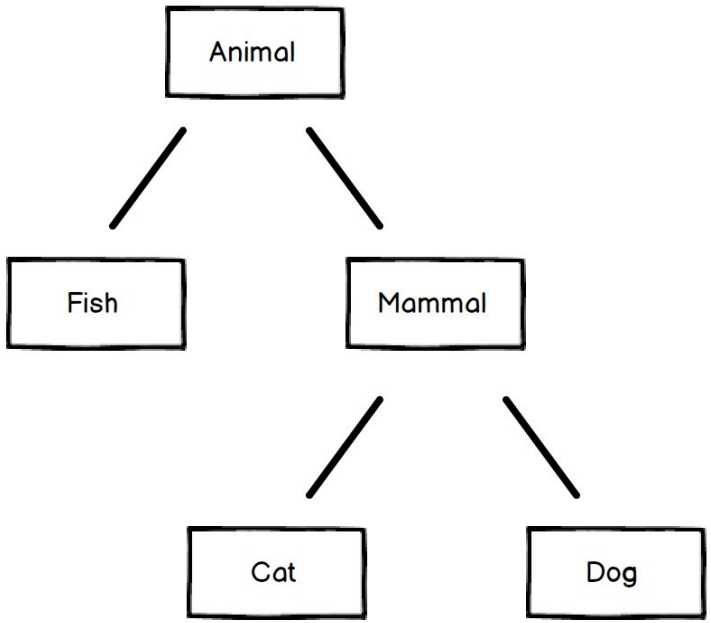
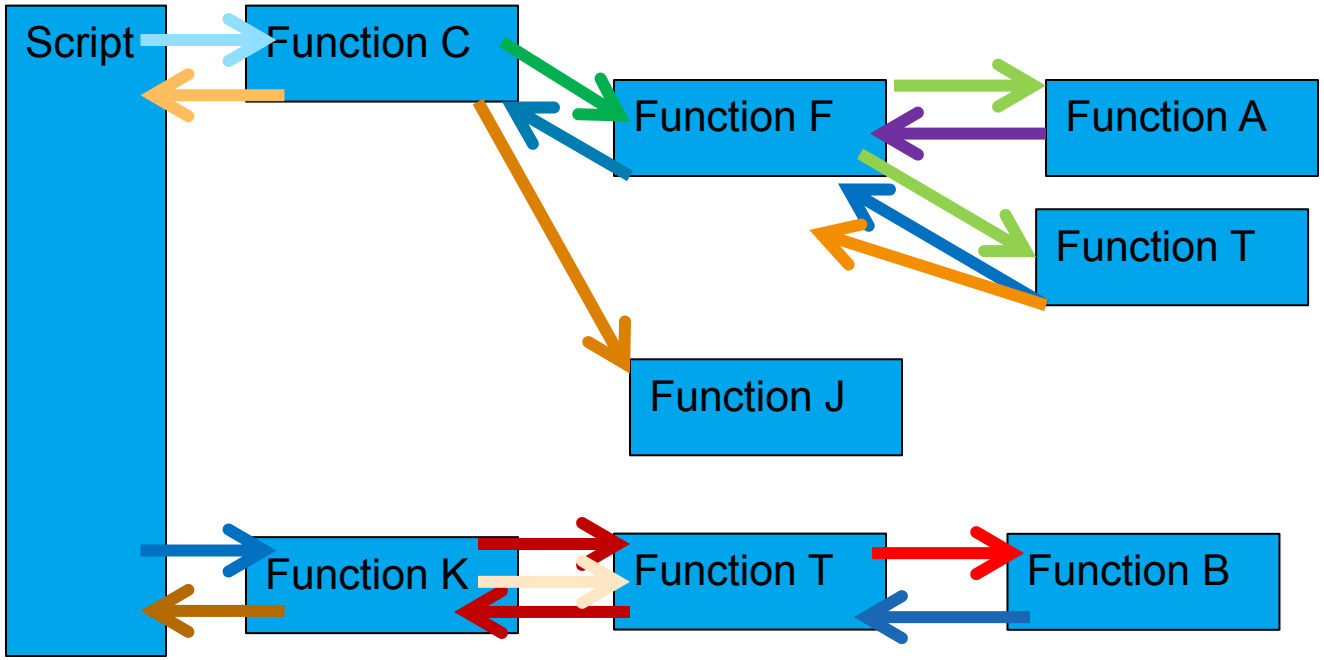
Introduction to OOP at PITZ Pt 2

