# DSL Design with Python

## Using TTRPG Examples

S.Lott

https://fosstodon.org/@slott56
https://github.com/slott56

1-Nov-2025

# Link to Slide Deck Repository

https://github.com/slott56/dsl-with-python

# Table of Contents

# You have a Problem...

> *Before a small brick building surrounded by forest.*
> *A stream flows out of the building and down a gully.*

Descend to find a door with a giant lock and two keys with labels.

Above the door is this quote:

> *Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.*

The labels:

1. "RE's are bad; RE $\subseteq$ DSL $\therefore \forall$ DSL's are bad. $\blacksquare$"
2. "They always lie."

# The "RE is two problems" fallacy

RE syntax is an ugly jumble of infix and postfix operators.

**Overuse** of regular expressions can be bad.
(The famous Jamie Zawinski quote was about PERL.)

You know: "When your only tool is a hammer. . ."

**Benefit: a DSL decomposes a problem.**

1. The Domain-Specific Language. . .
2. Used to solve the problem.

# Are there better RE's?

## A digression

Consider Al Sweigart's HUMRE.

It has all the regular expression features.
In **Python** syntax.

HUMRE is a DSL for regular expressions in Python syntax.

# Are there better RE's?

## A digression

Consider Al Sweigart's HUMRE.

It has all the regular expression features.
In **Python** syntax.

HUMRE is a DSL for regular expressions in Python syntax.

*And. Bonus. It supports my thesis.*

# Occult DSL's

You may have designed one of these DSL's.

| **DSL** | **Syntax** |
|---|---|
| Configuration file | TOML or INI |
| CLI | Shell syntax |
| RESTful web service | HTTP plus JSON |

Does the underlying object model count as a DSL?

# Occult DSL's

You may have designed one of these DSL's.

| **DSL** | **Syntax** |
|---|---|
| Configuration file | TOML or INI |
| CLI | Shell syntax |
| RESTful web service | HTTP plus JSON |

Does the underlying object model count as a DSL?

*Stick around and see where we wind up.*

# Section 2

# DSL Design

# Behind the entrance door is a dark hallway.

The hallway ends at a door, covered in images of boxes and arrows. A sign on the door says **Model**.

On your left, you find a door labeled **Syntax**.

# The **Syntax** workroom

**Door 1**
Two work tables:

- **Tokenizer** tools and libraries and modules.
- **Parser** more tools and libraries.

Books by **DABEAZ** everywhere.

A pile of parts labeled **Lark**.

**Door 2**
A shelf with packages:

- JSON
- YAML
- TOML
- HUML

**Archway**
Flanked by two friendly-looking snakes.

*[Pythons? Roll an* Intellect *check.]*

You entered turning left through the **Syntax** door. . .

The **Model** door is still ahead of you!

# Why the Python door?

Avoid writing yet another parser.

## Two *whattabout* gremlins appear

- ***Whattabout* security? The DSL is Code!**

# Why the Python door?

Avoid writing yet another parser.

## Two *whattabout* gremlins appear

- *Whattabout* security? The DSL is Code!

  A chaotic evil 9th level Sorcerer won't waste time hacking
  `rot13("vzcbeg bf; bf.flfgrz('sbezng p:')")`
  into DSL-based content.
  The **whole app** is visible, hackable Python.
  Consider a side-quest to query the Sphinx.

# Why the Python door?

Avoid writing yet another parser.

## Two *whattabout* gremlins appear

- *Whattabout* security? The DSL is Code!

  A chaotic evil 9th level Sorcerer won't waste time hacking
  `rot13("vzcbeg bf; bf.flfgrz('sbezng p:')")`
  into DSL-based content.
  The **whole app** is visible, hackable Python.
  Consider a side-quest to query the Sphinx.
- ***Whattabout* the really ugly data model?**

# Why the Python door?

Avoid writing yet another parser.

## Two *whattabout* gremlins appear

- *Whattabout* security? The DSL is Code!

