

Applied Hierarchical Reuse

*Capitalizing on Bloomberg's
Foundation Libraries*

John Lakos

Wednesday, May 15, 2013

Copyright Notice

© 2013 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

Designing one library is hard; designing an open-ended collection of interoperable libraries is harder. Partitioning functionality across multiple libraries presents its own unique set of challenges: Functionality must be easy to discover, redundancy must be eliminated, and interface and contract relationships across components and libraries should be easy to explore without advanced IDE capabilities. Further, dependencies among libraries must be carefully managed – the libraries must function as a coherent whole, defining and using a curated suite of *vocabulary types*, but clients should pay in compile time, link time, and executable size only for the functionality they need.

Creating a unified suite of interoperable libraries also has many challenges in common with creating individual ones. The software should be easy to understand, easy to use, highly performant, portable, and reliable. Moreover, all of these libraries should adhere to a uniform physical structure, be devoid of gratuitous variation in rendering, and use consistent terminology throughout. By achieving such a high level of consistency, performance, and reliability across all of the libraries at once, the local consistency within each individual library becomes truly exceptional. Additionally, even single-library projects that leverage such principles will derive substantial benefit.

There are many software methodologies appropriate for small- and medium-sized projects, but most simply do not scale to larger development efforts. In this talk we will explore problems associated with very large scale development, and the cohesive techniques we have found to address those problems culminating in a proven component-based methodology, refined through practical experience at Bloomberg. The real-world application of this methodology – including *three levels of aggregation, acyclic dependencies, nominal cohesion, fine-grained factoring, class categories, narrow contracts, and thorough component-level testing* – will be demonstrated using the recently released open-source distribution of Bloomberg’s foundation libraries.

What's The Problem?

What's The Problem?

Large-Scale C++ Software Design:

- Involves many subtle *logical* and *physical* aspects.

What's The Problem?

Large-Scale C++ Software Design:

- Involves many subtle logical and physical aspects.
- Requires an ability to isolate and modularize **logical functionality** within discrete, fine-grained **physical components**.

What's The Problem?

Large-Scale C++ Software Design:

- Involves many subtle *logical* and *physical* aspects.
- Requires an ability to isolate and modularize logical functionality within discrete, fine-grained physical components.
- Requires the designer to delineate **logical behavior** precisely, while managing the **physical dependencies** on other subordinate components.

What's The Problem?

Large-Scale C++ Software Design:

- Involves many subtle logical and physical aspects.
- Requires an ability to isolate and modularize logical functionality within discrete, fine-grained physical components.
- Requires the designer to delineate logical behavior precisely, while managing the physical dependencies on other subordinate components.
- Demands a host of additional considerations in order to maximize wide-spread **hierarchical reuse**.

Purpose of this Talk

There's lots to talk about:

Purpose of this Talk

There's lots to talk about:

- Understand *specific problems* associated with very large-scale development.

Purpose of this Talk

There's lots to talk about:

- Understand *specific problems* associated with very large-scale development.
- Present ***cohesive techniques*** we have found to address these problems.

Purpose of this Talk

There's lots to talk about:

- Understand *specific problems* associated with very large-scale development.
- Present *cohesive techniques* we have found to address these problems.
- Demonstrate *our methodology* using Bloomberg's foundation libraries.

Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment (BDE)

Rendered as Fine-Grained Hierarchically Reusable Components.

Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment (BDE)

Rendered as Fine-Grained Hierarchically Reusable Components.

0. Goals

What Are We Trying To Do?

What are the goals?

- Solve interesting problems.
- Employ new language features early.
- Strive for header-only implementations (i.e., no .cpp files).
- Write code that stress-tests compilers.
- Ensure that no C++ language construct goes unused.

0. Goals

What Are We Trying To Do?

What are the goals?

- Solve interesting problems.
- Employ new language features early.
- Strive for header-only implementations (i.e., no .cpp files).
- Write code that stress-tests compilers.
- Ensure that no C++ language construct goes unused.

0. Goals

What Are We Trying To Do?

Who are the intended clients?

- Enthusiasts hoping to learn about the latest C++ language features.
- Experts capable of reverse engineering advanced C++ implementations.
- Individuals who don't want to learn how to build libraries separately.
- Highly specialized programmers (e.g., those writing embedded systems).
- Folks willing to wait hours (or days) for their programs to compile.

0. Goals

What Are We Trying To Do?

Who are the intended clients?

- Enthusiasts hoping to learn about the latest C++ language features.
- Experts capable of reverse engineering advanced C++ implementations.
- Individuals who don't want to learn how to build libraries separately.
- Highly specialized programmers (e.g., those writing embedded systems).
- Folks willing to wait hours (or days) for their programs to compile.

0. Goals

Our Intended Clients

0. Goals

Our Intended Clients

Professional commercial *software engineers* and *application developers*:

0. Goals

Our Intended Clients

Professional commercial *software engineers* and *application developers*:

- Solving real-world industrial-strength problems.

0. Goals

Our Intended Clients

Professional commercial *software engineers* and *application developers*:

- Solving real-world industrial-strength problems.
- Working on a series of projects of arbitrary size.

0. Goals

Our Intended Clients

Professional commercial *software engineers* and *application developers*:

- Solving real-world industrial-strength problems.
- Working on a series of projects of arbitrary size.
- Using a variety of well-known, popular platforms.

0. Goals

Our Intended Clients

Professional commercial *software engineers* and *application developers*:

- Solving real-world industrial-strength problems.
- Working on a series of projects of arbitrary size.
- Using a variety of well-known, popular platforms.
- Committed to aggressive schedules.

0. Goals

Our Intended Clients

Professional commercial *software engineers* and *application developers*:

- Solving real-world industrial-strength problems.
- Working on a series of projects of arbitrary size.
- Using a variety of well-known, popular platforms.
- Committed to aggressive schedules.
- Held to high standards of quality/reliability.

0. Goals

Our Intended Clients

Professional commercial *software engineers* and *application developers*:

- Solving real-world industrial-strength problems.
- Working on a series of projects of arbitrary size.
- Using a variety of well-known, popular platforms.
- Committed to aggressive schedules.
- Held to high standards of quality/reliability.
- Constrained by limited resources.

0. Goals

Our Intended Clients

Professional commercial *software engineers* and *application developers*:

- Solving real-world industrial-strength problems.
- Working on a series of projects of arbitrary size.
- Using a variety of well-known, popular platforms.
- Committed to aggressive schedules.
- Held to high standards of quality/reliability.
- Constrained by limited resources.
- Driven to succeed.

0. Goals

Our Primary Goal

0. Goals

Our Primary Goal

Actively make our *intended clients*
successful, productive, and efficient:

0. Goals

Our Primary Goal

Actively make our *intended clients* ***successful, productive, and efficient:***

- Demonstrate exemplary methodology that scales to projects of all sizes.

0. Goals

Our Primary Goal

Actively make our *intended clients* **successful, productive, and efficient:**

- Demonstrate exemplary methodology that scales to projects of all sizes.
- Provide a framework for developing real-world software effectively.

0. Goals

Our Primary Goal

Actively make our *intended clients* **successful, productive, and efficient:**

- Demonstrate exemplary methodology that scales to projects of all sizes.
- Provide a framework for developing real-world software effectively.
- Address specific problems that are relevant to our clients.

0. Goals

Our Primary Goal

Actively make our *intended clients* **successful, productive, and efficient:**

- Demonstrate exemplary methodology that scales to projects of all sizes.
- Provide a framework for developing real-world software effectively.
- Address specific problems that are relevant to our clients.
- Maintain stable solutions used by many versions of many products.

0. Goals

Our Primary Goal

Actively make our *intended clients* **successful, productive, and efficient:**

- Demonstrate exemplary methodology that scales to projects of all sizes.
- Provide a framework for developing real-world software effectively.
- Address specific problems that are relevant to our clients.
- Maintain stable solutions used by many versions of many products.
- Teach our clients a proven way of writing scalable software.

0. Goals

Our Primary Goal

Actively make our *intended clients* ***successful, productive, and efficient:***

- Demonstrate exemplary methodology that scales to projects of all sizes.
- Provide a framework for developing real-world software effectively.
- Address specific problems that are relevant to our clients.
- Maintain stable solutions used by many versions of many products.
- Teach our clients a proven way of writing scalable software.
- Achieve *wide-spread, fine-grained, hierarchical* reuse.

0. Goals

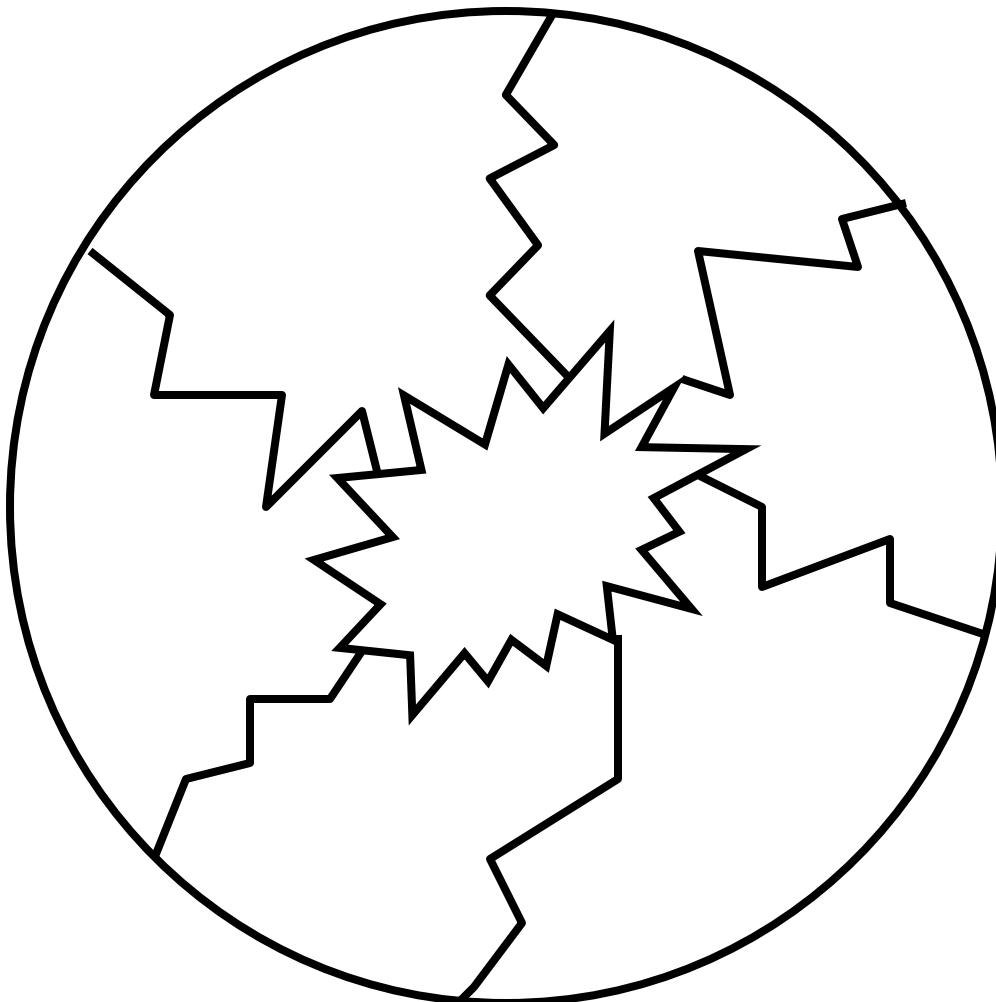
Our Primary Goal

Actively make our *intended clients* ***successful, productive, and efficient:***

- Demonstrate exemplary methodology that scales to projects of all sizes.
- Provide a framework for developing real-world software effectively.
- Address specific problems that are relevant to our clients.
- Maintain stable solutions used by many versions of many products.
- Teach our clients a proven way of writing scalable software.
- **Achieve wide-spread, fine-grained, hierarchical reuse.**

0. Goals

Achieving Reuse



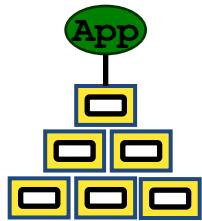
0. Goals

Achieving Reuse



0. Goals

Achieving Reuse

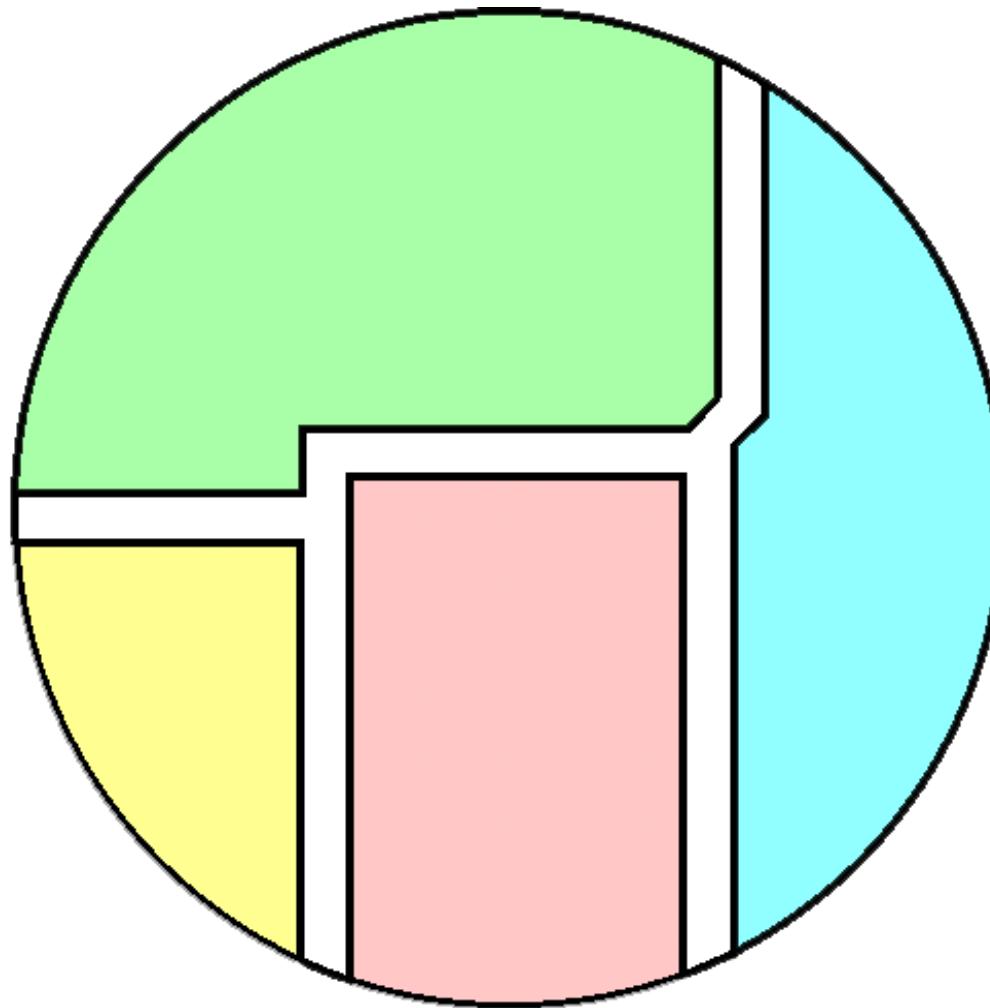


Brittle

Within just
one version of
a single App

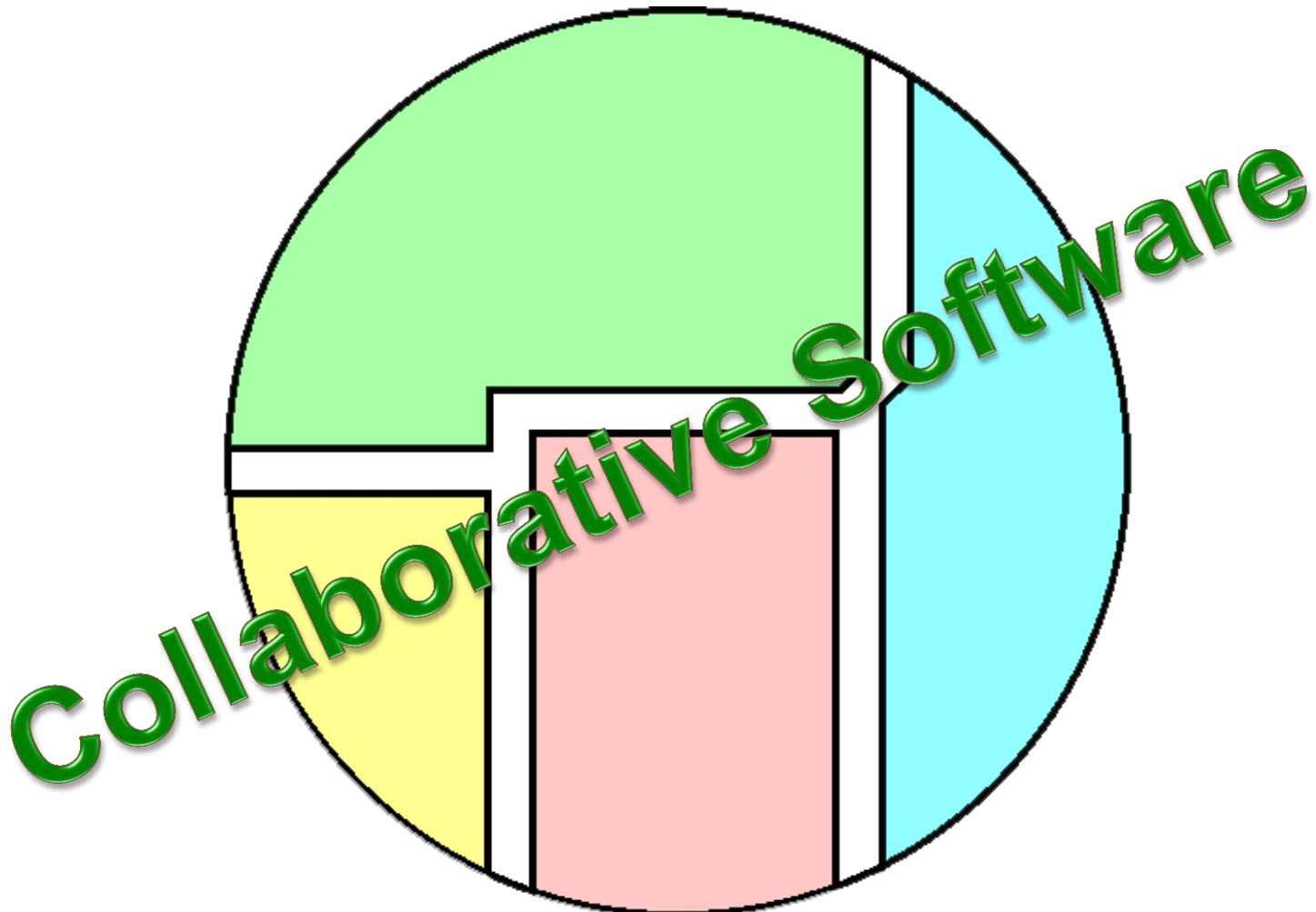
0. Goals

Achieving Reuse



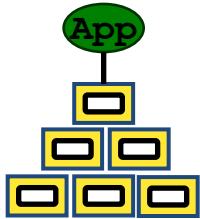
0. Goals

Achieving Reuse



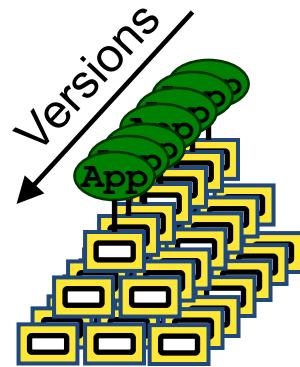
0. Goals

Achieving Reuse



Brittle

Within just
one version of
a single App

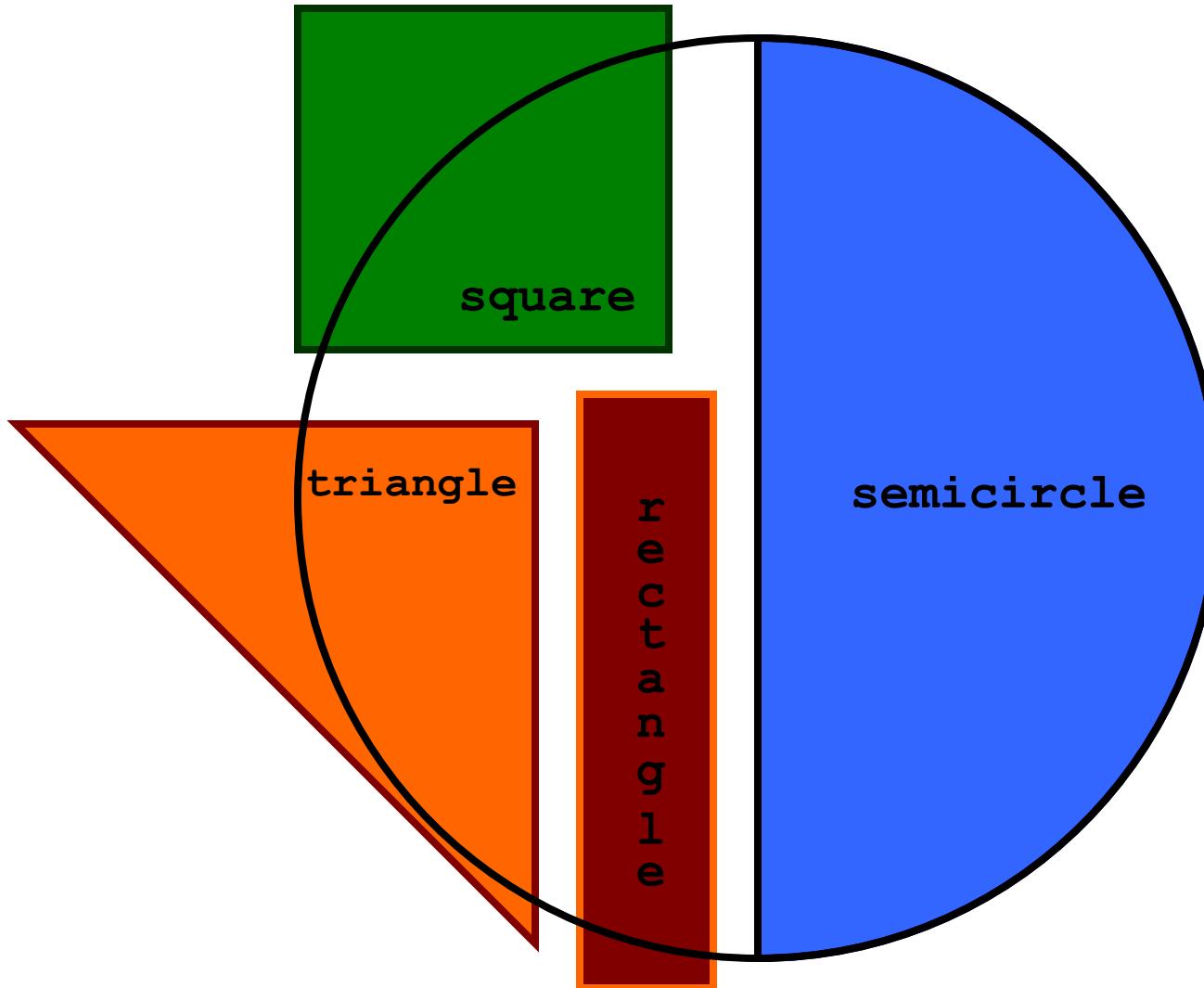


Collaborative

Across several
versions of a
single App

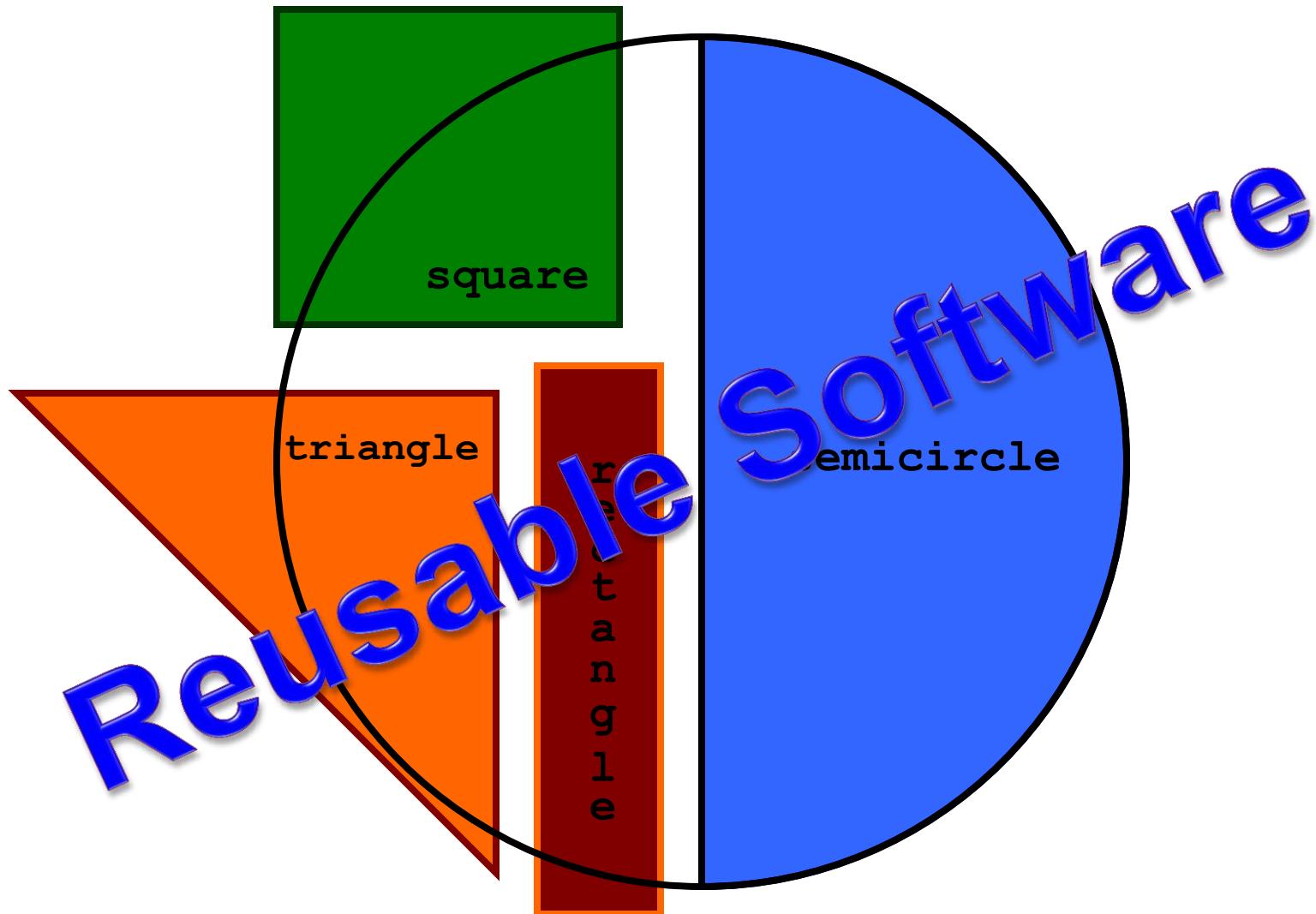
0. Goals

Achieving Reuse



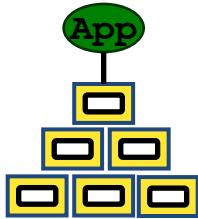
0. Goals

Achieving Reuse



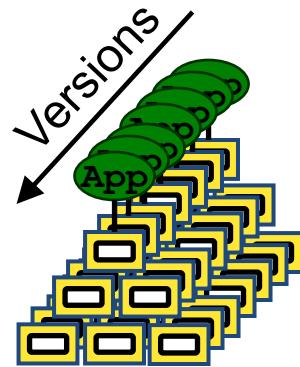
0. Goals

Achieving Reuse



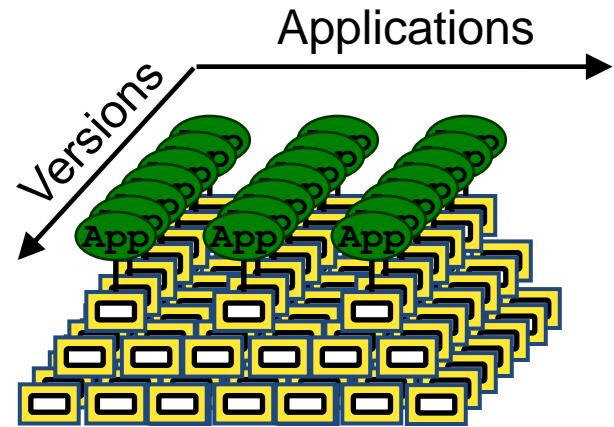
Brittle

Within just
one version of
a single App



Collaborative

Across several
versions of a
single App



Reusable

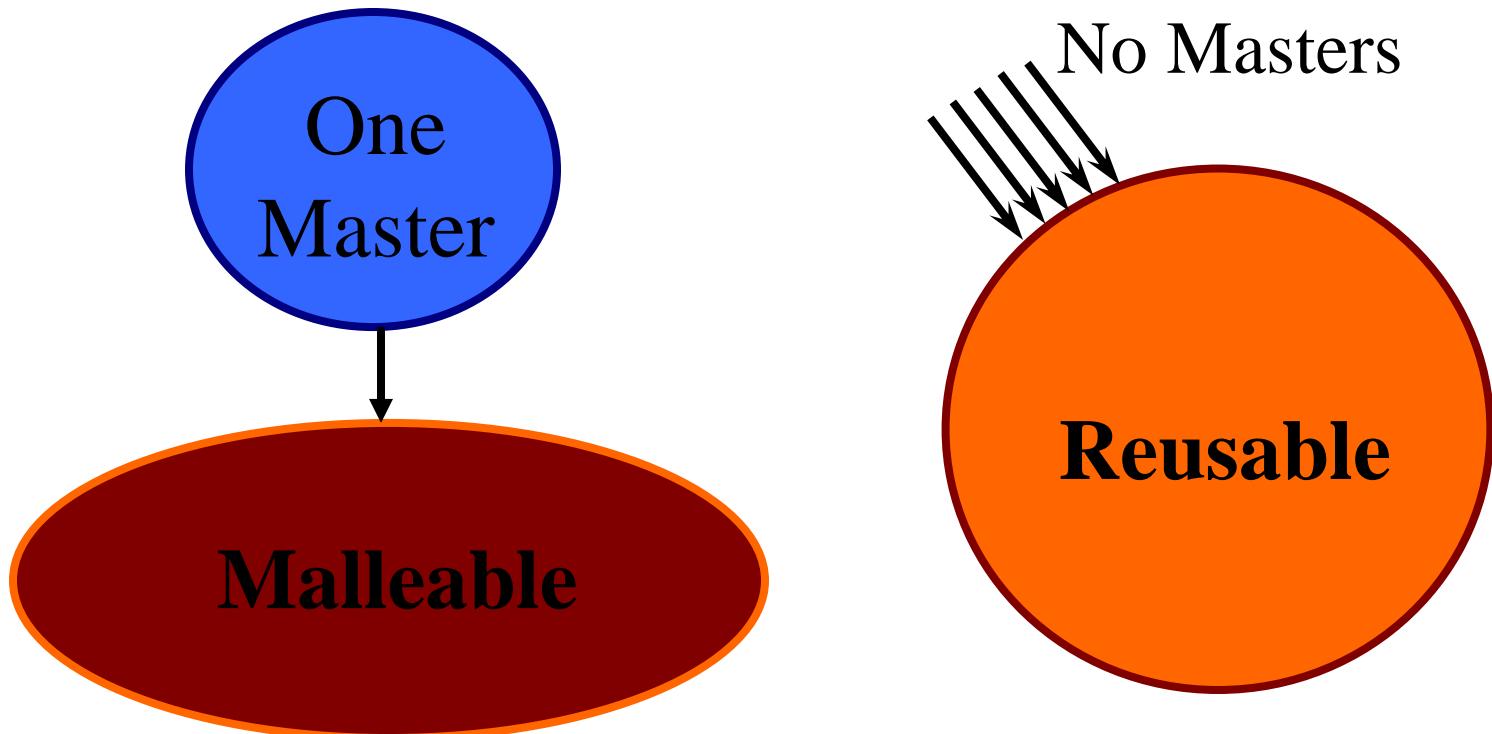
Across several versions
of many distinct
applications and products

0. Goals

Achieving Reuse

Good applications are *malleable*...

...but reusable software is **stable**!



0. Goals

Achieving Fine-Grained Reuse

Fundamental properties of modular software:

0. Goals

Achieving Fine-Grained Reuse

Fundamental properties of modular software:

- Fine-Grained Physical Modularity

0. Goals

Achieving Fine-Grained Reuse

Fundamental properties of modular software:

- Fine-Grained Physical Modularity
- Logical/Physical Coherence

0. Goals

Achieving Fine-Grained Reuse

Fundamental properties of modular software:

- Fine-Grained Physical Modularity
- Logical/Physical Coherence
- No Cyclic Physical Dependencies

0. Goals

Achieving Fine-Grained Reuse

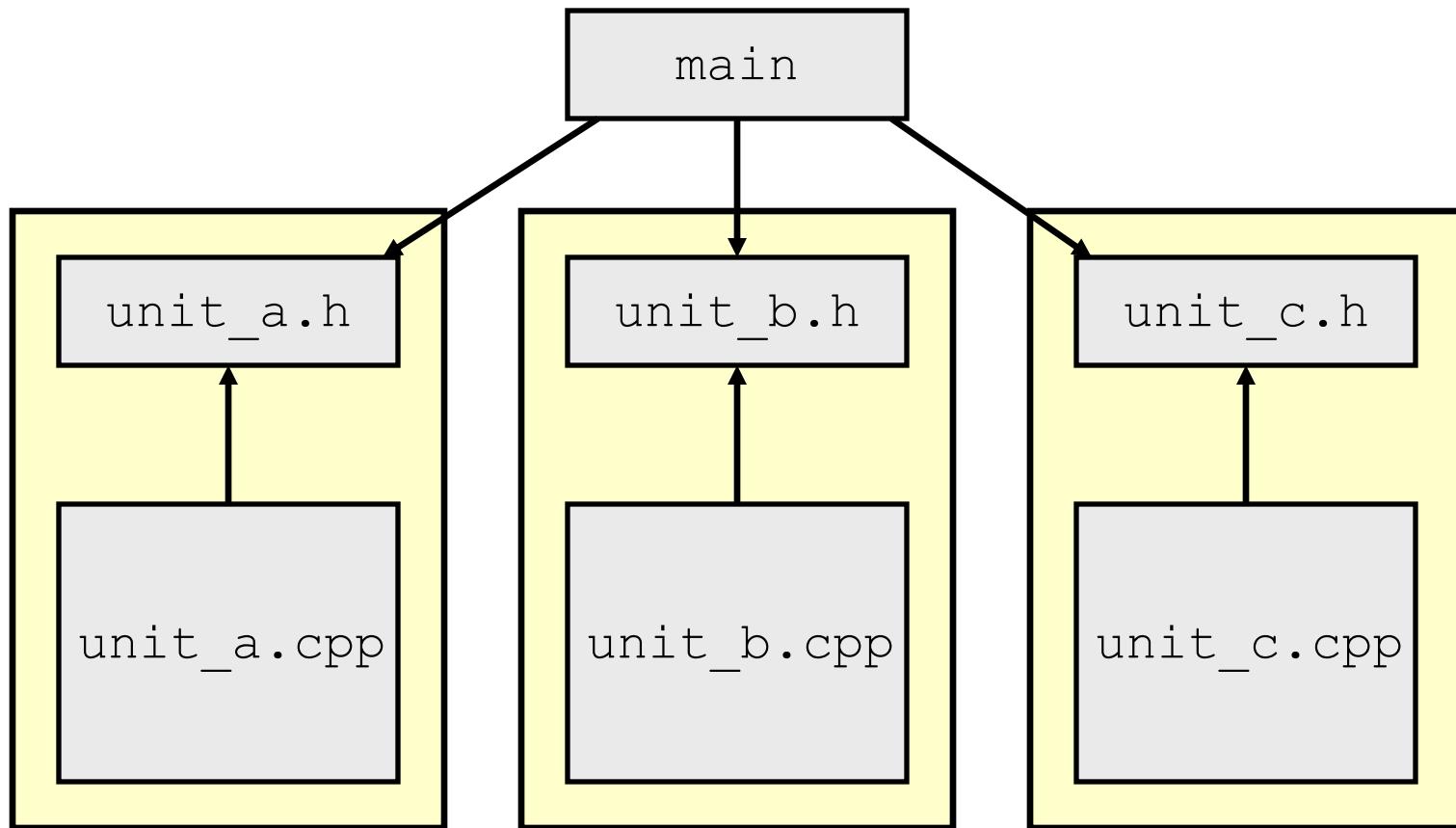
Fundamental properties of modular software:

- Fine-Grained Physical Modularity
- Logical/Physical Coherence
- No Cyclic Physical Dependencies
- No Private "Back Door" Access

0. Goals

Achieving Fine-Grained Reuse

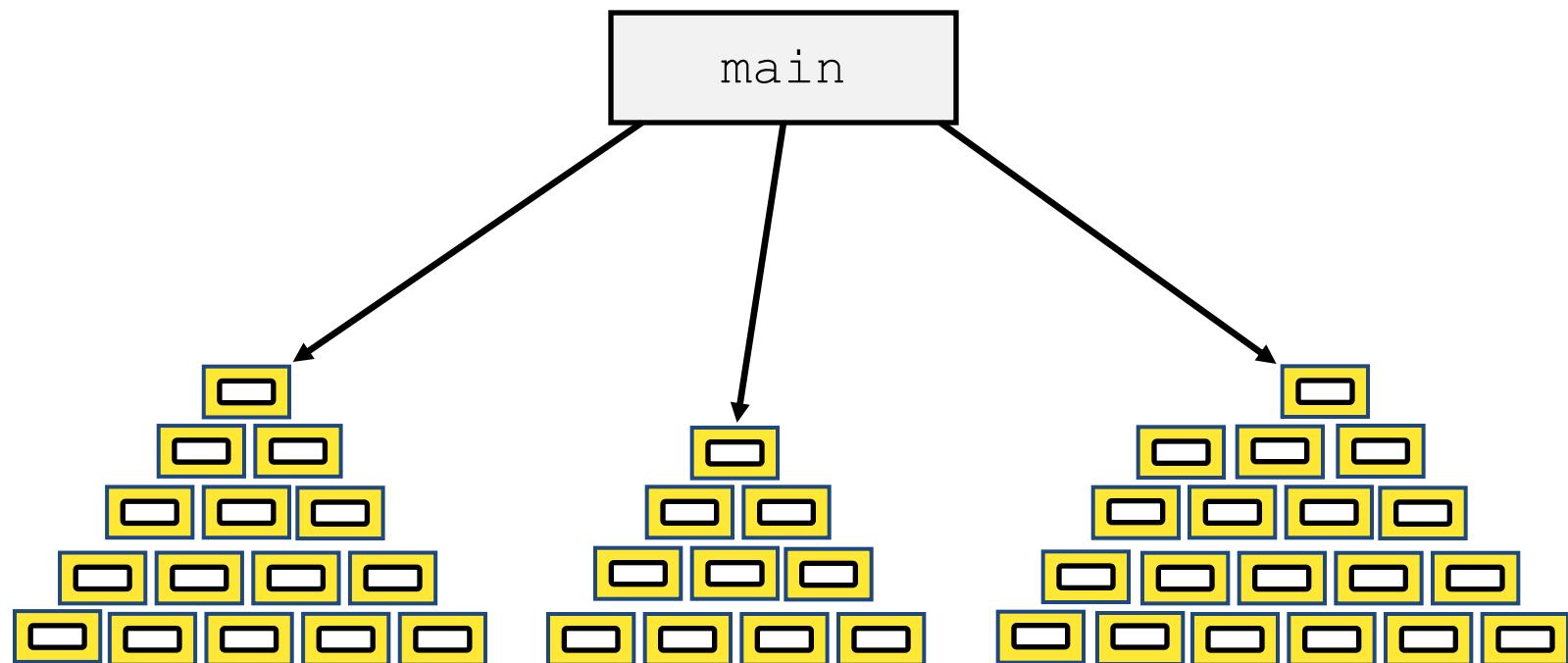
Coarse-Grained Physical Modularity



0. Goals

Achieving Fine-Grained Reuse

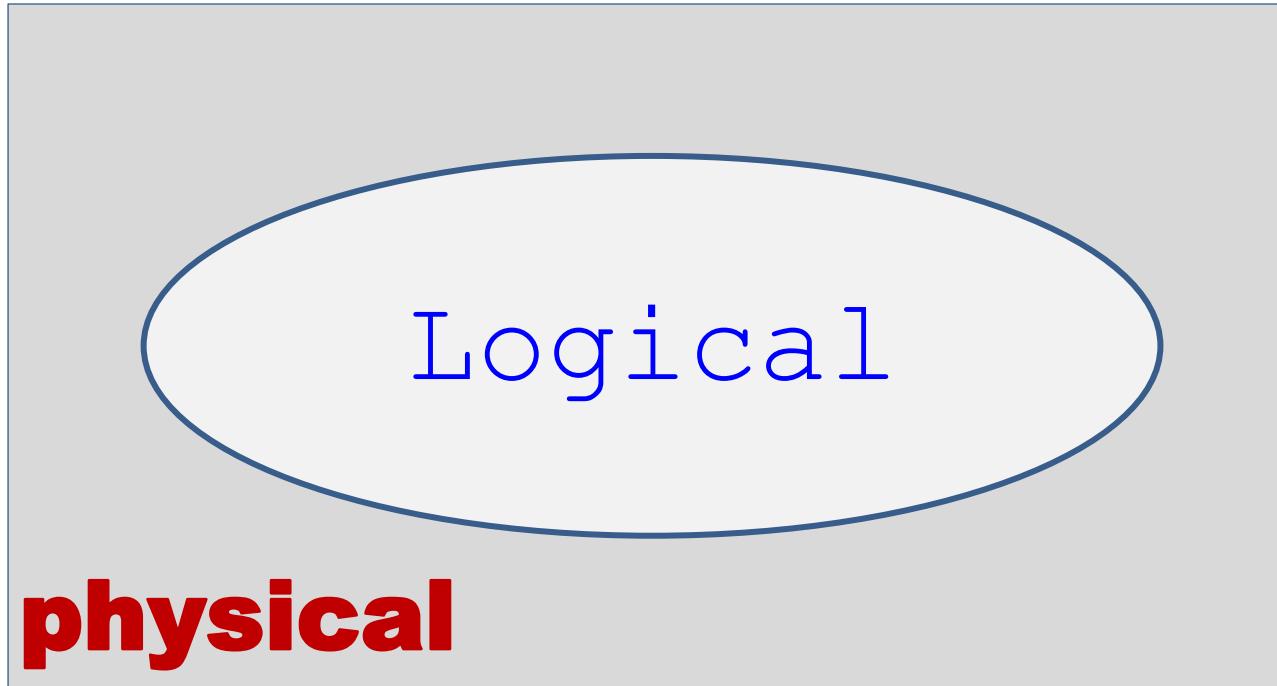
Fine-Grained Physical Modularity



0. Goals

Achieving Fine-Grained Reuse

Logical/Physical Coherence



0. Goals

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET
class IntSet {
    // ...
public:
    // ...
    // ...
    // ...
};
#endif
```

intset.h

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
class Stack {
    // ...
public:
    // ...
    void push(int i);
    int pop();
};
#endif
```

stack.h

```
// intset.cpp
#include <intset.h>
#include <stack.h>
Stack::push(int i)
{
    // ...
}
// ...
```

intset.cpp

```
// stack.cpp
#include <stack.h>
// ...
// ...
// ...
// ...
// ...
```

stack.cpp

```
// main.cpp
#include <intset.h>
#include <stack.h>
int Stack::pop()
{
    // ...
}
// ...
```

main.cpp

0. Goals

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET
class IntSet {
    // ...
public:
    // ...
    // ...
    // ...
};
#endif
```

intset.h

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
class Stack {
    // ...
public:
    // ...
    void push(int i);
    int pop();
};
#endif
```

stack.h

```
// intset.cpp
#include <intset.h>
#include <stack.h>
Stack::push(int i)
{
    // ...
}
// ...
```

intset.cpp

```
// stack.cpp
#include <stack.h>
// ...
// ...
// ...
// ...
// ...
```

stack.cpp

```
// main.cpp
#include <intset.h>
#include <stack.h>
int Stack::pop()
{
    // ...
}
// ...
```

main.cpp

0. Goals

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET
class IntSet {
    // ...
public:
    // ...
    // ...
    // ...
};
#endif
```

intset.h

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
class Stack {
    // ...
public:
    // ...
    void push(int i);
    int pop();
};
#endif
```

stack.h

```
// intset.cpp
#include <intset.h>
#include <stack.h>
Stack::push(int i)
{
    // ...
}
// ...
```

intset.cpp

```
// stack.cpp
#include <stack.h>
// ...
// ...
// ...
// ...
// ... ?
```

stack.cpp

```
// main.cpp
#include <intset.h>
#include <stack.h>
int Stack::pop()
{
    // ...
}
// ...
```

main.cpp

0. Goals

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET
class IntSet {
    // ...
public:
    // ...
    // ...
    // ...
};
#endif
```

intset.h

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
class Stack {
    // ...
public:
    // ...
    void push(int i);
    int pop();
};
#endif
```

stack.h

```
// intset.cpp
#include <intset.h>
#include <stack.h>
Stack::push(int i)
{
    // ...
}
// ...
```

intset.cpp

```
// stack.cpp
#include <stack.h>
// ...
// ...
// ...
// ...
// ... ?
```

stack.cpp

```
// main.cpp
#include <intset.h>
#include <stack.h>
int Stack::pop()
{
    // ...
}
// ...
```

main.cpp

0. Goals

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET
class IntSet {
    // ...
public:
    // ...
    // ...
    // ...
};
#endif
```

intset.h

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
class Stack {
    // ...
public:
    // ...
    void push(int i);
    int pop();
};
#endif
```

stack.h

```
// intset.cpp
#include <intset.h>
#include <stack.h>
Stack::push(int i)
{
    // ...
}
// ...
```

intset.cpp

```
// stack.cpp
#include <stack.h>
// ...
// ...
// ...
// ...
// ... ?
```

stack.cpp

```
// main.cpp
#include <intset.h>
#include <stack.h>
int Stack::pop()
{
    // ...
}
// ...
```

main.cpp

0. Goals

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET
class IntSet {
    // ...
public:
    // ...
    // ...
    // ...
};
#endif
```

intset.h

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
class Stack {
    // ...
public:
    // ...
    void push(int i);
    int pop();
};
#endif
```

stack.h

```
// intset.cpp
#include <intset.h>
#include <stack.h>
Stack::push(int i)
{
    // ...
}
// ...
```

intset.cpp

```
// stack.cpp
#include <stack.h>
// ...
// ...
// ...
// ...
// ... ?
```

stack.cpp

```
// main.cpp
#include <intset.h>
#include <stack.h>
int Stack::pop()
{
    // ...
}
// ...
```

main.cpp

0. Goals

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET
class IntSet {
    // ...
public:
    // ...
    // ...
    // ...
};
#endif
```

intset.h

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
class Stack {
    // ...
public:
    // ...
    void push(int i);
    int pop();
};
#endif
```

stack.h

```
// intset.cpp
#include <intset.h>
#include <stack.h>
Stack::push(int i)
{
    // ...
}
// ...
```

intset.cpp

```
// stack.cpp
#include <stack.h>
// ...
// ...
// ...
// ...
// ... ?
```

stack.cpp

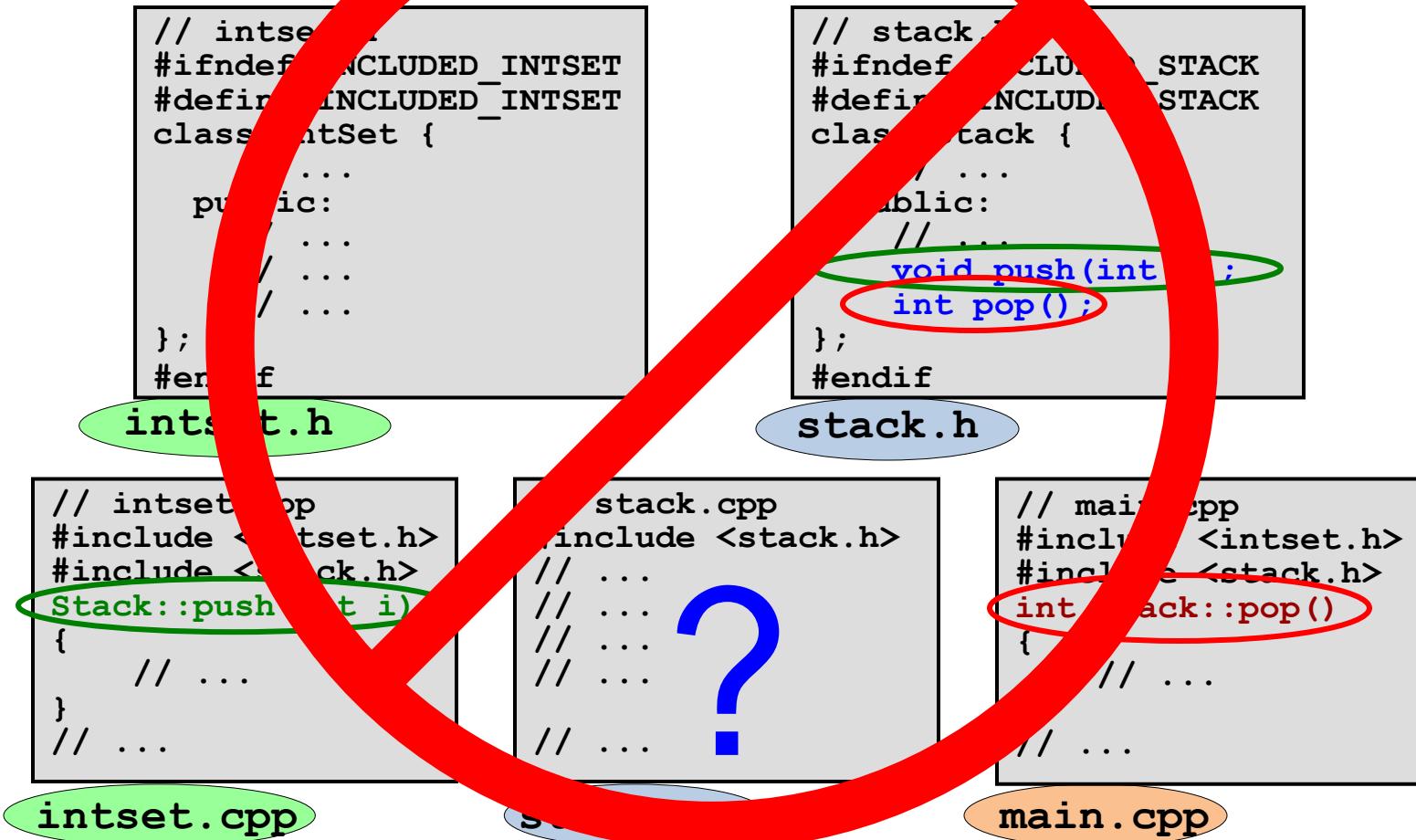
```
// main.cpp
#include <intset.h>
#include <stack.h>
int Stack::pop()
{
    // ...
}
// ...
```

main.cpp

0. Goals

Achieving Fine-Grained Reuse

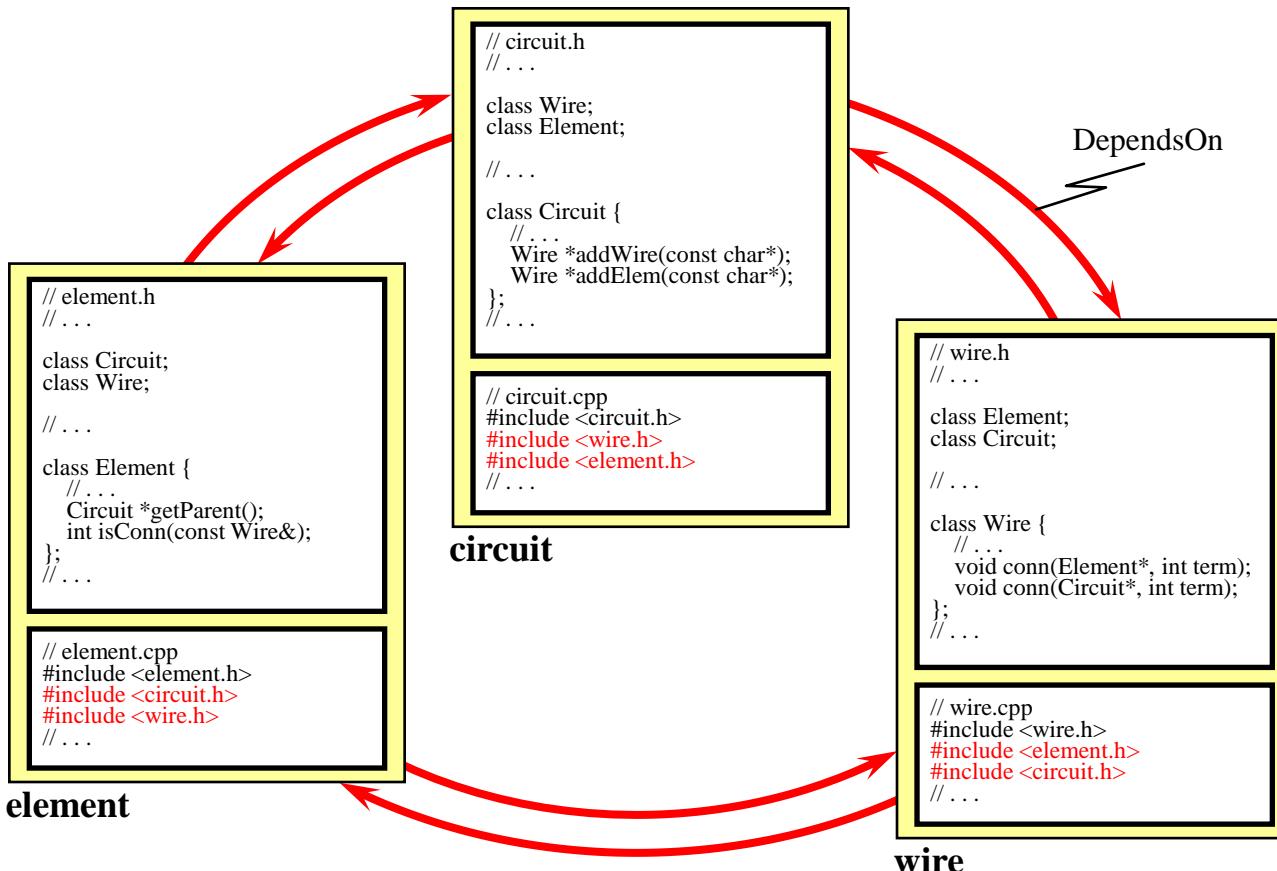
No Logical/Physical Incoherence



0. Goals

Achieving Fine-Grained Reuse

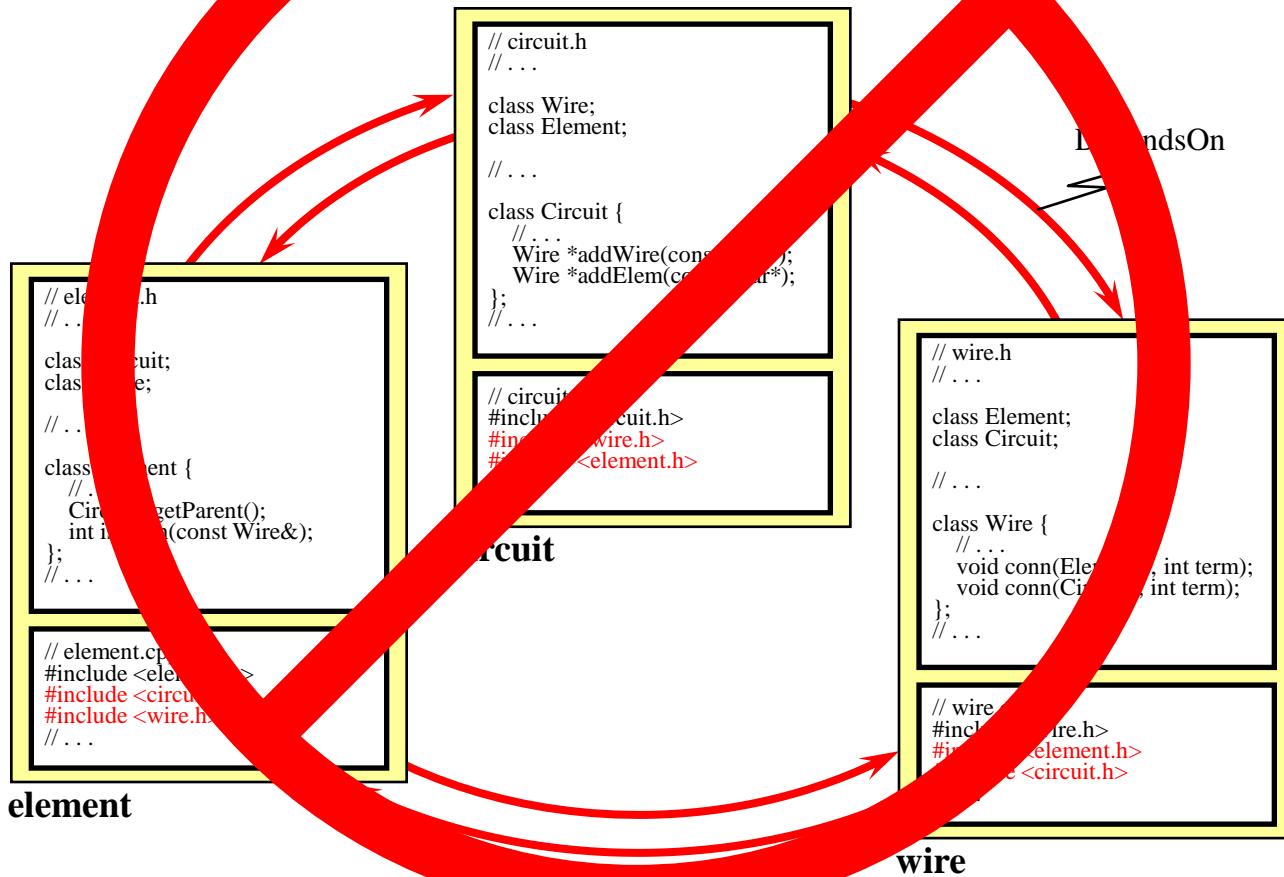
No Cyclic Physical Dependencies



0. Goals

Achieving Fine-Grained Reuse

No Cyclic Physical Dependencies

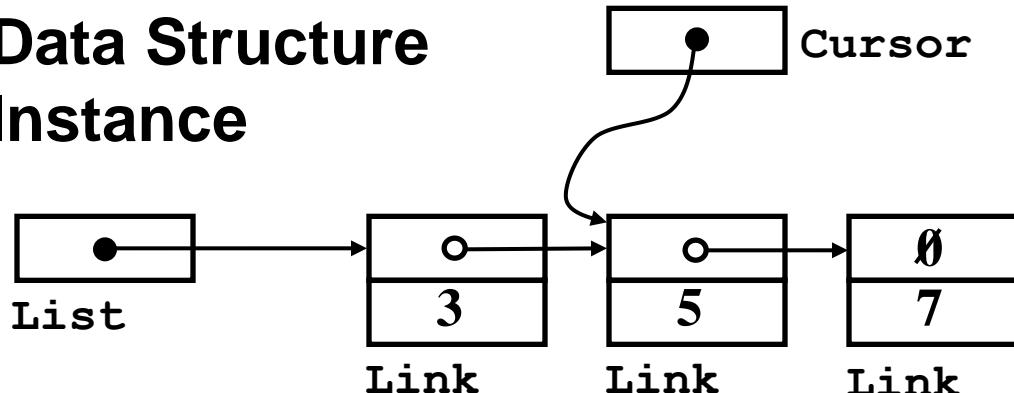


0. Goals

Achieving Fine-Grained Reuse

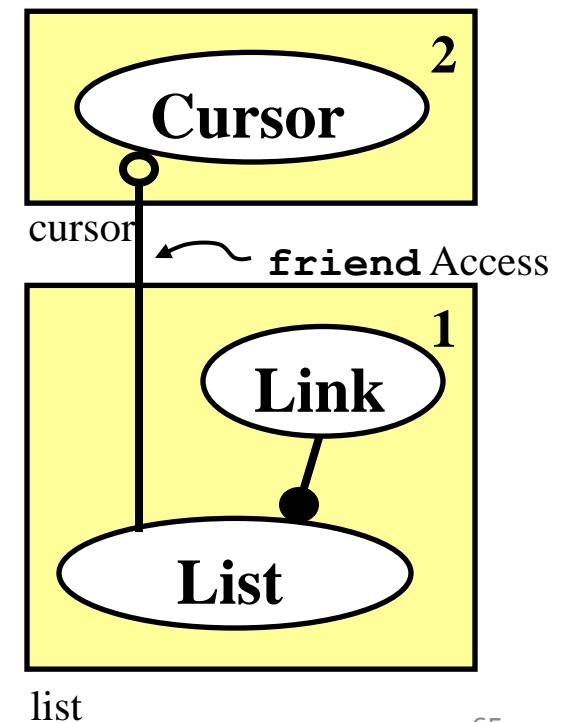
No Private “Back Door” Access

Data Structure Instance



Cursor

Component-Class Diagram

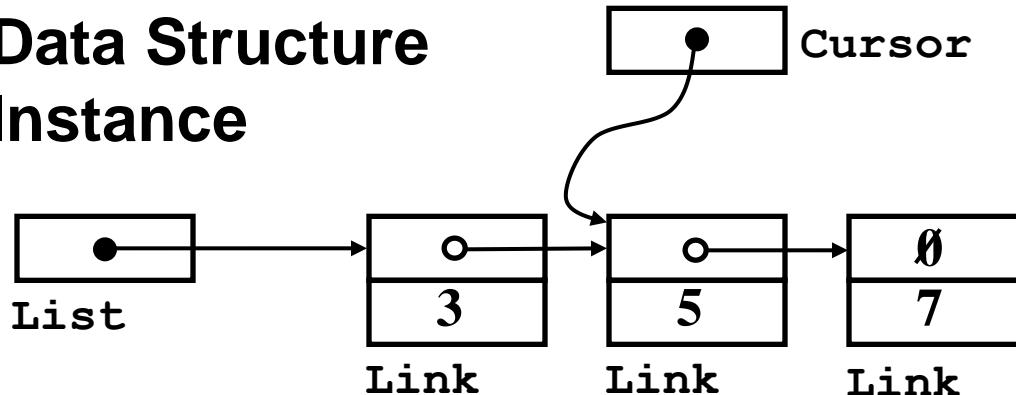


0. Goals

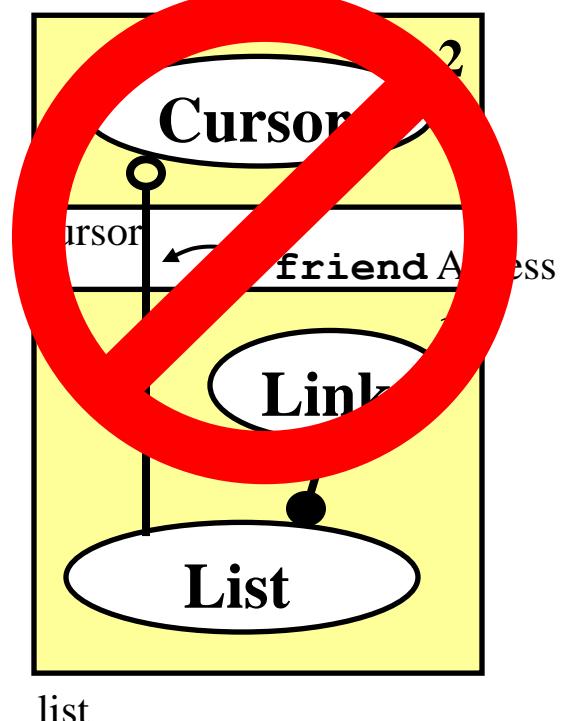
Achieving Fine-Grained Reuse

No Private “Back Door” Access

Data Structure Instance



Component-Class Diagram

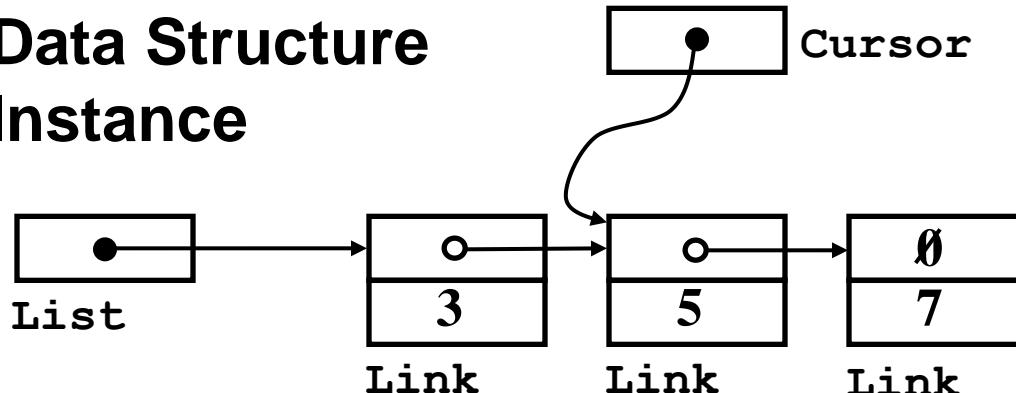


0. Goals

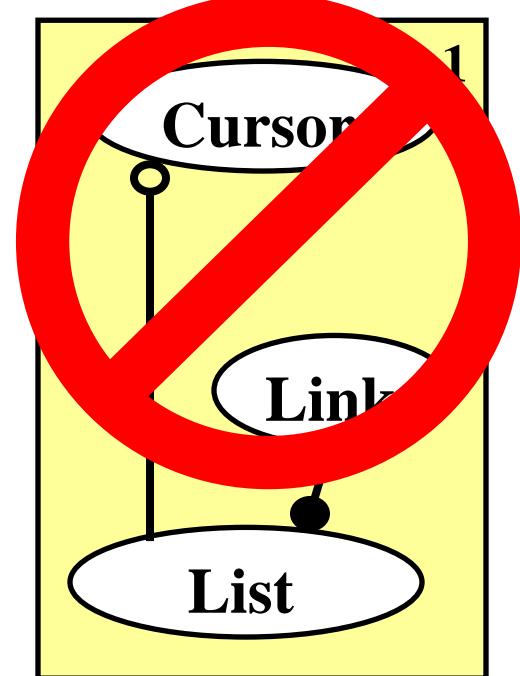
Achieving Fine-Grained Reuse

No Private “Back Door” Access

Data Structure Instance



Component-Class Diagram



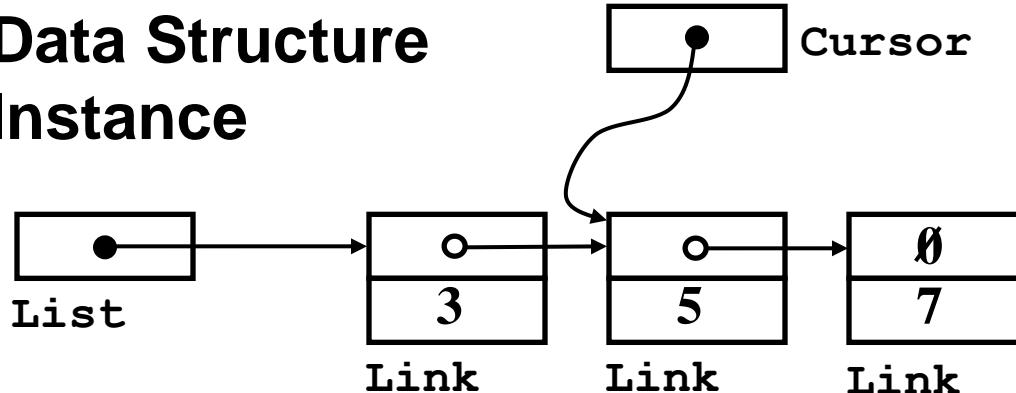
list

0. Goals

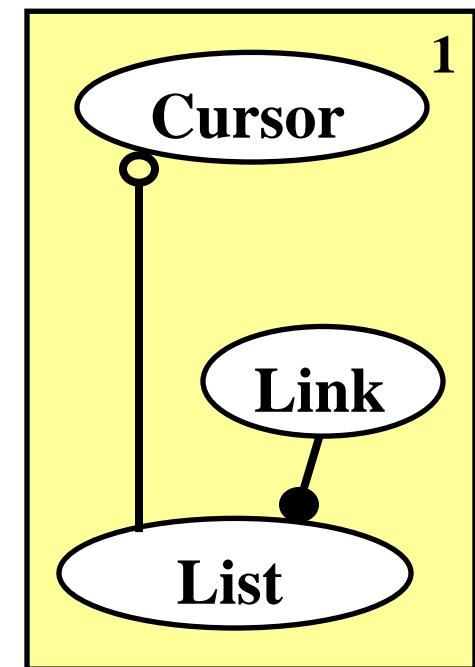
Achieving Fine-Grained Reuse

No Private “Back Door” Access

Data Structure Instance



Component-Class Diagram



0. Goals

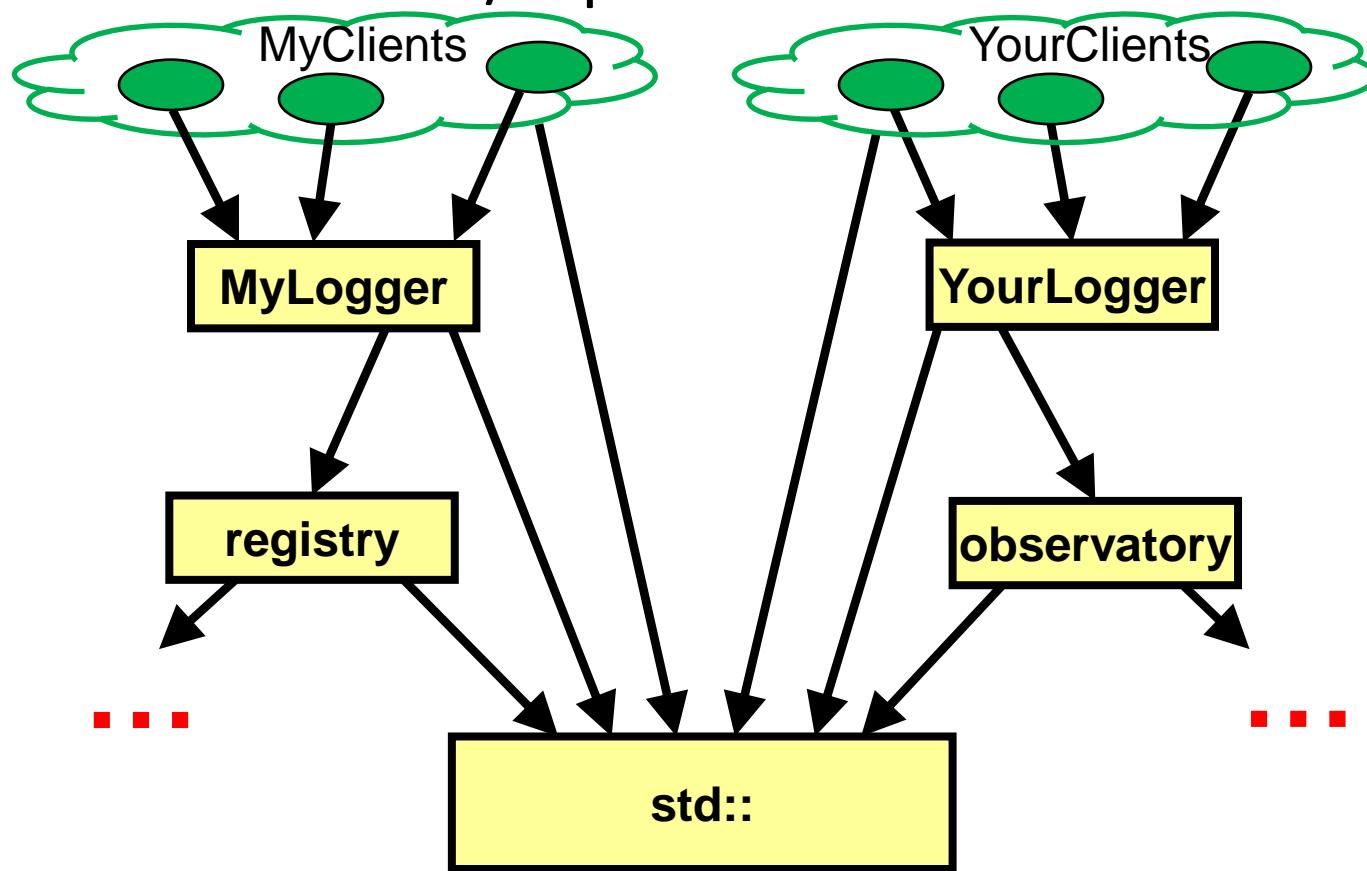
Achieving Hierarchical Reuse

Conventional Reuse: Only the architecturally significant pieces are accessible/exposed.

0. Goals

Achieving Hierarchical Reuse

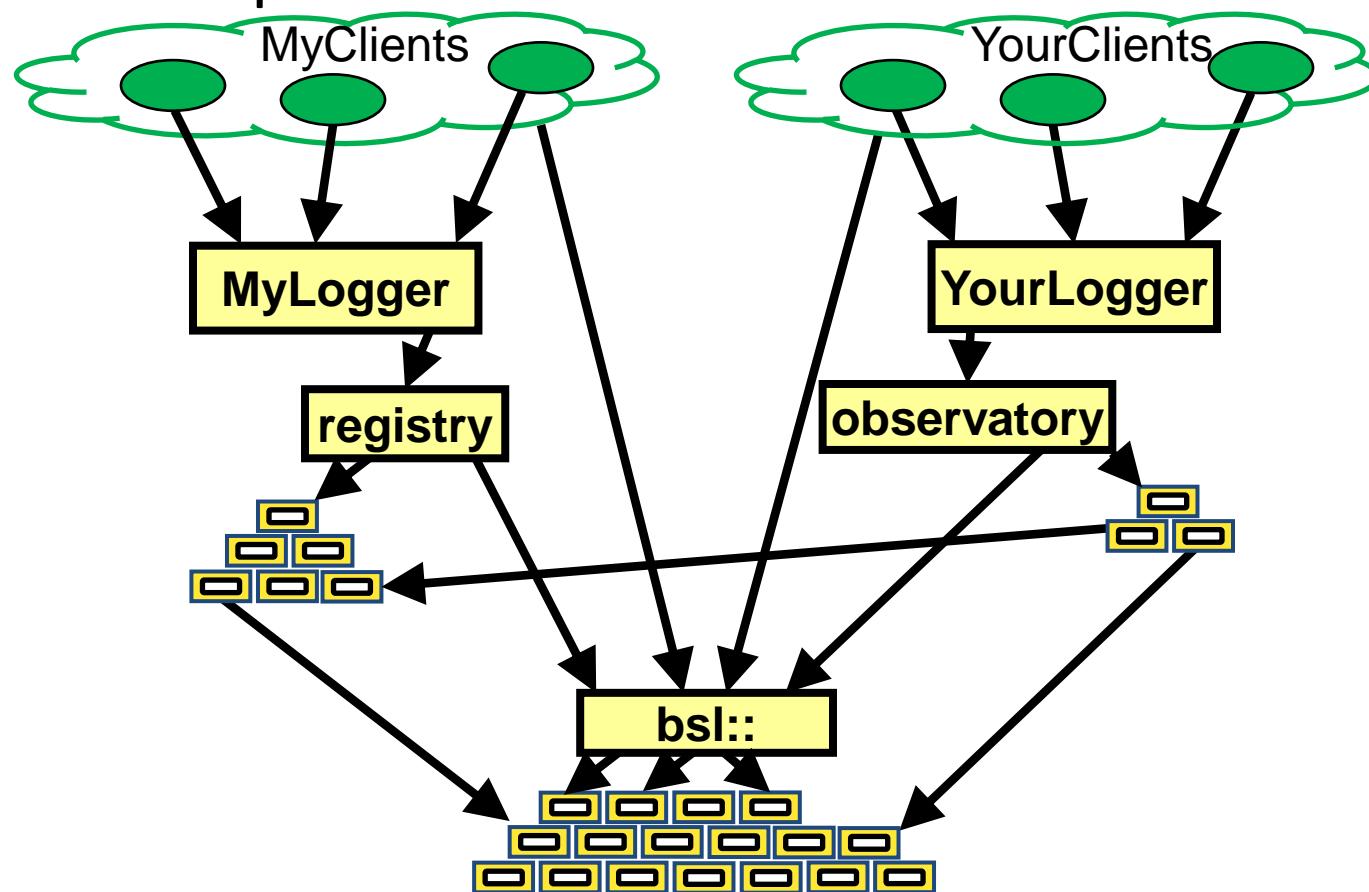
Conventional Reuse: Only the architecturally significant pieces are accessible/exposed.



0. Goals

Achieving Hierarchical Reuse

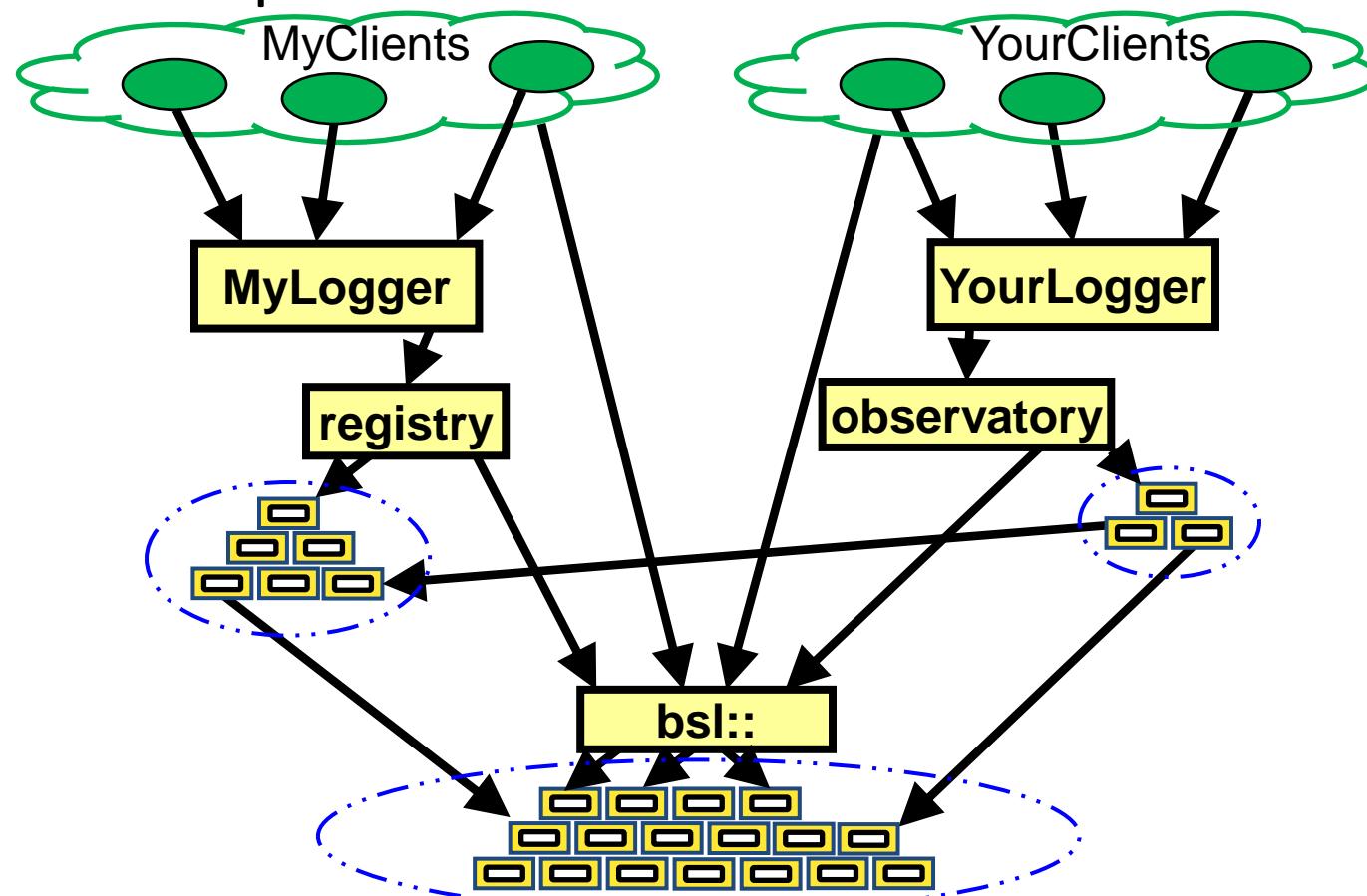
Hierarchical Reuse: Even the (stable) intermediate pieces are exposed for reuse.



0. Goals

Achieving Hierarchical Reuse

Hierarchical Reuse: Even the (stable) intermediate pieces are exposed for reuse.



0. Goals

Achieving Hierarchical Reuse

Application Software:

Library Software:

Solutions

Sub-Solutions

Sub-Sub-Solutions

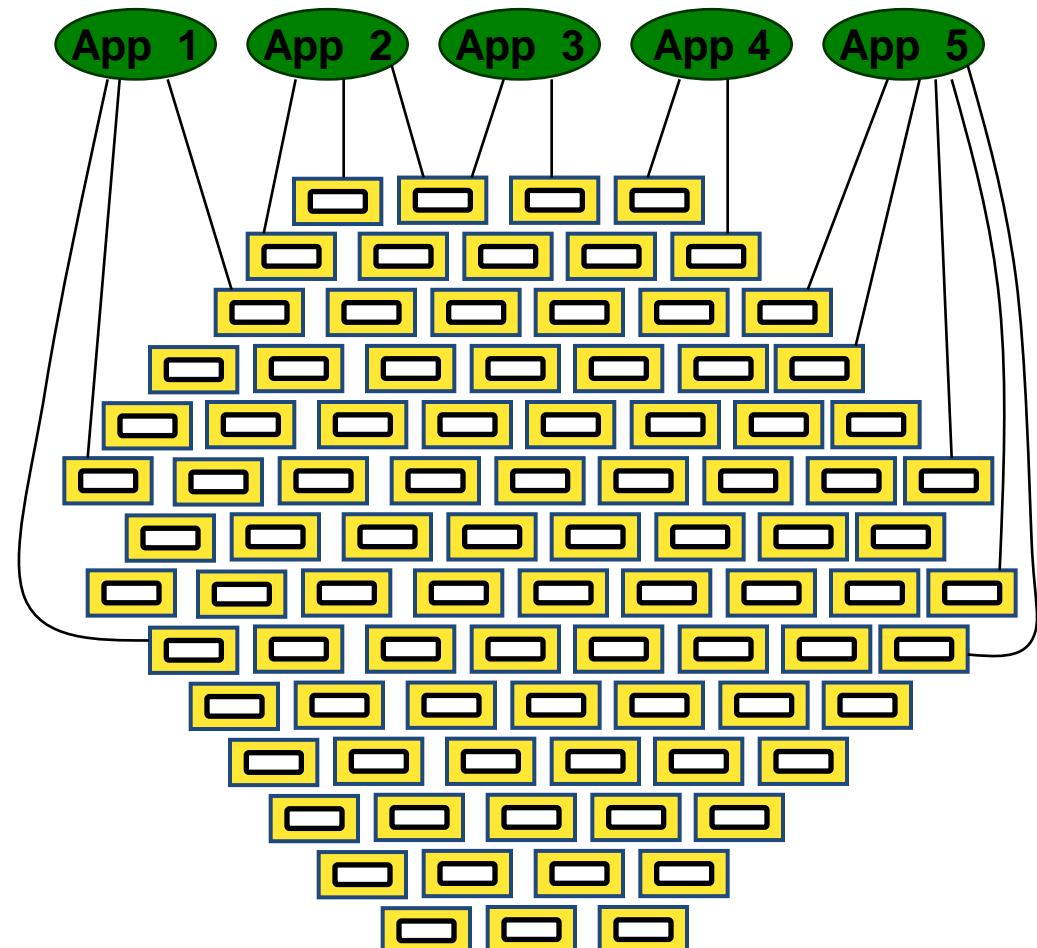
Vocabulary-Type Utilities

Vocabulary Types

Implementation Utilities

Low-Level Interfaces

Platform Adapters



0. Goals

Achieving Wide-Spread Reuse

As library developers, we must

0. Goals

Achieving Wide-Spread Reuse

As library developers, we must:

- Draw complexity inward; push simplicity outward.

0. Goals

Achieving Wide-Spread Reuse

As library developers, we must:

- Draw complexity inward; push simplicity outward.
- Provide correct, complete, yet concise function contract documentation.

0. Goals

Achieving Wide-Spread Reuse

As library developers, we must:

- Draw complexity inward; push simplicity outward.
- Provide correct, complete, yet concise function contract documentation.
- Avoid gratuitous variation in rendering.

0. Goals

Achieving Wide-Spread Reuse

As library developers, we must:

- Draw complexity inward; push simplicity outward.
- Provide correct, complete, yet concise function contract documentation.
- Avoid gratuitous variation in rendering.
- Achieve reliability *at least* as good as our compilers.

0. Goals

Achieving Wide-Spread Reuse

To the maximum extent practicable...

...every software component we write must be:

1. Easy to Understand
2. Easy to Use
3. High Performance
4. Portable
5. Reliable

0. Goals

Achieving Wide-Spread Reuse

1. Easy to Understand

- Canonical rendering.
- Clear and complete reference documentation.
- Relevant usage examples.

0. Goals

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use
 - Effective usage model.
 - Intuitive interface.
 - Appropriate level of safety.
 - Minimal physical dependencies.

0. Goals

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use
3. High Performance
 - Execution (i.e., wall and CPU) run time.
 - Process (i.e., in-core memory) size.
 - Compile time (or the degree of compile-time coupling).
 - Link time (or the extent of link-time dependency).
 - Executable (i.e., on-disk) code size.

0. Goals

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use
3. High Performance
4. Portable
 - Builds on all supported platforms.
 - Runs on all supported platforms.
 - Produces the same results on all supported platforms.
 - Achieves "reasonable" performance on all supported platforms.

0. Goals

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use
3. High Performance
4. Portable
5. Reliable

0. Goals

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use
3. High Performance
4. Portable
5. Reliable
- No core dumps.

0. Goals

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use
3. High Performance
4. Portable
5. Reliable
 - No core dumps.
 - No memory leaks.

0. Goals

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use
3. High Performance
4. Portable
5. Reliable
 - No core dumps.
 - No memory leaks.
 - No incorrect results.

0. Goals

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use
3. High Performance
4. Portable
5. Reliable
 - No core dumps.
 - No memory leaks.
 - No incorrect results.
 - No bugs!

0. Goals

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use
3. High Performance
4. Portable
5. Reliable
 - No core dumps.
 - No memory leaks.
 - No incorrect results.
 - No bugs!
 - No, we're not kidding.

0. Goals

Achieving Wide-Spread Reuse

Wait a minute...

Just how good does
software need to be?

0. Goals

Achieving Wide-Spread Reuse

Writing an *application* is somewhat analogous to building a house:

0. Goals

Achieving Wide-Spread Reuse

Writing an *application* is somewhat analogous to building a house:

- It must adequately perform its function.

0. Goals

Achieving Wide-Spread Reuse



© 2006 <http://philip.greenspun.com/copyright/>

0. Goals

Achieving Wide-Spread Reuse

Writing an *application* is somewhat analogous to building a house:

- It must adequately perform its function.
- It must be safe under normal conditions.

0. Goals

Achieving Wide-Spread Reuse



0. Goals

Achieving Wide-Spread Reuse

Writing an *application* is somewhat analogous to building a house:

- It must adequately perform its function.
- It must be safe under normal conditions.
- Beyond that, there are costs and benefits that have to be weighed.

0. Goals

Achieving Wide-Spread Reuse



0. Goals

Achieving Wide-Spread Reuse



0. Goals

Achieving Wide-Spread Reuse



0. Goals

Achieving Wide-Spread Reuse



0. Goals

Achieving Wide-Spread Reuse

Writing a ***Reusable library*** is different.

0. Goals

Achieving Wide-Spread Reuse

Writing a **Reusable library** is different.

The goal of reusable software is to be *reused* wherever “appropriate” and human beings – not computers – will make that determination.

– Lakos1x

0. Goals

Achieving Wide-Spread Reuse

``We conjecture that the barriers to reuse are not on the producer's side, but on the consumer's side. If a software engineer, a potential consumer of standardized components, perceives it to be more expensive to find a component that meets his needs, and so verify, than to write one anew, a new, duplicative component will be written. Notice that we said *perceives* above. It does not matter what the true cost of reconstruction is.''

—Van Snyder (Brooks95)

0. Goals

Achieving Wide-Spread Reuse

``We conjecture that the barriers to reuse are not on the producer's side, but on the consumer's side. If a software engineer, a potential consumer of standardized components, perceives it to be more expensive to find a component that meets his needs, and so verify, than to write one anew, a new, duplicative component will be written. Notice that we said *perceives* above. It does not matter what the true cost of reconstruction is.''

—Van Snyder (Brooks95)

0. Goals

Achieving Wide-Spread Reuse

``We conjecture that the barriers to reuse are not on the producer's side, but on the consumer's side. If a software engineer, a potential consumer of standardized components, perceives it to be more expensive to find a component that meets his needs, and so verify, than to write one anew, a new, duplicative component will be written. Notice that we said *perceives* above. It does not matter what the true cost of reconstruction is.''

—Van Snyder (Brooks95)

0. Goals

Achieving Wide-Spread Reuse

``We conjecture that the barriers to reuse are not on the producer's side, but on the consumer's side. If a software engineer, a potential consumer of standardized components, perceives it to be more expensive to find a component that meets his needs, and so verify, than to write one anew, a new, duplicative component will be written. Notice that we said *perceives* above. It does not matter what the true cost of reconstruction is.''

—Van Snyder (Brooks95)

0. Goals

Achieving Wide-Spread Reuse

Reusable library software:

0. Goals

Achieving Wide-Spread Reuse

Reusable library software:

- ❑ Must be perceived as *far* better than what a prospective client (or anyone else) could do in any practical time frame.

0. Goals

Achieving Wide-Spread Reuse

Reusable library software:

- ❑ Must be perceived as *far* better than what a prospective client (or anyone else) could do in any practical time frame.
- ❑ Unlike a house (or an App), can be consumed by many different (kinds of) clients.

0. Goals

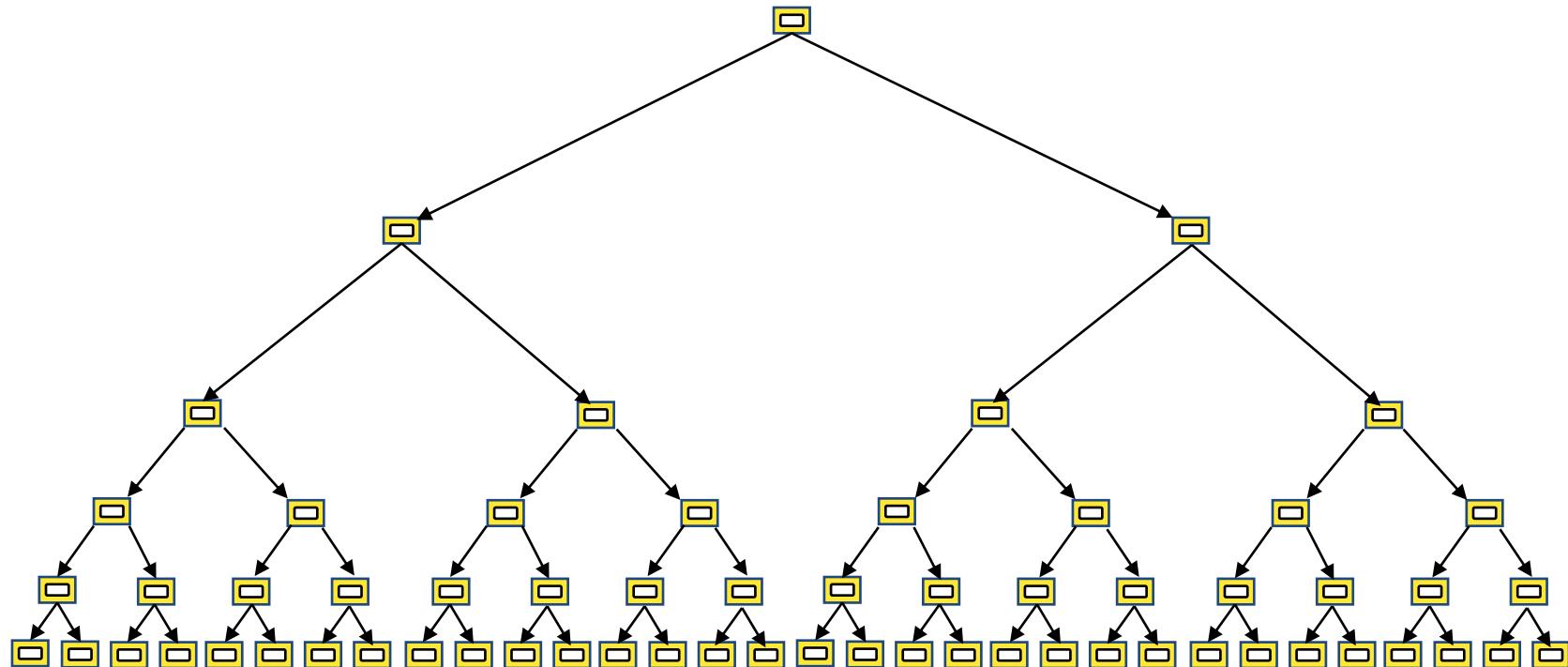
Achieving Wide-Spread Reuse

Reusable library software:

- ❑ Must be perceived as *far* better than what a prospective client (or anyone else) could do in any practical time frame.
- ❑ Unlike a house (or an App), can be consumed by many different (kinds of) clients.
- ❑ The more clients, the greater the utility (and vice versa).

0. Goals

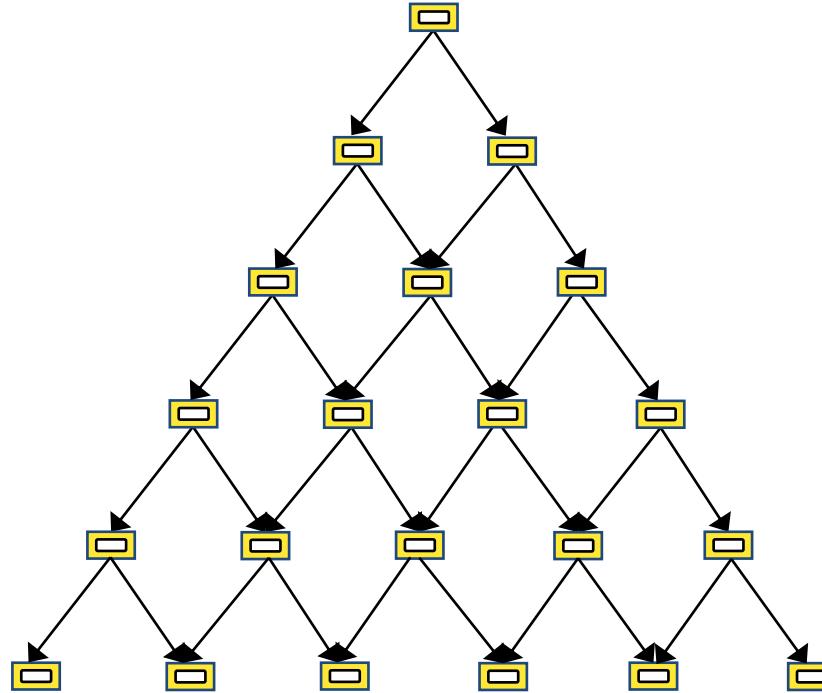
Achieving Wide-Spread Reuse



No Re-convergence

0. Goals

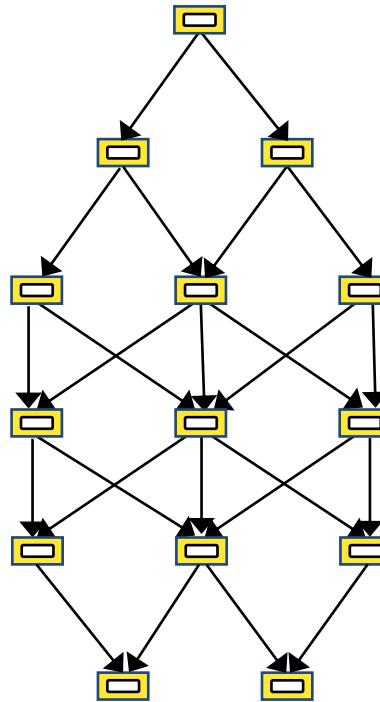
Achieving Wide-Spread Reuse



Significant Re-convergence

0. Goals

Achieving Wide-Spread Reuse



Maximal Re-convergence

0. Goals

Achieving Wide-Spread Reuse

So how good does
our *reusable library*
software need to be?

0. Goals

Achieving Wide-Spread Reuse



0. Goals

Achieving Wide-Spread Reuse

Nothing
Succeeds Like
Excess!

Achieving Wide-Spread Reuse

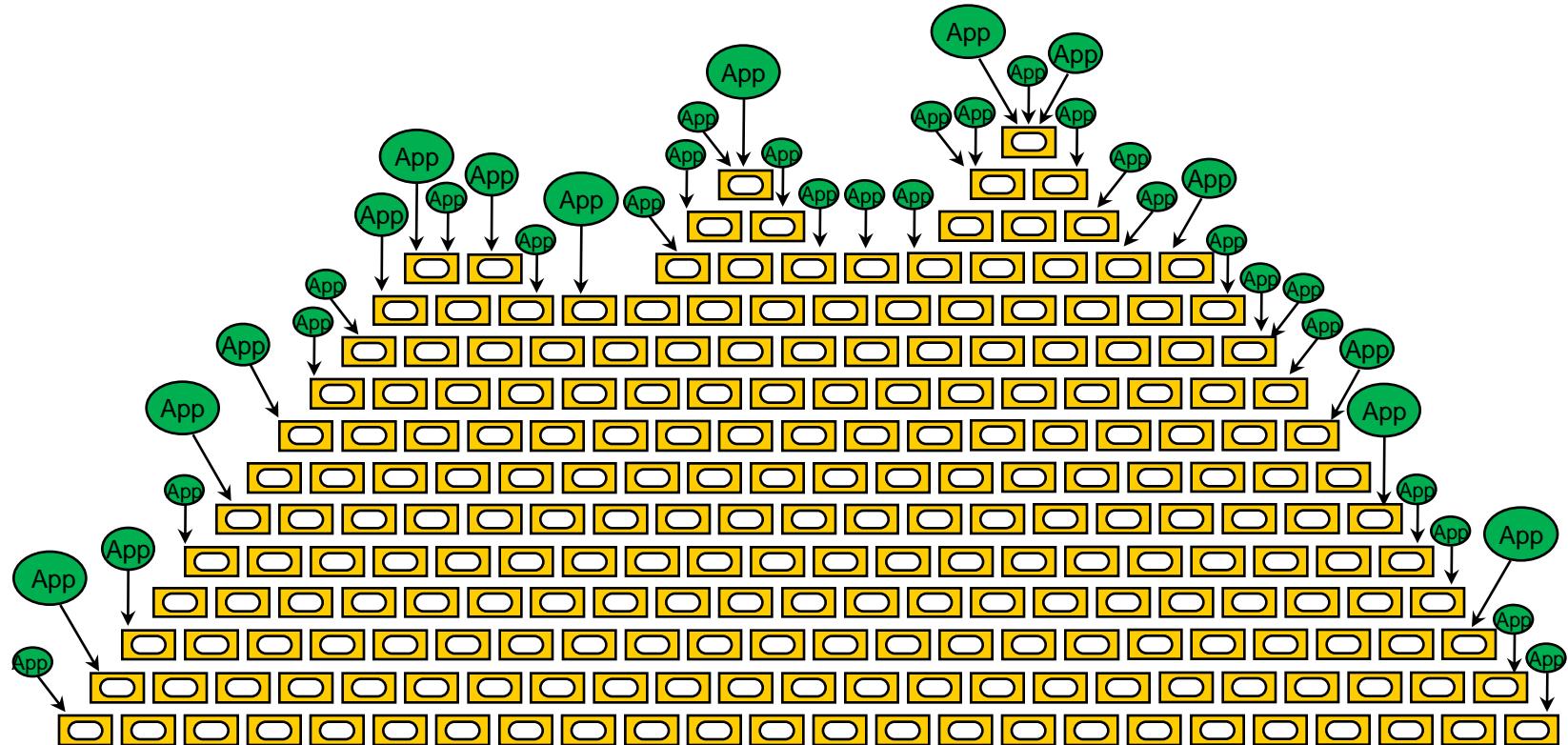
Nothing
Succeeds Like
Excess!

(If it's worth doing, it's worth overdoing.)

0. Goals

Achieving Wide-Spread Reuse

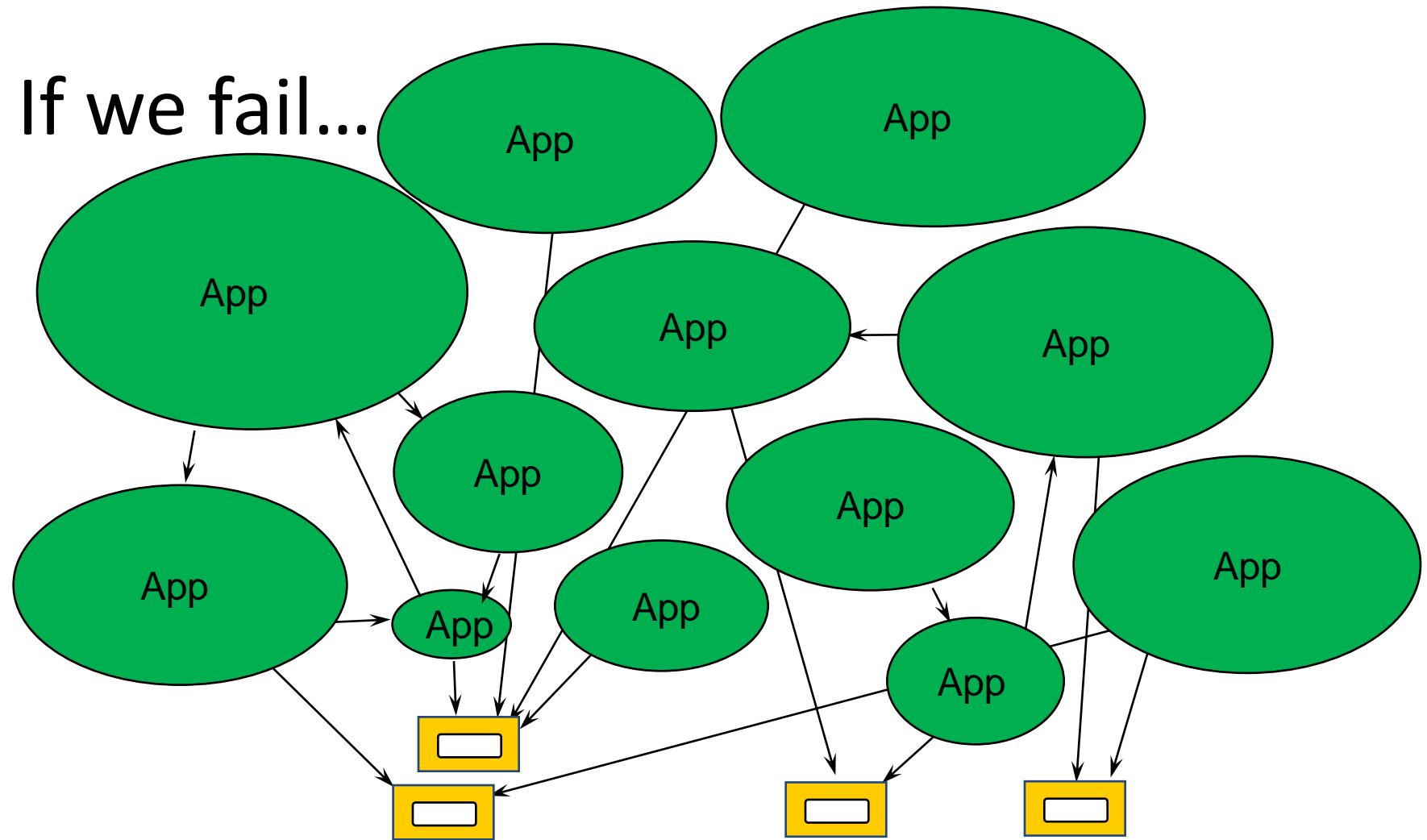
If we succeed...



0. Goals

Achieving Wide-Spread Reuse

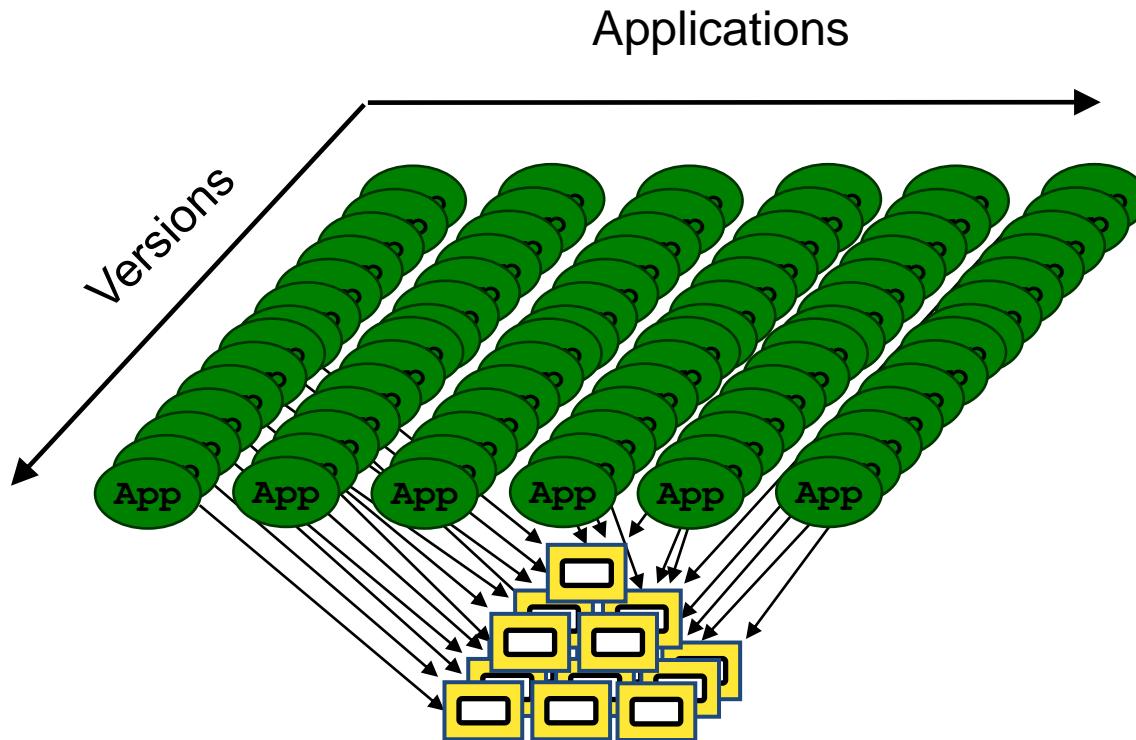
If we fail...



0. Goals

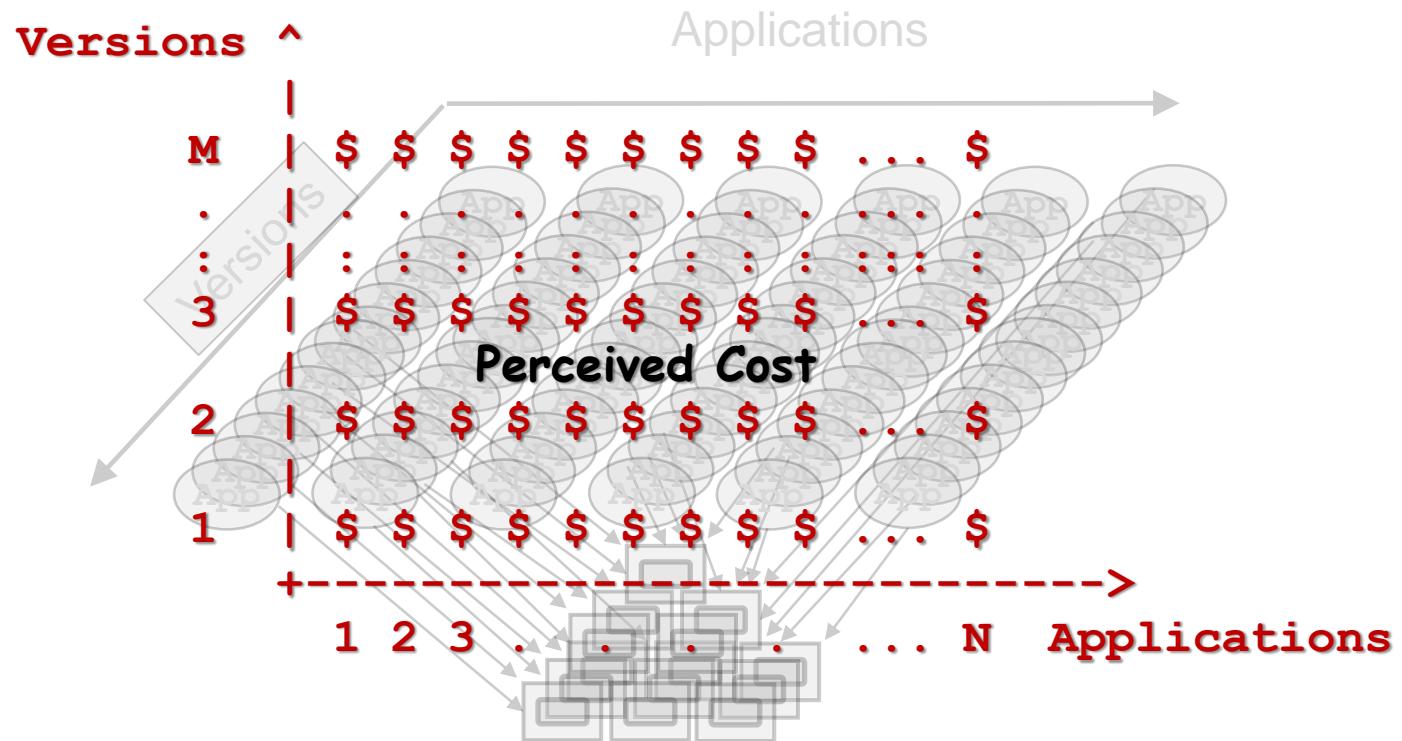
Achieving Wide-Spread Reuse

Fortunately, we can amortize the *perceived cost* over many **products X versions**:



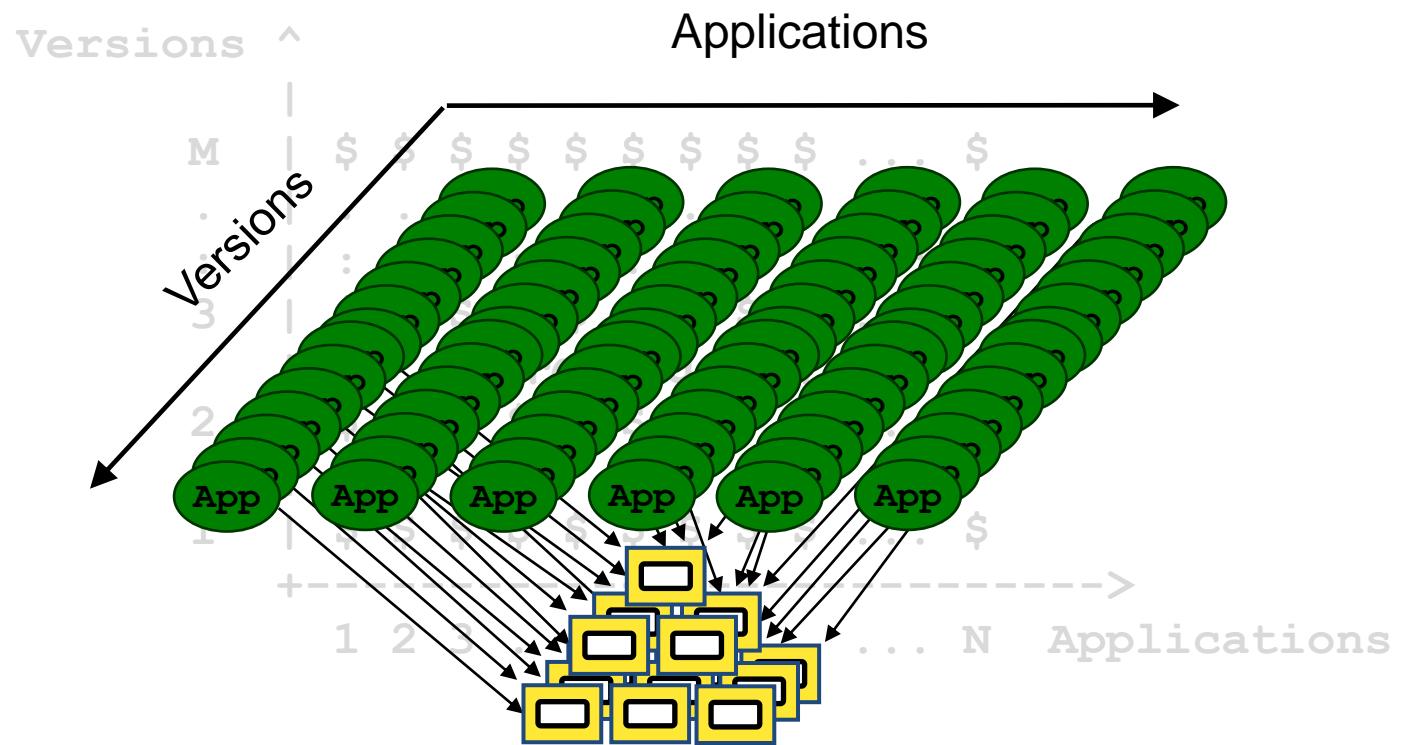
0. Goals

Achieving Wide-Spread Reuse



0. Goals

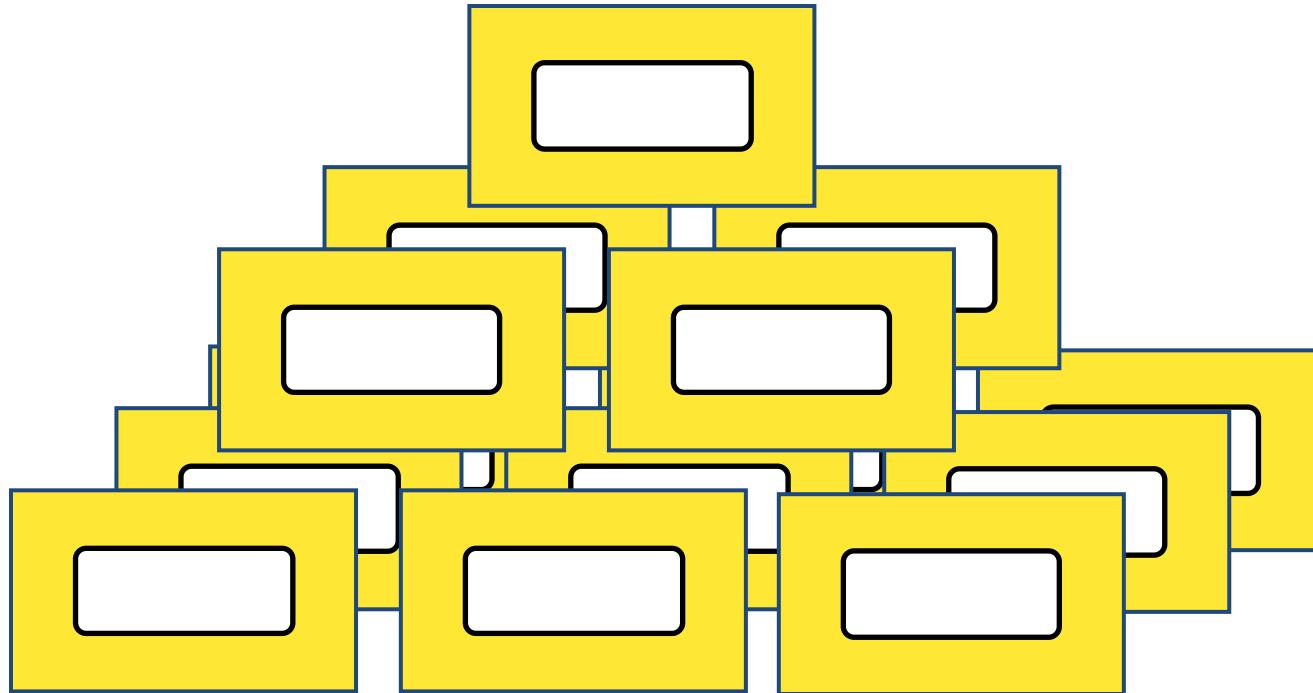
Achieving Wide-Spread Reuse



0. Goals

Achieving Wide-Spread Reuse

Hierarchically Reusable Software



0. Goals

Achieving Wide-Spread Reuse

Hierarchically Reusable Software

a.k.a.:



Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment

Rendered as Fine-Grained Hierarchically Reusable Components.

Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment

Rendered as Fine-Grained *Hierarchically Reusable* Components.

1. Process & Architecture

Introduction

All of the software we write is governed
by a common overarching set of
Organizing Principles.

1. Process & Architecture

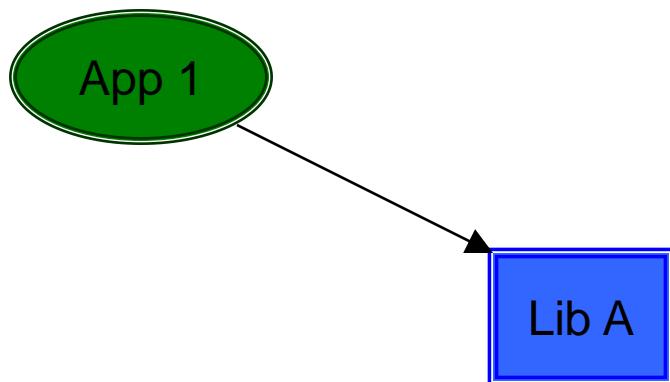
Introduction

All of the software we write is governed by a common overarching set of *Organizing Principles.*

Among the most central of which is achieving *Sound Physical Design.*

1. Process & Architecture

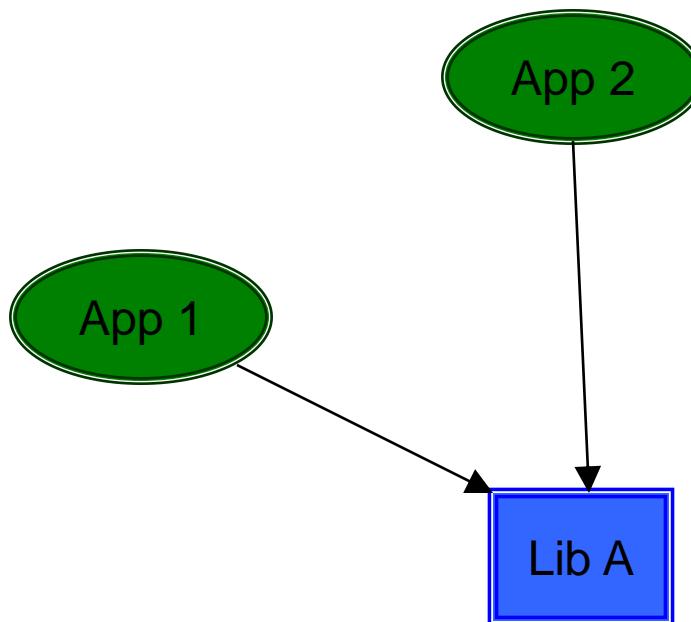
Creating a Big Ball of Mud



1. Process & Architecture

Creating a Big Ball of Mud

Where We Put Our Code Matters!

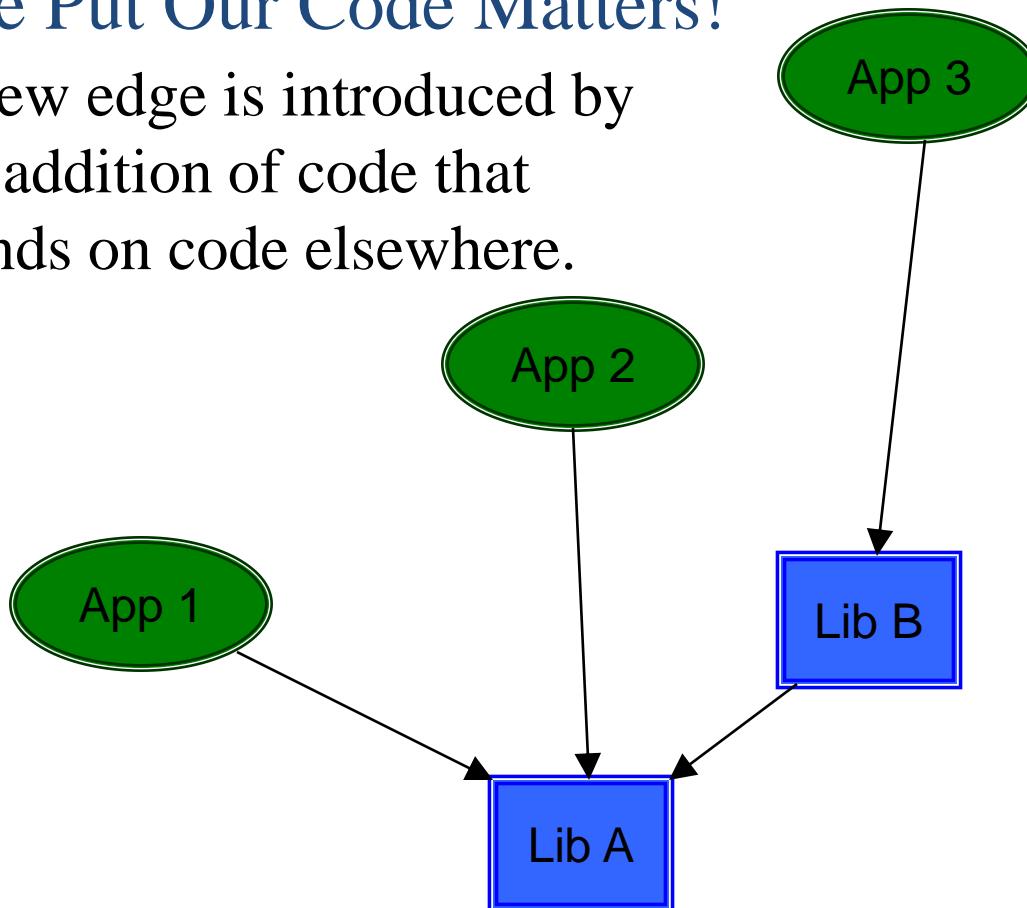


1. Process & Architecture

Creating a Big Ball of Mud

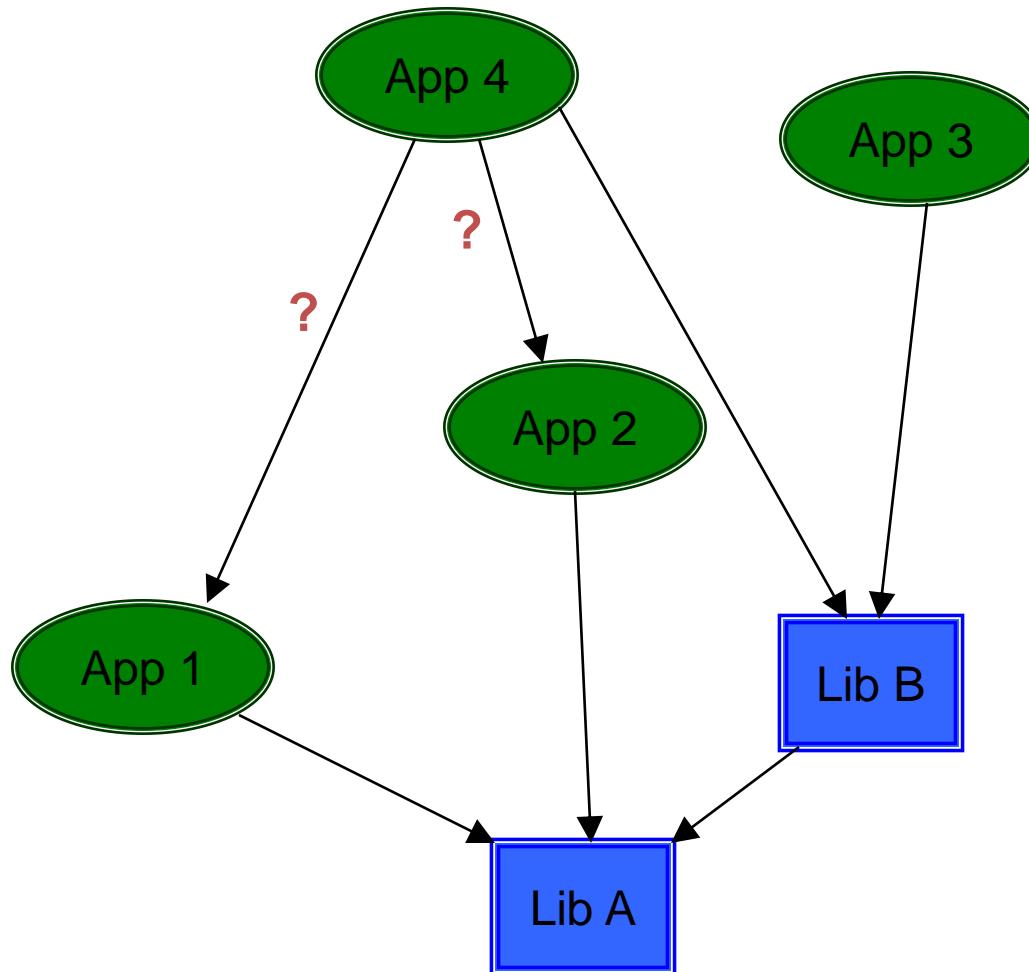
Where We Put Our Code Matters!

Each new edge is introduced by
the addition of code that
depends on code elsewhere.



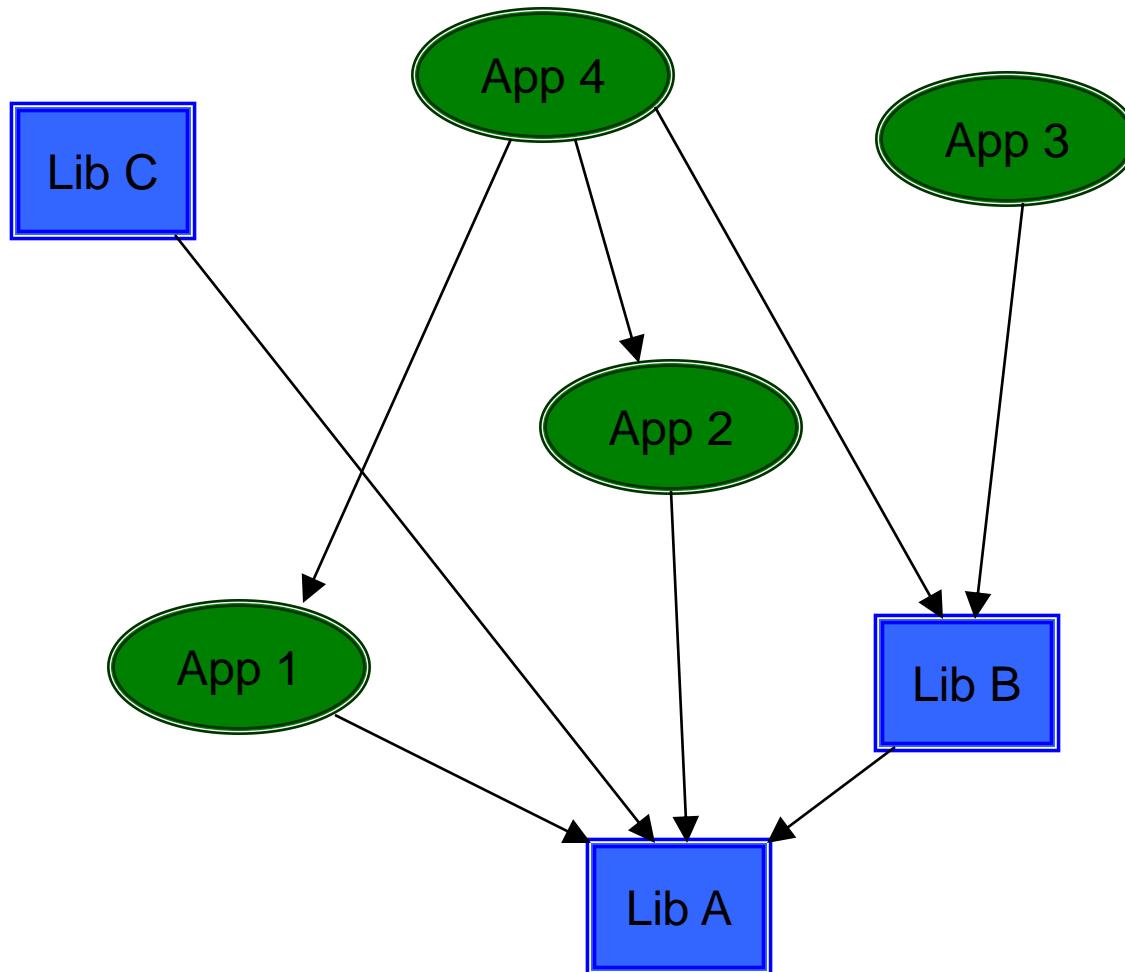
1. Process & Architecture

Creating a Big Ball of Mud



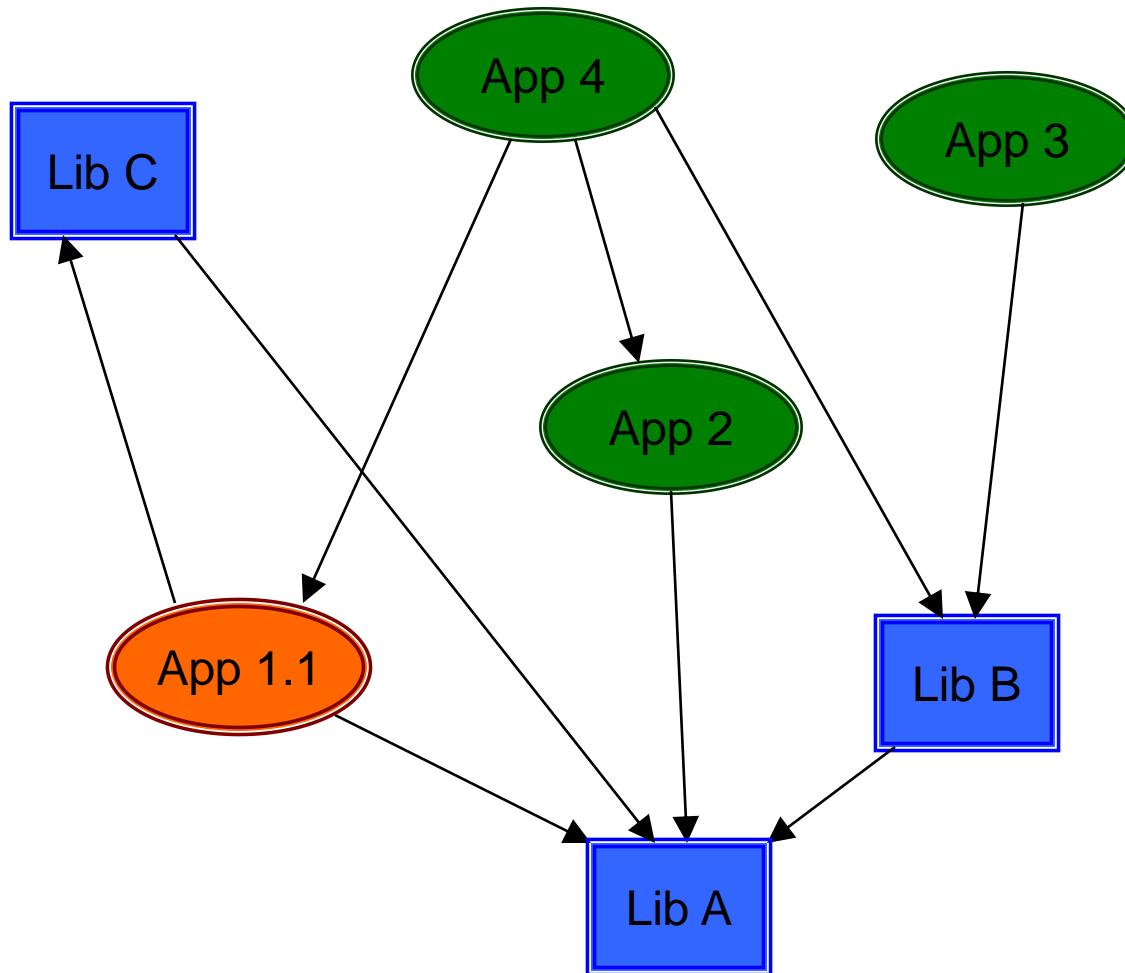
1. Process & Architecture

Creating a Big Ball of Mud



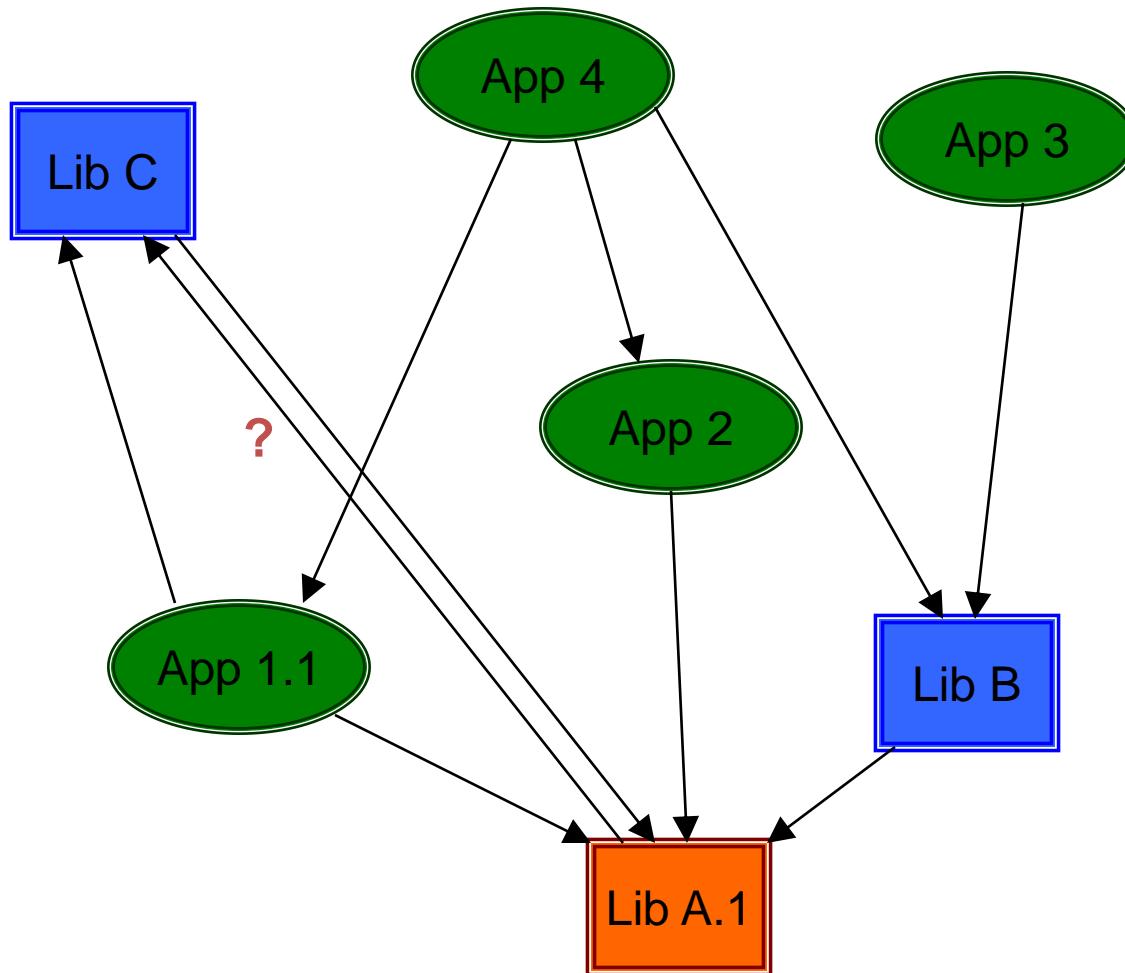
1. Process & Architecture

Creating a Big Ball of Mud



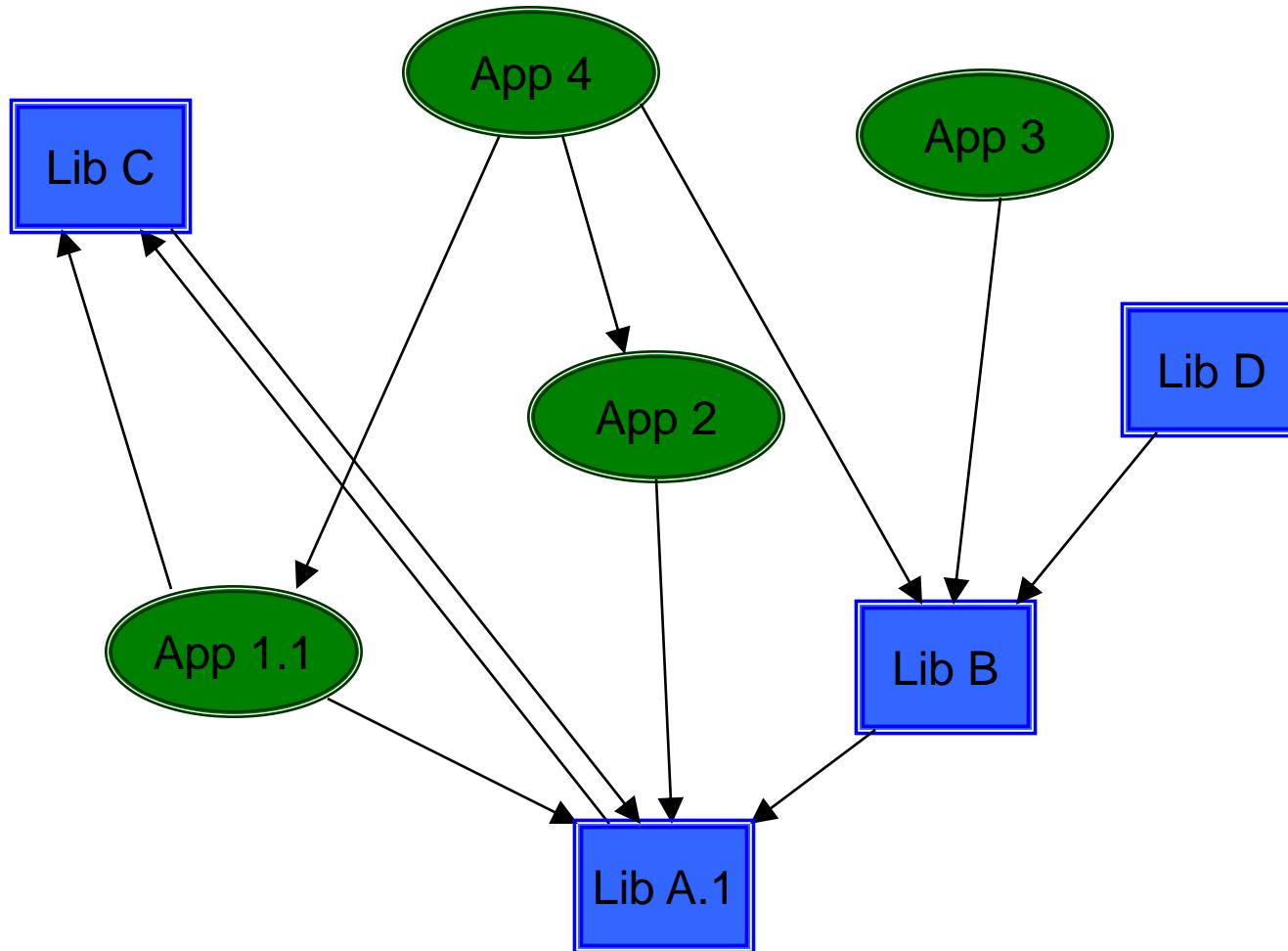
1. Process & Architecture

Creating a Big Ball of Mud



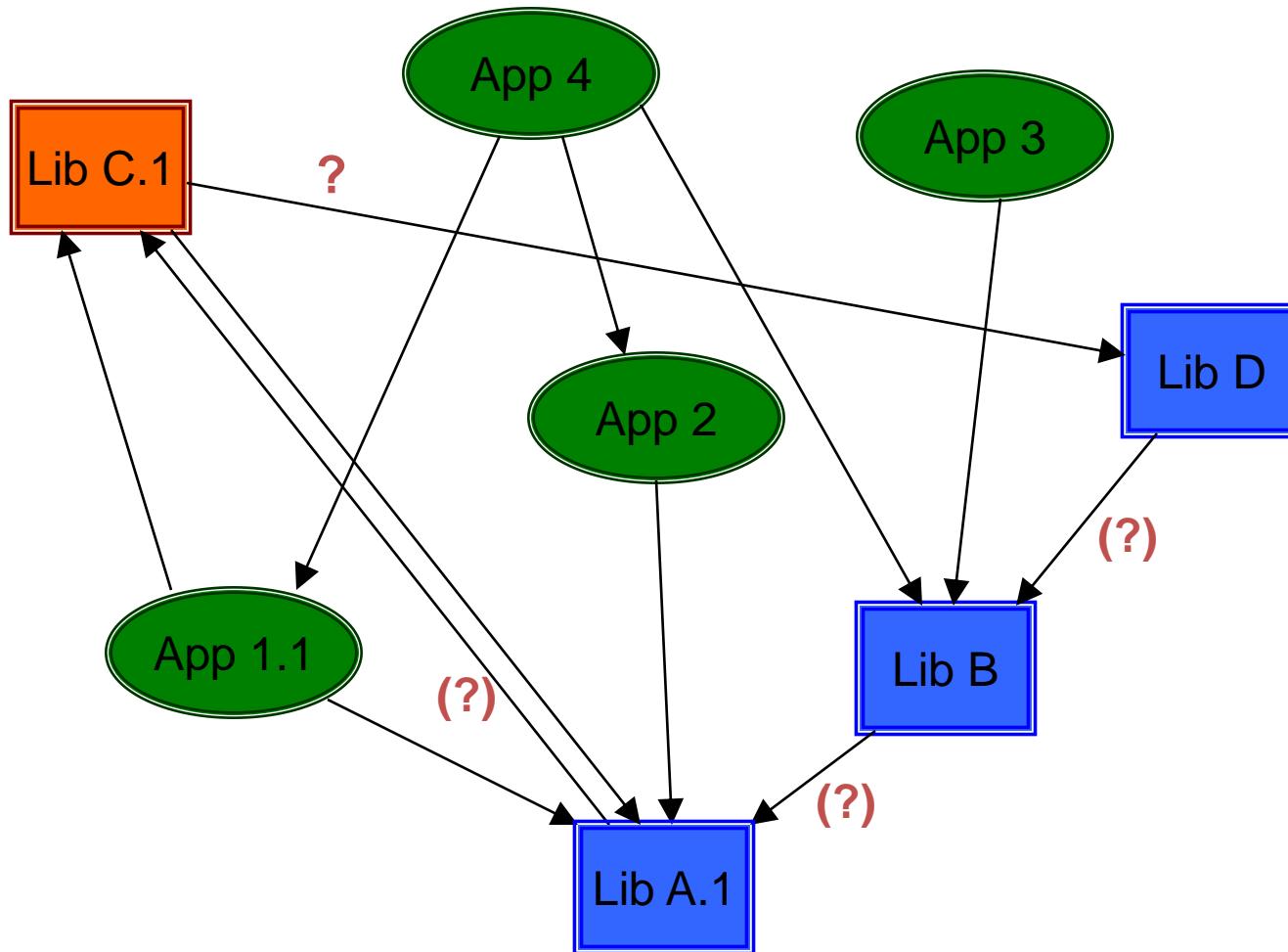
1. Process & Architecture

Creating a Big Ball of Mud



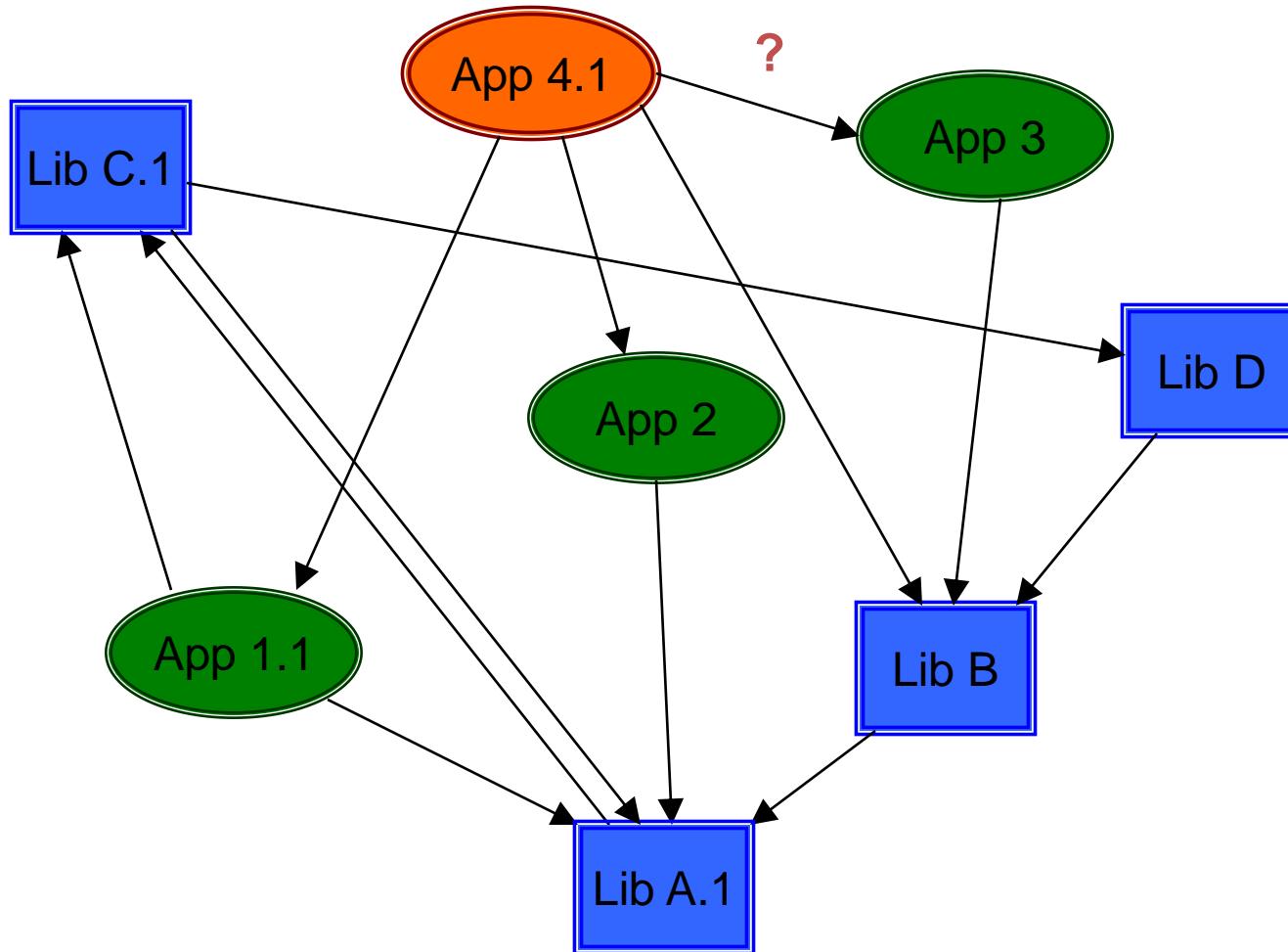
1. Process & Architecture

Creating a Big Ball of Mud



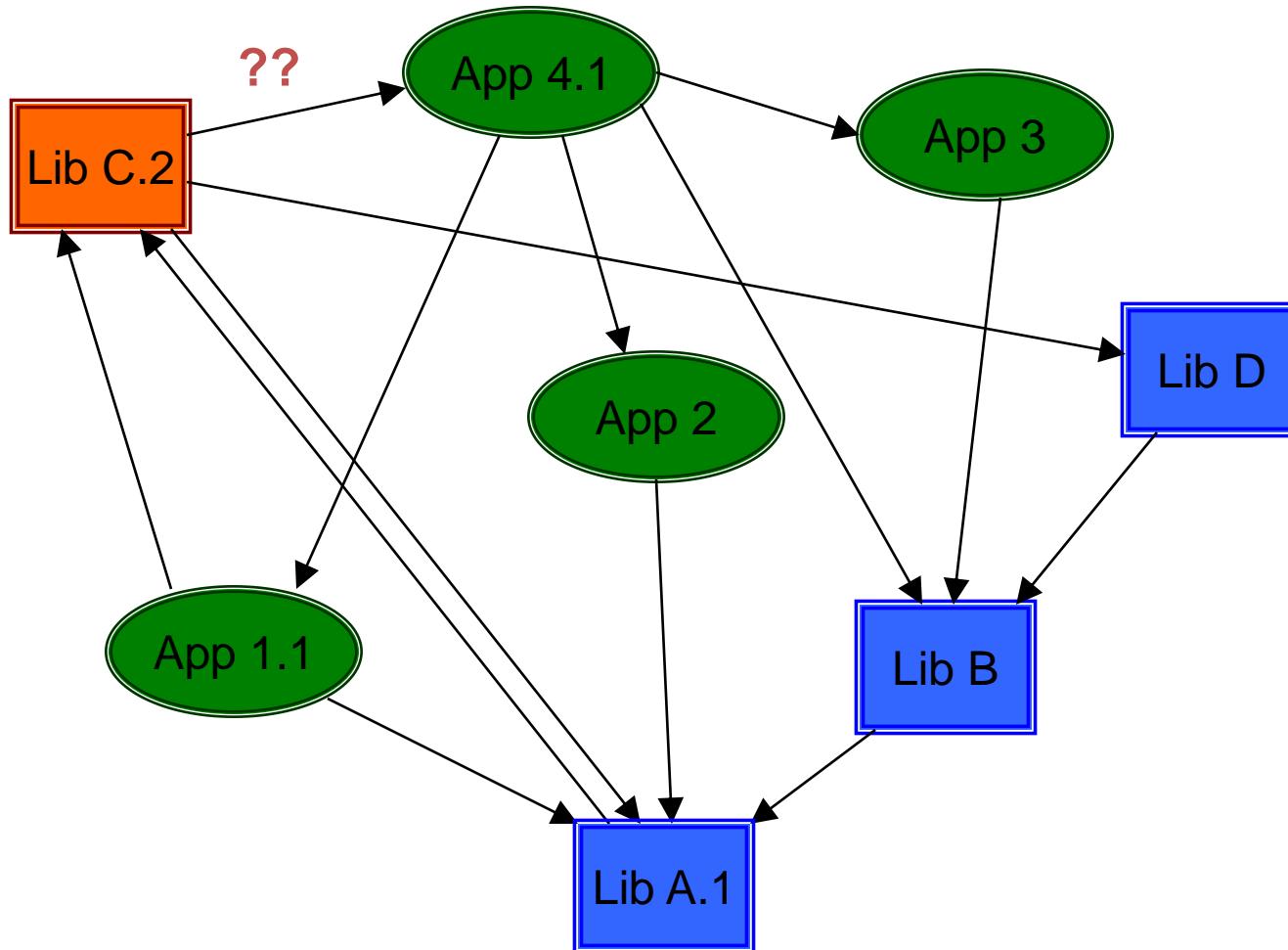
1. Process & Architecture

Creating a Big Ball of Mud



1. Process & Architecture

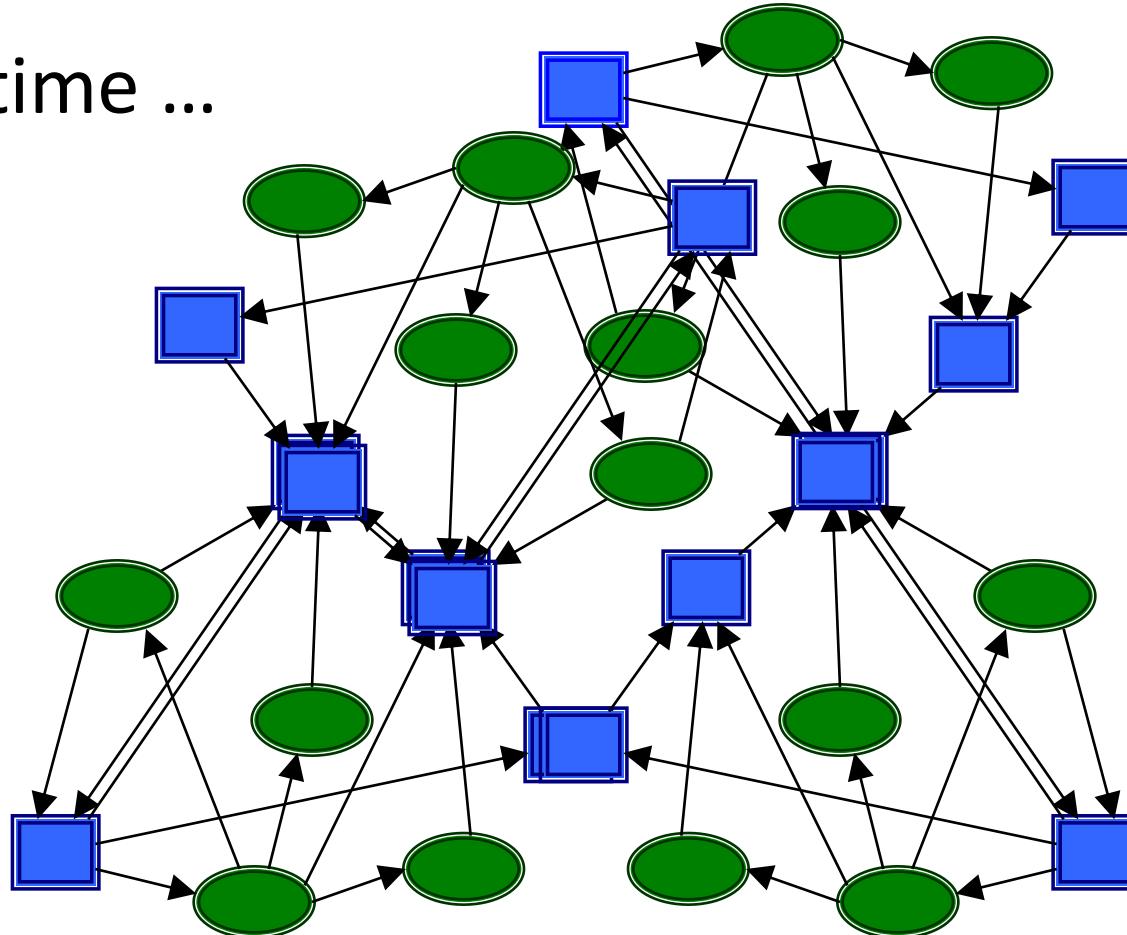
Creating a Big Ball of Mud



1. Process & Architecture

Creating a Big Ball of Mud

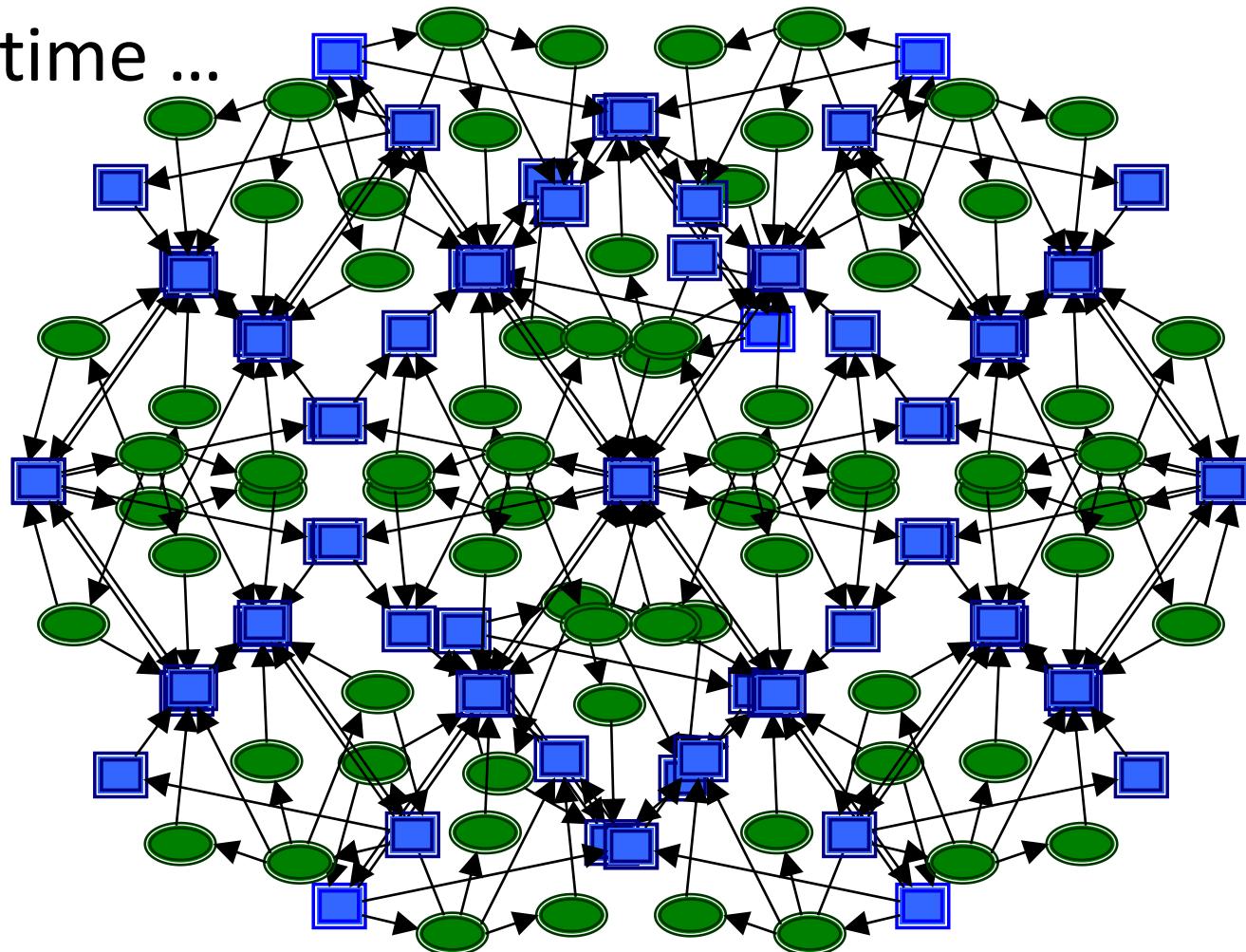
Over time ...



