

**Hochschule für Technik, Wirtschaft und Kultur Leipzig**

Fakultät Informatik, Mathematik und Naturwissenschaften  
Bachelorstudiengang Medieninformatik

Bachelorarbeit  
zur Erlangung des akademischen Grades

**Bachelor of Science (B.Sc.)**

# Integration des Tacton Produktkonfigurators in ein Open Source Shopsystem

**Autor:** Philipp Anders  
philipp.anders.pa@gmail.com

**Betreuer:** Prof. Dr. Michael Frank (HTWK Leipzig)  
Dipl.-Wirt.-Inf. (FH) Dirk Noack  
(Lino GmbH)

**Abgabedatum:** 29.09.2015



# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>V</b>
<b>Tabellenverzeichnis</b>	<b>VII</b>
<b>Listing-Verzeichnis</b>	<b>IX</b>
<b>Abkürzungsverzeichnis</b>	<b>XI</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Ziel der Arbeit . . . . .	2
1.2 Aufbau der Arbeit . . . . .	2
<b>2 Grundlagen</b>	<b>3</b>
2.1 Bezugsrahmen . . . . .	3
2.1.1 Ökonomischer Bezug . . . . .	3
2.1.2 Produktklassifizierung . . . . .	4
2.2 Produktkonfiguration . . . . .	5
2.2.1 Begriffsüberblick . . . . .	5
2.2.2 Wissensrepräsentation . . . . .	6
2.2.3 Konfigurationslösung . . . . .	9
2.2.4 Konfiguratoren . . . . .	10
2.3 Webservices . . . . .	11
2.3.1 SOAP . . . . .	12
2.3.2 REST . . . . .	13
2.3.2.1 Einschränkungen . . . . .	13
2.3.2.2 REST-Konformität . . . . .	16
2.3.2.3 Eigenschaften . . . . .	16
2.4 eCommerce . . . . .	17
2.4.1 Anwendungsrahmen . . . . .	17
2.4.2 eShop-Systeme . . . . .	18
<b>3 Analyse</b>	<b>21</b>
3.1 Tacton Produktkonfigurator . . . . .	21
3.1.1 Konfigurationsmodell . . . . .	21
3.1.1.1 Components und Configuration . . . . .	22
3.1.1.2 Execution . . . . .	25
3.1.2 TCsite . . . . .	28
3.1.2.1 Architektur . . . . .	28
3.1.2.2 Konfigurationsprozess . . . . .	32
3.1.2.3 Erweiterbarkeit . . . . .	35
3.1.3 Zwischenfazit . . . . .	36
3.2 Shopsystem . . . . .	37
3.2.1 Architektur . . . . .	38

---

3.2.2	Erweiterbarkeit . . . . .	39
3.2.3	Konfiguration . . . . .	40
3.2.3.1	Technische Analyse . . . . .	41
3.2.3.2	Einkaufsvorgang . . . . .	41
3.3	Fazit . . . . .	43
<b>4</b>	<b>Anforderungsanalyse</b>	<b>45</b>
4.0.1	Funktionale Anforderungen . . . . .	46
4.0.1.1	Shopware-Plugin . . . . .	46
4.0.1.2	TCsite-Plugin . . . . .	49
4.0.2	Nichtfunktionale Anforderungen . . . . .	50
4.0.2.1	Shopware-Plugin . . . . .	50
4.0.2.2	TCsite-Plugin . . . . .	51
4.1	Konzeption . . . . .	52
4.1.1	Shopware-Plugin . . . . .	52
4.1.1.1	Einkaufsvorgang . . . . .	53
4.1.1.2	Zusammenfassung . . . . .	61
4.1.2	TCsite-Plugin . . . . .	62
4.1.2.1	Ressourcen . . . . .	64
4.1.2.2	Serviceendpunkte . . . . .	64
4.1.2.3	Zusammenfassung . . . . .	66
4.2	Umsetzung . . . . .	67
4.3	Fazit . . . . .	70
	<b>Literaturverzeichnis</b>	<b>I</b>
	<b>A Anhang</b>	<b>VII</b>

## Abbildungsverzeichnis

Abb. 2.1	Unvereinbarkeitshypothese nach Porter . . . . .	3
Abb. 2.2	UML-Visualisierung der exemplarischen Notebook-Konfiguration . . . . .	7
Abb. 2.3	Visualisierung einer Konfigurationslösung als UML-Instanz-Diagramm . . . . .	9
Abb. 2.4	Generische Client-Server Kommunikation bei Webservices . . . . .	11
Abb. 2.5	HTTP-Methoden und ihre Eigenschaften . . . . .	15
Abb. 2.6	Grundformen des eCommerce . . . . .	17
Abb. 2.7	Grobarchitektur eines eShop-Systems . . . . .	18
Abb. 3.1	High-Level-Architektur des Tacton-Konfigurationsmodells . . . . .	22
Abb. 3.2	Part-Struktur der exemplarischen Notebook-Konfiguration . . . . .	23
Abb. 3.3	Zuordnung von Parts und Component Classes . . . . .	24
Abb. 3.4	Zuordnung von Part und Component Class der exemplarischen Notebook-Konfiguration . . . . .	24
Abb. 3.5	Ausschnitt der generischen Execution-Struktur . . . . .	25
Abb. 3.6	Mögliche Exection-Struktur der exemplarischen Notebook-Konfiguration . . . . .	27
Abb. 3.7	High-Level-Architektur von TCsite . . . . .	28
Abb. 3.8	Schichtenarchitektur von TCsite . . . . .	29
Abb. 3.9	Detailsansicht eines Produktes aus dem Produktkatalog . . . . .	30
Abb. 3.10	Quotation-View von TCsite . . . . .	31
Abb. 3.11	Kommunikation zwischen TCsite und TCserver . . . . .	32
Abb. 3.12	Configuration-View in TCsite der exemplarischen Notebook-Konfiguration . . . . .	34
Abb. 3.13	Configuration-View in TCsite der exemplarischen Notebook-Konfiguration nach Anwenderwahl . . . . .	34
Abb. 3.14	Configuration-View in TCsite während einer Konfliktsituation . . . . .	35
Abb. 3.15	TCsite Pluginkonzept . . . . .	36
Abb. 3.16	High-Level-Architektur von Shopware . . . . .	38
Abb. 3.17	Aktivitätsdiagramm des Einkaufsvorgangs eines Varianten-Artikels in Shopware . . . . .	42
Abb. 4.1	Systemkontext . . . . .	45
Abb. 4.2	systemcontextErweitert . . . . .	46
Abb. 4.3	shopwareArtikelRelationenModell . . . . .	52
Abb. 4.4	konzeptionEinkausvorgang1 . . . . .	54
Abb. 4.5	konzeptCreate . . . . .	56
Abb. 4.6	konzeptChange . . . . .	57
Abb. 4.7	dynamischeVariantengenerierung . . . . .	59
Abb. 4.8	konzeptCopy . . . . .	60
Abb. 4.9	konzeptConfigurableUML . . . . .	63
Abb. 4.10	konzeptRestZusammenfassung . . . . .	66
Abb. A.1	Vollständige Darstellung der generischen Execution-Struktur . . . . .	VII
Abb. A.2	tcsiteAdministration . . . . .	VIII
Abb. A.3	tcSiteConfigurationOptionalGroups . . . . .	VIII

Abb. A.4 vollständige High-Level-Architektur von Shopware . . . . .	IX
Abb. A.5 shopwareBackendArtikel . . . . .	IX
Abb. A.6 shopwareBackendArtikelVarianten . . . . .	X
Abb. A.7 shopwareNotebookDetail . . . . .	X
Abb. A.8 shopwareNotebookWarenkorb . . . . .	XI
Abb. A.9 shopwareArtikelListing . . . . .	XI
Abb. A.10mozillaCORS . . . . .	XII

## Tabellenverzeichnis

Tab. 2.1	Constraints des Konfigurationsmodells aus Abbildung 2.2 . . . . .	8
----------	---	---





## Listing-Verzeichnis

4.1	Beispiel einer configuration . . . . .	64
-----	--	----



## Abkürzungsverzeichnis

<b>AJAX</b>	asynchronous JavaScript and XML
<b>API</b>	Application Programming Interface
<b>ATO</b>	Assemble-to-Order
<b>B2B</b>	Business-to-Business
<b>B2C</b>	Business-to-Customer
<b>CORS</b>	Cross-Origin Resource Sharing
<b>CSP</b>	Constraint Satisfaction Problem
<b>ETO</b>	Engineer-to-Order
<b>i.d.R.</b>	in der Regel
<b>JSON</b>	JavaScript Object Notation
<b>MC</b>	Mass Customization
<b>MTO</b>	Make-to-Order
<b>MVC</b>	Model View Controller
<b>ORM</b>	objektrelationale Abbildung
<b>PTO</b>	Pick-to-Order
<b>RPC</b>	Remote Procedure Call
<b>SOP</b>	Same-Origin-Policy
<b>UML</b>	Unified Modeling Language
<b>URI</b>	Uniform Resource Identifier
<b>vgl.</b>	vergleiche
<b>WSDL</b>	Web Service Description Language

# 1. Einleitung

„I'd like a plain omelette, no potatoes, tomatoes instead, a cup of coffee and toast.“

„No substitutions.“

„What do you mean, you don't have any tomatoes?“

„Only what's on the menu. You can have a number 2 – a plain omelette. It comes with cottage fries and rolls.“

„I know what it comes with but it's not what I want.“

Diese Unterhaltung stammt aus der „Diner Scene“ des Films „Five Easy Pieces“ (1970). Darin versucht Schauspieler Jack Nicholson verzweifelt, sich ein individuelles Essen aus den Zutaten der Gerichte in der Karte zusammenzustellen. Die Szene endet mit der Explosion des angestauten Frusts über diese umständliche Bestellung, indem er das gesamte Geschirr mit dem Arm vom Tisch fegt.

Diese Situation illustriert auf humorvolle Weise das Problemfeld der Produktkonfiguration. In diesem möchte sich der Kunde ein individuelles Produkt zusammenstellen, da keiner der Standardartikel seinen Vorstellungen entspricht. Gleichzeitig muss die Zusammenstellung vom Anbieter auch machbar sein. Im obigen Beispiel wäre ein Omelett mit Tomaten sicherlich noch realisierbar gewesen. Hätte sich Jack Nicholson zum Nachtisch noch eine fachmännisch zubereitete Crème brûlée gewünscht, würde das Diner vor einer echten Problematik stehen.

Die Produktkonfiguration findet Anwendung in einem breiten Aufgabenfeld. Das kann die einfache Optionsauswahl beim Designen eines individuellen Nike-Schuhs via *NIKEiD* sein oder die Konfiguration eines Flugzeugs durch Fachpersonal mit technischer Expertise. Im Gegensatz dazu steht ein Onlineshop wie z.B. Zalando üblicherweise für den Vertrieb von Standardprodukten. Mit der Auswahl der richtigen Kleidergröße oder -farbe bieten diese Systeme einen eher begrenzten Entscheidungsspielraum.

Die vorliegende Arbeit entstand im Rahmen der Tätigkeit bei der Lino GmbH. Diese vertreibt unter anderem als autorisierter Reseller Produkte der Tacton Systems AB (Tacton), welche eine softwaretechnische Lösung für die Produktkonfiguration in Form des Tacton Produktkonfigurators anbietet. In dieser Arbeit geht es um eine Entwicklung zur Verheiratung dieses Konfigurators mit einem Open-Source Shopsystem.

## 1.1. Ziel der Arbeit

Das Ziel der Arbeit ist die Entwicklung einer softwaretechnischen Lösung, die es dem Anwender ermöglicht, ein Produkt in einem Open-Source Shopsystem zu konfigurieren und zu bestellen, wobei die Konfiguration durch den Tacton Produktkonfigurator realisiert wird. Dabei müssen zwei Systeme betrachtet werden – das Shopsystem und das Konfigurationssystem. Um letzteres in einer neuen Systemumgebung verfügbar zu machen, muss ein entsprechendes Integrationskonzept entworfen werden. Dabei wird geklärt, welche Daten auf welchem Wege auszutauschen sind. Wie weiterhin das Resultat einer Konfiguration in den Bestellprozess integriert werden kann, wird von der konkreten Wahl des Shopsystems abhängig gemacht. Letztendlich werden ein Plugin für das Konfigurationssystem und ein Plugin für das Shopsystem entwickelt, welche durch deren Zusammenarbeit das Integrationsszenario bewältigen.

## 1.2. Aufbau der Arbeit

Im Grundlagenteil (Kapitel 2) wird zunächst ein Bezugsrahmen für das Thema Produktkonfiguration geschaffen. Dabei wird geklärt, welche ökonomischen Rahmenbedingungen deren Einsatz motivieren und auf welche Produkte sie anwendbar sind. Das theoretische Fundament leitet sich aus dem Titel der Arbeit „**Integration** des Tacton **Produktkonfigurators** in ein Open-Source **Shopsystem**“ ab. Zunächst werden die Themen Produktkonfiguration sowie die Softwaresysteme, die diese realisieren, behandelt. Unter der Annahme, dass sowohl das Shopsystem als auch das Konfigurationssystem webfähig sind, werden danach Webservices als Integrationsinstrument erläutert. Abschließend werden Online-Shops in einen ökonomischen Rahmen eingeordnet und typisiert.

Im Analyseteil (Kapitel 3) werden die konkreten Systeme betrachtet. Dabei wird zunächst der Tacton Produktkonfigurator besprochen. Aus dessen Analyse werden Anforderungen an das Shopsystem abgeleitet. Die Evaluierung der möglichen Kandidaten ist nicht Bestandteil der Arbeit. Es wird Shopware als Shopsystem festgelegt und einer Detailbetrachtung unterzogen. Aus dem Kapitel resultieren die Erweiterungsmöglichkeiten und grundlegenden Verantwortungsbereiche der Systeme im Integrationsszenario.

Aufbauend auf die Analyse werden in Kapitel 4 die funktionalen und nichtfunktionalen Anforderungen der Integration formuliert. Diese dienen als Orientierungspunkte für die folgende Entwicklung eines Integrationskonzepts in Kapitel 5. Dabei wird klar, dass das Integrationsszenario eine große Zahl kleinteiliger Veränderungen im Shopsystem erfordert. Darum skizziert Kapitel 6 die Herangehensweise und den Ablauf der Implementierung.

In Kapitel 7 wird die Lösung diskutiert, wobei Potentiale für eine zukünftige Entwicklungsarbeit aufgezeigt werden. Die Arbeit schließt mit einer anwendungsbezogenen Einordnung des Resultats in den ökonomischen Kontext ab, der am Anfang der Arbeit umrissen wurde.

## 2. Grundlagen

### 2.1. Bezugsrahmen

Im Folgenden wird ein Bezugsrahmen für Produktkonfigurationssysteme (Konfiguratoren) geschaffen. Dazu wird zunächst die marktwirtschaftliche Situation erläutert, welche die Notwendigkeit *hybrider Wettbewerbsstrategien* begründet. Diese wiederum stellen zu ihrer Umsetzbarkeit Bedingungen an *Produktionskonzepte*, welche daraufhin vorgestellt werden.

#### 2.1.1. Ökonomischer Bezug

Neue Wettbewerbsbedingungen und gesteigerte Kundenansprüche haben den Druck auf Industrieunternehmen zur Produktion individualisierter Produkte erhöht (Piller, 1998). Das entspricht einer stärkeren Forcierung der *Leistungsdifferenzierung* (Lutz, 2011). Diese ist eine der von Porter (2002) beschriebenen „generischen Wettbewerbsstrategien“. Bei der Leistungsdifferenzierung werden Produkte mit höherem individuellen Kundennutzen zu höheren Preisen verkauft. Eine der anderen Wettbewerbsstrategien ist die *Kostenführerschaft*. Diese bezieht sich klassischerweise auf Anbieter von Massenproduktion. Dabei werden die Marktanteile durch die Unterbietung der Konkurrenzpreise gesteigert.

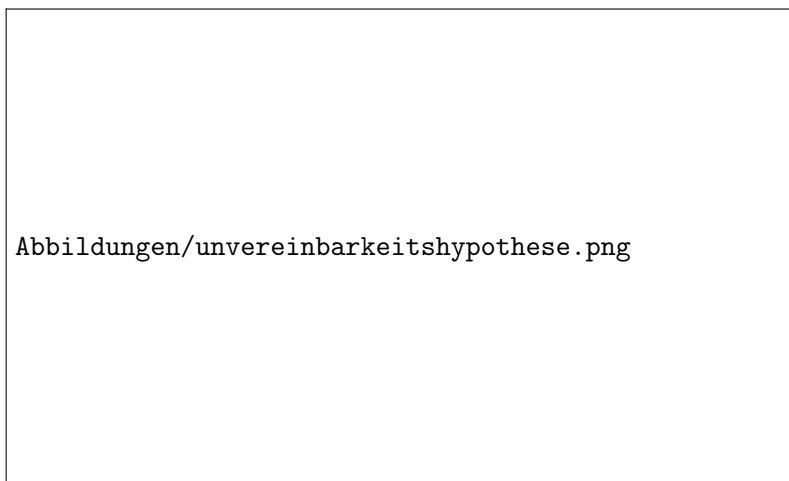


Abbildung 2.1.: Unvereinbarkeitshypothese nach Porter (1980), zitiert von Schuh (2005)

In diesem Zusammenhang formulierte Porter die *Unvereinbarkeitshypothese*. So sollen Kostenführerschaft und Leistungsdifferenzierung nicht gleichzeitig erreichbar sein. Eine uneindeutige Positionierung führe zu einem „stuck in the middle“ und damit zur Unwirtschaftlichkeit, wie Abbildung 2.1 darstellt.

Die Beobachtung der Unternehmensrealität zeichnet ein anderes Bild. Neue Organisationsprinzipien, Informationsverarbeitungspotentiale und Produktstrukturierungsansätze ermöglichen einen Kompromiss aus Preis- und Leistungsführerschaft (Schuh, 2005). Das Ergebnis wird als *hybride Wettbewerbsstrategien* bezeichnet. Eine dieser Strategien ist die sogenannte *Mass Customization (MC)*.

MC ist die „Produktion von Gütern und Leistungen für einen (relativ) großen Absatzmarkt, welche die unterschiedlichen Bedürfnisse jedes einzelnen Nachfragers dieser Produkte treffen, zu Kosten, die ungefähr denen einer massenhaften Fertigung vergleichbarer Standardgüter entsprechen“ (Piller, 1998). Mit anderen Worten: Preisvorteil (in der Regel (i.d.R.) durch Massenfertigung) wird mit Individualisierung (i.d.R. durch Variantenvielfalt) vereint.

Ein Beispiel für einen Anwendungsbereich von MC ist die Automobilindustrie. Hersteller bieten Modelle in verschiedenen Ausführungen an, die der Kunde beim Online-Bestellprozess weiter individualisieren kann. Beispiele hierfür sind Opel<sup>1</sup> und BMW<sup>2</sup>.

### 2.1.2. Produktklassifizierung

MC stellt zu deren Umsetzbarkeit gewisse Bedingungen an die Produktionsweisen der abzusetzenden Güter. Im Folgenden wird eine Klassifizierung von Produkten in Bezug zu deren Herstellung vorgestellt. Die Typen werden als *Produktionskonzepte* bezeichnet (nach Schuh 2006, zitiert von Lutz 2011):

**Pick-to-Order (PTO):** Herstellung ohne Kundenauftrag; Lagerhaltung auf Ebene ganzer Produkte; Keine Abhängigkeit dieser Produkte untereinander;

Beispiel: Ein Standardnotebook.

**Assemble-to-Order (ATO):** Herstellung ohne Kundenauftrag; Lagerhaltung auf Ebene der Baugruppen/-teilen; Teile mit Abhängigkeiten untereinander;

Beispiel: Ein Notebook, bei welchem auf Kundenwunsch statt des CD-Laufwerks eine zusätzliche Festplatte eingebaut wird. Die Festplatte wurde bereits im Lager vorgehalten.

**Make-to-Order (MTO):** Herstellung teilweise erst nach Kundenenauftrag; Lagerhaltung auf Ebene der Baugruppen/-teilen; Produktion oder parametrisierte Konstruktion (d.h. durch Regeln generierten Konstruktionsdaten) von Komponenten nach Kundenanforderung; Abhängigkeiten zwischen Teilen; keine unendliche Anzahl von Varianten;

Beispiel: Ein Notebook, bei dem der Kunde die Displaygröße abweichend von den Standarddiagonallängen bestimmen kann. Display und Notebookgehäuse müssen konstruiert/hergestellt und die technischen Standardkomponenten (z.B. Festplatte, Motherboard) eingepasst werden.

<sup>1</sup><http://www.opel.de/tools/konfigurator/personenwagen.html>

<sup>2</sup>[http://bmw.de/BMW\\_Konfigurator](http://bmw.de/BMW_Konfigurator)

**Engineer-to-Order (ETO):** Produkt ist nicht komplett vom Hersteller vorhersehbar;  
Wenig bis keine Lagerhaltung auf Ebene der Baugruppen/-teilen; Entwicklung und  
Fertigung von Teilen nach Kundenspezifikation; unendliche Variantenanzahl möglich;  
Beispiel: Herstellung eines Notebooks mit ausklappbarem Gamepad.

Die Produktionskonzepte unterscheiden sich hauptsächlich danach, wann die Produktion der Baugruppen/-teile beginnt – vor oder nach Auftragspezifikation. Eine Produktion vor Auftragseingang, also ohne Kundenspezifikation, erlaubt Lagerhaltung. Ein hoher Komponentenanteil, der erst nach Auftragseingang hergestellt werden kann oder sogar konstruiert werden muss, spricht für eine starke Kundenindividualisierung (Lutz, 2011). Die unterschiedlichen Produktionskonzepte haben jeweils einen Anwendungsbezug zu Konfiguratoren, welche im Folgenden vorgestellt werden.

## 2.2. Produktkonfiguration

Aus der MC (siehe Kapitel 2.1.1) resultiert mehr Produktvariabilität und damit Produktkomplexität. Die *Produktkonfiguration* (Konfiguration) ist ein Werkzeug zur Beherrschung dieser Komplexität. Sie unterstützt das Finden einer Produktvariante, die auf Kundenanforderungen angepasst und gleichzeitig machbar ist (Lutz, 2011).

### 2.2.1. Begriffsüberblick

Die **Konfiguration** ist eine spezielle Designaktivität, bei der der zu konfigurierende Gegenstand aus Instanzen einer festen Menge wohldefinierter Komponententypen zusammengesetzt wird, welche entsprechend einer Menge von Konfigurationsregeln (Constraints) kombiniert werden können (Sabin und Weigel, 1998).

Die Einordnung als Designaktivität erlaubt außerdem die Beschreibung der Konfiguration als ein Designtyp. Es werden das *Routine Design*, *Innovative Design* und *Creative Design* unterschieden. Die Konfiguration entspricht dem Routine Design. Dabei handelt es sich um ein Problem, bei der die Spezifikation der Objekte, deren Eigenschaften sowie kompositionelle Struktur vorgegeben ist und die Lösung auf Basis einer bekannten Strategie gefunden wird (Brown und Chandrasekaran, 1989). Damit ist Routine Design die simpelste der drei Formen. Die anderen Designtypen enthalten hingegen Objekte und Objektbeziehungen, die erst während des Designprozesses entwickelt werden.

Die Schlüsselbegriffe der Definition von Sabin und Weigel sind Komponententypen und Constraints. **Komponententypen** sind Kombinationselemente, welche durch Attribute charakterisiert werden und eine Menge alternativer (konkreter) Instanzen repräsentieren. Übertragen auf die objektorientierte Programmierung verhalten sich Komponententypen zu Instanzen wie Klassen zu Objekten. Komponententypen stehen zueinander in Beziehung. Diese kann entweder eine „Teil-Ganzes“-Beziehung oder Generalisierung sein (Felfernig u. a., 2014). **Constraints** (d.h. Konfigurationsregeln) im engeren Sinne sind Kom-



binationseinschränkungen (Felfernig u. a., 2014). Auf die eben genannten Begriffe wird in Kapitel 2.2.2 genauer eingegangen.

Zur besseren Nachvollziehbarkeit der weiteren Terminologie ist eine Definition des Variantenbegriffs angebracht. DIN 199 beschreibt Varianten als „Gegenstände ähnlicher Form und/oder Funktion mit einem in der Regel hohen Anteil identischer Gruppen oder Teile“. Varianten sind also Gegenstandsmengen. Ein Element dieser Menge unterscheidet sich von einer anderen durch mindestens eine Beziehung oder ein Element (Lutz, 2011).

Die Einheit aus Komponententypen und dem Wissen um deren Kombinierbarkeit in Form von Constraints wird als **Konfigurationsmodell** bezeichnet. Es definiert implizit alle Varianten eines Produkts (Soininen u. a., 1998). Dadurch muss nicht jede Variante explizit definiert und abgespeichert werden (z.B. in einer Datenbank). Die Anzahl möglicher Kombinationen kann in die Millionen gehen. Das würde die Suche nach einer bestimmten Variante, die einer vorgegebenen Optionskombination entspricht, sehr zeitaufwändig machen (Falkner u. a., 2011).

Die Einheit aus Konfigurationsmodell und Kundenanforderungen (d.h. den Vorstellungen und Wünschen des Kunden über das Produkt) wird als **Konfigurationsaufgabe** bezeichnet (Felfernig u. a., 2014). Auf dessen Grundlage kann die gewünschte Konfiguration errechnet werden. Demzufolge ist der Begriff Konfiguration inhaltlich überladen – er bezeichnet sowohl den technischen Verarbeitungsprozess der Lösungsfindung als auch die Lösung selbst. Im Folgenden werden daher die Begriffe **Konfigurationsprozess** sowie **Konfigurationslösung** verwendet. Der Konfigurationsprozess wird von einem System durchgeführt. Dieses wird als **Konfigurator** bezeichnet.

Die vorausgehende Beschreibung einer Konfigurationslösung suggeriert, dass zu Beginn des Konfigurationsprozesses einmalig die Konfigurationsaufgabe formuliert und daraufhin die Konfigurationslösung ermittelt wird. Diese Form wird als **statische Konfiguration** bezeichnet. Demgegenüber erlaubt die **interaktive Konfiguration** das schrittweise Treffen und Revidieren von Entscheidungen (Hadzic und Andersen, 2004). Die Menge aller bisher getroffenen Entscheidungen wird als **Konfigurationszustand** bezeichnet (Tacton Systems AB, 2014b). Ein interaktiver Konfigurator muss beim Konfigurationsprozess einen Mechanismus besitzen, der die Konfigurationsaufgabe in Bezug mit dem Konfigurationszustand bringt.

### 2.2.2. Wissensrepräsentation

Das Wissen, welches über ein konfigurierbares Produkt besteht, wird als *Konfigurationswissen* bezeichnet (Soininen u. a., 1998). Dieses Wissen kann auf unterschiedliche Art und Weise repräsentiert (d.h. dargestellt) werden. Die Repräsentation kann zur Definition eines Konfigurationsmodells genutzt werden (Felfernig u. a., 2014). Auf konzeptioneller Ebene werden die Begriffe Wissensrepräsentation und Konfigurationsmodell äquivalent verwendet. Auf technischer Ebene bezeichnet das Konfigurationsmodell jedoch das spezifische

Format, welches ein Konfigurator versteht (Soininen u. a., 1998).

Im Folgenden wird eine grafische Form der Wissensrepräsentation vorgestellt, die auf UML basiert. Die grafische Darstellung wurde gewählt, da durch diese eine Vorstellung über die Problemdomäne gebildet werden kann. Sie ist UML-basiert, da die Sprache in der Informatik zur Wissensmodellierung bekannt ist. Zur Erklärung wird eine Notebook-Konfiguration eingeführt, die im weiteren Verlauf der Arbeit immer wieder aufgegriffen wird. Die Modellierung basiert auf dem Arbeitsbeispiel von (Felfernig u. a., 2014). Einzelne Komponenten des Beispiels wirken aus aktueller Sicht überholt, sind zur Veranschaulichung bestimmter Sachverhalte aber bewusst gewählt.

