MARCO PAULO DE FIGUEIREDO CRAVEIRO

MODEL ASSISTED SOFTWARE DEVELOPMENT

MODEL ASSISTED SOFTWARE DEVELOMENT

MARCO PAULO DE FIGUEIREDO CRAVEIRO

A MDE-based Software Development Methodology

Doctor of Philosophy (PhD)
University of Hertfordshire

February 2022 — v1.2

Submitted to the University of Hertfordshire in partial fulfilment
of the requirements of the degree of PhD (Doctor of Philosophy)

Marco Paulo de Figueiredo Craveiro: *Model Assisted Software Development*, A MDE-based Software Development Methodology, Doctor of Philosohphy (PhD) © 2022

SUPERVISORS:
Dr. Michael Schmuker
Dr. Volker Steuber

LOCATION:
Hatfield

TIME FRAME:
September 2022

To my wife and kids: thank you.

Para meus avós, pais e irmãos: malembe malembe.

Para meus primos Elsa e Bruno: estamos juntos.

ABSTRACT

Model Driven Engineering (MDE) is a flexible approach for the creation and evolution of software systems, centred around models and their transformations. MDE provides a fundamental substrate upon which practitioners can create sophisticated solutions, invariably characterised by a high degree of automation, but tailored specifically to their problem domain. Adoption literature reports of widespread MDE use across industry and academia but also underscores its status as a niche technology. Meanwhile, the challenges it is purported to overcome continue to loom large over software engineering.

The present work identifies factors underlying the deficit in MDE adoption, both theoretical and practical, and determines the extent to which a new MDE-based Software Development Methodology (SDM) can be used to address them. It does so by putting forward Model Assisted Software Development (MASD), a novel SDM that aids in the design and implementation of software systems. MASD trades the flexibility and power of MDE for a reduction in complexity, and consequently has a restricted but better defined range of applications. MASD's problem space is a subset of the solution space itself: it provides well-defined abstractions over elements of the domain of software engineering and a conceptual framework for their manipulation. MASD targets software developers with little to no knowledge of MDE, and aims to act as a bridge between traditional software engineering and model-driven approaches.

This dissertation describes the motivation for MASD, the core elements that make up the methodology and how they interact, and, finally, its application. It includes empirical evidence of its adoption by means of case studies, and provides a detailed description of its reference implementation, itself created using MASD.

ACKNOWLEDGEMENTS

In Africa there is a concept known as "ubuntu" — the profound sense that we are human only through the humanity of others; that if we are to accomplish anything in this world it will in equal measure be due to the work and achievement of others.

— Nelson Mandela (Stengel, 2010)

A LONG TERM ENDEAVOUR as complex as this would certainly not come to fruition were it not for the help of a large number of people. First and foremost, I'd like to thank my wife *Shahinara* and kids *Isaac* and *Eliana* for putting up with my eccentricities, long work hours and all manner of odd behaviour in order to reach the finish line.

I'd like to thank all of my fellow Ph.D. colleagues at UH who undertook this journey with me; our conversations kept me sane and focused.

I'd like to thank all my friends at *The Financial Company*, as well as *Indranil* and *Ian*, for always cheering me on, no matter what I get myself into.

I'd like to thank *Piyush Shah* for having faith in me by giving me my first real programming job and for sponsoring me through a MSc programme that changed my life, and allowed me to meet *Scott* and *Andy*, who are probably responsible for making me think I could do a PhD.

Finally, I'd like to thank *Ricardo Pereira*, my high school teacher who taught me all I know about programming, and my friends *Adilson*, *Jójó* and *Lau*, with whom I shared so many moments in life.

Marco Craveiro, *Hatfield, January 2022.*

MASD

Model Assisted Software Development

CONTENTS

LIST OF FIGURES

LISTINGS

ACRONYMS

AC-MDSD	Architecture-Centric MDSD.
AMDD	Agile MDD.
AO	Aspect Oriented.
AO-MD-PLE	Aspect-Oriented Model Driven PLE.
AOP	Aspect Oriented Programming.
API	Application Programming Interface.
ATL	Atlas Transformation Language.
BDD	Binary Decision Diagram.
CASE	Computer-Aided Software Engineering.
CD	Continuous Delivery.
CI	Continuous Integration.
CI/CD	Continuous Integration / Continuous Delivery.
CIM	Computation Independent Model.
CLI	Command Line Interface.
CLR	Common Language Runtime.
CNF	Conjunctive Normal Form.
CORBA	Common Object Request Broker Architecture.
CSV	Comma-Separated Values.
CVL	Common Variability Language.
DDD	Domain Driven Design.
DLL	Dynamic-Link Library.
DOM	Document Object Model.
DRY	Don't Repeat Yourself.
DSL	Domain Specific Language.
DTO	Data Transfer Object.
DVCS	Distributed Version Control Systems.
EMF	Eclipse Modeling Framework.
FM	Feature Models.
FODA	Feature-Oriented Domain Analysis.
FOP	Feature-Oriented Programming.
FOSS	Free and Open Source Software.
GCC	GNU Compiler Collection.
GME	Generic Modeling Environment.
GMF	Graphical Modeling Framework.
GoF	Gang of Four.
GP	Generative Programming.
GPML	General Purpose Modeling Language.
GUI	Graphical User Interface.
HCI	Human-Computer Interaction.

HTM	Hierarchical Temporal Memory.
HTML	HyperText Markup Language.
HTTP	Hypertext Transfer Protocol.
IaC	Infrastructure-as-Code.
IDE	Integrated Development Environment.
IDL	Interface Description Language.
IPC	Implicit Presence Conditions.
J2EE	Java Platform Enterprise Edition.
JAR	Java Archive.
JSON	JavaScript Object Notation.
JVM	Java Virtual Machine.
LDL	Layer Definition Language.
LM	Logical Model.
LMM	Logical Metamodel.
LOC	Lines of Code.
LPS	Logical-Physical Space.
LSP	Language Server Protocol.
M2C	Model-to-Code.
M2M	Model-to-Model.
M2P	Model-to-Platform.
M2T	Model-to-Text.
MASD	Model Assisted Software Development.
MBE	Model Based Engineering.
MBSE	Model-based Systems Engineering.
MDA	Model Driven Architecture.
MDD	Model Driven Development.
MDE	Model Driven Engineering.
MDSD	Model Driven Software Development.
MDSE	Model Driven Software Engineering.
MIC	Model-Integrated Computing.
ML	Machine Learning.
MMM	MASD Meta-Model.
MOF	Meta-Object Facility.
MOP	Model Oriented Programming.
MRI	MASD Reference Implementation.
MSS	MASD SDM Specification.
MT	Model Transformation.
MTRR	Mean Time To Repair.
MTS	MASD Technical Space.
MWE	Minimum Working Example.
OCL	Object Constraint Language.
OMG	Object Management Group.
OO	Object-Oriented.
OOP	Object-Oriented Programming.
ORM	Object Relational Mappings.
OS	Operative System.
OVM	Orthogonal Variability Modeling.

PaC	Policy-as-Code.
PCL	Product Configuration Language.
PDF	Portable Document Format.
PDM	Platform Description Model.
PIM	Platform Independent Model.
PLE	Product Line Engineering.
PM	Physical Model.
PMM	Physical Metamodel.
POSIX	Portable Operating System Interface.
PSM	Platform Specific Model.
QVT	Query / View/ Transformation.
RDBMS	Relational Database Management System.
REST	Representational State Transfer.
RTE	Round-Trip Engineering.
SAX	Simple API for XML.
SDLC	Software Development Lifecycle.
SDM	Software Development Methodology.
SOA	Service Oriented Architecture.
SPLE	Software Product Line Engineering.
SRPP	Schematic and Repetitive Physical Pattern.
SRT	Schematic and Repetitive Text.
SWIG	Simplified Wrapper and Interface Generator.
T2M	Text-to-Model.
T2T	Text-to-Text.
T4	Text Template Transformation Toolkit.
TDD	Test Driven Development.
TS	Technical Space.
UI	User Interface.
UML	Unified Modeling Language.
UUID	Universally unique identifier.
VCS	Version Control System.
VM	Variability model.
VMM	Variability Metamodel.
XMI	XML Metadata Interchange.
XML	Extensible Markup Language.
XSD	XML Schema Definition Language.

Part I

MOTIVATION

1 INTRODUCTION

Computer science until now has mainly focused on the building of special kinds of models (mainly executable models written in programming languages). It may be time to extend the variety of languages we are using for building software models and to understand more precisely what are the exact sources of the models we are building.

— Bézivin (Bézivin, 2005)

Model Driven Engineering (MDE) is an approach to the creation and evolution of software systems centred on models and their transformations, and is used in industry as well as research (Mohagheghi and Dehlen, 2008; Paige and Varró, 2012; Andolfato et al., 2014). MDE promotes the use of high-level abstractions, and the cascading refinement of those abstractions towards the realisation of a target system. MDE is characterised by an increased focus on automation, only possible because abstractions are described by formal means. MDE is believed to have many benefits, including enhanced productivity and defect reduction (Torchiano et al., 2012), with empirical studies producing varying degrees of supporting evidence for these claims (Hutchinson et al., 2011; Andolfato et al., 2014; Shirtz, Kazakov, and Shaham-Gafni, 2007).

MDE's promises

Nevertheless, as we have elsewhere argued (Craveiro, 2021c) (Chapter 2), MDE itself is not a formal development process. Rather, it is a body of knowledge composed of a broad number of concepts, notations, tools, processes and rules. MDE is akin to a large and diverse toolbox from whence experienced practitioners take out tooling most suitable for the task at hand, with some tools requiring significant effort in assembly and training (*cf.* Section 2.3). On one hand, this approach provides ultimate flexibility, allowing MDE to be applied to almost any software engineering project, industrial or academic. On the other hand, the learning curve is steep and pitfalls abound. The MDE literature is rich in experience reports, adoption and meta-adoption studies, with ample supporting evidence for both sides of the argument (*cf.* Chapter 3). What is clear is that the present state of affairs is challenging for those seeking to dive into the deep and turbulent MDE waters for their first time.

MDE's dichotomy

This dissertation presents a novel MDE-based methodology to aid in the qdevelopment of software systems called *Model Assisted Software Development* (*MASD*). MASD is designed to address a well-defined subset of the issues raised by MDE adoption literature, gathered in the form of requirements (*cf.* Chapter 4). MASD trades MDE's flexibility and expressiveness for a reduction in complexity and, consequently, has a restricted but better defined range of applications. MASD targets all software developers, but focuses particularly

MASD

on those with limited or no knowledge of MDE, aiming to act as a bridge between the traditional and model-driven approaches.

Chapter overview

The remainder of this work describes the motivation for MASD, defines the methodology in detail and provides case studies of its application. The next sections delineate the intended audience for the dissertation (Section 1.1), the conventions employed within (Section 1.2), the research questions that guide the work (Section 1.3) and its main contributions (Section 1.4). The chapter concludes with Section 1.5, summarising the manuscript's organisation.

1.1 AUDIENCE

As well as introducing a new software development methodology, this dissertation spans a broad spectrum of topics related to modeling, making it relevant to a varied audience.

Target audience

- **MDE Researchers**: This work offers several items of interest to MDE Researchers, including a critical review of MDE adoption literature, and the analysis and application of several state of the art MDE techniques to address MASD requirements. Most significantly, MASD's overall approach offers new directions for MDE research, suggesting that a shift in focus towards usability and simplification may prove fruitful in addressing gaps identified by the literature thus far.

- **MDE Practitioners**: For those evaluating the use of MDE, the present work offers a valuable summary of the potential pitfalls of MDE application, as well as an alternative methodology that can serve as a starting point to explore the role of modeling in the development process. In cases where MASD does not prove to be a good fit, it may still provide useful insights when creating tailored processes and tooling. For those already using MDE, there may be value in integrating MASD's reference implementation with existing MDE tooling, in order to take advantage of its functionality.

- **Software Developers**: For those looking to increase automation on their projects, this dissertation presents a methodology to do so based on modeling, with an open source reference implementation ready to be used in industrial or research projects. Those only interested in application can skim Part ii to sketch out the methodology and focus mainly on the case studies in Part iii. On the other hand, developers who wish to start exploring the modeling domain will find that this work provides a compact but sufficient overview of MDE (*cf.* Chapter 2), when augmented with the supplementary material (Craveiro, 2021c).

1.2 CONVENTIONS

The following conventions are used in this book:

- the editorial *we* is employed throughout, even though all of the material here presented was written by the singular author of the dissertation.

- `constant width` font is used to denote source code listings or constructs defined within source code. For example "`object`" refers to a programming entity whereas "object" is a reference to a natural language concept.

1.3 RESEARCH QUESTIONS

Our research questions are as follows:

- What are the core factors in MDE theory and practice that: a) are acting as barriers to entry to software engineers unfamiliar with model driven practices; or b) are impeding the further progression of new MDE practitioners?

Once these factors have been identified, we will investigate and develop an approach to address them and supply empirical evidence of its successful application.

1.4 CONTRIBUTIONS

The primary contribution of this thesis is the MASD methodology itself. The secondary contributions are as follows:

- A metamodel for software development elements, targeting the code generation of well-defined software artefacts, with support for product lines and generative software architectures.

- A feature model for software artefacts that integrates with the metamodel to provide variability handling.

- The integration of literate modeling with the modeling process, allowing for the creation of expressive models which can target both source code as well as reference documentation.

- A fully-functional reference implementation for MASD which can be used in industrial and research projects.

1.5 ORGANISATION

What follows is a brief summary of the parts and chapters that make up this dissertation.

- **Part i: Motivation and Fundamentals** — Gathers the different strands that motivated this work, and summarises the fundamental concepts required to contextualise it. It is composed of the following chapters:
  - **Chapter 1: Introduction** — The present chapter; introduces the dissertation, the intended audience and its organisation.
  - **Chapter 2: State of the Art in Code Generation** — Underscores the systemic importance of source code to the modern software engineering discipline. Reviews historical approaches to code-generation, and introduces MDE as its state-of-the-art approach. Investigates two MDE variants which are key for the present work.
  - **Chapter 3: The State of MDE Adoption**: Performs a critical review of MDE adoption research, highlighting key themes emerging from its application.
  - **Chapter 4: Requirements** — Outlines the requirements for an approach that addresses a well-defined subset of the issues identified in Chapters 2 and 3, and in keeping with the Research Questions defined in Section 1.3.
- **Part ii: Methodology and Components** — Describes MASD in detail. It is composed of the following chapters:
  - **Chapter 5: The MASD Methodology** — Overview of the SDM, introducing all of its core concepts and describing how they relate to each other.
  - **Chapter 6: Domain Architecture** — Defines the MASD domain architecture in detail, describing its purpose and elements, as well as their roles and relationships.
- **Part iii: Application** — Contains case studies of MASD application. It is composed of the following chapters:
  - **Chapter 7: Literate Modeling with org-model** — Describes the introduction of org-model to MASD, adding a new codec with support for literate modeling.
  - **Chapter 8: MASD Reference Implementation** — Describes the reference implementation for MASD in detail, including all of its constituent products: the code generator and the two reference products.
- **Part iv: Outlook** — Places this thesis in a broader context. It is composed of the single chapter:
  - **Chapter 9: Conclusions** — Summarises the work, discusses key points that emerged from its development and outlines areas for future research.

2 STATE OF THE ART IN CODE GENERATION

Future tools will attack the more general problem of automatic code generation. Automatic programming is a difficult problem and it is still largely considered a research topic. Still, each new tool makes small innovations in this area, and eventually, code generation will become commonplace.

— Alan S. Fisher (Fisher and Fisher, 1988) (p. 30)

MDE and MASD

The trade-offs made by Model Assisted Software Development (MASD) can only be understood once the frame of reference of its parent approach, MDE, has been established and positioned against other techniques for the automatic development of software systems. The present chapter addresses this need by performing a state of the art review, centred on material of particular relevance to the methodology put forward by this dissertation.

Chapter overview

The chapter's structure mirrors our own trajectory across the MDE landscape. Section 2.1 starts by making a broad case for code generation within the modern software engineering environment. Section 2.2 outlines a brief description of earlier approaches, with the aim of establishing an historical context. The remainder of the chapter is dedicated to a detailed exposition of what we consider to be the modern approach to code generation: Model Driven Engineering (Section 2.3).

Our first step in this journey is then to establish the relevance of code generation to the present engineering moment by looking at the rise of code itself.

2.1 THE IMPORTANCE OF CODE GENERATION

Software's unstoppable growth

Software systems experienced immense growth in size over the past fifty years. In 1970, Lion's commented version of the UNIX Operative System had around 10 thousand Lines of Code (LOC) (Lions, 1996) and would fit comfortably in a book of less than 300 pages.[1] By 2011, Feitelson and Frachtenberg reported that Facebook's 7-year-old codebase had 8 million LOC (Feitelson, Frachtenberg, and Beck, 2013). Five years later, Potvin and Levenberg would nonchalantly tell us that "[Google's source code] repository contains 86TBa of data, including approximately two billion lines of code in nine million unique source files." (Potvin and Levenberg, 2016) As it is with size, so it is with scope; these vast

1 Much can be said about metrics used to describe the size of a software product. Whilst aware of LOC's limitations — Jones went as far as calling it "one of the most imprecise metrics ever used in scientific or engineering writing" (Jones, 1994) — we settled on this simplistic measure because the point under consideration is unaffected by its deficiencies.

software systems are now so completely pervasive that Andreessen was led to conclude that "software is eating the world." (Andreessen, 2011)

Modern development processes

Software development processes evolved in tandem with the new reality. Modern software engineering involves multidisciplinary teams with fluid roles, and rigid views on product development have been replaced with flexible approaches emphasising problem solving — hallmarks of agile thinking (Beck et al., 2001). Integration, testing and deployment activities, once considered distinct from the development activity itself, have now all but been fused into a contiguous delivery stage by the DevOps movement and enshrined in Continuous Integration / Continuous Delivery (CI/CD) pipelines (Bou Ghantous and Gill, 2017; Sánchez-Gordón and Colomo-Palacios, 2018). Some, such as Ameller *et al.*, envision (*emphasis ours*):

Continuous software engineering

> [...] *continuous software engineering* going one step further by establishing strong connections between software engineering activities. The objective of these connections is to *accelerate* and *increase the efficiency* of the software engineering process. (Ameller et al., 2017)

Automation and code

The jury may still be out on Ameller's all-encompassing vision, but what is already beyond doubt is the drive to remove the human element from any and all activity amenable to automation. Here one finds a less obvious corollary to Andreessen's insightful observation: software is also eating software engineering too, for automation in this context is often synonymous with substituting resources — human or otherwise — with more code.[2] The DevOps movement pushed forward a series of "X-as-code" initiatives, arguably the most noticeable of which is Infrastructure-as-Code (IaC) — the management of hardware infrastructure by programmatic means (Morris, 2016) — along with others, perhaps less visible but of import still, such as Policy-as-Code (PaC) — which aims to "support separation of concerns, allow security decisions to be separated from infrastructure and application logic, and make it possible to unify security controls." (Herardian, Marshall, and Prendergast, n.d.) What DevOps has shown, in our opinion, is that anything that can, will become code because automation's value-add acts as a powerful forcing function.[3]

X-as-code

Advantages of code

Automation may not be the only factor at play here, either. Where possible, software engineers prefer using code in the software development process over other kinds of artefacts, in no small part because its properties are now thought to be well understood.[4] Large tooling ecosystems have been built to manage all aspects of its lifecycle, including diffing tools, Distributed Version

2 That may change in the future as Machine Learning (ML) becomes more entrenched.

3 In (Beyer et al., 2016) (p. 67), Murphy makes a recent case for the value of automation by associating it with the following characteristics: a) *Scale*: automated systems can be designed to scale up and down very quickly, in response to external stimuli. b) *Consistency*: actions performed by machines yield results that are consistent over time. c) *Platform creation*: when designed adequately, the automated system becomes a platform upon which one can build and even leverage for other purposes. c) *Faster repairs*: Murphy alleges that the result of regular and successful automation is a reduced Mean Time To Repair (MTRR). d) *Faster action*: humans usually are unable to react as fast as machines, particularly in cases where the response is well-defined such as fail-over. e) *Time saving*: whilst difficult to calculate in practice, the most often cited benefit for automation is the freeing up of human resources to perform other tasks which cannot be so readily taken by machines.

4 Take its plain-text nature. Raymond dedicates an entire section in his opus (Raymond, 2003) to "The Importance of Being Textual" (Section 5.1), where he makes an impassioned defense of textual representations over binary formats. His thesis could be summed up with the following passage: "Text streams are a valuable universal format because they're easy for human beings

Control Systems (DVCS), text editors, IDEs and the like. Conversely, tools that do not support this paradigm seem to fall out of favour in the developer community. Badreddin *et al.* tell us about graphical modeling, where (*emphasis ours*) "[...] there is evidence that the adoption of visual modeling in software engineering remains low. The open source community remains *almost entirely code centric*."[5] (Badreddin, Forward, and Lethbridge, 2012)

If *X-as-code* — for all possible values of *X* — is the direction of travel, then its limit will surely be the tautological *code-as-code*; that is, the creation of code by automated means, or code generation. Given the rationale presented thus far for automation, one would naively expect code generation's role to increase hand-in-hand with the general growth of codebases, for, in a definite sense, it is the final frontier in the struggle for automation. It is the objective of the present work to shed light on the role of code generation on the modern software development process; to understand the limitations in its use; and, ultimately, to put forward an approach that addresses some of the identified weaknesses. As we shall see next, code generation has historically been an important component of the software engineer's toolkit.

Code-as-code, or code generation

2.2 HISTORICAL APPROACHES TO CODE GENERATION

The generation of source code by programmatic means, in the sense of automatic programming or programme synthesis[6], has had a long history within Computer Science. Whilst many avenues have been explored and are worthy of examination, in the interest of space we narrowed our scrutiny down to two approaches: Computer-Aided Software Engineering (CASE) and Generative Programming (GP). These were chosen both because of their influence on the present work, on MDE itself and on related methodologies — *e.g.*, (Jörges, 2013). Let us start our brief excursion by considering the first of the pair.

Automatic programming

2.2.1 *Computer Aided Software Engineering (CASE)*

The literature on CASE is expansive, and yet, in our opinion, it still fails to deliver an authoritative definition of the term. This is perhaps no more clearly illustrated than via Fisher's compendium (Fisher and Fisher, 1988), where he first calls it a "nebulous term" and then dispenses not just one but two definitions, both of which rather broad in scope (*emphasis ours*):

to read, write, and edit without specialized tools. These formats are (or can be designed to be) transparent." (Raymond, 2003) (p. 107)

5 The battle for textual representations has been fought in many fronts; from our perspective, the use of graphical versus textual notations in modeling is of particular significance. Whilst results remain far from conclusive, the anecdotal evidence in the literature does seem to tilt in favour of textual notations, at least for some types of activities (Meliá et al., 2016; Petre, 1995).

6 Biermann defines it as follows (Biermann, 1985) (*emphasis his*):

> *Computer programming* is the process of translating a variety of vague and fragmentary pieces of information about a task into an efficient machine executable program for doingthat task. *Automatic computer programming* or *automatic programming* occurs whenever a machine aids in this process.

Operational definition

> One definition of computer-aided software engineering is the use of tools that provide leverage *at any point in the software development cycle*. [...] A more restrictive but operationally better definition for computer-aided software engineering is the use of tools that provide leverage in the software *requirements analysis* and *design specification phases*, as well as those tools which *generate code automatically from the software design specification*. (Fisher and Fisher, 1988) (p. 6)

Wide domain boundaries

Whatever its precise meaning, what most definitions have in common is the sketching of a wide domain boundary for CASE systems — supporting the full range of activities in the software engineering lifecycle — as well as providing automated "methods of designing, documenting and development of the structured computer code in the desired programming language." (Berdonosov and Redkolis, 2011) CASE was, by any measure, an extremely ambitious programme, with lofty if laudable goals — as Fisher goes on to explain (*emphasis ours*):

> The ultimate goal of CASE technology is to *separate design from implementation*. Generally, the more detached the design process is from the actual code generation, the better. (Fisher and Fisher, 1988) (p. 5)

CASE's lessons

Though many of its ideas live on, perhaps unsurprisingly, the CASE programme as a whole did not take hold within the broad software engineering community — or, at least, not in the way most of those involved envisioned.[7] Understanding why it was so imparts instructive lessons, particularly if you have an interest in automatic programming as does the present work. With similar thoughts in mind, Jörges (Jörges, 2013) (p. 19) combed through the literature and uncovered a number of reasons which we shall now revisit, as well as supplementing them with two of our own towards the end.

Code generation inadequacies

1. **Deficiencies of translation into source code**. Though instituting an incredibly diverse ecosystem[8], most CASE tooling emphasised black-box transformation of graphical modeling languages into source code. There was a belief that automatic code generation for entire systems was looming in the horizon — with Fisher going as far as prophesying the emergence of "a software development environment so powerful and robust that we simply input the application's requirements specification, push a magic button, and out comes the implemented code, ready for release to the end user community." (Fisher and Fisher, 1988) (p. 283) Perhaps due to this line of reasoning, generated code was often not designed to be modified, nor were systems built to support user defined code generators — both of which were required in practice. The resulting solutions were convoluted and difficult to maintain.

2. **Vendor lock-in**. CASE predates the era of widely available Free and Open Source Software (FOSS), and therefore there was a predominance

7 Fisher outlines a compelling vision of that promised future in Chapter 17, "Technological Trends in CASE" (Fisher and Fisher, 1988) (p.281).

8 In its heyday, the CASE tooling market counted with hundreds of products. Fisher lists 35 tool vendors dedicated solely to design and analysis specification tools (Fisher and Fisher, 1988) (Appendix A).

of proprietary software. As with vendor lock-in in general, it was not in the vendor's best interest to facilitate interoperability since, by doing so, it would inadvertently help customers migrate to a competitor's product. As a result, reusability and interoperability were severely hampered. In addition, given the proliferation of small and mid-sized vendors, there was a real difficulty in choosing the appropriate tool for the job — the consequences of which could only be judged long after purchase.

3. **Lack of support for collaborative development**. Given the state of technology in the era when these tools were designed, it is understandable they did not adequately support the complex collaborative use cases that are required to facilitate the software engineering process. However, some of the difficulties transcended technology and were exacerbated by the vendor lock-in mentioned above; having data silos within each application — with narrow interfaces for data injection and extraction — meant it was difficult to supplement application workflows with external tooling.

Technological limitations

4. **Limitations in graphical modeling languages**. The generic nature of contemporary graphical modeling languages meant they were found wanting on a large number of use cases; oftentimes they were "too generic and too static to be applicable in a wide variety of domains" (Jörges, 2013) (p. 20).

5. **Unfocused and overambitious vision**. As already hinted by the lack of a formal definition, in our opinion CASE is a great example of a movement within software engineering that proposes an overly ambitious agenda, and one in which outcomes are extremely difficult to measure, either quantitatively or qualitatively. Due to this, it is hard to determine success and failure, and harder still to discern how its different components are fairing. In such a scenario, there is the risk of stating that "CASE failed" when, in reality, some of its key components may have been salvaged, modified and repackaged into other approaches such as MDE.

Intrinsic limitations

6. **Emphasis on full code generation**. Subjacent to CASE's goals is the notion that one of the main impediments to full code generation of software systems is a formal language of requirements which is fit for purpose.[9] However, in hindsight, it is now clear that attaining such a general purpose formal specification language is an incredibly ambitious target.

More certainly can be said on the subject of CASE's shortcomings, but, in our opinion, those six findings capture the brunt of the criticism. These lessons are important because, as we shall see (*cf.* Section 2.3), MDE builds upon much of what was learned from CASE — even if it does not overcome all of its stated problems. However, before we can turn in that direction, we must first complete our historical review by summarising an approach with similar ambitions in the field of automatic programming.

9 Fisher admits that "the mechanisms for automatically translating a requirement's specification [...] still lack rigorous definitions. The inherent problem is the diversity and the imprecision of the present specification techniques." (Fisher and Fisher, 1988) (p. 287).

2.2.2 *Generative Programming*

In sharp contrast with CASE's lighter approach to theory, Czarnecki's influential doctoral dissertation (Czarnecki, 1998) extended his prior academic work — standing thus on firmer theoretical grounds. In it, he puts forward the concept of *Generative Programming* (*GP*). GP focuses on (*emphasis ours*):

Definition

> [...] designing and implementing *software modules* which can be *combined* to generate *specialized and highly optimized systems* fulfilling *specific requirements*. The goals are to (a) decrease the conceptual gap between program code and domain concepts (known as achieving *high intentionality*), (b) achieve *high reusability* and adaptability, (c) simplify managing *many variants* of a component, and (d) increase efficiency (both in space and execution time). (Czarnecki, 1998) (p. 7)

Of particular significance is the emphasis placed by GP on software families rather than on individual software products, as Czarnecki *et al.* elsewhere explain (*emphasis ours*):

Software families

> Generative Programming [...] is about *modeling families of software systems* by software entities such that, given a particular *requirements specification*, a highly customized and optimized *instance of that family* can be *automatically manufactured* on demand from elementary, reusable implementation components by means of configuration knowledge [...]. (Czarnecki et al., 2000)

Conceptual model

In sharp contrast with CASE (*cf.* Section 2.2.1), GP provides a well-defined conceptual model, populated by a small number of core concepts which we shall now enumerate. At the centre lies the generative domain model, responsible for characterising the problem space (*i.e.*, the domain of the problem), the solution space (*i.e.* the set of implementation components) as well as providing the mapping between entities in these spaces by means of configuration knowledge.[10] Though GP does not dictate specific technological choices at any of the levels of its stack, Feature Models (Czarnecki, Helsen, and Eisenecker, 2005) are often used as a means to capture relevant features of the problem domain, as well as relationships amongst them. Concrete software systems are obtained by specifying valid configurations, as dictated by the modeled features, and by feeding the configuration knowledge to a generator, which maps the requested configuration to the corresponding implementation components.

As with CASE, GP did not come to dominate industrial software engineering[11], but many of the ideas it championed have lived on and are now central to MDE; the remainder of this chapter will cover some of these topics, with

10 Problem space and solution space are key concepts within MDE and MASD — the latter more so than the former. In (Craveiro, 2021c), Chapter 4 is dedicated to their exposition.

11 Rompf *et al.*'s recent assessment of GP had an unmistakably dour tone (*emphasis ours*): "While the general idea of program generation is already well understood and many languages provide facilities to generate and execute code at runtime [...], *generative programming remains somewhat esoteric — a black art*, *accessible only to the most skilled* and *daring* of programmers." (Rompf et al., 2015)

others described elsewhere (Craveiro, 2021c) (Chapters 4 and 6 in particular). In our opinion, GP also benefited from cross-pollination with CASE — for example, by attempting to address some of its most obvious shortcomings such as de-emphasising specific technological choices and vendor products, and focusing instead on identifying the general elements of the approach. Alas, one downside of generalisation is the difficulty it introduces in evaluating the approach in isolation; many of the factors that determine the success or failure of its application are tightly woven with the circumstances and choices made by actors within a given instance of the software engineering process — a ghost that will return to haunt MDE (*cf.* Chapter 3). And it is to MDE which we shall turn to next.

GP in a wider context

2.3 model driven engineering (mde)

MDE is the final and most consequential stop on our quest to characterise the state of the art in automatic programming. The present section briefly reviews the core theoretical foundations of the discipline (Section 2.3.1), and subsequently delves into the specifics of two MDE variants of particular significance to MASD: MDA (Section 2.3.2) and AC-MDSD (2.3.3). Let us begin, then, by attempting to answer the most pressing question of all.

Section overview

2.3.1 *What is Model-Driven Engineering*

Though the academic press has no shortage of literature on MDE[12], it is largely consensual when it comes to its broad characterisation. Cuestas, for example, states the following (*emphasis ours*):

> [Within MDE, software] development processes are conceived as *a series of steps* in which *specification models*, as well as those which describe the problem domain, are *continually refined*, until implementation domain models are reached — along with those which make up the verification and validation of each model, and the correspondence between them. In [MDE], the steps in the development process are considered to be *mere transformations between models*.[13] (Cuesta, 2016) (p. 1)

Informal characterisation

However, as we argued previously at length (Craveiro, 2021c), this apparent consensus is somewhat misleading, and a characterisation of the fundamental nature of MDE is not as easy as it might appear at first brush.[14] In the

12 The choices are varied, be it in the form of detailed assessments such as Völter's (Völter et al., 2013), syntheses of the kind put forward by Brambilla (Brambilla, Cabot, and Wimmer, 2012), or state of the art reviews in the vein of Oliveira's (OLIVEIRA, 2011) and Jörges' (Jörges, 2013).

13 This quote was translated from the original Spanish by the author. The reader is advised to consult the primary source.

14 A deeper questioning of the nature of MDE was performed by the author in (Craveiro, 2021c) (Chapter 2). More generally, MASD makes use of a much wider subset of MDE's theoretical underpinnings than it feasible to discuss in detail within the present manuscript, so its exposition was relegated to supplemental material (Craveiro, 2021c). Whilst these notes are extremely relevant to MASD, its absence on the primary material does not weaken the main argument of the

afore-cited manuscript, we concluded that the MDE nature is instead better summarised as follows (page 13, *emphasis ours*):

Discipline characterisation

- MDE is an *informal body of knowledge* centred on the employment of modeling as the principal driver of software engineering activities.
- MDE promotes the pragmatic application of a family of related approaches to the development of software systems, with the intent of generating automatically a part or the totality of a software product, from one or more formal models and associated transformations.
- MDE is best understood as a vision rather than a concrete destination. A vision guides the general direction of the approach, but does not dictate the solution, nor does it outline the series of steps required to reach it.
- It is the responsibility of the MDE practitioner to select the appropriate tools and techniques from the MDE body of knowledge, in order to apply it adequately to a specific instance of the software development process. By doing so, the practitioner will create — implicitly or explicitly — an MDE variant.

Importance of Variants

The onus is thus on specific MDE variants, rather than on the body of knowledge itself, to lay down the details of how the model-driven approach is to be carried out. In this light, we have chosen to focus on two MDE variants, in order to better grasp the detail. The first is MDA, chosen not only due to its historical significance — in our opinion, it remains the most faithful embodiment of MDE's original spirit and vision — but also because it serves as an exemplary framework to demonstrate the application of MDE concepts. The second variant is AC-MDSD, which was selected because of its importance for MASD (*cf.* Chapter 5). As we shall see, these two variants are also of interest because they put forward contrasting approaches to MDE. Let us begin then by looking at the first of the pair.

2.3.2 *Model Driven Architecture (MDA)*

Characterisation, specifications

MDA is a comprehensive initiative from OMG and arguably the largest industry-wide effort to date, attempting to bring MDE practices to the wider software engineering community.[15] Based on the OMG set of specifications — which include UML (OMG, 2017b) as a modeling language, the Meta-Object Facility (MOF) (OMG, 2016) as a metametamodel and Query / View/ Transformation (QVT) (OMG, 2017a) as a transformation language — MDA is designed to

dissertation — thus justifying their exclusion. If, however, you are seeking a more comprehensive background, the notes are recommended reading.

15 It is difficult to overstate MDA's significance in shaping MDE. Brambilla *et al.* believe it is "currently the most known modeling framework in industry" (Brambilla, Cabot, and Wimmer, 2012) (p. 43); in Jörges assessment, it is "perhaps the most widely known MD* approach" (Jörges, 2013) (p. 23); Asadi and Ramsin attribute MDE's familiarity amongst software engineers to "the profound influence of the Model Driven Architecture (MDA)." (Asadi and Ramsin, 2008)

support all stages of software development lifecycle, from requirements gathering through to business modeling, as well as catering for implementation-level technologies such as CORBA (OMG, 2012).

Approach, goals

Though more concrete and circumscribed than MDE, MDA is still considered an approach rather than a methodology, in and of itself.[16] The approach's primary goals are "portability, interoperability, and reusability of software". (Group, 2014). Beyond these, the MDA Manifesto (Booch et al., 2004) identifies a set of basic tenets that articulate its vision and which serve complementary purposes (Figure 2.1). These are as follows:

Basic tenets

- **Direct representation.** There is a drive to shift the locus of software engineering away from technologists and the solution space, and place it instead in the hands of domain experts and on the problem space. The objective is to empower experts and to reduce the problem-implementation gap.[17]

- **Automation.** The aim is to mechanise all aspects of the development process that "do not depend on human ingenuity" (Booch et al., 2004). Automation is also crucial in addressing the problem-implementation gap because it is believed to greatly reduce interpretation errors.

- **Open standards.** By relying on open standards, MDA hoped to diminish or even eliminate Booch *et al.*'s "gratuitous diversity" (Booch et al., 2004) and to encourage the development of a tooling ecosystem designed around interoperability, with both general purpose tooling as well as specialised tools for niche purposes.[18]

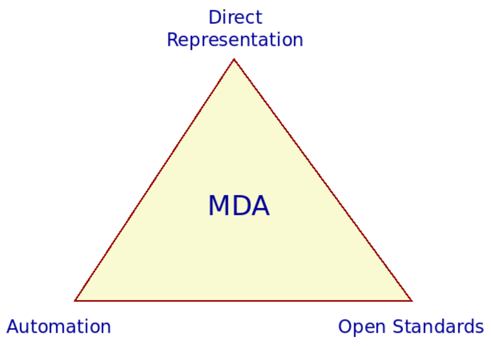


Figure 2.1: Basic tenets of the MDA. *Source*: Author's drawing based on an image from Booch *et al.* (Booch et al., 2004).

Terminology, viewpoints

Figure 2.2 presents a selection of MDA's basic terminology as per OMG documentation, which is, unsurprisingly, in line with the MDE terminology defined thus far — as well as that of the supplementary material (Craveiro, 2021c). This is to be expected given the central role of MDA in the early development of MDE itself.[19] A noteworthy term on that list is viewpoint, for MDA sees

16 For an analytical survey of MDA based methodologies, see Asadi and Ramsin (Asadi and Ramsin, 2008).

17 Problem space, solution space and problem-implementation gap are all described in detail on (Craveiro, 2021c) (Chapter 4).

18 Booch *et al.*'s adverse reaction to "gratuitous diversity" is best understood in the context of CASE (*cf.* Section 2.2.1).

19 According to Bézevin, "MDA may be defined as the realization of MDE principles around a set of OMG standards like MOF, XMI, OCL, UML, CWM, SPEM, *etc.*" (Bézivin, 2005)

systems modeling as an activity with distinct vantage points or perspectives. Viewpoints give rise to architectural layers at different levels of abstraction, each associated with its own kind of models:

TERM	DEFINITION
System	"[. . .] A collection of parts and relationships among these parts that may be organized to accomplish some purpose."
Model	"[. . .] Information selectively representing some aspect of a system based on a specific set of concerns. The model is related to the system by an explicit or implicit mapping."
Modeling Language	"[. . .] A model is said to conform to a modeling language. That is, everything that is said in some model is allowed to be said by the modeling language."
Architecture	"A set of models with the purpose of representing a system of interest." Also: "The activity and or practice of creating the set of models representing a system."
Platform	"[. . .] Set of resources on which a system is realized. This set of resources is used to implement or support the system."
Viewpoint	"[. . .] Specifies a reusable set of criteria for the construction, selection, and presentation of a portion of the information about a system, addressing particular stakeholder concerns."
View	"[. . .] A representation of a particular system that conforms to a viewpoint."
Transformation	"[. . .] Used to produce one representation from another, or to cross levels of abstraction or architectural layers."

Figure 2.2: Key MDA terms. *Source*: MDA Guide (Group, 2014).

Modeling levels

- **Computation Independent Models (CIMs)**: The "business or domain models" (Group, 2014). Describes business functionality only, including system requirements. CIMs are created by domain experts and serve as a bridge between these and software engineers.
- **Platform Independent Models (PIMs)**: The "logical system models" (Group, 2014). PIMs describe the technical aspects of a system that are not tied to a particular platform.[20]
- **Platform Specific Models (PSMs)**: The "implementation models" (Group, 2014). PSMs augment PIMs with details that are specific to a platform, and thus are very close to the implementation detail.

MDA standards and tooling

Figure 2.3 provides a simplified illustration of how the three types of models are related. Instance models are intended to be created using either UML — with appropriate extensions, as required, by means of UML Profiles — or via any other MOF based modeling language, preexisting or specifically created for the needs of the system. Model-to-Model (M2M) transforms can be handled by QVT — including cascading transformations from CIM to PIM and to PSM — or by any other MOF based Model Transformation (MT) language such as Atlas Transformation Language (ATL) (Jouault et al., 2008). Finally, a large ecosystem of code generation tools, frameworks and standards have evolved for MDA, such as MOFScript (Oldevik et al., 2005), MOFM2T (OMG,

20 The definition presented on Figure 2.2 gives a simplistic view of terms such as platform. To understand the difficulties surrounding this and other related terms, see Chapter 4 of (Craveiro, 2021c).

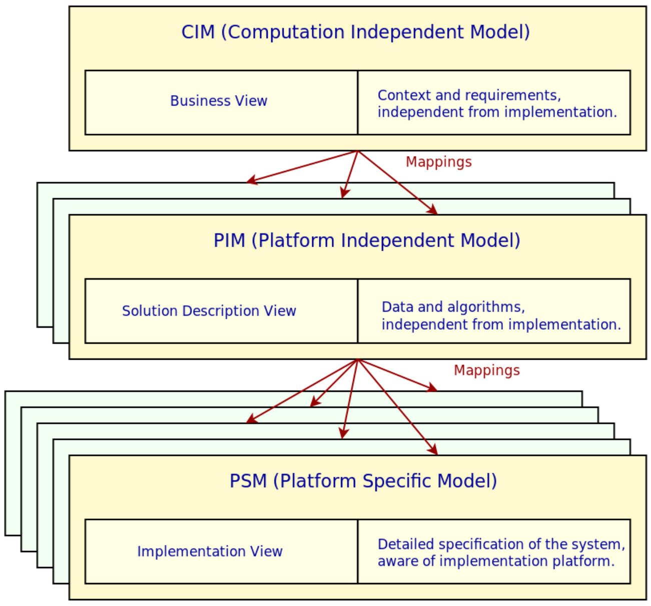

CIM (Computation Independent Model)
Business View
Context and requirements, independent from implementation.
Mappings
PIM (Platform Independent Model)
Solution Description View
Data and algorithms, independent from implementation.
Mappings
PSM (Platform Specific Model)
Implementation View
Detailed specification of the system, aware of implementation platform.

Figure 2.3: Modeling levels and mappings. *Source*: Author's drawing based on Brambilia *et al.*'s image (Brambilla, Cabot, and Wimmer, 2012) (p. 45).

2008) and, arguably most significantly of all, the Eclipse Modeling Framework (EMF) (Steinberg et al., 2008; Steinberg et al., 2009)[21] — all of which which facilitate the generation of code from PSMs.

Faced with a potentially large number of heterogeneous models, a requirement often emerges to weave them together to form a consistent overall picture.[22] Within MDA, this role is performed by model compilers. Whilst the literature does not readily supply a rigorous definition for the term, Mellor clearly delineates the role they are expected to play, as well as outlining their challenges (*emphasis ours*):

Weaving

> A model compiler takes a set of *executable UML models*[23] and *weaves* them together according to a *consistent set of rules*. This task involves executing the mapping functions between the various source and target models to produce a *single all-encompassing metamodel* [...] that the includes all the structure, behavior and logic — everything — in the system. [...] *Weaving the models together* at once addresses the problem of *architectural mismatch*, a term coined by David Garlan to refer to components that do not fit together without the addition of tubes and tubes of glue code, the *very problem MDA is intended to avoid!* A model compiler imposes a single architectural structure on the system as a whole. (Mellor, 2004)

Model compilers

21 The EMF is a modeling suite that is seen by some as a modern interpretation of the MDA ideals. Steinberg *et al.* called it "MDA on training wheels." (Steinberg et al., 2008) (p. 15)

22 For these and other challenges related to complex model topologies and model refinement, see (Craveiro, 2021c), Chapter 4.

23 While executable models are beyond the scope of the present dissertation, its worthwhile depicting its ambition. Mellor is once more of assistance (*emphasis ours*): "Executable models are neither sketches nor blueprints; as their name suggests, models run. [...] *Executable UML* is a profile of UML that defines an execution semantics for a carefully selected streamlined subset of UML." (Mellor, 2004)

Outside of executable models, model compilers are often associated with MDA code generation, transforming PIMs and PSMs directly into source code. The line between MDA's code generators and model compilers is blurry, both due to the imprecise terminology as well as the fact that code generators often need to conduct some form of model weaving prior to code generation. Several MDA code generators exist, including AndroMDA[24] and Jamda[25], and these typically allow for extensibility by means of plug-ins — the much maligned cartridges.[26]

Code generators, cartridges

And it is with cartridges that we round-off MDA's concepts relevant to MASD. Clearly, an overview as brief as the present cannot do justice to the breadth and depth of MDA. However, for all of its impressive achievements, MDA is not without its detractors. Part of the problem stems from the early overambitious claims, which, as we shall see in Chapter 3, were not entirely borne out by evidence.[27] In addition, UML itself has been a source of several criticisms, including sprawling complexity, a lack of formality in describing its semantics, too low a level of abstraction, and the difficulties in synchronising the various UML models needed to create a system.[28]

Overambitious claims, complexity

Certain challenges are wider than UML and pertain instead to OMG's stance towards standardisation. On one hand, open and detailed specifications undoubtedly facilitated MDA's adoption and helped create a large and diverse tooling ecosystem. On the other hand, they also abetted an heterogeneous environment with serious interoperability challenges, populated by large and complex standards that forced practitioners to have a deep technical knowledge in order to make use of them. As a result, numerous aspects of these standards are not fully utilised by practitioners, with many either ignoring them altogether or resorting to more trivial use cases. There is also a real danger of ossification, with some standards not seeing updates in years — partially because the processes for their development are drawn-out and convoluted.

Standardisation challenges

This state of affairs led Thomas to conclude that the best course of action is perhaps a lowering of expectations: "Used in moderation and where appropriate, UML and MDA code generators are useful tools, although not the panaceas that some would have us believe." (Thomas, 2004) Reading between the lines, one is led to conclude that at least part of MDA's problems stem from its

Focused use, managed expectations

24 http://andromda.sourceforge.net
25 http://jamda.sourceforge.net/
26 Völter's criticism of the term is scathing: "[...] Cartridges is a term that get (*sic.*) quite a bit of airplay, but it's not clear to me what it really is. A cartridge is generally described as a 'generator module', but how do you combine them? How do you define the interfaces of such modules? How do you handle the situation where to cartridges have implicit dependencies through the code they generate?" (Völter, 2009).
27 Slogans such as Bézivin's "Model once, Generate everywhere" (Bézivin, 2003) are examples of this optimism, as was the language of the MDA Manifesto itself (*emphasis ours*):

> We believe that MDA has the potential to *greatly reduce development time* and *greatly increase the suitability of applications*; it does so not by magic, but by providing mechanisms by which developers can capture their knowledge of the domain and the implementation technology more directly in a standardized form and by using this knowledge to produce automated tools that *eliminate much of the low-level work of development*. More importantly, MDA has the potential to *simplify the more challenging task of integrating existing applications* and data with new systems that are developed. (Booch et al., 2004)

28 For a brief but insightful overview of the lessons learned from UML, see France and Rumpe (France and Rumpe, 2007), Sections 5.1 ("Learning from the UML Experience: Managing Language Complexity") and 5.2 ("Learning from the UML Experience: Extending Modeling Languages").

vast scope. It is therefore interesting to compare and contrast it with the next variant under study, given it takes what could be construed as a diametrically opposed approach.

2.3.3 *Architecture-Centric MDSD (AC-MDSD)*

Definition, characterisation

Product of several years of field experience, Stahl *et al.* introduced Architecture-Centric MDSD (AC-MDSD) as a small part of their seminal work on Model Driven Software Development (MDSD) (Völter et al., 2013) (p. 21).[29] In their own words, "[...] AC-MDSD aims at increasing development efficiency, software quality, and reusability. This especially means relieving the software developer from tedious and error-prone routine work." Though it may be argued that the concepts around Model-to-Platform (M2P) transforms for infrastructural code were already well-established within MDE, having their roots in ideas such as MDA model compilers and MDA code generators (*cf.* Section 2.3.2), it is important to note that AC-MDSD has very few commonalities with MDA. It is a minimalist approach, specified only at a high-level of abstraction and inspired mainly by practical experimentation.

Narrow focus

AC-MDSD's very narrow focus makes it a suitable starting point for the exploration of model-driven approaches, as Stahl *et al.* go on to explain (*emphasis ours*):

> We recommend you first approach MDSD via architecture-centric MDSD, since this requires the smallest investment, while the effort of its introduction can pay off in the course of even a six-month project. Architecture-centric MDSD does not presuppose a functional/professional domain-specific platform, and *is basically limited to the generation of repetitive code* that is typically needed for use in commercial and Open Source frameworks or infrastructures (*sic.*). (Völter et al., 2013) (p. 369)

Source code categories

The core idea behind AC-MDSD emanates from Stahl *et al.*'s classification of source code into three categories:

- **Individual Code**: Code crafted specifically for a given application, and which cannot be generalised.
- **Generic Code**: Reusable code designed to be consumed by more than one system.
- **Schematic and Repetitive Code**: Also known as boilerplate or infrastructure code, its main purpose is to perform a coupling between infrastructure and the application, and to facilitate the development of the domain-specific code.

As represented diagrammatically in Figure 2.4, schematic and repetitive code can amount to a significant percentage of the total number of LOC in a given

29 For the purposes of this dissertation. MDSD is understood to be a synonym of MDE. See (Craveiro, 2021c), Section 2.4, for an explanation of the various names employed under the MDE umbrella.

system, with estimates ranging between 60% to 70% for web-based applications (Völter et al., 2013) (p. 369) and 90% or higher for embedded systems development (Czarnecki, 1998).[30,31] Besides the effort required in its creation, infrastructure code is also a likely source of defects because its repetitive nature forces developers to resort to error prone practices such as code cloning (Staron et al., 2015). Code generators do exist to alleviate some of the burden — such as wizards in IDEs and the like — but they are typically disconnected and localised to a tool, with no possibility of having an overarching view of the system. Thus, the goal of AC-MDSD is to provide an holistic, integrated and automated solution to the generation of infrastructural code.

Schematic and repetitive code

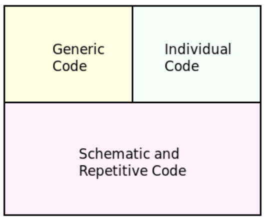


Figure 2.4: Categories of code in a system. *Source*: Author's drawing based on an image from Stahl *et al.* (Völter et al., 2013) (p. 15)

Following this line of reasoning, Stahl *et al.* theorise that systems whose software architecture has been clearly specified have an implementation with a strong component of schematic and repetitive programming; that is, the system's architecture manifests itself as patterns of infrastructural code, thus ultimately leading to the idea of generative software architectures. In a generative software architecture, the schemata of the architecture is abstracted as elements of a modeling language; domain models are created for an application, and, from these, code generators create the entire set of infrastructural code. In the simplest case, the modeling language can be created as a UML Profile with the required architectural concepts, and the instance models then become PIMs (*cf.* Section 2.3.2). For simplicity, Stahl *et al.* recommend bypassing explicit transformations into PSMs prior to code generation, and generate code directly from the PIM instead. Once a generative software architecture is put in place, only a small step is required to move towards the creation of product lines.

Generative software architectures

With regards to the construction of software systems, Stahl *et al.* propose a two-track iterative development model, where infrastructural engineering is expressly kept apart from application development, though allowing for periodic synchronisation points between the two — a useful take that is not without its dangers, as will be shown shortly. In addition, given only infrastructural code is targeted, AC-MDSD presupposes a need for integrating handcrafted code and generated code, with full code generation explicitly defined as a non-goal. The onus is on the MDE practitioner to determine the most suitable integration approach for the system in question, aided and

Two-track development

Partial generation

30 Stahl *et al.* may have discerned the general notion of schematic and repetitive code, but they left the gory details of their identification as an exercise for the modeler, noting only in passing that applications are composed of (*emphasis ours*) "[...] a *schematic part* that is not identical for all applications, but *possess the same systematics* (for example, based on the same design patters)." (Völter et al., 2013) (p. 16) Whilst most developers are likely in agreement with this sentiment, in truth it offers little additional clarity on how to identify those "same systematics".

31 Our own personal experiences (Craveiro, 2021b) corroborated these findings, both in terms of the existence of schematic and repetitive code as well as its relative size on a large industrial product.

abetted by the literature — for instance, by deploying the techniques such as those surveyed by Greifenberg *et al.* (Greifenberg et al., 2015a; Greifenberg et al., 2015b).

Absence of critiques in the literature

Experience reports of AC-MDSD usage in various contexts do exist, though they are by no means numerous and appear to lack a critical analysis of theory and application (Al Saad et al., 2008; Escott et al., 2011b; Escott et al., 2011a; Manset et al., 2006). The paucity may be attributable, at least in part, to researchers employing terminology other than Architecture-Centric MDSD, rather than to the principles espoused — the afore-cited evidence, anecdotal though it may be, does suggest a favouring of the overall approach by software engineers when they embark on MDE.[32] In the absence of authoritative points of view, we have chosen to undertake a critique from personal experience, rooted on our adoption of AC-MDSD on a large industrial project (Craveiro, 2021b). Whilst limited, and though it pre-empts the discussion on MDE adoption (*cf.* Chapter 3), there are nevertheless advantages to this take, since the principal difficulty with AC-MDSD lies on the specifics of its application rather than with the sparseness of the theoretical framework.

AC-MDSD's importance

We shall start by identifying the importance of AC-MDSD, which in our opinion is understated in the literature. As Stahl *et al.*'s quote above already hinted, infrastructural code is seen as low-hanging fruit for MDE because it is arguably the most obvious point to automate in the development of a software system. Carrying on from their analysis, our position is that the following interdependent factors contribute to this outcome:[33]

Infrastructural code and automation

- **Ubiquitous Nature**: Infrastructural code is prevalent in modern software systems, as these are composed of a large number of building blocks[34] which must be configured and orchestrated towards common architectural goals. It is therefore a significant problem, and its only increasing with the unrelentingly growth of software (*cf.* Section 2.1).

- **Deceptively Easy to State**: Unlike other applications of MDE, the issues addressed by AC-MDSD are easy to state in a manner comprehensible to all stakeholders. Existing systems have numerous exemplars that can serve as a basis for generalisation — employable simultaneously as a source of requirements, as well as a baseline for testing generated code. For new systems, engineers can manually craft a small reference implementation and use it as the target of the automation efforts, as did we, twice — (Craveiro, 2021b) Section 4.5, and Section 8.2 of the present document.

