

2015-04-08

Testing Battle.net

└ A bit about Battle.net

└ Battle.net infrastructure

Battle.net infrastructure

- About 325,000 lines of C++
 - Servers + client libraries
- "Battle.net Game Service"
 - Authenticate players
 - Social: friends, presence
 - Matchmaking (cooperative/competitive)
 - Achievements/profiles

- Explain what battle.net does.
- Game service vs Battle.net umbrella.
- Define terms.

2015-04-08

Testing Battle.net

└ A bit about Battle.net

└ Battle.net is highly...

Battle.net is highly...

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

- Many machines connected.
- Almost everything asynchronous, callback-driven.
- Lots of configuration read at startup time from git repo.
- Code is pretty good, but size => faults occur.

2015-04-08

Testing Battle.net

└ A bit about Battle.net

└ A familiar situation

A familiar situation

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

- A familiar situation for me and my colleagues.
- Game industry is not accustomed to unit testing.

2015-04-08

Testing Battle.net

└ A bit about Battle.net

└ What's typically well-tested?

What's typically well-tested?

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

These things are "easy mode" for tests.

- Usually well-tested.
- Not worth thinking about edge cases - can use off-the-shelf tests (eg UTF-8).

2015-04-08

Testing Battle.net

└ A bit about Battle.net

└ Not-so-well tested?

Not-so-well tested?

- Matchmaking algorithms
- Queueing/Load balancing algorithms
- Other high-dependency, asynchronous, "large" code

These things are harder to test. Where to start?

- This is what I started to think about.
- I read Kent Beck and Bob Martin. I watched Misko Hevery.
- Conclusion: we aren't practised at testing.
- Need to practise - use TDD
- Extend unit testing framework as I go

2015-04-08

Testing Battle.net

└ A bit about Battle.net

└ No magic bullet

No magic bullet

- I wrote a lot 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

- My journey.
- Things were messy for a while. (They even shipped messy.)
- But I found some useful things to share.

2015-04-08

Testing Battle.net

└ Testing legacy code

└ Monolithic classes

Monolithic classes

Problem 1: Getting started testing huge legacy classes.

(What idiot wrote this code? Oh, it was me, 3 months ago...)

- in a codebase this size, there are some classes that get large
- and they do complex things
- and we need to test them

2015-04-08

Testing Battle.net

└ Testing legacy code

└ Exhibit A: hard to test

Exhibit A: hard to test

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

- Explain types.
- Deep inheritance that mixes concerns.
 1. What is RPC doing in there?
 2. And protocol dependency.
 3. "Traditional" interface-impl hierarchy.
- Constructor takes 6 args.
 1. Some constructor args have a wide interface.
 2. Again RPC.
 3. Lots of configuration.
 4. These things are onerous to mock.

2015-04-08

Testing Battle.net

└ Testing legacy code

└ Exhibit B: hard to test

Exhibit B: hard to test

```
class AchievementServiceImpl {
    public test: AchievementService { AchievementService
    public AchievementServiceStaticDataLoader
}
{
    public:
    AchievementServiceImpl(
        test: internal::ServerHelper serverHelper,
        mysql: Database* mysql);
}
class AchievementServiceImpl {
    public test: AchievementService { AchievementService
    public AchievementServiceStaticDataLoader
}
{
    public:
    AchievementServiceImpl(
        test: internal::ServerHelper serverHelper,
        mysql: Database* mysql);
}
class ServerHelper {
    public:
    ServerHelper(...); // 12 args!
}
rpc: RPCServer GetInsideRPCServer() const;
rpc: RPCServer GetOutsideRPCServer() const;
```

- Achievements actually quite well-tested
- Again the pattern of deriving from protocol
- Static data loader => IO going on in constructor?
- Some DI going on (database interface)
- Constructor args have wide interfaces
- ServerHelper legitimized the pattern of coupling IO/RPC and functionality

2015-04-08

Testing Battle.net

└ Testing legacy code

└ Patterns inimical to testing

Patterns inimical to testing

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

- Everyone tells us that dependency injection is required for testing
- But it's not enough
- RAIL is bad: testable things shouldn't own resources
- Wide interfaces to construction are bad

