce, mas o prontuario vira lenda urbana dentro do repositorio.

8.12 Ostrich Syndrome Skill

O **Ostrich Syndrome Skill** e a habilidade de enterrar a cabeca tecnicamente: warning, deprecacao e alerta de analise estatica sao tratados como ruido de fundo.

A filosofia e ancestral:

- o que os olhos nao veem, o backlog nao sente
- se compila, ta pronto
- warning e ciume da IDE

8.12.1 Forma ritualistica

```
@SuppressWarnings("all")
public class ProcessadorLegado {
    // aqui jaz a paz de espirito da equipe
}
```

Esse artefato da tranquilidade elimina alertas visiveis, mas nao elimina risco real.

8.12.2 Sinais no projeto

- dezenas de supressoes globais sem justificativa
- upgrade de dependencia sempre adiado porque “vai quebrar tudo”
- build verde com log amarelo infinito
- regra de review: “nao mexe nisso agora”

Quando warning vira paisagem, defeito vira surpresa.

8.12.3 Por que acontece

Motivos praticos:

- pressao por entrega imediata
- base legada muito ruidosa

- pouca maturidade de observabilidade
- medo de abrir frente tecnica sem patrocinio

Ignorar alerta pode ser decisao temporaria legitima. O problema e quando temporario vira dogma.

8.12.4 Exemplo didatico

8.12.4.1 Versao POG

```
@SuppressWarnings("deprecation")
public void salvar(Data data) {
    repositorioAntigo.save(data); // API descontinuada ha
    ↪ anos
}
```

8.12.4.2 Versao com controle

```
public void salvar(Data data) {
    // TODO(POG-123): migrar para NovoRepositorio ate
    ↪ 2026-06-30
    repositorioAntigo.save(data);
}
```

Melhor ainda:

```
public void salvar(Data data) {
    if (featureFlags.usarNovoRepositorio()) {
        novoRepositorio.save(data);
        return;
    }
    repositorioAntigo.save(data);
}
```

Nesse formato, alerta vira plano. Nao e so silenciamento.

8.12.5 Risco acumulado

- vulnerabilidade de dependencia desatualizada
- comportamento removido em upgrade futuro
- dificuldade de onboarding (ninguem sabe o que pode quebrar)
- incidentes em cadeia quando enfim chega a migracao

8.12.6 Como tratar sem paralisar entrega

1. classificar warning por severidade
2. criar “orcamento de warning” por sprint
3. proibir novas supressoes globais
4. exigir comentario com ticket e prazo ao suprimir
5. priorizar deprecacoes em codigo mais usado

Isso reduz ruido progressivamente sem exigir limpeza total imediata.

8.12.7 Resumo POG

Ostrich Syndrome Skill da alivio emocional no curto prazo e ansiedade tecnica no longo. Silenciar alerta e facil. Gerenciar consequencia, nem tanto.

No evangelho POGrames: enterramos a cabeca para nao ver o problema, e depois abrimos incidente para descobrir por que ele cresceu no escuro.

8.12.8 Mini checklist de mitigacao

Toda supressao de warning deve trazer justificativa tecnica e prazo para revisao. Se nao houver ticket, dono e data, nao e supressao estrategica: e abandono controlado. A diferenca entre pragmatismo e negligencia esta na rastreabilidade da decisao.

Esse controle evita que o warning vire folklore tecnico.

8.13 Nonsense Flag Nonsense Naming

O **Nonsense Flag Nonsense Naming** transforma nomeacao em criptografia artesanal. Variaveis nao explicam intencao; elas insinuam, confundem e exigem mediunidade de quem le.

```
teste1, temp2, a, b, x
jaTrocouDeAba, botaoClicado, foiAtualizado, passouPorAqui
numeroMagico, naoAchou, temErro
anterior5, atual5, anteriorDoAnterior5
```

Esse padrao costuma vir acompanhado de flags booleans caoticas (`isOk`, `isReady2`, `podeTalvez`), criando fluxo de decisao que parece enquete de rede social.

8.13.1 Efeito semantico

Nome ruim nao e so “feio”. Ele altera custo cognitivo:

- leitura fica lenta
- regra de negocio vira adivinhacao
- review perde profundidade
- bug de entendimento aumenta

Quando o codigo exige reuniao para explicar cada variavel, a manutencao ja quebrou.

8.13.2 Exemplo didatico

8.13.2.1 Versao POG

```
if (a && !b && x > 0) {
    faz1();
} else if (a && b && x == 0) {
    faz2();
}
```

8.13.2.2 Versao legivel

```java
boolean clienteElegivel = cliente.estaAtivo();
boolean pedidoJaFaturado = pedido.isFaturado();
int quantidadeItens = pedido.getItens().size();

if (clienteElegivel && !pedidoJaFaturado &&
    quantidadeItens > 0) {
    gerarFatura();
} else if (clienteElegivel && pedidoJaFaturado &&
    quantidadeItens == 0) {
    registrarInconsistencia();
}
```

A logica pode ser a mesma. A diferenca e que agora o leitor entende o dominio sem abrir 12 arquivos.

8.13.3 Por que o time cai nisso

- codigo escrito sob estresse
- falta de padrao de nomeacao
- medo de “nome grande”
- copia de variavel antiga para novo contexto

E comum em legado com baixa cobertura de teste: ninguem renomeia por receio de quebrar algo invisivel.

8.13.4 Nonsense Flag: o primo perigoso

Flags sem semantica clara criam combinacoes explosivas.

```java
if (isOk && !isReady && podeAtualizar && modo2) {
    // o que exatamente isso significa?
}
```

Cada booleano adicional dobra os estados possiveis. Sem modelagem explicita, o fluxo fica impossivel de validar mentalmente.

8.13.5 Abordagem pragmatica

1. renomear primeiro as variaveis de maior impacto
2. extrair condicoes para metodos com nome de negocio
3. substituir multiplos booleans por enum/objeto de estado
4. registrar convencoes simples de nomeacao no time

Pequenas mudancas de semantica trazem ganho real sem refatoracao monstruosa.

8.13.6 Resumo POG

Nonsense Naming e Nonsense Flag dao sensacao de velocidade na digitacao e cobram pedagio eterno na leitura. O sistema roda, mas o entendimento nao escala.

Na tradicao POGristica: se nem voce entende o nome da variavel depois de uma semana, o ritual foi concluido com excelencia duvidosa.

8.13.7 Mini checklist de mitigacao

Renomeacao progressiva de variavel e melhoria de baixo risco e alto retorno. Cada nome claro reduz duvida em review, onboarding e debug. Sem semantica compartilhada, a equipe conversa sobre sintaxe e nunca sobre dominio.

8.14 Commented Code Implementation Comments Forever

O **Commented Code Implementation** e o padrao em que codigo morto, codigo desativado e blocos de experimento ficam comentados para sempre no arquivo “por seguranca”.

A narrativa e conhecida: “nao apaga, vai que precisa depois”.

8.14.1 Exemplo classico

```
public void calcular() {
    // antiga regra de desconto
    // if (cliente.isPremium()) {
    //     total = total.multiply(new BigDecimal("0.8"));
    // }

    // nova regra (temporaria desde 2019)
    if (cliente.isPremium()) {
        total = total.multiply(new BigDecimal("0.85"));
    }
}
```

O comentario vira arquivo historico embutido no fonte. O problema e que historico verdadeiro ja existe: chama-se Git.

8.14.2 Problemas que esse padrao cria

- arquivo cresce com ruido sem valor executavel
- leitor nao sabe qual regra vale de fato
- revisao fica lenta, porque ha muito texto irrelevante
- chance de “descomentar” trechos obsoletos por engano

Comentario deveria explicar decisao. Nao substituir versionamento.