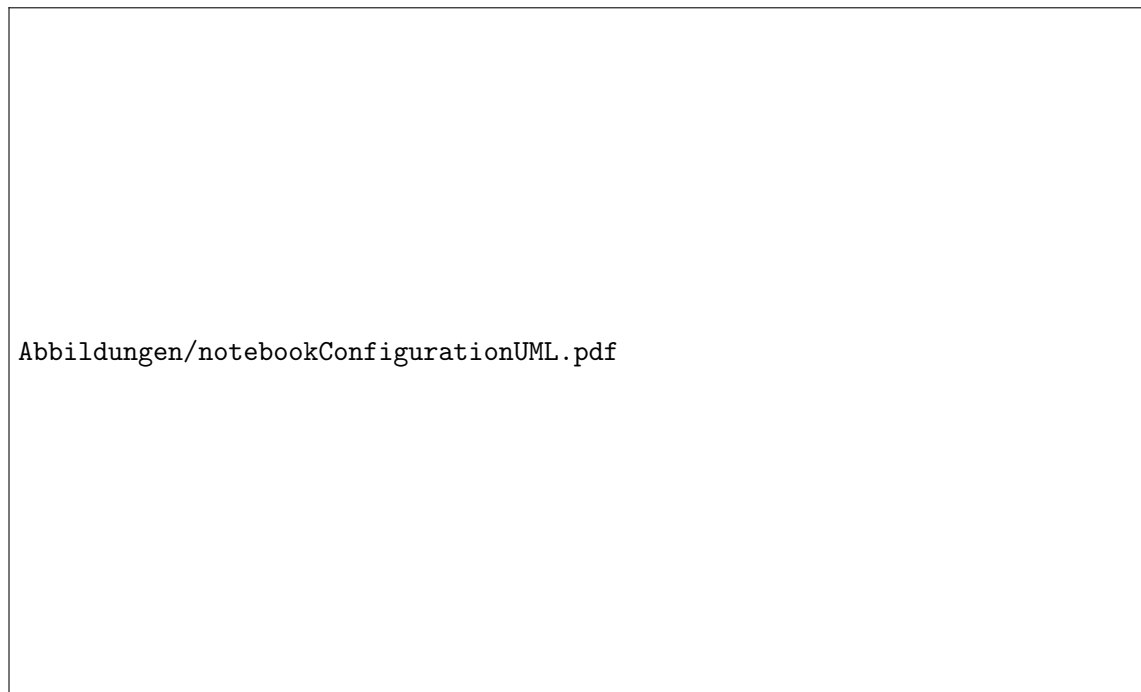


Abbildung 2.2.: UML-Visualisierung der exemplarischen Notebook-Konfiguration<sup>a</sup>

<sup>a</sup>Das 'iTunes-Music-Package' stellt ein Überraschungspaket mit Musik für iTunes dar.

Abbildung 2.2 beschreibt den Strukturteil der Visualisierung. Folgende Sprachelemente sind enthalten (Felfernig u. a., 2014):

- **Komponententypen** sind die dargestellten Entitäten. Sie besitzen einen eindeutigen Namen (z.B. 'Storage') und werden durch eine Menge von Attributen beschrieben (z.B. 'price', 'capacity'). Ein Attribut hat einen Datentyp, welcher eine Konstante, ein Wertebereich (z.B. [50...100]) oder eine Enumeration (z.B. ['gaming', 'media']) sein kann. Ein Komponententyp beschreibt das abstrakte Konzept eines Bauteils. Im dargestellten Beispiel ist eine 'CPU' etwas, was ein bis vier Kerne haben kann. Werden Attributwerte eines Komponententypen fest gewählt, wird daraus eine Instanz, d.h. ein konkretes Bauteil (z.B. ein Vierkernprozessor).

Tabelle 2.1.: Constraints des Konfigurationsmodells aus Abbildung 2.2

Name	Beschreibung
$GC_1$	Wird das 'iTunes-Music-Package' gewählt, muss auch das Betriebssystem ('OS') vom Typ 'OSX Yosemite' gewählt werden.
$GC_2$	Das 'OS' vom Typ 'OSX Yosemite' und der Arbeitsspeicher ('RAM') vom Typ 'Small RAM' können nicht gleichzeitig gewählt werden.
$PRC_1$	Der Preis des Notebooks ist die Summe der 'price'-Attribute der 'Storage'-, 'Motherboard'-, 'RAM'-, 'CPU'- und 'iTunes-Music-Package'-Komponenten.
$RESC_1$	Die Summe der 'needed capacity'-Attribute aller 'iTunes-Music-Package'-Komponenten darf die Summe der 'capacity'-Attribute aller 'Storage'-Komponenten nicht überschreiten.
$CRC_1$	Das 'OS' vom Typ 'OSX Yosemite' benötigt mindestens einen 'core'-Wert der 'CPU'-Komponente von 2.
$COMPC_1$	Das OS vom Typ 'Windows XP' ist inkompatibel mit einem 'size'-Wert des Notebooks von 13".

- **Generalisierungen** stellen die Verbindungen zwischen einem spezialisierten Subtyp zu einem allgemeineren Supertyp her. Damit muss der Wertebereich eines Attributs eines Subtypen eine Teilmenge des entsprechenden Attributwertebereichs des Supertypen sein. Komponententypen (z.B. 'Storage') werden so weiter spezialisiert (z.B. 'HDD'). Die Beziehung ist disjunkt und vollständig. Disjunkt bedeutet, dass jede Instanz eines Komponententypen nur genau einem seiner Subtypen entsprechen kann. Beispiel: Die Instanz eines 'Storage' kann eine 'HDD' oder eine 'SSD' sein, aber nicht beides. Vollständig bedeutet, dass die Subtypen alle tatsächlich möglichen Instanzen darstellen (z.B. gibt es für diese Konfiguration keine Komponente 'DVD' als möglichen Storage).
- **Assoziationen mit Kardinalitäten** beschreiben die Teil-Ganzes-Beziehungen zwischen Komponententypen. Die hier verwendete Variante ist die Komposition. Das bedeutet, dass keine Instanz eines Komponententypen Teil von mehr als einer anderen Instanz sein kann. Kardinalitäten beschreiben Assoziationen noch näher, indem sie sie durch Mengeninformationen ergänzen. Beispiel: Eine Notebook-Instanz besitzt ein oder zwei Storage-Instanzen. Eine Storage-Instanz kann nur Teil einer Notebook-Instanz sein.

Die Darstellung wird ergänzt durch Constraints. Sie gelten zwischen Komponententypen und/oder deren Attributen. Wenn möglich werden sie direkt im Diagramm dargestellt. Anderenfalls werden sie in einer Tabelle aufgelistet (siehe Tabelle 2.1). Es werden folgende Constrainttypen unterschieden (Felfernig u. a., 2014):

**GC: Grafische Constraints** können im Gegensatz zu anderen Constraints direkt im UML-Diagramm dargestellt werden. Ansonsten entsprechen sie einem der folgenden Typen.

**PRC: Preis-Constraints** beziehen sich auf die Preisbildung der Konfiguration. Bei realer Konfigurationssoftware sind diese jedoch meistens nicht Teil des Konfigurationsmodells. Stattdessen wird die Preisbildung durch eine eigene Softwarekomponente realisiert.

**RESC: Ressourcen-Constraints** beschränken die Produktion und den Verbrauch bestimmter Ressourcen.

Beispiel: Jedes 'iTunes-Music-Package' verbraucht 5(MB) Festplattenkapazität. Der verfügbare Speicher wird wiederum durch die Storage-Instanzen bestimmt. Wird nur ein Speichermedium in Form einer 'SSD' gewählt, hat das Notebook 250(MB) Festplattenkapazität. In diesem Falle ist die Obergrenze für 'iTunes-Music-Package'-Instanzen gleich 50.

**CRC: Abhängigkeits-Constraints** beschreiben, unter welchen Voraussetzungen bestimmte Instanzen Teil der Konfigurationslösung sein müssen.

**COMPC: Kompatibilitäts-Constraints** beschreiben die Kompatibilität oder Inkompatibilität bestimmter Komponententypen.

### 2.2.3. Konfigurationslösung

Auf Grundlage der eben vorgestellten Wissensrepräsentation lässt sich die Konfigurationsaufgabe genauer charakterisieren.

Die Lösung einer Konfigurationsaufgabe ist *vollständig* (jeder Komponententyp ist instanziiert) und konsistent (erfüllt alle Constraints) (angelehnt an Falkner u. a., 2011). Sind beide Bedingungen erfüllt, wird die Lösung als *korrekt* bezeichnet (Soininen u. a., 1998). Die Lösung einer Konfigurationsaufgabe ist gleichzeitig eine mögliche Variante des Produkts.

Eine Konfigurationslösung kann durch ein UML-Instanz-Diagramm visualisiert werden. Abbildung 2.3 zeigt eine mögliche Variante der Notebook-Konfiguration. Anstatt der Komponententypen sind nur noch konkrete Instanzen enthalten.

Abbildungen/notebookInstanceUML.pdf

Abbildung 2.3.: Visualisierung einer Konfigurationslösung als UML-Instanz-Diagramm

### 2.2.4. Konfiguratoren

Konfiguratoren „[...] führen den Abnehmer durch alle Abstimmungsprozesse, die zur Definition des individuellen Produktes nötig sind und prüfen sogleich die Konsistenz sowie Fertigungsfähigkeit der gewünschten Variante“ (Piller, 2006). Der erste Teil dieser Definition versteht den Konfigurationsprozess als eine Nutzerführung. Der zweite Teil der Definition entspricht der Charakterisierung als einen technischen Verarbeitungsprozess aus Kapitel 2.2.1. Es muss also die technische- und die anwenderbezogene Sicht unterschieden werden. Aus technischer Sicht ist der Konfigurationsprozess der Vorgang der Verarbeitung einer Konfigurationsaufgabe. Aus Anwendersicht bezeichnet er den Abstimmungsprozess. Beides gehört zum Verantwortungsbereich des Konfigurators. Es wird im Folgenden die Konvention verwendet, dass der Begriff „Konfigurationsprozess“ die Nutzerführung meint, der „technische Konfigurationsprozess“ die softwaretechnische Verarbeitung der Nutzereingaben.

Nach Piller (2006) besitzt ein Konfigurator drei Komponenten:

- Die **Konfigurationskomponente** führt den technischen Konfigurationsprozess durch. Sie wird auch als Konfigurationsengine bezeichnet (Tacton Systems AB, 2007).
- Die **Präsentationskomponente** erstellt eine Konfigurationsdarstellung in zielgruppenspezifischer Form. Sie stellt die Schnittstelle für die Abstimmungsprozesse mit dem Anwender dar.
- Die **Auswertungskomponente** präsentiert die Konfigurationslösung in einer Form, welche eine Interpretation der Variante außerhalb des Konfigurators erlaubt. Dies können z.B. Stücklisten, Konstruktionszeichnungen und Arbeitspläne sein.

#### Konfiguratorarten

Konfiguratoren für die Erhebung komplexer Anforderungen technischer Systeme müssen von Konfiguratoren für den Einsatz im Rahmen der MC unterschieden werden (Felfernig u. a., 2014). Erstere sind für den Experteneinsatz gedacht oder dienen nach Piller (2006) als Vertriebskonfiguratoren der Unterstützung des Verkaufsgespräches. Letztere werden direkt vom Kunden in einer Company-to-Customer Beziehung genutzt und werden auch als *Mass Customization Toolkits* bezeichnet. Die sogenannte *Selbstkonfiguration* ist eine Voraussetzung für MC, indem der zeitkonsumierende Prozess der Erhebung der Kundenbedürfnisse auf die Seite des Kunden verlagert wird (Piller, 2006)

Konfiguratoren können bei allen in Abschnitt 2.1.2 genannten Produktionskonzepten zum Einsatz kommen. Je nach Produktionskonzept hat der Konfigurationsprozess eine unterschiedliche Bedeutung. Bei PTO hat der Konfigurator eine Katalogfunktion, indem er den Anwender bei der Auswahl eines fertigen Produkts aus einer Produktpalette unterstützt. Bei ATO verhält sich der Konfigurator wie ein Variantengenerator, der den Anwender bei der Auswahl der richtigen Variante unterstützt. Dabei hat der Hersteller bereits alle möglichen Varianten vordefiniert. Daher werden diese als *herstellerspezifisch* bezeichnet (Schomburg, 1980). Der Anwendungsfall bei MTO ist ähnlich, jedoch wird durch die stär-

kere Einflussnahme des Kunden von *kundenspezifischen* Varianten gesprochen (Schomburg, 1980). Außerdem unterstützt ein Konfigurator die regelbasierte Generierung von Konstruktionsdaten. Bei ETO besteht ein erheblicher Neukonstruktionsbedarf. Dies widerspricht der Definition der Konfiguration als Routine Design aus Abschnitt 2.2.1 – die Spezifikation der beteiligten Objekte ist nicht vollständig bekannt. Konfiguratoren können hier nur einen begrenzten Beitrag leisten. Aus dieser Erläuterung lässt sich ableiten, dass das Haupteinsatzgebiet von Konfiguratoren im ATO/MTO Umfeld liegt.

### Zusammenfassung

In Abschnitt 2.1.2 wurde dargestellt, wie bestimmte Produktionskonzepte die Herstellung individualisierter Produktvarianten bei gleichzeitiger Lagerfertigung ermöglichen. Produkte werden mit dem Ziel gestaltet, so individuell und auftragsunabhängig wie möglich zu sein. Damit wurde einer der Schlüsselfaktoren für die Ermöglichung der hybriden Wettbewerbsstrategie MC erläutert. Diese verbindet die Vorteile effizienter Massenproduktion mit denen der kundenspezifischen Einzelfertigung (Piller, 1998). MC resultiert in Variantenvielfalt und damit in Produktkomplexität. In Abschnitt 2.2 wurde die Funktionsweise von Konfiguratoren vorgestellt, mit welcher sie zur Beherrschung der Produktkomplexität beitragen.

## 2.3. Webservices

Das W3C (2004) definiert Webservices lose als:

„[...] a software system designed to support interoperable machine-to-machine interaction over a network“

Die Definition schließt die Kommunikation heterogener Systeme ein. „Zwischen Systemen“ differenziert gleichzeitig klar von der klassischen Verwendung eines Programms, bei der ein (menschlicher) Anwender mit einem System kommuniziert. Tilkov (2011) bemerkt, dass Webservices damit sehr weich definiert sind; „nämlich eigentlich gar nicht“. Fest steht, dass hier ein System einen Dienst als Schnittstelle anbietet, welche von einem Clienten über Webtechnologien ansprechbar ist. Webservices sind demzufolge eine Möglichkeit zur Realisierung von Integrationsszenarien webbasierter Systeme.

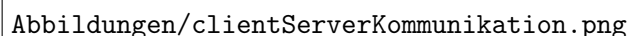
The image is a placeholder for a diagram titled 'Abbildungen/clientServerKommunikation.png'. It is represented by a rectangular box containing the filename text.

Abbildung 2.4.: Generische Client-Server Kommunikation bei Webservices

Abbildung 2.4 entspricht im wesentlichen der klassischen Client-Server Kommunikation im Web. Exemplarisch werden XML-Daten übertragen, was die zugrunde liegende Idee der

Webservices illustriert – die Übertragung anderer Daten als Webseiten mittels HTTP.

Wilde und Pautasso (2011) reden von zwei etablierten „Geschmäckern“ (flavors) in der Webservice-Welt: *SOAP* und *REST*. Die erste Geschmacksrichtung bedeutet Webservices „auf Basis von SOAP, WSDL und den WS-\*Standards - bzw. [...] deren Architektur“ (Tilkov, 2011). Hier wird ein XML-basierter Technologiestack beschrieben. REST hingegen ist ein Architekturstil, der 2000 in der Dissertation von Roy Fielding vorgestellt wurde. Der Versuch, beide Varianten direkt gegenüber stellen zu wollen, ist ein „[...] klassischer Apfel-Birnenvergleich: ein konkretes XML-Format gegen einen abstrakten Architekturstil“ (Tilkov, 2011).

Vor einer detaillierteren Diskussion von SOAP und REST wird zur Einordnung eine grundlegende Unterscheidung der Ansätze vorgestellt. Gemeinsam ist beiden, dass HTTP als Transportprotokoll zur Übertragung der Frage (Request) verwendet wird, die vom Server (Response) beantwortet werden soll. Eine HTTP-Nachricht besteht aus einem Header und einem Entity-Body zur Übertragung von Daten. Richardson und Ruby (2007) haben zwei Leitfragen herausgearbeitet, die in diesem Zusammenhang von den jeweiligen Ansätzen unterschiedlich beantwortet werden: Wo in der Nachricht sagt der Client dem Service, mit welchen Daten (*Fokusinformation*) was (*Methodeninformation*) gemacht werden soll?

Die **Fokusinformation** sagt aus, für welche Datenelemente sich der Client interessiert (z.B. ein Artikel eines Onlineshops). Bei REST ist dies der Uniform Resource Identifier (URI) (d.h. der Webadresse) zu entnehmen (z.B. <http://onlineshop.com/api/artikel/notebook>). ■ Bei SOAP steht diese Information in einer XML-Datei, welche im Entity-Body übertragen wird – die sogenannte Payload. Die **Methodeninformation** sagt aus, was mit dem identifizierten Datenelement geschehen soll (z.B. „erstelle einen neuen Notebook-Artikel“). Bei REST steht dies im Methodenfeld der HTTP-Headers, bei SOAP abermals im Entity-Body. Daraus lässt sich als grundlegender Unterschied ableiten: SOAP verwendet HTTP nur als Transportprotokoll, REST auch dessen Ausdruckskraft (Wilde und Pautasso, 2011).

### 2.3.1. SOAP

Bei SOAP-Web Services wird ein *Remote Procedure Call (RPC)* durchgeführt. Dabei handelt es sich um eine generelle Technik zur Realisierung von Systemverteilung. Ein System ruft die Funktion eines Systems aus einem anderen Adressraum auf. SOAP ist ein XML-basiertes Umschlagsformat, welches wiederum die Beschreibung eines Methodenaufrufs in XML-Form enthält. Bei SOAP-Webservices werden also RPCs über HTTP getunnelt (Wilde und Pautasso, 2011). Das ist Konvention, aber keine Notwendigkeit. Der SOAP-Umschlag ist transportunabhängig, könnte also auch von anderen Protokollen als HTTP übertragen werden (Tilkov, 2011). Solange es sich bei dem Transportprotokoll um eine Webtechnologie handelt, wird die Webservedefinition nicht verletzt.

Wie die Beschreibung des RPC aussehen muss, definiert die *Web Service Description Language (WSDL)*. Jeder SOAP-basierte Service stellt eine maschinenverarbeitbare WSDL-

Datei bereit. Darin werden die aufrufbaren Methoden, deren Argumente und Rückgabetypen beschrieben. Außerdem werden die Schemata der XML-Dokumente festgehalten, welche der Service akzeptiert und versendet (Richardson und Ruby, 2007).

Es existieren eine Vielzahl von Middleware-Interoperabilitätsstandards, die mit dem „WS“-Prefix versehen sind. Dabei handelt es sich um „XML-Aufkleber“ für den SOAP-Umschlag, die HTTP-Headern entsprechen (Richardson und Ruby, 2007). Sie erweitern die Ausdrucksmöglichkeit des SOAP-Formats (Wilde und Pautasso, 2011). Beispielsweise erlaubt *WS-Security* die Berücksichtigung von Sicherheitsaspekten bei der Client-Server-Kommunikation. Eine Übersicht der existierenden Standards ist dem Wiki für Webservices (WsWiki, 2009) zu entnehmen.

### 2.3.2. REST

„Eine Architektur zu definieren bedeutet zu entscheiden, welche Eigenschaften das System haben soll, und eine Reihe von Einschränkungen vorzugeben, mit denen diese Eigenschaften erreicht werden können.“ (Tilkov, 2011)

Dies ist in der Dissertation von Fielding geschehen, in der REST als *Architekturstil* definiert wird. Ein Architekturstil ist ein stärkerer Abstraktionsgrad als eine Architektur. Beispielsweise besitzt das Web eine Architektur, die eine HTTP-Implementierung von REST darstellt (Tilkov, 2011). Tatsächlich wurden die Einschränkungen von REST aber dem Web entnommen, indem Fielding es post-hoc als lose gekoppeltes, dezentralisiertes Hypermediasystem konzeptualisierte (Wilde und Pautasso, 2011) und dann von diesem Konzept abstrahiert hat. Einen Webservice nach dem REST-Architekturstil zu implementieren, passt ihn dem Wesen des Webs an und nutzt dessen Stärken (Tilkov, 2011).

Entsprechend Tilkovs Architekturdefinition werden im Folgenden die Einschränkungen von REST sowie die daraus resultierenden Eigenschaften besprochen.

#### 2.3.2.1. Einschränkungen

Einschränkungen sind (in eigenen Worten) Implementierungskriterien. Während Fielding in seiner theoretischen Abhandlung explizit vier solcher Kriterien nennt, basiert die folgende Erläuterung auf der praxiserprobten Variante der Sekundärliteratur (Wilde und Pautasso, 2011; Tilkov, 2011).

##### **Ressourcen mit eindeutiger Identifikation**

„Eine Ressource ist alles, was wichtig genug ist, um als eigenständiges Etwas referenziert zu werden“ (Richardson und Ruby, 2007). Identifiziert werden Ressourcen im Web durch *URIs*, die einen globalen Namensraum darstellen. Jede Ressource hat mindestens eine ID, eine ID kann jedoch nicht mehr als eine Ressource identifizieren. Es ist hervorzuheben, dass Ressourcen nicht das gleiche sind wie die Datenelemente aus der Persistenzschicht einer Anwendung. Sie befinden sich auf einem anderen Abstraktionsniveau (Tilkov, 2011).



Beispiel: Eine Warenkorbbressource (d.h. ein virtueller Warenkorb) mit der ID '1024' ist unter 'http://onlineshop.de/api/basket/1024'. Die einzelnen Warenkorbposition (d.h. die einzelnen Artikel im Warenkorb) haben jedoch keine URI.

Tilkov (2011) nimmt in diesem Zusammenhang eine Typisierung von Ressourcen vor. Von den sieben verschiedenen Ressourcentypen sind folgende im Rahmen der Arbeit interessant:

1. Bei einer **Projektion** wird die Informationsmenge verringert, indem eine sinnvolle Untermenge der Attribute einer abgerufenen Ressource gebildet wird. Zweck ist die Reduktion der Datenmenge.

Beispiel: Weglassen der Beschreibungstexte von Warenkorbpositionen.

2. Die **Aggregation** ist das Gegenteil. Hier werden Attribute unterschiedlicher Ressourcen zur Reduktion der Anzahl notwendiger Client/Server Interaktionen zusammengefasst.

Beispiel: Hinzufügen der Versandkosten beim Abruf des Warenkorbs.

## Hypermedia

Hypermedia beschreibt das Prinzip verknüpfter Ressourcen. Dies ermöglicht dem Client, neue Ressourcen zu entdecken und bestimmte Prozesse auszulösen (Wilde und Pautasso, 2011).

Beispiel (Ressource): Zu einer Warenkorbbressource wird für jede enthaltene Warenkorbposition eine URI zur jeweiligen Detailseite (d.h. die Seite, auf welcher der Artikel einzeln präsentiert wird) hinzugefügt.

Beispiel (Prozess): Zur einer Bestellbestätigungsressource wird der zugehörige Stornierungslink hinzugefügt.

## Standardmethoden/uniforme Schnittstelle

Oben wurde beschrieben, dass jede Ressource durch (mindestens) eine ID identifiziert wird. Jede URI unterstützt dabei den gleichen Methodensatz, welche mit den HTTP-Methoden korrespondieren. Übertragen auf die objektorientierte Programmierung bedeutet das: Jede Klasse implementiert das gleiche Interface. Folgende Teilmenge der neun verfügbaren HTTP-Methoden finden in der Literatur am häufigsten Erwähnung (Richardson und Ruby, 2007; Tilkov, 2011; Wilde und Pautasso, 2011):

**GET:** Das Abholen einer Ressource.

Beispiel: GET an 'http://onlineshop.com/api/artikel/notebook' holt die Notebook-Ressource. ■

**PUT:** Das Anlegen oder Aktualisieren einer Ressource. Je nachdem, ob unter dieser URI bereits eine Ressource existiert.

Beispiel: PUT an 'http://onlineshop.com/api/artikel/notebook' erzeugt eine Notebook-Ressource unter dieser Adresse, wenn noch keine andere Ressource besteht.

**POST:** Bei POST werden zwei Bedeutungen unterschieden. Im engeren Sinne bedeutet es das Anlegen einer Ressource unter einer URI, die vom Service bestimmt wird. Im weiteren Sinne kann durch Post ein Prozess ausgelöst werden.

Beispiel (im engeren Sinne): POST an 'http://onlineshop.com/api/artikel/notebook' mit einer JPG-Datei als Payload erzeugt eine neues Bild für die Notebook-Ressource. Die Bild-Ressource bekommt jedoch die URI 'http://onlineshop.com/api/images/notebook'.

Beispiel (im weiteren Sinne): POST an 'http://onlineshop.com/api/artikel/notebook' löst eine Bestellung für das Notebook aus.

**Delete:** Das Löschen einer Ressource.

Beispiel: DELETE an 'http://onlineshop.com/api/artikel/notebook' löscht die Notebook-Ressource.

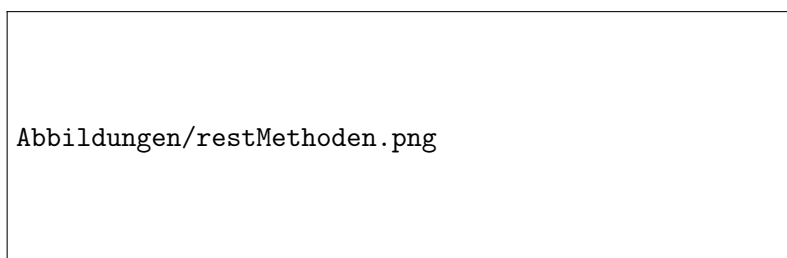


Abbildung 2.5.: HTTP-Methoden und ihre Eigenschaften<sup>a</sup> (Quelle: Tilkov (2011))

<sup>a</sup>Relevante Attribute im Rahmen der Arbeit: „Sicher“ bedeutet nebenwirkungsfrei, d.h. kein Ressourcen-zustand ändert sich durch diese Methode. „Idempotent“ bedeutet, dass das Resultat der Methode bei Mehrfachausführung das gleiche ist. „Identifizierbare Ressource“ bedeutet, dass die URI garantiert eine Ressource identifiziert.

Abbildung 2.5 fasst die Eigenschaften der Methoden aus der HTTP-Spezifikation 1.1 zusammen. Die Implementierung einer Methode muss dem erwarteten Verhalten aus dieser Spezifikation entsprechen. Die Praxis zeigt, dass nur die Methoden unterstützt werden, die für die jeweilige Ressource sinnvoll sind. Abbildung 2.5 macht außerdem klar, dass es für POST keinerlei Garantien gibt. Richardson und Ruby (2007) sehen bei der Verwendung des POST im weiteren Sinne (prozessbezogen) eine Verletzung der uniformen Schnittstelle. Dies wird damit begründet, dass das Resultat des Prozesses, der durch POST ausgelöst wird, der API-Beschreibung des Webservices und nicht der HTTP-Spezifikation zu entnehmen ist. Das schafft eine Abhängig von der Schnittstellenimplementierung, die gerade durch die uniforme Schnittstelle vermieden werden soll.

### Ressourcen und Repräsentationen

Beschreibt die Darstellungen einer Ressource in einem definierten Format. Der Client bekommt nie die Ressource selbst, sondern nur eine Repräsentation derer zu sehen. In der Praxis hat sich durchgesetzt, eine serialisierte Variante eines Objektes als JSON-Objekt zur Verfügung zu stellen (Tilkov, 2011).

Beispiel: Bereitstellung einer Bestellbestätigung als PDF und HTML.

### Statuslose Kommunikation

Bei REST soll ein serverseitig abgelegter, transienter, clientspezifischer Status über die Dauer eines Requests hinweg vermieden werden. Der Service benötigt also nie Kontextinformationen zur Bearbeitung eines Requests.

Beispiel: Ein Warenkorb wird nicht in einem Sessionobjekt, sondern als persistentes Datenelement gehalten.

### 2.3.2.2. REST-Konformität

Diese Auflistung legt folgende Frage nahe: Ist ein Webservice nur dann REST-konform, wenn alle Kriterien erfüllt werden? Was ist mit einem Webservice, der allen Einschränkungen gerecht wird, jedoch Ressourcen nur als JSON repräsentiert (ein Verstoß gegen die Forderung nach unterschiedlichen Repräsentationen)? Aus diesem Grund existiert das „Richardson Maturity Model“, welches die abgestufte Bewertung eines Webservices nach dessen REST-Konformität erlaubt. Es wird während der Auswertung in Kapitel 4.3 vorgestellt und zur Evaluierung der Implementierung genutzt.

### 2.3.2.3. Eigenschaften

Aus den vorgestellten Kriterien resultieren folgende Eigenschaften (Tilkov, 2011), welche die Vorteile REST-basierter Webservices gegenüber der SOAP-Konkurrenz darstellen (Richardson und Ruby, 2007):

**Lose Kopplung:** Beschreibt isolierte Systeme mit größtmöglicher Unabhängigkeit, die über Schnittstellen miteinander kommunizieren. Hierzu tragen die Standardmethoden bei.

**Interoperabilität:** Beschreibt die Möglichkeit der Kommunikation von Systemen unabhängig von deren technischen Implementierung. Dies ergibt sich durch die Festlegung auf Standards. Bei der Anwendung von REST auf Webservices sind dies Webstandards (z.B. HTTP, URIs).

**Wiederverwendbarkeit:** Jeder Client, der die Schnittstelle eines REST-basierten Service verwenden kann, kann auch jeden anderen beliebigen REST-basierten Service nutzen - vorausgesetzt, das Datenformat wird von beiden Seiten verstanden.

**Performance und Skalierbarkeit:** Webservices sollen schnell antworten, unabhängig von der Anzahl von Anfragen in einem definierten Zeitraum. Dies wird durch die Cachebarkeit (siehe HTTP-Methodenspezifikation in Abbildung 2.5) und Zustandslosigkeit erreicht. Da der Service keinen clientspezifischen Kontext aufbauen muss, müssen aufeinanderfolgende Requests nicht vom gleichen Server beantwortet werden.

