Italian C++ Community

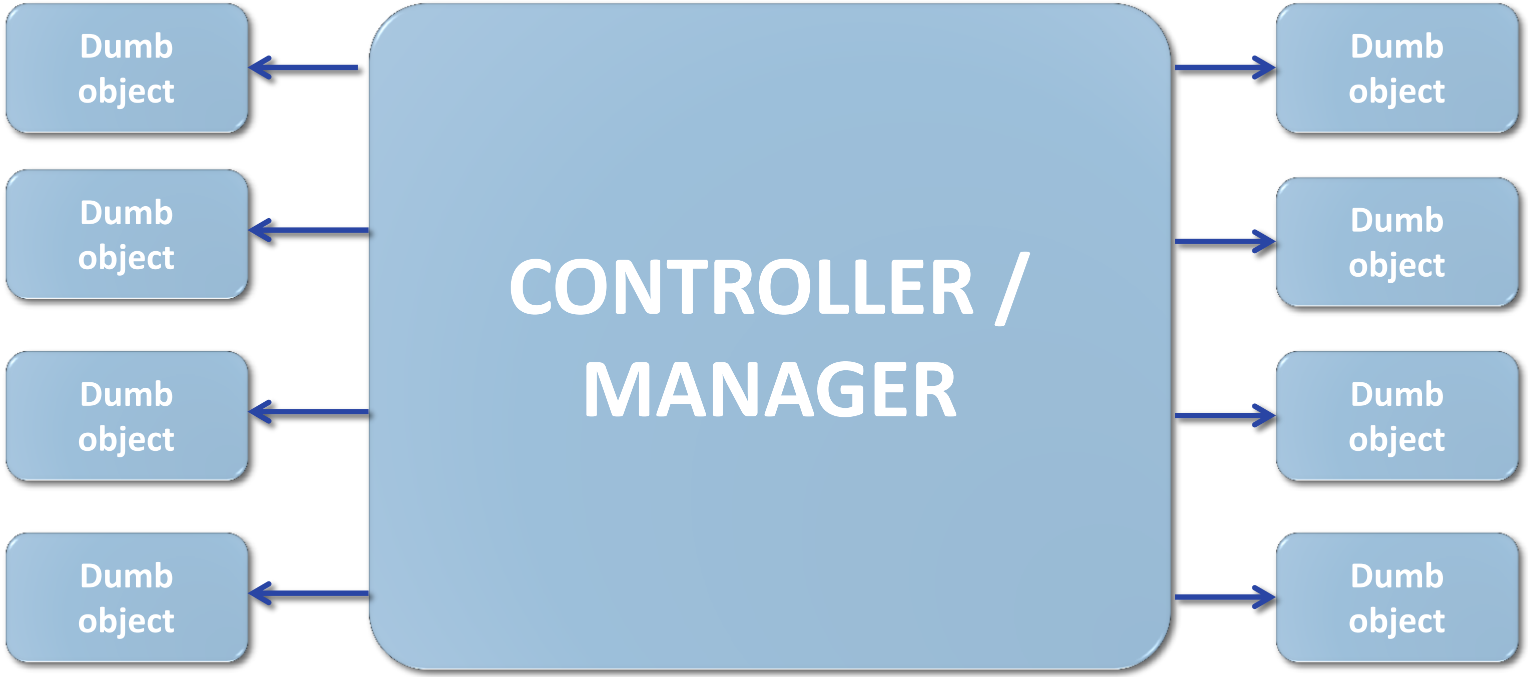
++it

www.italiancpp.org

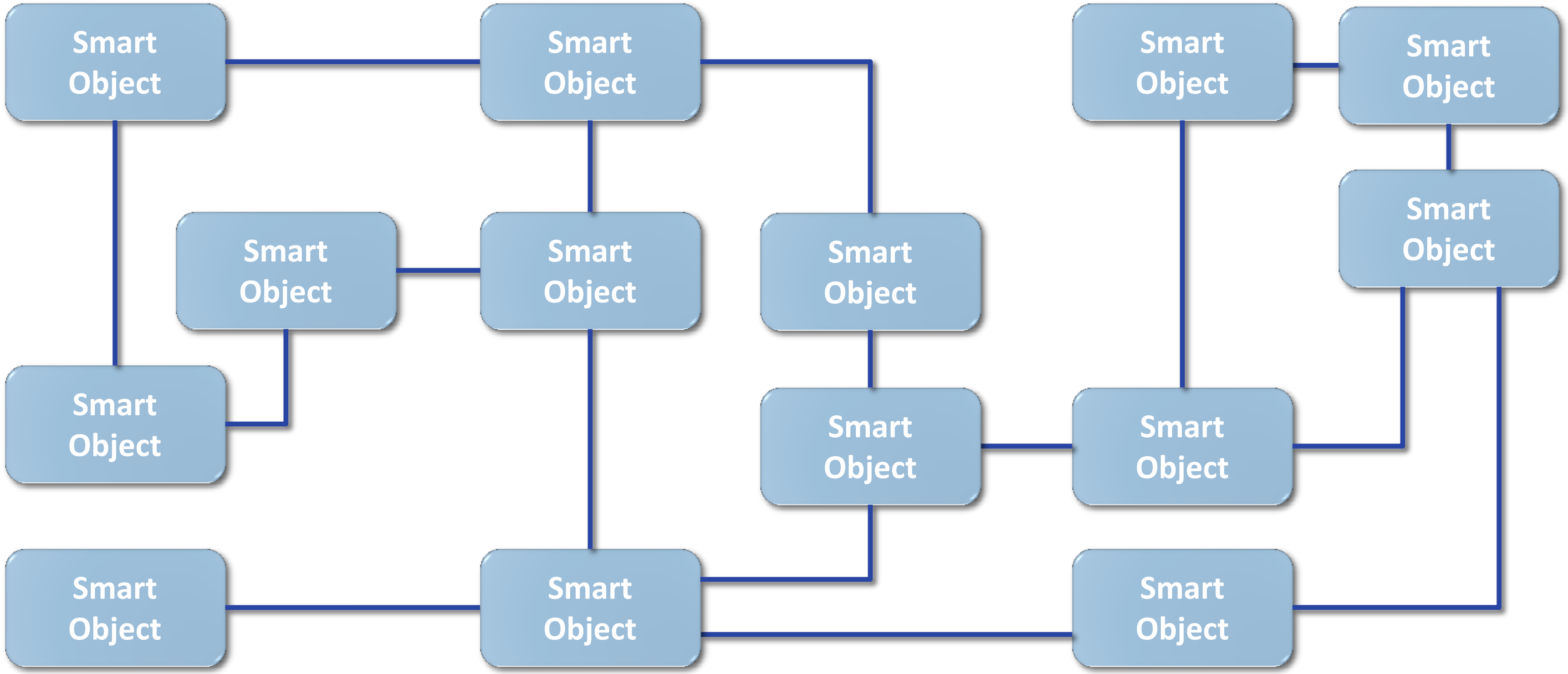
Going native with less coupling

Dependency Injection in C++

OO (?) Design – Style 1



(Real) OO design – Style 2



