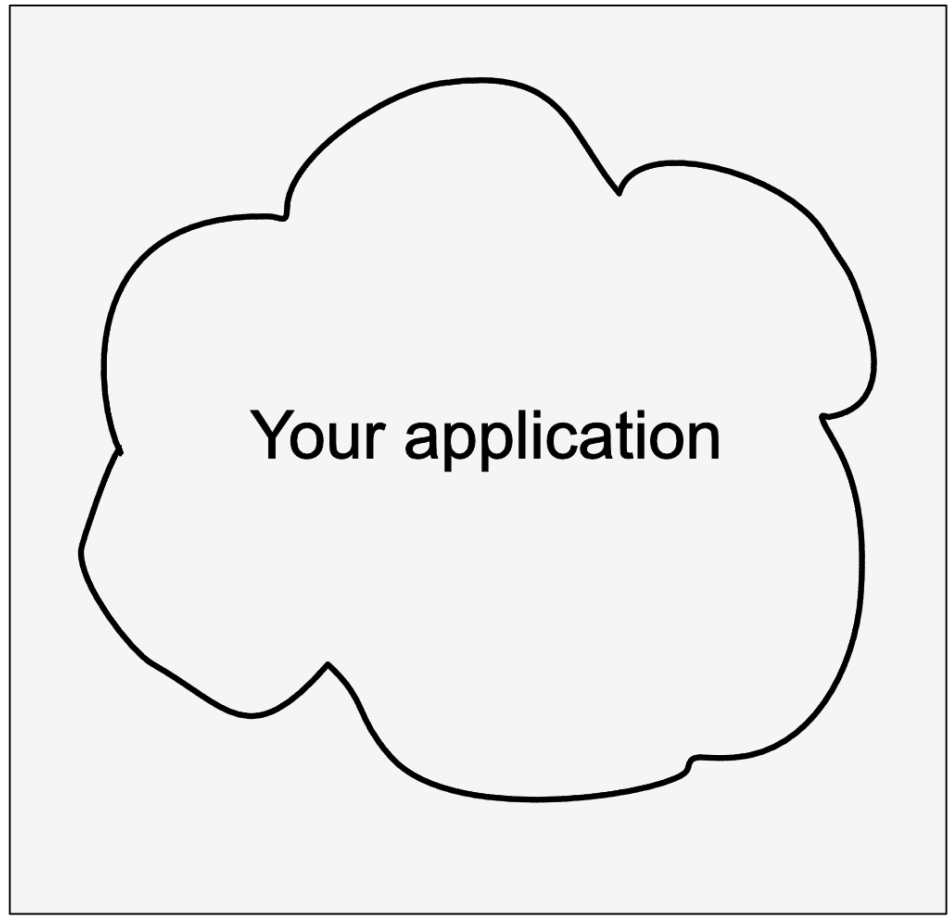
Refactoring Go

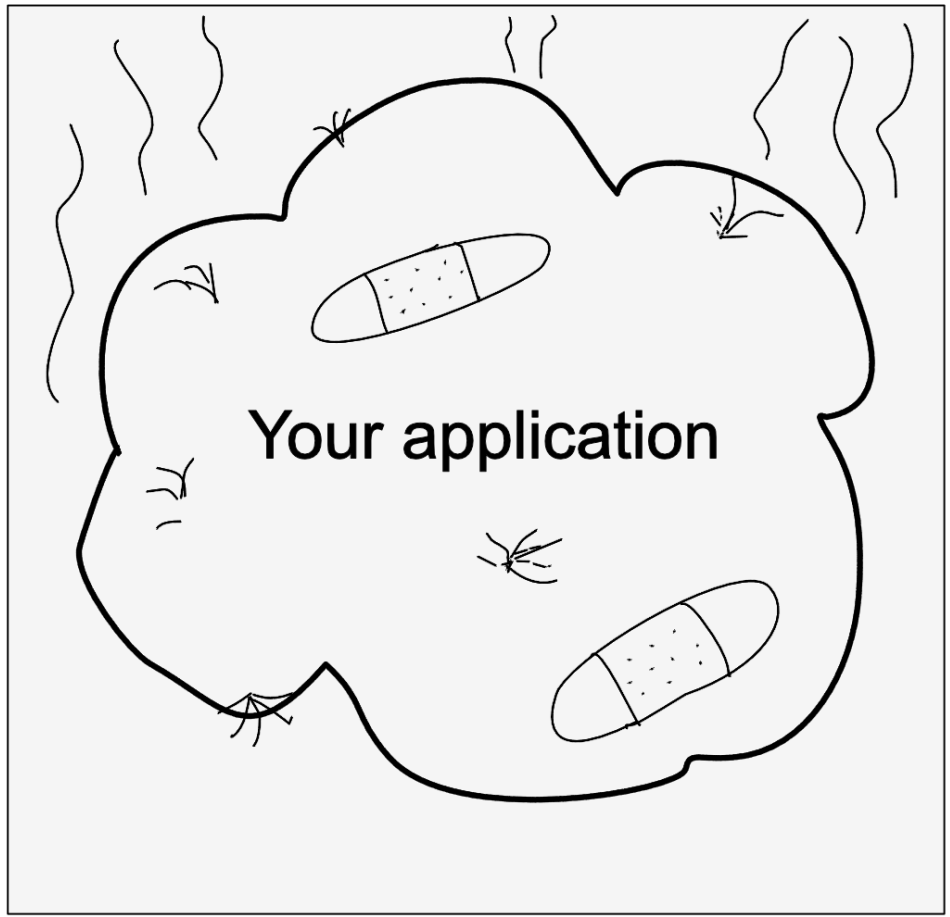
Patterns and Practices for Maintaining and Evolving Large Codebases

