Organizing Principles

Sound Physical Design:

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

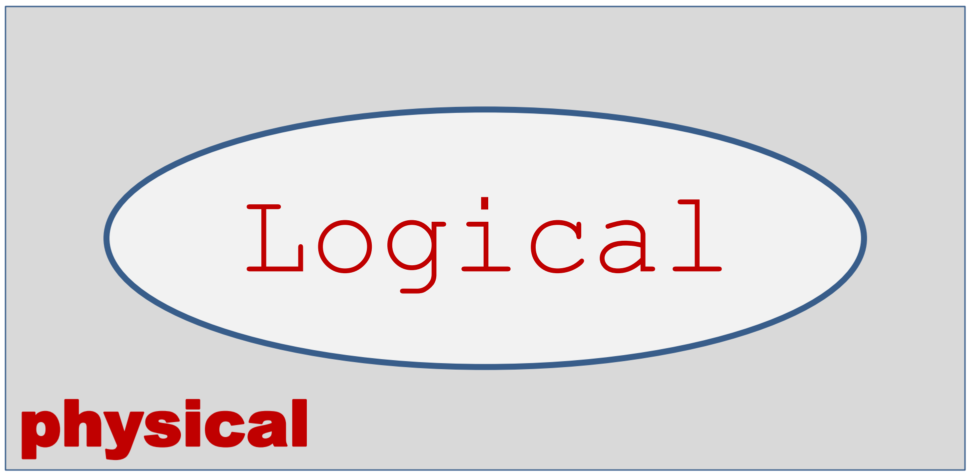
Organizing Principles

Sound Physical Design:

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

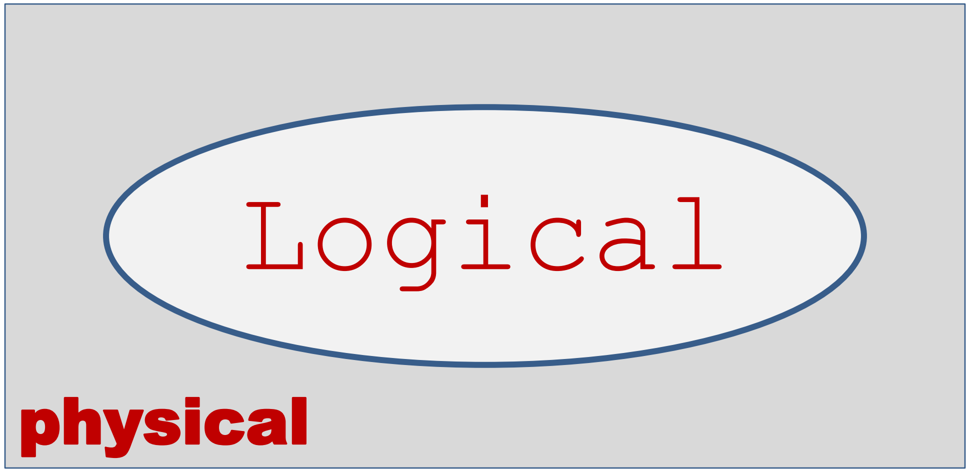
Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?



Logical versus Physical Design

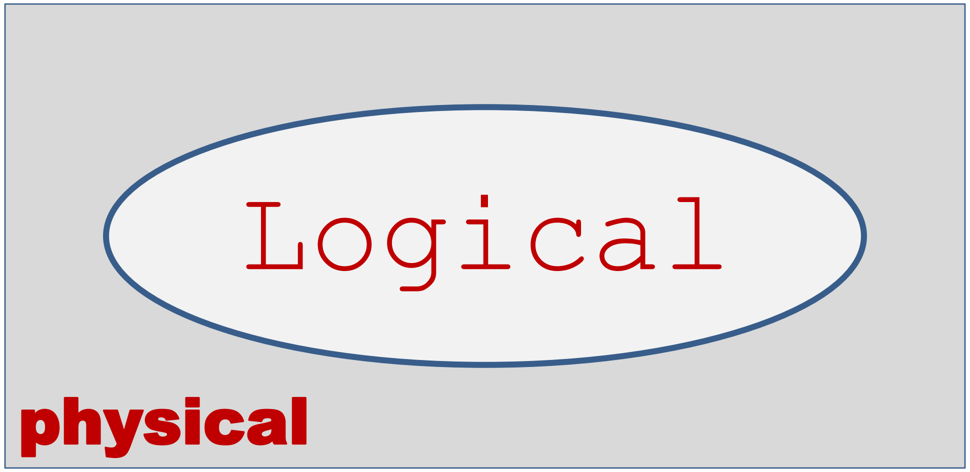
What distinguishes *Logical* from *Physical* Design?



Logical: Classes and Functions

Logical versus Physical Design

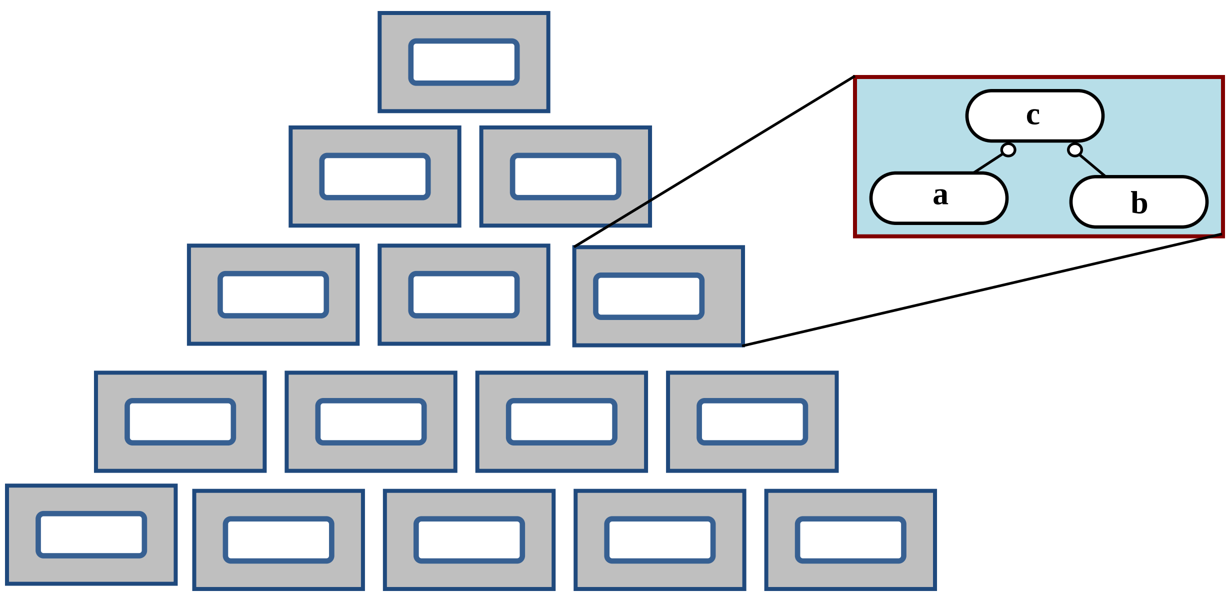
What distinguishes *Logical* from *Physical* Design?



Logical: Classes and Functions
Physical: Files and Libraries

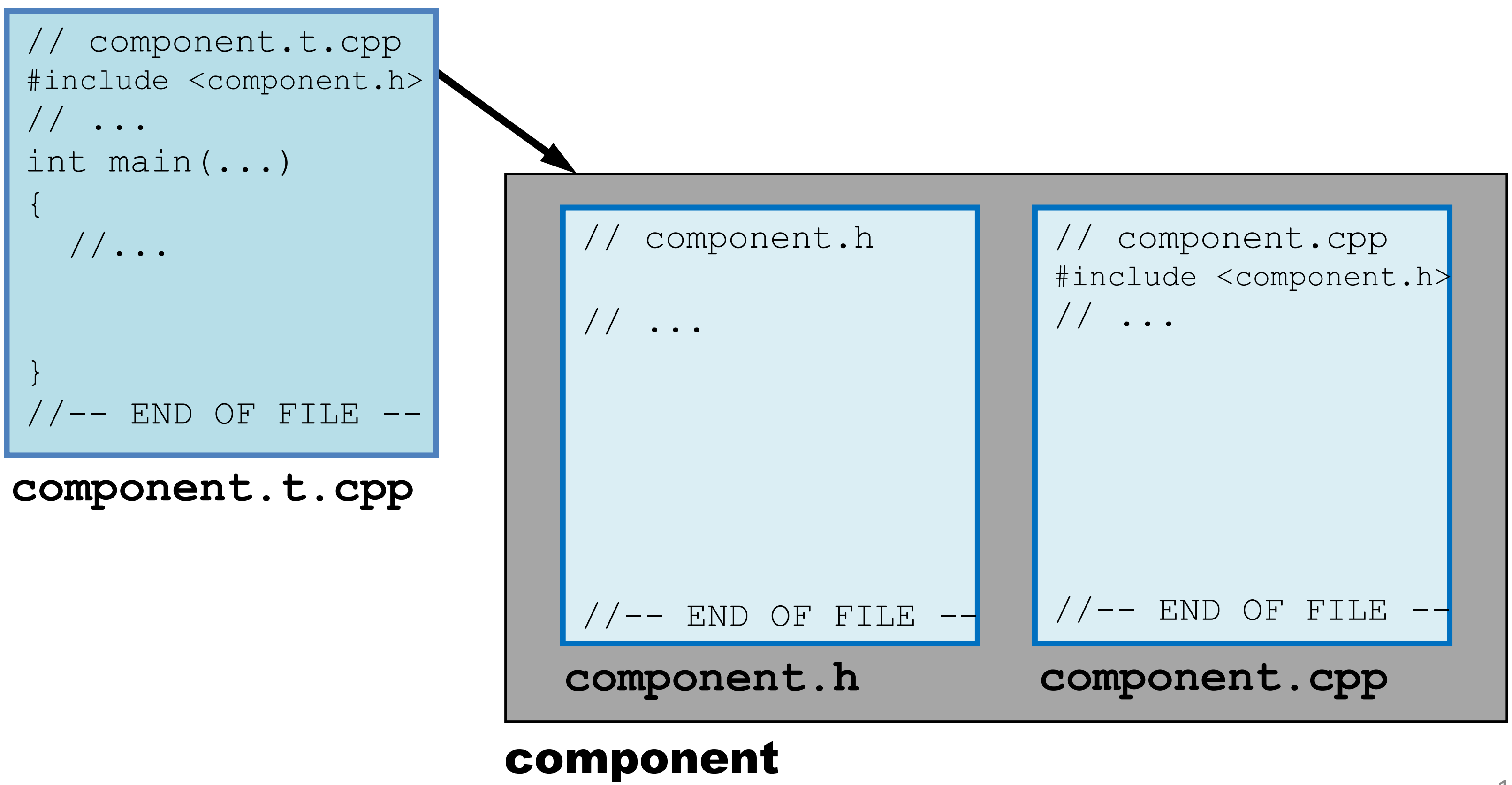
Logical versus Physical Design

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

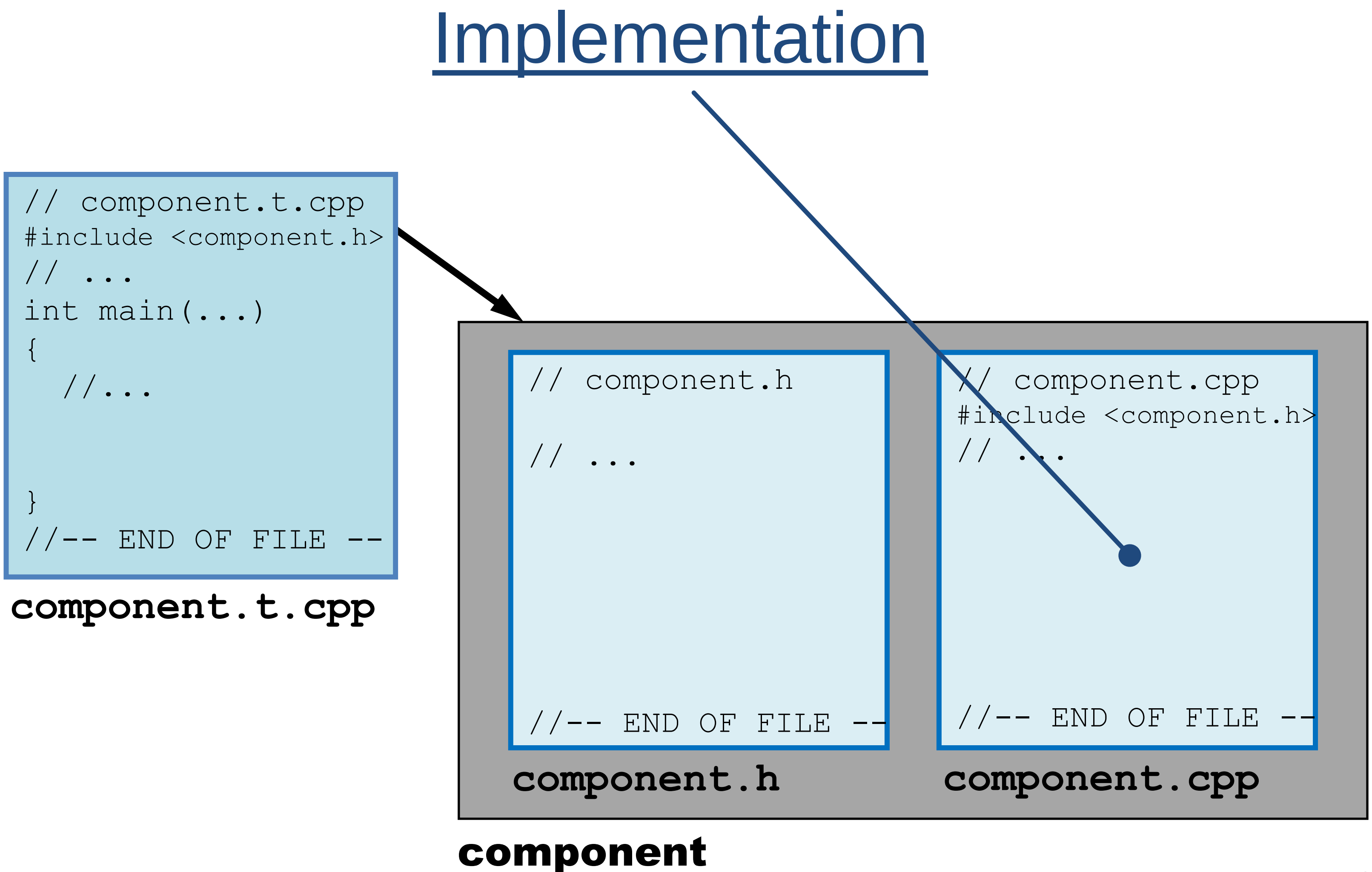


Component: Uniform Physical Structure

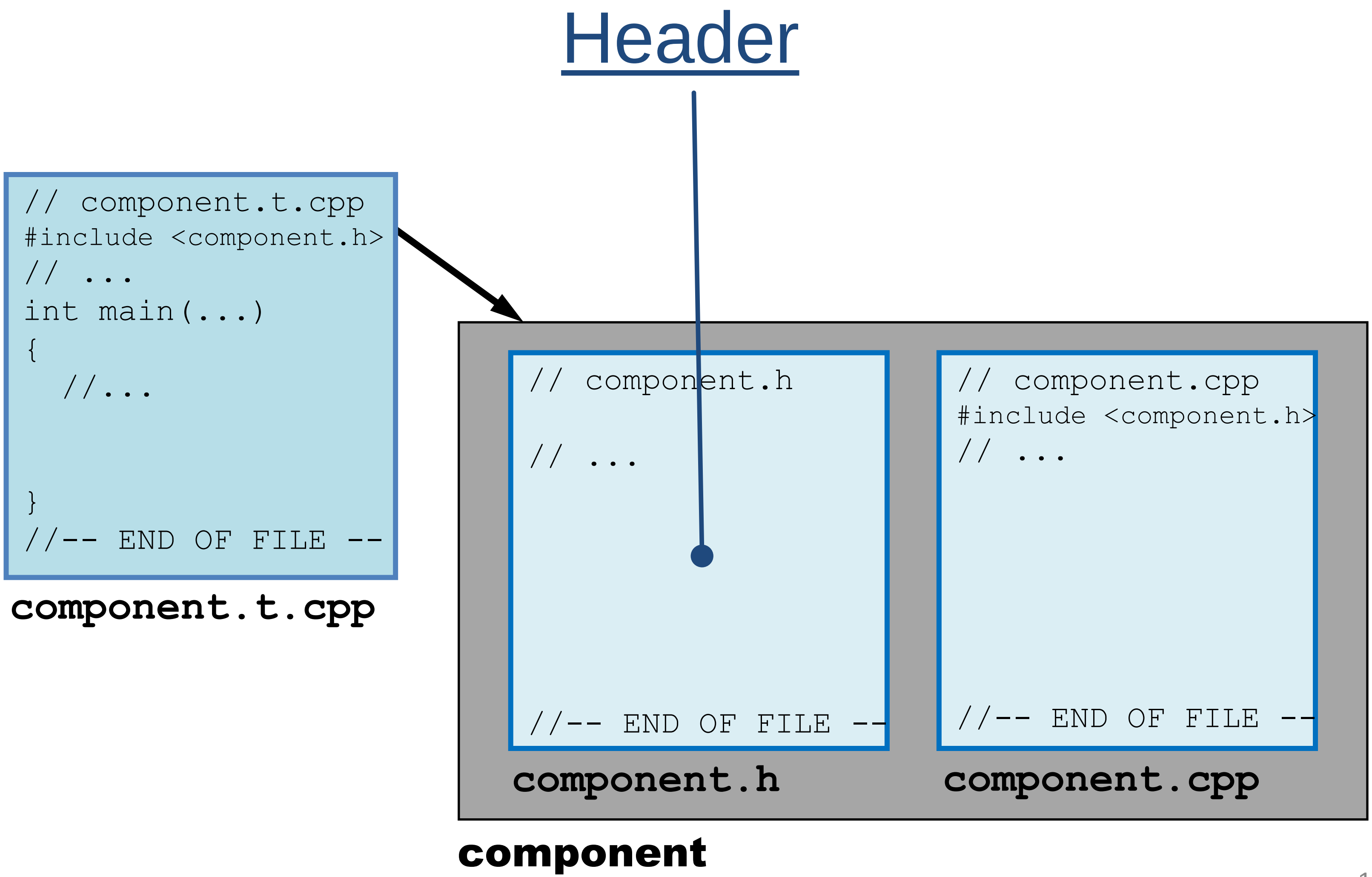
A Component Is Physical



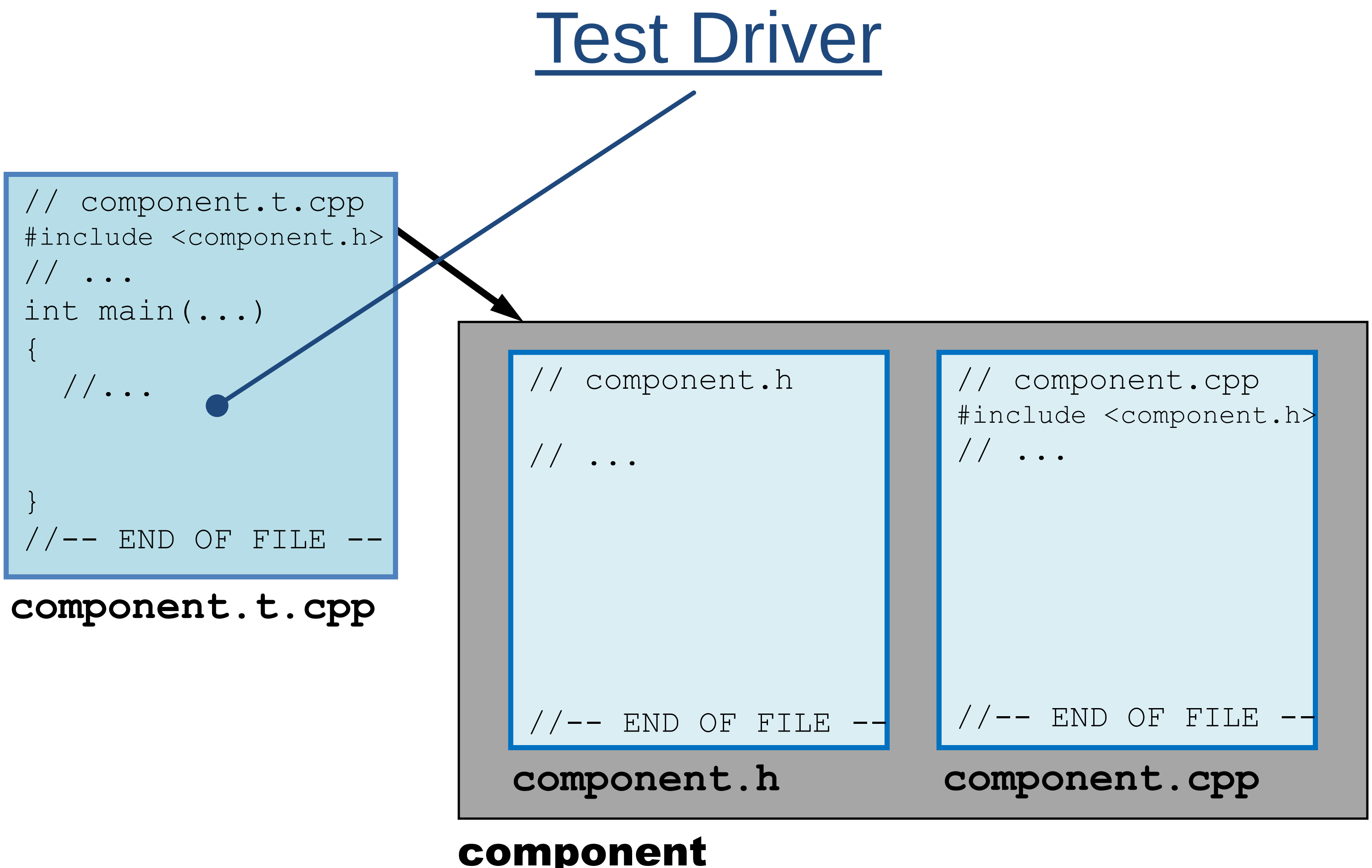
Component: Uniform Physical Structure



Component: Uniform Physical Structure

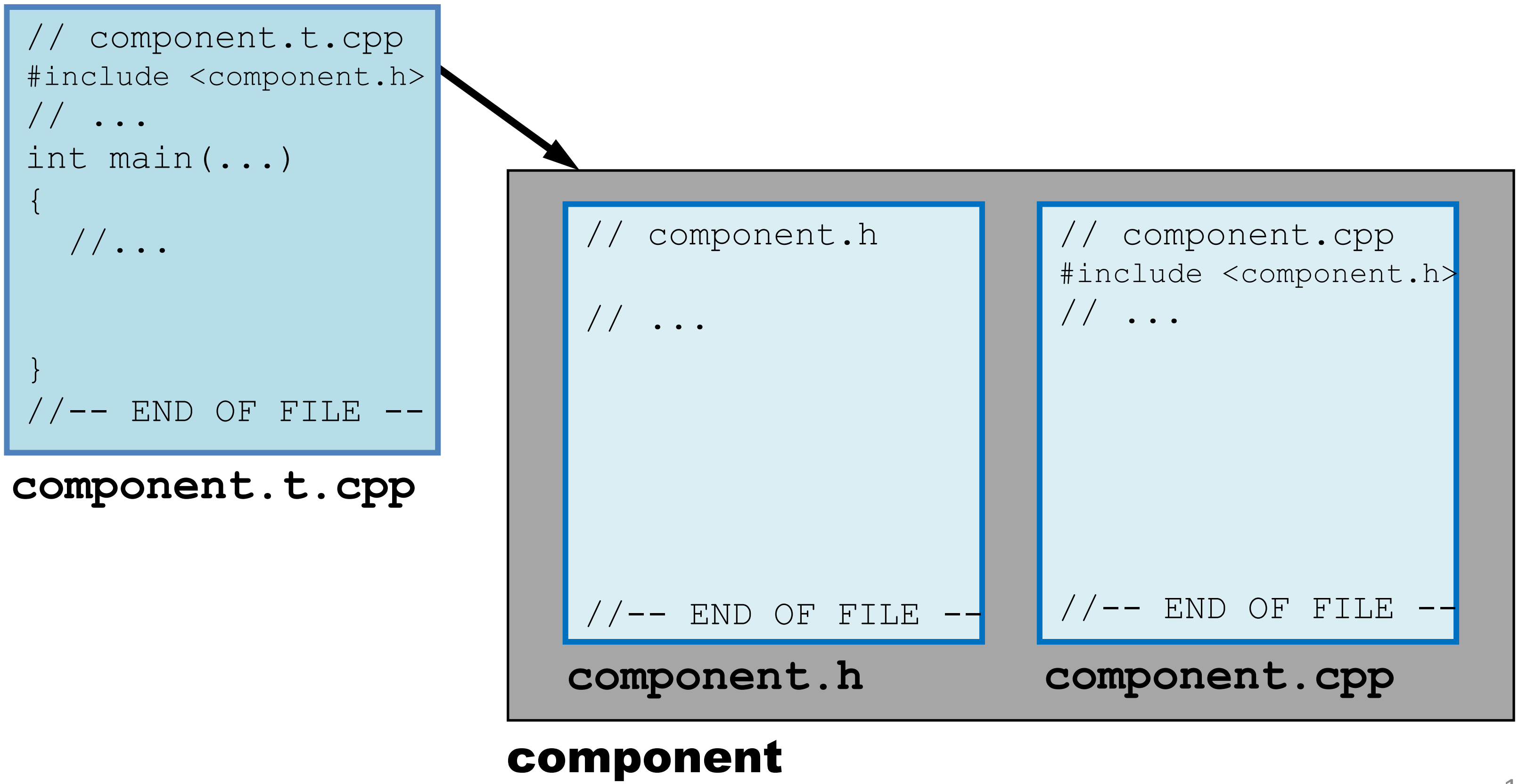


Component: Uniform Physical Structure

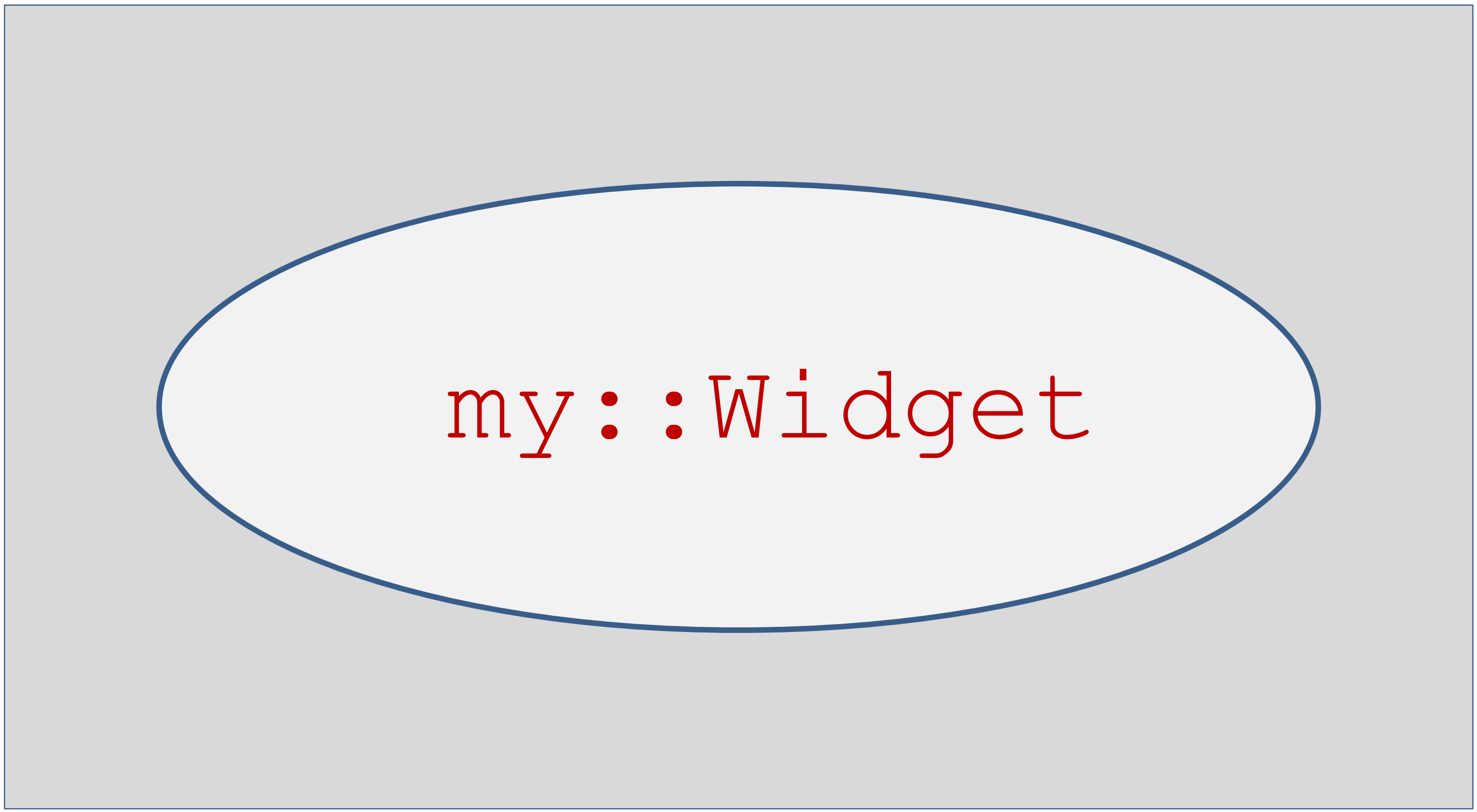


Component: Uniform Physical Structure

<u>The</u> Fundamental Unit of Design

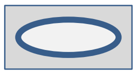


Component: Not Just a `.h` /`.cpp` Pair



`my_widget`

Component: Not Just a `.h` / `.cpp` Pair

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

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!

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.

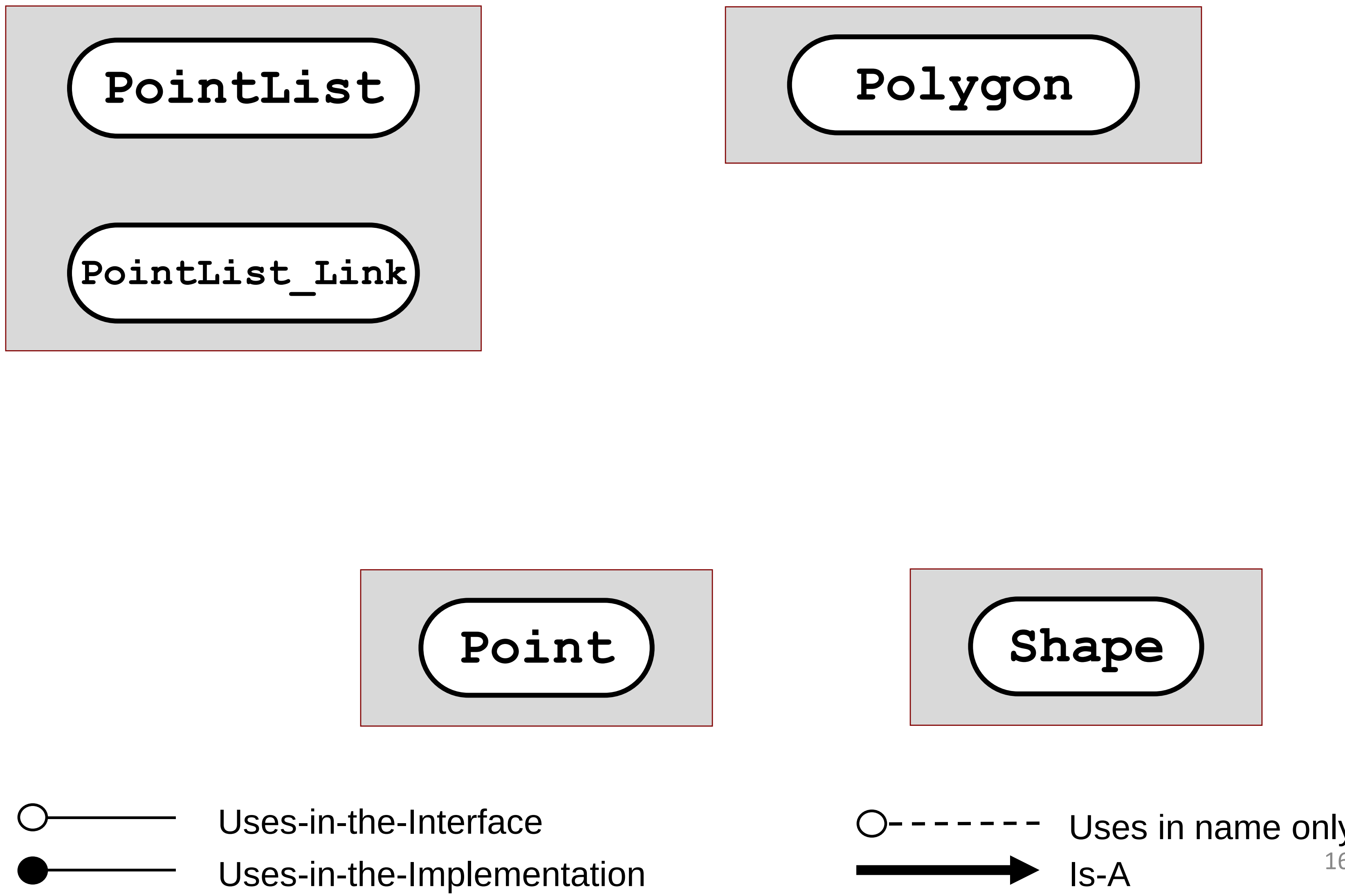
Component: Not Just a `.h` /`.cpp` Pair

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.

Component: Not Just a `.h` / `.cpp` Pair

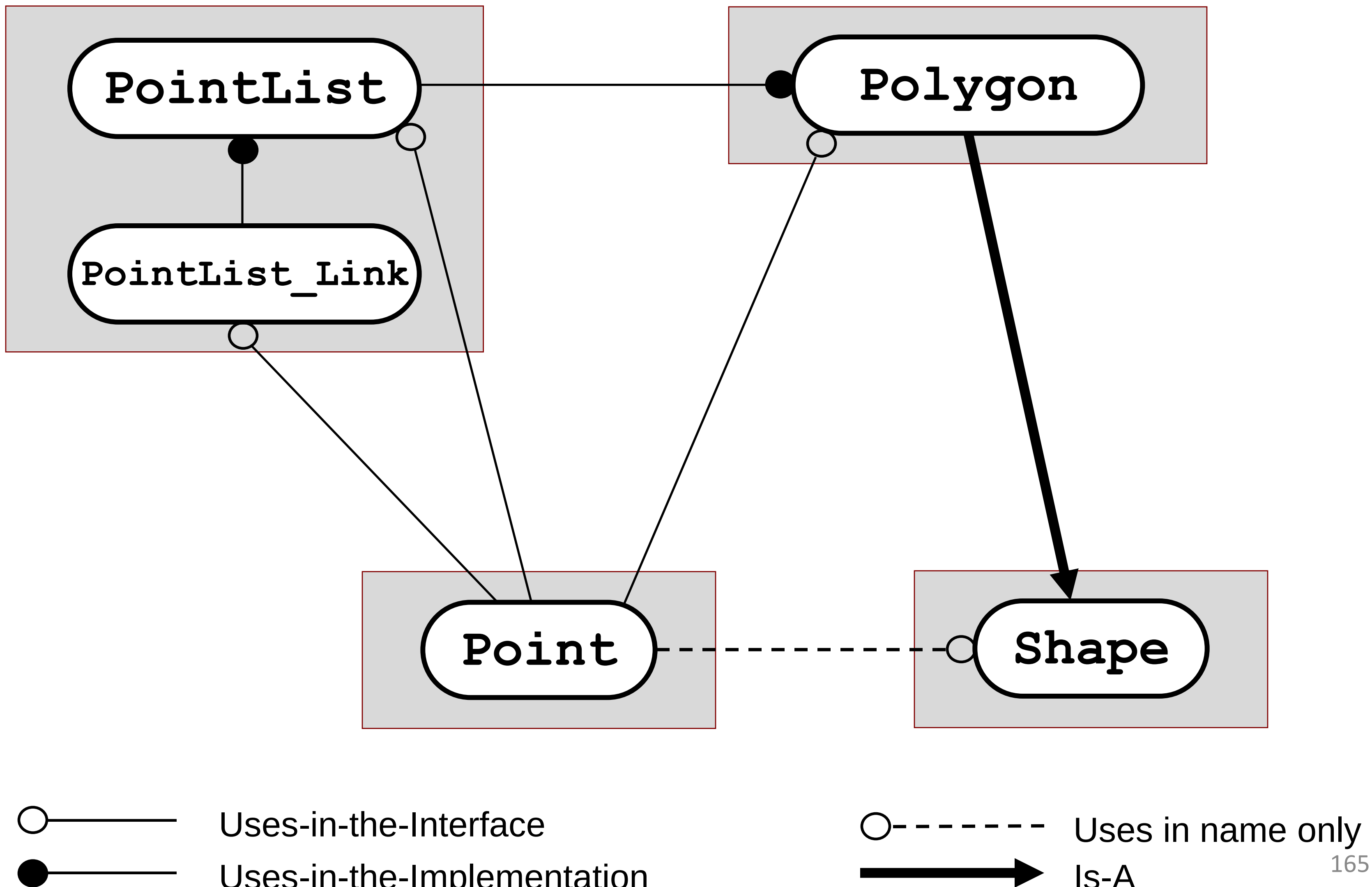
1. The **`.cpp`** file includes its **`.h`** file as the first substantive line of code.
2. All logical constructs (effectively) having external physical linkage defined in a **`.cpp`** file are declared in the corresponding **`.h`** file.
3. All constructs having external physical linkage declared in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

Logical Relationships

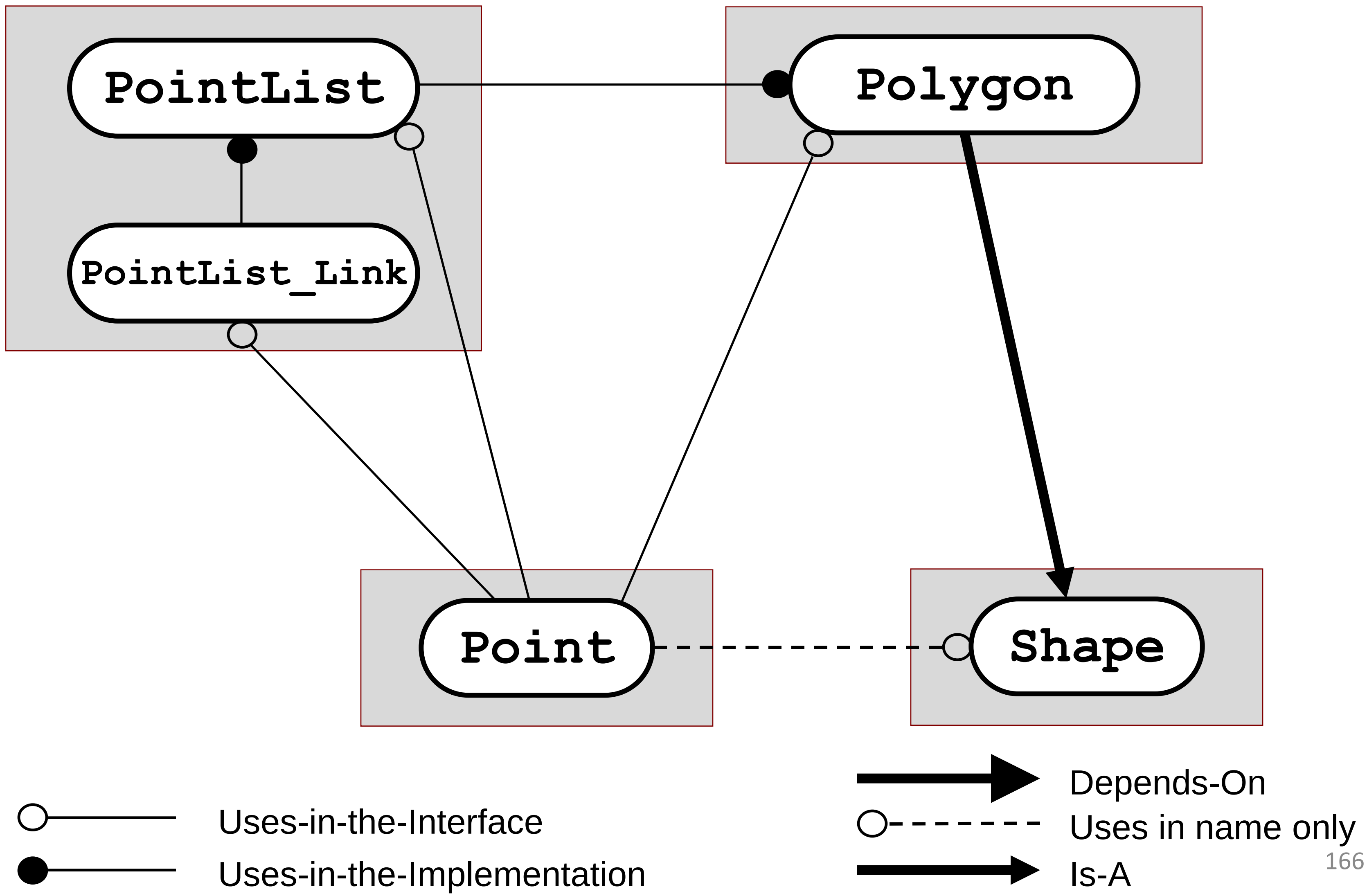


1. Process & Architecture

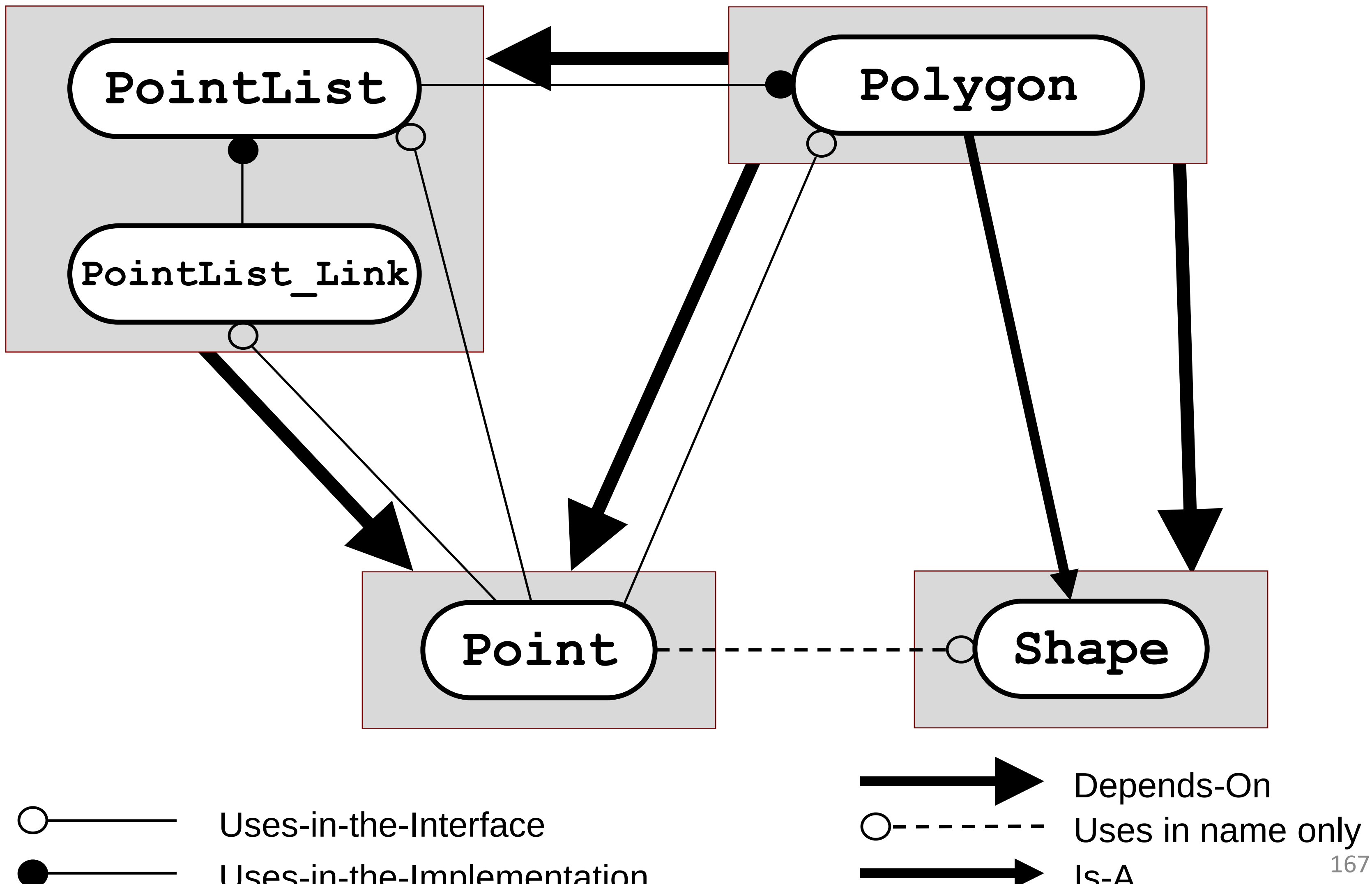
Logical Relationships



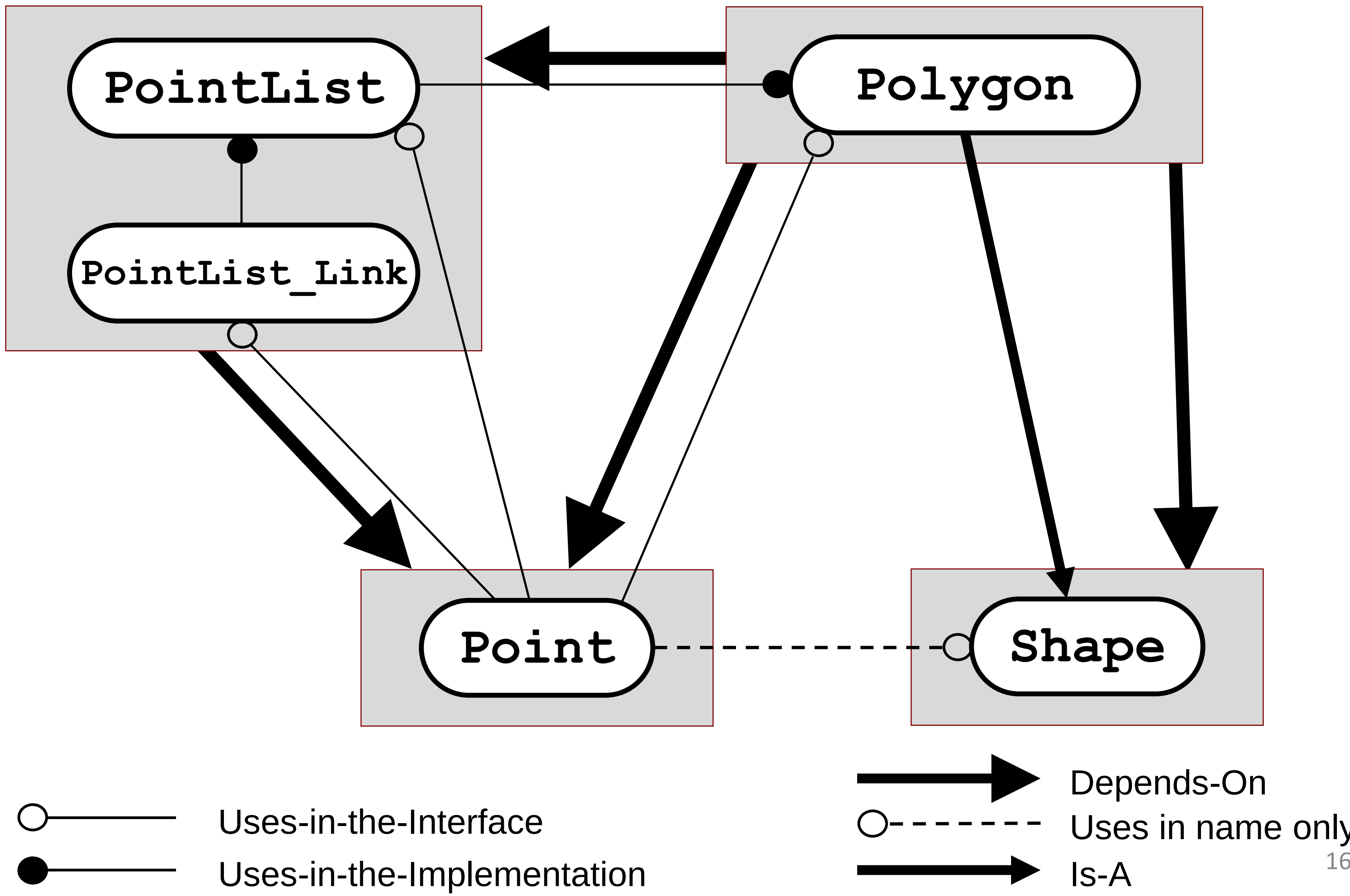
Implied Dependency



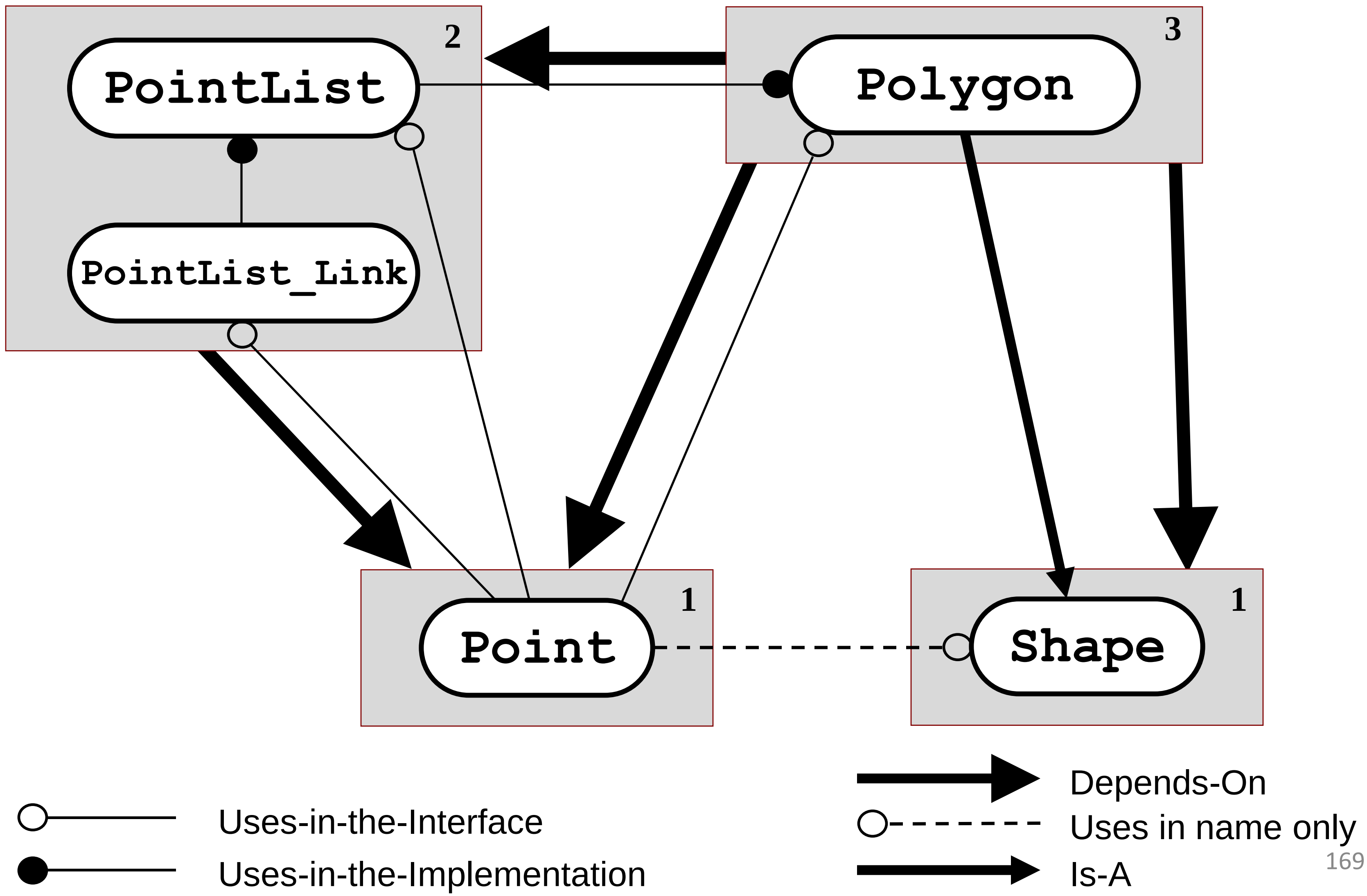
Implied Dependency



Level Numbers



Level Numbers



Levelization

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

Levelization

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

Usage:

Levelization

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

Usage:

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.