1. Process & Architecture

Creating a Big Ball of Mud

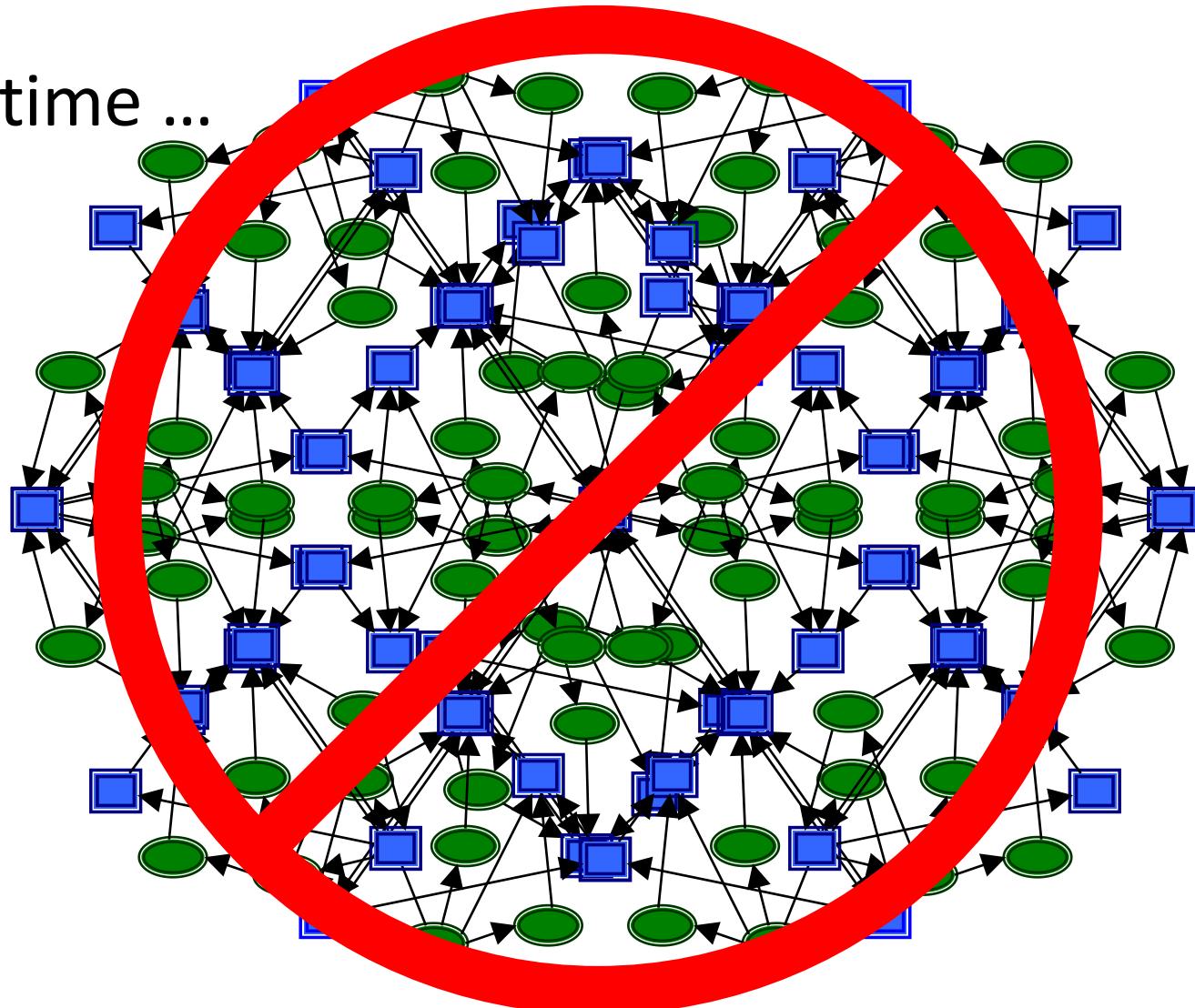
Over time ...



1. Process & Architecture

Creating a Big Ball of Mud

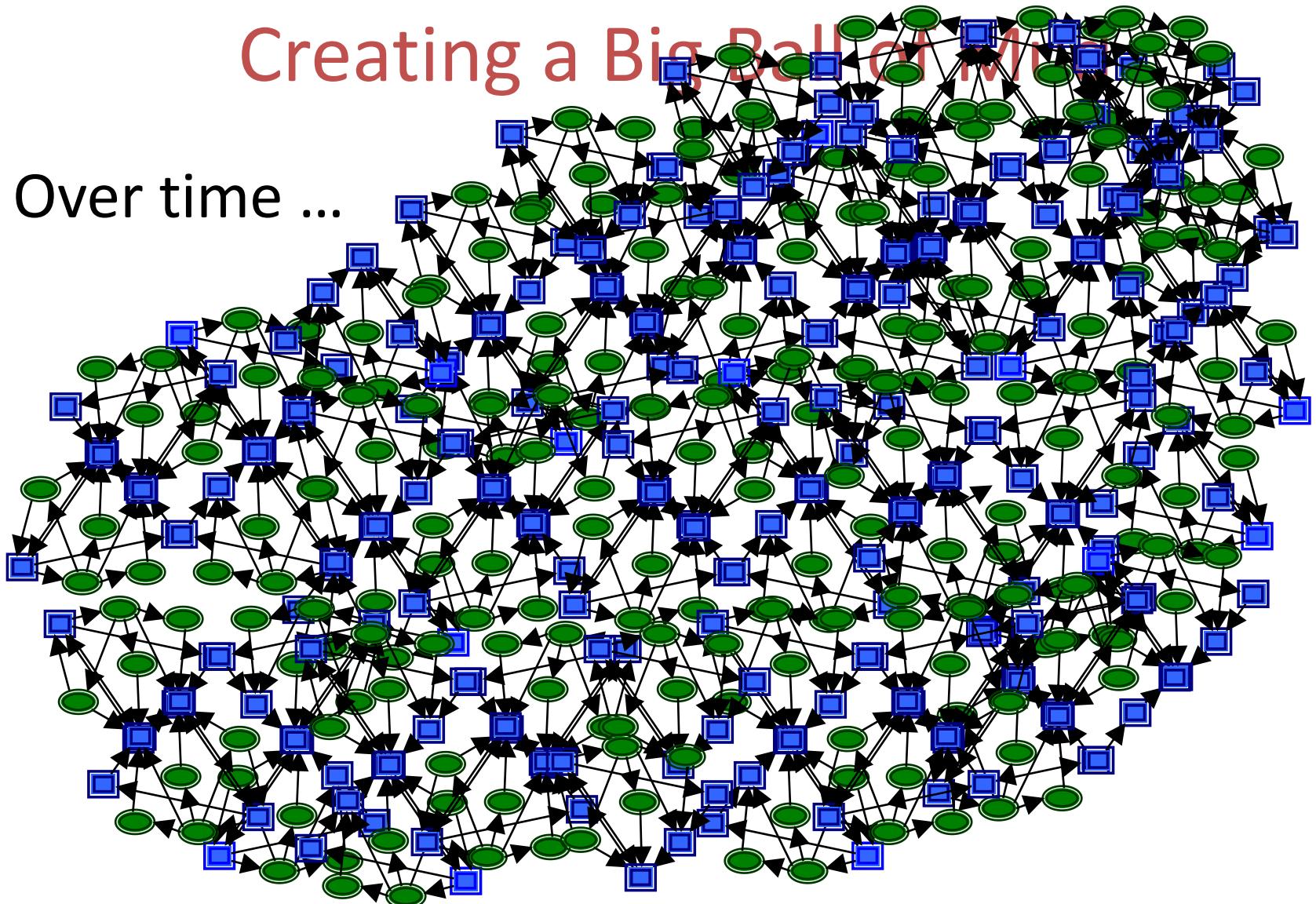
Over time ...



1. Process & Architecture

Creating a Big Ball of Mud

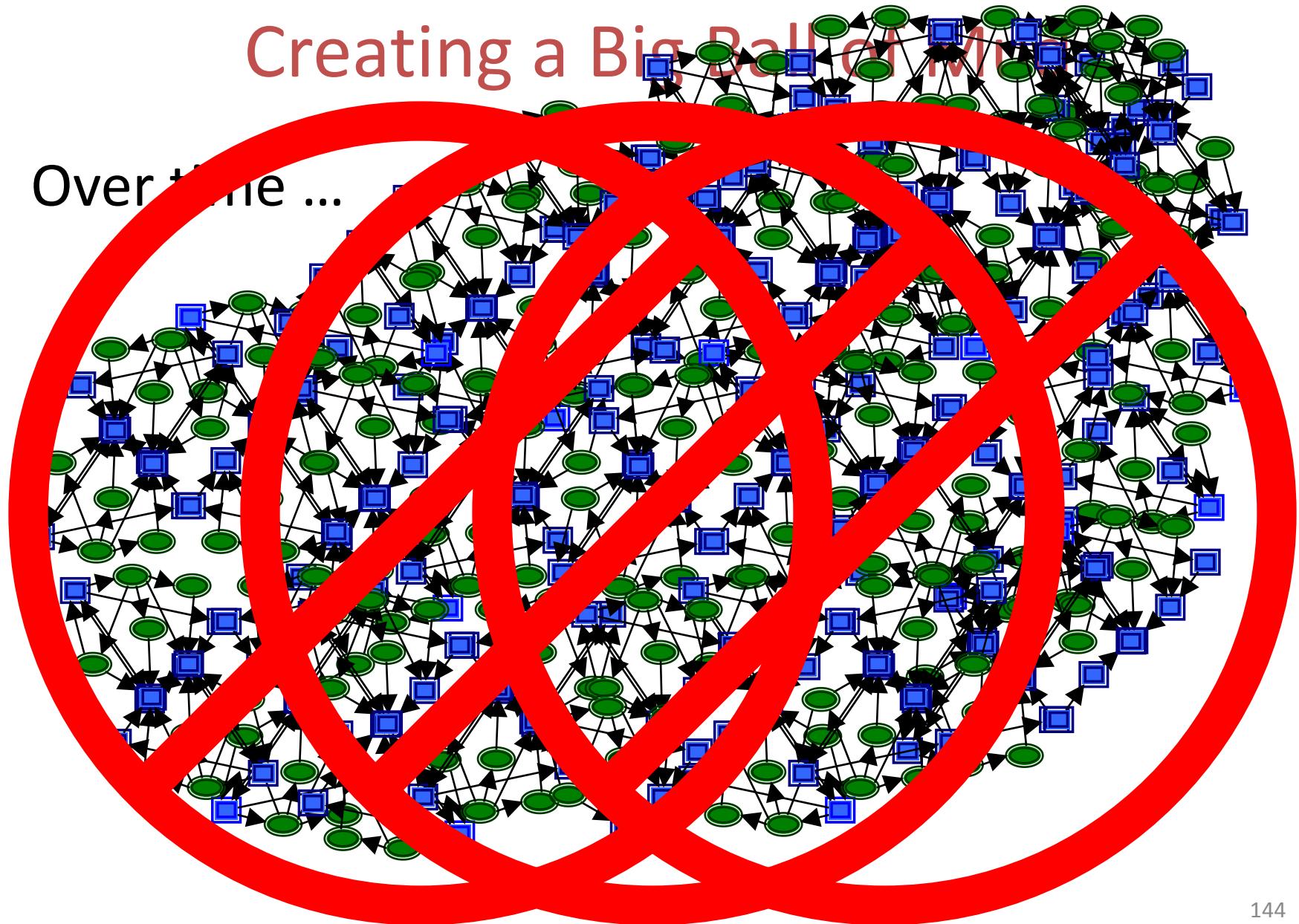
Over time ...

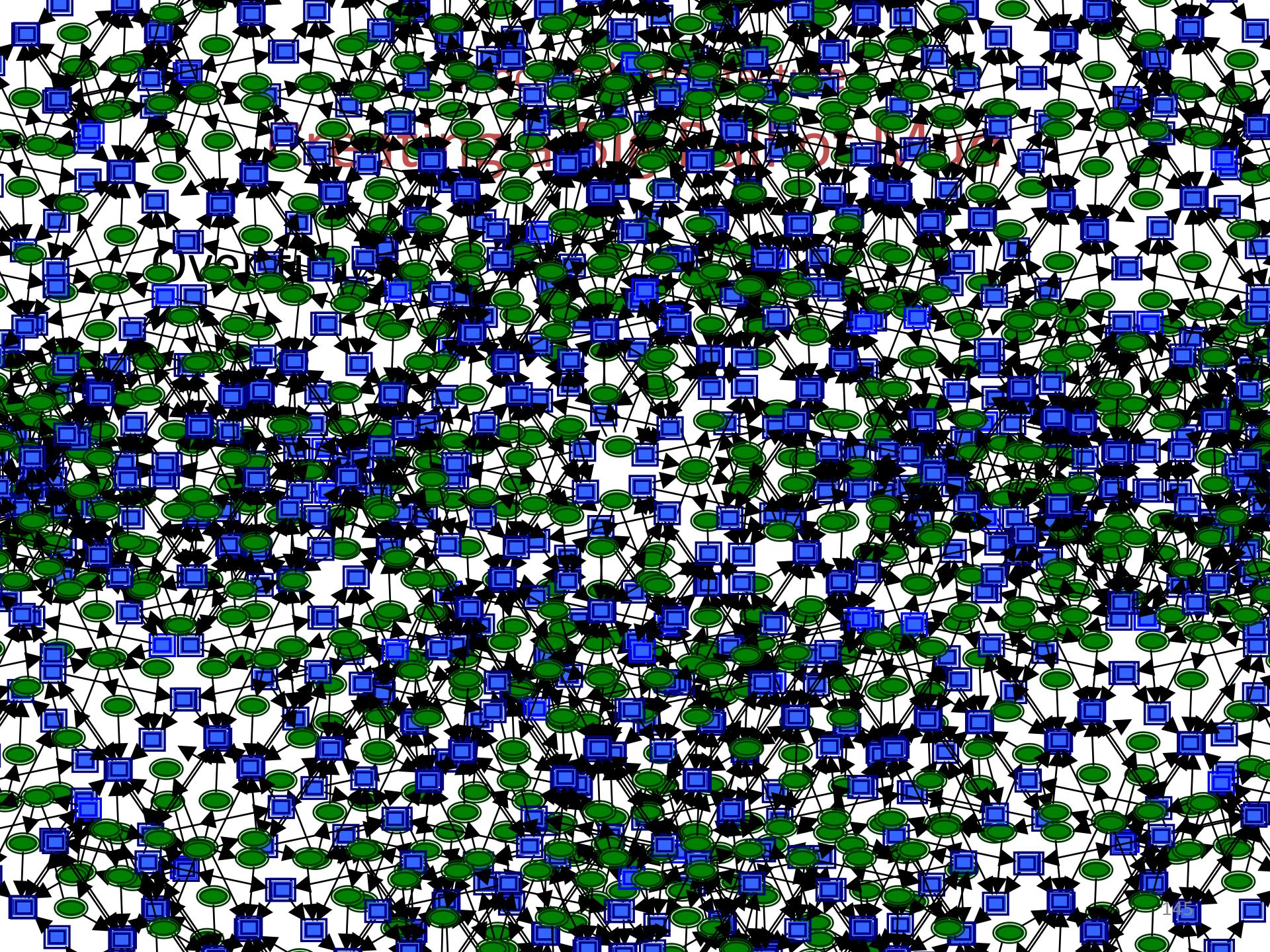


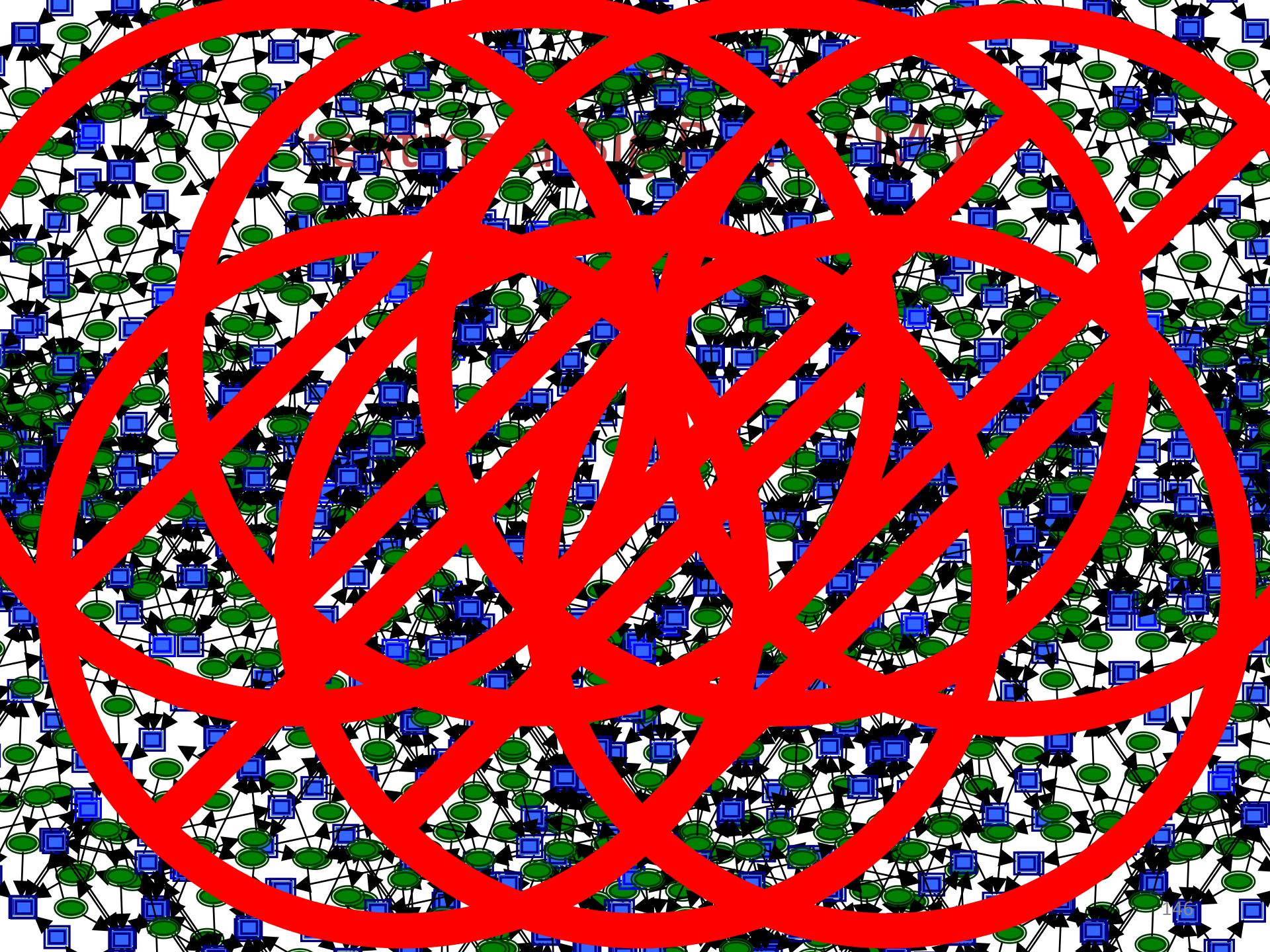
1. Process & Architecture

Creating a Big Ball of Mud

Over time ...







Organizing Principles

Sound Physical Design:

- Regular, Fine-Grained Physical Packaging.
- Uniform Depth of Physical Aggregation.
- Logical/Physical Synergy.

Organizing Principles

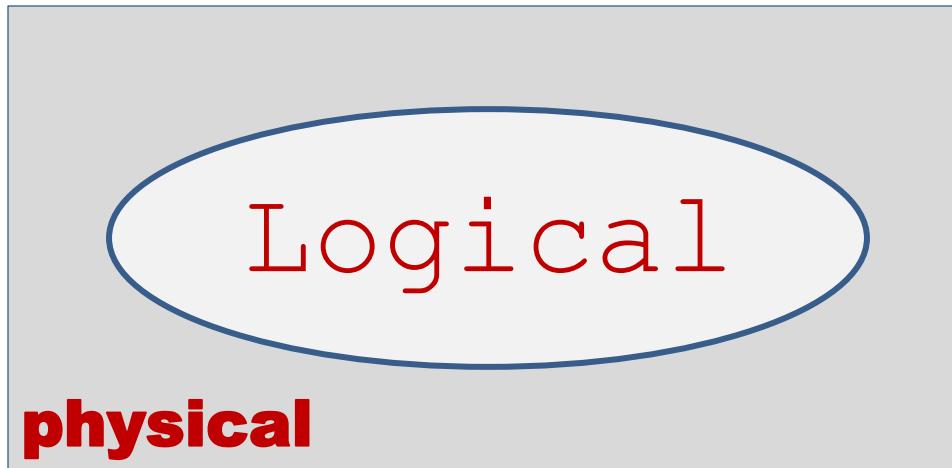
Sound Physical Design:

- Regular, Fine-Grained Physical Packaging.
- Uniform Depth of Physical Aggregation.
- Logical/Physical Synergy.

1. Process & Architecture

Logical versus Physical Design

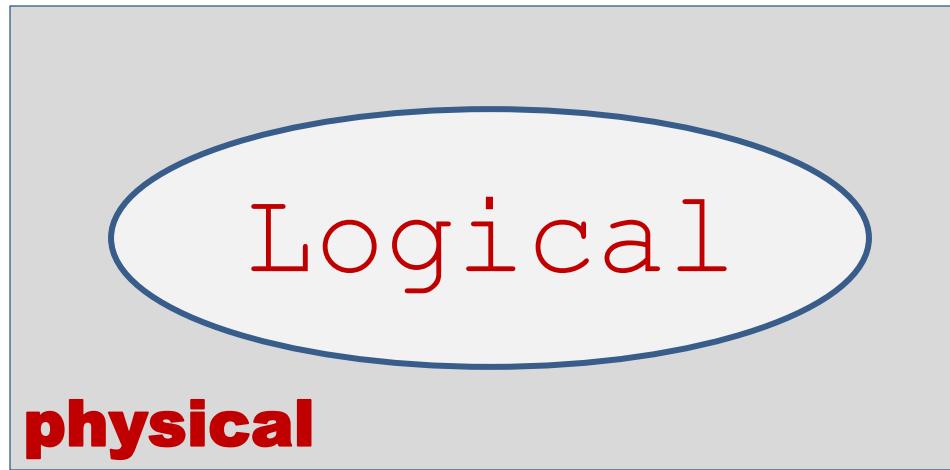
What distinguishes *Logical* from *Physical* Design?



1. Process & Architecture

Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?

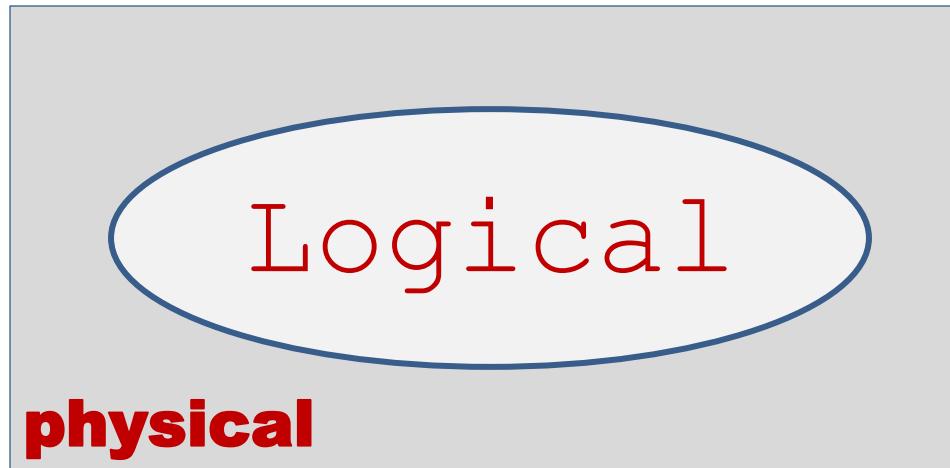


Logical: Classes and Functions

1. Process & Architecture

Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?



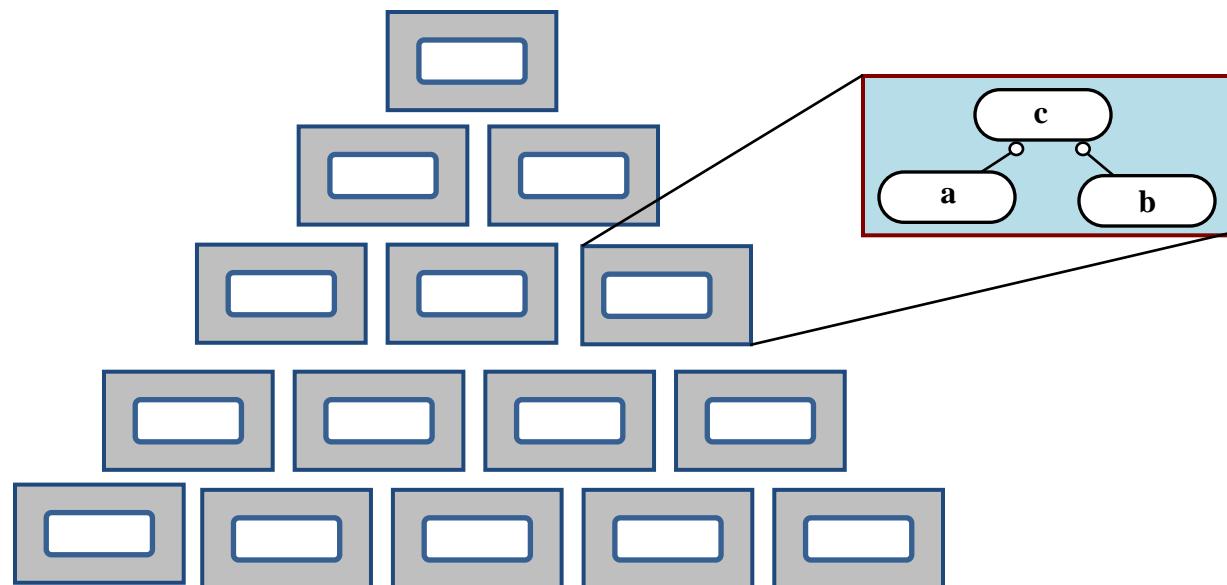
Logical: Classes and Functions

Physical: Files and Libraries

1. Process & Architecture

Logical versus Physical Design

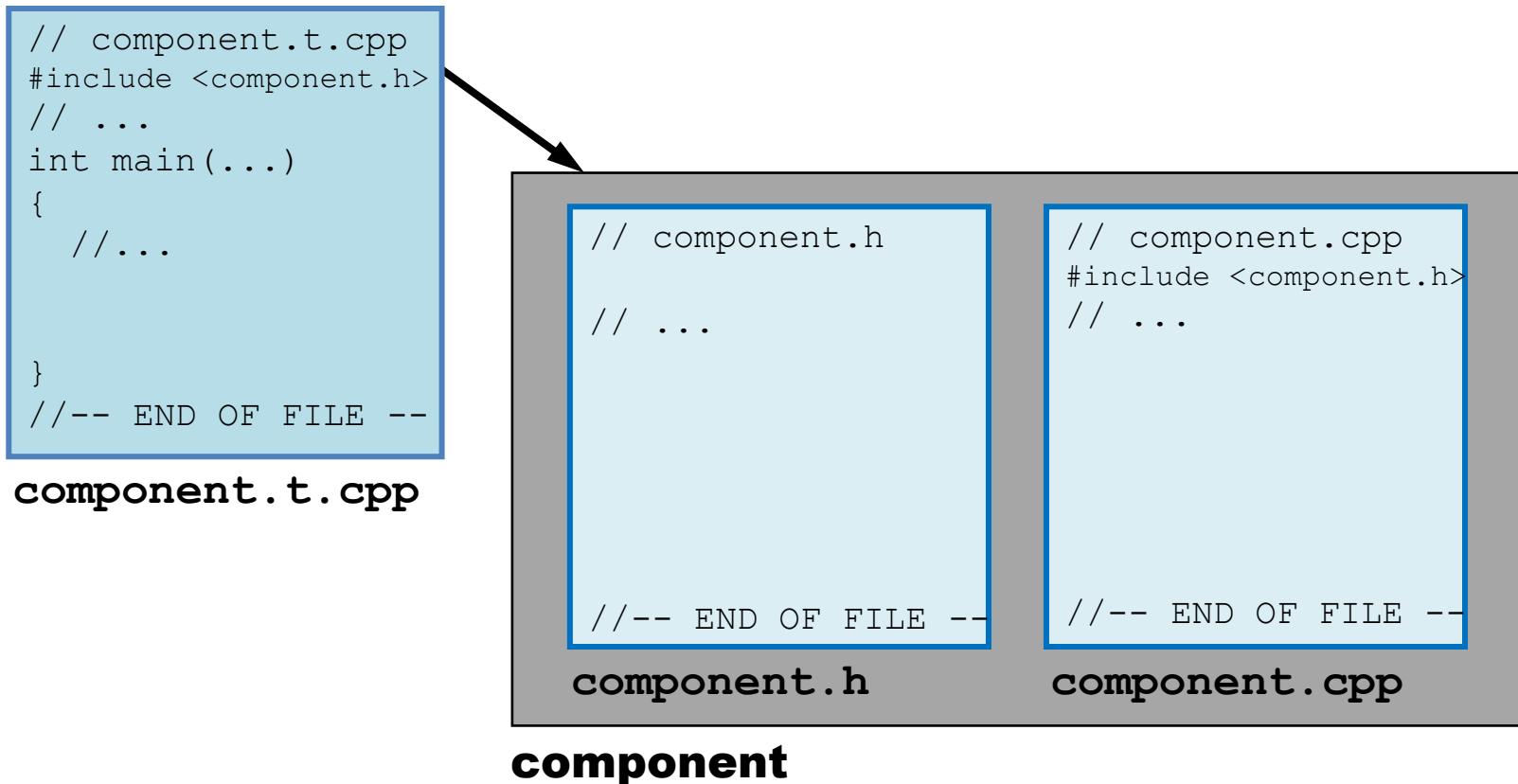
Logical content aggregated into a
Physical hierarchy of **components**



1. Process & Architecture

Component: Uniform Physical Structure

A Component Is Physical



1. Process & Architecture

Component: Uniform Physical Structure

Implementation

```
// component.t.cpp
#include <component.h>
// ...
int main(...)

{
    //...
}

//-- END OF FILE --
```

component.t.cpp

```
// component.h
```

```
// ...
```

```
///-- END OF FILE --
```

component.h

```
// component.cpp
#include <component.h>
// ...
```

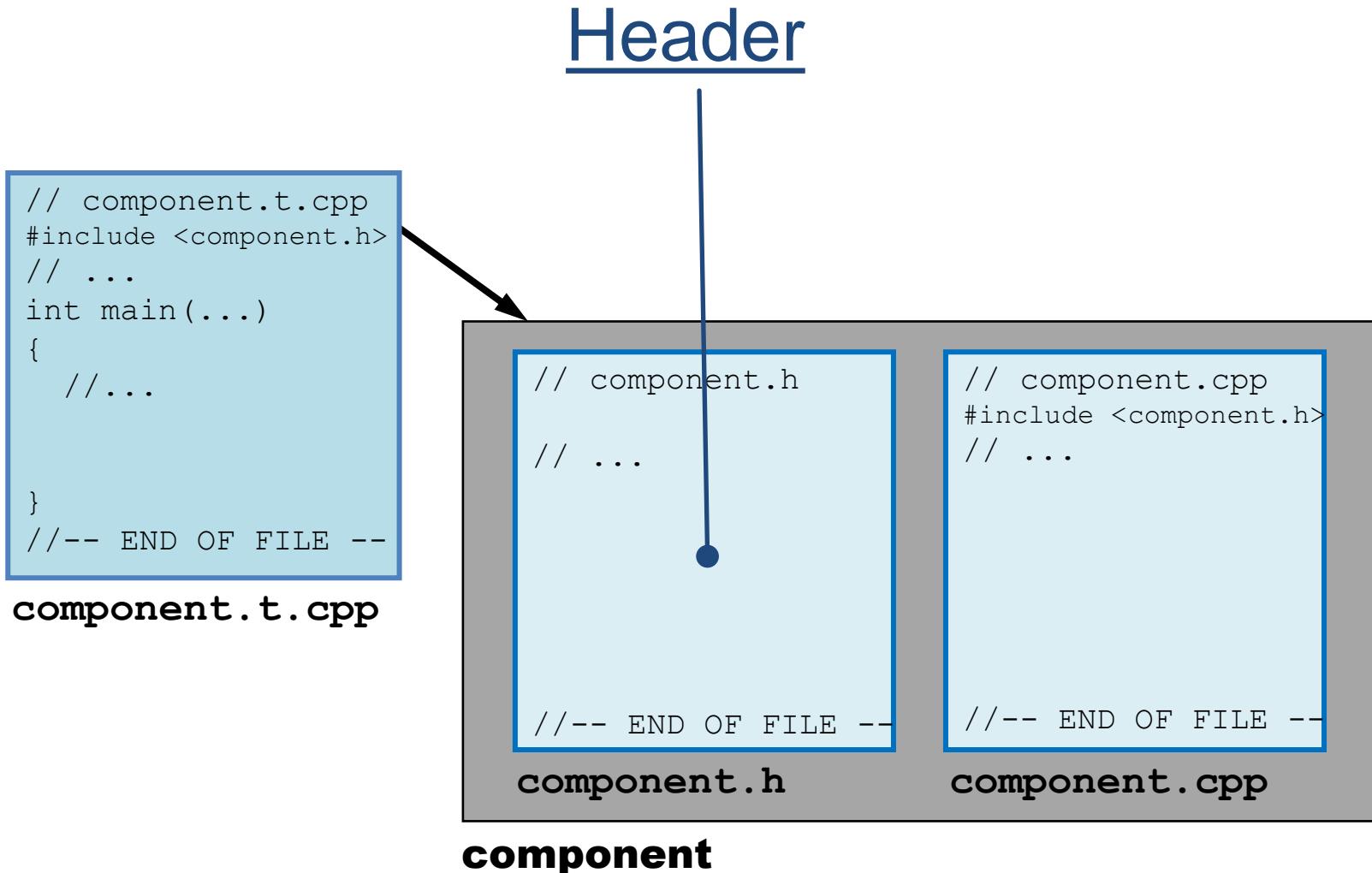
```
///-- END OF FILE --
```

component.cpp

component

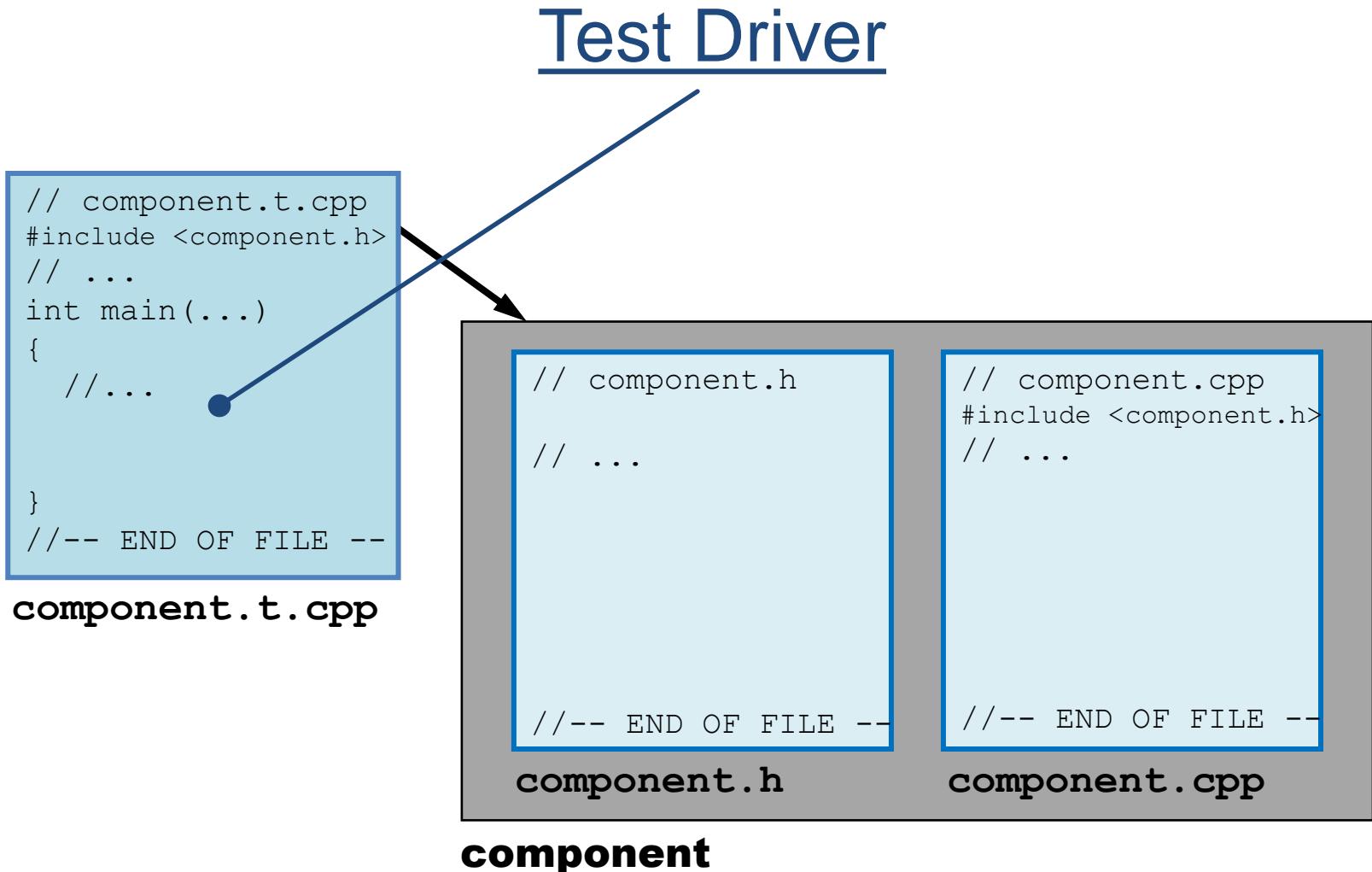
1. Process & Architecture

Component: Uniform Physical Structure



1. Process & Architecture

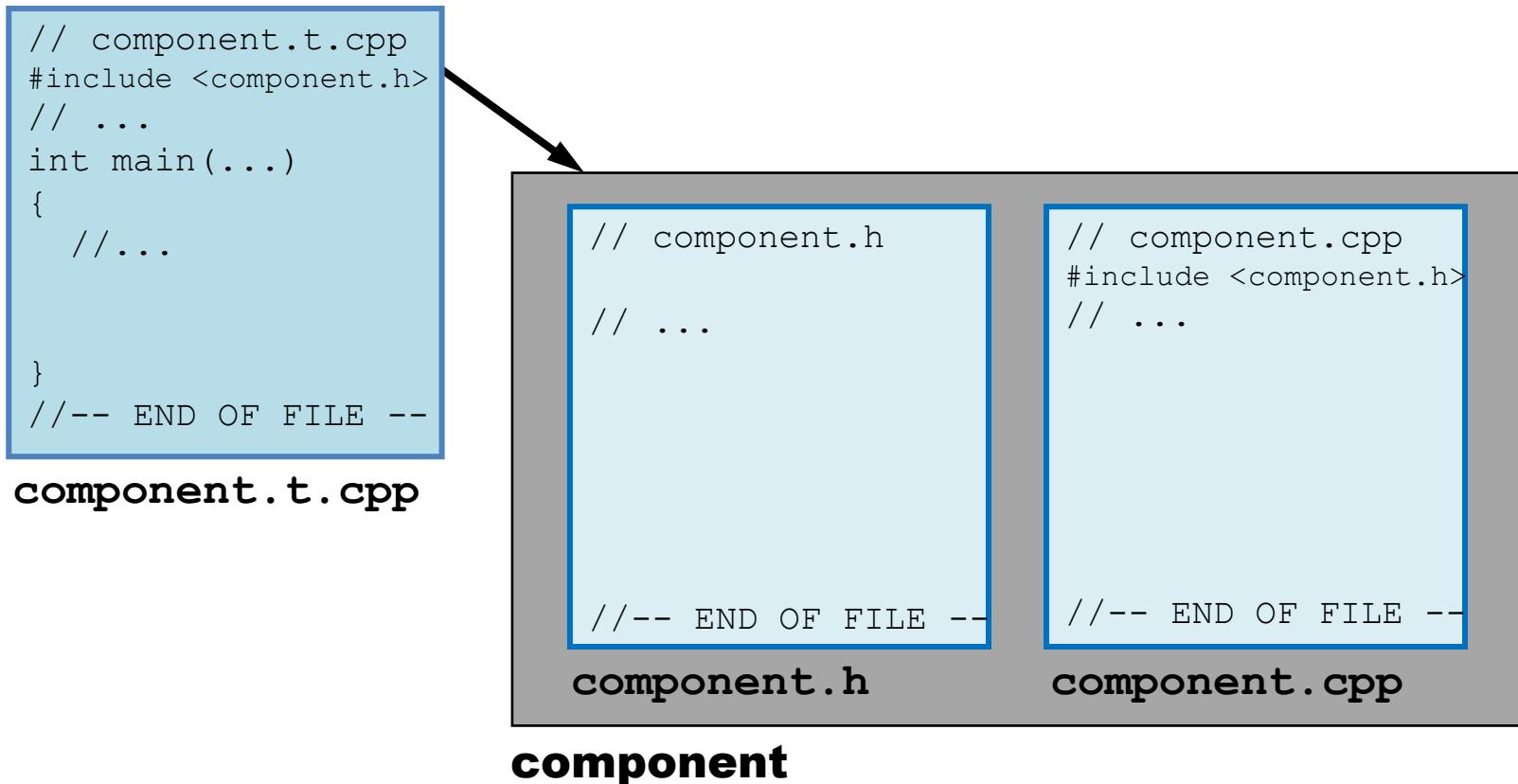
Component: Uniform Physical Structure



1. Process & Architecture

Component: Uniform Physical Structure

The Fundamental Unit of Design



1. Process & Architecture

Component: Not Just a .h / .cpp Pair



my::Widget

my_widget

1. Process & Architecture

Component: Not Just a .h / .cpp Pair

1.  The **.cpp** file includes its **.h** file as the first substantive line of code.

1. Process & Architecture

Component: Not Just a .h / .cpp Pair

1.  The .cpp file includes its .h file as the first substantive line of code.

EVEN IF .CPP IS
OTHERWISE EMPTY!

1. Process & Architecture

Component: Not Just a .h / .cpp Pair

1.  The **.cpp** file includes its **.h** file as the first substantive line of code.
2.  All logical constructs (effectively) having external physical linkage defined in a **.cpp** file are declared in the corresponding **.h** file.

1. Process & Architecture

Component: Not Just a .h / .cpp Pair

1.  The **.cpp** file includes its **.h** file as the first substantive line of code.
2.  All logical constructs (effectively) having external physical linkage defined in a **.cpp** file are declared in the corresponding **.h** file.
3.  All constructs having external physical linkage declared in a **.h** file (if defined at all) are defined within the component.

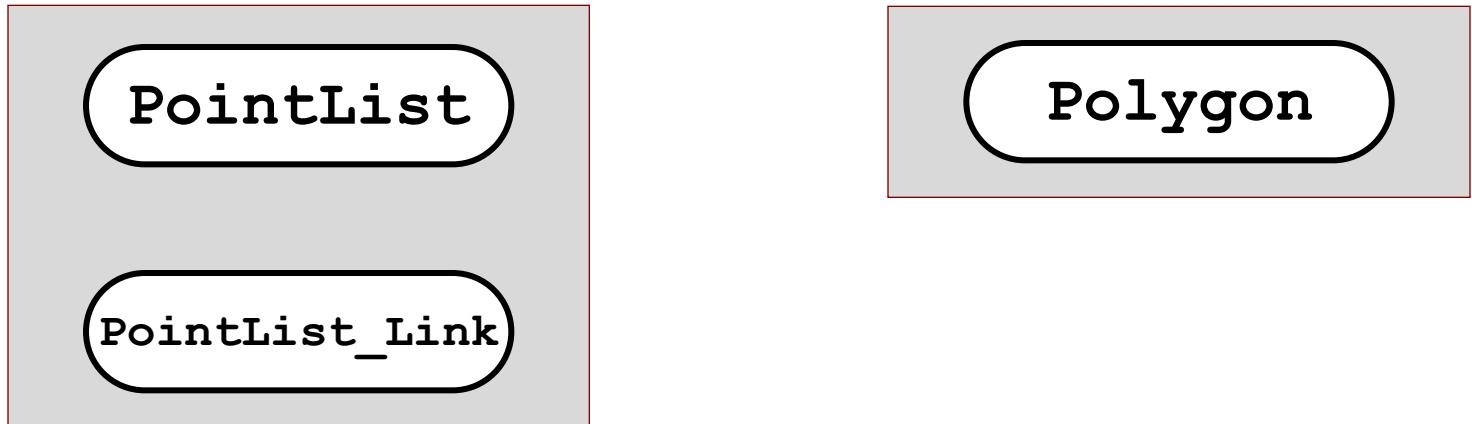
1. Process & Architecture

Component: Not Just a .h / .cpp Pair

1.  The **.cpp** file includes its **.h** file as the first substantive line of code.
2.  All logical constructs (effectively) having external physical linkage defined in a **.cpp** file are declared in the corresponding **.h** file.
3.  All constructs having external physical linkage declared in a **.h** file (if defined at all) are defined within the component.
4.  A component's functionality is accessed via a **#include** of its header, and never via a forward (**extern**) declaration.

1. Process & Architecture

Logical Relationships

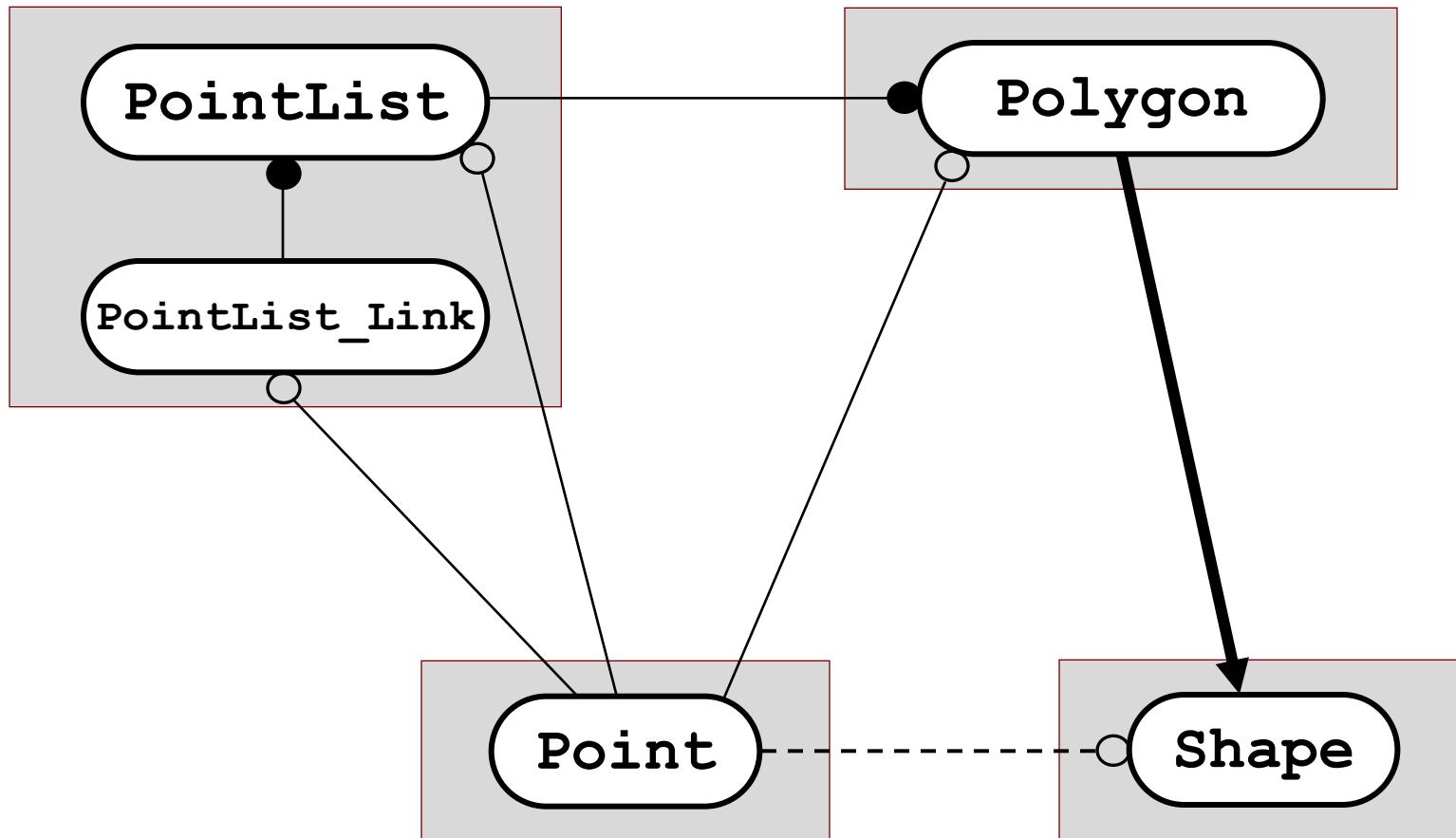


○— Uses-in-the-Interface
●— Uses-in-the-Implementation

○----- Uses in name only
→ Is-A

1. Process & Architecture

Logical Relationships

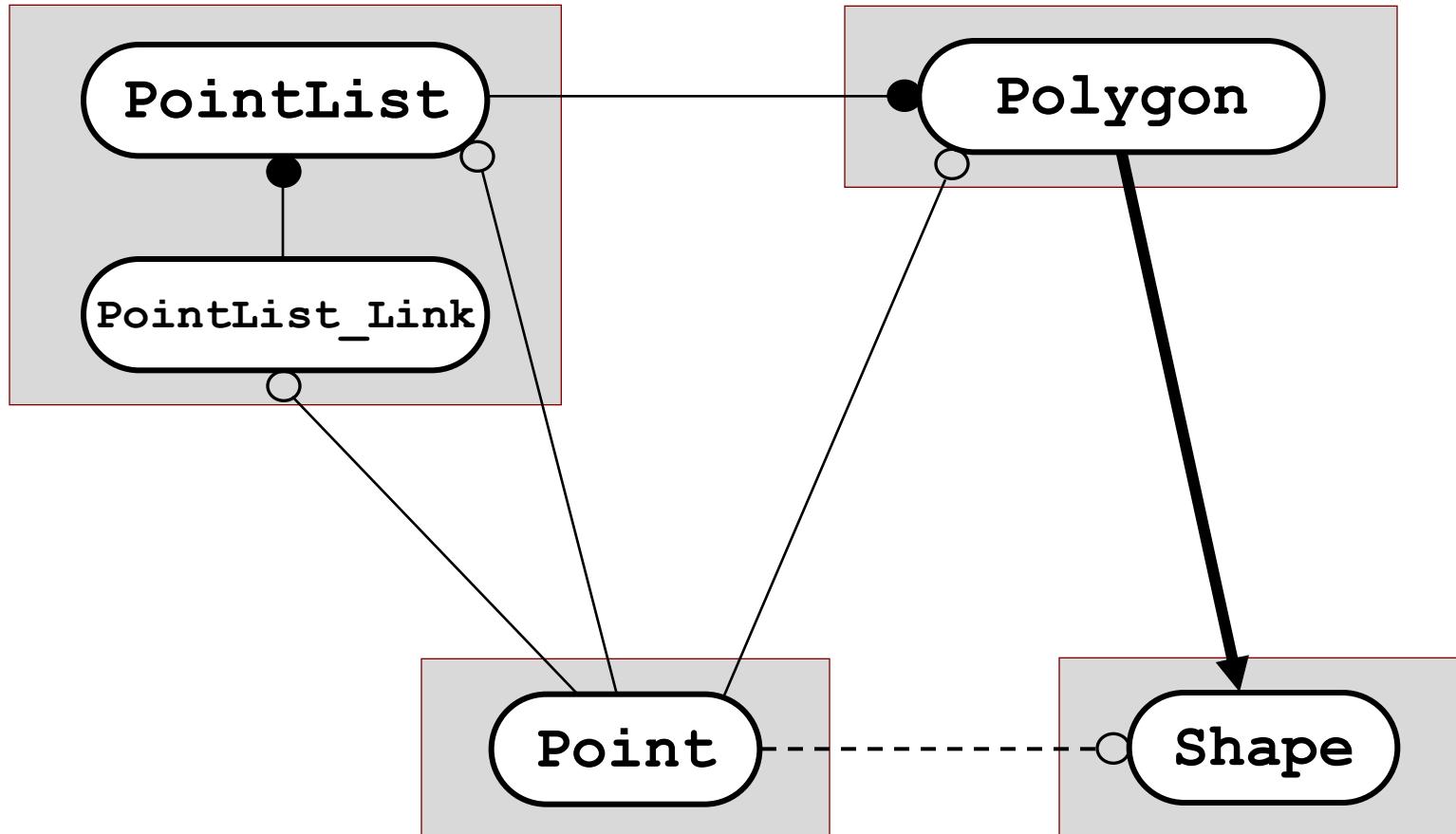


○————— Uses-in-the-Interface
●————— Uses-in-the-Implementation

○ - - - - - Uses in name only
———— Is-A

1. Process & Architecture

Implied Dependency

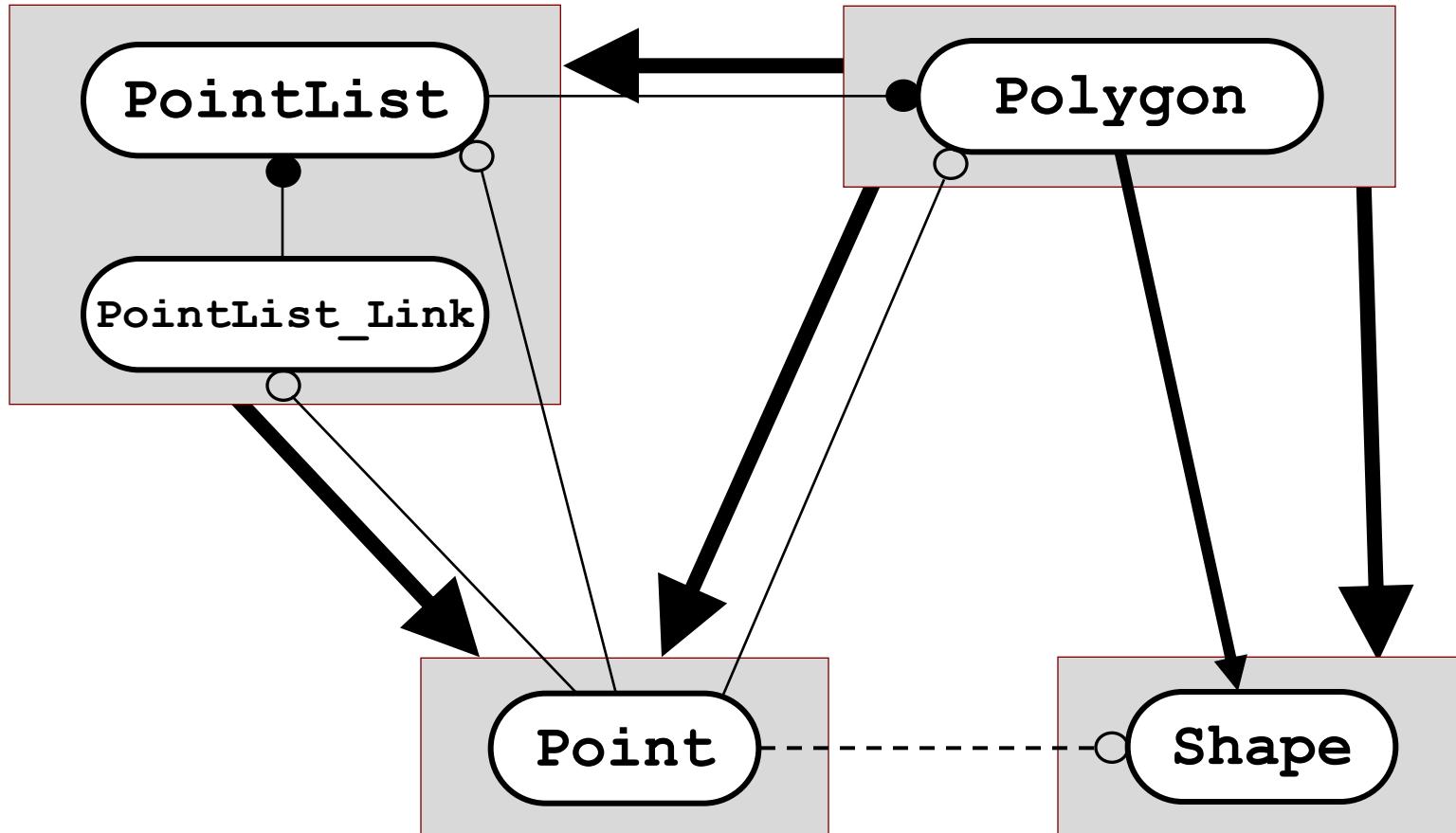


○— Uses-in-the-Interface
●— Uses-in-the-Implementation

→ Depends-On
○--- Uses in name only
→ Is-A

1. Process & Architecture

Implied Dependency

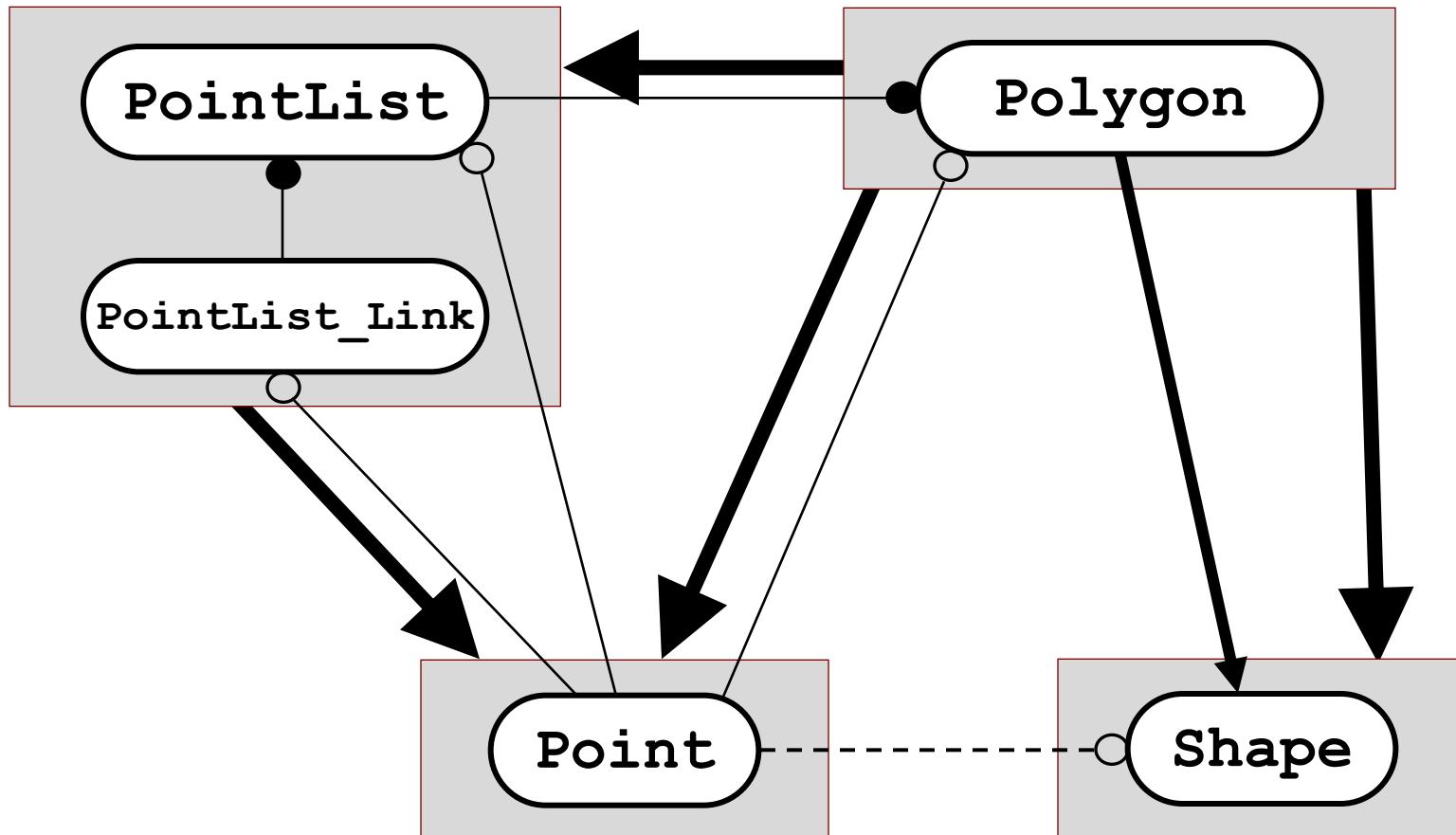


○—> Uses-in-the-Interface
●—> Uses-in-the-Implementation

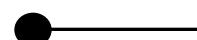
→ Depends-On
○-----> Uses in name only
→ Is-A

1. Process & Architecture

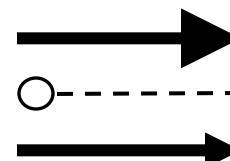
Level Numbers



Uses-in-the-Interface



Uses-in-the-Implementation



Depends-On



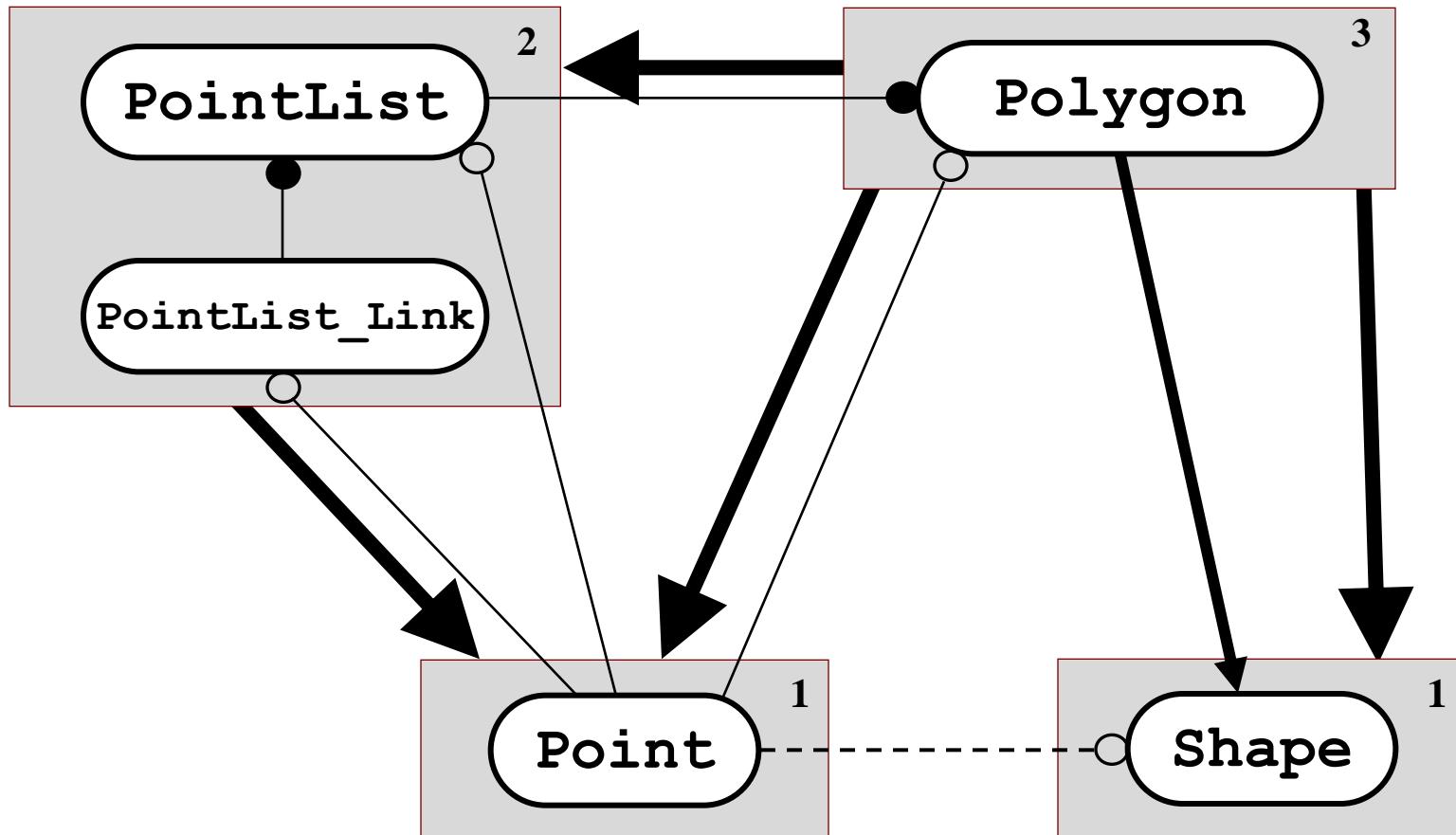
Uses in name only



Is-A

1. Process & Architecture

Level Numbers



○—> Uses-in-the-Interface
●—> Uses-in-the-Implementation

→ Depends-On
○-----> Uses in name only
→ Is-A

1. Process & Architecture

Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

1. Process & Architecture

Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

Usage:

1. Process & Architecture

Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

Usage:

1. Process & Architecture

Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.

1. Process & Architecture

Levelization

Levelize (v.); **Levelizable** (a.); Levelization (n.)

Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.

1. Process & Architecture

Levelization

Levelize (v.); **Levelizable** (a.); Levelization (n.)

Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.
- Are you sure that design is ***levelizable*** – i.e., do we know how to make its physical dependencies acyclic?

1. Process & Architecture

Levelization

Levelize (v.); Levelizable (a.); **Levelization (n.)**

Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.
- Are you sure that design is *levelizable* – i.e., do we know how to make its physical dependencies acyclic?

1. Process & Architecture

Levelization

Levelize (v.); Levelizable (a.); **Levelization (n.)**

Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.
- Are you sure that design is *levelizable* – i.e., do we know how to make its physical dependencies acyclic?
- What ***levelization*** techniques would you use – i.e., what techniques would you use to *levelize* your design?

1. Process & Architecture

Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.
- Are you sure that design is *levelizable* – i.e., do we know how to make its physical dependencies acyclic?
- What *levelization* techniques would you use – i.e., what techniques would you use to *levelize* your design?

Note that Lakos'96 described 9 different ways to untangle cyclic physical dependencies: ***Escalation, Demotion, Opaque Pointers, Dumb Data, Redundancy, Callbacks, Manager Class, Factoring, and Escalating Encapsulation.***¹⁷⁹

1. Process & Architecture

Levelization Techniques (Summary)

Escalation – Moving mutually dependent functionality higher.

Demotion – Moving common functionality lower.

Opaque Pointers – Having an object use

Dumb Data – Using Data that is shared by multiple objects. Only in the context of a separate, higher-level object.

Redundancy – Deliberately duplicating code or data to avoid coupling.

Callbacks – Invoking lower-level subsystems to perform specific tasks.

Manager – Manages lower-level objects and coordinates lower-level objects.

Factoring – Separating logically testable sub-behavior out of the implementation of complex components to reduce excessive physical coupling.

Escalating Encapsulation – Moving the point at which implementation details are hidden from clients to a higher level in the physical hierarchy.

(Shameless)

Advertisement

1. Process & Architecture

Levelization Techniques (Summary)

Escalation – Moving mutually dependent functionality higher in the physical hierarchy.

Demotion – Moving common functionality lower in the physical hierarchy.

Opaque Pointers – Having an object use another in name only.

Dumb Data – Using Data that indicates a dependency on a peer object, but only in the context of a separate, higher-level object.

Redundancy – Deliberately avoiding reuse by repeating a small amount of code or data to avoid coupling.

Callbacks – Client-supplied functions that enable lower-level subsystems to perform specific tasks in a more global context.

Manager Class – Establishing a class that owns and coordinates lower-level objects.

Factoring – Moving independently testable sub-behavior out of the implementation of complex component involved in excessive physical coupling.

Escalating Encapsulation – Moving the point at which implementation details are hidden from clients to a higher level in the physical hierarchy.

1. Process & Architecture

Essential Physical Design Rules

1. Process & Architecture

Essential Physical Design Rules

There are two:

1. Process & Architecture

Essential Physical Design Rules

There are two:

1. No *Cyclic* Physical
Dependencies!

1. Process & Architecture

Essential Physical Design Rules

There are two:

1. No *Cyclic* Physical
Dependencies!

2. No *Long-Distance*
Friendships!

1. Process & Architecture

Criteria for Collocating “Public” Classes

1. Process & Architecture

Criteria for Collocating “Public” Classes

There are four:

1. Process & Architecture

Criteria for Collocating “Public” Classes

There are four:

1. Friendship.

1. Process & Architecture

Criteria for Collocating “Public” Classes

There are four:

1. Friendship.
2. Cyclic Dependency.

1. Process & Architecture

Criteria for Collocating “Public” Classes

There are four:

1. Friendship.

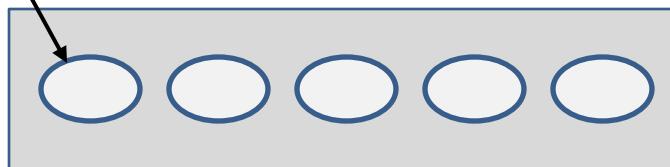
2. Cyclic Dependency.

3. Single Solution.

1. Process & Architecture

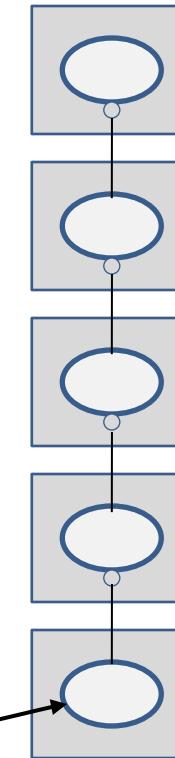
Criteria for Collocating “Public” Classes

Not reusable
independently.



Single Solution

Independently
reusable.



Hierarchy of Solutions

1. Process & Architecture

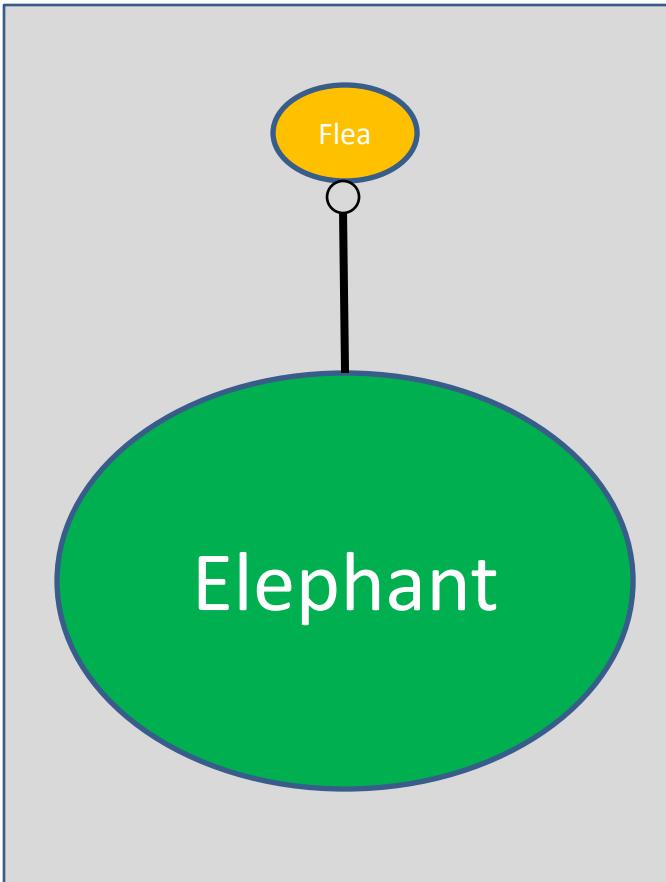
Criteria for Collocating “Public” Classes

There are four:

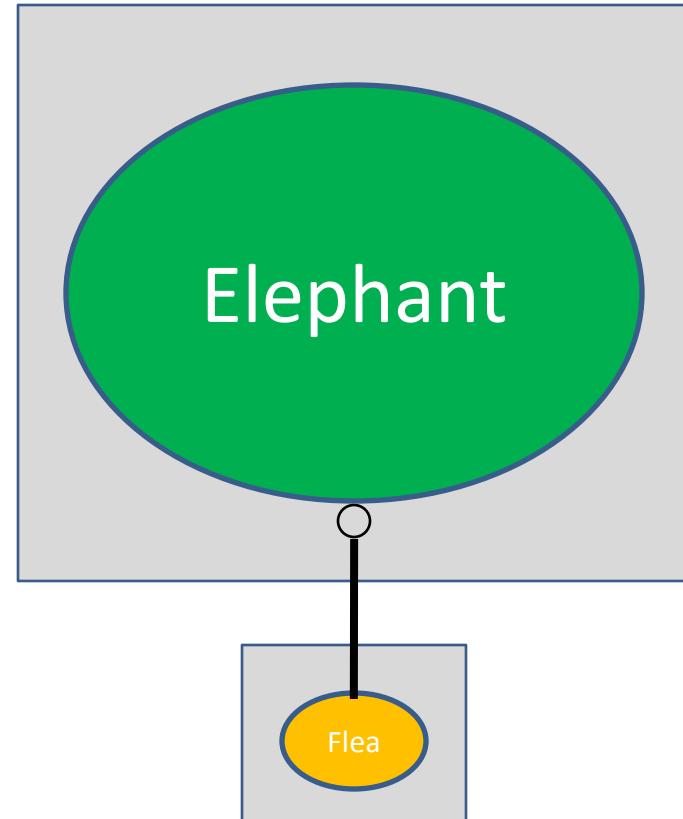
1. Friendship.
2. Cyclic Dependency.
3. Single Solution.
4. “Flea on an Elephant.”

1. Process & Architecture

Criteria for Collocating “Public” Classes



“Flea on an Elephant”



(Elephant on a Flea)

Organizing Principles

Sound Physical Design:

- Regular, Fine-Grained Physical Packaging.
- Uniform Depth of Physical Aggregation.
- Logical/Physical Synergy.

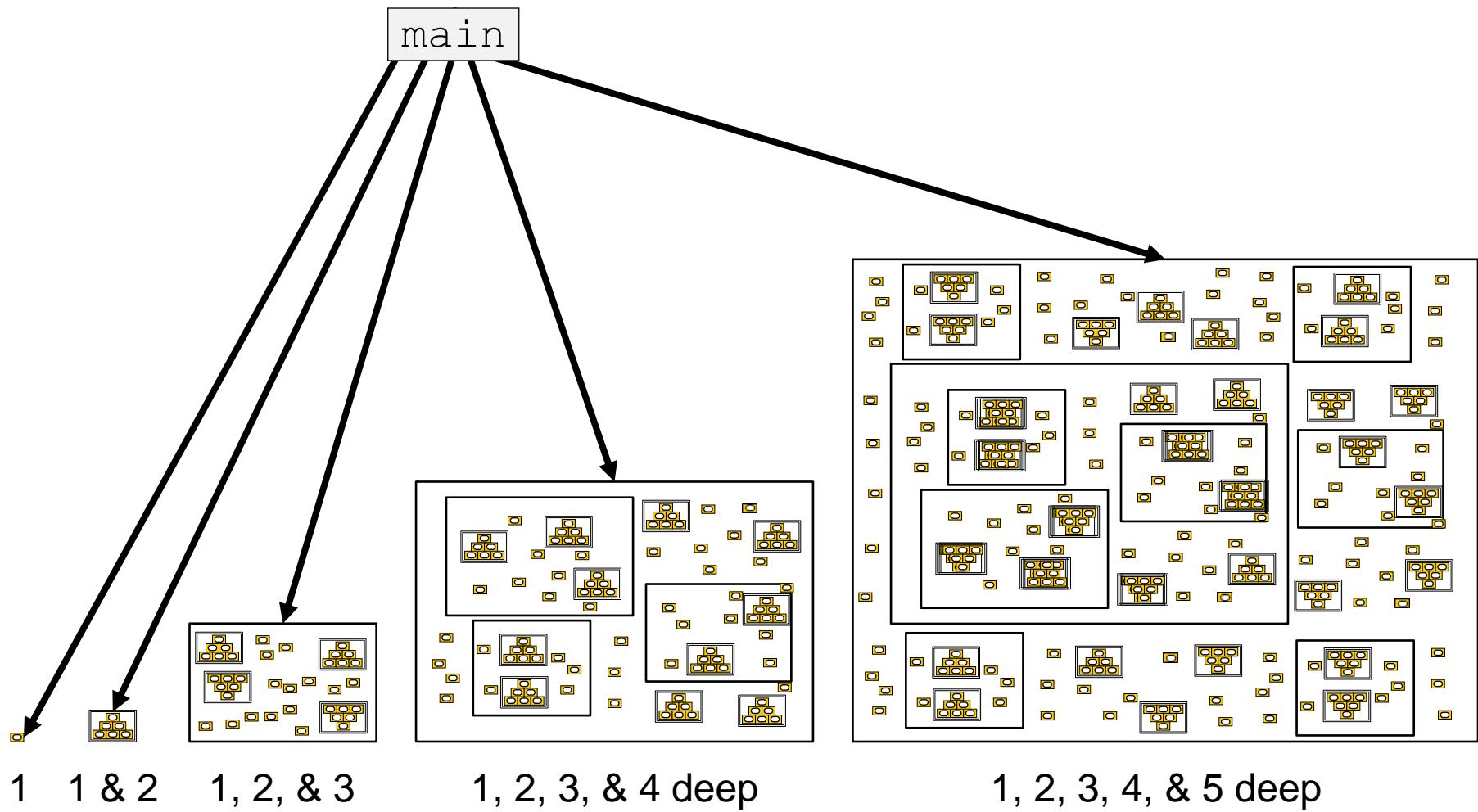
Organizing Principles

Sound Physical Design:

- Regular, Fine-Grained Physical Packaging.
- Uniform Depth of Physical Aggregation.
- Logical/Physical Synergy.

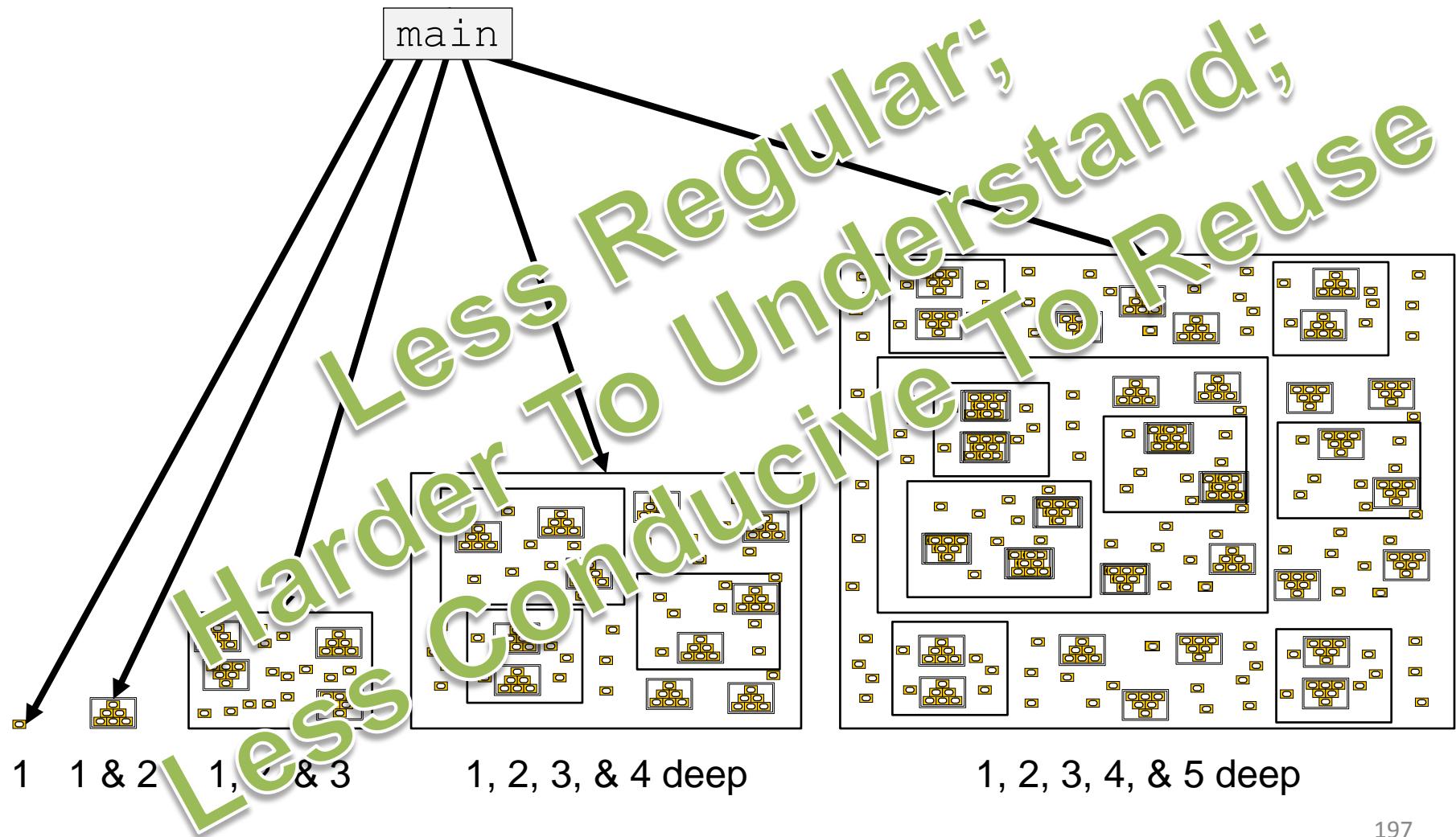
1. Process & Architecture

Uniform Depth of Physical Aggregation



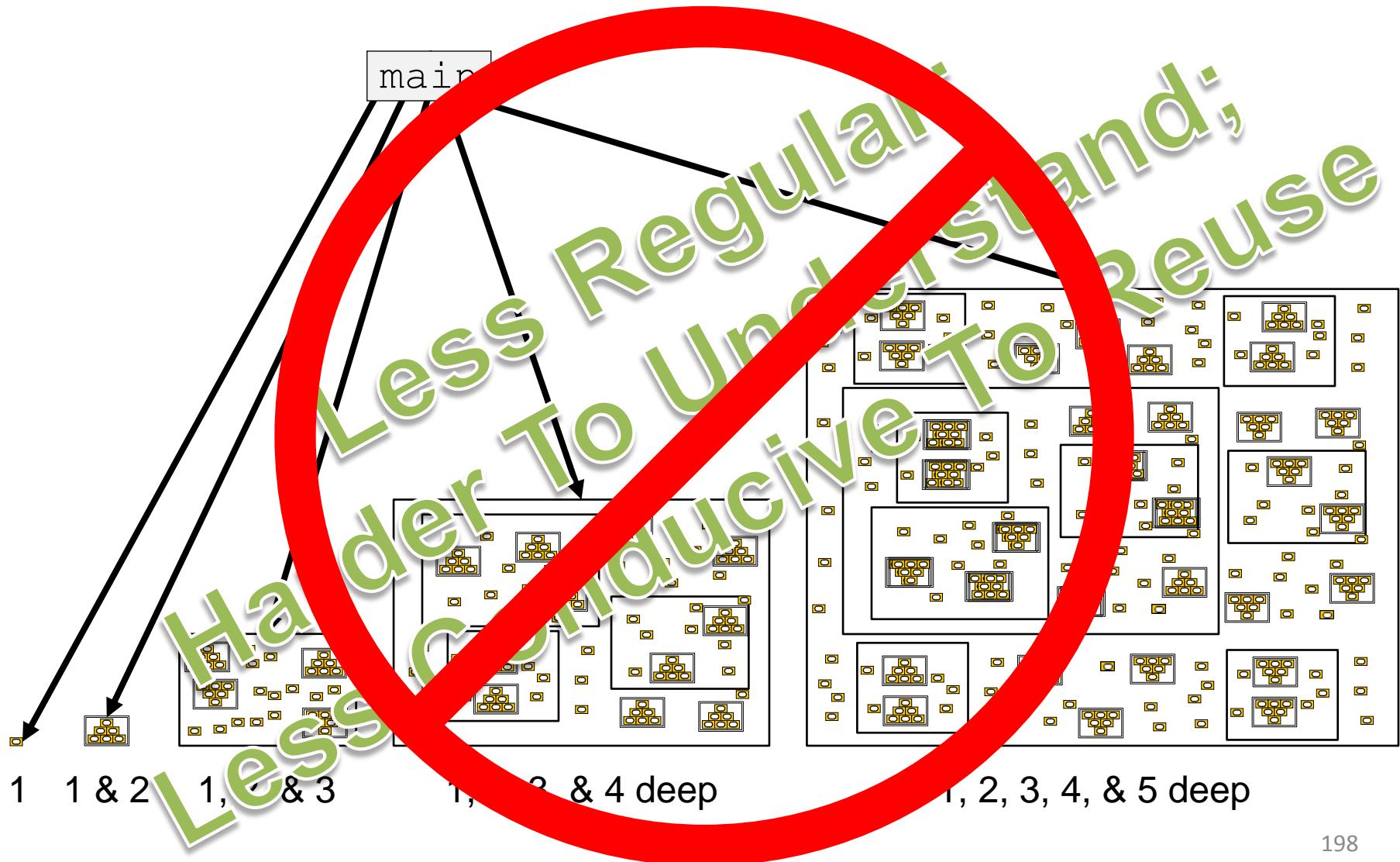
1. Process & Architecture

Uniform Depth of Physical Aggregation



1. Process & Architecture

Uniform Depth of Physical Aggregation



1. Process & Architecture

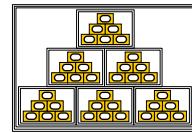
Uniform Depth of Physical Aggregation



Component



Package



Package Group

1. Process & Architecture

Uniform Depth of Physical Aggregation

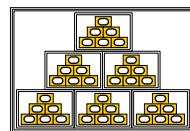
Exactly Three Levels
of Physical Aggregation



Component



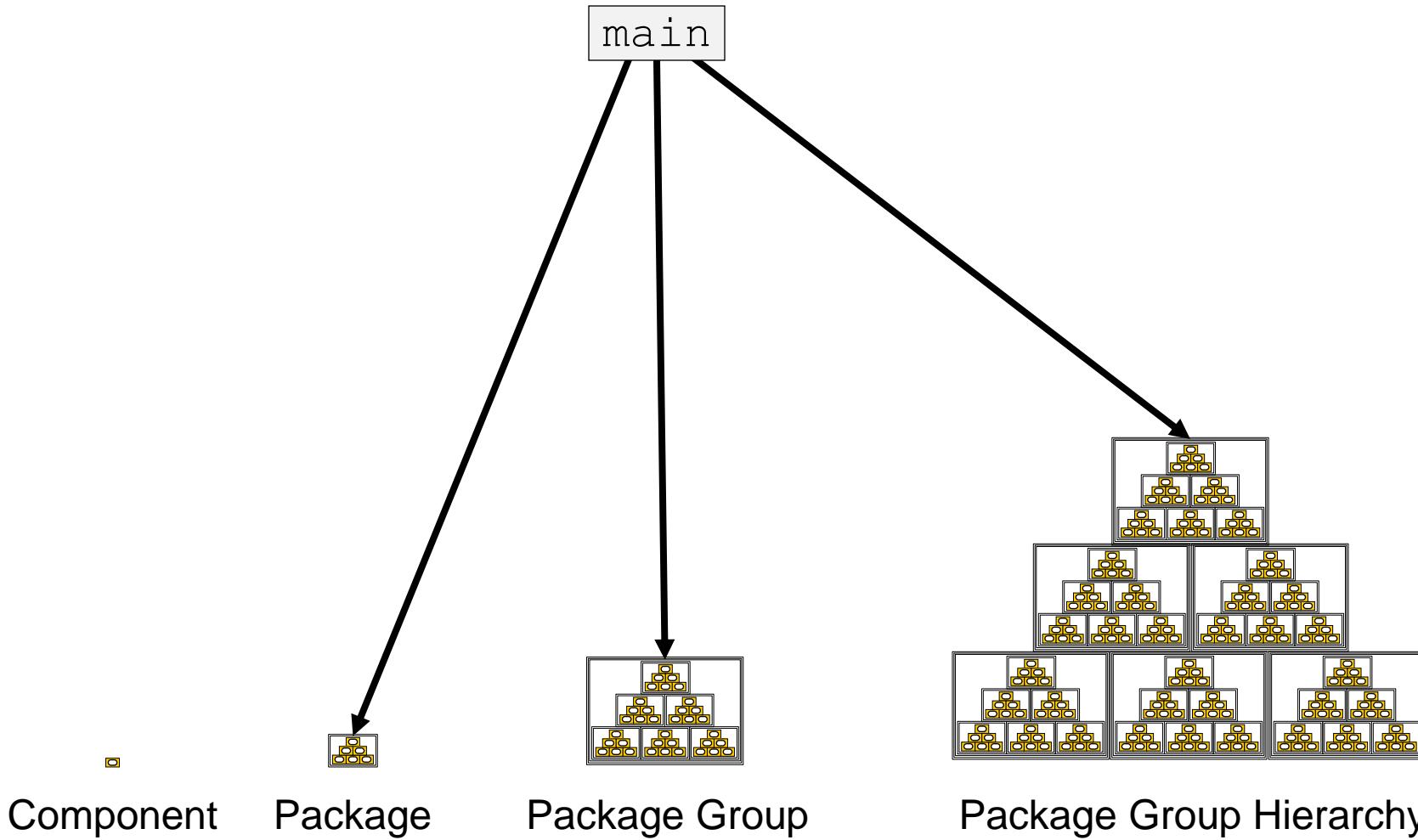
Package



Package Group

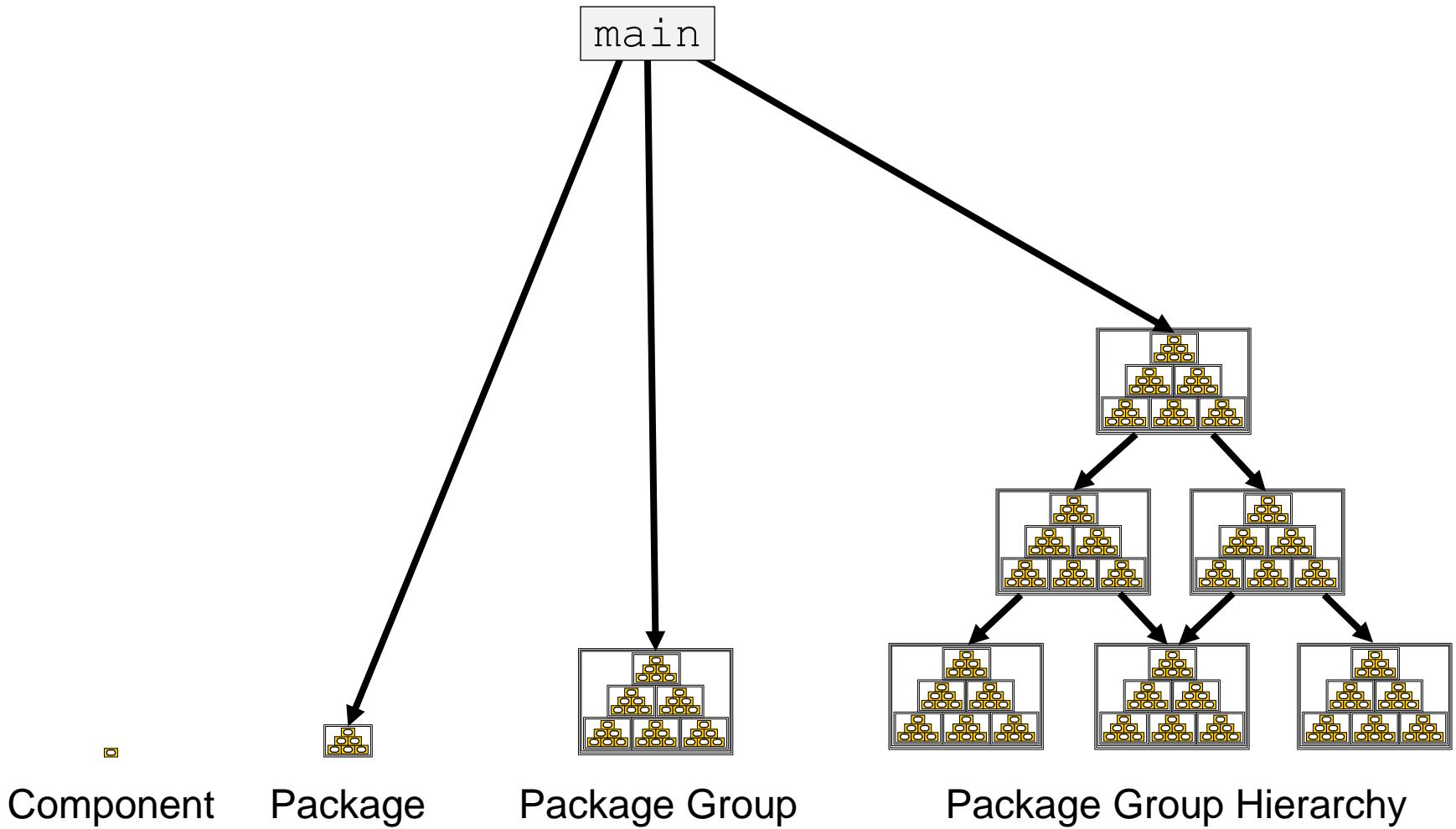
1. Process & Architecture

Uniform Depth of Physical Aggregation



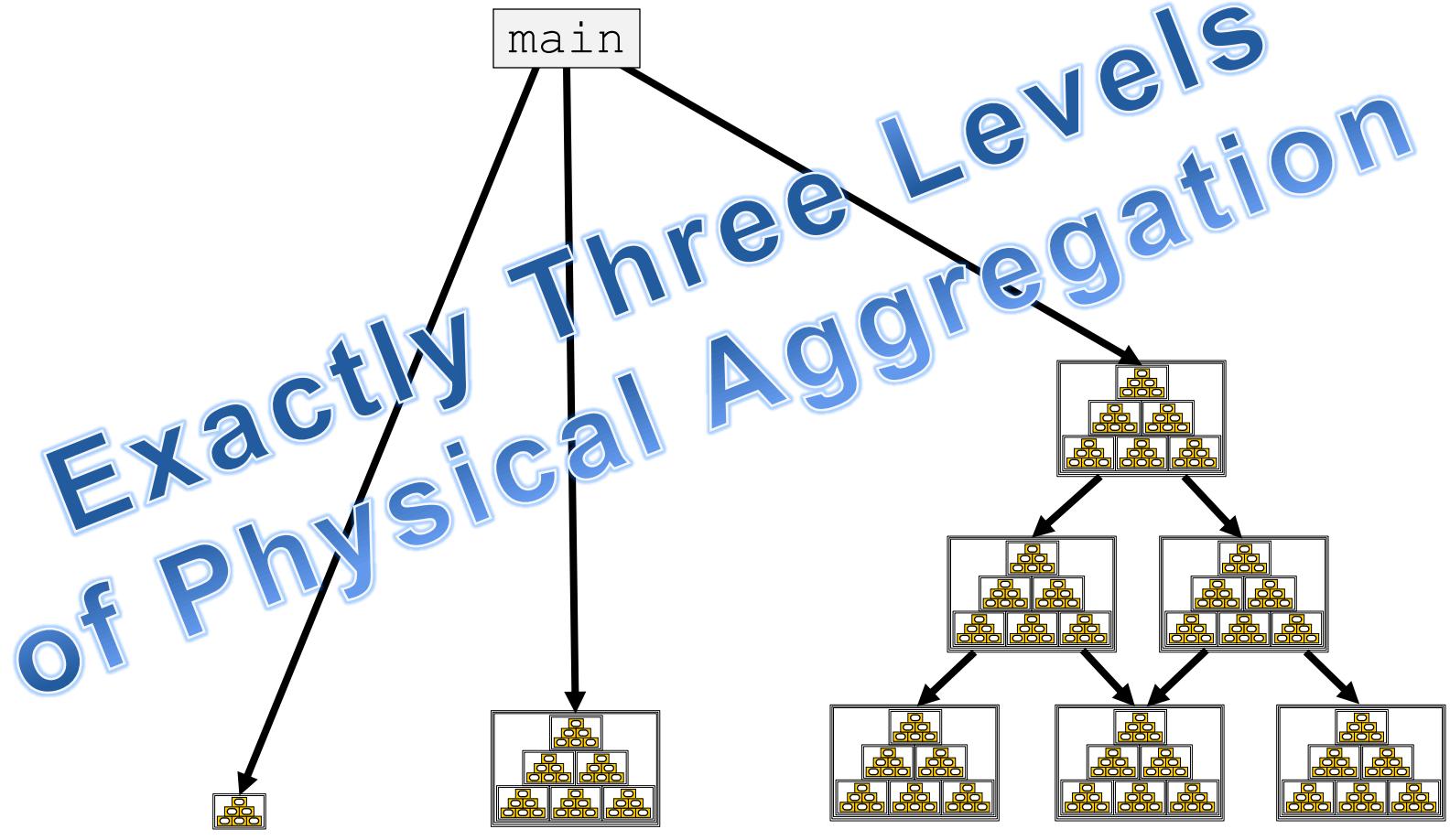
1. Process & Architecture

Uniform Depth of Physical Aggregation



1. Process & Architecture

Uniform Depth of Physical Aggregation



1. Process & Architecture Components

Five levels of **physical dependency**:

Level 5:



Level 4:



Level 3:



Level 2:



Level 1:



1. Process & Architecture

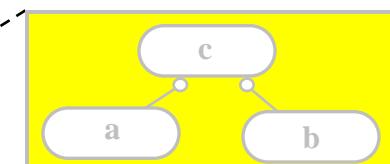
Components

Only one level of **physical aggregation**:

Level 5:



Level 4:



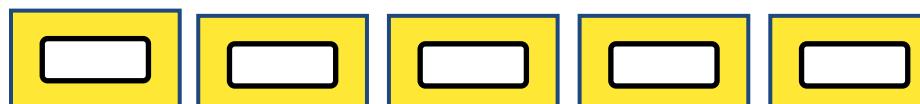
Level 3:



Level 2:



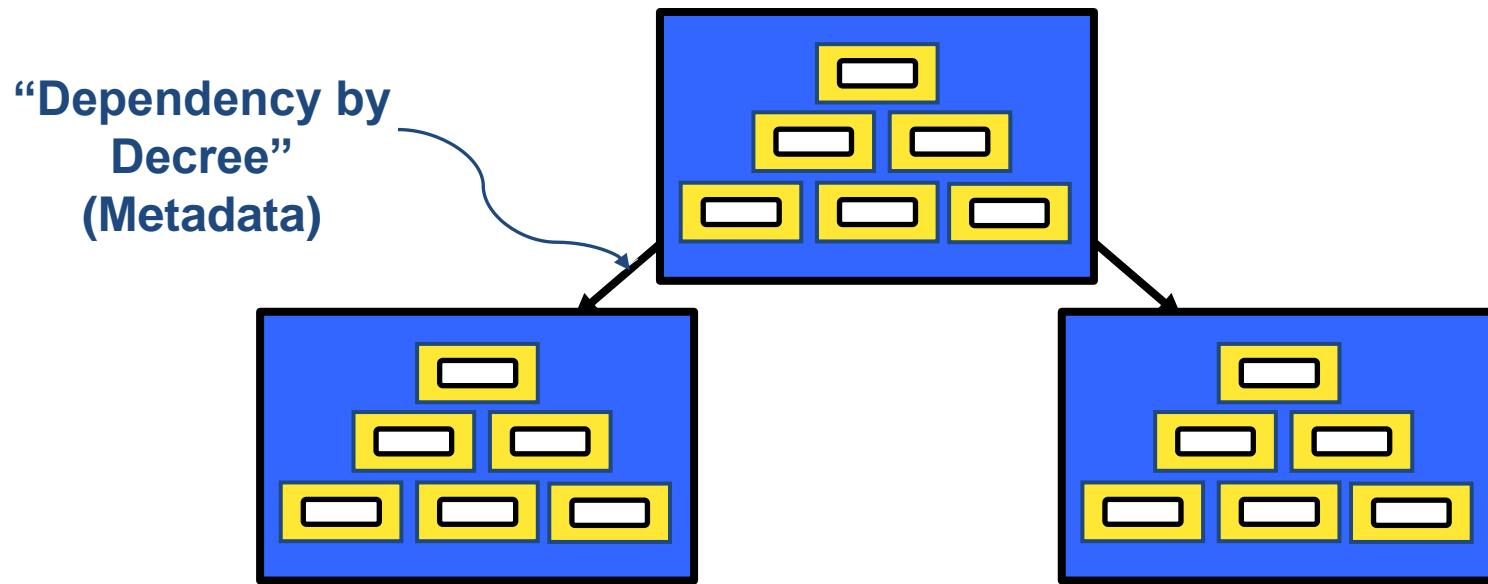
Level 1:



1. Process & Architecture

Packages

Two levels of physical aggregation:

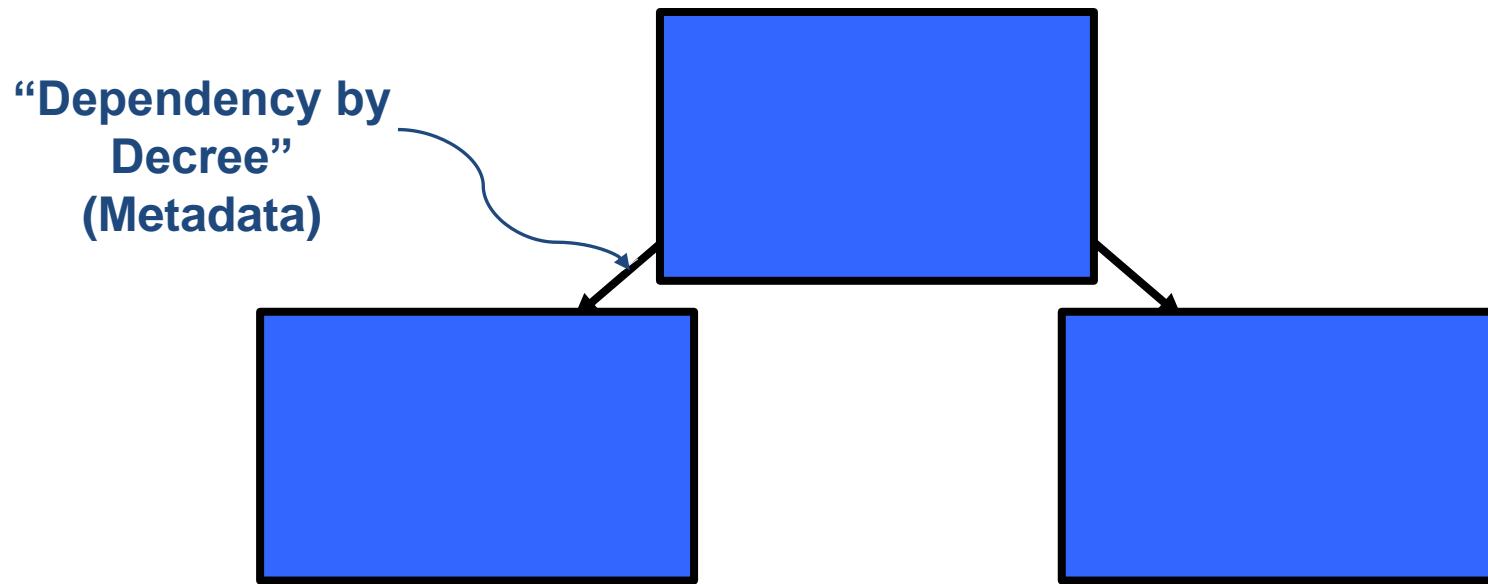


“A Hierarchy of Component Hierarchies”

1. Process & Architecture

Packages

Two levels of physical aggregation:

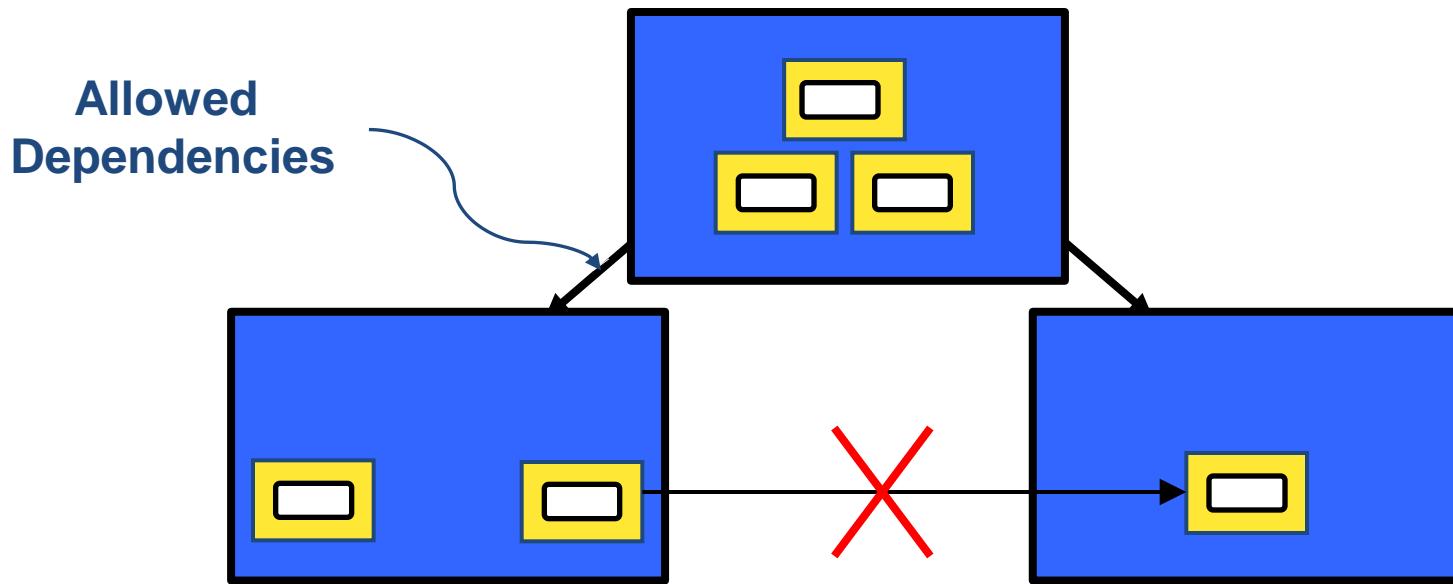


Metadata governs, even absent of any components!

1. Process & Architecture

Packages

Two levels of physical aggregation:

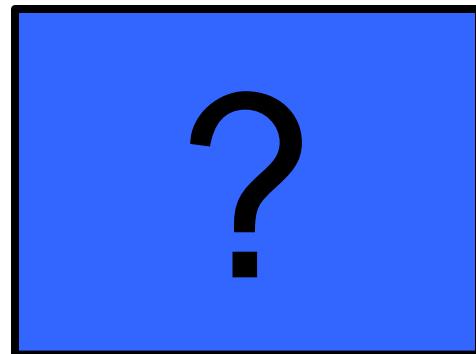


Metadata governs allowed dependencies.

1. Process & Architecture

Packages

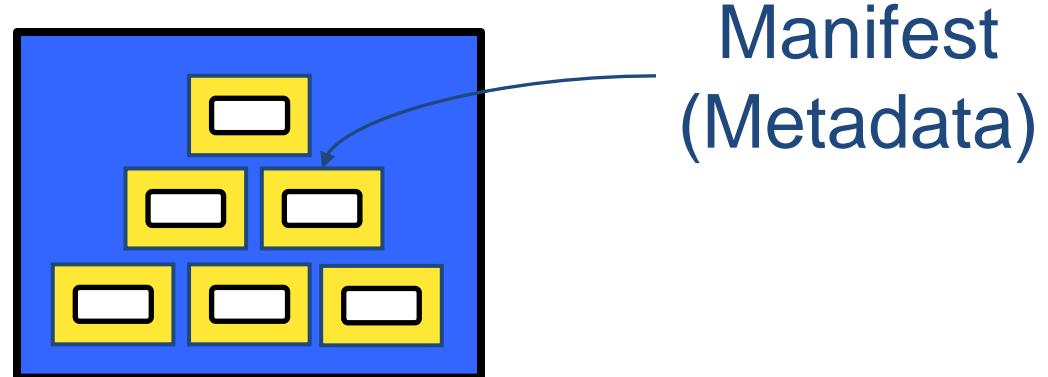
Properties of an aggregate:



1. Process & Architecture

Packages

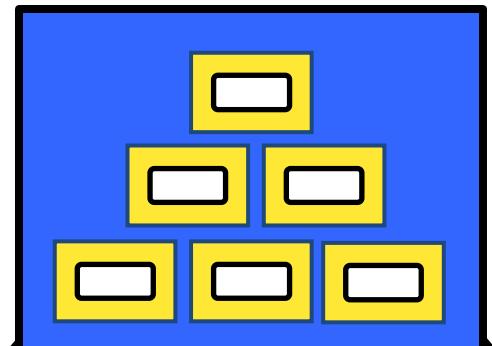
Properties of an aggregate:



1. Process & Architecture

Packages

Properties of an aggregate:



Allowed Dependencies
(Metadata)

1. Process & Architecture

Packages

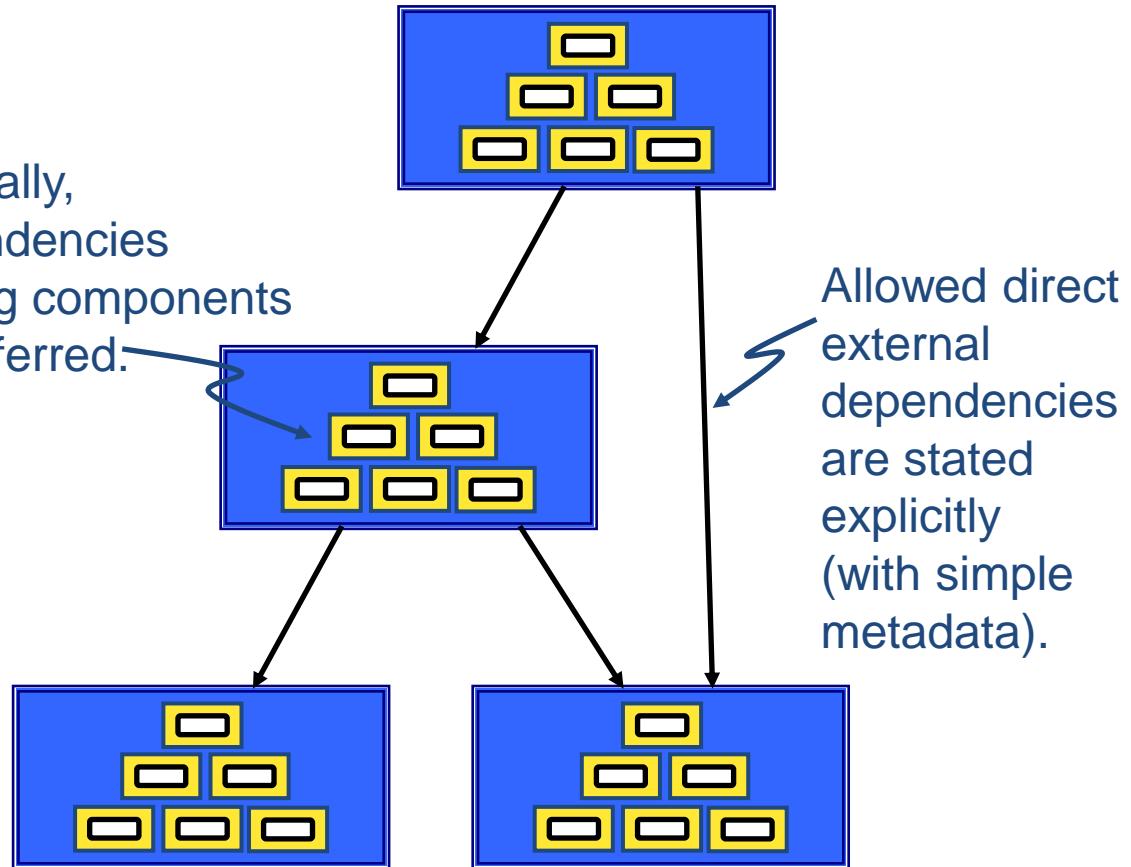
Aggregate dependencies:

Aggregate Level 3:

Internally,
dependencies
among components
are inferred.

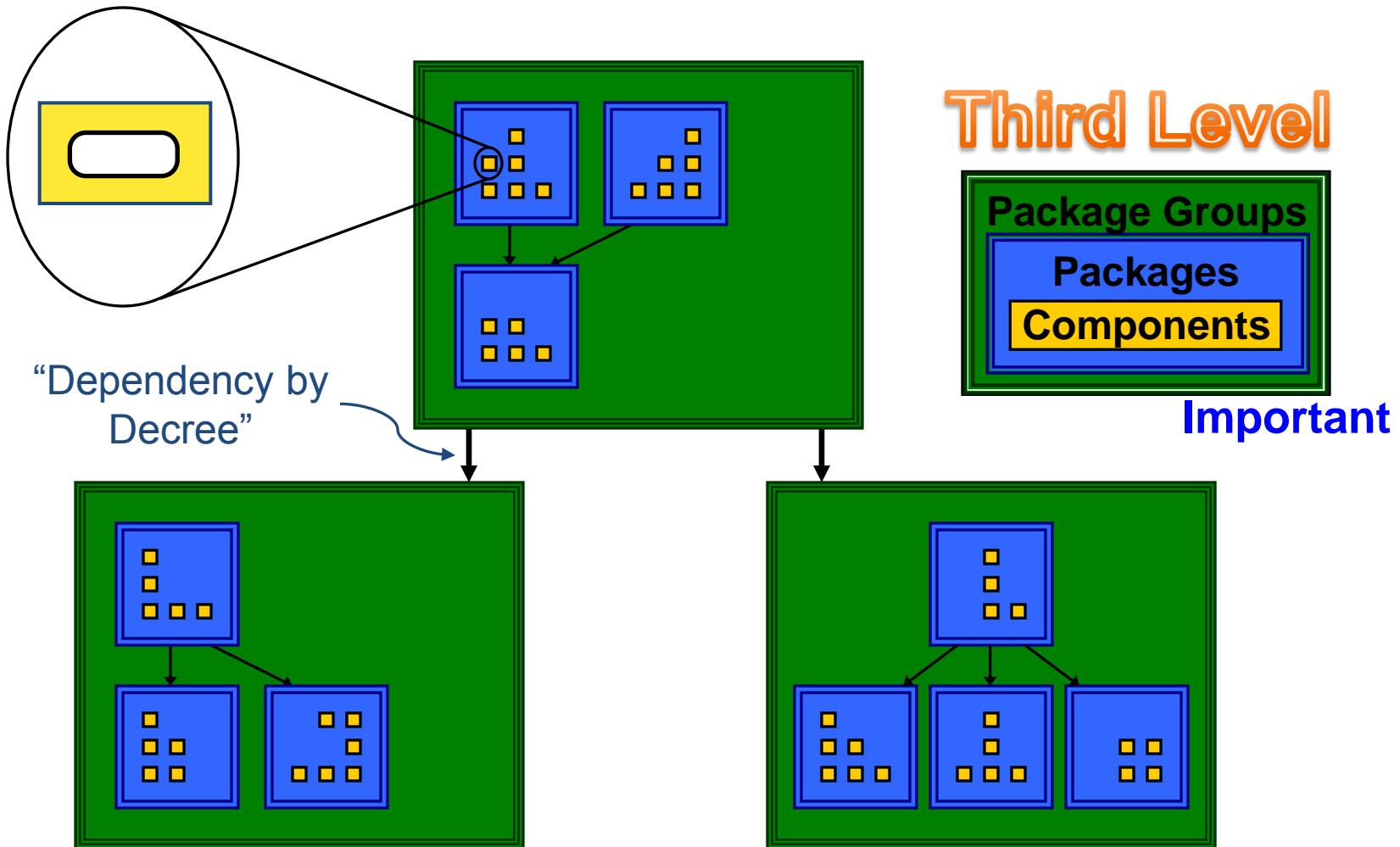
Aggregate Level 2:

Aggregate Level 1:



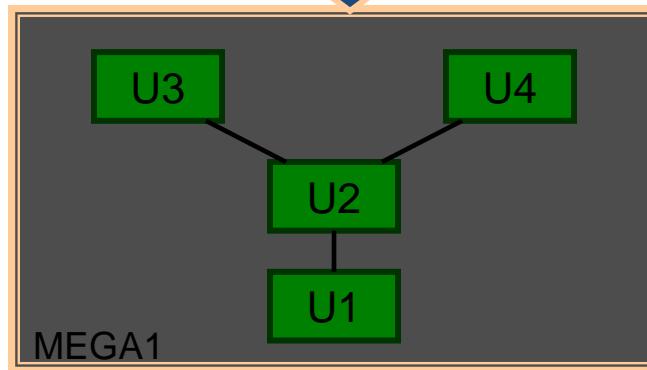
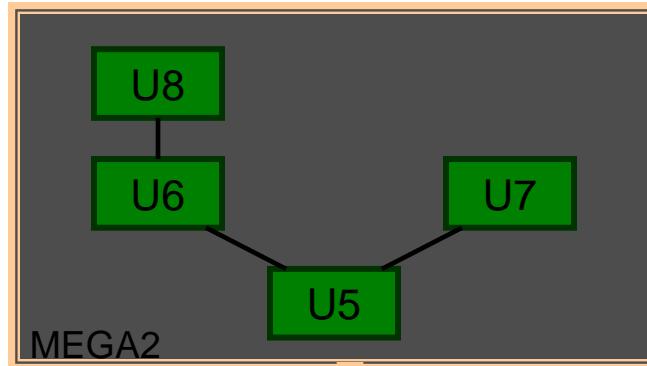
1. Process & Architecture

Package Groups



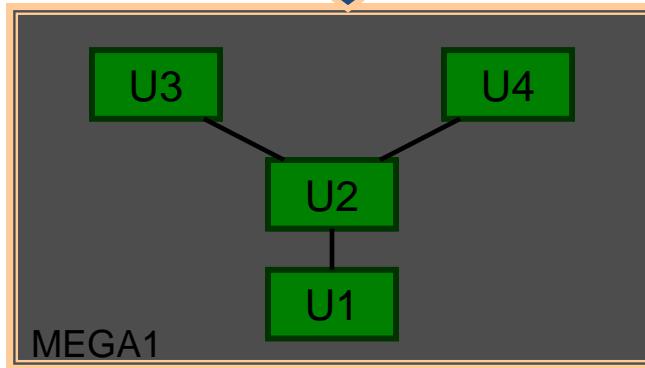
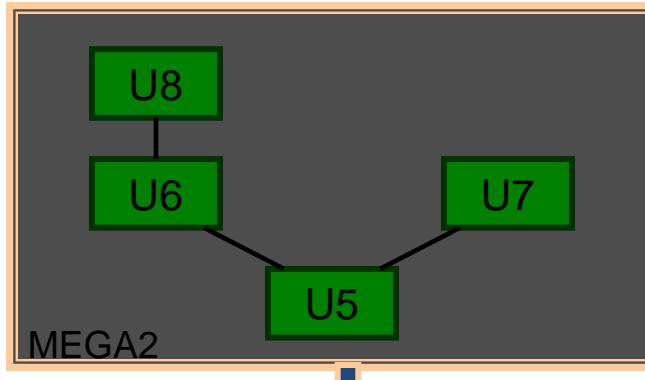
1. Process & Architecture

What About a Fourth-Level Aggregate?

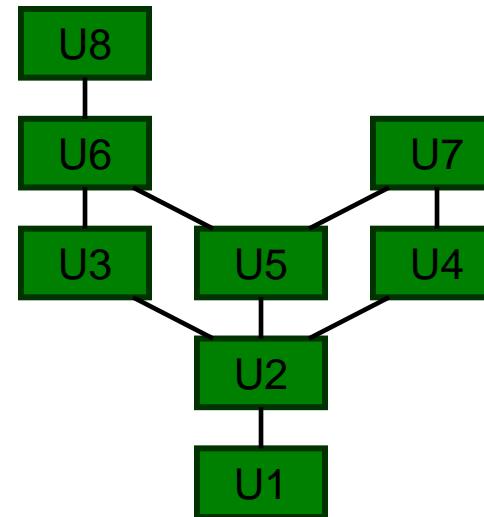


1. Process & Architecture

What About a Fourth-Level Aggregate?

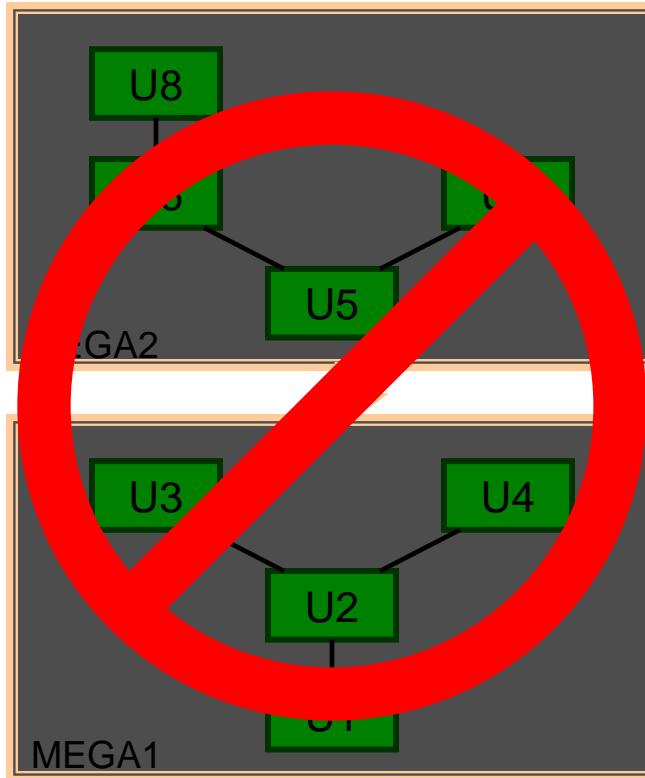


We find **two** or **three** levels of aggregation per library sufficient.

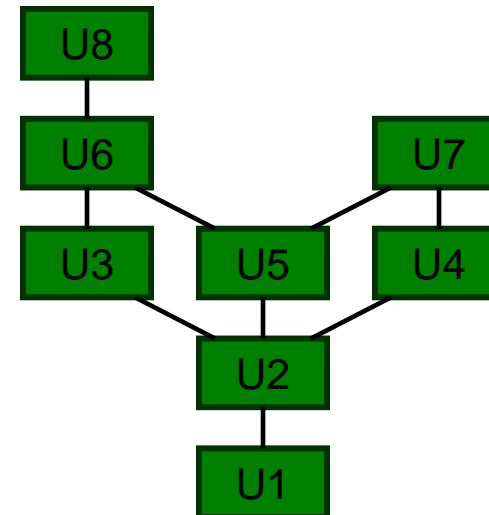


1. Process & Architecture

What About a Fourth-Level Aggregate?



We find **two** or **three** levels of aggregation per library sufficient.



Organizing Principles

Sound Physical Design:

- Regular, Fine-Grained Physical Packaging.
- Uniform Depth of Physical Aggregation.
- Logical/Physical Synergy.

Organizing Principles

Sound Physical Design:

- Regular, Fine-Grained Physical Packaging.
- Uniform Depth of Physical Aggregation.
- Logical/Physical Synergy.

1. Process & Architecture

Logical/Physical Synergy

There are two distinct aspects:

1. Logical/Physical Coherence

- ❖ Each logical subsystem is tightly encapsulated by a corresponding physical aggregate.

2. Logical/Physical Name Cohesion

- ❖ The precise physical location of the definition of a logical construct can be determined directly from its point of use (i.e., its **qualified** name).

1. Process & Architecture

Logical/Physical Synergy

There are two distinct aspects:

1. Logical/Physical Coherence

- ❖ Each logical subsystem is tightly encapsulated by a corresponding physical aggregate.

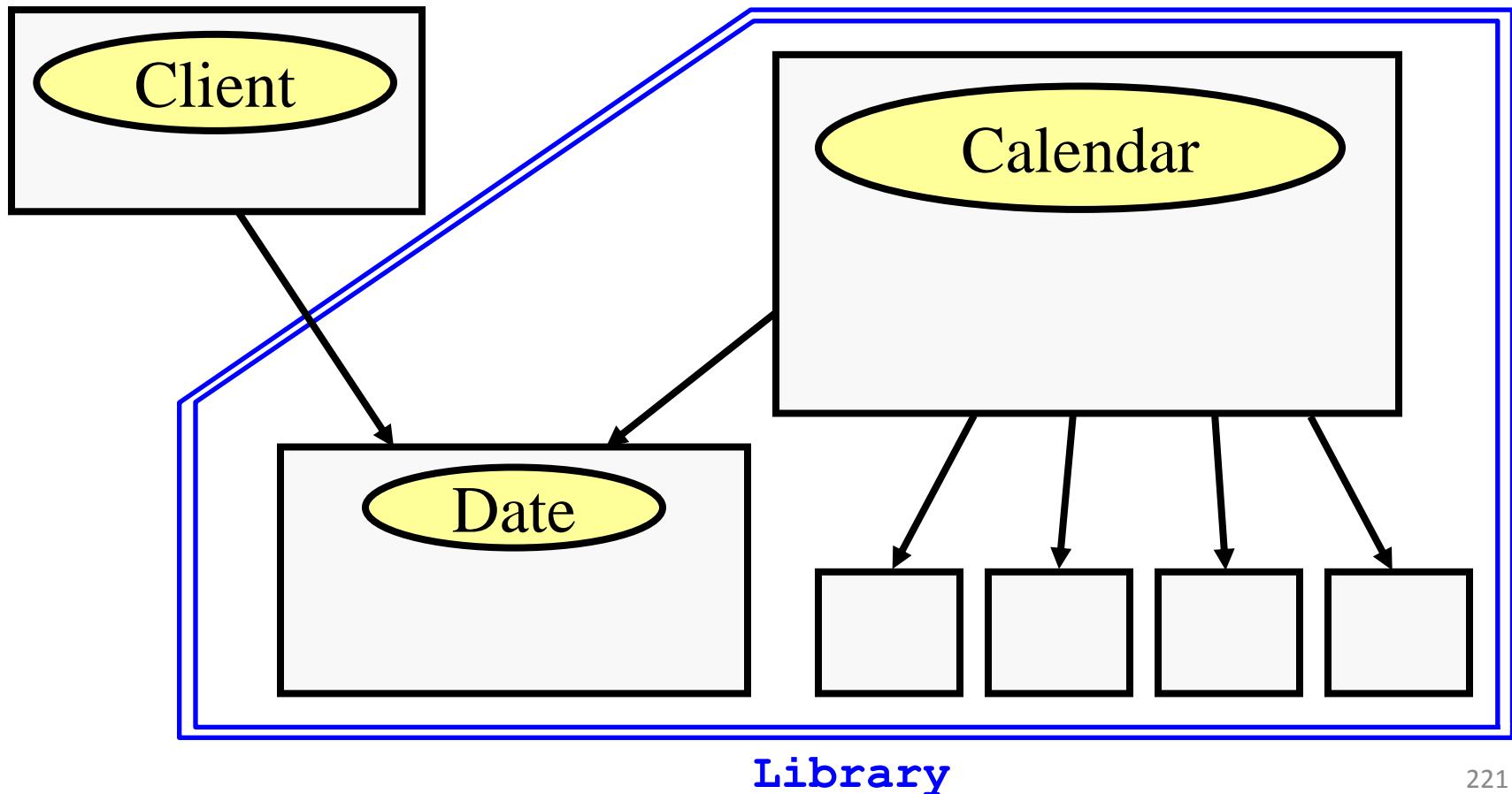
2. Logical/Physical Name Cohesion

- ❖ The precise physical location of the definition of a logical construct can be determined directly from its point of use (i.e., its *qualified* name).

1. Process & Architecture

Logical/Physical Incoherence

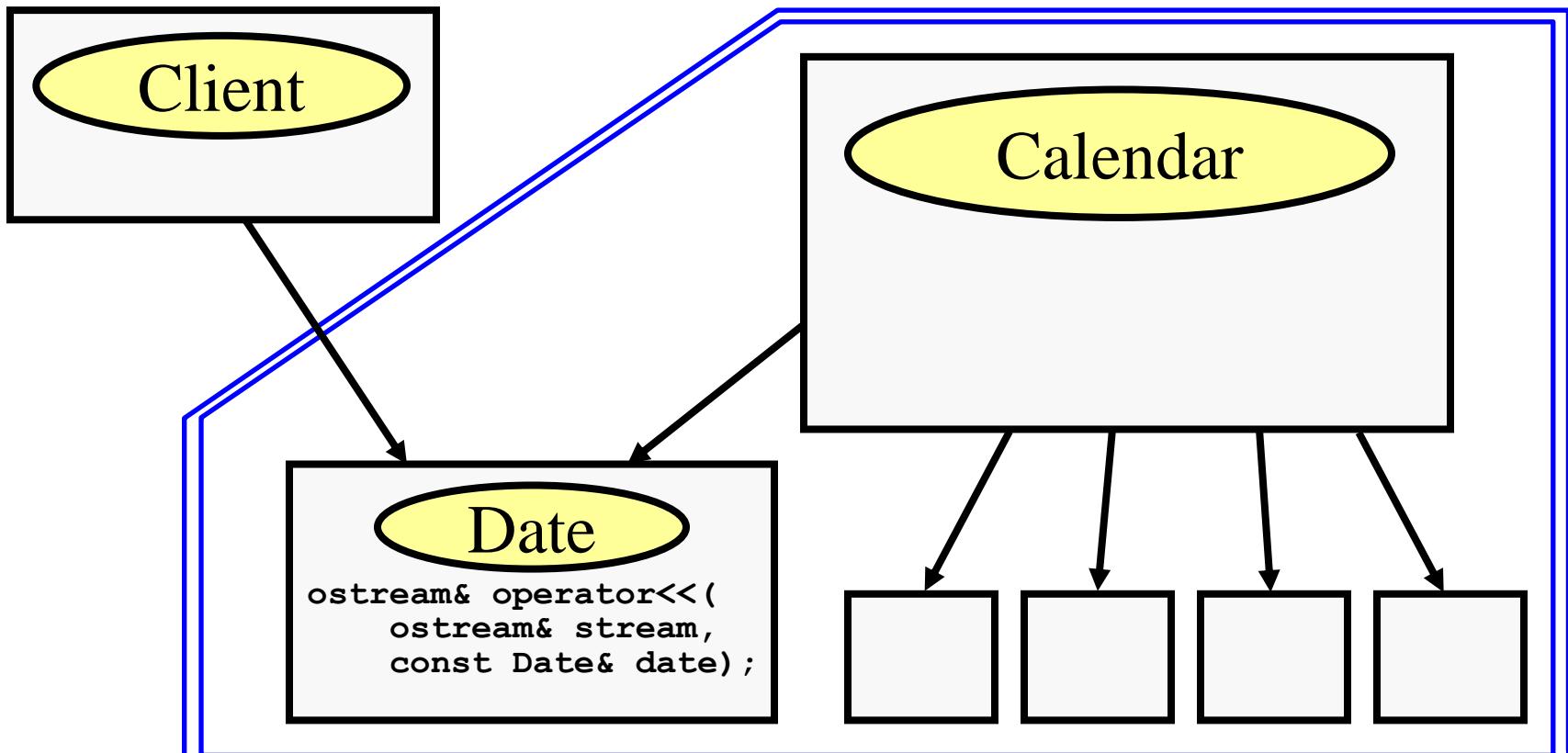
A Component Defines Only What It Declares.



