# Coupling and Cohesion

BRIAN LAMARCHE

COMPUTER SCIENCE 323 – SOFTWARE DESIGN

# What are coupling and cohesion?

Coupling and cohesion are measures of how “modular” and how “functionally similar” your application is.

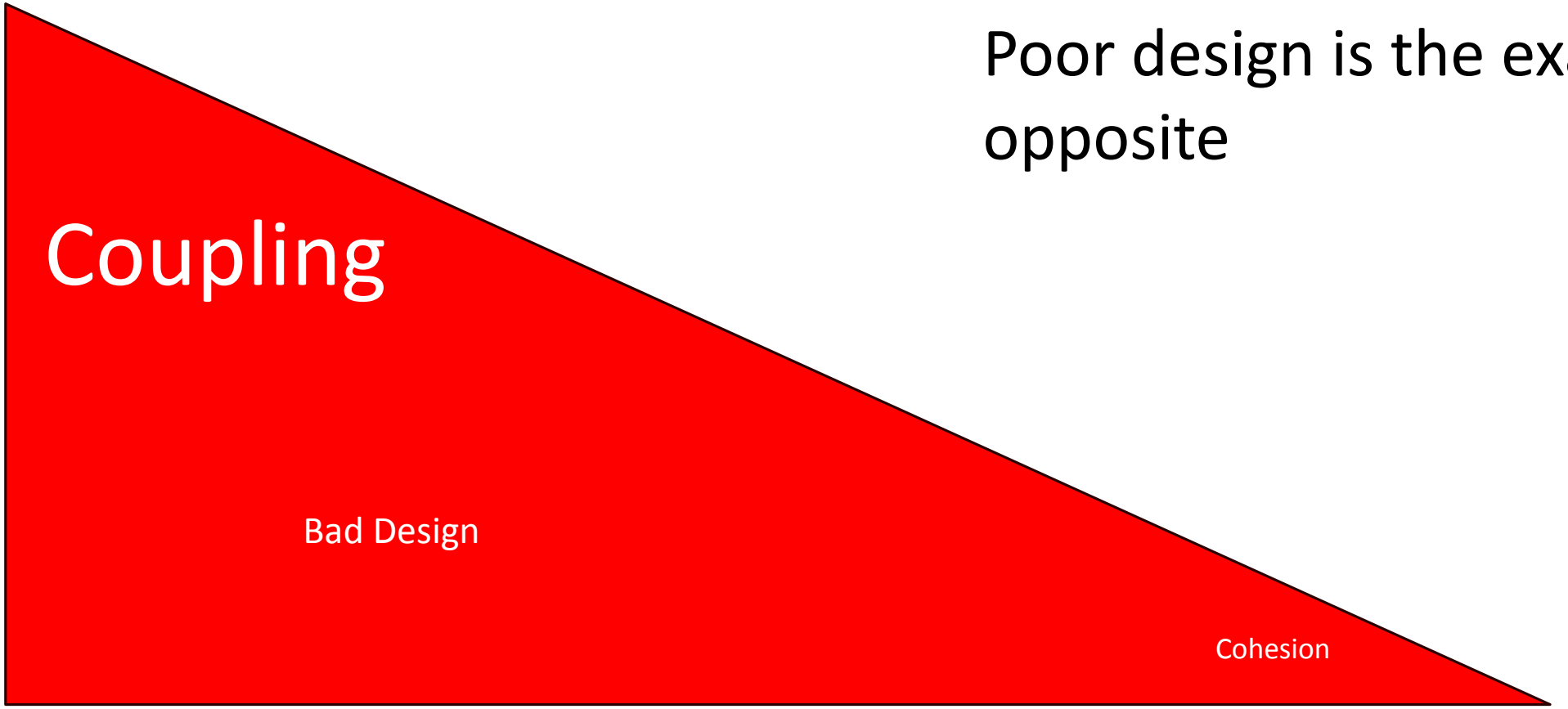
Coupling = modularity

Cohesion = functional similarity

# Design Principles:  What we desire

Good Design is
represented
by objects that are
loosely coupled and have
high cohesion

