ADVANCED REFACTORING PATTERNS & PRACTICES

Saxonia Systems
So geht Software.

Der Sprecher

Hendrik Lösch

Senior Consultant & Coach

Hendrik.Loesch@saxsys.de

@HerrLoesch

Just-About.Net

Hendrik-Loesch.de

Grundlagen der Programmierung: Test Driven Development

Business-Applikationen testgetrieben entwickeln

Grundlagen der Programmierung: Codemetriken

Softwarequalität einschätzen, sicherstellen und ...

Inversion of Control und Dependency Injection – Grundlagen

Prinzipien der modernen Software-Architektur ...

WPF-Anwendungen mit MVVM und Prism

Modulare Architekturen verstehen und umsetzen

LEARNING

Saxonia Systems
So geht Software.

Debugging at its best

Was ist schlechtes Design?!?

1. It is hard to change because every change affects too many other parts of the system. (Rigidity - Starr)

2. When you make a change, unexpected parts of the system break. (Fragility - Zerbrechlich)

3. It is hard to reuse in another application because it cannot be disentangled from the current application. (Immobility - Unbeweglich)

Quelle: http://www.objectmentor.com/resources/articles/dip.pdf

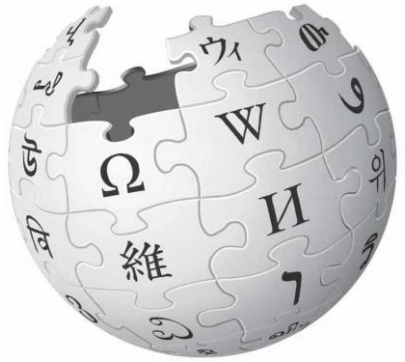
Refactoring

VORGEHEN

Saxonia Systems

So geht Software.

Refactoring

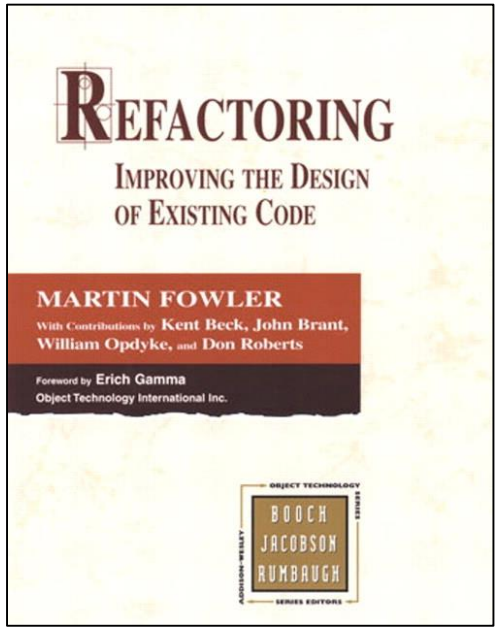


Refactoring bezeichnet in der Software-Entwicklung die manuelle oder automatisierte **Strukturverbesserung von Quelltexten unter Beibehaltung des beobachtbaren Programmverhaltens**.

Dabei sollen die Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit verbessert werden, mit dem Ziel, den jeweiligen Aufwand für Fehleranalyse und funktionale Erweiterungen deutlich zu senken.

Quelle: https://de.wikipedia.org/wiki/Refactoring

Refactoring

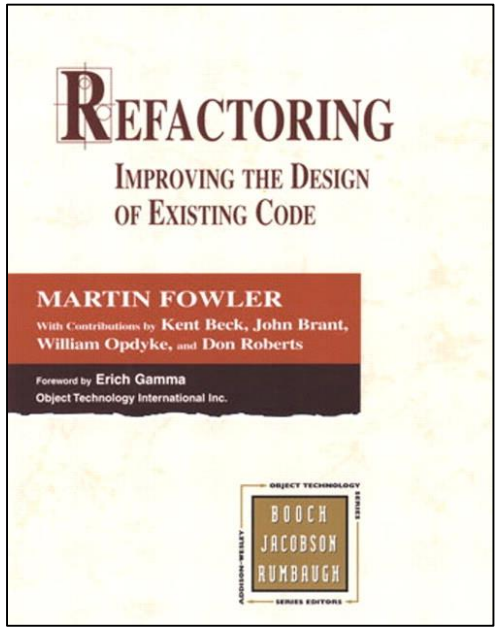


refactoring.com

…is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Its heart is a series of small behavior preserving transformations. ***Each transformation (called a "refactoring") does little, but a sequence of transformations can produce a significant restructuring****. Since each refactoring is small, it's less likely to go wrong. The system is kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.*

Refactoring Arten



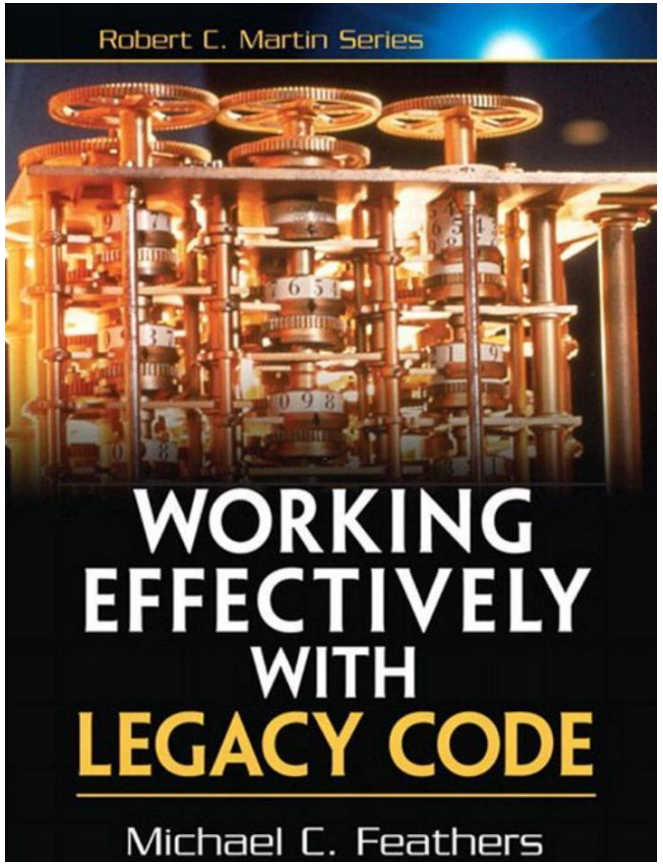
refactoring.com

Add Parameter
Change Bidirectional Association to Unidirectional
Change Reference to Value
Change Unidirectional Association to Bidirectional
Change Value to Reference
Collapse Hierarchy
Consolidate Conditional Expression
Consolidate Duplicate Conditional Fragments
Decompose Conditional
Duplicate Observed Data
Dynamic Method Definition
Eagerly Initialized Attribute
Encapsulate Collection
Encapsulate Downcast
Encapsulate Field
Extract Class
Extract Interface
Extract Method
Extract Module
Extract Subclass
Extract Superclass
Extract Surrounding Method
Extract Variable
Form Template Method
Hide Delegate
Hide Method
Inline Class
Inline Method
Inline Module
Inline Temp
Introduce Assertion
Introduce Class Annotation
Introduce Expression Builder
Introduce Foreign Method
Introduce Gateway
Introduce Local Extension
Introduce Named Parameter
Introduce Null Object
Introduce Parameter Object
Isolate Dynamic Receptor
Lazily Initialized Attribute
Move Eval from Runtime to Parse Time
Move Field
Move Method
Parameterize Method
Preserve Whole Object
Pull Up Constructor Body
Pull Up Field
Pull Up Method
Push Down Field
Push Down Method
Recompose Conditional
Remove Assignments to Parameters
Remove Control Flag
Remove Middle Man
Remove Named Parameter
Remove Parameter
Remove Setting Method
Remove Unused Default Parameter
Rename Method
Replace Abstract Superclass with Module
Replace Array with Object
Replace Conditional with Polymorphism
Replace Constructor with Factory Method
Replace Data Value with Object
Replace Delegation With Hierarchy
Replace Delegation with Inheritance
Replace Dynamic Receptor with Dynamic Method Definition
Replace Error Code with Exception
Replace Exception with Test
Replace Hash with Object
Replace Inheritance with Delegation
Replace Loop with Collection Closure Method
Replace Magic Number with Symbolic Constant
Replace Method with Method Object
Replace Nested Conditional with Guard Clauses
Replace Parameter with Explicit Methods
Replace Parameter with Method
Replace Record with Data Class
Replace Subclass with Fields
Replace Temp with Chain
Replace Temp with Query
Replace Type Code with Class
Replace Type Code with Module Extension
Replace Type Code With Polymorphism
Replace Type Code with State/Strategy
Replace Type Code with Subclasses
Self Encapsulate Field
Separate Query from Modifier
Split Temporary Variable
Substitute Algorithm

Saxonia Systems
So geht Software.

Legacy Code

“Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.”



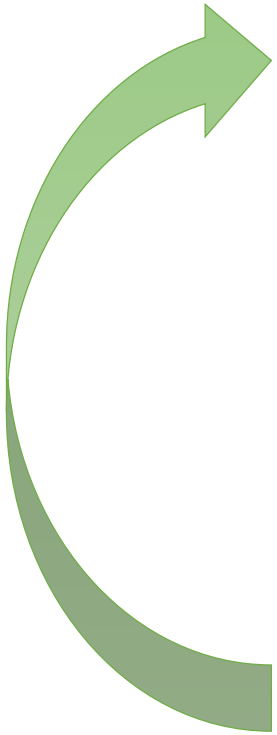
Probleme während des Refaktorisierens

Refactoring

PRACTICES

Saxonia Systems
So geht Software.

Wie vorgehen?



1. Quellcode einchecken.
2. Einen Überblick darüber verschaffen was geändert werden soll.
3. Mindestens einen Test schreiben der das bestehende Verhalten charakterisiert.

4. **Refaktorisieren**

5. Prüfen ob sich das Verhalten verändert hat.

Sokrates-Test

Situationen

„Ich weiß nicht was mein Code macht, soll diesen Zustand aber ändern.“

Vorgehen

Wir schreiben einen Test ohne Asserts und sehen uns anhand der Codeabdeckung an was passiert.

