DSL Design with Python

Using TTRPG Examples

S.Lott

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

1-Nov-2025

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.*

You 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:

- "RE's are bad; RE $\subseteq$ DSL $\therefore \forall$ DSL's are bad. $\blacksquare$"
- "He always lies."

You recognize the fallacy

It's not that regular expressions are **all** bad. The RE syntax is a jumble of infix and postfix operators.

It's that **overuse** of regular expressions can be bad.
(The Jamie Zawinski quote is about PERL.)
You know: "When your only tool is a hammer. . ."

Pragmatically, a DSL divides the problem domain.

- You make a Domain-Specific Language.
- Other people use the DSL to solve problems.

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.

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.

Section 2

DSL Design

Behind the first door is a dark hallway

The hallway leads to a door. A sign on the door says **Model**.

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

Behind the **Syntax** door is a workroom

There are two doors and an arch

- One door has a pair of worktables.
- One door has a stack of shelves.
- The arch is decorated with two snakes.

The two **Syntax** work tables

One table has tools and libraries and modules and is labeled Tokenizer.

The second table tools and libraries and is labeled Parser.

There are helpful books by DABEAZ everywhere.

You find a tidy pile of parts labeled **Lark** that looks promising.

The two **Syntax** work tables

One table has tools and libraries and modules and is labeled Tokenizer.

The second table tools and libraries and is labeled Parser.

There are helpful books by DABEAZ everywhere.

You find a tidy pile of parts labeled **Lark** that looks promising.

Warning: Work!

You'll have to build something to open the adjacent door.

The **Syntax** shelves

Each shelf has a label:

- JSON
- YAML
- TOML
- HUML

The **Syntax** shelves

Each shelf has a label:

- JSON
- YAML
- TOML
- HUML

That's easy.

Pick one to open the adjacent door.

The **Syntax** archway

Flanked by two friendly-looking snakes

Impossible!

Even though you turned left through the **Syntax** door. . .

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

The **Syntax** archway

Flanked by two friendly-looking snakes

Impossible!

Even though you turned left through the **Syntax** door. . .

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

Where else could it be?

Why Python?

Use Python and you don't need to write a parser.

But Whatabout...?

- **Security? The DSL is Code!**

Why Python?

Use Python and you don't need to write a parser.

But Whatabout...?

- 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 Python?

Use Python and you don't need to write a parser.

But Whatabout...?

- 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.
- **The really ugly data model?**

Why Python?

Use Python and you don't need to write a parser.

But Whatabout...?

- 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.
- The really ugly data model?

  Wait for case study 2.

Section 3

Case Study 1 — Dice

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?

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`

Why add the *?

Python syntax lets you teleport straight to the data model.

