Retiring the Singleton Pattern

Concrete suggestions for what to use instead

Bloomberg

Engineering

CppCon 2020
September 18, 2020

Peter Muldoon
Senior Software Developer

TechAtBloomberg.com

Questions

#include <slide_numbers>

What’s currently out there

Google: The Clean Code Talks - "Global State and Singletons"

https://www.youtube.com/watch?v=-FRm3VPhseI

Stack Overflow: What is so bad about Singletons?

“The worst part of this whole topic is that the people who hate singletons rarely give concrete suggestions for what to use instead.”

https://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons

What's currently out there

Google: The Clean Code Talks - "Global State and Singletons"

https://www.youtube.com/watch?v=-FRm3VPhseI

Stack Overflow: What is so bad about Singletons?

"The worst part of this whole topic is that the people who hate singletons rarely give ~~concrete~~ practical suggestions for what to use instead."

https://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons

Talk outline

1. Examine the Singleton Pattern
2. Refactoring out the Singleton in an example function
   - Ensuring the callers of that function do not need to modify code
3. Dealing with non-copyable types
4. Dealing with delayed construction
5. Dealing with phased introduction of the replacement pattern
6. Dealing with initialization order of interdependent Singletons
7. Dealing with groups of Singleton dependencies
8. Stateful groupings of dependencies
9. Review of covered topics & Questions

Classic Singleton

```
class Singleton
{
public:
  static Singleton* instance()
  {
    static Singleton* instance_ = NULL;
    if(!instance_)
      instance_ = new Singleton();
    return instance_;
  }
  void func(...);

private:
  Singleton();
  Singleton(const Singleton&);
  void operator=(const Singleton&);
};
// Somewhere else.cpp
Singleton::instance()->func(...);
```

Notable Characteristics

- Single Global instance of a Type
- Is globally accessible
- Holds a Global state that's mutable and tied to program lifetime
- Initialization is out of your control (private constructor, assignment)

Classic Singleton

```
class Singleton
{
public:
  static Singleton* instance()
  {
    static Singleton* instance_ = NULL;
    if(!instance_)
      instance_ = new Singleton();
    return instance_;
  }
  void func(...);
  static void init(...);
private:
  Singleton();
  Singleton(const Singleton&);
  void operator=(const Singleton&);
};
// Somewhere else.cpp
Singleton::instance()->func(...);
```

Notable Characteristics

- Single Global instance of a Type
- Is globally accessible
- Holds a Global state that's mutable and tied to program lifetime
- Initialization is out of your control (private constructor, assignment)

Drawbacks of a Singleton

- Acts as hidden dependencies in functions that use it
- No dependency injection for testing
- Initialization is out of your control
- Multiple runs can yield different results
- Usually in groups and may need initialization calls in a particular order to setup other singletons it depends on
- State is tied to program lifetime – frequently function calls in a particular order are necessary

Reasons given for using a Singleton anyway

- Passing parameters up & down long function call chains can be daunting so it's easier to have a global grab bag
- Other user groups using a long established API in legacy codebase are unwilling to change their function calls
- Efficiency, I only create one of them and reuse it

The point of a Singleton should ***not*** be to grant global access to a value, but to control the instantiation of a type

However it's frequently used for easy access

Singleton or Not ?

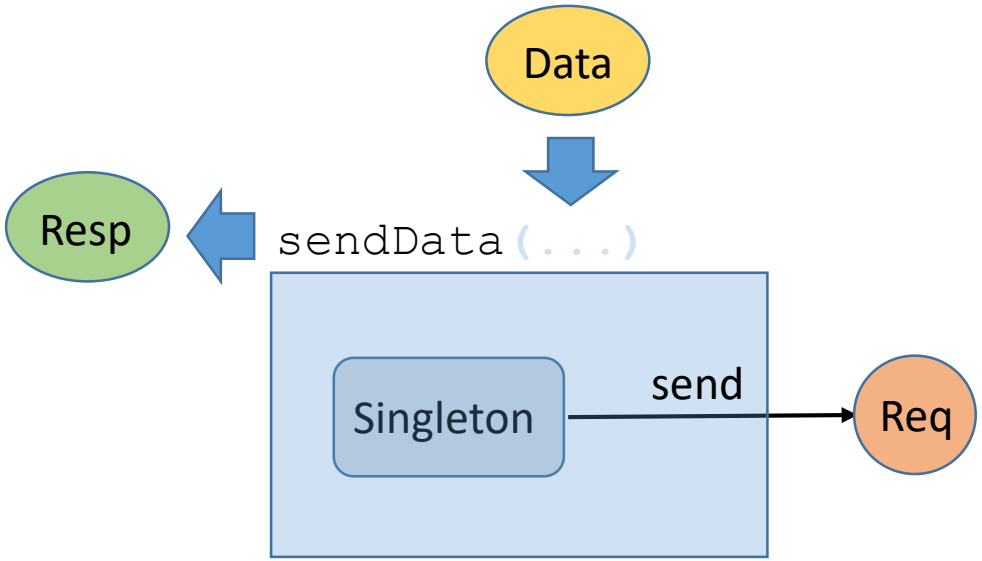
```
// in my_limits.h

extern int ThrottleLimit;

int getThreadLimit();
```

```
// in my_limits.cpp
int ThrottleLimit = 100;

int getThreadLimit()
{ static int y = figureLimit(); return y;}
```

None are singletons aka a type that can only have one instance.

```
#include <ios>
#include <streambuf>
#include <istream>
#include <ostream>


namespace std {
  extern istream cin;
  extern ostream cout;
  extern ostream cerr;
  extern ostream clog;
  extern wistream wcin;
  extern wostream wcout;
  extern wostream wcerr;
  extern wostream wclog;
}
```

```
// Original Code with singleton in processor.cpp
Response sendData(const Data& data)
{
    Request req;
    // Transform Data into Request
    // .....
    return CommSingleton::instance()->send(req);
}
```



```
// Original Code with singleton in processor.cpp
Response sendData(const Data& data)
{
    Request req;
    // Transform Data into Request
    // .....
    return CommSingleton::instance()->send(req);
}
```