  A chaotic evil 9th level Sorcerer won't waste time hacking
  `rot13("vzcbeg bf; bf.flfgrz('sbezng p:')")`
  into DSL-based content.
  The **whole app** is visible, hackable Python.
  Consider a side-quest to query the Sphinx.
- *Whattabout* the really ugly data model?

  Wait for case study 2.

# Section 3

# Case Study 1 — TTRPG “dice expressions”

# TTRPG “dice expressions”

**Example**

| | |
|---|---|
| `3d6+2` | “roll 3 six-sided dice, add 2” |

**Example**

| | |
|---|---|
| `4d8` | “roll 4 eight-sided dice” |

There’s more, but that’s for lower levels of the dungeon.

Which path?

- Turn left and write RE’s for the syntax?
- Define Python classes and objects?

# TTRPG "dice expressions"

**Example**

| | |
|---|---|
| `3d6+2` | "roll 3 six-sided dice, add 2" |

**Example**

| | |
|---|---|
| `4d8` | "roll 4 eight-sided dice" |

There's more, but that's for lower levels of the dungeon.

Which path?

- Turn left and write RE's for the syntax?
- Define Python classes and objects?

The syntax isn't Pythonic.

# How to proceed?

## Strategy Tip: Adjust the syntax to be Pythonic

From this: $3d6 + 2$.

To this: `3 * D6 + 2`

# How to proceed?

## Strategy Tip: Adjust the syntax to be Pythonic

From this: $3d6 + 2$.

To this: `3 * D6 + 2`

Python syntax lets you teleport straight to the data model.

# How to proceed?

## Strategy Tip: Adjust the syntax to be Pythonic

From this: $3d6 + 2$.

To this: `3 * D6 + 2`

Python syntax lets you teleport straight to the data model.

## A *whattabout* gremlin appears

But *whattabout* the users?

It's a *. *One character.*
If it helps them solve their problem, they'll embrace it.

# The Die class

```
class Die:
    def __init__(self, faces: int = 6, n: int = 1) -> None:
        ...
    def __rmul__(self, other: Any) -> Die:
        match other:
            case int():
                return Die(self.faces, n * other)
            case _:
                return NotImplemented
    def roll(self) -> int:
        ...

D6 = D(6)
```

This is the data model.

# The Die class

```
class Die:
    def __init__(self, faces: int = 6, n: int = 1) -> None:
        ...
    def __rmul__(self, other: Any) -> Die:
        match other:
            case int():
                return Die(self.faces, n * other)
            case _:
                return NotImplemented
    def roll(self) -> int:
        ...

D6 = D(6)
```

This is the data model. And, `3*D6` is valid DSL.

# Section 4

# Case Study 2 — **OpenD6** spells

# Context — legacy data

**OpenD6** rules have an Open Gaming License (OGL), making them open source with attribution.
Perfect for extensions and customizations.

I wanted **all** the magical spell definitions.

# Context — legacy data

**OpenD6** rules have an Open Gaming License (OGL), making them open source with attribution.
Perfect for extensions and customizations.

I wanted **all** the magical spell definitions.

Which are in PDF's.

# Context — legacy data

**OpenD6** rules have an Open Gaming License (OGL), making them open source with attribution.
Perfect for extensions and customizations.

I wanted **all** the magical spell definitions.

Which are in PDF's. Scanned from a printed copy.

# Context — legacy data

**OpenD6** rules have an Open Gaming License (OGL), making them open source with attribution.
Perfect for extensions and customizations.

I wanted **all** the magical spell definitions.

Which are in PDF's. Scanned from a printed copy.

How to capture legacy data?

# Context — legacy data

**OpenD6** rules have an Open Gaming License (OGL), making them open source with attribution.
Perfect for extensions and customizations.

I wanted **all** the magical spell definitions.

