

# Testing Battle.net

(Before deploying to millions of players)

Ben Deane

Principal Engineer, Blizzard Entertainment

bdeane@blizzard.com, @ben\_deane

13th May 2015

# Contents

- 1 A bit about Battle.net
- 2 Testing legacy code
- 3 Testing scalability (I)
- 4 Property-based testing
- 5 Testing scalability (II)
- 6 Future thoughts

# 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

# Battle.net is highly...

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

# 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

# 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?

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

These things are harder to test. Where to start?

# 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



# Monolithic classes

Problem 1: Getting started testing huge legacy classes.

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

# 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...

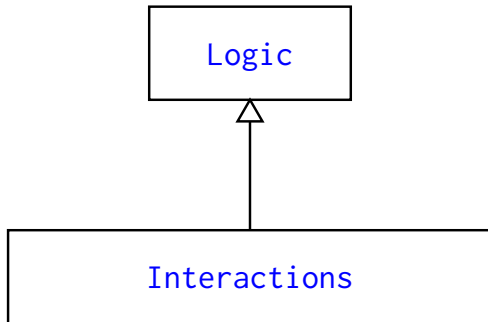
# Patterns inimical to testing

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



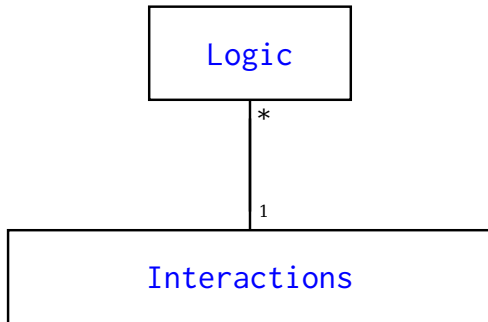
# Class structure for testing

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

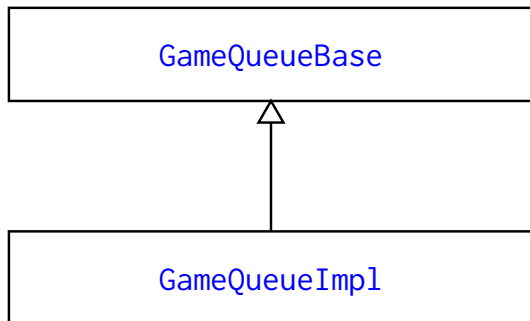


# Class structure for testing

- 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

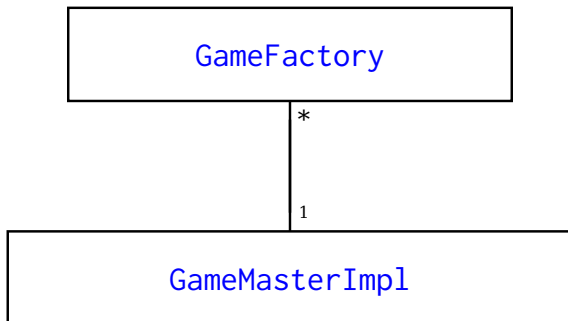
---

```
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:
    ...
    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:
    ...
    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:
    ...
    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

# 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 for scalability

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

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

# 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

# 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

- Run the same test with different sized inputs

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

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

$$T \propto N$$

$$T_1 = T(N) = aN$$

$$T_2 = T(kN) = akN$$

$$\frac{T_2}{T_1} = k$$

# Common cases

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

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

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

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

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

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

# Different-sized inputs

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);
...
```

# Different-sized inputs

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?

Result: a typical test

- ~40 lines 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



# Autogenerating test inputs

Problem 3: Generating test input automatically.

(Laziness, Impatience, Hubris. Check.)

# 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;
}
```

# 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 arithmetic 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
  - ADT-like approach

# 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;
}
```



# How to make a property test?

## 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& s);
};
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();
    }
    ...
};
```

# 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();
    }
    ...
};
```

# 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

```
struct Property
{
    template <typename F>
    Property(const F& f)
        : m_internal(std::make_unique<Internal<F>>(f))
    {}

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

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

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

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

# Property type-erases NonceStruct

```
struct Property
{
    template <typename F>
    Property(const F& f)
        : m_internal(std::make_unique<Internal<F>>(f))
    {}
```

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

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

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

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

# Property type-erases NonceStruct

```
struct Property
{
    template <typename F>
    Property(const F& f)
        : m_internal(std::make_unique<Internal<F>>(f))
    {}

