Bachelor-Thesis

Kozeptionierung eines Hard- und Softwareteststands für automatisierte Tests von Metalldetektoren für die Kampfmittelräumung

Tobias Braun
Matrikel-Nr.: 767 369

Erstprüfer: Christian Kücherer

Zweitprüferin: Madrid Martinez

Abgabedatum: 30.06.2023

Bachelor-Studiengang Medien und Kommunikationsinformatik

Fakultät Informatik

Hochschule Reutlingen

Zusammenfassung:
Das Ziel dieser Bachelor Arbeit ist die Verbesserung der Endkontrolle von Metalldetektoren des Typs VMx4. Dazu wird eine Architektur für einen neuen Hard- und Softwareteststand entwickelt und ein Prototyp implementiert. Damit diese Architektur die Wünsche und Bedürfnisse aller Beteiligten erfüllt, müssen diese zuerst erfasst werden. Dazu wird eine Literatur Recherche durchgeführt, um die nötigen Grundlagen zum Thema Testautomatisierung und Requirement Engineering (RE) zu erfassen. Anschließend wird der erarbeitete Requirement Engineering (RE) Prozess durchgeführt. Basierend auf dem erarbeiteten Anforderungskatalog wird eine Architektur für das Testautomatisierungssystem (TAS) entwickelt. Auf Grundlage dieser Architektur wird ein Prototyp implementiert und anhand diesem evaluiert, in welchem Umfang die erfassten Anforderungen erfüllt werden.

Diese Thesis basiert auf internen und vertraulichen Daten des Unternehmens Vallon GmbH. Diese Thesis darf unbefugten Dritten ohne ausdrückliche Zustimmung des Unternehmens und des Verfassers nicht zugänglich gemacht werden. Eine Vervielfältigung und Veröffentlichung der Thesis ohne ausdrückliche Genehmigung – auch in Auszügen – ist nicht erlaubt. Die Sperrfrist endet fünf Jahre nach Einreichung der Thesis am 30.06.2028. Dieser Sperrvermerk bezieht sich nur auf die folgenden Teile dieser Thesis: Kapitel 4.1 - 4.2, Kapitel 6, Anhang S.103 - 137.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

1.1 Motivation, Kontext und Gegenstand

Metalldetektoren stellen eines der wichtigsten Werkzeuge in der Kampfmittelräumung dar. Sie helfen gefährliche Objekte, wie Mienen, Sprengsätze, oder Munition aufzuspüren, damit diese sicher entschärft werden können. Das Leben der Bediener*innen kann von der korrekten Funktion dieser Geräte abhängen. Die Firma Vallon GmbH in Eningen unter Achalm produziert Metalldetektoren speziell für diesen Anwendungszweck. Vallon Detektoren werden weltweit von militärischen und zivilen Organisationen eingesetzt. Die korrekte Funktion und Zuverlässigkeit dieser Geräte kann über Leben und Tod entscheiden. Der aktuelle End of Line Funktionstests für die Metalldetektoren ist zwar erprobt, lässt aber noch viel Raum zur Verbesserung. Manuelle Tätigkeiten können automatisiert werden, um diese zu beschleunigen und besser reproduzierbare Ergebnisse zu erhalten. Es wurden bereits Hard- und Softwarekomponenten entwickelt, wie neue Schnittstellen zu den Geräten, die die Testqualität verbessern sollen, doch erlaubt es das aktuelle Testsystem nicht, diese effizient anzubinden, da es nicht für eine nachträgliche Erweiterung ausgelegt wurde.

1.2 Begriffsdefinitionen

Die Tabelle 1.1 erklärt wichtige Begriffe, die für das Verständnis dieser Arbeit benötigt werden.

Tabelle 1.1: Begriffsdefinitionen

Stakeholder*in	Personen, Systeme, Prozesse, oder Organisationen, die Interesse an einem System haben [PR21, 1]

Anforderung	Wunsch, oder Bedürfnis einer Stakeholder*in, die ein Produkt erfüllen soll [PR21, 1]
Requirement Engineering (RE)	Das disziplinierte und systematische Vorgehen bei der Erfassung und Verwaltung von Anforderungen an ein Produkt [GvSB22]
Software Architektur	Die Summe aller Architekturentscheidungen, die während der Entwicklung der Software getroffen wurden [Zör12, 32]
End of Line (EOL)	Nach der Produktion, vor dem Versand eines Produkts
Testautomatisierungssystem (TAS)	Gesamtheit der Hard- und Software eines System zur Automatisierung von Tests [Bau21, 8]
System under Test (SUT)	Hard- und/oder Softwaresystem das einem Test unterzogen wird [Bau21, 8]
Testscript	Formale Beschreibung eine Testablaufs [Bau21, 92]
Testbett	Gesamtheit aller Testschnittstellen, physisch oder digital zu einem System under Test (SUT) [ZGHY18]

1.3 Forschungsfragen

Das Ziel der Bachelor Arbeit ist es, die Qualitätssicherung der Produkte der Firma Vallon zu verbessern. Dazu soll ein neues Testautomatisierungssystem (TAS) für einen neuen End of Line (EOL)-Test entwickelt werden. Dazu müssen die folgenden Forschungsfragen geklärt werden, die in Tabelle 1.2 beschrieben sind.

Tabelle 1.2: Forschungsfragen

RQ1	Welche Anforderungen muss ein Hard- und Softwareteststand für automatisierte Tests von Metalldetektoren erfüllen?
RQ2	Wie kann eine Architektur für einen automatisierten, modularen Teststand für Metalldetektoren aussehen?

1.4 Lösungsansatz und Methodik

Für die Klärung der *RQ1* 1.2 wurde eine Literaturrecherche durchgeführt, um einen Prozess für die Erfassung der Anforderungen zu erarbeiten. Dieser wurde anschließend durchgeführt und die Ergebnisse dokumentiert werden. Die Ergebnisse dienten als Grundlage für *RQ2* 1.2. Weiter wurden für die Klärung von *RQ2* 1.2 Grundlagen im Bereich der Testautomatisierung und Softwarearchitektur erfasst, ebenfalls durch eine Literatur Recherche. Basierend auf den Ergebnissen von *RQ1* 1.2 und den erfassten Grundlagen, wurde ein Entwurf für eine Architektur entwickelt und dokumentiert. Um diese Architektur zu evaluieren, wurde anschließend ein Prototyp implementiert und geprüft, in wie weite er die gestellten Anforderungen erfüllt.

1.5 Struktur der Arbeit

Im Zuge dieser Arbeit werden zuerst im Kapitel 2 wichtige Grundlagen zu den betroffenen Themengebieten geklärt. Im darauf folgenden Kapitel 3 wird ein Requirement Engineering (RE) Prozess erarbeitet und verschiedene TAS Architekturen beschrieben. Die Forschungsfrage 1 (Tabelle 1.2) wird im Kapitel 4 behandelt. Dazu wird der zuvor erarbeitete Requirements Engineering Prozess durchgeführt und die Ergebnisse dokumentiert. Das Kapitel 4 beschäftigt sich mit dem Entwurf der Software Architektur für die Klärung von Forschungsfrage 2 (Tabelle 1.2). In Kapitel 5 wird ein Prototyp beschrieben, der die entworfene Architektur umsetzt. Dieser Prototyp wird in Kapitel 6 evaluiert, um festzustellen, in wie weit er die erfassten Anforderungen erfüllt.

2 Grundlagen

In diesem Kapitel werden die Grundlagen für die verschiedenen Themengebiete, Testautomatisierung, Requirements Engineering und Software Architektur, die diese Bachelor Thesis betreffen aufgearbeitet und zusammen gefasst. Es soll ein Testautomatisierungs-System entwickelt werden. Dazu müssen zuerst die Grundlagen zum Thema Testautomatisierung erfasst und ausgewertet werden, zu lesen im Unterkapitel 2.1. Für die anschließende Planung und Umsetzung des TAS muss bestimmt werden, was die Anforderungen an das gewünschte System ist. Diese werden im Zuge des Requirements Engineering erfasst. Die nötigen Grundlagen dazu werden im Unterkapitel 2.2 aufgearbeitet. Basierend auf den erhobenen Anforderungen kann anschließend eine Software Architektur entworfen werden, die diese Anforderungen erfüllt. Die Grundlagen für die Erstellung und Dokumentation dieser werden in Unterkapitel 2.3 behandelt. Beim Entwurf der Software wird auf Standard Entwurfsmuster zurück gegriffen. Die verwendeten Entwurfsmuster werden um letzten Unterkapitel 2.4 beschrieben.

2.1 Grundlagen Testautomatisierung

Mit der fortschreitender Automatisierung von Arbeitsprozessen rückt auch immer weiter die Automatisierung von Soft- und Hardwaretests in den Fokus. Ziele sind dabei die Reduktion von Kosten, die Beschleunigung des Arbeitsprozesses und die Steigerung der Testqualität [Bau21, 12]. Dazu werden speziell entwickelte Soft- und unter Umständen auch Hardwarewerkzeuge benötigt. Zusammen bilden sie ein Testautomatisierungssystem (TAS). Die Gesamtheit des TAS und der Prozesse, in die es eingebunden ist bezeichnet man wiederum als Testautomatisierungslösung (TAL) [Bau21, 8]. Das Ziel dieser Arbeit ist die Entwicklung eines TAS. Daher müssen die Grundlagen des automatisierten Testens erläutert werden.

2.1.1 Gründe zu Testen

Ziele des Testens sind nach Andreas Spillner und Tilo Linz [SL19, 14]:

- Qualitative Bewertung von Arbeitsergebnissen des Requirements Engineerings, der Software Architektur und Software Entwicklung
- Nachweis, dass Anforderungen vollständig umgesetzt und das Testobjekt funktioniert wie erwartet
- Informationen zur Qualität des SUT liefern
- Fehler und Mängel frühzeitig Aufdecken
- Informationen liefern, um zu Entscheiden, ob das SUT für die Integration in weitere Systemteile freigegeben werden kann
- Nachweisen, dass das SUT vertraglichen, rechtlichen, oder regulatorischen Anforderungen entspricht

Welche dieser Ziele mit Tests abgedeckt werden kann variieren, muss jedoch vor der Entwicklung des Tests festgelegt werden [SL19, 15]. Automatisierung kann helfen Tests durchzuführen, die manuell zu aufwändig, oder sogar unmöglich sind. Dazu zählen Performancetests, bei denen getestet wird, wie sich ein System unter steigender Auslastung verhält. Lasttests, bei denen die Maximale Kapazität eines System ermittelt wird. Stresstests die Testen, wie sich das System bei Überschreiten der Kapazitätsgrenze verhält. Hierfür müssen große Menge an Interaktionen mit dem System, in einem sehr engen Zeitrahmen durchgeführt werden. Diese Menge an Eingaben sind schwierig, bis Unmöglich manuell durchzuführen. Hier kann Automatisierung helfen. Aber auch Langzeittests, die nicht zeitkritisch sind, jedoch das konstante Wiederholen eine repetitiven Ablaufs erfordern, stellen eine große Belastung für die Bediener*innen dar. Hier können TAS diese entlasten. Durch die Reduktion der Interaktionen des Bedieners mit dem TAS steigt auch die Konsistenz und Wiederholbarkeit der Testabläufe und damit ihre Aussagekraft [Bau21, 12].

2.1.2 Testzeitpunkt

Tests können an verschiedenen Zeitpunkten in der Entwicklung, oder Produktion eines Produkts durchgeführt werden. Während der Entwicklung helfen sie festzustellen, ob das Produkt geforderte Anforderungen bereits zufriedenstellend erfüllt und können Fehler und Schwachstellen aufdecken. Am Ende der Entwicklung helfen sie zu validieren, ob das Produkt alle Anforderungen erfüllt. Bei Cyber Physical System (CPS) sollte auch nach der Entwicklung, am Ende

der Produktion geprüft werden, ob das Produkt die gewünschten Anforderungen erfüllt. Welche Tests, zu welchem Zeitpunkt der Entwicklung eines Produkts durchgeführt werden können beschreibt das V-Modell. Das V-Modell, eingeführt in [Dep85] wurde im laufe der Geschichte von verschiedenen Personen und Institutionen erweitert. Das V-Modell XT Bund [Inf19] ist eine Erweiterung des V-Modell, dass vom Informationstechnikzentrum Bund, für interne Software Entwicklungen, von Bundesbehörden heraus gegeben wurde. In [SL19, 26] wird das allgemeine V-Modell beschrieben. Die V-Modell beschreiben ein Strukturiertes Vorgehen bei der Software Entwicklung, ähnlich dem Wasserfallmodell [Roy87]. Es folgen jedoch auf die Entwicklungsphasen eine Reihe an Tests. Dabei wird jeder Entwicklungsphase ein korrespondierender Test zugeordnet, um deren Ergebnisse zu validieren. Bei der Entwicklung arbeitet die Entwickler*in die Stufen des Wasserfalls ab. Nach der Programmierung der Software, folgen die verschiedenen Tests und validiert die entwickelte Software in mehreren Schritten, siehe Tabelle 2.1.

Tabelle 2.1: Test Zeitpunkte

Test	Validiert
Komponententest	Komponentenspezifikation
Integrationstest	Technischer Systementwurf
Systemtest	Funktionaler Systementwurf
Abnahmetest	Anforderungsdefinition

2.1.3 Risiken und Chancen von automatisierten Tests

Das automatisieren von Test birgt Risiken, aber auch Chancen. Für die Testautomatisierung müssen unter Umständen Hard- und Softwarewerkzeuge neu entwickelt, oder adaptiert werden. Dies erzeugt Kosten und Entwicklungsaufwand. Auch müssen Mitarbeiter für die Bedienung des TAS geschult werden. Es müssen Prozesse um das TAS erstellt und implementiert, oder vorhandene Prozesse daran angepasst werden. Bei Änderungen der Hard- oder Software des SUT muss geprüft werden, ob vorhandenen Test Hard- und Software, oder Testabläufe daran angepasst werden müssen [Bau21, 14]. Auch können Tests eine falsche Sicherheit vermitteln. Wenn neue Features des SUT noch nicht von

Tests abgedeckt werden, können diese Fehlerhaft sein, das TAS meldet jedoch dennoch einen Erfolg der Tests [SL19, 23].

Eine Einführung automatisierter Tests früh im Entwicklungszyklus eines Systems hilft die Anforderungen und Vorgaben für dieses im Blick zu behalten [SL19]. Dadurch können teure und zeitintensive Fehlentwicklungen vermieden werden. Es kann jedoch auch zu enormen Mehrkosten kommen, da das TAS immer wieder an den aktuellen Entwicklungsstand des SUT angepasst werden muss [Bau21, 14].

2.1.4 Testqualität

Nach [Wit19, 56] sollte ein automatisierter Test folgende Kriterien erfüllen:

- Reproduzierbarkeit: Ergebnisse sind nachvollziehbar und werden systematisch erfasst. Zufälle, oder externe Faktoren haben keinen Einfluss auf die Ergebnisse des Tests
- Planbarkeit: Testaufwand und Testnutzen sind vorhersagbar
- Wirtschaftlichkeit: Testaufwand und Testnutzen werden evaluiert und optimiert
- Risiko- und Haftungsreduktion: Kritische Fehler werden erkannt und können behoben werden, bevor das Produkt ausgeliefert wird

2.1.5 Testmethodiken

Es gibt verschiedene Ansätze, wie getestet wird. Diese unterscheiden sich in Faktoren wie Aufwand, Genauigkeit und wie dynamisch die möglichen Tests sind. Beim Model-Based Testing wird das SUT in eine Umgebung gebracht, die verschiedene Betriebssituationen und Abläufe simuliert. Das TAS simuliert anhand von Vorgegebenen Testabläufen Eingaben und prüft ob das SUT das erwartete Verhalten zeigt. Hierbei wird zwischen dem Closed Loop Testing und dem Open Loop Testing unterschieden. Beim Closed Loop Test wird eine Feedbackschleife zwischen dem TAS und dem SUT erzeugt, zu sehen in Abbildung 2.1. Dadurch kann das TAS während des Tests auf die Ausgaben des SUT reagieren. Bei Open Loop Test Open Loop Tests sind Eingaben in das SUT und die Auswertung der Ausgaben entkoppelt, zu sehen in Abbildung 2.2. Das TAS läuft einen fixen Ablauf an Eingaben ab und vergleicht die Ausgaben mit einer Liste der erwarteten Ausgaben [ZGHY18].

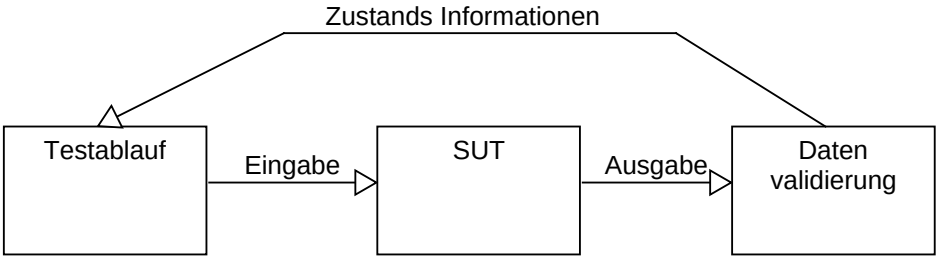


Abbildung 2.1: Closed Loop Test

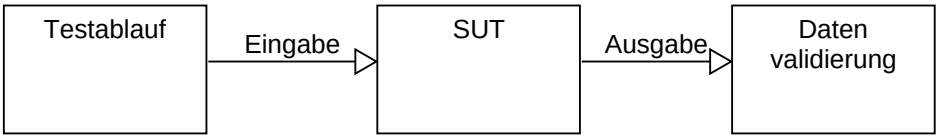


Abbildung 2.2: Open Loop Test

Das Search Based Testing [McM11] zielt darauf ab, Grenzfälle in Systemen zu Identifizieren. Dazu werden Optimierungs- und Genetischealgorithmen eingesetzt, um automatisiert Eingaben für Tests zu erzeugen. Diese nutzten selbst verbessernde Systeme, um Eingaben zu identifizieren, die zu bestimmtem erwünschtem, oder unerwünschtem Verhalten führt. Dadurch lassen sich leicht, automatisiert Sets an Eingabe Daten für Tests generieren, um Grenzfälle im System zu identifizieren und zu testen.

Bei Fault Injection Testing [ZGHY18] werden gezielt Fehler in der Umgebung des SUT herbeigeführt. Hierzu werden Daten verändert, gelöscht, oder erzeugt, oder der Ablauf von Prozessen verändert, oder gestört. Es wird überwacht, wie das SUT auf diese abnormalen Zustände reagiert.

2.1.6 Anbindung von System under Test (SUT)s

Bei jeder Testart muss das TAS mit dem System under Test (SUT) kommunizieren, um Eingaben zu machen und die Ausgaben des SUT auszulesen, um diese auswerten zu können. Je nach Art und Schnittstellen des SUT stehen verschiedene Schnittstellen für die Kommunikation mit diesem zur Verfügung. Diese

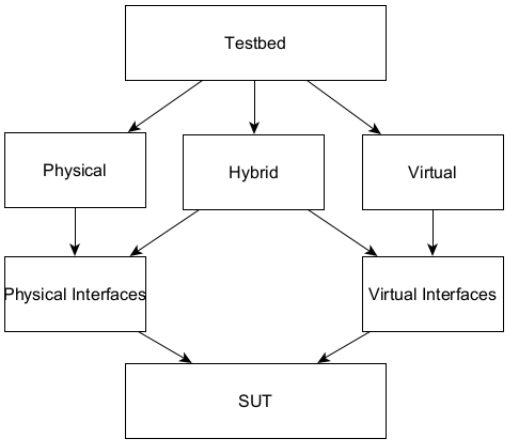


Abbildung 2.3: Testbett Arten

können physisch sein, in Form von Taste, Bildschirmen, Aktuatoren und Sensoren, oder digital in Form von Netzwerk, oder USB Schnittstellen. Entsprechend der Charakteristik dieser Schnittstellen muss ein entsprechendes System konstruiert werden, über das das TAS mit diesem interagieren kann. Die Gesamtheit dieser Schnittstellen bezeichnet man als Testbett [ZGHY18]. Es werden grundsätzlich drei Arten von Testbeds unterschieden, visualisiert in Abbildung 2.3:

- Hardwarebasiert: besteht aus physischen Systemen, die physisch mit dem System under Test (SUT) interagieren
- Simulationsbasiert: besteht nur aus Software und interagiert über digitale Schnittstellen mit dem System under Test (SUT)
- Hybrid: besteht sowohl aus physischen Systemen, als auch aus Software. Es interagiert mit dem System under Test (SUT) über physische und digitale Schnittstellen

Weiter unterteilt werden zentralisierte und verteilte Testbetts:

- Zentralisierte Testbeds: alle Schnittstellen mit dem SUT sind in einem System gesammelt
- Verteilte Testbeds: es existieren verschiedene Systeme, die unterschiedliche Schnittstellen des SUT ansprechen

Um die Kommunikation mit dem SUT zu vereinfachen, oder den Zugriff auf bestimmte Funktionen, oder Informationen überhaupt zu ermögliche, können sog. Test Hooks in das SUT integriert werden. Test Hooks sind Schnittstellen, die einem System hinzugefügt werden, um die Testbarkeit, oder Automatisierbarkeit

zu erhöhen. Diese Hook erleichtern das Anbinden an das TAS und erlauben eine detailliertere Analyse des SUT. Dabei ist jedoch zu beachten, dass sie, sollten sie vor der Auslieferung des SUT nicht entfernt, oder ausreichend gesichert werden, eine gravierende Sicherheitslücke darstellen können [Bau21, 128]. Dabei stellt jedoch das entfernen der Hooks eine Veränderung des Systems dar. Da dies erst nach dem Testvorgang erfolgen kann, kann das SUT im neuen Zustand nicht umfassend getestet werden. Fehler, die durch die Entfernung der Hooks entstehen, lassen sich so schwerer identifizieren [Bau21, 128]. Weiter stellt die Testbarkeit jedoch auch ein Qualitätskriterium nach ISO/IEC 25010 [ISO11] dar und muss bereits in der Planung und Entwicklung eines Systems bedacht werden. So können sichere Hooks bereits beim Entwurf der Software bedacht werden.

2.1.7 Methoden für die Testerstellung

Um Reproduzierbar und Planbar zu sein müssen Tests ausreichend definiert und dokumentiert werden [Wit19, 56]. Testscripte beschreiben Abfolgen von Eingaben, die das TAS in das SUT tätigen soll, sowie Informationen, wie die Ausgaben des SUT zu bewerten sind. Konkrete Testscripte können manuell, oder automatisiert erstellt werden [Bau21, 92]. Bei der manuellen Testschritterstellung, werden zuerst die Ziele des Tests, sowie Voraussetzungen für die Ausführung festgelegt. Dies beinhaltet eine Beschreibung des benötigten Ausgangszustand des SUT vor Beginn des Tests. Dann werden einzelne Testfälle entworfen. Diese bestehen aus den Eingabewerten, sowie Parametern, wie Ausgaben, sowie der Zustand des SUT auszuwerten und zu Interpretieren sind [Bau21, 136]. Testscripte können in verschiedenen Formaten erstellt werden. Als Quellcode, der direkt vom SUT ausgeführt wird, aber auch als abstrakte Beschreibung die das SUT interpretiert. Formate hierfür können klassische Beschreibungssprachen wie Extensible Markup Language (XML), oder JavaScript Object Notation (JSON), aber auch die in [iee10] beschriebene speziell hierfür entwickelte Automatic Test Markup Language (ATML), verwendet werden. Werden bereits manuelle Tests durchgeführt, können diese einfach und schnell automatisiert werden, indem die Aktionen der Testschritte aufgezeichnet werden. Diese können dann immer wieder abgespielt und der Testablauf damit repliziert werden. Dies ist jedoch nur möglich, wenn das SUT sich jedes mal identisch verhält. Auch sind so erstellte Tests schwerer anzupassen und zu verändern, da sie meist in wenig verständlicher Form vorliegen [SL19, 287]. Die Lineare Skripterstellung ist eine Weiterentwicklung des Capture Replay Ansatzes. Hierbei wird aus der Aufzeichnung des Manuellen Tests automatisiert ein Formales Testscripte generiert. Der Vorteil gegenüber dem Capture Replay ist, dass das

Testscripte leicht lesbar und veränderbar ist [Bau21, 104]. Das datengetriebene Testen baut auf den vorherigen Strategien auf. Es kann in Kombination mit jeder davon verwendet werden. Hierbei werden statt konkreter Werte, Platzhalter in den Testscripte verwendet. Zusätzlich wird ein Bestand an Testdaten angelegt. Aus der Kombination von Testscripten und Testdaten lassen sich konkrete Testfälle generieren. Dadurch können viele Tests mit gleichem Aufbau, aber unterschiedlicher Datenbasis leicht realisiert werden [Bau21, 135]. Das Schlüsselwortgetriebene-Testen [iso16] stellt eine Weiterentwicklung des Datengetriebene-Testen dar. Dabei werden Aktionen mit dem SUT als Schlüsselworte abstrahiert. Bei der Definition von Testabläufen werden diese Schlüsselworte verwendet, um bestimmte Interaktionen über SUT Schnittstellen aufzurufen. Diese Schlüsselwirte werden vom TAS interpretiert und in konkrete Interaktionen mit dem SUT übersetzt [Bau21, 104]. Modellbasiertes Testen automatisiert nicht nur den Testprozess, sondern auch die Testerstellung. Dazu wird ein Modell des SUT definiert. Ein Testgenerator analysiert dieses und erstellt daraus automatisch Testscripts. Als Grundlage für das Modell können Beispielsweise die Dokumentation des SUT [ABT10], oder die vor der Entwicklung erfassten Anforderungen dienen. Der Testgenerator analysiert das Modell und generiert Eingaben und dazu passende Prüfparameter für die Ausgaben. Bei Search Based Software Testing (SBST) werden Optimierungsalgorithmen eingesetzt um z.B. Grenz-, oder Min,- Max-Werte zu finden. So kann es Search Based Software Testing (SBST) verwendet werden, um Eingaben zu finden, die zur maximalen oder minimalen Ausführungszeit eines Ablaufs führen können. Es kann helfen die zeitlichen Grenzen von Aktionen des SUT zu finden. Auch kann SBST verwendet werden, um nach Eingaben zu Suchen, die ein gewünschtes, oder auch nicht gewünschtes Ergebnis liefern. Die Breite der Abdeckung kann ebenfalls als Optimierungskriterium genommen werden, um Testdaten zu generieren, die mit wenigen Tests eine möglichst breite Abdeckung erzielen. Mit den richtigen Metriken, kann das SBST automatisch benötigte Tests generieren. Eine große Schwäche des SBST sind die Einschränkungen der Eingabedaten-Menge und -Typ, die generiert werden können. Der Fokus liegt auf einer fixen Anzahl an numerischen Inputs mit fixer Größe. Für komplexere Datentypen, wie Strings, oder tiefere Datenstrukturen, ist es sehr schwierig den Möglichkeitenraum sinnvoll zu durchsuchen [McM11]. Auch Interaktionen der Software mit externen Systemen lassen sich meist nur unzureichend abbilden. Hier können Techniken wie Mock helfen. Dabei werden Systeme, mit denen das SUT im Betrieb interagiert simuliert. Diese Simulation wird als Mock bezeichnet und stellt keine vollständige Nachbildung dar, sondern liefert nur die für den Test notwendigen Interaktionen und Daten [SABB17]. Doch auch hier stellt sich die Frage wie die Mock Daten erzeugt werden. Dafür können erneut Search Based Algo-

rithmen angewendet werden, doch auch hier gibt es Grenzen, was diese leisten können [McM11].

2.1.8 Zusammenfassung Testautomatisierung

Testautomatisierung hilft Risiken bei der Entwicklung eines Produkts zu minimieren und Fehler in diesem frühzeitig aufzuspüren. Anhand der Art der Schnittstellen, die das SUT bietet, Soft- oder Hardware basiert, muss ein passendes Testbett geschaffen werden. Dies können virtuelle Schnittstellen sein, bei denen mit dem SUT nur auf Softwareebene kommuniziert wird, oder Hardware Schnittstellen bei denen physisch mit dem SUT interagiert wird. Die Testabläufe werden in Testscripts beschrieben. Diese können manuell, aber auch teil-, oder vollautomatisiert erzeugt werden. Verfasst werden sie als direkt ausführbarer Quellcode, oder in einer anderen Beschreibungssprache, wie XML, JSON, oder die speziell hierfür definierte ATML.

2.2 Requirements Engineering

Requirement Engineering (RE) ist das disziplinierte und systematische Vorgehen bei der Erfassung und Verwaltung von Anforderungen mit dem Ziel die Wünsche und Bedürfnisse der beteiligten Stakeholder*innen zu verstehen und die Risiken zu minimieren, ein System abzuliefern, dass diese nicht erfüllt [GvSB22]. Techniken des Requirements Engineering helfen das Risiko einer Fehlentwicklung, bei der Entwicklung eines Software Systems zu reduzieren, indem die Anforderungen, Umgebung und Rahmenbedingungen an das System im Vorhinein bestimmt und klar definiert und beschrieben werden. Auch helfen sie bei der Entwicklung des Systemarchitektur, indem sie Vorgaben und Grundlagen für das Design dieser liefern [Zör12, 18].

2.2.1 Kontextanalyse

Der Requirements-Engineering-Prozess der von Pohl und Rupp [PR21] beschrieben wird beginnt, mit der *Kontextanalyse*. Dabei wird die Umgebung analysiert, in der das zu entwickelnde System eingesetzt werden soll. Systeme können nie isoliert betrachtet werden. Sie sind immer in einem Kontext eingebettet, der Anforderungen vorgibt. Es wird hierbei zwischen fachlichen und technischen

Kontext unterschieden. Diese könne im Zuge des Prozesses separat, oder zusammen betrachtet und erfasst werden. Die Kontextanalyse dient dazu alle für das System wichtigen materiellen und immateriellen Aspekte zu Identifizieren [PR21, 42]. Wichtig ist hier das Festlegen der Systemgrenze. Diese legt fest, welche Aspekte während der Entwicklung verändert, oder gar komplett verworfen werden können, und auf welche kein Einfluss genommen werden kann. Darüber hinaus gibt es noch die *Kontextgrenze*, die Aspekte, die für die Entwicklung relevant sind, vom Rest der Welt abtrennt. Aspekte außerhalb dieser Grenze sind für das RE nicht relevant und dürfen nicht betrachtet werden. Fällt später auf, dass ein Aspekt dennoch benötigt wird, muss die Kontextgrenze angepasst werden. Ist für einen Aspekt zum Zeitpunkt der Kontextanalyse noch nicht sicher, ob dieser im Systemkontext, oder außerhalb liegt, wird dieser in der sog. *Grauzone* verortet. Stellen diese Aspekte Anforderungen an das zu entwickelnde System, werden sie als Stakeholder*innen bezeichnet. Stakeholder*innen können natürliche, oder juristische Personen sein, die ein Interesse am System haben. Übliche Stakeholder Gruppen sind hierbei [PR21, 21]:

- Benutzer die das System bedienen werden
- Kunden, an die das System ausgeliefert werden
- Auftraggeber, der die Entwicklung bezahlt
- Betreiber, der das System administrieren wird
- Regulierungsbehörden, die Normen, Standards und Rechtliche Vorgaben machen
- (Benutzer eines eventuellen legacy Systems, das durch die neu Entwicklung ersetzt wird)

Es kann, je nach Art des Projekts, eine Vielzahl weiterer Stakeholder Gruppen geben. Weiter werden technische und organisatorische Randbedingungen erfasst. Technische Randbedingungen sind Vorgaben und Einschränkungen die aus technischer, fachlicher Richtung kommen. Sie machen Vorgaben für Server- und Client-Umgebungen, wie Betriebssystem, minimale Hardwareanforderungen, Kompatibilität mit anderen Systemen. Einschränkungen für Wartungsfenster, oder Vorgaben für die erwartete Nutzerzahl. Organisatorische Randbedingungen sind Faktoren wie Entwickler Team Größe, geplanter Zeitrahmen für das Projekt, Vorkenntnisse der Entwickler, oder Vorgaben für den Einsatz von externen Bibliotheken und Frameworks. Der Kontext kann Schriftlich, oder auch Grafisch beschrieben werden. Hierfür gibt es verschiedene Diagramme. **Kontextdiagramm** Kontextdiagramm visualisieren mit welchen Aspekten das System interagiert. Dies hilft den Systemkontext einzugrenzen und zu beschreiben. Ein Kontextdiagramm kann den Technischen Kontext, den Fachlichen Kontext, oder beide beschreiben [SH23, 14]. **Datenflussdiagramm** Datenflussdiagramme be-

schreiben den Fluss von Daten zwischen Aktoren, hier Terminatoren, und Prozessen eines Systems. Sie helfen die System- und Kontextgrenze zu identifizieren und zu beschreiben [PR21, 87]. **Use Case Diagramm** Use-Case Diagramme visualisieren wie Aspekte, hier Aktoren, im System Kontext mit dem System interagieren [Obj17, 637]. Use Cases sind Verhaltensweise, zu denen das System, in der Interaktion mit Aktoren, in der Lage sein muss.

2.2.2 Anforderungsermittlung

Die in der Kontextanalyse ermittelten Stakeholder*innen, sowie der erfassten Randbedingungen geben Anforderungen an das System vor. Es gibt verschiedene Arten von Anforderungen die von Stakeholder*innen und anderen Aspekten des Systemkontexts explizit, oder implizit vorgegeben werden. Das International Requirements Engineering Board e.V. (IREB) [GvSB22, 10] unterscheidet hier zwischen:

- Funktionale Anforderungen: Beschreibt gewünschtes Ergebnis oder Verhalten, das durch Funktionalitäten des Systems erreicht werden soll. Eine verbreitete Möglichkeit diese zu dokumentieren sind Beschreibung einer Eigenschaft, oder Funktionalität, die ein Nutzer von einem System fordert. Jede Story beschreibt ein Verhalten, das die Stakeholder*in erwartet, oder eine Aktion, die diese ausführen möchte [GvSB22, 31]
- Qualitätsanforderungen: Beschreibt Qualitätsaspekte, die nicht direkt durch funktionale Anforderungen abgedeckt werden. Diese werden in Form konkreter Szenarien beschrieben [GvSB22, 34]. Dabei kann die [ISO11] als Leitfaden dienen.
- Constraints (Randbedingungen): Beschränkungen der möglichen Lösungen für die Anforderungen durch Vorgaben durch äußere Faktoren, wie Rechtliche-, oder Technische-Vorgaben [GvSB22, 36].

Klaus Pohl und Chris Rupp [PR21, 125] gliedern die funktionalen Anforderungen nochmal nach der Priorität für die Zufriedenstellung der Stakeholder:

- Basisfaktoren: werden von den Stakeholder*innen als selbstverständlich vorausgesetzt. Nicht erfüllen führt zu massiver Unzufriedenheit.
- Leistungsfaktoren: werden von den Stakeholder*innen explizit gefordert. Das Produkt erfüllt ohne sie dennoch einen Großteil der Gewünschten Funktion, ihr Fehlen kann jedoch zu Unzufriedenheit bei den Stakeholder*innen führen.
- Begeisterungsfaktoren: sind den Stakeholder*innen nicht bekannt, oder bewusst. Führen bei Umsetzung zu angenehmer Überraschung.

Dabei berufen sie sich auf das Kano Modell nach [SBMH96]. Das beschreibt, wie Kundenwünsche kategorisiert und priorisiert werden können. Für die Erfassung der Anforderungen von verschiedenen Stakeholder*innen Arten, gibt es verschiedene Techniken. Hierbei kann zwischen Kreativitäts- und Entwurfstechniken unterschieden werden [GvSB22, 94].

Das Ziel von Kreativitätstechniken ist es, dem Stakeholdern noch nicht bewusste, oder als unrealistisch eingeschätzte Anforderungen zu erarbeiten. Dabei wird eine Umgebung geschaffen, in der die Teilnehmer Zeit haben, um Ideen zu entwickeln und frei zu äußern. Es ist wichtig, dass Ideen nicht direkt positiv, oder negativ gewertet werde, um die Teilnehmer*innen nicht zu demotivieren [GvSB22, 94].