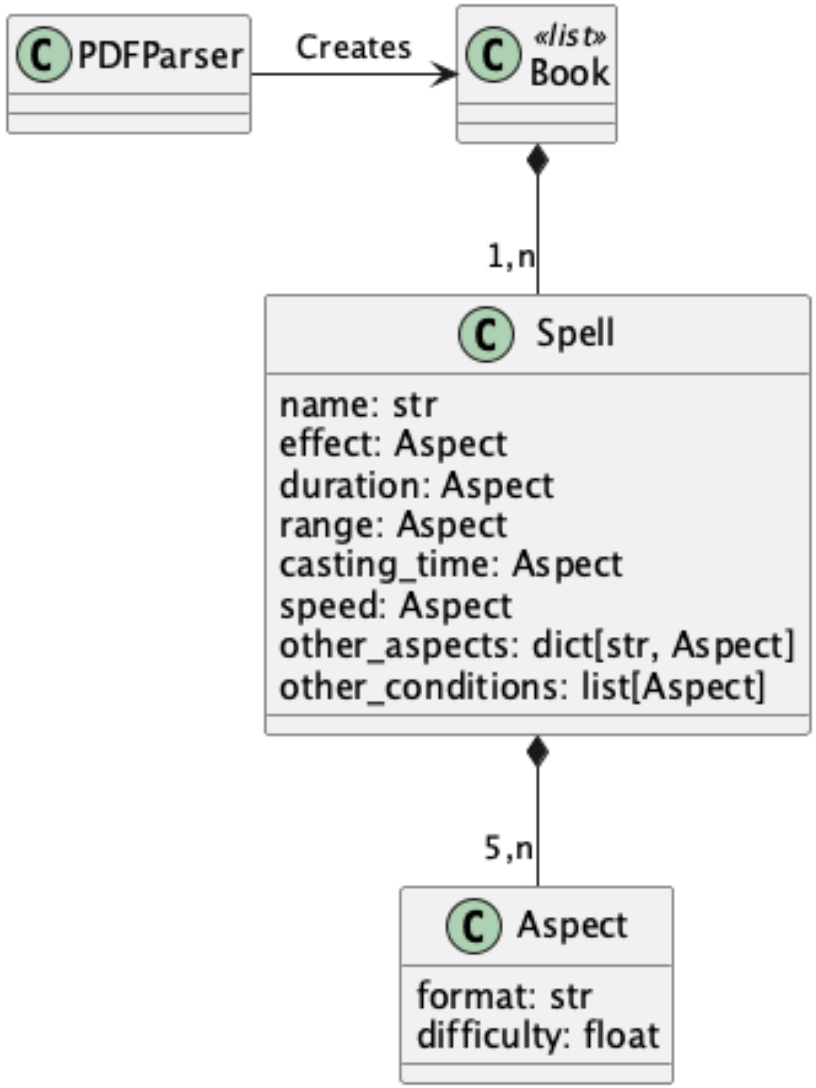
Which are in PDF's. Scanned from a printed copy.

How to capture legacy data?

