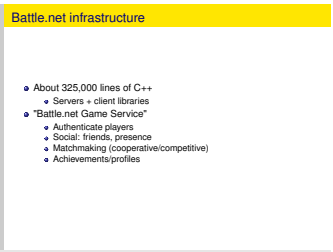
Testing Battle.net

Battle.net infrastructure

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

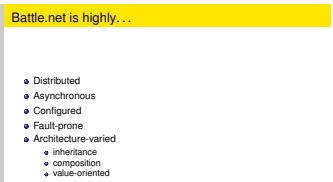


2015-04-06

Testing Battle.net

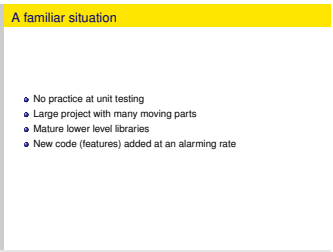
Battle.net is highly. . .

- 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.



Testing Battle.net

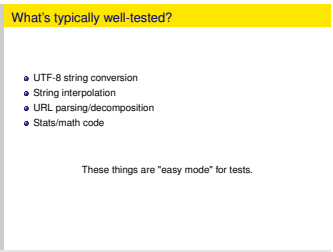
A familiar situation



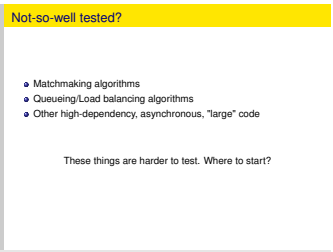
- A familiar situation for me and my colleagues.
- Game industry is not accustomed to unit testing.

Testing Battle.net

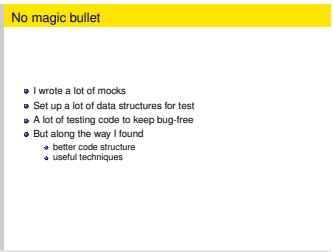
What's typically well-tested?



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



- 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!



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

Monolithic classes

Problem 1: Getting started testing huge legacy classes

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

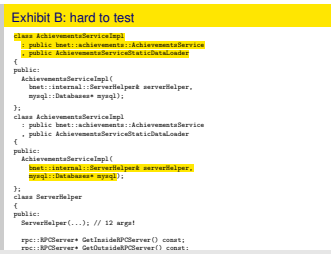
Exhibit A: hard to test

```
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);
};
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,
```

- 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.

Testing Battle.net

Exhibit B: hard to test

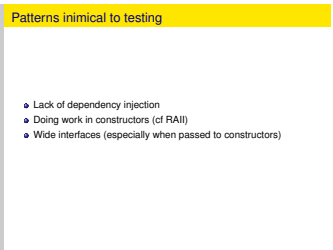


- Achievements actually quite well-tested
- Again the pattern of deriving from protocol
- Static data loader => IO going on in constructor?
- Some DI going on (database interface)
- Constructor args have wide interfaces
- ServerHelper legitimized the pattern of coupling IO/RPC and functionality

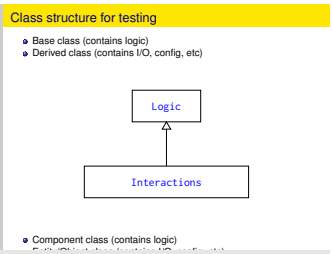
2015-04-06

Testing Battle.net

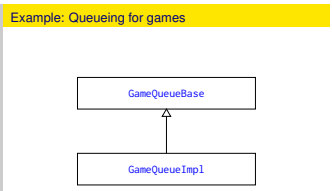
Patterns inimical to testing



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

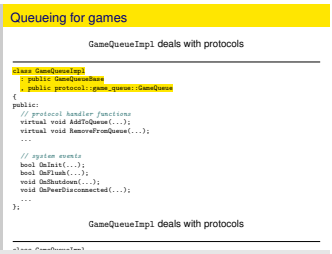


- Instead of "traditional" interface-impl split
- Use the split of logic vs interactions
  - Logic in base
  - Interactions in derived
  - Derived has as few dependencies as possible
  - Ruthlessly inject dependencies
- Good news: this is quite easy to apply



