Dependency Injection in C++
A Practical Guide

Peter Muldoon

2024
Dependency Injection in C++
A Practical Guide

C++Now
April 30th, 2024

Pete Muldoon
Senior Engineering Lead

TechAtBloomberg.com
Who Am I?

- Started using C++ professionally in 1991
- Professional Career
  - Systems Analyst & Architect
  - 21 years as a consultant
  - Bloomberg Ticker Plant Engineering Lead
- Talks focus on practical Software Engineering
  - Based in the real world
  - Demonstrate applied principles
  - Take something away and be able to use it
Questions

#include <slide_numbers>
Where will we absolutely not be going?

- All kinds of talk about interfaces and function contracts
- Examine various automatic dependency Injection Frameworks
- Talk is rooted in toy example systems and theory
- Bunch of high brow talk about keeping everything as a passed in parameter
Where will we be going?

• Talk will be about **using** Dependency Injection in applications
• Using various DI methods to achieve functionality swapping / instrumentation for flexibility and testability
• Focus on strategies / How to think to achieve DI in the real world without undue warping of Production Code (or just giving up).
• Talk is rooted in a real world systems not theory
Dependency Injection Myths:

- It’s simple
- Only for simplistic systems or small parts of real systems
- Overkill on smaller projects
- Forget for now, Easy to add in later
- Only for testing *
Pocket Universe Testing:

Setup a (barely) functional full environment:

• Set up toy configs, databases, components, etc to have a usually barely functional “Full environment” and test
• Testing is *not* unit testing as it engages all parts of the system simultaneously aka integration testing.
  ❑ Lack of specificity
  ❑ More difficult setup / error investigation
  ❑ Can lean entirely on regression A/B testing
What is the essence of Dependency Injection?

Data Holders
- Broker
- Security
- Trade

Dependency Injection

Base Implementations
- Pricing
- Sending
- Sizing

Marshalling
- Cancel Trade
- Reprice Trade
- Execute Trade

Component Injection

Bloomberg
Methods to inject different functionality

• linking
Uses Link-time switching of functionality

- Allows limited Testing
- No code changes/contamination in actual production application required
- The code using the dependent functionality has no say in which implementation is being executed.
  - Externally Injected during compilation via LIBPATH or #IFDEF
Twin implementations:

• One for Production
  • Real Functionality linked in with all of the real dependencies

• One for Testing
  • Simple implementation of some Test classes / functions
  • Alternate implementation files (.cpp) live in an alternate/test code branch.
  • One link, one testing scenario
// ActionHandler.cpp
struct ActionHandler {
    Coms coms_;

    void execute(const Action&) {
        Request req;
        //...
        auto result = coms_.send(req);
        check_response(result);
    }
};

// Com.cpp
Result Com::send(const Request& req) {
    ...
    return result;
}

/* Test/Com.cpp */
Request global_req;
Result Com::send(const Request& req) {
    global_req = req;
    return fixed_result;
}
Uses Link-time switching of functionality

• Allows Limited Testing
• No code changes in actual code required

The Drawbacks

• Logistics of unit testing many components impractical/unmanageable
• Undefined Behaviour / ODR violations
• More like Integration testing
• Brittle and confusing

Warning
Don’t Use It
Methods to inject different functionality

- Linking
- Inheritance/virtual functions
Dependency Injection via inheritance

Create a base class interface or extend from an existing Class

• Can handle lots of methods
  • Rich interface
• Well understood mechanism
  • Virtual functions + override
• Easier to add to older codebases
Dependency Injection via inheritance

```cpp
class CalcEngine
{
public:
    virtual bool execute(...);
    virtual bool apply(...);
    virtual bool calculate(...);
    virtual bool commit(...);
};

class TestCalcEngine : public CalcEngine
{
public:
    bool execute(...) override;
    bool apply(...) override;
    bool calculate(...) override;
    bool commit(...) override;
};

bool process(CalcEngine& engine, ...)
{
    // ...
    engine.apply(...);
    // ...
    return engine.calculate(...);
}
```
Dependency Injection via inheritance