2015-04-08

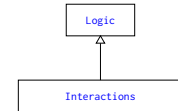
Testing Battle.net

└ Testing legacy code

└ Class structure for testing

Class structure for testing

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



- Component class (contains logic)

- Instead of "traditional" interface-impl split
- Use the split of logic vs interactions
 - Logic in base
 - Interactions in derived
 - Derived has as few dependencies as possible
 - Ruthlessly inject dependencies
- Good news: this is quite easy to apply

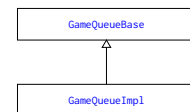
2015-04-08

Testing Battle.net

└ Testing legacy code

└ Example: Queueing for games

Example: Queueing for games



- Explain queueing for games.
- Manage multiple queues.
- Server capacity, link capacity. KR/TW problem.
- Rate limiting even in the presence of adequate server capacity.

2015-04-08

Testing Battle.net

└ Testing legacy code

└ Queueing for games

Queueing for games

GameQueueBase contains the queueing logic

```
class GameQueueBase
{
public:
    GameQueueBase(
        shared_ptr<ServerPoolInterface> interface,
        const PqCallback pqCb,
        const UpdateCallback updateCb,
        const PollTimeCallback pollTimeCb,
        const NotificationTimeCallback notificationTimeCb);

    bool Push(...);
    size_t Pqf(...);
    void Remove(...);
    size_t PollQueue(...);
    ...
};

class GameQueueImpl
{
public:
    GameQueueImpl(
        shared_ptr<ServerPoolInterface> interface,
```

- Moderately complex queueing logic all in the base.
- Logic in standalone class: no RPC inheritance.
- Constructor args have narrow interfaces.
 - callbacks (1-function interface)
 - server pool: a couple of functions for server capacity information
- Interface not cluttered with other concerns: just queueing stuff.

2015-04-08

Testing Battle.net

└ Testing legacy code

└ Queueing for games

Queueing for games

GameQueueImpl1 deals with protocols

```
class GameQueueImpl1
{
public:
    GameQueueImpl1(
        shared_ptr<ServerPoolInterface> interface,
        const PqCallback pqCb,
        const UpdateCallback updateCb,
        const PollTimeCallback pollTimeCb,
        const NotificationTimeCallback notificationTimeCb);

    // protocol handler functions
    virtual void AddToQueue(...);
    virtual void RemoveFromQueue(...);
    ...

    // system events
    bool DeInit(...);
    bool OnInit(...);
    void OnShutdown(...);
    void OnNewServerConnected(...);
    ...
};

class GameQueueImpl2
{
public:
    GameQueueImpl2(
        shared_ptr<ServerPoolInterface> interface,
```

- Derive impl from base, using the logic-interaction divide
- Derived class implements
 - rpc calls
 - config
 - interaction with system
- Some of this stays at the level of the impl
- Some is dependency-injected to control the logic
 - keep base testable with as little setup as poss

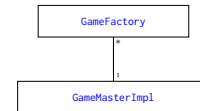
2015-04-08

Testing Battle.net

└ Testing legacy code

└ Example: Matchmaking

Example: Matchmaking



- Explain matchmaking
 - composition-based
 - game factory segments player base by difficulty, act, hardcore/non
 - deals with arbitrary groups of players
 - lots of telemetry
 - matching by attributes
- game factory implements matchmaking strategy

2015-04-08

Testing Battle.net

└ Testing legacy code

└ Matchmaking

Matchmaking

GameFactory contains matchmaking logic

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

    virtual bool Configure(const GameFactoryConfig& config);
    ...
    virtual Error RegisterPlayers(...);
    virtual bool UnregisterPlayers(...);
    virtual Error JoinGame(...);
    ...
};

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

- Small constructor interface
- Configuration required, but deferred => default config will be testable
 - Constructor leaves object initialised properly
- Just the MM logic in factory

2015-04-08

Testing Battle.net

└ Testing legacy code

└ Matchmaking

Matchmaking

GameMasterImpl deals with interactions

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

- system events
- config injection
- rpc interface

2015-04-08

Testing Battle.net

└ Testing legacy code

└ A successful pattern

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

- Side effect: not bad for optimization
 - layout: logic members at start of class
- If you have monolithic classes, you can start splitting logic out as a base class
 - you get something testable
 - once you have something testable, you can build on it
 - tested code is easier to refactor even if it starts out ugly

2015-04-08

Testing Battle.net

└ Testing legacy code

└ Testable classes

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.