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

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

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

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

# 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;
};
```

# A short demo

(Demo)

# 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

# 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

# Demo #2

(Demo)

# Algorithmic test inputs

Problem 2 revisited: Generating input for algorithmic tests.

(I like to delete code.)

# 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

# Demo #3

(Demo)



# Before and After

```
POWERSPECT_WTH_LEVEL(ConfigFactory, NatDef, test: RNGR_1, 1)
{
  ConfigFactory gf(wireline, FourCC(), 0);
  ConfigFactory(gf, 1, 0, 0);

  protocol_game_master: GameProperties(properties);
  protocol_attribute: AttributeFilter -> properties.mutable_filter();
  filter_west_of_protocol: Attribute -> AttributeFilter: NATCR_0_0;

  // Fill the joining game list with a lot of games.
  for (size_t a = 0; a < numElements; ++a)
  {
    AttributeList atts;
    atts.Append("Pos", Variant::MakeInt(a - 1));
    atts.SubValues(Filter::mutable_attribute());

    vector<game::Player> pfor_players;
    for (int i = 1; i <= 3; ++i)
    {
      game::Player p(for game::Player);
      p.w_id = NatDef(NatDef: RNGR_GAME_ACCOUNT, 0, 1);
      players.push_back(p);
    }
    rpe: ObjectAddress subscribe;
    GameRequest id = 0;
    GameId gameid = ChannelId1: GetTextChannelId(id);
    Error status = gf.RegisterPlayer(players, subscribe, properties, gameid, id);
    EXPECT_EQ(status, RNGR_OK);
  }
  EXPECT_EQ(GetFurValue("NumPlayersWatchingPos"), 2 * numElements);

  // Run measure mixing performance. Add a registration and match it, a
  // time.
  AttributeList atts;
  atts.Append("Pos", Variant::MakeInt(numElements - 1));
  atts.SubValues(Filter::mutable_attribute());

  a_momcall_ = 1000;
  int uniqueGameId = numElements;
  int start = microsec_clock::universal_time();
  for (size_t a = 0; a < a_momcall_ ++a, ++uniqueGameId)
  {
    {
      vector<game::Player> pfor_players;
      for (int i = 1; i <= 3; ++i)
      {
        game::Player p(for game::Player);
        p.w_id = NatDef(NatDef: RNGR_GAME_ACCOUNT, 0, 1);
        players.push_back(p);
      }
      rpe: ObjectAddress subscribe;
      GameRequest id = uniqueGameId;
      GameId gameid = ChannelId1: GetTextChannelId(uniqueGameId);
      Error status = gf.RegisterPlayer(players, subscribe, properties, gameid, id);
      EXPECT_EQ(status, RNGR_OK);
    }
    ++uniqueGameId;
  }

  {
    vector<game::Player> pfor_players;
    for (int i = 1; i <= 3; ++i)
    {
      game::Player p(for game::Player);
      p.w_id = NatDef(NatDef: RNGR_GAME_ACCOUNT, 0, 1);
      players.push_back(p);
    }
    rpe: ObjectAddress subscribe;
    GameRequest id = uniqueGameId;
    GameId gameid = ChannelId1: GetTextChannelId(uniqueGameId);
    Error status = gf.RegisterPlayer(players, subscribe, properties, gameid, id);
    EXPECT_EQ(status, RNGR_OK);
  }
}

time_duration t = microsec_clock::universal_time() - start;
EXPECT_EQ(GetFurValue("NumPlayersWatchingPos"), 2 * numElements);
EXPECT_EQ(GetFurValue("NumMonitors"), a_momcall_);
EXPECT_EQ(GetFurValue("NumPlayersWatching"), 4 * a_momcall_);

return t.total_microseconds();
}
```

# Before and After

```
POPMONITORSET_WITH_LEVEL(CppGameFactory, MatchPerf, test, ORDER_1, 1)

{
    CppGameFactory gf(Windows, FourCC(), 0);
    ConfigGameFactory(cfg, 1, 0, 0);

    protocol_game_master: GameProperties(properties);
    protocol_attribute: AttributeFilter: filter = properties.mutable_filter();
    filter->set_op(protocol_attribute: AttributeFilter: MATCH_ALL);