### Zusammenfassung

REST ist Ressourcenorientiert, während SOAP aufgrund von RPCs Methodenorientiert ist. Die Umsetzung von REST ist nicht an technische Voraussetzungen gebunden, vielmehr müssen eine Reihe von Kriterien erfüllt werden. Das Implementierungswerkzeug von REST sind Funktionen, die auf HTTP-Requests an definierte URIs reagieren, diese verarbeiten und mit einer HTTP-Nachricht beantworten können. Bei SOAP ist HTTP nur das Transportprotokoll. Die Verarbeitung des Inhalts erfordert einen weiteren Technologiestack.

## 2.4. eCommerce

Im Folgenden wird durch die Charakterisierung des Begriffs *eCommerce* ein Anwendungsrahmen für Onlineshops (*eShops*) hergestellt. Deren softwaretechnische Umsetzung wird durch *eShop-Systeme* realisiert. Durch eine Kategorisierung der Systeme nach Anbieterstrategie wird abschließend die Menge der Open-Source-Lösungen für eine Konfiguratorintegration identifiziert.

### 2.4.1. Anwendungsrahmen

eShops gehören zur Domäne des elektronischen Handels (eCommerce). eCommerce ist „die elektronisch unterstützte Abwicklung von Handelsgeschäften auf der Basis des Internet“ (Schwarze und Schwarze, 2002). Je nachdem, welche Marktpartner an dem Handelsgeschäft teilnehmen, werden verschiedene Formen des eCommerce unterschieden. Die in Abbildung 2.6 fett hervorgehobenen Varianten werden von Meier und Stormer (2012) als „die zwei Geschäftsoptionen des eCommerce“ bezeichnet – *Business-to-Customer (B2C)* und *Business-to-Business (B2B)*. Bei B2C erfolgt der Handel von Produkten und Dienstleistungen zwischen Unternehmen und Endverbraucher, bei B2B zwischen Unternehmen.

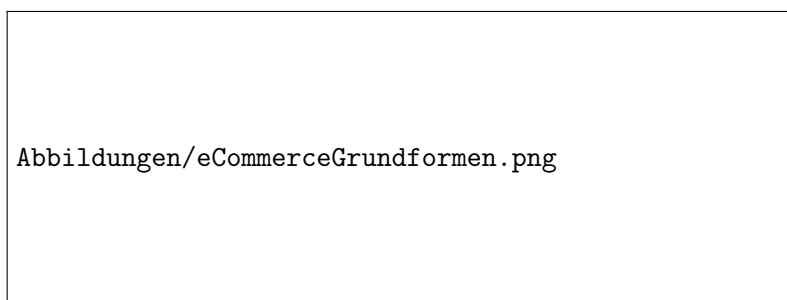


Abbildung 2.6.: Grundformen des eCommerce nach Marktpartnern (Quelle: Schwarze und Schwarze (2002))

Für die Umsetzung von eCommerce existieren unterschiedliche Geschäftsmodelle. Timmers (1998) nennt elf verschiedene Formen, wobei eShops eine davon sind. Es handelt sich dabei um ein „Geschäftsmodell der Angebotsveröffentlichung, bei dem ein Anbieter seine Waren oder Dienstleistungen über das Web den Nachfragern offeriert“ (Bartelt u. a., 2000).

Ein eShop bildet den traditionellen Einkaufsvorgang nach: Kunden können mittels einer Katalog- oder Suchfunktion über den Produktbestand navigieren (*Artikellisting*). Produkte können ausgewählt und ausführliche, mit Medien angereicherte Beschreibungen abgerufen werden (*Detailseite*). Wunschartikel werden einem virtuellen *Warenkorb* hinzugefügt. Ist die Produktauswahl abgeschlossen, begibt sich der Kunde zur „Kasse“, wo die Zahlungsmodalitäten erledigt werden (*Bestellung*) (Boles und Haber, 2000).

Ein eShop beschreibt das Geschäftsmodell, jedoch noch nicht dessen Umsetzung als Softwaresystem. Dieses wird als eShop-System bezeichnet (Boles und Haber, 2000) und im Folgenden behandelt.

### 2.4.2. eShop-Systeme

„eShop-Systeme sind Software-Systeme, die den Aufbau, die Verwaltung und den Einsatz von eShops unterstützen“ (Boles und Haber, 2000). Abbildung 2.7 zeigt die Grobarchitektur eines eShop-Systems nach Meier und Stormer (2012). Darin wird die Unterteilung zwischen Storefront und Backfront deutlich, welche in der Terminologie realer Shopsysteme als Front- und Backend bezeichnet werden (vgl. shopware AG, 2015e). Das Frontend ist der Interaktionsraum der Kunden, das Backend der administrative Bereich des Shopbetreibers.



Abbildung 2.7.: Grobarchitektur eines eShop-Systems (Quelle: Meier und Stormer (2012))

Die Hauptaufgaben eines eShop-Systems sehen Boles und Haber (2000) in den Bereichen *Merchandising* (z.B. Management von hierarchisch strukturierten Produktkatalogen, Beeinflussung des Shopdesigns), *Auftragsbearbeitung* (z.B. Festlegung der Abarbeitungs-Pipeline, Integration von Bezahlverfahren) und *Sonstiges* (z.B. die Kopplung mit externen ERP-Systemen). Der konkrete Funktionsumfang hängt vom gewählten eShop-System ab.

Die Systeme sind nach Strategie der Anbieter kategorisierbar, wie im Folgenden dargestellt wird.

### Open-Source

Der Quellcode von Open-Source-Systemen ist kostenlos verfügbar. Daher bieten sie völlige Gestaltungsfreiheit, aber keinen Herstellersupport. Die Dokumentationen sind schwächer und der Funktionsumfang geringer als bei kostenpflichtigen Alternativen. Andererseits existieren Communities, die Unterstützung bieten und die Entwicklung von Erweiterungen vorantreiben (Stahl u. a., 2015).

Open-Source bedeutet nicht per se, dass die Systeme ohne Kostenaufwand einsetzbar sind. Beim kommerziellen Handel der modularen Erweiterungen auf shopspezifischen Stores liegt eine der Erlösquellen der Open-Source-Strategie (vgl. der *Addon Marketplace* der *PrestShop SA*<sup>3</sup> oder das Extensionverzeichnis der *Opencart Limited*<sup>4</sup>).

### Kauf-Lösungen

Kauf-Lösungen werden kostenpflichtig lizenziert (z.B. shopware AG, 2015g). Sie bieten Herstellersupport, zusätzliche Dienstleistungen (z.B. Installation des Shops) und einen höheren Funktionsumfang (z. B. Schnittstellen zu verschiedenen Warenwirtschaftssystemen oder Zahlungsdienstleistern) (Stahl u. a., 2015). Die Hersteller bieten verschiedene Editionen mit teilweise erheblichen Preisunterschieden an<sup>5</sup>.

Im Rahmen eines Dual-License-Modells ist eine Open-Source **Community Edition** Teil des Editionsspektrums (t3n, 2014) (vgl. das Shopangebot der Magento Inc. (2015), shopware AG (2015g) oder OXID eSales AG (2015)). Durch den offenen Quellcode existiert auch hier der Handel modularer Erweiterungen, von dem auch die kostenpflichtigen Varianten profitieren (vgl. der *Plugin Store* der shopware AG<sup>6</sup>). Die Codebasis aller Editionen ist gleich. Daher kann später zu einer Kauf-Lösung migriert werden. Das bietet Flexibilität für wachsende Shopanforderungen.

### Miet-Shops

Miet-Shops entsprechen einer Cloud-Lösung als *Software-as-a-Service* (z.B. Strato AG (2015), Shopify (2015)). Die technische Infrastruktur wird vom Provider zur Verfügung gestellt. Systemwartung, Bereitstellung der Shopsoftware und Hosting werden unter dem Mietpreis abgerechnet. Stahl u. a. (2015) bewertet diese Variante als Einstiegslösung mit geringer Gestaltungsfreiheit.

### Eigenentwicklungen

Wenn die Standardsysteme die Anforderungen nicht erfüllen, eignen sich Eigenentwicklungen für individuelle Bedürfnisse (Stahl u. a., 2015; Graf, 2014).

Aus dieser Darstellung sind die (zumindest initial) kostenfreien Varianten ersichtlich: reine Open-Source eShop-Systeme sowie die Community-Editionen der Dual-License Modelle.

---

<sup>3</sup><http://addons.prestashop.com>

<sup>4</sup><http://www.opencart.com/index.php?route=extension/extension>

<sup>5</sup>Beispiel: die Preisdifferenz der *Magento Enterprise Edition* zu *Enterprise Premium* liegt bei über 35.000 \$ (vgl FWP shop, 2014)

<sup>6</sup><http://store.shopware.com/>

Eine Anbieterübersicht ist t3n (2014) zu entnehmen. Eine Kategorisierung der Systeme nach Anforderungsklassen ist Graf (2014) zu entnehmen.

**Zusammenfassung** Im vorangegangenen Kapitel wurde das theoretische Fundament für die Arbeit gelegt. Durch die Aufstellung eines Anwendungsrahmens wurde ein wirtschaftlicher Kontext für Konfiguratoren im Allgemeinen geschaffen. Im Zusammenhang mit den Geschäftsoptionen des eCommerce lässt sich eine genauere Aussage über den Anwendungsbezug von Konfiguratoren in eShops machen. Die Konfiguration im B2C wird durch den Endverbraucher durchgeführt, entspricht also der Selbstkonfiguration. Im B2B ist beispielsweise im Rahmen des Einsatzes von Vertriebskonfiguratoren durch ausgebildetes Personal die Konfiguration komplexerer Systeme möglich. REST und SOAP ermöglichen die Instrumentierung des Webs für Integrationsszenarien.

## 3. Analyse

Nachdem in Kapitel 2 Konfiguratoren und eShop-Systeme unabhängig von konkreten Systemumsetzungen betrachtet wurden, findet nun eine Analyse von existierenden softwaretechnischen Implementierungen statt. Festgelegt durch die *Lino GmbH* kommt der *Tacton Produktkonfigurator* zum Einsatz. Im Folgenden wird dessen herstellerspezifisches Konfigurationsmodell analysiert. Daraufhin wird eine Konfiguratoranwendung in Form des Vertriebskonfigurators *TCsite* vorgestellt. Aus der Analyse dieses Systems kann eine genauere Aussage über die in Frage kommenden eShop-Systeme für die Integration gemacht werden. Daraufhin wird ein System festgelegt und mit dessen Betrachtung fortgefahren. Davon ausgehend findet im Fazit des Kapitels ein Grobkonzept der jeweiligen Verantwortungsbereiche der Systeme im Integrationsszenario statt.

### 3.1. Tacton Produktkonfigurator

Die *Tacton Systems AB* (Tacton) wurde 1998 als Spin-Off des *Schwedischen Instituts für Informatik* (SICS) gegründet (Tacton Systems AB, 2007). In der Forschungseinrichtung wurde als Resultat der Untersuchungen im Bereich *Wissensbasierte Systeme* und *Künstliche Intelligenz* der Tacton Produktkonfigurator entwickelt. Dieser interaktive Konfigurator ist die Basis der verschiedenen Produkte der Firma. Tacton bietet Lösungen im Bereich *Vertriebskonfiguration* und *Design Automation* (Automatisierung der Konstruktion in CAD-Systemen) (Tacton Systems AB, 2015).

#### 3.1.1. Konfigurationsmodell

Das in Abschnitt 2.2.2 vorgestellte Visualisierungskonzept abstrahiert Konfigurationswissen in einen Struktur- und einen Regelteil. Im Tacton-Konfigurationsmodell wird ebenfalls abstrahiert, jedoch in andere Domänen (Tacton Systems AB, 2006):

- (a) **Strukturinformation:** Wie ist das Produkt hierarchisch aufgebaut?
- (b) **Komponenteninformation:** Aus was ist es aufgebaut?
- (c) **Constraint-Informationen:** Wann ist das Produkt korrekt?
- (d) **Ausführungsinformationen:** Welche Fragen bekommt der Nutzer während des Konfigurationsprozesses in welcher Reihenfolge gestellt?

Dabei sind zwei Sachverhalte feststellbar:



- (1) Typisch für einen modellbasierten Konfigurator wird Produktwissen und Problemlösungswissen bei der Wissensmodellierung separiert (Felfernig u. a., 2014).

Beispiel: Ein Constraint, der die Kompatibilität bestimmter Betriebssysteme mit einer bestimmten Anzahl Prozessorkerne ausdrückt, soll wirken, auch ohne die konkreten CPU-Typen (z.B. *Intel i7*) zu kennen.

Darum ist die Rede von *generischen Constraints* – sie beziehen sich auf alle *Komponenten* (erläutert im folgenden Kapitel) eines Typs (Felfernig u. a., 2014).

- (2) Gemäß (d) wird im Konfigurationsmodell auch die Nutzerinteraktion festgelegt. Damit geht der Funktionsumfang eines Tacton-Modells über die Definition aus Abschnitt 2.2.1 hinaus.

Bei der UML-Wissensrepräsentation aus Abschnitt 2.2.2 wurden die Domänen (a) und (b) als Diagramm sowie (c) als Tabelle ausgedrückt. Tacton wählt eine andere Aufteilung. (b) wird unter dem Begriff *Components* umgesetzt, (a) und (c) werden unter dem Begriff *Configuration* zusammengefasst. Components und Configuration setzen gemeinsam den Sachverhalt (1) um und werden im Folgenden besprochen. Daraufhin wird die *Execution* besprochen, welche Sachverhalt (2) darstellt.

#### 3.1.1.1. Components und Configuration

Abbildungen/tactonModellHighLevel.pdf

Abbildung 3.1.: High-Level-Architektur des Tacton-Konfigurationsmodells

Abbildung 3.1 zeigt das High-Level-Konzept der Modellarchitektur. Unter *configuration* wird die Produktstruktur als hierarchischer Baum von *Part*-Objekten dargestellt. Jeder Part kann Constraints enthalten, die sich auf den Knoten selbst und alle seine Kinder

beziehen. Ein Part ist ansonsten nur ein Komponentenplatzhalter. Noch ist keine Information darüber hinterlegt, welches Bauteil dort eigentlich verkörpert wird. Abbildung 3.2 veranschaulicht das Konzept am Notebook-Beispiel.

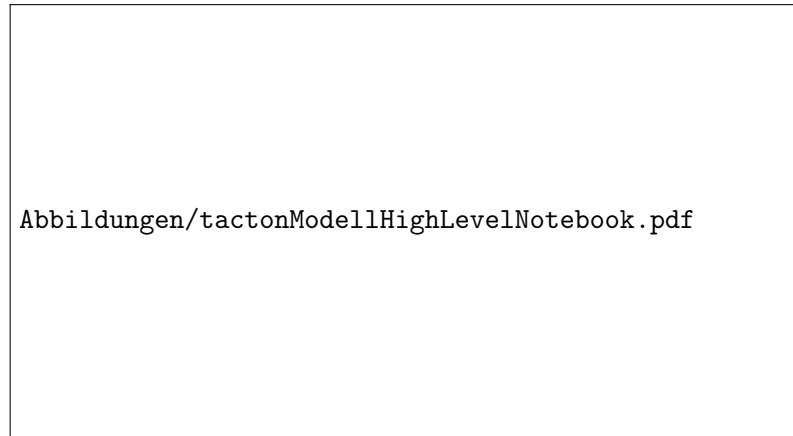


Abbildung 3.2.: Part-Struktur der exemplarischen Notebook-Konfiguration

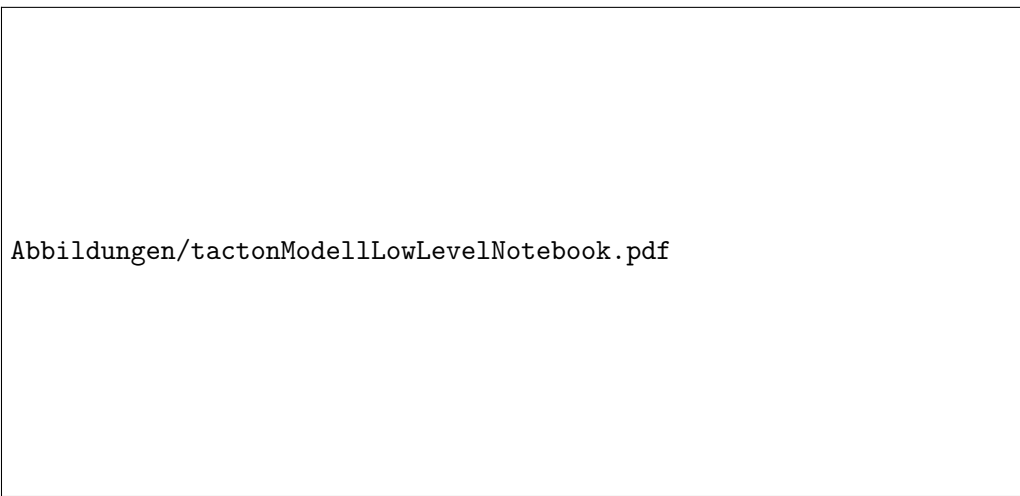
Die Informationen über die eben erwähnten Bauteile (d.h. Komponententypen) werden isoliert unter *components* definiert (siehe Abbildung 3.1). Ein Bauteil wird in *Component Classes* und *Components* abstrahiert. Das ist analog zu Komponententypen, die in einer Generalisierungsbeziehung zueinander stehen (siehe Kapitel 2.2.2). Hier wurde eine Terminologie gewählt, die Supertyp und Subtyp unterscheidbar macht. Eine *Component Class* (z.B. ein 'RAM') wird durch Eigenschaften beschrieben, die als *Features* bezeichnet werden. Jedes Feature besitzt einen Namen (z.B. 'memory') und einen Wertebereich (z.B. ['2GB', '4GB']), welche als *Domain* bezeichnet wird. Wertebereiche können Integer, Float, Boolean, andere Component Classes oder selbstdefinierte Enumerationen sein. *Components* (z.B. eine 'SSD') sind das, was eine Component Class (z.B. ein 'Storage') konkret sein kann. Sie übernehmen alle Features der übergeordneten Component Class und besitzen konkrete Werte (*Values*) aus dem jeweiligen Wertebereich.



Abbildungen/tactonModellLowLevel.pdf

Abbildung 3.3.: Zuordnung von Parts und Component Classes

Abbildung 3.3 zeigt die Zuordnung zwischen Parts und Component Classes via Verzeigerung. Dabei wird einem Part eine Component Class durch eine *realized by*-Beziehung zugeordnet. Dem Part werden dabei die Features der entsprechenden Component Class vererbt (türkis dargestellt). Nur werden sie zur besseren Differenzierung beim Part nicht mehr als Features, sondern als *Attributes* bezeichnet. Für einen Part können auch noch zusätzliche Attribute definiert werden (als 'additional Attributes' dargestellt), falls notwendig. Die Angabe *Instances* (grün dargestellt) entspricht den Kardinalitäten in Abschnitt 3.3. *Constraints* (gelb dargestellt) werden als logische Ausdrücke formuliert. Sie bestehen aus Attributwerten, die mit mathematischen Zeichen in Relation gebracht werden.



Abbildungen/tactonModellLowLevelNotebook.pdf

Abbildung 3.4.: Zuordnung von Part und Component Class der exemplarischen Notebook-Konfiguration

Abbildung 3.4 veranschaulicht den Sachverhalt anhand eines Ausschnitts der exemplarischen Notebook-Konfiguration. Der Part übernimmt die Attribute 'memory' und 'price'. Das zusätzliche Attribut 'type' kapselt beispielsweise die Bezeichnungen über die jeweiligen Components als Enumeration (z.B. ['Small RAM', 'Big RAM']). Der Constraint veranschaulicht exemplarisch, wie die Inkompatibilität mit dem Betriebssystem vom Typ 'OSX Yosemite' formuliert werden würde.

#### 3.1.1.2. Execution

Es wurde dargestellt, wie das Konfigurationswissen im Tacton Konfigurationsmodell definiert wird. Dadurch ist aber noch nicht gesagt, welche Entscheidungen ein Nutzer während der Konfiguration treffen kann. Nicht jeder Part und nicht jedes Attribut muss eine relevante Wahl darstellen. Vielleicht sollen dem Nutzer sogar Fragen auf einem anderen Abstraktionsniveau als auf Komponentenebene gestellt werden. Statt „Soll eine HD oder eine SSD als Festplatte in das Notebook eingebaut werden?“ kann auch gefragt werden: „Möchten Sie viel Speicherplatz oder einen schnellen Speicherzugriff?“.

Der Abstimmungsprozess durch den Anwender wird unter dem Begriff *Execution* definiert. Dabei wird nicht einfach nur eine Menge an Optionen festgelegt, die dem Nutzer am Ende als Liste präsentiert werden. Stattdessen werden die Optionen hierarchisch strukturiert.



Abbildung 3.5.: Ausschnitt der generischen Execution-Struktur

Abbildung 3.5 zeigt einen Ausschnitt der generischen Struktur der Execution als Baum. Die vollständige Darstellung ist Anhang A.1 zu entnehmen und wichtig zum weiteren Verständnis. Abbildung 3.6 veranschaulicht das Konzept durch eine mögliche Execution-Struktur der exemplarischen Notebook-Konfiguration.

1. Die Wurzel wird als **Applikation** bezeichnet. Sie beinhaltet die Gesamtheit aller Optionen.

Beispiel: Eine Notebook-Konfiguration.

2. Die nächste Knotenebene wird als **Steps** bezeichnet. Sie legen fest, in welcher Reihenfolge die Optionen präsentiert werden. Das ist in verschiedenen Fällen sinnvoll. Zum Beispiel dann, wenn komplexe Probleme in mehrere Schritte unterteilt oder Werte für spätere Schritte gesetzt werden sollen.

Beispiel: In Step 1 legt der Anwender eine Preisgrenze fest. In Step 2 wird die technische Spezifikation getroffen, wobei die Preisgrenze aus Step 1 nicht überschritten werden darf.

3. Nun folgen 1..n Knotenebenen, die als **Groups** bezeichnet werden. Sie fassen Optionen zu logischen Einheiten zusammen. Groups geben keine Bearbeitungsreihenfolge vor.

**Top-Level-Groups:** Die oberste Group-Ebene ist obligatorisch.

Beispiel: Eine 'Speicher'-Gruppe in Step 2.

**optional Groups:** Die Unterteilung in weitere Gruppen ist optional.

Beispiel: Unter der 'Speicher'-Gruppe wird eine weitere Gruppe namens 'Massenspeicher' angelegt, um sie vom 'RAM' zu unterscheiden.

4. **Fields** sind die Blätter des Execution-Baums. Sie sind das, worüber Optionen gewählt werden. Alle darüber liegenden Knoten haben den Fields nur eine Ordnungsstruktur gegeben.

Aus Anwendersicht besteht ein Field aus einer Beschreibung und einem Interaktionselement. Aus technischer Sicht repräsentiert ein Field einen *Parameter* mit einem Wertebereich (*Domain*), aus welchem ein Wert (*Value*) gewählt wird. Die Parameter stammen aus den Attributen der Parts. Das Wählen einer Option bedeutet also letztendlich das Wählen des Wertes eines Part-Attributs.

Je nach Interaktionselement wird in unterschiedliche Feldtypen unterschieden:

**Menu:** Wahl aus einem Wertebereich, z.B. via Dropdownmenü.

Beispiel Massenspeicher:

Parameter = 'Storage-Part.type', Domain = ['HD', 'SSD']

Beispiel CPU-Kerne:

Parameter = 'CPU-Part.cores', Domain = [1..4].

**Number:** Eingabe eines eigenen Wertes in ein Textfeld.

Beispiel Preisgrenze (Eingabe in Schritt 1):

Parameter = 'Notebook-Part.maxPrice', Domain = [integer]

**Label:** Anzeige eines Wertes aus Informationsgründen.

Beispiel Aktueller Preis (Anzeige in Schritt 2):

Parameter = 'Notebook-Part.price', Domain = [integer]

Die Feldtypen *Menu* und *Number* haben gemein, dass der Anwender einen Wert aus einem Wertebereich festlegt. Beim *Menu* ist dieser Wertebereich eine Menge vordefinierter Optionen, bei *Number* ein Zahlenbereich, wobei der konkrete Wert vom Anwender selbst eingegeben wird. Um die Formulierung kurz zu halten, werden im Folgenden sowohl die ausgewählte Option als auch der eingegebene Wert unter dem Begriff *Eingabeparameter* zusammengefasst.



Abbildung 3.6.: Mögliche Exection-Struktur der exemplarischen Notebook-Konfiguration

Das Modell wird in einem XML-basierten Format mit der Dateiendung '.tcx' abgelegt. Die Entwicklung einer solchen Datei wird von entsprechenden Programmen unterstützt. Hierfür kann zum Beispiel „TCstudio“ genutzt werden, welches eine grafische Oberfläche zur Modellentwicklung bietet (Tacton Systems AB, 2015).

### Zusammenfassung

Das Tacton-Modellierungskonzept wurde vorgestellt. Neben dem Konfigurationswissen werden darin auch die Optionen des Anwenders definiert. Diese werden durch Steps in eine Reihenfolge gebracht und durch Groups logisch gegliedert. Über Felder wählt der Anwender Optionen. Der Feldtyp Number ermöglicht die Eingabe eines eigenen Wertes. So werden Komponenten definiert, die nach Kundenanforderung konstruiert oder gefertigt werden

müssen. Entsprechend der vorgestellten Produktklassifizierung in Kapitel 2.1.2 bildet das Konfigurationsmodell so auch MTO-Produkte ab.

Das Konfigurationsmodell definiert die Interaktionsstruktur. Diese muss jedoch noch in Form einer Konfigurationsoberfläche dargestellt werden. Neben anderen Funktionalitäten realisiert TCsite eine solche Darstellung. Die Anwendung wird im folgenden analysiert.

### 3.1.2. TCsite

*TCsite* ist ein webbasierter Vertriebskonfigurator. Er könnte theoretisch von Endkunden genutzt werden, bildet aber nicht den Einkaufsprozess eines eShops gemäß Kapitel 2.4.1 nach. Stattdessen handelt es sich um eine **CPQ**-Lösung (**Configure-Price-Quote**). Das bedeutet: der Anwender wird durch die Konfiguration geführt, was in einer Preiskalkulation resultiert, woraus wiederum ein Angebot erstellt wird. Eine unmittelbare Bestellung und Zahlungsabwicklung ist nicht vorgesehen. Der Anwendungsbereich liegt also im B2B (Tacton Systems AB, 2015).

Technisch betrachtet ist TCsite eine Webanwendung. Der serverseitige Programmcode ist in Java verfasst. Im Lieferumfang sind *Apache Tomcat* als Application Server sowie *TC-server* enthalten. TCserver ist das, was gemäß Kapitel 2.2 als eigentlicher Konfigurator bezeichnet wird. Er beherbergt die *Konfigurationsengine*. Damit ist das System gemeint, welches Konfigurationsaufgaben verarbeitet (Tacton Systems AB, 2013). Abbildung 3.7 veranschaulicht die High-Level-Architektur des Standardsetups.



Abbildung 3.7.: High-Level-Architektur von TCsite

#### 3.1.2.1. Architektur

Abbildung 3.8 visualisiert die offene Schichtenarchitektur von TCsite. Offen bedeutet, dass jede Schicht mit allen darunter liegenden Schichten kommunizieren kann. Das Fundament bildet die Persistenzschicht. Sie realisiert die Datenhaltung. Die Plattform bildet ein Application Programming Interface (API) mit den Basisfunktionalitäten für die darüber lie-

genden Schichten. Die Module realisieren jeweils eine der drei Oberflächenbereiche der Anwendung. Außerdem bieten sie Services und Erweiterungspunkte für die oberste Schicht – die Plugins. Durch diese kann die Funktionalität von TCsite individuell erweitert werden (Tacton Systems AB, 2014b).

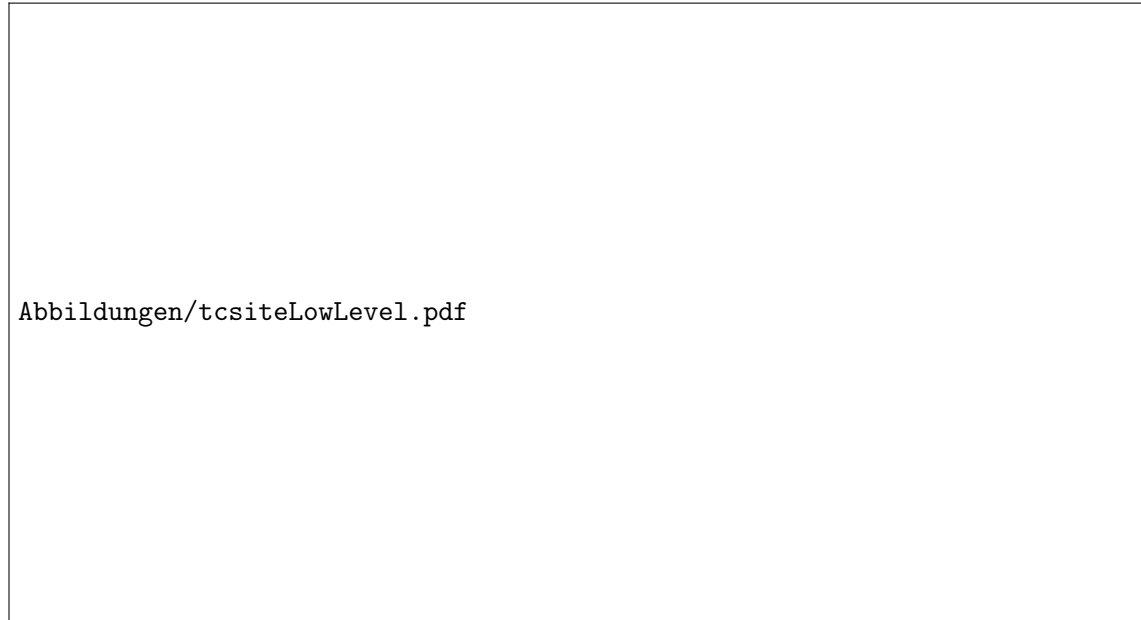


Abbildung 3.8.: Schichtenarchitektur von TCsite

Im Folgenden werden die Architekturkomponenten einzeln vorgestellt.