Charakterisierungstest

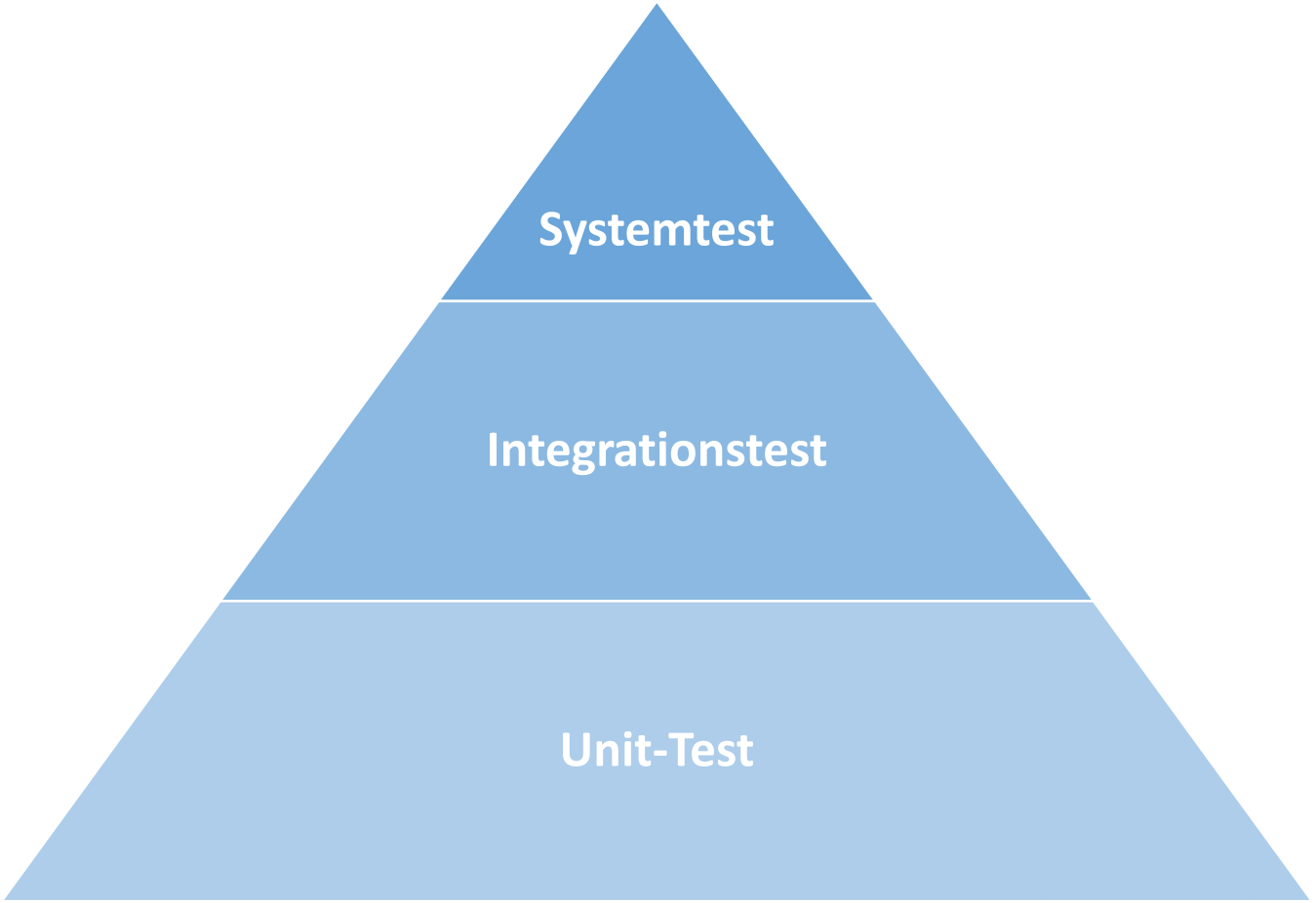
Situationen

„Ich will das mein Code sich nach den Änderungen so verhält wie vorher.“

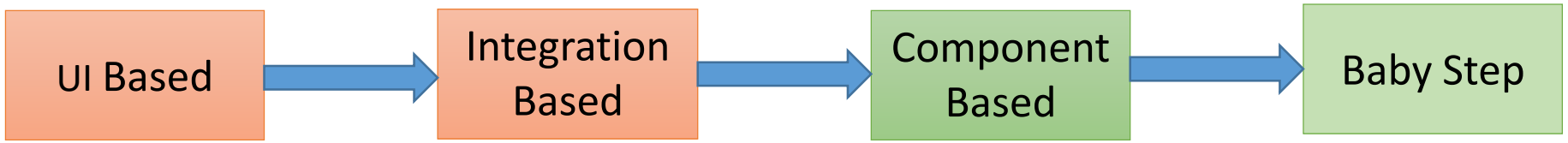
Vorgehen

Man schreibt oder generiert einen oder mehrere Tests um das Verhalten zu prüfen.

Charakterisierungstest



Testvorgehen



Refactoring

PATTERN

Saxonia Systems

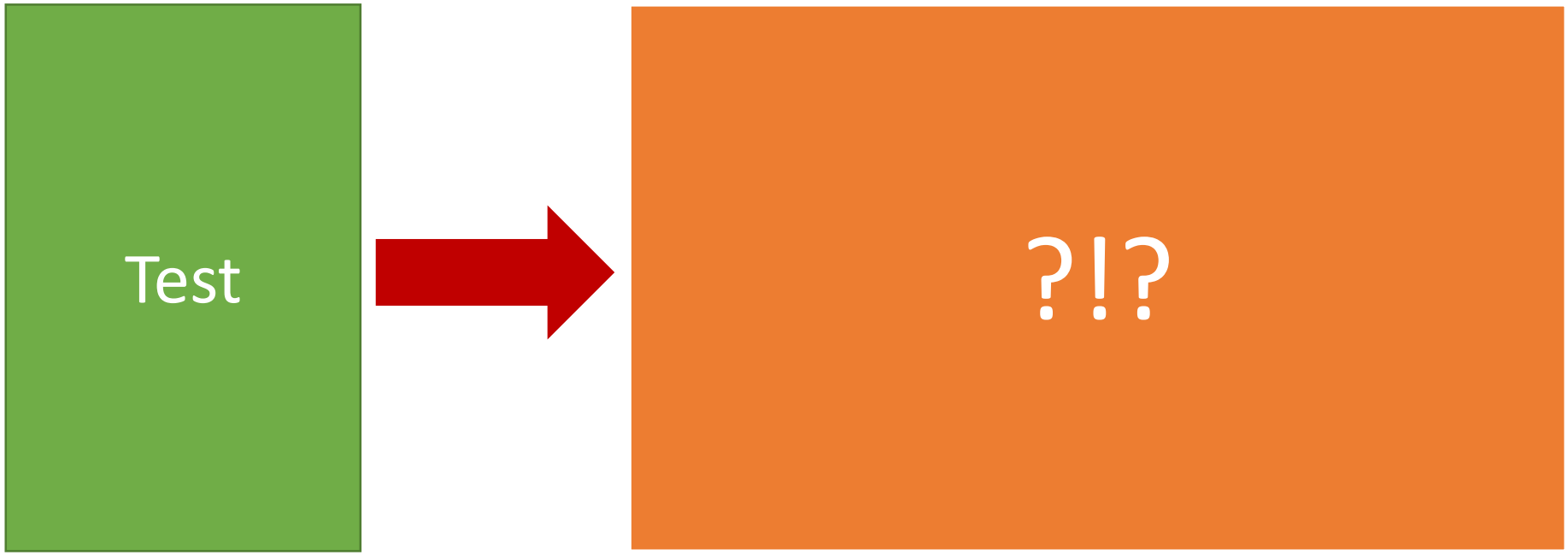
So geht Software.

Accessor

Situationen

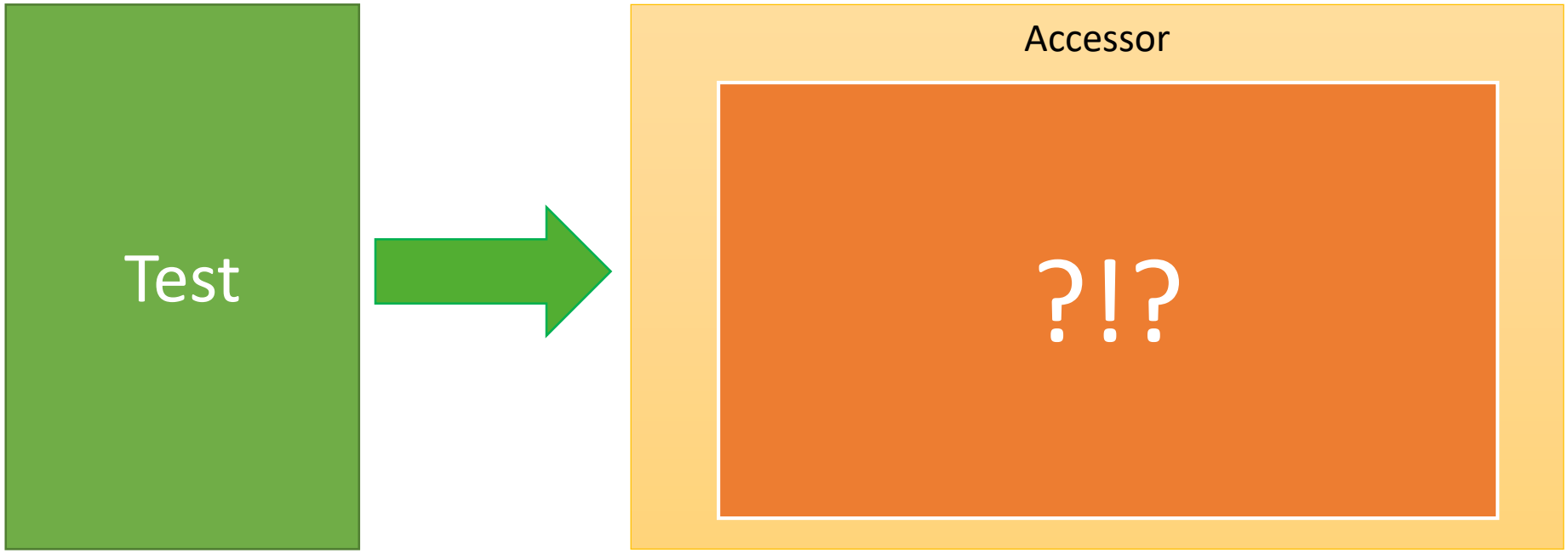
„Ich muss Interna einer Klasse manipulieren um etwas testen zu können, will dies aber nur für die Tests tun.“

Accessor



Accessor

Unbedingt nur als 1:1 Beziehung umsetzen!!!



Accessor

Vorteile

- Kann untestbaren Code testbar machen.
- Macht Tests robuster gegenüber Änderungen an der Implementierung

Nachteile

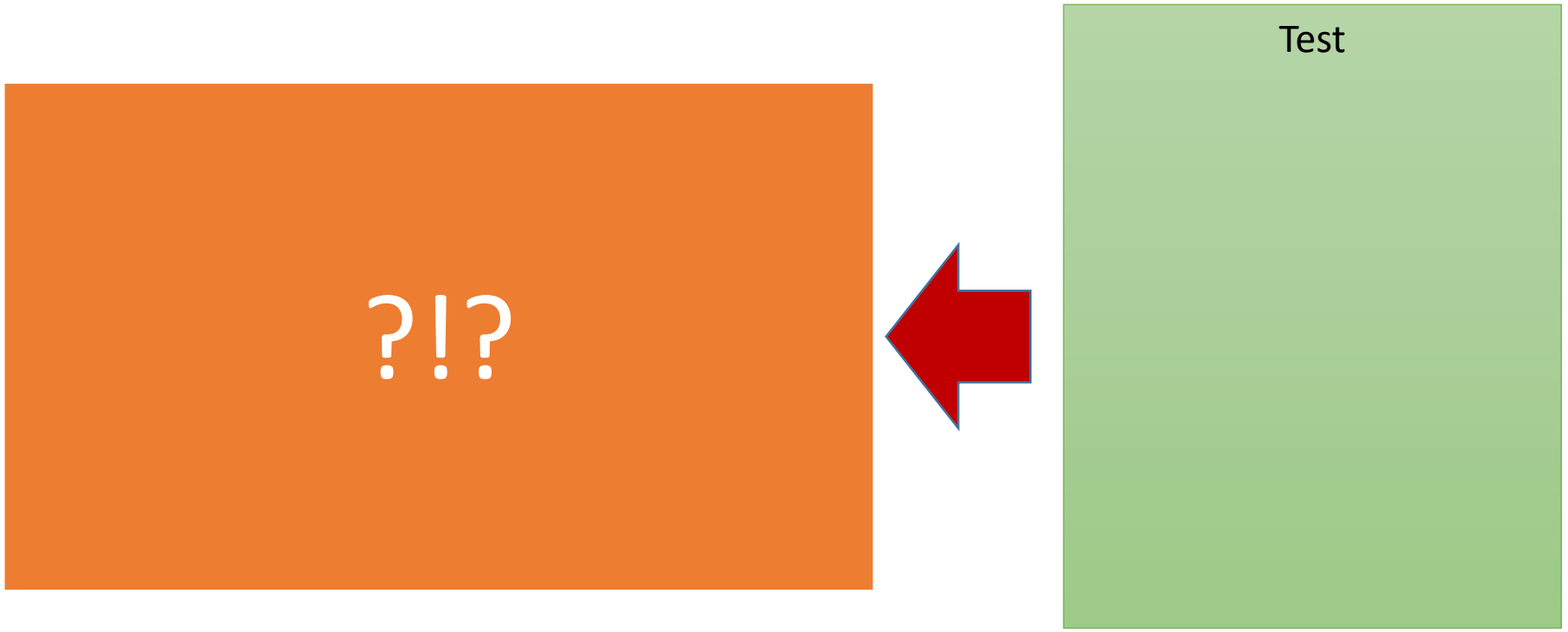
- Wird die Implementierung geändert muss auch der Accessor angepasst werden.
- Tests können fehleranfällig sein, da nebenläufige Abhängigkeiten nicht betrachtet werden.