- **Deceptively Easy to Measure**: The costs associated with the manual creation and ongoing maintenance of infrastructural code are apparent both to software engineers as well as to the management structure, because they are trivially measurable — *i.e.* the total resource-hours spent creating or maintaining specific areas of the code base against the resource-hour cost is a suitable approximation. Engineers also know

32 A trait we ourselves share, including the lack of awareness of the existence of AC-MDSD, as narrated in (Craveiro, 2021b).

33 This analysis is largely a byproduct of the analysis work done in Sections 7 and 8 of (Craveiro, 2021b), as well as Chapter 5 of (Craveiro, 2021c).

34 *Building blocks* are to be understood in the sense meant by Völter (Völter et al., 2013) (p. 59). See also Section 4.2.2 of (Craveiro, 2021c) (p. 33).

precisely which code they intend to replace, because they must identify the schematic and repetitive code. As a corollary, simplistic measures of cost savings are also straightforward to impute.[35]

- **Deceptively Easy to Implement**: The creation of technical solutions to realise AC-MDSD are deceptively simple to implement, as there is an abundance of template-based code generation tools that integrate seamlessly with existing programming environments — *e.g.* Microsoft's T4 (Vogel, 2010) (p. 249), EMF's XText (Eysholdt and Behrens, 2010), *etc.*[36] These tools are supplied with a variety of examples and target software engineers with little to no knowledge of MDE.

- **Produces Results Quickly**: As already noted by Stahl *et al.*, limited efforts can produce noticeable results, particularly in short to medium timescales but, importantly, the full consequence of its limitations play out at much longer timescales.

Perhaps because of these factors, many localised AC-MDSD solutions have been created which solve non-trivial problems, meaning the approach undoubtedly works. However, in our experience, AC-MDSD has inherent challenges which we ascribe to the following interrelated reasons.

Consequences of unorthodoxy

Firstly, it exposes end-users to the complexities of the implementation and that of MDE theory, discouraging the unfamiliar. Paradoxically, it may also result in approaches that ignore MDE entirely — that which we termed unorthodox practitioners[37] in (Craveiro, 2021b) — and thus present inadequate solutions to problems that have already been addressed competently within the body of knowledge. This happens because its easy to get up-and-running with the user friendly tooling — that is, without any foundational knowledge — but soon the prototype becomes production code, and mistakes become set in stone; before long, a Rubicon is crossed beyond which there is just too much code depending on the generated code for radical changes to be feasible.

Premature de-investment

Secondly, the more automation is used, the higher the cost of each individual solution because customisation efforts require a non-negligible amount of specialised engineering work, and will need continual maintenance as the product matures. The latter is of particular worry because these costs are mostly hidden to stakeholders, who may have been led to believe that the investment in infrastructural code "had already been made", rather than seeing it as an ongoing concern throughout the life of a software system.

Short term gain, long term pain

The third problem arises as the interest on the technical debt (Cunningham, 1992) accrued by the first three factors comes due. Naive interpretations of AC-MDSD inadvertently trade velocity and simplicity in the short term for complexity and maintenance difficulties in the long term — at which point

35 The adjective simplistic is used here because we are performing a trivial extrapolation. It would be non-trivial to account for qualitative factors present in manual code, such as efficiency, robustness and many other non-quantitative properties in the domain of software quality, such as those identified by Meyer (Meyer, 1988) (Chapter 1). These simplistic measurements can only indicate that generated code is no worse functionally than its manual counterpart.

36 Here we include tools such as AndroMDA and Jamda (*cf.* Section 2.3.2) because, whilst typically associated with MDA, they can be deployed to fulfil an AC-MDSD role. In addition, for a more general treatment of these approaches, see (Craveiro, 2021c), Chapter 3 (Section 3.7).

37 An unorthodox practitioner is one who engages in independent rediscovery of fundamental aspects of the MDE body of knowledge without the awareness of its existence.

all deceptions are unmasked.[38] This temporal displacement means that when the consequences are ultimately felt, they are notoriously difficult to quantify and address; by that time, the system may be on a very different phase of its lifecycle (*i.e.* maintenance phase).

Silos

Fourthly, end-users are less inclined to share solutions because the commonalities between individual approaches at the infrastructural level may not be immediately obvious due to a lack of generalisation. Knowledge transfer is impaired, we argue, because practitioners and tool designers view *their* infrastructural code as inextricably linked to the particular problem domain they are addressing or to a specific tool, and thus each developer becomes siloed on an island of their own making. This notion is reinforced by MDE's vision of every developer as a competent MDE practitioner, able to deploy the body of knowledge to fit precisely its own circumstances, and further compounded by MDE's focus on the problem space rather than the solution space.[39] Conversely, aiming for generalisation is only possible once practitioners have mastered the MDE cannon, which takes time and experience. Thus, systems with similar needs may end up with their own costly solutions, having little to no reuse between them.

Interaction with MDE principles

Problem domain decoupling

Alas, generalisation is no silver bullet either, as attested by the fifth and final challenge: that of problem domain decoupling.[40] This issue emerges as the emphasis shifts from special purpose AC-MDSD solutions towards a general purpose approach, within a two-track development framework. With this shift, the relative scopes of the application domain versus the infrastructural domain also shift accordingly, and what begins as a quantitative change materialises itself as a qualitative change. Figure 2.5 models a simplified version of the dynamic in pictorial form, though perhaps implying a discreteness to the phenomena which is not necessarily present in practice.

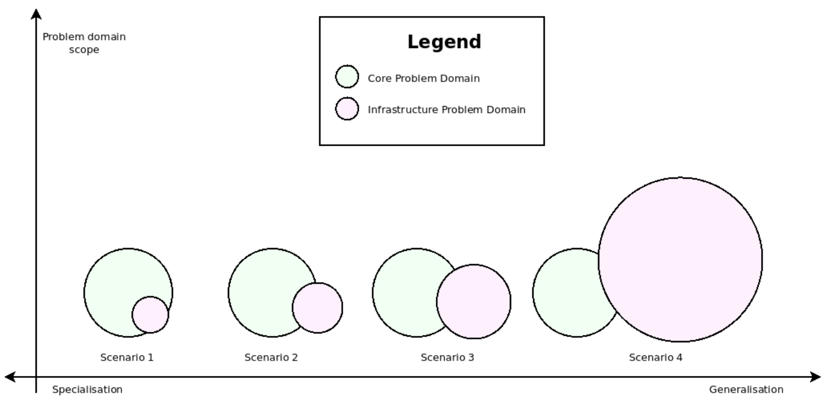


Figure 2.5: Problem domain decoupling. Source (Craveiro, 2021c) (p. 46)

38 For a reflection of our own experiences on the matter, see Section 6 of (Craveiro, 2021b).

39 It is perhaps for these reasons that MDA code generators put forward concepts such cartridges: so that their end-users can extend a core to match their particular requirements. These are, in effect, elaborate code generation frameworks to satisfy the needs of developers. Interestingly, Jörges concluded that "[...] there is a high demand for approaches that enable a simple and fast development of code generators." In our opinion, developers do not want to create code generators, but find themselves having to do so. As we'll see in Chapter 3, demand is largely a function of inadequate tooling.

40 Problem domain decoupling is addressed in pages 46 to 49 of (Craveiro, 2021c) (Chapter 5).

Though subtle at first, these changes are eventually felt for (*emphasis theirs*):

> [...] as the scope of the infrastructural domain grows, it becomes a software product in its own right. Thus, there is an attempt to simultaneously engineer *two tightly interlocked* software products, each already a non-trivial entity to start off with. At this juncture one may consider the ideal solution to be the use of vendor products as a way to insulate the problem domains. Unfortunately, experimental evidence emphatically says otherwise, revealing that isolation may be necessary *but only up to a point*, beyond which it starts to become detrimental. We name this problem *over-generalisation*.[41] (Craveiro, 2021c) (p. 47)

In other words, there is a fine balancing act to be performed between under and over generalisation, with regards to the infrastructural domain and the problem domain; finding the right balance is a non-trivial but yet essential exercise (*emphasis theirs*):

Barely general enough

> What is called for is a highly cooperative relationship between infrastructure developers and end-users, in order to foster feature suitability — a relationship which is not directly aligned with traditional customer-supplier roles; but one which must also maintain a clear separation of roles and responsibilities — not the strong point of relationships between internal teams within a single organisation, striving towards a fixed goal. Any proposed approach must therefore aim to establish an *adequate* level of generalisation by mediating between these actors and their diverse and often conflicting agendas. We named this generalisation sweet-spot *barely general enough*, following on from Ambler's footsteps (Ambler, 2007)[42], and created Figure 5.5 to place the dilemma in diagrammatic form. (Craveiro, 2021c) (p. 47)

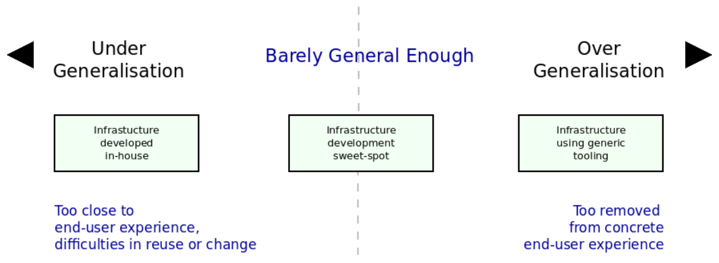

Under Generalisation
Barely General Enough
Over Generalisation
Infrastucture developed in-house
Infrastructure development sweet-spot
Infrastructure using generic tooling
Too close to end-user experience, difficulties in reuse or change
Too removed from concrete end-user experience

Figure 2.6: Different approaches to infrastructure development. Source (Craveiro, 2021c) (p. 48)

As we shall see (*cf.* Chapter 5), this quest for an approach targeting the barely general enough sweet-spot has greatly influenced the present work.

In summary, our opinion is that those very same attributes that make AC-MDSD amenable as a starting point for MDE exploration are also closely

41 Chapter 3 deal with the complex issues surrounding MDE adoption, including vendor tooling (Section 3.3 in particular).

42 Ambler states that (*emphasis ours*) "[...] if an artifact is just *barely good enough* then by definition it is at the most effective point that it could possibly be at." (Ambler, 2007)

associated with its most significant downsides. Interestingly, this double-edged sword characteristic is not unique to AC-MDSD, but instead generalises well to MDE — as the next chapter will describe in detail.

3 THE STATE OF MDE ADOPTION

But, as we look to the horizon of a decade hence we see no silver bullet. There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity.

— Brooks (Brooks, 1974) (p. 181)

You cannot reduce the complexity of your problem by increasing the complexity of your language.

— Wirth (*attributed*)

One of the most critical aspects of any new technological approach is the evaluation of its performance on the field — that is, an evidence-based analysis of the impact of its application under real-world conditions. It is particularly important to those embarking on the design of a new software development methodology, itself based on model-driven principles, to understand MDE's benefits and drawbacks from a practical standpoint. *Motivation* This chapter combines a review of MDE adoption literature with a limited amount of new research, highlighting lessons learned and barriers to entry, with the objective of clarifying MDE's current state of practice. As with theory (*cf.* Chapter 2), these findings will be used as input to MASD's requirements gathering process (*cf.* Chapter 4).[1]

Chapter Overview

The chapter is organised as follows. Section 3.1 sets out the reach of the analysis to be presented. The analysis proper is composed of two sections: Section 3.2 characterises the popularity of MDE within the broad software industry while Section 3.3 teases out the broad themes that emerge from its adoption. Lastly, the chapter concludes with a brief discussion that summarises our findings (Section 3.4). Let us begin by laying out the scope and limits of the effort.

3.1 LIMITATIONS

Our analysis is constrained by two types of impediments: intrinsic limitations of the model-driven approach, and shortcomings related to the way adoption research has been conducted. The next two sections delve into each of these categories respectively.

1 It may be argued that our deep interrogation of MDE is excessive. However, it is our firm opinion that its application — and the definition of methodologies based on it — can only be done successfully once its key strengths and weaknesses are well understood, and this requires looking at both theory and practice. To our knowledge, this is an undertaking thus far absent from the literature.

3.1.1 *Intrinsic Limitations to MDE*

Motivation

As recently as 2011, Hutchinson *et al.* (Hutchinson et al., 2011) warned that MDE had historically lacked evidence to back many of its claims (Hutchinson et al., 2011).[2] These difficulties led Whittle *et al.* to admit that "[...] there remains a lack of clarity on whether or not model-driven engineering (MDE) is a good way to develop software." Their wariness is not entirely unjustified. In the previously mentioned paper, Hutchinson *et al.* identified three key reasons for why, in practice, MDE may have a detrimental effect in the development of software systems:

Reasons for MDE detrimental effects

1. **Higher abstraction levels may not lead to better software.** Citing experiments in psychology and psychology of programming, they point out that individuals find thinking in abstract terms hard and, in general, there is a tendency to prefer exemplars over abstract conceptualisations.

2. **MDE activities may have both positive and negative effects.** Code generation is offered as an example, as it may have a positive effect on productivity but also negative consequences — including the time required to develop the models for code generation as well as the possible need to integrate manual code with generated code.[3] Nor is code generation the only MDE activity with this conflicting property, making it a very thorny issue: "How the balance between these two effects is related to context, and what might lead to one outweighing the other, is simply not known." (Hutchinson, Rouncefield, and Whittle, 2011)

3. **Determining the right approach is hard.** Given the proliferation of MDE variants, tools and frameworks[4], it is very hard to evaluate benefits and drawbacks in a rigorous way. Choices are highly dependent on context and thus difficult to compare in a fair manner. This is a theme we will return to in the next section.

Torchiano *et al.* (Torchiano et al., 2012) are even more critical of the *status quo*, managing to combine the challenges of practice with those of theory into one clear and incisive diagnosis (*emphasis ours*):

Lack of maturity, repeatability

> [MD*] is considered to be still evolving and not yet completely mature. The first success stories were heard a long time ago but *the knowledge to make those successes consistently repeatable is still missing*. Being the discipline not yet fully understood, and the underlying knowledge not yet codified, *expertise is the only resource we can rely on when a MD* solution is designed*.

Their criticism evokes the idea of a practitioner mastering a body of knowledge, as we have defended (Craveiro, 2021c) (Section 2.2) — a scenario that is inherently unsuitable for repeatable and comparative analysis as it is highly

2 The original wording by Hutchinson *et al.* is rather more intriguing (*emphasis ours*): "Although MDE claims many potential benefits in terms of gains in productivity, portability, maintainability and interoperability, it has been developed largely *without empirical support* for these claims."

3 Our personal experience mirrors this effect quite closely (Craveiro, 2021b) (Section 6 and 7).

4 As detailed in (Craveiro, 2021c), Chapter 2 — and Section 2.4 in particular.

dependent on context.[5] When taken together, all of these open questions raise even further the significance of adoption research. Unfortunately, it too faces several challenges.

3.1.2 *Adoption Literature Limitations*

The literature reveals a litany of experience reports and empirical studies on MDE application (Andolfato et al., 2014; Shirtz, Kazakov, and Shaham-Gafni, 2007; Paige and Varró, 2012; Clark and Muller, 2012), and to these we have added our own (Craveiro, 2021b). Taken together, they form a useful but somewhat fractious breeding ground from which to extract universal answers as to the suitability of MDE. Hutchinson *et al.* had already raised red flags when performing a similar exercise (*emphasis ours*):

Qualitative versus quantitative analysis

> As many of our respondents admit, *quantification of the benefits and failures of MDE is complex and difficult*. [...] [The] evidence for our understanding of MDE at this stage derives from the *quality and perceptiveness of our descriptions and our analysis* rather than any simple mathematization. (Hutchinson, Rouncefield, and Whittle, 2011)

Much the same said Mohagheghi and Dehlen (Mohagheghi and Dehlen, 2008), who, whilst identifying threats to the validity of their own work, inadvertently enumerated the challenges faced by anyone risking a similar undertaking. These are as follows:

- **Small sample size.** Though not lacking on variety, the number of reports found in the literature is insufficient to reach a "generalization to a population or theory." (Mohagheghi and Dehlen, 2008)

- **Survivorship bias.** In their opinion — which is also ours — successes are more likely to be reported than failures; therefore, the literature may portrait an inaccurate picture of the most likely outcome of MDE projects.

Pitfalls of adoption surveying

- **Incentives for biased reporting.** Research projects with external financing may report biased results, downplaying negative outcomes. Companies may also behave in a similar fashion to avoid negative publicity. Incentives amplify the survivorship bias.

- **Interference from competitive advantages.** Conversely, companies may avoid publishing results in order to keep a competitive advantage against their peers.

- **Large-scale projects are under-reported.** The success of MDE on small-scale projects such as academic research may not be indicative of its

5 Indeed, in this sense we are closer to McBreen's Software Craftsmanship (McBreen, 2002) rather than to Software Engineering. There is much to be said for McBreen's idea that, on the main, the creation of high-quality software may not be amenable to repeatable engineering processes, but sadly such discussion falls outside the remit of this dissertation.

success on large industrial projects. Results for large-scale projects are not as frequent as those of small-scale projects.

- **Lack of baseline data.** Most companies do not report baseline data, which makes evaluations subjective. In addition, it is very difficult to obtain relevant and unbiased baseline data.
- **Lack of quantitative data.** Most studies focus on the qualitative aspects of the experience and neglect a quantitative analysis. More generally, there is a lack of well-defined dimensions for the gathering of quantitative data, for reasons such as those outlined by Hutchinson *et al.* above.

Difficulties with "universal arguments"

From what has been stated thus far, it seems clear that it is very difficult to use the literature to make universal statements about MDE adoption with any degree of confidence.[6] Indeed, we argue that most universal statements about MDE are devoid of meaning from a scientific perspective because they must be localised to the specifics of a context and can only be extrapolated to other contexts with a great deal of care; even then, they would still be riddled with reservations. In other words, it is extremely difficult to compare MDE projects because they are highly sensitive to local conditions — conditions which are not readily replicable — much less lend themselves to easy aggregation at an industry-wide level. The essence of the problem was captured by Rumpe earlier, who had stated:

Wicked problems

> The pressing problems that we tackle in the software and system modeling research domain can be classified as "wicked problems": we learn more about the nature of the problems we tackle through experimentation with proposed solutions. Rigorous evaluation of these solutions invariably entails costly and lengthy experimentation in industrial contexts. Experiments that seek to evaluate solutions based on novel or radically different ideas are particularly difficult to sell to potential industrial partners because the risks are not well-understood by all involved. Even with committed industrial partners, the wide variations in industrial development environments makes it difficult (if not foolhardy) to extrapolate the results beyond the specific industries. Despite the difficulties, there is no getting away from the reality that evaluation is key to developing progressively better solutions to wicked problems. (France, 2008)

Importance of empirical studies

In this light, even though empirical studies are not particularly useful in proving or disproving universal claims, they are still extremely valuable because they capture the themes emerging from within application. From these we can build conceptual tooling to augment MDE's body of knowledge — best practices, patterns, guidelines and the like[7] — and, for the purposes of MASD, these observations can guide the requirements gathering process (*cf.* Chapter 4). On the main, it is in this spirit that the MDE adoption literature is to be understood within this dissertation. All of that said, we shall start by attempting to tackle one universal question — limitations described here notwithstanding.

6 By "universal statements" we mean blanket statements such as "MDE improves productivity", "MDE improves software quality", and suchlike.

7 Manuscripts such as Völter's "MD* Best Practices" (Völter, 2009) capture well the idea.

We will do so for two reasons. Firstly, because we believe it offers far-reaching insights into the application of modeling in general and therefore to MASD. Secondly, because it can be answered — even if only broadly. So it is to that pivotal question we turn to next.

3.2 how widely adopted is mde?

High expectations were set out in a seminal presentation by Bézivin (Bézivin, 2003), where he ambitiously declared model engineering to be the future of object technology, and outlined a twenty year horizon for its maturing. As we sail at speed towards Bézivin's evaluation date, it is increasingly clear that the adoption curves of object technology and model technology are distinct: the former became the mainstay of software engineering within less than twenty years of its inception, whereas the latter is yet to achieve similar levels of exposure. These thoughts are echoed by Mussbacher *et al.*, who lamented that (*emphasis ours*):

Object technology, model technology

> [...] *MDE is arguably still a niche technology*. It has not been adopted as widely as popular programming languages such as Java and C#, and, whilst some modeling languages like the UML have become widespread, they are often not used to their full potential and *the use of models to automatically generate systems is still relatively rare*. (Mussbacher et al., 2014)

Niche technology

In sharp contrast with the hopeful tone of the past[8], the literature now has a downcast mood[9], perhaps reflecting the realisation that MDE "[...] is currently not as widespread in industry as the modeling community hoped for." (Mussbacher et al., 2014) This state of affairs is all the more surprising when one considers that the claims associated with MDE are often decisive factors in an industrial setting.[10] There is therefore a clear disconnect between promises and adoption, perchance not unrelated to the difficulties in evidencing grandiose universal assertions (*cf.* Section 3.1.2).

Promise-adoption gap

Our objective for this section is twofold. First, we want to substantiate or disprove Mussbacher *et al.*'s claims by representing, however broadly, MDE's state of practice, because we believe it offers considerable insights as to the applicability of its vision to the reality of industrial software engineering. Second, we want to perform Bézivin's evaluation by characterising the direction

Objectives, approach

8 For a partial timeline of events related to MDE, see Clark and Muller (Clark and Muller, 2012), Section 2, "The MDD Landscape".

9 Articles such as Bell's "Death by UML Fever" (Bell, 2004), Thomas' "MDA: Revenge of the Modelers or UML Utopia?" (Thomas, 2004) and France *et al.*'s "Model-driven development using UML 2.0: promises and pitfalls" (France et al., 2006) accurately depict the zeitgeist.

10 The MDA Guide is well aware of this, stating (*emphasis ours*): "Automation *reduces the time and cost* of realizing a design, *reduces the time and cost* for changes and maintenance and produces results that ensure consistency across all of the derived artefacts." (Group, 2014)

of travel of MDE adoption — again, in broad strokes[11,12] — because it gauges industry's reaction to it. The next two sections analyse evidence from multiple sources to position MDE within this frame. Evidence was collected at two different scales: the macro-scale — that is, across the software engineering profession — and the micro-scale — that is, small samples and empirical studies. With this in mind, let us start our analysis from a high vantage point.

3.2.1 *Analysis of Evidence at a Macro-Scale*

MDE is "not cool"

Whilst the literature does provide detailed quantitative data for MDE adoption at small sample sizes (*cf.* Section 3.2.2), measurements that place the approach in an industry-wide context are harder to come by. In the above cited paper, Mussbacher *et al.* used data from search engine queries to demonstrate, somewhat amusingly, that "MDE is simply not considered cool". Unfortunately, their dataset was insufficient for our purposes; however, their methodology was promising, so we extended it to other large data sources freely available on the Internet.

Google Trends data source

Two such sources were used for our analysis. The first, Google Trends[13], allows measuring interest over time from a search engine perspective.[14] We started by gathering evidence to support Mussbacher *et al.*'s claim of MDE being a niche technology by comparing interest in MDE and UML to interest in the suggested programming languages — Java and C#.[15] Figure 3.1 does imply that, compared to Java and C#, both UML and MDE are fairly niche from the perspective of search engine querying.

Next, we analysed how queries related to MDE have evolved in the time dimension. For this we plotted interest over time for six different search terms related to MDE, with the results captured by Figure 3.2. The data allows us to make some general observations, as the graph clearly shows a marked decline from a peak around 2004, and a sharp descent soon thereafter; furthermore, it lacks any obvious upticks during the latter years, which would indicate some form of revival as the industry gets to grips with the approach.

11 The Hype Cycle Model (Linden and Fenn, 2003) is often deployed in this context — *e.g.* (Brambilla, Cabot, and Wimmer, 2012) (p. 22), Torchiano *et al.* (Torchiano et al., 2012) — and understandably so, for, in the words of Torchiano *et al.*, "[h]ype is frequently associated to software development processes/techniques until (*sic.*) they are not yet mainstream and fully understood; we think it is also the case for modeling and MD*." We opted for a dissenting view nonetheless, siding instead with those with concerns about the model such as Dedehayir and Steinert (Dedehayir and Steinert, 2016). Our main qualm is the difficulty in determining whether we are in the model's final moment (*i.e.* the so-called "plateau of productivity") or if we have entered a perpetual descent in the "through of disillusionment". Without adequate quantitative datasets, statements on this regard are problematic to substantiate and therefore we do not believe the model brings any additional clarity towards the evolution of MDE adoption.

12 Note that due to the limitations already described (*cf.* Section 3.1.2), the purpose of this analysis is not a rigorous determination, but merely the use of multiple sources to get a sense of where the theory has led us thus far.

13 `https://trends.google.com`

14 Google Trends defines *interest over time* as follows: "Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means that there was not enough data for this term."

15 These languages rank 2nd and 5th respectively in the TIOBE Index, at the time of writing (TIOBE, 2021). The TIOBE Index offers a measure of popularity for programming languages.

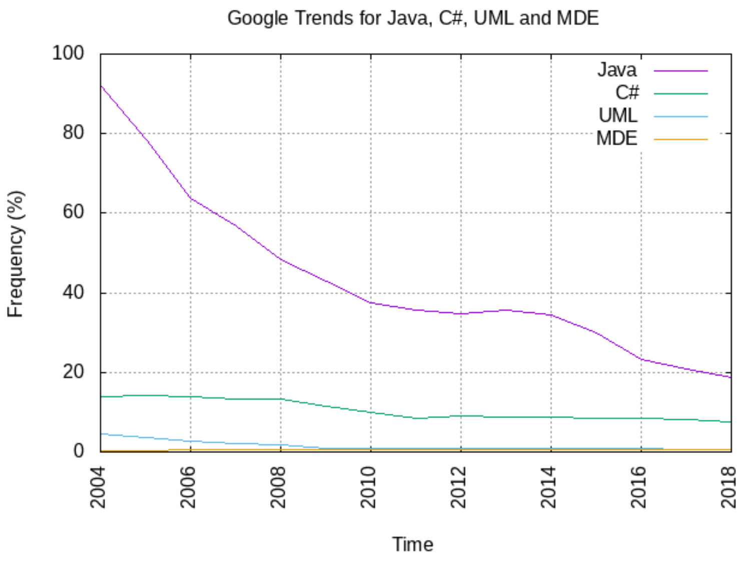


Figure 3.1: Google searches for Java, C#, UML and MDE. *Source:* Author's plot using Google Trends data (January 2004 to August 2018)

It is important to understand that there are numerous limitations with our analysis, such as the clipped time horizon (available data excludes the period from 2000 to 2004), the reliance on Google as the sole data source (searches may have been performed on other search engines, though Google's dominant market position mitigates this risk), the exclusion of queries using acronyms (adding MDE, MDA, *etc.* to the report caused false positives), and so forth. Nonetheless, in the absence of better data, it serves as a coarse approximation.

Google Trends threats to validity

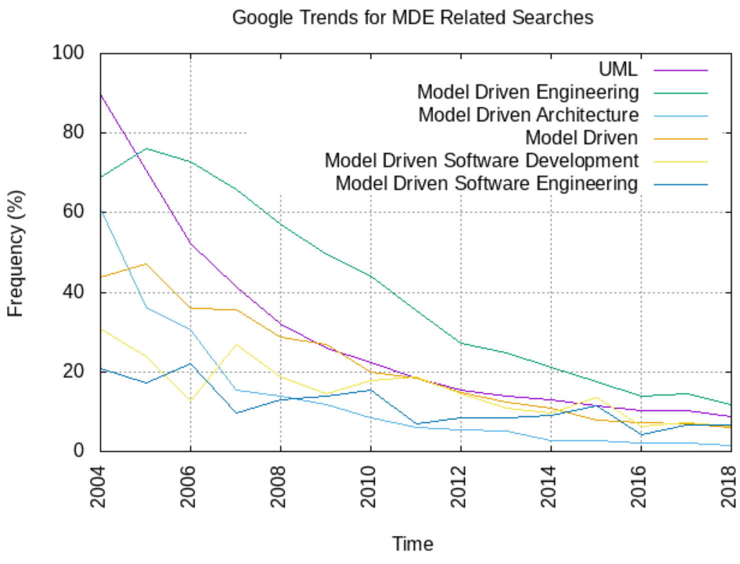


Figure 3.2: Google searches related to MDE. *Source*: Author's plot using Google Trends data (January 2004 to August 2018).

Stack Overflow data source

Our second data source was Stack Overflow[16], a question and answer website popular with software engineers[17], which provides access to statistical data via its *Insights* query interface.[18] Questions in Stack Overflow are tagged by its users to facilitate searching and aggregation[19], and the distribution of tags can be analysed via Insights. The absence of evidence was informative in itself, as no tags could be located with regards to "model-driven", "MDE", "MDD", "MDSE" or any other MDE variant described in this dissertation. UML was the only available tag related to modeling, and the resulting data is plotted in Figure 3.3. The number of questions for UML as a percentage of total questions reached a maximum of 0.12% and has declined to around 0.02%.

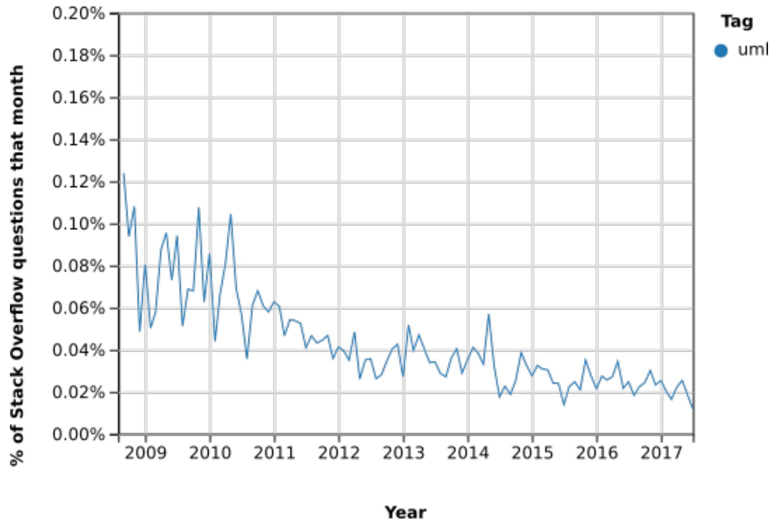


Figure 3.3: Questions tagged with UML on Stack Overflow. *Source*: Stack Overflow Insights from 2009 to July 2018.

We then tried to establish UML's position when compared to Java and C#. The results, plotted in Figure 3.4, are in line with the findings from Google Trends: UML is quite niche when compared to popular programming languages.

Stack Overflow threats to validity

As with Google Trends, it is important to highlight the limitations of Stack Overflow as a data source, since it excludes software engineers who need not ask questions about MDE (*e.g.* experienced practitioners) as well as those who use other sources of information (commercial product support, other web forums), questions may be incorrectly classified, questions may be tagged against specific tools rather than modeling terms, the data range is narrow (only covers the period from 2009 to 2018) and so forth. Similarly to Google Trends, these limitations were deemed acceptable for our purposes.

In summary, the picture emerging from the analysed data is in overall agreement with those who view MDE as a niche technology, facing a trend of decreasing interest in industry. We shall now contrast these findings with evidence at the opposite end of the scale.

16 `https://stackoverflow.com`

17 According to internal data, Stack Overflow has a total of 16 million questions, 25 million answers, 9.2 million active users and over 9.8 million visits per day (Exchange, 2018).

18 `https://insights.stackoverflow.com`

19 Stack Overflow defines *tags* as follows: "A tag is a word or phrase that describes the topic of the question. Tags are a means of connecting experts with questions they will be able to answer by sorting questions into specific, well-defined categories." (Overflow, 2018)

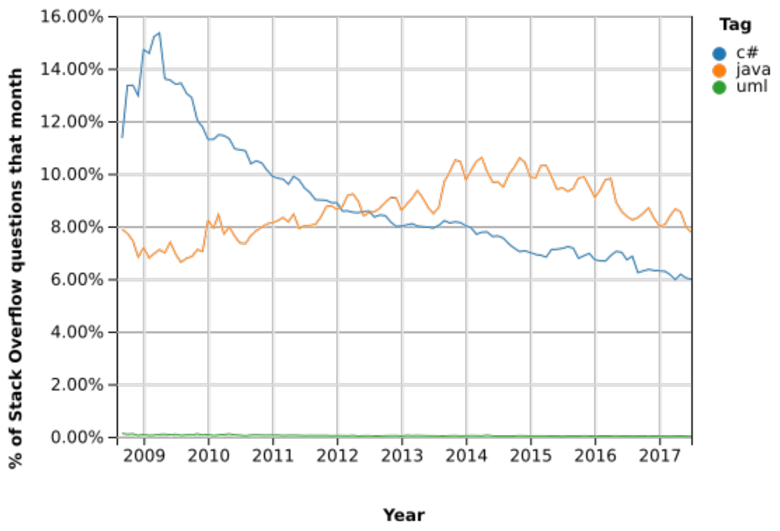


Figure 3.4: Questions tagged with Java, C# or UML on Stack Overflow. *Source*: Stack Overflow Insights from 2009 to July 2018.

3.2.2 *Analysis of Evidence at a Micro-Scale*

A change of perspective is often revealing when addressing wicked problems (*cf.* Section 3.1.2). On what the authors claimed was the first wide-range industry study of its kind, Whittle *et al.* (Whittle, Hutchinson, and Rouncefield, 2014) surveyed 450 MDE practitioners with the objective of characterising MDE practice, and their findings are in stark contrast to those described thus far. The authors were already aware of this discrepancy, stating (*emphasis ours*):

> Some claim that the application of MDE to software engineering is minimal. MDE, they argue, is only used by specialists in *niche markets*. Our data refutes such claims, however. We have found that *some form of MDE is practised widely*, across a diverse range of industries (including automotive, banking, printing, web applications *etc.*

Widespread use of MDE

The unmistakable conclusion of Whittle *et al.*'s study is that MDE *is* widely used in industry, but in ways that are extremely difficult to quantify in the aggregate. For example, their analysis points out that practitioners prefer creating DSLs over using general purpose modeling languages such as UML, and implement these DSLs using a dizzying array of tools, techniques and frameworks (*emphasis ours*): "We found no consensus on which modeling languages or tools developers use — they cited over *40 modeling languages* and *over 100 tools* as 'regularly used' in our survey." Whittle *et al.*'s analysis builds upon earlier work from Petre (Petre, 2013), who showed that, in a sample of 50 software designers, very few used UML, and those that did — a total of 11 — used it only for "selective" purposes.[20] Petre's findings are in agreement with

Extreme diversity

20 Petre defines *selective* in the following, somewhat recursive, manner: "UML is used in design in a personal, selective, and informal way, for as long as it is considered useful, after which it is discarded." (Petre, 2013)

the evidence from our macro-analysis. The picture that emerges from both of these studies is one of great diversity and fragmentation, which further reinforces our own views of MDE as a diffused body of knowledge (Craveiro, 2021c) (Chapter 2). Furthermore, these results are not isolated; a related study by Hutchinson *et al.*, with 250 respondents (Hutchinson et al., 2011), found similar heterogeneous patterns.

Limitations

One can, of course, question the exact meaning of the expression "practised widely", given the small sample size and the fact that Whittle *et al.* surveyed only MDE practitioners — all of which makes it rather difficult to place their work in the context of the wider industry. In addition, as previously demonstrated, the boundaries of MDE are rather porous (Craveiro, 2021c) (Chapter 2) so the criteria for classifying any given project as an "MDE project" — in their words, "some form of MDE" — is not entirely free of ambiguity; in the limit, any project using code generation or a DSL could be construed as a "MDE project", though that, perhaps, may not be in the spirit of the endeavour.[21] Nonetheless, even when taking these validity threats into consideration, it is clear that their work provides undeniable evidence of MDE adoption across a variety of scenarios; more so than its niche status would imply.

Mainstream absorption

Clark and Muller (Clark and Muller, 2012) uncover a second reason that may help explain why it is difficult to find evidence of MDE use in the large, when they conclude that "the spirit of model-driven technology is very alive, although absorbed by mainstream programming environments [...]." In their view, there has been a steady flow of ideas and concepts from MDE's body of knowledge to traditional programming environments, and this, they suggest, is a trend that is set to continue or even accelerate due to commercial demands (*emphasis ours*): "The next generation of companies making use of model-driven technologies might be more successful if they manage to *hide model-driven technology*, embedding it as a competitive advantage."[22]

Taking all of these views into account, the evidence at the micro-scale is consistent with the idea of MDE's body of knowledge being digested and repurposed into a series of technological changes that are fit for specific purposes. And, now that both macro and micro cases have been presented with contradictory results, we must attempt to reconcile these viewpoints.

3.2.3 *Discussion*

The evidence at a macro-level reinforces Mussbacher *et al.*'s intuition of MDE's niche status, whilst the evidence at the micro-level provides support for the idea that MDE is in widespread use but, crucially, not in accordance to its

21 A small sample of the inclusion criteria in the reviewed material should suffice to give a flavour of this dilemma. Hutchinson *et al.* stated (Hutchinson et al., 2011): "The study takes a deliberately broad interpretation of MDE, as it is intended to be exploratory. Therefore, all variants of MDE are covered, including both domain-specific modeling languages (DSMLs) and UML-based methods. [...] The only hard criterion for excluding/including data was that the company must have been using models as a primary development artifact (*sic*)." For their part, Mohagheghi and Dehlen (Mohagheghi and Dehlen, 2008) limit their study to approaches that generate "models, code and other artifacts from models".

22 These very words were inspirational to the MASD approach.

original vision.[23] Both statements are not incompatible. Our opinion is that this vision is yet to come to pass, and its now unlikely to do so within the Bézivin horizon. Further: the limited available data seems to point out that the industry is moving away from that direction altogether, and is instead choosing the path of dispersing the MDE body of knowledge into small and localised solutions, and this may ultimately prove to be the enlightenment that has been long sought. If so, Cook's words now sound eerily prophetic (Cook, 2006) (*emphasis ours*):

Detachment from original vision

> The notion that all of software development will somehow be replaced by modelling is at least as mistaken as "objects are just there for the picking". [...] *Specific kinds of models are useful in specific tasks*; the modelling language used for a specific task must be designed to be fit for that task. Today's increasing interest in Domain Specific Languages, rather than general-purpose modelling languages, clearly recognizes this.

Modeling as a tool in the toolbox

With these wise words, to which we subscribe fully, we conclude our outline of the industry-wide adoption picture. It is now time to turn our attention to the analysis of themes emerging from MDE's application, in order to gain a better understanding of the specific challenges faced by those using the approach.

3.3 EMPIRICAL ANALYSIS OF ADOPTION LITERATURE

A great deal of insightful information can be extracted from the MDE adoption literature, much of which is relevant to our work. The information is, however, in a form that is troublesome to analyse due to its qualitative nature. In order to address this problem, we decided to deploy the classification system put forward by Whittle *et al.* in (Whittle et al., 2013), and subsequently improved upon (Whittle et al., 2017). There, they describe it as "a loose taxonomy of tool-related considerations, based on empirical industry data, which can be used to reflect on the tooling landscape as well as inform future research on MDE tools."

Tooling Taxonomy

Our previous brushes with the taxonomy revealed it to be malleable, amenable not only for the analysis of MDE tooling but also of MDE adoption issues in general (Craveiro, 2021b; Craveiro, 2021d). This is likely a byproduct of the close relationship between MDE application and its tooling, for, as Mohagheghi and Dehlen perceptively noted, "[s]upporting MDE with a comprehensive tool environment is crucial, as many of the techniques promoted as necessary in MDE strongly depend on proper tool support." (Mohagheghi and Dehlen, 2008)

Applicability outside tooling

23 This vision is articulated clearly by France and Rumpe:

> In the MDE vision, domain architects will be able to produce domain specific application development environments (DSAEs) using what we will refer to as MDE technology frameworks. Software developers will use DSAEs to produce and evolve members of an application family. A DSAE consists of tools to create, evolve, analyze, and transform models to forms from which implementation, deployment and runtime artifacts can be generated. Models are stored in a repository that tracks relationships across modeled concepts and maintains metadata on the manipulations that are performed on models. (France and Rumpe, 2007)

Each of the sections below tackle one of the four top-level categories in the taxonomy. They start with a brief description of the category, in order to contextualise those unfamiliar, and proceed to analyse relevant issues gleaned from the adoption literature. As with Section 3.2, the objective is not to perform an exhaustive review but instead to capture overarching themes that are pertinent to the present dissertation. In addition, due to its qualitative nature, we have chosen to rely on the same approach as did Whittle *et al.* — namely, the use of extensive quoting from original sources in order to register more faithfully the underlying themes.

Approach

3.3.1 *Technical Factors*

Technical issues that affect the adoption of MDE, such as the features available in tooling and their integration with development environments, were prominent in the literature. In particular, there seems to be a noticeable mismatch between developer expectations and functionality available in tools, as Clark and Muller explain (*emphasis ours*):

Developers prefer black-boxes

> Industry would like to use MDD as *a shrink-wrapped black-box process*. Current technologies expose a great deal of the inner workings of PIM, PSM and transformation design. *Developers feel that they need to have a detailed knowledge of all aspects of the technology* which undermines its commercial value compared to the use of more trusted mature technologies such as compilers. (Clark and Muller, 2012)

The compiler metaphor is particularly apt, since developers are used to code generators that behave in a fashion similar to compilers (Craveiro, 2021d). Whittle *et al.*'s findings echoed similar thoughts, revealing that the available functionality often exceeds the typical needs of software engineers:

Complex graphical interfaces

> Our interviewees emphasized tool immaturity, complexity and lack of usability as major barriers. Usability issues can be blamed, at least in part, on an over-emphasis on graphical interfaces: "... I did an analysis of one of the IBM tools and I counted 250 menu items." More generally, tools are often very powerful, but it is too difficult for users to access that power; or, in some cases, they do not really need that power and require something much simpler: "I was really impressed with the power of it and on the other hand I saw windows popping up everywhere... at the end I thought *I still really have no idea how to use this tool* and I have only seen a glimpse of the power that it has." (Whittle et al., 2017)

Humans must adapt to tools

The reliance on graphical interfaces appears to be a common complaint from developers: "There is a large amount of evidence that software engineers prefer textual representations for system artifacts rather than diagrams." (Clark and Muller, 2012) The underlying thread that unifies all these observations is an impedance mismatch between how developers would like tools to behave versus how tool developers view the role of those tools. Whittle *et al.* express a concern for the egregious "[...] lack of consideration for how people work and think: 'basically it's still the mindset that the human adapts to the computer, not vice-versa.'" (Whittle et al., 2017)

They go on to explain that many practitioners addressed this mismatch by creating their own special purpose tooling, with reportedly better results (*emphasis ours*):

> The majority of our interviewees were very successful with MDE but all of them either built their own modeling tools, made heavy adaptations of off-the-shelf tools, or spent a lot of time finding ways to work around tools. The only accounts of easy-to-use, intuitive tools came from those who had developed tools themselves for bespoke purposes. Indeed, this suggests that *current tools are a barrier to success rather than an enabler* and "the fact that people are struggling with the tools... and succeed nonetheless requires a certain level of enthusiasm and competence." (Whittle et al., 2017)

Bespoke tools are more successful

Nevertheless, it is important not to underestimate the immense effort required to create industrial-grade MDE tooling, even when experienced practitioners are involved, as Paige and Varró's work amply demonstrates (Paige and Varró, 2012), and so does Andolfato *et al.*'s (Andolfato et al., 2014). Speaking in the context of tool vendors, Clark and Muller put the matter in more forceful terms (*emphasis theirs*): "*Tool development is expensive*. However much you think it will cost in terms of time and effort to develop a business based on a modelling tool, multiply by 10. Be prepared to be patient and support development through other activities." (Clark and Muller, 2012) Internal tool development does benefit from a narrower focus, of course, but it is no less difficult; the cost and the associated risk in developing new tools for a complex problem domain such as modeling — in most cases, a proposition that is entirely unrelated to the main business activity and the current engineering skill-set — is an obvious barrier to MDE adoption. Our own work sheds light on some of the challenges developers faced:

Tool development is expensive

> Interviewees developed an appreciation for the difficulty of creating a general purpose code generator: "[...] It was a massively ambitious project, right? [To] [b]uild a general purpose code generator, is a very, very difficult thing."
>
> Once the magnitude of the task was understood, a natural process of de-scoping started to take place: "But when you say, 'I want to write a code generator that is going to work for everything', well then now you need to define what everything is. And how do you define *everything*? [...] You can't, so you say 'right I'm guessing I'm going to need lists, but I'm guessing I won't really need dictionaries, I'm guessing it will [be] good enough just to have public setters but let's not worry about public/private, everything will be public and that's [...] something reasonable that I can write a code generator for, within a year and a half and [...] that'll have to do.' [...]" (Craveiro, 2021b) (Section 6.1)

Scope challenges

Thus, the trade-offs between off-the-shelf tooling and internal tool development are very complex, and highly dependent on situational context. Whittle *et al.*'s words summarise the issues with technical factors rather aptly (*emphasis ours*):

MDE as a source of accidental complexity

> It is ironic that MDE was introduced to help deal with the essential complexity of systems, "but in many cases, adds accidental complexity". Although this should not be surprising [...], it is interesting to describe this phenomenon in the context of MDE. For the technical categories, in almost every case, interviewees gave examples where the category helped to tackle essential complexity, but also other examples where the category led to the introduction of accidental complexity. So, interviewees talked about the benefits of code generation, but, at the same time, lamented the fact that "we have some problems with the complexity of the code generated [...] we are permanently optimizing this tool." (Whittle et al., 2017)

3.3.2 *Internal Organisational Factors*

The adoption of MDE takes place in the context of an organisation with its own distinctive structure, processes and procedures, as well as a unique culture. The literature clearly highlights both the importance and the difficulty in integrating MDE related infrastructure with what precedes it:

Difficult tooling integration

> One interviewee described how the company's processes had to be significantly changed to allow them to use the tool: a lack of control over the code generation templates led to the need to modify the generated code directly, which in turn led to a process to control these manual edits. Complexity also arises when fitting an MDE tool into an existing tool chain: "And the integration with all of the other products that you have in your environment..." Despite significant investment in providing suites of tools that can work together, this is clearly an area where it is easy to introduce accidental complexity. (Whittle et al., 2017)

This stumbling block was also observed by Mohagheghi and Dehlen: "Integrating a tool suite that satisfies these requirements into a coherent environment is evidently a challenge. In the MODELWARE project, a wide range of tools were used, but all partners experienced problems with instability of the tools and their integration." (Mohagheghi and Dehlen, 2008)

Sustainable tool development

Interestingly, whilst bespoke development facilitates tooling integration as the requirements are very specific to an organisation, it often has a clear downside with regards to the sustainability of tooling engineering because it lacks an alignment with core business activities. In general, developers have very little appetite for extraneous activities on an already congested development schedule, as we witnessed first-hand:

> A key point was the difficulty in justifying continued investment on a bespoke code generator from a business perspective: "Its quite product-y? So it almost feels like, you know, its something which

> [we] should be buying in or open source [...]. Not what you want to be focusing your interest on, if you can [...] avoid it." (Craveiro, 2021b) (Section 6.5)

Finally, there are also challenges due to the immaturity of the formal processes associated with MDE, as Mohagheghi and Dehlen note:

> The importance of utilizing a defined process in software engineering has been known for several years. However, most "tried and tested" processes are not tailored for MDE, which does not make any assumptions on the software development process or the design methodology. Baker *et al.* report that many teams in Motorola encountered major obstacles in adopting MDE due to the lack of a well-defined process, lack of necessary skills and inflexibility in changing the existing culture [...]. (Mohagheghi and Dehlen, 2008)

Process integration

3.3.3 *External Organisational Factors*

MDE adoption is also impaired by factors that are external to the organisation applying it. As already highlighted (*cf.* Section 3.2.2), the MDE tool offering is very large and diverse, bringing with it its own problems such as a difficulty in deciding on the appropriate tooling for a given project — much as did CASE before it (*cf.* Section 2.2.1). In addition, once tools are identified, there are always fears of vendor lock-in, as Mohagheghi and Dehlen report: "[t]he vendor lock-in problem persuades some users to use open source tools such as the Eclipse framework. Others combine third-party products with self-developed tools [...], or develop their own tools [...]." (Mohagheghi and Dehlen, 2008)

Tool selection, vendor lock-in

Relying on a vendor may also be problematic if the vendor is small, as Paige and Varró demonstrate by vividly narrating the many lessons they've learned whilst creating and shutting down two MDE tool vending start-ups (Paige and Varró, 2012). Our key take-away from their work is that the software tooling market is generally very competitive and MDE tooling is no different, so, as part of any feasibility analysis, it is very important to take into consideration the sustainability of vendors far into the future. This problem is exacerbated because interoperability between tools of different vendors is not yet a fully resolved issue, even in the presence of mature industry standards such XMI. In (Lundell et al., 2006), Lundell *et al.* analysed the impact of XMI on heterogeneous tooling environments and, whilst being generally positive about the standard, they also helped explain why interoperability remains such a thorny issue (*emphasis ours*):

Vendor sustainability

XMI

> In considering the results of the tests it should be noted that *anything short of complete success is of limited value in practice*. The work involved in repairing significant semantic loss in an interchanged model is often considered infeasible for industrial strength models. With this in mind, from the perspective of legacy systems and tool lock-in, the new generation of modelling tools has not generally

Tool interoperability challenges

> improved prospects for importing existing models exported from earlier tools. (Lundell et al., 2006)

As a result, practitioners are often wary of finding themselves involved with what France and Rumpe called the "the DSL-Babel challenge"; that is, the fear that the "[...] use of many DSLs can lead to significant interoperability, language-version and language-migration problems." (France and Rumpe, 2007)

3.3.4 *Social Factors*

Bottom-up approach

An influential aspect of MDE adoption is concerned with issues of control and trust, particularly with regards to the tools of third-party vendors or of those produced by large internal tooling teams with a degree of independence from their end-users. Whilst vendors have historically tried to promote holistic top-down solutions — very much in line with the MDE vision (*cf.* Section 3.2) — the adoption literature shows that small-scale, bottom-up approaches tends to yield better results in practice, as Whittle *et al.* explain (*emphasis ours*):

Top-down development

> Our findings also lead us to believe that *most successful MDE practice is driven from the ground up*. MDE efforts that are imposed by high-level management typically struggle; interviewees claimed that top-down management mandates fail if they do not have the buy-in of developers first. As a result, *there are fewer examples of the use of MDE to generate whole systems*. Rather than following heavyweight top-down methodologies, successful MDE practitioners use MDE as and when it is appropriate and combine it with other methods in a very flexible way. (Whittle, Hutchinson, and Rouncefield, 2014)

Regardless of whether tools are bespoke or sourced from external vendors, a common approach taken by developers to handle tooling inadequacies is to subvert the tools in order to achieve some desired behaviour:

Consequences of tooling inadequacy

> These and other speculative features [immutability, factory methods] were not optional, perhaps in order to restrict variability, so as a consequence software engineers started to make use of them best they could: "I think its a good point, you just learn to live with what you got, right? [...] These are the [...] constraints that we have, so we're going to have to [...] live with those constraints. [...] And you find a way, right?" (Craveiro, 2021b) (Section 6.3)

Tool subversion

Whittle *et al.* recorded eerily similar experiences: "A second example is a company that mandated the use of a commercial MDE tool. However, the developers could not get the tool to fit their processes, and, under pressure to 'make things work', they hacked it, messed with the generated code, and circumvented it when they had to." (Whittle, Hutchinson, and Rouncefield, 2014)

And it is with those telling words — with pragmatism overriding theory to get things done — that we must end our swift excursion through Whittle *et al.*'s taxonomy, as well as through the larger terrain of MDE adoption literature. Let's us now gather a set of lessons learned from the analysis.

3.4 DISCUSSION

The previous sections presented themes unveiled by empirical analysis in the MDE adoption literature. Of these, we would like to highlight those that are of key importance to MASD:

Emerging themes

- **Usability is often a concern with MDE tooling.** There is often a mismatch between what developers expect of a tool and what vendors view as the role of the tool. Empirical evidence has shown this is a barrier for adoption.

- **MDE tools are expensive to develop and maintain.** Whilst bespoke tooling has a better fit, it is very difficult to develop due to the need for specialist skills and the costs involved.

- **Developers prefer bottom-up and incremental approaches.** Management and tool vendors seem to prefer top-down approaches, but software engineers prefer to integrate new tools and approaches incrementally, experimenting with them over time and increasing their usage as their mastery of the tool improves.

- **Integration with existing tooling is a challenge.** Software engineers typically use a variety of tools to achieve their goals, and continually pick up new tools over their career. In general, tools that integrate with existing toolsets are favoured over tools that force wholesale changes to development workflows.

Conclusions

Our conclusion from the analysis of MDE adoption is that there is a gap in the literature for methodology and tooling that are better aligned with how practitioners actually use MDE rather than researcher and tool developer's expectations. Now that the gap has been established, it is the role of the next chapter to distil these and other findings into a set of requirements designed to address this impedance mismatch.

4 REQUIREMENTS

Current modeling approaches, techniques and tools do not live up to the challenge. Often, mature tools provide techniques that can successfully cope with software systems that we were building a decade ago, but fail when applied to model complex systems [...]. *Some academic techniques propose interesting ways of addressing these shortcomings, but the prototypical nature of academic tools often prohibits their application to the development of real-world software systems.*

— Mussbacher *et al.* (Mussbacher et al., 2014)

Chapter Overview

Several far-ranging themes emerged from Chapters 2 and 3, as well as previous analysis (Craveiro, 2021b; Craveiro, 2021c; Craveiro, 2021d). The present chapter distils these themes into a cohesive narrative of gaps exploitable by a new MDE-based approach, linking each requirement back to the theoretical or practical concerns which gave rise to it. Since, in our view, application is inextricably linked with theory, we have opted for two categories of requirements: requirements for the theoretical framework (*cf.* Section 4.1) and requirements related to tooling (*cf.* Section 4.2). Each requirement is comprised of a numbered requirement definition to facilitate cross-referencing by the remainder of the dissertation[1] and a short overview. Let us start then by looking at what is deemed necessary from a theoretical standpoint.