The Dice 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
```

```
D6 = D(6)
```

Now, `3 * D6` works.

Section 4

Case Study 2 — OpenD6 Spells

Context — legacy data

OpenD6 rules have an Open Gaming License (OGL), making them open source with attribution. Pefect 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. Pefect 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. Pefect 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. Pefect for extensions and customizations.
I wanted **all** the magical spell definitions.

Which are in PDF's.

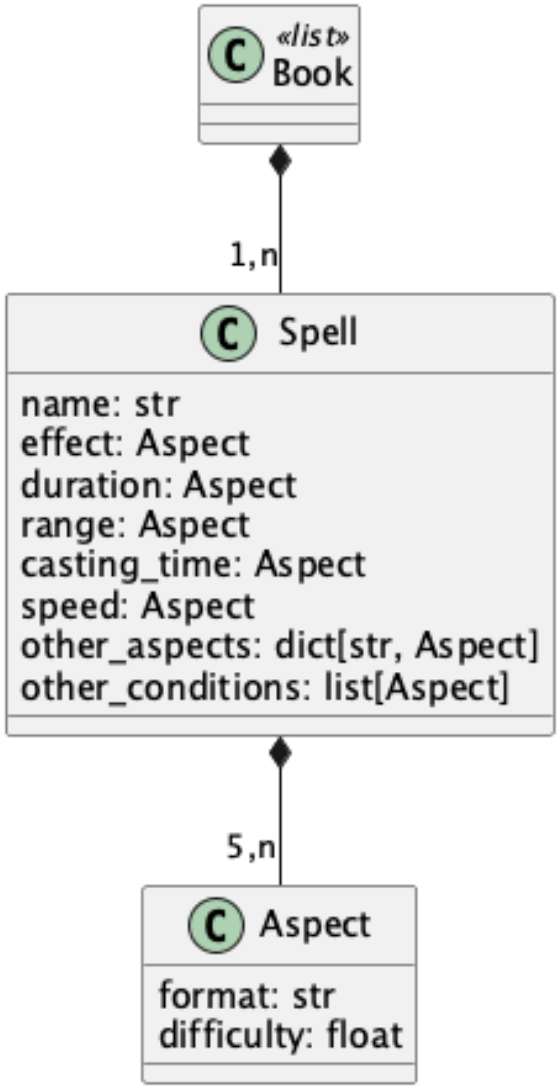
Scanned from a printed copy.

How to capture legacy data?

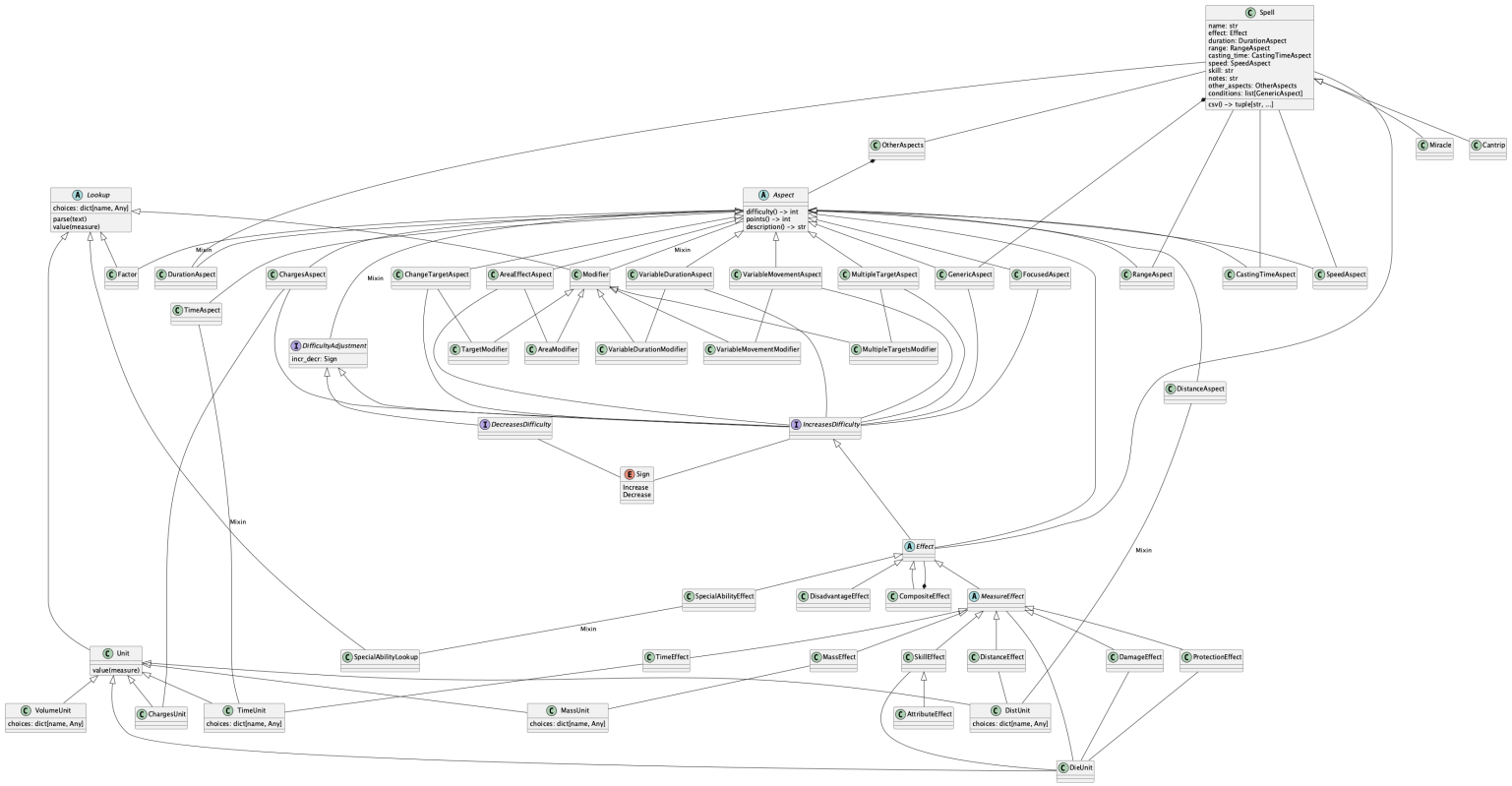
Strategy Tip

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

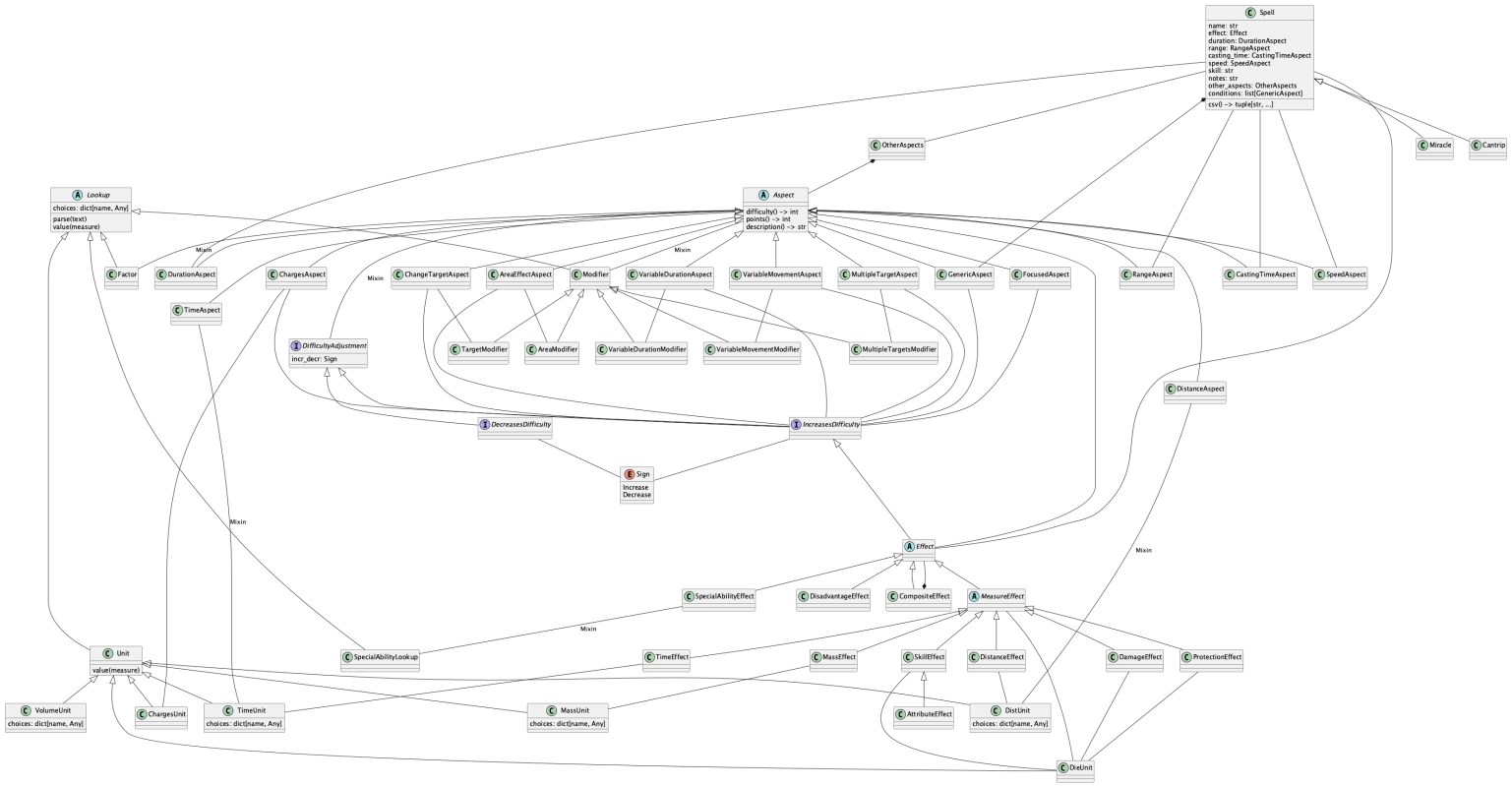
Legacy Data Model



DSL Data Model

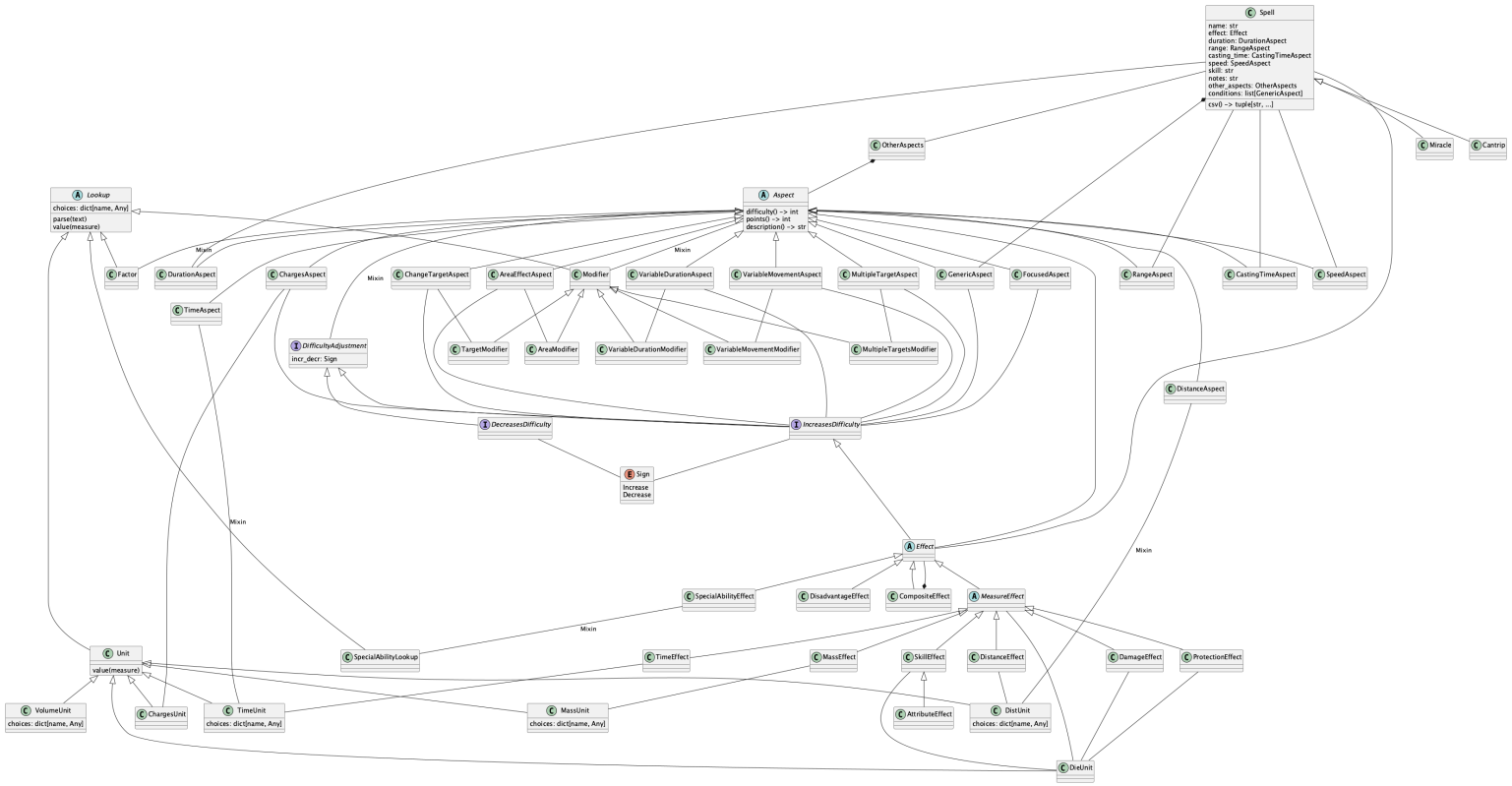


DSL Data Model



Yes, it's complicated.

DSL Data Model



Yes, it's complicated.

I omitted some stuff, and it was still vast.

Example

