**Refactoring** – Refaktorisierung des Codes

Allgemeines

* Systematischer Prozess der Codeverbesserung  
  🡪Ohne neue Funktionalität zu erzeugen
* Schrittweise Codeoptimierung ohne Einbußen in der Lauffähigkeit und im Inhalt
* „Keep it short and simple“
* Zur Vermeidung von „Dirty Code”
* Zentraler Bestandteil der agilen Softwareentwicklung

*Manuelle oder automatisierte Strukturverbesserung von Code unter Beibehaltung des beobachteten Programmverhaltens.*

Ziele / Nutzen

* Sauberer Code
* Einfache Code-Gestaltung
* Bessere Lesbarkeit
* Höhere Code-Qualität und –Struktur
* Bessere Verständlichkeit
* Reduzierung des Codes auf ein Minimum ohne Einsparnisse am Inhalt
* Vermeidung von Redundanzen
* Zeitersparnis, Kostenersparnis
* Einfache Wartung
* Testbarkeit
* Erweiterbarkeit

Informationsquellen

* <https://refactoring.guru/refactoring>

Clean Code

* **Ist für andere Programmierer verständlich und nachvollziehbar**
  + Bezieht sich sowohl auf Algorithmen, als auch auf die Benennung von Variablen, Methoden und Klassen und die Struktur und „Einfachheit“ der Methoden und Klassen
* **Enthält keine Duplikationen**
  + Zur Vermeidung von Inkonsistenzen und Anomalien
  + Damit der Code einfach wartbar bleibt
  + Zur Verringerung der Komplexität und des Zeitaufwands bei Änderungen
* **Enthält eine minimale Anzahl an Klassen, etc.**
  + Weniger Code = Weniger Wartung, weniger Bugs
* **Besteht alle Tests**
  + Um die „technische Schuld“ zu bereinigen
  + Es kostet zwar Zeit, Tests zu schreiben  
    🡪Jedoch beschleunigt es den Fortschritt des gesamten Projekts

„Technical debt“ und „dirty Code“ – Technische Schuld

Ursachen für technische Verschuldung:

* **Geschäftlicher Druck**
* **Mangelndes Verständnis für die Folgen von technischer Verschuldung**
  + Management kennt den Wert der Refaktorisierung nicht
* **Versäumnis der Bekämpfung von Kohärenzen**
* **Mangel an Tests**
* **Fehlende Dokumentation**
  + Neue Personen können nur schwierig in das Projekt integriert werden
  + Kann die Entwicklung des Projekts zum Stillstand bringen
* **Mangelnde Interaktion zwischen den Teammitgliedern**
  + Wenn die Wissensbasis nicht verteilt ist
  + Arbeit mit veraltetem Prozessverständnis und veralteten Informationen
  + Falsche/veraltete Schulungen für neue Teammitglieder
* **Langfristige gleichzeitige Entwicklung in mehreren Branchen**
  + Je mehr Änderungen isoliert vorgenommen werden, desto größer ist die technische Gesamtverschuldung
* **Verzögerte Refaktorisierung**
  + Jede Zeile neuer Code funktioniert mit den veralteten Teilen und macht Refaktorisierung schwieriger
  + Je mehr sich das Refactoring verzögert, desto mehr abhängiger Code muss in Zukunft überarbeitet werden
* **Fehlende Überwachung der Einhaltung**
  + Fehlende Regeln und Absprachen
* **Inkompetenz / Unwissenheit**

Wann wird refaktorisiert?

* **Rule of Three**  
  🡪Wenn man ähnlichen Code zum dritten Mal schreibt, sollte refaktorisiert werden
* **Wenn man ein Feature in den Code einfügt**  
  🡪Zuerst refaktorisieren, damit der Code sauber und direkt verständlich ist  
  🡪Refaktorisierung macht es einfacher, neue Funktionen hinzuzufügen, da es einfacher ist, Änderungen in sauberem Code vorzunehmen
* **Wenn man einen Fehler behebt**
* **Während der Code-Review**

Wie wird refaktorisiert?

