Testing Battle.net

(Before deploying to millions of players)

Ben Deane

Principal Engineer, Blizzard Entertainment

`bdeane@blizzard.com, @ben_deane`

13th May 2015

Battle.net infrastructure

- About 750,000 lines of C++
  - Servers + client libraries
- "Battle.net Game Service"
  - Authenticate players
  - Social: friends, presence
  - Matchmaking (cooperative/competitive)
  - Achievements/profiles

Battle.net is highly. . .

- Distributed
- Asynchronous
- Configured
- Fault-prone
- Architecture-varied
  - inheritance
  - composition
  - value-oriented

Battle.net integration testing

- API testing is pretty robust
  - and certainly valuable
- But this doesn't help me in the moment
  - slow to build
  - slow to run
  - needs a full environment

My journey towards effective unit testing

- No practice at unit testing
- Large project with many moving parts
- Mature lower level libraries
- New code (features) added at an alarming rate

What's typically well-tested?

- UTF-8 string conversion
- String interpolation
- URL parsing/decomposition
- Stats/math code

These things are "easy mode" for tests.

Not-so-well tested?

- Filesystem interaction (caching downloaded objects)
- Matchmaking algorithms
- Queueing/Load balancing algorithms

These things are harder to test. Where to start?

My conclusions

- We don't do unit testing because we aren't practised at it
- Because we don't do "TDD", we *can't* do unit testing
  - Legacy code is poorly structured
- We have a test framework

My goals for "unit" tests

- Fast
- No data/process dependencies
- Automated
- Binary pass/fail
- Independent
- No test-only interface support
- By me, for me

No magic bullet

- I wrote a lock of mocks
- Set up a lot of data structures for test
- A lot of testing code to keep bug-free
- But along the way I found
    - better code structure
    - useful techniques

Enemies of testing

- Global state
- Deep inheritance
- Mixing concerns, coupling
- I/O

Enemies of testing

- Doing work in constructors (cf RAII)
- Lack of dependency injection
- Wide interfaces (especially when passed to constructors)

Exhibit A: hard to test

```
class ChannelBase : public rpc::Implementor<protocol::channel::Channel>;
class ChannelImpl : public ChannelBase;

class PresenceChannelImpl : public ChannelImpl
{
public:
  PresenceChannelImpl(
    Process* process,
    rpc::RPCDispatcher* insideDispatcher,
    const EntityId& entityId,
    ChannelDelegate* channelDelegate,
    ChannelOwner* owner,
    const PresenceFieldConfigMap& fieldMap);
};
```

Exhibit A: hard to test

```
class ChannelBase : public rpc::Implementor<protocol::channel::Channel>;
class ChannelImpl : public ChannelBase;

class PresenceChannelImpl : public ChannelImpl
{
public:
  PresenceChannelImpl(
    Process* process,
    rpc::RPCDispatcher* insideDispatcher,
    const EntityId& entityId,
    ChannelDelegate* channelDelegate,
    ChannelOwner* owner,
    const PresenceFieldConfigMap& fieldMap);
};
```

Exhibit A: hard to test

```
class ChannelBase : public rpc::Implementor<protocol::channel::Channel>;
class ChannelImpl : public ChannelBase;

class PresenceChannelImpl : public ChannelImpl
{
public:
  PresenceChannelImpl(
    Process* process,
    rpc::RPCDispatcher* insideDispatcher,
    const EntityId& entityId,
    ChannelDelegate* channelDelegate,
    ChannelOwner* owner,
    const PresenceFieldConfigMap& fieldMap);
};
```

Exhibit B: hard to test

```
class AchievementsServiceImpl
  : public bnet::achievements::AchievementsService
  , public AchievementsServiceStaticDataLoader
{
public:
  AchievementsServiceImpl(
    bnet::internal::ServerHelper& serverHelper,
    mysql::Databases* mysql);

};
```

Exhibit B: hard to test

```
class AchievementsServiceImpl
  : public bnet::achievements::AchievementsService
  , public AchievementsServiceStaticDataLoader
{
public:
  AchievementsServiceImpl(
    bnet::internal::ServerHelper& serverHelper,
    mysql::Databases* mysql);

};
```