Sprout

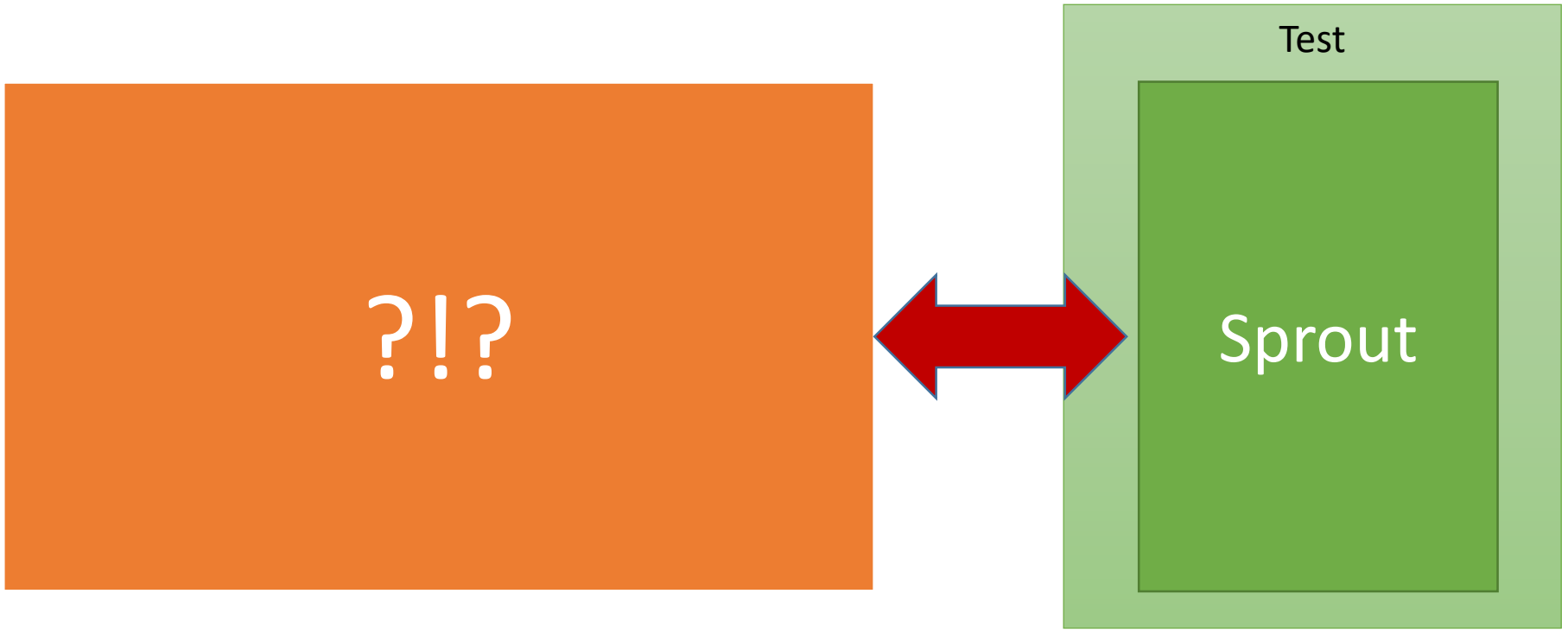
Situationen

„Wir haben keine Zeit für umfassende Refactorings.“
„Ich möchte Code testgetrieben in einer untestbaren Umgebung umsetzen.“
„Ich kann das Problem durch reines Hinzufügen von Code lösen.“

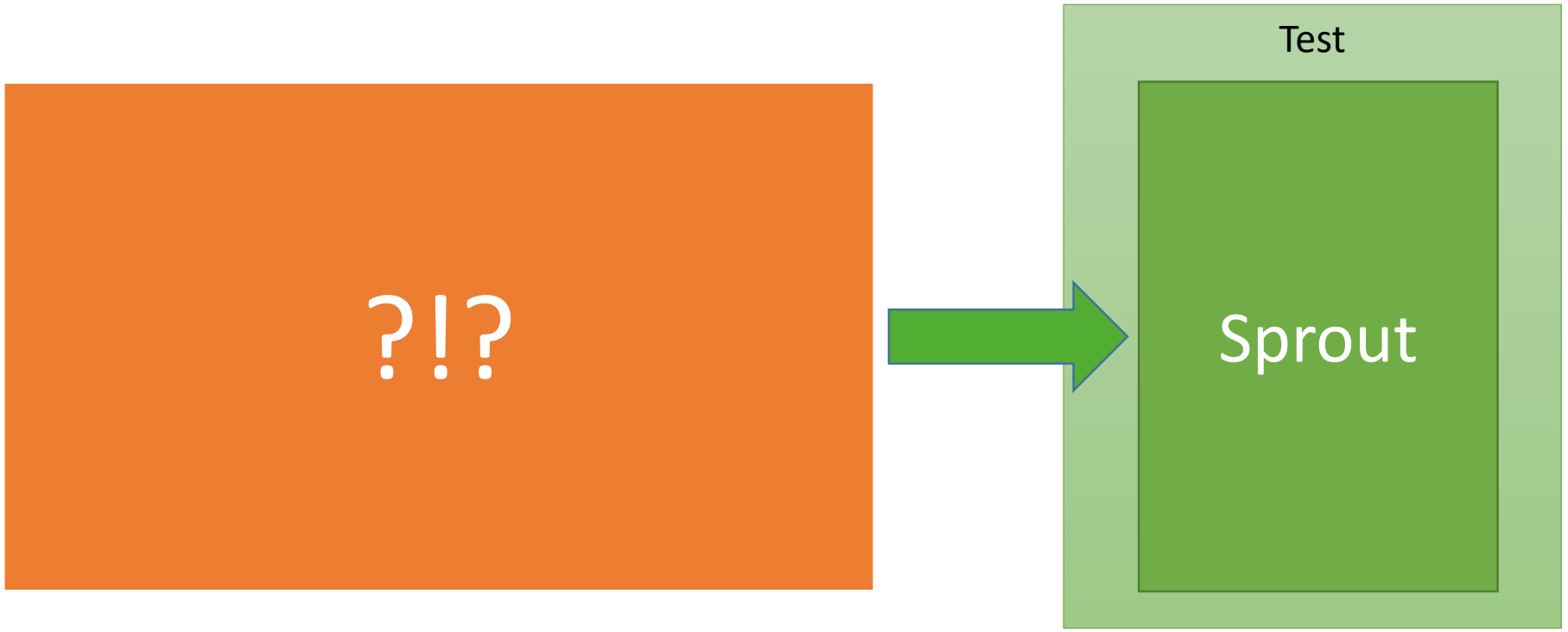
Sprout



Sprout



Sprout



Sprout

Vorteile

- Schnell einsetzbar wenn nicht viele Abhängigkeiten bestehen
- Leicht zu testen

Nachteile

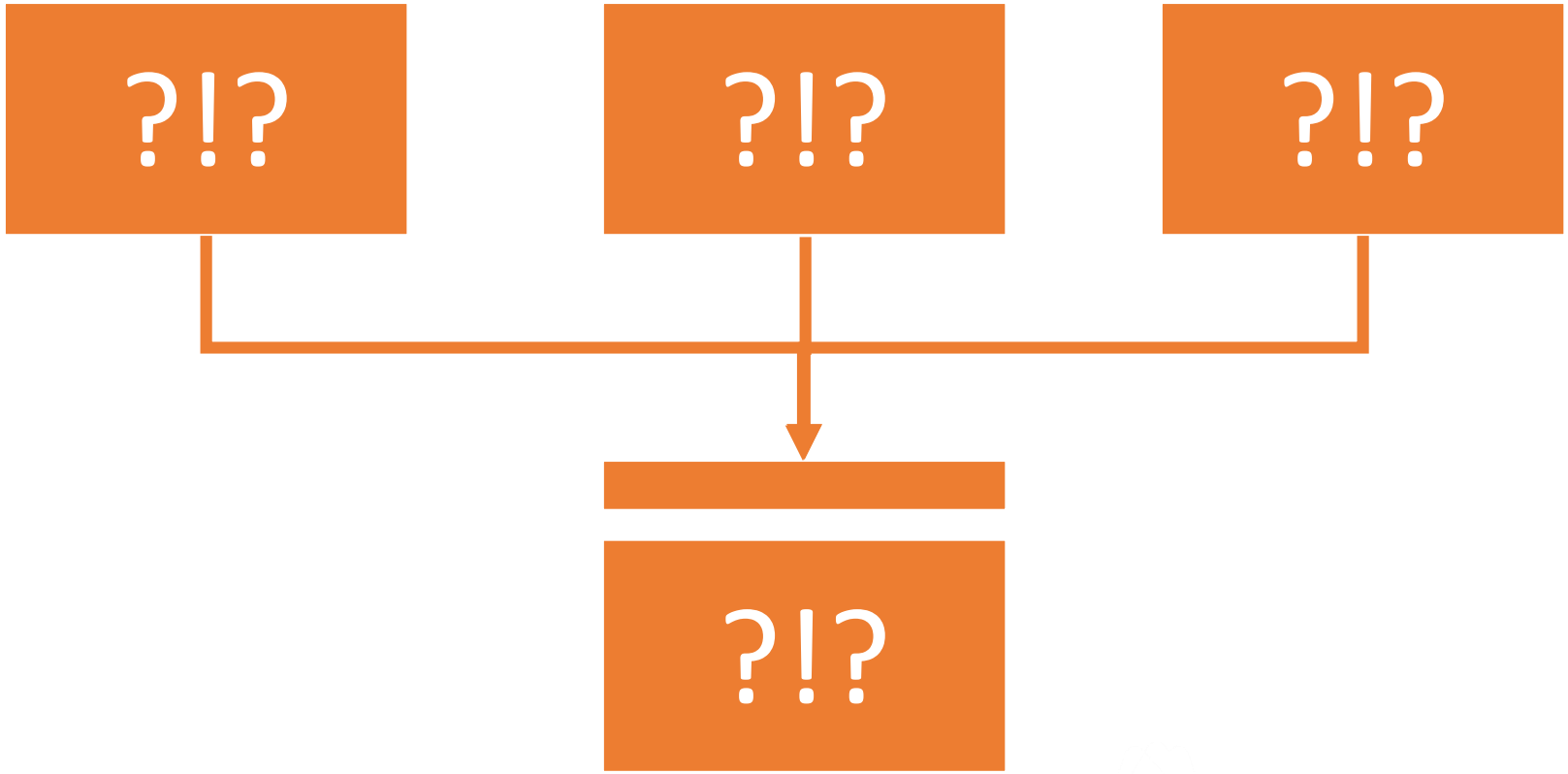
- Kann langfristig zu unübersichtlichem Code führen
- Keine Verbesserung der Gesamtsituation
- Evtl. Verletzung von Seperation of Concernce & DRY
- Wenn Sprouts falsch verwendet werden induzieren sie Abhängigkeiten