* In einer Reihe von kleinen Änderungen (schrittweise)
* Programm sollte nach jeder Änderung funktionsfähig bleiben

**Checkliste:**

* **Der Code soll sauberer werden**
  + Vorher sollten Testfälle geschrieben werden, um sicherzustellen, dass die Funktionalität gegeben bleibt
* **Dabei sollen keine neue Funktionalität erstellt werden**
* **Alle bestehenden Tests müssen nach dem Refaktorisieren durchlaufen**
  + Es kann sein, dass:
    - die Tests zusammenbrechen, weil das Refaktorisieren fehlerhaft war
    - die Tests auf einem zu niedrigen Level waren

🡪Tests refaktorisieren  
🡪Tests auf einer höheren Ebene schreiben

Was wird refaktorisiert? – Code Smells

**① Bloaters – Aufbläher**

* Methoden und Klassen, die viel zu groß sind
* Sammeln sich im Laufe der Zeit an, wenn sich das Programm weiterentwickelt

**🡪 Lange Methoden:**

* + Methoden sollten nicht länger als 10 Zeilen sein, da sie sonst unverständlich und schwieriger zu pflegen werden
  + Entstehen, weil es einfach ist, Code zu schreiben, als ihn zu lesen
  + Bleibt unbemerkt, bis die Methode übermäßig groß ist
  + Es entsteht Spaghetti-Code mit häufig versteckten Code-Duplikaten

Faustregel: *Wenn man das Bedürfnis hat, seine Methode zu kommentieren, da sie erklärungsbedürftig ist 🡪 Code in neue Methode auslagern*

* + Neue Methode hat dann beschreibenden Namen  
    🡪Methode muss nicht angeschaut werden, um zu verstehen, was sie bewirkt
* Verwendung von *Extract Method*
* Ersetzen von *Temp* durch *Query*, Einführen von Parameterobjekten, Nutzung von Objekten
* Ersetzen von *Method* durch *Method Object* (Verschieben der Methode in Objekt)
* Auslagern bedingter Operatoren (*Decompose Conditional*) und Schleifen (*Extract Method*) in separate Methoden

**🡪 Große Klassen:**

* + Enthält viele Felder, Methoden, Zeilen
  + Refaktorisierung erspart Entwicklern das Merken einer großen Anzahl an Attributen für eine Klasse
  + Aufteilung von Klassen in Teile vermeidet die Duplizierung von Code und Funktionalität
* Klassen extrahieren
* Unterklassen extrahieren, wenn ein Teil des Verhaltens der großen Klasse auf verschiedenen Weise implementiert werden kann
* Schnittstellen extrahieren, wenn es notwendig ist, eine Liste der Operationen und Verhaltensweisen zu haben, die der Client nutzen kann

**🡪 Verwendung vieler primitiver Datentypen:**

* + Verwendung von Primitiven anstelle von kleinen Objekten
  + Verwendung von Konstanten
  + Verwendung von String-Konstanten als Feldnamen zur Verwendung in Daten-Arrays

🡪Macht Klassen bei Häufung unübersichtlich, unflexibel, unverständlich und unorganisiert

* Verschieben des mit den Daten verbundenen Verhaltens in die Klasse
* Ersetzen des Datenvalues durch ein Objekt
* Wenn die Werte primitiver Felder in Methodenparametern verwendet werden, Verwendung von *Introduce Parameter Objects* oder *Preserve Whole Objects*
* Wenn komplizierte Daten in Variablen kodiert sind, Verwendung von *Replace Type Code with Class, Replace Type Code wuth Subclasses*  oder *Replace Type Code with State/Strategy*
* Wenn sich unter den Variablen Arrays befinden, Verwendung von *Replace Array with Object*
* Operationen auf bestimmte Daten befinden sich am selben Ort, anstatt verstreut
* Leichteres Auffinden von Code-Duplikaten
* Code wird flexibler, verständlicher und organisierter

**🡪 Lange Parameterlisten:**

* + Parameterlisten mit mehr als drei bis vier Parametern sind zu lang
  + Entstehen, wenn mehrere Arten von Algorithmen in einer einzigen Methode zusammengeführt wurden (,um zu steuern, welcher Algorithmus wie ausgeführt wurde)
  + Listen werden widersprüchlich und schwer zu verwenden
* Methode kann Daten ihres eigenen Objekts anstelle einer Parameterliste verwenden
* Wenn das aktuelle Objekt nicht alles notwendigen Daten enthält, kann ein anderes Objekt (mit notwendigen Daten) als Methodenparameter übergeben werden
* Wenn Parameter Ergebnisse von Methodenaufrufen eines anderen Objekts sind, Verwendung von *Replace Parameter with Method Call*
* Anstatt Daten zu übergeben: das eigentliche Objekt an Methode übergeben, Verwendung von *Preserve Whole Object*
* Wenn es mehrere Datenelemente gibt, können diese zu einem einzigen Parameterobjekt zusammengeführt werden, Verwendung von *Introduce Parameter Obejct*
* Besser lesbarer, kürzerer Code
* Bisher unbemerkter doppelter Code kann entdeckt werden

