# A Decade of D

@ funkwerk

# We Inform Passengers

- countrywide systems, deployed all over europe
- train, bus, tram, planes, ...
- automated, manual intervention trouble situations
- wide range of different interfaces
- multilingual
- announcements, displays, mobile, ...

https://www.rosenheim24.de/bayern/muenchen-gueter zug-entgleist-umschlagbahnhof-riem-stoerungen-bahn betrieb-9824593.html

12
11
12
r nach Cottbus -
weite
14:51
RE 1
Erkner - Fürstenwalde - Frankfurt (O)
Eisenhüttenstadt
A B C D E F G
Folgezüge:
15:14 RB 14 B-Schönefeld ✈
15:21 RE 1 Frankfurt (O)
DB
15:10
RB 21
A B C D E F G
B Friedrichstraße
11
Folgezüge:
15:22 +10 IC 143 von Amsterdam CS
15:35 +5 RE 2 Cottbus
Fahrkarten
Tickets/Billets/Biglietti
Fahrkarten
Tickets/Billets/Biglietti
G

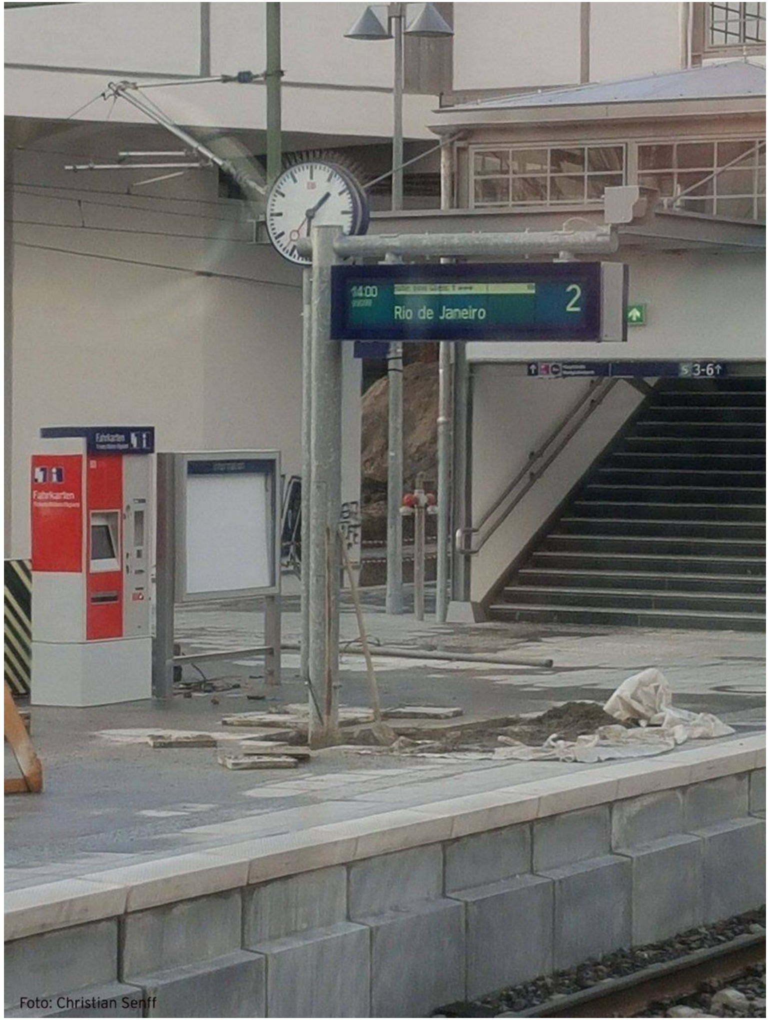
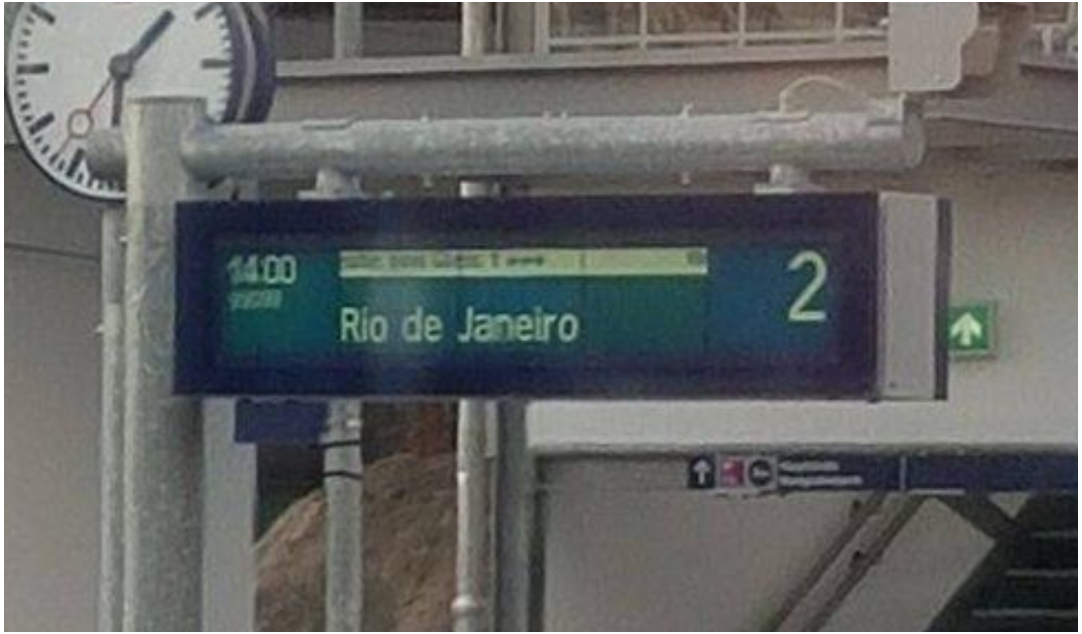