1. Process & Architecture

Logical/Physical Incoherence

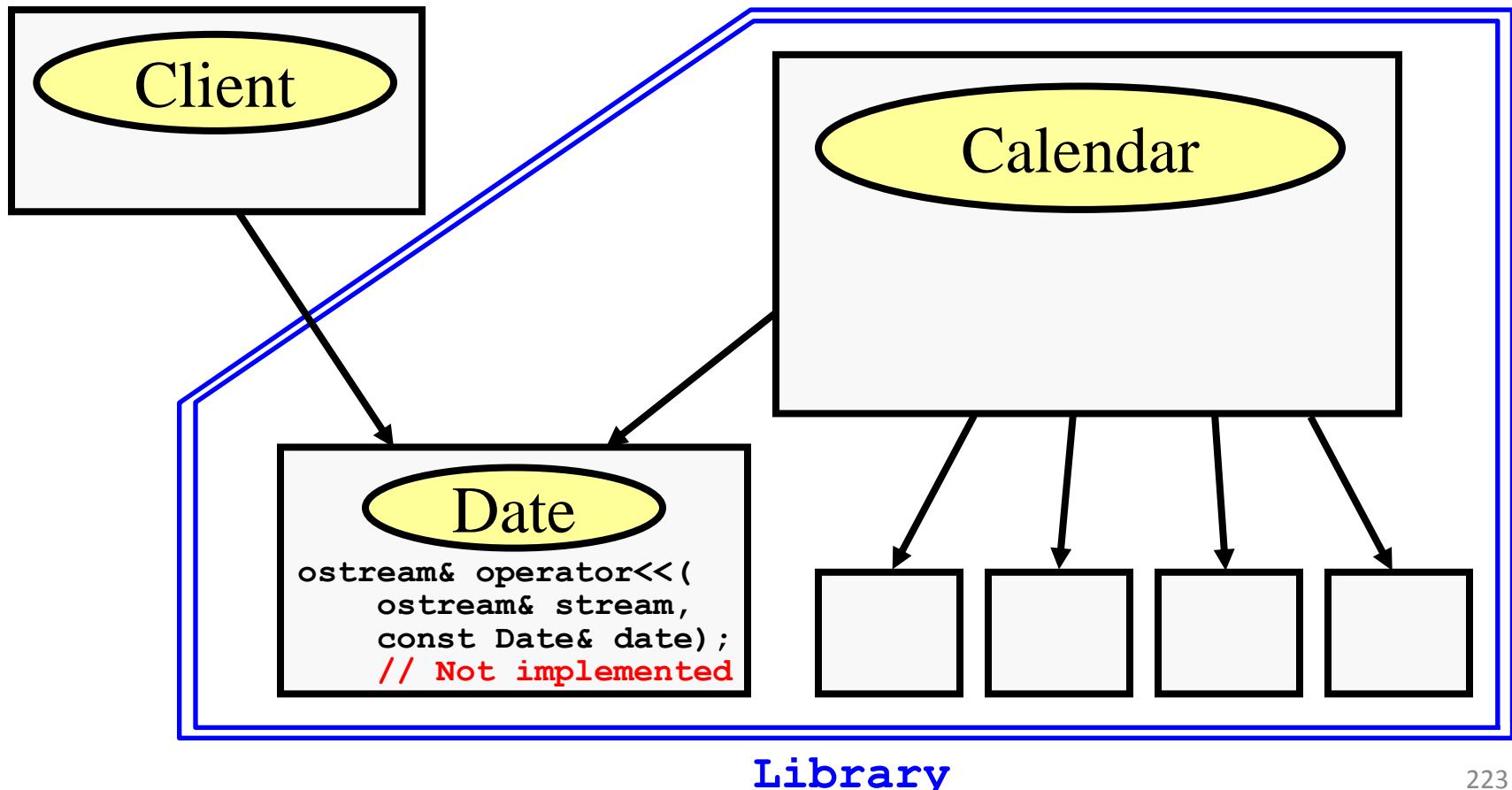
A Component Defines Only What It Declares.



1. Process & Architecture

Logical/Physical Incoherence

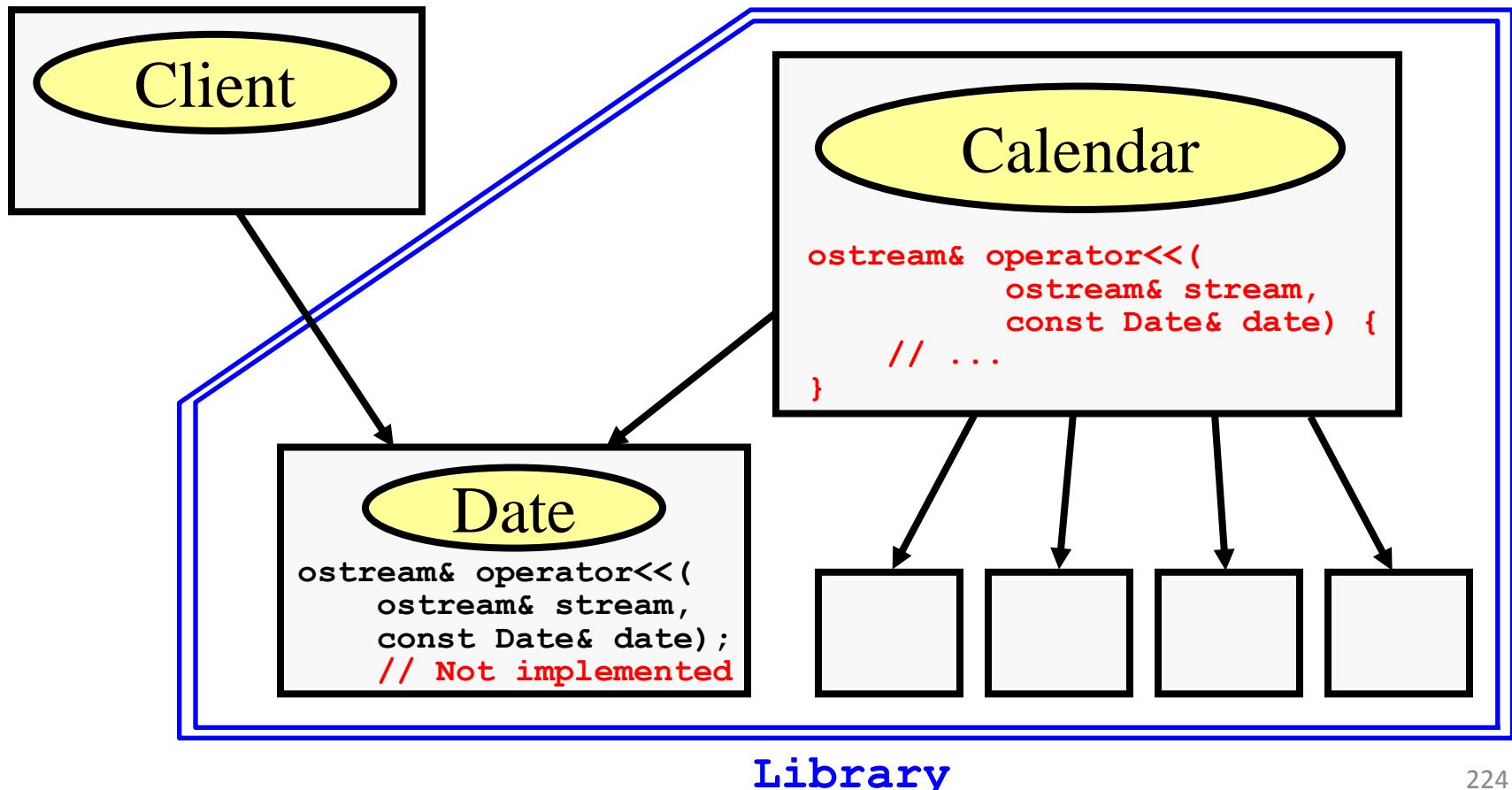
A Component Defines Only What It Declares.



1. Process & Architecture

Logical/Physical Incoherence

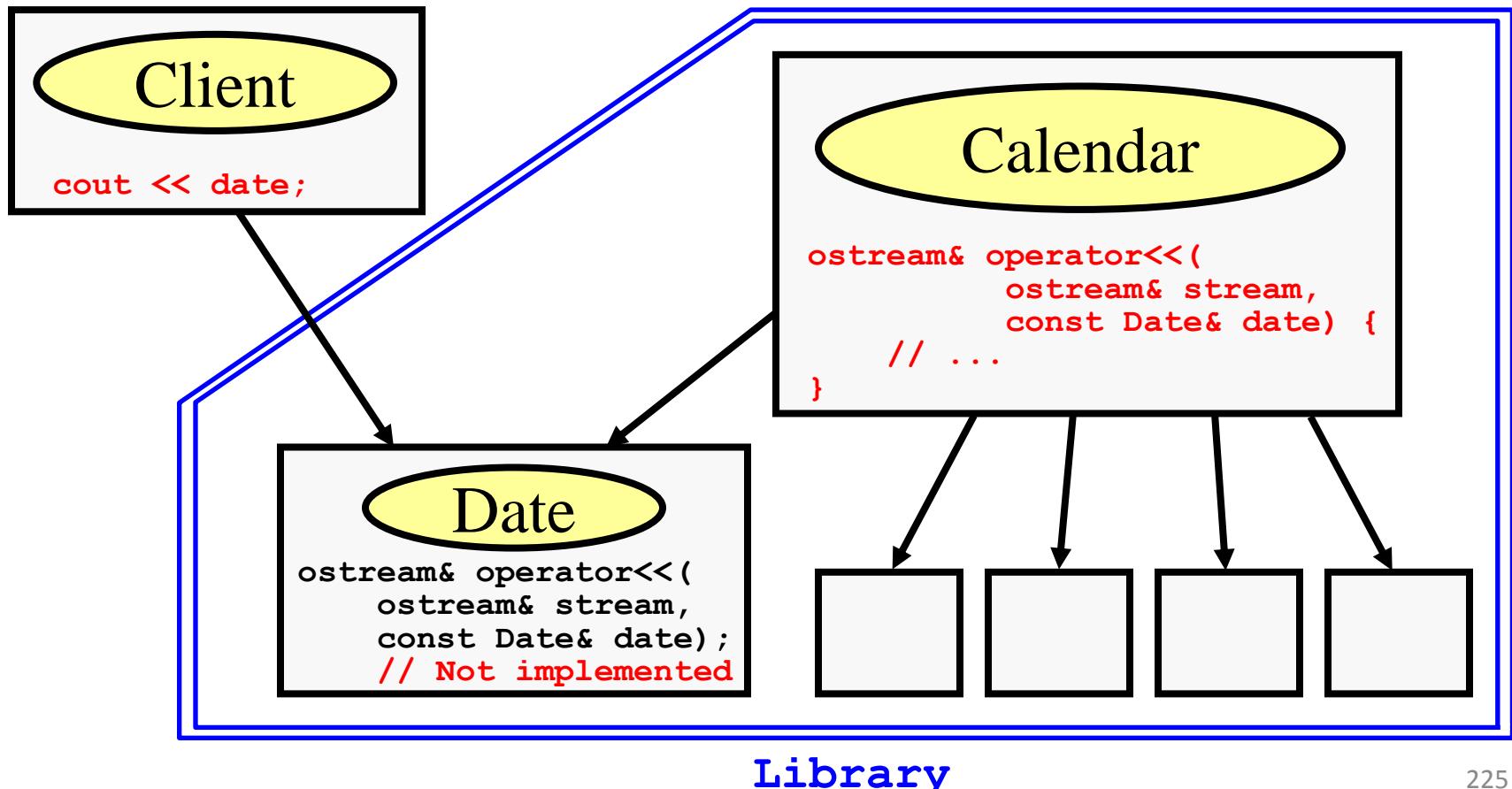
A Component Defines Only What It Declares.



1. Process & Architecture

Logical/Physical Incoherence

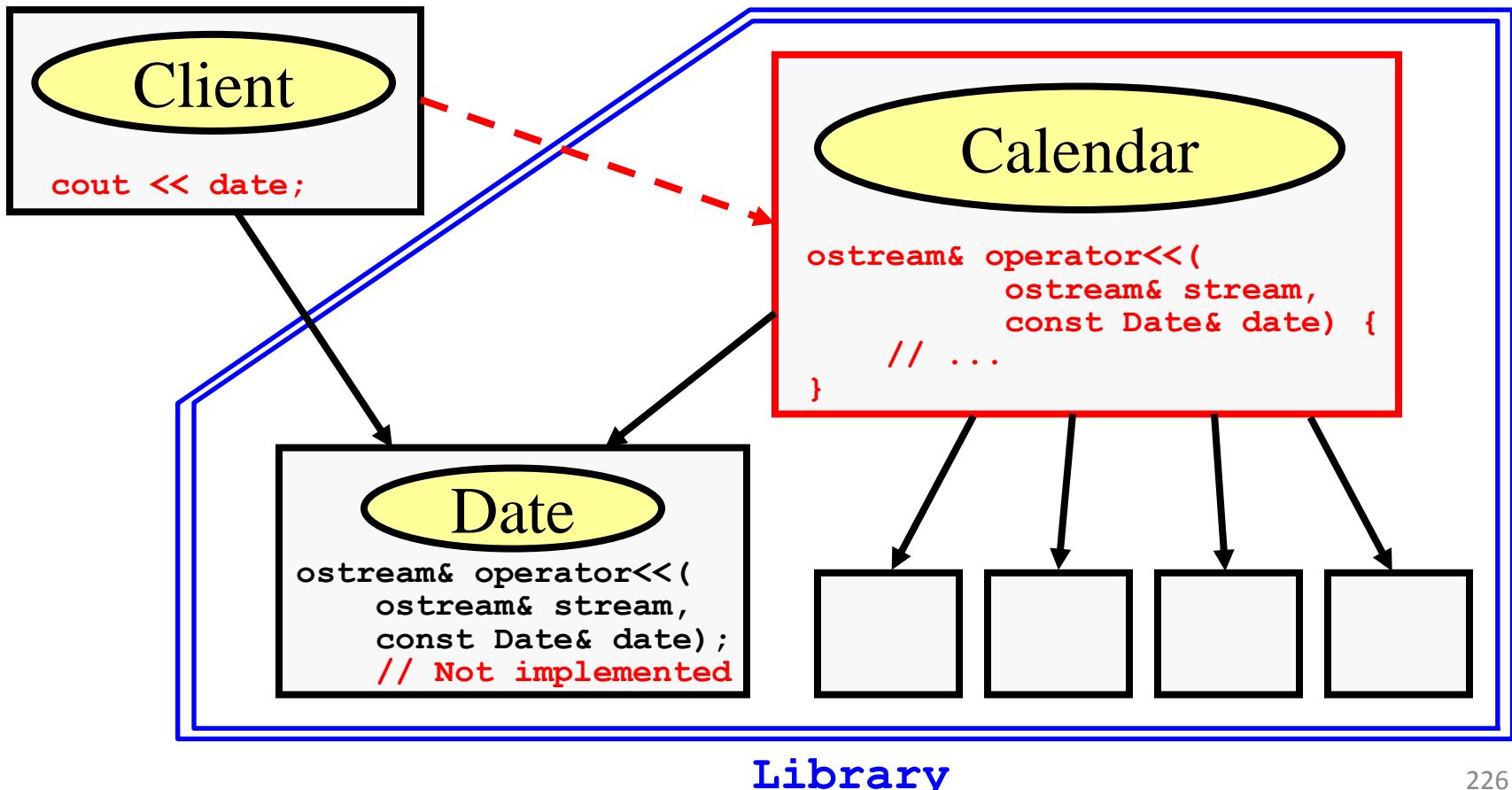
A Component Defines Only What It Declares.



1. Process & Architecture

Logical/Physical Incoherence

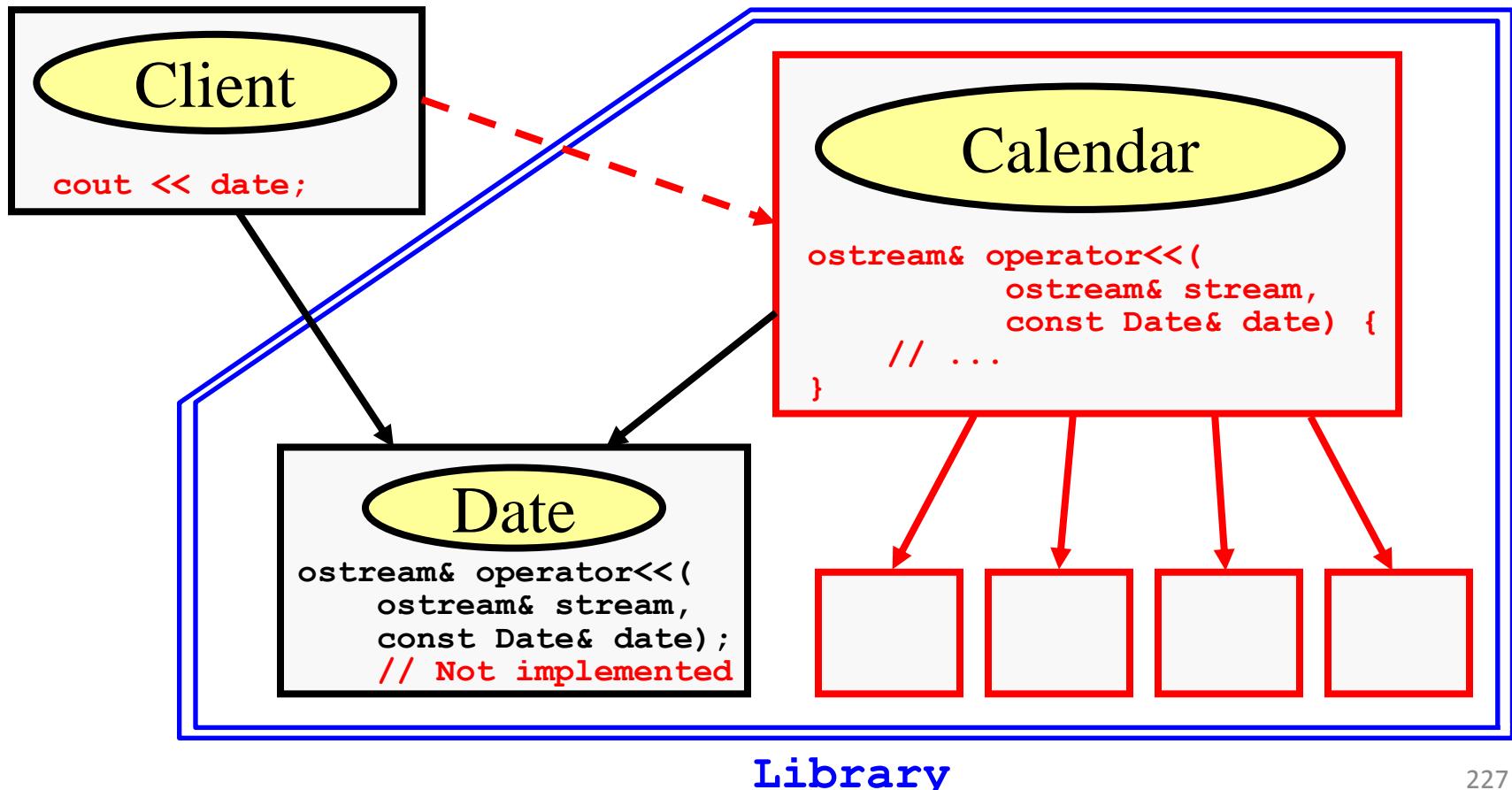
A Component Defines Only What It Declares.



1. Process & Architecture

Logical/Physical Incoherence

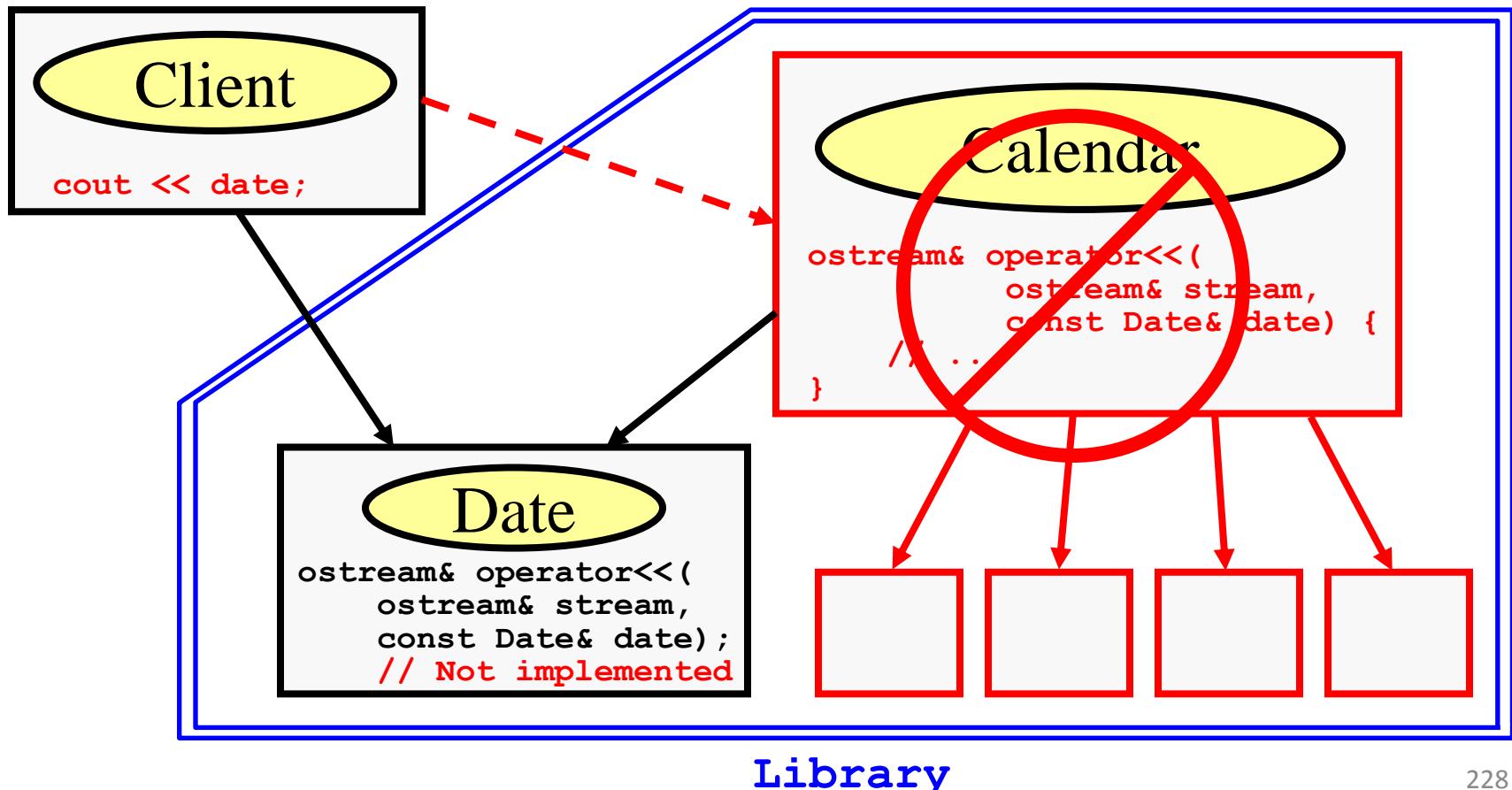
A Component Defines Only What It Declares.



1. Process & Architecture

Logical/Physical Incoherence

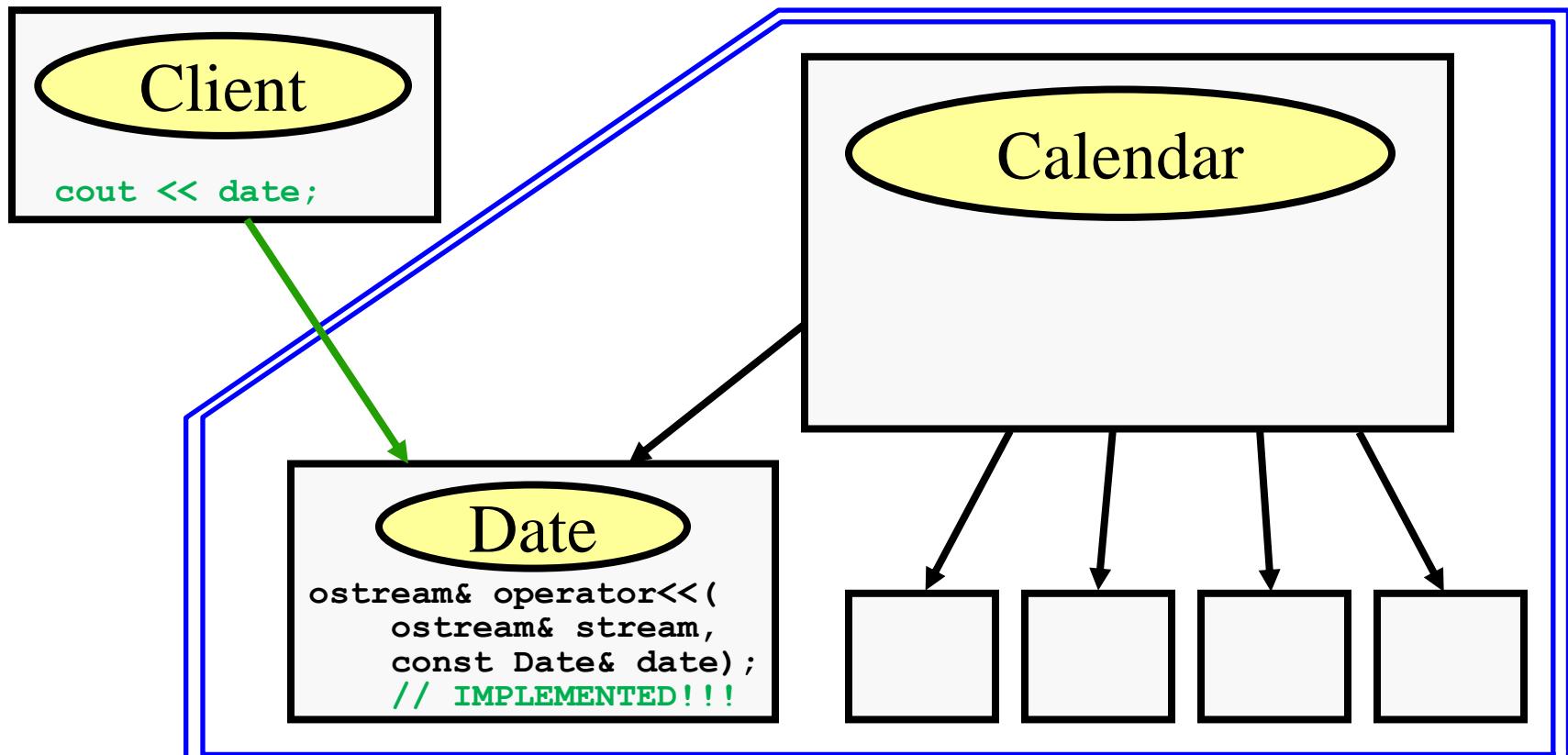
A Component Defines Only What It Declares.



1. Process & Architecture

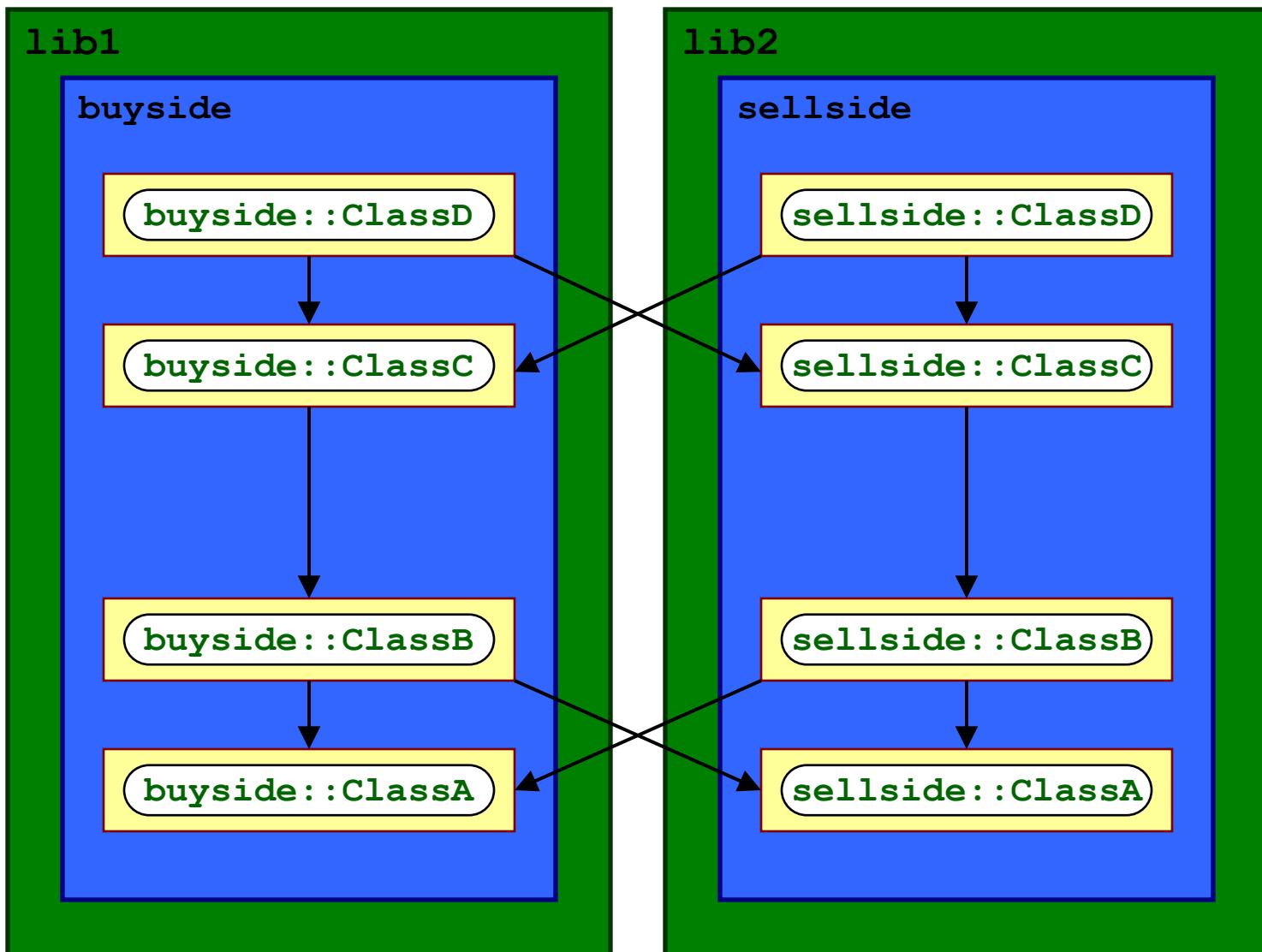
Logical/Physical Incoherence

A Component Defines Only What It Declares.



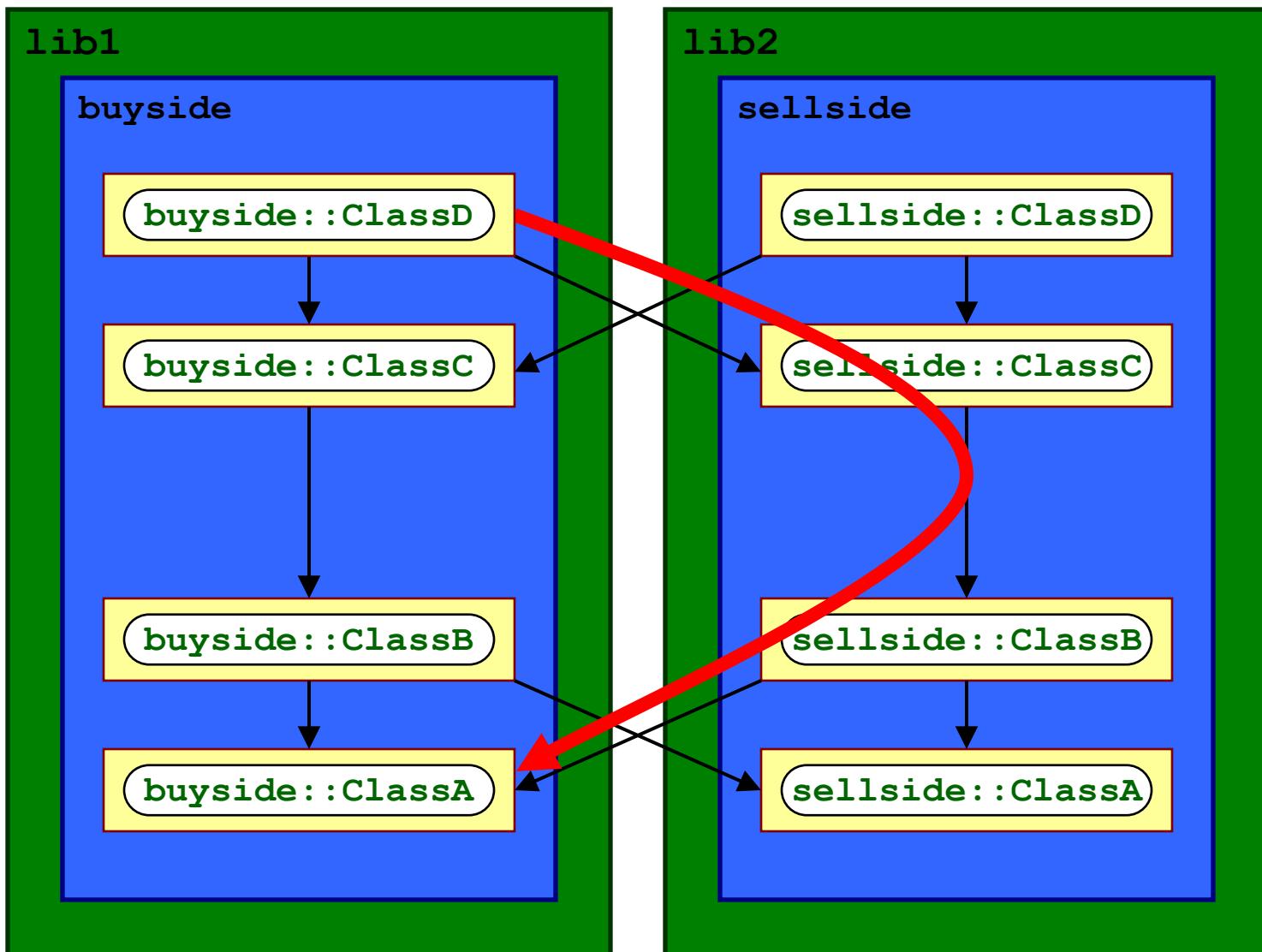
1. Process & Architecture

Logical/Physical Coherence



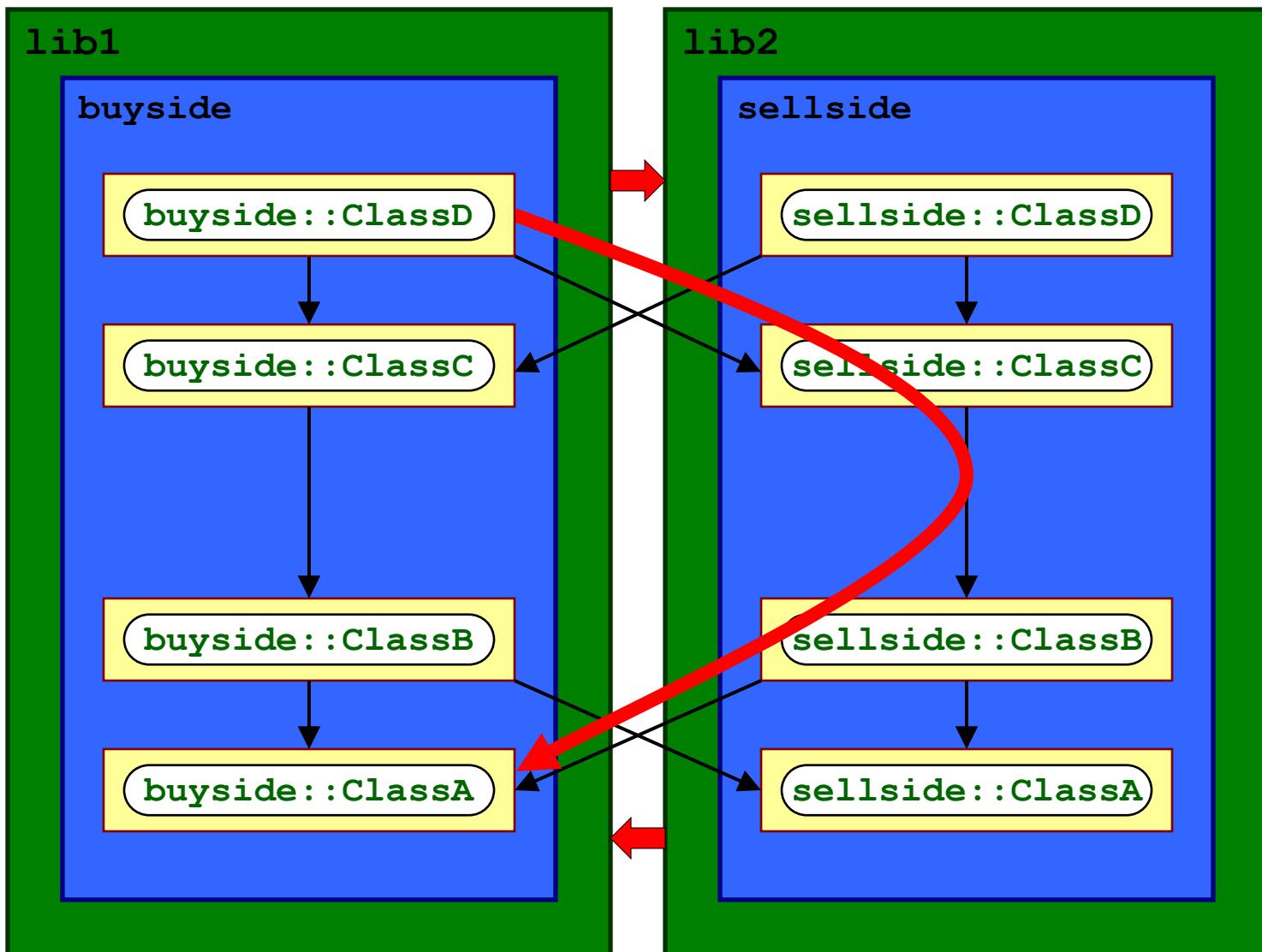
1. Process & Architecture

Logical/Physical Coherence



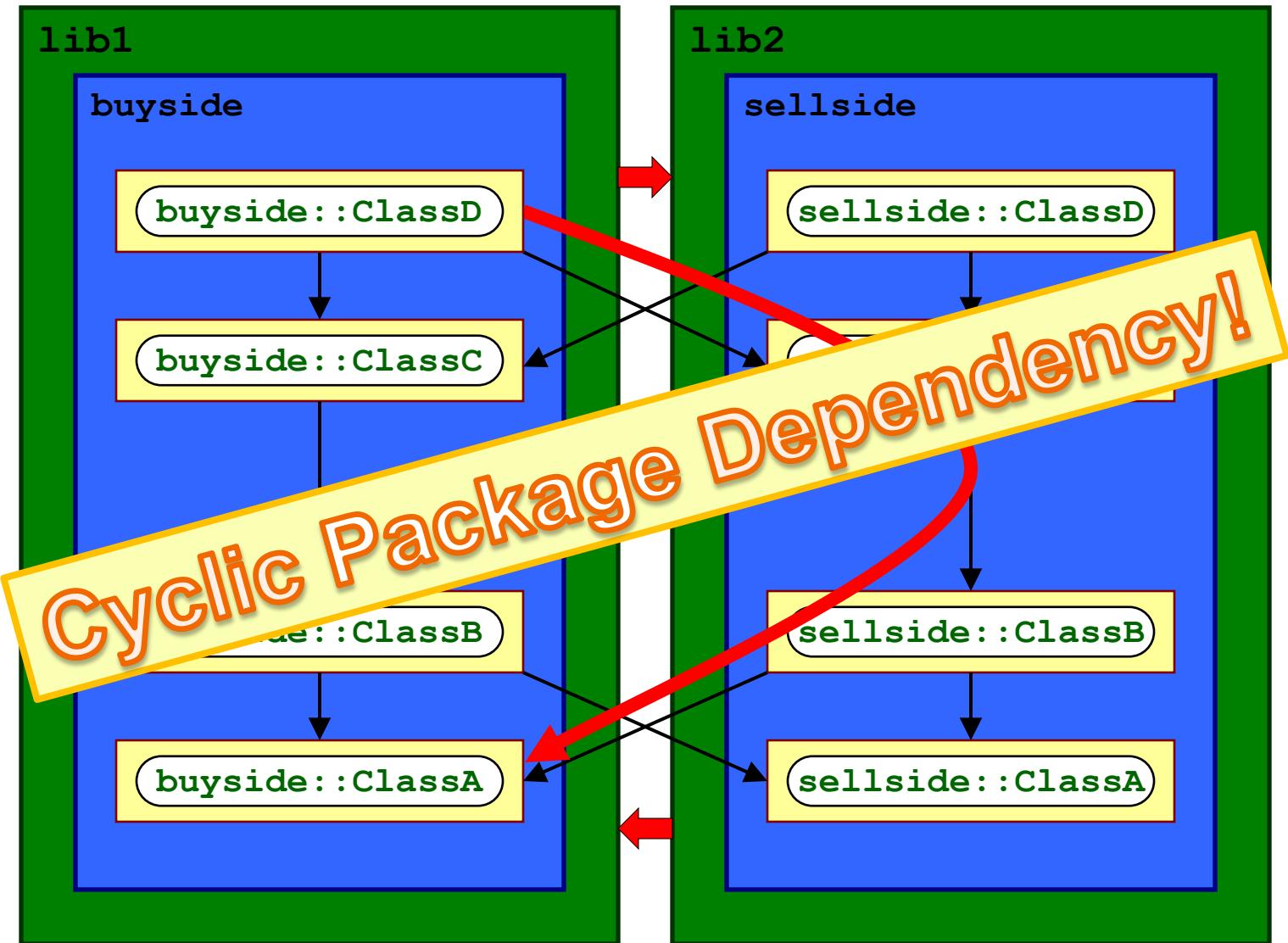
1. Process & Architecture

Logical/Physical Coherence



1. Process & Architecture

Logical/Physical Coherence



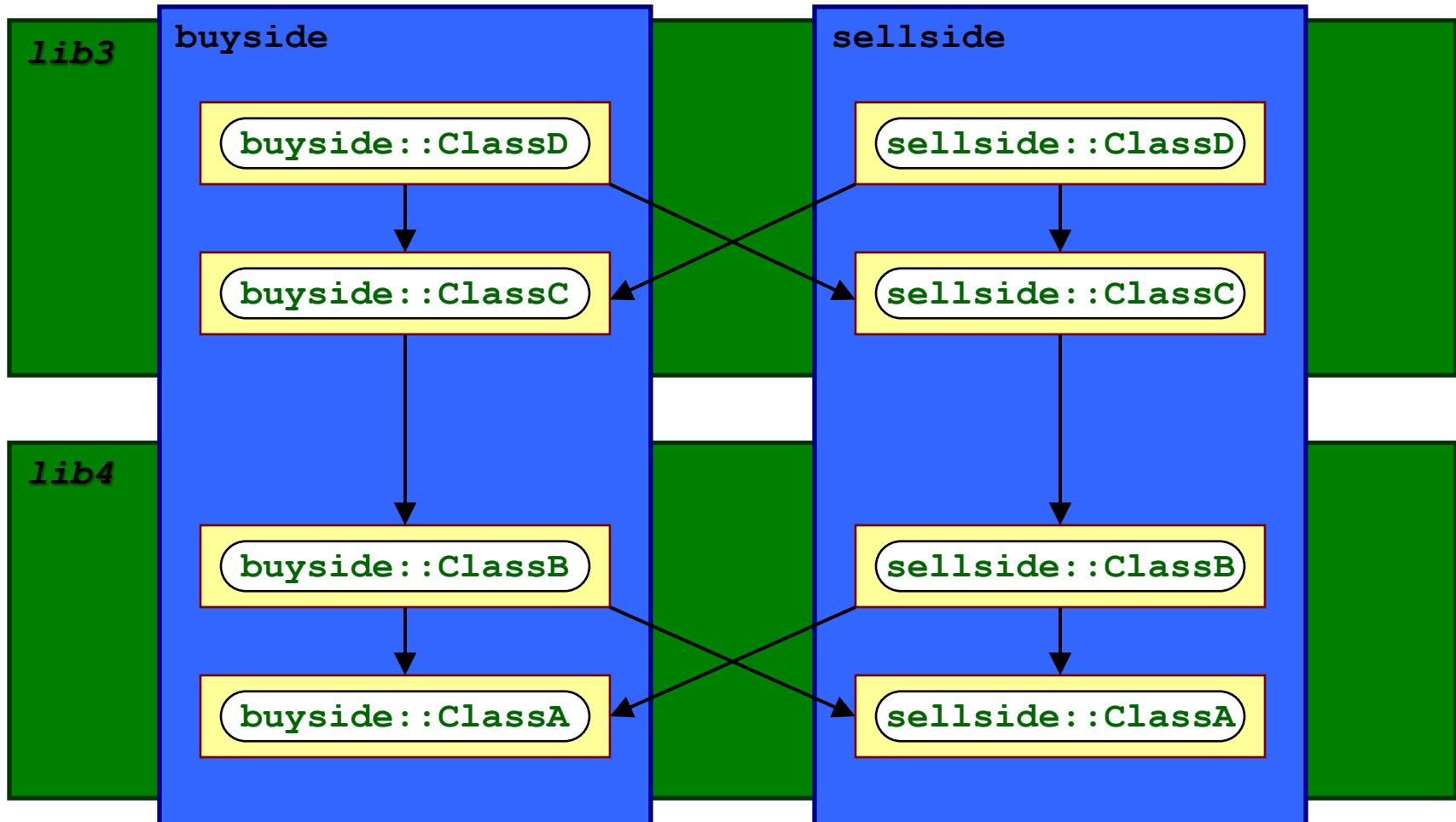
1. Process & Architecture

Logical/Physical Coherence



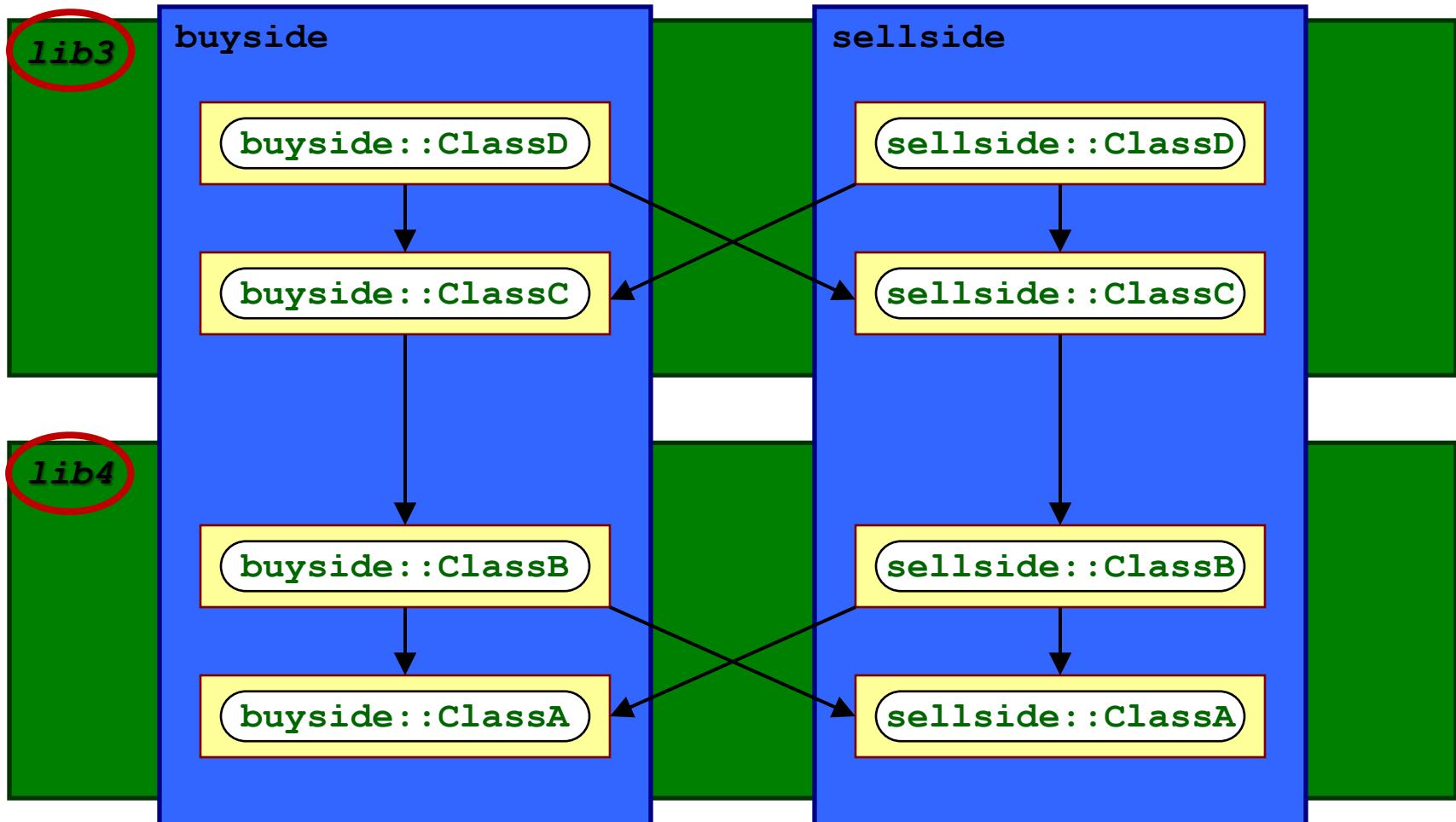
1. Process & Architecture

Logical/Physical Incoherence



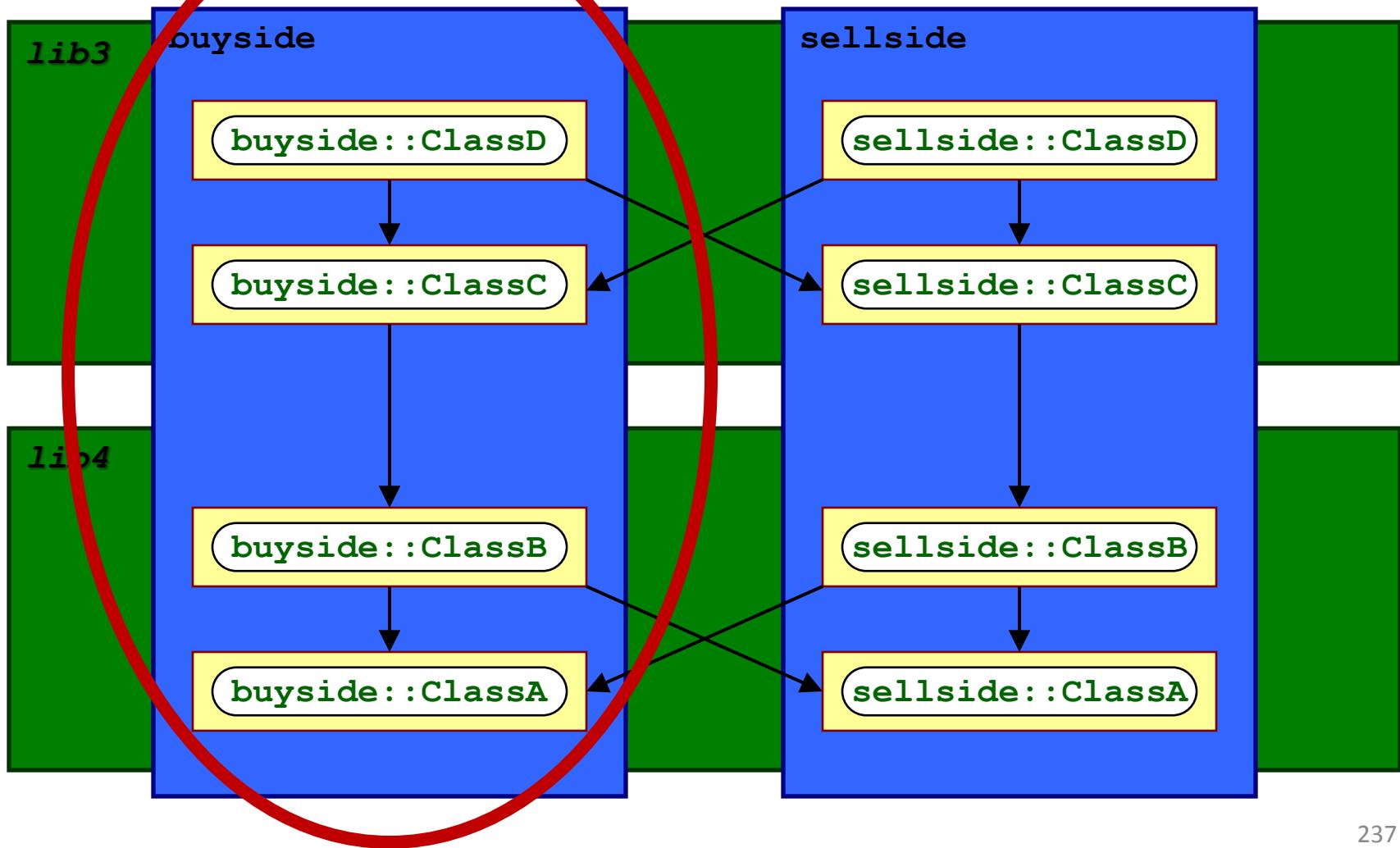
1. Process & Architecture

Logical/Physical Incoherence



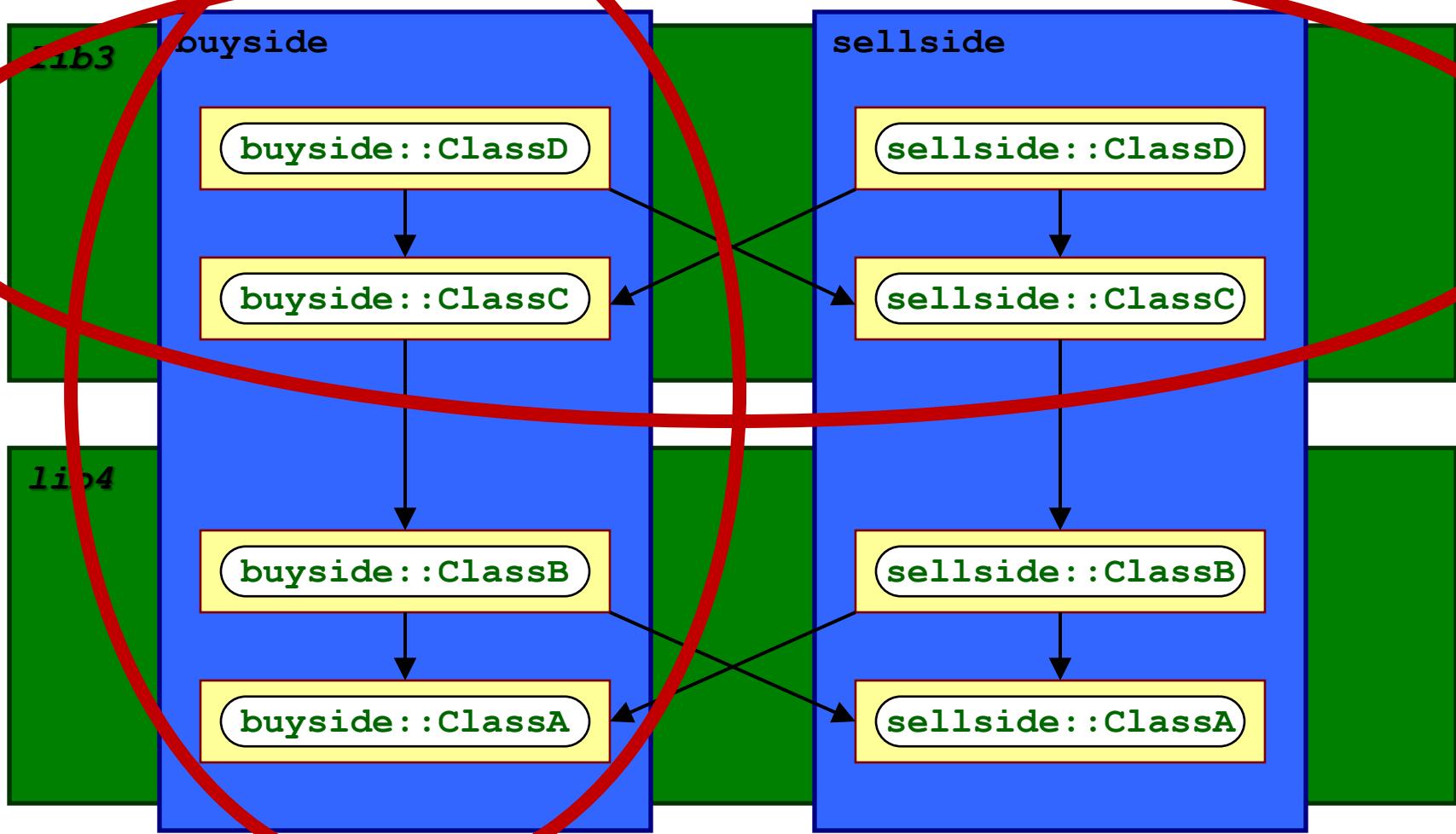
1. Process & Architecture

Logical/Physical Incoherence



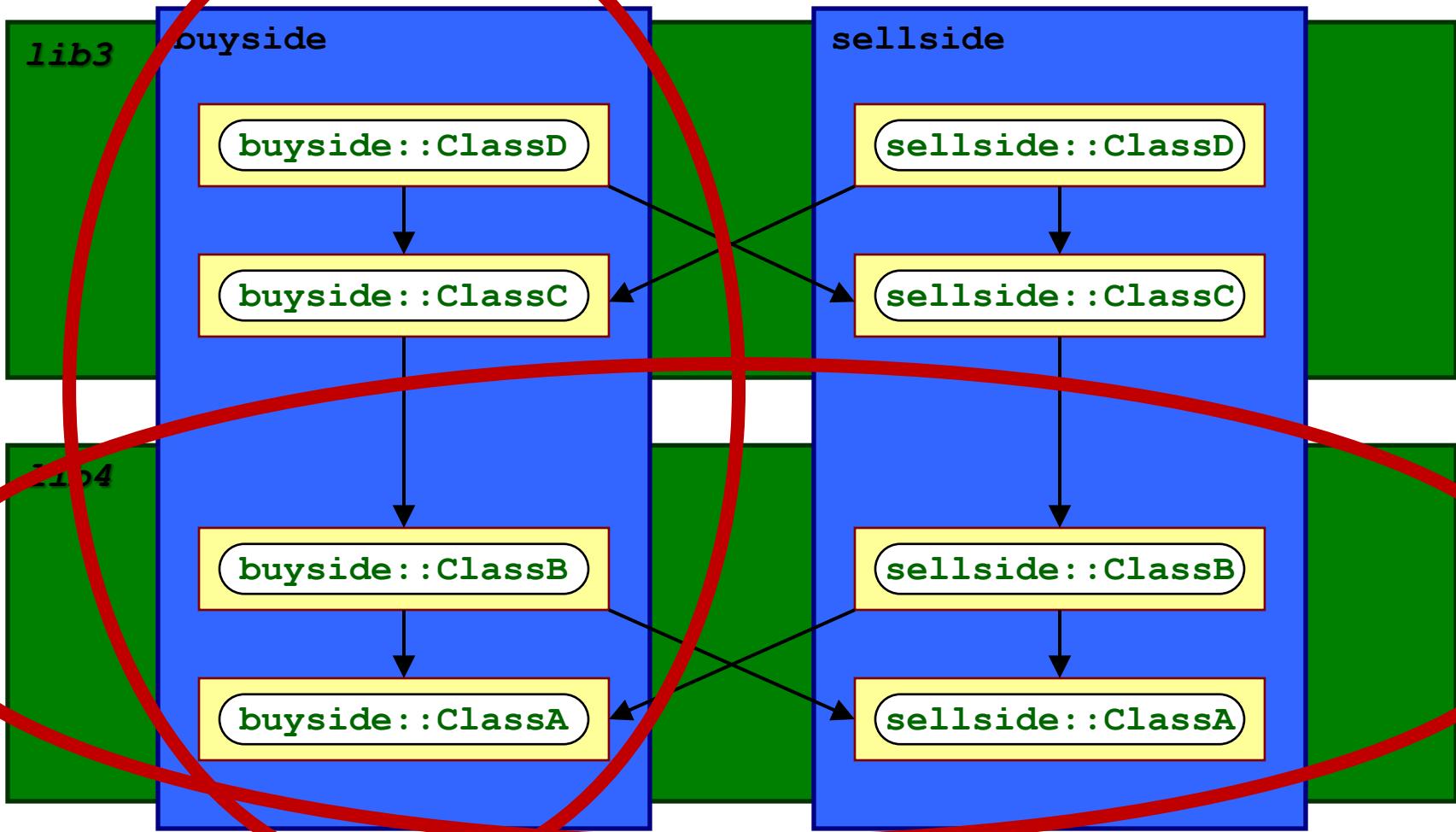
1. Process & Architecture

Logical/Physical Incoherence



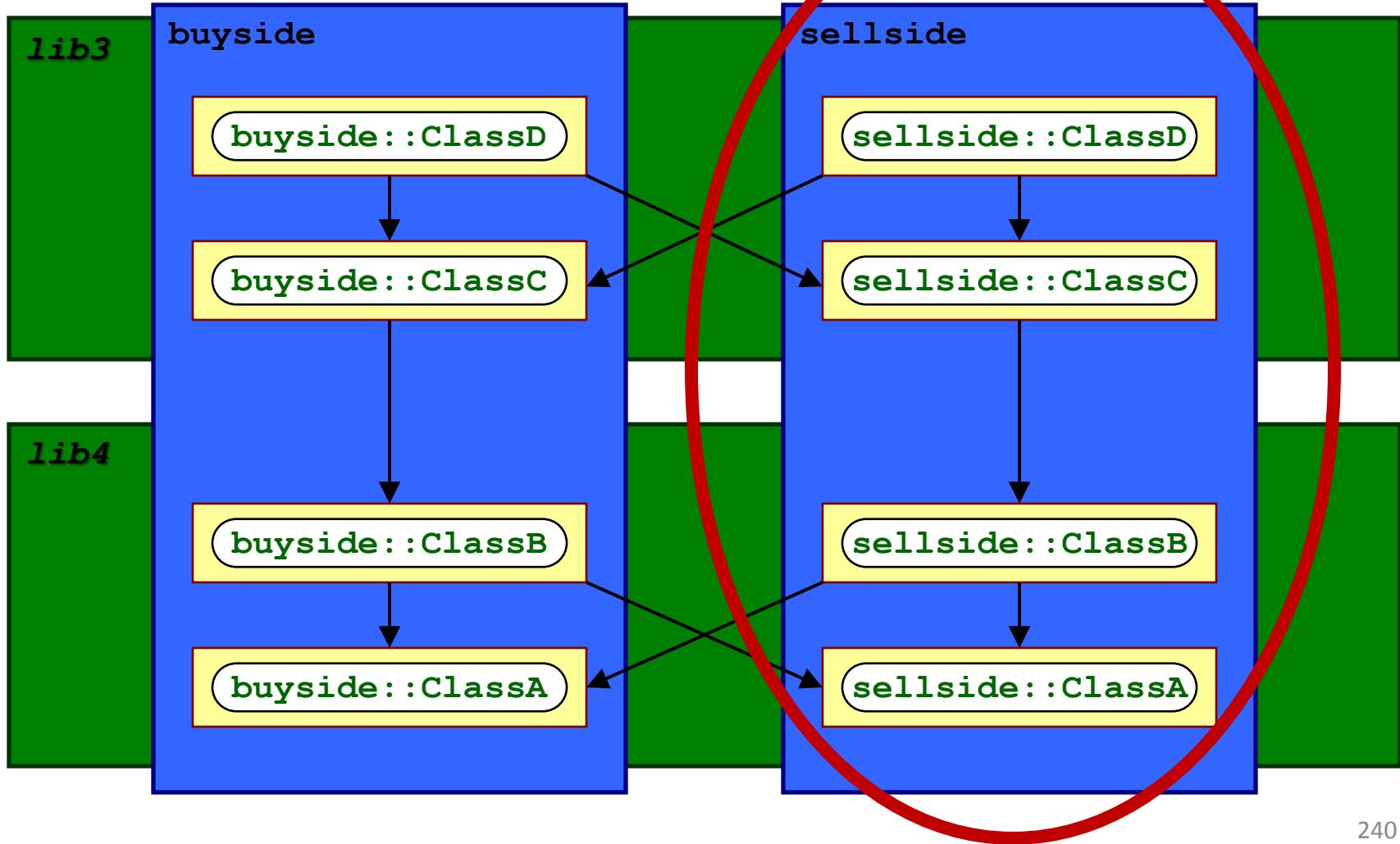
1. Process & Architecture

Logical/Physical Incoherence



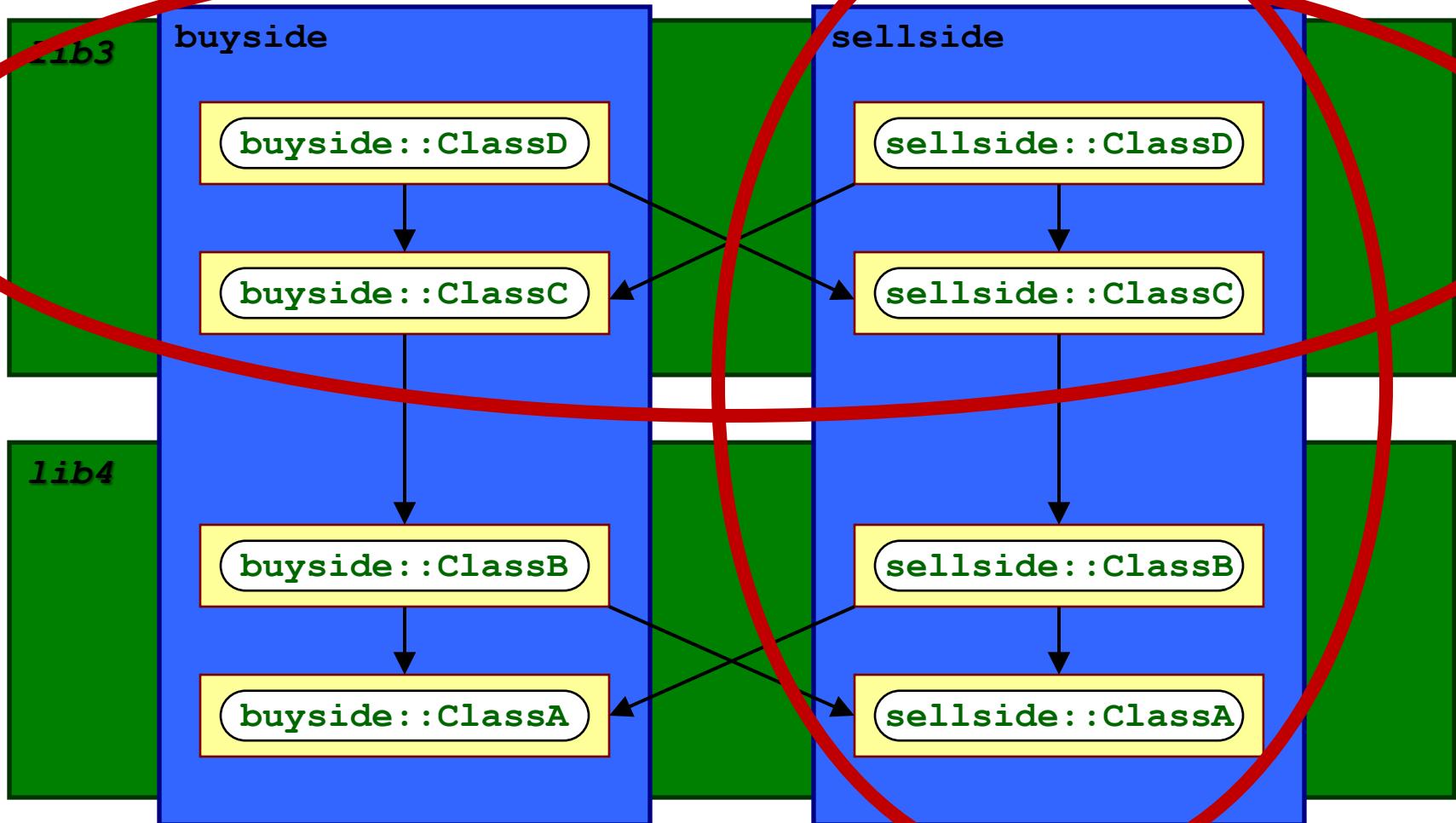
1. Process & Architecture

Logical/Physical Incoherence



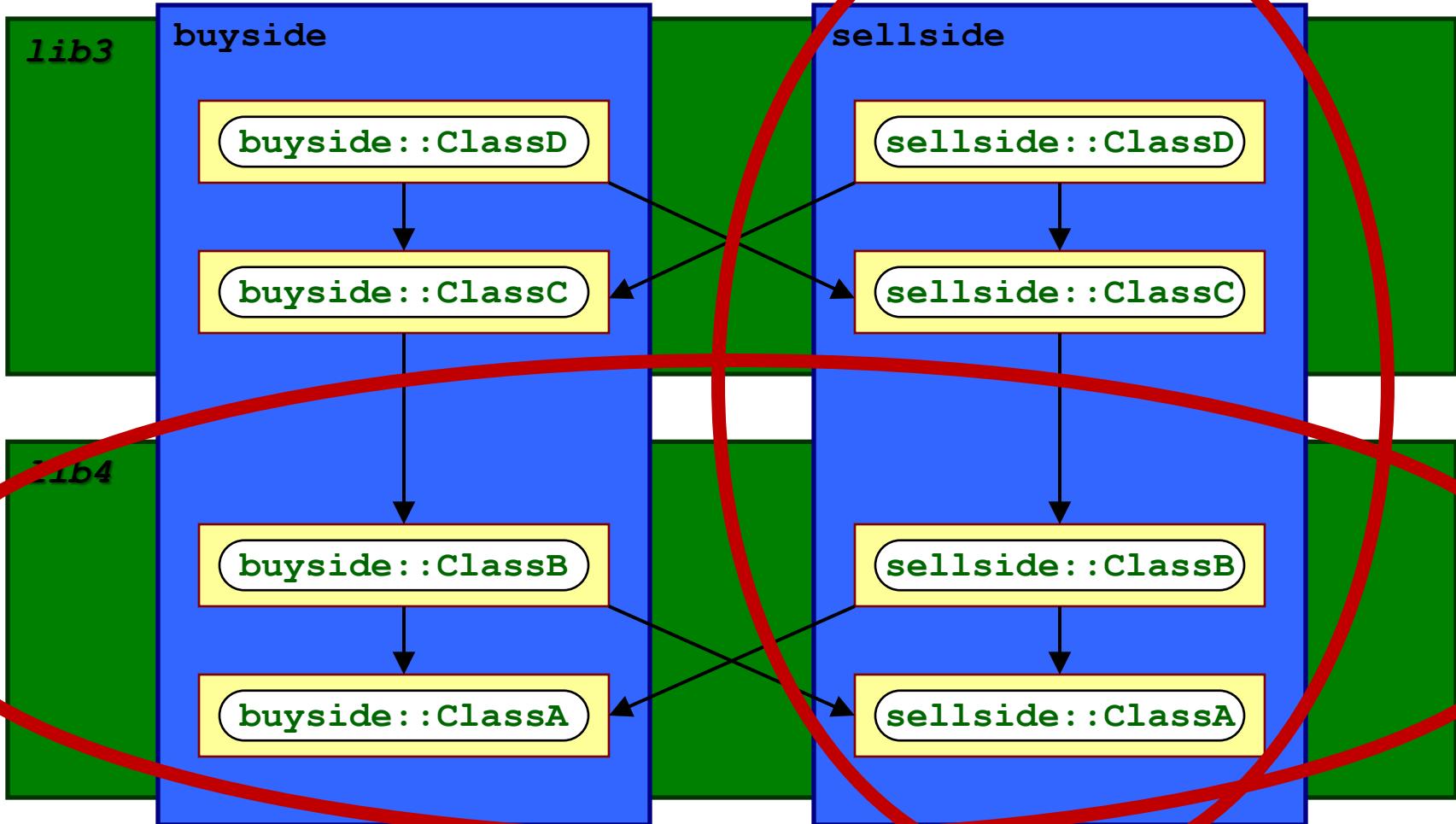
1. Process & Architecture

Logical/Physical Incoherence



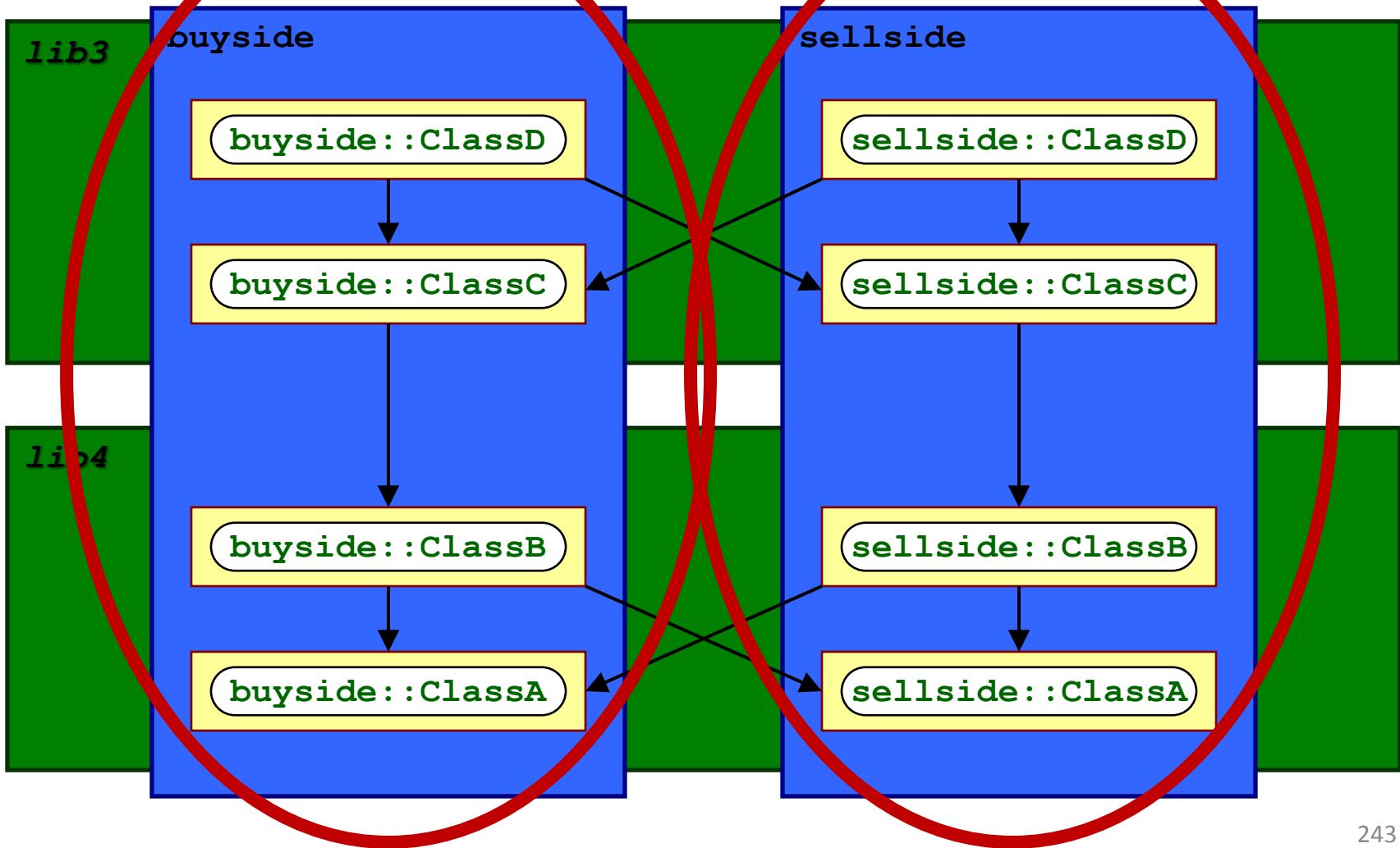
1. Process & Architecture

Logical/Physical Incoherence



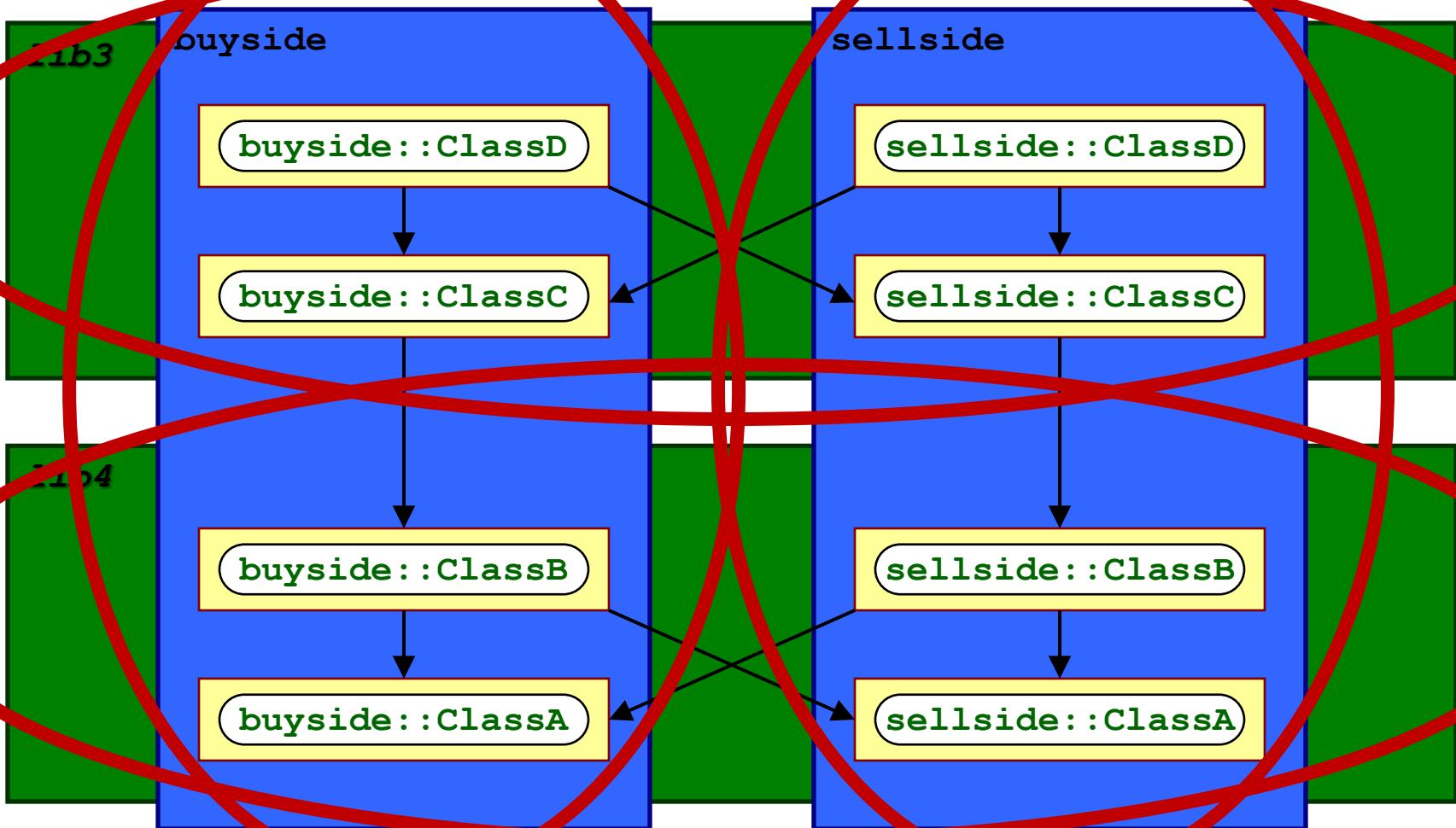
1. Process & Architecture

Logical/Physical Incoherence



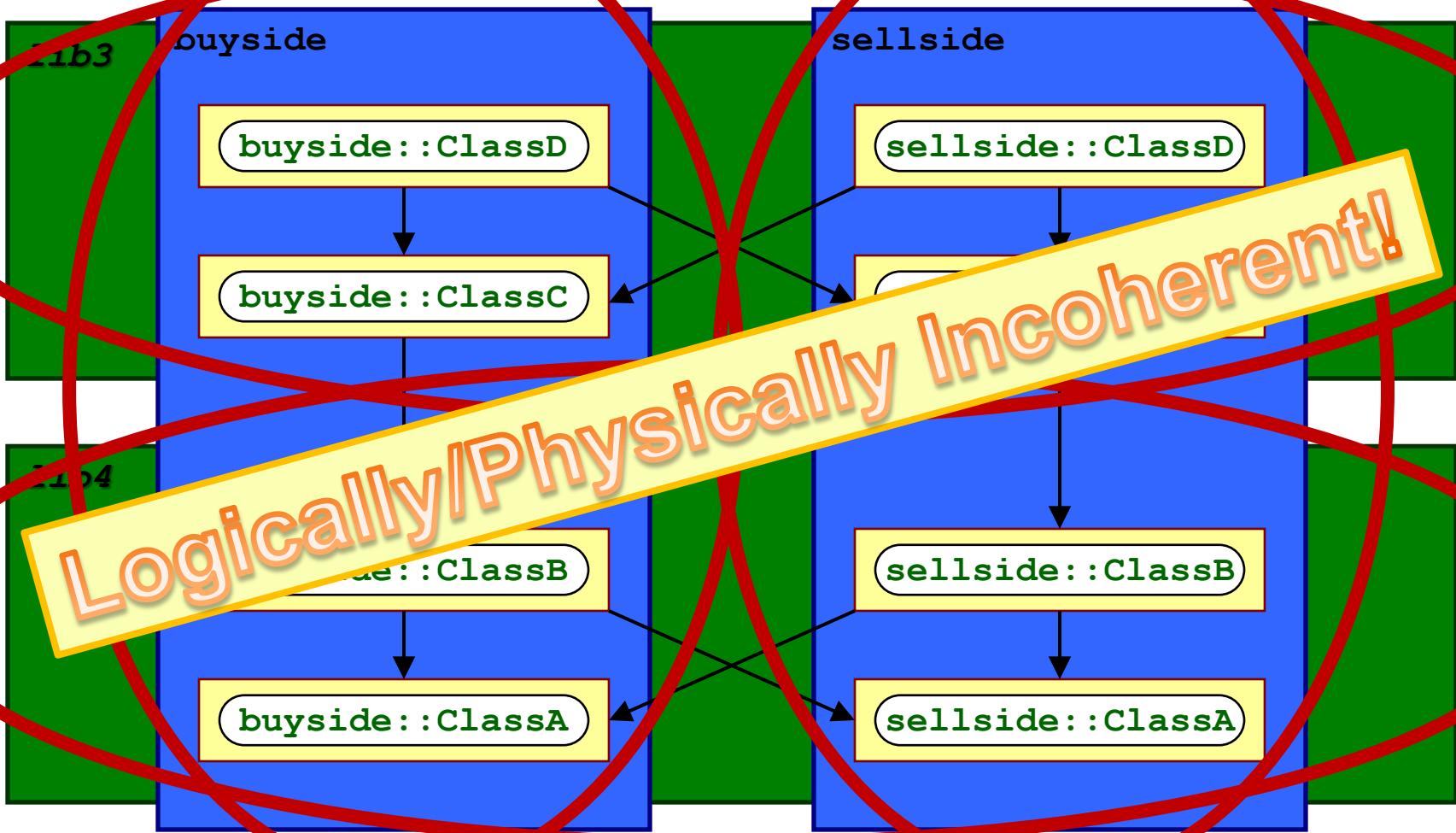
1. Process & Architecture

Logical/Physical Incoherence



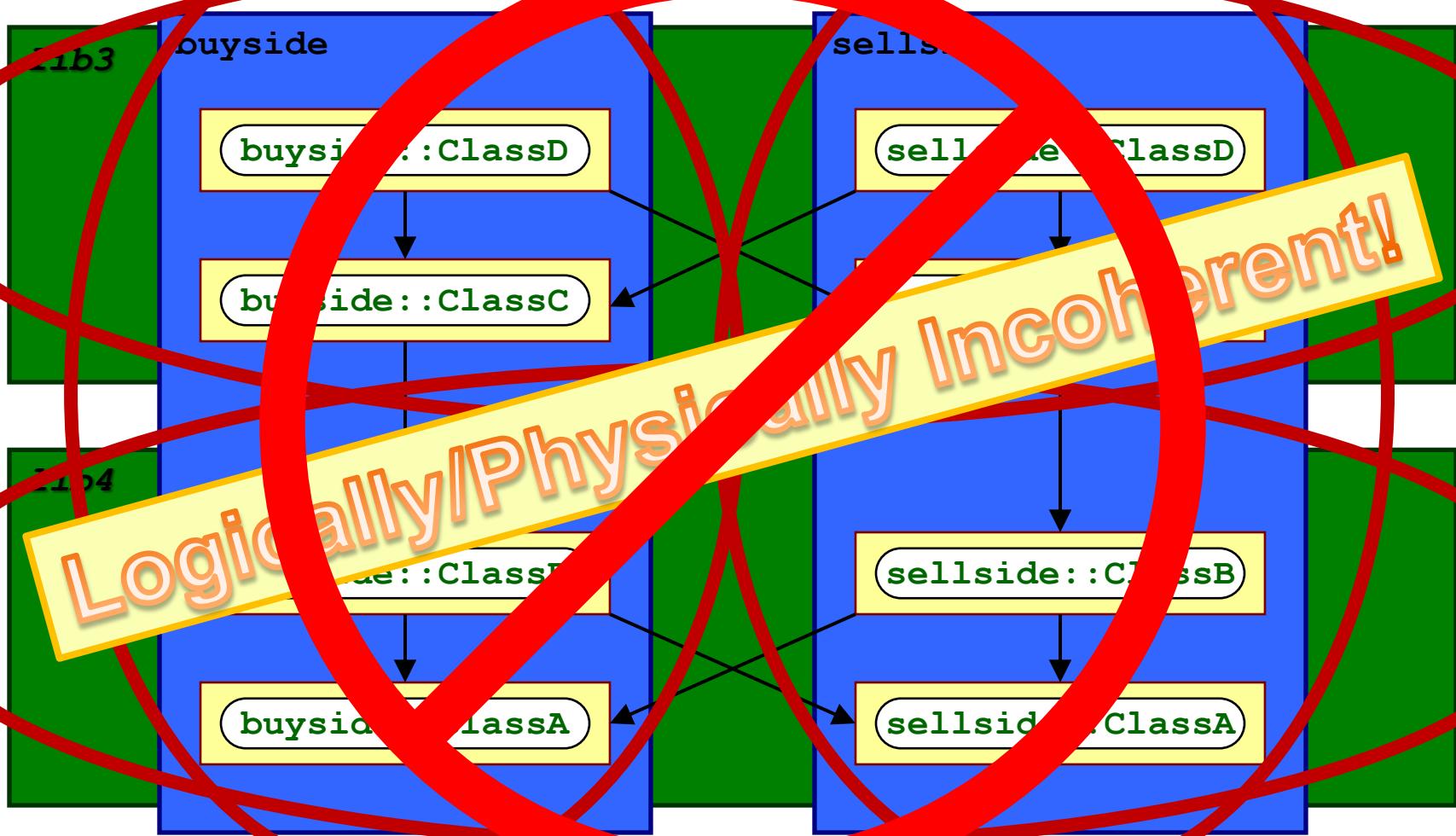
1. Process & Architecture

Logical/Physical Incoherence



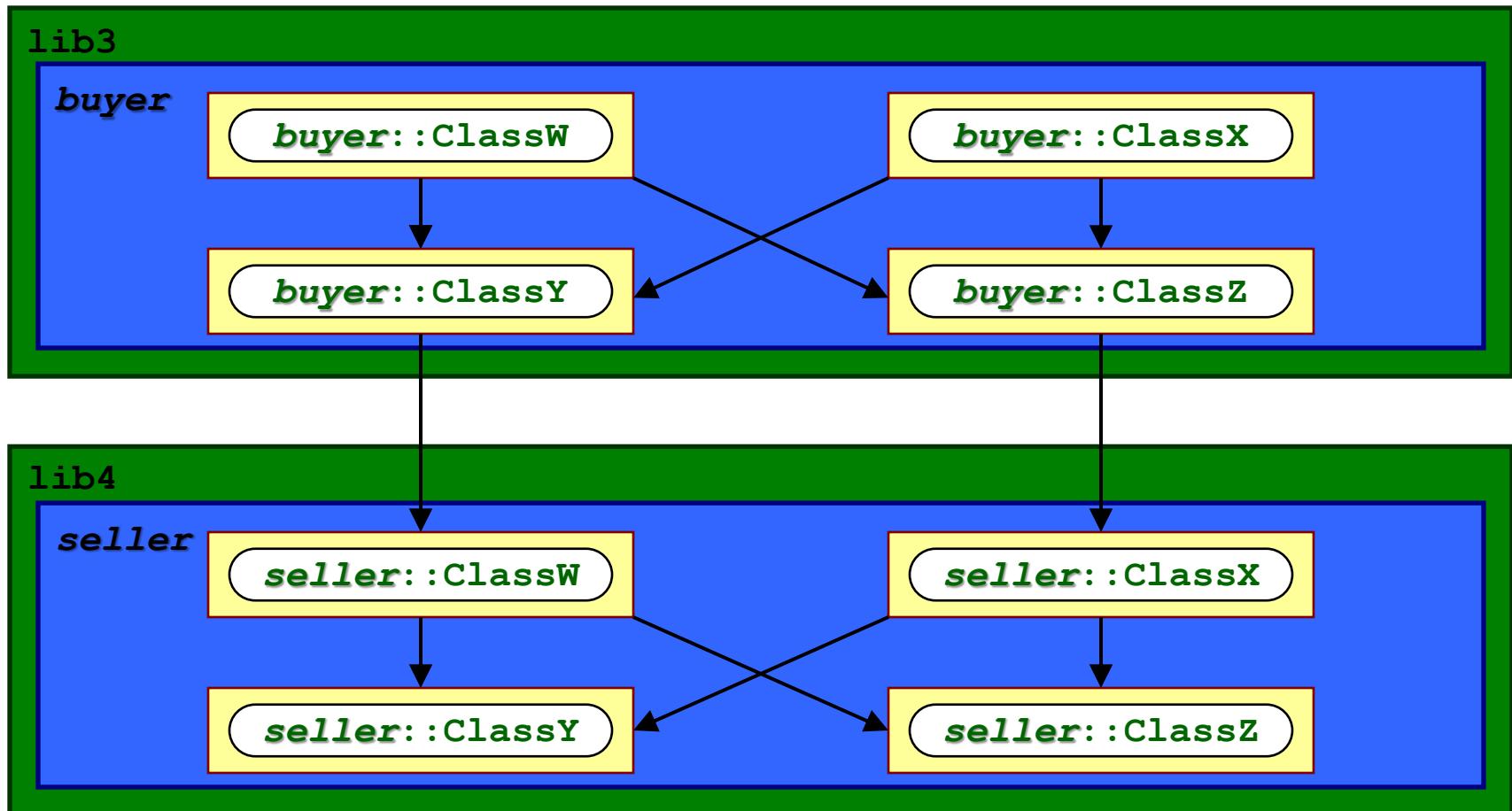
1. Process & Architecture

Logical/Physical Incoherence



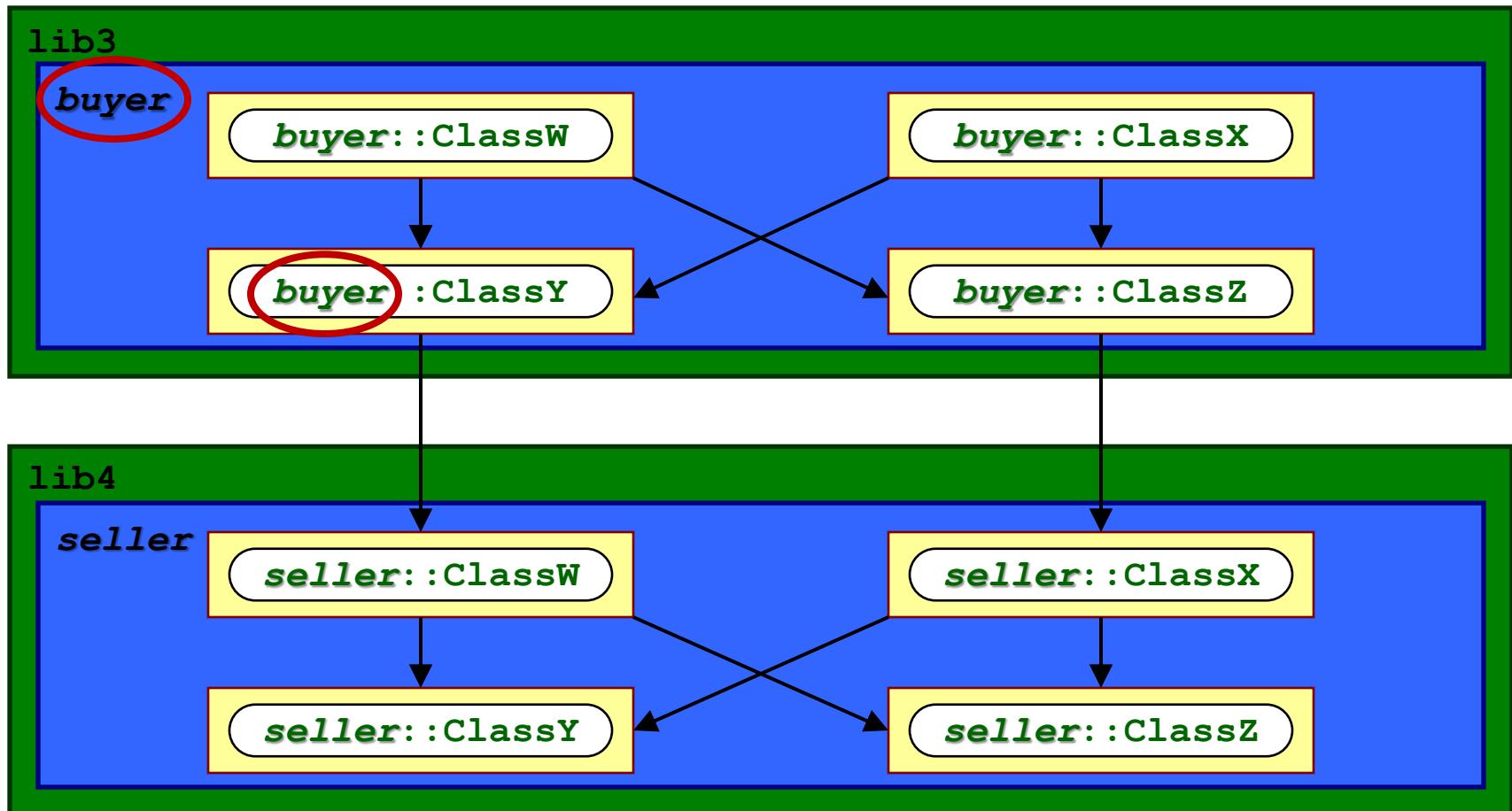
1. Process & Architecture

Logical/Physical Coherence



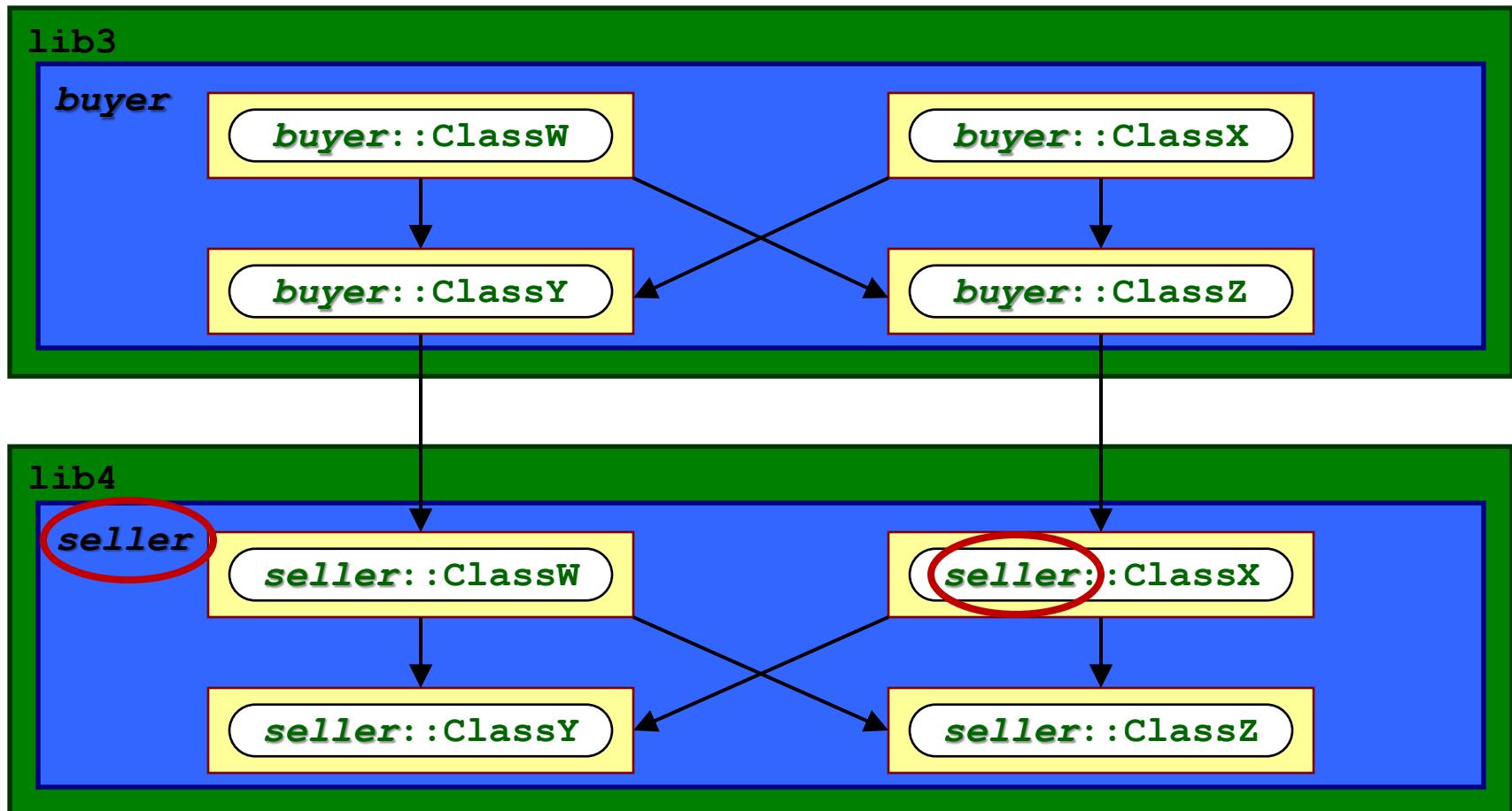
1. Process & Architecture

Logical/Physical Coherence



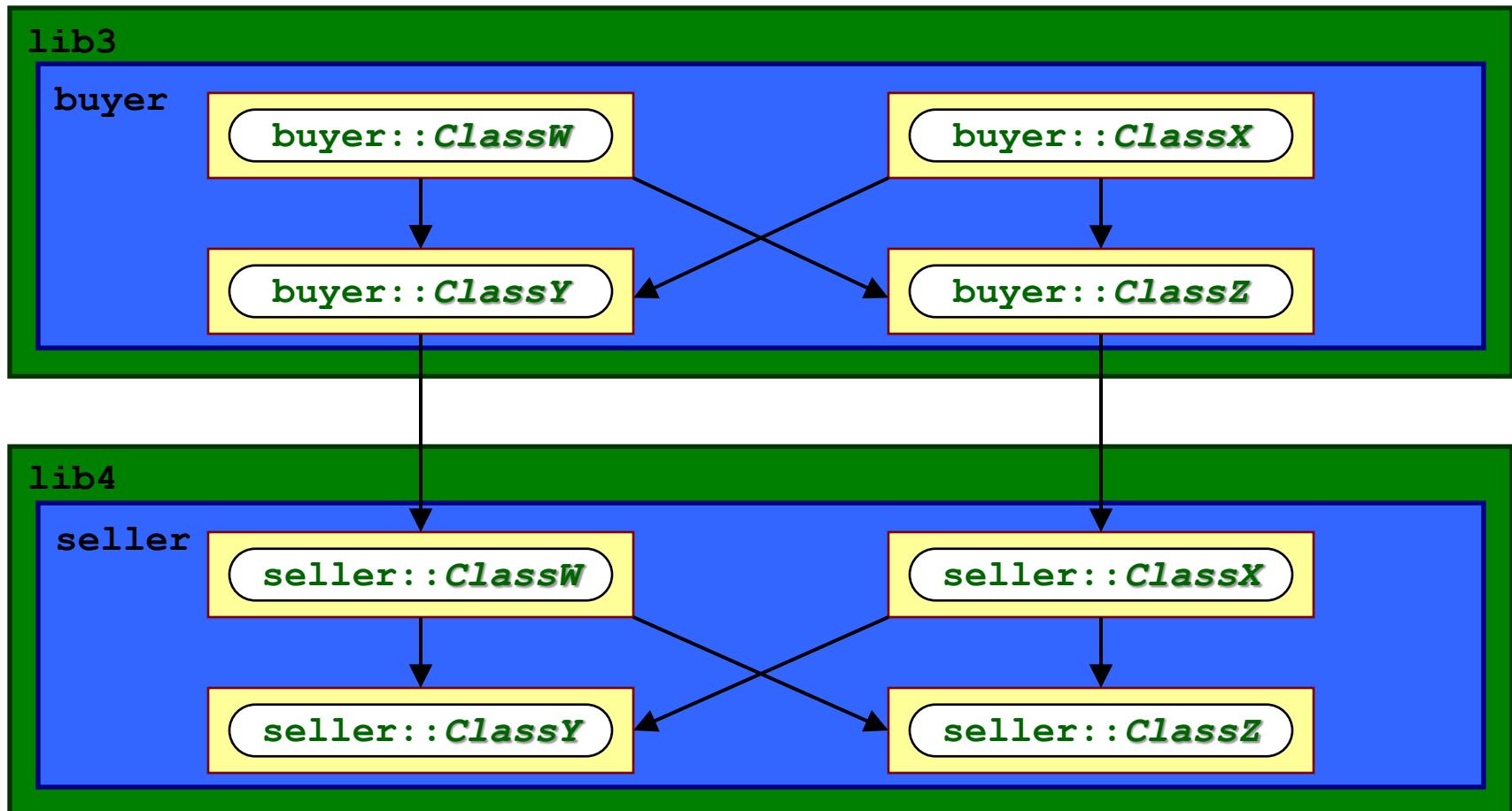
1. Process & Architecture

Logical/Physical Coherence



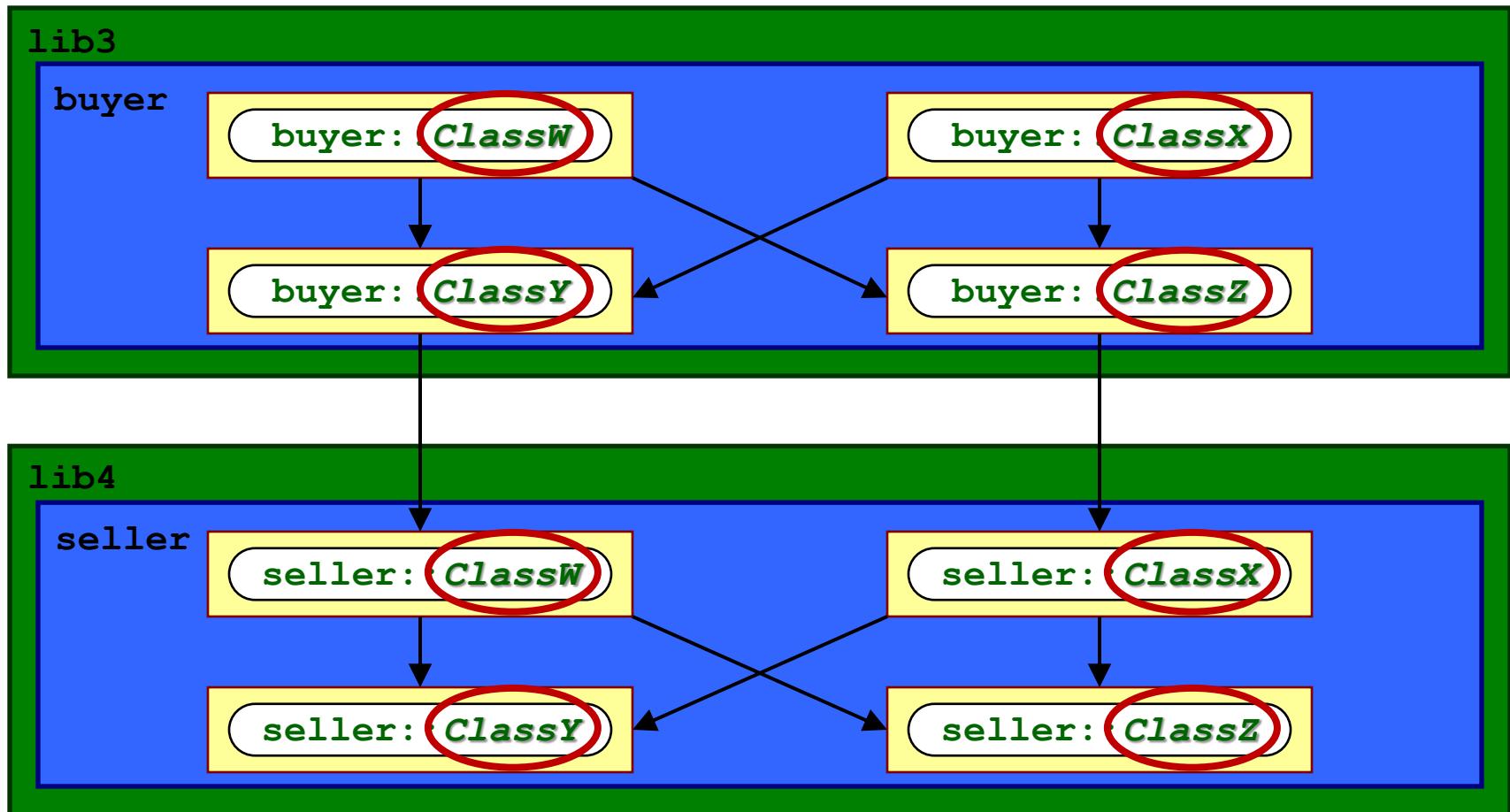
1. Process & Architecture

Logical/Physical Coherence



1. Process & Architecture

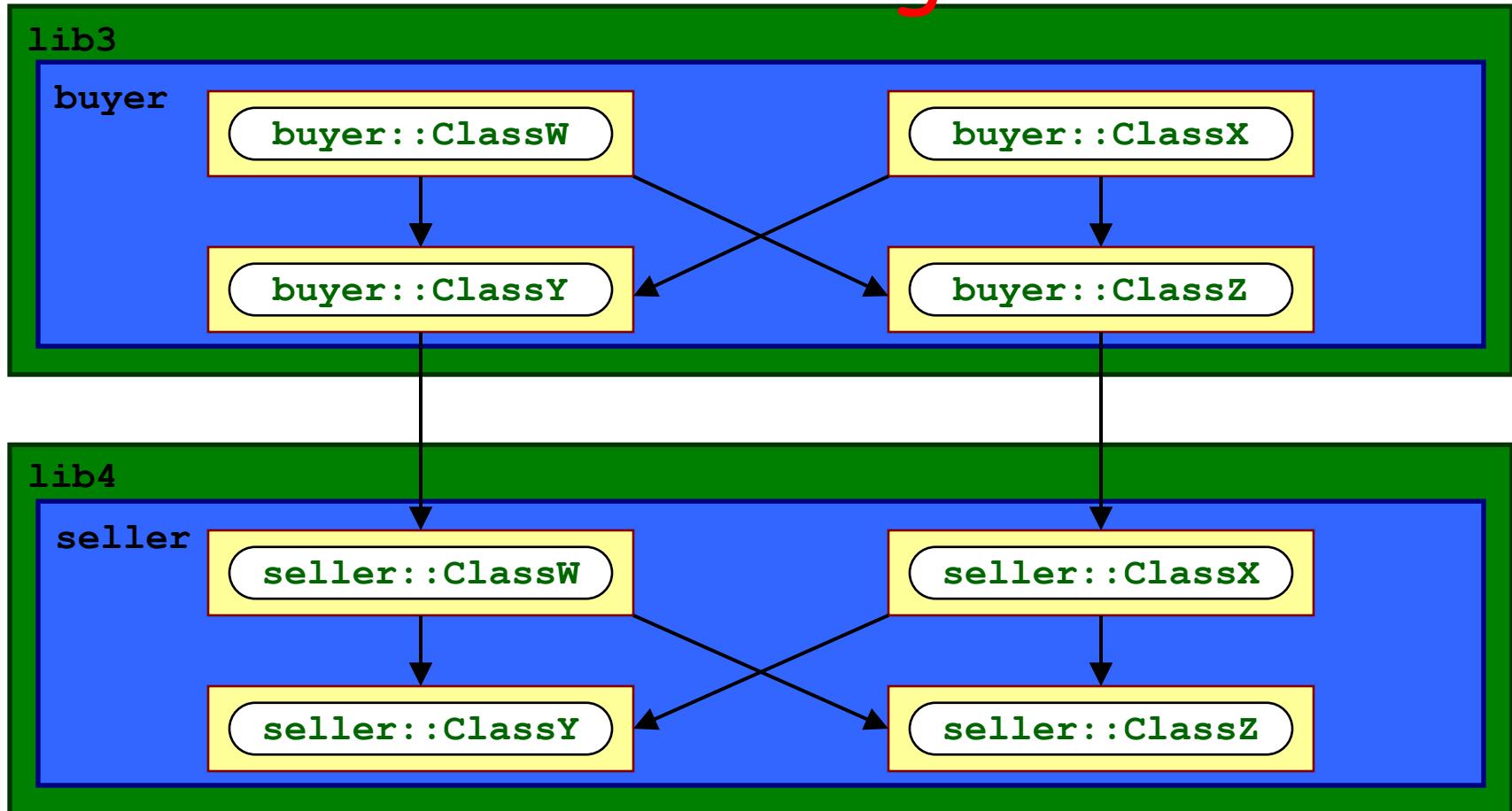
Logical/Physical Coherence



1. Process & Architecture

Logical/Physical Coherence

This is the goal!



1. Process & Architecture

Logical/Physical Synergy

There are two distinct aspects:

1. Logical/Physical Coherence

- ❖ Each logical subsystem is tightly encapsulated by a corresponding physical aggregate.

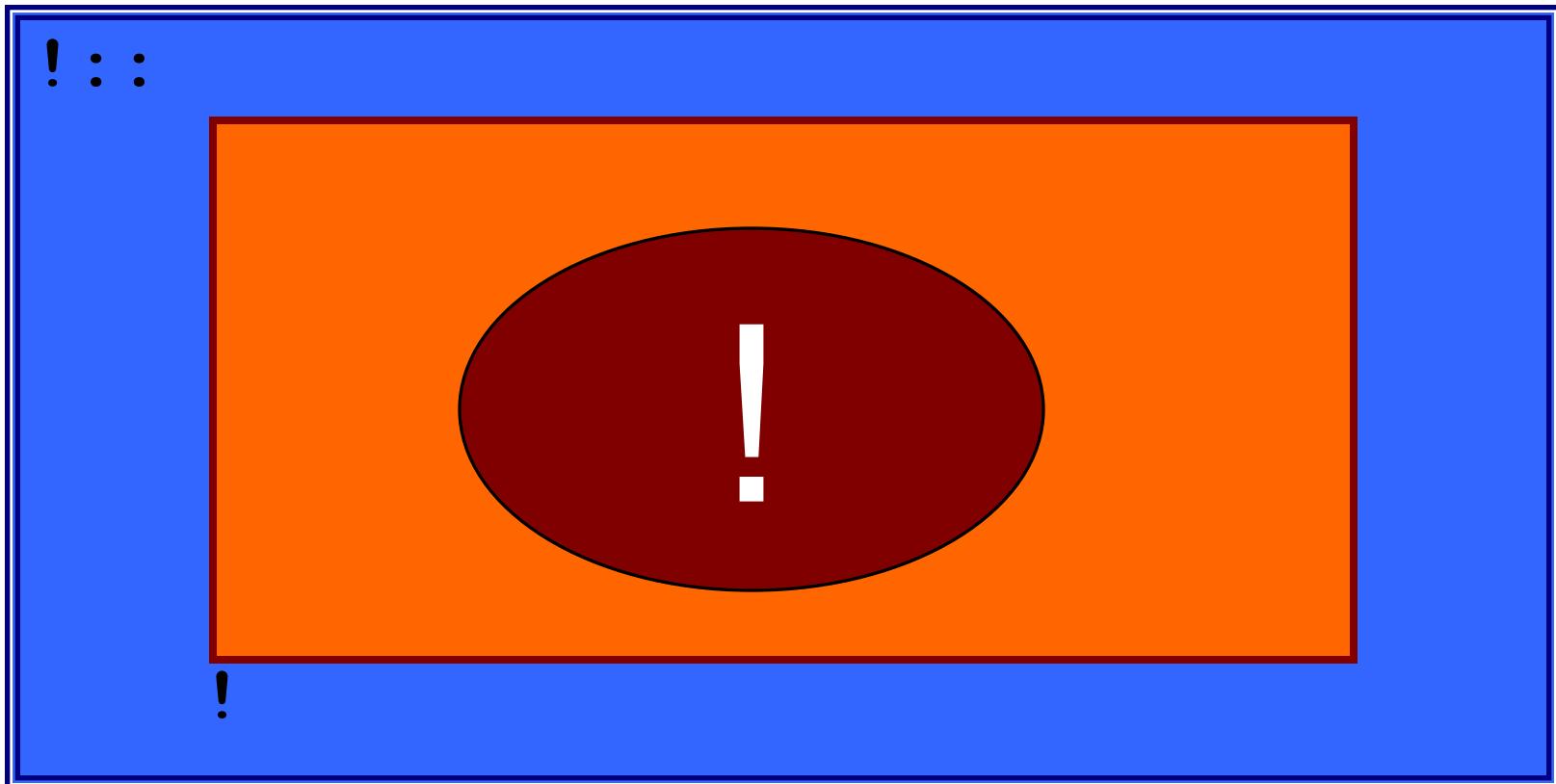
2. Logical/Physical Name Cohesion

- ❖ The precise physical location of the definition of a logical construct can be determined directly from its point of use (i.e., its **qualified** name).

1. Process & Architecture

Logical/Physical Name Cohesion

→ Key Concept ←

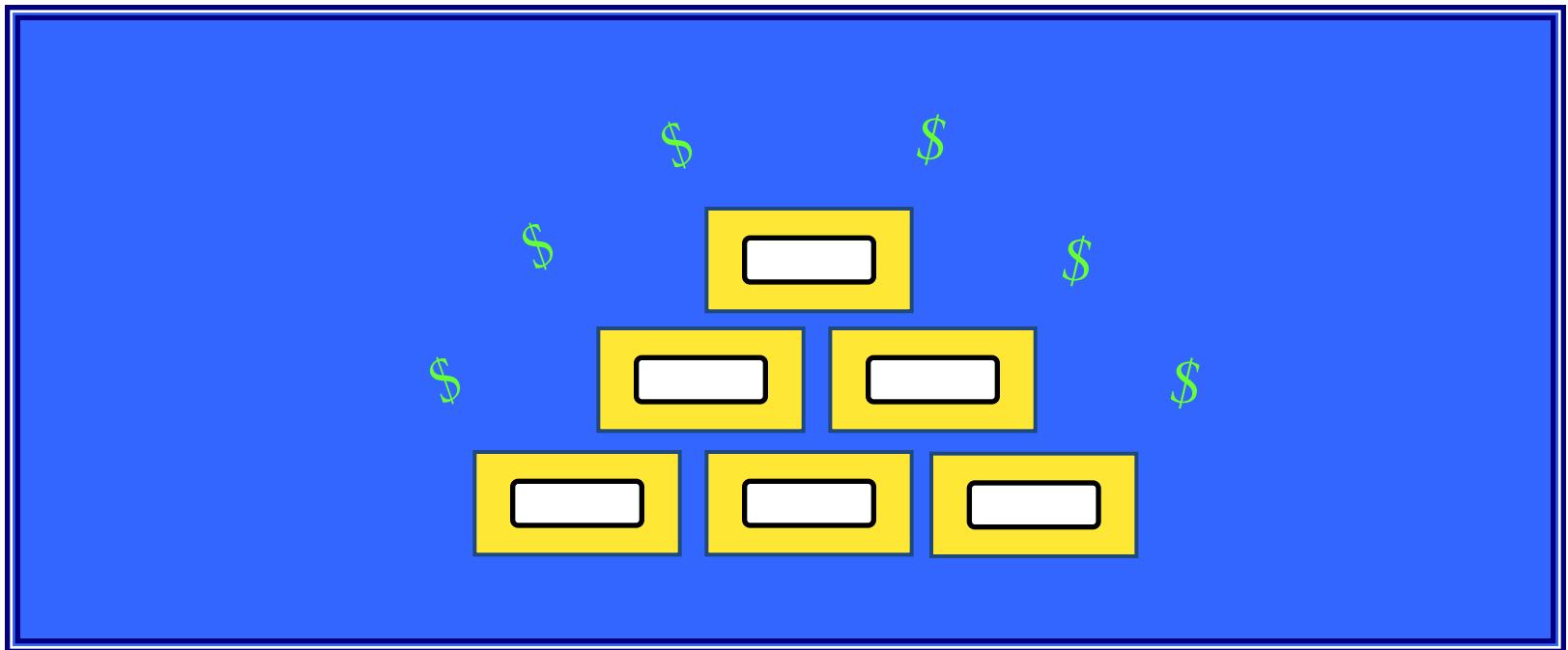


1. Process & Architecture

Packages

Classical Definition

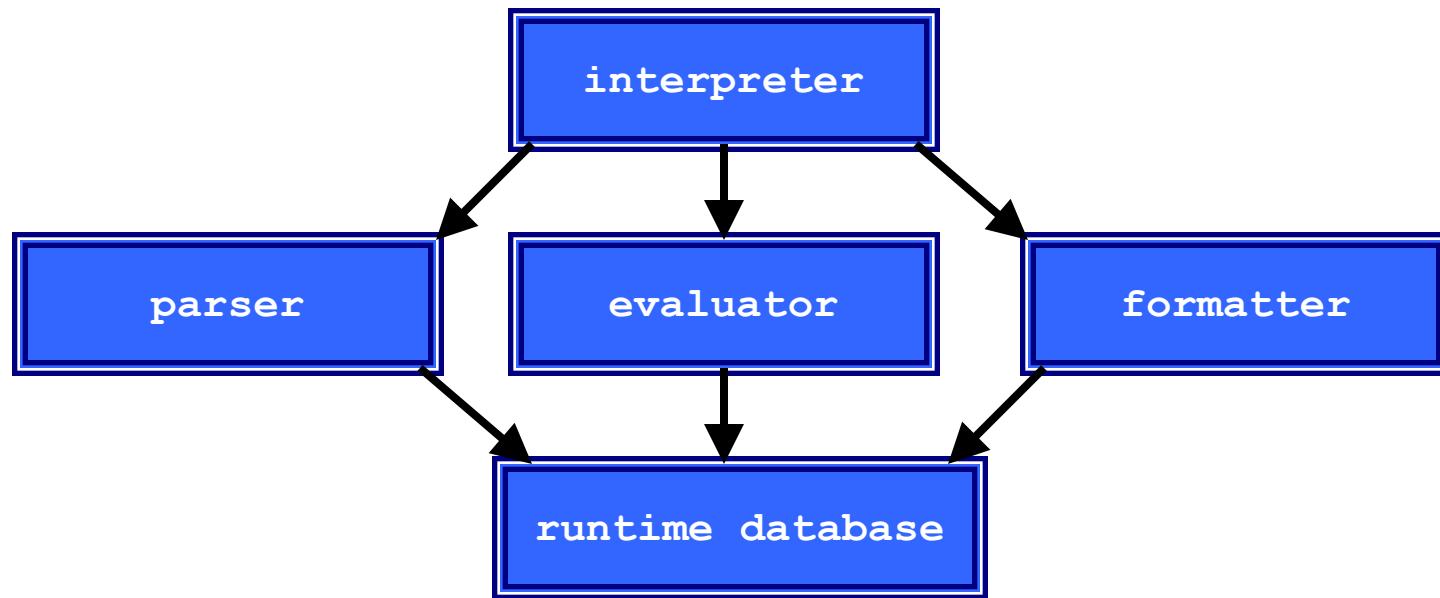
- A *package* is an acyclic collection of components organized as a logically and physically cohesive unit.



1. Process & Architecture

Packages

High-Level Interpreter Architecture

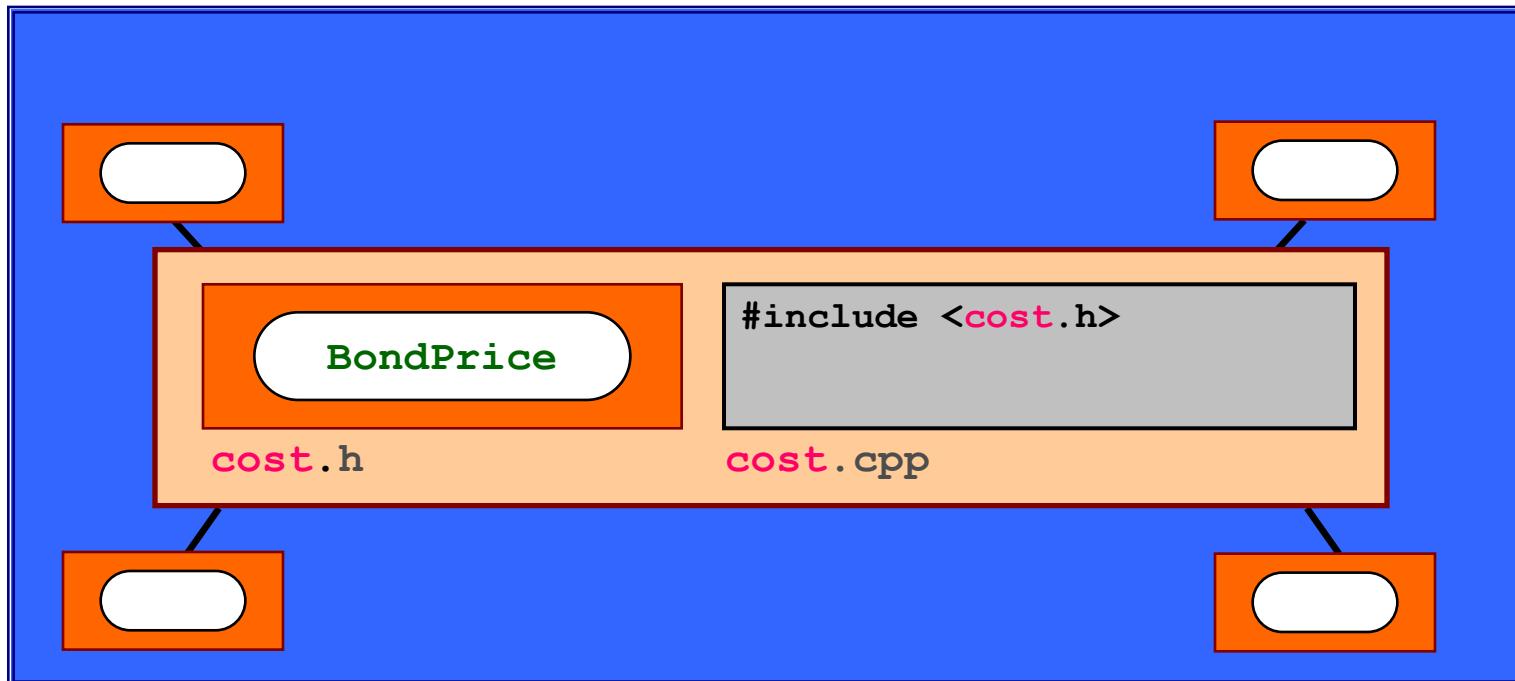


1. Process & Architecture

Architecturally Significant Names

Non-Cohesive Logical
and Physical Names

Package Name: **bts**
Component Name: **cost**
Class Name: **BondPrice**



BAD IDEA!

1. Process & Architecture

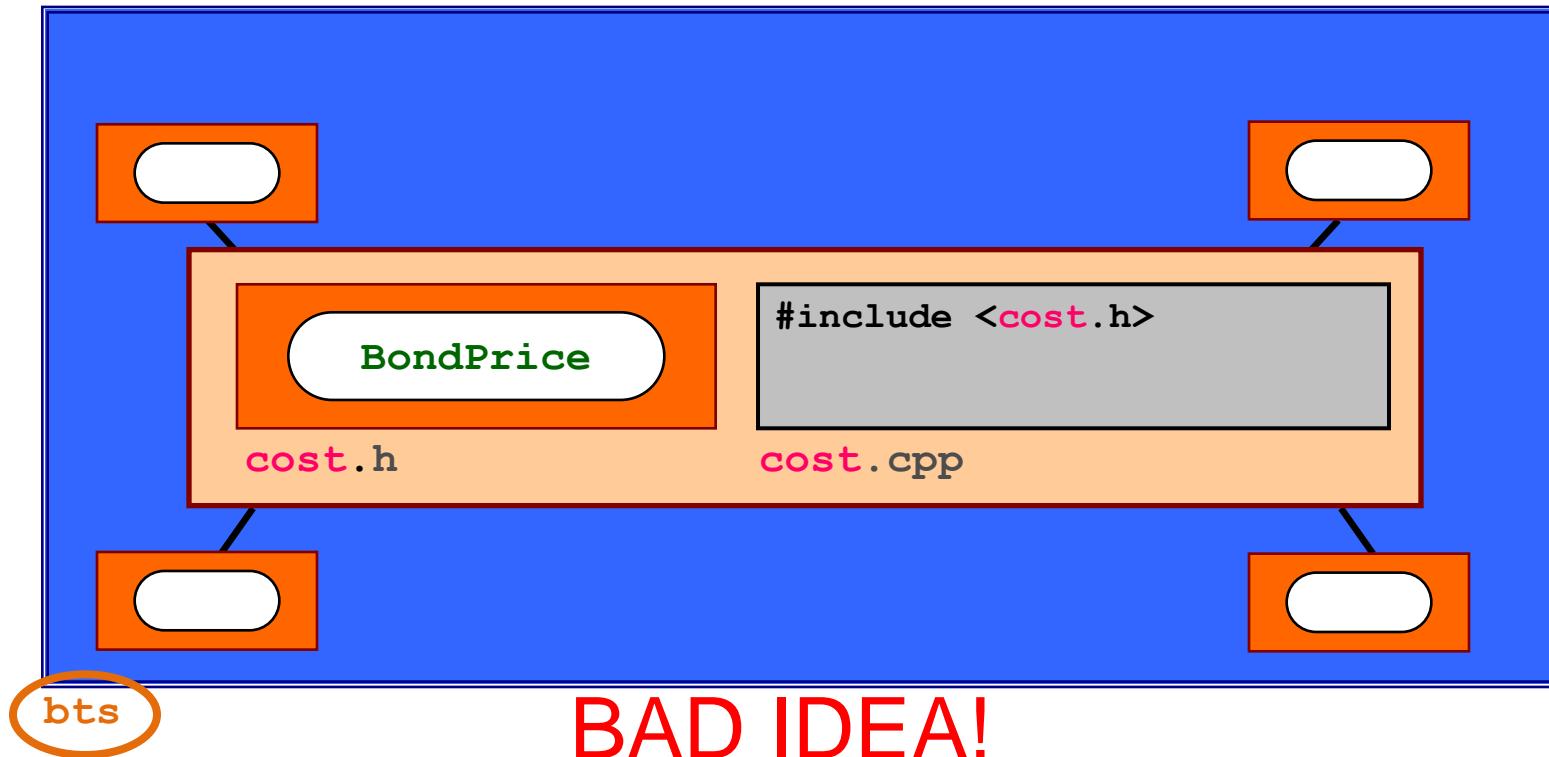
Architecturally Significant Names

Non-Cohesive Logical
and Physical Names

Package Name: **bts**

Component Name: **cost**

Class Name: **BondPrice**

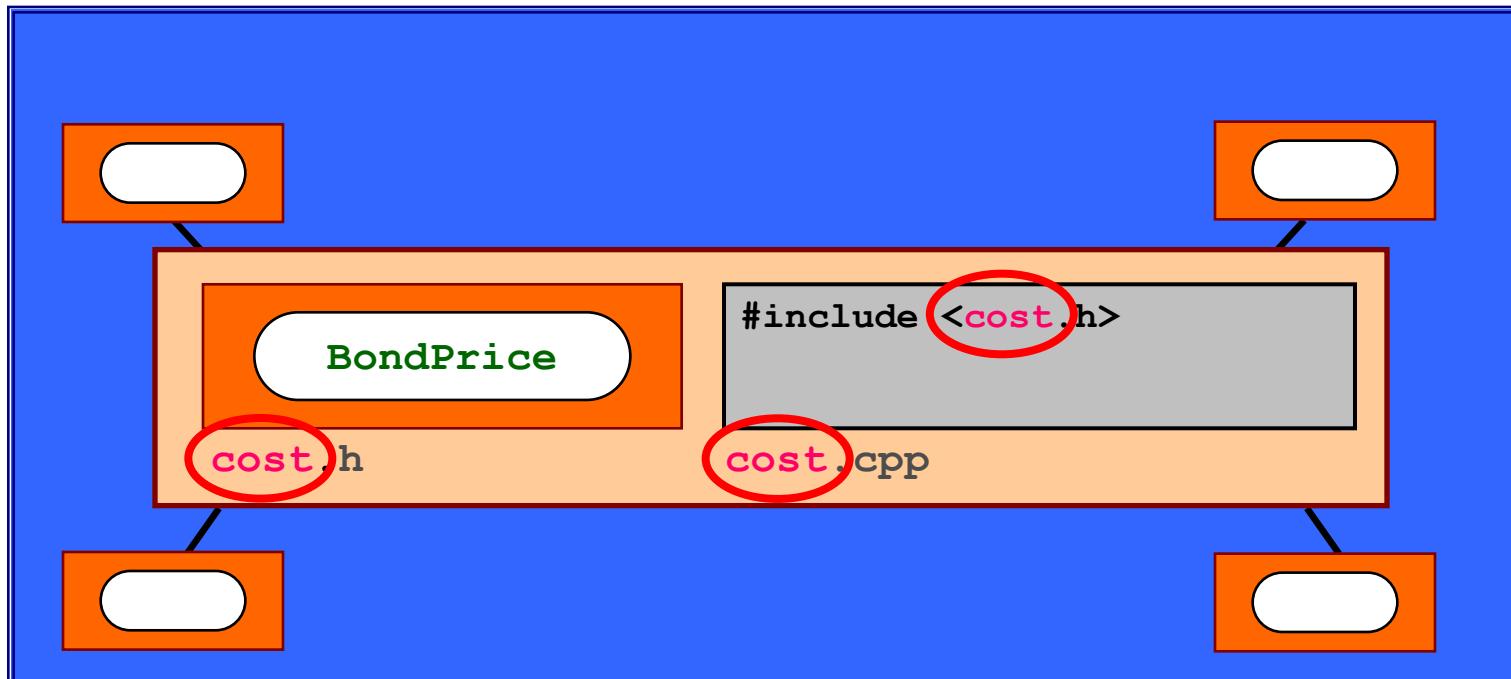


1. Process & Architecture

Architecturally Significant Names

Non-Cohesive Logical and Physical Names

Package Name: **bts**
Component Name: **cost**
Class Name: **BondPrice**



bts

BAD IDEA!

1. Process & Architecture

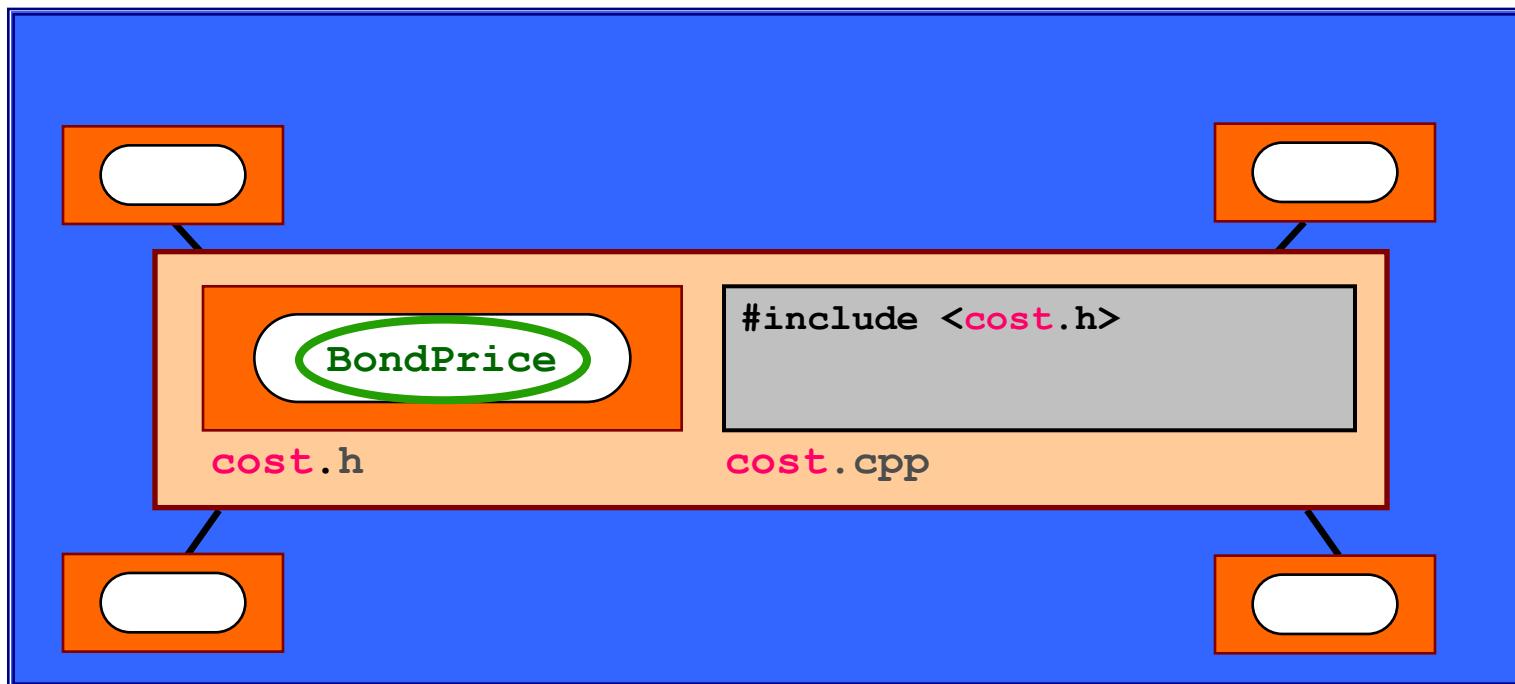
Architecturally Significant Names

Non-Cohesive Logical
and Physical Names

Package Name: `bts`

Component Name: `cost`

Class Name: `BondPrice`



BAD IDEA!

1. Process & Architecture

Architecturally Significant Names

Definition

An entity is ***Architecturally Significant*** if its name (or symbol) is intentionally **visible outside** the **UOR** that defines it.

1. Process & Architecture

Architecturally Significant Names

Definition

An entity is ***Architecturally Significant*** if its name (or symbol) is intentionally **visible outside** the **UOR** that defines it.

Design Rule

The **name** of each

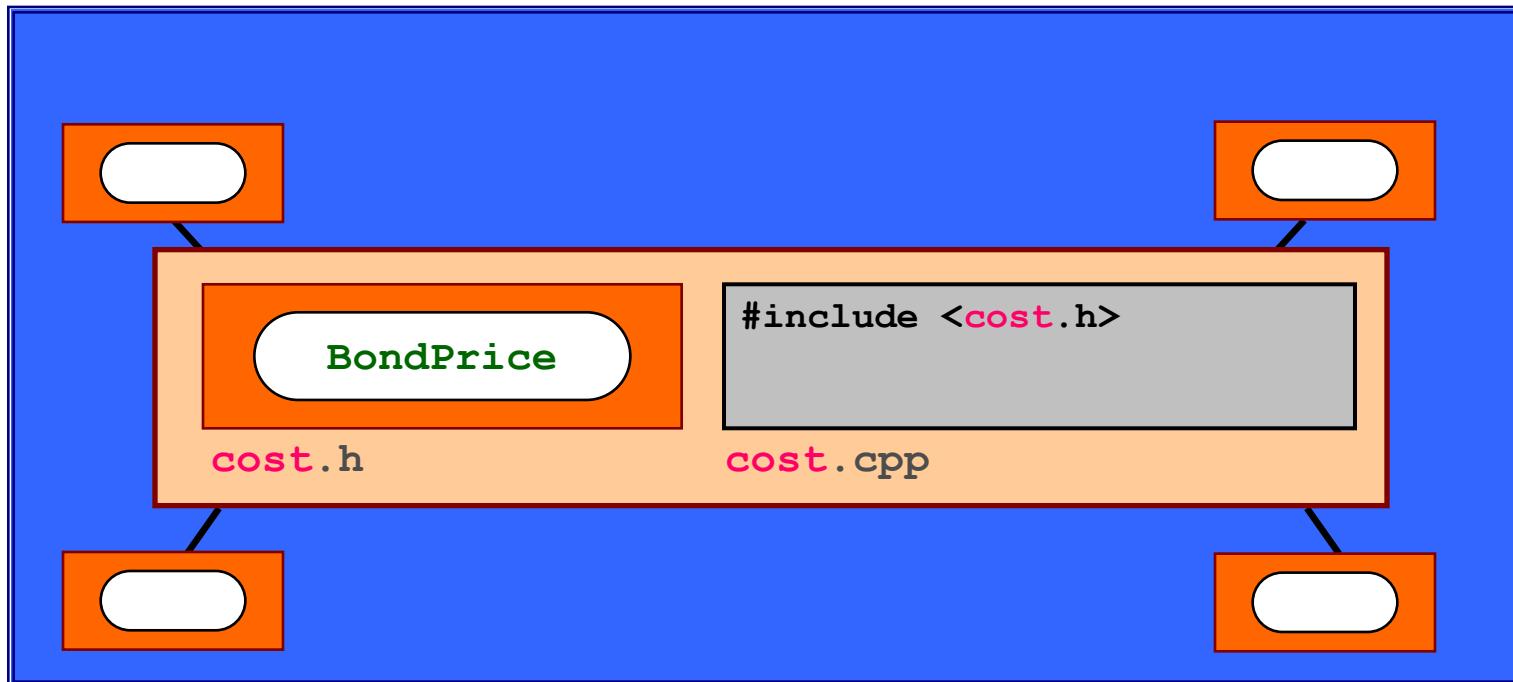
- Unit Of Release (**UOR**)
- (library) **component**

must be **unique** throughout the enterprise.

1. Process & Architecture

Physical Package Prefixes

Component Name **Not Matching** Package Name:
cost



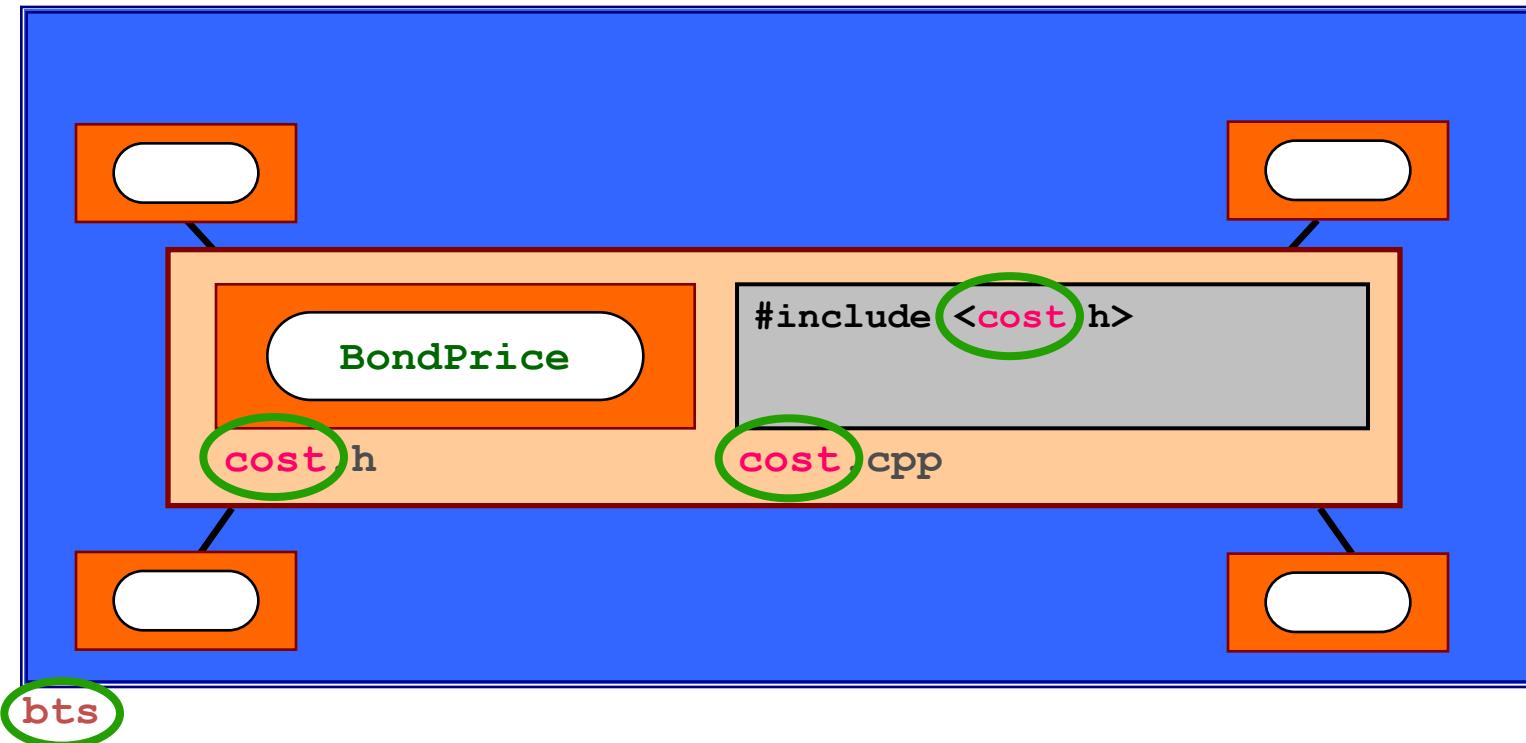
bts

1. Process & Architecture

Physical Package Prefixes

Component Name **Not Matching** Package Name:

cost



1. Process & Architecture

Physical Package Prefixes

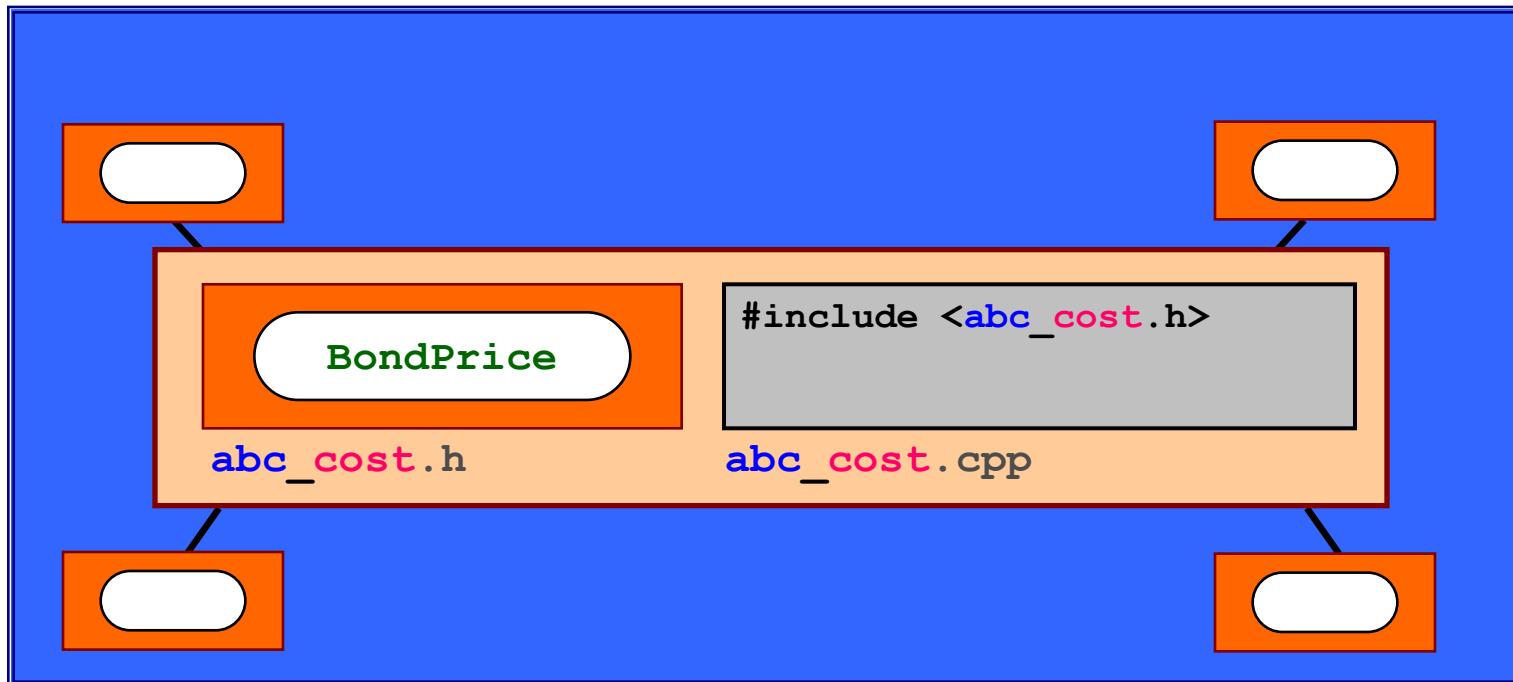
Design Rule

Each **component** name **begins** with the name of the **package** in which it resides, followed by an **underscore** ('_').