- A practical guideline

2015-04-08

Testing Battle.net

└ Testing scalability (I)

└ Testing for scalability

Testing for scalability

Problem 2: Confidence in my code's ability to scale.

(I don't want a 3am call from devops.)

- The code has to work when a million players come along

2015-04-08

Testing Battle.net

└ Testing scalability (I)

└ Testing Performance/Efficiency

Testing Performance/Efficiency

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

- data set in the thousands =>
 - performance is king (cache effects etc)
 - algorithms not really important
- billions =>
 - usually highly distributed (can't run on dev machine)
 - use abstract algebra to achieve correct-by-construction algorithms
- millions =>
 - can run on a single machine
 - performance is important (caching)
 - but algorithms are also important
 - can run on dev machines but without scalable data sets

2015-04-08

Testing Battle.net

└ Testing scalability (I)

└ Problems in million-land

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

- Perf only gets you so far
- You need algorithms to avoid blowup at scale

2015-04-08

Testing Battle.net

└ Testing scalability (I)

└ Testing for performance

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

- Of course I can time tests
- This can give me some gross idea of optimizations
- It's still hard to do things properly (my desktop isn't the production hardware)

2015-04-08

Testing Battle.net

└ Testing scalability (I)

└ Efficiency: easy to lose

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

- I work with good engineers, but we're all human
- I was concerned about this
- I want the computer to help enforce this

2015-04-08

Testing Battle.net

└ Testing scalability (I)

└ Testing for efficiency

Testing for efficiency

- Run the same test with different sized inputs

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

$$\begin{aligned} T &\propto N \\ T_1 = T(N) &= aN \\ T_2 = T(kN) &= akN \\ \frac{T_2}{T_1} &= k \end{aligned}$$

- empirical method
- explain

2015-04-08

Testing Battle.net

└ Testing scalability (I)

└ Common cases

Common cases

$$\begin{aligned} O(1) &\Rightarrow \frac{T_2}{T_1} = 1 \\ O(\log n) &\Rightarrow \frac{T_2}{T_1} = 1 + \frac{\log(k)}{\log(N)} \\ O(n) &\Rightarrow \frac{T_2}{T_1} = k \\ O(n \log n) &\Rightarrow \frac{T_2}{T_1} = k \left(1 + \frac{\log(k)}{\log(N)}\right) \\ O(n^2) &\Rightarrow \frac{T_2}{T_1} = k^2 \end{aligned}$$

- simple math to get figures for each bucket I care about

- 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 ($N = 100$, $k = 32$)

- Statistical mitigation = run multiple times, discard outliers, average
 - be clear: this is for machine effects
 - multiple runs occur on the same data
- constants need to be big enough to elicit the required effect
- but small enough not to make the test slow
- fast, high frequency timing function is desirable
- The nice thing is that you don't need to run this optimized
 - optimization tends only to make things better

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
uint sampleTime1 = test->Run(0);
test->Random();

test->Setup();
// run 2 - with size kN
uint sampleTime2 = test->Run(0 * MULT);
...
const int MULT = 32;
const int N = 32;
...
// run 1 - with size N
uint sampleTime1 = test->Run(0);
test->Random();