Minimum requirements to remove the hidden Singleton call

- New function must be - at least - source compatible
- Express the involvement of outside agencies
- Allow dependency injection for testing purposes

```
// processor.h
Response sendData(const Data& data,
                  CommSingleton* comms = CommSingleton::instance());


// processor.cpp
Response sendData(const Data& data, CommSingleton* comms)
{
    Request req;
    // Transform Data into Request
    // .....
    return comms->send(req);
}
```

Minimum requirements to remove the hidden Singleton call

- New function must be source compatible ✓
- Express the involvement of an outside agency ✓
- Allow dependency injection for testing purposes ✗

```
// New wrapper class to replace singleton - CommWrapper.h
class CommWrapper
{
    enum { SERVICE_ID = 249409 };

public :
    CommWrapper(int service_id = SERVICE_ID);
    Response send(const Request& req);

private:
    TcpClient  raw_client;
};

struct Service {
    static CommWrapper comm_;
};
```

```
// in processor.h
Response sendData(const Data& data, CommWrapper& comms=Service::comm_)


// Completed transformation of original function with backwards compatible
non-singleton version
// in processor.cpp
Response sendData(const Data& data, CommWrapper& comms)
{
    Request req;
    // Transform Data into Request
    // .....
    return comms.send(req);
}
```

Can we now test via dependency injection ? ✘

```
// New wrapper class to replace singleton CommWrapper.h
class CommWrapper
{
    enum { SERVICE_ID = 249409 };

public :
    CommWrapper(int service_id = SERVICE_ID);
    Response send(const Request& req);

private:
    TcpClient  raw_client;
};
```

```
// New wrapper class to replace singleton CommWrapper.h
class CommWrapper
{
    enum { SERVICE_ID = 249409 };

public :
    CommWrapper(int service_id = SERVICE_ID):raw_client(service_id){...};
    virtual Response send(const Request& req);

private:
    TcpClient  raw_client;
};
```

```
// in processor.h
Response sendData(const Data& data, CommWrapper& comms=Service::comm_)
```

```
// Completed transformation of original function with backwards compatible
non-singleton version
// in processor.cpp
Response sendData(const Data& data, CommWrapper& comms)
{
    Request req;
    // Transform Data into Request
    // .....
    return comms.send(req);
}
```

Can we now test via dependency injection ? ✓

```
class CommTester : public CommWrapper
{
    public:
    CommTester(Request& req) : req_(req){}
    Response send(const Request& req){req_ = req; return Response();}

    Request& req_;
};
int TestSendData()
{
    Data rec;
    rec.id = 999;
    // Fill in more rec values ...
    Request req;

    CommTester a_client(req);
    sendData(rec, a_client);
    if(req.senderId_ != rec.id)
        std::cout << "Error ..." << std::endl;

    // Further validation of rec values ...
}
```

```
class MockClient : public CommWrapper
{
    public:
    MOCK_METHOD1(send, Response(const Request&));
};

TEST(XTest, sendData)
{
    MockClient a_client;
    Response resp;
    Request req;

    EXPECT_CALL(a_client, send(_)).WillOnce(DoAll(SaveArg<0>(&req),
                  return(resp)));
    Data rec;
    rec.id = 999;
    // Fill in more rec values ....

    sendData(rec, a_client);

    ASSERT_EQ(req.senderId_, rec.id);
    // Further validation of rec values ...
}
```

Modern C++

```
// in processor.h
using comms_func = std::function<Response(Request)>;

Response sendData(const Data& data, comms_func comms=Service::comm_)


// in processor.cpp
Response sendData(const Data& data, comms_func comms)
{
    Request req;
    // Transform Data into Request
    // ...
    return comms(std::move(req));
}
```

Possible Problem - A copy has been introduced

Modern C++

```
// in processor.h
using comms_func = std::function<Response(Request)>;

Response sendData(const Data& data, comms_func comms = std::ref(Service::comm_)
```

```
// in processor.cpp
Response sendData(const Data& data, comms_func comms)
{
    Request req;
    // Transform Data into Request
    // ...
    return comms(std::move(req));
}
```

Modern C++

```
// New wrapper class to replace singleton CommWrapper.h
class CommWrapper
{
    enum { SERVICE_ID = 249409 };

public :
    CommWrapper(int service_id = SERVICE_ID):raw_client(service_id){...};
    Response operator()(Request req);

private:
    TcpClient  raw_client;
};

struct Service {
    static CommWrapper comm_;
};
```

```
struct MockClient
{
    MOCK_METHOD1(send, Response(Request));
    Response operator()(Request req) { return send(std::move(req)); }
};

TEST(XTest, sendData)
{
    MockClient a_client;

    Data rec;
    rec.id = 999;
    // Fill in more rec values ....
    Response resp;
    Request req;

    EXPECT_CALL(a_client, send(_)).WillOnce(DoAll(SaveArg<0>(&req),
                  Return(resp)));

    sendData(rec, std::ref(a_client));

    ASSERT_EQ(req.senderId_, rec.id);
    // Further validation of rec values ...
}
```

Modern C++

```
// in processor.h
using comms_func = std::function<Response(Request)>;

Response sendData(const Data& data, comms_func comms = std::ref(Service::comm_);


// in processor.cpp
Response sendData(const Data& data, comms_func comms)
{
    Request req;
    // Transform Data into Request
    // ...
    return comms(std::move(req));
}
```