Erhebungstechniken dienen dazu bereits vorhandenes Know-How und bekannte Bedürfnisse zu erfassen und zu dokumentieren. Dabei können sowohl Stakeholder, als auch Artefakte, wie Vorgänger Systeme, oder gesammeltes Nutzer Feedback, als Quelle dienen. Anforderungen können direkt von Stakeholdern gesammelt werden. Dazu dienen Befragungstechniken, wie Interviews, oder Fragebögen. Weiter können Feldbeobachtungen helfen, bei denen aktuelle Prozesse, sowie der Umgang der Stakeholder mit möglichen Altsystemen erfasst und dokumentiert wird [PR21, 127]. Existiert bereits ein Altsystem, oder wurde anderweitig Vorarbeit geleistet, können diese ausgewertet werden. Vorarbeit kann zum Beispiel die Form von gesammeltem Nutzer Feedback haben.

Entwurfstechniken dienen der Ausarbeitung und Erkundung von Ideen, die im Zuge von Kreativitätstechniken erarbeitet wurden. Dabei werden vage Ideen und Bedürfnisse definiert und konkretisiert. Es wird mit visuellen, oder physischen Artefakten gearbeitet, um Ideen leichter, konkreter Kommunizieren zu können [GvSB22, 95]. Klaus Pohl und Chris Rupp [PR21, 127] teilen die Erfassungstechniken in Erhebungstechniken und Entwurfs- und Ideenfindungstechniken. Weiter listen sie den Prozess des Design Thinkings als mögliche Technik. Als Kreativitätstechniken listeten sie das Brainstorming, aber auch Perspektiven Techniken, bei denen die Stakeholder verschiedene Rollen einnehmen, um andere Sichtweisen auf Probleme zu erlangen [PR21, 133].

Die Auswahl der Techniken für den Prozess hängt vom Umfeld, der Verfügbarkeit der verschiedenen Stakeholder*innen und der bereits vorhandenen Dokumentation ab [PR21, 127].

Tabelle 2.2: Konfliktarten

Sachkonflikt	es gibt einen Informationsmangel zwischen Stakeholder*innen vor, oder einige haben falsche Informationen
Datenkonflikt	Stakeholder*innen haben ein unterschiedliches Verständnis oder Bedeutung von Bezeichnern und Begriffen
Interessenkonflikt	die Zielvorgaben verschiedener Stakeholder*innen stehen im Wiederspruch
Wertekonflikt	unterschiedliche Bewertung von Sachverhalten unterschiedlicher Stakeholder, aufgrund verschiedener Hintergründe
Beziehungskonflikt	negative Beziehung oder prädisposition von Stakeholder*innen untereinander
Struktureller Konflikt	Geringschätzung, oder Ablehnung von Anforderungen anderer Stakeholder*innen, aufgrund unterschiedlicher Macht- und Autoritätspositionen

2.2.3 Anforderungsabstimmung und Konfliktlösung

Verschiedene Anforderungen aus verschiedenen, oder sogar derselben Quelle, können im Widerspruch zueinander stehen. In diesem Fall muss mitigiert werden um konsistente Anforderungen zu erhalten, die von allen Stakeholder*innen akzeptiert werden. In [PR21, 137] werden zwischen verschiedenen Konflikttypen unterschieden, siehe Tabelle 2.3. Gefundene Konflikte zwischen Anforderungen müssen dokumentiert werden, bevor an einer Lösung dieser gearbeitet wird [PR21, 137].

2.2.4 Anforderungsvalidierung

Dokumentierte Anforderungen müssen auf ihre Qualität und Korrektheit geprüft werden, bevor sie für die Verwendung im weiteren Entwicklungsprozess

freigegeben werden können. Hierbei sollten die beteiligten Stakeholder*innen einbezogen werden, da deren Wünsche und Bedürfnisse die Basis für die Anforderungen bilden. Dabei sollte die Fehlersuche und die Identifikation von der Fehlerkorrektur getrennt werden.

Mögliche Techniken für die Validierung sind laut [PR21, 150]:

- Walkthrough: Hier gibt es eine Moderator*in, der die Anforderungen vorträgt. Die Autor*in dieser kann dann noch um Zusatzinformationen ergänzen. Eine Gruppe Reviewer*innen kommentiert diese im Bezug auf mögliche Qualitätsmängel. Eine Protokollant*in hält diese Anmerkungen fest.
- Inspektion: Eine Inspektion ist ein strikter strukturiertes Vorgehen das auf eine feste Abfolge definierter Phasen setzt. Hier werden feste Rollen vergeben und die Anforderungen gemeinsam strukturiert durchgegangen. Dann werden einzeln, oder in Gruppen, Fehler gesucht und gesammelt. Zum Schluss werden diese gemeinsam erläutert und dokumentiert.

2.2.5 Ergebnis Validierung

Nach der Entwicklung der Software muss evaluiert werden, ob diese die Anforderungen der Stakeholder*innen erfüllt. Ein Werkzeug sind hier die erfassten Anforderungen. Für die Beschreibung einer Eigenschaft, oder Funktionalität, die ein Nutzer von einem System fordert können hierfür bereits bei der Erfassung Akzeptanzkriterien festgelegt werden, anhand derer geprüft werden kann, ob die User Story erfüllt wird [PR21, 76]. Ein Akzeptanz Kriterium ist die Beschreibung einer Teileigenschaft, Teilverhalten des Systems, oder einer Teilaktion, die die Nutzer*in damit durchführen kann, um die User Story zu erfüllen.

Für die Erfassung von Anforderungen gibt es nicht den einen, einheitlichen Prozess. Es existieren Frameworks, die helfen einen geeigneten Prozess zu konstruieren. Im Zug dieser Arbeit wurde der Requirements-Engineering-Prozess von Pohl und Rupp [PR21] gewählt. Die erfassten und dokumentierten Anforderungen dienen als Grundlage für die weitere Entwicklung der Software. Dabei ist zu beachten, dass diese Anforderungen niemals endgültig sind. Sollten während der Entwicklung, oder beim Einsatz der Software neue Anforderungen aufkommen, sind diese ebenso zu erfassen und zu dokumentieren.

2.3 Softwarearchitektur

Ziele der Softwarearchitektur ist es, nach [GKRS20, 3] Systeme zu designen, die **Nützlich** sind. Sie erfüllen die erwarteten Anforderungen. **Fest** sind. Sie erfüllen die erwarteten Qualitäts Merkmale. **Schön** sind. Sie sind sowohl für die Entwickler*in, als auch die Benutzer*in logische aufgebaut und leicht zu verstehen.

Definition Softwarearchitektur

Jede Software besitzt eine Architektur. Sie besteht aus den Architekturentscheidung (AE), die während der Entwicklung der Software getroffen wurden. Stefan Zörner et al. definiert Architektur Entscheidungen in [Zör12, 32] als ″Fundamentale Entscheidung, die im weiteren Verlauf nur schwer zurücknehmen ist″. Sie beschreiben und begründen, wie Probleme gelöst werden und warum diese Lösung gewählt wurde.

Die Software Architektur gibt die Struktur eines Softwaresystems vor. Sie definiert wichtige Komponenten und ihre Funktion, sowie ihr Verhältnis zueinander [Zör12, 20]. Szyperski et al. [SGM09, 41] definieren Software Komponenten als ″[...] Element des Zusammenbaus mit vertraglich festgelegten Schnittstellen und ausschließlich expliziten Kontextabhänigkeiten. Eine Softwarekomponente kann unabhängig ausgeliefert werden und ist Baustein für dritte.″

Architekturentscheidung (AE)

Die bereits genannten Architekturentscheidung (AE) sollten dokumentiert werden, damit diese während der Entwicklung und auch danach leicht Verständlich sind [Zör12] S.64. Diese Dokumentation sollte auch die Begründung für die Entscheidung enthalten, damit diese später leicht nachzuvollziehen ist. Strukturiert werden kann diese Dokumentation in

- Fragestellung: Zu welchem Problem soll eine Lösung gefunden werden
- Relevante Einflussfaktoren: Welche User Stories, Qualitäts Merkmale, Randbedingungen beeinflussen die Entscheidung
- (Annahmen): Welche weiteren Faktoren beeinflussen möglicherweise die Entscheidung
- Betrachtete Alternativen: Betrachtete Lösungsansätze, zwischen denen Entschieden wurde. Kurze Abwägung der Vor- und Nachteile der verschiedenen Ansätze

- Entscheidung: Welche der betrachteten Lösungsansätze wurde gewählt und warum.

Unified Modeling Language

Um Konzepte und Ideen visuell darzustellen können Diagramme und Grafiken helfen. Die Unified Modeling Language (UML) der Object Management Group ist Grafische Modellierungssprache für die Visualisierung, Spezifizierung, Konstruktion und Dokumentation von Artefakten verteilter Objekt Systeme [Obj17]. Sie bietet Standards für verschiedene Diagrammtypen.

Bausteine, Schnittstellen und Konfiguration

Im Zuge der Erstellung einer Softwarearchitektur zerlegt man das geplante System in kleinere Einheiten [GKRS20, 22]. Diese bezeichnet man als Bausteine und Schnittstellen. Dabei ist kein Grad der Abstraktion vorgeschrieben. Bausteine können konkrete Klassen, Bibliotheken, aber auch abstraktere Gliederungen darstellen, die in der Software keine direkte Entsprechung haben [GKRS20, 22]. Schnittstellen beschreiben Wege, über die mit Bausteinen kommuniziert werden kann. Sowohl von außerhalb, als auch innerhalb des Systems [GKRS20, 22]. Dies hilft es die geplante Aufteilung, aber auch bestimmte Konzepte und Ideen leichter verständlich zu machen. Bausteine stellen Komponenten, oder Teile des Systems dar. Werkzeuge für die strukturierte Dokumentation bietet die Unified Modeling Language (UML) [Obj17, 209] in Form des Komponentendiagramms.

2.3.1 Diagramme und Sichten

Ein UML Klassen Diagramm ist konkreter als ein Komponenten-Diagramm. Es beschreibt nicht nur eine abstrakte Gliederung, sondern Klassen und deren konkrete Schnittstellen und Beziehungen. Diese können weiterhin einen gewissen Abstraktionsgrad enthalten, können jedoch auch konkrete Klassen, deren Methoden und Felder beschreiben [Obj17, 209]. Bausteine, die im Komponenten Diagramm beschrieben wurden, können aus einer, oder mehreren Klassen bestehen. Klassen können auch zu mehreren Bausteinen gehören. Das Klassendiagramm kann helfen funktionale Komponenten, sowie deren Verhältnis zueinander zu beschreiben. Konkrete Abläufe, die fundamental für die Funktion

des Systems sind, können in Form von Aktivitätsdiagrammen dokumentiert werden. Dazu werden die Kommunikationspfade und Abläufe innerhalb des geplanten Systems dokumentiert. Dies kann auf der Bausteinebene, aber auch bereits der Klassenebene geschehen [SH23, 22]. Klassendiagramme dienen wie das Komponentendiagramm der Darstellung der Struktur einer Software. Doch ist dieses konkrete und näher an der eigentlichen Implementierung. So werden im Klassendiagramm die Schnittstellen zwischen Klassen, in Form von Attributen und Methoden beschrieben. Auch das Verhältnis der Klassen in Form von Assoziation, Vererbung, Aggregation, oder Komposition beschrieben [Obj17, 691]. Für die Dokumentation von Prozessen und Abläufen bietet die UML das Sequenz Diagramm [Obj17, 595]. Dieser Diagramm Typ legt den Fokus die Interaktion verschiedener Objekte durch den Austausch von Nachrichten. Das Use Case Diagramm [Obj17, 641] der UML dient der Visualiserung der verschiedenen Anwendungsfälle eines Systems, sowie deren Beziehung zu externen Aspekten. Es beschreibt, welche Aktionen verschiedene Benutzergruppen im System Auslösen. Es kann beschrieben werden aus welchen Unteraktionen diese bestehen, welche Artefakte daraus erzeugt werden und mit welchen weiteren externen Aspekten dabei kommuniziert wird.

2.3.2 Ergebnis

Software Architektur bildet sich aus den Entscheidungen, die beim Entwurf einer Software getroffen wurden. Um diese Entscheidungen nachvollziehbar zu machen und zu dokumentieren, stellt das UML verschiedene Werkzeuge bereit. Der Aufbau einer Software kann in Form von Komponentendiagrammen, oder Klassendiagrammen festgehalten werden. Für die Beschreibung von Abläufen innerhalb der Software bietet die UML Aktivitäsdiagramme, oder Sequenzdiagramme. Um Anwendungsfälle eines System zu beschreiben bietet die UML das Use Case Diagramm.

2.4 Grundlagen Software Entwurfsmuster

Da im Zuge dieser Arbeit ein Softwaresystem entworfen und implementiert werden soll, betrachtet dieser Abschnitt Entwurfsmuster, die im Softwareentwurf eingesetzt werden. Entwurfsmuster sind wiederkehrende Lösungsmuster. Sie stellen bewerte Techniken und Lösungsansätze dar, auf die ein Entwickler während dem Entwurf einer Software zurück greifen kann [GHJV94]. Ein Entwurfsmuster setzt sich aus vier Teilen zusammen, siehe Tabelle 2.5.

Tabelle 2.4: Teile eine Entwurfsmusters

Pattern-Name	Dient der Referenzierung des Problems, der Lösung und der daraus folgenden Auswirkungen
Problemstellung	Beschreibt die Umstände in denen das Entwurfsmuster angewendet werden kann
Lösung	Eine Schablone in der ein mögliches Vorgehen zur Lösung des Problems beschrieben wird
Konsequenzen	Mögliche Auswirkungen auf die Software, die die Anwendung des Entwurfsmusters nach sich zieht

2.4.1 Trennung von Oberfläche und Business Logik

Für bessere Übersicht und Modularität kann es Sinnvoll sein, die Darstellung und Business Logik einer Software von einander zu trennen. Business Logik beschreibt wie Daten erstellt, verarbeitet und gespeichert werden. Im Kontrast beschreibt die Darstellung wie diese Daten und Bearbeitungsmöglichkeiten dem Nutzer präsentiert werden [Lit21, 101]. **Name:** Model View Viewmodel (MVVM) **Problemstellung:** Oberfläche und Geschäftslogik sind fest verzahnt, sollen jedoch unabhängig ausgetauscht werden. **Lösung:** Das Model enthält alle Daten

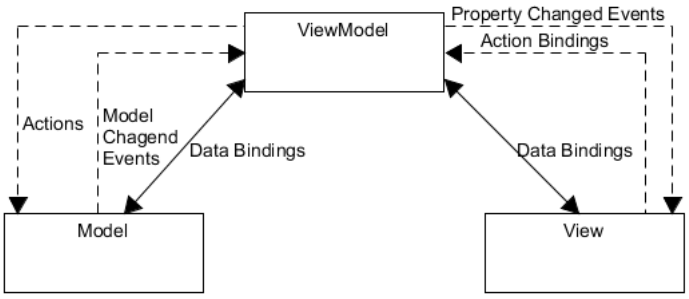


Abbildung 2.4: MVVM Muster

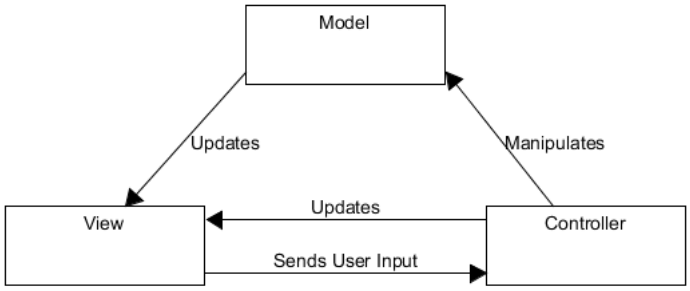


Abbildung 2.5: MVC Muster

und Business Logik. Die View enthält alle Bedienelemente und möglichst keine Logik. Das ViewModel verbindet Model und View. Die Verbindung der View und des Models mit dem ViewModel geschieht über Bindings, zu sehen in Abbildung 2.4. Wobei die Verbindungen nur locker definiert werden und erst beim Kompilieren, oder zur Laufzeit erzeugt werden [Smi09] **Konsequenz:** View und Model können leicht ausgetauscht werden. Es können mehrere Views implementiert werden, die die Daten des selben Models visualisieren. Doch entsteht ein Mehraufwand, durch die Notwendigkeit das ViewModel implementieren zu müssen, das weder User Interface, noch Business Code enthält [Lit21, 101].

Name: Model View Controller (MVC) **Problemstellung:** Oberfläche und Geschäftslogik sind fest verzahnt, verschiedene Views sollen jedoch auf die selben Datenbasis zugreifen könne. Außerdem soll die Datenbasis von der UI und der Logik gekapselt werden. **Lösung:** Model View Controller (MVC) teilt in Model, das die Daten enthält, View, die die UI darstellt und Controller, der die Business Logik enthält, zu sehen in Abbildung 2.5. Hier werden Daten und Business Logik getrennt. Dies ermöglicht zum Beispiel das einfache Speichern des aktuellen Zustands, da dieser in gekapselt von der Logik vorliegt. Änderungen der Daten im Model werden direkt an die View weiter gegeben. Nutzereingaben werden von der View jedoch zuerst an den Controller weiter gegeben, der diese verarbeitet und wenn nötig das Model aktualisiert [QS13]. **Konsequenz:** Die Datenbasis liegt gekapselt vor und kann einfach bearbeitet, gespeichert, oder ausgetauscht werden [AFS22].

Name: Model View Presenter (MVP) **Problemstellung:** Business Logik und UI sollen ausgelagert werden, um beide separat Testen zu können. **Lösung:** Model View Presenter (MVP) Teilt die Anwendung in das Model, das die Business Logik und die Daten enthält, die View, die UI bereitstellt und den Presenter, der die Kommunikation zwischen beiden Komponenten steuert. Die dadurch ent-

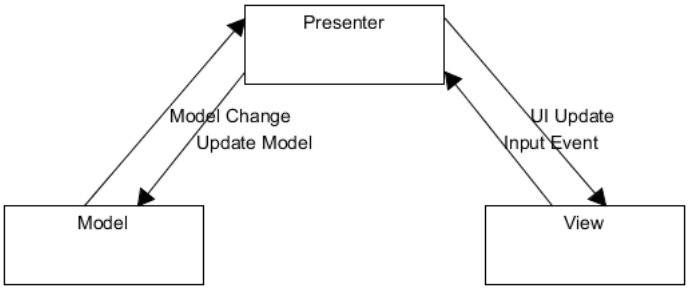


Abbildung 2.6: MVP Muster

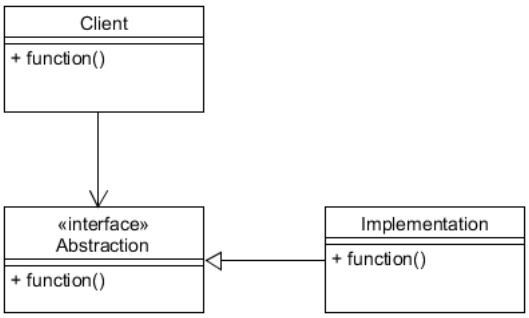


Abbildung 2.7: Bridge Muster

stehende Struktur ist in Abbildung 2.6 zu sehen. **Konsequenzen:** Sowohl View, als auch Model können für Testzwecke durch ein Mock ausgetauscht werden [QS13].

2.4.2 Das Bridge Muster

Name: Bridge Pattern **Problemstellung:** Auf eine Klasse soll über eine fest definierte Schnittstelle zugegriffen werden. Diese Schnittstelle soll vor der Logik definiert werden. Es soll möglich sein, mehrere Logiken für die selbe Schnittstelle zu implementieren. [GHJV94, 200] **Lösung:** Es wird eine Abstraktion, in Form einer Schnittstelle implementiert. Diese enthält keine Logik, sondern beschreibt nur, welche Methoden es gibt. Die eigentliche Programm Logik wird in einer Klasse definiert, die die Schnittstelle implementiert 2.7 [GHJV94, 200]. **Konsequenzen:**

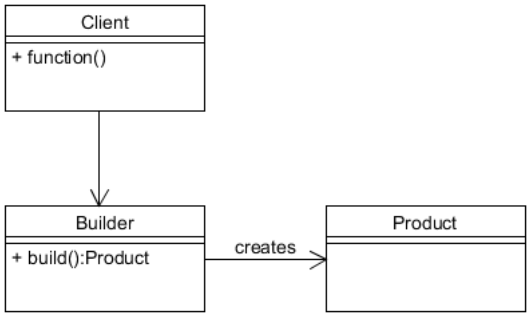


Abbildung 2.8: Builder Muster

2.4.3 Das Builder Muster

Name: Builder Pattern **Problemstellung:** Um ein Objekt einer Klasse nach der Erzeugung verwenden zu können ist ein Initialisierungsprozess nötig. Das Objekt muss Konfiguriert und vielleicht noch weitere Objekte anderer Klassen erzeugt werden, von denen das Objekt abhängig ist. **Lösung:** Das Builder Pattern Pattern beschreibt die Implementierung einer *Builder* Klasse, die Objekte *Product* einer anderen Klasse erzeugt, zu sehen in Abbildung 2.8. Dazu stellt die *Builder* Klasse Methoden zur Verfügung, die andere Klassen instantiieren und für die Verwendung vorbereiten [GHJV94, 139]. **Konsequenzen:** Den Entwickler*innen muss bewusst gemacht werden, dass, um Objekte der *Product* Klasse zu erzeugen, der *Builder* verwendet werden muss.

2.4.4 Das Strategy Muster

Name: Strategy **Problemstellung:** Es existieren verschiedene Lösungen für ein Problem. Welche angewendet wird wird zu Laufzeit bestimmt. **Lösung:** Es werden verschiedene Algorithmen für die Lösung eines Problems abstrahiert und über eine einheitliche Schnittstelle zugänglich gemacht, visualisiert in Abbildung 2.9 [GHJV94, 383]. **Konsequenzen:** Es muss eine Einheitliche Schnittstelle entworfen werden, die das Problem vollständig beschreibt. Sollten Änderungen an dieser Vorgenommen werden, müssen alle Algorithmen angepasst werden.

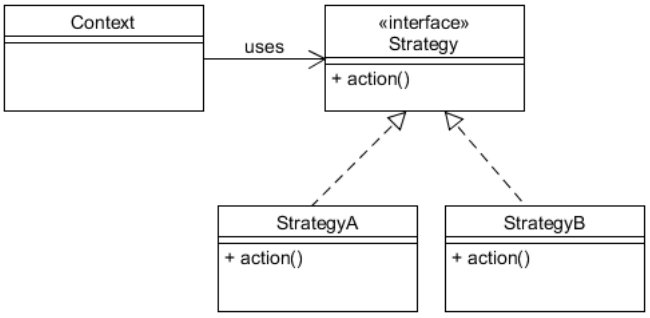


Abbildung 2.9: Strategy Muster

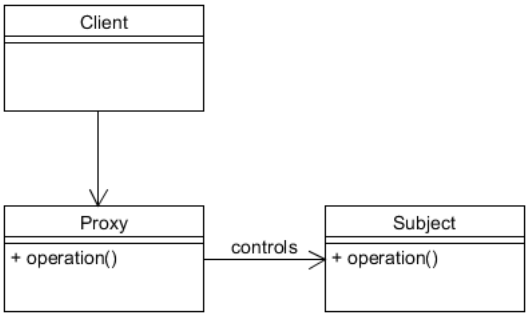


Abbildung 2.10: Remote Proxy Muster

2.4.5 Das Remote Proxy Muster

Name: Remote Proxy Pattern **Problemstellung:** Es existiert eine Ressource, die nicht Teil des Programms ist, jedoch im Programm verwendet werden soll. **Lösung:** Es wird eine Klasse Implementiert, die Methoden für den Zugriff auf die Ressource bereitstellt 2.10. Sie bietet Methoden für den Zugriff auf Informationen der Ressource und die Manipulation dieser [GHJV94, 264]. **Konsequenzen:** Alle Zugriff auf die abstrahierte Ressource müssen über den Proxy erfolgen.

Die beschrieben Entwurfsmuster helfen dabei bestimmte Probleme elegant zu lösen. Sie werden beim Entwurf der Software in Kapitel 4 eingesetzt.

3 Konzepte

In diesem Kapitel werden spezifische theoretische Aspekte und Vorarbeit die vor der Durchführung des praktischen Teils der Bachelor Thesis durchgeführt wurde beschrieben. Es beschreibt den entwickelten Requirement Engineering (RE) Prozess, der für die Erfassung der Anforderungen an das Projekt angewandt wurde. Weiter werden verschiedene Architekturansätze für TAS Systeme beschrieben.

3.1 Requirements Engineering Prozess

In dieser Thesis wird ein Requirement Engineering (RE) Prozess nach Klaus Pohl und Chris Rupp [PR21, 176] verfolgt, die sich auf Standards des International Requirements Engineering Board e.V. (IREB) berufen. Dabei wurde zusätzlich Stefan Zörner [Zör12] zugezogen. Der verwendete Prozess wird in Abbildung 3.1 gezeigt.

3.1.1 Rahmenbedingungen für den Prozess

Die Entwicklung des Projektes ist zwar aufgrund der Zeitbeschränkung sehr linear, da die weitere Entwicklung des Projektes später in einem agilen Kontext stattfinden wird, wurde sich für einen gemischten Ansatz zwischen linearer und interaktiver Entwicklung entschieden. Die Entwicklung des ersten Prototyps ist dabei linear, die Artefakte und die Erfassung dieses werden jedoch unter dem Gesichtspunkt einer späteren agilen Weiterentwicklung gewählt. Die Ermittlung und Abstimmung der Anforderungen soll nach einem linearen Ansatz geschehen, mit festem Ablauf, ohne Revisionen [PR21, 170].

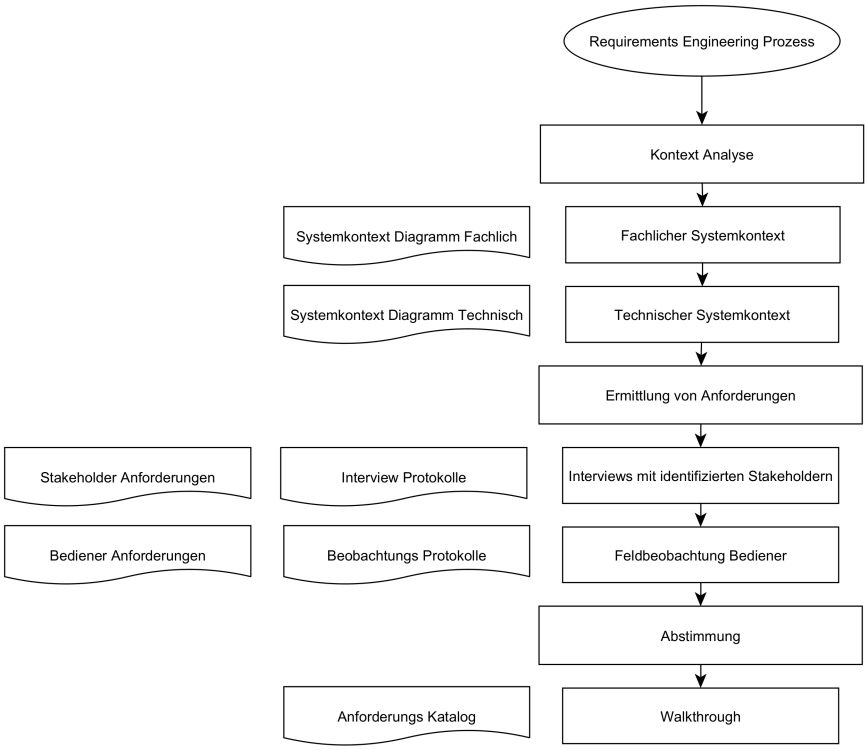


Abbildung 3.1: Geplanter RE Prozess

3.1.2 Der gewählte Prozess

Der RE Prozess beginnt mit der Kontext Analyse, wobei entschieden wurde, den fachlichen und technischen Systemkontext separat zu erfassen und zu dokumentieren. Für die Dokumentation wurden Schriftliche Beschreibungen gewählt, die um Kontext-, Use-Case- und Datenfluss-Diagramme ergänzt werden [Obj17].

Um die Anforderungen der Stakeholder*innen zu erfassen, die während der Kontextanalyse identifiziert wurden, werden Interviews mit diesen geführt und die Ergebnisse dieser in Form von User Stories erfasst [GvSB22, 38]. Weiter werden Interviews mit den verantwortlichen Abteilungsleitern geführt. Dadurch wird eine möglichst vollständige Sammlung der bewussten Anforderungen aller beteiligten Stakeholder*innen erstellt. Es wurde sich für Interviews entschieden, da die verschiedenen, identifizierten Stakeholder*innen Gruppen einzeln befragt werden müssen, um den betrieblichen Ablauf nicht zu sehr zu stören. Dadurch kann die zeitliche Inanspruchnahme der einzelnen Beteiligten minimiert werden.

Zusätzlich wurde der aktuelle Arbeitsprozess analysiert, indem eine Feldbeobachtung im laufenden Betrieb durchgeführt wurde. Dabei wurden weitere Anforderungen der Bediener*innen und des Kontexts erfasst, die bei den Interviews noch nicht genannt wurden, da sie den Beteiligten nicht bewusst waren. Es wurde sich entschieden, die Feldbeobachtung nach den Interviews durchzuführen, damit beim Beobachter bereits ein ausreichendes Wissen vorliegt, um den Prozess verstehen zu können und mögliche Abweichungen von den Beschreibungen im Interview feststellen zu können.

Die Funktionalen Anforderungen wurden in Form von User Stories dokumentiert. Anhand der Aussagen der Nutzer und der Beobachtungen wurden sie nach [PR21, 125] in Basis-, Leistungs- und Begeisterungsfaktoren eingeteilt. Weiter wurden Qualitätsmerkmale [Zör12, 48] erfasst, die geforderte nicht funktionale Anforderungen beschreiben.

Für die Anforderungsvalidierung wurde der Walkthrough gewählt. Dieser erfordert wenig vorgegeben Rollen und kann von einer Person moderiert und geleitet werden. Es wurden Vertreter aller relevanten Stakeholder Gruppen eingeladen. In der Vorbereitung wurden für die Erarbeiteten User Stories, wo Möglich, anhand der Ergebnisse der Interviews und der Feldbeobachtung Vorschläge für Akzeptanzkriterien notiert, die während des Walkthroughs besprochen, ergänzt, oder auch verworfen werden können. Das Ziel des Walkthroughs ist

Tabelle 3.1: User Story Vorlage

ID	Titel	Stakeholder	Typ
User Story	Beschreibung nach Satzschablone		
Akzeptanzkriterien	Liste der Akzeptanzkriterien		

es ein konsistentes Set an Anforderungen zu finden, das von allen Stakeholdern akzeptiert wird und alle wichtigen Wünsche abdeckt. Dann wurden alle befragten Stakeholder eingeladen und die erfassten Anforderungen mit diesen durchgesprochen. Anmerkungen und Vorschläge wurden dokumentiert und die Anforderungen entsprechen überarbeitet.

Für die Erfassung von User Stories wurde eine vereinfachte Variante der Vorlage aus [Bau21, 76] gewählt. Unsere Vorlage enthält eine eindeutige Id, einen Name, die Stakeholder, deren Anforderungen diese User Story erfüllt, eine Beschreibung nach der Wortschablone aus [PR21, 72] und Akzeptanzkriterien [PR21, 76] anhand derer geprüft werden kann, ob die User Story erfüllt wurde. Es wurde sich entschieden Vorbedingungen und auslösendes Ereignis aus der Vorlage zu entfernen, da diese Punkte bereits in der Beschreibung nach Satzschablone aus [Bau21, 72] enthalten sind. Aus diesen Vorgaben wurde eine Vorlage 3.1 erstellt.

3.1.3 Ergebnis

Basierend auf den vorliegenden Rahmenbedingungen wurde ein linearer Requirement Engineering (RE) Prozess gewählt. Die einzelnen Schritte wurden, basierend auf der Verfügbarkeit der jeweils notwendigen Ressourcen, sowie zeitlicher Einschränkungen gewählt. Eine Vorlage für die Erfassung der User Stories wurde erstellt.

3.2 TAS Architekturen

Eine TAA ist eine Softwarearchitektur für ein TAS. Sie beschreibt den Aufbau und zu implementierende Abläufe für ein TAS. Es gibt in der Literatur verschiedene Architekturen, die als Grundlage für die Entwicklung einer TAA verwendet werden können.