Exhibit B: hard to test

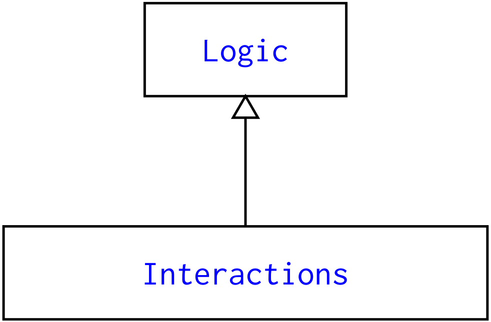
```
class ServerHelper
{
public:
  ServerHelper(...); // 12 args!

  rpc::RPCServer* GetInsideRPCServer() const;
  rpc::RPCServer* GetOutsideRPCServer() const;
  ...
};
```

In hindsight, this was a mistake

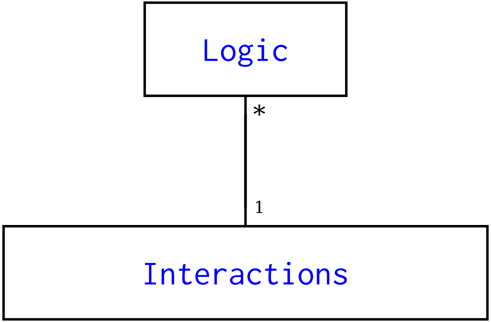
Class structure for testing

- Base class (contains logic)
- Derived class (contains I/O, config, etc)

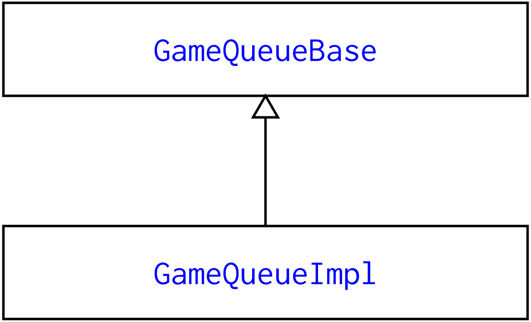


Class structure for testing

- Component class (contains logic)
- Entity/Object class (contains I/O, config, etc)



Example: Queueing for games



Queueing for games

`GameQueueBase` contains the queueing logic

```
class GameQueueBase
{
public:
  GameQueueBase(
    shared_ptr<ServerPoolInterface> interface,
    const PopCallback& popCb,
    const UpdateCallback& updateCb,
    const PollTimerCallback& pollTimerCb,
    const NotificationTimerCallback& notificationTimerCb);

  bool   Push(...);
  size_t Pop(...);
  void   Remove(...);
  size_t PollQueue(...);

  ...
};
```

Queueing for games

`GameQueueBase` contains the queueing logic

```
class GameQueueBase
{
public:
  GameQueueBase(
    shared_ptr<ServerPoolInterface> interface,
    const PopCallback& popCb,
    const UpdateCallback& updateCb,
    const PollTimerCallback& pollTimerCb,
    const NotificationTimerCallback& notificationTimerCb);

  bool   Push(...);
  size_t Pop(...);
  void   Remove(...);
  size_t PollQueue(...);

  ...
};
```

Queueing for games

`GameQueueBase` contains the queueing logic

```
class GameQueueBase
{
public:
  GameQueueBase(
    shared_ptr<ServerPoolInterface> interface,
    const PopCallback& popCb,
    const UpdateCallback& updateCb,
    const PollTimerCallback& pollTimerCb,
    const NotificationTimerCallback& notificationTimerCb);

  bool   Push(...);
  size_t Pop(...);
  void   Remove(...);
  size_t PollQueue(...);

  ...
};
```

Queueing for games

`GameQueueImpl` deals with protocols

```
class GameQueueImpl
  : public GameQueueBase
  , public protocol::game_queue::GameQueue
{
public:
  // protocol handler functions
  virtual void AddToQueue(...);
  virtual void RemoveFromQueue(...);
  ...

  // system events
  bool OnInit(...);
  bool OnFlush(...);
  void OnShutdown(...);
  void OnPeerDisconnected(...);
  ...
};
```