# Design Principles:  What we desire




Poor design is the exact opposite

# What does it mean?

Think about any light in your house

To illuminate this light, it needs power

But power requires a connection, e.g. wire

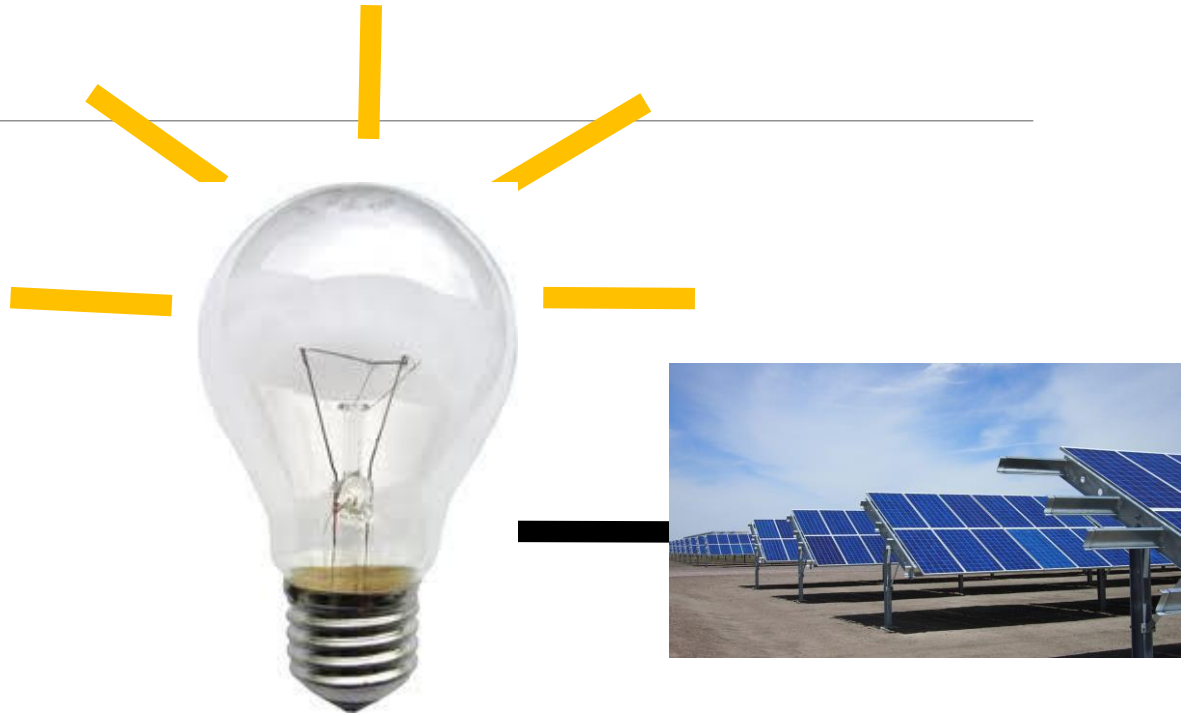
# What does it mean?

Think about any light in your house

To illuminate this light, it needs power

But power requires a connection, e.g. wire

So you wire your light to a power plant



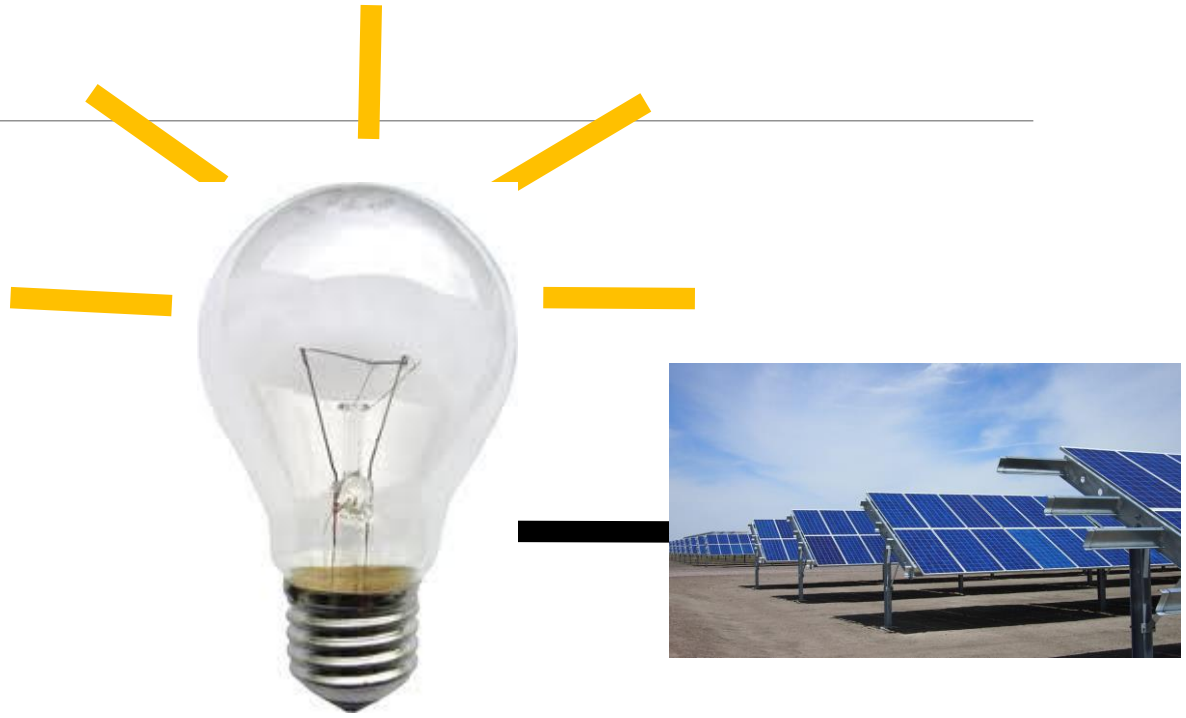
# What does it mean?