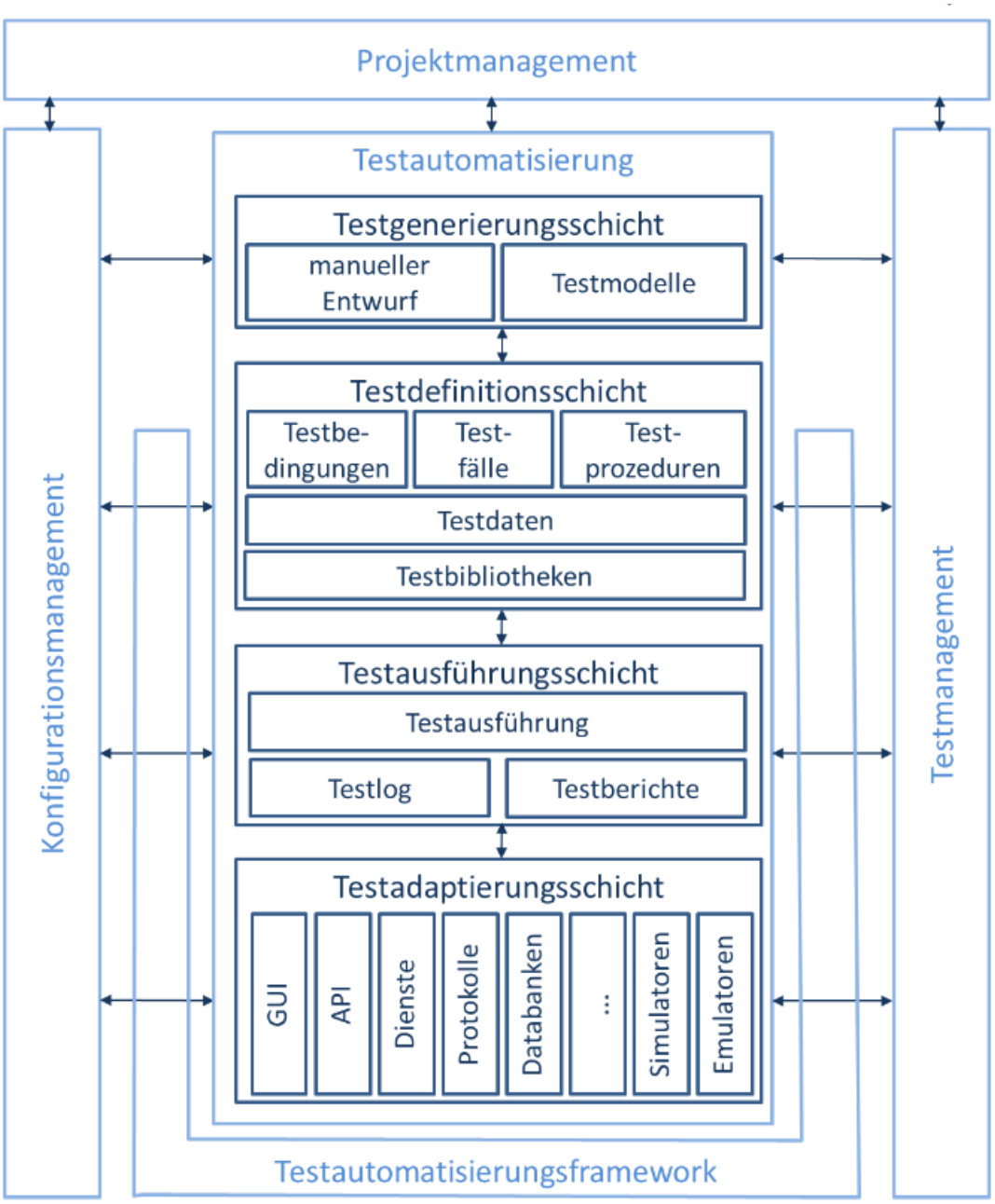


Abbildung 3.2: Generische Testautomatisierungs Architektur [PBB+, 26]

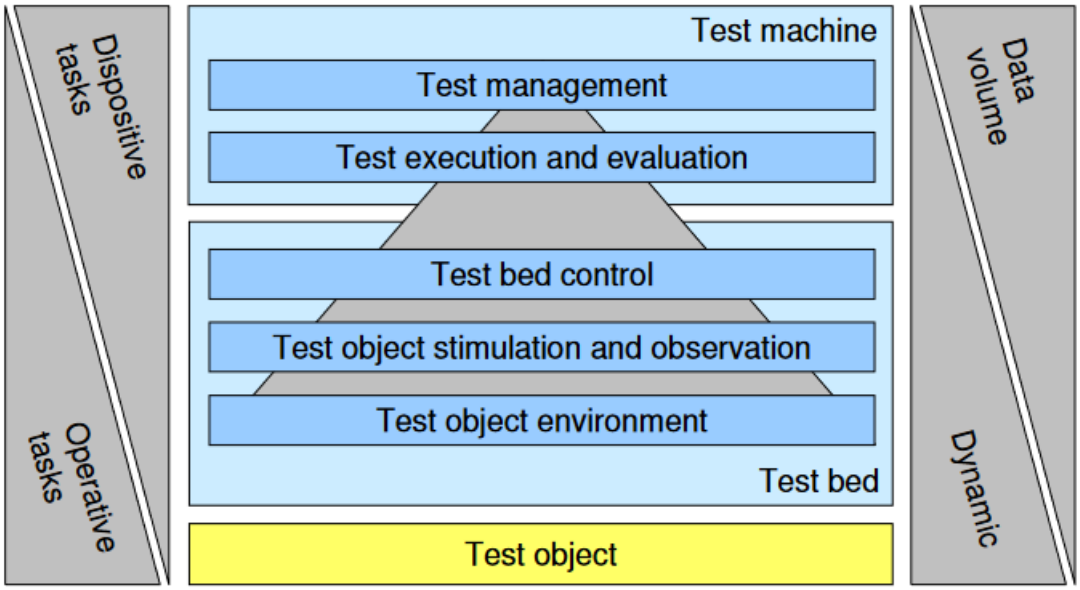


Abbildung 3.3: Universelle Testautomatisierungs Architektur [KD09, 3]

Die Generische Testsystem-Architektur (GTA) beschrieben in [PBB+, 26] beschreibt einen Aufbau mit der Unterteilung in vier Schichten:

- Test Generierungs Schicht (TGS): Ermöglicht die Generierung von Testfällen
- Test Definitions Schicht (TDS): Ermöglicht die Definition und Implementierung der Definierten Testfälle
- Test Ausführungs Schicht (TAuS): Führt die Tests durch und protokolliert die Ergebnisse
- Test Adaptions Schicht (TAdS): Stellt die Schnittstellen zur Kommunikation mit dem SUT zur Verfügung

Die Universelle Testsystem-Architektur (UTA) nach [KD09] Teilt das TAS in zwei Teile:

- Das Testmanagement, das die Aufgaben:
  - Testplanung: Zeit und Resourcen Planung und Verwaltung
  - Versionskontolle: Verwaltet die Versionierung der Tests
  - Testvorbereitung: Spezifikation und Erstellung, sowie Konfiguration und Parametrisierung der Testfälle
  - Testfortschrittskontrolle: Überwacht und Protokolliert den Ablauf der Tests

  - Testnachbearbeitung: Zusammenstellen von Testergebnissen und Testprotokollen ausführt
- Die Testausführung und -auswertung, die die Aufgaben:
  - Ausführung: führt die Einzelnen Testschritte aus, kommuniziert hierzu mit dem Testbett
  - Auswerten: Wertet die Reaktion des SUT aus, bestimmt das Ergebnis des Tests

  ausführt

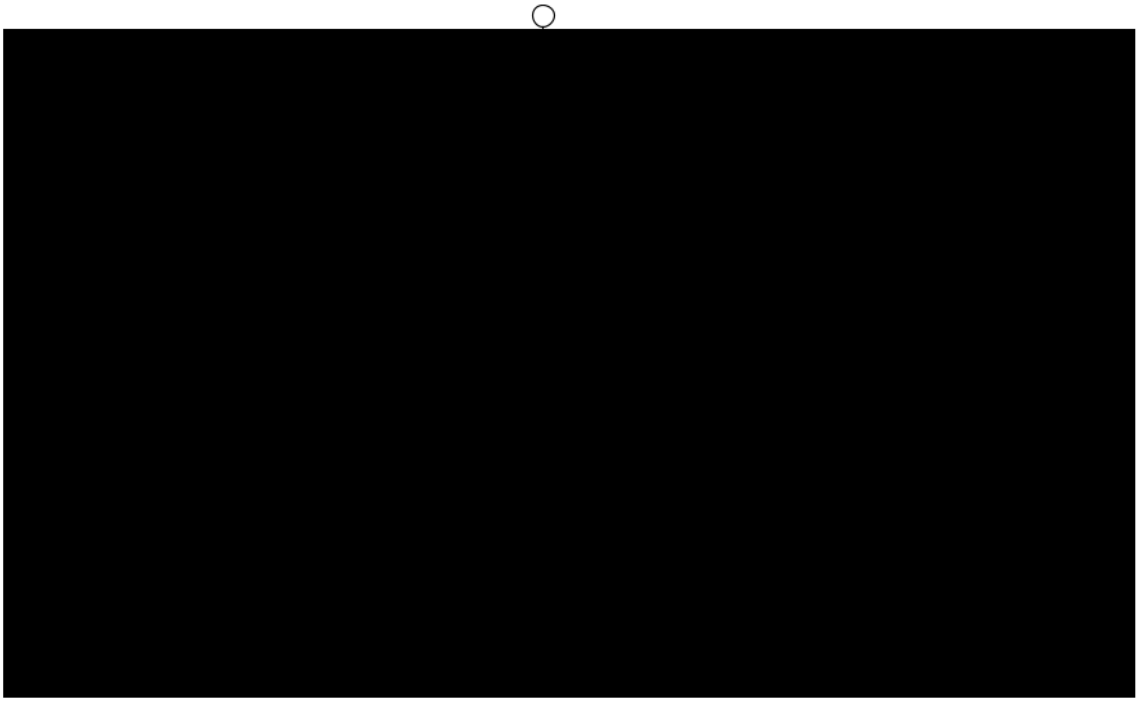
3.3 Ergebnis

In diesem Kapitel wurde eine RE Prozess erarbeitet, der in Kapitel 4 durchgeführt wird, um Forschungsfrage 1, zu finden in Tabelle 1.2, zu beantworten. Für die Beantwortung der Forschungsfrage 2, zu finden in Tabelle 1.2, wurden verschiedene Architekturen für Testautomatisierungssystem (TAS) betrachtet.

4 Spezifikation

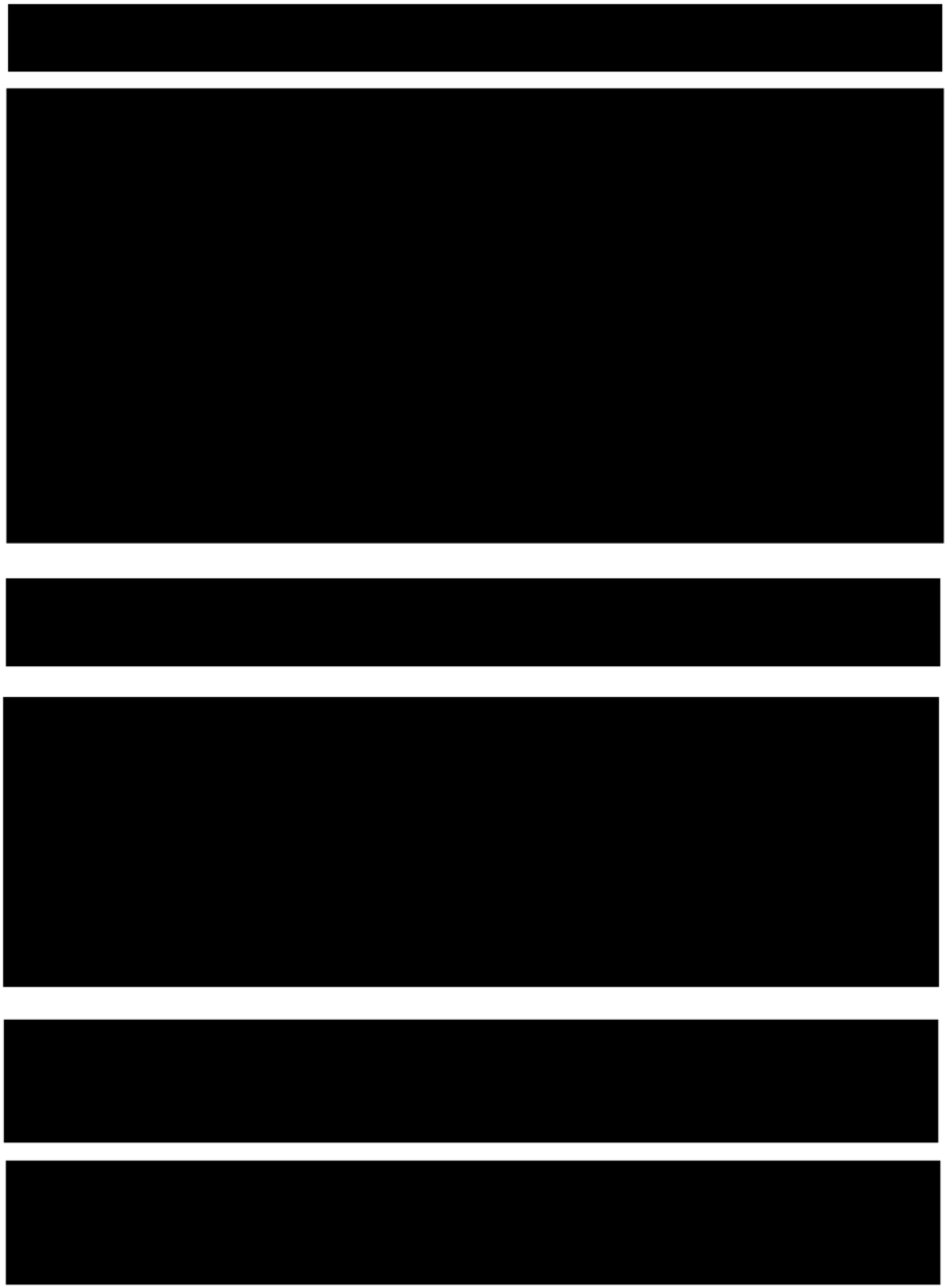
Dieses Kapitel beschreibt den, für die Beantwortung von Forschungsfrage 1, zu siehe in Tabelle 1.2, durchgeführten Software Entwurfsprozess, sowie die Ergebnisse dieses. Es umfasst die Kontext Analyse, zur Erfassung der relevanten Einflussfaktoren auf die Entwicklung. Die Anforderungsermittlung, im Zuge derer die Anforderungen, die durch die verschiedenen, gefundenen Faktoren, erfasst und ausgewertet werden. Darauf folgt der Entwurf einer Software Architektur, basierend auf diesen Anforderungen. Dies dient der Beantwortung von Forschungsfrage 2, zu finden in Tabelle 1.2.

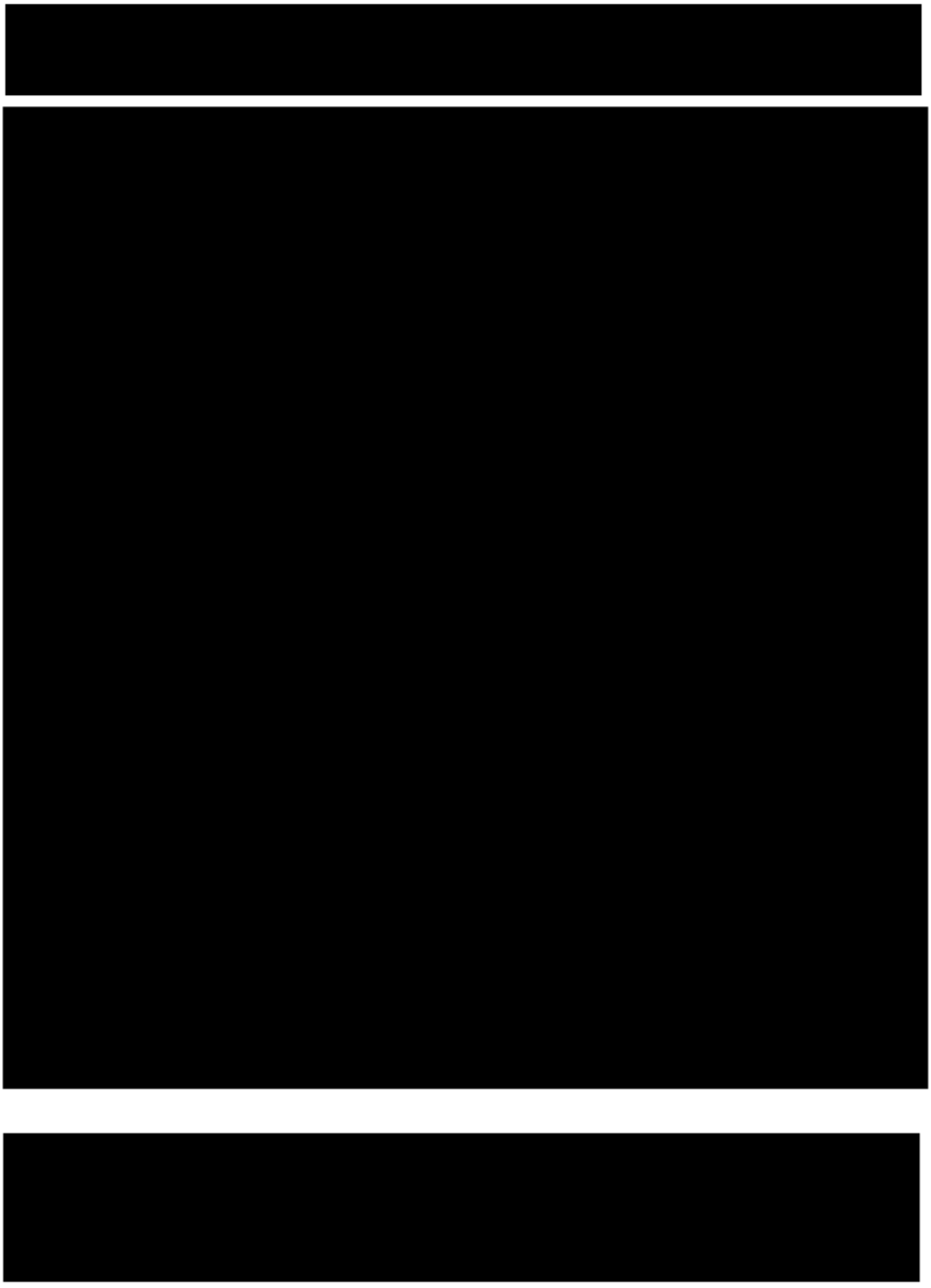
4.1 Kontext Analyse

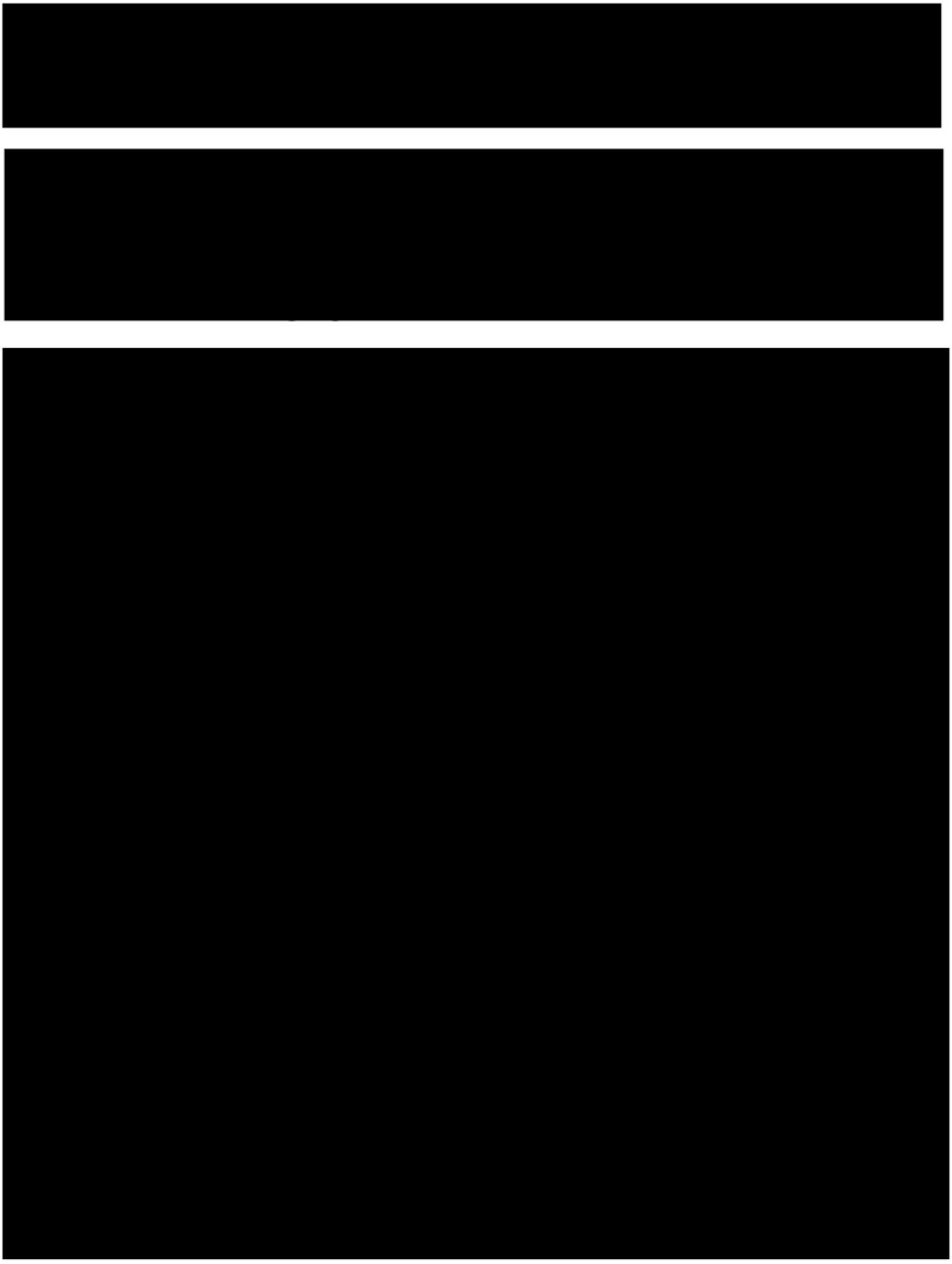


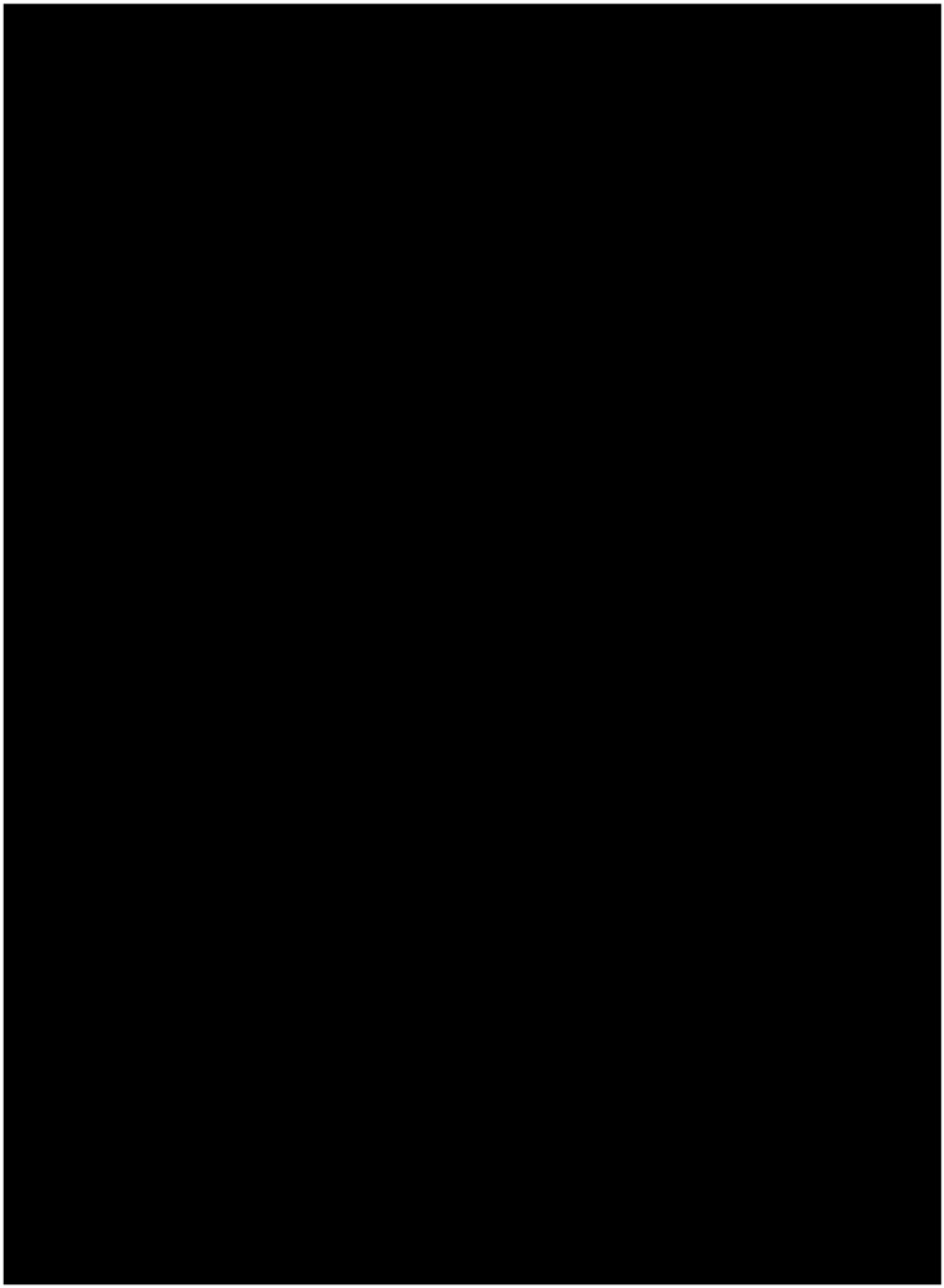
4.2 Ermittlung von Anforderungen

4.2.1 Anforderungen der Stakeholder*innen









4.3 Software Architektur

Um die Forschungsfrage 2 1.2 zu beantworten, soll, basieren auf den erfassten Anforderungen, eine Architektur für ein Softwaresystem entworfen werden, dass diese Erfüllen kann. Beim Entwurf wurden, basierend auf den Anforderungen ein Fragen formuliert, die ein bestimmte Probleme beschreiben [SH23, 232]. Für jede dieser Fragen wurden mögliche Lösungen gesucht, dokumentiert und evaluiert und zuletzt jeweils eine Lösung ausgewählt. Die detaillierte Evaluation betrachteten Möglichkeiten und Faktoren für die einzelnen Architekturentscheidung (AE) sind im Anhang 4 zu finden. Die gewählten Lösungen dienen als Grundlage für den Architektur Entwurf [Zör12, 64].

4.3.1 Architekturentscheidung (AE)

AE1.Wie sieht die grundlegende System Struktur aus? Das System kann verschieden Aufgebaut werden. Monolithisch, Verteilt, Micro Service, Geschichtet, etc. Es wurde sich entschieden auf der Generischen Testautomatisierungsarchitektur nach [PBB+, 26] aufzubauen, da diese Detaillierter ist und mehr Struktur bietet. Außerdem biete sie bereits Ansätze für die Implementierung neuer Testhardware und neuer Tests. Es wurde eine vereinfachte Variante dieser entworfen. Diese enthält nur drei Schichten:

- Test Implementierungs Schicht (TIS): Ermöglicht die Generierung, Definition und Implementierung der Definierten Testfälle
- TAS: Führt die Tests durch und protokolliert die Ergebnisse
- TAdS: Stellt die Schnittstellen zur Kommunikation mit dem SUT zur Verfügung

Die Trennung von Testgenerierungsschicht und Testdefintionsschicht wurde aufgehoben, da das TAS selber keine Entwicklungs- und Verwaltungsmöglichkeiten für Tests und Testabläufe bieten muss. Gemäß *US20* werden Tests über Jenkins verwaltet und ausgespielt. Informationen über verschiedene SUT Typen werden in einem SUT Daten Manager geladen und verwaltet. AE2.Wie wird das System an externe Datenquellen und Datensenken angeschlossen? Das TAS muss mit verschiedenen anderen Systemen kommunizieren, die nicht das SUT sind. Diese Kommunikation kann im System verteilt geschehen, oder in einer Schnittstelle gebündelt. Für das System wurde entschieden die externe Kommunikation soll in Form eines Zentralen Adapters implementiert werden. Dieser enthält Remote Proxies [GHJV94, 264] für den Zugriff auf die verschiedenen Datenquellen. AE3.Wie wird das Nutzerinterface an das TAS angebunden? Das Nutzerinterface soll dem Nutzer die Steuerung des Systems Ermöglichen und ihm Informationen über Abläufe und Zustand des TAS liefern. Dazu kann das Nutzerinterface entweder direkt in das TAS integriert werden, oder mit diesem über eine Schnittstelle kommunizieren. Es wurde Entschieden eine Client Server Architektur zu verfolgen. Dadurch werden Oberfläche und Testlogik Entkoppelt. Die bietet zusätzliche Stabilität. Darüber hinaus kann der Client auf einem anderen System ausgeführt werden, wie der Server. So kann der Testablauf überwacht und ferngesteuert werden. AE4.Wie kommuniziert der Bediener mit dem System? Das TAS benötigt Informationen über den gewünschten Testvorgang zu das SUT vom Bediener. Der Bediener wiederum möchte vom TAS über den aktuellen Fortschritt des Testablaufs, sowie dessen endgültiges Ergebnis informiert werden. Dazu benötigt das System eine UI. Hierfür gibt es verschiedene Technologien die verwendet werden können. Für das TAS soll eine Oberfläche mit dem Windows Presentation Foundation (WPF) [Lit21] zu erstellen, da hier bei den Entwickler*innen bereits ausführliche Kenntnisse vorliegen die für die Wartung des Systems benötigt werden. Außerdem wird WPF bereits in anderen Projekten der Firma eingesetzt, wodurch Ressourcen hierfür bereits existieren. AE5.Wie werden UI, Daten und Logik im Client organisiert? Das User Interface muss in der gewählten Architektur mit dem TAS über das Netzwerk Kommunizieren. Dazu ist das User Interface eine eigene Anwendung. Für den Internen Aufbau von Oberflächen Anwendungen gibt es verschiedene Architekturen. Model View Viewmodel (MVVM), glsmvp, oder glsmvc sind alles verbreitete Architekturen für die Trennung von Oberfläche und Logik. Alle betrachteten Ansätze ermöglichen dem Nutzer Eingaben und dem System Echtzeit Ausgaben. Eine Einschränkung der Funktionalität kann bei allen sowohl auf Ui Ebene, als auch auf Ebene der Logik geschehen. Alles sind etablierte Architekturen, die in WPF umgesetzt werden können. Es wurde sich für MVVM entschieden, da diese bereits in der Firma etabliert ist. Außerdem ermöglicht

die Entkopplung der Ui von den Daten und der Business Logik mehrere Ul s für die selbe Logik zu entwickeln. Diese Ul können unterschiedliche Funktionalitäten und Informationen bereitstellen. Dadurch können für Unterschiedliche Stakeholder*innen Gruppen, wie Benutzer und Entwickler, individuell Zugriff auf Funktionen und Informationen gewährt werden. AE6.Wie werden SUT Schnittstellen an das System angebunden? Das System soll es ermöglichen einfach neue Schnittstellen zum SUT einzubinden. Diese Schnittstellen sollen für die Implementierung in Tests zur Verfügung stehen. Das bedeutet sie müssen auf der Test Implementierungs Schicht (TIS), der Test Ausführungs Schicht (TAuS) und der Test Adaptions Schicht (TAdS) zur Verfügung stehen. Es wurde entschieden die Interfaces in eine Bibliothek auszulagern. Dadurch wird eine logische Trennung der Schnittstellen vom Rest des Systems herbeigeführt und sicher gestellt, dass es nicht zur Codedopplung kommt. Es wurde sich gegen eine vollständige Auslagerung in einzelne Module entschieden. Dies würde zu gesteigertem Entwicklungsaufwand führen und den Prozess für die Implementierung neuer Tests verkomplizieren. AE7.Wie sollen SUT Schnittstellen abstrahiert werden? Verschiedene SUT Schnittstellen sollen in der selben Bibliothek abgelegt werden. Sie sollen von außen, durch verschiedene Schichten des TAS zugreifbar sein. Es muss möglich sein, einfach neue Schnittstellen hinzuzufügen und bereits vorhandene anzupassen, ohne Tests, die diese verwenden verändern zu müssen. Als Abstraktion der Schnittstellen wurde die Brücken Strategie gewählt. Es werden Schnittstellen implementiert, die die benötigten Funktionalitäten abstrahiert. Eine Schnittstellenbasis, die den Zugriff auf Basisinformationen der Schnittstelle sowie Grundfunktionalitäten enthält, die zur Initialisierung der Schnittstelle benötigt werden. AE8.Wie werden Updates und Konfigurationen verwaltet? Das System soll es den Entwicklern ermöglichen neue Testabläufe und SUT Schnittstellen zu entwickeln und zu implementieren. Es soll möglichst einfach sein, Updates an die Teststände auszuspielen. Als einzige Lösung wurde Jenkins betrachtet, da die Verwendung von Jenkins für das Ausspielen und Aktualisieren der Software eine vorgegebene Randbedingung ist. Diese erfüllt darüber hinaus alle Anforderungen. AE9.Wie sollen neue Tests angebunden werden? Das System soll in der Lage sein, verschiedene Testabläufe auszuführen. Neue Abläufe sollen von den Entwicklern nach Bedarf implementiert werden können. Das System soll unterschiedliche Testabläufe speichern und verwalten können. Diese Tests können auf die unterschiedlichen SUT Schnittstellen zugreifen. Die Entwickler sollen einfach neue Tests erstellen und an das System ausspielen können. Die Entscheidung fiel auf das dynamische Laden von Modulen. Dies bietet die größere Flexibilität. Neue Tests, oder Änderungen an vorhandenen Tests können einzeln Ausgespielt werden. Dies erleichtert auch das Versions- und Konfigurationsmanagement in Zusammen-

arbeit mit Jenkins. AE10.Wie soll die Test Ausführungs Schicht (TAuS) auf Test in der Testdefinitionsschicht zugreifen? Tests werden auf der Test Definitions Schicht (TDS) geladen und verwaltet. Die Ausführung der Tests ist Aufgabe der Test Ausführungs Schicht (TAuS). Dazu benötigt diese Zugriff auf die Tests und muss mit ihnen Interagieren. Es wurde sich für das Strategie Muster entschieden. Dadurch wird die Aufteilung in Testdefinitionsschicht, in der die Tests erstellt und verwaltet werden und Testausführungsschicht, in der die Tests durchgeführt werden, beibehalten. Auch bleibt der Test gekapselt, was Abhängigkeiten reduziert und dadurch die Wartbarkeit und die Stabilität erhöht. Es wird sichergestellt, dass es keine ungewollten Abhängigkeiten zwischen dem Test und der Testausführungsschicht gibt. AE11.Wo werden die SUT Schnittstellen instantiiert? Die SUT Schnittstellen bestehen aus zwei Teilen. Der Schnittstelle zum Programm und der Implementierung. Die Schnittstelle kann auch ohne die die Implementierung verwendet werden, zum Beispiel bei der Implementierung der Tests. Beim Ausführen der Tests wird die Implementierung jedoch benötigt. Dazu muss eine Instanz erzeugt werden, auf die die Testausführungsschicht zugreifen kann. Für die Verwaltung der Schnittstellen ist in der Architektur die Test Adaptions Schicht (TAdS) vorgesehen. Um Aufwand für das neu Verbinden mit permanenten Schnittstellen (festen Teilen des Testaufbaus) zu reduzieren sollen die Schnittstellen als Singeltons existieren. Für eine bessere logische Trennung werden diese in der Test Adaptions Schicht (TAdS) erzeugt und gespeichert. Dadurch muss die Logik zum erzeugen und verwalten der Schnittstellen nicht mit in der *Inteface Bibliothek* gespeichert werden.

4.3.2 Bausteine

Basierend auf den getroffenen Architekturentscheidungen kann die Software logisch in Module zerlegt werden. Diese dienen erst der Planung und müssen in der Umsetzung nicht direkt übernommen werden. Doch helfen sie Konzepte zu verstehen und zu dokumentieren [Zör12, 103]. Diese Bausteine können dann weiter zerlegt werden. Die geplante TAA ist in Abbildung 4.6 zu sehen.

Bausteine der Testimplementierungsschicht

Entspricht der *Testerstellung-* und *Testdefinitionsschicht* der Generische Testsystem-Architektur (GTA) aus [Bau21, 70]. Sie lädt und verwaltet vordefinierte Testabläufe. Die Tests sind nach dem Bridge Pattern [GHJV94, 200] in eine Abstrakte Testbeschreibung und eine konkrete Testimplementierung geteilt. Die Testbeschreibung enthält Metainformationen, wie Name und eine sprachliche Be-

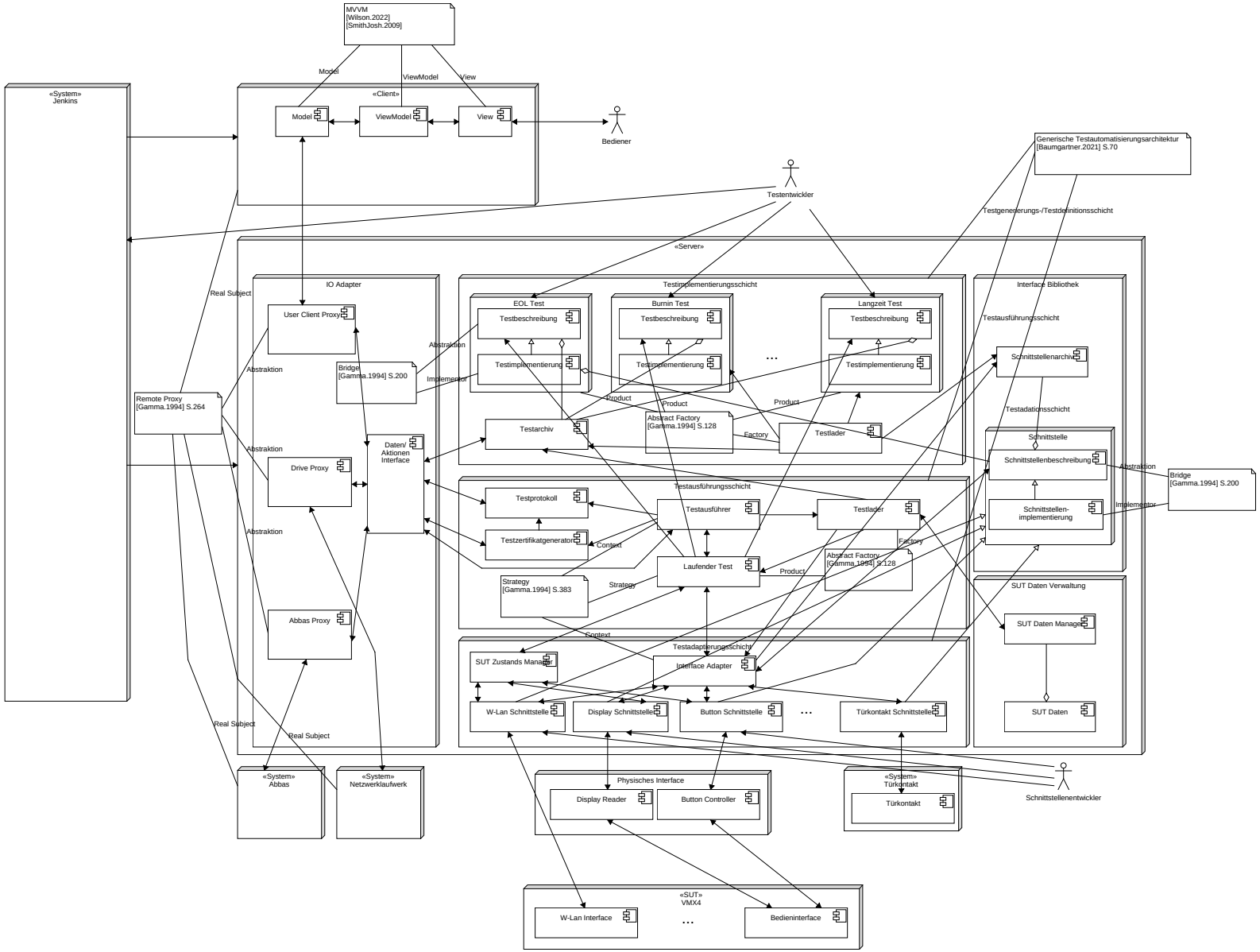


Abbildung 4.6: Geplante TAA

schreibung, für den Bediener. Die Testimplementierung enthält den Code für den Testablauf. Der Zugriff auf SUT Schnittstellen erfolgt über abstrakte *Schnittstellenbeschreibungen*, die aus der Interface Bibliothek bezogen werden. Testbeschreibung und Testimplementierung liegen, gemäß AE 9 4.3.1 in Form externer Plugins vor, die vom *Testlader*, zur Laufzeit, geladen und im *Testarchiv* abgelegt werden. Informationen über die vorhandenen Tests werden über das *Daten/Aktionen Interface* des *IO Adapters* dem Client zu Verfügung gestellt.

