WHZ Westsächsische Hochschule Zwickau
Hochschule für Mobilität

Appsfactory

Bachelorarbeit

Entwicklung eines Werkzeugs zur Automatisierung des Figma-Exports für die UI-Entwicklung mit Compose

Fakultät physikalische Technik/ Informatik

Studiengang Informatik

Vorgelegt von:	Aygerim Budaychieva
Matrikelnummer:	50781
Erstgutachter:	Prof. Dr. Ralf Laue
Zweitgutachter:	Prof. Dr. Rainer Wasinger
Firmenbetreuer:	B. Sc. Jakob Ulbrich

Abgabedatum:	26.02.2026

Zusammenfassung

Diese Bachelorarbeit adressiert die technologische Lücke zwischen Design-Spezifikation und technischer Implementierung in der App-Entwicklung. Ein zentrales Hindernis bildet dabei die effiziente Synchronisation von Design-Ressourcen. Die manuelle Überführung von Design-Tokens und Icons aus Figma-Dateien ist im internen Arbeitsablauf der Appsfactory GmbH zeitaufwändig und fehleranfällig. Dieses Problem wird im Rahmen dieser Arbeit durch die Automatisierung des Figma-Exports für die UI-Entwicklung mit Compose gelöst. Aktuell mangelt es an Werkzeugen zur effizienten Abbildung von Multi-Brand-Strukturen in plattformübergreifenden Projekten. Zudem fehlen Lösungen, die sich nahtlos in bestehende Entwicklungsprozesse integrieren lassen. Zur Schließung dieser Lücke wurde das Werkzeug FigmaResGen entwickelt.

Das Werkzeug nutzt die Figma REST API zum automatisierten Datenabruf von Design-Ressourcen. Entwickler konfigurieren die Auswahl der benötigten Ressourcen über spezifische Parameter. Das Werkzeug gleicht die polymorphen JSON-Objekte der API mit dieser Konfiguration ab. Referenzketten von Figma-Variablen werden mittels iterativer Alias-Auflösung in typsichere Kotlin-Quelldateien transformiert. Dabei werden sowohl verschiedene Modi als auch komplexe Markenidentitäten unterstützt. Die praktische Erprobung erfolgte innerhalb eines industriellen Kundenprojekts unter Verwendung von Compose Multiplatform. Die vollständige Konfiguration und Generierung beanspruchte weniger als 30 Minuten. Alle funktionalen Anforderungen an das Werkzeug wurden erfüllt. Künftige Versionen sollen die Unterstützung von Figma Extended Collections beinhalten.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In der modernen Benutzeroberflächen-Entwicklung (UI-Entwicklung) hat sich die Arbeitsweise von monolithischen Seitendesigns zu modularen Komponentenarchitekturen verändert. Den Grundstein für dieses Verständnis legte unter anderem Marcotte mit dem Konzept des *Responsive Webdesigns* [1]. Hay, Autor von „Responsive Design Workflow“, fasst diesen Paradigmenwechsel mit dem Leitsatz zusammen:

> *„We're not designing pages. We're designing systems of components“* [2].

Anschließend legte Frost im Jahr 2013 mit der Vorstellung des „Atomic Design“-Konzepts in seinem Blog den methodischen Grundstein für diesen Wandel [3]. In seinem im Jahr 2016 veröffentlichten gleichnamigen Fachbuch vertiefte er diese Methodik und beschrieb detailliert die Notwendigkeit, Benutzeroberflächen in ihre kleinsten Bestandteile zu zerlegen, um Konsistenz und Skalierbarkeit über verschiedene Plattformen hinweg zu gewährleisten [4].

Um ein konsistentes Erscheinungsbild über alle Komponenten hinweg zu gewährleisten, werden Designsysteme eingesetzt. Figma, ein marktführendes Design-Werkzeug [5], definiert ein Designsystem als eine Sammlung wiederverwendbarer Komponenten, Richtlinien und Ressourcen. Es fungiert als zentrale Wissensquelle für Designer, Entwickler und Auftraggeber. Die Erstellung eines Designsystems umfasst die Definition von Typografie-Stilen, Farbpaletten und Icon-Bibliotheken, die eng mit der Markenidentität verknüpft sind [6]. Neben der visuellen Konsistenz ermöglichen Designsysteme durch die zentralisierte Modifikation gestalterischer Parameter eine hohe Adaptivität.

Die Relevanz von Designsystemen wird durch aktuelle Branchendaten von Zeroheight, einem Anbieter einer Plattform für das Management von Designsystemen, unterstrichen. In jährlichen Studien wird die Adaption von Designsystemen in Unternehmen untersucht. Für das Jahr 2025 wurde berichtet, dass bereits 79 % der befragten Organisationen über ein dediziertes Team für Designsysteme verfügen [7].

Die Appsfactory GmbH (Appsfactory) ist eine Digitalagentur, die modernste Technologien einsetzt, um ihren Kunden digitale Lösungen bereitzustellen. Im Rahmen des Produktlebenszyklus nimmt die Entwicklung von Designsystemen eine signifikante Rolle ein. Der Prozess beginnt mit einer umfassenden Beratung der Kundschaft, woraufhin die Erstellung von Designkonzepten erfolgt. Diese werden mittels Low-Fidelity-Prototypen und High-Fidelity-Screendesigns visualisiert. In dieser Phase erfolgt die Erstellung des Designsystems unter Verwendung von Design-Tokens. Diese fungieren als zentrale Bausteine aller UI-Elemente und definieren grundlegende visuelle Eigenschaften wie Farben, Abstände und Typografie [8]. Nach erfolgter finaler Bestätigung durch den Auftraggeber

wird der agile Entwicklungsprozess unter Einbeziehung von DevOps-Prinzipien und einer kontinuierlichen Qualitätssicherung initiiert [9].

1.1 Motivation

Eine detaillierte Betrachtung des internen Entwicklungsprozesses ist notwendig, um die Herausforderungen des aktuellen Arbeitsablaufs zu verstehen. In der Abteilung für mobile Anwendungsentwicklung der Appsfactory wird zur Initialisierung neuer Android-Projekte eine Projektvorlage verwendet. Diese bietet einen Architekturentwurf sowie ein vorab konfiguriertes Build-System. Auf dieser Grundlage erfolgt die Entwicklung der projektspezifischen Funktionalitäten und der Benutzeroberflächen entsprechend der Screendesigns.

Einen kritischen Schritt in der anschließenden UI-Entwicklung stellt die Übertragung des Designsystems aus Figma in den Quellcode dar. Den Kern dieser Aufgabe bildet die Integration der Design-Tokens. Die Relevanz dieser Automatisierung wird durch die Studien von Zeroheight bestätigt: Auf die Frage nach der Automatisierung von Designsystemen gaben die Teilnehmenden an, dass die Implementierung von CI/CD-Pipelines (Continuous Integration/Continuous Delivery) für Design-Tokens die am häufigsten genutzte Methode sei [7].

Diese Branchenentwicklung spiegelt sich auch in der technologischen Strategie der Appsfactory wider. Als Teil der Strategie für die Mobile-Abteilung im Jahr 2025 wurde die Automatisierung der Integration von Design-Tokens und Icons angestrebt. Dieses Vorgehen optimiert den bisherigen manuellen Übertragungsprozess.

Angesichts des wachsenden Marktdrucks und der Erwartungshaltung der Auftraggeber an immer kürzere Entwicklungszyklen stellt die Automatisierung dieses Schrittes eine wirtschaftliche und technische Notwendigkeit dar. Eine automatisierte Lösung steigert die Effizienz der Anwendungsentwicklung. Design-Änderungen werden dadurch ohne Zeitverzögerung und fehlerfrei im Programmcode abgebildet.

1.2 Problemstellung

Die manuelle Überführung von Design-Ressourcen aus Figma in den Programmcode erweist sich als ein zeitaufwendiger und fehleranfälliger Prozess. Ein wesentlicher Grund hierfür ist die oft komplexe und verschachtelte Organisation der Ressourcen, insbesondere der Design-Tokens. Der Umfang eines Projektes kann dabei ausschlaggebend dafür sein, wie viele unterschiedliche Tokens und grafische Ressourcen in einem Designsystem enthalten sind. Die manuelle Synchronisation eines solchen Systems ist in der Regel mit einem hohen Aufwand verbunden.

Darüber hinaus existiert eine technologische Lücke im aktuellen Werkzeugmarkt. Obwohl bereits Plugins und externe Werkzeuge für die Übertragung von Design-Tokens existieren, sind diese meist nicht für die spezifischen internen Arbeitsablauf und Projektvorlagen der Appsfactory geeignet. Zahlreiche dieser Werkzeuge sind auf Web-Frameworks oder die Android-Entwicklung mit XML-basierten Layouts spezialisiert. Im Gegensatz dazu wird die moderne UI-Entwicklung mit Compose unzureichend unterstützt.

Eine weitere Herausforderung stellt die strategische Ausrichtung des Unternehmens auf plattformübergreifende Anwendungsentwicklung dar. Die Appsfactory entwickelt zudem White-Label-Anwendungen. Diese Anwendungen müssen mehrere Markenidentitäten innerhalb einer gemeinsamen Codebasis unterstützen. Man spricht hierbei von einem Multi-Brand-Ansatz.

Derzeit besteht ein Mangel an einem integrierten Werkzeug, welches:

- sowohl native Android- als auch plattformübergreifende Entwicklungen unterstützt;
- die spezifischen Anforderungen von Multi-Brand-Strukturen effizient abbilden kann;
- nahtlos in die bestehenden Arbeitsabläufe im Anwendungsentwicklungsprozess integriert werden kann.

1.3 Zielstellung

Das Ziel dieser Bachelorarbeit ist die Konzeption und Entwicklung eines Werkzeugs, das den Export von Design-Ressourcen aus Figma automatisiert und für die UI-Entwicklung mit Compose aufbereitet. Dieses Werkzeug soll die spezifischen Anforderungen und Bedürfnisse der Appsfactory adressieren und den identifizierten Medienbruch zwischen Design und technischer Umsetzung schließen.

Ein besonderer Fokus liegt dabei auf der Unterstützung von Multi-Brand-Architekturen sowie der nahtlosen Integration in den bestehenden Arbeitsablauf des Unternehmens. Durch die automatisierte Generierung von typsicherem Kotlin-Code sollen Fehlerquellen bei manuellen Arbeitsschnitten eliminiert und die Übereinstimmung zwischen Designs und tatsächlicher Anwendung garantiert werden.

2 Theoretische Grundlagen

Um ein fundiertes Verständnis der vorliegenden Arbeit zu ermöglichen, werden in diesem Kapitel die wesentlichen technologischen und konzeptionellen Grundlagen erläutert. Zunächst wird definiert, welche Design-Ressourcen für die Übertragung relevant sind. Darauf aufbauend wird die REST-basierte Programmierschnittstelle für den automatisierten Datenabruf aus der Design-Software beleuchtet. Anschließend folgt eine Einführung in Compose, um die Anforderungen an die Zielarchitektur zu verdeutlichen. Abschließend werden die Wege der Automatisierung erörtert.

2.1 Figma-Exports

Innerhalb der Appsfactory GmbH bildet Figma das zentrale Werkzeug für den gesamten Designprozess. Figma ist ein cloudbasiertes, kollaboratives Design-Werkzeug, welches die Erstellung, Verwaltung und Wartung von Designsystemen ermöglicht. Es bietet eine zentralisierte Umgebung, in der Design-Ressourcen gemeinschaftlich entwickelt, geteilt und für die spätere Implementierung bereitgestellt werden [6].

Unter dem Begriff Figma-Exports werden im Rahmen der vorliegenden Arbeit alle Design-Ressourcen zusammengefasst, die automatisiert aus diesem Designprozess extrahiert und in den Entwicklungsprozess integriert werden sollen. Dafür sind insbesondere zwei Kategorien von zentraler Bedeutung: Design-Tokens und grafische Ressourcen (in diesem Fall, Icons).

2.1.1 Design-Tokens

Die Design Tokens W3C Community Group (DTCG) definiert Design-Tokens als eine „zentrale Wissensquelle zur Benennung und Speicherung einer Design-Entscheidung, die so verteilt wird, dass Teams sie über verschiedene Design-Werkzeuge und Programmiersprachen hinweg nutzen können“ [10].

Präzisiert wird dies im Design Tokens Format Module, einer Spezifikation der DTCG. Demnach ist ein Token „eine Information, die mit einem menschenlesbaren Namen verknüpft ist, im einfachsten Fall handelt es sich um ein Name-Wert-Paar“ [11].

Für das in dieser Arbeit entwickelte Werkzeug sind insbesondere Tokens für Farben, Abstände und Typografie von Relevanz.

Ein wesentliches Merkmal in Figma ist die Fähigkeit zur Referenzierung (auch Aliasing genannt). Hierbei kann ein Token auf den Wert eines anderen Tokens verweisen, anstatt einen absoluten Wert zu speichern. Technisch ergeben sich daraus folgende Möglichkeiten:

- es gibt keine Limitierung, über wie viele Ebenen eine Referenzkette führen kann;
- ein einzelner Token kann von beliebig vielen anderen Tokens referenziert werden.

Diese Mechanismen ermöglichen den Aufbau von Design-Token-Strukturen, die Änderungen an zentraler Stelle im gesamten System propagieren.

Um die Skalierbarkeit zu gewährleisten, empfiehlt Figma folgende Organisation der Tokens in drei Ebenen [12]:

- **Primitive Tokens**: Diese Tokens beschreiben die grundlegenden visuellen Eigenschaften und Werte eines Designs. Sie definieren die Basiswerte, wie beispielsweise jede einzelne Farbe, die in einer Anwendung oder Marke vorkommt. Ebenso enthalten primitive Tokens für Abstände alle verfügbaren Werte für Innen- und Außenabstände (Paddings und Margins). Ein wesentliches Merkmal ist, dass primitive Tokens ausschließlich als Referenz dienen. Sie bilden das Fundament für alle weiteren Token-Ebenen und werden nicht direkt auf UI-Elemente angewendet.
- **Semantische Tokens**: Diese Ebene beschreibt die konkrete Verwendung und den Kontext eines Tokens. Semantische Bezeichnungen vermitteln die Bedeutung und den Zweck eines Assets. Sie geben vor, wie und an welcher Stelle ein Token im Design-Prozess eingesetzt werden soll (z. B. `color-text-primary` statt eines konkreten Blauwerts). Dadurch wird die Logik des Designs vom hartkodierten Wert entkoppelt.
- **Komponenten-Tokens**: Diese Tokens definieren spezifische Werte für einzelne UI-Komponenten, wie beispielsweise die Hintergrundfarbe eines bestimmten Buttons (z. B. `button-primary-background-color`). Diese Ebene ist sehr detailliert und findet vor allem in komplexen Systemen auf Enterprise-Ebene Anwendung. Da sie sehr spezifisch sind, sind Komponenten-Tokens nicht für jedes Designsystem zwingend erforderlich, bieten aber eine maximale Kontrolle über das visuelle Erscheinungsbild einzelner Elemente.

In Figma werden Design-Tokens meistens in Form von Variablen gespeichert und in sogenannten Collections organisiert. Innerhalb der Collection besteht die Möglichkeit, eine Gruppierung der Variablen vorzunehmen. Die weitere Struktur hängt von den individuellen Anforderungen des Projekts ab, allerdings werden semantische Tokens und Komponenten-Tokens häufig auf derselben logischen Ebene definiert.

2.1.2 Organisation von Collections mit Tokens

Zur Optimierung der Überführung von Design-Tokens in den Quellcode wurde eine spezifische Hierarchie für Figma-Collections etabliert. Diese Struktur basiert auf den von Figma empfohlenen besten Praktiken sowie einer engen interdisziplinären Zusammenarbeit zwischen den Design- und Entwicklungsteams der Appsfactory. Da die automatisierten Figma-Exports unmittelbar von den Daten in der Figma-Datei abhängen, ist die Struktur und Konsistenz von entscheidender Bedeutung.

In diesem Prozess definiert das Design-Team die inhaltliche Organisation der Ressourcen. Die Entwicklung adaptiert das Werkzeug an die Bedürfnisse der Designer. Im Gegenzug legt die Entwicklung technische Rahmenbedingungen fest, um eine reibungslose Übertragung zu gewährleisten. Hierzu zählen verbindliche Namenskonventionen: Token-Namen dürfen beispielsweise nicht mit Ziffern beginnen, keine Sonderzeichen enthalten und sollten vorzugsweise in englischer Sprache verfasst sein. Die Einhaltung dieser Konventionen ist essenziell, da syntaktische Unstimmigkeiten in Figma unmittelbar zu Kompilierfehlern oder Warnmeldungen innerhalb der Entwicklungsumgebung führen würden.

