

# Value Proposition:

## *Allocator-Aware (AA) Software*

John Lakos

Saturday, April 13, 2019

*This version is for ACCU'19.*

# Copyright Notice

© 2019 Bloomberg L.P. Permission is granted to copy, distribute, and display this material, and to make derivative works and commercial use of it. The information in this material is provided "AS IS", without warranty of any kind. Neither Bloomberg nor any employee guarantees the correctness or completeness of such information. Bloomberg, its employees, and its affiliated entities and persons shall not be liable, directly or indirectly, in any way, for any inaccuracies, errors or omissions in such information. Nothing herein should be interpreted as stating the opinions, policies, recommendations, or positions of Bloomberg.

# Abstract

The performance benefits of supplying local allocators are well-known and substantial [Lakos, ACCU'17]. Still, the real-world costs associated with orchestrating the integration of allocators throughout a code base, including training, supporting tools, enlarged interfaces (and contracts), and a heightened potential for inadvertent misuse cannot be ignored. Despite substantial upfront costs, when one considers collateral benefits for clients – such as rapid prototyping of alternative allocation strategies – the case for investing in a fully *allocator-aware* (AA) software infrastructure (SI) becomes even more compelling. Yet there remain many “concerns” based on hearsay or specious conjecture that are either overstated or incorrect.

In this densely fact-infused talk, we begin by introducing a familiar analogy to drive home the business case for AASI. Next we identify four syntactic styles based on three distinct models: C++11, C++17, and a brand new language-based approach being developed by Bloomberg for C++23 (or later). Costs – both real and imagined – will be contrasted with performance as well as other important (“collateral”) benefits. The talk will conclude with a closer look at the economic imperative of pursuing a low-cost language-based alternative to AA software in post-modern C++.

# Purpose of this Talk

Current state of affairs...

- Local Allocators -> performance!! [Lakos, ACCU'17]
- There are, however, *real-world costs*
- There are also important *collateral benefits*
- Yet there remain “concerns” (a.k.a. F.U.D.)

# Purpose of this Talk

What we will do today ...

- Present the four AA software styles
- Separate real from imagined costs
- Discuss important collateral benefits of AA
- Address common “concerns” surrounding AA
- Advocate for supporting AASI today
- Make business case using detailed analogy
- Hint at what C++2y allocators might look like

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”

# Introduction

Dynamic memory allocation is important!

- `new/delete` usually adequate
- Custom allocation is sometimes advantageous
  - (and sometimes it's absolutely necessary)
- But implementing custom allocation is costly.
- Thus, we are motivated to create (now):  
*Allocator-Aware (AA) Software Infrastructure*
  - (and soon): *BB20V* (Bloomberg's 2020 Vision)



# Introduction

Two approaches to custom memory allocation:

- Design bespoke (custom) data structures when needed.
  - Best possible performance
  - High development/maintenance costs
- Build on *Allocator Aware (AA)* components
  - *Nearly* best possible performance
  - Much lower costs

# Introduction

Two approaches to custom memory allocation:

- Design bespoke (custom) data structures when needed.
  - Best possible performance
  - High development/maintenance costs
- Build on *Allocator Aware (AA)* components
  - *Nearly* best possible performance
  - Much lower costs + **some collateral benefits**

# Introduction

Airline Analogy to Allocator Awareness (AA):

- *First Class*
  - Best possible
- *Economy*
  - Cheapest possible

# Introduction

Airline Analogy to Allocator Awareness (AA):