Think about any light in your house

To illuminate this light, it needs power

But power requires a connection, e.g. wire

So you wire your light to a power plant



**Modifications**

Now you want to change what kind of light you have

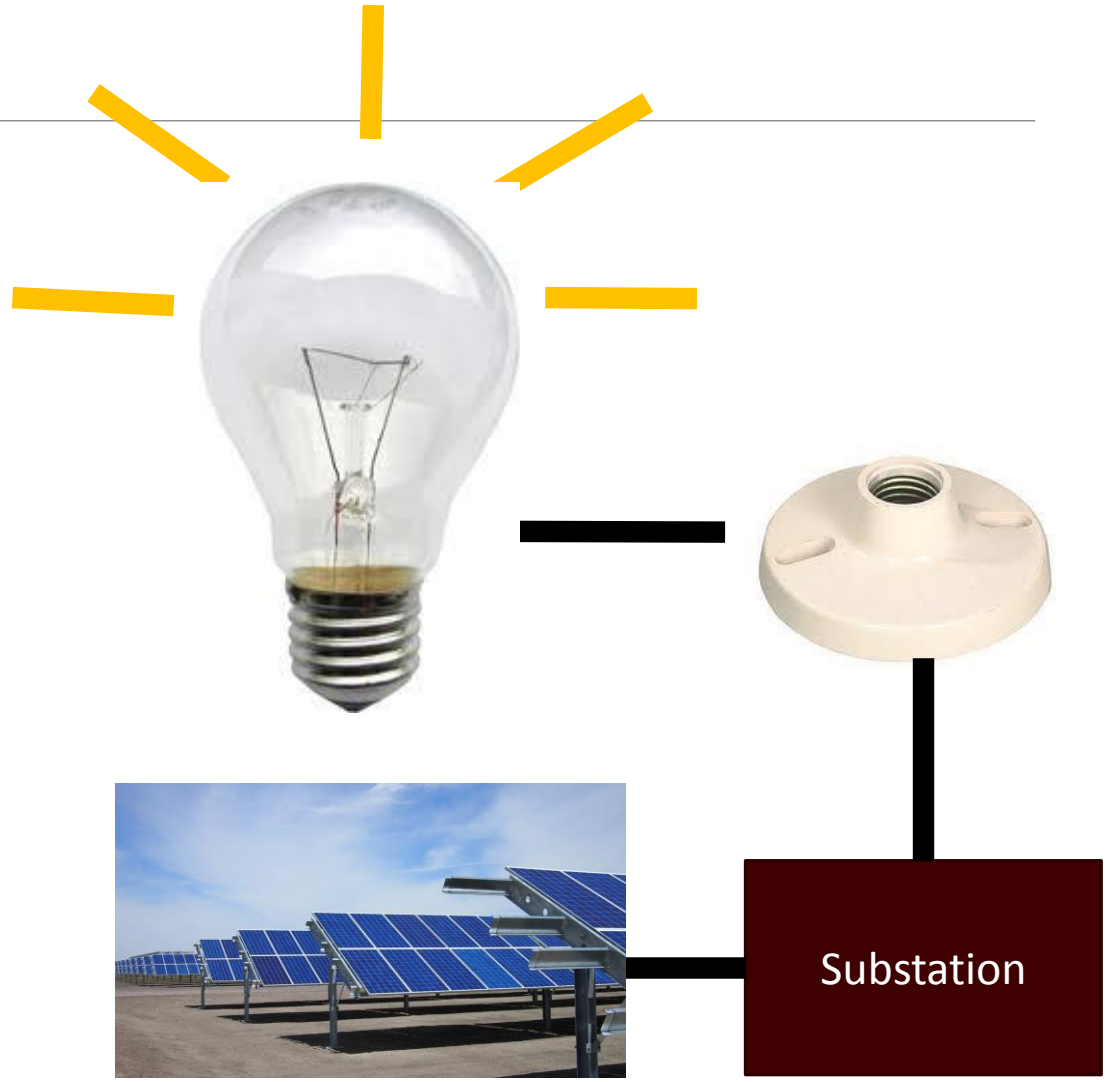
Or you want to change what power plant you connect to

# What does it mean?

This represents a form of coupling

To change the power source, the entire wiring from the light fixture to the power plant was required.

Changes like this are indications of poor design.



# Design Goal

We want to reduce coupling by

- Providing clean object interfaces
- Creating objects that have a unique and specific functionality

XMLFileReader

- Should not read text files

MissileLauncher

- Should not acquire images from a camera

Camera

- Should not detect targets

# Design Goal

We want to reduce coupling by

- Providing clean object interfaces
- Creating objects that have a unique and specific functionality
- **Minimize object relationships!  i.e. minimize their interfaces**