8.14.3 Quando isso comeca

- hotfix de madrugada com medo de perda
- ausencia de confianca em rollback
- equipe sem disciplina de branch/commit claro
- heranca de codigo antigo onde “apagar” e visto como risco

Em contexto de baixa previsibilidade, comentar parece seguro. Na pratica, so adia decisao tecnica.

8.14.4 Exemplo didatico de alternativa

8.14.4.1 Versao POG

```
// TODO remover depois
// chamadaServicoAntigo();
chamadaServicoNovo();

// if (featureX) {
//   fluxoVelho();
// }
```

8.14.4.2 Versao controlada

```
if (featureFlags.usarFluxoNovo()) {
    chamadaServicoNovo();
} else {
    chamadaServicoAntigo();
}
```

Com feature flag, o comportamento fica explicito e rastreavel. Quando migrar tudo, remove-se o fluxo antigo com commit unico e mensagem clara.

8.14.5 Comentario bom x comentario ruim

Comentario bom:

- registra contexto de negocio ou decisao arquitetural
- explica “por que” algo existe
- aponta ticket/issue quando ha debito tecnico assumido

Comentario ruim:

- replica o que o codigo ja diz
- guarda codigo morto
- serve de escudo para incerteza eterna

8.14.6 Estrategia pragmatica de limpeza

1. remover blocos comentados sem uso comprovado
2. migrar excecoes para tickets rastreaveis
3. usar feature flag para transicao real
4. adotar regra de review: codigo comentado executavel nao entra

Isso reduz ruido sem interromper entrega.

8.14.7 Resumo POG

Commented Code Forever e um museu de decisao incompleta. Parece prudente, mas degrada legibilidade e aumenta risco operacional.

Em modo POGramador: e guardar peças de carro velho na sala para “eventual necessidade” e chamar isso de estrategia de manutencao preventiva.

8.14.8 Mini checklist de mitigacao

Codigo morto deve sair do arquivo e ficar no historico do Git. Comentario bom explica decisao; comentario ruim armazena medo. Se o trecho precisa existir por transicao, feature flag com prazo e opcao mais segura.

8.15 Reinvented Square Wheel Helper

O **Reinvented Square Wheel Helper** e o padrao de reimplementar manualmente algo que a linguagem, framework ou biblioteca ja fornece com qualidade melhor.

A motivacao costuma ser nobre: “quero controle total”. O resultado, quase sempre, e uma roda quadrada de manutencao pesada.

8.15.1 Exemplo classico

```
if (number.equals("1")) {
    return 1;
} else if (number.equals("2")) {
    return 2;
} else if (number.equals("3")) {
    return 3;
} else if (number.equals("4")) {
    return 4;
} else if (number.equals("5")) {
    return 5;
} // ... ate o infinito
```

Aqui, algo que poderia ser `Integer.parseInt(number)` vira cascata manual sujeita a erro, inconsistencia e custo de manutencao absurdo.

8.15.2 Sintomas do padrao

- helpers enormes para funcao basica
- “framework interno” para resolver problema trivial
- implementacoes caseiras sem teste robusto
- divergencia entre comportamento esperado e padrao de mercado

Quando o time escreve parser de data na mao em projeto Java moderno, a roda quadrada ja esta em producao.

8.15.3 Por que isso acontece

- desconhecimento de recurso nativo
- trauma com biblioteca antiga
- desconfiança de dependencia externa
- ego tecnico (“eu faco melhor”)

Nem sempre e vaidade. Muitas vezes e falta de repertorio compartilhado no time.

8.15.4 Exemplo didatico

8.15.4.1 Versao POG

```
public boolean isEmailValido(String email) {
    if (email == null) return false;
    if (!email.contains("@")) return false;
    if (!email.contains(".")) return false;
    if (email.startsWith("@")) return false;
    // dezenas de regras incompletas...
    return true;
}
```

8.15.4.2 Versao mais segura

```
public boolean isEmailValido(String email) {
    if (email == null) return false;
    return javax.mail.internet.InternetAddress
        .parse(email, true)
        .length == 1;
}
```

Voce delega para implementacao madura, reduz bug e foca na regra de negocio real.

8.15.5 Custo oculto

- aumento de superficie de bug
- onboarding lento (aprender ferramentas internas desnecessarias)
- dificuldade de evolucao (cada helper caseiro vira dependente de contexto)
- retrabalho em manutencao corretiva

Em resumo: mais codigo para manter sem ganho proporcional de valor.

8.15.6 Correcao pragmatica

1. identificar helpers caseiros de alto risco
2. comparar com API nativa equivalente
3. migrar gradualmente com testes de comportamento
4. documentar quando realmente precisar de implementacao propria

Se houver requisito especifico legitimo, mantenha customizacao minima e justificada.

8.15.7 Resumo POG

Reinvented Square Wheel Helper e o orgulho de construir do zero o que ja existe pronto. Da sensacao de autoria e traz manutencao vitalicia.

No vocabulário POGristico: e trocar elevador por escada rolante movida a manivela para provar independencia tecnologica.

8.15.8 Mini checklist de mitigacao

Antes de criar helper caseiro, responda: existe API nativa madura para isso? Se existir, o onus da prova e de quem quer reinventar. Em geral, software de negocio ganha mais quando reutiliza base estavel.

8.16 You Shall Not Pass

O **You Shall Not Pass** é o padrão de captura total: tudo é envolvido por `try/catch` amplo, normalmente com `Exception` ou `Throwable`, para garantir que nada "escape".

A intenção parece nobre: proteger o sistema. O efeito real costuma ser o oposto: esconder causa raiz, diluir contexto e dificultar manutenção.

8.16.1 Sintoma clássico

```
public String processar(String entrada) {
    try {
        return servicoA.executar(entrada);
    } catch (Throwable t) {
        return "Falha ao processar";
    }
}
```

Nesse modelo, falhas completamente diferentes viram a mesma resposta:

- erro de validação
- timeout de rede
- bug de programação
- erro de banco
- bug de serialização

Tudo cai no mesmo balaio sem rastreabilidade adequada.

8.16.2 Por que isso é perigoso

Capturar `Throwable` é especialmente arriscado porque inclui `Error` (ex.: `OutOfMemoryError`), que em geral não deveria ser "tratado" como fluxo comum da aplicação.

Quando o código captura amplo demais:

- o sistema parece estável, mas está cego
- logs úteis somem
- retries automáticos podem repetir operações perigosas
- estado inconsistente pode continuar rodando sem alerta

É o equivalente operacional de desligar o alarme de incêndio porque ele faz barulho.

8.16.3 Exemplo didático (controle de granularidade)

8.16.3.1 Versão POG

```
public Resultado gerarRelatorio(Filtro filtro) {
    try {
        validar(filtro);
        Dados dados = repositorio.buscar(filtro);
        byte[] pdf = renderizador.gerarPdf(dados);
        return Resultado.ok(pdf);
    } catch (Exception e) {
        return Resultado.erro("Não foi possível gerar
        ↪  relatório");
    }
}
```

8.16.3.2 Versão com tratamento útil

```
public Resultado gerarRelatorio(Filtro filtro) {
    try {
        validar(filtro);
    } catch (ValidacaoException e) {
        return Resultado.erro("Filtro inválido: " +
        ↪  e.getMessage());
    }

    Dados dados;
    try {
```

```
        dados = repositorio.buscar(filtro);
    } catch (DataAccessException e) {
        logger.error("Falha no banco ao buscar relatório", e);
        return Resultado.erro("Falha temporária ao consultar dados");
    }

    try {
        byte[] pdf = renderizador.gerarPdf(dados);
        return Resultado.ok(pdf);
    } catch (RenderizacaoException e) {
        logger.error("Falha ao renderizar PDF", e);
        return Resultado.erro("Não foi possível gerar o arquivo PDF");
    }
}
```

Aqui cada tipo de problema recebe:

- tratamento adequado
- mensagem correta
- log contextualizado

8.16.4 Quando usar captura ampla, então?

Existe um uso legítimo: fronteiras globais de aplicação (filtro HTTP, middleware, handler global), para evitar queda abrupta e registrar erro inesperado.