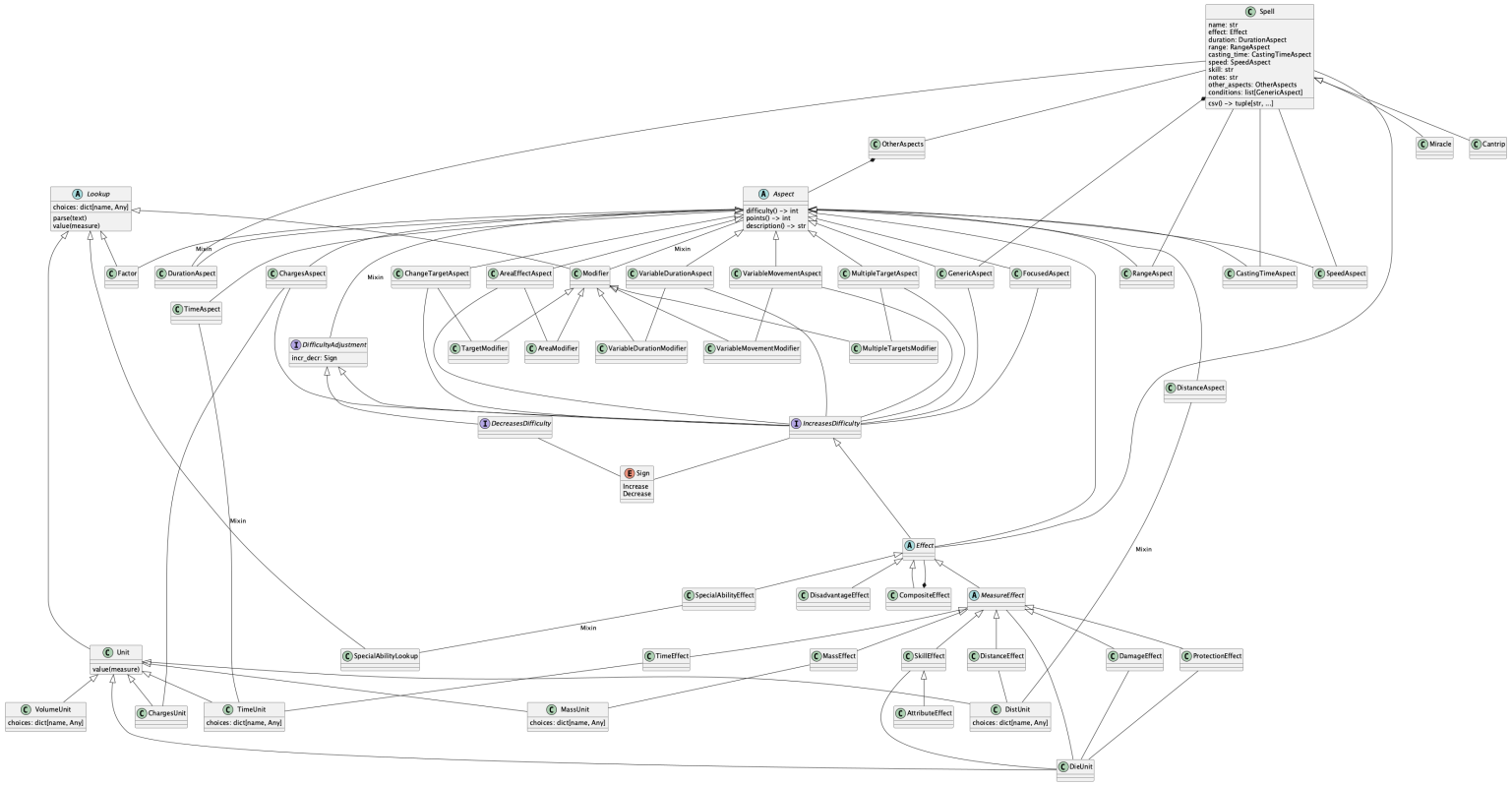
## Strategy Tip: Q&D (Quick & Dirty) Model

1. Define a throw-away data structure to capture legacy content.
2. Then, work out a more useful, semantically complete DSL.