Brittany Ellich, GitHub | brittanyellich.com

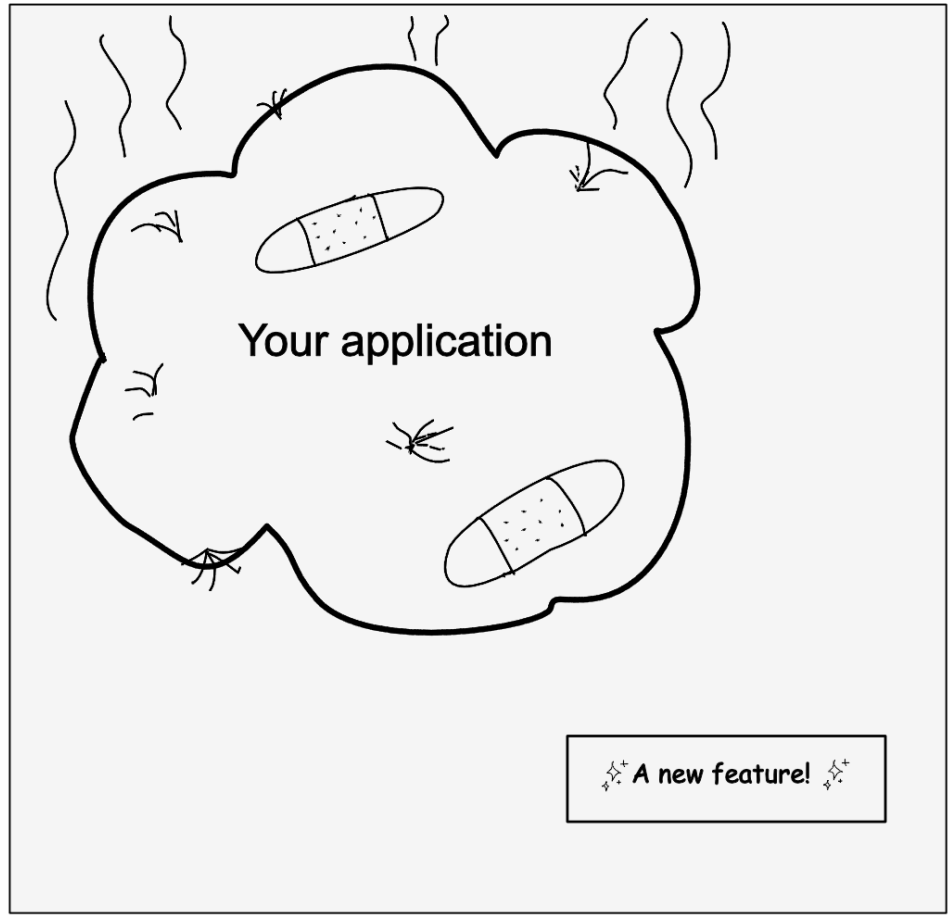
brittanyellich/refactoring-go

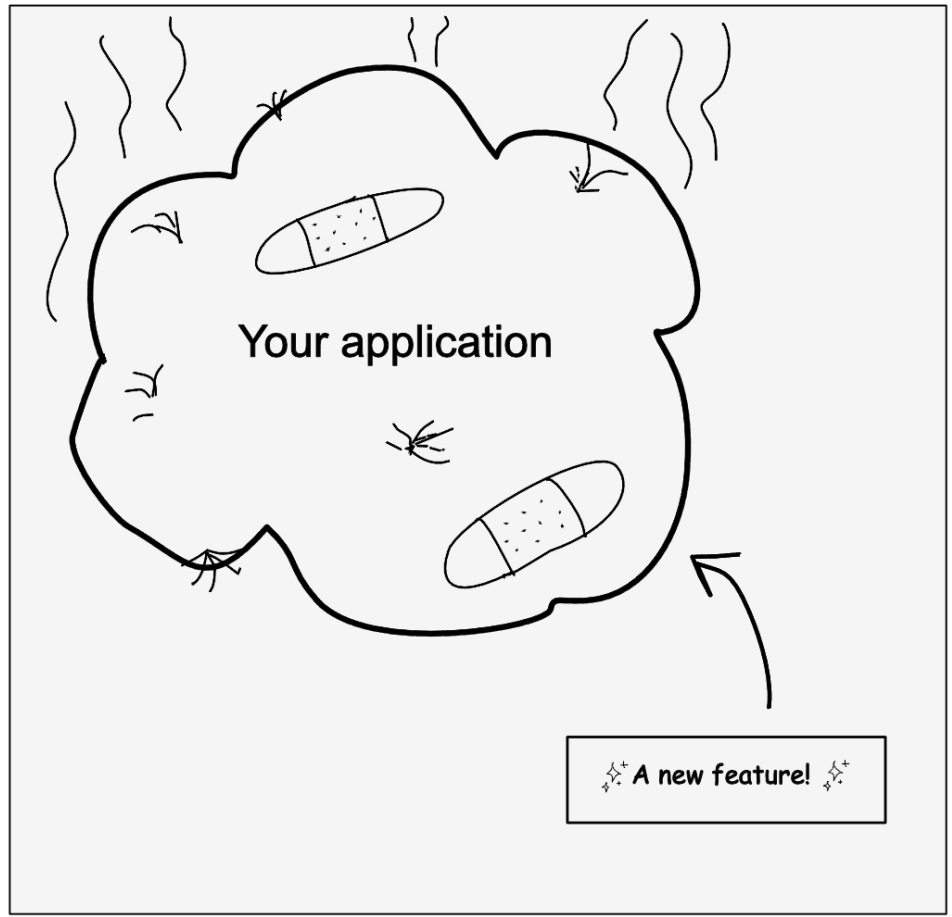


brittanyellich.com

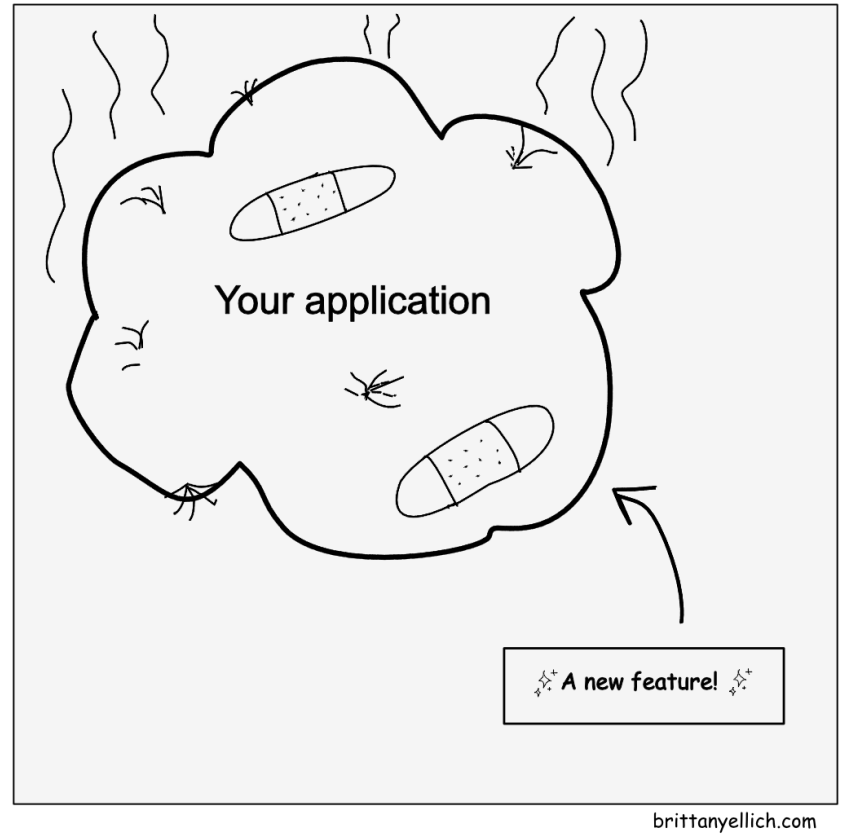


brittanyellich.com


Your application
A new feature!
brittanyellich.com

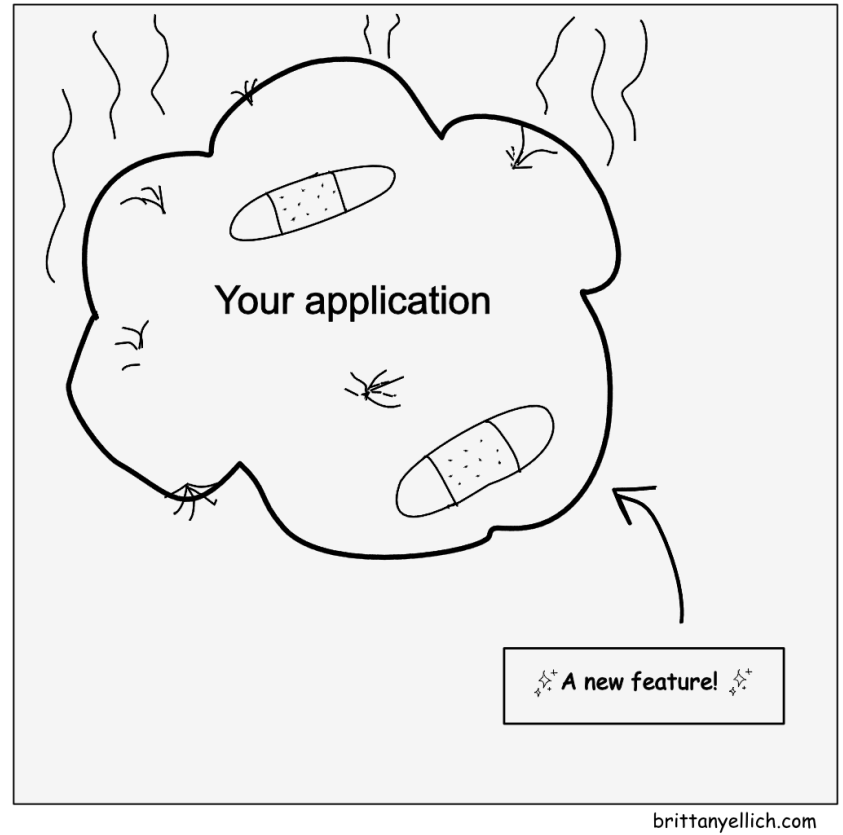

Your application
A new feature!
brittanyellich.com

"It would be faster to just rewrite the whole thing"



"It would be faster to just rewrite the whole thing"

Does this sound familiar?



You’re wrong.

You’re wrong.

(probably)

Refactoring Go

Patterns and Practices for Maintaining and Evolving Large Codebases

Brittany Ellich, GitHub

brittanyellich.com

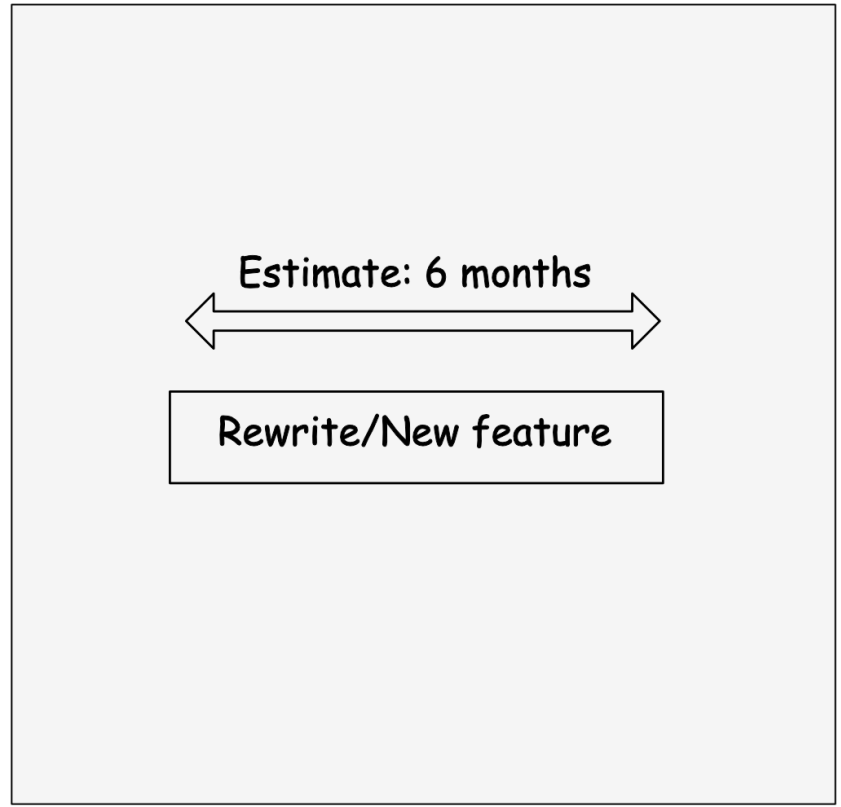
What we’re going to talk about

- Refactor, don’t rewrite: The top 3 arguments and why they’re wrong
- How to refactor: A systematic refactoring method
- How to prioritize your refactoring efforts: Focus on impact
- Getting AI to help you: How to use agents today