Mesmo nesses casos:

- capture para registrar e encerrar com segurança
- não converta tudo em “deu ruim” sem contexto
- não continue fluxo normal após falha crítica

8.16.5 Estratégia de correção gradual

Se seu legado está dominado por `catch` genérico:

1. mapeie os pontos com maior volume de erro
2. substitua captura genérica por exceções específicas
3. adicione logs com contexto de negócio (id, operação, usuário)
4. padronize respostas por categoria de erro
5. mantenha fallback global para o que for realmente inesperado

Essa abordagem reduz risco sem parar o trem.

8.16.6 Resumo POG

You Shall Not Pass nasce da boa intenção de blindar o sistema, mas frequentemente vira blindagem contra diagnóstico. O código até "não quebra" na frente do usuário, porém quebra a capacidade do time de entender e corrigir problemas.

No fim, erro que não aparece não desaparece. Ele só muda de lugar: sai da tela e vai morar no backlog eterno da sustentação.

8.17 Perfectness Execution Bulletproof

O **Bulletproof** é o padrão em que toda operação, independentemente do que aconteça, termina com mensagem de sucesso. Deu certo? Sucesso. Deu errado? Sucesso também. Explodiu? Sucesso com fé.

```
try {
    if (alterar(valor1, valor2)) {
        return new Mensagem("Operação concluída com
        ↪ sucesso!");
    } else {
        return new Mensagem("Operação concluída com
        ↪ sucesso!");
    }
} catch (Throwable e) {
    return new Mensagem("Operação concluída com
    ↪ sucesso!");
}
```

Na superfície, parece experiência positiva para o usuário. No fundo, é supressão sistemática da realidade.

8.17.1 Como esse padrão se instala

Ele costuma surgir quando o time sofre pressão por indicadores simplistas, tipo:

- “não pode aparecer erro para o usuário”
- “precisamos reduzir chamados”
- “a tela sempre deve retornar ok”

Em vez de melhorar validação, observabilidade e tratamento adequado, adota-se o atalho: uniformizar resposta de sucesso. O bug deixa de ser visível, mas continua existindo.

8.17.2 Exemplo didático (problema real disfarçado)

Imagine um endpoint de atualização cadastral:

```java
public Mensagem atualizarEmail(Long usuarioId, String novoEmail) {
    try {
        Usuario usuario = usuarioRepository.findById(usuarioId).orElse(null);

        if (usuario == null) {
            return new Mensagem("Operação concluída com sucesso!");
        }

        usuario.setEmail(novoEmail);
        usuarioRepository.save(usuario);

        // Se save falhar por constraint, cai no catch e também retorna sucesso.
        return new Mensagem("Operação concluída com sucesso!");
    } catch (Exception e) {
        return new Mensagem("Operação concluída com sucesso!");
    }
}
```

O usuário recebe sucesso mesmo quando:

- ID não existe
- e-mail é inválido
- banco está indisponível
- transação foi revertida

Isso sabota o ciclo de feedback da aplicação.

8.17.3 Efeito colateral em cadeia

O Bulletproof cria danos silenciosos:

- suporte não consegue reproduzir erro porque “o sistema diz que

deu certo”
- monitoramento perde sinal útil
- inconsistência de dados cresce sem alarme
- times consumidores da API tomam decisões erradas com base em falso positivo

É o equivalente a arrancar a luz do painel do carro para “resolver” o aviso do óleo.

8.17.4 Versão didática melhor (sem perder UX)

Você pode ser amigável com usuário sem mentir para ele:

```
public ResultadoAtualizacao atualizarEmail(Long usuarioId, String novoEmail) {
    if (novoEmail == null || !novoEmail.contains("@")) {
        return ResultadoAtualizacao.falha("E-mail inválido");
    }

    Usuario usuario = usuarioRepository.findById(usuarioId).orElse(null);
    if (usuario == null) {
        return ResultadoAtualizacao.falha("Usuário não encontrado");
    }

    try {
        usuario.setEmail(novoEmail);
        usuarioRepository.save(usuario);
        return ResultadoAtualizacao.sucesso("E-mail atualizado com sucesso");
    } catch (DataAccessException e) {
        // Log técnico detalhado para equipe
        logger.error("Falha ao atualizar e-mail do usuário {}", usuarioId, e);
        // Mensagem amigável para usuário
```

```
19          return ResultadoAtualizacao.falha("Não foi
            ↪  possível concluir agora. Tente novamente.");
20      }
21  }
```

Aqui você tem:

- resultado honesto
- mensagem compreensível
- log técnico para diagnóstico
- separação entre erro de negócio e erro de infraestrutura

8.17.5 Quando o Bulletproof já está em produção

Não precisa reescrever tudo de uma vez. Estratégia incremental:

1. mapear endpoints com maior taxa de chamado
2. trocar retorno único por contrato de sucesso/falha
3. manter compatibilidade externa temporária
4. instrumentar logs e métricas antes de mudar comportamento de UI
5. remover `catch` genérico com retorno otimista

8.17.6 Resumo POG

Bulletproof é a prova de bala mais famosa da POG: não impede o tiro, só apaga o buraco da parede no relatório. Ele melhora aparência de curto prazo e destrói confiança sistêmica no longo prazo.

Sistema confiável não é o que "sempre responde sucesso". É o que responde a verdade, com contexto e previsibilidade. O restante é maquiagem operacional com prazo de validade curto.

8.18 Exception Success

O **Exception Success** é o padrão em que a exceção deixa de representar situação excepcional e passa a ser usada como fluxo normal da aplicação. Em vez de retornar um resultado, o código "comunica" sucesso, validação, autorização e até regra de negócio por `throw`.

Na teoria, exceção deveria sinalizar algo fora do caminho esperado. Na prática POG, ela vira API oficial da casa.

```
public static void somar(int a, int b) {
    System.out.println(a + b);
    // POG clássica: sucesso tratado como "erro"
    throw new RuntimeException("Operação realizada com
    ↪  sucesso!");
}
```

8.18.1 Como reconhecer esse padrão

Você provavelmente está diante de um Exception Success quando vê este combo:

- métodos "felizes" que sempre terminam com `throw`
- `catch (Exception e)` decidindo regra de negócio
- mensagem de usuário final embutida em exception técnica
- sistema que "funciona" só porque alguém conhece a ordem dos `catch`

Outro sinal típico é a classe de serviço com assinatura sem retorno útil, e toda decisão sendo tomada no controlador por blocos de captura.

8.18.2 Exemplo didático (versão POG)

```
public void processarPagamento(Pagamento pagamento)
↪   throws Exception {
    if (pagamento == null) {
```

```
            throw new Exception("Pagamento inválido");
        }

        if (pagamento.getValor() <= 0) {
            throw new Exception("Valor deve ser maior que
            ↪ zero");
        }

        gateway.cobrar(pagamento);

        // "Sucesso" sinalizado por exceção para cair no
        ↪ catch correto
        throw new Exception("PAGAMENTO_OK");
    }

    public String concluir(Pagamento pagamento) {
        try {
            processarPagamento(pagamento);
            return "Fluxo inesperado"; // nunca chega aqui
        } catch (Exception e) {
            if ("PAGAMENTO_OK".equals(e.getMessage())) {
                return "Pagamento concluído";
            }
            return "Falha: " + e.getMessage();
        }
    }
```

Esse código parece “esperto” no curto prazo, porque centraliza tudo no `catch`. O problema é que mistura semânticas diferentes no mesmo canal:

- erro de infraestrutura
- erro de validação
- estado de sucesso

Quando tudo vira exceção, nada mais é exceção.