- *First Class*
  - Best possible
- *Economy*
  - Cheapest possible

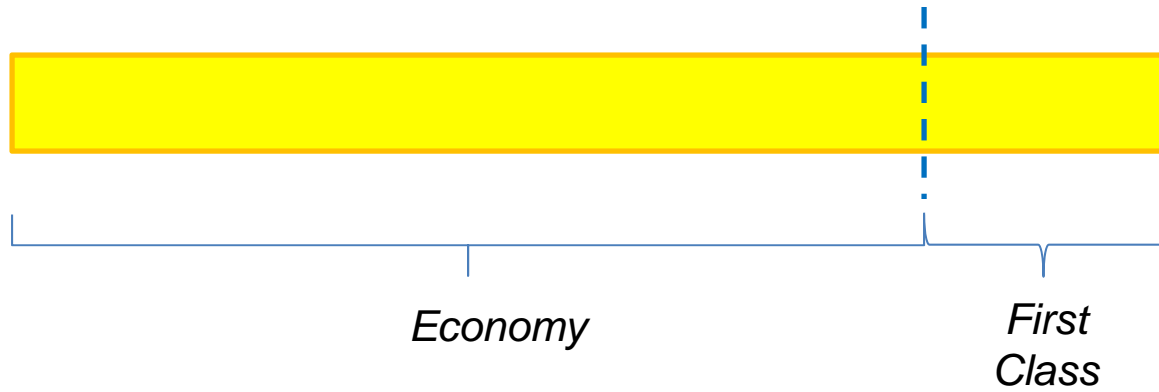


Client  
Perspective

Client  
Perspective

# Introduction

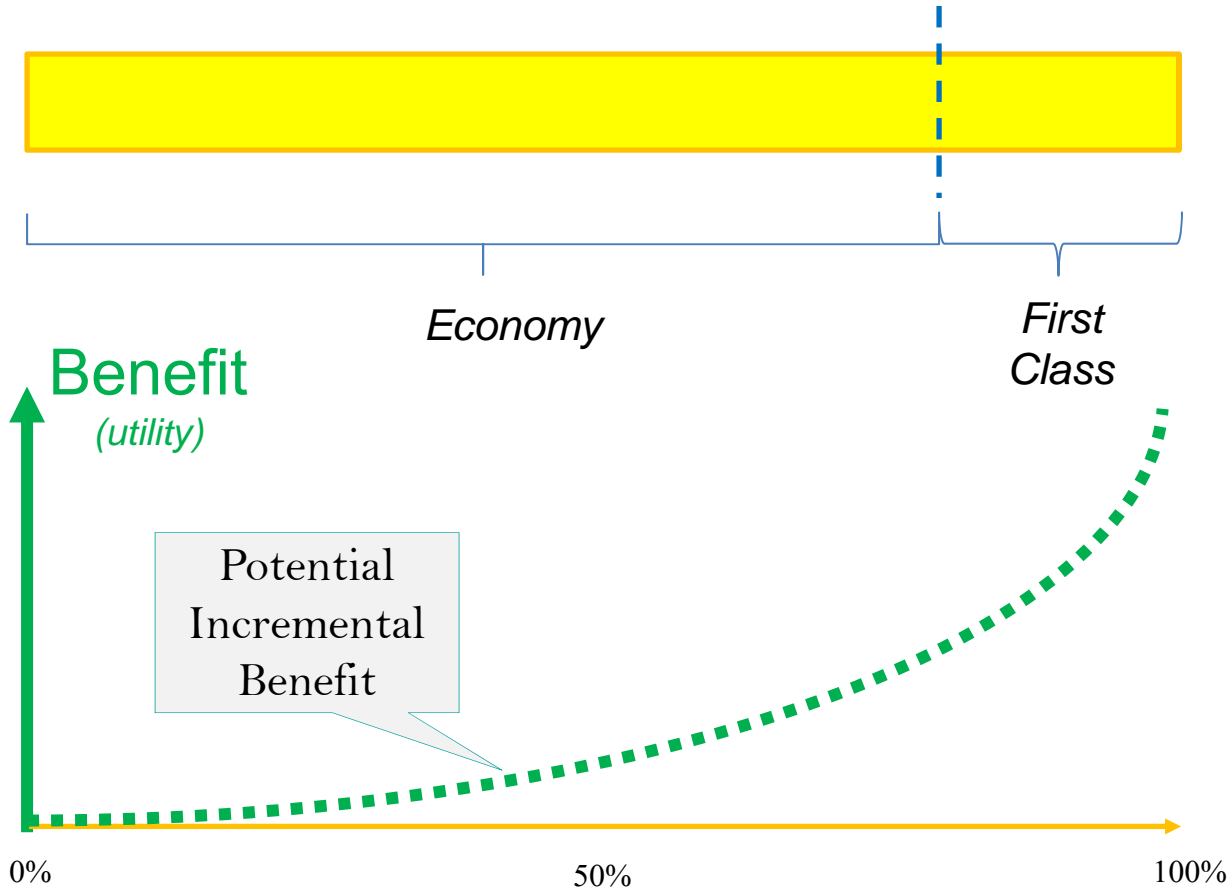
Client  
Perspective



Client Perspective

# Introduction

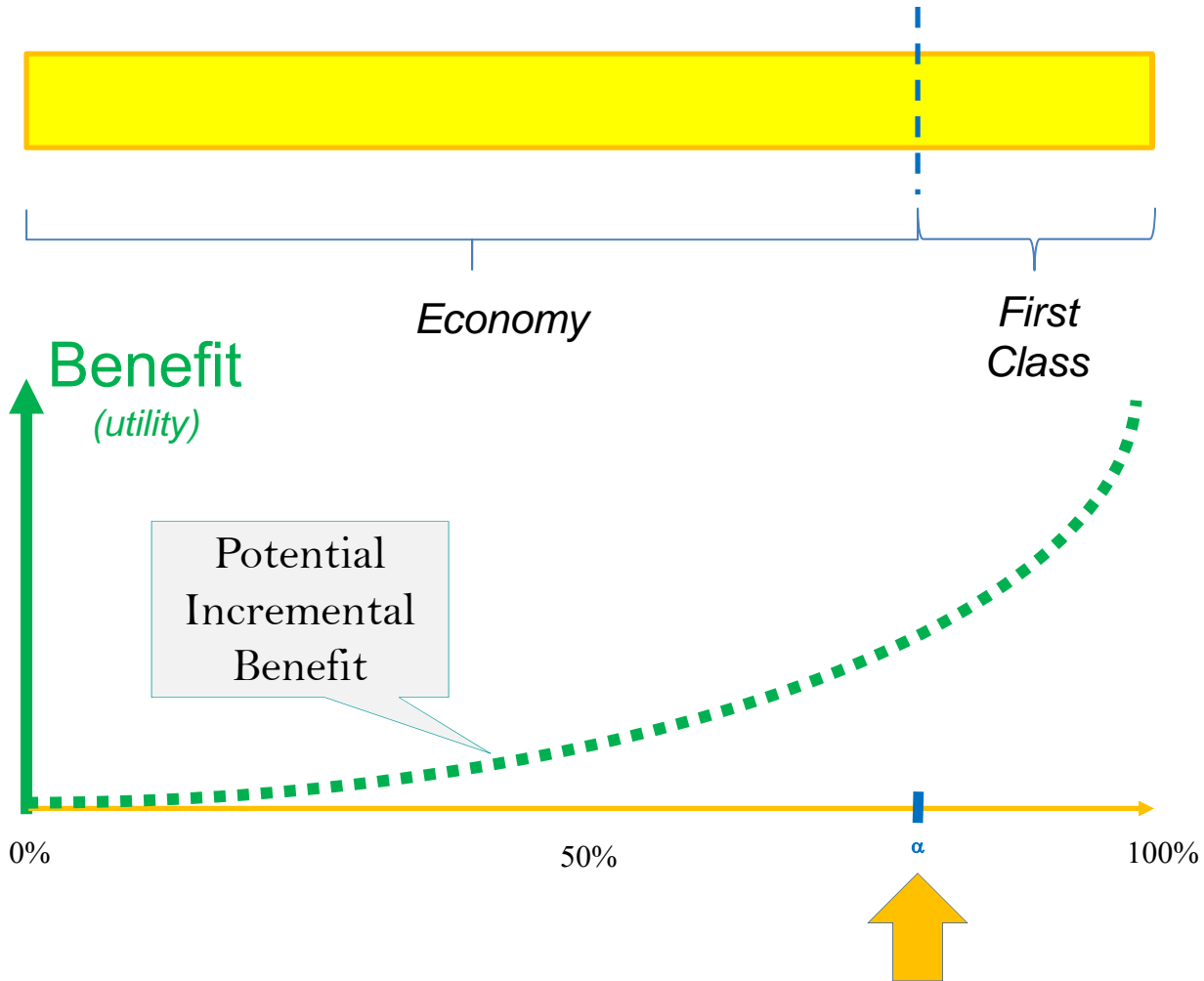
Client Perspective



Client Perspective

# Introduction

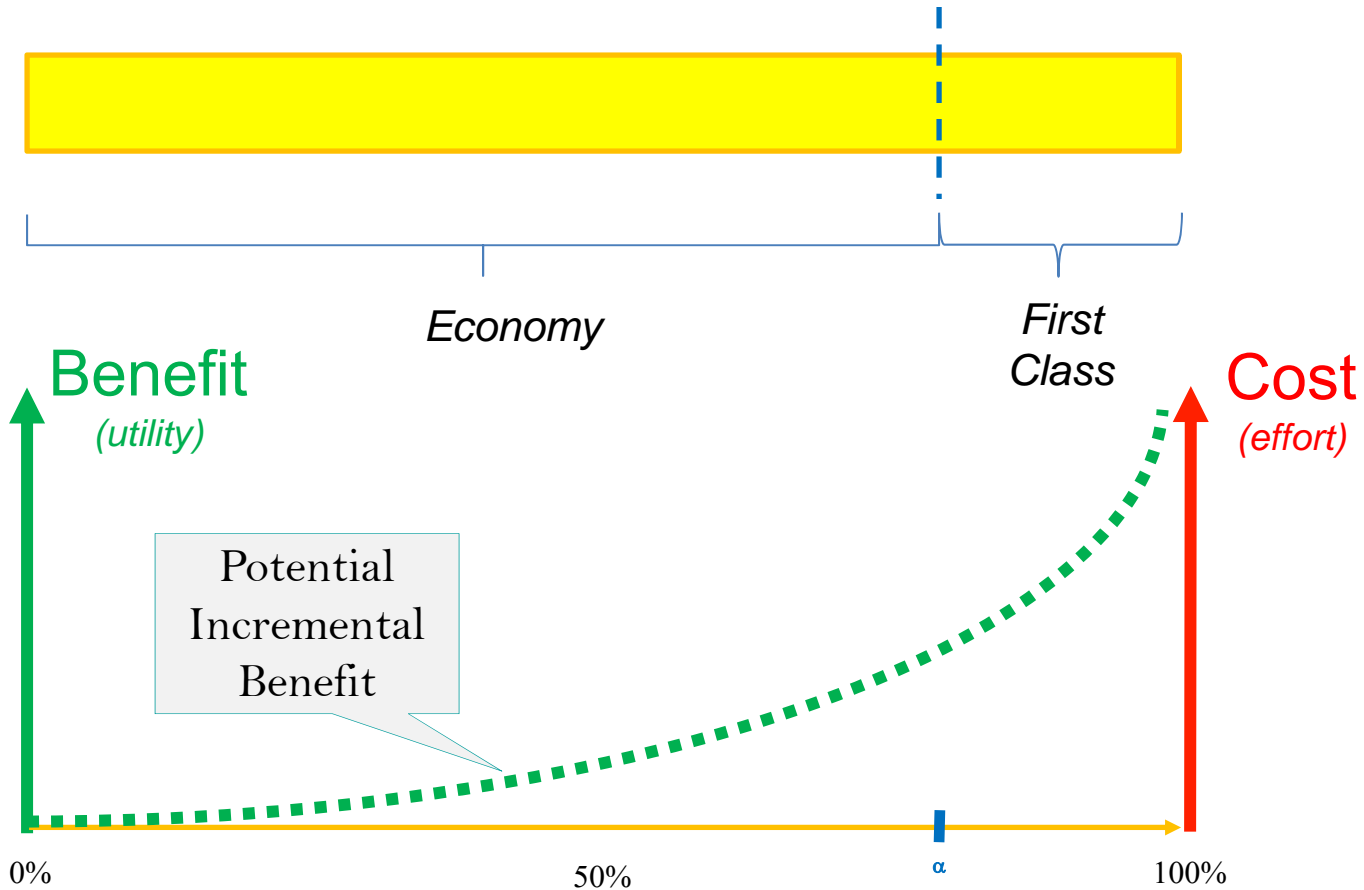
Client Perspective



Client Perspective

# Introduction

Client Perspective

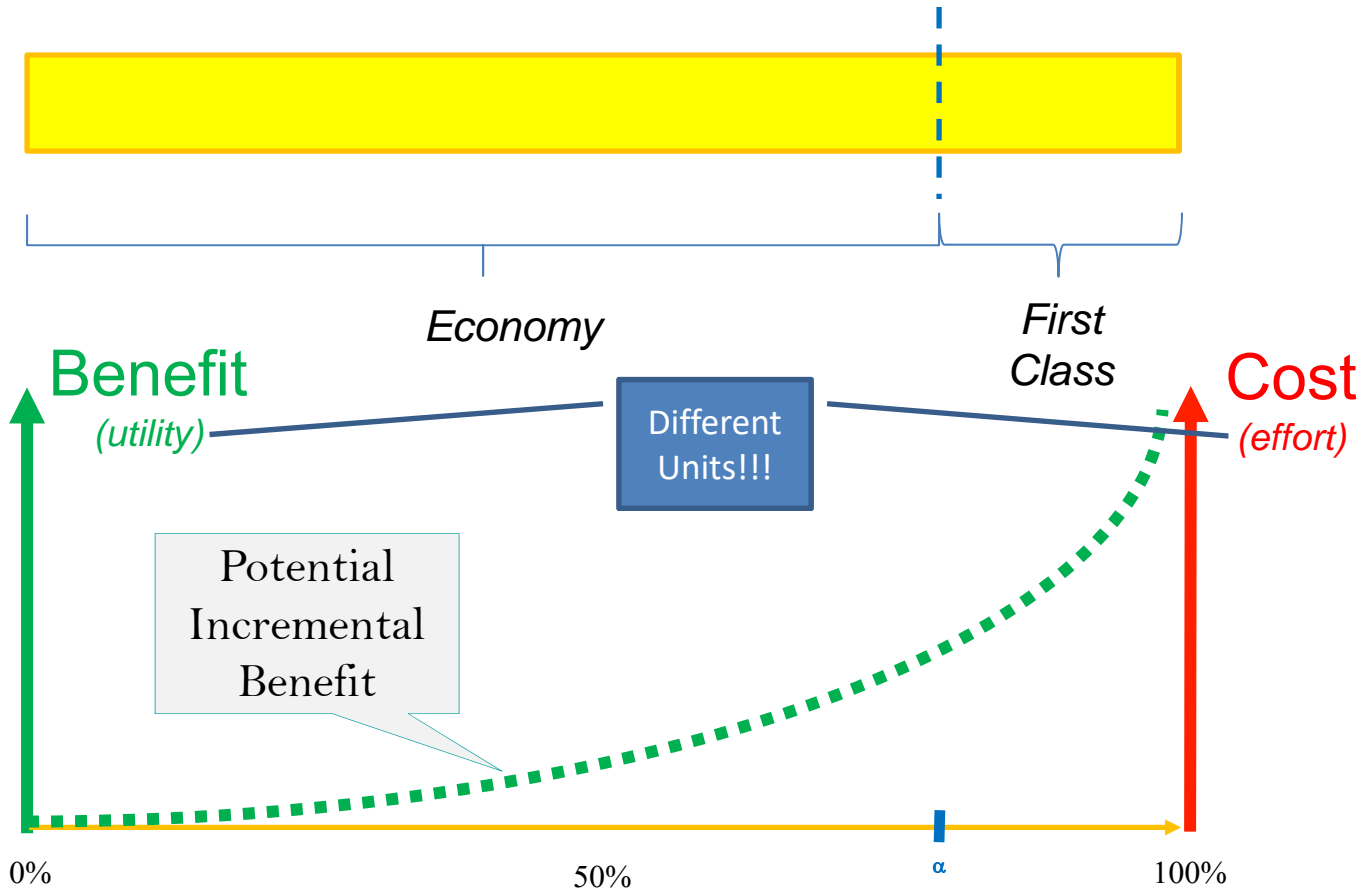




Client Perspective

# Introduction

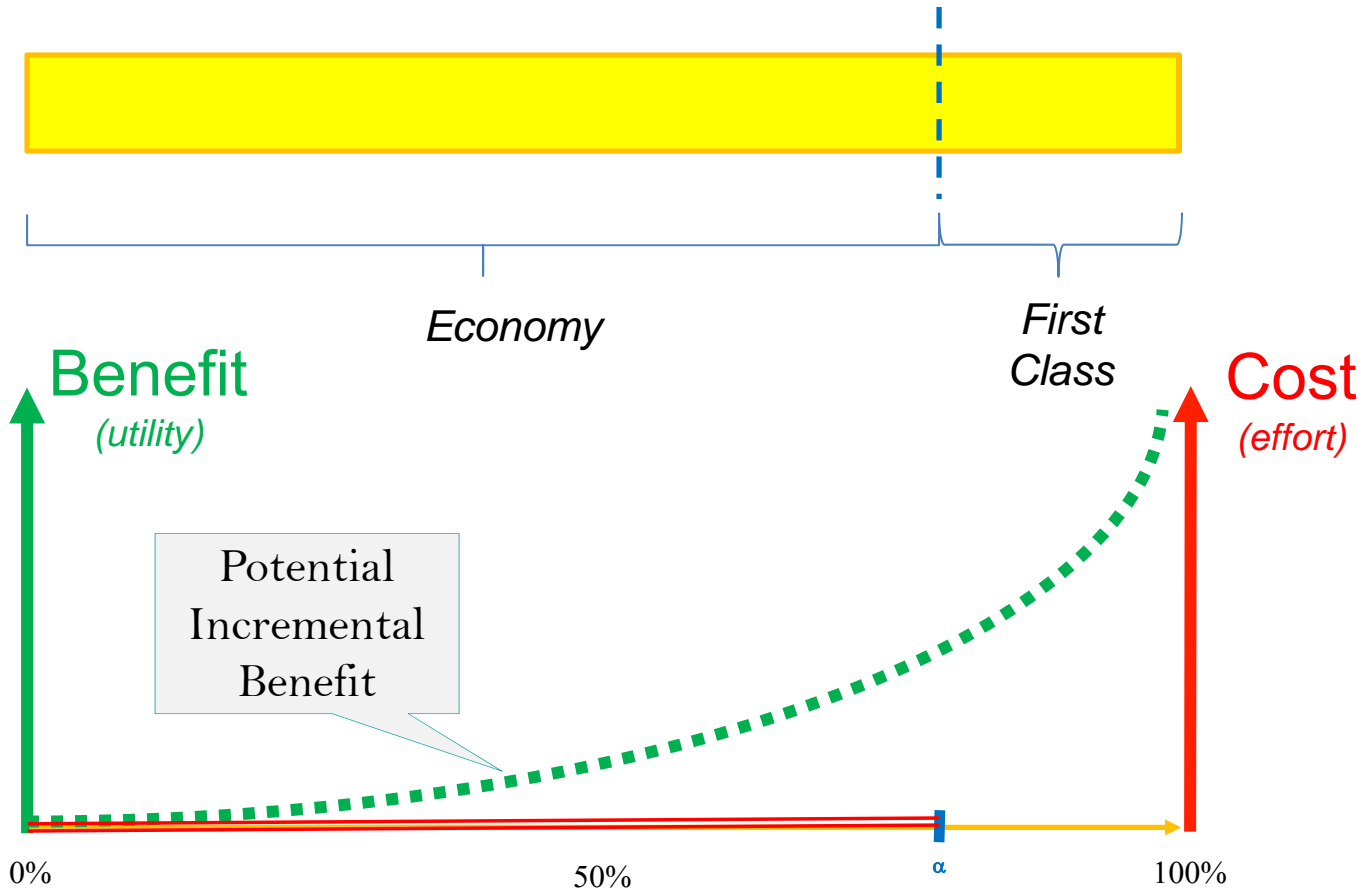
Client Perspective



Client Perspective

# Introduction

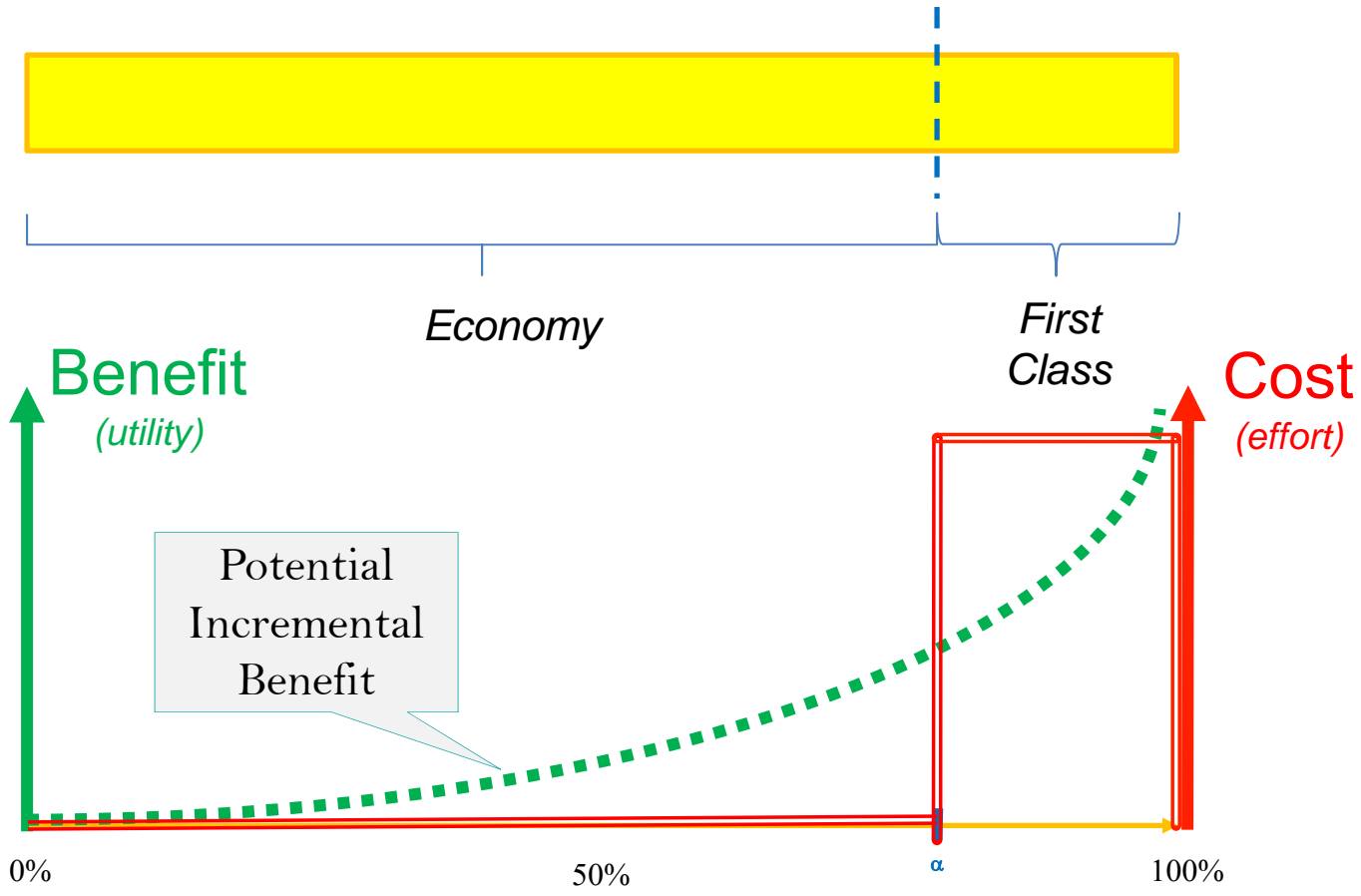
Client Perspective



Client Perspective

# Introduction

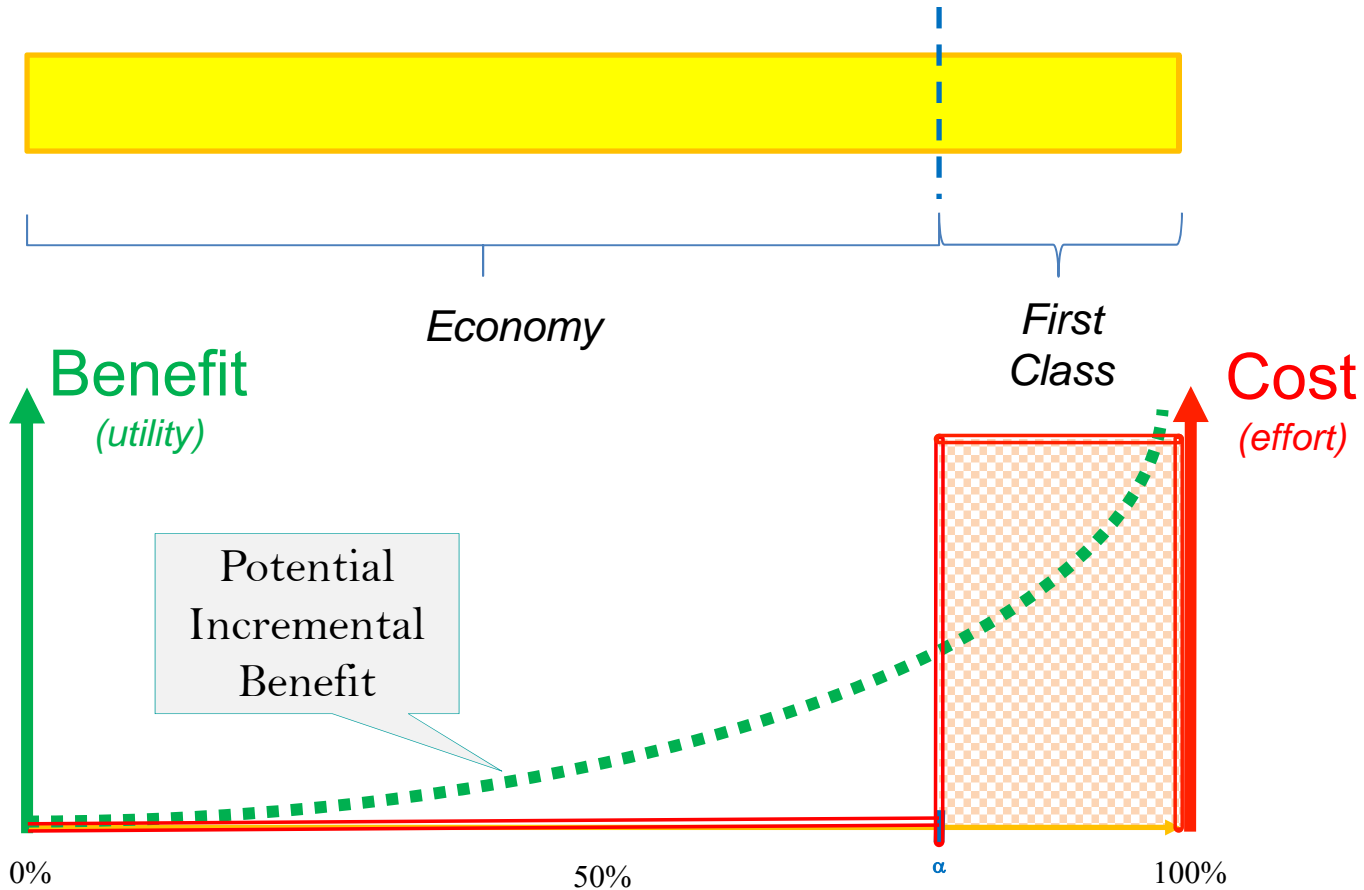
Client Perspective



Client Perspective

# Introduction

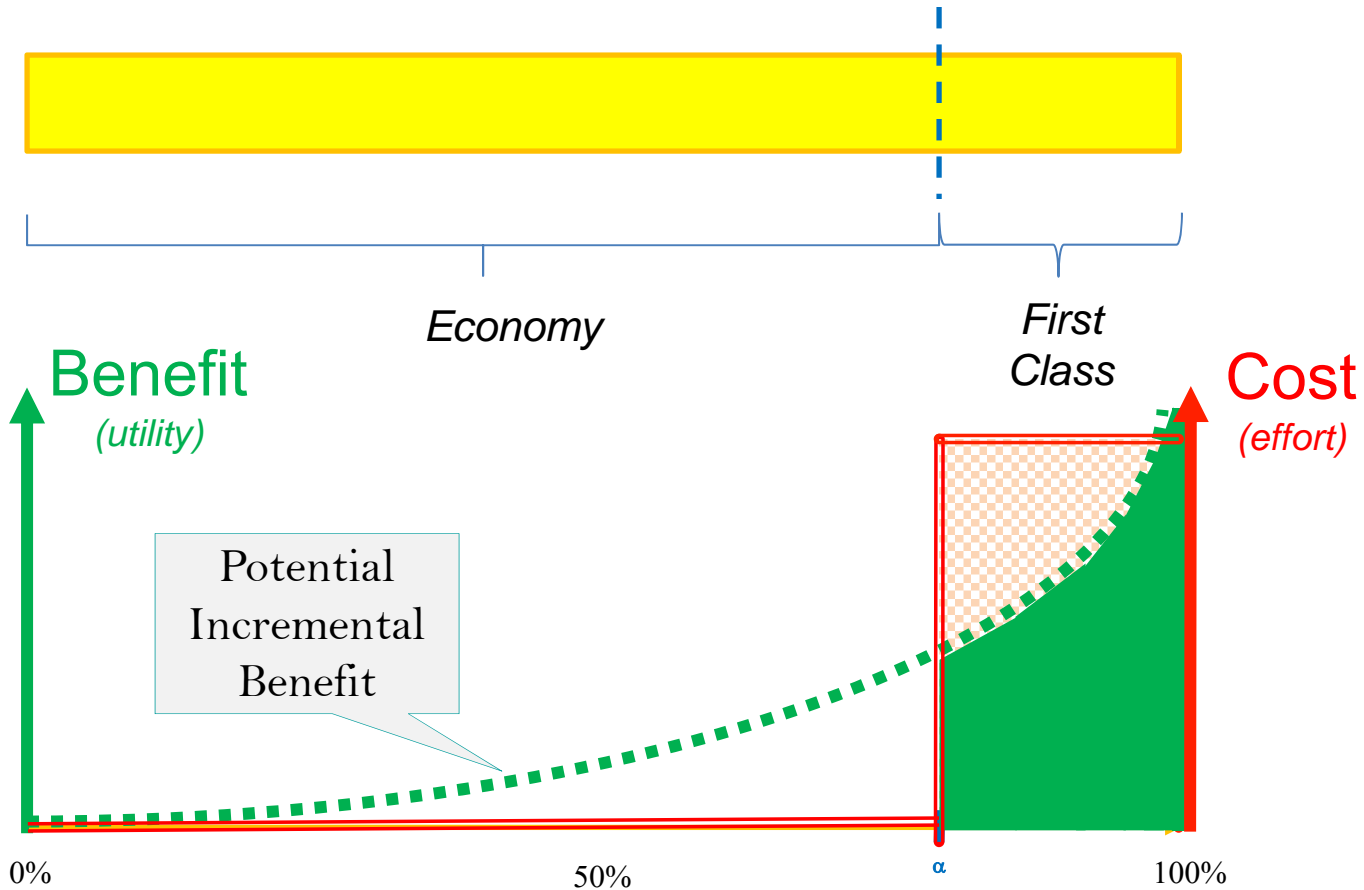
Client Perspective



Client Perspective

# Introduction

Client Perspective



Client  
Perspective

# Introduction

Client  
Perspective

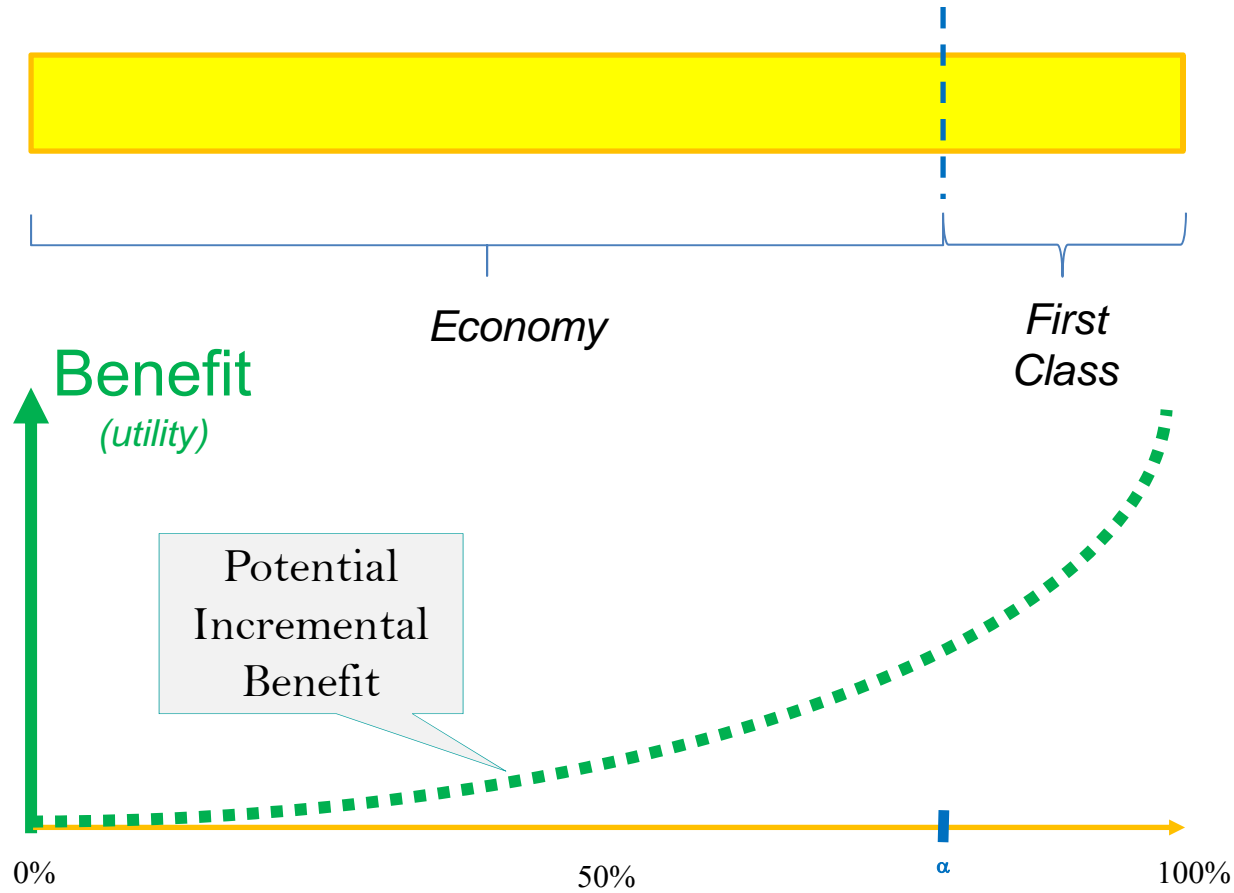
Airline Analogy to Allocator Awareness (AA):

- *First Class*
  - Best possible
- *Economy*
  - Cheapest possible
- ***Business Class and Premium Economy***
  - Almost as good as first class
  - Costs just slightly more than *Economy*