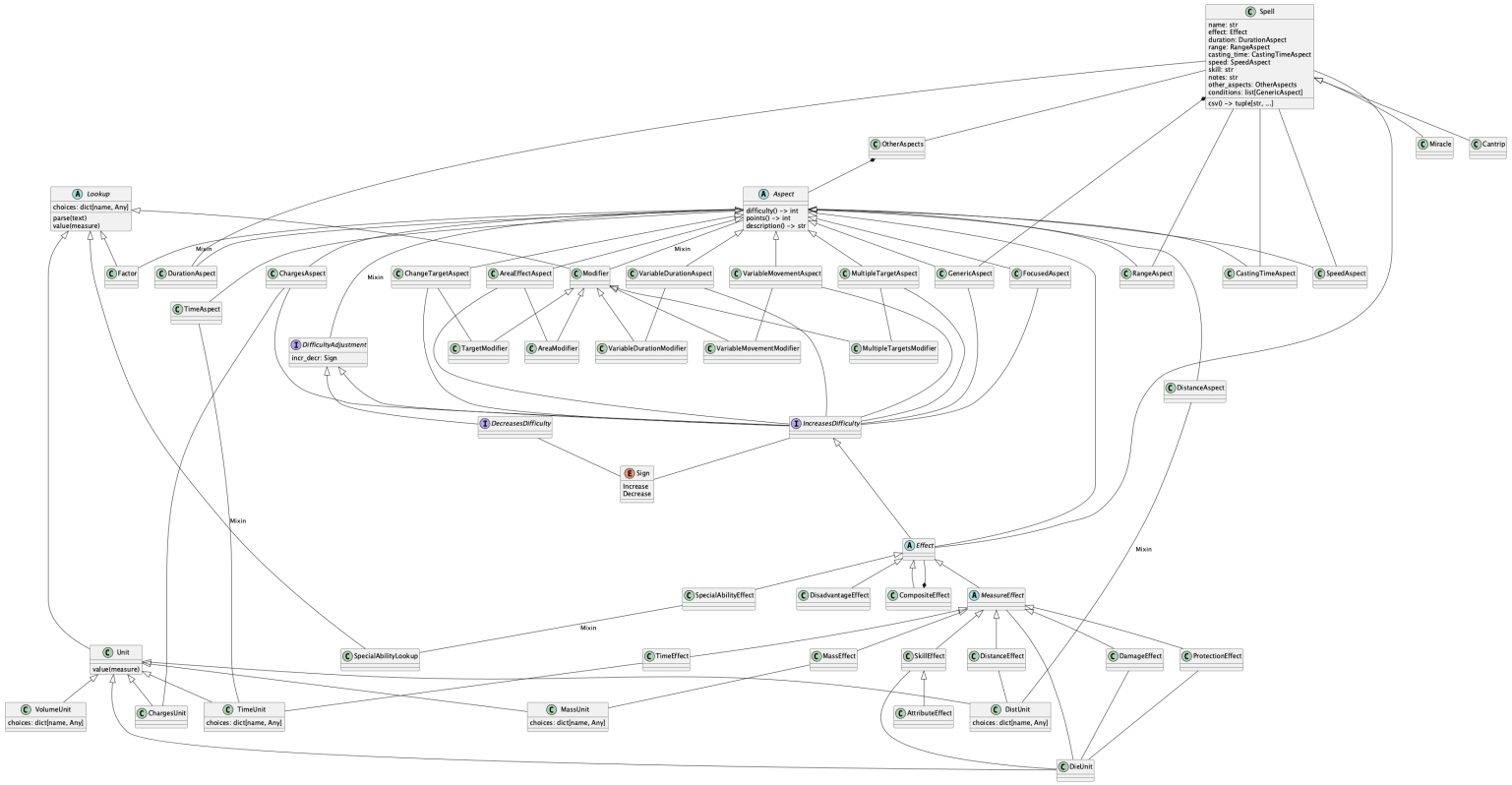
# Legacy Data Model



# Expanded DSL Data Model