8.18.3 Por que isso aparece em projeto real

Esse padrão nasce por combinação de pressa, legado e falta de contrato claro entre camadas. É comum em contexto onde o time precisa "fazer entrar em produção hoje" e adota soluções improvisadas:

- não havia tipo de retorno definido
- o sistema já tinha muito `try/catch` espalhado
- cada dev adicionou mais um `throw` para não quebrar fluxo antigo

Também aparece como versão digital do cargo cult programming: alguém viu que um `throw` resolveu um bug específico, copiou a técnica, e passou a reproduzir o ritual sem entender o efeito colateral.

8.18.4 Impactos técnicos

Os danos costumam ser progressivos:

- observabilidade piora, porque logs ficam poluídos com "erros" que não são erros
- monitoramento dispara alerta falso
- leitura do código fica ambígua
- testes ficam frágeis, pois dependem de mensagens textuais
- qualquer internacionalização quebra regra de negócio baseada em `e.getMessage()`

Em sistemas Java, isso ainda conflita com a intenção da própria linguagem e bibliotecas, que tratam exceções como mecanismo de anomalia de execução, não como retorno padrão.

8.18.5 Exemplo didático (versão menos caótica)

```
1  public final class ResultadoPagamento {
2      private final boolean sucesso;
3      private final String mensagem;
4
5      private ResultadoPagamento(boolean sucesso, String
       ↪  mensagem) {
```

```
        this.sucesso = sucesso;
        this.mensagem = mensagem;
    }

    public static ResultadoPagamento ok(String mensagem) {
        return new ResultadoPagamento(true, mensagem);
    }

    public static ResultadoPagamento falha(String mensagem) {
        return new ResultadoPagamento(false, mensagem);
    }

    public boolean isSucesso() { return sucesso; }
    public String getMensagem() { return mensagem; }
}

public ResultadoPagamento processarPagamento(Pagamento pagamento) {
    if (pagamento == null) {
        return ResultadoPagamento.falha("Pagamento inválido");
    }

    if (pagamento.getValor() <= 0) {
        return ResultadoPagamento.falha("Valor deve ser maior que zero");
    }

    try {
        gateway.cobrar(pagamento);
        return ResultadoPagamento.ok("Pagamento concluído");
    } catch (GatewayIndisponivelException e) {
        // aqui sim: exceção realmente excepcional
        return ResultadoPagamento.falha("Gateway indisponível");
```

```
37          }
38      }
```

Perceba a diferença didática:

- fluxo de negócio usa retorno explícito
- exceção fica para falha inesperada/infraestrutura
- contrato fica legível para quem mantém depois

8.18.6 Resumo POG

Exception Success é sedutor porque parece reduzir código no início. Só que ele troca clareza por truque, e truque em software envelhece mal. Em termos gambiarrísticos, é uma técnica de "entrega imediata com juros compostos".

Se ainda existir Exception Success no seu sistema, não precisa derrubar tudo. Comece isolando os pontos críticos e separando, pouco a pouco, **resultado de negócio** de **condição excepcional**. Assim você preserva produção e reduz o caos sem ferir o GLS.

8.19 String Sushiman

No **String Sushiman**, parametros estruturados sao compactados em uma string "linguicao" com delimitadores magicos. Depois, o codigo faz `split` em camadas e torce para cada posicao vir no formato correto.

8.19.1 Exemplo classico

```
public Tabela montaTabela(String linguicao) {

    String[] colunas = linguicao.split("\\|");

    for (String coluna : colunas) {
        String[] campos = coluna.split(",");
        // POGuices com os valores
    }
}
```

Parece rapido para enviar dados sem criar contrato formal. O custo vem depois: qualquer virgula fora do lugar quebra o parsing inteiro.

8.19.2 Sinais de maturidade Sushiman

- metodo com um unico `String` recebendo tudo
- documento externo explicando "ordem dos campos" em texto livre
- erros de parse intermitentes conforme dados reais
- codigo cheio de `split`, `trim`, `substring` e `try/catch`

Quando a validacao e "se nao explodiu, ta valido", o padrao esta em pleno vigor.

8.19.3 Por que aparece

- pressa para integrar sistemas heterogeneos
- aversao a criar DTO/JSON/XML formal
- legado com protocolo artesanal
- tentativa de economizar mudancas de assinatura

No curtissimo prazo, pode destravar entrega. No medio, vira debito tecnico dificil de auditar.

8.19.4 Exemplo didatico

8.19.4.1 Versao POG

```
String payload =
    "nome=Ana,idade=29,ativo=true|nome=Joao,idade=31,ativo=false"
    ;
```

Se um nome vier com virgula (`"Ana, Maria"`), tudo quebra.

8.19.4.2 Versao com contrato simples

```
public record UsuarioDTO(String nome, int idade, boolean
    ativo) {}

List<UsuarioDTO> usuarios = List.of(
    new UsuarioDTO("Ana", 29, true),
    new UsuarioDTO("Joao", 31, false)
);
```

Ou, se fronteira exigir texto, use formato estruturado (JSON/CSV formal) com parser robusto e esquema validado.

8.19.5 Impacto operacional

- bugs de integracao de dificil reproducao

- acoplamento forte ao “formato secreto”
- evolucao dolorosa (adicionar campo quebra consumidores antigos)
- testes extensos so para validar parsing

8.19.6 Mitigacao pragmatica

1. mapear strings-protocolo mais criticas
2. criar parser dedicado com validacao clara
3. converter cedo para objeto tipado
4. planejar migracao para contrato explicito

Mesmo sem reescrever tudo, so de isolar parsing em um ponto voce reduz caos.

8.19.7 Resumo POG

String Sushiman e arte de empilhar informacao heterogenea em texto linear e chamar isso de protocolo. Funciona enquanto todos decoram a ordem. Quando alguem esquece, estoura em producao.

No idioma POGramador: e servir feijoada em rolinho de sushi. Alimenta, mas cada mordida e um evento imprevisivel.

8.19.8 Mini checklist de mitigacao

Antes de aceitar uma linguica de string em producao, valide tres pontos: formato versionado, parser unico e erro com mensagem clara. Sem isso, cada consumidor interpreta o payload de um jeito e a integracao vira loteria. Em ambiente serio, protocolo textual sem contrato formal e convite para incidente recorrente.

8.20 Sleeper Human Factor

O **Sleeper Human Factor** aplica atraso artificial para simular processamento, sincronizar corridas acidentais ou "melhorar percepcao" do usuario. O instrumento ritual e `sleep`.

```
public class MedidorDePOGresso implements Runnable {
    public void run() {
        while (true) {
            // Realiza um processamento rapido aqui...
            try {
                // ... atrasa propositalmente aqui
                Thread.sleep(1000);
            } catch (InterruptedException exc) {
            }

↪   progress.setValue(blablabla.getPorcentagem());
        }
    }
}
```

No curto prazo, parece resolver sintomas. No longo, vira latencia institucionalizada.

8.20.1 Onde esse padrao aparece

- interface piscando rapido demais e alguem "acalma" com delay
- integracao eventual falhando e o time adiciona espera fixa
- teste instavel ficando "verde" com `sleep(2000)`
- fila/concorrencia sem sincronizacao correta

Quando o sistema depende de dormir para funcionar, o design acordou errado.

8.20.2 Motivos reais para adocao

- corrida de thread dificil de reproduzir

- deadline apertado com bug intermitente
- falta de mecanismo de sincronizacao/evento
- cultura de “se resolveu, nao mexe”

O Human Factor nao e burrice; e resposta emergencial. O problema e deixar permanente.