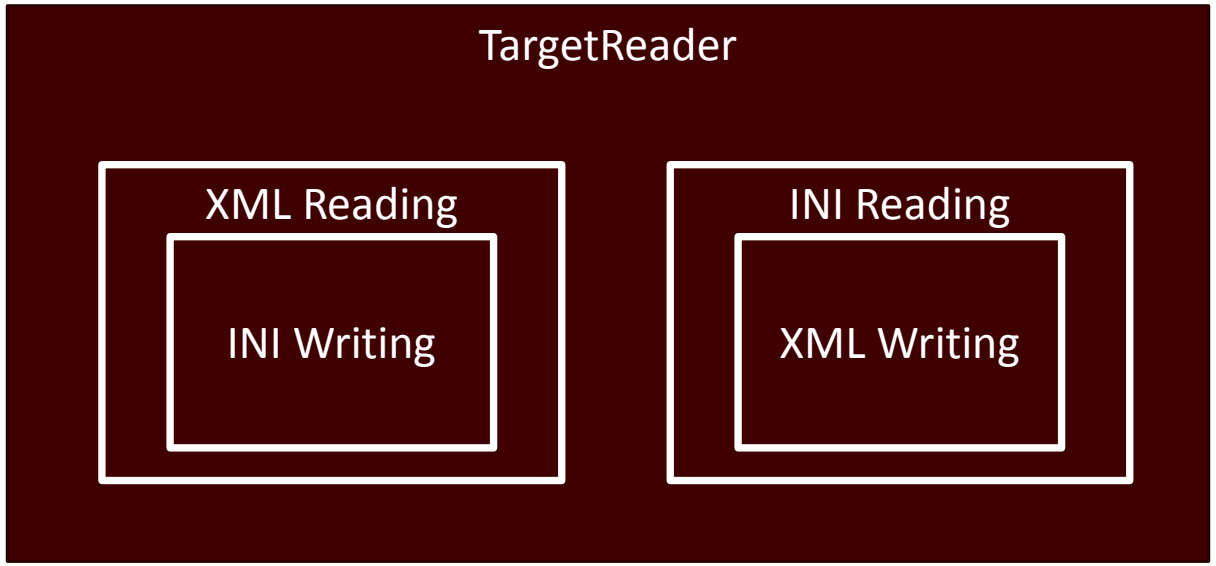
XMLFileReader
- Should read XML files

MissileLauncher
- Should fire missiles and move the turret

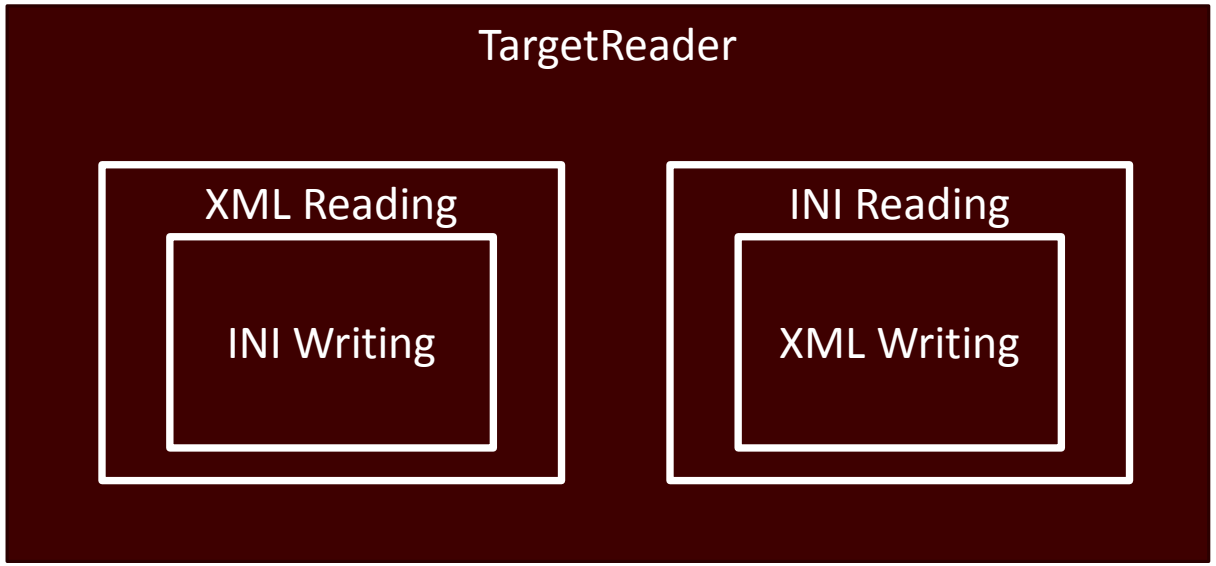
Camera
- Should acquire images

Target Detector
- Should process images for targets

# File Reader Problem



# File Reader Problem

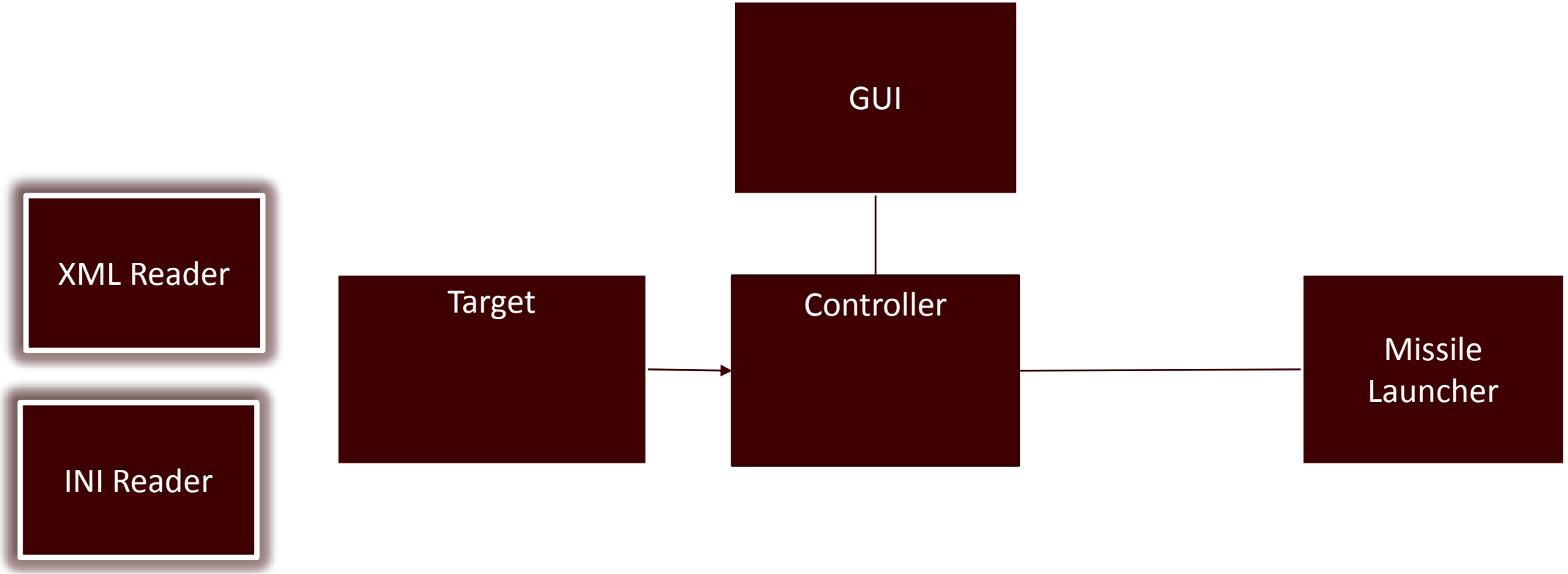



## How to expand?

GUI

Missile Launcher

# Missile Launcher Program



# Coupling Types

There 5 major types

- Data Coupling — Better
- Stamp coupling
- Control coupling
- Common coupling
- Content coupling — BAD