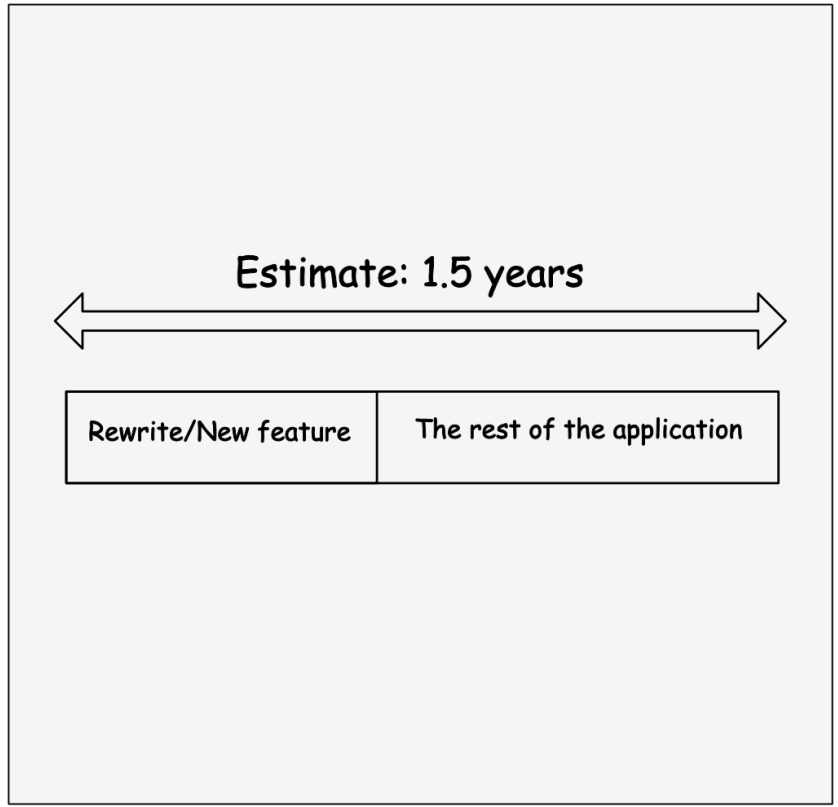
Refactor, don’t rewrite

The top 3 arguments and why they’re wrong

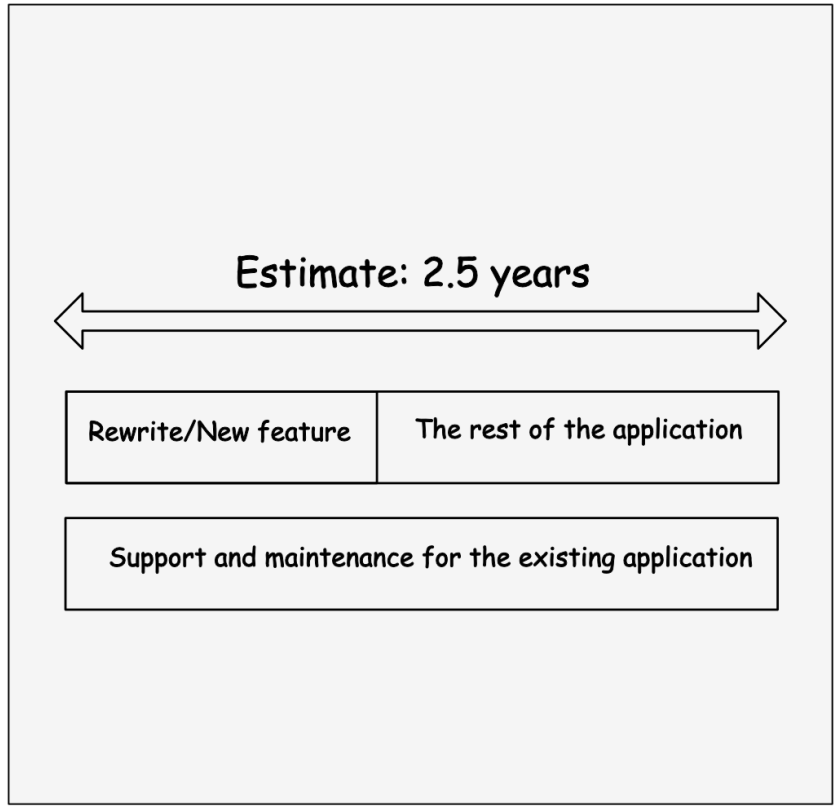
1. It will be faster to just rewrite it



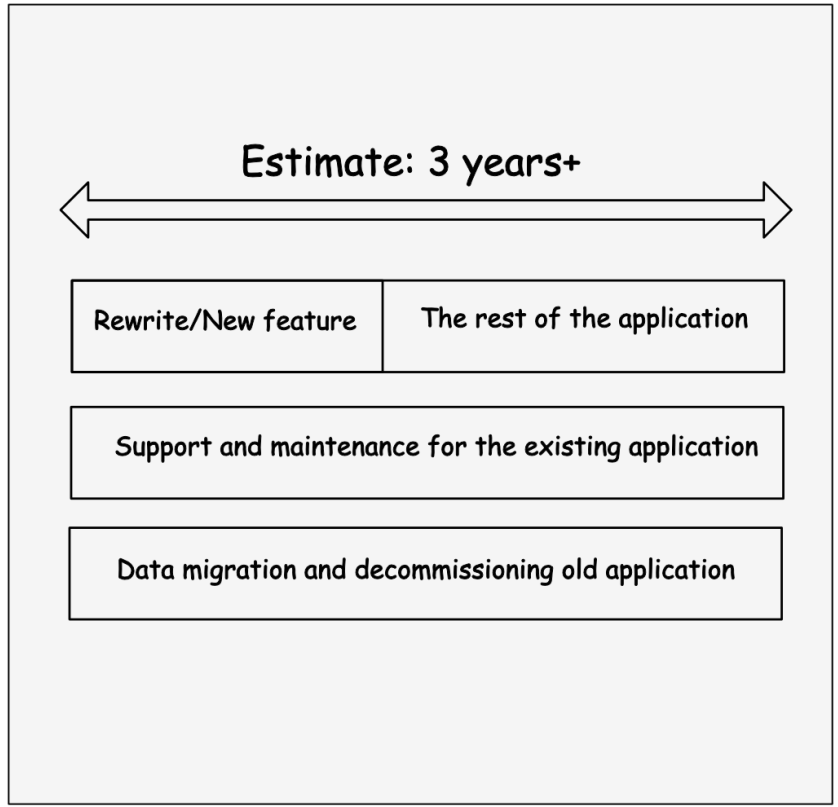
1. It will be faster to just rewrite it



1. It will be faster to just rewrite it

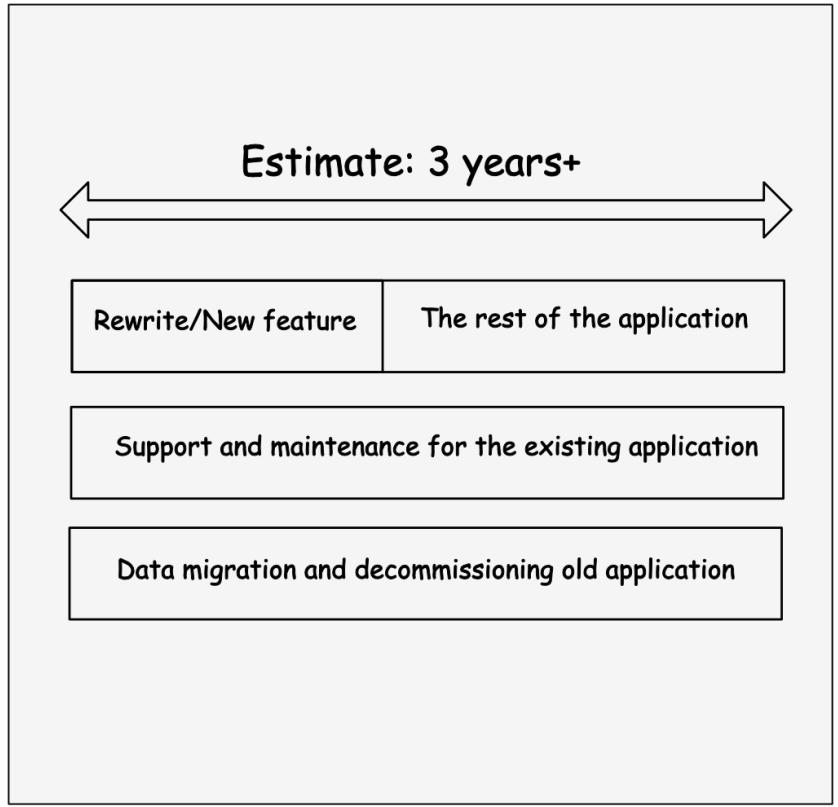


1. It will be faster to just rewrite it

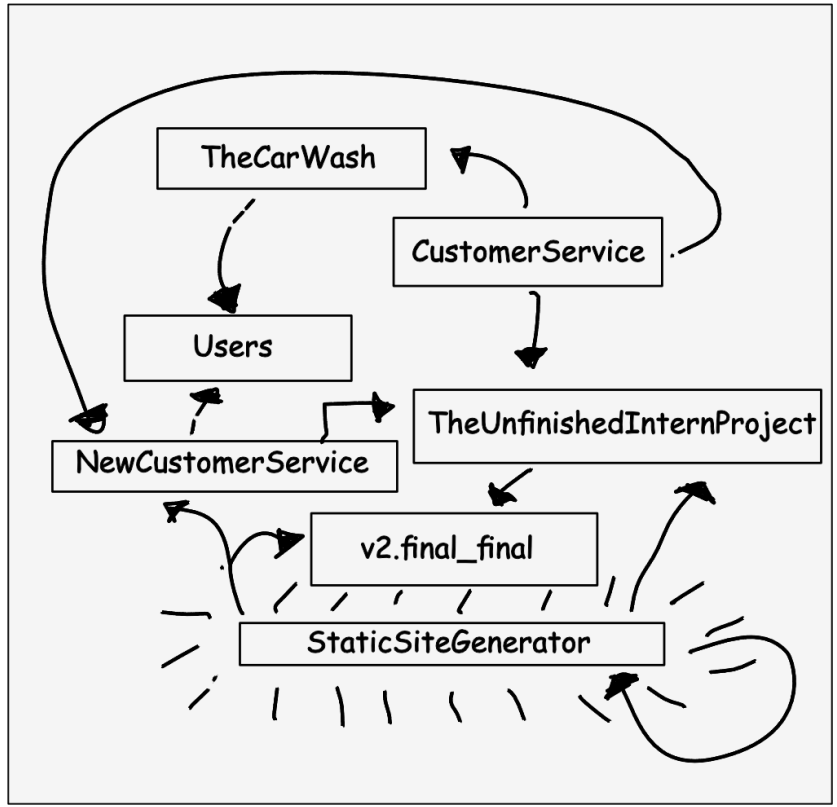


1. It will be faster to just rewrite it

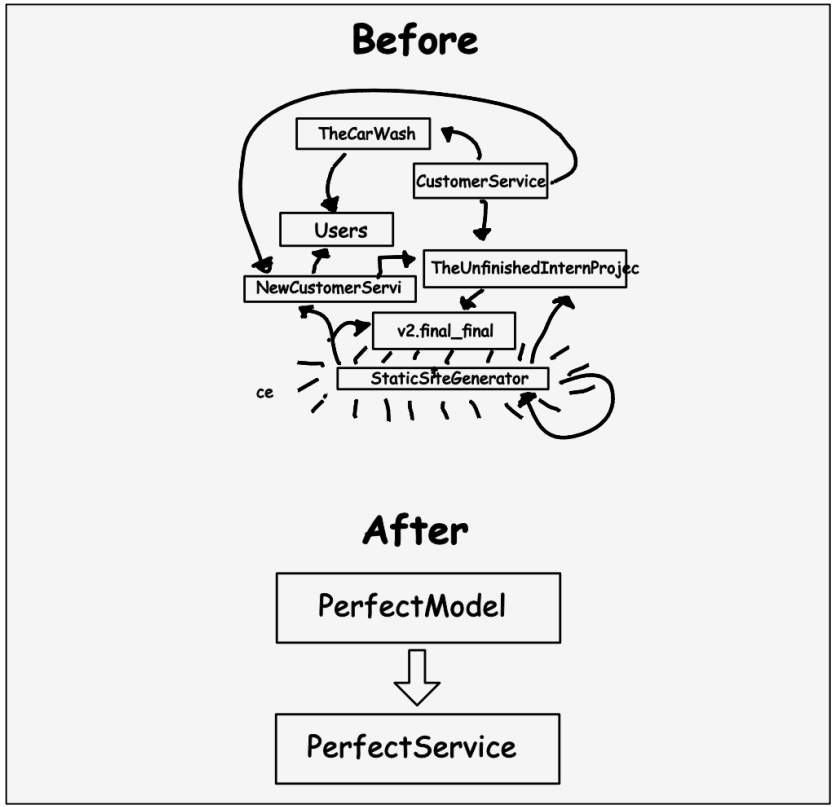
No, it won't be.