8.20.3 Exemplo didatico

8.20.3.1 Versao POG

```
public void enviarNotificacao(Pedido pedido) {
    salvar(pedido);
    try {
        Thread.sleep(3000); // espera "banco refletir"
    } catch (InterruptedException e) {
    }
    mensageria.publicar(pedido.getId());
}
```

8.20.3.2 Versao com sincronizacao explicita

```
public void enviarNotificacao(Pedido pedido) {
    Pedido salvo = repositorio.salvar(pedido);
    // publica quando ha id persistido, sem espera
    ↪ arbitraria
    mensageria.publicar(salvo.getId());
}
```

Se precisar de consistencia assincrona, use evento transacional, fila confirmada ou mecanismo de retry com backoff controlado. Nao tempo fixo magico.

8.20.4 Impacto tecnico

- tempo de resposta pior sem ganho funcional

- throughput reduzido sob carga
- comportamento imprevisivel conforme ambiente
- testes lentos e flakey

Delay fixo pode passar na maquina do dev e falhar em producao, ou vice-versa.

8.20.5 Como remover com baixo risco

1. localizar sleeps fora de UI de animacao intencional
2. classificar por finalidade (sincronizacao, UX, workaround)
3. substituir por evento, callback, lock ou polling robusto com timeout
4. medir antes/depois com metrica de latencia

8.20.6 Sobre UX real

Nem todo atraso e pecado. Em UX, feedback visual minimo pode ser util para comunicar estado. A diferenca e intencao e local:

- atraso visual controlado na camada de interface: ok
- atraso tecnico para esconder bug de fluxo: risco alto

8.20.7 Resumo POG

Sleeper Human Factor e anestesia operacional. O paciente para de reclamar por alguns segundos, mas a causa da dor permanece.

No catecismo POGristico: se o bug corre demais, deita ele no `sleep` e chama de experiencia humana otimizada.

8.21 Black Cat In A Dark Room

O **Black Cat In A Dark Room** é o padrão em que um método recebe um `Map` genérico (ou estrutura equivalente) com tudo dentro: parâmetros de entrada, flags de comportamento, contexto técnico e, às vezes, traumas da sprint passada.

É como procurar um gato preto num quarto escuro: você sabe que algo está lá, mas não sabe onde, nem em qual tipo.

8.21.1 Anatomia da gambiarra

A ideia inicial parece elegante: "em vez de 12 parâmetros, passo um `Map` só". O problema é que esse ganho de assinatura vira perda de contrato.

```
public Object processar(Map<String, Object> params) {
    String operacao = (String) params.get("op");
    Long clienteId =
    ↪  Long.valueOf(params.get("id").toString());
    Boolean urgente = Boolean.valueOf(params.get(
    ↪  "urgente").toString());

    // Se alguém enviou "true" como "S" já era.
    // Se "id" vier nulo, explode aqui.
    // Se a chave vier como "clienteId" em outro ponto,
    ↪  não funciona.

    return servico.executar(operacao, clienteId,
    ↪  urgente);
}
```

O compilador para de ajudar cedo. E a validação passa a ser uma colcha de retalhos em runtime.

8.21.2 Cheiro técnico associado

Esse padrão conversa diretamente com smells conhecidos:

- **Long Parameter List** disfarçado
- **Primitive Obsession** (muito dado cru, pouca modelagem)
- **Data Clumps** (os mesmos campos reaparecendo juntos em vários lugares)

Na prática, você troca uma assinatura verbosa por acoplamento implícito: todo mundo precisa "saber de cabeça" os nomes mágicos das chaves.

8.21.3 Exemplo didático de evolução

8.21.3.1 Versão POG

```
public void criarBoleto(Map<String, Object> map) {
    String nome = (String) map.get("nome");
    String documento = (String) map.get("doc");
    BigDecimal valor = new
    ↪  BigDecimal(map.get("valor").toString());
    String vencimento = (String) map.get("dataVenc");

    // várias conversões, vários riscos silenciosos
}
```

8.21.3.2 Versão com contrato explícito

```
public record CriarBoletoRequest(
    String nome,
    String documento,
    BigDecimal valor,
    LocalDate dataVencimento
) {}

public void criarBoleto(CriarBoletoRequest req) {
```

```
    // Aqui o compilador ajuda
    // e o contrato fica autoexplicativo
}
```

Benefícios imediatos:

- tipagem forte
- documentação natural na assinatura
- erro detectado antes da produção
- teste mais simples e legível

8.21.4 Por que times continuam usando Map genérico

Motivos reais, e não caricatos:

- integração com payload dinâmico/legado
- tentativa de evitar mudanças em cadeia
- medo de criar classes “demais”
- pressão de prazo

Ou seja: o padrão não nasce de burrice, nasce de contexto ruim. O problema é quando ele vira decisão padrão para tudo.

8.21.5 Como usar sem virar caos

Se precisar usar `Map` por fronteira técnica (por exemplo, parser de payload desconhecido), faça contenção:

1. converta para objeto tipado o mais cedo possível
2. valide presença e tipo das chaves logo na entrada
3. nunca propague `Map` cru pela regra de negócio
4. centralize mapeamento em um único ponto

Assim você transforma o quarto escuro em corredor iluminado.

8.21.6 Resumo POG

Black Cat In A Dark Room é irresistível no dia de entrega porque parece flexível. Só que flexibilidade sem contrato cobra caro na manutenção.

Em linguagem POGráfica: é uma mochila sem divisória. Cabe tudo. Você só não acha nada quando precisa, principalmente em produção às 17h58 de sexta-feira.

8.22 Mega Zord

O **Mega Zord** e o padrao da superfuncao: um metodo gigante que concentra multiplas responsabilidades para “facilitar manutencao”. Em vez de modularizar, funde tudo em uma unidade colossal.

No discurso: centralizacao. Na pratica: acoplamento total.

8.22.1 Caracteristicas classicas

- centenas ou milhares de linhas em um unico metodo
- muitos `if`, `switch` e variaveis de controle
- efeitos colaterais em banco, arquivo, API e tela no mesmo fluxo
- baixa cobertura de teste por medo de tocar no bloco

Quando um metodo exige mapa mental para ser lido, o Mega Zord ja atingiu forma completa.

8.22.2 Exemplo didatico (versao POG)

```
public Resultado processarTudo(Pedido pedido, Usuario
↪  usuario, Map<String, Object> cfg) {
    // valida entrada
    // calcula imposto
    // aplica desconto
    // grava banco
    // envia email
    // chama API externa
    // gera log
    // atualiza cache
    // devolve resposta
    // 800 linhas depois...
    return resultado;
}
```

O problema nao e tamanho por si so. E mistura de motivos de mudanca. Uma regra fiscal muda por motivo A. O email muda por

motivo B. Estao presos no mesmo bloco.

8.22.3 Por que times criam Mega Zord

- evolucao incremental sem refatoracao
- pressa para encaixar regra nova em ponto “que ja funciona”
- baixa confianca em extrair componentes
- ausencia de ownership claro do modulo

A cada sprint, entra “so mais um if”. Em um ano, nasce a criatura.

8.22.4 Efeito colateral

- regressao frequente
- review superficial (arquivo grande desencoraja analise profunda)
- dependencia de “guardiao do modulo”
- onboarding lento

O sistema fica robusto para quem criou e hostil para o resto da equipe.