### **Repository**

Das *Repository* bietet eine Datenbank sowie ein Reihe von Funktionen zum Lesen und Schreiben der TCsite-Objekte. Es ist in einen lokalen und einen globalen Speicher strukturiert. Wird ein Objekt erstellt oder geöffnet, geschieht die Bearbeitung immer auf einer Arbeitskopie in dem für jeden Nutzer spezifischen lokalen Speicher. Änderungen sind solange für andere Nutzer unsichtbar. Erst ein „Commit“ sorgt für die nutzerübergreifende Sichtbarkeit im globalen Speicher. Bei dieser Übertragung wird eine Revisionshistorie über Objektänderungen geführt, so dass alte Zustände wieder abrufbar sind (Tacton Systems AB, 2014b).

### **Plattform**

Die Plattform bietet keine eigene Oberfläche, sondern bildet eine API mit grundlegenden Funktionalitäten für die darüberliegenden Schichten.

### **Administration**

Dieses Modul bildet die Administrationsoberfläche, dargestellt in Anhang A.2. Hier werden Einstellungen getroffen und Verwaltungsaspekte realisiert. Dazu gehört zum Beispiel die Verwaltung der Nutzergruppen und Produkte.

Die Nutzergruppen definieren, welche Rechte ein TCsite-User hat. Standardmäßig verfü-



bar sind die Nutzergruppen „Standarduser“, „Systemadministrator“ und „Integrationsuser“. Letzterer Typ dient der Authentifizierung bei der Kommunikation mit externen Systemen, zum Beispiel bei einem Integrationsszenario. Bildlich gesprochen personifiziert jede externe Webanfrage einen Gast, der die Maske des Integrationsusers aufgesetzt und dessen Rechte bekommt.

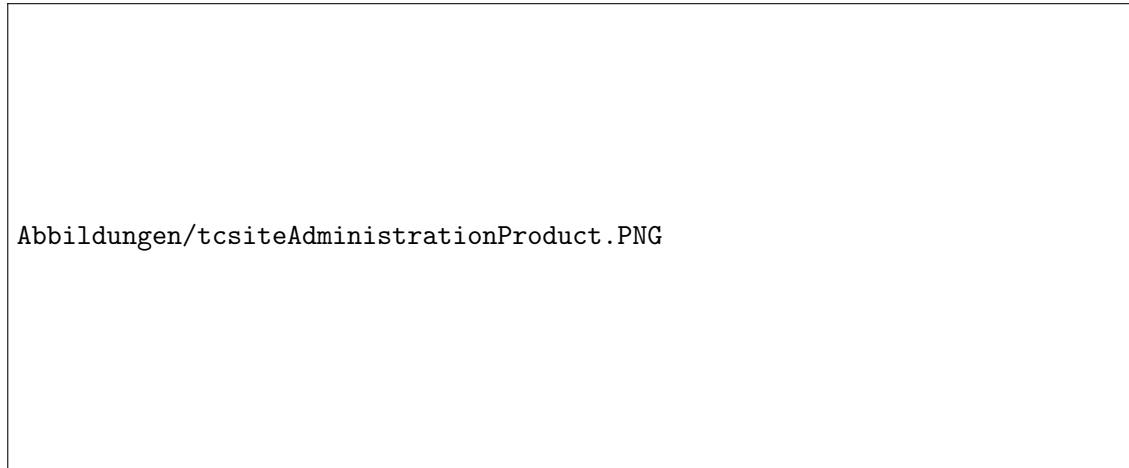



Abbildung 3.9.: Detailsansicht eines Produktes aus dem Produktkatalog

Außerdem wird hier der Produktkatalog verwaltet. Abbildung 3.9 zeigt die Detailsansicht eines Produkts. Die Darstellung macht deutlich, dass neben Produktbeschreibungen auch Dateien hinterlegt werden. Wird eine Modelldatei hochgeladen, ist das Produkt konfigurierbar. Ressourcen, wie z.B. Bilder oder eine Übersetzungsdatei können ergänzend hinzugefügt werden (Tacton Systems AB, 2014c). Entsprechend der Definition eines Konfigurationsmodells in Kapitel 2.2 werden so nicht alle Produktvarianten explizit ausdefiniert und angelegt, sondern implizit definiert.

### Quotation

Das Quotation-Modul erstellt die *Quotation-View*, dargestellt in Abbildung 3.10. Technisch betrachtet realisiert das Modul Methoden für die Manipulation und Anzeige von *Quotations* sowie zur Dokumentengenerierung aus den mit ihr im Zusammenhang stehenden Dateien. Aus Anwendersicht ist eine Quotation ein Angebot, dass die fertig konfigurierbaren Produkte enthält. Diese werden als *QuotationItems* bezeichnet und unter 'Products' gelistet (siehe [1]) (Tacton Systems AB, 2014b). Betrachtet man eine Quotation als einen (realen) Warenkorb, sind die *QuotationItems* die Artikel darin. Der Vergleich hinkt insofern, als dass die Artikel im Warenkorb auch unabhängig von diesem existieren. Sie stehen zum Beispiel im Warenregal. Ein *QuotationItem* ist jedoch eine konkrete Variante eines konfigurierbaren Produktes, die nur innerhalb der Quotation existiert.



Abbildungen/tcsiteQuotationNumbered.pdf

Abbildung 3.10.: Quotation-View von TCsite

Der Konfigurationsprozess wird über die Schaltfläche 'Add Product' gestartet (siehe [2]). Daraufhin wählt der Anwender das gewünschte Produkt aus einer Liste aus. Im Anschluss erfolgt der Übergang zum Configuration-Modul (Tacton Systems AB, 2014c).

Eine Quotation hat weiterhin einen Lebenszyklus, welcher unter 'Lifecycle State' dargestellt wird (siehe [3]). Das Angebot befindet sich zu jedem Zeitpunkt in einem bestimmten Zustand, welche über den Administrationsbereich verwaltet werden. Die Zustandsabfolge definiert den sogenannten *Workflow*. Zustände unterscheiden sich in der Sichtbarkeit für verschiedene Nutzergruppen und deren Editierbarkeit. Beispielsweise kann sich eine Quotation in den Zuständen „Design“ oder „Offered“ befinden. In letzterem Zustand ist sie nicht mehr änderbar, damit keine Inkonsistenz zum herausgegebenen Angebot entstehen kann (Tacton Systems AB, 2014c).

Die Box auf der rechten Seite stellt den Index aller Quotations dar (siehe [4]). Unter dem Suchfeld ist einstellbar, ob nur die Quotations des eingeloggten Nutzers gezeigt werden, oder ob nutzerübergreifend gelistet werden sollen. Wie im Administrationsbereich erwähnt, agieren auch externe Systeme als Nutzer, nämlich als Integrationsuser. Auch Quotations, die von dieser Nutzergruppe stammen, werden hier gelistet (Tacton Systems AB, 2014c).

## Configuration

Das Configuration-Modul realisiert den C-Teil des CPQ-Prozesses – die Konfiguration. Es rendert die Konfigurationsoberfläche und verwaltet die mit der Konfiguration im Zusammenhang stehenden Objekte (Tacton Systems AB, 2014b). Die technischen Verarbeitungsschritte des Konfigurationsprozesses geschehen jedoch im TCserver. Somit ist die Konfiguration als Client-Server Architektur implementiert. Das Configuration-Modul besitzt Serviceklassen, die die Kommunikation mit der Konfigurationsengine abstrahieren (Tacton Systems AB, 2014a). Ansonsten ist der TCserver nicht unmittelbar ansprechbar.

### 3.1.2.2. Konfigurationsprozess

Abbildung 3.11 veranschaulicht die Kommunikation zwischen TCsite und TCserver während des Konfigurationsprozesses. Das Configuration-Modul führt wiederholt zustandslose Aufrufe (*configure*) der Konfigurationsengine durch, welche jeweils eine Antwort produzieren (*result*) (Tacton Systems AB, 2014b). Dabei weichen *configure* und *result* semantisch von den Definitionen der Konfigurationsaufgabe und -lösung aus Kapitel 2.2.1 ab.



Abbildung 3.11.: Kommunikation zwischen TCsite und TCserver

Um diese Abweichung zu verstehen, muss rekapituliert werden, dass interaktive Konfiguratoren einen Mechanismus zum merken des Konfigurationszustands besitzen (siehe Kapitel 2.2.1). Ein Konfigurationszustand beinhaltet bei Tacton, welche Eingabeparameter in welchem Step gewählt wurden. Oben wurde festgestellt, dass die Aufrufe zustandslos sind. Also wird kein serverseitiger Kontext aufgebaut, der den Konfigurationszustand enthält. Die Alternative ist die Übertragung des Zustands in jedem Aufruf. Demzufolge enthält der *configure*-Aufruf (1.) das Konfigurationsmodell, (2.) den Konfigurationszustand und fakultativ (3.), den aktuellen Eingabeparameter (Tacton Systems AB, 2014a). Diese Bestandteile werden ab jetzt unter dem Begriff *Konfigurationsengine-Input* (KE-Input) zusammenge-

fasst. Die Antwort (*result*) beinhaltet primär den neuen Konfigurationszustand. Aus dem *result* können weiterhin diverse Informationen extrahiert werden – die finale Instanziierung (Konfigurationslösung), der Endpreis sowie auch die Datengrundlage zum Rendern der Konfigurationsoberfläche (Tacton Systems AB, 2014a). Diese Bestandteile werden ab jetzt unter dem Begriff *Konfigurationsengine-Output* (KE-Output) zusammengefasst.

Der Konfigurationszustand und ein Verweis auf das Konfigurationsmodell, auf das sich der Konfigurationszustand bezieht, werden in der Klasse *Configurable* zusammengefasst und persistiert. Das *QuotationItem* (vorgestellt in Kapitel 3.1.2.1) steht in einer 1-zu-1 Beziehung mit einem *Configurable*.

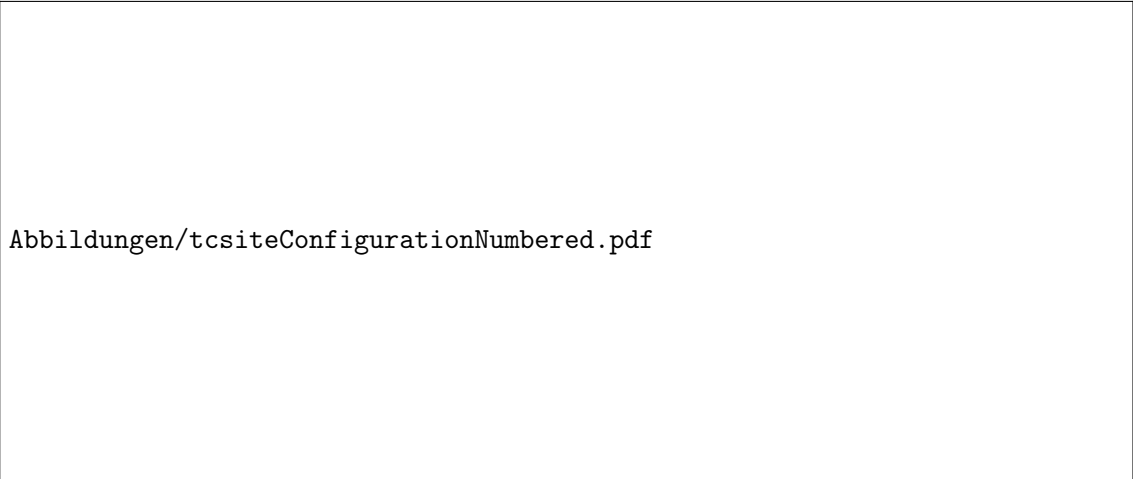
Die Kommunikation läuft in drei Phasen ab (siehe Abbildung 3.11):

**initiate configuration:** Der erste KE-Input beinhaltet noch keinen Eingabeparameter. Er dient dem initialen Aufbau der Konfigurationsoberfläche entsprechend der festgelegten Struktur in der Execution aus dem Konfigurationsmodell. Der Konfigurationsprozess startet immer in Step 1. Die Steps müssen in der vordefinierten Reihenfolge abgearbeitet werden.

**loop:** Nach jeder Wahl einer Option wird die Konfigurationsengine mit dem Eingabeparameter aufgerufen und daraufhin die Oberfläche neu gerendert. Die Oberfläche spiegelt dabei den Konfigurationszustand wieder, d.h. die bisher gewählten Optionen werden hervorgehoben.

**finish configuration:** Wird die Konfiguration im letzten Step vom Anwender abgeschlossen, erfolgt ein finaler Aufruf der Konfigurationsengine. Aus dem KE-Output ist die finale Instanziierung (Konfigurationslösung), extrahierbar (Tacton Systems AB, 2014b).

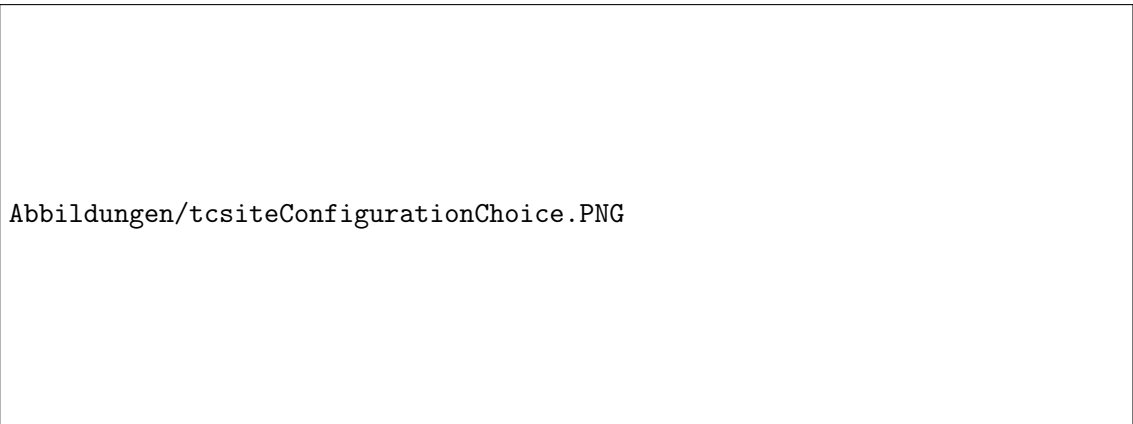
Der Konfigurationsprozess in TCsite wird anhand der exemplarischen Notebook-Konfiguration veranschaulicht. Abbildung 3.12 zeigt die Konfigurationsoberfläche in TCsite. Diese befindet sich im zweiten Step. Mittels [1] wird zwischen den Steps gewechselt. [2] erlaubt die Navigation durch die Top-Level-Groups des aktuellen Steps. [3] stellt die Felder der aktuellen Top-Level-Group dar. Anhang A.3 zeigt anhand der 'Speicher'-Gruppe die Darstellung der Unterteilung in weitere Gruppen. Außerdem wird unter [4] noch eine weitere Top-Level-Group angezeigt – die sogenannte *Info-Group*. Sie wird im Konfigurationsmodell durch die Bezeichnung „Info“ ausgezeichnet. Sie wird parallel zur gerade aktiven Top-Level-Group im jeweiligen Step dargestellt.



Abbildungen/tcsiteConfigurationNumbered.pdf

Abbildung 3.12.: Configuration-View in TCsite der exemplarischen Notebook-Konfiguration

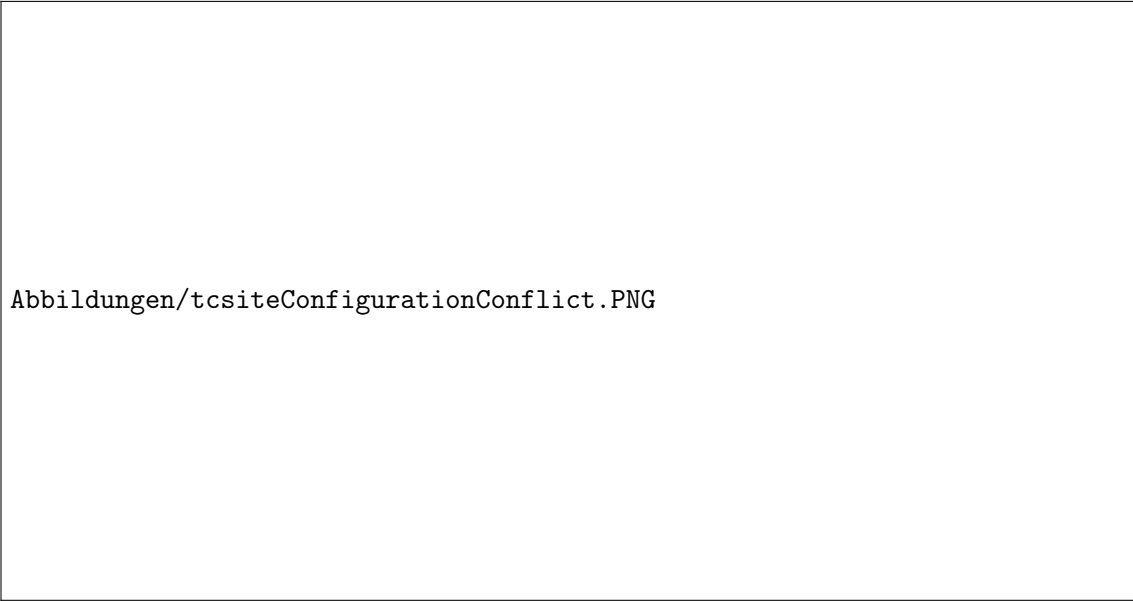
Es fällt auf, dass bereits Optionen gewählt sind - obwohl der Anwender noch gar nichts entschieden hat. Offensichtlich hat die Konfigurationsengine initial *Default-Values* gesetzt. Sie sind durch den grauen 'Please Confirm'-Hinweis gekennzeichnet.



Abbildungen/tcsiteConfigurationChoice.PNG

Abbildung 3.13.: Configuration-View in TCsite der exemplarischen Notebook-Konfiguration nach Anwenderwahl

Abbildung 3.13 zeigt die Konfigurationsoberfläche, nachdem der Anwender eine Option gewählt hat (Parameter: 'OS.type', Value: 'Windows XP'). Die gewählte Option wird *unverhandelbar*. Das wird gekennzeichnet durch das grüne 'Confirmed'. Außerdem wird überprüft, wie die gewählte Option die anderen verfügbaren Optionen beeinflusst. Was bei der Auswahl zu einem Konflikt führen würde, wird ausgegraut. Beim Feld 'Displaygröße' ist zu beobachten, dass die Wahl der Konfigurationsengine von '13' auf '15' geändert wurde. Offensichtlich sind Werte, die nicht vom Anwender gesetzt wurden, *verhandelbar*. Die Konfigurationsengine sorgt folglich immer für einen konfliktfreien Zustand. Durch einen Klick auf 'Confirm' bzw. 'Please Confirm' können Optionen zwischen verhandelbar und unverhandelbar gewechselt werden.



Abbildungen/tcsiteConfigurationConflict.PNG

Abbildung 3.14.: Configuration-View in TCsite während einer Konfliktsituation

Ausgegraute Optionen sind dennoch wählbar. Abbildung 3.14 zeigt die Konfigurationsoberfläche nach einem Klick auf die Displaygröße '13'. Dennoch ist '15' noch immer die aktive Option. Gleichzeitig wird am oberen Ende eine Konflikthinweis angezeigt. Die Konfigurationsengine weist darauf hin, dass Optionen im Widerspruch stehen und schlägt eine Konfliktlösung vor. Diese kann angenommen oder abgelehnt werden. Im letzteren Fall bleibt die Konfiguration einfach in dem Zustand vor der Wahl der konfliktverursachenden Option. Auch die Eingabe von Werten in ein Textfeld kann zu Konflikten führen. Beispielsweise dann, wenn diese außerhalb der Domain des Parameters liegen oder im Widerspruch mit einer anderen Wahl stehen.

Wird der Konfigurationsprozess über die Schaltfläche 'Ok' beendet, kehrt TCsite zum Quotation-Modul zurück. Dort erscheint die Variante in der Liste der QuotationItems.

### 3.1.2.3. Erweiterbarkeit

Zur Erweiterung der Funktionalität bietet TCsite ein Pluginkonzept, welches Abbildung 3.15 visualisiert. Neben den oben diskutierten Verantwortlichkeiten definieren die Plattform und die Module sogenannte *Extensionpoints* (Tacton Systems AB, 2014a). Extensionpoints sind das, was in anderen Erweiterungskonzepten als Events bezeichnet wird – Ereignisse im Kontrollfluss der Geschäftsprozesse. Plugins können Methoden bereitstellen, die an den entsprechenden Ereignisstellen aufgerufen werden (Tacton Systems AB, 2014b).



Abbildung 3.15.: TCsite Pluginkonzept

Jedes Plugin hat Zugriff auf das Repository, die Plattform und das Modul, zu dem es gehört. Die Plattform stellt außerdem ein paar spezielle Erweiterungspunkte bereit. Sie erlaubt unter anderem die Definition von *Webcontrollern*. Das sind Methoden, die auf HTTP-Requests an bestimmte URIs innerhalb des TCsite-Addressraums reagieren, diese verarbeiten und mit einer HTTP-Nachricht beantworten.

### Zusammenfassung

Der Produktkatalog von TCsite besteht nicht aus explizit ausdefinierten Varianten, sondern Produkten mit Konfigurationsmodellen. Erst innerhalb einer *Quotation* werden kundenspezifische Varianten durch einen Konfigurationsprozess generiert, welche als *QuotationItems* bezeichnet werden. Der Konfigurationsprozess ist die Führung des Anwenders durch die in einem Konfigurationsmodell definierte *Execution*, bei dem in jedem Step Optionen gewählt und Eingaben getätigt werden.

Der Konfigurationsprozess passiert im Configuration-Modul. Technisch betrachtet werden dabei *KE-Inputs* verarbeitet, die mit *KE-Outputs* beantwortet werden. Ein KE-Input besteht aus dem Konfigurationsmodell, dessen Konfigurationszustand und (fakultativ) einem Eingabeparameter. Der Konfigurationszustand bezeichnet die Menge aller bisherigen Eingabeparameter. Das Konfigurationsmodell und der Konfigurationszustand werden durch das *Configurable* gekapselt, der Eingabeparameter resultiert aus den Nutzerinteraktionen mit der Konfigurationsoberfläche. Die Konfigurationsoberfläche wird aus den Daten des *KE-Outputs* erstellt.

### 3.1.3. Zwischenfazit

Die Daten aus dem KE-Output können auch einem externen System zum Aufbau einer Konfigurationsoberfläche zur Verfügung gestellt werden. Über die Oberfläche wählt der Anwender Optionen und tätigt Eingaben, woraufhin die Eingabeparameter an TCsite zurückgegeben werden. Dort wird in Verbindung mit dem entsprechenden *Configurable* ein

KE-Input definiert. Über den KE-Output kann die Konfigurationsoberfläche im externen System wiederum aktualisiert werden und der Prozess wiederholt sich. Die für diesen Vorgang relevanten Informationen müssen nur irgendwie zwischen dem externen System und TCsite transportiert werden.

Das Pluginkonzept erlaubt die Definition von Webcontrollern. Die in Kapitel 3.1.2.3 vorgestellte Definition eines Webcontrollers entspricht dem Umsetzungswerkzeug von REST, welches im Fazit des Kapitels 2.3.2 genannt wurde. Dadurch ist es möglich, einen Webservice zu implementieren, der eine Schnittstelle für externe Konfigurationsoberflächen bereitstellt.

Potentieller Client dieses Services ist jedes System, welches HTTP nutzen kann. Das trifft auf jede Webanwendung zu. eShop-Systeme für den Endverbrauchermarkt sind per se Webanwendungen. Somit können technische Bedingungen an das eShop-System für das Integrationsszenario formuliert werden: (1) das System muss modifizierbar sein, (2) die Modifizierung muss einen Webservice in einer Client-Server Architektur nutzen können und (3) die Verwaltung von Varianten muss integraler Bestandteil des Systems sein.

Jedes Shopsystem, für welches auf Grundlage dieser Bedingungen eine Daten- und Workflowschnittstelle entwickelt werden kann, kommt für das Integrationsszenario in Frage. Ein interner Evaluierungsprozess der Lino GmbH, bei dem zusätzlich ökonomische Faktoren bewertet wurden, hat zur Favorisierung von *Shopware* für die weitere Entwicklungsarbeit geführt.

## 3.2. Shopsystem

Shopware wird von der *shopware AG* in Schöppingen entwickelt. Das eShop-System ist aus einer Individuallösung gewachsen, welche 2003 von den heutigen Geschäftsführern entwickelt wurde. Infolgedessen wurde das Unternehmen 2008 als Vertriebsgesellschaft für die Software gegründet. Aktuell liegt die fünfte Version vor. Shopware wird in einem Dual-License Modell vertrieben. Die Community-Edition wird unter der GNU Open-Source Lizenz *AGPLv3* angeboten. Außerdem existieren drei kommerzielle Versionen (shopware AG, 2015h).

Im Folgenden wird Shopware einer technischen Analyse unterzogen. Darauf aufbauend wird die Erweiterbarkeit des Systems besprochen. Anschließend wird diskutiert, wie die Produktkonfiguration in der Standardinstallation technisch realisiert wird. Daraus kann eine Abgrenzung zum Tacton-Produktkonfigurator gewonnen und gleichzeitig dessen Integration motiviert werden. Die abschließende Betrachtung der Konfiguration im Einkaufsprozess liefert Ansatzpunkte für das Integrationskonzept.




### 3.2.1. Architektur

Shopware basiert auf der Programmiersprache PHP und verwendet eine relationale MySQL Datenbank. Grundlage des Technologiestacks ist das *Zend-Framework*, welches in der eigenentwickelten Abwandlung namens *Enlight* vorliegt (shopware AG, 2015a).

Abbildung 3.16 zeigt einen Überblick über die Shopware-Architektur, welche im Folgenden erklärt wird. Eine mit Beispielen angereicherte Variante dieser Abbildung ist Anhang A.4 zu entnehmen.

Shopware implementiert das Architekturmuster *Model View Controller (MVC)*. Das *Model* definiert die Datenstrukturen des Systems (z.B. Artikel, Kategorien, Bestellungen etc.). *Doctrine* wird für die objektrelationale Abbildung (ORM) verwendet. Dadurch wird eine Abstraktionsschicht über der Datenbank aufgebaut (shopware AG, 2015a). Das Framework ermöglicht die zentrale Definition der Datenbankstruktur in PHP und einen objektorientierten Datenzugriff (shopware AG, 2014b).



Abbildungen/shopwareMVCSHORT.pdf

Abbildung 3.16.: High-Level-Architektur von Shopware

Die *View* nutzt die Templating-Engine *Smarty*. Sie erweitert die HTML-Syntax um spezielle *Smarty-Tags*. Diese ermöglichen unter anderem die Definition von Variablen zur Darstellung von Daten aus dem Model. Außerdem können über *Smarty-Tags* sogenannte *Blöcke* definiert werden. Es handelt sich hierbei um adressierbare Bereiche innerhalb eines Templates, die das Markup strukturieren. Sie spielen eine Rolle für das Erweiterungskonzept (shopware AG, 2015a).

*Controller* sind Klassen, die HTTP-Requests entgegennehmen und eine Präsentation der Antwort durch Vermittlung zwischen Model und View entwickeln. Sie besitzen Methoden, welche als *Actions* bezeichnet werden. Diese sind bestimmten URIs zugeordnet. Controller sind nach Verantwortungsbereichen aufgeteilt. Diese werden als Module bezeichnet und folgendermaßen typisiert (shopware AG, 2014b):

- **Frontend**-Controller sind für die Storefront zuständig. Das betrifft alle Seiten, die ein Kunde sieht.  
Beispiele: Artikelliste einer bestimmten Kategorie, Detailseite eines Artikels, Warenkorb, Nutzeraccount, ...
- **Widget**-Controller generieren wiederverwendbare Bestandteile der Storefront.  
Beispiel: Liste der meistverkauften Artikel.
- **Backend**-Controller sind für die Datenverwaltung der Shopadministration zuständig. Sie generieren jedoch keine Einzelseiten wie im Falle der Frontend-Controller. Stattdessen ist der administrative Bereich als Single-Page-Application mittels des Javascript-Frameworks *Ext JS* implementiert. Dieses stellt eine Menge an Steuerelementen (z.B. Menüs, Formulare, etc.) zur Verfügung.  
Beispiele: Nutzerverwaltung, Artikelverwaltung, Pluginverwaltung, ...
- Für externe Systeme, die mit den Ressourcen von Shopware interagieren möchten, ist eine **REST-API** implementiert. Ein vollständige Übersicht aller Ressourcenendpunkte ist shopware AG (2014a) zu entnehmen.  
Beispiel: Abrufen des Artikels mit der ID 167.