1. Process & Architecture

Physical Package Prefixes

Component Prefix **Doesn't Match** Package Name:
abc_cost



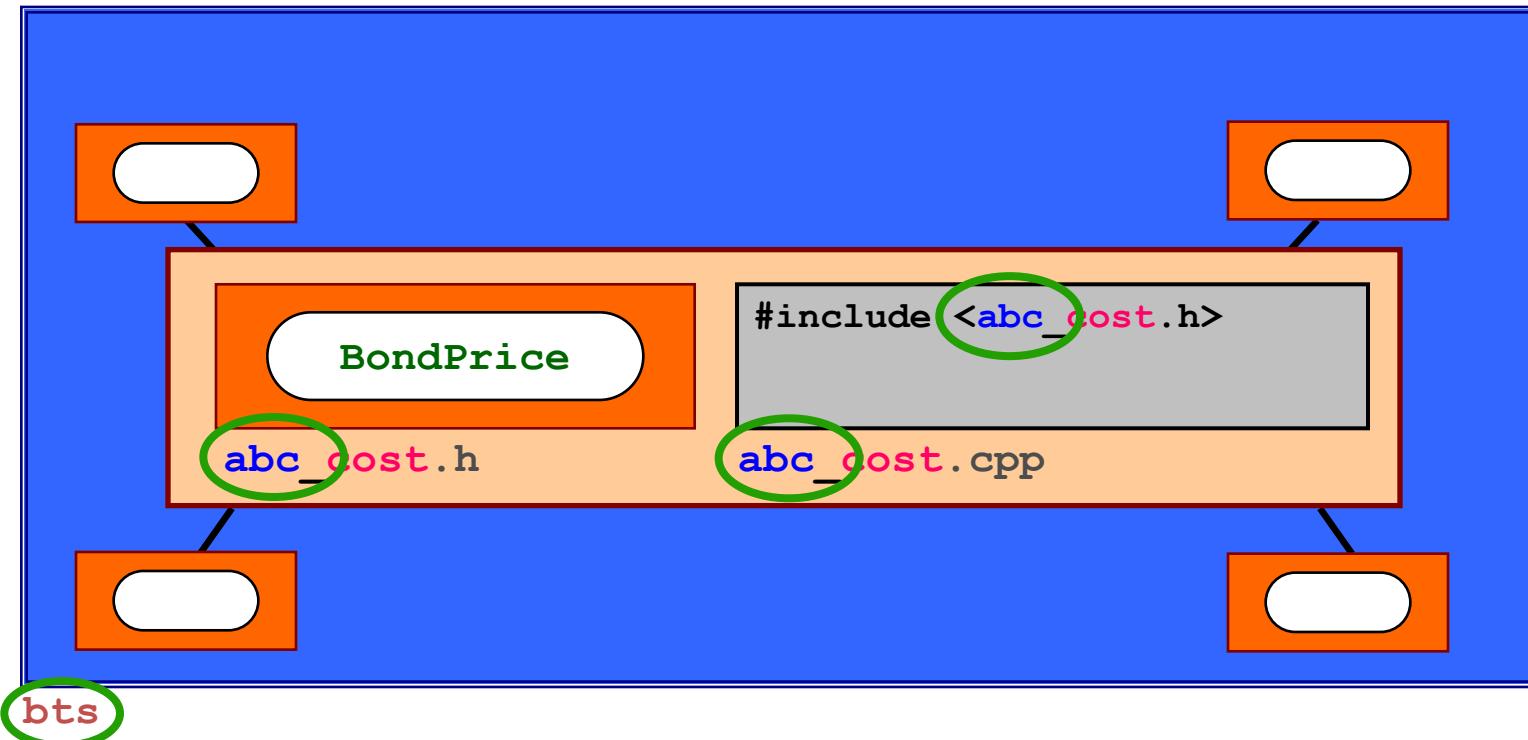
bts

1. Process & Architecture

Physical Package Prefixes

Component Prefix **Doesn't Match** Package Name:

abc cost

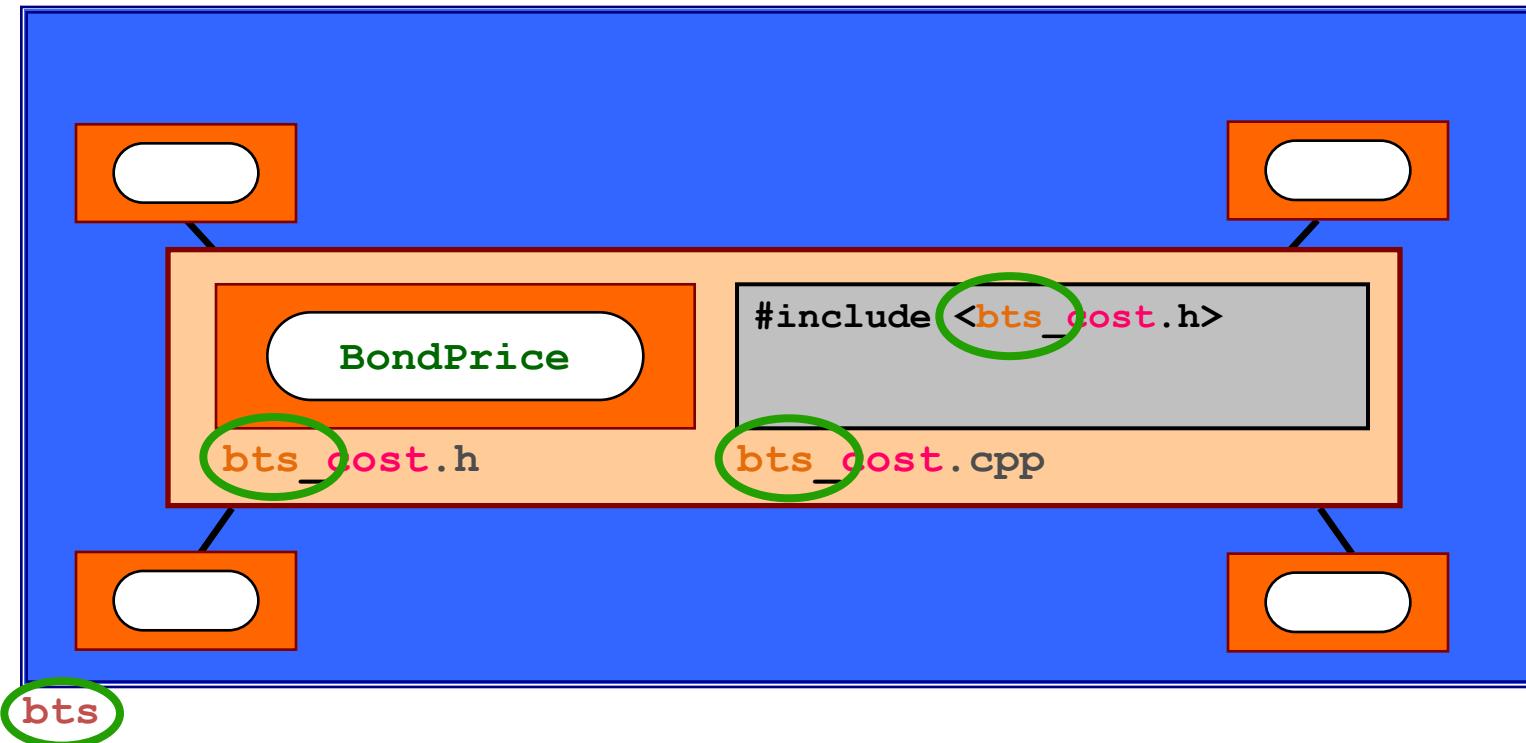


1. Process & Architecture

Physical Package Prefixes

Component Prefix **Matches** Package Name:

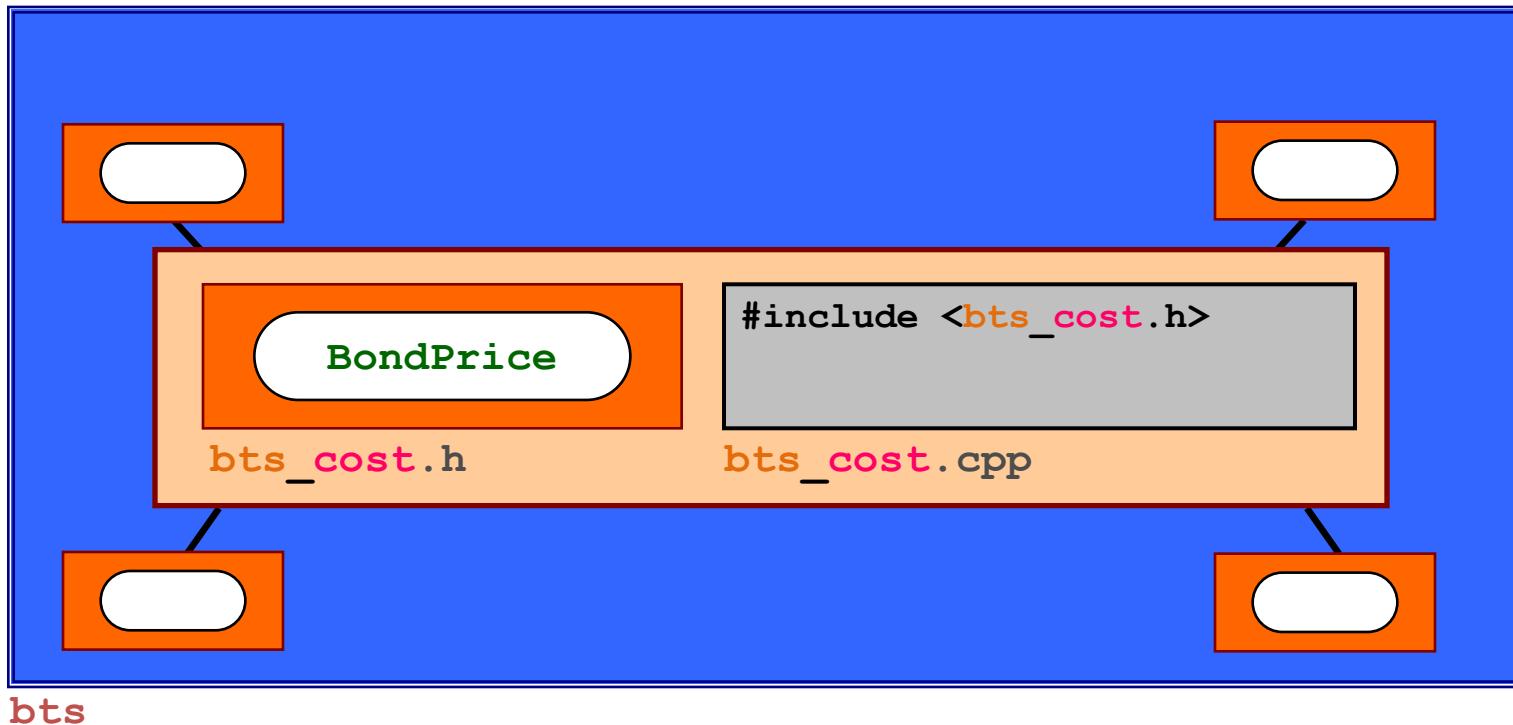
bts_cost



1. Process & Architecture

Physical Package Prefixes

Component Prefix **Matches** Package Name:
bts_cost

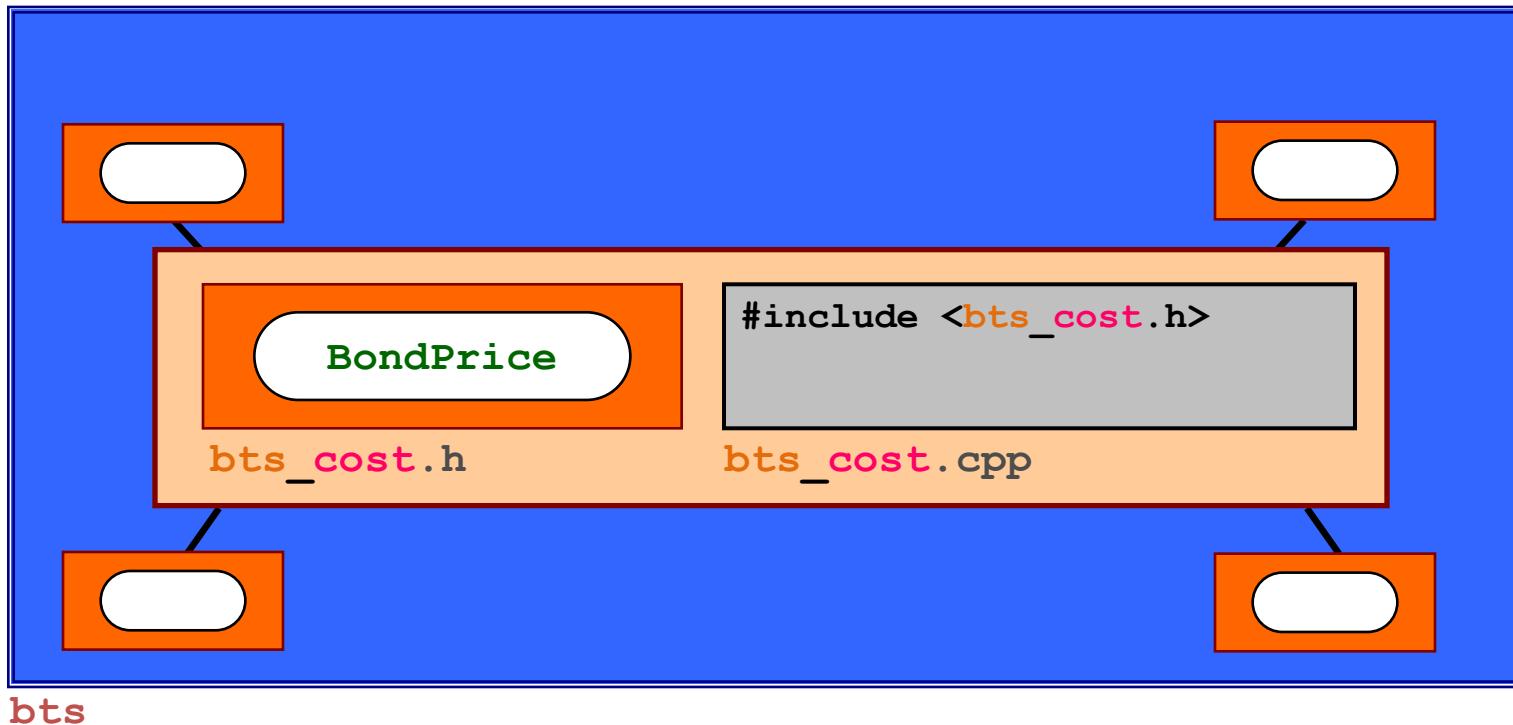


bts

1. Process & Architecture

Logical Package Namespaces

Package Namespace **Should Match** Package Name
bts

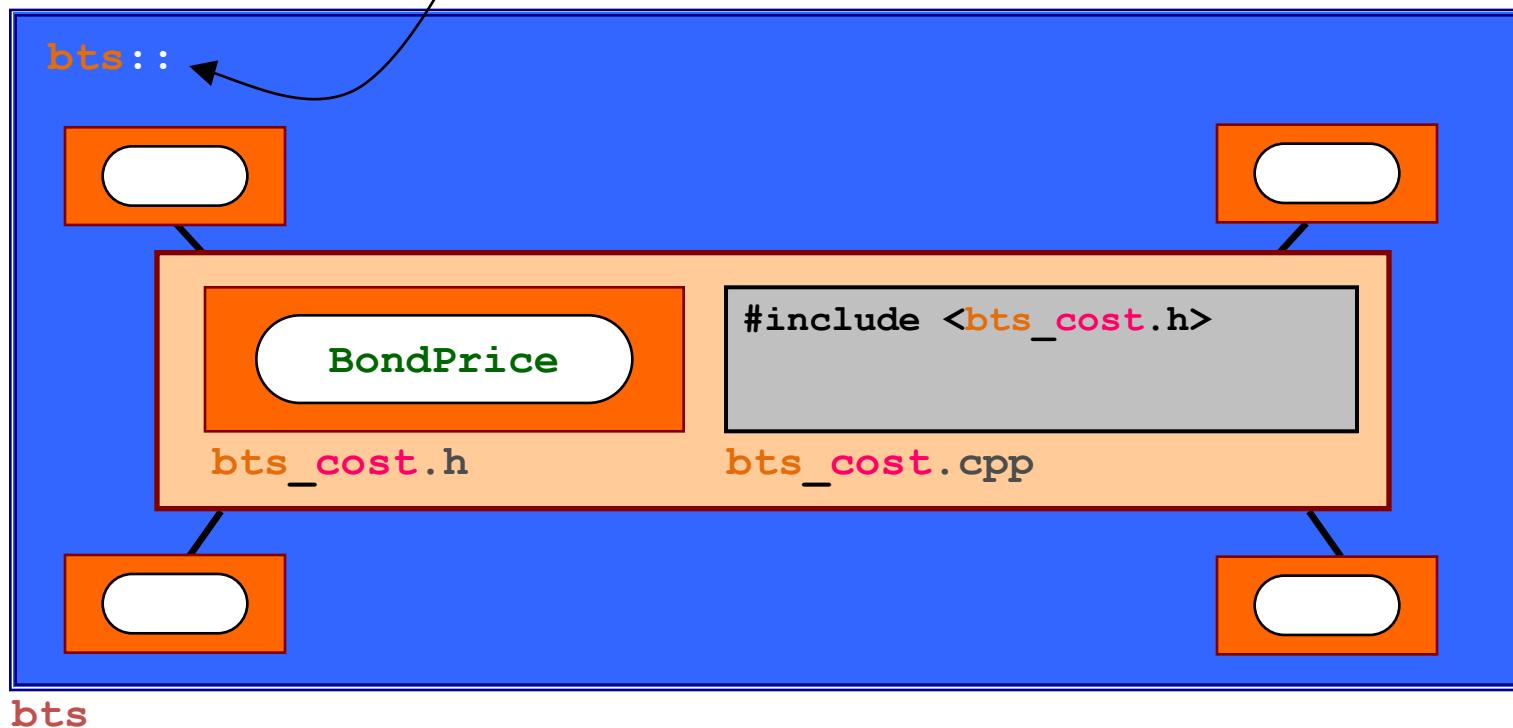


bts

1. Process & Architecture

Logical Package Namespaces

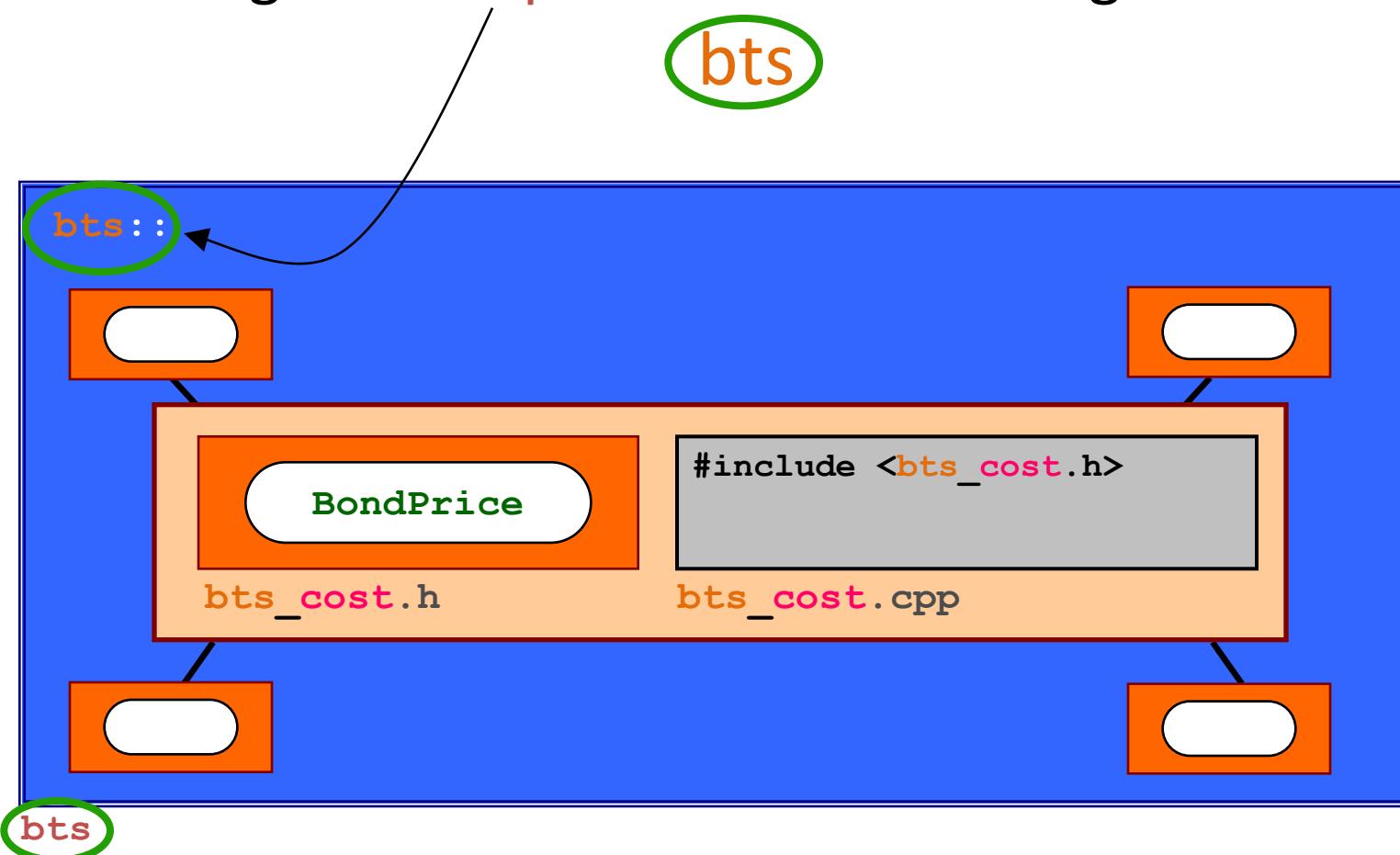
Package Namespace **Matches** Package Name
bts



1. Process & Architecture

Logical Package Namespaces

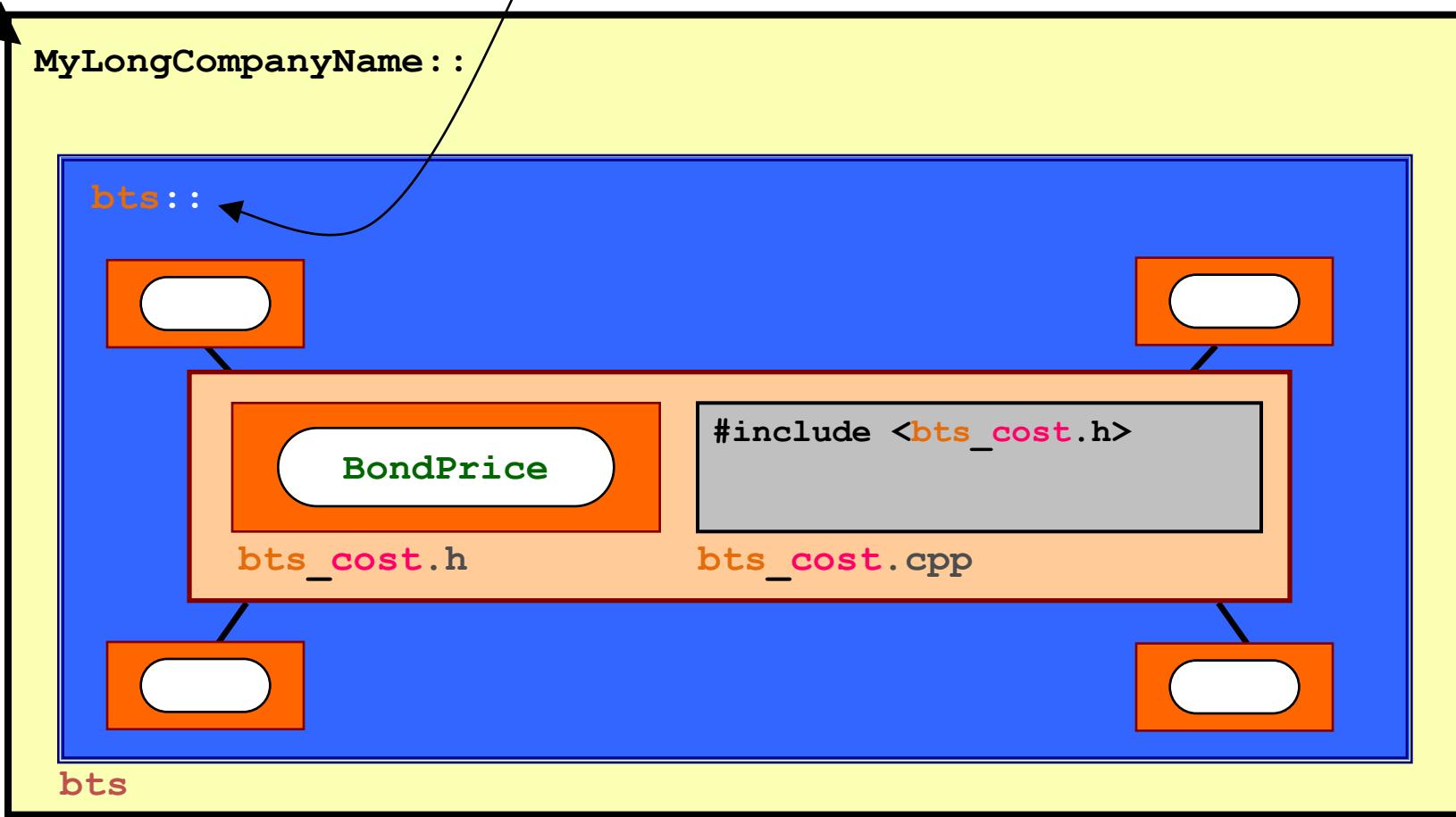
Package Namespace **Matches** Package Name



1. Process & Architecture

(Logical) Enterprise-Wide Namespace

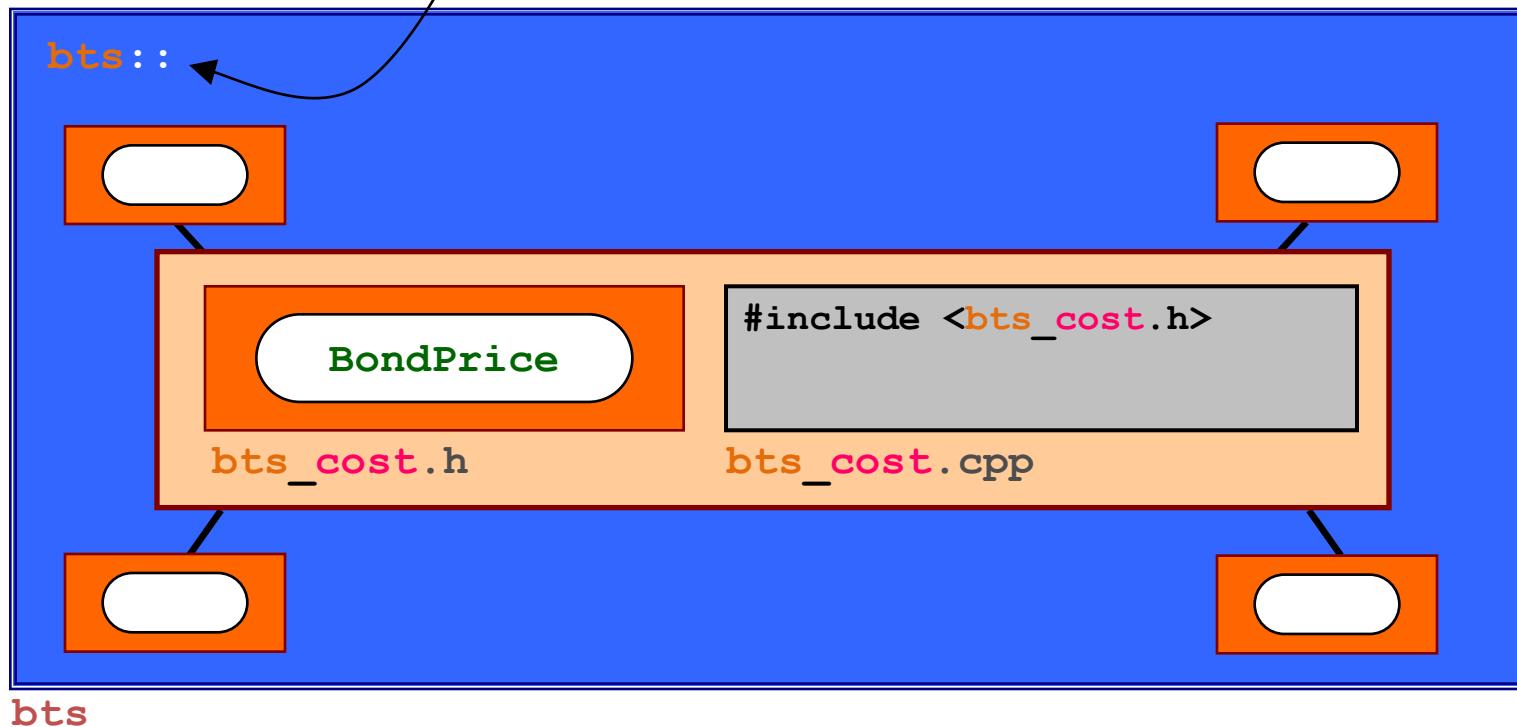
Package Namespace **Matches** Package Name



1. Process & Architecture

Logical Package Namespaces

Package Namespace **Matches** Package Name
bts



1. Process & Architecture

Logical/Physical Name Cohesion

Design Goal

The use of each logical entity should alone be sufficient to know the component in which it is defined.

1. Process & Architecture

Logical/Physical Name Cohesion

Design Goal

The use of each logical entity should alone be sufficient to know the component in which it is defined.

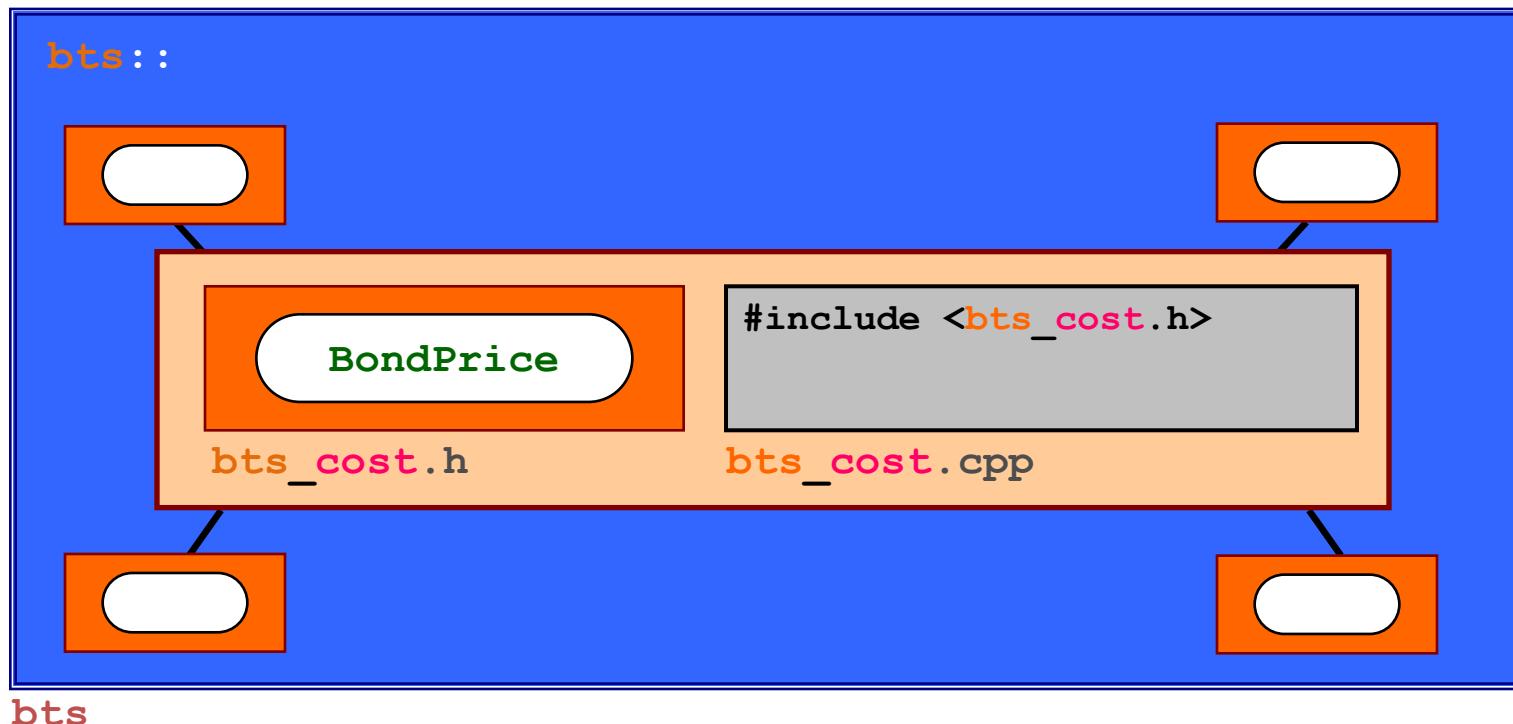
Design Rule

The (lowercased) name of every logical construct (other than free operators) declared at package-namespace scope must have, as a prefix, the name of the component that implements it.

1. Process & Architecture

Logical/Physical Name Cohesion

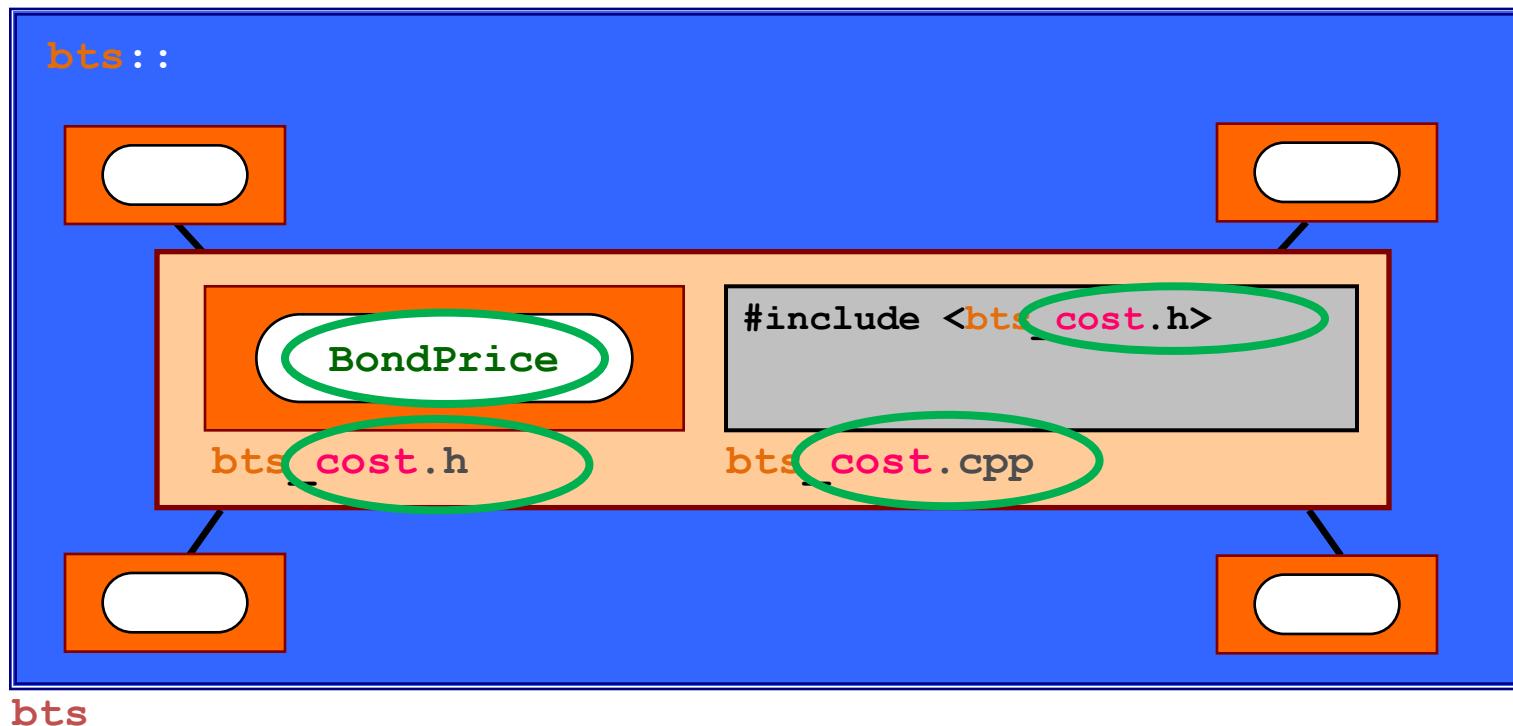
Class name **should** match Component name
BondPrice \longleftrightarrow **cost**



1. Process & Architecture

Logical/Physical Name Cohesion

Class name **should** match Component name
BondPrice \longleftrightarrow **cost**

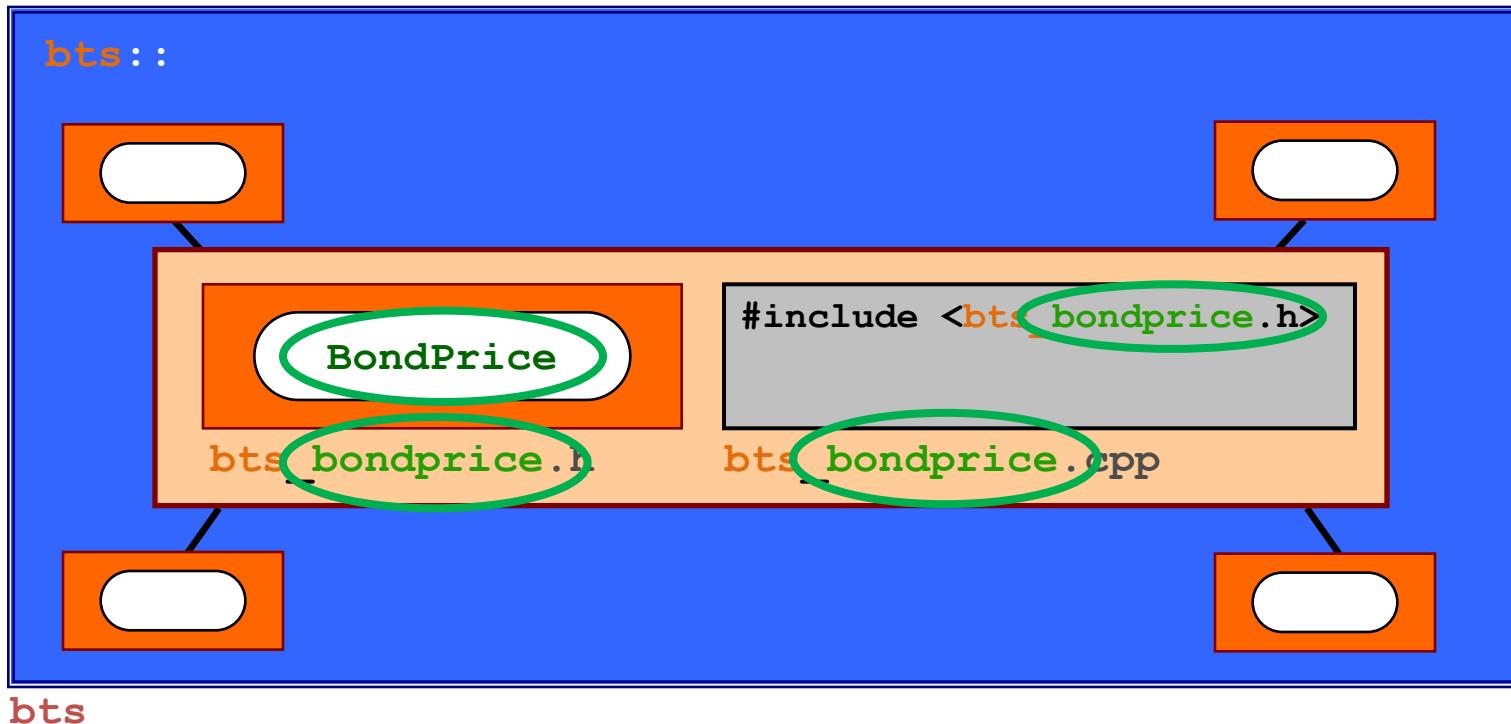


1. Process & Architecture

Logical/Physical Name Cohesion

Class name **does** match Component name

BondPrice \longleftrightarrow bondprice

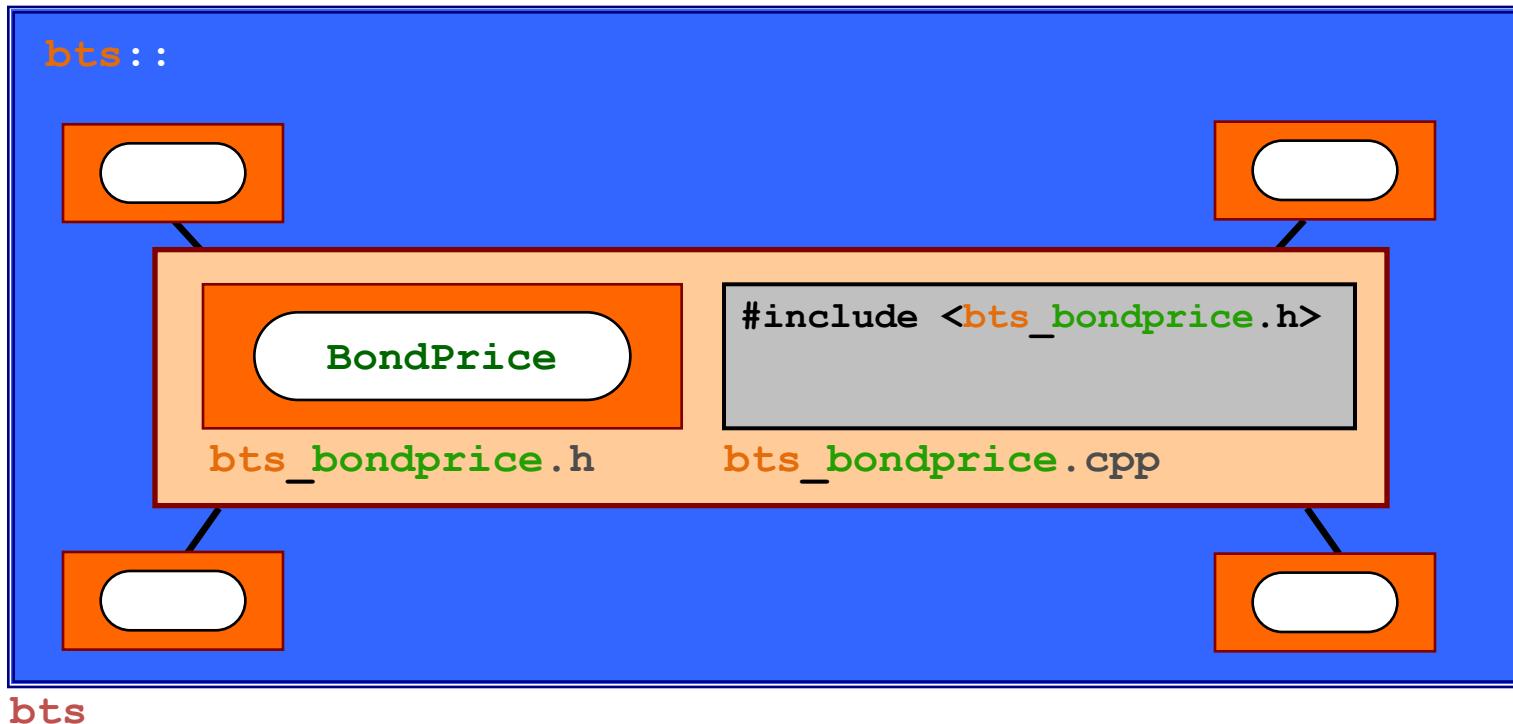


1. Process & Architecture

Logical/Physical Name Cohesion

Class name **does** match Component name

BondPrice \longleftrightarrow **bondprice**



1. Process & Architecture

Logical/Physical Name Cohesion

Some more details:

- ❑ Namespaces used for enterprise and package.
- ❑ Only classes* at package namespace scope.
- ❑ No *free* functions: C-style functions are implemented as static members of a struct.
- ❑ Operators are defined only in components that also define at least one of their parameter types.
- ❑ Ultra short package names mean: **No** using!

*Also structs, class templates, operators, and certain aspect functions (e.g., swap).

1. Process & Architecture

Logical/Physical Name Cohesion

Some more details:

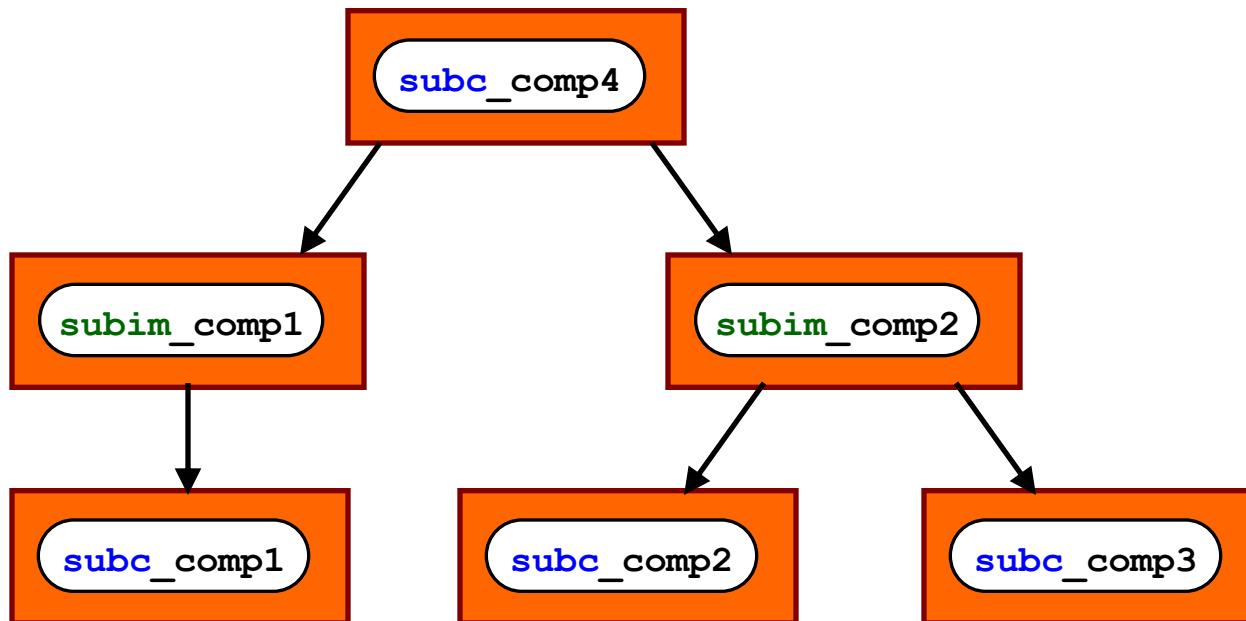
- ❑ Namespaces used for enterprise and package.
- ❑ Only classes* at package namespace scope.
- ❑ No *free* functions: C-style functions are implemented as static members of a struct.
- ❑ Operators are defined only in components that also define at least one of their parameter types.
- ❑ Ultra short package names mean: **No using!**

*Also structs, class templates, operators, and certain aspect functions (e.g., swap).

1. Process & Architecture

Logical/Physical Name Cohesion

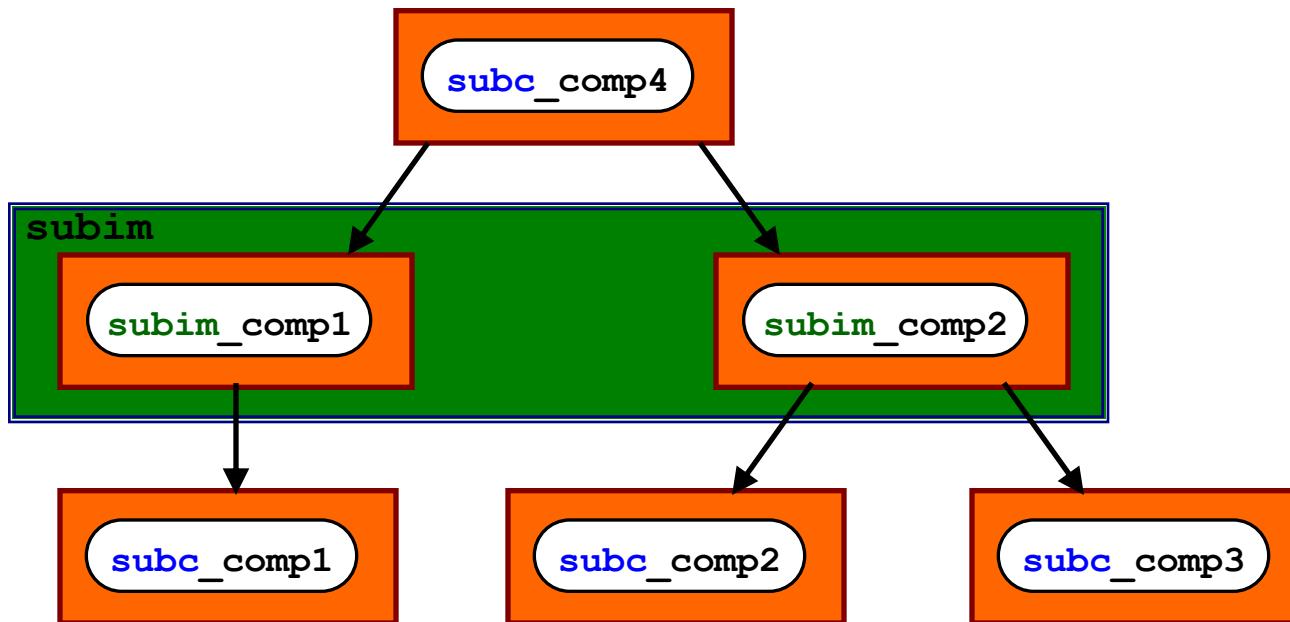
Package naming is more than just a convention:



1. Process & Architecture

Logical/Physical Name Cohesion

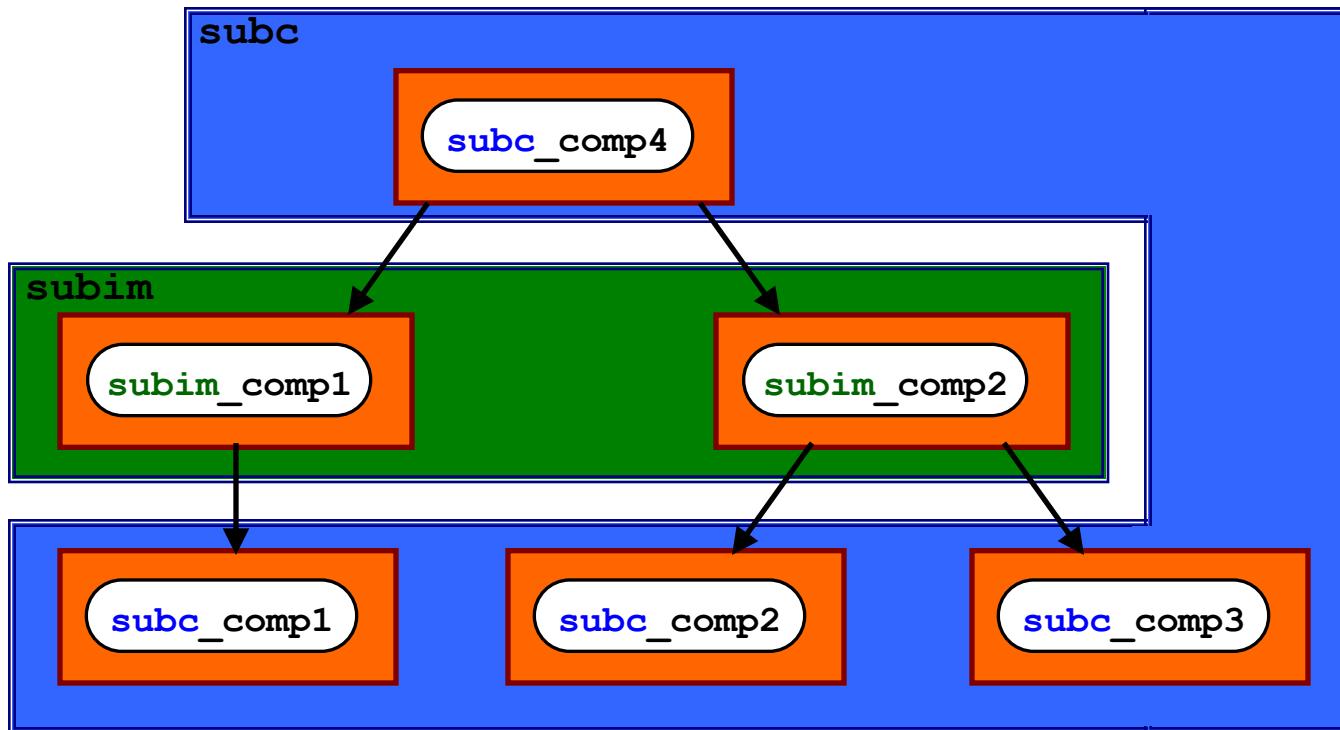
Package naming is more than just a convention:



1. Process & Architecture

Logical/Physical Name Cohesion

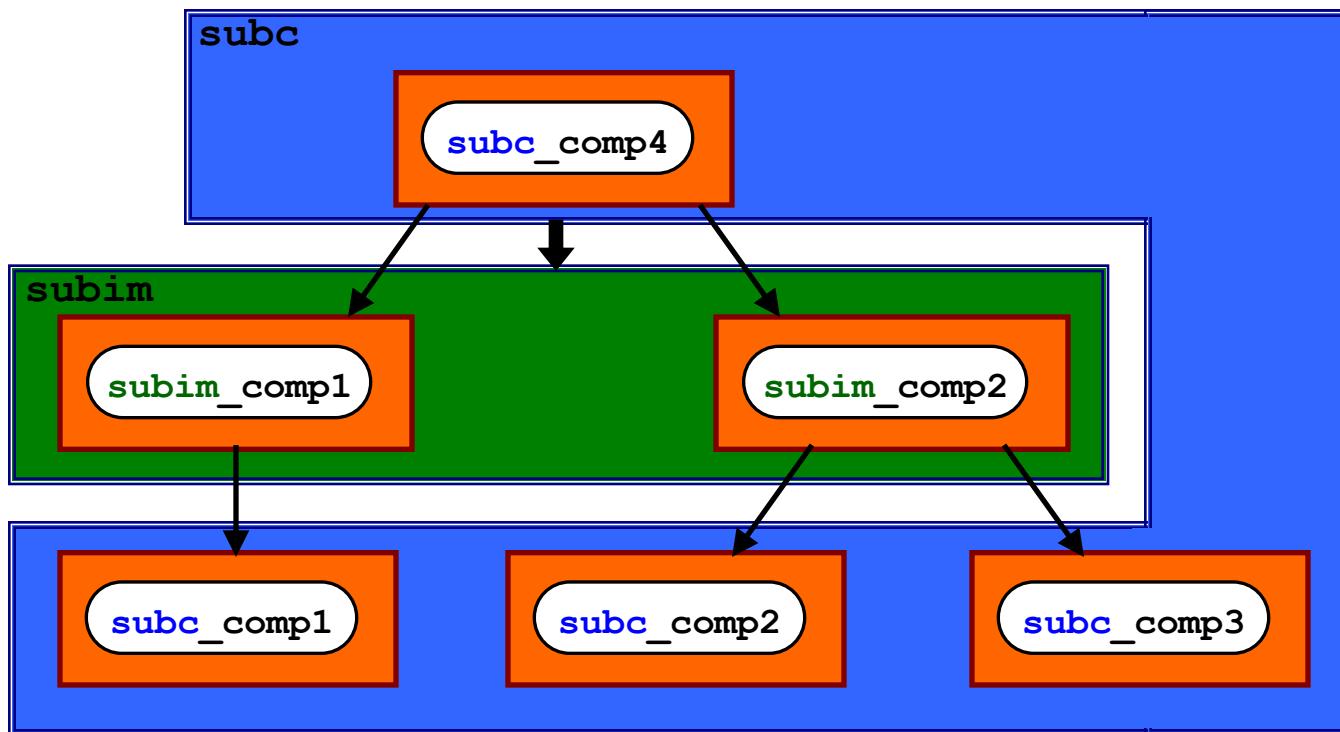
Package naming is more than just a convention:



1. Process & Architecture

Logical/Physical Name Cohesion

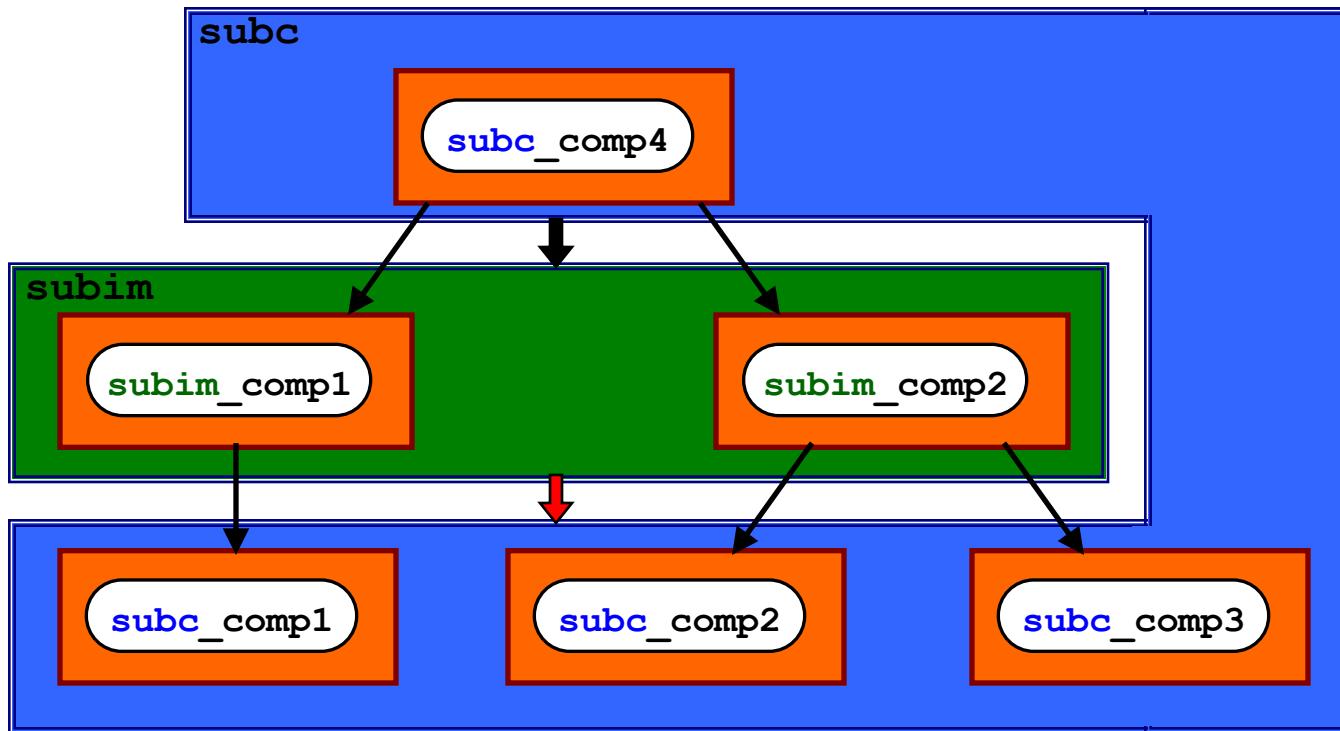
Package naming is more than just a convention:



1. Process & Architecture

Logical/Physical Name Cohesion

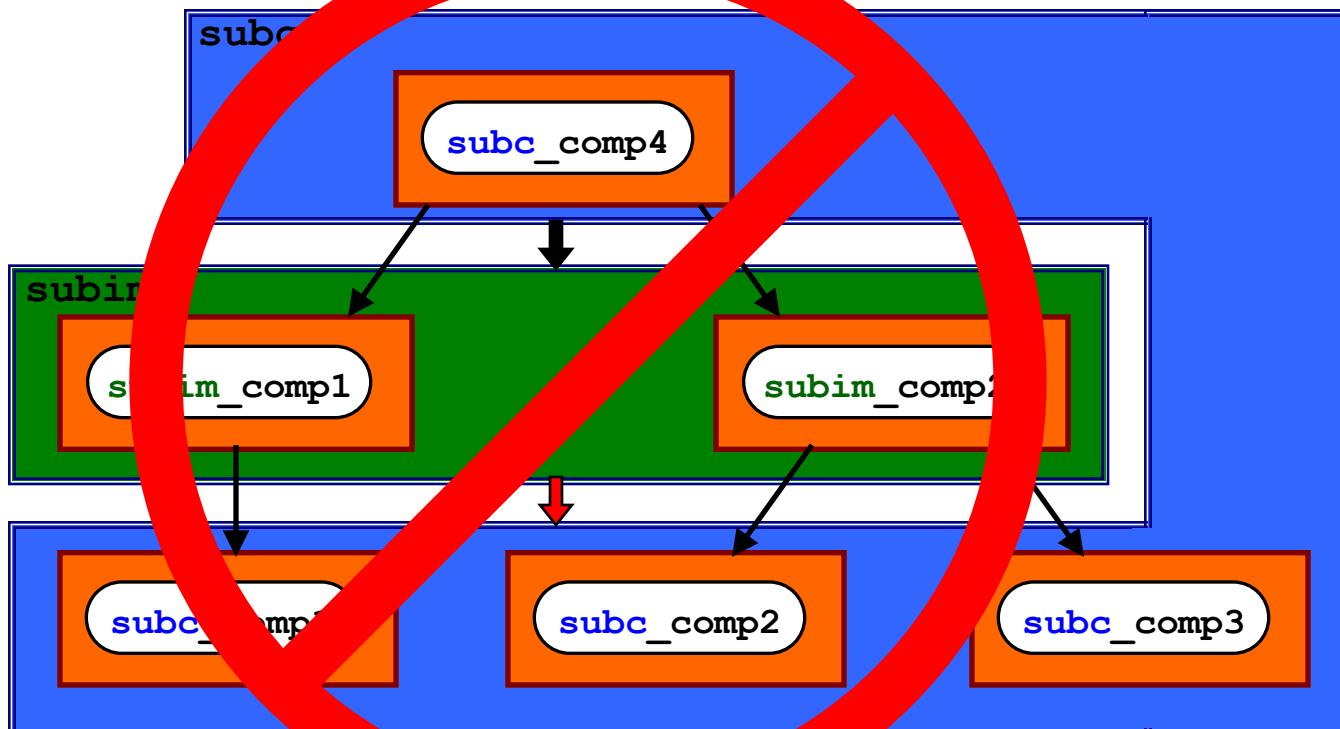
Package naming is more than just a convention:



1. Process & Architecture

Logical/Physical Name Cohesion

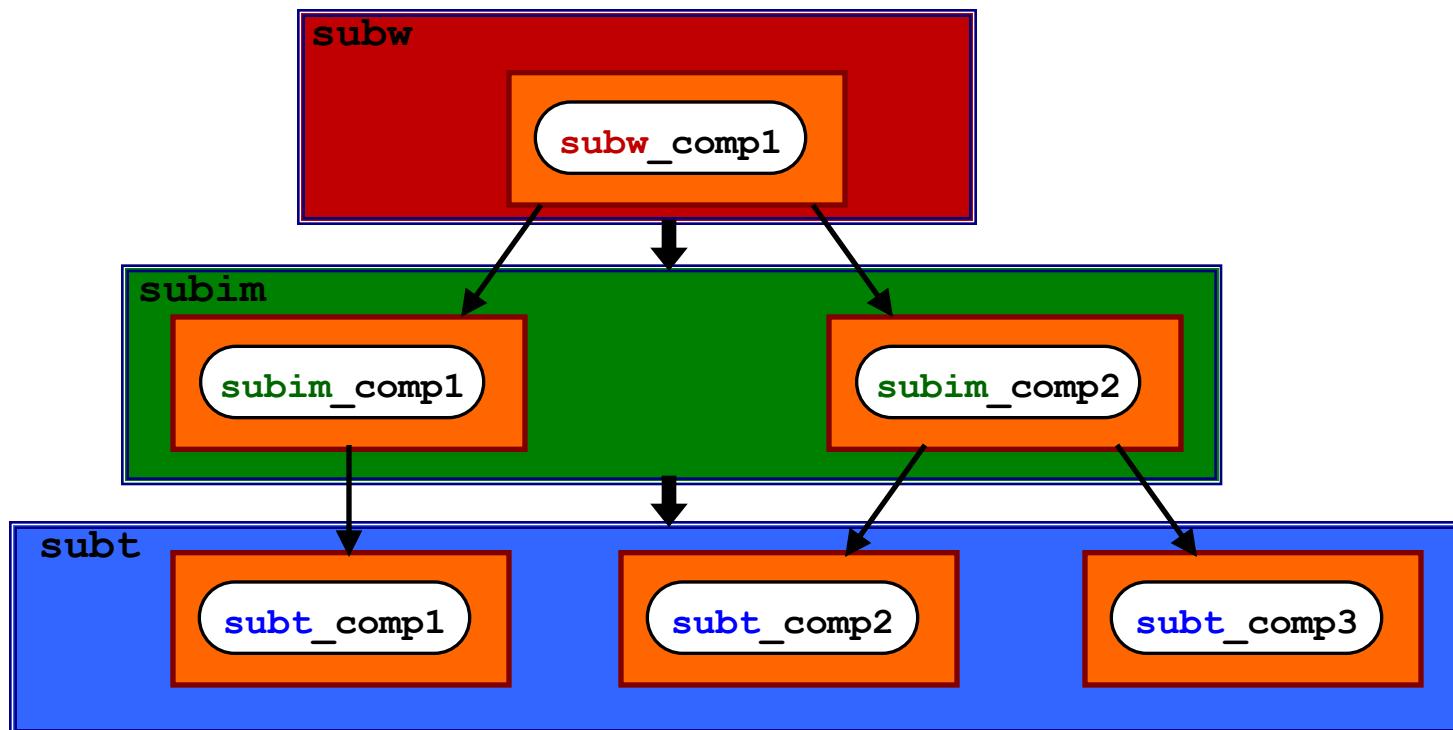
Package naming is more than just a convention:



1. Process & Architecture

Logical/Physical Name Cohesion

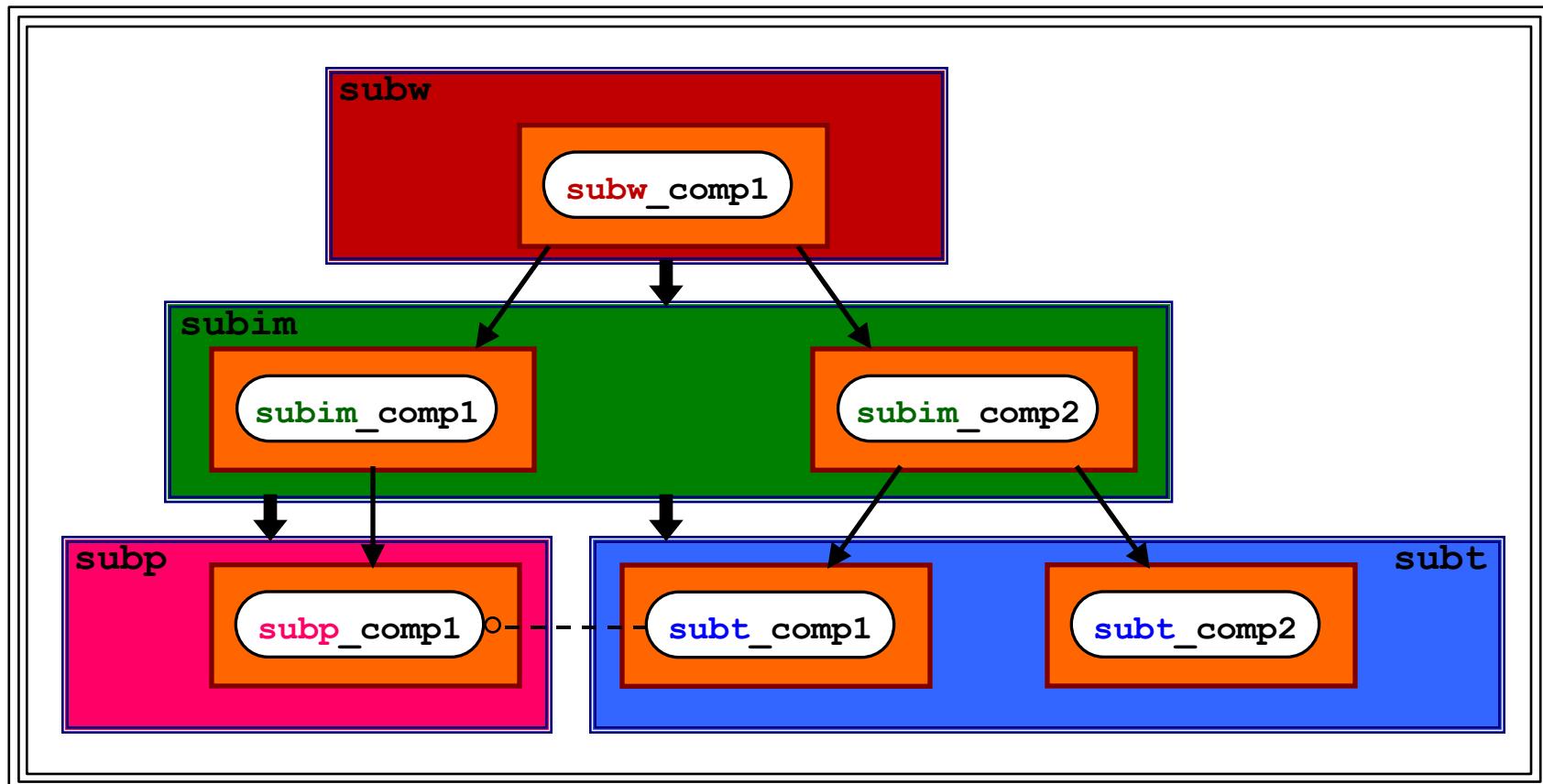
Package naming is more than just a convention:



1. Process & Architecture

Logical/Physical Name Cohesion

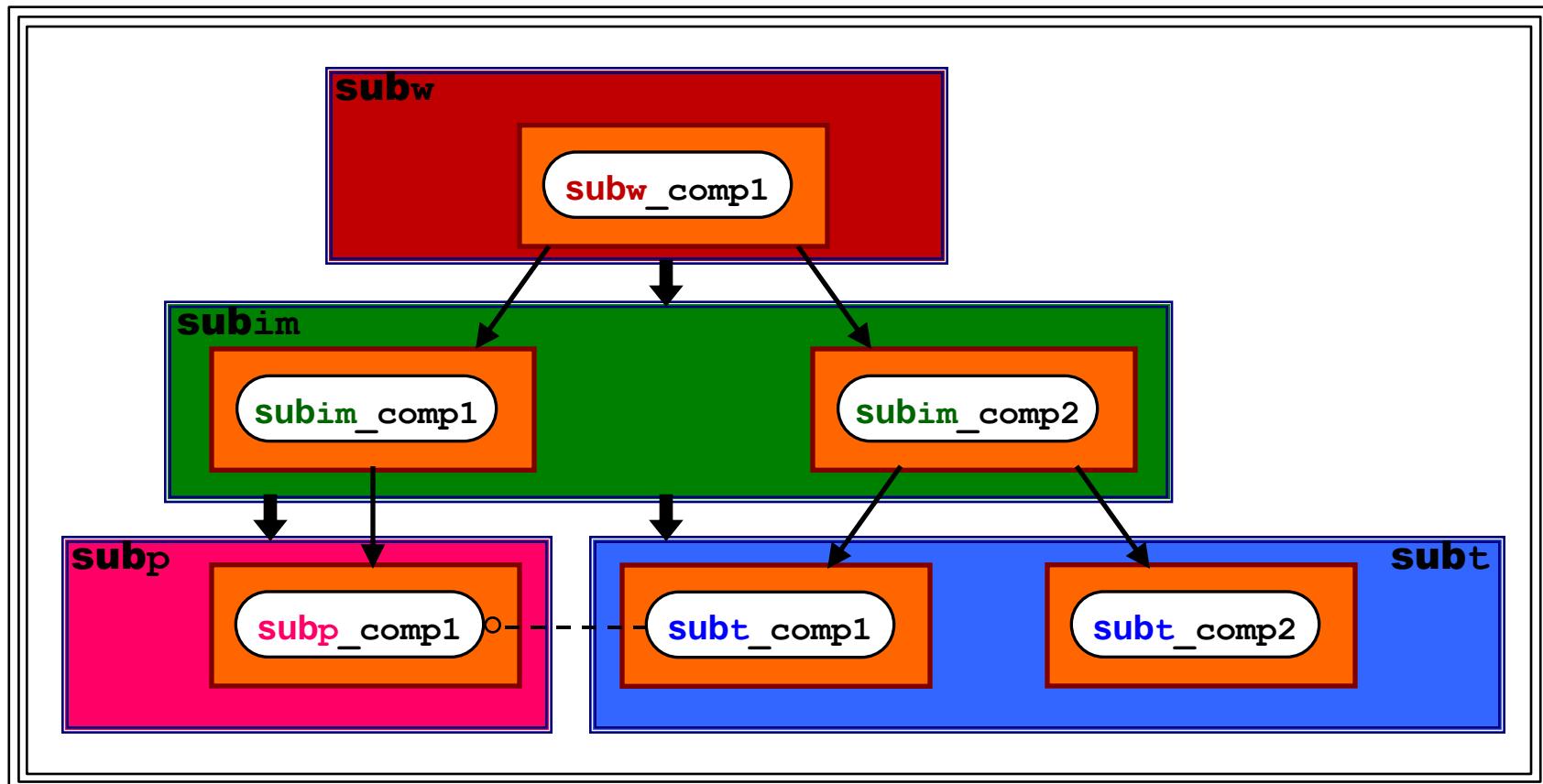
Package Group



1. Process & Architecture

Logical/Physical Name Cohesion

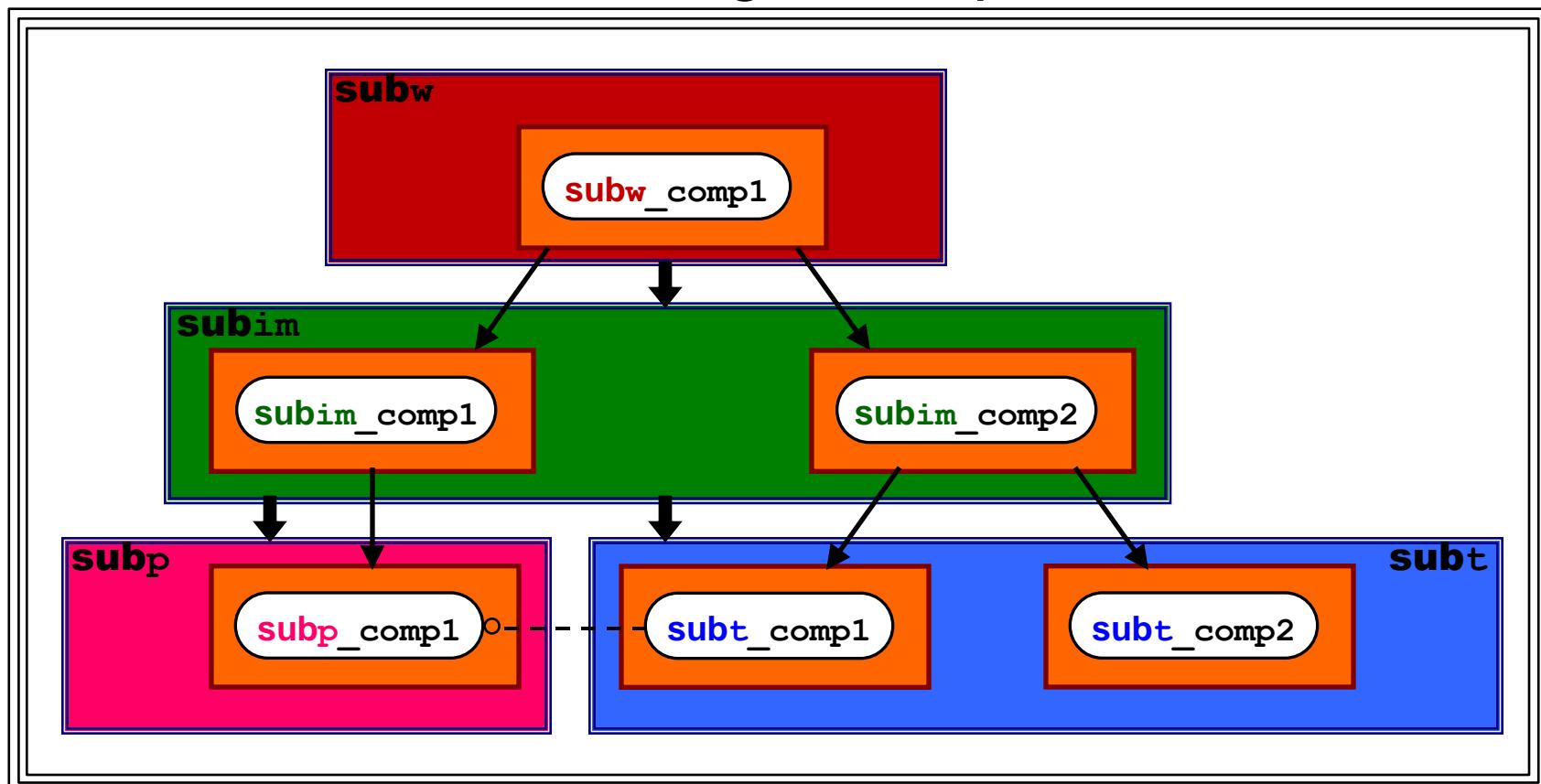
Package Group



1. Process & Architecture

Logical/Physical Name Cohesion

Package Group

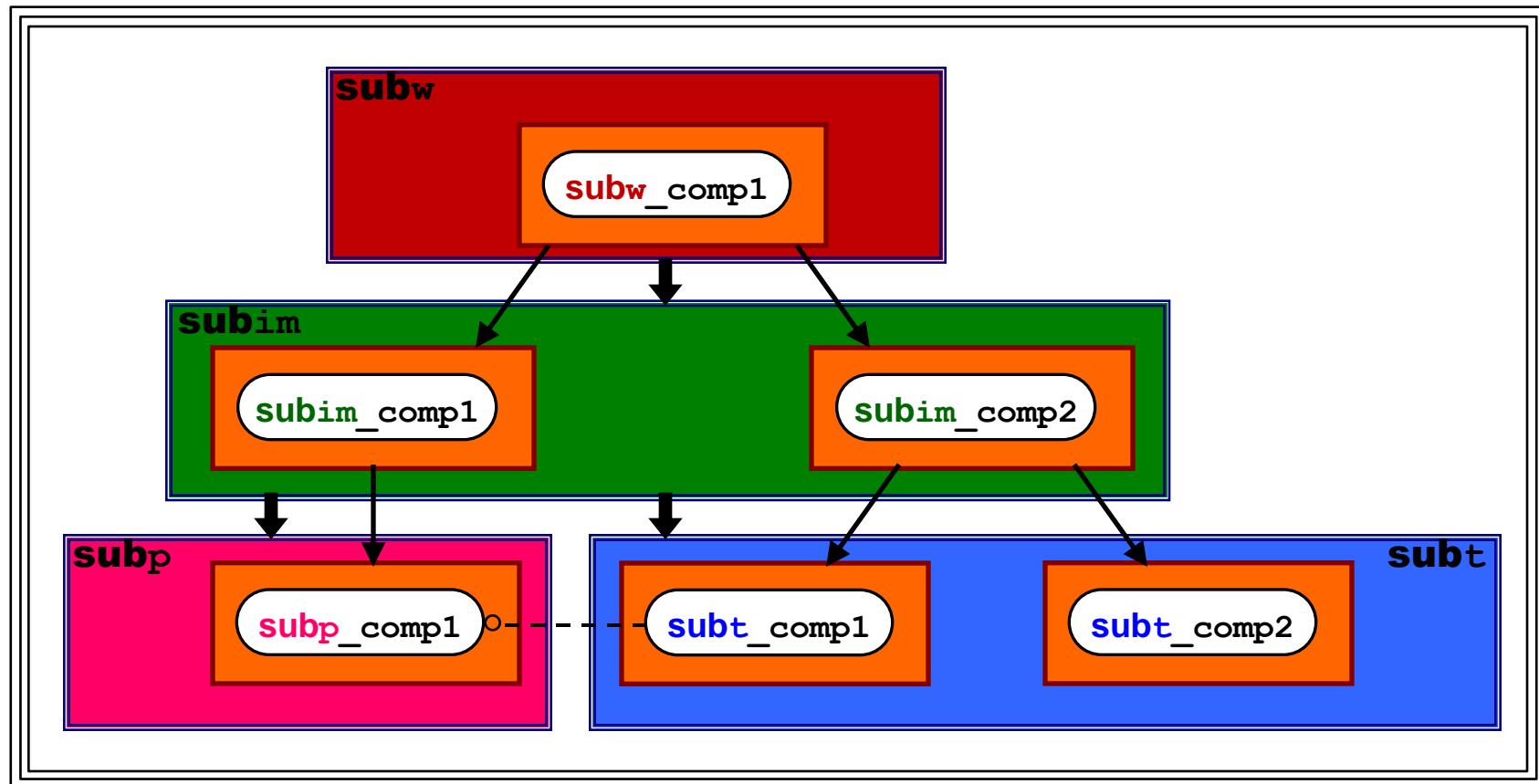


sub

1. Process & Architecture

Logical/Physical Name Cohesion

Package Group



sub ← Exactly Three Characters

1. Process & Architecture

Logical/Physical Name Cohesion

...

```
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
```

...

1. Process & Architecture

Logical/Physical Name Cohesion

...

```
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
```

...

Package Group: **bdl**

Package: **bdlt**

Component: bdlt_date

Class: bdlt::Date

Function: bdlt::Date::isValidYMD

1. Process & Architecture

Logical/Physical Name Cohesion

...

```
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
```

...

Package Group: **bdl**

Package: **bdl**t

Component: bdlt_date

Class: bdlt::Date

Function: bdlt::Date::isValidYMD

1. Process & Architecture

Logical/Physical Name Cohesion

...

bool flag = **bdlt::Date**::isValidYMD(1959, 3, 8);

...

Package Group: **bd1**

Package: **bdlt**

Component: **bdlt_date**

Class: **bdlt::Date**

Function: **bdlt::Date::isValidYMD**

1. Process & Architecture

Logical/Physical Name Cohesion

...

```
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
```

...

Package Group: **bd1**

Package: **bdlt**

Component: **bdlt_date**

Class: **bdlt::Date**

Function: **bdlt::Date::isValidYMD**

1. Process & Architecture

Logical/Physical Name Cohesion

...

```
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
```

...

Package Group: **bd1**

Package: **bdlt**

Component: **bdlt_date**

Class: **bdlt::Date**

Function: **bdlt::Date::isValidYMD**

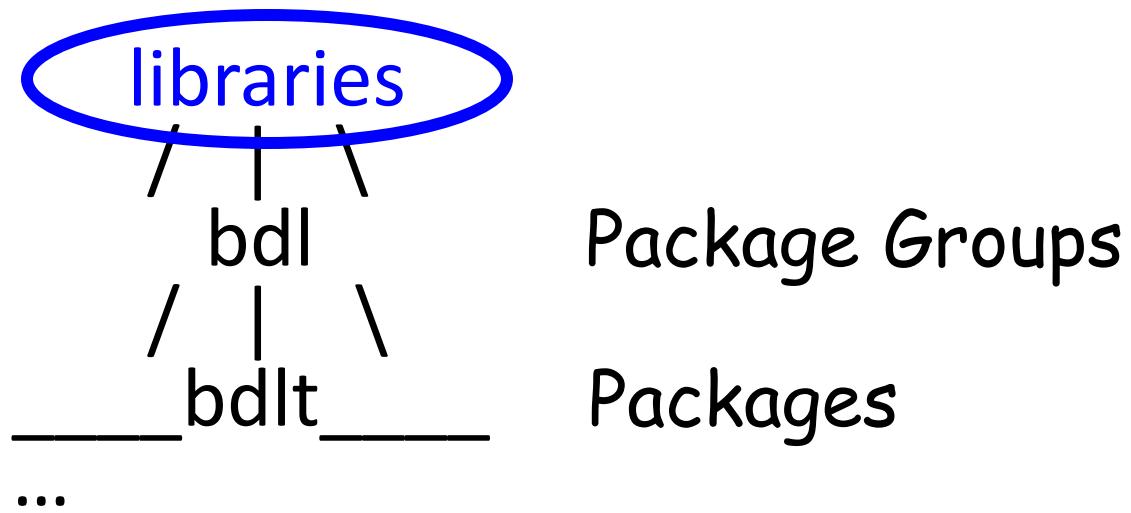
1. Process & Architecture

Logical/Physical Name Cohesion

...

```
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
```

...



...

bdlt_date.h Components
bdlt_date.cpp
bdlt_date.t.cpp

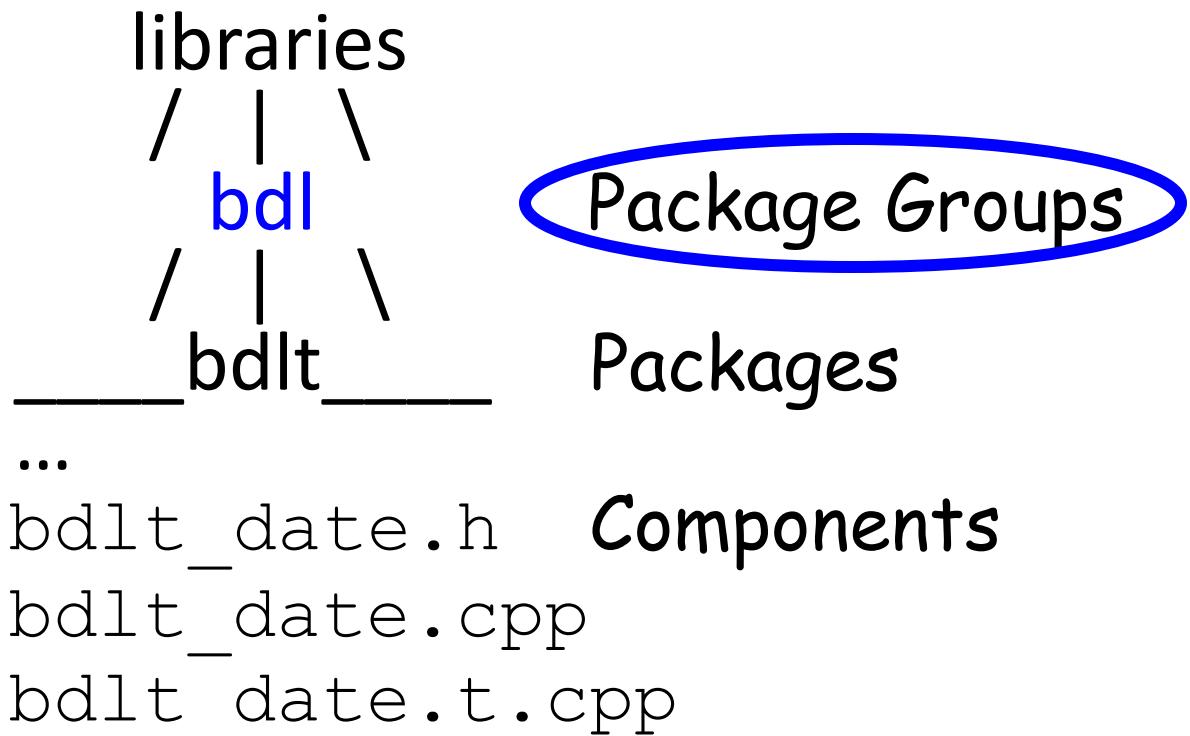
1. Process & Architecture

Logical/Physical Name Cohesion

...

```
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
```

...



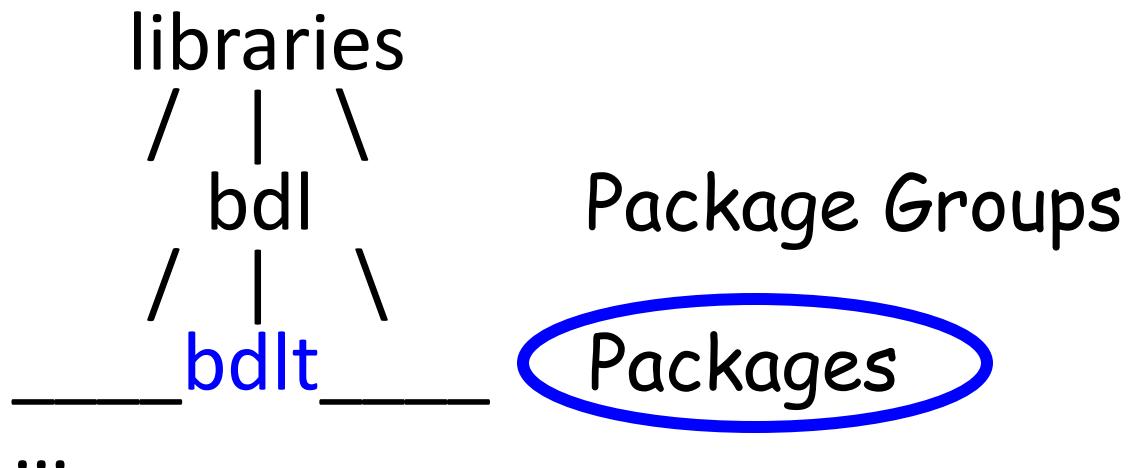
1. Process & Architecture

Logical/Physical Name Cohesion

...

```
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
```

...



...

bdlt_date.h Components
bdlt_date.cpp
bdlt_date.t.cpp

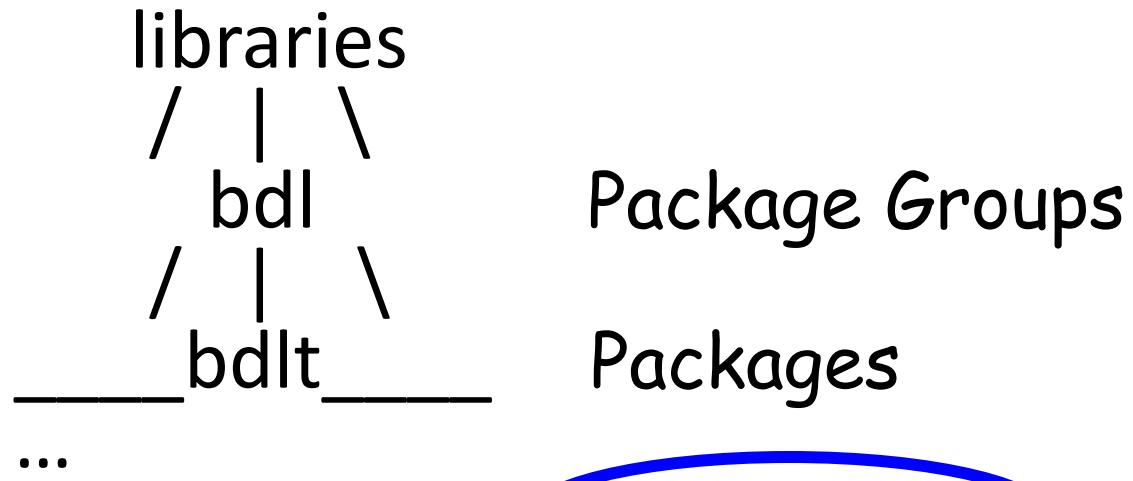
1. Process & Architecture

Logical/Physical Name Cohesion

...

```
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
```

...



```
bdlt_date.h  
bdlt_date.cpp  
bdlt_date.t.cpp
```

Components

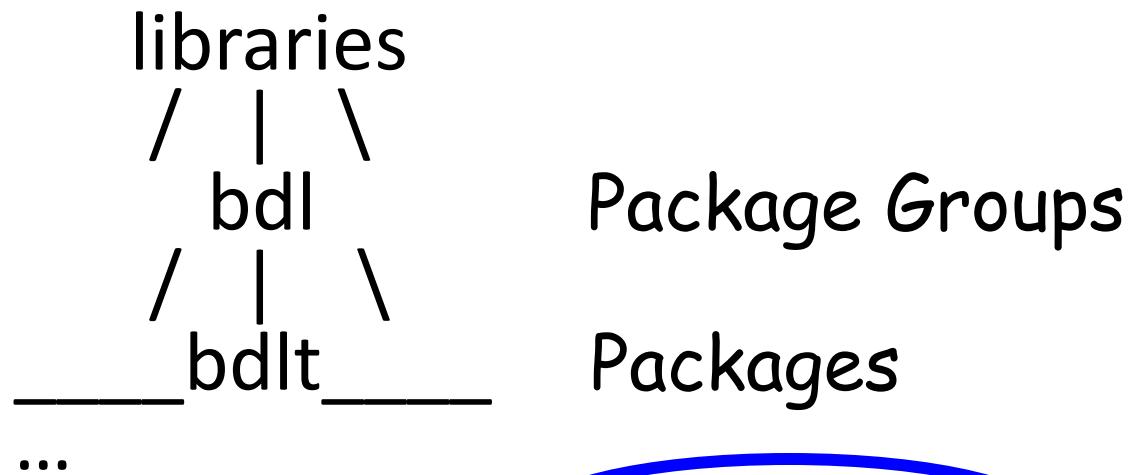
1. Process & Architecture

Logical/Physical Name Cohesion

...

```
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
```

...

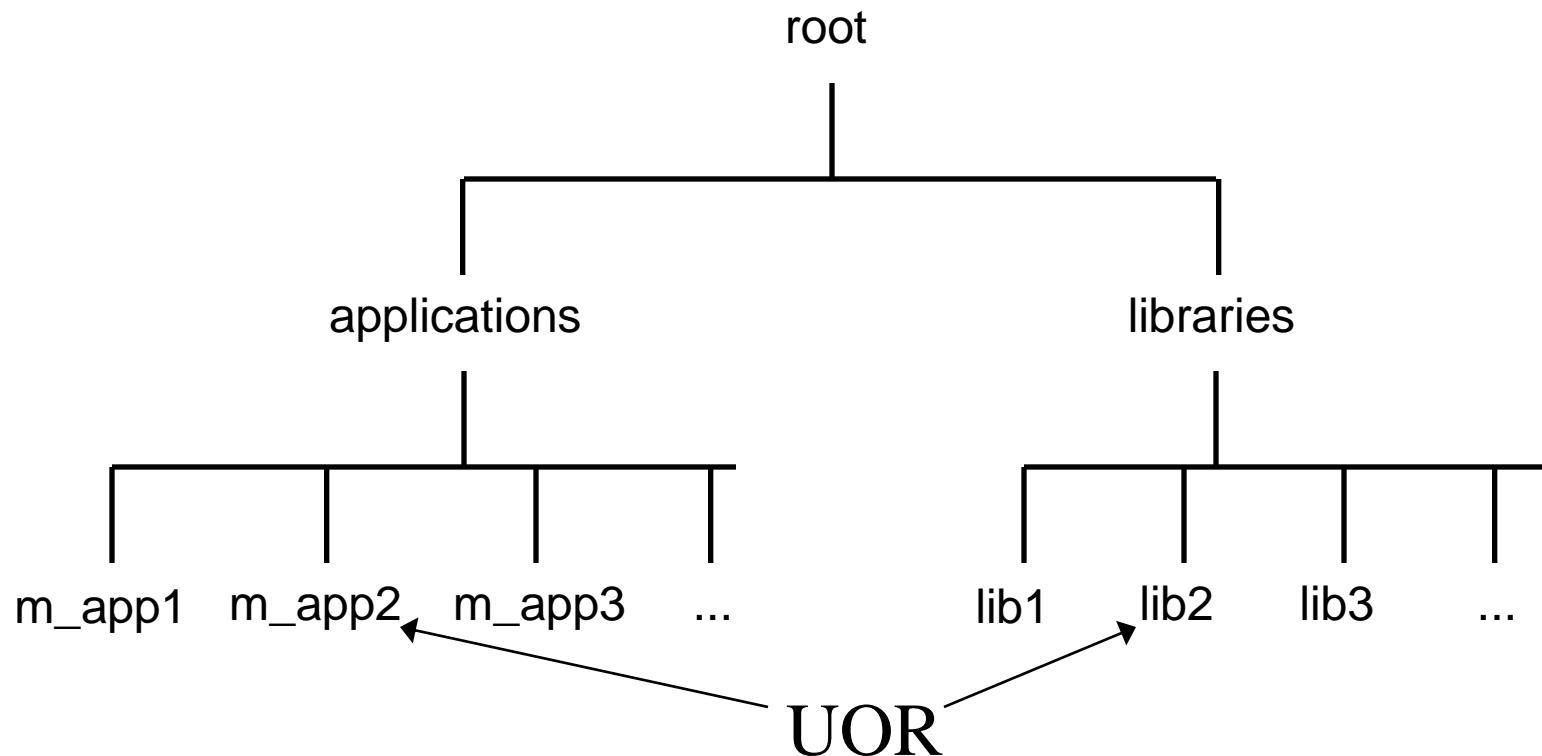


```
bdlt_date.h  
bdlt_date.cpp  
bdlt_date.t.cpp
```

Components

1. Process & Architecture

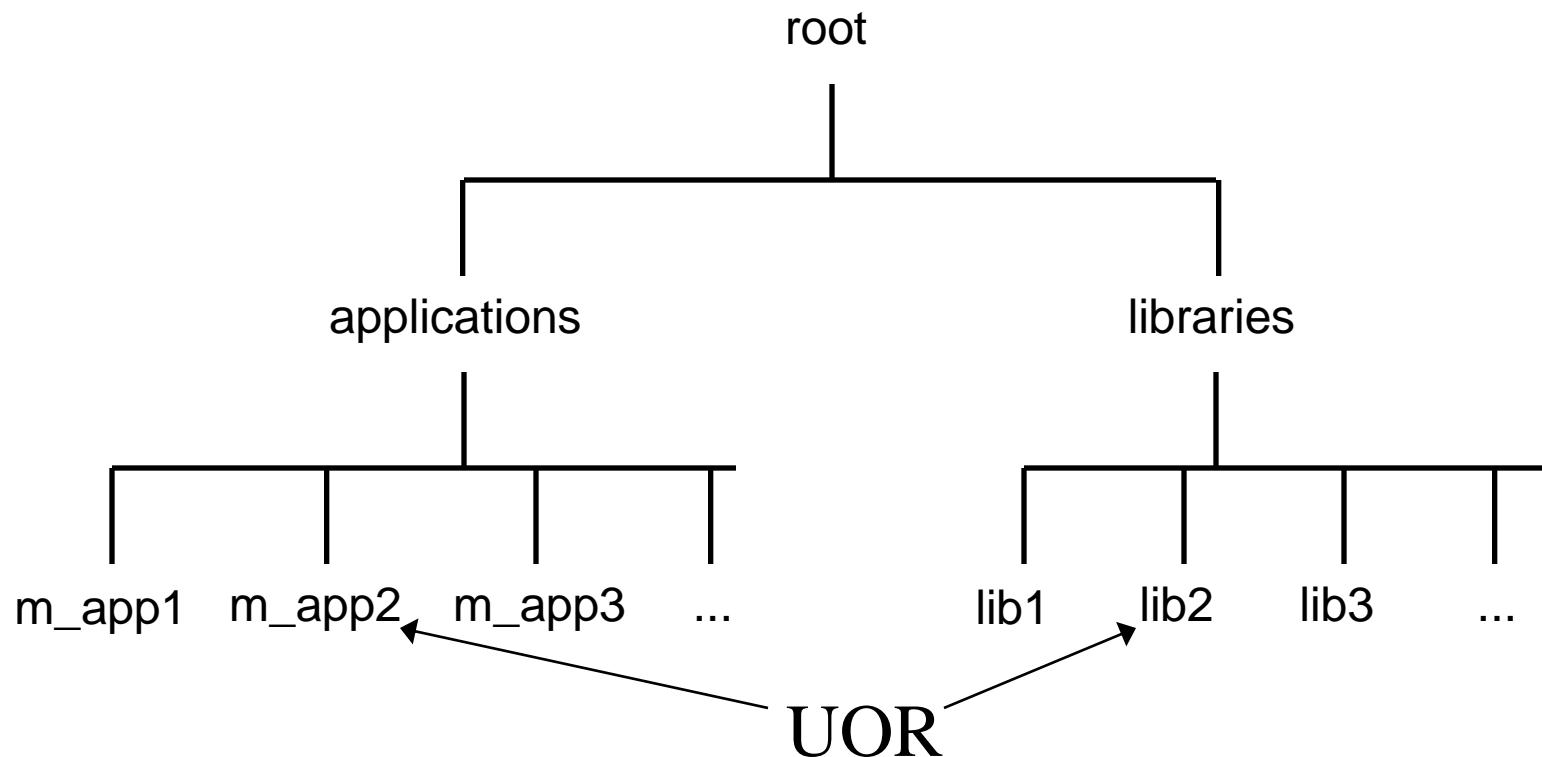
Unit Of Release



1. Process & Architecture

Unit Of Release

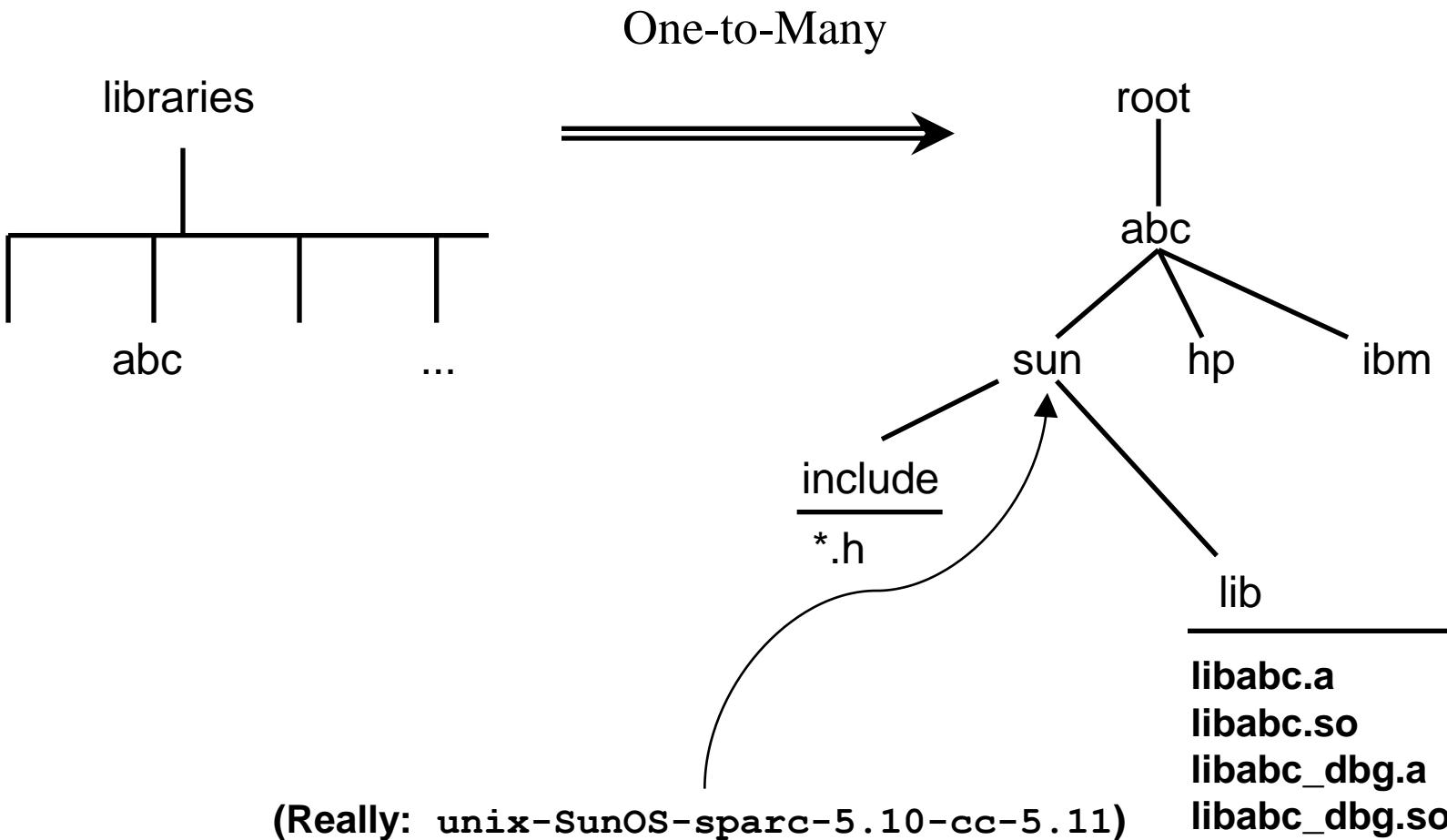
Package or Package Group



1. Process & Architecture

Development vs. Deployment

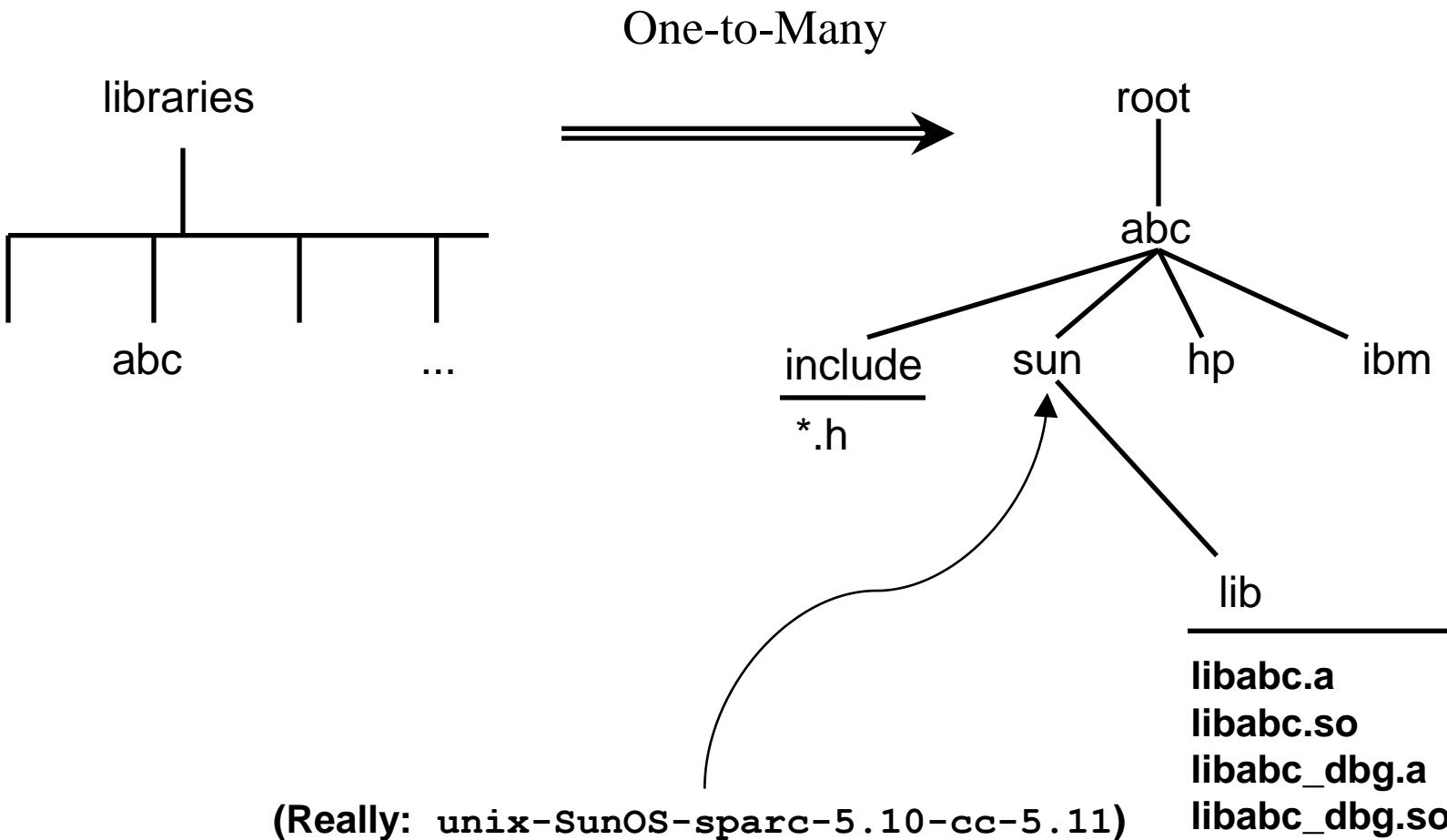
Source Code Deployment



1. Process & Architecture

Development vs. Deployment

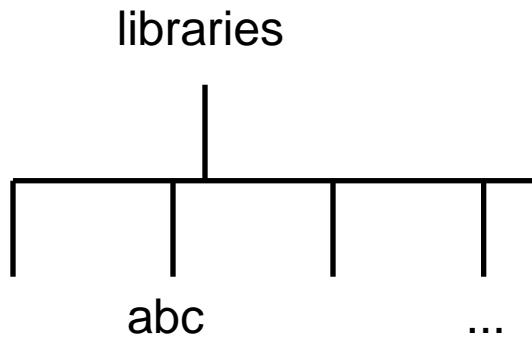
Source Code Deployment



1. Process & Architecture

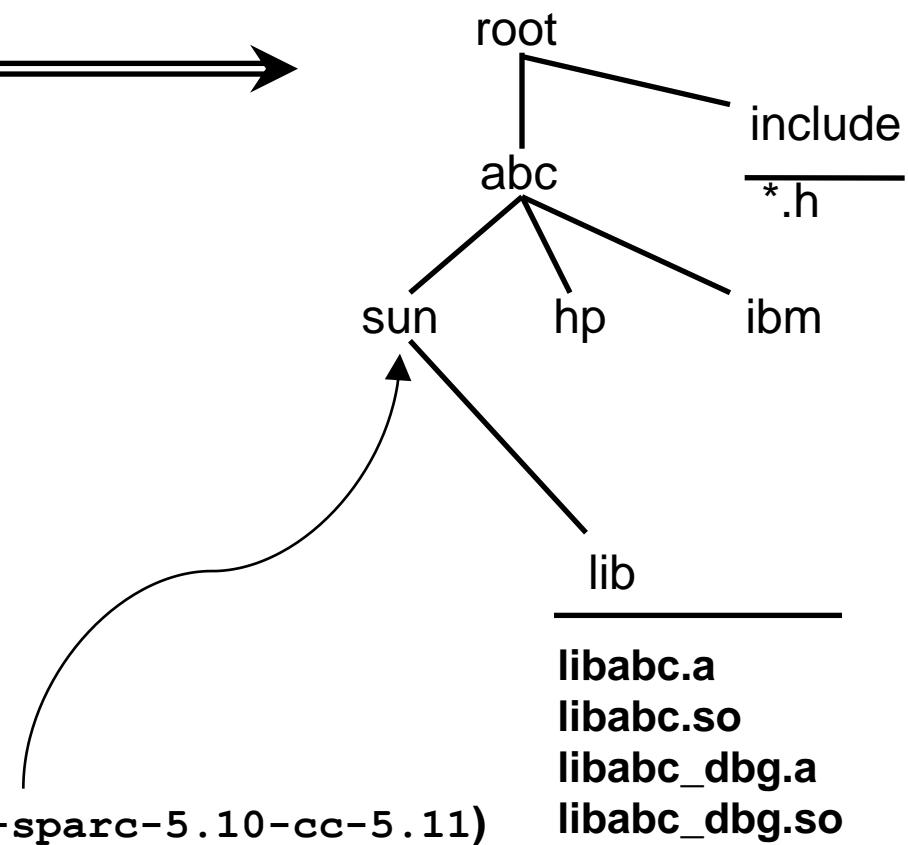
Development vs. Deployment

Source Code



One-to-Many

Deployment



(Really: unix-SunOS-sparc-5.10-cc-5.11)

1. Process & Architecture

Designing with Dependency in Mind

Good Physical Design...

1. Process & Architecture

Designing with Dependency in Mind

Good Physical Design...

- ✓ Is **not** an afterthought.

1. Process & Architecture

Designing with Dependency in Mind

Good Physical Design...

- ✓ Is not an afterthought.
- ✓ Is an integral part of logical design.

1. Process & Architecture

Designing with Dependency in Mind

Good Physical Design...

- ✓ Is not an afterthought.
- ✓ Is an integral part of logical design.
- ✓ Is something we first consider long before we start to write code.

1. Process & Architecture

Designing with Dependency in Mind

Good Physical Design...

- ✓ Is not an afterthought.
- ✓ Is an integral part of logical design.
- ✓ Is something we first consider long before we start to write code.
- ✓ Is something we must consider when decomposing the problem itself!

Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment

Rendered as Fine-Grained Hierarchically Reusable Components.

Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment

Rendered as Fine-Grained Hierarchically Reusable Components.

2. Design & Implementation

Essential Strategies and Techniques

Integral to our design process are:

- a) Common Class Categories
- b) Unique *Vocabulary* Types
- c) Design By Contract
- d) Appropriately *Narrow* Contracts
- e) An Overriding Customer Focus

2. Design & Implementation

Essential Strategies and Techniques

Integral to our design process are:

- a) Common Class Categories
- b) Unique *Vocabulary* Types
- c) Design By Contract
- d) Appropriately *Narrow* Contracts
- e) An Overriding Customer Focus

2. Design & Implementation

The *Value* of a “Value”

Getting Started:

- Not all useful C++ classes are value types.

2. Design & Implementation

The *Value* of a “Value”

Getting Started:

- Not all useful C++ classes are value types.
- Still, value types form an important category.

2. Design & Implementation

The *Value* of a “Value”

Getting Started:

- Not all useful C++ classes are value types.
- Still, value types form an important category.
- Let's begin with understanding properties of value types.

2. Design & Implementation

The *Value* of a “Value”

Getting Started:

- Not all useful C++ classes are value types.
- Still, value types form an important category.
- Let's begin with understanding properties of value types.
- Then generalize to build a small type-category hierarchy.

2. Design & Implementation

So, what do we mean by “value”?

```
class Date {  
    short d_year;  
    char d_month;  
    char d_day;  
  
public:  
    // ...  
    int year();  
    int month();  
    int day();  
};
```

2. Design & Implementation

So, what do we mean by “value”?

```
class Date {  
    short d_year;  
    char d_month;  
    char d_day;  
  
public:  
    // ...  
    int year();  
    int month();  
    int day();  
};
```



2. Design & Implementation

So, what do we mean by “value”?

```
class Date {  
    short d_year;  
    char d_month;  
    char d_day;  
  
public:  
    // ...  
    int year();  
    int month();  
    int day();  
};
```



2. Design & Implementation

So, what do we mean by “value”?

```
class Date {  
    short d_year;  
    char d_month;  
    char d_day;  
  
public:  
    // ...  
    int year() const;  
    int month() const;  
    int day() const;  
};
```

2. Design & Implementation

So, what do we mean by “value”?

```
class Date {  
    short d_year;  
    char d_month;  
    char d_day;  
  
public:  
    // ...  
    int year();  
    int month();  
    int day();  
};
```

```
class Date {  
    int d_serial;  
  
public:  
    // ...  
    int year();  
    int month();  
    int day();  
};
```

2. Design & Implementation

So, what do we mean by “value”?

```
class Date {  
    short d_year;  
    char d_month;  
    char d_day;  
public:  
    // ...  
    int year();  
    int month();  
    int day();  
};
```

```
class Date {  
    int d_serial;  
public:  
    // ...  
    int year();  
    int month();  
    int day();  
};
```

2. Design & Implementation

So, what do we mean by “value”?

Salient Attributes

```
int year();  
int month();  
int day();
```

2. Design & Implementation

So, what do we mean by “value”?

Salient Attributes

The documented set of (observable) named attributes of a type T that must respectively “have” (refer to) *the same* value in order for two instances of T to “have” (refer to) *the same* value.

2. Design & Implementation

So, what do we mean by “value”?

```
class Time {  
    char d_hour;  
    char d_minute;  
    char d_second;  
    short d_millisec;  
  
public:  
    // ...  
    int hour();  
    int minute();  
    int second();  
    int millisecond();  
};
```

```
class Time {  
    int d_mSeconds;  
  
public:  
    // ...  
    int hour();  
    int minute();  
    int second();  
    int millisecond();  
};
```

2. Design & Implementation

So, what do we mean by “value”?

```
class Time {
```

Internal Representation

```
public:
```

```
//
```

```
int hour();  
int minute();  
int second();  
int millisecond();
```

```
} ;
```

```
class Time {
```

Internal Representation

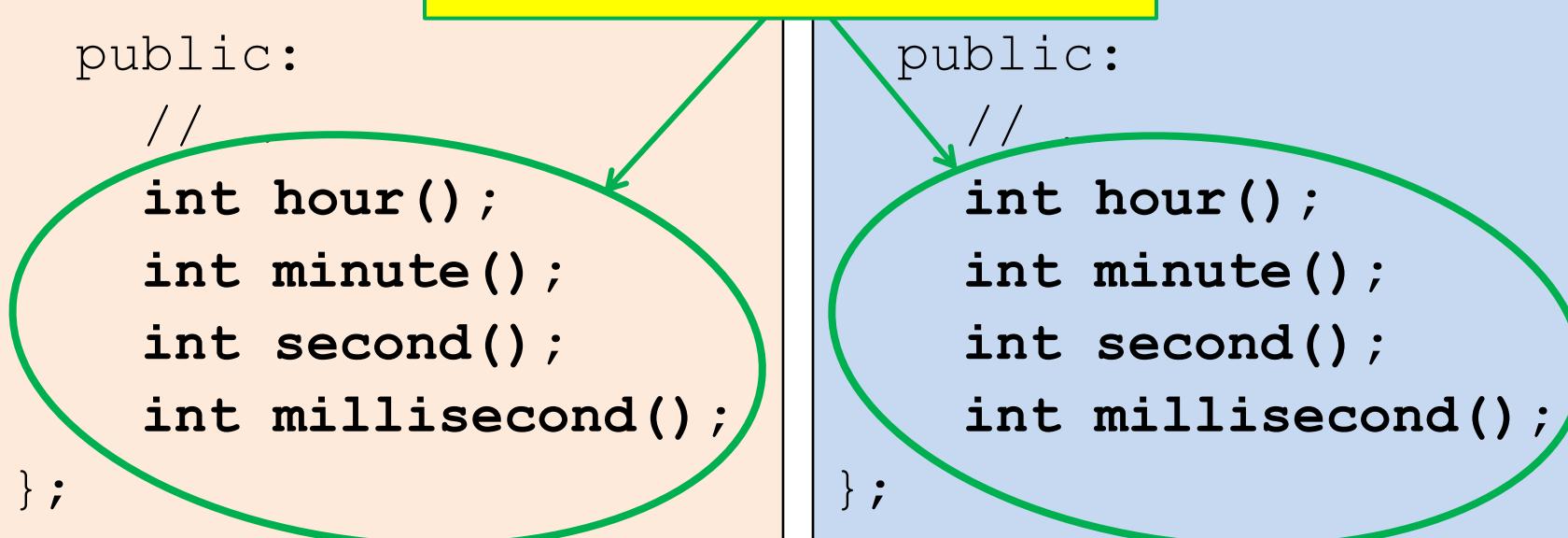
```
public:
```

```
//
```

```
int hour();  
int minute();  
int second();  
int millisecond();
```

```
} ;
```

VALUE



2. Design & Implementation

So, what do we mean by “value”?

Value:

2. Design & Implementation

So, what do we mean by “value”?

Value:

- An “interpretation” of object state –

2. Design & Implementation

So, what do we mean by “value”?

Value:

- An “interpretation” of object state –
i.e., Salient Attributes, not the object state itself.

2. Design & Implementation

So, what do we mean by “value”?

Value:

- An “interpretation” of object state –
i.e., Salient Attributes, not the object state itself.
- No non-object state is relevant.

2. Design & Implementation

What are “Salient Attributes”?

2. Design & Implementation

What are “Salient Attributes”?

```
class vector {  
    T             *d_array_p;  
    size_type     d_capacity;  
    size_type     d_size;  
    // ...  
public:  
    vector();  
    vector(const vector<T>& orig);  
    // ...  
};
```

2. Design & Implementation

What are “Salient Attributes”?

```
class vector {  
    T             *d_array_p;  
    size_type     d_capacity;  
    size_type     d_size;  
    // ...  
public:  
    vector();  
    vector(const vector<T>& orig);  
    // ...  
};
```

2. Design & Implementation

What are “Salient Attributes”?

Consider `std::vector<int>:`

What are its *salient attributes*?

2. Design & Implementation

What are “Salient Attributes”?

Consider `std::vector<int>:`

What are its *salient attributes*?

1. The number of elements: `size()`.

2. Design & Implementation

What are “Salient Attributes”?

Consider `std::vector<int>:`

What are its *salient attributes*?

1. The number of elements: `size()`.
2. The *values* of the respective elements.

2. Design & Implementation

What are “Salient Attributes”?

Consider `std::vector<int>:`

What are its *salient attributes*?

1. The number of elements: `size()`.
2. The *values* of the respective elements.
3. What about `capacity()`?

2. Design & Implementation

What are “Salient Attributes”?

Consider `std::vector<int>:`

What are its *salient attributes*?

1. The number of elements: `size()`.
2. The *values* of the respective elements.
- ~~3. What about `capacity()`?~~

How is the client supposed to know for sure?

2. Design & Implementation

What are “Salient Attributes”?

Consider `std::vector<int>:`

What are its *salient attributes*?

1. The number of elements: `size()`.
2. The *values* of the respective elements.
- ~~3. What about `capacity()`?~~

How is the client supposed to know for sure?

They must be documented (somewhere).

2. Design & Implementation

Value-Semantic Properties

Note that two *distinct* objects **a** and **b** of type T that have *the same value* might not exhibit “the same” ***observable behavior***.

2. Design & Implementation

Value-Semantic Properties

Note that two *distinct* objects **a** and **b** of type T that have *the same value* might not exhibit “the same” **observable behavior**.

```
std::vector<int> a;  
  
a.reserve(65536);  
std::vector<int> b(a); // is capacity copied?  
assert(a == b)  
a.resize(65536);           // no reallocation!  
b.resize(65536);           // memory allocation?
```

2. Design & Implementation

Value-Semantic Properties

Note that two *distinct* objects **a** and **b** of type T that have *the same value* might not exhibit “the same” **observable behavior**.

```
std::vector<int> a;  
  
a.reserve(65536);  
std::vector<int> b(a); // is capacity copied?  
assert(a == b)  
a.resize(65536);           // no reallocation!  
b.resize(65536);           // memory allocation?
```

2. Design & Implementation

Value-Semantic Properties

Note that two *distinct* objects **a** and **b** of type T that have *the same value* might not exhibit “the same” **observable behavior**.

```
std::vector<int> a;  
  
a.reserve(65536);  
std::vector<int> b(a); // is capacity copied?  
assert(a == b)  
a.resize(65536); // no reallocation!  
b.resize(65536); // memory allocation?
```

2. Design & Implementation

Value-Semantic Properties

Note that two *distinct* objects **a** and **b** of type T that have *the same value* might not exhibit “the same” **observable behavior**.

HOWEVER

2. Design & Implementation

Value-Semantic Properties

Note that two *distinct* objects **a** and **b** of type T that have *the same value* might not exhibit “the same” **observable behavior**.

HOWEVER

1. If **a** and **b** initially have the same value, and

2. Design & Implementation

Value-Semantic Properties

Note that two *distinct* objects **a** and **b** of type T that have *the same value* might not exhibit “the same” **observable behavior**.

HOWEVER

1. If **a** and **b** initially have the same value, and
2. the same operation is applied to each object, then

2. Design & Implementation

Value-Semantic Properties

Note that two *distinct* objects **a** and **b** of type T that have *the same value* might not exhibit “the same” **observable behavior**.

HOWEVER

1. If **a** and **b** initially have the same value, and
2. the same operation is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)

2. Design & Implementation

Value-Semantic Properties

Note that two *distinct* objects **a** and **b** of type T that have *the same value* might not exhibit “the same” **observable behavior**.

HOWEVER

1. If **a** and **b** initially have the same value, and
2. the same operation is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will again have the same value!**

2. Design & Implementation

Value-Semantic Properties

1. If **a** and **b** initially have the *same value*, and
2. the *same operation* is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will again have the *same value*!**

2. Design & Implementation

Value-Semantic Properties

SUBTLE ESSENTIAL PROPERTY OF VALUE

1. If **a** and **b** initially have the *same value*, and
2. the *same operation* is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will again have the *same value*!**

2. Design & Implementation

Value-Semantic Properties

Deciding what is (not) salient
is surprisingly important.

SUBTLE ESSENTIAL PROPERTY OF VALUE

1. If **a** and **b** initially have the same value, and
2. the same operation is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will again have the same value!**

2. Design & Implementation

Value-Semantic Properties

There is a lot more to this story!

Deciding what is (not) salient
is surprisingly important.

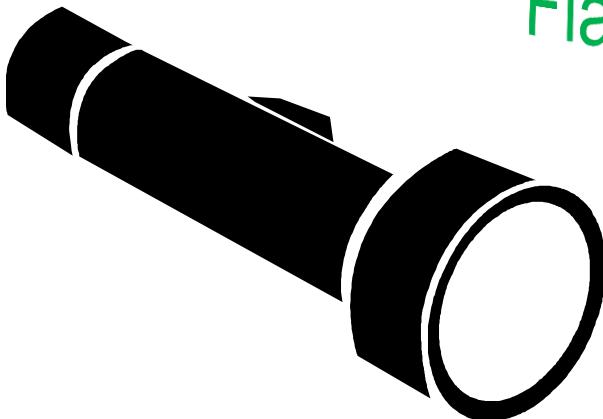
SUBTLE ESSENTIAL PROPERTY OF VALUE

1. If **a** and **b** initially have the same value, and
2. the same operation is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will again have the same value!**

2. Design & Implementation

Does state *always* imply a “value”?

Flashlight Object



2. Design & Implementation

Does state *always* imply a “value”?

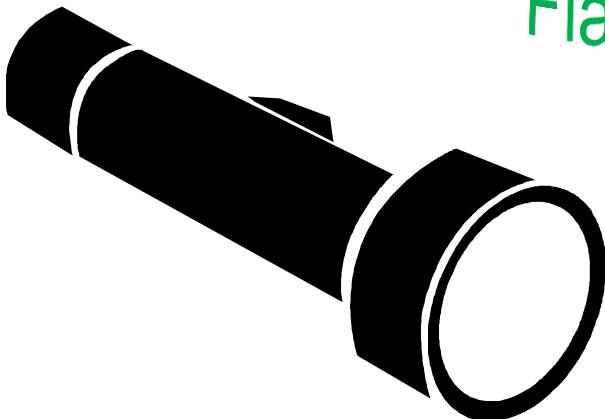
Flashlight Object



2. Design & Implementation

Does state *always* imply a “value”?

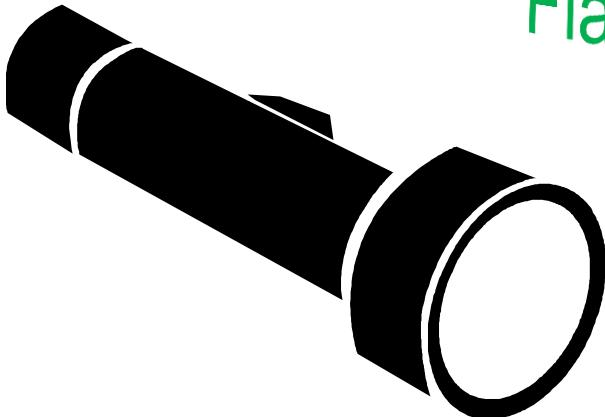
Flashlight Object



2. Design & Implementation

Does state *always* imply a “value”?

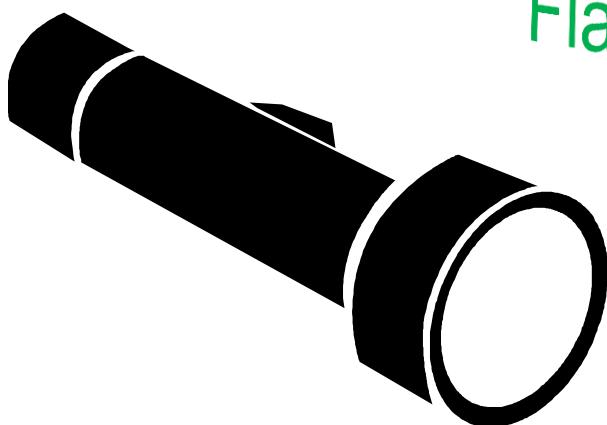
Flashlight Object



What is its state?

2. Design & Implementation

Does state *always* imply a “value”?



Flashlight Object

What is its state? OFF

2. Design & Implementation

Does state *always* imply a “value”?

Flashlight Object



What is its state?

2. Design & Implementation

Does state *always* imply a “value”?

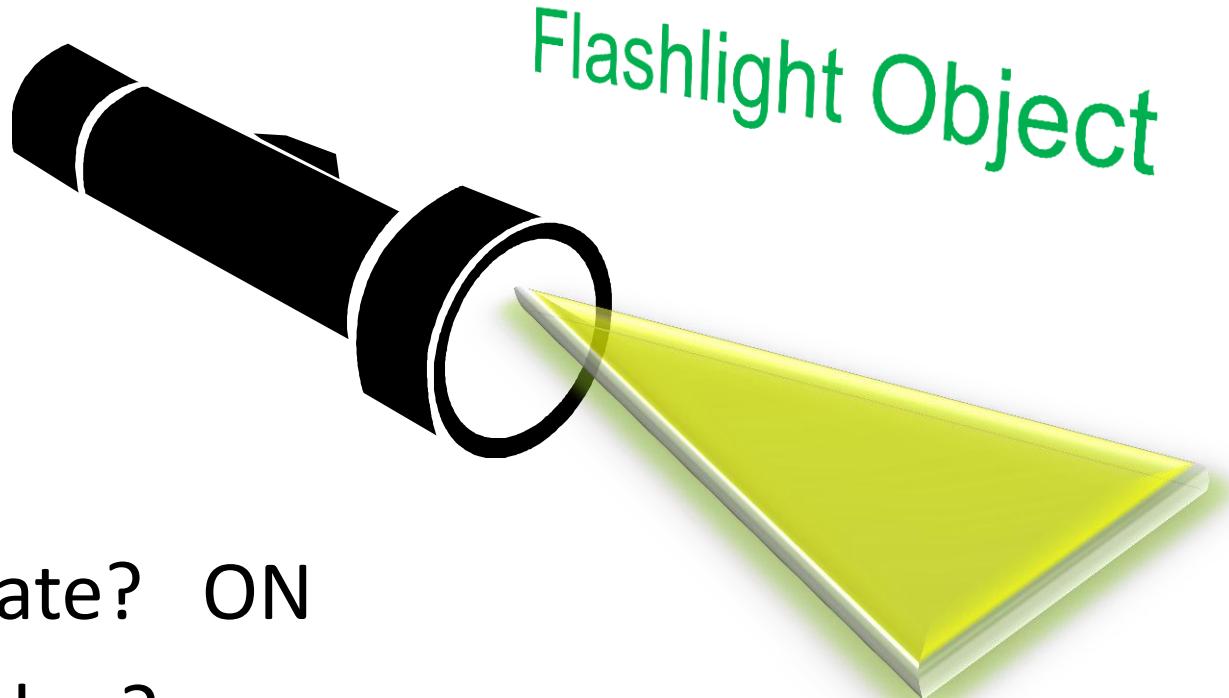
Flashlight Object



What is its state? ON

2. Design & Implementation

Does state *always* imply a “value”?



What is its state? ON

What is its value?

2. Design & Implementation

Does state *always* imply a “value”?

Flashlight Object



What is its state? ON

What is its value? ?

2. Design & Implementation

Does state *always* imply a “value”?

Flashlight Object



What is its state? ON

What is its value? £5.00 ?

2. Design & Implementation

Does state *always* imply a “value”?

Flashlight Object



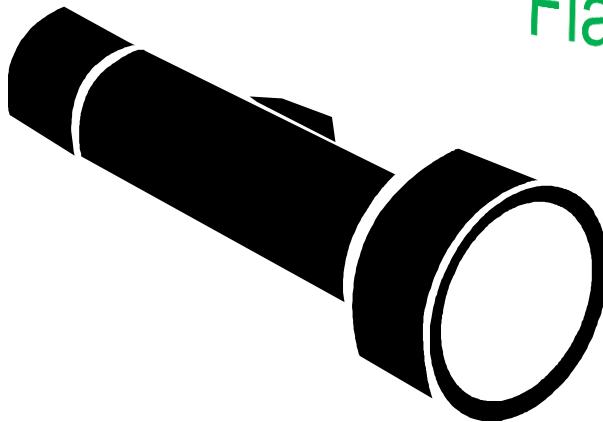
What is its state? ON

What is its value? **\$5.00 ?**

Cheap at half
the price!

2. Design & Implementation

Does state *always* imply a “value”?



Flashlight Object

What is its state? ON

What is its value? ?

Any notion of “value”
here would be artificial!

2. Design & Implementation

Does **state** *always* imply a “**value**”?

Not every ***stateful*** object has an ***obvious*** value.

2. Design & Implementation

Does **state** *always* imply a “**value**”?

Not every *stateful* object has an *obvious* value.

- TCP/IP Socket
- Thread Pool
- Condition Variable
- Mutex Lock
- Reader/Writer Lock
- Scoped Guard

2. Design & Implementation

Does state *always* imply a “value”?

Not every *stateful* object has an *obvious* value.

- TCP/IP Socket
- Thread Pool
- Condition Variable
- Mutex Lock
- Reader/Writer Lock
- Scoped Guard
- Base64 En(De)coder
- Expression Evaluator
- Language Parser
- Event Logger
- Object Persistor
- Widget Factory

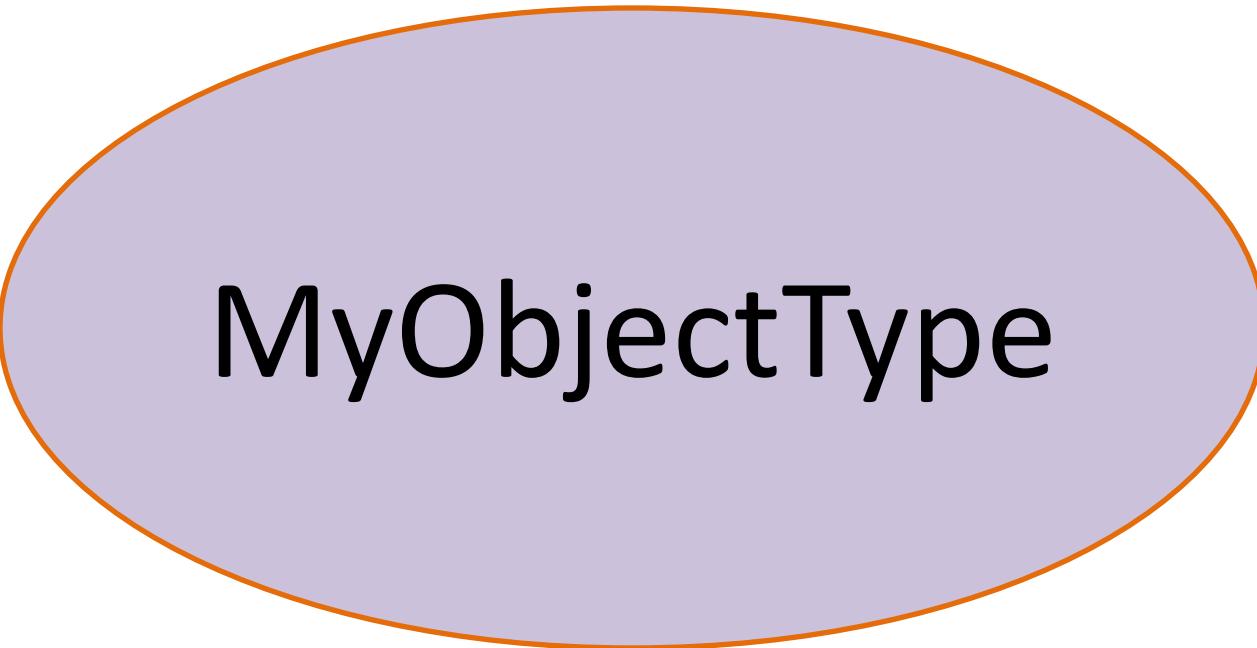
2. Design & Implementation

Does **state** *always* imply a “**value**”?

We refer to **stateful** objects that do not represent a value as “**Mechanisms**”.

2. Design & Implementation

Categorizing Object Types



MyObjectType

2. Design & Implementation

Categorizing Object Types

The first question: “Does it have state?”

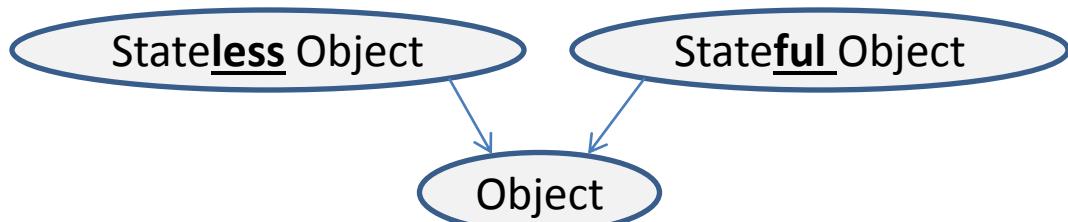
An oval shape with a blue border and a white interior, containing the word "Object".

Object

2. Design & Implementation

Categorizing Object Types

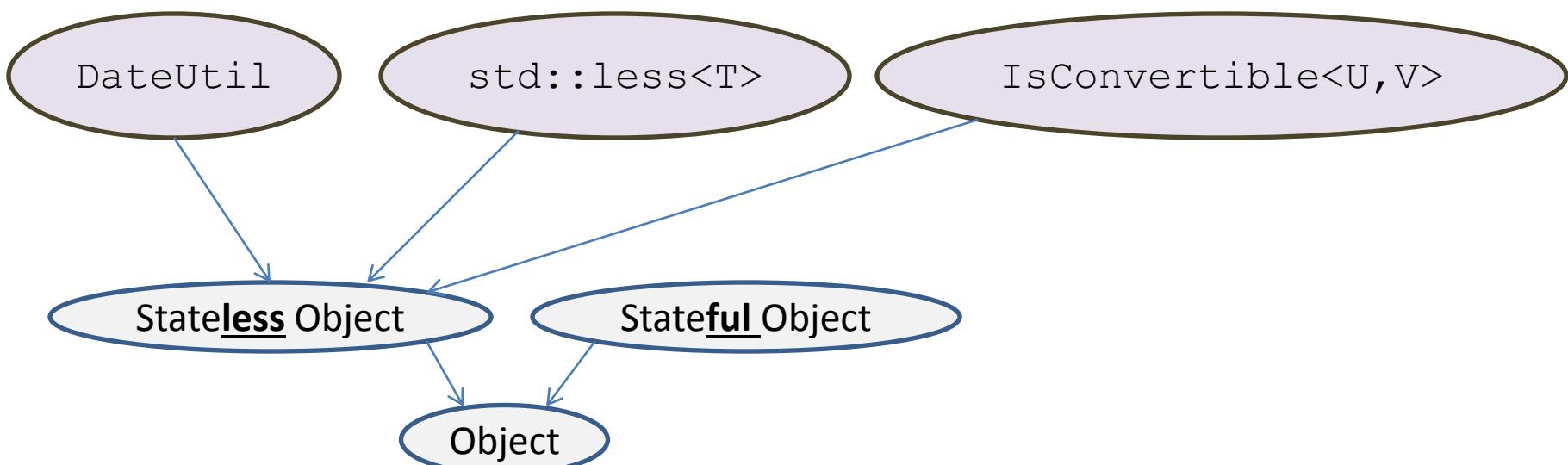
The first question: “Does it have state?”



2. Design & Implementation

Categorizing Object Types

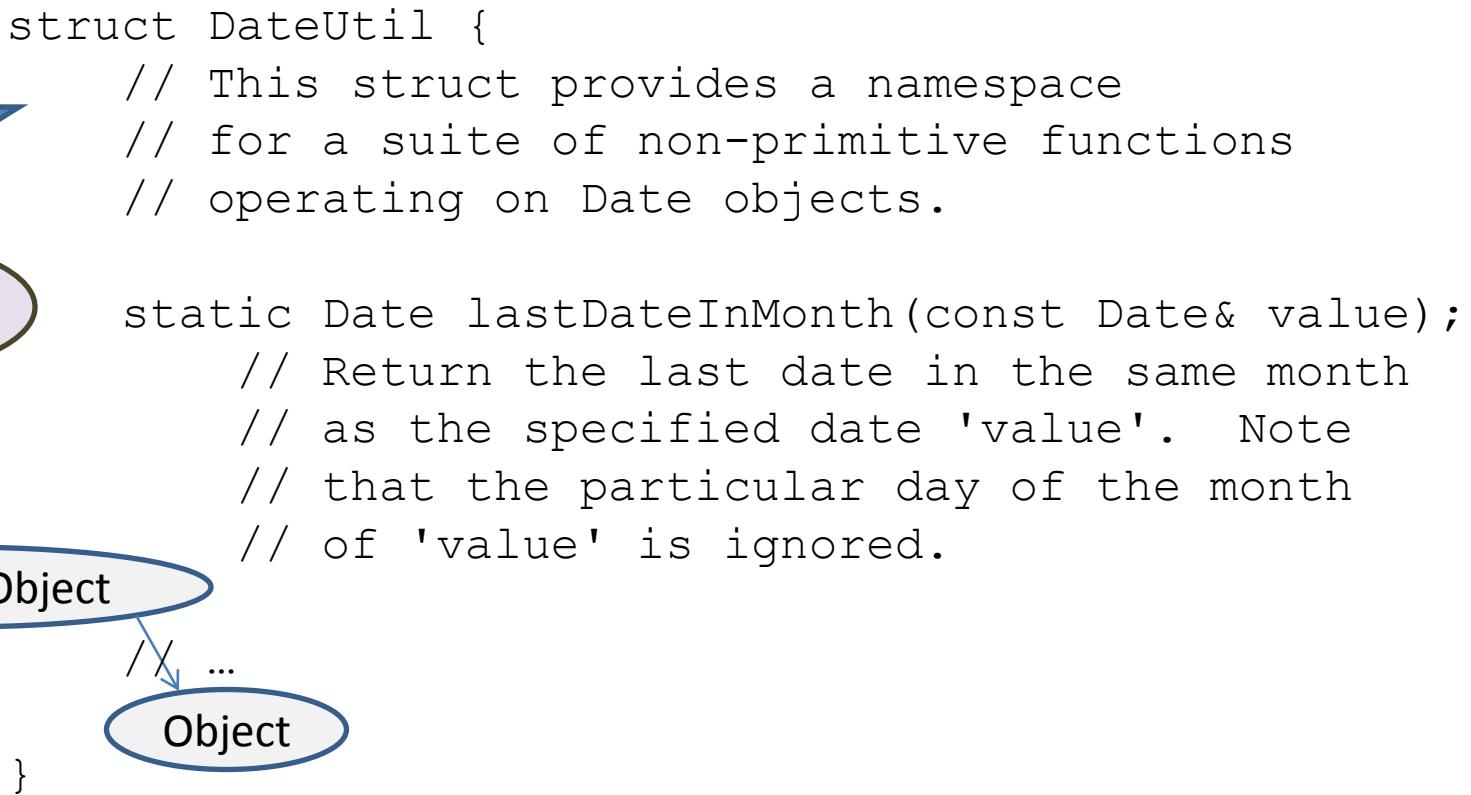
The first question: “Does it have state?”



2. Design & Implementation

Categorizing Object Types

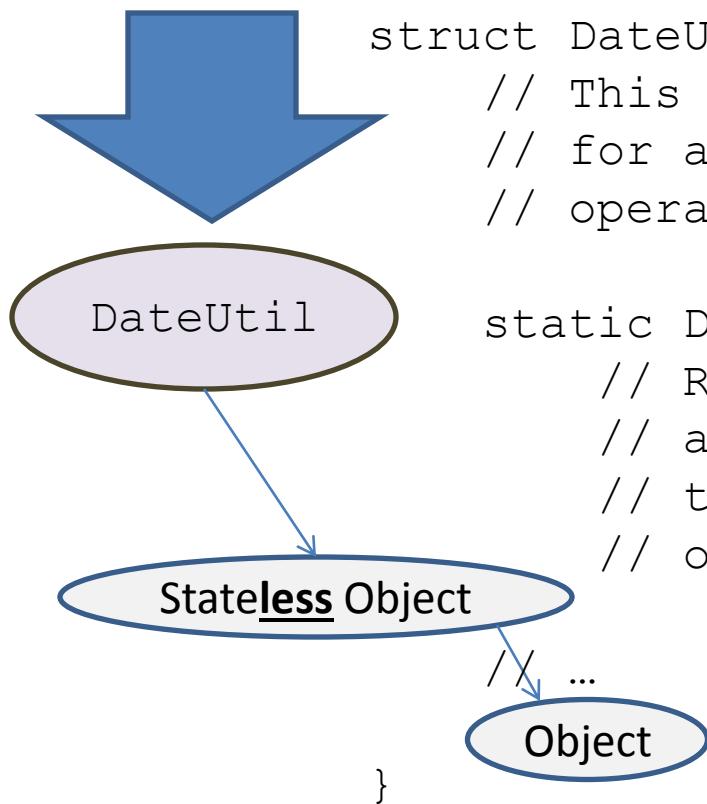
The first question: “Does it have state?”



2. Design & Implementation

Categorizing Object Types

The first question: “Does it have state?”



```
struct DateUtil {  
    // This struct  
    // for a suite  
    // operating on
```

```
static Date last  
    // Return the  
    // as the spe  
    // that the  
    // of 'value'
```

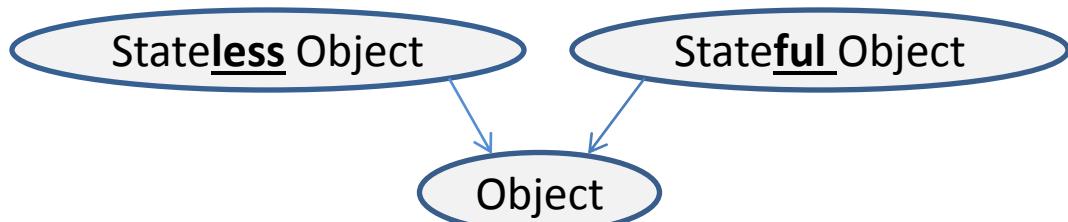
Utilities are an
important class
category!

e);
e

2. Design & Implementation

Categorizing Object Types

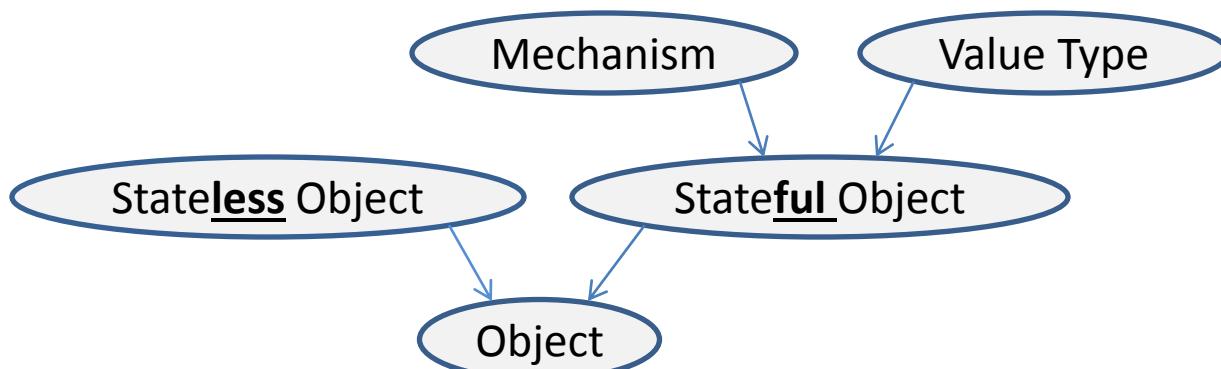
The second question: “Does it have value?”