Modern C++ - Better performance

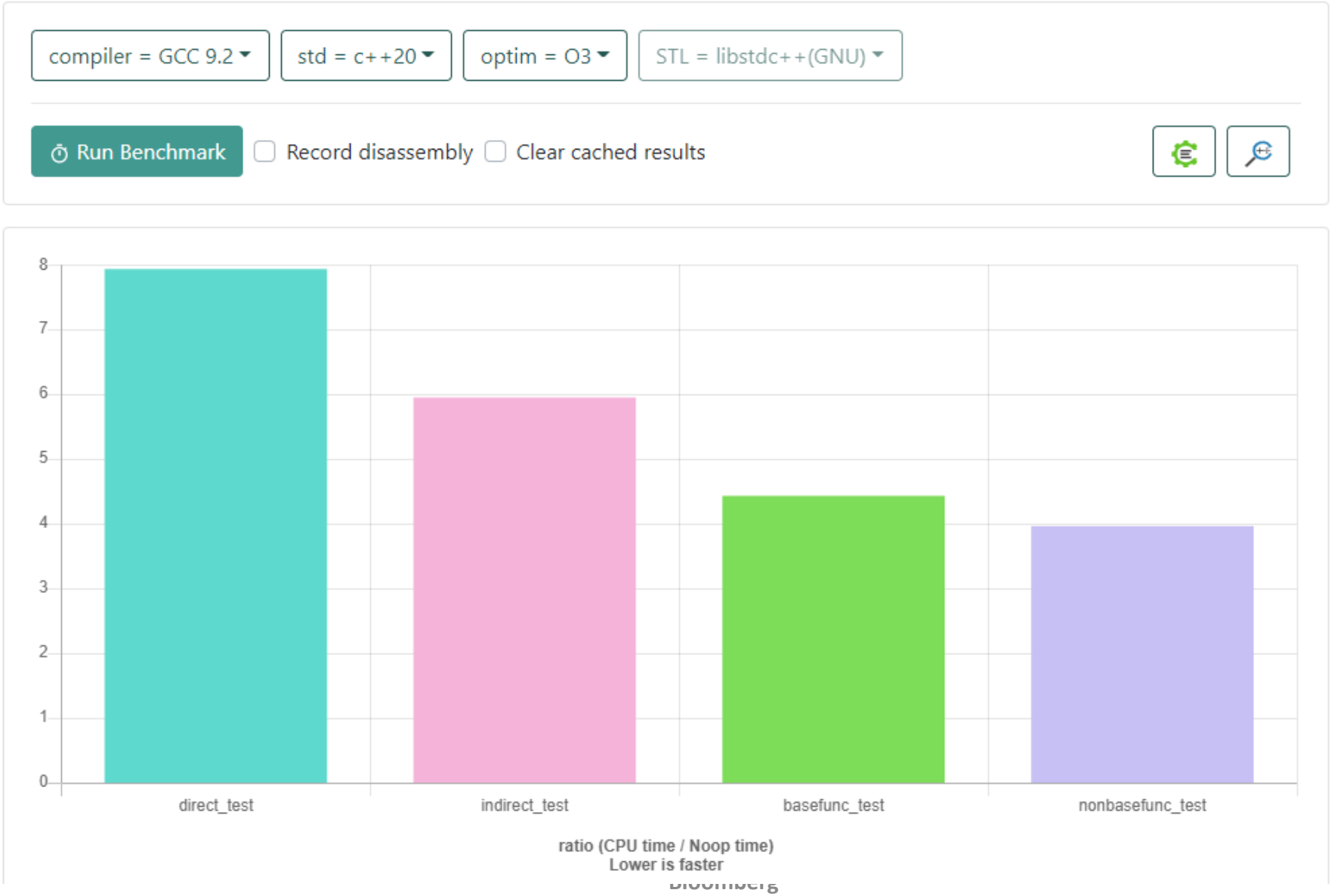
```
class CommWrapperImpl
{
public :
    CommWrapperImpl();
    Response operator()(const Request& req);
};

using comms_func = std::function<Response(Request)>;
class CommWrapper
{
public :
    CommWrapper(comms_func sender):sender(std::move(sender)){};
    Response operator()(const Request& req) { return sender(req); };
private:
    comms_func sender;
};

struct AutoClient {
    static CommWrapper comm_(CommWrapperImpl());
};
```

Modern C++ - Better performance

```
Response sendData(const Data& data, CommWrapper& comms = Service::comm_)

// in processor.cpp
Response sendData(const Data& data, comms_func comms)
{
    Request req;
    // Transform Data into Request
    // ...
    return comms(std::move(req));
}
```

Method Performance(noipa, -O1)

Method Performance(noipa, -O3)



Preserving The Application Binary Interface (ABI)

```
// in processor.h
Response sendData(const Data& data, CommWrapper& comms=Service::comm_)

// Completed transformation of original function with backwards compatible
// non-singleton version in processor.cpp
Response sendData(const Data& data, CommWrapper& comms)
{
    Request req;
    // Transform Data into Request
    // .....
    return comms.send(req);
}
```

- So far, new function is source compatible via unchanged API
  - requires recompile of application
- Shipping shared libraries, requires function signatures to be stable

Preserving The Application Binary Interface (ABI)

```
// New overload that replaces singleton
Response sendData(const Data& data, CommWrapper& comms)
{
    Request req;
    // Transform Data into Request
    // .....
    return comms.send(req);
}

// keep original signature
Response sendData(const Data& data)
{
    return sendData(data, Service::comm_);
}
```

```
// Holds default wrapper class to replace singleton.
struct Service {
    static CommWrapper comm_;
};
```

Potential problem : Default instance is created before main runs.

- There may be static dependencies across TUs
- Some setup initialization may occur prior to this code being usable

Need to delay creation of default instance post main() start preferably using lazy initialization

Lazy Initialization – pre C++11

```
// CommWrapper.cpp

static CommWrapper* comm_ = NULL;

// Lazy Initialization
CommWrapper& getDefaultComms()
{
    COMPILER_DO_ONCE {
      comm_ = new CommWrapper(arg1, arg2, ...);
    }
    return *comm_;
}
```

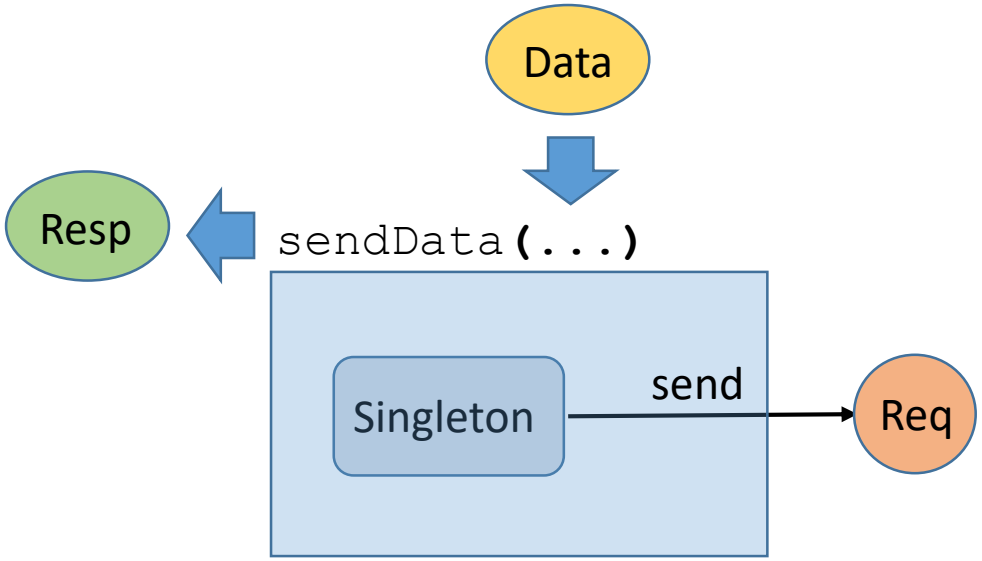
Lazy Initialization – Modern C++

```
// comm_wrapper.h
CommWrapper& getDefaultComms();



// comm_wrapper.cpp
struct Service {
    CommWrapper comm_;
};

// Lazy Initialization
CommWrapper& getDefaultComms() {
    static Service client;
    return client.comm_;
}
```