CalcEngine

virtual bool execute(...);
virtual bool apply(...);
virtual bool calculate(...);
virtual bool commit(...);

CalcEngineInterface

bool execute(...) override;
bool apply(...) override;
bool calculate(...) override;
bool commit(...) override;

TestCalcEngine

MOCK_METHOD(bool, execute, (...), (override));
MOCK_METHOD(bool, apply, (...), (override));
MOCK_METHOD(bool, calculate, (), (override));
MOCK_METHOD(bool, apply, (...), (override));

bool process(CalcEngineInterface& engine, ...) {
    // ...
    engine.apply(...);
    // ...
    return engine.calculate(...);
}
Dependency Injection via inheritance

Create an interface or extend from Class

• Can handle lots of methods being mocked
  • Rich interface

• Easy to add to older codebases

Drawbacks

• Interface can become messy or has purely test functions added
  • Pure virtual functions can be numerous and a nightmare to stub out
  • Data mixed in with interfaces
  • Uses virtual function table so extra hop
Methods to inject different functionality

• Linking
• Inheritance/virtual functions
• Templates
Create a Class that satisfies the calls made on the class by the function

- Can handle lots of methods being mocked
  - Only need to define the methods actually used
- Compile time so no runtime virtual calls overhead
- Can use concepts(C++20) to define an “interface”
Dependency Injection via templates

```cpp
template<typename CalcEngine>
bool process(CalcEngine& engine) {
    // ...
    engine.apply(rdata);
    rdata.data_ = "2";
    // ...
    return engine.calculate(rdata);
}

struct RealCalcEngine {
    RealCalcEngine(...);

    bool apply(const Data& rdata);
    bool calculate(const Data& rdata);
};

struct TestCalcEngine {
    TestCalcEngine();

    MOCK_METHOD(bool, apply, (const Data&));
    MOCK_METHOD(bool, calculate, (const Data&));
};
```
template<CalcEngineT CalcEngine>
bool process(CalcEngine& engine) {
    // ...
    engine.apply(rdata);
    rdata.data_ = "2";
    // ...
    return engine.calculate(rdata);
}

struct RealCalcEngine {
    RealCalcEngine(...);

    bool apply(const Data& rdata);
    bool calculate(const Data& rdata);
};

struct TestCalcEngine {
    TestCalcEngine();

    MOCK_METHOD(bool, apply, (const Data&));
    MOCK_METHOD(bool, calculate, (const Data&));
};
Create a Class - aka a concept - that satisfies the calls made on the class

- Can handle lots of methods being mocked
- Compile time so no runtime virtual calls overhead

**Drawbacks:**

- Templates all the way down
  - Hard to add in legacy code
- Increased compilation times
- More hieroglyphical
Methods to inject different functionality

- Linking
- Inheritance/virtual functions
- Templates
- Type erasure
Call any thing satisfying a function signature – via std::function/std::move_only_function/std::invoke

• Invokable on any callable target
• Versatile
Dependency Injection via type erasure

```cpp
using CalculateYield = std::function<double(const Data&, ...)>::;

struct YieldProcessor {
    Processor(CalculateYield yield_calc) : YieldCalculator_(std::move(yield_calc)){};
    // ...
    auto process(Data& data) {
        // ...
        auto yield = YieldCalculator_(data, ...);
        // ...
        return yield;
    }
};
```

Dependency Type

Injection

Testing

Dependency
TEST(Processor, test_yield) {

    auto y_calculator = [] (const YieldData& ydata) { return ydata.data_*0.01; };

    YieldProcessor processor(y_calculator);

    YieldData rdata{100};

    auto yield = processor.process(rdata);
    EXPECT_EQ(yield.realised, 1);
}

Dependency Injection via type erasure

Dependency Injection

Testing

Verification
Call any function satisfying a function signature – via
std::function/std::move_only_function/std::invoke