```
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 boilerplate. API design is ongoing.

Typical DSL Use Cases

- Represent legacy content and rules in the DSL.
- Compute derived values.
  (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.
  (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 to evolve.

Typical DSL Use Cases

- Represent legacy content and rules in the DSL.
- Compute derived values.
  (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 to evolve.
- DSL evolution will reflect the lessons learned.
  The SOLID **Open/Closed Principle** is your friend.

Whatabout 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.**

Whatabout 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.**

Whatabout 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.

Patterns to use:

- Façade – Koan 5: flat is better than nested
- Adapter – Koan 8: make special cases fit the rules

More ideas? Run `import this`

Section 5

Conclusion

Tips for DSL Design

- Use Python as the syntax for your DSL.

Tips for DSL Design

- Use Python as the syntax for your DSL.
- Capture legacy rules with a quick-and-dirty model.

Tips for DSL Design

- Use Python as the syntax for your DSL.
- Capture legacy rules with a quick-and-dirty model.
- As you learn. . . build a complete DSL model.

Tips for DSL Design

- Use Python as the syntax for your DSL.
- Capture legacy rules with a quick-and-dirty model.
- As you learn. . . build a complete DSL model.
- As you learn even more. . . wrap the complicated parts of the DSL model to simplify the API.

Appendix

Malicius Actors

Malicious Actors

Complicated Situation

Untrusted sources + large DSL docs + difficulty in doing Quality Assurance.

- Use JSON or TOML.
  Use a `@classmethod` to build `Spell` from `dict[str, Any]` content.
- Don't allow a spell-book module to use `import`.

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))
```