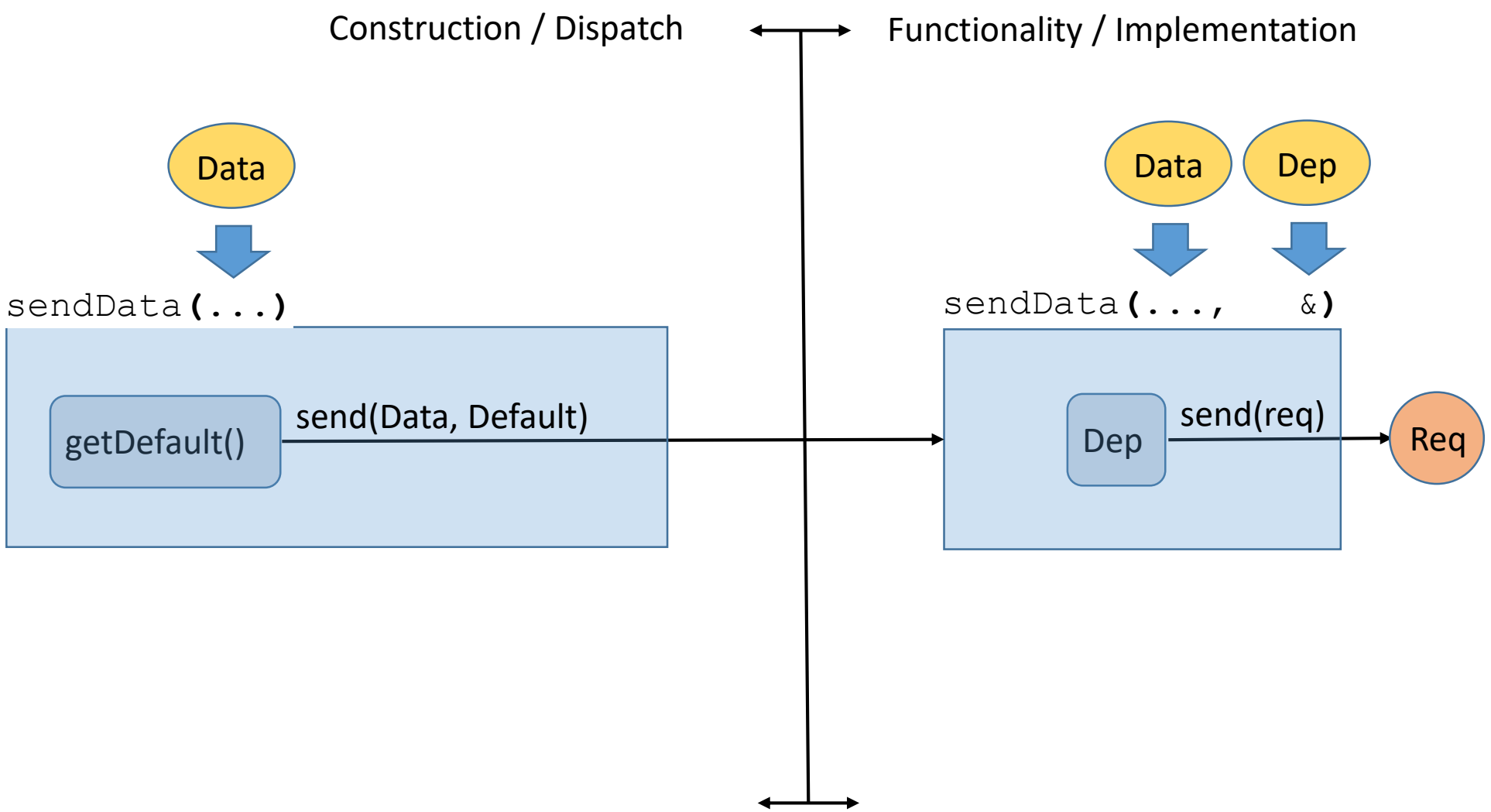
Lazy Initialization

```
// New overload that replaces singleton
Response sendData(const Data& data, CommWrapper& comms)
{
    Request req;
    // Transform Data into Request
    // .....
    return comms.send(req);
}

// keep original signature
Response sendData(const Data& data)
{
    return sendData(data, getDefaultComms());
}
```

Separation of Concerns



Separation of Concerns



Phased Introduction

```
// New overload that replaces
// singleton
Response sendData(const Data& data,
                  CommWrapper& comms)
{
    Request req;
    // Tranform Data into Request
    // .....
    return comms.send(req);
}

// keep original signature
Response sendData(const Data& data)
{
  return sendData(data,
            getDefaultComms());
}
```

```
// Other Code with singleton use
Response sendXData(const XData& data)
{
    Request req;
    // Tranform XData into Request
    // .....
    return CommSingleton::
                instance()->send(req);
}
```

Phased Introduction

```
class CommSingleton
{
public:
  static CommSingleton* instance()
  {
    COMPILER_DO_ONCE {
      static CommSingleton* instance_ = new CommSingleton();
    }
    return instance_;
  }
  Response send(const Request& req);
private:
  CommSingleton();
  CommSingleton(const CommSingleton&);
  void operator=(const CommSingleton&);
};


// elsewhere
CommSingleton::instance()->send(req);
```

Phased Introduction

```
class CommSingleton
{
public:
  static CommWrapper* instance()
  {
    return &(getDefaultComms());
  }
private:
  CommSingleton();
  CommSingleton(const CommSingleton&);
  void operator=(const CommSingleton&);
};
```

```
// elsewhere
CommSingleton::instance()->send(req);
```

