LESSONS FROM DDD

DOMAIN DRIVEN DEVELOPMENT

Author Owen Arnold

DOMAIN DRIVEN DEVELOPMENT

“Connecting the implementation to an evolving model.”

Object Orientation - the way it should be done

Extending the type system to fit the problem domain, not the other way around.

GUIDING PRINCIPLES

1. Place primary focus on domain and domain logic
2. Complex designs are based on domain
3. Collaboration between domain experts and techincal team achieved via the model
4. Code forms the Ubiquitous Language

CORE BUILDING BLOCKS

- **Entity Objects**

  - Things that have identity

- **Value Objects**

  - Mutable things that do not have identity

- **Services**

  - Things that provide functionality without the need for state

TIP 1: DEFINE WHAT OBJECT EQUALITY MEANS

For Value Objects, the first thing to do in developement is to define what equality means for objects of that type. A TDD approach works very well for this.

Advantages:

1. Greatly reduces semantic errors
2. Improves encapsulation
3. Frees the client from knowning the type internals
4. Makes testing of objects much easier

TIP 2: BUILD FOR TYPE SAFETY

Given this function:

```
void print_info(std::string name, std::string colour) {
  std::cout << name << "'s favourite color is " << color;
}
```

This works:

```
int main() {
  std::string favourite_colour = "Red";
  std::string first_name = "Dave";
  print_info(first_name, favourite_colour);
}
```

```
Dave's favourite color is Red
```

TIP 2: BUILD FOR TYPE SAFETY

But so does this:

```
int main()  {
  std::string favourite_colour = "Dave";
  std::string first_name = "Red";
  print_info(favourite_colour, first_name);
}
```

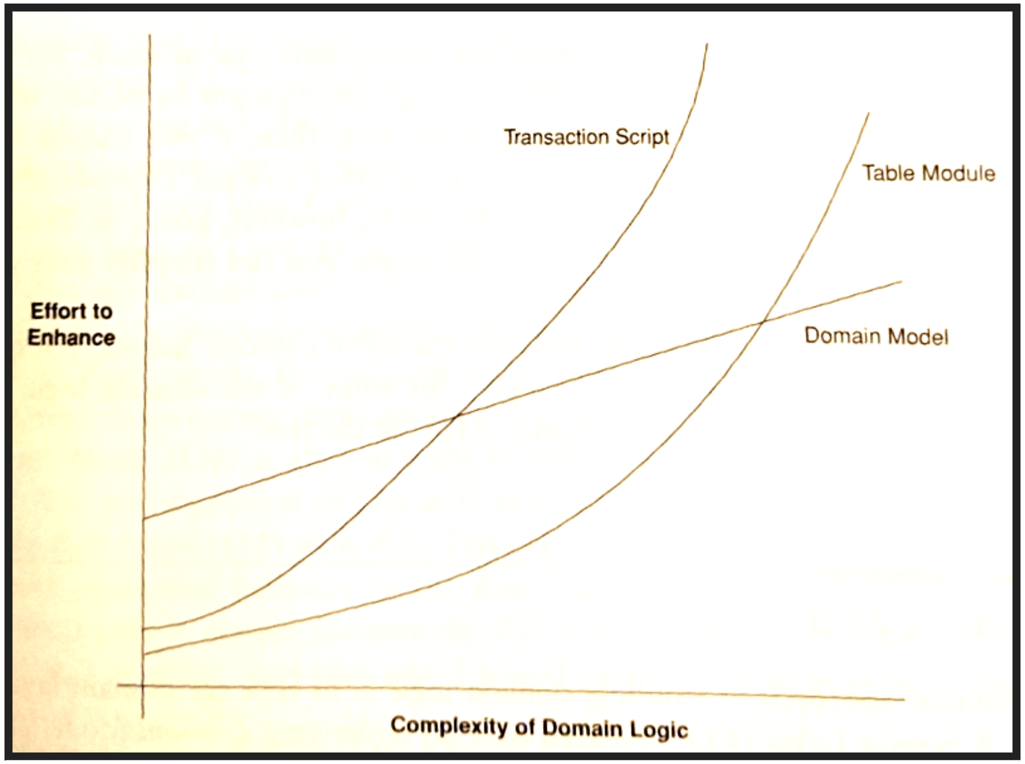
```
Red's favourite color is Dave
```

Solution is to make *favourite_colour* and *first_name* ValueObjects in the domain. They are both different values AND types. This can be caught at compile time.