Bausteine der Testausführungsschicht

Die Test Ausführungs Schicht (TAuS) bezieht Tests von der Test Definitions Schicht (TDS) und führt diese aus. Sie empfängt Befehle über den *IO Adapter* und schickt Informationen über den Status des aktuellen Tests an diesen. Der *Testausführer* steuert und überwacht den Ablauf von Tests. Er empfängt Befehle vom Client, über das *Daten/Aktionen Interface* des *IO Adapters* und weißt auf Anforderung den *Testlader* an, bestimmte Testabläufe aus dem *Testarchiv* zu laden und vorzubereiten. Er führt auf Befehl Tests aus und protokolliert deren Ablauf im *Testprotokoll*. Nach Ablauf des Tests gibt der *Testausführer* dem *Testzertifikatgenerator* den Befehl ein Testzertifikat zu generieren. Der *Testlader* lädt als Builder [GHJV94, 139], Tests aus dem *Testarchiv* der TIS und macht sie für die Ausführung bereit (als *Laufender Test*). Dazu lädt er benötigte SUT Schnittstellen aus dem *Interface Adapter* der TAdS und verknüpft sie mit den abstrakten Beschreibungen, die in der Test Implementierung verwendet werden. Der *Laufende Test* ist das Testscript das gerade ausgeführt wird. Der *Testausführer* sendet Befehle den Test schrittweise abzulaufen. Die einzelnen Testschritte interagieren mit dem SUT über den *Interface Adapter* der Test Adaptions Schicht (TAdS). Der *Testzertifikatgenerator* erzeugt aus dem *Testprotokoll* eines Tests ein Testzertifikat und legt dieses über des *Daten/Aktionen Interface* des *IO Adapters* auf dem Netzwerklaufwerk ab.

Bausteine der Testadaptierungsschicht

Die Test Adaptions Schicht (TAdS) lädt benötigte SUT Schnittstellen aus der *Interface Bibliothek* und verwaltet sie. sie instantiiert die Schnittstellen, als Singelton, der für alle Tests verwendet wird. Schnittstellen werden beim Laden des Moduls geladen und initialisiert. Interface Adapter Verwaltet alle geladenen Schnittstellen als Singeltons. Dadurch wird verhindert, dass Schnittstellen

mehrfach geladen und initialisiert werden. Da die SUT Schnittstellen nur einfach vorliegen und nicht für parallelen Zugriff vorgesehen sind. SUT Zustands Manager Erfasst Informationen über den Zustand des glssut von den verschiedenen Schnittstellen und kumuliert diese. Stellt grundlegende Informationen bereit, wie, ob das SUT eingeschaltet ist.

Bausteine der Interface Bibliotheken

Die Interface Bibliothek enthält Schnittstellen für die Kommunikation mit dem SUT. Schnittstelle Eine Schnittstelle ist nach dem Bridge Pattern [GHJV94, 200] in eine abstrakte *Schnittstellenbeschreibung*, in der die Befehle für die Kommunikation über diese bestimmte SUT Schnittstelle definiert sind und eine *Schnittstellenimplementierung* die die Logik für die Kommunikation mit dieser Speziellen Schnittstelle des SUT enthält, aufgeteilt. Die *Schnittstellenimplementierung Schnittstellenbasis*, erbt von der abstrakten Schnittstellenbasis, die Grundfunktionalitäten für alle SUT Schnittstellen bereit stellt. Schnittstellenarchiv Das Schnittstellenarchiv verwaltet alle Schnittstellen, die in der Interface Bibliothek vorhanden sind. Es bietet eine Liste der verfügbaren Schnittstellen und die Möglichkeit, für eine *Schnittstellenbeschreibung* die zugehörige *Schnittstellenimplementierung* zu beziehen.

Die SUT Daten Verwaltung

Verwaltet Basis Informationen über verschiedene SUT Typen. Diese werden beim Programmstart aus JSON Dateien geladen und verschiedenen Programmteilen zur Verfügung gestellt. SUT Daten Ein Satz an Informationen zu einem bestimmten SUT Typ. Dies beinhaltet allgemeine Informationen, wie der Name des SUT Typ, unter welchem Laufwerksname, das SUT sich mit dem Testsystem verbindet und Grenzwerte für verschiedene Hardware Parameter.

Aufbau des IO Adapter

Enthält gemäß AE 2 4.3.1 alle Schnittstellen des Systems zu externen Aspekten. Für jede externe Datenquelle, oder Senke, gibt es einen Proxy, der die Verbindung und Kommunikation mit diesem verwaltet und steuert [GHJV94, 264]. Alle Daten und Aktionsmöglichkeiten werden im *Daten/Aktionen Interface* gebündelt. User Client Proxy Stellt einen Server bereit, über den der Nutzer Client mit dem Rest des TAS kommuniziert, wie in *AE3* entschieden. Drive Proxy Stellt

eine Verbindung zum Netzwerklaufwerk bereit, auf dem das Testzertifikat und Spulengrenzwerte abgelegt werden sollen. Abbas Proxy Stellt eine Verbindung zu Abas bereit, über die Informationen zu das SUT abgerufen und Informationen über das SUT und den Testablauf erfasst werden können.

Bausteine User Interface

Ermöglicht dem Bediener die Interaktion mit dem System. Ist, wie in *AE3* entschieden, vom restlichen System entkoppelt. Wird gemäß AE 5 4.3.1 nach dem MVVM Muster aufgebaut.

Das Konfigurationsmanagement

Verwaltet verschiedene Versionen des TAS. In *AE8* wurde entschieden hierfür *Jenkins* einzusetzten.

4.3.3 Laufzeitsichten des TAS

Ablauf TAS Initialisierung

Nach dem Start des TAS müssen verschiedene Komponenten geladen und initialisiert werden, Ablauf zu sehen in Abbildung 4.7. Wie in AE 9 4.3.1 entschieden sollen Tests als externe Module vorliegen und zur Laufzeit geladen werden. Dies übernimmt der Testlader. Dieser detektiert die Testmodule, lädt die darin gespeicherten Test und trägt sie im Testarchiv ein. In *AE2* wurde entschieden externe Datenquellen und Datensenken in einem Daten/Aktionen Interface zusammen zu fassen. Dieses muss zum Start ebenfalls initialisiert werden. Das Interface selbst wiederum initialisiert die einzelnen Schnittstellen. Dies sind der User Client Proxy, über den die Nutzer Oberfläche angebunden ist. Der Drive Proxy, der die Verbindung zum Netzwerklaufwerk darstellt, auf dem die Zertifikate gespeichert werden. Und der Abbas Proxy, der die Kommunikation mit dem ERP Abbas steuert.

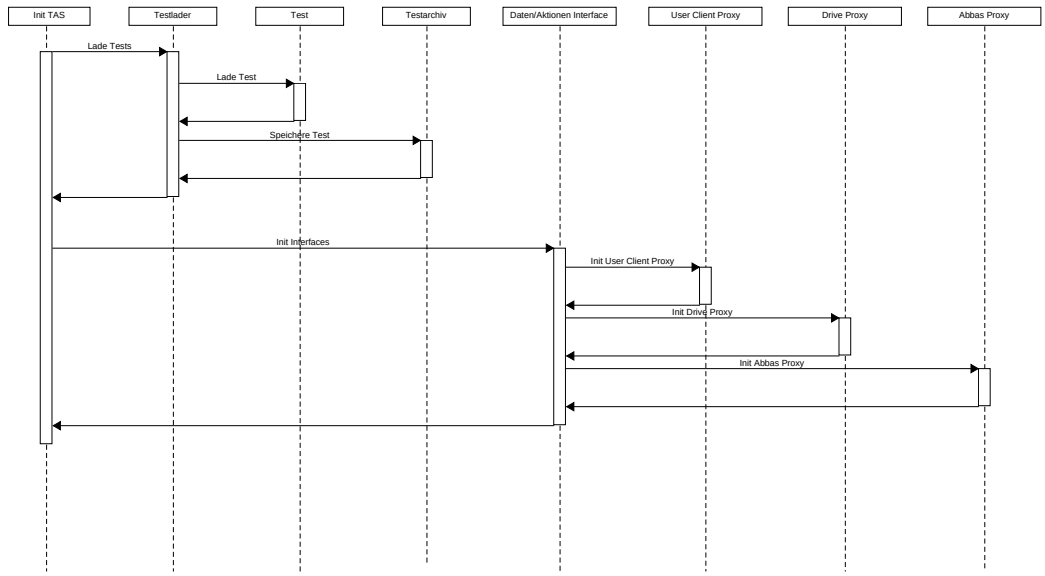


Abbildung 4.7: Ablauf Initialisierung TAS

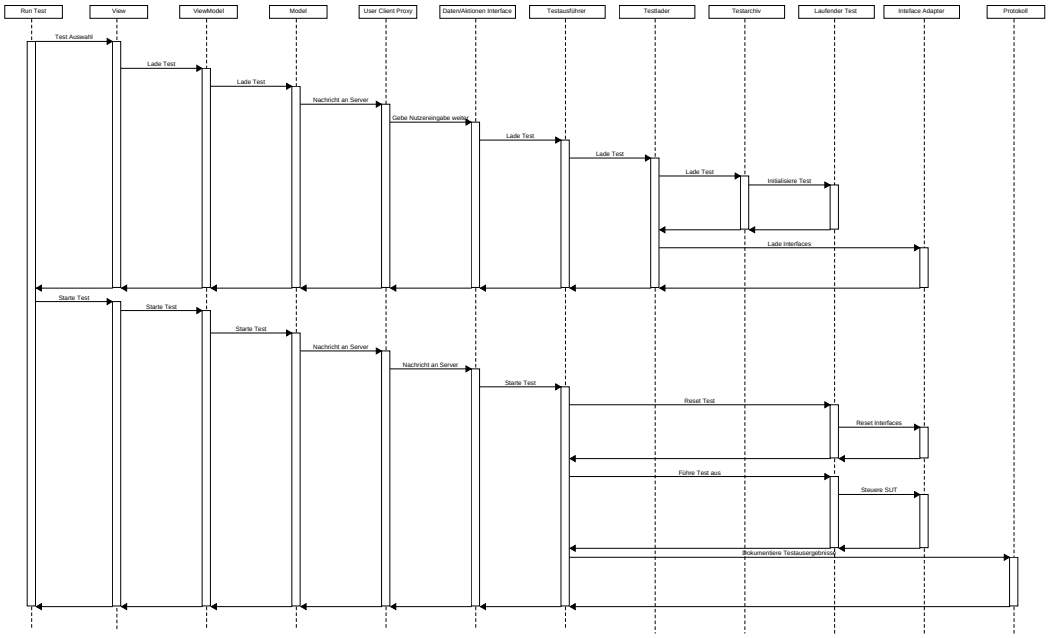


Abbildung 4.8: Ablauf Testausführung

Ablauf Test Ausführung

Die Ausführung ist ein mehrstufiger Prozess, zu sehen in Abbildung 4.8. Um einen Test auszuführen, wählt der Nutzer auf der *View* des Clients den Test den er ausführen möchte. Dieser Befehlt wird gemäß 4.3.1 durch ein *ViewModel* an das *Model* weitergeleitet. Dieses hält die Verbindung zum *User Client Proxy* im Server. Dieser leitet den Befehl über das *Daten/Aktionen Interface* an den *Testausführer* weiter. Diese befielt dem *Testlader,* den gewünschten Test zu laden. Der *Testlader* holt den gewünschten Test aus dem Testarchiv, als *Laufender Test* und Initialisiert ihn. Anschließend lädt er die für den Test notwendigen Schnittstellen über den *Interface Adapter* und Initialisiert sie. Ob das Laden des Tests erfolgreich war, wir dem Client zurück gemeldet. Diese gibt dem Nutzer nun die Möglichkeit den Test zu starten.

Möchte der Nutzer den gewählten Test starten, gibt er auf der *View* den Befehl. Dieser wird wieder über das *ViewModel,* an das *Model* und von diesem an den *User Client Proxy* im Server weiter gegeben. Dieser leitet den Befehl über das *Daten/Aktionen Interface* an den *Testausführer* weiter. Der *Testausführer* setzt nun den *Laufender Test* in den Startzustand zurück. Dazu setzt der *Laufender Test* die verwendeten Interfaces zurück. Der *Testausführer* kann nun den *Laufender Test* ausführen. Während des Testablaufs steuert der *Laufender Test* das SUT über die Interfaces, die er über den *Interface Adapter* geladen hat. Das Ergebnis des Tests dokumentiert der *Testausführer* im *Protokoll* und meldet dieses nach Abschluss des Testablaufs an den Client zurück.

4.4 Ergebnis

Um Forschungsfrage 1, siehe Tabelle 1.2, zu beantworten, wurden in diesem Kapitel die Anforderungen an das TAS erfasst. Für die Beantwortung von Forschungsfrage 2, zu finden in Tabelle 1.2, wurde basierend auf diesen Anforderungen wurde eine Softwarearchitektur entworfen, die diese erfüllen kann. Sie dient als Basis für die Implementierung eines Prototypen, dessen Implementierung im nächsten Kapitel 5 beschrieben wird.

5 Softwareentwurf

Um die Funktionalität der entworfenen TAA zu evaluieren, wurde ein Prototyp eines TAS basierend darauf implementiert. Dieser Prototyp beinhaltet alle beschriebenen Komponenten und die wichtigsten Funktionen. In diesem Kapitel werden wird die Struktur des Prototypen, sowie die Umsetzung wichtiger Features beschrieben.

5.1 Projektstruktur

Die entworfene Architektur wurde in Form eines Prototyps umgesetzt. Dieser beinhaltet alle beschriebenen Komponenten und die wichtigsten Funktionen. Zuerst wurde die Struktur des Prototypen in Form eines Klassendiagramms 5.2 festgehalten. Als Grundlage dienten die, in der Architektur beschriebenen Bausteine und Laufzeitsichten. Für die Implementierung wurden diese Klassen auf Pakete aufgeteilt, zu sehen im Paketdiagramm 5.1.

- *CommonLib* Datenstrukturen und Protobuf Nachrichte, die im Client und im Server benötigt werden
- *TASCoreLib* Lade-, Ausführungs- und Verwaltungslogik des TAS
- *InterfaceLib* Definitionen und Implementierungen der Schnittstellen zum SUT
- *TAS* Wrapper für die *CoreLib* um diese zu Laden und auszuführen
- *TASClient* Client Applikation für die Bedienung durch den Nutzer
- *DemoTests* Beispiel Testscripte zum testen und demonstrieren des TAS

Die Beispiel Tests werden in einer eigenen Bibliothek gespeichert, die als Modul im TAS geladen werden.

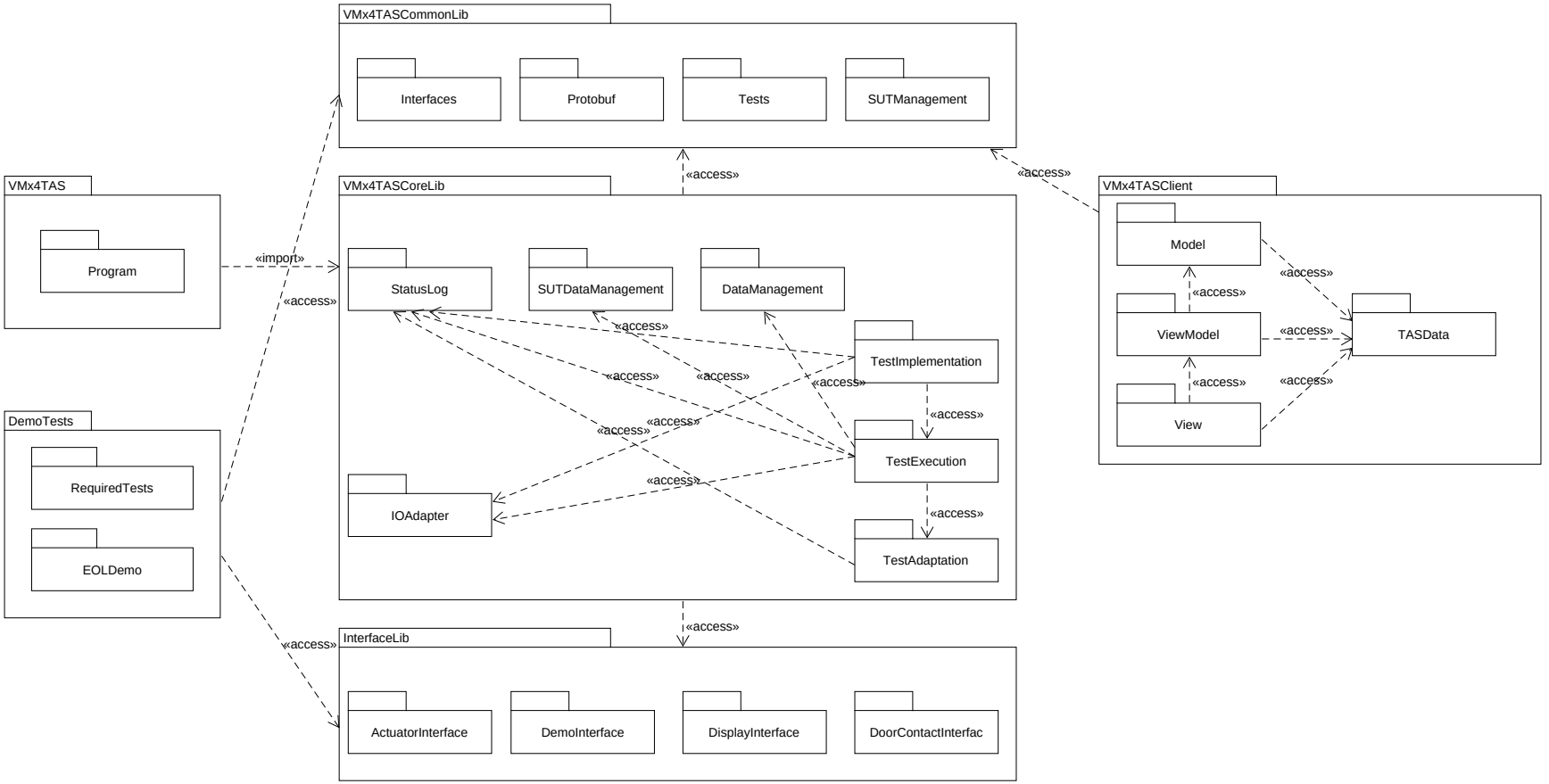


Abbildung 5.1: Paketdiagramm der geplanten Struktur und Aufteilung des TAS

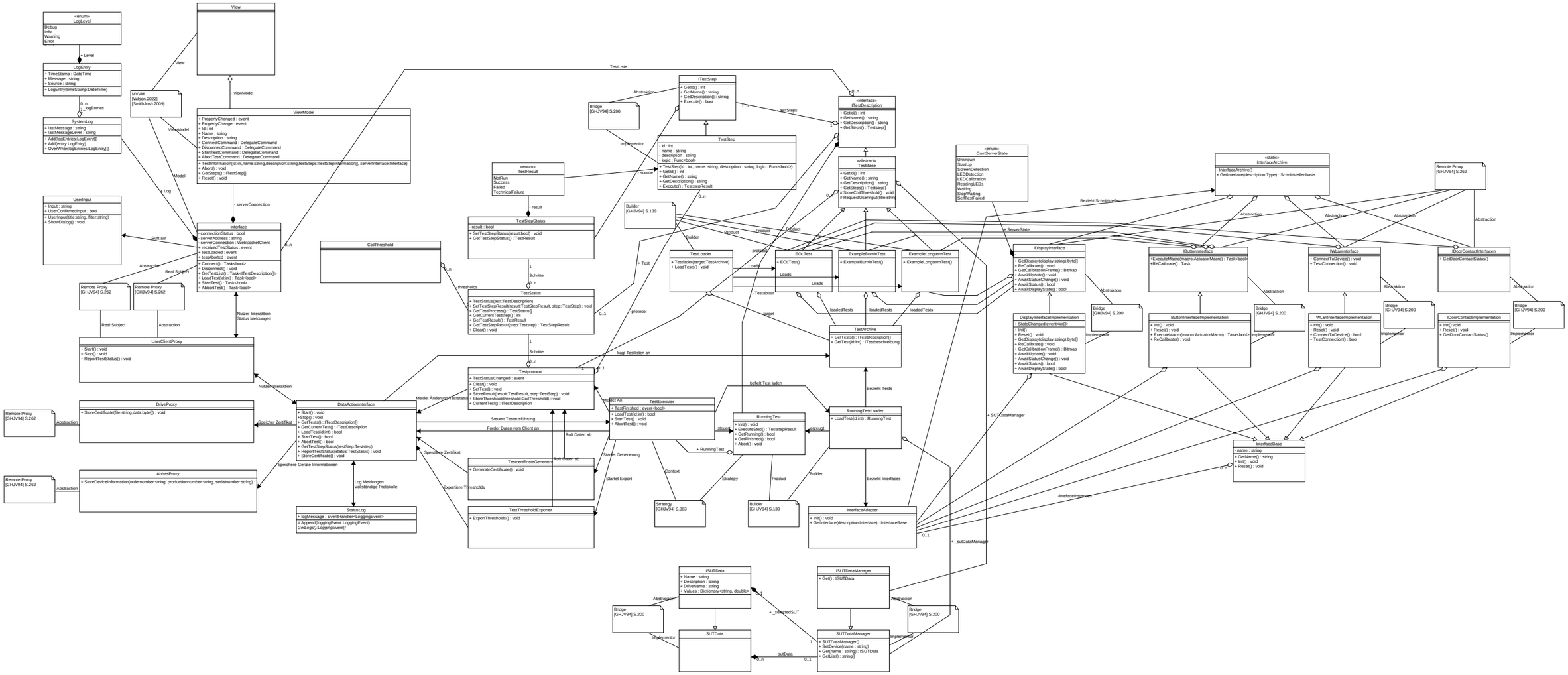


Abbildung 5.2: TAA Implementierung Klassendiagramm

5.2 Definition von SUT Schnittstellen

Um mit dem SUT interagieren zu können, benötigt das TAS Schnittstellen. Diese haben immer eine Softwarekomponenten, über die sie an das TAS angebunden werden. Interface Klassen erben von der abstrakten Klasse Interface Base, die grundlegende Funktionalitäten und Informationen für Schnittstellen bereitstellt und verwaltet. Dies sind ein Name und eine Beschreibung, sowie abstrakte Methoden für die Initialisierung, den Reset und die Deinitialisierung der Schnittstelle. Diese Informationen und Methoden müssen von jeder Schnittstelle bereit bereitgestellt werden. Gemäß AE 7 in Abschnitt 4.3.1 werden alle Methoden und Eigenschaften, über die mit der Schnittstelle bei der Testausführung interagiert werden, in einer Schnittstelle definiert. Diese ist unabhängig von der Schnittstellen Klasse, die jedoch umgekehrt die Schnittstelle implementiert.

Für den Prototypen wurden vier Schnittstellen konzeptioniert, die für die vorgesehen Demo Tests benötigt werden. 1) Button Interface, das mit dem neuen Interface Adapter, zu sehen in Abbildung 4.5, interagiert und darüber die Knöpfe des Bedienfelds des SUT betätigen kann. 2) Display Interface, das mit dem neuen Interface Adapter, zu sehen in Abbildung 4.5, interagiert und darüber die Anzeige des Geräts ausließt 3) WLan Interface, das über WLan eine Verbindung mit dem Gerät herstellt, um die WLan Funktionalität des SUT zu validieren 4) Door Contact, der mit einem Schalter an der Tür der Testkammer interagiert und sicherstellt, dass die Kammer während automatischer Testschritte geschlossen ist. Für den Prototyp wird hier ein simpler Schalter verwendet.

5.3 Definition, Laden und Ausführen von Tests

In AE aus Abschnitt 4.3.1 wurde entschieden Testscripte in Bibliotheken auszulagern und zur Laufzeit zu laden und zu verwalten. Klassen, die Testscripte definieren erben von der abstrakten Klasse TestBase. Diese dient als Markierung, um Testscripte erkennen zu können und stellt wichtige Grundfunktionalitäten für die Erstellung von Tests bereit. Interagiert wird mit dem SUT über SUT-Schnittstellen. Dazu werden bei der Implementierung der Tests die Abstrahierung der Schnittstelle in der Klasse als Feld definiert. Die Methoden und Properties dieser können anschließend in den Testabläufen verwendet werden. Dynamic Linked Library (DLL) ist Dateiformat für die Speicherung von dynamischen Programmbibliotheken unter Windows. Diese können in C#zur Laufzeit geladen und verwendet werden. Dazu werden sie mit der Methode *LoadFrom* der *Assembly* Klasse geladen. Die enthaltenen Klassen werden dazu in die Domäne der Anwendung geladen und können anschließend im Programm verwendet werden [MCA22]. Zur Laufzeit iteriert das Programm alle DLL-Datei Dateien, die im Unterordner *Tests* des Ausführungsverzeichnisses liegen und lädt diese. Testklassen werden

daran erkannt, dass sie von der abstrakten Klasse TestBase erben. Von allen Tests werden Instanzen erzeugt und im Testarchiv abgelegt. Um einen Test auszuführen muss die Testklasse zuerst befüllt werden. Das TAS erfasst dazu welche SUT-Schnittstellen benötigt werden. Die Klasse iteriert hierzu, unter Verwendung der *Reflection* Bibliothek welche Felder und Methoden sie selbst implementiert. Felder, die mit dem *TestBaseInformationAttribute* markiert sind stellen Informationen dar, die der Test vor der Ausführung vom Bediener benötigt. Methoden, die mit dem *TestStepAttribute* markiert wurden, sind die einzelnen Schritte, die bei der Ausführung des Testscript abgearbeitet werden. Beide Attribute erfassen den Name und eine Beschreibung für das Element. Methoden, die Testschritte definieren müssen ein Wert des Typs *TestResult* zurück geben und einen CancellationToken entgegen nehmen. Da C#einen Kooperativen Ansatz für den Abbruch von Threads verfolgt, sollten Threads nicht von außerhalb beendet werden [MCT22]. Ein Abbruch wird über den Token signalisiert, der Zustand des Tokens muss jedoch innerhalb der Methode aktiv abgefragt und verarbeitet werden.

5.4 Client Server Kommunikation

Wie in 4.3.1 entschieden wurde das TAS in Client und Server geteilt. Der Server enthält die Logik für das Laden, verwalten, ausführen und auswerten der Tests. Der Client enthält möglichst wenig Logik und dient nur der Darstellung des Zustands des TAS und der Bedienung durch den Nutzer. Für eine einfache, bidirektionale Kommunikation wird das WebSocket Protokoll verwendet [FM11]. Dieses erlaubt es auch, dass mehrere Clients sich gleichzeitig mit dem Server verbinden. Für die Serialisierung der Daten wird Protobuf verwendet [Goo08]. Die Protobuf Bibliothek wird bereits in anderen Projekten des Unternehmens eingesetzt und bietet, im Vergleich mit anderen üblichen Serialisierungstechniken, eine hohe Serialisierungs- und Deserialisierungsgeschwindigkeit und erzeugt kleine Nachrichten [PC20]. Da die Netzwerkkommunikation auf Server und auf Client Seite in einer eigenen Klasse gekapselt sind, können sowohl das Kommunikationsprotokoll, als auch die Serialisierungstechnologie einfach ausgetauscht werden.

Um die Fernüberwachung des Testablaufs zu ermöglichen, können sich mehrere Clients gleichzeitig mit dem Server verbinden. Jeder Client muss korrekt den aktuellen Zustand und die aktuelle Konfiguration des TAS anzeigen. Dazu werden Konfigurationsänderungen durch einen Client, sofort an alle anderen, verbundenen Clients weiter geleitet, damit diese die neue Konfiguration anzeigen.

5.5 Client Ui

Die Bedier*in soll mit dem TAS über eine Grafische Benutzer Oberfläche Interagiere. Als Technologie wurde sich in AE 4 4.3.1 für WPF entschieden. Der Client soll intern nach dem MVVM Entwurfsmuster aufgebaut werden AE 5 4.3.1. Dazu werden im Client ein Model, in Form der *Interface* Klasse Implementiert, die mit dem Server Kommuniziert und Informationen über den Zustand und die Konfiguration des TAS speichert und bereitstellt. Die Ui wird in der *MainWindow.xaml* und *MainWindows.xaml.cs* definiert. Verbunden werden beide durch das *MainViewModel*. Die Verbindung zwischen ViewModel und View geschieht über Bindings [dG22]. Die Ui ist in vier Tabs geteilt. Der **Setup** Tab, zu sehen in Anhang 8.1, erlaubt die Eingabe von grundlegenden Informationen, wie die Adresse des TAS Servers und Port, und der Name des Benutzers. Der **Test Procedure** Tab, zu sehen in Anhang 5.3, dient der Konfiguration, Ausführung und Überwachung von Testabläufen. Werden wiederholte Tests ausgeführt, bietet das **Long Term Protocol** 8.2 einen Überblick über die Ergebnisse der einzelnen Testabläufe. Der System Log Tab, zu sehen in Anhang 8.3, gibt der Bediener*in Informationen über den Zustand und die Abläufe des TAS, sowie Details zu aufgetretenen Fehlern.

5.6 Status Log

Das TAS verwendet log4net [log23] um die Abläufe innerhalb des TAS zu protokollieren. Um Bediener*innen und Entwickler*innen detailliertere Informationen über den Zustand und Ereignisse innerhalb des TAS zu geben, werden die Log Meldungen auf der Oberfläche angezeigt. Dazu wurde ein eigener LogAppender implementiert, der die Log Meldungen in einer Liste speichert und gesammelt bereit stellt. Diese Liste wird mit den verbundenen Clients synchronisiert. Auf den Clients werden die Logs als Tabelle, im Log Tab 8.3 dargestellt. Die einzelnen Meldungen sind, basierend auf ihrem Log Level, farbcodiert.

5.7 Schnittstellen

Alle konzeptionierten Schnittstellen wurden Umgesetzt, um diese in Beispiel Tests verwenden zu können. 1) Aktuator Interface: Es können Befehle gesendet werden, eine Taste zu drücken, los zu lassen, oder für eine gegebene Zeit gedrückt zu halten 2) Display Interface: Erlaubt die aktuelle Anzeige des Geräts auszulesen, mit Mustern zu vergleichen, oder auf bestimmte Muster zu warten 3) Door Contact Interface: Meldet, ob der Tür Kontakt geöffnet, oder geschlossen ist. Löst ein Ereignis aus, wenn die Tür geöffnet, oder geschlossen wird 4) WLan Interface: Erlaubt das Scannen nach aktiven W-Lan Netzen und das Testen dieser

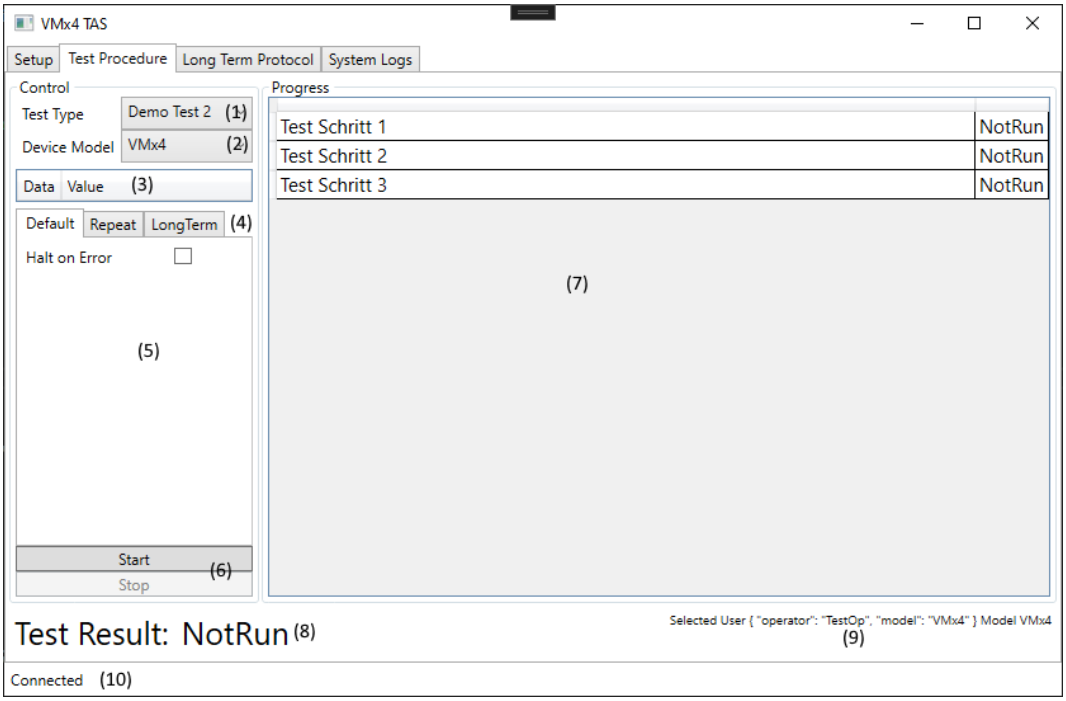


Abbildung 5.3: Test Konfiguration und Ausführung auf der Oberfläche (1) Auswahl des Testscripts (2) Auswahl welcher Geräte Typ getestet wird (3) Eingabe von Start Werten, die das Testscript erfordert (4) Auswahl des Modus des Tests (Einfach, Wiederhole X Mal, Langzeit Test bis Zeitpunkt) (5) Test Modus spezifische Einstellungen (6) Starte/Stoppe Ausführung des Tests (7) Auflistung der Testschritte und deren Ergebnis (8) Gesamt Ergebnis des Testablaufs (9) Letzte Statusmeldung vom TAS Server (10) Verbindungsstatus des Clients zum Server

5.8 Beispiel Tests

Um das TAS zu testen und seine Funktionalität zu demonstrieren, wurden eine Reihe an Beispiel Tests implementiert, vollständiger Code in Anhang 8.1. Der wichtigste ist der *EOL Demo* Test. Dieser demonstriert alle bereits implementierten Funktionalitäten, die für den EOL Prozess benötigt werden. Der Test umfasst 1) Das Erfassen der Seriennummer des Geräts 2) Das Einschalten des Geräts 3) Die Prüfung ob das Gerät auf Eingaben reagiert 4) Frage Daten vom Nutzer an

Ein zweiter Test *RequiredTests* wurde implementiert, um zwei komplexere Abläufe zu demonstrieren. Der Code des Test ist in 8.2 zu finden. Dieser Test 1) Stellt sicher, dass das VMx4 eingeschaltet ist 2) Wechselt über das Menü des VMx4 in den High Power Mode 3) Aktiviert über das Menü des VMx4 die WLan Funktionalität, prüft, ob das WLan verfügbar ist und funktioniert und deaktiviert die WLan Funktionalität anschließend wieder

5.9 Ergebnis

Der in diesem Kapitel beschriebene Prototyp dient der Evaluation der entworfenen Architektur. Im nächsten Kapitel 6 wird dieser, zusammen mit den beteiligten Stakeholder*innen, vollständige Liste in 4.2.1, evaluiert.