Ranked from loosest to tightest

# Data Coupling

–Two modules talk together with parameters, or single data table (object)

–Unavoidable, you have to communicate via parameters.

```
public bool IsTarget(Point [] outline);
```

# Stamp Coupling

**When two modules rely on the same data even when some of the information is not needed.**

- void FireMissile(Target t)

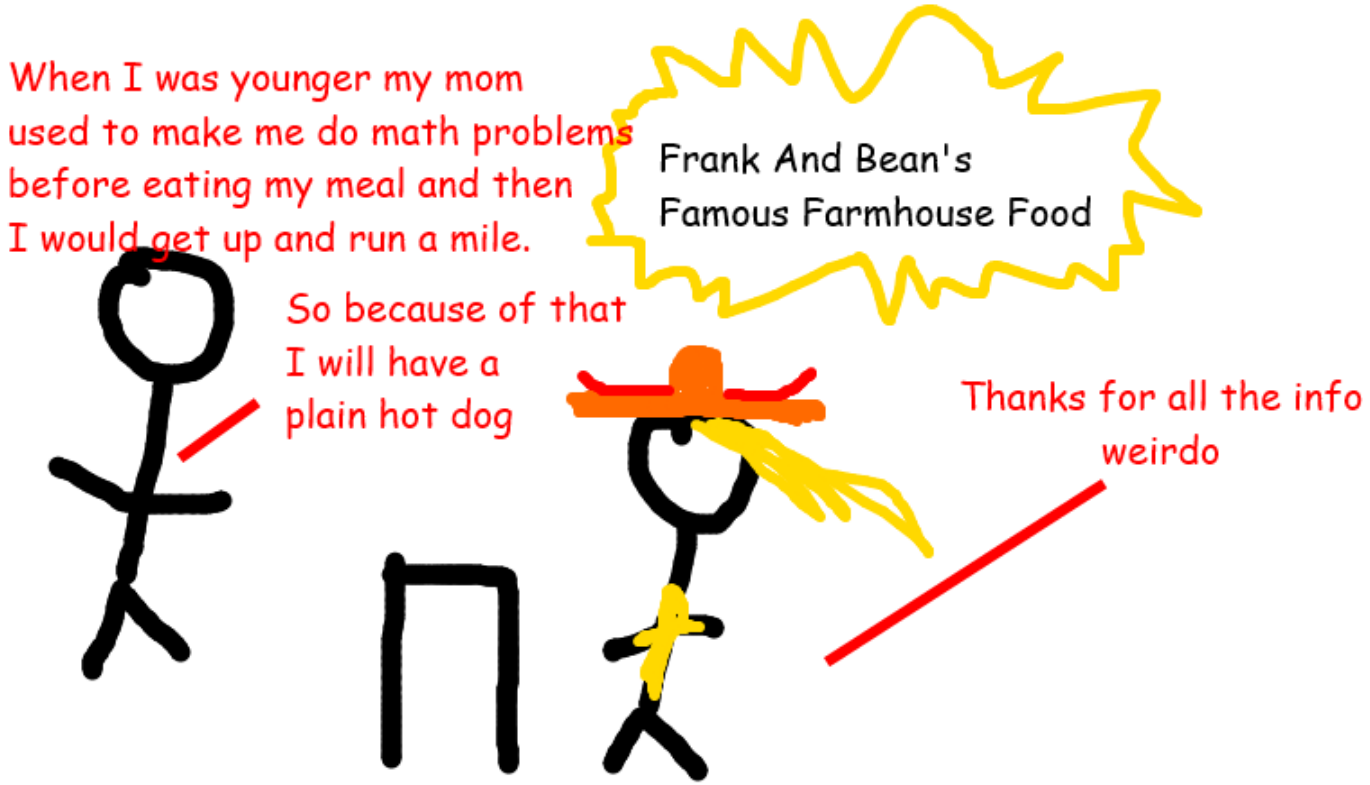
Target has a lot of information

- Color
- Size
- Distance
- Position
- Friend/Foe

If you are firing a missile, why does the missile launcher need to know if it's a friend or foe?

Don't give things more information than they need to know.

# Stamp Coupling



# Stamp Coupling

Moreover, the FireMissile could modify the reference data.

```
public void FireMissile(Target t)
{
        t.Friend = false;
    // what-whoa shaggy
}
```