Wann ignorieren:

* Wenn die Behandlung von Parameterlisten zu unerwünschten Abhängigkeiten zwischen Klassen führen würde

**🡪 Daten-Klumpen:**

* + Klumpen mit identischen Gruppen von Variablen (z.B. Parameter für die Verbindung zu einer Datenbank) sollten in eigene Klassen umgewandelt werden
  + Entstehen bei schlechter Programmstruktur
* Um Felder in eigene Klasse zu verschieben, Verwendung von *Extract Class*
* Wenn die gleichen Datenklumpen in den Parametern von Methoden übergeben werde, Verwendung von *Introduce Parameter Obejct*
* Gesamtes Datenobjekt an Methode übergeben, Verwendung von *Preserve Whole Object*
* Von Feldern verwendeten Code in Datenklasse verschieben
* Besseres Verständnis und Organisation des Codes
* Operationen mit bestimmten Daten werden an einer einzigen Stelle gesammelt
* Geringere Code-Größe

Wann ignorieren:

* Wenn die Behandlung zu einer unerwünschten Abhängigkeit zwischen Klassen führt

**② Object-Orientation Abusers – Falsche Anwendung der Objektorientierung**

* Unvollständige oder falsche Anwendung der Prinzipien der Objektorientierung

**🡪 Switch-Statements:**

* + Häufig kann Code eines Switch-Statements an verschiedenen Stellen im Programm verstreut sein
  + Wenn dann eine neue Bedingung hinzugefügt wird, muss der gesamte Switch-Block gefunden und modifiziert werden
* Switch-Block isolieren und in Klasse setzen, Verwendung von *Extract Method* und dann *Move Method*
* Wenn der Switch-Block auf Typcode basiert, Verwendung von *Replace Type Code with Subclasses* oder *Replace Type Code with State/Strategy*
* Nachdem die Vererbungsstruktur spezifiziert ist, Verwendung von *Replace Conditional with Polymorphism*
* Einteilung der Methode in kleinere Methoden, Verwendung von *Replace Parameter with Explict Methods*
* Wenn eine der bedingten Option null ist, Verwendung von *Introduce Null Object*
* Verbesserte Code-Organisation

Wann ignorieren:

* Wenn der Switch-Block einfache Aktionen ausführt
* Wenn sie in Kombination mit der *Factory Method*  oder der *Abstract Factory* verwendet werden, um eine erstellte Klasse auszuwählen

**🡪 Temporäre Felder:**

* + Erhalten Werte nur unter bestimmten Bedingungen, ansonsten sind sie leer
  + Z.B. wenn Felder nur in einem bestimmten Algorithmus verwendet werden
* Verwendung von *Extract Class*, um Felder in eine separate Klasse zu tun
* Einführung von Null-Objekten
* Verbesserte Code-Klarheit und -Organisation

**🡪 Abgelehntes Erbe:**

* + Wenn eine Unterklasse nur einige der von ihren Eltern geerbten Methoden und Eigenschaften verwendet, ist die Hierarchie nicht gefiltert
  + Die nicht benötigten Methoden können unbenutzt bleiben oder Ausnahmen auslösen
  + Wenn jemand Code wiederverwendet und Vererbung schafft, aber die Ober- und Unterklasse völlig verschieden sind
* Vererbung eliminieren durch *Replace Inheritance with Delegation*
* Wenn Vererbung sinnvoll ist: Alle nicht benötigten Felder und Methoden beseitigen, Extrahieren der Felder und Methoden, die von der Unterklasse benötigt werden, neue Unterklasse und Vererbung hinzufügen (*Extract Superclass)*
* Verbesserte Code-Klarheit und –Organisation

**🡪 Alternative Klassen mit unterschiedlichen Interfaces:**

* + Zwei Klassen führen identische Funktionen aus, haben aber unterschiedliche Methodennamen
* Schnittstelle der Klassen auf einen gemeinsamen Nenner bringen;
  + Methoden umbenennen, um sie in allen Klassen identisch zu machen
  + Signatur und Implementierung der Methoden gleich machen
* Wenn nur ein Teil der Funktionalität der Klasse dupliziert wird, Verwendung von *Extract Superclass*
* Eine der Klassen löschen
* Bereinigung von doppeltem Code
* Code wird lesbarer und verständlicher