Die Arbeit mit strukturierten Collections ist für viele Designer ein neuer methodischer Ansatz. Daher wurde die Komplexität auf ein für den Arbeitsalltag angemessenes Maß begrenzt. Eine zu tiefe Verschachtelung würde die Akzeptanz und die konsistente Pflege des Systems gefährden. Aus diesem Grund wird die Struktur kontinuierlich auf Basis des Feedbacks beider Fachbereiche evaluiert und iterativ verfeinert.

Farben Die Organisation von Tokens für Farben wird in Figma durch drei aufeinander aufbauende Collections realisiert, die eine skalierbare Steuerung ermöglichen.

Die unterste Ebene dieser Hierarchie bildet die Basis-Collection, in welcher die primitive Tokens hinterlegt werden. Um Redundanzen zu vermeiden, wird im Regelfall eine einzige Basis-Collection für alle Marken verwendet. Die Verwendung einer gemeinsamen Collection ist bei ähnlichen Farbpaletten oder Abstandsdefinitionen sinnvoll. Diese Ebene verfügt über lediglich einen Modus (Default-Modus), wobei die Variablen zur besseren Übersichtlichkeit in Gruppen unterteilt werden. Sollten sich die Basisfarben verschiedener Marken jedoch fundamental unterscheiden, können markenspezifische Basis-Collections erstellt werden. In Abbildung 1 ist eine beispielhafte Struktur einer Basis-Collection dargestellt. Die Variablen sind in die Gruppen „general“, „purple“, „green“ gegliedert.

Auf die Basis-Collection folgt die Marken-Collection, in der die semantische Zuweisung der Werte erfolgt. Ein Merkmal dieser Ebene ist die Nutzung von Figma-Modi, die es erlauben, einem semantischen Token je nach Kontext unterschiedliche Referenzen auf Primitiv-

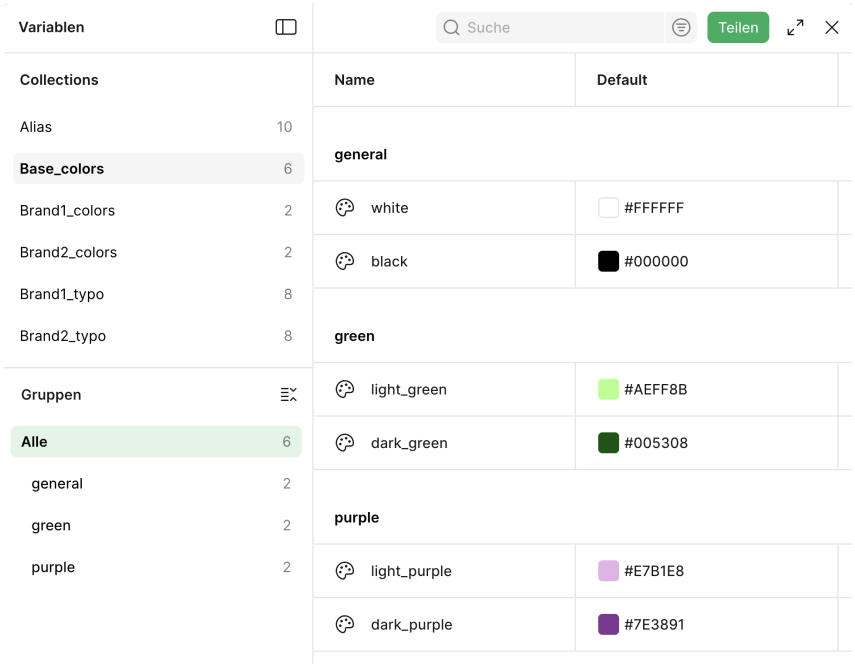
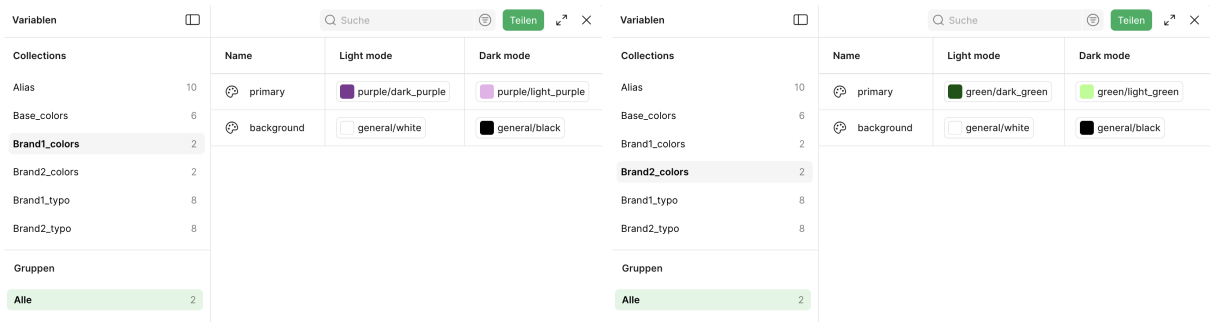


Abbildung 1: Beispiel der Basis-Collection von Tokens für Farben

Tokens zuzuweisen. Klassische Anwendungsfälle hierfür sind die Differenzierung zwischen Hell- und Dunkelmodus oder die Implementierung spezifischer Kontrasteinstellungen für die Barrierefreiheit. Für jede Marke wird eine eigene Marken-Collection angelegt, wobei eine strikt identische Nomenklatur der Tokens über alle Marken hinweg erforderlich ist. Abbildung 2 zeigt die Struktur der Marken-Collections für zwei verschiedene Marken.



(a) Brand1 (b) Brand2

Abbildung 2: Beispiel der Marken-Collection von Tokens für Farben

Den Abschluss der Hierarchie bildet die Alias-Collection (oder Multi-Brand-Collection). Innerhalb dieser Collection werden die verschiedenen Markenidentitäten selbst als Modi definiert. Jeder Marken-Modus verweist dabei auf die entsprechenden Variablen der jeweiligen Marken-Collection. Durch diese Referenzierung können bis zu 40 verschiedene Marken oder Modi innerhalb eines einzigen Designsystems verwaltet werden. Diese Struktur ermöglicht die Skalierbarkeit eines Projekts innerhalb einer einzigen Figma-Datei. In Abbildung 3 ist Beispiel einer Alias-Collection zu sehen.

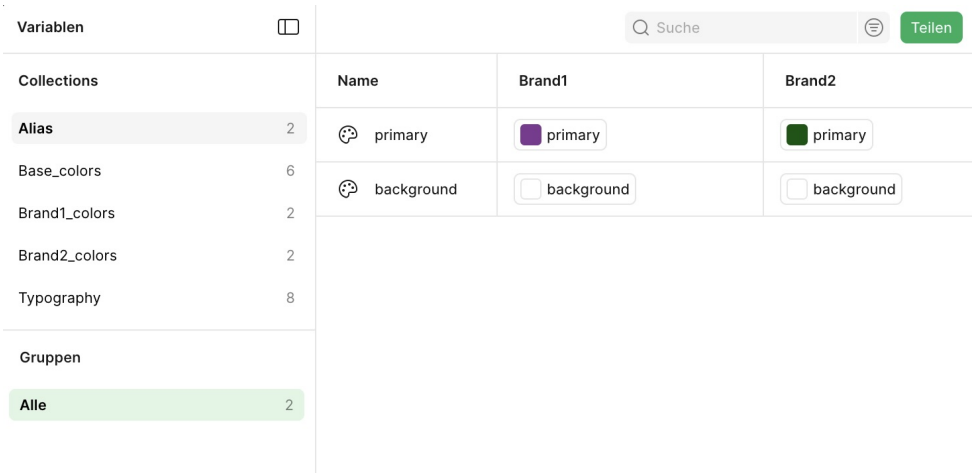


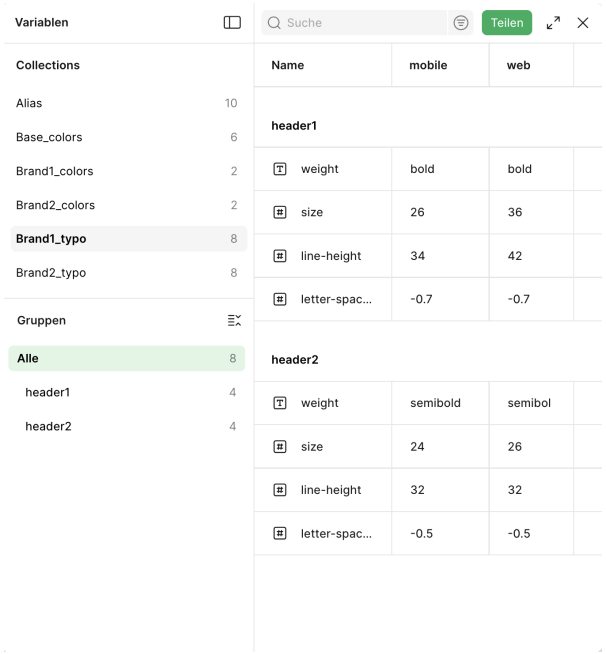
Abbildung 3: Beispiel der Alias-Collection von Tokens für Farben

Typografie Die Definition der Typografie umfasst verschiedene Kategorien, die innerhalb des Projekts implementiert werden, wie beispielsweise Titel oder Fließtexte für Beschreibungen. Grundlegende Eigenschaften wie Schriftart, Schriftgröße und Zeilenhöhe werden zunächst als Tokens in Collections hinterlegt.

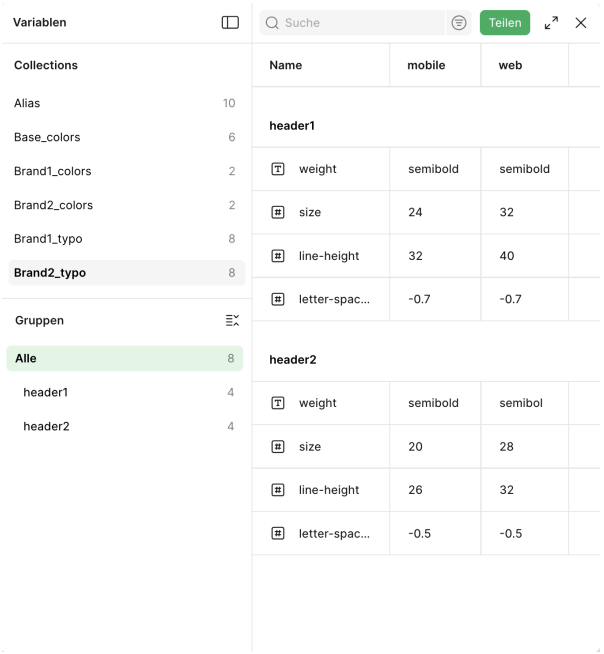
Die ursprüngliche Hierarchie und Struktur der Collections für Typografie glich der Struktur der Collections für Farben. Dies erforderte zunächst die Speicherung numerischer Werte in einer Basis-Collection, die Referenzierung durch semantische Tokens in der Marken-Collection und die abschließende Zusammenführung in der Alias-Collection.

Die praktische Anwendung zeigte jedoch, dass diese Struktur für Designer eine zu hohe Komplexität aufweist. Zudem benötigt Typografie im Gegensatz zu Farben keine separate Basis-Collection, da diese in diesem Kontext lediglich die Komplexität erhöhen würde, ohne einen nennenswerten funktionalen Mehrwert zu bieten. Aus diesem Grund werden die typografischen Werte nun direkt in der Marken-Collection definiert.

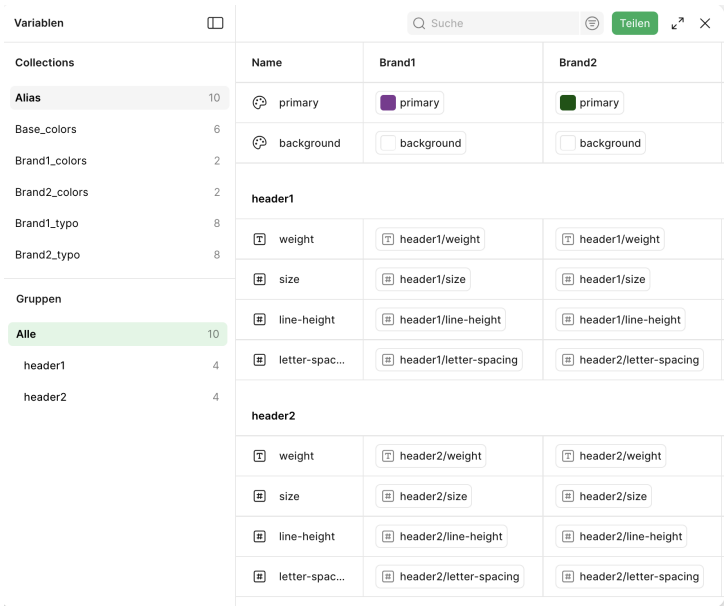
Die Anzahl der Modi in dieser Collection korrespondiert dabei mit der Anzahl der unterstützten Bildschirmgrößen. Unterschiedliche Modi sind hierbei nur dann relevant, wenn das Produkt für verschiedene Plattformen oder Endgeräte, wie beispielsweise Mobile, Web oder Tablet, optimiert werden soll. Die Variablen werden dabei nach Textstilen gruppiert. Innerhalb von Figma ermöglichen es Stile, eine Menge von Eigenschaften zu definieren, die konsistent über verschiedene Dateien und Projekte hinweg wiederverwendet werden können. Textstile dienen dazu, die Typografie innerhalb des Designs festzulegen. Die in der Marken-Collection erstellten Variablen werden diesen Textstilen zugewiesen.



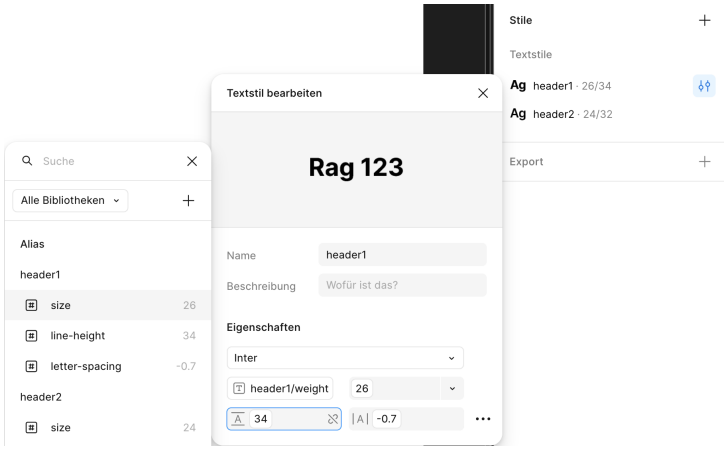
(a) Marken-Collection für Brand1



(b) Marken-Collection für Brand2



(c) Alias-Collection



(d) Anwendung von Variablen an Textstil

Abbildung 4: Beispiel der Struktur von Collections von Tokens für Typografie

Die Abbildungen 4a und 4b zeigen beispielhafte Marken-Collections für Tokens für Typografie. In dieser Konfiguration sind zwei Modi vorhanden, die zur Differenzierung zwischen verschiedenen Endgeräten dienen. Die Variablen sind nach den entsprechenden Textstilen gruppiert. Beispiele hierfür sind „header1“ und „header2“. Diese markenspezifischen Definitionen werden anschließend in einer zentralen Alias-Collection zusammengeführt, wie in Abbildung 4c dargestellt. In der Alias-Collection können alle Tokentypen gespeichert werden.

Abbildung 4d illustriert die finale Verknüpfung innerhalb von Figma. Hier ist ersichtlich, wie der Textstil auf die definierten Variablen referenziert. Im Eigenschaften-Panel des Textstils werden die Werte der verknüpften Variablen angezeigt.

2.1.3 Icons

Standardmäßig werden die innerhalb des Designsystems genutzten Icons direkt in der Figma-Datei hinterlegt. Zur Übersichtlichkeit werden diese meist an einer zentralen Stelle, z. B. auf einer dedizierten Seite, organisiert. Ein Beispiel für die organisatorische Strukturierung von Icons innerhalb einer Figma-Datei ist in Abbildung 5 dargestellt.