Foto: Christian Senff



# Quality Requirements

- highly reliable
- high level of customization
- maintainability
  - Test
  - Clean Code
  - Reviews

# Why D?

- neither C++ nor Java
- new language to break old habits
- run fast: native code
- modern, convenient
- `unittest` built-in

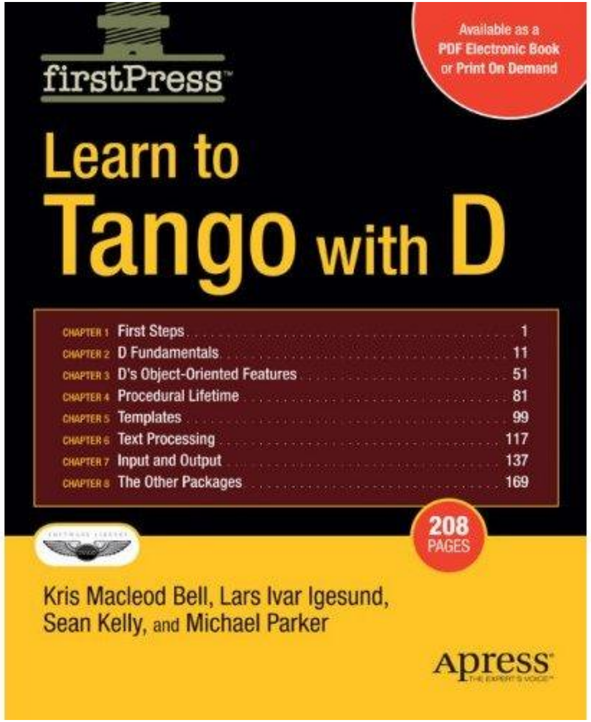
# Tango with D

## Pros

- fast XML parser
- logging
- network protocols
- familiar class library

## Cons

- not what later became
  “The D Programming Language”



Available as a PDF Electronic Book or Print On Demand

firstPress™

Learn to Tango with D

CHAPTER 1 First Steps . . . 1
CHAPTER 2 D Fundamentals . . . 11
CHAPTER 3 D's Object-Oriented Features . . . 51
CHAPTER 4 Procedural Lifetime . . . 81
CHAPTER 5 Templates . . . 99
CHAPTER 6 Text Processing . . . 117
CHAPTER 7 Input and Output . . . 137
CHAPTER 8 The Other Packages . . . 169

208 PAGES

Kris Macleod Bell, Lars Ivar Igesund,
Sean Kelly, and Michael Parker

Apress®
THE EXPERT'S VOICE™

# Short History

- 2008: First experiment in Tango with D
- 2009: Second experiment
- 2010: Tango: *“Tickets for the community”*
- 2010: Alexandrescu: *“The D Programming Language”*
- 2011: Alexandrescu: *“D1 to be discontinued on December 31, 2012”*
- 2012: SiegeLord: *“Tango for D2: All user modules ported”*
- 2012: Poor poll results for D at Funkwerk
- 2012: Porting to D2 and Phobos

# Short History

- 2013: DConf: *“Code Analysis for D with AnalyzeD”*
- 2015: GitHub: funkwerk
- 2016: Meetup: *Munich D Programmers*
- 2016: Add std.algorithm.iteration.cumulativeFold
- 2017: Greenfield Passenger Information System in D
- 2018: DConf: *“A Decade of D”*

# Effective D

- Prefer `foreach` loops to traditional `for` loops
- Use `std.algorithm` or `std.range` instead
- Take advantage of UFCS
  - `5.minutes`
  - function chaining
- Take advantage of UFCS where appropriate
  - don't: `“hello”.writeln`
  - don't: `“%s”.format(42)` like in Python  
    (thankfully it's `format!”%s”(42)` by now)
- ...