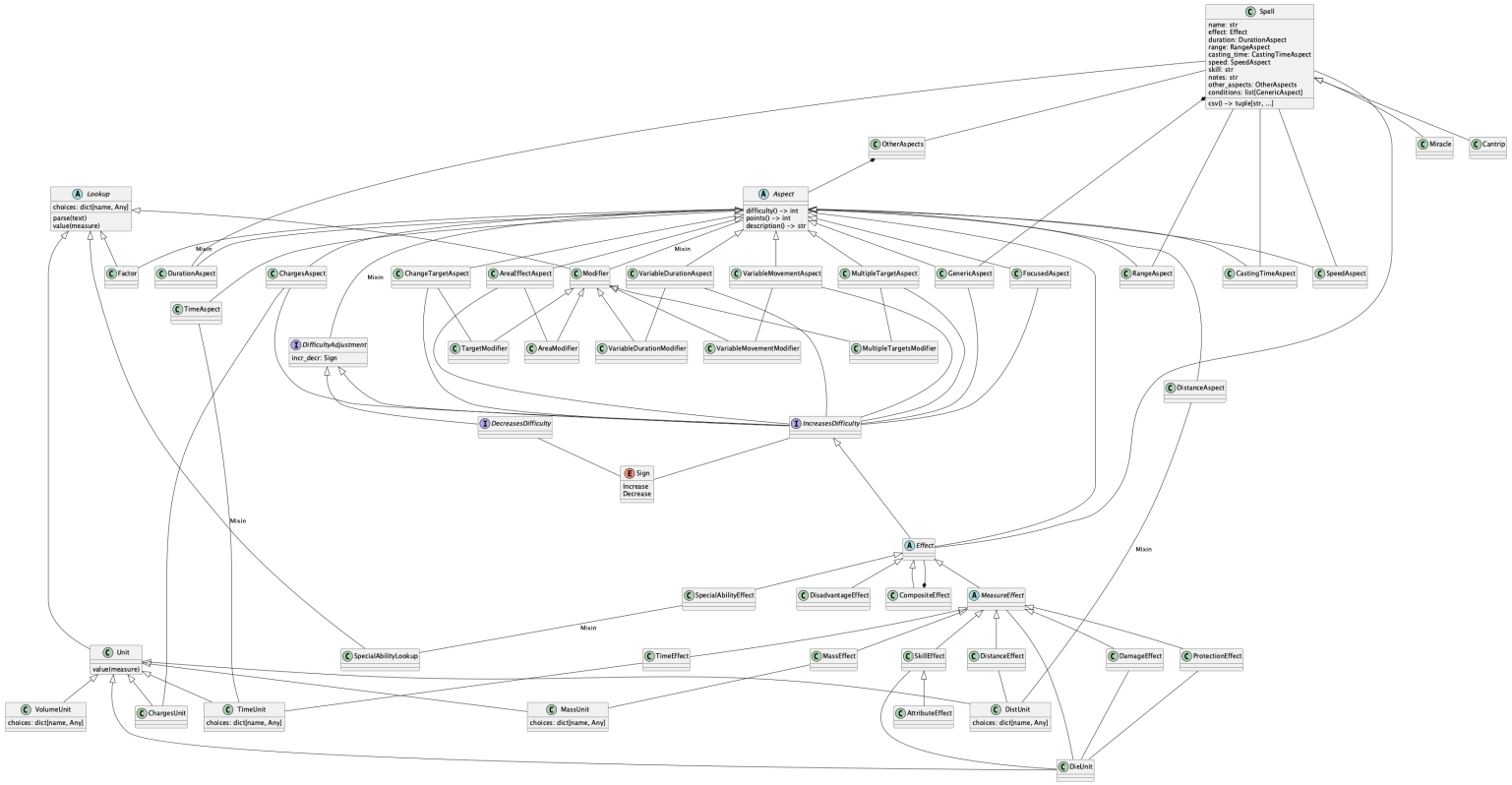


# Expanded DSL Data Model




Yes, it's complicated.