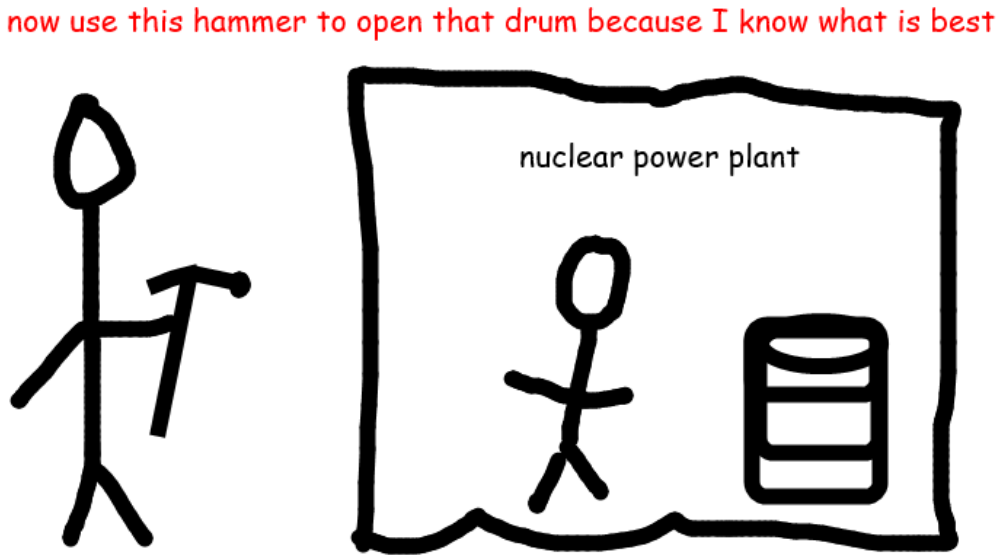
# Control Coupling

When one module passes information to another module to control internal logic.

The point of encapsulation is to abstract and hide underlying complexities from your modules.

You want to reduce any knowledge of how something works internally.

**MAKE YOUR OBJECTS A BLACK BOX**



# Control Coupling

```
Public void FireMissiles(Target t, int numMissiles, int state)
{
        if (state == 0)
        {
                m_hardware.Fire(t.x, t.y);
        }
        else if (state == 1)
        {
                for (int jj = 0; jj < numMissiles; jj++)
                {
                        m_hardware.Fire(t.x, t.y);
                }
        }
}
```

# Control Coupling

Inverted authority is the phenomenon when the subordinate should tell the boss what to do.

```
public enumError FireMissile(Target t)
{
        // returns false if the target was not fired.
        return m_hardware.FireMissile(t.x, t.y);
}
enum enumError
{
        PrintErrorMessage,
        MissileFired
}
```

The subordinate is telling the boss that a missile was not fired. Expecting the boss to print an error message. But there is no gaurantee that anything will happen and the return type is not know.

THROW AN EXCEPTION! Would be an even better approach.

# Control Coupling

Although, not all flags or return types are bad. It's often better, however, to use exceptions.

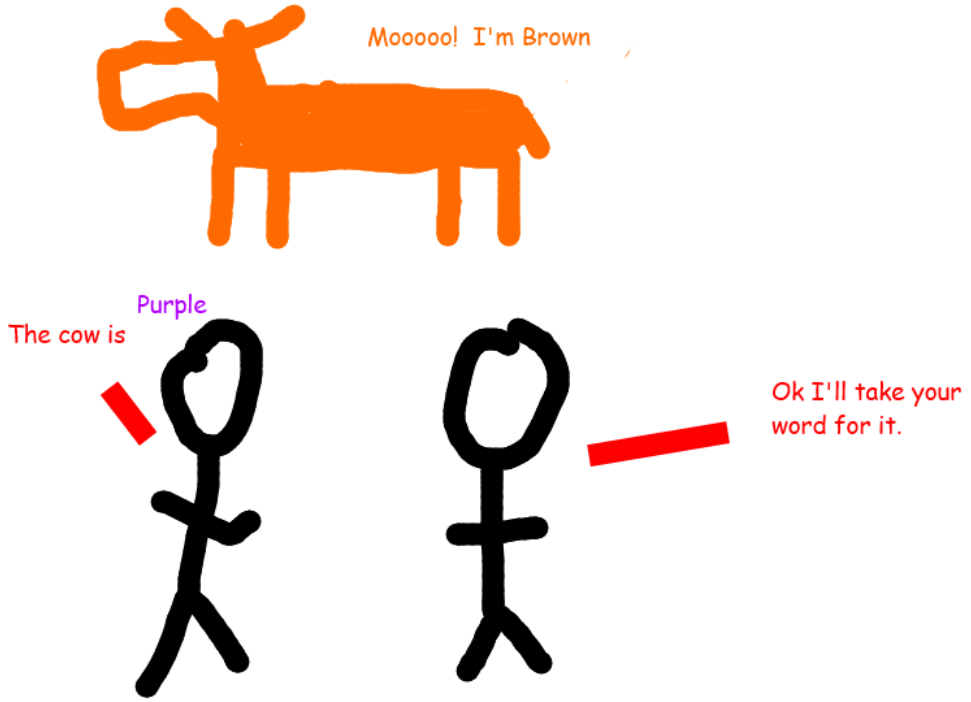
Catch the attention of the other module.

# Common Coupling

Occurs when two modules refer to the same global data.

Potential Problems

- A bug in one module, will be propagated through other modules.
- Modules may use global data to store different pieces of information with no size consistency.