Der Großteil der Logik ist jedoch nicht in den Actions, sondern den sogenannten *Core*-Klassen implementiert (shopware AG, 2014b). Beispielsweise wird der Request zum Hinzufügen einer Warenkorposition zwar vom Checkout-Controller entgegengenommen, die notwendigen Geschäftsprozesse finden aber in der Serviceklasse 'sBasket' statt.

### 3.2.2. Erweiterbarkeit

Der Quellcode der Community-Edition liegt offen. Theoretisch ist eine Erweiterung der Funktionalität über ein direktes Eingreifen in den Shopware-Kern denkbar. Der Hersteller sieht jedoch eine Anpassung über Plugins vor. Entsprechend der MVC-Architektur wird in *logische Erweiterungen* (betrifft Controller), *Daten-Erweiterungen* (betrifft Model) und *Template-Erweiterungen* (betrifft View) unterschieden (shopware AG, 2015a).

#### Logische Erweiterungen

Logische Erweiterungen werden über *Events* und *Hooks* realisiert. Events sind „definierte Ereignisse, die im Workflow des Shops auftreten“ (shopware AG, 2014b). Plugins können Code registrieren, welcher an den Ereignispunkten ausgeführt wird. Steht kein Event für die geplante Modifikation zur Verfügung, kann Plugin-Code über das Hooksystem unmittelbar auf Funktionen des Shopware-Kerns registriert werden (shopware AG, 2015a). Damit ist das Erweiterungskonzept flexibler als das von TCsite. Im Folgenden werden die Konzepte detaillierter vorgestellt.

Events werden in *Controller-Events* und *Notify-Events* unterschieden (shopware AG, 2014b). Controller-Events sind an den *Dispatch*-Vorgang gekoppelt. Der shopware-Entwickler Nögel (2015a) definiert Dispatching als einen Prozess, bei dem das Request-Objekt gehandhabt,

daraus das relevante Modul, der Controller und die Action extrahiert, der entsprechende Controller instanziiert und dieser zur Behandlung des Requests gebracht wird. Leitet ein Controller den Request zu einem anderen Controller weiter, wiederholt sich dieser Vorgang. Plugins können auszuführenden Code registrieren, der vor (*PreDispatch*) oder nach dem Dispatching (*PostDispatch*) ausgeführt werden soll. Außerdem können sie den Dispatch-Prozess auf eigene Funktionen umleiten und so ganze Controller-Methoden ersetzen. Notify-Events entsprechen hingegen den in Kapitel 3.1.2.3 erwähnten Extension-Points von TCsite. Sie finden beispielsweise in den Core-Klassen Verwendung. So kann abseits der Controller in den Programmablauf eingegriffen werden (shopware AG, 2014b).

Hooks bieten einen generischeren Ansatz. Events sind auf den Dispatchprozess und alle sonstigen Punkte beschränkt, an denen ein Shopware-Entwickler den Eingriff eines Plugins vorgedacht hat. Das Hooksystem bezeichnet die Möglichkeit, jede Public- und Protected-Funktion von Shopware zu erweitern. Hooks erlauben die Modifizierung der Eingangsparameter und Rückgabewerte der Originalfunktion sowie deren komplette Ersetzung (Nögel, 2015b).

### Daten-Erweiterungen

Im Gegensatz zu TCsite erlaubt Shopware die Erstellung und Modifizierung von Datenbanktabellen. Sollen bestehende Datenmodelle nur um bestimmte Eigenschaften ergänzt werden, kommt das *Attributsystem* in Frage. Gewisse Shopware-Entitäten (z.B. *s\_user* für Shopkunden) haben Attributtabellen (z.B. *s\_user\_attributes*) in einer 1-zu-1 Beziehung zugeordnet. Plugins können diesen Tabellen beliebige Spalten hinzufügen (z.B. die Lieblingsfarbe des Kunden) (shopware AG, 2015a).

### Template-Erweiterungen

Die View bietet Erweiterungspunkte über das Smarty-Block-System. Die Blöcke können durch eigenen Templatecode ersetzt oder erweitert werden. Weiterführend sind so eigene CSS- und Javascriptdateien im Seitenkopf einbindbar. Folglich kann mittels Template-Erweiterungen auch clientseitige Logik realisiert werden.

Abschließend wird bemerkt, dass Oberflächenerweiterungen im administrativen Bereich nicht durch Smarty realisiert werden, sondern durch das dort eingesetzte Framework *Ext JS*. Dies wird nicht in der offiziellen Dokumentation erwähnt. Stattdessen weisen die Orientierungsbeispiele für Backenderweiterungen implizit darauf hin (siehe shopware AG (2015c)). *Ext JS*-Erweiterungen sind ein komplexer Sachverhalt und werden in dieser Arbeit nicht weiter behandelt.

### 3.2.3. Konfiguration

Shopware behauptet in der offiziellen Funktionsübersicht, bereits in der Standardinstallation einen Konfigurator zu besitzen (shopware AG, 2015b). Im Folgenden wird dessen Implementierung analysiert. Daraufhin wird dessen Integration in den Einkaufsvorgang diskutiert.

### 3.2.3.1. Technische Analyse

Ein Artikel, den ein Kunde kaufen kann, wird über den entsprechenden Menüpunkt im Backend angelegt (siehe Anhang A.5). Über das Userinterface kann dieser als *Variante-Artikel* gekennzeichnet werden. Dadurch wird der Variantenreiter aktiviert, unter dem der Administrator alle Varianten des Artikels festlegt.

Dazu werden sogenannte *Gruppen*<sup>1</sup> (z.B. 'Storage') angelegt, die wiederum *Optionen*<sup>2</sup> (z.B. 'HDD' oder 'SSD') besitzen. Das ist äquivalent zu den Begriffen *Parameter* und *Domain* in der Execution des Tacton-Konfigurationsmodells, vorgestellt in Kapitel 3.1.1.2. Es werden also die Wahlmöglichkeiten des Anwenders definiert, jedoch ohne Konfigurationswissen. Die Gruppen und Optionen werden separat in der Datenbank gespeichert und sind Artikelübergreifend verwendbar. Über den Button 'Varianten generieren' (siehe Anhang A.6) werden alle Varianten explizit in der Datenbank angelegt, die sich aus der Kombination der Optionen ergeben. Übertragen auf das Notebook-Beispiel ergibt das folgende Rechnung:

$$\begin{aligned}
 & 3_{\text{Massenspeicher1}} \\
 & \cdot 3_{\text{Massenspeicher2}} \\
 & \cdot 2_{\text{Arbeitsspeicher}} \\
 & \cdot 2_{\text{Displaygröße}} \\
 & \cdot 2_{\text{Betriebssystem}} \\
 & \cdot 4_{\text{AnzahlCPU-Kerne}} \\
 & \cdot 99_{\text{AnzahlTunes-Music-Pakete}} \\
 & = 28512 \text{ Varianten}
 \end{aligned}$$

Damit passiert das, was durch das Konfigurationsmodell laut Kapitel 2.2.1 verhindert werden soll – das explizite Definieren und Abspeichern aller Varianten. Es gibt keine Constraints. Dementsprechend muss der Administrator händisch alle inkorrekten Varianten löschen und die Preise der korrekten Varianten nachtragen.

Die in Kapitel 2.2.1 vorgestellte Konfigurationsdefinition von Sabin und Weigel (1998) fordert Constraints. Somit stellt Shopware keinen Konfigurationsprozess bereit, sondern die Selektion einer explizit definierten Variante entsprechend der gewählten Optionen des Anwenders aus einer Datenbank.

### 3.2.3.2. Einkaufsvorgang

Ausgehend von dieser Analyse ist der Einkaufsvorgang eines Varianten-Artikels unter Berücksichtigung der technischen Prozesse als Aktivitätsdiagramm in Abbildung 3.17 darstellbar .

<sup>1</sup>Nicht zu verwechseln mit den Gruppen des TCsite-Modells, siehe Kapitel 3.1.1.2

<sup>2</sup>Ebenfalls nicht zu verwechseln mit den Optionen des TCsite-Modells

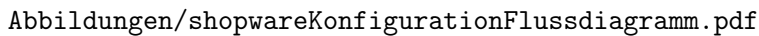
The image is a large rectangular box containing the text 'Abbildungen/shopwareKonfigurationFlussdiagramm.pdf'. This likely represents a placeholder for a diagram or image that is not rendered in this view.

Abbildung 3.17.: Aktivitätsdiagramm des Einkaufsvorgangs eines Varianten-Artikels in Shopware

Mittels der Katalog- oder Suchfunktion wählt der Kunde einen konfigurierbaren Artikel aus - d.h., er wurde im Backend als *Varianten-Artikel* gekennzeichnet. Auf dessen Detailseite wird die erste Variante des Artikels angezeigt. Dazu muss angemerkt werden, dass im Backend eingestellt wird, welche Variante eines Artikels initial geladen werden soll. Diese Variante wird ab jetzt als *Basisvariante* bezeichnet. Neben beschreibenden Texten und Bildern befindet sich auf der Detailseite die Konfigurationsoberfläche (siehe Anhang A.7). Zu jeder Gruppe wird ein Menü mit den entsprechenden Optionen angezeigt. Die Auswahl einer Option führt zum Reload der Seite mit der aktuellen Variante. Dieser Vorgang wiederholt sich so lange, bis der Artikel den Vorstellungen des Kunden entspricht. Außerdem wird die gewünschte Kaufmenge (Anzahl) eingestellt. Über den „In den Warenkorb“-Button wird der Artikel in den Warenkorb gelegt. Ist diese Variante bereits im Warenkorb, wird die Anzahl der entsprechenden Warenkorbposition inkrementiert. Anderenfalls entsteht eine neue Warenkorbposition. An der Bezeichnung der Warenkorbpositionen ist ablesbar, welche Optionskombination jeweils vorliegt (siehe Anhang A.8). Nun überprüft der Kunde den Warenkorb. Möchte er weitere Artikel hinzufügen, beginnt der Prozess mit der Auswahl des Wunschartikels von vorne. Ist der Kunde mit der Konfiguration einer Warenkorbposition noch nicht zufrieden, kann er diese löschen oder ändern. Möchte er sie ändern, kommt er über einen Klick auf die entsprechende Position zur Detailansicht dieser Variante zurück. Nun können andere Optionen gewählt werden. Ein erneuter Klick auf „In den Warenkorb“ aktualisiert nicht etwa die zugehörige Warenkorbposition, sondern erzeugt eine neue. Die

alte Variante verbleibt zusätzlich im Warenkorb. Ist der Nutzer mit allen Positionen im Warenkorb zufrieden, wird der Bestellvorgang über den Checkout-Prozess abgeschlossen.

Dieser Vorgang stellt den Lebenszyklus einer Variante in Shopware dar. Er beginnt mit dem Anlegen aller Varianten eines Artikels im Backend. Das ist ein Unterschied zu der Umsetzung in TCsite, welche in Kapitel 3.1.2.1 besprochen wurde. Hier beginnt der Lebenszyklus erst mit der Konfiguration.

### 3.3. Fazit

Shopware bietet in der Standardinstallation keine Funktionalität, die der Definition eines Konfigurationsprozesses gerecht wird. Stattdessen wird eine *Variantenselektion* durchgeführt. Da alle Varianten explizit vordefiniert sind, werden dadurch die in Kapitel 2.1.2 beschriebenen ATO-Produkte abgebildet. Die Analyse in Kapitel 3.1.1 hat ergeben, dass das Tacton-Konfigurationsmodell auch MTO-Produkte abbilden kann. Eine Integration des Tacton-Produktkonfigurators in Shopware würde somit dessen Angebotspektrum in Bezug auf Produktionskonzepte erweitern.

In Kapitel 3.2.2 wurde das Plugin-Konzept von Shopware vorgestellt. Template-Erweiterungen erlauben einen Eingriff in die Oberfläche. Damit ist eine Konfigurationsoberfläche in Shopware darstellbar. Die Analyse des Einkaufsvorgangs von konfigurierbaren Artikeln in Kapitel 3.2.3.2 impliziert jedoch Änderungen über die reine Integration einer Konfigurationsoberfläche hinaus. Nach dem Konfigurationsprozess absolvieren Varianten weitere Stationen im Einkaufsvorgang. Für die Arbeit bedeutet dies, dass ein Shopware-Plugin (Shop-Plugin) zu erstellen ist, welches (1.) eine Konfigurationsoberfläche bereitstellt und (2.) die resultierenden Varianten in den Einkaufsvorgang integriert.

Im Zwischenfazit der TCsite-Analyse (Kapitel 3.1.3) wurde geschlussfolgert, dass eine Konfigurationsoberfläche auch in einem externen System umsetzbar ist. Über Webcontroller kann ein Webservice implementiert werden, mit dem die Konfigurationsoberfläche kommuniziert. Webcontroller werden via Plugin definiert. Zusätzlich wurde oben festgestellt, dass das Shopware-Plugin nicht nur eine Konfigurationsoberfläche bereitstellen, sondern auch die resultierenden Varianten in den Einkaufsvorgang integrieren muss. Für die Arbeit bedeutet dies, dass ein TCsite-Plugin zu erstellen ist, welches (1.) eine Schnittstelle für externe Konfigurationsoberflächen anbietet und (2.) alle sonstigen Daten zur Integration einer Variante in den Einkaufsvorgang zur Verfügung stellt.

Nach diesem Grobkonzept der Verantwortungsbereiche wird in der folgenden Anforderungsanalyse definiert, was die beiden Plugins im Detail leisten müssen.



## 4. Anforderungsanalyse

Es ist ein System zu entwickeln, welches die Verbindung zwischen TCsite und Shopware herstellt. Dieses System wird in Abbildung 4.1 in eine Beziehung mit den Elementen gebracht, welche aus der Analyse des vorherigen Kapitels hervorgehen. Der Systemkontext stellt die Umgebung dar, der die für die Entwicklung relevanten Aspekte beinhaltet. Was das System hingegen nicht beeinflusst, ist Teil der irrelevanten Umgebung.

In Kapitel 3.1.2.1 wurde erwähnt, dass die Kommunikation mit dem TCserver vom Configuration-Modul abstrahiert wird. Von außen ist also nicht ersichtlich, dass der Konfigurationsprozess durch eine Serveranwendung realisiert wird. TCserver ist also Teil der irrelevanten Umgebung.

Zum Systemkontext gehört TCsite, da es das Configuration-Modul beherbergt. Außerdem verwaltet die Anwendung die Konfigurationsmodelle und -zustände. Weiterhin ist Shopware teil des Kontexts. Dort soll eine Konfigurationsoberfläche zur Verfügung gestellt und die resultierenden Varianten in den Einkaufsvorgang integriert werden. Der Shopkunde und -administrator sind die relevanten Stakeholder von Shopware. Sie werden mit dem Plugin interagieren. Im Zentrum des Systemkontexts befindet sich das zu entwickelnde System selbst.

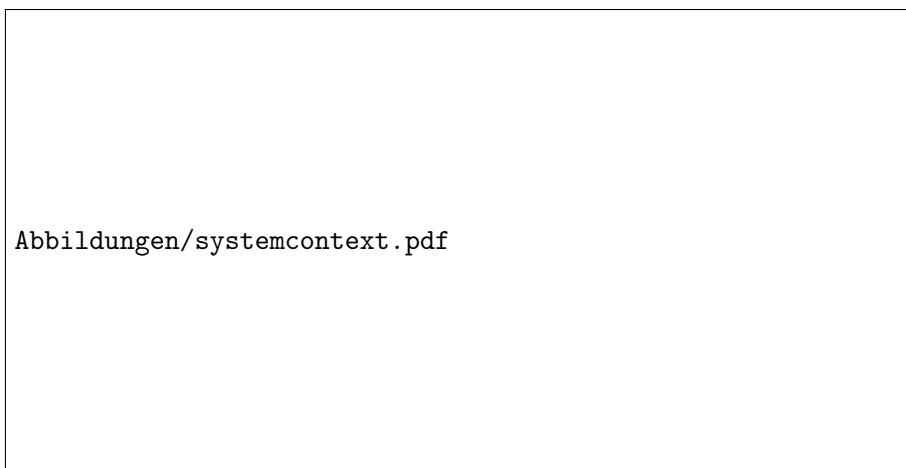


Abbildung 4.1.: Systemkontext

Ausgehend von dem Fazit des vorherigen Kapitels kann das System feingranularer bestimmt werden, wie Abbildung 4.2 darstellt. Im Fazit des vorangegangenen Kapitels wurde beschrieben, dass sowohl für TCsite als auch für Shopware Plugins entwickelt wer-



den müssen. Das TCsite-Plugin stellt eine Schnittstelle als Webservice bereit, welche vom Shopware-Plugin genutzt wird. Darum besteht eine Verbindung zwischen beiden Plugins.

In Kapitel 2.3 wurde besprochen, dass Webservices nicht direkt von menschlichen Anwendern genutzt werden. Darum steht keiner der Stakeholder in Verbindung zum TCsite-Plugin. Hingegen werden Shopkunden im Frontend und der Administrator im Backend mit dem Shop-Plugin interagieren.

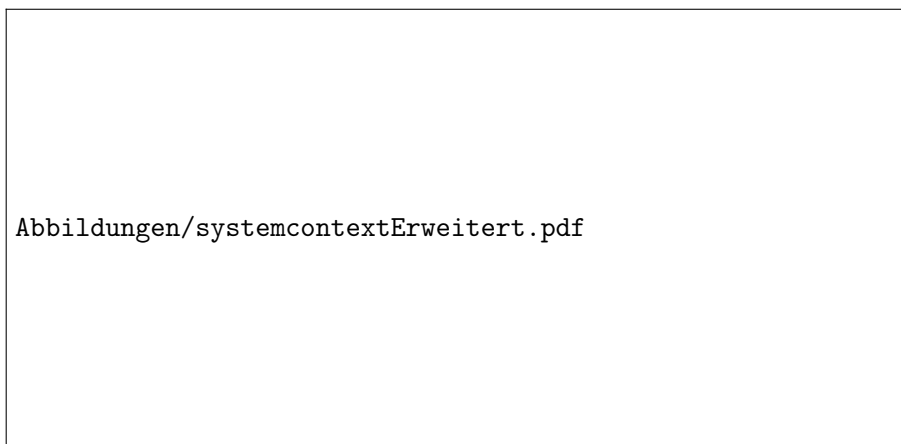


Abbildung 4.2.: Systemkontext

Basierend auf den Systemkontext werden nun Anforderungen spezifiziert, die sich auf das System und die Aspekte des Systemkontexts beziehen. Es wird in funktionale und nicht-funktionale Anforderungen unterschieden.

#### 4.0.1. Funktionale Anforderungen

Funktionale Anforderungen beschreiben das „was“, d.h. welche Funktionalitäten und Dienste das System erfüllen soll. Es werden zunächst die funktionalen Anforderungen des Shopware-Plugins erwähnt. Aus diesen ergibt sich, was das TCsite-Plugin, welches einen Webservice repräsentiert, liefern muss.

##### 4.0.1.1. Shopware-Plugin

Die Analyse des Einkaufsvorgangs hat ergeben, dass die Konfiguration eines Artikels auf dessen Detailseite stattfindet. Daraus ergibt sich die funktionale Anforderung SW.F01:

**SW.F01: Das Shop-Plugin muss auf der Detailseite eine Konfigurationsoberfläche bieten.**

Die Funktionalität der Konfigurationsoberfläche kann näher beschrieben werden. Aus der Analyse des Konfigurationsprozesses in TCsite (siehe Kapitel 3.1.2.1) ergibt sich die Anforderung SW.F02:

**SW.F02: Die Konfigurationsoberfläche muss einen Konfigurationsprozess äquivalent zu TCsite ermöglichen.**

Das beinhaltet: Aktualisierung der Oberfläche nach jeder Wahl des Anwenders; Darstellung von Steps, Toplevel-Groups, optionalen Gruppen, allen Feldtypen, Hervorhebung der verhandelbar/nichtverhandelbar gewählten Optionen; Navigation durch die Execution-Struktur (Steps in vordefinierter Reihenfolge, Toplevel-Groups); Wählen von Optionen, Eingabe von Werten; Wechsel des Optionszustands zwischen verhandelbar/nichtverhandelbar; Konfliktbehebung.

Aus der Analyse des Einkaufsvorgangs in Kapitel 3.2.3 geht hervor, dass nach der Konfigurationen die resultierende Variante weitere Stationen durchläuft (Warenkorb, Checkout, eventuell Umkonfiguration, etc...). Daraus resultiert Anforderung SW.03, welche die Basis der folgenden Anforderungen darstellt:

**SW.F03: Die aus dem Konfigurationsprozess resultierende Variante muss sich in den Einkaufsvorgang integrieren.**

Damit eine Variante gekauft werden kann, benötigt sie einen Preis. Dieser wird auf der Detailseite gezeigt. Er muss entsprechend des Konfigurationsprozesses laufend aktualisiert werden. Dementsprechend lautet Anforderung SW.F04:

**SW.F04: Die Detailseite muss den aktuellen Preis der Variante anzeigen.**

Von der Detailseite aus kommt die Variante in den Warenkorb. Daraus ergibt sich Anforderung SW.F05:

**SW.F05: Die Detailseite muss das Ablegen der Variante in den Warenkorb ermöglichen.**

In Shopware wird dabei die auf der Detailseite eingestellte Anzahl berücksichtigt. Gleiches gilt implizit für Anforderung SW.F05.

In Kapitel 3.1.2.1 wurde bei der Analyse des Konfigurationsprozesses in TCsite festgestellt, dass diese erst im letzten Step abgeschlossen werden kann. Äquivalent resultiert für das Shopszenario die Anforderung SW.F06, die eine detailliertere Beschreibung der Anforderung SW.F05 darstellt:

**SW.F06: Die Variante muss erst im letzten Step in den Warenkorb legbar sein.**

Varianten im Warenkorb sind Warenkorbpsitionen. Bei unterschiedlichen Varianten im Warenkorb muss der Kunde diese unterscheiden können. Daraus resultiert Anforderung SW.F07:

**SW.F07: Warenkorbpsitionen müssen eine Beschreibung bieten.**

Zur Beschreibung gehört der Preis, aber zum Beispiel auch eine Zusammenfassung der gewählten Optionen.

Der in Kapitel 3.2.3 vorgestellte Einkaufsvorgang beinhaltet die Umkonfiguration von Warenkorbposition. Daraus resultiert Anforderung SW.F08:

**SW.F08: Warenkorbpositionen müssen umkonfigurierbar sein.**

Umkonfiguration wird in der Arbeit so verstanden, dass sich ändert, welche Variante eine Warenkorbposition repräsentiert. Daraus resultiert implizit, dass eine Umkonfiguration nicht zur Erstellung einer neuen, sondern zur Aktualisierung der entsprechenden Warenkorbposition führt.

Die Umkonfiguration findet auf der Detailseite statt. Dementsprechend lautet Anforderung SW.F09:

**SW.F09: Die Detailseite einer Warenkorbposition muss aufrufbar sein.**

Auf der Detailseite soll die Konfiguration jedoch nicht neu beginnen, sondern dem letzten Konfigurationszustand entsprechen. Es ergibt sich Anforderung SW.F10:

**SW.F10: Die Detailseite muss den Konfigurationszustand einer Warenkorbposition abbilden.**

Das beinhaltet den Preis und die Konfigurationsoberfläche.

Im Einkaufsvorgang aus Kapitel 3.2.3 registriert der Warenkorb die Umkonfiguration erst beim erneuten Ablegen in den Warenkorb. Dementsprechend lautet Anforderung SW.F11:

**SW.F11: Auf der Artikeldetailseite muss die Umkonfiguration einer Warenkorbposition durch einen entsprechenden Button vom Kunden bestätigt werden.**

Durch das Umkonfigurieren kann es passieren, dass zwei Warenkorbpositionen die gleiche Variante repräsentieren. Die Varianten sollen jedoch weiterhin einzeln umkonfigurierbar sein. Daraus resultiert Anforderung SW.F13:

**SW.F13: Im Warenkorb muss jede Variante eine eigene Position darstellen.**

Warenkorbartikel sind auch löscher. Daraus folgt Anforderung SW.F13:

**SW.F13: Im Warenkorb müssen Varianten löscher sein.**

Die nächste Phase des Lebenszyklus ist die Bestellung. Dementsprechend lautet Anforderung SW.F14:

**SW.F14: In der Bestellung müssen Warenkorbpositionen übernommen werden.**

So wie die Varianten im Warenkorb beschrieben wurden, muss gleiches auch in der Bestellbestätigung stattfinden. Es resultiert Anforderung SW.F15:

**SW.F15: In der Bestellbestätigung für den Kunden müssen die Beschreibungen der Warenkorbpositionen übernommen werden.**

Bis jetzt wurde unterschlagen, dass der Lebenszyklus einer Variante zwar mit der Konfiguration beginnt, auf Übersichtsseiten wie z.B. dem Artikellisting trotzdem Preise verzeichnet sind. Der Preis steht vor der Konfiguration aber noch nicht fest. Dementsprechend lautet SW.F16:

**SW.F16: Im Artikellisting dürfen konfigurierbare Artikel keinen Preis auszeichnen.**

Ein eShop bietet nicht nur konfigurierbare Artikel an. Dementsprechend muss das Plugin einen Mechanismus anbieten, über den im Backend eingestellt wird, welche Artikel via Plugin konfigurierbar sind und welche nicht. Daraus ergibt sich Anforderung SW.F17:

**SW.F17: Das Shop-Plugin muss dem Administrator im Backend eine Angabe darüber ermöglichen, ob ein Artikel per Plugin konfigurierbar ist.**

Wenn ein Artikel per Plugin konfigurierbar ist, stellt sich die Frage, welchem Konfigurationsmodell die Konfiguration es entsprechen soll. In TCsite sind Konfigurationsmodelle Produkten zugeordnet. Daraus ergibt sich Anforderung SW.F18:

**SW.F18: Das Shop-Plugin muss dem Administrator im Backend eine Angabe darüber ermöglichen, welchem TCsite-Produkt ein Artikel entspricht.**

So wird die Zuordnung händisch vom Shopware-Administrator hergestellt. Er muss also auch Kenntnis von den in TCsite verfügbaren Produkten haben.

Abschließend muss das Plugin noch die Adresse des Webservices kennen, welches durch das TCsite-Plugins implementiert wird. Es folgt Anforderung SW.F19:

**SW.F19: Das Shop-Plugin muss dem Administrator im Backend eine Angabe darüber ermöglichen, unter welcher URI das TCsite-Plugin erreichbar ist.**

#### 4.0.1.2. TCsite-Plugin

Die Anforderungen des TCsite-Plugins ergeben sich aus den Anforderungen des Shopware-Plugins.

SW.F02 fordert Daten, auf deren Grundlage die Konfigurationsoberfläche gerendert werden kann. Dementsprechend lautet Anforderung TC.F01:

**TC.F01: Das TCsite-Plugin muss die Informationen zum Aufbau der Konfigurationsoberfläche bereitstellen.**

Aus SW.F10 ergibt sich, dass das Shopware-Plugin auch nach Abschluss der Konfiguration im Falle einer Umkonfiguration die Konfigurationsoberfläche erneut rendern muss. Ob und wann eine Umkonfiguration stattfindet, ist nicht klar. Daraus ergibt sich Anforderung TC.F02:

**TC.F02: Die Informationen zum Aufbau der Konfigurationsoberfläche müssen persistiert werden.**

Weiterhin muss das Shopware-Plugin wissen, wie es diese persistenten Informationen zu einem späteren Zeitpunkt wieder abrufen kann. Daraus ergibt sich Anforderung TC.03:

**TC.F03: Das TCsite-Plugin muss Informationen zum späteren Aufbau der Konfigurationsoberfläche bereitstellen.**

Im Fazit aus Kapitel 3.1.3 wurde festgelegt, dass über die Oberfläche die gewählten Optionen des Shopkunden erfasst werden, der technische Konfigurationsprozess aber durch das TCsite-Plugin realisiert wird. Daraus ergibt sich Anforderung TC.04:

**TC.F04: Das TCsite-Plugin führt den technischen Konfigurationsprozess durch.■**

Neben den Daten für die Oberfläche fordert SW.F04 Preisinformation. Dementsprechend lautet Anforderung TC.F04:

**TC.F04: Das TCsite-Plugin muss Informationen über den Preis einer Variante bereitstellen.**

Abschließend fordert SW.F07 nach einer Beschreibung von Warenkorbpositionen. Es resultiert Anforderung TC.F05:

**TC.F05: Das TCsite-Plugin muss die Beschreibung einer Variante bereitstellen.**

#### 4.0.2. Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen beschreiben das „wie“, d.h. welche Eigenschaften das System besitzen soll.