2. We will move faster with clean code.

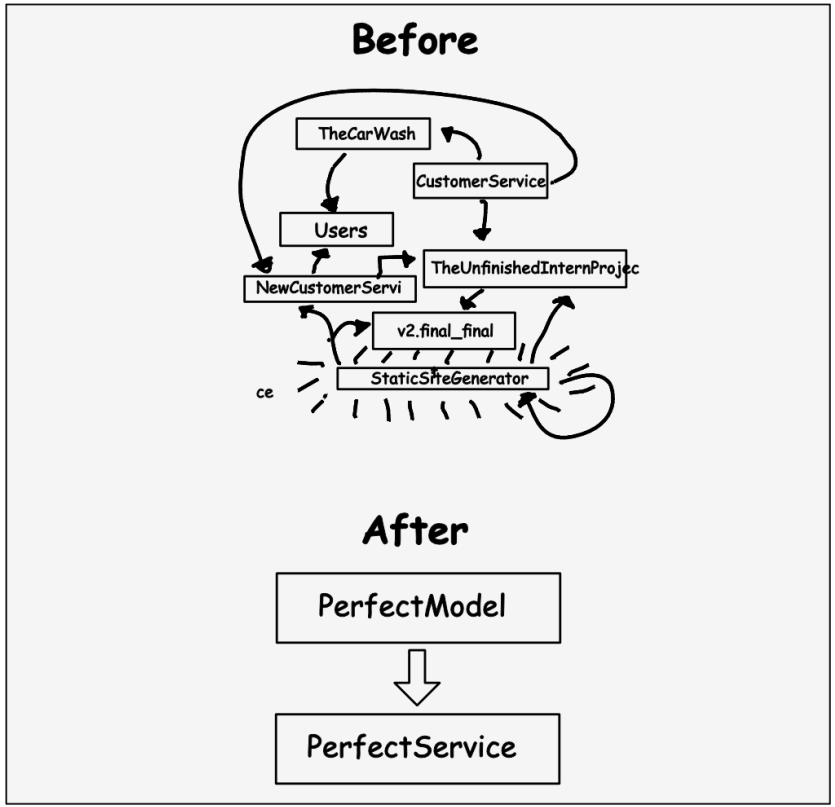


2. We will move faster with clean code.



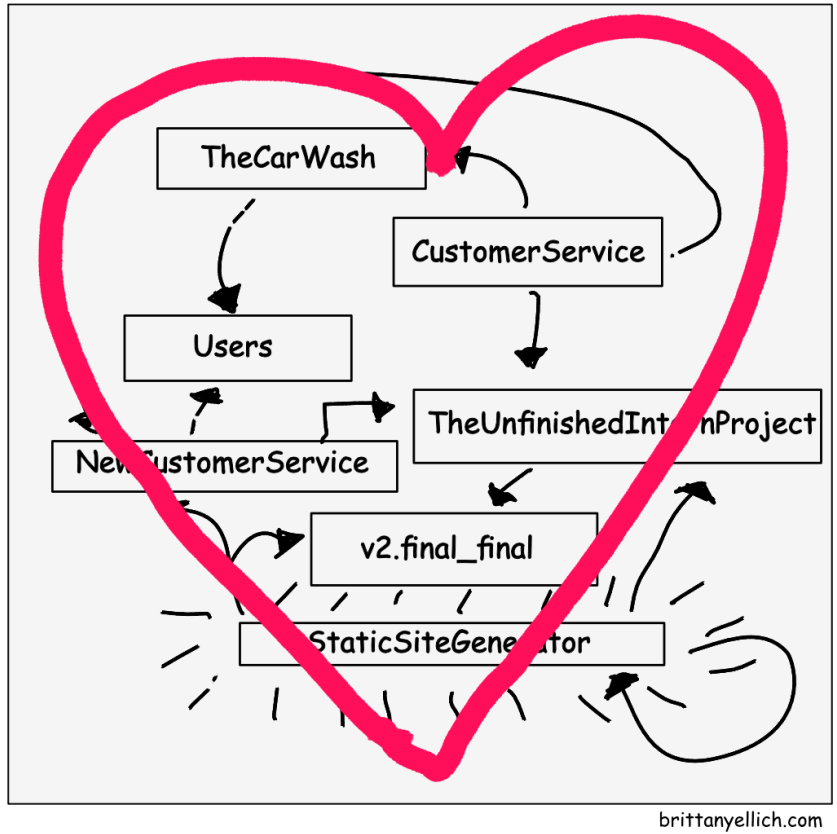
2. We will move faster with clean code.

- "If we start over, we can do it right this time"
- "We'll avoid all the mistakes from the legacy system"
- "The new code will be so much cleaner and more maintainable"



2. We will move faster with clean code.

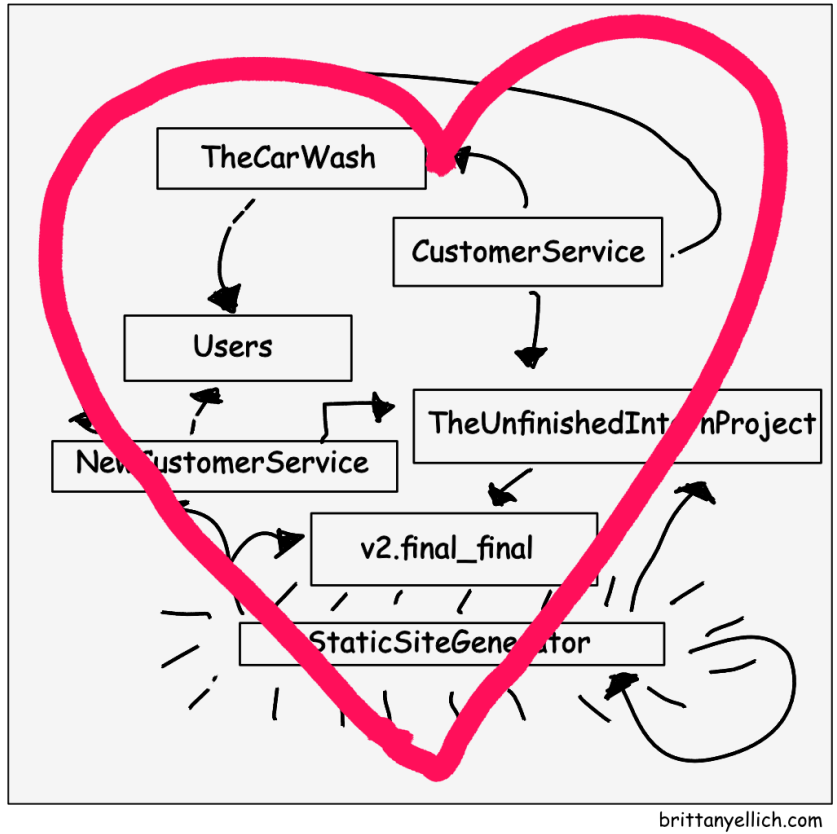
Production applications are inherently messy - and that's not a bug, it's a feature.



2. We will move faster with clean code.

Production applications are inherently messy - and that’s not a bug, it’s a feature.

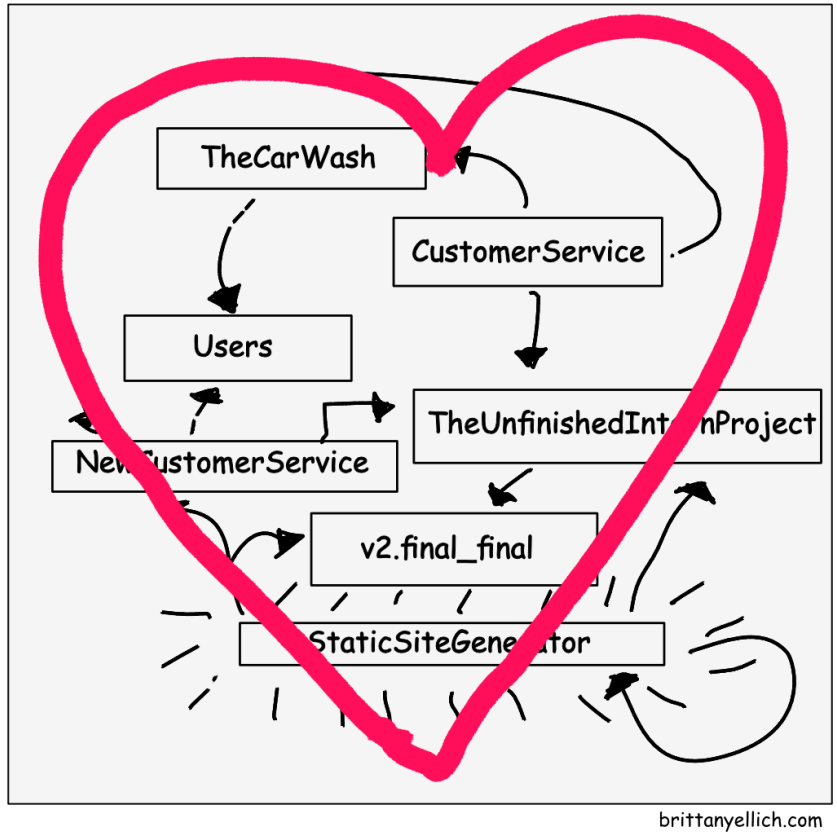
- Your "ugly" edge cases = real business requirements discovered through user feedback



2. We will move faster with clean code.

Production applications are inherently messy - and that’s not a bug, it’s a feature.

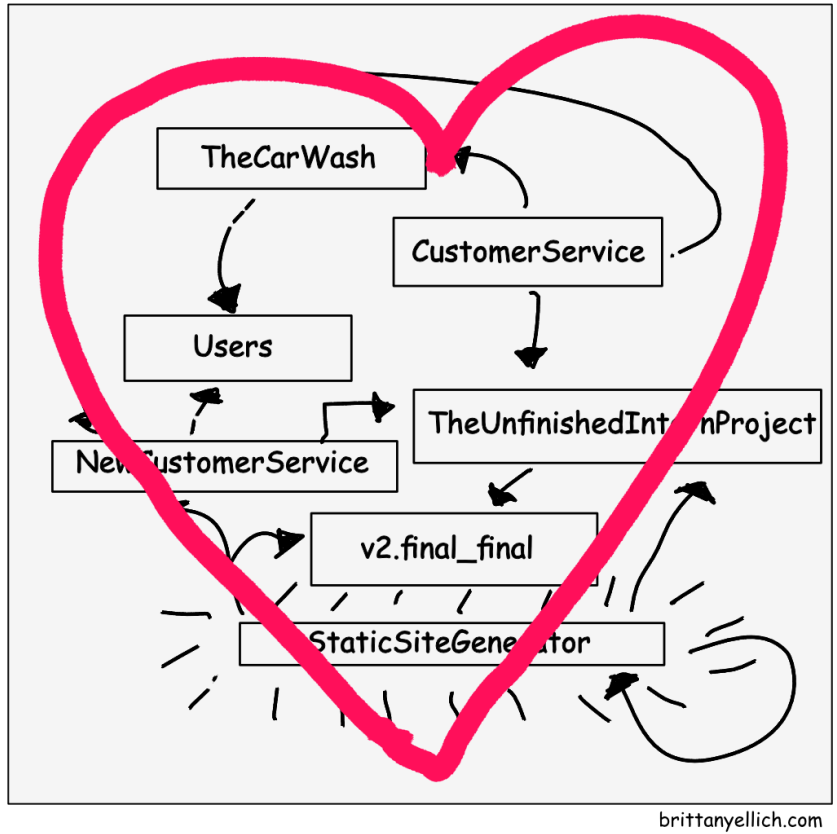
- Your "ugly" edge cases = real business requirements discovered through user feedback
- Workarounds = institutional knowledge



2. We will move faster with clean code.

Production applications are inherently messy - and that's not a bug, it's a feature.