Client Perspective

# Introduction

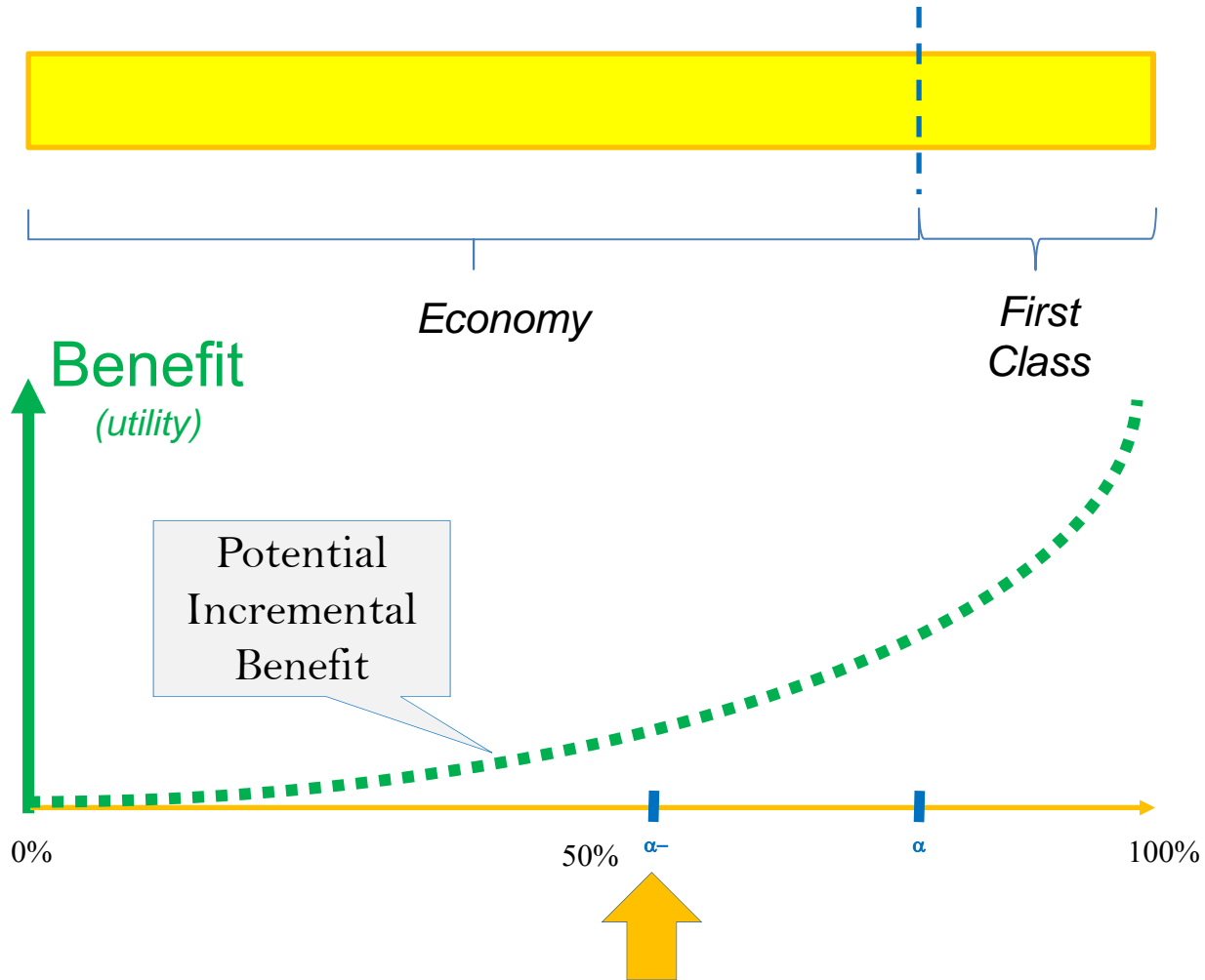
Client Perspective



Client Perspective

# Introduction

Client Perspective

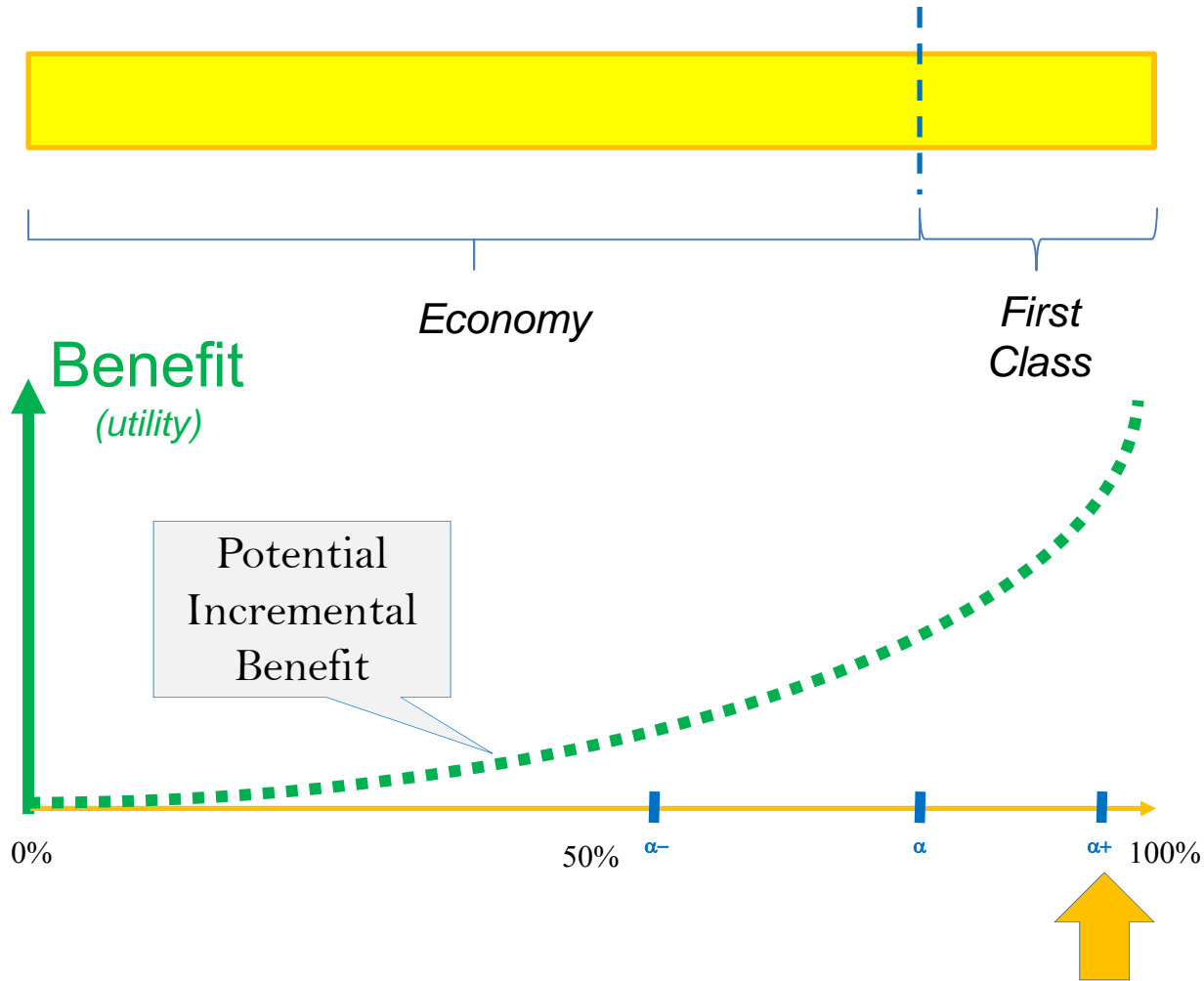




Client Perspective

# Introduction

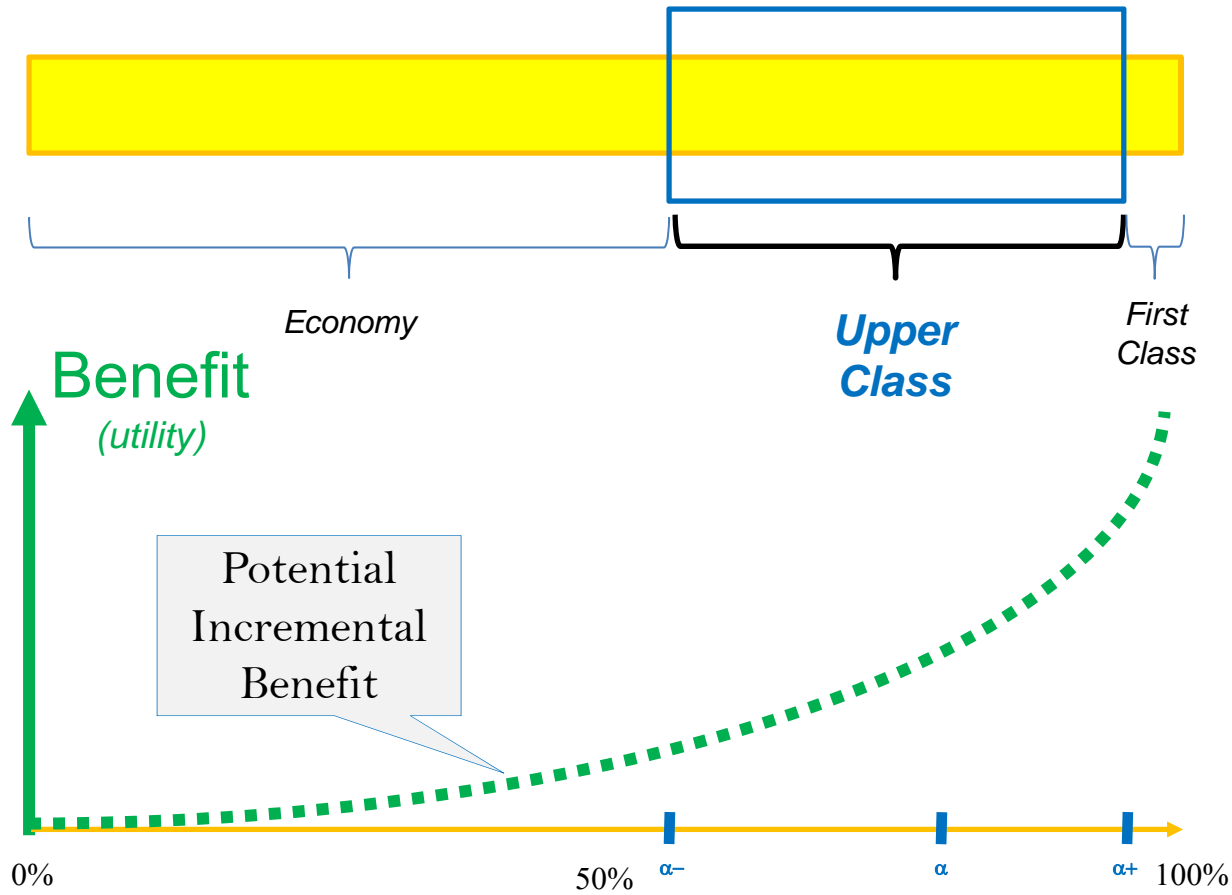
Client Perspective



Client Perspective

# Introduction

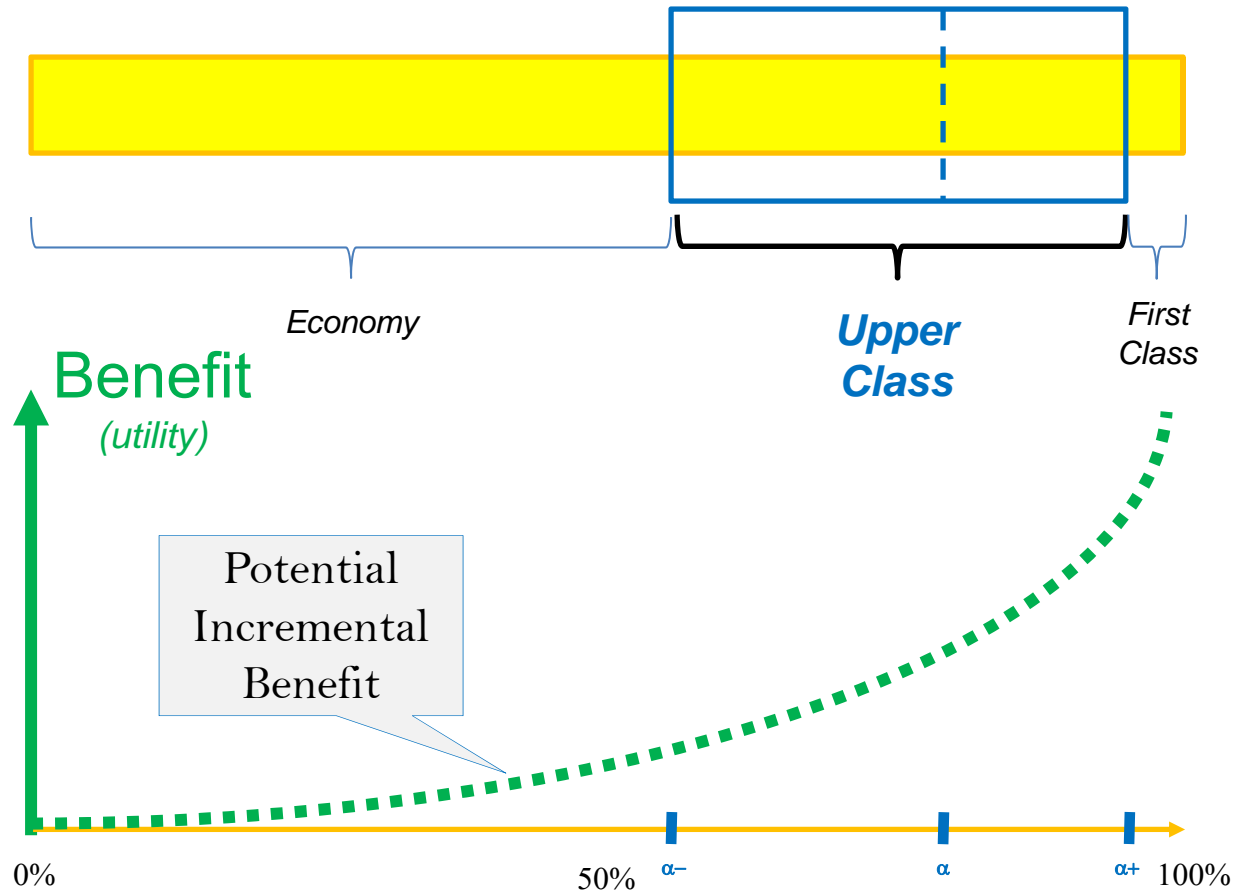
Client Perspective



Client Perspective

# Introduction

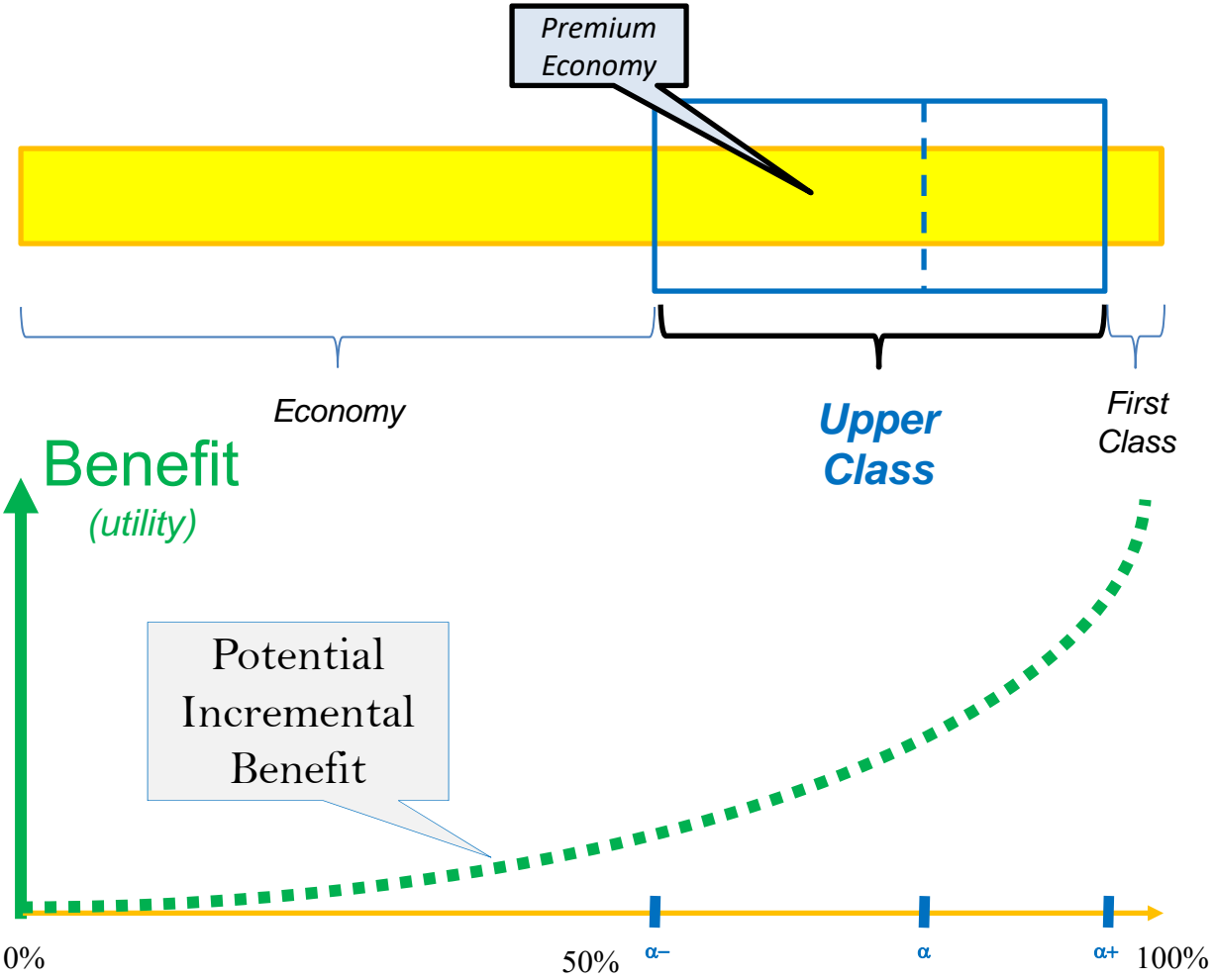
Client Perspective



Client Perspective

# Introduction

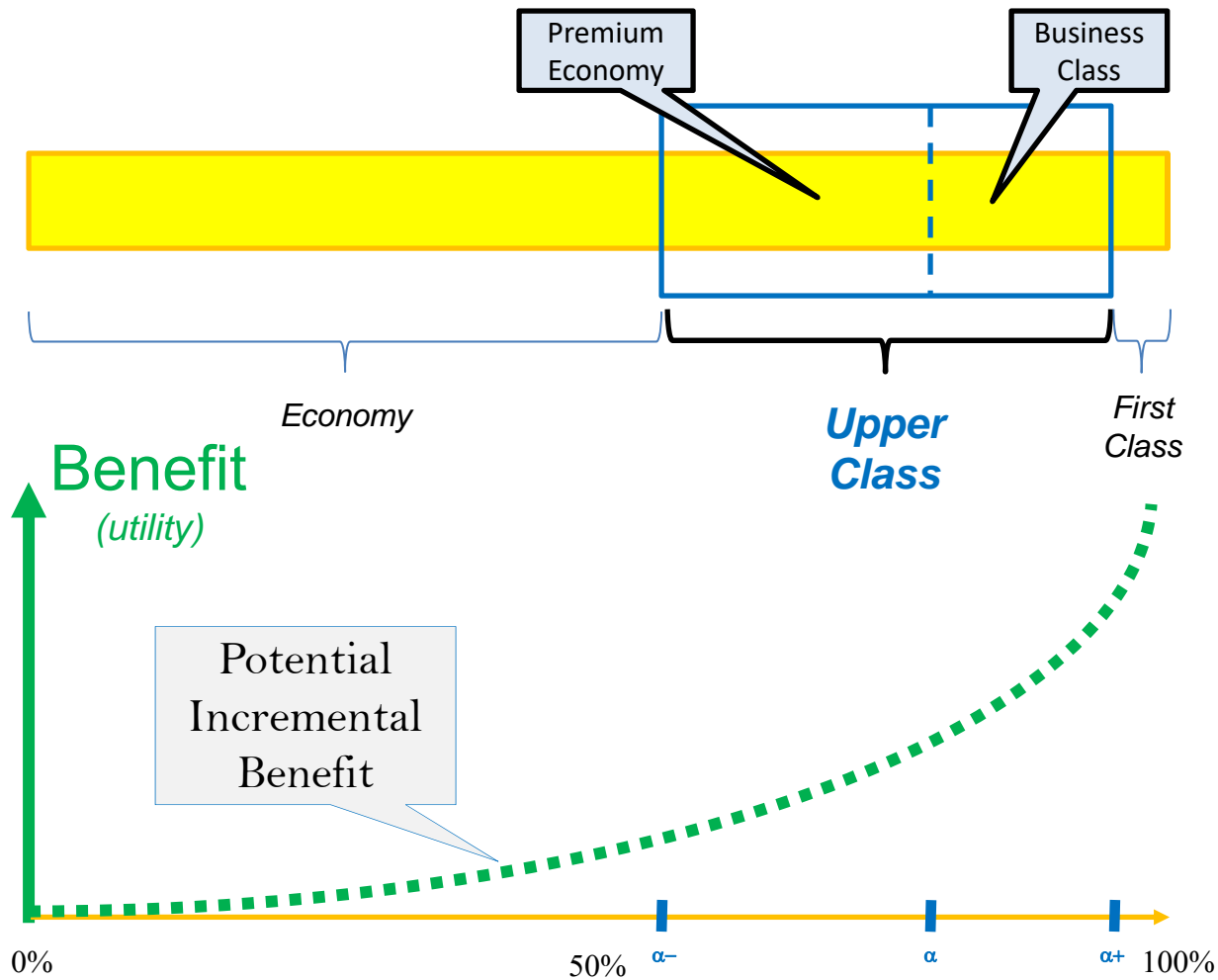
Client Perspective



Client Perspective

# Introduction

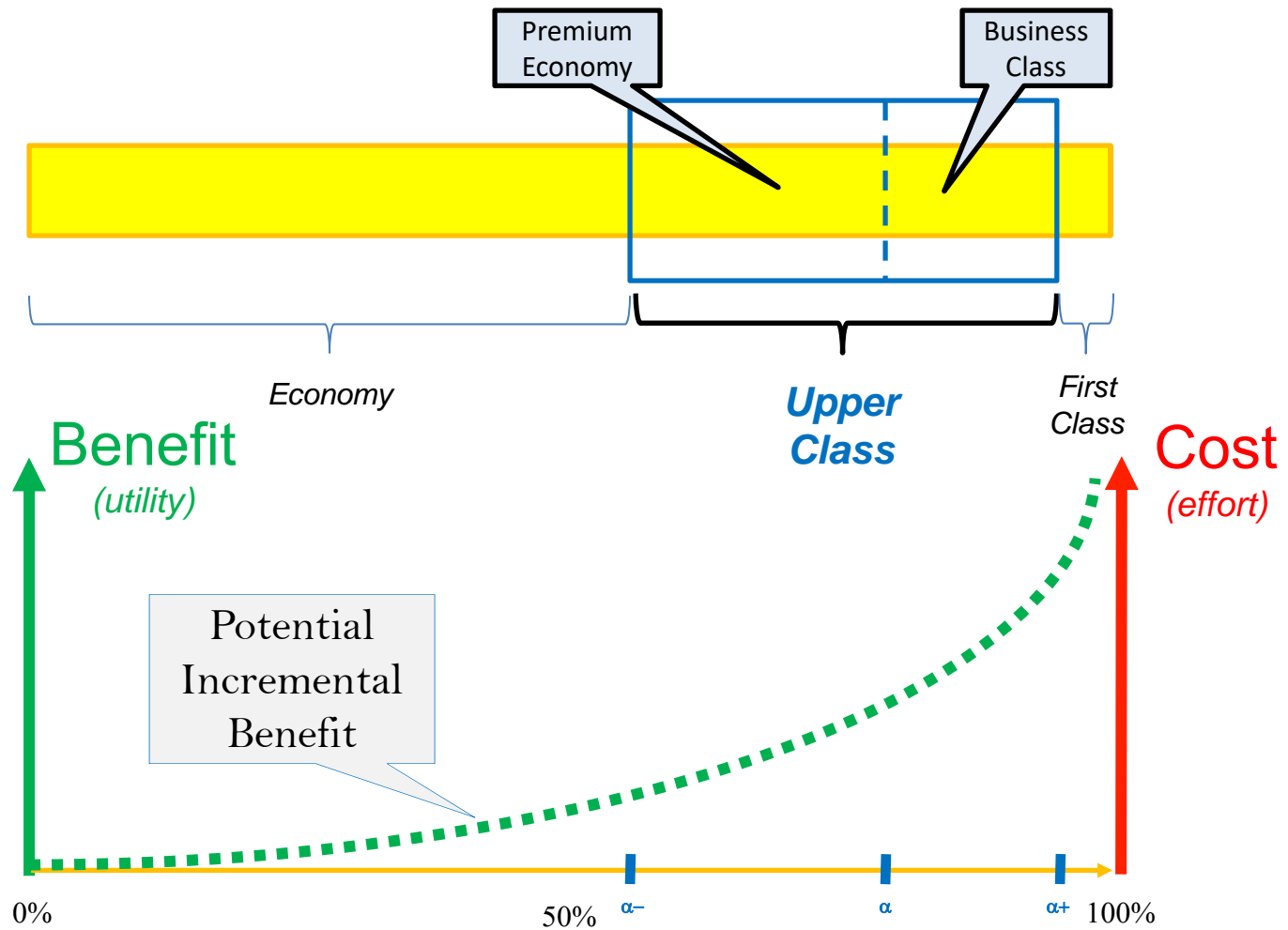
Client Perspective



Client Perspective

# Introduction

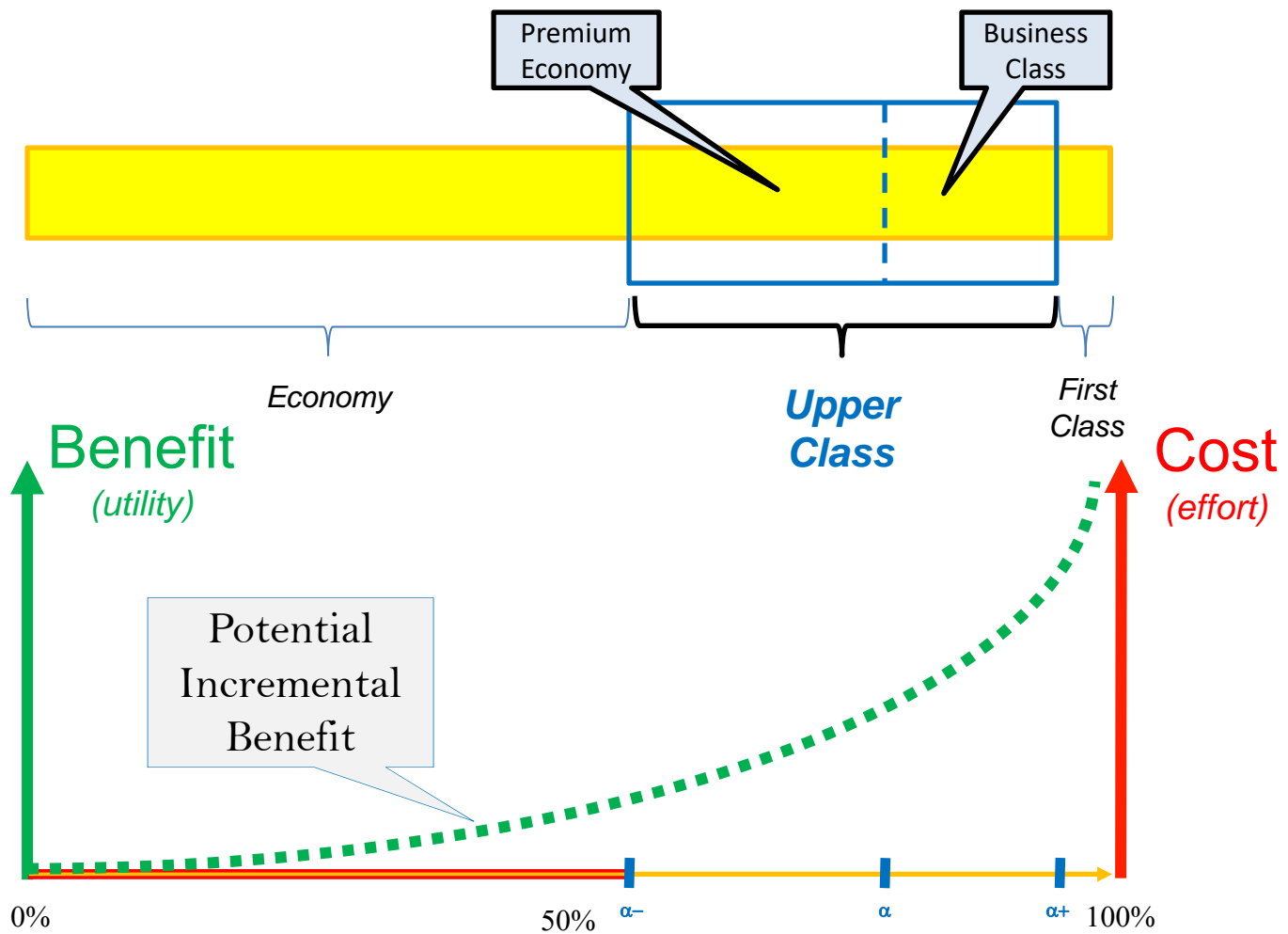
Client Perspective



Client Perspective

# Introduction

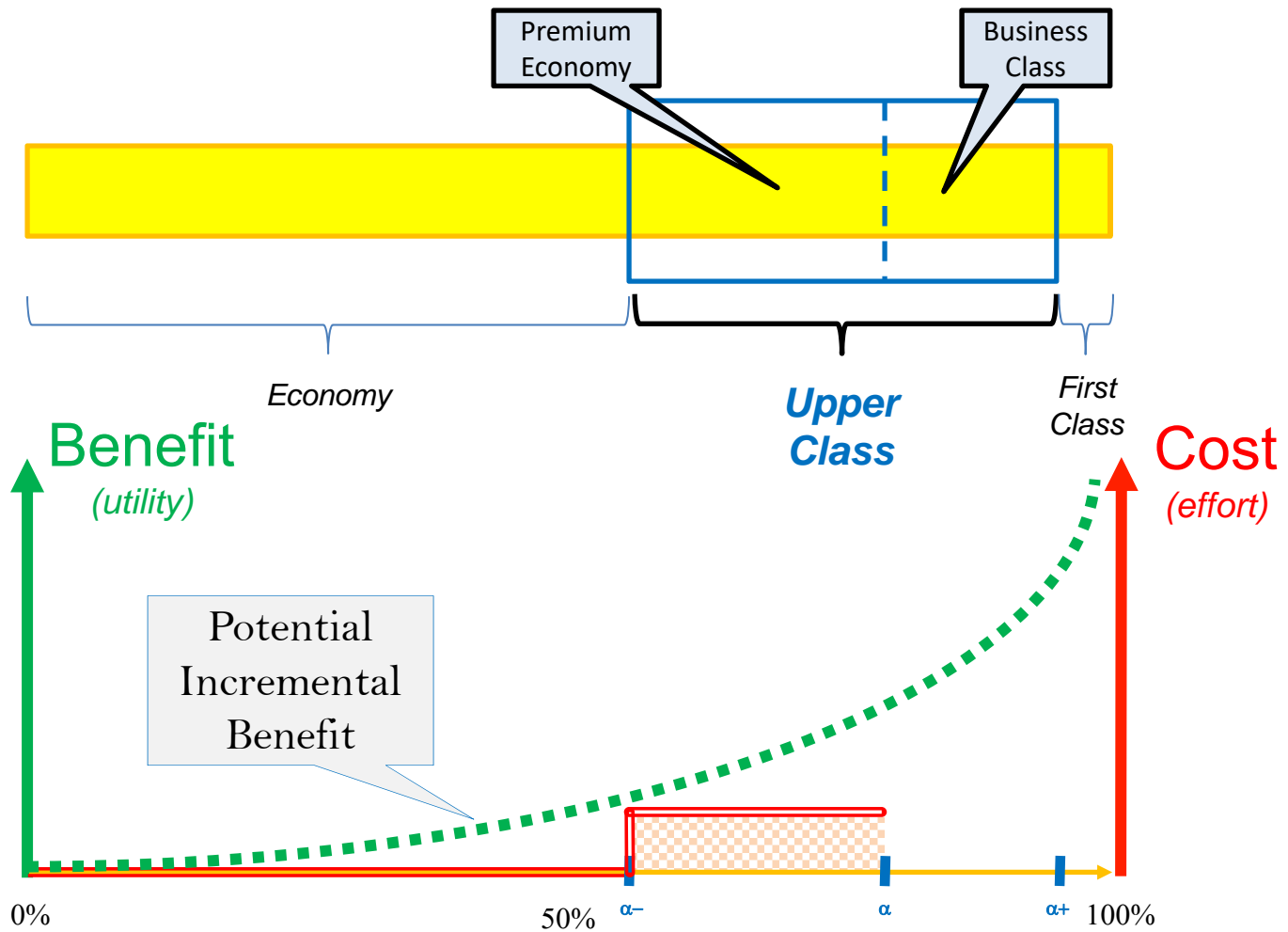
Client Perspective



Client Perspective

# Introduction

Client Perspective

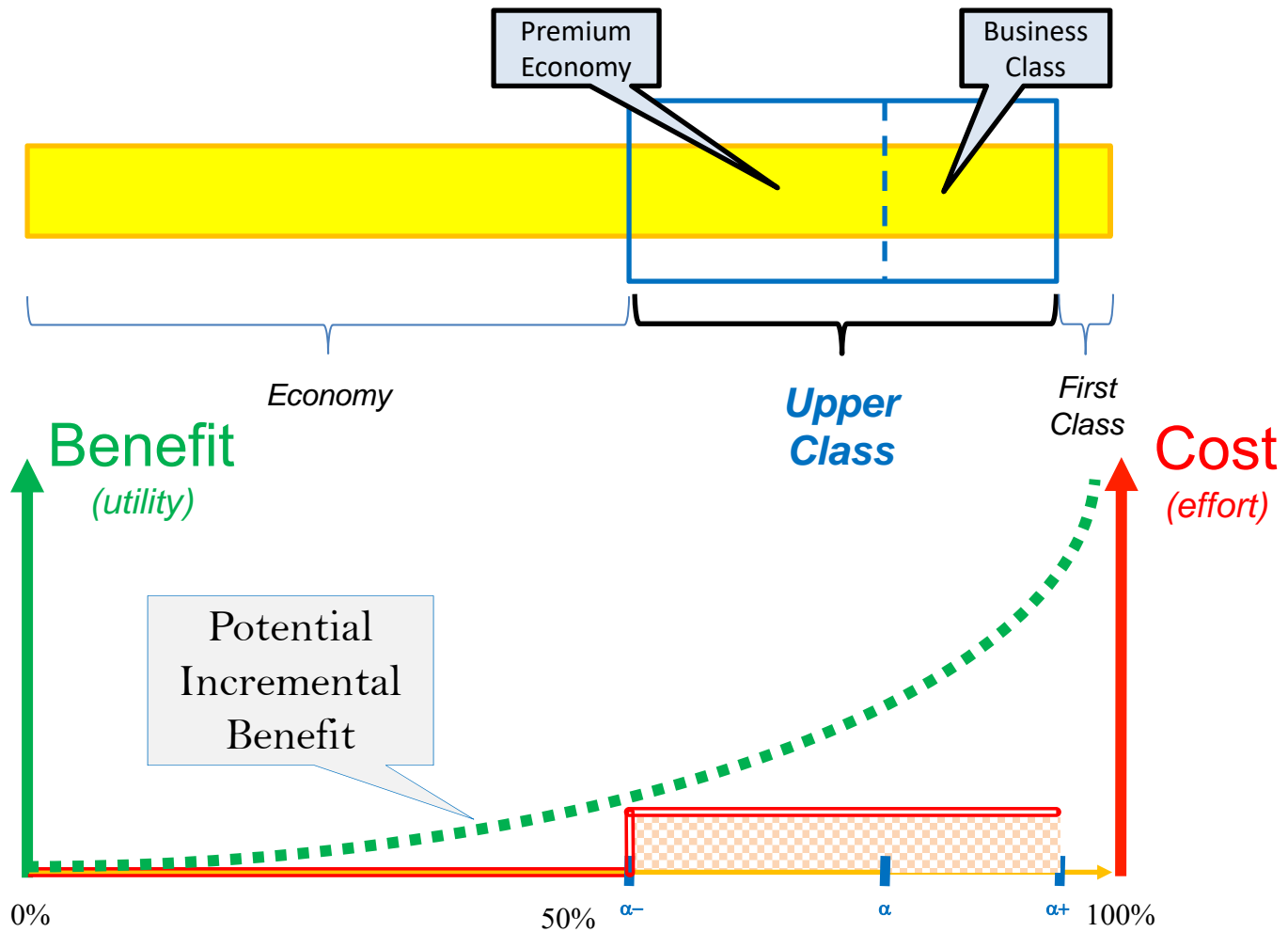




Client Perspective

# Introduction

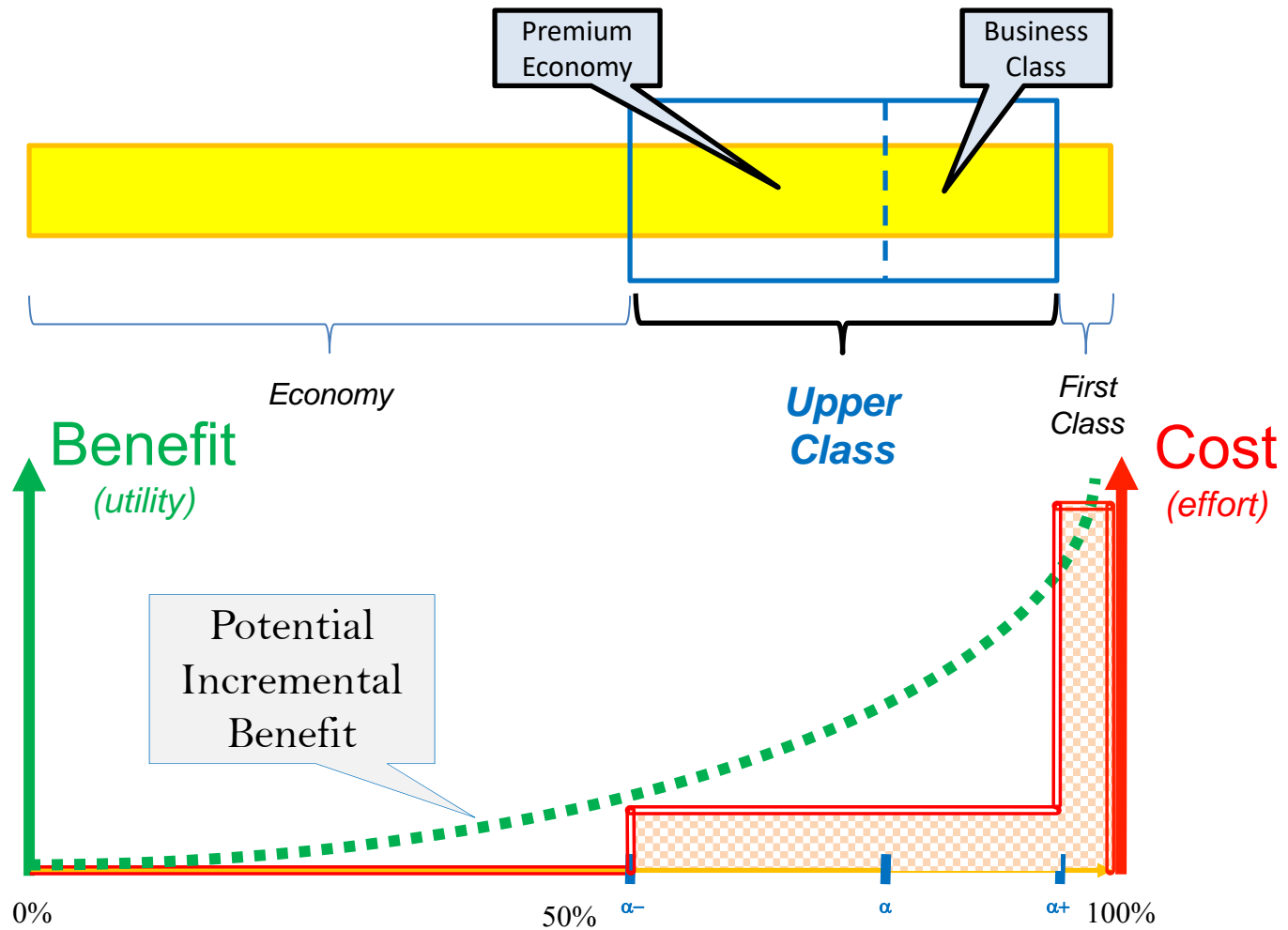
Client Perspective



Client Perspective

# Introduction

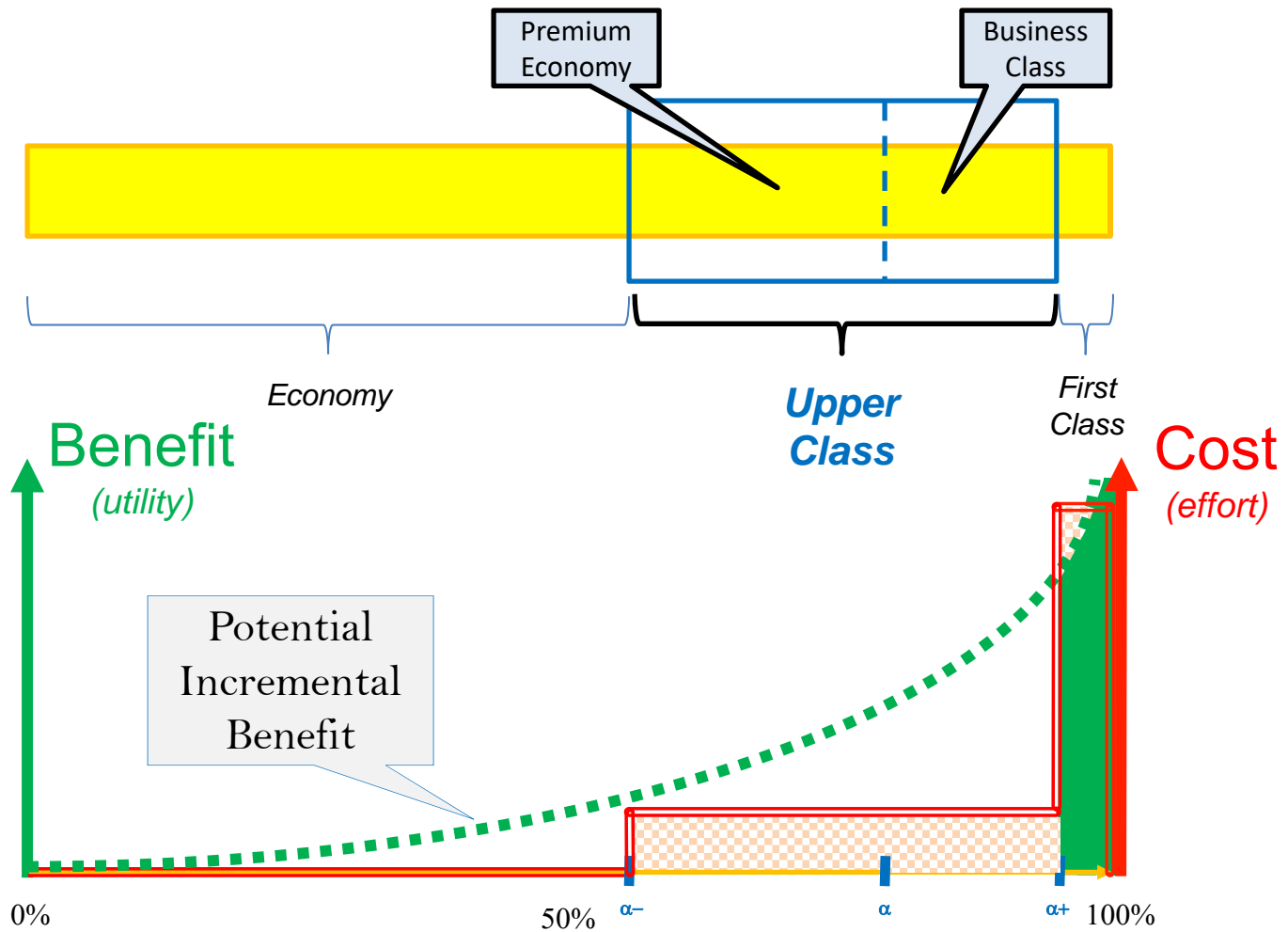
Client Perspective



Client Perspective

# Introduction

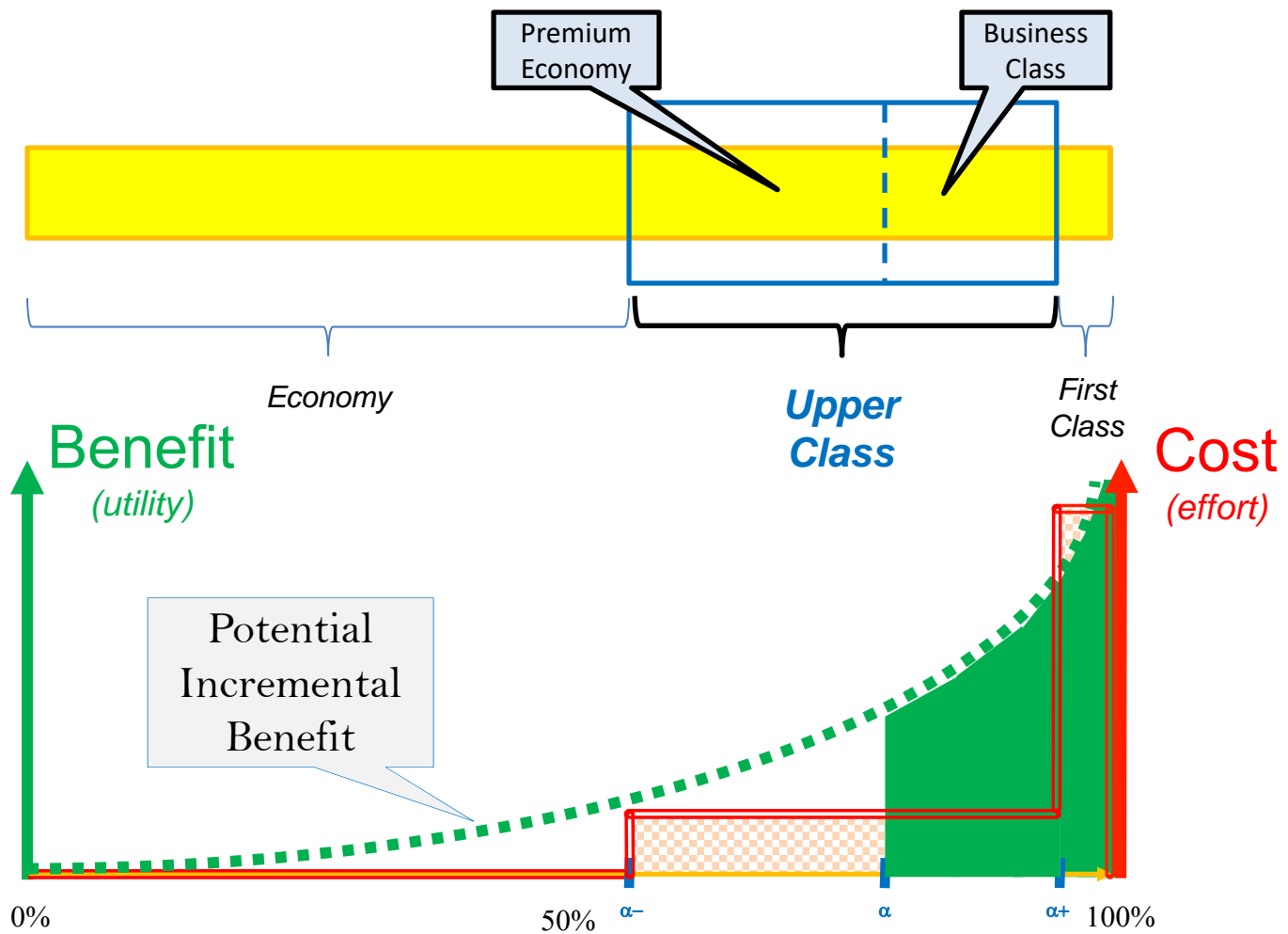
Client Perspective



Client Perspective

# Introduction

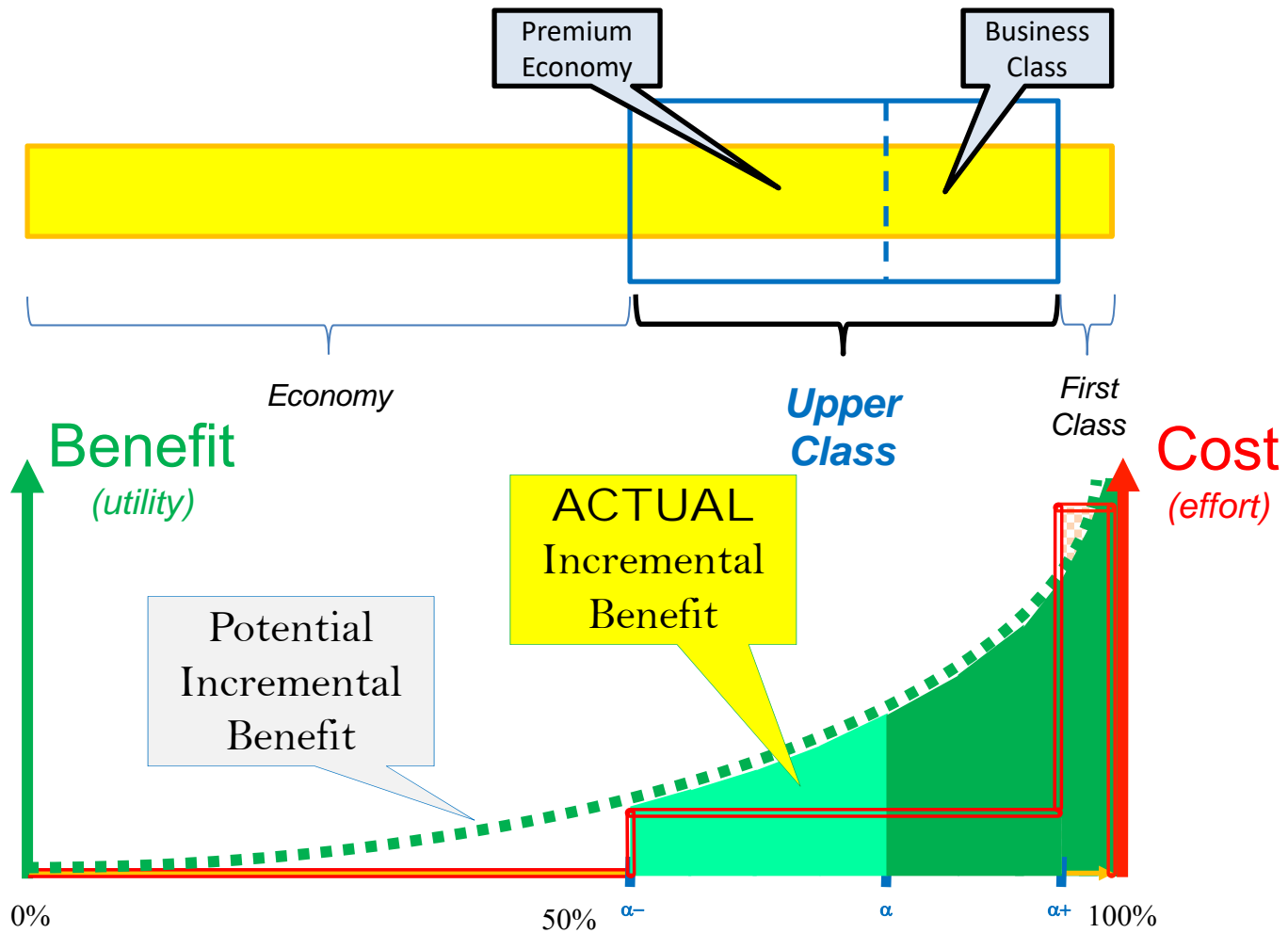
Client Perspective



Client Perspective

# Introduction

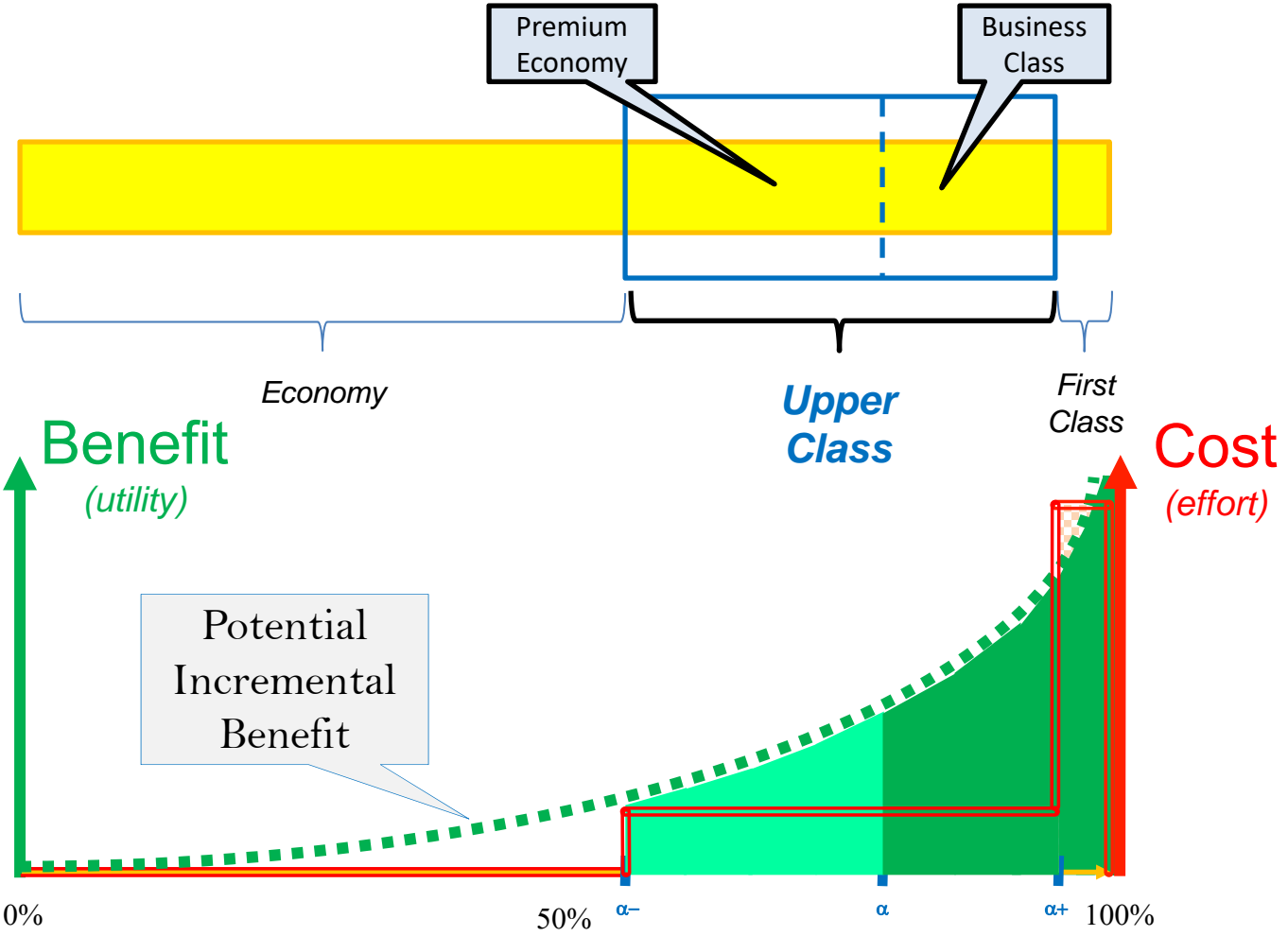
Client Perspective



Client Perspective

# Introduction

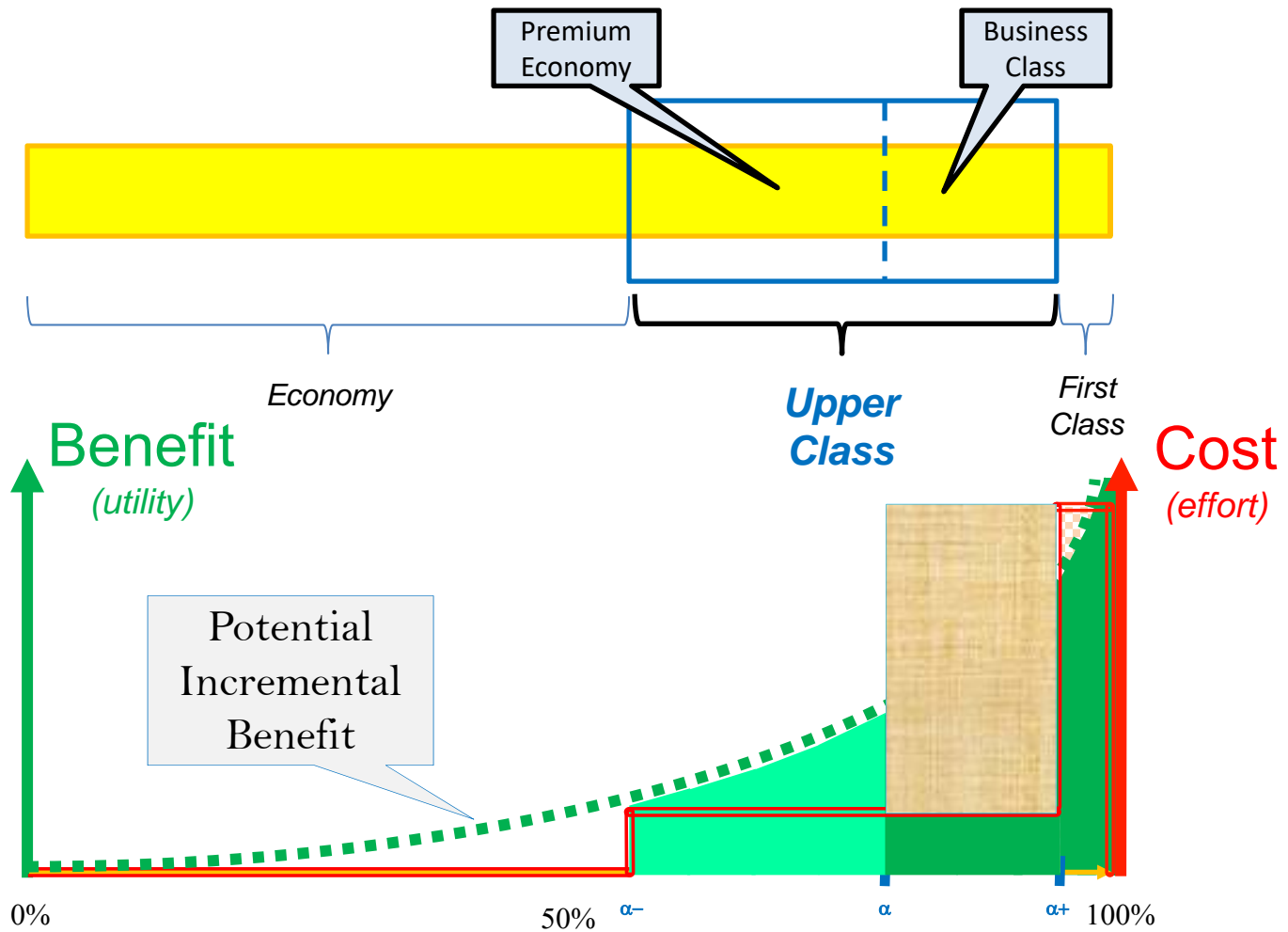
Client Perspective



Client Perspective

# Introduction

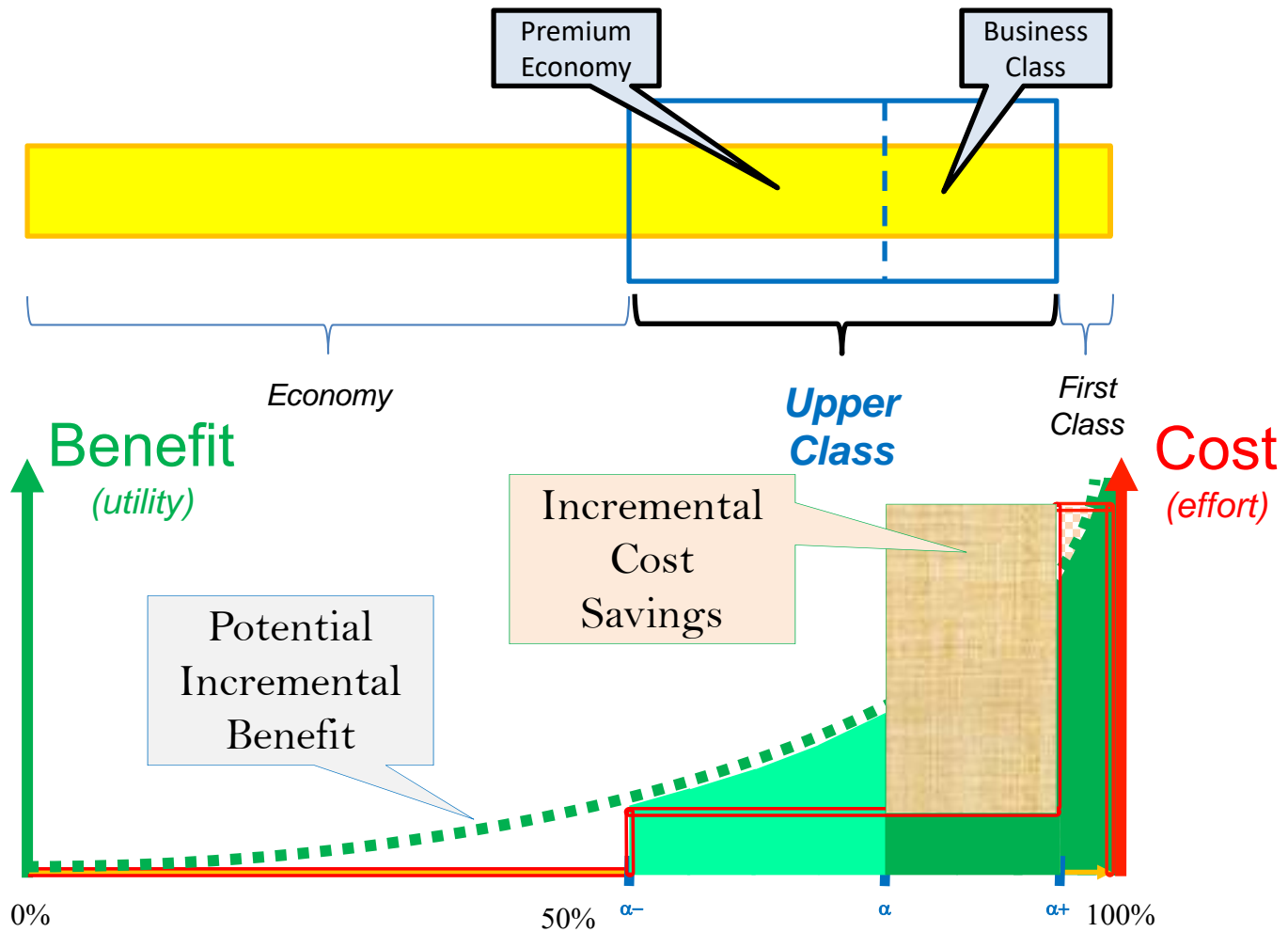
Client Perspective



Client Perspective

# Introduction

Client Perspective

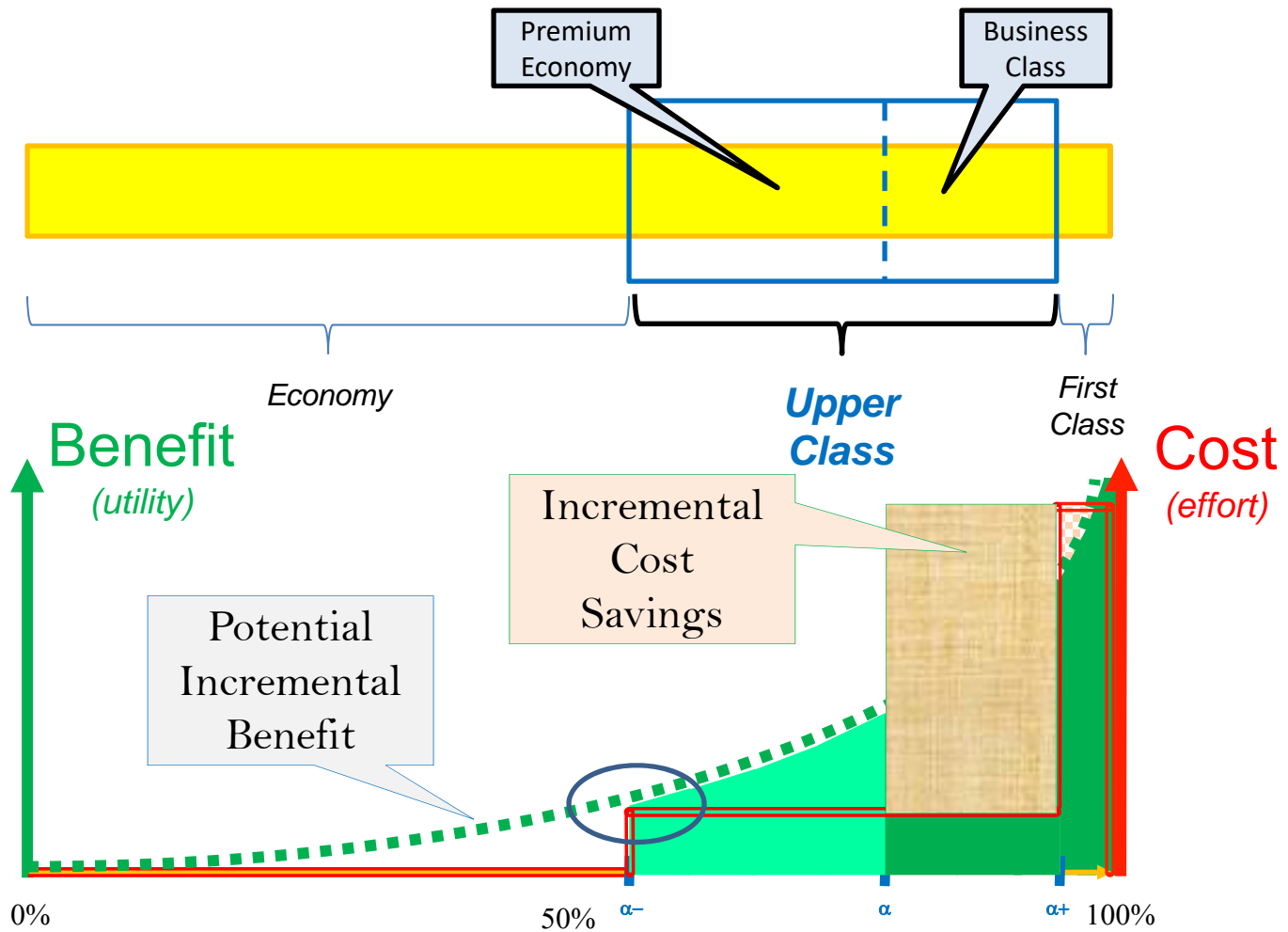




Client Perspective

# Introduction

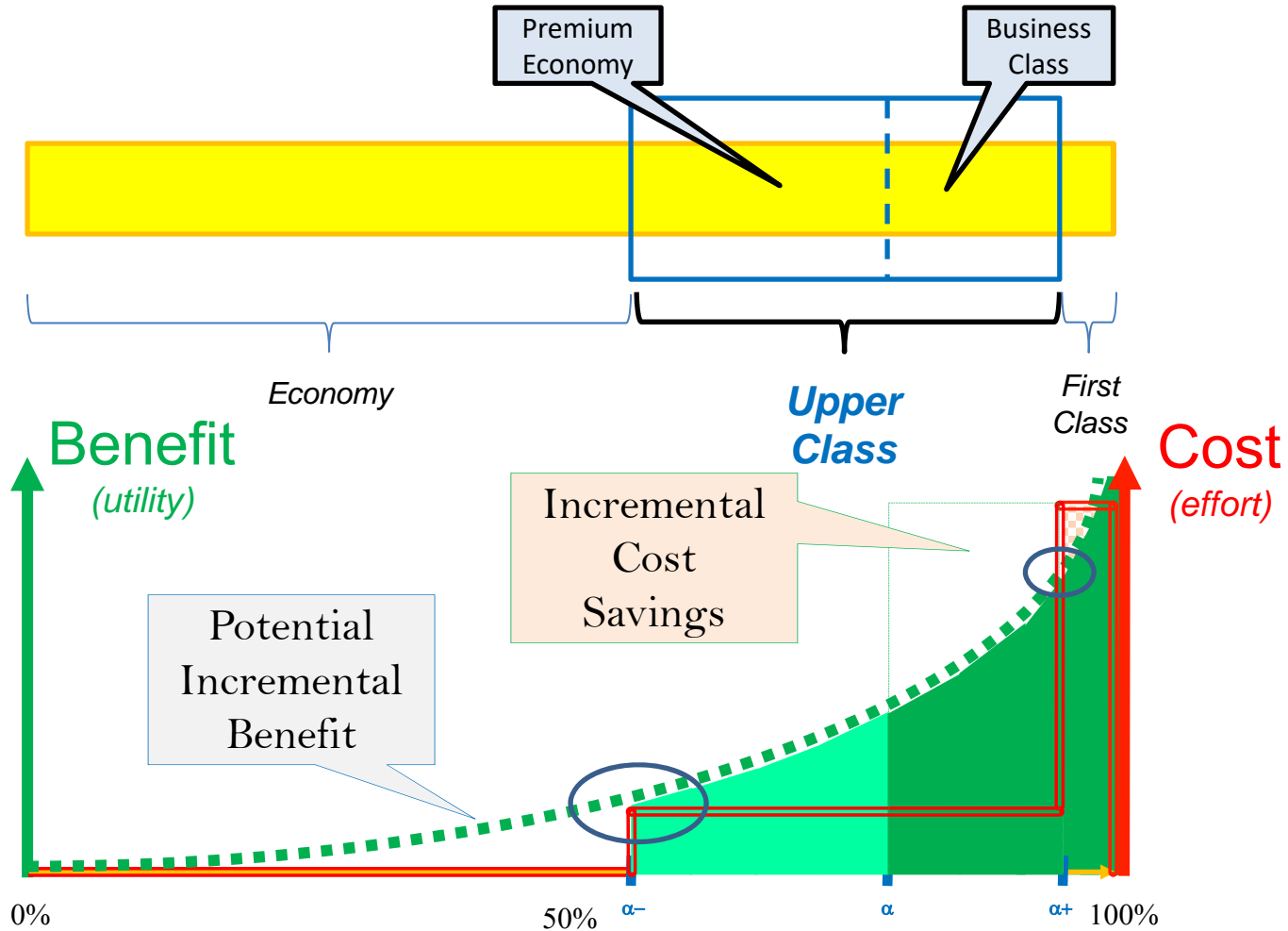
Client Perspective



Client Perspective

# Introduction

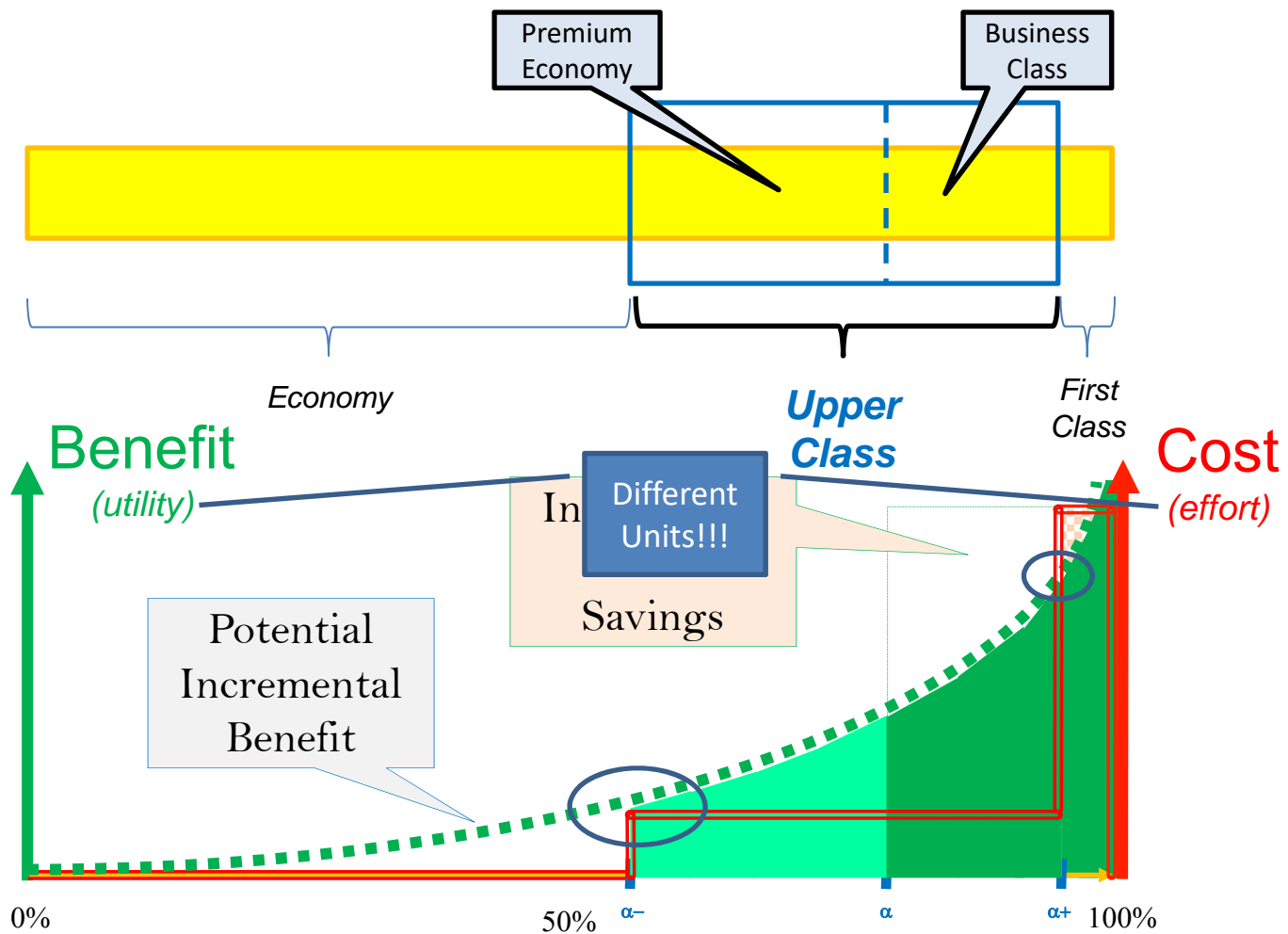
Client Perspective



Client Perspective

# Introduction

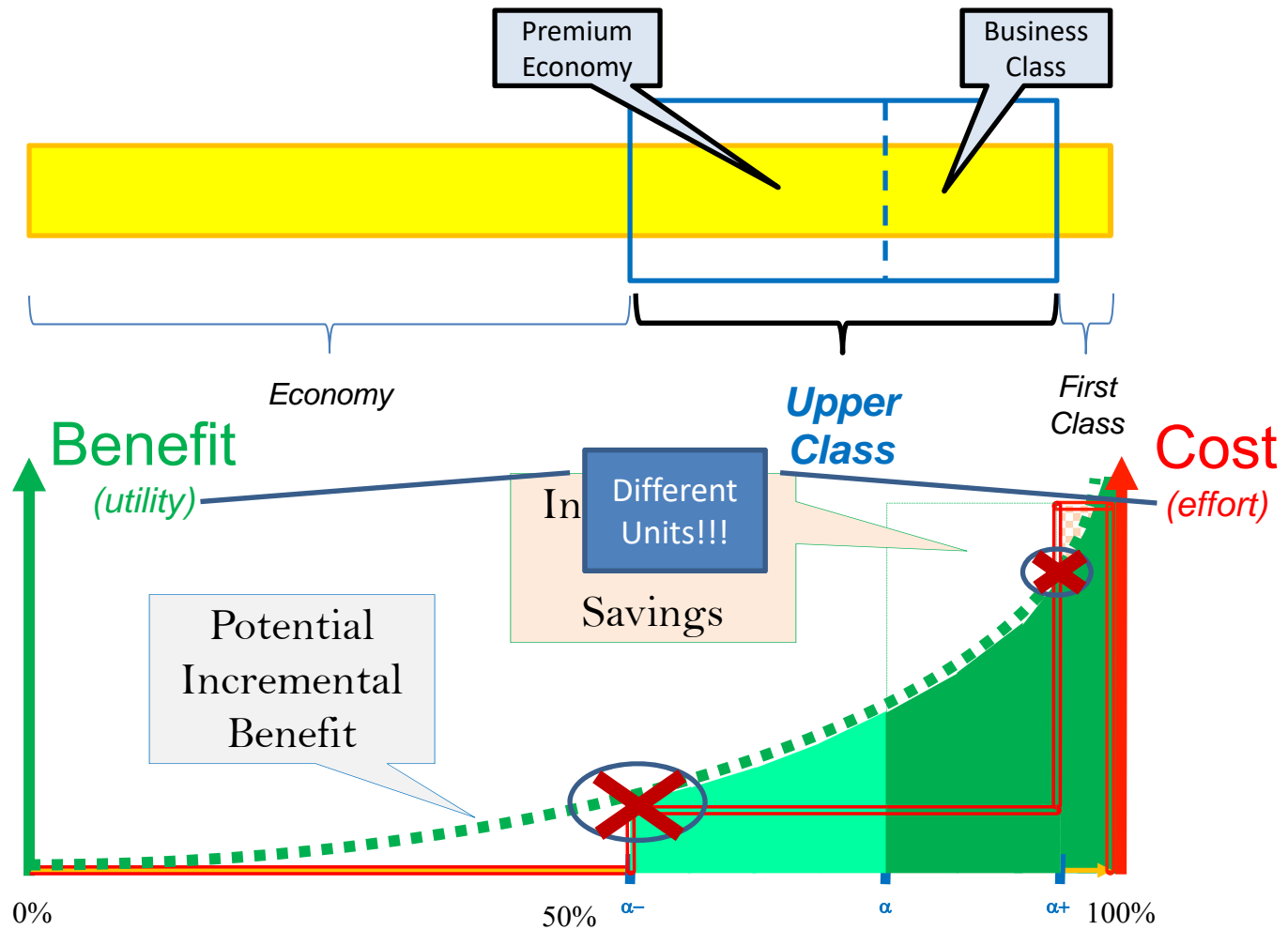
Client Perspective



Client Perspective

# Introduction

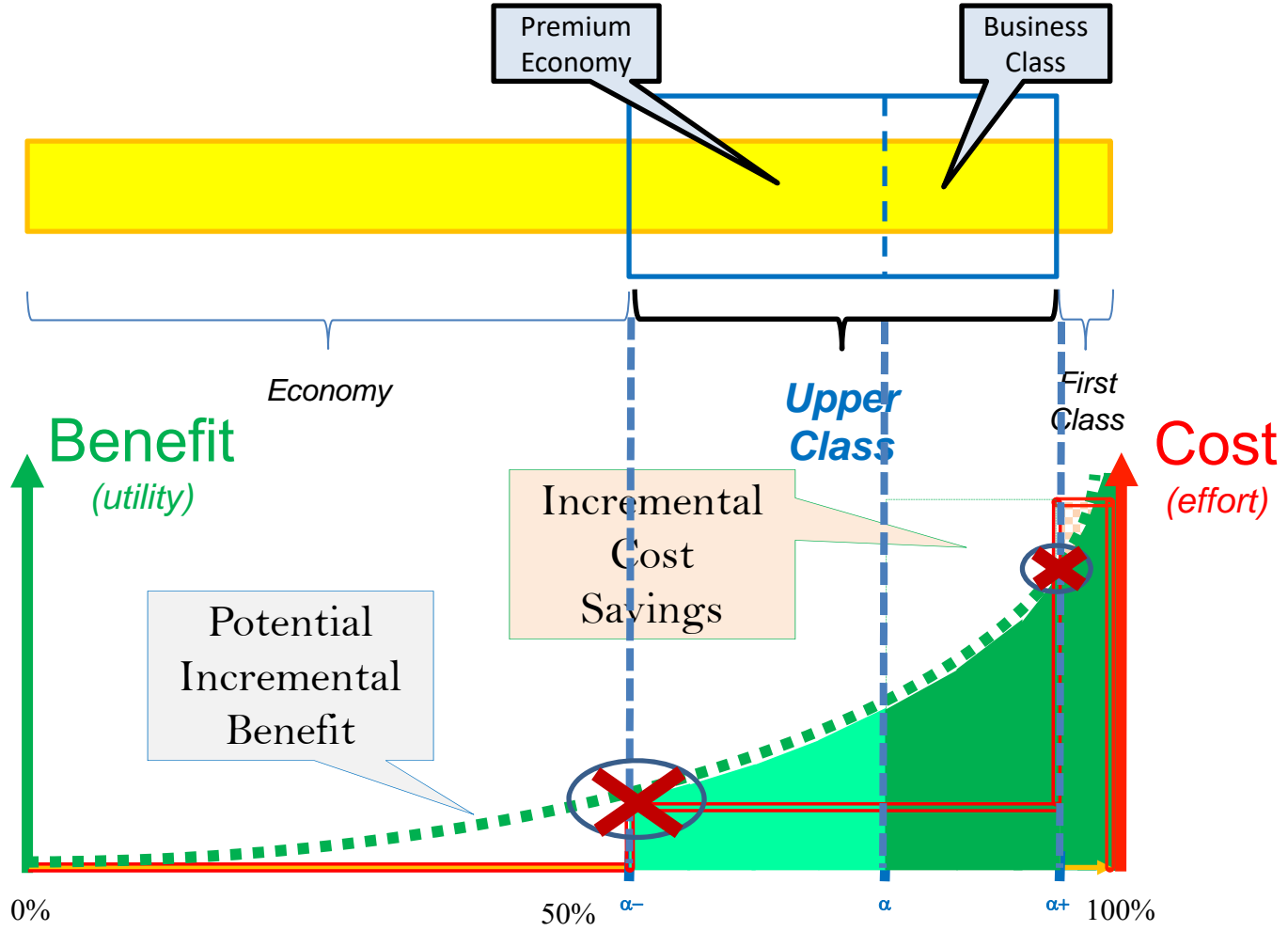
Client Perspective



Client Perspective

# Introduction

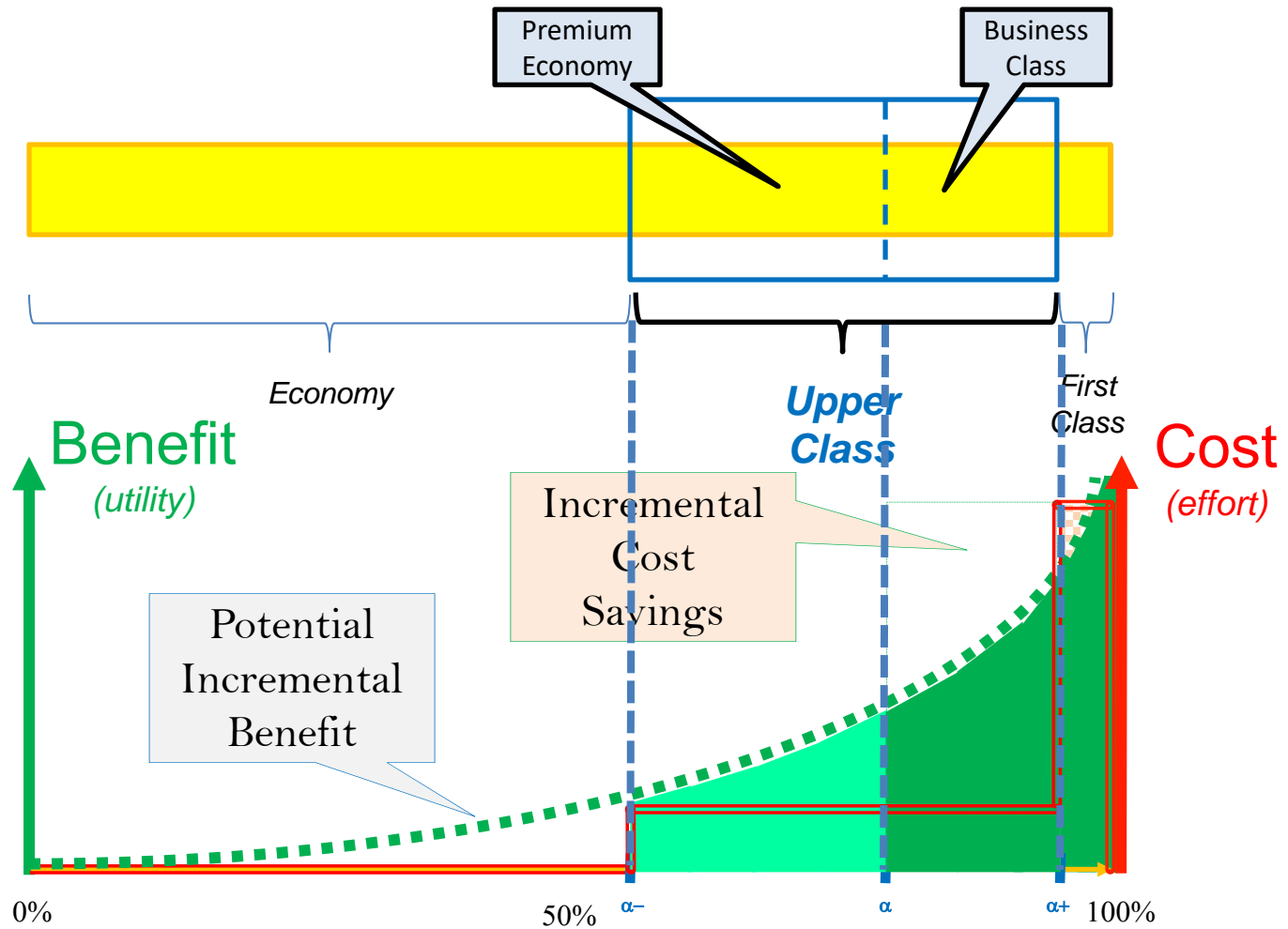
Client Perspective



Client Perspective

# Introduction

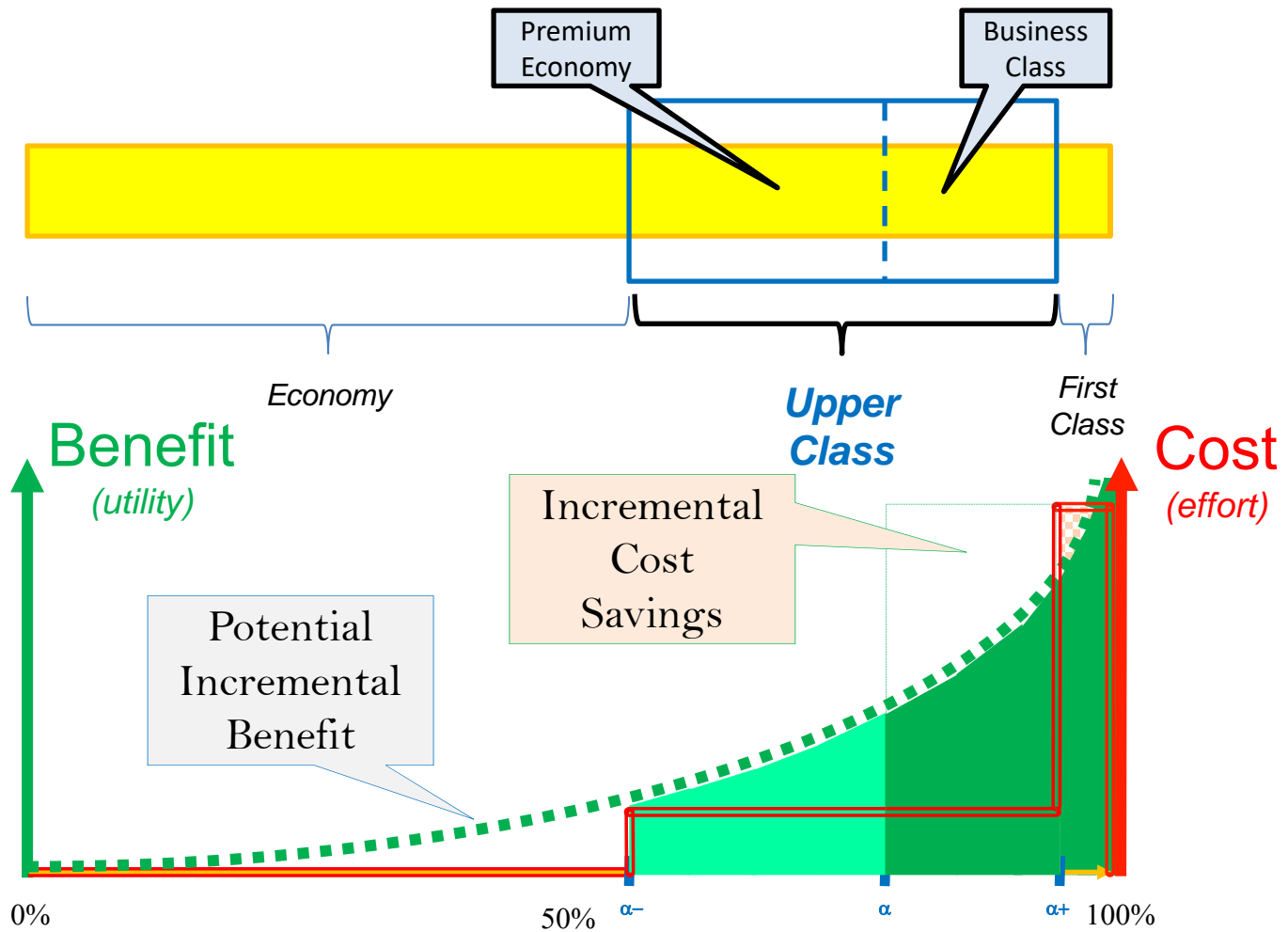
Client Perspective



Client Perspective

# Introduction

Client Perspective



# Introduction

Which airline do you think I fly most often?

- Delta Airlines (DA)
- Lufthansa (L)
- United Airlines (UA)
- *American Airlines (AA)*