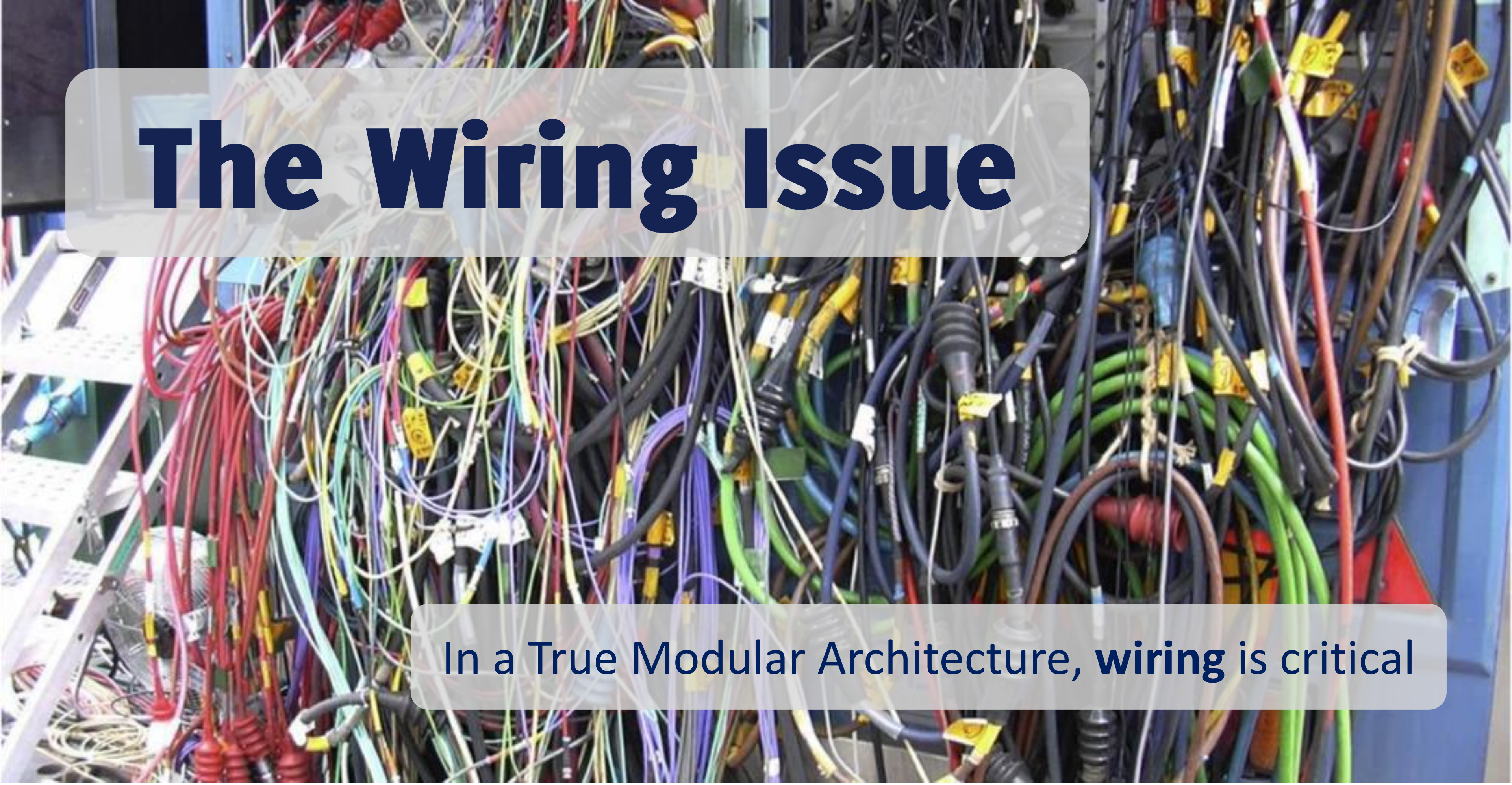
OO Architecture Benefits

Up Scalability (extension)

Down Scalability (contraction)

Reusability

Safety(testability)



The Wiring Issue

In a True Modular Architecture, **wiring** is critical

The Wiring Issue