# Contract Programming

# Contract Programming

## `assert`

- evaluates expression
- if the value is false, `AssertError` is thrown
- language keyword
- for verifying the logic of the program
- in principle provable
- no run-time checks for `-release` version

## `enforce`

- evaluates expression
- if the value is false, `Exception` (Throwable) is thrown
- function template from `std.exception`
- for validating data

# Example: Theory and Practice

```
int div(int x, int y)
in
{
    assert(y != 0);
}
out(z)
{
    assert(x == y * z + x % y);
}
body
{
    ...
}
```

# DIP 1003: Remove body as a Keyword

```
int div(int x, int y)
in
{
    assert(y != 0);
}
out(z)
{
    assert(x == y * z + x % y);
}
do
{
    ...
}
```

# DIP 1009: Add Expression-Based Contract Syntax

```
int div(int x, int y)
    in(y != 0)
    out(z; x == y * z + x % y)
{
    ...
}
```

# Design by Contract

- Non-Redundancy principle
- Assertion Violation rule
- Reasonable Precondition principle
- Precondition Availability rule
- Assertion Evaluation rule
- Invariant rule

OBJECT-ORIENTED
SOFTWARE CONSTRUCTION
SECOND EDITION
CD-ROM INCLUDED
The Most Comprehensive, Definitive O-O Reference Ever Published
An O-O Tour de Force by a Pioneer in the Field
CD-ROM Includes Complete Hypertext Version of Book AND Object-Oriented Development Environment
Winner SOFTWARE Development PRODUCT EXCELLENCE AWARD 1997
BERTRAND MEYER

# Design by Contract

- Non-Redundancy principle
  - Under no circumstances shall the body of a routine ever test for the routine's precondition.
- Assertion Violation rule
- Reasonable Precondition principle
- Precondition Availability rule
- Assertion Evaluation rule
- Invariant rule

# Design by Contract

- Non-Redundancy principle
- Assertion Violation rule
  - A run-time assertion violation is the manifestation of a bug in the software.
  - A precondition violation is the manifestation of a bug in the client.
  - A postcondition violation is the manifestation of a bug in the supplier.
- Reasonable Precondition principle
- Precondition Availability rule
- Assertion Evaluation rule
- Invariant rule

# Design by Contract

- Non-Redundancy principle
- Assertion Violation rule
- Reasonable Precondition principle
    - Every routine precondition must satisfy the following requirements:
        - The precondition appears in the official documentation distributed to authors of client modules.
        - It is possible to justify the need for the precondition in terms of the specification only.
- Precondition Availability rule
- Assertion Evaluation rule
- Invariant rule

# Design by Contract

- Non-Redundancy principle
- Assertion Violation rule
- Reasonable Precondition principle
- Precondition Availability rule
  - Every feature appearing in the precondition of a routine
    must be available to every client to which the routine is available.
- Assertion Evaluation rule
- Invariant rule

# Design by Contract

- Non-Redundancy principle
- Assertion Violation rule
- Reasonable Precondition principle
- Precondition Availability rule
- Assertion Evaluation rule
  - During the process of evaluating an assertion at run-time,
    routine calls shall be executed without any evaluation of the associated assertions.
- Invariant rule

# Design by Contract

- Non-Redundancy principle
- Assertion Violation rule
- Reasonable Precondition principle
- Precondition Availability rule
- Assertion Evaluation rule
- Invariant rule
  - An assertion *I* is a correct class invariant for a class *C*
    if and only if it meets the following two conditions:
    - Every creation procedure of *C*, when applied to arguments satisfying its precondition in a state where the attributes have their default values, yields a state satisfying *I*.
    - Every exported routine of the class, when applied to arguments and a state satisfying both *I* and the routine's precondition, yields a state satisfying *I*.

# Subcontracting

- Parents’ Invariant rule
- Assertion Redeclaration rule

OBJECT-ORIENTED
SOFTWARE CONSTRUCTION
SECOND EDITION
CD-ROM INCLUDED
The Most Comprehensive, Definitive O-O Reference Ever Published
An O-O Tour de Force by a Pioneer in the Field
CD-ROM Includes Complete Hypertext Version of Book AND Object-Oriented Development Environment
Winner
SOFTWARE Development
PRODUCT EXCELLENCE AWARD 1997
BERTRAND MEYER

# Subcontracting

- Parents’ Invariant rule
  - The invariants of all the parents of a class apply to the class itself.
- Assertion Redeclaration rule

# Subcontracting

- Parents’ Invariant rule
- Assertion Redeclaration rule
  - A routine redeclaration may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger.