    // Fill the playing games list with a lot of games.
    for (size_t i = 0; i < numElements; ++i)
    {
        AttributeList attres;
        attres.Append("Pos", Variant::MakeInt64(-1));
        attres.SubFunction(Filter::mutable_attribute());

        vector<game::Player> Ptr: players;
        for (int i = 1; i <= 3; ++i)
        {
            game::Player: Ptr p(new game::Player);
            p->w_id = BitUtility::BitUtility::KIND_GAME_ACCOUNT(0, i);
            players.push_back(p);
        }

        rpe: ObjectAddress subscriber;
        GameRequestID id = 0;
        GameId gameId = ChannelId: GetDestChannelId(a);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }
    EXPECT_EQ(GetReturnValue("TwoPlayersWatchingPos"), 2 + numElements);

    // Now measure matching performance. Add a registration and match it, n
    // times.
    AttributeList attres;
    attres.Append("Pos", Variant::MakeInt(numElements - 1));
    attres.SubFunction(Filter::mutable_attribute());

    a_mumcall_ = 1000;
    int uniqueGameId = numElements;
    prime start = microsec_clock::universal_time();
    for (size_t i = 0; i < a_mumcall_; ++i, --uniqueGameId)
    {
        vector<game::Player> Ptr: players;
        for (int i = 1; i <= 3; ++i)
        {
            game::Player: Ptr p(new game::Player);
            p->w_id = BitUtility::BitUtility::KIND_GAME_ACCOUNT(0, i);
            players.push_back(p);
        }

        rpe: ObjectAddress subscriber;
        GameRequestID id = uniqueGameId;
        GameId gameId = ChannelId: GetDestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }

    --uniqueGameId;

    vector<game::Player> Ptr: players;
    for (int i = 1; i <= 3; ++i)
    {
        game::Player: Ptr p(new game::Player);
            p->w_id = BitUtility::BitUtility::KIND_GAME_ACCOUNT(0, i);
            players.push_back(p);
        }

        rpe: ObjectAddress subscriber;
        GameRequestID id = uniqueGameId;
        GameId gameId = ChannelId: GetDestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }

    time_duration t = microsec_clock::universal_time() - start;
    EXPECT_EQ(GetReturnValue("TwoPlayersWatchingPos"), 2 + numElements);
    EXPECT_EQ(GetReturnValue("NumConnections"), a_mumcall_);
    EXPECT_EQ(GetReturnValue("TwoPlayersInConnection"), 4 + a_mumcall_);

    return t.total_microseconds();
}
```

```
POPMONITORSET_WITH_LEVEL(CppGameFactory, MatchPerf, test, ORDER_1, 1)

{
    CppGameFactory gf(Windows, FourCC(), 0);
    ConfigGameFactory(cfg, 1, 0, 0);

    protocol_game_master: GameProperties(properties);
    protocol_attribute: AttributeFilter: filter = properties.mutable_filter();
    filter->set_op(protocol_attribute: AttributeFilter: MATCH_ALL);

    // Fill the playing games list with a lot of games.
    for (size_t i = 0; i < numElements; ++i)
    {
        AttributeList attres;
        attres.Append("Pos", Variant::MakeInt64(-1));
        attres.SubFunction(Filter::mutable_attribute());

        vector<game::Player> Ptr: players;
        for (int i = 1; i <= 3; ++i)
        {
            game::Player: Ptr p(new game::Player);
            p->w_id = BitUtility::BitUtility::KIND_GAME_ACCOUNT(0, i);
            players.push_back(p);
        }

        rpe: ObjectAddress subscriber;
        GameRequestID id = 0;
        GameId gameId = ChannelId: GetDestChannelId(a);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }
    EXPECT_EQ(GetReturnValue("TwoPlayersWatchingPos"), 2 + numElements);

    // Now measure matching performance. Add a registration and match it, n
    // times.
    AttributeList attres;
    attres.Append("Pos", Variant::MakeInt(numElements - 1));
    attres.SubFunction(Filter::mutable_attribute());

    a_mumcall_ = 1000;
    int uniqueGameId = numElements;
    prime start = microsec_clock::universal_time();
    for (size_t i = 0; i < a_mumcall_; ++i, --uniqueGameId)
    {
        vector<game::Player> Ptr: players;
        for (int i = 1; i <= 3; ++i)
        {
            game::Player: Ptr p(new game::Player);
            p->w_id = BitUtility::BitUtility::KIND_GAME_ACCOUNT(0, i);
            players.push_back(p);
        }

        rpe: ObjectAddress subscriber;
        GameRequestID id = uniqueGameId;
        GameId gameId = ChannelId: GetDestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }

    --uniqueGameId;

    vector<game::Player> Ptr: players;
    for (int i = 1; i <= 3; ++i)
    {
        game::Player: Ptr p(new game::Player);
            p->w_id = BitUtility::BitUtility::KIND_GAME_ACCOUNT(0, i);
            players.push_back(p);
        }

        rpe: ObjectAddress subscriber;
        GameRequestID id = uniqueGameId;
        GameId gameId = ChannelId: GetDestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }

    time_duration t = microsec_clock::universal_time() - start;
    EXPECT_EQ(GetReturnValue("TwoPlayersWatchingPos"), 2 + numElements);
    EXPECT_EQ(GetReturnValue("NumConnections"), a_mumcall_);
    EXPECT_EQ(GetReturnValue("TwoPlayersInConnection"), 4 + a_mumcall_);

    return t.total_microseconds();
}
```

# Before and After

```
PERFORMANCE_TEST_WITH_LEVEL(ConfigFactory, MatchPerf, test, ORDER_1, 1)
{
    ConfigFactory gf(testcase, PouchC(), 0);
    ConfigFactory(gf, 1, 0, 0);

    protocol_game_master: GameProperties properties;
    protocol_attribute: AttributeFilter filter = properties.mutable_filter();
    filter->set_op(protocol_attribute: AttributeFilter::MATCH_ALL);

    // Fill the playing games list with a lot of games.
    for (size_t s = 0; s < s_maximal_size; ++s)
    {
        AttributeList attr;
        attr.Append("Pos", Variant::MakeInt(s + 1));
        attr.SubValues(filter.mutable_attribute());

        vector<game::Player> Ptr players;
        for (int i = 1; i <= 3; ++i)
        {
            game::Player: Ptr p(new game::Player);
            p->id = EntryId(EntryId: KID0_GAME_ACCOUNT, 0, i);
            players.push_back(p);
        }
        rpe: ObjectAddress subscriber;
        GameRequestId id = s;
        Channel gameid = ChannelImpl::GetTestChannelId(id);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameid, id);
        EXPECT_EQ(status, ERROR_OK);
    }
    EXPECT_EQ(GetValue("NumPlayersWatchingPos"), 3 + s_maximal_size);

    // Now measure matching performance. Add a registration and match it, s
    // times.
    AttributeList attr;
    attr.Append("Pos", Variant::MakeInt(s_maximal_size + 1));
    attr.SubValues(filter.mutable_attribute());
    s_maximal_size_ = 1000;
    int uniqueGameId = s_maximal_size;
    prime start = microsec_clock::universal_time();
    for (size_t s = 0; s < s_maximal_size_ + s; ++s, --uniqueGameId)
    {
        vector<game::Player> Ptr players;
        for (int i = 1; i <= 3; ++i)
        {
            game::Player: Ptr p(new game::Player);
            p->id = EntryId(EntryId: KID0_GAME_ACCOUNT, 0, i);
            players.push_back(p);
        }
        rpe: ObjectAddress subscriber;
        GameRequestId id = uniqueGameId;
        Channel gameid = ChannelImpl::GetTestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameid, id);
        EXPECT_EQ(status, ERROR_OK);
    }
    --uniqueGameId;

    {
        vector<game::Player> Ptr players;
        for (int i = 1; i <= 3; ++i)
        {
            game::Player: Ptr p(new game::Player);
            p->id = EntryId(EntryId: KID0_GAME_ACCOUNT, 0, i);
            players.push_back(p);
        }
        rpe: ObjectAddress subscriber;
        GameRequestId id = uniqueGameId;
        Channel gameid = ChannelImpl::GetTestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameid, id);
        EXPECT_EQ(status, ERROR_OK);
    }

    time_duration t = microsec_clock::universal_time() - start;
    EXPECT_EQ(GetValue("NumPlayersWatchingPos"), 3 + s_maximal_size);
    EXPECT_EQ(GetValue("NumConnectors"), s_maximal_size_);
    EXPECT_EQ(GetValue("NumPlayersInGamePos"), 4 + s_maximal_size_);
}

return t.total_microseconds();
```

```
PERFORMANCE_TEST_WITH_LEVEL(ConfigFactory, MatchPerf, test, ORDER_1, 1)
{
    // Game master: GameProperties properties;
    protocol_attribute: AttributeFilter filter = properties.mutable_filter();
    filter->set_op(protocol_attribute: AttributeFilter::MATCH_ALL);