Introduction to Inheritance in Object Oriented Programming



GOAL:
Reduce code duplication
Maximize maintainability, readability, reusability

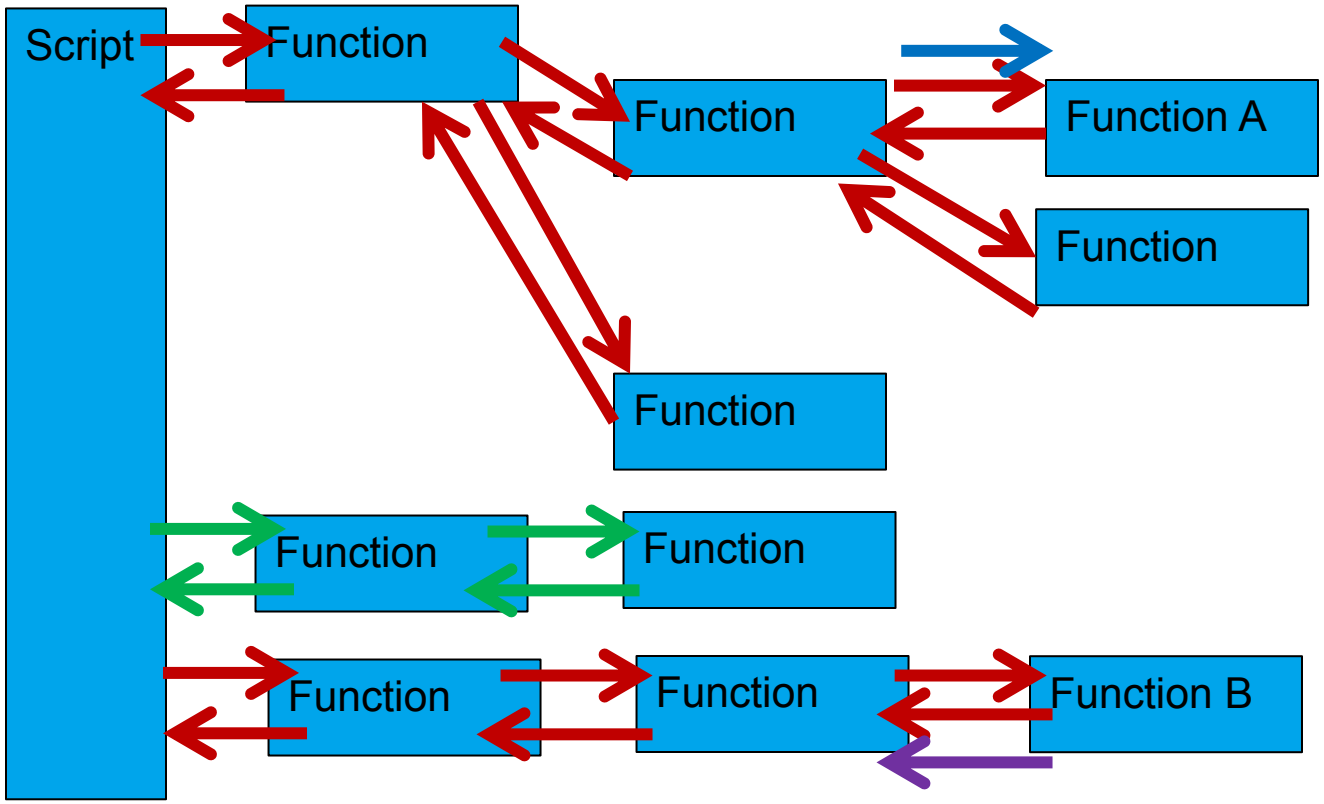
Dataflow



Data passing requires a lot of function input/ouput fields to be modified and the code where they are used

I now want to use data from A in B.
Oh No! I have to change ALL the functions (→data pipeline)

Dataflow



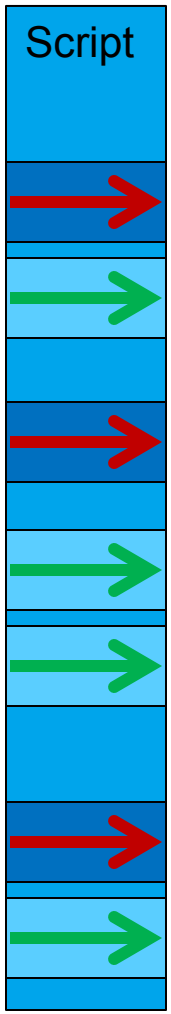

```
classdef
properties
  
end

methods
  
end
```

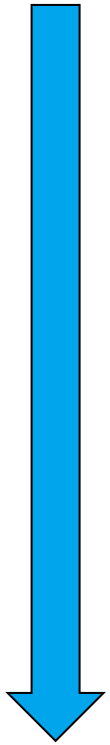

Data Pipeline is provided by OOP.
No need for global variables.

Objects come with lots of other useful features.

- **Polymorphism**
- **Inheritance**
- **Events**

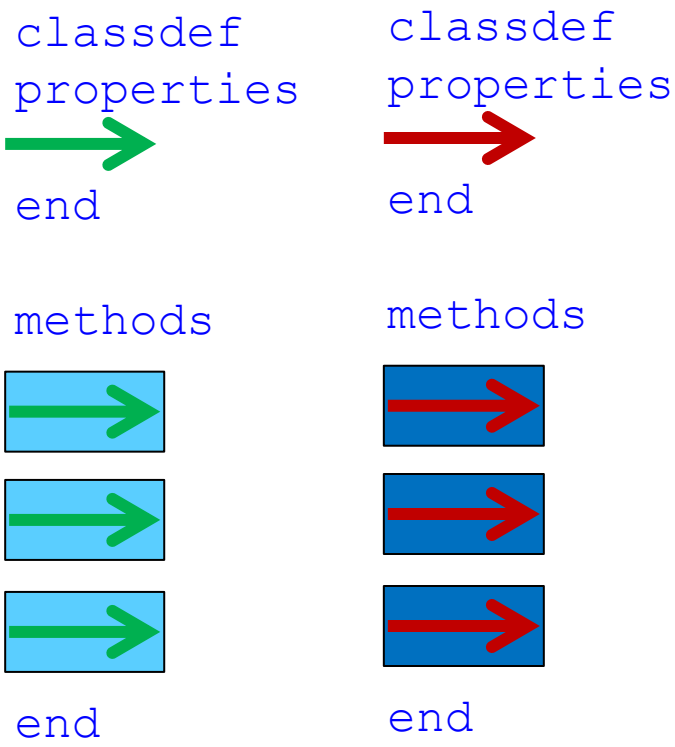
The Object Paradigm





Function: Turn input into output data

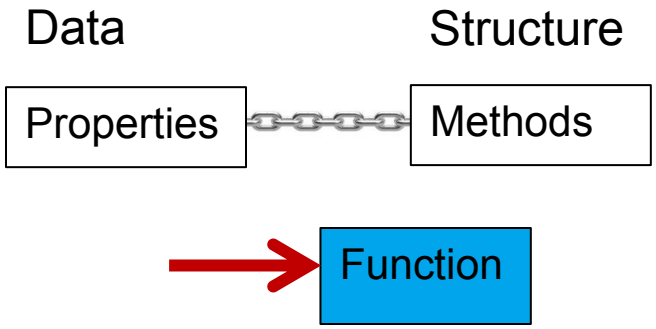
Class: Manages lifecycle of objects from creation to destruction



Solid, reusable and encapsulated logic

Using this concept needs practice

Polymorphism



No need for unique function names as type of class + function name define which function to execute

```
mot=USBMotor('USB1');
if ~mot.ishome; %~mot.isHomed()
    mot.HomeMotor();
    mot.WaitMotorArrive();
end
mot.SetPosition(5);
mot.WaitMotorArrive();
```

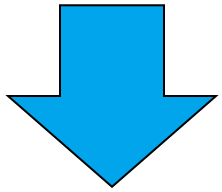
```
classdef USBMotor < handle
    properties
        USBAdr
        %mstat
        setpos
        rbkpos
        moving=0;
        ishome=0;
    end
    methods
        function isHomed(obj) end
        function HomeMotor(obj) end
        function WaitMotorArrive(obj) end
        function SetPosition(obj,pos) end
    end
end
```

Polymorphism allows code writing without explicit knowledge which function is executed

A parent class

```
function WaitMotorArrive(obj)
            while obj.moving
                obj.PollStatus;
                pause(0.2);
            end
end
```

This function would have to be copy pasted for both objects.