# In, Out and Inheritance

```
interface I
{
    int foo(int x)
        in(x != 0)
        out(y; y != 0);
}

class C : I
{
    override int foo(int x)
        in(false)
        out(; true)
    {
        ...
    }
}
```

# In, Out and Inheritance

```
interface I
{
    int foo(int x)
        in(x != 0)
        out(y; y != 0);
}

class C : I
{
    override int foo(int x)
        in(x != 0)
        out(y; y != 0)
    {
        ...
    }
}
```

# Contract Programming

## Pros

- “null safety” instead of segmentation faults
- clear statement what is required and ensured
- clear statement who is to blame
- living documentation

## Cons

- often misused as wish machine
- gaps between `synchronized` `in`, `out`, and `body`
- Issue 15984 - [REG2.071]
  Interface contracts retrieve garbage instead of parameters

# Unit Testing

# Theoretical Unit Testing

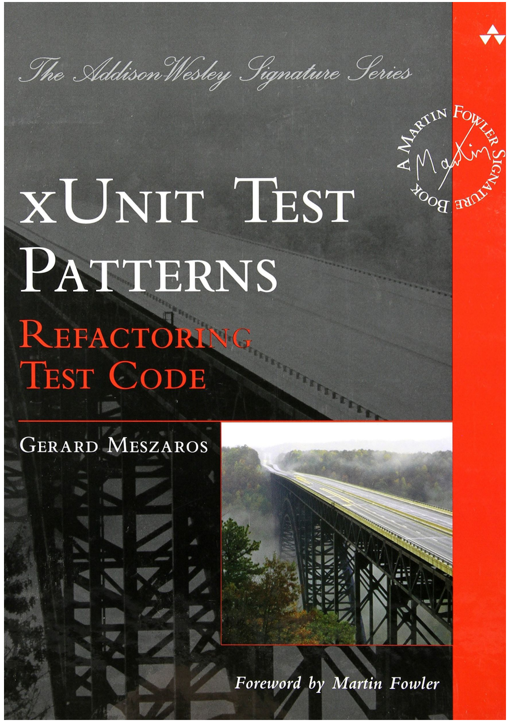
```
int div(int x, int y)
    in(y != 0)
    out(z; x == y * z + x % y)
{
    ...
}

// Don’t Try This at Home
unittest
{
    div(5, 2);
    div(-5, 2);
    div(5, -2);
    div(-5, -2);
}
```

# xUnit Testing Framework

How to get as much information as possible out of a failed test run?

GitHub: linkrope/dunit

- replacement of Dunit (for D1)
- forked from GitHub: jmcabo/dunit
- user-defined attributes `@Test`, ...
- by now, based on latest version JUnit 5


The Addison Wesley Signature Series
A Martin Fowler Signature Book
xUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
Foreword by Martin Fowler

# Example: Testcase Class

```
class TrainTest
{
    mixin UnitTest;

    @BeforeEach
    void setUp() ...

    @Test
    void testCase1() ...

    @Test
    void testCase2() ...

    @AfterEach
    void tearDown() ...
}
```

# xUnit Testing Framework for D

## Pros

- tests are organized in classes
- tests are always named
- tests can reuse a shared fixture
- all failed tests are shown at once
- more information about failures
- progress indication
- XML test report in `JUnitReport` format

## Cons

- `mixin UnitTest;` is mandatory

# Sentence Style for Naming Unit Tests

```
class TrainTest
{
    mixin UnitTest;

    @BeforeEach
    void setUp() ...

    @Test
    void canBeDelayed() ...

    @Test
    void canBeCanceled() ...

    @AfterEach
    void tearDown() ...
}
```

# @DisplayName…

```
@("train can be delayed")
unittest
{
    ...
}

@("train can be canceled")
unittest
{
    ...
}
```

# Pulling the Fixture into the `unittest`

```
unittest
{
    with (Fixture())
    {
        ...
    }
}

struct Fixture
{
    static Fixture opCall() ...  // set up

    ~this() ...  // tear down
}
```

# Test Execution

GitHub: atilaneves/unit-threaded

- tests can be named
- tests can be run selectively
- tests can be run in parallel
- subset of the features is compatible with built-in `unittest`

# Expectations

- `assert`
- `static assert`
- `assertEquals`
- Fluent Assertions

# Expectations

- `assert`
  - `assert(answer == 42);`
  - `core.exception.AssertError@test.d(5): unittest failure`
- `static assert`
- `assertEquals`
- Fluent Assertions

# Expectations

- assert
- static assert
  - `static assert(answer == 42);`
  - `test.d(5): Error: static assert:  54 == 42 is false`

- assertEquals
- Fluent Assertions

# Expectations

- `assert`
- `static assert`
- `assertEquals`
    - `assertEquals(42, answer);`
    - `dunit.assertion.AssertException@test.d(5): expected: <42> but was: <54>`