Queueing for games

GameQueueImpl deals with protocols

```
class GameQueueImpl
  : public GameQueueBase
  , public protocol::game_queue::GameQueue
{
public:
  // protocol handler functions
  virtual void AddToQueue(...);
  virtual void RemoveFromQueue(...);
  ...

  // system events
  bool OnInit(...);
  bool OnFlush(...);
  void OnShutdown(...);
  void OnPeerDisconnected(...);
  ...
};
```

Queueing for games

`GameQueueImpl` deals with system events

```
class GameQueueImpl
  : public GameQueueBase
  , public protocol::game_queue::GameQueue
{
public:
  // protocol handler functions
  virtual void AddToQueue(...);
  virtual void RemoveFromQueue(...);
  ...

  // system events
  bool OnInit(...);
  bool OnFlush(...);
  void OnShutdown(...);
  void OnPeerDisconnected(...);
  ...
};
```

Queueing for games

`GameQueueImpl` deals with config

```
class GameQueueImpl
  : public GameQueueBase
  , public protocol::game_queue::GameQueue
{
public:
  ...

  // setup/config
  bool ProcessProgramConfig(...);

  // queue polling
  void StartPollTimer(...);
  void ServicePollTimer(...);
  void StartNotificationPollTimer(...);
  void ServiceNotificationPollTimer(...);
  ...
};
```

Queueing for games

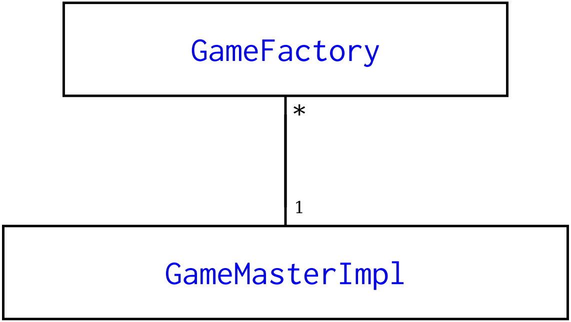
`GameQueueImpl` deals with polling logic

```
class GameQueueImpl
  : public GameQueueBase
  , public protocol::game_queue::GameQueue
{
public:
  ...

  // setup/config
  bool ProcessProgramConfig(...);

  // queue polling
  void StartPollTimer(...);
  void ServicePollTimer(...);
  void StartNotificationPollTimer(...);
  void ServiceNotificationPollTimer(...);
  ...
};
```

Example: Matchmaking



Matchmaking

`GameFactory` contains matchmaking logic

```
class GameFactory
{
public:
  GameFactory(const AttributeValue& version,
              const ProgramId& programId,
              GameFactoryId id);

  virtual bool Configure(const GameFactoryConfig& config);

  ...
  virtual Error RegisterPlayers(...);
  virtual bool UnregisterPlayers(...);
  virtual Error JoinGame(...);
  ...
};
```

Matchmaking

`GameFactory` contains matchmaking logic

```
class GameFactory
{
public:
  GameFactory(const AttributeValue& version,
              const ProgramId& programId,
              GameFactoryId id);

  virtual bool Configure(const GameFactoryConfig& config);

  ...
  virtual Error RegisterPlayers(...);
  virtual bool UnregisterPlayers(...);
  virtual Error JoinGame(...);
  ...
};
```

Matchmaking

`GameFactory` contains matchmaking logic

```
class GameFactory
{
public:
  GameFactory(const AttributeValue& version,
              const ProgramId& programId,
              GameFactoryId id);

  virtual bool Configure(const GameFactoryConfig& config);

  ...
  virtual Error RegisterPlayers(...);
  virtual bool UnregisterPlayers(...);
  virtual Error JoinGame(...);
  ...
};
```

Matchmaking

`GameMasterImpl` deals with interactions

```
class GameMasterImpl
  : public protocol::game_master::GameMaster
{
public:
  ...
  void OnPeerDisconnected(...);
  ...
  void InstantiateFactories(...);
  ...
  virtual void ListFactories(...);
  virtual void JoinGame(...);
  virtual void FindGame(...);
  virtual void GameEnded(...);
  virtual void PlayerLeft(...);
  ...
};
```