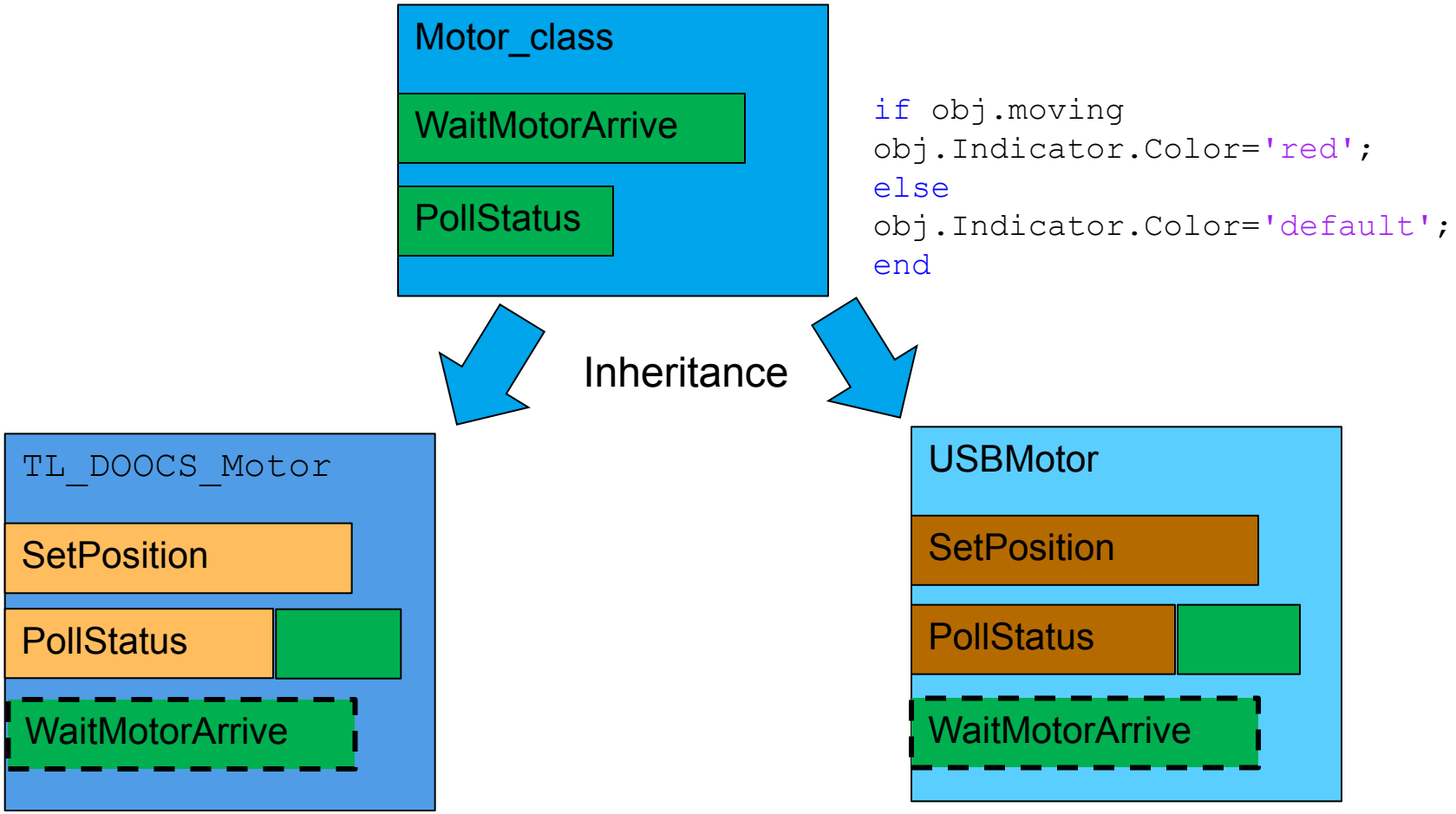
```
classdef Motor_class < handle
properties
        setpos
        rbkpos
        moving=0;
        ishome=0;
end

methods
         function PollStatus(obj) end
         function WaitMotorArrive(obj) end
    end
end
```

Write them into a common superclass

Copy & Paste of Code is always bad

```
classdef TL_DOOCS_Motor < Motor_class
classdef USB_Motor < Motor_class
```

Class Hierarchy

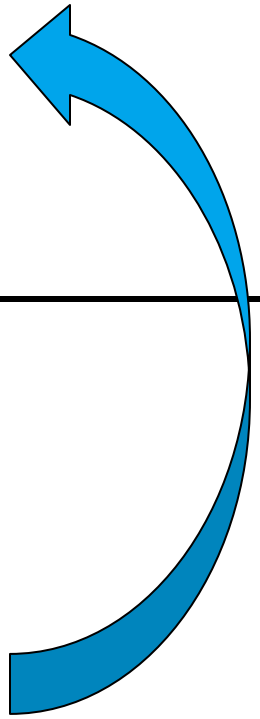



```
if obj.moving
obj.Indicator.Color='red';
else
obj.Indicator.Color='default';
end
```

Override Methods

```
classdef Motor_class < handle
properties
          setpos
          rbkpos
          moving=0;
          ishome=0;
          Indicator;
end
methods
          function WaitMotorArrive(obj) end
          function PollStatus(obj)
              if obj.moving
                  obj.Indicator.Color='red';
              else
                  obj.Indicator.Color='default';
              end
           end
end
```

Override Methods should have the same input & output arguments as their parent implementation



```
classdef TL_DOOCS_Motor < Motor_class
properties
          mstat
          STATUSadr
end
methods
    function PollStatus(obj)
          [s] = xcomm(obj.STATUSadr,'connection','sync');
          obj.mstat=s.data;
          obj.moving=(bitget(obj.mstat,3) & bitget(obj.mstat,5));
          obj.PollStatus@Motor_class()
    end
end
```

Method & Properties Attributes

```
methods(Access=public)
end
methods(Access=protected)
end
methods(Access=private)
end
```

Default

```
properties(Access=public)
end
properties(Access=protected)
end
properties(Access=private)
end
```

- public — Unrestricted access
- protected — Access from methods in class or subclasses
- private — Access by class methods only (not from subclasses)

If there is a connection between certain data or structure or if there is other additional information to your code, announcing it, usually comes with benefits.