4.1 THEORETICAL FRAMEWORK REQUIREMENTS

4.1.1 *Well-Defined Purpose*

Requirement 1 *The new approach will target a single, specific purpose, which is the automated generation of schematic and repetitive code.*

As explained at length in (Craveiro, 2021c) (Chapter 2), MDE is a vast body of knowledge with unclear boundaries. The new approach must avoid these difficulties by explicitly defining its purpose and clarifying its relationship with MDE and MD*. In addition, the boundaries of the new approach are circumscribed exclusively to AC-MDSD — namely, to the automatic generation of schematic and repetitive code (*cf.* Section 2.3.3).

1 *e.g.*, R-1 references the first requirement, defined in Section 4.1.1.

4.1.2 *Well-Defined Identity*

Requirement 2 *The new approach must have a cohesive and well-defined identity, distinct from that of MDE and MD*.*

Identity is a thorny problem within MDE (Craveiro, 2021c) (p. 10), and thus a frequent source of confusion to newcomers. In addition to having a well-defined purpose (R-1), any proposed solution must also have a well-defined identity, explicitly distinct from all existing approaches within MDE. An important consequence of a well-defined identity is that it will facilitate macro-analysis of its application (*cf.* Section 3.2.1).

4.1.3 *Well-defined Target Audience*

Requirement 3 *The new approach must be designed to serve specifically software engineers inexperienced in MDE.*

MDE accumulates knowledge on the subject of modeling, serving many distinct audiences in the software development process. As a consequence, it is difficult to find an entry point into the body of knowledge (Craveiro, 2021b), (Craveiro, 2021c) (Chapter 2). The new approach must set out its target audience explicitly; its focus is only on software engineers with little to no knowledge of MDE. The intent is not to discourage experienced MDE practitioners, but to bring clarity as to whom the approach is aiming to serve. All decisions must take into account the target audience.

4.1.4 *Well-defined Domain Architecture*

Requirement 4 *Resolve ambiguity in terminology by means of a single, well-defined, domain architecture that covers all core vocabulary.*

A key part in establishing a well-defined purpose (R-1) and identity (R-2) is to create a domain architecture that acts as the single source of truth for the approach. MDE concepts that are difficult to define rigorously — such as Technical Spaces (TSs), platforms and the like — must be made clear and unambiguous within the domain architecture. In addition, these more rigorous definitions must not disagree with their common usage within MDE.

4.1.5 *Cater for Evolution*

Requirement 5 *In order to adapt continually to the needs of end-users, the domain architecture must have an associated set of processes that cater for its continued evolution.*

The domain architecture (R-4) must be designed for continuous extension in order to remain relevant. The approach must define the processes by which it can be updated, as well as the criteria for inclusion or exclusion of changes. This requirement attempts to address France and Rumpe's warning regarding the "widening of the problem-implementation gap" through evolution (*cf.* Section 2.3.2), and reflects the importance placed on keeping up to date with changes on platforms and TSs.

4.1.6 *SDM Integration*

Requirement 6 *The new approach must be able to integrate seamlessly with existing SDMs.*

Given the limited scope of the new approach, software engineers will require an overall process to manage the software development lifecycle via a traditional SDM, which we call the host methodology. As with MDE, the new approach must be agnostic to the host methodology being applied. Unlike MDE, the new approach must integrate with its host without requiring any changes to the latter or to itself.[2]

4.1.7 *Clear Governance Model*

Requirement 7 *The new approach must have a clear governance model, identifying the responsibilities of all actors and allowing all interested parties to contribute.*

In order to allay concerns over cost, vendor lock-in and vendor survivability (*cf.* Section 3.3.3), the approach and all of its components must have a governance model that accepts contributions from a variety of sources. The approach must include a process by which any interested party can contribute changes. The governance model should be focused towards increasing the number of use cases for the approach (R-11), in accordance to its purpose (R-1) and target audience (R-3).

4.1.8 *Support for PIMs and PSMs*

Requirement 8 *The modeling language must support PIMs as well as PSMs.*

The modeling language for the new approach must support both PSMs and PIMs (*cf.* Sections 2.3.2). Platform independence must be built upon a set of well-defined mappings in accordance with the domain architecture (R-4), and extensible by users. Use of PIMs must remain optional.

2 The relationship between MDE and SDMs is discussed at length in (Craveiro, 2021c) (Chapter 5).

4.1.9 *Support for PDMs*

Requirement 9 *The modeling language must support Platform Description Models (PDMs).*

The new approach must cater for PDMs.[3] These provide mappings for external building blocks, making them accessible by the methodology's modeling language. PDMs must be extensible by users and subsequently incorporable into the domain architecture (R-4) and tooling via a well-defined process.

4.1.10 *Limited Support for Variant Management and Product Lines*

Requirement 10 *The approach's modeling language must provide support for variant management and product lines.*

The modeling language put forward by the approach must include limited support for product lines, constraining variability to a well-defined variability space.[4]

4.1.11 *Extensible Catalogue of Schematic and Repetitive Code*

Requirement 11 *The new approach must define a framework for the management of schematic and repetitive code.*

A framework for the management of schematic repetitive code (*cf.* Section 2.3.3) must be defined as part of the domain architecture (R-4). It must include a catalogue of the patterns already identified, including their taxonomy, variability and dependencies, as well as outlining a process to identify and propose new patterns.

4.2 TOOLING REQUIREMENTS

4.2.1 *"End-to-End" Solution*

Requirement 12 *The new approach must encompass all of the tooling required for its application.*

3 In the sense defined in (Craveiro, 2021c), Chapter 4 (p. 35); that is to say, a model responsible for mapping platform-level concepts into the domain architecture.

4 A review of the literature on variability falls outside of the scope of the present work. For a high-level overview of the subject — as well as an analysis of its relationship with MDE — see Chapter 6 of (Craveiro, 2021c).

In order to avoid the complexity created by heterogeneous tooling (*cf.* Section 3.3.3), the new approach must provide a single, end-to-end solution for the modeling and generation of schematic and repetitive code (R-11).

This entails specifying the behaviour of all necessary tooling (R-4) both in terms of their inputs — *i.e.* modeling languages — and outputs — *i.e.* programming languages. It also entails providing a comprehensive reference implementation. Significantly, this requirement does not preclude integrating with external tools, but instead attempts to minimise accidental complexity within its core competences (R-1).

4.2.2 *Prioritise Black-Boxing*

Requirement 13 *Where possible, MDE concepts should be made invisible to end-users.*

All tooling should use a black-box approach where possible. For example, end-users are not expected to have access to a MT language to operate on the input models; instead, from an end-user perspective, PIMs and PSMs must act as if transformed directly into source code (*cf.* Section 3.3.1). More generally, the approach should aim to promote tooling closer in spirit to special purpose code generators (Craveiro, 2021d) rather than to traditional MDE tools.

4.2.3 *Clear Separation of End-users and Tool Developers*

Requirement 14 *There must be a clear separation of roles and responsibilities for all actors.*

Closely related to R-13 is the need for a clear demarcation of responsibilities between those applying the new approach and those maintaining and extending it. End-users — the consumers of the approach — should only be required to learn a minimal set of modeling concepts in order to use it. Those working on the approach itself are expected to be competent MDE practitioners.

Note that this requirement does not preclude allowing end-users to get involved with tool development, but merely defines the distinct roles available.

4.2.4 *Prioritise Tooling Integration*

Requirement 15 *The new approach must be designed to integrate tightly with existing tooling and workflows.*

One of the most significant themes to come out of the adoption literature analysis were deficiencies in tooling integration (*cf.* Section 3.3.1). Consequently, the new approach must take special care in this regard, as follows:

- **Reuse input formats**: instead of proposing new native representations for its modeling language, the new approach should focus on specifying mappings (via R-4) to existing modeling languages. These mappings are intended to cover tooling used by developers such as IDEs and build systems, as well as existing modeling tools.

- **Standard error reporting**: the reporting of errors and warnings must be designed in accordance with existing tooling — in particular those which are already well supported in contemporary development environments (Craveiro, 2021d).

- **Minimal dependencies**: in order to facilitate integration, the reference implementation must be available as both as a command-line utility and as a library, with little to no external dependencies (Craveiro, 2021d).

The overall objective of these dimensions is to allow end-users to continue using preferred toolsets and to cause minimal disruption to existing workflows.

4.2.5 *Support Incremental Use of Features*

Requirement 16 *The approach must support different levels of use, from "basic" to "advanced".*

Following on from R-15, end-users must be able to make use of the approach according to their specific needs. These range from code-generation of trivial PSMs (R-8) to the management of products and even entire product lines (R-10), and thus span both top-down and bottom-up application (*cf.* Section 3.3.4). As a result, the domain architecture (R-4) must specify these different levels of use, and define how they are to be supported by the approach and its tooling — with the objective of guiding users towards more advanced uses as their mastery develops.

Last but not least, experienced MDE practitioners should also be able to make use of the approach if, and only if, such use cases do not raise the complexity bar for the target audience (R-3) and are within the remit of the approach's purpose (R-1).

4.2.6 *Conformance Testing*

Requirement 17 *The new approach should include a comprehensive set of conformance tests to validate implementations.*

The reference implementation must be validated by a set of automated conformance tests which determine the level of compliance. These can then be reused by third-party implementations. Conformance tests must cover scalability — *i.e.* expected behaviour for different model sizes — as well as any available integration with third party tooling.

Part II

METHODOLOGY AND COMPONENTS

5 THE MASD METHODOLOGY

Use computer-based tools to mechanize those facets of software development that do not depend on human ingenuity.

— Booch *et al.* (Booch et al., 2004)

THE PRESENT CHAPTER outlines the fundamental characteristics of the Model Assisted Software Development (MASD) methodology, and it is organised as follows. Section 5.1 starts by arguing that the multifaceted list of requirements set out by Chapter 4 can only be addressed by a new SDM, and explains how our methodology differs from typical SDM use cases. Attention is then turned towards the three canonical SDM components: the core philosophy (Section 5.2), the modeling conventions (Section 5.3) and the methodology's processes and actors (Section 5.4). The chapter ends with Section 5.5, where a comparison of MASD against like-minded approaches is performed. Let us begin then by establishing our motivation.

Chapter overview

5.1 MOTIVATION FOR A NEW SDM

After combining literature review with personal experience[1], it is our position that Stahl *et al.*'s two-track iterative software development model (*cf.* Section 2.3.3) did not advance far enough for our needs. In our opinion, problem domain decoupling becomes a significant factor in the context of the requirements gathered in Chapter 4. This happens because the infrastructure problem domain we've set out to explore is simultaneously sufficiently complex and sufficiently distinct from the core problem domain to warrant a clearer demarcation of responsibilities between the two.[2]

Two-track limitations

Accordingly, we argue instead for the logical consequence of Stahl *et al.*'s ideas, which is to have a dual-SDM approach, because one can only mediate such complex interactions by means of an SDM, crafted specifically to handle the infrastructure problem domain, and to ensure it is kept it at arms length from the core problem domain.[3] However, it does not suffice to merely isolate the two problem domains with two distinct SDMs. Using tool vendor offerings as a proxy, our analysis demonstrated they tend to fall in what we named the

Towards dual-SDMs

1 The traditional notion of an SDM was examined in (Craveiro, 2021c) (Chapter 5), as well as its integration with model-driven approaches. (Craveiro, 2021b) covers our personal experiences.
2 *n.b.*, we are not attempting to make a general argument about AC-MDSD; instead, the discussion is restricted to an application of AC-MDSD that targets our specific requirements (*cf.* Chapter 4). Simpler uses need not share this level of complexity, and for those, this analysis would not be relevant.
3 For details of the complex interactions alluded to here, please see (Craveiro, 2021c) (p. 46-47).

over-generalisation trap.[4] Therefore, in order to avoid this fate, it is also crucial for the SDM to explicitly target the "barely general enough" generalisation sweet-spot described towards the end of Section 2.3.3.

Versus traditional SDMs

It is in this sense that the application we put forward differs from the typical integration of MDE with SDMs; its purpose is not to manage the entirety of the software development lifecycle but merely a small but well-defined subset of its activities: those related to the management of Schematic and Repetitive Physical Patterns (SRPPs), which take central stage in the methodology. SRPPs are a furthering of Stahl *et al.*'s ideas around schematic and repetitive code (*cf.* Section 2.3.3), but generalised to other kinds of patterns within the physical space[5]; they result from a deep interrogation on the nature of physical and logical dimensions within a software project, and are discussed in great detail in Chapter 6. This approach is believed to hold the key in shielding end-users from many of the complex themes discussed in Chapter 2, allowing the methodology's modeling conventions — and consequently its domain architecture — to be reduced to only those concepts needed in order to realise its objectives.[6]

Approach summary

The endeavour can now be summarised in a more concrete fashion. MASD is a new SDM designed to aid in the engineering of software systems by fulfilling the requirements previously gathered in Chapter 4. An SDM is needed in order to orchestrate the many complex moving parts into an organised form of collaboration towards a common, well-established goal. MASD makes use of a well-defined subset of MDE techniques, and applies them to the problem domain of infrastructure, striving to attain an adequate level of generalisation. Significantly, MASD has not been designed as a replacement for the application of MDE to the core problem domain, but as a complementary approach with which it can be integrated.

We shall now build upon this motivational framework, and tackle the philosophical considerations that underpin the methodology.

4 MDE vendor tooling is deemed to be a good proxy for the level of complexity of our requirements because they target a generalised problem domain, in the same fashion as we intend to tackle the infrastructure problem domain.

5 MASD employs the terms *physical* and *logical* in the same manner as put forward by Lakos in his systematic study of development artefacts in a software system. He states (*emphasis ours*):

> Developing successful software on a large scale demands a thorough understanding of two distinct but highly interrelated aspects of design: *logical* design and *physical* design. Logical design, as we will use the term, address all *functional* aspects of the software we develop. [...] Physical design [...] addresses issues surrounding the placement of logical entities, such as classes and functions, into physical ones, such as files and libraries. *All design has a physical aspect*. That is because all of the source code that makes up a typical C++ program resides in files, which are physical. (Lakos, 2019) (p. 44)

Lakos appears to be quite unique within the literature for his incessant questioning of compile time physical design, and its implications to logical design. Regrettably, a literature review of Lakos' copious material is unfeasible, given the space and time constraints of the present work. The interested audience is directed to Lakos multi-volume opus as our preferred recommendation (Lakos, 2019). However, as far as this dissertation is concerned, the essence of the argument had already been made in the prior incarnation of the work (Lakos, 1996), of a more modest size.

6 It may be argued that the UML has long had provisions for a physical view, distinct from the logical view — *e.g.*, (Stevens and Pooley, 1999) (p. 155), (Booch, Rumbaugh, and Jacobson, 1999) (p. 343). Nonetheless, in stark contrast to our intentions, the physical view has often been relegated to deployment as well as runtime concerns, whereas our focus is on compile-time artefacts. This disconnect forced MASD to turn its gaze towards Lakos.

5.2 PHILOSOPHY

In order to plot a clear and concise narrative for MASD, we have divided its philosophy into three distinct but interdependent concerns: a *vision*, a *mission statement* and a set of fundamental *core values*.[7,8] The vision provides MASD with a overarching purpose, setting the overall direction for the methodology. The mission statement is comprised of a set of actions that will be taken to fulfil the vision. Finally, the core values are the basic principles that act as guides or sign-posts for all decision making on the long road towards making the vision a reality. The next three sections describe these elements in detail.

Components

5.2.1 *Vision*

MASD's vision is as follows:

> To accelerate MDE adoption by traditional software engineers for the generation of infrastructural code.

5.2.2 *Mission Statement*

MASD's mission is to provide a systematic and integrated approach to the lifecycle management of SRPPs, a class of patterns found in software systems. The lifecycle management of these patterns calls for:

Scope of SRPP management

- their identification, capture and classification in a open, centralised and continuously evolving library;
- the analysis and characterisation of their inter-dependencies and variability requirements;
- their generation by automated means;
- the associated conformance testing, required to empirically verify and validate their generation;
- their evolution, as related use cases are considered; and, ultimately,
- their decommission, once no longer required.

7 A literature review on vision and mission statements is deemed to be beyond the scope of the present work. The interested reader is directed to Stallworth Williams (Stallworth Williams, 2008) for an accessible treatment.

8 In our opinion, setting out a clear vision and mission statement for MASD is crucial due to the deficit in clarity that surrounds MDE's variants (Craveiro, 2021c) (p. 10).

5.2.3 *Core Values*

Overview

MASD's core values are anchored on six complementary principles, all centred around increasing automation, as depicted in Figure 5.1. The next six sections discuss each principle in turn, anti-clockwise, supplying context and a rationale for their inclusion. Each principle is composed of a numbered *principle definition* for cross-referencing purposes[9], a list of the associated requirements it addresses (*cf.* Chapter 4) and an overview detailing the objective of the principle. Note that the principles are stated in imperative form by design, as they are intended as exhortations to MASD actors (*cf.* Section 5.4).

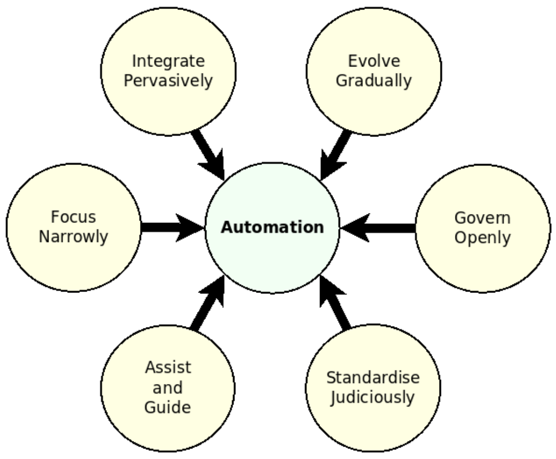


Figure 5.1: MASD principles.

5.2.3.1 *First Principle: Focus Narrowly*

Principle 1 *MASD has a* narrow focus *on its problem domain, with well-defined identity, boundaries, audience and responsibilities.*

***Related Requirements**: R-1, R-2, R-3, R-12, R-13.*

Single focus, user base

MASD has a single focus on solution space concepts — given that's where software infrastructure resides — and commits itself to only serving one set of users: software engineers. With a narrower focus comes a smaller conceptual framework and hence a smaller cognitive load, making it suitable for new practitioners. In addition, a well-defined scope also provides a more straightforward filtering function with which to circumscribe the methodology's boundaries.[10]

MASD is able to externalise a large subset of modeling concerns because its only focus is the code generation of infrastructural code. Therefore, all

9 *e.g*, P-1 references the first principle, defined in Section 5.2.3.1.

10 Besides the lessons learned from MDE, MASD's narrow focus is also a product of our research into special purpose code generators (Craveiro, 2021d) (Section 3.1), as well as the incredibly insightful comments by Clark and Muller regarding the industry's search for a "shrink-wrapped black-box process" that hides its MDE internals (Clark and Muller, 2012) (*cf.* Section 3.3.1)

other functionality related to modeling is deemed to be external to MASD's domain, such as support for graphical notations, DSLs, MTs, model evolution, model synchronisation, reverse engineering and many other aspects of MDE. Where required, these must be addressed elsewhere; and their integration with MASD is the role of the next tenet.

Externalisation

5.2.3.2 *Second Principle: Integrate Pervasively*

Principle 2 *MASD adapts to users' tools and workflows, not the converse. Adaptation is achieved via a strategy of* pervasive integration.

Related Requirements*: R-4, R-13, R-15.*

MASD promotes tooling integration: developers preferred tools and workflows must be leveraged and integrated with rather than replaced or subverted. First and foremost, MASD's integration efforts are directly aligned with its mission statement (*cf.* Section 5.2.2) because integration infrastructure is understood to be a key source of SRPPs.[11] Secondly, integration efforts must be subservient to MASD's narrow focus (P-1); that is, MASD is designed with the specific purpose of being continually extended, but only across a fixed set of dimensions. For the purposes of integration, these dimensions are the projections in and out of MASD's TS, as Figure 5.2 illustrates.[12]

Role of integration within MASD

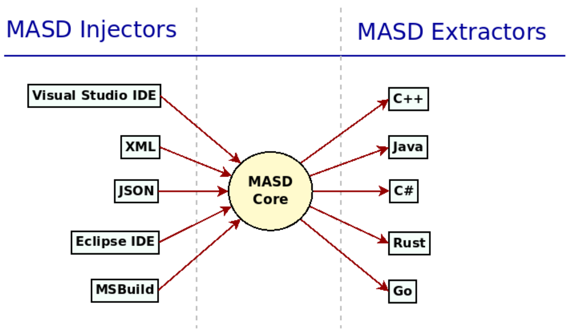


Figure 5.2: MASD Pervasive integration strategy.

Within these boundaries, MASD's integration strategy is one of pervasive integration. MASD encourages mappings from any tools and to any programming languages used by developers — provided there is sufficient information publicly available to create and maintain those mappings, and sufficient interest from the developer community to make use of the functionality. Significantly, the onus of integration is placed on MASD rather than on the external tools, with the objective of imposing minimal changes to the tools themselves. To demonstrate how the approach is to be put in practice, MASD's research includes both the integration of org-mode (*cf.* Chapter 7), as well as a survey on the integration strategies of special purpose code generators (Craveiro,

Integration strategy

11 In other words, much of the machinery required for integration is believed to be schematic and repetitive in nature. This belief is justified empirically via the author's two decades of industrial software development, as well as the experiences in MDE application (Craveiro, 2021c).

12 See Chapter 4 of (Craveiro, 2021c) (p. 31) for details on Technical Spaces (TSs) and associated projections.

2021d); subsequent analysis generalised these findings so that MASD tooling can benefit from these integration strategies. Undertakings of a similar nature are expected as the tooling coverage progresses.

Limitations

Clearly, if left to its own devices, pervasive integration could be construed as an unachievable target due to its overly ambitious scope — particularly given the very large size of the pool of potential integration candidates. Moreover, this is a problem that affects the capture of SRPPs in general. It is the role of the next principle to provide direction for the exploration of such an immense problem space.

5.2.3.3 *Third Principle: Evolve Gradually*

Principle 3 *MASD is designed to grow gradually and deliberately over time, covering an increased surface area of its problem domain.*

Related Requirements*: R-4, R-5, R-10, R-11.*

Problem space landscape

MASD sets out to create a conceptual framework for the exploration of the problem space of infrastructural code; in the limit, it views all infrastructural code as part of one single but extremely large problem domain — allowing for a globalised view — and its objective is to identify and model its entities along with their associated variability. Thus, MASD sees the processes related to the continual discovery of the shape of the problem domain as a indissoluble part of the methodology, as is the resulting library of patterns of infrastructural code, and strives for comprehensive coverage over long timescales.

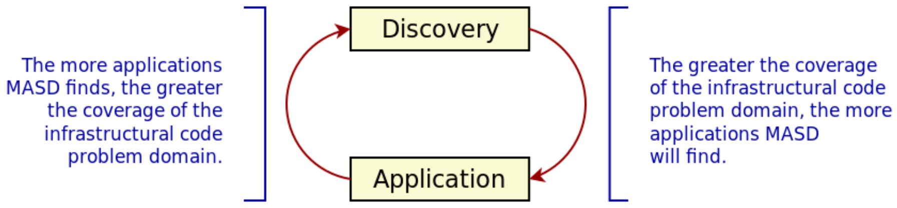

The more applications MASD finds, the greater the coverage of the infrastructural code problem domain.
Discovery
Application
The greater the coverage of the infrastructural code problem domain, the more applications MASD will find.

Figure 5.3: Interrelationship between discovery and application.

Incremental growth

The exploration of infrastructural code is made through empirical and iterative means, and guided by pragmatism. In this manner, MASD expects to fulfil its vision through a very large number of gradual steps over a long period of time, slowly and incrementally building up its SRPP library in a cohesive manner — which also includes integrations (P-2). The driver for changes is *practice*; that is, in general, features in MASD must be driven by concrete use cases from users in the field rather than through the inclusion of speculative features.[13,14]

13 Nonetheless, there is a significant caveat: MASD's reference implementation is a key end-user of the methodology, meaning that features will be specifically added for this purpose, even if they have no external use cases.

14 Chapter 3 alluded to the challenges posed by making "too many" features available to end users. For a more specific example of the dangers in adding speculative features to MDE tooling, see our own experience report (Craveiro, 2021b) (Section 6.3).

However, even taking long timescales into account, such a large undertaking can only be performed by leveraging an equally large and motivated group of individuals. This in turn raises issues of community and organisation, addressed by next principle.

5.2.3.4 *Fourth Principle: Govern Openly*

Principle 4 *MASD promotes an open governance model because a thriving community is a necessary condition to fulfil its vision.*

***Related Requirements**: R-5, R-7, R-14.*

Community relevance

Whilst all work on MASD thus far was performed by the dissertation's author, careful consideration has been given to creating a governance model that will allow external parties to contribute.[15] These contributions are viewed as an essential ingredient for the fulfilment of the methodology's vision (*cf.* Section 5.2.1), because, from a end-user perspective, MASD's usefulness is closely related to the breadth and depth of its SRPP pattern library, and the library can only grow if the methodology is pitted against the full breadth and depth of its target problem domain.

Precedents

These lofty aspirations are not without precedent. MDE has a long-standing experience with FOSS development in leading projects such as EMF (Steinberg et al., 2008), Generic Modeling Environment (GME) (Davis, 2003) and many others; in the most successful cases, the approach has yielded several positive results such as a growing and diverse community as well as source code reuse — characteristics which we seek to emulate.

Governance model

Benefiting from their experience, we thought important to structure MASD in a fashion similar to a FOSS project, in anticipation of its expected future.[16] Once chosen, the governance model had implications on actors and processes, so that we could mirror more closely the structures typically found in FOSS — *i.e.* roles such as *maintainer*, *contributor*, *end-user* and so forth can be directly mapped to MASD's actors and processes defined in Section 5.4.

A plurality of voices has benefits but is not without its hazards, requiring careful orchestration to ensure all involved pull in the same direction. Therefore, in addition to P-1, we also set a clear stance on the use of standards, as the next principle will demonstrate.

15 Availing themselves of Renz's ideas on the matter (Renz, 2007), Capra *et al.* define governance in this context as follows: "The governance of software projects is defined as the complex process that is responsible for the control of project scope, progress, and continuous commitment of developers. Governance is recognized to have a key role in enabling software project success [...]." (Capra, Francalanci, and Merlo, 2008)

16 A review of the available literature was carried out before embarking in this direction, but it is deemed to lay outside the scope of the dissertation. For those interested, we relied mainly on Crowston *et al.*'s comprehensive survey of the literature on FOSS development, spanning over 135 published empirical studies and including both quantitative and qualitative methods (Crowston et al., 2012). The review highlighted several points of interest, but the findings most relevant to the present work relate to the different roles in FOSS projects. Once these were identified, Lee and Cole's analysis was used to attain an increased understanding of their properties (Lee and Cole, 2003).

5.2.3.5 *Fifth Principle: Standardise Judiciously*

Principle 5 *MASD employs* de facto *standardisation at its core to promote agility, and* de jure *standardisation at the edges to ensure stability.*

***Related Requirements**: R-1, R-2, R-4, R-11.*

de jure standards

After reflecting on the MDA experience with international standards (*cf.* Section 2.3.2), we decided to limit the use of *de jure* standards in MASD to projections in and out of MASD's TS (P-2), as that is where we see them adding most value. Projections from and to the outside world are expected to remain stable, and are aligned to the target of the projection, so it makes sense to use well-known standards where those are available. Conversely, MASD's core shall be restricted to *de facto* standards because, there, empiricism and agility are deemed more important than stability.

de facto standards

Within the core, *de facto* standards are based on two pillars: this dissertation, as the basis for the definition of the methodology in a future document called the MASD SDM Specification (MSS); and the MASD Reference Implementation (MRI) (*cf.* Chapter 8) as the single source of truth of everything else — tooling interfaces, the SRPP library of patterns and associated variability, and the like. The duality is justified as follows. In order to remain relevant to its users, the MRI is expected to undergo constant change (P-3), making it a good candidate for an Agile process (Beck et al., 2001).

Philosophical changes

On the other hand, the SDM itself is expected to experience only minor revisions, reconcilable with the vision and principles here put forward. This does not mean the MSS is frozen; change is strongly encouraged in MASD and the philosophy is itself liable to change just as much as any other of its components — provided there is sufficient justification. However, it is important to understand that, by design, the identity of MASD is deeply embedded within its philosophy. Therefore, we anticipate it to change in a small and incremental manner, maintaining a similar direction to what has been proposed by this document — or else for a completely new (and distinct) SDM to be put forward in its place.

This multi-layered approach with regards to standards is designed to provide an adequate support for the speed of change of each of these aspects, in order to better serve the target audience. That said, the target audience is multi-layered as well, and thus demands support for distinct levels of usage. And that is the role of the next and final principle.

5.2.3.6 *Sixth Principle: Assist and Guide*

Principle 6 *MASD's role is to* continually assist *its end users in choosing the appropriate level of automation for their projects.*

***Related Requirements**: R-6, R-8, R-9, R-10, R-16.*

MASD is designed from the ground up to support both top-down and bottom-up approaches, and is focused on identifying a set of levels of usage that mirror the behaviour uncovered from adoption literature and personal experience. As a result, instead of enforcing a model-driven view of software engineering, MASD views the use of automation in the development of a software system as a spectrum of possibilities (Figure 5.4), ranging from no automation to the automated generation of all infrastructural code, as the full MASD vision is eventually realised.[17] All points in the spectrum are equally valid and a system may be composed of an heterogeneous mix of automation approaches, both from within and outside of MASD — though hopefully orchestrated via its strategy of pervasive integration (P-2).

Automation spectrum

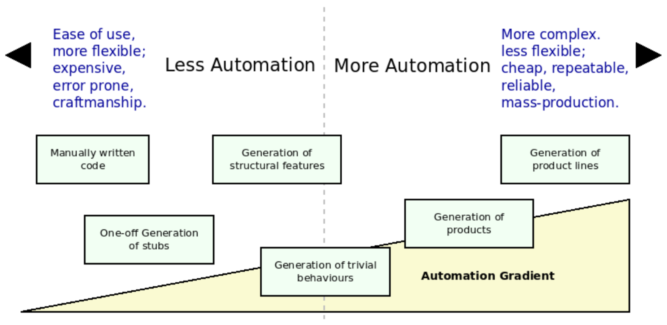


Figure 5.4: The automation spectrum and the automation gradient.

MASD aims to empower its practitioners in determining the adequate level of automation for a given context, as a function of their prior experience. Over time, as they master methodology and tooling, practitioners are expected — but not forced — to progressively climb the automation gradient, though always remaining within the narrow confines of an AC-MDSD approach. Crucially, MASD does not promote the use of automation as a uniquely positive development regardless of context, but rather views it as a set of engineering trade-offs that must be made during the software development process. Though *assisted* by the methodology, it is ultimately the practitioner's responsibility to make those trade-offs.[18] MASD views software engineering not as driven by modeling but by the manual writing of code; modeling is considered a subsidiary activity that can be of assistance to the development process in distinct but well-defined capacities, within the scope of infrastructural code.[19]

Model-driven vs model-assisted

And it is with this message of end-user focus that we conclude our incursion through MASD's philosophy. It's also worthwhile noting that the methodology's philosophy has had far reaching implications to all the work carried out; for one, it was instrumental in shaping MASD's modeling language and other related aspects of its domain architecture. And it is to these we shall turn our attentions to next.

17 The MASD automation spectrum is inspired on Groher and Völter's's analysis of variability and their modeling of it as a continuum (Groher and Voelter, 2007). Figure 5.4 in particular is our take on their idea of the "Expressive power of DSLs" as a spectrum of possibilities. The automation gradient was inspired on Bosch *et al.*'s work on variability (Bosch et al., 2001).

18 Hence why the adjective *assisted* was chosen, as opposed to the more traditional choices such as *oriented*, *driven*, *based* and the like, as touched upon in (Craveiro, 2021c) (p. 10, note 5).

19 At this juncture, one can begin to see the key differences between MASD and model-driven methodologies emerging, and it is certainly a topic worthy of further elaboration; it is addressed at the end of the chapter (*cf.* Section 5.5).

5.3 MODELING CONVENTIONS

Versus domain architecture

The second aspect of an SDM are its conventions with regards to modeling. In our view, the notion carries additional weight within MDE application, because modeling conventions manifest themselves as a modeling language, and the modeling language is but one component of the domain architecture. All components of the domain architecture must be designed to work in concert and cannot be understood in isolation, leading us to defer their wholesale exposition to Chapter 6. Therefore, the present section is tasked exclusively with painting the backdrop with which to understand the domain architecture — *i.e.*, elucidating the conventions and motivation in the MASD's modeling landscape. And it is here that DDD, as championed by Evans (Evans, 2004), takes central stage.

Ubiquitous language

Though making selective use of the approach, MASD was nonetheless greatly influenced by DDD, particularly with regards to modeling conventions. A clear example is MASD's ubiquitous language[20]: it is explicitly composed of both an informal model and a formal model, as depicted in Figure 5.5, and we view it as a significant statement of intent with regards to the limits of a formal modeling approach. As does Evans, we do not believe it is possible — or even desirable — to express all entities in MASD's problem domain formally. Furthermore, when an entity is captured formally, effort must be made to ensure only those aspects needed to recreate the associated SRPPs are modeled. That is, MASD seeks to keep the footprint of the formal model as small as possible because, from experience, we have found its maintenance to be costly (Craveiro, 2021c). Since the formal model does not tell the whole story, the informal model is then responsible for weaving an all-encompassing narrative that imbues entities with meaning.

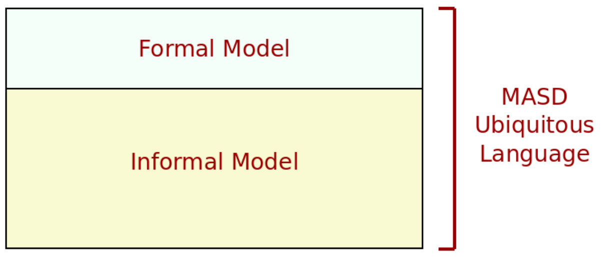


Figure 5.5: MASD modeling language.

That said, the MASD modeling landscape is somewhat convoluted at present; due to this, its distinct moving parts and their relationships are perhaps best explained from a high vantage viewpoint.[21] Figure 5.6 provides a birds-eye view of the top-level components and their associations, which we shall now describe. As mentioned previously (*cf.* Section 5.2.3.5), the MSS is a subset

20 Evans states (*capitalisation his*):

> The vocabulary of that UBIQUITOUS LANGUAGE includes the names of classes and prominent operations. The LANGUAGE includes terms to discuss rules that have been made explicit in the model. It is supplemented with terms from high-level organising principles imposed on the model [...]. Finally, this language is enriched with the names of patterns the team commonly applies to the domain model. (Evans, 2004) (p. 23)

21 Brining order to this complex picture will be part of our future work.

of the present document, containing only the aspects relevant to the MASD SDM. The MRI (*cf.* Chapter 8) is a software product family composed of several software products: a code generator and a set of reference products — at present, one per supported TS. There is only one implementation of the code generator, called *Dogen* and described in Section 8.1. The formal model is specified and implemented within the code generator and it is used to document and generate the code generator itself, as well as for testing its functionality. The reference products (*cf.* Section 8.2) also serve a dual purpose, being used for both conformance testing as well as canonical instances of the domain architecture for demonstrative purposes. Finally, the informal model is scattered across the MSS and the MRI — in particular, the developer documentation included in the code generator's source code repository.

Top-level components

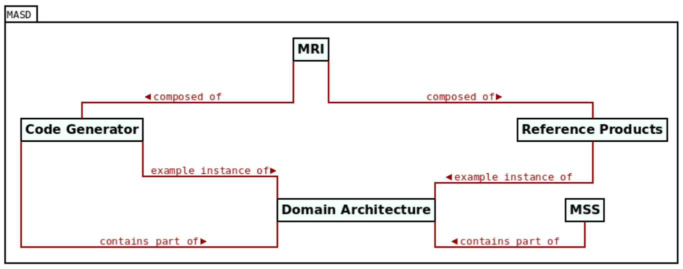


Figure 5.6: MASDs top-level components.

Given this context, it should by now be clear that MASD views the determination of its modeling conventions as a dynamic process. Consequently, the manner in which the domain is explored is of key importance to the methodology, as are those responsible for performing the exploration; and it is to them we shall turn to in the next section.

5.4 PROCESSES AND ACTORS

The third and final component of MASD as an SDM are its processes and associated actors. Accordingly, both the application of MASD as well as its internal development are governed by well-defined processes and their corresponding actors playing well-defined roles. This section provides an overview of both actors (Section 5.4.1) and processes (Section 5.4.2), linking them back to the concepts defined thus far.

5.4.1 *Actors*

Within MASD, there are three distinct personas with different responsibilities: the *maintainer*, the *developer* and the *user*.[22] These represent the canonical roles available in MASD software development. The next three sections shed light on these actors.

22 To avoid conflating these names with the more general usage of the terms in a FOSS context, we have henceforth prefixed them with MASD (*e.g.*, the *MASD maintainer*).

5.4.1.1 *MASD Maintainer*

Role

The MASD Maintainer is the gatekeeper of the MSS and the MRI. Its role is to review change requests, often made as pull requests to the desired component, ensuring they are consistent with the methodology's philosophy (*cf.* Section 5.2) and fit the MRI codebase. All changes that impact the MRI must have a set of tests that specify and validate empirically the new behaviours, in order to be considered for inclusion; they must therefore update all relevant reference products. Once the changes have been reviewed and approved, the MASD Maintainer is responsible for merging them and releasing new versions of the affected components.

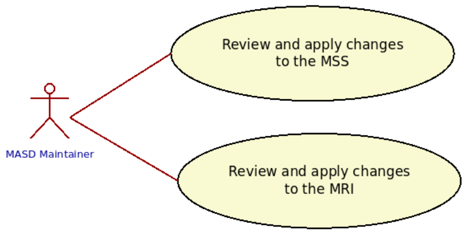

Review and apply changes to the MSS
MASD Maintainer
Review and apply changes to the MRI

Figure 5.7: Use case diagram for MASD Maintainer.

Changes and compatibility

Since MASD is continuously evolving, the management of backwards and forwards compatibility is a significant challenge for the MASD Maintainer. In general, maintaining compatibility should be the main priority, unless there are explicit technical reasons not to do so. In cases where compatibility must be broken, the MASD Maintainer will communicate the change to its users by means of semantic versioning (Preston-Werner, 2018), supplying detailed examples on how to update legacy models to the newest version.[23] The reference products are used for this purpose, demonstrating the before and after states.

5.4.1.2 *MASD Developer*

Role

The MASD Developer is responsible for creating change requests for the MSS and the MRI, in response to demands for new features or fixes to existing features. The MASD Developer is expected to be familiar with state of the art approaches in MDE as well as MASD itself, and to use those selectively, in accordance with the MASD vision.

Example activities

As an example, the MASD Developer can create mappings for new platforms, add new injectors or extend existing ones with new functionality, add new modeling elements to the MASD metamodels, target new TSs or augment

23 Stahl *et al.* suggest avoiding this problem altogether by always maintaining backwards compatibility: "The DSL typically continues to be developed in the course of one (or more) projects. The knowledge and understanding of the domain grows and deepens, so the DSL will thus be extended. To make life simpler, one must make sure that the DSL remains backwards-compatible during its evolution." However, this approach limits the evolution of the DSL so MASD forfeits it. Nevertheless, breaking compatibility should be seen as the alternative of last resort.

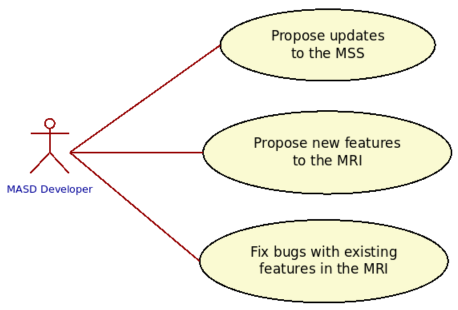

MASD Developer
Propose updates to the MSS
Propose new features to the MRI
Fix bugs with existing features in the MRI

Figure 5.8: Use case diagram for MASD Developer.

the features on the already supported TSs and so forth. All changes must be submitted and reviewed by the MASD Maintainer.

5.4.1.3 *MASD User*

Role As a consumer of the MASD methodology, the MASD User is a traditional software developer who applies MASD to the development of a software system. The typical use cases for the MASD User is to create new models and to transform them into code. However, as MASD is continuously evolving, users are also encouraged to identify limitations and opportunities to extend it.

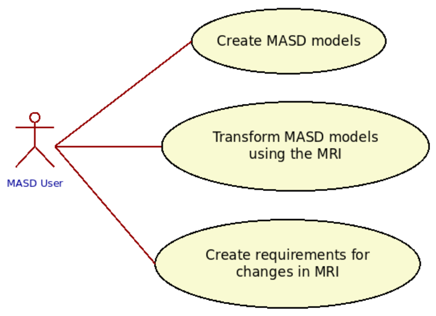

MASD User
Create MASD models
Transform MASD models using the MRI
Create requirements for changes in MRI

Figure 5.9: Use case diagram for MASD User.

Extending MASD MASD Users communicate their requests for new features to MASD Developers, who are then responsible for its development. Change requests must include a Minimum Working Example (MWE) demonstrating the proposed change. The MWE is used as a target for the reference products and will form the basis of the conformance tests used to ensure the new change was implemented correctly. The MWE will make its way into the code base via the appropriate processes, as described in the next section.

5.4.2 *Processes*

There are three processes within MASD catering for both the internal development of the methodology as well as its application: the *MRI Development Process* (Section 5.4.2.1), the *MSS Development Process* (Section 5.4.2.2) and the *MASD Application Process* (Section 5.4.2.3). All these processes are then combined under the *MASD Composite Process* (Section 5.4.2.4). The next four sections describe these processes in more detail.

5.4.2.1 *MRI Development Process*

SRPP discovery

The MRI is a dynamic software product family whose evolution takes place via the *MRI Development Process*. A simplified version of the process is shown on Figure 5.10. MASD Users and MASD Developers are responsible for identifying patterns in source code deemed to be SRPPs. The identification process must include a distilling of the schematic and repetitive structure into a MWE so that the fundamental characteristics of the pattern can be isolated from irrelevant content. If the isolated patterns are not yet covered by the MRI, MASD Developers act according to Test-Driven practices and start by updating the impacted reference products to match the desired result.

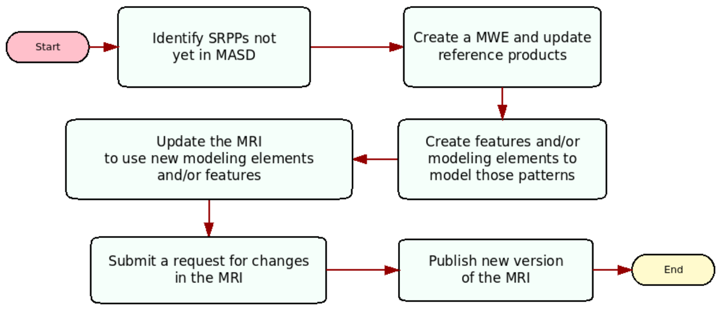

Start
Identify SRPPs not yet in MASD
Create a MWE and update reference products
Create features and/or modeling elements to model those patterns
Update the MRI to use new modeling elements and/or features
Submit a request for changes in the MRI
Publish new version of the MRI
End

Figure 5.10: MRI Development Process.

SRPP incorporation

MASD Developers then attempt the automated generation of the reference products, which now incorporate the MWE, by making changes to the MRI until all tests pass. Often, the code generation of the requested changes forces a cascading set of modifications across the MRI metamodels, in order to reflect the introduction of the new feature throughout the stack. However, simpler cases may be able to make use of the existing infrastructure.

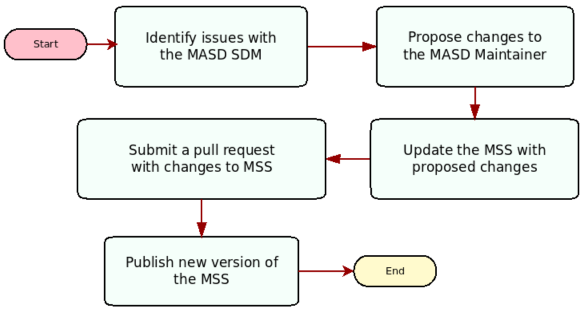
Implementation of new features

Once the MWE has been successfully reproduced and both MASD User and MASD Developer are in agreement with how the feature has been implemented — *e.g.* configuration settings, impact on existing features, and so on — the changes are then proposed to the MASD Maintainer as a change request to the affected MRI repositories. The MASD Maintainer reviews and validates the changes and, once approved, publishes the new versions of the impacted software products. Via this process, the MRI is expected to evolve and grow over time, eventually providing a rich SRPP library of out of the box.

5.4.2.2 *MSS Development Process*

Overview

As described in Section 5.2.3.5, changes to the MSS are expected to be few and far between. Nonetheless, for completeness, a process catering for these changes is provided — as illustrated by figure 5.4.2.2. Once a shortcoming to the MASD methodology is located and discussed with the MASD Maintainer, the instigator of the change is expected to create a patch to the MSS as a change request. The MASD Maintainer will then apply the change and release a new version of the MSS.


Start
Identify issues with the MASD SDM
Propose changes to the MASD Maintainer
Update the MSS with proposed changes
Submit a pull request with changes to MSS
Publish new version of the MSS
End

5.4.2.3 *MASD Application Process*

Overview

MASD Users apply the methodology in the development of software systems by using the MRI to generate code for their system. Before they can do so, they must first settle on the appropriate injector to use — *e.g.* IDE or preferred modeling tool support if those injectors are available in MASD, or otherwise a simpler format such as JSON — and on a MASD *application level*. The application level will determine how much MASD will influence the development process, via P-6. There are four levels of application, with an increasing level of dependency in MASD:

Application levels

- **Level 0: Stub and Skeleton Generation**. The MRI is used as a one-off generator to create stubs for methods, skeletons for classes and the like. Once the code is generated, it must be manually maintained.
- **Level 1: Artefact Generation**. The MRI acts like a special purpose code generator (Craveiro, 2021d), albeit with a potentially wider remit given the feature set of MASD (*cf.* Chapter 6). The project structure, integration of generated artefacts and other development decisions are unaffected by MASD.
- **Level 2: Component Generation**. MASD is used to model and generate one or more components in a larger system, such as a library or an executable. The component follows the MASD component directory structure layout and may use MASD generated build files. However, the remaining system remains unaffected.

- **Level 3: Product Generation**. MASD is used to model and generate an entire product — *i.e.* a complete software system, composed of executables and libraries. In this case, all (or most) components that make up a product are modeled and generated using MASD, with the user writing code to implement problem space specific behaviour. The product follows all of MASD's conventions, including its product directory structure layout.

- **Level 4: Product Line Generation**. MASD is used to model and generate a family of related products.

Determining the application level

Typically, new MASD Users start at *Level 0* and explore the methodology incrementally — use more features, model different types of elements, *etc.* Over time, as they gain experience, they are expected (but not required) to progress towards *Level 4*. Experienced MASD Users may then decide that MASD is too limiting for them and move to a more traditional MDE modelware stack, or work on integrating existing MDE tooling with MASD.

Process description

Once the level of application has been decided upon, the process of application itself is straightforward, as Figure 5.11 depicts. It works as follows: MASD Users identify the elements to model and the features that are of interest. If those features are already available in MASD — as implemented by the MRI, and explored in more detail in Chapter 6 — users can freely create their models, generate code and consume the generated code from handcrafted artefacts. As with most development today, the process is an iterative one, with identification and modeling progressing side-by-side with traditional programming. If the required features are not available, the MASD User is encouraged to report it via the MRI Development Process (*cf.* Section 5.4.2.1).

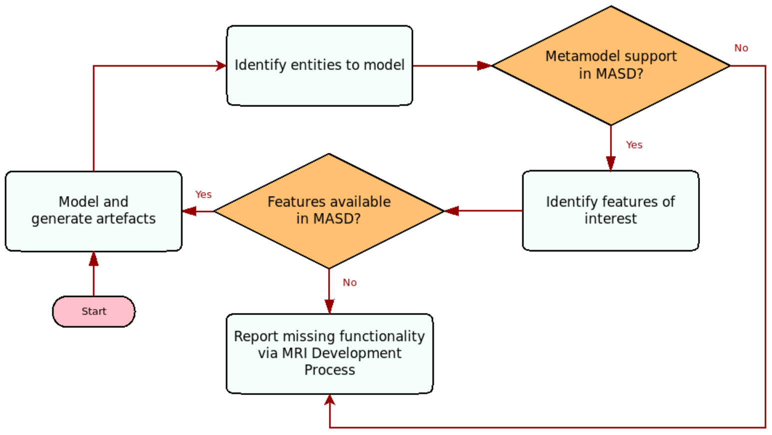

Identify entities to model
Metamodel support in MASD?
No
Yes
Model and generate artefacts
Yes
Features available in MASD?
Identify features of interest
No
Start
Report missing functionality via MRI Development Process

Figure 5.11: MASD Application process.

Importance of application

The simplicity of the workflow described above may not adequately convey its significance within the methodology, so it is worthwhile stating explicitly that all moving parts of MASD exist solely to service the MASD application process. Furthermore, application and empiricism are the main drivers for MASD — an ideal we tried to enshrine within its core values (*cf.* Section 5.2.3). This notion of interconnectivity between processes in MASD can be modeled at a higher level of abstraction, giving rise to a composite process.

5.4.2.4 *MASD Composite Process*

Bootstrapping, dogfooding

MASD's development has been characterised by severe resource constraints — more so than any of our previous experiences with MDE (Craveiro, 2021c). Due to this, there has been an emphasis on doing more with less, forcing us to apply a strategy of bootstrapping — that is, using the MRI to develop the MRI itself — and dogfooding — a generalisation of bootstrapping, applying it not just tooling but to MASD itself; in other words, using the methodology to develop the methodology.[24] The next obvious step in this generalisation ladder is the notion of a *MASD Composite Process*, using MASD's processes for its own development.[25] Figure 5.12 illustrates the idea.

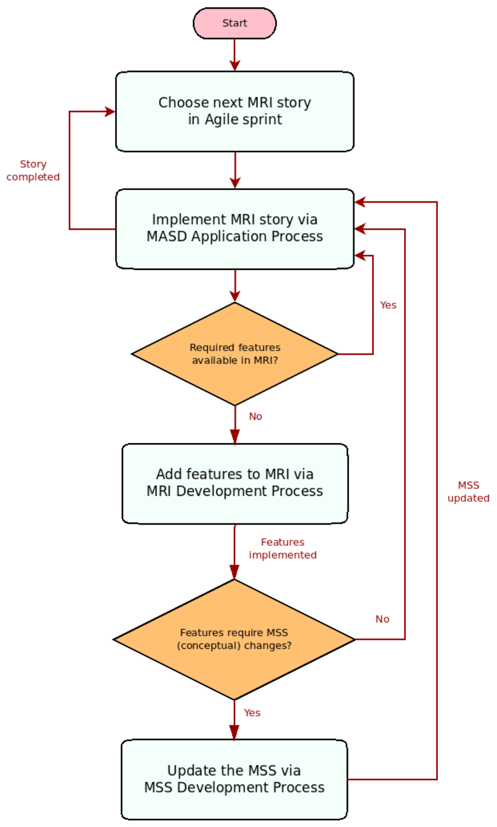

Start
Choose next MRI story in Agile sprint
Story completed
Implement MRI story via MASD Application Process
Yes
Required features available in MRI?
No
Add features to MRI via MRI Development Process
MSS updated
Features implemented
No
Features require MSS (conceptual) changes?
Yes
Update the MSS via MSS Development Process

Figure 5.12: MASD Composite Process.

Flowchart description

The centrepiece of the flowchart is the use of MASD Application Process (*cf.* Section 5.4.2.3) to drive the development of MASD (*i.e.*, the MRI and the MSS). Since the MRI is a non-trivial software product, it allows us to realistically emulate the traditional two-track environment one would find in the industry, without having to absorb the cost of simultaneously developing two distinct

24 Harrison states: "[T]he idea that someone would use the products they were making became known as 'eating your own dog food.' [...] Regardless of its genesis, the software industry has adopted the phrase to mean that a company uses its own products. Somewhere along the line, the noun 'dog food' appears to have morphed into a verb." (Harrison, 2006)

25 Further theoretical work is required with regards to the classification of this composite process. An argument could be made in favour of naming it a meta-process, which Rolland *et al.* define as "[...] a process for the construction of a process model." (Rolland, Prakash, and Benjamen, 1999) The composite process could be said to drive the underlying processes, but as it is not acting as a constructor for its components, we decided against this approach.

software products.[26] In addition, by forcing the MRI to travel through the same four levels of the MASD Application Process we expect end users to travel, we obtain first-hand knowledge of the benefits and pitfalls of the process. More generally, this recursive application forces us to gain direct experience on the roles enacted by all actors (*cf.* Section 5.4.1), as well as all processes (*cf.* Section 5.4.2), meaning we can ensure they are fit for purpose.[27]

The nature of the MASD Composite Process means that all of our application case studies (*cf.* Part iii) will have two distinct levels of application:

- **Application level**: that is, the use of MASD as a regular user. Conclusions at this level will be applicable to any user of the methodology and/or the MRI.

- **Meta-Application level**: that is, the impact of this particular application on the methodology itself.

Fitness function

Finally, the Composite Process also supplies a very useful fitness function with which to measure progress for both the methodology and MRI: the objective of the MRI is to be able to generate itself, via the MASD Application Process, at Level 4 (product line generation), without requiring further changes to MASD. Achieving this milestone marked the end of the development of MASD's basic framework.

With this words we complete our analysis of MASD's processes and actors. The picture painted thus far is quite distinct from typical applications of MDE. The next section will bring the chapter to a close by elaborating on these differences.

5.5 COMPARISON WITH OTHER APPROACHES

Versus MDE

The previous sections have outlined a system of beliefs which underlies all thinking within MASD. Before proceeding, it is worthwhile comparing MASD's beliefs against others already analysed in Chapter 2, with MDE being the obvious starting point.[28] In contrast with the latter, MASD's vision is

26 Using the MRI as *a model* for a typical software product is in the Rothenberg spirit (Craveiro, 2021c) (p. 16), for he had stated: "Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose." (Rothenberg et al., 1989) This is precisely our intent.