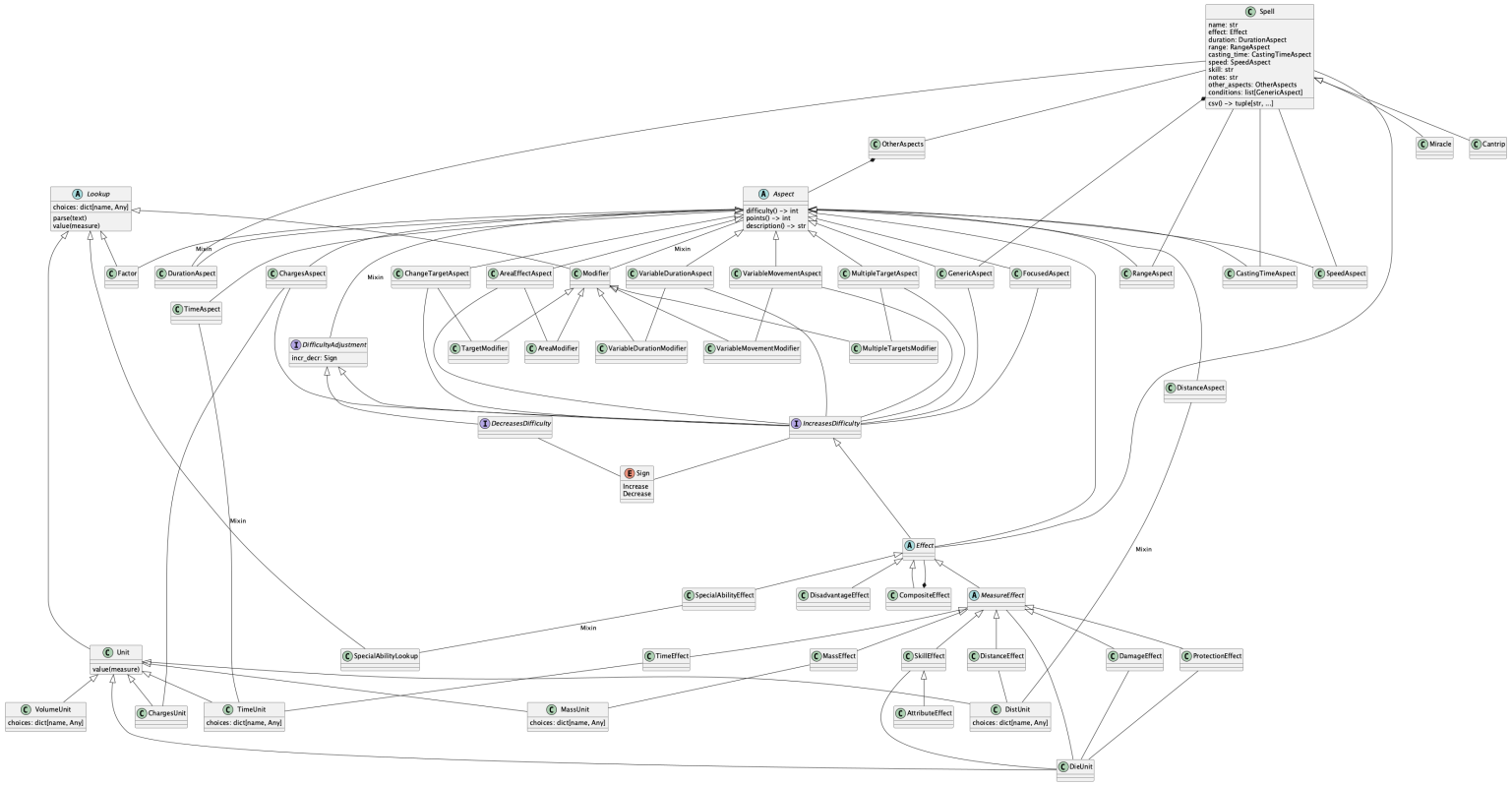
# Expanded DSL Data Model




Yes, it's complicated.

A (second) bulk conversion was required.

# Expanded DSL Data Model




Yes, it's complicated.

A (second) bulk conversion was required.

I omitted some stuff, and it was still vast.

# Example DSL statement

```
Spell(
    name="Push",
    skill="Apportation",
    notes="Even the most subtle of things,,,",
    effect=TimeEffect("sends a small object into the future", "10 min"),
    duration=DurationAspect(measure="10 minutes"),
    range=RangeAspect(measure="touch"),
    casting_time=CastingTimeAspect(measure="1 round"),
    speed=SpeedAspect.based_on(("range",), ""),
    other_aspects={
        "gestures": GesturesAspect("Wave one hand ...", "simple"),
        "incantations": IncantationsAspect(
           "Where did it go?", "sentence"
        ),
    },
)
```

It has a lot of boilerplate. The API design needs work.

# Typical DSL use cases

- Represent legacy content and rules in the DSL.
- Compute derived values from DSL.
  (Compare with legacy sources as acceptance test case.)
- Present content in RST format for publication.
- Identify special cases and possible errors.

# Typical DSL use cases

- Represent legacy content and rules in the DSL.
- Compute derived values from DSL.
  (Compare with legacy sources as acceptance test case.)
- Present content in RST format for publication.
- **Identify special cases and possible errors.**

# Typical DSL use cases

- Represent legacy content and rules in the DSL.
- Compute derived values from DSL.
  (Compare with legacy sources as acceptance test case.)
- Present content in RST format for publication.
- Identify special cases and possible errors.