• Invokable on any callable target
• Versatile

Drawbacks
• Can handle only the one method being substituted
• Can’t substitute for a class instance
• Similar overhead to runtime virtual calls
Methods to inject different functionality

- linking
- Inheritance/virtual functions
- Templates
- Type erasure

Null valued objects / stubs
Null Valued Objects

A stub with no functionality - only satisfying the type requirements

• Disables a part(s) of the system not under test
  • Null pointers/std::unique_ptr/std::shared_ptr – with checks – could do the same
  • Null references are illegal, so a stub is needed

• Supplies the correct type but no actual implementation logic
  • Supplied arguments discarded
  • Returns fixed values
Null Valued Objects

DBInterface

virtual bool commit(...) = 0;
virtual bool rollback(...) = 0;
virtual bool statement(...) = 0;

auto process(DBInterface& db, ...) {
  // ...
  db.apply(...);
  // ...
  db.commit(...);
  return results;
}

NullDB

bool commit(...) override { return true; }
bool rollback(...) override { return true; }
bool statement(...) override { return true; }

Disabled via NullDB
Dependency Injection Basics

Methods to inject different functionality

• Linking
• Inheritance/virtual functions
• Templates
• Type erasure

➢ Null valued objects / stubs
Dependency Injection Basics

```cpp
class Com { // Dispatches after execute
  public:
    Result execute(const Action& act); // Sends after execute
};

//派遣after execute
```
Types of Dependency Injection:

- Setter Dependency Injection
class DataProcessor {
    public:
    void setSender(std::unique_ptr<Com> sender) {
        sender_ = std::move(sender);
    }

    bool execute() {
        ...
        sender_->send(...);
        ...
    }

    private:
    std::unique_ptr<Com> sender_;}

Note: Class spends time in an unusable state
Note: Function probably not used in Production
Types of Dependency Injection:

- Setter Dependency Injection
- Method Injection
Dependency Injection Basics

```
execute(...) -> Action
```

```
Com -> send -> Req
```

result
Method Dependency Injection

```cpp
// Dispatches
Response Execute
Com &
Bloomberg

Method Dependency Injection

result
execute(..., &)
Com
Action

send
Com
Req

send
Req
Com

// Dispatches
Response Execute(..., Com&);
```
Method Dependency Injection

bool execute(..., Com& sender)
{
    // ...
    sender.send(...);
    // ...
}
Types of Dependency Injection:

- Setter Dependency Injection
- Method Injection
- Constructor Injection
class Processor {
    public:
        // Constructors
        Processor(...);
    ...
    // Dispatches
    Response Execute(..., Com& com);
    ...
private:
    Com& com_;
};

Dependency Injection Basics

Dependency
Injection

execute(..., &)

send
result

execute(..., &

Com

class Processor {
    public:
        // Constructors
        Processor(..., Com& com);
    // Dispatches
    Response Execute(...);
    ...
    private:
        Com& com_;
    };
Constructor dependency Injection

```cpp
class DataProcessor {
    public:
        DataProcessor(…, Com& sender) : sender_(sender){…}

    bool process(...) {
        // ...
        sender_.send(...);
        // ...
    }

    private:
        Com& sender_;
};
```

**Note**: function signature unchanged

**Note**: Constructor signature has changed
Types of Dependency Injection:

- Setter Dependency Injection
- Method Injection
- Constructor Injection
Conceptual Dependency Injection

Control all Dependencies in a system:

• Identify functional change blocks
  ❑ Allow Injection of flexible functionality
  ❑ Capture inputs, control outputs

• Where to insert Dependencies:
  ❑ Drop all “constant” dependencies into constructors
  ❑ Drop all other dependencies into function method calls

Warning

Real World starts here
Dependency Injection road blocks:

- Objects full creation hidden inside functions/classes
  - No handle to inject new functionality
  - Default class constructors initialized via Singletons/Globals
Object construction isolated inside functions

Class Handler {
    bool processA(Data& data, ...) {
        // ...
        Processor proc(<fixed args>);
        // ...
        return proc.apply(data);
    }

    bool processB(Data& data, ...) {
        // ...
        Processor& proc = ProcessorSingleton::instance->getProcessor(proc_tag);
        // ...
        return proc.apply(...);
    }
};
Dependency Injection road blocks:

- Objects full creation hidden inside functions/classes
  - No handle to inject new functionality
  - Default class constructors initialized via Singletons/Globals
- Reaching through multiple objects
  - Long chains of mock classes needed as boilerplate
  - Breaks the principle of least knowledge
Dependency Injection Hazards

Reaching through multiple objects

```cpp
void Processor::buildQuoteNZFlag(const Side& side) {
    // ...
    const Exch::TickHelper& hp = updater_.processingContext().exchanges().get(side.exchangeNumber()).legacyTickHelper();
    // ...
}
```

**Law of Demeter**: Only talk to your immediate friends
Dependecy Injection road blocks:

- Objects full creation hidden inside functions/classes
  - No handle to inject new functionality
  - Default class constructors initialized via Singletons/Globals

- Reaching through multiple objects
  - Long chains of mock classes needed as boilerplate
  - Breaks the principle of least knowledge

- Having too many dependencies in a class / functional block
  - Impractical to pass large number of Dependencies in constructor / function method
bool execute(DB&, Com&, FileLdr&, Calc&, string, double, string, Cache&, const Data&, ...) {
    // ...
}

Refactoring for DI
struct Bucket {
    DB& db_;  
    Com& com_;  
    FileLdr& ldr_;  
    Calc calc_;  
    string filename_;  
    double multiplier_;  
    string mode_;  
    Cache& cache_;  
    const Data& data_;  
    // ...
};

bool execute(Bucket& bucket)
{
    // ...
}
Applied Dependency Injection

Dependency Injection road blocks:

• Objects full creation hidden inside functions/classes
  □ No handle to inject new functionality
  □ Default class constructors initialized via Singletons/Globals

• Reaching through multiple objects
  □ Long chains of mock classes needed as boilerplate
  □ Breaks the principle of least knowledge

• Having too many dependencies in a class / functional block
  □ Impractical to pass large number of Dependencies in constructor / method

• Classes (hierarchies) packed with huge chunks of functionality
  □ God Classes doing too many things
  □ Many dependencies too numerous to inject
Refactoring for DI

GodLike Class

- verifying
- Pricing
- Saving
- Sizing
- Actions
- Rules
- Business logic
- Sending
Refactoring for DI

GodLike Class

- Pricing
- Sizing
- verifying
- Business logic
- Saving
- Sending

Actions

Rules

✓ Encapsulated functionality
✓ Unit testing
✓ Prelude to DI

Glue Logic
Refactoring for DI

GodLike Class

Gather Information

- Pricing
- Sizing
- Actions

Business Logic

- verifying
- Business logic
- Rules

Distribution Logic

- Saving
- Sending

✓ Unit testing
✓ DI

Glue Logic
Refactoring for DI

Marshaling Class

Gather Information

Business Logic

Distribution Logic
Refactoring for DI

Marshaling Class
- Gather Information
- Business Logic
- Distribution Logic

Gather Information
- Pricing
- Sizing
- Actions

Pricing Class
- Raw Pricing
- Pricing Adjustments
- Aggregate Prices
Dependency Injection roadblocks:

- Objects full creation hidden inside functions/classes
  - No handle to inject new functionality
  - Default class constructors initialized via Singletons/Globals

- Reaching through multiple objects
  - Long chains of mock classes needed as boilerplate
  - Breaks the principle of least knowledge

- Having too many dependencies in a class / functional block
  - Impractical to pass large number of Dependencies in constructor / method