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.

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?

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?

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?

Levelization

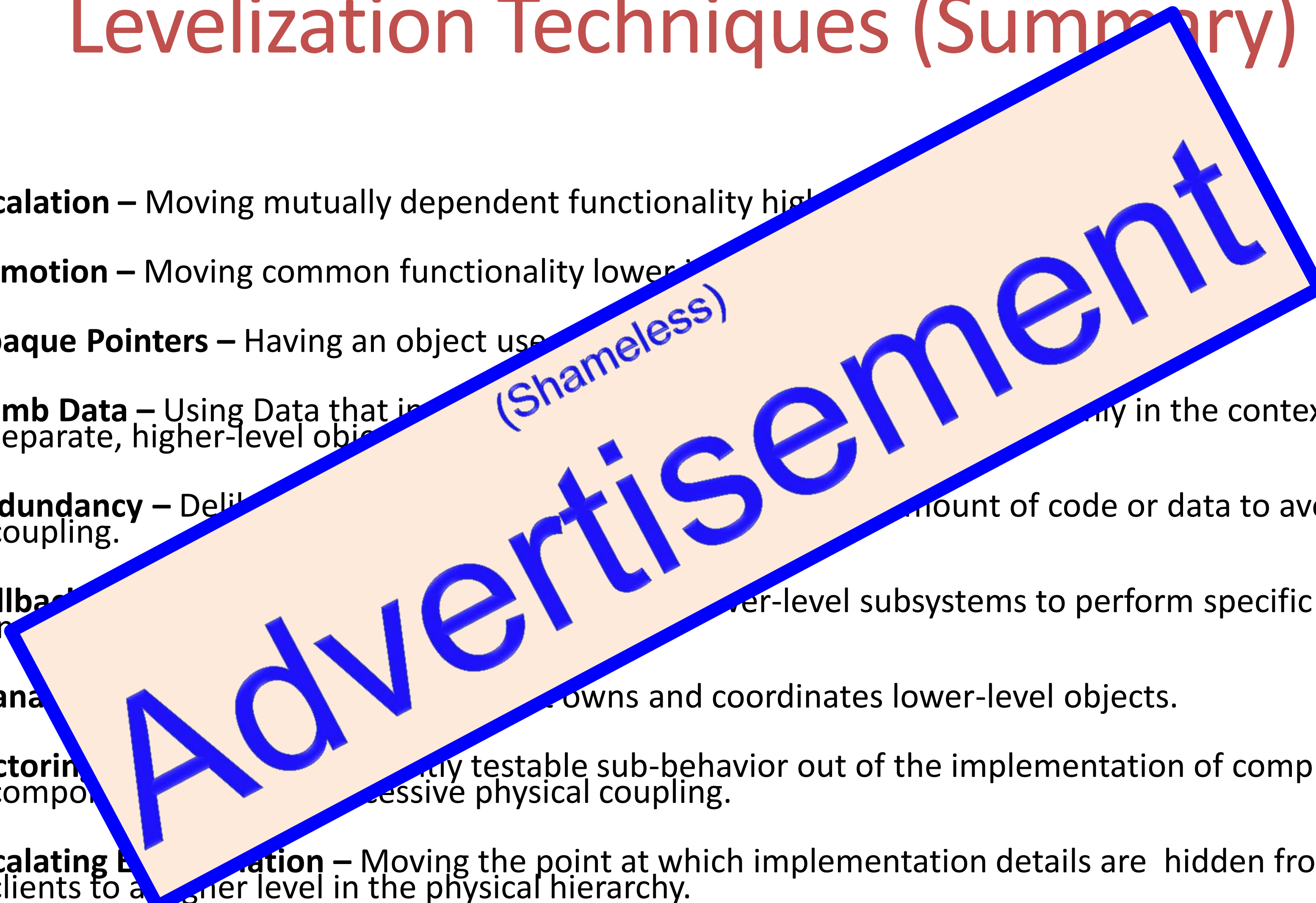
Levelize (v.); Levelizable (a.); Levelization (n.)

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.***

Levelization Techniques (Summ ary)



Escalation – Moving mutually dependent functionality hig

Demotion – Moving common functionality lower i

Opaque Pointers – Having an object use

Dumb Data – Using Data that i ... ly in the context of a separate, higher-level obj

Redundancy – Deli ... mount of code or data to avoid coupling.

Callba ... er-level subsystems to perform specific tasks i

Mana ... owns and coordinates lower-level objects.

Factorin ... tly testable sub-behavior out of the implementation of complex compo ... essive physical coupling.

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

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

Essential Physical Design Rules

There are two:

Essential Physical Design Rules

There are two:

1. No *Cyclic* Physical Dependencies!

Essential Physical Design Rules

There are two:

1. No *Cyclic* Physical Dependencies!

2. No *Long-Distance* Friendships!

Criteria for Collocating "Public" Classes

Criteria for Collocating “Public” Classes

There are four:

Criteria for Collocating “Public” Classes

There are four:

1. Friendship.

Criteria for Collocating "Public" Classes

There are four:

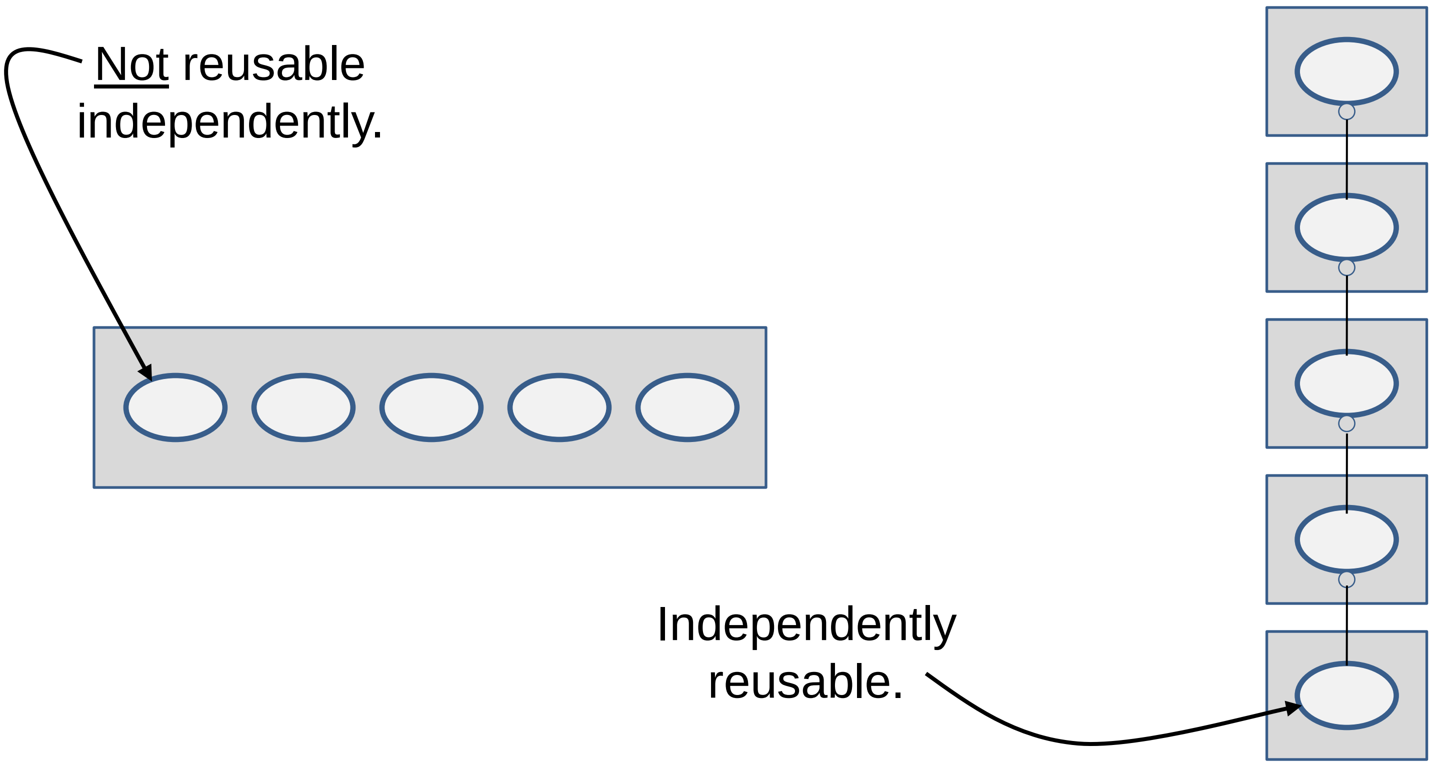
1. Friendship.
2. Cyclic Dependency.

Criteria for Collocating "Public" Classes

There are four:

1. Friendship.
2. Cyclic Dependency.
3. Single Solution.

Criteria for Collocating "Public" Classes



Single Solution

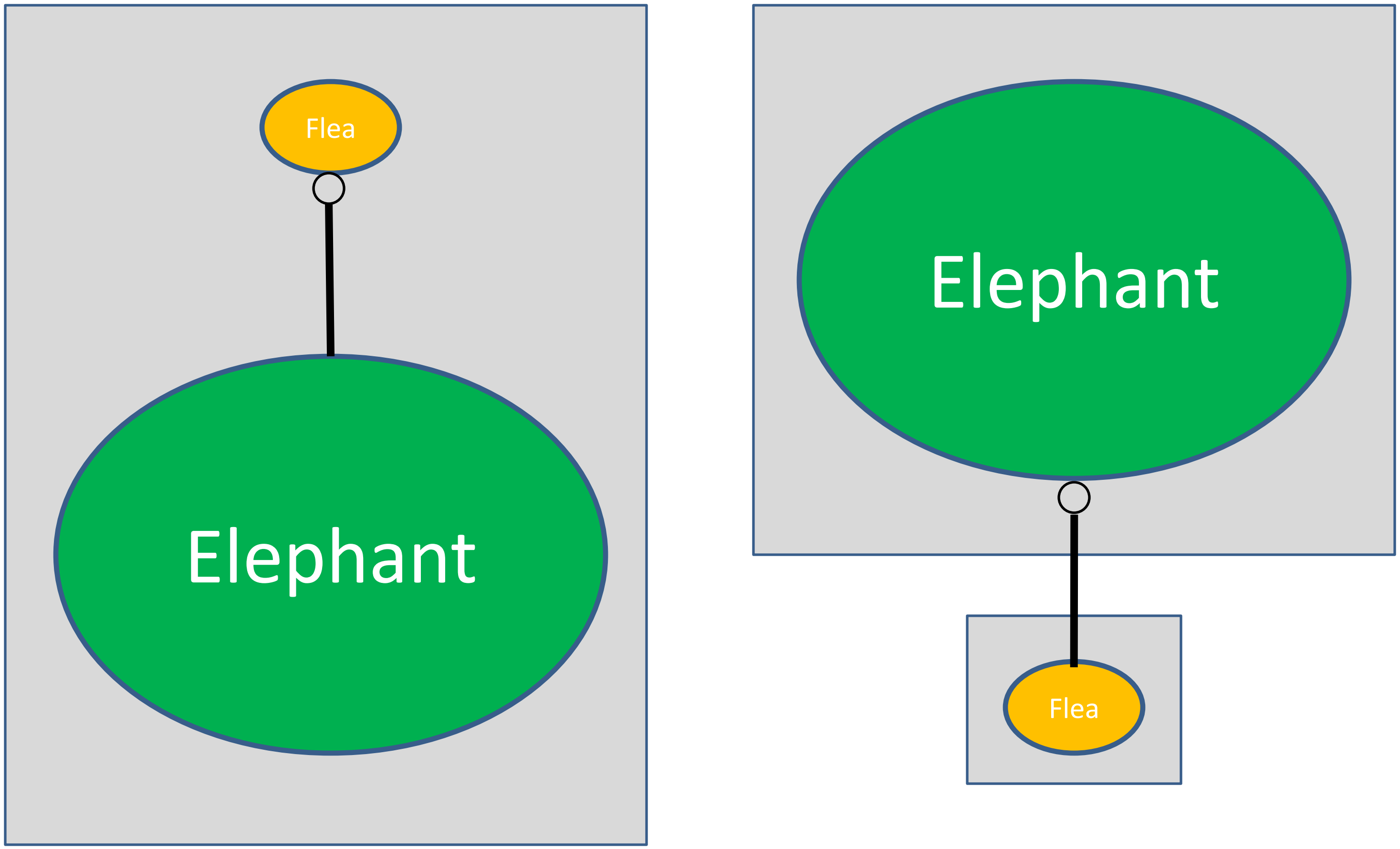
Hierarchy of Solutions

Criteria for Collocating “Public” Classes

There are four:

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

Criteria for Collocating "Public" Classes



"Flea on an Elephant" (Elephant on a Flea)

Organizing Principles

Sound Physical Design:

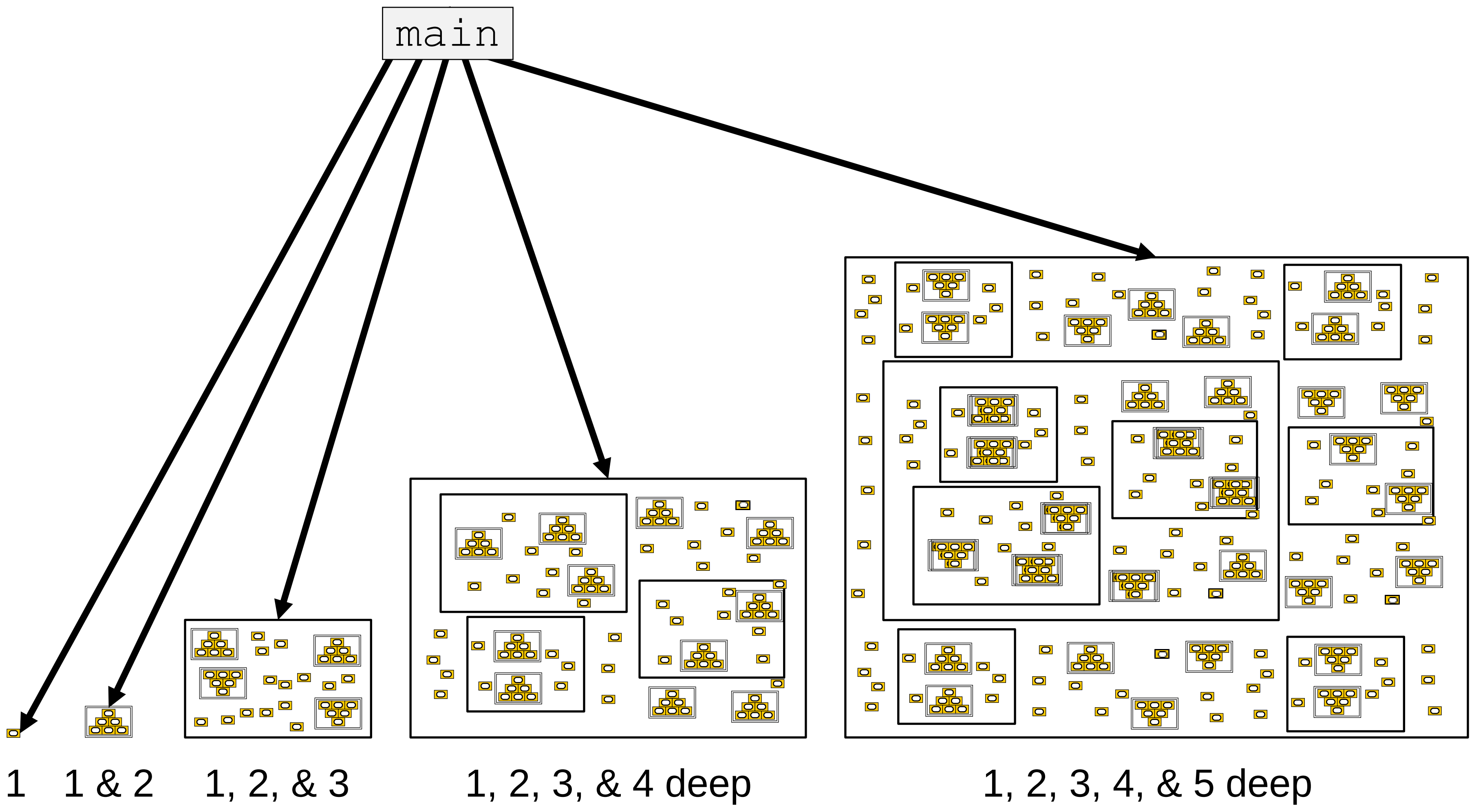
- Regular, Fine-Grained Physical Packaging.
- Uniform Depth of Physical Aggregation.
- Logical/Physical Synergy.

Organizing Principles

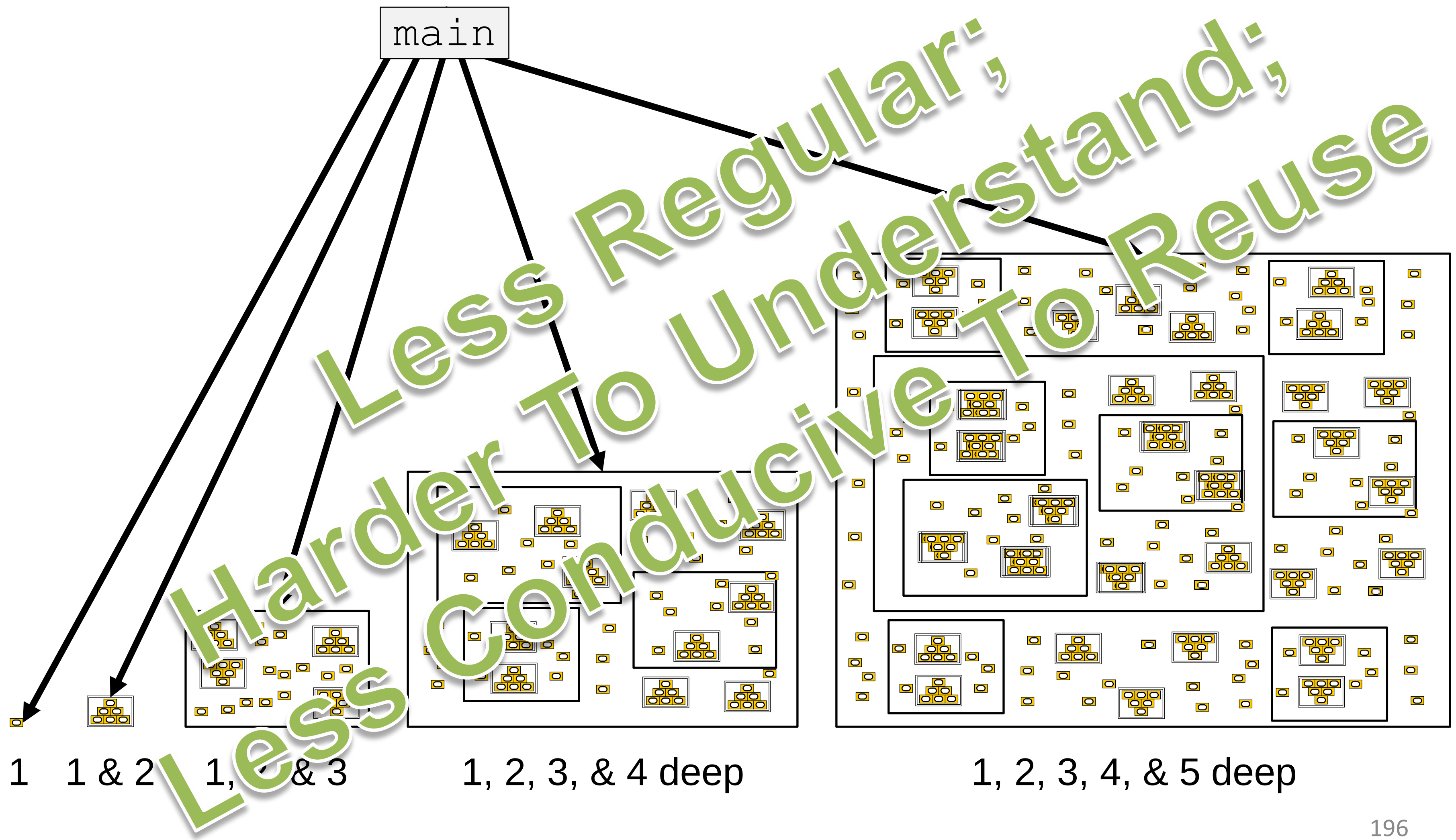
Sound Physical Design:

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

Uniform Depth of Physical Aggregation

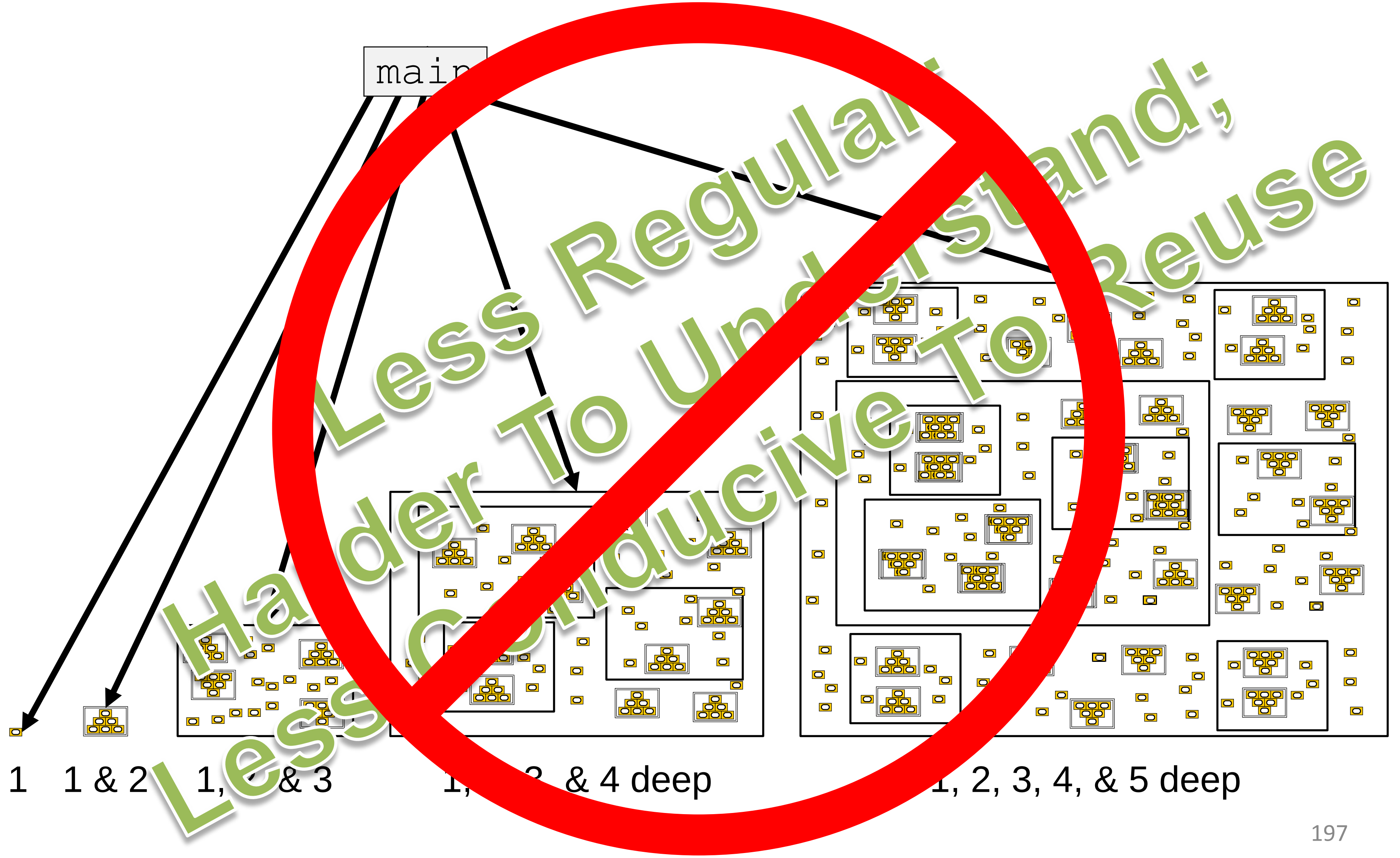


Uniform Depth of Physical Aggregation

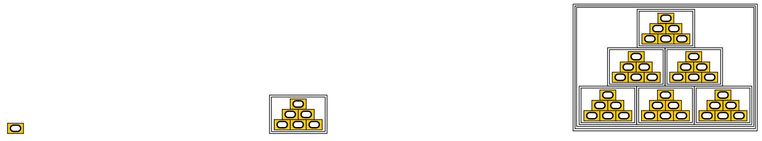


1. Process & Architecture

Uniform Depth of Physical Aggregation



Uniform Depth of Physical Aggregation



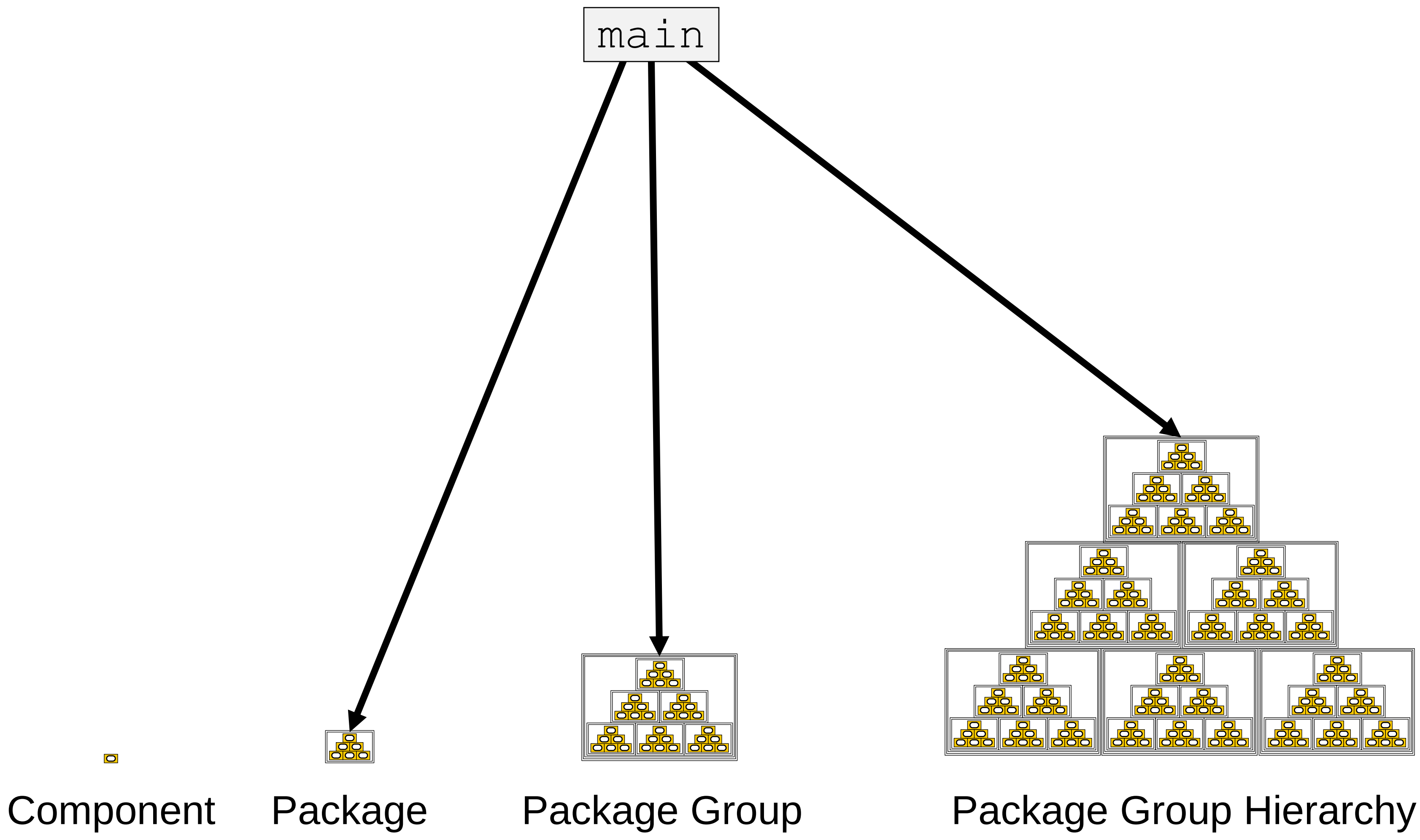
1. Process & Architecture

Uniform Depth of Physical Aggregation

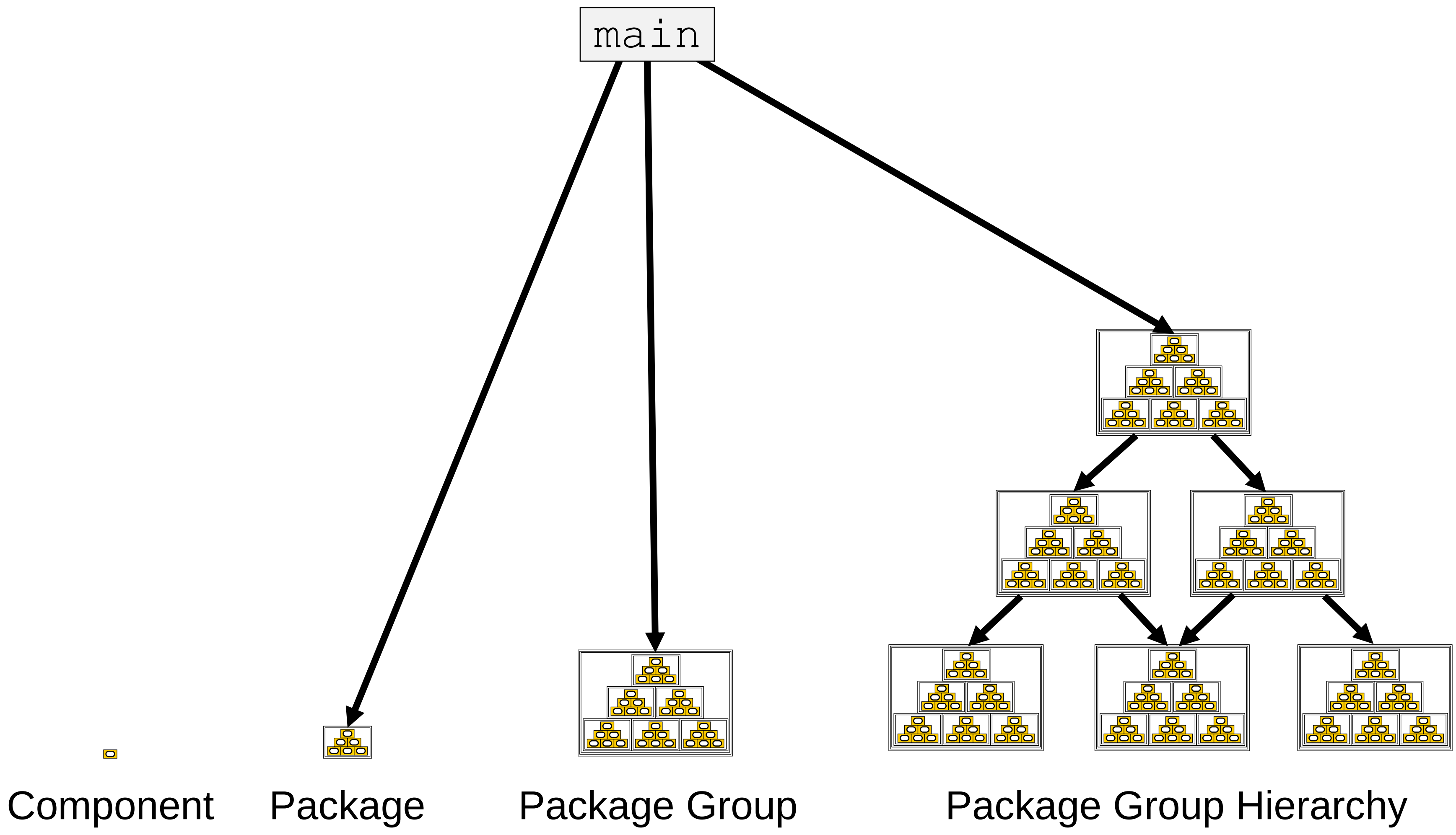
Exactly Three Levels of Physical Aggregation

Component Package Package Group

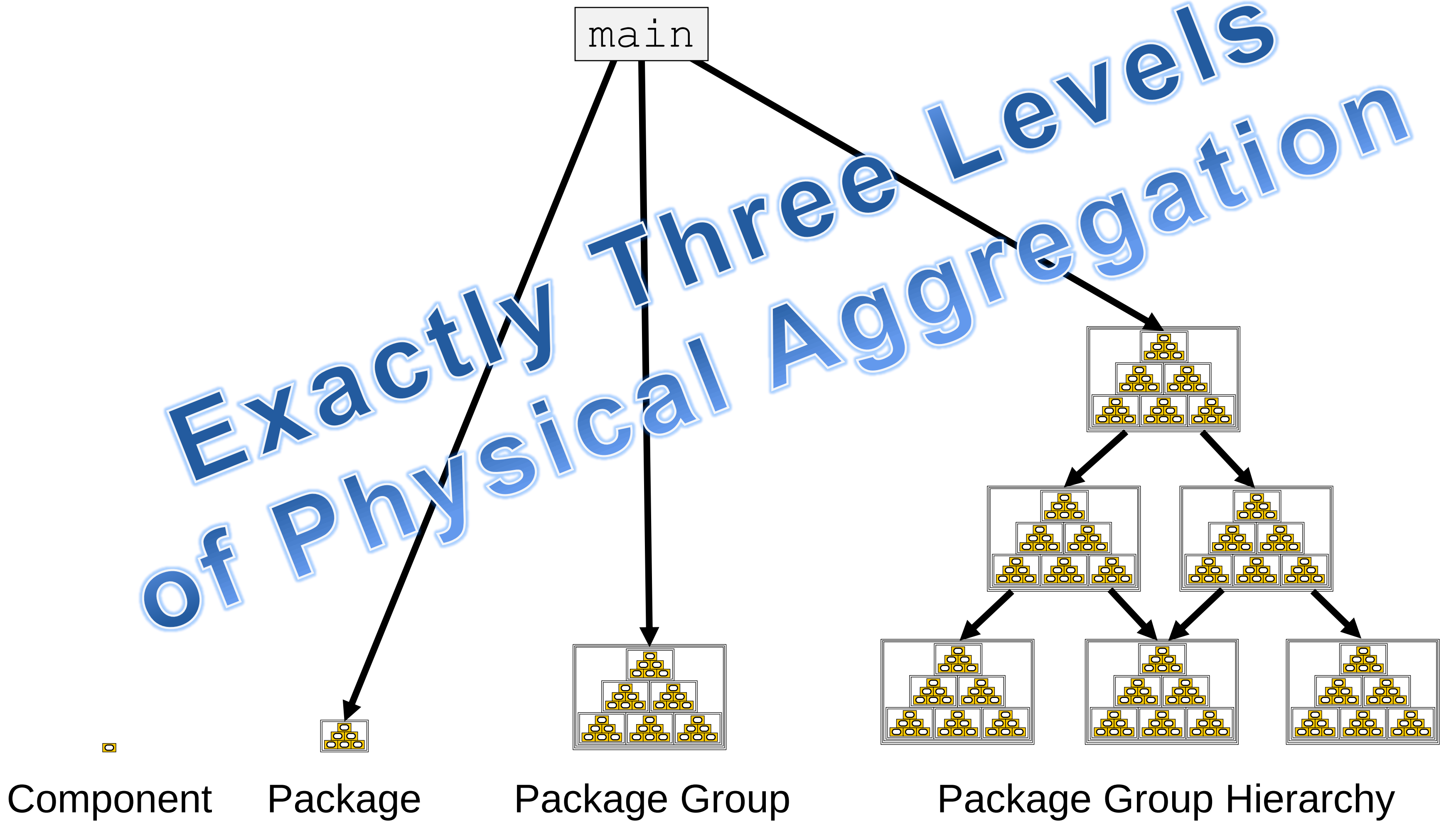
Uniform Depth of Physical Aggregation



Uniform Depth of Physical Aggregation



Uniform Depth of Physical Aggregation



Components

Five levels of **physical dependency**:

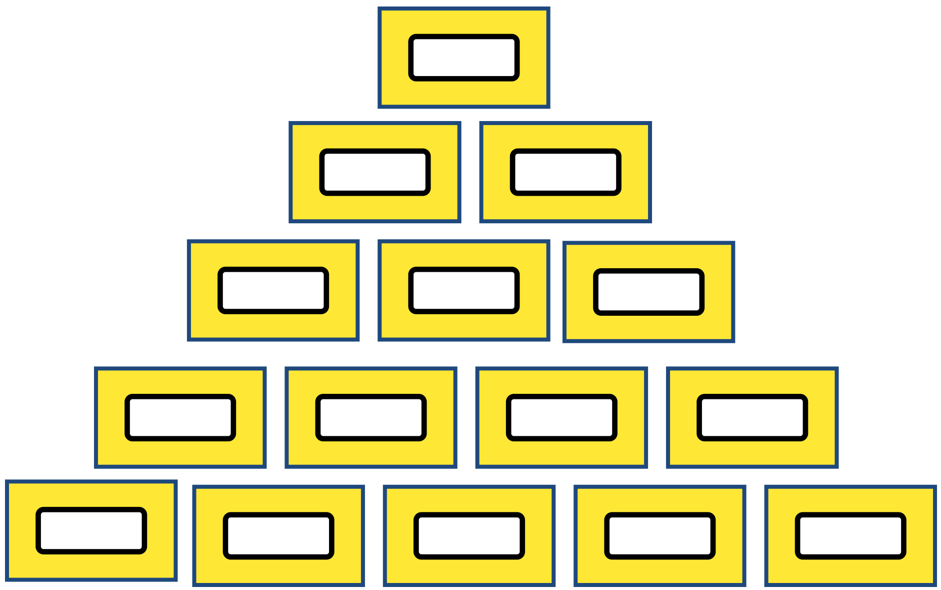
Level 5:

Level 4:

Level 3:

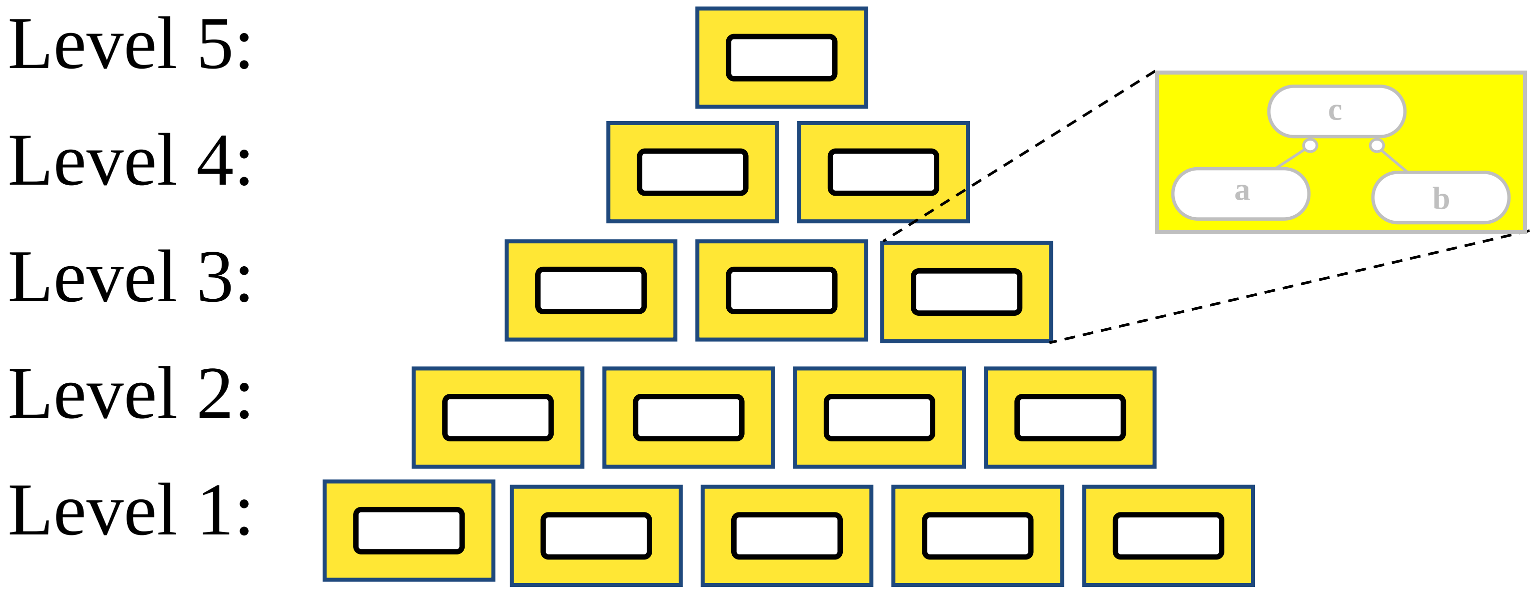
Level 2:

Level 1:



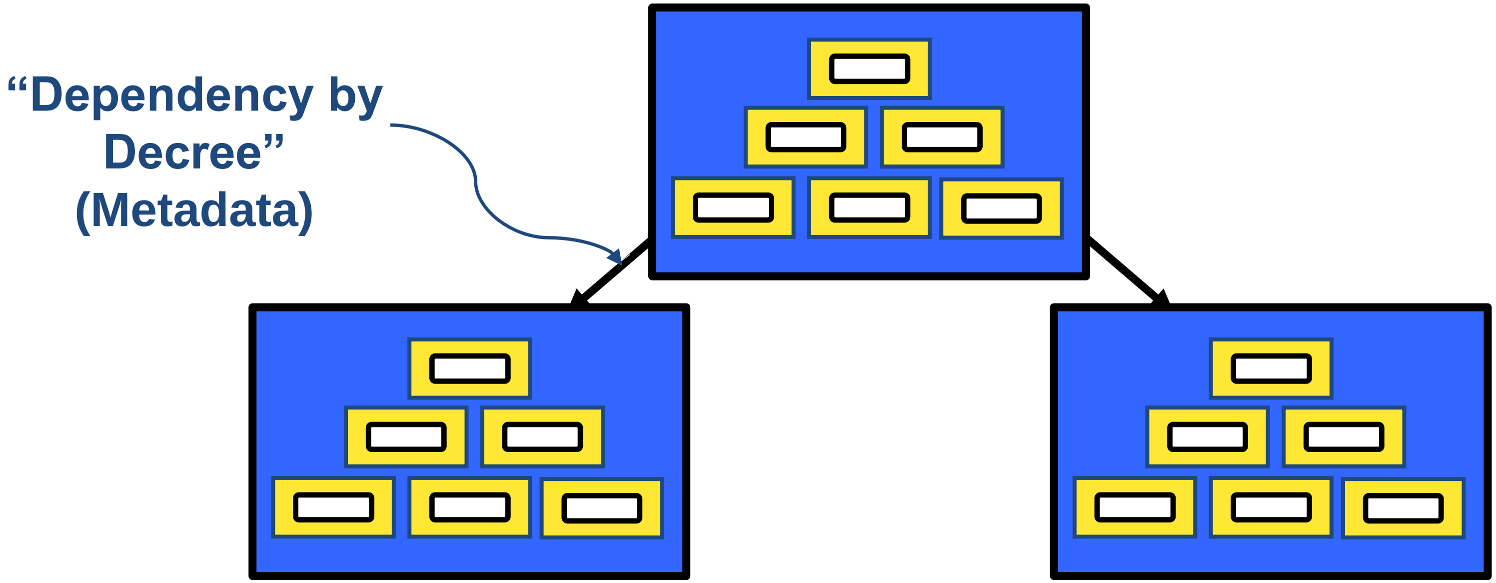
Components

Only one level of **physical aggregation**:



Packages

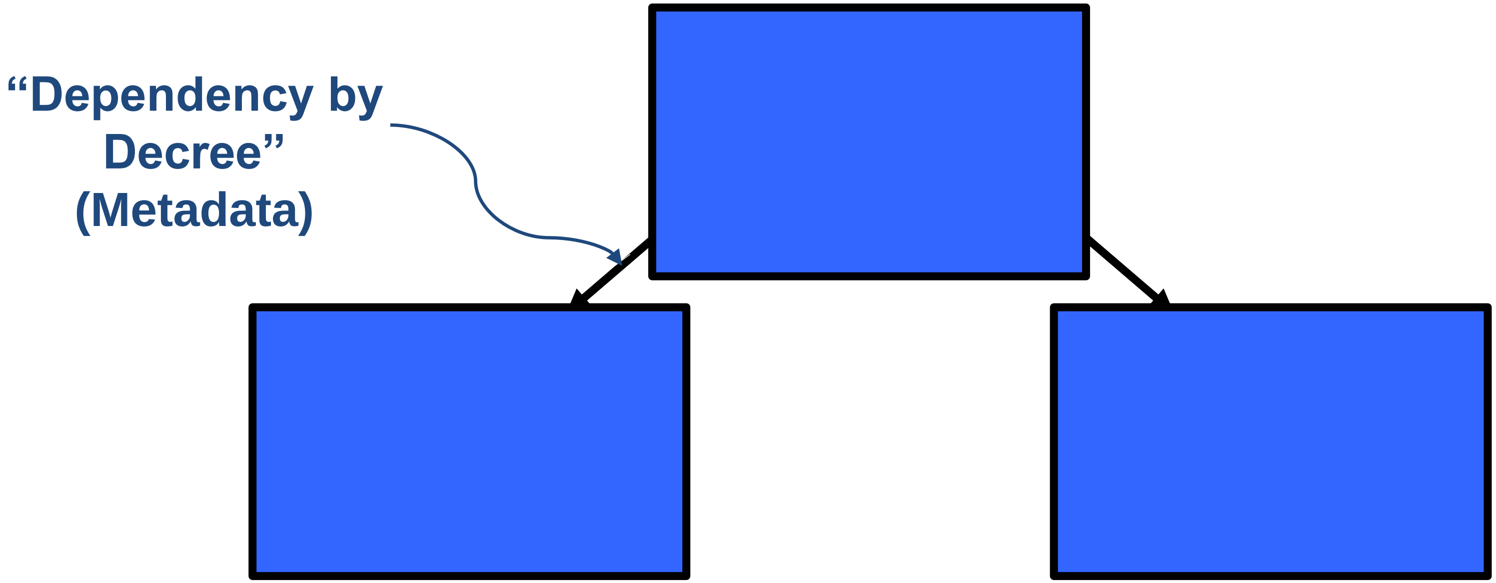
Two levels of physical aggregation:



“A Hierarchy of Component Hierarchies”

Packages

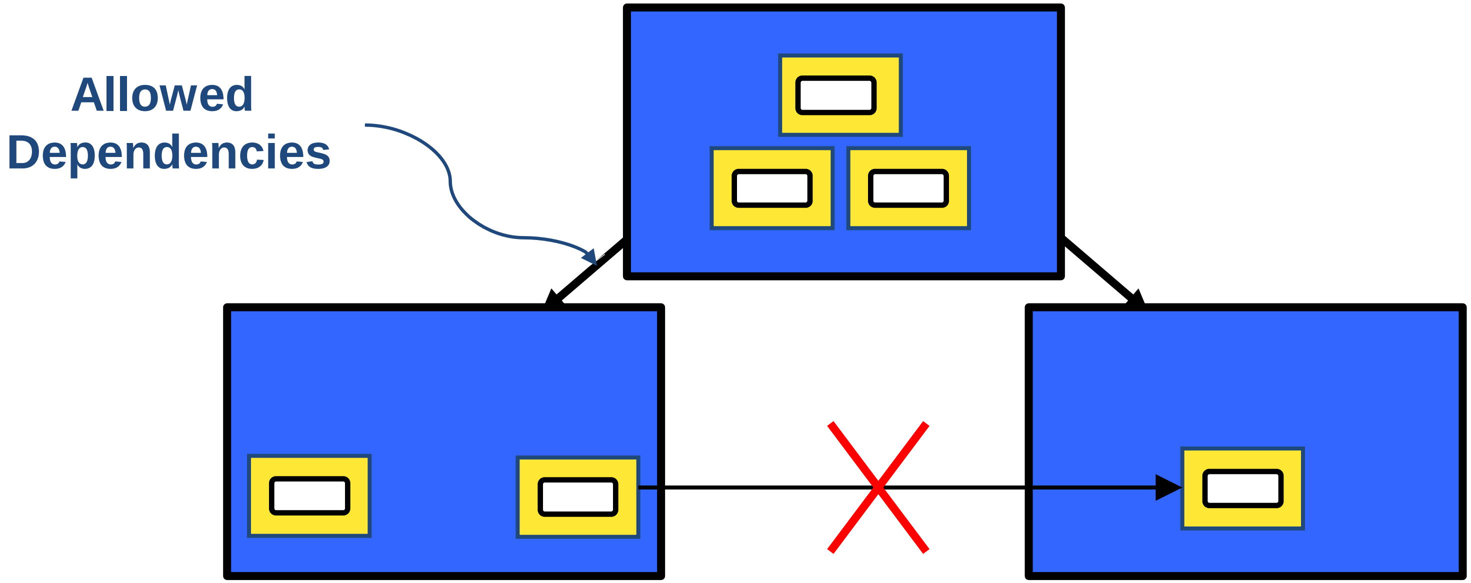
Two levels of physical aggregation:



Metadata governs, even absent of any components!