6 Evaluation

In diesem Kapitel wird geprüft, ob die entworfene Architektur und der darauf basierende Prototyp die gewünschten Anforderungen erfüllt. Dazu wird evaluiert, in wie weit die, für die Anforderungen festgelegten Akzeptanzkriterien erfüllt werden. Anschließend wird bestimmt, welche Schritte nötig sind, um die noch nicht erfüllte Anforderungen zufrieden zu stellen. Bei der Evaluation wird versucht so viele der Stakeholder*innen wie möglich hinzu zu ziehen. Der vollständige Anforderungskatalog ist im Anhang 2 zu finden.

6.1 Prüfung Anforderungen

Im Zuge der Anforderungserfassung in Abschnitt 4.2 wurden User Stories, Qualitäts Merkmale und Randbedingungen erfasst, die das System erfüllen muss, oder sollte. Im ersten Schritt der Evaluation werden diese gesichtet und festgestellt, in wie weit diese erfüllt wurden, oder was noch getan werden muss, um diese zu erfüllen.

6.1.1 User Stories

Für die User Stories werden die erfassten Akzeptanzkriterien als Leitfaden für die Überprüfung verwendet. Den Stakeholder*innen wurden die Funktionalitäten vorgeführt und ihr Feedback dokumentiert, zu finden in Anhang 1. Dabei wurde festgestellt, welche User Stories bereits erfüllt sind und was noch getan werden muss, um die verbleibenden zu erfüllen. Die relevanten Punkte sind:

6.1.2 Qualitätsmerkmale

Es wurden Tests durchgeführt, um die erfassten Qualitätsmerkmale zu validieren, doch aufgrund des Stands des Prototyp, war die Evaluation vieler Merkmale nicht, oder nur unvollständig möglich. Siehe Tabelle 6.1.

6.1.3 Randbedingungen

Da Randbedingungen sehr konkret sind und fundamental für die Software Architektur kann durch einen direkten Test, oder eine Analyse geprüft werden, ob diese Erfüllt ist. Tabelle 6.2 zeigt die Ergebnisse.

6.2 Akzeptanz Prüfung

Eine Prüfung des Prototypen nach dem Technology Acceptance Model [ANT92] war angedacht, war jedoch aufgrund Zeitlicher und Organisatorischer Einschränkungen, innerhalb des Thesiszeitraums nicht möglich.

6.3 Was bleibt zu tun

7 Fazit

7.1 Zusammenfassung

Im Zuge dieser Bachelor Arbeit wurde eine Architektur für ein automatisiertes Testsystem für Metalldetektoren entworfen. Dazu wurde eine Kontextanalyse durchgeführt um Stakeholder*innen des Projekts zu identifizieren. Diese wurden befragt um ihre Anforderungen an das System zu erfassen. Eine Feldbeobachtung wurde durchgeführt, um die aktuellen EOL Prozess zu erfassen. Die erfassten Anforderungen wurden mit den Stakeholder*innen in Zuge eines Walkthroughs abgestimmt. Für die Beantwortung von Forschungsfrage 1, zu finden in Tabelle 1.2, wurden die Ergebnisse in einem Anforderungskatalog 2 zusammengefasst. Zur Klärung von Forschungsfrage 2, zu finden in Tabelle 1.2, wurde auf Basis dieses wurde eine Architektur für ein modulares Testautomatisierungssystem (TAS) entworfen 4.3. Dieses erlaubt die einfache Einbindung neuer SUT Schnittstellen und Testscripte. Sie lässt sich außerdem leicht um weitere Schnittstellen zu externen Datenquellen, wie Datenbanken erweitern. Um diese Architektur zu demonstrieren und zu testen, wurde ein Prototyp basierend darauf implementiert 5. Weiter wurden die Display- und die Button-Schnittstelle des physischen Interface Adapters angebunden. Beispiel Tests demonstrieren die Funktionalitäten des Systems und der Architektur. Anhand dieses Prototyps wurde geprüft, welche Anforderungen erfüllt wurden und was noch getan werden muss, um die Stakeholder*innen zufrieden zu stellen. Diese können in Kapitel 6 gefunden werden.

7.2 Fazit

Das Ziel dieser Bachelor Arbeit war die Verbesserung der Qualitätssicherung der Firma Vallon. Dazu sollte ein neues Testautomatisierungssystem (TAS) für die Automatisierung des End of Line (EOL) Test entwickelt werden. Diese Entwicklung wurde in zwei Teile geteilt. Für die Beantwortung der Forschungsfrage 1, zu finden in Tabelle 1.2, wurden die Anforderungen an ein modulares automatisiertes Testsystem für Metalldetektoren in 2 erfasst. Diese Anforderungen stellen sicher, dass das entwickelte TAS benötigte Features hat, um für den geplanten Zweck eingesetzt zu werden. Dieser Katalog umfasst User Stories, die erwünschte Features beschreiben, Qualitäts Merkmale, die nicht Funktionale Vor-

gaben und Randbedingungen, die Verwendete Technologien vorgeben. Basierend darauf wurde eine Architektur für ein Testautomatisierungssystem (TAS) entworfen, um Forschungsfrage 2, zu finden in Tabelle 1.2, zu beantworten. Es wurde die Generische Testsystem-Architektur (GTA) als Basis für die neue TAA gewählt, da diese bereits eine Aufteilung und damit eine gewisse Modularität beinhaltet. Die entworfene Architektur ist in acht Module aufgeteilt. Testscripte werden durch die Test Implementierungs Schicht (TIS) geladen und verwaltet. Dazu werden Testscripte in einer separaten Bibliotheken gespeichert und zur Laufzeit, dynamisch geladen. Dies erlaubt es einfach neue Testscripte zu verteilen, oder vorhandene zu aktualisieren. Die Test Ausführungs Schicht (TAuS) steuert die Ausführung von Testscripte und speichert die Ergebnisse der einzelnen Testschritte. Außerdem generiert sie, nach Abschluss eines erfolgreichen Tests, ein Testzertifikat. Die Test Adaptions Schicht (TAdS) stellt die Verbindung zwischen dem TAS und dem SUT her. Dazu lädt und verwaltet sie SUT Schnittstellen aus dem *Schnittstellenarchiv*. Dadurch können Tests leicht angepasst und schnell individuelle Tests für die gezielte Fehlersuche implementiert werden. Die Entworfene Architektur wurde darauf ausgelegt, möglichst flexibel zu sein, um sie leicht erweitern und anpassen zu können. So können auch neue Anforderungen, die in Zukunft an das TAS gestellt werden, leicht umgesetzt werden. Die entworfene Architektur soll als Grundlage für das neue TAS dienen, das eine umfangreichere Automatisierung des EOL Prozesses erlaubt, um so diesen Test zu beschleunigen, zuverlässiger und die Ergebnisse besser reproduzierbar zu machen. Die Evaluation, anhand des entwickelten Prototyps hat ergeben, dass nur ein Teil der Anforderungen bisher erfüllt werden. Der Prototyp ist in der Lage Testscripte zu laden und auszuführen. Basierend darauf wurde erfasst, was noch getan werden muss, damit das TAS voll Funktionsfähig ist. Die wichtigsten Punkte sind fehlende Schnittstellen, um die das TAS noch erweitert werden muss und Probleme mit der Zuverlässigkeit der bereits vorhandenen Schnittstellen. Wobei diese Probleme außerhalb des TAS liegen. Prozesse für die Erfassung and Schnittstellen zur Anbindung externer Datenquellen wurden in der Architektur angedacht und vorbereitet, jedoch noch nicht vollständig implementiert.

7.3 Weitere Arbeiten

Der EOL Prozess wurde noch nicht vollständig auf das neue TAS übertragen. Es muss, basierend auf den Anforderungen an den Testprozess ein neuer Testprozess entworfen werden, der möglichst gut automatisierbar ist. Dieses Thema würde sich, meiner Einschätzung nach, für eine fortführende Bachelor Arbeit eigenen. Für viele der Testschritte müssen spezielle Hard- und Softwareschnittstellen entworfen und konstruiert werden. Weiter muss das TAS noch in Jenkins eingepflegt werden. Weitergehend kann hier eine Struktur zur Verwaltung verschiedener Test Versionen etabliert werden. Die geplante Anbindung an

das Enterprise Resourcen Planning (ERP) System fehlt. Hier muss noch geklärt werden, welche Daten von welchem System benötigt werden und ein Prozess für den Austausch dieser etabliert werden. Auch dies könnte sich für eine Bachelor Arbeit eignen. Die entworfene Architektur ist modular Ausgelegt, daher kann sie auch für andere Testarten adaptiert werden. So könnte weiter Untersucht werden, welche Entwicklungs- und Produktionsprozesse um automatisierte Tests ergänzt werden können. So könnte das TAS in eine Continuous Integration und Continous Deployment Pipeline integriert werden, um neue Firmware Versionen direkt auf echter Hardware zu testen.

8 Bilder, Tabellen und Listings

8.1 Bilder

8.1.1 Ui Views

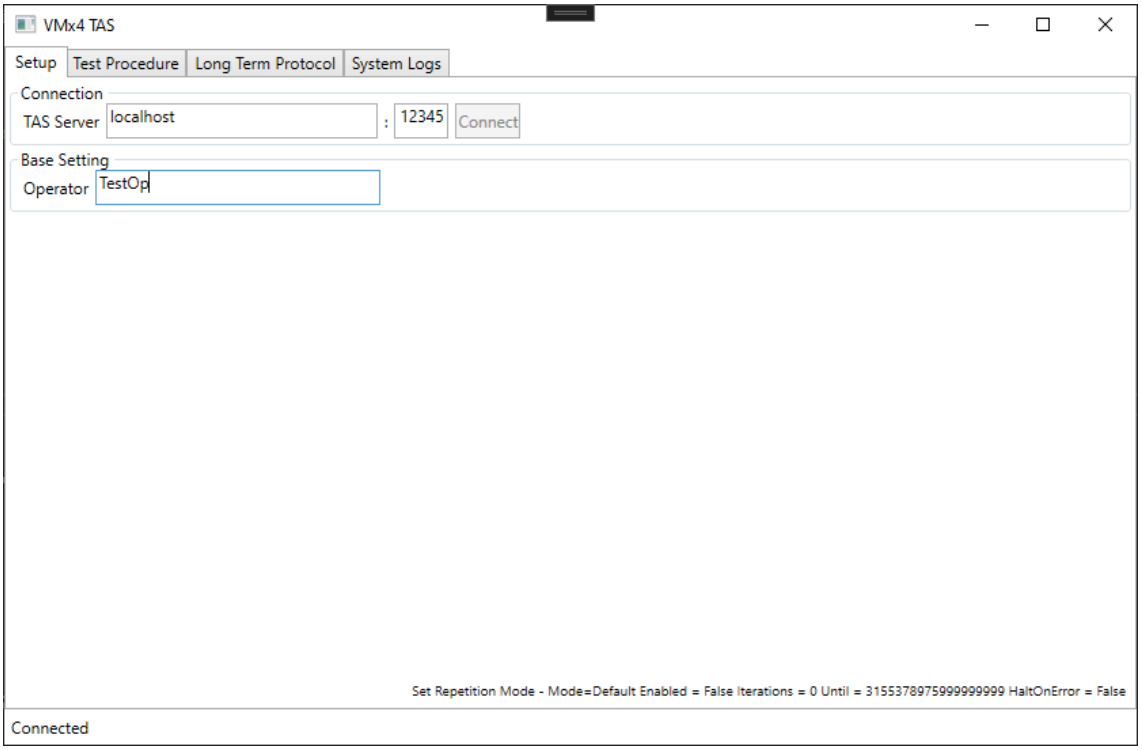


Abbildung 8.1: Setup-Tab

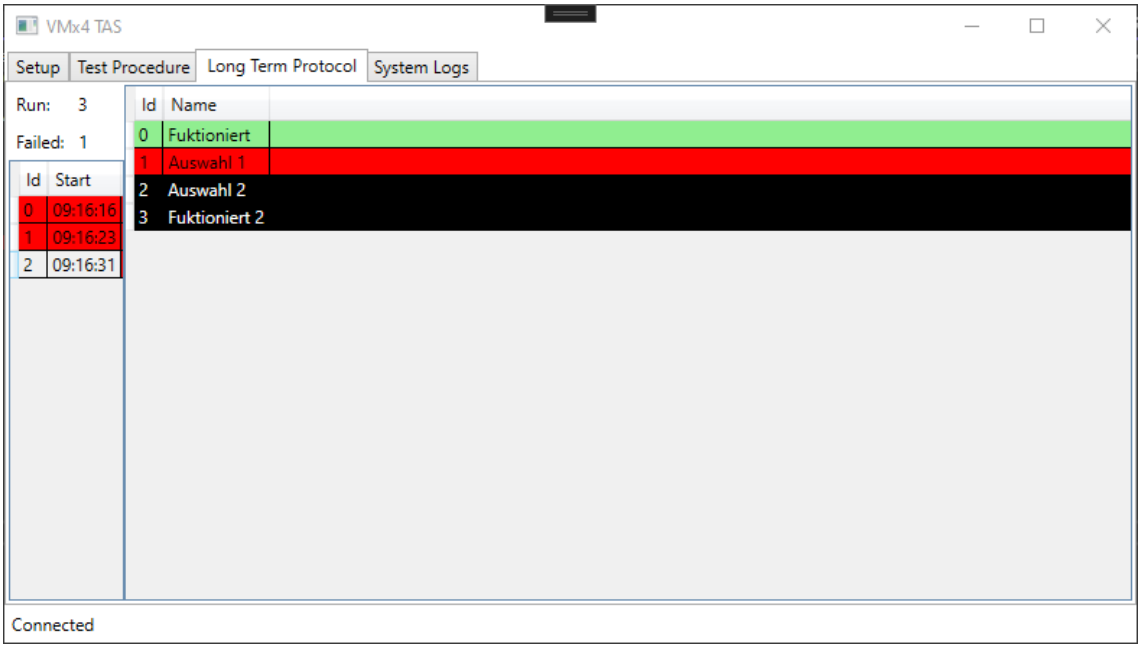


Abbildung 8.2: Langzeit Testprotokoll-Tab

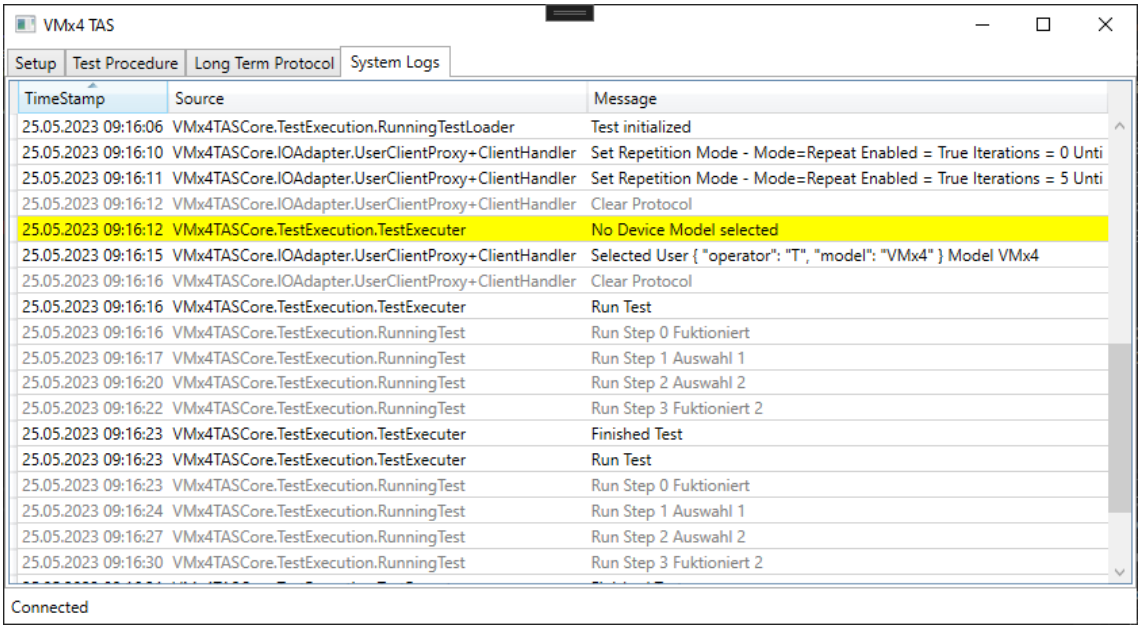


Abbildung 8.3: System Log-Tab

8.2 Listings

8.2.1 Beispiel Tests

Listing 8.1: EOL Demo Test

```
public class EOLDemo : TestBase
{
    public IDemoInterface DemoInterface;
    public IActuatorInterface Actuator;
    public IDisplayInterface Display;
    public IDoorContactInterface DoorContact;

    [TestBaseInformation("Auftragsnummer","")]
    public int OrderNumber;

    string productionNumber;

    private bool _generateCertificate;

    public EOLDemo() : base("EOL Demo", "Erster Test für das neue TAS")
    {

    }

    #region Setup
    public override void Init()
    {
        DoorContact.OpenStateChanged += DoorContact_OpenStateChanged;
    }
    public override void Finished(TestResult result)
    {
        if (result == TestResult.Success && _generateCertificate)
            GenerateCertificate();
    }

    private void DoorContact_OpenStateChanged(object? sender, bool e)
    {
        if(!e)
            Abort();
    }
    #endregion
    [TestStep("Daten Validierung","Validiere Basisdaten")]
    public TestResult DatenValidierung(CancellationToken cancel)
    {
        if (OrderNumber == 0)
        {
            _logger.Error("No Ordernumber set");
            return TestResult.TechnicalFailure;
        }
```

```
    SetOrderNumber(OrderNumber.ToString());

    return TestResult.Success;
}

[TestStep("Daten Erfassung","Erfasse Seriennummer")]
public TestResult DatenErfassung(CancellationToken cancel)
{
    productionNumber = RequestUserInput("Fertigungsnummer", new
        Regex("^\\d{6}$"));
    string serialNumber = RequestUserInput("Seriennummer", new
        Regex("^\\d{6}$"));
    SetSerialNumber(serialNumber);
    return TestResult.Success;
}

[TestStep("Initialisiere Gerät","Stelle sicher, dass das Gerät im
    korrekten Startzustand ist")]
public TestResult InitialisiereGerät(CancellationToken cancel)
{
    SUTStateManager.Reset().Wait(5000);

    if (SUTStateManager.DeviceActive)
    {
        _logger.Warn("Device not available");
        return TestResult.Success;
    }
    else
    {
        _logger.Warn("Device not available");
        return TestResult.Failed;
    }
}

[TestStep("Test Schritt 1","Einfache Demo")]
public TestResult TestSchritt1(CancellationToken cancel)
{
    Console.WriteLine("Testschritt 1");
    DemoInterface.DemoAction();
    Actuator.Extend("Options");
    Wait(0, 5, 0);
    Actuator.Retract("Options");

    return TestResult.Success;
}
[TestStep("Test Schritt 2","Ander Demo")]
```

```
    public TestResult TestSchritt2(CancellationToken cancel)
    {
        Console.WriteLine("Testschritt 2");
        string ergebnis = RequestUserInput("Nummer eingeben:", new
            System.Text.RegularExpressions.Regex("\\d+"));
        _generateCertificate = RequestUserConfirmation("Passt alles?");
        Wait(0, 5, 0);

        return TestResult.Success;
    }

    public override void Dispose()
    {
        DoorContact.OpenStateChanged -= DoorContact_OpenStateChanged;
    }
}
```

Listing 8.2: Beispiel Test

```
internal class RequiredTests : TestBase
{
    public IActuatorInterface Actuator;
    public IDisplayInterface Display;
    public IDoorContactInterface DoorContact;
    public IWLanInterface WLan;

    public RequiredTests() : base("Required Tests", "Test for Bachelor
        Thesis")
    {
    }
    [TestStep("Setup Device","")]
    public TestResult Setup()
    {
        SUTStateManager.PowerOn().Wait(5000);
        if (SUTStateManager.DeviceActive)
        {
            _logger.Warn("Device not available");
            return TestResult.Success;
        }
        else
        {
            _logger.Warn("Device not available");
            return TestResult.Failed;
        }
    }
    [TestStep("Change Mode","")]
```

```
public TestResult ChangeMode(CancellationToken cancel)
{
    //Goto Mode Menue
    int maxCycles = 3;
    bool foundState = false;
    do
    {
        Actuator.Push("Options", new TimeSpan(0, 0, 0, 0, 500));
        foundState = Display.AwaitDisplayState("Status LEDs", new
            byte[] { 0, 1, 1 }, new TimeSpan(0, 0, 2));
        maxCycles--;
    }
    while (maxCycles >= 0 && !foundState &&
        !cancel.IsCancellationRequested);

    if (cancel.IsCancellationRequested)
        return TestResult.Aborted;

    if(!foundState)
    {
        _logger.Warn("Couldn't find Mode Menue");
    }

    //Change mode to High Power
    maxCycles = 3;
    bool foundMode = false;
    do
    {
        Actuator.Push("Plus", new TimeSpan(0, 0, 0, 500));
        foundMode = Display.AwaitDisplayState("Main Display", new
            byte[] { 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            new TimeSpan(0, 0, 1));
        maxCycles--;
    }
    while(maxCycles >= 0 && !foundState &&
        !cancel.IsCancellationRequested);

    if (cancel.IsCancellationRequested)
        return TestResult.Aborted;

    Wait(0, 5, 0);

    return foundState && foundMode ? TestResult.Success :
        TestResult.Failed;
}
[TestStep("Test WLan","")]
```

```
public TestResult TestWLan(CancellationToken cancel)
{
    //Goto Connectivity Menue
    bool foundMenue1 = false;
    //Cycle through Menue options
    for(int maxCycles = 3; maxCycles >= 0 && !foundMenue1 &&
        !cancel.IsCancellationRequested; maxCycles--)
    {
        Actuator.Push("Options", new TimeSpan(0, 0, 0, 0, 500));
        //Check if in Connectivity Menue
        foundMenue1 = Display.AwaitDisplayState("Status LEDs", new
            byte[] { 1, 0, 0 }, new TimeSpan(0, 0, 2));
        maxCycles--;
    }

    if(cancel.IsCancellationRequested)
        return TestResult.Aborted;

    if (!foundMenue1)
    {
        _logger.Warn("Couldn't find Connectivity Menue");
    }

    //Cycle through connectivity options
    bool foundBlinking = false;
    for(int maxCycles = 2; maxCycles >= 0 && !foundBlinking &&
        !cancel.IsCancellationRequested; maxCycles--)
    {
        Actuator.Push("Power", new TimeSpan(0, 0, 0, 0, 500));
        //Check if Power LED is blinking (WLan active)
        bool foundOnState = Display.AwaitDisplayState("Status LEDs",
            new byte[] { 1, 0, 1 }, new TimeSpan(0, 0, 1));
        bool foundOffState = Display.AwaitDisplayState("Status LEDs",
            new byte[] { 1, 0, 0 }, new TimeSpan(0, 0, 1));
        bool found2ndOnState = Display.AwaitDisplayState("Status
            LEDs", new byte[] { 1, 0, 1 }, new TimeSpan(0, 0, 1));

        foundBlinking = foundOnState && foundOffState &&
            found2ndOnState;
    }

    if (cancel.IsCancellationRequested)
        return TestResult.Aborted;

    if (!foundBlinking)
    {
```

```
    _logger.Warn("Couldn't detect wireless mode");
}

Wait(0, 2, 0);

//Test if WLan available and working
bool wlanWorking = false;
string[] available = WLan.ScanForWLans();
if(!available.Contains(CurrentDeviceWLan))
{
    wlanWorking = WLan.TestWLan(CurrentDeviceWLan);
    if(!wlanWorking)
    {
        _logger.Warn("Device WLan not responding");
    }
}
else
{
    _logger.Warn("Couldn't find Device WLan");
}

//Goto Connectivity Menue
bool foundMenue2 = false;
for (int maxCycles = 3; maxCycles >= 0 && !foundBlinking &&
    !cancel.IsCancellationRequested; maxCycles--)
{
    Actuator.Push("Options", new TimeSpan(0, 0, 0, 0, 500));
    foundMenue2 = AsyncContext.Run(() =>
        Display.AwaitDisplayState("Status LEDs", new byte[] { 1,
        0, 0 }, new TimeSpan(0, 0, 2)));
    maxCycles--;
}

if (cancel.IsCancellationRequested)
    return TestResult.Aborted;

if (!foundMenue2)
{
    _logger.Warn("Couldn't find Connectivity Menue");
}

//Disable WLan
bool foundLEDOff = false;
for (int maxCycles = 3; maxCycles >= 0 && !foundBlinking &&
    !cancel.IsCancellationRequested; maxCycles--)
{
```

```
            Actuator.Push("Power", new TimeSpan(0, 0, 0, 0, 500));
            bool foundOffState = AsyncContext.Run(() =>
               Display.AwaitDisplayState("Status LEDs", new byte[] { 1,
               0, 0 }, new TimeSpan(0, 0, 1)));
            bool foundOnState = AsyncContext.Run(() =>
               Display.AwaitDisplayState("Status LEDs", new byte[] { 1,
               0, 1 }, new TimeSpan(0, 0, 2)));

            foundLEDOff = !foundOnState && foundOffState;
         }

         return (foundMenue1 && foundBlinking && wlanWorking &&
            foundMenue2 && foundLEDOff) ? TestResult.Success :
            TestResult.Failed;
      }

      public override void Dispose()
      {
      }

      public override void Finished(TestResult result)
      {
      }

      public override void Init()
      {
      }
}
```

Abkürzungen

AE Beschreiben auf welche Art Probleme beim Entwurf eine Softwarearchitektur gelöst wurden.
ATML Automatic Test Markup Language. Formale, auf XML Basierende Beschreibungssprache für die Definition und den Austausch von Testabläufen

CPS Cyber Physical System. Technisches System, das Software und Hardwarekomponenten vereint

DLL-Datei Dynamic Linked Library. Dateiformat für die Speicherung von dynamischen Programmbibliotheken unter Windows

EOL End of Line. Nach Abschluss der Produktion, vor der Auslieferung
ERP Enterprise Resourcen Planning. Verwaltung von Aufgaben, Personal, Ressourcen, Kapital, Betriebsmitteln und Material in einem Unternehmen

GTA Generische Testsystem-Architektur. Architektur für ein Testsystem nach [PBB+, 26]

IREB International Requirements Engineering Board e.V. Verein mit Sitz in Fürth in Deutschland, der Material, Schulungen und Zertifizierungen zum Thema Requirements Engineering anbietet

JSON JavaScript Object Notation. Datenformat für den Datenaustausch zwischen Anwendungen

MVC Model View Controller. Strategie für den Aufbau von Frontends
MVP Model View Presenter. Strategie für den Aufbau von Frontends
MVVM Model View Viewmodel. Strategie für den Aufbau von Frontends

RE Requirement Engineering. Das disziplinierte und systematische Vorgehen bei der Erfassung und Verwaltung von Anforderungen

SBST Search Based Software Testing. Verfahren zur Testerstellung, das Optimierungsalgorithmen verwendet, um nach Grenzfällen zu suchen
SUT System under Test. Hard- oder Softwaresystem das getestet wird

TAA Testautomatisierungsarchitektur. Architektur für ein TAS
TAdS Test Adaptions Schicht. Schicht eines Test Automatisierungssystem, die Schnittstellen zum SUT bereitstellt

TAL Testautomatisierungslösung. Gesamtheit aus TAS und anhängenden Prozessen

TAM Technology Acceptance Model. Eine Theorie, wie erfasst und bewertet werden kann, wie Nutzer Technologie nutzen und akzeptieren.

TAS Testautomatisierungssystem. Gesamtheit der Hard- und Software eines System zur Automatisierung von Tests

TAuS Test Ausführungs Schicht. Schicht eines Test Automatisierungssystem, die Testabläufe durchführt

TDS Test Definitions Schicht. Schicht eines Test Automatisierungssystem, die es erlaubt Abläufe für Tests zu definieren

TGS Test Generierungs Schicht. Schicht eines Test Automatisierungssystem, die das Anlegen von Tests ermöglicht

TIS Test Implementierungs Schicht. Schicht eines Test Automatisierungssystem, die das Anlegen und Implementieren von Tests ermöglicht

UI User Interface. Schnittstelle zwischen Programm und Benutzer

UML Unified Modeling Language. Grafische Modellierungssprache für die Visualisierung, Spezifizierung, Konstruktion und Dokumentation von Artefakten verteilter Objekt Systeme

UTA Universelle Testsystem-Architektur. Architektur für ein Testsystem nach [KD09]

WPF Windows Presentation Foundation. Framework von Microsoft für die Erstellung von Grafischen Nutzeroberflächen mit C#

XML Extensible Markup Language. Auszeichnungssprache für die strukturierte Darstellung hierarchischer Daten in Maschinen und Menschen lesbarer Form

Glossar

Abas Enterprise Resource Planning Software der Firma abas Software GmbH

Bridge Pattern Entwurfsmuster, dass es erlaubt eine Abstraktion und eine zugehörige Implementierung zu trennen [GHJV94, 139].

Builder Pattern Entwurfsmuster, bei dem eine Klasse (Builder) Implementiert wird, die Methoden bereitstellt, um Objekte einer anderen Klasse (Product) zu erzeugen [GHJV94, 139]

Closed Loop Test Es wird zum Testen eine Feedback Schleife zwischen SUT und TAS erzeugt. Der Test reagiert auf Ausgaben des SUT und verwendet diese zur Steuerung des Testablaufs

Hook Schnittstelle zu einem System, die speziell implementiert wurde, um die Testbarkeit des System zu ermöglichen, oder zu verbessern

Jenkins Eine Open Source Software für die kontinuierliche Integration und Ausspielung von Software

Klimakammer Kammer zum erzeugen und aufrechterhalten einer Umgebung mit kontrollierter Temperatur und Luftfeuchtigkeit

log4net Portierung des Plattform übergreifenden Logging Frameworks log4j für C#/.Net

Magnetfeldzünder Zünder für Sprengsätze, der auf große, sich bewegende Metallobjekte reagiert. Kann auch durch das Magnetfeld eines Metalldetektors ausgelöst werden

Mock Simplifizierte Simulation eines Systems, oder einer Umgebung

Open Loop Test Das ausführen von Eingaben und auswerten der Ausgaben des SUT sind beim Test entkoppelt. Das TAS führt eine Vorgegebene Abfolge an Eingaben aus und vergleicht die Ausgaben mit einer Liste der erwarteten Ausgaben

Proxy Pattern Entwurfsmuster, bei dem eine Klasse (Proxy) den Zugriff auf eine Ressource steuert und abstrahiert [GHJV94, 262]

Qualitäts Merkmal Nicht funktionale Vorgaben, die ein System erfüllen muss

Randbedingung Vorgaben und Einschränkungen aus technischer oder organisatorischer Richtung die den möglichen Lösungsraum für die Software Architektur einschränken

Stakeholder*in Personen, Systeme, Prozesse, oder Organisationen, die Interesse an einem System haben

Strategy Software Entwurfsmuster das es erlaubt zur Laufzeit aus verschiedenen Lösungen für ein Problem zu wählen

String Datentyp der eine beliebig lange Zeichenkette speichert

Testbett Gesamtheit aller Test Schnittstellen, physisch oder digital, zu einem SUT

Testscript Formale Beschreibung eines Testablaufs

User Story Beschreibung einer Eigenschaft, oder Funktionalität, die ein Nutzer von einem System fordert

V-Modell Theoretisches Modell, das Beschreibt, welche Tests zu welchem Zeitpunkt in der Entwicklung eines Produkts durchgeführt werden sollten

VMx4 Tragbarer Metalldetektor der Firma Vallon für den Einsatz in der Kampfmittel Räumung

Literaturverzeichnis

[ABT10] Fredrik Abbors, Andreas Bäcklund, and Dragos Truscan. Matera - an integrated framework for model-based testing. In *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 321–328, 2010. doi:10.1109/ECBS.2010.46.

[AFS22] A. Wilson, F. Wedyan, and S. Omari. An empirical evaluation and comparison of the impact of mvvm and mvc gui driven application architectures on maintainability and testability. In *2022 International Conference on Intelligent Data Science Technologies and Applications (IDSTA)*, pages 101–108, 2022. doi:10.1109/IDSTA55301.2022.9923083.

[ANT92] Dennis A. Adams, R. Ryan Nelson, and Peter A. Todd. Perceived usefulness, ease of use, and usage of information technology: A replication. *MIS Q*, 16(2):227–247, 1992. doi:10.2307/249577.

[Bau21] Manfred Baumgärtner. *Basiswissen Testautomatisierung; Aus- und Weiterbildung zum ISTQB Advanced Level Specialist: Aus- und Weiterbildung zum ISTQB® Advanced Level Specialist – Certified Test Automation Engineer*. DPUNKT Verlag, 3., aktualisierte und überarbeitete auflage edition, 2021.

[Dep85] Department of Defense. Defense system software development, 1985.

[dG22] Andy de George. Data binding overview (wpf .net), 2022. URL: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/data/?view=netdesktop-7.0 [cited 24.05.2023].

[FM11] I. Fette and A. Melnikov. The websocket protocol: Rfc. *2070-1721*, (6455), 2011. URL: https://www.rfc-editor.org/rfc/rfc6455.txt.

[GHJV94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Boston, Mass., forty-fourth printing edition, 1994.

[GKRS20] Mahbouba Gharbi, Arne Koschel, Andreas Rausch, and Gernot Starke. *Basiswissen für Softwarearchitekten, 4th Edition: Aus- und Weiterbildung nach iSAQB-Standard zum Certified Professional for Software Architecture - Foundation Level*. Basiswissen. dpunkt, Heidelberg, 4th edition edition, 2020. URL: http://www.content-select.com/index.php?id=bib_view&ean=9783969100127 [cited 24.05.2023].

[Goo08] Protocol buffers, 2008. URL: https://protobuf.dev/ [cited 23.05.2023].

[GvSB22] Martin Glinz, Hans van Loenhound, Stefan Staal, and Stan Bühne. Handbuch für das cpre foundation level nach dem ireb-standard: Aus- und weiterbildung zum certified professional for requirements engineering (cpre) foundation level, 2022. URL: https://www.ireb.org/content/downloads/3-cpre-foundation-level-handbook/cpre_foundationlevel_handbook_de_v1.1.1.pdf [cited 13.03.2023].

[iee10] Ieee standard for automatic test markup language (atml) for exchanging automatic test equipment and test information via xml. *IEEE Std 1671-2010 (Revision of IEEE Std 1671-2006)*, 2010. doi:10.1109/IEEESTD.2011.5706290.

[Inf19] V-model xt bund, 2019. URL: https://www.cio.bund.de/SharedDocs/downloads/Webs/CIO/DE/digitaler-wandel/architekturen-standard/v_modell_xt_bund_pdf.pdf [cited 30.05.2023].

[ISO11] ISO Interanationale Organisation für Normung. Softwareengineering - qualitätskriterien und bewertung von softwareprodukte (square) - qualitätsmodell und leitlinien, 01.03.2011.

[iso16] Iso/iec/ieee international standard - software and systems engineering – software testing – part 5: Keyword-driven testing. *ISO/IEC/IEEE 29119-5 First edition 2016-11-15*, pages 1–69, 2016. doi:10.1109/IEEESTD.2016.7750539.

[KD09] Dmitry Korotkiy and Hendrik Dettmering. Universal test system architecture in mechatronics: An approach for systematization of today's existing test tools. *IEEE International Conference on Industrial Technology*, pages 1–5, 2009. doi:10.1109/ICIT.2009.4939560.

[Koh] Kohsuke Kawaguchi. Jenkins: Build great things at any scale. URL: https://www.jenkins.io/ [cited 16.06.2023].

[Lit21] Taurius Litvinavicius. *EXPLORING WINDOWS PRESENTATION FOUNDATION: With practical applications in .net 5*. Apress, [S.l.], 2021. doi:10.1007/978-1-4842-6637-3.