https://de.mathworks.com/help/matlab/matlab_oop/method-attributes.html

Abstract Classes

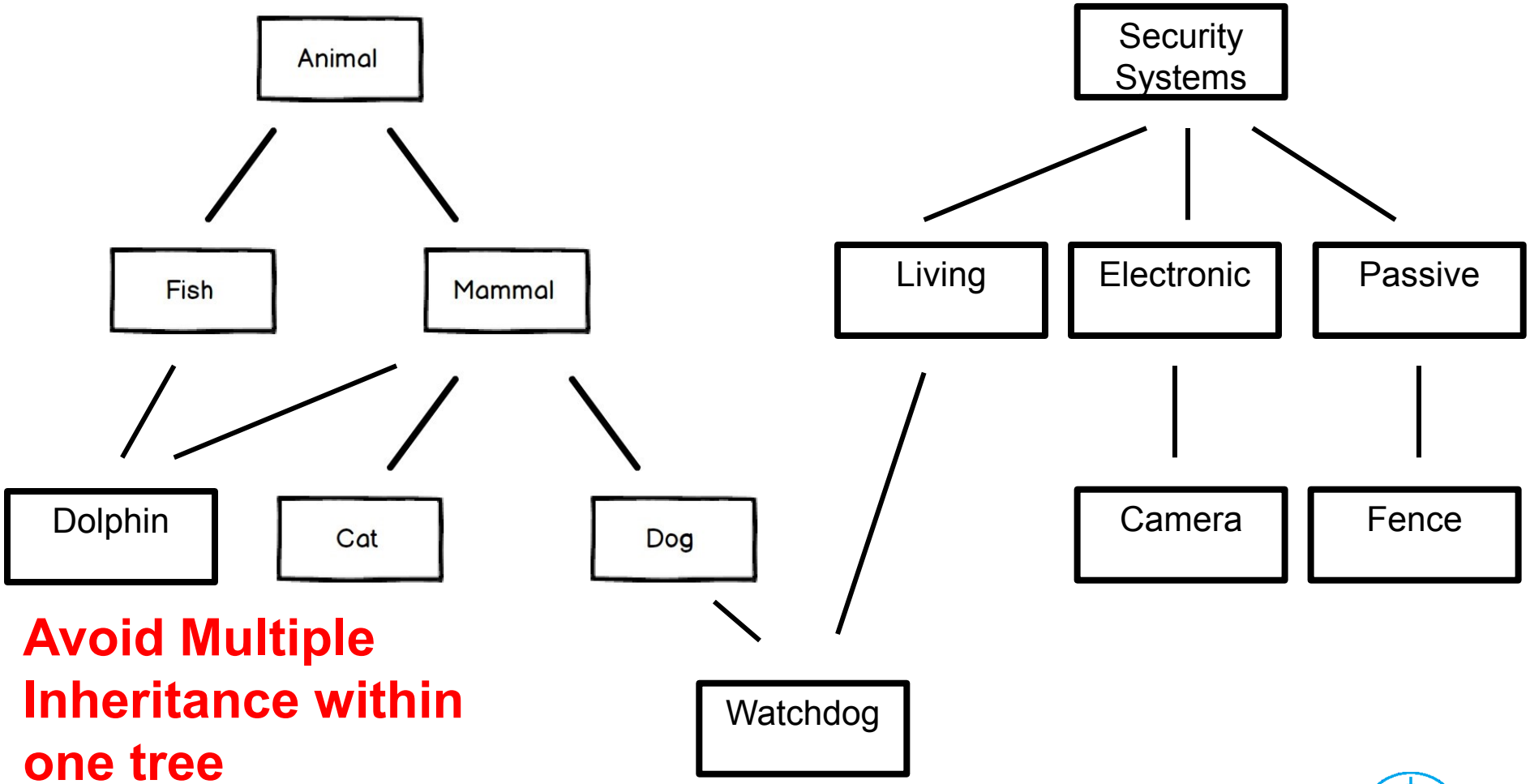
```
classdef (Abstract) Motor_class < handle
properties
            setpos
            rbkpos
            moving=0;
            ishome=0;
            Indicator;
end

methods
    function WaitMotorArrive(obj) end
    function PollStatus(obj)
        if obj.moving
            obj.Indicator.Color='red';
        else
            obj.Indicator.Color='default';
              end
     end
end

methods (Abstract)
    SetPosition(obj,pos)
    HomeMotor(obj)
end
```

- Abstract classes can not be instantiated. They can only be inherited from.
- Defining abstract Methods or Properties makes the class abstract
- Explicitly abstract classes don't need to define abstract Methods or properties

https://de.mathworks.com/help/matlab/matlab_oop/abstract-classes-and-interfaces.html

Multiple Inheritance

```
classdef MyClass < SuperClass1 & SuperClass2
```



Avoid Multiple Inheritance within one tree

(matlab.mixin classes)

Set & Get Functions

```
classdef Circle_class < handle
    properties
        radius
        circum
    end
    methods
       function value = get.radius(obj)
       value=obj.radius;
       end
       function set.radius(obj,value)
           obj.radius=value;
           obj.circum=2*pi*value;
       end
    end
end
```

Set&Get functions can only be used with handle classes

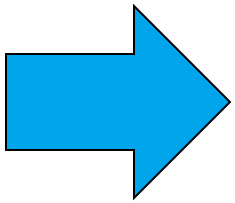
Default automatic functions

Keep the behavior intuitive

Use
- checking data type
- updating dependent value
- notify change
- restrict access to callers

Polymorphism VS conditional statements

```
switch data.type
    case 1
        data.Value=ThisFunction()
    case 2
        data.Value=ThatFunction()
    case 3
        data.Value=AnotherFunction()
end
```



```
dataobject.TheFunction()
```

```
classdef Data
    methods
        function TheFunction() end
    end
end

classdef DType2 < Data
    methods
        function TheFunction()
            obj.Value=ThatFunction()
        end
    end

classdef DType1 < Data
    methods
        function TheFunction()
            obj.Value=ThisFunction()
        end
    end
```

Using child classes will reduce the amount of conditional statements in my main script.