- British Airways (BA)



# Introduction

Which airline do you think I fly most often?

- Delta Airlines (DA)
- Lufthansa (L)
- United Airlines (UA)
- *American Airlines (AA)*\*
- British Airways (BA)

\*And I use their *American Advantage (AA)* credit card!

# Introduction

Which airline do you think I fly most often?

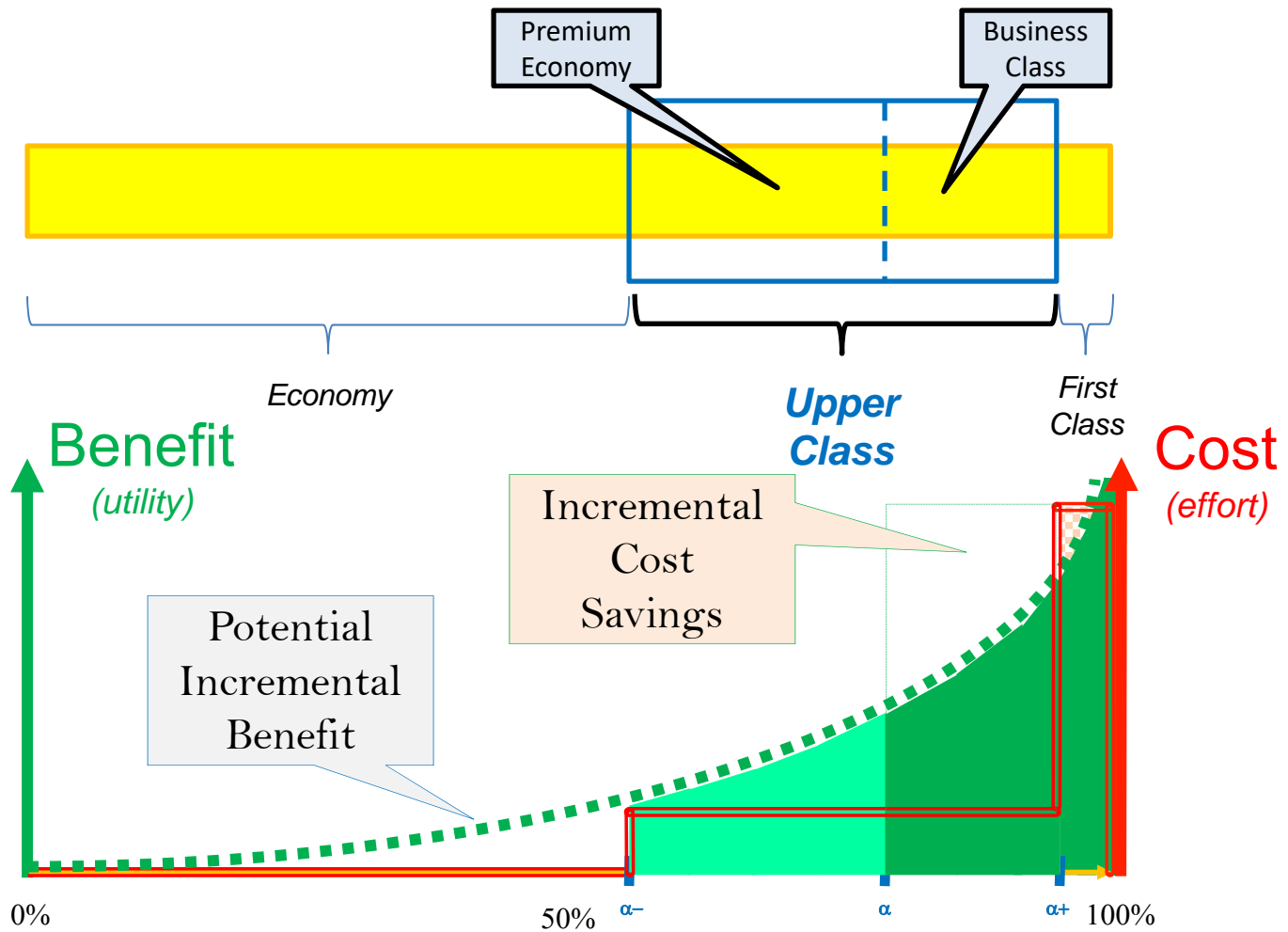
- Delta Airlines (DA)
- Lufthansa (L)
- United Airlines (UA)
- *American Airlines (AA)*\*
- British Airways (BA)

\*And I use their *American Advantage (AA)* credit card!  
(Consider this talk an AA meeting)

Client Perspective

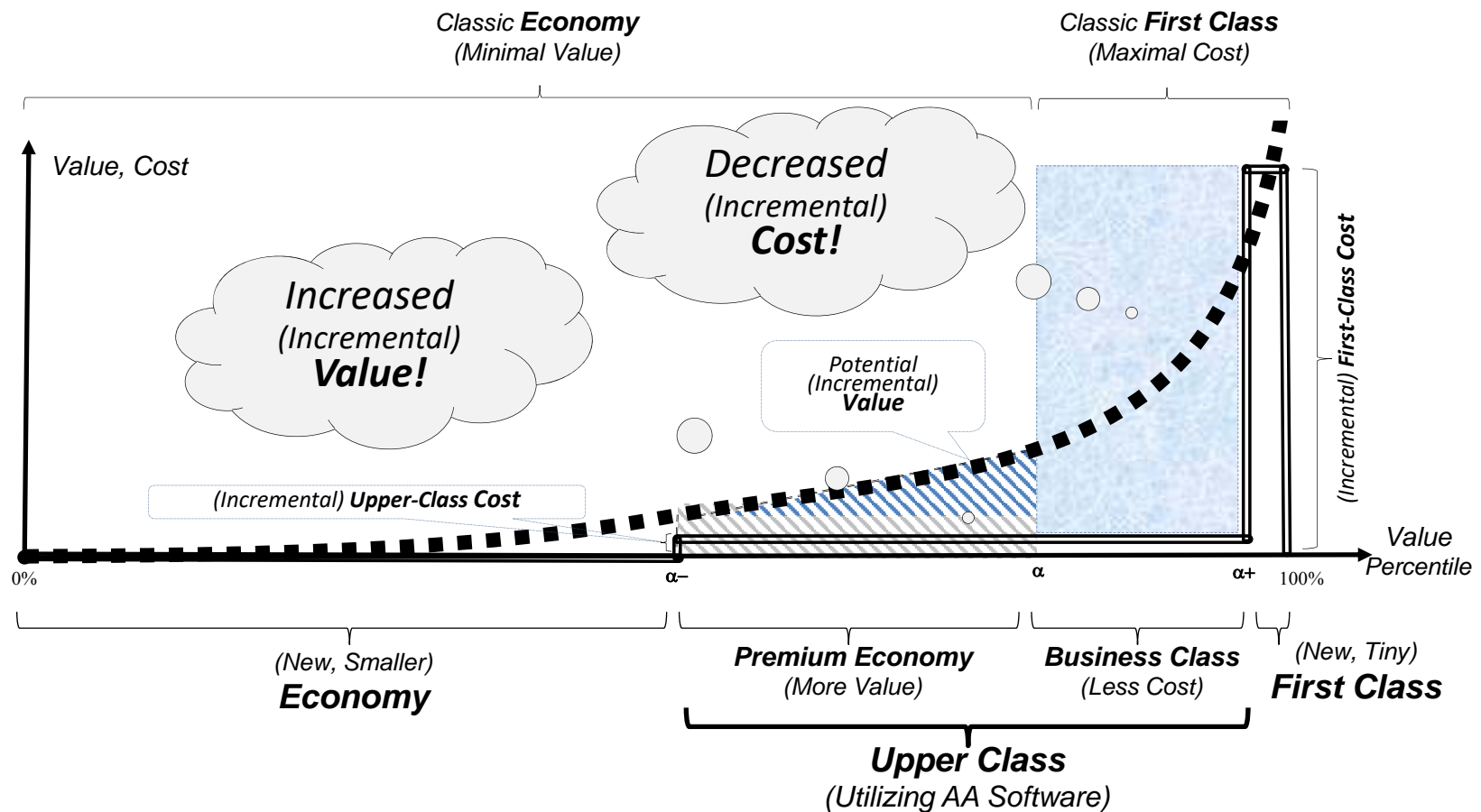
# Introduction

Client Perspective



# Introduction

## Cost/Benefit of Utilizing Allocator-Aware (AA) Software



1. Introduction  
End of Section

**Discussion?**

1. Introduction  
End of Section

Questions?

## 1. Introduction

# What Questions Are We Answering?

- Question 0
- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Question 7

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”



# Outline

1. Introduction
2. *Styles for Allocator-Aware (AA) Software*
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”

## 2. Style for *Allocator-Aware (AA)* Software

# Style Alternatives

TIMTOWTDI (Pronounced “*Tim Toady*”)

- Three models
  - C++11
  - PMR (a.k.a. C++17)
  - BB20V
- Four interface styles
  - C++11
  - BDE and C++17/PMR
  - BB20V

## 2. Style for *Allocator-Aware (AA)* Software

# C++11-Style [HIGH-COST]

Compile-time centric:

- Pros:
  - Zero overhead (runtime/space): default allocator
  - Allows non-standard addressing: shared memory
- Cons:
  - Forces clients to be templates
  - Raises interoperability issues
  - Complex and difficult (extremely) to implement
  - Not widely used except default; *very* widely disliked

## 2. Style for *Allocator-Aware (AA)* Software

# BDE-Style [MODERATE-COST]

Runtime Centric (Doesn't Invade Object's Type):

- Pros:
  - Client's of AA objects need not be templated
  - Enhanced Interoperability (e.g., vocabulary types)
  - Reduced implementation cost (can be automated)
- Cons:
  - Non-zero runtime and spatial overhead
  - Significant implementation and maintenance costs

## 2. Style for *Allocator-Aware (AA)* Software

# PMR-style [MODERATE-COST]

### PMR-style (a.k.a. C++17 Style)

- Based on same Model as BDE style
- Small syntactic difference:
  - base-class pointer is wrapped in a C++11-style-compliant object
- Expected to supplant BDE-style at Bloomberg
  - E.g., BDE 4.0

