Boost.DI

C++ dependency injection

Andrew Rafas

What is dependency injection

Pushing SOLID principles to the limit

Single Responsibility - 1 function class

Open/closed

Liskov substitution

Interface segregation - 1 function interface

Dependency inversion - depend on abstractions (interfaces)

- Note, that is what TDD advocates advocate.

Typical TDD class

```
struct IDoSomethingable {
	virtual void DoSomething(C c, D d) = 0;
};
class DoSomethinger : public IDoSomethingable {
	IA* _a; IB* _b;
public:
	DoSomethinger(IA* a, IB* b) : _a{a}, _b{b} { ...no new... }
	void DoSomething(C c, D d) override { _a->something(_b, c, d); }
};
```

- Note that this is a first class function or closure (example less<>).

The result of this transformation

- Pro: easy to test
- Pro: easy to mock
- Pro: TDD books claim that this is a more flexible design
- Con: twice as complex
- Con: loses meaning of virtual
- Con: header interfaces
- Con: I claim that TDD leads to bad abstractions
- Con: **lot of instantiating boilerplate**

But that is what we have for **runtime** mocking.

How dependency injection frameworks work

NInject (.NET)

```
class TestModule : Ninject.Modules.NinjectModule {
	public override void Load() {
		Bind<IDoSomethingable>().To<DoSomethinger>();
		Bind<IA>().To<A>(); ...
	}
}
public static void Main() {
	Ninject.IKernel kernel = new StandardKernel(new TestModule());
	var dosomethingable = kernel.Get<IDoSomethingable>();
	dosomethingable.DoSomething(3, 4);
}
```

NInject continued...

- You can create multiple modules (dynamically loaded dlls)
- After resolution, it will create code during runtime (really fast)
- You can modify the binding in tests
- There are constructor & property binding
- There are object creation callbacks & providers
- There is object lifetime (ownership)
- And a lot of other features (RTFM...)

Note that, if dynamic object creation is required, you have to pass around the IKernel instance!!!

Boost.DI

- One day will be included in Boost
- A lot of medium quality documentation
- C++ has no runtime code generation, so bindings will be immutable
- But at least, it is zero overhead (they claim)
- There is build-in type erasure support!!!
  - With unknown performance
  - With unknown declspec(dllexport) behaviour
  - But you can load modules dynamically
  - And you can pass around a type erased injector

Boost.DI continued...

- There is a lot more stuff
    - Singleton, Ambiguous constructor, Scope, etc support
    - A lot of sparsely documented extension for common tasks
- They push typed integer parameters
    - There are supporting Annotations BOOST_DI_INJECT(T, (named = Name) type type_name [= default_value], ...);
    - It can be useful to inject static configuration into objects
    - I think it is a bad idea, better to pass in StaticConfiguration const& config

Questions?