Matchmaking

`GameMasterImpl` deals with interactions

```
class GameMasterImpl
  : public protocol::game_master::GameMaster
{
public:
  ...
  void OnPeerDisconnected(...);
  ...
  void InstantiateFactories(...);
  ...
  virtual void ListFactories(...);
  virtual void JoinGame(...);
  virtual void FindGame(...);
  virtual void GameEnded(...);
  virtual void PlayerLeft(...);
  ...
};
```

Matchmaking

`GameMasterImpl` deals with interactions

```
class GameMasterImpl
  : public protocol::game_master::GameMaster
{
public:
  ...
  void OnPeerDisconnected(...);
  ...
  void InstantiateFactories(...);
  ...
  virtual void ListFactories(...);
  virtual void JoinGame(...);
  virtual void FindGame(...);
  virtual void GameEnded(...);
  virtual void PlayerLeft(...);
  ...
};
```

A successful pattern

- Decouple logic from other concerns
  - Dependency injection for config etc
  - Makes the logic testable
- This can be fairly easily applied even to monolithic classes
  - Just apply the inheritance pattern
  - Some testing beats no testing

Goals for testable classes

Dependency injection is probably the biggest factor affecting whether or not code *is testable at all*.

Even with DI, classes are *onerous to test* unless constructors take few arguments, using narrow interfaces.

Testing Performance/Efficiency

- Different solutions for
  - thousands (performance)
  - millions (performance + algorithms)
  - billions (algorithms by construction)
- Battle.net's working sets are in the millions

Problems in million-land

- Computations can run on a single machine
- Data structures are important to performance
  - Caching concerns, optimizations can get you 100x
  - But they can't get you 100,000x
- Algorithms are important to efficiency

Testing for performance

- Timed tests are easy, not so useful
- My machine is a Windows desktop
- Production machine is a CentOS blade
- Timed tests
  - compare times when optimizing
  - can’t tell me if code is fast enough in an absolute sense

Efficiency: easy to lose

- Team of engineers hacking away on features
- $O(log\, n)$ or less is required
- Easy to accidentally turn it into $O(n)$ (or worse)
- I need a way to test for algorithmic efficiency

Testing for efficiency

- A simple idea
- Run the same test with different sized inputs
- Compute ratio of times

$$T_1 = (\textit{time for run on data of size } N)$$
$$T_2 = (\textit{time for run on data of size } kN)$$

Bucketing

$$O(1) \Rightarrow \frac{T_2}{T_1} = 1$$

$$O(log\, n) \Rightarrow \frac{T_2}{T_1} = 1 + \frac{log(k)}{log(N)}$$

$$O(n) \Rightarrow \frac{T_2}{T_1} = k$$

$$O(n\, log\, n) \Rightarrow \frac{T_2}{T_1} = k\,(1 + \frac{log(k)}{log(N)})$$

$$O(n^2) \Rightarrow \frac{T_2}{T_1} = k^2$$

This sounds easy, but...

- Timing is hard
  - sensitive to machine load
  - sensitive to caching effects (CPU/OS)
  - sensitive to timing function: granularity/perf
- Statistical mitigation
- Somewhat careful choice of k, N
  - I settled on 32 for each ($N = 32, kN = 1024$)

OK, but. . .

Where do you get different-sized inputs?
You can let the test make them...

```
const int MULT = 32;
const int N = 32;
...
// run 1 - with size N
auto sampleTime1 = test->Run(N);
test->Teardown();

test->Setup();
// run 2 - with size kN
auto sampleTime2 = test->Run(N * MULT);
...
```

OK, but. . .

Where do you get different-sized inputs?
You can let the test make them...

```
const int MULT = 32;
const int N = 32;
...
// run 1 - with size N
auto sampleTime1 = test->Run(N);
test->Teardown();

test->Setup();
// run 2 - with size kN
auto sampleTime2 = test->Run(N * MULT);
...
```

Let the test make them?

- Affects the timing if done naively (i.e. wrongly)
  - Adds an $O(n)$ component to the test
  - So move the timing code inside the test also
- Boilerplate in test code
- It's not ideal...

Let the test make them?

Result: a typical test

- ~20 lines setup
- ~20 lines size-related setup
- ~10 lines timing
- ~5 lines actual logic
- ~5 lines test macros

Yuck.

Let the test make them?