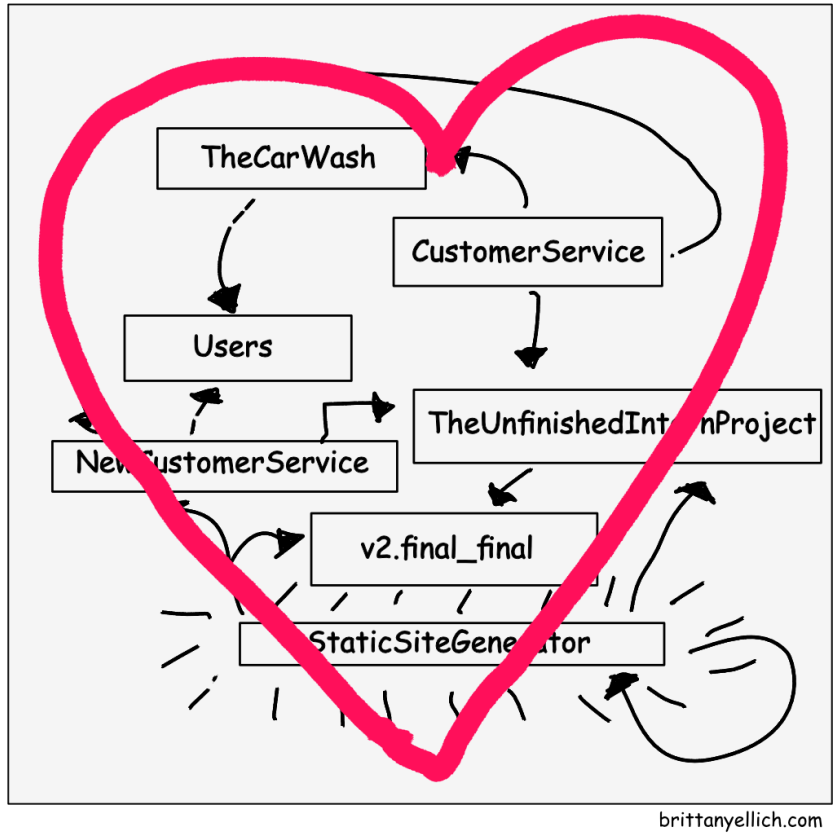
- Your "ugly" edge cases = real business requirements discovered through user feedback
- Workarounds = institutional knowledge
- Weird conditionals = patches for critical customer scenarios from a 2AM outage



2. We will move faster with clean code.

Production applications are inherently messy - and that's not a bug, it's a feature.

- Your "ugly" edge cases = real business requirements discovered through user feedback
- Workarounds = institutional knowledge
- Weird conditionals = patches for critical customer scenarios from a 2AM outage
- "Unnecessary" features = critical dependencies for actual users



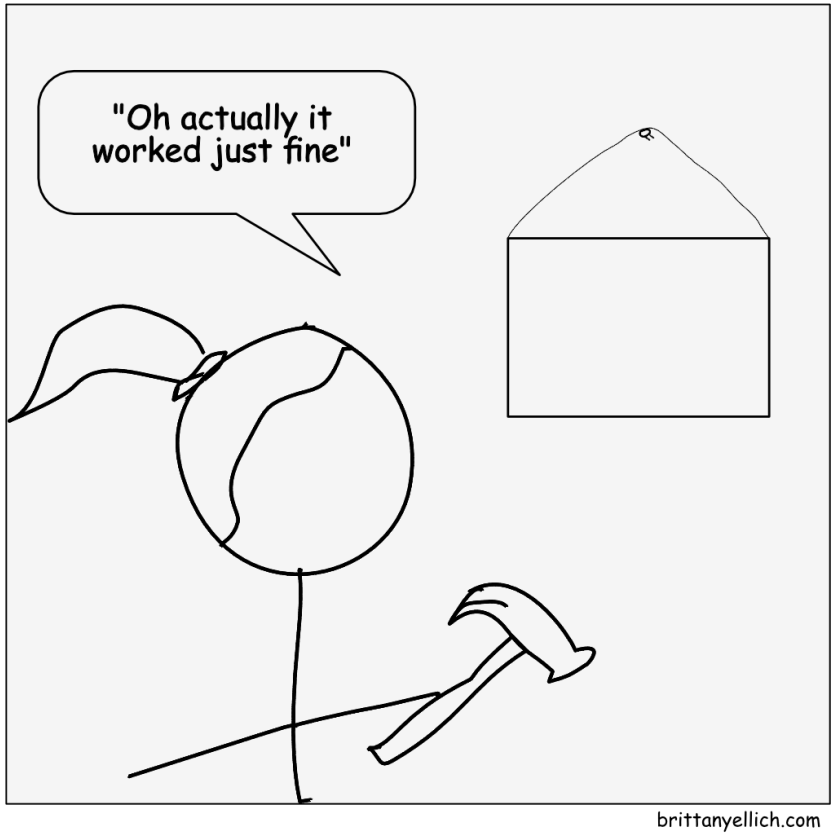
3. The current technology stack is holding us back.

3. The current technology stack is holding us back.

- "If we rebuild in X, we will solve all our performance problems"
- "We're spending too much time fighting against outdated tools"
- "New developers can't be productive in this legacy stack"

3. The current technology stack is holding us back.

It isn’t.



Refactor, don’t rewrite.

Refactor, don’t rewrite.

You don't need a rewrite.

Refactor, don’t rewrite.

You don't need a rewrite.

It will take too long.

Refactor, don’t rewrite.

You don't need a rewrite.

It will take too long.

All production applications are messy.

Refactor, don’t rewrite.

You don't need a rewrite.

It will take too long.

All production applications are messy.

You won't suddenly work better by adopting a new technology.

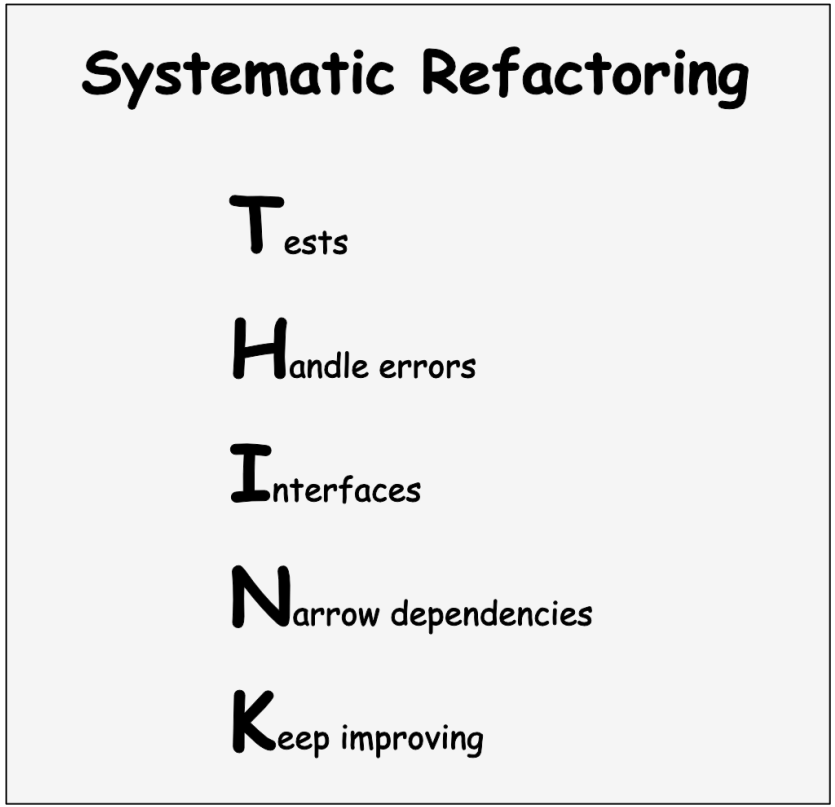
The best software products emerge from continuous improvement, not perfect planning.

So how do we identify what to refactor?