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

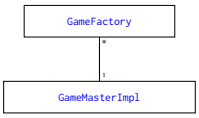
- Moderately complex queueing logic all in the base.
- Logic in standalone class: no RPC inheritance.
- Constructor args have narrow interfaces.
  - callbacks (1-function interface)
  - server pool: a couple of functions for server capacity information
- Interface not cluttered with other concerns: just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



- 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

Testing Battle.net

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

2015-04-06

Testing Battle.net

Matchmaking

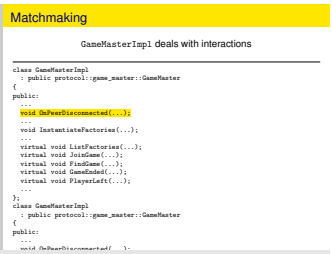
`GameFactory` contains matchmaking logic

```
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(...);
  ...
};
class GameFactory
{
public:
  GameFactory(const AttributeValue& 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

Testing Battle.net

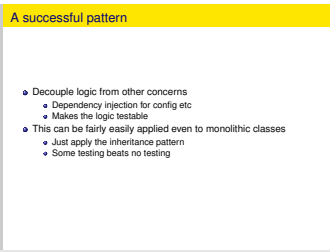
Matchmaking



- system events
- config injection
- rpc interface

Testing Battle.net

A successful pattern



- 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

Testing Battle.net

Testable classes

- A practical guideline

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.

2015-04-06

Testing Battle.net

Testing for scalability

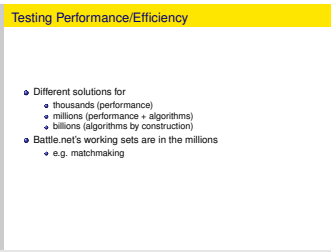
- The code has to work when a million players come along

Testing for scalability

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

Testing Battle.net

Testing Performance/Efficiency

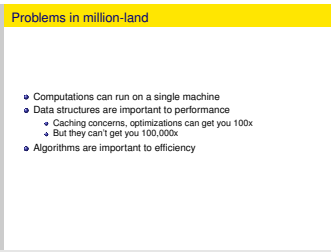


- data set in the thousands =>
  - performance is king (cache effects etc)
  - algorithms not really important
- billions =>
  - usually highly distributed (can't run on dev machine)
  - use abstract algebra to achieve correct-by-construction algorithms
- millions =>
  - can run on a single machine
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

2015-04-06

Testing Battle.net

Problems in million-land



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

Testing Battle.net

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)

2015-04-06

Testing Battle.net

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

2015-04-06

