0.1 Responsibility-Driven Design

Describes a systematic approach to think about the design of software objects and components in terms of **responsibilities**, **roles** and **collaborations**.

0.1.1 Responsibility Types

Responsibilities in general are related to

- **Obligations** of an object
- **Behaviour** of an object

in terms of its role in the software design.

Type: Doing Responsibility

Doing Responsibilities describe responsibilities that are related to performing a task.

- Doing it (perform a calculation, create an object)
- Initiate action in other objects
- Control and coordinate activities in other objects

For Example: A Bill object is responsible for calculating the total price.

Type: Knowing Responsibility

Knowing Responsibilities describe Responsibilities that are related to **knowing** and **providing** information to other objects.

- Knowing private, encapsulated data
- Knowing related objects

For Example: A Car is responsible for knowing the driven distance.

0.1.2 Responsibilities and Methods

A responsibility is **NOT** the same as a method. A responsibility can be modelled with a method, but in many cases its better to model it with multiple. Therefore a method is part of a responsibility and can be the whole responsibility, but a responsibility can also be split into multiple methods.

There is no real method to determining how to split responsibilities. It is often very circumstancial and needs to be adjusted to the needs at hand.

0.2 More Design Principles

Ideally a system design should follow the **Single Responsibility Principle (SRP)**. This means that a class / object should have a responsibility which is its primary reason to change. Therefore one responsiblity per class is the ideal.

0.2.1 Collaboration of Multiple Classes

Oftentimes a oolicy requires collaboration of multiple classes. This makes it hard to choose which of these classes should handle the responsibility. There is also no easy answer for this. Sometimes there is a clear class that makes access to the others easier. To figure this out multiple drafts may be needed.

0.2.2 Delegation vs. Inheritance

To figure out where to place specific responsibilities the concepts of delegation and inheritance are useful.

Delegation

Delegates responsibilities to other objects:

- Get objects from other class with the needed functionality
- Use the object to fulfil only the needed functionality
- Inheritance hierarchy remains unchanged

Inheritance

Inherit responsibilities from baseclass:

- Violates SRP
- All subclasses of the current class are forced to also inherit the responsibility
- Not required functionality from the baseclass is also inherited
- Inheritance hierarchy is changed harder to maintain and understand

Most of the time delegation is preferred. As the design is more understandable and maintainable. It also is evaluated at runtime rather than compiletime. Inheritance should only be used when the responsiblity extends the functionality **organically**.

0.3 Encapsulation

Interface

An Interface declares the method signatures and public constants of its implementing classes. They provide independence of functionality from implementation.

Always Program to Interfaces

- Fields, return types, method parameters etc. should be declared with interface type
- Public methods should not expose implementation details
- Fields in implementing classes should be private. Retrieval & modification of information via getters and setters should be used.

This process has the advantages:

- Avoids unjustified assumptions about implementation
- Interfaces are more stable than implementations
- To change the implementation its sufficient to exchange the constructor
  - Can factor out implementation-independent code into abstract classes
  - Reduces coupling

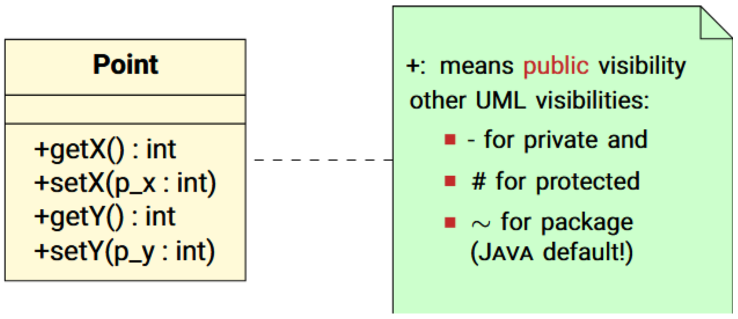
0.3.1 Field Access

Instance fields should **NEVER** be public. This ensures that information is hidden and adheares to the "Always Program to Interfaces" rule.

It also makes sure that there is a distinction between implementation-specific data and public data. Client should not be able to access implementation-specific data.

0.3.2 Accessor Methods Should not Expose More Than Necessary

Accessors shouldn't always be used. Oftentimes the use of accessors is unnecessary and only creates more coupling. In many cases it's better to outsource the responsibility of the accessor / the surrounding responsibility to another class instead.



UML Access Modifiers

There are some reasons to use accessors though:

- Responsible for aspects of the UI (Data important for visualization)
- Class using accessors implements a policy
- Class is a static container

0.4 Design Knowledge: The God Class Problem

God Class

A class that contains most of the system logic:
- Promotes poorly distributed responsibilities
- Not object oriented design

To avoid this you can use the following criteria:

- Avoid classes with unclear responsibilities
  - Classes that fail the SRP principle
  - Solution: Split class and relocate the responsibilities
- Avoid classes with low cohesion and non-communicating classes
  - Classes with methods operating on a small subsets of its fields but not with other objects
  - Solution: Split class and relocate the responsibilities
- Avoid classes with public field accessors:
  - Public field accessors can indicate wrongly located responsibilities
  - Solution: Redesign interface and relocate the responsibilities

In general, when designing a system, one should try to model the real world. This however can produce complex systems. Therefore often it is advised to put the real world model aside and design the system as according to the design principles.