[log23] What is apache log4net™, 2023. URL: https://logging.apache.org/log4net/ [cited 25.05.2023].

[MCA22] How to: Load assemblies into an application domain, 2022. URL: https://learn.microsoft.com/en-us/dotnet/framework/app-domains/how-to-load-assemblies-into-an-application-domain [cited 23.05.2023].

[McM11] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, 2011. doi:10.1109/ICSTW.2011.100.

[MCT22] Cancellation in managed threads, 2022. URL: https://learn.microsoft.com/en-us/dotnet/standard/threading/cancellation-in-managed-threads [cited 23.05.2023].

[Obj17] Object Management Group. Omg® unified modeling language® (omg uml®), 2017. URL: https://www.omg.org/spec/UML/2.5/PDF [cited 14.03.2023].

[PBB+] Adrew Pollner, Backker Bryan, Armin Born, Mark Fewster, Jani Haukine, Raluca Popescu, and Ina Schieferdecker. Cerfitified tester advanced level syllabus testautomatisierungsentwickler. URL: https://www.german-testing-board.info/wp-content/uploads/2019/12/Advanced-Testautomatisierungsentwickler-Syllabus_DE_2019-12-16_Version_H.pdf [cited 30.03.2023].

[PC20] Daniel Persson Proos and Niklas Carlsson. Performance comparison of messaging protocols and serialization formats for digital twins in iov. In *2020 IFIP Networking Conference (Networking)*, pages 10–18, 2020. URL: https://dl.ifip.org/db/conf/networking/networking2020/1570620395.pdf.

[PR21] Klaus Pohl and Chris Rupp. *Basiswissen Requirements Engineering: Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Leven*. dpunkt.verlag, Heidelberg, 5., überarbeitete und aktualisierte auflage edition, 2021.

[QS13] M. Rizwan Jameel Qureshi and Fatima Sabir. A comparison of model view controller and model view presenter. *Science International-Lahore*, 25(1):7–9, 2013. URL: https://arxiv.org/pdf/1408.5786 [cited 24.05.2023].

[Roy87] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International*

Conference on Software Engineering, ICSE '87, page 328–338, Washington, DC, USA, 1987. IEEE Computer Society Press.

[SABB17] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. To mock or not to mock? an empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 402–412, 2017. doi:10.1109/MSR.2017.61.

[SBMH96] Elmar Sauerwein, Franz Bailom, Kurt Matzler, and Hans Hinterhuber. The kano model: How to delight your customers. *International Working Seminar on Production Economics*, 1, 1996.

[SGM09] Clemens Szyperski, Dominik W. Gruntz, and Stephan Murer. *Component software: Beyond object-oriented programming*. Addison-Wesley component software series. Addison-Wesley, London and Munich, 2. ed., [repr.] edition, 2009.

[SH23] Gernot Starke and Peter Hruschka. *arc42 in Aktion: Praktische Tipps zur Architekturdokumentation*. Hanser eLibrary. Hanser, München, 2., überarbeitete auflage edition, 2023. doi:10.3139/9783446465558.

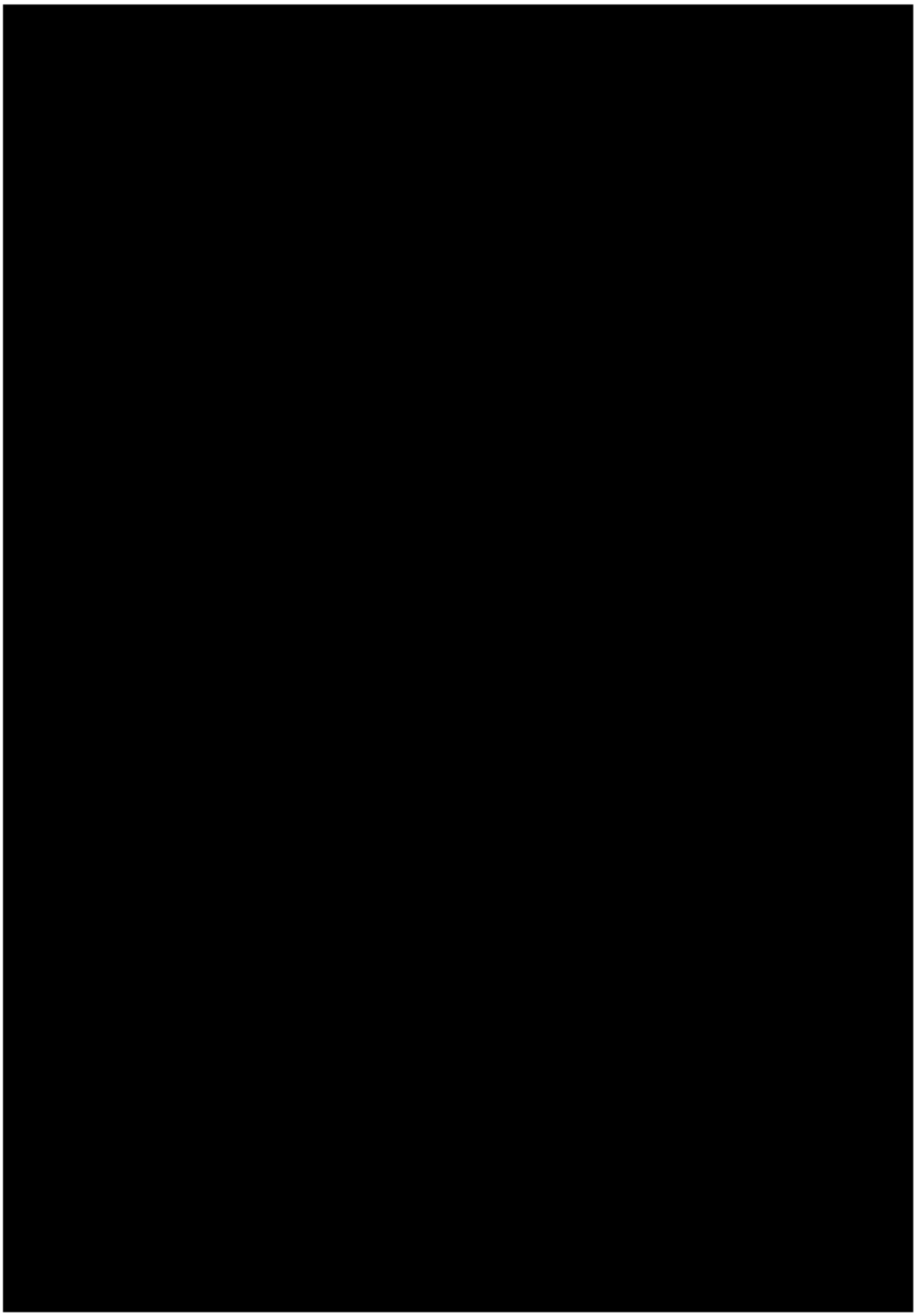
[SL19] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard*. iSQI-Reihe. dpunkt.verlag, Heidelberg, 6. überarbeitete und aktualisierte auflage edition, 2019.

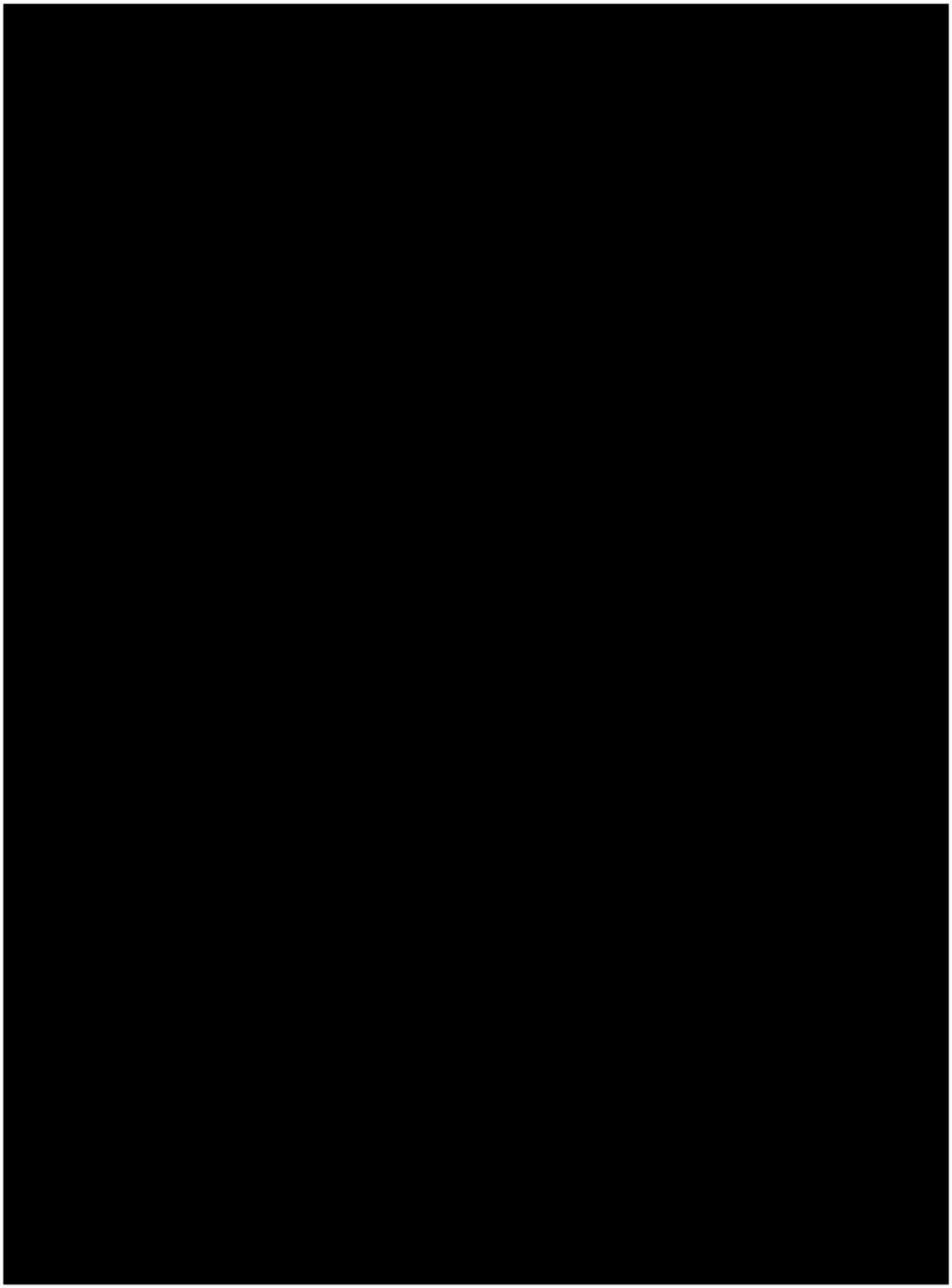
[Smi09] Smith Josh. Patterns - wpf apps with the model-view-viewmodel design pattern, 2009. URL: https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern [cited 23.03.2023].

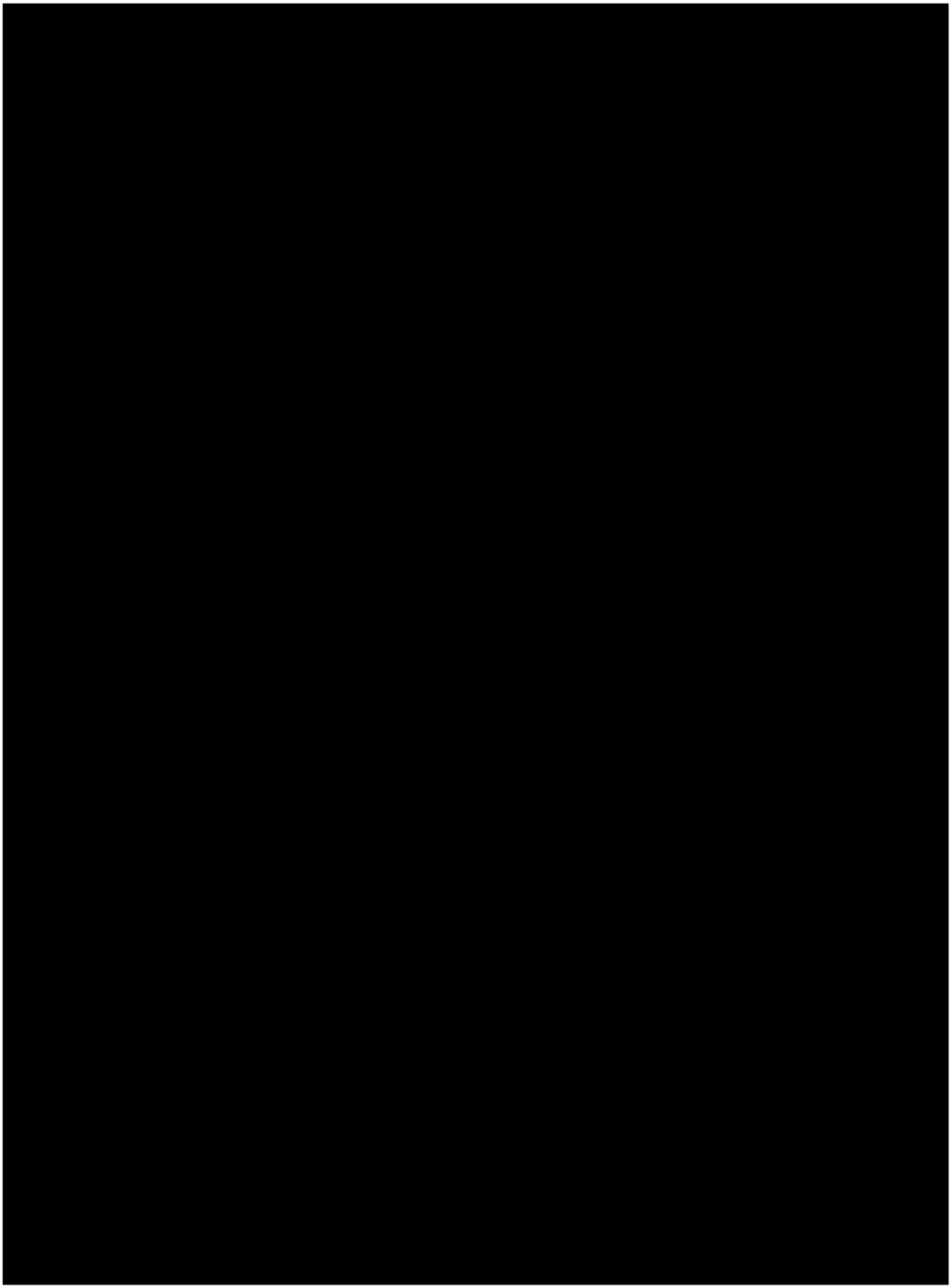
[Wit19] Frank Witte. *TESTMANAGEMENT UND SOFTWARETEST: Theoretische grundlagen und praktische umsetzung*. SPRINGER VIEWEG, [S.l.], 2019. doi:10.1007/978-3-658-25087-4.

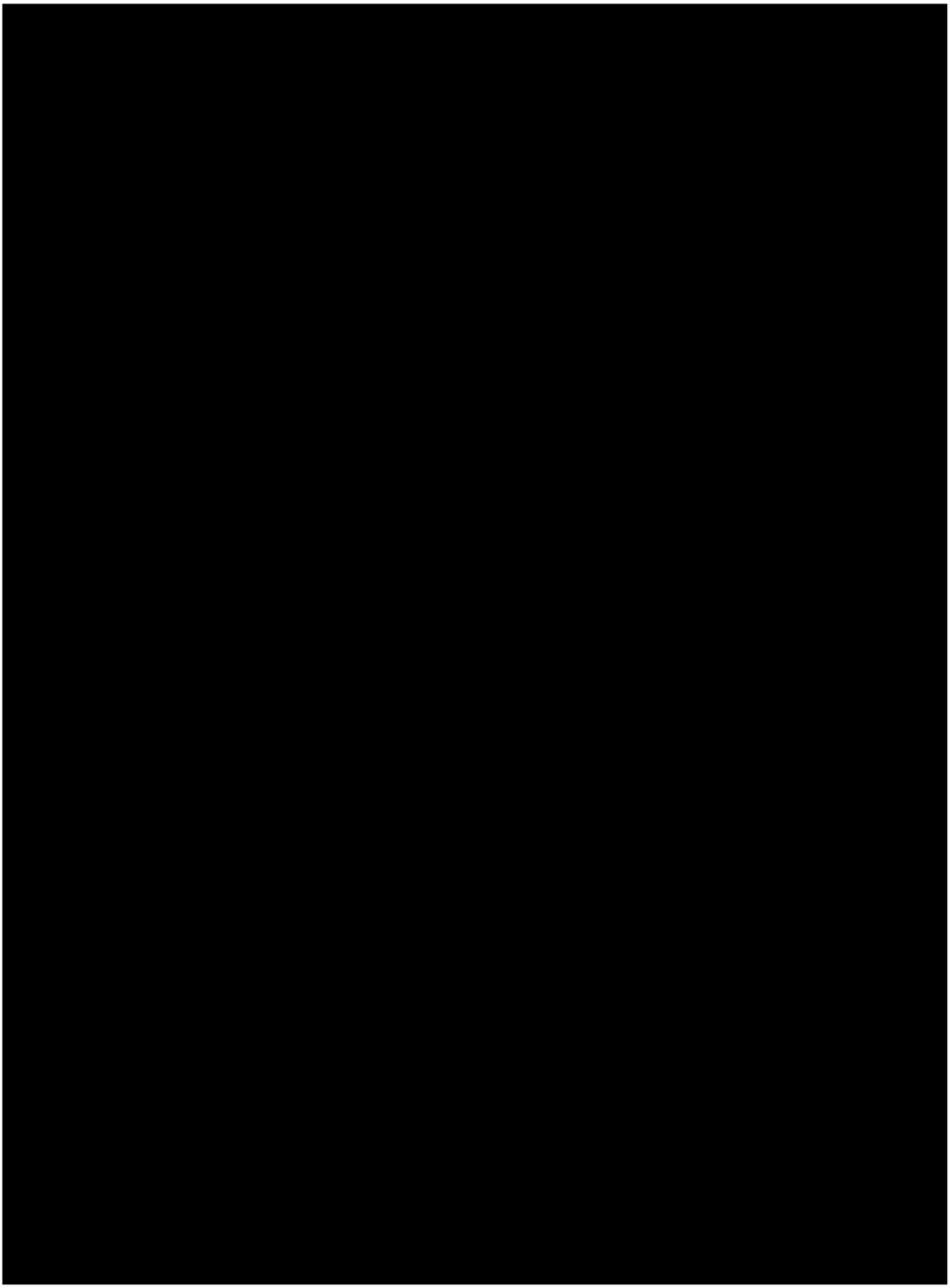
[ZGHY18] Xin Zhou, Xiaodong Gou, Tingting Huang, and Shunkun Yang. Review on testing of cyber physical systems: Methods and testbeds. *IEEE Access*, 6:52179–52194, 2018. doi:10.1109/ACCESS.2018.2869834.

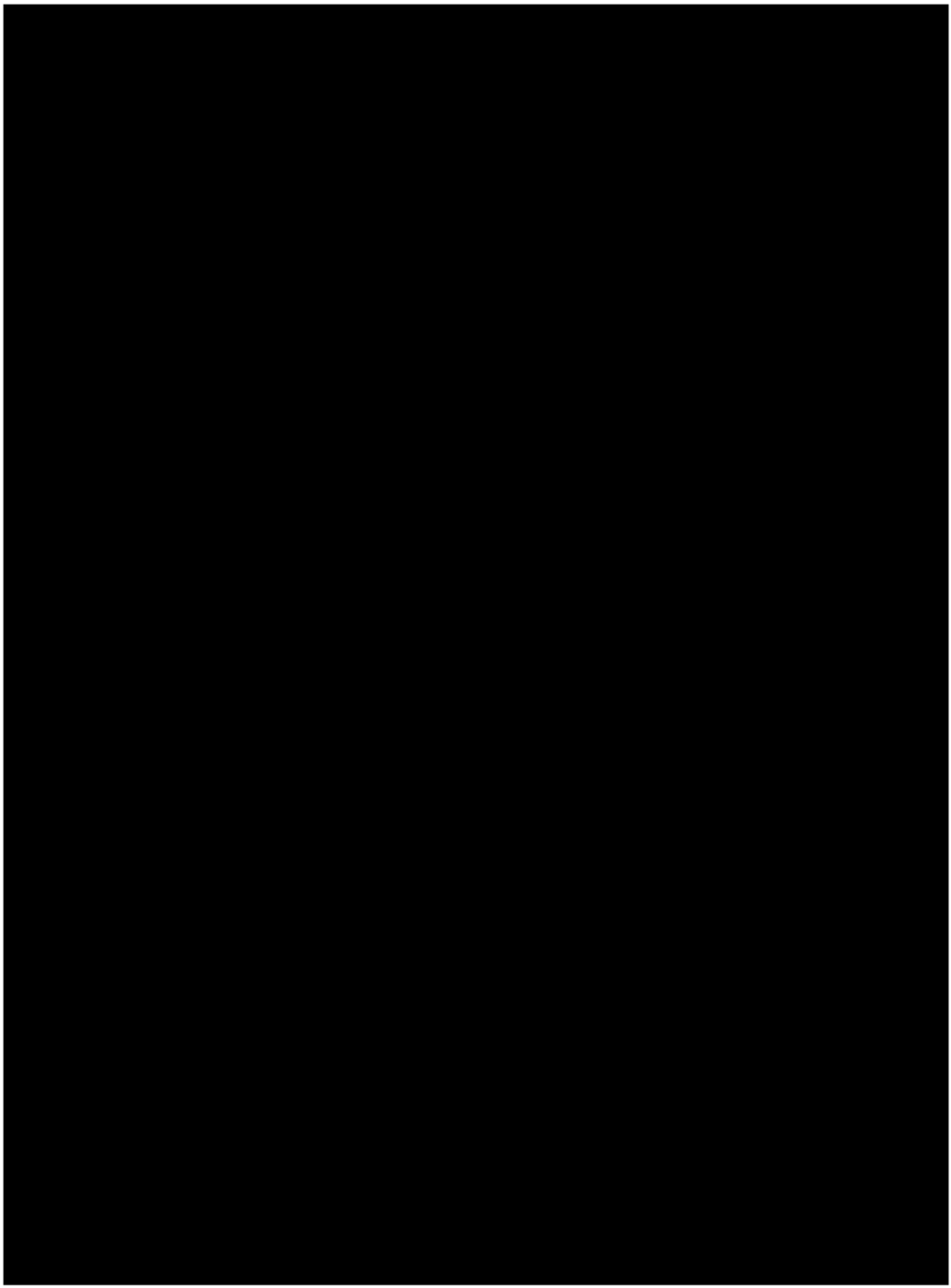
[Zör12] Stefan Zörner. *Softwarearchitekturen dokumentieren und kommunizieren: Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten*. Hanser, München, 2012. doi:10.3139/9783446431287.