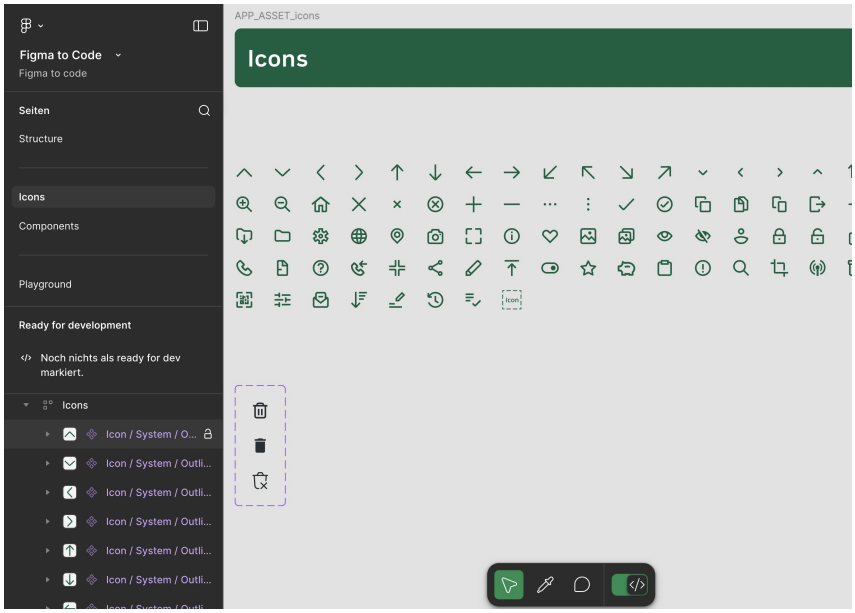


Abbildung 5: Beispiel der Icons-Organisation in Figma-Datei

Obwohl Figma native Funktionen für den Export dieser Ressourcen zur Verfügung stellt, erfordert dies im herkömmlichen Arbeitsablauf einen manuellen Aufwand auf der Entwicklungsseite. Die Entwickler müssen die benötigten grafischen Ressourcen extrahieren, in die passenden Formate überführen und anschließend manuell in die entsprechende Verzeichnisstruktur des Quellcodes integrieren.

2.2 Die Figma REST API

Um Figma-Exports automatisiert in den Entwicklungsprozess zu integrieren, bietet Figma eine REST-basierte Programmierschnittstelle (Figma API) an [13].

Für die Funktionalität des in dieser Arbeit entwickelten Werkzeugs werden die Endpunkte in vier funktionale Gruppen unterteilt:

1. Variables API
   Die Variables API stellt das Hauptelement für den Export von Design-Tokens dar. Variablen speichern wiederverwendbare Werte, die auf verschiedene Design-Eigenschaften von Elementen angewendet werden können.

   - GET local variables (`GET /v1/files/:file_key/variables/local`)
     Dieser Endpunkt dient dazu, alle lokal in einer Datei definierten Variablen aufzulisten. Es werden keine Query-Parameter zur Filterung der Ergebnisse bereitgestellt. Dies führt dazu, dass bei einer Anfrage alle innerhalb der Datei erstellten Variablen übermittelt werden.

2. File und Node API
   Jede Datei in Figma ist als Baumstruktur von Knoten organisiert. Die Wurzel bildet ein `DOCUMENT`-Knoten, gefolgt von `CANVAS`-Knoten, die wiederum verschiedene Ebenen und Objekte enthalten.

   - GET file nodes (`GET /v1/files/:file_key/nodes`)
     Durch die Verwendung des Query-Parameters `ids` können einzelne Knoten oder deren Unterbäume adressiert werden. Eine beispielhafte Anfrage der Form `GET /v1/files/:file_key/nodes?ids=1:2,1:3` liefert die Daten der referenzierten Knoten als JSON-Objekt zurück.

3. Image API
   Da Icons im Gegensatz zu Tokens grafische Ressourcen sind, müssen sie als Bilddateien gerendert werden.

   - GET image (`GET /v1/images/:file_key`)
     Dieser Endpunkt rendert die gewünschten Objekte aus einer Datei und stellt sie über eine temporäre URL für den Download bereit. Über den Query-Parameter `ids` werden die Objekte anhand ihrer Node-ID adressiert. Dabei unterstützt der Endpunkt die Abfrage mehrerer Ressourcen innerhalb eines einzelnen Aufrufs. Eine beispielhafte Anfrage hat die Form `GET /v1/images/:key?ids=1:2,1:3,1:4`. Über den Query-Parameter `format` kann das Format der gerenderten Datei spezifiziert werden. Die Formate JPG,

PNG, PDF und SVG werden hierbei unterstützt.

4. Styles API
   Stile definieren eine Menge von Eigenschaften, in dem Fall Typografie-Vorgaben, die auf Knoten angewendet werden können.

   - GET file styles (`GET /v1/files/:file_key/styles`)
     Liefert eine Liste aller veröffentlichten Stile innerhalb einer Datei-Bibliothek.

2.3 UI-Entwicklung mit Compose

Nachdem die Design-Ressourcen erfolgreich über die Figma-API extrahiert wurden, erfolgt deren Überführung in den Quellcode des Zielprojekts. Hierbei ist die technologische Umgebung von entscheidender Bedeutung. Diese definiert die Art der Verwendung von Tokens. Zudem bilden spezifische Frameworks die Basis für die Integration. Innerhalb der Appsfactory werden neue Projekte standardmäßig auf Basis von Kotlin und den Frameworks Jetpack Compose oder Compose Multiplatform realisiert. In diesem Kapitel werden die theoretischen Grundlagen dieser Frameworks erläutert. Ein Fokus liegt dabei auf der Unterstützung für Designsysteme. Abschließend wird die standardisierte Implementierung von Design-Tokens sowie die Integration grafischer Ressourcen in der betrieblichen Praxis dargelegt.

2.3.1 Jetpack Compose und Compose Multiplatform

Jetpack Compose ist das von Google empfohlene, moderne Framework zur Erstellung nativer Benutzeroberflächen auf Android. Es bricht mit dem traditionellen imperativen Ansatz (XML-Layouts) und basiert stattdessen auf einem deklarativen Programmierparadigma [14].

In Jetpack Compose wird die Benutzeroberfläche durch sogenannte Composable Functions (kurz: Composables) definiert. Diese Funktionen werden mit der `@Composable`-Annotation gekennzeichnet. Anstatt UI-Elemente manuell zu instanziieren und deren Zustand zu verwalten, beschreiben Composables die UI basierend auf den übergebenen Daten [14]. Ein Beispiel für das Composable ist im Listing 1 zu sehen.

```
1 @Composable
2 fun Greeting() {
3     Text(text = "Hello World!")
4 }
```

Listing 1: Composable-Funktion Beispiel

Ändern sich die zugrunde liegenden Daten, löst das Framework eine Recomposition aus,

wodurch die betroffenen Teile der UI automatisch aktualisiert werden [14]. Dieser datenzentrierte Ansatz ist die technische Voraussetzung dafür, dass Design-Tokens direkt als Datenquelle für das visuelle Erscheinungsbild dienen können.

Eine Erweiterung von Jetpack Compose stellt das Compose Multiplatform Framework (CMP) dar. Laut der offiziellen Dokumentation handelt es sich hierbei um ein von JetBrains entwickeltes Framework zur plattformübergreifenden Nutzung von Kotlin-UI-Code. Compose Multiplatform stellt nahezu alle gängigen Compose-APIs bereit. Dies erlaubt es den Entwicklern, geteilten UI-Code zu verfassen, der gleichermaßen auf Android, iOS, dem Desktop sowie im Web lauffähig ist [15].

Da der Fokus dieser Arbeit auf der UI-Entwicklung liegt, werden die spezifischen Details der Multiplattform-Entwicklung mittels Kotlin Multiplatform nicht weiter vertieft. Im Rahmen dieser Arbeit wird die Bezeichnung *Compose* vereinheitlichend für beide Frameworks verwendet. Diese Verallgemeinerung gilt jedoch ausschließlich für Funktionen, die in beiden Frameworks identische Logik haben. Ein wesentlicher Unterschied zwischen Jetpack Compose und CMP liegt in der Art der Handhabung von Ressourcen. Dieser wird im folgenden Kapitel 2.3.3 detailliert erläutert.

2.3.2 Designsystem-Integration mit Compose

Das Designsystem einer mit Compose erstellten Anwendung wird durch das Konzept des *Themings* bestimmt. Mithilfe von Theming und wiederverwendbaren Komponenten kann ein konsistentes Erscheinungsbild über die gesamte Anwendung hinweg gewährleistet werden [14]. Dabei wird zwischen dem von Google bereitgestellten Standard und benutzerdefinierten Implementierungen unterschieden. Im Folgenden wird erläutert, welche Funktionen und Lösungen das Framework bereits nativ bereitstellt, welche konzeptionellen Lücken bestehen bleiben und wie diese in der Praxis der Appsfactory standardmäßig adressiert werden.

Material Theming Compose bietet eine native Implementierung von Material Design 3 (M3), dem aktuellen Designsystem von Google [8]. Ein zentrales Element ist die `MaterialTheme`-Komponente, die Subsysteme für Farbschemata, Typografie und Formen bereitstellt [14].

Material Design stellt hierfür eine Vielzahl von vorkonfigurierten Komponenten zur Verfügung (M3-Komponente). Diese Komponenten entsprechen standardmäßig den Material-Spezifikationen und ermöglichen eine effiziente Erstellung der Benutzeroberflächen. Durch die Modifikation der `MaterialTheme`-Komponente werden Änderungen automatisch auf alle M3-Komponenten propagiert.

```
MaterialTheme(
    colorScheme = lightColorScheme(/* ... */),
    typography = AppTypography,
    shapes = AppShapes
) {
    // App Inhalt
}
```

Listing 2: MaterialTheme-Komponente Beispiel

Ein einfacher Button kann ohne zusätzliche Konfiguration wie folgt implementiert werden:

```
@Composable
fun M3Button() {
    Button(onClick = { /*..*/ }) {
        Text(text = "My Button")
    }
}
```

Listing 3: Beispiel für Nutzung einer M3-Komponente

M3 fördert dabei die Flexibilität durch Parametrisierung. Während für alle Komponenten Standardwerte hinterlegt sind, können diese über spezifische APIs individuell angepasst werden. Das folgende Beispiel zeigt die Modifikation der Hintergrundfarbe eines Buttons über das `colors`-Argument:

```
@Composable
fun ColorfulButton(){
    Button(
        onClick = { /*..*/ },
        colors = ButtonDefaults.buttonColors(
            containerColor = MaterialTheme.colorScheme.primaryContainer
        )
    ) {
        Text(text = "Customized Button")
    }
}
```

Listing 4: Beispiel für die Anpassung von Attributen einer M3-Komponente

Benutzerdefiniertes Theming In der Praxis zeigt sich oft, dass die Standard-M3-Komponenten nicht ausreichen, um spezifische Figma-Vorgaben exakt abzubilden. Viele Designsysteme nutzen eigene Logiken, die über das M3 hinausgehen. Daher ist es erforderlich, individuelle Komponenten zu entwickeln und das Theming individuell zu konfigurieren.

In der internen Projektvorlage der Appsfactory GmbH wird hierfür ein benutzerdefiniertes `AppTheme`-Composable implementiert. Dieses stellt die projektspezifischen Design-Tokens für Farben, Typografie und Abstände zentral bereit. Um die Hierarchie der Figma-

Collection abzubilden, werden diese Tokens oft innerhalb verschachtelter Datenklassen definiert.

```
data class AppColors(
    val background: AppColors.Background,
    val accent: Color,
) {
    data class Background(
        val primary: Color,
        val secondary: Color,
    )
}
```

Listing 5: Benutzerdefinierte Datenklasse für Tokens einer Farbe

Auf Basis dieser Modelle werden Instanzen für verschiedene Modi, wie im Listing 6 zu sehen, erstellt. Diese enthalten die konkreten, aus Figma extrahierten Werte (z. B. Hex-Farbcodes).

```
val AppColorLightTokens = AppColors(
    background = AppColors.Background(
        primary = Color(color = 0xFFD0BCFF),
        secondary = Color(color = 0xFFCCC2DC),
    ),
    accent = Color(color = 0xFF381E72),
)
```

Listing 6: Instanz von AppColors-Datenklasse

Dann werden diese Instanzen an `AppTheme`-Composable übergeben. Die technische Bereitstellung erfolgt über das Konzept des `CompositionLocal`. Dieses Prinzip erlaubt es, Daten zu überreichen, ohne sie explizit als Parameter an jedes einzelne Composable übergeben zu müssen. Die Definition eines Themes mit benutzerdefinierten Farben könnte wie im Listing 7 aussehen.

UI-Komponenten können anschließend an jeder Stelle im Projekt auf diese Tokens referenzieren, wodurch eine saubere Trennung zwischen Design-Daten und UI-Logik erreicht wird. Das ist im Listing 8 zu sehen.

2.3.3 Integration von Ressourcen

Während die native Android-Entwicklung fest in das plattformspezifische Android-Ressourcensystem integriert ist, strebt CMP eine weitgehende Plattformunabhängigkeit an. In einem nativen Android-Projekt werden Ressourcen im Verzeichnis `res` gespeichert und über die kompilierte `R`-Klasse adressiert. In CMP-Projekten kommt hingegen das Verzeichnis `composeResources` zum Einsatz. Nach dem Build-Prozess des Projekts ermöglicht eine generierte `Res`-Klasse den Zugriff auf diese Ressourcen.

```kotlin
val LocalAppColors = staticCompositionLocalOf<AppColors> {
    error("No AppColors definition provided")
}

@Composable
fun AppTheme(
    /* ... */
    content: @Composable () -> Unit
) {
    CompositionLocalProvider(
        LocalAppColors provides AppColorLightTokens,
    ) {
        MaterialTheme(
            content = content
        )
    }
}

object AppTheme {
    val colors: AppColors
        @Composable
        get() = LocalAppColors.current
}
```

Listing 7: Darstellung eines benutzerdefinierten Themes

```kotlin
@Composable
fun CustomButton(){
    Button(
        onClick = { /*..*/ },
        colors = ButtonDefaults.buttonColors(
            containerColor = AppTheme.colors.background.primary
        )
    ) {
        Text(text = "Button with AppTheme colors")
    }
}
```

Listing 8: Darstellung eines benutzerdefinierten Themes

Zunächst wird die Integration der für diese Arbeit relevanten Ressourcen erklärt. Hierzu zählen Icons und Schriftarten.

Icons Die Android-Entwicklung unterstützt die direkte Verwendung von skalierbaren Vektorgrafiken (SVG) nicht. Icons müssen üblicherweise als XML-Dateien importiert werden. Zur Konvertierung von SVG-Dateien bietet Android Studio integrierte Werkzeuge wie das „Vector Asset Studio“ an. Dieses Werkzeug übernimmt die automatische Transformation von SVG zu XML [14]. Dieser Prozess ist jedoch unmittelbar an die grafische Benutzeroberfläche der Entwicklungsumgebung gebunden. Ein wesentlicher Nachteil im Kontext der angestrebten Automatisierung besteht darin, dass diese nativen Konver-

tierungswerkzeuge nicht extern oder programmatisch über eine Befehlszeilenschnittstelle (CLI) angesteuert werden können. Ohne externe Bibliotheken ist ein automatisierter Export aus Figma daher schwer realisierbar.

Compose bietet hierzu eine Alternative durch die Verwendung der Klasse `ImageVector`. Im Gegensatz zu klassischen XML-Ressourcen werden Icons hierbei als Instanzen einer Kotlin-Klasse repräsentiert [14]. Da die Figma API Icons als gerenderte SVG-Dateien bereitstellen kann, ist für einen vollständig automatisierten Workflow ein Werkzeug zur Transformation dieser Bilddaten in Kotlin-Code erforderlich.

An dieser Stelle setzt die Open-Source-Bibliothek *Valkyrie* an. Valkyrie ermöglicht die automatisierte Konvertierung von SVG-Dateien in `ImageVector`-Instanzen [16]. Ein entscheidender Vorteil dieses Ansatzes liegt in der Unterstützung der plattformübergreifenden Entwicklung. Dies gewährleistet, dass die aus Figma extrahierten grafischen Ressourcen ohne zusätzliche manuelle Anpassungen in Projekten für verschiedene Plattformen verwendet werden können.