test->Setup();
// run 2 - with size kN

```

- 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. . .

2015-04-08

Testing Battle.net

└─ Testing scalability (I)

└─ Let the test make them?

Let the test make them?

Result: a typical test

- ~40 lines setup
- ~10 lines timing
- ~5 lines actual logic
- ~5 lines test macros

Yuck.

- I was working with objects that needed some setup
- monolithic classes, remember?

2015-04-08

Testing Battle.net

└─ Testing scalability (I)

└─ Let the test make them?

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

- Shipped with this because sometimes good enough is good enough

2015-04-08

Testing Battle.net

└ Property-based testing

└ Autogenerating test inputs

Autogenerating test inputs

Problem 3: Generating test input automatically.

(Laziness, Impatience, Hubris. Check.)

- I'm a student of Haskell (Quickcheck)
- The idea of property-Based testing
- Usually established in languages with reflection
- Or sufficiently powerful type systems
- Explain property-based testing

2015-04-08

Testing Battle.net

└ Property-based testing

└ Wish-driven development

Wish-driven development

What I have

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

What I want

```
def PROPERTY(TestName, Suite, what storage is)
{
  // do something with x
  // that should be true for any input
  ...
  return property_holds;
}
```

- I need a way to generate values of "any type"
- There are lots of things we already do for any type (eg print)

2015-04-08

Testing Battle.net

└ Property-based testing

└ How to generate TYPE?

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...

- The basic form
- generation is some idea of how complex the generated thing is
- and plumb through a random seed for reproducibility

2015-04-08

Testing Battle.net

└ Property-based testing

└ Specializing Arbitrary<T>

Specializing Arbitrary<T>

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

- Explain
- Generating arithmetic types is easy

2015-04-08

Testing Battle.net

└ Property-based testing

└ Specializing Arbitrary<T>

Specializing Arbitrary<T>

For int-like types

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

static int generate(int, unsigned long int seed)
{
    switch (g)
    {
        case 0: return 0;
```

- (Code formatted for slide: in reality, I don't create a mersenne twister on the stack every call)
- For bools, it's trivial
- For chars, generate printable values

2015-04-08

Testing Battle.net

└ Property-based testing

└ Specializing Arbitrary<T>

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
 - ADT-like approach

- Compound types are made of other types of course
- Can be built up recursively

2015-04-08

Testing Battle.net

└ Property-based testing

└ Specializing Arbitrary<T>

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, [g] () {
            return Arbitrary<T>::generate(g, seed+1);
        });
    return v;
}
```

- Explain
- The idea of a "generation" deals with things like how long to make vectors, strings etc
- Generate for compound type works recursively by generating the contained types

2015-04-08

Testing Battle.net

└ Property-based testing

└ How to make a property test?

How to make a property test?

What I want

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

- So far, I know how to generate the type
- Now I needed to figure out how to deal with the test
- Normally, tests don't have arguments

2015-04-08

Testing Battle.net

└ Property-based testing

└ Test macros expand into functions

Test macros expand into functions

```
Macro...  
  
REF_PROPERTY(TestName, Suite, const string& s)  
{  
    ...  
}  
  
Expands to...  
  
struct NonceStruct  
{  
    ...  
    bool operator()(const string& s);  
};  
bool NonceStruct::operator()(const string& s)  
{  
    ...  
}
```

- the macro instantiates a function object
- I can discover the type of the operator() argument

2015-04-08

Testing Battle.net

└ Property-based testing

└ Discover the type of the function argument

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 B, typename A>  
struct function_traits<B(A)>  
{  
    using argType = A;  
};  
  
template <typename C, typename B, typename A>  
struct function_traits<B(C, A)>  
{  
    public function_traits<B(A)>  
};  
  
...  
  
template <typename T>  
struct function_traits  
{  
    public function_traits<decltype(&T::operator())>  
};
```

- googling function traits turns up something very like this
- explain (slowly)
- omitted further specializations dealing with various const & ref qualifiers
- now I know
 - The argument type to generate
 - How to generate it
- All I need to do is figure out how to write Run() for a property test
- I need to take the operator() function, whose type varies for each test
- And make it callable in a uniform way
- Single-function interface on a varying-type object
- tailor-made for type erasure

2015-04-08

Testing Battle.net

└ Property-based testing

└ Implement a Run function

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();
    }
    ...
};