# Content Coupling (WORST!)

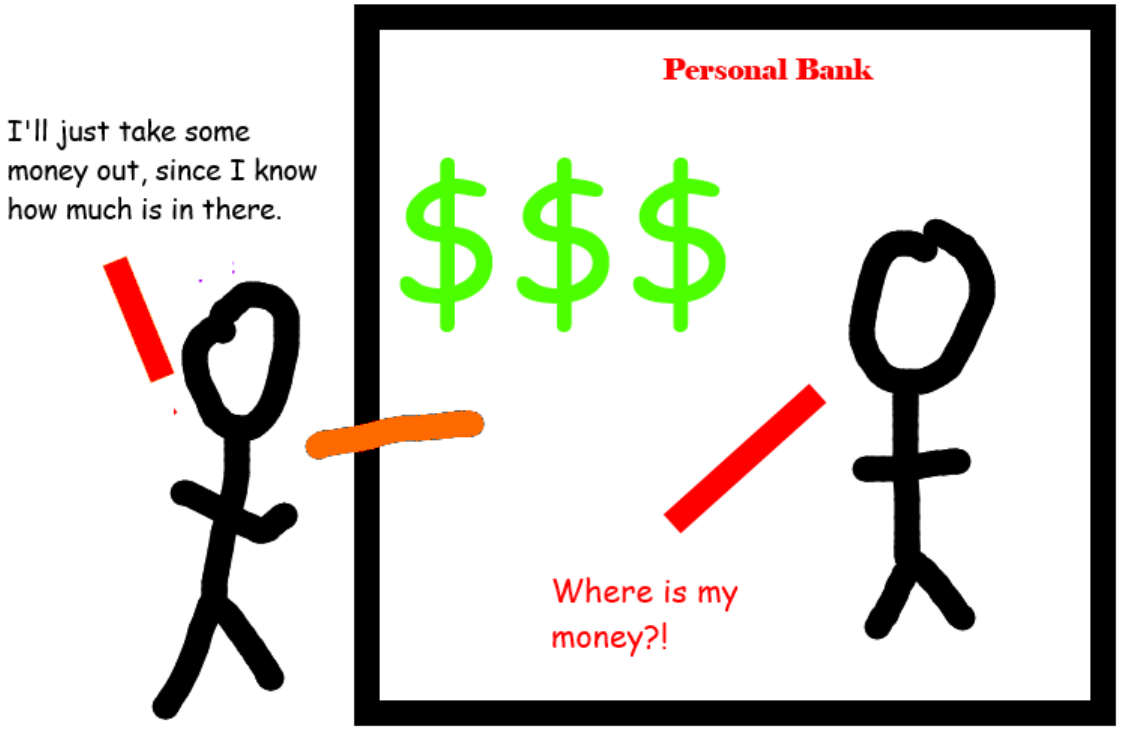
Referring to data inside another module in any way (not publicly exposed).

If one module refers to data in another

If one module changes/alters data in another

This is why we use encapsulation

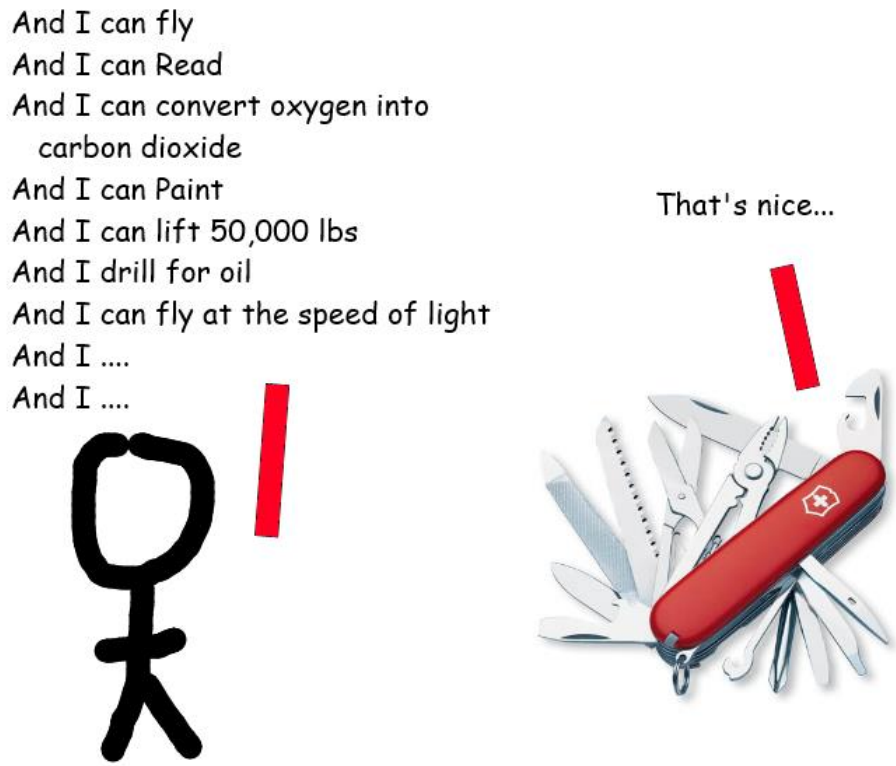
- Getters/Setters (C++, etc)
- Properties (C#)
- Methods
- Make your class variables private.
- Only allow access to private members through controlled access!



# Cohesion

Measure of how functionally relevant methods are within the same module.

- Often inverse of coupling.
- C & C are measures of partitioning.
- Cohesion came to Larry Constantine in the mid 60's.



# Types of Cohesion

Functional

- Sequential
- Communicational
- Procedural
- Temporal
- Logical
- Coincidental

# Functional

Functional elements, methods, focus on solving one related task

- Math
  - –Cosine
  - –Sine
  - –Pow
- String
  - –Trim
  - –Replace
  - –Find
- Simple, modular

# Sequential

Output of one method serves as input to another.

MissileLauncherCamera Object

- Detect Target
- Fire Missiles
- Determine If Hit

These do not belong together in the same module.

Their call sequence, however, does not make sense listed any other way than listed above.

# Communicational

When one or more methods use the same input and/or output to achieve the same task.