Schriftarten Die Einbindung von Schriftarten unterscheidet sich zwischen den Frameworks grundlegend in der Art des Zugriffs auf Ressourcen. In Jetpack Compose werden Schriftarten im Ordner `res/font` abgelegt. Laut der Dokumentation erfolgt der Zugriff über den `font`-Ressourcentyp, beispielsweise mittels `R.font.myfont`. Auf dieser Basis wird über die `Font`-API eine `FontFamily` erstellt [14]. Diese dient als Grundlage für die Definition von Typografie-Tokens. Listing 9 zeigt die schematische Implementierung in Jetpack Compose.

```
// Definition und Laden der Schriftarten
private val regular = Font(R.font.raleway_regular, FontWeight.W400)
private val medium = Font(R.font.raleway_medium, FontWeight.W500)

// Erstellung einer FontFamily
private val appFontFamily = FontFamily(regular, medium)

// Benutzerdefinierte Typografie
data class AppTypography(
    val headline1: TextStyle,
    val headline2: TextStyle
)

val TypographyTokens = Typography(
    headline1 = TextStyle(fontFamily = appFontFamily),
    headline2 = TextStyle(fontFamily = appFontFamily)
)
```

Listing 9: Typografie-Tokens für Android

In CMP werden Schriftarten im Verzeichnis `composeResources/font` gespeichert. Das

Laden einer Schriftart erfolgt über ein `Font`-Composable. Deshalb müssen auch alle abhängigen Komponenten, wie Typografie-Tokens, als Composable definiert werden. Ein Design-Token für Typografie ist in CMP somit keine Variable, sondern ein Composable. Listing zeigt 10 die schematische Implementierung in CMP.

```
@Composable
private fun fontFamily(): FontFamily = FontFamily(
  Font(Res.font.raleway_regular, W400),
  Font(Res.font.raleway_medium, W500),
)

@Composable
fun typographyTokens(): AppTypography {
  val fontFamily = fontFamily()

  return AppTypography(
    headline1 = TextStyle(fontFamily = appFontFamily),
    headline2 = TextStyle(fontFamily = appFontFamily)
  )
}
```

Listing 10: Typografie-Tokens für CMP

2.4 Unterstützung der mehreren Marken für Android-Anwendungen

Die Appsfactory entwickelt für ihre Kunden auch sogenannte White-Label-Anwendungen. Gemäß Stein u. a., „White-Label-Produkte werden nicht exklusiv für ein Unternehmen produziert, sondern ebenfalls für zahlreiche andere Unternehmen mit unterschiedlichen Markennamen“.

Technisch wird dies unter Android durch Build-Flavors und Build-Varianten realisiert. Jede Build-Variante repräsentiert eine unterschiedliche Version der Anwendung. Eine Variante ist dabei eine Kombination aus einem Build-Typ und einem Build-Flavor. Während ein Build-Flavor (z. B. eine spezifische Marke) markenspezifische Funktionen und Ressourcen definiert, legt der Build-Typ (z. B. „debug“) Einstellungen für den Build-Prozess und das Packaging fest. Die funktionale Kernlogik verbleibt dabei in gemeinsamen Modulen.

Laut offizieller Dokumentation werden der Quellcode und die Ressourcen eines Moduls logisch in sogenannten Source Sets gruppiert. Das Source Set `src/main/` enthält den Code und die Ressourcen für alle Varianten einer Anwendung. Durch zusätzliche Source Sets lassen sich Dateien für spezifische App-Versionen organisieren. Zur Erstellung einer Version führt das Build-System die Inhalte verschiedener Verzeichnisse zusammen. Für die Version „brand1Debug“ kombiniert das System beispielsweise Ressourcen aus folgenden Pfaden:

- `src/brand1Debug/` - die spezifische Build-Variante,
- `src/debug/` - der Build-Typ,
- `src/brand1/` - der Build-Flavor,
- `src/main/` - die gemeinsame Basis.

Durch die Nutzung dieser Source Sets werden markenspezifische Ressourcen wie Icons oder unterschiedliche Token-Instanzen für das `AppTheme` geladen. Das in dieser Arbeit entwickelte Werkzeug integriert für jeden konfigurierten Build-Flavor automatisch die passenden Design-Tokens aus der jeweiligen Marken-Collection in Figma.

Aktuell unterstützen Projekte auf Basis von CMP keine nativen Multi-Brand-Strukturen, die mit dem Android-Build-System vergleichbar sind. Aus diesem Grund berücksichtigt die vorliegende Arbeit die Unterstützung der mehreren Marken ausschließlich für native Android-Projekten.

2.5 Generierung von Kotlin-Quelldateien

Um die aus Figma extrahierten Design-Tokens in den Entwicklungsprozess zu überführen, ist eine Transformation in Quellcode erforderlich. Ein zentrales Ziel hierbei ist die Gewährleistung der Typsicherheit zur Kompilierzeit. Für die Realisierung dieser Anforderung wird die Bibliothek *KotlinPoet* verwendet.

KotlinPoet ist eine Java- und Kotlin-API zur Generierung von `.kt`-Quelldateien. Sie ermöglicht es, Datenklassen, Objekte oder Funktionen programmatisch zu erstellen [17]. Durch die Verwendung von KotlinPoet wird sichergestellt, dass der generierte Code stets syntaktisch korrekt ist und aktuelle Sprachmerkmale (z. B. Default-Werte oder spezifische Modifikatoren) unterstützt.

```
val colorClass = TypeSpec.classBuilder("AppColors")
    .addModifiers(KModifier.DATA)
    .addProperty(PropertySpec.builder("accent", Color::class)
        .initializer("Color(0xFF381E72)")
        .build())
    .build()

val file = FileSpec.builder("de.af.generated", "AppColors")
    .addType(colorClass)
    .build()
```

Listing 11: Beispiel für die Code-Generierung mit KotlinPoet

2.6 Interaktionsmöglichkeiten

Damit das entwickelte Werkzeug in den Arbeitsablauf der Appsfactory GmbH integriert werden kann, muss es in einem flexiblen und benutzerfreundlichen Format bereitgestellt werden. Hierbei werden zwei Bereitstellungswege unterschieden.

2.6.1 Als Gradle-Plugin

Gradle fungiert als das primäre Build-System für die Android-Entwicklung und stellt die notwendige Infrastruktur bereit, um Quellcode und Ressourcen automatisiert zu analysieren, zu kompilieren und in eine ausführbare Anwendung zu transformieren [14]. Ein wesentliches Merkmal von Gradle ist dessen Erweiterbarkeit durch Plugins. Gemäß der offiziellen Dokumentation kapseln Plugins die Logik für spezifische Aufgaben oder Integrationen, wie das Kompilieren von Quellcode, das Durchführen von Tests oder das Deployment von Artefakten. Durch die Anwendung von Plugins können Entwickler zusätzliche Funktionalitäten effizient in den Build-Prozess integrieren, ohne komplexe Logiken von Grund auf neu implementieren zu müssen [18].

Die Realisierung von Werkzeugen als Gradle-Plugin ermöglicht somit eine nahtlose Einbindung in den bestehenden Entwicklungszyklus.

Da Figma-Dateien oft eine Vielzahl von Collections und Modi enthalten, muss das Werkzeug individuell konfigurierbar sein. Hierfür kommen Gradle-Extensions zum Einsatz. Diese dienen als programmatische Schnittstelle, die es den Entwicklern erlaubt, spezifische Parameter direkt innerhalb der Konfigurationsdatei `build.gradle.kts` zu definieren. Somit können gezielt Collections, Modi oder Markenidentitäten für den Export ausgewählt werden, die für die UI-Entwicklung relevant sind.

```
figmaResGen {
    figmaFileKey = "FILE_KEY"
    packageInfo.base = "de.af.ui.common"
    tokens {
        register("colors") {
            className = "AppColors"
            figmaCollectionName = "colors_tokens"
            tokenType = "color"
        }
    }
}
```

Listing 12: Beispiel einer Gradle-Extension-Konfiguration

Die eigentliche Ausführung der im Plugin implementierten Logik wird schließlich durch die Initiierung dedizierter Gradle-Aufgaben realisiert. Diese Aufgaben fungieren als ausführbare Einheiten innerhalb der Build-Pipeline und stoßen den automatisierten Export-

prozess an, sobald sie entweder manuell durch den Entwickler oder automatisiert durch ein Build-Skript aufgerufen werden [18].

2.6.2 Als lokale Applikation

Neben der Integration als Plugin ist eine lokale Nutzung des Werkzeugs vorgesehen. In diesem Szenario erfolgt die Steuerung über YAML-Dateien.

```yaml
figmaFileKey: FILE_KEY
outputDir: /path/to/project
package:
    base: de.af.ui.common
tokens:
    colors:
      - className: AppColors
        figmaCollectionName: colors_tokens
```

Listing 13: Beispiel einer YAML-Konfiguration

Die Unterstützung von YAML-Konfigurationen ist besonders im Kontext von CI/CD-Pipelines von Bedeutung. Es ist darauf zu achten, dass die YAML-basierte Konfiguration auch innerhalb der Gradle-Plugin-Version des Werkzeugs unterstützt wird, um eine konsistente Konfigurationslogik über verschiedene Umgebungen hinweg zu gewährleisten.

3 Stand der Forschung und Technik

Die automatisierte Überführung von Design-Ressourcen in den Quellcode ist ein essenzieller Bestandteil der modernen UI-Entwicklung. In der Fachliteratur wird dieser Prozess oft als Design-to-Code bezeichnet. Dieser Begriff beschreibt die automatisierte Umwandlung von visuellen Komponenten und Layout-Vorgaben aus digitalen Design-Werkzeugen in produktionsbereiten Quellcode. Sowohl wissenschaftliche Arbeiten als auch industrielle Lösungen unterstreichen die Notwendigkeit, Ineffizienzen zu reduzieren und manuelle Übertragungsfehler zu minimieren.

Im Folgenden werden aktuelle wissenschaftliche Arbeiten vorgestellt sowie praktische Lösungen hinsichtlich ihrer methodischen Ansätze und Limitierungen evaluiert.

3.1 Wissenschaftliche Arbeiten

Aktuelle Arbeiten im Bereich Design-to-Code untersuchen verschiedene Methodiken, um die Lücke zwischen Design-Prototypen und Quellcode zu schließen. Das Spektrum reicht von Modellen der Künstlichen Intelligenz (KI) bis hin zu deterministischen Methoden.

Ein Großteil der aktuellen Literatur konzentriert sich auf die Übersetzung vollständiger UI-Layouts mittels maschinellem Lernen. Chen u. a. schlugen beispielsweise ein Framework „DesignCoder“ vor. Es nutzt Multimodal Large Language Models, um komplexe UI-Hierarchien vorherzusagen und React Native-Code zu generieren. „DesignCoder“ enthält zudem einen Selbstkorrektur-Mechanismus. Dieser nutzt Metadaten der Design-Datei zur Identifizierung und Behebung von Fehlern im Quellcode [19].

Ähnlich demonstriert das Framework „Figma2Code“ von Zhu u. a. die Realisierbarkeit einer direkten Übersetzung von Figma-Metadaten in Frameworks wie Jetpack Compose. Figma2Code nutzt einen hybriden Ansatz. Es kombiniert Bildklassifizierung durch Deep Learning mit dem Parsen von Metadaten. So werden komplette UI-Layouts erzeugt [20].

KI-gesteuerte Ansätze sind für das schnelle Prototyping ganzer Screendesigns sehr leistungsstark. Dennoch weisen sie für den Einsatz in Unternehmen erhebliche Nachteile auf. Die Generierung vollständiger Benutzeroberflächen führt oft zu starren Strukturen der UI-Komponenten und erfordert eine rechenintensive, externe Infrastruktur. Zudem kann eine mangelnde Präzision bei der Erkennung von Komponententypen zu Code fehlerhaftem Code führen. Solcher Quellcode erfüllt oft nicht die Entwicklungsanforderungen.

Für eine absolute Zuverlässigkeit sind automatisierte und deterministische Prozesse notwendig. Oscarsson demonstrierte dies erfolgreich durch die Entwicklung eines Figma-Plugins. Das Werkzeug extrahiert Variablen und Icons eines Designsystems. Ein

in der Amazon-Web-Services-Cloud gehostetes Backend übersetzt diese Ressourcen in TypeScript- und React-Komponenten. Das Werkzeug erstellt anschließend automatisch Merge Requests in GitLab. Diese Arbeit bestätigt, dass die Automatisierung der Token-Extraktion den manuellen Aufwand und Unklarheiten im Workflow signifikant reduziert [21]. Die Architektur ist jedoch explizit auf die Web-Entwicklung zugeschnitten.

3.2 Praktische Lösungen

Die Automatisierung der Überführung von Design-Ressourcen hat in der Web-Entwicklung bereits einen hohen Reifegrad erreicht. Es mangelt jedoch an spezialisierten Lösungen für Compose-Anwendungen. Nachfolgend werden existierende Ansätze bewertet, um die technologische Lücke präzise zu definieren. Dieser Abschnitt behandelt ausschließlich Werkzeuge mit Fokus auf die Automatisierung des Figma-Exports. Die Generierung kompletter Screendesigns wird hier nicht betrachtet.

3.2.1 Figma-Plugins

Auf dem Figma-Marktplatz findet sich eine Vielzahl von Community-Plugins, die den Export von Design-Entscheidungen unterstützen. Technisch ist hierbei zu beachten, dass Figma-Plugins nicht direkt mit dem Dateisystem des Nutzers interagieren können, mit Ausnahme des Downloads von Dateien. Ein verbreiteter Workflow besteht daher darin, Design-Tokens in ein Intermediärformat wie JSON zu exportieren.

Das am weitesten verbreitete Werkzeug ist dabei Tokens Studio for Figma (ehemals Figma Tokens), welches derzeit von über 300.000 Nutzern eingesetzt wird [22]. Das Plugin ermöglicht sowohl den Import als auch den Export von Design-Tokens und weiteren Ressourcen. Es unterstützt fortgeschrittene Konzepte wie Multi-Brand-Theming, wobei diese Funktionen den Nutzern der kostenpflichtigen Pro-Version vorbehalten sind. Da die Daten jedoch im JSON-Format exportiert werden, ist eine nachgelagerte Verarbeitung zwingend erforderlich. Für Android-Projekte bedeutet dies in der Praxis entweder den Einsatz zusätzlicher Drittanbieter-Werkzeuge oder die Entwicklung projektspezifischer Skripte, um die strukturelle Lücke zwischen dem generischen JSON-Format und typsicherem Kotlin-Code zu schließen.

Neben Tokens Studio existieren weitere Plugins mit ähnlichem Funktionsumfang (z. B. Design Tokens Manager). Diese unterscheiden sich primär in der Breite der unterstützten Token-Typen oder der Ausgereiftheit der Unterstützung von Multi-Brand-Theming. Allen gemein bleibt jedoch die Notwendigkeit einer externen Übersetzungsinstanz, um aus den JSON-Dateien nutzbaren Quellcode zu generieren.

Eine weitere Kategorie bilden Plugins, die Figma-Ressourcen in Quellcode transformieren. Diese sind jedoch oft auf die Web-Entwicklung fokussiert. Ein Beispiel für die Extraktion von Icons ist das Plugin Icons Code, welches den Export in das XML-Format ermöglicht, das für Android und Compose Multiplatform geeignet ist. Auch hier bleibt der Prozess jedoch manuell und ist nicht in einen automatisierten Build-Ablauf integriert.

3.2.2 Style Dictionary als Transformationswerkzeug

Ein wesentliches Bindeglied in vielen Designsystem-Workflows stellt das von Amazon entwickelte Open-Source-Projekt Style Dictionary dar [23]. Es agiert als zentrale Übersetzungsinstanz, um Design-Ressourcen plattformunabhängig zu verwalten und in verschiedene Formate für das Web, iOS oder Android zu transformieren. Dabei werden aktuelle Spezifikationen wie die der DTCG sowie das Referenzieren von Werten unterstützt [24].

Trotz dieser Flexibilität beschränkt sich die automatische Generierung für die Android-Plattform zumeist auf XML-Ressourcen oder einfache Objekt-Instanzen in Kotlin, die nicht für Compose-Theming-Strukturen optimiert sind. Zudem gestaltet sich die Integration in Build-Prozesse als komplex, da Style Dictionary validiertes JSON als Eingabe benötigt, was meist den Einsatz eines wie oben geschriebenen Figma-Plugins für den Export voraussetzt.

3.2.3 FigmaExport-Utility

Die im Funktionsumfang am ehesten vergleichbare Lösung ist das Open-Source-Projekt *FigmaExport*. Dabei handelt es sich um ein CLI-Werkzeug, das darauf spezialisiert ist, Farben, Typografie, Icons und Bilder direkt von Figma in Xcode- oder Android-Studio-Projekte zu exportieren. Es unterstützt unter anderem den dunkel Modus, SwiftUI und Jetpack Compose [25].

Das Werkzeug nutzt YAML-Dateien zur Konfiguration. Bei Ausführung eines Befehls werden die Ressourcen direkt in die entsprechenden Projektverzeichnisse geladen. Trotz dieses hohen Automatisierungsgrades wurde das Tool im Rahmen einer Evaluierung durch die Appsfactory als nicht ausreichend eingestuft. Die Gründe hierfür liegen in folgenden:

- Das Werkzeug unterstützt keine Multi-Brand-Strukturen über Figma-Collections.
- Die Ressourcen werden primär über Figma-Stile bezogen. Das Abrufen von Design-Tokens über Variablen befindet sich in einem instabilen Beta-Stadium.
- Es mangelt an der Unterstützung von CMP.
- Als reines CLI-Werkzeug lässt es sich nicht nativ in das Gradle-Build-System inte-

grieren. Entwickler müssten das Tool separat installieren und manuell konfigurieren.