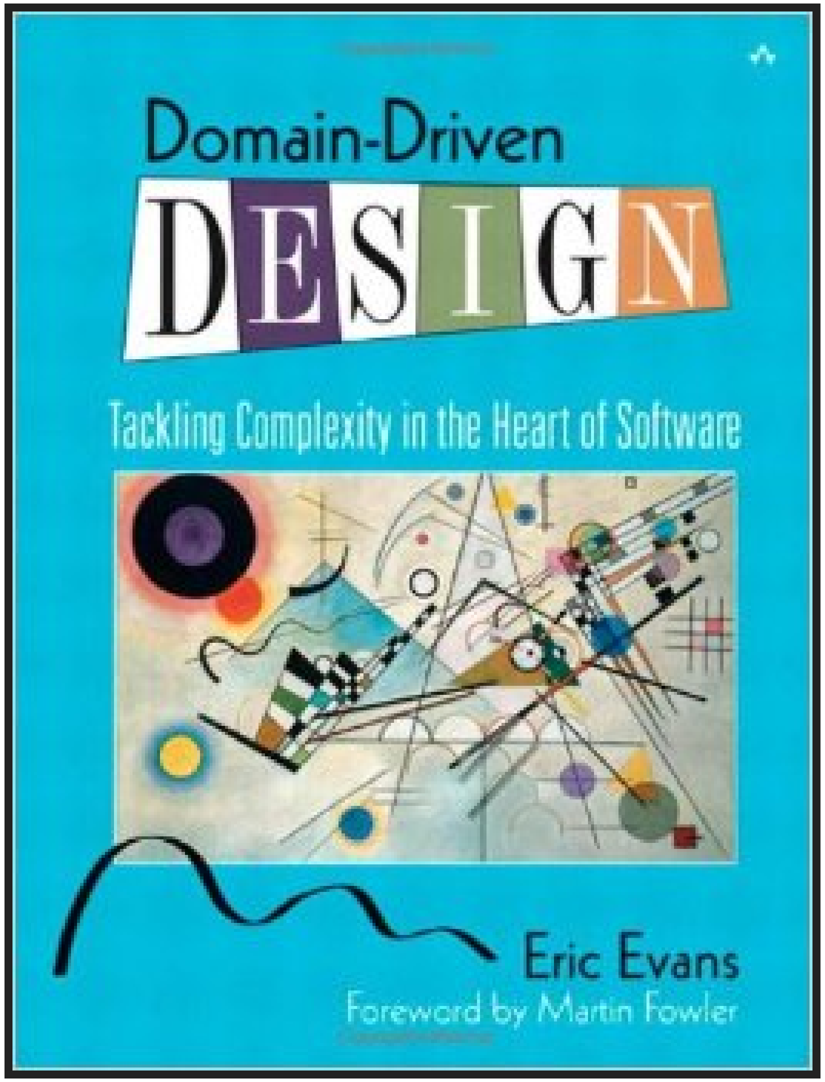
TIP 3: SERVICES ARE INJECTABLE

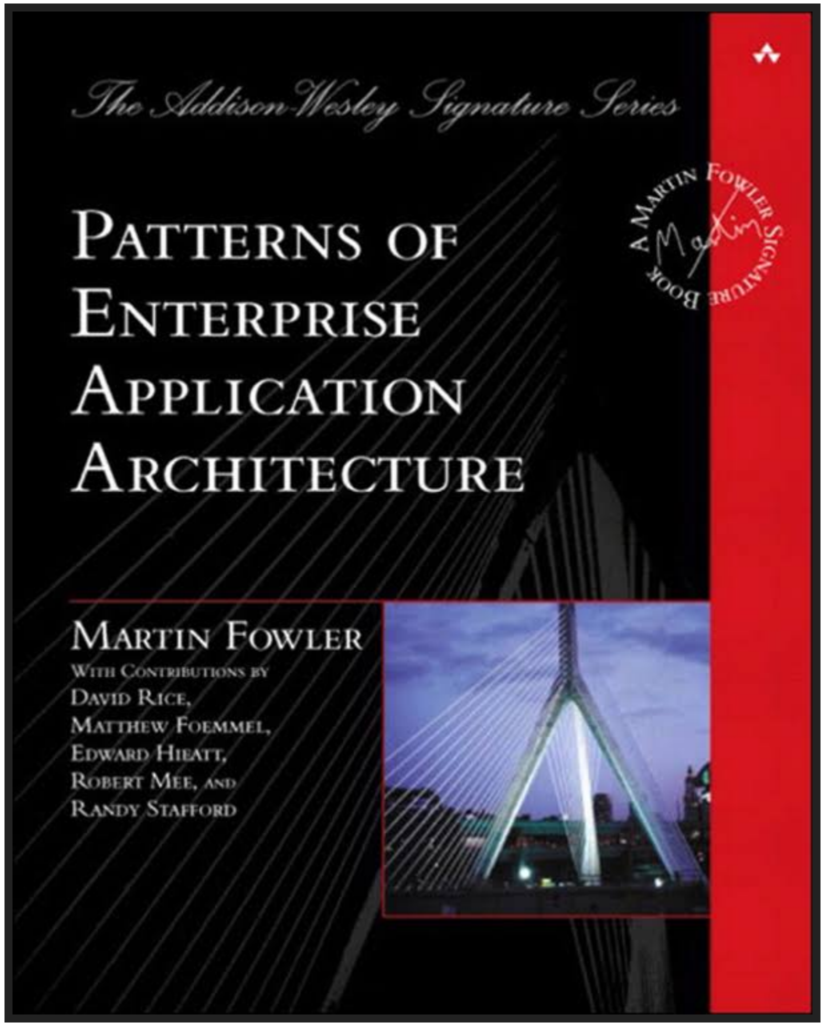
1. Reusable things that have no state should be considered services
2. Write client code to defer choice of concrete services (IOC)
3. Never hard-code a concrete service behind an API
4. Mock services in tests to make tests run very fast
5. Add services for missing 'Axes'

PROVEN BENEFITS



FURTHER READING


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler


The Addison-Wesley Signature Series
Patterns of Enterprise Application Architecture
A Martin Fowler Signature Book
Martin Fowler
With Contributions by
David Rice,
Matthew Foemmel,
Edward Hieatt,
Robert Mee, and
Randy Stafford

https://en.wikipedia.org/wiki/Domain-driven_design