Temporary Clone*

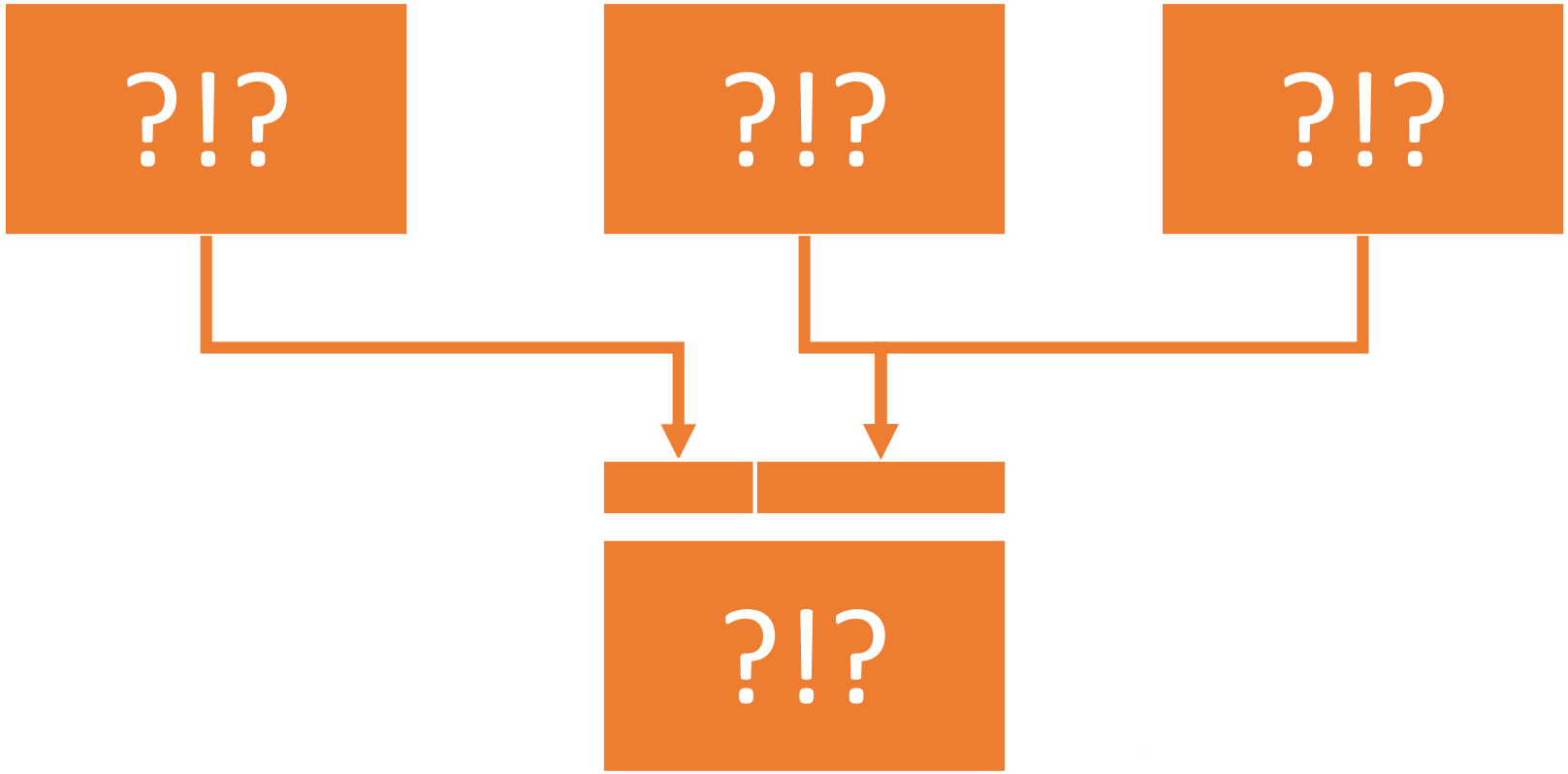
Situationen

„Ich muss eine vorhandene Schnittstelle ändern, möchte das Verhalten aber nicht unvorhergesehen schädigen.“