##### 4.0.2.1. Shopware-Plugin

Wie bereits erwähnt, werden die Funktionalitäten im Rahmen eines Plugins implementiert. Die Anforderung SW.NF01 lautet dementsprechend:

**SW.NF01: Die Integration des Tacton-Produktkonfigurators muss über ein Shop-Plugin umgesetzt werden.**

Weiterhin kann aufgrund der technischen Analyse eine Aussage darüber getroffen werden, mit welchen Programmiersprachen das Shop-Plugin zu entwickeln ist. In der View kommt Smarty-Syntax zum Einsatz, wobei Javascript und CSS eingebunden werden kann. Die logischen Erweiterungen werden in PHP verfasst. Anforderung SW.NF02 lautet demzufolge:

**SW.NF02: Das Shop-Plugin muss unter der Verwendung von PHP, Smarty-Syntax, Javascript, und CSS erstellt werden.**

#### 4.0.2.2. TCsite-Plugin

Die nichtfunktionalen Anforderungen des TCsite-Plugins fallen äquivalent aus. TC.NF01 lautet:

**TC.NF01: Die Konfigurationsschnittstelle muss über ein TCsite-Plugin umgesetzt werden.**

In Kapitel 3.1.2 wurde erwähnt, dass die Architektur in Java implementiert ist. Daraus resultiert für die Anforderung TC.NF02:

**TC.NF02: Das TCsite-Plugin muss unter der Verwendung von Java erstellt werden.**

Aus dem Erweiterungskonzept von TCsite (siehe Kapitel 3.1.2.3) kann schon jetzt das zu verwendende Webservice-Paradigma abgeleitet werden. Im Fazit des vorangegangenen Kapitels wurde festgelegt, dass Web-Controller genutzt werden. Entsprechend des Kurfazits in Kapitel 2.3 steht damit das Werkzeug zur Implementierung eines REST-Websservices zur Verfügung. Es resultiert die Anforderung:

**TC.NF03: Die Konfigurationsschnittstelle wird als Webservice entsprechend des REST-Architekturstils implementiert.**


## 4.1. Konzeption

Aus der Anforderungsanalyse ist hervorgegangen, dass zwei Plugins konzipiert werden müssen. Es wird erst das Shopware-Plugin und danach das TCsite-Plugin behandelt. Grund dafür ist, dass das TCsite-Plugin einen Webservice darstellt, den das Shopware-Plugin als Client nutzt. Indem der Client zuerst konzipiert wird, ergeben sich daraus die Ressourcen und Prozesse, die vom Service bereitgestellt werden müssen.

### 4.1.1. Shopware-Plugin

#### Vorbetrachtung

Vor der Konzeption des Shopware-Plugins wird betrachtet, wie sich Artikel, Varianten, Warenkorbpositionen etc. zueinander im Datenbankschema verhalten. Daraus wird abgeleitet, wie die Ergebnisse einer Konfiguration – die Varianten – in den Einkaufsvorgang integriert werden. Es wird geklärt, ob neue Klassen bzw. Tabellen erzeugt oder bestehende Strukturen verwendet werden können.



Abbildungen/shopwareArtikelRelationenModell.pdf

Abbildung 4.3.: Ausschnitt des Relationenmodells in Shopware

Abbildung 4.3 zeigt einen Ausschnitt des Shopware-Datenbankschemas als Relationenmodell. Eine vollständige Übersicht ist Quelle shopware AG (2015d) zu entnehmen. Auf der Abbildung sind die im Analysekapitel ?? verwendeten Begriffe rot zugeordnet. Dem Relationenmodell ist zu entnehmen, dass Artikel (*s\_articles*) und deren Varianten (*s\_articles\_details*) in unterschiedlichen Tabellen gespeichert werden. Artikel und Varianten stehen in einer

1-zu-n Beziehung. Dabei kennen Varianten ihren jeweiligen Basisartikel, aber nicht umgekehrt. Es ist anzumerken, dass auch zu einem nichtkonfigurierbaren Artikel zumindest eine Zeile in *s\_articles\_details* angelegt wird<sup>1</sup>. Der Artikel steht dann mit einer Variante in einer 1-zu-1 Beziehung. Diese Variante ist damit automatisch immer die Basisvariante<sup>2</sup>. Daraus lässt sich ableiten, dass in *s\_articles* Informationen stehen, die für alle Varianten gelten (z.B. die allgemeine Produktbeschreibung *description*) während *s\_articles\_details* detailliertere Informationen enthält (z.B. der jeweilige Lagerbestand *instock*). Es ist hervorzuheben, dass *s\_articles\_details* zwei Primärschlüssel besitzt – *articledetailsID* und *ordernumber*. Die *articledetailsID* wird vom System vergeben, während die *ordernumber* vom Administrator bestimmt werden kann. Somit ist die *ordernumber* ein menschenlesbarer Identifikator, der vom Vertrieb dem Produktkatalog des Unternehmens zugeordnet werden kann.

Die Verbindung zwischen einer Variante und den Optionen (*s\_article\_configurator\_options*), aus denen sie generiert wurde, wird über die Tabelle *s\_article\_configurator\_option\_relations* hergestellt. Weder sind Optionen inhärenter Bestandteil einer Variante, noch besitzt *s\_articles\_details* Fremdschlüsselverweise auf *s\_article\_configurator\_options*. Damit ist eine Variante nicht existentiell auf Optionen angewiesen. Daraus wird geschlussfolgert, dass während des Einkaufsprozesses Varianten dynamisch, d.h. bei Bedarf, generiert werden können.

Im Gegensatz dazu hat eine Warenkorbposition (*s\_order\_basket*) einen obligatorischen Fremdschlüsselverweis auf eine Variante – und ist damit von deren Existenz abhängig. Ein Warenkorbposition kann zeitlich nicht vor der zugehörigen Variante angelegt werden. Es wird geschlussfolgert, dass eine Variante noch nicht zum Beginn des Einkaufsvorgangs, aber spätestens beim Ablegen in den Warenkorb erstellt werden muss. Damit steht der zeitliche Spielraum für die dynamische Variantengenerierung fest.

#### 4.1.1.1. Einkaufsvorgang

Abbildung 4.4 stellt den neuen Einkaufsvorgang dar. Er entsteht aus dem Einkaufsvorgang in Abbildung 3.17, wenn folgende Sachverhalte berücksichtigt werden:

1. Es wird mit dem TCsite-Plugin kommuniziert (dargestellt als Wolken).
2. Varianten werden dynamisch angelegt.
3. Wird eine Warenkorbposition umkonfiguriert, entsteht keine neue Warenkorbposition. Stattdessen wird die Warenkorbposition aktualisiert.

<sup>1</sup> Solche Kardinalitäten können in einem Relationenmodell nicht ausgedrückt werden.

<sup>2</sup> Der Begriff Basisvariante wurde in Kapitel 3.2.3 definiert



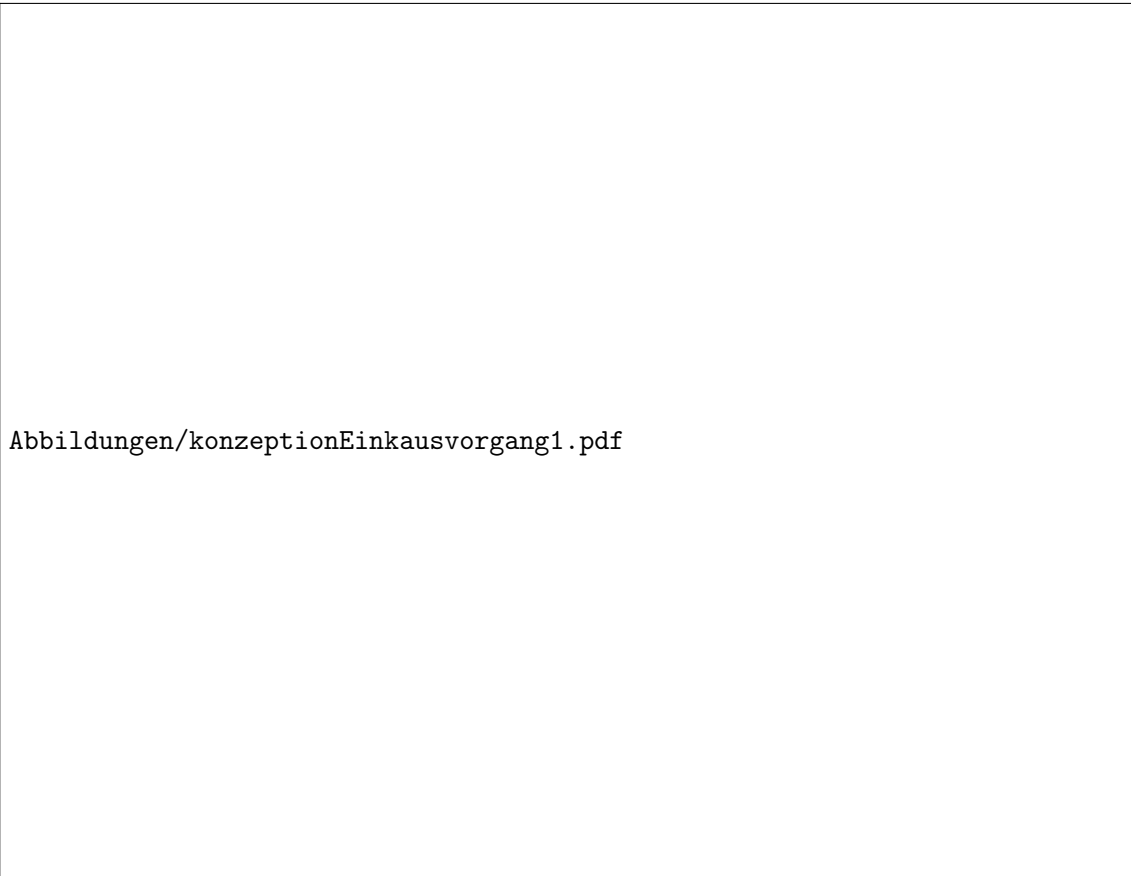
Abbildungen/konzeptionEinkausvorgang1.pdf

Abbildung 4.4.: Aktivitätsdiagramm des Einkaufsvorgangs von Artikeln, die per Plugin konfiguriert werden

Das Aktivitätsdiagramm (Abbildung 4.4) bildet den „roten Faden“ für die folgende Konzeption. Während der chronologischen Erläuterung des Einkaufsvorgangs werden die Daten und Prozesse erläutert, die vom Service bereit gestellt werden müssen. Gleichzeitig wird besprochen, welche Modifikationen der Shopware-Funktionalität vorgenommen werden müssen und welche der Erweiterungskonzepte aus Kapitel 3.2.2 dafür jeweils in Frage kommen.

### Artikellisting

Der Einkaufsvorgang beginnt mit dem Auswählen des gewünschten Artikels. Das kann über verschiedene Wege passieren, zum Beispiel über das Artikellisting. Anhang A.9 zeigt, dass dabei standardmäßig Preise angezeigt werden. Gemäß Anforderung *SW.F16* dürfen konfigurierbare Artikel aber keinen Preis ausweisen. Es muss also ein Hinweis darüber hinterlegt werden, welche Artikel per Plugin konfigurierbar sind. Das entspricht Anforderung *SW.F16*, gemäß derer konfigurierbare Artikel im Backend als solche markierbar sein müssen. Gleichzeitig fordert *SW.F17* die Angabe, welchem Produkt in TCsite ein Shopware-Artikel entspricht. Im Folgenden wird diese Zuordnung als *korrespondierendes TCsite-Produkt* bezeichnet. Es wird festgelegt: besitzt ein Artikel diese Zuordnung, ist es via Plugin konfigurierbar.

Zum Hinterlegen der Zuordnung wird eine Datenbank-Erweiterung genutzt. Konkret wird das in Kapitel 3.2.2 angesprochene System der Attributtabelle verwendet. Plugins können Attributtabelle (Tabellenbezeichnung endet auf *\*\_attributes*) leicht Spalten hinzufügen können. Abbildung 4.3 zeigt *s\_articles\_attributes*, die Attributtabelle von Artikeln und ihren Varianten. Shopware legt zu jeder Variante in *s\_articles\_details* automatisch eine zugehörige Zeile in der Attributtabelle an. Es wird die Entwurfsentscheidung getroffen, der Tabelle eine Spalte *correspondingTcsiteProduct* hinzuzufügen. Um diese Spalte befüllen zu können, wird ein entsprechendes Textfeld per *Ext JS*-Erweiterung dem Artikelmenü (dargestellt in Anhang A.5) hinzugefügt. In Kapitel 3.2.2 wurde außerdem erwähnt, dass in Smarty-Templates logische Einheiten existieren, die als Blöcke bezeichnet werden. Der Block namens *frontend\_listing\_box\_article\_price\_default* beinhaltet den Preis eines Artikels im Artikellisting. Über eine Template-Erweiterung muss dieser Block ersetzt werden. Er soll künftig überprüfen, ob ein Eintrag über ein korrespondierendes TCsite-Produkt vorliegt und die Ausgabe entsprechend anpassen. Statt des Preises wird dann die Nachricht „Preis entsprechend Konfiguration“ angezeigt. Somit werden die Anforderungen *SW.F16*, *SW.F17* und *SW.F18* erfüllt.

### Detailseite

Der Anwender befindet sich nur auf der Detailseite eines Artikels, der via Plugin konfigurierbar ist. Es wird die Basisvariante geladen und angezeigt. Der Artikel wurde im Backend nie als *Varianten-Artikel* markiert. Darum wird nicht der Standard-Konfigurationsmechanismus von Shopware getriggert. Demzufolge wird auch kein Konfigurationsmenü wie in Anhang A.7 angezeigt. Aus Shopware-Sicht handelt es sich um einen normalen, nichtkonfigurierbaren Artikel und lädt die Basisvariante.

Stattdessen startet nun der Konfigurationsprozess über das Shopware-Plugin. Dazu müssen Daten vom TCsite-Plugin geladen werden. Diese Daten werden im Rest der Konzeption unter dem Begriff **configuration** zusammengefasst. Aus Shopware-Sicht ist das ein Aggregat, welches alle Informationen zum Rendern der Konfigurationsoberfläche und zur Integration der entstehenden Variante in den Einkaufsvorgang enthält. Ob und wie dieses Aggregat als einzelne Ressourcen modelliert wird, ist Verantwortungsbereich der späteren Webservice-Konzeption. Fest steht jetzt bereits, dass *configuration* in Form von JSON repräsentiert wird. TCsite hat für Webcontroller bereits Komponenten zum serialisieren und deserialisieren von JSON-Objekten eingebaut (Tacton Systems AB, 2014b).



Abbildung 4.5.: Erstellen & Laden einer neuen *configuration*-Ressource

Abbildung 4.5 zeigt, wie das Erstellen & Laden der *configuration* abläuft. Das Shopware-Plugin teilt dem TCsite-Plugin mit, für welches Produkt (z.B. Notebook) eine Konfiguration starten soll. Das TCsite-Plugin erstellt und speichert ein *Configurable*-Objekt, welches in Verbindung mit dem entsprechenden Konfigurationsmodell und dem Konfigurationszustand steht. Der Konfigurationszustand ist leer, da zu diesem Zeitpunkt noch keine Optionen gewählt wurden. Es wird ein *KE-Input* definiert, der mit einem *KE-Output* beantwortet wird. Aus diesem werden die relevanten Informationen extrahiert, als JSON verpackt und dem Shopware-Plugin geschickt.

Der neuerstellten *configuration*-Ressource muss außerdem eine ID zugewiesen werden, die dem Shopware-Plugin in der Antwort mitgeteilt wird. Diese ID wird von nun an als *configurationID* bezeichnet. Die Generierung der ID wird während der Konzeption des TCsite-Plugins erklärt. Damit Shopwareseitig die ID für spätere Aufrufe gespeichert werden kann, muss die Datenbanktabelle *s\_articles\_details* um die Spalte *configurationID* ergänzt werden. Der Eintrag der ID folgt jedoch später.

Die Beschreibung des Aktivitätsdiagramms (Abbildung 4.4) wird fortgeführt. Oben wurde der Punkt „neue *configuration* erstellen & laden“ behandelt. Die Erläuterung von „bestehende *configuration* kopieren & laden“ geschieht zu einem späteren Zeitpunkt. Es folgt das initiale Rendern der Konfigurationsoberfläche und die Aktualisierung der Preisanzeige mittels der Daten aus der *configuration*-Ressource. Hierfür werden Template-Erweiterungen

genutzt. Der Smarty-Block *frontend\_detail\_index\_detail* zeichnet den Bereich der Detailseite aus, der das Artikelbild enthält (siehe Anhang A.7). Unterhalb wird die Konfigurationsoberfläche angehängen. Außerdem wird die Preisanzeige aktualisiert. Für diese Vorgänge wird DOM-Manipulation und clientseitige Logik benötigt. Das bedeutet, es müssen Javascript-Dateien eingebunden werden. Hierfür steht der Smarty-Block *frontend\_index\_header\_javascript* zur Verfügung.

### Konfigurationsprozess

Es folgt der Konfigurationsprozess. Der Anwender wählt über die Konfigurationsoberfläche Optionen, tätigt Eingaben und wechselt Steps. Abbildung 4.6 stellt die Kommunikation zwischen dem Shopware-Plugin und dem TCsite-Plugin nach jeder Nutzerinteraktion dar.



Abbildung 4.6.: Ändern & Laden einer *configuration*-Ressource

Das Shopware-Plugin übermittelt die gewählte Option an das TCsite-Plugin. Außerdem muss die *configurationID* übertragen werden, welche das Shopware-Plugin beim initialen Anfordern der *configuration* erhalten hat. So kann das TCsite-Plugin zuordnen, auf welches *Configurable* sich die Option bezieht und einen *KE-Input* definieren. Der *KE-Output* enthält den neuen Konfigurationszustand. Das *Configurable* wird entsprechend aktualisiert und gespeichert. Das Shopware-Plugin nimmt die *configuration* in der Serverantwort entgegen und aktualisiert die Konfigurationsoberfläche und Preisanzeige. Jede Interaktion des Anwenders mit der Oberfläche löst den eben beschriebenen Prozess aus. Dieser Kreislauf wiederholt sich solange, bis der Anwender im letzten Step angekommen und mit allen Op-

tionen zufrieden ist. Gemäß Anforderung SW.F06 wird der „In den Warenkorb“-Button erst jetzt aktiviert, wodurch die Konfiguration abgeschlossen werden kann.

Insgesamt werden so die Anforderungen SW.F01 (Bereitstellung einer Konfigurationsoberfläche) und SW.F04 (Anzeige des aktuellen Preises) erfüllt. SW.F02 fordert außerdem, dass der Konfigurationsprozess äquivalent zu TCsite verläuft. Die genaue Nachempfindung der Abläufe ist ein Implementierungsdetail, welches in Kapitel 4.2 angerissen wird.

### Dynamische Variantengenerierung

Der Klick auf den „In den Warenkorb“-Button führt bei einer neuen Konfiguration zum Anlegen einer Warenkorposition (Anforderung SW.F05). Die Unterscheidung dieses Falles von einer Umkonfiguration wird später erläutert. Standardmäßig wird durch den Button die *ajaxAddArticleCart*-Action des Checkout-Controllers ausgelöst<sup>3</sup>. In der Action wird erst die Warenkorposition angelegt und dann der Warenkorb angezeigt. In der Vorbesprechung dieses Kapitels wurde festgestellt, dass bei der Erstellung einer Warenkorposition die zugehörige Variante bereits existieren muss. Der Konfigurationsprozess fand aber statt, während auf der Detailseite noch die Basisvariante geladen war. Die aus dem Konfigurationsprozess resultierende Variante muss also erst noch generiert werden. Die Analyse der Shopware-Erweiterbarkeit in Kapitel 3.2.2 hat ergeben, dass Controller-Actions überschreibbar sind. *ajaxAddArticleCart* wird also durch eine vom Shopware-Plugin bereitgestellte Funktion ersetzt, um die dynamische Variantengenerierung zu implementieren.

Abbildung 4.7 visualisiert die Vorgänge in der *ajaxAddArticleCart* des Plugins. Zunächst wird eine neue Variante angelegt, die sich auf den gleichen Artikel wie die Basisvariante bezieht. Das bedeutet, es wird eine neue Zeile in der Tabelle *s\_articles\_details* mit einem Fremdschlüsselverweis auf *s\_articles* erzeugt (siehe Abbildung 4.3). Außerdem wird die entsprechende *configurationID* in der Tabelle *s\_articles\_details* hinterlegt. So ist die Variante mit der *configuration*-Ressource verknüpft, die während des Konfigurationsprozesses verwendet wurde. Der Preis der Variante wird in der Tabelle *s\_articles\_details* angelegt. Jetzt kann die Warenkorposition in der entsprechenden Anzahl eingetragen werden. Die *ordernumber* (d.h. der menschenlesbare Primärschlüssel) wird automatisch generiert. Zum Erfüllen der Anforderung SW.F07 muss Name der Warenkorposition zu einer Kurzbeschreibung<sup>4</sup> der Variante geändert werden (z.B. „Notebook OSX Yosemite 4GB RAM“). Zusammengefasst muss die überschriebene *ajaxAddArticleCart*-Action im Gegensatz zur Standardmethode drei zusätzliche Eingabeparameter verarbeiten – die *configurationID*, den Preis und die Kurzbeschreibung.

<sup>3</sup>Die Controller-Terminologie wurde in Kapitel 3.2.1 vorgestellt

<sup>4</sup>Die Kurzbeschreibung stammt ebenfalls aus der *configuration*-Ressource.

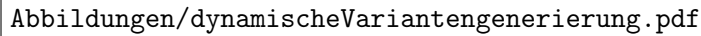
The image is a rectangular box containing the text 'Abbildungen/dynamischeVariantengenerierung.pdf'. It appears to be a placeholder for a diagram or figure related to dynamic variant generation.

Abbildung 4.7.: Dynamische Variantengenerierung

### Warenkorb

Die aktuelle Station des Einkaufsvorgangs ist der Warenkorb (siehe Abbildung 4.4). Möchte der Anwender weitere Artikel konfigurieren, schließt er den Warenkorb und beginnt den Einkaufsvorgang von vorne. Damit wiederholt sich der bisher beschriebene Prozess. Ist der Anwender mit einer Warenkorbbposition unzufrieden, kann er diese Löschen (SW.F13) oder Umkonfigurieren (SW.F08).

Im Gegensatz zum ursprünglichen Einkaufsvorgang (siehe Abbildung 3.17) hat das Löschen einer Warenkorbbposition weitere Löschvorgänge zur Folge. Eine Variante wurde nur erstellt, um als Bezugspunkt einer Warenkorbbposition zu dienen. Das Löschen einer Position macht auch die zugehörige Variante obsolet. Dementsprechend wird diese aus der Datenbank entfernt. Durch die in Abbildung dargestellten 4.3 Fremdschlüsselverweise werden auch die entsprechenden Zeilen aus *s\_articles\_prices* und *s\_articles\_details* gelöscht. Gleichzeitig wird die zugehörige *configuration*-Ressource funktionslos. Dementsprechend muss ein Löschbefehl an das TCsite-Plugin abgesetzt werden.

Der Löschvorgang einer Warenkorbbposition wird von der Methode *sDeleteArticle* des Moduls *sBasket* realisiert (siehe Abbildung 3.8). Weder handelt es sich um eine Controller-Action, noch enthält die Funktion Notify-Events. Entsprechend der Erläuterung der logischen Erweiterungen in Kapitel 3.2.2 muss also auf das Hooksystem zurückgegriffen werden. Eine Plugin-Funktion wird der *sDeleteArticle* nachgeschaltet, worin das Löschen der entsprechenden Variante und *configuration* nachgeholt wird. SW.F13 wird so erfüllt.

### Umkonfiguration

Durch einen Klick auf eine Warenkorbbposition gelangt der Anwender zur Detailansicht der Variante. Dies entspricht soweit dem Shopware-Standard und erfordert keine zusätzliche

Logik. Somit wird Anforderung SW.F09 erfüllt. Zusätzlich muss die Konfigurationsoberfläche wieder aufgebaut und der Preis abgeholt werden, um SW.F10 zu erfüllen. Beim Anlegen der Variante wurde eine *configurationID* hinterlegt. Somit kann die entsprechende *configuration* geladen werden. Dies entspricht der Aktivität „bestehende *configuration* kopieren und laden“ aus Abbildung 4.4.

Abbildung 4.8 stellt den dabei ablaufenden Prozess dar. Er entspricht insofern der Aktivität „neue *configuration* erstellen & laden“ (siehe Abbildung 4.5), als dass der KE-Input ohne eine gewählte Option verarbeitet wird. Dementsprechend ist der Konfigurationszustand vor und nach dem Aufruf der Konfigurationsengine der selbe. Jedoch geschieht der Umkonfigurationsprozess auf einer Arbeitskopie, wobei der Konfigurationszustand der ursprünglichen *configuration* übernommen wird. Grund dafür ist Anforderung SW.F11. Gemäß dieser muss die Umkonfiguration erst abschließend vom Anwender bestätigt werden. Es ist also möglich, dass der Anwender zunächst andere Optionen wählt, die Änderungen aber verwerfen möchte (z.B. indem er die Detailseite ohne Bestätigung der Umkonfiguration verlässt). Ohne das Anfertigen einer Arbeitskopie würde im Falle des Verwerfens eine Diskrepanz zwischen einer Variante und der zugeordneten *configuration* entstehen.



Abbildung 4.8.: Kopieren & Laden einer bestehenden *configuration*

Die Beschriftung des „In den Warenkorb“-Buttons wird auf „Neue Konfiguration speichern“ geändert. So wird darauf hingewiesen, dass das Verhalten des Buttons sich ändert. Die entsprechende Warenkorposition wird nicht inkrementiert, sondern aktualisiert. Die

Aktualisierung betrifft den Preis (Spalte: *price*; Tabelle *s\_articles\_prices*), die Beschreibung (Spalte: *articlename*; Tabelle *s\_order\_basket*) und die Anzahl (Spalte: *quantity*; Tabelle *s\_order\_basket*). Außerdem muss die *configurationID* auf die der Arbeitskopie aktualisiert werden, da diese nun den aktuellen Konfigurationszustand enthält.

Zwar wurde die Beschriftung des „In den Warenkorb“-Buttons geändert, er triggert aber immer noch die gleiche Controllerfunktion. Darum kann zur Implementierung des neuen Verhalten die bereits überschriebene *ajaxAddArticleCart*-Action genutzt werden. In dieser muss unterschieden werden, ob es sich um eine neue Konfiguration oder eine Umkonfiguration handelt. Letzteres ist dann der Fall, wenn für die Variante bereits eine *configurationID* hinterlegt ist.

Der vorangegangenen Erläuterung ist zu entnehmen, dass zwei Varianten nie auf inhaltliche Gleichheit überprüft werden. Auch wenn der Anwender zwei mal hintereinander genau die gleiche Konfiguration vornimmt, werden separate Warenkorbbpositionen angelegt, da jede im Zusammenhang mit einer eigenen *configuration* steht. Somit wird Anforderung SW.F12 erfüllt.

### Bestellung

Die erstellten Varianten gehen als normale Artikel in den Bestellprozess ein (SW.F14). Durch die Entscheidung, die Kurzbeschreibung einer Variante zum Namen der Warenkorbbposition zu machen, erscheint diese auch in der Bestellbestätigung (SW.F15).

Im Eingang dieses Kapitels wurde erwähnt, dass über die *ordnernumber* einer Variante (Tabelle *s\_articles\_details*) die bestellte Warenkorbbposition vom Vertrieb dem Produktkatalog zugeordnet werden kann. Die *ordnernumber* wurde im eben dargestellten Ablauf jedoch automatisiert generiert. Allein durch die Bestellbestätigung ist also noch nicht klar, welche Varianten hier eigentlich beauftragt wurden. Zur Lösung des Problems gibt es zwei Ansätze:

1. Die Kurzbeschreibung der Variante ist aussagekräftig genug. Das ist allerdings nur bei Varianten mit wenig Optionen realistisch.
2. Die *configurationID* ermöglicht die Auswertung der Daten, die bei TCsite mit der jeweiligen Variante im Zusammenhang stehen. Genauer zur dieser Vorgehensweise wird in der Konzeption des TCsite-Plugins besprochen.

#### 4.1.1.2. Zusammenfassung

Vorausgehend wurden die notwendigen Erweiterungen entlang des Einkaufsvorgangs (Abbildung 4.4) beschrieben und begründet. Implizit wird so Anforderung SW.F03 erfüllt (Integration der Varianten in Einkaufsvorgang). Im Folgenden werden die Erweiterungen nach deren Typ zusammengefasst:

1. **Daten-Erweiterungen:**



- (a) Hinzufügen von Variantenattributen  
Spalten: *correspondingTcsiteProduct* / *configurationID*, Tabelle: *s\_articles\_details*

## 2. Template-Erweiterungen:

- (a) Ersetzen des Preises im Artikellisting  
Smarty-Block: *frontend\_listing\_box\_article\_price\_default*
- (b) Einfügen der Konfigurationsoberfläche in die Detailseite  
Smarty-Block: *frontend\_detail\_index\_detail*
- (c) Einfügen von clientseitiger Logik mittels Einbindung von Javascript-Dateien in der Detailseite  
Smarty-Block: *frontend\_index\_header\_javascript*
- (d) Einfügen eines Textfeldes im Artikelmenü des Backends zur Eingabe des korrespondierenden TCsite-Produkts  
Block: *frontend\_index\_header\_javascript*

## 3. logische Erweiterungen:

- (a) Überschreiben der Controller-Action zum Hinzufügen von Warenkorbpositionen  
Action: *ajaxAddArticleCart*, Controller: *Checkout*
- (b) „Hooken“ der Methode zum Löschen von Warenkorbbartikeln  
Methode: *sDeleteArticle*, Modul: *sBasket*

Anforderung SW.F19 schlägt die Brücke zur folgenden Konzeption des TCsite-Plugins. Dieses ist ein Webservice. Webservices haben eine Adresse im Web. Die Adresse muss dem Shopware-Plugin noch bekannt gegeben werden. Shopware erlaubt dem Administrator im Backend neben der Aktivierung und Deaktivierung von Plugins auch dessen „Konfiguration“ über Formularfelder (shopware AG, 2014c). Durch die Einrichtung eines Textfeldes „Webadresse des TCsite-Plugins“ kann auch SW.F19 erfüllt werden. Alle funktionalen Anforderungen wurden somit entworfen.