Wann ignorieren:

* Wenn es zu schwierig ist, Klassen zusammenzulegen (z.B. Wenn sich die alternativen Klassen in verschiedenen Bibliotheken befinden, die jeweils ihre eigene Version der Klasse haben)

**🡪 Alternative Klassen mit unterschiedlichen Interfaces:**

* + Zwei Klassen führen identische Funktionen aus, haben aber unterschiedliche Methodennamen
* Schnittstelle der Klassen auf einen gemeinsamen Nenner bringen;
  + Methoden umbenennen, um sie in allen Klassen identisch zu machen
  + Signatur und Implementierung der Methoden gleich machen
* Wenn nur ein Teil der Funktionalität der Klasse dupliziert wird, Verwendung von *Extract Superclass*
* Eine der Klassen löschen
* Bereinigung von doppeltem Code
* Code wird lesbarer und verständlicher

**③ Change Preventers – Verhinderer des Wandels**

* Wenn an einer Stelle im Code etwas geändert werden muss, dann müssen viele weitere Änderungen vorgenommen werden

**🡪 Divergierender Wandel:**

* + Änderungen an einer Klasse 🡪 Änderungen an vielen unverknüpften Methoden (Wenn viele Veränderungen an einer einzigen Klasse vorgenommen werden)
  + Aufgrund von schlechter Programmstruktur
* Aufteilen des Verhaltens über *Extract Class*
* Wenn verschiedene Klassen das gleiche Verhalten haben, können sie über Vererbung kombiniert werden (*Extract Superclass* und *Extract Subclass)*
* Verbesserte Code-Organisation
* Reduzierung der Code-Duplizierung
* Vereinfacht den Support

**🡪 Shotgun Surgery:**

* + Viele kleine Änderungen an vielen verschiedenen Klassen (Wenn eine einzige Änderung an mehreren Klassen gleichzeitig vorgenommen wird)
* Verwendung der *Move Method* und des *Move Field*, um vorhandenes Klassenverhalten in eine einzige Klasse zu verschieben
* Redundante Klassen über *Inline Class* loswerden
* Verbesserte Code-Organisation
* Reduzierung der Code-Duplizierung
* Einfachere Wartung

**🡪 Parallele Vererbungshierarchien:**

* + Immer wenn für eine Unterklasse eine Klasse angelegt wird, muss für eine andere Klasse eine Unterklasse angelegt werden
  + Besonders problematisch bei großen Hierarchien
* Deduplizierung:
  + Instanzen einer Hierarchie auf Instanzen einer anderen Hierarchie verweisen lassen
  + Hierarchie in der verwiesenen Klasse entfernen (*Move Method* und *Move Field)*
* Verbesserte Code-Organisation
* Reduzierung der Code-Duplizierung

Wann ignorieren:

* Wenn die Deduplizierung den Code noch unorganisierter macht

**④ Dispensables – Entbehrliches**

* Etwas Sinnloses und Unnötiges, dessen Fehlen den Code sauberer, effizienter und leichter verständlich machen würde

**🡪 Kommentare:**

* + Kommentare sollten vermieden werden
  + Wenn ein Codefragment ohne Kommentare nicht verstanden werden kann, sollte die Code-Struktur so geändert werden, dass Kommentare überflüssig werden
* Kommentar, der einen komplexen Ausdruck erklärt: Unterteilung des komplexen Ausdrucks in verständliche Unterausdrücke (*Extract Variable)*
* Kommentar, der Codeabschnitt erklärt: Auslagerung in Extraktionsmethode (*Extract Method)*, Methodenname beschreibt dann den Code
* *Rename Method*
* Regeln über einen Zustand: *Introduce Assertion*
* Code wird intuitiver und offensichtlicher

Wann ignorieren:

* Wenn man erklärt, warum etwas auf bestimmte Art und Weise umgesetzt wird
* Beim Erklären komplexer Algorithmen, die nicht vereinfacht werden können