When you've got a **Good Architecture**, you deal with requirements changes by **adding/removing/substituting** objects

So, we need a simple way to:

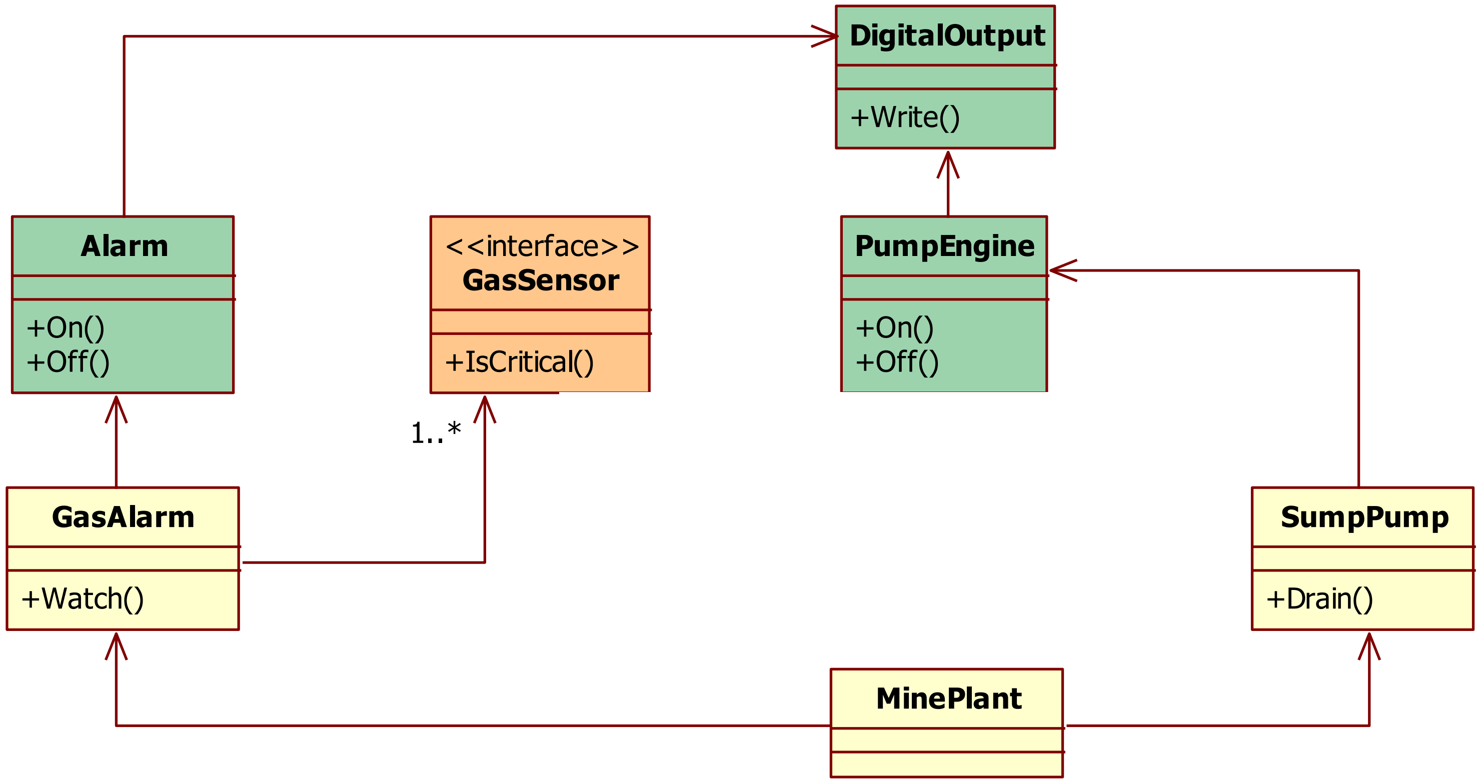
- Change the type of the objects
- Create new objects
- Modify the wiring of the objects