Packages

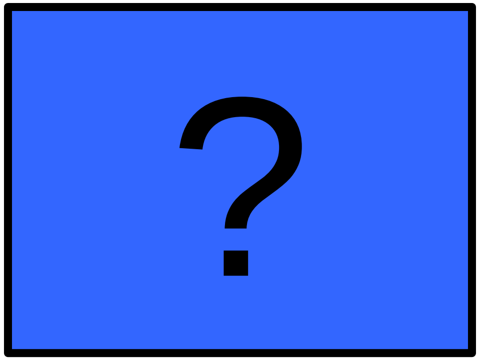
Two levels of physical aggregation:



Metadata governs allowed dependencies.

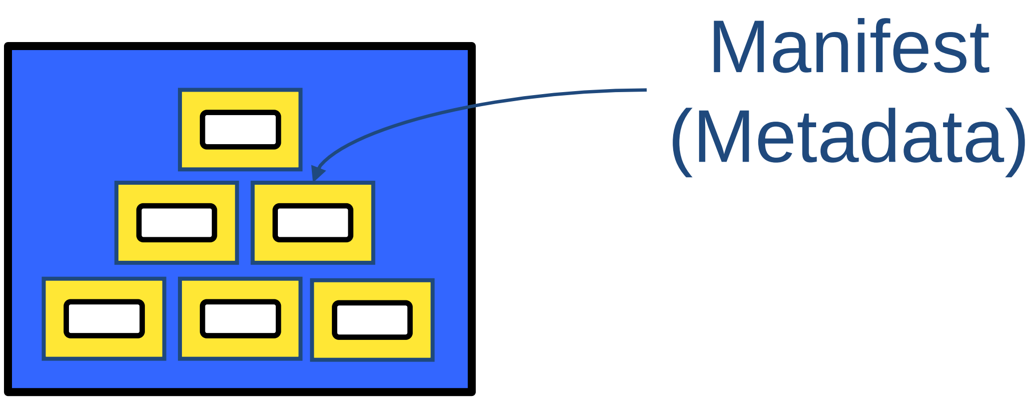
Packages

Properties of an aggregate:



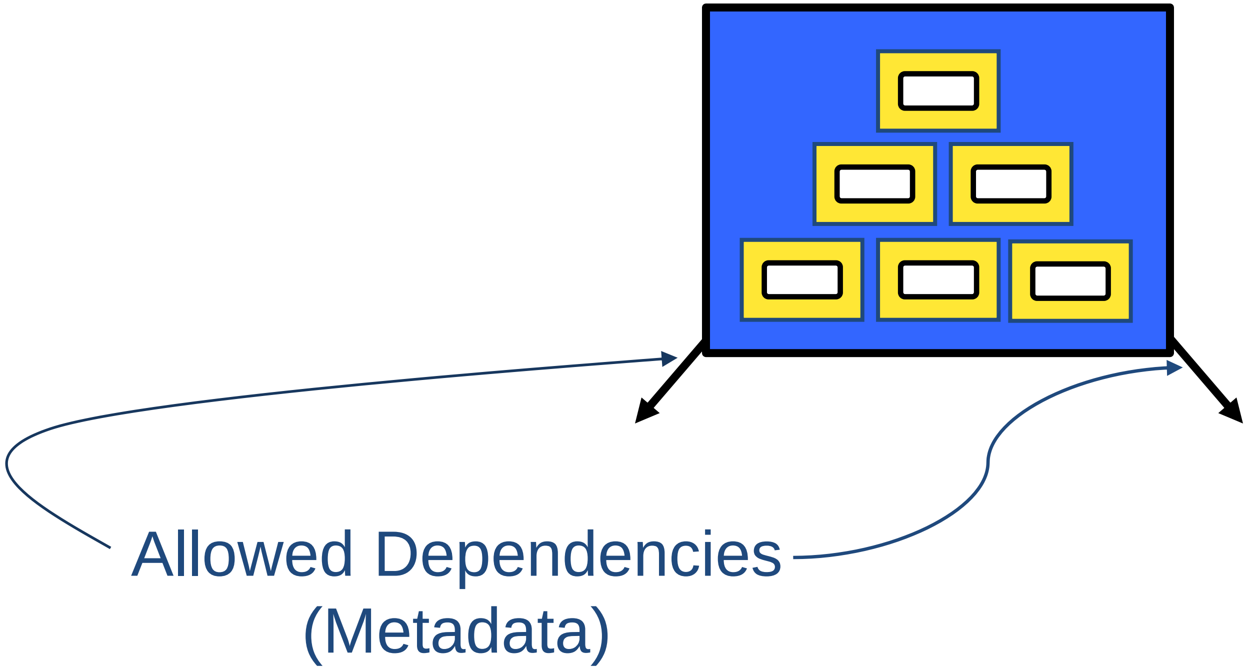
Packages

Properties of an aggregate:



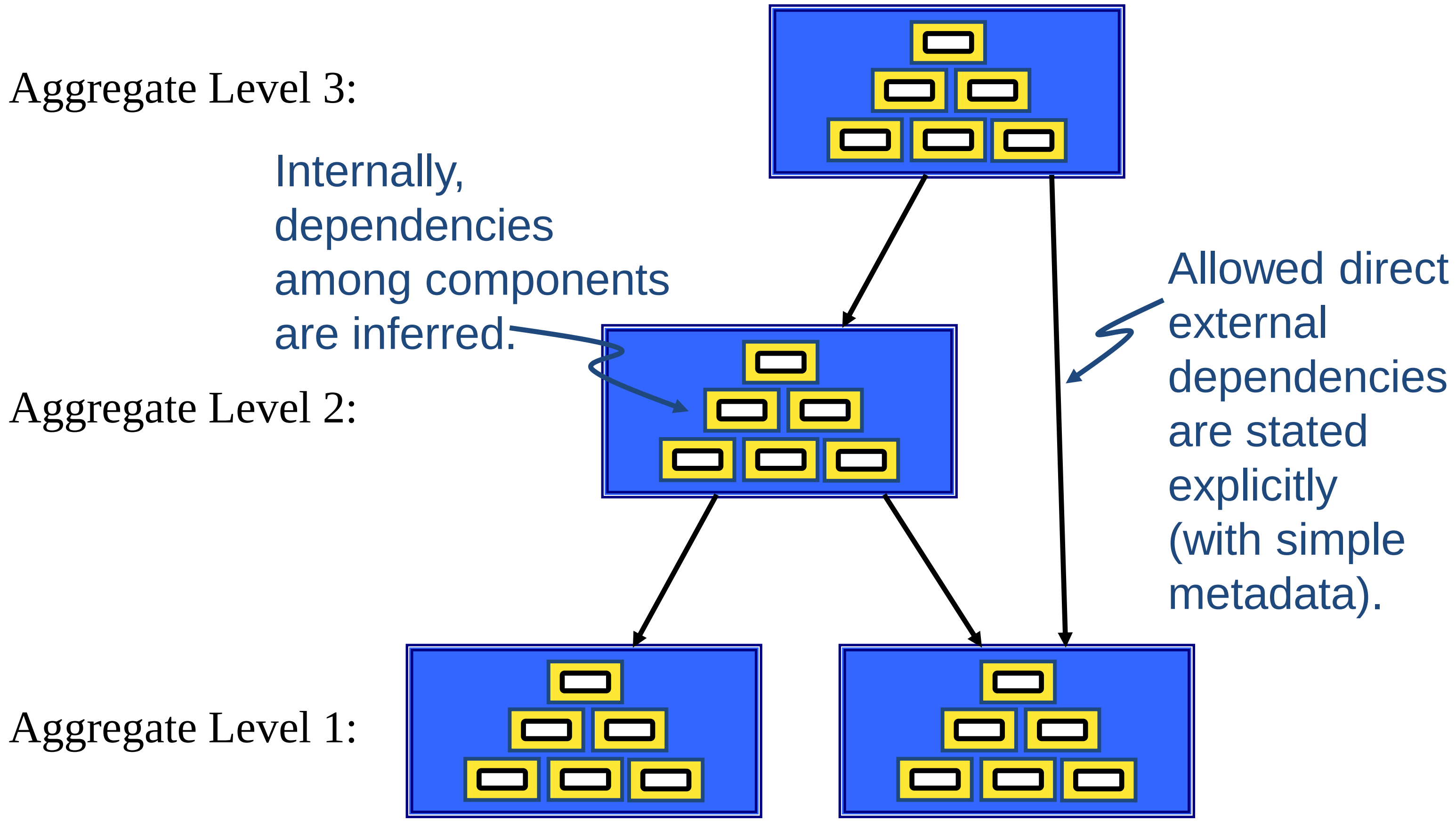
Packages

Properties of an aggregate:



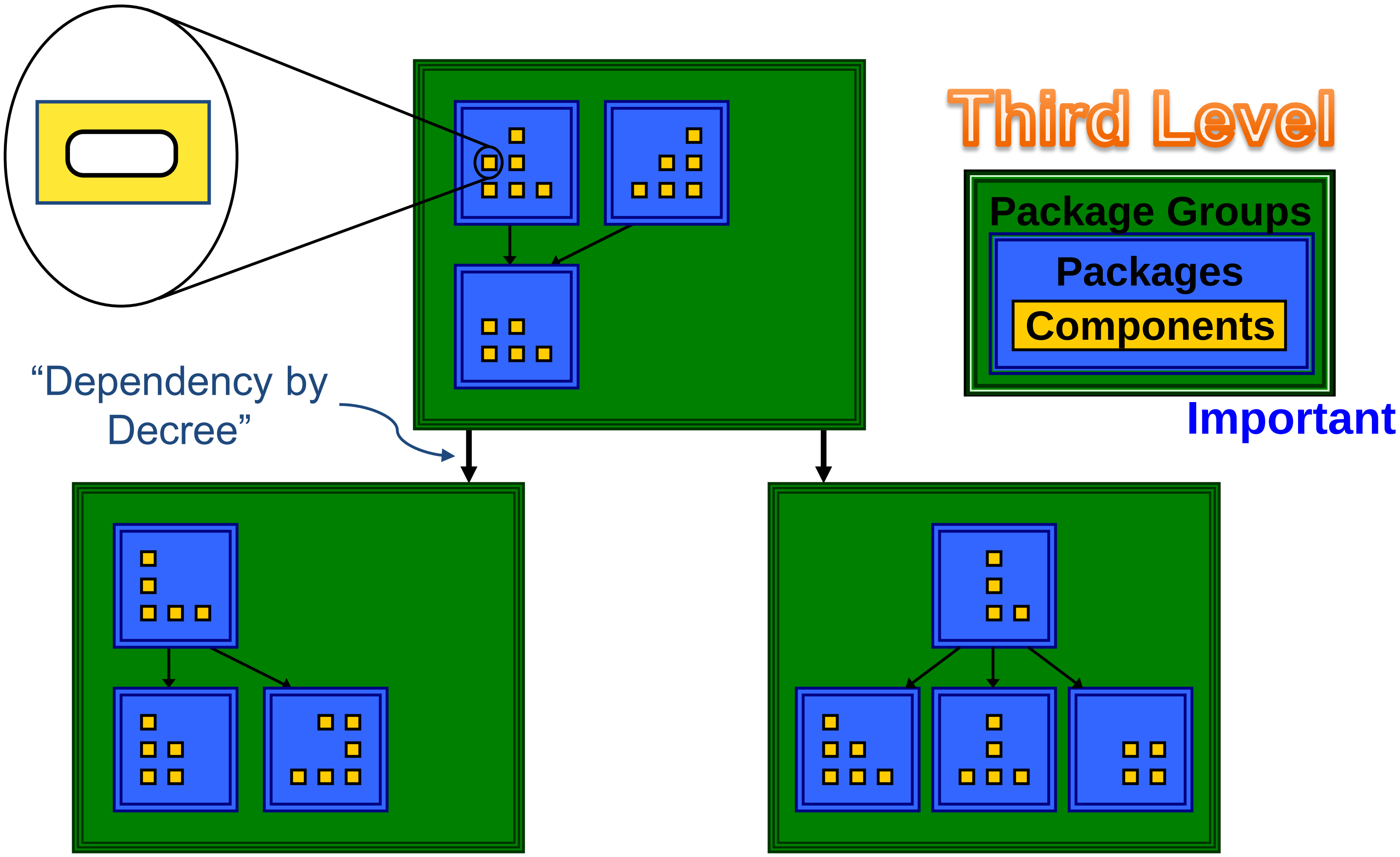
Packages

Aggregate dependencies:

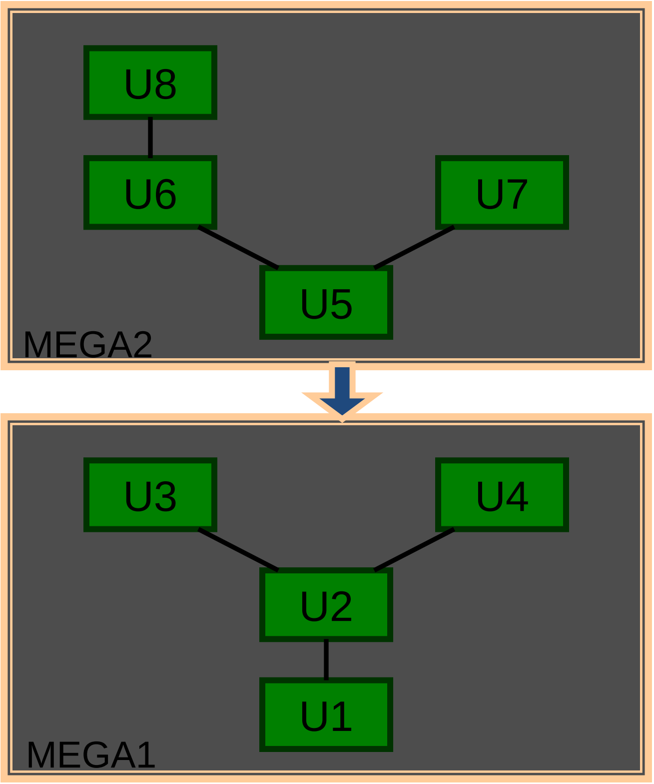


1. Process & Architecture

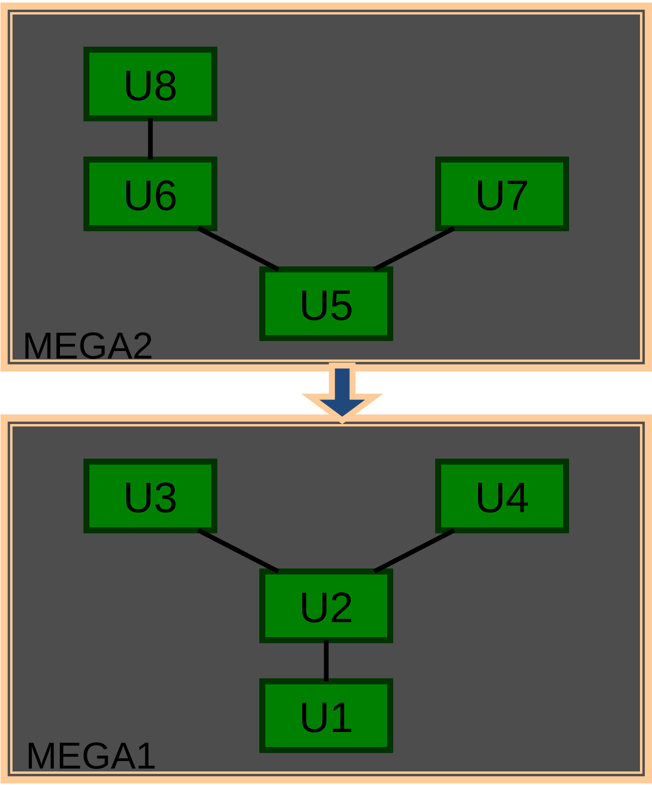
Package Groups



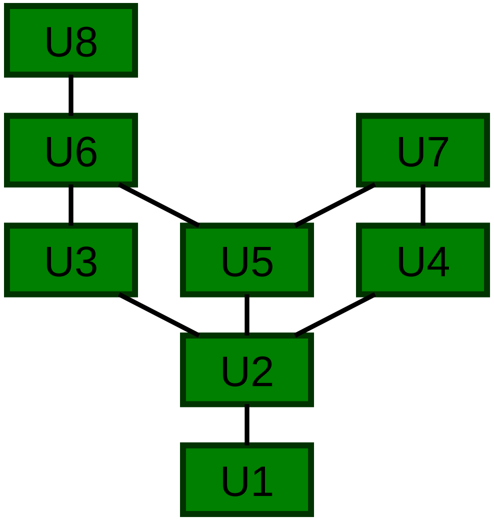
What About a Fourth-Level Aggregate?



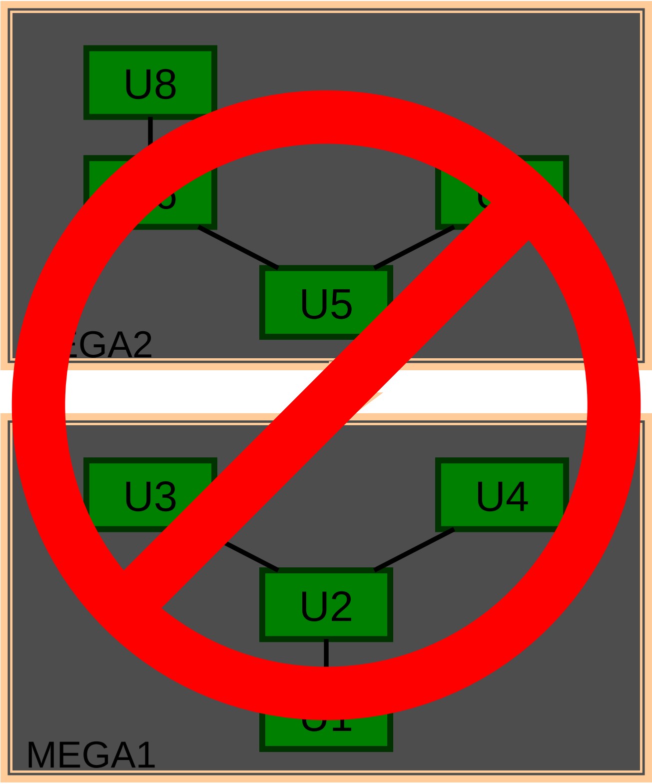
What About a Fourth-Level Aggregate?



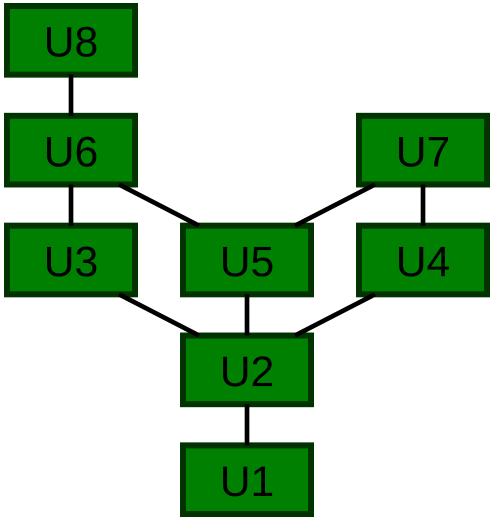
We find **two** or **three** levels of aggregation per library sufficient.



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.

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).

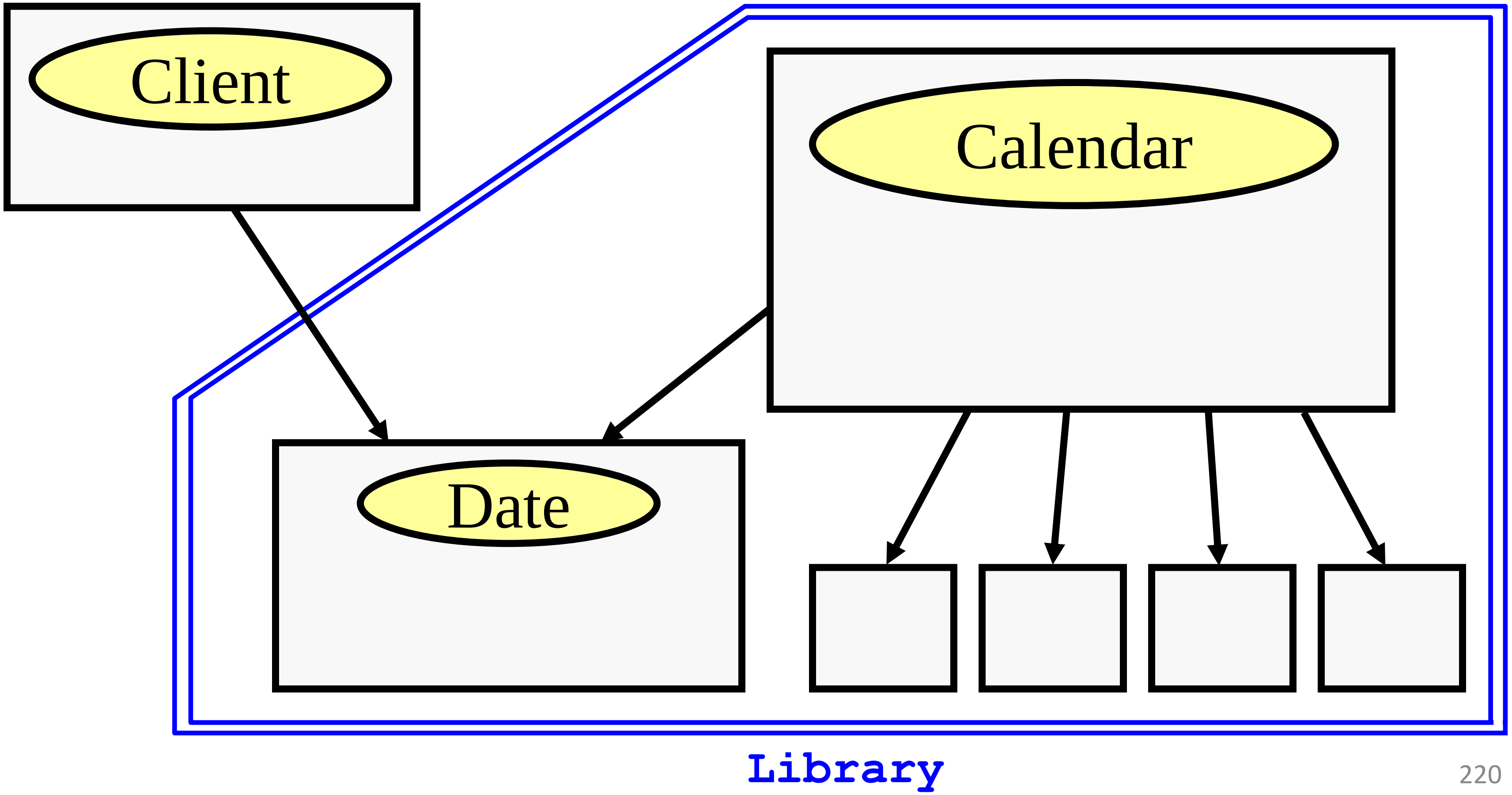
Logical/Physical Synergy

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).

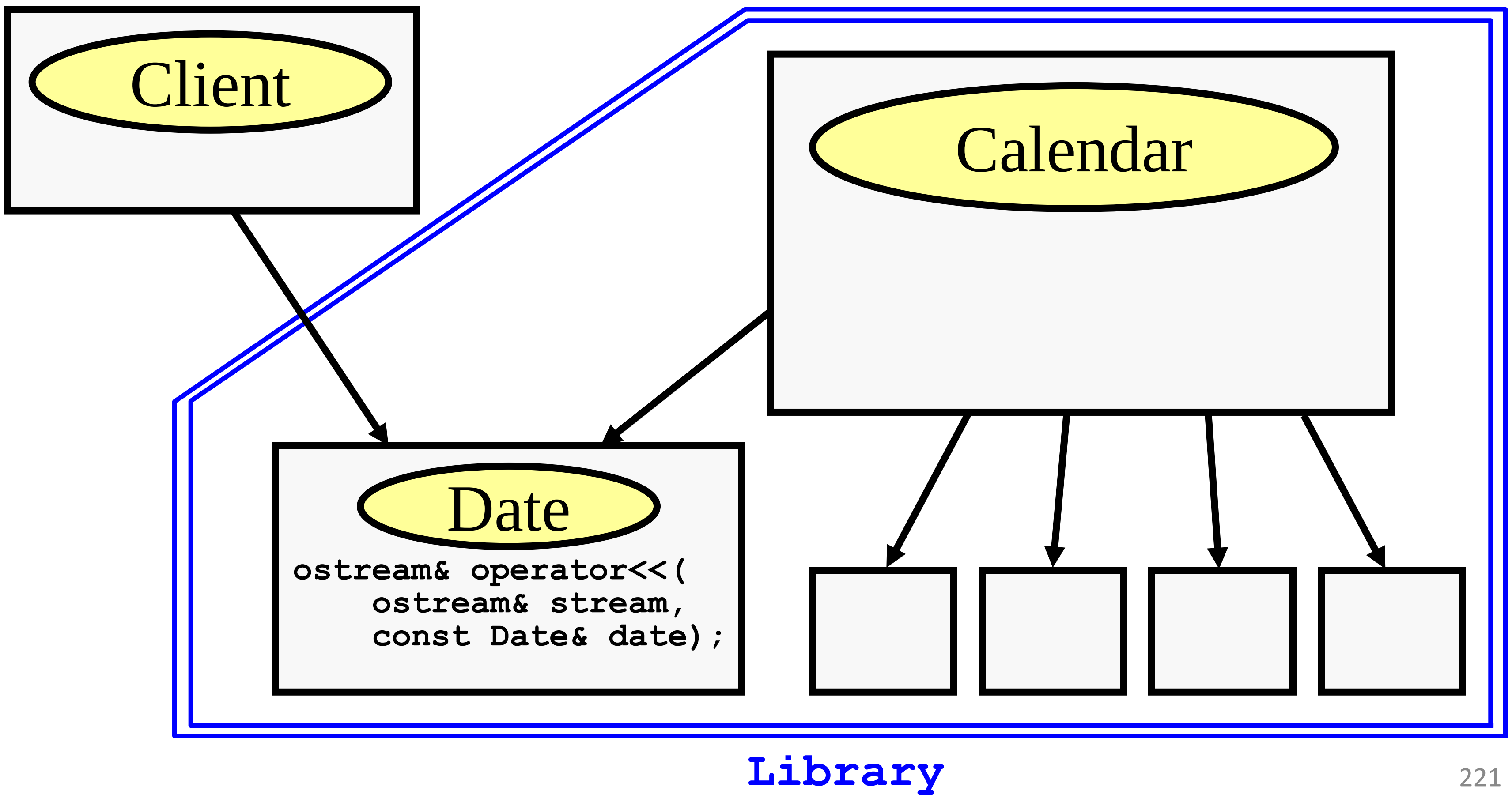
Logical/Physical <u>**In**</u>coherence

A Component Defines Only What It Declares.



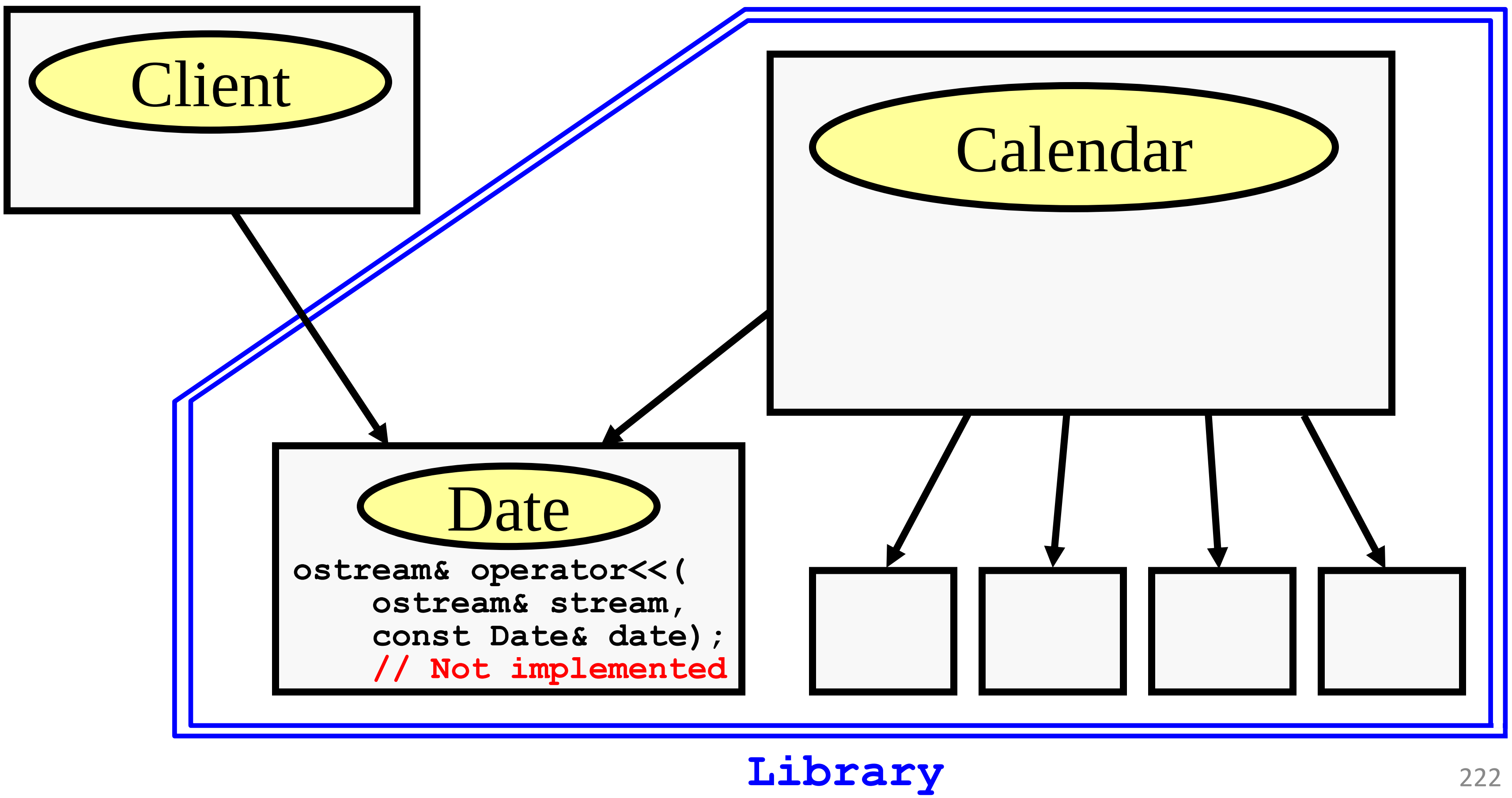
Logical/Physical **Incoherence**

A Component Defines Only What It Declares.



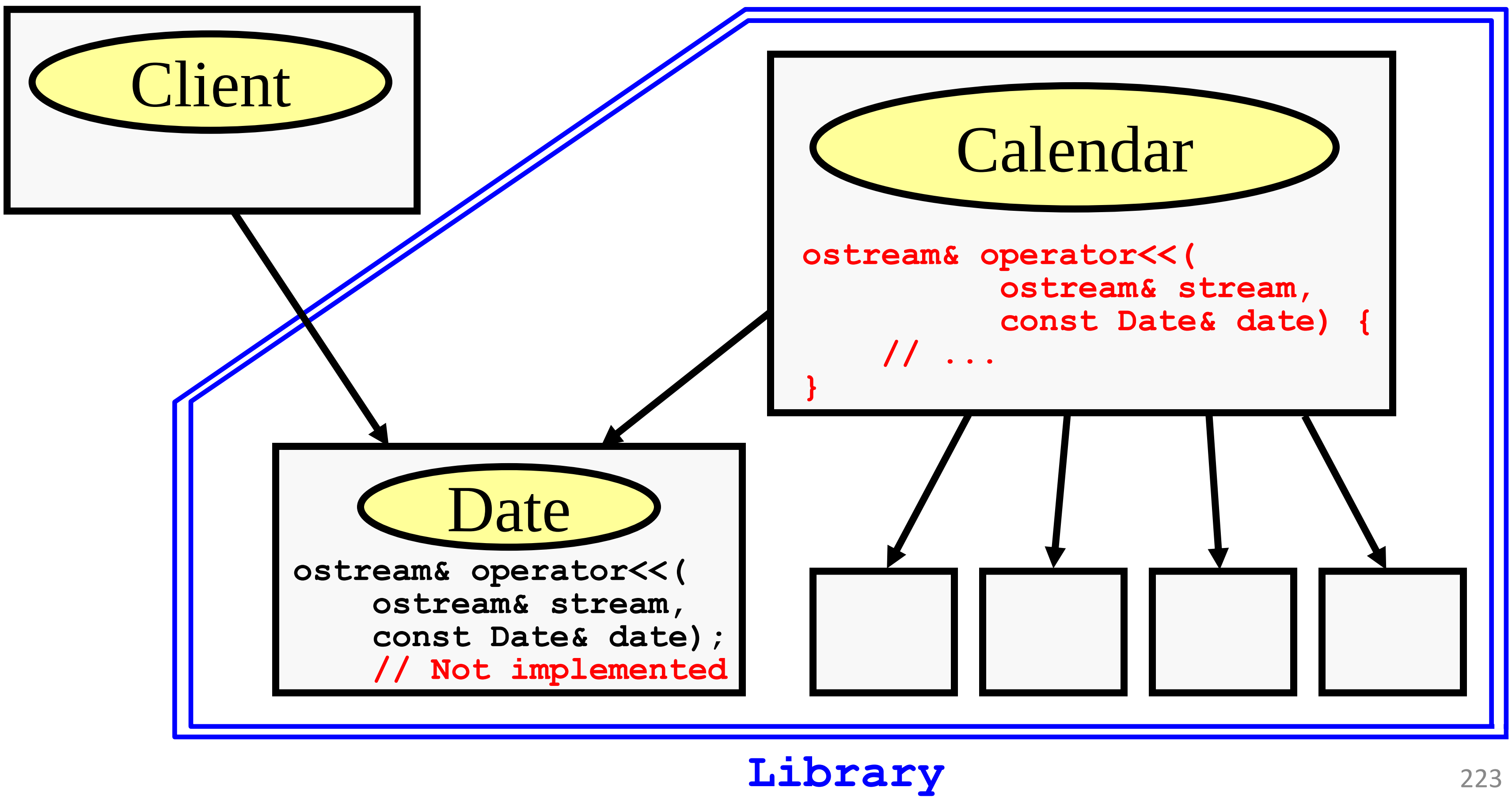
Logical/Physical **Incoherence**

A Component Defines Only What It Declares.



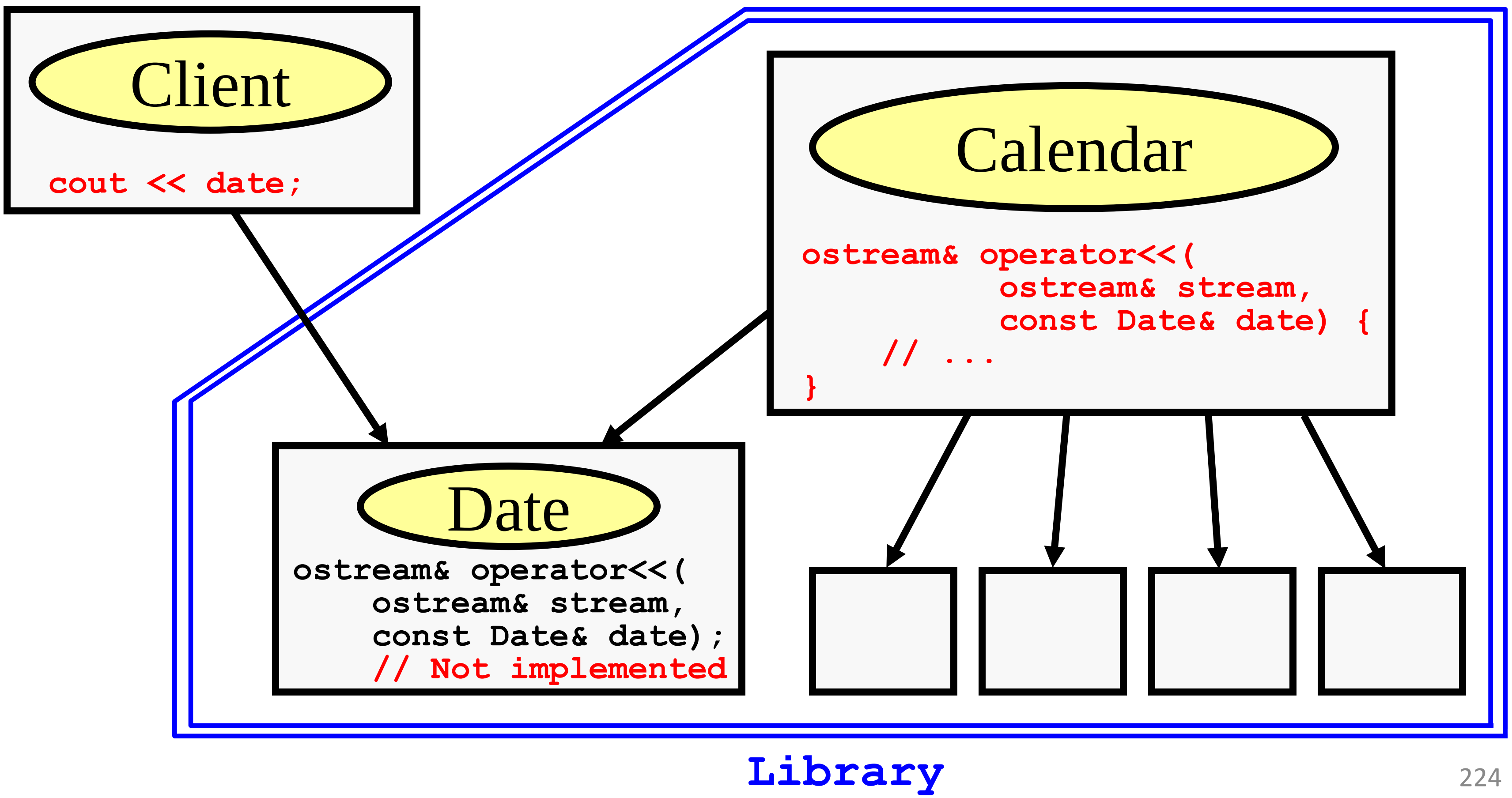
Logical/Physical **Incoherence**

A Component Defines Only What It Declares.



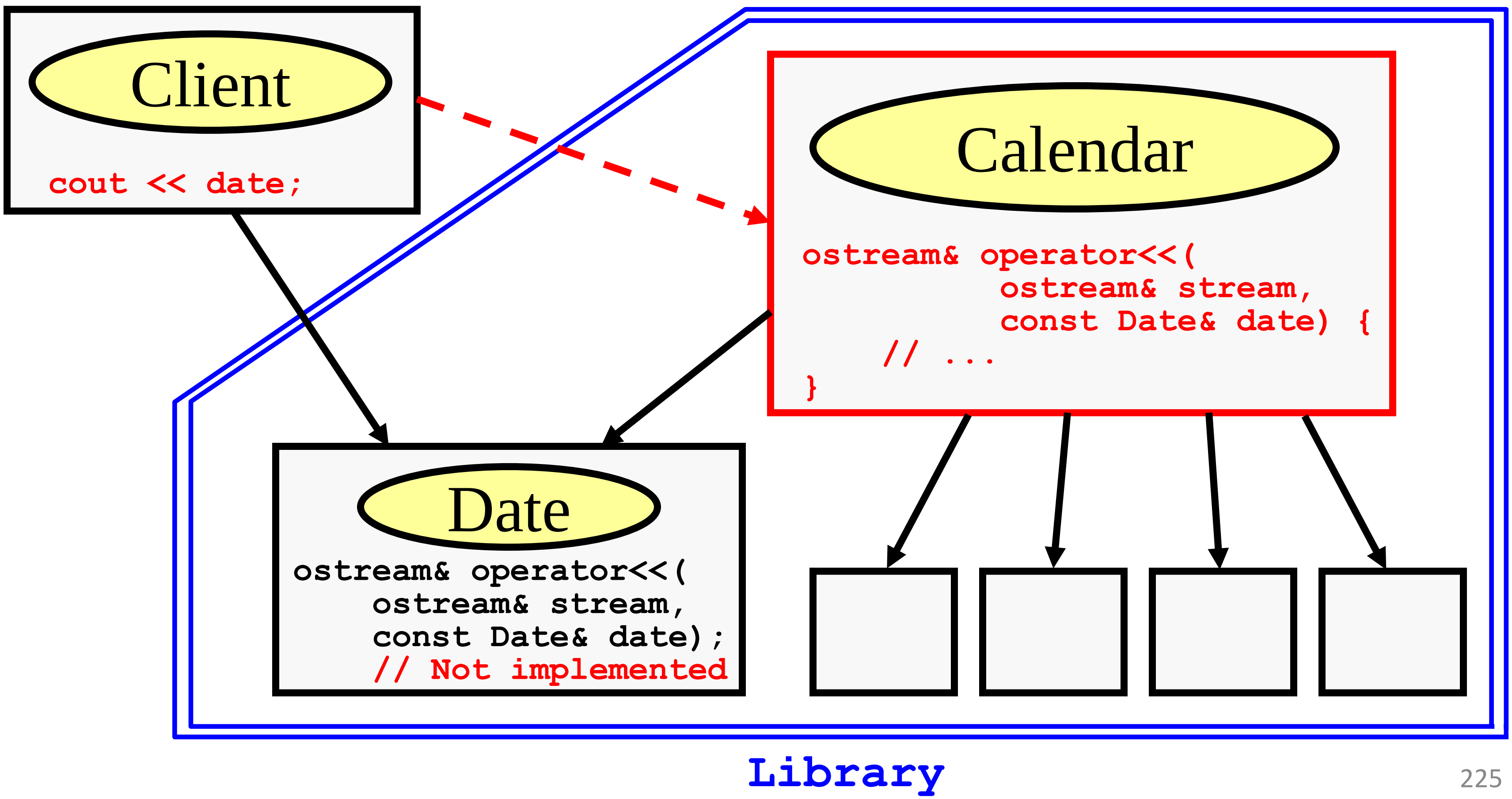
Logical/Physical **Incoherence**

A Component Defines Only What It Declares.



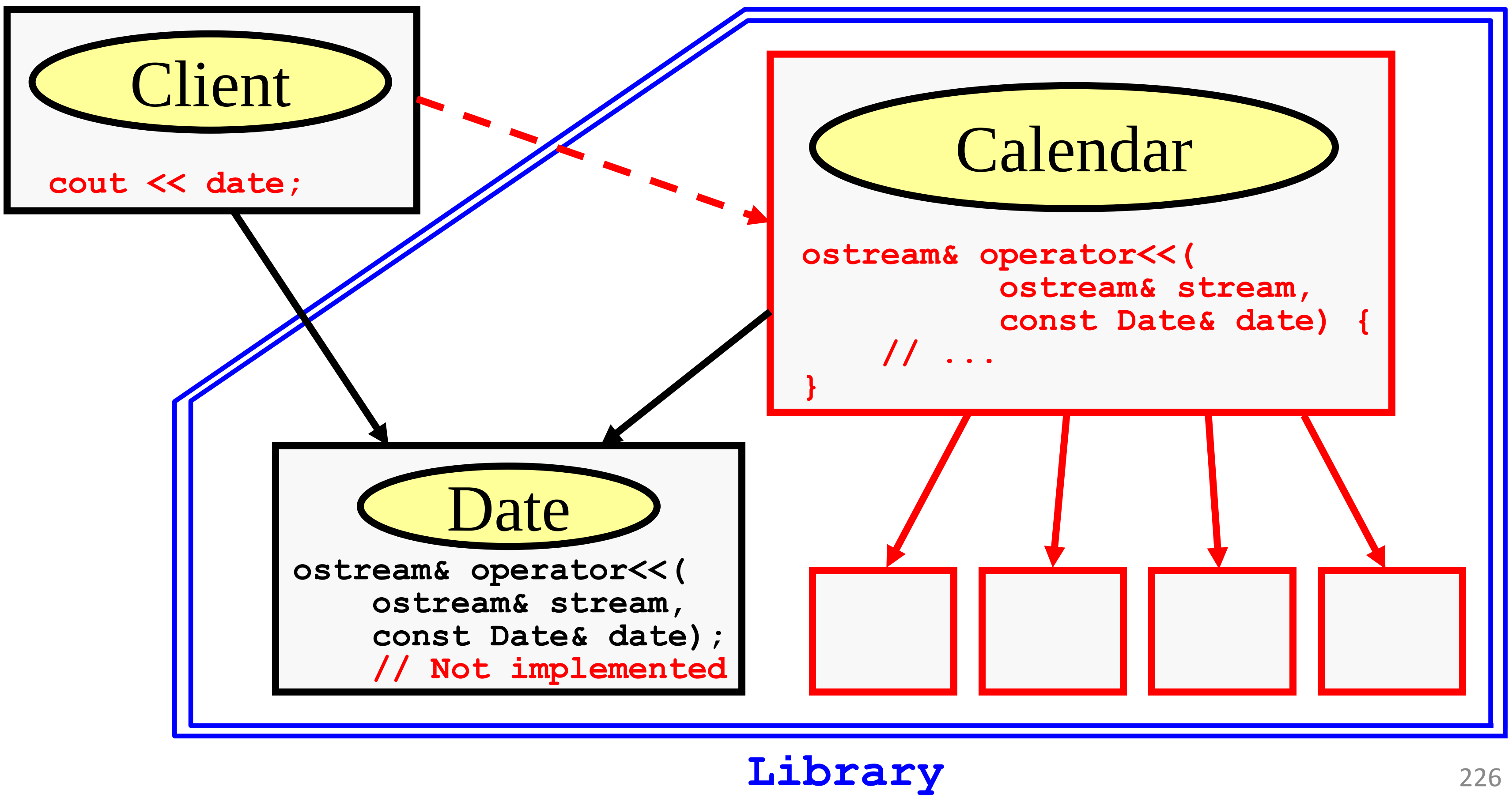
Logical/Physical Incoherence

A Component Defines Only What It Declares.