Phased Introduction

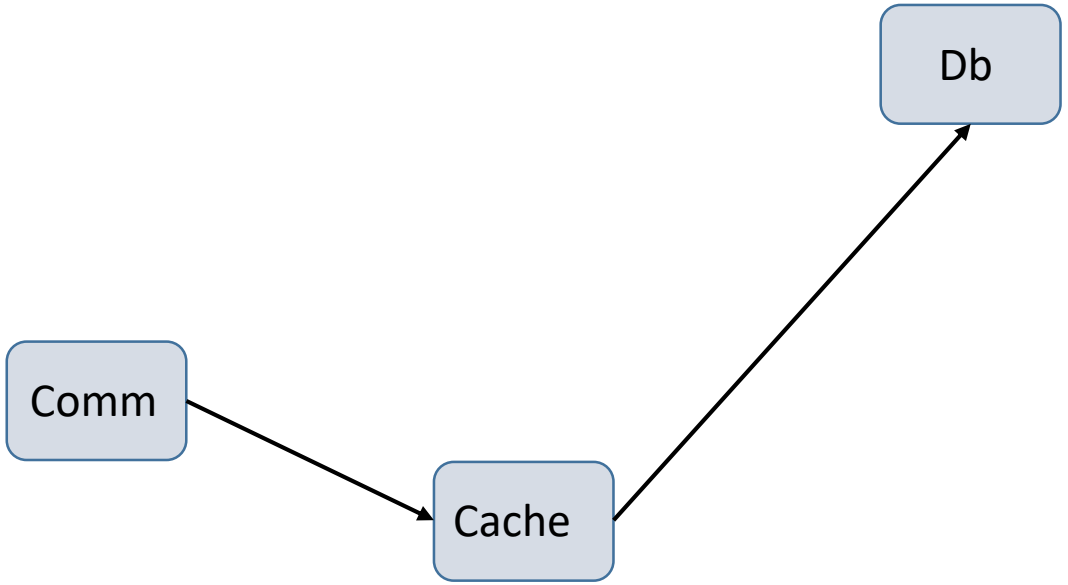
```
// New overload that replaces
singleton
Response sendData(const Data&
data, CommWrapper& comms)
{
    Request req;
    // Transform Data into Request
    // .....
    return comms.send(req);
}

// keep original signature
Response sendData(const Data&
data)
{
  return sendData(data,
           getDefaultComms());
}
```

```
// Other Code with singleton use
Response sendXData(const XData& data)
{
    Request req;
    // Transform Data into Request
    // .....
    return CommSingleton::
               instance()->send(req);
}
```

Questions

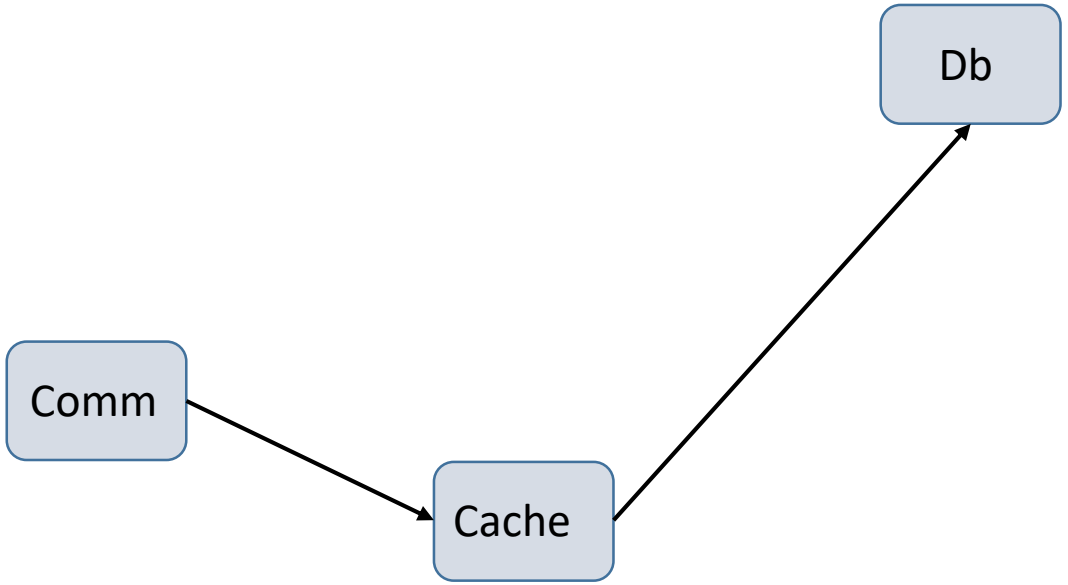
#include <slide_numbers>

Initialization Dependencies




```
int main(int argc, char* argv[])
{
   ...
   Comm::init();
   Cache::init();
   Db::init();
   ...
}
```

Initialization Dependencies



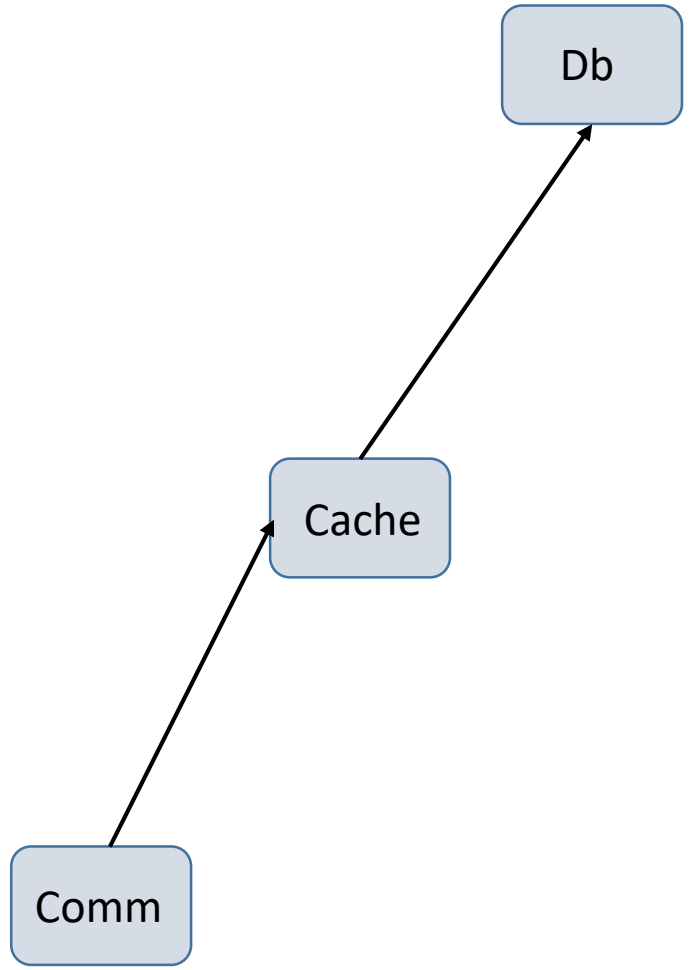

```
int main(int argc, char* argv[])
{
   ...
   Db::init(); // Correct
   Cache::init();
   Comm::init();
   ...
}
```