## 2. Style for *Allocator-Aware (AA)* Software

### BB20V-style [LOW-COST]

Language support for PMR-style allocators

- Some annotation will denote a class as *AA*.
- Compiler does (almost) all of the “plumbing”...  
... compiler-generated constructors too!
- Allocators are injected independently of the constructor signatures...  
... vaguely similar to installing a VTAB PTR.

## 2. Style for *Allocator-Aware (AA)* Software

# Style Alternatives

No matter what the *AA* style ...

- Near same performance as bespoke solutions
- Much lower cost
- Important additional collateral benefits

2. Style for *Allocator-Aware (AA)* Software

End of Section

Discussion?



2. Style for *Allocator-Aware (AA)* Software

End of Section

Questions?

## 2. Style for *Allocator-Aware (AA)* Software

# What Questions Are We Answering?

- Question 0
- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Question 7

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”

### 3. Performance Benefits

# Considerations

- Performance gains arise from:
  1. Faster allocator/deallocation calls
  2. Improved memory access (locality)
- Which dominates?
  - Short-running programs: *faster allocation calls*
  - Long-running programs: *improved memory access*

### 3. Performance Benefits

## Allocation/Deallocation

#### Common Usage Pattern 1:

- Build up data structures (few or no deletes), access them (briefly), then tear them down.
- ***Monotonic allocator:***
  - Deallocation is a no-op
  - Memory returned when allocator destroyed
  - Typically used from within a single thread

### 3. Performance Benefits

## Allocation/Deallocation

### Common Usage Pattern 2:

- Repeatedly allocate/deallocate blocks of a few distinct sizes.
- ***Multipool* allocator:**
  - Dynamically growing pools of fixed-size blocks based on usage
  - Deallocated blocks are retained for re-allocation
  - Two variants: Thread-safe or not

### 3. Performance Benefits

## Allocation/Deallocation

### Common Usage Pattern 3:

- Need to destroy many objects *en masse*, **and** objects own no resources except memory
- ***Managed* allocator:**
  - Has method that releases all memory for reuse
  - Object destructors are **not** called
  - Supported by both ***monotonic*** and ***multipool***
  - Most local allocators are naturally *managed* ones



### 3. Performance Benefits

## Memory Locality

Locality of data in time/space is important.

- Multi-level hardware caching most effective when related data is physically close.
- Long-running programs that repeatedly allocate and deallocate can *diffuse* initially localized data.