8.22.5 Exemplo de decomposicao minima

```
public Resultado processarTudo(Pedido pedido, Usuario
↪ usuario, Map<String, Object> cfg) {
    validarEntrada(pedido, usuario);
    Valores valores = calcularValores(pedido, cfg);
    PersistenciaOut persistencia =
↪ persistirPedido(pedido, valores);
    integrarServicosExternos(persistencia);
    notificarPartes(persistencia);
    return montarResultado(persistencia);
}
```

Ainda e um fluxo central, mas com fronteiras internas claras. Isso ja permite teste por etapa e reduz risco de alteracao.

8.22.6 Estrategia pragmatica de reducao

1. mapear secoes logicas no metodo gigante
2. extrair uma secao por vez para metodo privado
3. adicionar testes de regressao antes/depois da extracao
4. mover etapas estaveis para classes dedicadas

Sem reescrita completa. Sem promessa de refatoracao epica.

8.22.7 Resumo POG

Mega Zord e poderoso para entrega imediata e aterrorizante para evolucao sustentavel. Quanto mais cresce, mais caro fica tocar nele.

No sotaque POG: e juntar todos os fios do painel num unico disjuntor e comemorar que “agora ta centralizado”.

8.23 THUNDER MEGA ZORD

O **Thunder Mega Zord** e a fusao entre duas potencias da gambiarra: metodo gigantesco + contrato opaco com `Map` de entrada e `Object[]` de saida. E a tempestade perfeita do acoplamento.

```
/**
 * Processa
 *
 * @param parametros
 * @return
 * @throws Throwable
 */
public static Object[] processar(Map parametros) throws
↪  Throwable {
    // Aí é aquilo, mermão...
    // ...
    // ...
    return processado;
}
```

A assinatura nao diz quase nada. So promete incerteza com confianca.

8.23.1 Como identificar

- `Map` sem tipo para entrada complexa
- `Object[]` com indices sem semantica
- throws amplo (`Throwable`/`Exception`) para tudo
- javadoc generico sem contrato util

Quando a documentacao diz “Processa” e o retorno e `Object[]`, voce nao tem API: voce tem adivinhacao.

8.23.2 Exemplo didatico de risco

```
Object[] retorno = processar(params);
String status = (String) retorno[0];
BigDecimal total = (BigDecimal) retorno[1];
Date data = (Date) retorno[2];
```

Se alguem mudar a ordem interna para `[total, status, data]`, o compilador nao reclama. O bug aparece em runtime, geralmente em producao.

8.23.3 Por que esse padrao surge

- metodo legado cresceu sem contrato formal
- tentativa de evitar criacao de classes de entrada/saida
- integracao rapida entre equipes sem alinhamento de tipos
- “nao mexe na assinatura que quebra tudo”

Em ambientes de prazo extremo, e compreensivel. Em ambiente de manutencao continua, e erosao programada.

8.23.4 Versao didatica mais segura

```
public record ProcessarRequest(Long clienteId,
↪  BigDecimal valor, boolean urgente) {}
public record ProcessarResponse(String status,
↪  BigDecimal total, LocalDate dataProcessamento) {}

public ProcessarResponse processar(ProcessarRequest req)
↪  {
    // regra aqui
    return new ProcessarResponse("OK", req.valor(),
    ↪  LocalDate.now());
}
```

Agora:

- contrato e autoexplicativo
- compilador ajuda
- mudanca de campo exige ajuste explicito
- teste fica legivel

8.23.5 Migracao incremental possivel

1. manter assinatura antiga como adaptador temporario
2. converter `Map` para request tipado internamente
3. devolver response tipado e mapear para `Object[]` apenas no adaptador
4. migrar consumidores gradualmente

Assim voce moderniza sem quebrar tudo de uma vez.

8.23.6 Resumo POG

Thunder Mega Zord entrega flexibilidade instantanea e debito estrutural de longo prazo. Ele parece universal porque aceita tudo e devolve qualquer coisa.

No evangelho da TelePOG: se nao souber diagnosticar, reinicia. Se continuar ruim, culpa a internet e abre outro chamado.

8.23.7 Mini checklist de mitigacao

Contrato opaco precisa de quarentena: converta entradas e saidas genericas em objetos tipados na fronteira do metodo. Mesmo que internamente continue legado por um tempo, essa adaptacao reduz risco imediato e prepara migracao segura dos consumidores.

8.24 Controller Confusion

O **Controller Confusion** é a evolução natural do MVC cansado. No discurso, o projeto ainda "usa camadas". No código real, o controller virou templo monolítico: valida, transforma, persiste, chama API externa, gera relatório e decide mensagem de tela.

É o padrão VCC: **View/Controller Confusion**. Em estágio avançado, vira CCC: **Chaotic Controller Confusion**.

8.24.1 De onde isso vem

Esse padrão quase sempre nasce em projeto com uma mistura de:

- prazo curto com escopo longo
- time mudando frequentemente
- ausência de limites claros entre camadas
- cultura de "só mais esse if aqui no endpoint"

No início, parece uma economia. Você evita criar serviço, evita DTO, evita caso de uso. Só que cada economia dessas vira dívida semântica.

Com o tempo, o controller acumula responsabilidades demais e vira equivalente ao anti-pattern conhecido como **God Object**: uma entidade central que conhece tudo e acopla tudo.

8.24.2 Exemplo didático (Controller Confusion clássico)

```
1  @PostMapping("/pedidos")
2  public ResponseEntity<?> criar(@RequestBody Map<String,
   ↪  Object> body) {
3      try {
4          // 1) Validação de entrada
5          if (body.get("clienteId") == null) {
6              return ResponseEntity.badRequest().body(
               ↪  "clienteId
               ↪  obrigatório");
7          }
```

```
8
9           // 2) Regra de negócio direto no controller
10          BigDecimal total = new
            ↪  BigDecimal(body.get("total").toString());
11          if (total.compareTo(BigDecimal.ZERO) <= 0) {
12              return
                ↪  ResponseEntity.badRequest().body("total
                ↪  inválido");
13          }
14
15          // 3) Persistência direto aqui
16          PedidoEntity pedido = new PedidoEntity();
17          pedido.setClienteId(Long.parseLong(body.get(
        ↪  "clienteId").toString()));
18          pedido.setTotal(total);
19          pedidoRepository.save(pedido);
20
21          // 4) Integração externa também aqui
22          String token = authClient.login("usuario",
            ↪  "senha");
23          freteClient.calcular(token, pedido.getId(),
        ↪  pedido.getTotal());
24
25          // 5) Formatação de resposta
26          Map<String, Object> resp = new HashMap<>();
27          resp.put("id", pedido.getId());
28          resp.put("status", "CRIADO");
29          return ResponseEntity.ok(resp);
30      } catch (Exception e) {
31          // 6) Tratamento genérico sem contexto
32          return ResponseEntity.internalServerError().body
            ↪  ("erro
            ↪  inesperado");
33      }
34  }
```

Repare na sobrecarga cognitiva. Um único método mistura várias

preocupações que mudam por motivos diferentes. Resultado: qualquer ajuste simples vira cirurgia de alto risco.

8.24.3 Sinais de que virou confusão

- controller com centenas ou milhares de linhas
- mesmo endpoint mexendo em banco, fila, arquivo e API externa
- testes de controller gigantes tentando cobrir regra de negócio
- bugs regressivos frequentes por efeitos colaterais não intencionais

Isso bate diretamente com smells clássicos de engenharia de software: *long method*, *long parameter list*, *divergent change* e *shotgun surgery*.