- It works well enough to give me confidence
  - Matchmaking won't blow up with a million players
- So I lived with this for a while…
- But I'm lazy, I don't want to maintain all this code
- And I'm a student of Haskell…

Wish-driven development

What I have

```
DEF_TEST(TestName, Suite)
{
  ...
  return test_result;
}
```

What I want

```
DEF_PROPERTY(TestName, Suite, const string& s)
{
  // do something with s
  // that should be true for any input
  ...
  return property_holds;
}
```

Test macros expand into functions

Macro...

```
DEF_PROPERTY(TestName, Suite, const string& s)
{
  ...
}
```

Expands to...

```
struct NonceStruct
{
  ...
  bool operator()(const string&);
};
bool NonceStruct::operator()(const string& s)
{
  ...
}
```

Discover the type of the function argument

Simple `function_traits` template

```
template <typename T>
struct function_traits
  : public function_traits<decltype(&T::operator())>
{};

template <typename R, typename A>
struct function_traits<R(A)>
{
  using argType = A;
};

template <typename C, typename R, typename A>
struct function_traits<R(C::*)(A)>
  : public function_traits<R(A)>
{};

...
```

Discover the type of the function argument

Simple `function_traits` template

```
template <typename T>
struct function_traits
  : public function_traits<decltype(&T::operator())>
{};

template <typename R, typename A>
struct function_traits<R(A)>
{
  using argType = A;
};

template <typename C, typename R, typename A>
struct function_traits<R(C::*)(A)>
  : public function_traits<R(A)>
{};

...
```

Discover the type of the function argument

Simple `function_traits` template

```
template <typename T>
struct function_traits
  : public function_traits<decltype(&T::operator())>
{};

template <typename R, typename A>
struct function_traits<R(A)>
{
  using argType = A;
};

template <typename C, typename R, typename A>
struct function_traits<R(C::*)(A)>
  : public function_traits<R(A)>
{};

...
```

Implement a `Run` function

`Run()` for a property test

```
// DEF_PROPERTY(TestName, Suite, TYPE) becomes...
struct NonceStruct : public Test
{
  ...
  virtual bool Run() override
  {
    // Property will type-erase NonceStruct, discover its argument type
    Property p(*this);
    // check() generates arguments to call NonceStruct(TYPE)
    return p.check();
  }
  ...
};
```

Property type-erases NonceStruct

Inside Property

```
template <typename T>
struct Internal : public InternalBase
{
  ...

  using paramType = std::decay_t<typename function_traits<T>::argType>;

  virtual bool check()
  {
    ...
    // generate a value of the right type
    paramType p = Arbitrary<paramType>::generate(...);
    // feed it to the struct's operator()
    return m_t(p);
  }

  T m_t;
};
```

Property type-erases NonceStruct

Inside Property

```
template <typename T>
struct Internal : public InternalBase
{
  ...

  using paramType = std::decay_t<typename function_traits<T>::argType>;

  virtual bool check()
  {
    ...
    // generate a value of the right type
    paramType p = Arbitrary<paramType>::generate(...);
    // feed it to the struct's operator()
    return m_t(p);
  }

  T m_t;
};
```

How to generate `TYPE`?

Use a template, naturally

```
template <typename T>
struct Arbitrary
{
  static T generate(size_t /*generation*/, unsigned long int /*seed*/)
  {
    return T();
  }
};
```

And specialize...

Specializing `Arbitrary<T>`

- Easy to write `Arbitrary<T>` for fundamental types
- Front-load likely edge cases
  - `0`
  - `numeric_limits<T>::min()`
  - `numeric_limits<T>::max()`
- Otherwise use uniform distribution over range

Specializing `Arbitrary<T>`

For `int`-like types

```
static int generate(size_t g, unsigned long int seed)
{
  switch (g)
  {
    case 0: return 0;
    case 1: return std::numeric_limits<T>::min();
    case 2: return std::numeric_limits<T>::max();
    default:
    {
      std::mt19937 gen(seed);
      std::uniform_int_distribution<T> dis(
        std::numeric_limits<T>::min(), std::numeric_limits<T>::max());
      return dis(gen);
    }
  }
}
```

