

Inhaltsverzeichnis

1. Einführung	1
1.1. Motivation	2
1.2. Zielsetzung	3
1.3. Aufbau der Arbeit	3
2. Grundlagen	4
2.1. Modellgetriebene Softwareentwicklung (MDSD)	5
2.1.1. Domäne	5
2.1.1.1. Definition	5
2.1.1.2. Domänenanalyse	5
2.1.1.3. Feature Modelling	5
2.1.2. Metamodell	5
2.1.2.1. Definition	5
2.1.2.2. Abstraktes Modell	5
2.1.2.3. Konkretes Modell	5
2.1.3. Domänenspezifische Sprache (DSL)	5
2.1.3.1. Definition	5
2.1.3.2. General Purpose Language (GPL)	5
2.1.3.3. Interne DSLs	5
2.1.3.4. Externe DSLs	5
2.1.3.5. Parser	5
2.1.4. Code Generator	5
2.1.4.1. Definition	5
2.1.4.2. Techniken zur Generierung von Code	5
2.1.4.3. Zusammenhang zu Transformatoren	5
2.1.4.4. Abgrenzung zum Compilerbau	5
2.2. Software Architektur	5
2.2.1. Prinzipien der Softwaretechnik	5
2.2.1.1. Abstraktion	5
2.2.1.2. Bindung und Kopplung	5
2.2.1.3. Modularisierung	5
2.2.1.4. Weitere Prinzipien	5
2.2.1.5. Abhängigkeiten	5
2.2.2. Trennung durch Modelle	5
2.2.3. Modell-Transformations-Pipeline	5

2.3.	Design Pattern objektorientierter Programmierung	5
2.3.1.	Visitor Pattern	5
2.3.1.1.	Zweck	5
2.3.1.2.	Beschreibung	5
2.3.1.3.	Anwendbarkeit	5
2.3.2.	Builder Pattern und Fluent Interfaces	5
2.3.2.1.	Zweck	5
2.3.2.2.	Beschreibung	5
2.3.2.3.	Anwendbarkeit	5
2.3.3.	Factory Pattern	5
2.3.3.1.	Zweck	5
2.3.3.2.	Beschreibung	5
2.3.3.3.	Anwendbarkeit	5
3.	Problemstellung	6
3.1.	Allgemeine Probleme	7
3.1.1.	Analyse der Domäne	7
3.1.1.1.	Entwicklung einer Referenzimplementation	7
3.1.1.2.	Definition von Variabilitäten	7
3.1.1.3.	Formulierung der Domäne als abstraktes Metamodell . .	7
3.1.2.	Anreicherung des Metamodells mit konkreten Daten	7
3.1.2.1.	Wahl einer geeigneten Schnittstelle	7
3.1.2.2.	Implementation einer DSL	7
3.1.3.	Abwägung zwischen Umsetzung als Generator oder als Transformator	7
3.1.3.1.	Eingesetzte Technik	7
3.1.3.2.	Neuentwicklung	7
3.1.3.3.	Vorhandene Frameworks als Basis	7
3.1.4.	Verwendung von Zwischenmodellen	7
3.2.	Wirtschaftliche Probleme	7
3.2.1.	Aufwandsvergleich zwischen MDSD und konventioneller Softwareentwicklung	7
3.2.2.	Erhöhung der Wirtschaftlichkeit durch Verwendung eines Meta-Generators	7
3.3.	Spezielle Probleme bei der Entwicklung eines Meta-Generators für Java-Code	7
3.3.1.	Java-Code als Ausgangsmodell	7
3.3.1.1.	Informationsanreicherung des Java-Code-Modells	7
3.3.1.2.	Parsen des Codes und Umsetzung in ein neues Modell . .	7
3.3.1.3.	Einschränkung auf eine Teilmenge der Java Features . .	7
3.3.2.	Modell zur Plattform unabhängigen Abbildung von Code	7
3.3.2.1.	DSL als beschreibende Schnittstelle für den Entwickler .	7
3.3.2.2.	xxx	7

3.3.3. Besonderheiten bei der Abwägung zwischen Generator und Transformator	7
4. Lösung: Spectrum (Proof of Concept)	8
4.1. Verwendete Bibliotheken	9
4.2. Verwendeter Glossar	9
4.2.1. JavaParser mit JavaSymbolSolver	9
4.2.2. JavaPoet	9
4.3. Architekturübersicht	9
4.3.1. Amber	9
4.3.1.1. Annotationen	9
4.3.1.2. Parser	9
4.3.1.3. Modell	9
4.3.2. Cherry	9
4.3.2.1. Generator	9
4.3.2.2. Modell	9
4.3.2.3. Generierte Builder	9
4.3.3. Jade	9
4.3.3.1. Transformator	9
4.3.4. Scarlet	9
4.3.4.1. Generator	9
4.3.4.2. Modell	9
5. Evaluierung	11
5.1. Softwarequalität	11
5.1.1. Functionality	11
5.1.2. Maintainability	11
5.1.3. Performance	11
5.1.4. Usability	11
5.2. Grenzen des Lösungsansatzes	11
6. Zusammenfassung und Ausblick	13
6.1. Zusammenfassung	13
6.2. Ausblick	13
A. Dokumentation	14
A.1. Verwendung der Annotationen	14
A.2. Verwendung der generierten CodeUnit-Builder	14
A.3. Klassendokumentation	14
A.3.1. Amber	14
A.3.2. Cherry	14
A.3.3. Jade	14
A.3.4. Scarlet	14

Inhaltsverzeichnis

Verzeichnisse	15
Literatur	17
Eidesstattliche Erklärung	18
Zustimmung zur Plagiatsüberprüfung	19