// DEF_PROPERTY(testName, Suite, TYPE) becomes...
struct NonceStruct : public Test
{
    ...
    virtual bool Run() override
    {
        // Property will type-erase NonceStruct, discover its argument type
        NonceStruct n;
        ...
    }
};
```

- Run() function is inherited from Test: this is quite standard
- "this" is the struct whose operator() varies
 - gets type-erased by Property
- Property exposes check() which calls the type-erased operator()

2015-04-08

Testing Battle.net

└ Property-based testing

└ Property type-erases NonceStruct

Property type-erases NonceStruct

```
struct Property
{
    template<typename P>
    Property(const P& p)
    {
        m_internalTest = make_unique<InternalTest>(p);
    }

    bool check(...)
    {
        return m_internal->check(...);
    }
};

struct InternalBase
{
    virtual ~InternalBase() {}
    virtual bool check(...) = 0;
};

template<typename I>
struct Internal : public InternalBase
{
    ...
};

std::unique_ptr<InternalBase> m_internal;
...
};
```

- formatted for slide
- standard type-erasure pattern
- here's the constructor that's a template and captures the passed-in type
- here's the stored type-erased thing
- here's the exposed interface: the check function
- the omitted args are the generation and random seed params we saw earlier that will be used with the call to Arbitrary::generate
- let's look inside Internal

2015-04-08

Testing Battle.net

└ Property-based testing

└ Property type-erases NonceStruct

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(...);
        // pass it to the struct's operator()
        return m_t(p);
    }
};

T m_t;

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

- check generates a value using Arbitrary::generate
- passes it to the operator() of the NonceStruct

2015-04-08

Testing Battle.net

└ Property-based testing

└ Now we have property tests

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

- recap
- now we can use this ability to generate to power algorithmic tests
- but before we get to that, shrink

2015-04-08

Testing Battle.net

Property-based testing

Better checks for compound types

- borrowed from Quickcheck
- we can do more than just generate
- shrink returns a vector of T's

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 v; }
};
```

shrink returns a vector of "reduced" T's

2015-04-08

Testing Battle.net

Property-based testing

Better checks for compound types

- base case: return empty vector
- recurse, making the returned vector elements smaller
- for the containers, just use a binary search strategy
- explain how the calling code will follow failing cases

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 i = t.size() / 2;
    v.push_back(t.substr(0, i));
    v.push_back(t.substr(i));
    return v;
}

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 i = t.size() / 2;
    v.push_back(t.substr(0, i));
    v.push_back(t.substr(i));
    return v;
}
```

2015-04-08

Testing Battle.net

└ Testing scalability (II)

└ Algorithmic test inputs

Algorithmic test inputs

Problem 2 revisited: Generating input for algorithmic tests.

(I like to delete code.)

- Now I can take my property test code and apply it to the algorithmic complexity tests

2015-04-08

Testing Battle.net

└ Testing scalability (II)

└ Testing for efficiency (again)

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 ArbitraryT::generate(g, seed++);
        });
    return v;
}

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

- For algorithmic tests, we need to lock down a specific size
- Otherwise generate_n works exactly the same as generate
- the calling code doesn't need to follow failures
- these tests are just for timing

2015-04-08

Testing Battle.net

└ Testing scalability (II)

└ Now I can write

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

- can use $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$
- if the test comes in at or under the specified order, that's a pass
- specialize my own type generation:
 - random for average case data
 - bastard mode for worst case data
 - for ranges
 - unfortunately c++ has no newtype

2015-04-08

Testing Battle.net

└ Testing scalability (II)

└ Before and After

Before and After



- Get rid of
 - generation code
 - timing code
- refactor code made unnecessary by the new framework
- ~80 lines -> ~20 lines

2015-04-08

Testing Battle.net

└ Future thoughts

└ The reward for good work is more work

The reward for good work is more work

Status quo/future possibilities.

(People are never satisfied.)

- Arbitrary opens up new possibilities
- Next slide is a roundup

2015-04-08

Testing Battle.net

└ Future thoughts

└ Where I am now

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

- regular tests are still good
- property tests make you think harder
- in practice, the efficiency bar for Battle.net efficiency is $< O(n)$

2015-04-08

Testing Battle.net

└ Future thoughts

└ The future?

The future?

- Arbitrary opens the door for fuzz testing?
- Alternative walk strategies through the input space
 - Hilbert
 - Morton
 - etc
- Using Arbitrary to find poorly-performing data (P99)
- I'm still lazy; the computer isn't doing enough for me yet

- Fuzz testing is possible, but I didn't need it so much at the protocol level
 - Protobufs have sum and product types now
 - Illegal states can be unrepresentable
- Exercise poor performance in a couple of ways
 - Make tests do bad things
 - Make Arbitrary generation give bad data

2015-04-08

Testing Battle.net

└ Epilogue: more property-based testing

Epilogue: more property-based testing

Cool, can you do multiple arguments?