27 The MASD Composite Process is a translation of compiler bootstrapping to MASD. On the former, Sjölund *et al.* explain:

> One of the advantages [of compiler bootstrapping] is assumed to be higher quality since the designers and developers of a language and its compiler will be major users, and therefore will be highly motivated to correct possible design flaws and errors. Another advantage is portability — the bootstrapped compiler is primarily dependent on itself, not on other languages, once it has been bootstrapped. [...] Bootstrapping means that the language and its compiler is defined and implemented using itself.

28 In so far as only one philosophy exists for MDE. For the purpose of this exercise, we can consider the statements by France and Rumpe in Section 3.2.3 as representative of the MDE vision. However, the subject is by no means uncontroversial. This, we posit, is yet another side-effect of MDE's role as a diverse and constantly evolving body of knowledge (Craveiro, 2021c) (Chapter 2).

carefully designed to be compact in scope. One might go as far as saying it is only small subset of the overall MDE vision, and with a different direction of travel: where MDE shows a tendency towards increased rigour, abstraction and completeness[29], MASD tilts the balance towards empiricism and engineering. This focus on the small is a fundamental characteristic of MASD and is also in line with our findings about MDE's state of practice (*cf.* Chapter 3). In this sense, MASD's use of MDE is seen mostly as an implementation detail. This is a subtle but crucial point so perhaps worthy of further elaboration.

Problem space / solution space inversion

Just as MDE's vision is to insulate domain experts from the intricacies of the implementation details, allowing them to focus on the problem space and narrowing the problem-implementation gap, so does MASD use MDE itself to insulate its domain experts — *i.e.*, software engineers — from the implementation details — that is, the MDE techniques used by MASD, as well as the solution space implementation details — in order to allow them to focus on their problem space — the engineering of software systems — therefore reducing their problem-implementation gap: the distance between models of infrastructural code and infrastructure code itself.

Versus MDA

On the face of it, this distinctive focus on software engineers and the solution space may appear to confront MDA's calls for direct representation (*cf.* Figure 2.1). MASD's intent is not to make any statements with regards to the validity of either of these approaches, but merely to clarify the distinctiveness of their scopes. There is thus a need to integrate MASD with other MDE based approaches, such as MDA, in order to cater for direct representation and the problem space.

Versus AC-MDSD

The relationship between MASD and AC-MDSD is also of great relevance. Whilst MASD can be thought of as one possible realisation of the AC-MDSD principles, it is also designed to address our deep-seated concerns with naive applications of the approach (*cf.* Section 2.3.3). What is novel about MASD, when compared against AC-MDSD, is its view of infrastructural code as a problem domain on its own right, worthy of study by employing MDE techniques. That is, unlike typical AC-MDSD solutions, which are customised for the specifics of a given problem, the objective of MASD is to find solutions that are applicable to the problem of infrastructural code in general.

Generalisation, variability

This is a very important conceptual leap because we no longer view infrastructure code as a system-specific attribute, with perhaps some aspects that are generalisable; instead, we view it as a completely generalisable problem domain, which requires a degree of customisation for specific systems. Where AC-MDSD sees a localised solution that may only share few commonalities with other localised solutions, MASD sees it merely as the expression of variability between two different MASD models. As all quests for generality are fraught with difficulties, MASD has taken special care to temper these efforts via its core values (*cf.* Section 5.2.3).

And so ends our exploration of MASD as a methodology for the development of software systems. The next chapter's focus is on exploring the conceptual model imposed by MASD and all of its associated machinery.

29 As inferred by looking at research roadmaps such as France and Rumpe's (France and Rumpe, 2007), as well as Mussbacher *et al.*'s (Mussbacher et al., 2014).

6 DOMAIN ARCHITECTURE

One thing that has not changed and that has been proven repeatedly is that all real-world software benefits from physical design. *That is, the way in which our logical content is factored and partitioned within files and libraries will govern our ability to identify, develop, test, maintain and reuse the software we create. In fact, the architecture that results from thoughful physical design at every level of aggregation continues to demonstrate its effectiveness in industry every day.*

— John Lakos (Lakos, 2019)

Chapter overview

At the centre of MASD lies its domain architecture. The present chapter builds upon the backdrop sketched by Section 5.3 and provides a comprehensive picture of its motivation, core entities and associations.[1] The chapter first discusses the three individual domains that make up the domain architecture: the physical domain (Section 6.1), the logical domain (Section 6.2) and the variability domain (Section 6.3). Section 6.4 then concludes the chapter by bringing these three domains together to form the Logical-Physical Space (LPS).

Initial considerations

Before entering the analysis proper, a word is warranted with respect to the descriptions and figures employed within. These are not intended as formal models but are instead at a higher level of abstraction, and should be understood as exemplary cartoons, freely mixing architectural levels as necessary — *e.g.*, metamodels, models and object instances may be combined on the same plane, if doing so makes an explanation more accessible.[2] Secondly, a note on typography: a `constant width` font is used to highlight terms of MASD's ubiquitous language.[3]

With that in mind, let us enter the first and most important domain within MASD.

1 MASD's domain architecture and all of its related topics developed in this chapter fall under the remit of the *MSS* and the *MSS Development Process* (*cf.* Section 5.4.2.2).

2 In other words, we are not portraying implementation level details by design, because doing so invariably obscures the point at hand. Furthermore, implementations change as frequently as our understanding does, so, at best, one can only hope to describe a snapshot in time, soon to be superseded. This concern will be mitigated with the chapter's abstract descriptions, as their focus is to convey the process by which the implementation reached its present state rather than the state itself.

3 Whilst agreeing in general with the argument Evans puts forward (Evans, 2004), his use of CAPITAL LETTERS to highlight terms of the ubiquitous language was deemed detrimental to readability. Using a different font for the same end, in our opinion, is an acceptable compromise. That said, as MASD's vocabulary is large, we opted for highlighting only those keywords defined in the current sub-domain under analysis.

6.1 PHYSICAL DOMAIN

Section Overview

The *physical domain* is the subset of MASD's problem domain comprised of physical artefacts, as depicted by Figure 6.1, and it is predictably dominated by file artefacts and folder artefacts. Prior to analysing these entities in detail, one must first describe the processes that led to the present state of affairs, in terms of physical analysis and design (Section 6.1.1), physical modeling (Section 6.1.2) and, subsequently, by characterising its relationship with input variability (Section 6.1.3). The remaining subsections will then cover in detail each of the core physical elements and their associations (Sections 6.1.4 to 6.1.7).

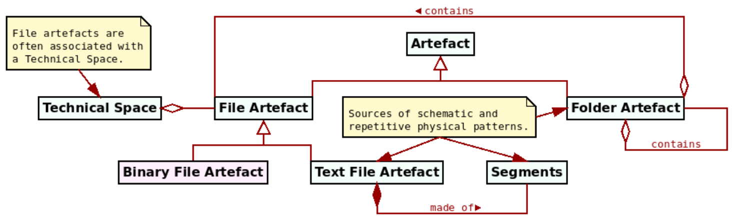


Figure 6.1: Key entities in the physical domain.

6.1.1 *Physical Analysis and Design*

Versus Domain Engineering

At a first glance, MASD's physical analysis and design appears to be a simple expression of Domain Engineering's Domain Analysis and Domain Design (Craveiro, 2021c) (Chapter 6), when applied to MASD's physical domain. However, since the shift in perspective caused by the "problem space / solution space inversion" adds significant complexity (*cf.* Section 5.5), it is important to understand the specificity of this application. Figure 6.2 helps in doing so by portraying both in simplified form: on the left, we have Domain Analysis and Domain Design — often used in an MDE context — and, on the right, MASD's physical analysis and design. The two approaches are separated by a bold black line.

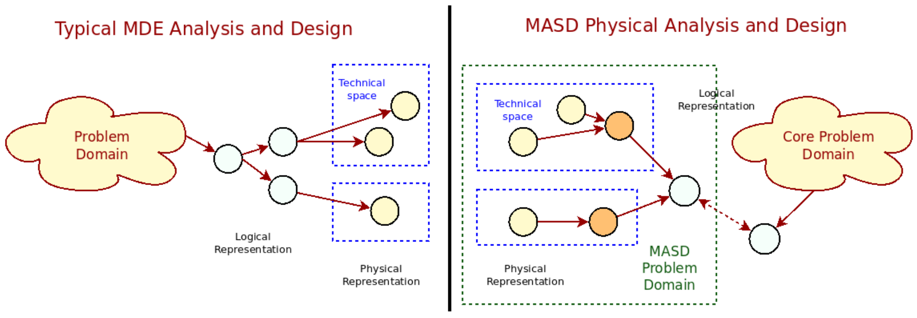


Figure 6.2: MDE analysis versus MASD physical analysis.

In the figure, yellow circles represent physical artefacts such as files and directories, clustered around dashed blue squares that denote individual TSs. For their part, light-blue circles represent logical elements such as classes and other high-level modeling constructs, likely in TS agnostic form. On the left side of the figure, one begins by observing a nebulous problem domain and designing a set of logical entities to form one or more models that fit requirements. Those logical entities will ultimately give rise to concrete physical entities through refinement, although these are often viewed as mere by-products of the process.[4]

MDE figure commentary

In contrast, the right-hand side of the picture reflects MASD's call for the reversal of this approach: physical entities themselves become the problem domain and one arrives at their logical representation, denoted by orange circles, through empirical analysis, with the core problem domain losing relevance in the process (far right).[5] Though perhaps not clear from the diagram, MASD's emphasis is on a comparative analysis of physical elements across TSs, not on the metamodels associated with each TS. It is so because MASD's analysis is driven by empirical evidence within the physical domain, rather than by the logical entities and meta-entities that inhabit each TS — even if the latter is more in keeping with MDE's ethos of metamodel to metamodel transforms (Craveiro, 2021c) (Chapter 3).[6]

MASD figure commentary

A consequence of this favouring of empirical evidence is that physical analysis and design became distinct from that of their traditional MDE counterparts, where discussions with stakeholders and UML diagrams from a business viewpoint abound.[7] In contrast, their application within MASD is closer, at least in spirit, to the modeling of scientific objects such as neurons, which were inspirational.[8] Since the object of our study are Schematic and Repetitive Physical Patterns (SRPPs), we opted for designing a specific approach for their handling which combines physical analysis and design into a single process, described next.[9]

Analysis and design approach

6.1.2 *Physical Modeling Process*

The first challenge in modeling SRPPs is in defining their nature. As already alluded to in Section 5.2.2, SRPPs are conceptually close to Stahl *et al.*'s schematic and repetitive code; however, *physical patterns* were preferred over *code* so as to bring clarity to our ubiquitous language. Whilst source code is MASD's primary target — it is, by definition, the central artefact type in a conventional software product — the methodology aims to model any physical entity man-

Versus schematic and repetitive code

4 Mellor (Mellor, 2004) and others in the Executable UML camp call for executable models, presumably bypassing the need for a physical representation altogether.

5 Note that we are referring to the development of MASD, rather than its application (*cf.* Section 5.4.2.3).

6 That is not to say TS metamodels are disregarded entirely. As we shall see later on (*cf.* Section 6.2), they are very useful as an input to the design process, but cannot be the main driver because much of its detail is too low-level to be suitable for our purposes.

7 MDA's CIM is a good example of modeling at this level of abstraction (*cf.* Section 2.3.2).

8 Here we find a typical example of cross-pollination in disparate fields. The first two years of our PhD were spent with the Computational Neuroscience lab, working with microscopic imagery and 3D mesh generation for neurons. The scientific processes used for this kind of modeling forms the basis of the process put forward by the present analysis.

9 Whilst SRPPs have already been mentioned (*cf.* Section 5.2.2), their discussion was purposely left to the present moment, so that it could be articulated in the broader context of physical modeling.

ifesting schematic and repetitive patterns, making the new nomenclature a better reflection of these broader aspirations.[10]

Axioms

Terminology aside, an open question remained on what was meant precisely by the term, so placing the concept on firmer ground was a priority. We did so by stating three axioms upon which all our analysis was to rest. First, ascertaining what is "schematic and repetitive" was declared to be a subjective matter, relative to many qualitative factors such as the experience of the observer, and thus demanding extensional definitions.[11,12]

Secondly, regardless of their subjective nature, we decided that:

- not all physical entities have patterns, nor are all patterns schematic and repetitive; these we deemed to lay largely beyond the scope of MASD.[13]
- a physical entity may be partially composed of schematic and repetitive patterns, and thus only partially under the remit of MASD (*cf.* Section 6.1.4.3).

Reproduction

Thirdly, and most significantly, we equated the discovery of physical entities with schematic and repetitive patterns to the modeling of physical entities at a level of detail sufficient for their reproduction. In other words, one determines if a physical pattern is schematic and repetitive by reproducing it by automated means; if it is possible to do so, then a pattern is deemed to be an SRPP.

What follows from these axioms is that an empirical process of discovery is needed in order to uncover patterns of interest. To arrive at such process, *ad-hoc* experimentation was carried out on a number of artefacts from an initial set of projects, until reproduction was achieved. Reflecting on the endeavour, we identified a number of well-defined steps:

Modeling process

1. **Sampling**: one must first determine the size of the physical sample, ensuring adequate coverage — *e.g.* across different TSs, possibly of different kinds, files of different types, and so forth.
2. **Decomposition (or Segmentation)**: each entity in the sample must be analysed in detail, and divided into well-defined constituent parts called *segments*.

10 The term *text* was also considered, for much the same reasons Model-to-Code (M2C) transforms became known as Model-to-Text (M2T) transforms within MDE. However, *schematic and repetitive text* was deemed insufficiently general because patterns can span other types of physical artefacts besides text files.

11 Extensional definitions are also referred to as *definition by enumeration*; that is, the definition of a concept by enumerating all of its parts. For details on definition by enumeration and other nuances related to definitions, the interested reader is directed towards Hebenstreit (Hebenstreit, 2007). For a high-level overview, Del Gaudio and Branco's introduction may suffice (Del Gaudio and Branco, 2009).

12 This assumption does not preclude using formal methods to define the nature of schematic and repetitive patterns, but merely states that this avenue was not exploited by the present work, relying instead on empirical forms of analysis. Similarly, another interesting but unexplored avenue is the use of ML techniques.

13 *Largely*, because one can still make use of MASD's infrastructure as a "helper", and then manually override the generated artefacts as needed.

3. **Labelling and Classification**: the identified parts must be named and categorised by means of taxonomic and morphological analysis. These will give rise to models of the physical entities and their constituent parts.

4. **Reconstruction**: the models are then used to recreate the original physical entity — for example via M2T transforms — at which point the pattern is declared to be a SRPP.

5. **Cataloguing and Parameterisation**: finally, the model for the entity is placed in the broader context of the existing catalogue of patterns. Doing so may entail merging the functionality with existing entities, adding additional parameterisation to existing entities, and so on.

Whilst well-defined in theory, note that the modeling process laid out here is a generalisation of what happens in practice, for these steps are seldom applied in such a clear-cut manner. There often is an overlap between steps, as well as a need for continued iteration in order to obtain the best results. In addition, though our work incorporated all of these steps, several challenges were faced initially due to the *ad-hoc* nature of the approach, as we tried to gain a better understanding of its mechanics. Sampling proved to be particularly problematic. Firstly, our original sample was composed of two trivial C++ and C# projects — in software engineering parlance, "hello world" projects — but these evolved over time and ultimately become the MRI's C++ and C# reference products (*cf.* Section 8.2).[14,15] Secondly, the source code of the MRI code generator was also incorporated into our sample, even as it mutated drastically over time (*cf.* Section 8.1).[16,17] Clearly, a more disciplined experimental approach would have been beneficial, with a rigorous process for artefact selection and better care taken in managing artefact evolution.

Sampling challenges

A second type of challenge was in distinguishing between physical elements, models and their instances, since we initially referred to all using the same terms (*i.e.* files and directories). In order to avoid confusion between real files and directories, as found in a filesystem, and their representation within MASD, we decided to qualify MASD physical entities with *artefact* (*e.g.* file artefact, folder artefact). This terminology was incorporated into MASD's ubiquitous language and is used consistently throughout the present manuscript. In the same vein, though not often mentioned to avoid confusion, MASD's physical elements are still logical representations of concrete physical elements.

Vocabulary challenges

As Figure 6.3 should make clear, we are not referring to the logical dimension within MASD (*c.f* Section 6.2), but instead to the modeling and subsequent refining of MASD's logical representation of physical entities into its final form, encompassed by the area of the dashed blue square in the picture. There, we take an instance of a physical model element (circles in orange) and create the corresponding file or folder in the filesystem (circles in yellow). Once the terminology was modified to reflect this, the ubiquitous language became unambiguous.

Logical representations

14 https://github.com/MASD-Project/csharp_ref_impl
15 https://github.com/MASD-Project/cpp_ref_impl
16 https://github.com/MASD-Project/dogen
17 All findings within this chapter are based on this moving sample of physical entities, which we cover in more detail on Chapter 8.

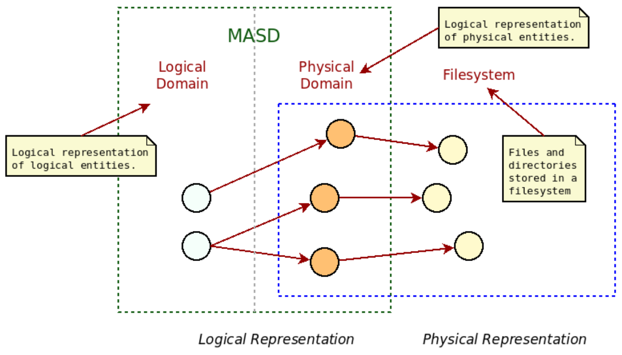

MASD
Logical Domain
Physical Domain
Logical representation of physical entities.
Filesystem
Logical representation of logical entities.
Files and directories stored in a filesystem
Logical Representation
Physical Representation

Figure 6.3: Transforms from logical representation to filesystem.

Regardless of challenges, the process has allowed us to identify and incorporate a number of physical patterns, many of which proved useful in the implementation of the MRI's code generator itself. What follows is a non-exhaustive list of SRPPs:

SRPPs in the MRI

- type definitions, including constructors and properties (getters and setters);
- GoF design patterns (Vlissides et al., 1995);
- serialisation support for different formats such as JSON and XML, possibly using different serialisation libraries;
- several mechanisms for test data generation;
- pretty-printing — *i.e.* dumping object state into a human-readable string representation;
- ORM mapping, providing RDBMS support;
- hashing — the generation of a hash function for the given state of an object;
- lexical casting, converting a C++ type into a string representation;
- unit test generation, validating the functionality of the generated features;
- generation of mocks for unit testing;
- and many more.

Patterns of patterns

As more physical patterns were identified and implemented, further empirical evidence was accumulated on their general characteristics. Since the continued growth of the pattern catalogue is a key concern of the methodology, and given that reconstruction is not always feasible, distinguishing what is reconstructible from what is not became a central question to MASD. In order to better understand the commonalities between the patterns within the catalogue, we decided to classify them with regards to input variability[18].

6.1.3 *Taxonomy of Functions of Input Variability*

Trivial functions

A property common to the captured SRPPs is that they all are *trivial functions* of input variability. Somewhat tautologically, a trivial function of input variability is defined to be any physical entity that can be fully or partially reproduced, given arbitrary (but valid) input describing structural or non-structural variability.[19] Reflecting on the modeling process (*cf.* Section 6.1.2), we realised that the steps of decomposition, labelling and classification are in effect an exercise in teasing apart functional dependencies on input variability from each physical entity, as explained by the flowchart in Figure 6.4.

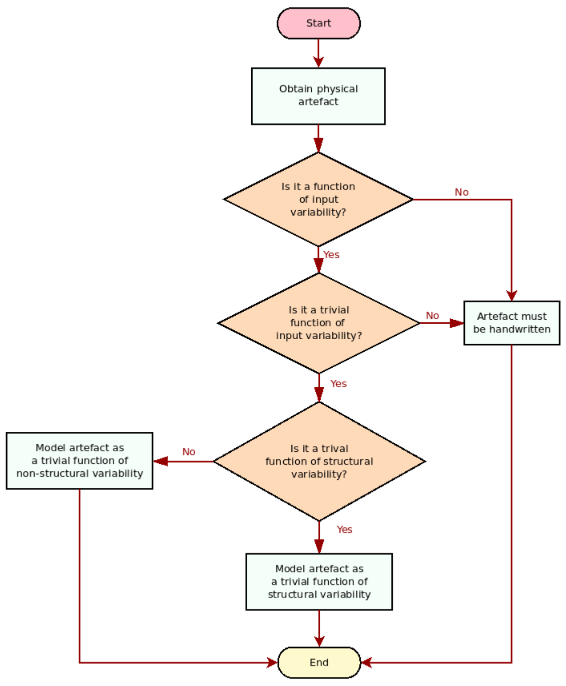

Start
Obtain physical artefact
Is it a function of input variability?
No
Yes
Is it a trivial function of input variability?
No
Artefact must be handwritten
Yes
Model artefact as a trivial function of non-structural variability
No
Is it a trival function of structural variability?
Yes
Model artefact as a trivial function of structural variability
End

Figure 6.4: Physical elements and variability.

18 A taxonomy of variability has been discussed in detail on Section 6.4 of (Craveiro, 2021c) (p. 56). It is largely based on the work of Groher and Völter (Groher and Voelter, 2007; Groher and Voelter, 2009). However, a short summary is sufficient for the purposes of this chapter. Variability can be grouped into two kinds: input variability, reflecting variation within the input models, and generational variability, concerning variability within the generated variants. Input variability is further split into structural and non-structural variability: "Structural variability is described using creative construction DSLs, whereas non-structural variability can be described using configuration languages." Generational variability is subdivided into positive and negative variability: "[in positive variability, the] assembly of the variant starts with a small core, and additional parts are added depending on the presence or absence of features in the configuration model. [...] [For negative variability, the] assembly process starts by first manually building the 'overall' model with all features selected."

19 There is an implicit assumption here; empirical evidence is required in order to decide if a given text can be generated or not. This can be achieved by creating M2T transforms that regenerate the desired text. In addition, in keeping with our pragmatic approach, we are not looking for formal proof that all structural permutations result in well-defined output, merely empirical evidence regarding a set of common cases.

From this perspective, one can then create a taxonomy of the identified SRPPs with regards to input variability, leading us to Figure 6.5. Physical elements are first classified as dependent or independent of input variability. Any element which is independent of input variability is inherently not reproducible — for example, free text, arbitrary directory structures and the like — and thus ignored (marked in red). Next, physical elements which are functions of input variability can either be complex functions — that is, functions which cannot be described in a mechanical manner, and thus must be ignored — or they are trivial functions of input variability. As there are two kinds of input variability, there are also two kinds of trivial functions: trivial functions of structural variability and trivial functions of non-structural variability.[20] Two sample trivial functions of non-structural variability are supplied: boilerplate and a long-form licence text such as the GNU Public Licence.[21,22] Finally, trivial functions of structural variability — which we often abbreviate to just trivial structural functions — are classified into two kinds, and several examples are provided for both.

Taxonomy description

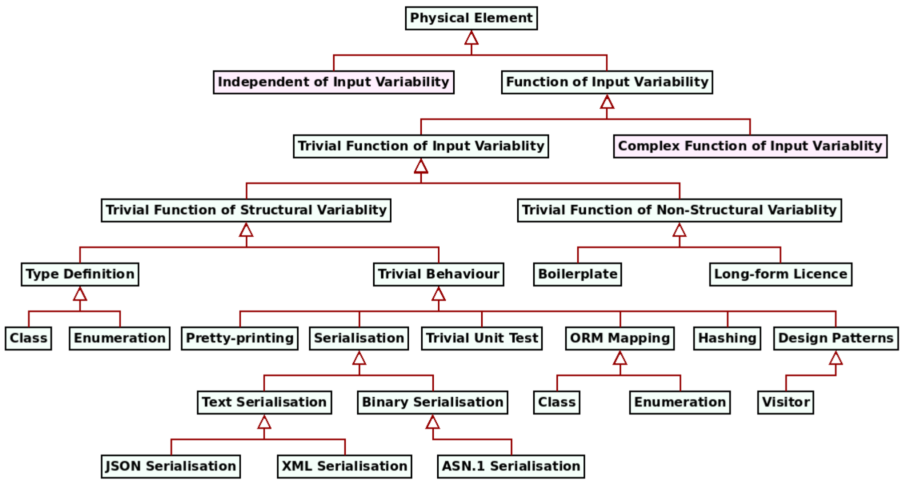


Figure 6.5: Taxonomy of physical elements with regards to variability.

Trivial structural functions have two main use cases. The first and most obvious is the definition of data types — *e.g.*, classes and their attributes.[23] The second use case is the implementation of simple behaviours for those data structures which depend on structural variability, such as serialisation and ORM support.

20 Of course, the matter is simplified here for didactic purposes, since physical elements tend to be functions of both types of input variability simultaneously. For example, a type definition is a function of structural variability, but we often then further parameterise it — *e.g.* generate a full constructor, generate getters only, *etc.* Nonetheless, one variability type can be said to be *dominant* over the other, for any given physical entity.

21 https://www.gnu.org/licenses/gpl-3.0.en.html

22 Wikipedia provides the following definition (*emphasis ours*): "*Boilerplate text*, or simply *boilerplate*, is any written text (copy) that can be reused in new contexts or applications without significant changes to the original." (Boilerplate text, 2021)

23 More specifically, we are building upon the DTO concept. Wikipedia tells us that (*emphasis ours*)

> The difference between data transfer objects and business objects or data access objects is that *a DTO does not have any behavior except for storage, retrieval, serialization and deserialization* of its own data (mutators, accessors, parsers and serializers). In other words, DTOs are simple objects that should not contain any business logic but may contain serialization and deserialization mechanisms for transferring data over the wire.

These we refer to as *trivial behaviours*, in contrast to complex behaviours which transcend "simple mathematization" — to borrow Hutchinson *et al.*'s wise words, out of context though they may be (Hutchinson, Rouncefield, and Whittle, 2011) — and thus demand manual handling. From this lens, special purpose code generators (Craveiro, 2021d) are seen to either generate type definitions and a single trivial behaviour — *e.g.* `protobuf` and the XSD tool generate the type definition and a serialisation format — or solely a trivial behaviour — *e.g.* ODB generates the ORM infrastructure, but relies on an existing type definition. With SRPP, what MASD proposes is the inventorisation of all such trivial behaviours and their unification under a single, integrated, framework.[24]

Trivial and complex behaviours

Once a taxonomy for input variability functions had been arrived at, we then set on devising a scheme for their composition. The literature provided ample material in this regard, which was found to be inspirational but ultimately unsuitable for our needs.[25] Using MASD's philosophy as a guide (*cf.* Section 5.2), we settled on a simple — and consequently inflexible — approach: a generated artefact may only be composed of zero or one trivial structural functions and zero or more trivial non-structural functions.[26] Admittedly, the limitation is severe, but the trade-off removes most of the complexity inherent to composition, and is therefore in keeping with the methodology's goals.

Function composition

The limitation is perhaps best understood by means of an example. In traditional OO programs, objects accumulate different *kinds* of behaviours, in an attempt to model entities found in the problem domain. Thus, it is common for a class to contain both domain-specific behaviours — *e.g.* a `Shape` may be able to `Draw`, `Rotate` and so forth — as well as infrastructural behaviours — *e.g.* the `Shape` may also be able to `SerialiseToJson`, `ToString` and the like. It is often the case that all of these behaviours are implemented as methods in one file. The restrictions on composition described above mean that MASD explicitly forbids such behavioural composition with regards to infrastructure; each of these behaviours — *e.g.* "JSON Serialisation", "Pretty-printing", *etc.* — is mapped to a separate trivial structural function and is associated with a single file artefact.[27]

Limits to composition

Having embraced the unsophisticated approach, it was then straightforward to align generational variability with the specific use cases identified within the physical space:

24 In fact, we propose to go even further and integrate special purpose code generators themselves within the framework as cartridges, (*cf.* Section 6.1.7).

25 For instance, there was an attempt to use Groher and Völter's Aspect-Oriented Model Driven PLE (AO-MD-PLE) (Groher and Voelter, 2007; Groher and Voelter, 2009) — a capable AOP-based approach which "integrates model-driven software development and product line engineering by providing means for expressing variability on (*sic.*) model level." Given that the composition of trivial functions is reminiscent of AOP's *concerns*, it seemed a good fit; however, in practice, it proved too complex: "In our opinion, its main downside is complexity, not only due to challenges inherent to AOP itself, but also because it uses several different tools to implement the described functionality and, understandably, requires changes at all levels of the stack." (Craveiro, 2021c) (p. 57).

26 Both are allowed to be zero because the empty or null file artefact is theoretically composed of zero trivial structural functions and zero trivial non-structural functions. In practice, it is implemented as a trivial non-structural function.

27 The problem domain specific behaviours are, of course, either not functions of structure or non-trivial (*i.e.* complex) functions of structure, and so must be handcrafted (*cf.* Section 6.1.4.3).

- **Positive variability** was deemed to be best suited to modeling the inter-artefact composition of trivial structural functions. By making these responsible for separate artefacts, we greatly simplified the process of stitching together the final implementation — which now becomes a mere expression of artefact relations (*cf.* Section 6.1.4). In addition, non-structural variability can be used to configure the presence or absence of each top-level function, in turn determining artefact relations. Thus, whilst still a difficult problem, it opens the door to the application of solving (*cf.* Section 6.3).

Application of generational variability

- **Negative variability** was deemed to be better suited to modeling intra-artefact instances of non-structural variability. For example, one can enable or disable class constructors as part of a type definition in a straightforward manner, adding little complexity to the overall process.

All of these moving parts can now be summarised into a cohesive narrative:

Approach Summary

- MASD locates physical entities with SRPP by empirical means. Content is deemed to be a SRPP if a trivial function can be created that reconstructs the target.
- MTs (M2Ts in particular) are be used to implement trivial functions of input variability. These make use of negative variability techniques to handle non-structural variability.
- each generated artefact must have a single responsibility, and that responsibility imbues the artefact with *a type* within MASD's physical representation.[28] The type is the trivial structural function.
- a broader framework is required to handle the inter-artefact composition of trivial functions of input variability, making use of positive variability techniques. As we shall see, MASD's solution is to embed the framework into the geometry of physical space itself (*cf.* Section 6.1.5).

Now that both the physical modeling process as well as the effect of variability on the object of study are understood, we can put these ideas in practice by analysing physical entities. Given their centrality, file artefacts are a most fitting point from whence to begin our exploration.

6.1.4 *File Artefacts*

`File artefacts` in MASD are a generalisation of regular files as defined by the POSIX specification ("IEEE Standard for Information Technology–Portable Op-

28 The approach echoes a well known principle within software engineering called the single-responsibility principle, which Wikipedia defines as follows:

> The single-responsibility principle (SRP) is a computer-programming principle that states that every module, class or function in a computer program should have responsibility over a single part of that program's functionality, and it should encapsulate that part. All of that module, class or function's services should be narrowly aligned with that responsibility. (Single-responsibility principle, 2021)

erating System Interface (POSIX(TM)) Base Specifications, Issue 7" 2018).[29] They can be organised hierarchically using `folder artefacts`, a generalisation of POSIX's directories which will be the subject of further scrutiny in Section 6.1.5. The next three sections demonstrate how physical modeling helped reveal the deep structure[30] within file artefacts by dissecting their taxonomy, morphology and relationships.

Versus POSIX

6.1.4.1 *Taxonomy*

MASD divides `file artefacts` into two types, according to their content encoding: `text file artefacts` and `binary file artefacts` (Figure 6.1). The latter are outside the methodology's present remit, and hence marked in red in this and subsequent figures. Encoding is a salient property of file artefacts, but it is only a starting point for their taxonomy; Figure 6.6 provides an example of a more detailed taxonomic view.

Encoding

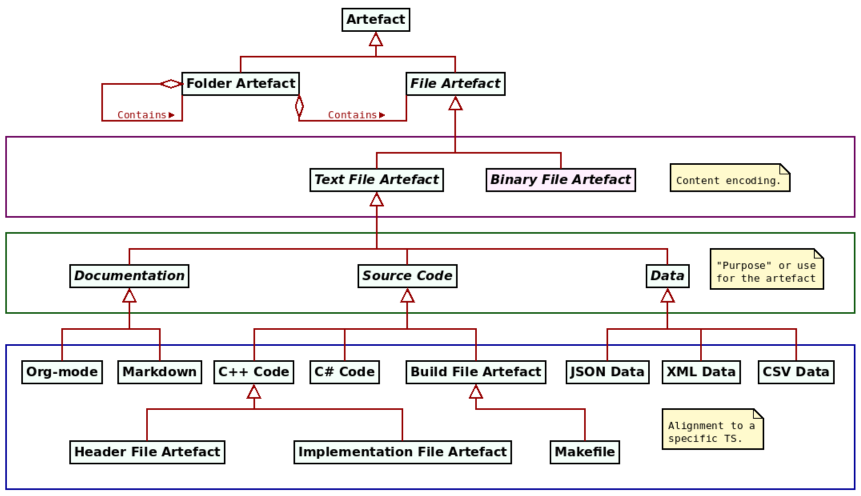


Figure 6.6: Example file taxonomy.

In this taxonomy, the green box contains three classes of `file artefacts` according to their purpose: `documentation`, `source code` and `data`. Conversely, the blue box consists of artefacts that are aligned with a specific TS. For example, `documentation` is sub-divided into `org-mode`[31] and `markdown`[32], two popular formats used in FOSS projects. `Source code` has two sample programming languages, `C++ code` and `C# code`. The C++ TS is of particular interest because it supports several distinct file types. The figure depicts `header file artefacts` — typically used to declare types and functions to be called elsewhere — and `implementation file artefacts`; other types do exist within this TS, such as `module definitions`, as per the latest revisions of the language

File artefact taxonomy

29 For context, the relevant POSIX concepts are as follows. A *file* is a stream of bytes. Files are classified as either *regular files* or *special files*. Regular files are stored in media such as a disk drive and support random access. A *directory* is a special file that lists a set of files and their associated attributes.
30 To make use of Kottemann and Konsynsk vivid terminology (Kottemann and Konsynski, 1984).
31 `https://orgmode.org/`
32 `https://daringfireball.net/projects/markdown/`

(ISO, n.d.).[33,34] `Build file artefacts` were added to the diagram to demonstrate that not all source code is connected with a programming language, with `makefiles` illustrating the kind of instances to be found in this category. Finally, three `data` file artefacts are shown, targetting JSON, XML and CSV representations.

Classification challenges

One may infer from the above description that artefact classification is largely mechanical, but the process proved more challenging in practice. Let us consider the case of the Visual Studio[35] IDE projects, where there are at least two file types conveying project information: `.csproj` for C# projects and `.vcxproj` for C++ projects. These files have been expressed in XML for a number of releases, but later versions use JSON instead. In either case, they could be plausibly classified as `data` or `source code`, so an amount of judgement is needed to guide the decision making. Within MASD they were classified as `build file artefacts`, because the internal representation was deemed less important than the role they play on the development process. Similarly, it's also debatable whether `build file artefacts` constitute `source code` or are a separate category altogether, such as IaC. These and other questions are grey areas in the classification process.

6.1.4.2 *Morphology*

Structural analysis

Beyond taxonomy, physical modeling also interrogated the composition of `file artefacts` by inspecting their internal structure. Using elements from the before-mentioned sample projects, a morphology was constructed by dividing each file into its constituent parts until atomic segments were reached.[36] The analysis greatly benefited from our prior research on special purpose code generators (Craveiro, 2021d), because tools such as ODB had already identified segments such as `prologue` and `epilogue`.

C++ example

Figure 6.7 exemplifies the segmentation process by decomposing a C++ enumeration header file into its fundamental parts, which shall now be described. The file in question has three top-level segments: the `prologue`, the `body` and the `epilogue`. The `prologue` and the `epilogue` make up the `boilerplate`, thusly named because it has little sensitivity to structural variability.[37] The `prologue` is composed of the following sections:

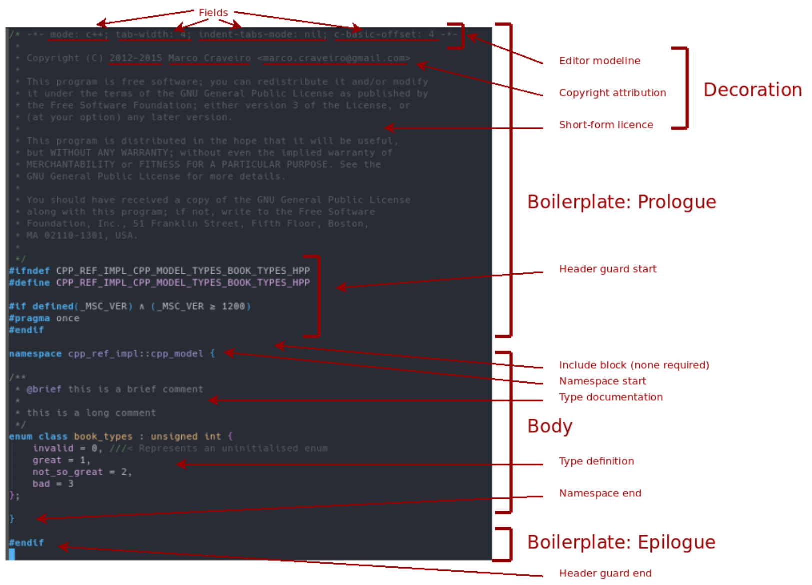
33 This observation may appear to be cursory but it is indeed significant: not all file types are supported for all versions of a given TS. The TS version has, of course, an impact on the available syntax of the language as well.

34 Note that file types have a complex relationship with file extensions. C++ is, as always, the most exotic of all TSs surveyed. In general, header files have a different extension from implementation files, but C++ custom allows for a diverse set of extensions. For example, header files can use `.h`, `.H`, `.hxx`, `.hpp`, *etc.* — with many other extensions having been observed in the wild (`.ixx`, `ipp` and the like). Similarly, implementation files use `.C`, `.cpp`, `.cxx` and so on. More generally, MASD models file extensions as non-structural variability associated with file artefacts.

35 https://visualstudio.microsoft.com/

36 *n.b.*, segments are atomic *from a reconstruction perspective*, not from a TS metamodel perspective This may perhaps be obvious given they are not a TS metamodel concept, but its worthwhile clarifying.

37 The wording was chosen carefully here for, as we shall see, the boilerplate may have a small amount of sensitivity towards structural variability. It is, of course, highly sensitive to non-structural variability.



```
/* -*- mode: c++; tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 4 -*-
 *
 * Copyright (C) 2012-2015 Marco Craveiro <marco.craveiro@gmail.com>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
 * MA 02110-1301, USA.
 *
 */
#ifndef CPP_REF_IMPL_CPP_MODEL_TYPES_BOOK_TYPES_HPP
#define CPP_REF_IMPL_CPP_MODEL_TYPES_BOOK_TYPES_HPP

#if defined(_MSC_VER) ∧ (_MSC_VER ≥ 1200)
#pragma once
#endif

namespace cpp_ref_impl::cpp_model {

/**
 * @brief this is a brief comment
 *
 * this is a long comment
 */
enum class book_types : unsigned int {
    invalid = 0, ///< Represents an uninitialised enum
    great = 1,
    not_so_great = 2,
    bad = 3
};

}

#endif
```

Figure 6.7: Morphology of a sample C++ file.

- the decoration, so named because it mainly contains informational elements. It is made up of the following:

  – the editor modeline, where editor-specific parameters are configured such as the spacing to use in Emacs or Vi, and variables of a similar ilk. The editor modeline is composed of a start marker, a set of key-value-pairs called fields, a separator between them and an end marker.

  – the copyright attribution, identifying the author or authors of the file. The copyright attribution is composed of zero or more copyright attribution entries, each made up of a date range, a copyright holder and a copyright email address.

  – a short-form licence, detailing the terms and conditions for the source code. A long-form licence is also available, but it is a standalone file whereas the short-form licence is a file sub-component.

Prologue

- the header guard, used by C++ to ensure a type is defined only on first inclusion, with subsequent mentions acting as *no-ops*. Header guards are the first scoped segment, with a start and an end; the start is part of the prologue whereas the end belongs to the epilogue. Also of significance is the fact that the header guard name is a function of structural variability — more specifically, of namespace containment.

Next we have the body, containing the core of the file and highly sensitive to structural variability. The body in the figure is made up of:

- a namespace, the second scoped segment within the file, with its own start and end.

Body

- the `type documentation`, expressed in Doxygen notation.[38]

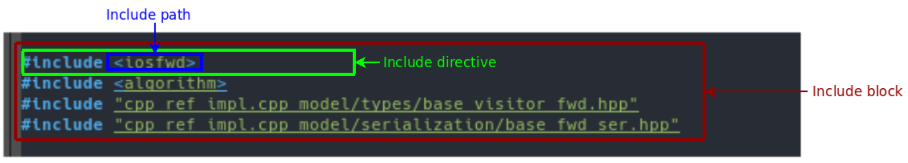
- the `type definition`. Note that each individual entry within the enumeration can have its own documentation, if supplied. In the example, only the `invalid` enumerator makes use of this feature. Significantly, note that the type definition is not a scoped segment in MASD because it is contiguous; that is, it does not contain other elements.

Epilogue

The file ends with the `epilogue`, in this particular case catering only for the closing of the `header guard`. Variability does allow for the `editor modeline` to be moved onto the `prologue` when requested, via configuration, but this feature is not used by the example.

Include block

Absent from Figure 6.7's `body` is the `include block`, as the enumeration does not depend on any other file.[39] Since `include blocks` are a significant element in MASD's support for the C++ TS, a sample was sourced from elsewhere to demonstrate the concept (Figure 6.8). The `include block`, bounded by a red box in the picture, is typically located right after the `header guard`'s start. It is composed of a set of `include directives`, often abbreviated to just `includes`, one of which is labelled in green. `Include directives` contain the `inclusion path` for all files the current artefact depends on, as exemplified in blue.



```
#include <iosfwd>
#include <algorithm>
#include "cpp_ref_impl.cpp_model/types/base_visitor_fwd.hpp"
#include "cpp_ref_impl.cpp_model/serialization/base_fwd_ser.hpp"
```

Figure 6.8: Example include block.

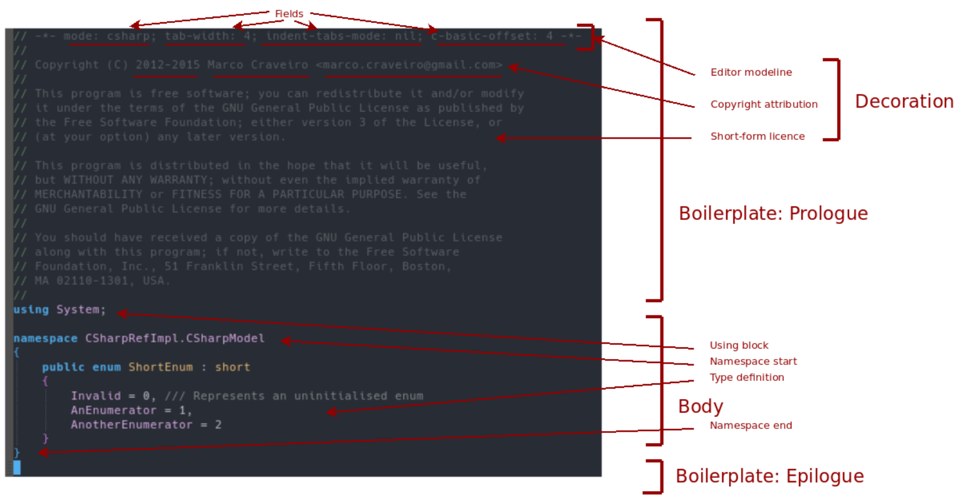
Departures from TS terminology

`Include blocks` were also chosen for this analysis because they demonstrate how and why MASD departs from TS concepts. In this particular case, the language of the ISO Standard was overridden because `include paths` are a clearer statement of intent when compared to the standard's wording.[40] Similarly, the C and C++ programming language specifications do not require the notion of a "block" — includes may be placed anywhere in a file, according to language syntax — whereas MASD finds having a cohesive entity to handle inclusion extremely helpful for its modeling and reconstruction needs. In other words, MASD cares about the observed patterns of use rather than the full universe of possibilities allowed by the TS metamodel.

Versus C# example

For completeness, Figure 6.9 carries out a similar morphological examination on an file from the C# TS, it too depicting an enumeration. There are some noteworthy points, so a brief comparison between figures is in order. Most of the elements are common to both Figure 6.7 and 6.9, with a few notable exceptions. The `boilerplate` of Figure 6.9 is composed entirely of the

38 `https://www.doxygen.nl/index.html`
39 File dependencies will be revisited towards the end of this section.
40 The C11 ISO Standard, in which the C++ ISO Standard depends, states: "Each library function is declared, with a type that includes a prototype, in a header, whose contents are made available by the `#include` preprocessing directive." (ISO, 2011) (p. 181) It then goes on to muddy the waters further, stating "[a] header is not necessarily a source file, nor are the < and > delimited sequences in header names necessarily valid source file names." The term "header" therefore does not seem particularly enlightening for the purposes of MASD's domain language. Many similar decisions were taken across the supported TSs.



```
// -*- mode: csharp; tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 4 -*-
//
// Copyright (C) 2012-2015 Marco Craveiro <marco.craveiro@gmail.com>
//
// This program is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation; either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
// MA 02110-1301, USA.
//
using System;

namespace CSharpRefImpl.CSharpModel
{
    public enum ShortEnum : short
    {
        Invalid = 0, /// Represents an uninitialised enum
        AnEnumerator = 1,
        AnotherEnumerator = 2
    }
}
```

Figure 6.9: Morphology of a sample C# file.

`decoration`, for no other elements are available to this TS, and the comment syntax used in the `decoration` is shown to be sensitive to the TS.[41] Within the `body`, the `using block` replaces the before-mentioned `include block` in the latter figure, though performing a similar role.

Overall, a surprising degree of symmetry emerges between these two examples, though they belong to two distinct TSs. To further the similarities, one could — and indeed, MASD does — generalise the `include block` and the `using block` into a `dependencies block`, as shown in Figure 6.10 below. This generalisation of relations was carried out as part of our third and final take on `file artefacts`, discussed in the next section.

Dependencies block

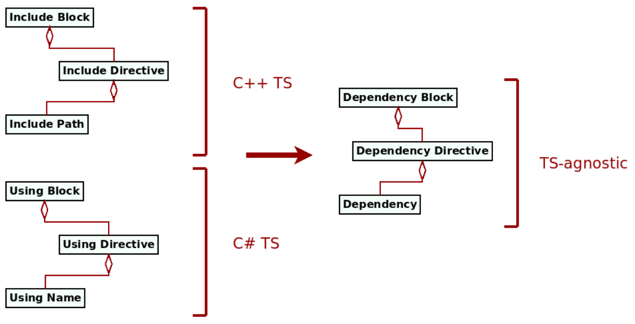


Figure 6.10: Dependencies generalisation.

6.1.4.3 *Relations*

Our last line of enquiry on `file artefacts` examined how they relate to each other. In MASD, a file a_1 is related to another file a_2 if the content of

41 In truth, both C# and C++ support the C style of comments, so one could conceivably have exactly the same syntax. We chose not to do so because this style is more idiomatic to C# code, as well as illustrating the point at hand. In MASD, idiomatic expressions are preferred unless they add significant complexity.

Definition

a_1 has a functional dependency on a_2, meaning that a_1 *references* a_2.[42] The reference can be an include path, the use of a type, or any other form of textual dependency. Mapping this example to MASD's physical domain, the `file artefact` A_1 — instantiated by the file a_1 — has a `relation` with `file artefact` A_2 — instantiated by the file a_2. In the `relation`, A_2 is known as the `referee` whereas file A_1 is the `referrer`, as per Figure 6.11.

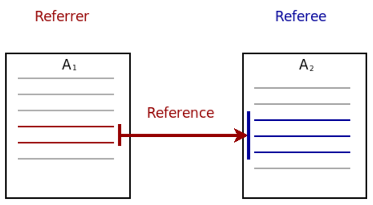


Figure 6.11: `Relation` between two files A_1 and A_2.

Having identified relations, the next task was to study their characteristics. Reusing our initial project sample, three factors were uncovered which affect relations: input variability, `origin` and `mode of production`. With regards to input variability, the following types of `file artefact` relations were observed:

Relations and input variability

- **Constant relations**: these cater for cases where a `file artefact` is always related to other well known `file artefacts`, insensitive to both structural and non-structural variability. For example, if all C++ files implement the `std::swap` algorithm, an include of the C++ Standard Library header file <`algorithm`> must always be present.[43] Similarly, a typical C# class will require a `using System` statement in order to implement `ToString`. Both examples presume there are no switches with which to toggle these features.

- **Variable relations**: these are dependencies on `file artefact` as a function of input variability. Given that there are two types of input variability, it is unsurprising that two types of variable relations were found:

  – **As a function of structural variability**: that is, the shape of the `body` of a file induces a dependency on another file. For example, if type t_1, defined in file a_1, has an attribute of type t_2, defined in file a_2, it will induce a dependency on the definition of type t_2, manifesting itself as a `relation` between `file artefacts` A_1 and A_2.

  – **As a function of non-structural variability**: meaning the configuration selected by the user creates relationships between files. For example, if a configuration enables an optional method, the method itself may necessitate the inclusion of additional types.

Origin

`File artefacts` can also be categorised in terms of their `origin`, as per Figure 6.12. When viewed from a MASD perspective, `file artefacts` can either

42 Referencing doesn't just apply to file content, but to identity, meaning its path, as well as to any other associated meta-data. This requirement is necessary in order to support the full gamut of variation associated with relations — *e.g.* C++ `include` statements, C# `using` statements *etc.*

43 The C++ `std::swap` function exchanges the values of two objects, `a` and `b`. It is a utility method often present in domain types.

be `exogenous` — that is, created externally — or `endogenous` — created and managed internally within MASD.

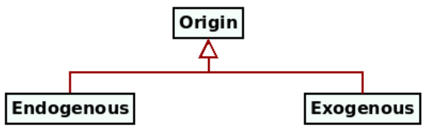


Figure 6.12: Taxonomy of file origins.

`File artefact` relations are impacted by its `origin`, giving rise to the following:

Relations and origin

- **Exogenous relations**: when an `File Artefact` is related to one or more `file artefacts` which are not generated by MASD. This encompasses, for example, all of the files in the C++ Standard Library. The `inclusion path` for external files is *irregular* — that is, it may follow any number of conventions regarding folder nesting and file naming, all of which are outside of MASD's control. If the `referee` is `exogenous`, it must first be exposed to MASD via a PDM, containing all required information about the file via non-structural variability, including a mapping to irregular paths.

- **Endogenous relations**: when a `file artefact` is related to one or more `file artefacts` generated by MASD, either within the same product or from a different product. The `inclusion path` of internal files is *regular*; that is to say, MASD is able to enforce a convention for the include path, making it largely — if not entirely — a function of structural variability (*cf.* Section 6.1.5).

Mode of production

Finally, a concept closely related to a `file artefact` origin is its `mode of production` — that is, how it was originated. Files have three distinct `modes of production`: `manual`, when produced by humans, `automated`, when produced by machines and `partially automated`, when produced by a combination of the two. Figure 6.13 depicts these three different modes in diagrammatic form. A file produced manually is commonly known as handwritten or handcrafted. Since the main method for the automated production of files is code generation, these are known as code-generated or simply just generated. Finally, files produced in part by automated means are known as partially generated / automated, and require the merging of handwritten and generated content to attain the file's final form.

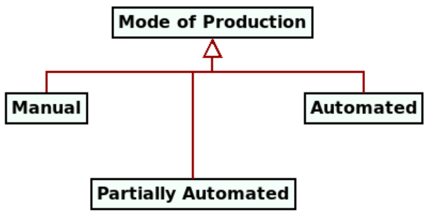


Figure 6.13: Taxonomy of modes of production.

A software project that contains both handwritten and generated files, partially or fully, will require one or more integration strategies (Greifenberg et al., 2015a; Greifenberg et al., 2015b). The `mode of production` is significant to the MASD domain architecture because it is responsible for setting out the menu of available integration strategies to its end users; these strategies impact file relations. Given that `exogenous Referrers` are outside of MASD's remit by definition, one needs to focus only on `endogenous referrers` and thus arrives at the following permutations:

Integration strategies

- **Fully generated referrer**. MASD is made aware of this relationship via input variability — structural or non-structural, depending on the case. This encompasses a number of sub-cases for the `referee` — *e.g.* endogenously generated, endogenously partially generated, endogenously handwritten, exogenous — but on all cases, its details must exposed into the system; this is done via regular MASD models for all endogenous cases and via PDMs for the exogenous case.

- **Partially generated referrer**. The generated portion of the file is handled as per the previous case. However, the handwritten portion of the file, created via protected regions, may bring in additional relations which MASD must generate. These are made known to MASD via input variability.

Relations and integration strategies

- **Handwritten referrer**. The user is responsible for creating the file, as well as managing its relations. However, the relations must also be made known to MASD via input variability, because they may impact other files such as build files.

Joining all of these dots, one is forced to conclude that all file relations must be exposed to MASD, regardless of `origin` or `mode of production`, if text file reconstruction is to be achieved; and input variability, either structural or non-structural, is how MASD is to be made aware of those relations. This isn't by any means a novel conclusion — it has been MDE's long held position that everything within a software product should be modeled — but it is nonetheless significant that bottom-up analysis (*i.e.* physical to logical) is in agreement with its top-down counterpart (*i.e.* logical to physical).

Pervasive modeling

This conclusion is also an apt end to the physical analysis of files. A similar process to what is described here was carried out for different types of text files, across multiple TSs and with bodies carrying different payloads; once we established a basic taxonomy and morphology that satisfied all our samples, our attention then turned towards characterising folders.

6.1.5 *Folder Artefacts*

Like `file artefacts`, `folder artefacts` also have an underlying structure, albeit simpler, and therefore it too can be unearthed via the physical modeling process (*cf.* Section 6.1.2). The analysis is presented in three parts. First, we discuss the folder taxonomy revealed by dissecting our sample, which includes terse descriptions of the identified elements (Section 6.1.5.1). Next,

Overview

the interaction with different forms of input variability is investigated. Finally, examples are provided to clarify all concepts discussed (Section 6.1.5.3).