Life **without** a CONTROLLER

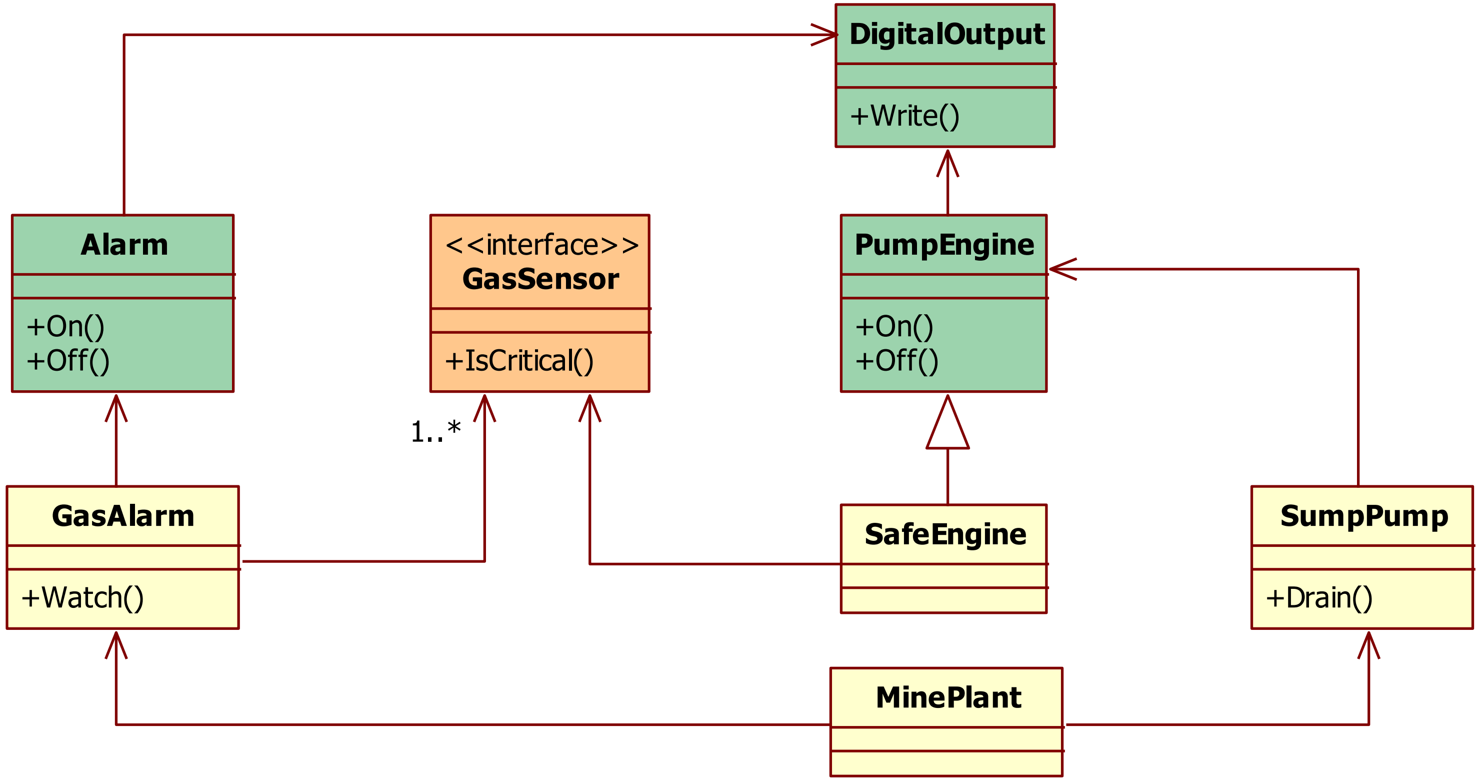
Example taken from:

Carlo Pescio's blog – March 2012

First Design



Requirements Change



Extension

The design is **robust**: I only need to add a class

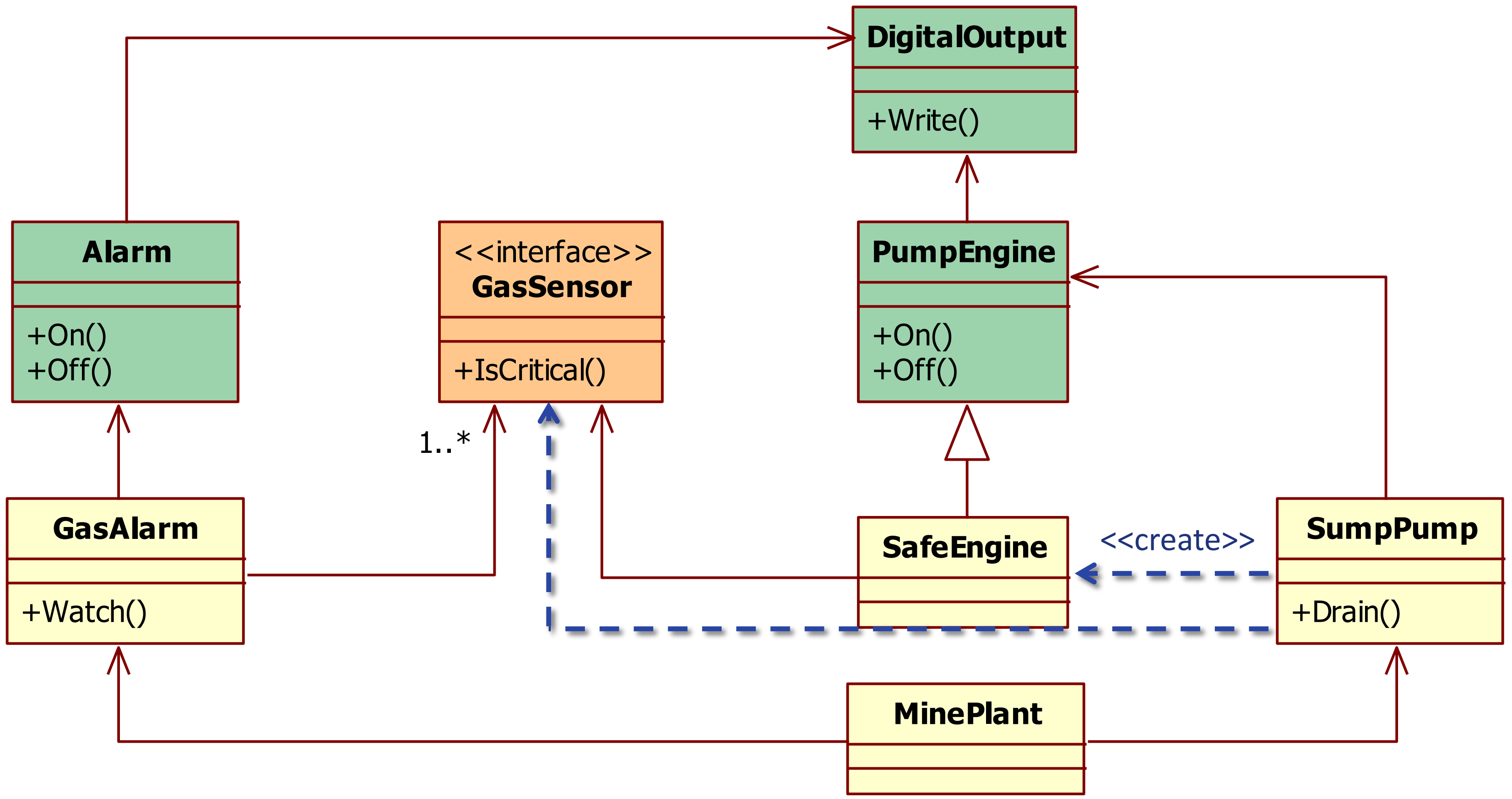
But…

Who creates SafeEngine instead of PumpEngine?

How does SafeEngine get the pointer to the GasSensor (the same used by GasAlarm)?

Solution #1: Local Creation

Each class creates its own dependencies


DigitalOutput
+Write()
Alarm
+On()
+Off()
<<interface>>
GasSensor
+IsCritical()
PumpEngine
+On()
+Off()
1..*
GasAlarm
+Watch()
SafeEngine
<<create>>
SumpPump
+Drain()
MinePlant

Solution #1: Consequences

The **SumpPump** constructor creates a **SafeEngine** instead of a **PumpEngine**.