2. Design & Implementation

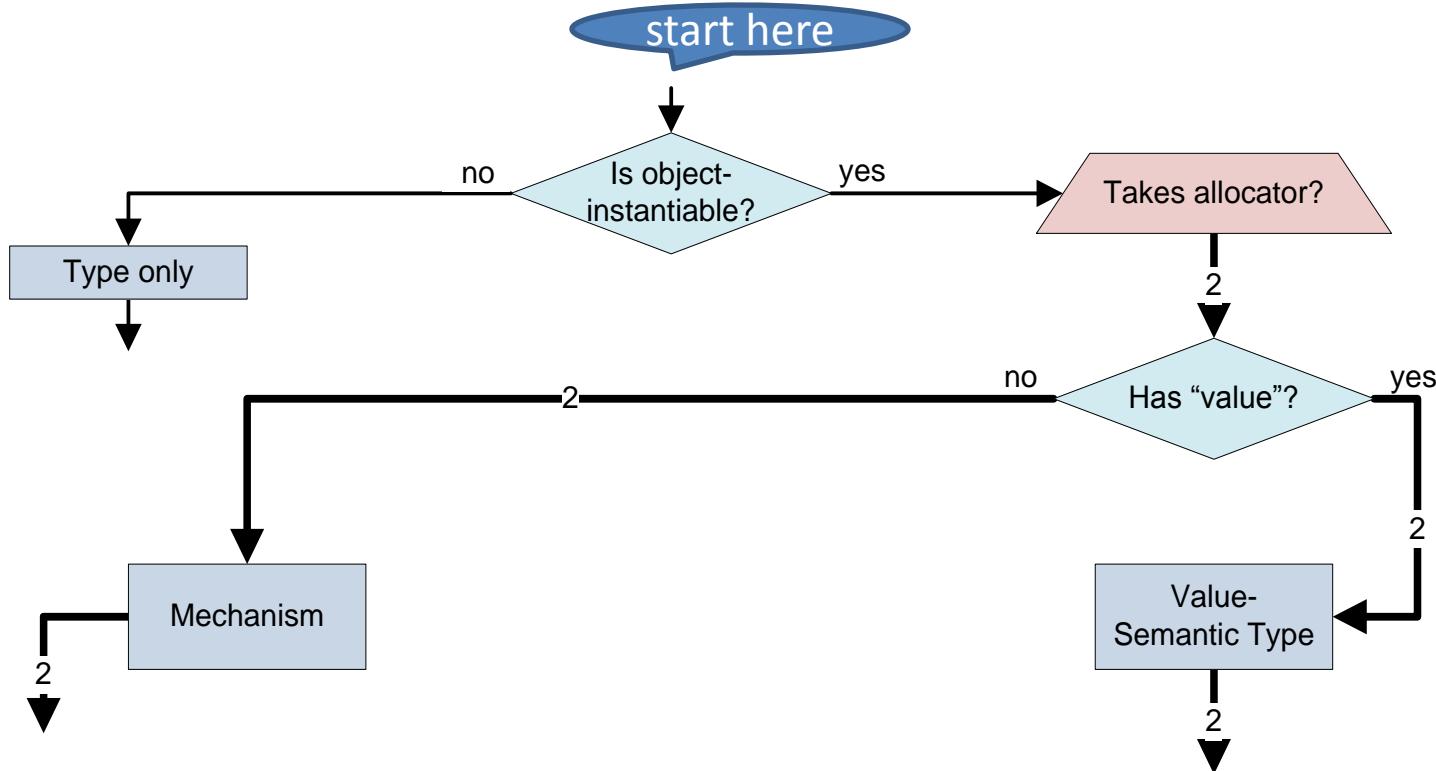
Categorizing Object Types

The second question: “Does it have value?”



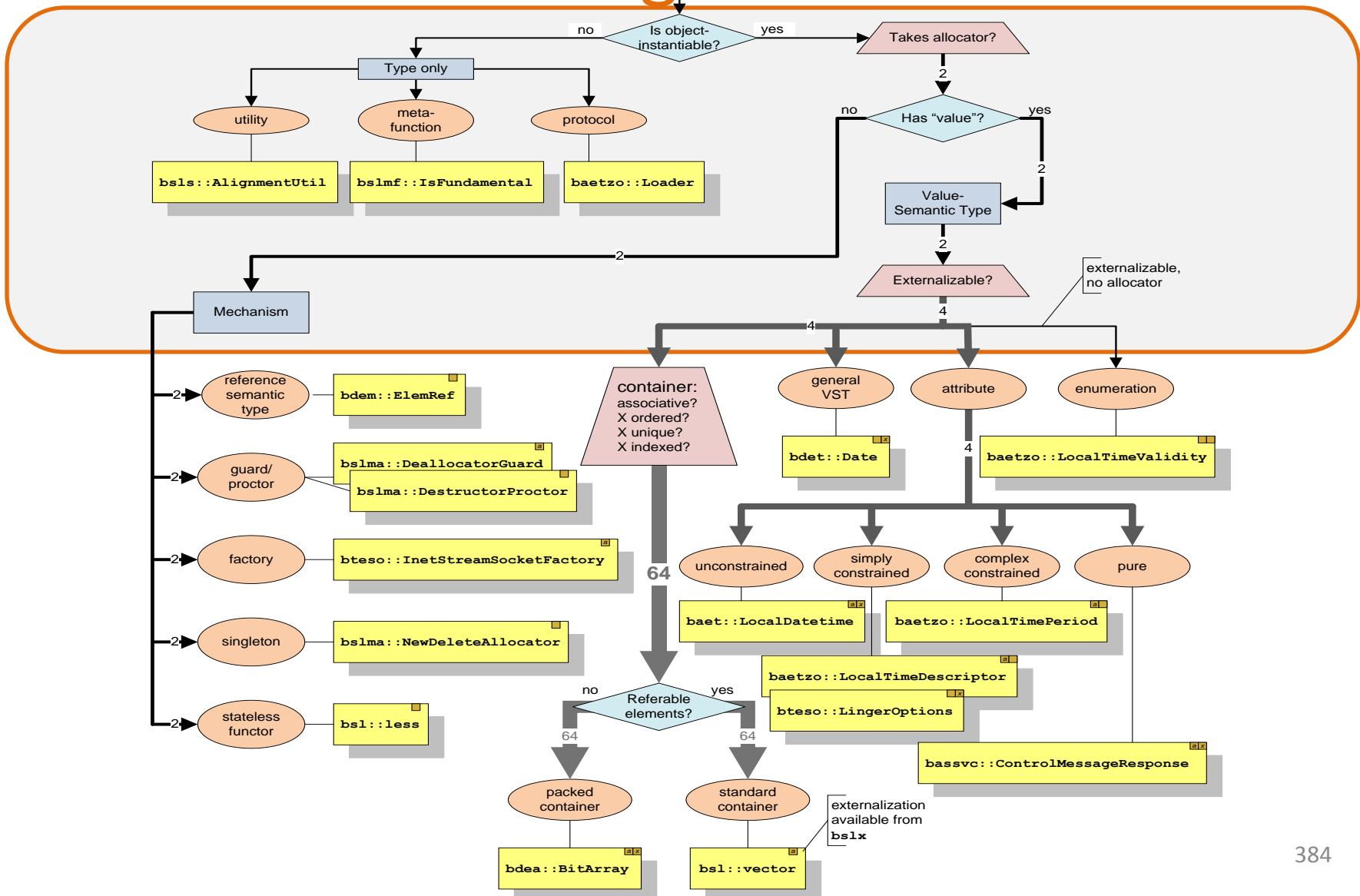
2. Design & Implementation

Top-Level Categorizations



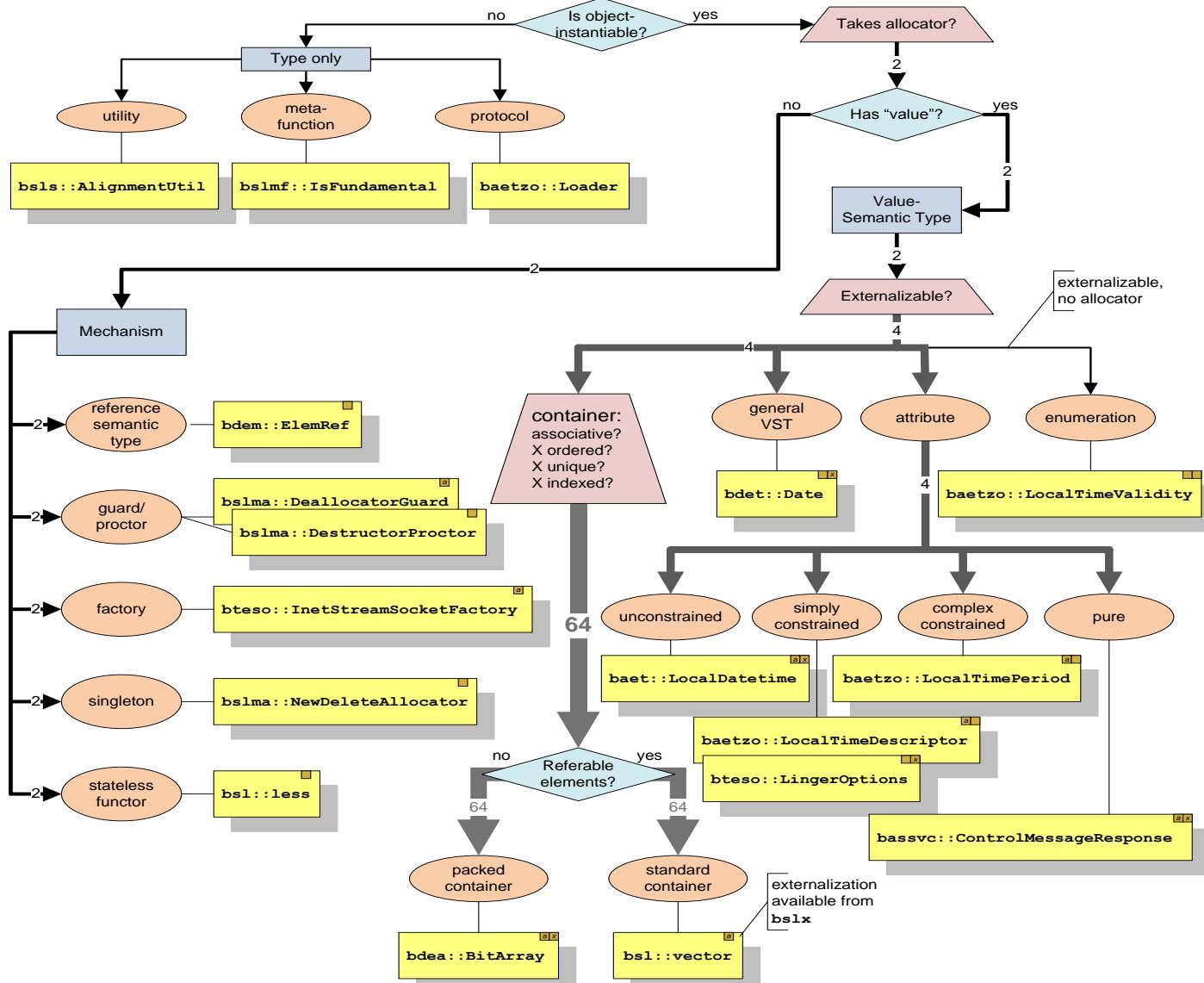
2. Design & Implementation

The Big Picture



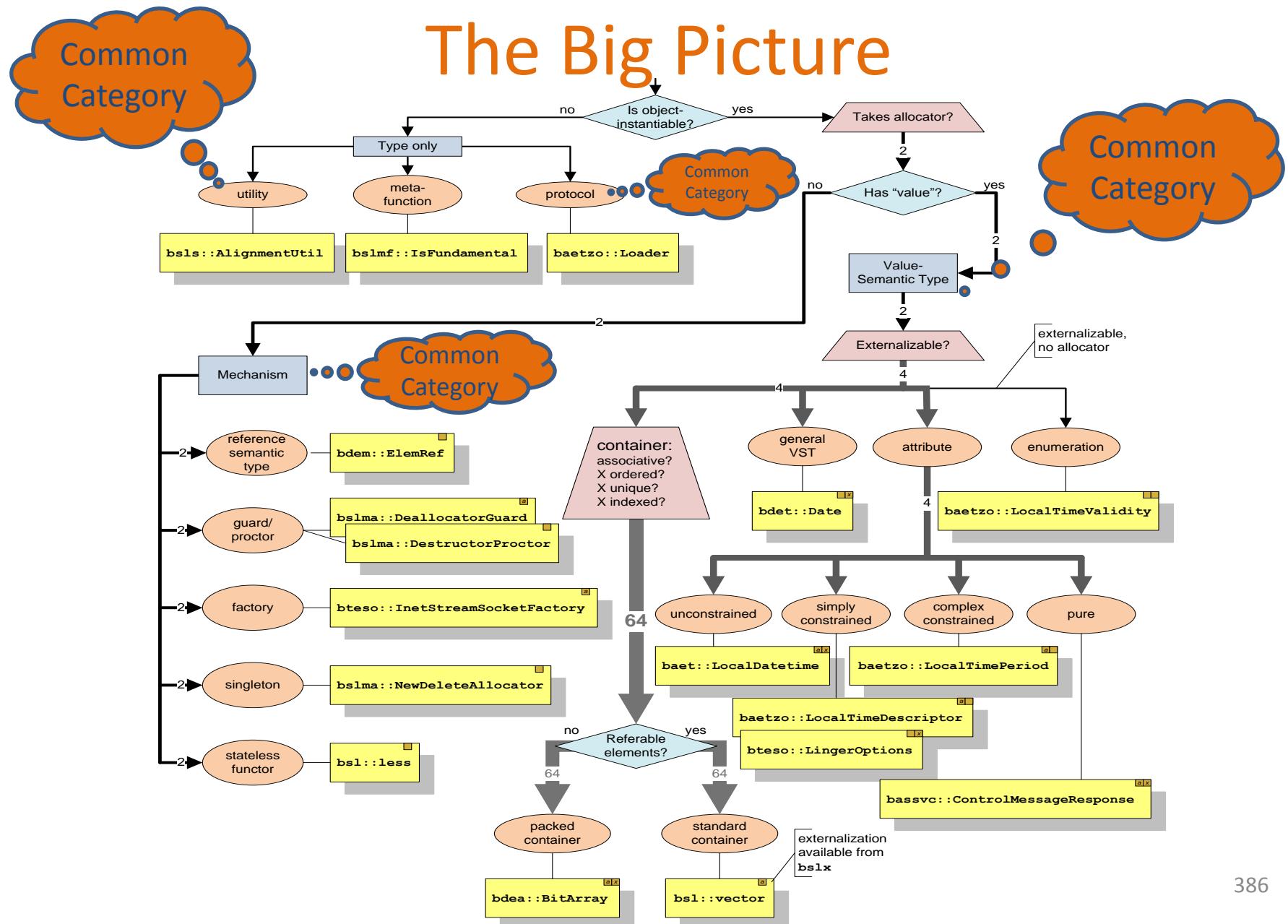
2. Design & Implementation

The Big Picture



2. Design & Implementation

The Big Picture



2. Design & Implementation

Essential Strategies and Techniques

Integral to our design process are:

- a) Common Class Categories
- b) Unique *Vocabulary* Types
- c) Design By Contract
- d) Appropriately *Narrow* Contracts
- e) An Overriding Customer Focus

2. Design & Implementation

Essential Strategies and Techniques

Integral to our design process are:

- a) Common Class Categories
- b) Unique *Vocabulary* Types
- c) Design By Contract
- d) Appropriately *Narrow* Contracts
- e) An Overriding Customer Focus

2. Design & Implementation

Vocabulary Types

A key feature of reuse is **interoperability**.

2. Design & Implementation

Vocabulary Types

A key feature of reuse is **interoperability**.

- We achieve interoperability by the ubiquitous use of:

Vocabulary Types

2. Design & Implementation

Vocabulary Types

(An Example)

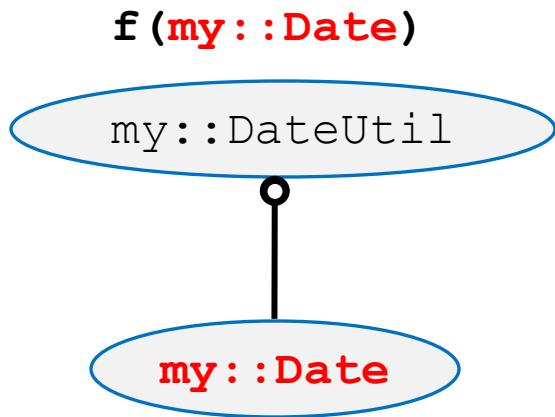


my::Date

2. Design & Implementation

Vocabulary Types

(An Example)



2. Design & Implementation

Vocabulary Types

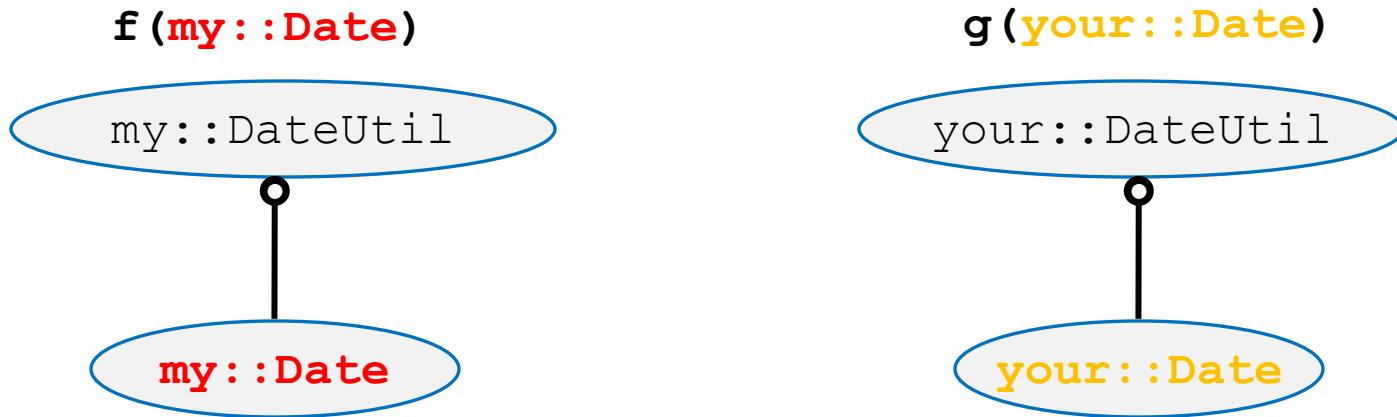
(An Example)



2. Design & Implementation

Vocabulary Types

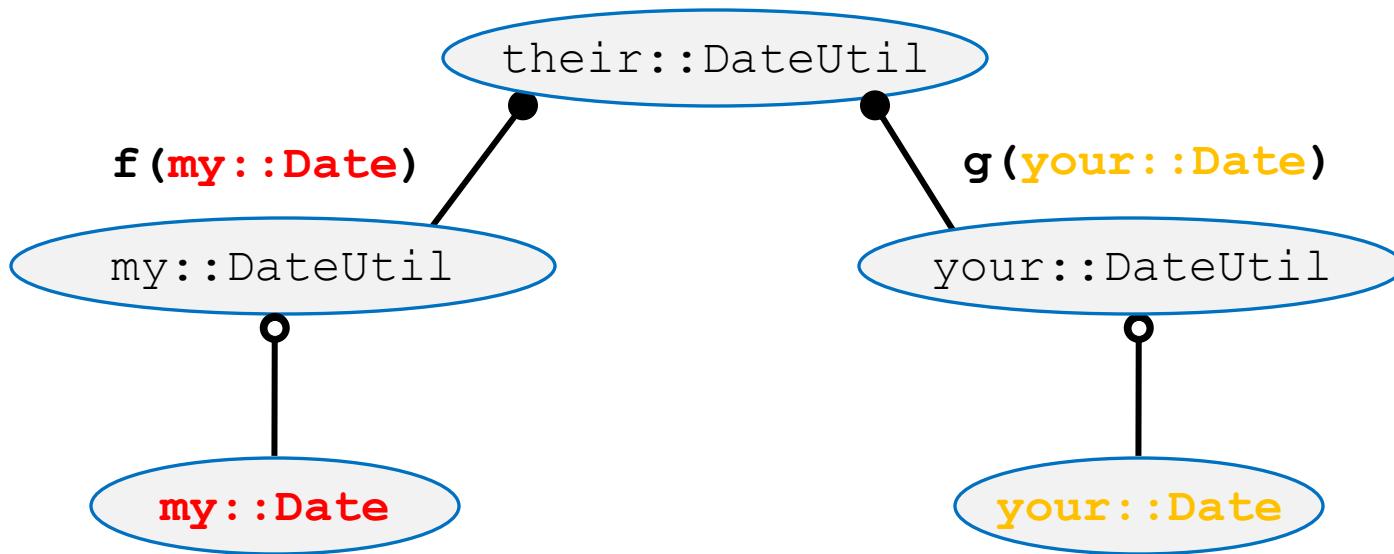
(An Example)



2. Design & Implementation

Vocabulary Types

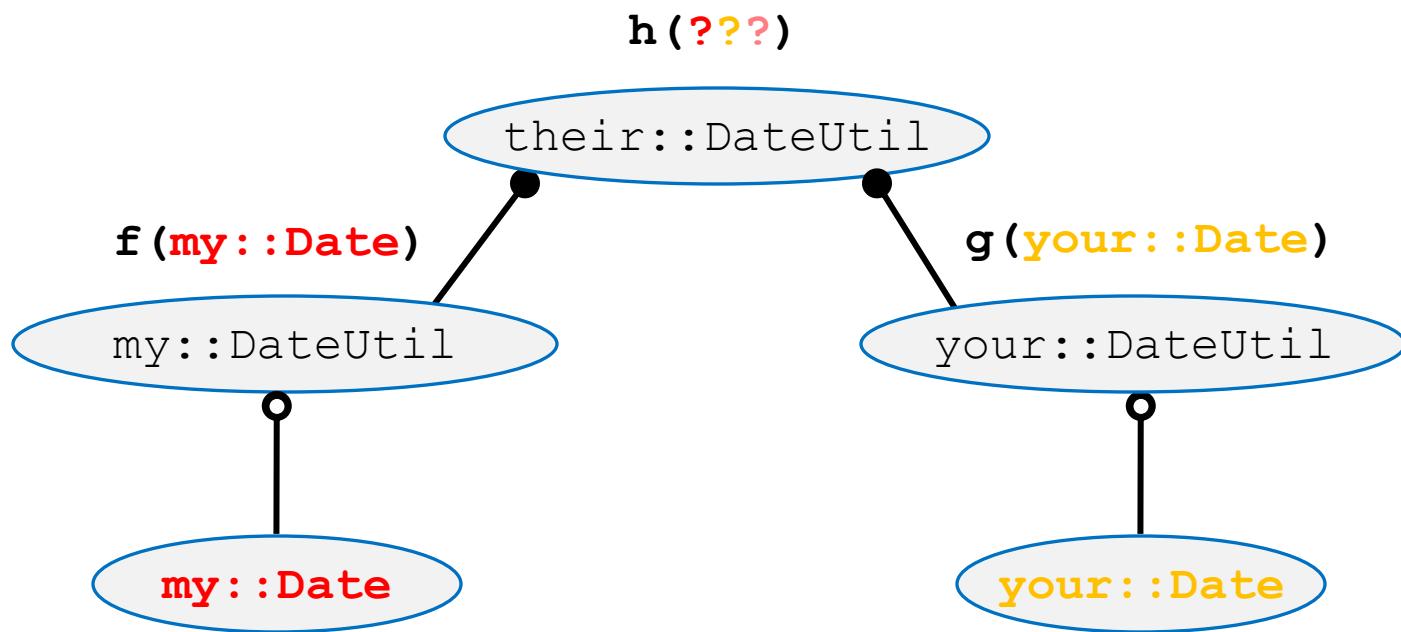
(An Example)



2. Design & Implementation

Vocabulary Types

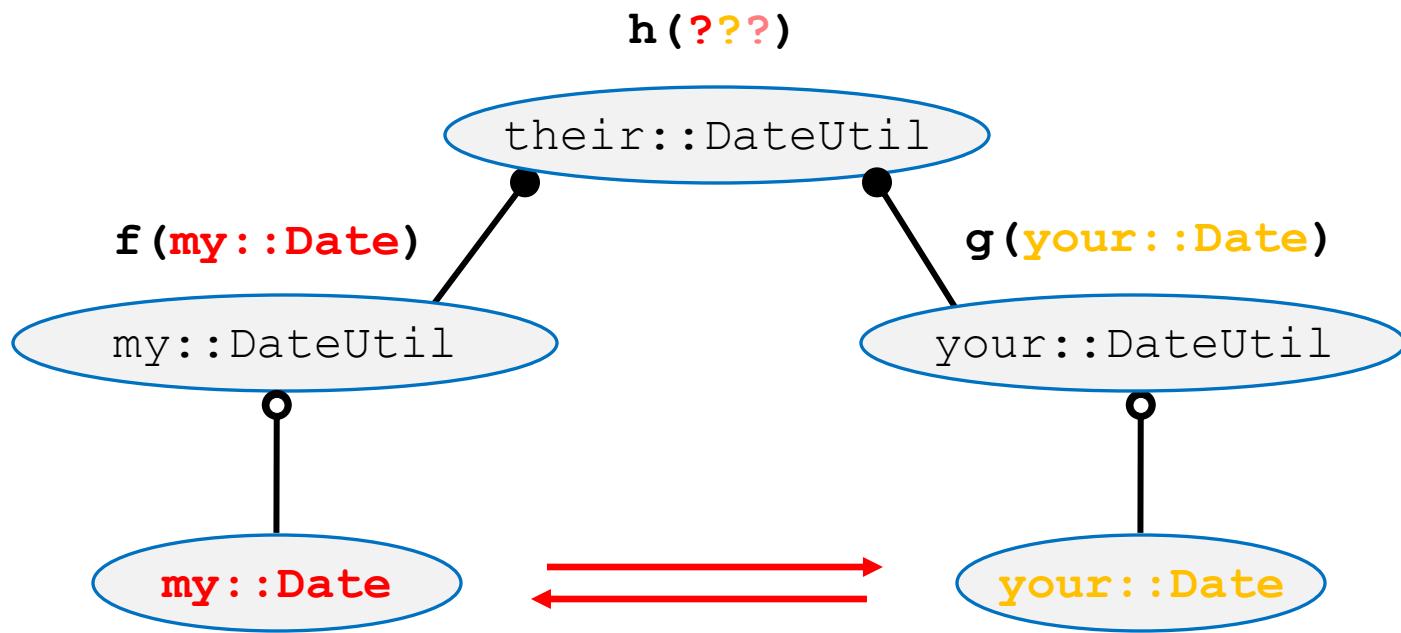
(An Example)



2. Design & Implementation

Vocabulary Types

(An Example)

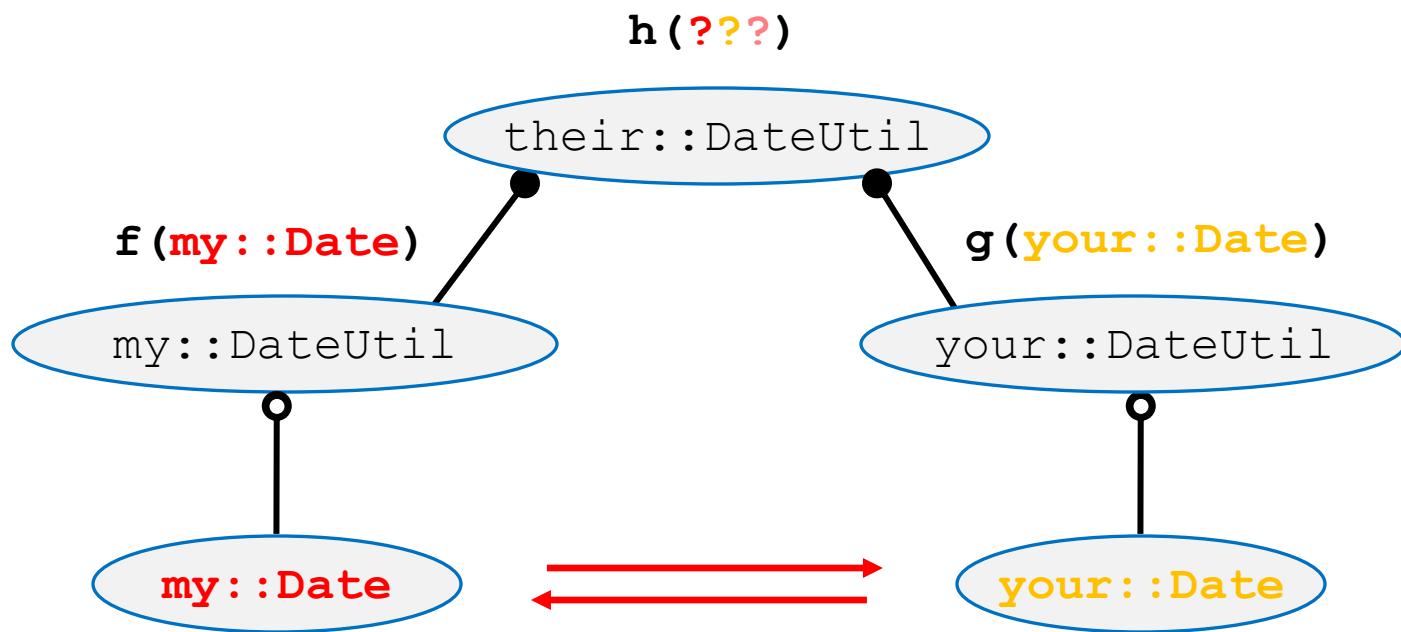


Interoperability Problem!

2. Design & Implementation

Vocabulary Types

(An Example)

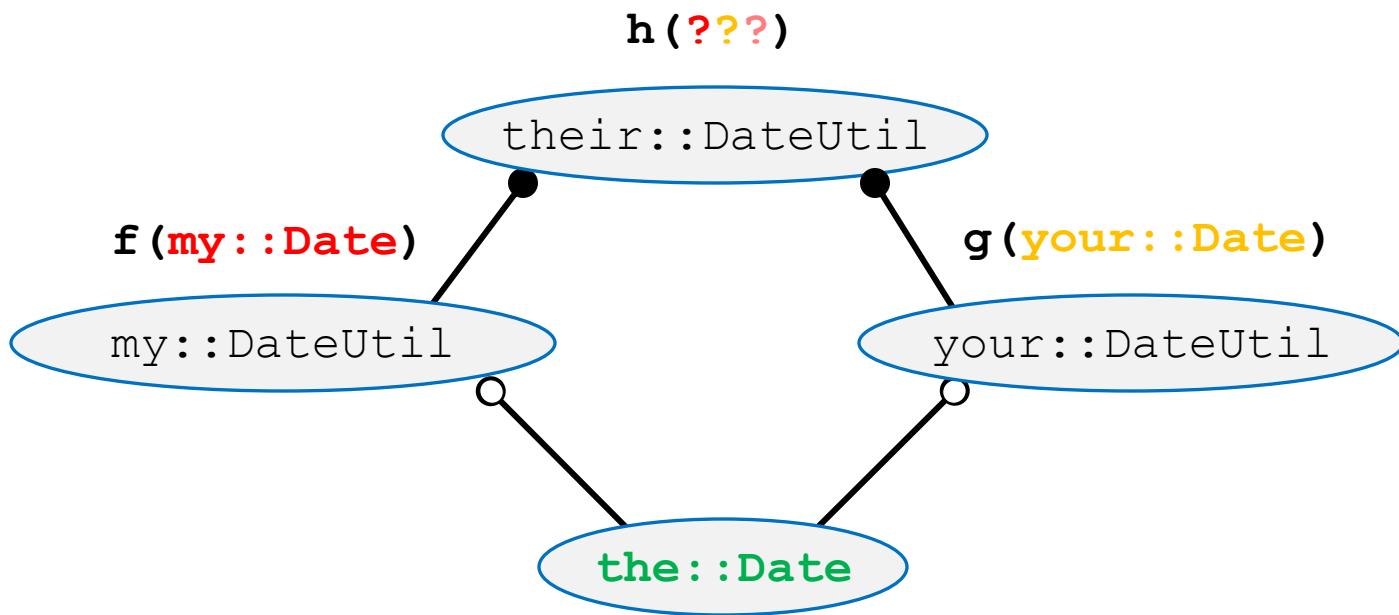


What should we do?

2. Design & Implementation

Vocabulary Types

(An Example)

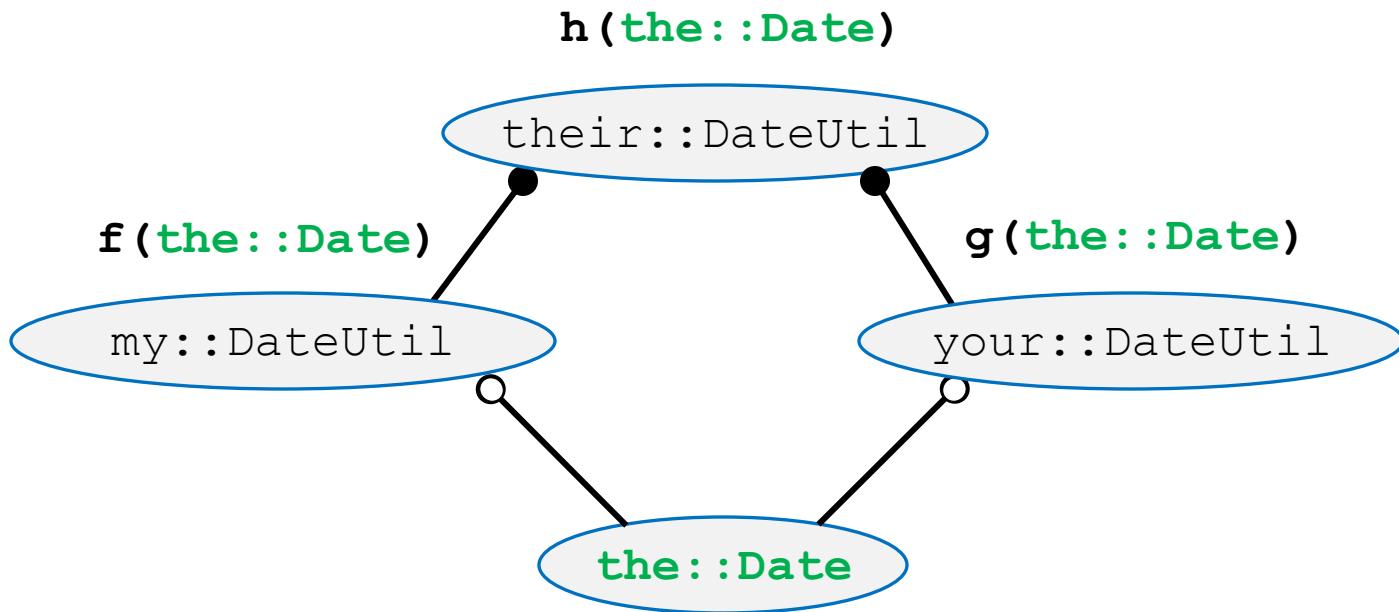


What should we do?

2. Design & Implementation

Vocabulary Types

(An Example)

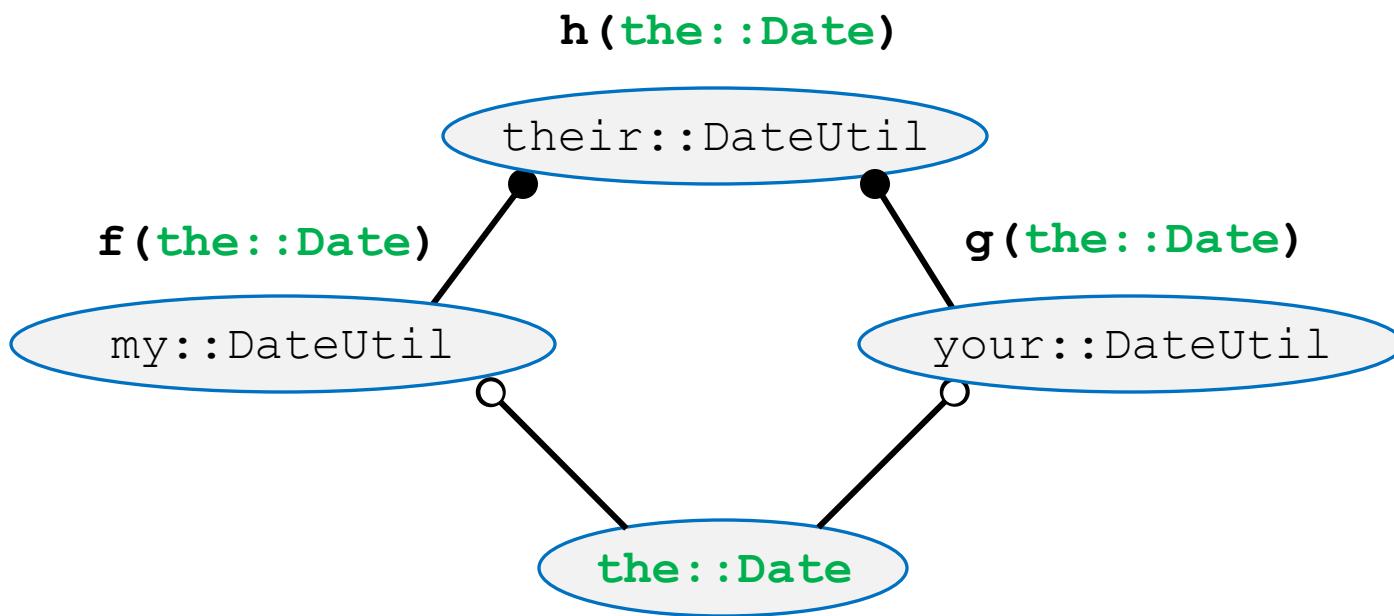


What should we do?

2. Design & Implementation

Vocabulary Types

(An Example)



No Interoperability Problem!

2. Design & Implementation

Vocabulary Types

On the other hand...

Distinct algebraic structures
deserve distinct C++ types.

2. Design & Implementation

Vocabulary Types

Consider `operator++` on an `int` versus a `Date`:

2. Design & Implementation

Vocabulary Types

Consider `operator++` on an `int` versus a `Date`:

```
int x(20080331);
```

2. Design & Implementation

Vocabulary Types

Consider `operator++` on an `int` versus a `Date`:

```
int x(20080331);  
Date y(2008, 03, 31);
```

2. Design & Implementation

Vocabulary Types

Consider `operator++` on an `int` versus a `Date`:

```
int x(20080331);
```

```
Date y(2008, 03, 31);
```

`++x`:

2. Design & Implementation

Vocabulary Types

Consider operator++ on an int versus a Date:

```
int x(20080331);
```

```
Date y(2008, 03, 31);
```

```
++x: 20080332
```

Basic operations for
type int lead to
invalid “date” values.

2. Design & Implementation

Vocabulary Types

Consider `operator++` on an `int` versus a `Date`:

```
int x(20080331);
```

```
Date y(2008, 03, 31);
```

`++x`: 20080332

`++y`:

2. Design & Implementation

Vocabulary Types

Consider operator++ on an int versus a Date:

```
int x(20080331);
```

```
Date y(2008, 03, 31);
```

```
++x: 20080332
```

```
++y: (2008, 04, 01)
```

Operations for
type Date
preserve
invariants.

2. Design & Implementation

Vocabulary Types

Consider `operator++` on an `int` versus a `Date`:

```
int x(20080331);
```

```
Date y(2008, 03, 31);
```

`++x`: 20080332

`++y`: (2008, 04, 01)

Hence, ***date*** values deserve their own C++ type!

2. Design & Implementation

Vocabulary Types

The “*type name*” and “*variable name*” of an object serve two distinct roles:

1. The *type name* defines the algebraic structure.
2. The *variable name* indicates intent/purpose in context.

```
int      age;
```

```
string  filename;
```

2. Design & Implementation

Vocabulary Types

The “*type name*” and “*variable name*” of an object serve two distinct roles:

1. The ***type name* defines the algebraic structure.**
2. The *variable name* indicates intent/purpose in context.

int age;

string filename;

2. Design & Implementation

Vocabulary Types

The “*type name*” and “*variable name*” of an object serve two distinct roles:

1. The *type name* defines the algebraic structure.
2. **The *variable name* indicates intent/purpose in context.**

```
int      age;
```

```
string  filename;
```

2. Design & Implementation

Vocabulary Types

An ***integer*** or ***string*** value used in a particular context should not be a separate type:

integer

- Age
- Shoe Size
- Account Number
- Year
- Day of Month
- Number of Significant Digits

string

- Text
- Word
- Username
- Filename
- Password
- Regular Expression

2. Design & Implementation

Vocabulary Types

An ***integer*** or ***string*** value used in a particular context should not be a separate type:

integer

- Age
- Shoe Size
- Account Number
- Year
- Day of Month
- Number of Significant Digits

string

- Text
- Word
- Username
- Filename
- Password
- Regular Expression

2. Design & Implementation

Vocabulary Types

An *integer* or *string* value used in a particular context should not be a separate type:

integer

- Age
- Shoe Size
- Account Number
- Year
- Day of Month
- Number of Significant Digits

string

- Text
- Word
- Username
- Filename
- Password
- Regular Expression

2. Design & Implementation

Template Policies

TEMPLATES CAN
PRESENT A
VOCABULARY
PROBLEM

2. Design & Implementation

Template Policies

Template parameters can be partitioned into three basic categories:

2. Design & Implementation

Template Policies

Template parameters can be partitioned into three basic categories:

- **Essential Parameters**
 - Parameters that must be specified in all cases.

2. Design & Implementation

Template Policies

Template parameters can be partitioned into three basic categories:

- Essential Parameters
 - Parameters that must be specified in all cases.
- Interface Policies
 - Optional parameters that do affect logical behavior.

2. Design & Implementation

Template Policies

Template parameters can be partitioned into three basic categories:

- Essential Parameters
 - Parameters that must be specified in all cases.
- Interface Policies
 - Optional parameters that do affect logical behavior.
- Implementation Policies
 - Optional parameters that do not affect logical behavior.

2. Design & Implementation

Template Policies

Essential Parameters

- Are necessary for basic operation.
- Typically do not have reasonable defaults.

2. Design & Implementation

Template Policies

Essential Parameters

- Are necessary for basic operation.
- Typically do not have reasonable defaults.

Example:

```
template <class T> class vector;
```

2. Design & Implementation

Template Policies

Essential Parameters

- Are necessary for basic operation.
- Typically do not have reasonable defaults.

Example:

```
template <class T> class vector;
```



Essential
Parameter

2. Design & Implementation

Template Policies

Essential Parameters

- Are necessary for basic operation.
- Typically do not have reasonable defaults.

Example:

```
template <class T> class vector;
```

```
template <class Iter>
```

```
void sort(Iter begin, Iter end);
```

2. Design & Implementation

Template Policies

Essential Parameters

- Are necessary for basic operation.
- Typically do not have reasonable defaults.

Example:

```
template <class T> class C { ... } or;  
template <class Iter>  
void sort(Iter begin, Iter end);
```

Essential
Parameter

2. Design & Implementation

Template Policies

Interface Policies

- Affect intended “logical” behavior.
- Typically do have reasonable defaults.

2. Design & Implementation

Template Policies

Interface Policies

- Affect intended “logical” behavior.
- Typically do have reasonable defaults.

Example:

```
template <class T, class C = less<T>>
class OrderedSet;
```

2. Design & Implementation

Template Policies

Interface Policies

- Affect intended “logical” behavior.
- Typically do not have sensible defaults.

Example:



Essential
Parameter

```
template <class T, class C = less<T>>
class OrderedSet;
```

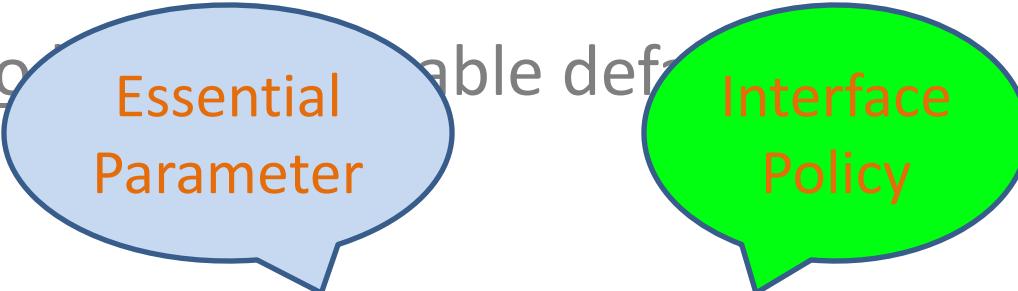
2. Design & Implementation

Template Policies

Interface Policies

- Affect intended “logical” behavior.
- Typically do ~~not~~ have to be defined in the template definition.

Example:



```
template <class T, class C = less<T>>
class OrderedSet;
```

2. Design & Implementation

Template Policies

Implementation Policies

- DO NOT affect intended “logical” behavior.
- Typically do have reasonable defaults.

2. Design & Implementation

Template Policies

Implementation Policies

- DO NOT affect intended “logical” behavior.
- Typically do have reasonable defaults.

Example:

```
template <class T,  
          class C = hash<T>,  
          int LOAD_FACTOR = 1>  
class UnorderedSet;
```

2. Design & Implementation

Template Policies

Implementation Policies

- DO NOT affect intended “logical” behavior.
- Try to provide reasonable defaults.

Example:

```
template <class T,  
          class C = hash<T>,  
          int LOAD_FACTOR = 1>  
class UnorderedSet;
```

Essential
Parameter

2. Design & Implementation

Template Policies

Implementation Policies

- DO NOT affect intended “logical” behavior.
- Try to make them generic.

Example:

```
template <class T>
    class C = hash<T>,
    int LOAD_FACTOR = 1>

class UnorderedSet;
```

Essential
Parameter

Implementation
Policy

2. Design & Implementation

Template Policies

Implementation Policies

- DO NOT affect intended “logical” behavior.
- Try to make them generic.

Example:

```
template <class T>
{
    class C = hash<T>
    int LOAD_FACTOR = 1>

    class UnorderedSet;
```

Essential
Parameter

Implementation
Policy

Implementation
Policy

2. Design & Implementation

Template Policies

Implementation Policies

- DO NOT affect intended “logical” behavior.
- Typically do have reasonable defaults.

Example:

```
template <class T,  
          class L = DefaultLock>  
class Queue;
```

2. Design & Implementation

Template Policies

Implementation Policies

- DO NOT affect intended “logical” behavior.
- Try to provide reasonable defaults.

Example:

```
template <class T,  
          class L = DefaultLock>  
class Queue;
```

Essential
Parameter

2. Design & Implementation

Template Policies

Implementation Policies

- DO NOT affect intended “logical” behavior.
- Try to provide reasonable defaults.

Example:

```
template <class T,  
          class L = DefaultLock>  
  
class Queue;
```

Essential
Parameter

Implementation
Policy

2. Design & Implementation

Template Policies

Problem!

**Template Parameters
Affect Object Type.**

2. Design & Implementation

Template Policies

Essential Parameters:

```
vector<int> a;
```

```
vector<int> b;
```

```
a = b;
```

2. Design & Implementation

Template Policies

Essential Parameters:

```
vector<int> a;
```

```
vector<double> b;
```

```
a = b;
```

Compiler
Error

FINE!

2. Design & Implementation

Template Policies

Interface Policies:

```
OrderedSet<int> a;
```

```
OrderedSet<int> b;
```

```
if (a == b) {
```

```
    // ...
```

2. Design & Implementation

Template Policies

Interface Policies:

```
OrderedSet<int> a;
```

```
OrderedSet<int, MyLess> b;
```

```
if (a == b) {
```

```
// ...
```

Compiler
Error



2. Design & Implementation

Template Policies

Implementation Policies:

```
void f (Queue<double> *queue) ;  
  
void g ()  
{  
    Queue<double> q;  
    f (&q) ;  
}
```

2. Design & Implementation

Template Policies

Implementation Policies:

```
void f (Queue<double> *queue) ;  
  
void g ()  
{  
    Queue<double, MyLock> q;  
    f (&q);  
}
```

Compiler
Error



2. Design & Implementation

Template Policies

Implementation Policy

```
template<class T>
void f(T *queue);
void g()
{
    Queue<double, MyLock> q;
    f(&q);
}
```

Compiles
Fine

*The Entire
Implementation
Must Now
Reside In the
Header File*

**NOT
GOOD!**

2. Design & Implementation

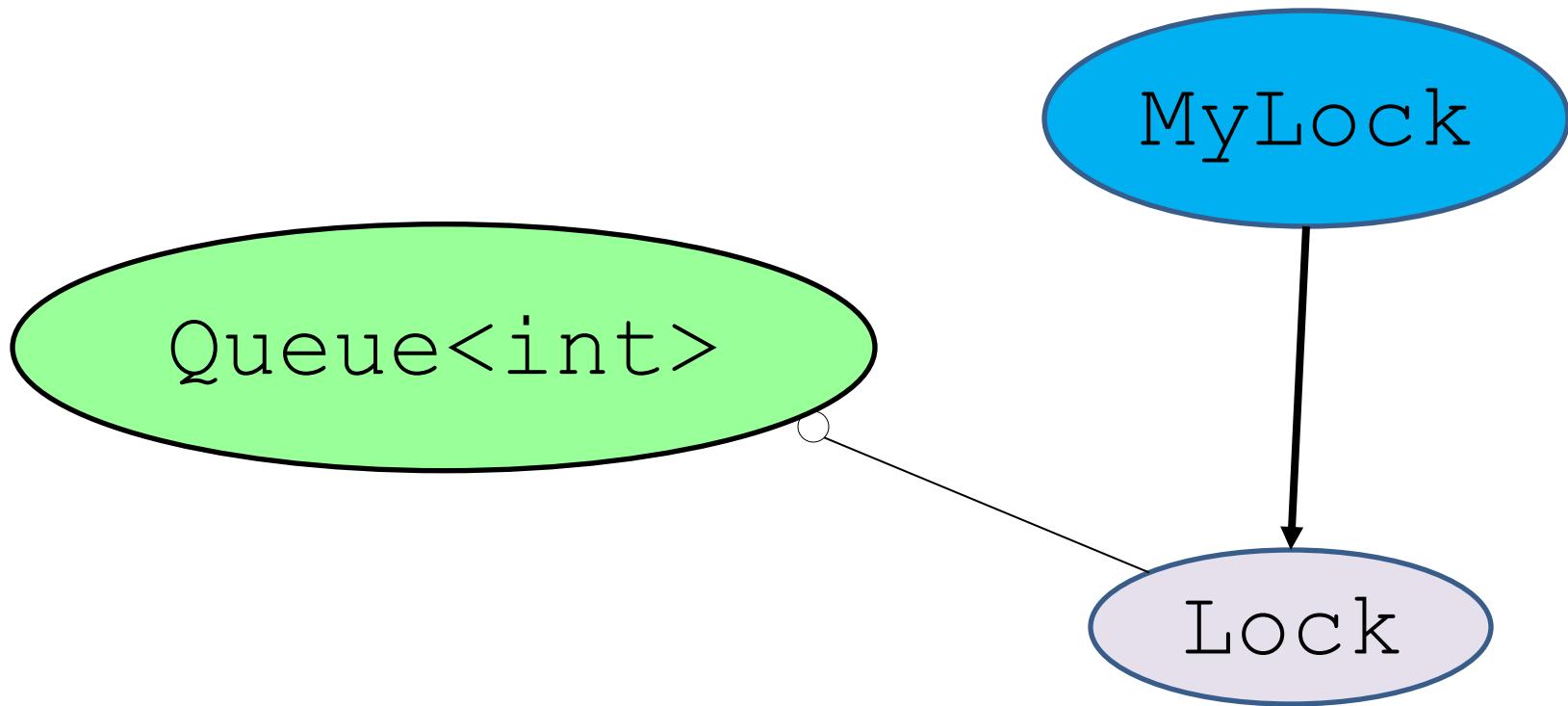
Template Policies

Solution!

**Runtime
Implementation
Policies**

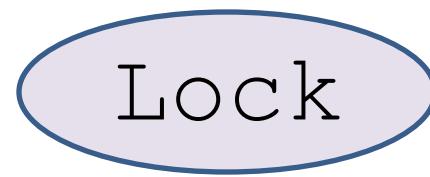
2. Design & Implementation

Runtime Implementation Policies



2. Design & Implementation

Runtime Implementation Policies



2. Design & Implementation

Runtime Implementation Policies

```
class Lock {  
    // Pure abstract (protocol) class.  
public:  
    virtual ~Lock() ;  
  
    virtual void lock() = 0;  
    virtual void unlock() = 0;  
};
```

2. Design & Implementation

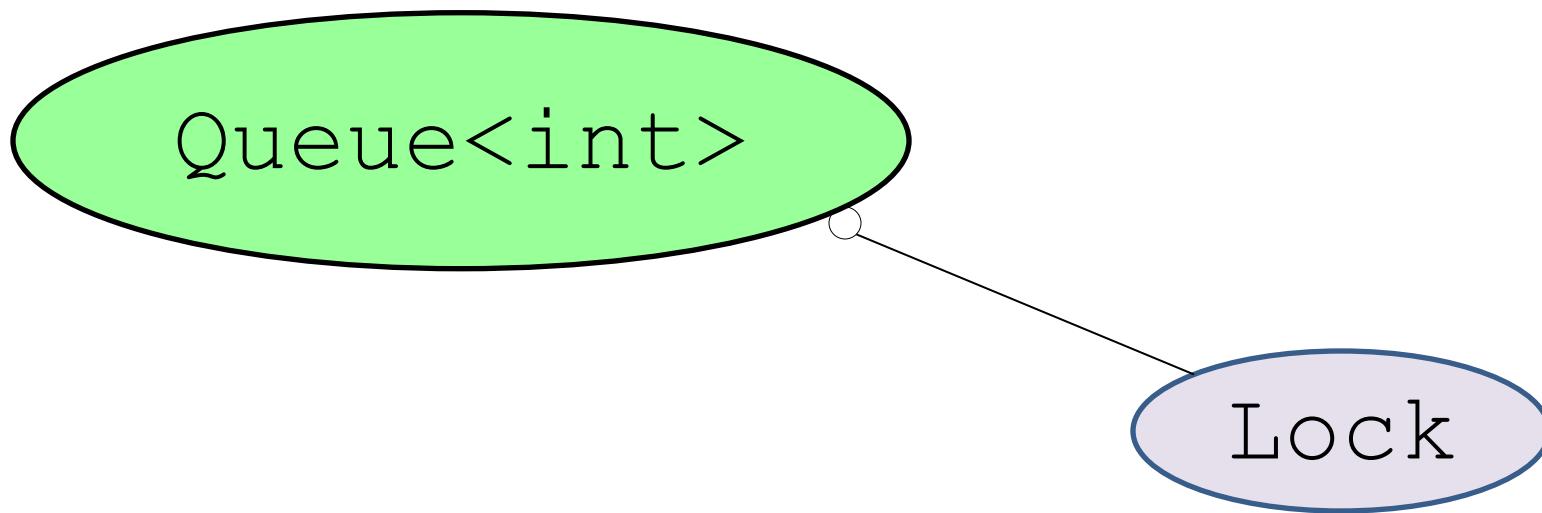
Runtime Implementation Policies

```
class Lock {  
    // Pure abstract (protocol) class.  
public:  
    virtual ~Lock();  
    virtual void lock();  
    virtual void unlock();  
};
```

Common
Class
Category

2. Design & Implementation

Runtime Implementation Policies



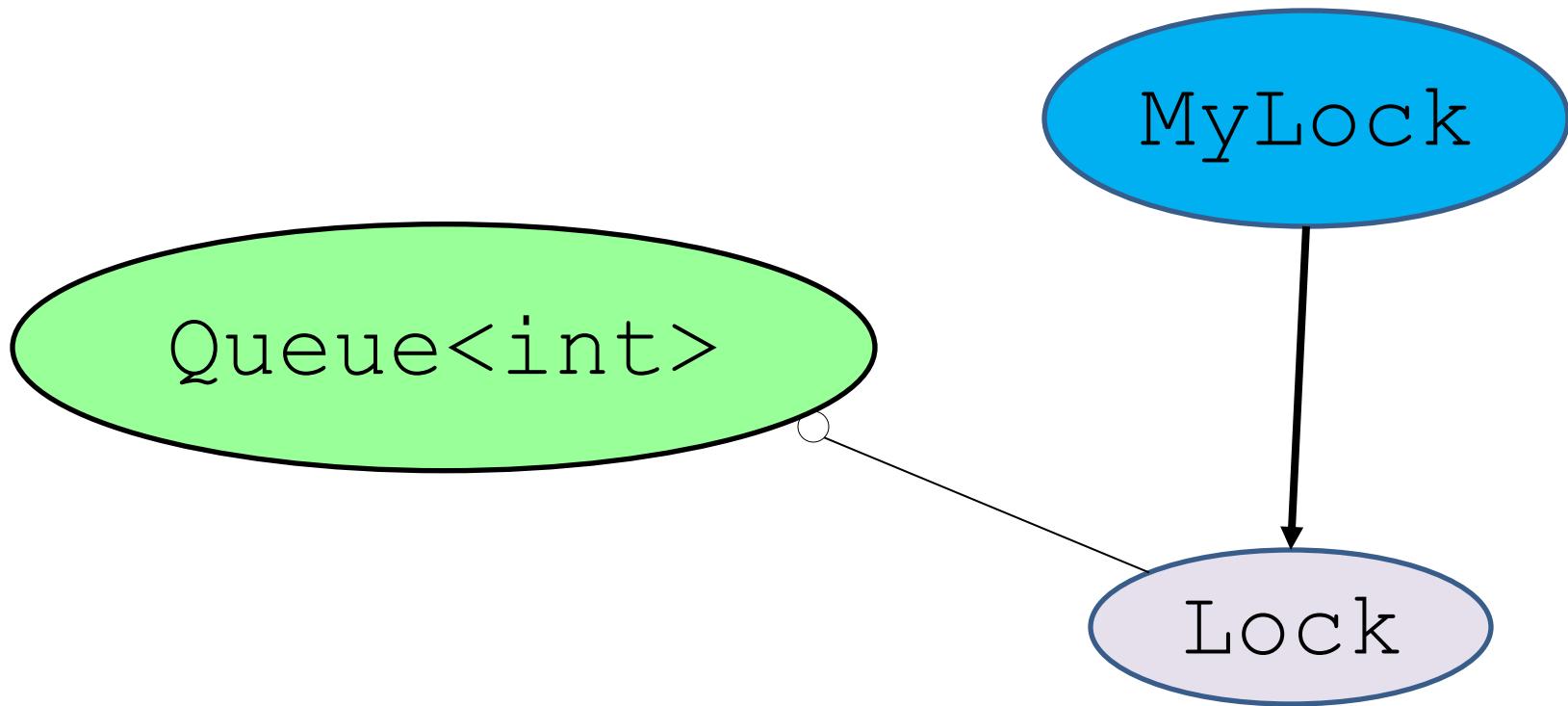
2. Design & Implementation

Runtime Implementation Policies

```
template<class T> class Queue {  
    // ... Concrete value-semantic container type.  
    Lock *d_lock_p;  
  
public:  
    Queue(Lock *lock = 0);  
    Queue(const Queue<T>& other, Lock *lock = 0);  
    // ...  
    void pushBack(const T& value);  
    // ...  
};
```

2. Design & Implementation

Runtime Implementation Policies



2. Design & Implementation

Runtime Implementation Policies

```
class MyLock : public Lock {  
    // ... Concrete mechanism.  
  
private:  
    MyLock(const MyLock&);  
    MyLock& operator=(const MyLock&);  
  
public:  
    MyLock();  
    virtual ~MyLock();  
    virtual void lock();  
    virtual void unlock();  
};
```

2. Design & Implementation

Runtime Implementation Policies

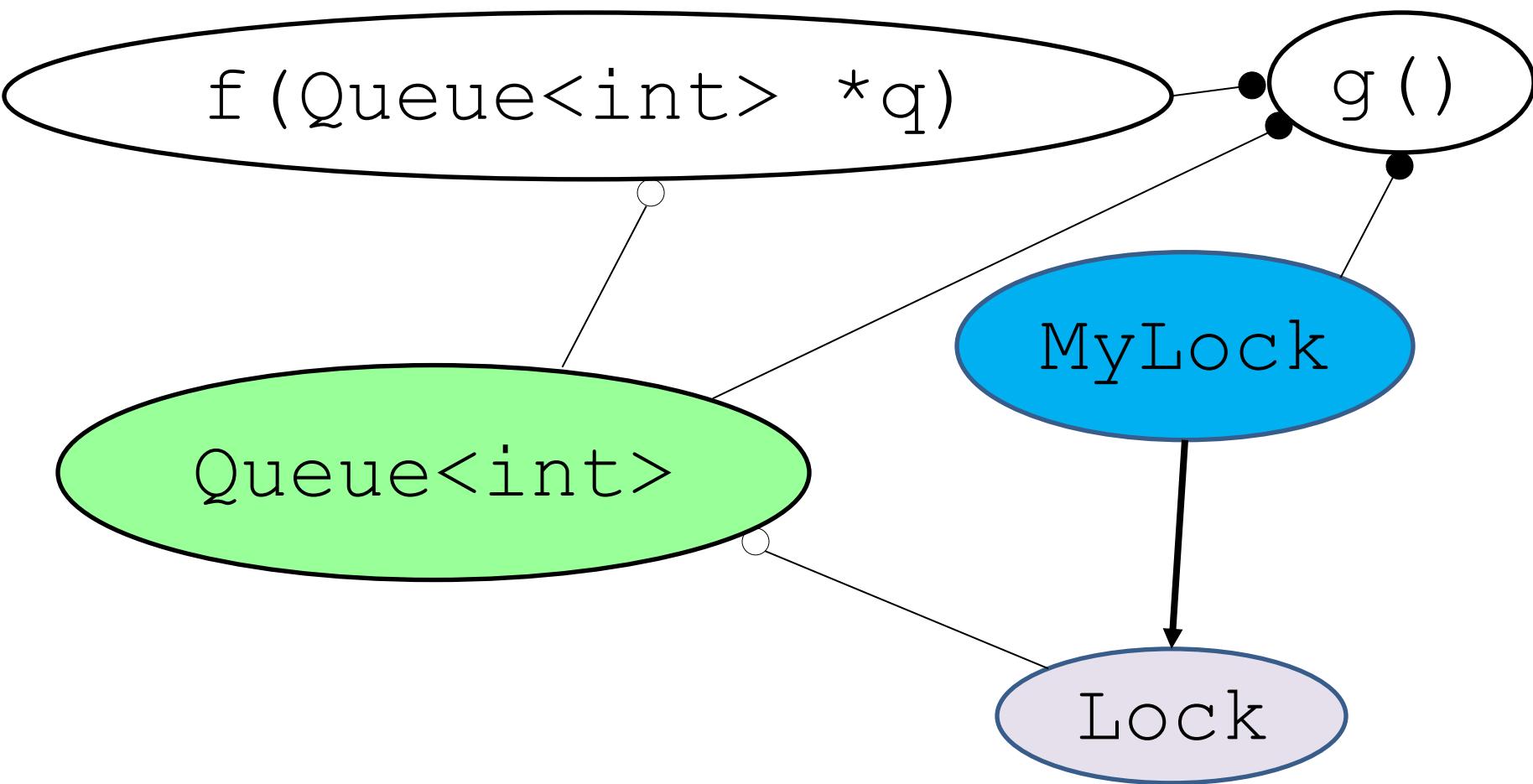
```
class MyLock : public Lock {  
    // ... Concrete mechanism.  
  
    MyLock(const MyLock&) = delete;  
    MyLock& operator=(const MyLock&) = delete;  
  
public:  
    MyLock();  
    virtual ~MyLock();  
    virtual void lock();  
    virtual void unlock();  
};
```

Or, in
C++11

= delete;

2. Design & Implementation

Runtime Implementation Policies



2. Design & Implementation

Runtime Implementation Policies

```
void f (Queue<double> *q) ;  
  
void g ()  
{  
    MyLock lock;  
    Queue<double> queue (&lock) ;  
    f (&queue) ;  
}
```

2. Design & Implementation

Runtime Implementation Policies

```
void f (Queue<double> *q) ;  
  
void g ()  
{  
    MyLock lock;  
    Queue<double> queue (&lock) ;  
    f (&queue) ;  
}
```

2. Design & Implementation

Runtime Implementation Policies

```
void f (Queue<double> *q) ;  
  
void g ()  
{  
    MyLock lock;  
    Queue<double> queue (&lock) ;  
    f (&queue) ;  
}
```

2. Design & Implementation

Runtime Implementation Policies

```
void f (Queue<double> *q) ;  
  
void g ()  
{  
    MyLock lock;  
    Queue<double> queue (&lock) ;  
    f (&queue) ;  
}
```

2. Design & Implementation

Runtime Implementation Policies

```
void f (Queue<double> *q) ;  
  
void g ()  
{  
    MyLock lock;  
    Queue<double> queue (&lock) ;  
    f (&queue) ;  
}
```

2. Design & Implementation

Runtime Implementation Policies

```
void f (Queue<double> queue)
{
    void g ()
    {
        MyLock lock;
        Queue<double> queue (&lock);
        f (&queue);
    }
}
```

Question:
What is the *lifetime* of the **lock** relative to the **queue**?

2. Design & Implementation

Memory Allocators

What is a memory allocator?

2. Design & Implementation

Memory Allocators

What is a memory allocator?

- It is a *mechanism* used to supply memory.

2. Design & Implementation

Memory Allocators

What is a memory allocator?

- It is a *mechanism* used to supply memory.
- It does not have *value semantics*.

2. Design & Implementation

Memory Allocators

What is a memory allocator?

- It is a *mechanism* used to supply memory.
- It does not have *value semantics*.
- It is an *Orthogonal Implementation Policy*.

2. Design & Implementation

Memory Allocators

What is a memory allocator?

- It is a *mechanism* used to supply memory.
- It does not have *value semantics*.
- It is an *Orthogonal Implementation Policy*.
- It *can* (should) be a *Runtime Policy*.

2. Design & Implementation

Memory Allocators

What is a memory allocator?

- It is a *mechanism* used to supply memory.
- It does not have *value semantics*.
- It is an *Orthogonal Implementation Policy*.
- It *can* (should) be a *Runtime Policy*.

2. Design & Implementation

Polymorphic Memory Allocators

What is a memory allocator?

- It is a *mechanism* used to supply memory.
- It does not have *value semantics*.
- It is an *Orthogonal Implementation Policy*.
- It *can* (should) be a *Runtime Policy*.

2. Design & Implementation

Polymorphic Memory Allocators

What is a memory allocator?

It should look like a
“Lock” or any other
abstract mechanism.

2. Design & Implementation

Polymorphic Memory Allocators

What is a memory allocator?

It should look like a
vocabulary like a
“Lock” or any other
Type abstract mechanism.

2. Design & Implementation

Polymorphic Memory Allocators

An allocator is a *mechanism*.

```
double f(double *a, size_t n)
{
    double result = init(a, n);
    bdlma::BufferedSequentialAllocator a;
    bsl::vector<double> tmp(&a);

    // ...

    return result;
}
```

2. Design & Implementation

Polymorphic Memory Allocators

An allocator is a *mechanism*.

```
double f(double *a, size_t n)
{
    double result = init(a, n);
    bdlma::BufferedSequentialAllocator a;
    bsl::vector<double> tmp(&a);

    // ...
    return result;
}
```

2. Design & Implementation

Polymorphic Memory Allocators

An allocator is a *mechanism*.

```
double f(double *a, size_t n)
{
    double result = init(a, n);
    bdlma::BufferedSequentialAllocator a;
    bsl::vector<double> tmp(&a);
    // ...
    return result;
}
```

See the
bslma_allocator
component.

2. Design & Implementation

Essential Strategies and Techniques

Integral to our design process are:

- a) Common Class Categories
- b) Unique *Vocabulary* Types
- c) Design By Contract
- d) Appropriately *Narrow* Contracts
- e) An Overriding Customer Focus

2. Design & Implementation

Essential Strategies and Techniques

Integral to our design process are:

- a) Common Class Categories
- b) Unique *Vocabulary* Types
- c) **Design By Contract**
- d) Appropriately *Narrow Contracts*
- e) An Overriding Customer Focus

2. Design & Implementation

Interfaces and Contracts

What do we mean by *Interface* versus *Contract* for

- A *Function*?
- A *Class*?
- A *Component*?

2. Design & Implementation

Interfaces and Contracts

Function

```
std::ostream& print(std::ostream& stream,  
                     int           level          = 0,  
                     int           spacesPerLevel = 4) const;
```

2. Design & Implementation

Interfaces and Contracts

Function

```
std::ostream& print(std::ostream& stream,  
                     int          level      = 0,  
                     int          spacesPerLevel = 4) const;
```

Types Used
In the Interface

2. Design & Implementation

Interfaces and Contracts

Function

```
std::ostream& print(std::ostream& stream,
                     int           level          = 0,
                     int           spacesPerLevel = 4) const;
// Format this object to the specified output 'stream' at the (absolute
// value of) the optionally specified indentation 'level', and return a
// reference to 'stream'. If 'level' is specified, optionally specify
// 'spacesPerLevel', the number of spaces per indentation level for
// this and all of its nested objects. If 'level' is negative,
// suppress indentation of the first line. If 'spacesPerLevel' is
// negative, format the entire output on one line, suppressing all but
// the initial indentation (as governed by 'level'). If 'stream' is
// not valid on entry, this operation has no effect.
```

2. Design & Implementation

Interfaces and Contracts

Class

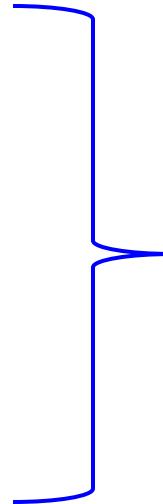
```
class Date {  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
  
    Date(const Date& original);  
  
    // ...  
};
```

2. Design & Implementation

Interfaces and Contracts

Class

```
class Date {  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
  
    Date(const Date& original);  
  
    // ...  
};
```



Public
Interface

2. Design & Implementation

Interfaces and Contracts

Class

```
class Date {  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
  
    Date(const Date& original);  
  
    // ...  
};
```

2. Design & Implementation

Interfaces and Contracts

Class

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.
```

```
//...
```

```
public:  
    Date(int year, int month, int day);
```

```
    Date(const Date& original);
```

```
// ...
```

```
};
```

2. Design & Implementation

Interfaces and Contracts

Class

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
  
    // ...  
};
```

2. Design & Implementation

Interfaces and Contracts

```
class Date {  
    // ...  
public:  
    // ...  
};
```

Component

```
bool operator==(const Date& lhs, const Date& rhs);
```

```
bool operator!=(const Date& lhs, const Date& rhs);
```

```
std::ostream& operator<<(std::ostream& stream, const Date& date);
```

2. Design & Implementation

Interfaces and Contracts

```
class Date {  
    // ...  
public:  
    // ...  
};
```

Component

- bool operator==(const Date& lhs, const Date& rhs);
 - bool operator!=(const Date& lhs, const Date& rhs);
 - std::ostream& operator<<(std::ostream& stream, const Date& date);
-
- “Public” Interface

2. Design & Implementation

Interfaces and Contracts

```
class Date {  
    // ...  
public:  
    // ...  
};
```

Component

```
bool operator==(const Date& lhs, const Date& rhs);
```

```
bool operator!=(const Date& lhs, const Date& rhs);
```

```
std::ostream& operator<<(std::ostream& stream, const Date& date);
```

2. Design & Implementation

Interfaces and Contracts

```
class Date {  
    // ...  
public:  
    // ...  
};
```

Component

```
bool operator==(const Date& lhs, const Date& rhs);  
// Return 'true' if the specified 'lhs' and 'rhs' dates have the same  
// value, and 'false' otherwise. Two 'Date' objects have the same  
// value if the corresponding values of their 'year', 'month', and 'day'  
// attributes are the same.  
  
bool operator!=(const Date& lhs, const Date& rhs);  
// Return 'true' if the specified 'lhs' and 'rhs' dates do not have the  
// same value, and 'false' otherwise. Two 'Date' objects do not have  
// the same value if any of the corresponding values of their 'year',  
// 'month', or 'day' attributes are not the same.  
  
std::ostream& operator<<(std::ostream& stream, const Date& date);  
// Format the value of the specified 'date' object to the specified  
// output 'stream' as 'yyyy/mm/dd', and return a reference to 'stream'.  
490
```

2. Design & Implementation

Preconditions and Postconditions

2. Design & Implementation

Preconditions and Postconditions

Function

2. Design & Implementation

Preconditions and Postconditions

Function

```
double sqrt(double value);
```

```
// Return the square root of the specified  
// 'value'. The behavior is undefined unless  
// '0 <= value'.
```

2. Design & Implementation

Preconditions and Postconditions

Function

```
double sqrt(double value);
```

```
// Return the square root of the specified  
// 'value'. The behavior is undefined unless  
// '0 <= value'.
```

2. Design & Implementation

Preconditions and Postconditions

Function

```
double sqrt(double value);
```

```
// Return the square root of the specified  
// 'value'. The behavior is undefined unless  
// '0 <= value'.
```

Precondition

2. Design & Implementation

Preconditions and Postconditions Function

```
double sqrt(double value);
// Return the square root of the specified
// 'value'. The behavior is undefined unless
// '0 <= value'.
```

Precondition

For a Stateless Function:
Restriction on syntactically legal inputs.

2. Design & Implementation

Preconditions and Postconditions

Function

```
double sqrt(double value);
```

```
// Return the square root of the specified  
// 'value'. The behavior is undefined unless  
// '0 <= value'.
```

2. Design & Implementation

Preconditions and Postconditions

Function

```
double sqrt(double value);
```

```
// Return the square root of the specified  
// 'value'. The behavior is undefined unless  
// '0 <= value'.
```

Postcondition

2. Design & Implementation

Preconditions and Postconditions Function

```
double sqrt(double value);
```

```
// Return the square root of the specified  
// 'value'. The behavior is undefined unless  
// '0 <= value'.
```

Postcondition

For a Stateless Function:
What it “returns”.

2. Design & Implementation

Preconditions and Postconditions

Object Method

2. Design & Implementation

Preconditions and Postconditions

Object Method

- Preconditions: What must be true of both (object) state and method inputs; otherwise the behavior is undefined.

2. Design & Implementation

Preconditions and Postconditions

Object Method

- Preconditions: What must be true of both (object) state and method inputs; otherwise the behavior is undefined.

- Postconditions: What must happen as a function of (object) state and method inputs if all preconditions are satisfied.

2. Design & Implementation

Preconditions and Postconditions

Object Method

a.k.a.
*Essential
Behavior*

- Preconditions: What must be true of both (object) state and method inputs if all postconditions are satisfied.
- Postconditions: What must happen as a function of (object) state and method inputs if all preconditions are satisfied.

Note that *Essential Behavior* refers to a superset of *Postconditions* that includes behavioral guarantees, such as runtime complexity.

a.k.a.
*Essential
Behavior*

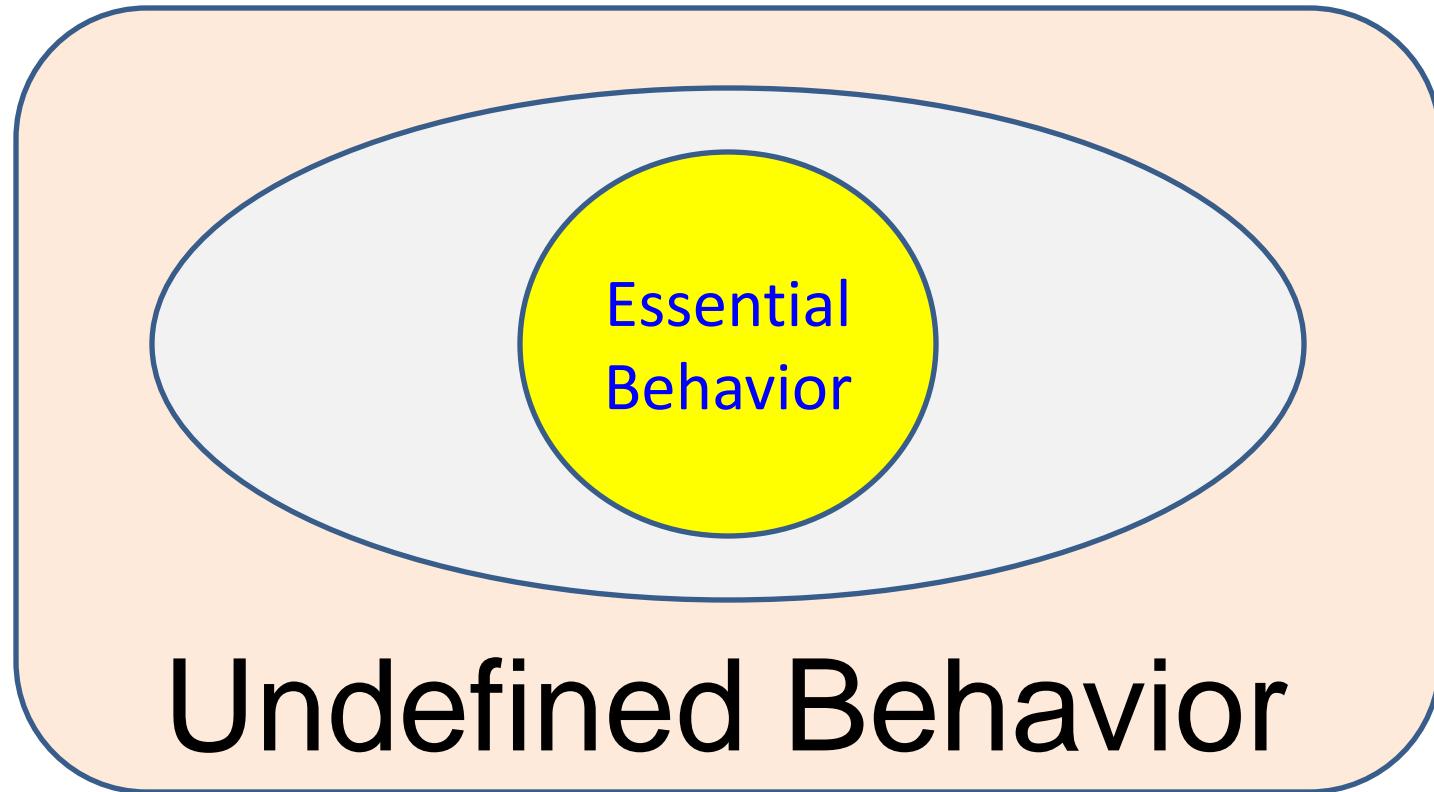
- Preconditions: What must happen as a function of (object) state and method inputs if all postconditions are satisfied.
- Postconditions: What must happen as a function of (object) state and method inputs if all preconditions are satisfied.

Observation By
Kevlin Henny

2. Design & Implementation

Preconditions and Postconditions

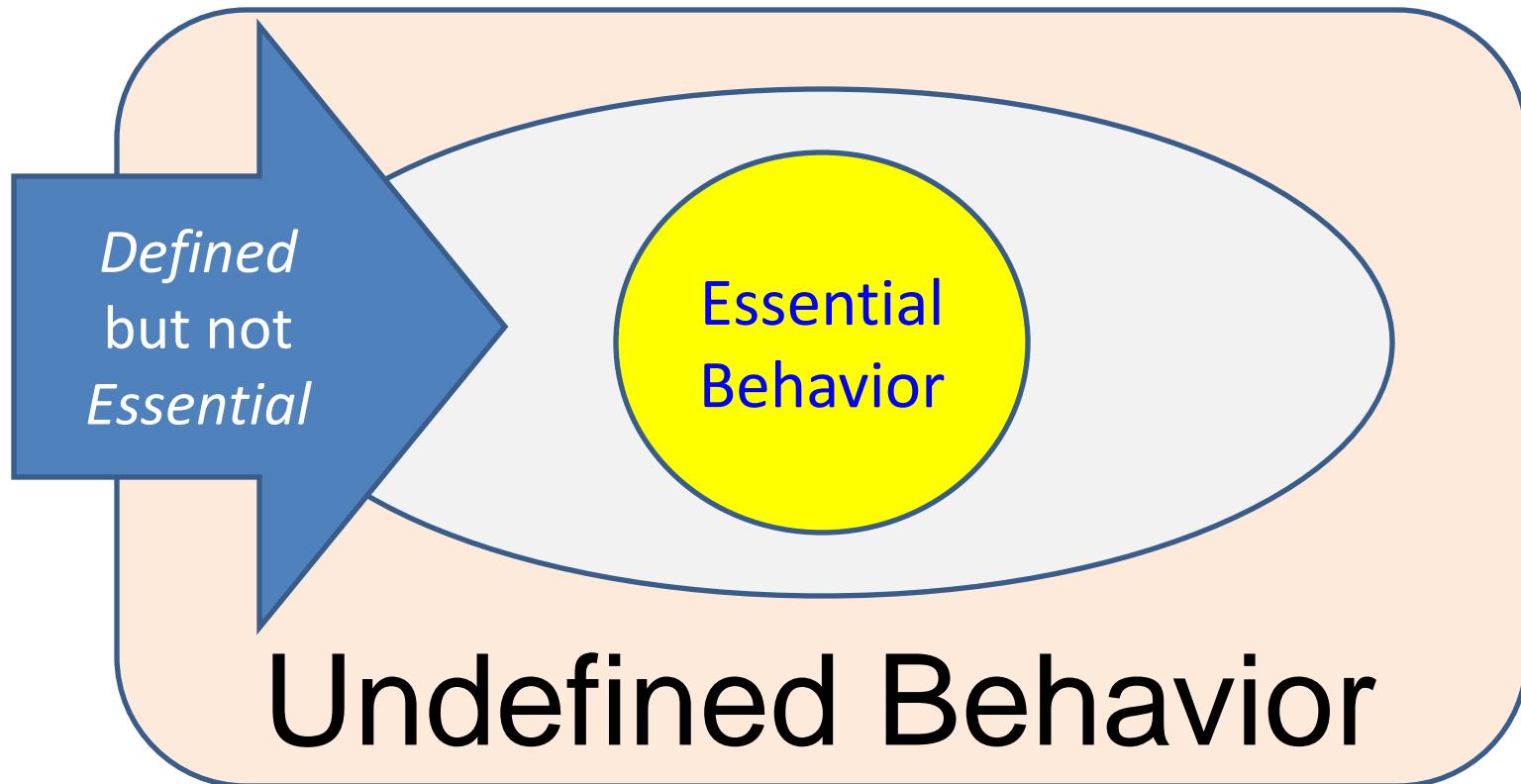
Defined & Essential Behavior



2. Design & Implementation

Preconditions and Postconditions

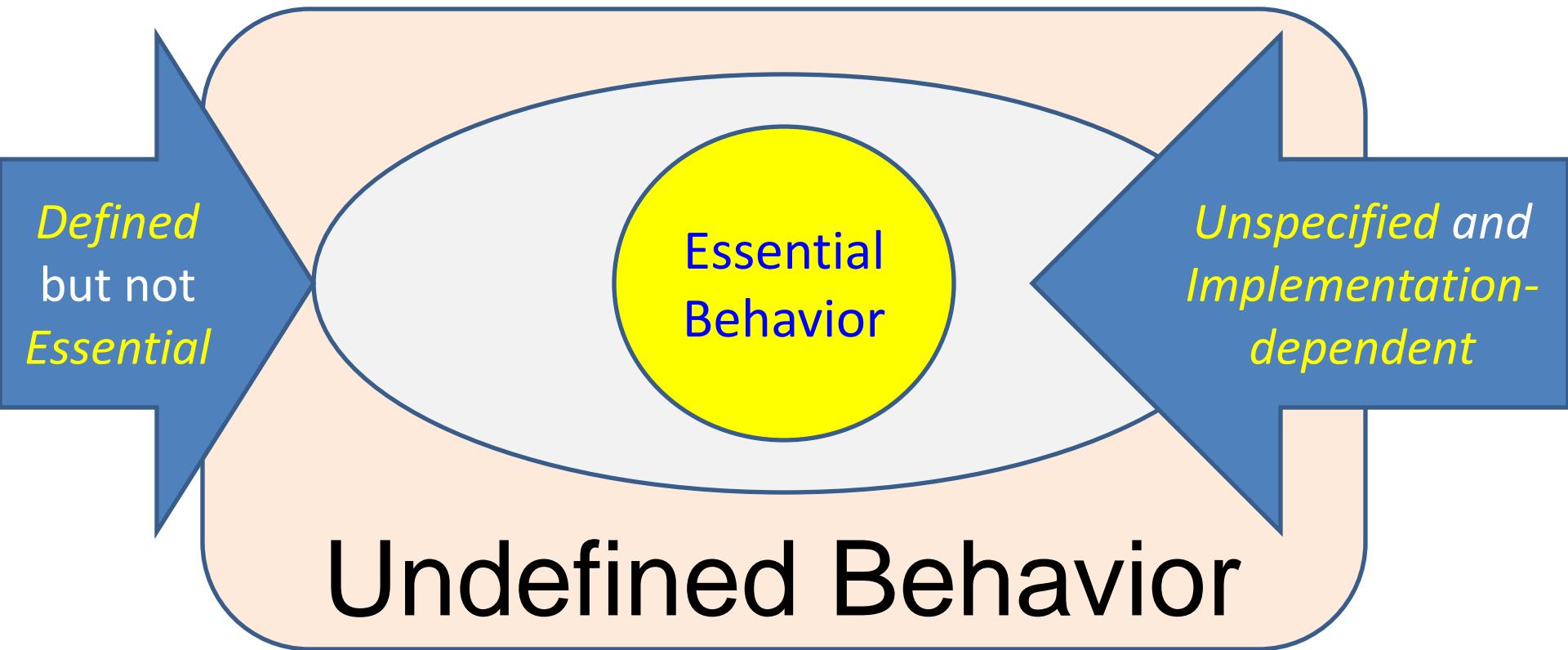
Defined & Essential Behavior



2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior



2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,  
                    int          level      = 0,  
                    int          spacesPerLevel = 4) const;  
  
// Format this object to the specified output 'stream' at the (absolute  
// value of) the optionally specified indentation 'level', and return a  
// reference to 'stream'. If 'level' is specified, optionally specify  
// 'spacesPerLevel', the number of spaces per indentation level for  
// this and all of its nested objects. If 'level' is negative,  
// suppress indentation of the first line. If 'spacesPerLevel' is  
// negative, format the entire output on one line, suppressing all but  
// the initial indentation (as governed by 'level'). If 'stream' is  
// not valid on entry, this operation has no effect.
```

2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,  
                    int          level      = 0,  
                    int          spacesPerLevel = 4) const;  
  
// Format this object to the specified output 'stream' at the (absolute  
// value of) the optionally specified indentation 'level', and return a  
// reference to 'stream'. If 'level' is specified, optionally specify  
// 'spacesPerLevel', the number of spaces per indentation level for  
// this and all of its nested objects. If 'level' is negative,  
// suppress indentation of the first line. If 'spacesPerLevel' is  
// negative, format the entire output on one line, suppressing all but  
// the initial indentation (as governed by 'level'). If 'stream' is  
// not valid on entry, this operation has no effect.
```

2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,  
                    int          level      = 0,  
                    int          spacesPerLevel = 4) const;  
  
// Format this object to the specified output 'stream' at the (absolute  
// value of) the optionally specified indentation 'level', and return a  
// reference to 'stream'. If 'level' is specified, optionally specify  
// 'spacesPerLevel', the number of spaces per indentation level for  
// this and all of its nested objects. If 'level' is negative,  
// suppress indentation of the first line. If 'spacesPerLevel' is  
// negative, format the entire output on one line, suppressing all but  
// the initial indentation (as governed by 'level'). If 'stream' is  
// not valid on entry, this operation has no effect.
```

2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,
                    int           level        = 0,
                    int           spacesPerLevel = 4) const;
// Format this object to the specified output 'stream' at the (absolute
// value of) the optionally specified indentation 'level', and return a
// reference to 'stream'. If 'level' is specified, optionally specify
// 'spacesPerLevel', the number of spaces per indentation level for
// this and all of its nested objects. If 'level' is negative,
// suppress indentation of the first line. If 'spacesPerLevel' is
// negative, format the entire output on one line, suppressing all but
// the initial indentation (as governed by 'level'). If 'stream' is
// not valid on entry, this operation has no effect.
```

2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,
                    int           level      = 0,
                    int           spacesPerLevel = 4) const;
// Format this object to the specified output 'stream' at the (absolute
// value of) the optionally specified indentation 'level', and return a
// reference to 'stream'. If 'level' is specified, optionally specify
// 'spacesPerLevel', the number of spaces per indentation level for
// this and all of its nested objects. If 'level' is negative,
// suppress indentation of the first line. If 'spacesPerLevel' is
// negative, format the entire output on one line, suppressing all but
// the initial indentation (as governed by 'level'). If 'stream' is
// not valid on entry, this operation has no effect.
```

2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,
                    int           level      = 0,
                    int           spacesPerLevel = 4) const;
// Format this object to the specified output 'stream' at the (absolute
// value of) the optionally specified indentation 'level', and return a
// reference to 'stream'. If 'level' is specified, optionally specify
// 'spacesPerLevel', the number of spaces per indentation level for
// this and all of its nested objects. If 'level' is negative,
// suppress indentation of the first line. If 'spacesPerLevel' is
// negative, format the entire output on one line, suppressing all but
// the initial indentation (as governed by 'level'). If 'stream' is
// not valid on entry, this operation has no effect.
```

2. Design & Implementation

Preconditions and Postconditions

Defined & Essential

Any
Undefined
Behavior?

```
std::ostream& print(std::ostream& stream,
```

```
    int           level      = 0,  
    int           spacesPerLevel = 4) const;
```

```
// Format this object to the specified output 'stream' at the (absolute  
// value of) the optionally specified indentation 'level', and return a  
// reference to 'stream'. If 'level' is specified, optionally specify  
// 'spacesPerLevel', the number of spaces per indentation level for  
// this and all of its nested objects. If 'level' is negative,  
// suppress indentation of the first line. If 'spacesPerLevel' is  
// negative, format the entire output on one line, suppressing all but  
// the initial indentation (as governed by 'level'). If 'stream' is  
// not valid on entry, this operation has no effect.
```

2. Design & Implementation

Preconditions and Postconditions

Defined & Essential

Any
Non-Essential
Behavior?

```
std::ostream& print(std::ostream& stream,  
                    int      level      = 0,  
                    int      spacesPerLevel = 4) const;
```

// Format this object to the specified output 'stream' at the (absolute
// value of) the optionally specified indentation 'level', and return a
// reference to 'stream'. If 'level' is specified, optionally specify
// 'spacesPerLevel', the number of spaces per indentation level for
// this and all of its nested objects. If 'level' is negative,
// suppress indentation of the first line. If 'spacesPerLevel' is
// negative, format the entire output on one line, suppressing all but
// the initial indentation (as governed by 'level'). If 'stream' is
// not valid on entry, this operation has no effect.

2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/month'/day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantics.  
    // a valid date between the dates 0001/01/01  
    // 9999/12/31 inclusive.  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```



2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantics.  
    // a valid date between the dates 0001/01/01  
    // 9999/12/31 inclusive.  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/month'/day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```



2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

2. Design & Implementation

Preconditions and Postconditions

Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
    //...  
public:  
    Date(int year, int month, int day);  
    // Create a valid date from the specified  
    // 'day'. The behavior is undefined if  
    // represents a valid date in the range [0001/01/01, 9999/12/31].  
    Date(const Date& original);  
    // Create a date having the value of the specified 'original' date.  
    // ...  
};
```



2. Design & Implementation

Preconditions and Postconditions (Object) Invariants

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

2. Design & Implementation

Preconditions and Postconditions (Object) Invariants

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
    public:  
        Date(int year, int month, int day);  
            // Create a valid date from the specified 'year', 'month', and  
            // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
            // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
        Date(const Date& original);  
            // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

2. Design & Implementation

Preconditions and Postconditions (Object) Invariants

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.
```

```
//...
```

```
p
```

```
};
```

Question: Must the code itself preserve invariants even if one or more Preconditions of a method's contract is violated?

e.

2. Design & Implementation

Preconditions and Postconditions (Object) Invariants

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/month'/day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

2. Design & Implementation

Preconditions and Postconditions (Object) Invariants

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
    public:  
        Date(int year, int month, int day);  
            // Create a valid date from the specified 'year', 'month', and  
            // 'day'. The behavior is undefined unless 'year'/month'/day'  
            // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
        Date(const Date& original);  
            // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

Answer: No!

2. Design & Implementation

What happens
when behavior
is undefined
is undefined!

public:

```
Date(int year, int month, int day);  
    // Create a valid date from the specified 'year', 'month', and  
    // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
    // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
Date(const Date& original);  
    // Create a date having the value of the specified 'original' date.  
// ...  
};
```

and Postconditions
arians

semantic type representing
001/01/01 and

Answer: No!

2. Design & Implementation

Design by Contract

(DbC)

“If you give me valid input*,
I will behave as advertised;
otherwise, all bets are off!”

*including state

2. Design & Implementation

Design by Contract

Documentation

There are five aspects:

1. What it does.
2. What it returns.
3. *Essential Behavior.*
4. *Undefined Behavior.*
5. Note that...

2. Design & Implementation

Design by Contract

Documentation

There are five aspects:

1. **What it does.**
2. What it returns.
3. *Essential Behavior.*
4. *Undefined Behavior.*
5. Note that...

2. Design & Implementation

Design by Contract

Documentation

There are five aspects:

1. What it does.
2. **What it returns.**
3. *Essential Behavior.*
4. *Undefined Behavior.*
5. Note that...

2. Design & Implementation

Design by Contract

Documentation

There are five aspects:

1. What it does.
2. What it returns.
- 3. *Essential Behavior.***
- 4. *Undefined Behavior.***
5. Note that...

2. Design & Implementation

Design by Contract

Documentation

There are five aspects:

1. What it does.
2. What it returns.
3. *Essential Behavior.*
4. ***Undefined Behavior.***
5. Note that...

2. Design & Implementation

Design by Contract

Documentation

There are five aspects:

1. What it does.
2. What it returns.
3. *Essential Behavior.*
4. *Undefined Behavior.*
5. **Note that...**

2. Design & Implementation

Design by Contract

Verification

2. Design & Implementation

Design by Contract

Verification

➤ **Preconditions:**

2. Design & Implementation

Design by Contract

Verification

- **Preconditions:**
 - ✓ RTFM (Read the Manual).

2. Design & Implementation

Design by Contract

Verification

➤ **Preconditions:**

- ✓ RTFM (Read the Manual).
- ✓ Assert (only in ‘*debug*’ or ‘*safe*’ mode).

2. Design & Implementation

Design by Contract

Verification

➤ Preconditions:

- ✓ RTFM (Read the Manual).
- ✓ Assert (only in ‘*debug*’ or ‘*safe*’ mode).

For more about
Assertions and “**Safe Mode**”
see the **bsls_assert** component.

2. Design & Implementation

Design by Contract

Verification

➤ **Preconditions:**

- ✓ RTFM (Read the Manual).
- ✓ Assert (only in ‘*debug*’ or ‘*safe*’ mode).

➤ **Postconditions:**

2. Design & Implementation

Design by Contract

Verification

➤ Preconditions:

- ✓ RTFM (Read the Manual).
- ✓ Assert (only in ‘debug’ or ‘safe’ mode).

➤ Postconditions:

- ✓ Component-level test drivers.

2. Design & Implementation

Design by Contract

Verification

➤ Preconditions:

- ✓ RTFM (Read the Manual).
- ✓ Assert (only in ‘debug’ or ‘safe’ mode).

➤ Postconditions:

- ✓ Component-level test drivers.

➤ Invariants:

2. Design & Implementation

Design by Contract

Verification

➤ **Preconditions:**

- ✓ RTFM (Read the Manual).
- ✓ Assert (only in ‘*debug*’ or ‘*safe*’ mode).

➤ **Postconditions:**

- ✓ Component-level test drivers.

➤ **Invariants:**

- ✓ Assert invariants in the destructor.

2. Design & Implementation

Essential Strategies and Techniques

Integral to our design process are:

- a) Common Class Categories
- b) Unique *Vocabulary* Types
- c) Design By Contract
- d) Appropriately *Narrow* Contracts
- e) An Overriding Customer Focus

2. Design & Implementation

Essential Strategies and Techniques

Integral to our design process are:

- a) Common Class Categories
- b) Unique *Vocabulary* Types
- c) Design By Contract
- d) Appropriately *Narrow Contracts*
- e) An Overriding Customer Focus

2. Design & Implementation

Defensive Programming

2. Design & Implementation

Defensive Programming

(DP)

- What is it?

2. Design & Implementation

Defensive Programming

(DP)

- What is it?

Redundant Code that provides runtime checks to detect and report (but **not** “handle” or “hide”) defects in software.

2. Design & Implementation

Defensive Programming (DP)

- What is it?
- Is it Good or Bad?

2. Design & Implementation

Defensive Programming

(DP)

- What is it?
- Is it Good or Bad?

Both: It adds overhead, but can help identify defects early in the development process.

2. Design & Implementation

Defensive Programming

(DP)

- What is it?
- Is it Good or Bad?
- Which is Better: DP or DbC?

2. Design & Implementation

Defensive Programming

(DP)

- What is it?
- Is it Good or Bad?
- Which is Better: DP or DbC?

Do you ride the bus to school
or do you take your lunch?

2. Design & Implementation

Defensive Programming

What are we defending against?

2. Design & Implementation

Defensive Programming

What are we defending against?

➤ Bugs in software
that we use in our
implementation?

2. Design & Implementation

Defensive Programming

What are we defending against?

- Bugs in software that we use in our implementation?
- Bugs we introduce into our own implementation?

2. Design & Implementation

Defensive Programming

What are we defending against?

- Bugs in software that we use in our implementation?
- Bugs we introduce into our own implementation?
- Misuse by our clients.

2. Design & Implementation

Defensive Programming

What are we defending against?

- Bugs in software that we use in our implementation?
- Bugs we introduce into our own implementation?
- Misuse by our clients?

2. Design & Implementation

Defensive Programming

What are we defending against?

- Bugs in software that we use in our implementation?
- Bugs we introduce into our own implementation?
- Misuse by our clients?

2. Design & Implementation

Defensive Programming

What are we defending against?

- Bugs in software that we use in our implementation?
- Bugs we introduce into our own implementation?
- Misuse by our clients?

2. Design & Implementation

Defensive Programming

What are we defending against?

MISUSE BY
OUR CLIENTS

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

What should happen with the following call?

```
std::size_t x = std::strlen(0);
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

What should happen with the following call?

```
std::size_t x = std::strlen(0);
```

How about it must return 0?

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
size_t strlen(const char *s)
{
    if (!s) return 0;           } Wide
    // ...
}
```

How about it must return 0?

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
size_t strlen(const char *s)
{
    if (!s) return 0;           } Wide
    // ...
}
```

Likely to mask a defect

How about it must return 0?

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
size_t strlen(const char *s)
{
    if (!s, ...)
        return 0;
}
```

More code
Run slowly to mask a defect
How about it must return 0?
...
} Wide

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

What should happen with the following call?

```
std::size_t x = std::strlen(0);
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

What should happen with the following call?

```
std::size_t x = std::strlen(0);
```

Undefined Behavior

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
size_t strlen(const char *s)
{
    assert(s);
    // ...
}
```

} Narrow

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
size_t strlen(const char *s)
{
    // ...
}
```

}] Narrow

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
size_t strlen(const char*s)  
{  
    // ...  
}
```

Just Don't Pass 0! } Narrow

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

Should

Date:: setDate(int, int, int);

Return a status?

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

Should

Date::setDate(int, int, int);

Return a status?

Absolutely
Not!

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

I “know” this date is valid (It’s my birthday)!

```
date.setDate(3, 8, 59);
```

Therefore, why should I bother to check status?

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

I “know” this date is valid (It’s my birthday)!

```
date.setDate(3, 8, 59);
```

Therefore, why should I bother to check status?

```
date.setDate(1959, 3, 8);
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

I “know” this date is valid (It’s my birthday)!

```
date.setDate(1959, 8, 59);
```

Therefore, why should I bother to check status?

```
date.setDate(1959, 3, 8);
```

Double Fault!!

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

- Returning status implies a wide interface contract.

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

- Returning status implies a wide interface contract.
- Wide contracts prevent defending against such errors in any build mode.

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
void Date:: setDate(int y,  
                    int m,  
                    int d)  
{  
  
    d_year    = y;  
    d_month   = m;  
    d_day     = d;  
}
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
void Date:: setDate(int y,  
                    int m,  
                    int d)  
{  
    assert(isValid(y,m,d));  
    d_year = y;  
    d_month = m;  
    d_day = d;  
}
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
void Date:: setDate(int y,  
                    int m,  
                    int d)  
{  
    assert(isValid(y,m,d));  
    d_year = y;  
    d_month = m;  
    d_day = d;  
}
```

Narrow Contract:
Checked Only In
“Debug Mode”

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
int Date:: setDateIfValid(int y,  
                           int m,  
                           int d)  
{  
    if (!isValid(y, m, d)) {  
        return !0;  
    }  
    d_year = y;  
    d_month = m;  
    d_day = d;  
    return 0;  
}
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

```
int Date:: setDateIfValid(int y,  
                           int m,  
                           int d)  
{  
    if (!isValid(y, m, d)) {  
        return !0;  
    }  
    d_year = y;  
    d_month = m;  
    d_day = d;  
    return 0;  
}
```

Wide Contract:
Checked in
Every Build Mode

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

- What should happen when the behavior is undefined?

```
TYPE& vector<TYPE>::operator[] (int idx);
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

- What should happen when the behavior is undefined?

```
TYPE& vector<TYPE>::operator[] (int idx);
```

- Should what happens be part of the contract?

```
TYPE& vector<TYPE>::at (int idx);
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

- What should happen when the behavior is undefined? **It depends on the build mode.**

```
TYPE& vector<TYPE>::operator[] (int idx);
```

- Should what happens be part of the contract?

```
TYPE& vector<TYPE>::at (int idx);
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

- What should happen when the behavior is undefined? **It depends on the build mode.**

```
TYPE& vector<TYPE>::operator[] (int idx);
```

- Should what happens be part of the contract? **If it is, then it's defined behavior!**

```
TYPE& vector<TYPE>::at (int idx);
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

- What should happen when the behavior is undefined? **It depends on the build mode.**

```
TYPE& vector<TYPE>::operator[] (int idx);
```

• Must check in every build mode!

What happens be part of the
If it is, then it's defined behavior!

```
TYPE& vector<TYPE>::at (int idx);
```

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

- What should happen when the behavior is undefined? **It depends on the build mode.**

```
TYPE& vector<TYPE>::operator[] (int idx);
```

• Must check
in every
build mode!

What happens be part
If it is, then it's defi

```
TYPE& vector<TYPE>::at (ir
```

**Bad
Idea!** or!

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

- What about undefined behavior?

CRASH!

Must check
in every
build mode!

What happens be part
If it is, then it's defi
TYPE& vector<TYPE>::at (ir

Bad
Idea!
or!

2. Design & Implementation

Narrow versus Wide Contracts

Narrow Contracts Admit Undefined Behavior:

- What should happen under

TYPE

Or, as we will soon see, ...
Something Much Better!

- Must check what happens in every build mode!

TYPE& vector<TYPE>::at (ir

at happens be part
If it is, then it's defi

Bad Idea!
or!

2. Design & Implementation

Contracts and Exceptions

Preconditions always Imply Postconditions:

2. Design & Implementation

Contracts and Exceptions

Preconditions always Imply Postconditions:

- If a function cannot satisfy its contract (given valid preconditions) it must not return normally.

2. Design & Implementation

Contracts and Exceptions

Preconditions always Imply Postconditions:

- If a function cannot satisfy its contract (given valid preconditions) it must not return normally.
- **abort ()** should be considered a viable alternative to **throw** in virtually all cases (if exceptions are disabled).

2. Design & Implementation

Contracts and Exceptions

Preconditions always Imply Postconditions:

- If a function cannot satisfy its contract (given valid preconditions) it must not return normally.
- `abort()` should be considered a viable alternative to `throw` in virtually all cases (if exceptions are disabled).
- Good library components are *exception agnostic* (*via RAII*).

2. Design & Implementation

Appropriately Narrow Contracts

*Narrow contracts admit *undefined behavior*.*

2. Design & Implementation

Appropriately Narrow Contracts

*Narrow contracts admit *undefined behavior*.*

- Appropriately narrow contracts are GOOD:

2. Design & Implementation

Appropriately Narrow Contracts

*Narrow contracts admit *undefined behavior*.*

- Appropriately narrow contracts are GOOD:
 - Reduce costs associated with development/testing.

2. Design & Implementation

Appropriately Narrow Contracts

*Narrow contracts admit *undefined behavior*.*

- Appropriately narrow contracts are GOOD:
 - Reduce costs associated with development/testing.
 - Improve performance and reduces object-code size.

2. Design & Implementation

Appropriately Narrow Contracts

*Narrow contracts admit *undefined behavior*.*

- Appropriately narrow contracts are GOOD:
 - Reduce costs associated with development/testing.
 - Improve performance and reduces object-code size.
 - Allow useful behavior to be added as needed.

2. Design & Implementation

Appropriately Narrow Contracts

*Narrow contracts admit *undefined behavior*.*

- Appropriately narrow contracts are GOOD:
 - Reduce costs associated with development/testing.
 - Improve performance and reduces object-code size.
 - Allow useful behavior to be added as needed.
 - Enable practical/effective *Defensive Programming*.

2. Design & Implementation

Appropriately Narrow Contracts

*Narrow contracts admit *undefined behavior*.*

- Appropriately narrow contracts are GOOD:
 - Reduce costs associated with development/testing.
 - Improve performance and reduces object-code size.
 - Allow useful behavior to be added as needed.
 - Enable practical/effective *Defensive Programming*.
- *Defensive programming* means:

Fault Intolerance!

2. Design & Implementation

Essential Strategies and Techniques

Integral to our design process are:

- a) Common Class Categories
- b) Unique *Vocabulary* Types
- c) Design By Contract
- d) Appropriately *Narrow* Contracts
- e) An Overriding Customer Focus

2. Design & Implementation

Essential Strategies and Techniques

Integral to our design process are:

- a) Common Class Categories
- b) Unique *Vocabulary* Types
- c) Design By Contract
- d) Appropriately *Narrow* Contracts
- e) An Overriding Customer Focus

2. Design & Implementation

An Overriding Customer Focus

(1) Capture Practical Usage Example(s):

2. Design & Implementation

An Overriding Customer Focus

(1) Capture Practical Usage Example(s):

- **First thing we think about** when designing a component...

2. Design & Implementation

An Overriding Customer Focus

(1) Capture Practical Usage Example(s):

- **First thing we think about** when designing a component...
...its *raison d'être*.

2. Design & Implementation

An Overriding Customer Focus

(1) Capture Practical Usage Example(s):

- **First thing we think about** when designing a component...
...its *raison d'être*.
- *Bona fide*, yet appropriately *elided* real-world examples.

2. Design & Implementation

An Overriding Customer Focus

(1) Capture Practical Usage Example(s):

- **First thing we think about** when designing a component...
...its *raison d'être*.
- *Bona fide*, yet appropriately *elided* real-world examples.
- **Last thing we validate** in our component-level test drivers.

2. Design & Implementation

An Overriding Customer Focus

(1) Capture Practical Usage Example(s):

Easy to
Understand

2. Design & Implementation

Usage Example

```
///Usage
///-----
// This section illustrates intended use of this component.
//
///Example 1: Converting Between UTC and Local Times
// -----
// When using the "Zoneinfo" database, we want to represent and access the
// local time information contained in the "Zoneinfo" binary data files. Once
// we have obtained this information, we can use it to convert times from one
// time zone to another. The following code illustrates how to perform such
// conversions using 'baltzo::LocalTimeDescriptor'.
//
// First, we define a 'baltzo::LocalTimeDescriptor' object that characterizes
// the local time in effect for New York Daylight-Saving Time in 2010:
// ...
// enum { NEW_YORK_DST_OFFSET = -4 * 60 * 60 }; // -4 hours in seconds
//
// baltzo::LocalTimeDescriptor newYorkDst(NEW_YORK_DST_OFFSET, true, "EDT");
//
// assert(NEW_YORK_DST_OFFSET == newYorkDst.utcOffsetInSeconds());
// assert(      true == newYorkDst.dstInEffectFlag());
// assert(      "EDT" == newYorkDst.description());
// ...
// Then, we create a 'bdlt::Datetime' representing the time
// "Jul 20, 2010 11:00" in New York:
// ...
// bdlt::Datetime newYorkDatetime(2010, 7, 20, 11, 0, 0);
//
// Next, we convert 'newYorkDatetime' to its corresponding UTC value using the
// 'newYorkDst' descriptor (created above); note that, when converting from a
// local time to a UTC time, the *signed* offset from UTC is *subtracted* from
// the local time:
```

```
///
// bdlt::Datetime utcDatetime = newYorkDatetime;
// utcDatetime.addSeconds(-newYorkDst.utcOffsetInSeconds());
//
// Then, we verify that the result corresponds to the expected UTC time,
// "Jul 20, 2010 15:00":
// ...
// assert(bdlt::Datetime(2010, 7, 20, 15, 0, 0) == utcDatetime);
//
// Next, we define a 'baltzo::LocalTimeDescriptor' object that describes the
// local time in effect for Rome in the summer of 2010:
// ...
// enum { ROME_DST_OFFSET = 2 * 60 * 60 }; // 2 hours in seconds
//
// baltzo::LocalTimeDescriptor romeDst(ROME_DST_OFFSET, true, "CEST");
//
// assert(ROME_DST_OFFSET == romeDst.utcOffsetInSeconds());
// assert(      true == romeDst.dstInEffectFlag());
// assert(      "CEST" == romeDst.description());
// ...
// Now, we convert 'utcDatetime' to its corresponding local-time value in Rome
// using the 'romeDst' descriptor (created above):
// ...
// bdlt::Datetime romeDatetime = utcDatetime;
// romeDatetime.addSeconds(romeDst.utcOffsetInSeconds());
//
// Notice that, when converting from UTC time to local time, the signed
// offset from UTC is *added* to UTC time rather than subtracted.
//
// Finally, we verify that the result corresponds to the expected local time,
// "Jul 20, 2010 17:00":
// ...
// assert(bdlt::Datetime(2010, 7, 20, 17, 0, 0) == romeDatetime);
// ...
```

2. Design & Implementation

An Overriding Customer Focus

(2) Canonical Organization:

2. Design & Implementation

An Overriding Customer Focus

(2) Canonical Organization:

- The **categories** into which information is partitioned.

2. Design & Implementation

An Overriding Customer Focus

(2) Canonical Organization:

- The **categories** into which information is partitioned.
- The **order** in which information is presented.

2. Design & Implementation

An Overriding Customer Focus

(2) Canonical Organization:

- The **categories** into which information is partitioned.
- The **order** in which information is presented.
- The **vocabulary** and **phrasing** ...

2. Design & Implementation

An Overriding Customer Focus

(2) Canonical Organization:

- The **categories** into which information is partitioned.
- The **order** in which information is presented.
- The **vocabulary** and **phrasing** ...
...especially contracts.

2. Design & Implementation

An Overriding Customer Focus

(2) Canonical Organization:

• The organization is easy to find and use.

*Easy to Find
and Use*

2. Design & Implementation

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

2. Design & Implementation

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

- Make it look like one person wrote all the code:

2. Design & Implementation

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

- Make it look like one person wrote all the code:
 - ✓ Unambiguous standard function names...

2. Design & Implementation

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

- Make it look like one person wrote all the code:
 - ✓ Unambiguous standard function names:
`clear` v. `removeAll`
`empty` v. `isEmpty`

2. Design & Implementation

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

- Make it look like one person wrote all the code:
 - ✓ Unambiguous standard function names:
`clear` v. `removeAll`
`empty` v. `isEmpty`
 - ✓ Consistent Argument Order...

2. Design & Implementation

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

- Make it look like one person wrote all the code:
 - ✓ Unambiguous standard function names:
`clear` v. `removeAll`
`empty` v. `isEmpty`
 - ✓ Consistent Argument Order: **Outputs, Inputs, Parameters.**

2. Design & Implementation

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

- Make it look like one person wrote all the code:
 - ✓ Unambiguous standard function names:
`clear` v. `removeAll`
`empty` v. `isEmpty`
 - ✓ Consistent Argument Order: **Outputs, Inputs, Parameters.**
 - ✓ Appropriate use of pointers/references to indicate intent...

2. Design & Implementation

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

- Make it look like one person wrote all the code:
 - ✓ Unambiguous standard function names:
`clear` v. `removeAll`
`empty` v. `isEmpty`
 - ✓ Consistent Argument Order: **Outputs, Inputs, Parameters.**
 - ✓ Appropriate use of pointers/references to indicate intent *directly from the client source code.*

2. Design & Implementation

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

- Make it look like

*Helpful
visual Cues*

Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment (BDE)

Rendered as Fine-Grained Hierarchically Reusable Components.

Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment (BDE)

Rendered as Fine-Grained Hierarchically Reusable Components.

3. Verification & Testing

Essential Strategies and Techniques

Ensuring our own reliability while improving that of our clients:

- a) Component-Level Testing
- b) Peer Review
- c) Static Analysis Tools
- d) Defensive (Precondition) Checks

3. Verification & Testing

Essential Strategies and Techniques

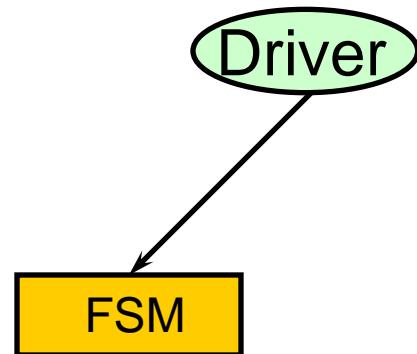
Ensuring our own reliability while improving that of our clients:

- a) Component-Level Testing
- b) Peer Review
- c) Static Analysis Tools
- d) Defensive (Precondition) Checks

3. Verification & Testing

Testing Proximately?

A small state machine is easy to test.

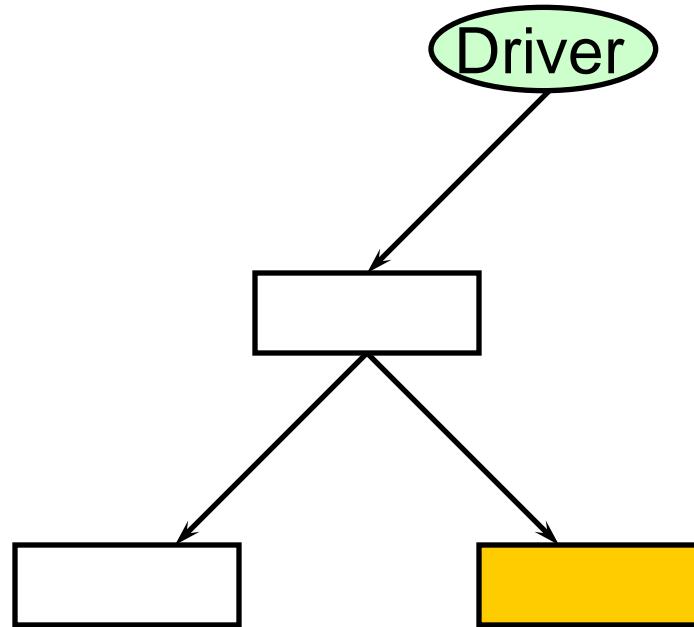


3. Verification & Testing

Testing Proximately?

But even if all states are theoretically reachable

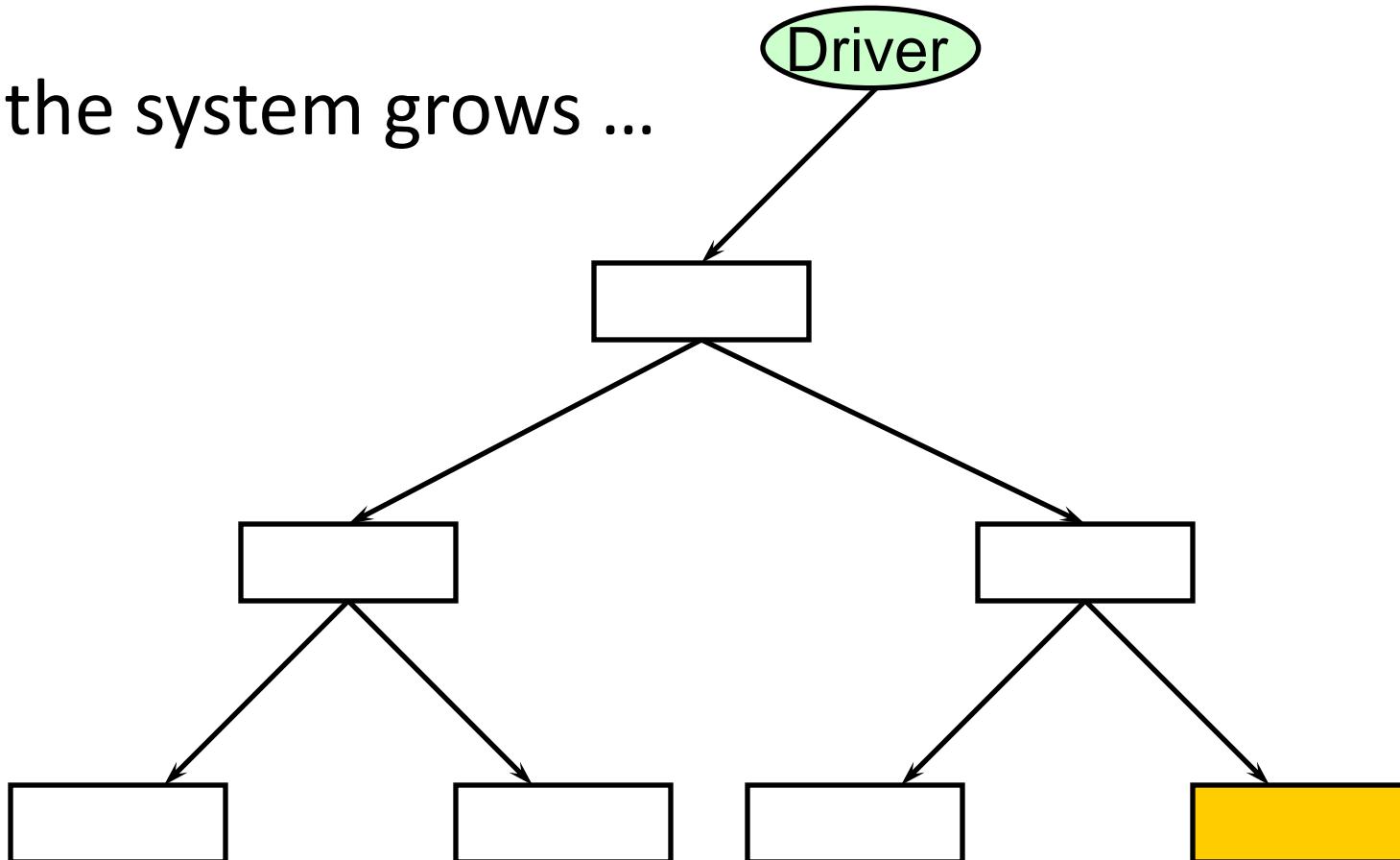
...



3. Verification & Testing

Testing Proximately?

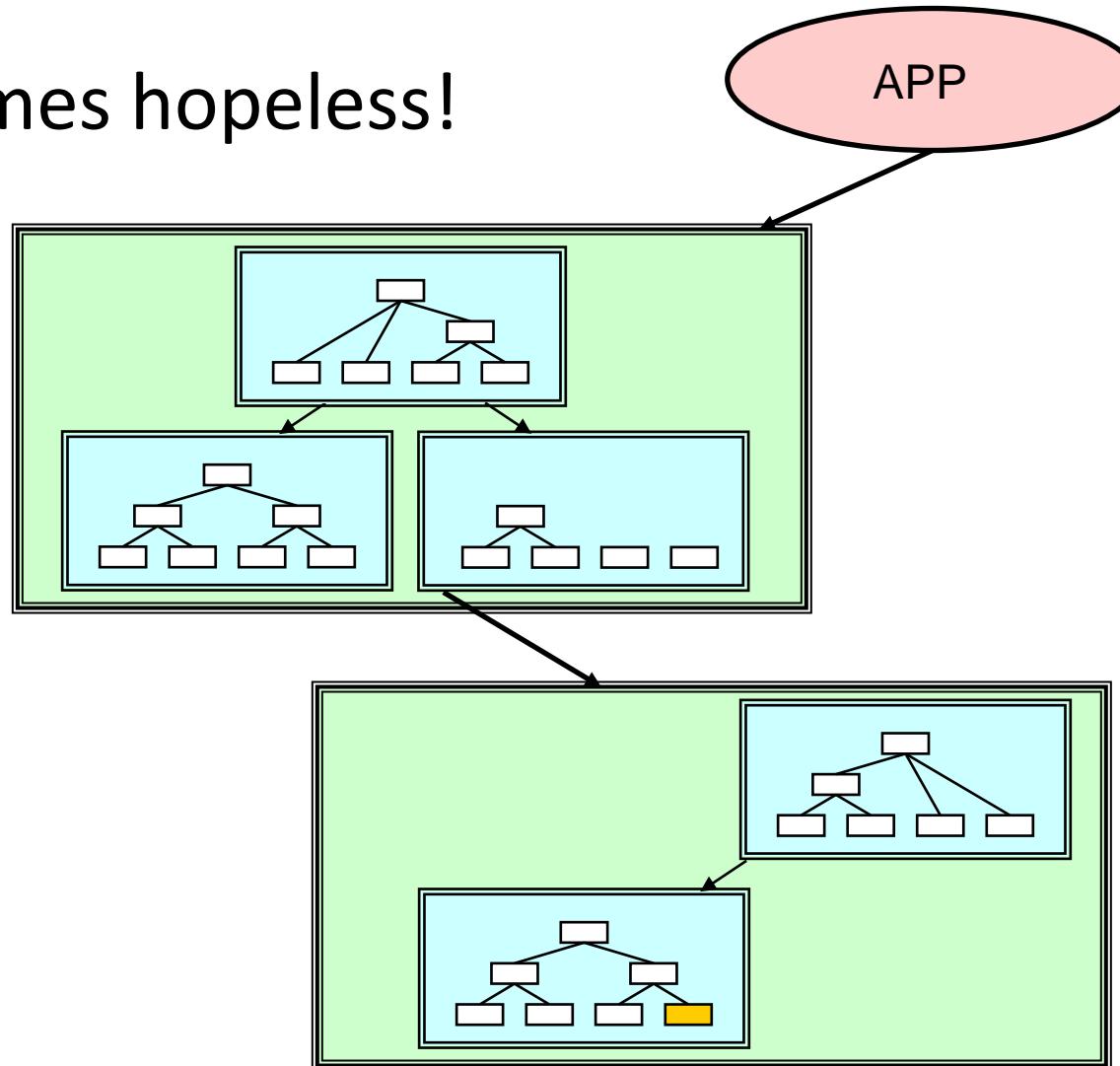
As the system grows ...



3. Verification & Testing

Testing Proximately?

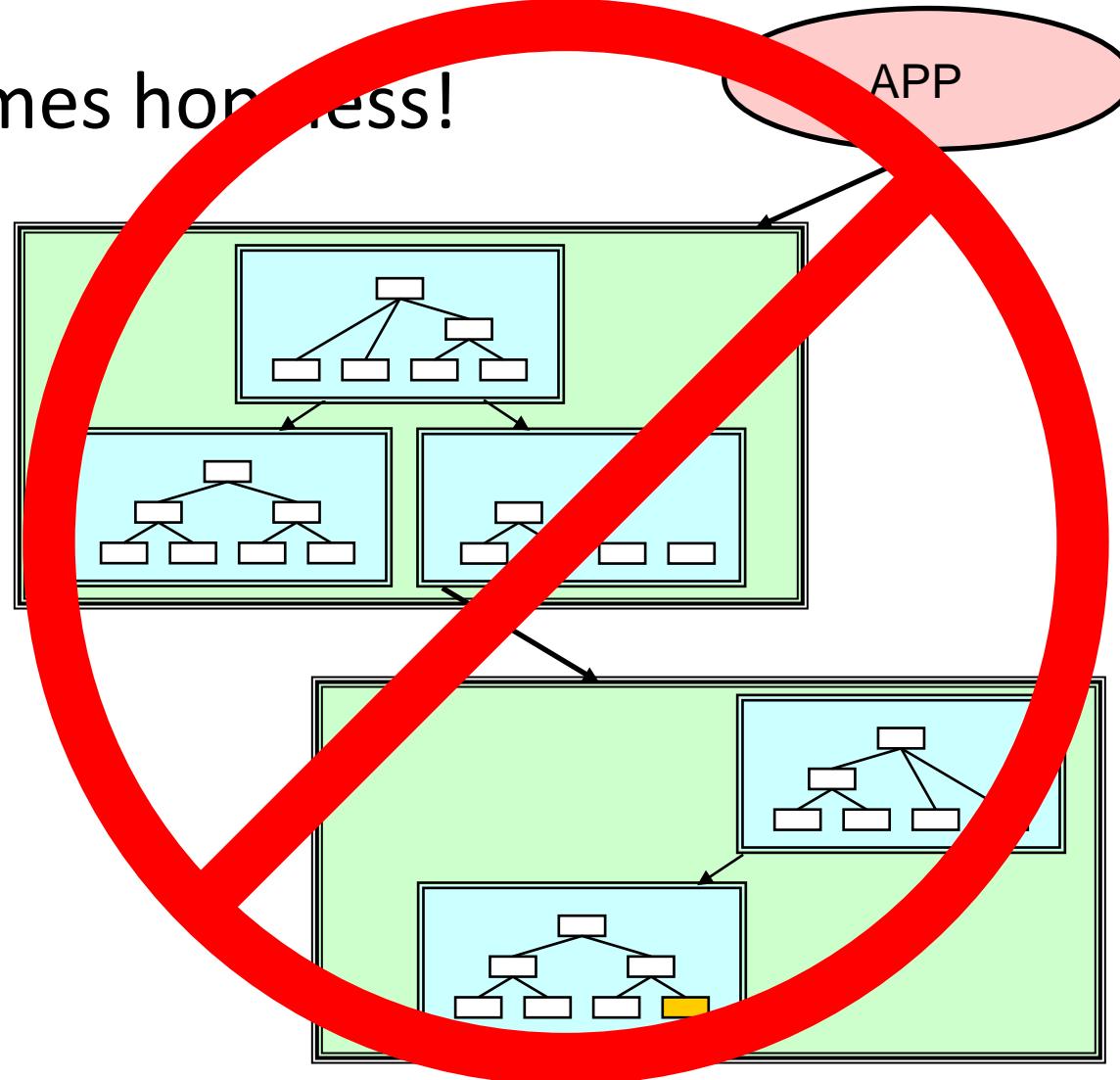
It becomes hopeless!



3. Verification & Testing

Testing Proximately?

It becomes hopeless!

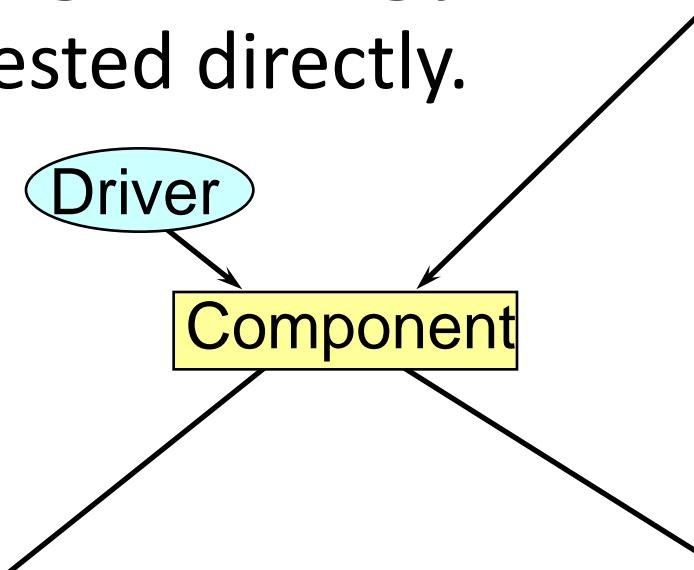


3. Verification & Testing

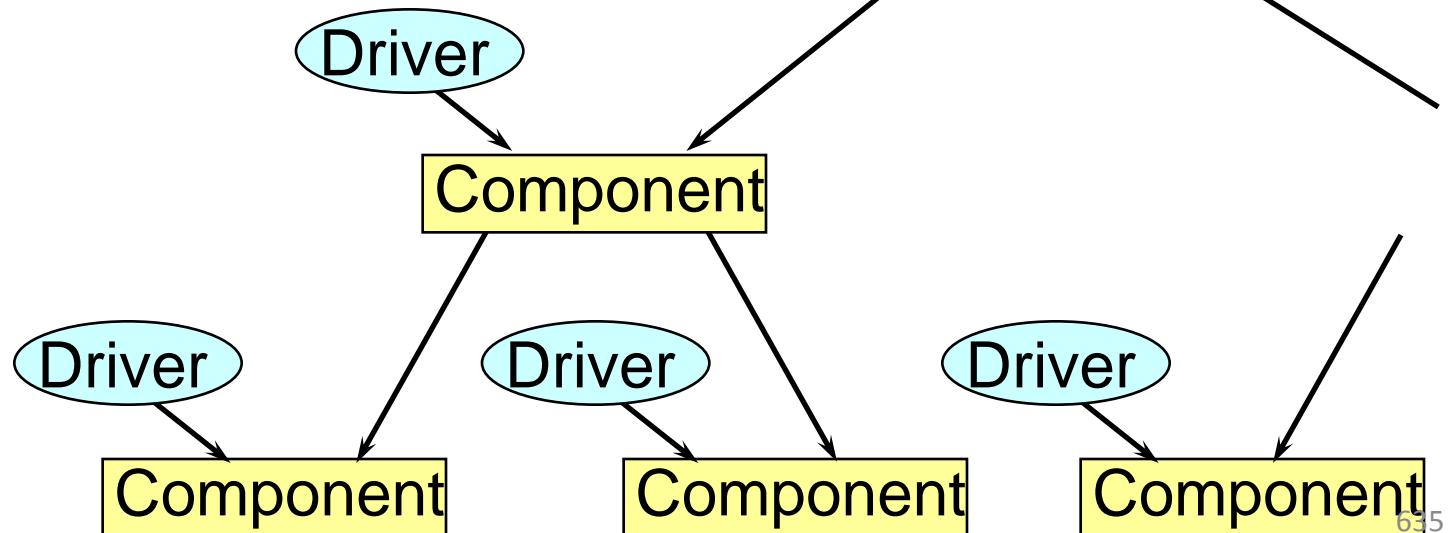
Component-Level Testing

Hierarchical Testing Strategy:
Each component is tested directly.

Level 3:



Level 2:

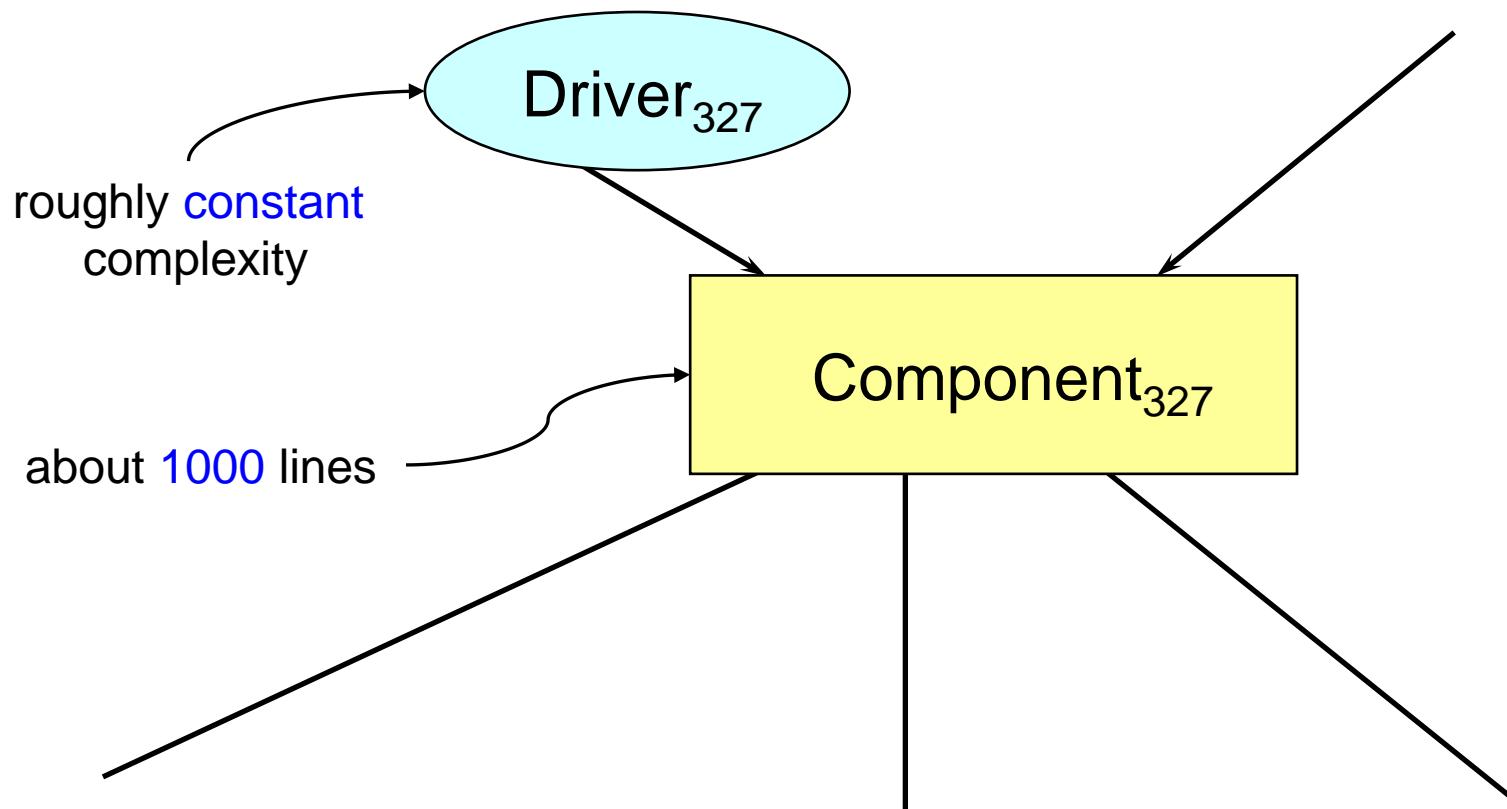


Level 1:

3. Verification & Testing

Component-Level Testing

Incremental Functionality Testing:
Test only the **value added** by a component.
No need to retest subordinate functionality.



3. Verification & Testing

Component-Level Testing

Component-level testing methodology overview:

3. Verification & Testing

Component-Level Testing

Component-level testing methodology overview:

1. Provide a fundamentally different representation of behavior.

3. Verification & Testing

Component-Level Testing

Component-level testing methodology overview:

1. Provide a fundamentally different representation of behavior.

2. Use one of our systematic *Test Data Selection Methods*.

3. Verification & Testing

Component-Level Testing

Component-level testing methodology overview:

1. Provide a fundamentally different representation of behavior.
2. Use one of our systematic *Test Data Selection Methods*.
3. Apply our standard *Test Case Implementation Techniques*.

3. Verification & Testing

Component-Level Testing

Component-level testing methodology overview:

1. Provide a fundamentally different representation of behavior.
2. Use one of our systematic *Test Data Selection Methods*.
3. Apply our standard *Test Case Implementation Techniques*.
4. Order test cases so as to exploit already tested functionality.

3. Verification & Testing

Component-Level Testing

Component-level testing methodology overview:

1. Provide a fundamentally different representation of behavior.
2. Use one of our systematic *Test Data Selection Methods*.
3. Apply our standard *Test Case Implementation Techniques*.
4. Order test cases so as to exploit already tested functionality.

Lots more to this story!

3. Verification & Testing

The Component-Level Test Driver

3. Verification & Testing

The Component-Level Test Driver

What is a Test Driver?

3. Verification & Testing

The Component-Level Test Driver

Test Driver

```
// component.t.cpp
#include <component.h>
// ...
int main(...)
{
    //...
}
//-- END OF FILE --
```

component.t.cpp

```
// component.h
```

```
// ...
```

```
//-- END OF FILE --
```

component.h

```
// component.cpp
```

```
#include <component.h>
// ...
```

```
//-- END OF FILE --
```

component.cpp

component

3. Verification & Testing

The Component-Level Test Driver

What is a Test Driver?

- It's a **tool** for developers
 - used during the initial development process.

3. Verification & Testing

The Component-Level Test Driver

What is a Test Driver?

- It's a **tool** for developers
 - used during the initial development process.
- It's a **"cartridge"** for an automated regression-testing system
 - used throughout the lifetime of the component.

3. Verification & Testing

The Component-Level Test Driver

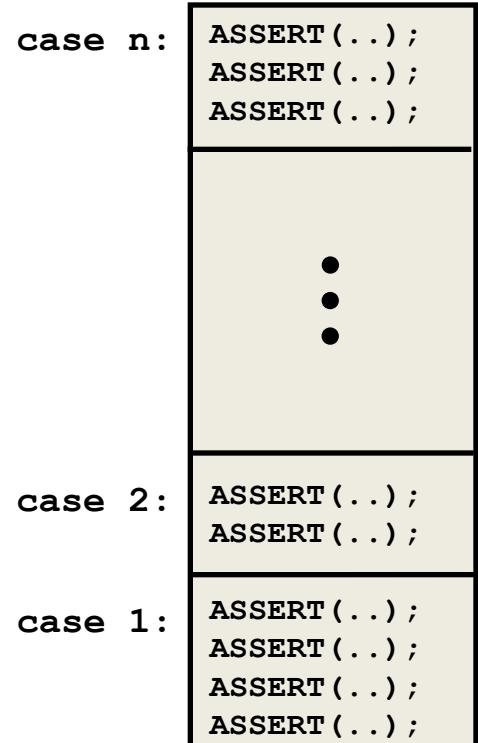
What does a BDE Test Driver comprise?

3. Verification & Testing

The Component-Level Test Driver

What does a BDE Test Driver comprise?

- Set of consecutively numbered **test cases**.

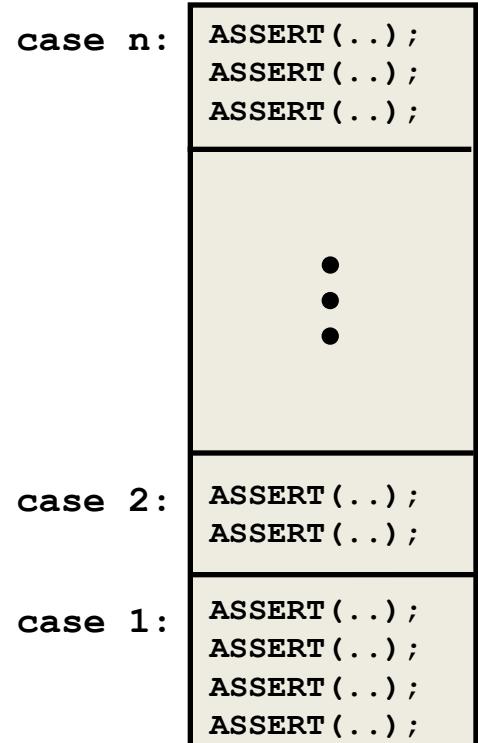


3. Verification & Testing

The Component-Level Test Driver

What does a BDE Test Driver comprise?

- Set of consecutively numbered ***test cases***.
- Each *test case* performs some number of individual ***ASSERTIONS***.



3. Verification & Testing

The Component-Level Test Driver

What is the *User Experience*?

3. Verification & Testing

The Component-Level Test Driver

What is the *User Experience*?

- A test driver should succeed quietly in production.

3. Verification & Testing

The Component-Level Test Driver

What is the *User Experience*?

- A test driver should succeed quietly in production.
- When an error occurs, the test driver should report the offending expression along with the line number:

```
filename(line #): 2 == sqrt(4)  (failed)
```

3. Verification & Testing

The Component-Level Test Driver

Verbose Mode:

```
Testing length 0
    without aliasing
    with aliasing
Testing length 1
    without aliasing
    with aliasing
Testing length 2
    without aliasing
    with aliasing
• • •
```

3. Verification & Testing

BDE Test-Driver Layout

3. Verification & Testing

BDE Test-Driver Layout

```
#include
TEST PLAN
// [ 2] Point(int x, int y)
// [ 1] void setX(int x)
// [ 1] int y() const
// [ 4] void moveBy(int dx, int dy)
// [ 3] void moveTo(int x, int y)
TEST APPARATUS
main(int argc, char argv[]) {
TEST SETUP
    switch (testCase) { case 0:
        case 3: {
            // ...
        }
        case 2: {
            // ...
        }
        case 1: {
            // ...
        }
        default: status = -1;
TEST SHUTDOWN
}
```

- include directives
- test plan identifying case in which each public function is fully tested
- ASSERT macro definition, supporting functions, etc.
- common setup for all test cases
- switch on test case number (actual test code goes here)
- any common cleanup code (rare)

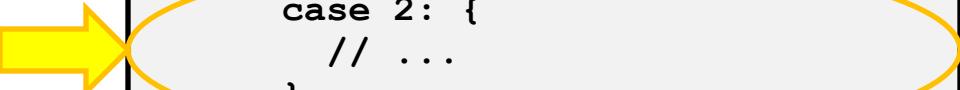
3. Verification & Testing

Test Case

```
#include
TEST PLAN
// [ 2] Point(int x, int y)
// [ 1] void setX(int x)
// [ 1] int y() const
// [ 4] void moveBy(int dx, int dy)
// [ 3] void moveTo(int x, int y)
TEST APPARATUS
main(int argc, char argv[])
TEST SETUP

switch (testCase) { case 0:
    case 3: {
        // ...
    }
    case 2: {
        // ...
    }
    case 1: {
        // ...
    }
    default: status = -1;
}
TEST SHUTDOWN
```

- include directives
- test plan identifying case in which each public function is fully tested
- ASSERT macro definition, supporting functions, etc.
- common setup for all test cases
- switch on test case number (actual test code goes here)
- any common cleanup code (rare)



3. Verification & Testing

Test Case

- **TITLE**
 - Short Label (printed in verbose mode) + optional intro.
- **CONCERNS**
 - Precise (and concise) description of “what could go wrong”
with this particular implementation.
- **PLAN**
 - How this test case will address each of our concerns.
- **TESTING**
 - Copy-and-paste cross-reference from the overall test plan.

3. Verification & Testing

Test Case

- **TITLE**
 - Short Label (printed in verbose mode) + optional intro.
- **CONCERNS**
 - Precise (and concise) description of “what could go wrong”
with this particular implementation.
- **PLAN**
 - How this test case will address each of our concerns.
- **TESTING**
 - Copy-and-paste cross-reference from the overall test plan.

3. Verification & Testing

```
} break;
case 2: { BDE Test-Case Layout
//-----
// UNIQUE BIRTHDAY
// The value returned for an input of 365 is small.

// Concerns:
// 1. That it can represent the result as a double.
// 2. ...
// ...
// 6. That the special-case input of 0 returns 1.
// 7. ...

// Plan:
// Test for explicit values near 0, 365, and INT_MAX.

// Testing:
// double uniqueBirthday(int value);
//-----

if (verbose) cout << endl << "UNIQUE BIRTHDAY" << endl
                << "======" << endl;

// ... test code goes here

} break;
case 1: {
```

3. Verification & Testing

```
} break;
case 2: { BDE Test-Case Layout
//-----
// UNIQUE BIRTHDAY
// The value returned for an input of 365 is small.

// Concerns:
// 1. That it can represent the result as a double.
// 2. ...
// ...
// 6. That the special-case input of 0 returns 1.
// 7. ...

// Plan:
// Test for explicit values near 0, 365, and INT_MAX.

// Testing:
// double uniqueBirthday(int value);
//-----

if (verbose) cout << endl << "UNIQUE BIRTHDAY" << endl
                << "======" << endl;

ASSERT(1 == uniqueBirthday(0));
ASSERT(1 == uniqueBirthday(1));
ASSERT(1 > uniqueBirthday(2));
// ...
ASSERT(0 < uniqueBirthday(365));
ASSERT(0 == uniqueBirthday(366));
```

3. Verification & Testing

```
} break;  
case 2: { BDE Test-Case Layout  
//-----  
// UNIQUE BIRTHDAY  
// The value returned for an input of 365 is small.  
  
// Concerns:  
// 1. That it can represent the result as a double.  
// 2. ...  
// ...  
// 6. That the special-case input of 0 returns 1.  
// 7. ...  
  
// Plan:  
// Test for explicit values near 0, 365, and INT_MAX.  
  
// Testing:  
// double uniqueBirthday(int value);  
//-----  
  
if (verbose) cout << endl << "UNIQUE BIRTHDAY" << endl  
    << "======" << endl;  
  
ASSERT(1 == uniqueBirthday(0));  
ASSERT(1 == uniqueBirthday(1));  
ASSERT(1 > uniqueBirthday(2));  
// ...  
ASSERT(0 < uniqueBirthday(365));  
ASSERT(0 == uniqueBirthday(366));
```

3. Verification & Testing

Essential Strategies and Techniques

Ensuring our own reliability while improving that of our clients:

- a) Component-Level Testing
- b) Peer Review
- c) Static Analysis Tools
- d) Defensive (Precondition) Checks

3. Verification & Testing

Essential Strategies and Techniques

Ensuring our own reliability while improving that of our clients:

- a) Component-Level Testing
- b) Peer Review
- c) Static Analysis Tools
- d) Defensive (Precondition) Checks

3. Verification & Testing

Peer Review

Having another developer review your code helps to ensure that:

- Documentation
- Code
- Tests

are clear, correct, and effective.

3. Verification & Testing

Peer Review

Having another developer review your code helps to ensure that:

- Documentation
- Code
- Tests

are clear, correct, and effective.