6.1.5.1 *Taxonomy*

`Folder artefacts` possess an underlying taxonomy because folders serve different purposes within a software product, storing distinct types of `file artefacts` (*cf.* Section 6.1.4). Of course, products may be organised in any number of possible folder hierarchies, each a function of complex variables such as the prevailing coding standards, IDEs and other build tooling, the target TS and many more. For example, a "typical" Java folder structure (Maven Project, 2021) is noticeably distinct from that of C# (Microsoft, 2021), with both TSs having experienced a considerable amount of change since inception. C++ is further complex still, spanning the widest range of structural variation of all considered TSs.[44]

Structural variation

Nevertheless, as it was with `file artefacts`, so it is with `folder artefacts`: our objective is not to address all possible permutations found in the wild, but instead to model variation within a sample set and then grow outwards from this baseline. This incremental approach enabled us to extract key entities for these specific use cases by abstracting away TS-specific details, and adding parametrisation via non-structural variability as needed. When combined with the requirements for the composition of input variability functions (*cf.* Section 6.1.3), it allowed us to arrive at the taxonomy depicted by Figure 6.14, and which will form the basis for the discussion in the present section.

Physical modeling

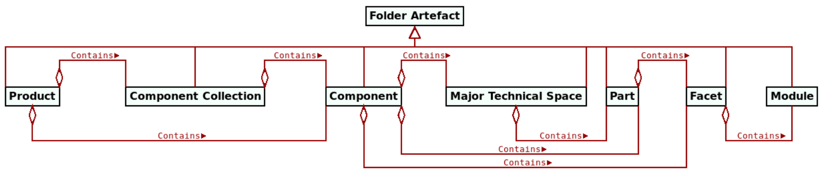


Figure 6.14: Simplified folder artefact taxonomy.

The first `folder artefact` of interest to MASD represents the software product itself, and it is expected to be the topmost of the hierarchy — typically, the root in a version controlled repository. A `product`, from MASD's physical viewpoint, is a named and versioned artefact hierarchy evolving over time. `Products` are ensembles of `components`[45], and these can have zero or more TS-specific targets — *e.g.*, executables, shared libraries, PDFs and the like. Typically, each `component` is associated with one major TS, but multiple major

Products, components

44 Unfortunately, we did not find material within the academic literature surveying folder layout. It is however an issue of great interest within the FOSS community, as demonstrated by a survey carried out by Pike (Colby Pike, 2018). There is also a wealth of community documents on project layout such as (Colby Pike, 2021) and (Boris Kolpackov, 2021) (Section "Canonical Project Structure"), which proved invaluable to our research and, to an extent, corroborate the above statements. An area of further research is to collate the community contributions into a disciplined academic survey.

45 Whilst Lakos' work is of great significance to our own, we nevertheless disagreed with some of his choices on terminology. For example, Lakos couples components with the C++ TS, defining these as an amalgamation of header and implementation files. This was not a useful definition from a MASD perspective, given our quest to find TS-agnostic entities, so it required redefinition. More generally, Lakos work is best seen as inspirational to our analysis rather than its literal substrate.

TSs are also supported, catering for more exotic topologies.[46] For brevity, `components` with a single major TS can elide the TS folder via non-structural variability. `Components` within the ensemble can also form cohesive sub-groups called `component collections`; where the `component collection` is made up of a single `component`, the `component collection` folder may be elided.

Parts

A `component` is divided into one or more `parts`; `components` with a single `part` may omit the `part` folder. `Parts` were originally introduced to cater for the filesystem layout of C++ projects, which often store public header files in a different directory from that of private headers and implementation files — *e.g.*, the idiomatic `include` and `src` directories. However, the concept has been subsequently generalised to cater for other artefact groupings such as MASD models, as well as `component` documentation, both of which reside on their own top-level folder within a `component`. Figure 6.15 illustrates the generalisation via a `part` taxonomy with two TSs, and depicts a number of sample `parts` for each.

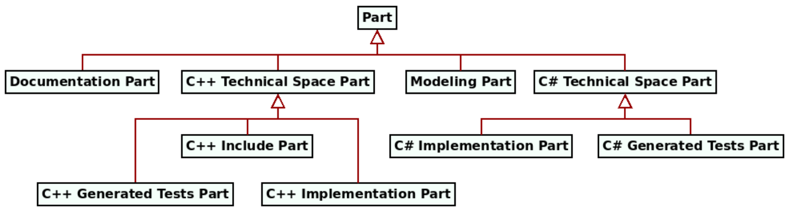


Figure 6.15: Simplified part taxonomy.

Facets

`Parts` may be further sub-divided into `facets`. A `facet` is a container for a set of related `file artefacts` , all belonging to the same TS, and was introduced as the mechanism to implement the composition of trivial structural functions via positive variability techniques (Figure 6.16). `Facets` align closely with the types of input variability functions identified in Figure 6.5, and can be thought of as the containers of the concrete artefacts that emerge from this process — *e.g.* the `C++ type definition facet` contains class type definitions, enumeration type definitions and so on.

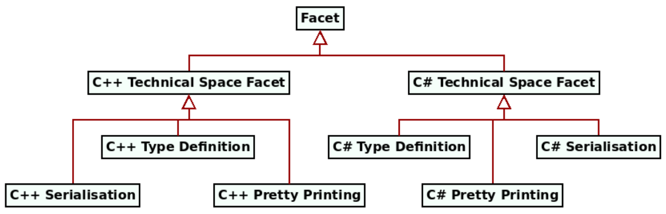


Figure 6.16: Sample facet taxonomy.

Modules

It is within the `facet` that MASD's physical domain finally meets the major TS's domain via `modules`. `Modules` are the physical representation of the programming language concept of `package` or `namespace`, and their expression is

46 This additional complexity is required in order to support `components` with implementations in multiple programming languages. It is of particular relevance in the context of tools such as SWIG (Craveiro, 2021d) (Section 5.3).

mostly controlled by structural variability, though non-structural variability also plays a vital role as the next section will explain.

6.1.5.2 *Input Variability*

`Folder artefacts` are also functions of input variability, though, as with the taxonomy, the observed variation is narrower than that of files. Input variability interacts with `folder artefacts` in the following manner:

- **Structural variability** determines the object graph of the physical model entities within MASD that are part of the `folder artefact` taxonomy. For example, the `product` and its associated properties, its `components`, available `facets` and so on.

Folders and input variability

- **Non-structural variability** controls, amongst other things, how the object graph is transformed into entities on a filesystem. For example, if a `component` targets a single TS, non-structural variability determines whether a TS `folder artefact` is expressed into a folder or not. Non-structural variability can also used to override default names of expressed folders — *e.g.* changing the name of the C++ TS folder from `cpp` to say `cxx`, *etc.*[47]

And with this we have now introduced the main concepts regarding `folder artefacts` in MASD. Let us now turn to examples of their application, to bring these concepts to life.

6.1.5.3 *Examples*

Approach

The terse definitions of the previous sections will now be made clearer by reviewing four snapshots taken from our sample projects. These were selected so as to portray the elements identified thus far from different viewpoints. A UML package diagram is used to represent `folder artefacts`, with composition indicating containment — thus simplifying diagram structure — and stereotypes signifying MASD physical types. The diagrams also introduce the use of colour, which is exploited throughout MASD to convey the various meta-elements in a distinctive manner.[48,49]

47 To be clear, the `folder artefact` representing the C++ TS is always called `cpp` within the physical domain, as this is an intrinsic property of the entity. However, the MT chain that transforms the `folder artefact` into a folder takes non-structural variability into account, allowing users to override the folder's name.

48 MASD's use of colour is inspired on Coad *et el.*'s work (Coad, Luca, and Lefebvre, 1999), though heavily customised for its purposes. Coad *et el.* put forward a compelling argument for the use of colour in UML diagrams, stating (*emphasis theirs*): "Color gives us a way to encode additional layers of information. The wise use of color increases the *amount of content* we can express." (Coad, Luca, and Lefebvre, 1999) (p. 6) Their analysis rests on the shoulders of Tufte (Tufte, Goeler, and Benson, 1990), but it goes further by performing a thorough analysis of the *types* of logical entities found in domain models. Part of MASD future work is to reconcile their approach with the physical and logical entities we have identified.

49 At present, MASD makes use of an *ad-hoc* palette for its metamodel elements. A review of the literature on the use of colour is needed, including material such as Waskom's interesting work (Waskom, 2021), so that solid foundations can be laid in this regard.

The first example is sourced from MASD's C++ reference model `cpp_ref_impl` (*cf.* Chapter 8). It contains a fragment of the `product` directory structure and is depicted by Figure 6.17. The image shows a selection of top-level `components` for the `product`, with `facets` and `modules` being deferred to subsequent examples.

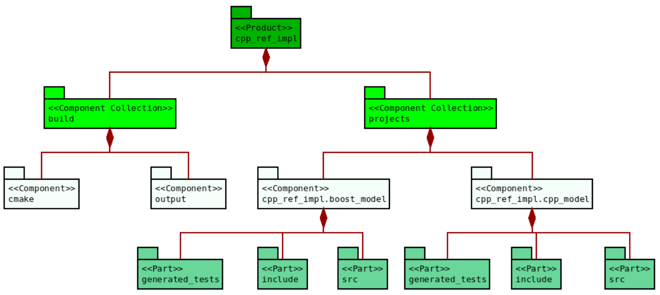


Figure 6.17: Sample top-level folders in `cpp_ref_impl`.

C++ reference implementation example

At the top we have the `product folder artefact`, `cpp_ref_impl`, with two visible `component collections`: `build` and `projects`. `Build` contains an assortment of files related to the build process such as CMake[50] modules (`cmake` component), as well as the `output component`, dealing with the artefacts created by the build system. The main source code for the product lives under the `projects component collection`, of which two `components` are shown: `cpp_ref_impl.boost_model` and `cpp_ref_impl.compressed`. Prefixing a `component` name with the `product` name is a configuration choice which can be disabled via non-structural variability. Both `components` have the same internal structure, with C++ as the single major TS, and the TS folder suppressed by non-structural variability; and both have three parts on show: `generated_tests`, housing all code-generated test code, `include`, with all header files for public inclusion, and `src`, with the implementation files.

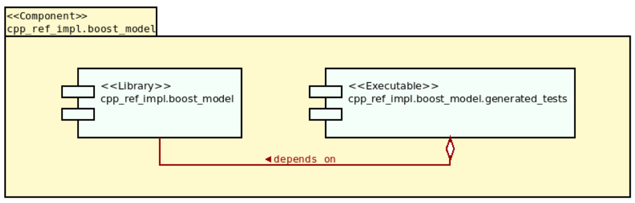


Figure 6.18: Targets in a sample `Component`.

Targets

The association between targets and `components` is not shown in the previous diagram but, for completeness, it is displayed in Figure 6.18. Both `components` have two targets: an executable target — responsible for making the `generated_tests` executable, which is associated with the `generated_tests part` and runs all code-generated tests — and a library target, associated with the `src part`. However, it is important to note that targets are not directly related to `folder artefacts` in the MASD domain architecture, and MASD is not interested in the targets themselves; instead, they are seen as by-products of build file state.

50 https://cmake.org/

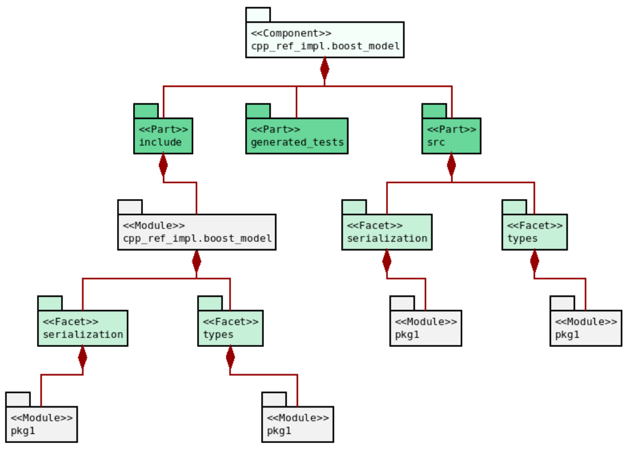


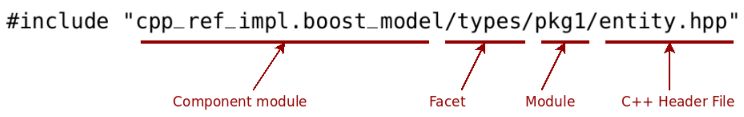
Figure 6.19: Folder artefacts in the Boost component.

Boost component

The next example zooms in on `component` `cpp_ref_impl.boost_model`, allowing us to see a selection of `facets` and `modules` (Figure 6.19). As with the previous figure, three `parts` are shown: `generated_tests`, `include` and `src`. The `include part` has a top-most `module` named after the `component` itself (`cpp_ref_impl.boost_model`). This `module` is known as *the component module* because it is used by C++ to create distinctive `include paths` for header files, as we shall see momentarily. The diagram displays two of the available `facets`: `types`, containing type definitions, and `serialization`, containing support for the Boost serialisation library. Each facet contains the `module` `pkg1`, a user-defined namespace in the C++ TS.

Regular includes

These elements can now be put together to form an example `include directive` for a file called `entity.hpp`, defined in the `types facet` and the `pkg1 module` (Figure 6.20). The `include path` within the directive is regular (*cf.* Section 6.1.4.3), being a function of both MASD's physical topology and structural variability.



```
#include "cpp_ref_impl.boost_model/types/pkg1/entity.hpp"
```

Figure 6.20: Example of a regular include path.

TS folder artefact

Our third example focuses on the *Language Agnostic Model* or LAM, which exercises PIM support. This `component` is designed to export the same generated code to two TSs, C++ and C#. As shown on Figure 6.19, each TS is stored on its own folder, named `cpp` and `cs` respectively. The internal structure of each TS is as per previous examples — with `parts`, `facets` and the like — and thus omitted from the picture.

C# reference implementation example

The fourth and final example is a snapshot from the C# reference product `CSharpRefImpl` (Figure 6.22). Whilst conceptually simpler than the examples covered thus far due to having a single `part` in each `component`, this TS is not without its challenges. Marked in red in the picture is the `component` with the generated tests for the main component `CSharpRefImpl.CSharpModel`. It is so

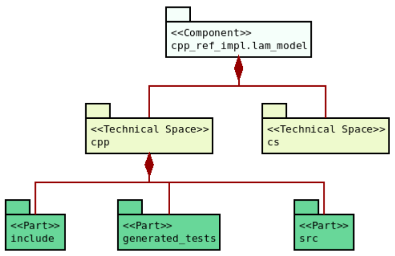
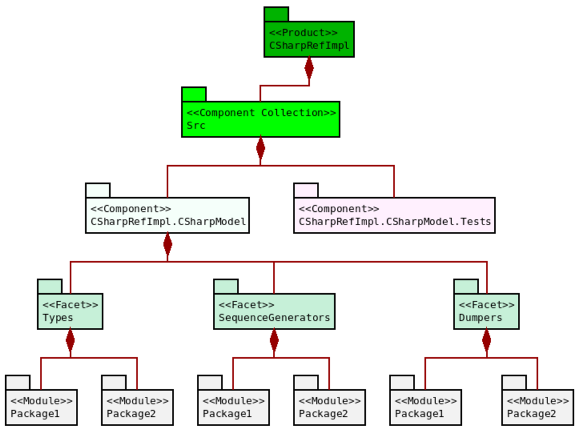


Figure 6.21: Folders in the LAM component.

because MASD does not yet support a topology where the generated tests can be placed outside the main `component` from whence they were generated; at present, the notion of hierarchical composition of physical entities is enforced strictly, so the same `component` cannot exist in more than one place.[51] Tests notwithstanding, the remainder of the diagram is straightforward, with three `facets` being shown: `types`, with type definitions, `SequenceGenerators`, for test data generation, and `dumpers` for pretty-printing. Each of these facets has two `modules` — namespaces in the C# TS: `Package1` and `Package2`.



<<Product>> CSharpRefImpl
<<Component Collection>> Src
<<Component>> CSharpRefImpl.CSharpModel
<<Component>> CSharpRefImpl.CSharpModel.Tests
<<Facet>> Types
<<Facet>> SequenceGenerators
<<Facet>> Dumpers
<<Module>> Package1
<<Module>> Package2
<<Module>> Package1
<<Module>> Package2
<<Module>> Package1
<<Module>> Package2

Figure 6.22: Folders in the C# Reference Product.

Emergent features

This example demonstrates the importance of covering multiple TSs in order to increase the range of variation within MASD's domain architecture. As further TSs are incorporated, the architecture will evolve to support those use cases, providing end users with additional flexibility. Moreover, non-obvious connections between TSs are also revealed, resulting in the unexpected emergence of features. For example, it is likely that the solution to support tests in a idiomatic manner in C# will enable additional use cases for all TSs.[52]

51 This issue is under active investigation at present, and will likely require minor changes to the domain architecture.

52 At present, all facets are combined into a single library per `component`. This has sufficed so far, but does have its inconveniences because you pay the full cost of a `component` regardless of the features in use. If one could separate say serialisation from type definitions, users of a `component` that do not require serialisation support would not have to link against it. It seems plausible both external test generation and facet splitting are related at a meta-level.

This example concludes our dissection of folders within MASD. Now that the main characters of the physical domain have been identified, they must be brought together into the methodology's metamodel architecture.

6.1.6 *The Physical Metamodel*

Overview

The entities described thus far in this chapter are part of MASD's PMM. The PMM defines the geometry of physical space in MASD. Physical space has an hierarchical nature, which is to be expected given its entities emanate from a hierarchical filesystem. Figure 6.23 exemplifies the relationship between the metametamodel (M3), the PMM (M2) and the PM (M1) by populating a metamodel hierarchy with a small number of sample entities. Files in the filesystem (M0) have been omitted for brevity.

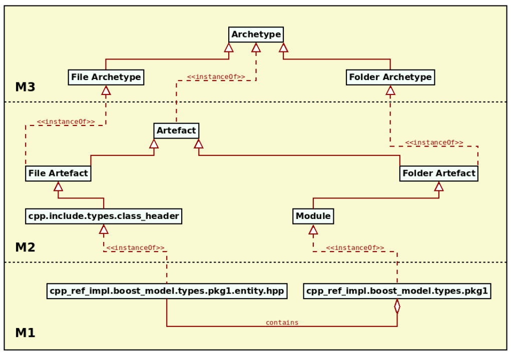


Figure 6.23: Example MASD metamodel hierarchy.

In the figure, MASD's metametamodel is comprised of `archetypes`, divided into `file archetypes` and `folder archetypes`. Within the metamodel, the `archetypes` are instantiated by artefacts such as `file artefacts` and `folder artefacts`. The figure then goes on to supply examples for both artefact types and, of these, the `file artefact` example is of special interest due to its name:

```
cpp.include.types.class_header
```

The notation denotes a fully-qualified `physical meta-name` and it is a `physical address` within MASD's `physical space`; it conveys a point in that space. To bring the notion home, let us look at a few more example archetypes:

Example archetypes

- `cpp.include.types.enum_header` is the archetype responsible for generating the definition for a C++ enumeration. It exists within the `cpp` TS, the `include` part and the `types` facet.
- `cpp.src.types.class_implementation` is the archetype that generates the implementation of a C++ class. It exists within the `cpp` TS, the `src` part and the `types` facet.

- `csharp.main.types.class` is the archetype responsible for generating the definition of a C# class. It exists within the `csharp` TS, the `main` part and the `types` facet.

More generally, `physical addresses` take the following form:

```
[technical space].[part].[facet].[archetype]
```

Physical regions

Besides just points, `physical addresses` can also be used to denote regions of `physical space`, which are sets, in the mathematical sense, of physical entities. For example, `cpp` contains all `parts`, `facets` and `artefacts` in the C++ TS; `cpp.include` is a subset of `cpp` and contains all `facets` and `artefacts` within that `part`. The geometry of `physical space` brings structure to the modeled physical patterns but, more significantly, the space is arranged in this fashion to facilitate the management of variability of physical entities — a topic of later analysis (*cf.* Section 6.3).

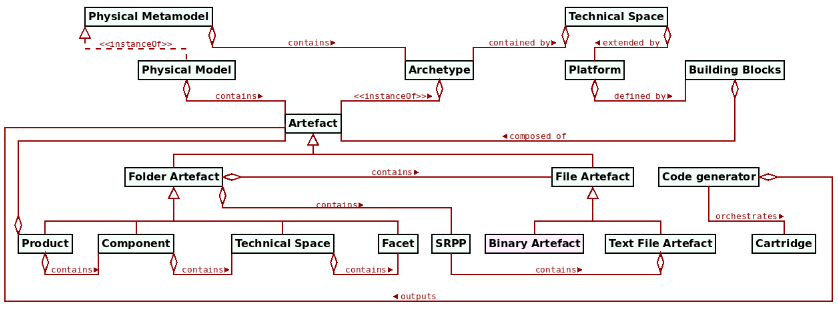


Figure 6.24: Fragment of the PMM.

PMM figure description

Figure 6.24 provides a birds-eye view of the PMM and related entities, bringing together all of the elements discussed thus far as well as introducing two new ones — `platforms` and `cartridges` — which, due to their complexity, will be dealt with by the next section (*cf.* Section 6.1.7). The image is largely a collage of previous diagrams, with a few noteworthy points. First, TS are shown twice to illustrate their dual nature in MASD: for `endogenous` purposes, they represent `folder artefacts`, but for `exogenous` purposes they are seen as sets of `platforms`. In addition, we see `archetypes` associated with the PMM though we previously alluded to a metametamodel; in practice, MASD opted for folding the metametamodel into the metamodel — that is, for employing loose metamodeling rather than strict metamodeling (Craveiro, 2021c) (Section 3.4). It was done so because the PMM was designed to cater for specific two use cases:

PMM use cases

- **To generate the code generator**; that is, elements such as TS, `parts`, `facets` and `artefacts` were modeled as SRPPs themselves. This was done by applying the physical modeling process to the development of the code generator, and extracting physical patterns, which were modeled and catalogued just as any other physical pattern.
- **To serve as the target of refinement**; that is, logical entities are transformed into physical entities via transform chains, and these physical entities are subsequently transformed into files and directories in the filesystem.

Loose metamodeling was sufficient to satisfy both of these use cases, so it was preferred to strict metamodeling. This "use case focused" approach is also in keeping with the vision for the methodology (*cf.* Section 5.2.1) — targetting narrow application and thus affording simpler solutions when compared to more ambitious applications of MDE. Loose metamodeling is not without its costs, however, and terms such as `archetypes` and `artefacts` are where its limitations start to show. These two terms are used through MASD, and at times it may appear that they are interchangeable; however, as the metamodel hierarchy already alluded to, they serve different roles in MASD's domain architecture. The choice of term is meant to denote the level of abstraction at which one is operating:

Narrow role, limitations

- the term `archetype` is used in the following contexts: 1) when we are referring to the generating function that creates instances of individual artefacts; 2) when we want to describe sets of artefacts, such as regions of physical space.
- the term `artefact` is employed when we want to classify a set of files, or when we have an artefact instance in memory for example.

Artefact versus archetype

And with this clarification, we are close to completing our survey of the physical domain. Before we do so, there are two additional physical entities we need to cover, and these are of a different nature of those identified thus far.

6.1.7 *Platforms and Cartridges*

The artefact-centric view of the world posited by the physical domain is instrumental in addressing some of MDE's ambiguities in its vocabulary, previously described in Chapter 2 and (Craveiro, 2021c) (Chapter 4). This section explains how it is used to characterise two important concepts, pertaining to two key entities within MASD's physical domain: `platforms` and `cartridges`. We start with the former.

Overview

A `platform` is understood to be an aggregation of building blocks within a TS (*cf.* Figure 6.24) — that is, a named and possibly versioned set of `artefacts` which, for all intents and purposes, is indistinguishable from a software product. The only difference between the two is that MASD deems `products` to be `artefact collections` it generates — *i.e.*, `endogenous` — whereas `platforms` are external to it and can only be accessed by means of an associated PDM (*cf.* Section 6.2) — *i.e*, `exogenous`. In other words, `platforms` lack the regularity afforded to MASD software products — thus requiring mapping — and are responsible for raising the abstraction level — thus simplifying the generating functions. Figure 6.25 illustrates MASD's `artefact`-centric and hierarchical view of platforms.

Platform definition

Whilst broad, this is nonetheless a definition that contains no ambiguity; a TS defines the syntax via its metamodel, and `platforms` are sets of `artefacts` that are valid instantiations of the syntax, and which have not been created by MASD. With this we avoid questions such as "is the CLR a platform?" or "is the JVM a platform?" (Craveiro, 2021c) (Chapter 5), as these are not meaningful

Addressing ambiguity

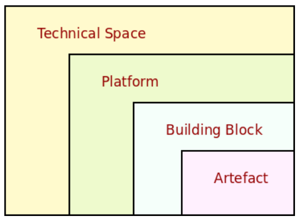


Figure 6.25: Technical Space composition

in a MASD context. If there is a library exposing the internals of the CLR or the JVM, then these libraries are considered platforms. More broadly, MASD is only interested in statements involving sets of physical `artefacts`.

Cartridge definition

`Cartridges` are of a similar nature to `platforms` with regards to their endogeneity.[53] They are defined as any code generator external to MASD whose input can be modeled as a `text artefact` generated by MASD, and whose output is a set of `text artefacts` on which MASD may perform further processing. The `cartridge` entity in the PMM (*cf.* Figure 6.24) models aspects of the external code generator such as its version and any other properties that have a functional dependency on the input `artefacts` generated by MASD.[54] In this way, the role of the `cartridge` entity is similar to that of the PDM: it regularizes external entities for consumption by MASD. From this perspective, MASD works as an orchestration framework for `cartridges`, requiring only minimal knowledge about the `cartridges` themselves; significantly, this is merely a by-product of the fact that `cartridge` inputs are sources of SRPPs.

MASD's interplay with `cartridges` is perhaps best understood by example. Let us consider MASD's integration of the tools ODB[55] and `clang-format`[56], two tools popular with C++ developers. MASD consumes these tools via a workflow with the following steps:

Example workflow

- **Step 1**: MASD generates the input files for ODB whenever users request RDBMS support for a given C++ project.
- **Step 2**: MASD supplies the input files to ODB, which generates a set of C++ files implementing the database layer.
- **Step 3**: MASD supplies the output of ODB to `clang-format`, which indents the source code according to a user-defined convention.

Cartridge pipeline

This workflow is implemented in the MRI as a `cartridge` pipeline, as depicted graphically in Figure 6.26. Regular M2T transforms are used to implement Step 1, whereas Steps 2 and 3 are exposed to the MASD framework as Text-to-Text transforms — *i.e.*, receiving `artefacts` as inputs and producing `artefacts`. Note that the T2T transforms are composed into a transformation pipeline

53 Cartridges were originally introduced in the context of MDA, on Section 2.3.2; there, the challenges with the term were also discussed. The notion, however, is generally applicable to any MDE-based methodology.

54 Examples of candidates for cartridges are the XSD tool and Protocol Buffers as well as other tools described in (Craveiro, 2021d). These become cartridges once they have been modeled and integrated with MASD.

55 `https://www.codesynthesis.com/products/odb/`

56 `https://clang.llvm.org/docs/ClangFormat.html`

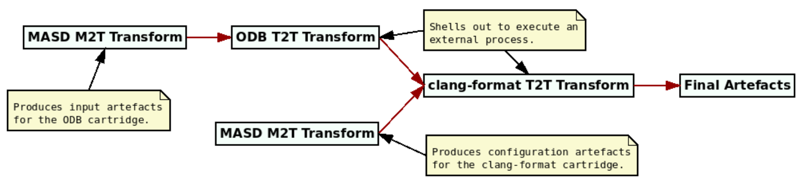

MASD M2T Transform
ODB T2T Transform
Shells out to execute an external process.
clang-format T2T Transform
Final Artefacts
Produces input artefacts for the ODB cartridge.
MASD M2T Transform
Produces configuration artefacts for the clang-format cartridge.

Figure 6.26: Example cartridge pipeline.

to produce the desired output, as they are parametrised by non-structural variability — *i.e.*, users can decide if they would like RDBMS support and/or source code formatting. In addition, the T2T transform encapsulating `clang-format` has an input `artefact` with configuration specific to the tool; it is generated by a M2T transform within MASD.[57]

`Platforms` and `cartridges` enable MASD to access the outside world. However, their modeling is seen as an instance of a more general process: ultimately, MASD extracts a general set of modeling entities from within the physical domain. These entities live in MASD's logical domain, and it is to it we shall turn to next.

6.2 LOGICAL DOMAIN

Section Overview

The *logical domain* is the portion of MASD's problem domain that deals with logical elements extracted from supported TSs, alongside with their relationships and variability requirements.[58] Unlike its physical counterpart, the logical domain is large in terms of footprint as it models an ever growing number of entities, and manifests a trend that is expected to continue over time. Nevertheless, from the perspective of the domain architecture, its role can be condensed to just two key themes: an overall characterisation of its composition (Section 6.2.1) and an analysis of the projections in and out of logical space (Section 6.2.2). The next two sections cover these two themes respectively.

6.2.1 *Composition*

LMM, LM

MASD's Logical Model (LM) is an instance of the Logical Metamodel (LMM), responsible for housing meta-elements modeling entities of interest within the logical domain. The LMM also caters for the various types of logical models such as PIMs, PSMs and PDMs — which, as already mentioned, are the gateway for exposing platforms to MASD. The LMM was created with Piefel and Neumann's ideas in mind (Piefel and Neumann, 2006), in that it is

57 Cartridges are a good example of the power of integrating generation within a single framework. MASD initially added targets in build files to invoke ODB, with the tool invocation performed by the end user during the build process. By moving the ODB invocation into a T2T transform, we have opened the possibility of performing further processing on the generated files (via `clang-format` in this case).

58 In order to improve readability, `constant width` font is used only for source code entities and vocabulary pertaining to the logical domain. Physical space terms are depicted in regular font.

an intermediate model designed specifically for code generation and serves no other purpose.[59] The majority of the entities housed in the LMM originate from generalisations of elements uncovered via physical analysis and design (*cf.* Section 6.1.1), via the process depicted by Figure 6.27.

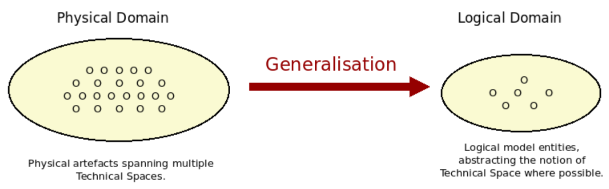

Physical Domain
Generalisation
Logical Domain
Physical artefacts spanning multiple Technical Spaces.
Logical model entities, abstracting the notion of Technical Space where possible.

Figure 6.27: Logical generalisation of physical concepts.

Entity sources

Nonetheless, it is important to note that the TS metamodel is also of great relevance to the shape of the logical entities; there is a natural relationship between constructs in the TS's metamodel and the patterns we are trying to capture in the LMM, as demonstrated by Figure 6.28. The bottom part of the diagram, in yellow, points out useful sources for logical entities given common elements in TSs; it is a *logical view* of the argument already put forward from a physical perspective via trivial structural functions (*cf.* Section 6.1.3).[60]

LMM versus TS metamodel

Abstraction levels

The free-style depiction used in the figure also tries to elicit the dangers of reading too much into the relationship between the TS's metamodel and MASD's logical representation. A construct is only relevant to MASD's logical model if it captures all the information required to create a projection of said construct into the physical domain (*cf.* Section 6.2.2). The TS metamodel offers a good source of inspiration for the identification of these logical elements; however, the objective of the LMM is not to replicate a TS's metamodel but instead to abstract commonalities between them, from the perspective of the projections. In this vein, as depicted in Figure 6.28, it is often necessary to break down TS metamodel entities, such as classes, into constituent parts — assignment, cloning, construction and so on — or finding higher level constructs which are not directly relevant to the TS metamodel — such as design patterns. Therefore, by observing patterns of use within text artefacts, we model at varying levels of abstraction when compared to the TS metamodel, as well as modeling languages such as UML — both of which designed for

59 In the before-cited paper, Piefel and Neumann explain that (*emphasis ours*)

> [...] the ultimate purpose of modelling is often code generation. While code can be generated from any model, we propose to *use an intermediate model that is tailored to code generation* instead. In order to be able to easily support different target languages, this model should be general enough; in order to support the whole process, the model has to contain behavioural as well as structural aspects. (Piefel and Neumann, 2006)

60 The analysis also raises the question of where to place the boundary between logical and physical domains. For Lakos, there is a stark distinction between the two: files and directories are part of the physical domain, classes and other TS constructs belong to the logical domain. In MASD the split is not quite as clear because we view file morphology and taxonomy as part of the physical domain; therefore some of what Lakos places in the logical domain is understood to be in the physical domain from a MASD perspective. In addition, the role of the logical domain in MASD is *to model the physical domain*, which is a very different take on the matter when compared to Lakos, where these two domains are largely unconnected. The lack on rigour in determining these boundaries has not yet brought about challenges, so the subject remains a candidate for future work.

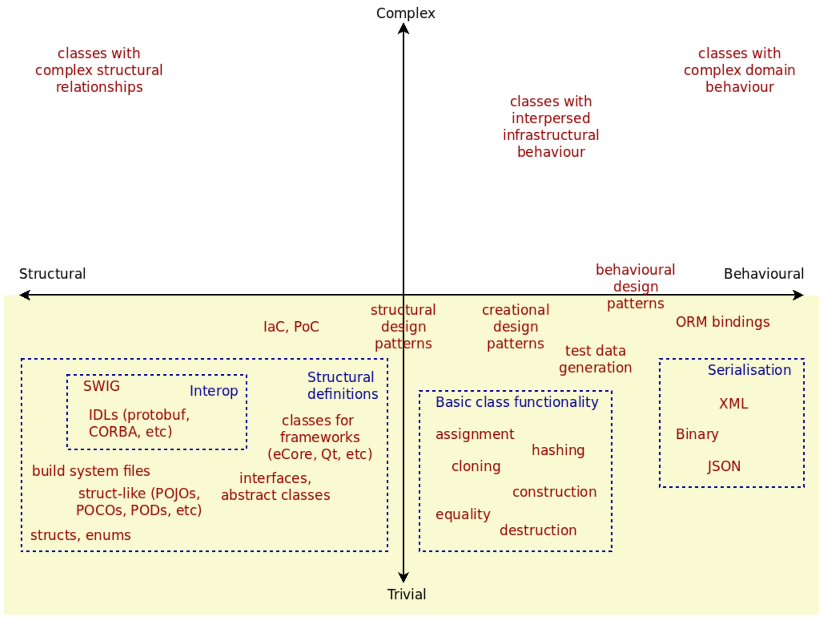

Complex
classes with complex structural relationships
classes with complex domain behaviour
classes with interpersed infrastructural behaviour
Structural
behavioural design patterns
Behavioural
structural design patterns
creational design patterns
IaC, PoC
ORM bindings
test data generation
SWIG
Interop
Structural definitions
IDLs (protobuf, CORBA, etc)
classes for frameworks (eCore, Qt, etc)
build system files
interfaces, abstract classes
struct-like (POJOs, POCOs, PODs, etc)
structs, enums
Basic class functionality
assignment
hashing
cloning
construction
equality
destruction
Serialisation
XML
Binary
JSON
Trivial

Figure 6.28: Characterisation of TS entities.

different use cases. This is perhaps best understood by looking at modeled entities and their groupings. Figure 6.29 provides a high-level overview.

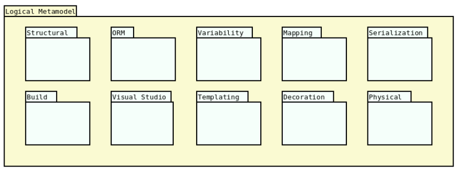


Figure 6.29: Packages within the LMM.

At present, the LMM is made up of ten distinct packages, each targetting a distinct area of a software product.[61] Their intent can be summarised as follows:

- **Structural**: Models structural variability within programming languages. It is the portion of the LMM that is the closest to programming language constructs.
- **Build**: contains entities responsible for build files such as Makefiles and CMake files, often used on UNIX-like operative systems.
- **ORM**: provides support for RDBMS, including tools such as ODB.
- **Visual Studio**: models the infrastructure needed to support the Visual Studio IDE, used on Windows platforms.

61 Please note that these packages form a snapshot of current state, since the LMM is expected to change as our understanding of the domain improves; new packages may be added, existing packages may be merged when deemed to have overlapping functionality, and so on. The LMM has changed considerably since its inception.

LMM packages

- **Variability**: contains all entities required to generate non-structural variability support in the MRI (*cf.* Section 6.3).[62]

- **Templating**: models entities related to logic-less templates, used in MASD to create M2T transforms.

- **Mapping**: support for PIMs is attained via these entities, which provide the infrastructure to map TS-agnostic types to their TS-specific counterparts.

- **Decoration**: contains all entities modeling file artefact decoration, as uncovered by morphological analysis (*cf.* Section 6.1.4).

- **Serialization**: provides support for entities required in a serialisation context.

- **Physical**: Models MASD's physical entities such as parts, archetypes, relations and the like; contains the LMM's representation of the PMM, and it is used to generate it.

Structural package

As there are over one hundred individual classes in the LMM, it is not feasible — nor necessary — to cover each of them in detail. It is however worthwhile sampling one of the packages in order to get a flavour for how these elements are modeled. Figure 6.30 does just so, providing a glimpse of the main components in the LMM's `structural` package. Most of the depicted elements are TS-agnostic representations of metatypes commonly found in programming languages, but there are a few noteworthy exceptions which warrant a fuller description — as do the varying levels of abstraction, denoted in the figure by the colour scheme.[63]

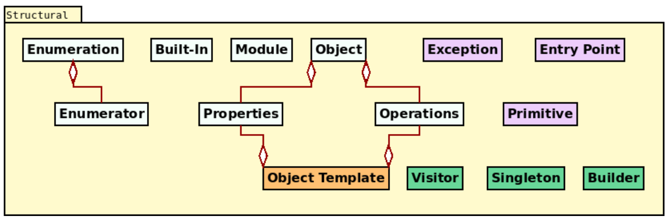


Figure 6.30: Classes in the `structural` namespace.

"Traditional" metatypes

`Module`, `enumeration` and `built-in` model, respectively, namespaces (or packages), enumerations and built-in types. `Object` is intended to stand for DTOs, though at present is synonymous with the TS concept of `class`, containing both properties (attributes) and operations.[64] Grouped together, these four

62 Within the LMM, we took the somewhat unfortunate decision of abbreviating non-structural variability to just *variability*. This is a decision that may be reviewed in the future to avoid confusion.

63 Unlike in previous diagrams (*e.g.* Figure 6.22), this figure employs colouring merely to facilitate the present explanation. Typical MASD models do not follow this colour scheme.

64 The names of these types are somewhat atypical, by design. We sought to avoid terms which were either keywords in MRI's implementation language (C++) or which were already well-known from other popular TSs — the idea being that MASD's interpretation may not necessarily map to their understanding on a given TS. It is for this reason that names such as `class`, `namespace`, `package` and so on were eschewed. The prefix *meta* was also explicitly avoided because it was felt to be a noise word (Kevlin Henney, 2021), given that all the types within the LMM are metatypes.

elements and their dependent types (in light blue) can be thought of generalisations of the "traditional metatypes" found in a programming language TS, as well as modeling languages such as UML.

Moving upwards in the abstraction ladder, we then have a second group of metatypes, in purple, which exist at a higher level of abstraction from that of the TS metamodel, and which we named *idiomatic metatypes*. These are as follows:

"Idiomatic" metatypes

- **Exception** represents types that denote error conditions. During the projection into physical space, MASD takes care of all of the machinery needed to make them idiomatic in the target TS, such as inheriting from `std::exception` in C++ or `System.Exception` in C#. They can also be mapped to error codes where the TS has support for these — *e.g.* the C language.

- **Primitive** is a wrapper around a built-in type that allows the creation of strong primitive types. For example, instead of using `std::string` to denote a unique identifier for a person, MASD allows the creation of a specialised primitive type for this purpose — *e.g.*, a `PersonId`, with an underlying type of `string`.[65]

- **Entry point** represents the function where program execution begins — *e.g.*, `main` in the C/C++ TS and, typically, its C# counterpart of `Main`.

All of these entities should be familiar to software developers, with more of their ilk to be added in the near future such as named key-value pairs, units (*e.g.* support for units of measure like the metric system), named bitsets and the like. Types in this grouping are mainly related to idioms, uses and conventions that often span multiple TSs, but are restricted to a single type.

Design patterns

Moving up the abstraction ladder once more takes us to the GoF's design patterns (Vlissides et al., 1995); these are shown in green on the diagram. Design patterns distinguish themselves from idiomatic use cases by being larger aggregates, usually involving the collaboration of multiple classes. The MRI only supports the `visitor` pattern (p. 331) at present — applicable whenever inheritance is employed — but other patterns such as `singleton` (p. 127) as well as `builder` (p. 87) are currently under development. Though they are being accrued incrementally, it is expected that the majority of the GoF patterns will eventually be represented within the LMM.[66]

Nonetheless, these policies may be reviewed in the near future, particularly where greater clarity can be attained.

65 Some modern languages such as Nim (Nim Project, 2022b) have built-in support for strong types. Nim calls these *distinct types*, defined as follows: "A distinct type is a new type derived from a base type that is incompatible with its base type. In particular, it is an essential property of a distinct type that it does not imply a subtype relation between it and its base type. Explicit type conversions from a distinct type to its base type and vice versa are allowed." (Nim Project, 2022a) Primitives highlight another advantage of the MASD approach, which is to cross-pollinate ideas across TSs.

66 Whilst GoF designed patterns are referenced often, please note that MASD does not limit itself to this set of patterns. It just so happens that there is abundant literature as well as implementations — and therefore ready-made use cases for MASD to consume. However, the expectation is that other kinds of patterns will follow. As an example, work has begun on integrating dependency injection — a technique of inversion of control that echoes the `strategy` pattern (p. 315).

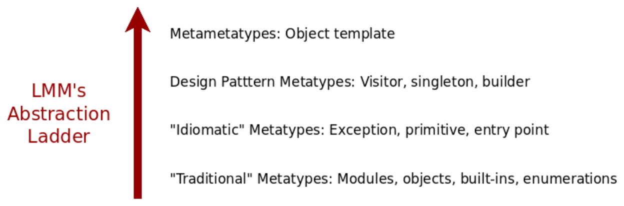

LMM's Abstraction Ladder
Metametatypes: Object template
Design Patttern Metatypes: Visitor, singleton, builder
"Idiomatic" Metatypes: Exception, primitive, entry point
"Traditional" Metatypes: Modules, objects, built-ins, enumerations

Figure 6.31: Abstraction ladder in the *Structural* package.

Object templates

From design patterns, the level of abstraction is raised one final time; Figure 6.31 captures the entire ascent over the abstraction ladder. `Object templates` are to be found at this highest level (Figure 6.30, in orange). They allow the creation of modeling entities that exist only at modeling-time, and demonstrate the power of employing loose metamodeling in this context. `Object templates` were inspired on C++ concepts[67], because they abstract classes with the same shape, though entirely unrelated from the TS's type system perspective. In addition, now that C++ 20 has introduced language-level concept specifications, MASD is looking into projecting `object templates` into the physical domain via the new language feature. Regardless of this new use case, `object templates` have already proven extremely useful to the MRI, and are used extensively throughout MASD's code generator. Their use may not be entirely obvious, however, so a small example is required to clarify how it and other LMM metatypes are employed in practice.

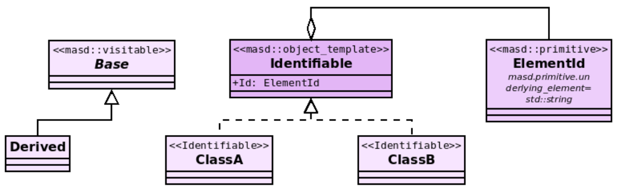


Figure 6.32: Example model with a selection of LMM metatypes.

Example object template

Figure 6.32 does so by portraying a UML class diagram that instantiates `object templates`, `objects`, `primitives` and `visitor`. MASD metatypes are supplied as UML stereotypes, contained within the MASD UML profile. First, we turn out attention to `object templates`. The metatype `Identifiable` dynamically generates a new stereotype, in this case applied to types `ClassA` and `ClassB`; both classes will be generated with a property called `Id`, but there will be no reference to the `object template Identifiable` within the generated C++ code.[68]

With regards to the primitive `ElementId`, its underlying type is defined via UML's tagged values, located just below the class name:

```
masd.primitive.underlying_element=std::string
```

67 Abrahams and Gurtovoy define C++ Concepts as follows (*emphasis theirs*): "A *concept* is a description of the requirements placed by a generic component on one or more of its arguments." (Abrahams and Gurtovoy, 2004) (p. 77) Similar notions exist on other TS such as C#'s generics, though it seems to be particularly developed within C++.

68 As previously mentioned, the generation of *constraints* for concepts, as per C++ 20, is not yet supported.

The tag in the tagged value — `masd.primitive.underlying_element` — represents an entity within MASD's variability domain (*cf.* Section 6.3). The value of the tag represents the type `std::string`, sourced from the C++ Standard Library. It is made accessible to MASD via the PDM `cpp.std`, containing all exposed types within the C++ Standard Library.

Example primitive

Next, we turn to `visitor` support, which, at present, is not without its flaws. The element `Base` is annotated with the stereotype of `masd:visitable`, triggering the generation of a visitor for this base type, dispatching to all of its derived types (`Derived`, in this case). Alas, the approach is now understood to be a misuse of the LMM's type system because the visitor class itself is not present in the class diagram, being instead generated internally.[69] And on the theme of implicit associations, the `object` metatype is also used implicitly in Figure 6.32: UML classes without a MASD stereotype denoting a LMM metatype default to `masd::object`; thus `Base`, `Derived`, `ClassA` and `ClassB` are all implicitly tagged as `masd::object`.

Example visitor and object

The analysis of the model put forward in Figure 6.32 concludes with a demonstration of how `object templates` can be linked back to TS-specific features such as concepts. In the listing below, a sample `print` function was handcrafted, with a template parameter whose name matches the `object template`; the generic function can be instantiated by any type meeting the requirements of the `Identifiable` concept — *e.g.* `ClassA` or `ClassB`. In other words, the C++ concept maps to our logical representation of the `Identifiable object template`. Note that the listing presupposes the presence of all necessary includes for pretty-printing of the `ElementId` primitive. The listing also demonstrates the initialisation of primitive types — *e.g.* `idA` and `idB`.

Example source code

```
template<typename Identifiable>
void print(const Identifiable& ident) {
    std::cout << "Id:" << ident.Id() << std::endl;
}

void caller() {
    ElementId idA("A");
    ClassA a(idA);
    print(a);

    ElementId idB("B");
    ClassB b(idB);
    print(b);
}
```

Listing 6.1: C++ class with SWIG macros.

This example of a logical entity projected into the physical domain brings us into the general topic of projections, which the next section will develop.

6.2.2 *Projections*

As shown previously (*cf.* Section 6.1.6), MASD's physical domain can be thought of as a physical space, with an associated notation for points. The

69 Having intermediary types generated implicitly without representation in the model was found to be detrimental to the modeling process. This is because it is no longer possible to associate non-structural variability with the implicit type. For example, visitors at present must always be named `CLASS_visitor`, with `CLASS` being the name of the type annotated with `masd::visitable`. Nor can one add other configurable features to the type — for example, generate immutable visitation only, *etc.* The correct solution is to force users to create the visitor type as a UML class with the stereotype `masd::visitor`, and then, via tagged values, associate the visitor with the visited base class. This approach will be implemented in the next releases of the MRI.

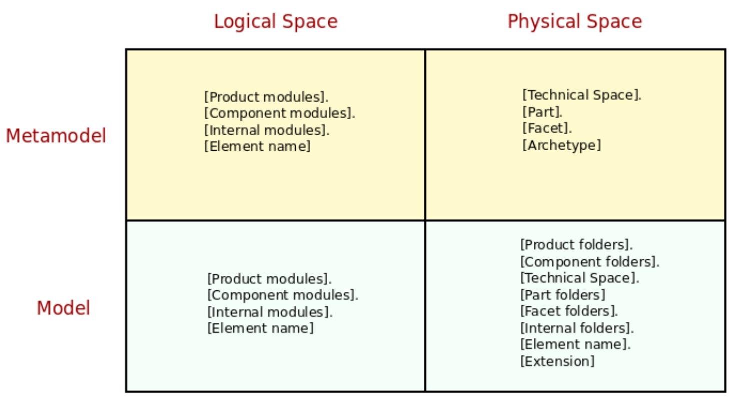
Logical space

logical space is a similar construct, with its own point notation derived from element containment. As a result, much like physical space, logical space is also hierarchical in nature. Since `modules` are the only LMM element that can contain other elements[70], the following notation describes any point in logical space (with the word `modules` omitted due to space constraints):

```
[product].[component].[internal].[element name]
```

Logical point

`Product` stands for product `modules` and represents the set of one or more modules associated with the product name — *e.g.* `Some.Product`, using dot notation, is made up of product `modules` `Some` and `Product`. `Component` stands for component `modules` and represents one or more modules associated with the component (*e.g.* `Some.Component`); and `internal` — *i.e.*, internal `modules` — represents zero or more `modules` used internally within the component (*e.g* `ModuleA.ModuleB`).[71] Finally and predictably, `element name` represents the name of the logical element, *e.g.* `ElementA`. There are clear similarities between this approach and what was put forward in the physical domain; Figure 6.33 joins them together into a single viewpoint.[72,73]



Logical Space
Physical Space
Metamodel
Model
[Product modules]. [Component modules]. [Internal modules]. [Element name]
[Technical Space]. [Part]. [Facet]. [Archetype]
[Product modules]. [Component modules]. [Internal modules]. [Element name]
[Product folders]. [Component folders]. [Technical Space]. [Part folders] [Facet folders]. [Internal folders]. [Element name]. [Extension]

Figure 6.33: Notation for points in physical and logical space.

70 As implied by this statement, MASD does not support inner classes at the logical model level — *i.e.* classes defined within classes, which would map to the LMM as objects defined within objects. These constructs are supported in many TSs such as C++ and C#. Note that inner classes may be generated as part of the projection into physical space; however, their creation cannot be arbitrarily driven by structural variability. As typical in MASD, this feature will only be supported when there are well-defined use cases for their implementation.

71 The sub-typing of `modules` with regards to ownership at a logical level may appear counter-intuitive, given that all of these three types of modules ultimately represent containment. MASD does so because we often treat these groupings separately. For example, product `modules` are at times necessary — when creating include paths for instance — but in other cases omitted. In addition, both product and component `modules` are combined using dot notation when creating folders rather than creating one folder per module. Clearly these entities are treated specially when compared to "regular" `modules`, justifying the distinction. Furthermore, these different types of `modules` are also projected to distinct physical elements (*cf.* Section 6.1.5).

72 For historical reasons, product `modules` are currently named *external* `modules` and component modules are named *model* `modules` in the MRI. These legacy names are part of the evolution of the understanding of the domain architecture, since the purpose of these modules wasn't obvious initially. The MRI will be corrected to match the description here presented.

73 Composite product names are useful to denote product families, *e.g.* `Famly.Product`. Composite component names allow creating groups of components within a product, *e.g.* `ComponentGroup.Component`.

The first interesting point in this comparison is that all points in logical space use the same notation, whether when representing elements in the metamodel (*e.g.* the LMM) or any instance model (*e.g.* any LM). Since there is nothing distinctive about the LMM — it is just a regular model after all — and since we prefer loose metamodeling, there is no requirement to make a distinction between its types and any user types. On the right hand side of the diagram, at the top, we have the previously described notation for points in physical space at the metamodel level (*cf.* Section 6.1.6). Finally, at the bottom right, physical paths are shown — *i.e.* physical points at the model level. These represent the projection of logical elements across to physical space and are a function of:

Point notations

- **The geometry of physical space**, as dictated by the PMM, which enforces regularity — *e.g.* specifies the placement of `product` folders, `component` folders, `part` folders, `facet` folders, *etc.*;
- **Structural variability** in the logical model, which instantiates each of these elements: the product and component names are supplied by the user, as are all internal folders and the element name;
- **Non-structural variability**, dictating the exact configuration to select; for example, what extension to use for C++ headers, whether to express product and component folders, whether to override the name of the technical space folder, and so forth.

Path projection

Paths are just one of many projections within MASD. Almost all logical entities are projected into the physical dimension — the most obvious exception being `object templates`, which at present are consumed by the transform chains during processing and do not have a physical representation.[74] Typically, projections are functions of structural variability, parametrised by non-structural variability, and often implemented as M2T transforms. To simplify matters, we shall ignore non-structural variability for the purposes of the present discussion, as it is covered elsewhere (*cf.* Section 6.3); but it is important to bear in mind that any such projection will offer a number of configurable parameters which will have a significant effect on the result of the projection.

Element projection

Projections are best understood with examples. Figure 6.34 shows an example projection of the logical element `masd::object` (*cf.* Section 6.2.1).

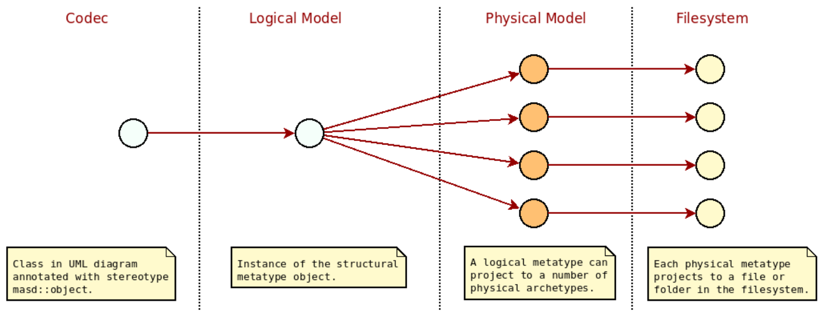

Codec
Logical Model
Physical Model
Filesystem
Class in UML diagram annotated with stereotype masd::object.
Instance of the structural metatype object.
A logical metatype can project to a number of physical archetypes.
Each physical metatype projects to a file or folder in the filesystem.

Figure 6.34: Projection across MASD spaces.

74 As previously mentioned, this is expected to change in the near future with the addition of C++ concept constraints. The C# TS also has a notion of constraints, but as these are more limited, the mapping to object templates will require additional analysis.

Codec representation

In the diagram, an initial representation is used as input to the process; this is known as the *codec representation* and it is designed to be as simple as possible.[75] The idea is to make the creation of extractors a straightforward matter, allowing the implementation of a codec for each required tool, in keeping with the methodology's tenets (P-2 in particular). In addition, we want to keep the number of bespoke transforms in each codec to a bare minimum, leaving all the heavy lifting to common transforms. Figure 6.35 contains a fragment of the codec model, with the key entities.