    // Now measure matching performance. Add a registration and match it, s
    // times.
    AttributeList attr;
    attr.Append("Pos", Variant::MakeInt(s_maximal_size + 1));
    attr.SubValues(filter.mutable_attribute());
    s_maximal_size_ = 1000;
    int uniqueGameId = s_maximal_size;
    prime start = microsec_clock::universal_time();
    for (size_t s = 0; s < s_maximal_size_ + s; ++s, --uniqueGameId)
    {
        vector<game::Player> Ptr players;

        rpe: ObjectAddress subscriber;
        GameRequestId id = uniqueGameId;
        Channel gameid = ChannelImpl::GetTestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameid, id);
        EXPECT_EQ(status, ERROR_OK);
    }
    --uniqueGameId;

    {
        vector<game::Player> Ptr players;

        rpe: ObjectAddress subscriber;
        GameRequestId id = uniqueGameId;
        Channel gameid = ChannelImpl::GetTestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameid, id);
        EXPECT_EQ(status, ERROR_OK);
    }

    time_duration t = microsec_clock::universal_time() - start;
    EXPECT_EQ(GetValue("NumPlayersWatchingPos"), 3 + s_maximal_size);
    EXPECT_EQ(GetValue("NumConnectors"), s_maximal_size_);
    EXPECT_EQ(GetValue("NumPlayersInGamePos"), 4 + s_maximal_size_);
}

return t.total_microseconds();
```

# Before and After

```
PERFORMANCE_TEST_WITH_LEVEL(ConfigFactory, MatchPerf, test, ORDER_1, 1)
{
    ConfigFactory cf(gameInfo, FourCC(), 0);
    ConfigFactory(cfgf, 1, 0, 0);

    protocol_game_gamester: GameProperties properties;
    protocol_attribute: AttributeFilter filter = properties.mutable_filter();
    filter->set_op(protocol_attribute: AttributeFilter::MATCH_ALL);

    // Fill the playing games list with a lot of games.
    for (size_t a = 0; a < numElements; ++a)
    {
        AttributeList attr;
        attr.Append("Pos", Variant::MakeInt(a - 1));
        attr.SubValues(filter.mutable_attribute());

        vector<game::Player> pfor_players;
        for (int i = 1; i <= 4; ++i)
        {
            game::Player p(for game::Player);
            p->id = EntryId(EntryId::KIND_GAME_ACCOUNT, 0, i);
            players.push_back(p);
        }
        rpe: ObjectAddress subscriber;
        GameId gameId = ChannelId::GetTestChannelId(a);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }
    EXPECT_EQ(GetVarValue("NumPlayersWatchingPos"), 2 + numElements);

    // Now measure matching performance. Add a registration and match it, a
    // time.
    AttributeList attr;
    attr.Append("Pos", Variant::MakeInt(numElements + 1));
    attr.SubValues(filter.mutable_attribute());

    a_numcalls_ = 1000;
    int uniqueGameId = numElements;
    int start = microsec_clock::universal_time();
    for (size_t a = 0; a < a_numcalls_; ++a, --uniqueGameId)
    {
        vector<game::Player> pfor_players;
        for (int i = 1; i <= 4; ++i)
        {
            game::Player p(for game::Player);
            p->id = EntryId(EntryId::KIND_GAME_ACCOUNT, 0, i);
            players.push_back(p);
        }
        rpe: ObjectAddress subscriber;
        GameId gameId = uniqueGameId;
        ChannelId channelId = GetTestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }
    --uniqueGameId;

    {
        vector<game::Player> pfor_players;
        for (int i = 1; i <= 4; ++i)
        {
            game::Player p(for game::Player);
            p->id = EntryId(EntryId::KIND_GAME_ACCOUNT, 0, i);
            players.push_back(p);
        }
        rpe: ObjectAddress subscriber;
        GameId gameId = uniqueGameId;
        ChannelId channelId = GetTestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }

    time_duration t = microsec_clock::universal_time() - start;
    EXPECT_EQ(GetVarValue("NumPlayersWatchingPos"), 2 + numElements);
    EXPECT_EQ(GetVarValue("NumMatchers"), a_numcalls_);
    EXPECT_EQ(GetVarValue("NumPlayerMatchers"), 4 + a_numcalls_);
}

return t.total_microseconds();
```

```
PERFORMANCE_TEST_WITH_LEVEL(ConfigFactory, MatchPerf, test, ORDER_1, 1)
{
    protocol_game_gamester: GameProperties properties;
    protocol_attribute: AttributeFilter filter = properties.mutable_filter();
    filter->set_op(protocol_attribute: AttributeFilter::MATCH_ALL);

