Design Advice

1. Class-Level Advice
   - Standalone versus Base Classes
   - Interfaces
   - Code Smells (Indicators of Poor Design)
     - Monolithic Class
     - Data Class
   - Abstractions at Different Levels
2. Architecture Advice
   - Layering
   - Protecting The Domain Layer

Design Advice and Code Smells

Be Clear About What Kind of Class You Are Writing

The design rules for *standalone* and *base classes* classes are very different, and client code treats base classes very differently from standalone classes.

Decide what kind of class you need before you design it.

Standalone Classes

Standalone classes:

- have a public destructor, copy constructor and assignment operator with *value semantics*
- have no virtual functions
- are usually instantiated on the stack or as a directly held member of another class (direct composition)
- are not intended to be used as a base class

Base Classes

Base classes should:

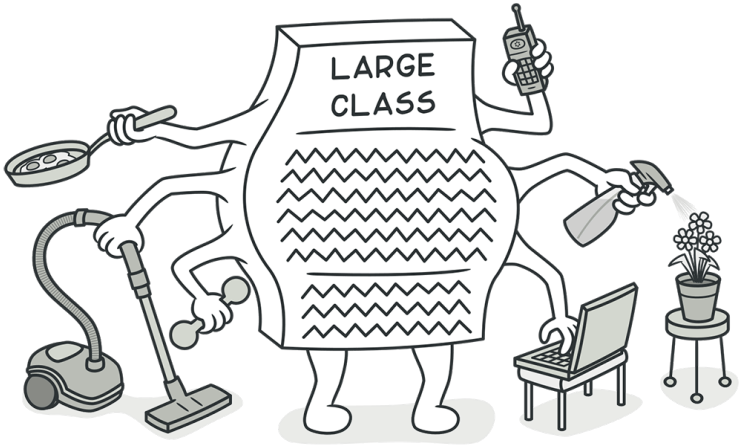
- establish interfaces which are as simple as possible while still effectively modelling a role in the problem domain
- have a destructor which is public and virtual
- shield code from knowing about the actual type(s) being operated on, by being used as:
  - reference parameters of functions
  - the type for instantiating smart pointers, or vectors of smart pointers

Pay Attention to Your Interfaces

> *The most important thing to get right is the interface. Everything else can be fixed later. Get the interface wrong and you may never be allowed to fix it.*
>
> — Sutter's Law of Second Chances

> *Interfaces, like diamonds, are forever.*

Monolithic or Large Class



Catalog of Refactoring — Shvets
https://refactoring.guru/smells/large-class

What are the different concerns that `Flight` deals with? Give the names of smaller, more focused classes that you could extract from `Flight`.

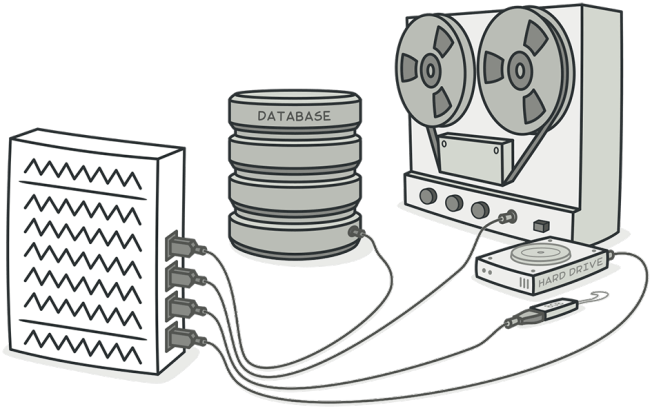
```
class Flight {
public:
  Flight(Aircraft plane,Place orig, Place dest);
  unsigned int GetMaxSpeed() const;
  void ScheduleTakeOff(const Time& time);
  Time GetFlyingTime() const;
  void AdjustFlightPath(Paths other);
  void AddPassenger(const Person& p);
  void RemovePassenger(const Person& p);
  vector<Person> GetPassengerList() const;
  Clearance SecurityCheckPassenger(const Person& p) const;
  unsigned int EstimateNoInflightMeals() const;
  void SetMealTypeAndNumber(MealType meal, unsigned int num);
  Rands TotalMealCost() const;
};
```

Avoid Monolithic Classes, Prefer Minimal Classes

- A minimal class is easier to comprehend and more likely to be used and reused in a variety of situations
- A minimal class embodies one concept at the right level of granularity. A monolithic class is likely to embody several separate concepts and using one implies understanding all of the others

Divide and conquer: small, focused classes are easier to write, get right, test and use.

Beware the Data Class

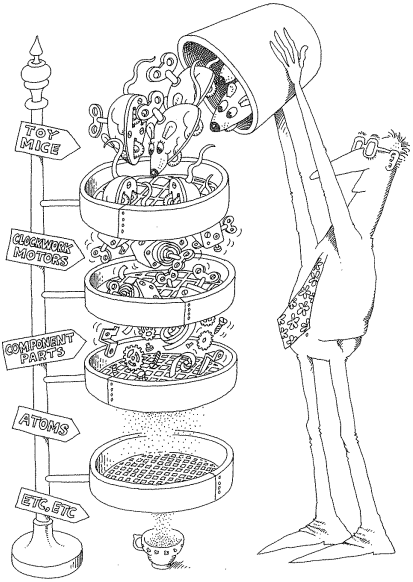


Catalog of Refactoring — Shvets
https://refactoring.guru/smells/data-class

Lots of getter and setter methods but no real behaviour.

Abstractions at Different Levels

- Classes and their objects are the building blocks of an application
- High-level abstractions build upon lower-level abstractions



Source: Object-Oriented Analysis and Design, 2nd ed., G. Booch, 1994