- Loss in locality often dominates (“pwnz”) runtime performance of allocate/deallocate.
- Local (arena) allocators attenuate *diffusion*.

### 3. Performance Benefits

## Thread Locality

Local memory allocators facilitate threading

- If distinct threads have their own allocators, synchronization (e.g., using mutexes) can often be avoided or drastically reduced.
- If distinct threads use separate arena allocators, accidental cache-line contention (a.k.a. *destructive interference*, *false sharing*) is naturally avoided.

### 3. Performance Benefits

## Maximizing Performance

Achieving maximum performance requires

- Global knowledge of the application
- Solid understanding of different allocator characteristics

3. Performance Benefits  
End of Section

Discussion?

3. Performance Benefits  
End of Section

Questions?

### 3. Performance Benefits

## What Questions Are We Answering?

- Question 0
- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Question 7

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. **Costs**
5. Collateral Benefits
6. “Concerns”



## 4. Costs

# Creating and Exploiting AA

Two different kinds of costs

### 1. Up-front costs **creating** (and maintaining) AASI

E.g., “Plumbing” constructors to propagate user-supplied allocators to all the various subobjects

➤ Borne (mostly) by library/infrastructure developers

### 2. Incremental costs **exploiting** (or ignoring) AASI

E.g, Ongoing cognitive burden due to increased interface (and contract) complexity; chance for misuse

➤ Borne by many (*most?*) application developers

## 4. Costs

### *Up-Front* (LIBRARY DEVELOPMENT) Costs

Converting an allocator-unaware class to AA

- For typical\* classes, relatively straightforward
  - Add optional trailing allocator to every constructor.
  - Forward the new argument to base classes, data members, and any other managed sub-objects.
  - Denote the type as AA using an allocator-trait metafunction.
- Non-typical classes are more challenging:
  - E.g., Generic, template, and container types

## 4. Costs

### *Up-Front* (LIBRARY DEVELOPMENT) Costs

Converting a generic container/template to AA

- Template types – e.g., `std::complex`
  - Requires interacting with AA-ness of element type
- Container types – e.g., `std::vector`
  - Involves touching methods other than constructors
- Non-allocating templates – e.g., `std::pair`
  - Templated type does not itself allocate memory
- Irregular types – e.g., `std::shared_ptr`
  - Requires domain knowledge of intended purpose

## 4. Costs

# *Up-Front* (LIBRARY DEVELOPMENT) Costs

### Maintenance burden

- More source code