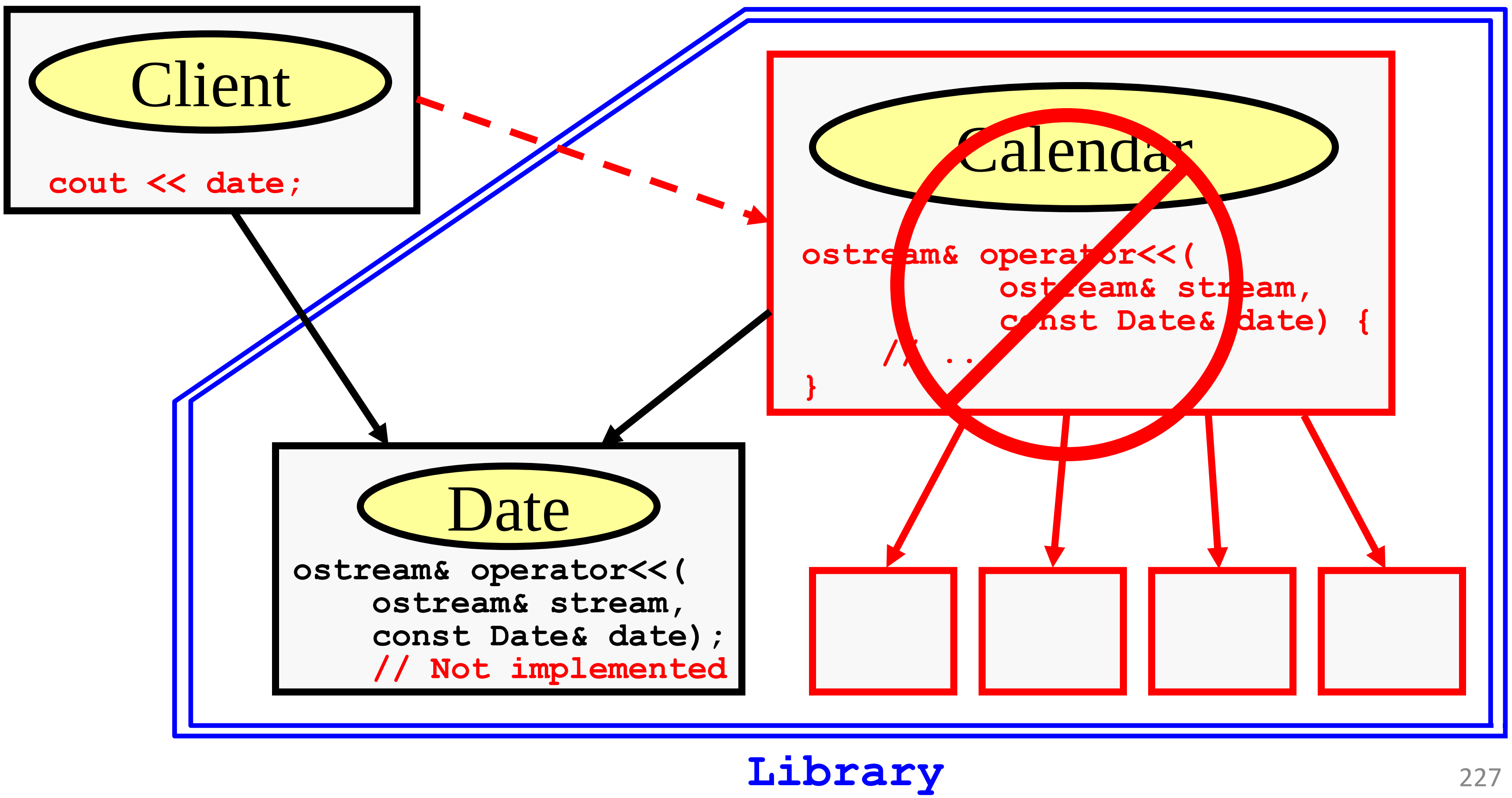
Logical/Physical <u>**In**</u>coherence

A Component Defines Only What It Declares.



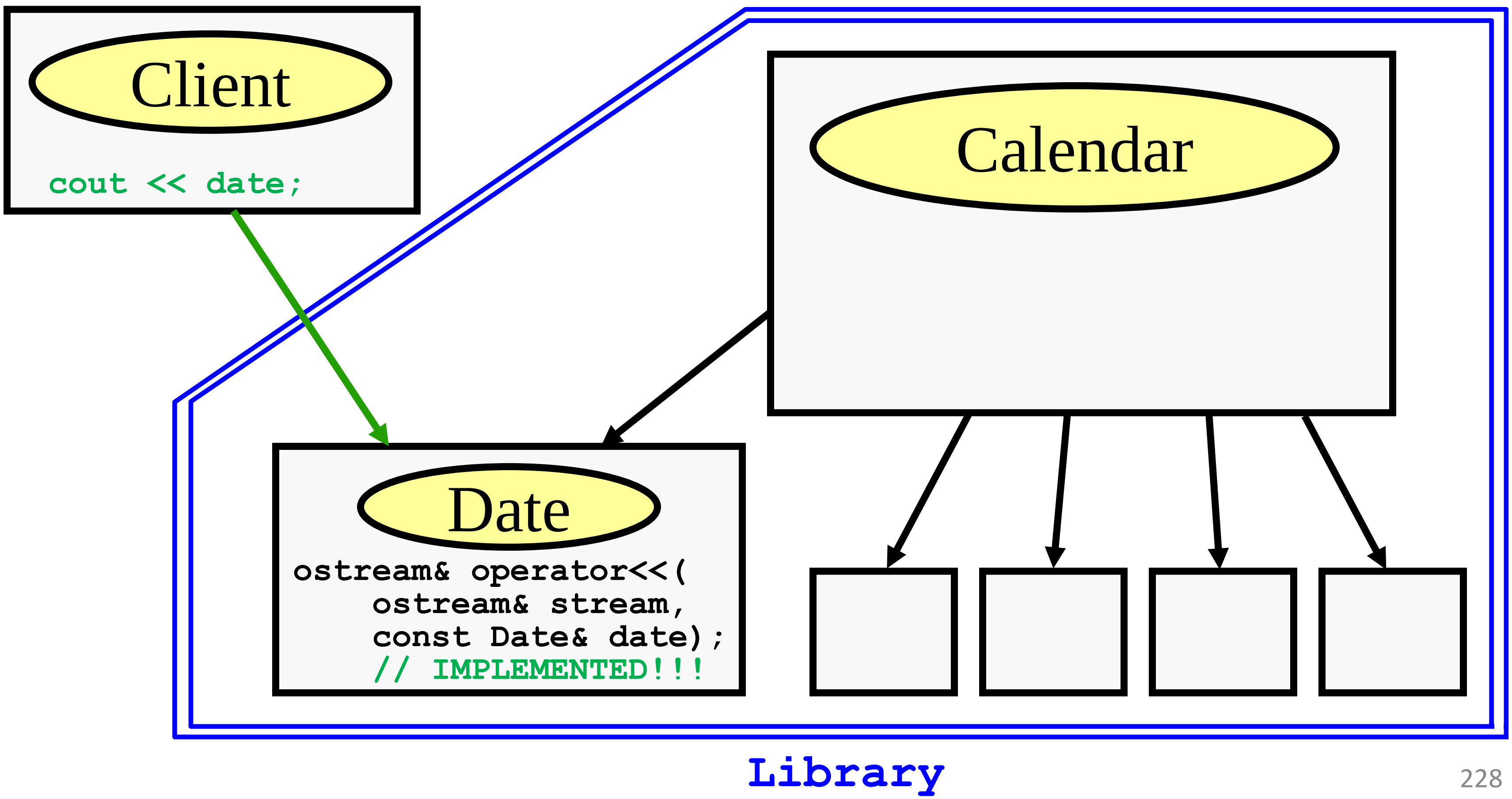
Logical/Physical **<u>In</u>**coherence

A Component Defines Only What It Declares.

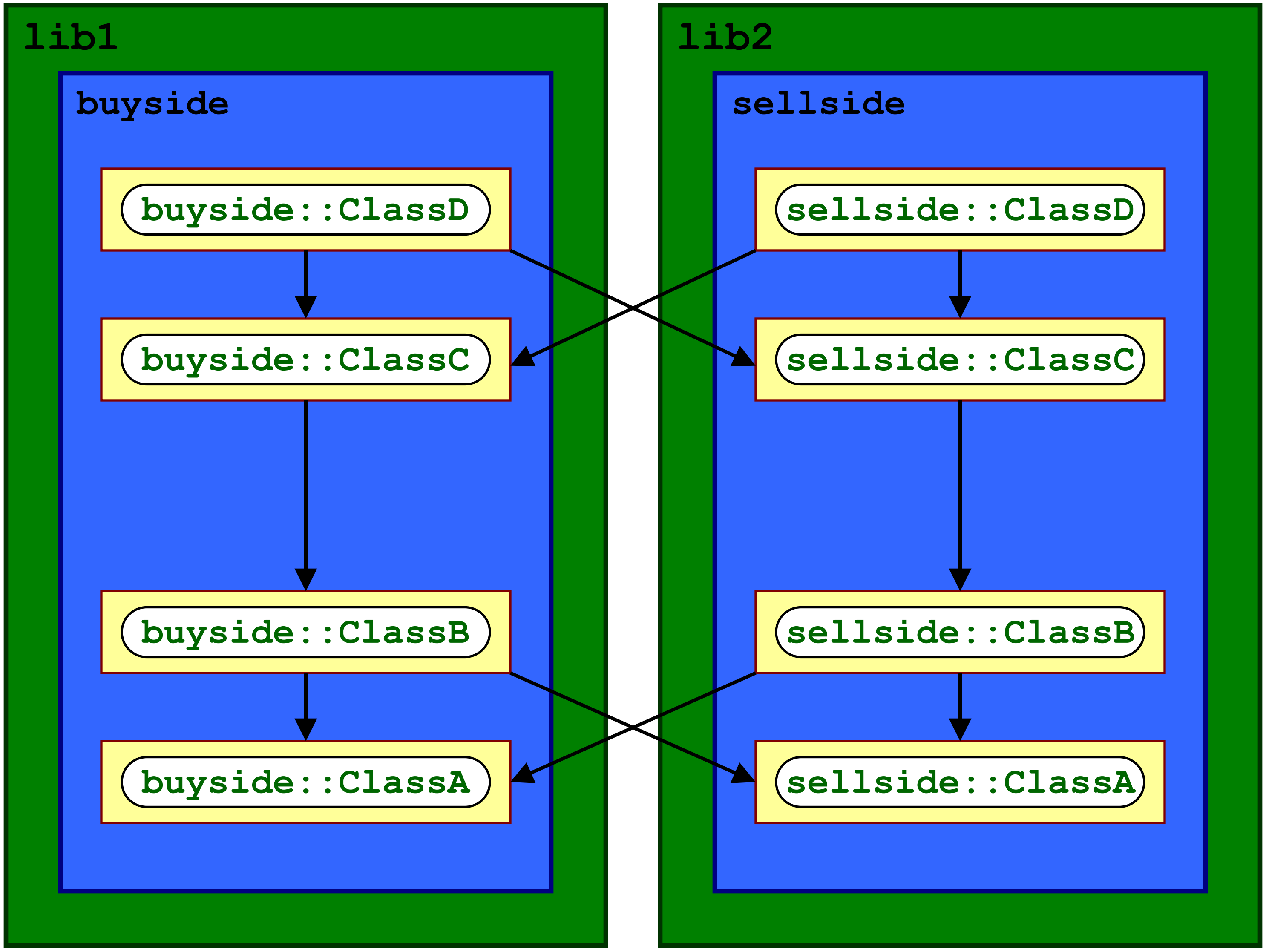


Logical/Physical **Incoherence**

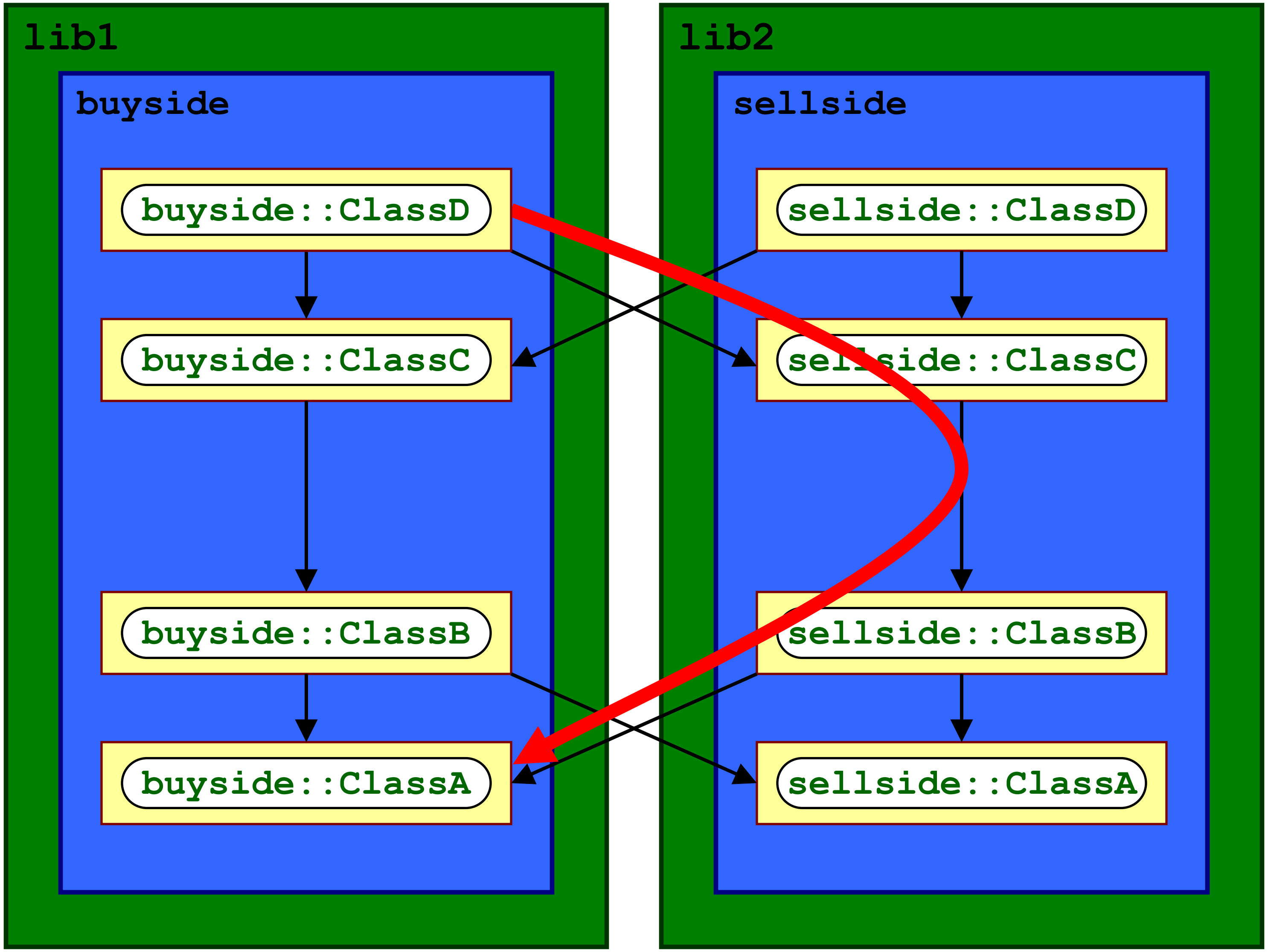
A Component Defines Only What It Declares.