3. Verification & Testing

Peer Review

Having another developer review your code helps to ensure that:

- Documentation
- Code
- Tests

are **clear**, **correct**, and **effective**.

3. Verification & Testing

Peer Review

Having another developer review your code

- Documentation
 - Coding style
 - Test cases
- are clear, correct, and effective.
- Is
- Complementary** to
and
- Synergistic** with
- Component-Level Testing.*

3. Verification & Testing

Essential Strategies and Techniques

Ensuring our own reliability while improving that of our clients:

- a) Component-Level Testing
- b) Peer Review
- c) Static Analysis Tools
- d) Defensive (Precondition) Checks

3. Verification & Testing

Essential Strategies and Techniques

Ensuring our own reliability while improving that of our clients:

- a) Component-Level Testing
- b) Peer Review
- c) Static Analysis Tools
- d) Defensive (Precondition) Checks

3. Verification & Testing

Static Analysis Tools

- ❖ Tools (e.g., clang-based) provide additional consistency checks...

3. Verification & Testing

Static Analysis Tools

- ❖ Tools (e.g., clang-based) provide additional consistency checks **that can also be used by our clients!**

3. Verification & Testing

Static Analysis Tools

- ❖ Tools (e.g., clang-based) provide additional consistency checks that can also be used by our clients!
- ❖ Having a highly **consistent** and **regular** implementation structure...

3. Verification & Testing

Static Analysis Tools

- ❖ Tools (e.g., clang-based) provide additional consistency checks that can also be used by our clients!
- ❖ Having a highly **consistent** and **regular** implementation structure **makes** the use of such **tools** all the more **practical** and **effective**.

3. Verification & Testing

Essential Strategies and Techniques

Ensuring our own reliability while improving that of our clients:

- a) Component-Level Testing
- b) Peer Review
- c) Static Analysis Tools
- d) Defensive (Precondition) Checks

3. Verification & Testing

Essential Strategies and Techniques

Ensuring our own reliability while improving that of our clients:

- a) Component-Level Testing
- b) Peer Review
- c) Static Analysis Tools
- d) **Defensive (Precondition) Checks**

3. Verification & Testing

Addressing Client Misuse

As library developers, ...

3. Verification & Testing

Addressing Client Misuse

As library developers, how much CPU should we spend detecting misuse?

3. Verification & Testing

Addressing Client Misuse

As library developers, how much CPU should we spend detecting misuse?

- a. Less than 5%

3. Verification & Testing

Addressing Client Misuse

As library developers, how much CPU should we spend detecting misuse?

- a. Less than 5%
- b. 5% to 20%

3. Verification & Testing

Addressing Client Misuse

As library developers, how much CPU should we spend detecting misuse?

- a. Less than 5%
- b. 5% to 20%
- c. More than 20%, but not more than a constant factor.

3. Verification & Testing

Addressing Client Misuse

As library developers, how much CPU should we spend detecting misuse?

- a. Less than 5%
- b. 5% to 20%
- c. More than 20%, but not more than a constant factor.
- d. Sky's the limit: factor of $O[\log(n)]$, $O[n]$, or more.

3. Verification & Testing

Addressing Client Misuse

As library developers, how much CPU should we spend detecting misuse?

- a. Less than 5%
- b. 5% to 20%
- c. More than 20%, but not more than a constant factor.
- d. Sky's the limit: factor of $O[\log(n)]$, $O[n]$, or more.



3. Verification & Testing

Addressing Client Misuse

As library developers, ...

3. Verification & Testing

Addressing Client Misuse

As library developers, what should happen if we detect misuse?

3. Verification & Testing

Addressing Client Misuse

As library developers, what should happen if we detect misuse?

- a. Be fired?

3. Verification & Testing

Addressing Client Misuse

As library developers, what should happen if we detect misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)

3. Verification & Testing

Addressing Client Misuse

As library developers, what should happen if we detect misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)
- c. Return immediately, but normally? (See a.)

3. Verification & Testing

Addressing Client Misuse

As library developers, what should happen if we detect misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)
- c. Return immediately, but normally? (See a.)
- d. Immediately terminate the program?

3. Verification & Testing

Addressing Client Misuse

As library developers, what should happen if we detect misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)
- c. Return immediately, but normally? (See a.)
- d. Immediately terminate the program?
- e. Throw an exception?

3. Verification & Testing

Addressing Client Misuse

As library developers, what should happen if we detect misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)
- c. Return immediately, but normally? (See a.)
- d. Immediately terminate the program?
- e. Throw an exception?
- f. Spin, waiting to break into a debugger?

3. Verification & Testing

Addressing Client Misuse

As library developers, what should happen if we detect misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)
- c. Return immediately, but normally? (See a.)
- d. Immediately terminate the program?
- e. Throw an exception?
- f. Spin, waiting to break into a debugger?
- g. Something else?

3. Verification & Testing

Addressing Client Misuse

As library developers, what should happen if we detect misuse?



3. Verification & Testing

Addressing Client Misuse

How do we as an enterprise decide what to do?

3. Verification & Testing

Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

3. Verification & Testing

Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?

3. Verification & Testing

Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?
2. Are we in *alpha*, *beta*, or *production*?

3. Verification & Testing

Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?
2. Are we in *alpha*, *beta*, or *production*?
3. Is this a performance-critical application?

3. Verification & Testing

Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?
2. Are we in *alpha*, *beta*, or *production*?
3. Is this a performance-critical application?
4. Is there something sensible to do?
 - a. Save client work before terminating the program.
 - b. Log the error, abandon the current transaction, & proceed.
 - c. Send a message to the console room and just wait.

3. Verification & Testing

Addressing Client Misuse

Who should decide...

3. Verification & Testing

Addressing Client Misuse

Who should decide...

- 1. How much time** the library component should spend checking for preconditions?

3. Verification & Testing

Addressing Client Misuse

Who should decide...

- 1. How much time** the library component should spend checking for preconditions?
- 2. What happens** if preconditions are violated?

3. Verification & Testing

Addressing Client Misuse

Who should decide...

- 1. How much time** the library component should spend checking for preconditions?
- 2. What happens** if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?

3. Verification & Testing

Addressing Client Misuse

Who should decide...

1. **How much time** the library component should spend checking for preconditions?
2. **What happens** if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?
- b. The developer of the immediate client?

3. Verification & Testing

Addressing Client Misuse

Who should decide...

1. **How much time** the library component should spend checking for preconditions?
2. **What happens** if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?
- b. The developer of the immediate client?
- c. The owner of the application, who:

3. Verification & Testing

Addressing Client Misuse

Who should decide...

1. **How much time** the library component should spend checking for preconditions?
2. **What happens** if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?
- b. The developer of the immediate client?
- c. The owner of the application, who:
 - i. Is responsible for building the application.

3. Verification & Testing

Addressing Client Misuse

Who should decide...

1. **How much time** the library component should spend checking for preconditions?
2. **What happens** if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?
- b. The developer of the immediate client?
- c. The owner of the application, who:
 - i. Is responsible for building the application.
 - ii. Owns main .

3. Verification & Testing

Addressing Client Misuse

Who should decide...

1. **How much time** the library component should spend checking for preconditions?
2. **What happens** if preconditions are violated?

It should be:

- a. The (reusable) library component developer?
- b. The developer of the immediate client?
- c. **The owner of the application, who:**
 - i. Is responsible for building the application.
 - ii. Owns main .



3. Verification & Testing

Addressing Client Misuse

Who should decide...

1. **How much time** the library component should spend checking for preconditions?
2. **What happens** if preconditions are violated?

It should be:

- a. The (reusable) library component owner?
- b. The developer of the application.
- c. The owner of the application, who:
 - i. Is responsible for building the application.
 - ii. Owns main .

See the
bsls_assert
component.



3. Verification & Testing

Addressing Client Misuse

CPU Usage for Checking



Specified at Compile Time

Behavior if Misuse is Detected



Specified at Runtime

c. The owner of the application, who:

- i. Is responsible for building the application.
- ii. Owns main .



Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment (BDE)

Rendered as Fine-Grained Hierarchically Reusable Components.

Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

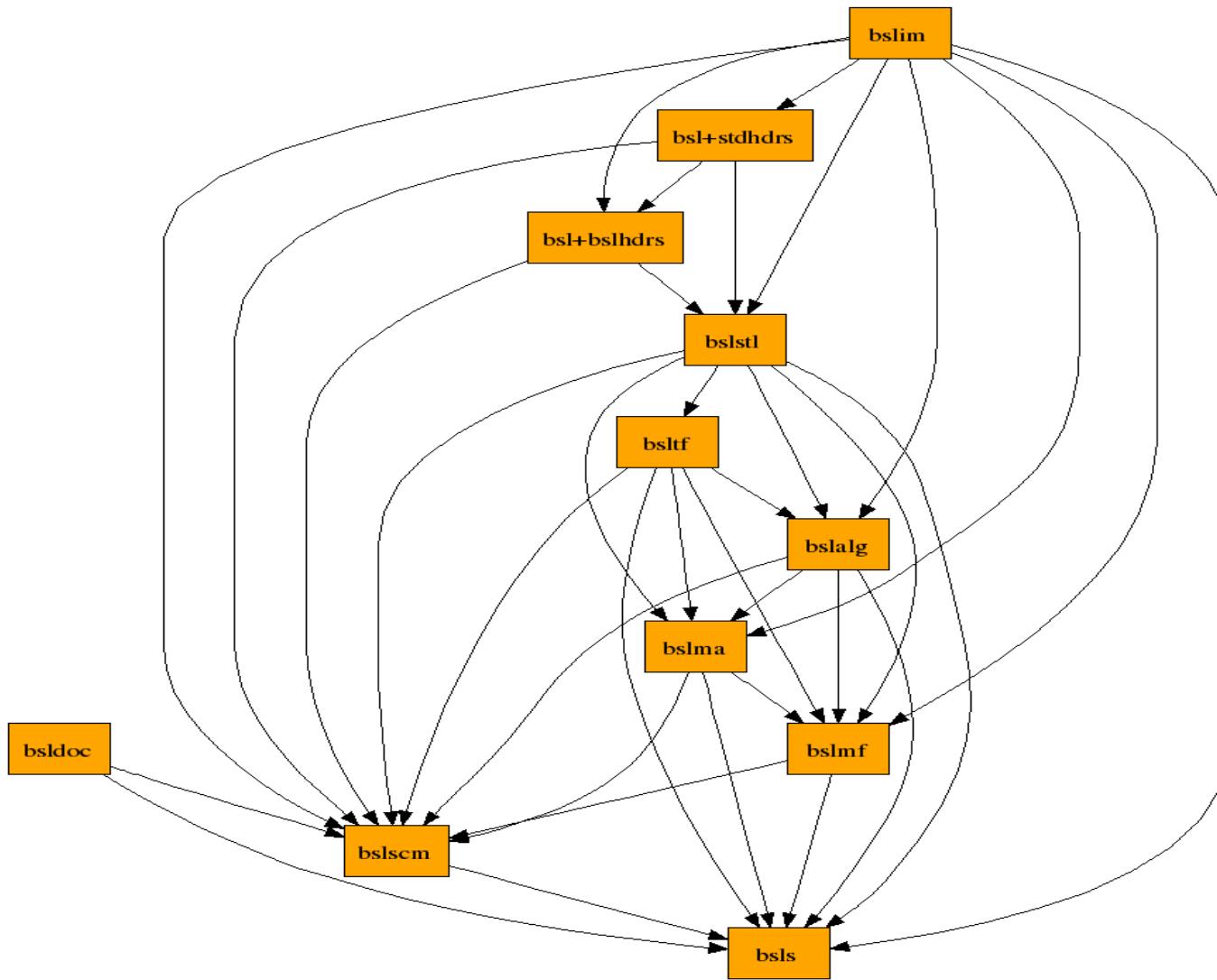
Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment (BDE)

Rendered as Fine-Grained Hierarchically Reusable Components.

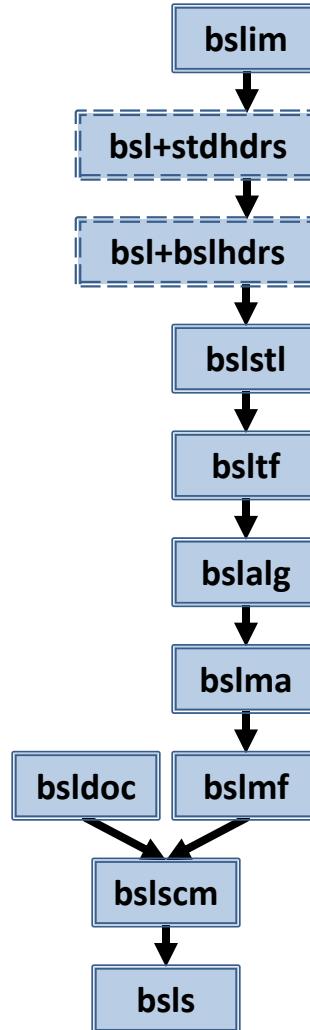
4. Bloomberg Development Environment

The BSL Package Group



4. Bloomberg Development Environment

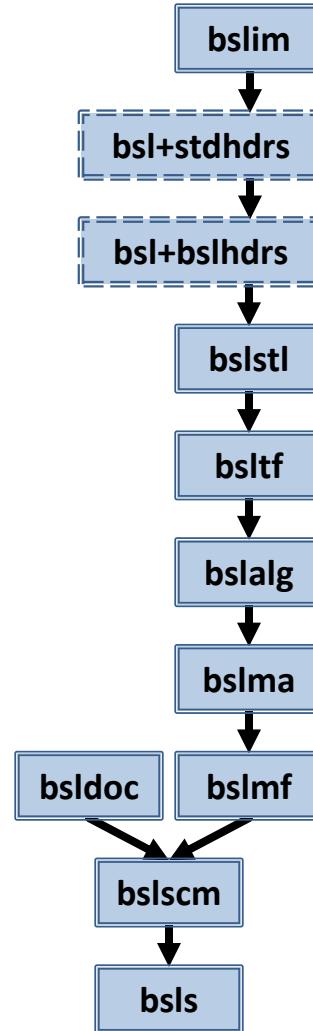
The BSL Package Group



4. Bloomberg Development Environment

The BSL Package Group

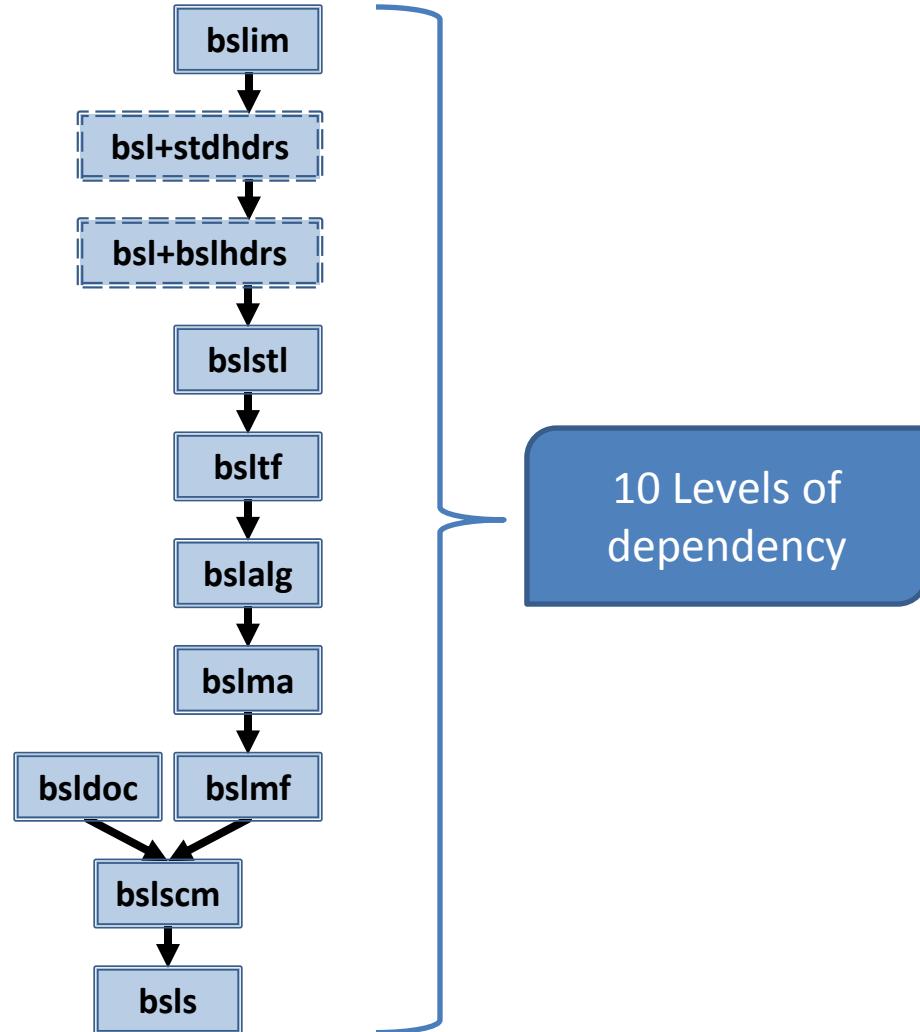
11 Packages



4. Bloomberg Development Environment

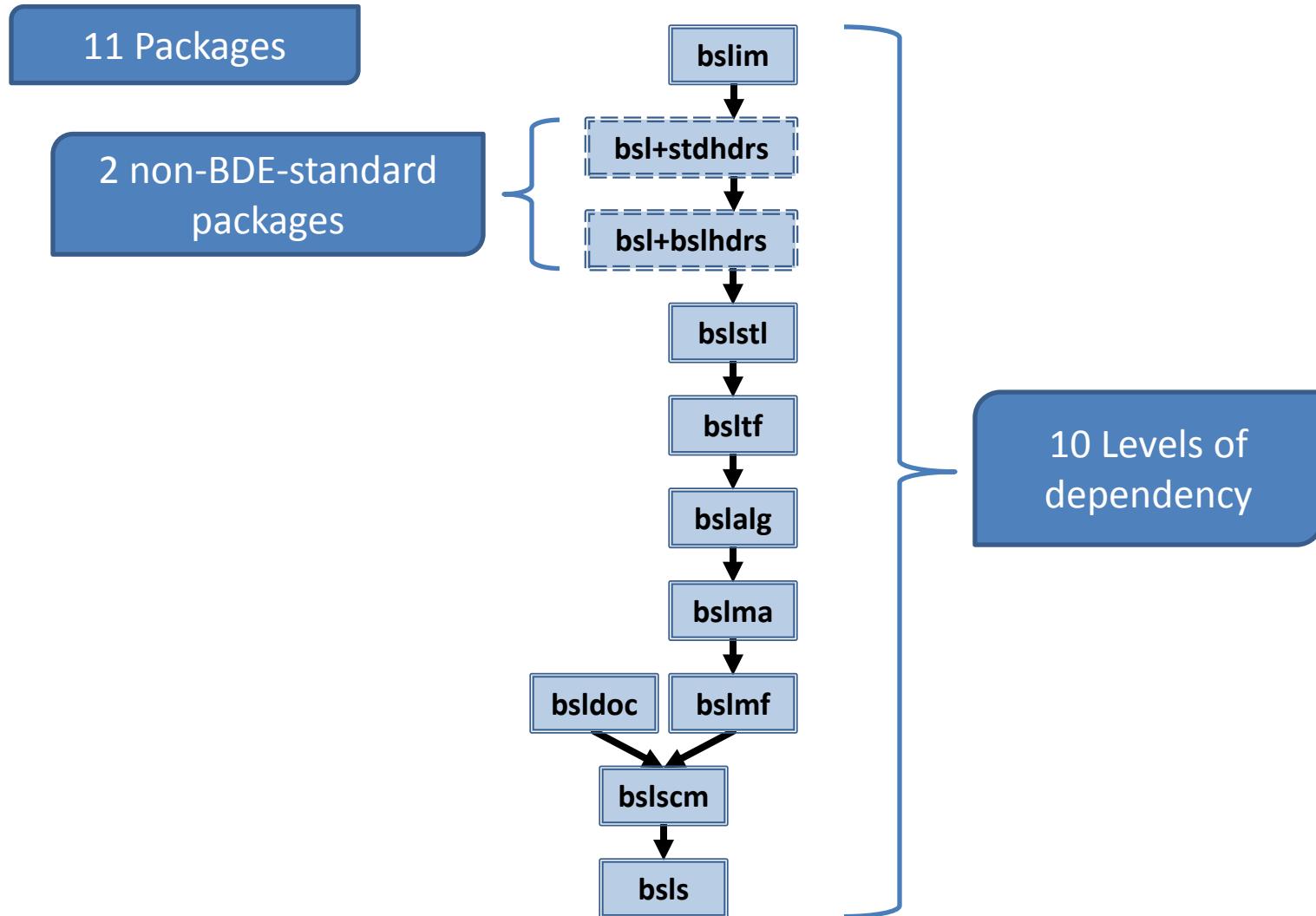
The BSL Package Group

11 Packages



4. Bloomberg Development Environment

The BSL Package Group



4. Bloomberg Development Environment

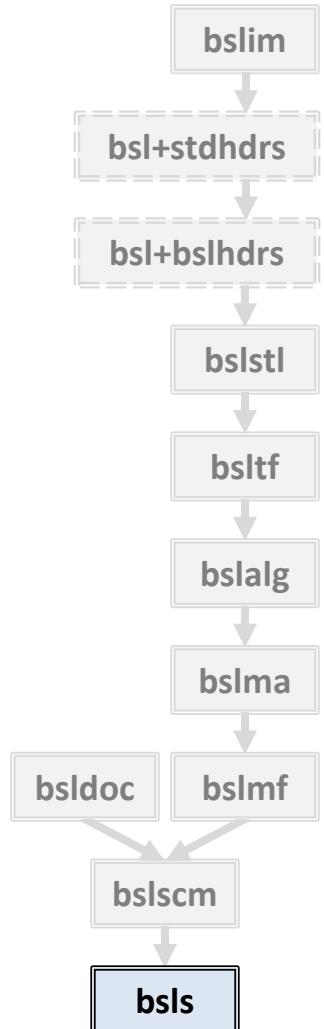
Package `bsls`



- System utilities

4. Bloomberg Development Environment

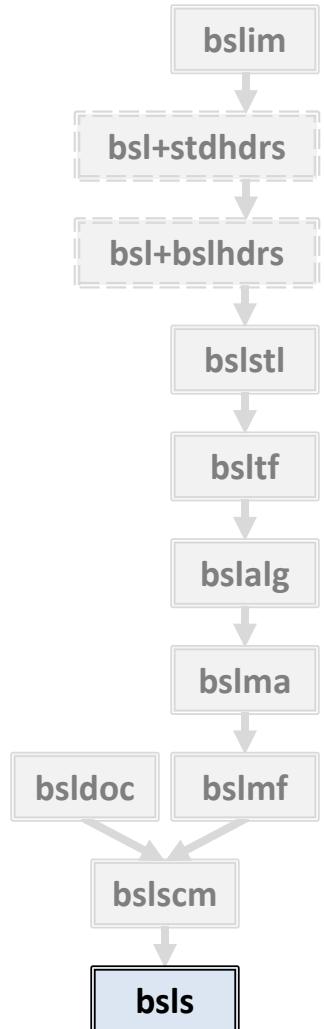
Package **bsls**



- System utilities
- Provides uniform handling of:
 - alignment, endian-ness, integer sizes, ...
 - clocks, atomic ops, and other system facilities

4. Bloomberg Development Environment

Package `bsls`



- System utilities
- Provides uniform handling of:
 - alignment, endian-ness, integer sizes, ...
 - clocks, atomic ops, and other system facilities
- Support for BDE methodology: e.g.,
 - **`bsls_bsltestutil`**
 - **`BSLS_ASSERT*`** macros

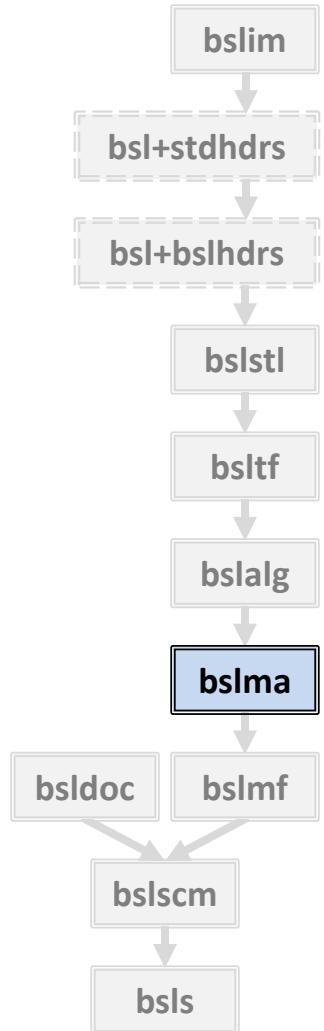
4. Bloomberg Development Environment

Package **bsls**

| | |
|---------------------------------|---------------------------------------------------------------------------------------|
| bsls_alignedbuffer | Provide raw buffers with user-specified size and alignment. |
| bsls_alignmentfromtype | Provide a meta-function that maps a TYPE to its alignment. |
| bsls_alignment | Provide a namespace for enumerating memory alignment strategies. |
| bsls_alignmentimp | Provide implementation meta-functions for alignment computation. |
| bsls_alignmenttotype | Provide a meta-function mapping an ALIGNMENT to a primitive type. |
| bsls_alignmentutil | Provide constants, types, and operations related to alignment. |
| bsls_annotation | Provide support for compiler annotations for compile-time safety. |
| bsls_assert | Provide build-specific, runtime-configurable assertion macros. |
| bsls_asserttestexception | Provide an exception type to support testing for failed assertions. |
| bsls_asserttest | Provide a test facility for assertion macros. |
| bsls_atomic | Provide types with atomic operations. |
| bsls_atomicoperations | Provide platform-independent atomic operations. |
| bsls_blockgrowth | Provide a namespace for memory block growth strategies. |
| bsls_bsltestutil | Provide test utilities for bsl that do not use <code><iostream></code> . |
| bsls_buildtarget | Provide build-target information in the object file. |
| bsls_byteorder | Provide byte-order manipulation macros. |
| bsls_compilerfeatures | Provide macros to identify compiler support for C++11 features. |
| bsls_exceptionutil | Provide simplified exception constructs for non-exception builds. |
| bsls_ident | Provide macros for inserting SCM Ids into source files. |
| bsls_macroincrement | Provide a macro to increment preprocessor numbers. |
| bsls_nativestd | Define the namespace native_std as an alias for <code>::std</code> . |
| bsls_nullptr | Provide a distinct type for null pointer literals. |
| bsls_objectbuffer | Provide raw buffer with size and alignment of user-specified type. |
| bsls_performancehint | Provide performance hints for code optimization. |
| bsls_platform | Provide compile-time support for platform/attribute identification. |
| bsls_protocoltest | Provide classes and macros for testing abstract protocols. |
| bsls_stopwatch | Provide access to user, system, and wall times of current process. |
| bsls_timeutil | Provide a platform-neutral functional interface to system clocks. |
| bsls_types | Provide a consistent interface for platform-dependent types. |
| bsls_unspecifiedbool | Provide a class supporting the "unspecified bool" idiom. |
| bsls_util | Provide essential, low-level support for portable generic code. |

4. Bloomberg Development Environment

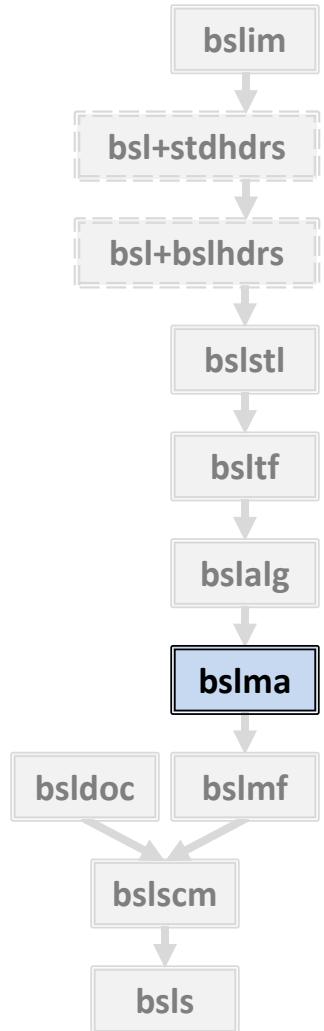
Package **bslma**



- Memory Allocators

4. Bloomberg Development Environment

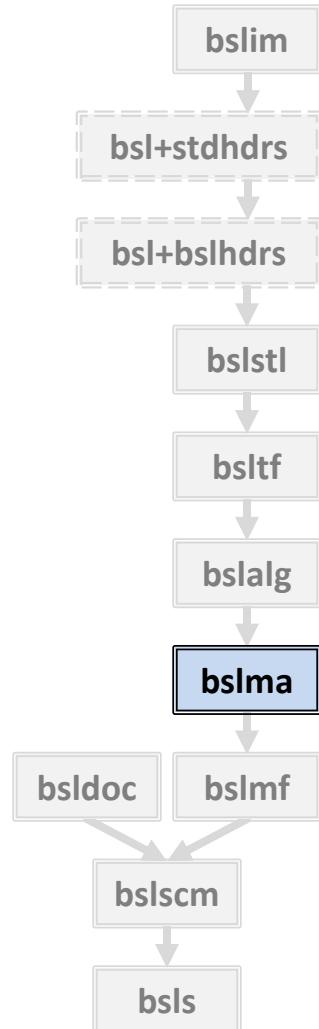
Package **bslma**



- Memory Allocators
- Allocator protocol:
bslma_allocator

4. Bloomberg Development Environment

Package **bslma**

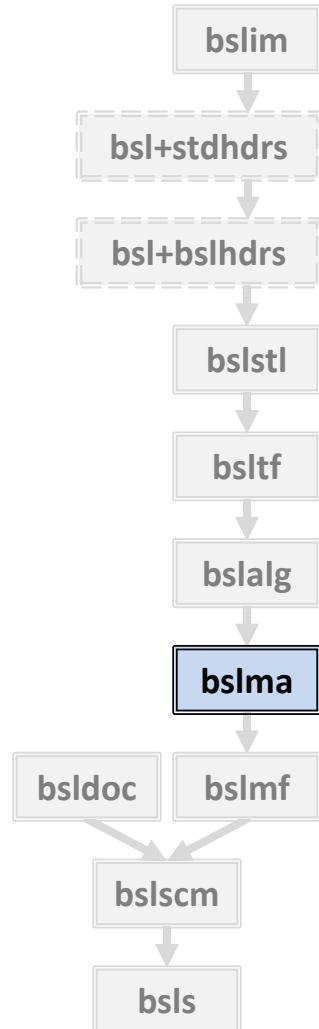


- Memory Allocators
- Allocator protocol: `bslma_allocator`

Quintessential
Vocabulary Type

4. Bloomberg Development Environment

Package **bslma**

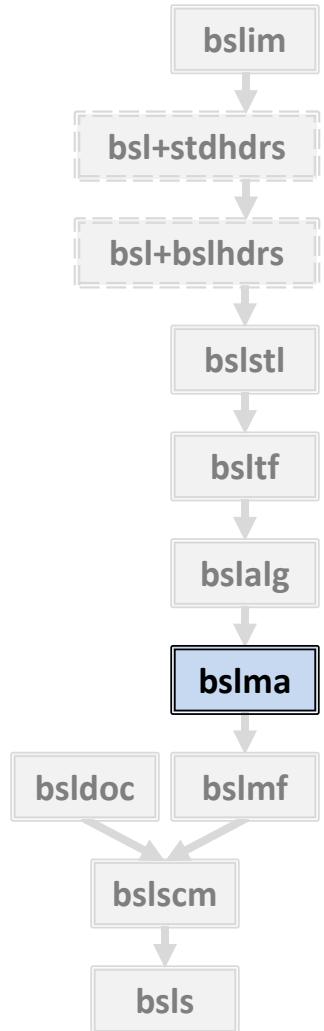


- Memory Allocators
- Allocator protocol:
- Mechanisms

Quintessential
Vocabulary Type

4. Bloomberg Development Environment

Package **bslma**

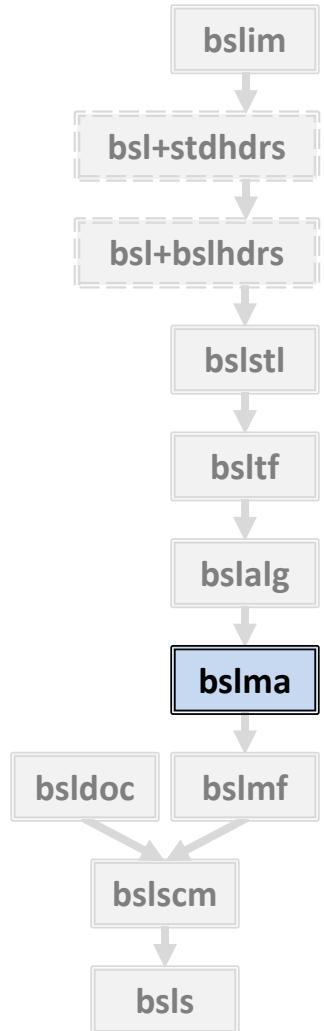


- Memory Allocators
- Allocator protocol:
bslma_allocator
- Mechanisms
 - The default default-allocator,
bslma_newdeleteallocator

Quintessential
Vocabulary Type

4. Bloomberg Development Environment

Package **bslma**

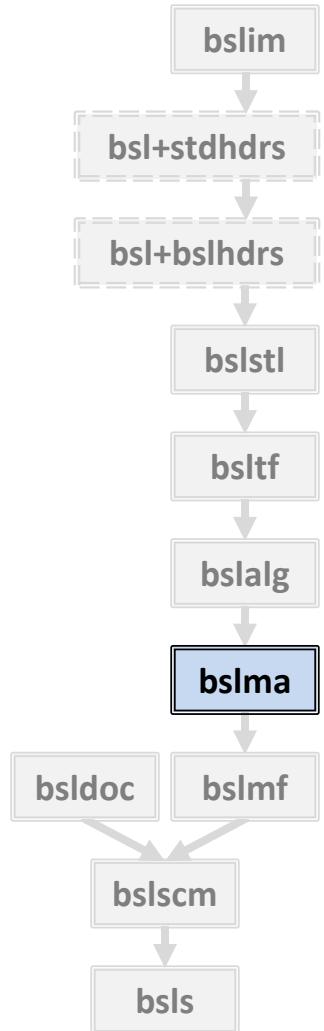


- Memory Allocators
- Allocator protocol:
bslma_allocator
- Mechanisms
 - The default default-allocator,
bslma_newdeleteallocator
 - Managing the default, **bslma_default**

Quintessential
Vocabulary Type

4. Bloomberg Development Environment

Package **bslma**

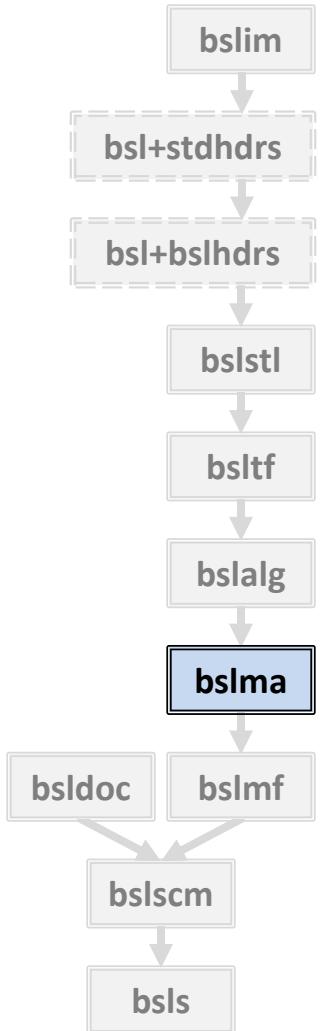


- Memory Allocators
- Allocator protocol: **bslma_allocator**
- Mechanisms
 - The default default-allocator, **bslma_newdeleteallocator**
 - Managing the default, **bslma_default**
 - Development, **bslma_testallocator**

Quintessential
Vocabulary Type

4. Bloomberg Development Environment

Package **bslma**

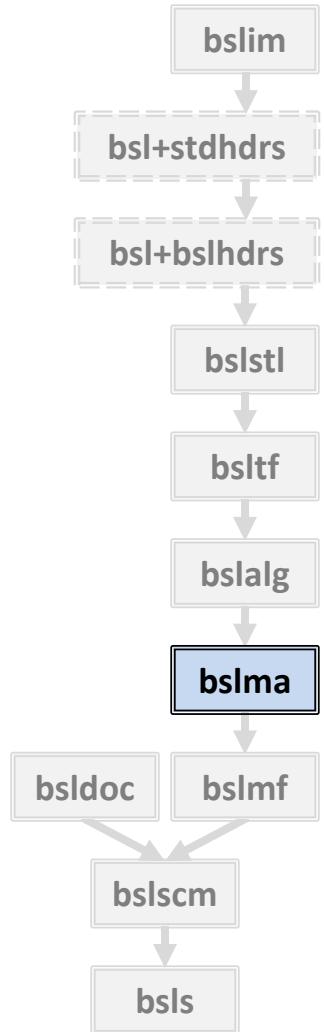


- Memory Allocators
- Allocator protocol:
bslma_allocator
- Mechanisms
 - The default default-allocator,
bslma_newdeleteallocator
 - Managing the default, **bslma_default**
 - Development, **bslma_testallocator**
- Guards and proctors for single objects and ranges

Quintessential
Vocabulary Type

4. Bloomberg Development Environment

Package **bslma**



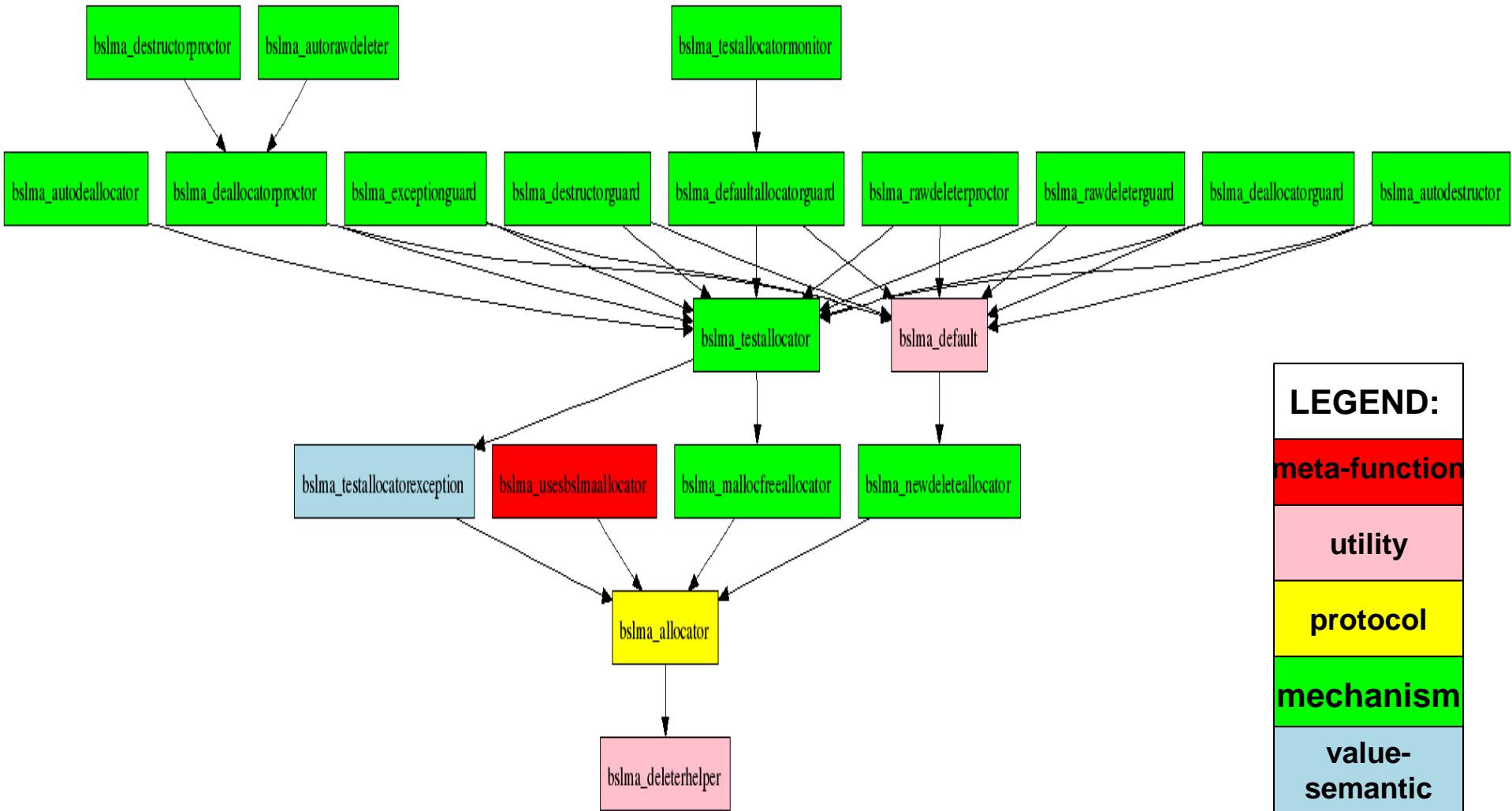
- Memory Allocators
- Allocator protocol:
bslma_allocator
- Mechanisms
 - The default default-allocator,
bslma_newdeleteallocator
 - Managing the default, **bslma_default**
 - Development, **bslma_testallocator**
- Guards and proctors for single objects and ranges

Quintessential Vocabulary Type

Have **release** method

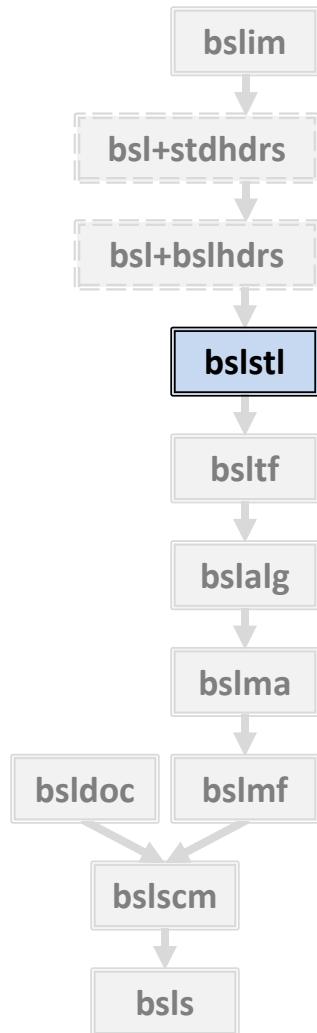
4. Bloomberg Development Environment

Package **bslma**



4. Bloomberg Development Environment

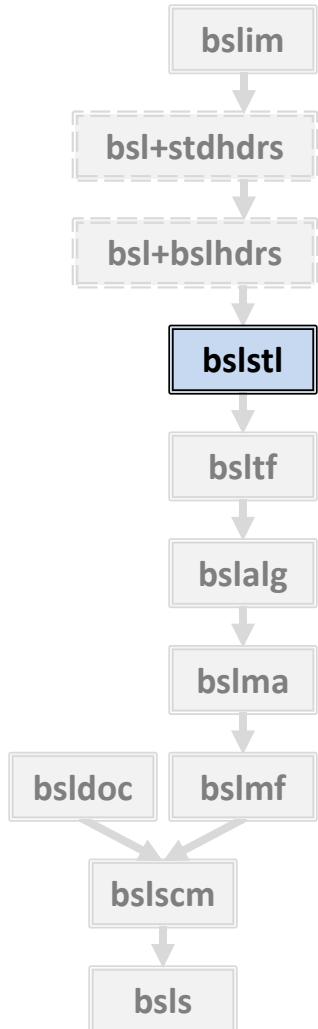
Package **bslstl**



- **STL**

4. Bloomberg Development Environment

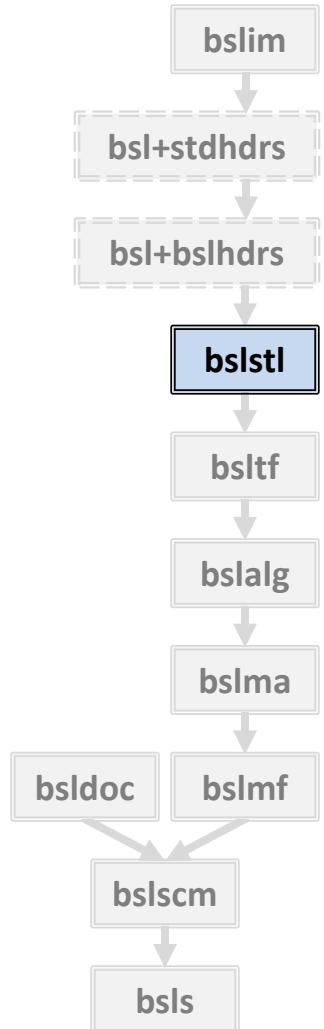
Package **bslstl**



- **STL**
- C++ Standard library from BDE allows
 - Standard allocators, *and*
 - BDE runtime polymorphic allocators for allocator-aware types (e.g., **vector**, **list**, **unordered_map**)

4. Bloomberg Development Environment

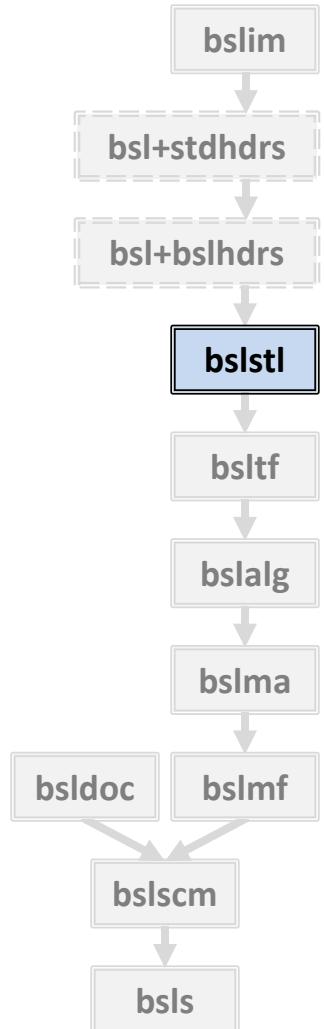
Package `bslstl`



- **STL**
- C++ Standard library from BDE allows
 - Standard allocators, *and*
 - BDE runtime polymorphic allocators for allocator-aware types (e.g., `vector`, `list`, `unordered_map`)
- Non-allocator facilities pass through to native library

4. Bloomberg Development Environment

Package **bslstl**



- **STL**
- C++ Standard library from BDE allows
 - Standard allocators, *and*
 - BDE runtime polymorphic allocators for allocator-aware types (e.g., **vector**, **list**, **unordered_map**)
- Non-allocator facilities pass through to native library
- Used via **bsl+bslhdtrs** (not directly)

4. Bloomberg Development Environment

Our Open Source Distribution

How do you find what you need?

4. Bloomberg Development Environment

Our Open Source Distribution

How do you find what you need?

- BDE group, package, and component-level doc converted to **doxygen** markup.

4. Bloomberg Development Environment

Our Open Source Distribution

How do you find what you need?

- BDE group, package, and component-level doc converted to **doxygen** markup.
- Hierarchically organized home page provides overview of all components.

4. Bloomberg Development Environment

Our Open Source Distribution

Collapse
All
Groups

Expand
All
Packages

| | Group | Package | Component | Purpose |
|---|--------------|------------------------------|------------------|--------------------------------------------------------------------|
| - | bsl | | | Provide a comprehensive foundation for component-based development |
| - | | bsl+bslhdrs | | Provide a compatibility layer to enable BDE -STL mode in Bloomberg |
| - | | bsl+stdhdrs | | Provide a compatibility layer to enable BDE -STL mode in Bloomberg |
| + | | bslalg | | Provide algorithms and traits used by the BDE STL implementation |
| - | | bsldoc | | Provide documentation of terms and concepts used throughout BDE |
| | | bsldoc_glossary | | Provide definitions for terms used throughout BDE documentation |
| - | bslim | | | Provide implementation mechanisms |
| | | bslim_printer | | Provide a mechanism to implement standard print methods |
| - | bslma | | | Provide allocators, guards, and other memory-management tools |
| | | bslma_allocator | | Provide a pure abstract interface for memory-allocation mechanisms |
| | | bslma_autodeallocator | | Provide a range proctor to manage a block of memory |
| | | bslma_block_allocator | | Provide a range proctor to manage an |

4. Bloomberg Development Environment

Our Open Source Distribution

- What License Applies?

4. Bloomberg Development Environment

Our Open Source Distribution

- What License Applies?
 - Software License: MIT

4. Bloomberg Development Environment

Our Open Source Distribution

- What License Applies?
 - Software License: MIT
 - Deliberately liberal

4. Bloomberg Development Environment

Our Open Source Distribution

- What License Applies?
 - Software License: MIT
 - Deliberately liberal
 - We intend that anyone can use our software freely

4. Bloomberg Development Environment

Our Open Source Distribution

- What License Applies?
 - Software License: MIT
 - Deliberately liberal
 - We intend that anyone can use our software freely
 - for any legitimate purpose

4. Bloomberg Development Environment

Our Open Source Distribution

- What License Applies?
 - Software License: MIT
 - Deliberately liberal
 - We intend that anyone can use our software freely
 - for any legitimate purpose
 - including as part of a product for sale

4. Bloomberg Development Environment

Our Open Source Distribution

- Find our open-source distribution at:
<http://www.openbloomberg.com/bsl>
- Moderator: kpfleming@bloomberg.net
- How to contribute? *See our site.*
- All comments and criticisms welcome...

4. Bloomberg Development Environment

Our Open Source Distribution

- Find our open-source distribution at:
<http://www.openbloomberg.com/bsl>
- Moderator: kpfleming@bloomberg.net
- How to contribute? *See our site.*
- All comments and criticisms welcome...

We will come back to this...

4. Bloomberg Development Environment

Moving Upward and Onward

Beyond BS:

4. Bloomberg Development Environment

Moving Upward and Onward

Beyond BSF:

- The Allocator protocol is defined in **bslma**

4. Bloomberg Development Environment

Moving Upward and Onward

Beyond BSL:

- The Allocator protocol is defined in `bslma`
- Most concrete allocators reside above `bsl`

4. Bloomberg Development Environment

Moving Upward and Onward

Beyond BSL:

- The Allocator protocol is defined in `bslma`
- Most concrete allocators reside above `bsl`
- Some will be in `bdlma` (when released)

4. Bloomberg Development Environment

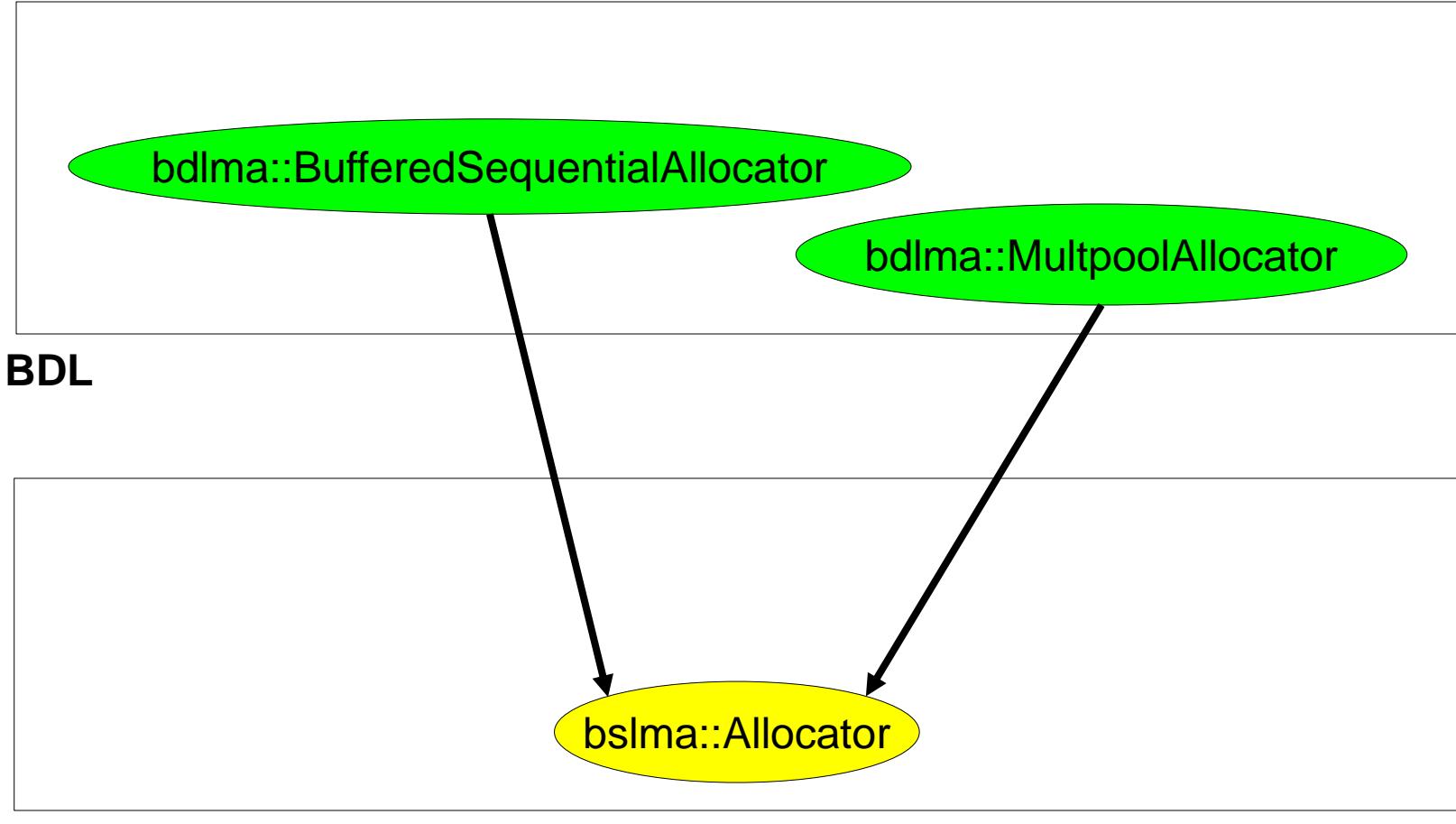
Moving Upward and Onward

Beyond BSL:

- The Allocator protocol is defined in `bslma`
- Most concrete allocators reside above `bsl`
- Some will be in `bdlma` (when released)
- Examples:
 - **Buffered Sequential Allocator**
 - **Multipool Allocator**

4. Bloomberg Development Environment

Moving Upward and Onward



BSL

4. Bloomberg Development Environment

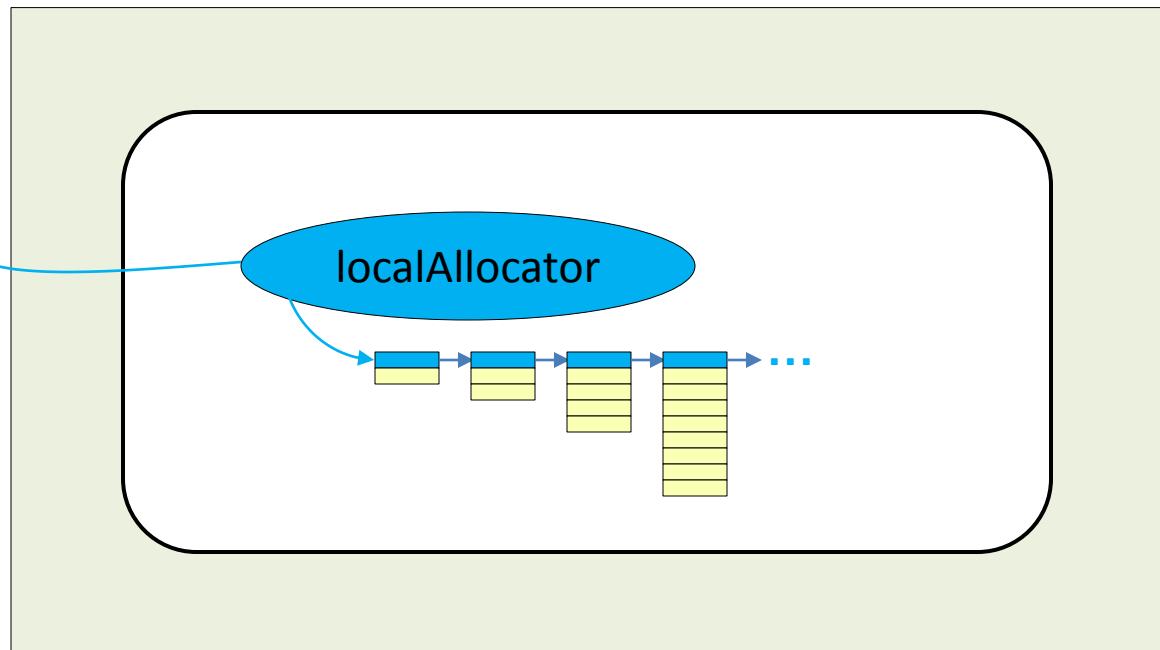
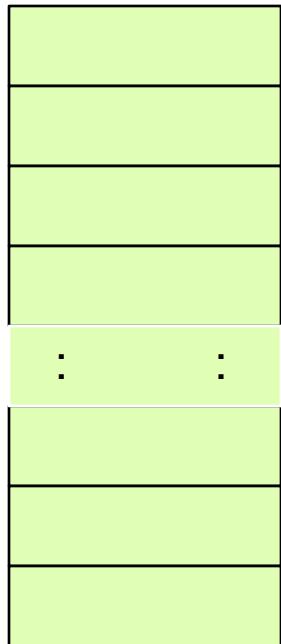
Buffered Sequential Allocator



4. Bloomberg Development Environment

Buffered Sequential Allocator

```
void myFunction(...) {  
    char buffer[1024];
```

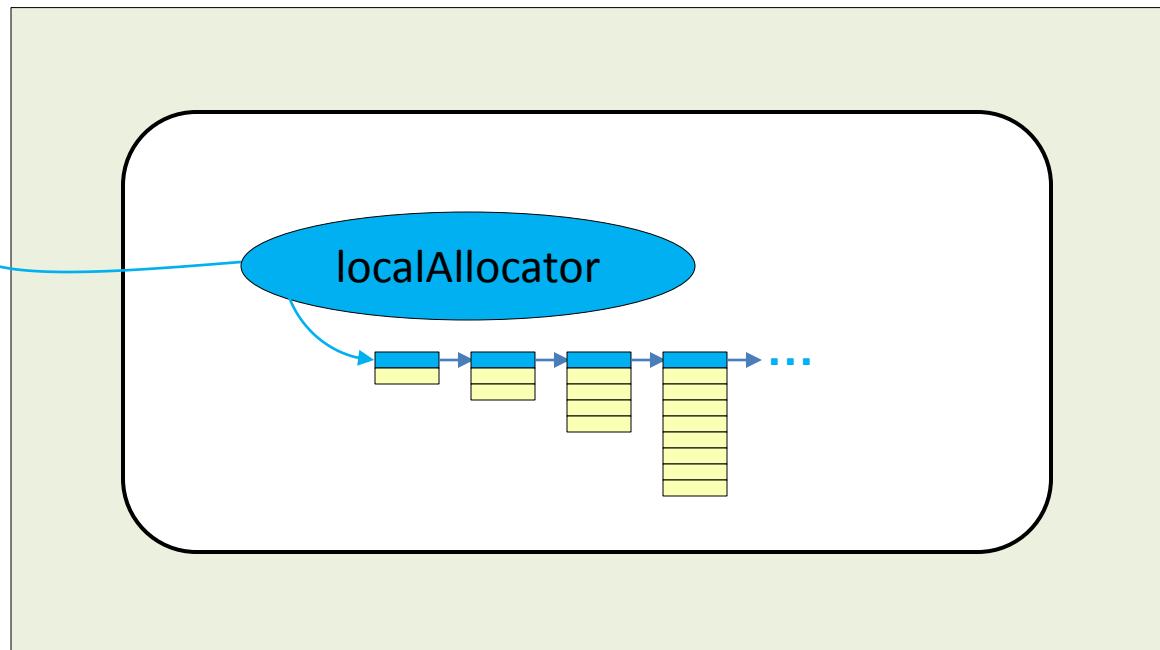
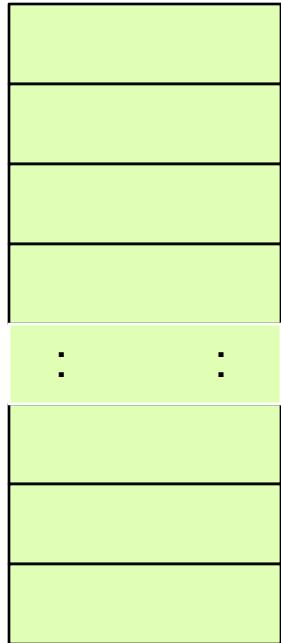


```
bdlma::BufferedSequentialAllocator local Allocator(buffer, sizeof buffer);  
bsl::vector(&local Allocator);  
// ...  
}
```

4. Bloomberg Development Environment

Buffered Sequential Allocator

```
void myFunction(...) {  
    char buffer[1024];
```



```
bdlma::BufferedSequentialAllocator local Allocator(buffer, sizeof buffer);  
bsl::vector(&local Allocator);  
// ...
```

Note that deallocate is a No-Op!

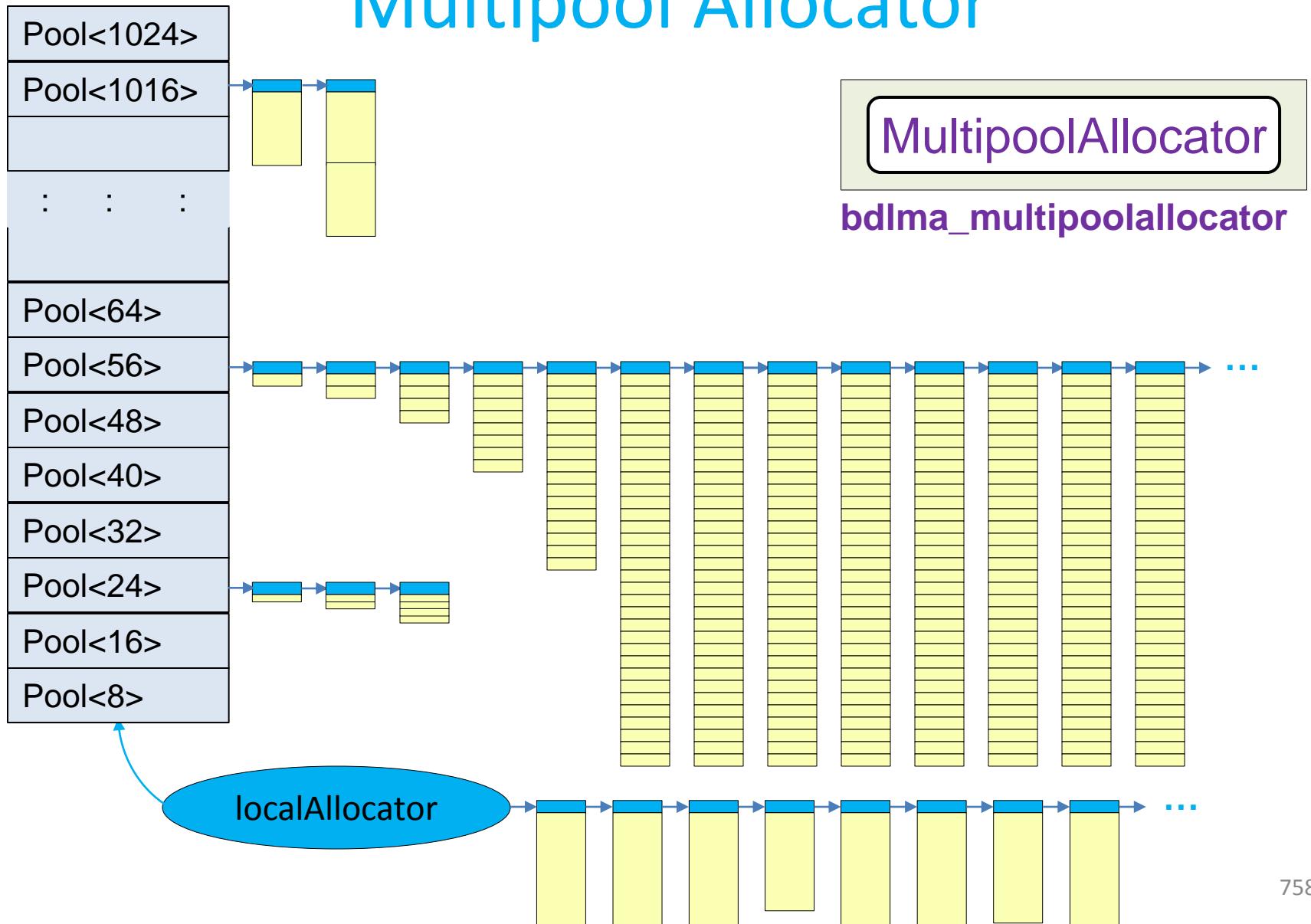
4. Bloomberg Development Environment

Multipool Allocator



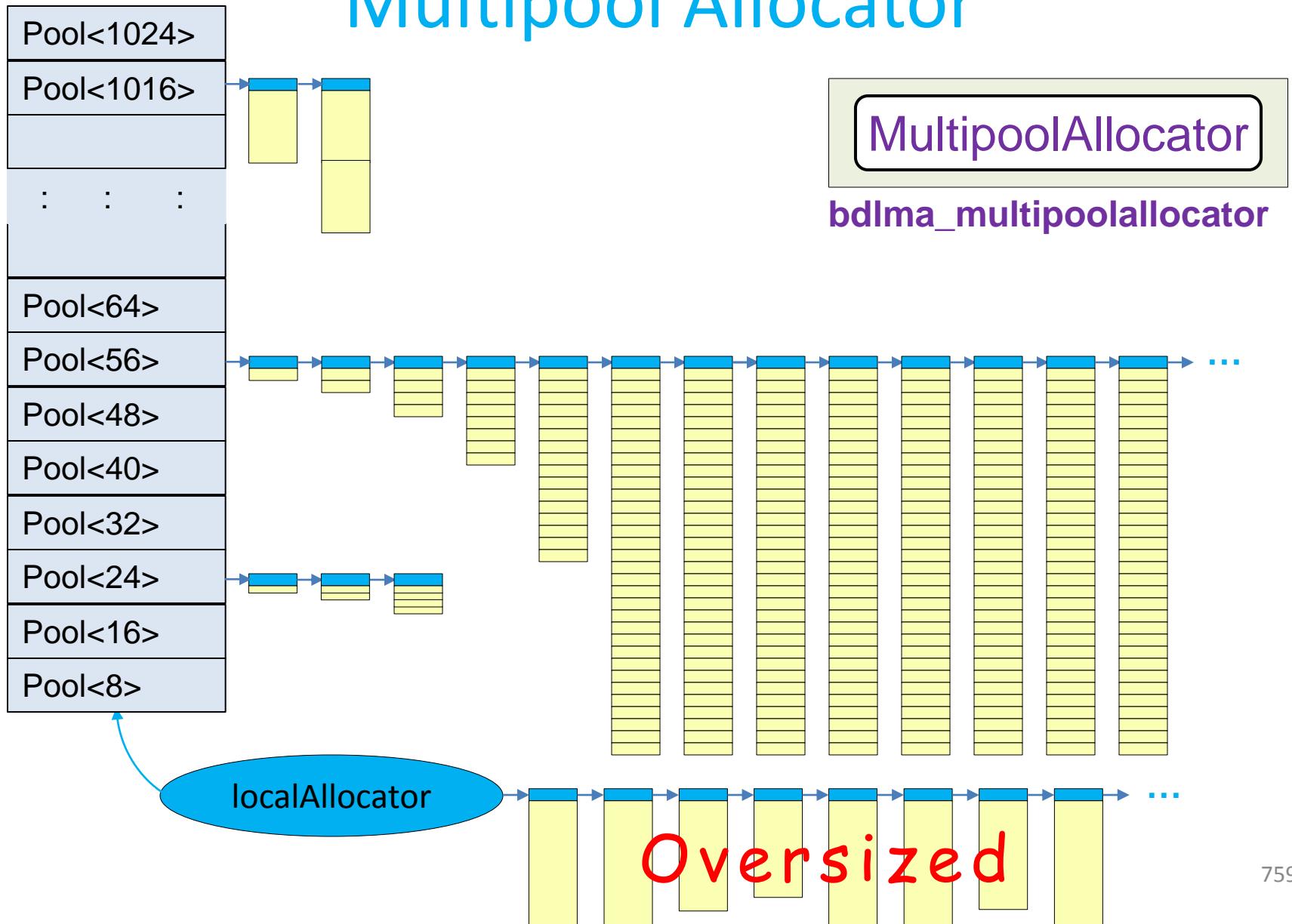
4. Bloomberg Development Environment

Multipool Allocator



4. Bloomberg Development Environment

Multipool Allocator



4. Bloomberg Development Environment

A Business Request

Suppose you are asked to provide some business functionality:

"Write me a 'Date' class that tells me whether today is a business day."

4. Bloomberg Development Environment

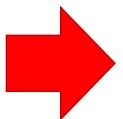
What's the Problem?

"Write me a 'Date' class that tells me whether **today** is a **business day**."

4. Bloomberg Development Environment

What's the Problem?

"Write me a 'Date' class that tells me whether **today** is a **business day**."

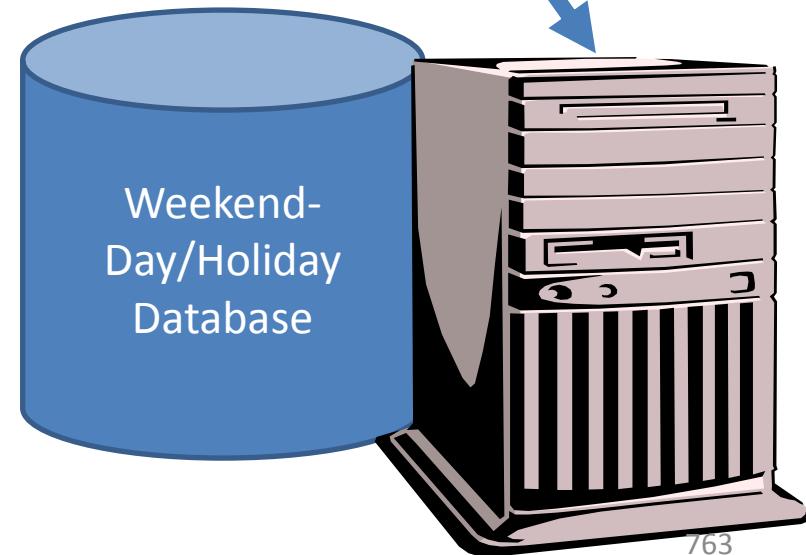
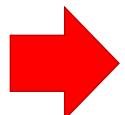


Date

4. Bloomberg Development Environment

What's the Problem?

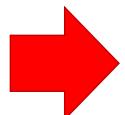
"Write me a 'Date' class that tells me whether **today** is a **business day**."



4. Bloomberg Development Environment

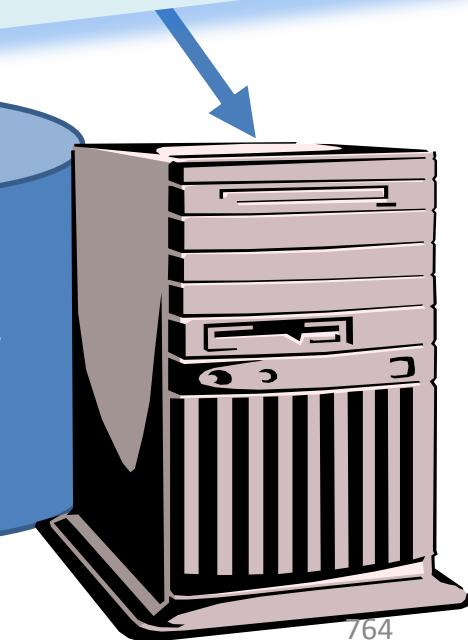
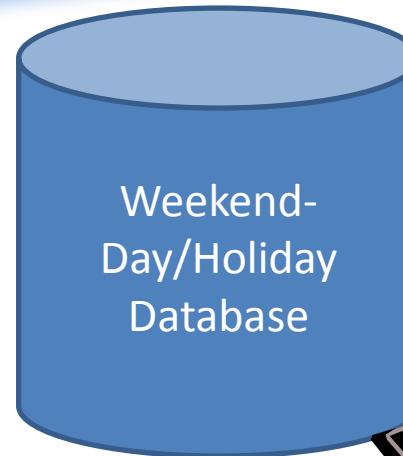
What's the Problem?

"Write me a 'Date' class that tells me whether **today** is a business day."



Date

Poor Logical Factoring



4. Bloomberg Development Environment

What's the Problem?

"Write me a 'Date' class that tells whether today is a business day."

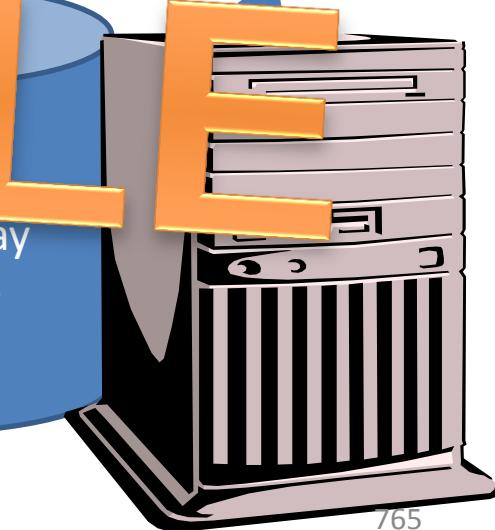
NOT

PFor Logical Factoring

FLEXIBLE



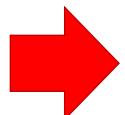
Wacker
Day/Holiday
Database



4. Bloomberg Development Environment

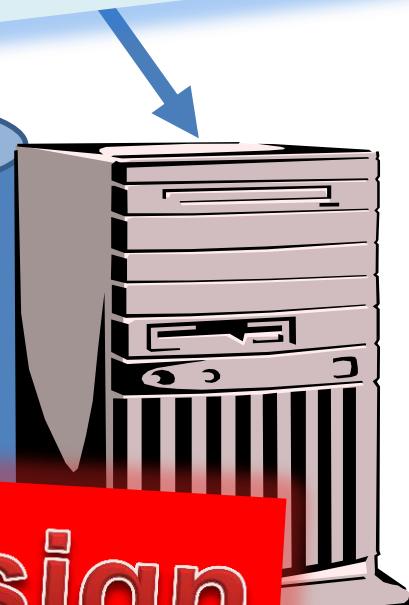
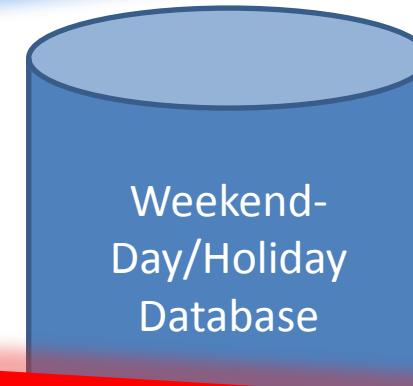
What's the Problem?

"Write me a 'Date' class that tells me whether **today** is a business day."



Date

Poor Logical Factoring



Poor Physical Design

4. Bloomberg Development Environment

What's the Problem?

"Write me a 'Date' class that tells me whether today is a business day."

NOT

Date

Poor Logical Factoring
MAINTAINABLE

Weekend-
Day/Holiday
Database

Poor Physical Design

4. Bloomberg Development Environment

The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

4. Bloomberg Development Environment

The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.

4. Bloomberg Development Environment

The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. Determine what date value *today* is.

4. Bloomberg Development Environment

The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. Determine what date value *today* is.
3. Determine if a date value is a *business day*.

4. Bloomberg Development Environment

The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. Determine what date value *today* is.
3. Determine if a date value is a *business day*.
4. **Provide well-factored useful components that we'll need over and over again!**

4. Bloomberg Development Environment

The Original Request

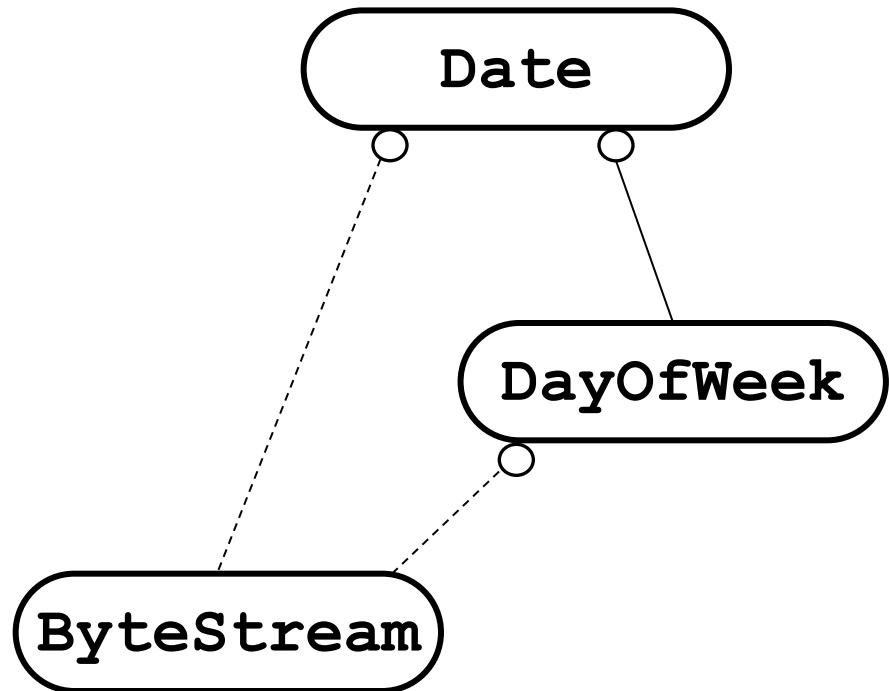
"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. **Represent a *date value* as a C++ Type.**
2. Determine what date value *today* is.
3. Determine if a date value is a *business day*.
4. **Provide well-factored useful components that we'll need over and over again!**

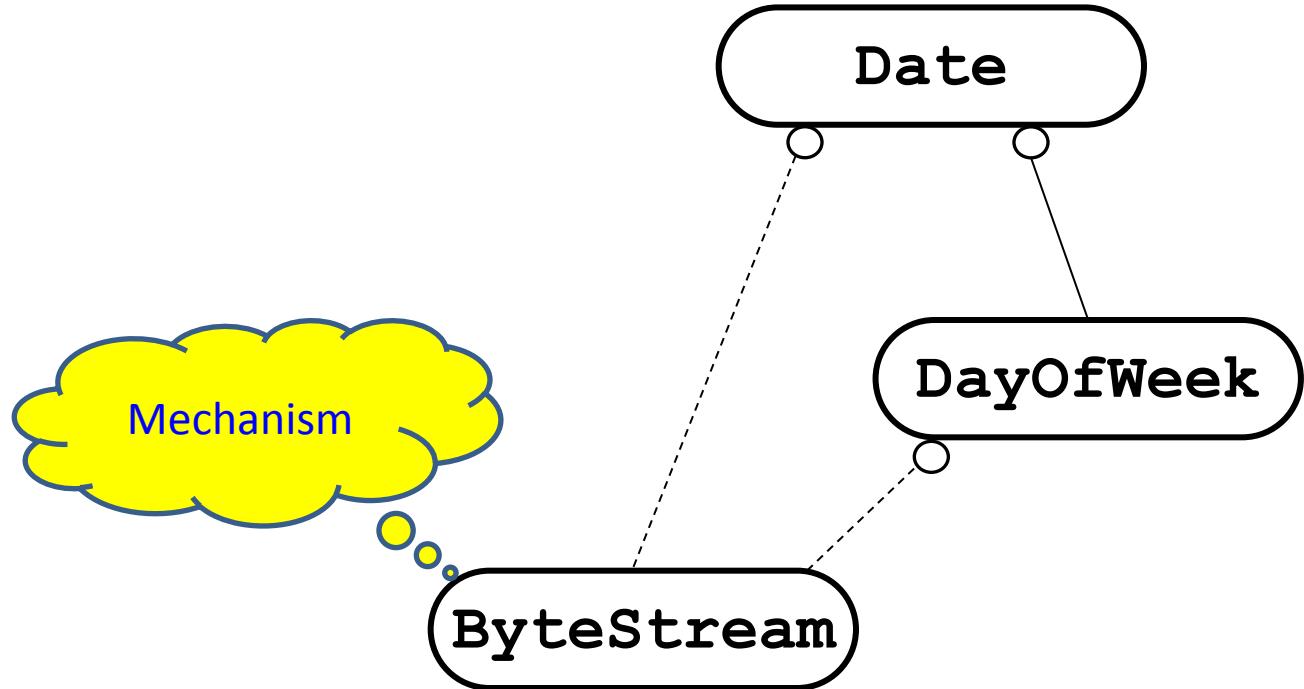
4. Bloomberg Development Environment

Represent a *Date Value* as a C++ Type



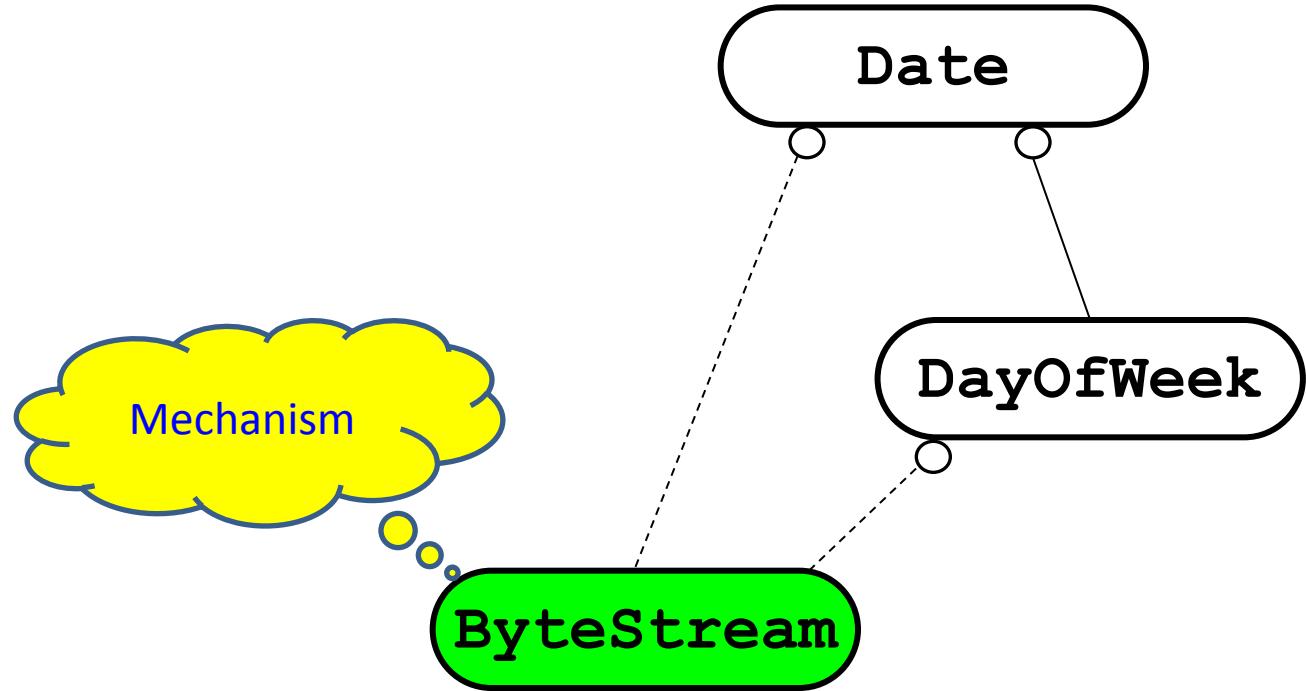
4. Bloomberg Development Environment

Represent a *Date Value* as a C++ Type



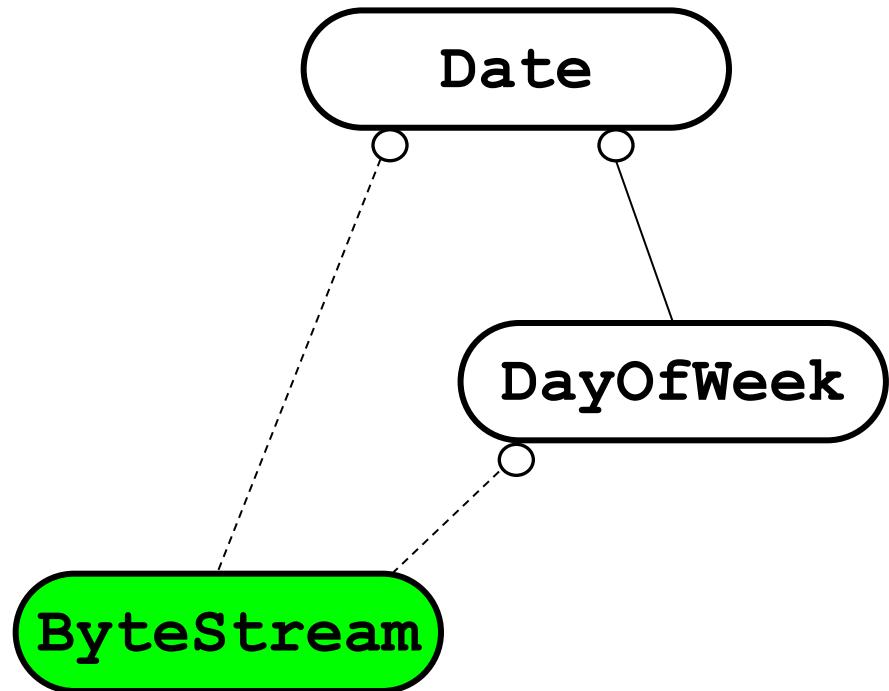
4. Bloomberg Development Environment

Represent a *Date Value* as a C++ Type



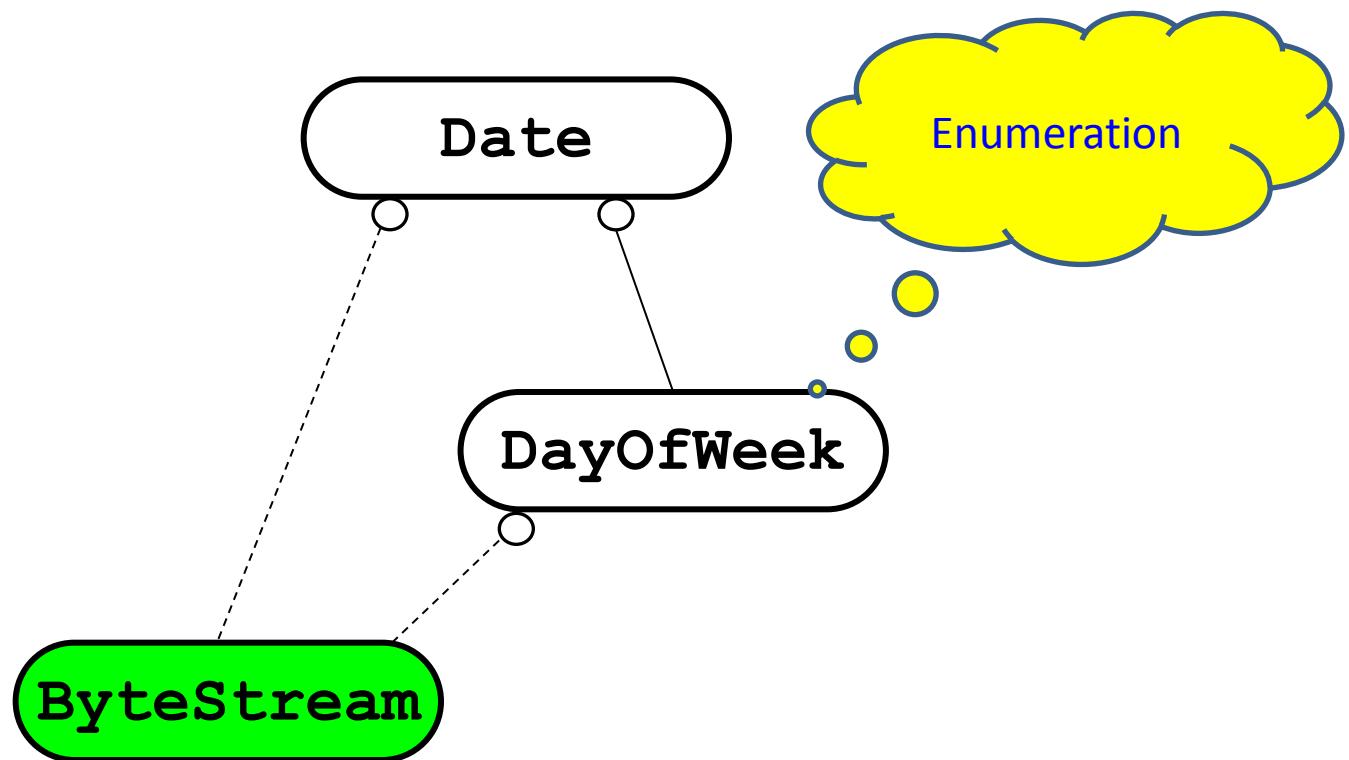
4. Bloomberg Development Environment

Represent a *Date Value* as a C++ Type



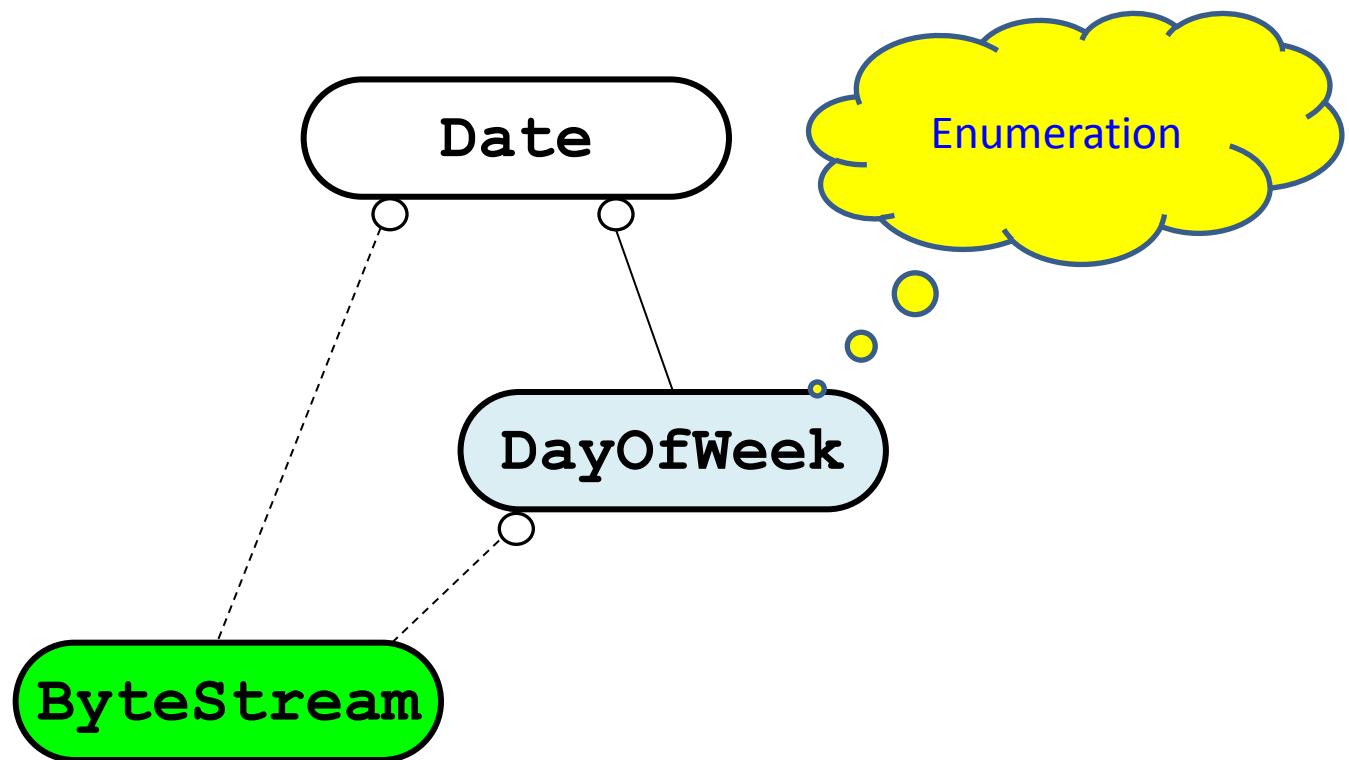
4. Bloomberg Development Environment

Represent a *Date Value* as a C++ Type



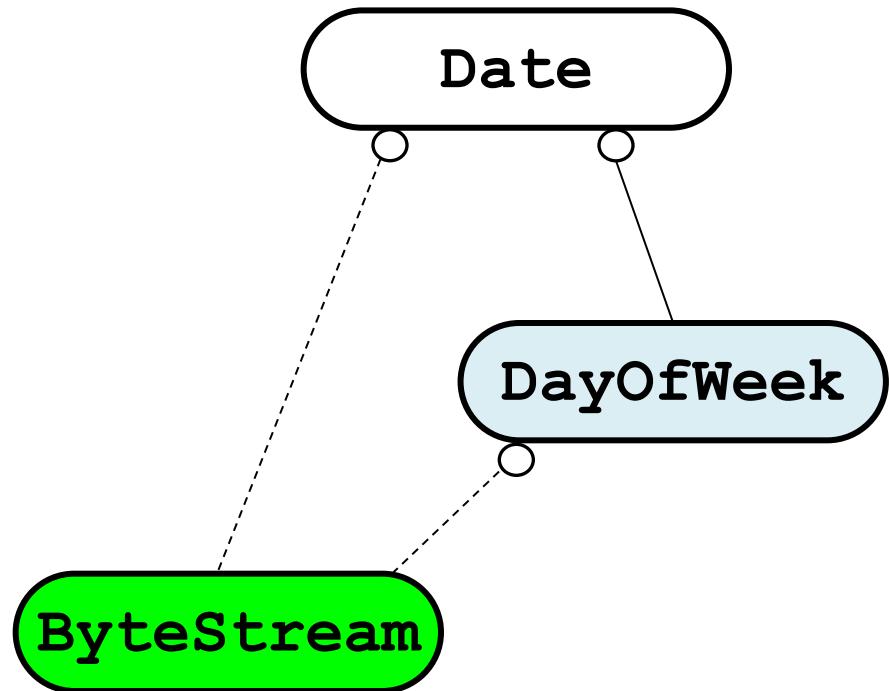
4. Bloomberg Development Environment

Represent a *Date Value* as a C++ Type



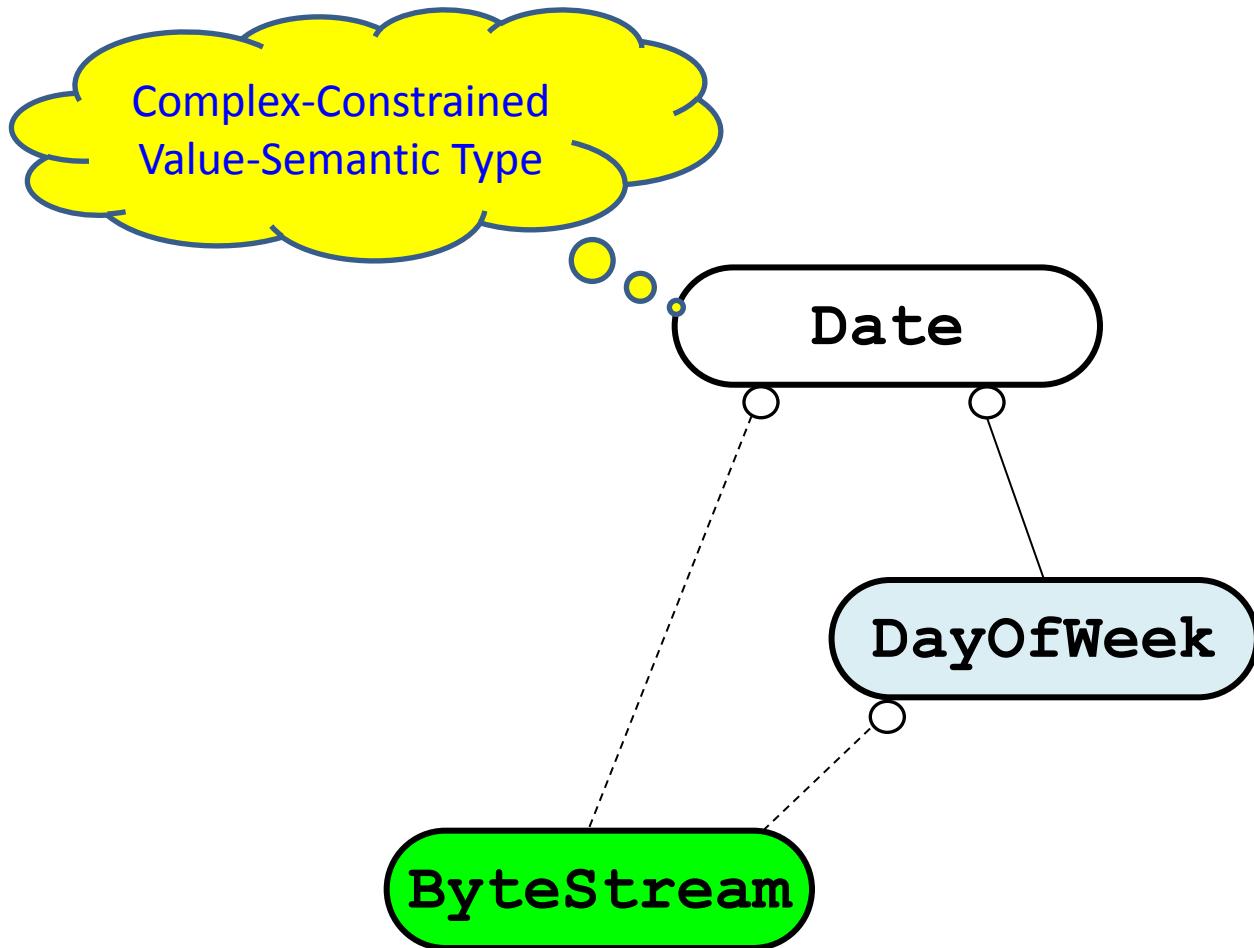
4. Bloomberg Development Environment

Represent a *Date Value* as a C++ Type



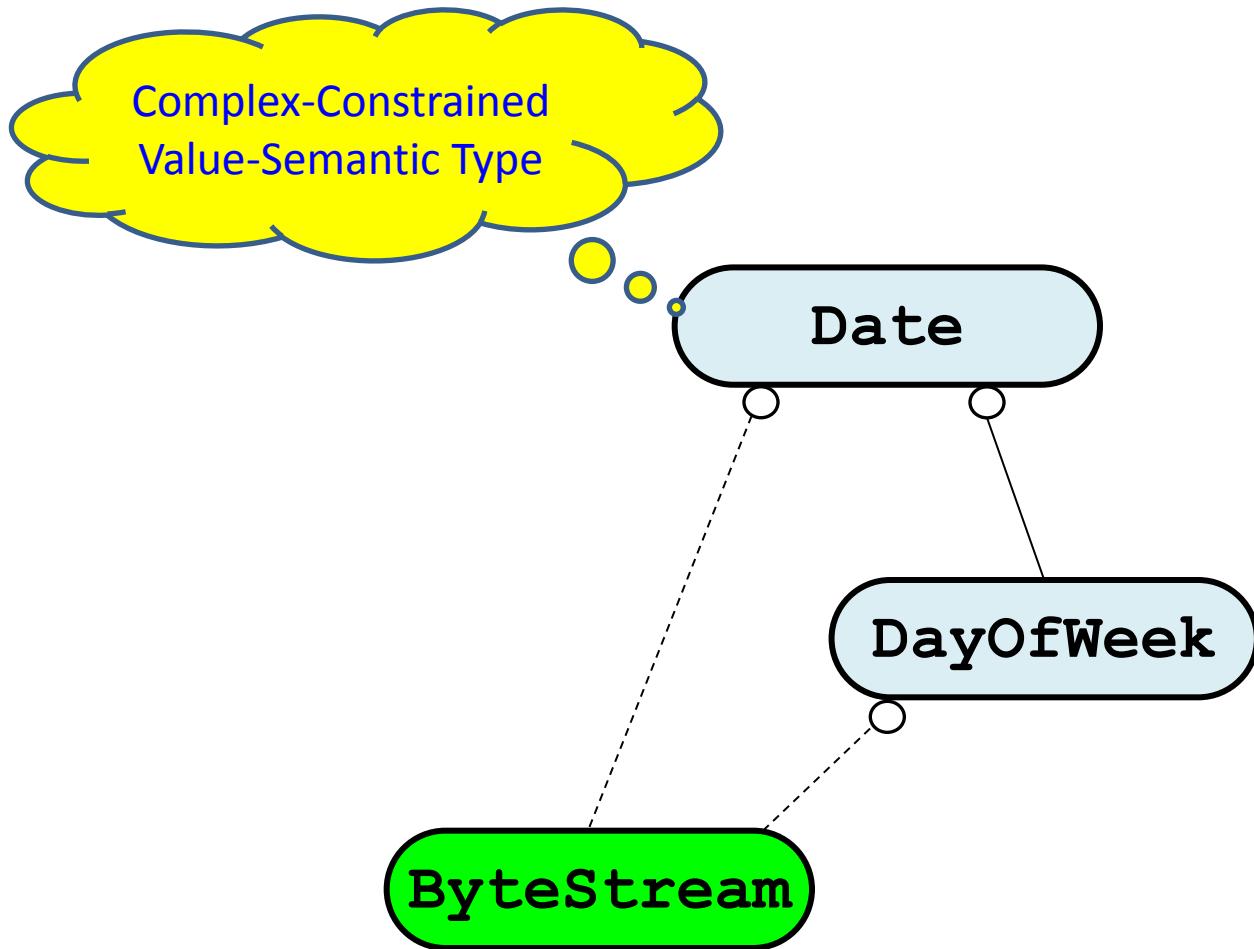
4. Bloomberg Development Environment

Represent a *Date Value* as a C++ Type



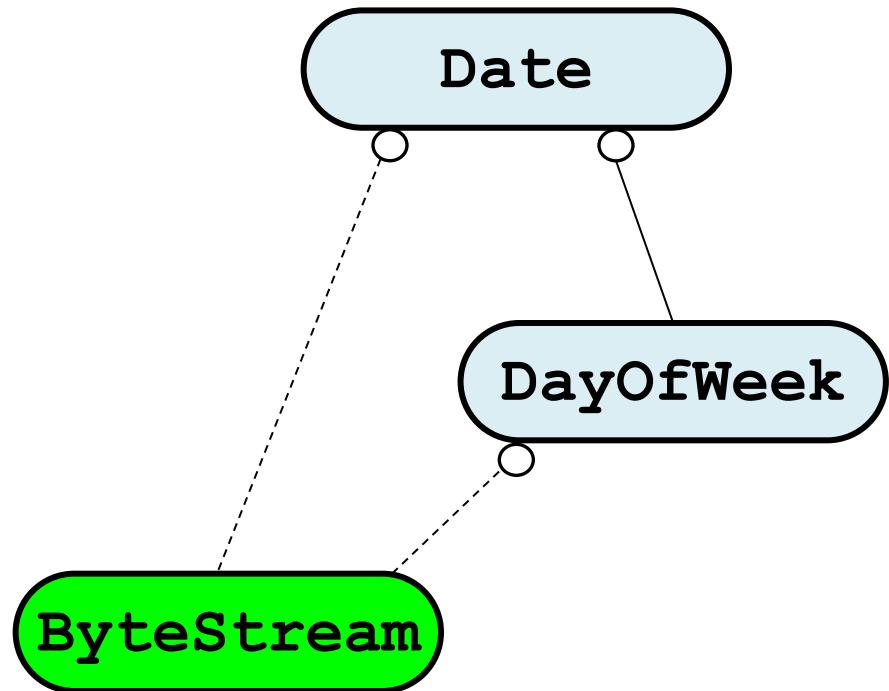
4. Bloomberg Development Environment

Represent a *Date Value* as a C++ Type



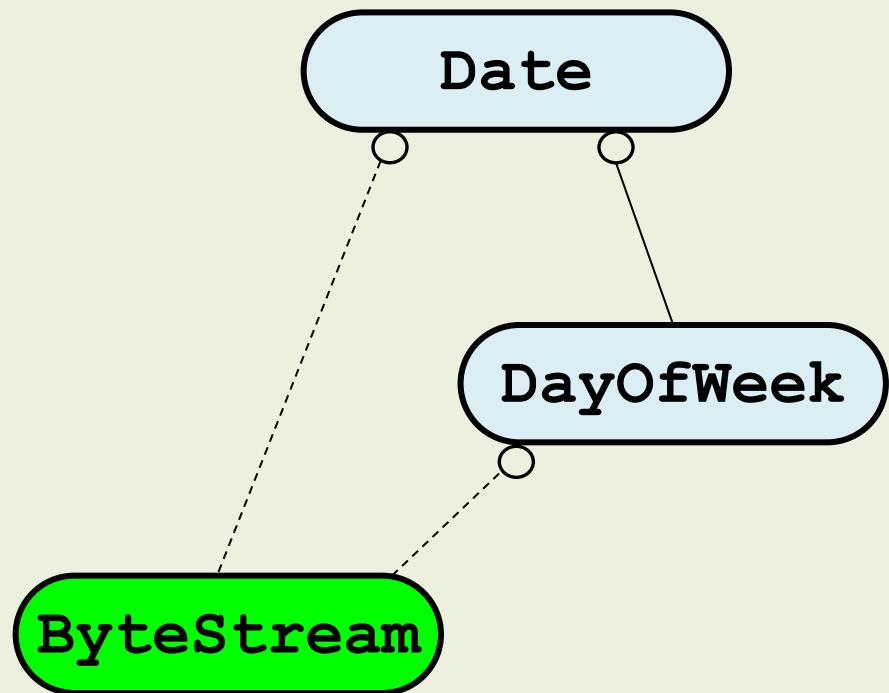
4. Bloomberg Development Environment

Represent a *Date Value* as a C++ Type



4. Bloomberg Development Environment

Solution 1: Represent a Date Value.



4. Bloomberg Development Environment

The Original Request

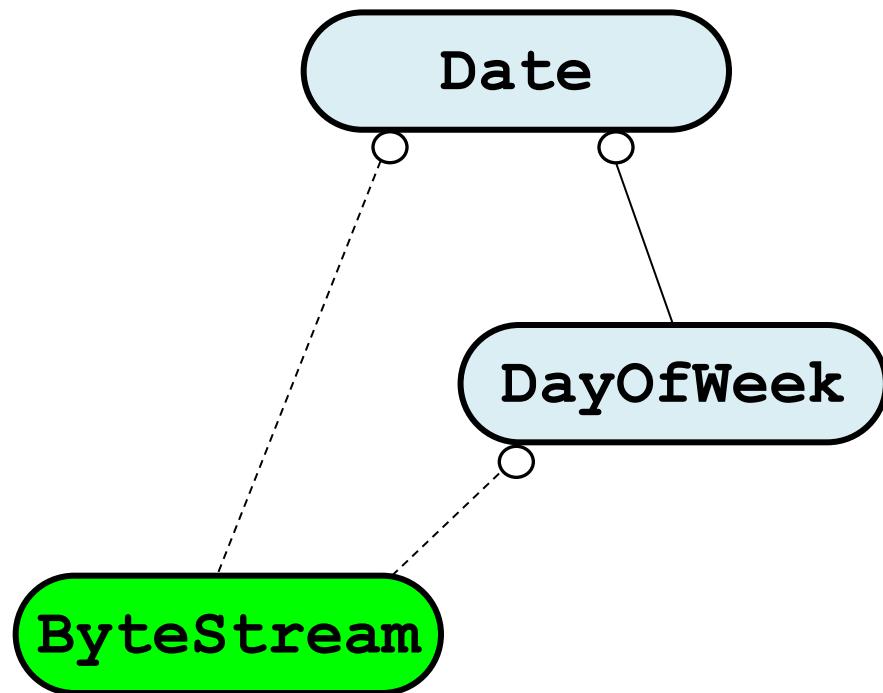
"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. **Determine what date value *today* is.**
3. Determine if a date value is a *business day*.
4. **Provide well-factored useful components that we'll need over and over again!**