    // Now measure matching performance. Add a registration and match it, a
    // time.
    AttributeList attr;
    attr.Append("Pos", Variant::MakeInt(numElements + 1));
    attr.SubValues(filter.mutable_attribute());

    int uniqueGameId = numElements;

    {
        vector<game::Player> pfor_players;

        rpe: ObjectAddress subscriber;
        GameId gameId = uniqueGameId;
        ChannelId channelId = GetTestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }
    --uniqueGameId;

    {
        vector<game::Player> pfor_players;

        rpe: ObjectAddress subscriber;
        GameId gameId = uniqueGameId;
        ChannelId channelId = GetTestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayer(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, ERROR_OK);
    }

    EXPECT_EQ(GetVarValue("NumPlayersWatchingPos"), 2 + numElements);
}
```

# Before and After

```
PERFORMANCE_TEST_WITH_LEVEL(ConfigFactory, MatchPerf, test, ORDER_1, 1)
{
    ConfigFactory gf(testEnv, FauxCC(), 0);
    ConfigFactory(gf, 1, 0, 0);

    protocol: game_master: GameProperties properties;
    protocol: attribute: AttributeFilter filter = properties.mutable_filter();
    filter->set_op(protocol: attribute: AttributeFilter: MATCH_ALL);

    // Fill the playing games list with a lot of games.
    for (size_t a = 0; a < numElements, ++a)
    {
        AttributeList attr;
        attr.Append("foo", Variant::MakeInt(a + 1));
        attr.SubVariant(filter->mutable_attribute());

        vector<game: Player> pfor players;
        for (int i = 1; i <= 5; ++i)
        {
            game: Player p(for game: Player);
            p->w_id = IntifyId(IntifyId: KID0_GAME_ACCOUNT, 0, i);
            players.push_back(p);
        }
        rpe: ObjectAddress subscriber;
        GameRequestId id = 0;
        GameId gameId = ChannelId1: GetTestChannelId(id);
        Error status = gf.RegisterPlayers(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, KERR_OK);
    }
    EXPECT_EQ(GetFwValue("NumLayersOutstanding"), 2 + numElements);

    // Now measure matching performance. Add a registration and match it, n
    // times.
    AttributeList attr;
    attr.Append("foo", Variant::MakeInt(numElements + 1));
    attr.SubVariant(filter->mutable_attribute());

    a_mscalls_ = 1000;
    int uniqueGameId = numElements;
    int64 start = microsec_clock::universal_time();
    for (size_t a = 0; a < a_mscalls_; ++a, ++uniqueGameId)
    {
        vector<game: Player> pfor players;
        for (int i = 1; i <= 5; ++i)
        {
            game: Player p(for game: Player);
            p->w_id = IntifyId(IntifyId: KID0_GAME_ACCOUNT, 0, i);
            players.push_back(p);
        }
        rpe: ObjectAddress subscriber;
        GameRequestId id = uniqueGameId;
        GameId gameId = ChannelId1: GetTestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayers(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, KERR_OK);
    }
    --uniqueGameId;

    {
        vector<game: Player> pfor players;
        for (int i = 1; i <= 5; ++i)
        {
            game: Player p(for game: Player);
            p->w_id = IntifyId(IntifyId: KID0_GAME_ACCOUNT, 0, i);
            players.push_back(p);
        }
        rpe: ObjectAddress subscriber;
        GameRequestId id = uniqueGameId;
        GameId gameId = ChannelId1: GetTestChannelId(uniqueGameId);
        Error status = gf.RegisterPlayers(players, subscriber, properties, gameId, id);
        EXPECT_EQ(status, KERR_OK);
    }

    time_duration t = microsec_clock::universal_time() - start;
    EXPECT_EQ(GetFwValue("NumLayersOutstanding"), 2 + numElements);
    EXPECT_EQ(GetFwValue("NumConnections"), a_mscalls_);
    EXPECT_EQ(GetFwValue("NumLayersOutstanding"), 4 + a_mscalls_);
}

return t.total_microseconds();
```

```
PERFORMANCE_TEST_WITH_LEVEL(ConfigFactory, MatchPerf, test, ORDER_1, 1)
{
    protocol: game_master: GameProperties properties;
    protocol: attribute: AttributeFilter filter = properties.mutable_filter();
    filter->set_op(protocol: attribute: AttributeFilter: MATCH_ALL);