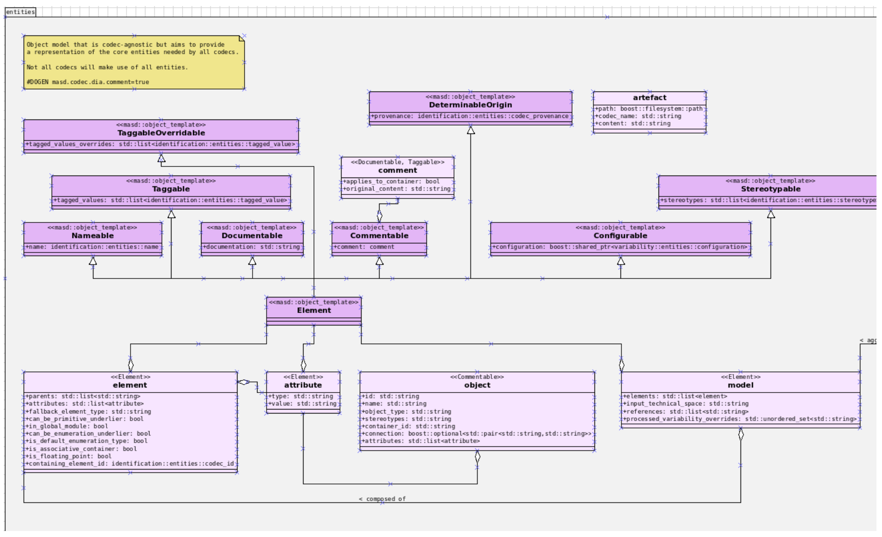

entities
Object model that is codec-agnostic but aims to provide
a representation of the core entities needed by all codecs.
Not all codecs will make use of all entities.
#DOGEN masd.codec.dia.comment=true
<<masd::object_template>> DeterminableOrigin
+provenance: identification::entities::codec_provenance
artefact
+path: boost::filesystem::path
+codec_name: std::string
+content: std::string
<<masd::object_template>> TaggableOverridable
+tagged_values_overrides: std::list<identification::entities::tagged_value>
<<Documentable, Taggable>> comment
+applies_to_container: bool
+original_content: std::string
<<masd::object_template>> Taggable
+tagged_values: std::list<identification::entities::tagged_value>
<<masd::object_template>> Stereotypable
+stereotypes: std::list<identification::entities::stereotype
<<masd::object_template>> Nameable
+name: identification::entities::name
<<masd::object_template>> Documentable
+documentation: std::string
<<masd::object_template>> Commentable
+comment: comment
<<masd::object_template>> Configurable
+configuration: boost::shared_ptr<variability::entities::configuration>
<<masd::object_template>> Element
< agg
<<Element>> element
+parents: std::list<std::string>
+attributes: std::list<attribute>
+fallback_element_type: std::string
+can_be_primitive_underlier: bool
+in_global_module: bool
+can_be_enumeration_underlier: bool
+is_default_enumeration_type: bool
+is_associative_container: bool
+is_floating_point: bool
+containing_element_id: identification::entities::codec_id
<<Element>> attribute
+type: std::string
+value: std::string
<<Commentable>> object
+id: std::string
+name: std::string
+object_type: std::string
+stereotypes: std::string
+container_id: std::string
+connection: boost::optional<std::pair<std::string,std::string>>
+attributes: std::list<attribute>
<<Element>> model
+elements: std::list<element>
+input_technical_space: std::string
+references: std::list<std::string>
+processed_variability_overrides: std::unordered_set<std::string>
< composed of

Figure 6.35: Key entities in the codec model.

Codec projections

The codec representation defines the projection of its elements into the logical model proper, taking these ideas into account; it is always a one-to-one projection, but because LMM elements are highly specialised, many such projections have been defined. By and large, UML stereotypes determine the routing to a logical element — *e.g.*, an element with stereotype of `masd::obejct` will be converted into the LMM's structural metatype of `object`, an element with a stereotype of `masd::enumeration` will be mapped to a structural metatype of `enumeration` and so on. Elements without stereotypes are assigned a default mapping; for example, UML classes without stereotypes default to `masd::obejct`.

Physical projections

Next we have the projections from the LMM into the PMM. These projections are functions that take points in logical space and map them into sets of points in physical space, often spanning multiple regions. Returning to our example, the UML class at the codec level is first projected into a `masd::object`, and then projected into the physical locations depicted by Figure 6.36. The figure uses the same colouring scheme as before, with TS, parts and facets containing archetypes. Its not necessary to go into the details on each archetype shown — hopefully most have self-explanatory names — but it is significant that there are a large number of them (18, in yellow) and their number is expected to grow considerably over time, as more patterns are added to MASD.

75 From the perspective of the domain architecture, the codec representation is merely an implementation detail and as such we will not spend a lot of time describing it. The chapter on the MRI will provide implementation-level details in this regard (*cf.* Chapter 8).

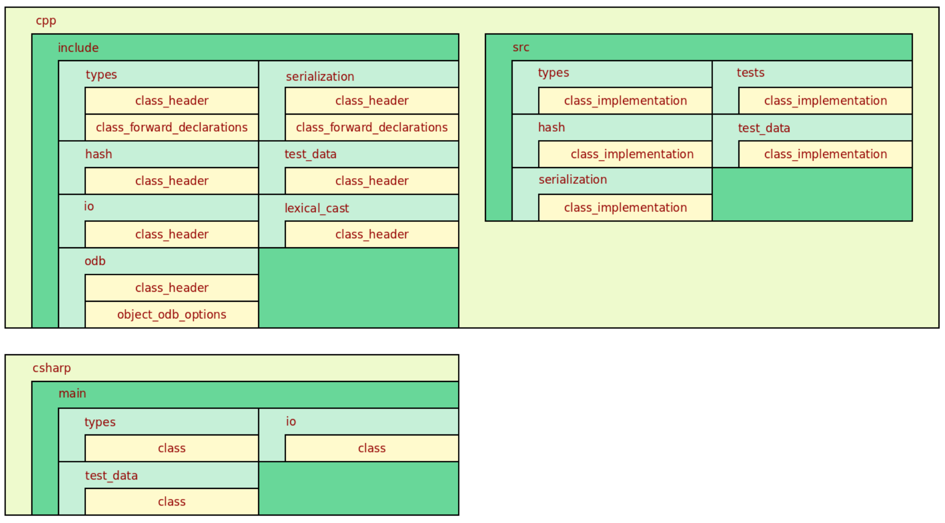


Figure 6.36: Projection of `masd::object` into physical space.

Clearly, not all functionality is required for all use cases; for example, one may require type definitions only, or type definitions with serialisation support, meaning that all other projections would not be necessary. And it is here that we enter the last domain within MASD, dealing with the configurability of logical and physical model elements, as well as the configurability of the projections between spaces.

6.3 VARIABILITY DOMAIN

The third and last domain of interest to MASD is the *variability domain*; it is only concerned with the modeling of non-structural variability. As this choice may be surprising, we begin by justifying the approach (Section 6.3.1), and then move on to discuss the metamodel entities in the VMM (Section 6.3.2). Finally, Section 6.3.3 discusses the VM, which is concerned with how instances of the VMM are used to enable support for Software Product Line Engineering (SPLE) in MASD.

Section Overview

6.3.1 *Approach*

Variability is a vast and complex topic within MDE, so, to avoid confusion, all mentions in this document have been carefully qualified — up to the present section.[76,77] Unfortunately, given its prominence within MASD, it is impractical to enunciate so clearly each use of variability within the domain architecture, as doing so would make naming entities unwieldy. Further-

76 Chapter 6 of (Craveiro, 2021c) summarises our incursion into variability, and introduces, at a high level, all concepts used by MASD in the variability domain.

77 This entailed, for instance, making a clear distinction between structural and non-structural variability, with input variability taken to mean the superset of both kinds; and using *variability*, in isolation, when referencing the entire field of variability modeling within software engineering.

Domain boundaries

more, a natural alignment was observed between certain variability kinds and MASD's domains, meaning that, in practice, confusion seldom arises.[78] For all of these reasons, the variability domain specialises only on non-structural variability; and the term "variability", when used in a MASD-only context, is understood to be synonymous with this kind of variability, with other uses explicitly qualified.

Integrated approach

Once boundaries had been established, the question of how to integrate domain modeling with variability modeling emerged. Clauß (Clauß, 2001) and Thibaut *et al.*'s (Possompès et al., 2010; Possompès et al., 2011) take on the matter was preferred over others, mainly due to their emphasis on a single integrated modeling approach that encompasses variability requirements. The simplicity of the implementation was of particular interest, since having a single model meant augmenting MASD's UML profile with a limited number of variability concepts. Whilst not as expressive as Feature Modeling or Orthogonal Variability Modeling (OVM), the approach is sufficient for the well-defined needs of MASD — especially because it lowers the cognitive load of end-users by reducing the number of concepts needed to model effectively. Having settled on the boundaries and the approach to variability modeling, our efforts then shifted towards identifying the entities of interest within this domain, covered in the next section.

6.3.2 *The Variability Metamodel*

The Variability Metamodel (VMM) is designed to provide variability services to MASD's logical and physical domains. Due to this, it's deeply intertwined with both domains, and it is used in many complex workflows. However, at its core it was created to address two simple needs:

Use cases

- enabling and disabling regions of physical space, such as TS, facets and artefacts;
- configuring various aspects of the projections to physical space: naming the directories for facets, configuring file names and extensions, enabling or disabling certain features in code generation, *etc.*

Since in MASD variability is built atop of UML class diagrams, we made use of tagged values to convey `configuration`.[79] As an example, a `masd:object` can be `configured` to enable two regions of physical space — `types` and `hash` — by supplying the following tagged values:

```
masd.cpp.include.types.enabled=true
masd.cpp.src.types.enabled=true
masd.cpp.include.hash.enabled=true
```

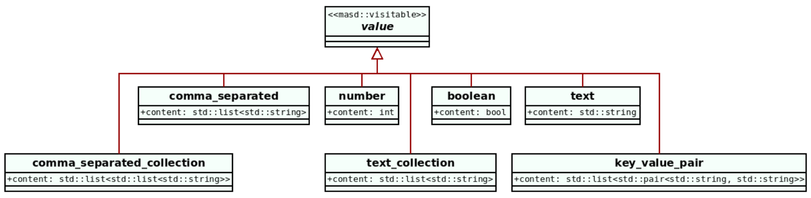
78 Structural variability is modeled in the logical domain, whereas generational variability is deployed mainly within the physical domain. Non-structural variability is dealt with in isolation because it was shown to be self-contained; as we shall see, once modeled, the variability domain is then superimposed over the logical and physical domains.

79 Please note that this is a simplification; MASD does not *require* the use of UML *per se*, as we shall soon see (*cf.* Chapter 7). Historically, however, the majority of MASD's modeling was done using UML class diagrams, and they will remain a first class citizen. In addition, any other representation supported by MASD must be isomorphic to the entities in UML class diagrams.

```
masd.cpp.src.hash.enabled=true
```

Value type system

Boolean values are one of many possible types for tagged values. Over time, MASD accrued many additional types and a type system was created in order to validate user input, as well as to facilitate the processing of these entities within the MRI. Figure 6.37 shows the available value types at present, with more on the pipeline.

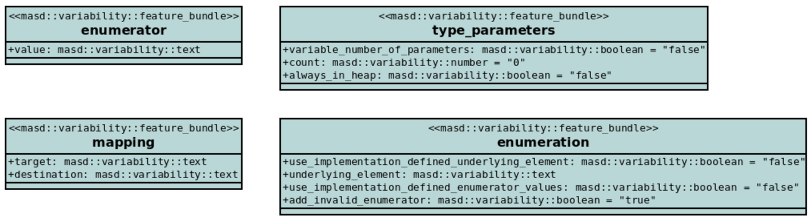


<<masd::visitable>>
value
comma_separated
+content: std::list<std::string>
number
+content: int
boolean
+content: bool
text
+content: std::string
comma_separated_collection
+content: std::list<std::list<std::string>>
text_collection
+content: std::list<std::string>
key_value_pair
+content: std::list<std::pair<std::string, std::string>>

Figure 6.37: Value and its descendant types.

Features, feature bundles

As with values, a similar problem was faced with regards to tag validation. Initially, *ad-hoc* code was written for each new tag as they were introduced but, once enough use cases were collected, the notion was generalised via the introduction of `features` and `configurations`. `Features` implement a simplified version of the concept as found in Feature Modeling, allowing the creation new `configuration points` within the domain architecture. `Features` are grouped into semantically related sets called `feature bundles`. Figure 6.38 shows a small subset of the `feature bundles` defined in the LMM. There we can see that the LMM metatype `feature_bundle` is instantiated, with each instantiation containing a number of `features` of varying types.



<<masd::variability::feature_bundle>>
enumerator
+value: masd::variability::text
<<masd::variability::feature_bundle>>
type_parameters
+variable_number_of_parameters: masd::variability::boolean = "false"
+count: masd::variability::number = "0"
+always_in_heap: masd::variability::boolean = "false"
<<masd::variability::feature_bundle>>
mapping
+target: masd::variability::text
+destination: masd::variability::text
<<masd::variability::feature_bundle>>
enumeration
+use_implementation_defined_underlying_element: masd::variability::boolean = "false"
+underlying_element: masd::variability::text
+use_implementation_defined_enumerator_values: masd::variability::boolean = "false"
+add_invalid_enumerator: masd::variability::boolean = "true"

Figure 6.38: Fragment of feature bundles defined within the LMM.

Features for features

Note that `feature bundles` themselves also make use of the variability machinery. For example, `features` have `binding points` — that is, each `feature` must enunciate the set of meta-entities that can legally make use of it — and these are declared via variability, as are other configurable elements:[80]

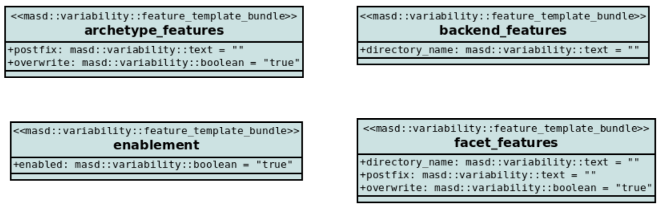
```
masd.variability.default_binding_point=element
masd.variability.key_prefix=masd.type_parameters
```

`Features` are useful in isolation, but MASD's approach of having a dynamically expanding PMM posed a challenge: as new TSs, parts, facets and archetypes

80 `Bindings` are deficient at present, in that they do not support referring to specific elements in the LMM. For example, the correct binding for the `enumerator feature bundle` should have been `masd::enumerator`, given that this is the only metatype in the LMM that can make use of this `feature`. In the future, the address of the logical elements will be used for the `binding`.

Feature templates

were added, there was a need to model individually their respective features, such as for example `enabled` as per previous listing. The process was error prone and repetitive, so the notion of `feature templates` was introduced. These are abstract `features` which must be instantiated over a domain in order to become concrete `features` — *i.e.*, made available to end-user diagrams. Figure 6.39 shows how `features` such as `enabled` are defined as templates.



<<masd::variability::feature_template_bundle>>
archetype_features
+postfix: masd::variability::text = ""
+overwrite: masd::variability::boolean = "true"
<<masd::variability::feature_template_bundle>>
backend_features
+directory_name: masd::variability::text = ""
<<masd::variability::feature_template_bundle>>
enablement
+enabled: masd::variability::boolean = "true"
<<masd::variability::feature_template_bundle>>
facet_features
+directory_name: masd::variability::text = ""
+postfix: masd::variability::text = ""
+overwrite: masd::variability::boolean = "true"

Figure 6.39: Fragment of feature templates defined within the PMM.

Each of these modeling elements declares a domain over which template instantiation is to be performed. For example, `archetype_features` has the following tagged value:

```
masd.variability.instantiation_domain=masd.archetype
```

Template domains

The domain `masd.archetype` covers all available archetypes across the entirety of the PMM. Other domains exist such as `masd` — spanning the whole physical space — `masd.facet`, including only facets — and so forth. This scheme allows the fine-grained definition of `features` across the different regions of the PMM. At present, the main source of domains has been the geometry of physical space, but there is no direct connection between the domain as a variability concept and the PMM; these are merely seen as sets of strings, meaning other applications are possible. At present, no additional use cases have emerged.

Iterative process

Significantly, the VMM resulted from the application of the physical modeling process (*cf.* Section 6.1.2) to the MRI itself. All of the generalisations presented here emerged from a long iterative process, with several years of experimentation — from detecting SRPP's within the variability domain, through to modeling them in the LMM and ultimately to generating code to encapsulate them as trivial structural functions — and this process is still ongoing. For example, one area where support is limited at present is in declaring relationships between `features`; once implemented, it will allow solving for valid configurations.[81] And `configurations` brings us to the final topic within variability: the Variability model (VM).

81 The matter is best understood via an example. Given two types `A` and `B`, where type `A` has a member variable of type `B`; if type `A` enables certain regions of physical space — for example the `hash` and `serialization` facets — then this implies that these regions must also be enabled on type `B` in order to create a valid `configuration`. Solving, via methods such as BDDs (Czarnecki and Wasowski, 2007) or SAT (Batory, 2005), will allow the automated resolution of these dependencies. This is an area of future work.

6.3.3 *The Variability Model*

Configurations

All entities described in the VMM thus far are mainly used for the code generation of variability support in the MRI. However, a second aspect of variability is the creation of run-time `configurations`, which instantiate the available `features` with specific values. The simplest case of a `configuration` was already covered, which is to add all `configuration points` to the affected elements via tagged values. However, a problem soon emerged with regards to reusing `configurations`: once `features` were made available and used throughout, it became obvious that component models for a given product shared a great degree of commonality configuration-wise, as did products from the same product line.

Profiles

This use case was addressed by introducing `profiles` to the LMM[82]. These are bundles of `configuration points` that can be bound to logical elements such as component models and objects. With the introduction of `profiles`, the VM took a renewed relevance, meaning each product can now define a `configuration model` describing a `configuration language` for the product or product line, and it can then be reused across a set of component models. Figure 6.40 shows a fragment of the MRI's `configuration`, with a number of `profiles` (stereotype `masd::variability::profile`). The use of inheritance enables the construction of elaborate trees of `profiles`, supporting simultaneously minimal duplication and fine grained configuration.

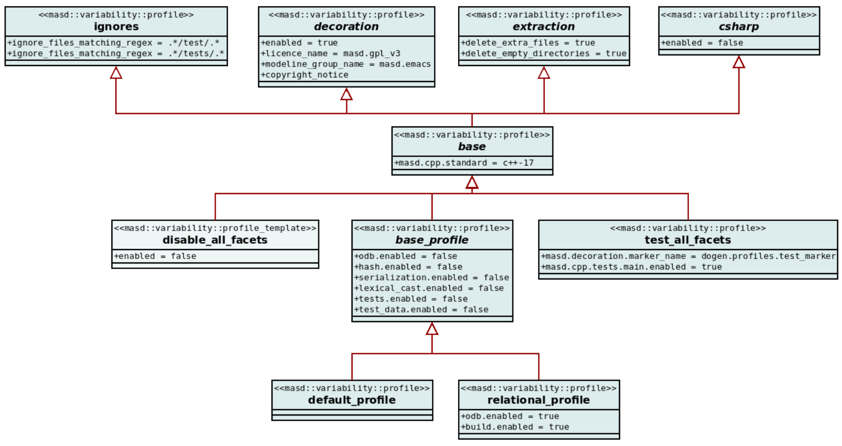

<<masd::variability::profile>>
ignores
+ignore_files_matching_regex = .*/test/.*
+ignore_files_matching_regex = .*/tests/.*
<<masd::variability::profile>>
decoration
+enabled = true
+licence_name = masd.gpl_v3
+modeline_group_name = masd.emacs
+copyright_notice
<<masd::variability::profile>>
extraction
+delete_extra_files = true
+delete_empty_directories = true
<<masd::variability::profile>>
csharp
+enabled = false
<<masd::variability::profile>>
base
+masd.cpp.standard = c++-17
<<masd::variability::profile_template>>
disable_all_facets
+enabled = false
<<masd::variability::profile>>
base_profile
+odb.enabled = false
+hash.enabled = false
+serialization.enabled = false
+lexical_cast.enabled = false
+tests.enabled = false
+test_data.enabled = false
<<masd::variability::profile>>
test_all_facets
+masd.decoration.marker_name = dogen.profiles.test_marker
+masd.cpp.tests.main.enabled = true
<<masd::variability::profile>>
default_profile
<<masd::variability::profile>>
relational_profile
+odb.enabled = true
+build.enabled = true

Figure 6.40: Fragment of the MRI configuration.

Profile templates

A second point of interest in the figure is the use of `profile templates`. These follow the same logic as do `feature templates`, allowing for the instantiation of a `configuration point` over a domain. Its main use, as per the image, is in enabling or disabling all regions of physical space of a given type (facets, in the example). Much like `feature templates`, they are very useful given the ever expanding geometry of PMM, saving users from duplicating `configuration points`.

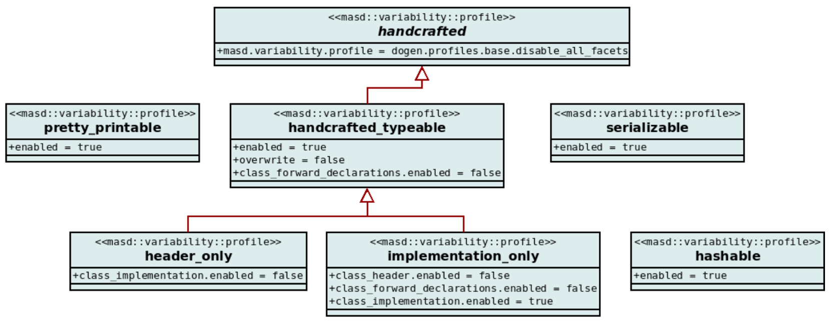
DSL

As already alluded to, another extremely important aspect of `profiles` is how, via their names, they can be used to create a very useful DSL that describes

82 The name `profile` is unfortunate because it is easily conflated with the notion of "UML profile". Presently, there is ongoing analysis to determine a better name for this concept.

the intended characteristics of various model elements. Figure 6.41 shows a number of `profiles` used in the MRI, all named after the abilities they convey, such as for example `serializable` or `hashable`. To further increase the relevance of this DSL, one can associate a colouring scheme with profiles (*cf.* Section 6.1.5.3), making the diagrams visually distinctive.

Colouring



<<masd::variability::profile>>
handcrafted
+masd.variability.profile = dogen.profiles.base.disable_all_facets
<<masd::variability::profile>>
pretty_printable
+enabled = true
<<masd::variability::profile>>
handcrafted_typeable
+enabled = true
+overwrite = false
+class_forward_declarations.enabled = false
<<masd::variability::profile>>
serializable
+enabled = true
<<masd::variability::profile>>
header_only
+class_implementation.enabled = false
<<masd::variability::profile>>
implementation_only
+class_header.enabled = false
+class_forward_declarations.enabled = false
+class_implementation.enabled = true
<<masd::variability::profile>>
hashable
+enabled = true

Figure 6.41: Sample MRI profiles.

Binding

Once defined, `profiles` can then be `bound` to models and model elements in one of two ways: either via variability or via stereotypes. The latter takes the same approach as object templates — *i.e.* one can populate element stereotypes with one or more profiles names, thus `binding` the `profile` with the element.[83] The former is achieved by applying the `profile` tag to the element in question:

```
masd.variability.profile=default_profile
```

Binding scopes

To facilitate the detection of `binding` errors — such as `binding` a `profile` that was meant for a model against an element such as `object` — MASD allows setting the scope of the `binding` on a `profile`; *e.g.*, the following `configuration point` ensures that a `profile` can only be bound to a model:

```
masd.variability.binding_point=global
```

Whilst workable, this high-level specification of scopes does not address all of the present use cases. As with `feature binding`, more work is needed in order to satisfy the fine-grained `binding specifications` we need for the current uses across logical and physical domains. And with this, we have now introduced all of the core domains that make up MASD's domain architecture. What remains is to join the dots between these three domains, forming a combined space.

83 The rules defining the application of multiple `profiles` are, at present, deceptively straightforward: each `profile` is applied, in turn, in the order defined by the `binding`. However, there are clear limitations with this approach such as the definition of conflicting configurations (*e.g.* enabling feature `A` in profile `P0` and disabling feature `A` in profile `P1`). As with solving for valid configurations, an element of upfront validation is required to, at a minimum, alert users to potential conflicts.

6.4 the logical-physical space

The domain architecture as described thus far suggests three unconnected domains with their associated models and metamodels, which may be composed to address the methodology's requirements (*cf.* Section 4). MASD has indeed started in this disjointed manner but, as the problem domain understanding deepened, they were fused together conceptually and the amalgam became the *Logical-Physical Space* (*LPS*). The key idea behind the LPS is that the PMM, LMM and the VMM are distinct dimensions of a multidimensional space, but they only make sense when viewed as a whole. The LPS is designed to allow elements to move seamlessly from representation to representation, all the while catering for configurability — as shown in Figure 6.42.

Fused domains

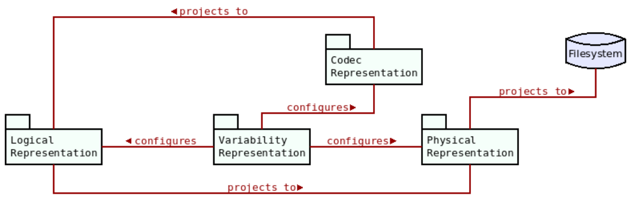


Figure 6.42: High-level view of MASDs LPS.

From this standpoint, MASD's role is two-fold: a) to define the composition of all dimensions in the LPS — that is, the metamodels with their elements and associations; and, b) to define a framework of projections between LPS dimensions. The MRI is the canonical implementation of both of these points. The framework is responsible for taking models from a codec representation, processing them through a series of transform chains that aggregate transforms of various kinds (*e.g.* M2M, M2T, T2T), most of which parametrised by non-structural variability, to their ultimate destination which are files and directories in the filesystem. More succinctly: the *LPS is MASD's domain architecture*.

Framework of projections

And so concludes our incursion into MASD's domain architecture, which also marks the end of our exposition of the methodology and components. The next part of this work will concern itself with its practical application, starting with its approach to the modeling activity itself.

Part III

APPLICATION

7 LITERATE MODELING WITH ORG-MODEL

I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature*. Hence, my title: "Literate Programming."*
Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer *what to do, let us concentrate rather on explaining to* human beings *what we want a computer to do.*

— Donald Knuth (Knuth, 1984)

Approach

HAVING EMPLOYED MDE techniques extensively for over a decade, we became keenly aware of the importance of both the medium and tooling with which to compose and edit models, as well as of the significance of its integration with the remainder of the software engineering stack. This chapter concerns itself with *org-model*, a new technology developed within MASD that benefits from lessons learned and is designed specifically to satisfy our modeling use cases. Org-model marks MASD's internal departure from graphical representations, and underlines a new emphasis on textual representations and literate modeling.[1]

Chapter overview

The chapter is organised as follows. Section 7.1 motivates the need for a new approach, and points to the general direction in which to follow. Section 7.2 performs a brief literature review on the topic, with particular attention paid to literate modeling.[2] An overview of org-mode is then provided by Section 7.3, and org-model — a solution built atop of it — is put forward by Section 7.4. The chapter ends by discussing the benefits and drawbacks of the new approach (Section 7.5). Let's start by looking into why a change was necessary in the first place.

7.1 MOTIVATION

UML integration

Pervasive integration with development tooling is one of MASD's core tenets (P-2). As the previous sections alluded to, great care was taken in creating a codec representation, specifically designed to facilitate tooling integration (*cf.* Section 6.2.2). The idea is to lower barriers to entry, allowing bindings to any

1 To be clear, this chapter discusses MASD's internal modeling needs. Our approach may be suitable to others, wholesale, or merely seen as an experience report with restricted application. Either way, MASD remains strictly neutral with regards to its users' choice of tooling.

2 The topic is not without its representation in the literature, and certainly much could be said about it. However, given the role of this chapter in the present manuscript, the review was circumscribed to only items with a direct impact on our solution. Clearly, a thorough state of the art review would be beneficial, but its left as a future direction.

modeling tool used by developers. The initial expectation was that extraction — *i.e.* projections from external TSs into MASD — would entail parsing a file format in JSON or XML, preferably XMI (*cf.* Section 3.3.3), because UML was seen as the obvious language for model editing. For the majority of MASD's lifespan, Dia[3] has been used in this regard. Dia is a general-purpose FOSS diagramming tool, with support for UML amongst great many other notations. Figure 7.1 shows Dia with a fragment of a model from MASD's MRI.

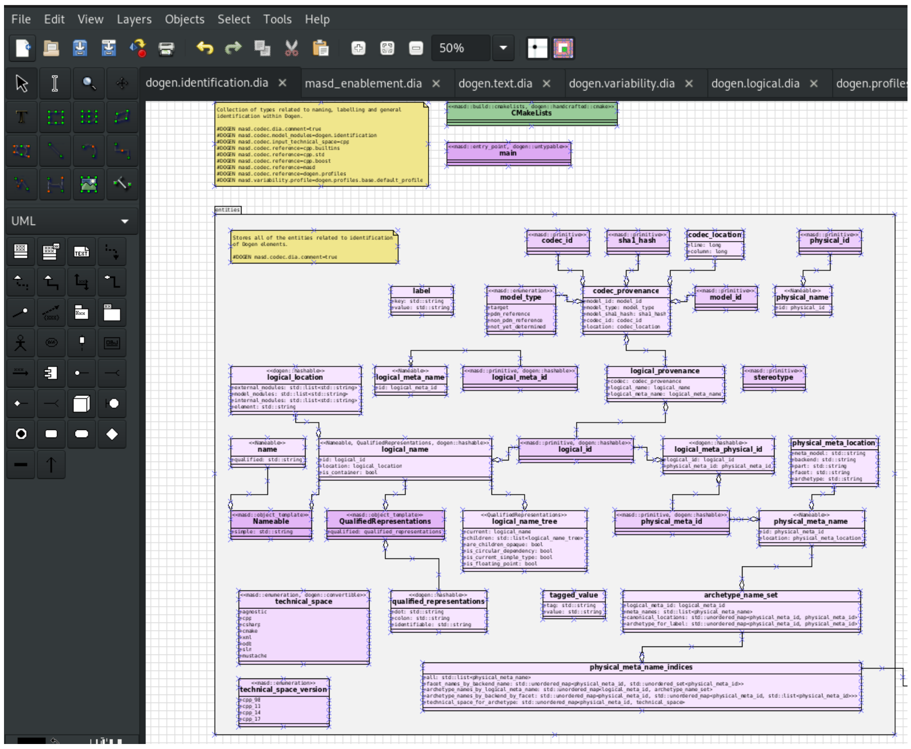


Figure 7.1: Dia modeling tool with a MASD model.

From a perspective of industrial-grade modeling, Dia suffers from significant deficiencies: it has limited understanding of UML — not much beyond seeing it as a set of shapes — and supports several core features in an inconsistent manner. For example, model-level stereotypes and tagged values are not available, though they exist for other elements — hence why these have been added to UML notes via a MASD-specific extension, as demonstrated in Figure 7.1 (top-left, in yellow). That said, these limitations were incidental to MASD because our express intent is to avoid coupling the domain architecture to any one tool, and to use tooling merely as a means for model composition. In other words, the richer the feature set of the tool, the higher the risk of creating unwanted dependencies between it and MASD; conversely, having basic UML support acts as a mitigating factor.

Dia Characteristics

Nevertheless, after over a decade of extensive Dia usage, an unexpected conclusion was reached: the tool had begun to have a restrictive impact in MASD's development, and on the MRI in particular (*cf.* Chapter 8). A reflection on the challenges faced revealed that these issues were not related to Dia *per se* but were instead properties of graphical modeling in a MASD context — all the

Tooling challenges

3 https://gitlab.gnome.org/GNOME/dia/

more noticeable because models are central to our work.[4,5] Whilst a graphical representation aids comprehension — particularly when carefully curated and aided by colour schemes, as is done in the MRI — a large proportion of the modeling work is related to editing elements and their properties. Often times it entails going backwards and forwards between different representations, as depicted by Figure 7.2.

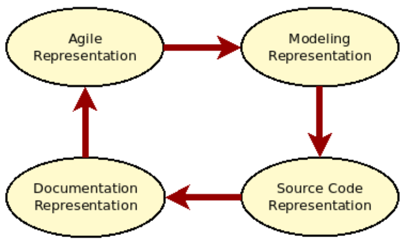


Figure 7.2: Information flow across representations in MASD.

Information flows

The image provides a simplified view of information flows across representations within the MRI, as observed during the development of new features.[6] Typically, one begins with a textual representation in natural language, captured as a user story during an agile sprint. The story may evolve over time, remaining in analysis until mature and ready for implementation. At this stage, the story is often then modeled in a graphical representation (*i.e.*, Dia), and one begins to iterate between it and a source code representation during one or more sprints. Finally, as the story nears completion, work commences on updating the documentation in light of the changes it brought about, which in turn may spur the development of new and related stories, change the understanding of existing stories, and so forth, in a continuous loop that lasts the lifetime of the software product.

Representation normalisation

With the exception of modeling, all representations are in a textual form. Thus, an obvious way in which to reduce friction in information flow is to decrease the impedance mismatch between representations; this can be achieved by normalising them all to a similar textual form. To understand how best to go about this, we consulted the literature.

7.2 LITERATURE REVIEW

Our views on the limitations of graphical representations are amply echoed within the literature, such as by the work of Arlow *et al.* (Arlow, Emmerich, and Quinn, 1998), subsequently augmented by (Arlow and Neustadt, 2004). There, Arlow *et al.* substantially advanced our own diagnosis:

4 Note that our intention is not to make a general statement about graphical modeling — though, as our review of adoption literature points out, perhaps such a case can indeed be made (*cf.* Section 2.1, and also Chapter 3). Instead, our personal view is that textual modeling is better suited to our workflow, because, as software developers, manipulating text is at the core of our profession.

5 For a more detailed discussion on this subject, please consult the MRI's release notes when this feature was introduced. (Craveiro, 2021a)

6 For additional details on how agile is used in the MRI's development, see Section 8.1.3.

> One problem you will find with UML models (and in visual models in general) is that the valuable information captured in the model is only accessible to those who know the visual syntax of the modelling language. In a sense, valuable information about the business becomes encrypted in a concise, elegant modelling language [...].
>
> In fact, it is not just the visual syntax of UML models that creates problems. If you need to access the information embedded in a model, you may also need to know how to work a CASE tool to navigate that information effectively. [...]
>
> Also, unless you already know the general "shape" of a model, knowing precisely where to start with either the model in a CASE tool, or with a generated report can be difficult. (Arlow and Neustadt, 2004) (p. 88)

Challenges with graphical modeling

Moreover, their suggestion on how to address these issues pointed towards literate modeling, which resonated closely with MASD's needs:

> Literate modelling applies of Knuth's idea of Literate Programming (Knuth, 1984) to UML models. The approach is very simple — you interleave UML models with a narrative text that explains the model to both the author of the model and to all the roles discussed above [e.g., the actors involved in the development process].

Literate modeling

Whilst in overall agreement with their diagnosis, we nonetheless disagreed with the proposed solution, preferring a text-only approach over the suggested content interleaving. Text was then the direction of travel, but there was still an important decision to be made with regards to the specific notation to employ. Research on textual representations for modeling languages has a prolific presence in the literature; our excursion uncovered approaches such as Model Oriented Programming (MOP) — the work of Badreddin and Lethbridge being of particular interest (Badreddin and Lethbridge, 2013), as well as Executable UML (Mellor, Balcer, and Foreword By-Jacoboson, 2002). However, the requirements of MASD directed us away from such complex endeavours since models within MASD cater only for structural aspects, with simpler behaviours encoded in the geometry of physical space (*cf.* Section 6.1). In this light, complex behaviour is entirely relegated to the programming language representation, thus negating the main advantages of an expressive modeling language in the vein on MOP.

MOP

In addition to the more comprehensive MOP languages, a host of lightweight representations was also found, including MetaUML[7], TextUML[8], PlantUML[9], yUML[10] and others of a similar ilk, all characterised by a compact scope and a focus on outputting diagrams as images; one could conceivably target the structural subset of any one of these implementations to satisfy MASD's requirements. Alas, by carefully observing MASD's information flow (Figure 7.2), these approaches were seen to be better suited for extractive projections

FOSS modeling approaches

7 http://metauml.sourceforge.net
8 http://sourceforge.net/projects/textuml/
9 https://github.com/plantuml/plantuml
10 https://yuml.me/

out of MASD rather than as injections. It is so because both the agile representation and the documentation representation involve natural language; the literate modeling approach is closer to either of these two representations than it is to the source code representation.[11]

Org-mode

The analysis of information flows also revealed that both MASD's documentation as well as its agile processes rely extensively on org-mode[12], a textual representation available in the Emacs editor used for MRI development — but also supported by a host of other editors and IDEs.[13] Org-mode had been shown to be a competent environment for literate programming and reproducible research by Schulte *et al.* (Schulte et al., 2012); its tree-like structure and support for annotations provide ample scaffolding with which to map all of the modeling constructs required to represent UML constructs.

Tooling ecosystem

Furthermore, as Schulte *et al.* had already noticed, org-mode has a broad and diverse tooling ecosystem that can be applied directly to both modeling and the weaving of MASD's information flow. The following should suffice as a demonstration of the strength of the tooling ecosystem: org-roam[14] augments org-mode with support for *zettelkasten*[15]; org-brain[16] adds support for concept mapping[17]; and org-ref[18] manages citations, cross-references, indexes, glossaries and bibliographies for org-mode documents.[19] These are all useful tasks in a modeling environment driven by literate modeling and DDD (*cf.* Section 5.3). With this in mind we begun to explore the possibility of modeling using org-mode documents. But before delving into how this was achieved, we shall first provide a brief overview of org-mode.

7.3 overview of org-mode

Markup languages

There are a great number of markup languages widely used in software development, starting with arguably the most known of all, HTML.[20] Whilst org-mode is part of the family of markup languages, it distinguishes itself due

11 In other words, it is not the case that diagrams require contextual text; instead, the text describing a software product can be augmented by contextual diagrams.

12 https://orgmode.org/

13 At the time of writing, org-mode plugins were found for Visual Studio Code, Vim, Atom, and Sublime. Note that this is not an exhaustive list. However, not all features are supported and Emacs is still its reference implementation.

14 https://www.orgroam.com/

15 Wikipedia defines *zettelkasten* as follows: "The zettelkasten [...] is a method of note-taking and personal knowledge management used in research and study. [...] A zettelkasten consists of many individual notes with ideas and other short pieces of information that are taken down as they occur or are acquired." (Zettelkasten, 2021)

16 https://github.com/Kungsgeten/org-brain

17 Wikipedia states that "A concept map or conceptual diagram is a diagram that depicts suggested relationships between concepts.[1] Concept maps may be used by instructional designers, engineers, technical writers, and others to organize and structure knowledge." (Concept map, 2021)

18 https://github.com/jkitchin/org-ref

19 All of our research, including writing the present manuscript, was performed using org-mode, making ample use of org-mode's tooling ecosystem — for example for note taking, reference management, *etc.*

20 Wikipedia defines markup languages as follows: "In computer text processing, a markup language is a system for annotating a document in a way that is visually distinguishable from the content. It is used only to format the text, so that when the document is processed for display, the markup language does not appear." (Markup language, 2021)

to a deep integration with the Emacs editor (Stallman, 1981) and its programming language Emacs Lisp (Lewis, LaLiberte, Stallman, et al., 1993). In the previously mentioned paper, Schulte *et al.* convey enthusiastically both the simplicity and the power of this combination (*emphasis ours*):

Org-mode characterisation

> At the core of Org-mode is the Emacs text editor and Emacs Lisp, a dialect of Lisp that supports the editing of text documents. [...] Org-mode extends Emacs with a simple and powerful markup language that turns it into a language for *creating, parsing, and interacting with hierarchically-organized text documents*. Its rich feature set includes *text structuring*, project management, and a publishing system that can export to a variety of formats. *Source code and data are located in active blocks*, distinct from text sections, where "active" here means that code and data blocks can be evaluated to return their contents or their computational results. (Schulte et al., 2012)

Org-mode and MASD

Since, for our purposes, we need not leverage all org-mode features, we shall only unpick those which are relevant to MASD. First and foremost, the hierarchical nature of org-mode is a great fit for MASD's modeling data, given its focus on structural modeling as emphasised constantly in the domain architecture (*cf.* Chapter 6). Secondly, the support for source code interleaving is also of great interest to MASD, allowing us to carry fragments of code or even complete source files within a model — a theme for later development (*cf.* Section 8.1.5). Finally, the fact that org-mode is tightly integrated with Emacs — the tool used for the majority of software development in MASD — is a very important point and goes to the heart of our ideas on pervasive integration (P-2); modeling should be integrated as tightly as possible with existing developer workflows.[21]

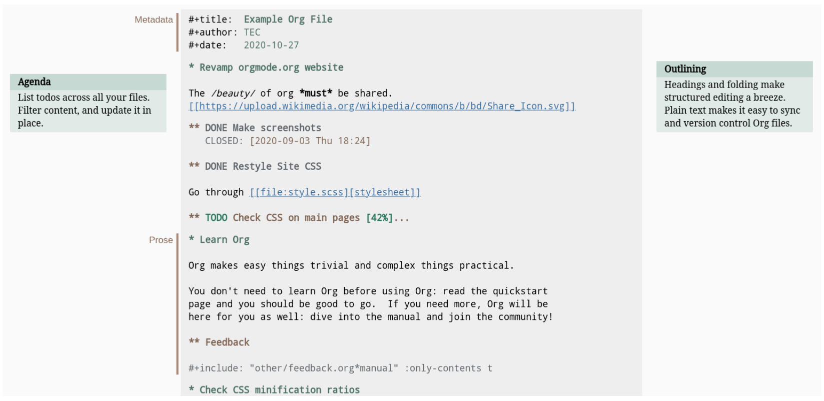
Document and headlines

As org-mode is well documented, physical modeling (*cf.* Section 6.1.2) relied on existing sources such as (Org Syntax, 2021). Two brief examples will be shown to demonstrate the morphology of org-mode files. Figure 7.3 displays a fragment of a sample org-mode document, as sourced from the previously cited project's website.[12] The document is at the centre, with a darker background; to its left and right are annotations to help identify elements and features. The outline structure of org-mode is created via the star or asterisk character (`*`), with the number of stars conveying depth — shown in the picture with headlines of varying colours.

Metadata

Different kinds of metadata can be associated with the document — such as `title`, `author` and `date`, top left — as well as task management annotations — for example, the so called *TODO keywords* are in upper case, such as `TODO` and `DONE`. In addition, though not shown in the picture, each headline can have one or more associated tags, in the form `:TAG_NAME:`. Note also how headlines can be folded, making them non-obtrusive (headline "Check CSS on main pages"). Finally, the example also demonstrates a variety of mechanisms for linkage to other sources of information: from HTTP links, to links to arbitrary files in the filesystem, to including other org-mode documents via the `#+include` keyword.

21 By this we are not stating that all developers working on MASD should use Emacs or even org-mode; instead, they should either perform a similar analysis to the present one, figuring out how to best integrate modeling to their workflow, or find ways of using org-mode with their current setup.

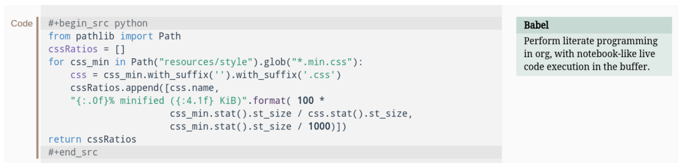

Metadata
Prose
Agenda
List todos across all your files. Filter content, and update it in place.
Outlining
Headings and folding make structured editing a breeze. Plain text makes it easy to sync and version control Org files.

```
#+title:  Example Org File
#+author: TEC
#+date:   2020-10-27

* Revamp orgmode.org website

The /beauty/ of org *must* be shared.
[[https://upload.wikimedia.org/wikipedia/commons/b/bd/Share_Icon.svg]]

** DONE Make screenshots
   CLOSED: [2020-09-03 Thu 18:24]

** DONE Restyle Site CSS

Go through [[file:style.scss][stylesheet]]

** TODO Check CSS on main pages [42%]...
* Learn Org

Org makes easy things trivial and complex things practical.

You don't need to learn Org before using Org: read the quickstart
page and you should be good to go.  If you need more, Org will be
here for you as well: dive into the manual and join the community!

** Feedback

#+include: "other/feedback.org*manual" :only-contents t

* Check CSS minification ratios
```

Figure 7.3: Fragment of example org-mode document. *Source:* Project website.

Our second example, also sourced from the org-mode website, demonstrates the usage of source code blocks within a org-mode document (Figure 7.4). The `#+begin_src` keyword denotes the start of the source code block, followed by the language used within the block (`python` in this particular case). In addition to providing syntax highlighting relative to the language used, org-mode also allows editing the source code block within an environment that is similar — if not exactly identical — to a full programming environment, with access to IDE-like features such as code completion, compilation errors and the like.

Source code blocks


Code
Babel
Perform literate programming in org, with notebook-like live code execution in the buffer.

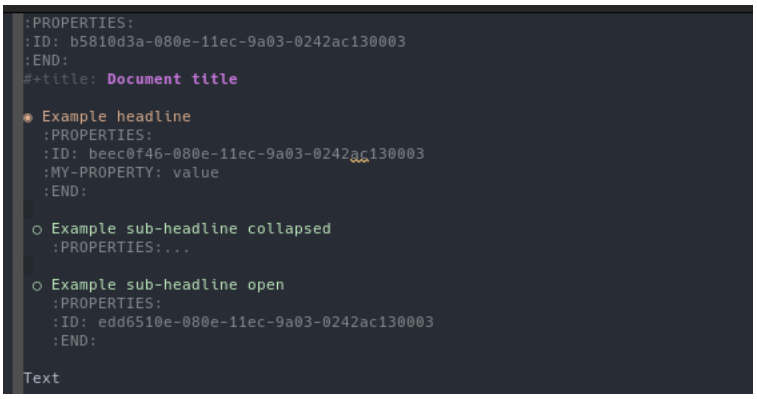
```
#+begin_src python
from pathlib import Path
cssRatios = []
for css_min in Path("resources/style").glob("*.min.css"):
    css = css_min.with_suffix('').with_suffix('.css')
    cssRatios.append([css.name,
    "{:.0f}% minified ({:4.1f} KiB)".format( 100 *
                     css_min.stat().st_size / css.stat().st_size,
                     css_min.stat().st_size / 1000)])
return cssRatios
#+end_src
```

Figure 7.4: Fragment with source code blocks. *Source:* Project website.

The third and final example focuses on property drawers. Documents, headlines and other org-mode elements can be associated with an arbitrary number of properties. As with headlines, these can be kept open or folded away for convenience. Figure 7.5 demonstrates the use of property drawers, both at the document level, as well as at the headline level. In the example, we show the `ID` property associating a UUID with each element, including the document itself (at the top). One of the property drawers is folded away (headline "Example sub-headline collapsed"). For completeness, we also created a sample property called `MY-PROPERTY` that is only associated with the top-most headline "Example headline".

Property drawers

This brief overview portrays only a sliver of org-mode's power and abilities. Nonetheless, these examples demonstrate all of the core features needed by MASD in order to store its models within an org-mode document. The next section shall now explain how this was achieved.



```
:PROPERTIES:
:ID: b5810d3a-080e-11ec-9a03-0242ac130003
:END:
#+title: Document title

● Example headline
  :PROPERTIES:
  :ID: beec0f46-080e-11ec-9a03-0242ac130003
  :MY-PROPERTY: value
  :END:

 ○ Example sub-headline collapsed
   :PROPERTIES:...

 ○ Example sub-headline open
   :PROPERTIES:
   :ID: edd6510e-080e-11ec-9a03-0242ac130003
   :END:

Text
```

Figure 7.5: Org-mode document with property drawers.

7.4 CREATING THE ORG-MODE CODEC

Given that the purpose of the new codec was to introduce org-mode based modeling, we decided to name it *org-model*.[22] The first step to introducing a new codec to MASD involves locating an existing platform library that can parse files of this format, implemented on the TS of choice for the MRI (C++). As mentioned before, there had been an expectation that input formats for codecs would be in JSON, XML or other popular data interchange formats, for which there are many supporting libraries, providing adequate platforms from which to work from. For org-mode, the number of choices was very limited; few libraries could be located, and of those only `OrgModeParser`[23] was of industrial grade quality, with support for all the required features. Unfortunately, its large number of dependencies and sparse maintenance made it incompatible with the approach used in the MRI.

Library challenges

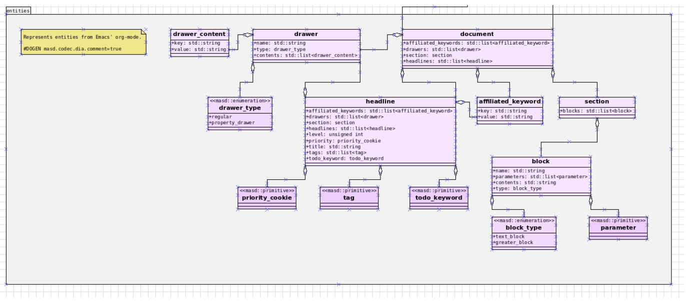


Figure 7.6: Dia representation of org-mode model.

Having no other alternative, we implemented org-mode support by extending the MRI. This was done by creating a component model to store all of the data structures required by the codec (Figure 7.6), and writing a parser that reads input files in org-mode notation and instantiates entities with MASD's org-mode model. Thanks to the codec pipeline, the integration of the org-mode model with the remainder of the framework was quite straightforward, and required only two trivial M2T transforms that are specific to org-mode. The

New library implementation

22 The codec name is not of great significance in the near future as it is only used internally, and as a way to name the set of conventions MASD has created for its org-mode documents. As a future direction, we would like to add org-model specific support to Emacs to facilitate the authoring of org-model diagrams.

23 https://github.com/mirkoboehm/OrgModeParser

key decisions were related to the mapping of the modeling elements to the org-mode document elements, which we shall now describe.

The mapping was surprisingly simple:

- The org-mode document itself was mapped to a model. Its stereotypes and tagged values are supplied as properties in the document-level property drawer.

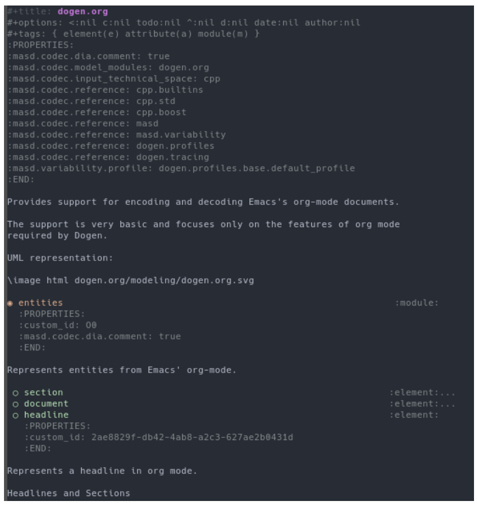
- Headlines were mapped to different modeling elements, depending on their tags. We mapped each meta-element in the codec model to its own tag: `module`, `element` and `attribute`.

Codec to org-mode mapping

- Finally, the content of the headline was used to populate the comment for the codec element.

The mapping can be seen in action on Figure 7.7, depicting an org-mode representation of the org-mode model previously shown in its Dia incarnation (Figure 7.6). The property drawers have been left open for illustrative purposes but, even in their expanded state, it should be apparent how documentation now dominates the model when compared to its UML representation, taking us much closer to literate modeling without subverting the org-mode file format.

Mapping example



```
#+title: dogen.org
#+options: <:nil c:nil todo:nil ^:nil d:nil date:nil author:nil
#+tags: { element(e) attribute(a) module(m) }
:PROPERTIES:
:masd.codec.dia.comment: true
:masd.codec.model_modules: dogen.org
:masd.codec.input_technical_space: cpp
:masd.codec.reference: cpp.builtins
:masd.codec.reference: cpp.std
:masd.codec.reference: cpp.boost
:masd.codec.reference: masd
:masd.codec.reference: masd.variability
:masd.codec.reference: dogen.profiles
:masd.codec.reference: dogen.tracing
:masd.variability.profile: dogen.profiles.base.default_profile
:END:

Provides support for encoding and decoding Emacs's org-mode documents.

The support is very basic and focuses only on the features of org mode
required by Dogen.

UML representation:

\image html dogen.org/modeling/dogen.org.svg

◉ entities                                                         :module:
  :PROPERTIES:
  :custom_id: O0
  :masd.codec.dia.comment: true
  :END:

Represents entities from Emacs' org-mode.

 ○ section                                                        :element:...
 ○ document                                                       :element:...
 ○ headline                                                       :element:
   :PROPERTIES:
   :custom_id: 2ae8829f-db42-4ab8-a2c3-627ae2b0431d
   :END:

Represents a headline in org mode.

Headlines and Sections
```

Figure 7.7: Org-mode model in org-mode notation.

This outcome was not completely left to chance. Indeed, a key principle when designing the mapping was to try to keep to org-mode's native idioms as much as possible, so that MASD models would not appear surprising to regular (*i.e.* non-MASD) org-mode users. The advantages extend beyond humans since such an approach also ensures compatibility with the tooling ecosystem. However, note that the mapping is not completely native as of yet, with further analysis work still ongoing; for example, references could make use of links to facilitate navigation between org-mode documents, but at present these are properties within the model drawer (*e.g.* `masd.codec.reference` in Figure 7.7, near the top).[24]

Native behaviour

24 Using org-roam to enable cross-document referencing appears to be a fruitful way with which to address this issue, though additional analysis still remains on how best to perform this integration.

A final note is warranted on the processing of headline titles. In the interest of pushing forward with the literate modeling agenda, titles can make use of natural language conventions such as spaces and capitilisation, greatly improving readability: *e.g.*, ″`An example title`″ instead of ″`an_example_title`″. Doing so means benefiting directly from org-mode's extensive export machinery without requiring preprocessing — such that a model generate documents in formats such as PDF, HTML and many others. Conversely, for the purposes of code generation, headlines are converted internally into valid identifiers, as a function of non-structural variability; for instance, users can decide to replace spaces by underscores, camel or pascal casing as well as other conventions popular with software engineers.