- Classes (hierarchies) packed with huge chunks of functionality
  - God Classes doing too many things
  - Many dependencies to numerous to inject

- Functionality splintered and spread throughout the codebase
  - Fragmented throughout the inheritance chain
  - Duplicated throughout the codebase
  - Blended into general utility classes
Refactoring for DI

Splintered Functionality

Application Class

Verifying

Utility 1

Component

Utility 2

✓ Unit testing
✓ DI

Utility 3

Utility 4

V1

F4

V2

V3
Applied Dependency Injection

Dependency Injection road blocks:

• Objects full creation hidden inside functions/classes
  ❑ No handle to inject new functionality
  ❑ Default class constructors initialized via Singletons/Globals
• Reaching through multiple objects
  ❑ Long chains of mock classes needed as boilerplate
  ❑ Breaks the principle of least knowledge
• Having too many dependencies in a class / functional block
  ❑ Impractical to pass large number of Dependencies in constructor / method
• Classes (hierarchies) packed with huge chunks of functionality
  ❑ God Classes doing too many things
  ❑ Many Dependencies to numerous to inject
• Functionality splintered and spread throughout the codebase
  ❑ Fragmented throughout the inheritance chain
  ❑ Duplicated throughout the codebase
  ❑ Blended into general utility classes
Dependency Injection highway express:

- Object creation done outside the logic of functions
  - Pass in Dependencies directly
  - Pass in Dependency suppliers
- Invoke methods on immediate objects
  - Avoid invoking methods on an object returned by other methods
- Refactor God classes
  - Functionality clustered and pushed into tiered abstraction layers
  - Lessen dependencies
- Refactor fragmented functionality
  - Cluster splintered functionality together
  - Lessen dependencies
Legacy Code: Code that is working in Production for real users

Harder to apply Dependency Injection after code is released in Production

• Code not as malleable / External dependencies
• Large scale complex changes are
  • Riskier
  • Take substantial time
• Preference for Phased/localized changes

Need Tools & Tricks to implement Dependency Injection
Problem: API is used far and wide and so interface cannot be changed
DI for immutable APIs

Method dependency Injection

```cpp
bool process (int key, const std::string&, CalcDep& calc) {defaultCalc} {

    // ...
    calc.estimate(...);
    // ...
}
```

// Forwarding function - deprecated
```cpp
bool process (int key, const std::string& index) {
    process (key, index, defaultCalc);
}
```

Default Injection
Constructor dependency Injection

class DataProcessor {
    DataProcessor (int key, const std::string& index, CalcDep& calc = defaultCalc);
    // ...
};
DI for immutable APIs

Constructor dependency Injection

class DataProcessor {
    DataProcessor (int key, const std::string& index, CalcDep& calc);

    // Delegating constructor – deprecated
    DataProcessor (int key, const std::string& index) : DataProcessor (key, index, defaultCalc){{};

    // ...
};
Problem: API is used far and wide and so interface cannot be changed

Solution: Transparent Dependency Injection using
  • Default arguments
  • Delegating
    ❑ Functions
    ❑ Constructors
Encapsulated functionality: No internal side effects - outside of returned values

- Not using hidden external dependencies.
Problem: DI for Encapsulated functionality

Result

```cpp
compute(const FastData& data,
         const CalcDep& calc) {
  struct MockCalcDep : public CalcDep {
    MOCK_METHOD(bool, fxcalc, (...), (override));
    MOCK_METHOD(bool, eqcalc, (...), (override));
    // ...
  };
  struct MockFastData : public CalcFastData {
    MOCK_METHOD(bool, getData, (...), (override));
    MOCK_METHOD(bool, prime, (...), (override));
    // ...
  };
```
Problem: Encapsulated functionality, no internal side effects - outside of returned values
- Not using hidden external dependencies.