TargetDetector

- FindSizeOfTarget
- FindColorOfTarget
- FindShapeOfTarget

Why couldn't there just be *FindTarget* that returns a target object with appropriate data?

TargetDetector could internally call the appropriate private methods, or better yet, detector objects.

# Communicational

Not always hard to maintain, but outside modules may need more information at once.

- Purpose: find Targets!

Still does not make an object tightly coupled

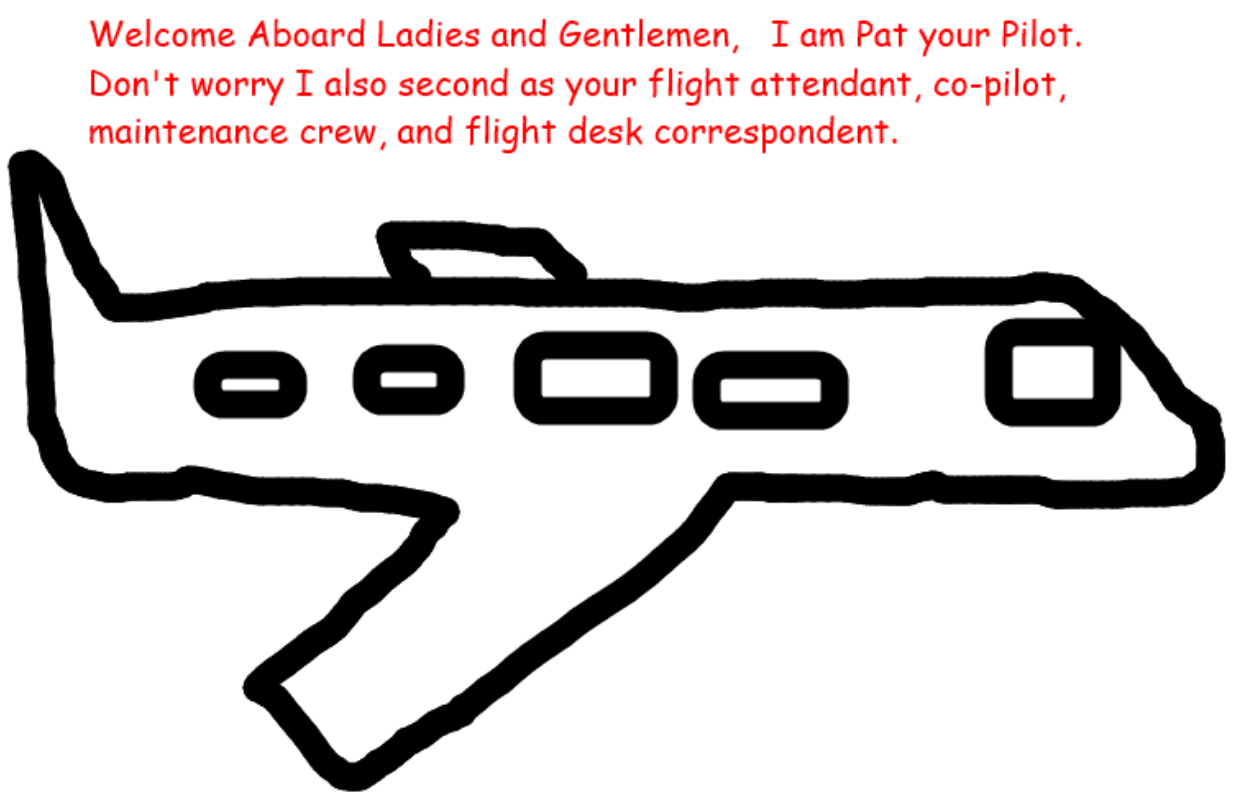
# Procedural

When activities (functions) often cross operational bounds

TargetMissileDetectorAndShooterAndStuff

- FindTargets
- PrepareRockets
- DetermineSpeedOfTarget
- IsFriend
- FireZeMissiles

They often have very little to do with each other.



# Temporal

Methods that are organized usually with little functional relationship, except they happen at the same time.

InitializationClass

- InitializeCamera
- InitializeLauncher
- StartDetection

Something has to coordinate the initialization, true, but building an object that does all of it for the sake of putting in one spot, is not functionally relevant.

# Temporal

The point of temporal cohesion is to realize that although, you may have to do things around the same time, that those functions are not coupled together.

For example:

- IF you have to reinitialize the camera, you shouldn’t have to reinitialize the launcher, target detector, etc. etc.

# Temporal


I've got your hair dryer, shower gel,
stove, and microwave in the moving van still.
Where do you want them?
Put them in my bathtub, because I
use all of those things in the morning around when
I shower.

# Logical Cohesion

When you group the same category of activity into the same module

- Grouping a Kinect and Webcam interface into the same class.
- They both are in charge of acquiring information, but they do it in a different way.
- What if you want to add another interface (keyboard, mouse, voice from bluetooth headset)?

# Coincidental Cohesion

Random grouping of functionality

- E.g. Utility classes.


Worst kind of cohesion.

How was your first homework?

How did you read the file and process the data?

Related to high coupling.

# Cohesion

Low functional cohesion is generally noticeable with Utility classes that group functionality because they have no where else to go.

Often indicative that you just stuck something somewhere in a hurry.

```
public class MathAndStuff
{
        public string Process(string stuff)
        {
                return stuff.Trim();
        }
        public bool CosineTester(int arcData)
        {
                return Math.Cos(arcData) == 1;
        }
}
```

# References

Page-Jones, Meilir. The pratical guide to structured systems design. Yourdin Press. New York, New York. 1980.