Testing Battle.net

Testing for efficiency

- empirical method
- explain

Testing for efficiency

- Run the same test with different sized inputs

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

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

2015-04-06

Testing Battle.net

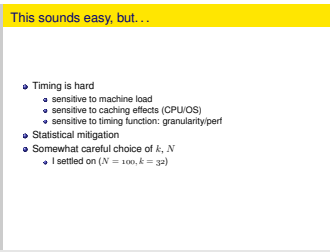
Common cases

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

Common cases

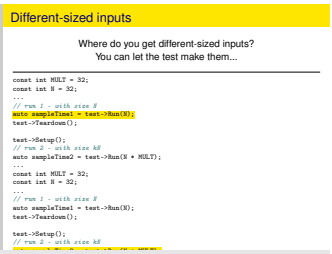
$$O(1) \Rightarrow \frac{T_2}{T_1} = 1$$
$$O(\log n) \Rightarrow \frac{T_2}{T_1} = 1 + \frac{log(k)}{log(N)}$$
$$O(n) \Rightarrow \frac{T_2}{T_1} = k$$
$$O(n \log n) \Rightarrow \frac{T_2}{T_1} = k\,(1 + \frac{log(k)}{log(N)})$$
$$O(n^2) \Rightarrow \frac{T_2}{T_1} = k^2$$

This sounds easy, but...



- Statistical mitigation = run multiple times, discard outliers, average
- 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

Different-sized inputs

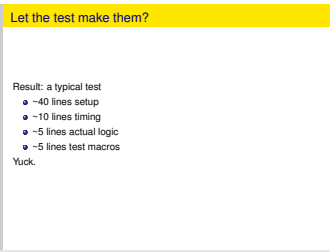


- Affects the timing if done naively (i.e. wrongly)
  - Adds an $O(n)$ component to the test
  - So move the timing code inside the test also
- Boilerplate in test code
- It's not ideal...

Testing Battle.net

Let the test make them?

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

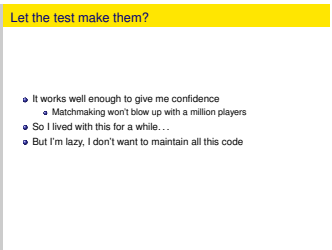


2015-04-06

Testing Battle.net

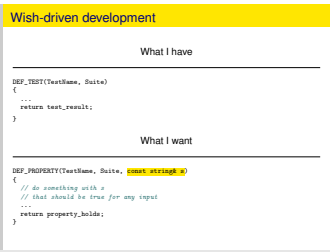
Let the test make them?

- 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



Testing Battle.net

Wish-driven development

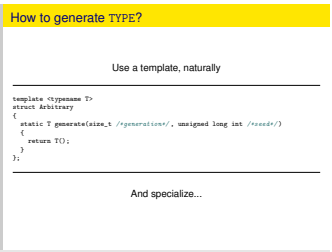


- I need a way to generate values of "any type"
- There are lots of things we already do for any type (eg print)

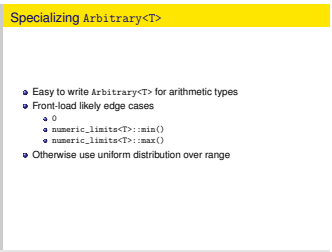
2015-04-06

Testing Battle.net

How to generate `TYPE`?

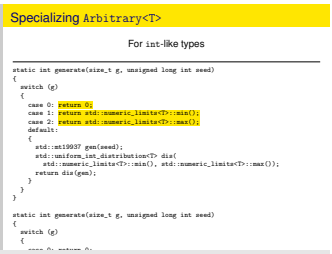


- The basic form
- generation is some idea of how complex the generated thing is
- and plumb through a random seed for reproducibility



- Explain
- Generating arithmetic types is easy

2015-04-06 Testing Battle.net
└Specializing Arbitrary<T>



- (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

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

Specializing `Arbitrary<T>`

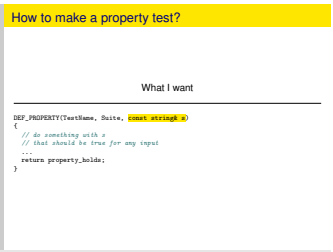
For compound types (eg `vector`)