- Die Generierung von Tokens für Abstände ist nicht vorgesehen.

3.3 Fazit

Die Analyse der aktuellen Stand der Forschung und Technik zeigt ein Spannungsfeld zwischen Ansätzen. Auf der einen Seite stehen KI-basierte Modelle zur automatisierten Generierung vollständiger Benutzeroberflächen. Diese eignen sich für schnelles Prototyping. Sie weisen jedoch Defizite bei der strukturellen Präzision und Verlässlichkeit auf. Auf der anderen Seite stehen deterministische Methoden, die jedoch meist auf Web-Technologien fokussiert.

Zusammenfassend lässt sich festhalten, dass gegenwärtig kein integriertes Werkzeug existiert, welches den gesamten Prozess von der Figma-API-Abfrage bis zur Generierung des typsicheren Compose-Quellcodes für Multi-Brand-Projekte automatisiert. Bestehende Lösungen wie Figma Export decken lediglich Teilmengen der erforderlichen Funktionalität ab. Diese technologische Lücke begründet die Notwendigkeit für die Entwicklung eines Werkzeugs.

Das in dieser Arbeit entwickelte Werkzeug konzentriert sich auf die Extraktion von Design-Tokens und Icons. Dies garantiert eine exakte Übereinstimmung zwischen Design und Code ohne das Risiko von KI-Halluzinationen. Das Werkzeug ist explizit für die UI-Entwicklung mit Compose geeignet. Die Integration kann über Gradle-Plugin erfolgen. Dies ermöglicht eine lokale, reproduzierbare Ausführung innerhalb des Build-Systems. Das Werkzeug schließt die Lücke bei der effizienten Abbildung komplexer Markenidentitäten innerhalb einer gemeinsamen Codebasis.

4 Entwurf

In diesem Kapitel werden zuerst die Anforderungen an das Werkzeug aufgelistet, anschließend die Konzeption und der funktionale Aufbau beschrieben. Das Werkzeug wird intern als *Figma Resource Generator (FigmaResGen)* bezeichnet und findet unter diesem Namen auch im weiteren Verlauf der Arbeit Anwendung.

4.1 Anforderungen

Basierend auf den Bedürfnissen der Appsfactory sowie den identifizierten Defiziten bestehender Alternativlösungen wurden funktionale und nicht-funktionale Anforderungen an das Werkzeug festgelegt. Diese werden nach Balzert formuliert [26].

Funktionale Anforderungen

- Das Werkzeug muss dem Entwickler die Möglichkeit bieten, Tokens für Farben, Abstände, Typografie zu generieren.
- Das Werkzeug muss dem Entwickler die Möglichkeit bieten, Icons für ein Zielprojekt bereitzustellen.
- Falls mehrere Modi vorliegen, muss das Werkzeug dem Entwickler die Möglichkeit bieten, Design-Tokens für die jeweiligen Kontexte zu generieren.
- Falls das Designsystem mehrere Markenidentitäten umfasst, muss das Werkzeug dem Entwickler die Möglichkeit bieten, Design-Tokens für jede Marke zu generieren.
- Das Werkzeug muss dem Entwickler die Möglichkeit bieten, die Export-Parameter zu konfigurieren.

Nicht-funktionale Anforderungen

- Das Werkzeug muss eine Kompatibilität zu nativen Android- und CMP-UI-Entwicklung aufweisen.
- Das Werkzeug muss fähig sein, im CI/CD-Pipeline integriert zu werden.
- Das Werkzeug muss fähig sein, auf dem lokalen Rechner des Entwicklers ausgeführt zu werden.

4.2 Architektur

Die Architektur dieses Werkzeugs wurde nach „Clean Architecture“ konzipiert. Diese zeichnet sich durch eine Trennung von Belangen in Schichten aus. Dabei sind Abhängigkeiten

gemäß der „Dependency Rule“ ausschließlich nach innen gerichtet. Dadurch bleibt der funktionale Kern unabhängig von externen Frameworks oder Benutzeroberflächen und ist hochgradig testbar [27]. Da diese Architektur innerhalb der Appsfactory als Standard für Softwareprojekte etabliert ist, wurde entschieden, das Werkzeug nach derselben Architektur zu gestalten. Dies gewährleistet eine hohe Wartbarkeit und ermöglicht es anderen Entwicklern innerhalb des Unternehmens, den Quellcode schnell zu verstehen und zu erweitern.

Aufgrund der Anforderungen an die Integration in CI/CD-Pipelines sowie der lokalen Nutzung wurde FigmaResGen als modularer Kern konzipiert. Dieser kann entweder als eigenständige Applikation lokal ausgeführt oder als Gradle-Plugin direkt in den Build-Prozess eines Anwendungsprojekts integriert werden. Die funktionale Trennung des Werkzeugs orientiert sich an den verschiedenen Endpunkten der Figma API und wird in zwei Hauptprozesse unterteilt: die Generierung von Design-Tokens sowie den Export von Icons.

4.3 Konfigurations- und Interaktionsmöglichkeiten

Bevor die Prozesse im Detail erläutert werden, müssen die zur Verfügung stehenden Interaktionsmöglichkeiten definiert werden.

Soll das Werkzeug lokal genutzt werden, ist der Bezug des Quellcodes von FigmaResGen aus dem internen Repository des Unternehmens erforderlich. Die im Aktivitätsdiagramm spezifizierte Aktivität *Konfiguration* illustriert (Abb. 6), dass in diesem Szenario die Erstellung einer YAML-Konfigurationsdatei erforderlich ist. In dieser Datei werden die notwendigen Parameter für den Export festgelegt.

Wird das Werkzeug hingegen als Gradle-Plugin eingesetzt, erfolgt die Integration über eine vordefinierte Einbindung innerhalb der internen Projektvorlage der Appsfactory GmbH. Dadurch steht den Entwicklern eine Konfigurationsbasis zur Verfügung, die lediglich an projektspezifische Anforderungen angepasst werden muss. Das Plugin wird dabei in dem Modul angewendet, welches `AppTheme`-bezogene Objekte und Klassen sowie allgemeine Composables beinhaltet. Die Konfiguration erfolgt über eine `figmaResGen`-Gradle-Extension. Das Diagramm (Abb. 6) verdeutlicht zudem, dass das Plugin alternativ eine YAML-basierte Konfiguration unterstützt.

Unabhängig von der gewählten Interaktionsform definiert die Konfiguration zentrale Parameter wie den `file_key` für den Zugriff auf das Figma-Datei sowie die spezifischen Zielverzeichnisse für den generierten Code.

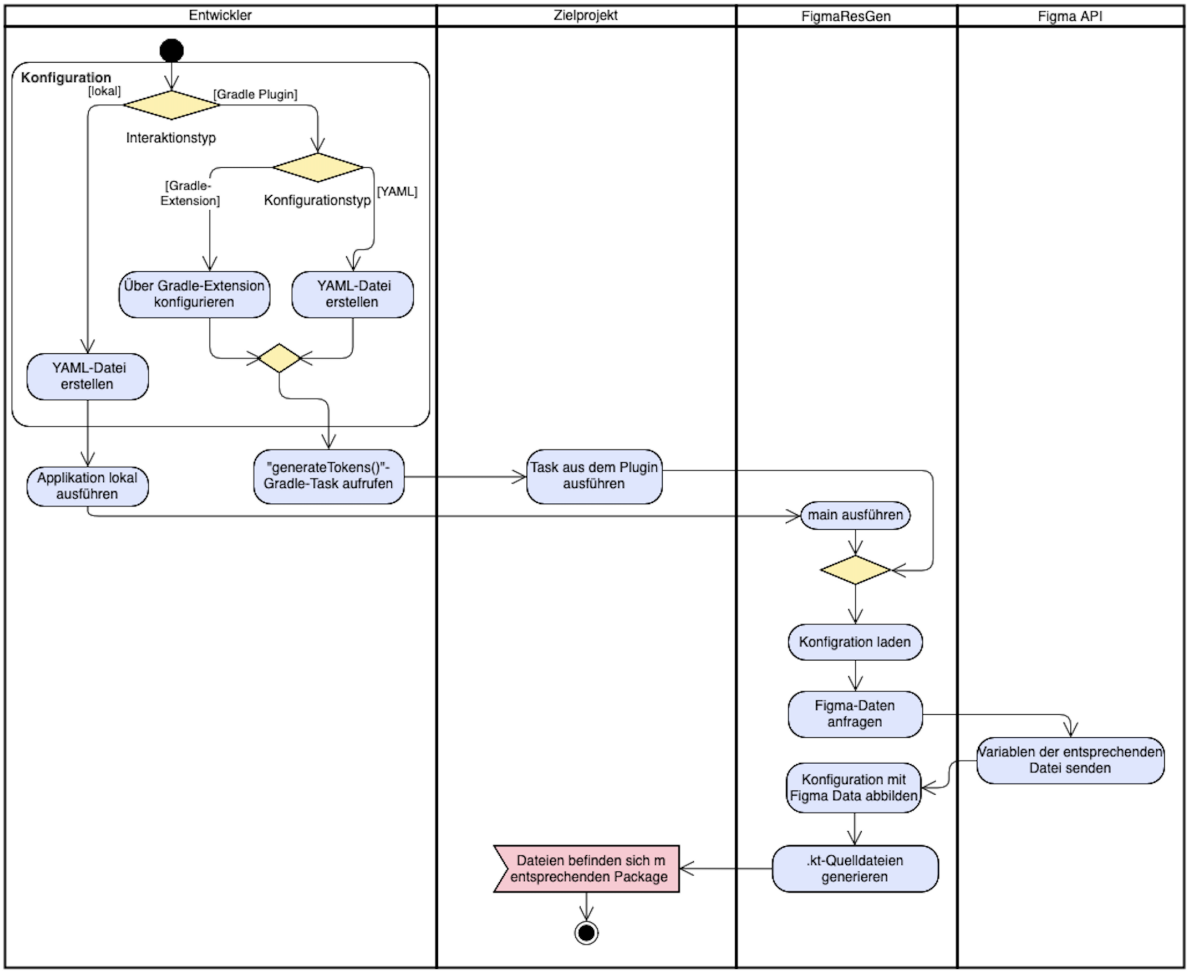


Abbildung 6: Aktivitätsdiagramm zur Token-Generierung

4.4 Prozessablauf der Token-Generierung

Die Token-Generierung umfasst die Extraktion und Transformation von Tokens für Typografie, Farben und Abstände, was im Diagramm (Abb. 6) zusammenfassend dargestellt ist. Sobald die Kernlogik des Werkzeugs entweder über die `main()`-Funktion oder die `generateTokens()`-Gradle-Aufgabe initiiert wird, parst das Werkzeug die vorliegenden Konfigurationsdaten und leitet den Datenabruf über die Figma Variables API ein. Die vom Server gelieferten Daten werden anschließend gefiltert und gegen die spezifische Nutzerkonfiguration gemappt. Dadurch werden ausschließlich die für das Projekt relevanten Design-Tokens verarbeitet.

Diese gefilterten Objekte werden im nächsten Schritt in ein internes Datenmodell überführt. Mithilfe der im Kapitel 2.5 erwähnten KotlinPoet-Bibliothek erfolgt schließlich die Transformation in Kotlin-Datenklassen. Hierbei berücksichtigt der Generator sowohl die in Figma definierte Gruppenstruktur als auch die Hierarchie der Design-System-Strukturen innerhalb der firmeninternen Projektvorlage. Dies stellt eine konsistente Einbindung der

generierten Ressourcen in das `AppTheme` sicher.

4.5 Prozessablauf des Icon-Exports

Im Gegensatz zur Token-Generierung folgt der Export von Icons einer eigenständigen Logik. Der Prozess basiert auf einer Kombination von FigmaResGen und dem Valkyrie-Gradle-Plugin. Das Diagramm illustriert diesen Prozess (Abb. 7). Basierend auf der Konfiguration identifiziert das Werkzeug die zu exportierenden Icons und fordert über die Figma Image API die entsprechenden SVG-Renderings an.

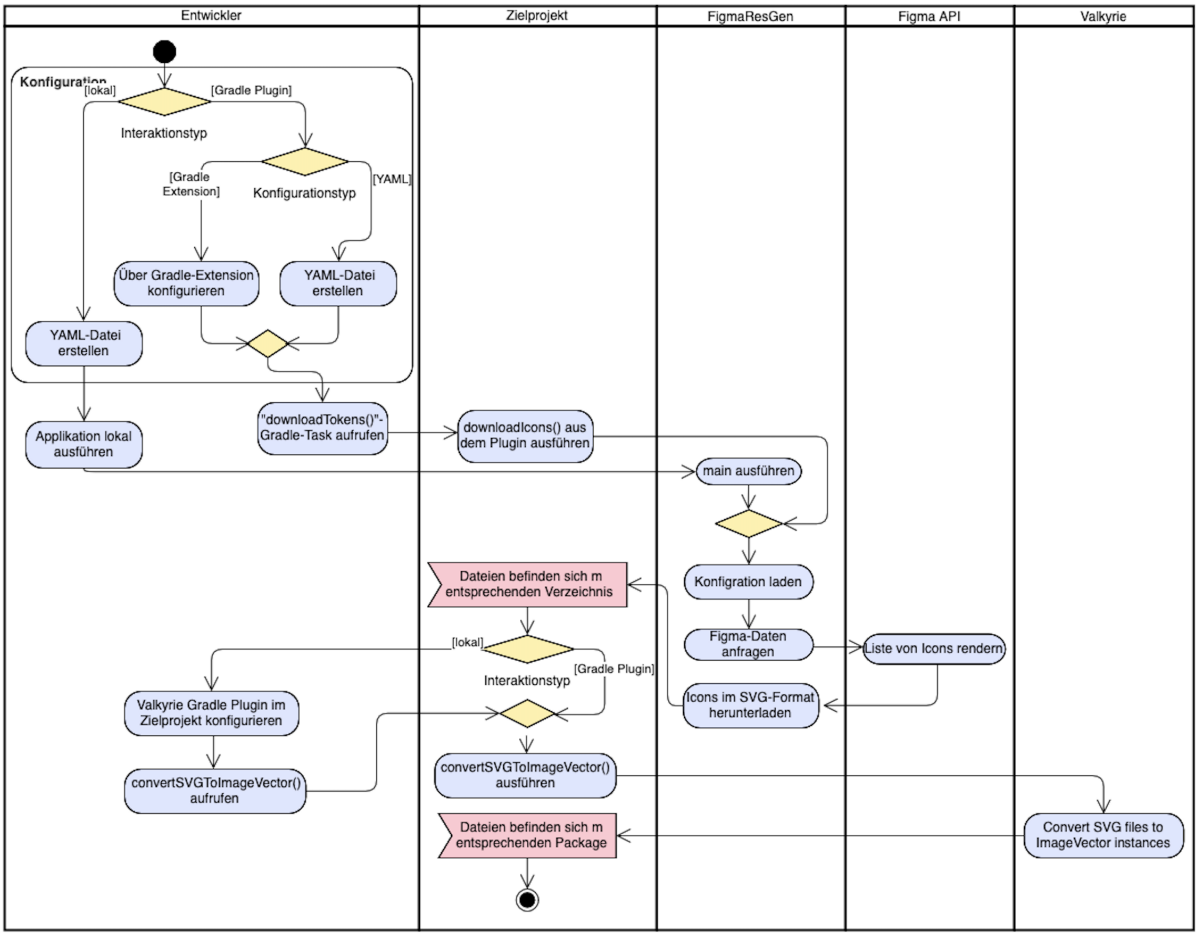


Abbildung 7: Aktivitätsdiagramm zum Prozessablauf des Icon-Exports

Diese Dateien werden automatisiert heruntergeladen und in einem definierten temporären Verzeichnis gespeichert. Wie im Kapitel 2.3.3 erwähnt wurde, ist eine Konvertierung von SVG in das `ImageVector`-Format notwendig. Innerhalb des Gradle-Ablauf sorgt die `generateIcons()`-Aufgabe dafür, nach dem erfolgreichen Download automatisch die `convertSVGToImageVector`-Aufgabe des Valkyrie-Plugins auszuführen.

Für Szenarien ohne Gradle-Anbindung sieht der Entwurf vor, dass der Nutzer zunächst die `main()`-Funktion für den Download-Prozess ausführt und die anschließende Konvertierung

separat über die im Zielprojekt konfigurierte Valkyrie-Gradle-Aufgabe initiiert.