### 4.1.2. TCsite-Plugin

Äquivalent zur Konzeption des Shopware-Plugins findet zunächst eine technische Vorbetrachtung statt. Während der Besprechung des neuen Einkaufsvorgangs wurde dargestellt, dass jeder Prozess im TCsite-Plugin mit der Verarbeitung eines Configurable-Objekts verbunden ist. Im Folgenden wird untersucht, wie der Zugriff auf das Objekt realisiert wird. Daraus resultieren Erkenntnisse, die Entscheidend für die Konzeption der Serviceendpunkte sind.

#### Vorbetrachtung

Abbildung 4.9 visualisiert die Verhältnisse von *Quotations* zu *QuotationItems* und *Configurables*. Eine *Quotation* besitzt eine beliebige Anzahl *QuotationItems*. Jedes *QuotationI-*

*tem* steht in einer 1-zu-1 Beziehung mit einem *Configurable*. Tacton Systems AB (2014b) rät davon ab, ein *Configurable* direkt per Konstruktor zu erstellen. Zur vollen Einsatzbereitschaft des Objekts seien weitere initialisierende Schritte notwendig. Welche, verrät die Dokumentation nicht. Stattdessen wird auf die Methode *createAndAddQuoteItem* der Serviceklasse *QuotationService* verwiesen. Diese erstellt zu einer gegebenen *Quotation* ein *QuotationItem* mit dem entsprechenden *Configurable*. Dabei werden alle initialisierenden Schritte übernommen. Die Implementierung von TCsite zwingt so zur zusätzlichen Verwaltung von *Quotations* und *QuotationItems*, auch wenn nur das *Configurable* für den Konfigurationsprozess unmittelbar relevant ist.

Abbildungen/konzeptConfigurableUML.pdf

Abbildung 4.9.: Verhältnis von *Quotations*, *QuotationItems* und *Configurables*

*Quotations* und *QuotationItems* sind nicht nur Mittel zum Zweck der Erstellung eines *Configurables*. Das Auswerten von Preisinformationen und die Zusammenstellung der Kurzbeschreibung wird ebenfalls durch API-Methoden des *QuotationService* realisiert. Die hierfür zuständigen Methoden benötigen neben dem *KE-Output* und dem *Configurable* auch die zugehörige *Quotation* und das *QuotationItem*. Zusammenfassend wird festgestellt, dass jede Verarbeitung eines *Configurable* nur im Kontext einer *Quotation* möglich ist.

Es wird die Entwurfsentscheidung getroffen, zu jedem *Configurable* genau eine *Quotation* anzulegen und zu persistieren (Anforderung TC.F02). Diese Entscheidung hat zwei Vorteile:

1. Da jede *Quotation* nur ein *Configurable* enthält, identifiziert die ID der *Quotation* gleichzeitig das *Configurable*. Damit ist klar, was die *configurationID* im Entwurf des Shopware-Plugins ist – die ID einer *Quotation*. Dementsprechend müssen im TCsite-Plugin keine zusätzlichen Schlüssel generiert und verwaltet werden.
2. In der Quotation-View von TCsite (Abbildung 3.10) existiert ein Index, der alle *Quotations* auflistet. Wird eine Variante in Shopware bestellt, kann so über die hinterlegte *configurationID* die zugehörige *Quotation* in TCsite geöffnet werden. In der *Quotation* befindet sich nur ein *QuotationItem* – und zwar die Variante, die der Shopwarekunde bestellt hat. So wird das Problem gelöst, welches im Abschnitt „Bestellung“ der Konzeption des Shopware-Plugins angesprochen wurde. Der Vertrieb kann interpretieren, welche Varianten in Shopware eigentlich bestellt wurden.

Durch Anforderung TC.NF03 steht bereits fest, dass das TCsite-Plugin einen Webservice implementiert, der den Einschränkungen des REST-Architekturstils gerecht werden

soll. Entsprechend Kapitel 2.3.2 müssen die Ressourcen mit deren Repräsentationen und Verknüpfungen (Hypermedia) entworfen werden. Weitergehend wird Entschieden, unter welcher URI und mit welchen Methoden mit den Ressourcen interagiert werden kann.

#### 4.1.2.1. Ressourcen

Im Entwurf des Shopware-Plugins wurde bereits die Ressource *configuration* modelliert. Entsprechend der Ressourcentypisierung, die in Kapitel 2.3.2 vorgestellt wurde, handelt es sich um ein Aggregat. Es enthält unterschiedliche Daten – Preisinformationen (Anforderung TC.F04), eine Kurzbeschreibung für die Warenkorbbposition (Anforderung TC.F05) und die Datengrundlage zum Rendern der Konfigurationsoberfläche (Anforderung TC.F01). Diese Informationen könnten auch in mehrere Ressourcen unterteilt werden. Da das vorliegende Konzept des Shopware-Plugins aber alle Daten zum gleichen Zeitpunkt benötigt, würde eine feingranularere Ressourcenmodellierung nur zu mehr Serveraufrufen führen.

Außerdem wurde festgestellt, dass eine *configuration* die eigene ID enthalten muss (entspricht Anforderung TC.F03). Das spielt in den Aktivitäten „neue *configuration* erstellen & laden“ sowie „bestehende *configuration* kopieren & laden“ eine Rolle (siehe Abbildung 4.4). Hier erfährt das Shopware-Plugin erst in der Serverantwort die *configurationID* für die weitere Kommunikation. In diesem Grenzfall verweist eine Ressource auf sich selbst, was dennoch eine Orientierungshilfe für den Clienten darstellt. Somit wird die REST-Einschränkung „Hypermedia“ erfüllt.

In der Konzeption des Shopware-Plugins wurde festgelegt, dass die Ressourcen als JSON repräsentiert werden. Listing 4.1 skizziert die Form der *configuration* am Notebook-Beispiel mit exemplarischen Werten. Die Daten für die Konfigurationsoberfläche (*ui-data*) sind ein komplexes Konstrukt. TCsite stellt Methoden bereit, die den *KE-Output* auslesen und die *ui-data* durch die Ineinanderschachtelung von JSON-Objekten als serialisierten Baum darstellen. Die dabei stattfindenden Prozesse sind für den Plugin-Entwickler nicht einsehbar und werden in der Arbeit nicht weiter ausgeführt. Letztendlich muss das Shopware-Plugin diesen Baum traversieren und so die Konfigurationsoberfläche aufbauen.

---

Listing 4.1: Beispiel einer configuration

#### 4.1.2.2. Serviceendpunkte

Im nächsten Schritt werden die URIs der Ressourcen entworfen und festgelegt, welche Methoden diese jeweils unterstützen.

Zunächst einmal hat das Plugin einen Basispfad im Web, der Teil jeder Ressourcen-URI ist. Wie dieser aussieht, hängt von der Adresse der TCsite-Installation ab. Im Folgenden wird mit der exemplarischen URI „<http://tcsite-uri.de/path-to-the-plugin/>“ gearbeitet. Als

Adresse für die *configuration* wird entschieden, den Namen der Ressource als Pfadende zu verwenden (z.B. „http://tcsite-uri.de/path-to-the-plugin/configuration“).

Der Identifikator einer Ressource ist ein URI-Parameter. Laut Tilkov (2011) ist es Geschmackssache, ob er als Pfad-Parameter (z.B. „http://tcsite-uri.de/path-to-the-plugin/configuration/1“ oder Query-Parameter (z.B. „http://tcsite-uri.de/path-to-the-plugin/configuration?id=1024“) entworfen wird. Da im Laufe der folgenden Erläuterung noch weitere Query-Parameter eingeführt werden, wird aus Konsistenzgründen letztere Variante gewählt.

Als nächstes müssen die HTTP-Methoden definiert werden, welche die Ressource unterstützt. Die Wolken in Abbildung 4.4 zeigen alle Operationen, die vom Service implementiert werden müssen. Im Folgenden wird unter Berücksichtigung der HTTP-Spezifikation (siehe Abbildung 2.5) besprochen, auf welche HTTP-Methode jede der Operation abgebildet werden darf. Außerdem werden bei Bedarf zusätzliche Query-Parameter definiert.

1. ***configuration* löschen:** entspricht semantisch der HTTP-Methode *DELETE*.

Beispiel: Ein *DELETE*-Request an

„http://tcsite-uri.de/path-to-the-plugin/configuration?id=1024“

löscht die *Configuration* mit der ID 1024.

2. **neue *configuration* erstellen & laden:** Das Anlegen einer Ressource entspricht semantisch der HTTP-Methode *PUT*. Allerdings fordert die Spezifikation, dass die Ressource künftig unter der gleichen Adresse erreichbar ist, an der sie via *PUT*-Request angelegt wurde. Zum Erstellungszeitpunkt kennt der Client die URI der entstehenden *configuration* aber noch nicht. Diese wird ihm erst in der Antwort über die *configurationID* mitgeteilt. Die Operation entspricht also eher der Methode *POST*, d.h. das Anlegen einer Ressource unter einer vom Service bestimmten URI. Da im Folgenden noch weitere Operationen *POST* verwenden, wird ein Query-Parameter *operation* eingeführt. Durch diesen wird dem Webservice mitgeteilt, welche Operation im Falle eines *POST* erwünscht ist. Das Erstellen einer *configuration* bekommt das Schlüsselwort *create* zugeordnet. Weiterhin muss kommuniziert werden, für welches TCsite-Produkt eine *configuration* erstellt werden soll. Hierfür wird der Query-Parameter *product* eingeführt.

Beispiel:

Ein *POST*-Request an „http://tcsite-uri.de/path-to-the-plugin/configuration?operation=create&product=notebook“

erstellt eine neue Notebook-Konfiguration.

3. **bestehende *configuration* kopieren & laden:** Es existiert keine HTTP-Methode, die semantisch dem Kopieren entspricht. Die Operation muss also auf *POST* abgebildet werden, da sie die einzige Methode ohne sichtbare Semantik ist. Für den *operation*-Parameter wird das Schlüsselwort *copy* gewählt.

Beispiel:

Ein *POST*-Request an „[http://tcsite-uri.de/path-to-the-plugin/  
configuration?id=1024&operation=copy](http://tcsite-uri.de/path-to-the-plugin/configuration?id=1024&operation=copy)“  
kopiert die *Quotation* mit der ID 1024.

4. **bestehende *configuration* ändern & laden:** Bedeutet die Verarbeitung einer gewählten Option des Anwenders während der Konfiguration in Shopware (Anforderung TC.F05). Semantisch wird dadurch ein Prozess ausgedrückt. Prozesse werden (im weiteren Sinne) der Methode *POST* zugeordnet. Für den *operation*-Parameter wird das Schlüsselwort *change* gewählt, da sich der Konfigurationszustand innerhalb der zugehörigen *Configuration* ändert. Weiterhin muss der Methode mitgeteilt werden, welche Option der Anwender gewählt hat. Die Codierung dieser Information ist ein Implementierungsdetail und wird in der Arbeit nicht weiter besprochen. Genauerer Aufschluss gibt Tacton Systems AB (2014b) im Abschnitt „ConfigurationActionBean“. Um nicht zu viele Query-Parameter zu erzeugen, können die Daten über die Wahl des Anwenders als JSON verpackt und im Entity-Body des HTTP-Requests mitgeschickt werden.

Beispiel:

Ein *POST*-Request an „[http://tcsite-uri.de/path-to-the-plugin/  
configuration?id=1024&operation=change](http://tcsite-uri.de/path-to-the-plugin/configuration?id=1024&operation=change)“  
mit einer JSON-Payload, in welcher die Wahl des Betriebssystems *OSX Yosemite* codiert wird, setzt diese Option unverhandelbar im Konfigurationszustand (falls sie nicht im Konflikt mit einer anderen Option steht).

#### 4.1.2.3. Zusammenfassung

Abbildung 4.10 fasst die Spezifikation des *configuration*-Ressourcenendpunkts tabellarisch zusammen.

Abbildungen/konzeptRestZusammenfassung.png

Abbildung 4.10.: Zusammenfassung der REST-Schnittstelle

## 4.2. Umsetzung

Im Folgenden findet ein Überblick über die Umsetzung und deren Herangehensweise statt. Der Ablauf der Implementierung wird episodisch besprochen. Währenddessen werden ausgewählte Umsetzungsdetails vorgestellt. Jeder Abschnitts schließt mit einer Zusammenfassung des jeweiligen Ergebnisses ab. Voraussetzung zum Verständnis dieses Kapitels sind Kenntnisse über die Grundlagen der Webprogrammierung.

Im Gegensatz zur Konzeptionsreihenfolge wurde die Entwicklung mit dem TCsite-Plugin begonnen. Zunächst wurde eine Webcontroller-Erweiterung zum Testen der Verbindung geschrieben. Um den Zugriff auf TCsite-Funktionalitäten zu ermöglichen, musste der in Kapitel 3.1.2.1 erwähnte *Integrationsuser* angelegt und dem Plugin zugewiesen werden. Als Testclient diente eine einfache HTML-Seite, die XMLHttpRequests (d.h. HTTP-Kommunikation mit Javascript) via *AJAX* an den Webcontroller absetzt. Hierbei hat sich herausgestellt, dass Cross-Origin Resource Sharing (CORS) beachtet werden muss. Im Folgenden wird die Problematik näher erläutert.

Browser implementieren ein Sicherheitskonzept namens Same-Origin-Policy (SOP). Dieses verhindert XMLHttpRequests an eine andere Domain als diejenige, von welcher der Request stammt. Beispielsweise würde eine Anfrage von der Domain „http://shopware-test.de“ an „http://tcsite-test.de“ blockiert werden. Die Serverantwort kommt zwar an, jedoch wird vom Browser vor dessen Auswertung der HTTP-Header *Access-Control-Allow-Origin* kontrolliert. Durch diesen wird signalisiert, welche Domains mit der angeforderten Ressource interagieren dürfen. Ist die Domain des Clienten nicht eingetragen, wirft der Browser einen Fehler (Mozilla Foundation, 2015). Eine genauere Darstellung des Sachverhalts ist Anhang A.10 zu entnehmen. Da Sicherheitsaspekte im Prototypen noch keine Rolle spielen, wurde der Tomcat von TCsite so eingestellt, dass er Requests von allen Domains akzeptiert. Das Ergebnis dieses Abschnitts ist die Herstellung einer Client-Server Kommunikation aus unterschiedlichen Domains.

Nachdem die Kommunikation zwischen dem Testclienten und dem TCsite-Plugin erfolgreich funktioniert hat, wurde das Erstellen einer *Quotation* samt *QuotationItem* und *Configurable* umgesetzt. Als Produkt diente dabei eine Umsetzung des Notebook-Beispiels. Im nächsten Schritt wurde die Generierung und Verarbeitung eines *KE-Input* erprobt. Der *KE-Input* wird in TCsite durch die Klasse *ConfigurationActionBean* implementiert. Aus dem *KE-Output* wurden die Daten für den initialen Aufbau der Konfigurationsoberfläche mittels Standardmethoden von TCsite zusammengestellt. Diese wurden als JSON-Payload an den Testclienten in der HTTP-Response zurückgeschickt. Während der Konzeption des TCsite-Plugins wurde erwähnt, dass diese Daten als serialisierter Baum übertragen werden. Der Baum wird von TCsite als *grouptree* bezeichnet. Die erste Version der *configuration*-Ressource enthält also nur UI-Daten. Das Ergebnis dieses Abschnitts ist ein erster Austausch von Ressourcen.

Danach wurde das Wählen von Optionen innerhalb des TCsite-Plugins simuliert. Dabei

wurde mit verschiedenen Werten für die Attribute der *ConfigurationActionBean* experimentiert. Im Testclient konnte anhand der Serverantwort beobachtet werden, ob der *KE-Input* erfolgreich von der Konfigurationsengine verarbeitet werden konnte, welche Optionen gesetzt wurden und welche Auswirkungen das auf den *grouptree* hat. Nun wurde im Testclienten eine sporadische Eingabemaske eingerichtet, die das Wählen von Optionen ermöglicht und diese zur Verarbeitung an den Service schickt. Das Ergebnis dieses Abschnitts ist das Wählen von Optionen über einen Clienten.

Bis jetzt wurde die *Quotation* mit den dazugehörigen Objekten nach der Generierung der Serverantwort – welche bis dahin nur aus dem *grouptree* bestand – sofort wieder vergessen. Mittels der Serviceklasse *ResourceService* des Repository-Moduls wurde das Speichern und Laden von *Quotations* erprobt. In Kapitel 3.1.2.1 wurde der Unterschied zwischen dem globalen und lokalen Speicher erklärt. Da die Anforderungsanalyse keine weiteren Detailbedingungen an die Persistierung des Konfigurationszustands stellt (TC.F02), wurde immer mit dem globalen Repository gearbeitet. Entitäten im globalen Speicher sind nutzerübergreifend sichtbar, was das Öffnen und Überprüfen der im Plugin gespeicherten *Quotations* in der Nutzeroberfläche von TCsite erlaubt. Um zu gewährleisten, dass bei aufeinanderfolgenden Optionen immer die gleiche *Quotation* geladen wird, musste die *configuration*-Ressource um die *configurationID* ergänzt werden. Diese wurde von nun an bei jedem Request mitgeschickt und die entsprechende *Quotation* geladen. Das Ergebnis dieses Abschnitts ist das sukzessive Wählen von Optionen per Client und das Speichern der sich ändernden Konfigurationszustände in TCsite. Somit konnte der Konfigurationsprozess über eine externe Domain gesteuert werden – wenn auch ohne Konfigurationsoberfläche.

Nun begann die Entwicklung des Shopware-Plugins. Mittels der im Konzeptionsteil vorgestellten Template-Erweiterungen wurde die sporadische Eingabemaske des Testclients in die Detailseite von Shopware übertragen. So wurde zunächst sichergestellt, dass die Kommunikation mit TCsite auch aus Shopware funktioniert. Als nächstes wurde die Konfigurationsoberfläche zum Ersatz der bisherigen Eingabemaske implementiert. Dafür wurde ein jQuery-Plugin geschrieben, welches mit einem rekursivem Algorithmus den *grouptree* parst und währenddessen die entsprechenden DOM-Elemente generiert und verkettet. Die resultierende HTML-Struktur wurde in den *DOM-tree* der Detailseite einhängt. Per CSS wurde die Darstellung der Oberfläche an das Shopware-Design angepasst. Als nächstes wurde über die Registrierung von Eventhandlern die Funktionalität der Interaktionselemente (Steps, Felder vom Typ *Menu* und *Number*) hergestellt. Außerdem wurde der Mechanismus zum Wechseln des Optionszustände zwischen verhandelbar/unverhandelbar implementiert. Jede Nutzerinteraktion wurde via *AJAX* an den Server übertragen, darauf hin der neue *grouptree* entgegen genommen und die Konfigurationsoberfläche aktualisiert. Somit läuft der Konfigurationsprozess wie eine Single-Page-Application ab, d.h. ohne Seitenreload. Ergebnis dieses Abschnitts ist die Erstellung der Konfigurationsoberfläche, die einen Konfigurationsprozess äquivalent zu TCsite ermöglicht (entspricht Anforderung SW.F02). Dieser Teil der Umsetzung hat den größten Teil der praktischen Bearbeitungszeit ausgemacht.

Nun wurde der konfigurierte Artikel in den Einkaufsvorgang integriert. Alle Umsetzungsschritte entsprechen den Beschreibungen aus Kapitel 4.1.1. Darin wurden die verwendeten Erweiterungspunkte bereits erläutert. Der Lebenszyklus der dynamisch generierten Varianten wurde stets mit der mySQL-Administrationsanwendung *phpMyAdmin* überprüft und auf Fehler kontrolliert. Die Funktionalität des Webservices wurde nur bei Bedarf erweitert. So wurde der Serviceendpunkt zum Löschen einer *Quotation* erst eingerichtet, als das Löschen von dynamisch generierten Varianten in Shopware implementiert wurde. Das Umkonfigurieren von Varianten wurde als letztes umgesetzt. Ergebnis dieses Abschnitts ist die Integration der Varianten in den Einkaufsvorgang, die aus dem Konfigurationsprozess resultieren. Damit ist die Implementierung abgeschlossen.

### **Zusammenfassung**

Zunächst wurde eine Grundlage für die Kommunikation von TCsite mit externen Systemen geschaffen (CORS/*Integrationsuser*). Danach wurden die Serviceendpunkte im TCsite-Plugin soweit implementiert, dass ein Client einen Konfigurationsprozess durchführen konnte. Anschließend wurde die Konfigurationsoberfläche implementiert. Letztendlich wurden die entstehenden Varianten in den Einkaufsvorgang integriert.



### 4.3. Fazit

Während der Umsetzung konnten alle funktionalen und nichtfunktionalen Anforderungen erfüllt werden. Resultat dieser Arbeit sind zwei Plugins. Nach deren Installation in den jeweiligen Systemen sowie geringem Einstellungsaufwand (Eintragung der Serviceadresse im Shopware-Plugin, CORS-Aktivierung bei TCsite) ermöglichen sie die Tacton-Produktkonfiguration in einem Open-Source Shopsystem. Damit war die Integration erfolgreich. Der Kunde merkt während des Einkaufsvorgangs nicht, dass der Konfigurationsprozess durch einen externen Dienst begleitet wird. Im Folgenden findet eine Diskussion des Ergebnisses statt. Dies geschieht sowohl aus einem technischen, als auch einem anwendungsbezogenen Standpunkt. Dabei werden Alternativen und Potentiale für die zukünftige Entwicklung aufgezeigt.

Es wird mit der Bewertung des Webservices begonnen. In Kapitel 4.1.2 wurde die Behauptung aufgestellt, dass dieser nach dem REST-Architekturmuster entworfen wurde. Im Grundlagenteil der Arbeit (Kapitel 2.3.2) wurde erwähnt, dass das „Richardson Maturity Model“ zur abgestuften Bewertung der REST-Konformität existiert (Wilde und Pautasso, 2011). Je nachdem, welche Kriterien erfüllt werden, unterscheidet es vier aufeinander aufbauende Stufen:

**Level 0:** Dokumentenaustausch via HTTP. Auch SOAP würde dieses Kriterium erfüllen.

**Level 1:** Exponierung adressierbarer Ressourcen. Die Interaktion mit dem Webservice findet dabei nur über diese Ressourcen statt.

**Level 2:** Einhaltung der uniformen Schnittstelle, d.h. die Implementierung der Methoden entsprechend der HTTP-Spezifikation.

**Level 3:** Verknüpfung der Ressourcen untereinander, d.h. Hypermedia.

Im Folgenden wird das Bewertungsmodell auf die Implementierung des TCsite-Plugins angewandt:

**Level 0:** Das Plugin verwendet Webcontroller, welche HTTP-Requests verarbeiten und beantworten. Damit ist die Bedingung für Level 0 erfüllt.

**Level 1:** Das Plugin exponiert die *configuration*-Ressource. Es gibt keine weiteren Endpunkte, die nicht mit dieser Ressource in Verbindung stehen. Damit ist die Bedingung für Level 1 erfüllt.

**Level 2:** In Kapitel 2.3.2 wurde erwähnt, dass laut Richardson und Ruby (2007) die uniforme Schnittstelle verletzt wird, wenn *POST* im weiteren Sinne verwendet wird (d.h. zum Starten von Prozessen). Damit wird Level 2 nicht erfüllt. Da die Stufen aufeinander aufbauen, wird Level 3 nicht mehr geprüft.

Der Implementierung des REST-Webservice entspricht also Level 1 des „Richardson Maturity Models“. Die Implementierung ist an der Einhaltung der uniformen Schnittstelle

gescheitert. Verarbeitungsschritte sind aber inhärenter Bestandteil des Konfigurationsprozesses. Damit wird klar, dass sich eine Produktkonfiguration nicht problemlos auf einen REST-Webservice übertragen lässt. Das Erweiterungskonzept von TCsite hat durch die Bereitstellung von Webcontrollern einen leichten Zugang zum REST-Ansatz ermöglicht. Es ist zu überprüfen, ob ein Konfigurationsprozess bei Integrationsszenarien nicht besser über die RPCs des SOAP-Ansatzes vereinbar sind.

Durch die momentane CORS-Aktivierung würde die Inbetriebnahme des TCsite-Plugins dem Anbieten einer öffentlichen Web-API gleichkommen. Dafür berücksichtigt die Lösung noch nicht ausreichend nichtfunktionale Aspekte. Dazu gehören zum Beispiel der Umgang mit Denial-of-Service (DoS-)Attacken oder anderen Angriffen. Ein Überblick von Best Practices für Web-APIs ist Tilkov (2011) zu entnehmen.

In der momentan entworfenen Lösung wird jedes mal, wenn die Detailseite eines konfigurierbaren Artikels in Shopware aufgerufen wird, eine Quotation in TCsite angelegt. Zu keinem Zeitpunkt wird überprüft, welche der Quotations mit einer Variante im Zusammenhang steht, die tatsächlich bestellt wurde. Wird eine Detailseite nur aus Interesse geöffnet, die Konfiguration abgebrochen oder der Warenkorb nie in eine Bestellung umgesetzt, werden „*Quotation*-Leichen“ in der Datenbank von TCsite akkumuliert. Damit wirkt sich der Entwurf negativ auf die Performance von TCsite aus, was nicht im Sinne eines Integrationskonzepts sein kann. Allerdings wurde in Kapitel 4.1.2 begründet, dass das Erstellen von *Quotations* nicht vermieden werden kann. Ein Lösungsansatz ist die Entwicklung eines Mechanismus zur Garbage-Collection, der unnötige *Quotations* post hoc entfernt. Außerdem ist zur Reduktion der Anzahl angelegter *Quotations* zu überlegen, ob das Verhältnis von *Configurables* respektive *QuotationItems* zu *Quotations* überarbeitet wird. Intuitiver als die bisherige Lösung wäre die Registrierung einer *Quotation* pro Warenkorb in Shopware. Innerhalb der *Quotation* gäbe es dann für jede Warenkorbbposition ein *QuotationItem*.

Weiterhin kann der Lebenszyklus von *Quotations*<sup>5</sup> genutzt werden, um neben der Container-Funktionalität auch die Ausdruckskraft einer *Quotation* zu nutzen. Die Einrichtung weiterer Zustände, wie z.B. „Bestellung im Shop ausgelöst“ würde Vertriebsmitarbeitern die Verwaltung der Bestellungen erleichtern.

Dies leitet zu dem in Kapitel 4.1 angesprochenen Problem über, dass der Vertrieb eine Möglichkeit zur Interpretation der bestellten Varianten haben muss. Der momentane Lösungsentwurf ist das Öffnen der Varianten in TCsite. Das erfordert zur Behandlung einer Bestellung den Wechsel zwischen zwei Anwendungen – Shopware und TCsite. Neben dem umständlichen Workflow setzt diese Lösung implizit voraus, dass der entsprechende Mitarbeiter überhaupt Zugang zu TCsite hat. Da TCsite ein kostenpflichtiges Lizenzmodell besitzt, ist das nicht nur eine technische Problematik. Ein Lösungsansatz wäre das Bereitstellen weiterer Dokumente in der Bestellübersicht von Shopware, wie z.B. Stücklisten oder Konstruktionszeichnungen.

---

<sup>5</sup>Der *Quotation*-Lebenszyklus wurde in Kapitel ?? vorgestellt

Momentan gewährleistet die Konfigurationsoberfläche nur, dass der Shopkunde eine korrekte Variante bestellt (vorausgesetzt, das Konfigurationsmodell ist richtig definiert). Ansonsten hat der Konfigurationsprozess für den Anwender keinerlei informativen Charakter. Abgesehen von der Felddescription bietet die Oberfläche keine Begleitinformation. Diese muss noch mit Bildern, beschreibenden Texten oder sonstigen medialen- und interaktiven Elementen angereichert werden. Im Augenblick ist die Nutzung des Konfigurators nur unter der Voraussetzung möglich, dass der Anwender bereits Vorwissen über alle wählbaren Optionen hat. Davon kann nicht ausgegangen werden, weshalb der momentan Prototyp für den Einsatz im Endverbrauchermarkt noch ungeeignet ist.

Abschließend wird das Ergebnis der Entwicklung in den ökonomischen Bezugsrahmen eingeordnet, der zu Beginn der Arbeit eröffnet wurde. Das Potential der Integration wird sowohl im B2B- als auch B2C-Bereich erläutert.

Im B2B-Bereich ist der entscheidende Faktor der Implementierung, dass Artikel, die im Shop konfiguriert wurden, in TCsite geöffnet werden können. Damit wird dem CPQ-Prozess eine neue Dimension gegeben. Mitarbeiter des Kundenunternehmens können Artikel selbst konfigurieren, welche daraufhin die Basis zur Angebotserstellung im verkaufenden Unternehmen darstellen. So können die einzelnen CPQ-Bestandteile noch stärker auf die jeweiligen Verantwortungsbereichen aufgeteilt werden. Der Kunde drückt seine Vorstellungen im Konfigurationsprozess aus (C..) während der Anbieter die Preiskalkulation und Angebotserstellung (..PQ) durchführt.