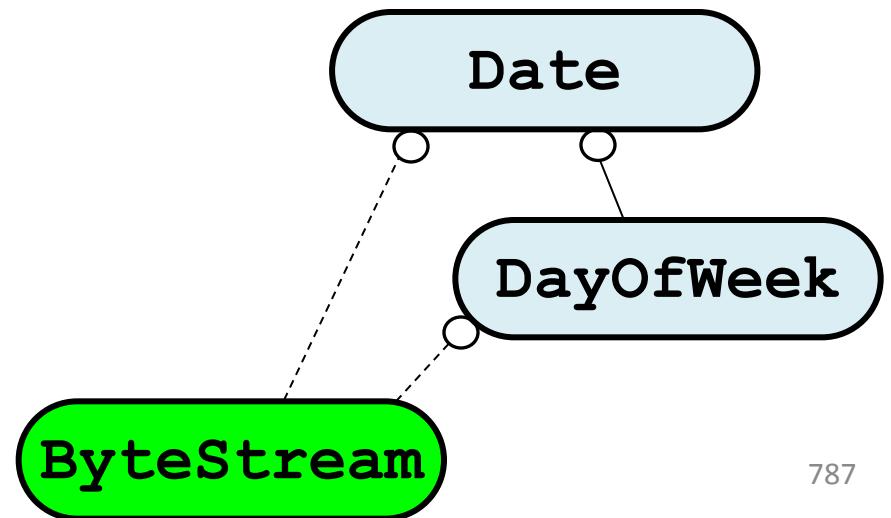
4. Bloomberg Development Environment

Determine what Date Value *today* is



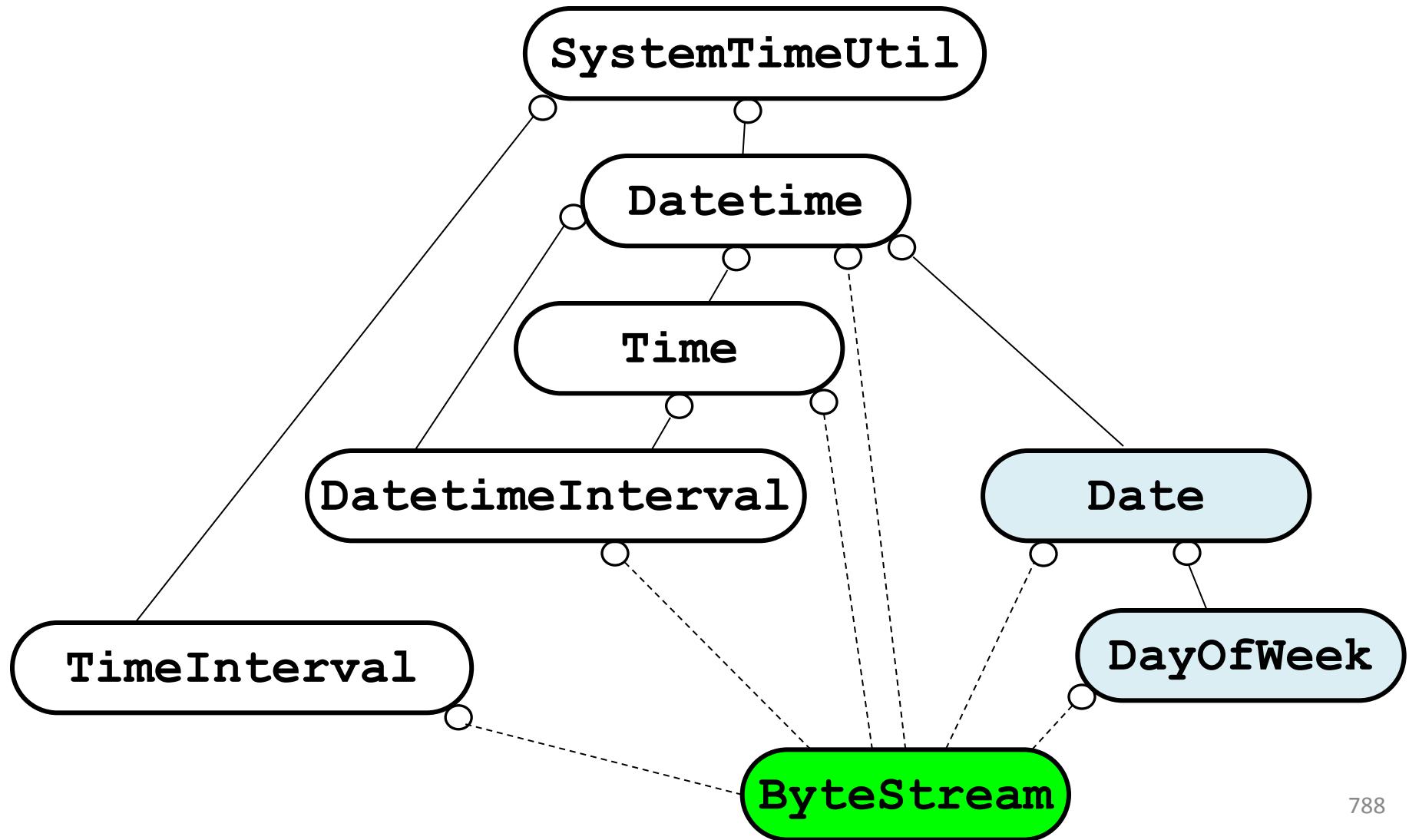
4. Bloomberg Development Environment

Determine what Date Value *today* is



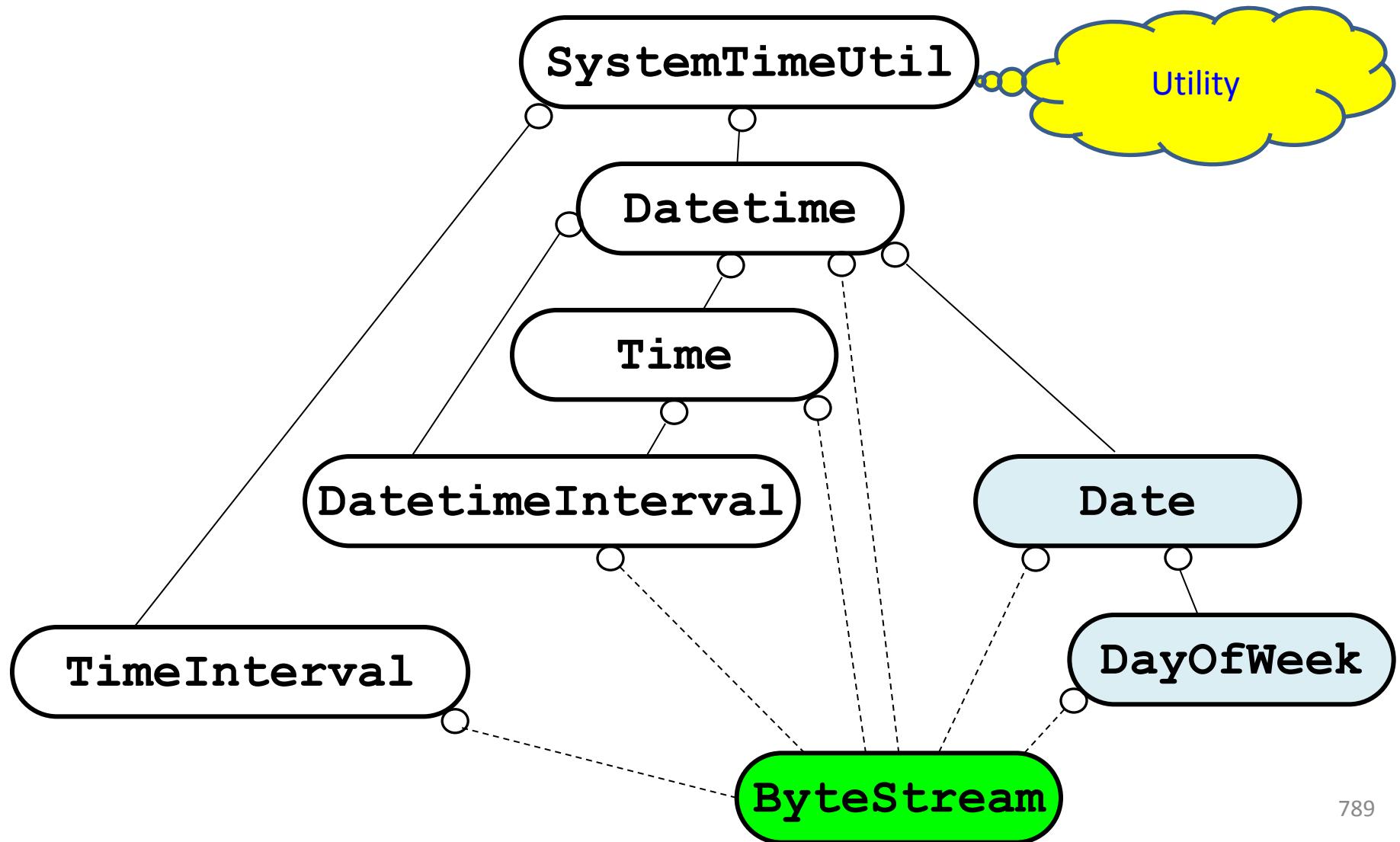
4. Bloomberg Development Environment

Determine what Date Value *today* is



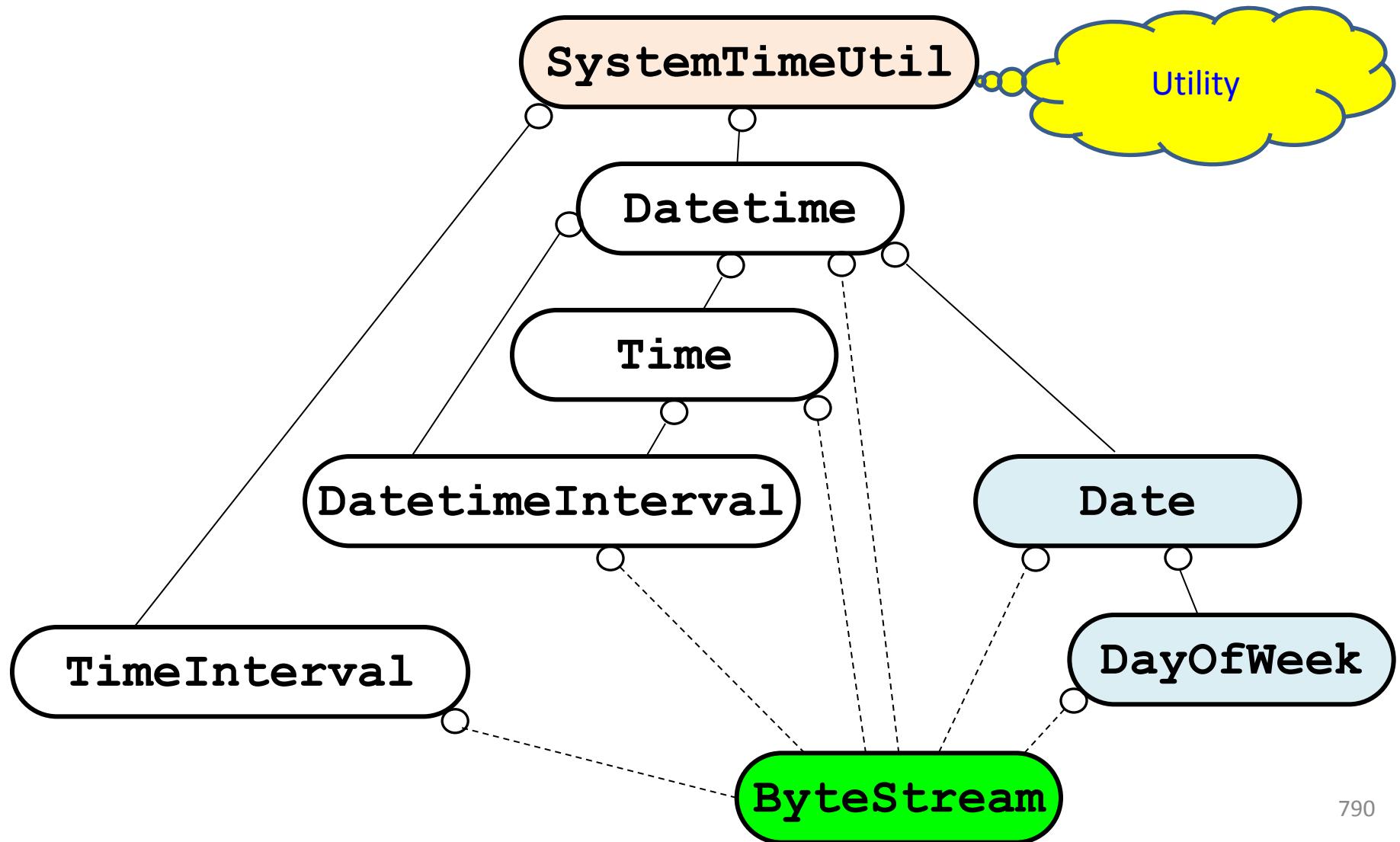
4. Bloomberg Development Environment

Determine what Date Value *today* is



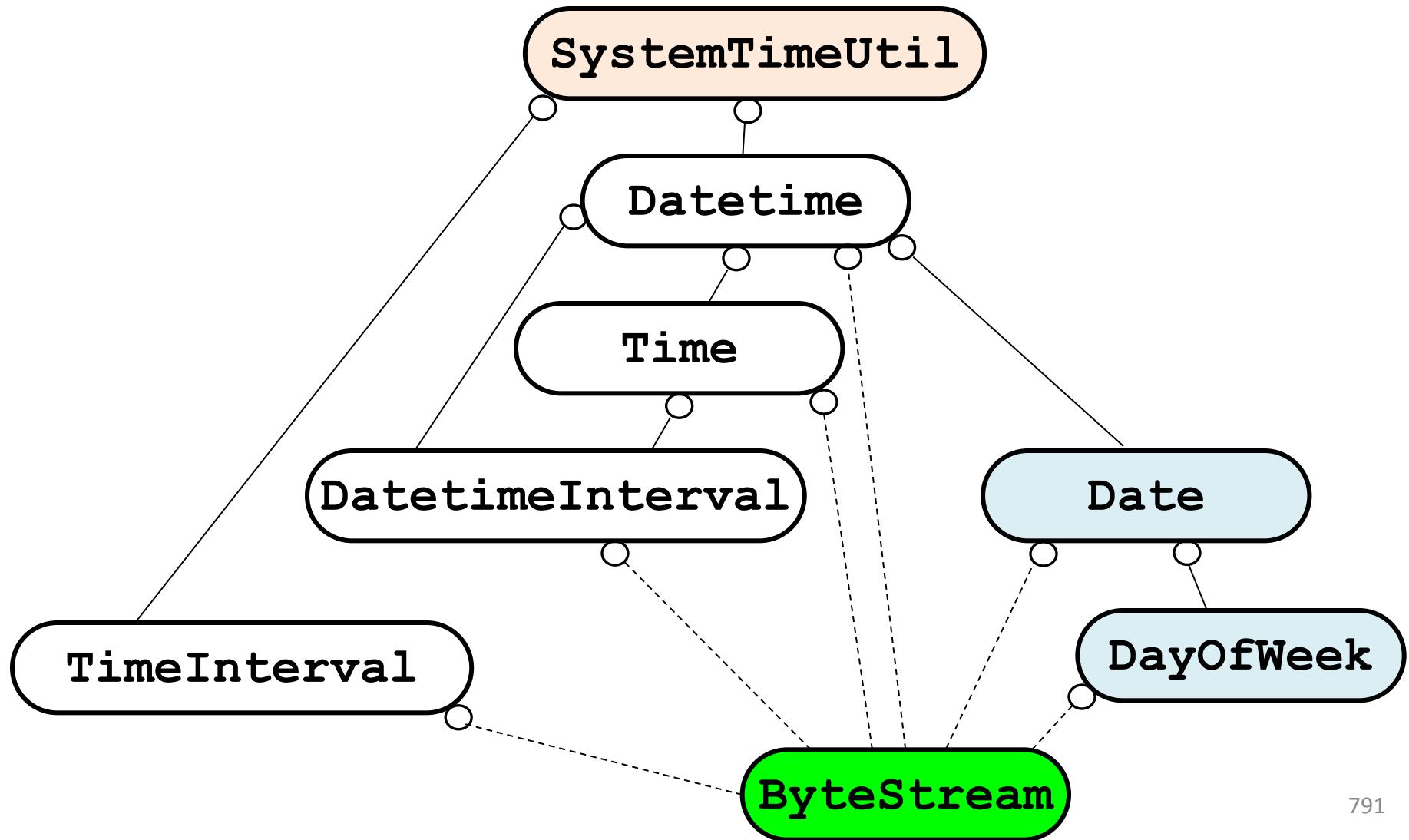
4. Bloomberg Development Environment

Determine what Date Value *today* is



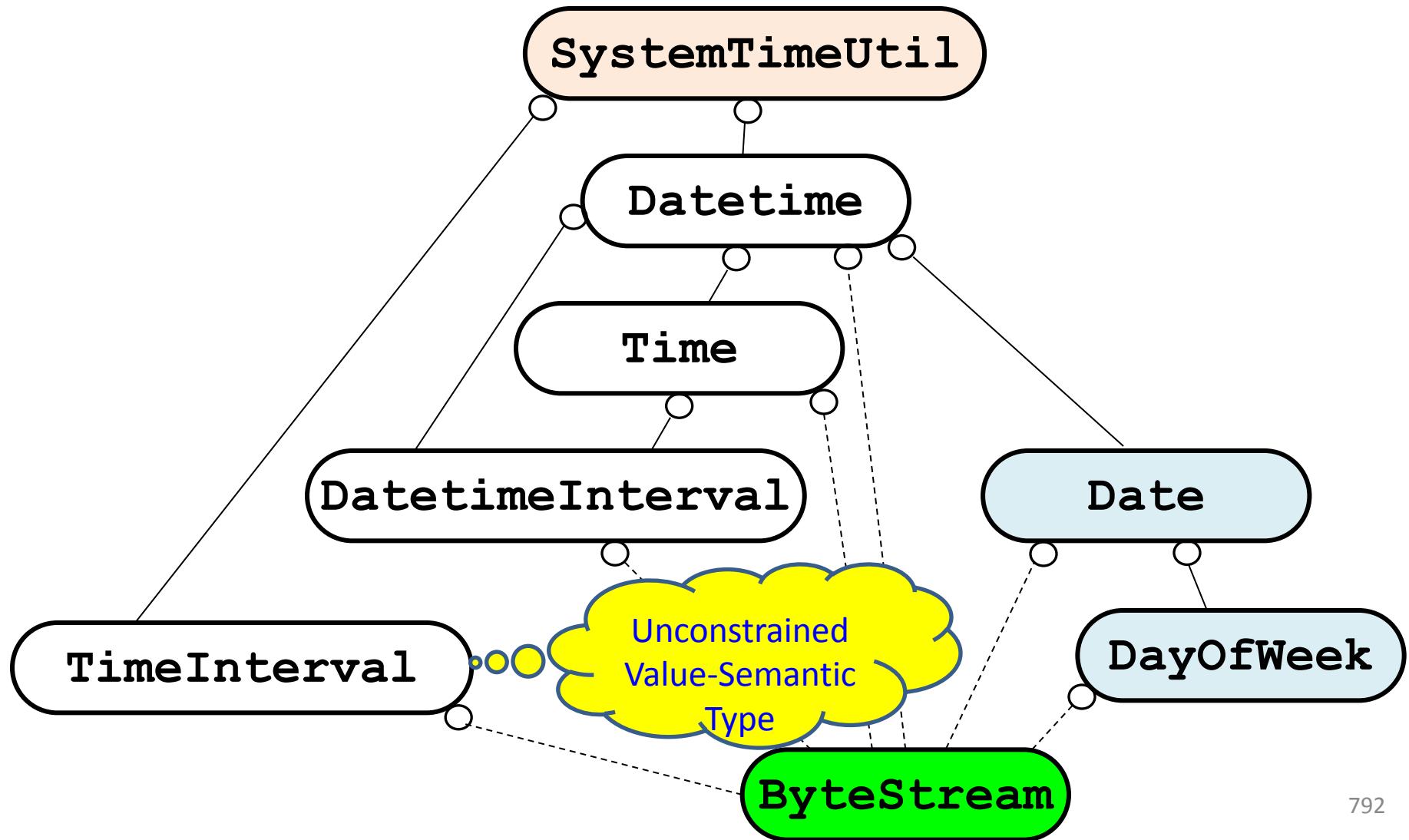
4. Bloomberg Development Environment

Determine what Date Value *today* is



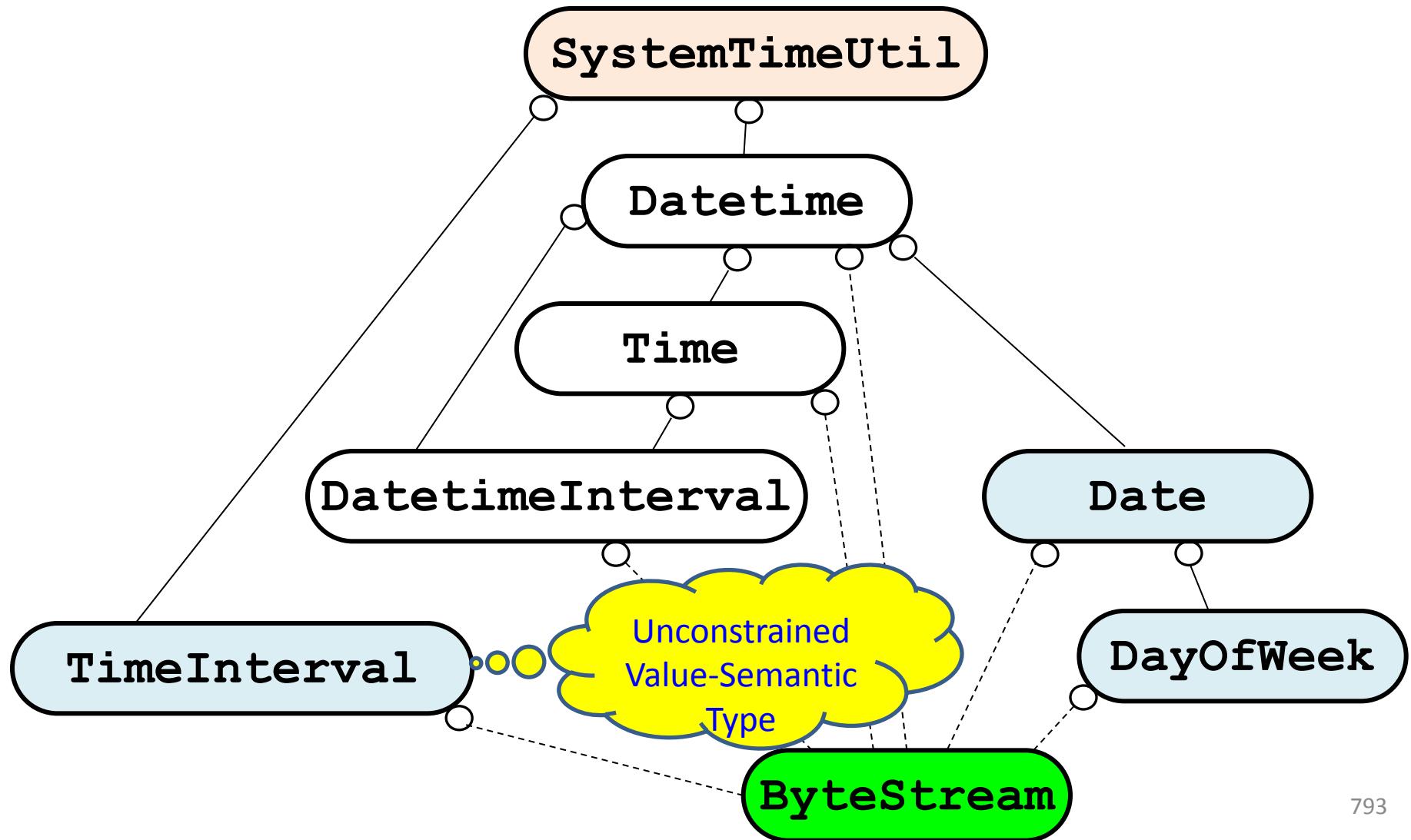
4. Bloomberg Development Environment

Determine what Date Value *today* is



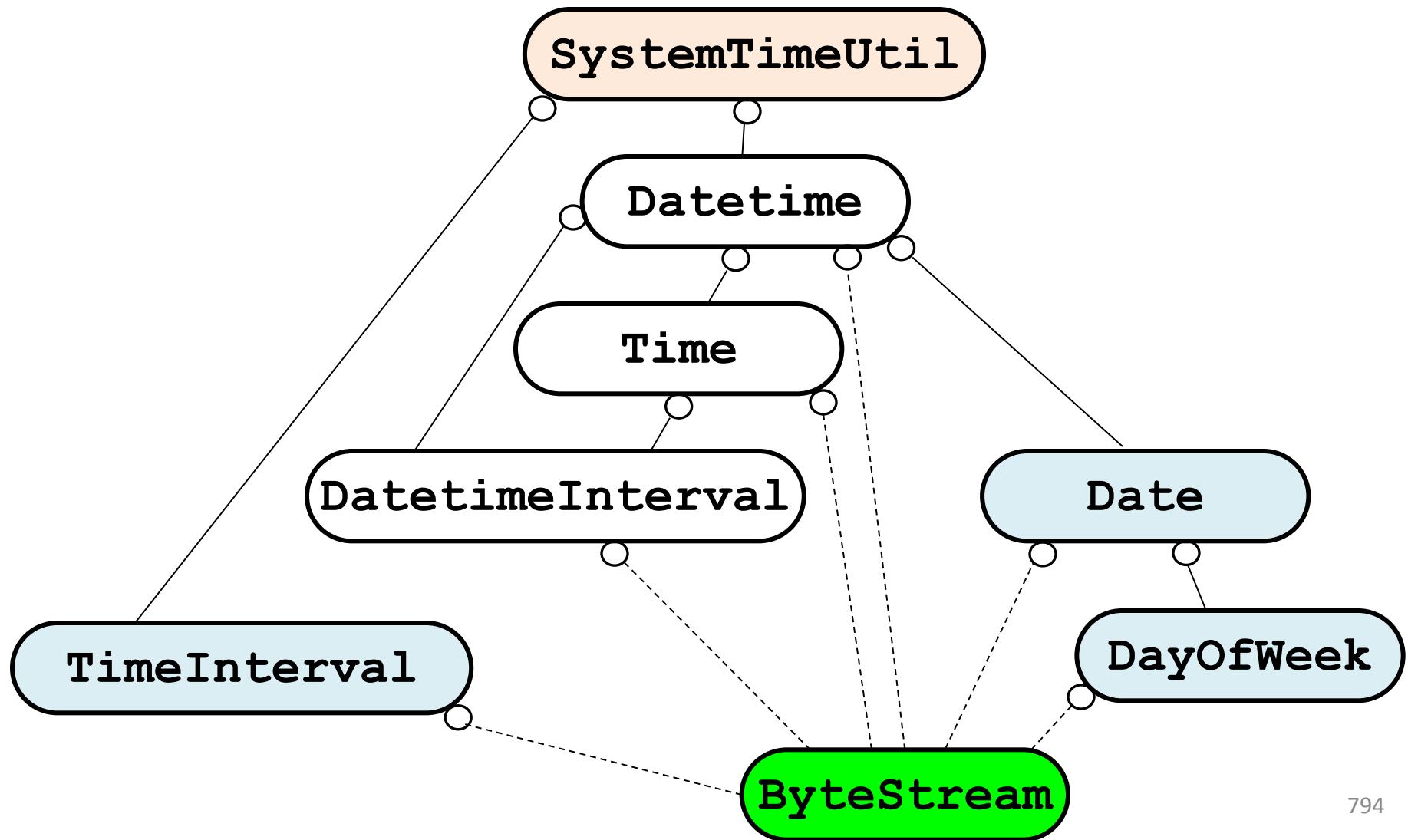
4. Bloomberg Development Environment

Determine what Date Value *today* is



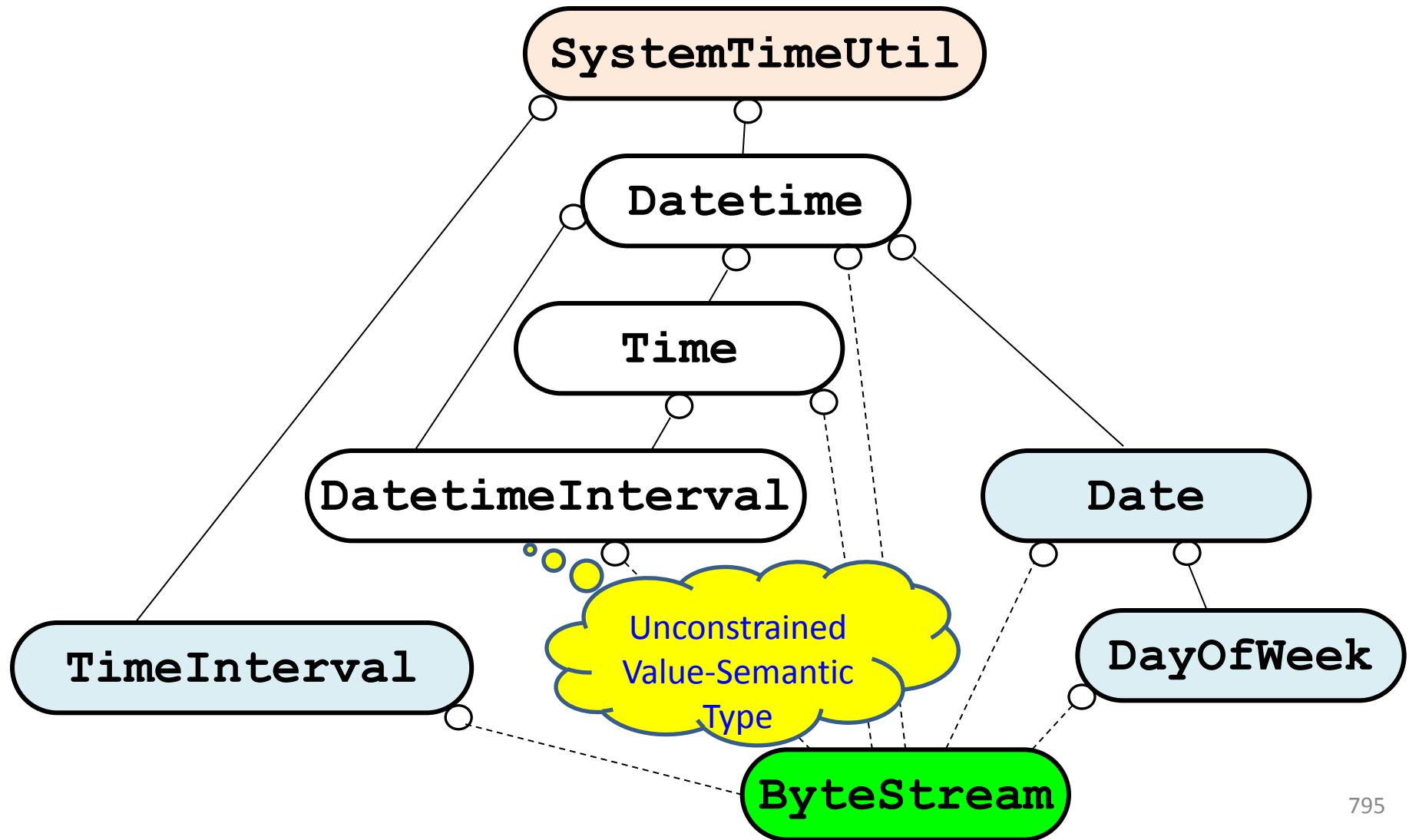
4. Bloomberg Development Environment

Determine what Date Value *today* is



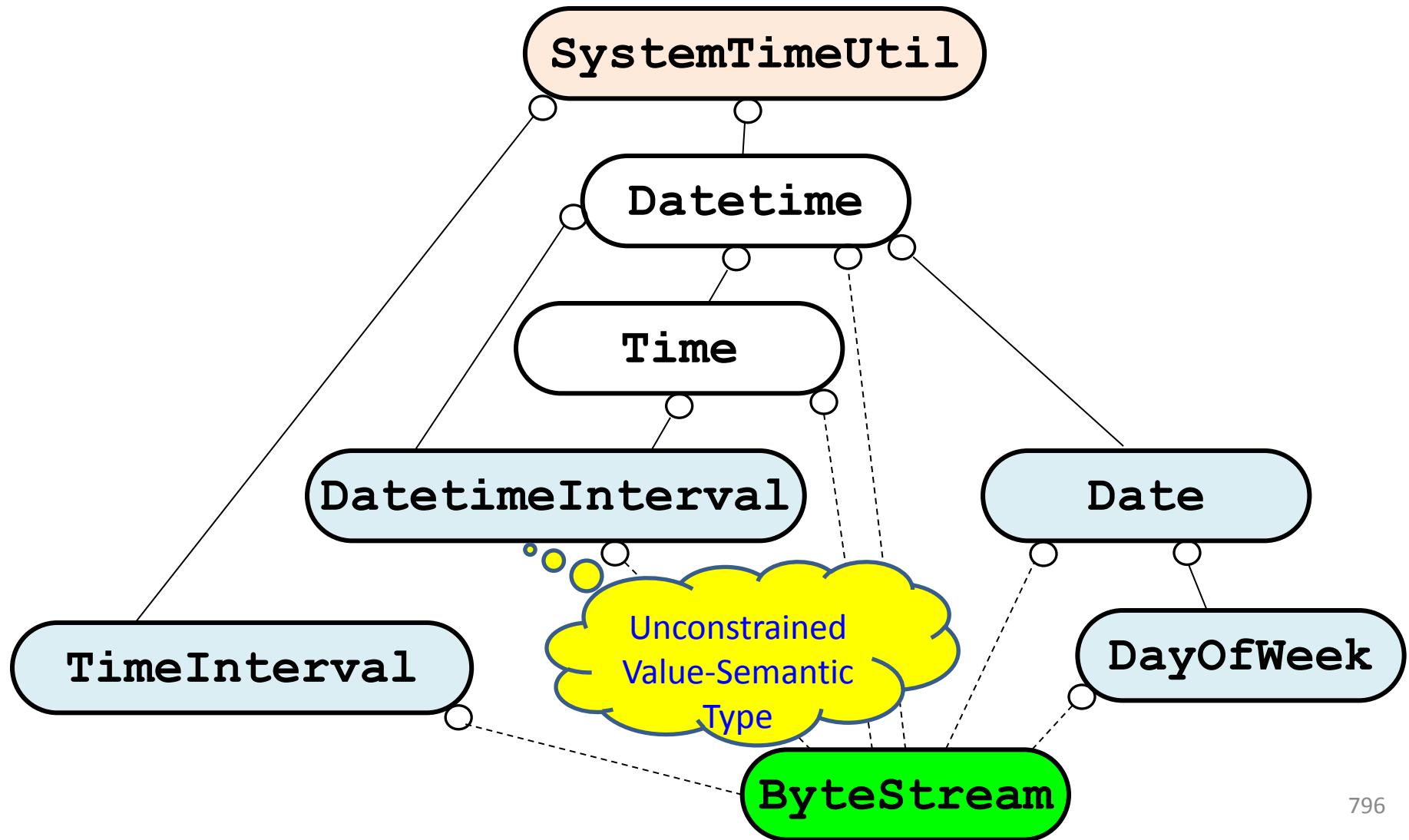
4. Bloomberg Development Environment

Determine what Date Value *today* is



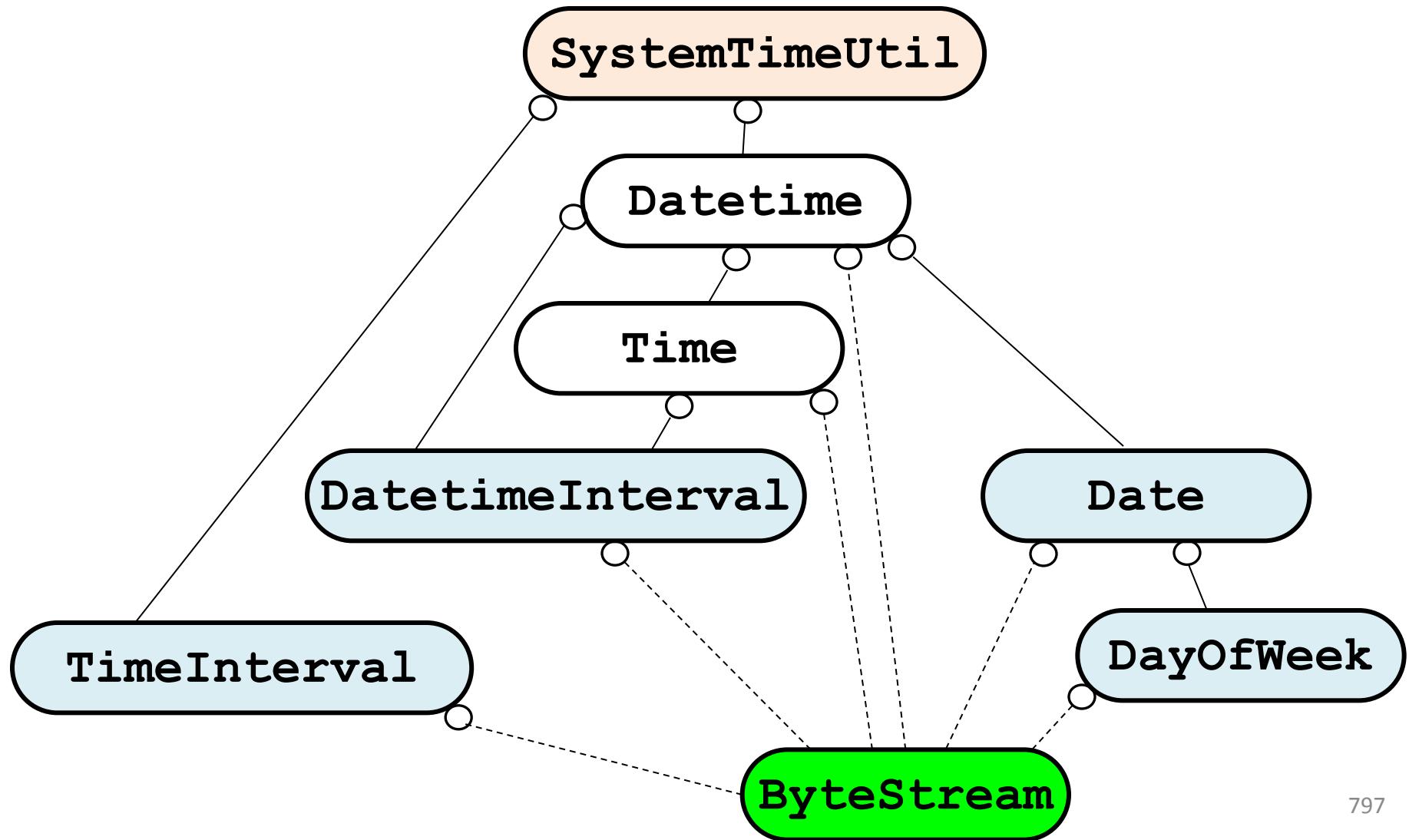
4. Bloomberg Development Environment

Determine what Date Value *today* is



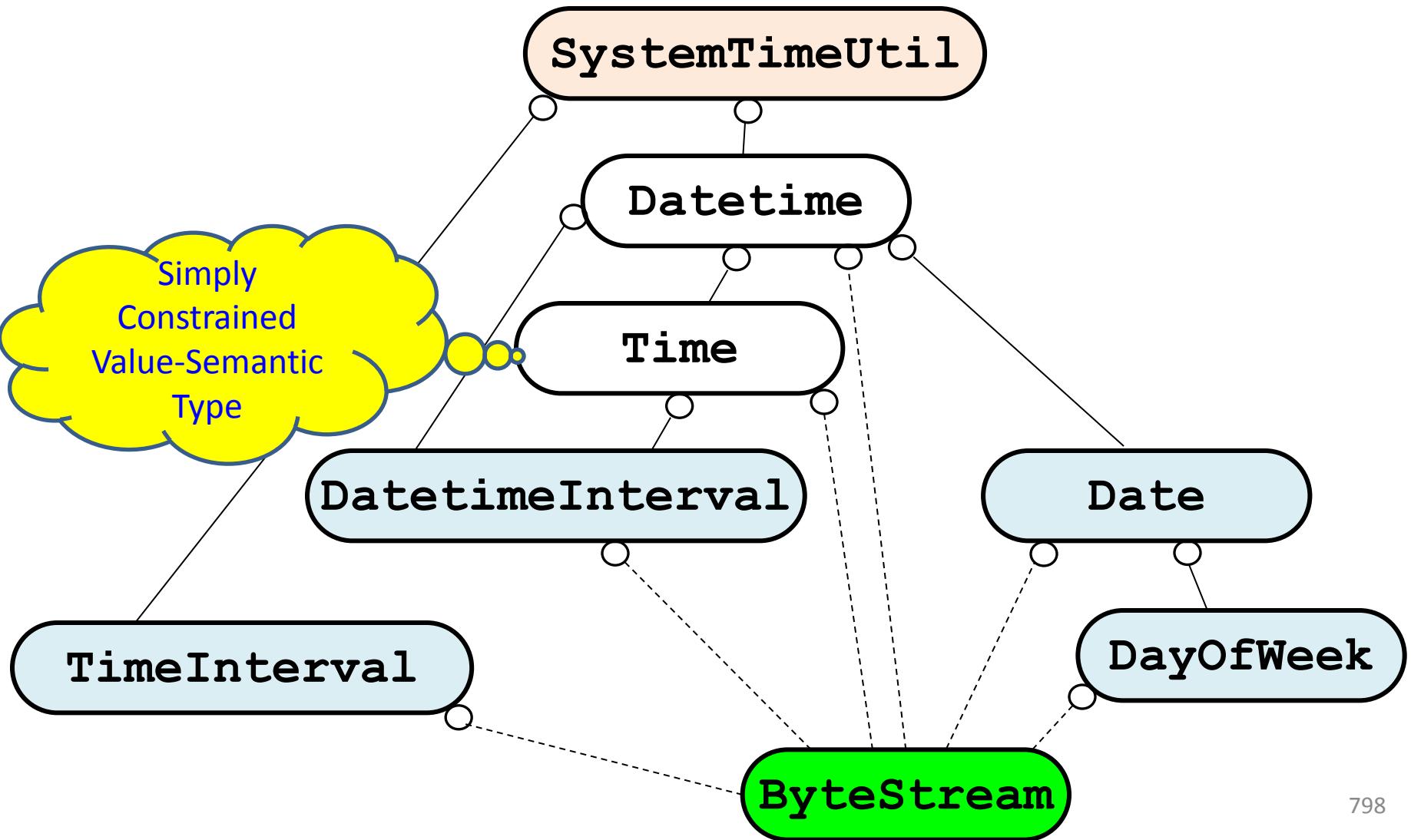
4. Bloomberg Development Environment

Determine what Date Value *today* is



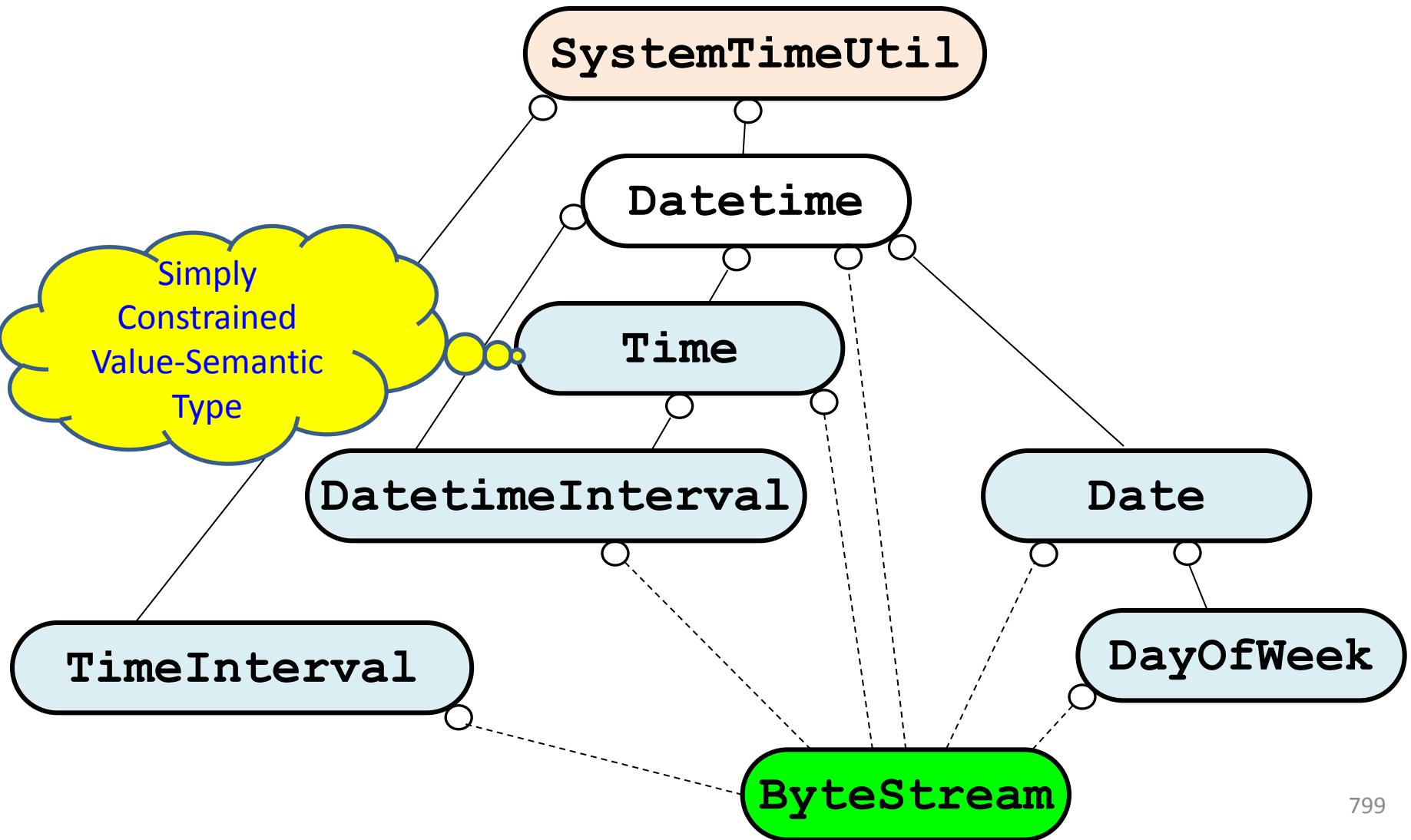
4. Bloomberg Development Environment

Determine what Date Value *today* is



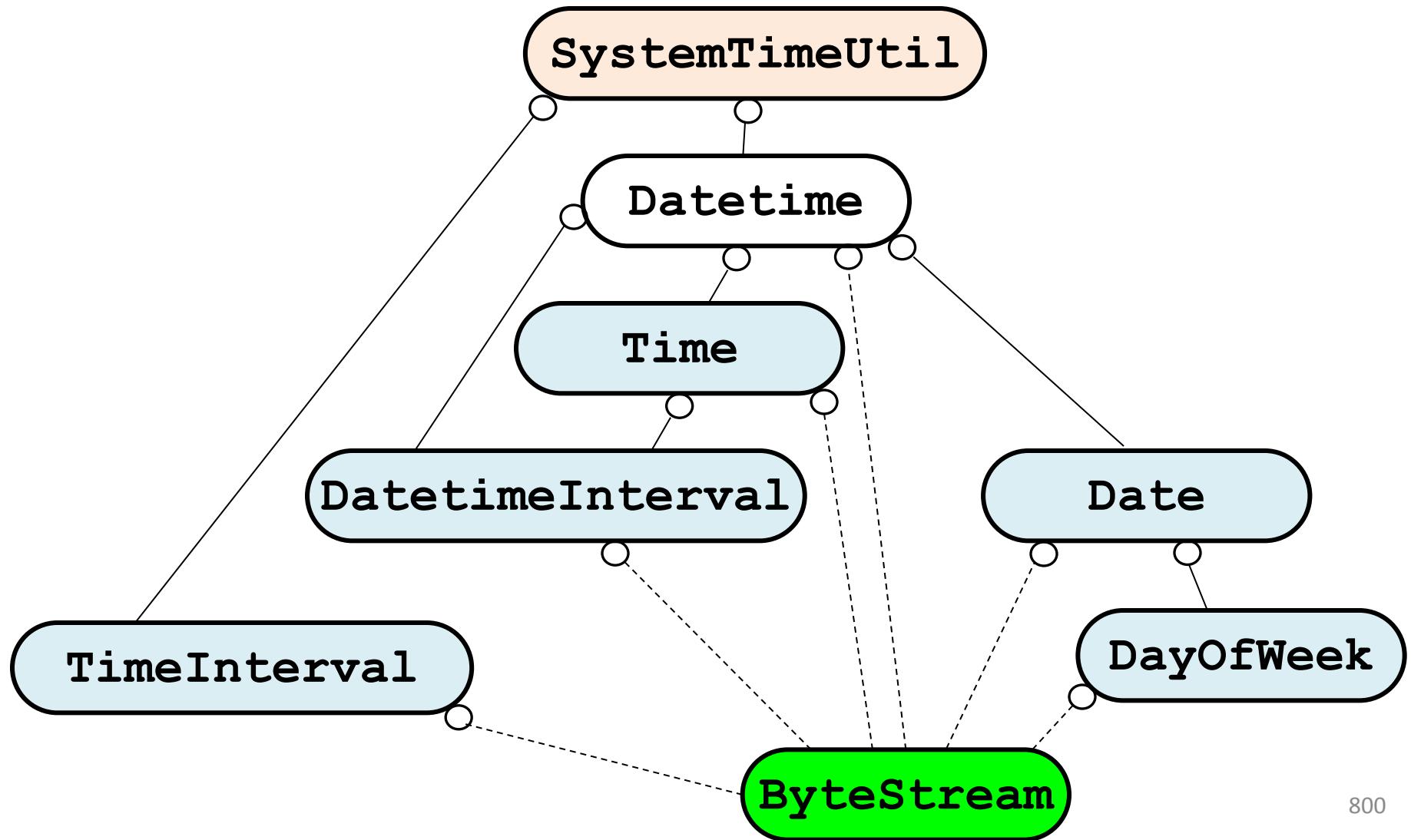
4. Bloomberg Development Environment

Determine what Date Value *today* is



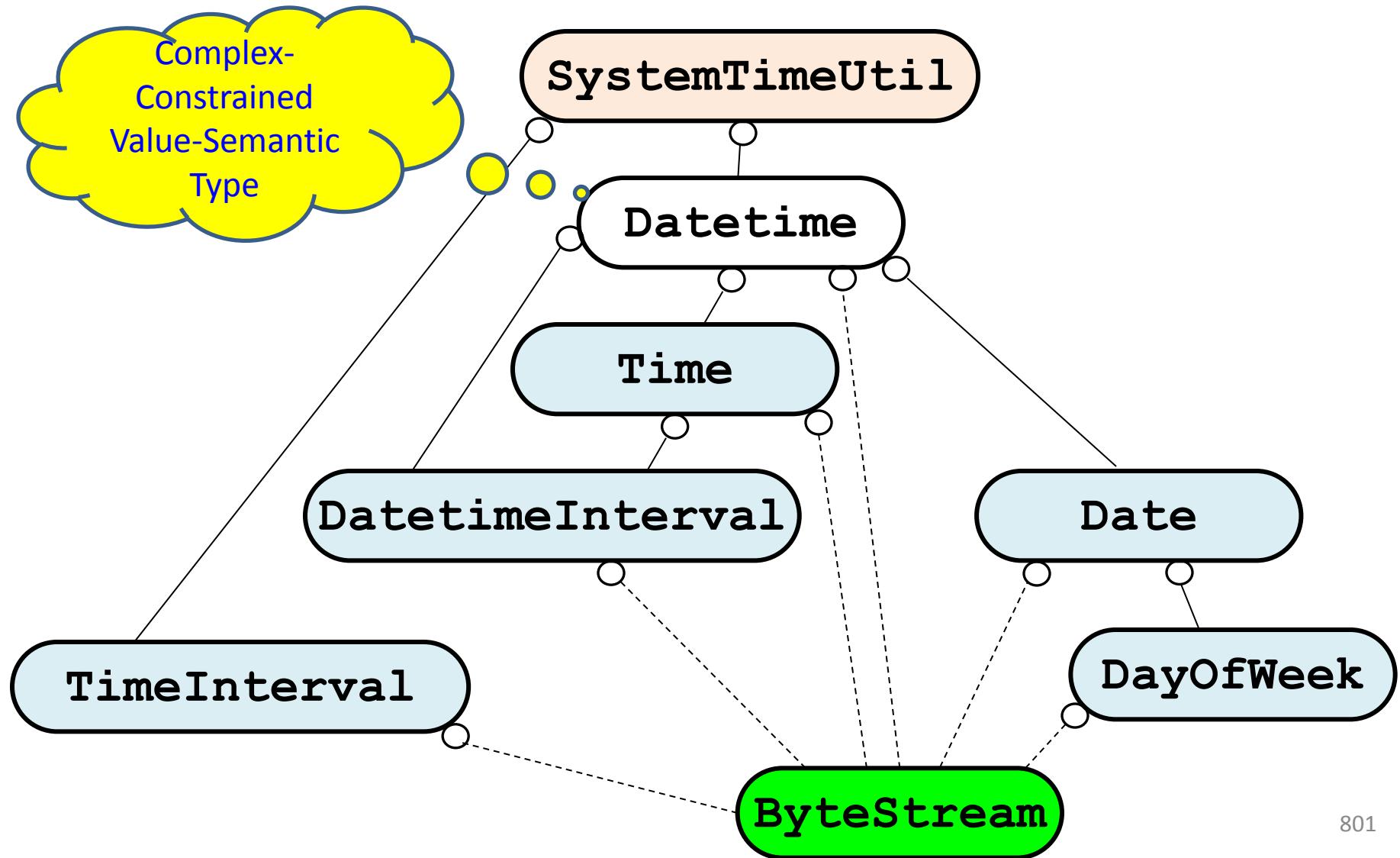
4. Bloomberg Development Environment

Determine what Date Value *today* is



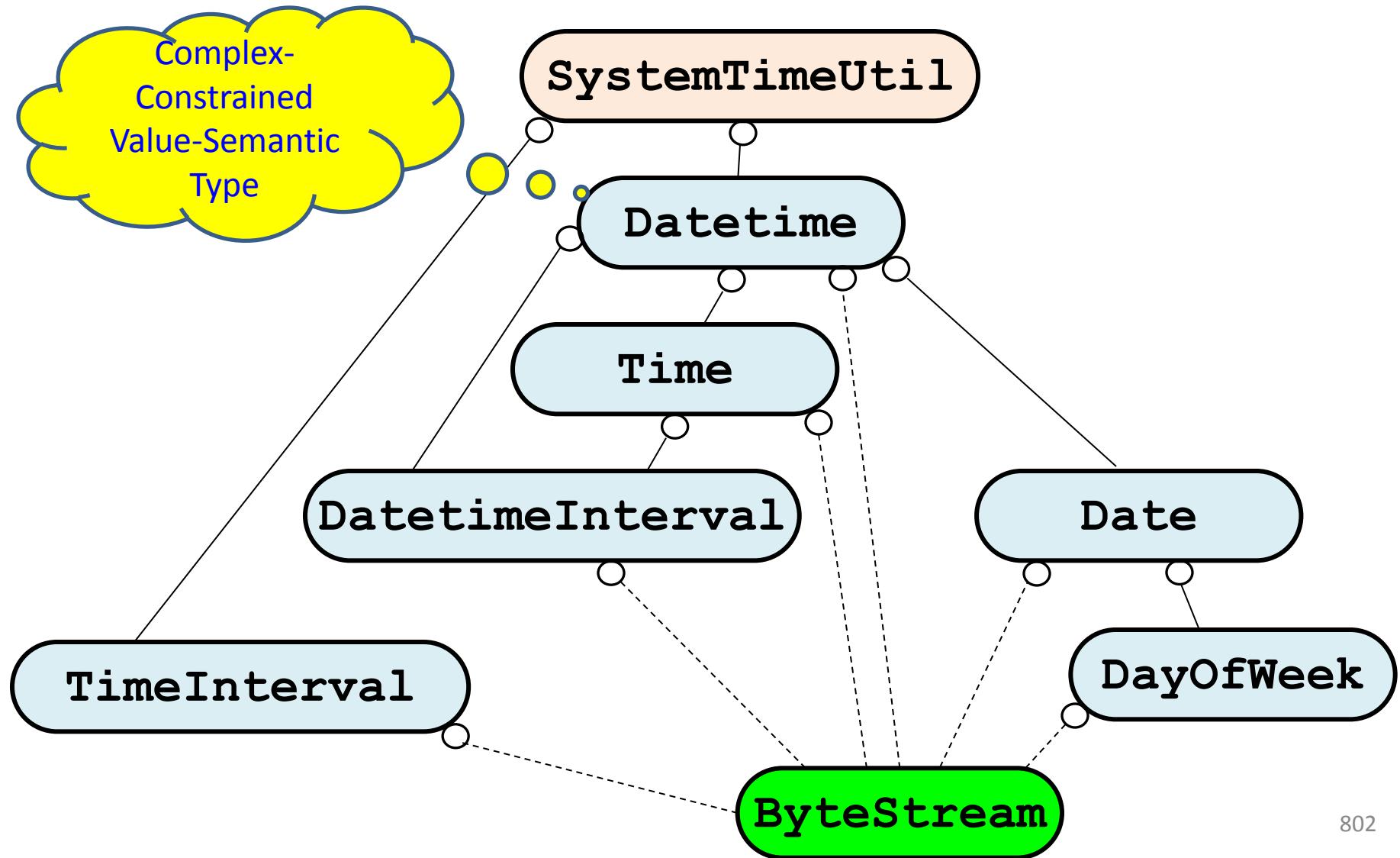
4. Bloomberg Development Environment

Determine what Date Value *today* is



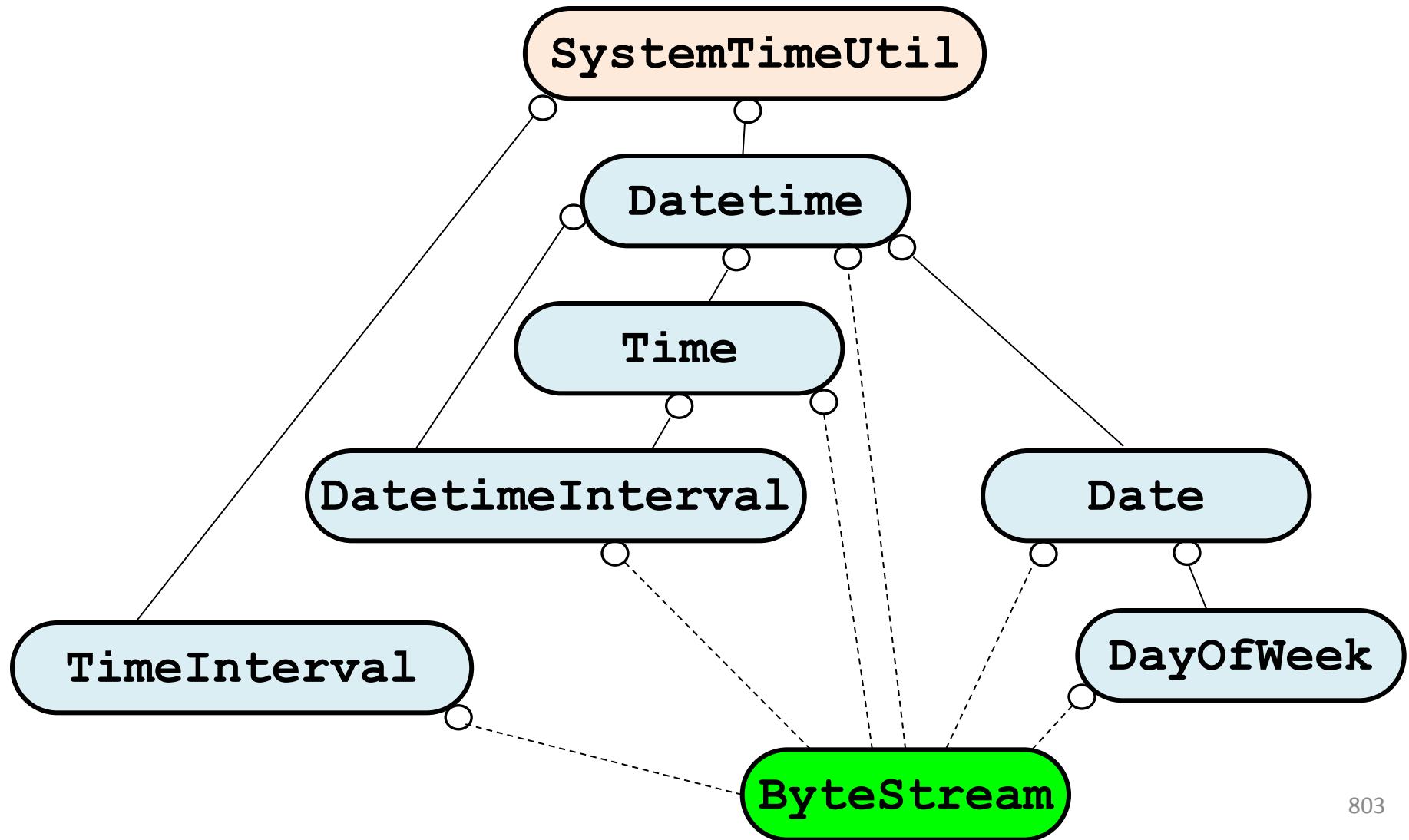
4. Bloomberg Development Environment

Determine what Date Value *today* is



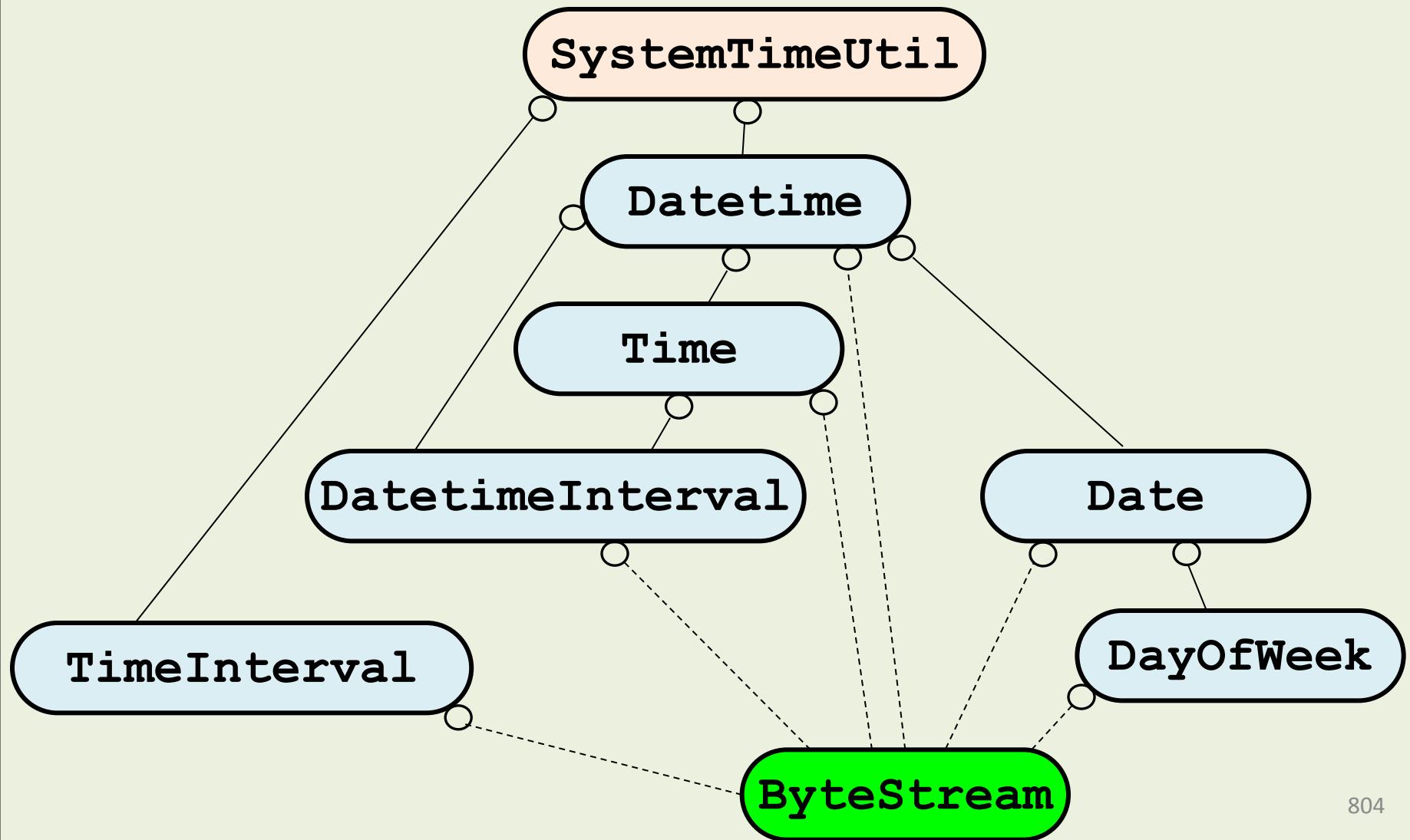
4. Bloomberg Development Environment

Determine what Date Value *today* is



4. Bloomberg Development Environment

Solution 2: What Date is Today?



4. Bloomberg Development Environment

The Original Request

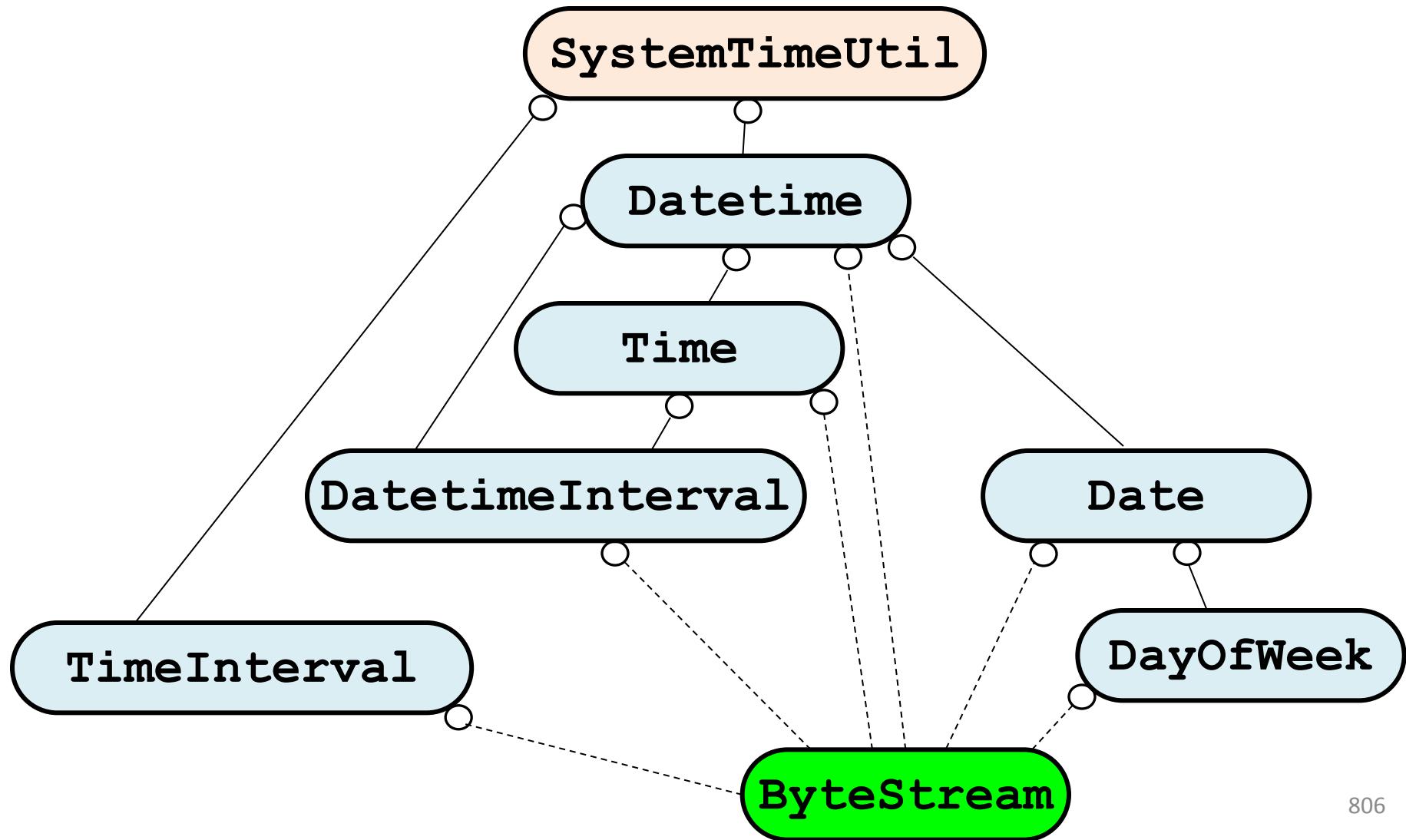
"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. Determine what date value *today* is.
3. **Determine if a date value is a *business day*.**
4. Provide well-factored useful components that we'll need over and over again!

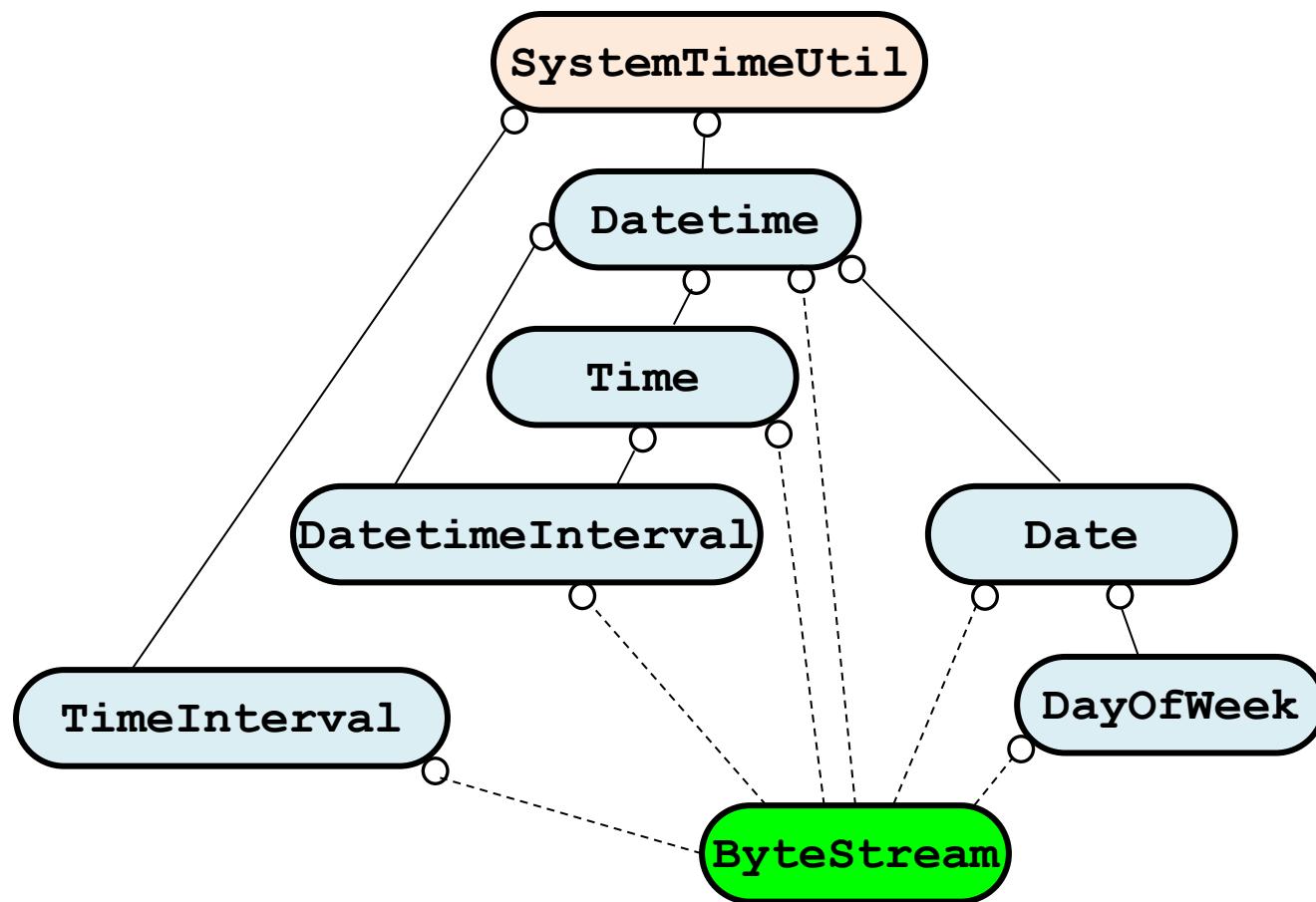
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



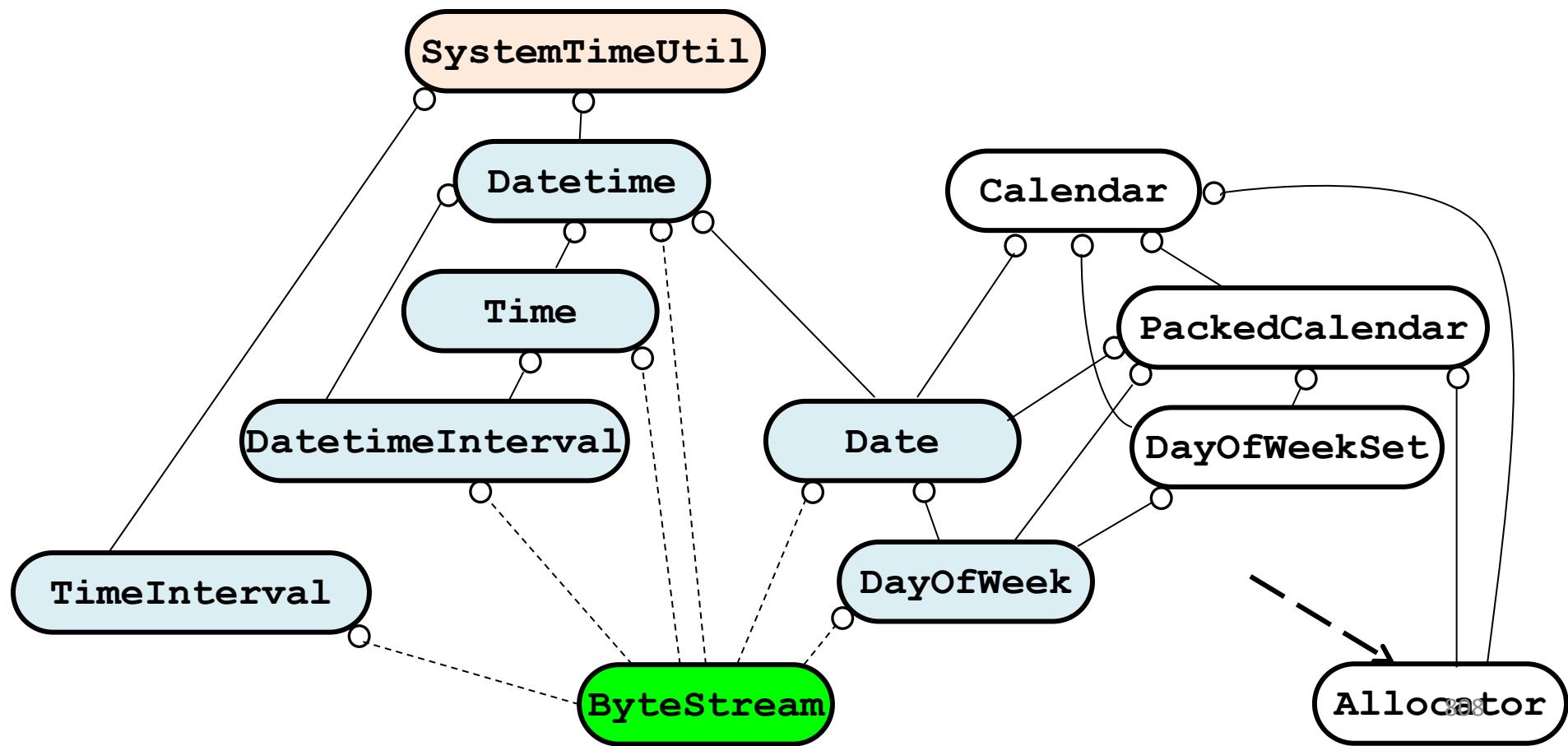
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



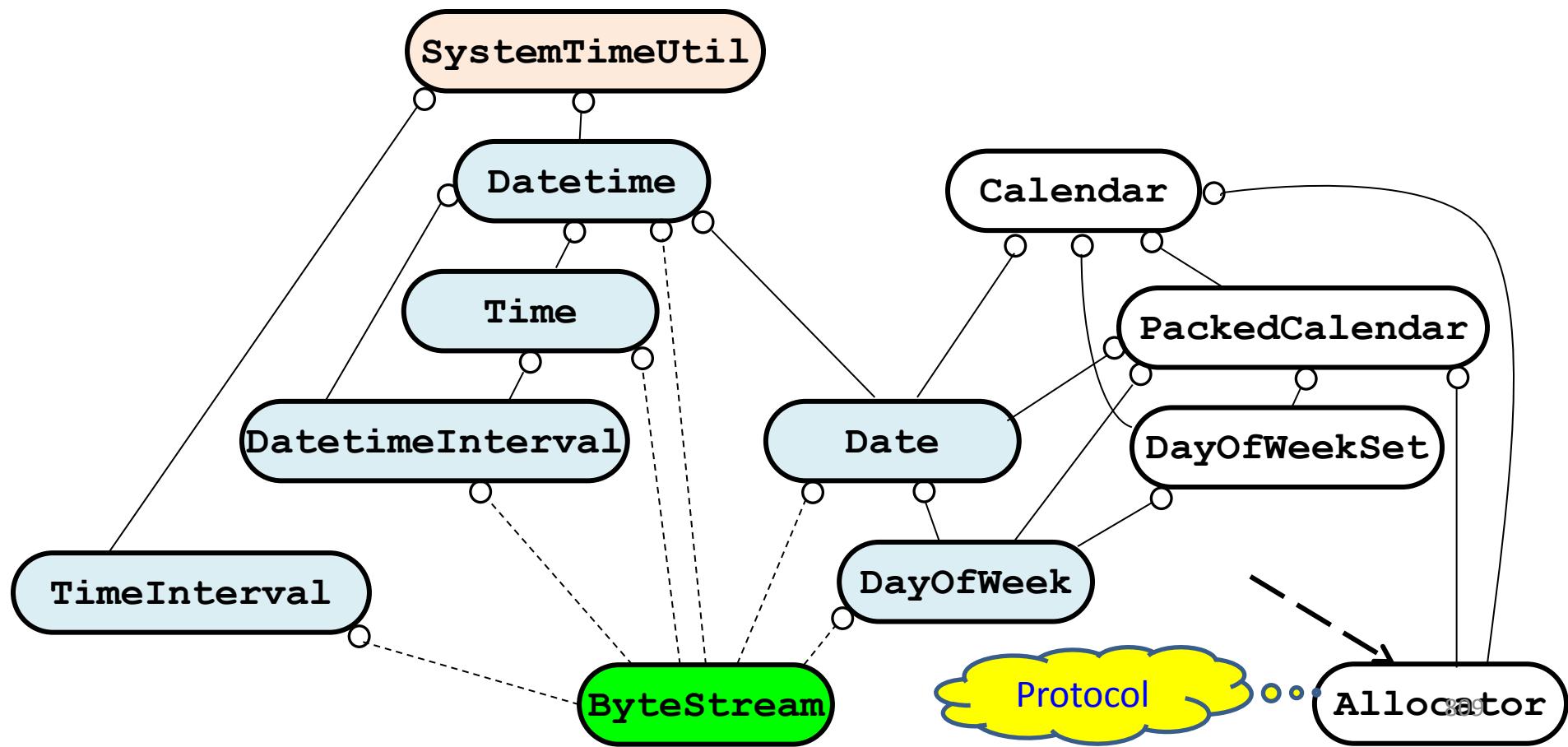
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



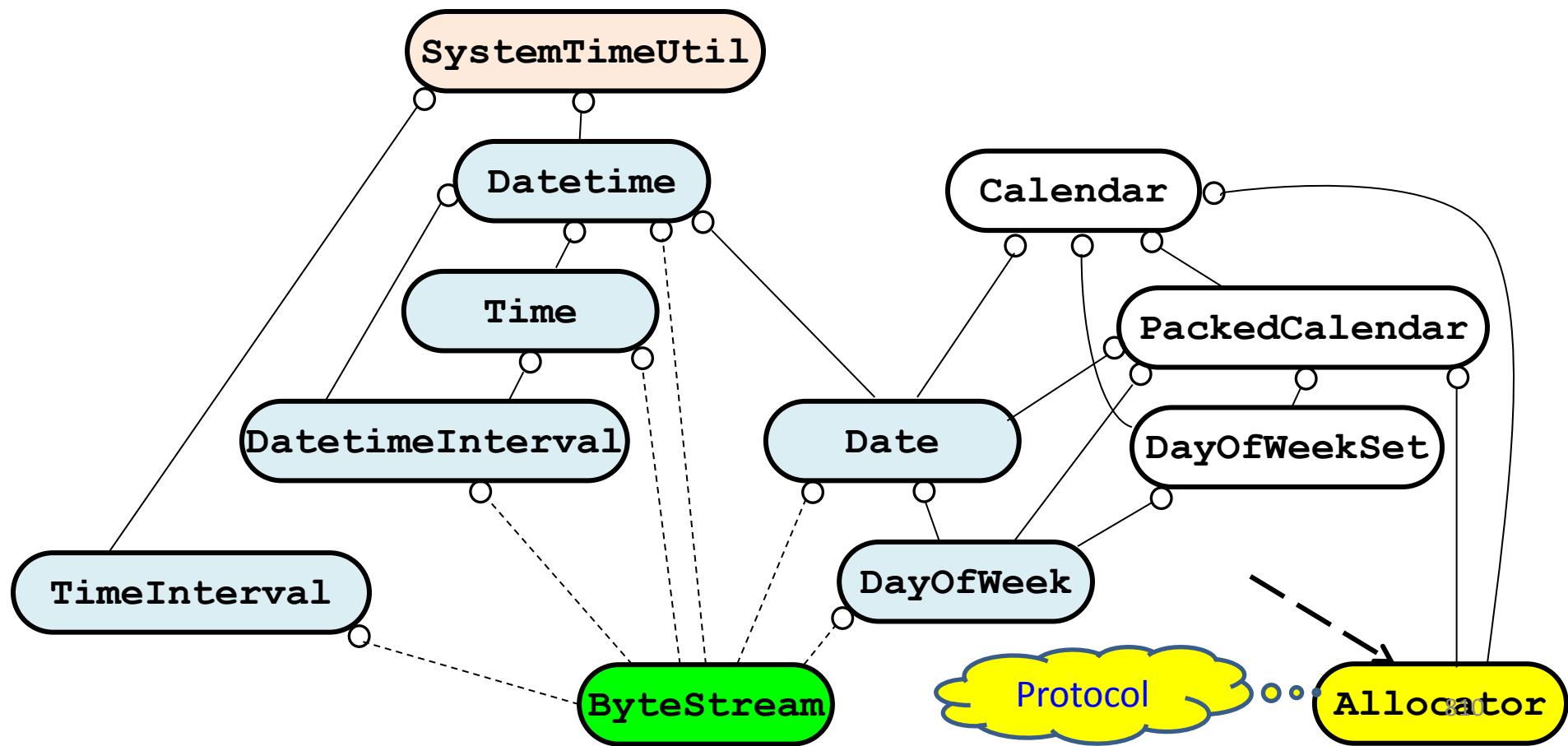
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



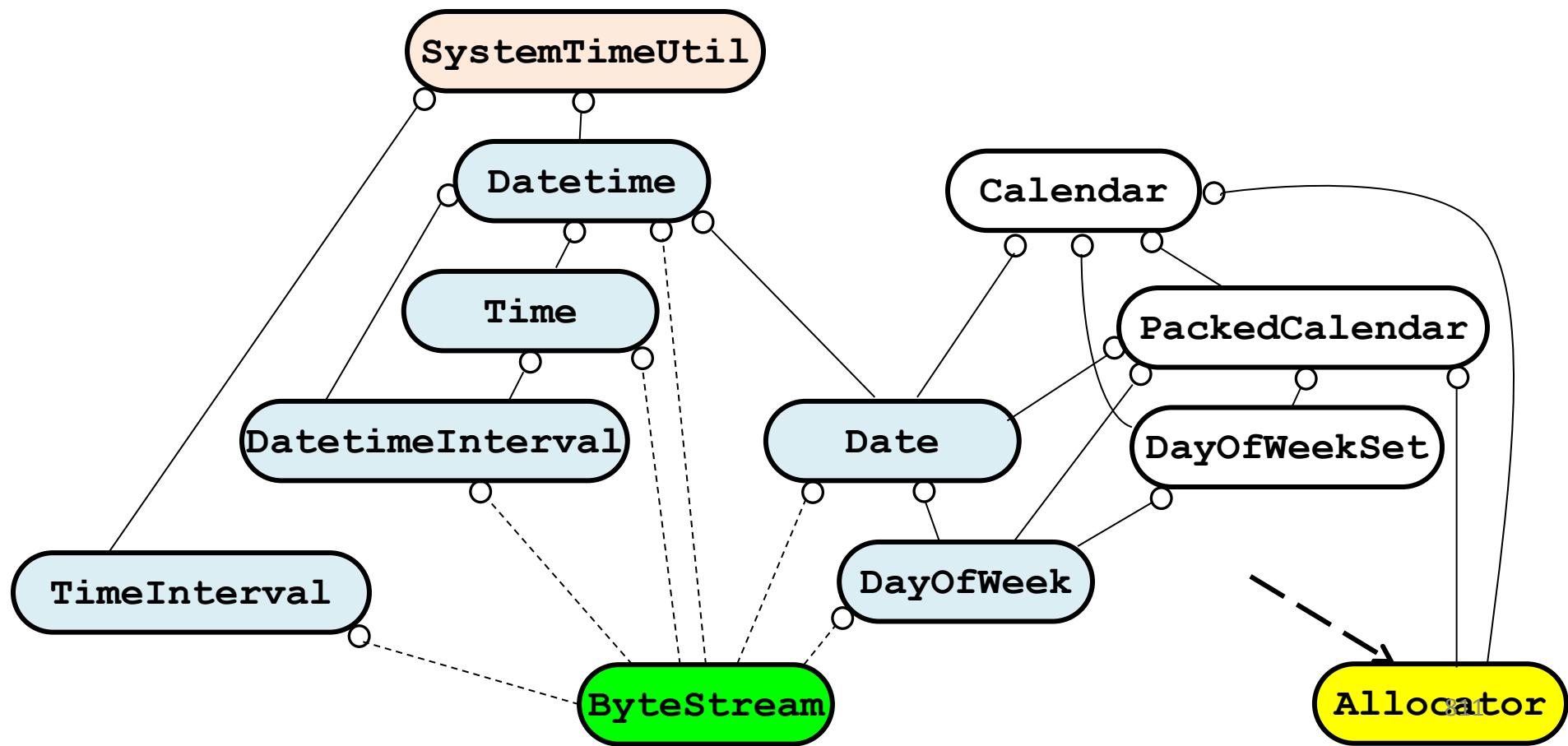
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



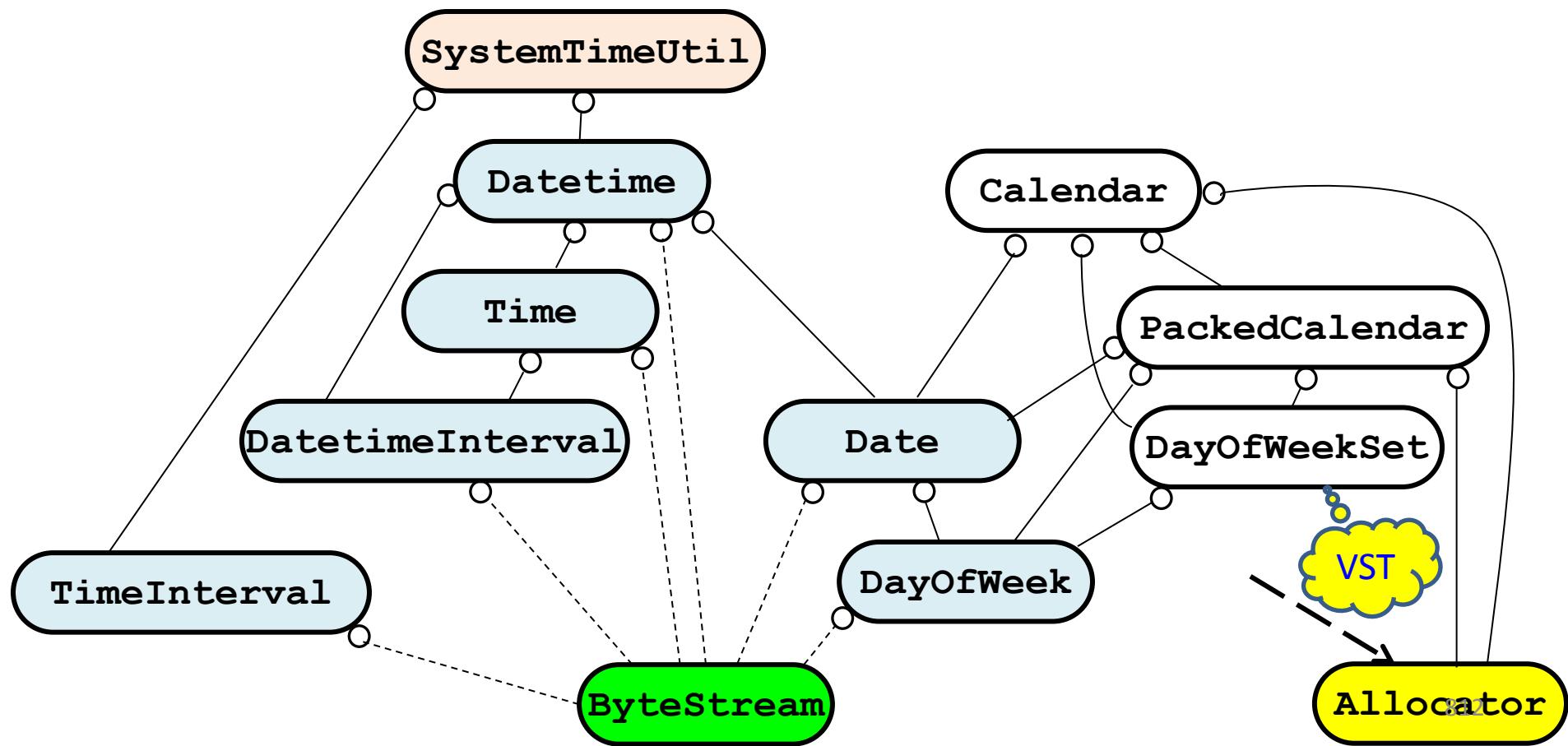
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



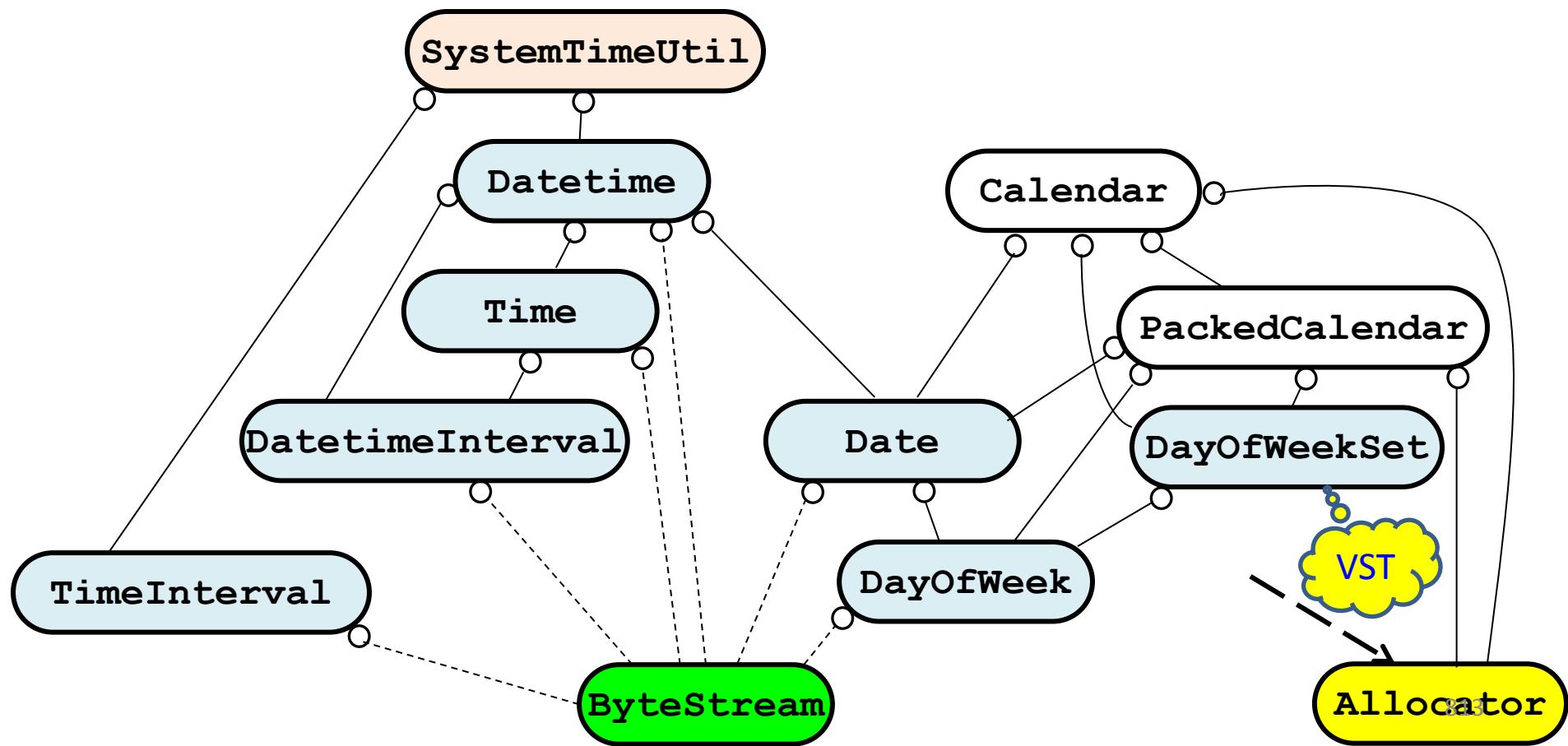
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



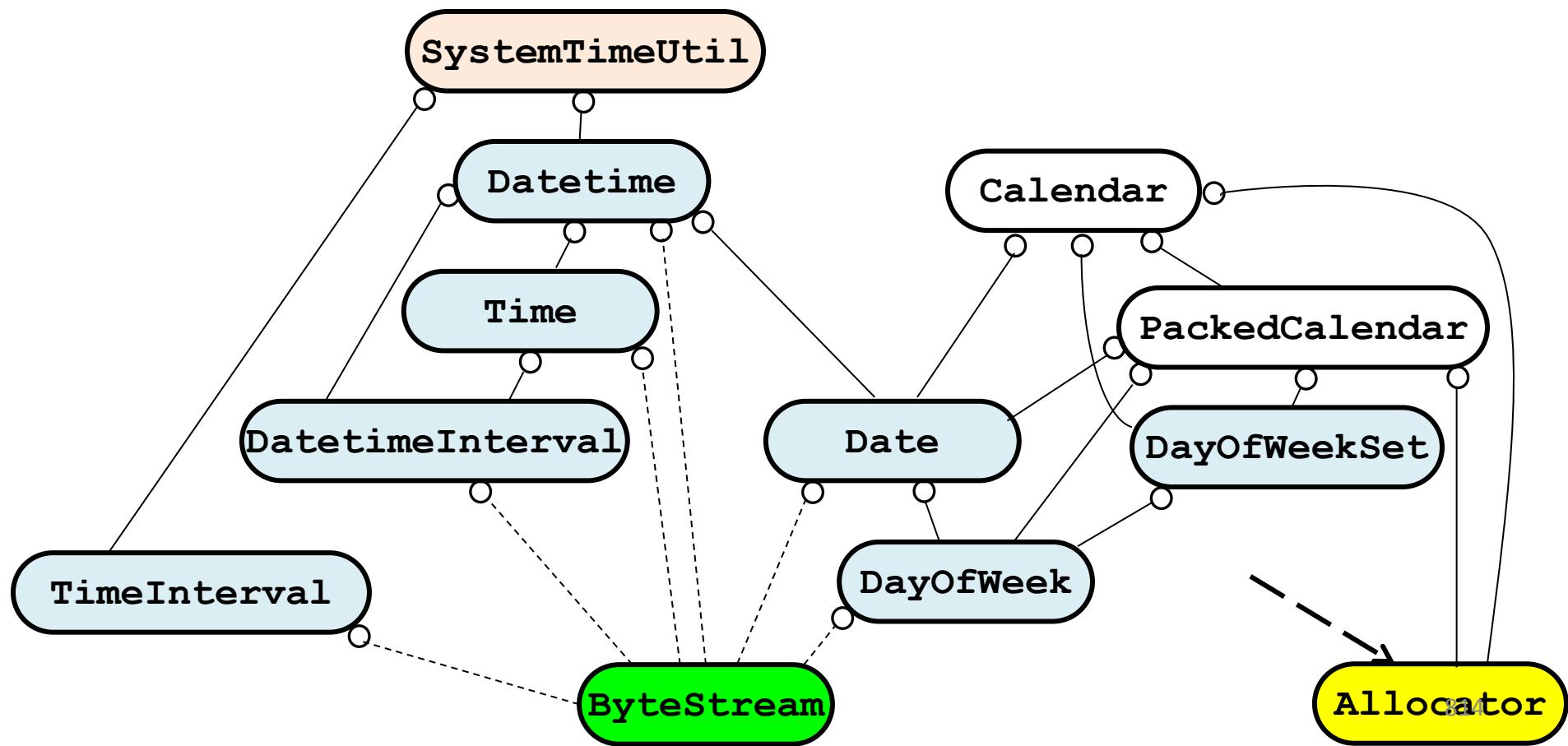
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



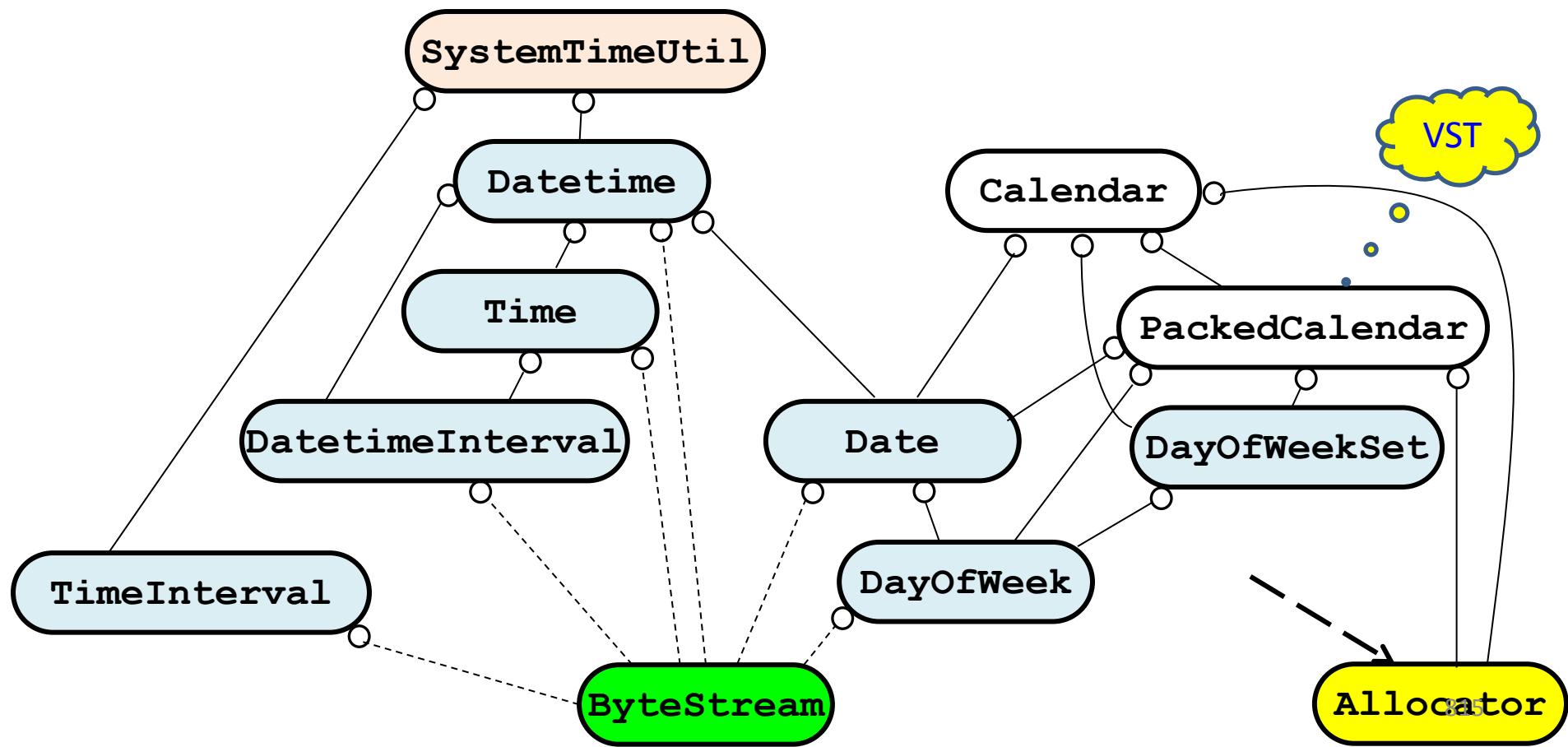
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



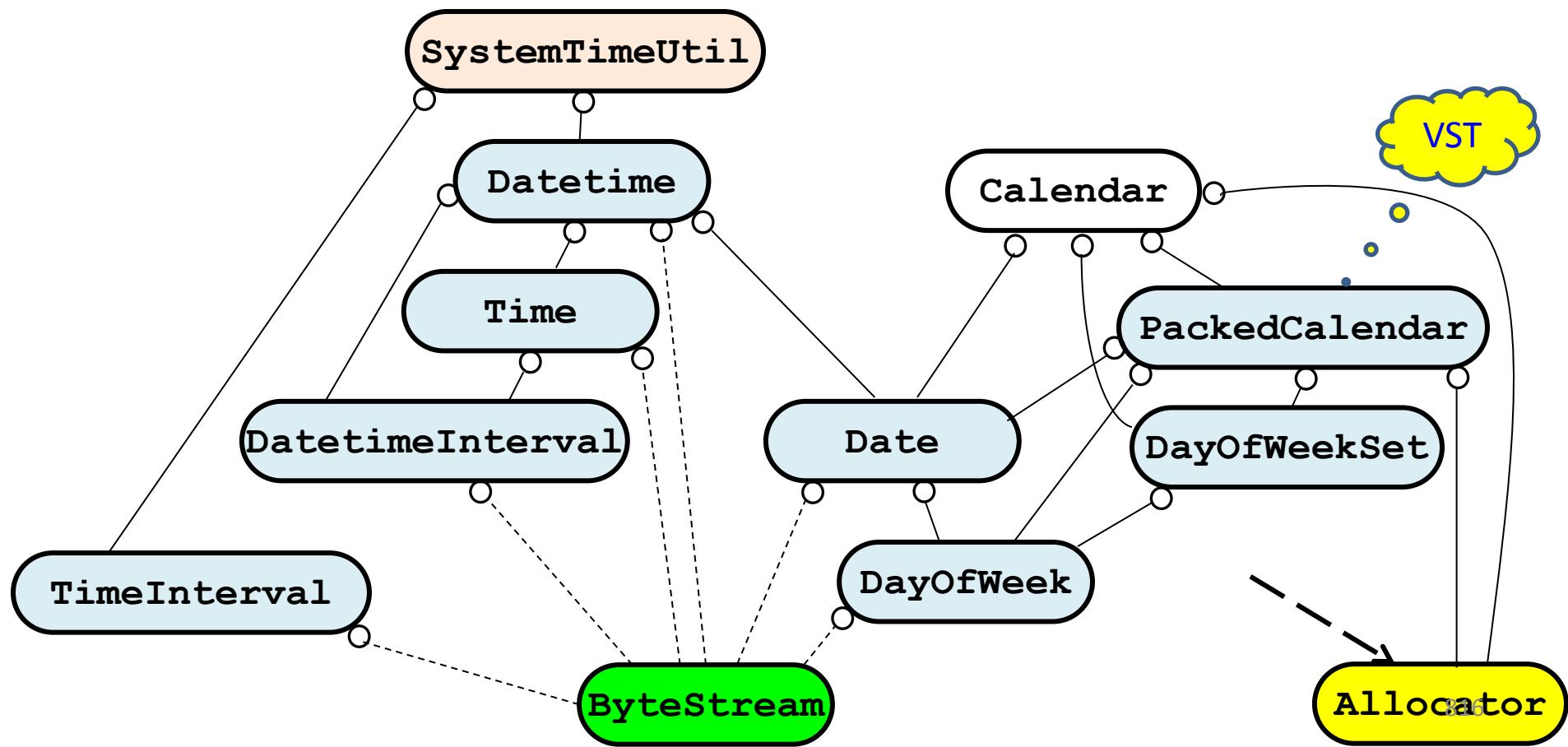
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



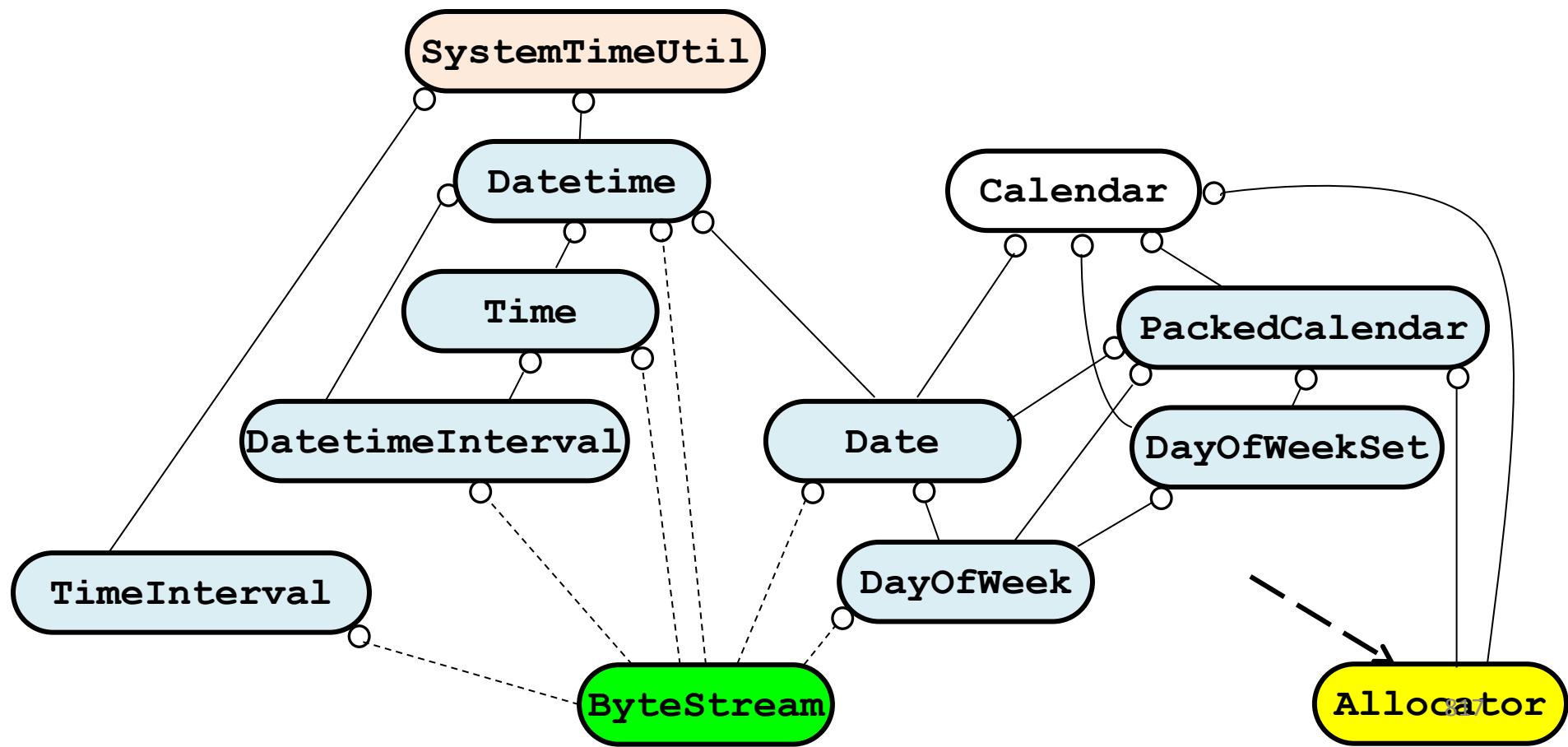
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



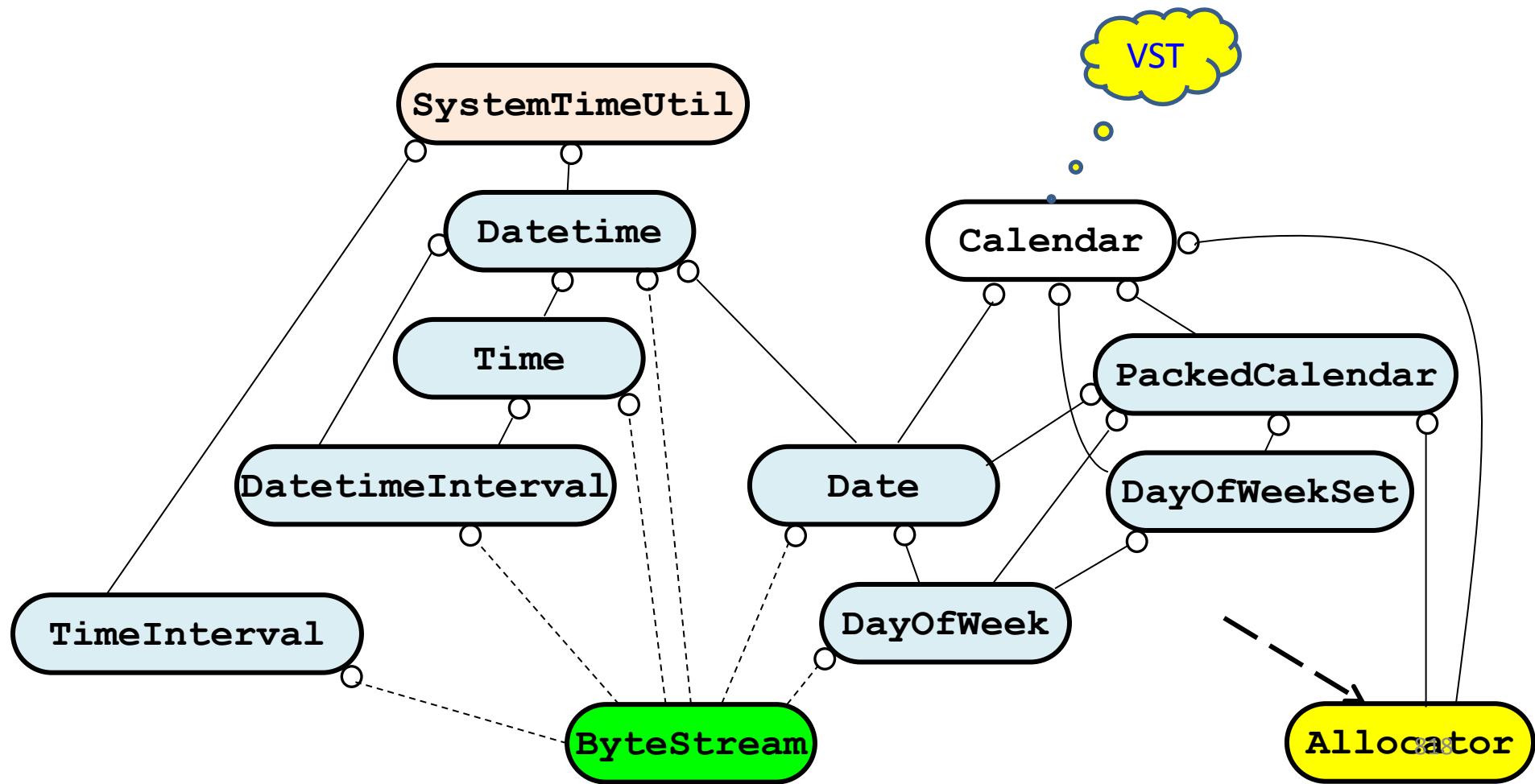
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



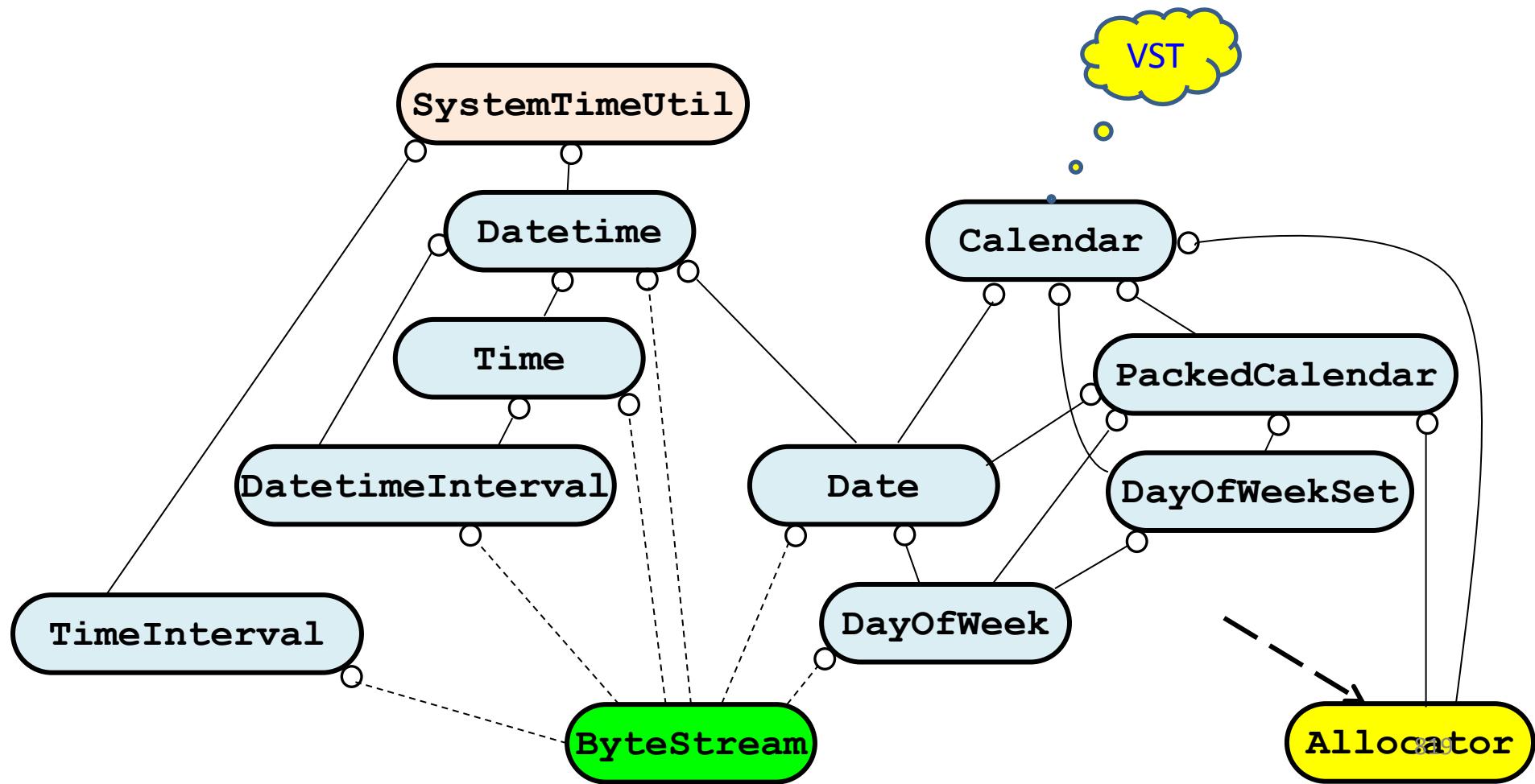
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



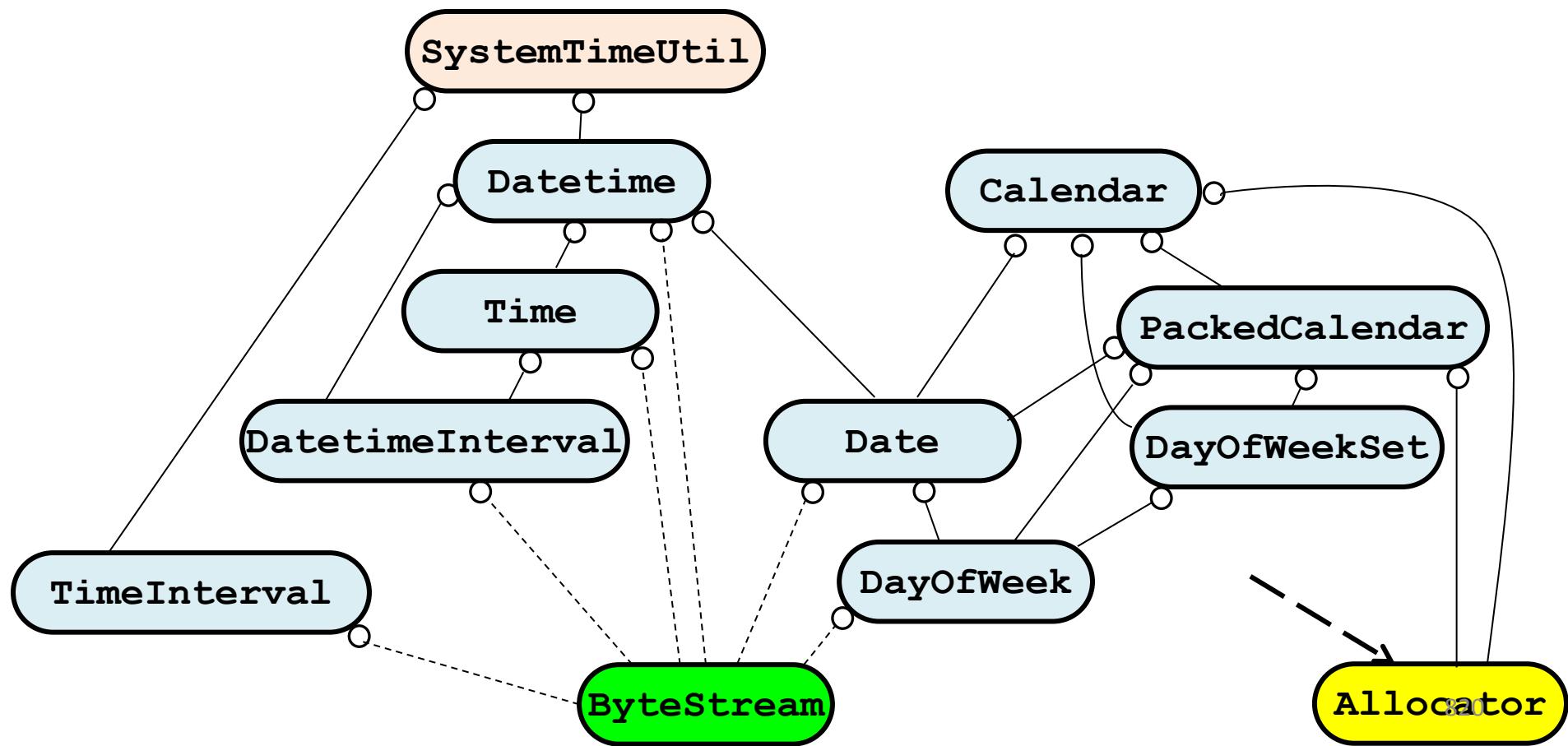
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



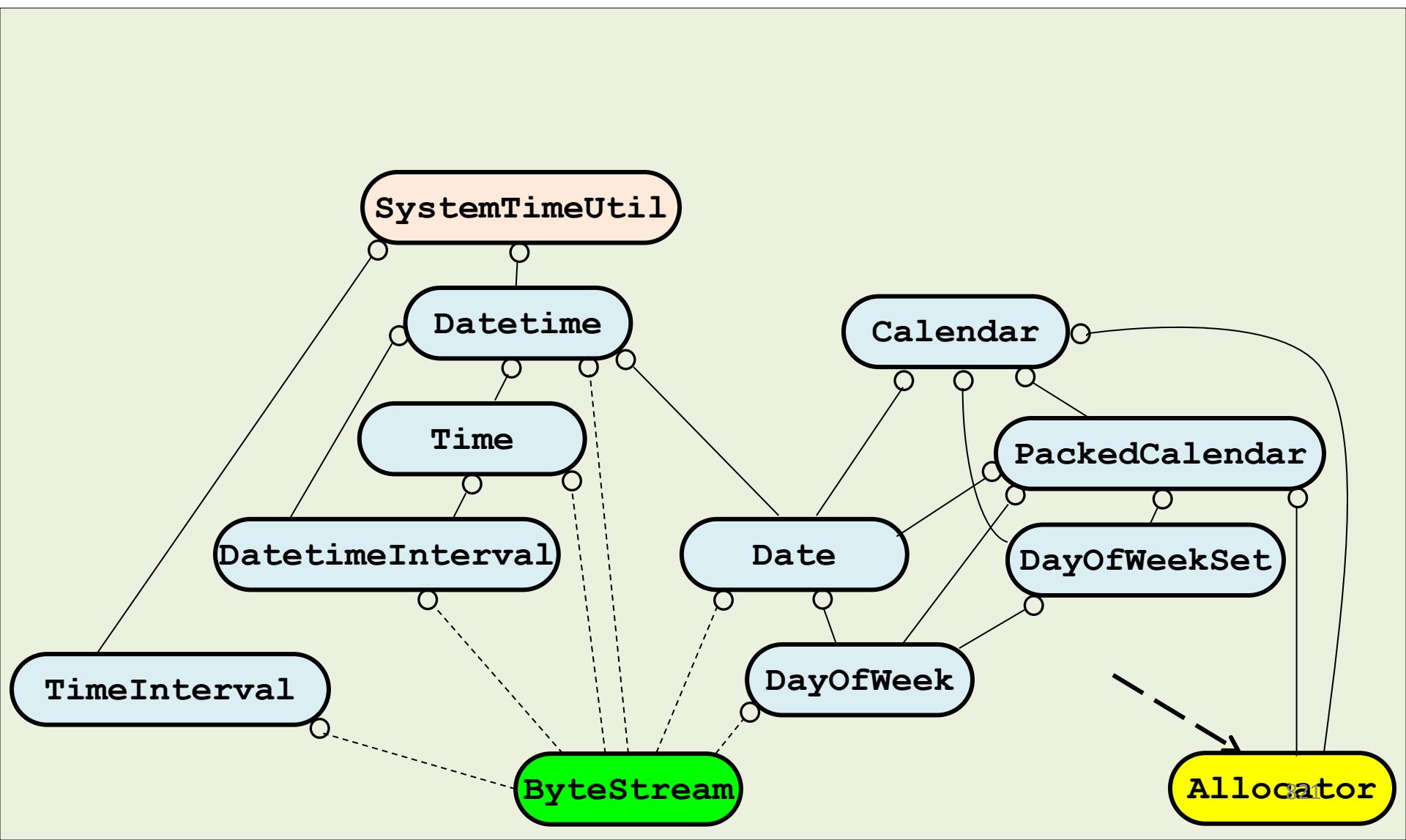
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



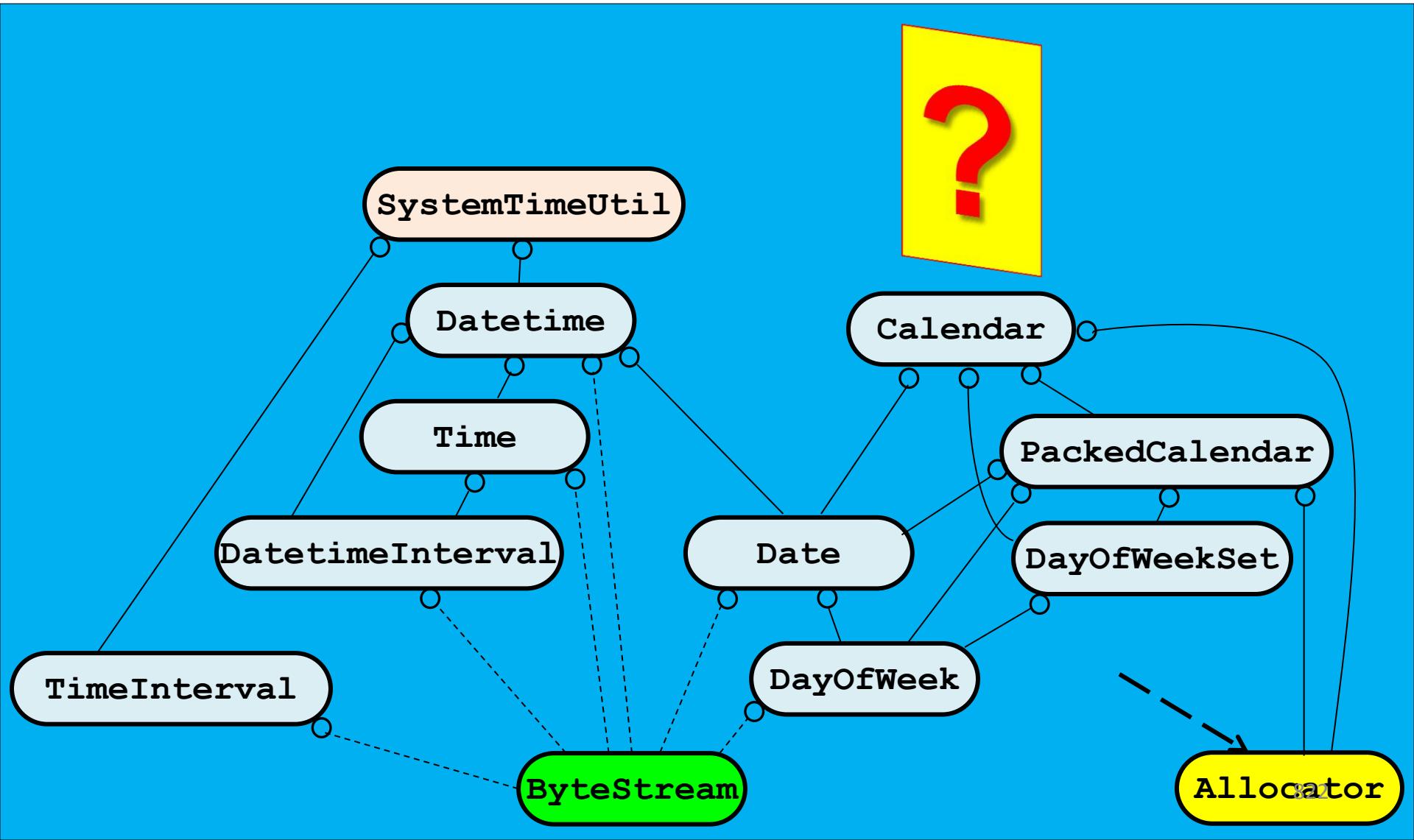
4. Bloomberg Development Environment

Determine if a Date Value is a *Business Day*



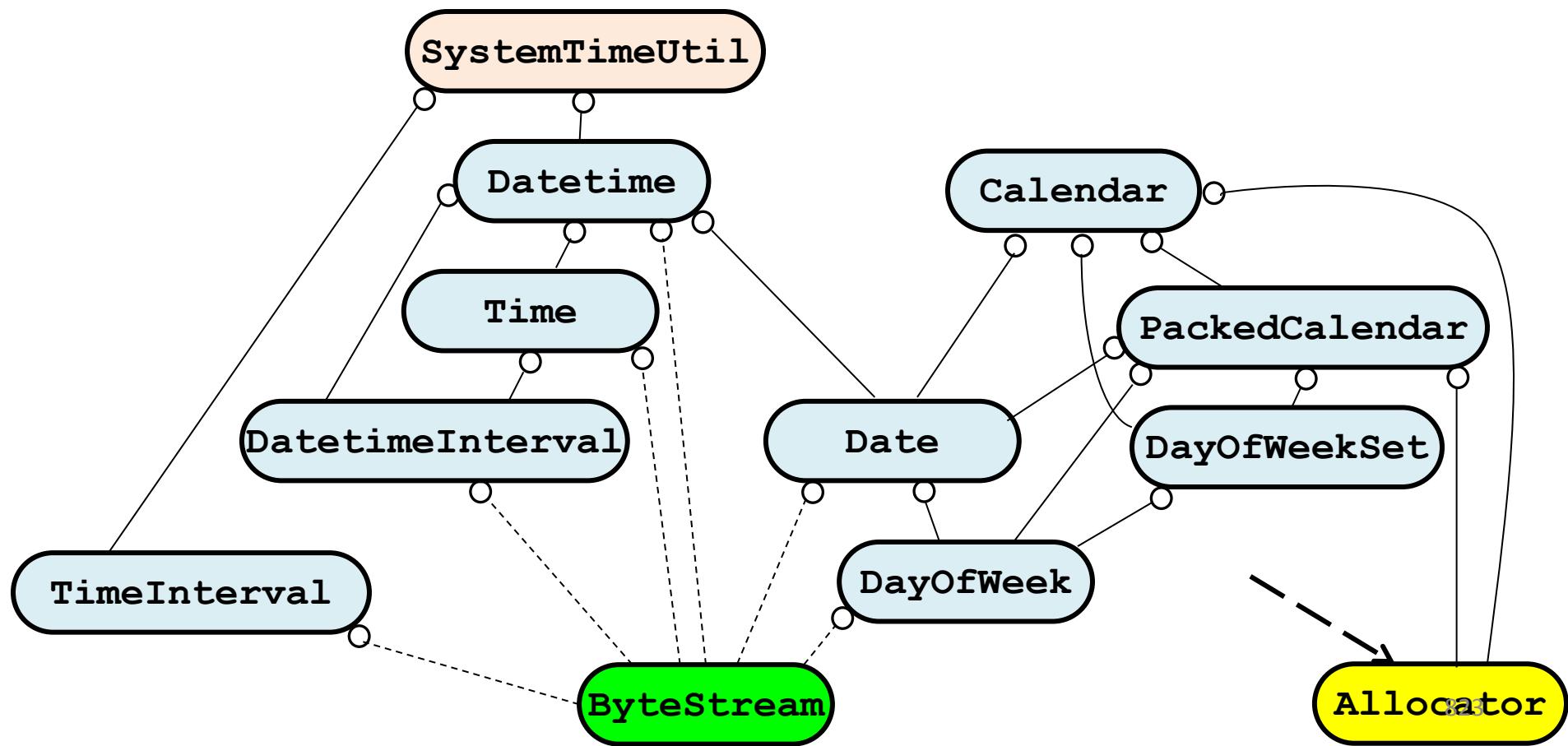
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



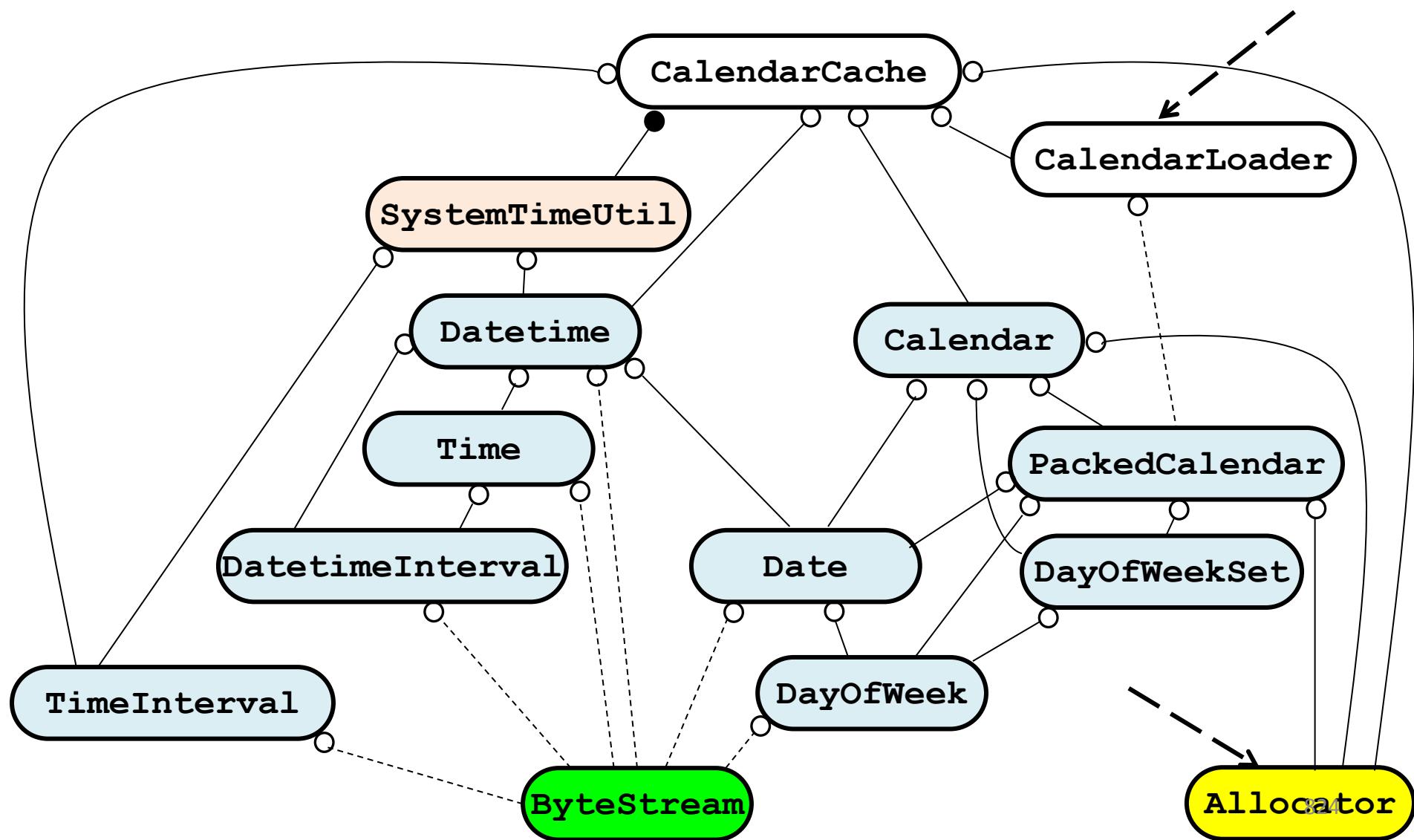
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



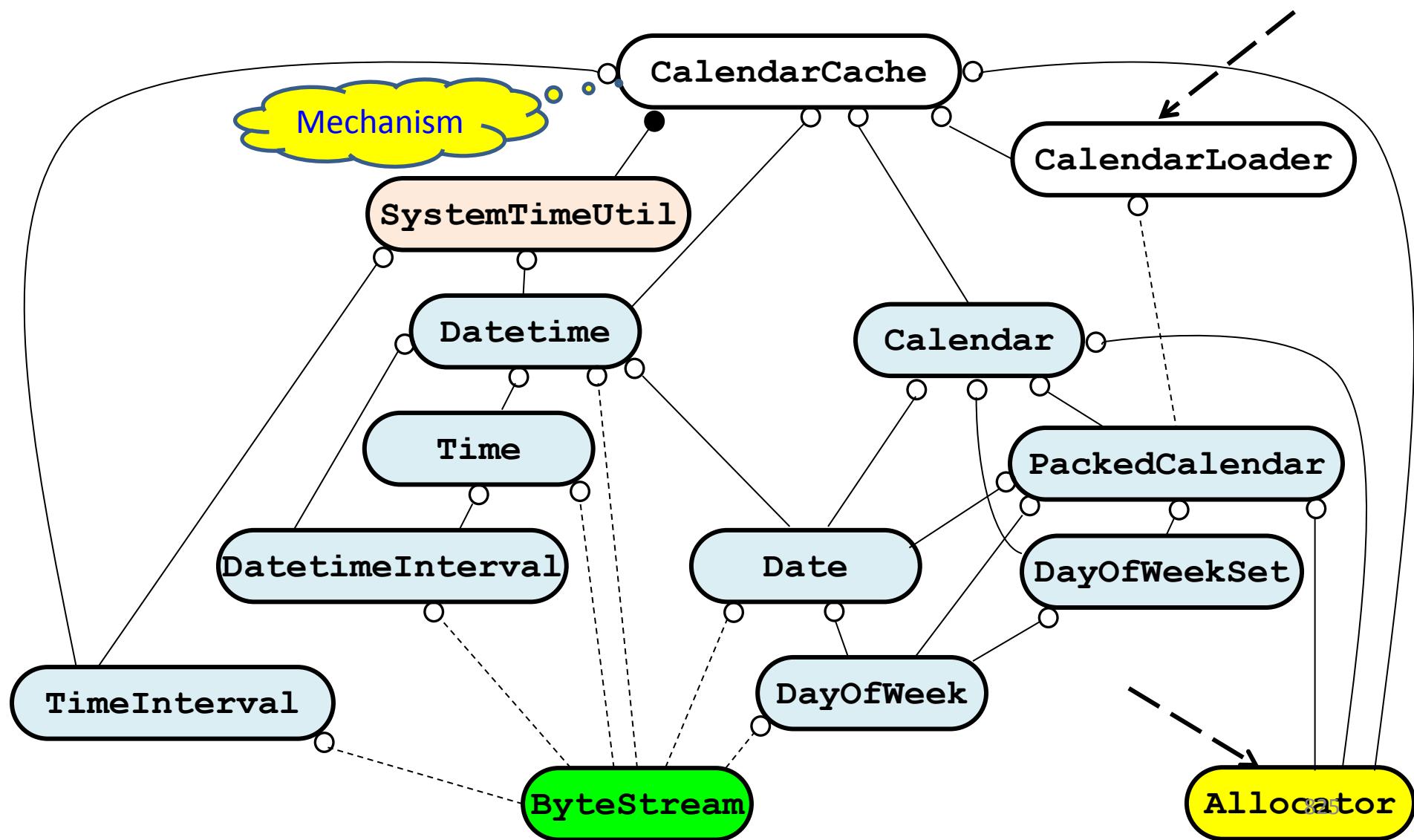
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



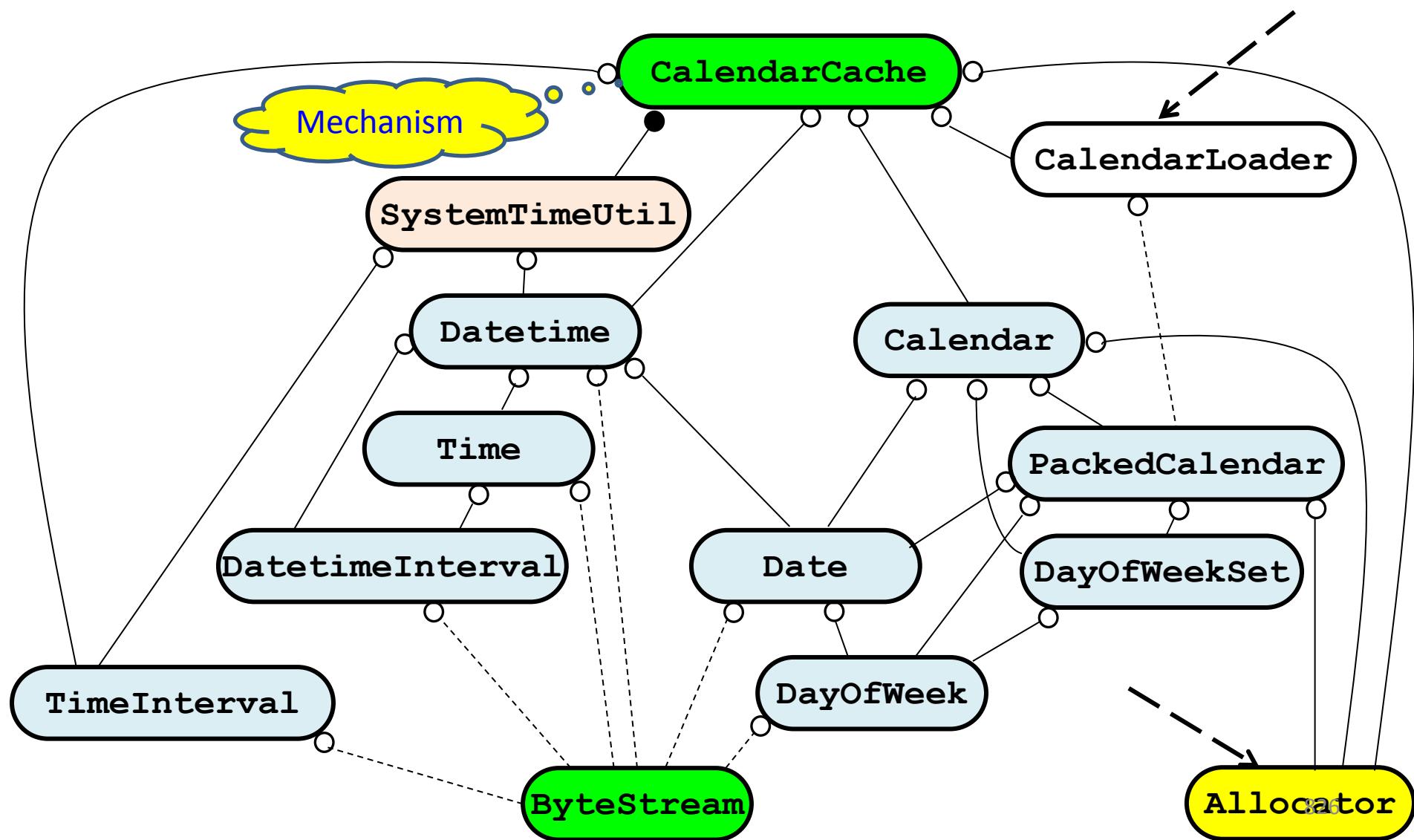
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



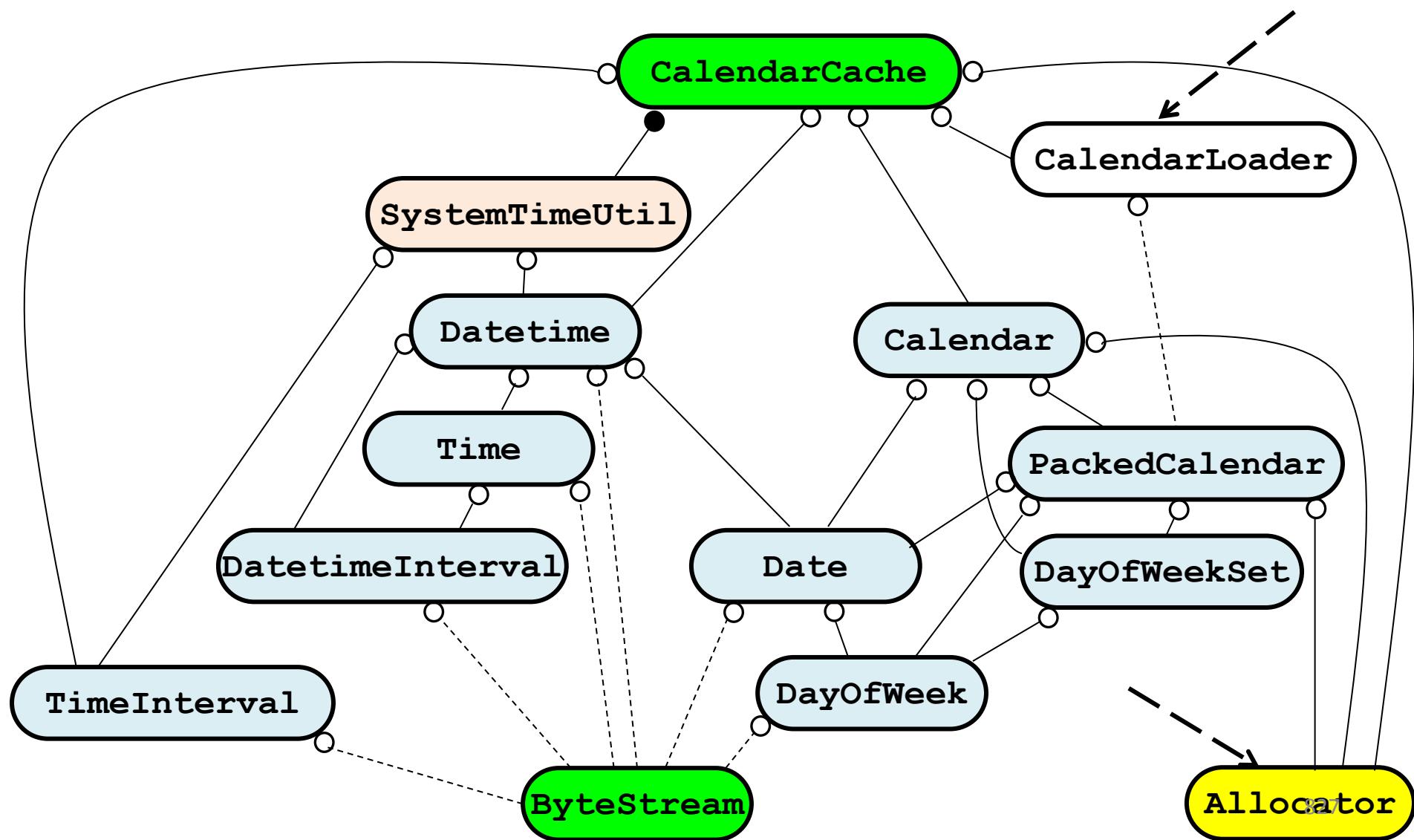
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



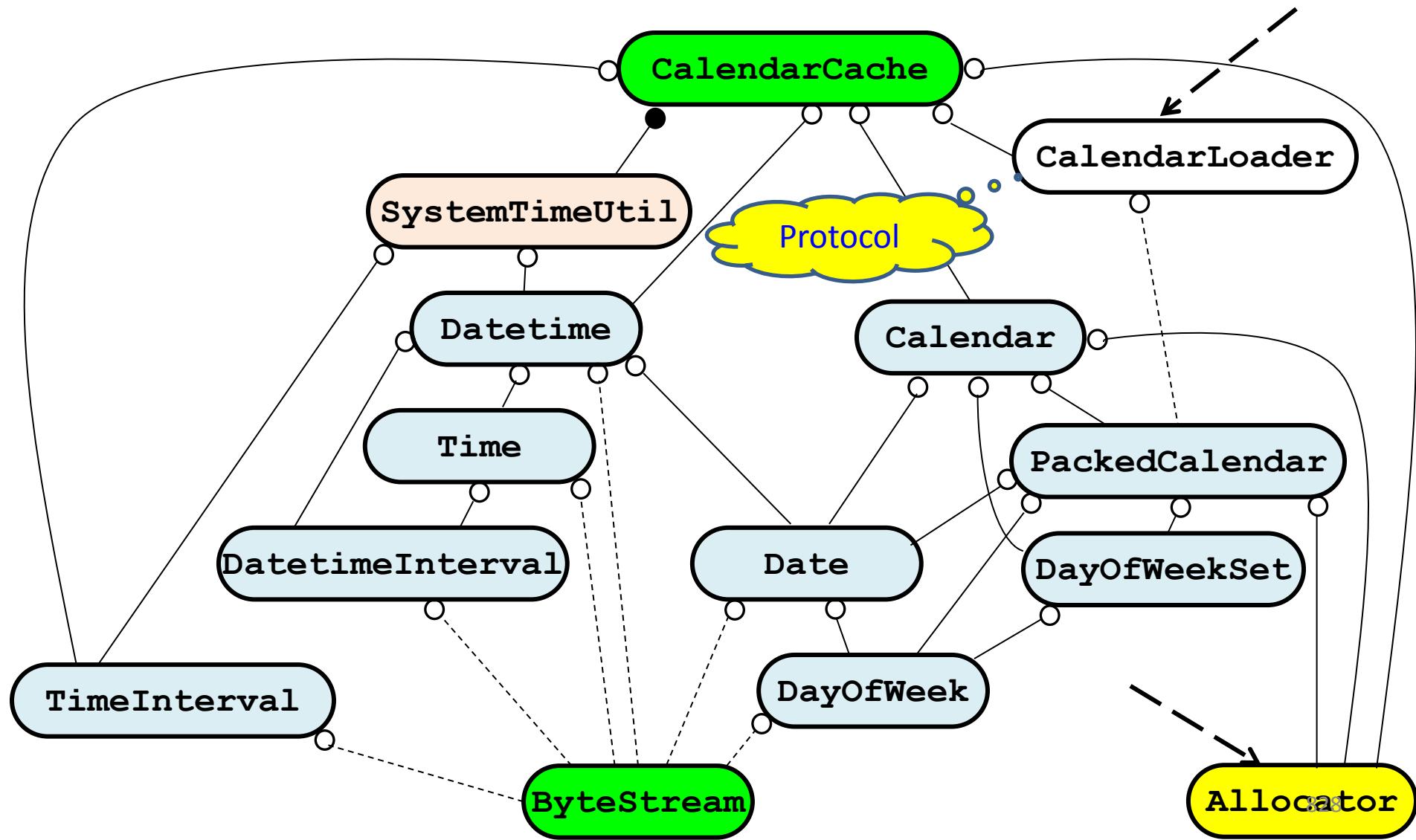
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



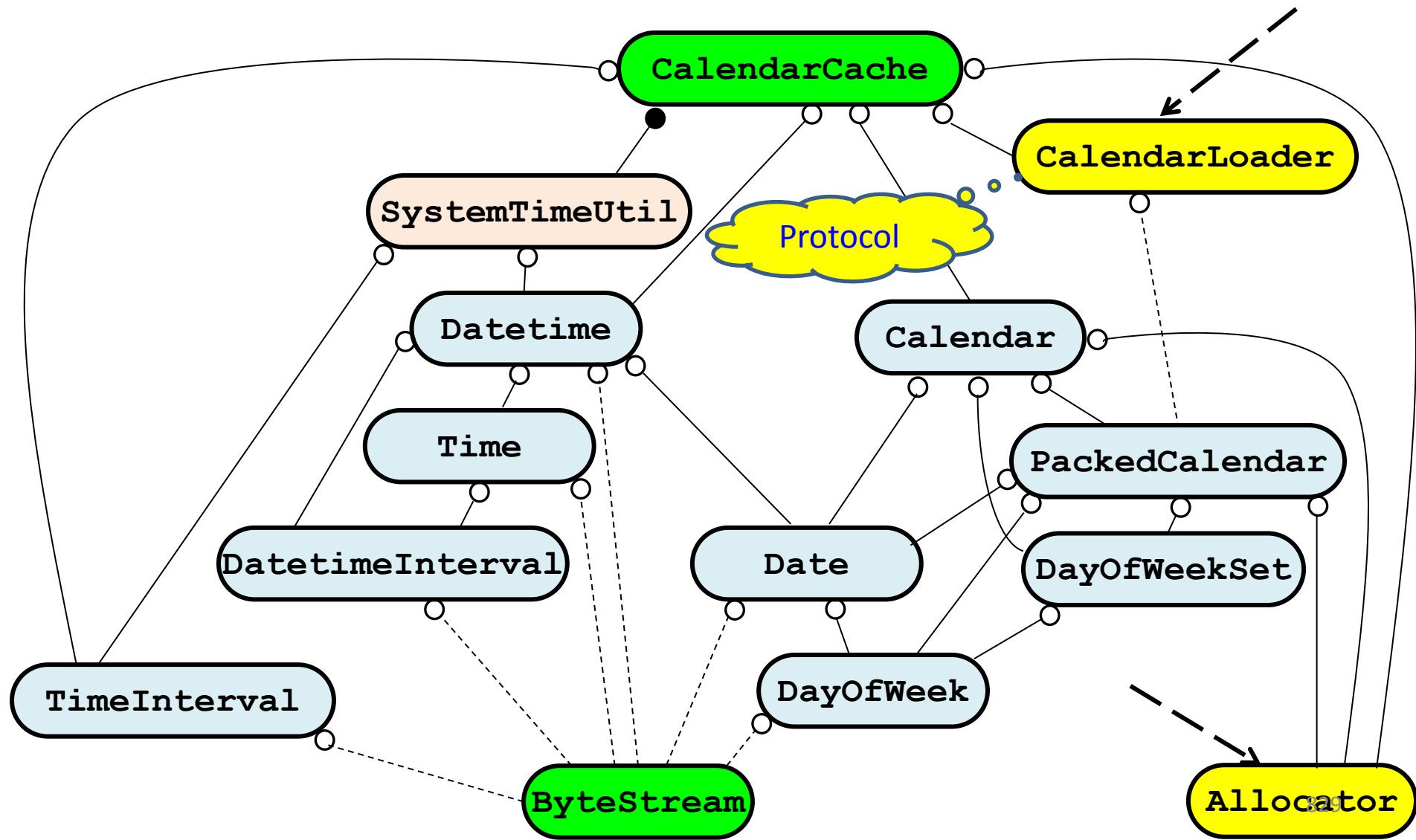
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



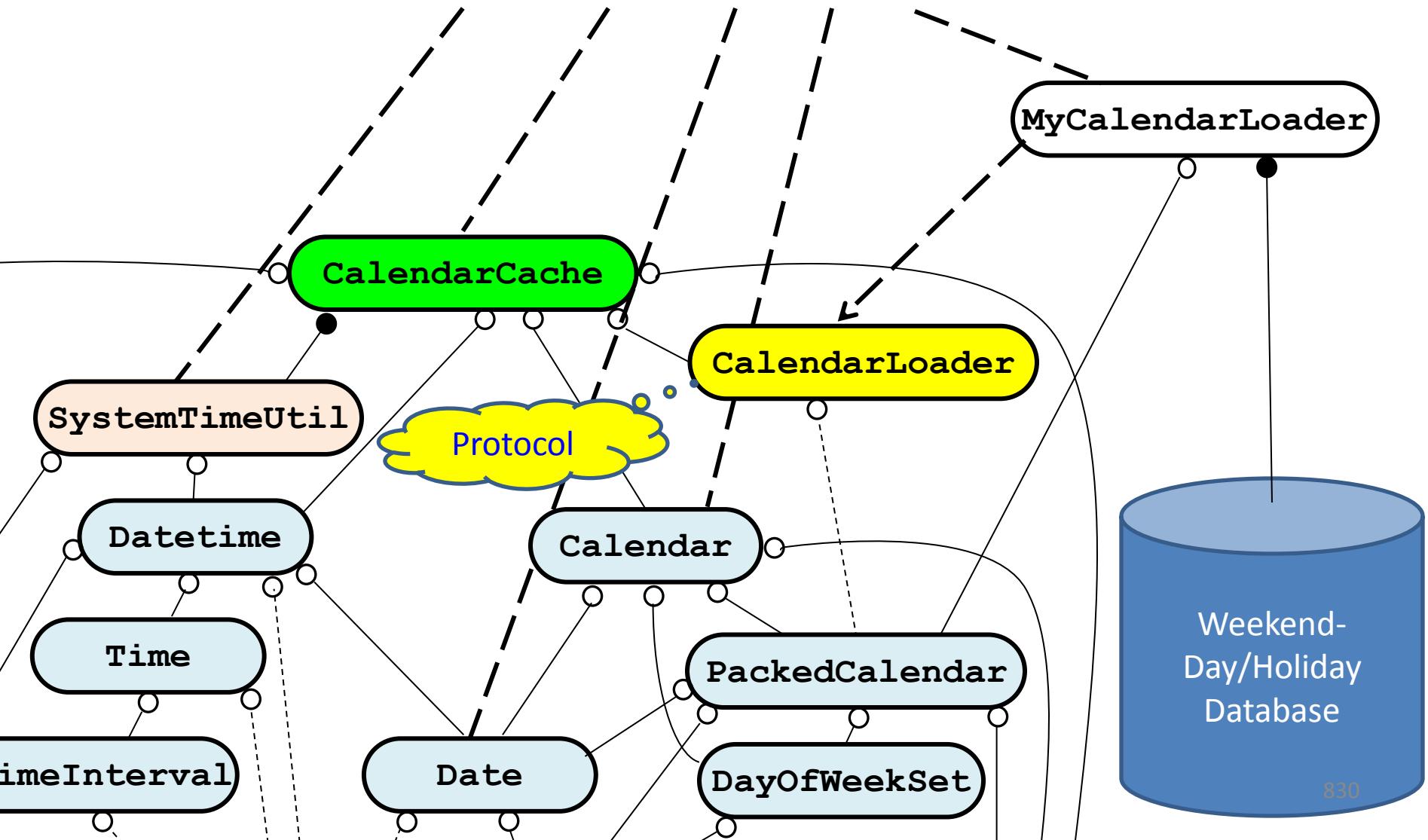
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



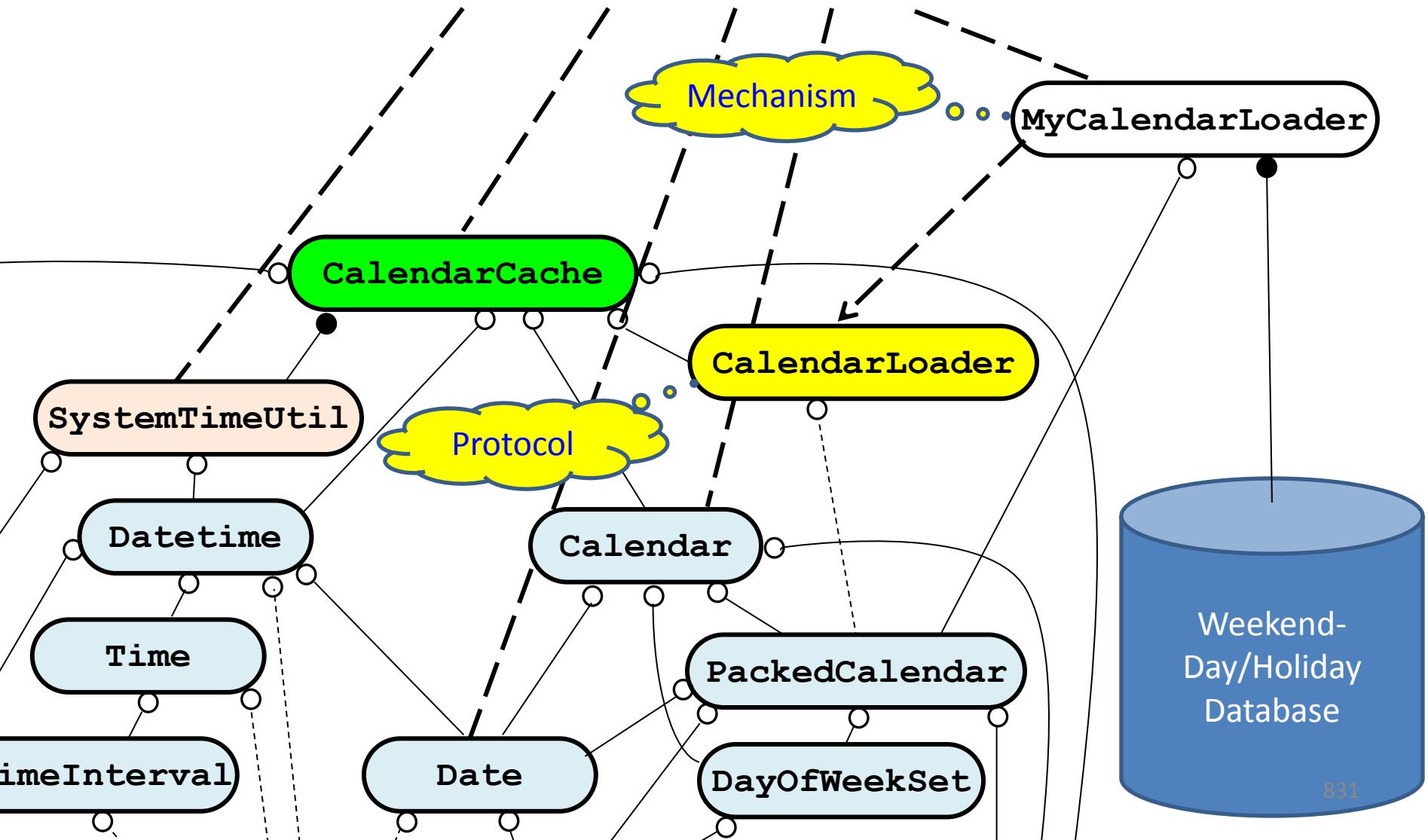
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



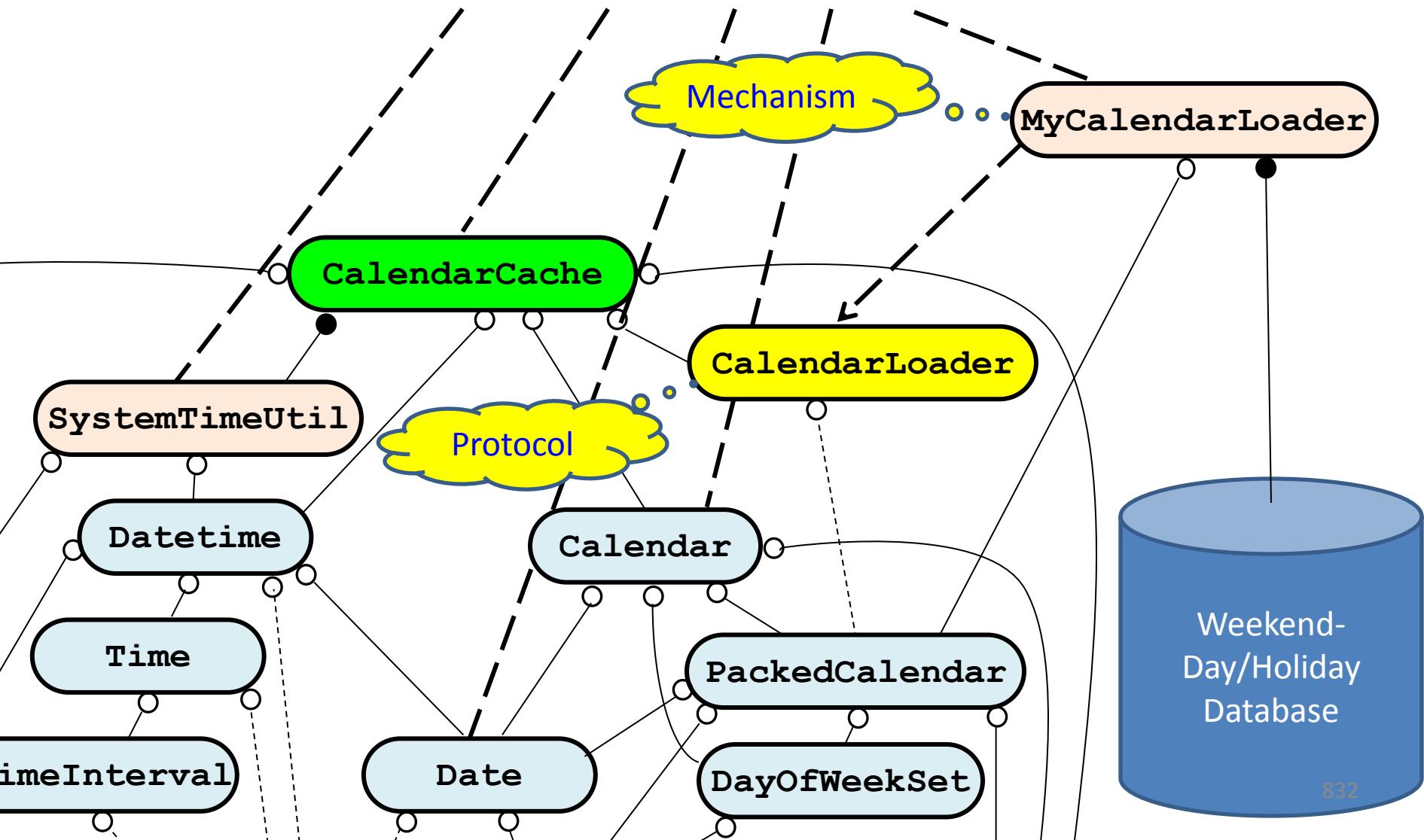
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



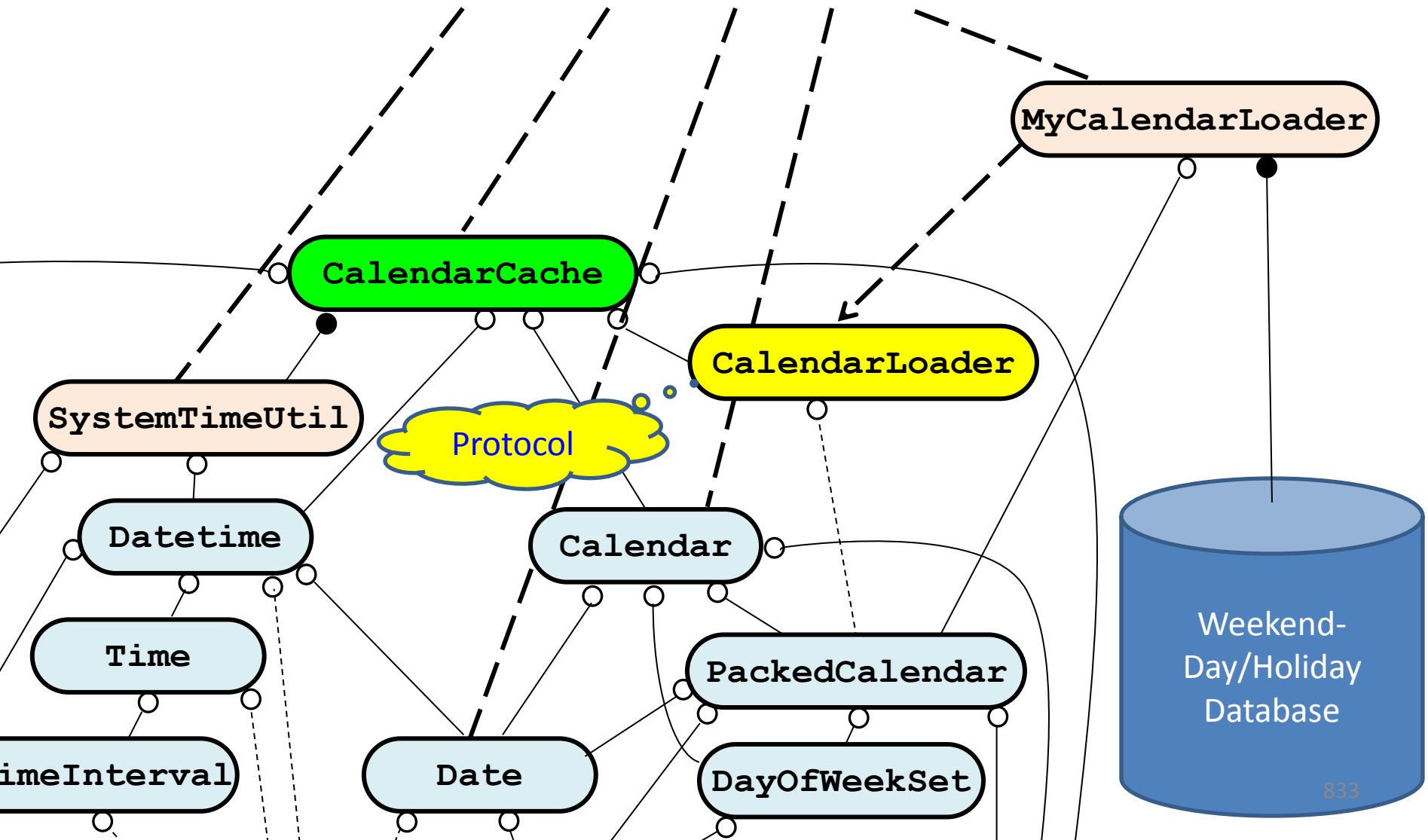
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



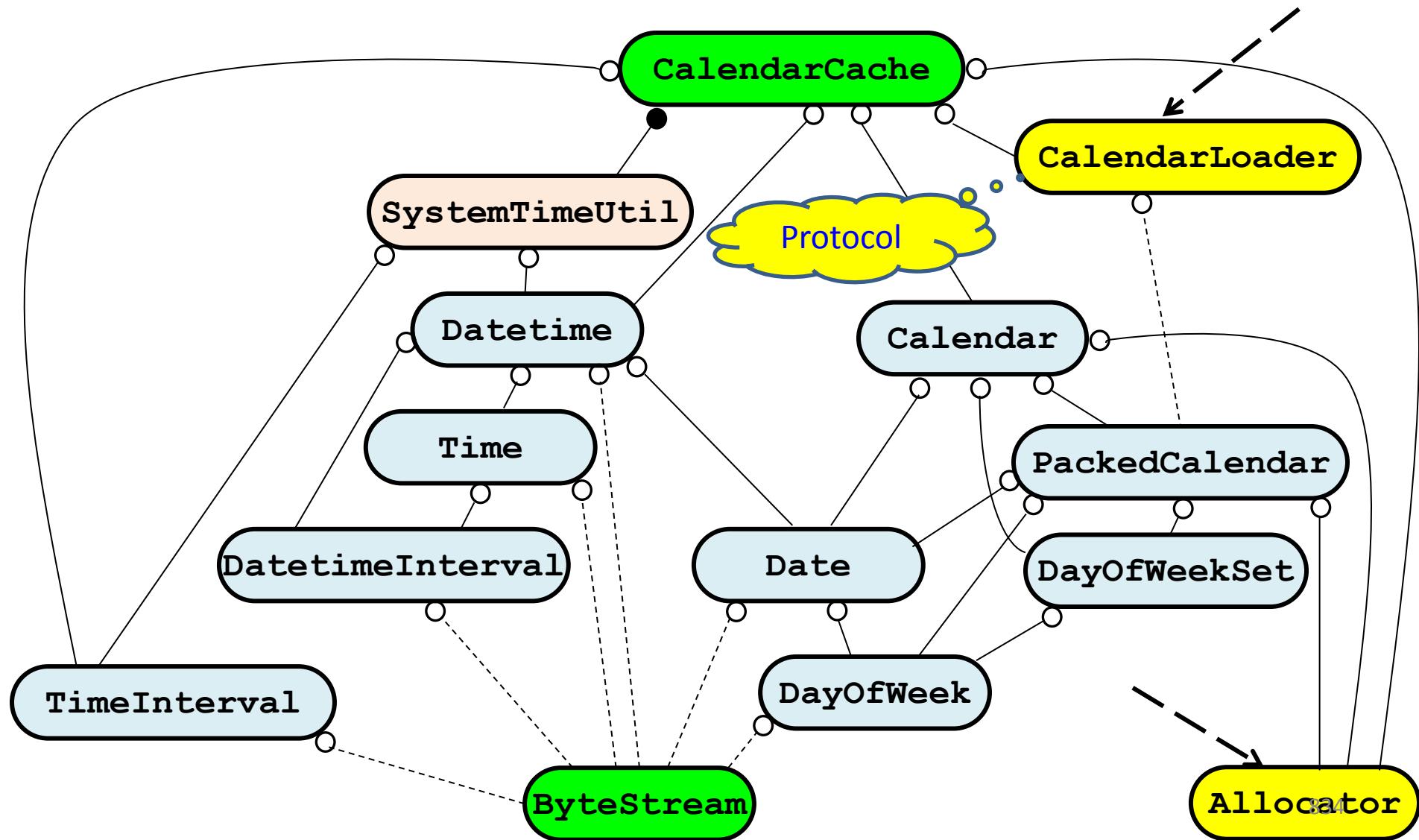
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



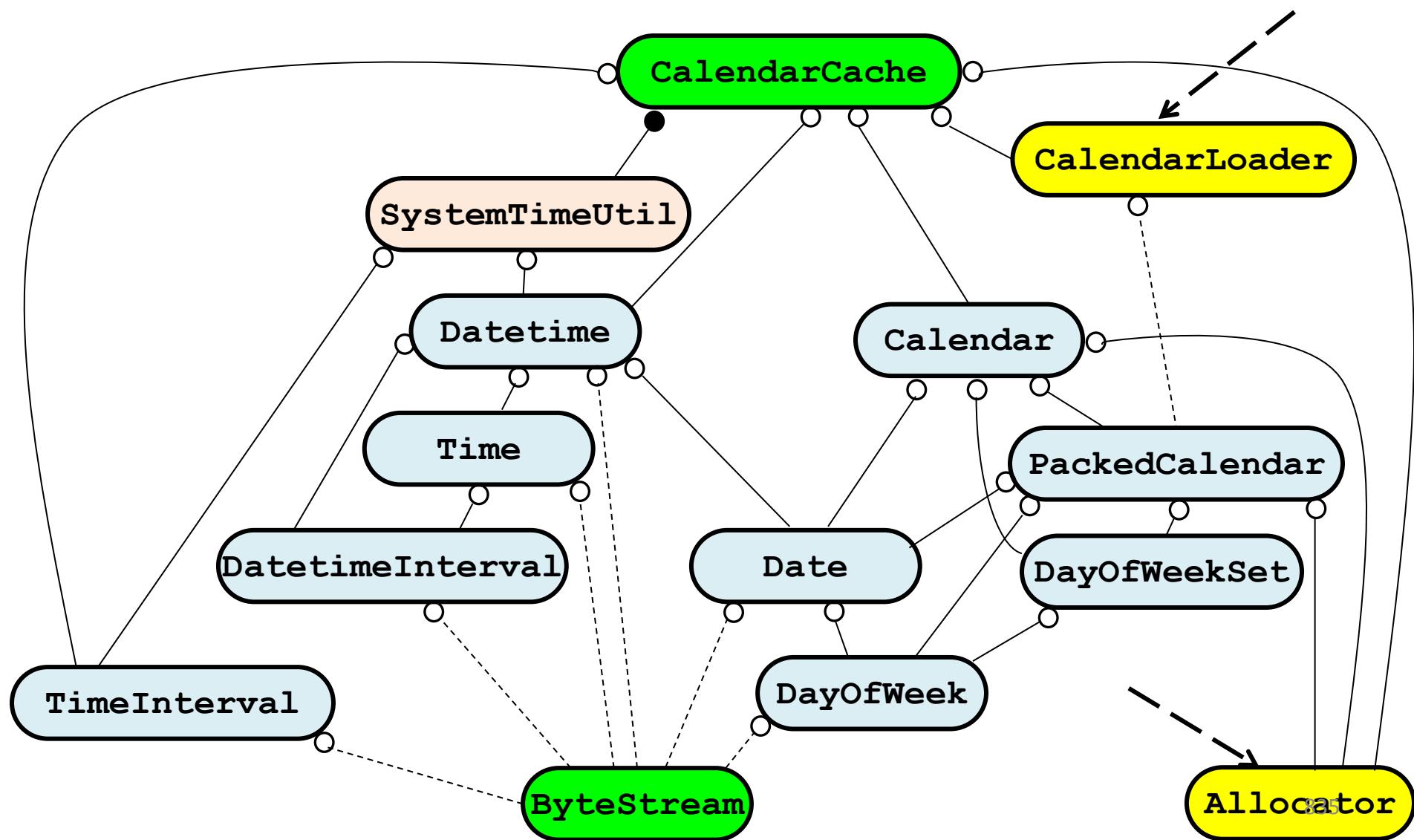
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



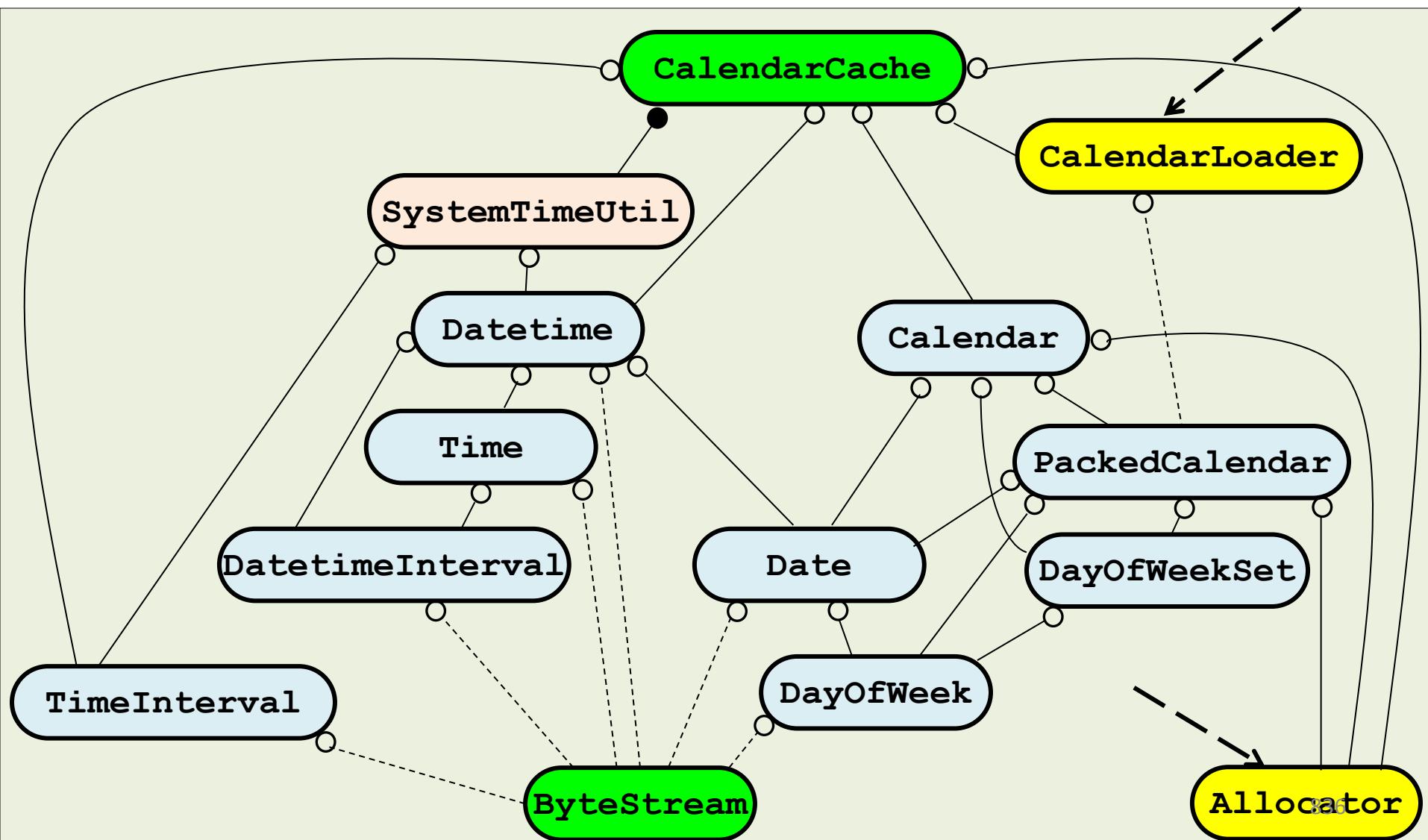
4. Bloomberg Development Environment

Wait a Minute: Where is the Data Source?



4. Bloomberg Development Environment

Solution 3: Is Date a Business Day?