    // Now measure matching performance. Add a registration and match it, n
    // times.
    AttributeList attr;
    attr.Append("foo", Variant::MakeInt(numElements + 1));
    attr.SubVariant(filter->mutable_attribute());

    vector<game: Player> pfor players;

    rpe: ObjectAddress subscriber;
    GameRequestId id = uniqueGameId;
    GameId gameId = ChannelId1: GetTestChannelId(uniqueGameId);
    Error status = gf.RegisterPlayers(players, subscriber, properties, gameId, id);
    EXPECT_EQ(status, KERR_OK);
}

vector<game: Player> pfor players;

rpe: ObjectAddress subscriber;
GameRequestId id = uniqueGameId;
GameId gameId = ChannelId1: GetTestChannelId(uniqueGameId);
Error status = gf.RegisterPlayers(players, subscriber, properties, gameId, id);
EXPECT_EQ(status, KERR_OK);
}

EXPECT_EQ(GetFwValue("NumLayersOutstanding"), 2 + numElements);
```

# The reward for good work is more work

Status quo/future possibilities.

(People are never satisfied.)

# 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
- Using Arbitrary to find poorly-performing data (P99)
- 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 safety net for correctness/efficiency/scalability.

# Thanks for listening

*Errors using inadequate data are much less  
than those using no data at all.*

- Charles Babbage

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

Me: bdeane@blizzard.com, @ben\_deane

# Epilogue: more on properties

Cool, can you do multiple arguments?

---

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

# Epilogue: more on properties

Cool, can you do multiple arguments?

---

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

# DEF\_PROPERTY uses \_\_VA\_ARGS\_\_

```
define DEF_PROPERTY(NAME, SUITE, ...) \
... \
bool operator()(__VA_ARGS__)

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

Expands to...

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

# function\_traits captures args in a tuple

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

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

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

# function\_traits captures args in a tuple

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

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

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

# 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?



# 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 = Arbitrary<T1>::shrink(p.first);
    for (T1& e : first_v)
    {
        ret.push_back(std::make_pair(std::move(e), p.second));
    }

    // shrink the second
    auto second_v = Arbitrary<T2>::shrink(p.second);
    for (T2& e : second_v)
    {
        ret.push_back(std::make_pair(p.first, std::move(e)));
    }

    return ret;
}
```

# 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 = Arbitrary<T1>::shrink(p.first);
    for (T1& e : first_v)
    {
        ret.push_back(std::make_pair(std::move(e), p.second));
    }

    // shrink the second
    auto second_v = Arbitrary<T2>::shrink(p.second);
    for (T2& e : second_v)
    {
        ret.push_back(std::make_pair(p.first, std::move(e)));
    }

    return ret;
}
```

# From pairs to tuples

- So I go to [cppreference.com](http://cppreference.com)
  - `make_tuple`
  - `tie`
  - `forward_as_tuple`
  - `std::get`
  - `tuple_cat`

# 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)

# Shrinking tuples

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

    // shrink the head
    using H = std::decay_t<decltype(std::get<0>(t))>;
    auto head_v = Arbitrary<H>::shrink(std::get<0>(t));
    for (H& e : head_v)
    {
        ret.push_back(tuple_cons(std::move(e), tuple_tail(t)));
    }

    ...

    return ret;
}
```

# Shrinking tuples

```
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<decltype(tuple_tail(t))>;
    auto tail_v = Arbitrary<T>::shrink(tuple_tail(t));
    for (T& e : tail_v)
    {
        ret.push_back(tuple_cons(std::get<0>(t), std::move(e)));
    }

    return ret;
}
```

# 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_cons(std::forward<U>(u), std::forward<T>(t),
                      std::make_index_sequence<std::tuple_size<Tuple>::value>());
}

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<U>(u),
                           std::get<Is>(std::forward<T>(t))...);
}
```

# tuple\_cons and tuple\_tail

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

```
template <typename T, std::size_t ...Is>
auto tuple_tail(T&& t, std::index_sequence<Is...>)
{
    return std::make_tuple(std::get<Is + 1>(std::forward<T>(t))...);
}
```



# Shrinking tuples

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

# Thanks for listening (again)

C++14 code: `https://github.com/elbeno/testinator`

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