Initialization Dependencies

```
class CacheWrapper {
public:
    CacheWrapper(DataBaseWrapper& db):db_(db){...}
    virtual int save(const Request& req);
private:
    DataBaseWrapper& db_;
};

class CommWrapper {
public:
    CommWrapper(CacheWrapper& cache):cache_(cache){...};
    virtual Response send(const Request& req);
private:
    CacheWrapper& cache_;
};
```

Initialization Dependencies

```
DataBaseWrapper& getDefaultDb()
{
    static DataBaseWrapper db;
    return db;
}


CacheWrapper& getDefaultCache()
{
    static CacheWrapper cache(getDefaultDb());
    return cache;
}


CommWrapper& getDefaultComms()
{
    static CommWrapper comms(getDefaultCache());
    return comms;
}
```



Initialization Dependencies

```
struct MockDbClient : public DataBaseWrapper
{
    MOCK_METHOD1(save, Response(const Request&));
};

struct MockCacheClient : public CacheWrapper
{
    MockCacheClient(MockDbClient& mdb):CacheWrapper(mdb){}
    MOCK_METHOD1(save, Response(const Request&));
};

struct MockCommClient : public CommWrapper
{
    MockCommClient(MockCacheClient& mch):CommWrapper(mch){}
    MOCK_METHOD1(send, Response(const Request&));
};
```

```
TEST(XTest, sendData)
{
    MockDbClient db_client;
    MockCacheClient cache_client(db_client);
    MockCommClient comm_client(cache_client);

    Data rec;
    rec.id = 999;
    //....
    Response resp;
    Request db_req, cache_req, comm_req;

    EXPECT_CALL(db_client, save(_)).WillOnce(DoAll(SaveArg<0>(&db_req),
                Return(resp)));
    EXPECT_CALL(cache_client, save(_)).WillOnce(DoAll(SaveArg<0>(&cache_req),
                Return(resp)));
    EXPECT_CALL(comm_client, send(_)).WillOnce(DoAll(SaveArg<0>(&comm_req),
                Return(resp)));
    sendData(rec, comm_client);

    ASSERT_EQ(comm_req.senderId_, rec.id);
    //Further validation of various req values;
}
```

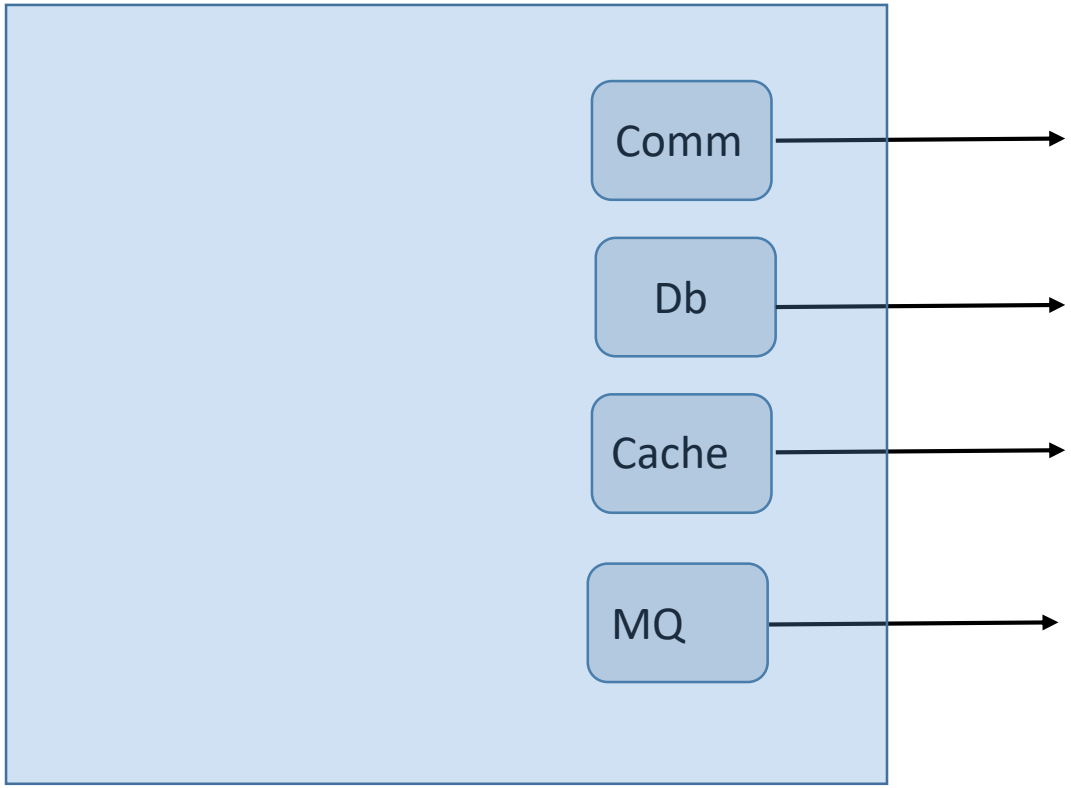
Grouping Dependencies

In Reality, Singletons run in groups

- There may be multiple singletons embedded in a large legacy function
- How to pass in a group of dependencies
  - Without a lot of boilerplate
  - Be natural looking