- Fluent Assertions

# Expectations

- `assert`
- `static assert`
- `assertEquals`
    - `assertEquals(42, answer);`
    - `dunit.assertion.AssertException@test.d(5): expected: <42> but was: <54>`

    - `answer.assertEquals(42);`
    - `dunit.assertion.AssertException@test.d(5): expected: <54> but was: <42>`

- Fluent Assertions

# Expectations

- `assert`
- `static assert`
- `assertEquals`
- Fluent Assertions
    - `answer.should.equal(42);`
    - TBD

# Mock Object Framework

GitHub: funkwerk/dmocks

- forked from GitHub: QAston/DMocks-revived
- reactivation of DMocks

# Code Coverage

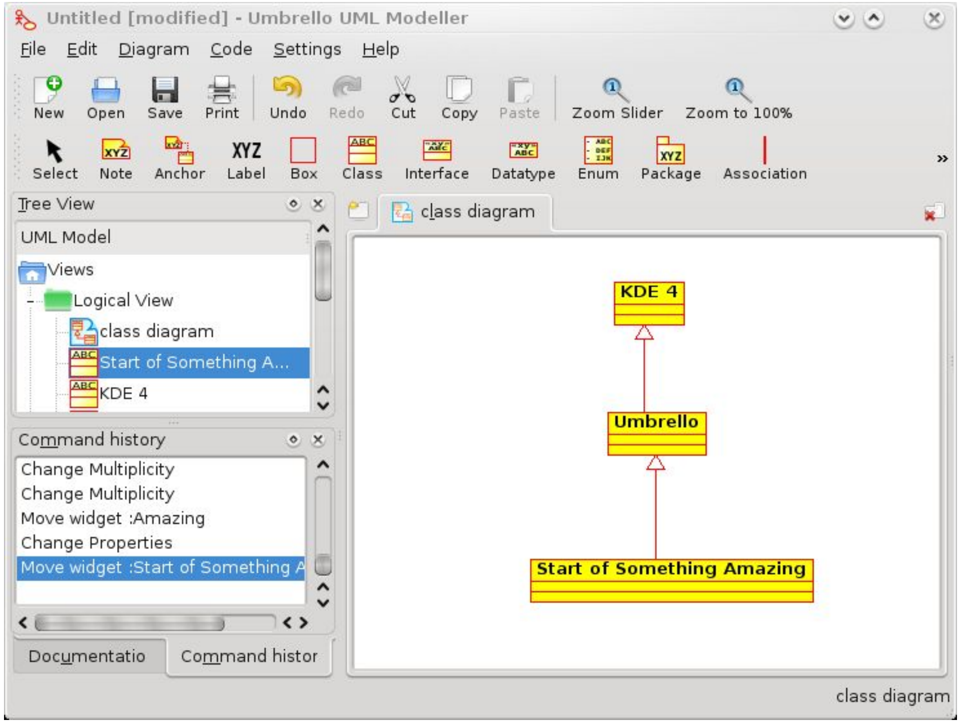
We use separate `src` and `unittest` directories.

GitHub: ohdatboi/covered

- shows coverage result per file
- shows average coverage
- moves `*.lst` files out of the way

# Architecture and Design

# Umbrello UML Modeller



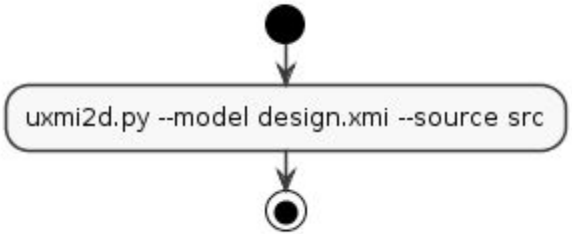
Change Multiplicity
Change Multiplicity
Move widget :Amazing
Change Properties
Move widget :Start of Something A

*Umbrello UML Modeller 2 supports ActionScript, Ada, C++, C#, D, IDL, Java™, Javascript, MySQL, and Pascal source code.*

# UML to D

## uxmi2d

- forward engineering from class diagrams to D skeleton code
- tries to keep existing code
- for Umbrello’s XMI

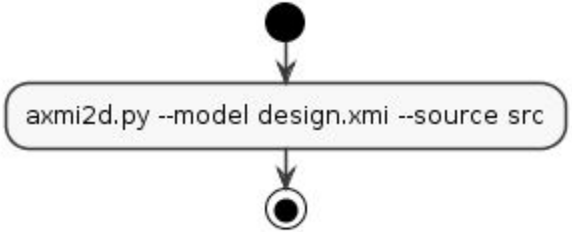


```
uxmi2d.py --model design.xmi --source src
```