4. Bloomberg Development Environment

The Original Request

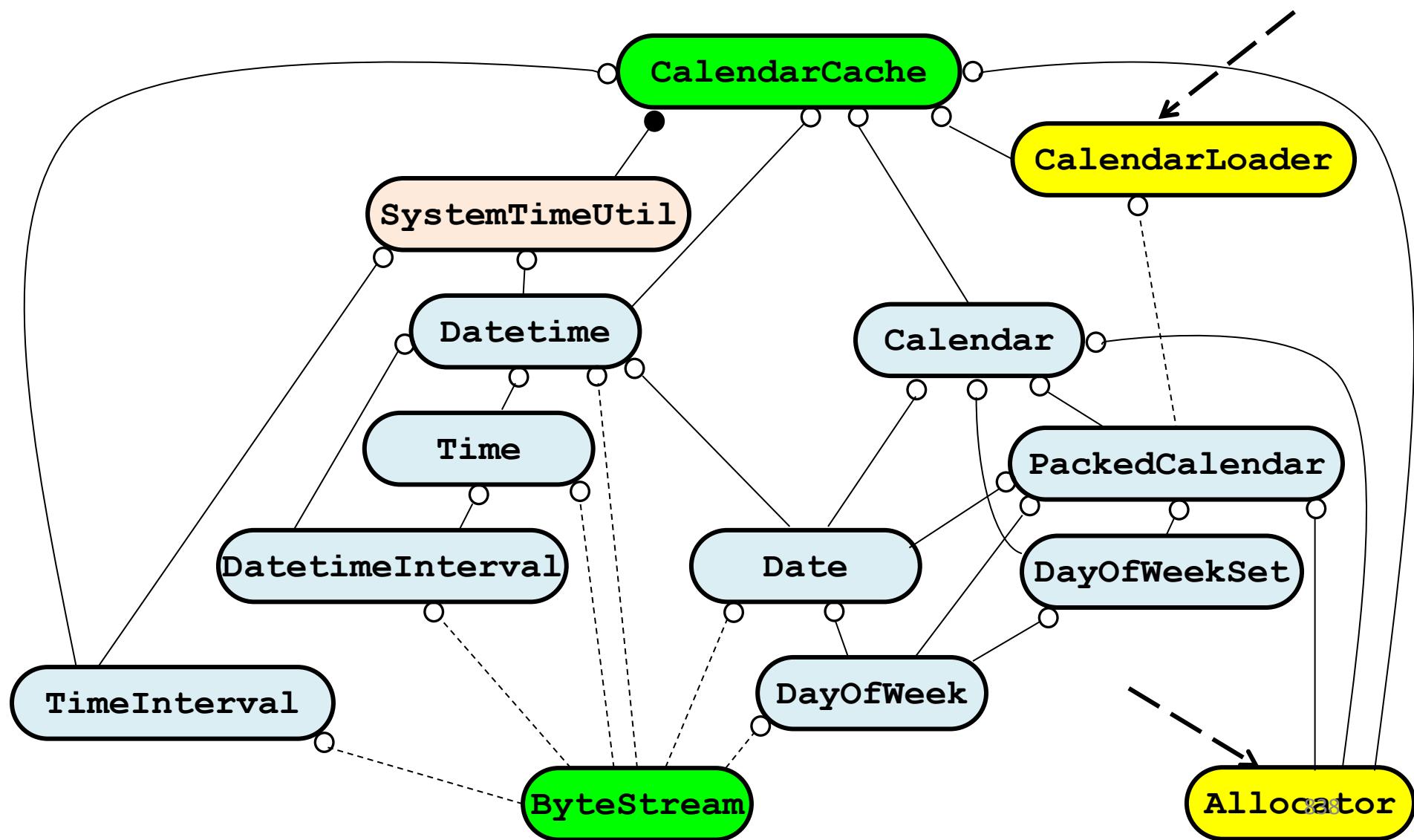
"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. Determine what date value *today* is.
3. Determine if a date value is a *business day*.
4. **Provide well-factored useful components that we'll need over and over again!**

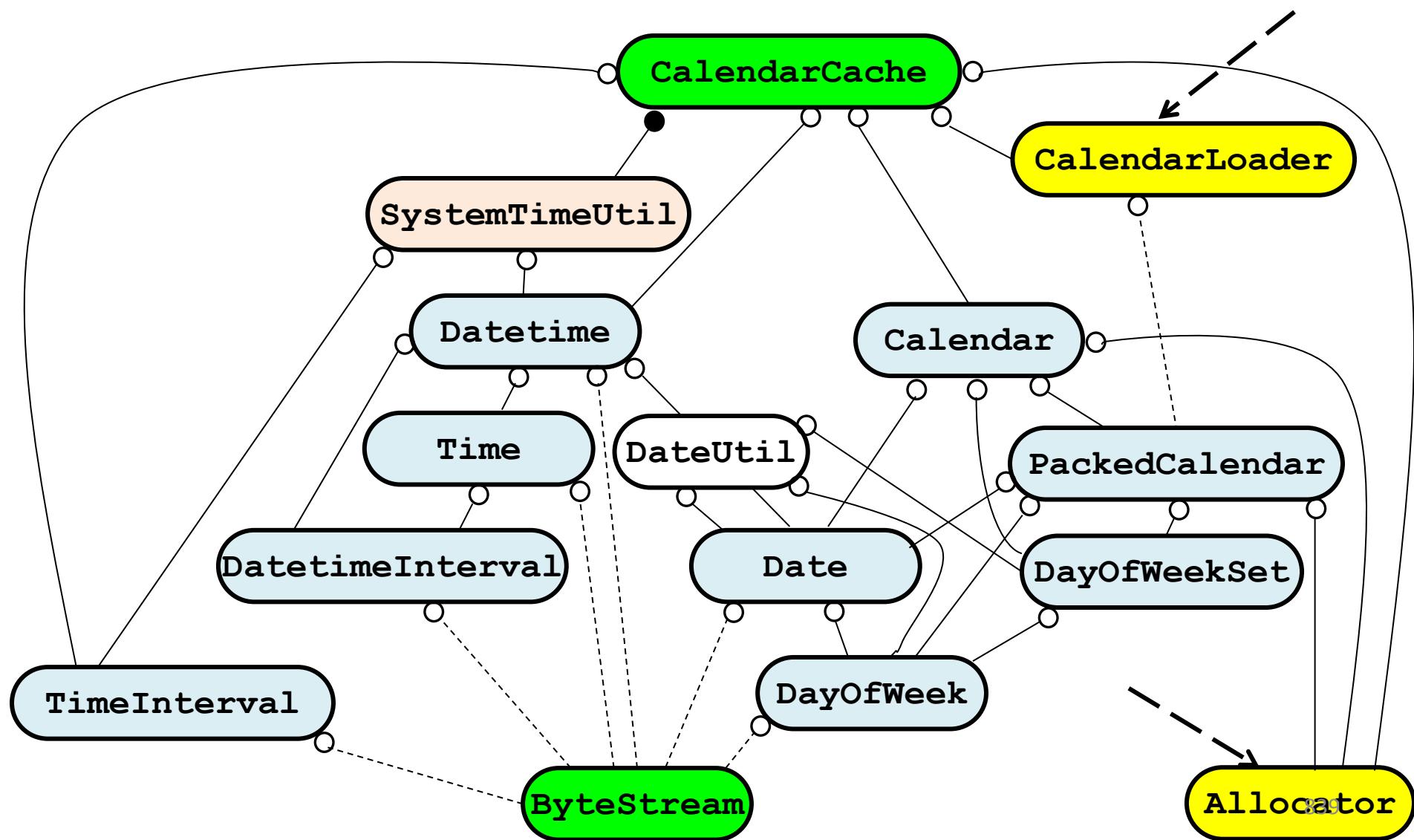
4. Bloomberg Development Environment

Non-Primitive Functionality



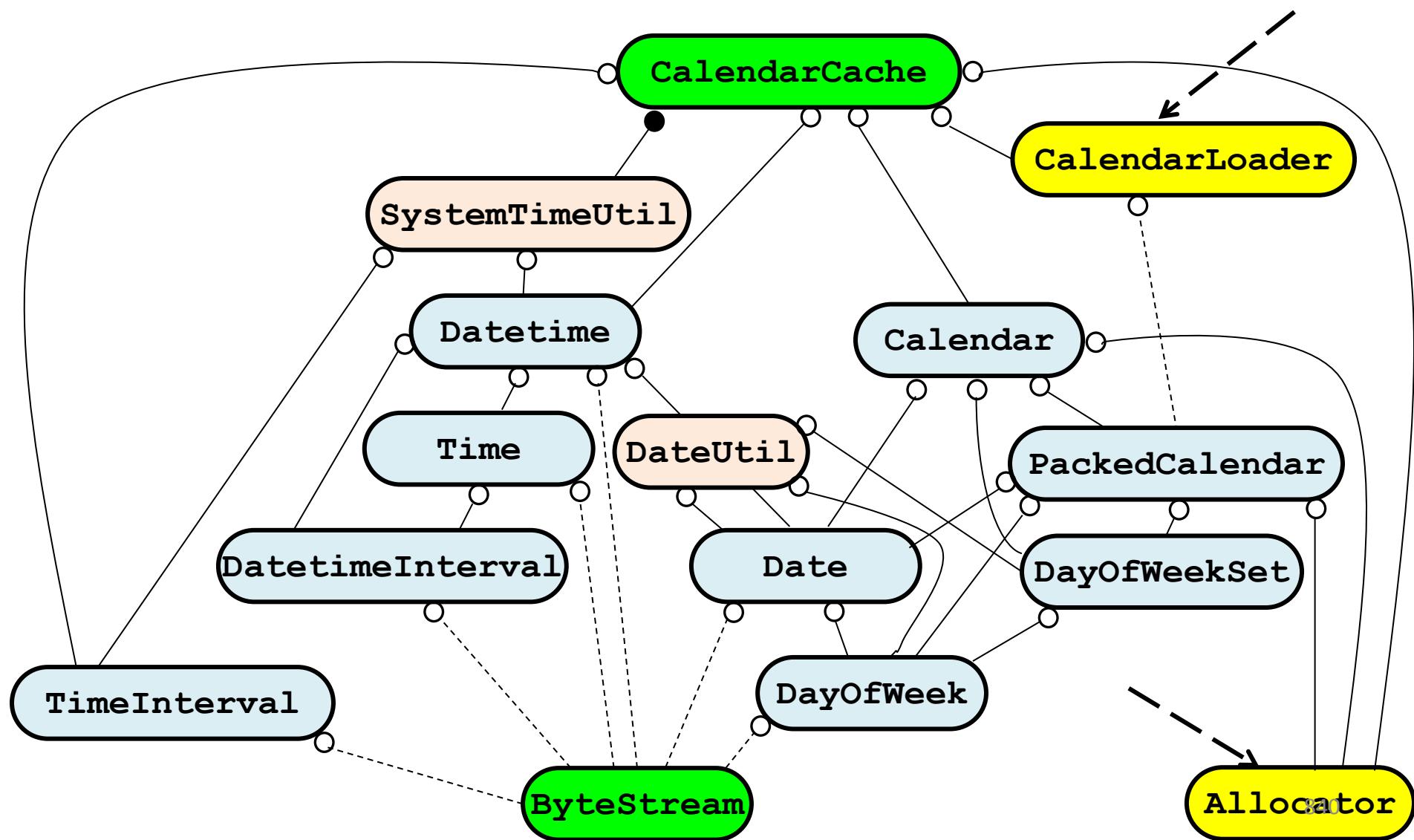
4. Bloomberg Development Environment

Non-Primitive Functionality



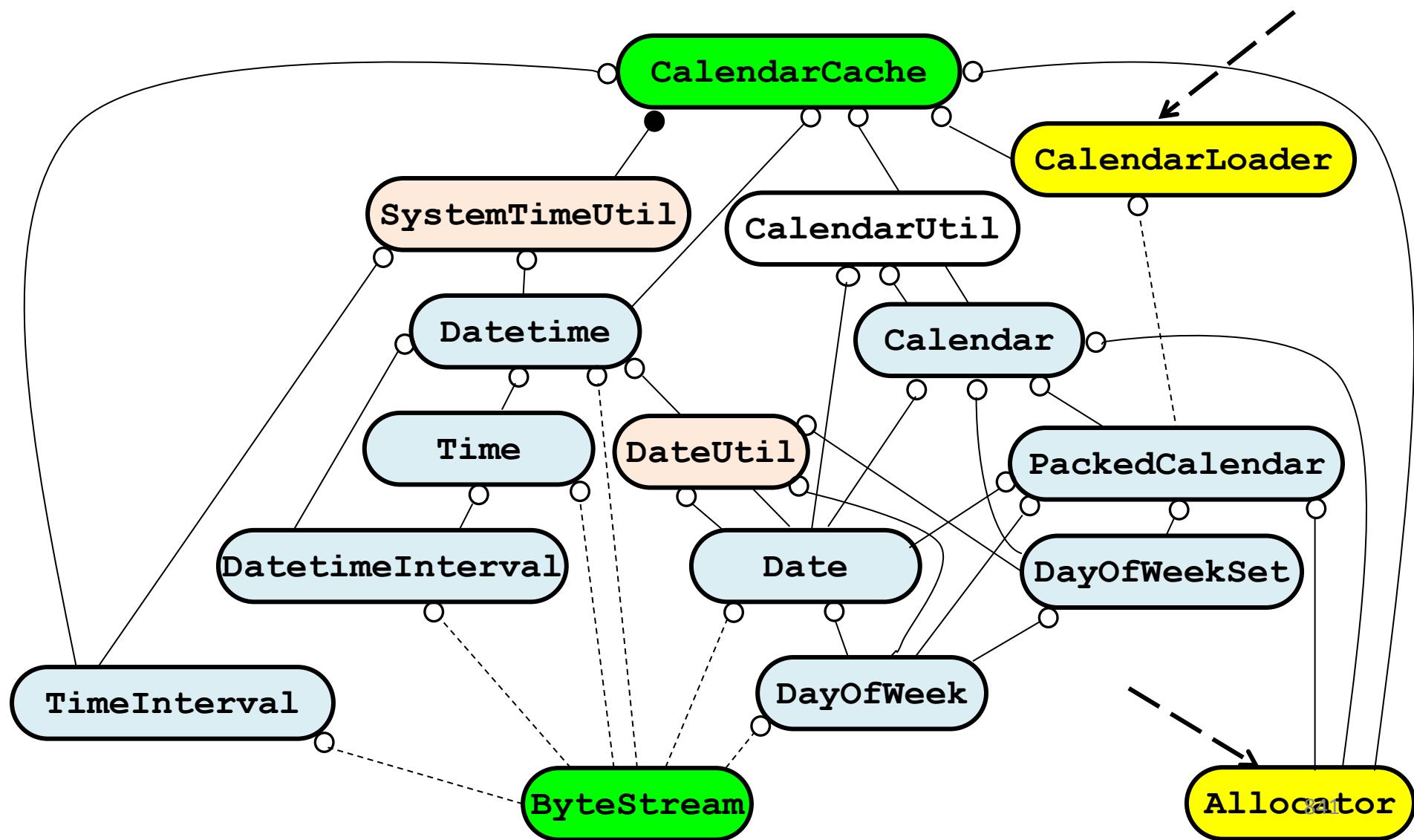
4. Bloomberg Development Environment

Non-Primitive Functionality



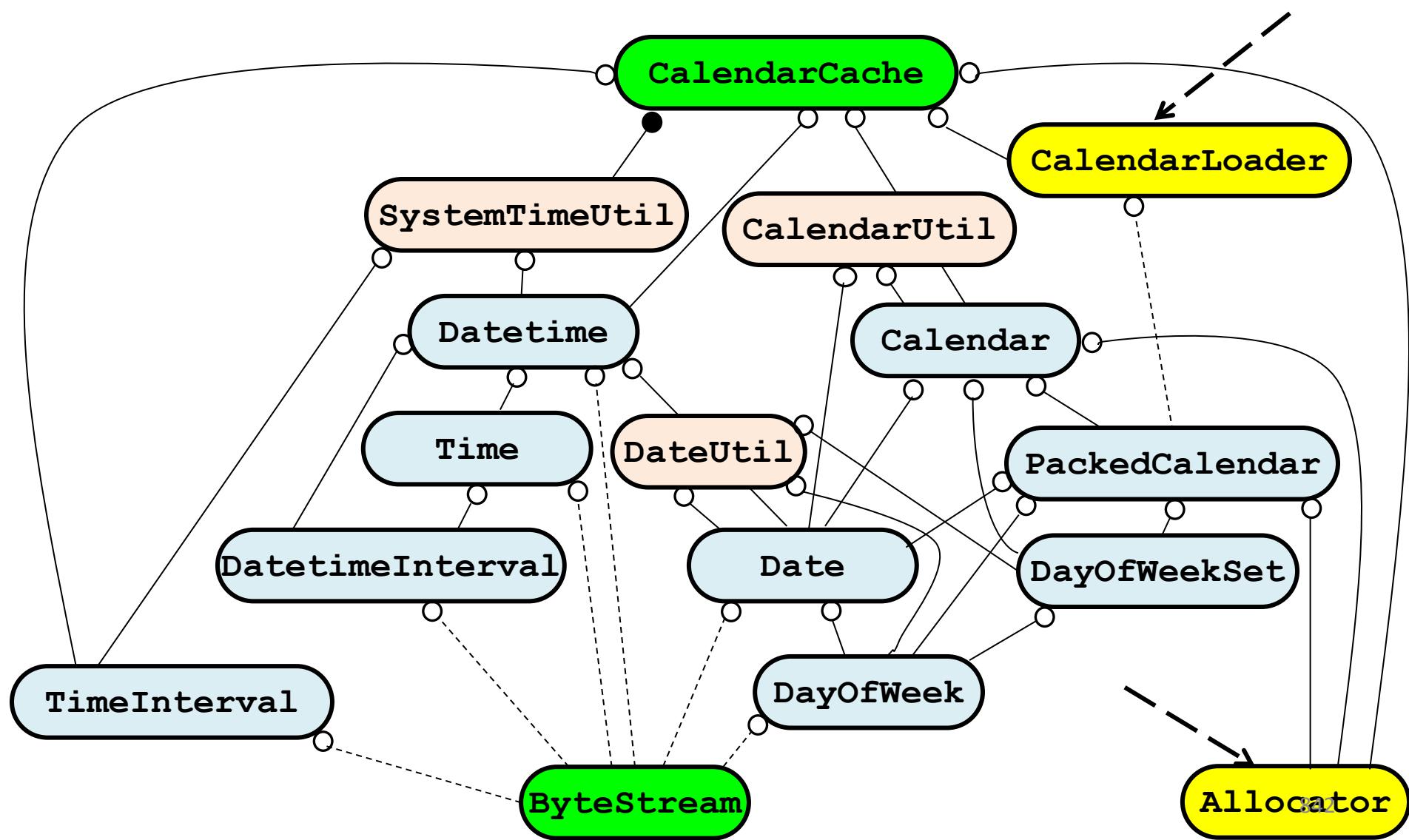
4. Bloomberg Development Environment

Non-Primitive Functionality



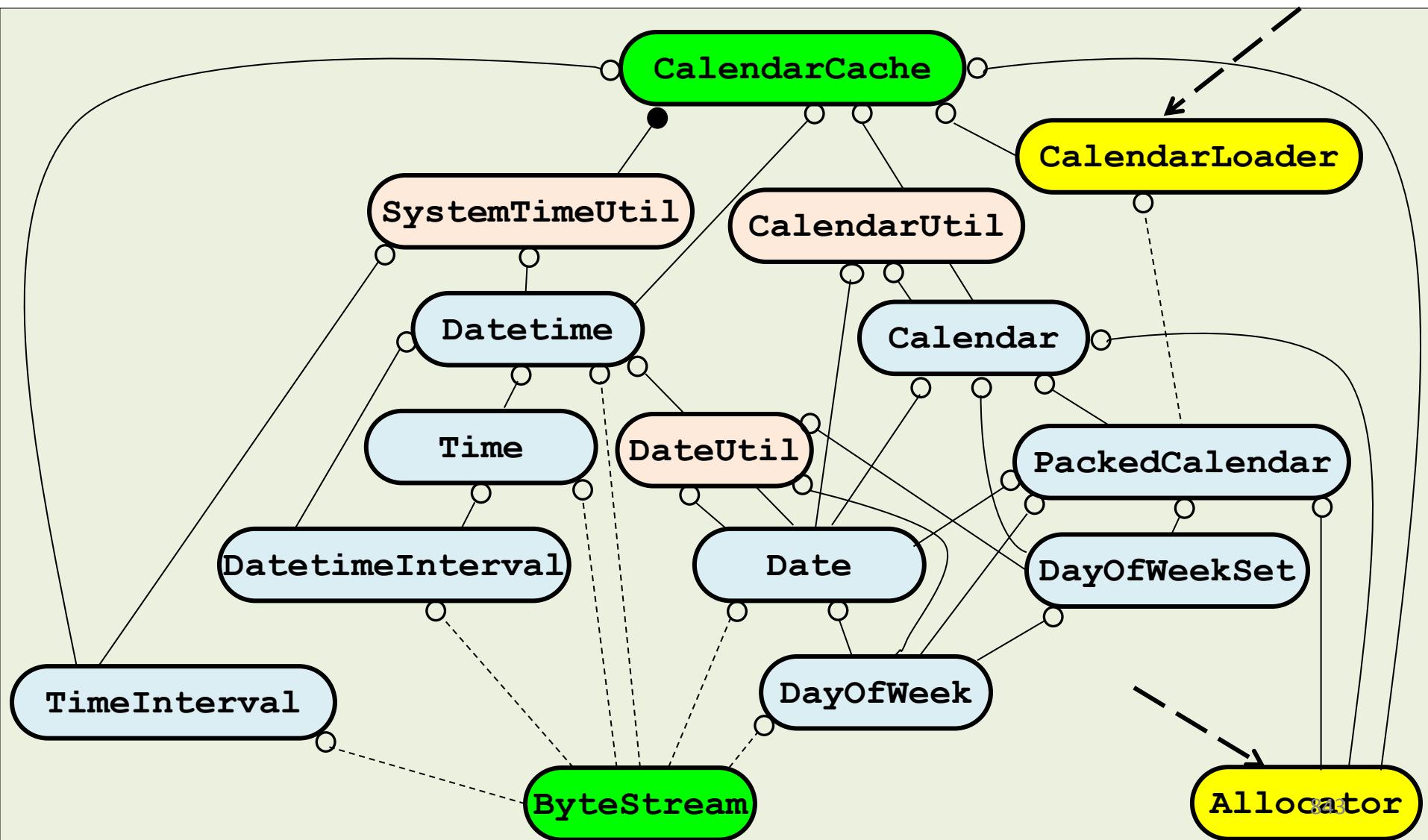
4. Bloomberg Development Environment

Non-Primitive Functionality



4. Bloomberg Development Environment

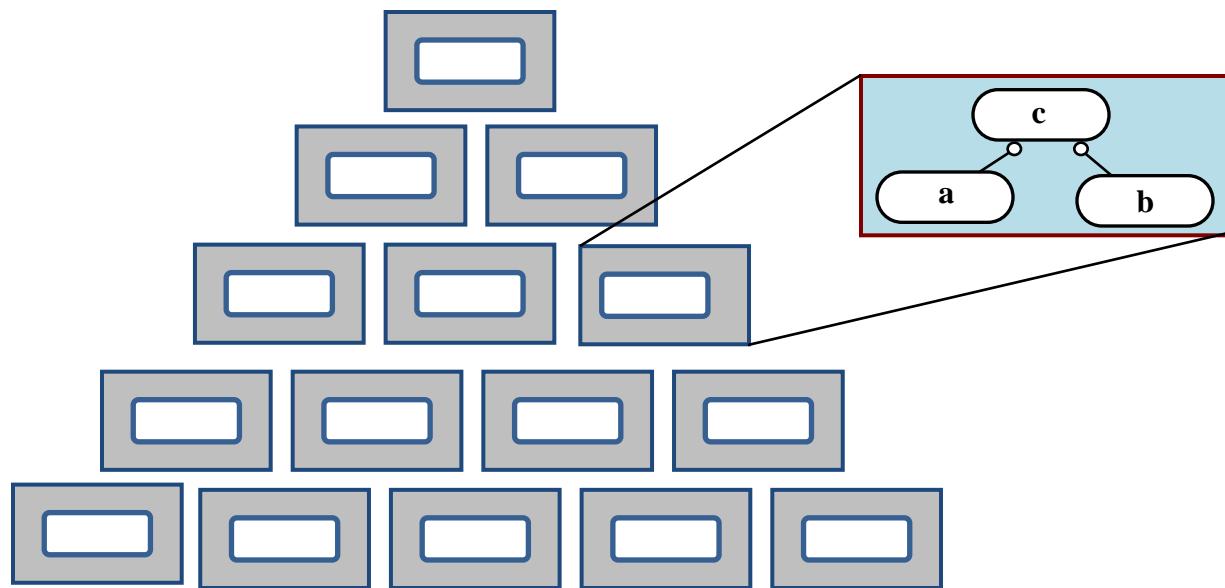
Fine-Grained Reusable Class Design



4. Bloomberg Development Environment

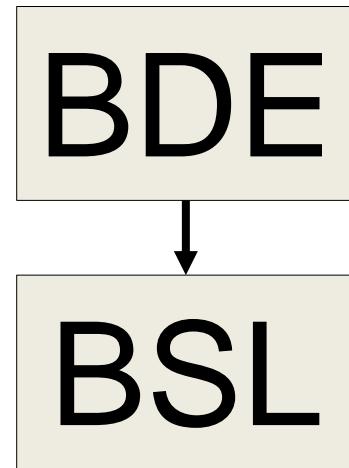
Rendering Software as Components

Logical content aggregated into a
Physical hierarchy of **components**



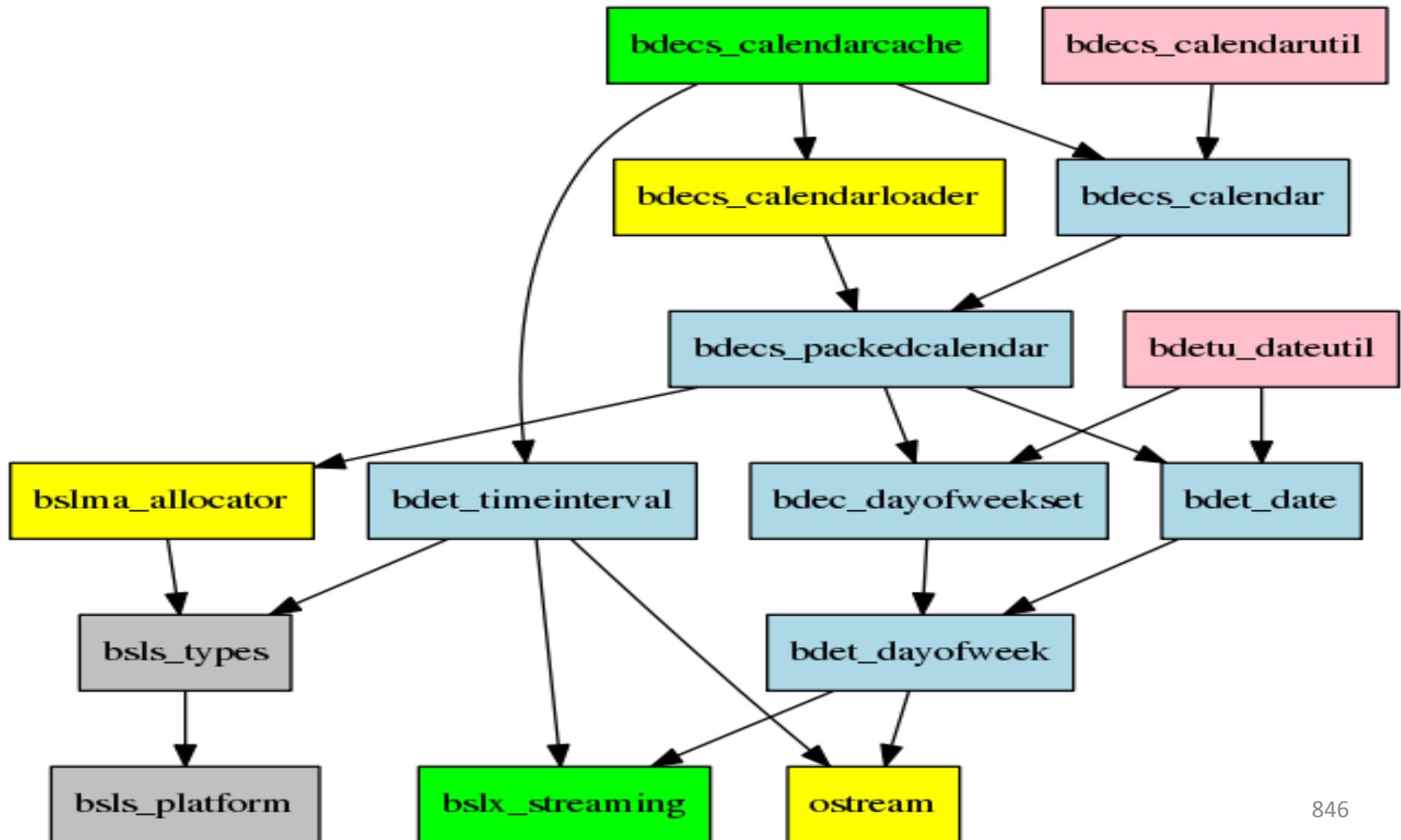
4. Bloomberg Development Environment

Package Group Dependencies



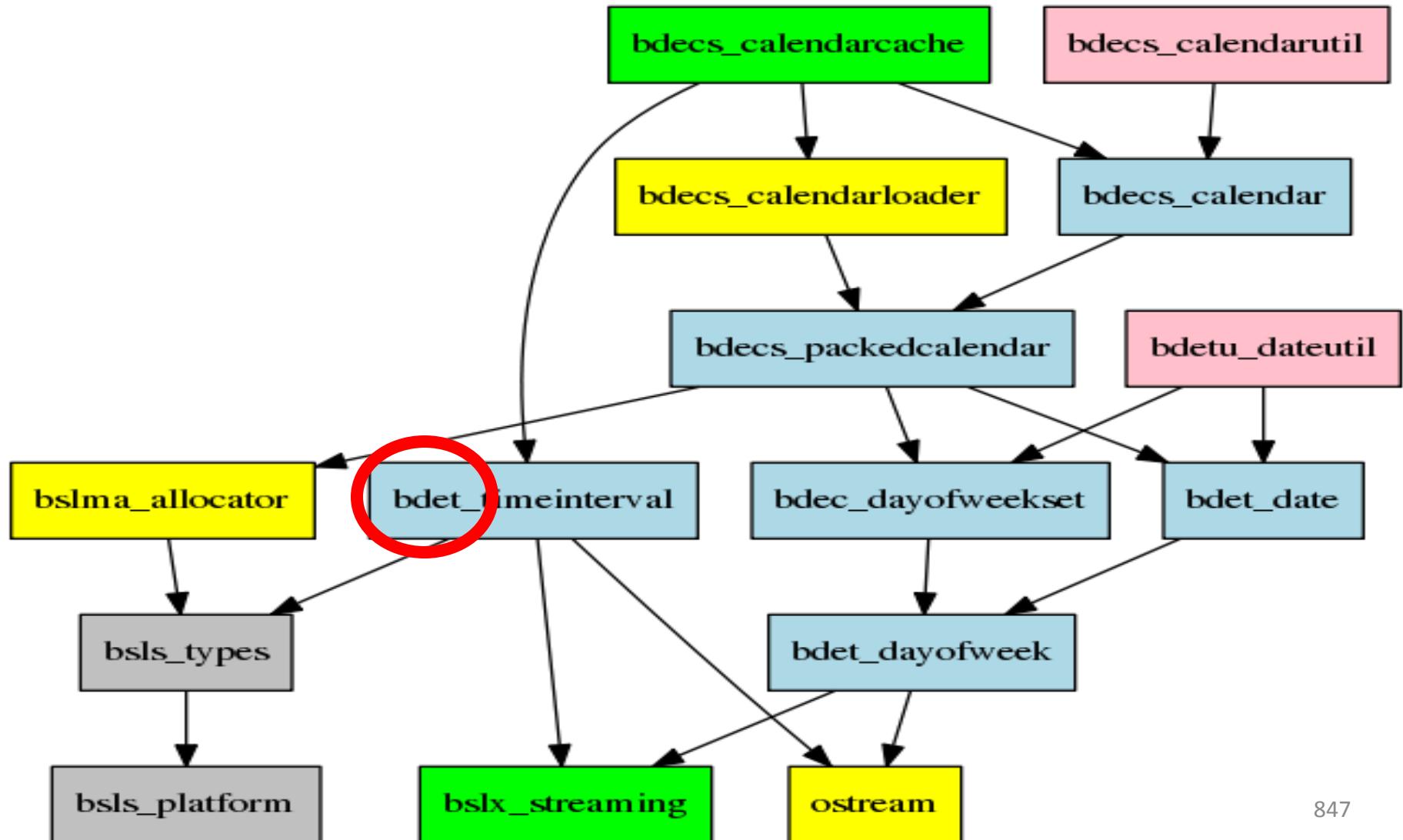
4. Bloomberg Development Environment

Client-Facing Component Diagram



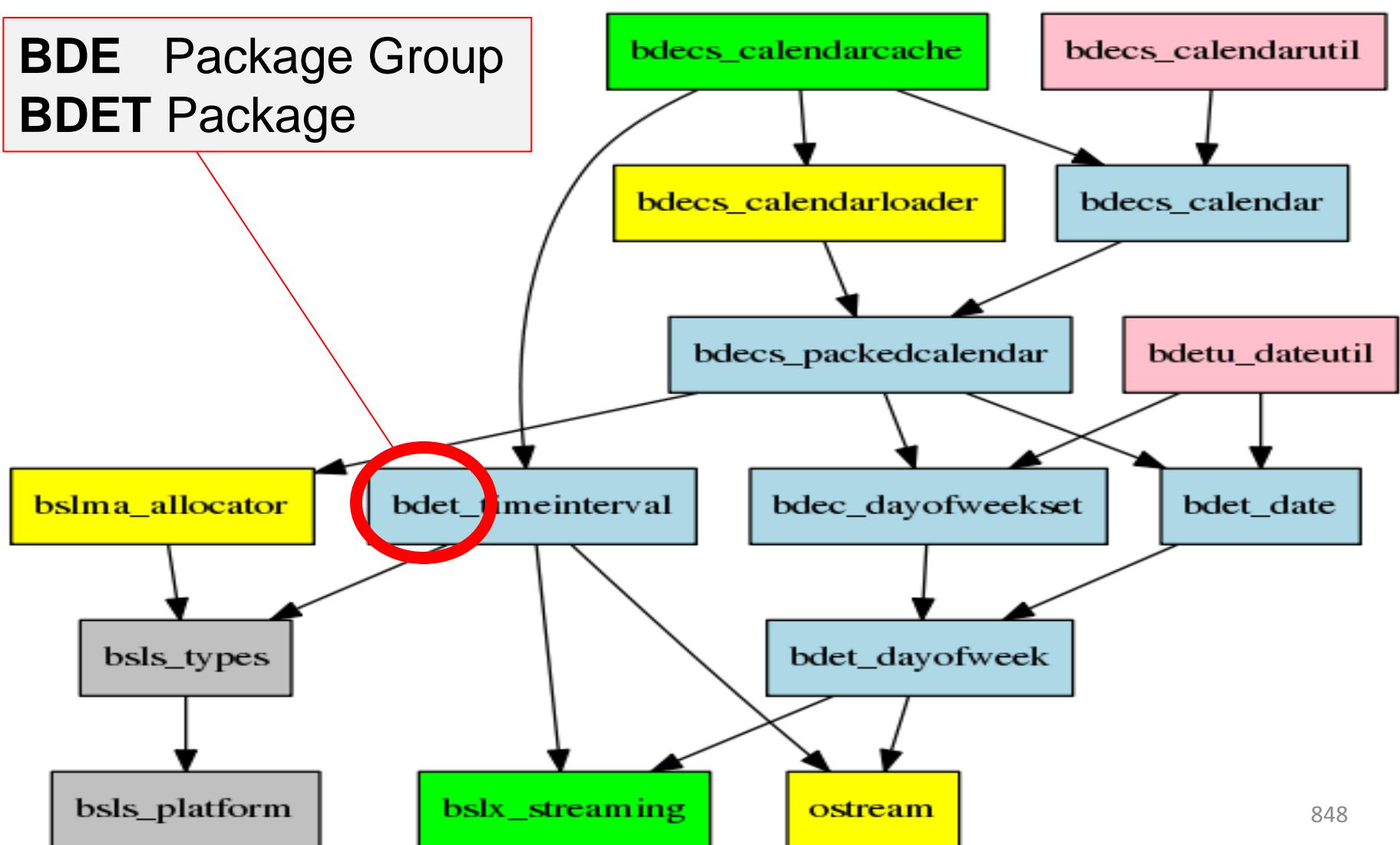
4. Bloomberg Development Environment

Client-Facing Component Diagram



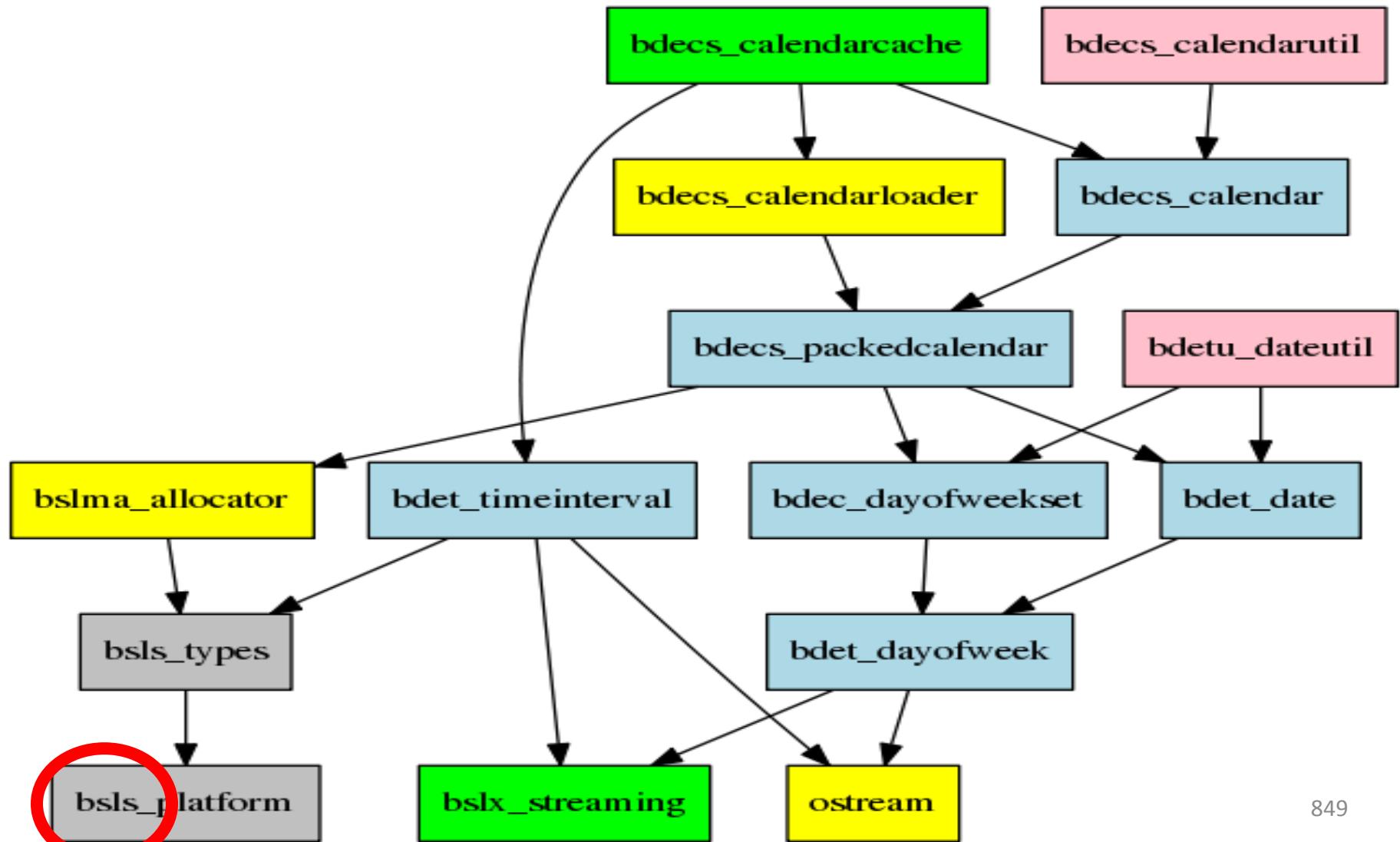
4. Bloomberg Development Environment

Client-Facing Component Diagram



4. Bloomberg Development Environment

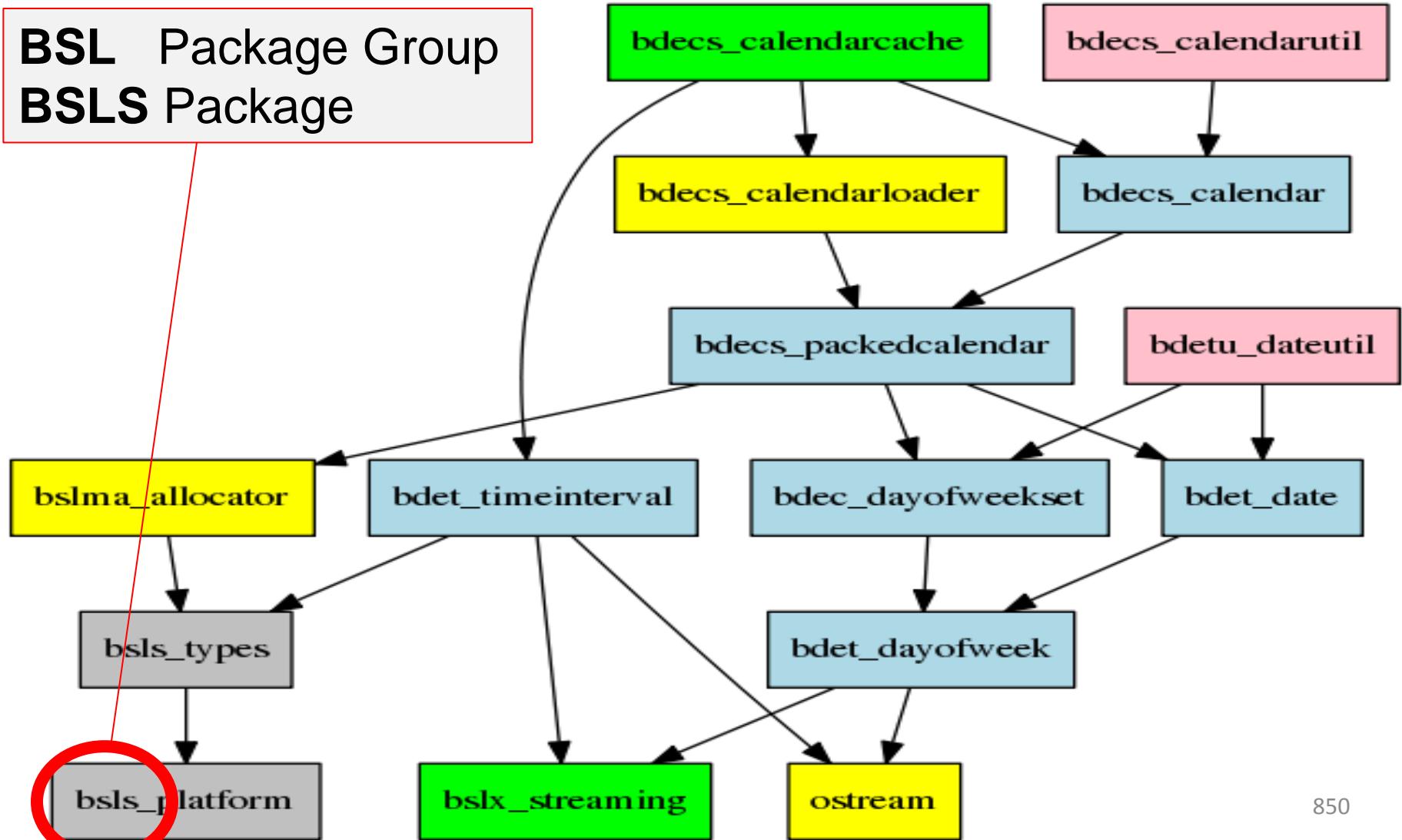
Client-Facing Component Diagram



4. Bloomberg Development Environment

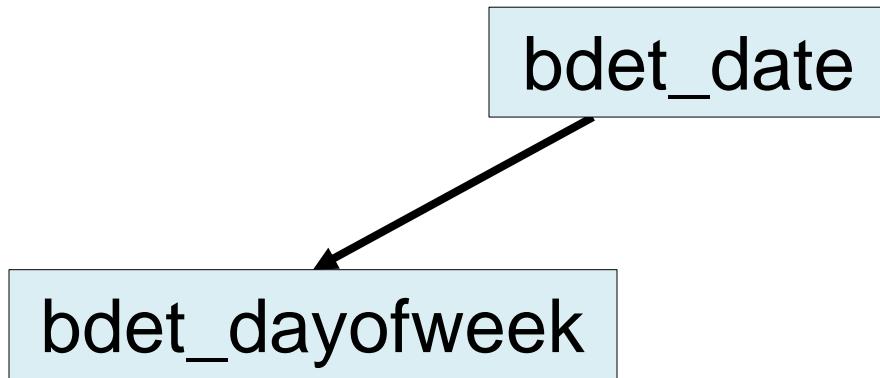
Client-Facing Component Diagram

BSL Package Group
BSLS Package



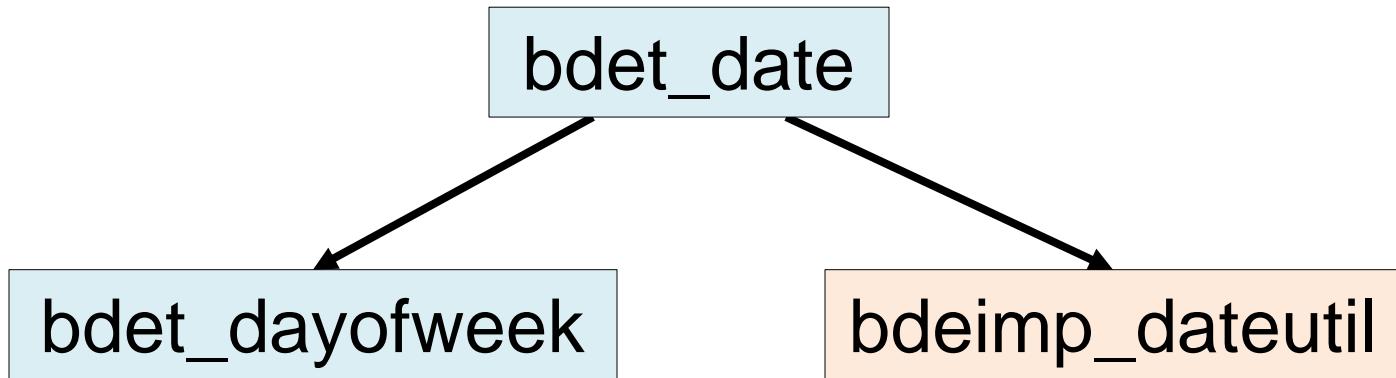
4. Bloomberg Development Environment

Implementing bdet_date



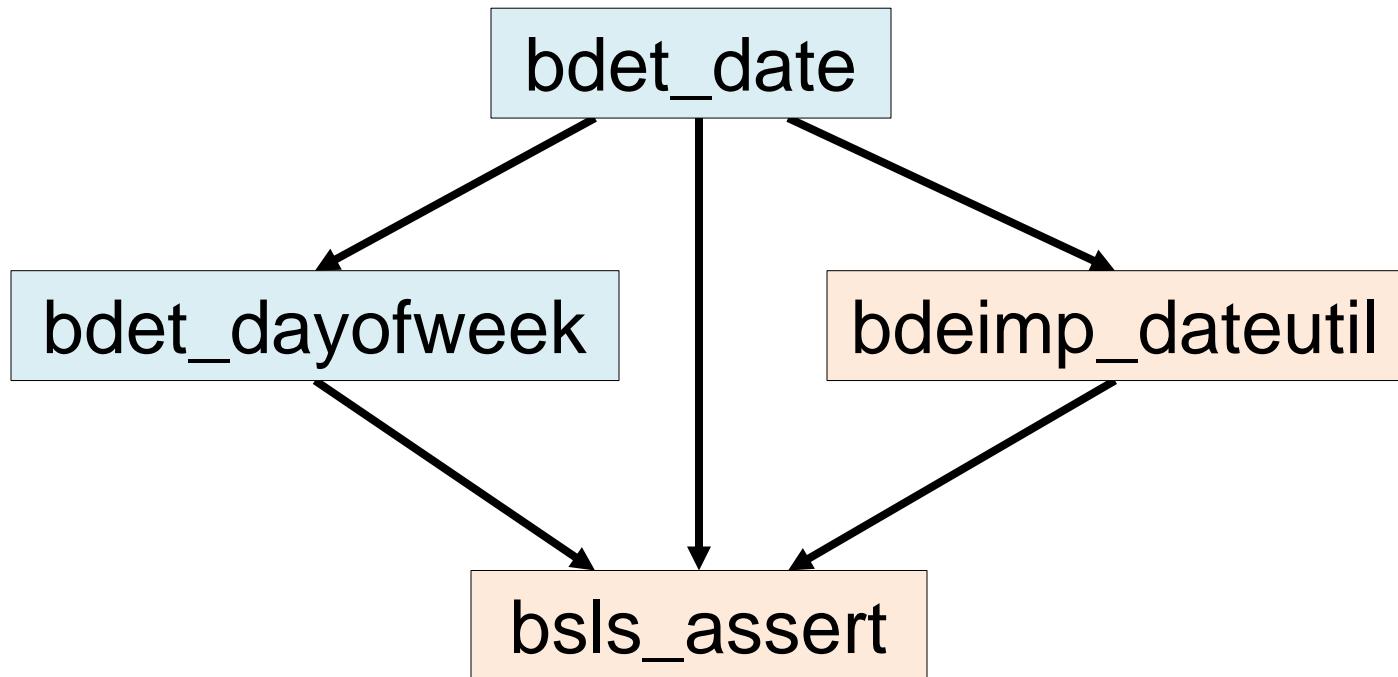
4. Bloomberg Development Environment

Implementing bdet_date



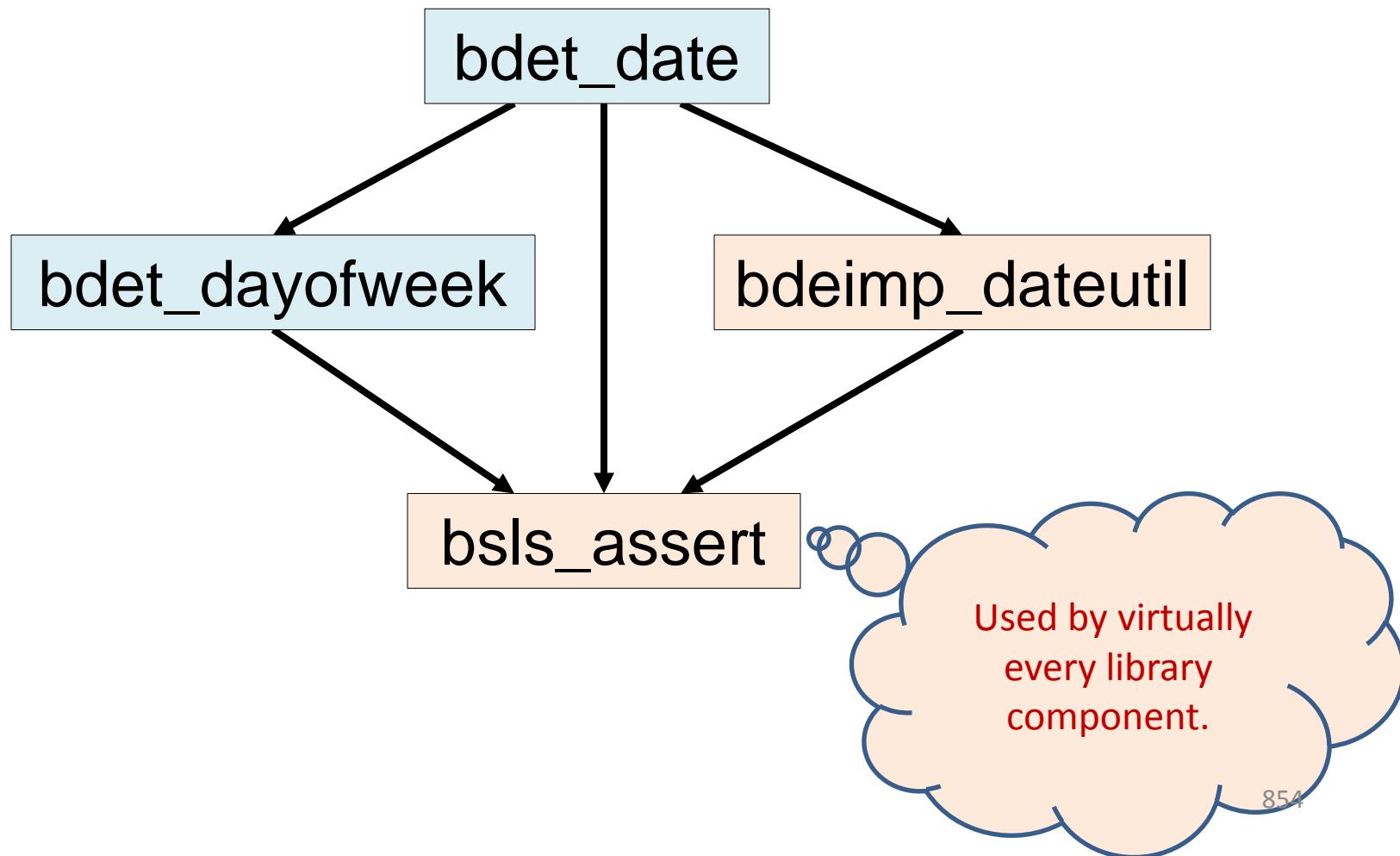
4. Bloomberg Development Environment

Implementing bdet_date



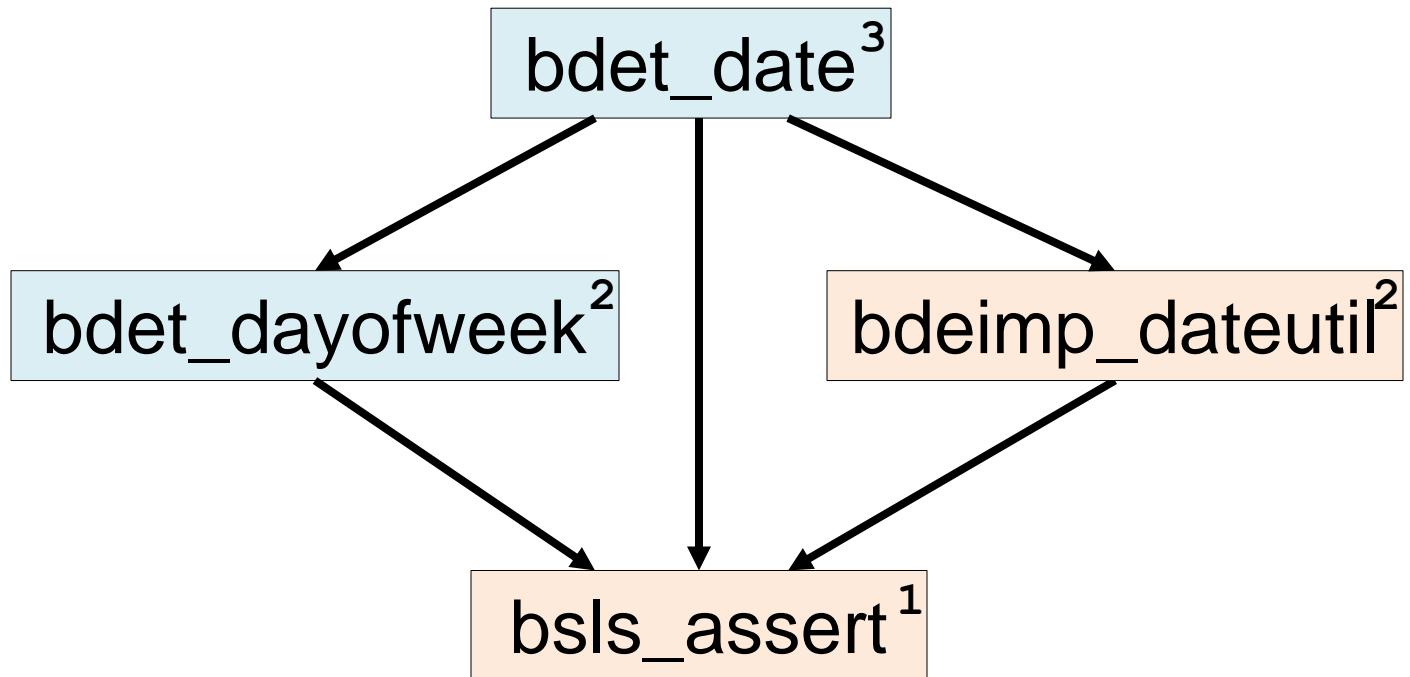
4. Bloomberg Development Environment

Implementing bdet_date



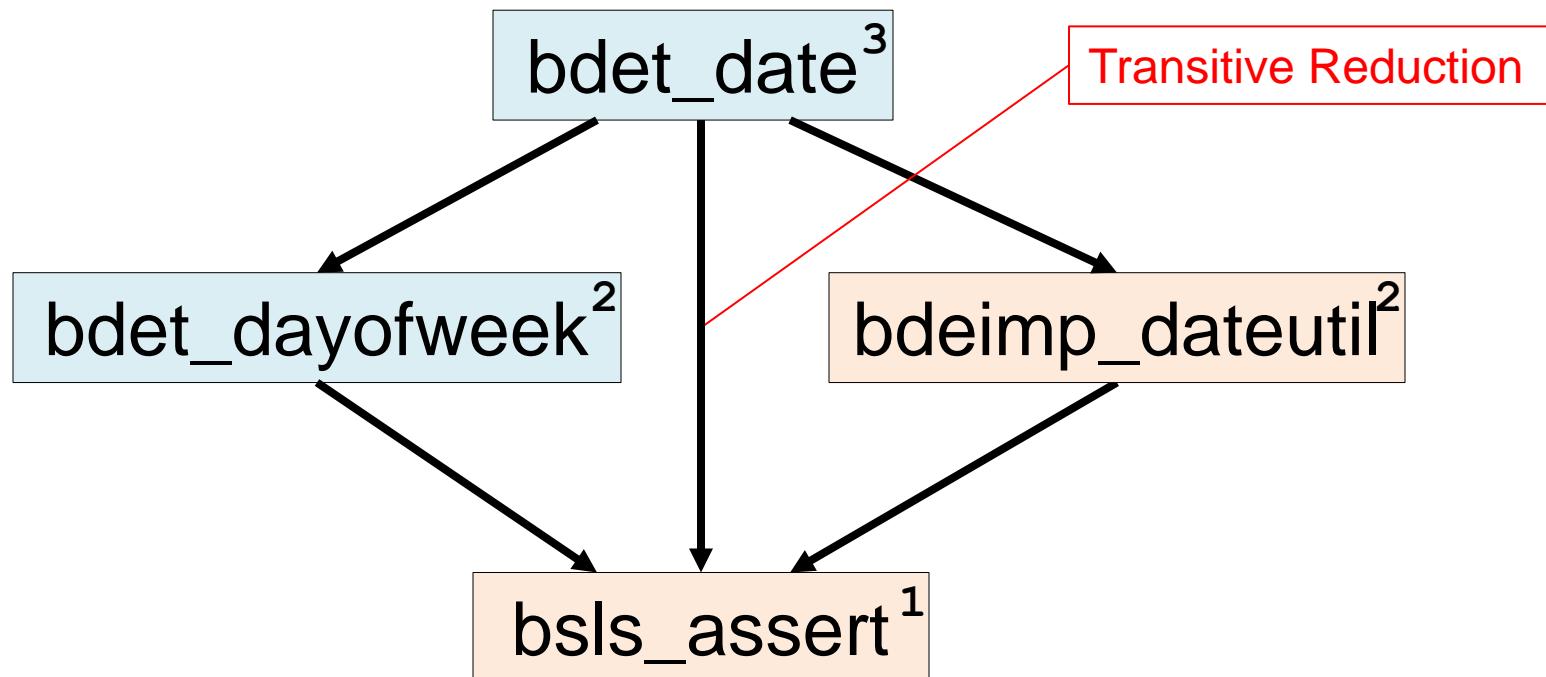
4. Bloomberg Development Environment

Implementing bdet_date



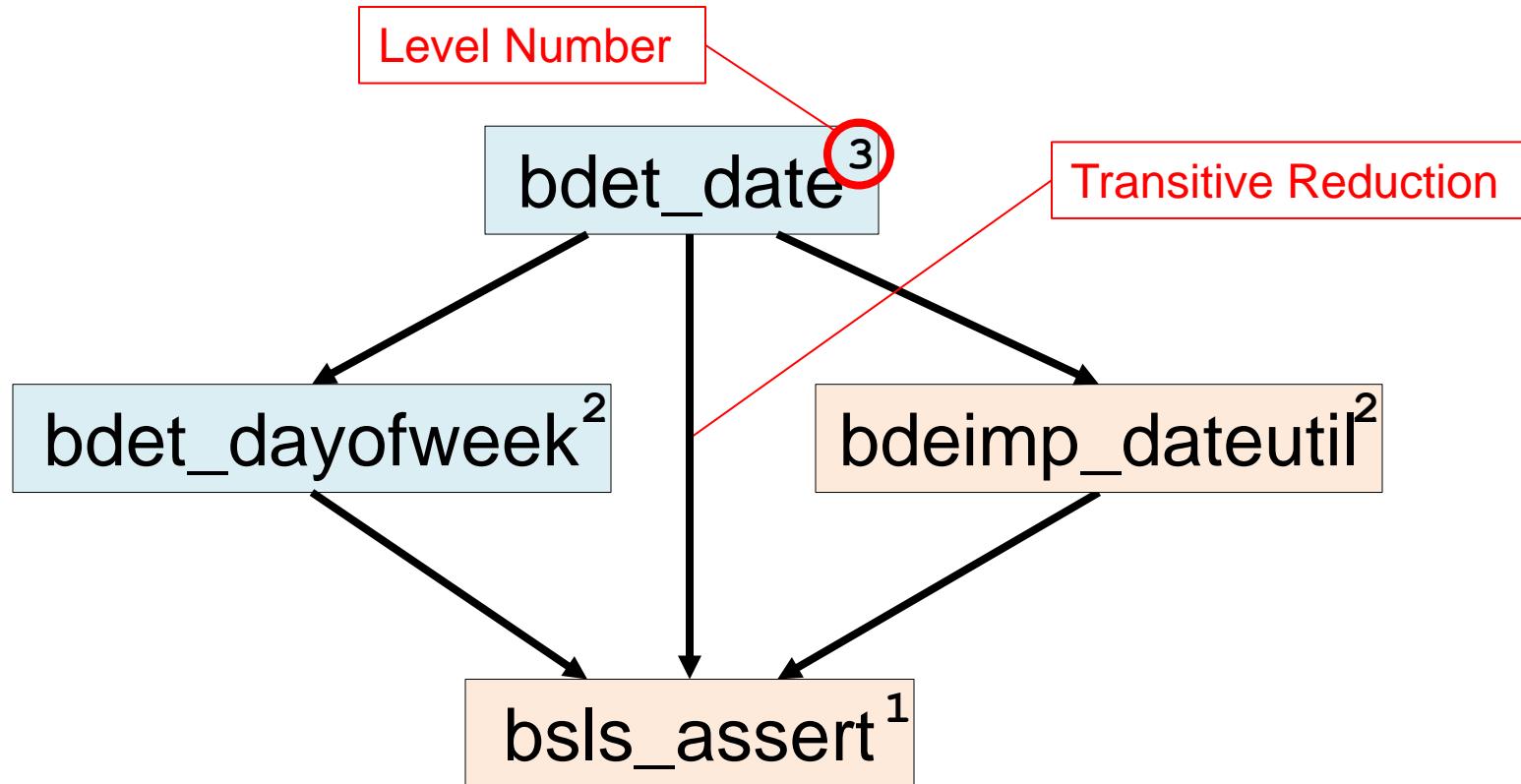
4. Bloomberg Development Environment

Implementing bdet_date



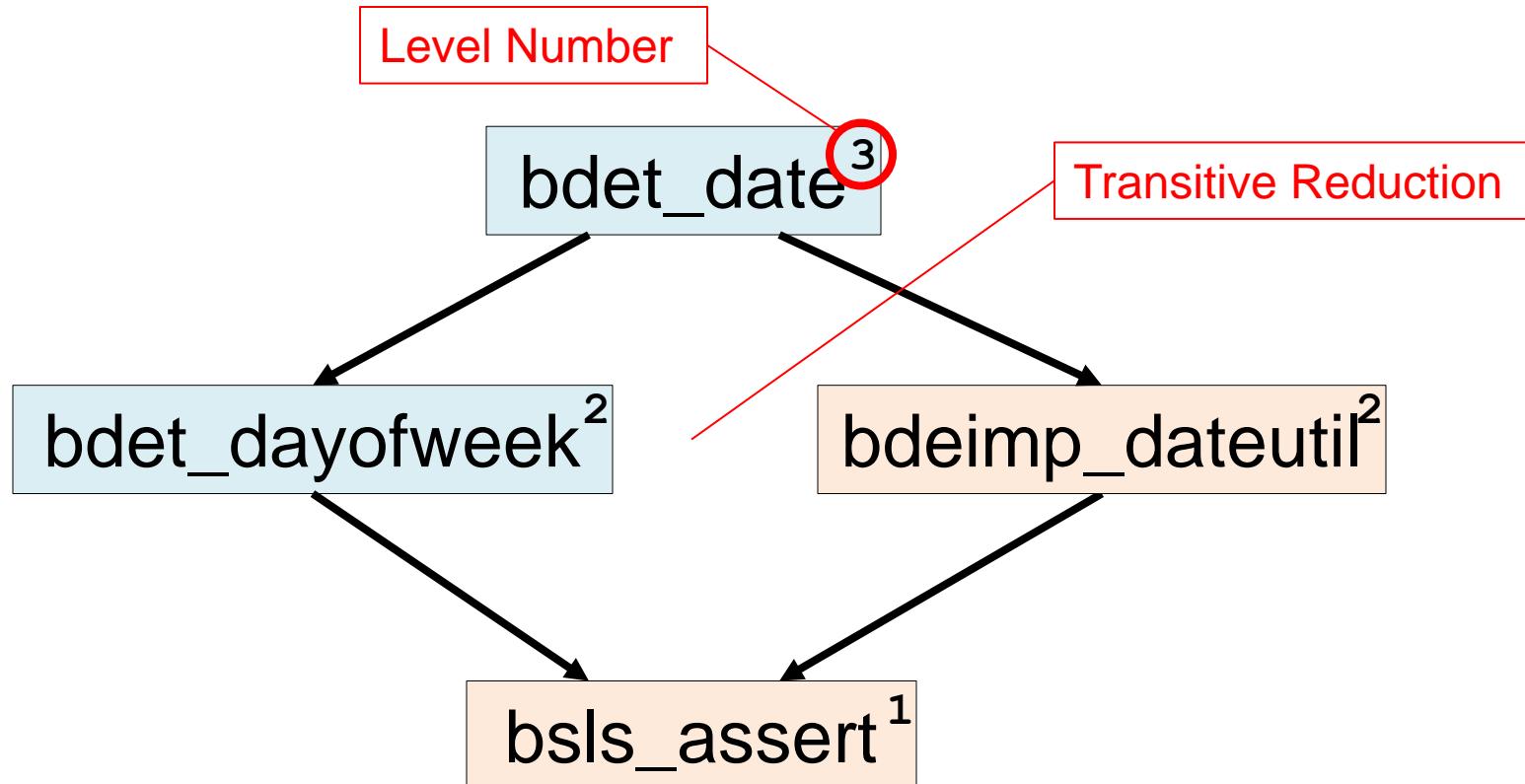
4. Bloomberg Development Environment

Implementing bdet_date



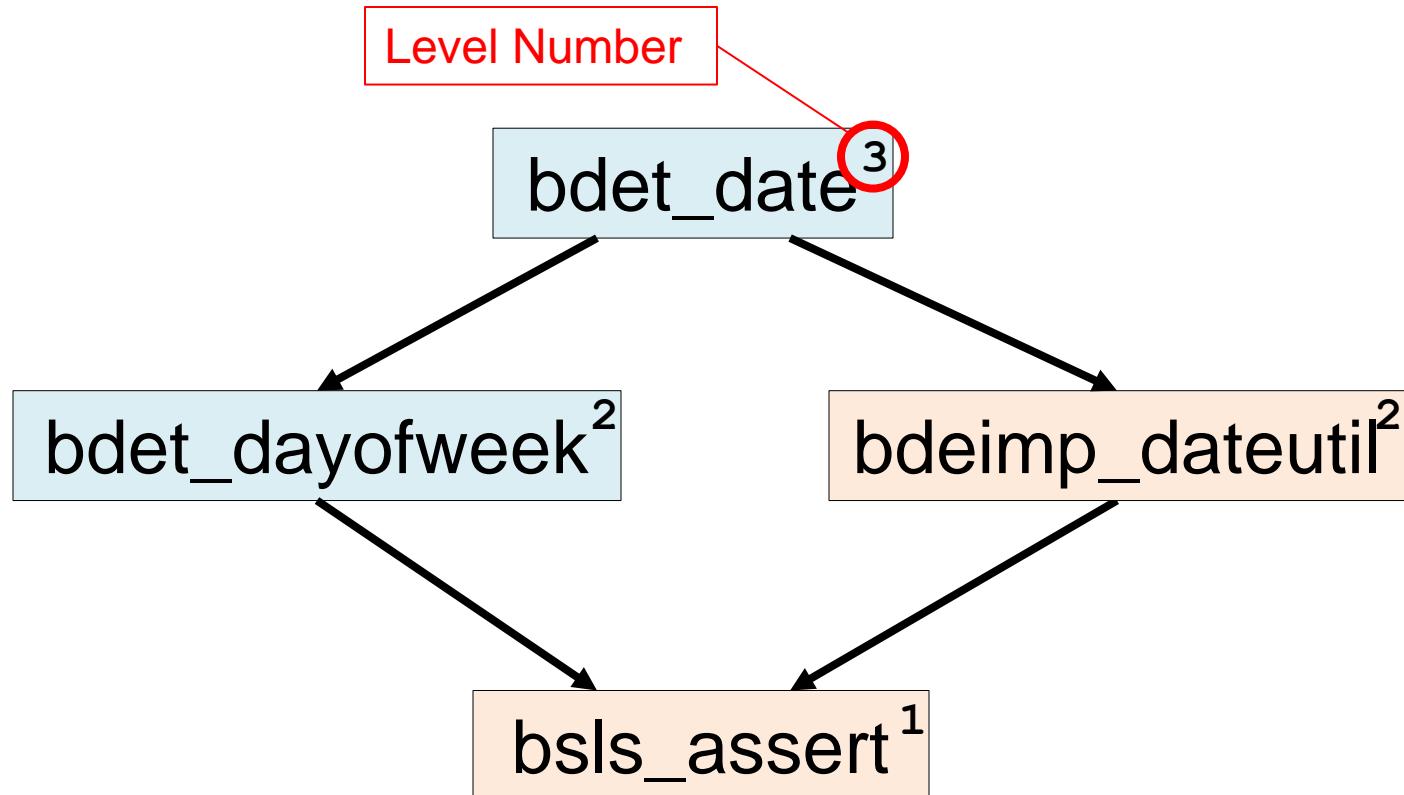
4. Bloomberg Development Environment

Implementing bdet_date



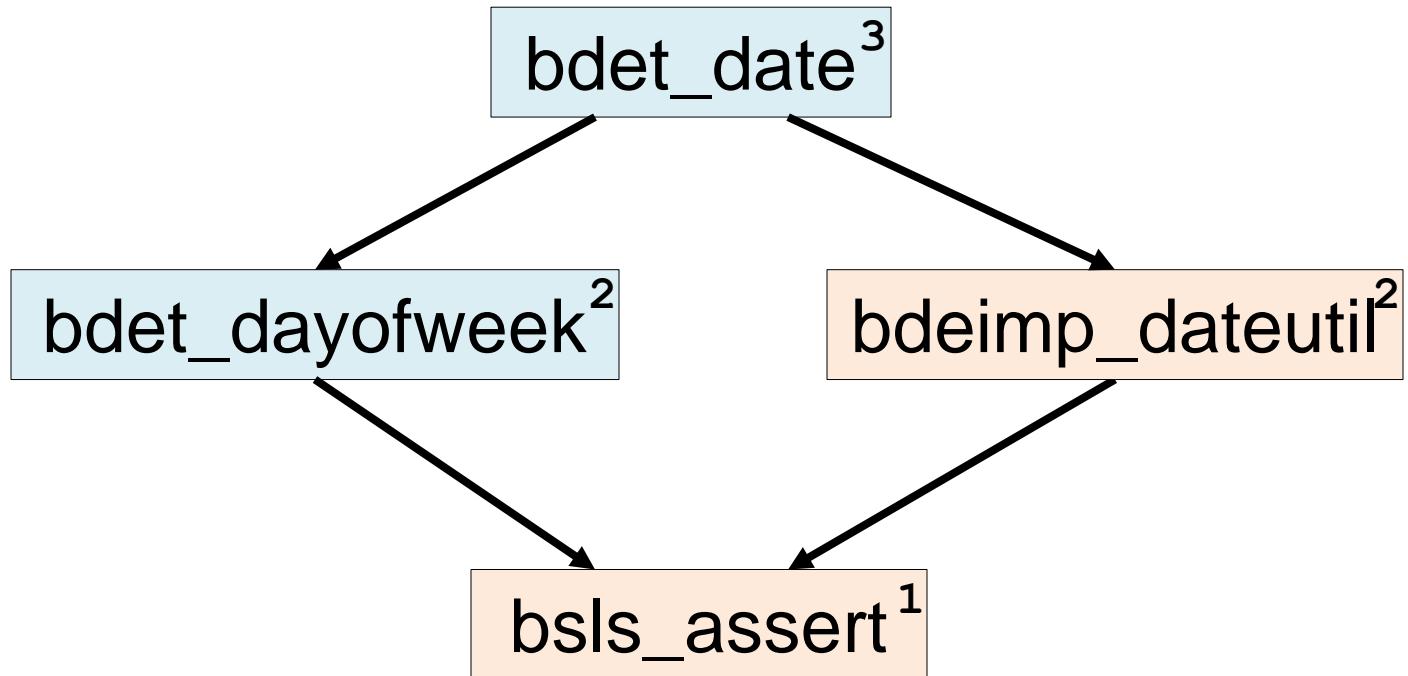
4. Bloomberg Development Environment

Implementing bdet_date



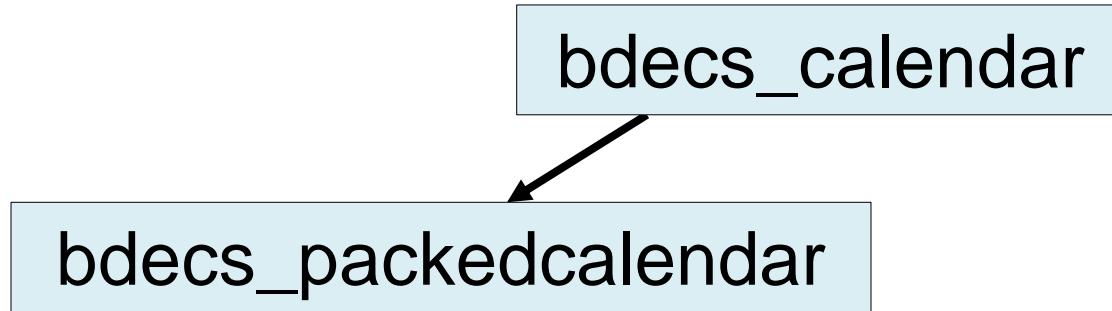
4. Bloomberg Development Environment

Implementing bdet_date



4. Bloomberg Development Environment

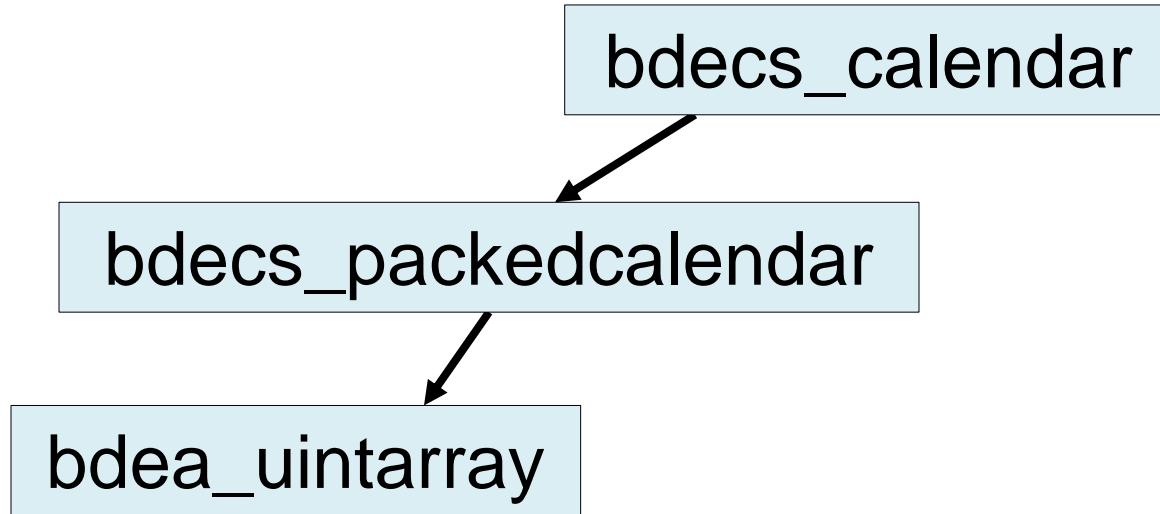
Implementing bdecs_calendar



bslma_allocator

4. Bloomberg Development Environment

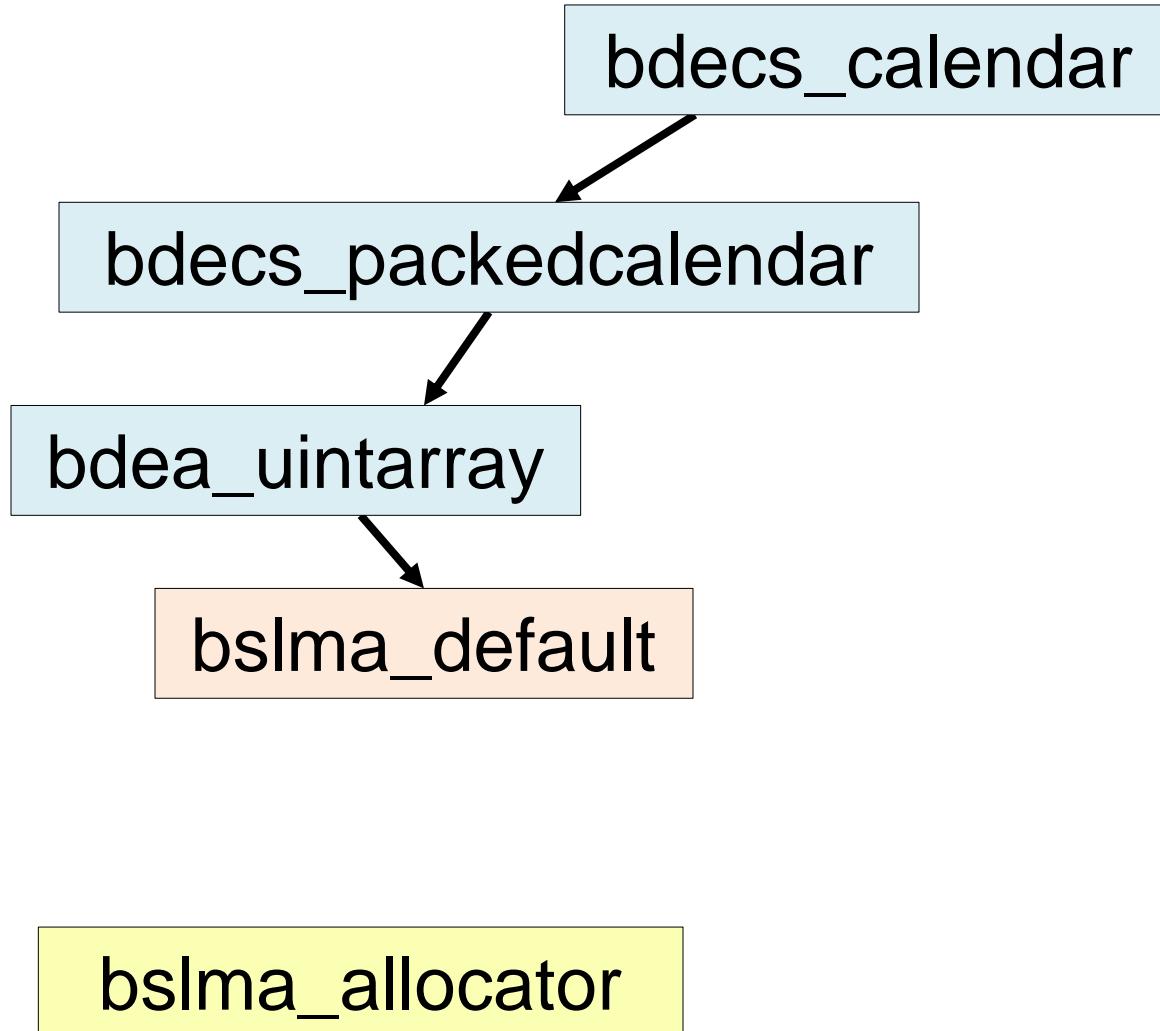
Implementing bdecs_calendar



bslma_allocator

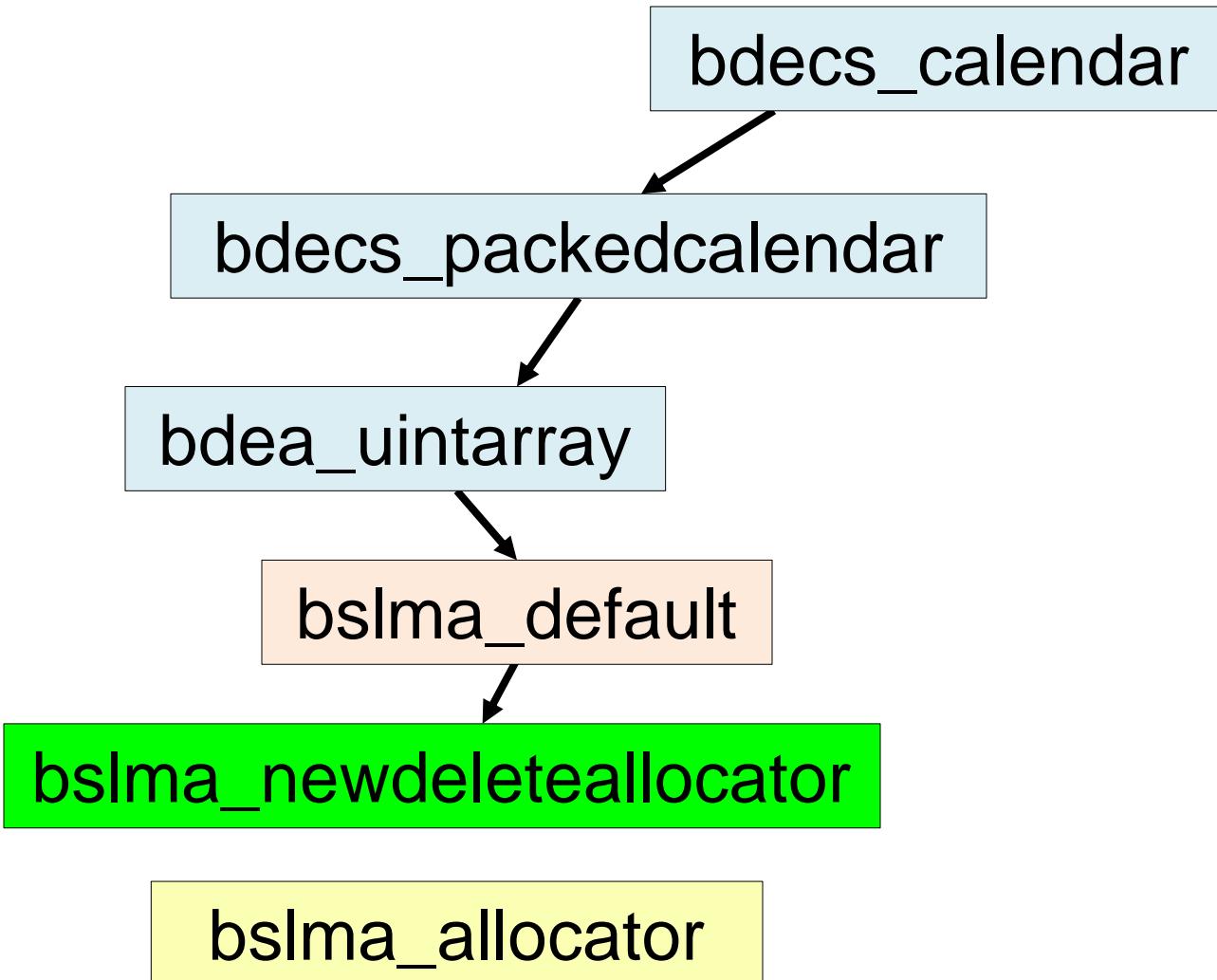
4. Bloomberg Development Environment

Implementing bdecs_calendar



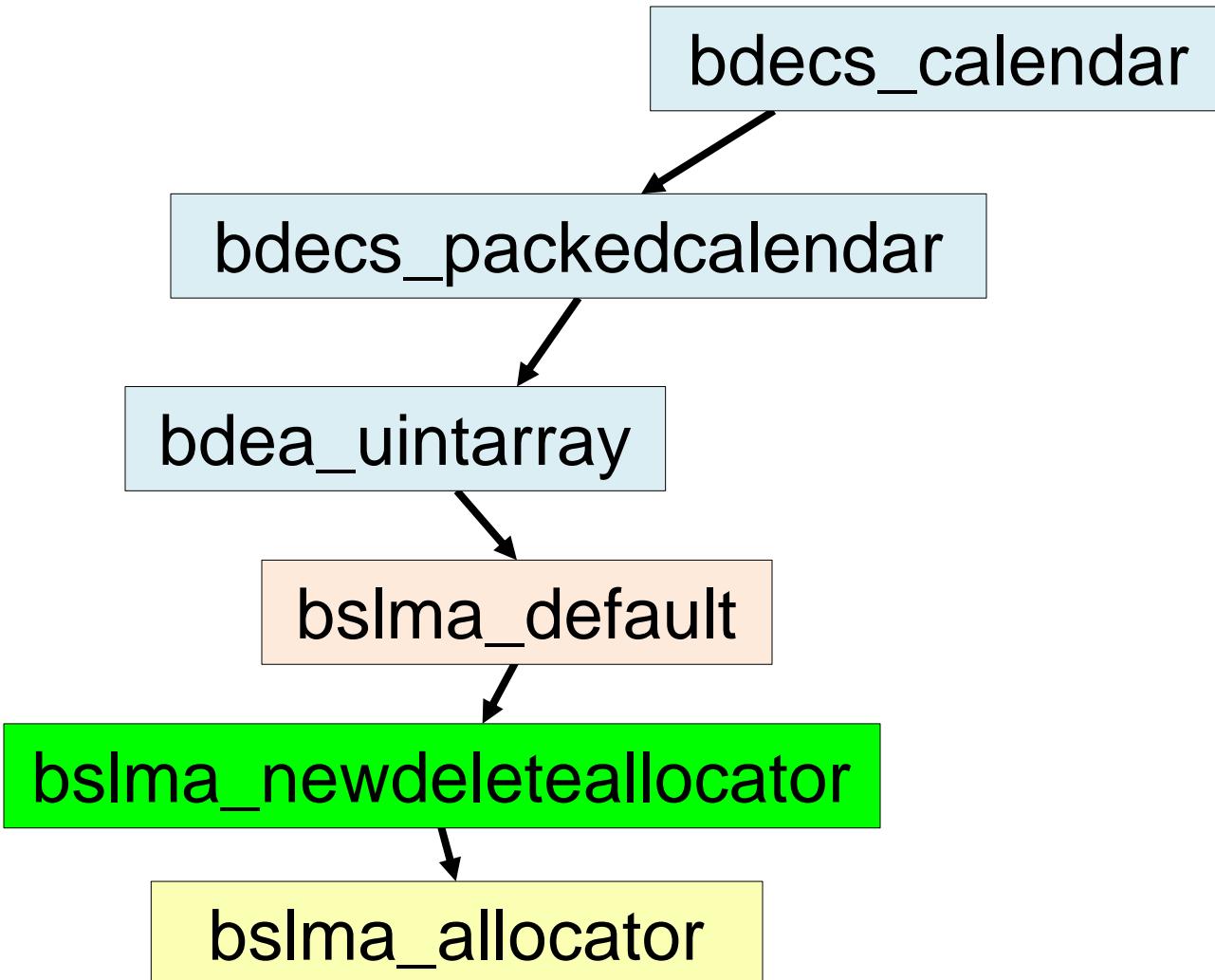
4. Bloomberg Development Environment

Implementing bdecs_calendar



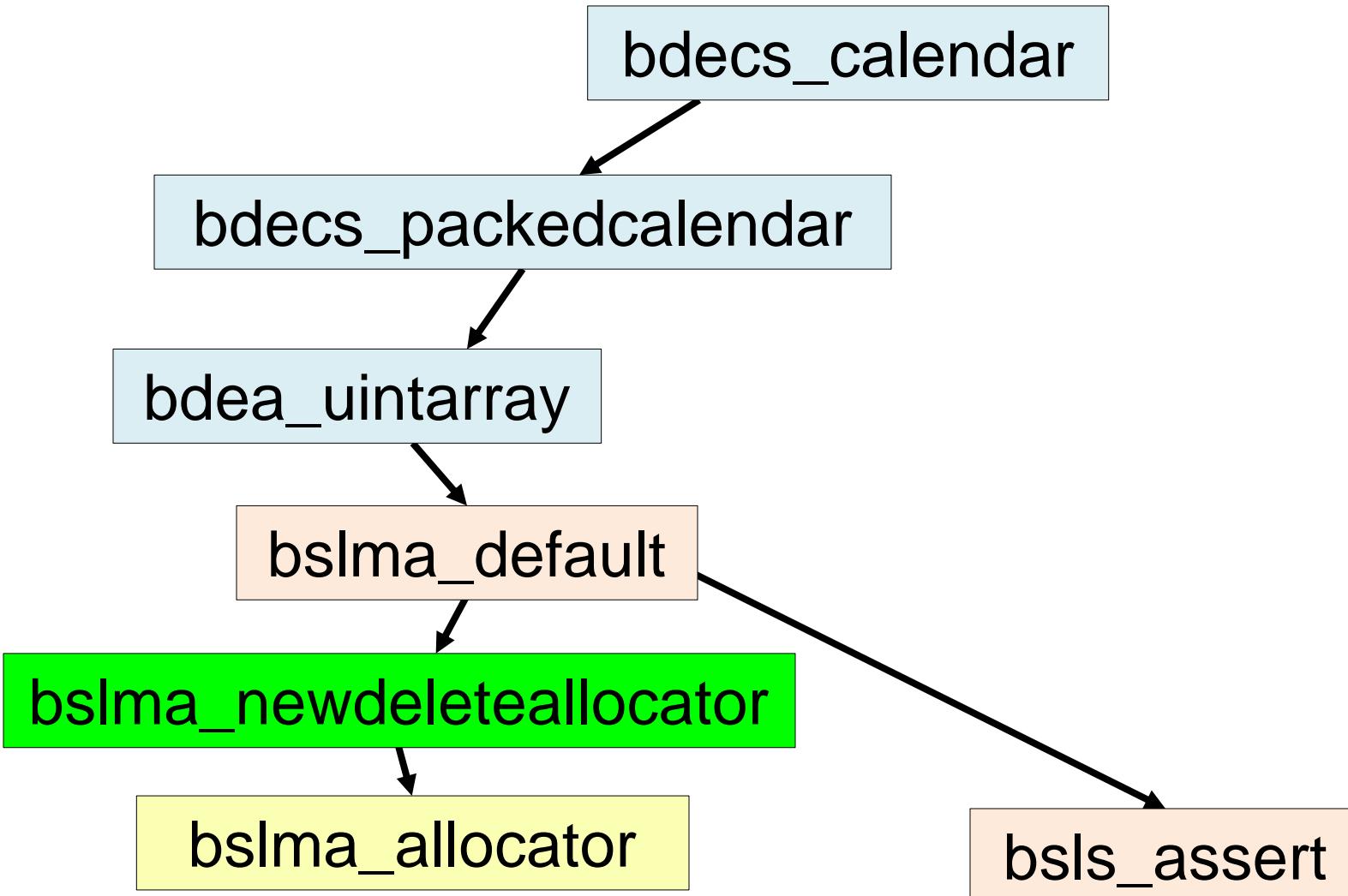
4. Bloomberg Development Environment

Implementing bdecs_calendar



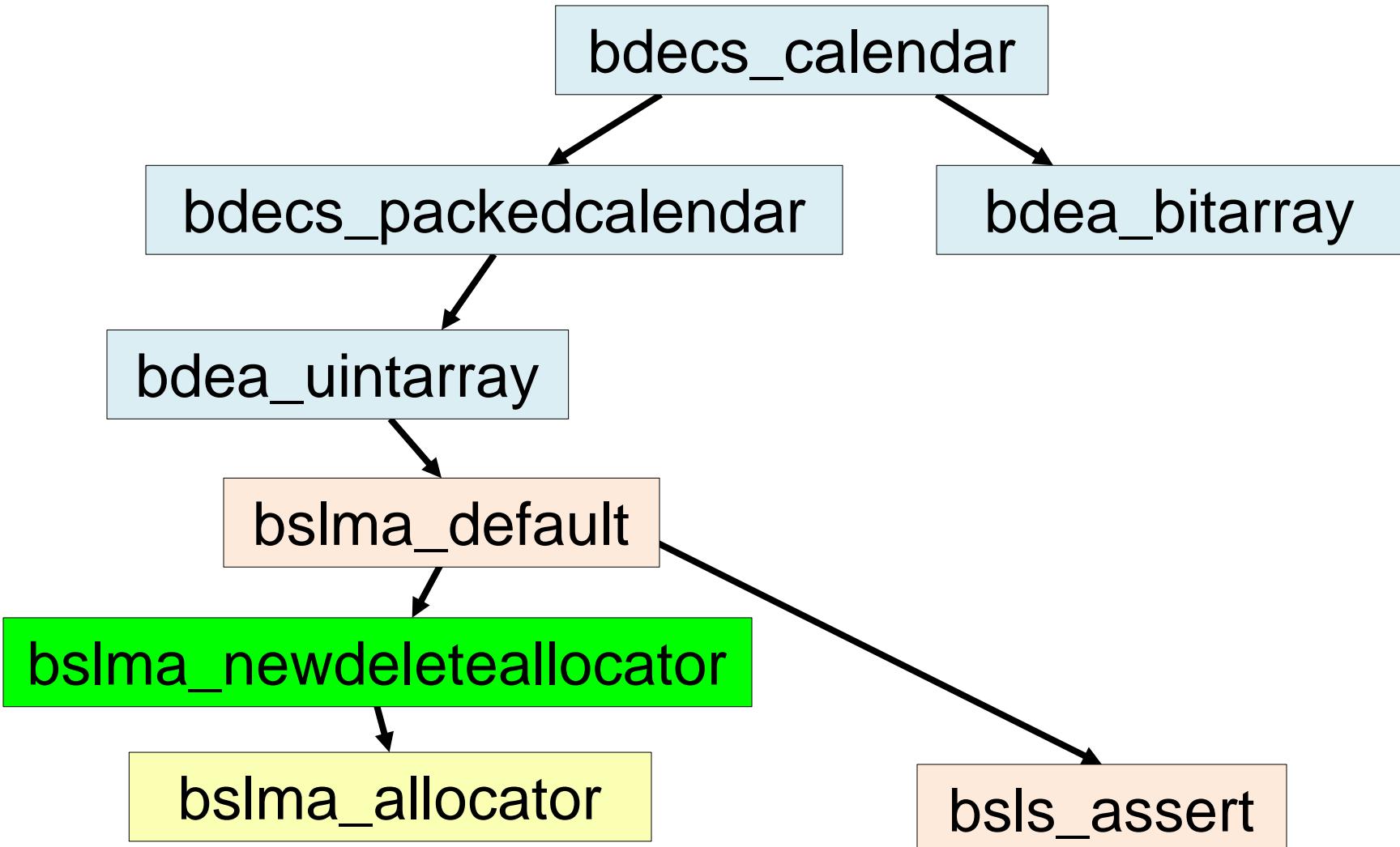
4. Bloomberg Development Environment

Implementing bdecs_calendar



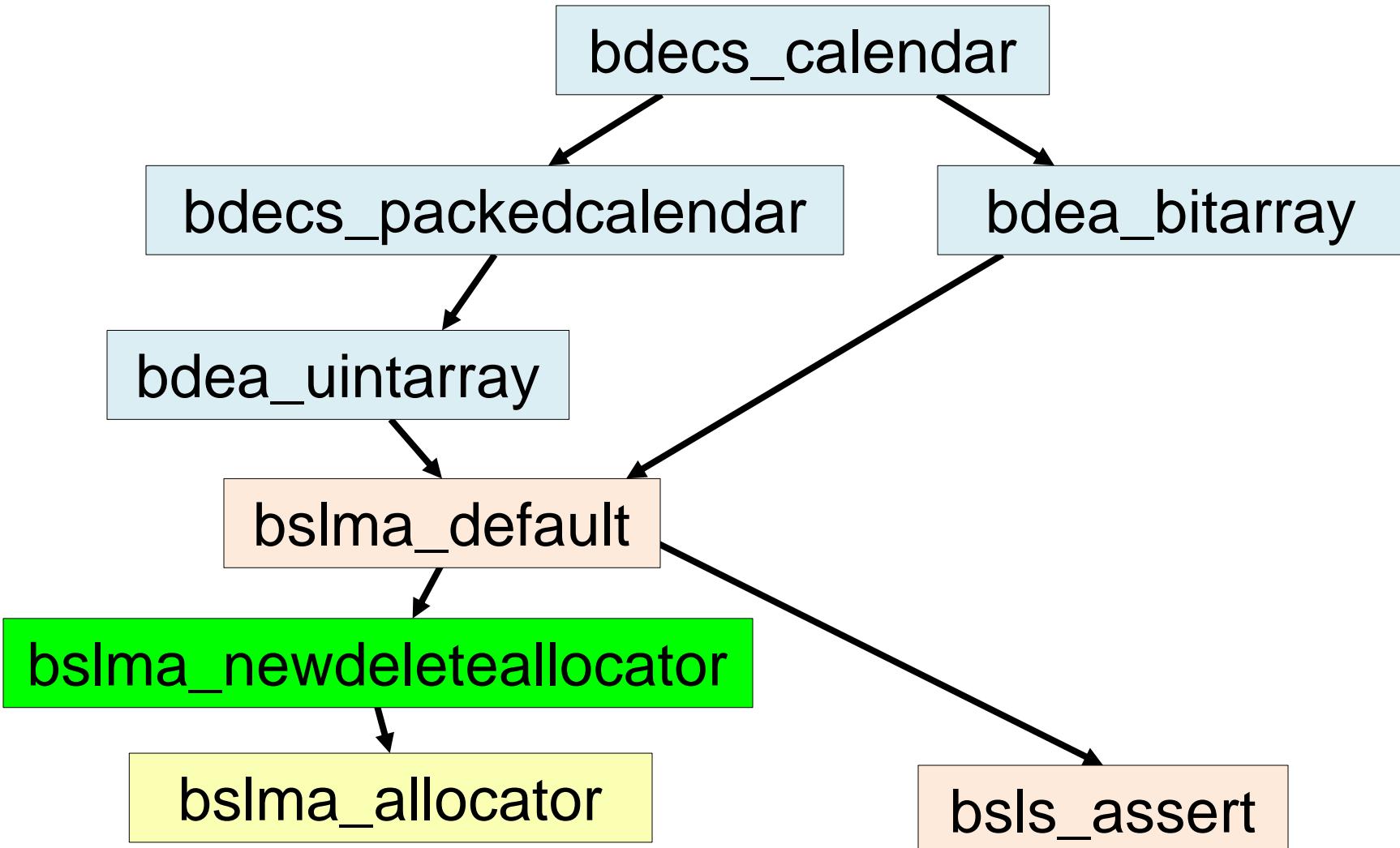
4. Bloomberg Development Environment

Implementing bdecs_calendar



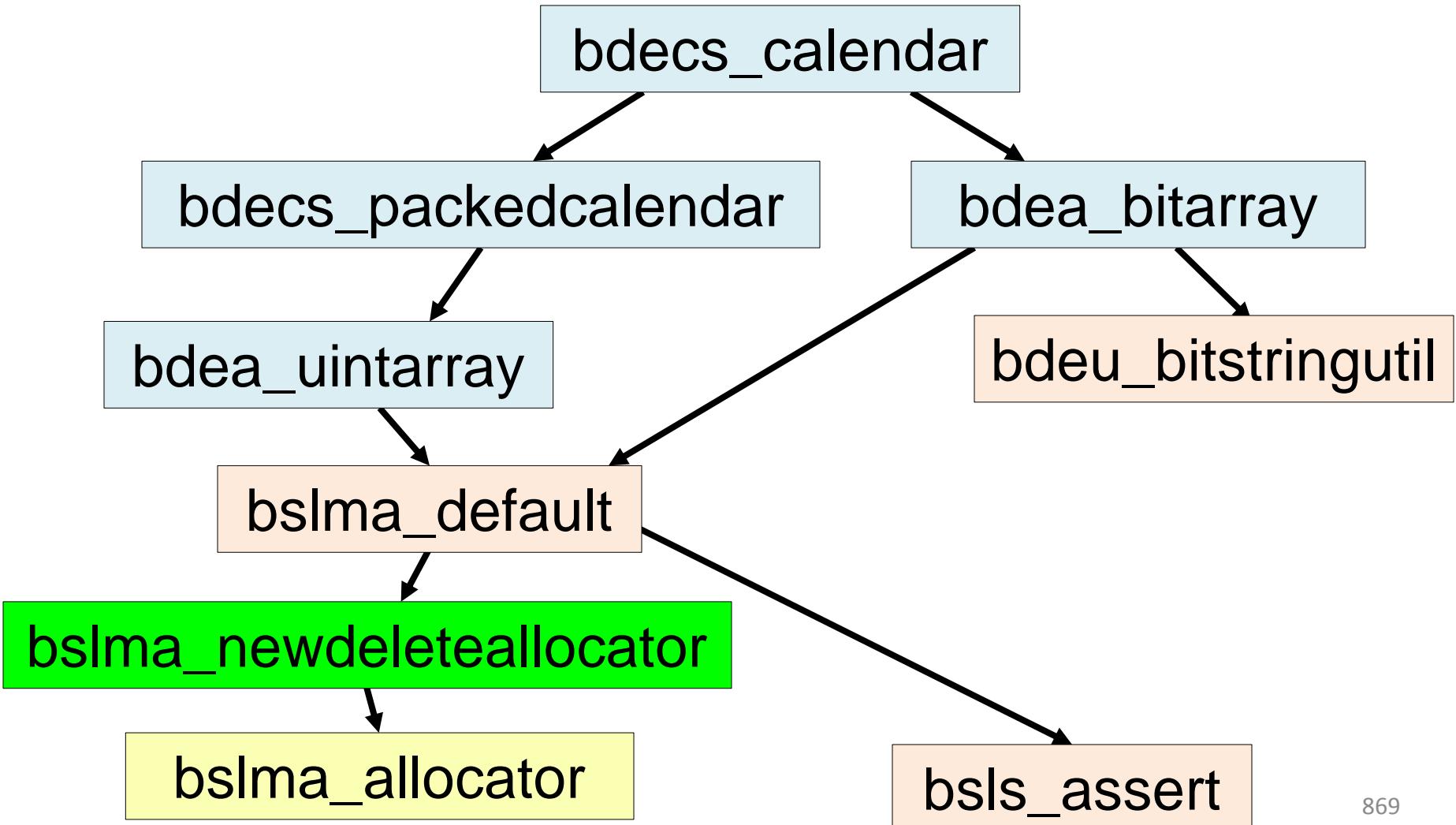
4. Bloomberg Development Environment

Implementing bdecs_calendar



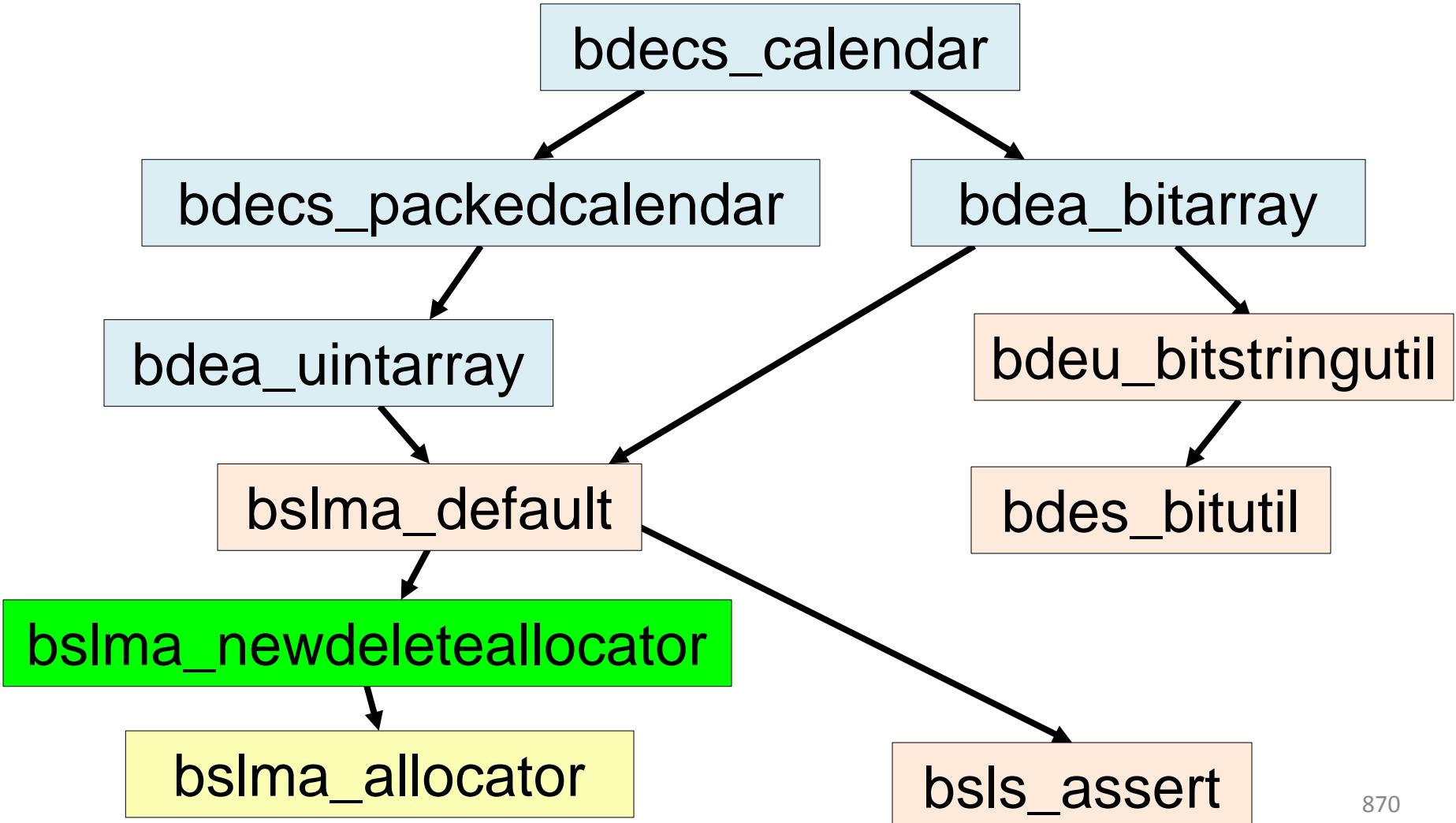
4. Bloomberg Development Environment

Implementing bdecs_calendar



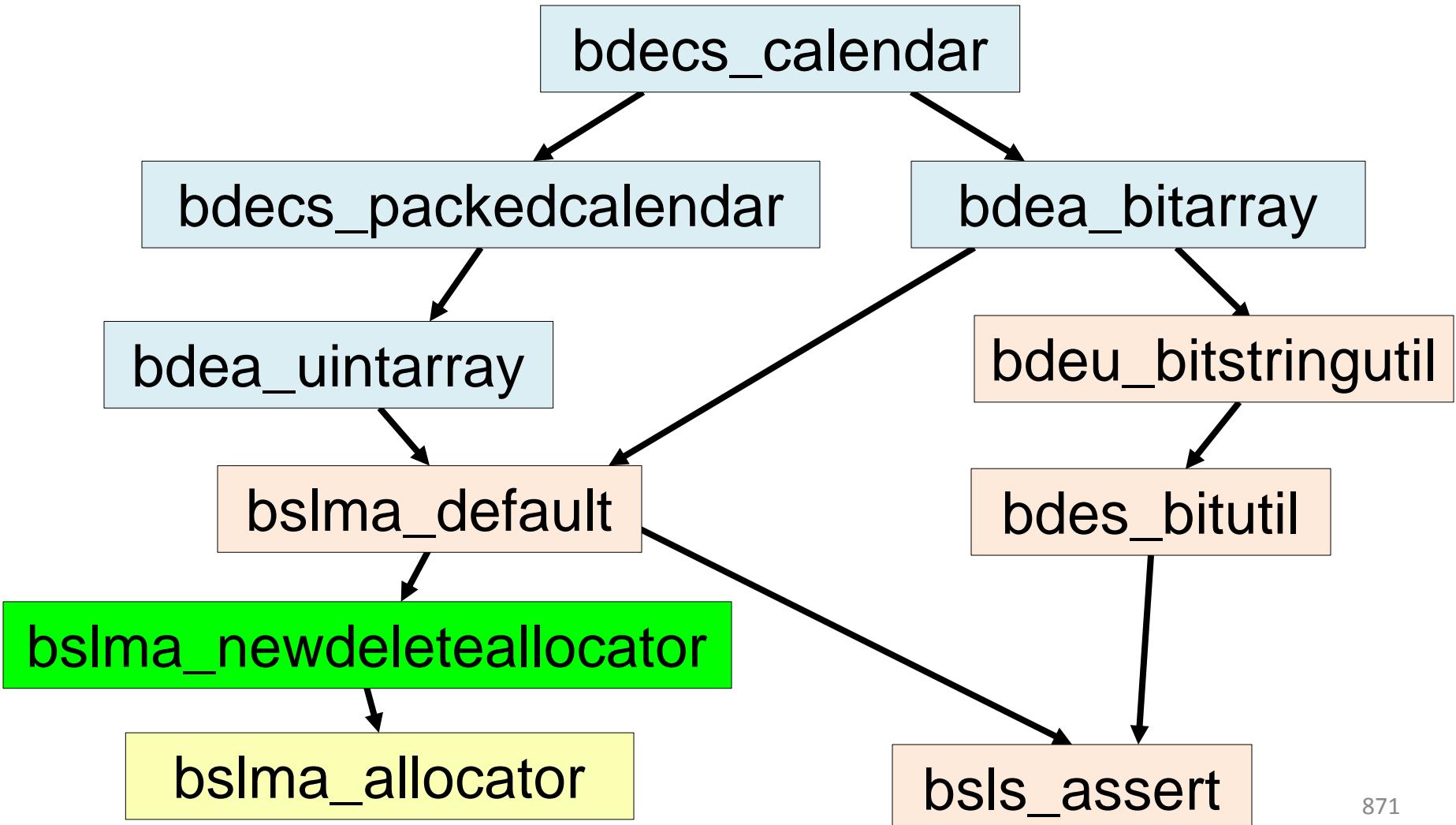
4. Bloomberg Development Environment

Implementing bdecs_calendar



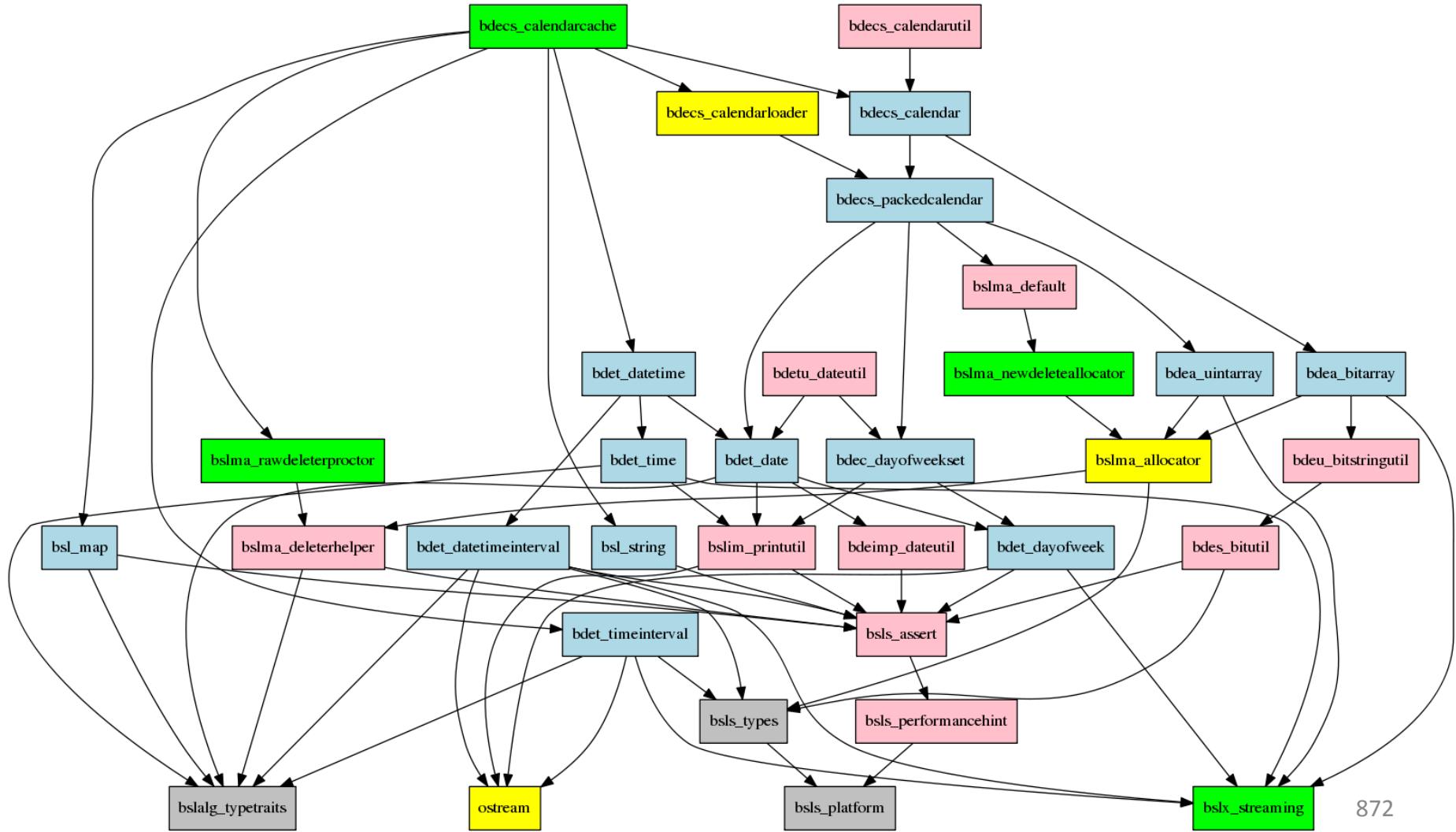
4. Bloomberg Development Environment

Implementing bdecs_calendar



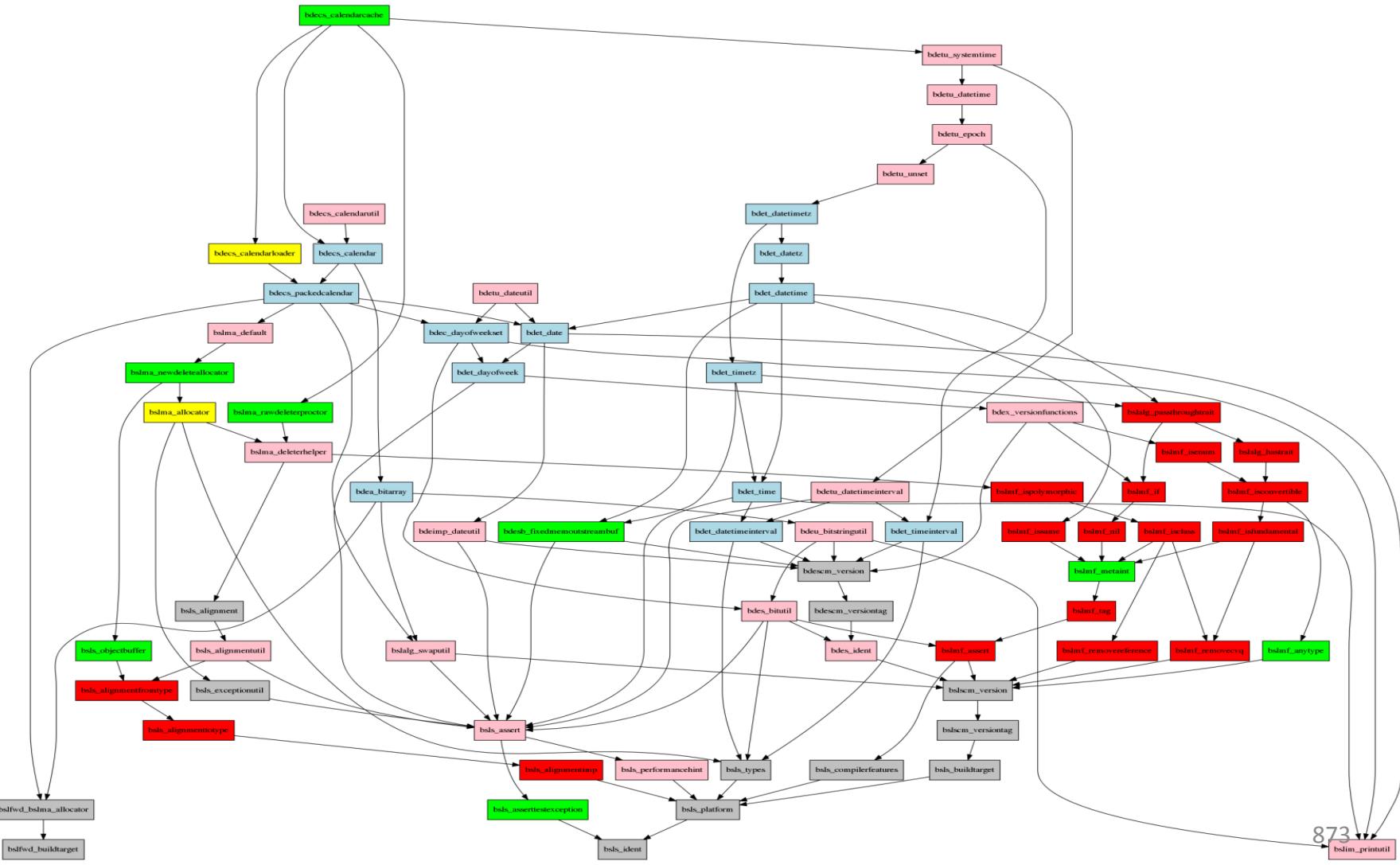
4. Bloomberg Development Environment

Hierarchically Reusable Implementation



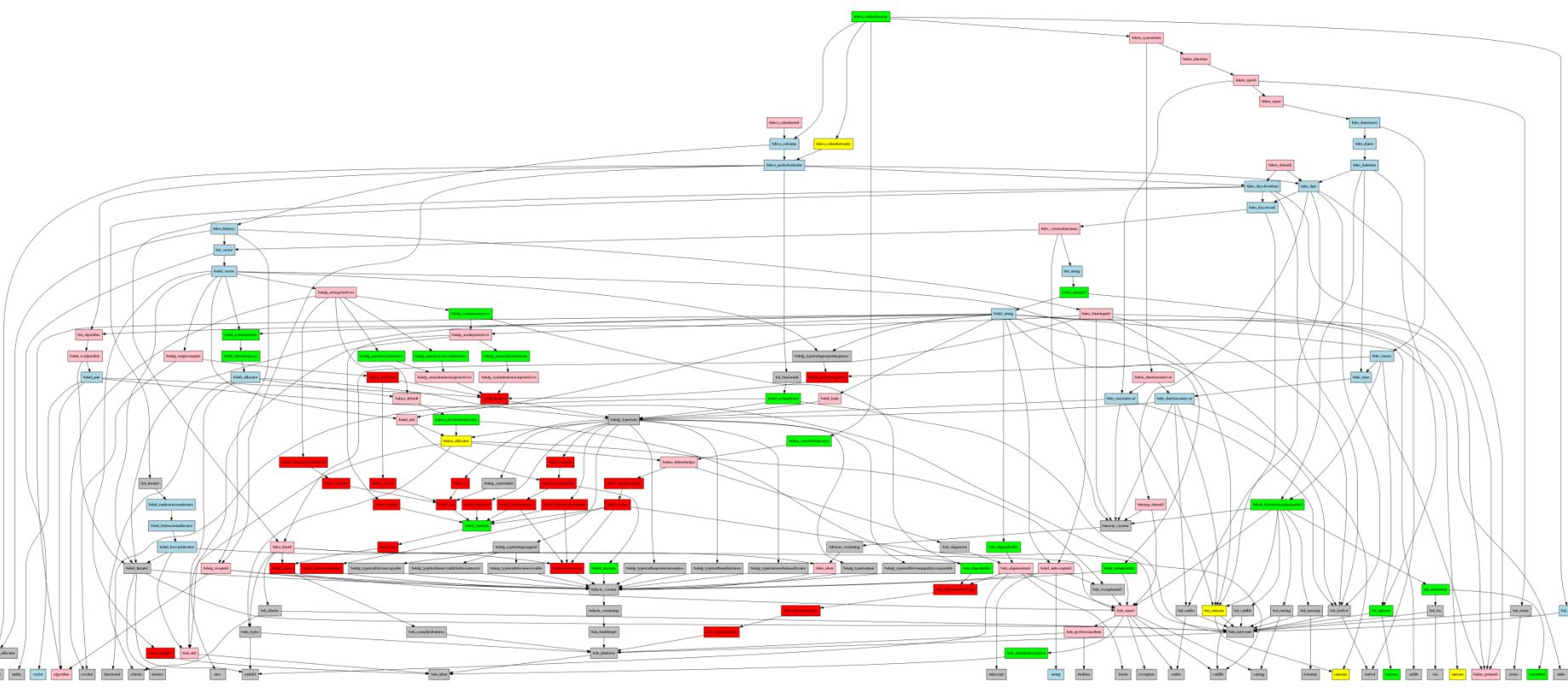
4. Bloomberg Development Environment

Hierarchically Reusable Implementation



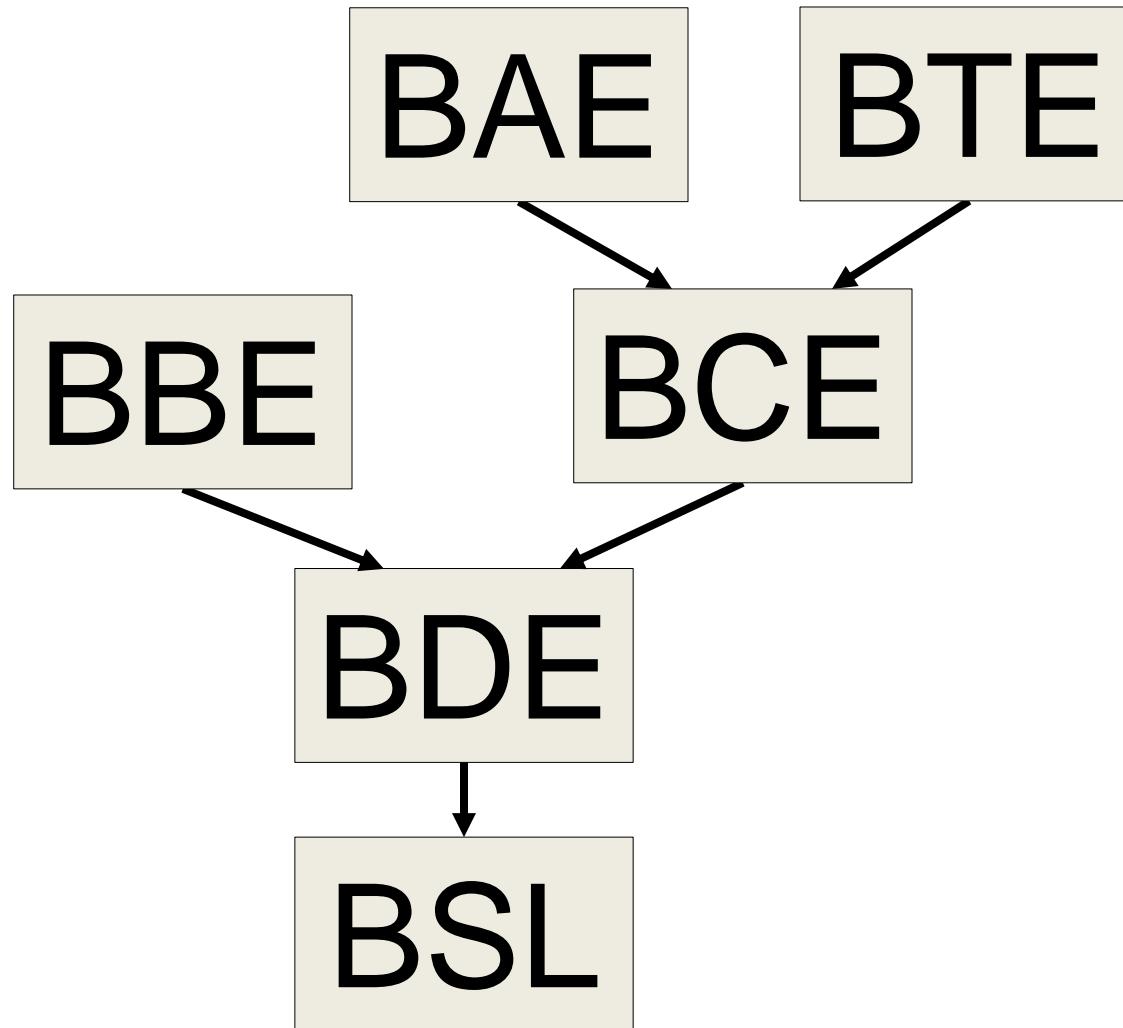
4. Bloomberg Development Environment

Hierarchically Reusable Implementation



4. Bloomberg Development Environment

Foundation “Package-Group” Libraries



Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

4. Bloomberg Development Environment (BDE)

Rendered as Fine-Grained Hierarchically Reusable Components.

Outline

0. Goals

What we are trying to do, for whom, and how.

1. Process & Architecture

Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

Component-Level Test Drivers, Peer Review, & Defensive Checks.

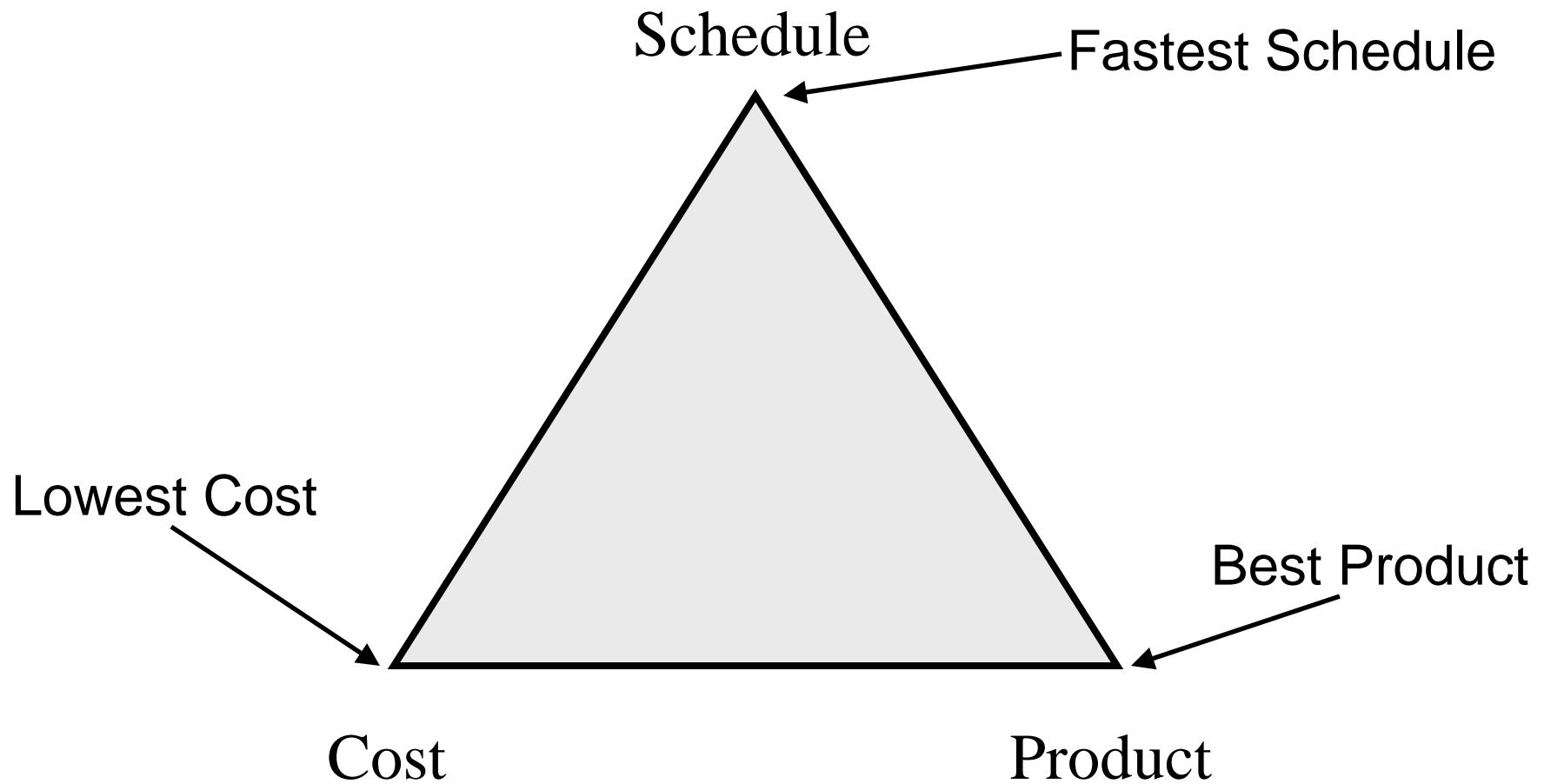
4. Bloomberg Development Environment (BDE)

Rendered as Fine-Grained Hierarchically Reusable Components.

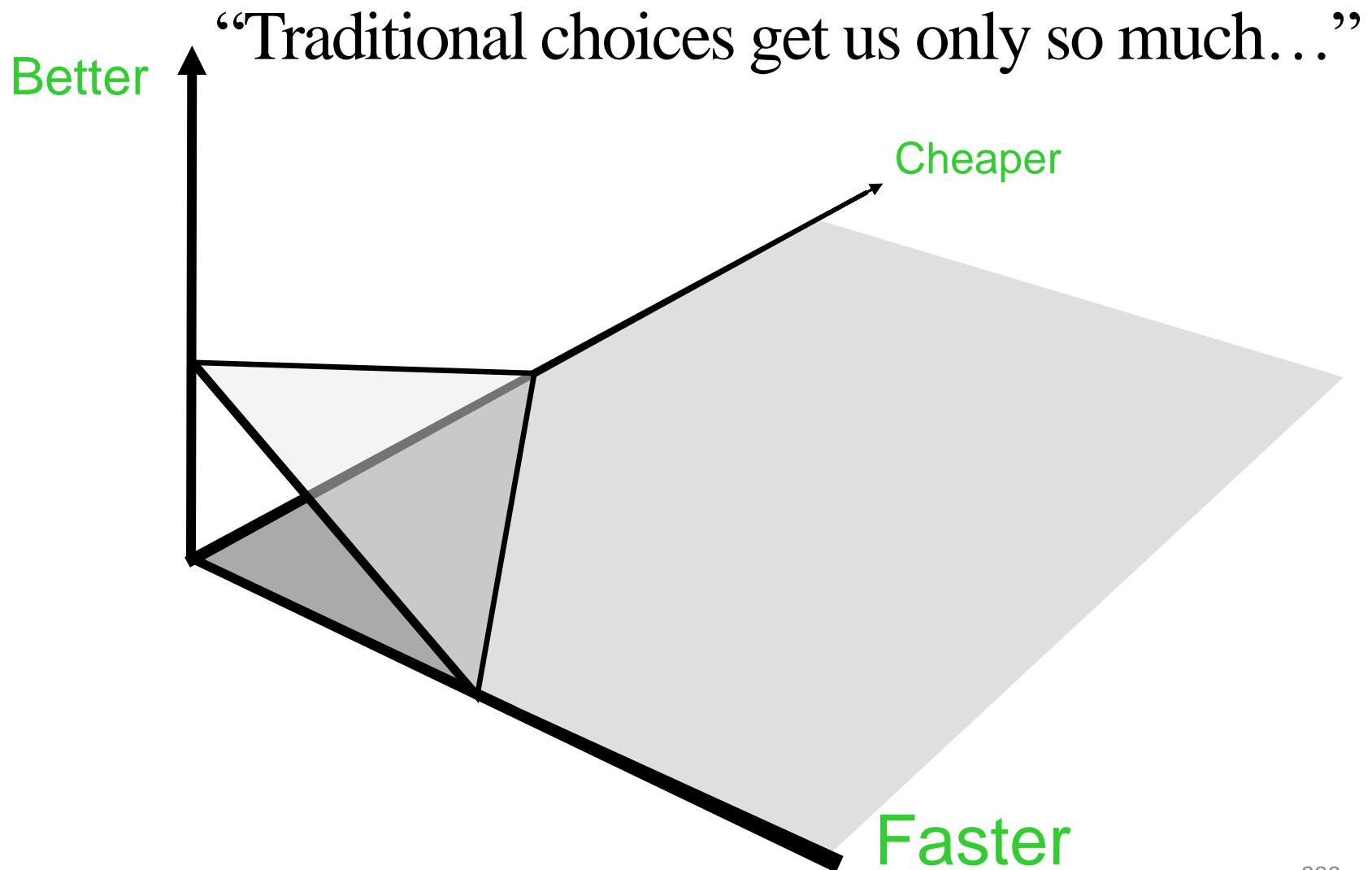
Conclusion

Conclusion

The Goal: Faster, Better, Cheaper!

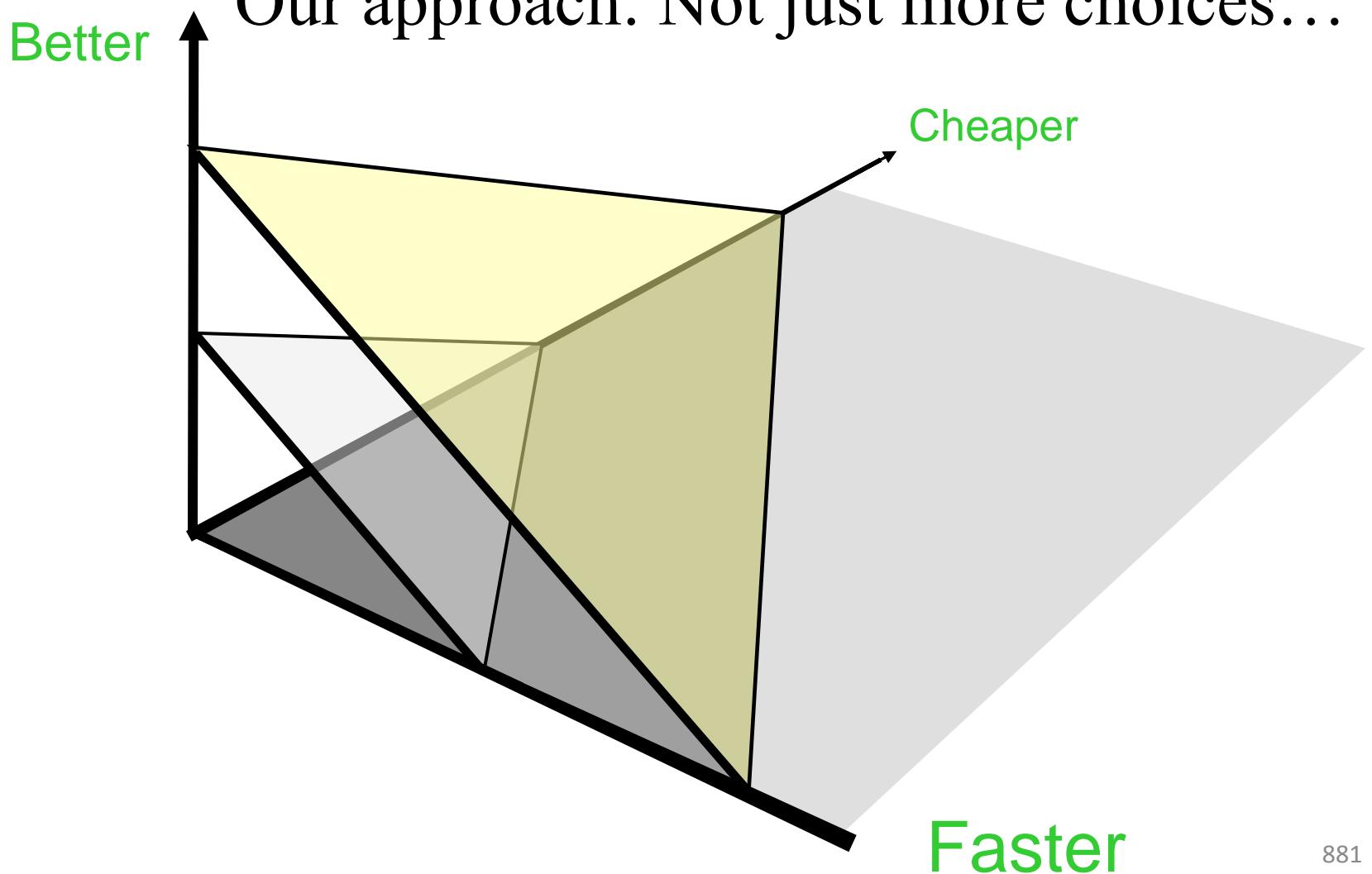


Conclusion



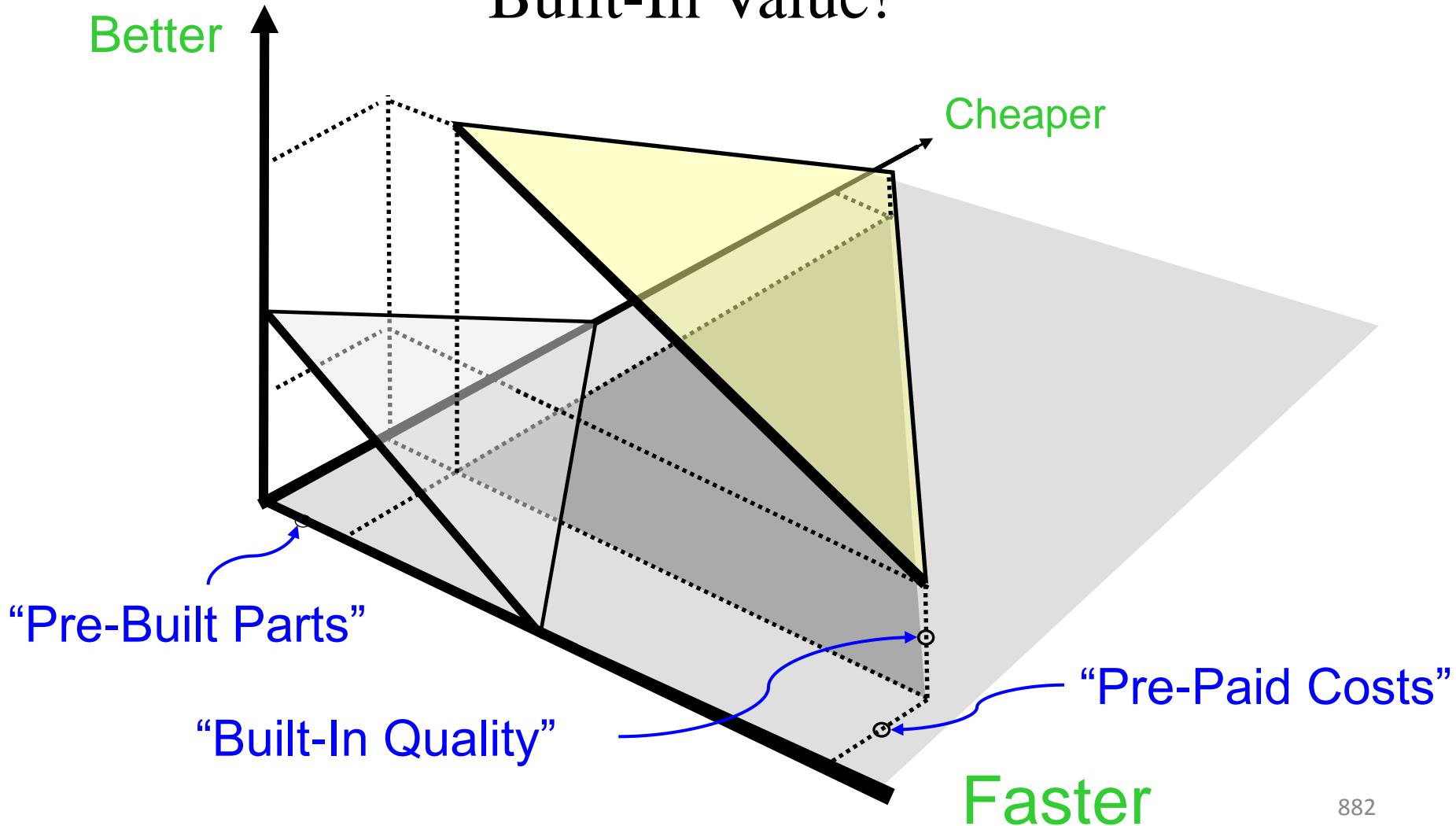
Conclusion

Our approach: Not just more choices...



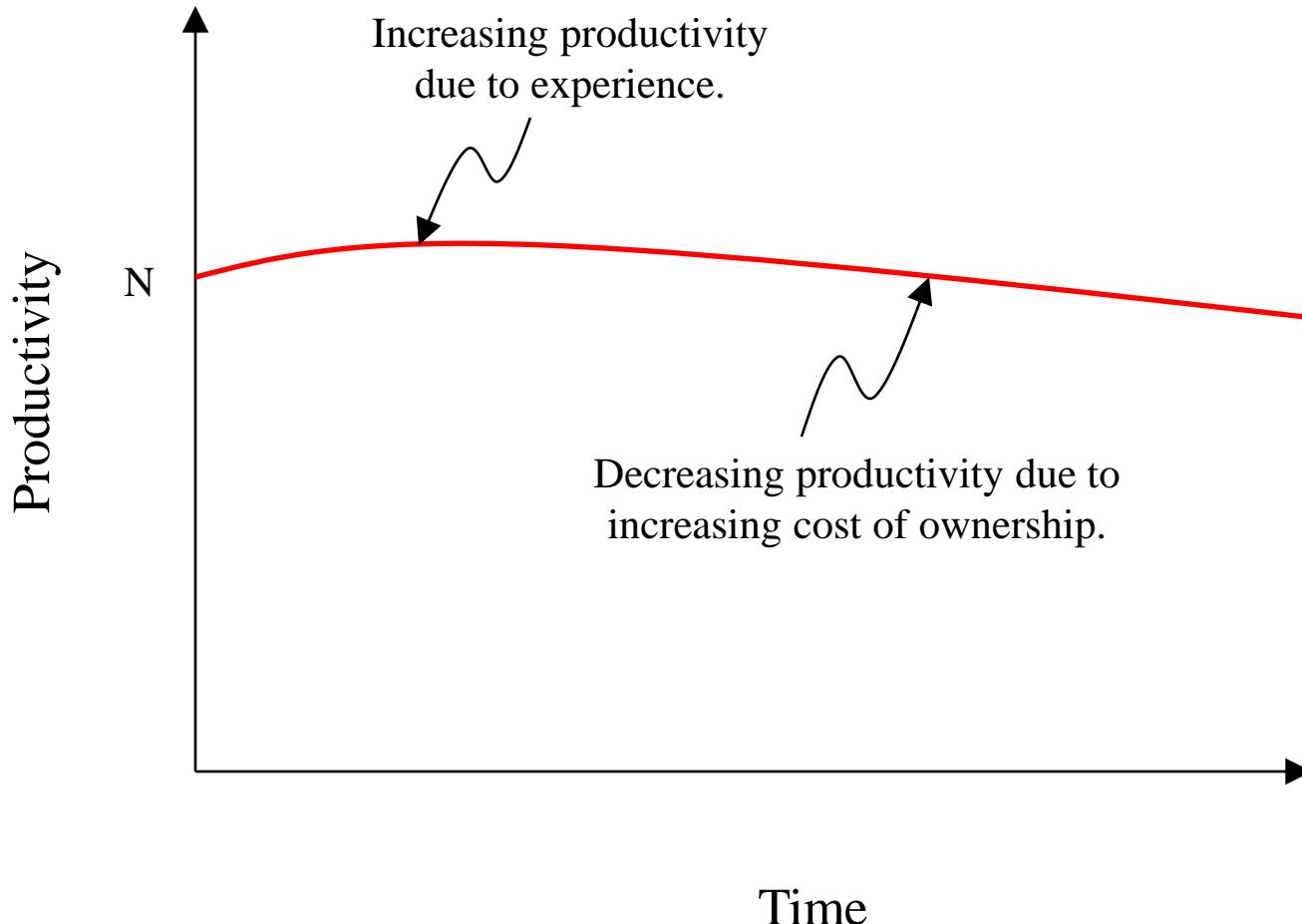
Conclusion

Built-In Value!



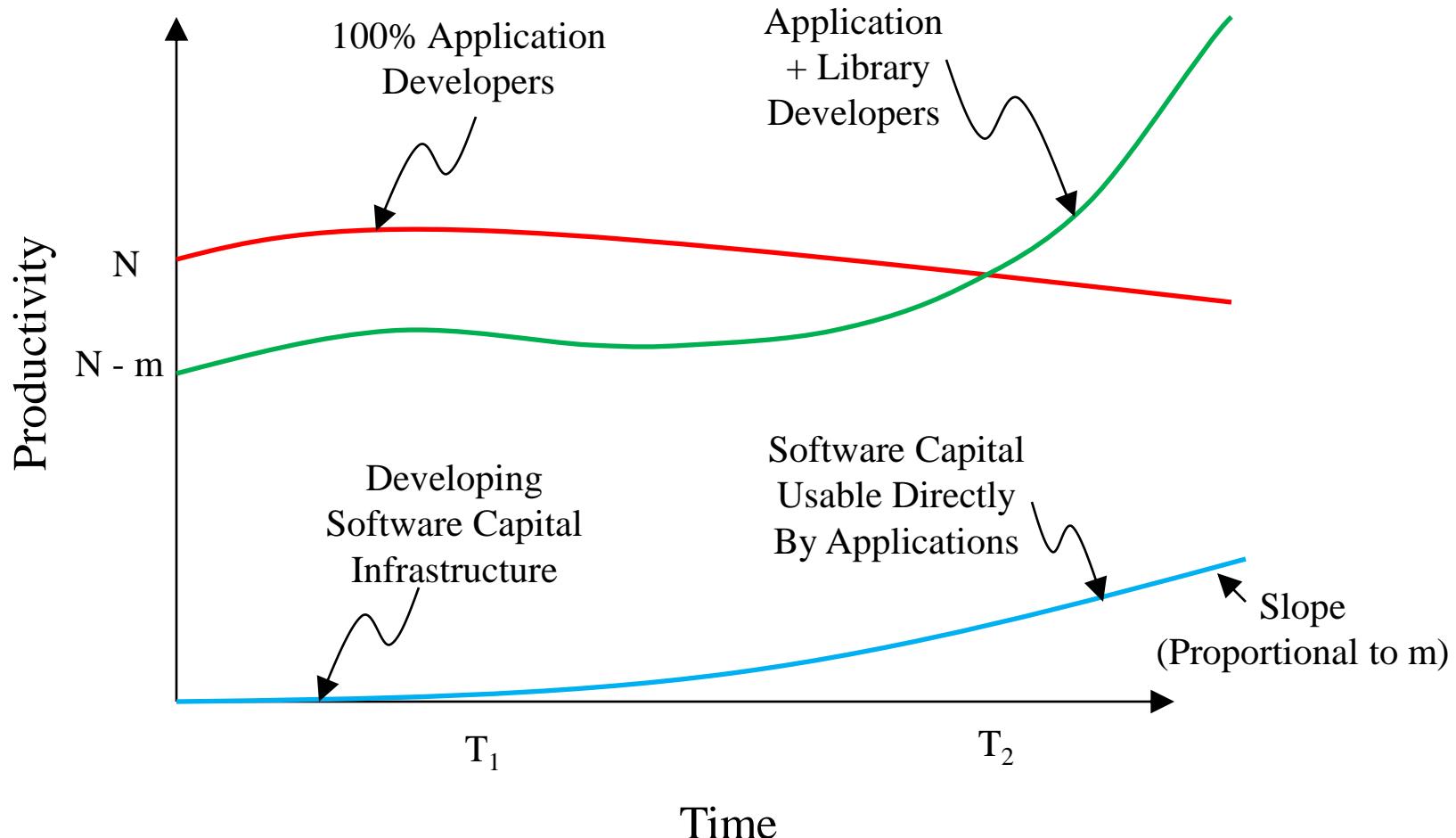
Conclusion

Productivity: **Homogeneous** Development Team



Conclusion

Productivity: Heterogeneous Development Team



Conclusion

So what are the take-aways?

Conclusion

So what are the take-aways?

- We have exhibited a proven methodology that yields hierarchically reusable libraries.

Conclusion

So what are the take-aways?

- We have exhibited a proven methodology that yields hierarchically reusable libraries.
- We are open-sourcing the root of such a hierarchy as a framework and to demonstrate how it is done.

Conclusion

- Find our open-source distribution at:
<http://www.openbloomberg.com/bsl>
- Moderator: kpfleming@bloomberg.net
- How to contribute? *See our site.*
- All comments and criticisms welcome...

Conclusion

- Find our open-source distribution at:
<http://www.openbloomberg.com/bsl>
- Moderator: kpfleming@bloomberg.net
- How to contribute? *See our site.*
- All comments and criticisms welcome...

The End