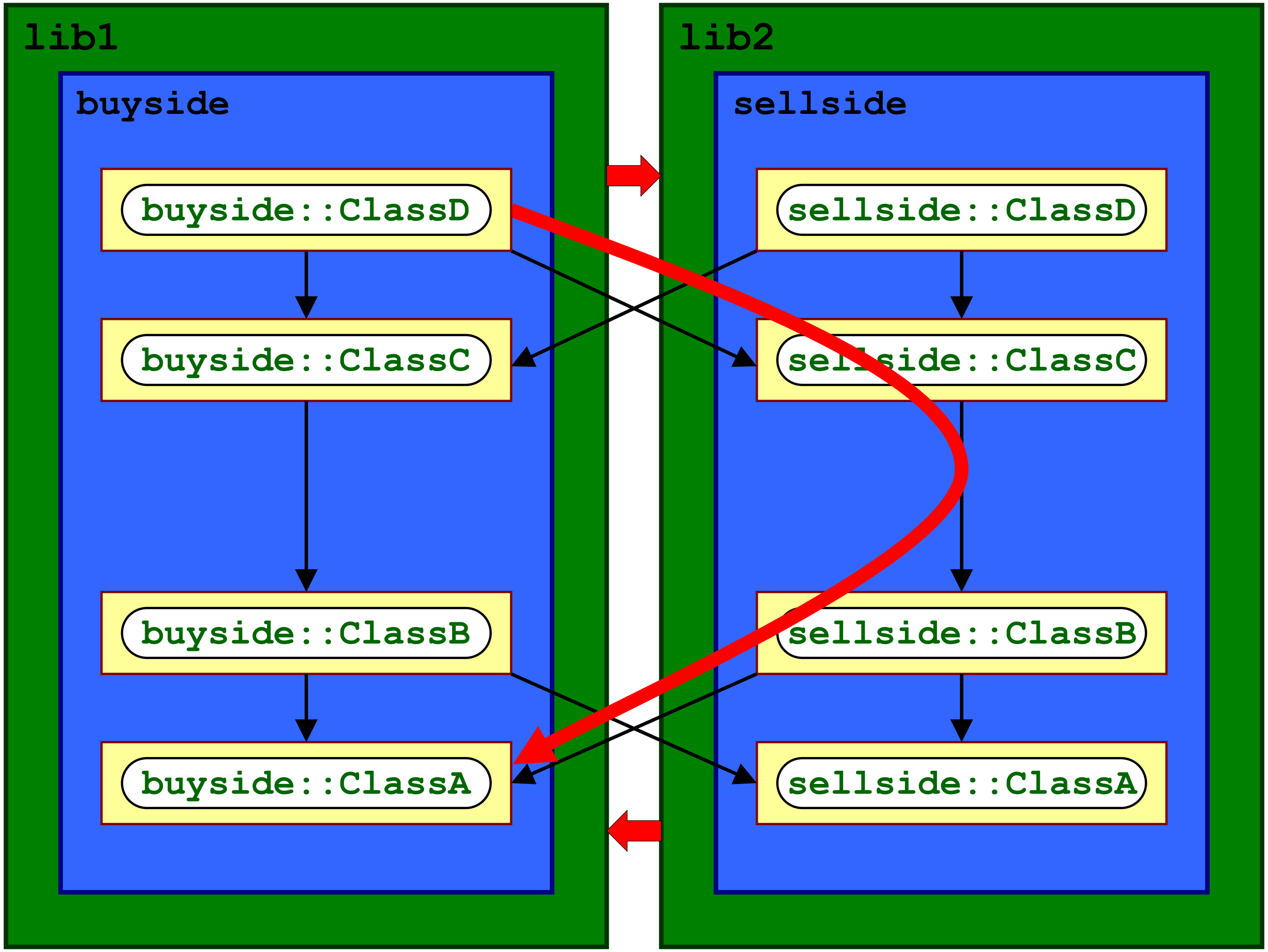
Logical/Physical Coherence



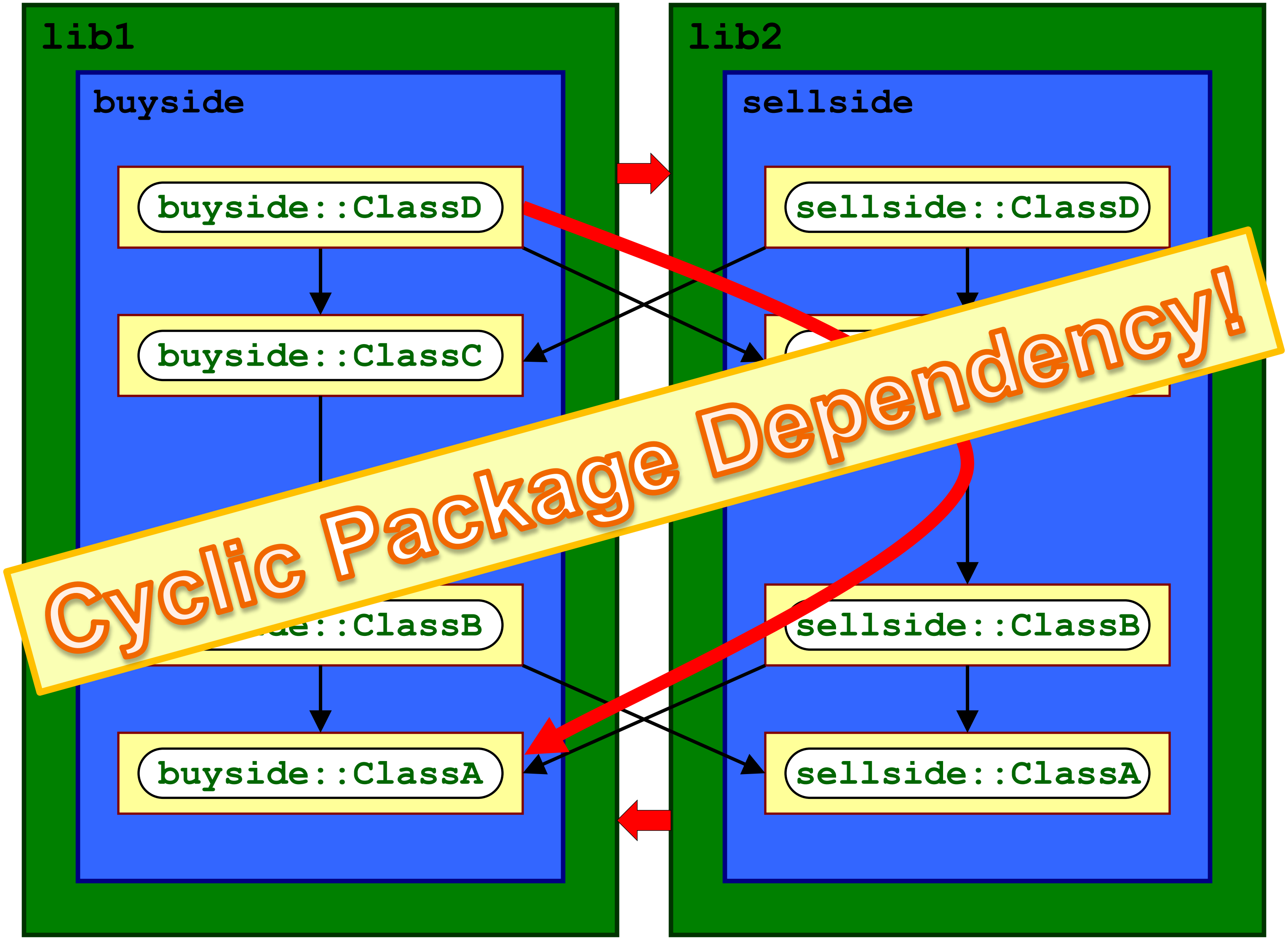
Logical/Physical Coherence



Logical/Physical Coherence

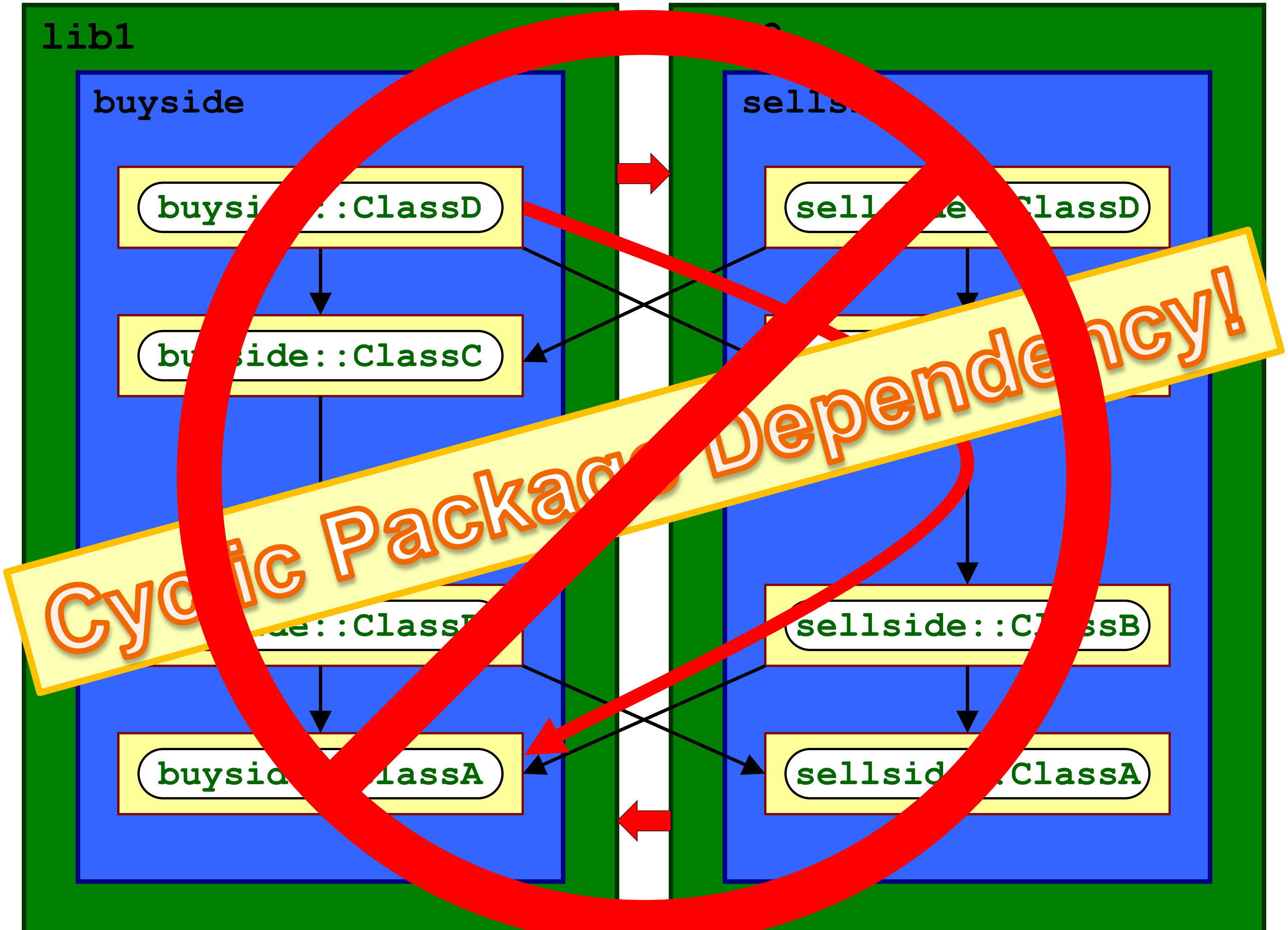


Logical/Physical Coherence

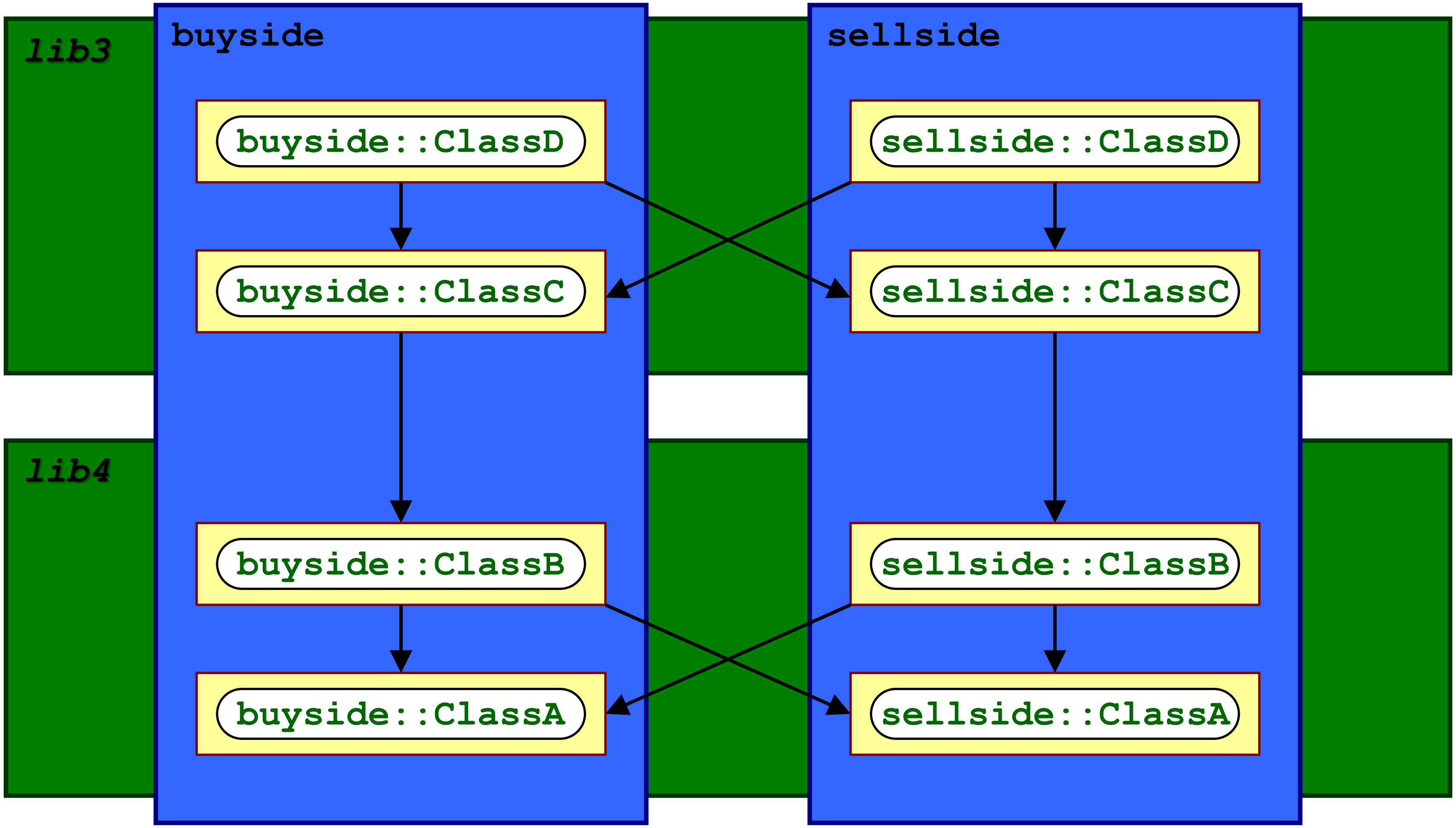


1. Process & Architecture

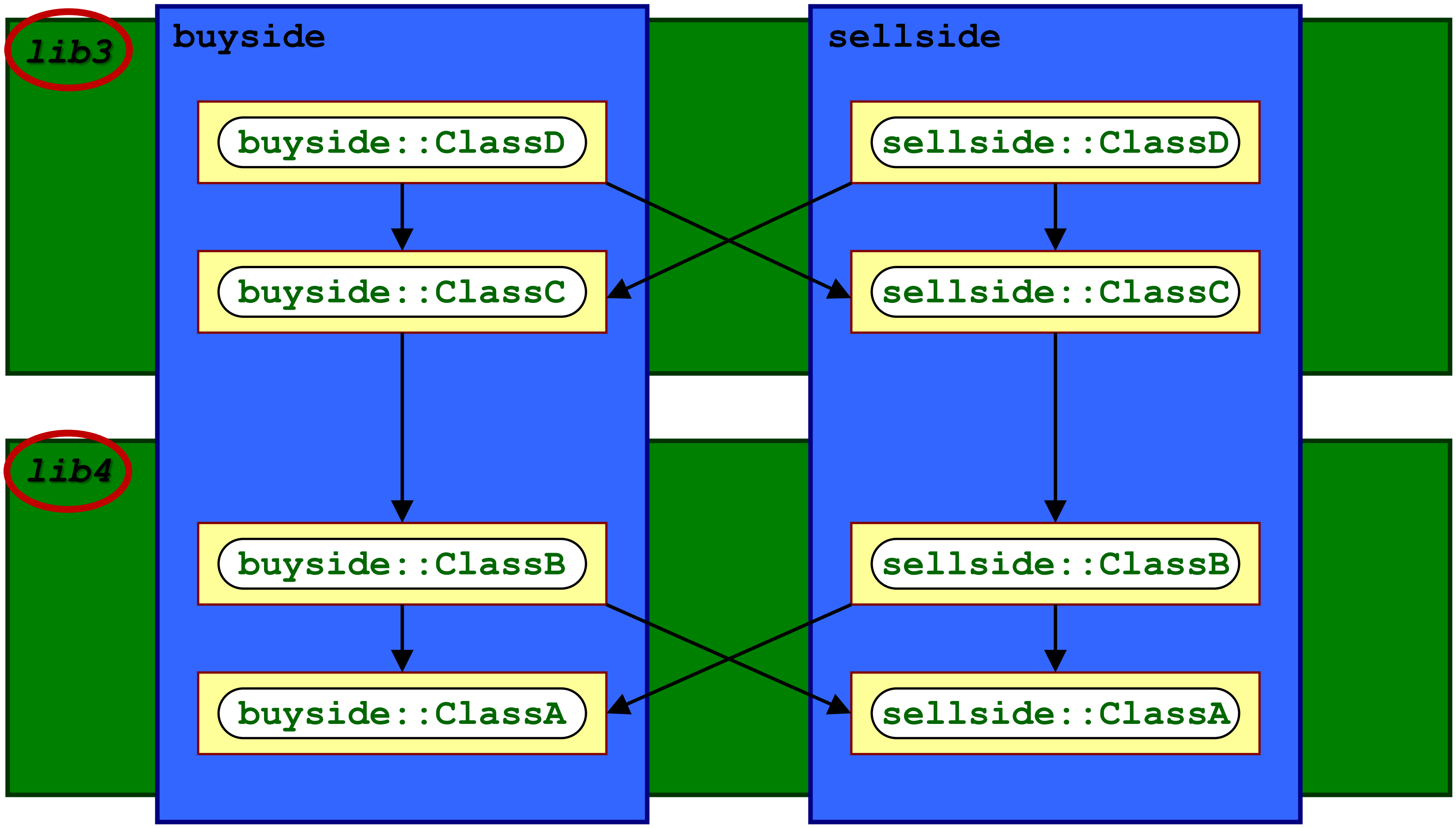
Logical/Physical Coherence



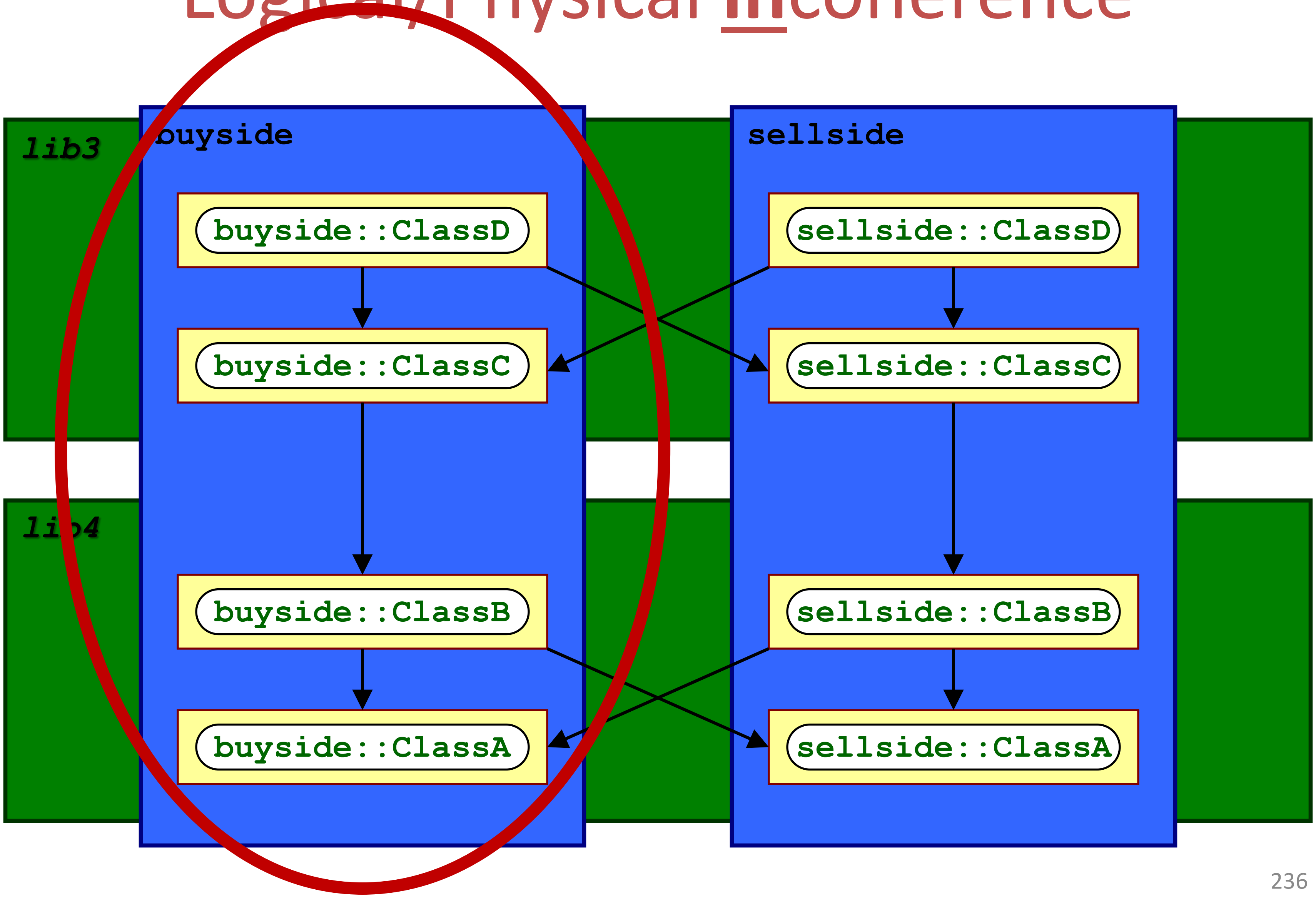
Logical/Physical **Incoherence**



Logical/Physical **Incoherence**

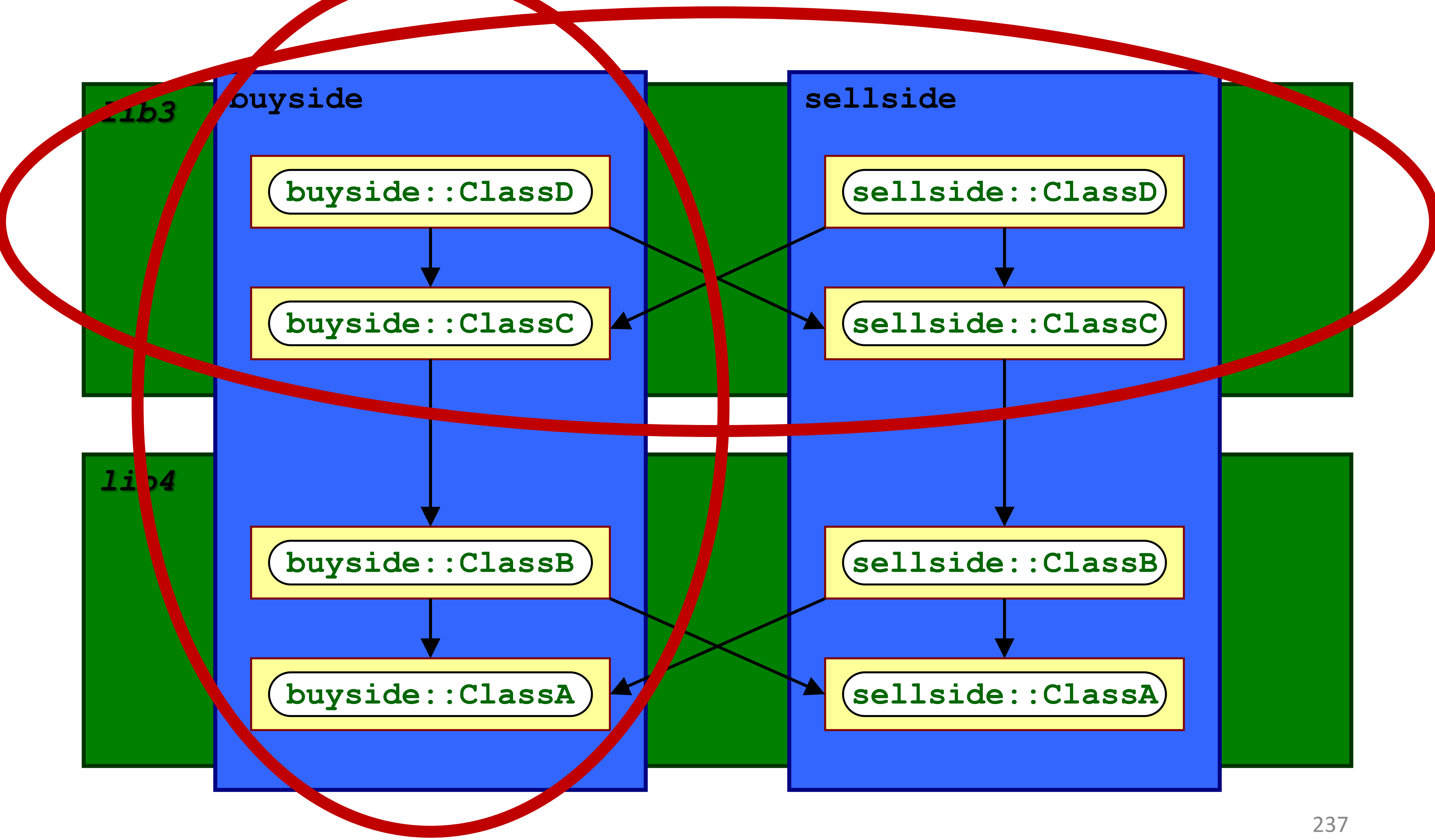


Logical/Physical **Incoherence**



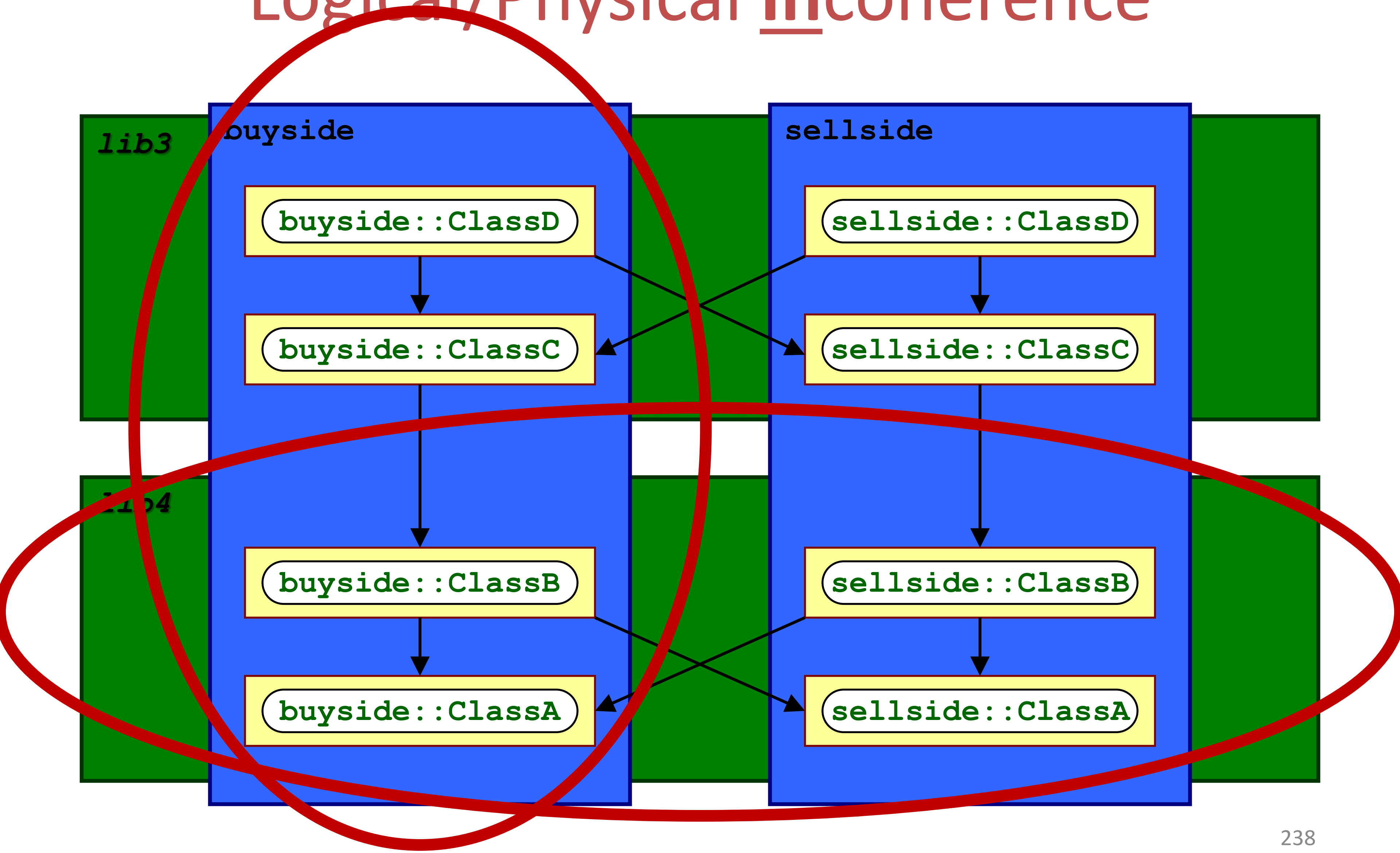
1. Process & Architecture

Logical/Physical **Incoherence**

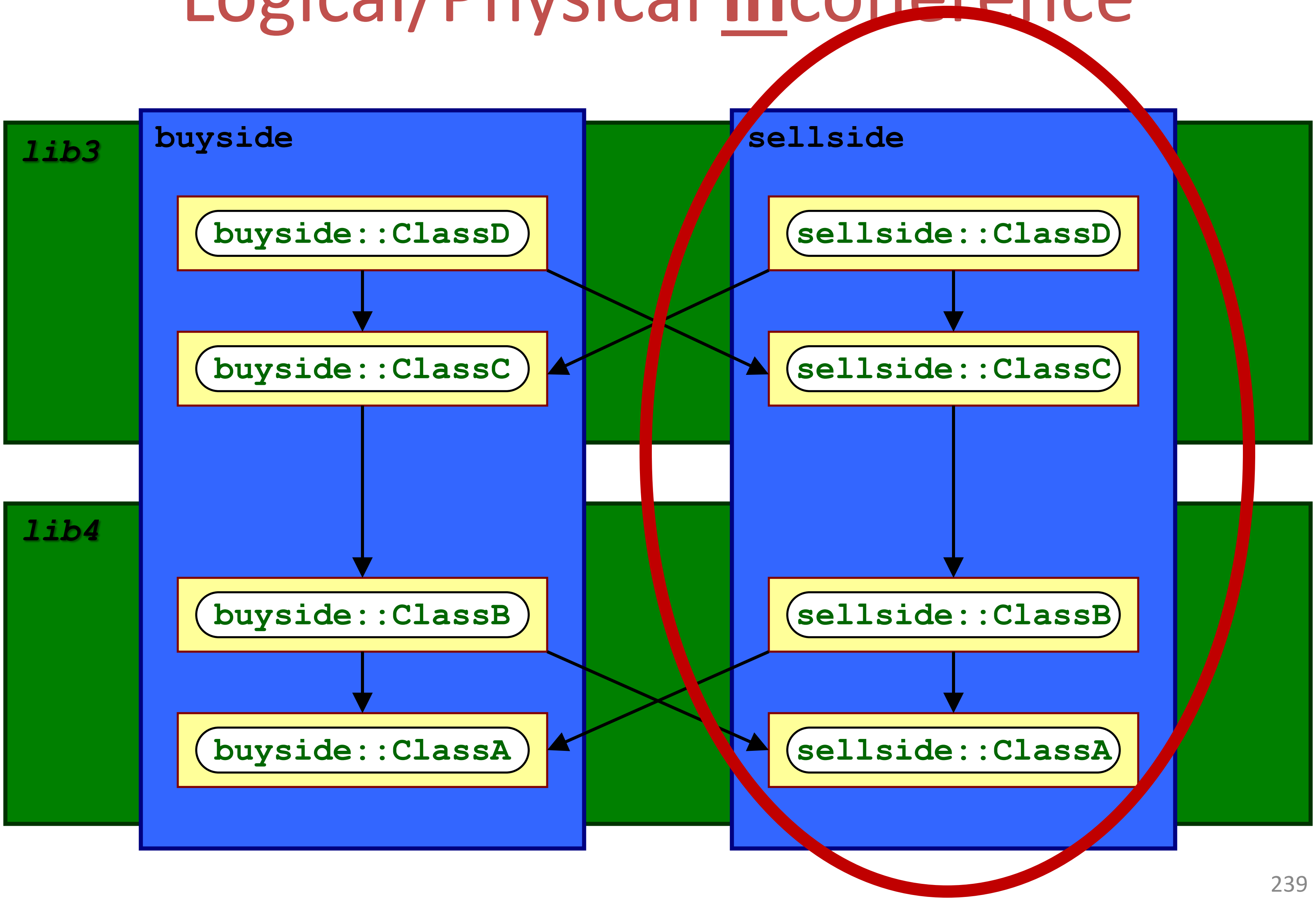


1. Process & Architecture

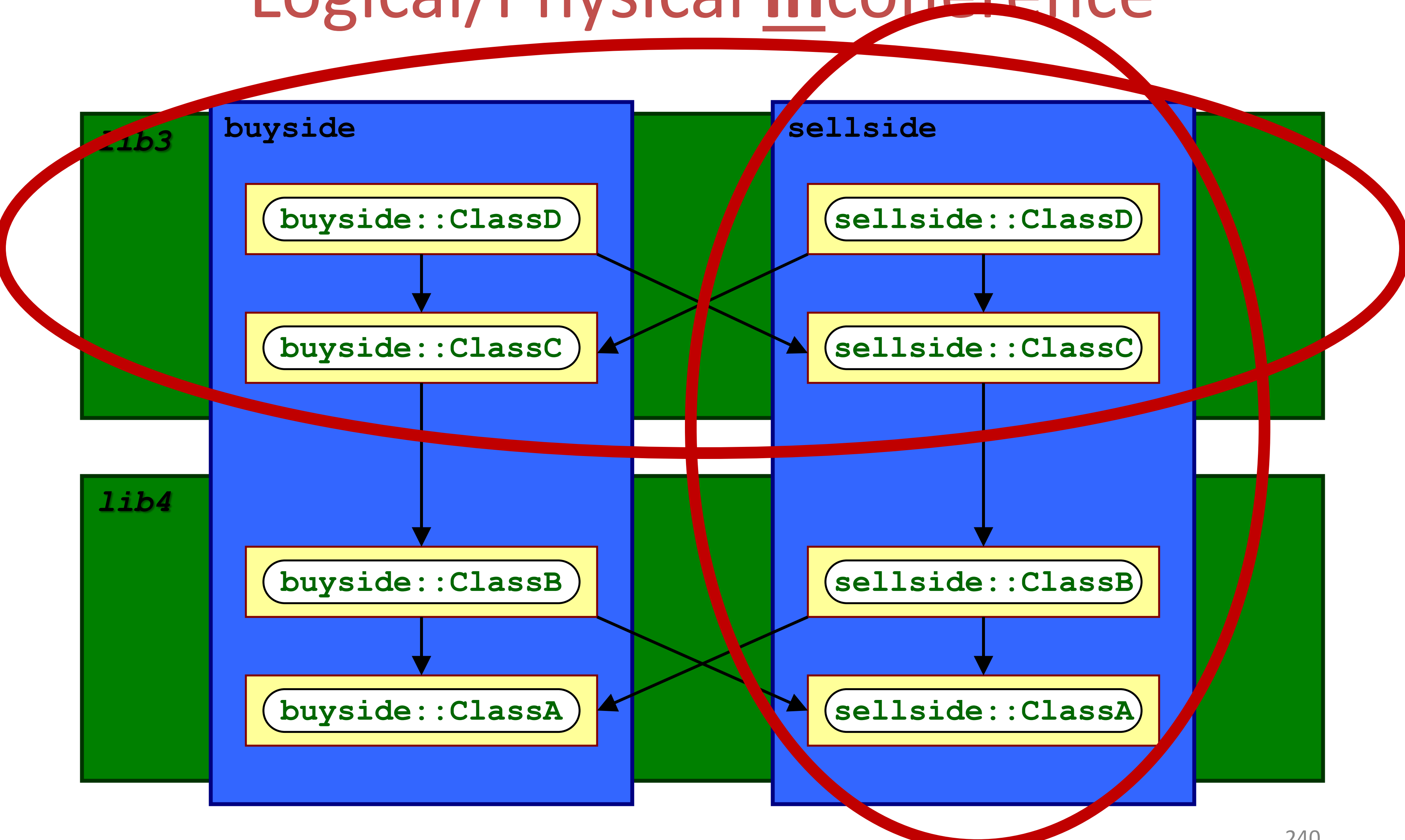
Logical/Physical **Incoherence**



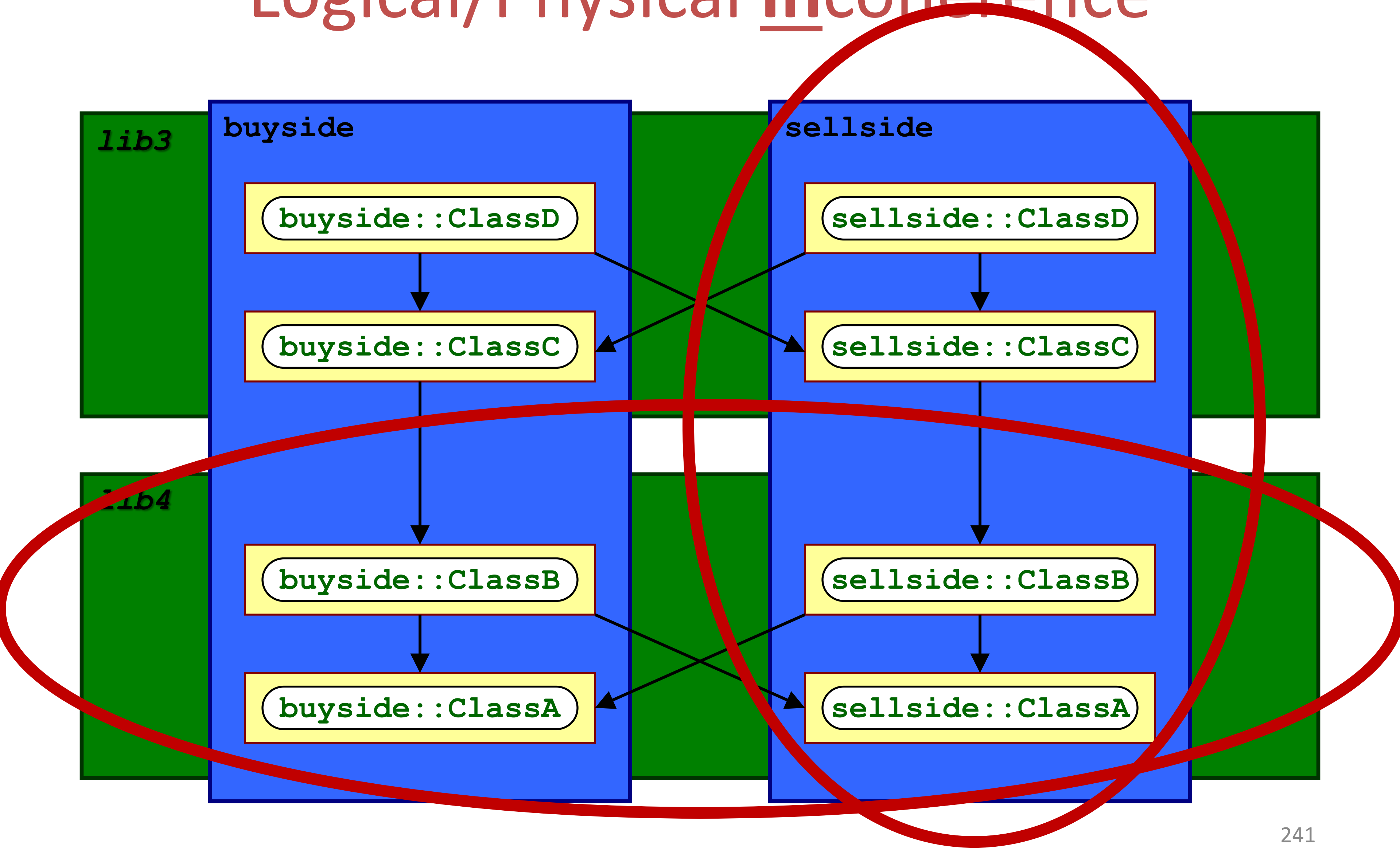
Logical/Physical **In**coherence



Logical/Physical **Incoherence**

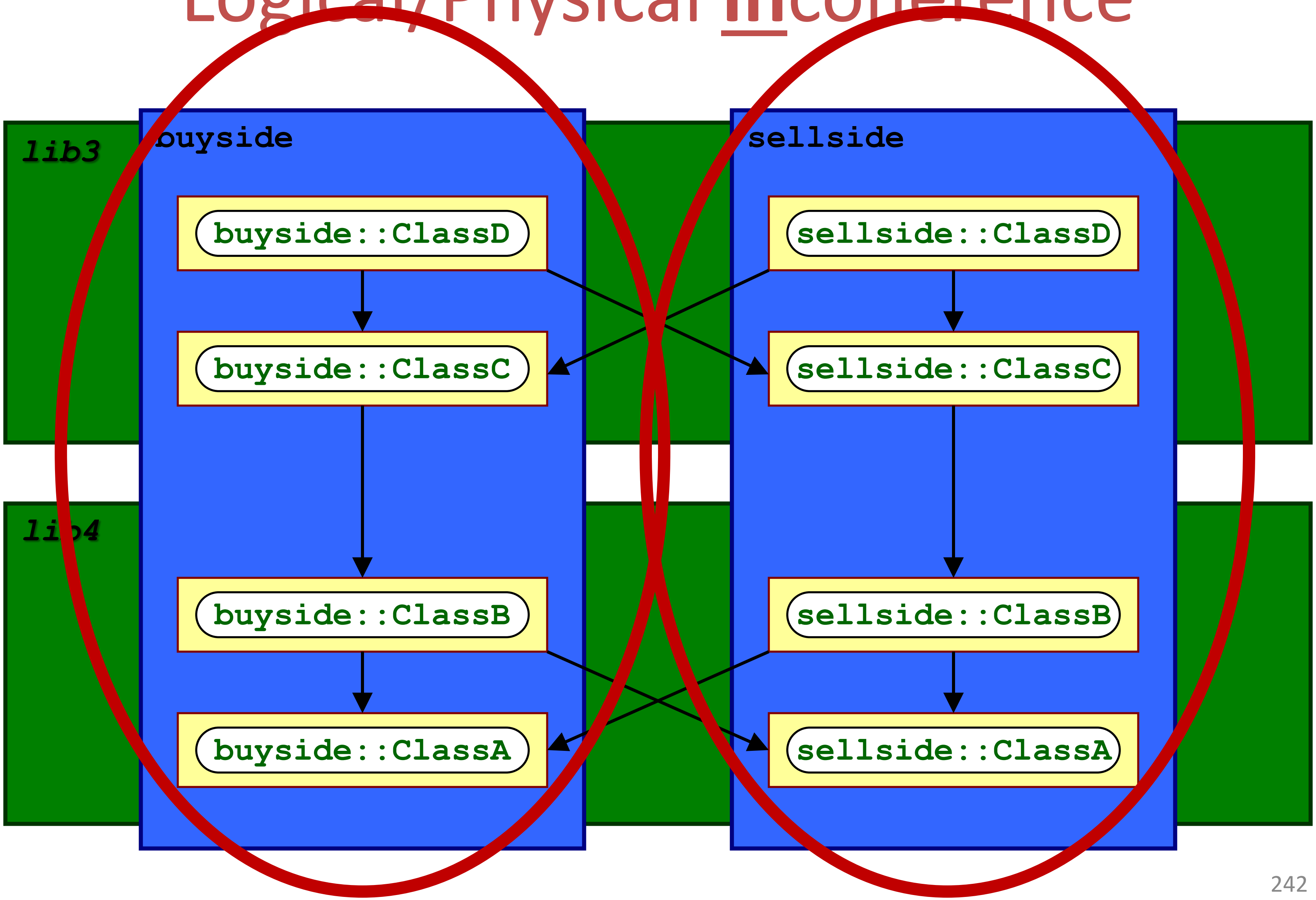


Logical/Physical **Incoherence**



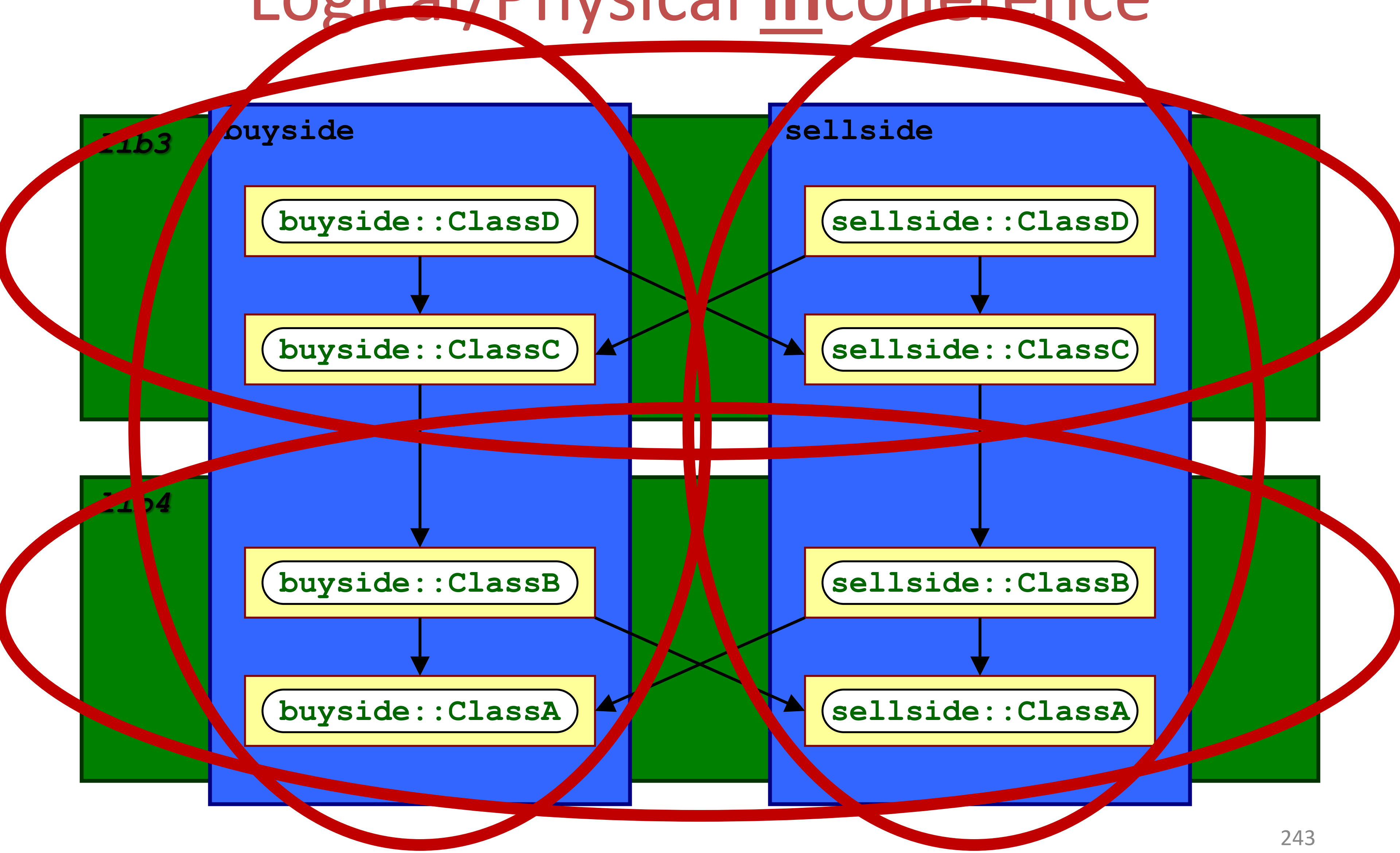
1. Process & Architecture

Logical/Physical **Incoherence**

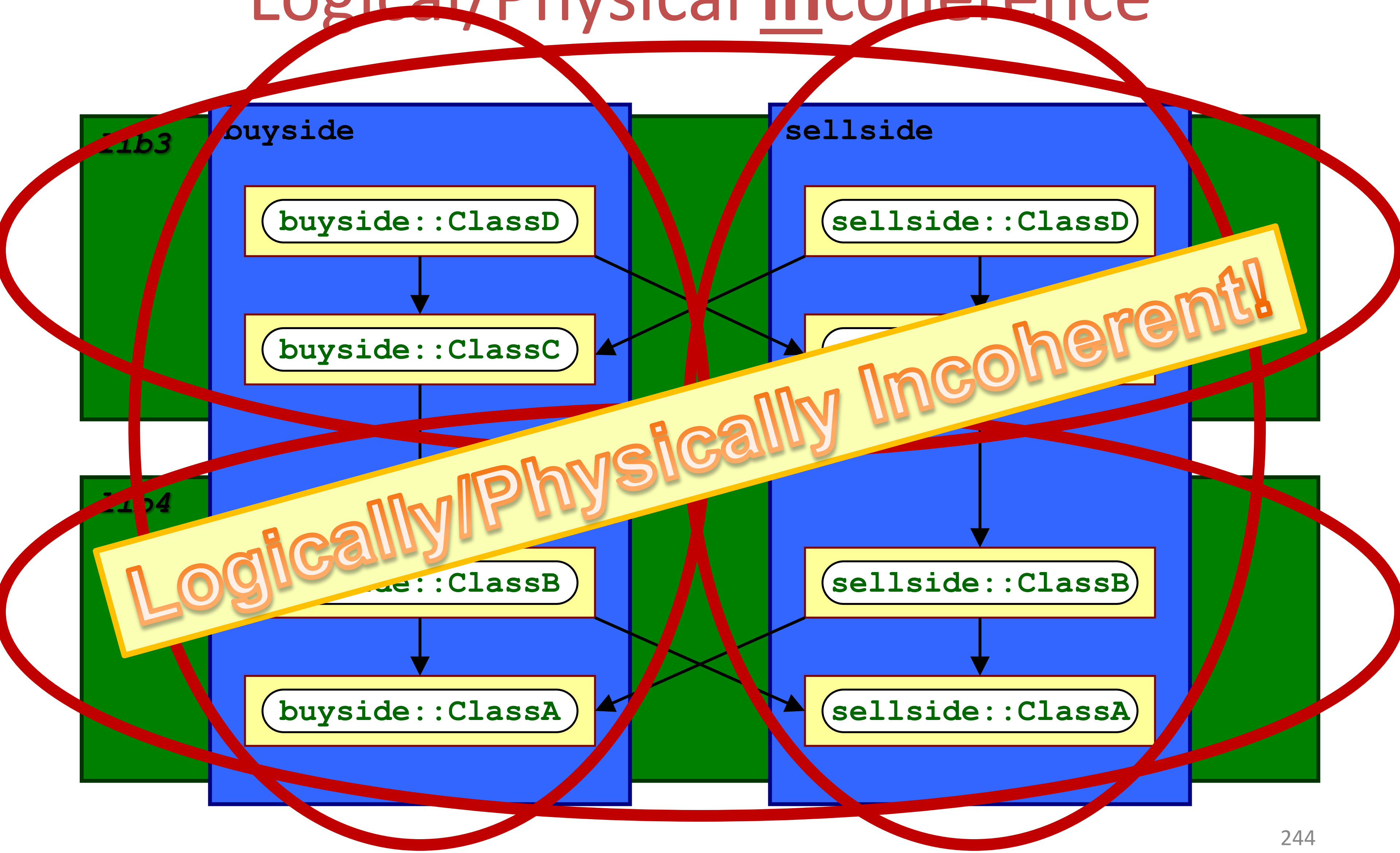


1. Process & Architecture

Logical/Physical <u>In</u>coherence

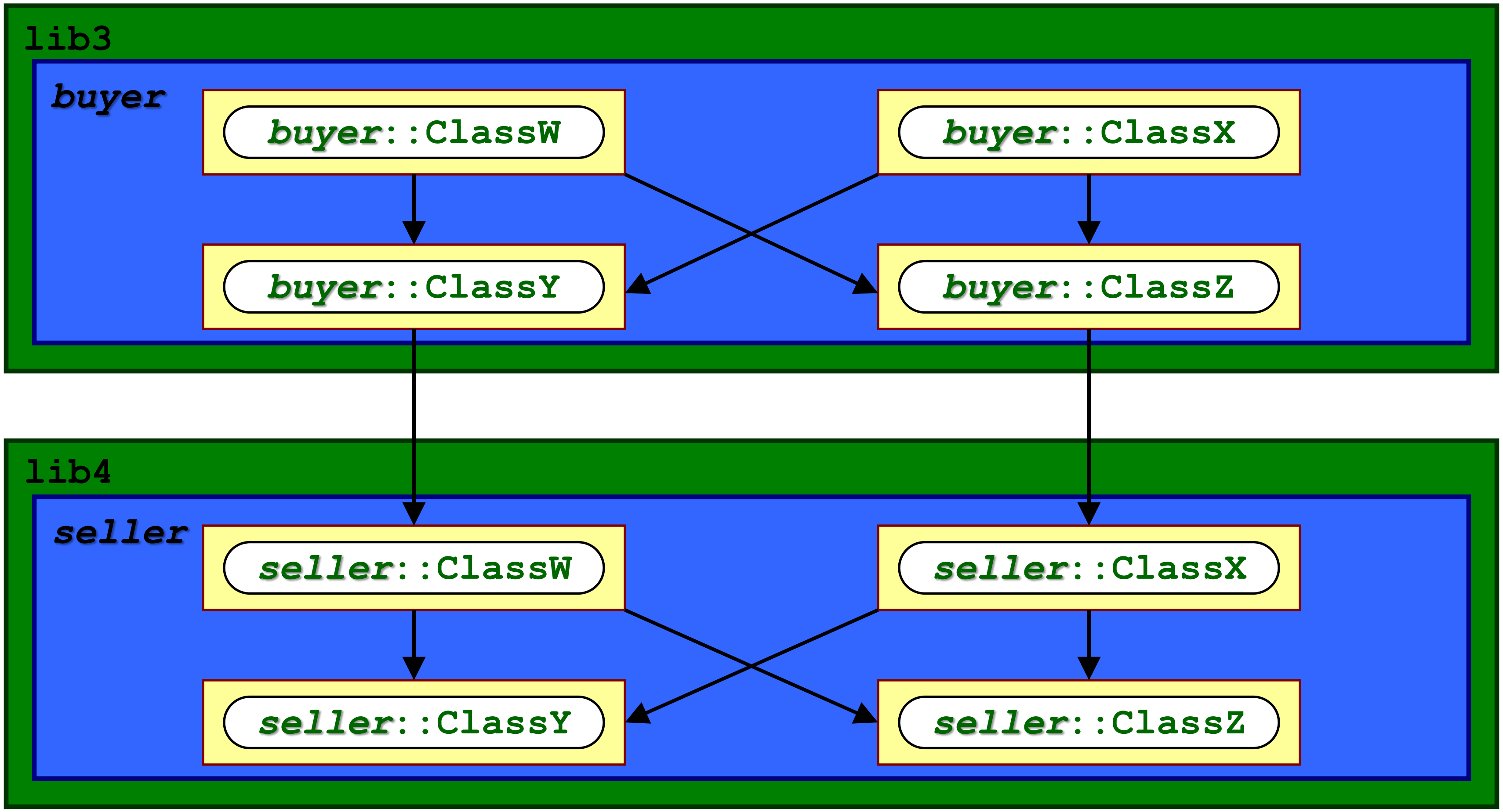


Logical/Physical **Incoherence**

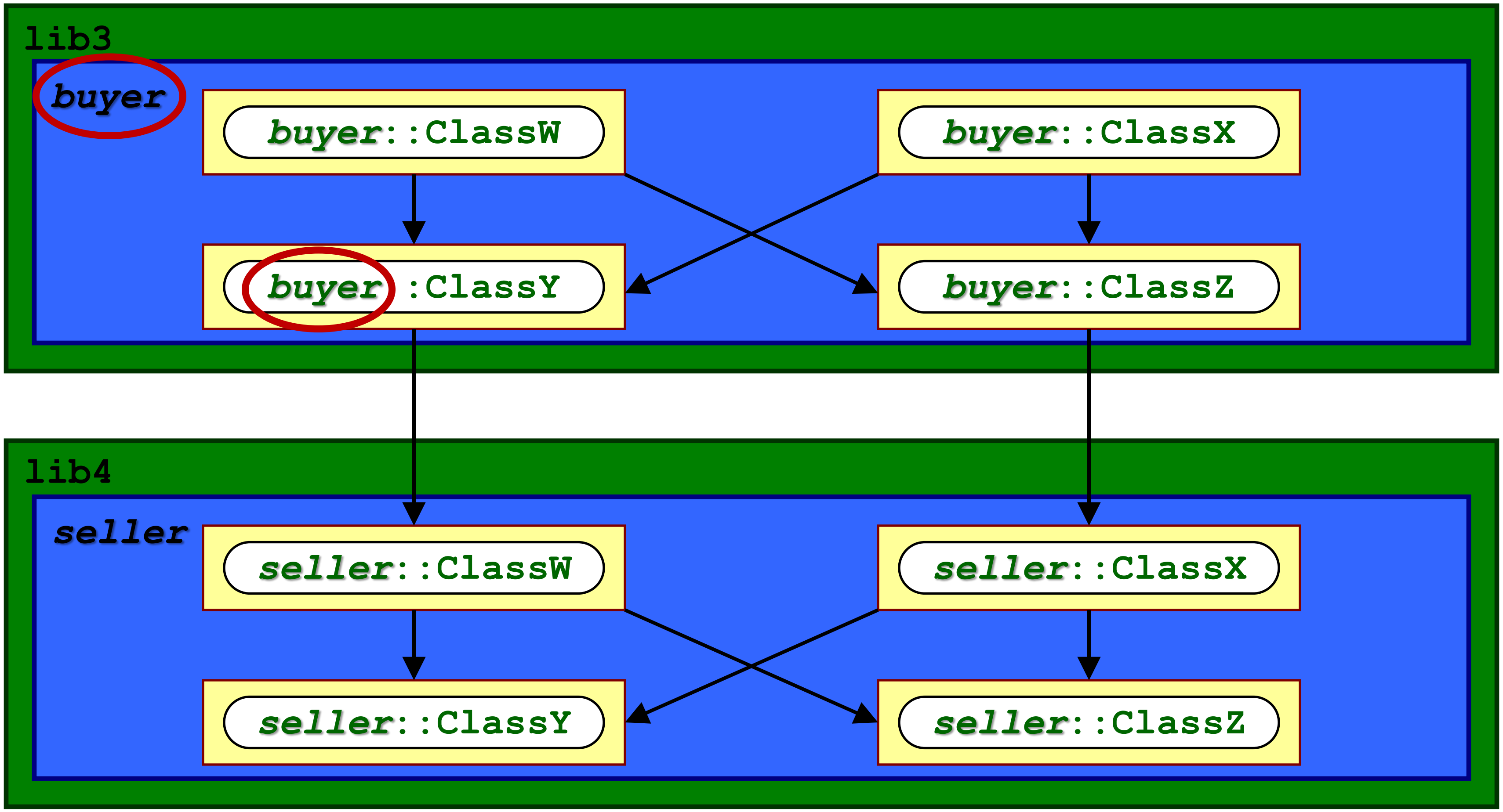


Logical/Physical **<u>In</u>**coherence

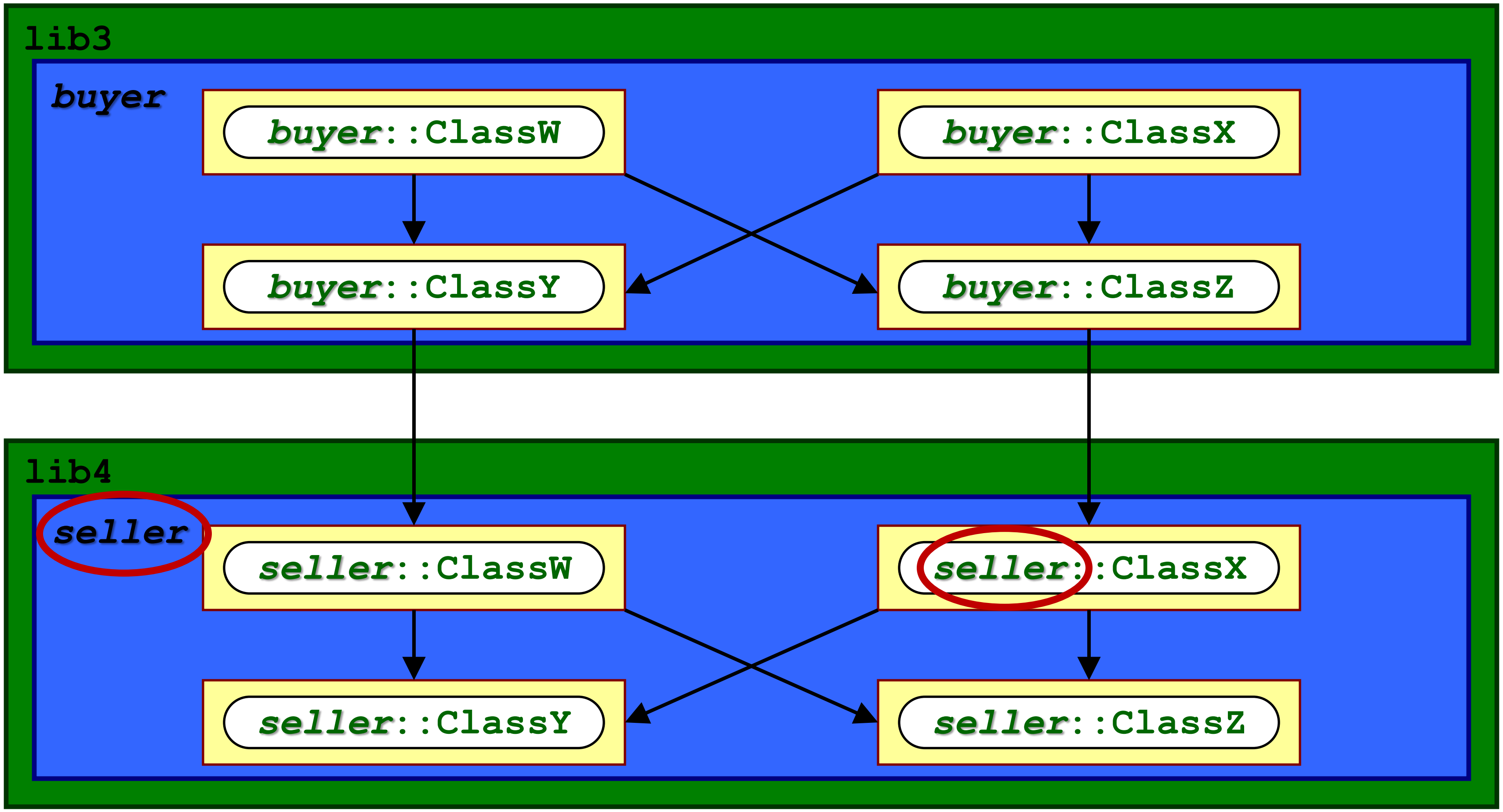
Logical/Physical Coherence



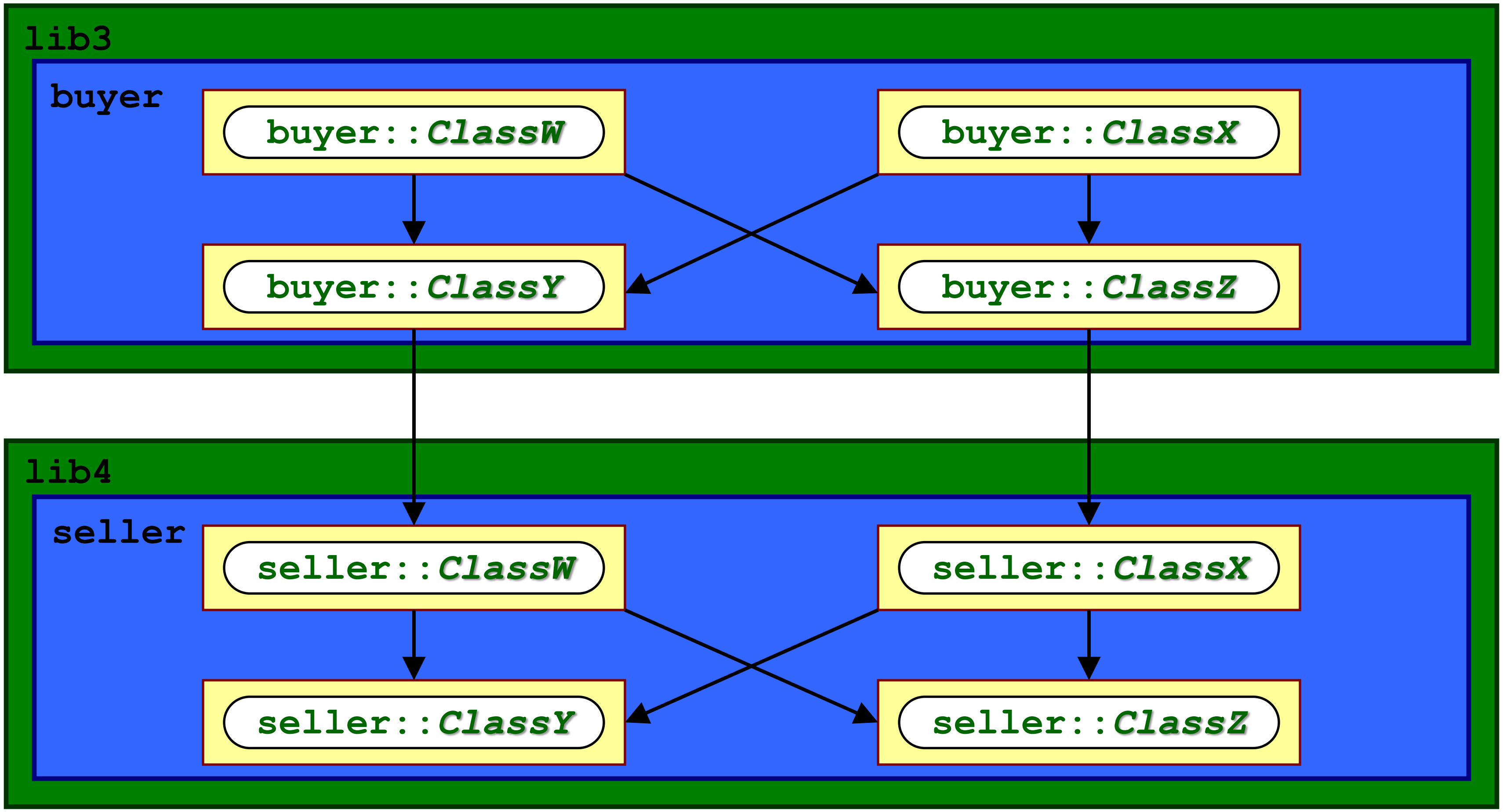
Logical/Physical Coherence



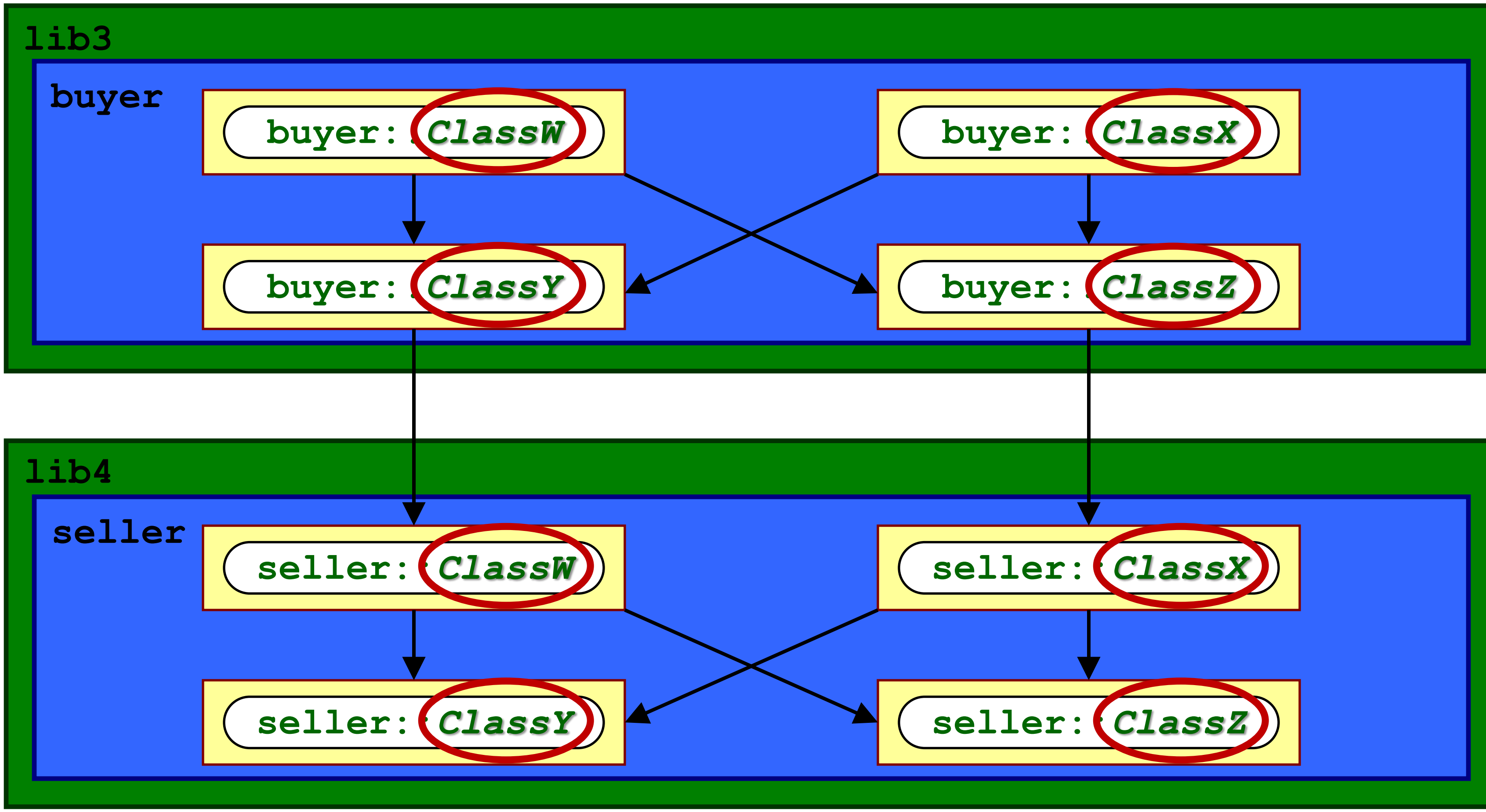
Logical/Physical Coherence



Logical/Physical Coherence

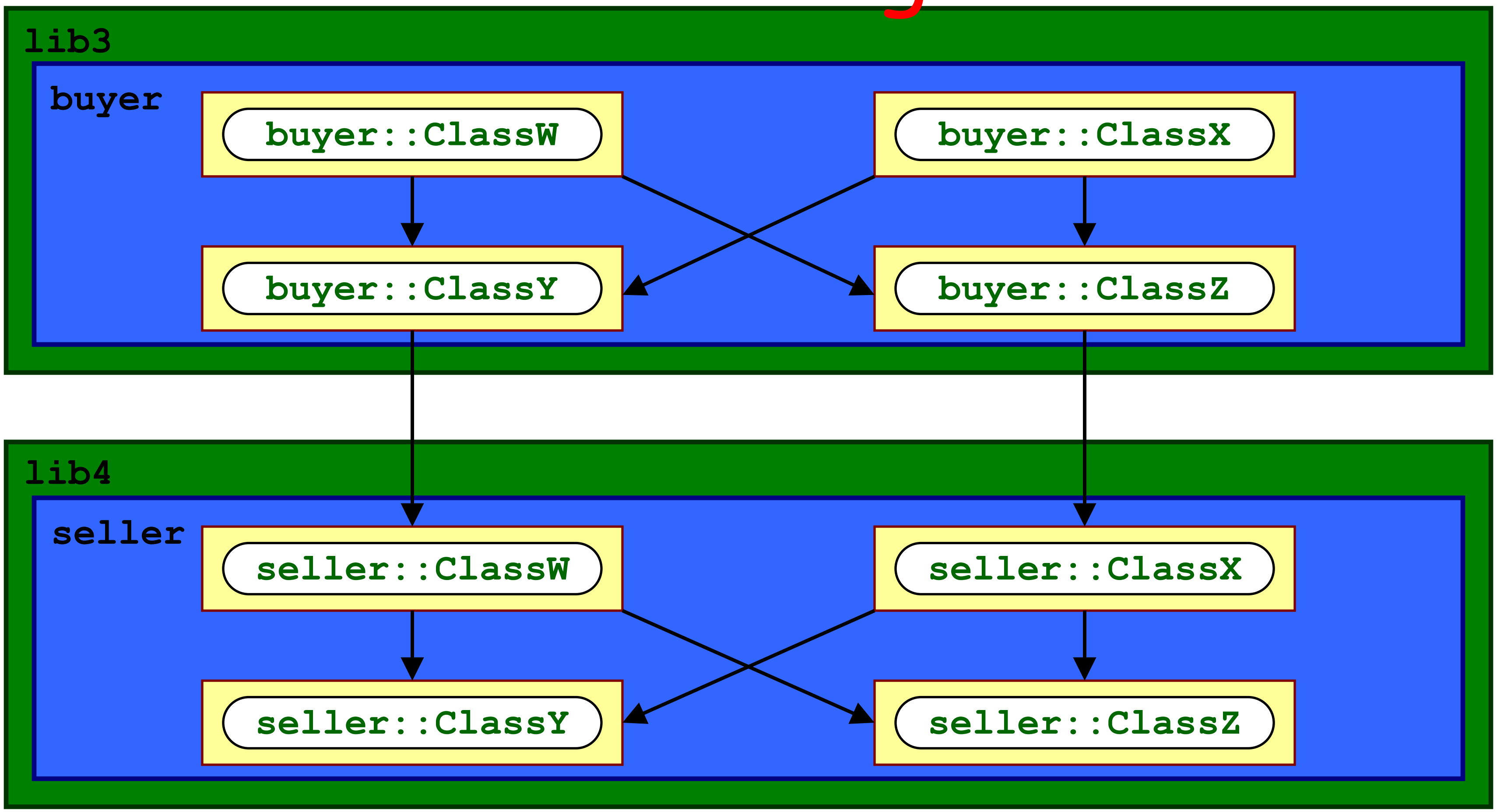


Logical/Physical Coherence



Logical/Physical Coherence

This is the goal!



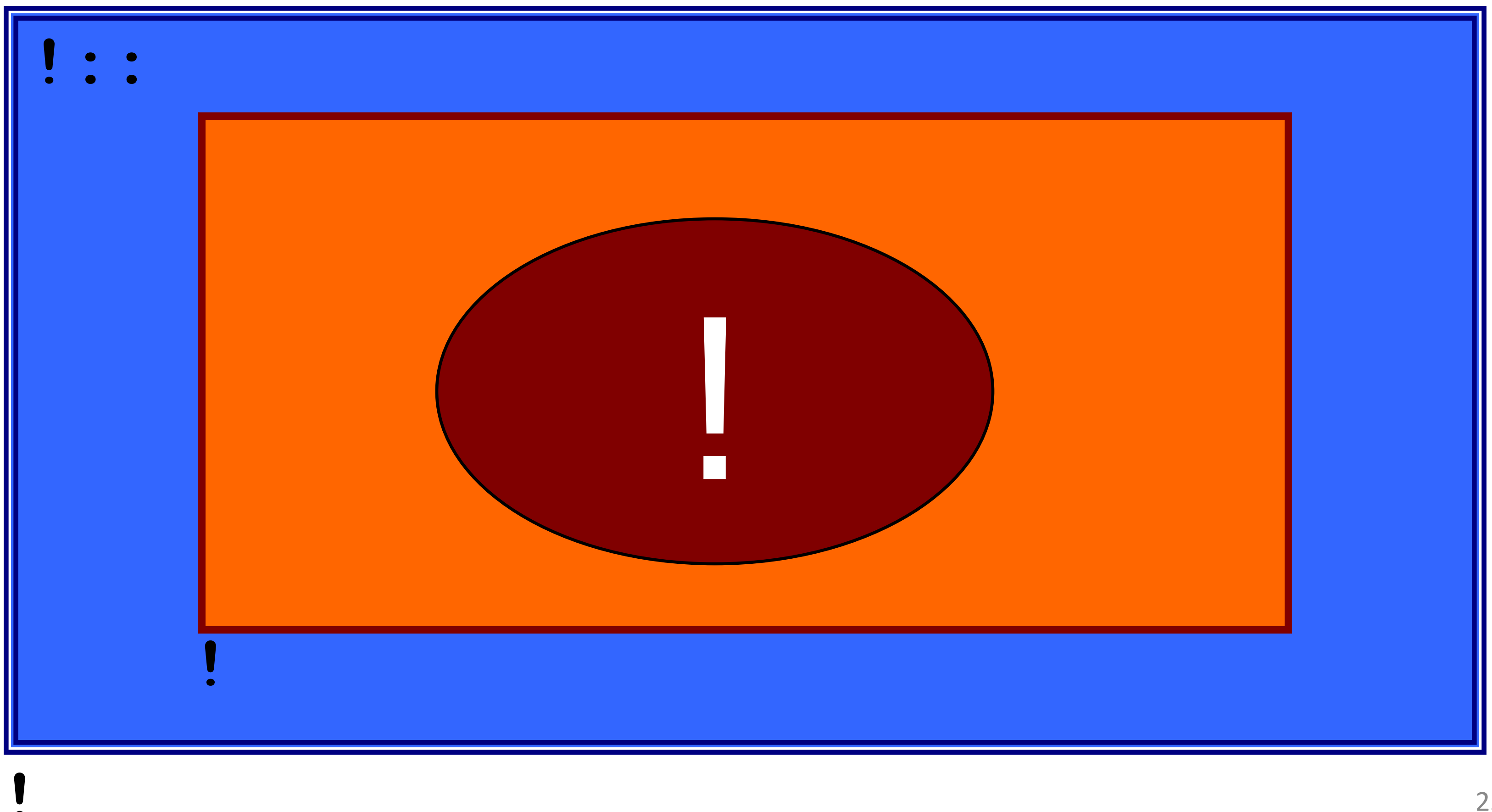
Logical/Physical Synergy

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).