# UML to D

## axmi2d

- forward engineering from class diagrams to D skeleton code
- tries to keep existing code
- for ArgoUML’s XMI



```
axmi2d.py --model design.xmi --source src
```

# UML to D

## Pros

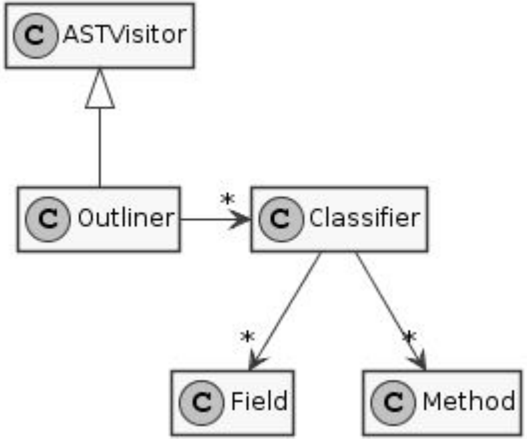
- living documentation
- generation of getters and setters
- documentation comments for contracts
- enforced style
    - one class per file
    - fields first, then member functions
    - (alphabetical) order of attributes

## Cons

- refactoring with a drawing tool sucks

# PlantUML

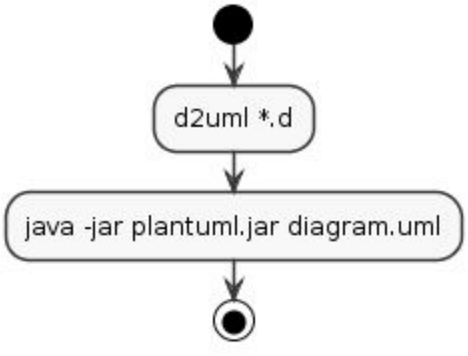
```
ASTVisitor <|-- Outliner
Outliner -> "*" Classifier
Classifier --> "*" Field
Classifier --> "*" Method
```



# D to UML

GitHub: funkwerk/d2uml

- reverse engineering
  from D source code
  to PlantUML class outlines

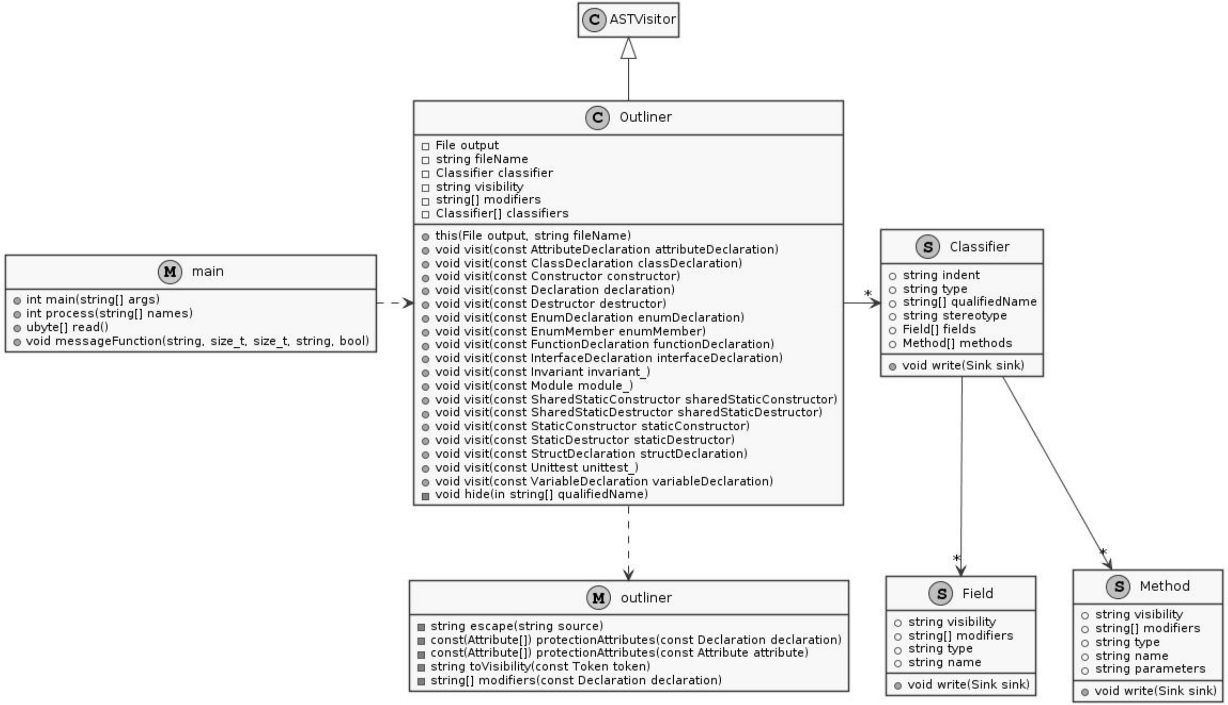


d2uml *.d

java -jar plantuml.jar diagram.uml

# Example: Self-Portrait

```
!include classes.plantuml

main .> Outliner
ASTVisitor <|-- Outliner
Outliner -> "*" Classifier
Classifier --> "*" Field
Classifier --> "*" Method
Outliner ..> outliner
```