Readability and exporting

7.5 EVALUATION

Introducing org-model into the MRI was a typical application of the MASD Composite Process (*cf.* Section 5.4.2.4). Consequently we must analyse the impact of the application at two distinct levels: *application* and *meta-application*. The first level is the most obvious: there was a practical requirement to address a set of diagnosed issues and, in doing so, a significant new feature was added to the MRI. However, the exercise was also a test on how the fundamental ideas of MASD are to be applied in the real world, and thus a meta-application of the methodology. The next two sections delve into the details of these two levels.

Levels of application

7.5.1 *Application Evaluation*

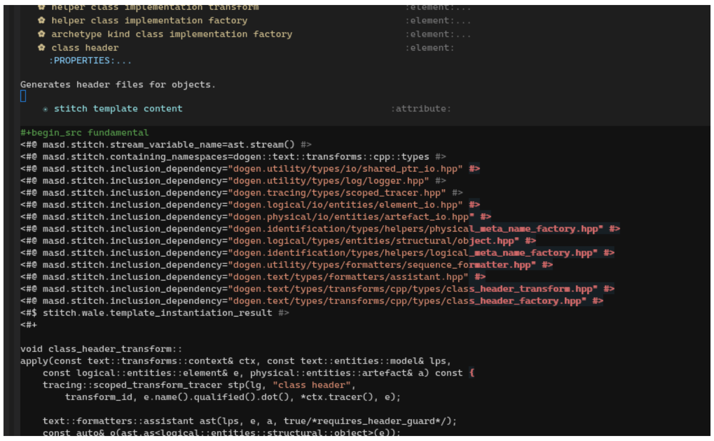
Though all models within MASD's MRI have been converted to the org-mode representation, this feature has been in production for less than a year, making it difficult to perform a thorough evaluation. Nonetheless, given the limitations previously had faced with graphical modeling, certain improvements are unequivocal:

Approach advantages

- **Improved information flow**: it is clearly easier to move information between the different representations, particularly between the agile representation and the documentation representation as we had hoped. Having all documentation in the same format greatly helps in this regard.
- **Improved tooling integration**: in the past, it was very difficult to move models backwards and forwards between versions in version control. Though Dia stored diagrams in a text representation, its XML file format is quite verbose and clearly designed for machine consumption rather than human. With org-mode, it is now trivial to diff versions of models, and to synchronise them using the exact same tools we use to manage source code
- **Better model navigation and editing**: closely related to the previous topic, it is now much easier to find and edit model elements — again, because we are making use of the same tooling as for programming. For

example, we often use Unix tools such as `grep`, `sed`, `sort` *etc.* to retrieve and manipulate information stored in models — something which was not possible in the past due to the complexity of the Dia file format; these issues are common to any XML-based file format, so XMI would be no different.

- **Text templating integration**: org-mode source code blocks allowed us to integrate the text templates used to generate M2T transforms in a very natural way. These will be discussed in more detail on Chapter 8, but the example shown on Figure 7.8 demonstrates the principle. The templates are carried within source code blocks in org-mode documents, facilitating their editing and composition.



```
   ✿ helper class implementation factory                          :element:...
   ✿ archetype kind class implementation factory                  :element:...
   ✿ class header                                                 :element:
     :PROPERTIES:...

Generates header files for objects.

    ⊛ stitch template content                                   :attribute:

#+begin_src fundamental
<#@ masd.stitch.stream_variable_name=ast.stream() #>
<#@ masd.stitch.containing_namespaces=dogen::text::transforms::cpp::types #>
<#@ masd.stitch.inclusion_dependency="dogen.utility/types/io/shared_ptr_io.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.utility/types/log/logger.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.tracing/types/scoped_tracer.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.logical/io/entities/element_io.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.physical/io/entities/artefact_io.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.identification/types/helpers/physical_meta_name_factory.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.logical/types/entities/structural/object.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.identification/types/helpers/logical_meta_name_factory.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.utility/types/formatters/sequence_formatter.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.text/types/formatters/assistant.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.text/types/transforms/cpp/types/class_header_transform.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.text/types/transforms/cpp/types/class_header_factory.hpp" #>
<#$ stitch.wale.template_instantiation_result #>
<#+

void class_header_transform::
apply(const text::transforms::context& ctx, const text::entities::model& lps,
    const logical::entities::element& e, physical::entities::artefact& a) const {
    tracing::scoped_transform_tracer stp(lg, "class header",
        transform_id, e.name().qualified().dot(), *ctx.tracer(), e);

    text::formatters::assistant ast(lps, e, a, true/*requires_header_guard*/);
```

Figure 7.8: Example text template in a org-mode model.

However, not all aspects of this transition have been positive. By far, the most significant downside of moving towards a text representation was the loss of a graphical representation for modeling. Given that UML class diagrams had been very central to modeling within MASD, this was not an acceptable trade-off, so we decided to address this shortcoming by adding a PlantUML codec to the MRI. As Figure 7.9 attests, basic support for class diagrams is already available, though their expressiveness is not yet at the same level of past Dia diagrams.

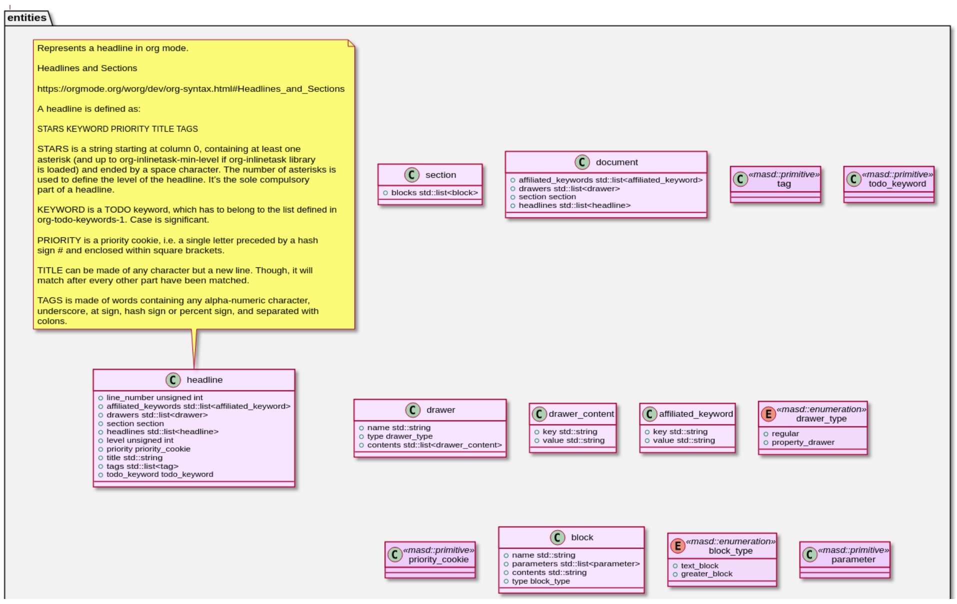
Approach disadvantages

Figure 7.6 had shown the Dia representation for a fragment of this very model. It should be apparent that the manually crafted diagram is "easier on the eye" when compared to the automatically generated PlantUML diagram, though the qualitative nature of this judgement makes it difficult to measure. Nonetheless, the easiness with which both org-mode and PlantUML codecs were added to the MRI validate the approach in creating a framework for codecs. They also make a broader statement with regards to pervasive integration, which the next section shall address.

PlantUML challenges

7.5.2 *Meta-application Evaluation*

The implementation of org-model as a new feature in the MRI is a canonical example of how we expect MASD to change and adapt to developer workflows. By reflecting on the process, the following salient characteristics were found:



Represents a headline in org mode.

Headlines and Sections

https://orgmode.org/worg/dev/org-syntax.html#Headlines_and_Sections

A headline is defined as:

STARS KEYWORD PRIORITY TITLE TAGS

STARS is a string starting at column 0, containing at least one asterisk (and up to org-inlinetask-min-level if org-inlinetask library is loaded) and ended by a space character. The number of asterisks is used to define the level of the headline. It's the sole compulsory part of a headline.

KEYWORD is a TODO keyword, which has to belong to the list defined in org-todo-keywords-1. Case is significant.

PRIORITY is a priority cookie, i.e. a single letter preceded by a hash sign # and enclosed within square brackets.

TITLE can be made of any character but a new line. Though, it will match after every other part have been matched.

TAGS is made of words containing any alpha-numeric character, underscore, at sign, hash sign or percent sign, and separated with colons.

entities
section
blocks std::list<block>
document
affiliated_keywords std::list<affiliated_keyword>
drawers std::list<drawer>
section section
headlines std::list<headline>
«masd::primitive» tag
«masd::primitive» todo_keyword
headline
line_number unsigned int
affiliated_keywords std::list<affiliated_keyword>
drawers std::list<drawer>
section section
headlines std::list<headline>
level unsigned int
priority priority_cookie
title std::string
tags std::list<tag>
todo_keyword todo_keyword
drawer
name std::string
type drawer_type
contents std::list<drawer_content>
drawer_content
key std::string
value std::string
affiliated_keyword
key std::string
value std::string
«masd::enumeration» drawer_type
regular
property_drawer
«masd::primitive» priority_cookie
block
name std::string
parameters std::list<parameter>
contents std::string
type block_type
«masd::enumeration» block_type
text_block
greater_block
«masd::primitive» parameter

Figure 7.9: PlantUML representation of org-mode model.

- Friction in the modeling process was detected. Parts of the development process had changed specifically to accommodate modeling, and demanded the use of tools that were not needed previously (*i.e.*, Dia).

- Over a period of time, analysis was carried out on how to best perform the same modeling activities within the existing tooling; the aim was to discover a more efficient way of integrating the new processes with the existing development process, lowering the overall friction.

- Org-mode was chosen as the technology with the best fit with the current development environment. Physical modeling was applied to org-mode artefacts, even though their role was not to augment the PMM. This was not a use case we had anticipated for physical modeling, but it proved extremely useful.

- Additional use cases that had not been anticipated originally in MASD were made available by the new technological choice, with literate modeling being an important addition, as well as the many tools that make up the org-mode ecosystem.

Our conclusion is that the processes defined by MASD worked precisely as intended, and their application resulted in an overall improvement of the MRI.

8 MASD REFERENCE IMPLEMENTATION

High-level models are quite different from programs in conventional programming languages. They abstract from most of the detail that a programming language exhibits. Once you want to generate real code, all this detail has to be filled in. This makes code generation from those models a difficult task. Moreover, many decisions in this process are similar for different target languages, but it is hard to make use of these commonalities.

— Piefel and Neumann (Piefel and Neumann, 2006)

It's hard to overestimate the importance of the *MASD Reference Implementation* (*MRI*) to MASD; it is a key component of the methodology, playing three distinct roles. First, due to our reliance on dogfooding (*cf.* Section 5.4.2.4), all ideas and features must be battle-tested in the MRI before they can be incorporated and exposed to end users; in this guise, the MRI acts as their litmus test.[1] Secondly, the MRI is the *de facto* standard for anything not covered in the present document (*cf.* Section 5.2.3.5). This includes core elements such as the MASD UML profile, non-structural variability definitions, the menu of supported codecs and much more. Finally, and perhaps most significantly, the MRI supplies the tooling needed by end users in order to apply MASD to their own software projects.[2]

Multifaceted roles

The MRI is a software product line presently composed of three individual products. This chapter will present an overview of each of these, as follows. Section 8.1 covers Dogen, the code generator, and contains the bulk of the chapter's material. Section 8.2 discusses the two reference products for each of the supported TS's (C++ and C#). The chapter ends with a discussion of the lessons learned with the MRI (Section 8.3). Let us then start with the software project at the root of it all.

Chapter overview

8.1 DOGEN

Dogen[3], the *domain generator*, is a FOSS project implementing a code generator based on MDE principles and is deeply interwoven with MASD, as this section

1 These tests are both literal — in that the MRI contains an automated test suite to validate its features — and conceptual, in that the MRI is expected to have use cases for all relevant aspects of MASD as a methodology. The MRI is *primus inter pares* of all software products generated using the MRI.

2 Note that MASD is not against using third party implementations, but at present these do not exist; and when they do, it is likely they will lag behind the MRI. Thus, this option is not considered in the present document.

3 https://github.com/MASD-Project/dogen

will explain. The scene will be first set by discussing the historical context and the motivation for the project (Section 8.1.1). Section 8.1.2 then discusses the additional requirements the tool had to satisfy, followed by Section 8.1.3 which elaborates on the SDM used to deliver those requirements. Section 8.1.4 provides a brief summary of its implementation details, including its architecture, and relates it back to MASD's own domain architecture (*cf.* Chapter 6). Section 8.1.6 concludes Dogen's synopsis by taking on end user concerns such as tool usage and application, and supplies a quick tour of the main features of the command-line tool.

Section overview

8.1.1 *Historical Context*

Releases

Dogen was started in 2011 by the author of the present document, who acts as both its maintainer and main contributor. Dogen has been in continuous development since inception, with a frequent if somewhat irregular release cadence, totalling 130 releases over the decade-long span — 110 of which are publicly available. Figure 5.11 shows the distribution of releases per year, starting in 2012, which is the year the project was first made available in GitHub.[4] Its latest release is v1.0.30, published at the start of January 2021.[5] Focus then shifted towards completing the present document, meaning no further releases have been made since.

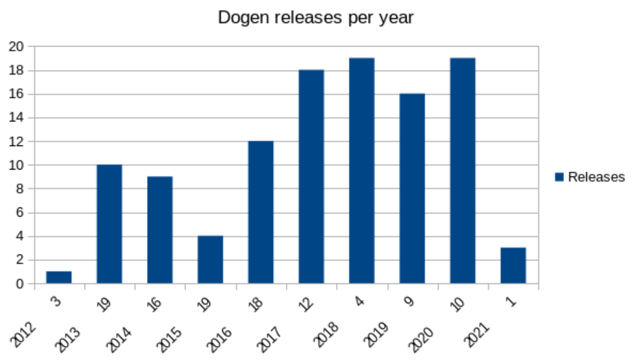


Figure 8.1: Dogen releases per year.

The current hiatus in development is not uncommon in Dogen's history, as periods dominated by software engineering have been intentionally interleaved with periods dedicated mainly to theoretical work. Looking back on over a decade of engineering, one can divide it into the following phases:

- **The second-system effect[6] phase (2011 to 2013)**: Though Dogen was created from scratch, it is conceptually the second iteration of the experiences narrated in (Craveiro, 2021c). At inception, a naive belief was held that most of the previous problems were rooted in inexperienced

4 For completeness, the first public release was v0.20.588. Prior to this release, there had been 19 releases which were not made available to the outside world.

5 https://github.com/MASD-Project/dogen/releases/tag/v1.0.30

6 Brooks describes the second-system effect vividly: "This second [system] is the most dangerous system a man (*sic.*) ever designs. [...] The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one. The result, as Ovid says, is a 'big pile'." (Brooks, 1974) (p. 55)

engineering decisions; and that these could be overcome, quickly, via a sounder software engineering approach. To a small extent this conjecture was proved correct, as improvements in the testing framework, parsing and other key areas resulted on an enhanced code generator. However, more issues were created than resolved overall and the architectural complexity increased significantly. By the end of this phase, the magnitude of the original ask was finally understood. As a result, the scope of the endeavour was reduced to a manageable size, and scope narrowing became enshrined in the approach's core values (P-1).

- **The experimental phase (2013 to 2015)**: The boundaries may have been set but the internal architecture remained unclear, so the next two years were spent on architectural experimentation. Whilst neither inputs or outputs changed a great deal, the architecture was redesigned multiple times, and all of its components were renamed several times over. As an example, a TS-level model was introduced, using constructs from the supported programming languages directly. Predictably, the approach resulted in failure as the resulting models and associated transforms were too complex. Several other attempts of this ilk were made, and though some of the ideas were kept — most notable of which is *stitch*, a new text templating language (*cf.* Section 8.1.4) — no overall breakthroughs were achieved.

Phases of development

- **The research phase (2015 to 2020)**: This phase begun with the realisation that a strong theoretical foundation was necessary. During this period we familiarised ourselves with the literature on code generation, with a particular focus on MDE, and conducted a formal programme of research that produced the present manuscript as its final outcome. The theoretical foundations for Dogen begun as a series of concepts which were iterated upon for just over 70 releases, until the emergence and consolidation of the domain architecture (*cf.* Chapter 6). This phase included the separation of MASD, the methodology, from Dogen, the tool.

- **The productionisation[7] phase (2021 onwards)**: The current objective for Dogen is to add a small number of missing features needed to fully support product line generation, at which point it can be said to have passed the MRI's fitness function (*cf.* Section 5.4.2.4). Appropriately, the completion of these features will be marked by the release of Dogen v2.0.[8] Subsequent to this release, we will begin to implement some of the many ideas for new features which have been captured over the product's lifetime.

An important aspect in our drive towards the productionisation of Dogen has been the consolidation of the requirements for the tool. Further to the require-

7 Wikipedia tells us that: "Productionisation [...] is the process of turning a prototype of a design into a version that can be more easily mass-produced. It is mostly a necessary step in the development of any product, since it is rare that the initial design is free from flaws or construction methods which make it difficult or more expensive to manufacture." (Productionisation, 2021)

8 Originally, we had intended to release v2.0 before publishing the present document. However, there were inevitable delays in coding, which is to be expected given the difficulty in estimating outstanding effort. Since all of the conceptual work had been completed, we opted for finalising the document first and then resuming the programming activities at a later stage.

ments laid out in Chapter 4 — mainly driven by theoretical considerations — more practical concerns have been added, as described next.

8.1.2 *Requirements*

Dogen introduces four additional non-functional requirements which significantly constrain the set of choices for its implementation; of these, three are refinements of what has already been outlined in Chapter 4. They are as follows:

Non-functional requirements

- **Performance requirements**. Generating all its models must not take longer than 5 seconds, and running the entire test suite — including regenerating all of the C++ and C# reference implementation models (*cf.* Section 8.2) — must not take over 10 seconds. Whilst somewhat arbitrary, these numbers were chosen as an acceptable upper bound on wait time during development. Higher values would result in a degradation of the edit-compile-run cycle, negatively impacting developer experience.[9]

- **Dependency requirements** (furthering of Requirement 15). In order to facilitate integration to development environments, Dogen must be as self-contained as possible. This means it cannot necessitate the installation of run times such as JVM or the CLR, as these would act as a further barrier to entry to developers not using those technologies. Thus, only technologies which produce native binaries can be used.

- **Cross-platform requirements** (furthering of Requirement 15). Dogen must support all of the main platforms used by developers. These include Windows, Linux and Mac OS X (OSX). The code base must support compilation across all of these platforms, implying that the generated code must also have cross-platform support.

- **Integration requirements** (furthering of Requirement 15). Finally, Dogen must supply a library with access all of its functionality; the library can be used as a basis to extend existing programming environments. In order to make the library compatible with any of the myriad of technologies used to engineer development environments, the interface should be available on a low-level programming language such as C. This is because binding C interfaces into any of the modern languages via tooling such as SWIG (Craveiro, 2021d) (Section 5.3) is a straightforward exercise.

These non-functional requirements stem in no small part from our own experiences in developing MDE tooling (Craveiro, 2021b), as well as our reading of the adoption literature (*cf.* Chapter 3), where performance and disregard for developer workflows are often cited as pitfalls. Furthermore, the long list of requirements, both functional and non-functional, together with the many

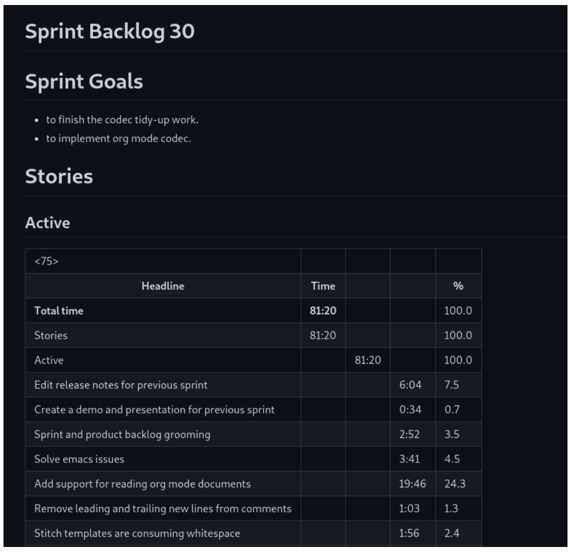
9 Though by no means fans of the edit-compile-run cycle, Lemma and Lanza do describe it rather aptly: "Most mainstream programming languages [...] are based on the traditional edit-compile-run cycle. This approach allows developers to recognize clear boundaries between the different phases to focus on one activity at a time: first write the code, then compile it, and finally observe and test the system at runtime." (Lemma and Lanza, 2013)

phases that the development has gone through (*cf.* Section 8.1.1) are reflective of a difficult search for a solution over a vast problem space. Our choice of SDM proved decisive in leading us through such a long and open-ended search, as the next section will explain.

8.1.3 *Software Development Methodology*

Sprints, sprint backlog

A cornerstone of Dogen has been its extensive use of agile, in itself a demonstration of how MASD integrates with typical SDMs.[10] The practice of sprints has been adopted from inception, with a sprint elapsing around 80-developer hours of effort and culminating with a release. Each sprint has a sprint backlog checked in to version control, documenting all development activity undertaken in the form of stories. Figure 8.2 captures a fragment of the sprint backlog, showing the time-keeping aspect. Note that whilst publicly accessible, the sprint backlog is meant mainly for internal consumption.



Sprint Backlog 30

Sprint Goals

- to finish the codec tidy-up work.
- to implement org mode codec.

Stories

Active

<75>				
Headline	**Time**			**%**
Total time	**81:20**			100.0
Stories	81:20			100.0
Active		81:20		100.0
Edit release notes for previous sprint			6:04	7.5
Create a demo and presentation for previous sprint			0:34	0.7
Sprint and product backlog grooming			2:52	3.5
Solve emacs issues			3:41	4.5
Add support for reading org mode documents			19:46	24.3
Remove leading and trailing new lines from comments			1:03	1.3
Stitch templates are consuming whitespace			1:56	2.4

Figure 8.2: Fragment of sprint backlog for Dogen's 130th sprint.

Retrospectives, subsidiary material

The second salient aspect of our agile approach has been the use of retrospectives.[11] Lacking traditional forms of interaction and validation that come naturally with being part of a wider engineering team, we turned instead towards publishing online content in order to emulate this feedback loop. Unlike the sprint backlog, the material produced in this context is meant for external consumption, and so provides the additional background required by a lay audience. As part of this outreach effort, detailed release notes have been authored from 2016 onwards, made available in the project's repository.[12] The release notes provide user-friendly descriptions of the main stories carried out, discussing trade-offs and supplying a rationale for the changes, as well

10 For more details on SDM's, including agile, as well as their integration with MDE, please consult Chapter 5 of (Craveiro, 2021c).

11 The term is used in the sense put forward by Derby *et al.*: "[retrospectives are] a special meeting where the team gathers after completing an increment of work to inspect and adapt their methods and team work. Retrospectives enable whole-team learning, act as catalysts for change, and generate action." (Derby, Larsen, and Schwaber, 2006)

12 https://github.com/MASD-Project/dogen/releases

as providing links to a video demonstrating the changes. Figure 8.3 is taken from v1.0.30's release notes, and shows a graphical summary of development effort across stories, measured as story duration as a percentage of total sprint duration.[13] Release notes and other online material has helped enormously as Dogen progressed through the development phases (*cf.* Section 8.1.1).

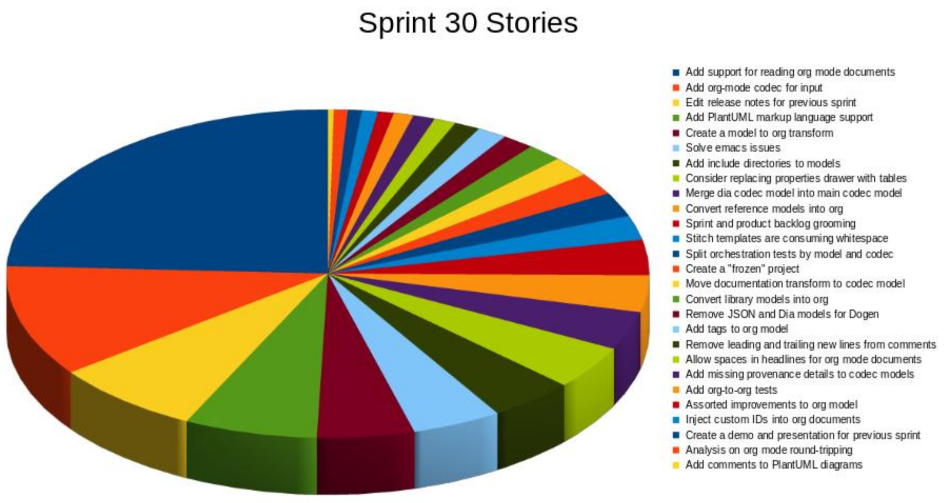


Figure 8.3: User-facing description of story effort for Sprint 30.

Product backlog

The final but by no means less significant component of our agile approach has been the product backlog. All ideas, thoughts and notes pertaining to Dogen which cannot be immediately acted upon are kept in the product backlog. At present count, it contains just over 800 stories and proto-stories awaiting further attention. The stories in the product backlog are loosely organised into buckets, capturing the notion that some relate to present work whereas others are of a more visionary (or aspirational) nature. Each sprint allocates resources towards grooming the product backlog, thus ensuring its relevance: stories that are no longer required are pruned, duplicate stories are merged and so on, keeping the information up-to-date.

Story workflow

Sprint and product backlogs are connected via the story workflow. Planning for a new sprint begins by moving relevant stories from the product backlog into the sprint backlog. Some stories, such as for example org-mode support (*cf.* Chapter 7), can remain on the product backlog for several years, accumulating more details until they are selected for implementation. In this manner, the product backlog has been vital in organising our search across the problem space, acting as a repository of accumulated knowledge, which reaches its actionable stage in the sprint backlog.

13 The interested reader is directed towards these documents in order to understand the historical context in greater detail. Where available, the release notes also link to associated audiovisual material, produced within the scope of each release. Moreover, all audiovisual material can be accessed via the author's YouTube channel (Marco Craveiro, 2021). Finally, assorted blog posts have also been published in the author's blog — *e.g.*, "The Refactoring Quagmire" (Marco Craveiro, 2018) — which narrate significant episodes during the before-mentioned development phases.

It is a fair assessment to state that Dogen would not have reached its present state were it not for our choice of SDM. Its ability to break down an open-ended problem into a series of small, actionable stories, has propelled us forward, even when the way ahead was unclear. Whilst each sprint delivered only a sliver of change, when aggregated over a ten-year span, the iterative and incremental approach conducted us to the comprehensive architecture we have today — and which the next section will describe, taking us towards implementation-level concerns.

Agile's significance

8.1.4 *Architecture*

Dogen is written entirely in the C++ programming language, chosen specifically to meet our non-functional requirements, particularly with regards to performance and dependencies (*cf.* Section 8.1.2). The decision had the unfortunate side-effect of closing the doors on a plethora of MDE tooling, platforms and libraries — such as EMF and T4 — as they are often written in Java or C#.[14] This helps explain Dogen's size, weighing in at just under 150 thousand LOC, as measured by the `sloccount` tool.[15]

Implementation language

Figure 8.4: Collage of all Dogen models in a UML representation.

On the other hand, around 50% of the total line count is generated by Dogen itself, and all components of Dogen are modeled as Dogen models. Figure 8.4

Generated code

14 Lack of access to MDE tooling in C++ is by no means a novel problem. Jäger *et al.* had already encountered this very difficulty in 2016 (Jäger et al., 2016), as had González *et al.* before, six years prior (González, Ruiz, and Perez, 2010). Though differing in the details, both opted for a native C++ port of Ecore, the EMF metamodel. After analysing both implementations, we decided against using either of them due to their complexity (largely a byproduct of Ecore rather than the projects themselves), fears about vendor lock-in (both projects have a small user and developer base) and, most significantly, the fact that Dogen did not require many of Ecore's features such as runtime introspection, for which there is likely a runtime penalty associated. However, we do intend to support Ecore as a codec, allowing users to create their models using EMF and generating them via Dogen. Thus, as future work, we will likely integrate with one of these libraries, but relegating their use to only *input* (and possibly *output*).

15 https://dwheeler.com/sloccount/

shows a collage of all seventeen Dogen models in Dia's UML representation, prior to their move to org-model format (*cf.* Chapter 7). Though perhaps not particularly helpful with regards to detail, its bird's eye view does portray adequately the size and complexity of the application, as well as demonstrating the use of colour in our UML diagrams (*cf.* Section 6.1.5.3). Figure 8.5 zooms in on the largest of these components, `dogen.logical`, implementing the LMM (*cf.* Section 6.2).

Each component in Figure 8.4 represents a distinct sub-system within Dogen, with a well-defined set of responsibilities. What follows is a summary of the components, with a reference back to MASD's domain architecture (*cf.* Chapter 6) where applicable.

- `dogen.codec`: (*cf.* Figure 6.35) contains the codec framework and its associated transform chains. It is responsible for the extraction of modeling information from foreign TS, and converting them into the simplified codec representation. It collaborates with external platforms (*e.g.* for JSON) or internal components (*e.g.* `dogen.dia`, `dogen.org`) to read the external representations. For certain limited cases it also provides extraction support — at present org-mode and PlantUML.

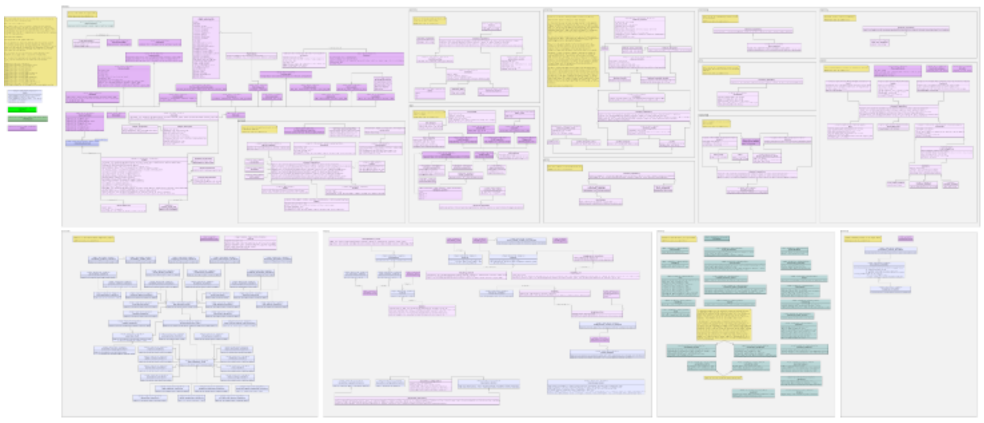


Figure 8.5: UML representation of Dogen's logical model.

- `dogen.dia`: internal implementation of the Dia object model, including a parser that uses an XML library to read Dia diagram files. The library is asymmetric — that is, it can only read diagrams.

- `dogen.org`: internal implementation of an org-mode parser. Contains all of the domain objects representing org-mode entities, as well as transforms that convert from and to this representation.

- `dogen.identification`: defines all identifiers across LPS dimensions (*i.e*, logical, physical and variability dimensions), as well as identifiers for codecs, used in projections across these spaces (*cf.* Section 6.2.2).

- `dogen.logical`: implements the logical dimension of the LPS (*cf.* Section 6.2), as well as supplying all the transformations required to process logical elements (bottom left in Figure 8.5).

- `dogen.physical`: implements the physical dimension of the LPS (*cf.* Section 6.1), and contains all the transformations required to generate files in the filesystem. Also contains an implementation of the PMM, encoding with it at compile time the geometry of physical space. Conceptually, this can be thought of as a reflection layer on the physical space, allowing us to query it at run time — *e.g.*, "list all facets for the C++ TS", *etc.*

- `dogen.variability`: implements the variability dimension of the LPS — *i.e.*, the VMM (*cf.* Section 6.3) — and contains all of the transformations required to extract configurations from any of the supported consumer models (*i.e.* `dogen.logical`, `dogen.physical` and `dogen.codec`).

- `dogen.profiles`: contains the configuration for the Dogen product, instantiating the VMM and defining the configuration language for Dogen. Figure 6.40 shows a fragment of this model.

- `dogen.text`: projection framework that takes logical model elements and generates their physical representation. These are implemented as M2T transforms, defined using the internal text templating languages discussed next.

- `dogen.templating`: provides the two templating engines used throughout Dogen. The first one is a trivial implementation of the mustache logic-less templates[16] and is expected to be replaced in the near future by a platform library such as `mstch`.[17] The second templating engine is called *stitch*, and it is used to create M2T transforms (*cf.* Section 8.1.5).

- `dogen.tracing`: provides transform tracing infrastructure. When enabled, the tracing sub-system creates trace dumps of model state before and after transform execution as text files in JSON format. These files can then be diffed using command line utilities such as `diff` or further processed with tools like JQ[18], which specialises in JSON processing.

- `dogen.relational`: provides a RDBMS backend for the tracing subsystem, allowing trace data to be stored in database tables. However, at present, the information is stored as JSON documents, making querying difficult. Further work is required in order to fulfil the initial promise of this approach, making deeper use of the relational model. It will also serve as an improved test of the ORM support in Dogen.

- `dogen.orchestration`: orchestration engine for Dogen that creates the top-level transformation chain, calling out to each sub-system as required. It is responsible for creating the pipeline that converts from a codec representation, to a logical model representation and finally to the physical representation. Figure 8.6 shows a fragment of the transforms defined in the orchestration model.

- `dogen.utility`: assorted utility functions such as logging, XML processing, testing helpers and other miscellaneous functionality.

16 https://mustache.github.io/
17 https://github.com/no1msd/mstch
18 https://stedolan.github.io/jq/

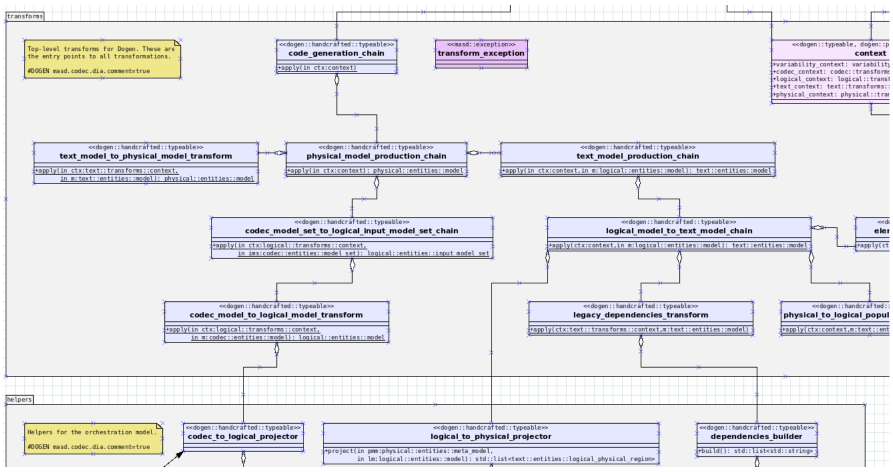

transforms
Top-level transforms for Dogen. These are the entry points to all transformations.
#DOGEN masd.codec.dia.comment=true
<<dogen::handcrafted::typeable>>
code_generation_chain
+apply(in ctx:context)
<<masd::exception>>
transform_exception
<<dogen::typeable, dogen::p
context
+variability_context: variability
+codec_context: codec::transforms
+logical_context: logical::transf
+text_context: text::transforms::
+physical_context: physical::tran
<<dogen::handcrafted::typeable>>
text_model_to_physical_model_transform
+apply(in ctx:text::transforms::context,
in m:text::entities::model): physical::entities::model
<<dogen::handcrafted::typeable>>
physical_model_production_chain
+apply(in ctx:context): physical::entities::model
<<dogen::handcrafted::typeable>>
text_model_production_chain
+apply(in ctx:context,in m:logical::entities::model): text::entities::model
<<dogen::handcrafted::typeable>>
codec_model_set_to_logical_input_model_set_chain
+apply(in ctx:logical::transforms::context,
in ims:codec::entities::model_set): logical::entities::input_model_set
<<dogen::handcrafted::typeable>>
logical_model_to_text_model_chain
+apply(ctx:context,in m:logical::entities::model): text::entities::model
ele
+apply(ct
<<dogen::handcrafted::typeable>>
codec_model_to_logical_model_transform
+apply(in ctx:logical::transforms::context,
in m:codec::entities::model): logical::entities::model
<<dogen::handcrafted::typeable>>
legacy_dependencies_transform
+apply(ctx:text::transforms::context,m:text::entities::model)
<<dogen::handcrafted::
physical_to_logical_popul
+apply(ctx:context,m:text::ent
helpers
Helpers for the orchestration model.
#DOGEN masd.codec.dia.comment=true
<<dogen::handcrafted::typeable>>
codec_to_logical_projector
<<dogen::handcrafted::typeable>>
logical_to_physical_projector
+project(in pmm:physical::entities::meta_model,
in lm:logical::entities::model): std::list<text::entities::logical_physical_region>
<<dogen::handcrafted::typeable>>
dependencies_builder
+build(): std::list<std::string>

Figure 8.6: Fragment of Dogen's orchestration model with transforms.

- `dogen`: the API model. Contains the top-level entry point to Dogen. This component will be exposed as a library via SWIG using a language such as C, to enable bindings for a number of programming languages.

- `dogen.cli`: implements the command-line driver for Dogen. This is the application entry point from a coding perspective, if using Dogen as a stand-alone tool.

Most of these subsystems have complex implementation details which cannot be adequately covered in the present manuscript. We shall, however, look into one such subsystem because it offers some insight on the overall approach. This is the templating language used in `dogen.text`.

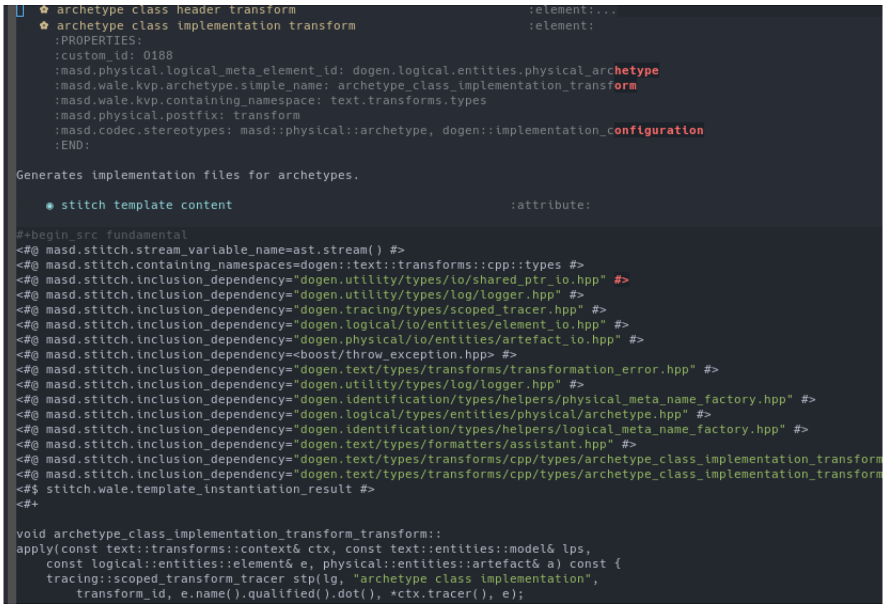
8.1.5 *Stitch*

Relationship with T4

As previously mentioned (*cf.* Section 7.5.1), Dogen has support for a text templating language which can be embedded into org-mode source code blocks. This language was implemented specifically for Dogen, and now binds closely to its use cases. Its syntax is inspired by Microsoft's T4, a text templating engine popular in industry (*e.g.* (Craveiro, 2021c)). Figure 8.7 depicts a small fragment of a larger text template defined in the `dogen.text` model.

Increased integration

As explained, stitch was created because T4 was unavailable for our development stack. In its initial incarnation, stitch was envisioned as a stand-alone tool that pre-processed a set of text templates and generated compilable C++ code, acting like a cartridge in a cartridge pipeline (*cf.* Section 6.1.6). However, as the problem domain became better understood over the years, stitch was wired ever more closely to the domain architecture, to the point where its templates are now modeled in the LMM and the external templating tool has been removed altogether. The process leading to the current state of affairs is instructive, as it has implications for cartridges in general.



```
  ✿ archetype class header transform                               :element:...
  ✿ archetype class implementation transform                       :element:
     :PROPERTIES:
     :custom_id: 0188
     :masd.physical.logical_meta_element_id: dogen.logical.entities.physical_archetype
     :masd.wale.kvp.archetype.simple_name: archetype_class_implementation_transform
     :masd.wale.kvp.containing_namespace: text.transforms.types
     :masd.physical.postfix: transform
     :masd.codec.stereotypes: masd::physical::archetype, dogen::implementation_configuration
     :END:

Generates implementation files for archetypes.

    ● stitch template content                                    :attribute:

#+begin_src fundamental
<#@ masd.stitch.stream_variable_name=ast.stream() #>
<#@ masd.stitch.containing_namespaces=dogen::text::transforms::cpp::types #>
<#@ masd.stitch.inclusion_dependency="dogen.utility/types/io/shared_ptr_io.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.utility/types/log/logger.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.tracing/types/scoped_tracer.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.logical/io/entities/element_io.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.physical/io/entities/artefact_io.hpp" #>
<#@ masd.stitch.inclusion_dependency=<boost/throw_exception.hpp> #>
<#@ masd.stitch.inclusion_dependency="dogen.text/types/transforms/transformation_error.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.utility/types/log/logger.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.identification/types/helpers/physical_meta_name_factory.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.logical/types/entities/physical/archetype.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.identification/types/helpers/logical_meta_name_factory.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.text/types/formatters/assistant.hpp" #>
<#@ masd.stitch.inclusion_dependency="dogen.text/types/transforms/cpp/types/archetype_class_implementation_transform
<#@ masd.stitch.inclusion_dependency="dogen.text/types/transforms/cpp/types/archetype_class_implementation_transform
<#$ stitch.wale.template_instantiation_result #>
<#+

void archetype_class_implementation_transform_transform::
apply(const text::transforms::context& ctx, const text::entities::model& lps,
    const logical::entities::element& e, physical::entities::artefact& a) const {
    tracing::scoped_transform_tracer stp(lg, "archetype class implementation",
        transform_id, e.name().qualified().dot(), *ctx.tracer(), e);
```

Figure 8.7: Fragment of a stitch text template in the text model.

Applying the physical modeling process (*cf.* Section 6.1.2) to stitch's text templates revealed these contained a great deal of SRPP's. In addition, the analysis also demonstrated templates could be simplified in two significant ways:

Template simplification

- **Reducing the variability surface**: given that their purpose is solely to create M2T transforms for Dogen's transform framework, they need not cater for general use cases like T4 does. The generated code can thus be hard-coded to fit precisely the transform framework, minimising the use of non-structural variability to a few select cases. Doing so reduces considerably the size of each template.
- **Reusing metamodel information**: adding stitch to the LMM gives access to a wealth of information about modeled types, such as their relationships (*cf.* Section 6.1.4.3). This removes the need for handcrafting relations in the template, as well as any other parameter that can be inferred from the metamodel and its instances such as the boilerplate, *etc.*

Cartridges and trade-offs

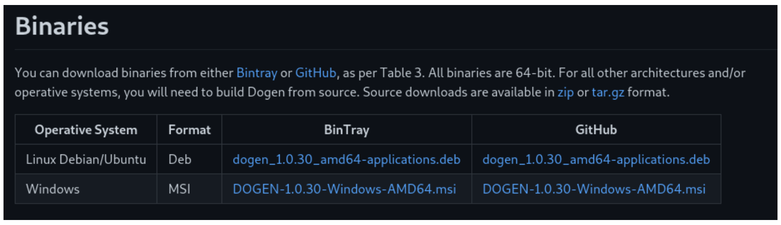
In summary, though in an initial stage stitch was deemed to be outside MASD's remit — as stipulated by its core values, and P-1 in particular — over time it was shown to be an integral part of the approach. This is a general trade-off that will be faced with all cartridges: there are obvious benefits of keeping their functionality external to Dogen, reducing the tool's footprint. On the other hand, tighter integration has many benefits, such as improving the overall user experience.[19] And it is to the user experience that we shall turn to next.

19 For example, end users need to install and configure the tooling for external cartridges. At present this is a significant source of accidental complexity for some cartridges such as ODB, due to its complex installation procedures.

8.1.6 *Basic Usage*

Packages

The present section gives a brief overview on the command line tool supplied with Dogen, `dogen.cli`. In order to it, you must first install the Dogen package. Packages for supported operative systems are available for download from the release notes, at the bottom, as shown in Figure 8.8.[20] GitHub is the recommended provider, as BinTray will be decommissioned in the near future.



Binaries

You can download binaries from either Bintray or GitHub, as per Table 3. All binaries are 64-bit. For all other architectures and/or operative systems, you will need to build Dogen from source. Source downloads are available in zip or tar.gz format.

Operative System	Format	BinTray	GitHub
Linux Debian/Ubuntu	Deb	dogen_1.0.30_amd64-applications.deb	dogen_1.0.30_amd64-applications.deb
Windows	MSI	DOGEN-1.0.30-Windows-AMD64.msi	DOGEN-1.0.30-Windows-AMD64.msi

Figure 8.8: Binary downloads section in Dogen's release notes.

The installation process is as per native packages in each operative system; for example, for Debian GNU/Linux, the installation can be performed using the root user, as follows:

```
# dpkg -i dogen_1.0.30_amd64-applications.deb
Selecting previously unselected package dogen-applications.
(Reading database ... 625869 files and directories currently installed.)
Preparing to unpack dogen_1.0.30_amd64-applications.deb ...
Unpacking dogen-applications (1.0.30) ...
Setting up dogen-applications (1.0.30) ...
```

Listing 8.1: Installation of Dogen on Debian GNU/Linux.

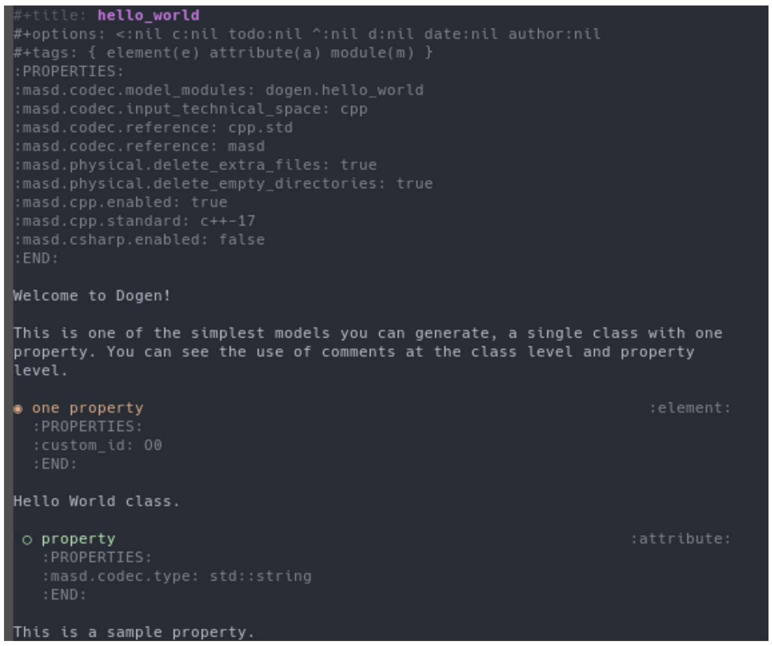
Once installed, the application should be ready to use by regular users. You can validate your installation by running the command `dogen.cli` with `--version` or `--help`. The below listing shows the truncated output of the `--help` command.

```
$ dogen.cli --help
Dogen is a Model Driven Engineering tool that processes models encoded in [...]
Dogen is created by the MASD project.
dogen.cli uses a command-based interface: <command> <options>.
See below for a list of valid commands.

Global options:

General:
  -h [ --help ]                 Display usage and exit.
  -v [ --version ]              Output version information and exit.

Output:
  --byproduct-directory         Directory in which to place all of the
                                byproducts of the run such as log files,
                                traces, etc.

Logging:
  -e [ --log-enabled ]          Generate a log file.
  -l [ --log-level ] arg        What level to use for logging. Valid values:
                                trace, debug, info, warn, error. Defaults to
                                info.
[...]
Commands:

   generate       Generates source code from input models.
   convert        Converts a model from one codec to another.
   dumpspecs      Dumps all specs for Dogen.

For command specific options, type <command> --help.
```

Listing 8.2: Dogen's help command.

20 Due to issues with our CI/CD provider, Mac OS X builds have been discontinued. This is an issue which will be addressed in the near future. For now only Windows and Linux packages are available, as shown in the screenshot.

A model is required in order to drive the tool. A trivial model is supplied with Dogen for this purpose, called `hello_world.org`. Figure 8.9 shows its org-mode representation which, though simple, still deserves a closer inspection. First, there are several of configuration options at the model level, as follows:



```
#+title: hello_world
#+options: <:nil c:nil todo:nil ^:nil d:nil date:nil author:nil
#+tags: { element(e) attribute(a) module(m) }
:PROPERTIES:
:masd.codec.model_modules: dogen.hello_world
:masd.codec.input_technical_space: cpp
:masd.codec.reference: cpp.std
:masd.codec.reference: masd
:masd.physical.delete_extra_files: true
:masd.physical.delete_empty_directories: true
:masd.cpp.enabled: true
:masd.cpp.standard: c++-17
:masd.csharp.enabled: false
:END:

Welcome to Dogen!

This is one of the simplest models you can generate, a single class with one
property. You can see the use of comments at the class level and property
level.

* one property                                                    :element:
  :PROPERTIES:
  :custom_id: O0
  :END:

Hello World class.

 o property                                                     :attribute:
   :PROPERTIES:
   :masd.codec.type: std::string
   :END:

This is a sample property.
```

Figure 8.9: Example Hello World model in org-model notation.

- the model targets the C++ input TS (*e.g.*, `input_technical_space`). Amongst other things, this enables C++ notation for specifying types, such as `std::string`. When targeting the C# TS, C# notation must be used instead (*e.g.* `System.DateTime`).

- the model references two additional models: `cpp.std` and `masd`. The first is a PDM that gives access to C++ Standard Library types such as `std::string`. The second model contains MASD infrastructural types, providing support for decorative elements such as modelines, licences, *etc.* (*cf.* Section 6.1.4.2).

- both the deletion of extra files as well as of empty directories have been disabled. Doing so delegates the management of artefacts to the user. As we move up the generation levels towards product family generation (Level 4), all of the filesystem management features should be enabled, allowing Dogen manage all artefacts in the filesystem for the user (*cf.* Section 5.4.2.3).

Model configuration

- the code generator will output C++ code (*i.e.* `masd.cpp.enabled` is set to `true`) but it will not output C# code (*i.e.* `masd.csharp.enabled` is set to `false`). The version used for C++ is 17.

The model then defines a single element which, via defaulting, results in the instantiation of a `masd::object` LMM entity, containing a single property called `one property`. To help visualisation, Figure 8.10 depicts the same model in UML notation as created by PlantUML, from a Dogen-generated source. Documentation is used both at the model level, the element level and the

PlantUML representation

property level, some of which visible in the PlantUML representation as UML notes.

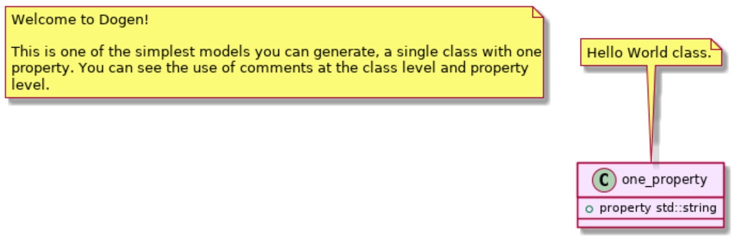
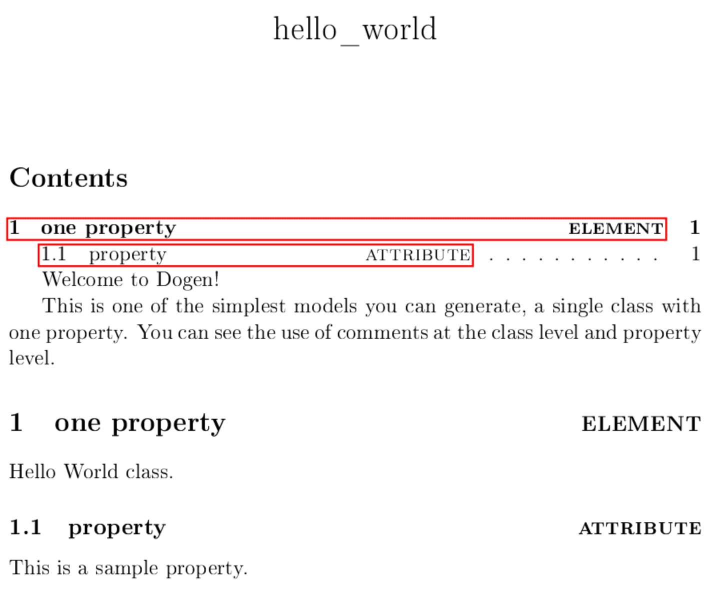

Welcome to Dogen!
This is one of the simplest models you can generate, a single class with one property. You can see the use of comments at the class level and property level.
Hello World class.
one_property
property std::string

Figure 8.10: Example Hello World model in UML notation.

PDF representation

It is also useful to visualise the literate modeling view. Figure 8.11 does so by displaying the PDF output of the model as generated by org-mode. Note that, for the purpose of this simplistic example, we rely on org-mode's default configuration, resulting on a less visually appealing document; for example, the red boxes representing hyperlinks to document sections can be configured to use a more idiomatic link notation. In addition, as explained in Chapter 7, spaces in headlines are allowed by Dogen to facilitate literate modeling; internally, headline titles are normalised to valid identifiers according to a selected scheme, involving for example converting spaces to underscores.



hello_world

Contents

1 one property ELEMENT 1
1.1 property ATTRIBUTE 1

Welcome to Dogen!

This is one of the simplest models you can generate, a single class with one property. You can see the use of comments at the class level and property level.

1 one property ELEMENT

Hello World class.

1.1 property ATTRIBUTE

This is a sample property.

Figure 8.11: Example Hello World model as a PDF document.