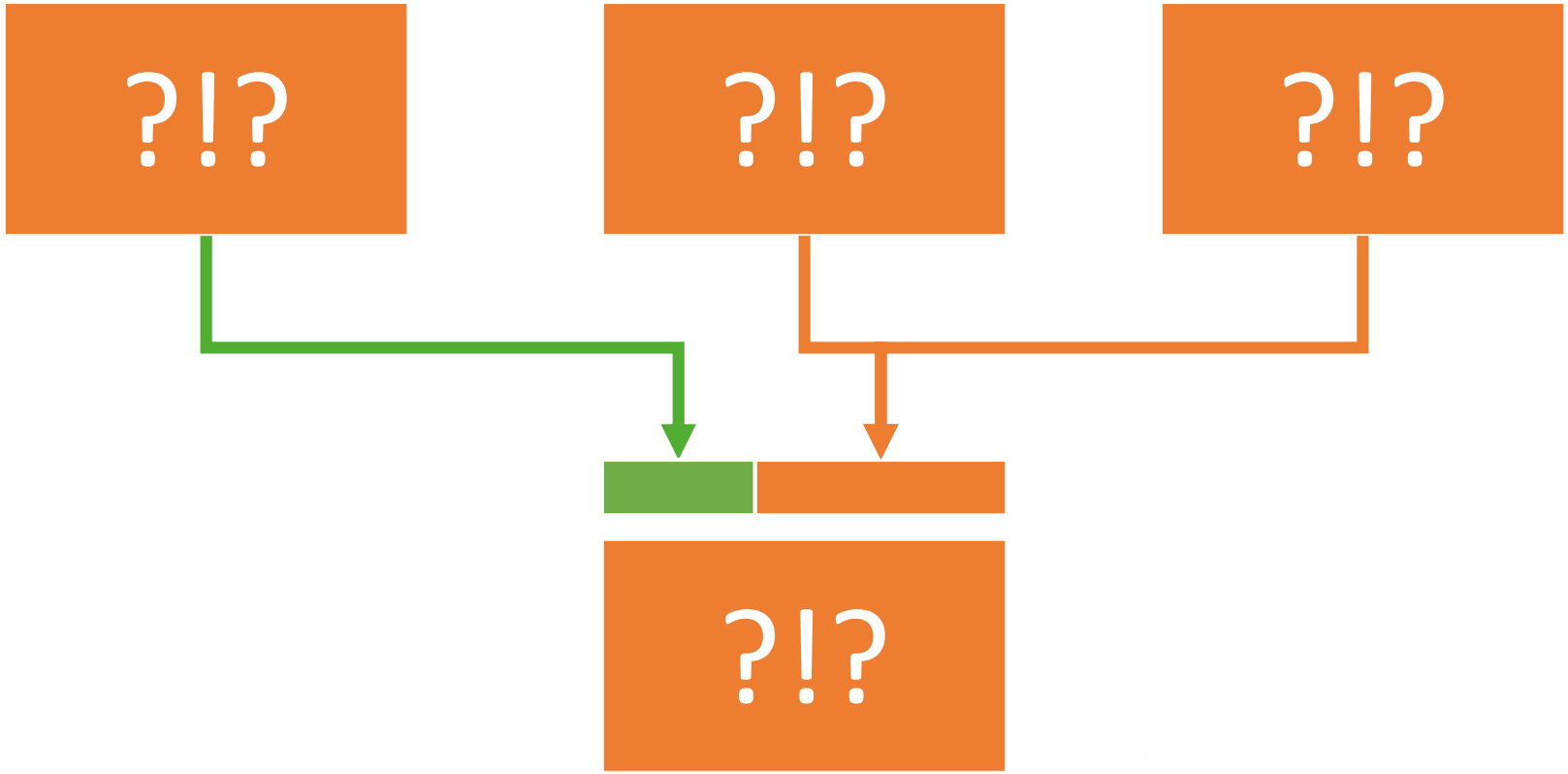
Temporary Clone



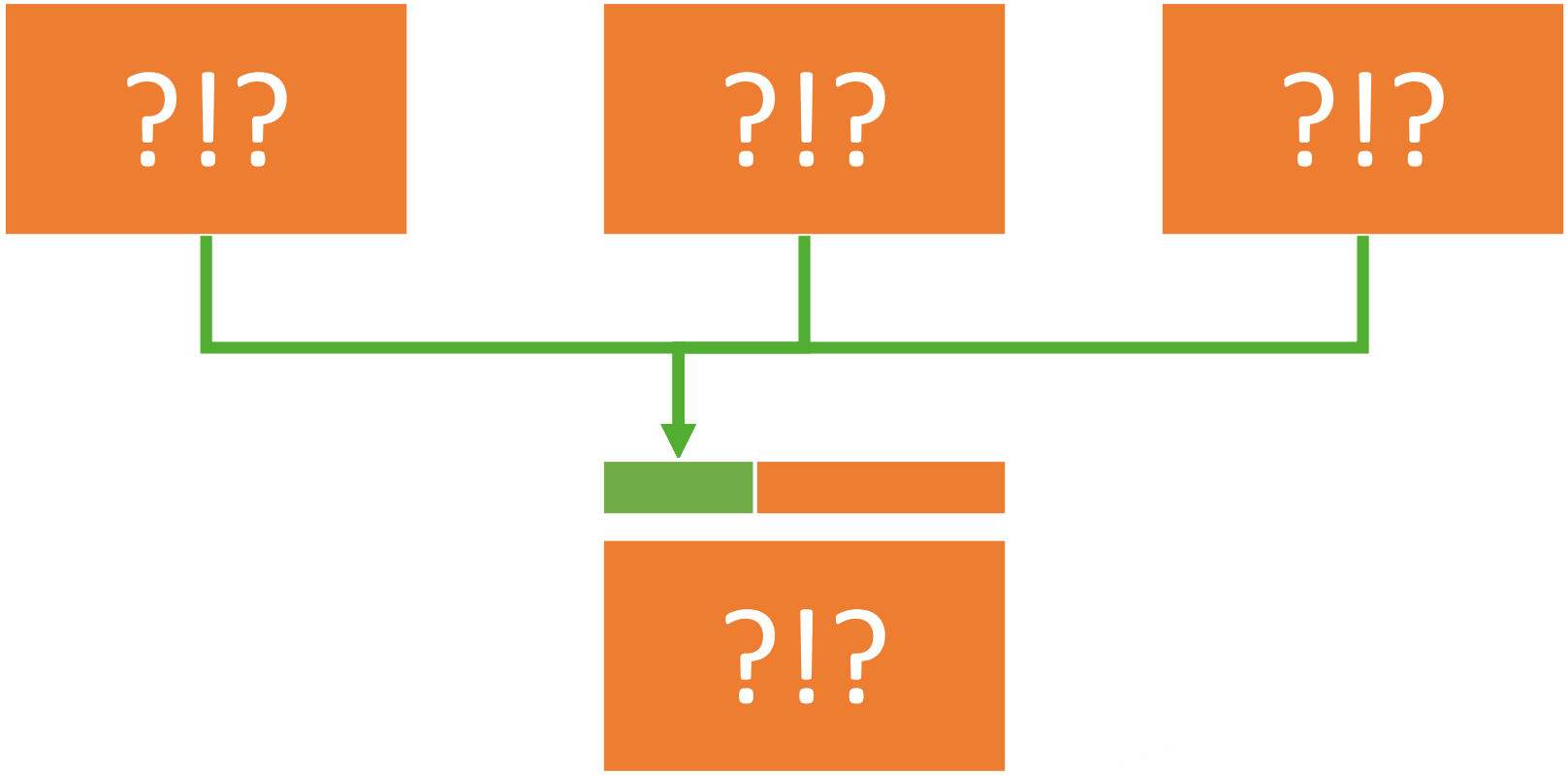
Temporary Clone



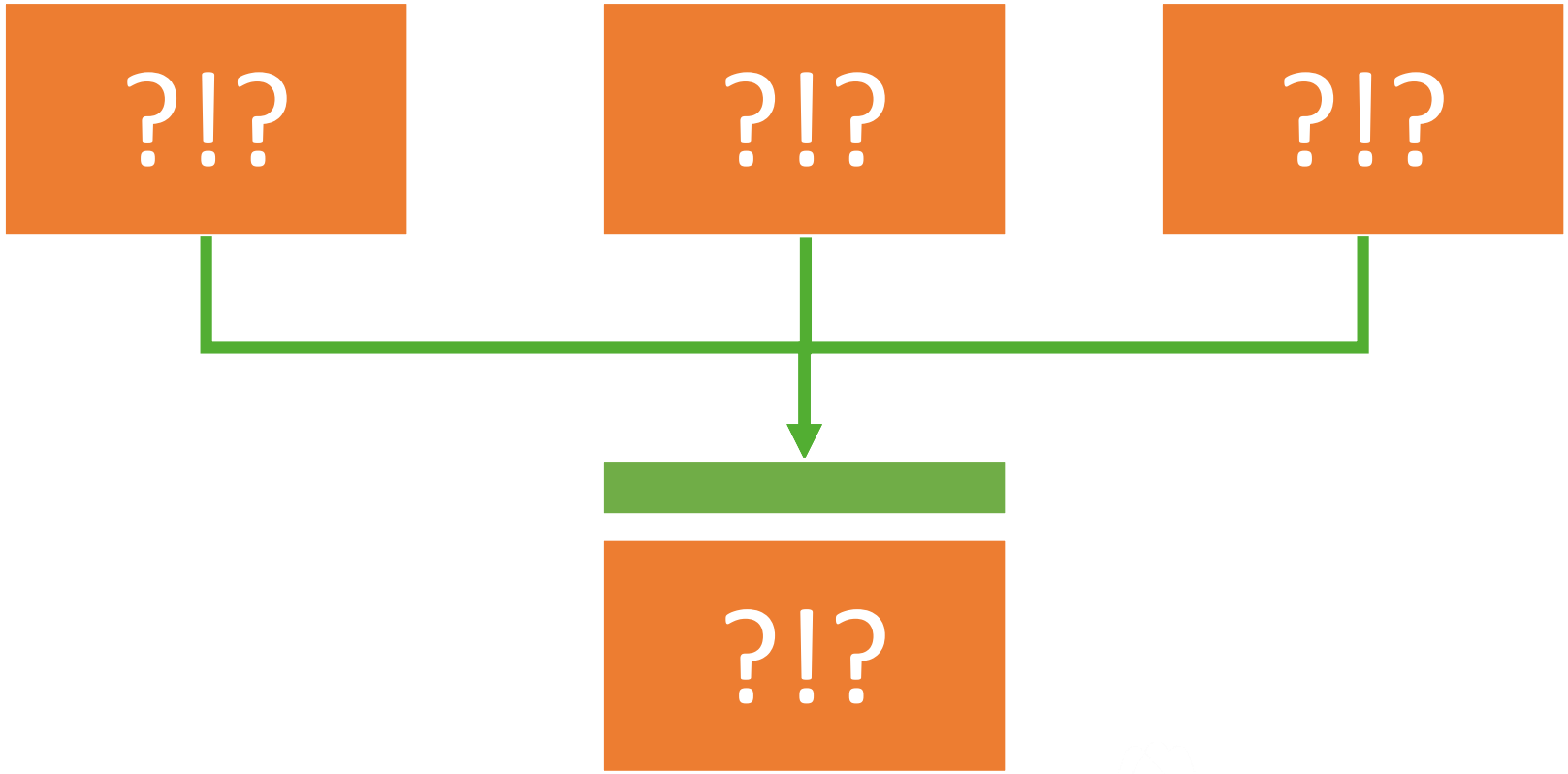
Temporary Clone



Temporary Clone



Temporary Clone



Temporary Clone

Vorteile

- Erlaubt es Veränderungen in kleinen Schritten und lokal begrenzt vorzunehmen.

Nachteile

- Kann das Laufzeitverhalten negativ beeinflussen.
- Bedeutet einen gesteigerten Test und Implementierungsaufwand.
- Kann den Code unübersichtlich machen.

Method Object

Situationen

„Ich habe eine Klasse/Methode die viel zu komplex ist.“

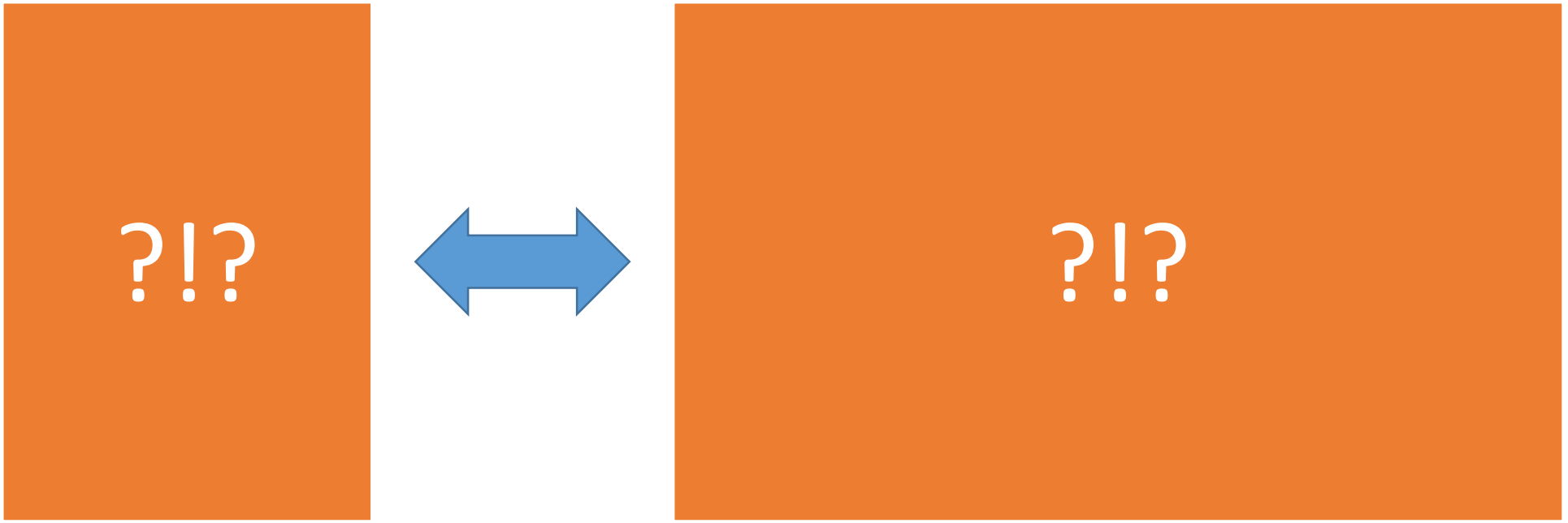
Vorgehen

Wir lagern komplexe Methoden in eigene Klassen aus und brechen sie dort in weitere Methoden auf.

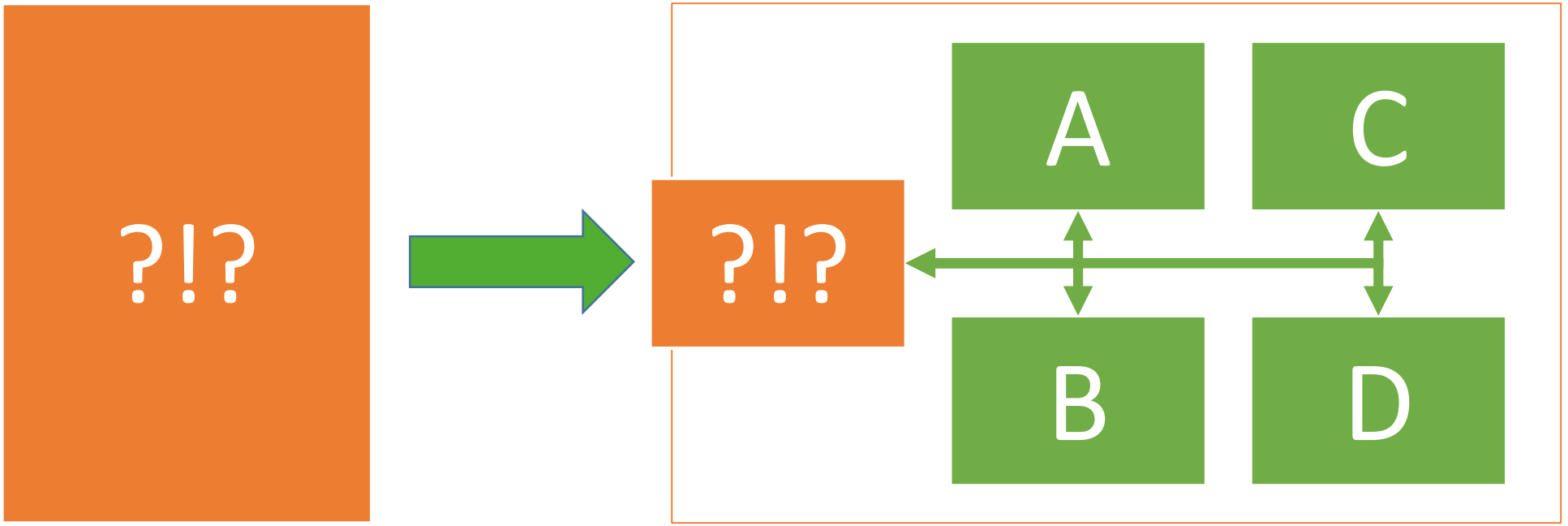
Method Object

?!?

Method Object



Method Object



Method Object

Vorteile

- Kann untestbaren Code testbar machen.
- Hilft das Verständnis für einen Vorgang zu verbessern.
- Kann bereits die Architektur verbessern.

Nachteile

- Kann den internen Status der Klasse ändern wenn mit anderen Klassenmembern vermischt -> **unbedingt eigene Klasse anlegen**.

Wrapper

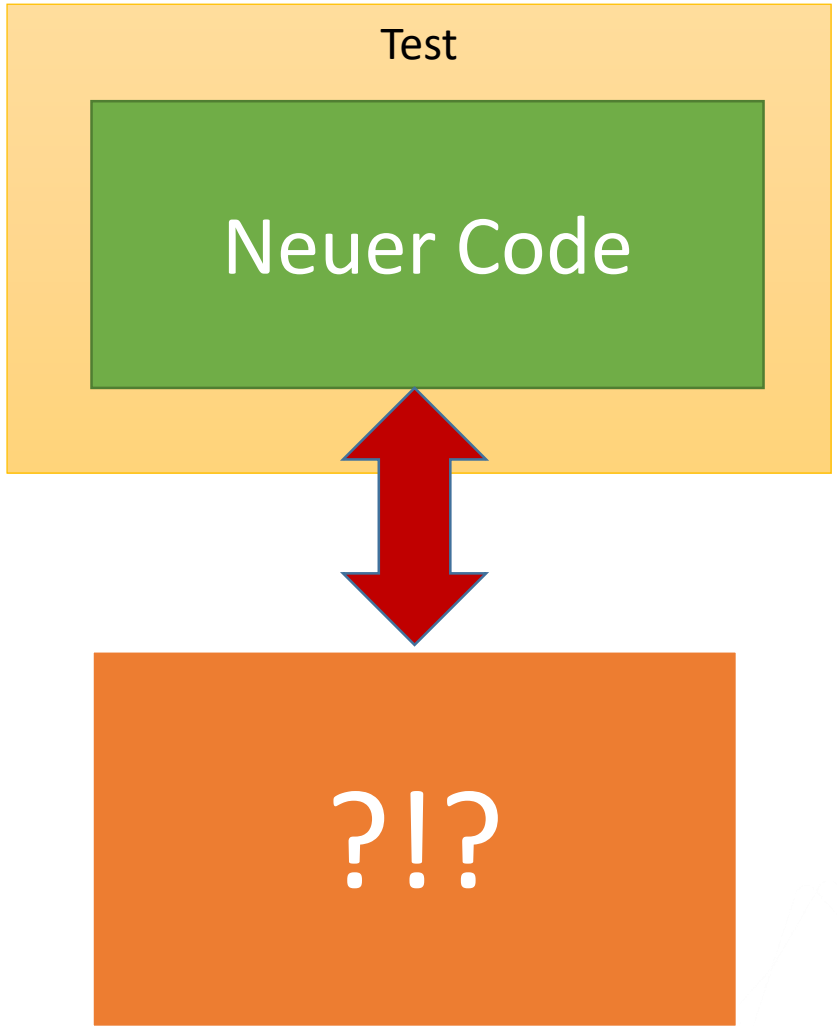
Situationen

„Ich möchte neuen Code schreiben, muss aber auf alten zugreifen.“
„Ich möchte ein Refactoring vorbereiten aber noch nicht alles umstellen.“
„Ich kann das Problem durch reines Hinzufügen von Code lösen.“