… but **SafeEngine** needs a pointer to the **GasSensor** instance already used by **GasAlarm**.

So, we must pass it as a parameter to **SumpPump** constructor.

Solution #1: Properties

If I need to change the concrete type, I have to modify the client.

It's difficult to reuse the same client class (even in the same application).

Solution #1: Summary

SafeEngine class added

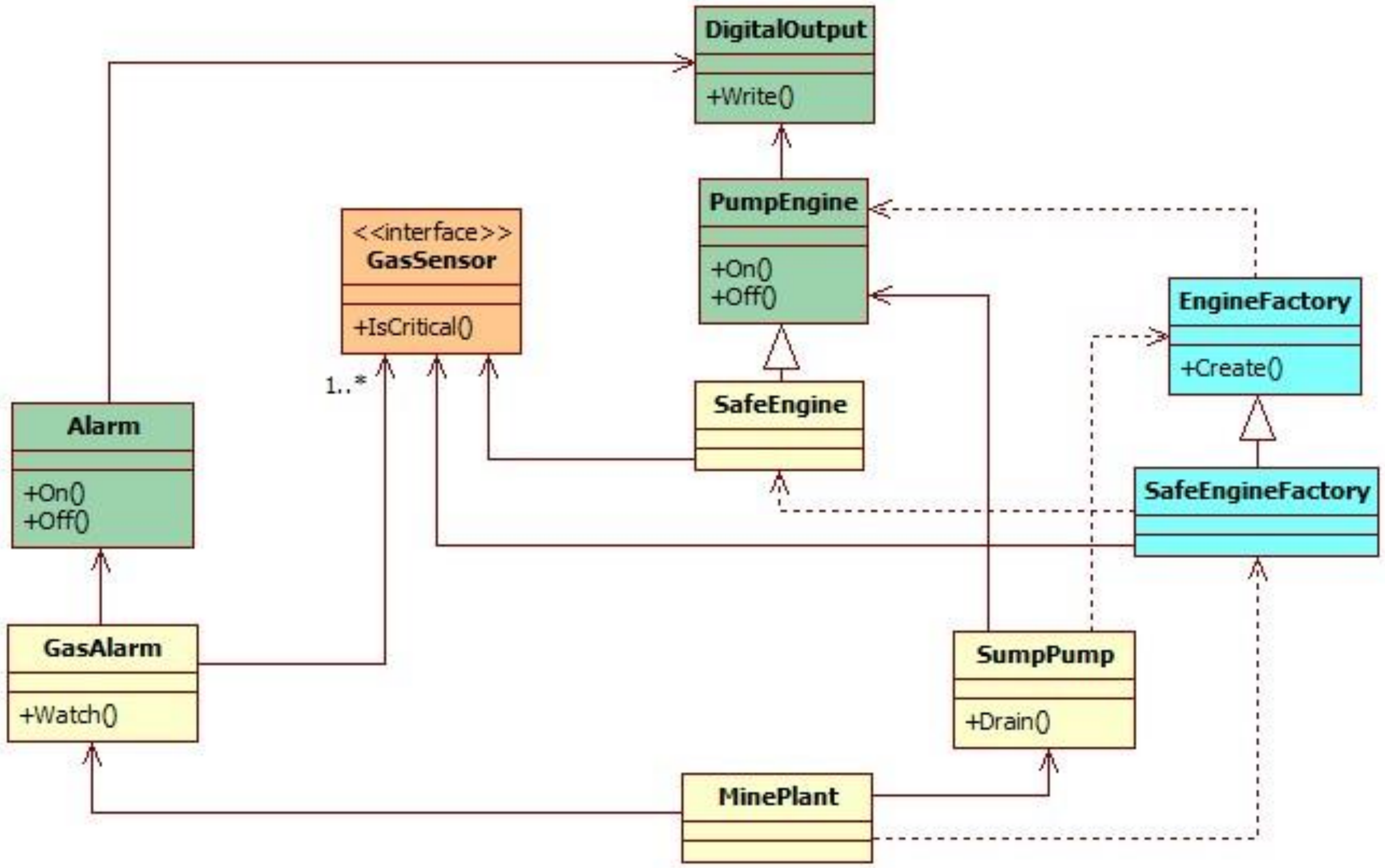
SumpPump constructor modified

MinePlant modified

Solution #2: Factory

(not the GoF factory)

Create objects without exposing the instantiation logic to the client


DigitalOutput
+Write()
PumpEngine
+On()
+Off()
<<interface>>
GasSensor
+IsCritical()
1..*
EngineFactory
+Create()
SafeEngine
Alarm
+On()
+Off()
SafeEngineFactory
GasAlarm
+Watch()
SumpPump
+Drain()
MinePlant

Solution #2: Consequences

The SumpPump constructor takes a Factory as parameter.