How to refactor

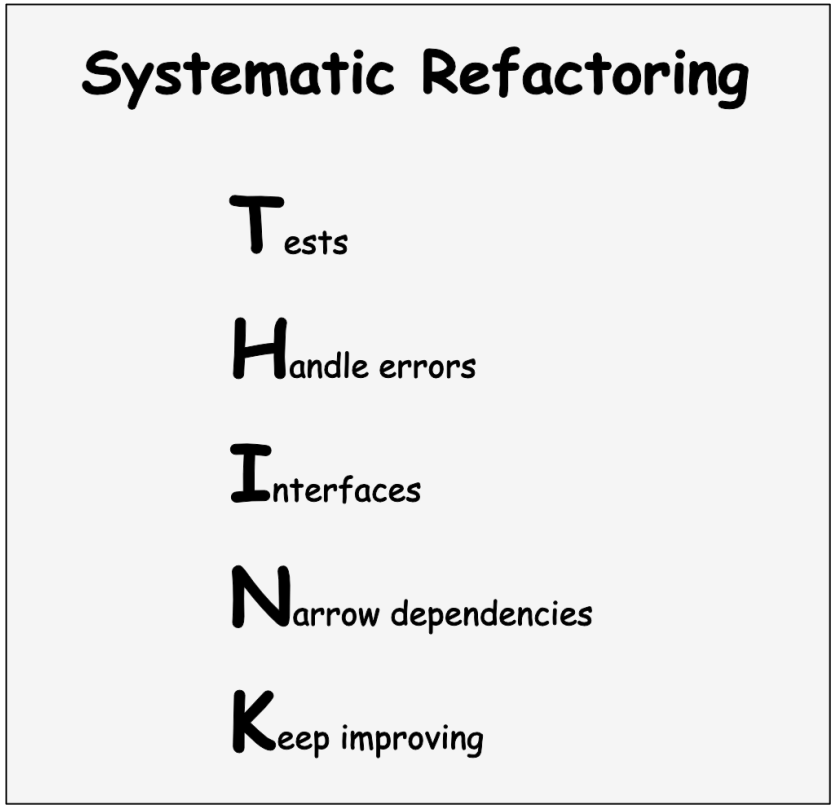
A systematic refactoring method

THINK: A Systematic Refactoring Mindset



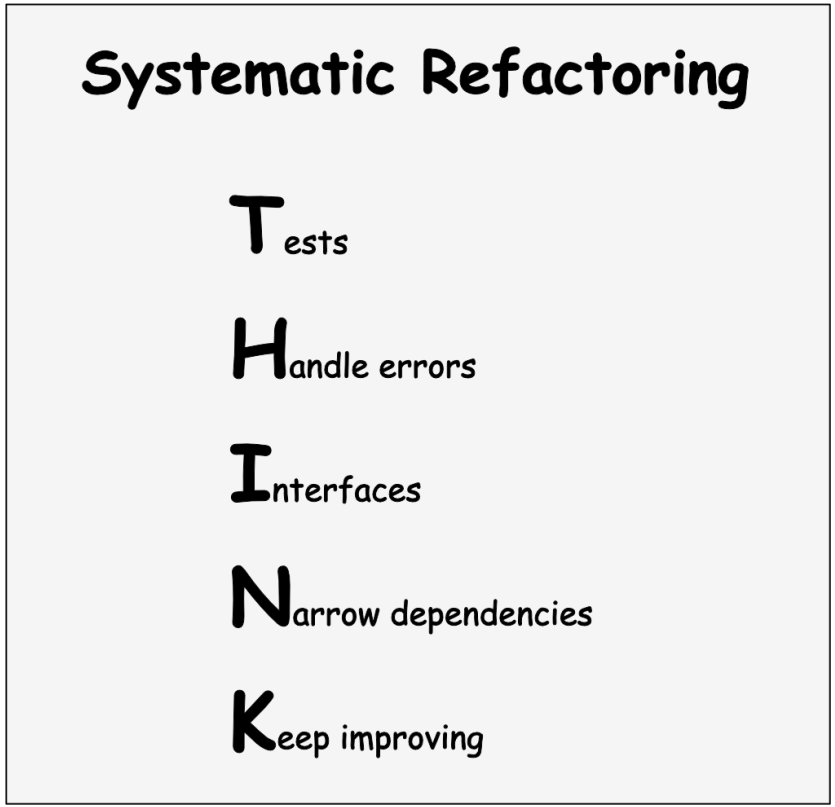
THINK: A Systematic Refactoring Mindset

- Tests: A safety net before changing anything



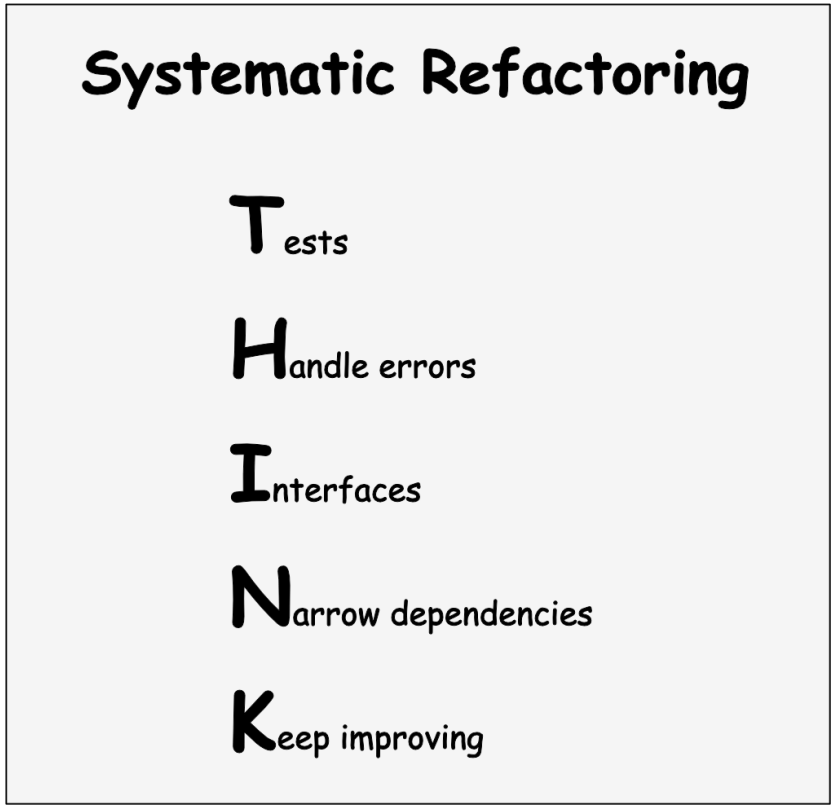
THINK: A Systematic Refactoring Mindset

- Tests: A safety net before changing anything
- Handle errors: This affects the rest of your application's structure



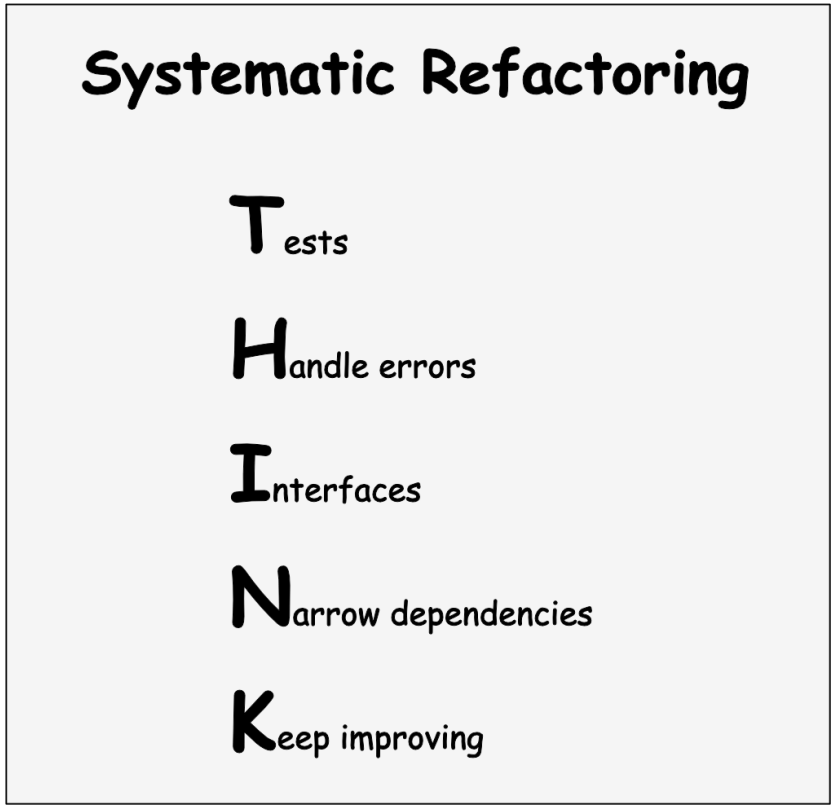
THINK: A Systematic Refactoring Mindset

- Tests: A safety net before changing anything
- Handle errors: This affects the rest of your application's structure
- Interfaces: These define your system's boundaries



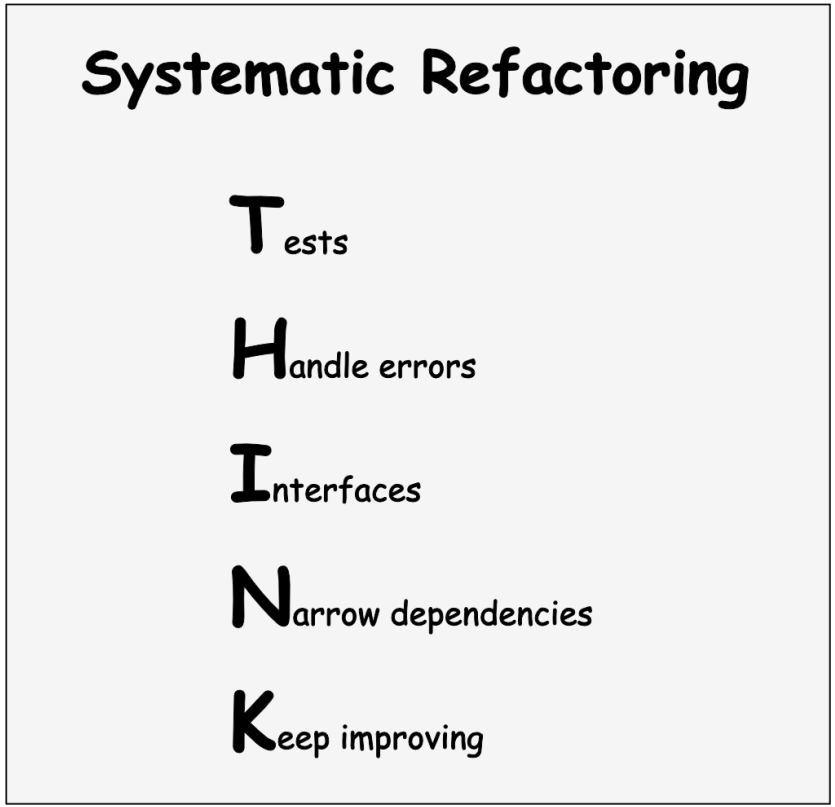
THINK: A Systematic Refactoring Mindset

- Tests: A safety net before changing anything
- Handle errors: This affects the rest of your application's structure
- Interfaces: These define your system's boundaries
- Narrow dependencies: Remove library dependencies (where possible), reduce hard-coding, use dependency injection

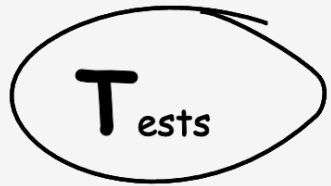


THINK: A Systematic Refactoring Mindset

- Tests: A safety net before changing anything
- Handle errors: This affects the rest of your application's structure
- Interfaces: These define your system's boundaries
- Narrow dependencies: Remove library dependencies (where possible), reduce hard-coding, use dependency injection
- Keep improving: Continuous improvement > dramatic changes