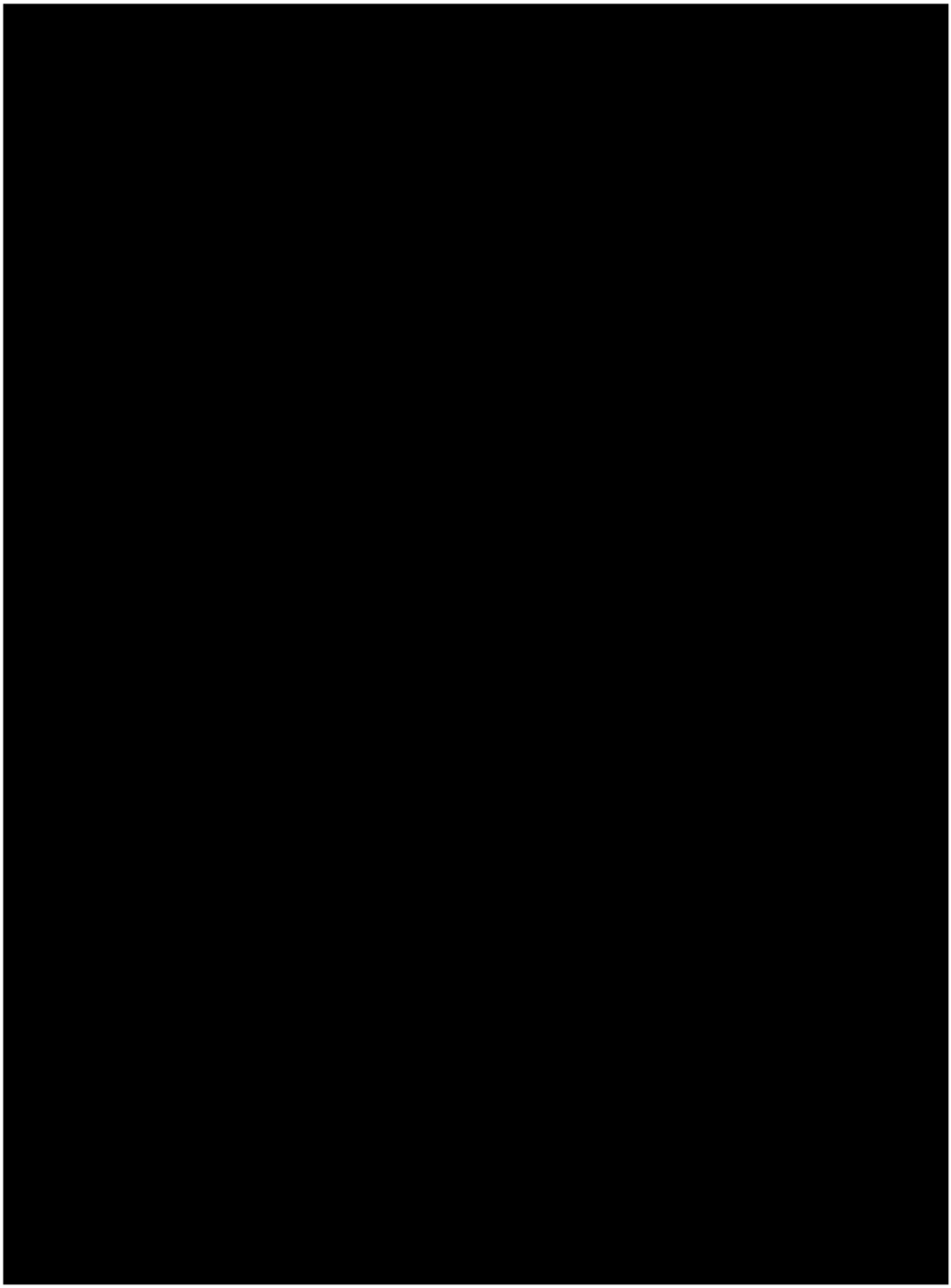


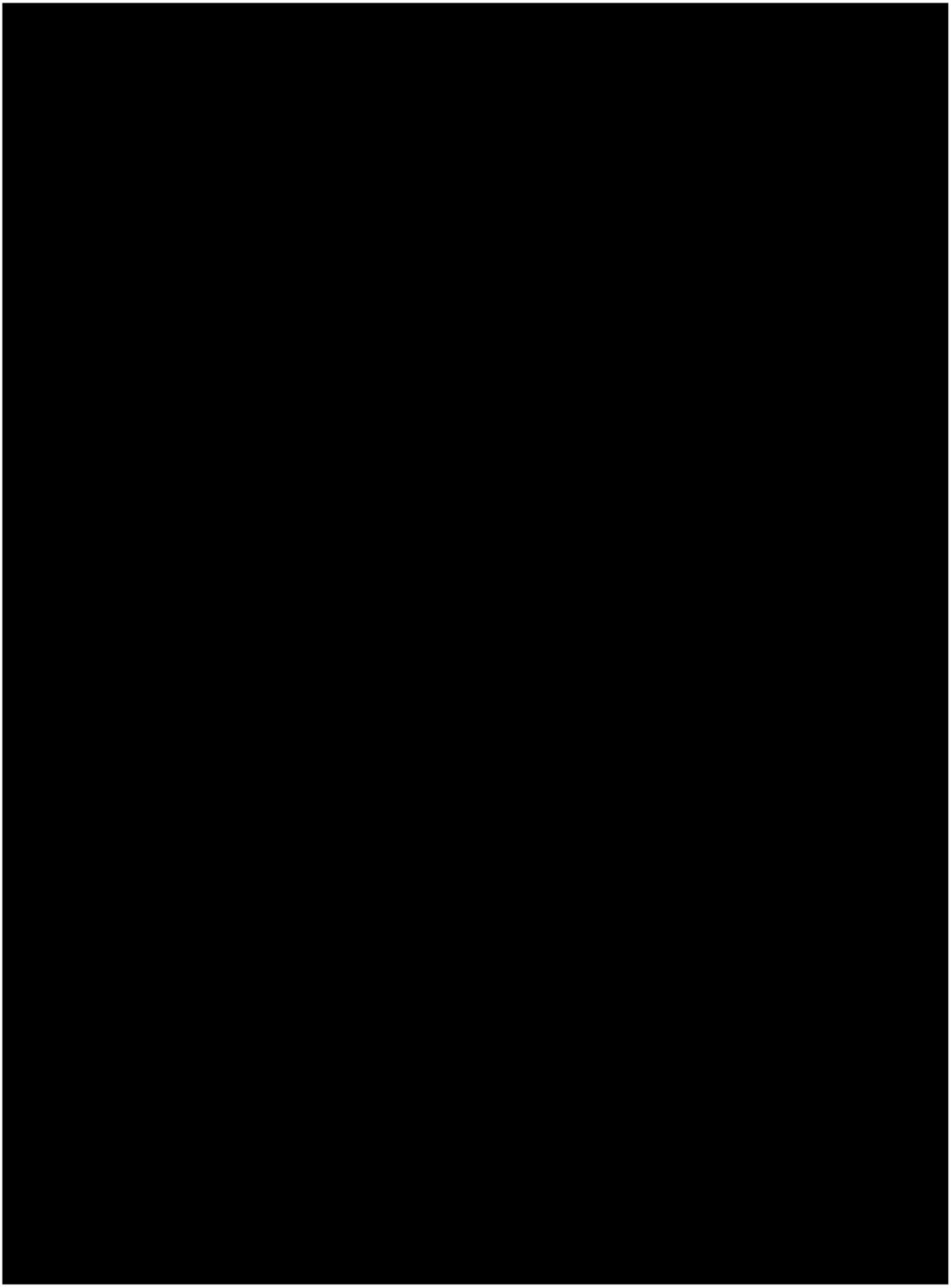


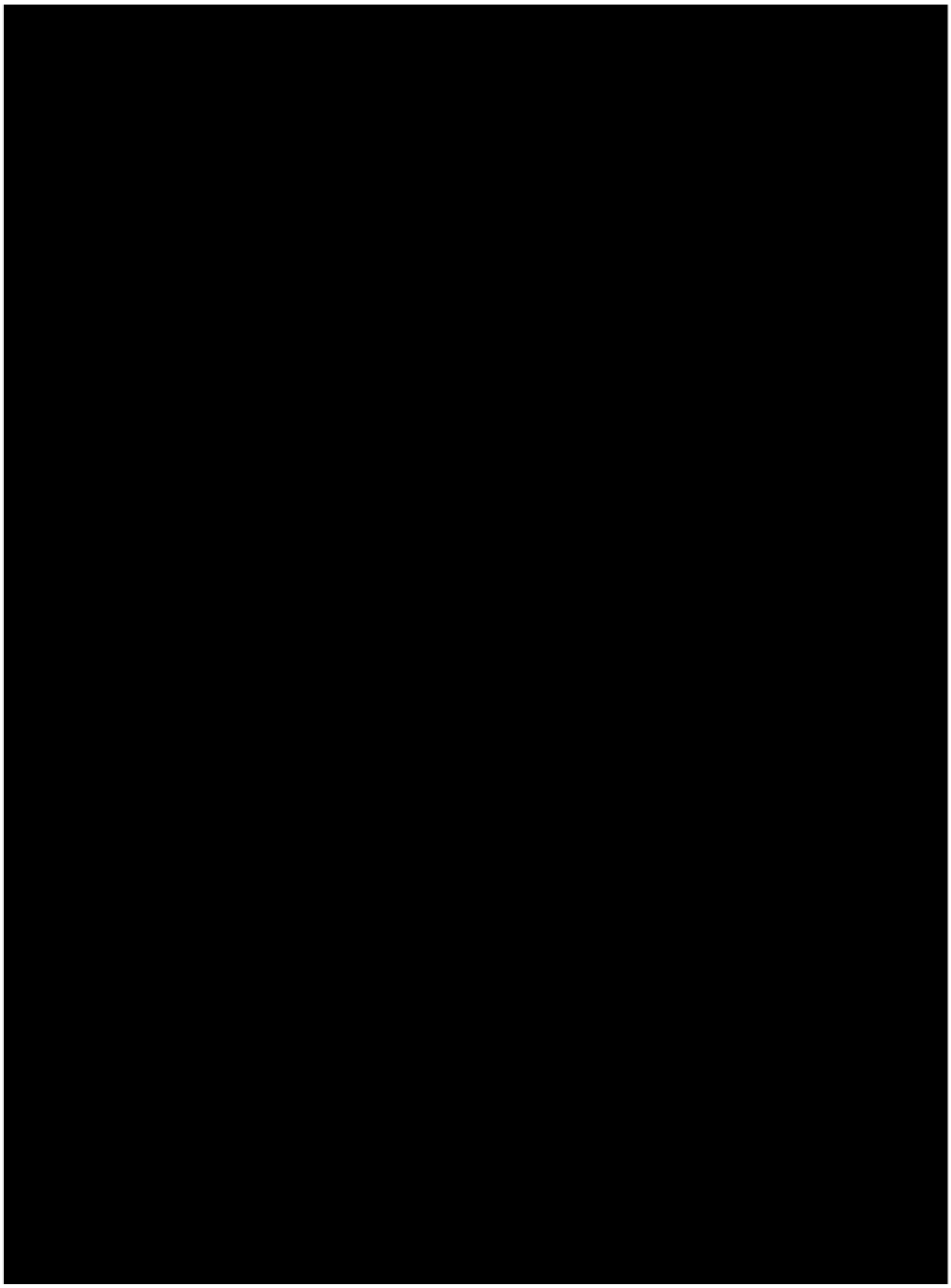


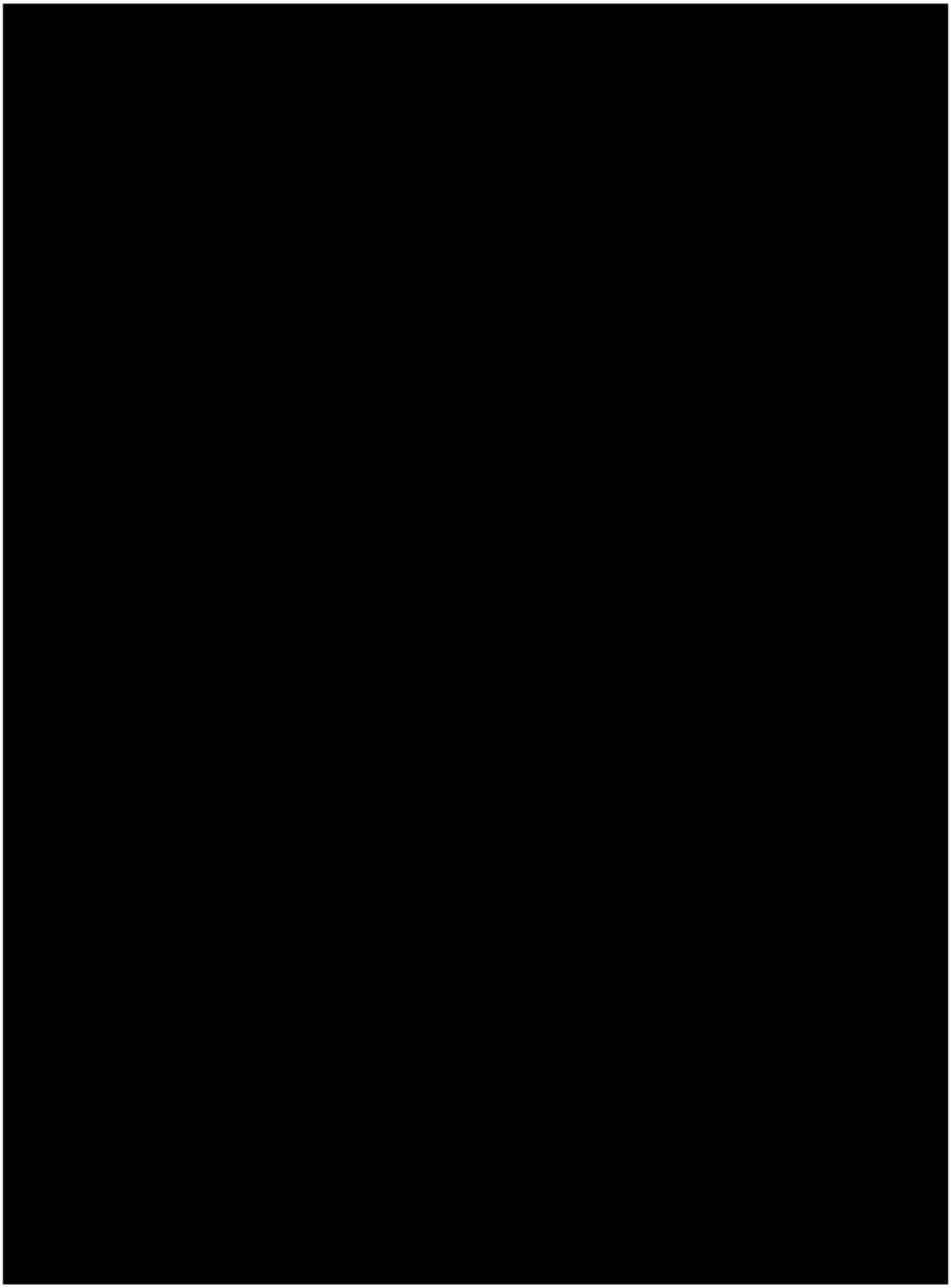


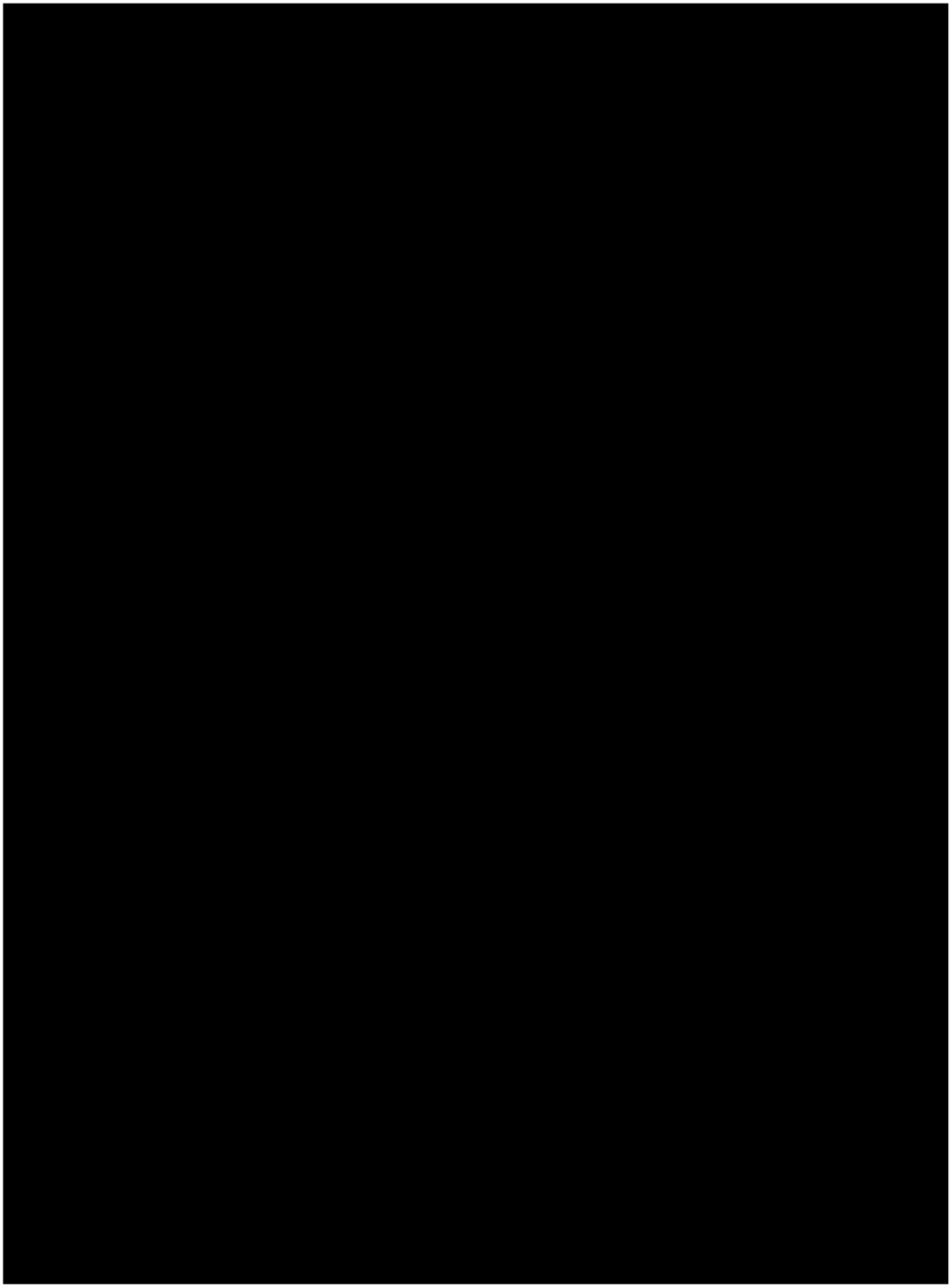


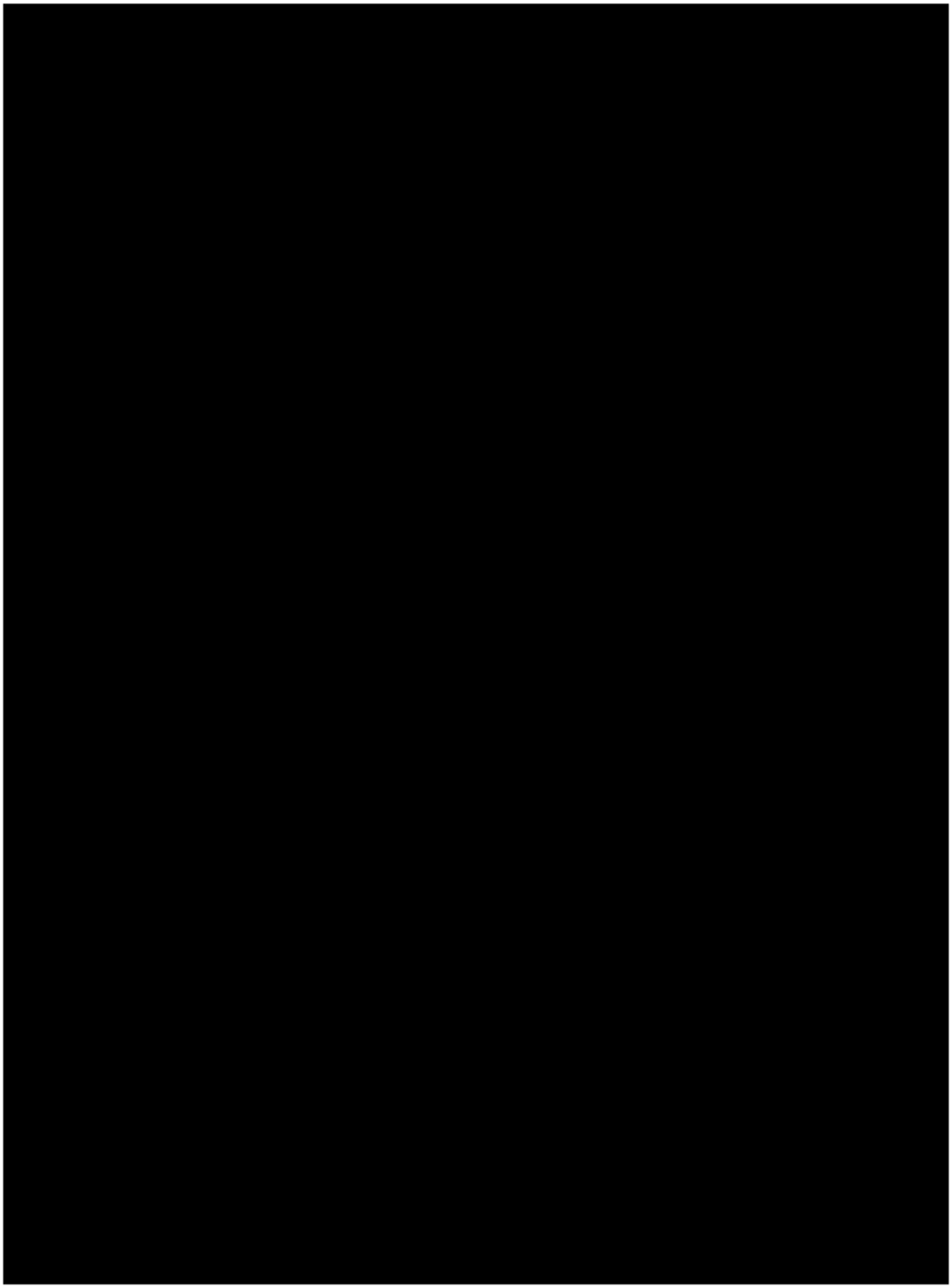


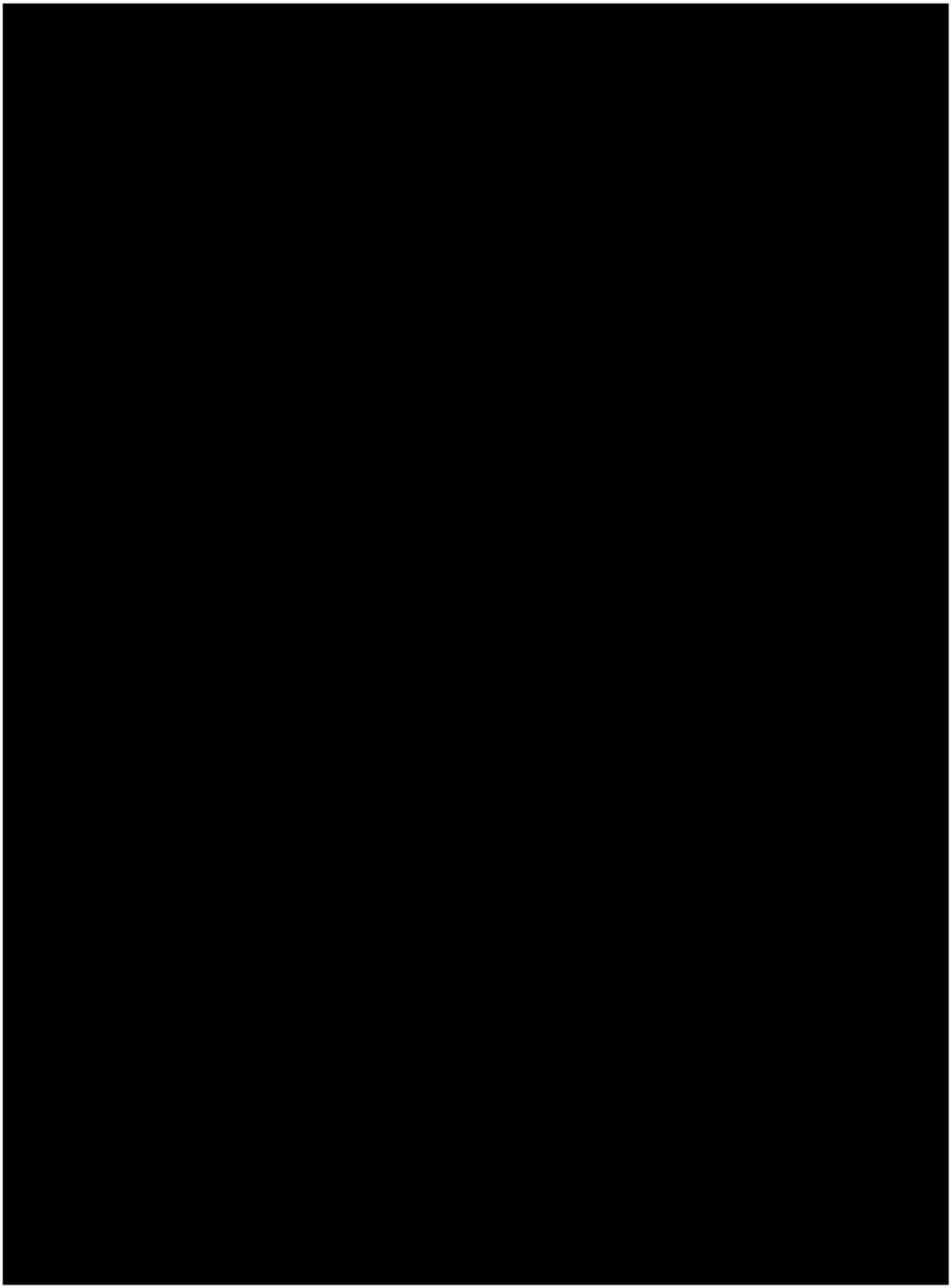


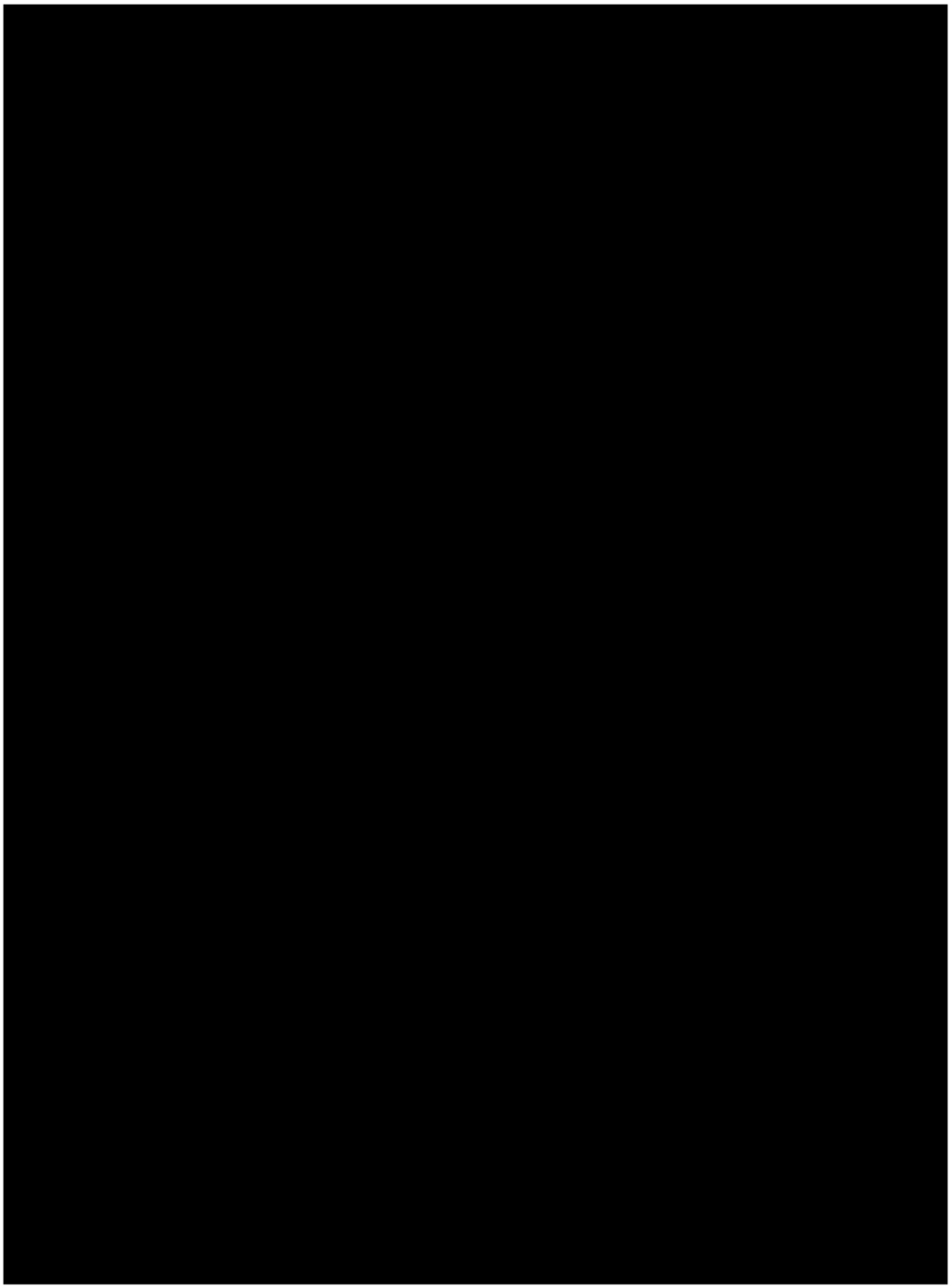


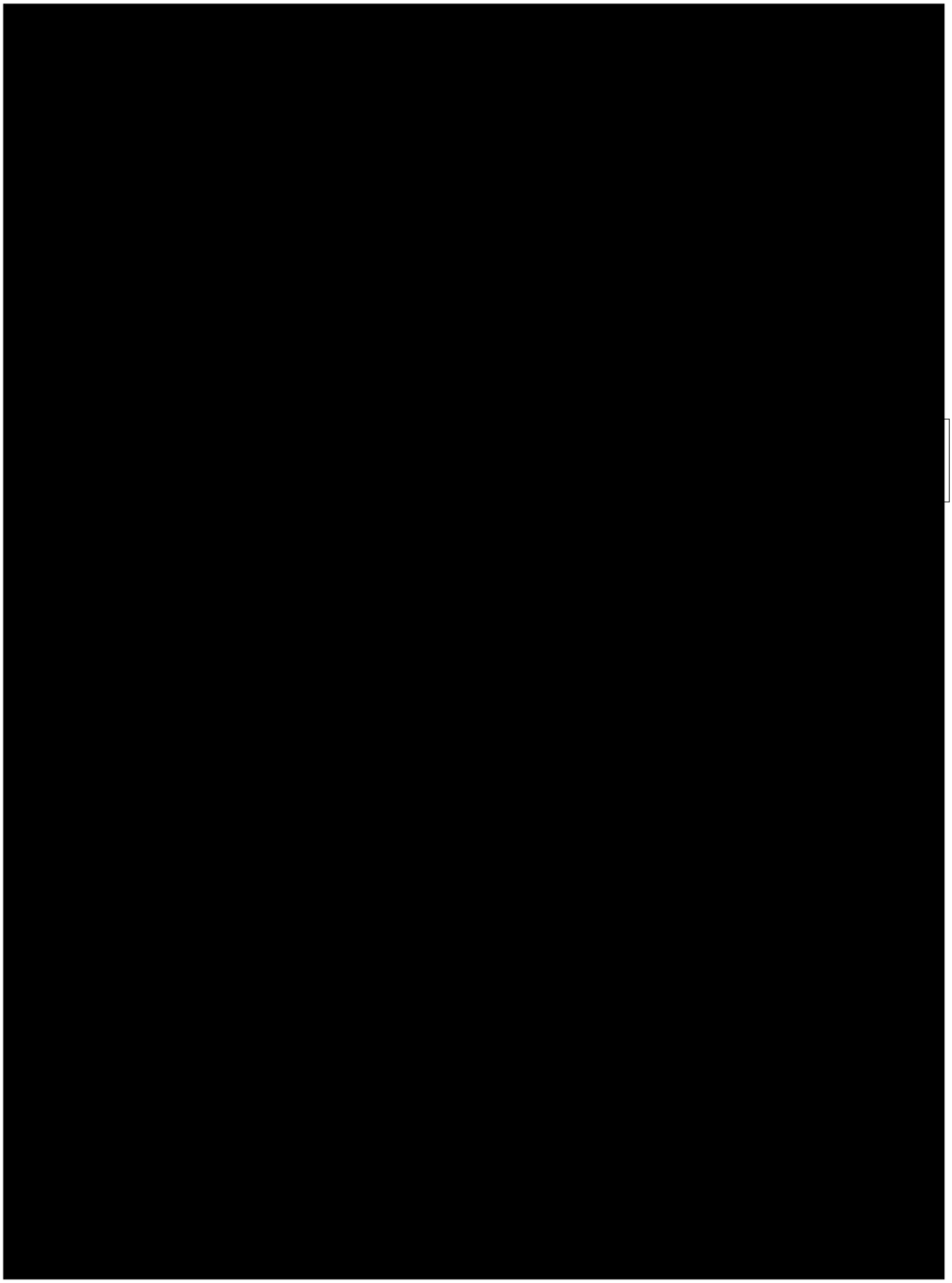


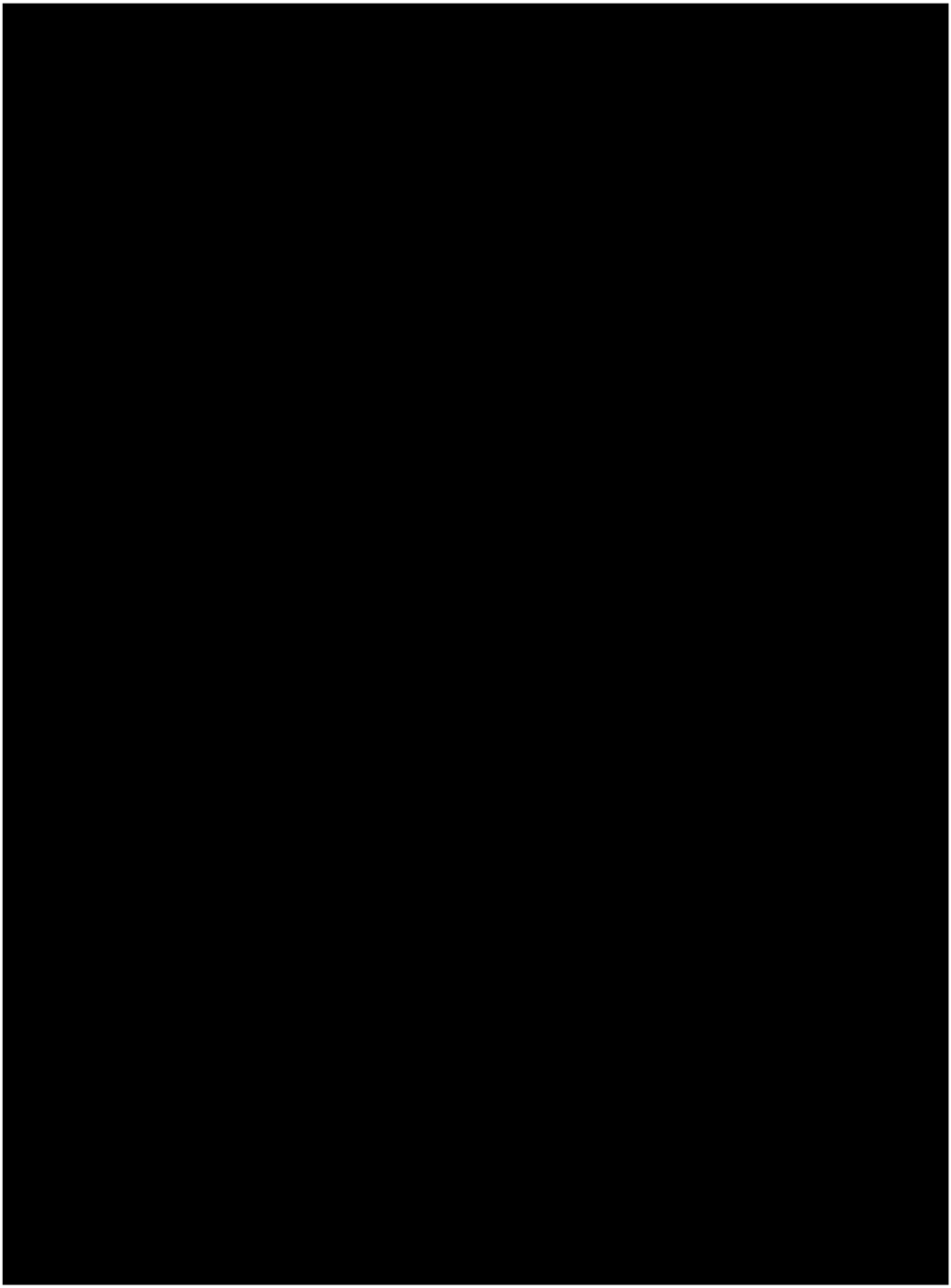


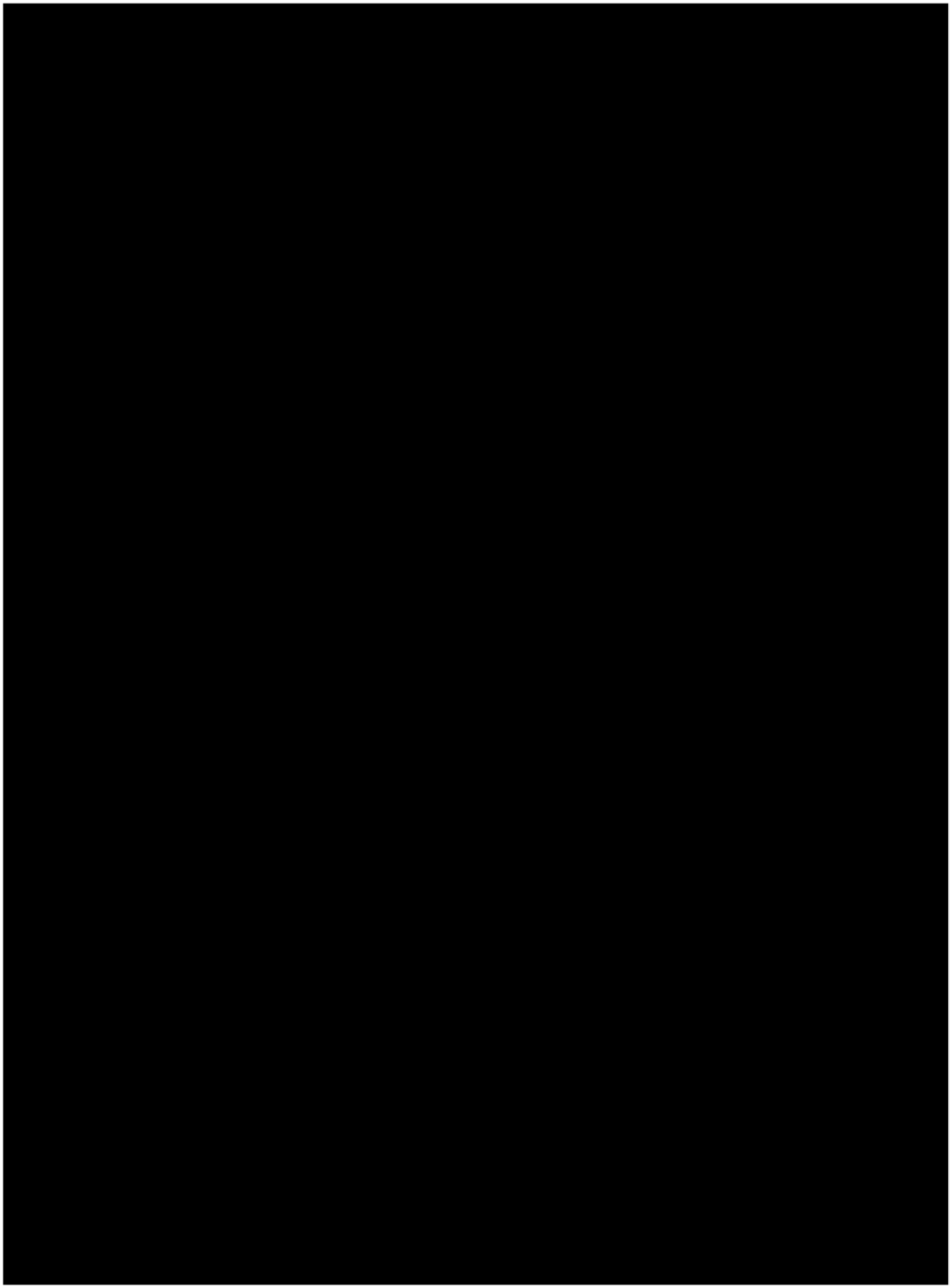


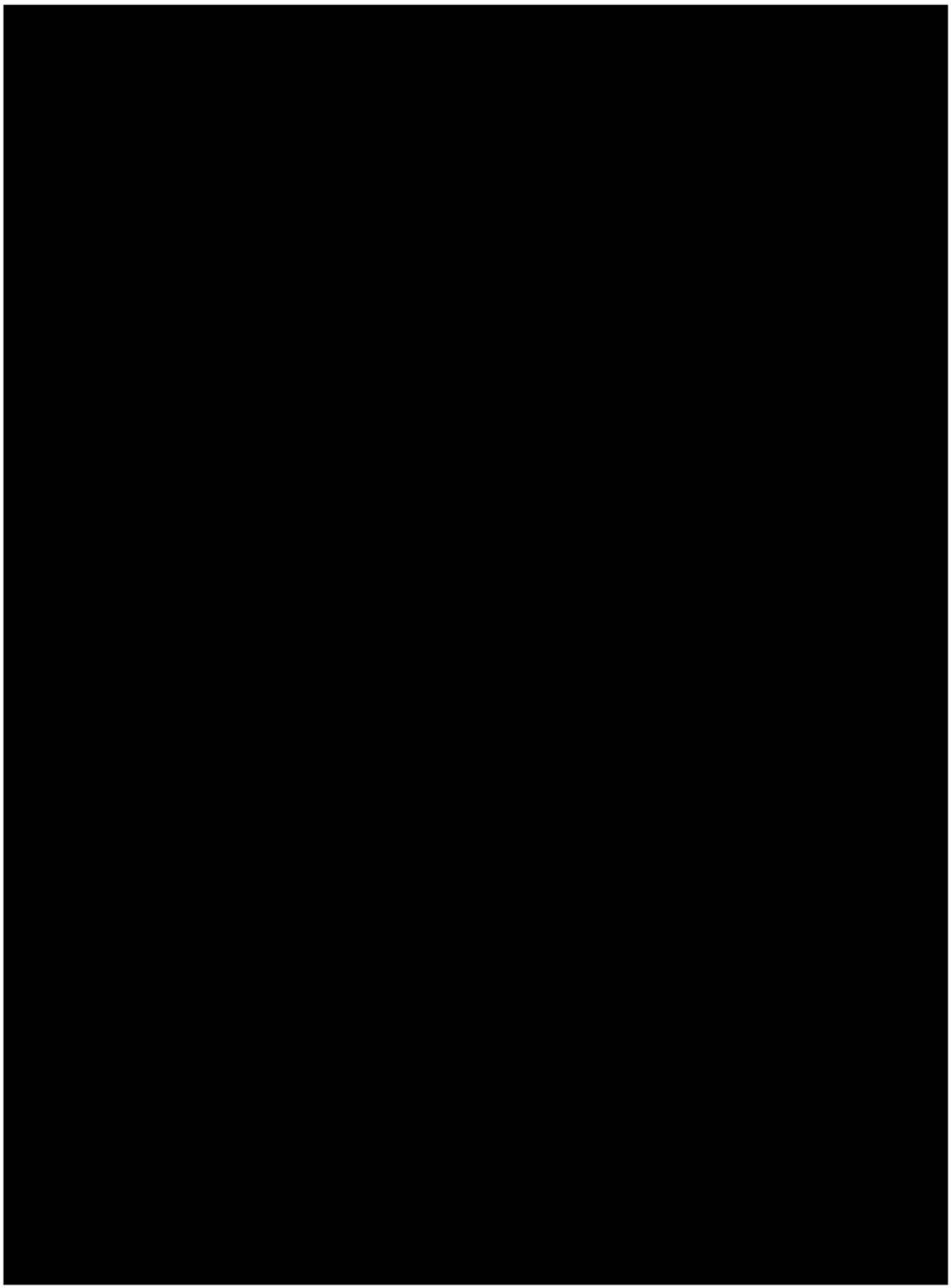


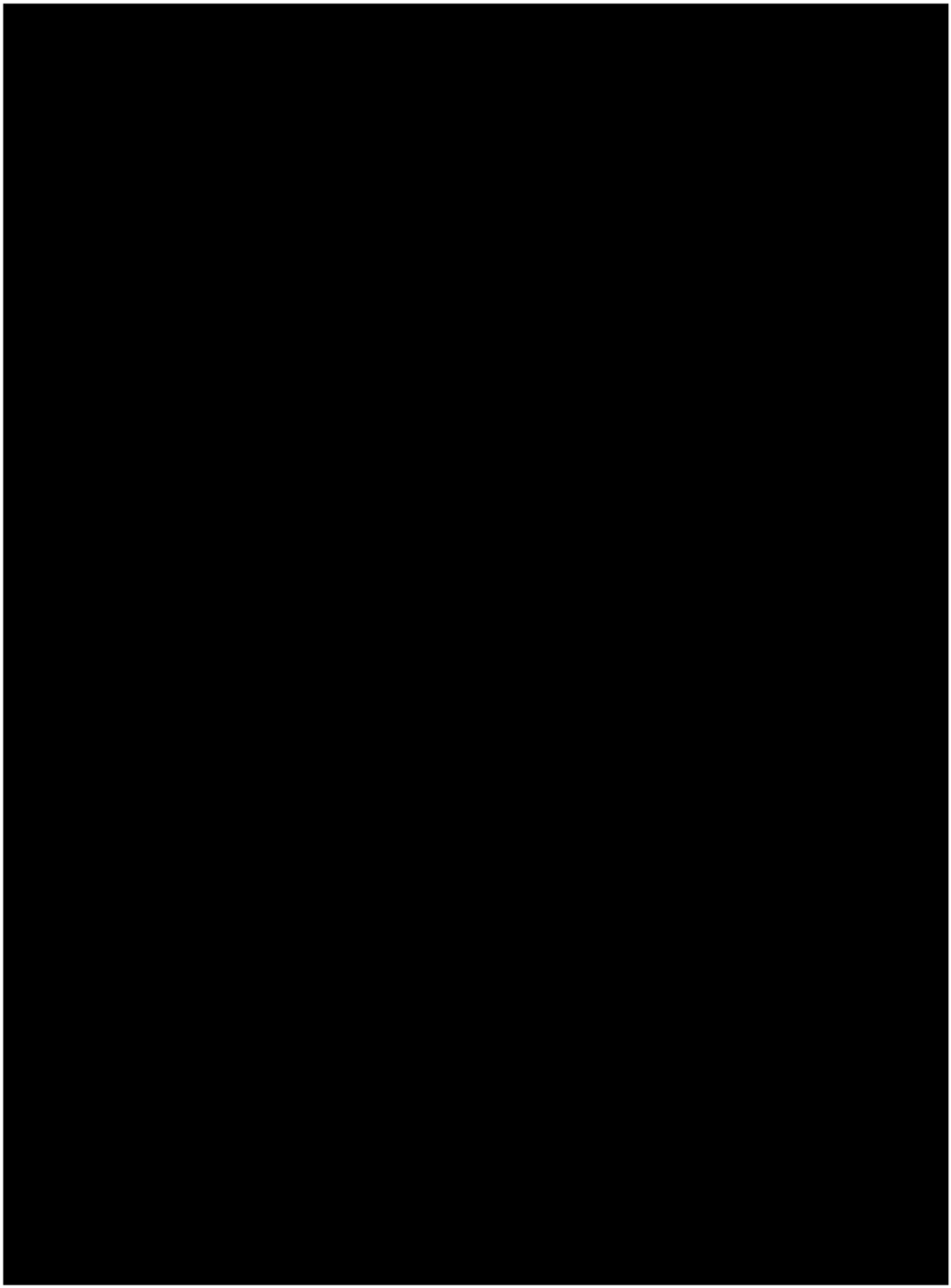


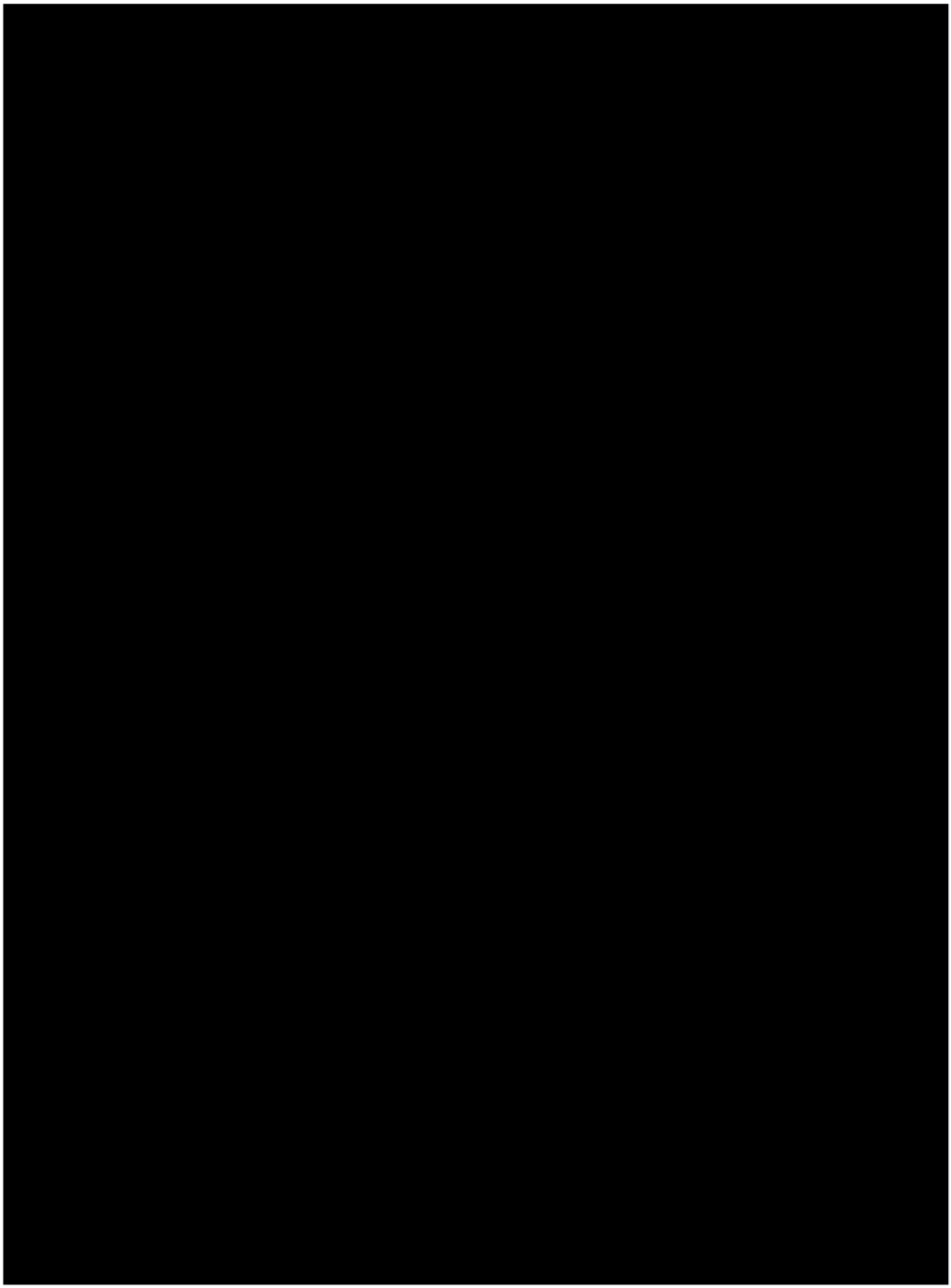


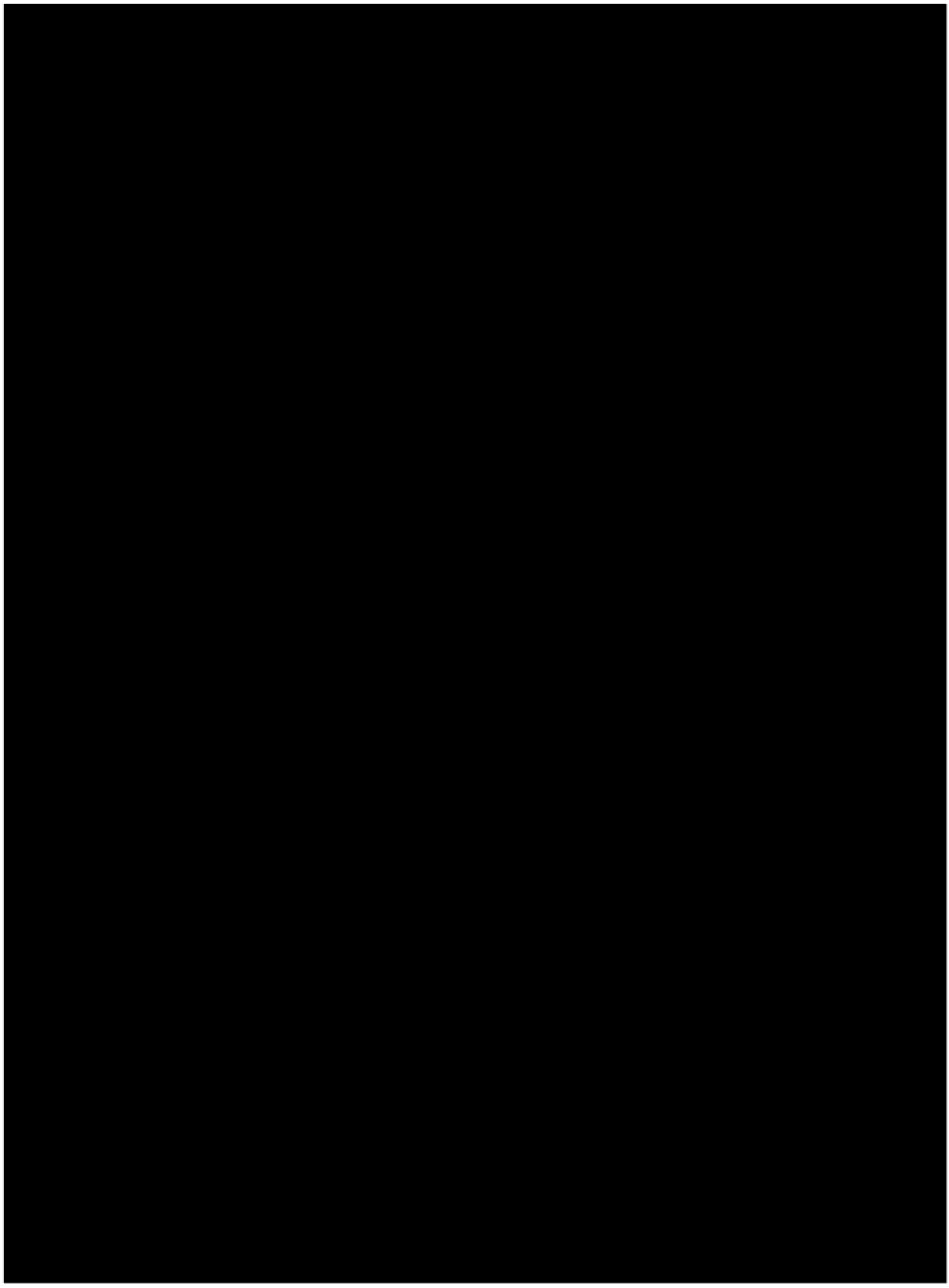


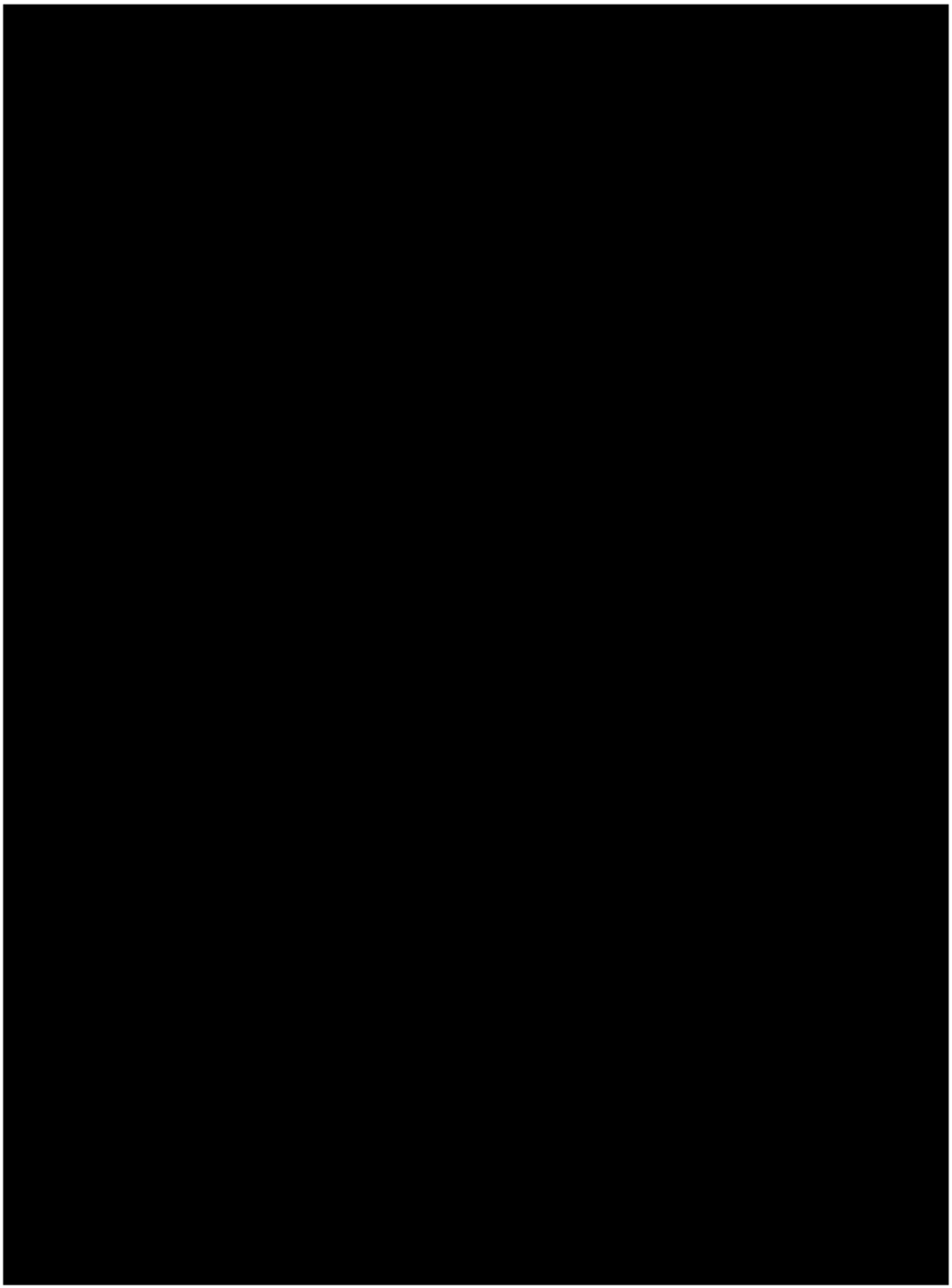


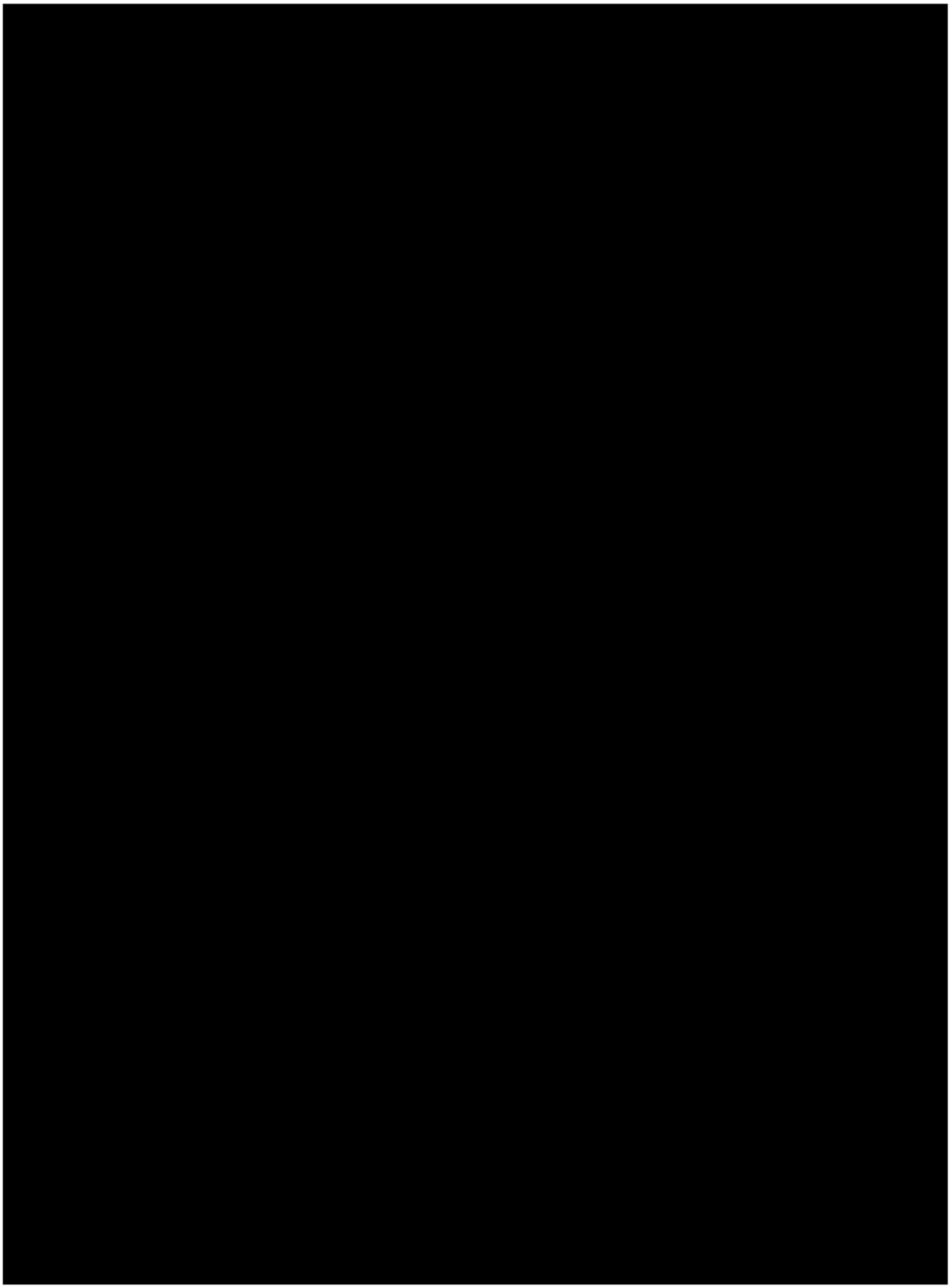


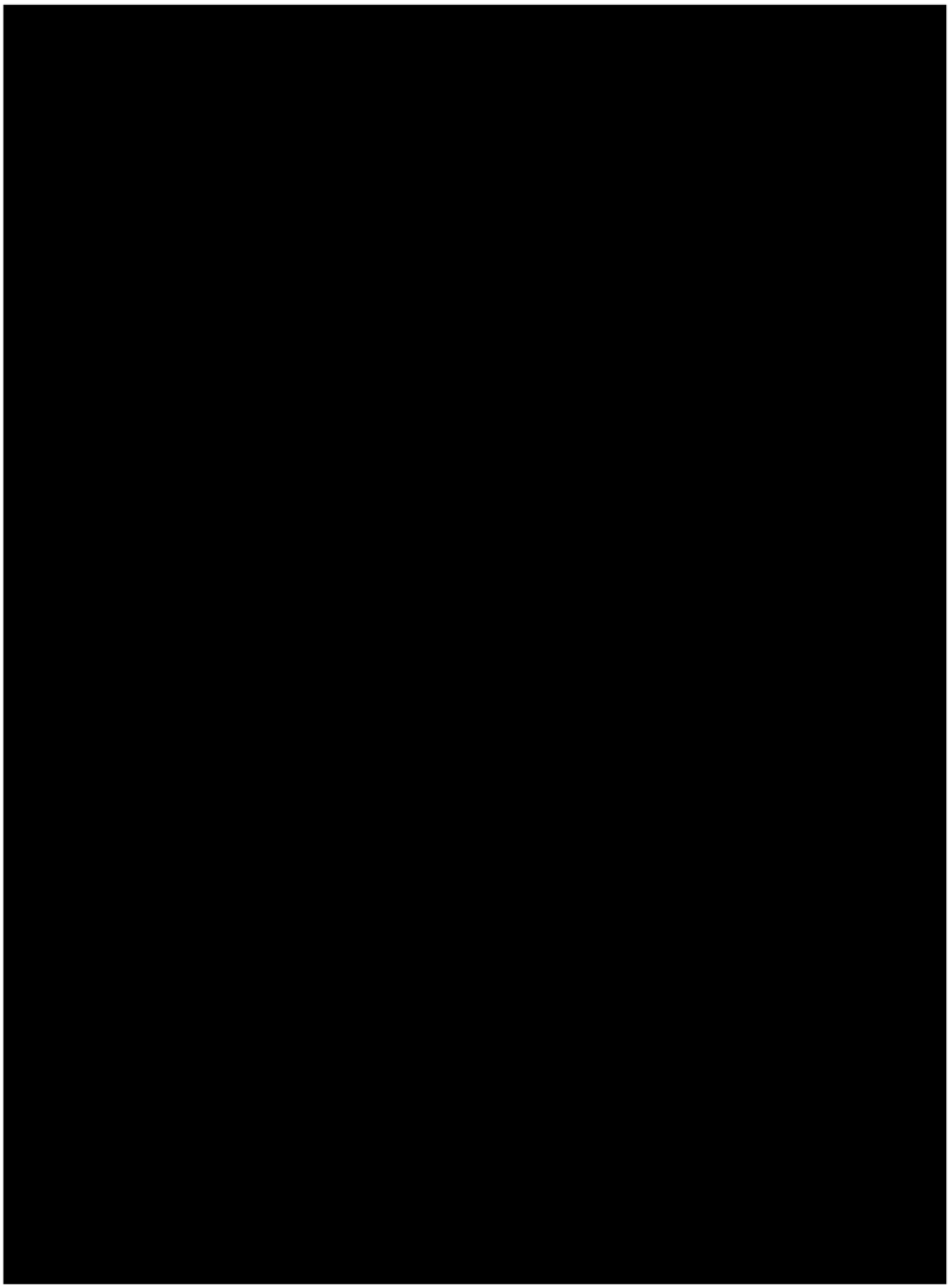


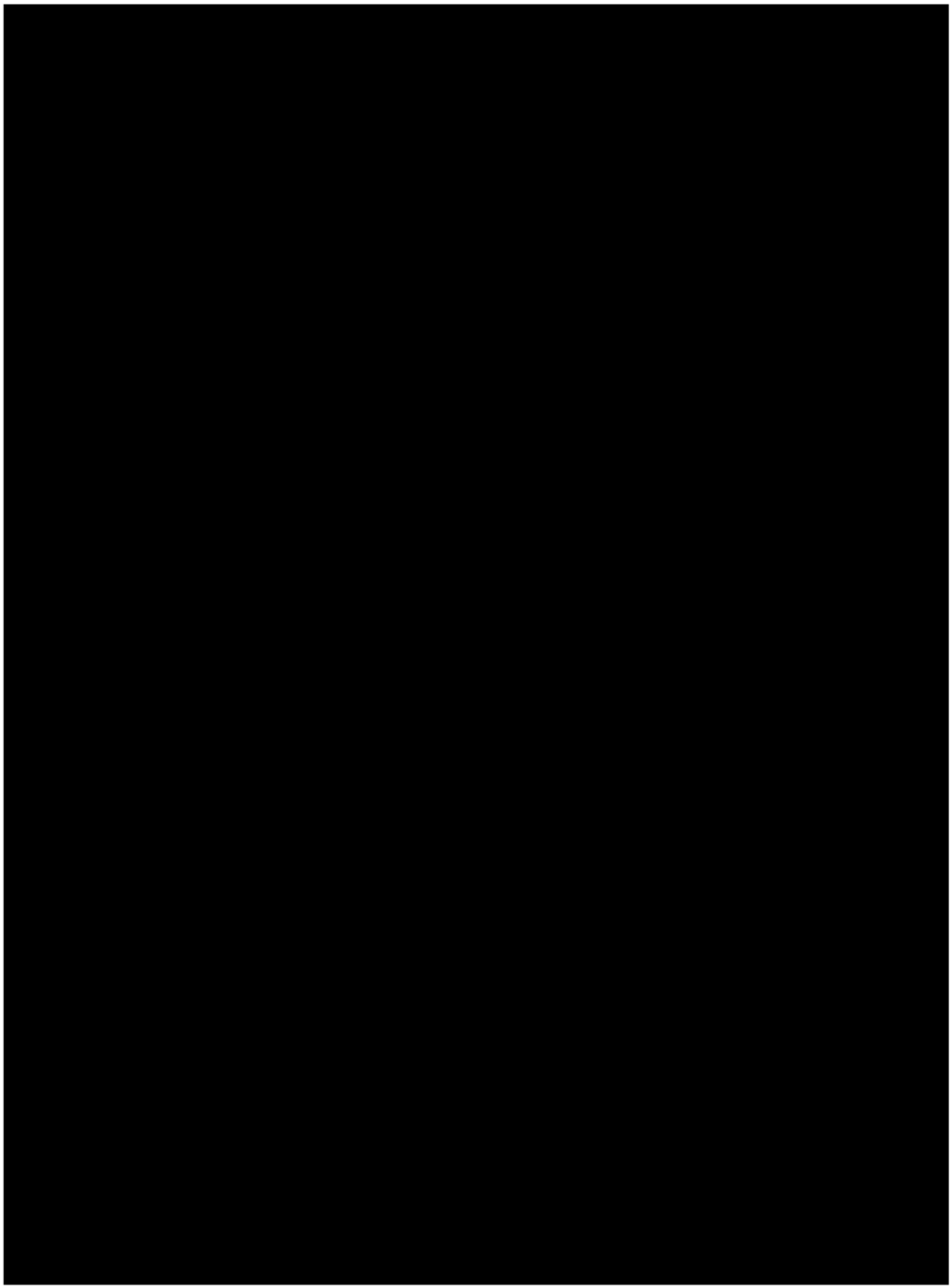


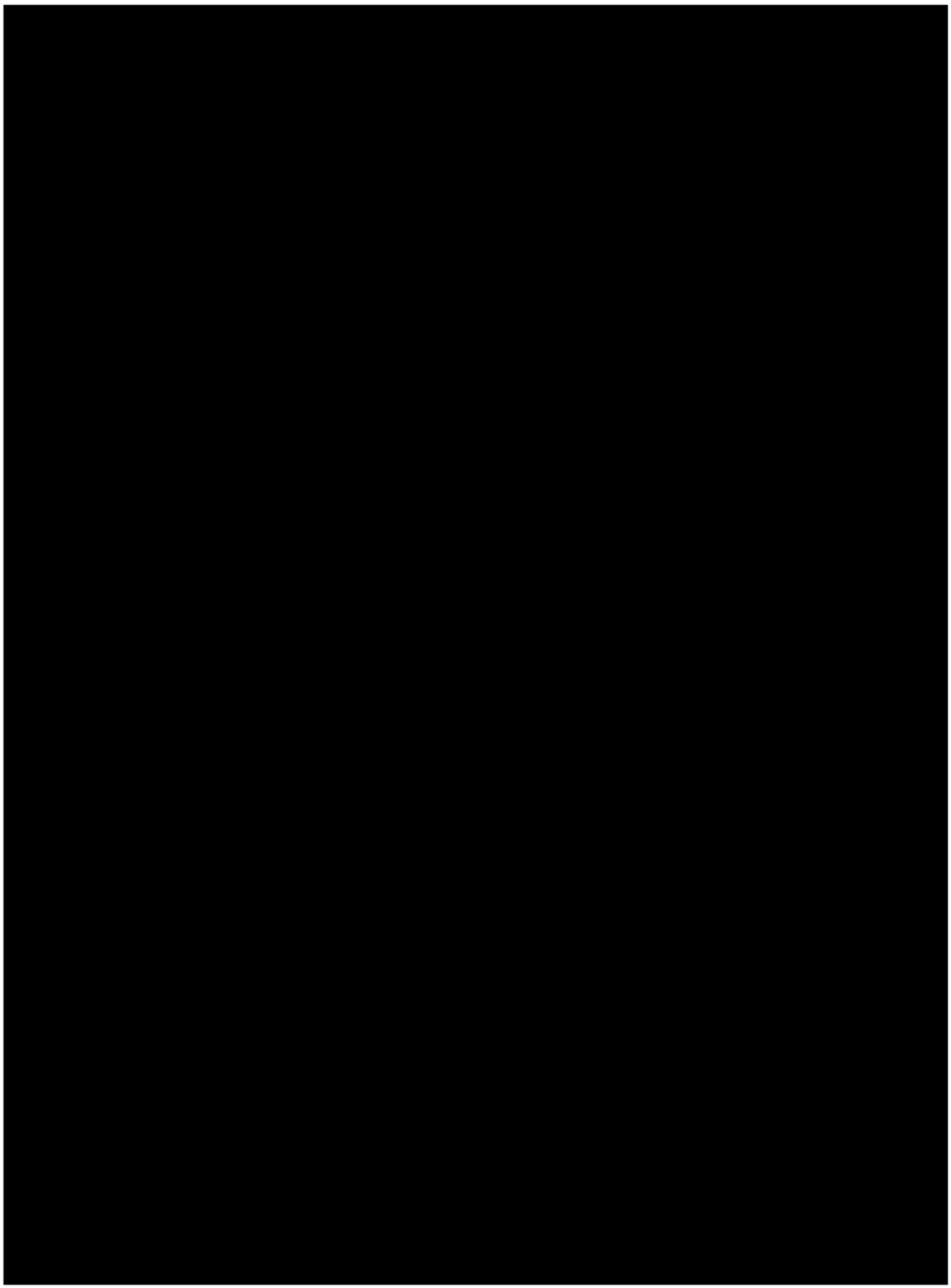


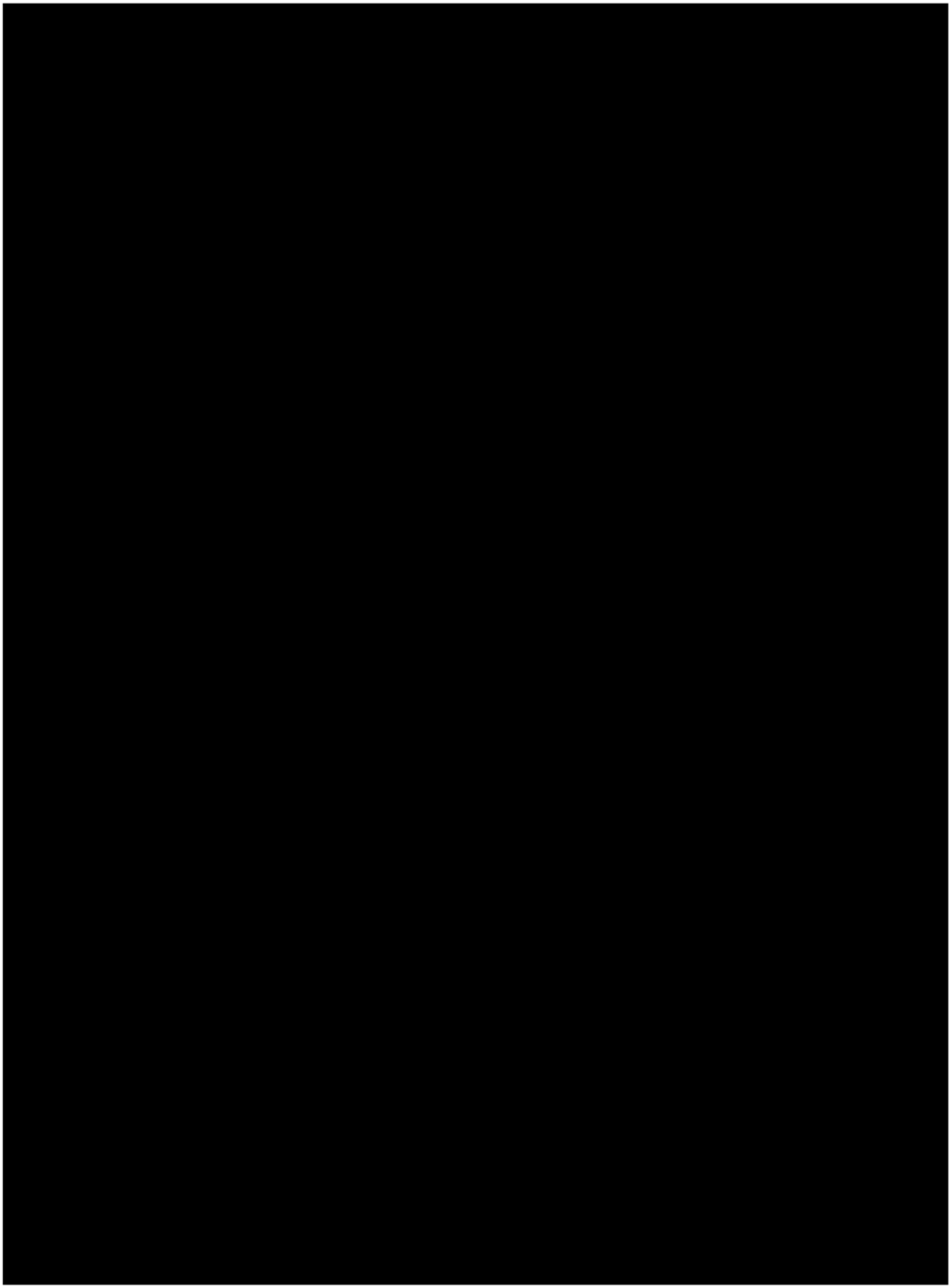


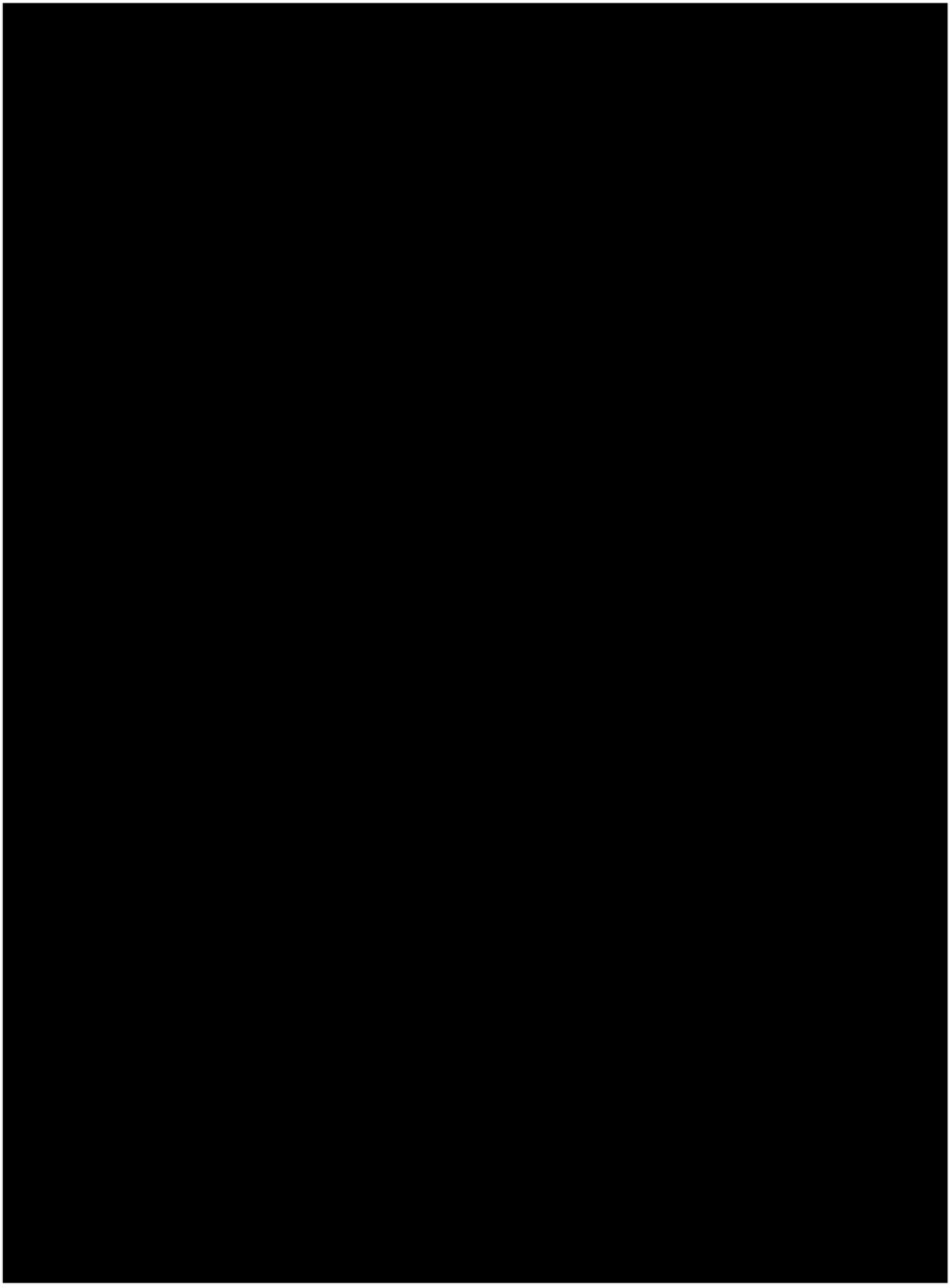












Hochschule Reutlingen
Reutlingen University

Erklärung zur Abgabe einer Bachelor- / Master-Thesis

Ich versichere ehrenwörtlich, dass ich

- die abgegebene Thesis selbständig verfasst habe,
- alle benutzten Quellen und Hilfsmittel - dazu zählen auch sinngemäß übernommene Inhalte, leicht veränderte Inhalte sowie übersetzte Inhalte - in Quellenverzeichnissen, Fußnoten oder direkt bei Zitaten angegeben habe,
- alle wörtlichen und sinngemäßen Zitate von Textstücken, Tabellen, Grafiken, Fotos, Quellcode usw. aus fremden Quellen als solche gekennzeichnet und mit seitengenauen Quellenverweisen versehen habe,