Im B2C-Bereich wurde während der Arbeit ein Mass Customization Toolkit umgesetzt. Durch die Verwendung von Open-Source-Technologie berücksichtigt die Implementierung die Grundwerte des MC, die sich in der Individualisierung bei gleichzeitiger Kostenreduktion ausdrücken. Die Verwendung frei zugänglicher Software ist eine eindeutige Einsparmaßnahme. Somit wird eine Weiterverfolgung der Entwicklung nicht nur von technischem, sondern auch wirtschaftlichem Interesse sein.

# Literaturverzeichnis

- [Bartelt u. a. 2000] BARTELT, Andreas ; WEINREICH, Harald ; LAMERSDORF, Winfried: Kundenorientierte Aspekte der Konzeption von Online-Shops / Universität Hamburg, Fachbereich Informatik, Verteilte Systeme (VSYS). 2000. – Forschungsbericht
- [Boles und Haber 2000] BOLES, Dietrich ; HABER, Cornelia: DDS: Ein Shop-System für den Informationcommerce / Informatik-Institut OFFIS Oldenburg. 2000. – Forschungsbericht
- [Brown und Chandrasekaran 1989] BROWN, David C. ; CHANDRASEKARAN, B.: *Design Problem Solving: Knowledge Structures and Control Strategies. Research Notes in Artificial Intelligence*. London : Pitman, 1989
- [Falkner u. a. 2011] FALKNER, Andreas ; FELFERNIG, Alexander ; HAAG, Albert: Recommendation Technologies for Configurable Products. In: *AI Magazine* 32 (2011), Nr. 3, S. 99–108
- [Felfernig u. a. 2014] FELFERNIG, Alexander ; HOTZ, Lothar ; BAGLEY, Claire ; TIHONEN, Juha: *Knowledge-Based Configuration: From Research to Business Cases*. Elsevier, 2014
- [Fielding 2000] FIELDING, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Dissertation, 2000
- [Frayman und Mittal 1989] FRAYMAN, Felix ; MITTAL, Sanjay: Towards a generic model of configuration tasks. In: *International Joint Conference on Artificial Intelligence (IJCAI-89)*, 1989
- [FWP shop 2014] FWP SHOP: *Unterschied: Magento Community vs Enterprise Edition*. 2014. – URL <http://www.fwpshop.org/shopsysteme/magento-shop/versionsunterschiede>. – Zugriff: 25.08.2015
- [GNU 2014] GNU: *GNU Affero General Public License*. 2014. – URL <http://www.gnu.org/licenses/agpl-3.0.de.html>. – Zugriff: 06.09.2015
- [Graf 2014] GRAF, Alexander: *Das beste Shopsystem 2014 – Erwartung vs. Wirklichkeit*. 2014. – URL <http://www.fwpshop.org/shopsysteme/magento-shop/versionsunterschiede>. – Zugriff: 26.08.2015
- [Hadzic und Andersen 2004] HADZIC, Tarik ; ANDERSEN, Henrik R.: An introduction

- to solving interactive configuration problems / IT University of Copenhagen. 2004. – Forschungsbericht
- [Lutz 2011] LUTZ, Christoph: *Rechnergestuetztes Konfigurieren und Auslegen individualisierter Produkte*, Technischen Universitaet Wien, Dissertation, 2011
- [Magento Inc. 2015] MAGENTO INC.: *Magento Products Overview*. 2015. – URL <http://magento.com/products/overview>. – Zugriff: 25.08.2015
- [Meier und Stormer 2012] MEIER, Andreas ; STORMER, Henrik: *eBusiness und eCommerce: Management der digitalen Wertschöpfungskette*. 3. Auflage. Springer Gabler, 2012
- [Mozilla Foundation 2015] MOZILLA FOUNDATION: *HTTP access control (CORS)*. 2015. – URL [https://developer.mozilla.org/en-US/docs/Web/HTTP/Access\\_control\\_CORS](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS). – Zugriff: 18.09.2015
- [Nögel 2015a] NÖGEL, Daniel: *Bootstrapping Shopware: The dispatch loop*. 2015. – URL <https://developers.shopware.com/blog/2015/08/26/bootstrapping-shopware-the-dispatch-loop/>. – Zugriff: 07.09.2015
- [Nögel 2015b] NÖGEL, Daniel: *Bootstrapping Shopware: The dispatch loop*. 2015. – URL <https://developers.shopware.com/blog/2015/06/09/understanding-the-shopware-hook-system/>. – Zugriff: 07.09.2015
- [Opencart Limited 2015] OPENCART LIMITED: *Opencart Extensions*. 2015. – URL <http://www.opencart.com/index.php?route=extension/extension>. – Zugriff: 25.08.2015
- [OXID eSales AG 2015] OXID ESALES AG: *OXID eSales Produkte*. 2015. – URL <https://www.oxid-esales.com/de/produkte.html>. – Zugriff: 25.08.2015
- [Piller 1998] PILLER, Frank T.: *Kundenindividuelle Massenproduktion: die Wettbewerbssstrategie der Zukunft*. Carl Hanser Verlag München Wien, 1998
- [Piller 2006] PILLER, Frank T.: *Mass Customization. Ein wettbewerbsstrategisches Konzept im Informationszeitalter*. 4., überarbeitete und erweiterte Auflage. Wiesbaden : Deutscher Universitäts-Verlag | GWV Fachverlage GmbH, 2006
- [Porter 1980] PORTER, Michael: *Competetive Strategy: Techniques for Analyzing Industries and Competitors*. New York, 1980
- [Porter 2002] PORTER, Michael: *Wettbewerbsstrategie: Methoden zur Analyse von Branchen und Konkurrenten*. 11. Frankfurt, New York, 2002
- [PrestaShop SA 2015] PRESTASHOP SA: *Prestashop Addon Marketplace*. 2015. – URL <http://addons.prestashop.com/>. – Zugriff: 25.08.2015
- [Richardson und Ruby 2007] RICHARDSON, Leonard ; RUBY, Sam: *Web Services mit REST*. Köln : O'Reilly Verlag, 2007

- [Sabin und Weigel 1998] SABIN, Daniel ; WEIGEL, Rainer: Product Configuration Frameworks-A Survey. In: *IEEE Intelligent Systems* 13 (1998), Nr. 4, S. 42–49
- [Schomburg 1980] SCHOMBURG, Eckart: *Entwicklung eines betriebstypologischen Instrumentariums zur systematischen Ermittlung der Anforderungen an EDV-gestützte Produktionsplanungs- und -steuerungssysteme im Maschinenbau*, RWTH Aachen, Dissertation, 1980
- [Schuh 2005] SCHUH, Günther: *Produktkomplexität managen: Strategien - Methoden - Tools*. Carl Hanser Verlag München Wien, 2005
- [Schuh 2006] SCHUH, Günther: *Produktionsplanung und -steuerung. Grundlagen, Gestaltung und Konzepte*. 3. Berlin : Springer, 2006
- [Schwarze und Schwarze 2002] SCHWARZE, Jochen ; SCHWARZE, Stephan: *Electronic Commerce: Grundlagen und praktische Umsetzung*. Herne/Berlin : Verlag Neue Wirtschafts-Brife, 2002
- [Shopify 2015] SHOPIFY: *Shopify Pricing*. 2015. – URL <http://www.shopify.com/pricing>. – Zugriff: 26.08.2015
- [shopware AG 2014a] SHOPWARE AG: *REST API Ressourcen Übersicht*. 2014. – URL [http://community.shopware.com/REST-API-Ressourcen-%C3%9Cbersicht\\_detail\\_1680.html](http://community.shopware.com/REST-API-Ressourcen-%C3%9Cbersicht_detail_1680.html). – Zugriff: 07.09.2015
- [shopware AG 2014b] SHOPWARE AG: *Shopware 4 Developers Guide*. 2014. – URL [http://community.shopware.com/Developers-Guide\\_cat\\_796.html](http://community.shopware.com/Developers-Guide_cat_796.html). – Zugriff: 07.09.2015
- [shopware AG 2014c] SHOPWARE AG: *Shopware 4 Plugin Konfiguration*. 2014. – URL [http://community.shopware.com/Shopware-4-Plugin-Konfiguration\\_detail\\_978.html](http://community.shopware.com/Shopware-4-Plugin-Konfiguration_detail_978.html). – Zugriff: 17.09.2015
- [shopware AG 2015a] SHOPWARE AG: *Shopware 5 Developers Guide*. 2015. – URL <https://developers.shopware.com/developers-guide/>. – Zugriff: 07.09.2015
- [shopware AG 2015b] SHOPWARE AG: *Shopware 5 Funktionsübersicht*. 2015. – URL [https://en.shopware.com/media/pdf/shopware\\_5\\_funktionsuebersicht.pdf](https://en.shopware.com/media/pdf/shopware_5_funktionsuebersicht.pdf). – Zugriff: 07.09.2015
- [shopware AG 2015c] SHOPWARE AG: *Shopware Backenderweiterungen Beispielprojekte*. 2015. – URL [http://community.shopware.com/Backend\\_cat\\_871.html](http://community.shopware.com/Backend_cat_871.html). – Zugriff: 14.09.2015
- [shopware AG 2015d] SHOPWARE AG: *Shopware Datenbankschema*. 2015. – URL <http://community.shopware.com/files/downloads/sw4-orm-1733172.png>. – Zugriff: 13.09.2015

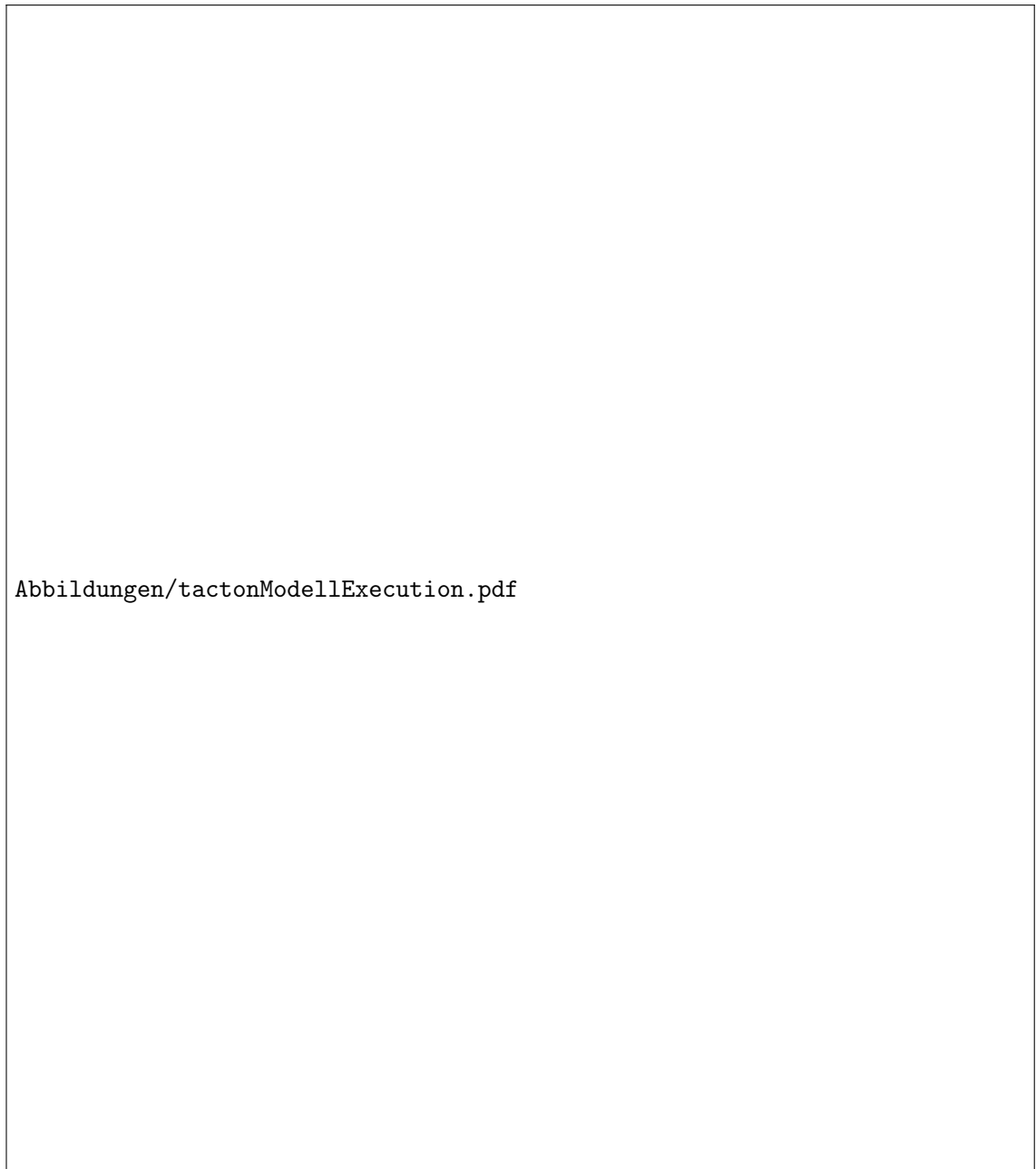
- [shopware AG 2015e] SHOPWARE AG: *Shopware Dokumentation*. 2015. – URL [http://community.shopware.com/Doku\\_cat\\_938.html](http://community.shopware.com/Doku_cat_938.html). – Zugriff: 25.08.2015
- [shopware AG 2015f] SHOPWARE AG: *Shopware Plugin Store*. 2015. – URL <http://store.shopware.com/>. – Zugriff: 25.08.2015
- [shopware AG 2015g] SHOPWARE AG: *Shopware Pricing*. 2015. – URL <https://en.shopware.com/pricing/>. – Zugriff: 25.08.2015
- [shopware AG 2015h] SHOPWARE AG: *Über die shopware AG*. 2015. – URL <http://store.shopware.com/unternehmen>. – Zugriff: 06.09.2015
- [Soininen u. a. 1998] SOININEN, Timo ; TIIHONEN, Juha ; MÄNNISTÖ, Tomi ; SULONEN, Reijo: Towards a general ontology of configuration. In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 12 (1998), Nr. 4, S. 357–372
- [Stahl u. a. 2015] STAHL, Ernst ; WITTMANN, Georg ; KRABICHLER, Thomas ; BREITSCHAFT, Markus: *E-Commerce-Leitfaden: Noch erfolgreicher im elektronischen Handel*. 3., vollständig überarbeitete und erweiterte Auflage. Universitätsverlag Regensburg, 2015
- [Strato AG 2015] STRATO AG: *Strato Webshops*. 2015. – URL <https://www.strato.de/webshop>. – Zugriff: 25.08.2015
- [t3n 2014] T3N: *Open Source: 16 Shopsysteme im Überblick*. 2014. – URL <http://t3n.de/news/open-source-shopsysteme-13-losungen-uberblick-286546/>. – Zugriff: 26.08.2015
- [Tacton Systems AB 2006] TACTON SYSTEMS AB: *Introduction to Modeling with Tacton Configurator Studio 4*. 2006
- [Tacton Systems AB 2007] TACTON SYSTEMS AB: *Bridging the gap between engineering and sales for complex products*. 2007. – internes Präsentationsdokument zur Produktübersicht
- [Tacton Systems AB 2013] TACTON SYSTEMS AB: *TCsite Application Development Manual*. 2013
- [Tacton Systems AB 2014a] TACTON SYSTEMS AB: *TCsite API Documentation*. 2014
- [Tacton Systems AB 2014b] TACTON SYSTEMS AB: *TCsite Application Development Manual*. 2014
- [Tacton Systems AB 2014c] TACTON SYSTEMS AB: *TCsite Reference Manual*. 2014
- [Tacton Systems AB 2015] TACTON SYSTEMS AB: *About Tacton*. 2015. – URL <http://www.tacton.com/about-tacton/>. – Zugriff: 27.08.2015
- [Tilkov 2011] TILKOV, Stefan: *REST und HTTP - Einsatz der Architektur des Web für Integrationsszenarien*. 2. Heidelberg : dpunkt.verlag, 2011

- [Timmers 1998] TIMMERS, Paul: Business Models for Electronic Markets. In: *Electronic Markets* 8 (1998), Nr. 2, S. 3–8
- [W3C 2004] W3C: *Web Services Glossary*. 2004. – URL <http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice>. – Zugriff: 21.08.2015
- [Wilde und Pautasso 2011] WILDE, Eric ; PAUTASSO, Cesare: *REST: From Research to Practice*. Springer Verlag, 2011
- [WsWiki 2009] WSWIKI: *Web Service Specifications*. 2009. – URL <https://wiki.apache.org/ws/WebServiceSpecifications>. – Zugriff: 21.08.2015



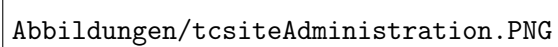


## A. Anhang



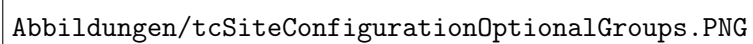
Abbildungen/tactonModellExecution.pdf

Abbildung A.1.: Vollständige Darstellung der generischen Execution-Struktur



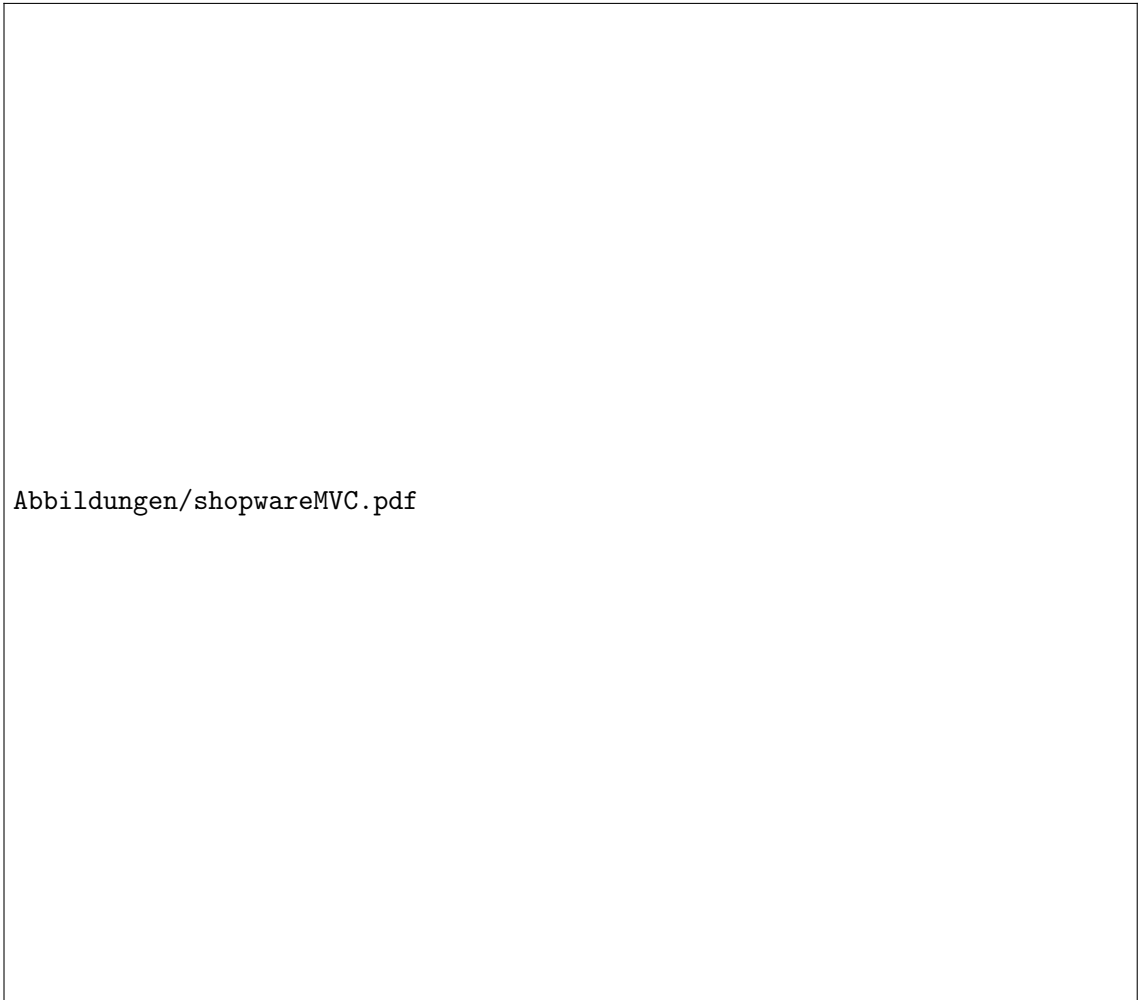
Abbildungen/tcsiteAdministration.PNG

Abbildung A.2.: TCsite Administrationsoberfläche




Abbildungen/tcSiteConfigurationOptionalGroups.PNG

Abbildung A.3.: Darstellung optionaler Groups in TCsite




Abbildungen/shopwareMVC.pdf

Abbildung A.4.: vollständige Shopware High-Level-Architektur



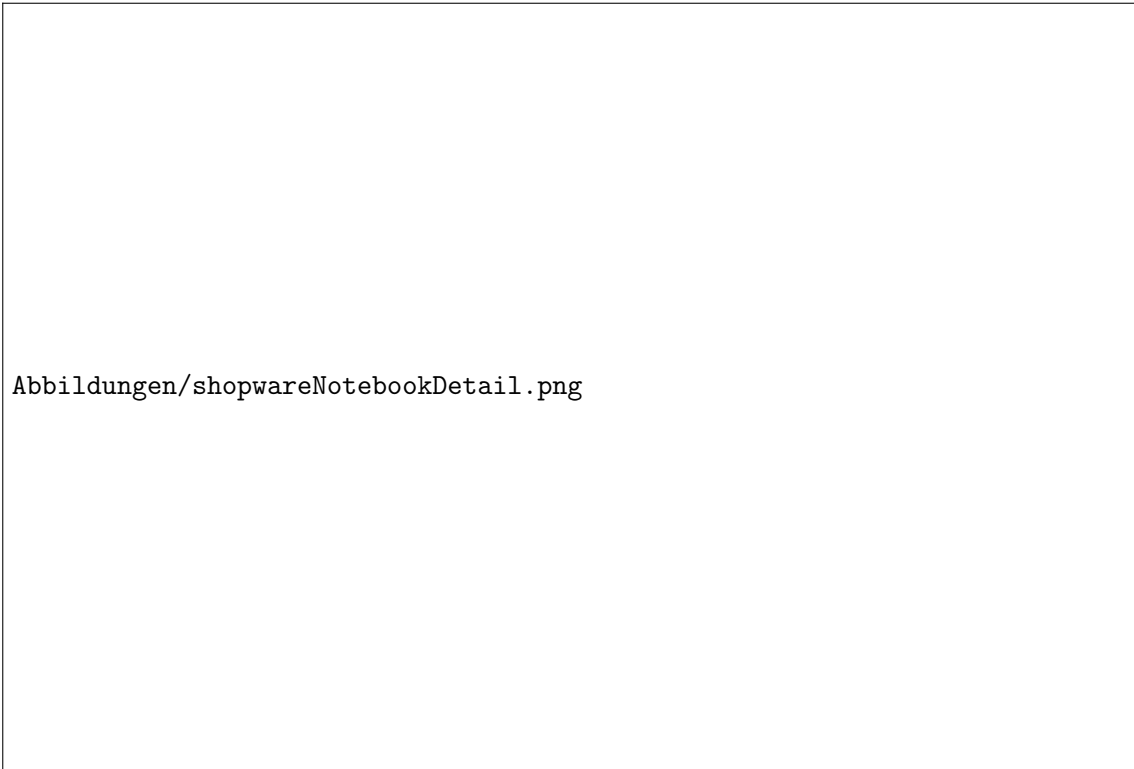
Abbildungen/shopwareBackendArtikel.png

Abbildung A.5.: Anlegen eines Artikels im Backend



Abbildungen/shopwareBackendArtikelVarianten.png

Abbildung A.6.: Generieren der Varianten im Backend



Abbildungen/shopwareNotebookDetail.png

Abbildung A.7.: Detailansicht eines konfigurierbaren Notebooks in shopware.



Abbildung A.8.: Warenkorb mit konfiguriertem Artikel.

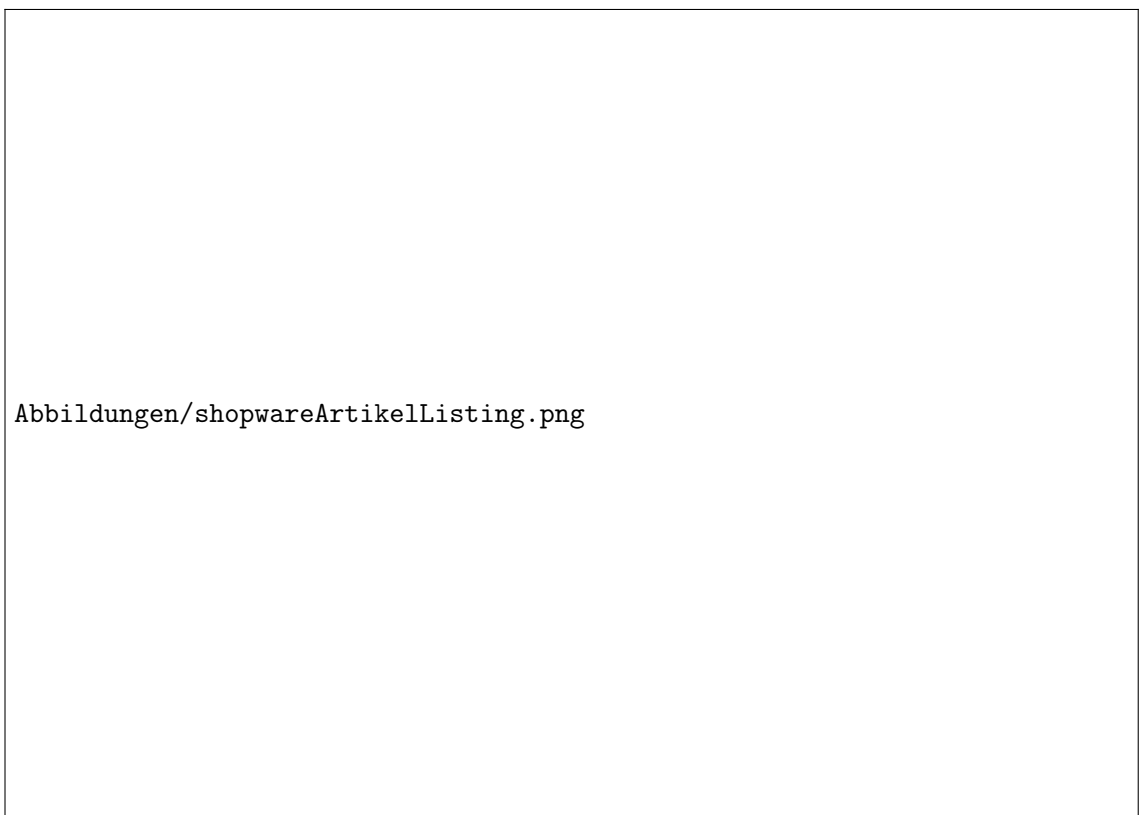
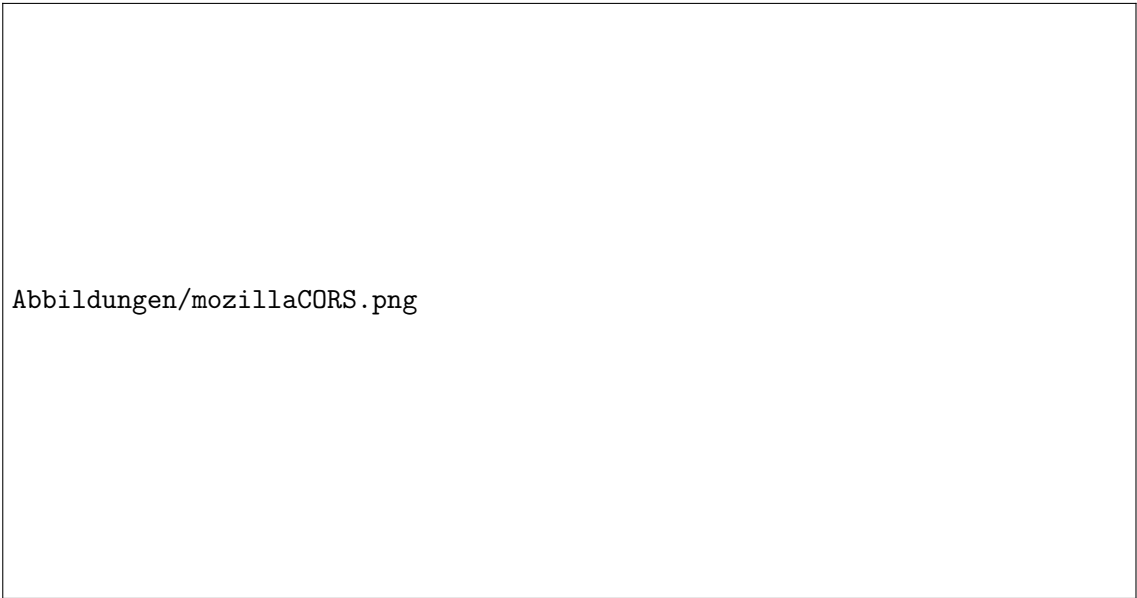


Abbildung A.9.: Artikellisting in Shopware.



Abbildungen/mozillaCORS.png

Abbildung A.10.: CORS-Flowchart.

# Erklärung

Hiermit versichere ich, dass ich meine Abschlussarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum:

.....

(Unterschrift)