Solution: Mock classes passed in as parameters
Problem: Lazy initialization

- Not able to pass in a constructed object
class Security {
  public

  Security(..., const std::string& index_name)
  : index_name_(index_name){};

  void apply(const ActionX& action) {
    ensureLoaded();
    ...
  }

  private
  void ensureLoaded() {
    if(!db_helper_)
      db_helper_ = createDbHelper(index_name_);
  }

  const std::string index_name_;
  std::unique_ptr<DBHelper> db_helper_;
Phasing DI into Legacy Codebase

class Security {
  public

  Security(..., const std::string& index_name)
      : index_name_(index_name), db_helper_(createDbHelper(index_name_)){};

  void apply(const ActionX& action) {
    ensureLoaded();
    ...
  }

  private

  void ensureLoaded() {
    if (!db_helper_)
      db_helper_ = createDbHelper(index_name_);
  }

  const std::string index_name_;
  std::unique_ptr<DBHelper> db_helper_;
class Security {
  public

  Security(...) : index_name_(index_name){};

  void apply(const ActionX& action) {
    ensureLoaded();
    ...
  }

  bool setDBHelper(std::unique_ptr<DBHelper> dbh) {
    if(!db_helper_) db_helper_ = dbh; ...
  }

  private

  void ensureLoaded() {
    if(!db_helper_)
      db_helper_ = createDbHelper(index_name_);
  }

  std::unique_ptr<DBHelper> db_helper_;
class Security {
    public
    Security(..., const std::string& index_name) : index_name_(index_name){};

    void apply(const ActionX& action) {
        ensureLoaded();
        ...
    }

    private
    void ensureLoaded() {
        if (!db_helper_)
            db_helper_ = createDbHelper(index_name_);
    }

    std::unique_ptr<DBHelper> db_helper_;  
};

using ProvideDBHelper = std::function<
    std::unique_ptr<DBHelper>(const std::string&)>;
Phasing DI into Legacy Codebase

class Security {
    public
    Security(..., const std::string& index_name,
        ProvideDBHelper provide_dbhelper = createDbHelper)
        : index_name_(index_name),
        provide_dbhelper_(provide_dbhelper) {};

    void apply(const ActionX& action) {
        ensureLoaded();
        ...
    }

    private
    void ensureLoaded() {
        if (!db_helper_)
            db_helper_ = provideDbHelper(index_name_);
    }

    ProvideDBHelper provide_dbhelper_;
    std::unique_ptr<DBHelper> db_helper_;
};

using ProvideDBHelper = std::function<
    std::unique_ptr<DBHelper>(const std::string&)>;

Provider Injection

Dependency Provider

Injection
Problem: Dependency injection unexpected snags
Problem: Templated member functions cannot be virtual

class Header {
public:
    virtual double getDividend(double rate) const;
    virtual void setModel(const ModelTag &, int modelID);

    template<typename T>
    TypeNum isType(const T &) const;
    virtual TypeNum Handler::isType(const string &) const;

    // ...
};

class MockHeader : public Header {
    MOCK_METHOD(double, getDividend, (double), (override, const));
    MOCK_METHOD(void, setModel, (const ModelTag &, int), (override));
    MOCK_METHOD(void, isType, (const string &), (override));
    // ???
};

template<typename T>
TypeNum Header::isType(const T & t) const{
    // ...
}

template<>
TypeNum Header::isType(const string & str) const{
    // ...
}
**Problem**: Templated member functions cannot be virtual

```cpp
auto real_typenum = [](const Calc& calc, int val) {
    return calc.isType(val);
};

auto test_typenum = [](const Calc&, int) {
    return TypeNum::A;
};

class Processor {
public:
    using istype_fn = std::function<TypeNum(const Calc&, int)>;
    Processor(is_type_fn istype=real_typenum): istype_(istype) {
        //...
    }

    void apply() {
        //...
        string val = ...;
        Calc calc(...);
        TypeNum typenum = istype_(calc, val);
        //...
    }

    istype_fn istype_;
};
```

*Injection*

*Testing*

*Dependency*
Problem: Templated member functions *cannot* be *virtual*.