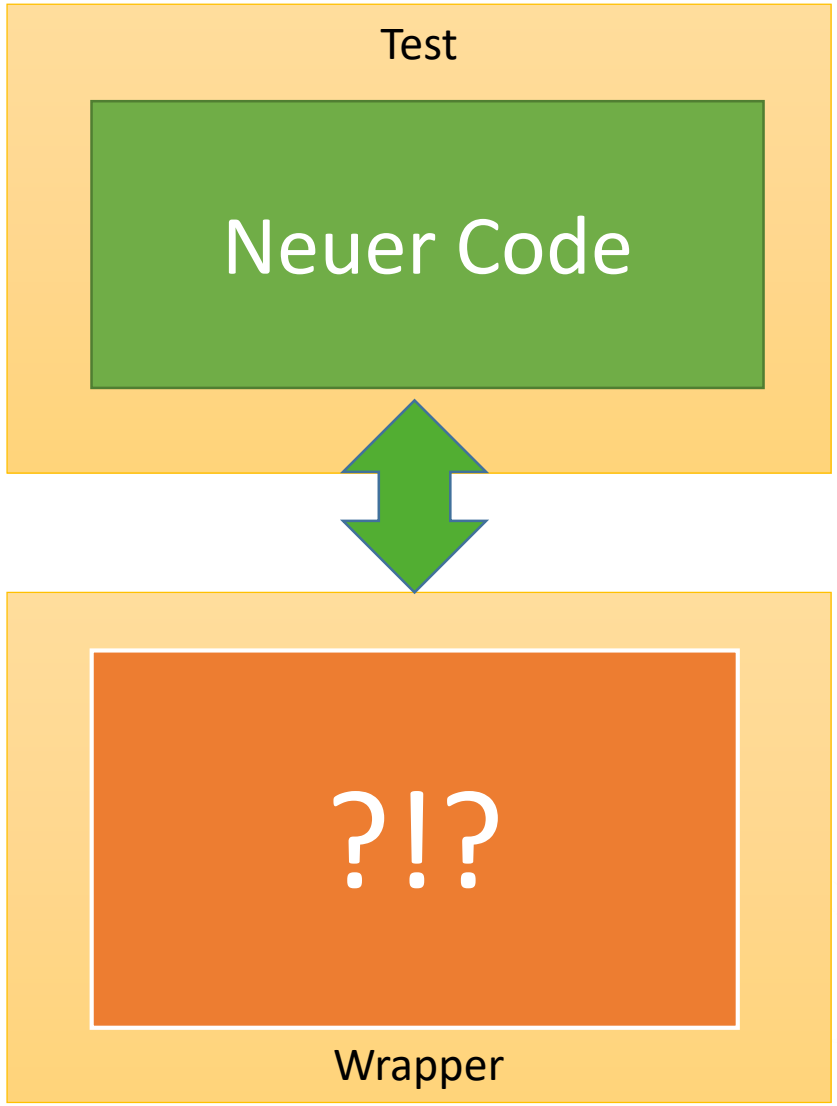
Vorgehen

Wir greifen mit dem neuen Code auf eine Kapsel, statt auf den alten Code zu.

Wrapper



Wrapper



Wrapper

Vorteile

- Erlaubt Schnittstellen für großflächige Umstrukturierungen vorzubereiten
- Erlaubt es neue Funktionalität testbar umzusetzen ohne den bestehenden Code anpassen zu müssen

Nachteile

- Kann langfristig zu unübersichtlichem Code führen
- Keine Verbesserung der Gesamtsituation
- Kann selbst komplex werden
- Kann das Laufzeitverhalten negativ beeinflussen

Demilitarized Zone*

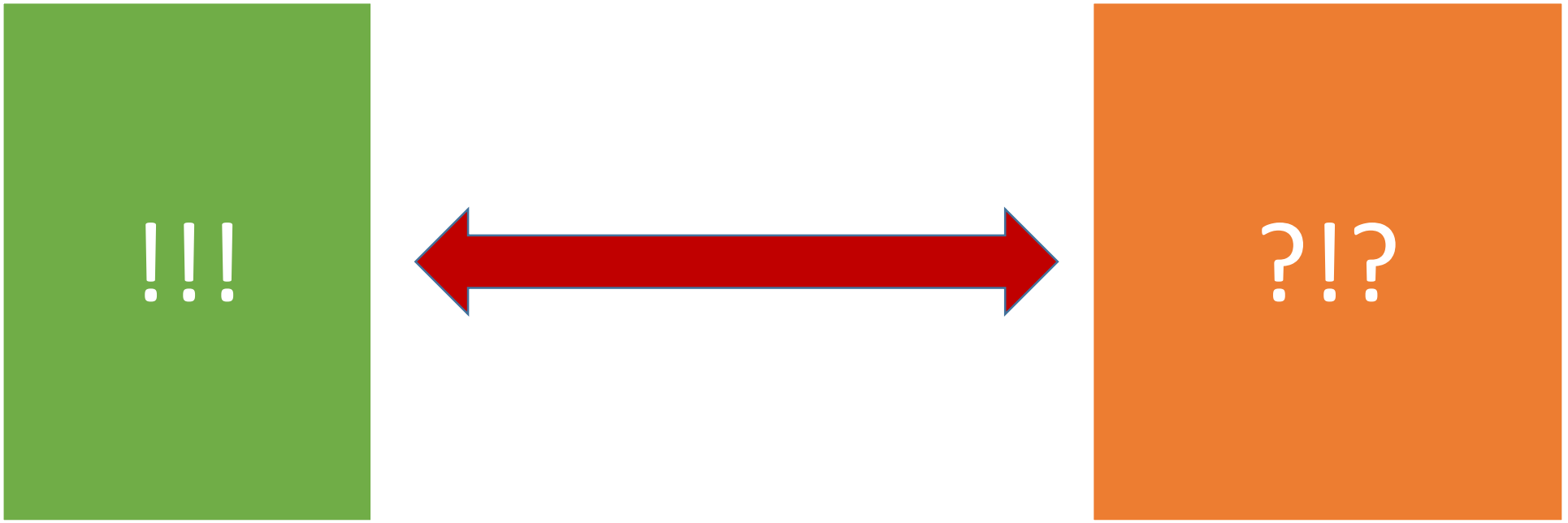
Situationen

„Ich habe eine neue Komponente einzufügen und möchte sie von Beginn an richtig umsetzen.“

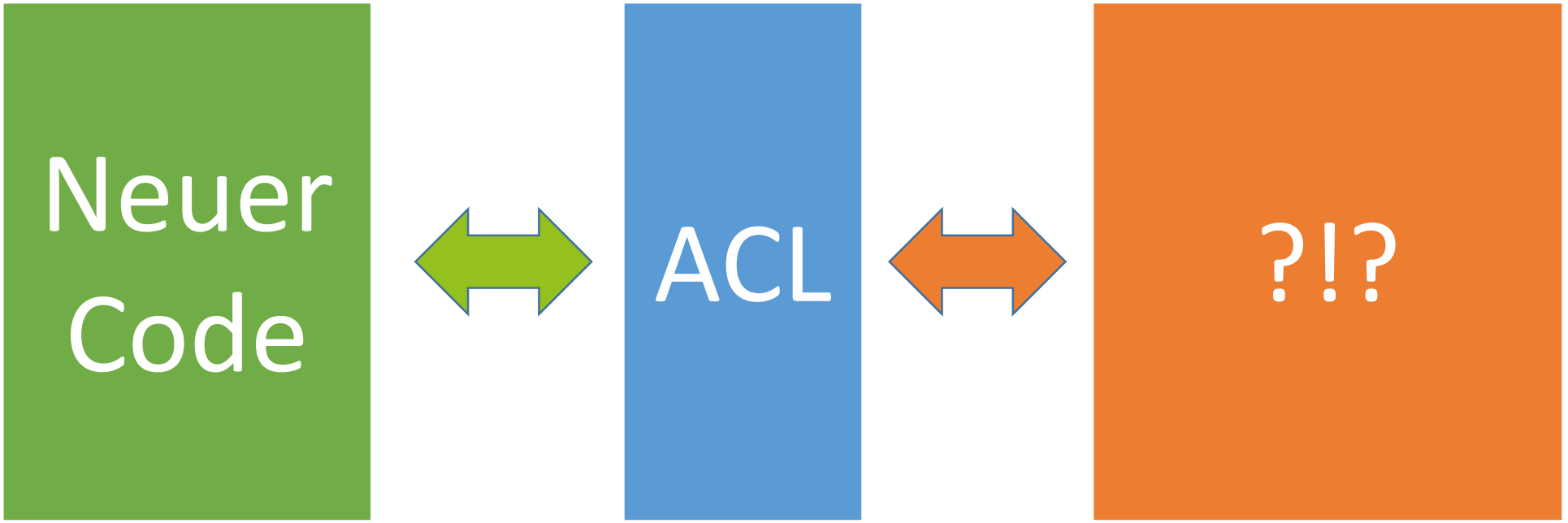
Vorgehen

Wir schaffen einen Layer aus Adaptern an das Altsystem die beide Systeme mit einander kommunizieren lassen.

Demilitarized Zone



Demilitarized Zone



Demilitarized Zone

Vorteile

- Erlaubt das neue System ohne Beeinflussung vom Altsystem zu entwickeln.
- Bereitet ein schrittweises Ersetzen des alten Codes vor.

Nachteile

- Kann das Laufzeitverhalten negativ beeinflussen.
- Bedeutet einen gesteigerten Test und Implementierungsaufwand.
- Kann den Code unübersichtlich machen.

Strangler

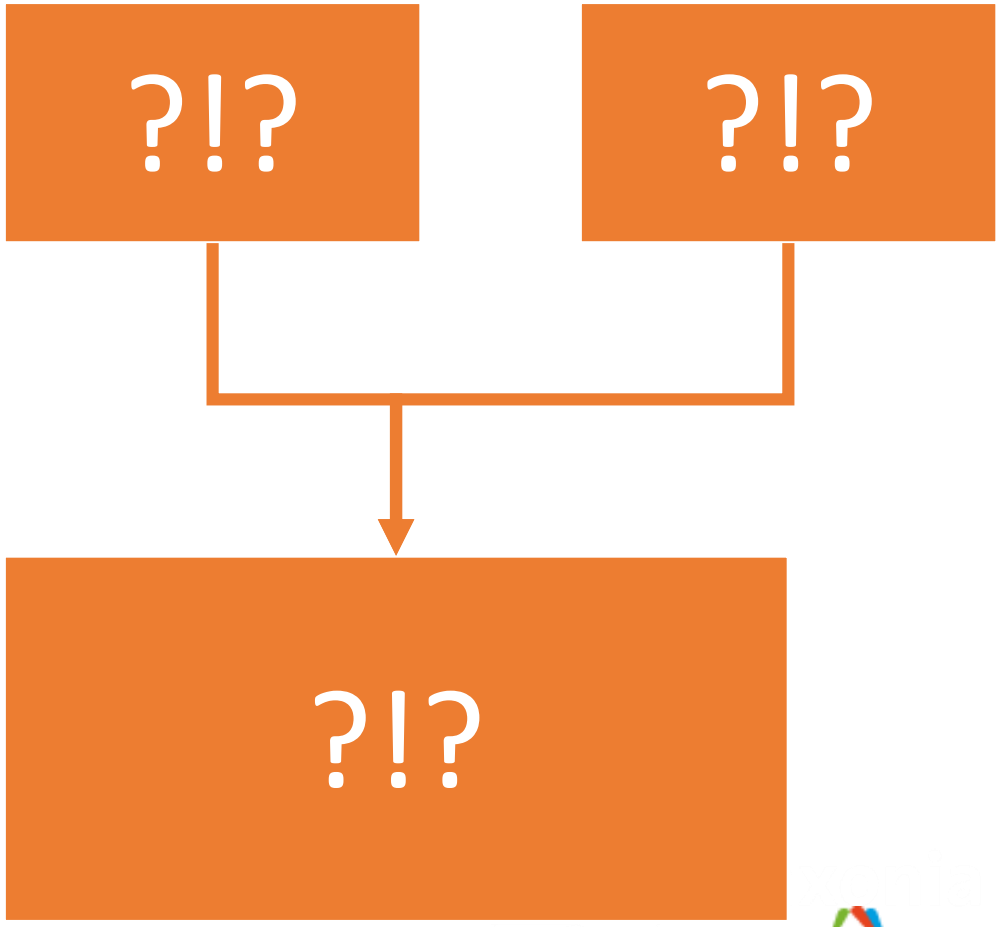
Situationen

„Ich muss ein altes System ablösen, es ist aber so komplex, dass ich dies nicht mit einen Mal tun kann. “