Für Szenarien ohne Gradle-Anbindung sieht der Entwurf einen zweistufigen Prozess vor. Der Nutzer führt zunächst die `main()`-Funktion für den Download-Prozess aus. Die anschließende Konvertierung wird über die im Zielprojekt konfigurierte Valkyrie-Gradle-Aufgabe initiiert.

5 Implementierung

In diesem Kapitel wird die Implementierung des Werkzeugs FigmaResGen detailliert dargelegt. Die Darstellung folgt solcher Struktur: Zunächst wird die Aufteilung des Projekts in Modulen erläutert, gefolgt von einem Überblick über die funktionalen Komponenten und deren Abhängigkeiten. Im weiteren Verlauf werden die zentralen Schwerpunkte der Entwicklung vertieft behandelt. Dies umfasst die Spezifikation der Konfigurationsparameter, welche die Grundlage für die Steuerung der Generierungsprozesse bilden. Darauf aufbauend wird die interne Arbeitsweise der Kernmodule beschrieben.

5.1 Projektstruktur

Die Struktur des Quellcodes von FigmaResGen ist als Gradle Multi-Modul-Projekt realisiert. Diese Entscheidung basiert auf den Prinzipien der Trennung von Belangen sowie der Wiederverwendbarkeit einzelner Systemkomponenten. Abbildung 8 zeigt das Komponentendiagramm des Werkzeugs, wobei jede logische Komponente exakt durch ein entsprechendes Modul im Quellcode repräsentiert wird.

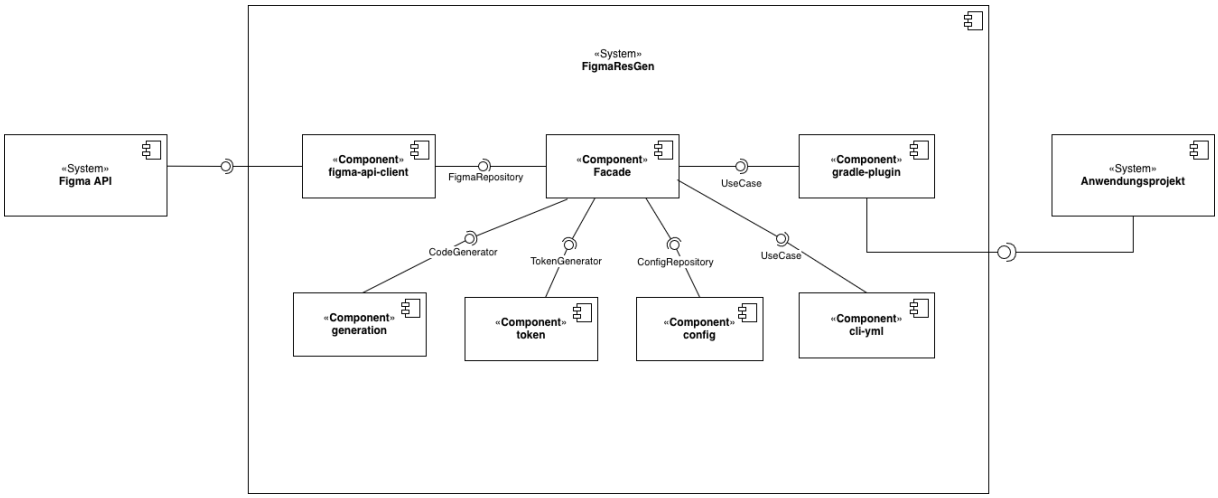


Abbildung 8: Komponentendiagramm

Die Modularisierung des Projekts ermöglicht eine klare Trennung zwischen Infrastruktur, Domänenlogik und den verschiedenen Schnittstellen.

Die Struktur gliedert sich in folgende Module:

Das Modul `gradle-plugin` stellt die Schnittstelle zum Android- oder CMP-Anwendungsprojekt dar. Es enthält die Klasse `FigmaResGenPlugin`, welche das Werkzeug in den Build-System integriert. Die Konfiguration erfolgt über eine `FigmaResGenExtension`-Gradle-Extension. Die Ausführung der Kernfunktionen erfolgt

über `GenerateTokensTask`, `DownloadIconsTask` und `YmlGenerateTokensTask` Gradle-Aufgaben, welche intern auf die Use-Cases des `facade`-Moduls zurückgreifen.

Das `cli-yml`-Modul dient als alternativer Eintrittspunkt für die lokale Nutzung des Werkzeugs. Es beinhaltet die Logik für das Parsen und Validierung der Konfigurationsdaten aus einer YAML-Datei. Hierfür wird die Bibliothek `kaml` zur Deserialisierung verwendet. Die zentrale `main()`-Funktion innerhalb dieses Moduls steuert den Programmablauf und initiiert die Ausführung der Logik ebenfalls über die im `facade`-Modul definierten Use-Cases.

Das `facade`-Modul fungiert als Orchestrierebene der gesamten Werkzeuglogik. Es verwaltet die Abhängigkeiten zwischen den Modulen und stellt sicher, dass die über die Konfiguration definierten Anforderungen in eine koordinierte Abfolge von Prozessschritten überführt werden.

Das `config`-Modul bildet die zentrale Abstraktionsschicht für alle Konfigurationsdaten. Mithilfe des `ConfigDataSource`-Interfaces werden die unterschiedlichen Eingabequellen (Gradle oder YAML) in ein einheitliches Domänenmodell (`Config`) überführt. Das Repository-Pattern (`ConfigRepository`) dient hierbei als definierte Schnittstelle für den Abruf dieser Daten.

Das Modul `figma-client` verantwortet die Kommunikation mit der Figma API. Technisch basiert die Umsetzung auf dem Ktor-HTTP-Client.

Das `token`-Modul beinhaltet die zentrale Geschäftslogik des Werkzeugs und verantwortet die Transformation von Figma-Rohdaten in ein für die Generierung passendes Format.

Das Modul `generation` transformiert das von `token`-Modul erstellten Objekt mittels der Bibliothek *KotlinPoet* in Quellcode. Der Prozess folgt einer Zwei-Phasen-Generierung. Zuerst werden die Datenklassen-Hierarchien erstellt, welche die Struktur des Designsystems abbilden. Dann erfolgt die Generierung der Instanzen pro Marke oder Modus.

Die Module `model:figma`, `model:token`, `model:task` enthalten ausschließlich reine Datenklassen ohne Logik, die als gemeinsame Sprache zwischen den Schichten dienen.

Das Modul `utils` stellt gemeinsame Hilfsfunktionen für String-Operationen, Logging-Adapter für Gradle und CLI sowie allgemeine Erweiterungsfunktionen bereit.

5.2 Konfiguration

Die Konfigurationsparameter des FigmaResGen wurden in einem iterativen Prozess evaluiert und optimiert. Das Ziel bestand darin, die Anzahl der notwendigen Einstellungen auf ein Minimum zu reduzieren, um die Integrationshürden für die Entwickler so gering

wie möglich zu halten. Dabei wurde konsequent das Entwurfsprinzip „Convention over Configuration“ angewandt.

Dies bedeutet, dass das Werkzeug von sinnvollen Standardwerten ausgeht, sofern der Anwender keine expliziten Abweichungen definiert. Beispielsweise präsumiert das Werkzeug, dass alle in einer Collection vorhandenen Design-Tokens, Modi und Marken generiert werden sollen. Erst wenn eine Einschränkung auf spezifische Markenidentitäten, Modi oder Token-Gruppen erforderlich ist, muss dies manuell konfiguriert werden.

Ein Minimalbeispiel für eine funktionale Konfiguration umfasst die Identifikation der Figma-Datei, den API-Zugriffsschlüssel sowie die Definition der zu generierenden Token-Typen ist im Listing 14 zu sehen.

```
figmaResGen {
    // Eindeutige ID der Figma-Datei
    figmaFileKey = "your_figma_file_key"

    // Authentifizierungsschluessel fuer die Figma API
    figmaApiKey = "your_figma_api_key"

    packageInfo {
        // Basis-Package-Pfad, meist der Namespace der Anwendung
        base = android.namespace
    }

    tokens {
        // Registrierung eines Token-Typs
        register("color") {
            className = "Colors"                    // Ziel-Klassenname
            figmaCollectionName = "Design Tokens"   // Name von Collection in Figma
            tokenType = "color"                     // Typ des Design-Tokens
        }
    }

    iconConfig {
        // URL zum Figma-Frame, der die Icons enthaelt
        frameUrl = "url_to_frame_with_icons"
    }
}
```

Listing 14: FigmaResGen-Gradle-Extension

Für White-Label-Projekte erlaubt die Extension eine Definition der Marken-Mappings. Hierbei wird jeder Marke ein lokaler Source-Set-Name sowie der entsprechende Markenname aus der Figma-Collection zugewiesen. Das Beispiel dafür ist im Listing 15 zu sehen.

5.3 Das figma-client-Modul

Das Modul `figma-client` bildet die Netzwerk-Infrastrukturschicht des Werkzeugs. Seine primäre Aufgabe liegt in der Kapselung sämtlicher Interaktionen mit der Figma API.

Somit wird die restliche Geschäftslogik von der Komplexität der Remote-Schnittstelle entkoppelt. Dies betrifft insbesondere die Handhabung polymorpher JSON-Strukturen sowie die Koordination verschiedener Endpunkte.

```
brands {
    register("brandA") {
        packageName.set("brandA")
        figmaBrandName.set("Brand A")
    }
    register("brandB") {
        packageName.set("brandB")
        figmaBrandName.set("Brand B")
    }
}
```

Listing 15: Konfiguration von mehrere Marken in FigmaResGen-Gradle-Extension

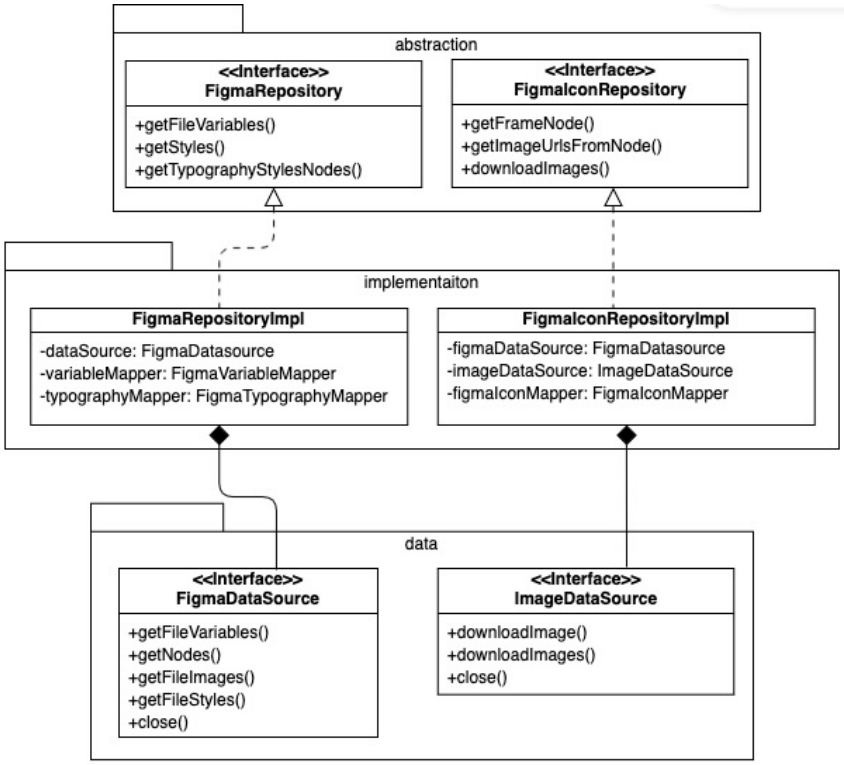


Abbildung 9: Klassendiagramm zum figma-client-Modul

Um eine saubere Trennung zwischen Infrastruktur und Domänenlogik zu erreichen, wird das Repository Pattern eingesetzt. Dies ermöglicht es, das Netzwerk-Spezifika auszutauschen oder zu simulieren, ohne die Kernlogik des Werkzeugs zu beeinflussen. Nach außen hin sind für das `facade`-Modul lediglich abstrakte Schnittstellen sichtbar. Eine davon ist `FigmaRepository`, die den Abruf von Design-Tokens und Typografie-Informationen über die im Kapitel 2.2 aufgelisteten Endpunkte verwaltet. Die andere ist `FigmaIconRepository`, die den dreistufigen Prozess des Icon-Exports steuert. Dies umfasst die Lokalisierung des Ziel-Containers (Frame oder Sektion) mit Icons in Figma

anhand der URL. Daraufhin ruft das System die temporären Download-Links für diese Icons über die Figma-API ab. Anschließend werden die Ressourcen in das lokale Dateisystem des Zielprojekts heruntergeladen.

5.3.1 Serializer für JSON-Objekt

Eine technologische Herausforderung stellt die Verarbeitung der Variablenwerte innerhalb der Figma Variables API dar. Im Feld `valuesByMode` einer API-Antwort können Werte mit fundamental unterschiedlichen Datentypen auftreten: hexadezimale Farbmodelle (als Objekt `{r, g, b, a}`), Fließkommazahlen für Abstände, Zeichenketten, boolesche Werte oder Referenzen auf andere Variablen.

Da herkömmliche JSON-Parser bei derartigen heterogenen Strukturen an funktionale Grenzen stoßen, wurde ein benutzerdefinierter `ApiModeValueSerializer` auf Basis von `kotlinx.serialization` implementiert. Dieser arbeitet auf der Ebene der Rohdaten in zwei Schritten. Zuerst wid das JSON-Element vor der Transformation analysiert. Danach anhand struktureller Merkmale (z. B. das Vorhandensein des Schlüssels `type: "VARIABLE_ALIAS"`) wird zur Laufzeit entschieden, welche Kotlin-Datenklasse instanziiert werden soll.

Dieser Prozess transformiert eine strukturell uneinheitliche API-Antwort in eine typsichere Sealed Class (`ApiModeValue`).

5.3.2 Performance-Optimierung beim Icons-Download

Beim Export von Icons stößt das Werkzeug auf ein Performance-Problem: Ein Designsystem kann hunderte von Icons enthalten. Ein sequenzieller Download dieser Dateien würde die Laufzeit des Plugins unverhältnismäßig verlängern und die Effizienz des Build-Prozesses mindern.

Zur Lösung dieser Problematik nutzt die Komponente `KtorImageDataSource` das Koroutinen-Modell von Kotlin. Dabei werden alle Downloads als asynchrone Aufgaben parallel initiiert. Um eine Überlastung der lokalen Systemressourcen zu vermeiden, wird ein `Semaphore` mit einer Kapazität von zehn Permits eingesetzt. Dies limitiert die Anzahl der gleichzeitig aktiven Netzwerkverbindungen, während weitere Anfragen in einer Warteschlange verwaltet werden, bis Ressourcen frei werden.

5.3.3 Mapping und Datenextraktion

Den Abschluss der Verarbeitung bildet die Mapping-Schicht. Hier werden die technischen Api-Modelle in Domänen-Modelle transformiert. Hierbei erfüllt der Mapper zwei wesent-

liche Funktionen:

- Basierend auf der Benutzerkonfiguration werden Daten bereits hier reduziert. Beispielsweise werden Variablen verworfen, die in Figma als „Hidden from Publishing“ markiert sind. Darüber hinaus werden nur spezifisch definierte Collections oder Gruppen für die weitere Verarbeitung zugelassen.
- Der Mapper zerlegt die von Figma als flache Zeichenketten gelieferten Pfade (z. B. `colors/buttons/primary`) in eine Liste von Zeichenketten. Diese Segmentierung ist die notwendige Voraussetzung für das Modul `token`, um später eine korrespondierende verschachtelte Klassenstruktur im Programmcode zu erzeugen.

Letztendlich liefert die FigmaRepository ein Objekt vom Typ der `FigmaData`-Datenklasse für die Generierung der Tokens über die Figma Variables API:

```
data class FigmaData(
    val variables: List<FigmaVariable>,
    val collections: List<FigmaVariableCollection>,
)
```

Listing 16: FigmaData-Datenklasse

5.4 Mapping von Figma-API-Daten und Konfiguration

Das Modul `token` bildet das funktionale Zentrum des FigmaResGen. In dieser Schicht ist die primäre Geschäftslogik gekapselt. Die Daten des Figma-Clients und die Parameter der Nutzerkonfiguration werden in eine spezifikationsunabhängige Repräsentation transformiert. Der Transformationsprozess bereitet die Daten der Figma-API auf. Dabei werden komplexe Referenzketten von Variablen aufgelöst. Abschließend werden die Ergebnisse in einer aufbereiteten Form für die Codegenerierung bereitgestellt.