Logical/Physical Name Cohesion

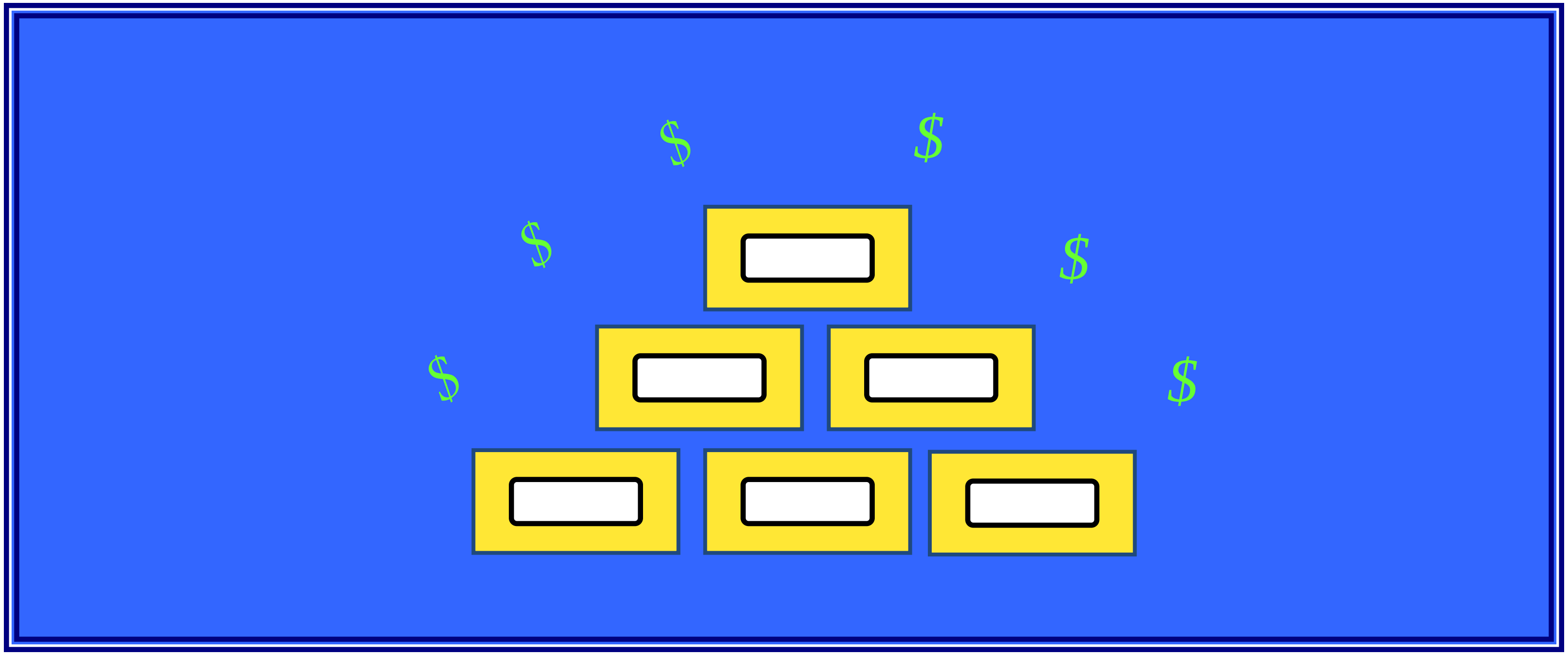
→ Key Concept ←



Packages

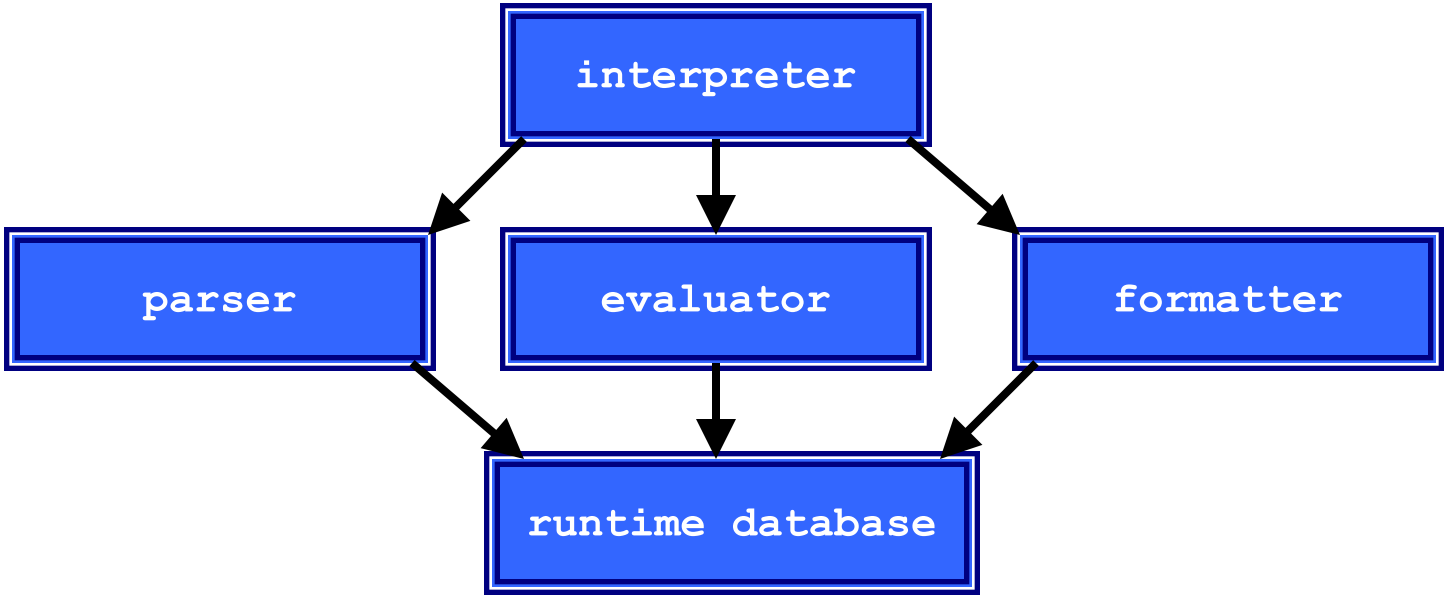
Classical Definition

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



Packages

High-Level Interpreter Architecture



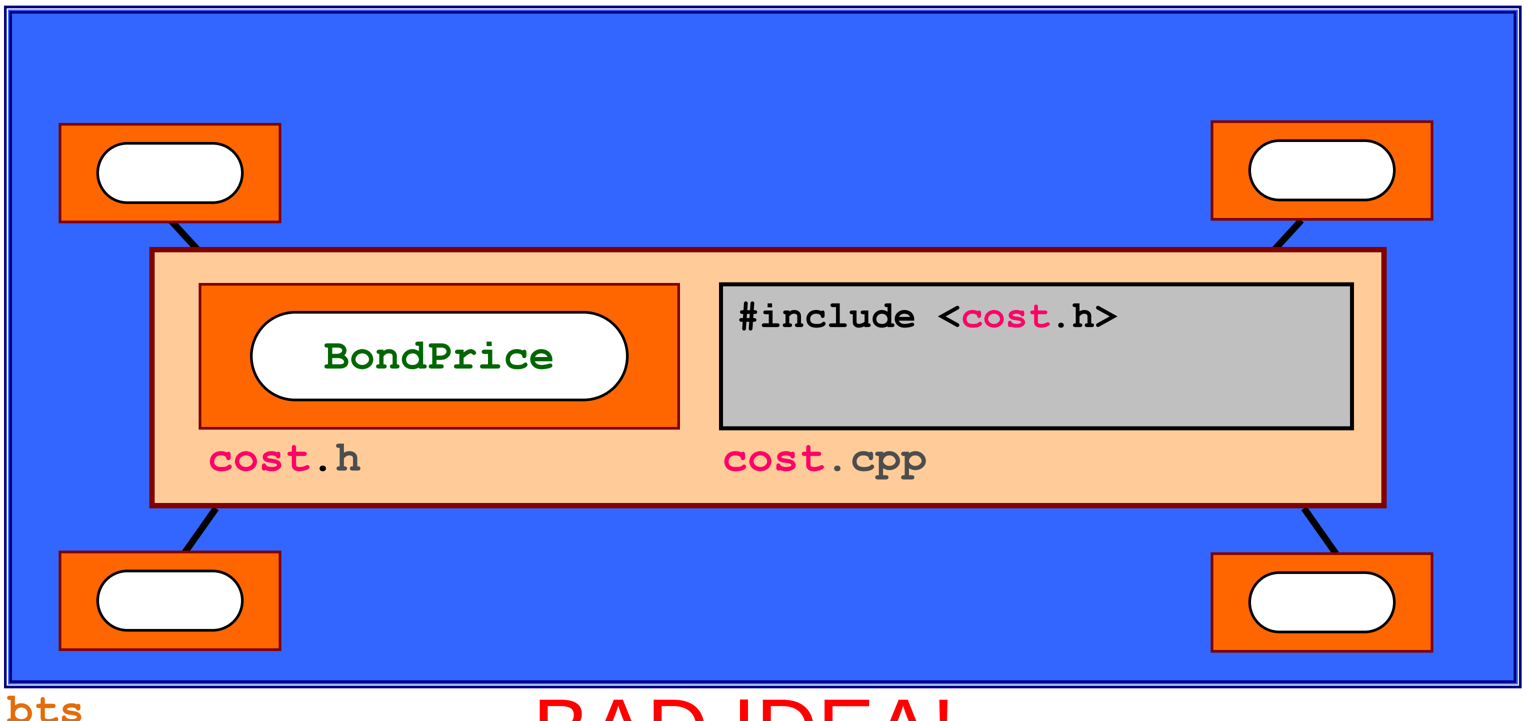
Architecturally Significant Names

Non-Cohesive Logical and Physical Names

Package Name: bts

Component Name: cost

Class Name: BondPrice



BAD IDEA!

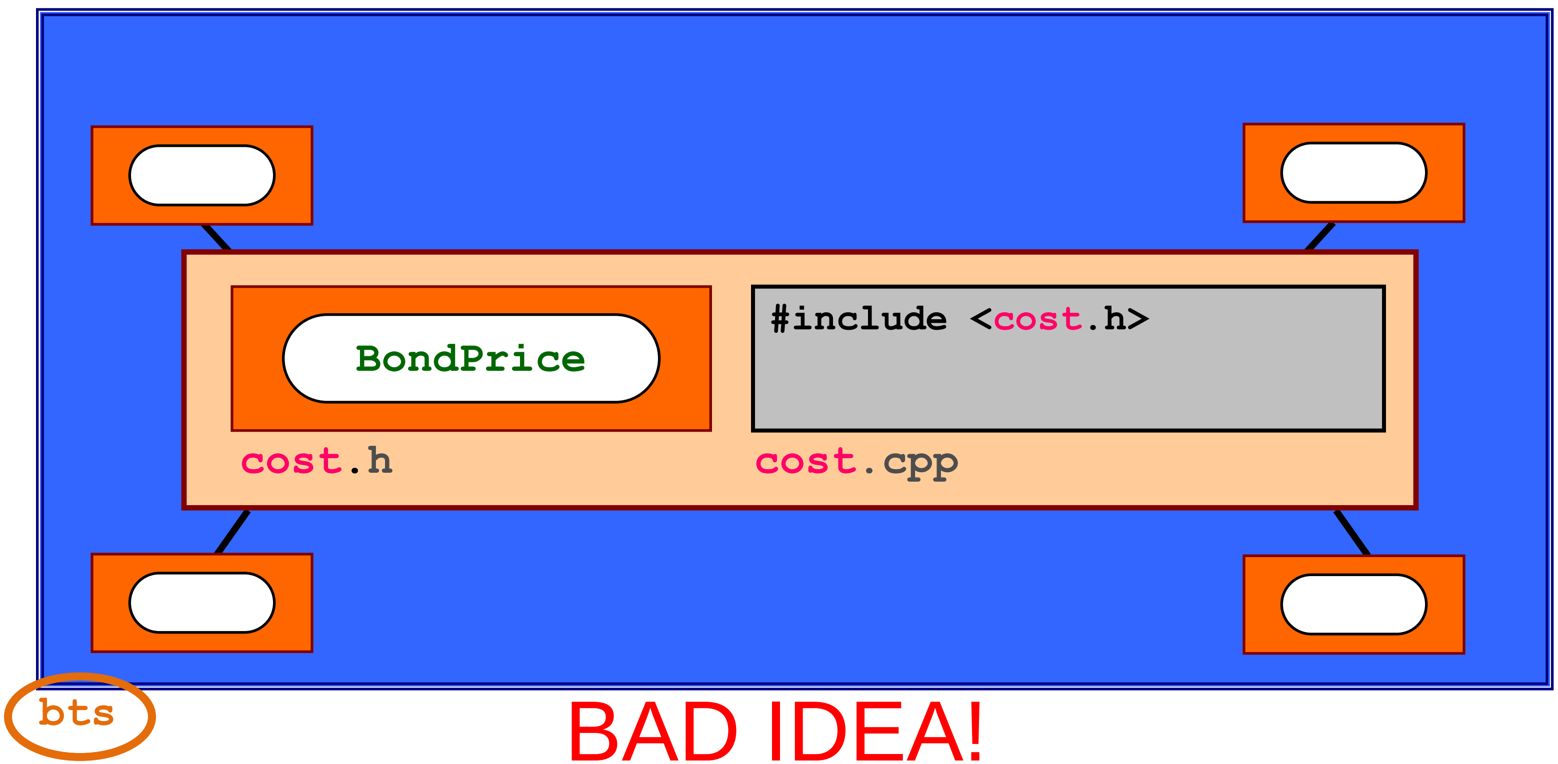
Architecturally Significant Names

Non-Cohesive Logical and Physical Names

Package Name: bts

Component Name: cost

Class Name: BondPrice



BAD IDEA!

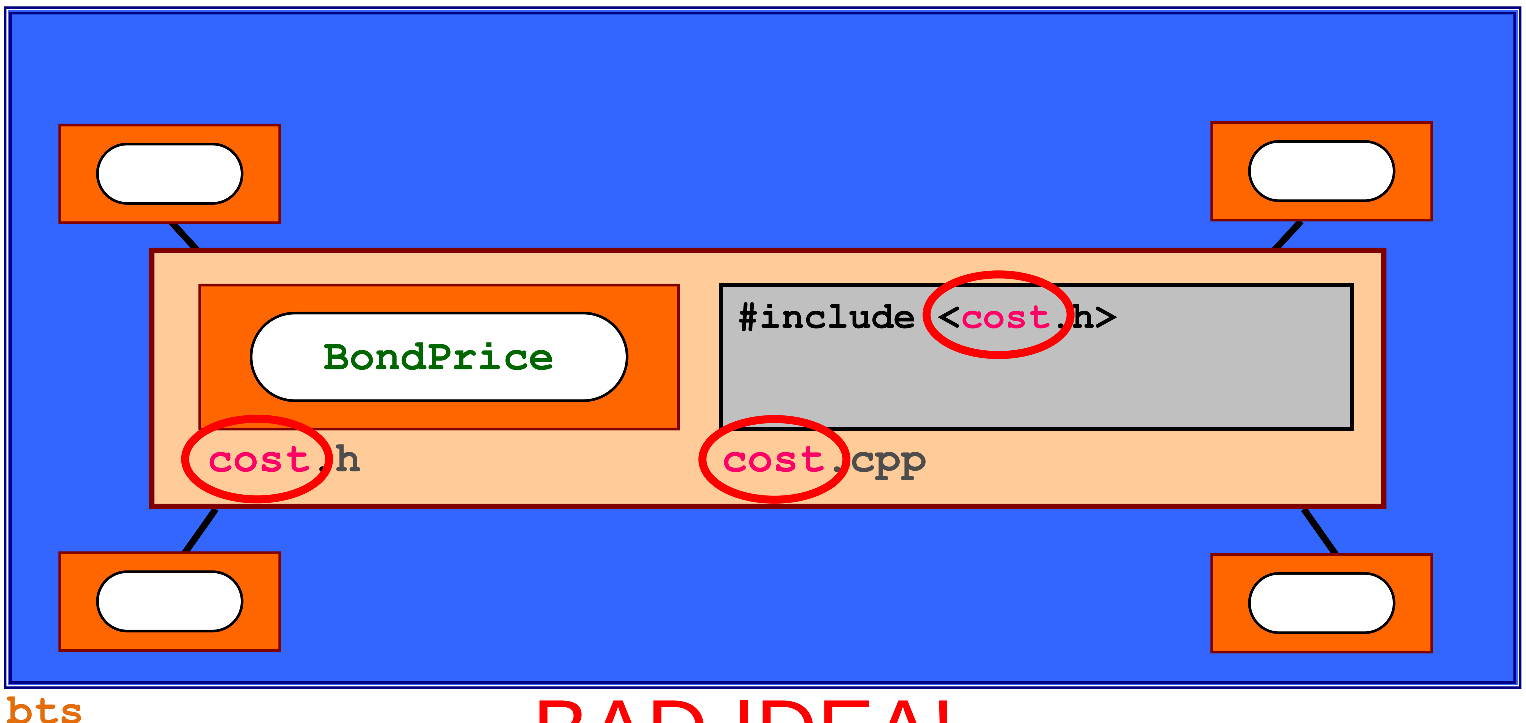
Architecturally Significant Names

Non-Cohesive Logical and Physical Names

Package Name: bts

Component Name: cost

Class Name: BondPrice



BAD IDEA!

Architecturally Significant Names

Non-Cohesive Logical and Physical Names

Package Name: bts

Component Name: cost

Class Name: BondPrice

BAD IDEA!

Architecturally Significant Names

Definition

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

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

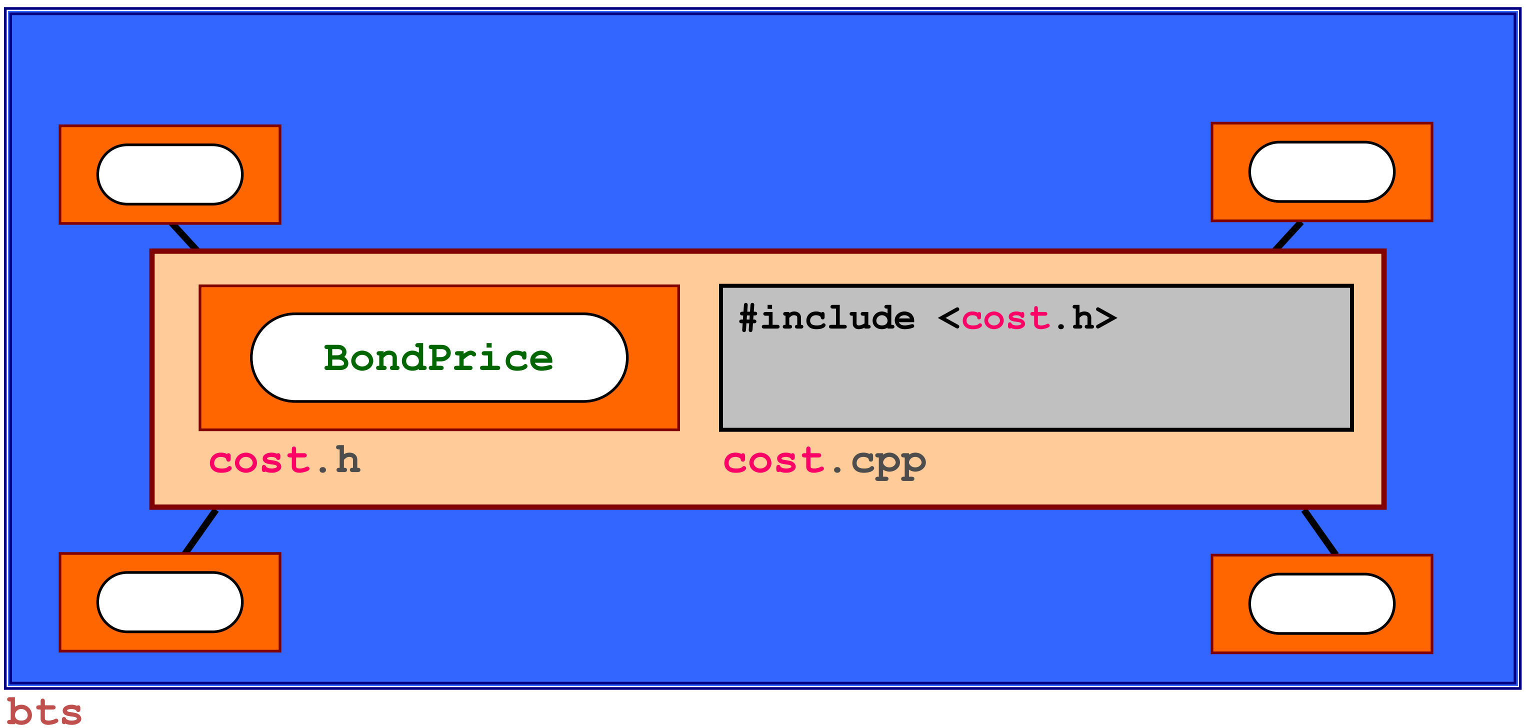
- Unit Of Release **(UOR)**
- (library) **component**

must be **unique** throughout the enterprise.

Physical Package Prefixes

Component Name **Not Matching** Package Name:

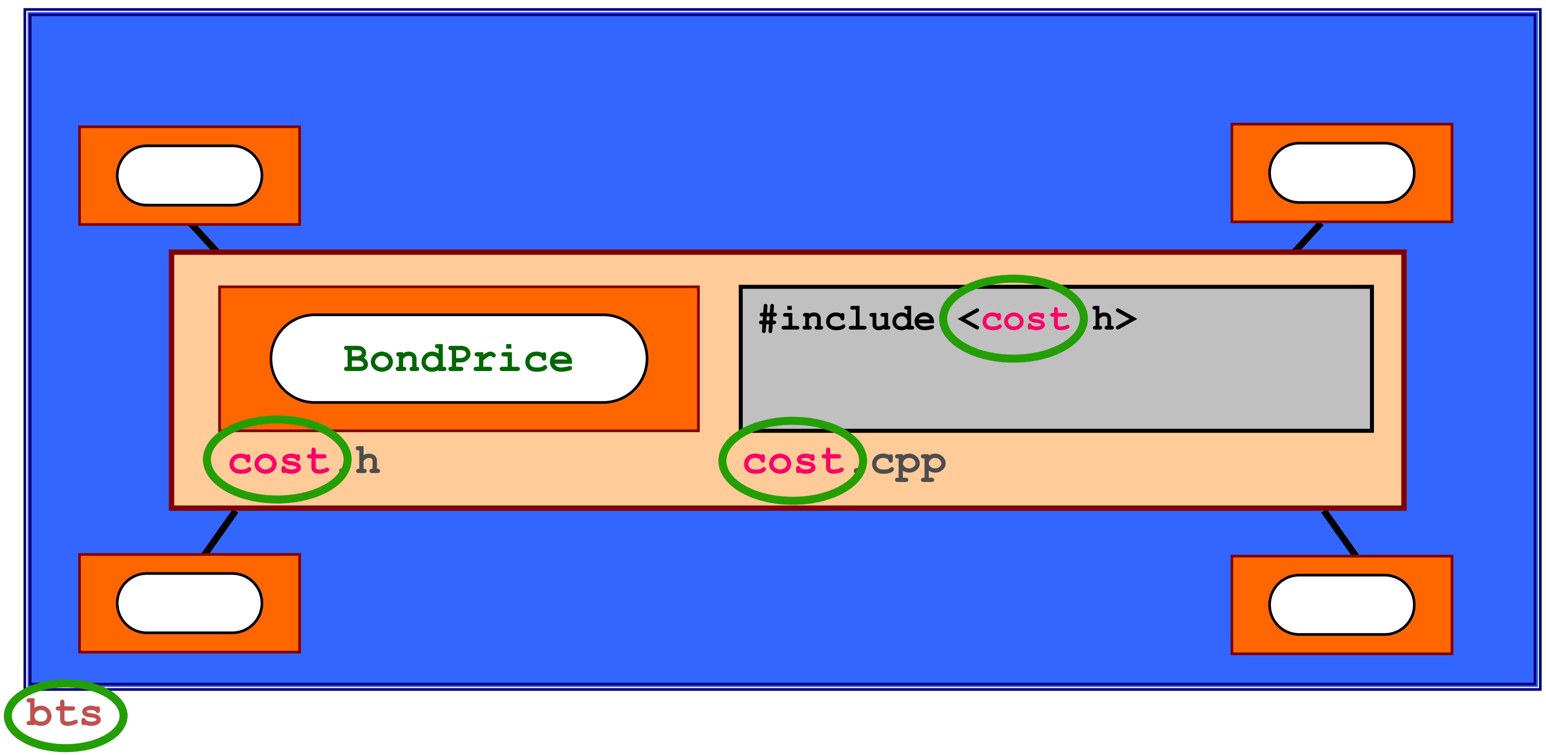
cost



Physical Package Prefixes

Component Name **Not Matching** Package Name:

cost



Physical Package Prefixes

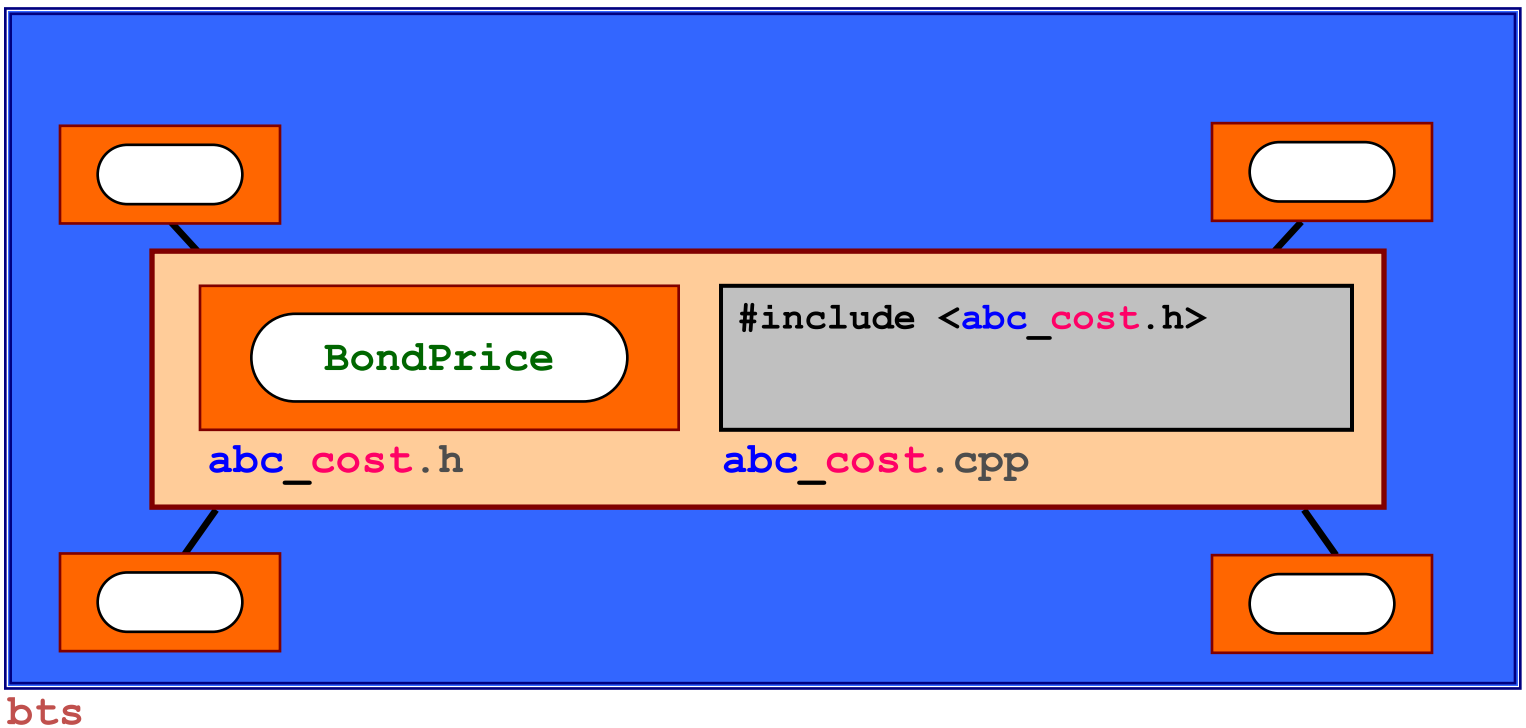
Design Rule

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

Physical Package Prefixes

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

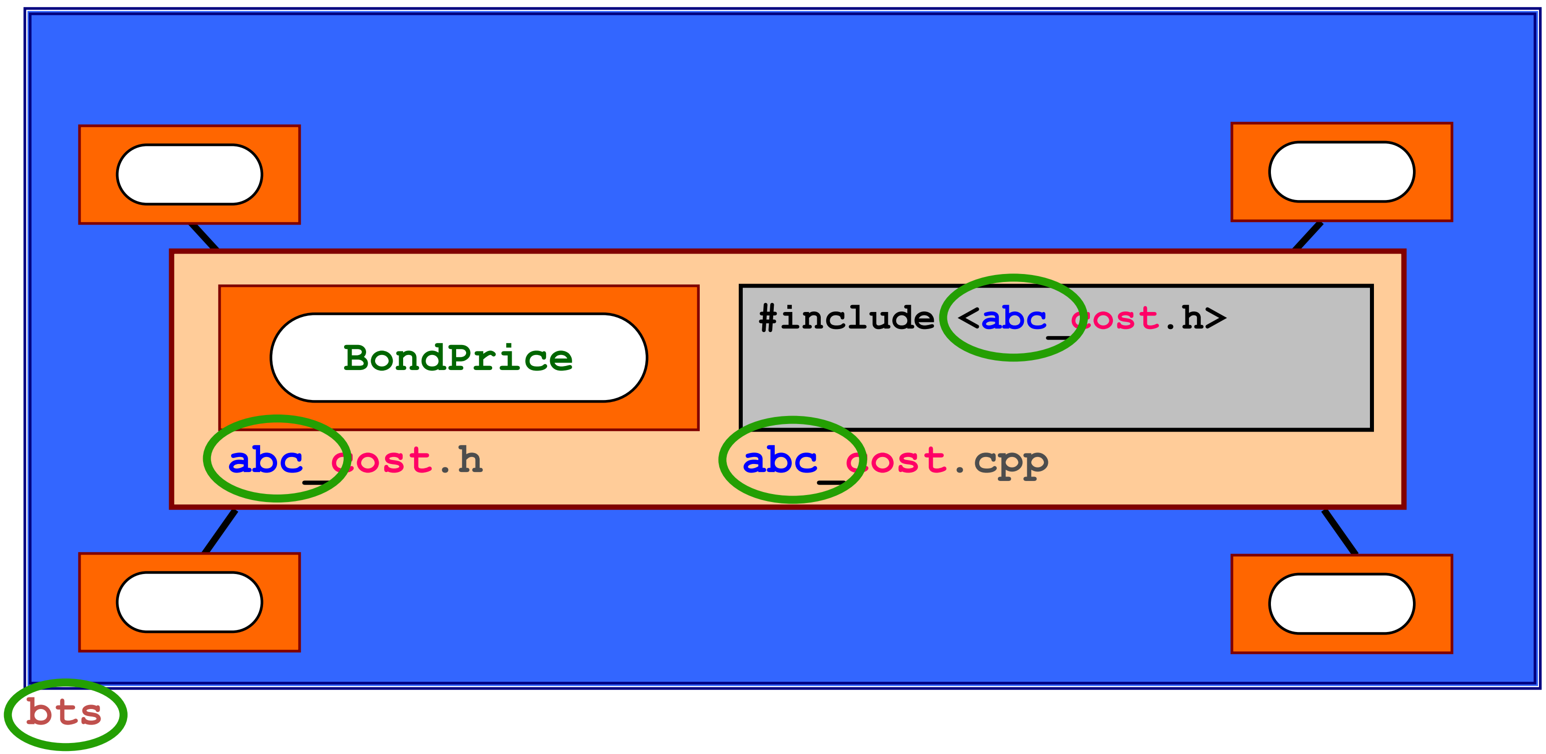
abc_cost



Physical Package Prefixes

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

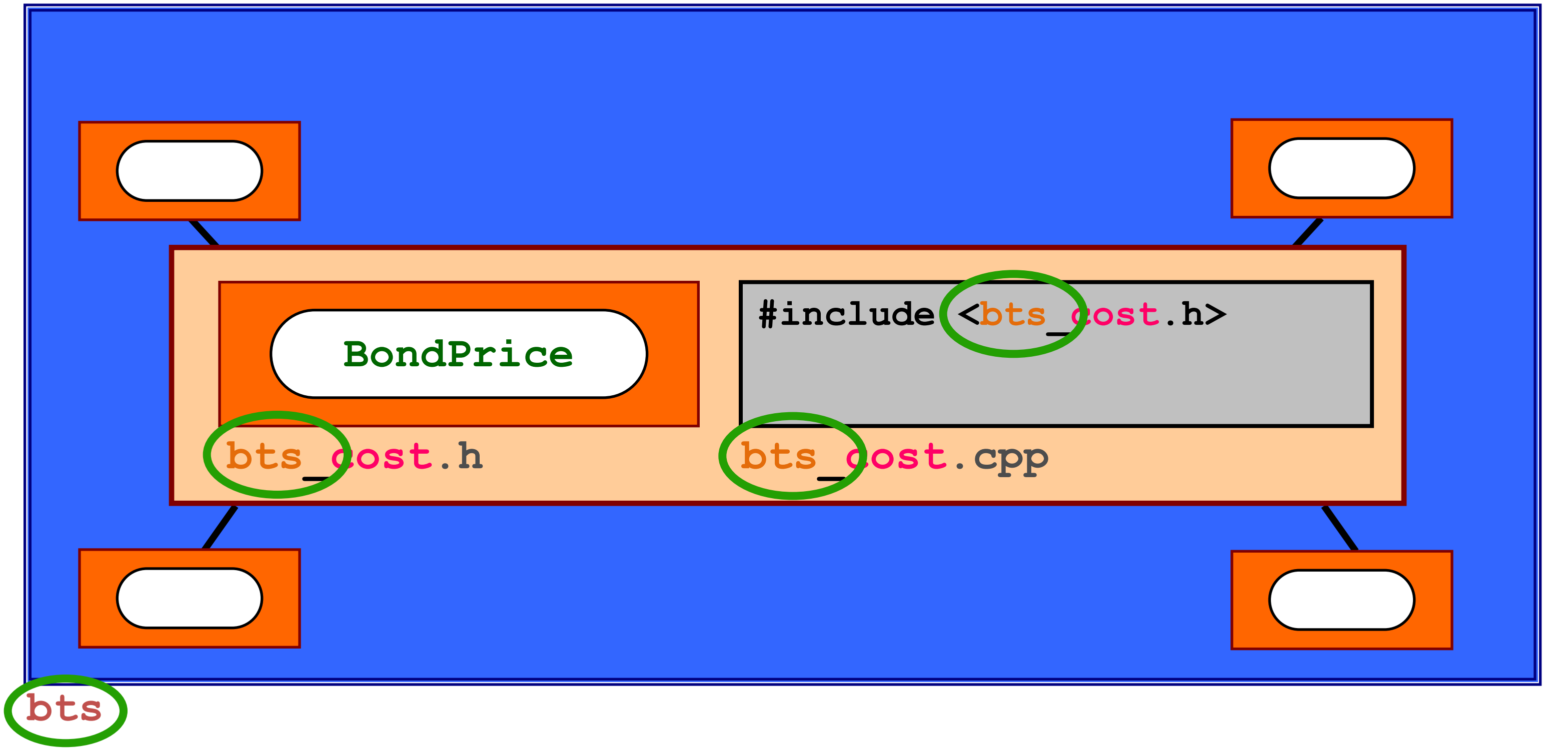
abc_cost



Physical Package Prefixes

Component Prefix **Matches** Package Name:

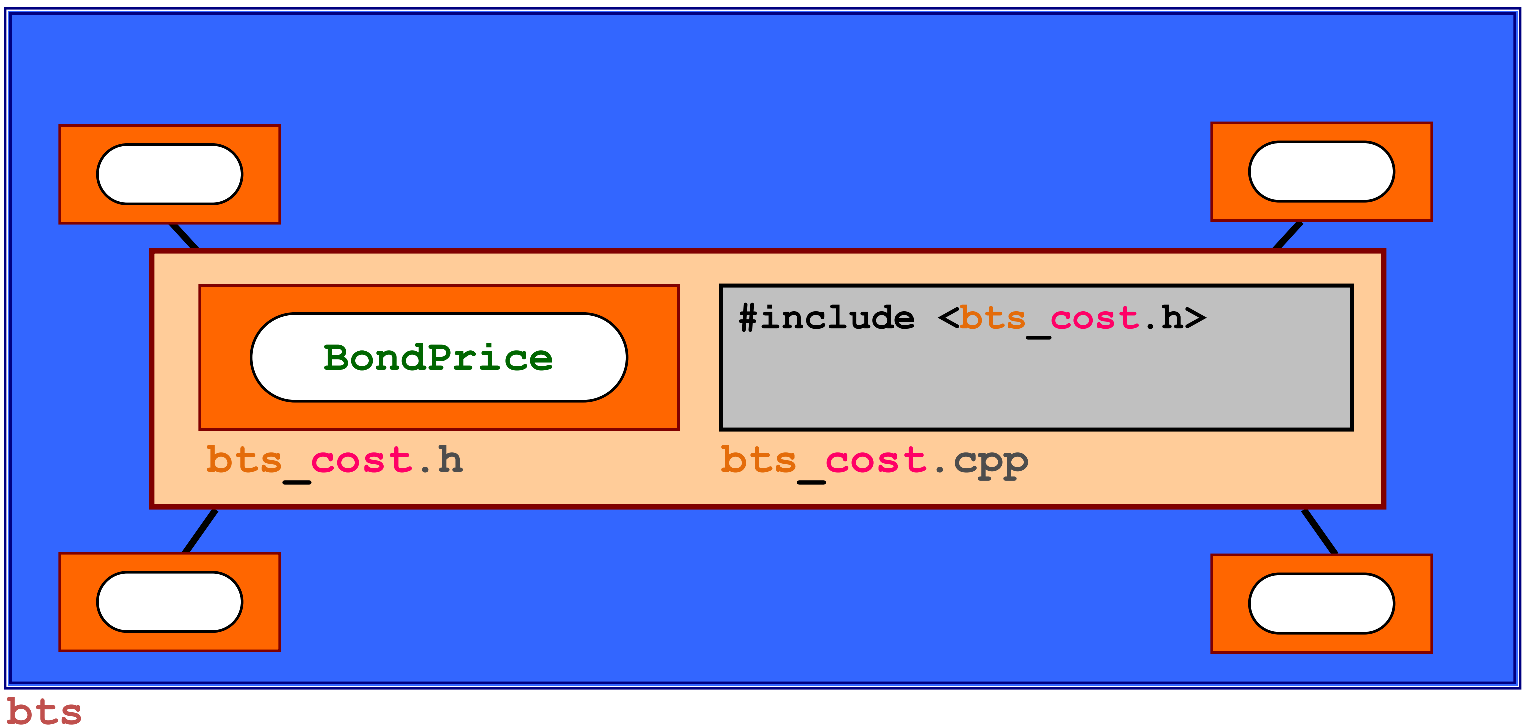
bts_cost



Physical Package Prefixes

Component Prefix **Matches** Package Name:

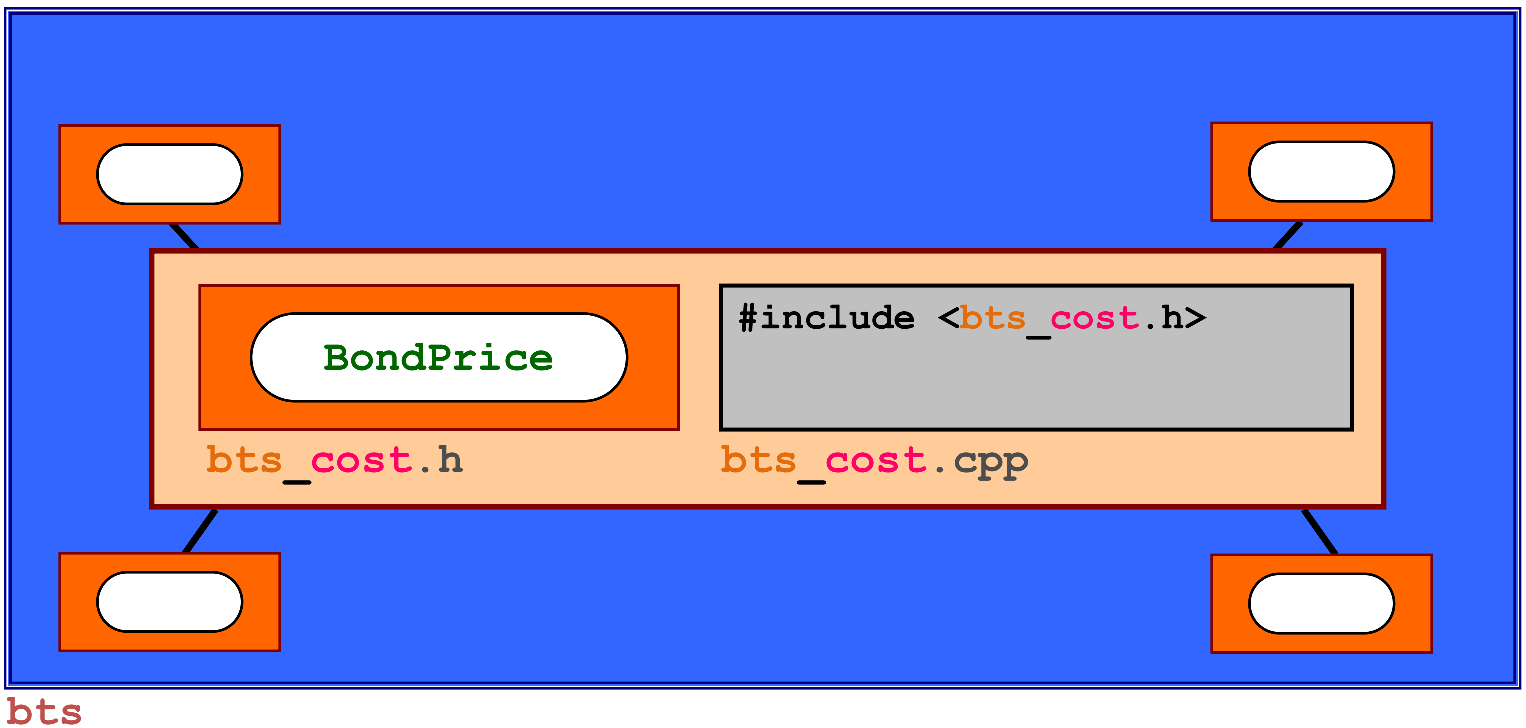
bts_cost



Logical Package Namespaces

Package Namespace **Should Match** Package Name

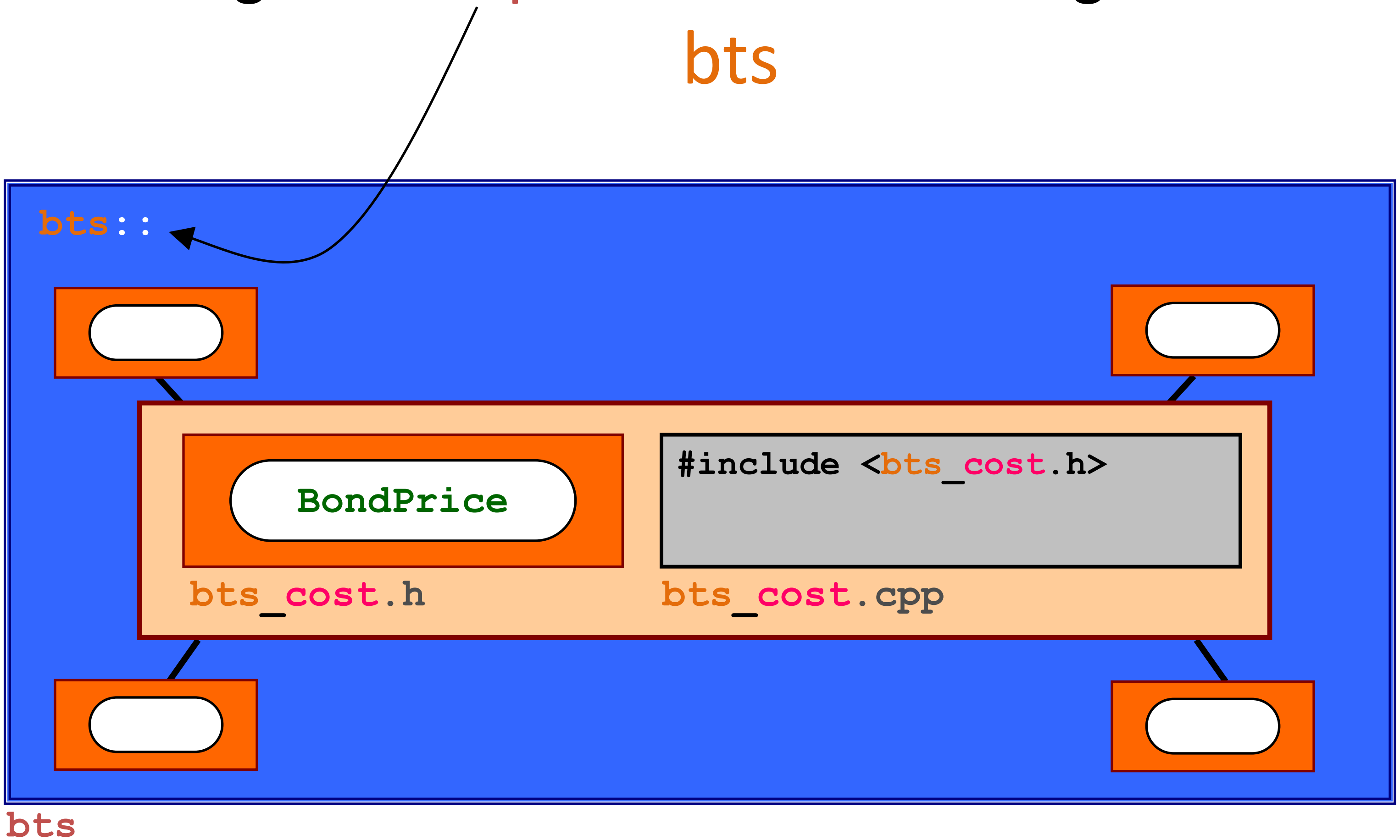
bts



Logical Package Namespaces

Package Namespace **Matches** Package Name

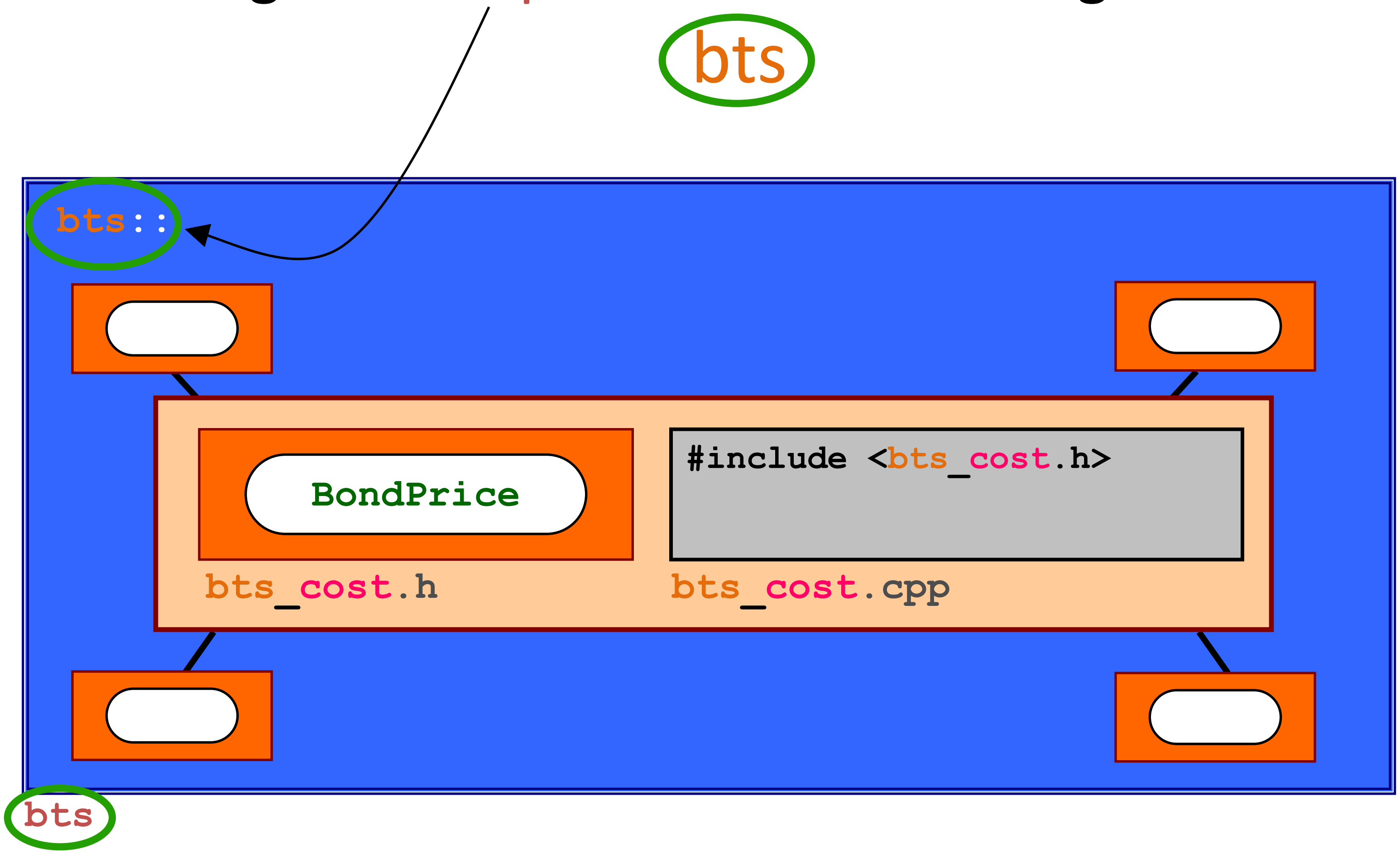
bts



Logical Package Namespaces

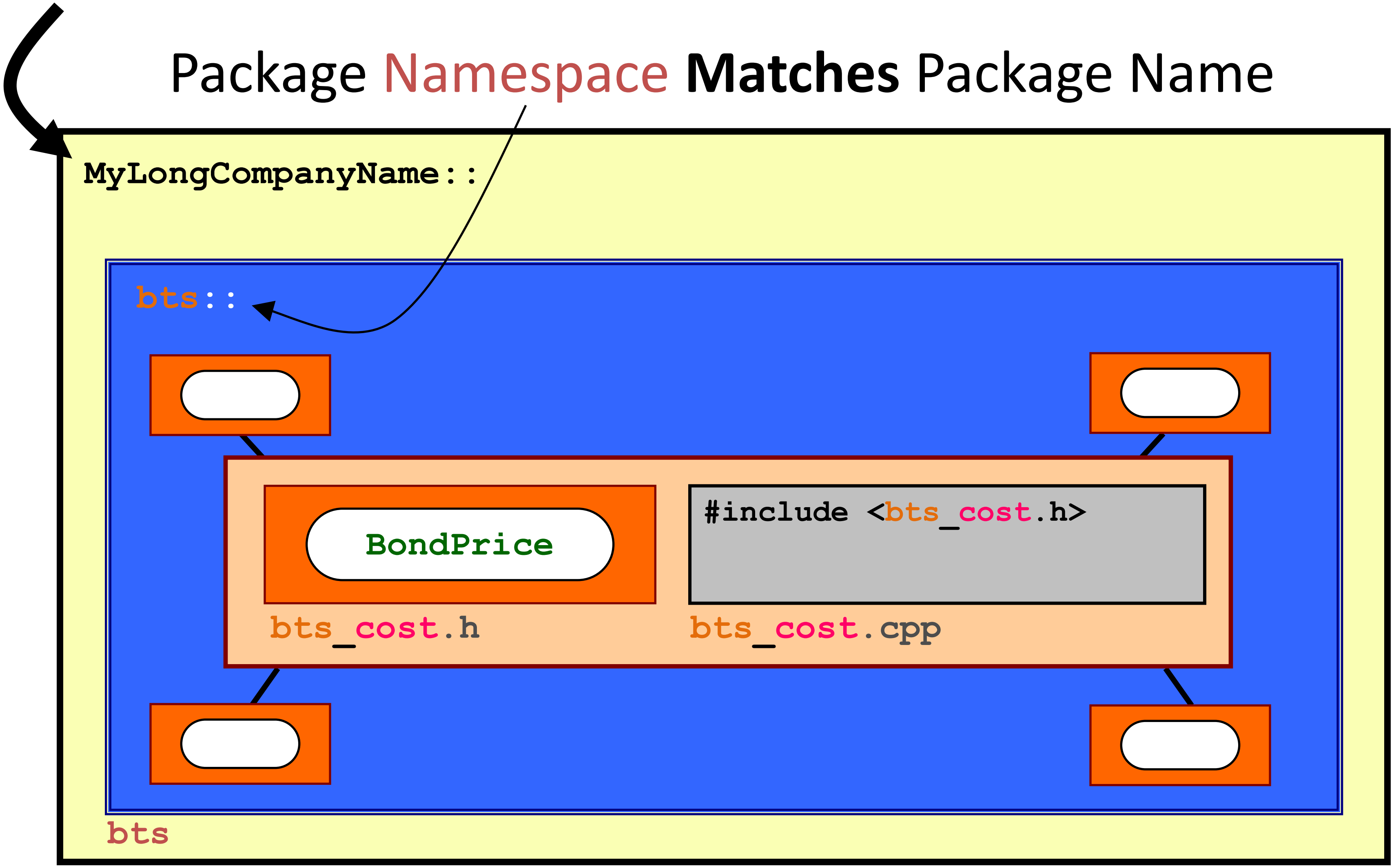
Package Namespace **Matches** Package Name

