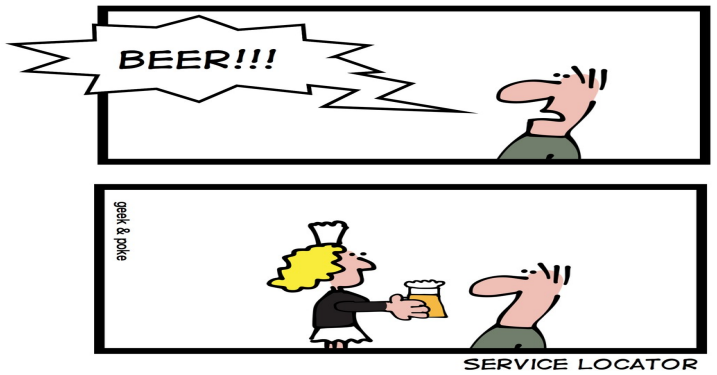
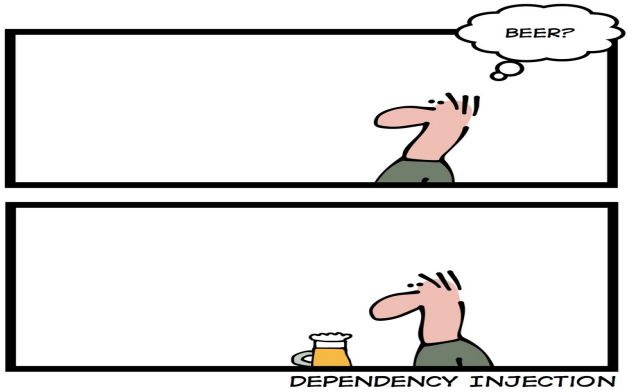
THE GEEK & POKE PATTERN WEEKEND



SERVICE LOCATOR



DEPENDENCY INJECTION

FACTORY

Factory Method

- In our “Abstract factory” design:
- // Factory Method

  Static FinanceToolFactory* createFactory(std::string& country);
- **Abstract Factory: Creates factories**
- **Factory Method: Creates Products**

Desigining an online course module

- C++ Course
  - Info
  - Schedule
- Java Course
  - Info
  - Schedule

Improving Factory Method

- We ended up creating lots of concrete factories
  - How to avoid them?

```
std::shared_ptr<Course> MyCourseFactory::generatecourse(std::string& coursename) {

Course* courseptr = NULL;

   if (coursename == "C++") {
      courseptr = new Cpp;
   }
   else if (coursename == "Java") {
      courseptr = new Java;
   }

  // Evaluating if the course exists
   if(courseptr != nullptr) {
      return std::shared_ptr<Course>(courseptr);
   }
   else {
      std::cerr << "Invalid course name. Program terminating" << std::endl;
      std::exit(EXIT_FAILURE);
   }
}
```

Can we improve it further?

- What are the drawbacks?
  - Everytime we have to compare / alter the factory code.
  - However, we still need to allow the factory to trigger the creation of instances.
  - How to avoid it?
    - Lets make a registry and check against that registry.
    - map<string, function<Course*(void)>> factoryFunctionRegistry;
    - It is a map, keyed on a string with values as functions that return a pointer to an instance of a class based on Course*

- We can then have a method on MyCourseFactory which can add a factory function to the registry:

```
void MyCourseFactory::RegisterFactoryFunction(string name,
function<Course*(void)> classFactoryFunction)
{
    // register the class factory function
    factoryFunctionRegistry[name] = classFactoryFunction;
}
```

The Create method can then be changed as follows:

```
shared_ptr<Course> MyCourseFactory::Create(string name)
{
    Course* instance = nullptr;

    // find name in the registry and call factory method.
    auto it = factoryFunctionRegistry.find(name);
    if(it != factoryFunctionRegistry.end())
        instance = it->second();

    // wrap instance in a shared ptr and return
    if(instance != nullptr)
        return std::shared_ptr<Course>(instance);
    else
        return nullptr;
}
```

- So how do we go about registering the classes in a way that keeps dependencies to a minimum?
- We cannot easily have instances of the derived classes register themselves
  as we can't create instances without the class being registered.
- The fact that we need the
  class registered, not the object gives us a hint that we may need some static variables
  or members to do this.

- Firstly we define a method on MyCourseFactory to obtain the singleton instance:

```
MyCourseFactory * MyCourseFactory::Instance()
{
    static MyCourseFactory factory;
    return &factory;
}
```

We cannot call the following from the global context:

```
MyCourseFactory::Instance()->
    RegisterFactoryFunction(name,
classFactoryFunction);
```

- created a Registrar class that will do the call for us in it's constructor:

```
class Registrar {
public:
    Registrar(std::string className, function<Course*(void)>
classFactoryFunction);
};
...
Registrar::Registrar(std::string name, function<Course*(void)>
classFactoryFunction)
{
    // register the class factory function
    MyCourseFactory::Instance()->RegisterFactoryFunction(name,
classFactoryFunction);
}
```

- Once we have this, we can create static instances of this in the source files of the derived classes as follows (C++):

```
static Registrar registrar(”C++",
[](void) -> Course* { return new Cpp();});
```

- so a quick pre processor define as follows:

```
#define REGISTER_CLASS(NAME, TYPE) \
    static Registrar registrar(NAME, \
        [](void) -> Course* { return new TYPE();});
```

- We then only need add the following to each derived class source file:

```
REGISTER_CLASS(”C++", cpp);
```

We Can Do Better …

- Registrar class into a template class as follows:

```
template<class T>
class Registrar {
public:
    Registrar(string className)
    {
        // register the class factory function
        MyCourseFactory::Instance()->
       RegisterFactoryFunction(name,[](void) -> Course*
            { return new T();});
    }
};
```

- And now we can replace the use of the macro by:

```
static Registrar<Cpp> registrar(”C++")
```

References

- http://blog.jsolutions.co.uk/?p=545
- http://www.codeproject.com/Articles/492900/From-No-Factory-to-Factory-Method