Source code representation

Calling `dogen.cli` with the `generate` command produces source code for this model, as per listing below. Full logging at log level `trace` is also enabled in the example, which is the highest. It is a useful setting for troubleshooting — as are other options such as transform tracing — but these are not to be used unless required, for they have a significant impact on execution speed. After running the command, all byproducts such as logs and trace files are stored under the directory `dogen.byproducts`, whereas generated source code is saved in the component directory `dogen.hello_world`.

```
$ dogen.cli generate -t hello_world.org --log-enabled --log-level trace
$ ls -l
total 9
drwxr-xr-x   3 marco        marco     4096 2021-09-01 17:10 dogen.byproducts
```

```
drwxr-xr-x   5 marco       marco    4096 2021-09-01 17:10 dogen.hello_world
-rw-r--r--   1 marco       marco     940 2021-09-01 17:10 hello_world.org
```

Listing 8.3: Generate source code for Hello World model.

As we used the default directory structure and naming, and since all facets in the C++ region of physical space are enabled, the directory tree generated in the filesystem is a simple reflection of the geometry of the PMM (*cf.* Section 6.1.6) in this region:

```
$ tree --charset nwildner
.
|-- dogen.byproducts
|   `-- cli.generate.hello_world.org
|       `-- cli.generate.hello_world.org.log
|-- dogen.hello_world
|   |-- generated_tests
|   |   `-- one_property_tests.cpp
|   |-- include
|   |   `-- dogen.hello_world
|   |       |-- hash
|   |       |   `-- one_property_hash.hpp
|   |       |-- io
|   |       |   `-- one_property_io.hpp
|   |       |-- odb
|   |       |   `-- one_property_pragmas.hpp
|   |       |-- serialization
|   |       |   |-- one_property_fwd_ser.hpp
|   |       |   `-- one_property_ser.hpp
|   |       |-- test_data
|   |       |   `-- one_property_td.hpp
|   |       `-- types
|   |           |-- hello_world.hpp
|   |           |-- one_property_fwd.hpp
|   |           `-- one_property.hpp
|   `-- src
|       |-- hash
|       |   `-- one_property_hash.cpp
|       |-- io
|       |   `-- one_property_io.cpp
|       |-- odb
|       |   `-- one_property_options.odb
|       |-- serialization
|       |   `-- one_property_ser.cpp
|       |-- test_data
|       |   `-- one_property_td.cpp
|       `-- types
|           `-- one_property.cpp
`-- hello_world.org
```

Listing 8.4: Filesystem tree for Hello World component after generation.

Type definition

Enabled facets include `hash`, `io`, `odb`, *etc.*[21] For completeness, we'll also have a peek at one of the generated files, the type definition of `one_property.hpp` shown below. As decoration-related options were not selected — such as licence, modeline and so on — no decoration was generated. For the same reason, default methods have been outputted, such as the "complete constructor" — a constructor that takes all properties as arguments, which in this case is just `property`. Dogen supplies a large number of options to control the generation of these constructs via non-structural variability, but for simplicity all of these have been omitted.

```
#ifndef DOGEN_HELLO_WORLD_TYPES_ONE_PROPERTY_HPP
#define DOGEN_HELLO_WORLD_TYPES_ONE_PROPERTY_HPP

#if defined(_MSC_VER) && (_MSC_VER >= 1200)
#pragma once
#endif

#include <string>
#include <algorithm>
#include "dogen.hello_world/serialization/one_property_fwd_ser.hpp"

namespace dogen::hello_world {

/**
 * @brief Hello World class.
 */
class one_property final {
public:
    one_property() = default;
    one_property(const one_property&) = default;
    one_property(one_property&&) = default;
```

21 The roles and responsibilities of each facet shown are as detailed in see Section 6.1.2.

```
    ~one_property() = default;

public:
    explicit one_property(const std::string& property);

[...]

public:
    /**
     * @brief This is a sample property.
     */
    /**@{*/
    const std::string& property() const;
    std::string& property();
    void property(const std::string& v);
    void property(const std::string&& v);
    /**@}*/

public:
    bool operator==(const one_property& rhs) const;
    bool operator!=(const one_property& rhs) const {
        return !this->operator==(rhs);
    }

public:
    void swap(one_property& other) noexcept;
    one_property& operator=(one_property other);

private:
    std::string property_;
[...]
```

Listing 8.5: Generated source code for class in Hello World model.

The source code listing concludes our brief overview of Dogen. Many features have been left out due to space constraints, such as C# support, as did the advanced usage of the code generator. Presently, the best example of Dogen's usage is Dogen itself, so the interested reader is directed to the project in GitHub. The next section will perform a brief overview of the testing framework used in the MRI, centred around reference products.

8.2 REFERENCE PRODUCTS

Historical context

Originally Dogen tried to make use of all of its features within the main code base, in keeping with our views on dogfooding. Eventually, as the number of features increased, it became impractical to continue doing so, resulting in the introduction of *test models*. At the start, all test models were kept with the Dogen code base to facilitate project management. Unfortunately, as the code base became larger, it became unfeasible to continue using this mono-repository approach due to long build times and checkouts; at this point, all test code was separated into its own repositories.

As our understanding of the theory improved this approach was shown to be the correct one, and so a reference product was assigned to each supported TS. The purpose of the reference product is to exercise all features available on a TS, for three interrelated reasons:

- **Conformance**: reference products fulfil MASD's requirements around conformance testing (*cf.* Requirement 4.2.6), ensuring that all supported features are working as specified. Dogen runs a suite of unit tests that regenerate all reference products in its CI/CD pipeline, to ensure correctness.

- **Test Driven Development (TDD)**: features must be added to the reference models first, with the Dogen code base being subsequently updated to get the tests to pass. In the future, as we expand to external users, the expectation is that they will create MWE's by changing the reference models via change requests as part of the MRI Development Process (*cf.* Section 5.4.2.1).

- **Documentation and Samples**: reference products serve as samples for new users who wish to gain a better understanding of Dogen's capabilities. As these products do not have any additional behaviour, they are "bare-bones" demonstrations of specific Dogen features.

Second line of defense

Note that the Dogen code base is still deemed as vital for both conformance testing as well as a form of documentation for advanced use cases; and all features that are not intended to be user facing are *only* tested within Dogen models. The size of this feature set is still considerable because a significant portion of Dogen exists only to satisfy Dogen's internal use cases. Amongst many other features, this includes the transform framework and its integration with stitch (*cf.* Section 8.1.5). In other words, since we could not cover all supported features and since its use cases are too advanced for new comers, Dogen is seen as a second line of defense rather than the primary mechanism for these aspects.

The reference products are named after the TS they cover: `cpp_ref_impl` is the *C++ Reference Implementation* — now known as the C++ Reference Product — and `CSharpRefImpl` is the *C# Reference Implementation* — now known as the C# Reference Product. The next two section provide a brief overview of each product.

8.2.1 *C++ Reference Product*

At just over 85 thousand lines of code, 90% of which code-generated, the C++ Reference Product is the largest reference product of the MRI. It is composed of a number of component models, which can be described by grouping related functionality.

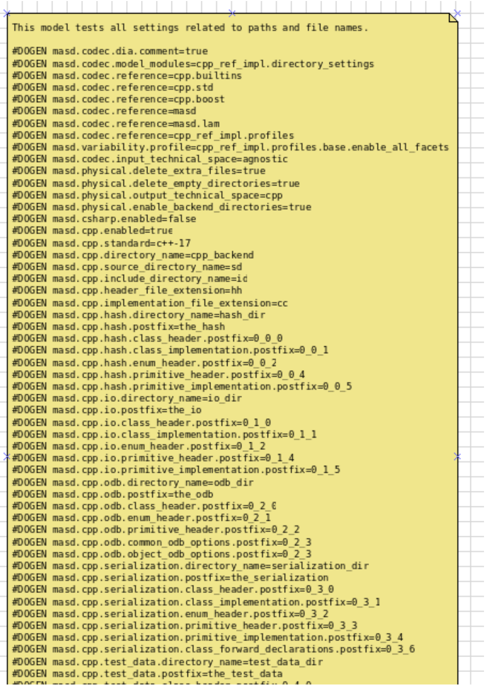
- **Facet enablement tests**: A number of models are used just to ensure that enabling and disabling facets works as expected. These include `enable_facet_hash`, `enable_facet_io` and so on, with all models following the same naming convention.

- **Platform tests**: Some models test the integration with PDM's, such as the C++ Standard Library (`std_model`) and the Boost C++ library[22] (`boost_model`). In general, for each new PDM added, there should be an associated PDM component to test it.

- **Language features**: The `cpp_model` is responsible for testing the core features supported in the C++ language. It covers only the latest stable version, C++ 17. In the future, this model will be renamed to take the version into account. We also have `cpp_98` to validate our support for a legacy version of C++.

22 https://www.boost.org/

- **MASD-levels support**: Dogen has a number of configuration options designed to support the varying usage levels defined by MASD (*cf.* Section 5.4.2.3), including `force_write`, `delete_extra` and `out_of_sync` to check for the deletion of unmanaged files, `disable_facet_folders` to check for the flattening of the directory structure, `ignore_extra` to check that we ignore unmanaged files when requested via regular expressions, `skip_empty_dirs` to test our management of empty directories, *etc.*



```
This model tests all settings related to paths and file names.

#DOGEN masd.codec.dia.comment=true
#DOGEN masd.codec.model_modules=cpp_ref_impl.directory_settings
#DOGEN masd.codec.reference=cpp.builtins
#DOGEN masd.codec.reference=cpp.std
#DOGEN masd.codec.reference=cpp.boost
#DOGEN masd.codec.reference=masd
#DOGEN masd.codec.reference=masd.lam
#DOGEN masd.codec.reference=cpp_ref_impl.profiles
#DOGEN masd.variability.profile=cpp_ref_impl.profiles.base.enable_all_facets
#DOGEN masd.codec.input_technical_space=agnostic
#DOGEN masd.physical.delete_extra_files=true
#DOGEN masd.physical.delete_empty_directories=true
#DOGEN masd.physical.output_technical_space=cpp
#DOGEN masd.physical.enable_backend_directories=true
#DOGEN masd.csharp.enabled=false
#DOGEN masd.cpp.enabled=true
#DOGEN masd.cpp.standard=c++-17
#DOGEN masd.cpp.directory_name=cpp_backend
#DOGEN masd.cpp.source_directory_name=sd
#DOGEN masd.cpp.include_directory_name=id
#DOGEN masd.cpp.header_file_extension=hh
#DOGEN masd.cpp.implementation_file_extension=cc
#DOGEN masd.cpp.hash.directory_name=hash_dir
#DOGEN masd.cpp.hash.postfix=the_hash
#DOGEN masd.cpp.hash.class_header.postfix=0_0_0
#DOGEN masd.cpp.hash.class_implementation.postfix=0_0_1
#DOGEN masd.cpp.hash.enum_header.postfix=0_0_2
#DOGEN masd.cpp.hash.primitive_header.postfix=0_0_4
#DOGEN masd.cpp.hash.primitive_implementation.postfix=0_0_5
#DOGEN masd.cpp.io.directory_name=io_dir
#DOGEN masd.cpp.io.postfix=the_io
#DOGEN masd.cpp.io.class_header.postfix=0_1_0
#DOGEN masd.cpp.io.class_implementation.postfix=0_1_1
#DOGEN masd.cpp.io.enum_header.postfix=0_1_2
#DOGEN masd.cpp.io.primitive_header.postfix=0_1_4
#DOGEN masd.cpp.io.primitive_implementation.postfix=0_1_5
#DOGEN masd.cpp.odb.directory_name=odb_dir
#DOGEN masd.cpp.odb.postfix=the_odb
#DOGEN masd.cpp.odb.class_header.postfix=0_2_0
#DOGEN masd.cpp.odb.enum_header.postfix=0_2_1
#DOGEN masd.cpp.odb.primitive_header.postfix=0_2_2
#DOGEN masd.cpp.odb.common_odb_options.postfix=0_2_3
#DOGEN masd.cpp.odb.object_odb_options.postfix=0_2_3
#DOGEN masd.cpp.serialization.directory_name=serialization_dir
#DOGEN masd.cpp.serialization.postfix=the_serialization
#DOGEN masd.cpp.serialization.class_header.postfix=0_3_0
#DOGEN masd.cpp.serialization.class_implementation.postfix=0_3_1
#DOGEN masd.cpp.serialization.enum_header.postfix=0_3_2
#DOGEN masd.cpp.serialization.primitive_header.postfix=0_3_3
#DOGEN masd.cpp.serialization.primitive_implementation.postfix=0_3_4
#DOGEN masd.cpp.serialization.class_forward_declarations.postfix=0_3_6
#DOGEN masd.cpp.test_data.directory_name=test_data_dir
#DOGEN masd.cpp.test_data.postfix=the_test_data
```

Figure 8.12: Fragment of the directory settings model configuration.

- **Physical configuration**: Though somewhat misleadingly named, the model `directory_settings` is responsible for exercising all of Dogen's configuration options regarding the naming of physical entities. Figure 8.12 contains a fragment of these options.

- **PIM support**: The `lam_model` is designed to generate both C++ and C# code, ensuring our PIM support works as intended — LAM standing for Language Agnostic Model. At present Dogen only supports basic type mapping, with much work still outstanding for PIMs to be considered first class citizens.

- **Codec-specific features**: In some cases we need to validate features that are only available on a given codec. At present there are two such models: `compressed`, which tests compressed diagrams in Dia, and `two_layers`, which contains a Dia model using multiple layers. In general, the policy is to avoid having codec-specific features and, by implication, avoid codec-specific tests, so this group of components is not expected to grow.

- **RDBMS support**: The `northwind` model exercises Dogen's relational database support, which at present requires ODB. In the future, it may serve as the basis for an implementation that does not rely on a cartridge — an approach which is presently under analysis.

8.2.2 C# Reference Product

The C# reference product, `CSharpRefImpl`, has the same aims as the C++ reference product but the feature coverage is not symmetric across TSs. Its much smaller size in LOC is indicative of this asymmetry: 12 thousand versus 85 thousand. This is to be expected: the C++ TS is the main target of our work because Dogen relies on it directly; conversely, C# was introduced largely as a device to protect ourselves against hard-wiring the domain architecture and implementation to a specific TS.

At present, the product is composed of the following component models:

- **`CSharpModel`**: Tests the C# language, and a small number of types from the base library. There is no separation between language and library for the moment because we only support a small number of types from C#'s Base Class Library.
- **`DirectorySettings`**: C# version of the directory settings model, with the same objectives as the C++ version — to exercise all features related to the configuration of the PMM.
- **`LamModel`**: C# version of the Language Agnostic Model, the test model used to verify PIM functionality.

Our product backlog has a series of stories related to missing features in C# support, most of which are on the roadmap for the v2.0 release. As part of the work on implementing those features, the C# reference product will be augmented with the missing test components — making it similar in shape to the C++ reference product.

These words conclude our brief introduction to the MRI product line. The next and final section of the chapter discusses the lessons learned by implementing Dogen and the reference products.

8.3 evaluation

Overview

Dogen is the main application of both MASD and the MRI, which is to say of Dogen itself, giving us firm grounds from whence an evaluation of both methodology and tooling can be performed. As with org-mode (*cf.* Section 7.5), our conclusions are split between the application (Section 8.3.1) — that is, the use of the methodology and tooling from the perspective of a regular user — and meta-application (Section 8.3.2) — that is, how the development of Dogen impacted the methodology as a whole.

8.3.1 *Application Evaluation*

By far, the biggest problem faced with Dogen application as a user were the frequent *stalls* due to fundamental issues with the tool, either at the architectural level or with its conceptual framework. In many cases, these issues have taken long periods of time to be addressed adequately, as the size and complexity of the present document attests. As explained at length in (Craveiro, 2021c), the creation of MDE tooling is an extremely expensive exercise in general, and it is a cost most software companies are unwilling to absorb. Dogen's development has demonstrated precisely why it is so. From the perspective of more than two decades of industrial software engineering, it is our firm opinion that a project such as Dogen could not be achieved within an industrial setting because the pressure to deliver working products would not allow for the necessary time on fundamental research.

Costly exercise

The "stop-start" development also had a negative impact on Dogen itself; we have often been side-tracked when implementing a given story due to one or more missing features in the tool, the implementation of which also required adding additional features, creating a recursive loop that continued several levels deep. As a result, it proved very difficult to keep a focus on the features being implemented. A small example should suffice to give a flavour of the dilemmas faced:

Stop-start development

- Whilst trying to add code generation support for PMM entities, such as facets, parts, *etc.*, we ran into a limitation on how stitch templates were handled via an external tool.
- In order to address this problem, we promoted the stitch templates themselves to the LMM as regular metamodel entities. This enabled us to perform a tighter workflow, including the expansion of stitch templates as a regular M2T transform, rather than a cartridge.
- However this then resulted in a new problem: since stitch templates were model elements rather than stand-alone files, it was necessary to extract them in and out of Dia diagrams for editing purposes. The process was cumbersome and error prone, and greatly reduced development velocity.
- Whilst reflecting on this matter, we realised that a long-running org-mode story in the product backlog could be used to solve both problems elegantly, so we implemented org-model (*cf.* Chapter 7).
- Finally, the original task was resumed.

Recursive stories

This entire loop took over a year to play out, and many excursions of a similar (or even greater) magnitude were made during the development of Dogen.

On the other hand, these two negative findings reinforce our belief in the approach. No project other than Dogen will need to absorb this large cost once the framework and the methodology have been put in place, provided one is willing to accept MASD's rigid structure. It is a one-off cost that Dogen itself will pay, with subsequent applications having a much smaller cost base

One-off cost

because their features are expected to fit in with the existing framework. For example, though not completely trivial, adding a new TS should be a much easier affair in the future, with the main requirement being to follow the patterns supplied in the C++ and C# implementations. In summary, Dogen demonstrates the approach works, but also that it will only be suitable for end users once the tool is mature enough to handle all of its own use cases — leading us to meta-application.

8.3.2 *Meta-application Evaluation*

Self-validation

It is our firm opinion that the application of MASD to Dogen, and the use of Dogen to develop MASD, has proven the validity of both tooling and methodology. The six core principles of MASD (*cf.* Section 5.2) have been forged in empirical application during Dogen's development, and were selected after many different approaches had been tried; in the same vein, all of MASD processes and actors (*cf.* Section 5.4) have evolved precisely from the continued observation and reflection on the practices within Dogen development — even if somewhat restricted by having a single developer taking on different roles.

Tight feedback loop

Furthermore, we believe that this virtuous cycle was instrumental in shaping both Dogen and MASD (*cf.* Section 5.4.2.4), since it allowed us to escape most of the challenges of a dual track process. Whilst not without its flaws — such as the deeply nested recursion described in the previous section, and a risk of over-fitting to Dogen's use cases — in the end, the approach supplied us with a tight feedback loop that removed all distractions that come from having to develop two distinct software products.

In conclusion, dogfooding and bootstrapping aren't merely an expedient approach taken for the development of Dogen; we believe they are a crucial ingredient to the approach, and the main reason why both Dogen and MASD reached its present state.

Part IV

OUTLOOK

9 CONCLUSIONS

It is our view that software engineering is inherently a modeling activity, and that the complexity of software will overwhelm our ability to effectively maintain mental models of a system. By making the models explicit and by using tools to manipulate, analyze and manage the models and their relationships, we are relieving significant cognitive burden and reducing the accidental complexities associated with maintaining mentally held models.

— France and Rumpe (France and Rumpe, 2007)

Monograph overview

This monograph introduced *Model Assisted Software Development* (*MASD*), a novel methodology for the construction and evolution software products and product lines, as well as its associated tooling, the *MASD Reference Implementation* (*MRI*). The document started by identifying a number of issues with the state of the art in code generation, most pertinent of which a lack of a developer-centric approach, and captured a subset of these problems as a set of requirements. The MASD SDM was then put forward as a solution, including its domain architecture, after which we demonstrated the feasibility of the approach by means of two case studies: the incorporation of org-mode support as an enabler for literate modeling; and the implementation of the MRI itself as a product line, using MASD principles and driving MASD's development via a carefully designed virtuous circle.

Chapter overview

The present chapter reflects on the work carried out and is comprised of two sections. Section 9.1 looks backwards, re-examining the original requirements to assess how well they were addressed by methodology and tooling. Then, Section 9.2 looks forwards to the future, discussing a number of future applications for MASD — particularly in the realm of integration with existing MDE tooling, which was deliberately absent from the present work. We begin then by assessing how MASD has fared.

9.1 MASD REQUIREMENTS REVISITED

MASD is best evaluated by revisiting the requirements formulated earlier in this manuscript (*cf.* Chapter 4). Previous chapters have already alluded to how these were addressed by MASD in different contexts — particularly at the methodology level (*cf.* Chapter 5) — so our objective is merely to summarise conclusions to paint the overall picture. To facilitate the review, requirements have been grouped into related topics.

9.1.1 *Identity Related Requirements*

R-1:	Well-Defined Purpose
R-2:	Well-Defined Identity
R-3:	Well-Defined Target Audience

The objective of this group of requirements is to ensure that the new methodology is distinct from MDE and related *MD** approaches, and that both its *raison d'être* as well as its boundaries are clear and obvious to prospective end users. The requirements are all explicitly addressed by the methodology's core values (Section 5.2). By first discerning the issues surrounding scope and identity within MDE (*cf.* Section 2.3) and then by subsequently positioning MASD in a clear manner across all of the selected dimensions, we ensured that the methodology addressed all of the identified issues.

9.1.2 *Process and Integration Requirements*

R-6:	SDM Integration
R-7:	Clear Governance Model
R-5:	Cater for Evolution
R-14:	Clear Separation of End-users and Tool Developers

Collectively, these four requirements deal with issues of integration and governance, ensuring all actors and processes have been clearly identified and all responsibilities have been attributed. These requirements are fundamental to the approach, in light of our characterisation of MDE as an unstructured body of knowledge (*cf.* Section 2.3); our target audience can only be serviced with a structured process, and a clear path for SDM integration (*e.g.* working with Agile). The methodology dedicated a section to specifically address these requirements: *Processes and Actors* (Section 5.4). It ensured clarity and transparency in this regard, with roles and responsibilities clearly specified. In addition, the development of Dogen using Agile (*cf.* Section 8.1.3) was a test bed to ensure SDM integration worked as planned (*c.f.* Section 8.1.3)..

9.1.3 *Modeling Requirements*

R-8:	Support for PIMs and PSM
R-9:	Support for PDM
R-10:	Support for Variant Management and Product Lines
R-11:	Extensible Catalogue of Schematic and Repetitive Code

Whilst MASD's objective is to allow for a restricted and well-defined use of MDE techniques, some core aspects of MDE are still considered to be extremely important and must be supported as first-class citizens. The domain architecture (Chapter 6) was designed specifically to address all of these requirements, from product lines to the various supported model kinds (*e.g.*

PIMs, PDMs, PSMs). In addition, due to MASD's artefact-centric view of the software engineering world, the methodology also provides clear definitions of key concepts such as platforms (*cf.* Section 6.1.7), meaning the roles of PIMs, PSMs and PDMs are clearly defined.

9.1.4 *Tooling Requirements*

R-12:	"End-to-End" Solution
R-13:	Prioritise Black-Boxing
R-15:	Prioritise Tooling Integration
R-16:	Support Incremental Use of Features

Dogen, the product in the MRI product line responsible for implementing the code generator, has addressed all of these requirements comprehensively (Section 8.1). It provides a single black box tool which reads input models and produces generated code; it also supplies a library ready for integration with different development environments such as IDEs, as well as a codec framework that facilitates the development of new input formats — as demonstrated by our addition of org-model (*cf.* Chapter 7). Last but not least, via non-structural variability, Dogen is able to support the incremental use of features specified by the MASD Application Process (*cf.* Section 5.4.2.3).

9.1.5 *Testing Requirements*

R-17:	Conformance Testing

The reference products within the MRI product line are responsible for both declaring the available features and ensuring conformance to the declared feature set (Section 8.2). With over 90 thousand LOC, these products perform an extensive testing of Dogen's features that are user facing. In addition, Dogen itself provides coverage for the remaining features, designed specifically for its own development. Finally, a TDD approach ensures that no feature is implemented without a corresponding change; any future changes to the code base are validated against this large test set to ensure no feature is broken inadvertently.

9.2 FURTHER WORK

MASD and the MRI have had a long gestation period, taking at times circuitous paths. The present document marks the end of this difficult introductory phase, with the conclusion of the bulk of the theoretical work, and the beginning of what we envision as the growth phase. In this next phase of MASD's lifecycle, the focus will shift towards software engineering, striving for strong growth across a fixed set of dimensions and opening avenues for engineering:

- the addition of new platforms to existing TSs;
- the addition of new facets to existing TSs, in some cases requiring new metamodel elements in the LMM;

Engineering avenues

- the addition of new TSs such as Java, and bringing existing TSs to feature parity;
- the increase of non-structural variability on existing archetypes.

All of these activities are characterised by a strong emphasis in software development, and are expected to have a narrow impact on the domain architecture described in the present manuscript (*cf.* Chapter 6). By implication, they have a limited potential for research, outside experience reports. The majority of the 800 stories captured in the product backlog fall under one or more of these four dimensions.

Nonetheless, the product backlog does contain several lines of enquiry that will form the basis for new research. These can be summarised as follows:

- **Solving in the VMM**: The geometry of physical space was designed to allow for automated solving of valid configurations, both for trivial scenarios — *e.g.* if the region of physical space for the C++ TS is disabled this implies that all archetypes within that region are disabled — as well as more complex ones. In particular, the introduction of dependencies between archetypes will make it possible to determine implied configuration graphs in VMM space, given an input configuration: *e.g.*, if a user selects facet `F0` for class `A` which has a property `p` of class `B`, that implies that `B` must also enable facet `F0`.
- **Typed test-data generators**: At present our test data generation is not intelligent; if a user creates a property of type `std::string`, Dogen's data fountains generate trivial strings with a hard-coded prefix and a counter that ensures uniqueness. Similarly, for integral types we rely on counters. A more intelligent approach which is currently under analysis is the introduction of metatypes for the data generators, and associate typed data fountains to each metatype. For example, a primitive can be associated with a fountain metatype of "telephone numbers", possibly further parameterised via non-structural variability with a country of origin, and then be used to generate realistic phone numbers.

Research avenues

- **Integrating LSP with MASD modeling**: Following on from the work of Rodriguez-Echeverria *et al.* (Rodriguez-Echeverria et al., 2018), and in line with our objective of pervasive integration (P-2), analysis has begun on supporting IDE-like features for modeling via the *de facto* standard of LSP. In addition, LSP can also provide a route towards integrating roundtrip engineering, which at present is a non-goal for MASD due to its perceived complexity. The main benefit of using LSP is its agnosticism; it is supported by a range of tooling from text editors such as Emacs to full featured IDEs like Eclipse.
- **Adding support for Ecore**: Though ostensibly just another codec with which to represent MASD input models, Ecore also acts as a gateway

for the integration of MASD with the wider MDE tooling infrastructure. Once Ecore has been integrated, many of the complex use cases which have been externalised out of MASD — *e.g.* model synchronisation, merging, querying, evolution, *etc.* — would be supported.

- **Integration of fuzzing with modeling**: Sharing some similarities with typed test-data generators, analysis is presently underway on integrating fuzzing platforms in each TS with generated code. Whilst this activity has a strong engineering component, we also expect it will provide novel approaches to the problem of integrating modeling and fuzzing.

Further applications

Lastly, a word on further applications. An explicit trade-off was made as part of the present work was to deliberately avoid any applications of MASD outside of the MRI until we could comprehensively pass the defined fitness function (*cf.* Section 5.4.2.4). We considered such work to fall under further applications, for, in the words of Ritchie, "it's best to confuse only one issue at a time." (Ritchie, 1978) (p. 22) However, having MASD applications outside of MASD is undeniably a significant milestone, particularly with a view to strengthening our position of MASD as a methodology for software engineers with little to no MDE experience. So the final avenue for further work will be to undertake the creation of an industrial grade software product, with a complexity larger than that of the MRI.

BIBLIOGRAPHY

Abrahams, David and Aleksey Gurtovoy (2004). *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Pearson Education (cit. on p. 110).

Al Saad, Mohammad et al. (2008). "ScatterClipse: a model-driven tool-chain for developing, testing, and prototyping wireless sensor networks". In: *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on*. IEEE, pp. 871–885 (cit. on p. 23).

Ambler, Scott W (2007). "Agile Model driven development (AMDD)". In: *XOOTIC MAGAZINE, February* (cit. on p. 26).

Ameller, David et al. (2017). "Towards continuous software release planning". In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 402–406 (cit. on p. 10).

Andolfato, Luigi et al. (2014). "Experiences in Applying Model Driven Engineering to the Telescope and Instrument Control System Domain". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer, pp. 403–419 (cit. on pp. 3, 31, 41).

Andreessen, Marc (2011). "Why software is eating the world". In: *Wall Street Journal* 20.2011, p. C2 (cit. on p. 10).

Arlow, Jim, Wolfgang Emmerich, and John Quinn (1998). "Literate modelling—capturing business knowledge with the uml". In: *International Conference on the Unified Modeling Language*. Springer, pp. 189–199 (cit. on p. 127).

Arlow, Jim and Ila Neustadt (2004). *Enterprise patterns and MDA: building better software with archetype patterns and UML*. Addison-Wesley Professional (cit. on pp. 127, 128).

Asadi, Mohsen and Raman Ramsin (2008). "MDA-based methodologies: an analytical survey". In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, pp. 419–431 (cit. on pp. 16, 17).

Badreddin, Omar Bahy, Andrew Forward, and Timothy C Lethbridge (2012). "Model oriented programming: an empirical study of comprehension." In: *CASCON*. Vol. 12, pp. 73–86 (cit. on p. 11).

Badreddin, Omar and Timothy C Lethbridge (2013). "Model oriented programming: bridging the code-model divide". In: *Proceedings of the 5th International Workshop on Modeling in Software Engineering*. IEEE Press, pp. 69–75 (cit. on p. 128).

Batory, Don (2005). "Feature models, grammars, and propositional formulas". In: *International Conference on Software Product Lines*. Springer, pp. 7–20 (cit. on p. 118).

Beck, Kent et al. (2001). "Manifesto for agile software development". In: (cit. on pp. 10, 64).

Bell, Alex E (2004). "Death by UML fever". In: *Queue* 2.1, p. 72 (cit. on p. 33).

Berdonosov, Victor and Elena Redkolis (2011). "TRIZ-fractality of computer-aided software engineering systems". In: *Procedia Engineering* 9, pp. 199–213 (cit. on p. 12).

Beyer, Betsy et al. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc." (cit. on p. 10).

Bézivin, Jean (2003). "MDA: From hype to hope, and reality". In: *The 6th International Conference on the Unified Modeling Language* (cit. on pp. 20, 33).

– (2005). "On the unification power of models". In: *Software & Systems Modeling* 4.2, pp. 171–188 (cit. on pp. 3, 17).

Biermann, Alan W (1985). "Automatic programming: A tutorial on formal methodologies". In: *Journal of Symbolic Computation* 1.2, pp. 119–142 (cit. on p. 11).

Boilerplate text (2021). *Boilerplate text — Wikipedia, The Free Encyclopedia*. [Online; accessed 26-June-2021]. URL: https://en.wikipedia.org/wiki/Boilerplate_text (cit. on p. 84).

Booch, Grady, James Rumbaugh, and Ivar Jacobson (Jan. 1999). *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Vol. 10. ISBN: 0321267974 (cit. on p. 58).

Booch, Grady et al. (2004). "An MDA manifesto". In: *Business Process Trends/MDA Journal* (cit. on pp. 17, 20, 57).

Boris Kolpackov (2021). *The build2 Toolchain Introduction*. [Online; accessed 07-July-2021]. URL: https://build2.org/build2-toolchain/doc/build2-toolchain-intro.xhtml (cit. on p. 95).

Bosch, Jan et al. (2001). "Variability issues in software product lines". In: *International Workshop on Software Product-Family Engineering*. Springer, pp. 13–21 (cit. on p. 65).

Bou Ghantous, G and Asif Gill (2017). "DevOps: Concepts, practices, tools, benefits and challenges". In: *PACIS2017* (cit. on p. 10).

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer (2012). *Model-driven software engineering in practice*. Vol. 1. 1. Morgan & Claypool Publishers, pp. 1–182 (cit. on pp. 15, 16, 19, 34).

Brooks, Frederick P (1974). "The mythical man-month". In: *Datamation* 20.12, pp. 44–52 (cit. on pp. 29, 140).

Capra, Eugenio, Chiara Francalanci, and Francesco Merlo (2008). "An empirical study on the relationship between software design quality, development effort and governance in open source projects". In: *IEEE Transactions on Software Engineering* 34.6, pp. 765–782 (cit. on p. 63).

Clark, Tony and Pierre-Alain Muller (2012). "Exploiting model driven technology: a tale of two startups". In: *Software & Systems Modeling* 11.4, pp. 481–493 (cit. on pp. 31, 33, 38, 40, 41, 60).

Clauß, Matthias (2001). "Generic modeling using UML extensions for variability". In: *Workshop on Domain Specific Visual Languages at OOPSLA*. Vol. 2001 (cit. on p. 116).

Coad, Peter, Jeff de Luca, and Eric Lefebvre (1999). *Java modeling color with UML: Enterprise components and process with Cdrom*. Prentice Hall PTR (cit. on p. 97).

Colby Pike (2018). *Project Layout - Survey Results and Updates*. [Online; accessed 07-July-2021]. URL: https://vector-of-bool.github.io/2018/09/16/layout-survey.html (cit. on p. 95).

– (2021). *The Pitchfork Layout (PFL)*. [Online; accessed 07-July-2021]. URL: https://api.csswg.org/bikeshed/?force=1&url=https://raw.githubusercontent.com/vector-of-bool/pitchfork/develop/data/spec.bs (cit. on p. 95).

Concept map (2021). *Concept map — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-June-2021]. URL: https://en.wikipedia.org/wiki/Concept_map (cit. on p. 129).

Cook, Steve (2006). “Object technology - a grand narrative?” In: *European Conference on Object-Oriented Programming*. Springer, pp. 174–179 (cit. on p. 39).

Craveiro, Marco (2021a). *Dogen v1.0.30, "Estádio Joaquim Morais"*. URL: https://github.com/MASD-Project/dogen/releases/tag/v1.0.30 (visited on 08/27/2021) (cit. on p. 127).

– (2021b). “Experience Report of Industrial Adoption of Model Driven Development in the Financial Sector”. In: DOI: 10.5281/zenodo.5767247 (cit. on pp. 22–25, 30, 31, 39, 41, 43, 44, 47, 48, 57, 62, 142).

– (2021c). *Notes on Model Driven Engineering*. Zenodo. DOI: 10.5281/zenodo.5789846 (cit. on pp. 3, 4, 14, 15, 17–19, 21, 23–26, 30, 38, 47–50, 57, 59, 61, 65, 66, 73, 74, 78, 79, 83, 85, 102, 103, 115, 140, 143, 148, 158).

– (2021d). “Survey of Special Purpose Code Generators”. In: DOI: 10.5281/zenodo.5790875 (cit. on pp. 39, 40, 47, 51, 52, 60, 61, 71, 85, 88, 96, 104, 142).

Crowston, Kevin et al. (2012). “Free/Libre open-source software development: What we know and what we do not know”. In: *ACM Computing Surveys (CSUR)* 44.2, p. 7 (cit. on p. 63).

Cuesta, César Cuevas (2016). “Metaherramientas MDE para el diseño de entornos de desarrollo de sistemas distribuidos de tiempo real”. PhD thesis. Universidad de Cantabria (cit. on p. 15).

Cunningham, Ward (1992). “The WyCash portfolio management system”. In: *ACM SIGPLAN OOPS Messenger* 4.2, pp. 29–30 (cit. on p. 24).

Czarnecki, Krzysztof (1998). *Generative programming: Principles and techniques of software engineering based on automated configuration and fragment-based component models*. Computer Science Department, Technical University of Ilmenau (cit. on pp. 14, 22).

Czarnecki, Krzysztof, Simon Helsen, and Ulrich Eisenecker (2005). “Formalizing cardinality-based feature models and their specialization”. In: *Software process: Improvement and practice* 10.1, pp. 7–29 (cit. on p. 14).

Czarnecki, Krzysztof and Andrzej Wasowski (2007). “Feature diagrams and logics: There and back again”. In: *Software Product Line Conference, 2007. SPLC 2007. 11th International*. IEEE, pp. 23–34 (cit. on p. 118).

Czarnecki, Krzysztof et al. (2000). “Generative programming and active libraries”. In: *Generic Programming*. Springer, pp. 25–39 (cit. on p. 14).

Davis, James (2003). “GME: the Generic Modeling Environment”. In: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, pp. 82–83 (cit. on p. 63).

Dedehayir, Ozgur and Martin Steinert (2016). “The hype cycle model: A review and future directions”. In: *Technological Forecasting and Social Change* 108, pp. 28–41 (cit. on p. 34).

Del Gaudio, Rosa and António Branco (2009). “Language independent system for definition extraction: First results using learning algorithms”. In: *Proceedings of the 1st Workshop on Definition Extraction*. Association for Computational Linguistics, pp. 33–39 (cit. on p. 80).

Derby, Esther, Diana Larsen, and Ken Schwaber (2006). *Agile retrospectives: Making good teams great*. Pragmatic Bookshelf (cit. on p. 143).

Escott, Eban et al. (2011a). “Architecture-centric model-driven web engineering”. In: *2011 18th Asia-Pacific Software Engineering Conference*. IEEE, pp. 106–113 (cit. on p. 23).

Escott, Eban et al. (2011b). “Model-driven web form validation with UML and OCL”. In: *International Conference on Web Engineering*. Springer, pp. 223–235 (cit. on p. 23).

Evans, Eric (2004). *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley. ISBN: 0321125215 9780321125217 (cit. on pp. 66, 77).

Exchange, Stack (2018). *Stack Exchange Users*. URL: https://stackexchange.com/sites?view=list#users (visited on 08/12/2018) (cit. on p. 36).

Eysholdt, Moritz and Heiko Behrens (2010). "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, pp. 307–309 (cit. on p. 24).

Feitelson, Dror G, Eitan Frachtenberg, and Kent L Beck (2013). "Development and deployment at facebook". In: *IEEE Internet Computing* 17.4, pp. 8–17 (cit. on p. 9).

Fisher, Alan S and Alan S Fisher (1988). *CASE: using software development tools*. Vol. 2. Wiley New York (cit. on pp. 9, 11–13).

France, Robert B et al. (2006). "Model-driven development using UML 2.0: promises and pitfalls". In: *Computer* 39.2, pp. 59–66 (cit. on p. 33).

France, Robert (2008). "Fair treatment of evaluations in reviews". In: *Software and Systems Modeling* 7.3, pp. 253–254 (cit. on p. 32).

France, Robert and Bernhard Rumpe (2007). "Model-driven development of complex software: A research roadmap". In: *2007 Future of Software Engineering*. IEEE Computer Society, pp. 37–54 (cit. on pp. 20, 39, 44, 75, 163).

González, AS, DS Ruiz, and GM Perez (2010). "Emf4cpp: a c++ ecore implementation". In: *DSDM 2010-Desarrollo de Software Dirigido por Modelos, Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2010), Valencia, Spain* (cit. on p. 145).

Greifenberg, Timo et al. (2015a). "A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages". In: *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*. IEEE, pp. 74–85 (cit. on pp. 23, 94).

Greifenberg, Timo et al. (2015b). "Integration of handwritten and generated object-oriented code". In: *International Conference on Model-Driven Engineering and Software Development*. Springer, pp. 112–132 (cit. on pp. 23, 94).

Groher, Iris and Markus Voelter (2007). "Expressing feature-based variability in structural models". In: *In Workshop on Managing Variability for Software Product Lines*. Citeseer (cit. on pp. 65, 83, 85).

– (2009). "Aspect-oriented model-driven software product line engineering". In: *Transactions on aspect-oriented software development VI*. Springer, pp. 111–152 (cit. on pp. 83, 85).

Group, Object Management (2014). *MDA Guide Version 2.0* (cit. on pp. 17, 18, 33).

Harrison, Warren (2006). "Eating your own dog food". In: *IEEE Software* 23.3, pp. 5–7 (cit. on p. 73).

Hebenstreit, Gernot (2007). "Defining patterns in translation studies: Revisiting two classics of German Translationswissenschaft". In: *Target. International Journal of Translation Studies* 19.2, pp. 197–215 (cit. on p. 80).

Herardian, Ron, Forrest Marshall, and N Hunter Prendergast (n.d.). "Basil Policy-as-code Platform". In: () (cit. on p. 10).

Hutchinson, John, Mark Rouncefield, and Jon Whittle (2011). "Model-driven engineering practices in industry". In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM, pp. 633–642 (cit. on pp. 30, 31, 85).

Hutchinson, John et al. (2011). "Empirical assessment of MDE in industry". In: *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, pp. 471–480 (cit. on pp. 3, 30, 38).

"IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7" (2018). In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153 (cit. on p. 86).

ISO (n.d.). *ISO/IEC 14882:2020(E) Information technology — Programming languages — C++* (cit. on p. 88).

ISO (2011). *C11 Standard*. ISO/IEC 9899:2011. URL: /bib/iso/C11/n1570.pdf (cit. on p. 90).

Jäger, Sven et al. (2016). "An EMF-like UML generator for C++". In: *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE, pp. 309–316 (cit. on p. 145).

Jones, Capers (1994). "Software metrics: good, bad and missing". In: *Computer* 27.9, pp. 98–100 (cit. on p. 9).

Jörges, Sven (2013). *Construction and evolution of code generators: A model-driven and service-oriented approach*. Vol. 7747. Springer (cit. on pp. 11–13, 15, 16).

Jouault, Frédéric et al. (2008). "ATL: A model transformation tool". In: *Science of computer programming* 72.1-2, pp. 31–39 (cit. on p. 18).

Kevlin Henney (2021). *Exceptional Naming*. [Online; accessed 27-June-2021]. URL: https://kevlinhenney.medium.com/exceptional-naming-6e3c8f5bffac (cit. on p. 108).

Knuth, Donald Ervin (1984). "Literate programming". In: *The computer journal* 27.2, pp. 97–111 (cit. on pp. 125, 128).

Kottemann, Jeffrey E and Benn R Konsynski (1984). "Dynamic Metasystems for Information Systems Development." In: *ICIS*, p. 14 (cit. on p. 87).

Lakos, John (1996). *Large-scale C++ software design*. Vol. 1. Addison-Wesley Reading (cit. on p. 58).

– (2019). *Large-Scale C++ Volume I: Process and Architecture*. Addison-Wesley Professional (cit. on pp. 58, 77).

Lee, Gwendolyn K and Robert E Cole (2003). "From a firm-based to a community-based model of knowledge creation: The case of the Linux kernel development". In: *Organization science* 14.6, pp. 633–649 (cit. on p. 63).

Lemma, Remo and Michele Lanza (2013). "Co-evolution as the key for live programming". In: *2013 1st International Workshop on Live Programming (LIVE)*. IEEE, pp. 9–10 (cit. on p. 142).

Lewis, Bil, Daniel LaLiberte, Richard Stallman, et al. (1993). *GNU Emacs Lisp Reference Manual*. Free Software Foundation (cit. on p. 130).

Linden, Alexander and Jackie Fenn (2003). "Understanding Gartner's hype cycles". In: *Strategic Analysis Report Nº R-20-1971. Gartner, Inc* (cit. on p. 34).

Lions, John (1996). "A Commentary on UNIX 6th Edition with Source Code". In: *Peer-To-Peer Communications* (cit. on p. 9).

Lundell, Björn et al. (2006). "UML model interchange in heterogeneous tool environments: an analysis of adoptions of XMI 2". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer, pp. 619–630 (cit. on pp. 43, 44).

Manset, David et al. (2006). "A formal architecture-centric model-driven approach for the automatic generation of grid applications". In: *arXiv preprint cs/0601118* (cit. on p. 23).

Marco Craveiro (2018). *The Refactoring Quagmire*. [Online; accessed 27-June-2021]. URL: https://mcraveiro.blogspot.com/2018/01/nerd-food-refactoring-quagmire.html (cit. on p. 144).

Marco Craveiro (2021). *Marco Craveiro YouTube Channel*. [Online; accessed 27-June-2021]. URL: https://www.youtube.com/channel/UCZLcCjqOG1VmbSfoAJAf2mA (cit. on p. 144).

Markup language (2021). *Markup language — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-June-2021]. URL: https://en.wikipedia.org/wiki/Markup_language (cit. on p. 129).

Maven Project (2021). *Introduction to the Standard Directory Layout*. [Online; accessed 07-July-2021]. URL: https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html (cit. on p. 95).

McBreen, Pete (2002). *Software craftsmanship: The new imperative*. Addison-Wesley Professional (cit. on p. 31).

Meliá, Santiago et al. (2016). "Comparison of a textual versus a graphical notation for the maintainability of MDE domain models: an empirical pilot study". In: *Software Quality Journal* 24.3, pp. 709–735 (cit. on p. 11).

Mellor, Stephen J (2004). "Agile mda". In: *MDA Journal, www. bptrends. com June* (cit. on pp. 19, 79).

Mellor, Stephen J, Marc Balcer, and Ivar Foreword By-Jacoboson (2002). *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc. (cit. on p. 128).

Meyer, Bertrand (1988). *Object-oriented software construction*. Vol. 2. Prentice hall New York (cit. on p. 24).

Microsoft (2021). *Organize your project to support both .NET Framework and .NET*. [Online; accessed 07-July-2021]. URL: https://docs.microsoft.com/en-us/dotnet/core/porting/project-structure (cit. on p. 95).

Mohagheghi, Parastoo and Vegard Dehlen (2008). "Where is the proof? A review of experiences from applying MDE in industry". In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, pp. 432–443 (cit. on pp. 3, 31, 38, 39, 42, 43).

Morris, K (2016). "Infrastructure as Code: Managing Servers in the Cloud, OReilly Media". In: *Inc, Sebastopol, CA* (cit. on p. 10).

Mussbacher, Gunter et al. (2014). "The relevance of model-driven engineering thirty years from now". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer, pp. 183–200 (cit. on pp. 33, 47, 75).

Nim Project (2022a). *Nim Manual*. [Online; accessed 12-January-2022]. URL: https://nim-lang.org/docs/manual.html#types-distinct-type (cit. on p. 109).

– (2022b). *Nim Programming Language Homepage*. [Online; accessed 12-January-2022]. URL: https://nim-lang.org/ (cit. on p. 109).

OLIVEIRA, Thiago Araújo Silva de (2011). "Geração de código estrutural implantável em nuvens a partir de modelos de componentes independentes de plataforma". MA thesis. Universidade Federal de Pernambuco (cit. on p. 15).

OMG, Object Management Group (2008). "MOF Model to Text Transformation Language 1.0 Specification". In: *Final Adopted Specification (January 2008)* (cit. on p. 18).

– (2012). "Common Object Request Broker Architecture 3.3 Specification". In: *Final Adopted Specification (October 2012)* (cit. on p. 17).

– (2016). "Meta Object Facility (MOF) 2.5.1 Specification". In: *Final Adopted Specification (November 2016)* (cit. on p. 16).

– (2017a). "MOF Query/View/Transformation (QVT) 1.3 Specification". In: *Final Adopted Specification (June 2016)* (cit. on p. 16).

– (2017b). “Unified Modeling Language (UML) 2.5.1 Specification”. In: *Final Adopted Specification (December 2017)* (cit. on p. 16).

Oldevik, Jon et al. (2005). “Toward standardised model to text transformations”. In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, pp. 239–253 (cit. on p. 18).

Org Syntax (2021). *Org Syntax — Org-mode project*. [Online; accessed 27-June-2021]. URL: https://orgmode.org/worg/dev/org-syntax.html (cit. on p. 130).

Overflow, Stack (2018). *What are tags, and how should I use them?* URL: https://stackoverflow.com/help/tagging (visited on 08/12/2018) (cit. on p. 36).

Paige, Richard F and Dániel Varró (2012). “Lessons learned from building model-driven development tools”. In: *Software & Systems Modeling* 11.4, pp. 527–539 (cit. on pp. 3, 31, 41, 43).

Petre, Marian (1995). “Why looking isn’t always seeing: readership skills and graphical programming”. In: *Communications of the ACM* 38.6, pp. 33–44 (cit. on p. 11).

– (2013). “UML in practice”. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, pp. 722–731 (cit. on p. 37).

Piefel, Michael and Toby Neumann (2006). “A Code Generation Metamodel for ULF-Ware”. In: (cit. on pp. 105, 106, 139).

Possompès, Thibaut et al. (2010). “A UML Profile for Feature Diagrams: Initiating a Model Driven Engineering Approach for Software Product Lines”. In: *Journée Lignes de Produits*, pp. 59–70 (cit. on p. 116).

Possompès, Thibaut et al. (2011). “Design of a UML profile for feature diagrams and its tooling implementation”. In: *Software Engineering & Knowledge Engineering*, pp. 693–698 (cit. on p. 116).

Potvin, Rachel and Josh Levenberg (2016). “Why Google stores billions of lines of code in a single repository”. In: *Communications of the ACM* 59.7, pp. 78–87 (cit. on p. 9).

Preston-Werner, Tom (2018). “Semantic Versioning 2.0.0”. In: URL: http://semver.org (visited on 08/12/2018) (cit. on p. 68).

Productionisation (2021). *Productionisation — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-June-2021]. URL: https://en.wikipedia.org/wiki/Productionisation (cit. on p. 141).

Raymond, Eric S (2003). *The art of Unix programming*. Addison-Wesley Professional (cit. on pp. 10, 11).

Renz, Patrick S (2007). *Project governance: implementing corporate governance and business ethics in nonprofit organizations*. Springer Science & Business Media (cit. on p. 63).

Ritchie, Dennis M (1978). “The C Programming Language”. In: (cit. on p. 167).

Rodriguez-Echeverria, Roberto et al. (2018). “Towards a language server protocol infrastructure for graphical modeling”. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pp. 370–380 (cit. on p. 166).

Rolland, Colette, Naveen Prakash, and Adolphe Benjamen (1999). “A multi-model view of process modelling”. In: *Requirements engineering* 4.4, pp. 169–187 (cit. on p. 73).

Rompf, Tiark et al. (2015). “Go meta! A case for generative programming and dsls in performance critical systems”. In: *1st Summit on Advances in Programming Languages (SNAPL 2015)* 32, pp. 238–261 (cit. on p. 14).

Rothenberg, Jeff et al. (1989). “The nature of modeling”. In: *in Artificial Intelligence, Simulation and Modeling* (cit. on p. 74).

Sánchez-Gordón, Mary and Ricardo Colomo-Palacios (2018). "Characterizing DevOps culture: a systematic literature review". In: *International Conference on Software Process Improvement and Capability Determination*. Springer, pp. 3–15 (cit. on p. 10).

Schulte, Eric et al. (2012). "A multi-language computing environment for literate programming and reproducible research". In: *Journal of Statistical Software* 46.3, pp. 1–24 (cit. on pp. 129, 130).

Shirtz, Dov, Michael Kazakov, and Yael Shaham-Gafni (2007). "Adopting model driven development in a large financial organization". In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, pp. 172–183 (cit. on pp. 3, 31).

Single-responsibility principle (2021). *Single-responsibility principle — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-June-2021]. URL: https://en.wikipedia.org/wiki/Single-responsibility_principle (cit. on p. 86).

Stallman, Richard M (1981). "EMACS the extensible, customizable self-documenting display editor". In: *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pp. 147–156 (cit. on p. 130).

Stallworth Williams, Linda (2008). "The mission statement: A corporate reporting tool with a past, present, and future". In: *The Journal of Business Communication (1973)* 45.2, pp. 94–119 (cit. on p. 59).

Staron, Miroslaw et al. (2015). "Classifying obstructive and nonobstructive code clones of Type I using simplified classification scheme: a case study". In: *Advances in Software Engineering* 2015, p. 5 (cit. on p. 22).

Steinberg, Dave et al. (2008). *EMF: eclipse modeling framework*. Pearson Education (cit. on pp. 19, 63).

Steinberg, David et al. (2009). *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional (cit. on p. 19).

Stengel, Richard (2010). *Mandela's Way: Lessons in Life*. Random House (cit. on p. ix).

Stevens, Perdita and Rob Pooley (1999). *Using UML: software engineering with objects and components*. Addison-Wesley Longman Publishing Co., Inc. (cit. on p. 58).

TIOBE, I (2021). "Tiobe Index". In: *Retrieved from Tiobe Index: https://www.tiobe.com/tiobe-index* (cit. on p. 34).

Thomas, Dave (2004). "MDA: Revenge of the Modelers or UML Utopia?" In: *IEEE software* 21.3, pp. 15–17 (cit. on pp. 20, 33).

Torchiano, Marco et al. (2012). "Benefits from modelling and MDD adoption: expectations and achievements". In: *Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling*. ACM, p. 1 (cit. on pp. 3, 30, 34).

Tufte, Edward R, Nora Hillman Goeler, and Richard Benson (1990). *Envisioning information*. Vol. 2. Graphics press Cheshire, CT (cit. on p. 97).

Vlissides, John et al. (1995). "Design patterns: Elements of reusable object-oriented software". In: *Reading: Addison-Wesley* 49.120, p. 11 (cit. on pp. 82, 109).

Vogel, Peter (2010). *Practical code generation in. NET: covering Visual Studio 2005, 2008, and 2010*. Addison-Wesley Professional (cit. on p. 24).

Völter, Markus (2009). "MD* Best Practices". In: *Journal of Object Technology* 8, pp. 79–102 (cit. on pp. 20, 32).

Völter, Markus et al. (2013). *Model-driven software development: technology, engineering, management*. John Wiley & Sons (cit. on pp. 15, 21–23).

Waskom, Michael L (2021). "Seaborn: statistical data visualization". In: *Journal of Open Source Software* 6.60, p. 3021 (cit. on p. 97).

Whittle, Jon, John Hutchinson, and Mark Rouncefield (2014). “The state of practice in model-driven engineering”. In: *IEEE software* 31.3, pp. 79–85 (cit. on pp. 37, 44).

Whittle, Jon et al. (2013). “Industrial adoption of model-driven engineering: Are the tools really the problem?” In: *International Conference on Model Driven Engineering Languages and Systems*. Springer, pp. 1–17 (cit. on p. 39).

– (2017). “A taxonomy of tool-related issues affecting the adoption of model-driven engineering”. In: *Software & Systems Modeling* 16.2, pp. 313–331 (cit. on pp. 39–42).

Zettelkasten (2021). *Zettelkasten — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-June-2021]. URL: https://en.wikipedia.org/wiki/Zettelkasten (cit. on p. 129).

MASD

Model Assisted Software Development