C ASTVisitor

C Outliner

□ File output
□ string fileName
□ Classifier classifier
□ string visibility
□ string[] modifiers
□ Classifier[] classifiers

● this(File output, string fileName)
● void visit(const AttributeDeclaration attributeDeclaration)
● void visit(const ClassDeclaration classDeclaration)
● void visit(const Constructor constructor)
● void visit(const Declaration declaration)
● void visit(const Destructor destructor)
● void visit(const EnumDeclaration enumDeclaration)
● void visit(const EnumMember enumMember)
● void visit(const FunctionDeclaration functionDeclaration)
● void visit(const InterfaceDeclaration interfaceDeclaration)
● void visit(const Invariant invariant_)
● void visit(const Module module_)
● void visit(const SharedStaticConstructor sharedStaticConstructor)
● void visit(const SharedStaticDestructor sharedStaticDestructor)
● void visit(const StaticConstructor staticConstructor)
● void visit(const StaticDestructor staticDestructor)
● void visit(const StructDeclaration structDeclaration)
● void visit(const Unittest unittest_)
● void visit(const VariableDeclaration variableDeclaration)
■ void hide(in string[] qualifiedName)

M main

● int main(string[] args)
● int process(string[] names)
● ubyte[] read()
● void messageFunction(string, size_t, size_t, string, bool)

S Classifier

○ string indent
○ string type
○ string[] qualifiedName
○ string stereotype
○ Field[] fields
○ Method[] methods

● void write(Sink sink)

M outliner

■ string escape(string source)
■ const(Attribute[]) protectionAttributes(const Declaration declaration)
■ const(Attribute[]) protectionAttributes(const Attribute attribute)
■ string toVisibility(const Token token)
■ string[] modifiers(const Declaration declaration)

S Field

○ string visibility
○ string[] modifiers
○ string type
○ string name

● void write(Sink sink)

S Method

○ string visibility
○ string[] modifiers
○ string type
○ string name
○ string parameters

● void write(Sink sink)

# D to UML

## Pros

- living documentation
- easy retrofitting

## Cons

- no support for relationships between classes
  (good arrangement is essential for creating effective diagrams)
- no code generation

# Generate Getters, Setters

GitHub: funkwerk/accessors

```
import accessors;

class C
{
    @Read
    @Write
    private int bar_;

    mixin(GenerateFieldAccessors);
}
```

# Generate Getters, Setters

GitHub: funkwerk/accessors

DEPRECATED

```
import accessors;

class C
{
    @Read
    @Write
    private int bar_;

    mixin(GenerateFieldAccessors);
}
```

# Generate Getters, Setters and Everything

GitHub: funkwerk/boilerplate

```
import boilerplate;

class C
{
    ...

    mixin(GenerateFieldAccessors);
    mixin(GenerateInvariants);
    mixin(GenerateThis);
    mixin(GenerateToString);
}
```

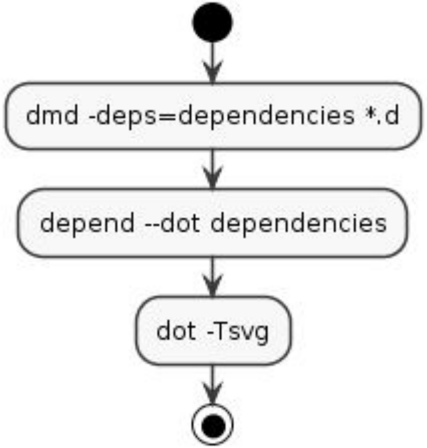
# Dependency Tool

*“The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition.”* (Big Ball of Mud)

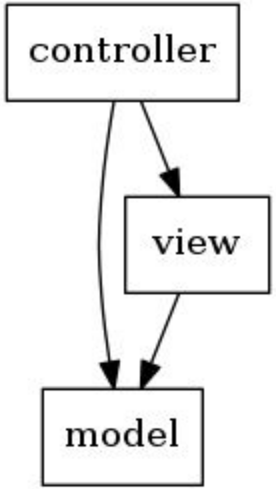
GitHub: funkwerk/depend

- visualizes `import` dependencies
- checks actual `import` dependencies
  against a UML model of target dependencies
- considers module or package dependencies