→ easier to read
→ easier to maintain

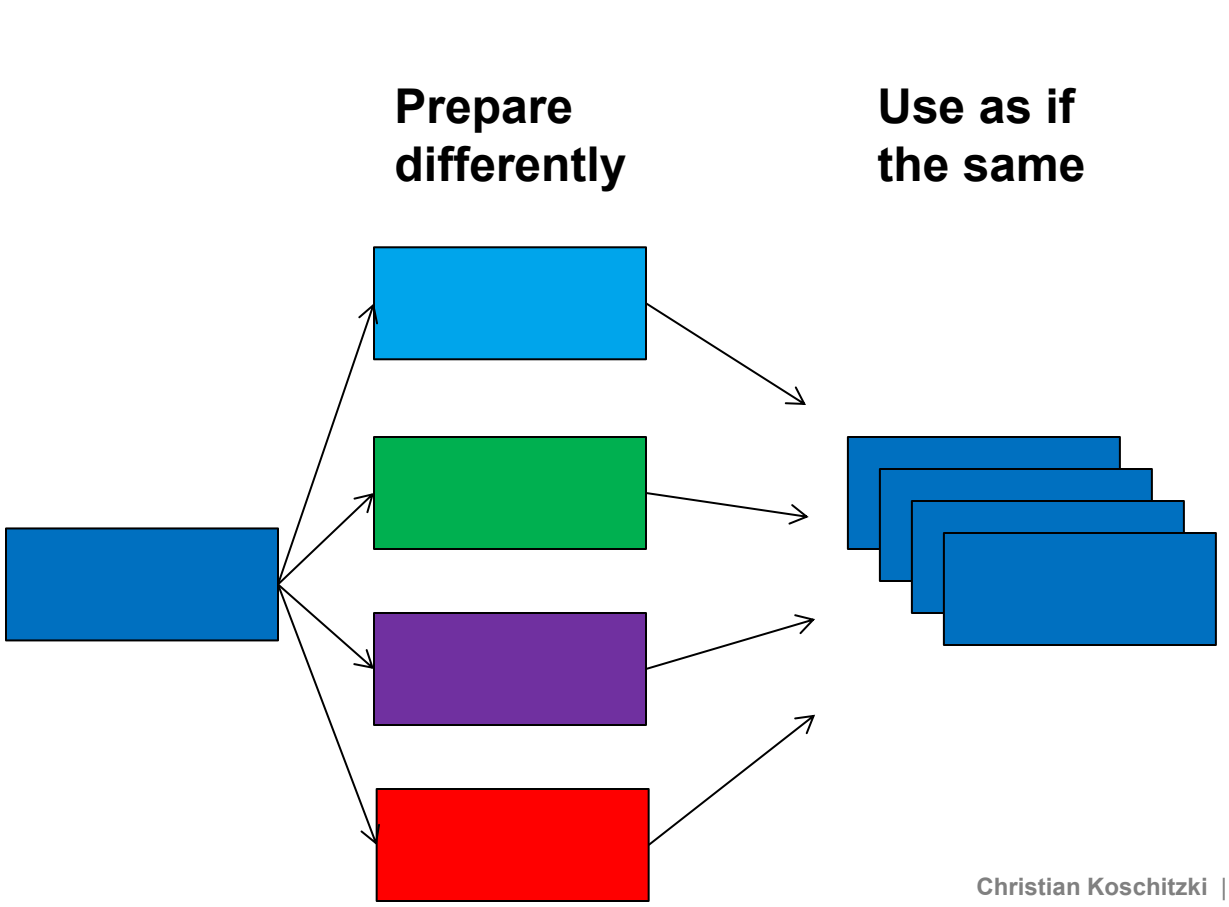
Conditional statements give clues, that inheritance may be used

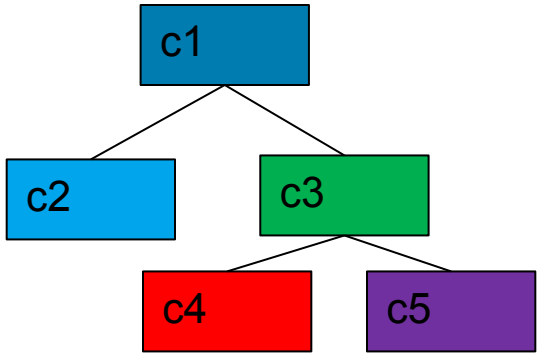
Matlab mixin classes → Heterogeneous Arrays

```
mot1=TL_DOOCS_Motor ('COM1',9600,…
         8,'even',1);
mot2=USBMotor('Port2');
motors=[mot1,mot2];
```

Matlab mixin classes are designed to customize object behavior and to be compatible for multiple inheritance

```
classdef Motor_class < handle & matlab.mixin.Heterogeneous
```

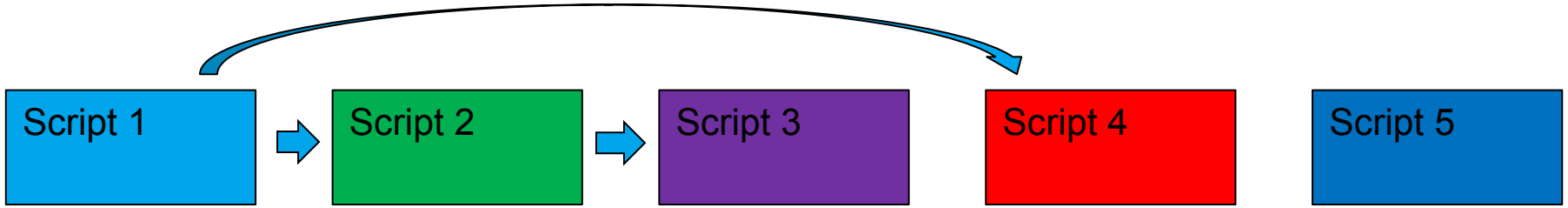




arr1=[c5,c5,c5]
arr1=[c5,c5,c4]
arr1=[c2,c5,c4]

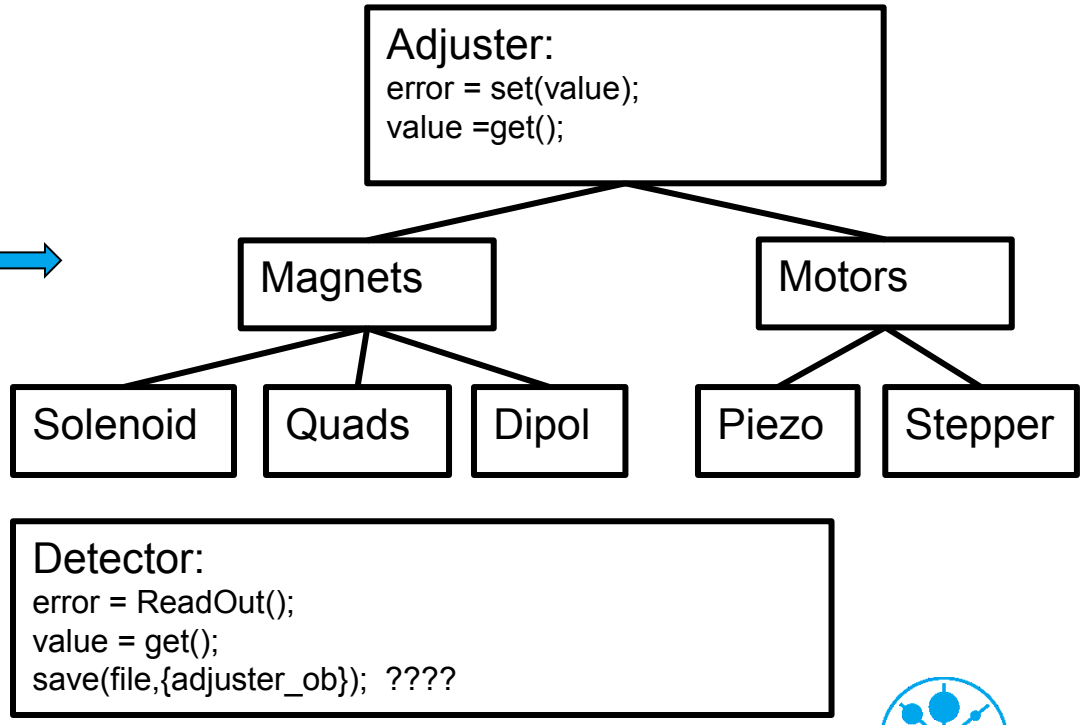
Object array type appears as of the highest common superclass