  - AA code is roughly  $\sim 10\%^*$  [4% – 17%] larger
- More training
  - Learning to write (and properly test) AA types
- Opportunity cost
  - Can require a lot of expert library developers' time
  - Other important projects might be delayed

\*-Measurment made on BDE code base (c 2017).

## 4. Costs

# *Up-Front* (LIBRARY DEVELOPMENT) Costs

### Mitigating factors

- Readily lends it self to automation
  - `bde_verify`, a currently-available static-analysis tool, catches most common errors.
  - *BB20V* will eliminate (*most?*) manual “plumbing” ...
- Developing *BB20V* technology is itself a substantial one-time up-front cost.
  - Analogous to self-driving car technology... (tbc...)

## 4. Costs

# *Incremental* (APPLICATION-DEVELOPER) Costs

Typical cost of using AASI is comparatively small\*

- Much easier/faster than “rolling your own”
  - Simply supply desired allocator at construction
  - Does (of course) require additional testing effort
- No need for custom memory allocation?
  - Ignore AA parameters
  - Use and test normally
  - Use is entirely “opt in”

\*We'll discuss *modern C++ style* later in this talk.

## 4. Costs

# *Incremental* (APPLICATION-DEVELOPER) Costs

### Additional cognitive burden

- Users will still see AA features
  - Enlarged (programmatic) interface:  
e.g., Trailing allocator argument in every constructor
  - Enlarged (English) contracts (e.g., for constructors):  
e.g., “Optionally specify a basic allocator to supply...”
- Although the net benefit for those who exploit AA clear, the overall net user benefit is less so.

## 4. Costs

### *Incremental* (APPLICATION-DEVELOPER) Costs

Additional opportunity for client misuse

- Allowing an object to outlive its allocator
  - [rare] by, say, returning a dynamically allocated object, created using a local (e.g., stack) allocator
- Inappropriate use of special-purpose allocator
  - [common] by, say, repeatedly reusing a monotonic allocator created outside of a long-running loop
- Misuse can be catastrophic or simply fail to improve performance – either way it's a cost!



## 4. Costs

# *Incremental* (APPLICATION-DEVELOPER) Costs

Incompatibility with some modern C++ features

- AA classes require non-trivial CTORs
  - Compiler-generated copy operations won't work
  - Problem is exacerbated by C++11 move variants
  - Aggregate initialization is not currently\* available
- The assertion that “*allocators do not interact well with modern C++ move semantics*” is **false!**
  - We will demonstrate why/how later on in this talk.