```
DEF_PROPERTY(TestName, Suite, count storage)
{
  // do something with x
  // that should be true for any input
  ...
  return property_holds;
}

DEF_PROPERTY(TestName, Suite, count storage x, int y)
{
  // do something with x, y
  // that should be true for any input
  ...
  return property_holds;
}
```

- when people see me riding a unicycle, they ask if I can juggle at the same time

2015-04-08

Testing Battle.net

└ DEF_PROPERTY uses __VA_ARGS__

DEF_PROPERTY uses __VA_ARGS__

```
define DEF_PROPERTY(NAME, SUITE, ID) \
...
bool operator()(const string& s, int i) \
{
...
}

Expands to...

struct HoundStruct
{
...
bool operator()(const string& s, int i)
{
...
}
};
bool HoundStruct::operator()(const string& s, int i)
{
...
}
```

- C++ has variadic macros in the standard now

2015-04-08

Testing Battle.net

└ function_traits captures args in a tuple

function_traits captures args in a tuple

```
template <typename R, typename... A>
struct function_traits {
    using argTuple = std::tuple<decay_t<A>...>;

    // apply a function to a tuple of arguments
    template <typename F>
    static R apply(F& f, const argTuple& a)
    {
        return unpackApply(f, a, std::index_sequence_for<A>...{});
    }

    template <typename F, std::size_t... Is>
    static R unpackApply(F& f, const argTuple& a, std::index_sequence<Is...>)
    {
        return f(std::get<Is>(a)...);
    }
    ...
};

template <typename R, typename... A>
struct function_traits {
    using argTuple = std::tuple<decay_t<A>...>;

    // apply a function to a tuple of arguments
    ...
};
```

- I was surprised how easy it was to apply a type transformation
- This is basically apply from the library fundamentals TS
 - But without the forwarding references

2015-04-08

Testing Battle.net

└ Shrinking tuples

Shrinking tuples

- All property tests effectively take tuples as arguments
- So I need a way to shrink tuples
- First, think about pair
 - shrink first -> vector
 - shrink second -> vector
 - cartesian product of vectors?

- tuples were a relatively late addition
- at first I didn't implement shrink very well
- I went back to pair and I had a comment there
- cartesian product not necessary because of machinery
- N+M solution

2015-04-08

Testing Battle.net

└ Shrinking pairs

Shrinking pairs

```
static std::vector<std::pair<T1, T2>> shrink(const std::pair<T1, T2>& p)
{
    std::vector<std::pair<T1, T2>> ret{};

    // shrink the first
    auto first_v = ArbitraryT1::shrink(p.first);
    for (T1& a : first_v)
    {
        ret.push_back(std::make_pair(a, p.second));
    }

    // shrink the second
    auto second_v = ArbitraryT2::shrink(p.second);
    for (T2& a : second_v)
    {
        ret.push_back(std::make_pair(p.first, a));
    }

    return ret;
}

static std::vector<std::pair<T1, T2>> shrink(const std::pair<T1, T2>& p)
{
    std::vector<std::pair<T1, T2>> ret{};

    // shrink the first
```

- I thought about doing the cartesian product
- like applicative on lists in haskell