8.24.4 Versão didática com separação mínima

```
@PostMapping("/pedidos")
public ResponseEntity<?> criar(@RequestBody
↪  CriarPedidoRequest req) {
    try {
        ResultadoCriacaoPedido resultado =
↪  criarPedidoUseCase.executar(req);
        return
        ↪  ResponseEntity.status(201).body(resultado);
    } catch (ValidacaoException e) {
        return ResponseEntity.badRequest().body(e.
        ↪  getMessage());
    } catch (IntegracaoException e) {
        return ResponseEntity.status(502).body("Falha em
        ↪  integração externa");
    }
}

public class CriarPedidoUseCase {
    public ResultadoCriacaoPedido
    ↪  executar(CriarPedidoRequest req) {
```

```
        // validação e regras aqui, de forma testável
        // persistência via gateway/repositório
        // integrações encapsuladas
        // retorno explícito
    }
}
```

Aqui o controller volta ao papel dele: orquestrar I/O HTTP e traduzir resultado para resposta. A regra deixa de ficar refém de framework web.

8.24.5 Como reduzir sem reescrever tudo

Abordagem pragmática, sprint por sprint:

1. escolha um endpoint crítico (o mais alterado)
2. extraia uma regra para um serviço/caso de uso
3. mantenha assinatura antiga para não quebrar cliente
4. adicione teste no caso de uso extraído
5. repita até o controller emagrecer

Isso evita refatoração épica e reduz risco operacional.

8.24.6 Resumo POG

Controller Confusion é confortável no curto prazo, cruel no médio e impagável no longo. É o padrão ideal para gerar chamados em série e sustentar o emprego de meio time de sustentação.

Se a meta é continuar entregando sem criar um cemitério de endpoint, trate controller como fronteira e não como depósito. Caso contrário, cedo ou tarde, o MVC vira apenas uma lenda oral contada para estagiário.

8.25 No More Layers

No **No More Layers**, arquitetura em camadas e considerada burocracia. Tudo acontece no mesmo lugar, normalmente na tela/controlador: validacao, regra de negocio, acesso a dados e formatacao de resposta.

A promessa e velocidade. O custo e acoplamento total.

8.25.1 Exemplo classico

```
private void botaoSalvar_Click(Object sender, EventArgs
↪  e) {
    // 1) le campos da tela
    // 2) valida regra
    // 3) monta SQL
    // 4) executa no banco
    // 5) monta mensagem de retorno
    // 6) atualiza grid
}
```

Tudo numa unica rotina de interface. Parece eficiente enquanto o sistema e pequeno. Quando cresce, cada alteracao de regra exige tocar na tela.

8.25.2 Consequencias praticas

- baixa reutilizacao de regra de negocio
- testes automatizados dificeis
- dependencia forte de framework de UI
- regressao em cascata a cada ajuste visual

Quando a troca de banco exige alterar formulario, a separacao de responsabilidades ja morreu.

8.25.3 Onde esse padrao e comum

- legados desktop (Delphi, VB6, WinForms)
- sistemas web antigos com script + SQL inline
- projetos que cresceram sem desenho arquitetural
- times pressionados por entregas imediatas

Nao e um problema de tecnologia especifica. E um problema de limite de responsabilidade.

8.25.4 Exemplo didatico de separacao minima

```
// camada de interface
public Resultado salvarPedido(FormPedido form) {
    CriarPedidoInput input = mapear(form);
    return criarPedidoUseCase.executar(input);
}

// caso de uso
public Resultado executar(CriarPedidoInput input) {
    validar(input);
    Pedido pedido = Pedido.novo(input);
    repositorio.salvar(pedido);
    return Resultado.sucesso(pedido.getId());
}
```

Aqui a tela para de saber SQL e regra fiscal. Ela apenas traduz entrada/saida.

8.25.5 Correcao gradual

1. escolher um fluxo com muita manutencao
2. extrair regra para servico/caso de uso
3. manter UI como adaptador
4. repetir por partes sem reescrita global

Abordagem incremental reduz risco de parada total.

8.25.6 Beneficio real de manter camadas

- mudanca de regra sem mexer na tela
- possibilidade de reaproveitar fluxo em API/job
- testes de negocio sem subir interface
- codigo mais legivel para onboarding

Arquitetura em camadas nao e luxo academico. E estrategia para reduzir custo de mudanca.

8.25.7 Resumo POG

No More Layers e gostoso no curto prazo: menos arquivos, mais entrega rapida. No longo prazo, transforma cada ajuste simples em operacao delicada.

Na linguagem POG: e cozinhar, atender cliente e lavar prato no mesmo fogao. Da para fazer. Escalar e outra historia.

8.25.8 Mini checklist de mitigacao

Se a tela conhece SQL, regra fiscal e formato de resposta externa, a camada de interface ja esta sobrecarregada. Comece separando apenas uma responsabilidade por sprint. Em poucos ciclos, o ganho de teste e previsibilidade aparece sem precisar pausar o roadmap.

8.26 Db Is Our God

No **Db Is Our God**, o banco de dados deixa de ser camada de persistencia e vira centro do universo: regra de negocio, orquestracao de fluxo, validacao, transformacao, geracao de relatorio e ate HTML.

Tambem conhecido como **In DB We Trust**.

8.26.1 Dogmas do padrao

Tudo vai para o banco:

- dados e arquivos
- imagens e logs
- regra de negocio em procedure
- tratamento de erro em trigger
- composicao de resposta em SQL

A promessa e “centralizar para padronizar”. O risco e concentrar complexidade e gargalo no mesmo ponto.

8.26.2 Exemplo didatico

```sql
CREATE PROCEDURE processar_pedido(
    IN p_cliente_id BIGINT,
    IN p_valor DECIMAL(10,2)
)
BEGIN
    -- valida cliente
    -- calcula imposto
    -- grava pedido
    -- atualiza estoque
    -- chama funcao de notificacao
    -- retorna mensagem formatada
END;
```

Procedure grande pode funcionar bem em cenario especifico. O problema surge quando ela vira lugar padrao para toda regra, sem

fronteira clara entre dominio e persistencia.

8.26.3 Sintomas de culto ao banco

- alteracao de regra exige deploy de script + janela de manutencao
- time de aplicacao nao entende mais o fluxo completo
- logica espalhada entre app e SQL sem contrato
- dificuldade de testar regra fora do ambiente de banco

Quando o dominio mora em trigger, a aplicacao vira um cliente passivo de eventos invisiveis.

8.26.4 Por que isso acontece

- historico forte de time DBA-centric
- performance local excelente em consultas complexas
- legado construido antes de camada de servico madura
- tentativa de garantir consistencia “na marra”

Existe valor real em banco: transacao, integridade referencial, constraints, consulta. O excesso e que vira anti-pattern.

8.26.5 Exemplo de equilibrio pragmatico

- banco cuida de integridade e consulta eficiente
- aplicacao cuida de caso de uso e orquestracao
- procedures ficam para cenarios realmente justificados

```
public void criarPedido(CriarPedidoInput input) {
    validarRegras(input);      // regra de negocio
    Pedido pedido = mapper.map(input);
    repositorio.salvar(pedido); // persistencia
}
```

No banco:

```
ALTER TABLE pedido
ADD CONSTRAINT chk_valor_positivo CHECK (valor > 0);
```

Cada camada cumpre seu papel.

8.26.6 Estrategia de migracao

1. mapear procedures criticas por dominio
2. separar validacoes de negocio das constraints de integridade
3. expor regras em camada de aplicacao com testes
4. manter no banco o que e estrutural e transacional

Sem guerra santa. Com criterio.

8.26.7 Resumo POG

Db Is Our God da sensacao de controle total, mas centraliza risco e reduz flexibilidade de evolucao. Banco e essencial, mas nao precisa ser divindade onipotente do sistema.

No catecismo POGramador: quando tudo e milagre de procedure, qualquer manutencao vira peregrinacao com janela de madrugada.

8.27 Snow White Returns

O **Snow White Returns** celebra multiplos pontos de retorno em funcoes gigantes. A ideia original era simplificar casos locais. O uso extremo transforma fluxo em labirinto.