But Wait

You didn’t solve the problem.
Dependency Injection Snags

**Problem**: Templated member functions *cannot* be *virtual*

class Processor {
public:
    Processor(...) ;
    { //... }

    template<typename T, CalcEngineT CalcT>
    void apply(T& val, CalcT& calc) {
        //...

        TypeNum typenum = calc.isType(val);
        //...
    }
};
Problem: Templated member functions cannot be virtual

```cpp
class Header {
public:
    virtual double getDividend(double rate) const;
    virtual void setModel(const ModelTag&, int modelID);

    template<typename T>
    TypeNum isType(const T&) const;

    // ...
};

class MockHeader : public Header {
    MOCK_METHOD(double, getDividend, (double), (override, const));
    MOCK_METHOD(void, setModel, (const ModelTag&, int), (override));
    MOCK_METHOD(void, isType, (const string &), (override));
    // ???
};
```
Dependency Injection Snags

Problem: Templated member functions cannot be virtual

class Header {
public:
  virtual double getDividend(double rate) const;
  virtual void setModel(const ModelTag&, int modelID);

  template<typename T>
  TypeNum isType(const T&) const;

  // ...
};

class MockHeader {
  MOCK_METHOD(double, getDividend, (double), (override, const));
  MOCK_METHOD(void, setModel, (const ModelTag&, int), (override));
  template<typename T>
  TypeNum isType(const T&) const;
};
Dependency Injection Snags

Dependency injection unexpected snag

• Templated Functions when using Inheritance
  ❑ Turn into regular function if template is fully specialized
  ❑ For limited types, add type erasure at point of call to template function
  ❑ Move from Inheritance DI to Template DI
Dependency Injection Myths

DI Myths:
- It’s simple
- Only for simplistic systems parts of real systems
- Overkill on smaller projects
- Only for testing
- Forget for now, Easy to add in later
Dependency Injection Truths:

- It's hard, for real Production systems.
- Properly factored code is ultimate KEY to Dependency Injection for anything not a toy example.
- Give weight to local refactoring prior to DI and allow for the extra time.
- Poorly formed code needs many tricks for DI.
- Improves the flexibility/reusability – and so testability - of a system.
- Better long term maintainability of code.
Dependency Injection ultimately boils down to

- Lessening number of dependencies needing injection
  - Horizontal abstraction: Refactoring code into decoupled functional chunks
  - Vertical abstraction: Refactoring code into tiered layers
Other Engineering talks by yours truly:

Retiring The Singleton Pattern: Concrete Suggestions on What to Use Instead
Redesigning Legacy Systems: Keys to success
Managing External APIs in Enterprise Systems
Exceptionally Bad: The Story on the Misuse of Exceptions and How to Do Better
(Exceptions in C++: Better Design Through Analysis of Real World usage)
Godbolt listings

https://godbolt.org/z/M497dsfbT - Link time Dependency injection
https://godbolt.org/z/4hzPnWWcY - Lazy Initialization, no DI
https://godbolt.org/z/e4z58qKoe - Lazy Initialization, proper DI
https://godbolt.org/z/a9TTK9sWb - Inheritance problem with template
https://godbolt.org/z/rhMET79f8 - Inheritance problem with template fixed
https://godbolt.org/z/j6P81eM8Y - Inheritance DI
https://godbolt.org/z/5aro8dKTz - Inheritance Modern Mock
https://godbolt.org/z/6G7MEt9o - template DI
https://godbolt.org/z/EszM1ahW5 - template DI with concepts
https://godbolt.org/z/b95os3r3M - burying templates in constructor only
https://godbolt.org/z/3x3h3Yze4 - Std:move_only_function example
Questions?

Bloomberg is still hiring

Contact : pmuldoon1@Bloomberg.net