**🡪 Code-Duplikationen:**

* + Wenn mehrere Programmierer gleichzeitig an verschiedenen Teilen desselben Programms arbeiten
  + Wenn Code unterschiedlich aussieht, aber das gleiche tut (Schwer auffindbar und behebbar)
* Verwendung von *Extract Method* und Platzieren von Aufrufen der neuen Methode an beiden Stellen
* Wenn der gleiche Code in zwei Unterklassen der gleichen Ebene gefunden wird:
  + Verwendung von *Extract Method* für beide Klassen, gefolgt von *Pull Up Field* für die Felder, die in der Methode verwendet werden
  + Doppelter Code innerhalb eines Konstruktors: *Pull Up Constructor Body*
  + Doppelter Code ist ähnlich, aber nicht gleich: *Form Template Method*
  + Wenn zwei Methoden das Gleiche tun, aber unterschiedliche Algorithmen verwenden: Besten Algorithmus auswählen und *Substitute Algorithm*
* Wenn der gleiche Code in zwei verschiedenen Klassen gefunden wird:
  + Verwendung von *Extract Superclass*
  + Verwendung von *Extract Class* + Verwendung der neuen Komponente in der anderen
* Wenn eine große Anzahl bedingter Ausdrücke vorhanden ist, die denselben Code ausführen: *Consolidate Conditional Expression* und *Extract Method*
* Wenn in allen Zweigen des Bedingungsbaums derselbe Code ausgeführt wird: *Consolidate Duplicate Conditional Fragments*
* Code wird kürzer und vereinfacht die Struktur

🡪Code ist einfacher zu vereinfachen und kostengünstiger zu unterstützen

Wann ignorieren:

* Wenn das Zusammenführen den Code weniger intuitiv und offensichtlich macht

**🡪 Faule Klasse:**

* + Eine Klasse, die nicht genug tut
* Verwendung von *Inline Class*
* Für Unterklassen, Verwendung von *Collapse Hierarchie*
* Reduzierung der Code-Größe
* Einfacherer Wartung

Wann ignorieren:

* Manchmal wird eine Lazy Class erstellt, um die Absichten für die zukünftige Entwicklung zu skizzieren

**🡪 Datenklasse:**

* + Bezieht sich auf eine Klasse, die nur Felder und Getter und Setter enthält
  + Einfacher Container für Daten, die von anderen Klassen verwendet werden
  + Haben keine zusätzlichen Funktionalitäten
* Klasse mit öffentlichen Feldern: *Encapsulate Field* und verlangen, dass der Zugriff nur über Getter und Setter erfolgt
* Für Sammlungen (z.B. Arrays): *Encapsulate Collection*
* *Move Methode* oder *Extract Method*
* *Remove Setting Method* und *Hide Method*
* Verbesserung des Verständnis
* Verbesserung der Organisation des Codes  
  🡪Operationen mit bestimmten Daten werden jetzt an einer einzigen Stelle gesammelt, anstatt wie bisher im gesamten Code
* Hilft, Duplizierungen von Client-Code zu erkennen

**🡪 Toter Code:**

* + Variable/ Parameter/ Feld/ Methode/ Klasse wird nicht mehr verwendet
  + Z.B. in Bedingungen, die nicht mehr erreichbar sind, weil sich Umstände geändert haben
* Verwendung einer guten IDE
* Löschen von unbenutztem Code und nicht benötigten Dateien
* Unnötige Klasse: *Inline Class* oder *Collapse Hierarchie*
* *Remove Parameter*
* Reduzierung der Code-Größe
* Einfacherer Unterstützung

**🡪 Spekulative Allgemeingültigkeit:**

* + Es gibt eine unbenutzte Klasse, eine Methode, ein Feld oder einen Parameter
  + Wenn Code „nur für den Fall“ erstellet wird
* *Collapse Hierarchie*
* Unnötige Delegation von Funktionalität an eine andere Klasse kann über *Inline Class* eliminiert werden
* *Inline Method*
* *Remove Parameter*
* Unbenutzte Felder einfach löschen
* Schlanker Code
* Einfache Unterstützung

Wann ignorieren:

* Arbeit an einem Framework: Erstellen von Funktionen, die im Framework selbst nicht verwendet werden, solange Funktionalität von Benutzern des Frameworks benötigt
* Sicherstellen, dass zu löschende Elemente nicht in Unit-Tests verwendet werden