5.4.1 Datenmodell und Repräsentation

Bevor der Transformationsprozess im Detail betrachtet wird, ist ein Verständnis des zugrunde liegenden Zielmodells, der `TokenClass`, essenziell. Diese Struktur bildet das Rückgrat für die spätere Erzeugung von Kotlin-Quellcode.

Die `TokenClass`-Klasse ist als Baum organisiert, der aus rekursiven `TokenTree`-Knoten besteht. Diese Knoten bilden die Namensräume innerhalb des Designsystems ab. An den Endknoten dieser Struktur befinden sich die eigentlichen `Token`-Instanzen, welche ihre Werte differenziert nach Marken und Modi speichern. Diese hierarchische Organisation wurde gewählt, um die im Design oft verwendeten Schachtelungen (z. B.

`colors/background/primary`) eins zu eins in verschachtelte Datenklassen im Programmcode überführen zu können.

```
data class TokenClass(
    val modelClassName: String,
    val tokensClassName: String,
    val tokenTrees: List<TokenTree>,
    val tokens: List<Token> = emptyList(),
)

data class TokenTree(
    val name: String,
    val tokenTrees: List<TokenTree> = emptyList(),
    val tokens: List<Token>,
)

data class Token(
    val name: String,
    val pathSegments: List<String>,
    val valuesByBrand: Map<String, TokenValue>,
)

data class TokenValue(
    val valuesByMode: Map<String, ConcreteValue>,
)
```

Listing 17: Struktur von TokenClass

Die kleinste atomare Einheit ist die `ConcreteValue`-Datenklasse. Dabei handelt es sich um ein Sealed Interface, welches die aufgelösten Endwerte repräsentiert. Es wird zwischen vier Varianten unterschieden:

- `Color`: Speichert hexadezimale ARGB-Farbwerte.
- `Dimension`: Repräsentiert numerische Werte inklusive ihrer Maßeinheit (z. B. DP für Abstände oder SP für Textgrößen).
- `Typography`: Bündelt Typografie-Eigenschaften.
- `Unresolved`: Markiert Zustände, in denen eine Auflösung aufgrund technischer Fehler oder fehlender Daten nicht möglich war.

Das Werkzeug wirft im Fehlerfall keine Ausnahme, sondern bettet den Typ `Unresolved` in den Wertbaum ein. Dies ermöglicht es nachgelagerten Prozessen flexibel auf Teilfehler zu reagieren. Beispielsweise durch die Generierung von Platzhaltern oder detaillierten Fehlermeldungen im Zielprojekt.

5.4.2 Der Mappingsprozess

Die Koordination der Transformation erfolgt über das Interface `TokenClassGenerator`. Dessen Implementierung, die `TokenClassGeneratorImpl`, prozessiert die Daten in folgender Reihenfolge.

```
sealed interface ConcreteValue {
    data class Color(val argb: UInt) : ConcreteValue {
        val hex: String
            get() = argb.toString(16).padStart(8, '0').uppercase()
    }

    data class Dimension(
        val value: Float,
        val unit: DimensionUnit
    ) : ConcreteValue

    data class Typography(
        val fontFamily: String,
        val fontWeight: Int,
        val fontSize: Dimension,
        val lineHeight: Dimension? = null,
    ) : ConcreteValue

    data class Unresolved(
        val unresolvedPath: String,
        val error: ResolutionError
    ) : ConcreteValue
}
```

Listing 18: Struktur der ConcreteValue-Datenklasse

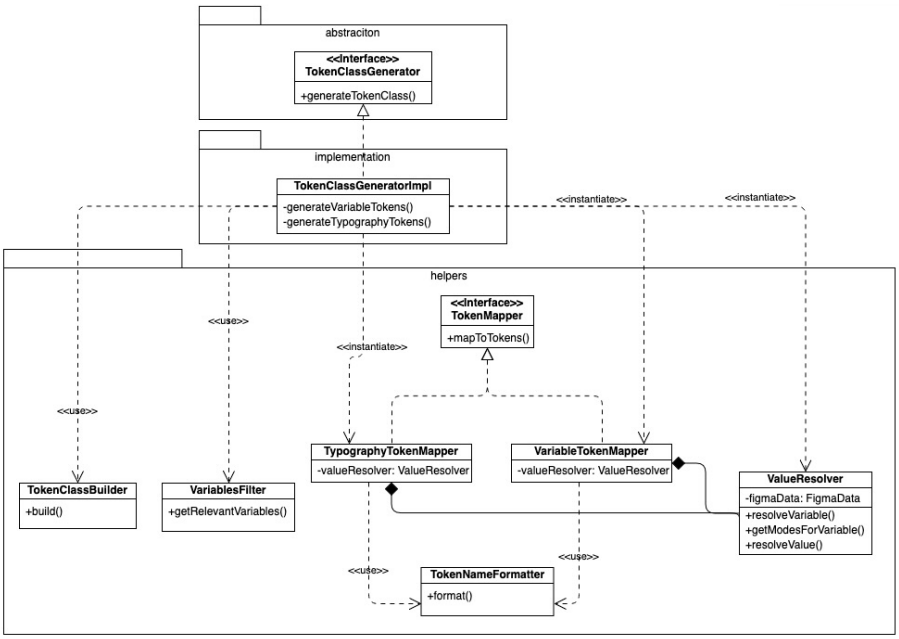


Abbildung 10: Klassendiagramm zum token-Modul

Filterung Zuerst validiert der Generator die Eingabedaten. Basierend auf der Benutzerkonfiguration wird innerhalb der `FigmaData`-Instanzen nach der entsprechenden Variablen-Collection gesucht. Nach erfolgreicher Identifikation der Collection erfolgt eine Filterung der enthaltenen Variablen. Hierbei werden nur jene Figma-Variablen berücksichtigt, die:

- der spezifizierten Collection angehören,
- sich innerhalb der definierten Namensräume befinden,
- und dem angeforderten `TokenType` (z. B. nur Farben) entsprechen.

Auswahl des Verarbeitungsprozesses Nach der Filterung verzweigt sich der Prozess in zwei spezialisierte Verarbeitungswege:

- **Variablen-Mapping**. Dieser verarbeitet Farben und numerische Abstände. Im `VariableTokenMapper` wird hierbei zwischen Single-Brand- und Multi-Brand-Konfigurationen unterschieden. Während bei Single-Brand-Projekten alle Modi einer Variable aufgelöst werden, identifiziert das System im Multi-Brand-Szenario für jede definierte Marke den spezifischen Modus und verfolgt die entsprechenden Werte über Marken-Grenzen hinweg.
- **Typografie-Mapping**. Dieser nutzt Figma-Nodes als Ausgangspunkt. Hierbei werden gezielt verbundene Variablen extrahiert. Das bedeutet, dass Eigenschaften wie Schriftgröße oder Zeilenhöhe, die im Design-Tool mit Variablen verknüpft wurden, dynamisch aufgelöst werden. Sollten keine Verknüpfungen vorliegen, greift das System auf die statischen Rohwerte des Stils zurück.

Modus-Identifikation und Werteauflösung Für jeden Token muss der korrekte Wert pro Modus ermittelt werden. Hierbei kommt der `ValueResolver` zum Einsatz. Er indiziert alle Variablen und Collections, um einen schnellen Zugriff zu ermöglichen. Bei Single-Brand-Konfigurationen werden die Modi entweder explizit aus der Konfiguration übernommen oder automatisch aus der ersten verarbeiteten Variable extrahiert. Bei Multi-Brand-Konfigurationen wird für jede Marke der entsprechende Marken-Modus gesucht. Falls ein Wert ein Alias ist, wird die Referenzkette bis zur ursprünglichen Quelle verfolgt.

Iterative Alias-Auflösung Die Auflösung von Aliassen erfolgt im `ValueResolver` iterativ statt rekursiv. Dies ist eine gezielte Entwurfsentscheidung, um Stack Overflows bei tiefen Referenzketten zu vermeiden. Der Resolver folgt den Alias-Pointern Schritt für Schritt:

- Jede besuchte Variablen-ID wird protokolliert, um zirkuläre Referenzen frühzeitig zu erkennen und als `CircularReference`-Fehler zu markieren.
- Bei jedem Schritt wird geprüft, ob die Ziel-Collection den angeforderten Modus unterstützt. Falls eine Collection nur über einen einzigen Modus verfügt (der Fall bei Basis-Collections), dient dieser automatisch als Fallback.
- Die Schleife terminiert, sobald ein primitiver Wert erreicht wird, der dann in die entsprechenden `ConcreteValue` konvertiert wird.

Formatierung und Baumkonstruktion Nachdem die Werte erfolgreich aufgelöst wurden, müssen die Bezeichner für den Zielcode normalisiert werden. Der `TokenNameFormatter` transformiert die hierarchischen Pfade von Figma in gültige Kotlin-Bezeichner im Camel-Case-Format. Hierbei greift die sogenannte Trim-Logik, die nutzerdefinierte Präfixe entfernt, um unnötige Verschachtelungstiefen im Code zu vermeiden.

Im finalen Schritt überführt der `TokenClassBuilder` die flache Liste der generierten Tokens in die endgültige Baumstruktur. Er partitioniert die Tokens anhand ihrer Pfadsegmente und erstellt rekursiv die `TokenTree`-Instanzen. Durch eine abschließende alphabetische Sortierung wird ein deterministisches Ergebnis sichergestellt.

5.5 Generierung des Quellcodes

Das Modul `generation` stellt die finale Stufe von FigmaResGen dar. Seine primäre Funktion besteht in der Überführung der prozessierten `TokenClass`-Objekte in kompilierbaren Kotlin-Quellcode. Dabei werden neben dem aufgelösten `TokenClass` auch die Nutzerkonfiguration und ein optionaler `FontGenerationContext` mit Metadaten heruntergeladener Schriftarten verarbeitet.

Die Erzeugung erfolgt ohne unmittelbare Dateisystem-Interaktion durch die Bibliothek *KotlinPoet*. Dabei wird der Quellcode zunächst als abstrakter Syntaxbaum (Abstract Syntax Tree, AST) im Arbeitsspeicher modelliert. Ein AST stellt hierbei eine strukturierte, baumartige Repräsentation der Programm-Logik dar, die alle syntaktischen Elemente des Codes (wie Klassen, Eigenschaften und Parameter) definiert [28], bevor diese in eine textuelle Dateiform umgewandelt werden. Die so generierten Dateispezifikationen werden anschließend an das `facade`-Modul zurückgegeben, welches die weitere physische Persistierung auf dem Datenträger aufruft.

Die Steuerung dieses Prozesses wird durch den `CodeGeneratorImpl` orchestriert, wobei eine Trennung zwischen struktureller Definition und wertbasierter Initialisierung eingehalten wird. Zunächst erfolgt die Erzeugung der zugrunde liegenden Datenklassen-Hierarchie.

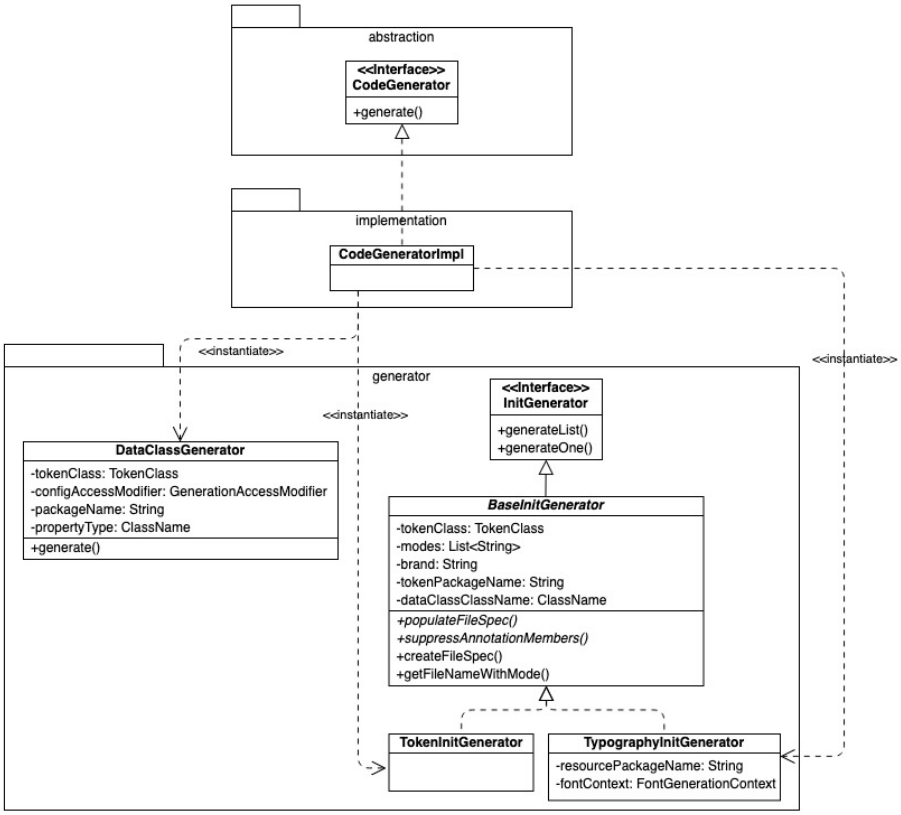


Abbildung 11: Klassendiagramm zum generation-Modul

Diese Struktur fungiert als markenunabhängiges, stabiles Fundament, welches die logische Architektur des Designsystems widerspiegelt. Im Anschluss daran werden die konkreten Instanzen generiert, in denen die spezifischen Werte für verschiedene Marken und Modi definiert sind. Dieser entkoppelte Ansatz gewährleistet, dass unterschiedliche Marken dieselbe strukturelle Basis nutzen können, während ihre visuellen Ausprägungen in isolierten Dateien verwaltet werden.

Für die Abbildung des hierarchischen `TokenTree` ist das `DataClassGenerator` verantwortlich. Dieses nutzt einen rekursiven Algorithmus, um eine verschachtelte Struktur von Datenklassen aufzubauen, welche die Gruppen-Hierarchie aus Figma präzise abbildet. Jedes Pfadsegment innerhalb des Token-Baums wird dabei systematisch in eine statisch typisierte Unterklasse überführt. Ein zentrales Element ist hierbei die automatisierte Typ-Zuweisung. Basierend auf den Metadaten der Design-Tokens entscheidet das Generator, welche spezifischen Klassen des Compose-Frameworks – wie beispielsweise `Color` für Farbwerte oder `Dp` für Abstände – im Quellcode verwendet werden. Über Konfigurationsparameter lässt sich zudem die Sichtbarkeit der generierten Klassen steuern.

Die technische Basis für die Erzeugung von Dateien mit Instanzen von Tokens bildet das gemeinsame Interface `InitGenerator`. Dieses definiert die Methoden `generateList` und `generateOne`. Die Methode `generateList` erzeugt eine Liste von `FileSpec`, wobei

die Tokens für jeden Modus in einer separaten Datei abgelegt werden. Im Gegensatz dazu bündelt `generateOne` alle Modi eines Tokens innerhalb einer einzigen Datei. Die abstrakte Klasse `BaseInitGenerator` implementiert dieses Interface. Sie deklariert die notwendigen Attribute für die abgeleiteten Klassen und stellt Basis-Implementierungen sowie allgemein verwendbare Hilfsfunktionen bereit.

Auf dieser Architektur bauen zwei spezialisierte Generatoren für Instanzen auf. Der `TokenInitGenerator` erzeugt Werte für Farben und Abstände. Die Werte werden dabei als Literale, wie beispielsweise `Color(0xFF...)` oder `16.dp`, direkt in den Quellcode geschrieben. Der `TypographyInitGenerator` hingegen erstellt komplexe Hilfsfunktionen zur Initialisierung von Schriftfamilien sowie zur Erzeugung der Textstile. Je nach gewählter Zielplattform generiert dieser entweder ein Composable oder eine Variable, welche die Typografie-Tokens bereitstellt.

5.6 Orchestrierungsprozess

Das Modul `facade` bildet die Anwendungsschicht des Code-Generierungsprozesses. Seine primäre Aufgabe ist die Orchestrierung aller Module zu zwei durchgängigen Prozessen. Das Modul weist eine minimale öffentliche Schnittstelle auf, um die interne Komplexität vor den Eintrittspunkten zu verbergen. Sämtliche Implementierungsdetails sind als internal gekapselt. Das gewährleistet eine saubere Modulgrenze und Unabhängigkeit.

Ein zentraler Aspekt dieser Orchestrierung ist die Unterstützung von White-Label-Anwendungen durch eine markenbasierte Iteration. Für jede in der Konfiguration hinterlegte Marke wird ein dedizierter Generierungszyklus durchlaufen. Die resultierenden `FileSpec`-Objekte werden so aufbereitet, dass sie den korrekten Source Sets innerhalb des Zielprojekts zugeordnet werden können.