Multiple Dependencies

```
Response sendData(...)
```



Brute force

```
// New overload that replaces singleton
Response sendData(const Data& data, CommWrapper& comms, MqWrapper& mq,
CacheWrapper& cache, DbWrapper& db)
{
    Request req;
    // Transform Data into various data structures
    // ...
    db.save(db_data);
    cache.save(cache_struct);
    mq.send(req);
    return comms.send(req);
}

// keep original signature
Response sendData(const Data& data)
{
   // Inject defaults here
   return sendData(data, getDefaultComms(), getDefaultMq(),
getDefaultCache(), getDefaultDb());
}
```

Grouping Dependencies

```
// Groups Dependencies
struct Service {
    CommWrapper comms_;
    DataBaseWrapper db_;
    CacheWrapper cache_;
    MqWrapper mq_;
};

// Lazy Initialization
Service& getDefaultServices() {
    static Service services;
    return services;
}
```

Grouping Dependencies

```
// Refactored function that replaces singleton
Response sendData(const Data& data, Service& services)
{
    Request req;
    // Transform Data into Request
    // ...
    services.db_.save(req);
    services.cache_.save(req);
    services.mq_.send(req);
    services.comms_.send(req);
}

// keep original signature
Response sendData(const Data& data)
{
    // Inject default here
    return sendData(data, getDefaultServices());
}
```

```
#include "MockService.t.h"

// Refactored function that replaces singleton
template< typename SERVICE >
Response sendData(const Data& data, SERVICE& services)
{
    Request req;
    // Transform Data into Request
    // ...
    services.db_.save(req);
    services.cache_.save(req);
    services.mq_.send(req);
    return services.comms_.send(req);
}

// keep original signature
Response sendData(const Data& data)
{
    // Inject default here
    return sendData(data, getDefaultServices());
}

template Response sendData<MockService>(const Data& data, MockService&
services);
```

```
// Service.h : Groups Dependencies
struct Service
{
    Service(CommWrapper& comms, DataBaseWrapper& db,
      CacheWrapper& cache, MqWrapper& mq)
      : comms_(comms), db_(db), cache_(cache), mq_(mq) {};

    CommWrapper& comms_;
    DataBaseWrapper& db_;
    CacheWrapper& cache_;
    MqWrapper& mq_;
};
```

```
// Service.cpp : Lazy Initialization
Service& getDefaultServices()
{
    static CommWrapper comms;
    static DataBaseWrapper db;
    static CacheWrapper cache;
    static MqWrapper mq;
    static Service services(comms, db, cache, mq);
    return services;
};
```

```
// Refactored function that replaces singleton
Response sendData(const Data& data, Service& services)
{
    Request req;
    // Transform Data into Request
    // ...
    services.db_.save(req);
    services.cache_.save(req);
    services.mq_.send(req);
    return services.comms_.send(req);
}

// keep original signature
Response sendData(const Data& data)
{
    // Inject default here
    return sendData(data, getDefaultServices());
}
```

```
struct MockCommClient : public CommWrapper
{
    MOCK_METHOD1(send, Response(const Request&));
};
struct MockDbClient : public DatabaseWrapper
{
    MOCK_METHOD1(save, int(const Request&));
};
struct MockCacheClient : public CacheWrapper
{
    MOCK_METHOD1(save, int(const Request&));
};
struct MockMqClient : public MqWrapper
{
    MOCK_METHOD1(send, int(const Request&));
};
```

```
TEST(XTest, sendData)
{
    // Setup
    MockCommClient comms;
    MockDbClient db;
    MockMqClient mq;
    MockCacheClient cache;

    Service services(comms, db, cache, mq);

    Request comm_upd;
    Request cache_upd;
    Request db_upd;
    Request mq_upd;

    EXPECT_CALL(comms, send(_)).WillOnce(DoAll(SaveArg<0>(&comm_upd), Return(resp)));
    EXPECT_CALL(mq, send(_)).WillOnce(DoAll(SaveArg<0>(&mq_upd), Return(1)));
    EXPECT_CALL(db, save(_)).WillOnce(DoAll(SaveArg<0>(&db_upd), Return(1)));
    EXPECT_CALL(cache, save(_)).WillOnce(DoAll(SaveArg<0>(&cache_upd), Return(1)));
    ...
```

```
TEST(XTest, sendData)
{

    // Previous Mock Setup
    ...

    // Input Data Setup
    Data rec;
    rec.id = 999;
    // ....
    sendData(rec, services);

    ASSERT_EQ(comm_upd.senderId_, rec.id);
    ASSERT_EQ(cache_upd.senderId_, rec.id);
    ASSERT_EQ(db_upd.senderId_, rec.id);
    ASSERT_EQ(mq_upd.senderId_, rec.id);
    // …
}
```

```
using CallFunc=std::function<int(Request)>;

// Groups Dependencies
struct Service
{
    CallFunc comms_;
    CallFunc db_;
    CallFunc cache_;
    CallFunc mq_;
};

// Lazy Initialization
Service& getDefaultServices()
{
    static CommWrapper comms;
    static DatabaseWrapper db;
    static CacheWrapper cache;
    static MqWrapper mq;
    static Service services{std::ref(comms), std::ref(db),
                            std::ref(cache), std::ref(mq)};
    return services;
}
```

```
// Refactored function that replaces singleton
int sendData(const Data& data, Service& services)
{
    Request req;
    // Transform Data into Request
    // ...
    services.db_(req);
    services.cache_(req);
    services.mq_(req);
    return services.comms_(req);
}

// keep original signature
int sendData(const Data& data)
{
    // Inject default here
    return sendData(data, getDefaultServices());
}
```

Stateful Dependencies

```
class SendProcessor {
 public:

  // Refactored implementation functions that removes singleton
  Response sendData(const Data& data, AutoClient& services);
  Response sendXData(const XData& xdata, AutoClient& services);
  ...

  // Dispatch functions
  Response sendData(const Data& data);
  Response sendXData(const XData& xdata);
  ...
};
```

Stateful Dependencies

```
class SendProcessor {
 public:

  SendProcessor(AutoClient& services) : services_(services){…} // Move to cpp
  SendProcessor() : SendProcessor(getDefaultServices()){…} // Move to cpp

  // Refactored functions that replaces singleton
  Response sendData(const Data& data, AutoClient& services);
  Response sendXData(const XData& xdata, AutoClient& services);
  ...

  // Dispatch functions
  Response sendData(const Data& data);
  Response sendXData(const XData& xdata);
  ...
 private :
    AutoClient& services_;
};
```