```
ghci> [(\(x,y) -> (1,y)), (\(x,y) -> (2,y))] <*> [(0,4),(0,5),(0,6)]
[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6)]
```

- but the N+M solution works just fine
- when you're done following the first shrink path, use the second

2015-04-08

Testing Battle.net

└ From pairs to tuples

From pairs to tuples

```
• So I go to cppreference.com
  • make_tuple
  • tie
  • forward_as_tuple
  • std::get
  • tuple_cat
```

- I see these things (go through them)
- tuple_cat? hmmm...

2015-04-08

Testing Battle.net

└ From pairs to tuples

From pairs to tuples

```
• first is std::get<0>()
  • Or tuple_head()?
• second is tuple_tail()
• make_pair is tuple_cons
  • put a head together with a tail
```

(Pretend these functions exist so we can write shrink for tuples)

- I've done some functional programming

2015-04-08

Testing Battle.net

└ Shrinking tuples

- pretend these exist
- as with pairs, so with tuples
- first shrink the head, cons them onto the tail
- then shrink the tail (it will work recursively)
- cons normal heads on to the shrunk tails

Shrinking tuples

```
static std::vector<std::tuple<Ts...>> shrink(const std::tuple<Ts...>& t)
{
    std::vector<std::tuple<Ts...>> ret{};

    // shrink the head
    using T = std::decay_t;
    auto head, y = Arbitrary{}; shrink(y, get<0>(t));
    for (int i = 1; i < t.size(); ++i)
    {
        ret.push_back(std::tuple_cat(std::move(head), tuple_tail(t, i)));
    }

    return ret;
}

static std::vector<std::tuple<Ts...>> shrink(const std::tuple<Ts...>& t)
{
    std::vector<std::tuple<Ts...>> ret{};

    // shrink the tail recursively
    using T = std::decay_t;
    auto tail = shrink(tuple_tail(t, 1));
    ret.push_back(std::tuple_cat(tuple_head(t), tail));
}
```

2015-04-08

Testing Battle.net

└ tuple_cons and tuple_tail

- tuple_cons is easy (explain)
- tuple_tail (explain)
- the power of tuple, variadic templates and index_sequence is great

tuple_cons and tuple_tail

```
template<typename U, typename T>
auto tuple_cons(U&& u, T&& t)
{
    using Tuple = std::decay_t<T>;
    return tuple_cat(forward<0>(u), std::forward<1>(t));
}

template<typename U, typename T, std::size_t... Is>
auto tuple_cons(U&& u, T&& t, std::index_sequence<Is...>)
{
    return std::make_tuple(std::forward<0>(u),
        std::get<Is>(std::forward<1>(t))...);
}

template<typename T>
auto tuple_tail(T&& t)
{
    using Tuple = std::decay_t<T>;
    return tuple_cat(std::forward<1>(t),
        std::make_index_sequence<
            std::tuple_size<Tuple>() - 1>());
}

template<typename T, std::size_t... Is>
auto tuple_tail(T&& t, std::index_sequence<Is...>)
```

2015-04-08

Testing Battle.net

└ Shrinking tuples

Shrinking tuples

- Shrink head -> shrunken heads
- Cons shrunken heads onto normal tail
- Shrink tail -> shrunken tails
- Cons normal head onto shrunken tails

- easy
- and that's really it

2015-04-08

Testing Battle.net

└ Thanks for listening (again)

Thanks for listening (again)

C++14 code: <https://github.com/elbano/testinator>

Me: bdeane@blizzard.com, @ben_deane

Notes

- Introductory (short)
- Brief overview of Battle.net server topology
- The problem: moving beyond "easy-mode" unit testing of base libraries to testing real components with real interactions, IO, configuration, etc
- Designing for testability
- Separating and injecting dependencies
- Test-friendly class hierarchy design
- Identifying invariants, structuring logic for tests
- Testing strategies (and the C++ that powers them)
- Regular edge cases
- Planning for and testing failure in a distributed system
- Gaining confidence in scalability without incurring the cost of running a full environment*