Systematic Refactoring



Handle errors

Interfaces

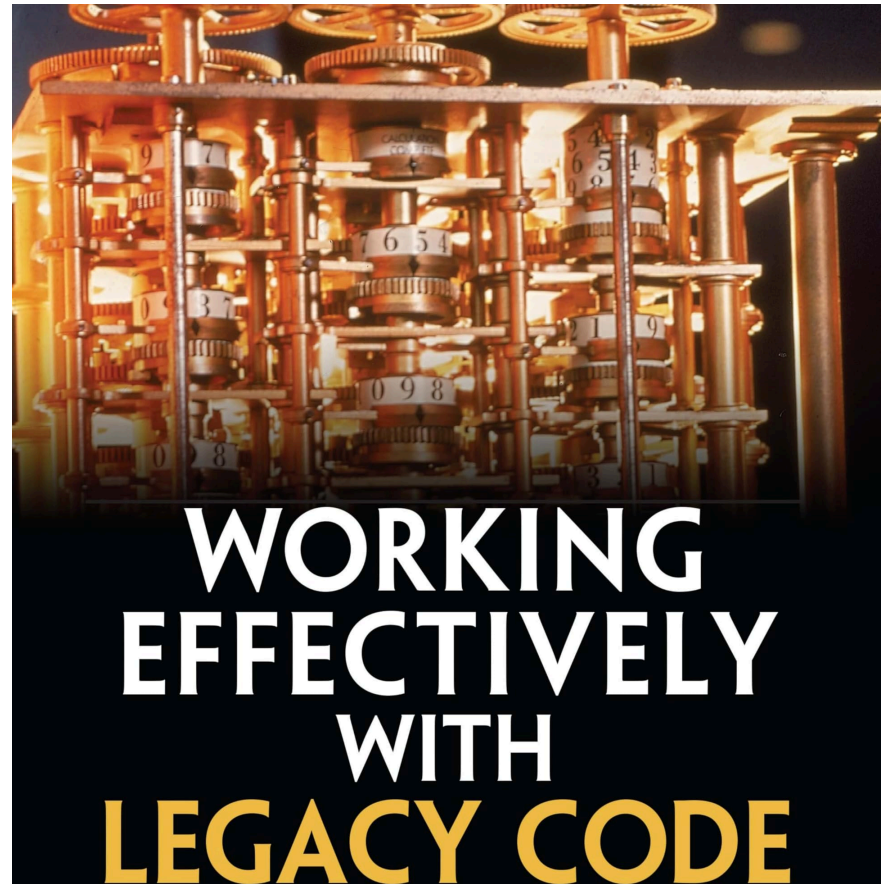
Narrow dependencies

Keep improving

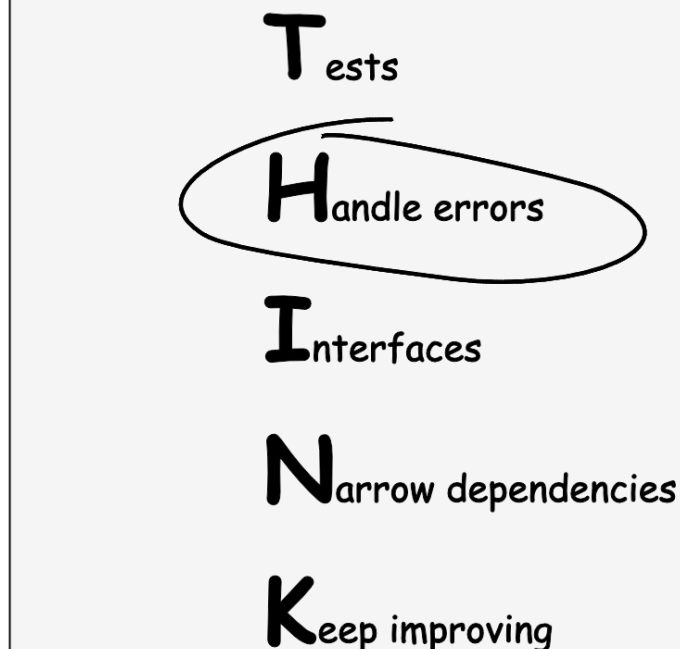
brittanyellich.com

Tests

- A safety net before changing anything
- A guide to the refactoring process
- Add characterization tests if you’re missing tests
- Working Effectively with Legacy Code by Michael Feathers



Systematic Refactoring



brittanyellich.com

Handle errors

The Problem: Lost context, double handling (logging and returning an error), silent failures (no error handling at all)

```
// Bad: No context, inconsistent patterns
return errors.New("invalid amount")

// Good: Clear context and consistent wrapping
return fmt.Errorf("process payment: invalid amount %.2f", amount)
```

Go Proverb: "Errors are values" - program with them, don't just check them

Reading: "100 Go Mistakes" #49-52 on error handling patterns

Systematic Refactoring

Tests

Handle errors

Interfaces

Narrow dependencies

Keep improving

brittanyellich.com

Interfaces

The Problem: Large interfaces force clients to depend on methods they don't use

```
// Bad: Clients depend on methods they don't use
type UserService interface {
    CreateUser(u User) error
    SendEmail(to string) error
    ...
}

// Good: Focused, testable interfaces
type UserManager interface {
    CreateUser(u User) error
}
type NotificationService interface {
    SendEmail(to, subject, body string) error
}
```

Go Proverb: "The bigger the interface, the weaker the abstraction"

Reading: "100 Go Mistakes" #5-7 on interface design

Interfaces

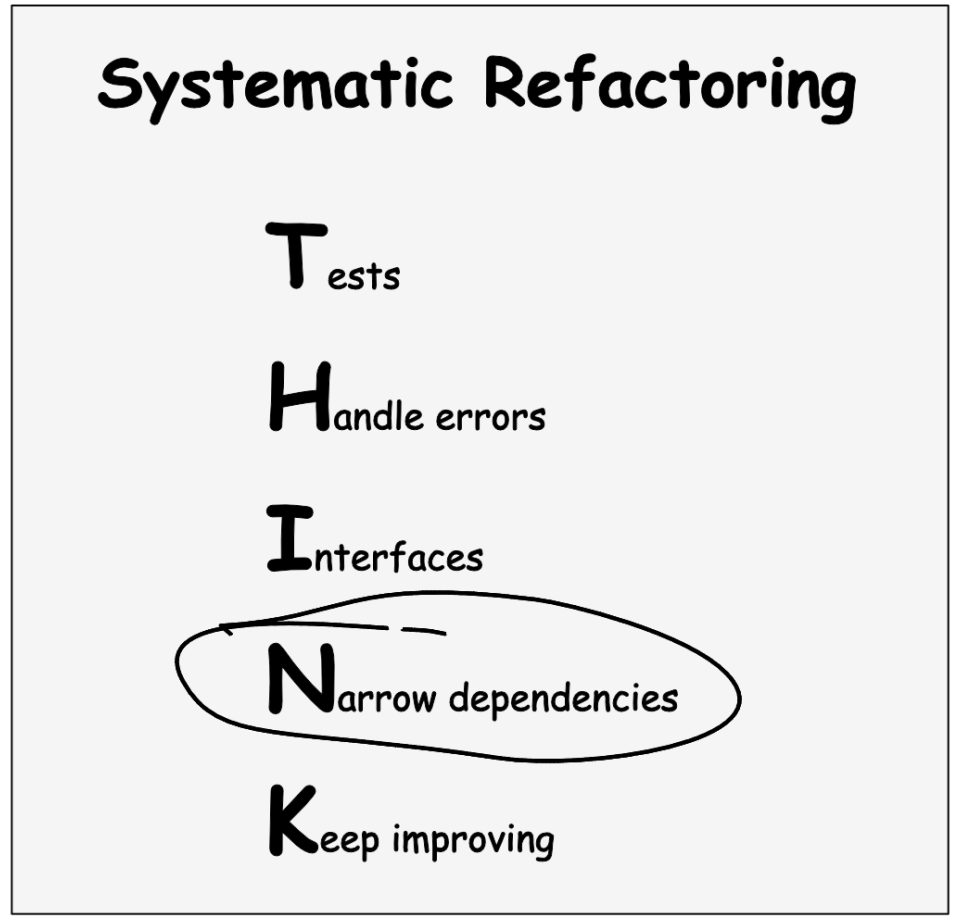
The Problem: Using any/interface when specific types would be better

```
// Bad: Runtime type checking required
func Process(data interface{}) error

// Good: Specific interfaces or generics
func Process[T Processor](data T) error
```

Go Proverb: "The empty interface says nothing"**Reading**: "100 Go Mistakes" #8 on any usage

Systematic Refactoring



brittanyellich.com

Narrow dependencies

The Problem: Adding direct dependencies increase coupling and make things hard to test

```
// Bad: Hard to test, tightly coupled
func ProcessOrder(customerID string, items []string) error {
    db := sql.Open("postgres", "...") // Direct dependency
    // ... processing logic
}

// Good: Dependency injection
type OrderService struct {
    repository OrderRepository
}
func (s *OrderService) ProcessOrder(req OrderRequest) error
```

Why prioritize: Make testing easier

Narrow dependencies

The Problem: Single structs mixing API responses, database, and validation concerns

```
// Bad: Mixed concerns
type User struct {
    ID       int    `json:"id" db:"user_id" validate:"required"`
    Name     string `json:"name" db:"full_name" validate:"min=2,max=50"`
    Email    string `json:"email" db:"email_addr" validate:"email"`
    Password string `json:"-" db:"password_hash"`
}

// Good: Separated concerns
type UserAPI struct {     // API representation
    ID    int    `json:"id"`
    Name  string `json:"name"`
    Email string `json:"email"`
}
type UserDB struct {      // Database model
    ID       int    `db:"user_id"`
    Name     string `db:"full_name"`
    Email    string `db:"email_addr"`
    Password string `db:"password_hash"`
}
```

Narrow dependencies

The Problem: Hard-coded business rule changes require code deployments instead of configuration updates

```
// Bad: Business logic embedded in code
func CalculateDiscount(total float64, customerType string) float64 {
    if customerType == "premium" {
        return total * 0.15  // Hardcoded 15%
    }
    return 0
}

// Good: Configurable service
type DiscountService interface {
    CalculateDiscount(total float64, customer Customer) float64
}
```