Stateful Dependencies

```
class SendProcessor {
 public:

  SendProcessor(AutoClient& services) : services_(services); // Move to cpp
  SendProcessor() : SendProcessor(getDefaultServices()); // Move to cpp

  // Refactored functions with internal dispatch
  Response sendData(const Data& data);
  Response sendXData(const XData& xdata);
  ...
 private :
    AutoClient& services_;
};
```

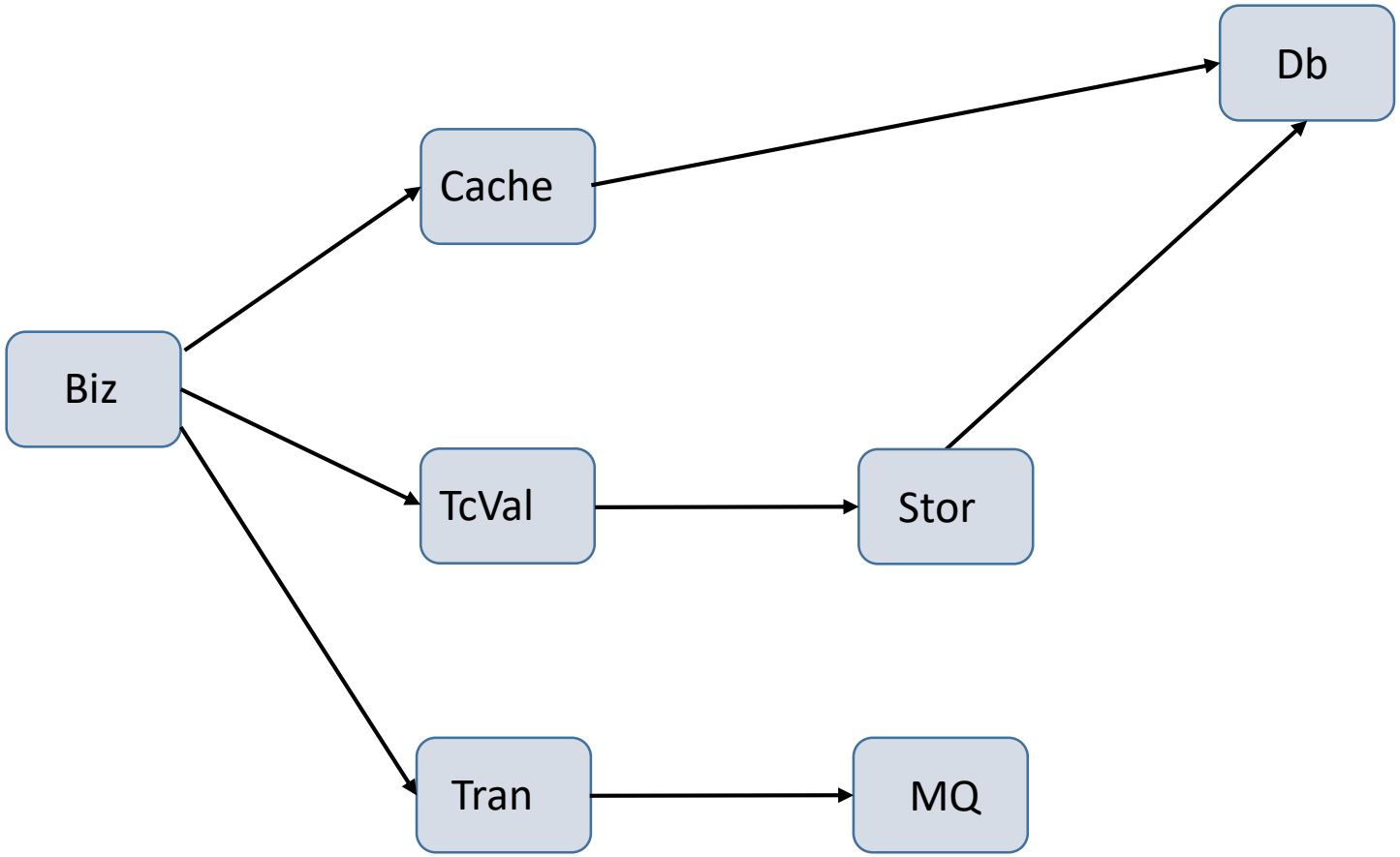
Stateful Dependencies

```
TEST(XTest, sendData)
{
    // Setup
    MockCommClient comms;
    MockDbClient db;
    MockMqClient mq;
    MockCacheClient cache;

    Service services(comms, db, cache, mq);
    Request comm_upd;
    Request cache_upd;
    Request db_upd;
    Request mq_upd;
    Response resp;

    EXPECT_CALL(comms, send(_)).WillOnce(DoAll(SaveArg<0>(&comm_upd), Return(resp)));
    EXPECT_CALL(mq, send(_)).WillOnce(DoAll(SaveArg<0>(&mq_upd), Return(1)));
    EXPECT_CALL(db, save(_)).WillOnce(DoAll(SaveArg<0>(&db_upd), Return(1)));
    EXPECT_CALL(cache, save(_)).WillOnce(DoAll(SaveArg<0>(&cache_upd), Return(1)));
    ...
```

Stateful Dependencies

```
TEST(XTest, sendData)
{
    // Previous Mock Setup
    ...
    // Input Data Setup
    Data rec;
    rec.id = 999;

    // ...
    Processor processor(services);
    processor.sendData(rec);

    ASSERT_EQ(comm_upd.senderId_, rec.id);
    ASSERT_EQ(cache_upd.senderId_, rec.id);
    ASSERT_EQ(db_upd.senderId_, rec.id);
    ASSERT_EQ(mq_upd.senderId_, rec.id);
    // ...
    processor.sendXData(rec2);
    ASSERT_EQ(comm_upd.senderId_, rec2.id);
}
```

Initialization Dependencies



Review

Replacing Singletons while …

- Keeping API Source compatible
- Keeping ABI compatible
- Avoiding Copies for classes with deleted/private copy constructor
- Delayed Initialization of resources, if necessary
- Phased Introduction for replacing singleton calls
- Initialization order of interdependent singletons
- Grouping Multiple Singleton dependencies together
- Stateful grouping of dependencies

* With Testable code

How has it worked out ? / What’s been the uptake ?

Contact : pmuldoon1@Bloomberg.net

Questions ?