**⑤ Couplers – Koppler**

* Übermäßige Kopplung zwischen Klassen o. übermäßige Delegation statt Kopplung

**🡪 Feature-Neid:**

* + Methode greift mehr auf Daten eines anderen Objekts zu als auf ihre eigenen
  + Kann auftreten, wenn Felder in eine Datenklasse verschoben wurden
  + Operationen auf die Daten sollten auch in diese Klasse verschoben werden
* Dinge, die sich zur gleichen Zeit ändern, sollten am gleichen Ort sein
* Verwendung von *Move Method*, um eine Methode zu verschieben
* Verwendung von *Extract Method*, um einen Teil einer Methode, der auf die Daten eines anderen Objekts zugreift, zu verschieben
* Wenn Methode Funktionen aus mehreren Klassen verwendet, sollte die Methode mit den anderen Daten in die Klasse verschoben werden, welche die meisten der verwendeten Daten enthält oder die Methode wird mit *Extract Method* in mehrere Teile aufgeteilt, die an verschiedenen Stellen platziert werden können
* Weniger Code-Duplizierung
* Bessere Code-Organisation

Wann ignorieren:

* Wenn das Verhalten absichtlich getrennt gehalten wird (, um das Verhalten dynamisch zu verändern)

**🡪 Unangemessene Intimität:**

* + Klasse verwendet interne Felder und Methoden einer anderen Klasse
* Verwendung von *Move Method* und *Move Field*, um Teile einer Klasse in die Klasse zu verschieben, wo sie verwendet werden
* Verwendung von *Extract Class* und *Hide Delegate*, um Code-Relationen „offiziell“ zu machen
* Wenn Klassen wechselseitig voneinander abhängen: *Change Bidrirectional Association to Unidirectional*
* Verwendung von *Replace Delegation with Inheritance*, wenn „Intimität“ zwischen Unterklasse und Oberklasse besteht
* Bessere Code-Organisation
* Vereinfachter Support und Wiederverwendung von Code

**🡪 Nachrichtenketten:**

* + Reihe von Aufrufen *($a->b()->c()->d())*
  + Nachrichtenkette entsteht, wenn ein Client ein weiteres Objekt anfordert, dieses Objekt ein weiteres Objekt anfordert und so weiter
  + Mandant ist auf die Navigation entlang der Klassenstruktur angewiesen
  + Jede Änderung dieser Beziehungen erfordert Modifikation des Mandanten
* Verwendung von *Hide Delegate*, um eine Nachrichtenkette zu löschen
* Verwendung von *Extract Method* und *Move Method*, um Funktionalität an den Anfang der Kette zu stellen
* Reduzierung der Abhängigkeiten zwischen Klassen einer Kette
* Reduzierung der Menge an aufgeblähtem Code

Wann ignorieren:

* Wenn dadurch schwer zu erkennen ist, wo Funktionalität tatsächlich auftritt (Vermeidung von *Middle Man*)

**🡪 Mittlerer Mann:**

* + Wenn eine Klasse nur eine Aktion durchführt und die Arbeit an eine andere Klasse delegiert, warum gibt es sie dann überhaupt?
  + Kann das Ergebnis eine Beseitigung von Nachrichtenketten sein
  + Kann das Ergebnis der Arbeit einer Klasse sein, die nach und nach in andere Klassen verlagert wird
  + Klasse bleibt leere Hülle, die nichts anderes tut, als zu delegieren
* *Remove Middle Man*
* Weniger sperriger Code

Wann ignorieren:

* Wenn Middle Man aus einem bestimmten Grund geschaffen wurde:
  + Z.B. um klassenübergreifende Abhängigkeiten zu vermeiden
  + Z.B. bei Erstellung durch *Proxy* oder *Decorator*

**⑥ Sonstige Smells**

* Fallen in keine der aufgeführten Kategorien

**🡪 Unvollständige Bibliotheksklasse:**

* + Wenn Bibliothek die Anforderungen des Benutzers nicht mehr erfüllt, ist es oft unmöglich, die Bibliothek zu verändern, weil sie schreibgeschützt ist
  + Der Autor der Bibliothek hat die benötigten Funktionen nicht bereitgestellt oder sich geweigert, sie zu implementieren
* *Introduce Foreign Method*
* *Introduce Local Extension*, um große Änderungen vorzunehmen
* Reduzierung der Code-Duplizierungen (es muss keine eigene Bibliothek von Grund auf neu erstellt werden)

Wann ignorieren:

* Erweiterung kann zusätzlichen Arbeitsaufwand verursachen, wenn Änderungen der Bibliothek Änderungen im Code nach sich ziehen