Why prioritize: Make it easy to change your application

Systematic Refactoring

Tests

Handle errors

Interfaces

Narrow dependencies

Keep improving

brittanyellich.com

Keep improving

Keep improving

Refactor as you go

Keep improving

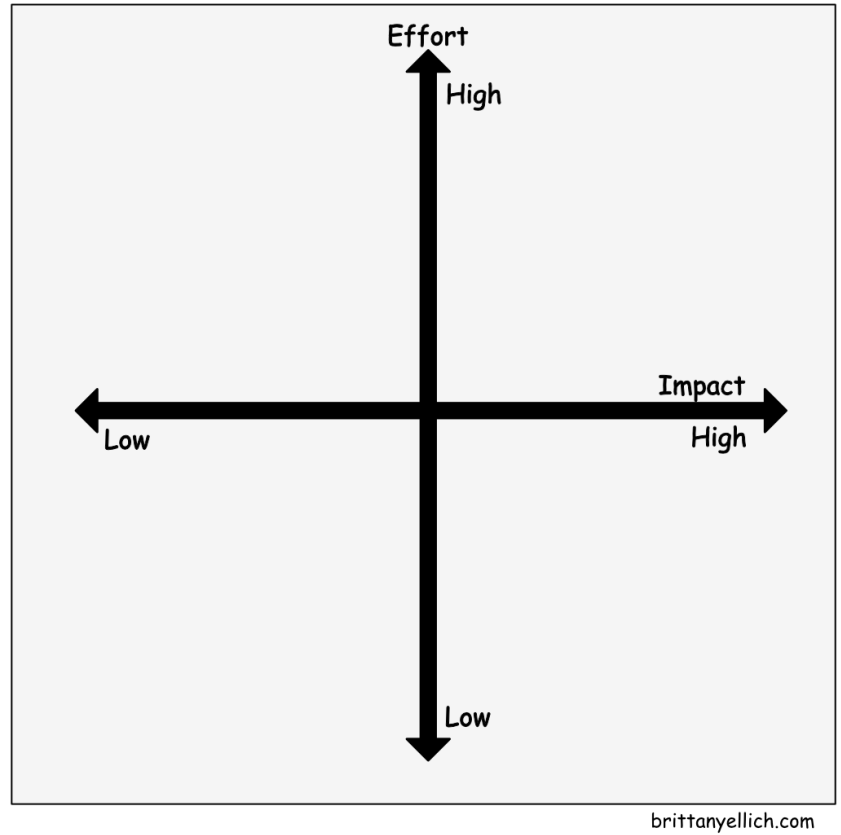
Refactor as you go

Continuously iterate and deliver value

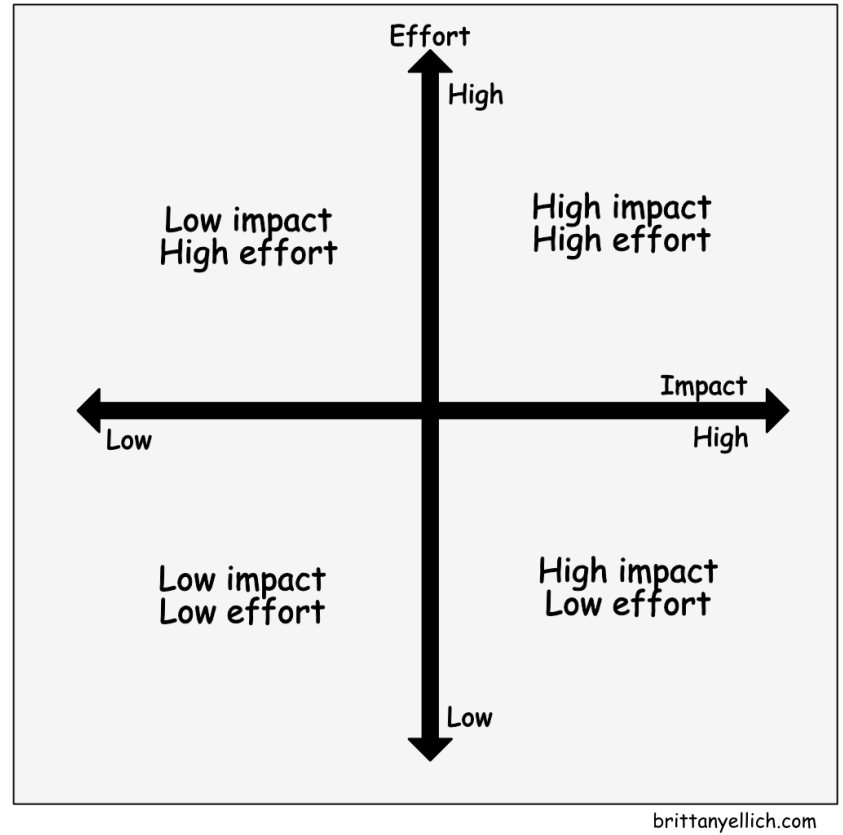
The best software products emerge from continuous improvement, not perfect planning.

How to prioritize your refactoring efforts

A prioritization framework



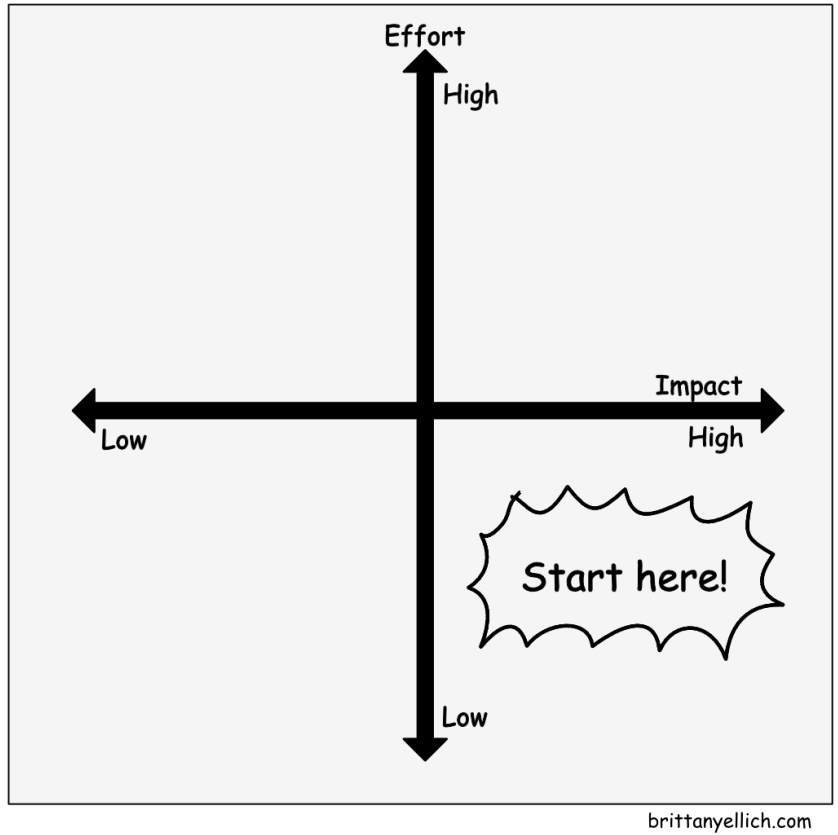
A prioritization framework



A prioritization framework

The Three-Bucket Approach

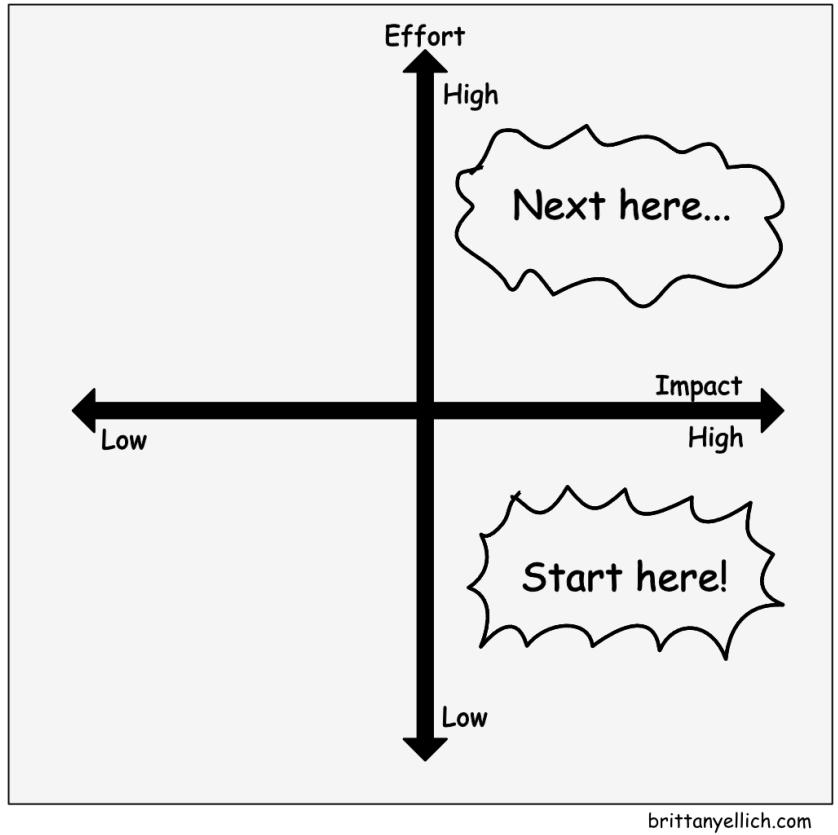
- High Impact, Low Effort: Start here



A prioritization framework

The Three-Bucket Approach

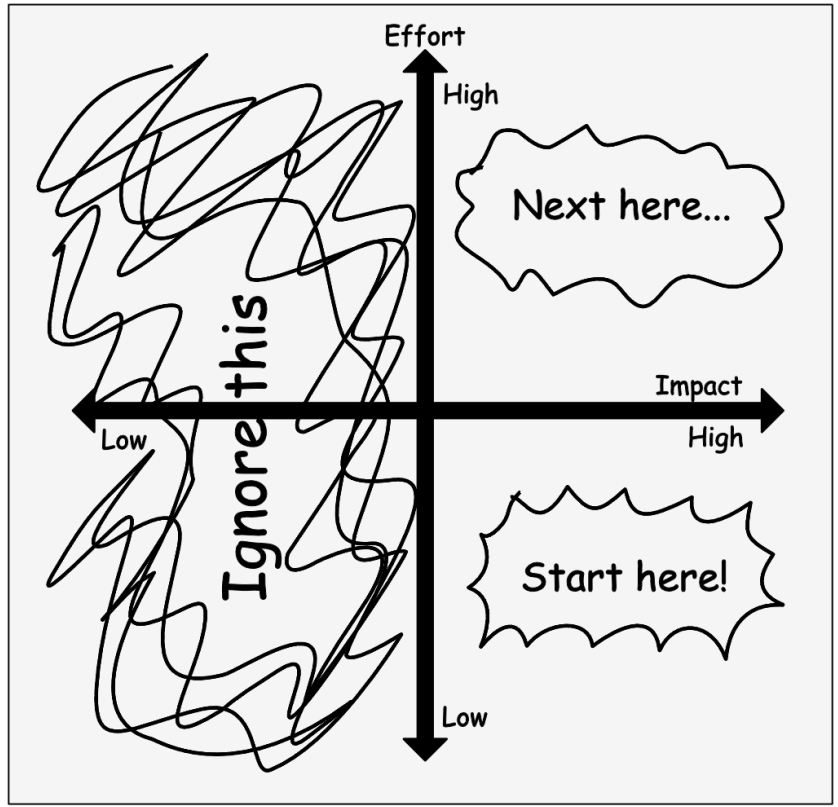
- High Impact, Low Effort: Start here
- High Impact, High Effort: Plan next



A prioritization framework

The Three-Bucket Approach

- High Impact, Low Effort: Start here
- High Impact, High Effort: Plan next
- Low Impact, Any Effort: Ignore this



How to get AI to help you

Coding agent use cases

Coding agent use cases

- Increasing test coverage

Coding agent use cases

- Increasing test coverage
- Searching your codebase to find inconsistent approaches

Coding agent use cases

- Increasing test coverage
- Searching your codebase to find inconsistent approaches
- Taking patterns and applying them across your codebase incrementally

Coding agent use cases

- Increasing test coverage
- Searching your codebase to find inconsistent approaches
- Taking patterns and applying them across your codebase incrementally
- How the GitHub billing team uses the coding agent in GitHub Copilot to continuously burn down technical debt

Increasing test coverage with the coding agent in GitHub Copilot

> In order to build more confidence in our code changes, we should try to get as close to 100% as possible in our test coverage. Search the test files in the lib/services directory and identify if there are any paths that aren't currently tested. Create table-driven tests to cover those scenarios.

Dave Cheney: Prefer Table-Driven Tests

Searching your codebase to find inconsistent approaches with the coding agent in GitHub Copilot

> Use this PR as an example to swap from the existing mocking library to this new one for mocks in the lib/services directory.

Taking patterns and applying them across your codebase incrementally

> We would like to standardize on the following error handling specification for errors in the lib/services directory. Help me identify anywhere that we are not currently handling errors in this way in this directory and update them to use this syntax, with reasonable error messages to support them.

```
return fmt.Errorf("failed to process payment for order %s: %w", orderID, err)
```

Recap

- Refactor, don’t rewrite: The top 3 arguments and why they’re wrong
- How to refactor: A systematic refactoring method
- How to prioritize your refactoring efforts: Focus on impact
- Getting AI to help you: How to use agents today

The best software products emerge from continuous improvement, not perfect planning.

Resources

- Book: Working Effectively with Legacy Code - Michael Feathers
- Book: 100 Go Mistakes - Teiva Harsanyi
- Video: Go Proverbs
- Article: Dave Cheney: Prefer Table-Driven Tests
- Article: How the GitHub billing team uses the coding agent in GitHub Copilot to continuously burn down technical debt

Acknowledgements

- To my incredible partner who is hanging out with our three young kids while I travel
- To my GitHub colleagues who have given me so much feedback, advice, and motivation
- To my brothers, including Brad Heller who is here today, who are my fellow software engineers and supporters

Thank you!

My website is brittanyellich.com, let’s be internet friends!