PumpEngineFactory instantiates a a PumpEngine.

SafeEngineFactory instantiates a SafeEngine.

SafeEngine still needs a pointer to the GasSensor instance already used by GasAlarm, so we must pass it to the SafeEngineFactory constructor.

Solution #2: Summary

SafeEngine class added

SafeEngineFactory added

MinePlant modified

Solution #3: Service Locator

It’s a registry containing the instances to use

Solution #3: Service Locator

```
class ServiceLocator
{
public:
    ServiceLocator& Instance();
    shared_ptr< PumpEngine > Engine();
    void Engine( const shared_ptr< PumpEngine >& engine );
private:
    ...
};
```

Solution #3: Service Locator

```cpp
// MinePlant:

auto e =
    make_shared< SafeEngine >( engineOutput, gasSensor );
ServiceLocator::Instance().Engine( e );
```

```cpp
// SumpPump:

SumpPump::SumpPump()  :
    engine( ServiceLocator::Instance().Engine() )
{
}
```

Solution #3: Properties

Clients aware of the locator

Dependencies not explicit / evident

Dependencies not checked by compiler

Solution #3: Summary

SafeEngine class added

MinePlant modified

Solution #4: Dependency Injection

“Dependency Injection is when you have something setting the dependencies for you.”

Solution #4: Dependency Injection

Classes don't create their own dependencies

They're passed from outside

Dependency Injection

```
auto gasSensor = ...

auto alarmOutput = make_shared<DigitalOutput>("/dev/ttyS0");
auto alarm = make_shared<Alarm>(alarmOutput);
auto gasAlarm = make_shared<GasAlarm>(gasSensor,alarm);

auto engineOutput = make_shared<DigitalOutput>("/dev/ttyS1");
auto engine = make_shared<PumpEngine>(engineOutput);
auto pump = make_shared<SumpPump>(engine);
```

Dependency Injection

```
auto gasSensor = ...

auto alarmOutput = make_shared<DigitalOutput>("/dev/ttyS0");
auto alarm = make_shared<Alarm>(alarmOutput);
auto gasAlarm = make_shared<GasAlarm>(gasSensor,alarm);

auto engineOutput = make_shared<DigitalOutput>("/dev/ttyS1");
// auto engine = make_shared<PumpEngine>(engineOutput);
auto engine = make_shared<SafeEngine>(engineOutput,gasSensor);

auto pump = make_shared<SumpPump>(engine);
```

Solution #4: Properties

Complete separation between:

application logic (classes)
wiring (main/builder)

Solution #4: Summary

SafeEngine class added

MinePlant modified (one liner)

Can we do BETTER?

SafeEngine must be added anyway.

… can we remove the one liner in MinePlant?

Configuration Driven WIRING

moving creation and wiring outside the code,
in a configuration file

Why?

To easily get extensibility/contraction

(without having to touch zillion files and recompile everything)

From Identifiers to Strings

Improving Previous Solution:

Objects creation from **string**

Objects identified **by name**

Objects connected **by name**

Run-time Reflection Missing…

`Create("Foo")` vs `new Foo`

Enumerate the dependencies

“Inject” the right object address in a class dependency

Solution #5: Dependency Injection +

“Dependency Injection is when you have something setting the dependencies for you.

…this something is usually a framework.”

Existing libraries (C++)

QtIOCContainer

Sauce

DICPP

Hypodermic2

Pococapsule

Main issues:

Compile time injection only

Code generators needed

Enter Wallaroo Library

wallaroo

C++ Dependency Injection

wallaroo.googlecode.com

Creating objects

```
Catalog catalog;
...
catalog.Create("alarmOutput","DigitalOutput","/dev/ttyS0");
catalog.Create("alarm","Alarm");
catalog.Create("gasAlarm","GasAlarm");
catalog.Create("engineOutput","DigitalOutput","/dev/ttyS1");
catalog.Create("pump","SumpPump");
catalog.Create("engine","SafeEngine");
```

Creating objects (from cfg)

```xml
<parts>

  <part>
    <name>pump</name>
    <class>SumpPump</class>
  </part>

  <part>
    <name>engine</name>
    <class>SafeEngine</class>
  </part>

</parts>
```

```cpp
...
Catalog catalog;
XmlConfiguration
    file("wiring.xml");
file.Fill( catalog );
...
```