- die von mir eingereichten Dokumente und Artefakte noch nicht in dieser oder ähnlicher Form einer anderen Kommission zur Prüfung vorgelegt wurden,
- alle nicht als Zitat gekennzeichneten Inhalte selbst erstellt habe und dass ich
- den „Leitfaden für gute wissenschaftliche Praxis im Studiengang MKI“[1] kenne und achte.

Mir ist bekannt, dass unmarkierte und unbelegte Zitate und Paraphrasen Plagiate sind und nicht als handwerkliche Fehler, sondern als eine Form vorsätzlicher Täuschung der Prüfer gelten, da fremde Gedanken als eigene Gedanken vorgetäuscht werden mit dem Ziel der Erschleichung einer besseren Leistungsbewertung.
Mir ist bekannt, dass Plagiarismus die Standards guter wissenschaftlicher Praxis, die Regeln des Studiengangs Medien- und Kommunikationsinformatik, die Studien- und Prüfungsordnung der Hochschule Reutlingen (§ 10 Täuschung und Ordnungsverstoß) und das Landeshochschulgesetz von Baden-Württemberg (§ 3 Wissenschaftliche Redlichkeit Abs. 5, § 62 Exmatrikulation Abs. 3) missachtet und seine studienrechtlichen Folgen vom Nichtbestehen bis zur Exmatrikulation reichen.
Mir ist auch bekannt, dass Plagiate sogar das Urheberrechtsgesetz (§ 51 Zitate, § 63 Quellenangabe, § 106 Unerlaubte Verwertung urheberrechtlich geschützter Werke) verletzen und zivil- und strafrechtliche Folgen nach sich ziehen können.

Nachname: _____

Vorname: _____

Matrikelnummer: _____

Datum: _____

Unterschrift: _____

[1] https://bscwserv.reutlingen-university.de/bscw/bscw.cgi/d2871027/GWP.pdf