Mutliple Scripts



Every time code is copy pasted the new project gets disconnected from the old project

```
%Example Scan script
%Sequencer
mpos=linspace(start_pos,end_pos,nsteps);
%Scan Loop
for i=1:nsteps
    MoveMotor(motor_id,mpos(i));
    detector_dat(i)=ReadDetector(det_id);
end
%PreAnalysis
plot(mpos,detector_dat);
%Data Saving
save(outfile,'mpos','detector_dat');
```



Make a Toolbox

```
%Example Scan script
%Sequencer Setup
adj1=StepperMotor_class(motor_id);
det1=PressureGauge_class(gauge_id);
seq_ob=LinearSeq(adj1,startpos,endpos,nsteps);
seq_ob.AddDetector(det1);
%Scan Loop
%seq_ob.RunSequence()
for i=1:nsteps
    seq_ob.Adjust(i);
    seq_ob.Measure(i);
end
%PreAnalysis
seq_ob.PlotDetector(1);
%Data Saving
seq_ob.SaveData(outfile);
```

Modular set of solutions is more likely to be adapted.

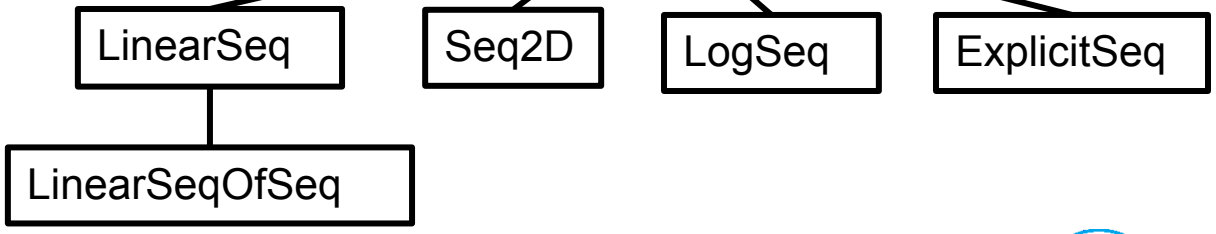
Sequencer_class

properties

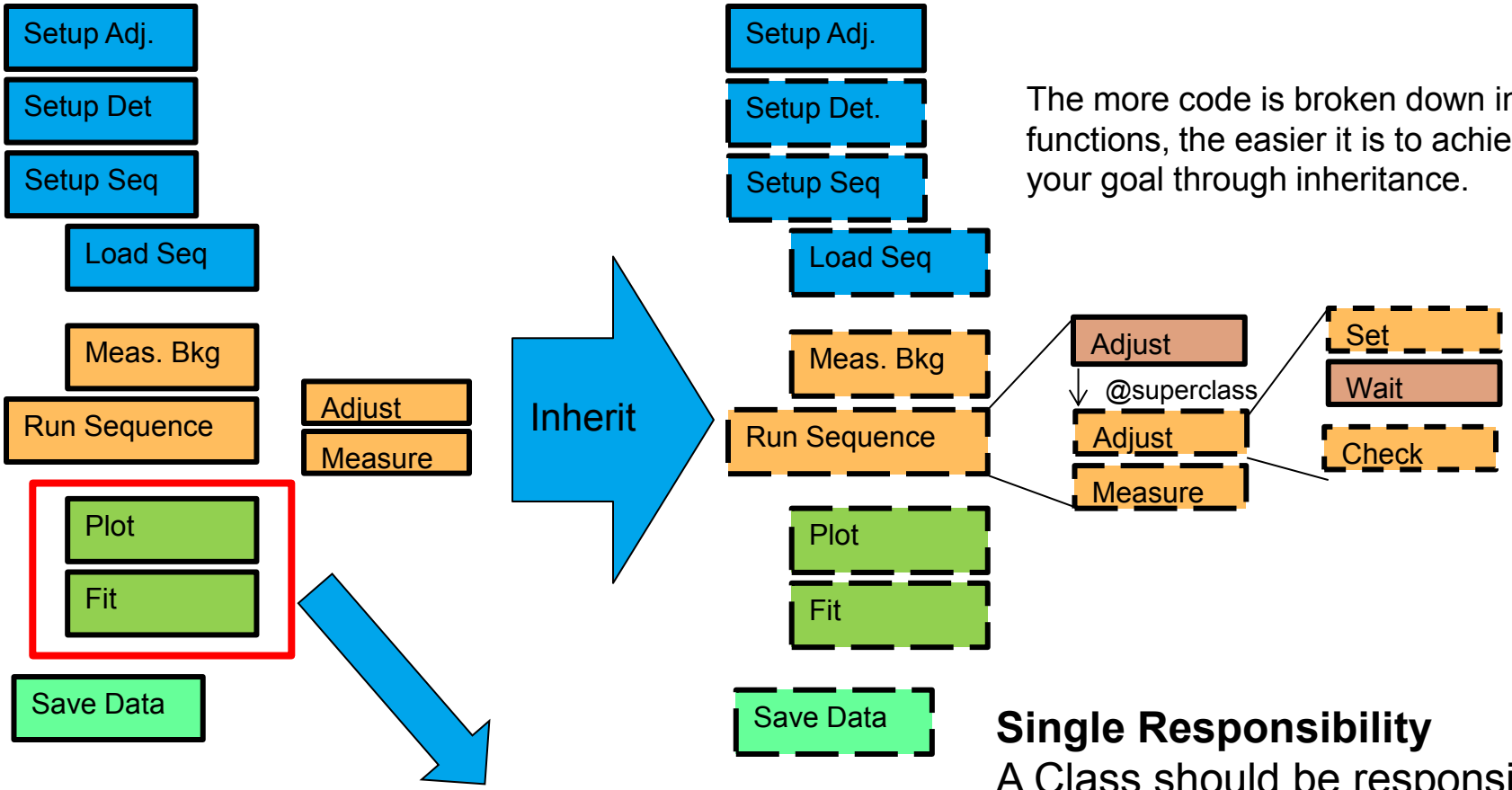
adjusters=[adjuster_class];
detectors=[struct{ detob,…
n_average,…
detector_data }];

methods

function Adjust(obj,index)
function Measure(obj,ind)
function AddDetector(det_ob, n_average)
function SaveData(outfile)
function SaveSequencer(outfile)
function LoadSequencer(file)



Single Responsibility Principle



The more code is broken down in functions, the easier it is to achieve your goal through inheritance.