Den funktionalen Eintrittspunkt bilden Use-Case-Klassen. Dabei initialisiert der `GenerateTokensUseCase` die Generierung der Design-Tokens, während der `DownloadIconsUseCase` den Export der grafischen Ressourcen steuert. Zur Entkopplung dieser Use-Cases von der konkreten Geschäftslogik dienen Repository-Schnittstellen als abstrakte Abstraktionsebene. Die zentrale Instanziierung aller Objekte übernimmt der `DIContainer`. Dieser verarbeitet die externe Konfiguration und nutzt Lazy Delegation zur effizienten Assemblierung der Abhängigkeiten. Dadurch werden Objekte erst bei der tatsächlichen Verwendung erzeugt, was die Systemressourcen schont. Da beide Use-Cases technisch als `suspend`-Funktionen definiert sind, kapseln die Aufrufer diese an der Systemgrenze mittels `runBlocking`. Dies geschieht beispielsweise innerhalb einer Gradle-Task-Action. Durch dieses Verfahren bleibt die asynchrone Coroutine-Nutzung innerhalb der Kernmodule isoliert und der Eintrittspunkt operiert nach außen hin

vollständig synchron.

5.6.1 Orchestrierung der Token-Generierung

Die Generierung der Design-Tokens folgt innerhalb der `TokenGenRepositoryImpl` einer Abfolge:

1. Das `ConfigRepository` lädt die nutzerdefinierten Parameter.
2. Das `FigmaRepository` ruft sämtliche Variablen und Collections über die Figma API ab.
3. Für Textstile werden spezifische Figma-Nodes abgefragt, die Bindungen zu Variablen für Schriftgröße, Zeilenhöhe und Schriftschnitte enthalten.
4. Die `TokenTaskFactory` erzeugt für jede Gruppe (Farben, Abstände, Typografie) ein spezifisches `GenerationTask`-Objekt.
5. Für jede Aufgabe wird ein `TokenClass`-Objekt durch Auflösung der Referenzketten generiert. Abschließend erfolgt die Code-Erzeugung und die Speicherung der `.kt`-Dateien.

Der `FontContextBuilder` fungiert als Brücke zwischen der Token-Auflösung und der Code-Generierung für Typografie-Aufgaben. Er wird nur aktiv, wenn die Integration von Google Fonts in der Konfiguration aktiviert ist.

Der Prozess analysiert das `TokenClass` und sammelt alle referenzierten Schriftarten sowie deren Gewichte. Das Werkzeug bestimmt den Zielpfad dabei basierend auf der Zielplattform.

Der Download-Prozess ist fehlertolerant konzipiert. Schlägt der Abruf einer Schriftfamilie fehl, wird eine Warnung ausgegeben und im generierten Code automatisch auf `FontFamily.Default` zurückgegriffen. Andere Schriftarten der Pipeline bleiben davon unberührt. Die resultierenden Metadaten werden in einen `FontGenerationContext` überführt, der Daten über Plattform enthält.

6 Evaluation

In diesem Kapitel wird die Anwendung von FigmaResGen innerhalb eines Kundenprojekts der Appsfactory Group evaluiert. Ziel der Untersuchung ist es, die Praxistauglichkeit des Werkzeugs unter industriellen Entwicklungsbedingungen zu prüfen und den Erfüllungsgrad der funktionalen sowie nicht-funktionalen Anforderungen kritisch zu hinterfragen.

6.1 Anwendung im Projektkontext

Die erste Evaluation des Werkzeugs erfolgte im Rahmen eines CMP-Projekts für die Plattformen Android und iOS. Die Autorin war als Teil des Entwicklungsteams an der Umsetzung beteiligt und integrierte das Werkzeug, um die automatisierte Generierung der Figma-Exports und deren Nutzbarkeit zu evaluieren.

Der Fokus dieser Untersuchung lag darauf, ob das Werkzeug fähig ist, die in Kapitel etablierte, hierarchische Collection-Struktur stabil zu verarbeiten und in ein Anwendungsprojekt zu überführen. Das betrachtete Projekt wies folgendes hinsichtlich der Design-Tokens auf:

- Tokens für Farben: Implementierung eines Light- und Dark-Modes.
- Tokens für Abstände und Typografie: Differenzierung zwischen Mobile- und Tablet-Modi zur Gewährleistung einer responsiven Benutzeroberfläche.

Das Werkzeug wurde hierbei als Gradle-Plugin integriert.

6.2 Ergebnisse und Performance

Die praktische Erprobung bestätigte eine signifikante Beschleunigung der Initialisierungsphase. Die vollständige Konfiguration, der automatisierte Abruf und die Generierung sämtlicher Design-Tokens, sowie der Export der grafischen Ressourcen beanspruchte weniger als 30 Minuten. Dieser Prozess umfasste die folgenden Schritte:

- Erstellung der Gradle-Konfiguration,
- automatisierte Generierung der Figma-Exports in typsicheren Kotlin-Quellcode,
- Integration der generierten Ressourcen in das zentrale `AppTheme`.

Damit wurde nachgewiesen, dass das Werkzeug dem Entwickler die Möglichkeit bietet, Ressourcen automatisiert bereitzustellen und dabei fähig ist, Tokens in mehreren Modi konsistent zu generieren.

6.3 Erkenntnisse und Optimierungspotenziale

Trotz der positiven Performance-Metriken identifizierte der Einsatz im Projekt spezifische Schwachstellen und Optimierungspotenziale, die über eine rein funktionale Betrachtung hinausgehen.

Der Prozess des Herunterladen der Icons offenbarte eine starke Abhängigkeit von der Datenqualität innerhalb der Figma-Datei. Es wurde festgestellt, dass Namenskollisionen den Export-Prozess beeinträchtigen können, sofern Icons identische Bezeichner tragen. Künftig ist eine integrierte Validierungslogik erforderlich, die bereits während des Exportvorgangs auf solche Redundanzen hinweist. Ein weiterer Kritikpunkt betrifft die Effizienz des Transfers. Das vollständige erneute Herunterladen sämtlicher Icons bei jedem Aufruf der `downloadIcons`-Aufgabe erwies sich als zeitintensiv, insbesondere wenn lediglich vereinzelte Ressourcen hinzugefügt oder modifiziert wurden. Eine notwendige Erweiterung zur Schonung der Build-Ressourcen wäre daher die Implementierung einer inkrementellen Synchronisation, die nur ausgewählte oder neue Ressourcen überträgt.

Da das Projekt keine Multi-Brand-Anforderungen stellte, konnte diese Funktionalität nicht unter Live-Bedingungen im Team-Prozess getestet werden. Um die Funktionalität dennoch zu validieren, wurde der Autorin Zugriff auf eine Figma-Datei eines bestehenden Multi-Brand-Projektes gewährt. Technisch verlief die Generierung der Tokens für verschiedene Markenidentitäten erfolgreich. Dennoch offenbarte dieser Test eine konzeptionelle Schwäche: Die Konfiguration des Icon-Exports in Multi-Brand-Szenarien ist derzeit zu redundant. Da grafische Ressourcen oft in unterschiedliche markenspezifische Verzeichnisstrukturen verteilt werden müssen, ist momentan für jedes Zielverzeichnis eine separate Konfiguration erforderlich. Dies widerspricht dem Ziel einer minimalen Konfigurationslast und stellt einen zentralen Ansatzpunkt innerhalb der Gradle-Extension dar.

Zusammenfassend lässt sich festhalten, dass die Muss-Anforderungen funktional erfüllt wurden. Dennoch zeigt die Evaluierung, dass für einen skalierten Einsatz in White-Label-Projekten weitere Optimierungen hinsichtlich der Konfiguration und des inkrementellen Ressourcen-Handlings notwendig sind.

7 Zusammenfassung und Ausblick

Die vorliegende Arbeit befasste sich mit der Konzeption und Implementierung des Werkzeugs FigmaResGen, um den identifizierten Medienbruch zwischen dem Design in Figma und der technischen Umsetzung in Compose zu schließen. Der manuelle Übertragungsprozess von Figma-Exports wurde als zeitaufwendige und fehleranfällige Schwachstelle innerhalb des Entwicklungsprozesses der Appsfactory GmbH identifiziert. Im Rahmen der Arbeit wurde eine Lösung entwickelt, die über die Figma REST API Design-Tokens für Farben, Abstände und Typografie sowie grafische Ressourcen automatisiert extrahiert. Durch die Anwendung der Clean Architecture und einer modularen Projektstruktur wurde sichergestellt, dass das Werkzeug sowohl als Gradle-Plugin nahtlos integriert als auch als eigenständige lokale Applikation genutzt werden kann.

Ein wesentliches technisches Merkmal ist die iterative Alias-Auflösung, die komplexe Referenzketten innerhalb von Figma-Variablen stabil auflöst und in typsicheren Kotlin-Quellcode transformiert. Das Werkzeug unterstützt dabei sowohl die Generierung in verschiedenen Modi als auch komplexe Multi-Brand-Architekturen, was für die Entwicklung von White-Label-Anwendungen von zentraler Bedeutung ist. Zudem wurde durch die Berücksichtigung von CMP eine plattformübergreifende Nutzbarkeit der Ressourcen sichergestellt.

Die Evaluation in einem industriellen Kundenprojekt auf Basis von CMP bestätigte die Praxistauglichkeit, da der initiale Einrichtungsprozess des Designsystems auf unter 30 Minuten reduziert werden konnte. Dennoch zeigte die kritische Analyse, dass insbesondere bei der Skalierung von White-Label-Projekten Optimierungsbedarf besteht. Während die verpflichtenden Anforderungen erfüllt wurden, offenbarten das vollständige Herunterladen von Icons sowie die Redundanz in der markenspezifischen Konfiguration Ansatzpunkte für eine effizientere Ressourcenverwaltung in künftigen Versionen.

7.1 Ausblick

Trotz der erfolgreichen Implementierung unterliegt das Werkzeug der stetigen technologischen Evolution der Figma-Plattform sowie den Fortschritten im Bereich der KI-gestützten Softwareentwicklung. Am 28.10.2025 führte Figma die sogenannten „Extended Collections“ ein, die speziell für die Verwaltung komplexer Multi-Brand-Architekturen optimiert sind. Erste Analysen im Februar 2026 ergaben, dass die JSON-Antwortstrukturen dieser neuen Funktionen signifikant von den bisherigen API-Modellen abweichen. Da die Unterstützung dieser Features nicht im ursprünglichen Umfang dieser Arbeit lag, stellt die Anpassung der Transformationslogik an diese neuen Strukturen einen zentralen Mei-

lenstein für die Weiterentwicklung von FigmaResGen dar.

Ein weiterer relevanter Aspekt für künftige Untersuchungen ist der Effizienzvergleich mit aufstrebenden KI-gestützten Ansätzen. Es gilt zu evaluieren, wie leistungsfähig FigmaResGen im Vergleich zur Nutzung von KI-Agenten operiert, die über das Figma Model Context Protocol direkt auf Designsysteme zugreifen. Hierbei sollte untersucht werden, ob ein KI-Agent durch entsprechende Prompts Design-Ressourcen ebenso zuverlässig und typsicher extrahieren kann wie eine spezialisierte, regelbasierte Lösung. Zudem bleibt die Implementierung einer inkrementellen Synchronisation ein wichtiges Ziel.

Zusammenfassend bietet FigmaResGen eine funktionale Basis zur Reduktion von Übertragungsfehlern und des manuellen Aufwands zwischen Design und Entwicklung. Die langfristige Relevanz des Werkzeugs wird davon abhängen, wie flexibel es an neue Schnittstellenparadigmen und automatisierte KI-Workflows angepasst werden kann.

Literaturverzeichnis

[1] Ethan Marcotte. *Responsive Design Workflow*. 25. Mai 2010. URL: https://alistapart.com/article/responsive-web-design/ (besucht am 31. 01. 2026).

[2] Stephen Hay. *Responsive Design Workflow*. New Riders, 2013. ISBN: 9780321887863.

[3] Brad Frost. *Atomic Design*. 10. Juni 2013. URL: https://bradfrost.com/blog/post/atomic-web-design/ (besucht am 31. 01. 2026).

[4] Brad Frost. *Atomic Design*. Brad Frost, 2016. ISBN: 9780998296609.

[5] UX Tools. *2024 Design Tools Survey*. 2024. URL: https://www.uxtools.co/survey/interface-design/overview (besucht am 31. 01. 2026).

[6] Figma. *Figma for Design Systems*. URL: https://www.figma.com/design-systems/ (besucht am 30. 01. 2026).

[7] Zero Height Limited. *Design Systems Report 2025*. Industry Report. 2025. URL: https://zeroheight.com/how-we-document/ (besucht am 31. 01. 2026).

[8] Google. *Material Design - Design tokens*. URL: https://m3.material.io/foundations/design-tokens/overview (besucht am 01. 02. 2026).

[9] Appsfactory GmbH. *Arbeitsweise*. URL: https://www.appsfactory.de/de/arbeitsweise (besucht am 31. 01. 2026).

[10] Design Tokens Community Group. *DTCG Glossary*. URL: https://www.designtokens.org/ (besucht am 01. 02. 2026).

[11] Design Tokens Community Group. *Design Tokens Format Module 2025.10*. URL: https://www.w3.org/community/reports/design-tokens/CG-FINAL-format-20251028/ (besucht am 04. 02. 2026).

[12] Figma Learn. *Update 1: Tokens, Variables, and Styles*. URL: https://help.figma.com/hc/en-us/articles/18490793776023 (besucht am 30. 01. 2026).

[13] Figma Developers. *Rest API - Introduction*. URL: https://developers.figma.com/docs/rest-api/ (besucht am 30. 01. 2026).

[14] Google. *Develop - Core areas - UI -Dokumentation*. URL: https://developer.android.com (besucht am 04. 02. 2026).

[15] JetBrains. *Compose Multiplatform UI framework*. URL: https://kotlinlang.org/docs/multiplatform/compose-multiplatform.html (besucht am 22. 02. 2026).

[16] ComposeGears. *Valkyrie — SVG/XML to Compose ImageVector converter*. URL: https://github.com/ComposeGears/Valkyrie (besucht am 16. 02. 2026).

[17] Square. *Kotlin Poet*. URL: https://square.github.io/kotlinpoet/ (besucht am 16. 02. 2026).

[18] Gradle. *Gradle User Manual*. URL: https://docs.gradle.org/current/userguide/userguide.html (besucht am 16. 02. 2026).

[19] Yunnong Chen u. a. „DesignCoder: Hierarchy-Aware and Self-Correcting UI Code Generation with Large Language Models“. In: *arXiv preprint arXiv:2506.13663* (2025). DOI: 10.48550/arXiv.2506.13663. URL: https://arxiv.org/abs/2506.13663.

[20] Zhu Lin u. a. „Figma2Code: Automatic Code Generation Method for Figma Design Drafts“. In: *Journal of Computer-Aided Design and Computer Graphics* 37 (2025), S. 321–329. DOI: 10.3724/SP.J.1089.2023-00336. URL: https://www.jcad.cn/en/article/doi/10.3724/SP.J.1089.2023-00336.

[21] Wilhelm Löfsten Oscarsson. „From Pixels to Production: Automating Design-to-Code Work-flows with Figma“. Magisterarb. inköping University, 2025. URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2:1964634.

[22] Tokens Studio. *Tokens Studio for Figma*. URL: https://www.figma.com/community/plugin/843461159747178978/tokens-studio-for-figma (besucht am 09. 02. 2026).

[23] Deirdré Straughan. *Style Dictionary: An Open Source Tool for Building Customer Trust Through Design Consistency*. URL: https://aws.amazon.com/blogs/opensource/style-dictionary-trust-design-consistency/ (besucht am 09. 02. 2026).

[24] Style Dictionary. *Documentation*. URL: https://styledictionary.com/getting-started/installation/ (besucht am 09. 02. 2026).

[25] Daniil Subbotin. *FigmaExport*. URL: https://github.com/RedMadRobot/figma-export (besucht am 22. 02. 2026).

[26] Helmut Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Spektrum Akademischer Verlag Heidelberg, 2009. ISBN: 9783827417053.

[27] Robert C. Martin. *Clean Architecture: das Praxis-Handbuch für professionelles Softwaredesign*. Spektrum Akademischer Verlag Heidelberg, 2018. ISBN: 9783958457249.

[28] The Object Management Group. *ASTM™ — Abstract Syntax Tree Metamodel*. 2011. URL: https://www.omg.org/spec/ASTM (besucht am 23. 02. 2026).