```
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;
}
```

- 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

Testing Battle.net

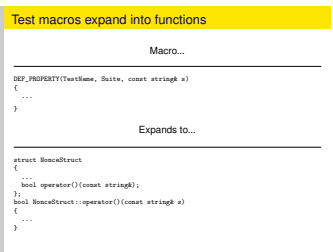
How to make a property test?



- 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

Testing Battle.net

Test macros expand into functions



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

Testing Battle.net

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 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)>
{};

...

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

Testing Battle.net

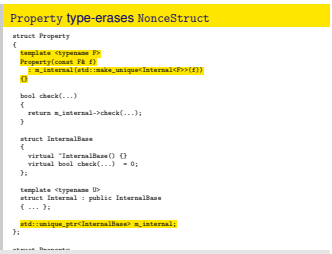
Implement a `Run` function

Implement a `Run` function

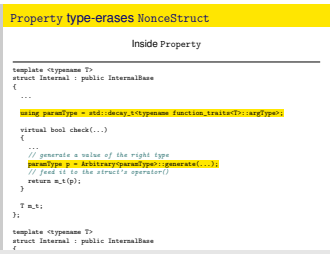
`Run()` for a property test

```
// 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
```

- 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()



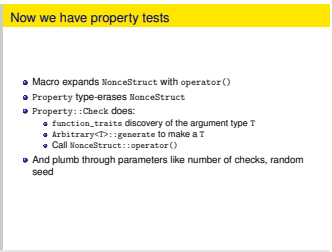
- formatted for slide
- standard type-erasure pattern
- here's the constructor that's a template and captures the passed-in type
- here's the stored type-erased thing
- here's the exposed interface: the check function
- the omitted args are the generation and random seed params we saw earlier that will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



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

Testing Battle.net

Now we have property tests

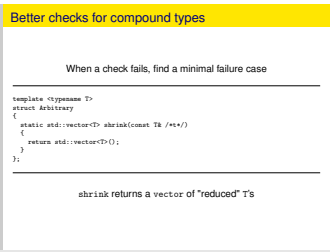


- recap
- now we can use this ability to generate to power algorithmic tests
- but before we get to that, shrink

2015-04-06

Testing Battle.net

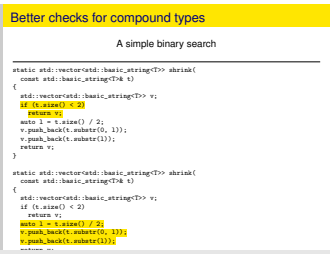
Better checks for compound types



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

Testing Battle.net

Better checks for compound types

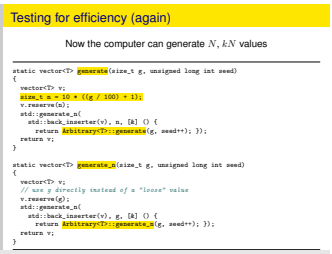


- base case: return empty vector
- recurse, making the returned vector elements smaller
- for the containers, just use a binary search strategy
- explain how the calling code will follow failing cases

2015-04-06

Testing Battle.net

Testing for efficiency (again)

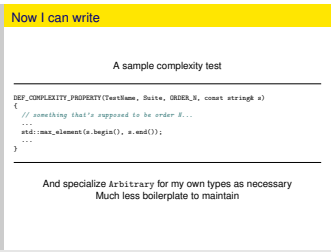


- For algorithmic guarantees, we need to lock down a specific size
- Otherwise generate_n works exactly the same as generate
- the calling code doesn't need to follow failures
- these tests are just for timing

2015-04-06

Testing Battle.net

Now I can write

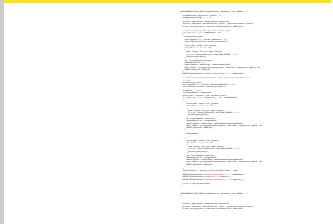


- can use O(1), O(log n), O(n), O(n log n), O(nˆ2)
- if the test comes in at or under the specified order, that's a pass

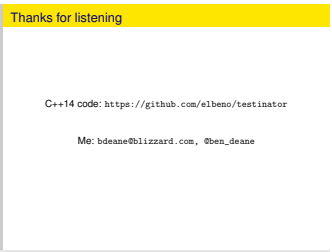
2015-04-06

Testing Battle.net

Before and After



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



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*
- Property-based testing*

> ——— REVIEW ——— > I'm alway trying to increase the testing-related content at C++Now. > > In my opinion, C++ programmers don't talk and think enough about > testing and don't design for testability. It think that younger > languages were created after the TDD revolution and TDD is embedded in > their culture. This is not true for C++, so we have to play catchup. > Sessions like the one that Ben has submitted are step in that direction. > > Ben's project was designed with testability in mind and was also one > with a particular testing challenge (testing at the required scale > isn't really possible). The lessons learned from this project will be > valuable to C++ developers. > > > ——————— REVIEW 2 ——————— > PAPER: 3 > TITLE: Testing Battle.net (before deploying to millions of players) > AUTHORS: Ben Deane > > OVERALL EVALUATION: 3 (strong accept) > REVIEWER'S CONFIDENCE: 4 (high) > > ——— REVIEW ——— > Let's not waste time on commenting a keynote-quality submission. > > > ——————— REVIEW 3 ——————— > PAPER: 3 > TITLE: Testing Battle.net (before deploying to millions of players) > AUTHORS: Ben Deane > > OVERALL EVALUATION: 3 (strong accept) > REVIEWER'S CONFIDENCE: 4 (high) > > ———