Build Analysis & Visualization classes that take Sequencer Outputs

Single Responsibility
A Class should be responsible only for handling any one kind of responsibility. Big features should be moved to another Class(if necessary).

SOLID Principles

SOLID

How do I define Adjuster and Detector Superclasses?

Open to Extension:

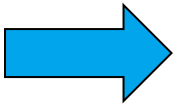
- Write new Adjuster/Detector class
- Open the Sequencer to extend the usable hardwares

Closed to modification:

- No modification to sequencer because of new Adj./Det. class
- Close the Sequencer to modifications because of new hardware

Litkov Substitution Principle:

Every Child Class must be substitutable in the place of Parent Class.



Purpose of Detector and Adjuster Child classes is to be substituted in Sequencer

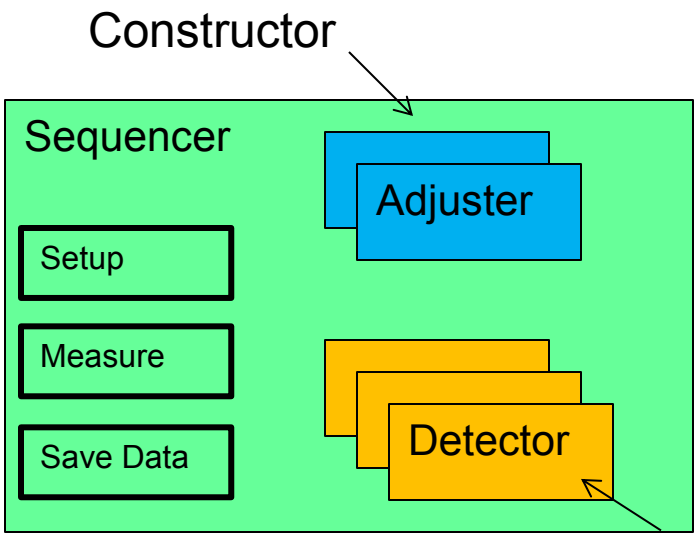
Interface Segregation Principle:

A User should not be forced to implement an interface that it doesn't use.

```
adj1.SetTimeout(timeout)
adj1.Adjust(targetpos ,timeout)
```


DESY

Dependency Inversion



Should the sequencer call Constructors of Adjusters or Detectors?

Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

obj.AddDetector(det_ob)

NO!! It would need to know about the specific child classes of Det/Adj Superclass, which we were trying to hide in the first place. → Separate creation & use.

```
mot=TL_DOOCS_Motor('SN_27000839');
```

Dependency Injection:

- The objects our class depends on, must be injected
- The injector knows the specifics (in our case the end user script)

https://en.wikipedia.org/wiki/Dependency_injection

A little recipe

Problem: Close the Sequencer against changing hardware!
Solution: Use abstraction of the hardware.

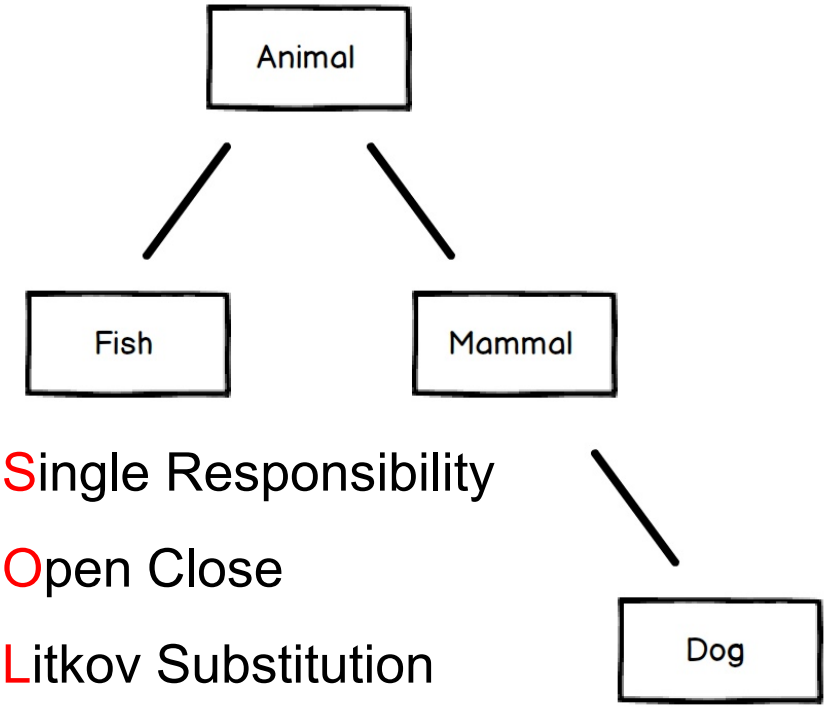
More abstract:
- Suits more possible Detectors
- Useful for more applications
- Can be hard to write

More restrictive :
- Assumptions allow functionality
- May have to change later

1. Take the most complicated Detector(s) and write a class for them.
2. Move aspects to the Detector super class
3. Take an easy “standard” Detector.
4. Derive easy Detector from Superclass
5. Use in same Application
6. Change Superclass as you program application
7. Derive all other needed detectors

Complex problem should be feasible, but adaptions to superclass invisible to easy problem.