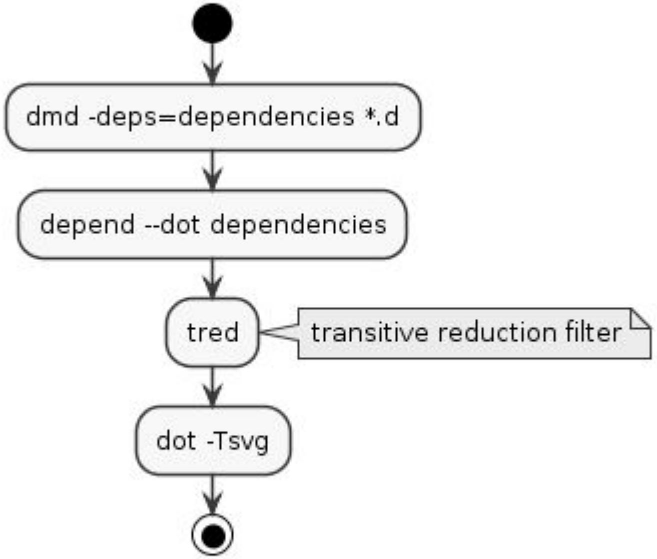
# depend: Visualize Dependencies



```
dmd -deps=dependencies *.d
depend --dot dependencies
dot -Tsvg
```



# depend: Visualize Dependencies



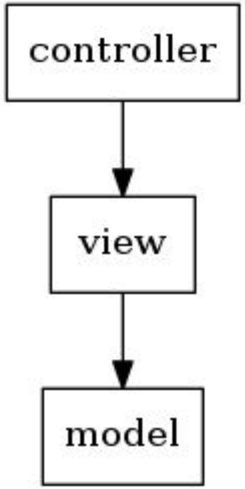
dmd -deps=dependencies *.d

depend --dot dependencies

tred

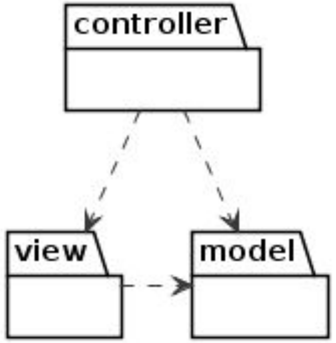
transitive reduction filter

dot -Tsvg

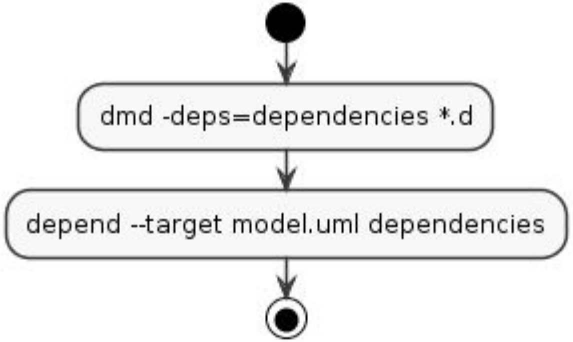


# Example: model-view-controller

```
package model {}
package view {}
package controller {}

controller ..> view
controller ..> model
view .> model
```



# depend: Check Dependencies



dmd -deps=dependencies *.d

depend --target model.uml dependencies

```
error: unintended dependency controller.controller -> model.model
error: unintended dependency controller.controller -> view.view
error: unintended dependency view.view -> model.model
```

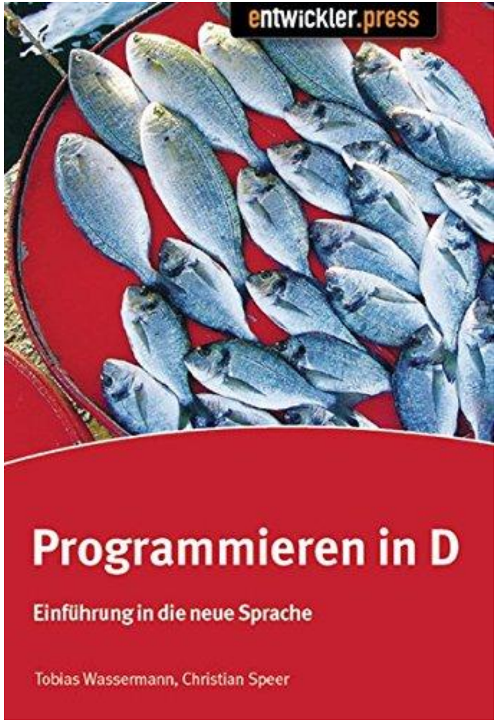
# Summary

## Code For The Maintainer

- Use Contract Programming
- Write Helpful Unit Tests
- Safeguard the Structure

# A Decade of D

in Germany



entwickler.press

Programmieren in D

Einführung in die neue Sprache

Tobias Wassermann, Christian Speer

One more thing...