\*The (language-based) BB20V-style eliminates all such syntactic incompatibilities

## 4. Costs

# *Incremental* (APPLICATION-DEVELOPER) Costs

### Lifetime management issues

- The (productive) lifetime of an object must not exceed that of its allocator.
  - Requires additional care by application developers
- Limits applicability of certain standard facilities that manage object lifetimes as they neither track nor extend allocator lifetimes.
  - E.g., `std::shared_ptr` and `std::weak_ptr`

## 4. Costs

# *Incremental* (APPLICATION-DEVELOPER) Costs

Education, tools, and governance

- Additional administrative costs of AA software
  - Proper training (continuing education)
  - Code reviews (by properly trained reviewers)
  - Developer-facing (e.g., static analysis) tools
  - Company-wide policies (on allocator-usage)
- Not atypical of other powerful paradigms
  - E.g., Multithreading, unit testing, and C++ itself!

## 4. Costs

# Creating and Exploiting AA

### Bottom line

- Real, substantial costs exist
  - [substantial] Up-front library development costs
  - [modest] Incremental application developer costs
- A credible value proposition remains
  - If we don't have (hierarchically) reusable AASI then some application developers will need to write it.
  - All the rest will be forced to do without it.

4. Costs

End of Section

Discussion?

4. Costs

End of Section

Questions?

## 4. Costs

# What Questions Are We Answering?

- What is the % of code that benefits allocators?
- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Question 7

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”



# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”

## 5. Collateral Benefits

# ... but Wait! There's More!

Apart from frequent (and sometimes dramatic) performance gains...

- ...investing in an AASI provides other benefits
  - rapid prototyping; modularity; (hierarchical) reuse; testing; instrumentation; object placement
- Investment in ultra-performance-tuned, “one-off” data structures are unlikely to provide any of these valuable **collateral benefits**.

## 5. Collateral Benefits

# Rapid Prototyping, and Predictability

- Design with AA is low-cost and low-risk.
  1. Select from suite of existing allocation algorithms.
  2. Plug into AASI components in application.
  3. Measure!
  4. Tune.
  5. Repeat, as needed.
- **Deploy immediately!**

## 5. Collateral Benefits

# Rapid Prototyping, and Predictability

- Design with AA is low-cost and low-risk.
  1. Select from suite of existing allocation algorithms.
  2. Plug into AASI components in application.
  3. Measure!
  4. Tune.
  5. Repeat, as needed.
- Deploy immediately! And/or **use as proof-of-concept for custom-data-structure project!**

## 5. Collateral Benefits

# Modularity and Composition (Reuse)

The BDE-style allocators are *chainable*.

- I.e., One allocator provides some functionality, then goes to its *backing* allocator when additional memory is needed.
- Examples:
  - A “small block” allocator can “fall back” on a “large block” one for big memory chunks as needed.
  - One allocator provides some features (e.g. metrics gathering) and “falls back” to another for memory.

## 5. Collateral Benefits

# Testing and Instrumentation

**Testing:** `bslma::TestAllocator`

- Check for memory leaks
  - Log allocate/deallocate calls
  - Match deallocations with known allocations
- Test exception safety
  - Throw `bsl::bad_alloc` on cue in tests
- Test for memory-range overwrites (sentinels)
- Non-invasive
  - Works on arbitrarily large-scale code

## 5. Collateral Benefits

# Testing and Instrumentation

### **Instrumentation:** *Tagged Allocator Store (TAS)*

- Monitor memory usage on an object basis
  - Leverages `bslma::Allocator` vocabulary type
  - Multiply inherits `gtkma::AllocatorStore`
  - Uses `dynamic_cast` to “opt in” to reporting
- Strictly better than other solutions
  - “Opt In” is fine-grained and entirely optional
  - Provides object- as opposed to class-based info
  - Works on arbitrarily large-scale code

## 5. Collateral Benefits

# Whole-Object Placement

Placement of objects in memory is important!

- Allocators facilitate the placement of (entire) objects in “special” memory.
  - (placement `new` is for only the top-level footprint)
- Examples
  - High-bandwidth memory (HBM)
  - Hardware protected (no read and/or write access)
  - Persistent or file-mapped (`mmap`) memory
    - The `gmalloc` allocator is but one relevant example



## 5. Collateral Benefits

# Garbage Collection

Sometimes we need to get down to the metal

- Traditional use of managed pointers can be unnecessarily expensive in both time and space.
- Large (many-node) data structures built out of *raw pointers* can be summarily “winked out”!
  - The release method of a (managed) allocator will unilaterally reclaims all memory (*w/o destructors*).
  - **Requirement:** The data structures own no resources other than memory from that allocator.

## 5. Collateral Benefits

# Pluggable Customization

The utility of an AASI for realizing performance and other, collateral benefits are open-ended.

- Most (but not all) of these benefits depend largely (albeit indirectly) on the ability to inject allocators into a system ***at runtime***.
- Without having invested in an AASI, the cost of pursuing such benefits would require prohibitive expenditure of time and effort – especially w.r.t. to bespoke data structures.

5. Collateral Benefits  
End of Section

Discussion?

5. Collateral Benefits  
End of Section

Questions?

## 5. Collateral Benefits

# What Questions Are We Answering?

- With use of smart pointer, do we need to test?
- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Question 7

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”

## 6. “Concerns”

# Why the Quotes?

Classical allocators specifications have sucked!

- C++98 allocators didn't work at all!
  - stateless (and completely useless) – Lakos'96
- C++03 allocators had “weasel words”
  - Not portable: allowed but not required to work
- C++11 allocators are a pain in the @SS!
  - Hard to write; invade object type; *very* hard to use
- C++17 allocators are much better
  - Runtime polymorphic; much easier to write/use



## 6. “Concerns”

# Why the Quotes?

People *invent* “reasons” for *not* liking allocators

- State-of-the-art allocators are as good or better
- PMR violates the zero-overhead principle (ZOP)
- (Generally) poor runtime performance trade-off
- (Unmanageable) verification/testing complexity
- (Gross) incompatibility with modern C++ style
- Don’t play nice w/modern C++ move semantics
- *Object pools* and *factories* are as good or better

## 6. “Concerns”

# State-of-the-Art Global Allocators

“Advances in global memory allocators have led to dramatic performance improvements – especially with respect to real-world multithreaded applications; wouldn’t replacing the compiler-supplied global memory allocator with a newer, “state-of-the-art” one achieve most (if not all) of the real benefits derived from assiduous use of local allocators designed into a program?”

## 6. “Concerns”

# State-of-the-Art Global Allocators

Global allocators are not (cannot be) sufficient.

- General-purpose global allocators are ignorant of application-specific details.
- They cannot achieve the locality that local allocators can.
- They cannot not provide the collateral benefits.

## 6. “Concerns”

# Zero-Overhead-Principle Compliance

“For all but the C++11 model, AA objects (1) require maintaining extra state – even for the most common case (i.e., where the default allocator is used) – and (2) necessarily employ virtual-function dispatch when allocating and deallocating memory; isn’t that too inefficient for AA software to be viable in C++?”

## 6. “Concerns”

# Zero-Overhead-Principle Compliance

Neither *letter* nor *spirit* of ZOP is violated.

- The needed “extra space” can be addressed
  - Used only upon allocation
  - Stored outside the footprint
  - Elided in common case(s) especially the default
- The virtual-function dispatch “overhead”
  - Can be bound *at compile time* in relevant cases
  - Is invariably negligible compared to added locality
  - Is generally a red herring: allocators boost runtime

## 6. “Concerns”

# Zero-Overhead-Principle Compliance

AA software makes prudent design trade-offs

1. Benefits to some with negligible cost to others
  - Implementation change:  $O(N) \rightarrow O(\log N)$
2. Solid benefits for a few but small cost to other
  - `std::list<T>::size()` must be  $O(1)$
3. Large benefit for expected case but significant cost for others
  - Short-string optimization (SSO)
    - Especially costly for (sparse) vectors of string data

## 6. “Concerns”

# Zero-Overhead-Principle Compliance

AA software makes these design trade-offs

### 1. Benefits to some with negligible cost to others

– Implementation change:  $O(N) \rightarrow O(\log N)$

### 2. Solid benefits for a few but small cost to others

– `std::list<T>::size()` must be  $O(1)$

## Allocator Tax Analogy

Everyone must buy auto insurance:

Accidents are unusual – but not *rare!*

## 6. “Concerns”

# Verification/Testing Complexity

“Failure to properly annotate types or propagate allocators can undermine the effectiveness of the allocation strategy and can lead to memory leaks, especially when ‘winking out’ memory; aren't the extensive verification, testing, and/or peer review required to avoid such errors impracticable?”



## 6. “Concerns”

# Verification/Testing Complexity

Almost every new library or language feature has a learning curve and requires additional testing.

- Allocators are entirely opt-in (can ignore them)
- Special-purpose allocators do require training
- “Winking out” is inherently for experts only
- Static analysis tools (e.g., `bde_verify`) can help
- `bslma::TestAllocator` (e.g., leak testing)
- BB20V-styled *will* help dramatically!

## 6. “Concerns”

# Compatibility with Modern C++ Style

“C++11 encourages a style of programming where objects are more often passed and returned by value, sometimes relying on rvalue references to move these objects efficiently whereas BDE style relies on passing AA objects (by address) as arguments to achieve optimal efficiency and control over the allocator employed; isn't this ‘old-fashioned’ style unjustifiably restrictive?”

## 6. “Concerns”

# Compatibility with Modern C++ Style

Custom allocators do not affect function style

- Returning by value is inherently inefficient
  - The returned object must be constructed each time
  - Supplying an allocator doesn't help
- Returning an object by argument is faster
  - Can reuse object to return multiple values
  - Full control over result allocator in client context
  - Can build returning style on top (but not vice versa)

## 6. “Concerns”

# Move vs. Allocate

“When two objects use different allocators, *move* assignment degenerates to a copy operation and *swap* becomes undefined behavior; doesn't that imply that local allocators should be avoided to enable such operations?”

## 6. “Concerns”

# Move vs. Allocate

Move assignment is often not as efficient as copy!

- Object returned by value are *not* moved
  - They are constructed in place via RVO (or NRVO)
- Moving objects around “mucks” with memory
  - i. Locality (cache-lines, caches, pages, etc.)
  - ii. Constructive interference (a.k.a. “true sharing”)
  - iii. Prefetching
  - iv. Optimal N-way cache access
- Moving within a container (or an “arena”) is OK
  - Preserves i and ii (above) but not necessarily iii or iv.

## 6. “Concerns”

### Compared to Non-AA Alternatives

“Object pools and factories serve to reduce overhead caused by allocating memory; so why aren’t these other approaches as good (if not better) alternatives to allocators?”

## 6. “Concerns”

### Compared to Non-AA Alternatives

“Object pools and factories serve to reduce overhead caused by allocating memory; so why aren’t these other approaches as good (if not better) alternatives to allocators?”

- Memory allocation is reduced, not obviated, and only in certain cases.

## 6. “Concerns”

### Compared to Non-AA Alternatives

“Object pools and factories serve to reduce overhead caused by allocating memory; so why aren’t these other approaches as good (if not better) alternatives to allocators?”

- Memory allocation is reduced, not obviated, and only in certain cases.
- Do moving vans eliminate the need for furniture companies?



## 6. “Concerns”

# Compared to Non-AA Alternatives

Object pools are **not** replacements for allocators.

1. Object pools are not faster than allocators.
2. They are at different levels of abstraction:
  - Object pools minimize construction/destruction
  - Memory pools minimize allocation/deallocation
3. Object pools are created using memory pools
4. Object pools themselves should naturally be AA
  - That way they *too* can enjoy the *collateral benefits!*

6. “Concerns”  
End of Section

Discussion?

6. “Concerns”  
End of Section

Questions?

## 6. “Concerns”

# What Questions Are We Answering?

- Question 0
- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Question 7

# Outline

1. Introduction
2. Styles for *Allocator-Aware (AA)* Software
3. Performance Benefits
4. Costs
5. Collateral Benefits
6. “Concerns”

# Conclusion

## Allocator-Aware Software Infrastructure (AASI):

- Custom memory allocation strategies' impact:
  - Performance
  - Instrumentation
  - Object placement ...
- Historically, required bespoke data structures:
  - Long delivery time
  - Any collateral benefits cost extra
  - No reuse ...

# Conclusion

AASI has real costs:

- “Fixed” engineering costs (for SI developers)
- Added operational costs
  - Documentation
  - Training
  - Developer-facing tools
  - Risk of misuse
- Resistance based on C++11-style allocators

# Conclusion

Investing in an AASI is an economic decision:

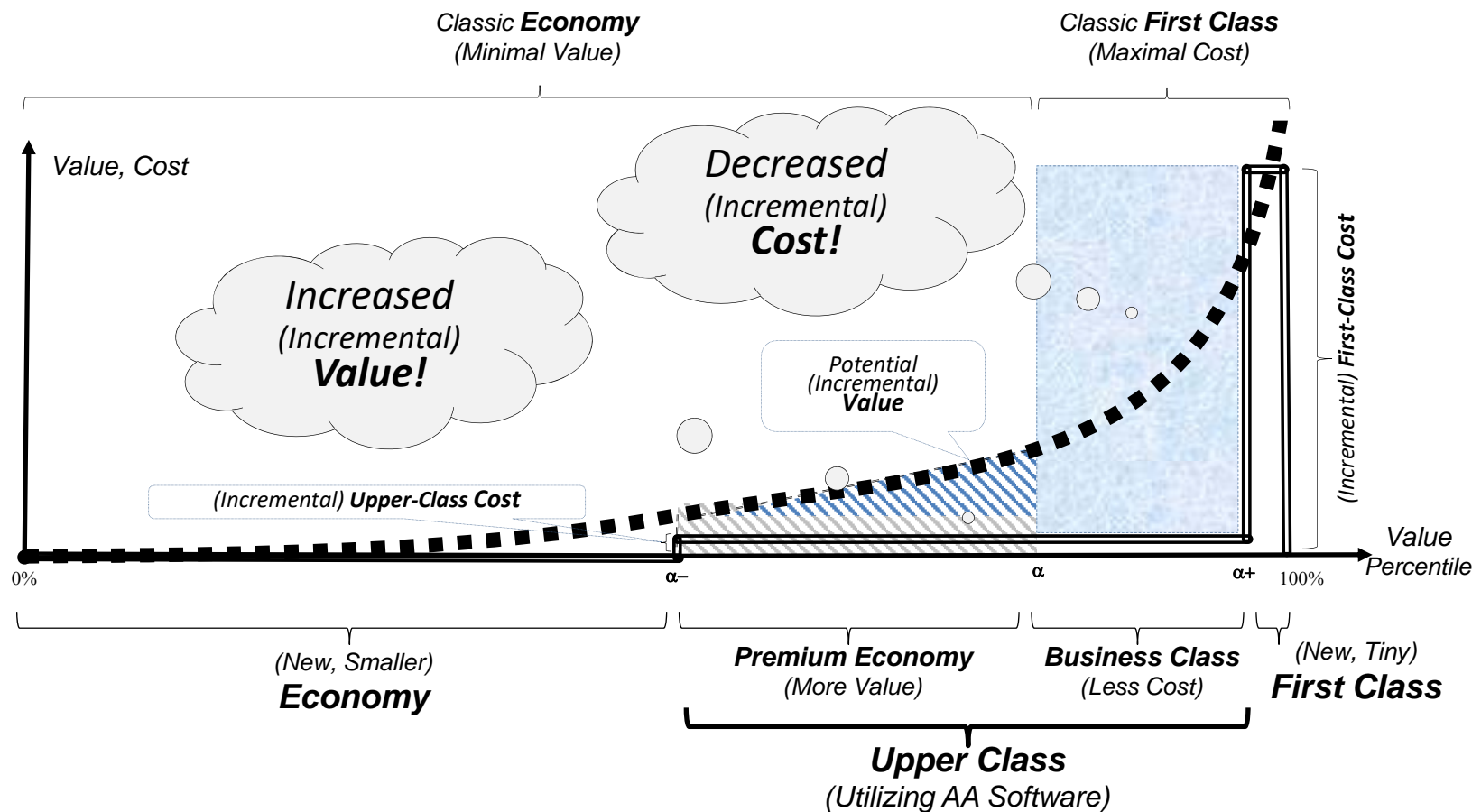
- Provides nearly same runtime performance
- Lower incremental cost -> used more often
- Requires substantial up-front cost
- Comes with important collateral benefits
- C++11 experience -> “concerns” (F.U.D)

Do the benefits outweigh the costs?



# Conclusion

## Cost/Benefit of Utilizing Allocator-Aware (AA) Software



Conclusion

**WAIT!!**

## Conclusion

WAIT!!

What if ***BB20V*** could eliminate  
*all fixed costs entirely?*

## Conclusion

WAIT!!

What if *BB20V* could eliminate  
*all fixed costs entirely?*

Now what do you say?

# Conclusion

Should we (e.g., Bloomberg) invest in AASI?

**→ How can we afford not to?! ←**

- The user benefits outweigh the costs now!

# Conclusion

Should we (e.g., Bloomberg) invest in AASI?

**→ How can we afford not to?! ←**

- The user benefits outweigh the costs now!

## What about BB20V?

- Eliminates (SI-library) “fixed” costs **entirely!**
  - Analogous to self-driving car technology

# Conclusion

Should we (e.g., Bloomberg) invest in AASI?

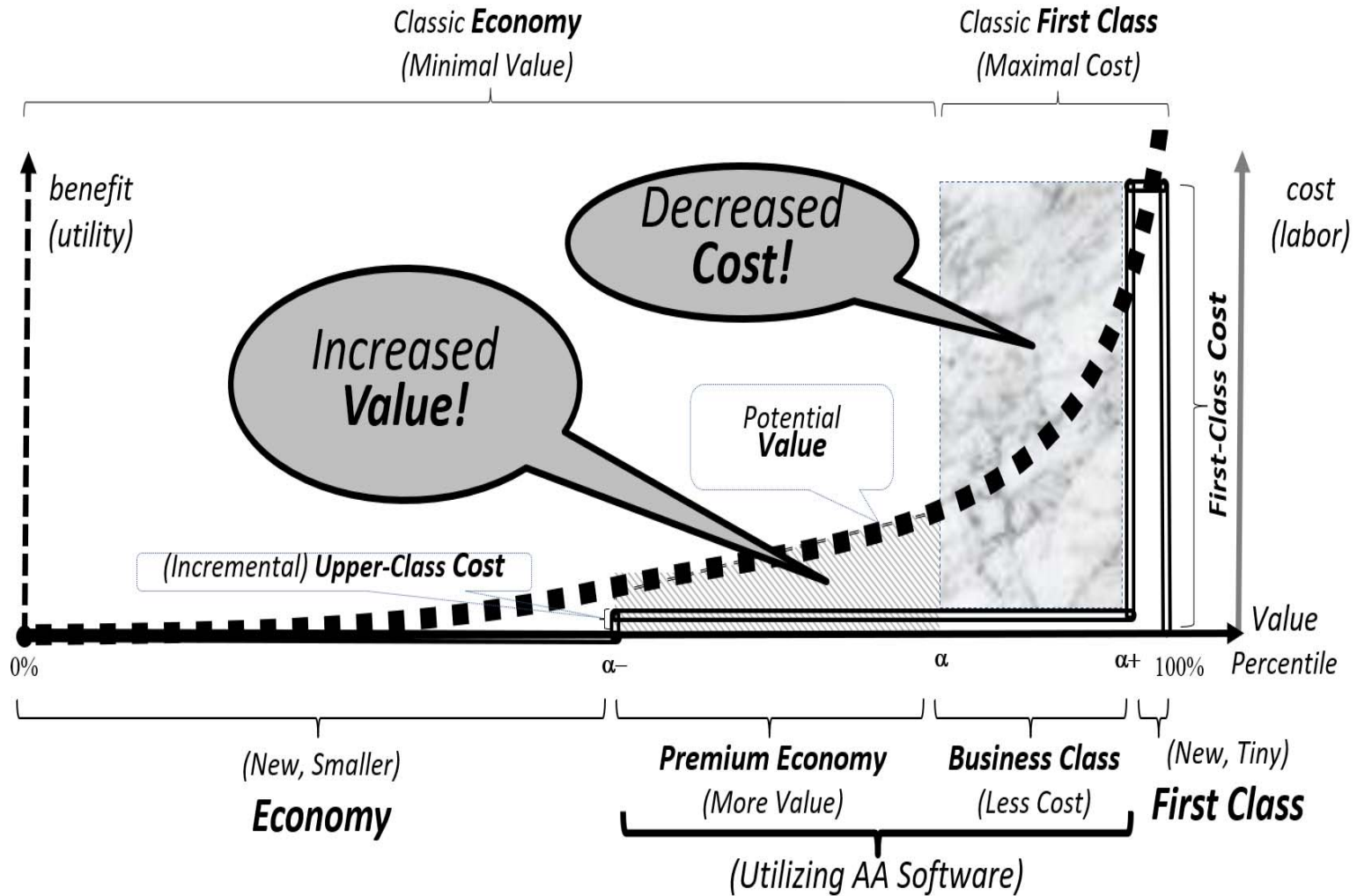
➔ **How can we afford not to?! ←**

- The user benefits outweigh the costs now!

## What about BB20V?

- Eliminates (SI-library) “fixed” costs entirely!
  - Analogous to self-driving car technology
- Reduces (client) “use” costs to **bare minimum**
  - Akin to using virtual functions in C++ today

# Conclusion





Conclusion

The End