Object lookup by name

```
shared_ptr< SumpPump > pump = catalog[ "pump" ];
```

Connect Things by name (DSL)

```
Catalog catalog;

// fill catalog
...

use(catalog["alarmOutput"]).as("out").of(catalog["alarm" ]);
use(catalog["safeEngine"]).as("engine").of(catalog["pump"]);
```

Connect Things by name (DSL)

```
Catalog catalog;

// fill catalog
...

wallaroo_within( catalog )
{
    use( "alarmOutput" ).as( "out" ).of( "alarm" );
    use( "safeEngine" ).as( "engine" ).of( "pump" );
}
```

Connect Things by name (from cfg)

```xml
<wiring>

  <wire>
    <source>alarm</source>
    <dest>alarmOutput</dest>
    <collaborator>out</collaborator>
  </wire>

  <wire>
    <source>pump</source>
    <dest>safeEngine</dest>
    <collaborator>engine</collaborator>
  </wire>

</wiring>
```

```cpp
Catalog catalog;
...
XmlConfiguration
     file( "wiring.xml" );
file.Fill( catalog );
catalog.CheckWiring();
...
```

Class Declaration

```
#include "wallaroo/registered.h"
using namespace wallaroo;

class SumpPump : public Part
{
public:
    SumpPump( int id );
private:
    Collaborator< Engine > engine;
};
```

Class Registration

```
WALLAROO_REGISTER( SumpPump, int )

SumpPump::SumpPump( int id ) :
    engine( "engine", RegistrationToken() )
{
...
}

// other methods definition here
...
```

Shared Libraries – AKA plugins (code)

```
Plugin::Load( "safeengine" + Plugin::Suffix() );
// Plugin::Suffix() expands to .dll or .so according to the OS
```

Shared Libraries – AKA plugins (cfg)

```
<plugins>
    <shared>safeengine</shared>
</plugins>
```

```
Catalog catalog;
XmlConfiguration file( "wiring.xml" );
// load the shared libraries specified in the configuration file:
file.LoadPlugins();
file.Fill( catalog );
// throws a WiringError exception if any plug is missed:
catalog.CheckWiring();
```

Collections

```
class Car : public wallaroo::Part
{
    ...
private:
  Collaborator< Engine > engine;
  Collaborator< AirConditioning, optional > airConditioning;
  Collaborator< Airbag, collection > airbags;
  Collaborator< Speaker, collection, std::list > speakers;
  Collaborator< Seat, bounded_collection< 2, 6 > > seats;
};
```

Checks

```
if ( !catalog.IsWiringOk() )
{
        // error handling
}


catalog.CheckWiring()  // throws exception
```

Initialization

```
class Part
{
...
public:
    virtual void Init() {}
...
};

catalog.Init() // calls Part::Init for each part in catalog
```

Wallaroo Internals

WALLAROO_REGISTER declares a static object.

Its constructor creates a factory and puts it in a table, with the class name as key.

Catalog::Create uses the factory to put a new instance in the catalog.

wallaroo::Part has a table of <name, Collaborator>

Wallaroo Internals

shared_ptr< Foo > foo = catalog[“foo”];

catalog[“foo”] returns a class that defines **operator shared_ptr< T >()**

Collaborator uses **weak_ptr** for the dependency

Collaborator defines **operator shared_ptr()** and **operator->()**

Wallaroo Internals

```
wallaroo_within( catalog )
{
    use("alarmOutput").as("out").of("alarm");
    use("safeEngine").as("engine").of("pump");
}
```

```
for (Context c(catalog);c.FirstTime();c.Terminate())
{
    use("alarmOutput").as("out").of("alarm");
    use("safeEngine").as("engine").of("pump");
}
```

Wallaroo Strengths

Lightweight (header file only)

Portable

Type Safe

DSL syntax for object creation and wiring

Configuration driven wiring (xml and json)

Shared library support (plugin)

C++11 or boost interface

No code generators

Design is a balance of forces

Intrusive VS Non Intrusive

Non Intrusive Solutions can manage existing classes but require code generators for configuration driven wiring

Design is a balance of forces

Configuration-driven wiring

VS static type checking

By moving the wiring in a configuration file, we give up the static type checking.

But it’s ok, since you build your system at startup.

Action Points

Real OOD (no controllers / managers)

Manual Dependency Injection

Wallaroo (configuration-drive wiring)

References & Credits

	Me:	@DPallastrelli
	Me:	it.linkedin.com/in/pallad
	Rate me:	https://joind.in/12277
	Wallaroo:	wallaroo.googlecode.com

MinePlant example from Carlo Pescio's blog (http://www.carlopescio.com/2012/03/life-without-controller-case-1.html)

Wiring Picture: By Gael Mace (Own work (Personal photograph)) [CC-BY-3.0 (http://creativecommons.org/licenses/by/3.0)], via Wikimedia Commons