## DSL and Learning

# Typical DSL use cases

- Represent legacy content and rules in the DSL.
- Compute derived values from DSL.
  (Compare with legacy sources as acceptance test case.)
- Present content in RST format for publication.
- Identify special cases and possible errors.

## DSL and Learning

- Unless you're already the leading expert in the problem domain, expect the DSL evolution.

# Typical DSL use cases

- Represent legacy content and rules in the DSL.
- Compute derived values from DSL.
  (Compare with legacy sources as acceptance test case.)
- Present content in RST format for publication.
- Identify special cases and possible errors.

## DSL and Learning

- Unless you're already the leading expert in the problem domain, expect the DSL evolution.
- DSL evolution $\equiv$ Learning
  The SOLID **Open/Closed Principle** is your friend.

# But *whattabout* the ugly data model?

## Strategy Tip: Replace tears with tiers

**Foundation** **Your ugly domain model that really captures everything the end users are prattling on about.**

# But *whattabout* the ugly data model?

## Strategy Tip: Replace tears with tiers

**Foundation** Your ugly domain model that really captures **everything** the end users are prattling on about.

**API** **A layer on top of the ugly foundation that creates the classes and objects for a pleasant, easier-to-use DSL.**

# But *whattabout* the ugly data model?

## Strategy Tip: Replace tears with tiers

Foundation Your ugly domain model that really captures **everything** the end users are prattling on about.

API A layer on top of the ugly foundation that creates the classes and objects for a pleasant, easier-to-use DSL.

## This is Pythonic OO design.

Design patterns to use:

- Façade [koan 5]: Flat is better than nested
- Adapter [koan 8]: Special cases aren't special enough to break the rules

Want more ideas? Run `import this` for all the koans.

# Section 5

# Conclusion

# Tips for DSL design

- Use Python as the syntax for a DSL.

**Pursue Pythonic Practices**

# Tips for DSL design

- Use Python as the syntax for a DSL.
- Capture legacy rules with a Q&D model.

**Pursue Pythonic Practices**

# Tips for DSL design

- Use Python as the syntax for a DSL.
- Capture legacy rules with a Q&D model.
- As you learn, build a complete DSL model.

**Pursue Pythonic Practices**

# Tips for DSL design

- Use Python as the syntax for a DSL.
- Capture legacy rules with a Q&D model.
- As you learn, build a complete DSL model.
- Tears $\rightarrow$ tiers — layer the model for usability.

**Pursue Pythonic Practices**

# Link to Slide Deck Repository

https://github.com/slott56/dsl-with-python

# Appendix

## Malicious Actors

# How to handle malicious actors



## Are **all** elements present?

- Untrusted sources of DSL,
- And large DSL docs,
- And difficulty in doing code validation.

Two strategies:

1. Use JSON or TOML.
   Use a `@classmethod` to build DSL model objects from `dict[str, Any]` content.
2. Don't allow DSL modules to use `import`.
   Use the **Compile-Check-Exec** pattern.

# Avoid `import` with Compile-Check-Exec

## Compile

```
source = Path(module).with_suffix(".py").read_text()
spell_book_code = ast.parse(source, module, "exec")
```

## Check

```
visitor = AllImports(source)
visitor.visit(spell_book_code)
if visitor.imports:
    raise ValueError(f"import detected: {visitor.imports}")
```

## Exec

```
global_defs: dict[str, Any] = {}
local_vars: dict[str, Any] = {}
exec("from magic2 import *", global_defs, local_vars)
exec(source, global_defs, local_vars)
return local_vars["spells"]
```

# The AllImports visitor



```
class AllImports(ast.NodeVisitor):
    def __init__(self, source: str) -> None:
        self.source = source
        self.imports: list[str | None] = []

    def visit_Import(self, node: ast.Import) -> None:
        self.imports.append(ast.get_source_segment(self.source, node))

    def visit_ImportFrom(self, node: ast.ImportFrom) -> None:
        self.imports.append(ast.get_source_segment(self.source, node))
```