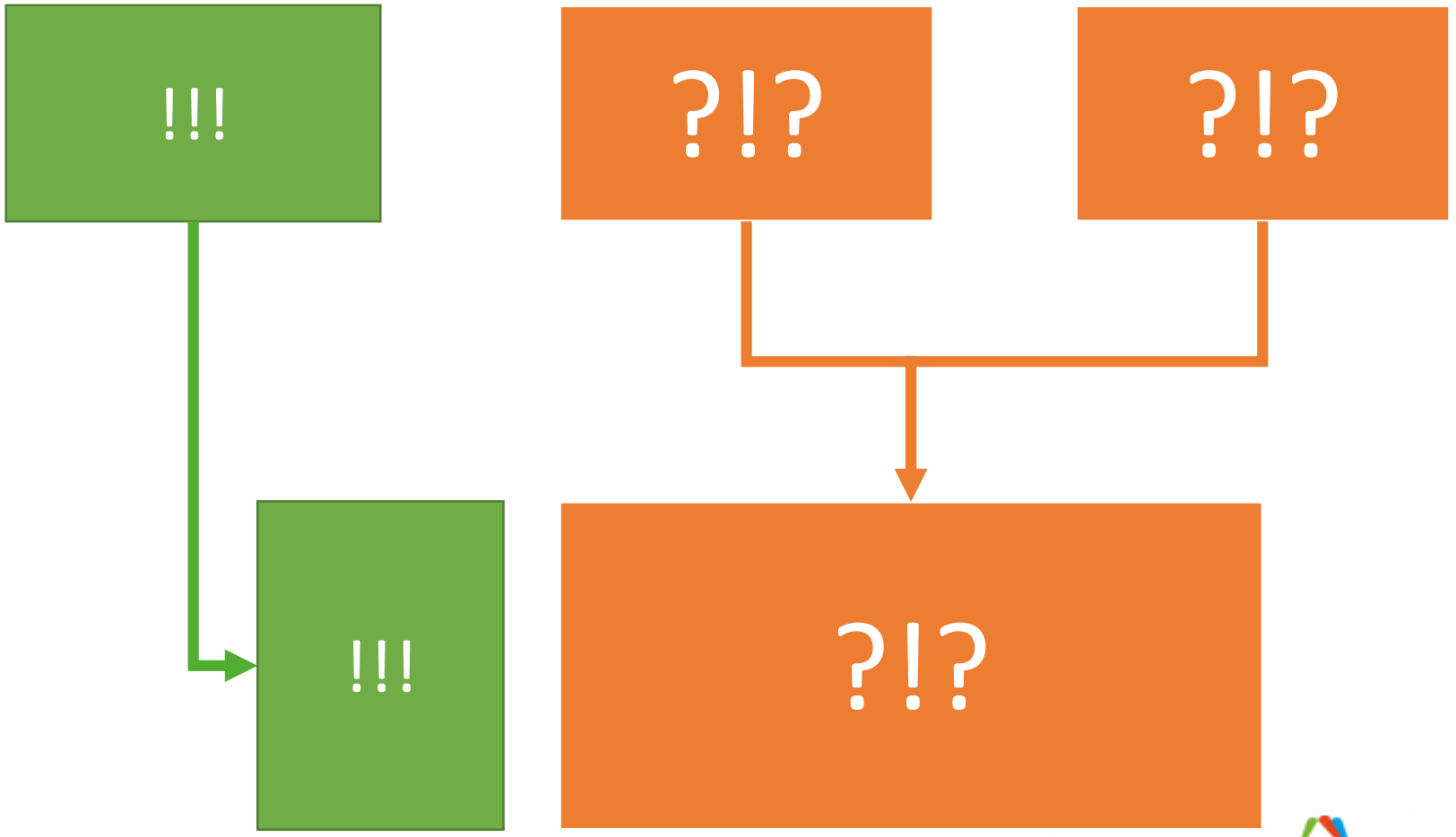
Vorgehen

Wir bauen das neue System als Sprouts, ziehen schrittweise die Funktionalität aus dem alten und entfernen all die Teile die nicht mehr im alten gebraucht werden.