POrque um `return` claro quando voce pode ter sete, doze ou vinte e um?

8.27.1 Como o padrao se forma

- metodo cresce sem refatoracao
- cada condicao ganha um `return` de emergencia
- caminhos de saida se multiplicam sem estrategia
- leitura sequencial deixa de representar fluxo real

Em funcoes pequenas, early return pode melhorar legibilidade. Em funcoes enormes e sem estrutura, vira desorientacao.

8.27.2 Exemplo didatico (caotico)

```
public Resultado processar(Pedido pedido) {
    if (pedido == null) return Resultado.erro("pedido
    ↪  nulo");
    if (pedido.getItens().isEmpty()) return
    ↪  Resultado.erro("sem itens");

    if (!estoqueDisponivel(pedido)) return
    ↪  Resultado.erro("sem estoque");

    if (pedido.isRetirada()) {
        if (!validarLoja(pedido)) return
        ↪  Resultado.erro("loja invalida");
        return Resultado.ok("retirada liberada");
    }

    if (pedido.isEntrega()) {
```

```
        if (!validarEndereco(pedido)) return
        ↪  Resultado.erro("endereco invalido");
        if (pedido.getFrete() == null) return
        ↪  Resultado.erro("frete ausente");
        return Resultado.ok("entrega liberada");
    }

    return Resultado.erro("tipo de entrega
    ↪  desconhecido");
}
```

Aqui ainda parece legivel porque e curto. Agora imagine isso com 700 linhas e efeitos colaterais entre condicoes.

8.27.3 Risco principal

- ponto de saida demais dificulta rastrear estado
- logging e auditoria ficam inconsistentes
- manutencao adiciona novos retornos sem revisar os antigos
- mudanca de regra quebra caminhos esquecidos

No fim, o bug nao esta em um return especifico. Esta na falta de desenho do fluxo.

8.27.4 Versao mais organizada

```
public Resultado processar(Pedido pedido) {
    validarEntrada(pedido);

    if (pedido.isRetirada()) {
        return processarRetirada(pedido);
    }

    if (pedido.isEntrega()) {
        return processarEntrega(pedido);
    }

```

```
        return Resultado.erro("tipo de entrega
        ↪  desconhecido");
    }

    private Resultado processarRetirada(Pedido pedido) {
        validarLojaRetirada(pedido);
        return Resultado.ok("retirada liberada");
    }

    private Resultado processarEntrega(Pedido pedido) {
        validarDadosEntrega(pedido);
        return Resultado.ok("entrega liberada");
    }
```

Ainda existem retornos multiplos, mas cada funcao tem escopo pequeno e intencao clara.

8.27.5 Como corrigir sem guerra

1. medir funcoes com maior complexidade ciclomática
2. extrair blocos por responsabilidade
3. manter retornos apenas onde aumentam clareza
4. padronizar log de entrada/saida por fluxo

Nao e sobre proibir `return` cedo. E sobre evitar floresta de saidas em metodo sem fronteira.

8.27.6 Resumo POG

Snow White Returns e divertido enquanto o autor lembra o caminho de cada saida. Quando o contexto muda, vira castelo sem planta baixa.

No idioma POG: cada return extra e uma porta secreta. Bom para quem construiu. Terrivel para quem herdou.

8.27.7 Mini checklist de mitigacao

Retornos multiplos so sao problema quando escondem complexidade acidental. Se cada retorno estiver em funcao pequena e com intencao clara, tudo bem. O anti-pattern surge quando os retornos viram atalho para evitar modelagem do fluxo principal.

Capítulo 9

Conclusões

Chegamos ao fim deste tomo maldito. Se voce leu ate aqui, ha duas possibilidades:

1. voce realmente se interessa por engenharia de software
2. voce esta fugindo de uma task com prazo suicida

Nos dois casos, parabens. Voce demonstrou coragem.

9.1 O que este livro tentou mostrar

A Programacao Orientada a Gambiarra nao e apenas uma piada interna da area. Ela e um fenomeno real, repetivel e sistemico.

POG nao nasce so de “dev ruim”. Ela nasce do encontro entre:

- pressao de prazo
- processo torto
- contexto incompleto
- incentivo desalinhado
- tomada de decisao sob estresse

Quando esses elementos se alinham, ate equipe boa produz artefardo.

9.2 As quatro grandes licoes

9.2.1 1. Gambiarra e inevitavel

Todo sistema vivo acumula improviso. Isso nao e falha moral. E caracteristica de software em producao.

Negar essa realidade so piora a qualidade das decisoes.

9.2.2 2. Nem toda POG e igual

Existe gambiarra tatica, conscientemente aplicada para conter incidente. Existe gambiarra estrutural, reproduzida por meses sem plano de saida.

Confundir as duas e o caminho mais rapido para virar refem do proprio codigo.

9.2.3 3. Nomear padrao aumenta lucidez

Quando voce chama algo de `Controller Confusion`, `Zipomatic Versioning` ou `Exception Success`, deixa de discutir no campo da opiniao e passa a discutir no campo da engenharia.

Nome reduz neblina.

9.2.4 4. Saida sempre e gradual

Projeto real nao aceita reforma espiritual instantanea. Quem promete “refatorar tudo” em uma sprint esta vendendo fanfic.

A evolucao sustentavel vem de pequenos movimentos:

- mapear pontos criticos
- reduzir risco incrementalmente
- proteger fluxo de negocio
- melhorar sem parar entrega

9.3 O paradoxo do POGramador

Quanto mais experiente, menos inocente. Quanto mais conhecimento, menos dogma. Quanto mais disciplina, menos heroicismo vazio.

O POGramador maduro nao e o que nunca faz gambiarra. E o que sabe exatamente **quando**, **por que** e **ate quando** vai conviver com ela.

9.4 Sobre culpa e responsabilidade

Se voce se reconheceu em varios capitulos, relaxe: todos nos ja passamos por isso.

A diferenca entre amador e profissional nao esta em nunca errar. Esta em:

- reconhecer o erro cedo
- assumir o impacto
- aprender com padrao recorrente
- nao terceirizar culpa para "o sistema"

Redirecao Tangencial diverte por cinco minutos. Responsabilidade tecnica sustenta carreira por decadas.

9.5 Um compromisso para levar daqui

Se este livro precisasse terminar com um pacto simples, seria este:

Continue entregando. Mas nunca entregue no automatico.

Pergunte sempre:

- qual problema estou resolvendo agora?
- qual problema estou criando para depois?
- quem vai pagar essa conta futura?

Essas tres perguntas, repetidas com honestidade, ja evitam metade das pogs catastroficas que assombram times inteiros.

9.6 Encerramento

POG e uma arte dominada por muitos, confessada por poucos e negada por quase todos.

Que voce saia deste livro com mais repertorio, mais senso de realidade e menos ilusao de pureza arquitetural.

E que Lady Murphy siga ao seu lado, nao como maldicao, mas como lembrete:

se algo pode dar errado, alguem vai dar deploy sexta-feira 18h.

POGae.

Referências

Desciclopedia. 2016. “Programação Orientada a Gambiarras - Desciclopédia”. *https://desciclopedia.org/wiki/Programa*. https://desciclopedia.org/wiki/Programação_Orientada_a_Gambiarras.

Jacquin, Érick. 2019. “Pesadelo na Cozinha: Pé de Fava – Parte 1 - YouTube”. *Pesadelo na Cozinha - Youtube*. https://www.youtube.com/watch?v=7WnrnIGRI6I&list=PLhLCtESG30THwTx9yu706hunJjd0aJHZZ.

Minha Cabeça, Vozes da. 2020. *Delírios de uma Mente Imundiçada*. Organizado por Juro por Deus. 17º ed. Instituto de Pesquisa Tireido Ku.