Specializing `Arbitrary<T>`

For `int`-like types

```
static int generate(size_t g, unsigned long int seed)
{
  switch (g)
  {
    case 0: return 0;
    case 1: return std::numeric_limits<T>::min();
    case 2: return std::numeric_limits<T>::max();
    default:
    {
      std::mt19937 gen(seed);
      std::uniform_int_distribution<T> dis(
        std::numeric_limits<T>::min(), std::numeric_limits<T>::max());
      return dis(gen);
    }
  }
}
```

Specializing `Arbitrary<T>`

- Once we have `Arbitrary<T>` for fundamental types...
- Easy to write for compound types
  - `vector<T>` etc
  - `generate` works in terms of `generate` on the contained type
  - the power of ADTs!

Specializing `Arbitrary<T>`

For compound types (eg `vector`)

```
static vector<T> generate(size_t g, unsigned long int seed)
{
  vector<T> v;
  size_t n = 10 * ((g / 100) + 1);
  v.reserve(n);
  std::generate_n(
    std::back_inserter(v), n, [&] () {
      return Arbitrary<T>::generate(g++, seed++); });
  return v;
}
```

Now we have property tests

- Macro expands `NonceStruct` with `operator()`
- `Property` type-erases `NonceStruct`
- `Property::Check` does:
  - `function_traits` discovery of the argument type `T`
  - `Arbitrary<T>::generate` to make a `T`
  - Call `NonceStruct::operator()`
- And plumb through parameters like number of checks, random seed

Better checks for compound types

When a check fails, find a minimal failure case

```
template <typename T>
struct Arbitrary
{
  static std::vector<T> shrink(const T& /*t*/)
  {
    return std::vector<T>();
  }
};
```

`shrink` returns a `vector` of "reduced" `T`'s

Better checks for compound types

A simple binary search

```
static std::vector<std::basic_string<T>> shrink(
  const std::basic_string<T>& t)
{
  std::vector<std::basic_string<T>> v;
  if (t.size() < 2)
    return v;
  auto l = t.size() / 2;
  v.push_back(t.substr(0, l));
  v.push_back(t.substr(l));
  return v;
}
```

Call `shrink` repeatedly to find a minimal fail case

A short demo

(Demo)

Testing for efficiency (again)

Now the computer can generate N, kN values

```
static vector<T> generate(size_t g, unsigned long int seed)
{
  vector<T> v;
  size_t n = 10 * ((g / 100) + 1);
  v.reserve(n);
  std::generate_n(
    std::back_inserter(v), n, [&] () {
      return Arbitrary<T>::generate(g++, seed++); });
  return v;
}
```

Add `generate_n` as a tighter form of `generate`

Testing for efficiency (again)

Now the computer can generate N, kN values

```
static vector<T> generate_n(size_t g, unsigned long int seed)
{
  vector<T> v;
  // use g directly instead of a "loose" value
  v.reserve(g);
  std::generate_n(
    std::back_inserter(v), g, [&] () {
      return Arbitrary<T>::generate_n(g, seed++); });
  return v;
}
```

Add `generate_n` as a tighter form of `generate`

Now I can write

A sample complexity test

```
DEF_COMPLEXITY_PROPERTY(TestName, Suite, ORDER_N, const string& s)
{
  // something that's supposed to be order N...
  std::max_element(s.begin(), s.end());
}
```

And specialize `Arbitrary` for my own types as necessary
Much less boilerplate to maintain

So that's where I am now

- Dependency injection (little work in constructors)
- Separate logic from interaction (even in monolithic classes)
- Regular tests for "normal, identified" cases
- Timed tests when I'm optimizing
- Property-based tests for invariants
- Algorithmic complexity tests for scalability confidence

The future?

- `Arbitrary` opens the door for fuzz testing?
- Alternative walk strategies through the input space
  - Hilbert?
  - Morton
  - etc
- I'm still lazy; the computer isn't doing enough for me yet

Battle.net is still highly. . .

- Distributed
- Asynchronous
- Configured
- Fault-prone
- Architecture-varied

But more parts of it are well-tested before they leave a developer's machine.

And I'm more confident changing code with a guarantee that correctness/efficiency/scalability won't be affected.

Thanks for listening

Code: `https://github.com/elbeno/testinator`

Me: `bdeane@blizzard.com, @ben_deane`