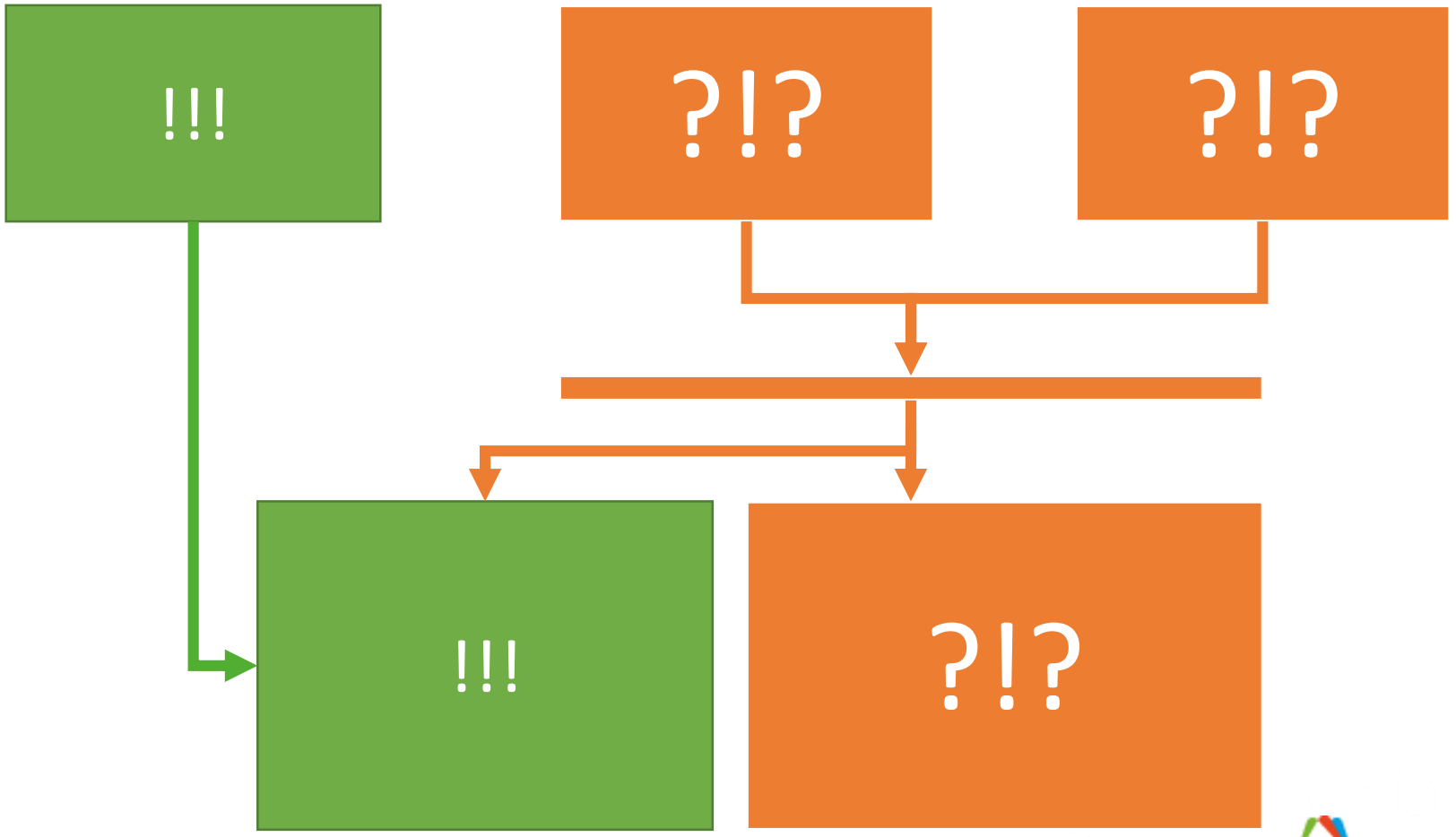
Strangler



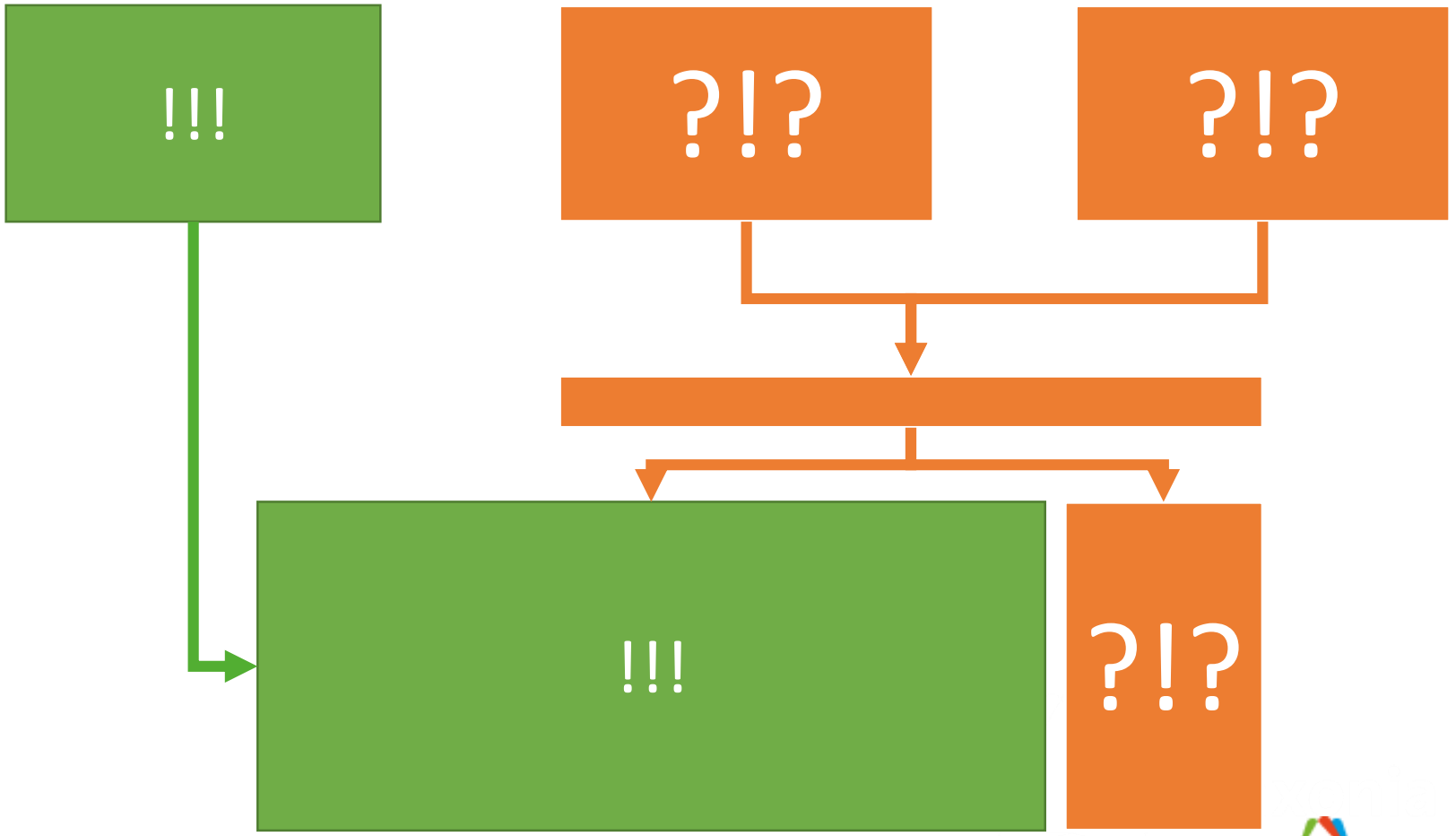
Strangler



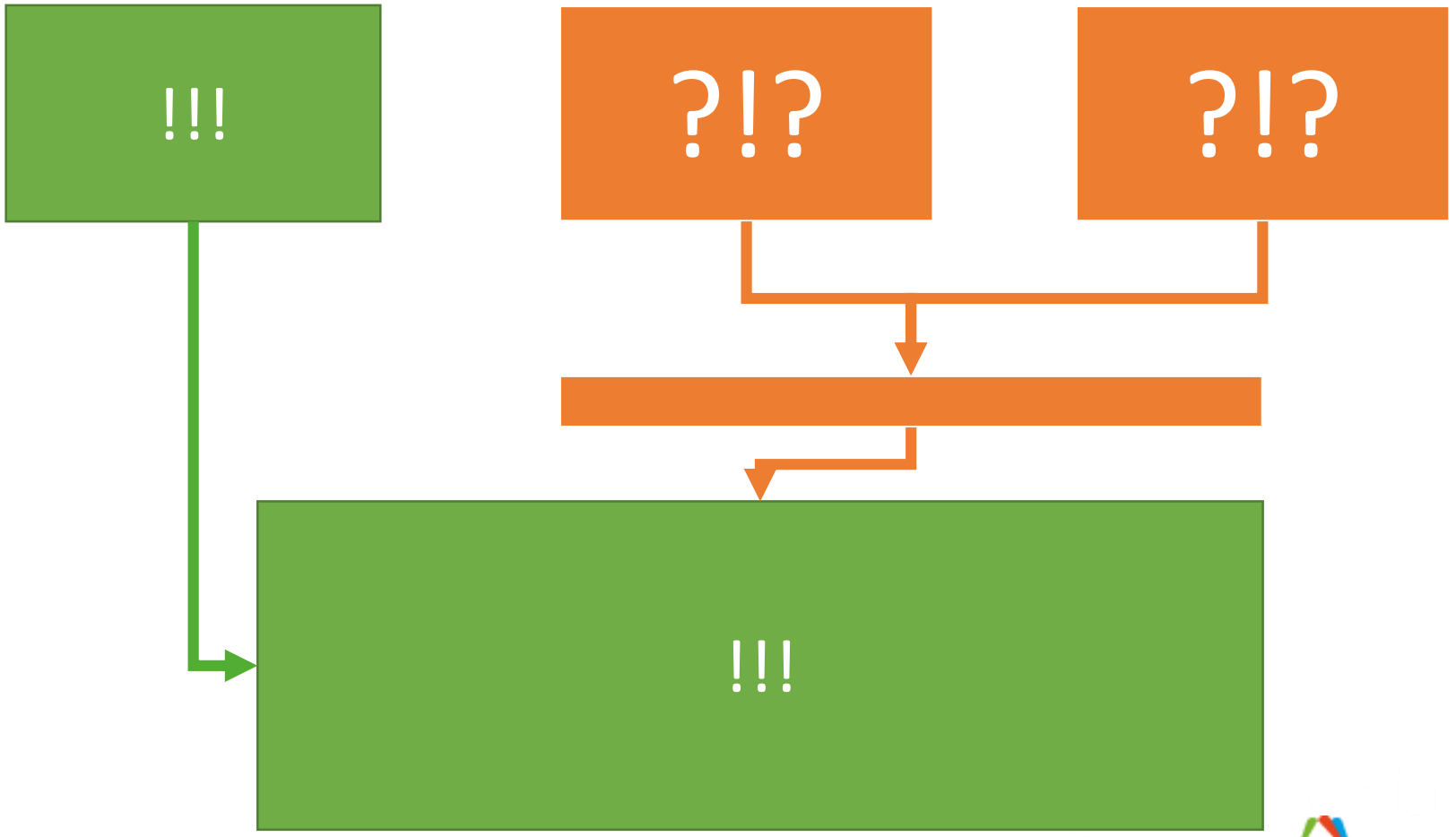
Strangler



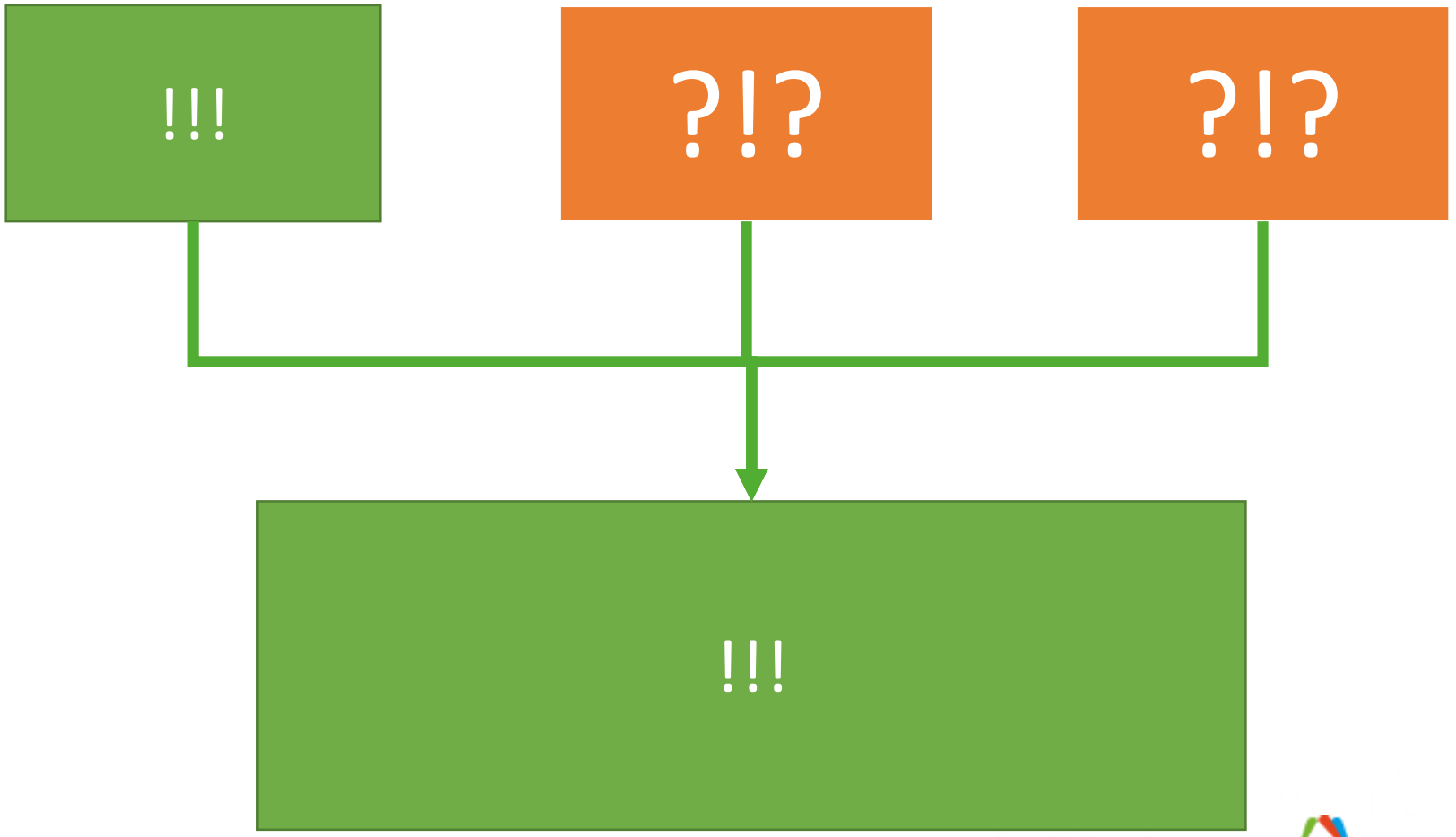
Strangler



Strangler



Strangler



Strangler

Vorteile

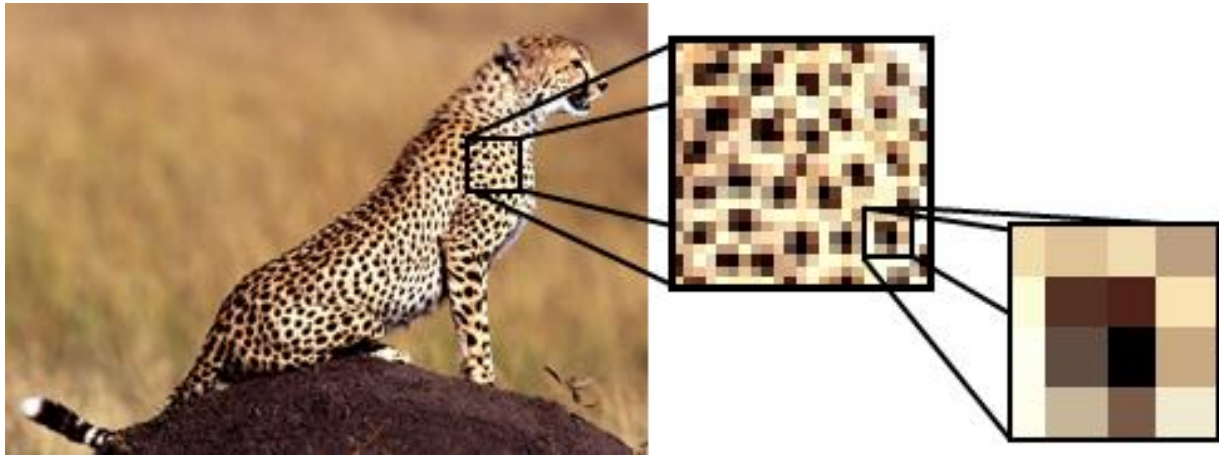
- Erlaubt es Systeme iterativ inkrementell zu ersetzen.
- Altes wie neues System werden zeitgleich gepflegt.

Nachteile

- Wenn es zu lange dauert, veraltet das neue System ebenfalls.
- Die Grenzen müssen klar definiert sein, ansonsten behindern sich die Systeme gegenseitig.

RESTRUCTURING VS. REFACTORING

Refactoring vs. Restructuring



Refactoring vs. Restructuring

RefactoringMalapropism

Martin Fowler
3 January 2004

Once a term known to only a few, "Refactoring" is now commonly tossed around the computer industry. I like to think that I'm partly responsible for this and hope it's improved some programmers lives and some business's bottom lines. (Important point, I'm not the father or the inventor of refactoring - just a documenter.)

However the term "refactoring" is often used when it's not appropriate. If somebody talks about a system being broken for a couple of days while they are refactoring, you can be pretty sure they are not refactoring. If someone talks about refactoring a document, then that's not refactoring. Both of these are restructuring.

I see refactoring as a very specific technique to do the more general activity of restructuring. Restructuring is any rearrangement of parts of a whole. It's a very general term that doesn't imply any particular way of doing the restructuring.

Refactoring is a very specific technique, founded on using small

Refactoring

- Kleine lokale Änderungen.
- Schnelle Anpassungen mit geringen Auswirkungen auf andere.
- Können ohne größere Absprache durchgeführt werden.
- Keine tiefgreifenden architekturellen Anpassungen in bestehendem Code.

Restructuring

- Großflächige architekturelle Anpassungen die sich auf die Gesamtapplikation auswirken können.
- Können andere Entwickler behindern.
- Sollten nicht ohne Absprache und Planung durchgeführt werden.

Quelle: https://martinfowler.com/bliki/RefactoringMalapropism.html

Refactoring

ALLES WIRD GUT

Saxonia Systems

So geht Software.

Refactorings sind nicht so schwer,
Restructurings dagegen sehr!

Schreiben Sie Tests **bevor** Sie sie brauchen.

Nutzen Sie eine Quellcodeverwaltung
die **viele lokale Commits** erlaubt.

Planen Sie Restrukturierungen im Team.

Wenn etwas **schmerzt**,
muss man es **häufiger** tun.

Der Sprecher

Hendrik Lösch

Senior Consultant & Coach

Hendrik.Loesch@saxsys.de

@HerrLoesch

Just-About.Net

Grundlagen der Programmierung: Test Driven Development

Business-Applikationen testgetrieben entwickeln

Grundlagen der Programmierung: Codemetriken

Softwarequalität einschätzen, sicherstellen und ...

Inversion of Control und Dependency Injection – Grundlagen

Prinzipien der modernen Software-Architektur ...

WPF-Anwendungen mit MVVM und Prism

Modulare Architekturen verstehen und umsetzen

LEARNING

Saxonia Systems
So geht Software.