General guidelines for class definition



Single Responsibility

Open Close

Litkov Substitution

Interface Segregation

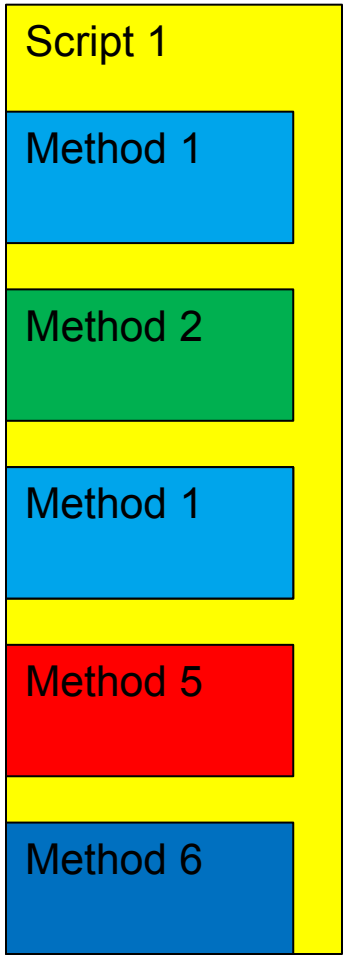
Dependency Inversion

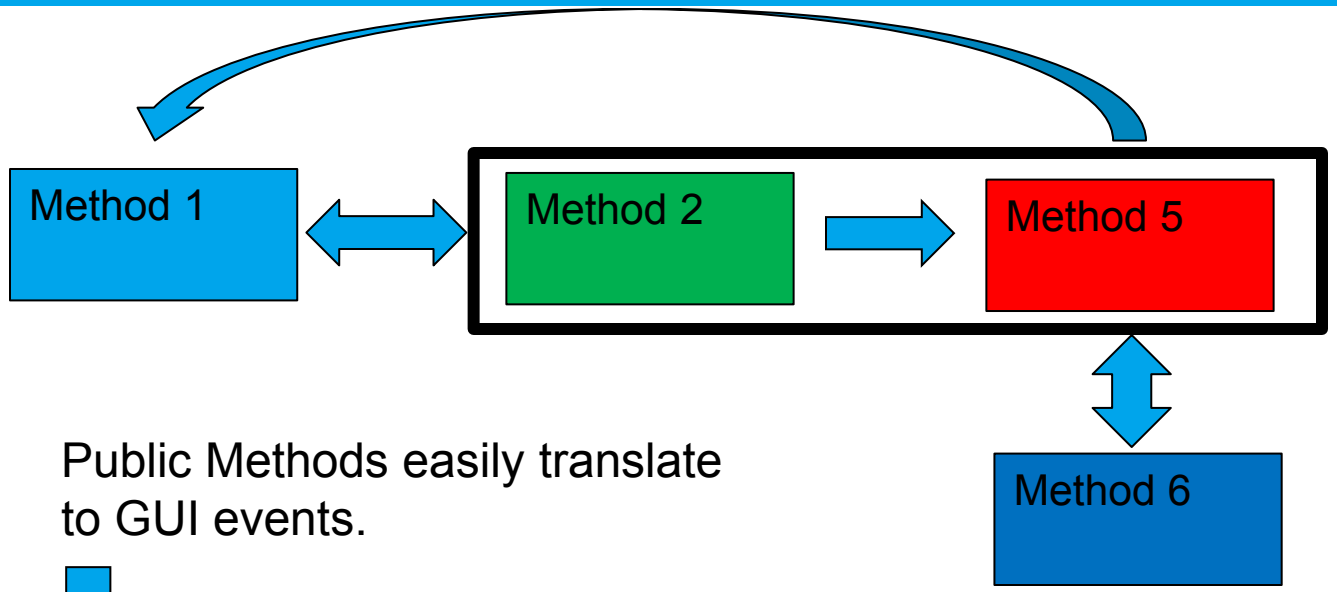
- Respect abstraction layers
  - specific aspects → subclass
  - general aspects → superclass
  - add data accordingly
- Keep function names and behaviors as intuitive as possible
- Announce as many aspects about the use as possible
  - method & property attributes
- Keep scripts, methods & functions below 100 lines of code
- Write every software as if others would use it

Good Software will be reused. Bad software will be rewritten.

DESY

Outlook: Nonlinear Code Execution & GUI





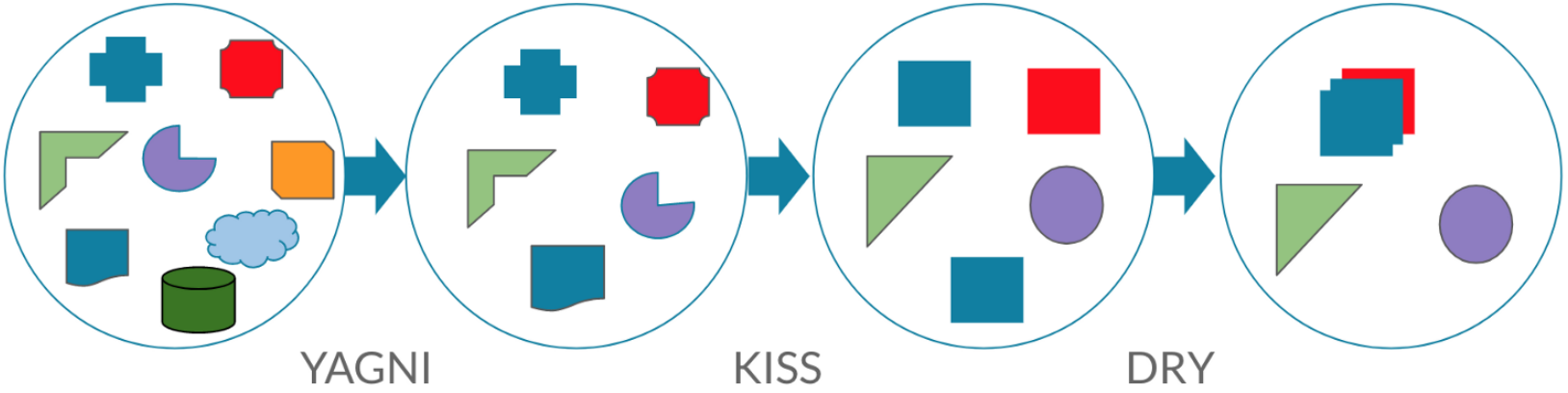
Public Methods easily translate to GUI events.

Public Methods are identical for ancestor classes. (Litkov Subst.)

One GUI per Superclass

→ GUI, Hardware Controller
→ Nonlinear Execution
→ State Machine Concept
→ Object Events

Project Management Advice Part 2



YAGNI
You **a**in't **g**onna **n**eed **i**t

KISS
Keep **i**t **s**imple & **s**tupid

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."—Antoine de Saint-Exupéry

DRY
Don't **r**epeat **y**ourself

- not saying the same multiple times
- avoid repetition
- never state things twice