bts



(Logical) Enterprise-Wide Namespace

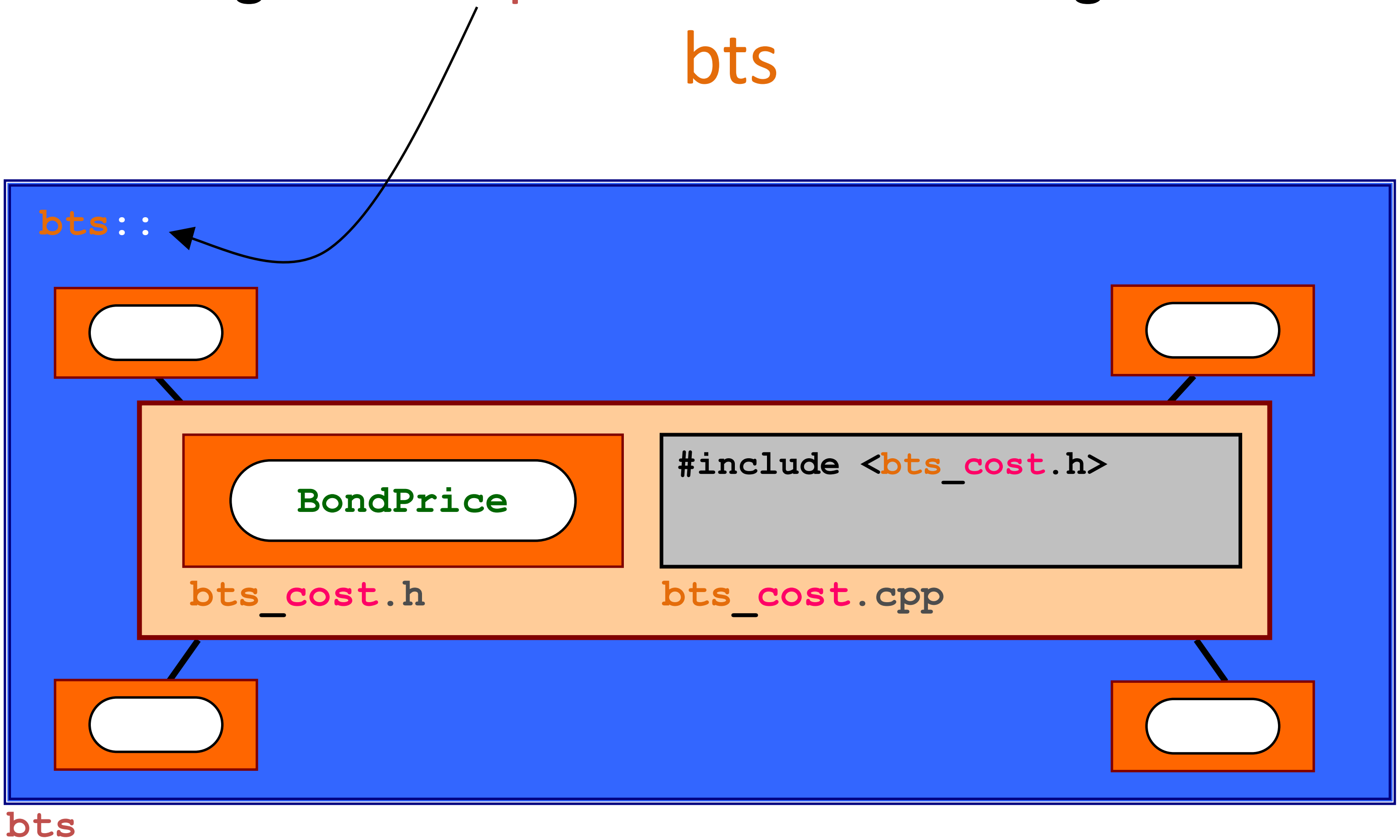
Package Namespace **Matches** Package Name



Logical Package Namespaces

Package Namespace **Matches** Package Name

bts



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.

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.

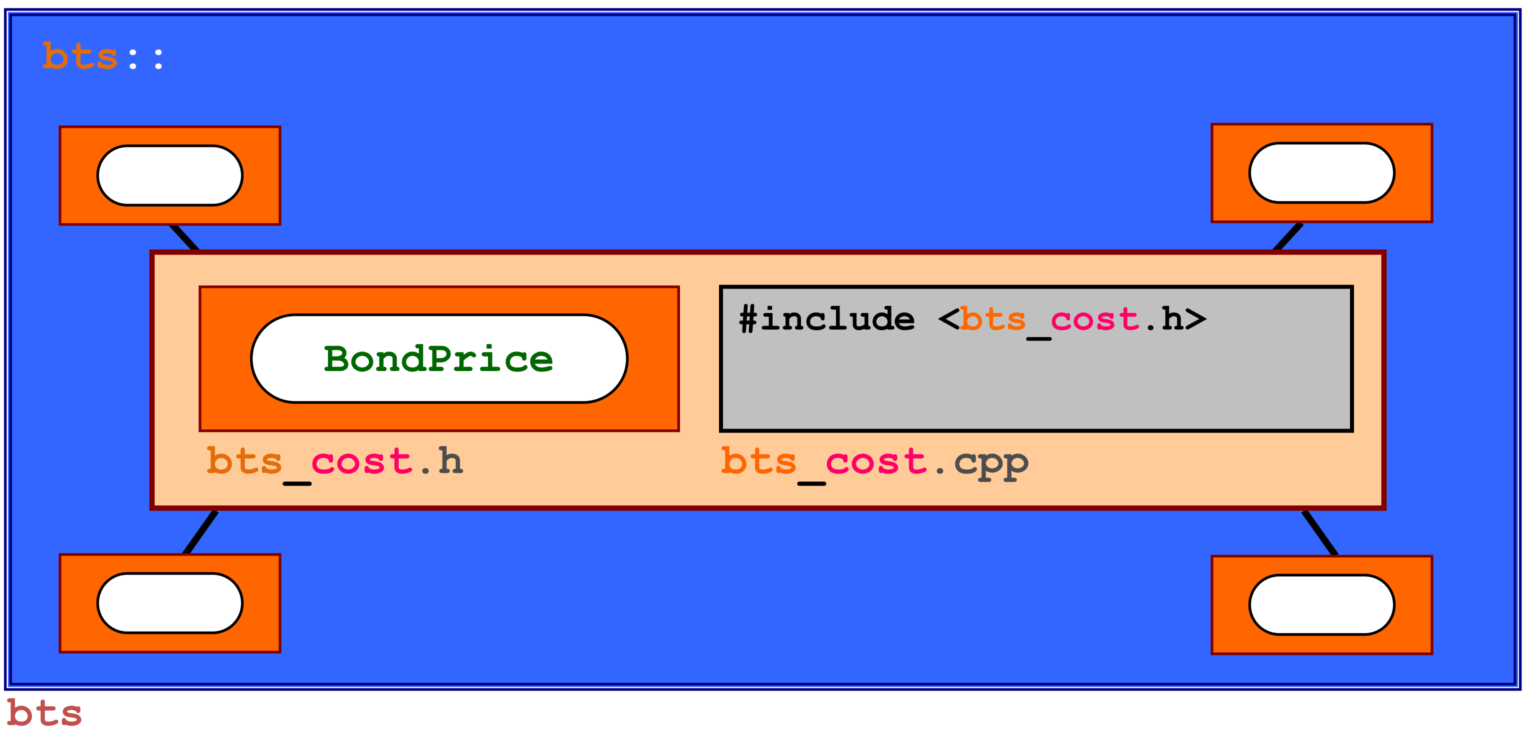
Design Rule

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.

Logical/Physical Name Cohesion

Class name **should** match Component name

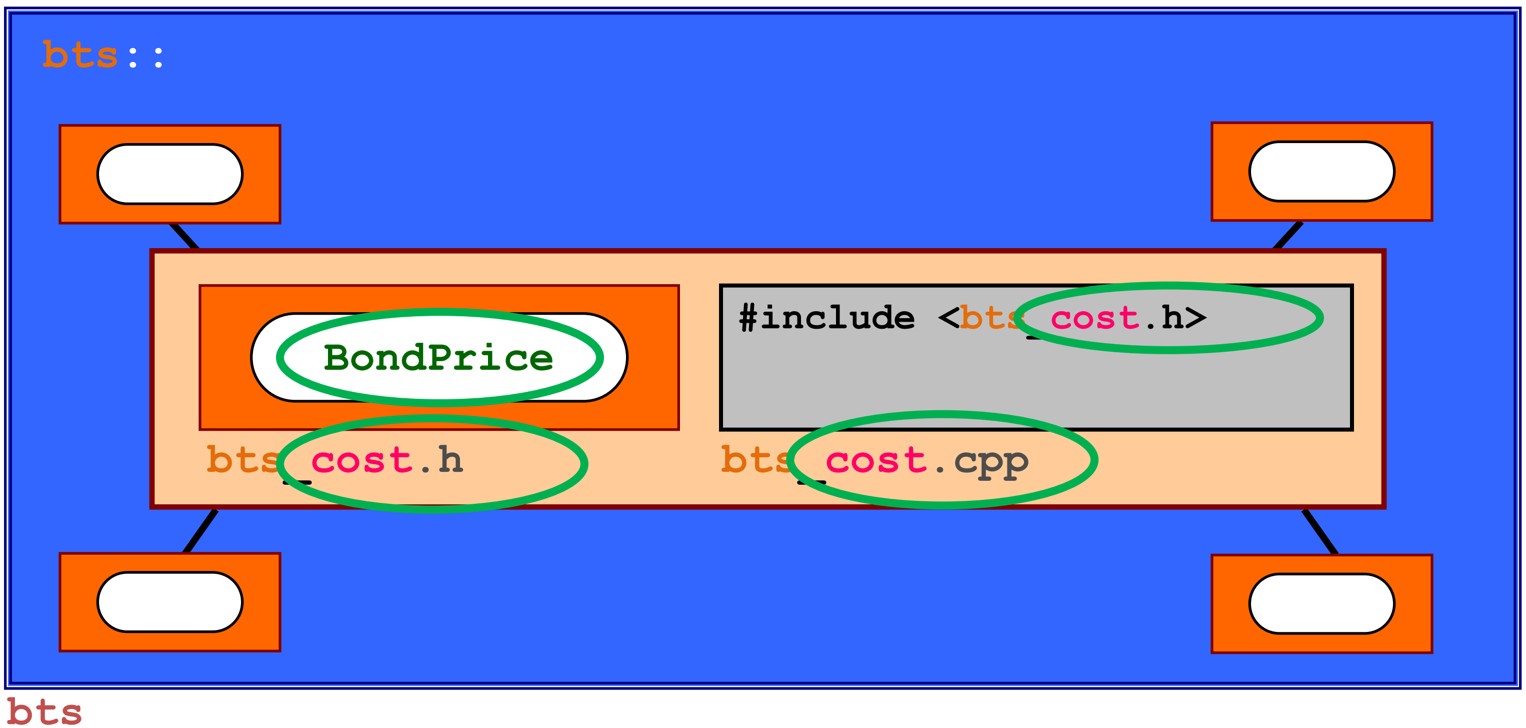
BondPrice ←→ **cost**



Logical/Physical Name Cohesion

Class name **should** match Component name

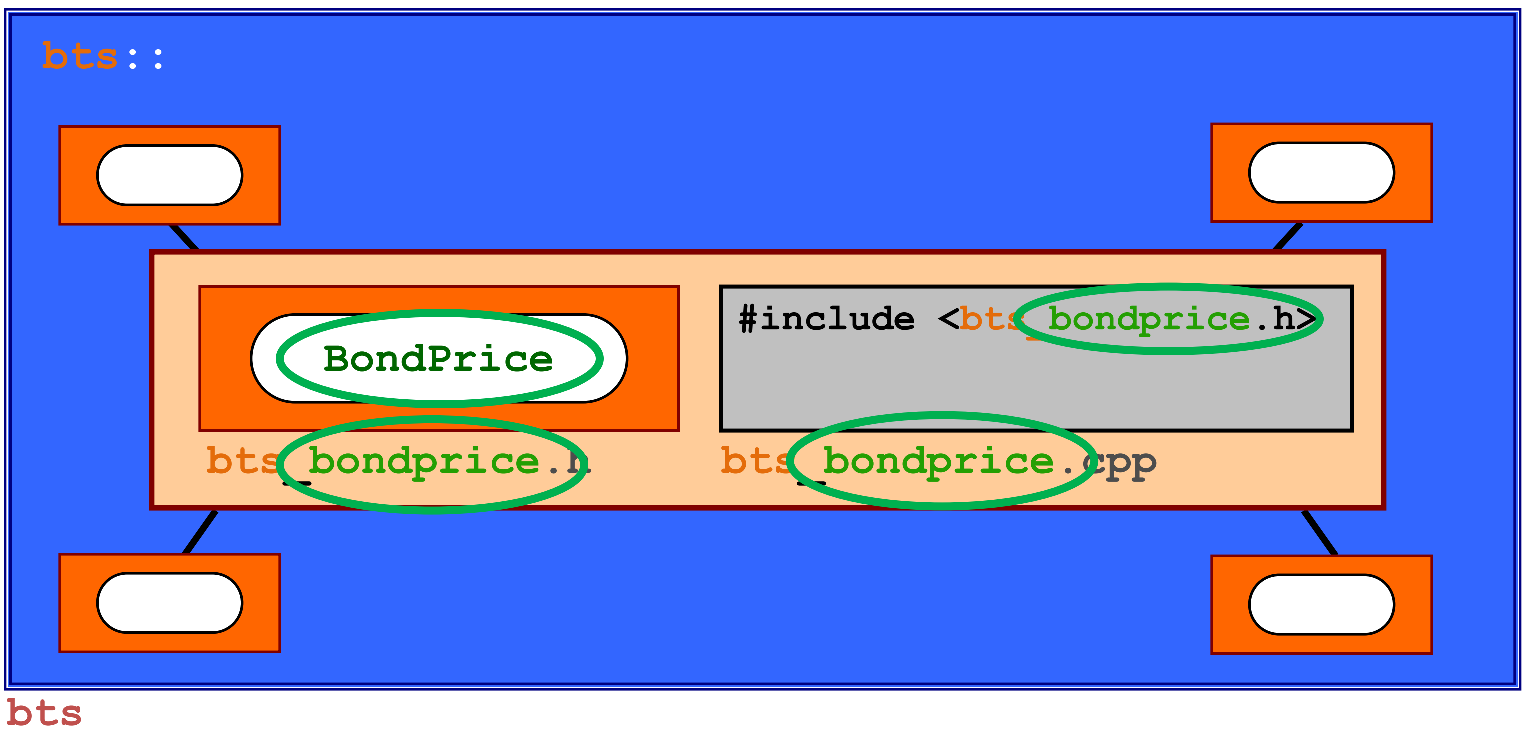
BondPrice ←→ **cost**



Logical/Physical Name Cohesion

Class name **does** match Component name

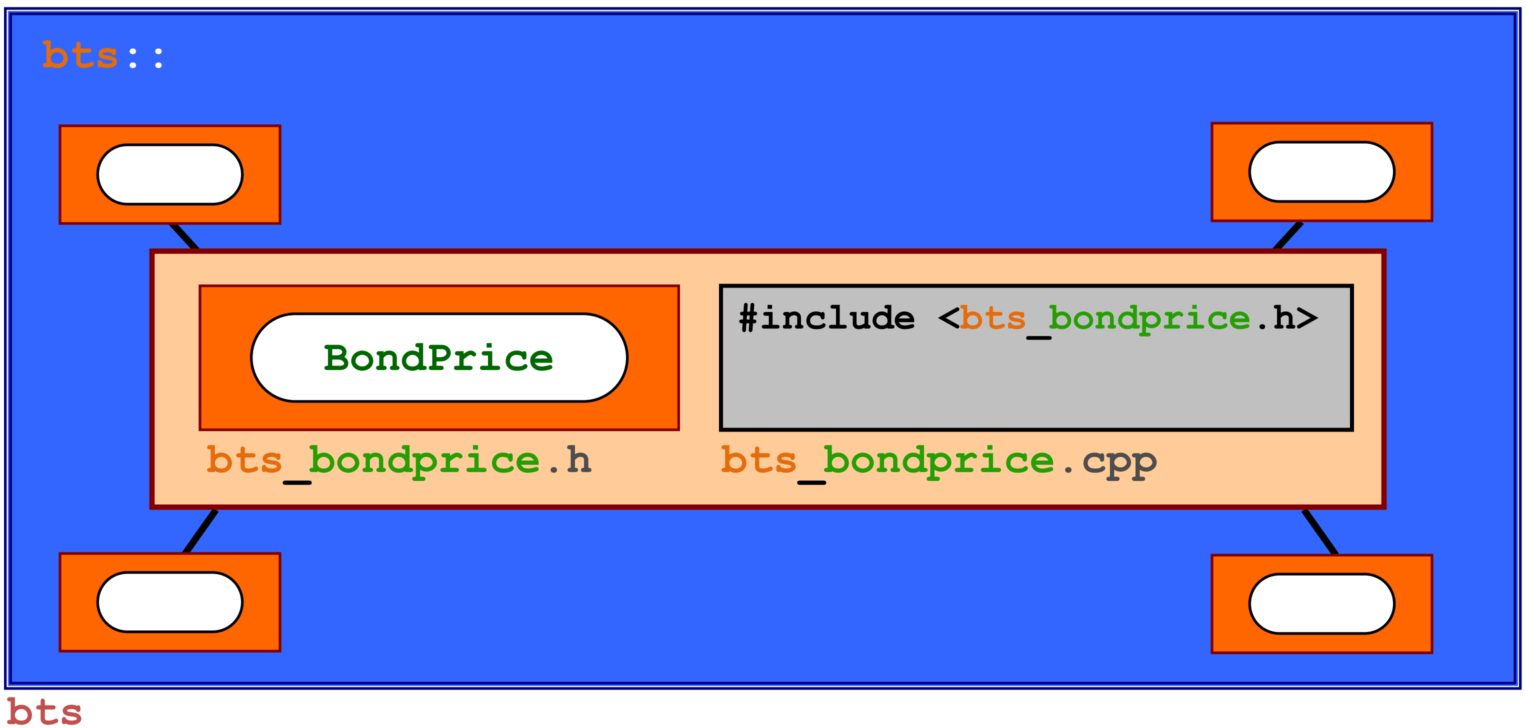
BondPrice ↔ **bondprice**



Logical/Physical Name Cohesion

Class name **does** match Component name

BondPrice ←→ **bondprice**



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 `struct`s, class templates, operators, and certain *aspect* functions (e.g., `swap`).

Logical/Physical Name Cohesion

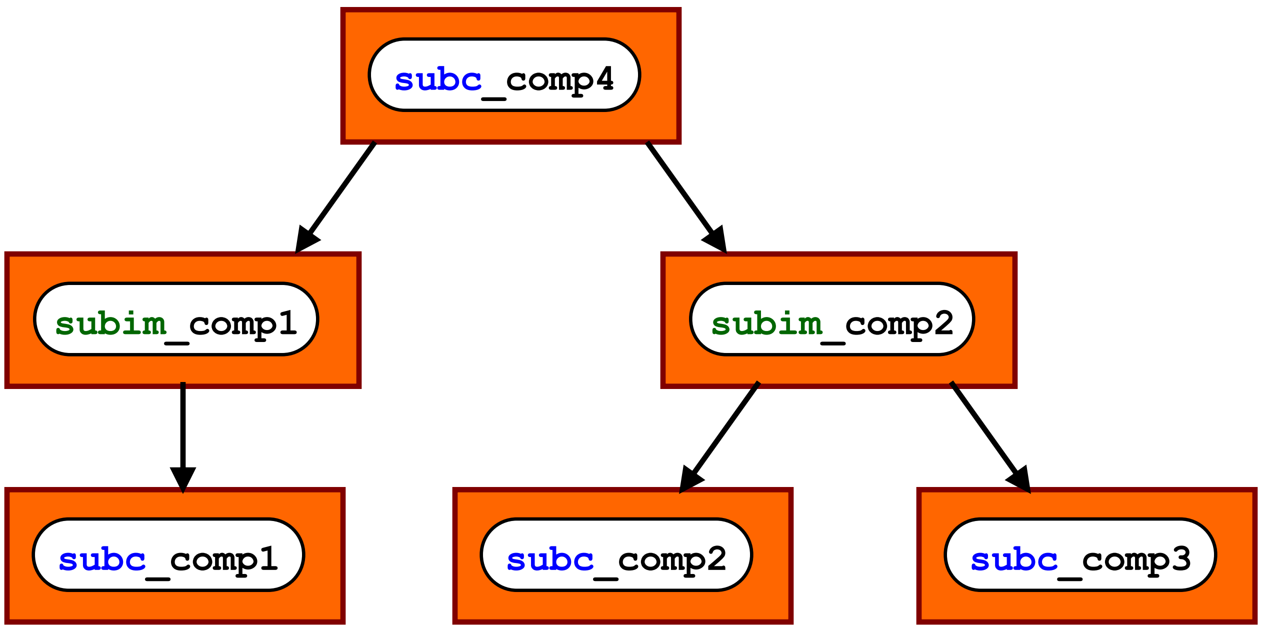
Some more details:

- ❑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 `struct`s, class templates, operators, and certain *aspect* functions (e.g., `swap`).

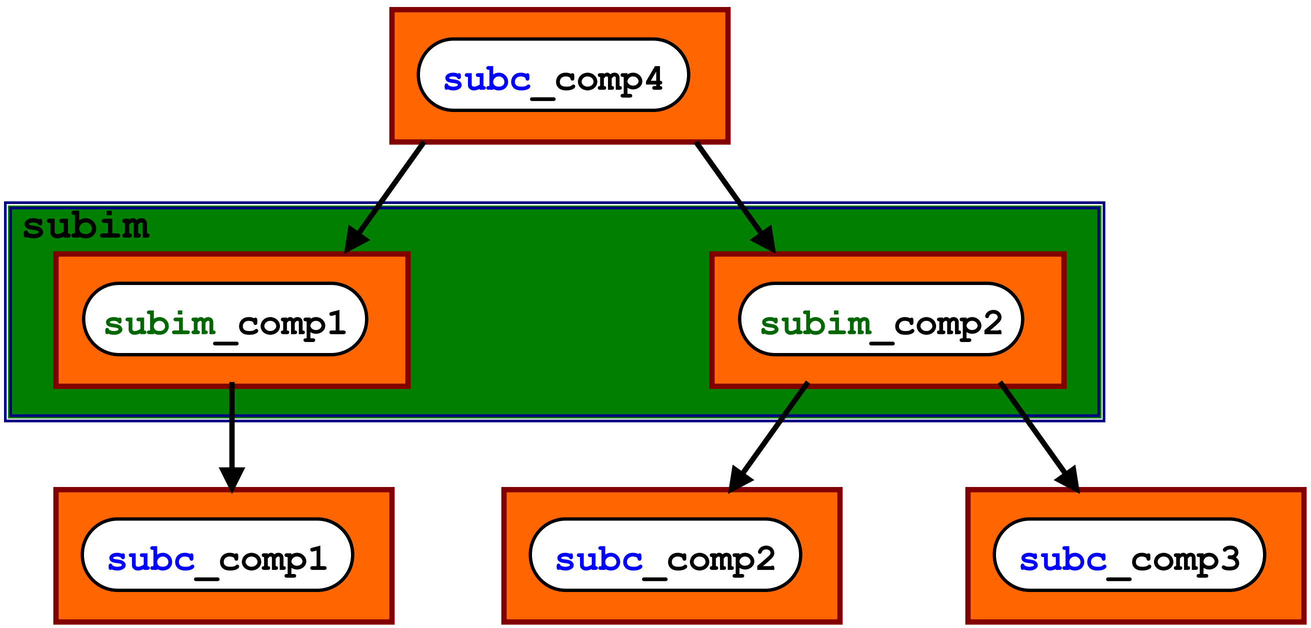
Logical/Physical Name Cohesion

Package naming is more than just a convention:



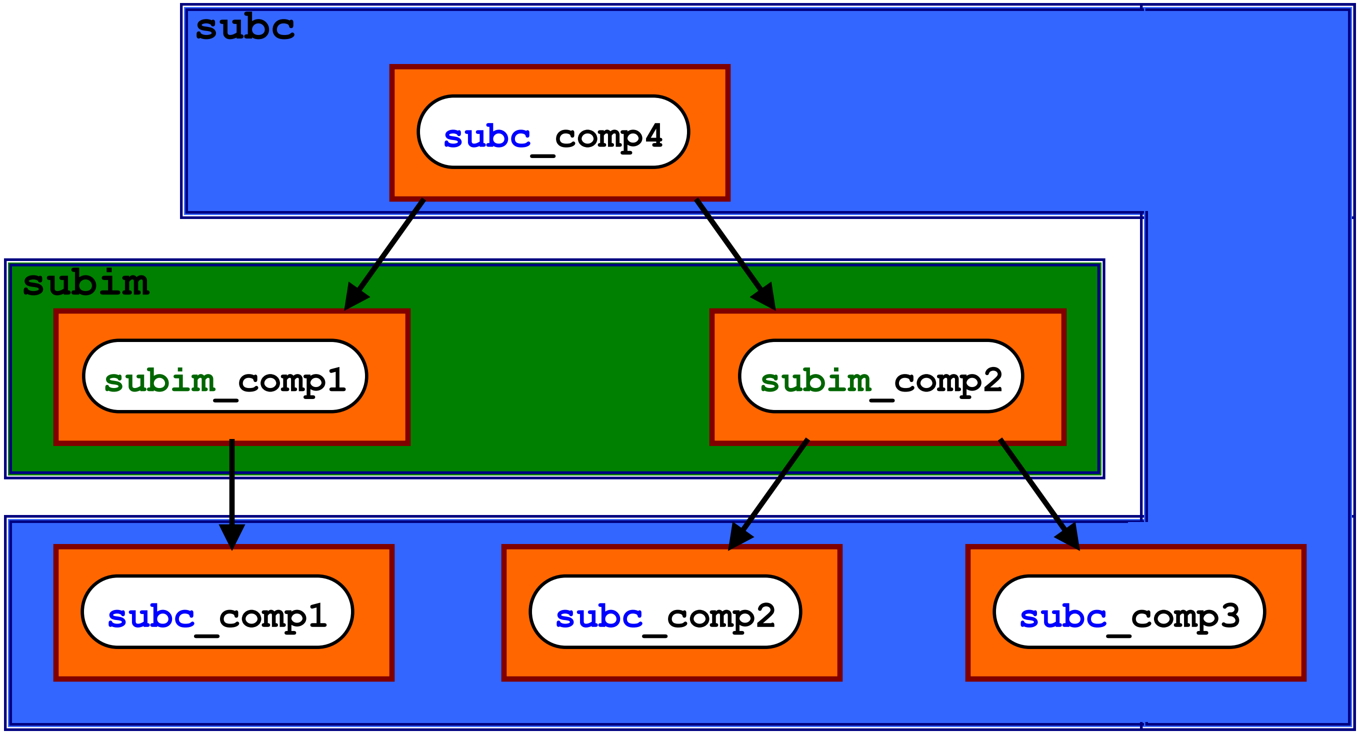
Logical/Physical Name Cohesion

Package naming is more than just a convention:



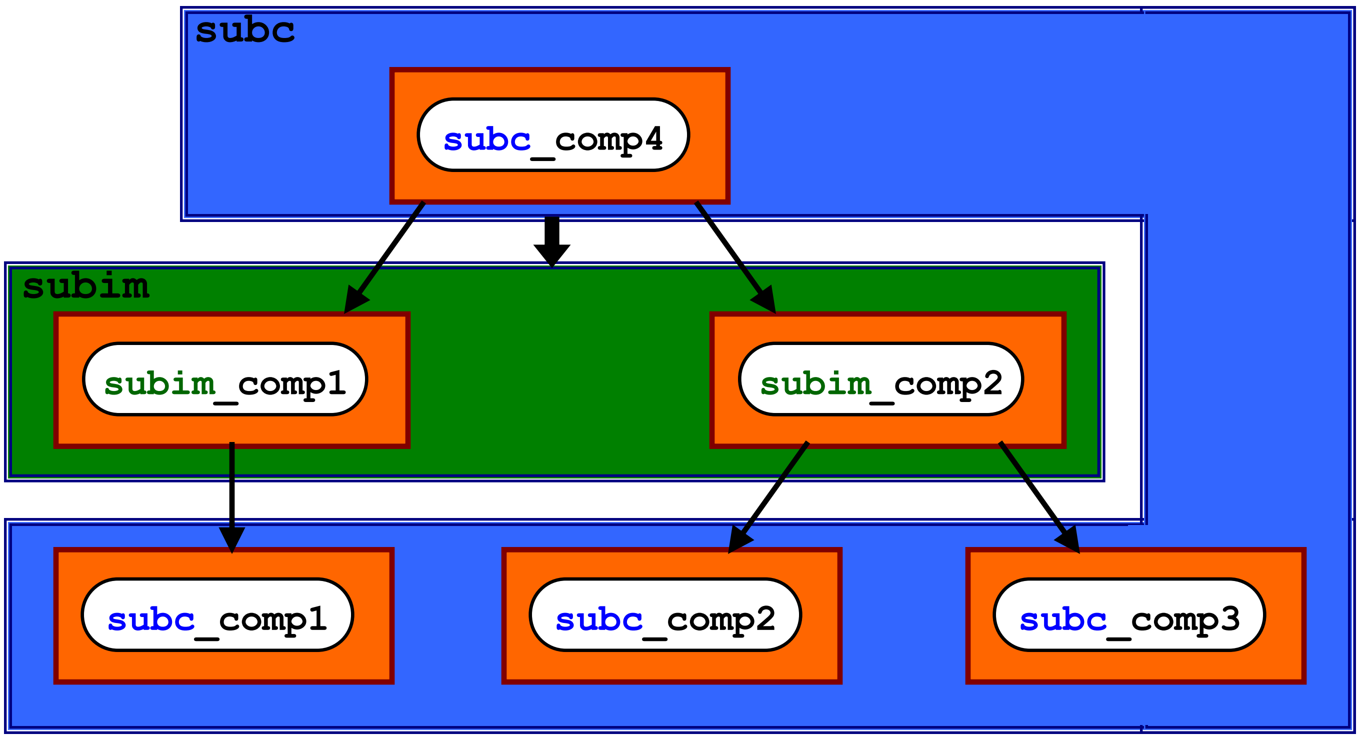
Logical/Physical Name Cohesion

Package naming is more than just a convention:



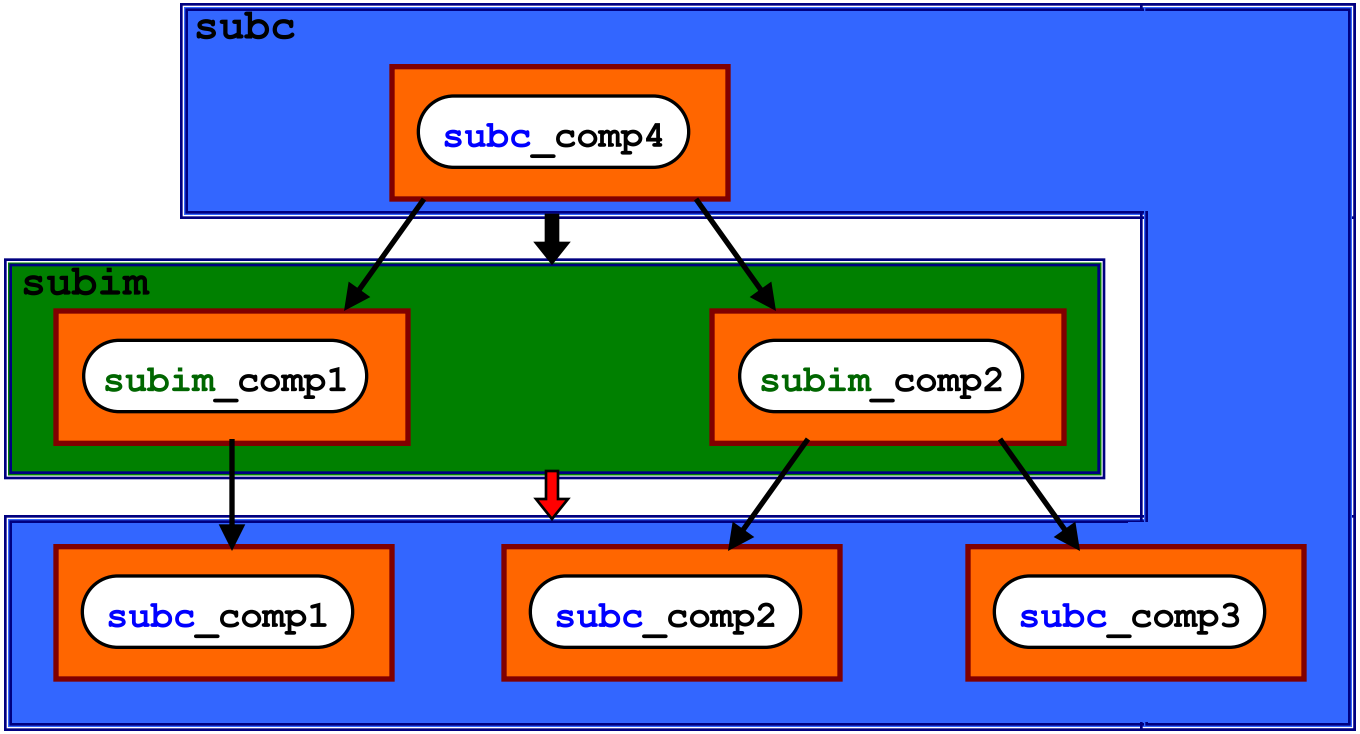
Logical/Physical Name Cohesion

Package naming is more than just a convention:



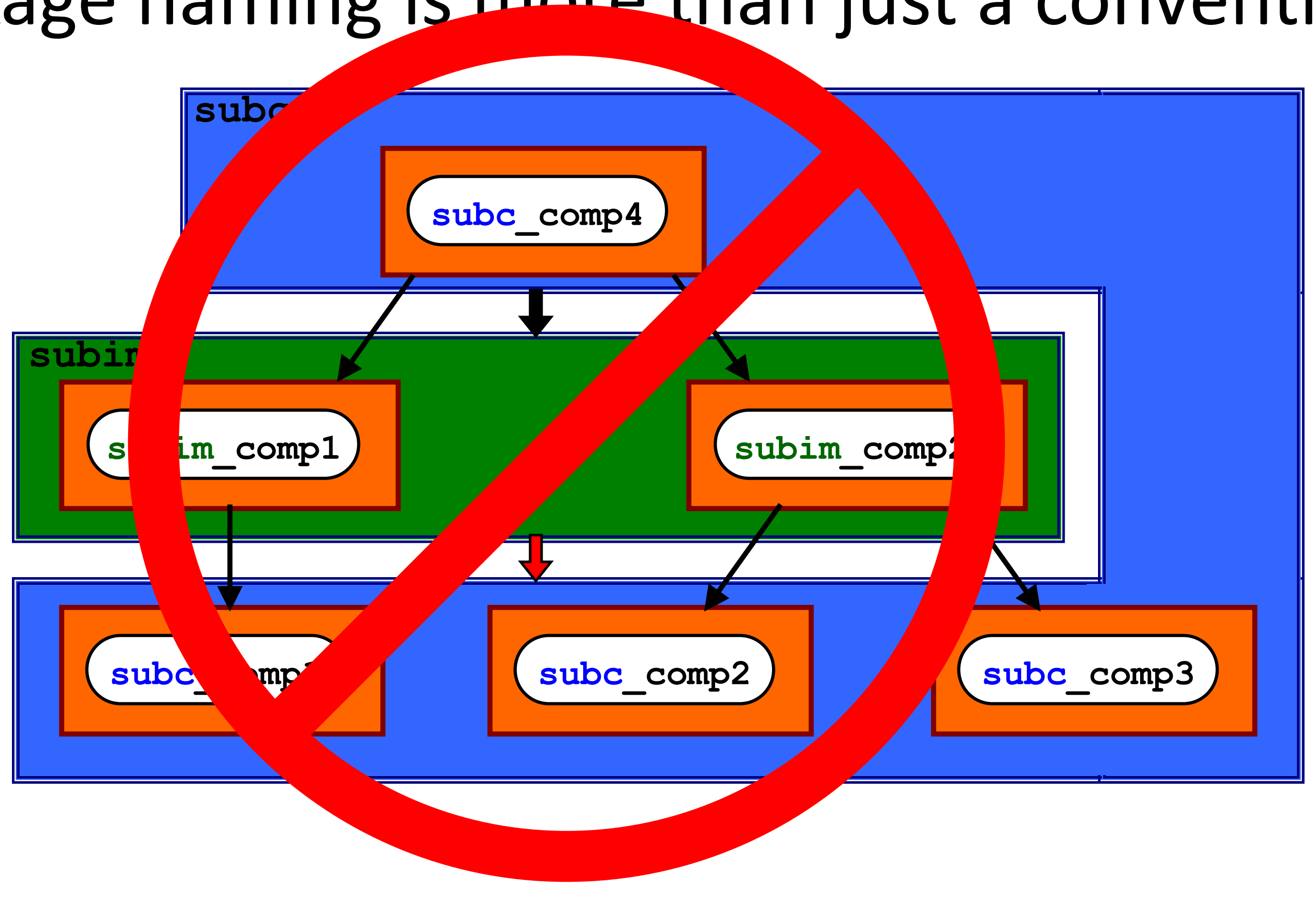
Logical/Physical Name Cohesion

Package naming is more than just a convention:



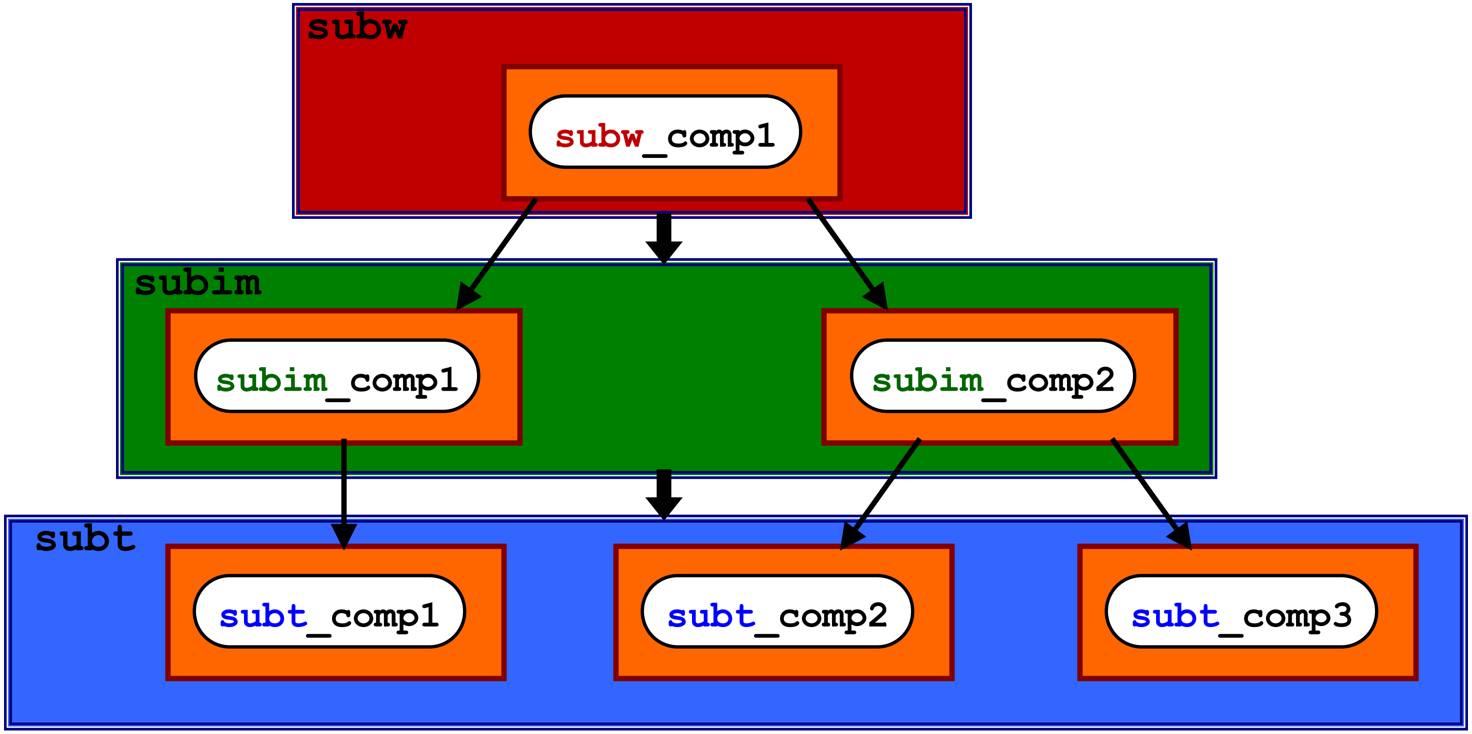
Logical/Physical Name Cohesion

Package naming is more than just a convention:



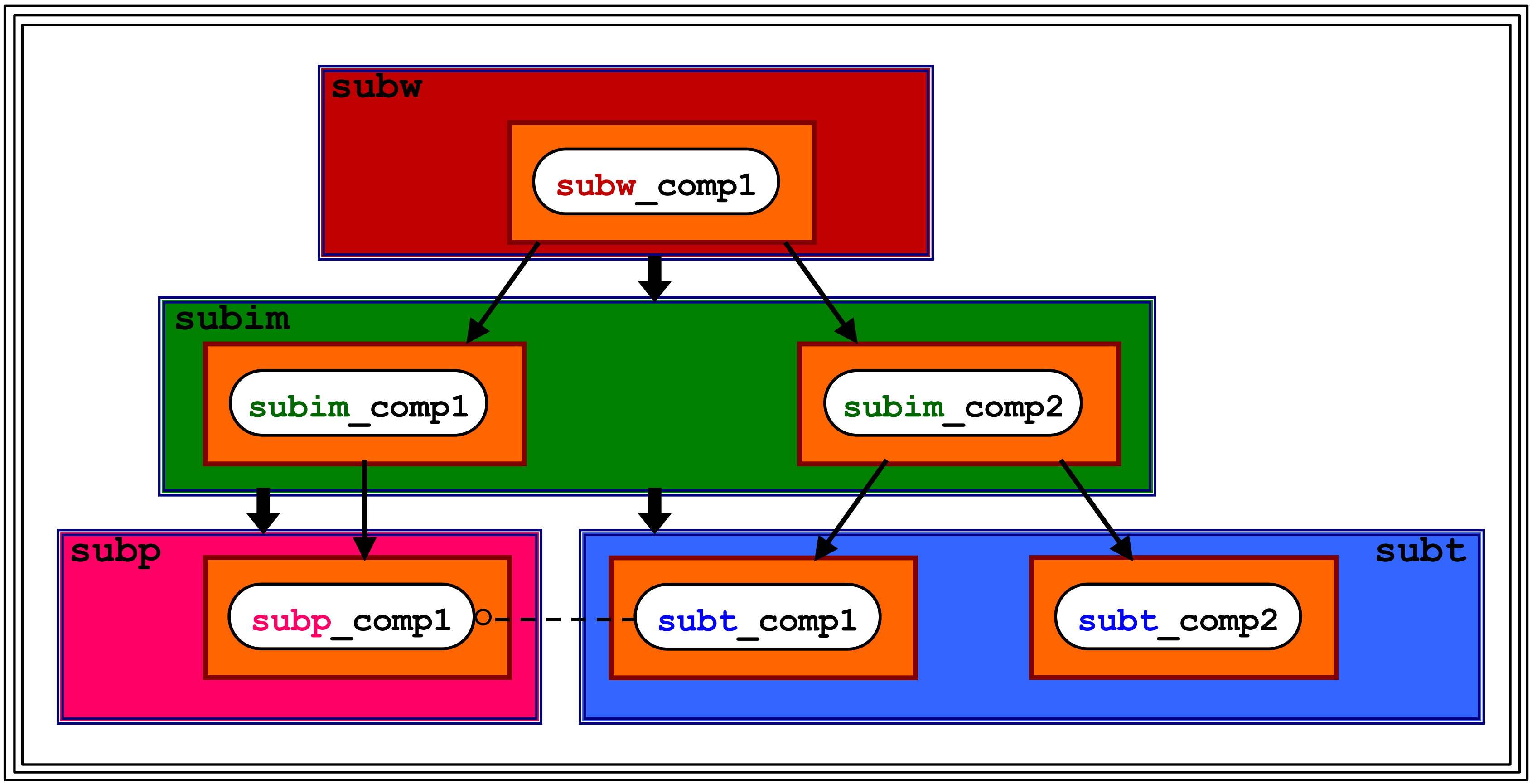
Logical/Physical Name Cohesion

Package naming is more than just a convention:



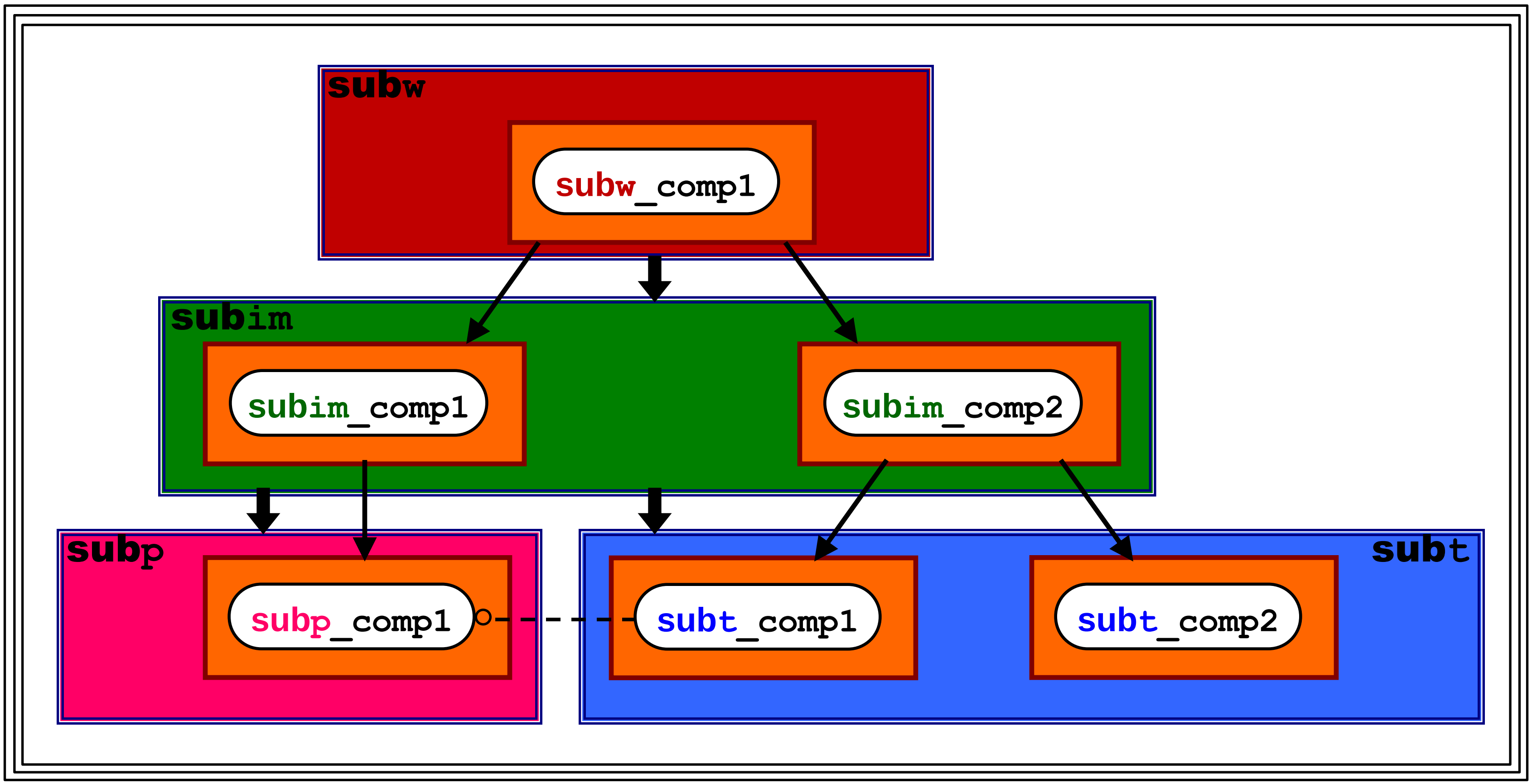
Logical/Physical Name Cohesion

Package Group



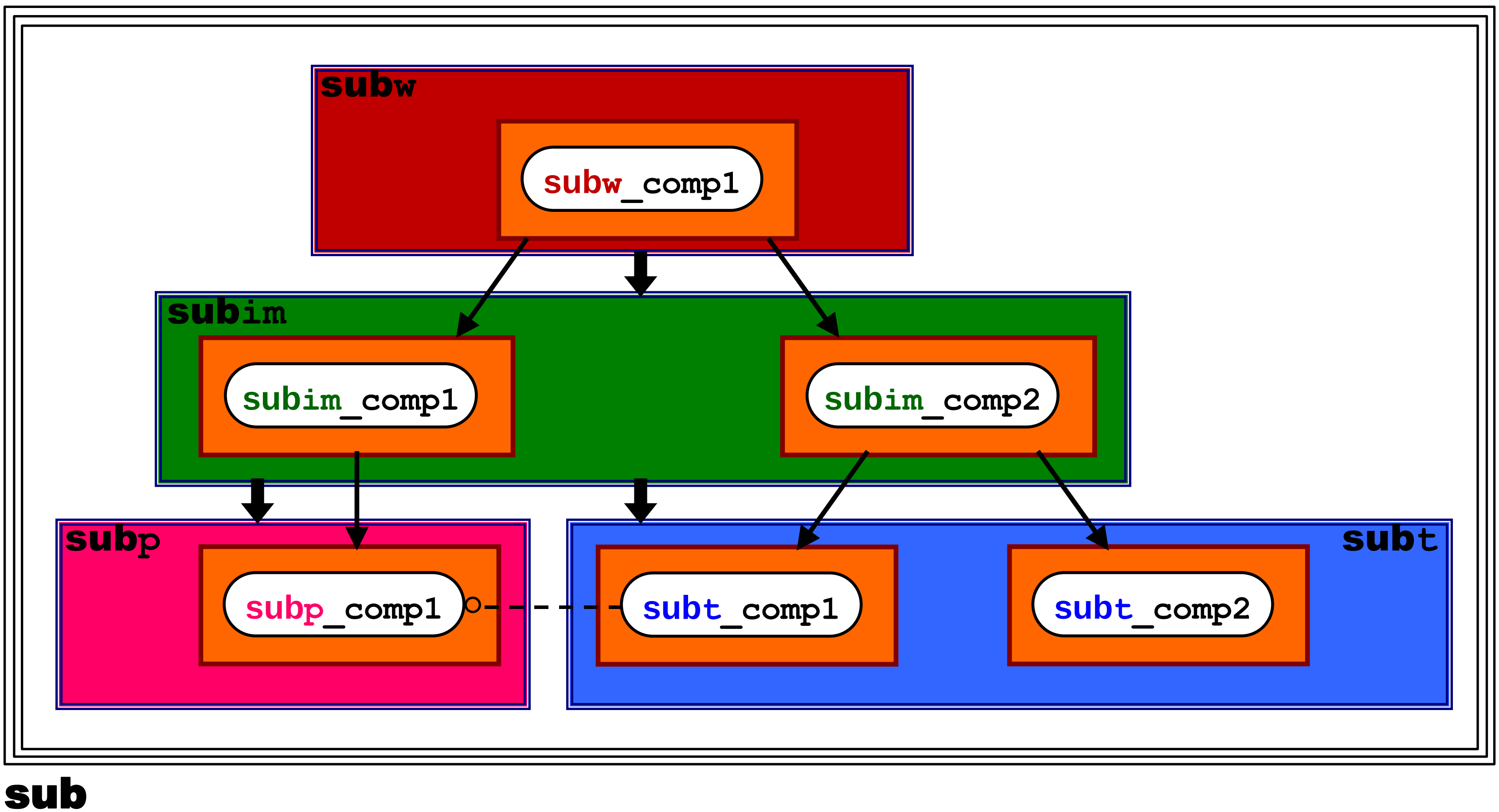
Logical/Physical Name Cohesion

Package Group



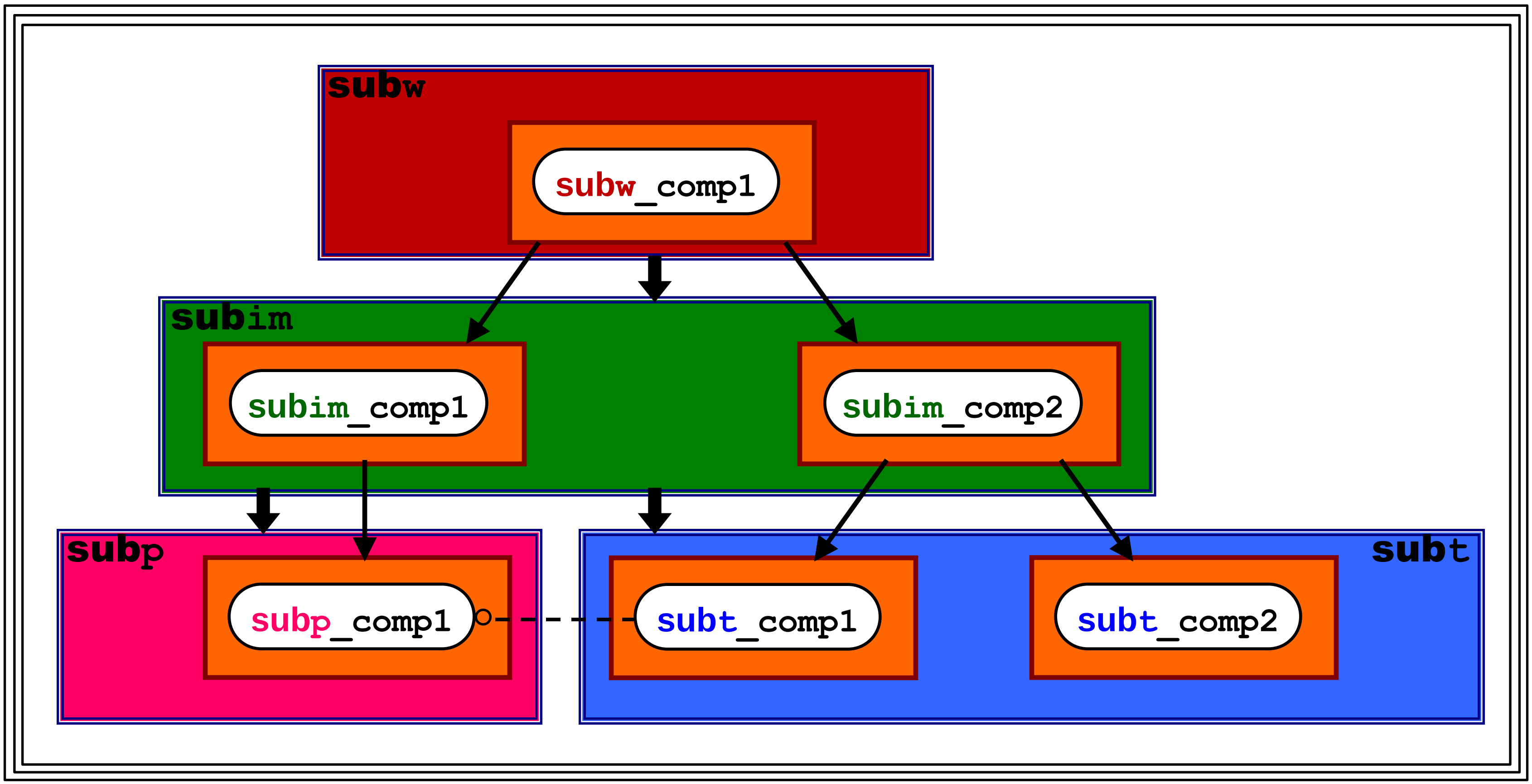
Logical/Physical Name Cohesion

Package Group



Logical/Physical Name Cohesion

Package Group



sub Exactly Three Characters

Logical/Physical Name Cohesion

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

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`

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`

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`

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`

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**

Logical/Physical Name Cohesion

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

```
          libraries
         /    |    \
             bdl            Package Groups
         /    |    \
     ____bdlt____           Packages
     ...
     bdlt_date.h            Components
     bdlt_date.cpp
     bdlt_date.t.cpp
```

Logical/Physical Name Cohesion

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

```
            libraries
           /    |    \
               bdl             Package Groups
           /    |     \
      ____bdlt_____            Packages
      ...
      bdlt_date.h              Components
      bdlt_date.cpp
      bdlt_date.t.cpp
```

Logical/Physical Name Cohesion

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

```
            libraries
            /   |   \
               bdl             Package Groups
            /   |    \
      _____bdlt_____          Packages
      ...
      bdlt_date.h             Components
      bdlt_date.cpp
      bdlt_date.t.cpp
```

Logical/Physical Name Cohesion

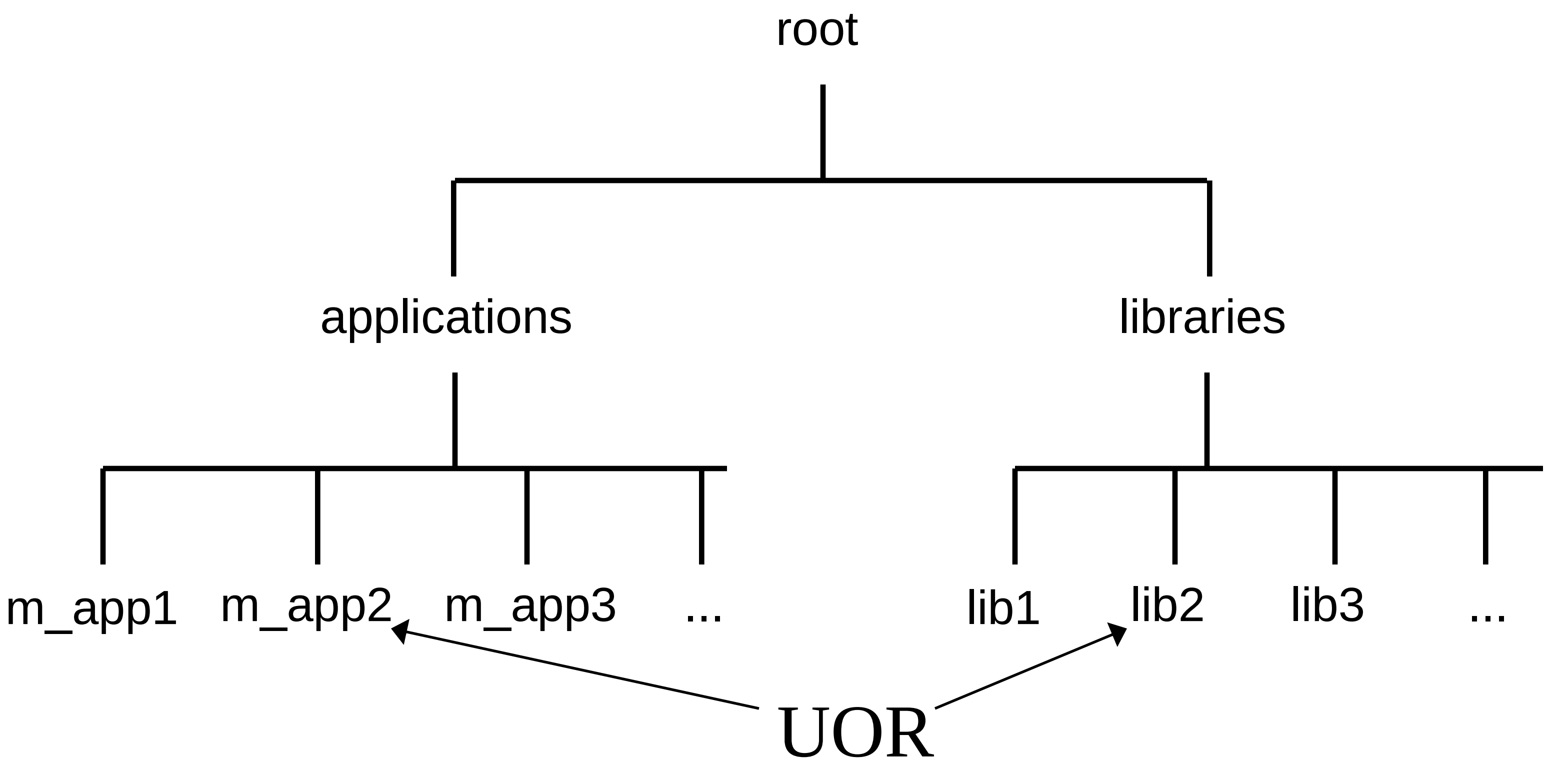
```
...
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
...
             libraries
             /   |   \
                bdl            Package Groups
             /   |    \
        ____bdlt____           Packages
        ...
        bdlt_date.h            Components
        bdlt_date.cpp
        bdlt_date.t.cpp
```

Logical/Physical Name Cohesion

```
...
bool flag = bdlt::Date::isValidYMD(1959, 3, 8);
...
            libraries
             /  |  \
               bdl            Package Groups
             /  |   \
        ____bdlt____          Packages
        ...
        bdlt_date.h           Components
        bdlt_date.cpp
        bdlt_date.t.cpp
```

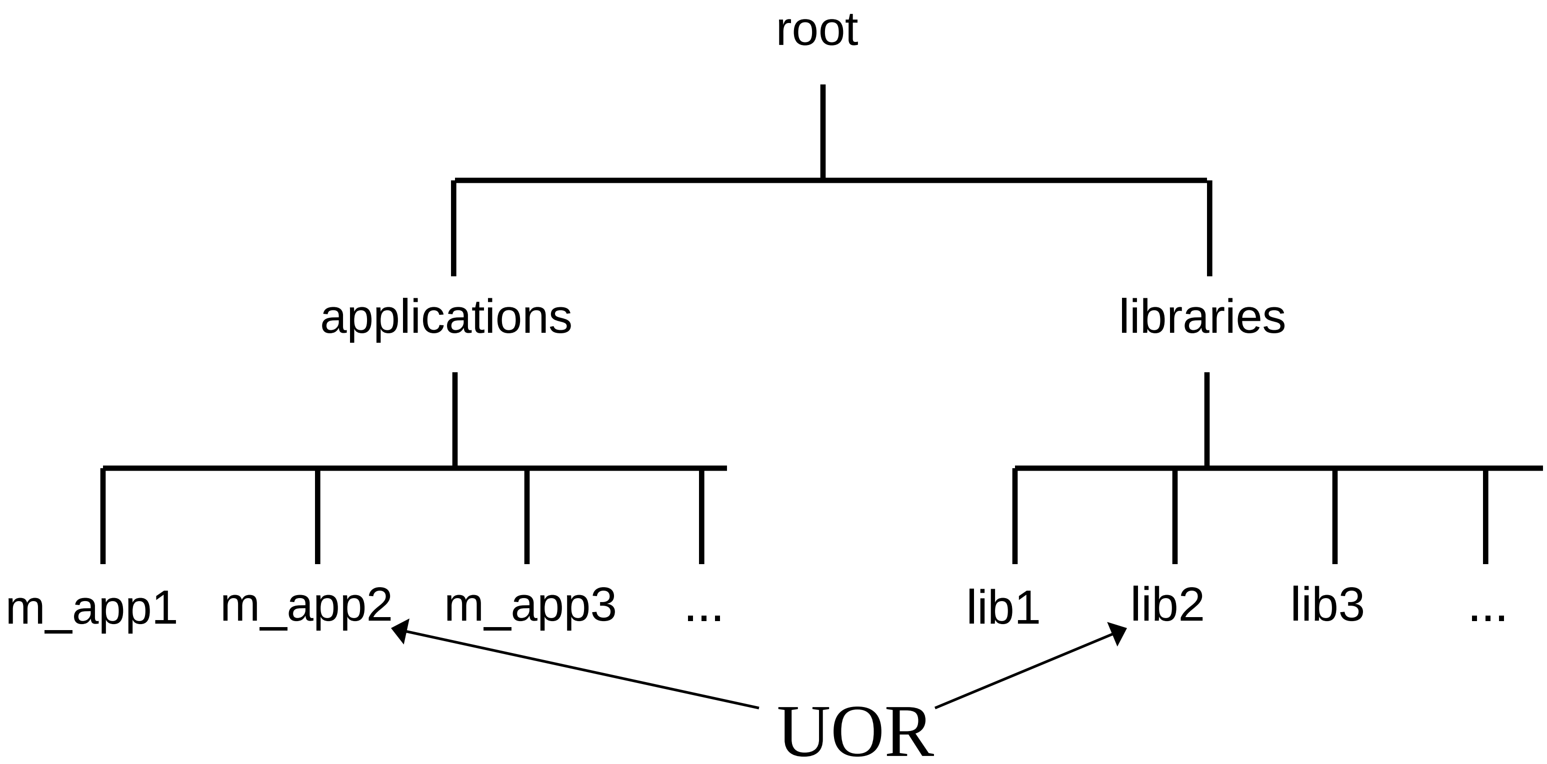
1. Process & Architecture

Unit Of Release

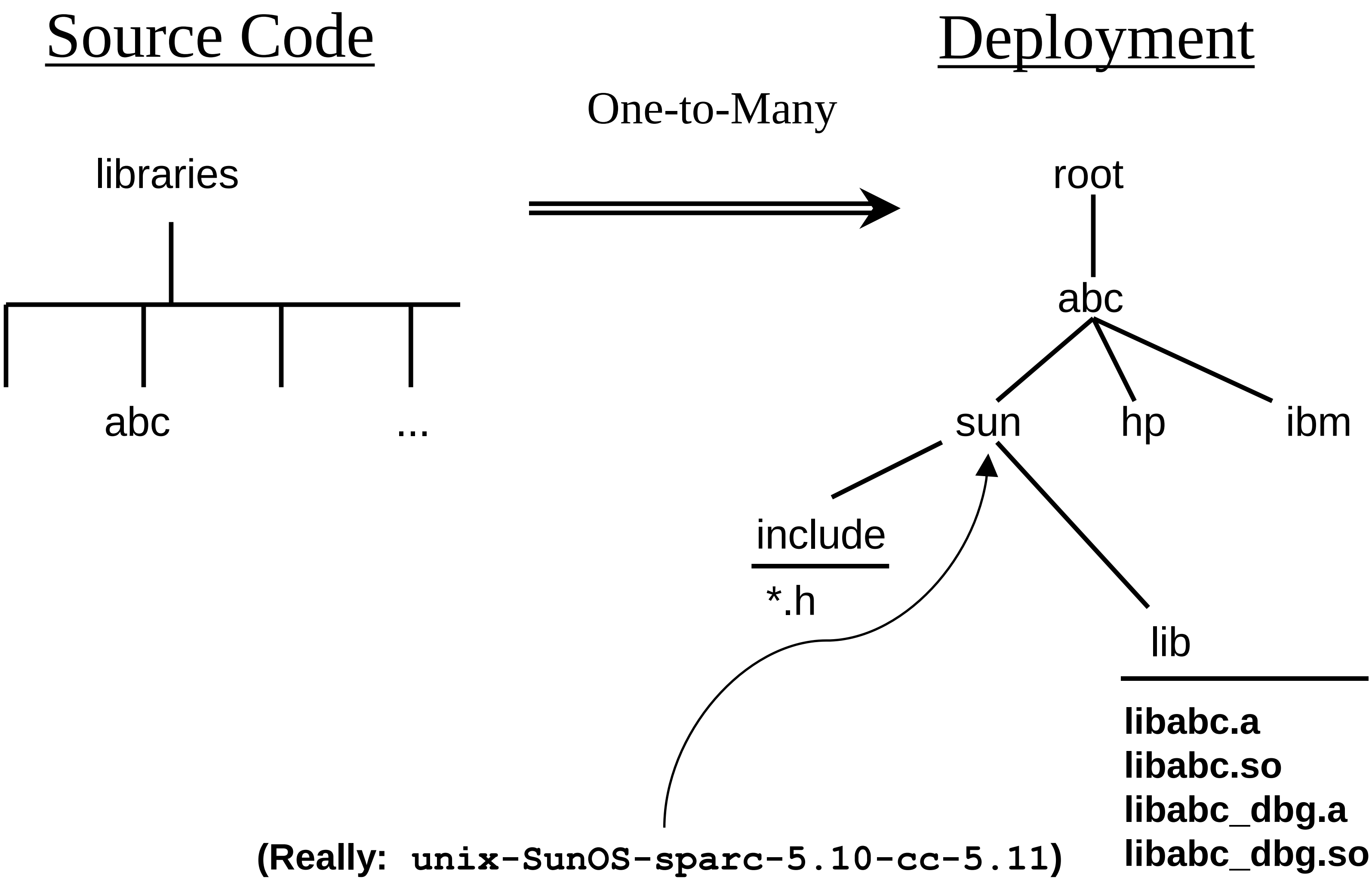


Unit Of Release

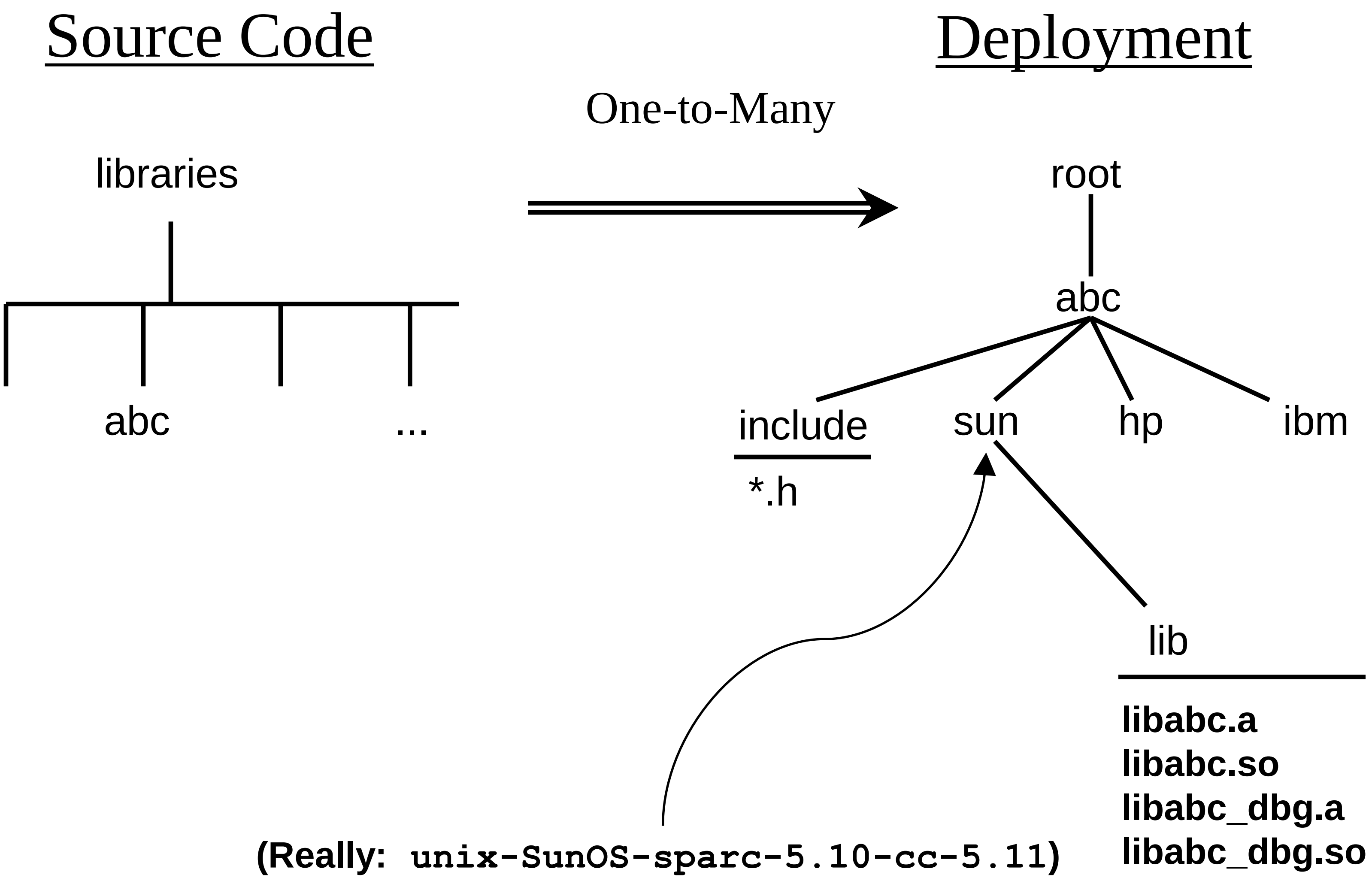
Package or Package Group



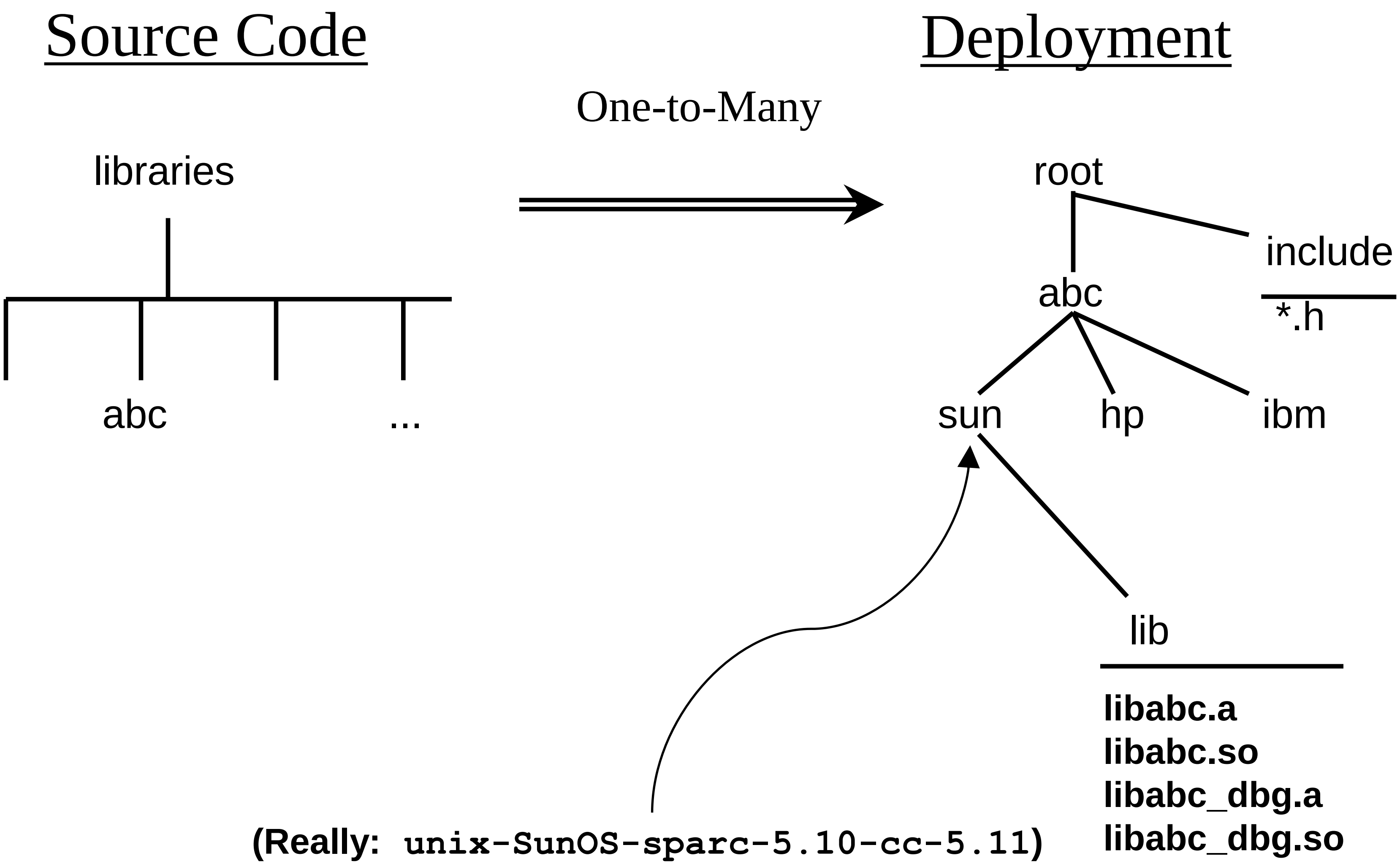
Development vs. Deployment



Development vs. Deployment



Development vs. Deployment



Designing with Dependency in Mind

Good Physical Design…

Designing with Dependency in Mind

Good Physical Design…

✓Is ***not*** an afterthought.

Designing with Dependency in Mind

Good Physical Design…

✓Is ***not*** an afterthought.

✓***Is*** an integral part of logical design.

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.

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.

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

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

The *Value* of a "Value"

Getting Started:

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

The *Value* of a "Value"

Getting Started:

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

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.

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.

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();
};
```

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();
};
```

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();
};
```

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;
};
```

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();
};
```

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();
};
```

So, what do we mean by "value"?

Salient Attributes

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

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.

So, what do we mean by “value”?

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

```cpp
class Time {
    int d_mSeconds;



  public:
    // …
    int hour();
    int minute();
    int second();
    int millisecond();
};
```

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

So, what do we mean by "value"?

Value:

So, what do we mean by “value”?

Value:

- An “interpretation” of object state –

So, what do we mean by "value"?

Value:

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

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.

What are "Salient Attributes"?

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);
    // ...
};
```

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);
    // ...
};
```

What are “Salient Attributes”?

Consider `std::vector<int>`:

What are its *salient attributes*?

What are "Salient Attributes"?

Consider `std::vector<int>`:

What are its *salient attributes*?

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

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.

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()`?

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?

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).

Value-Semantic Properties

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

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?
```

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?
```

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?
```

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

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

Value-Semantic Properties

Note that two *distinct* objects **a** and **b** of type T that have *the same* ***value*** *might* <u>*not*</u> exhibit "the same" ***observable behavior***.

HOWEVER

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

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*)

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***!

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*!**

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*!**

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*!**

Value-Semantic Properties

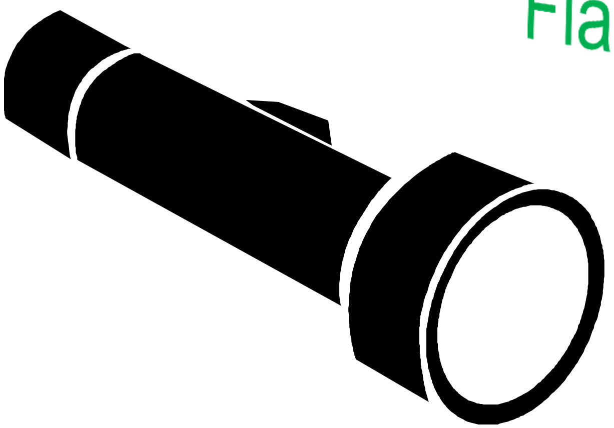
There is a lot more to this story!

Deciding what is (not) *salient* is surprisingly important.

SUBTLE ESSENTIAL PROPERTY OF VALUE

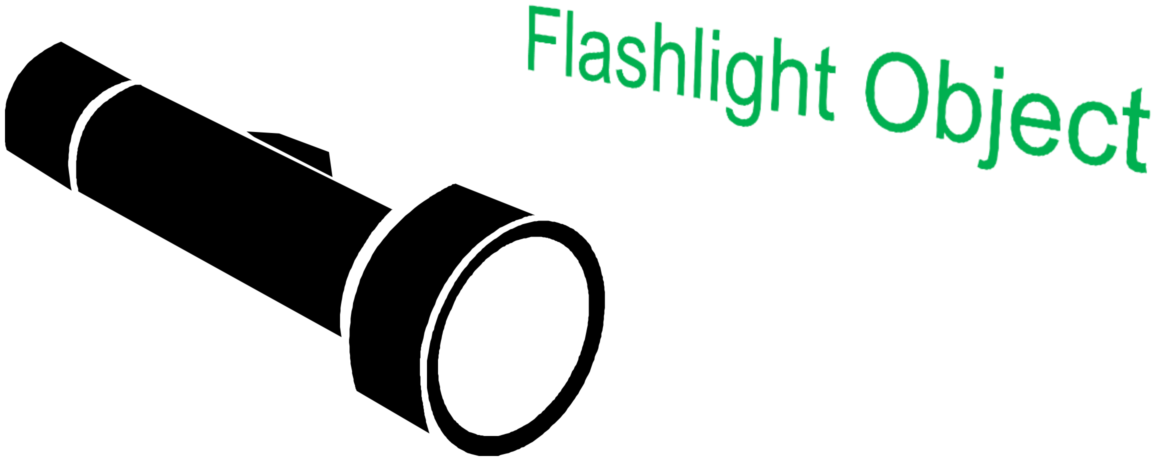
1. If **a** and **b** initially have the *same value*, and
2. the same operation is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will again have the *same value***!

Does **state** *always* imply a "**value**"?

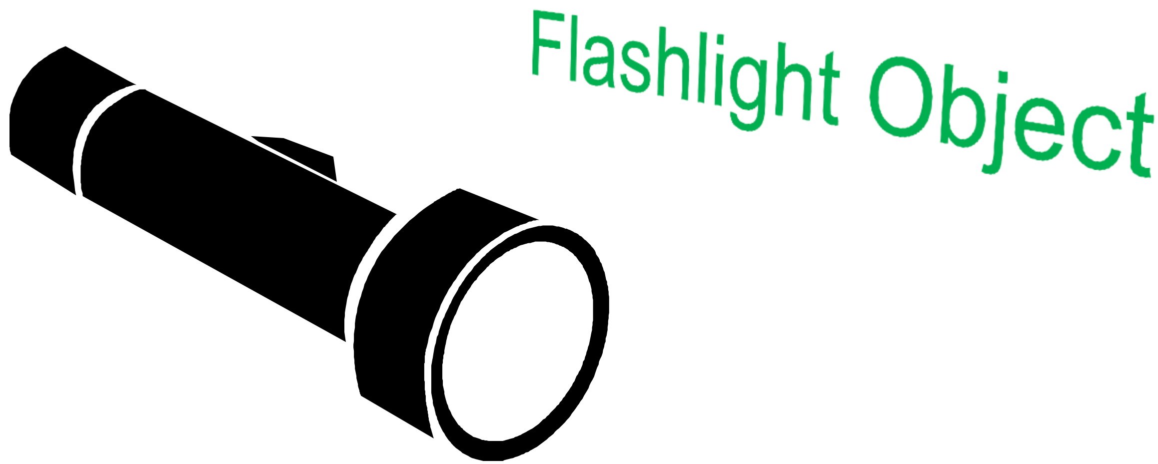


Does **state** *always* imply a “**value**”?

Does **state** *always* imply a "**value**"?

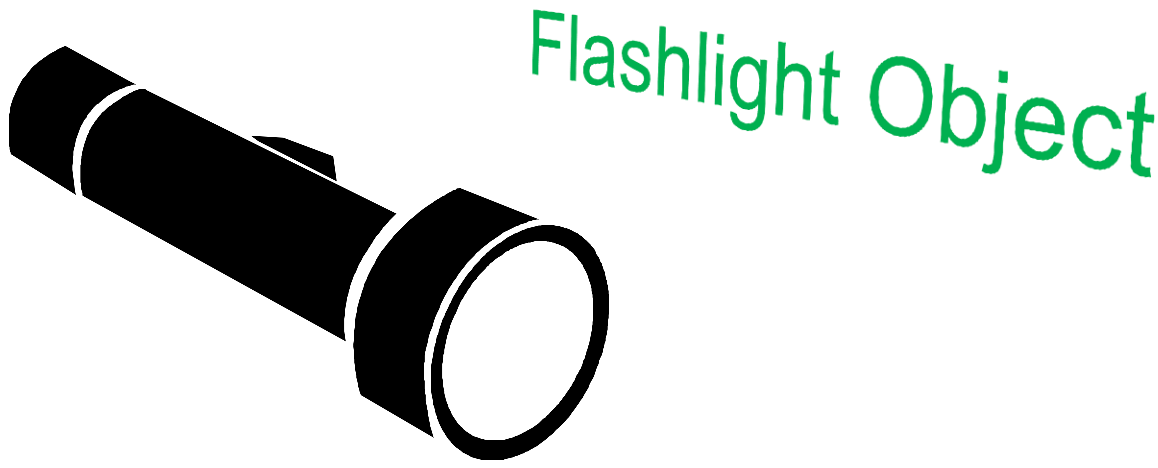


Does **state** *always* imply a "**value**"?



What is its state?

Does **state** *always* imply a "**value**"?

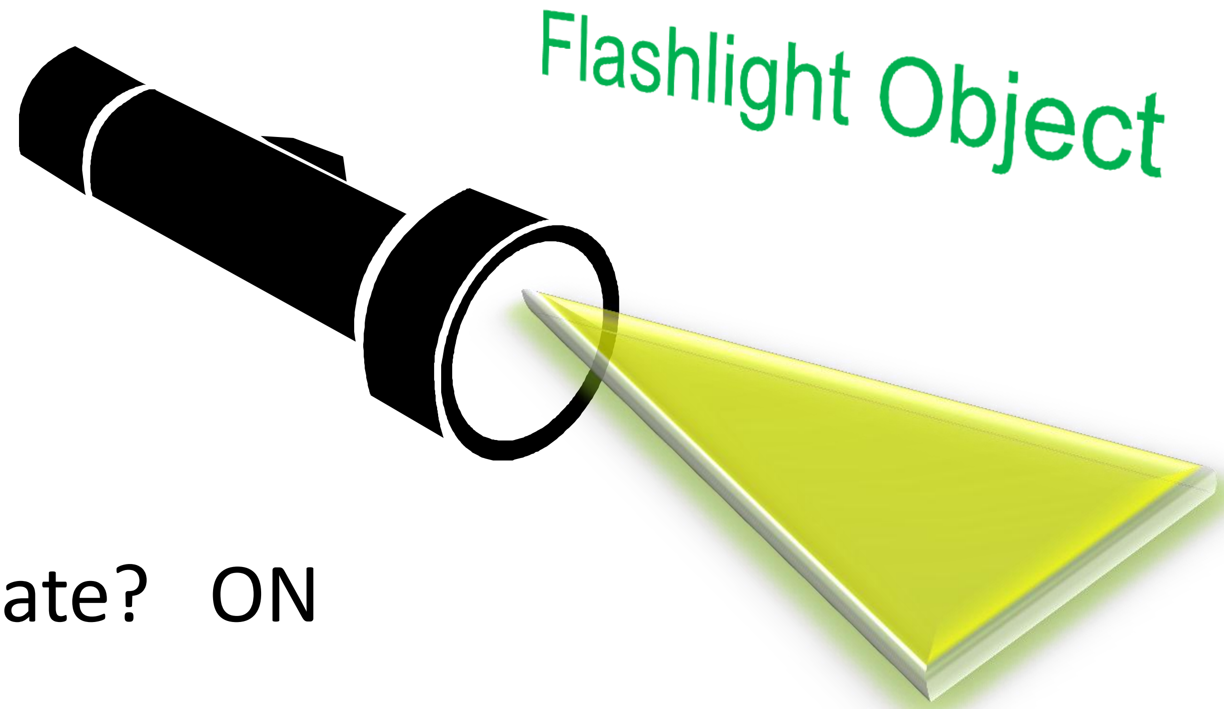


What is its state? OFF

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

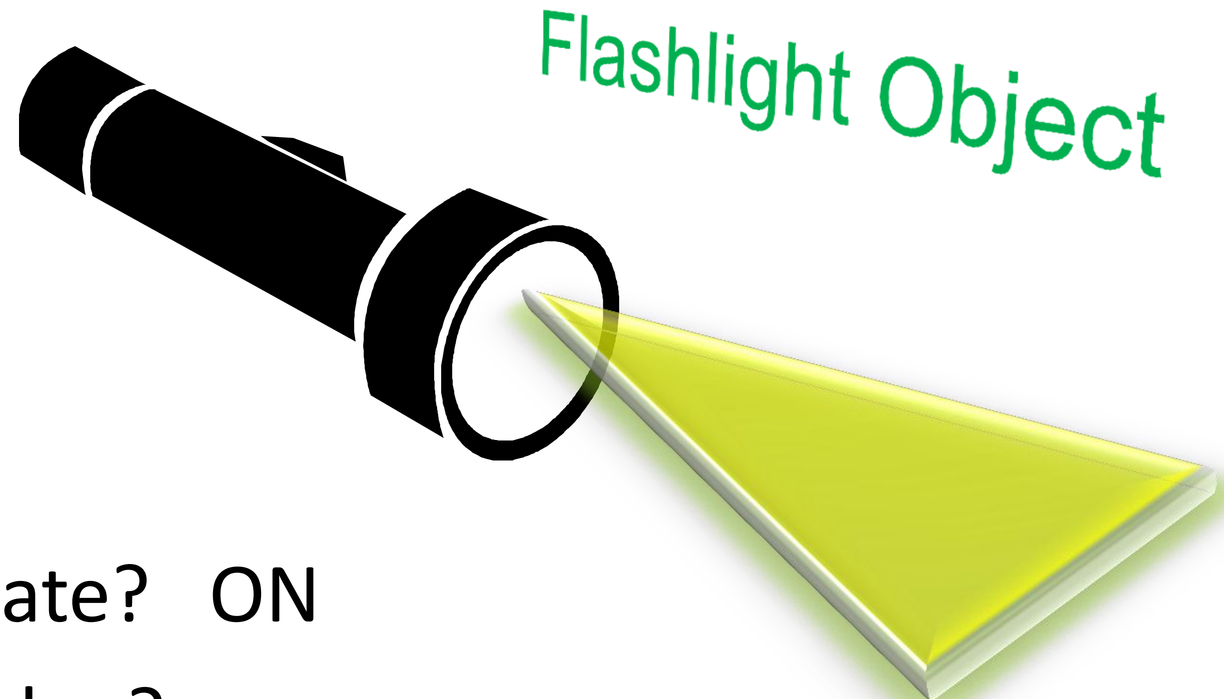
What is its state?

Does **state** *always* imply a "**value**"?



What is its state? ON

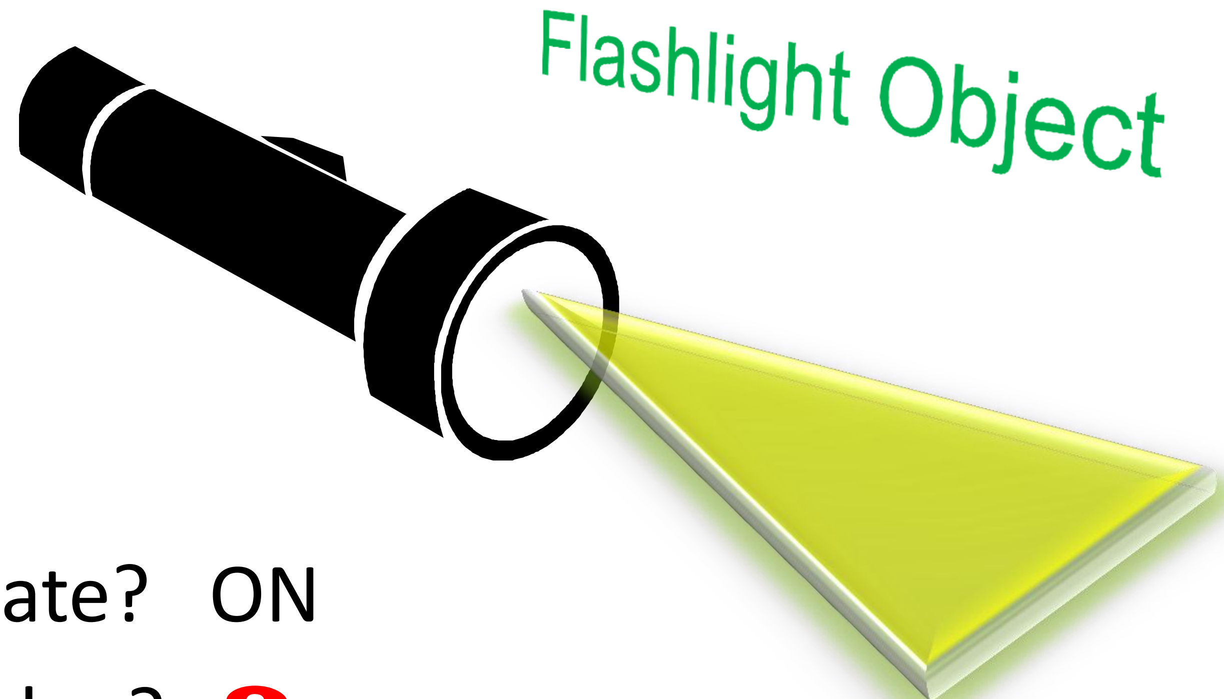
Does **state** *always* imply a "**value**"?



What is its state? ON

What is its value?

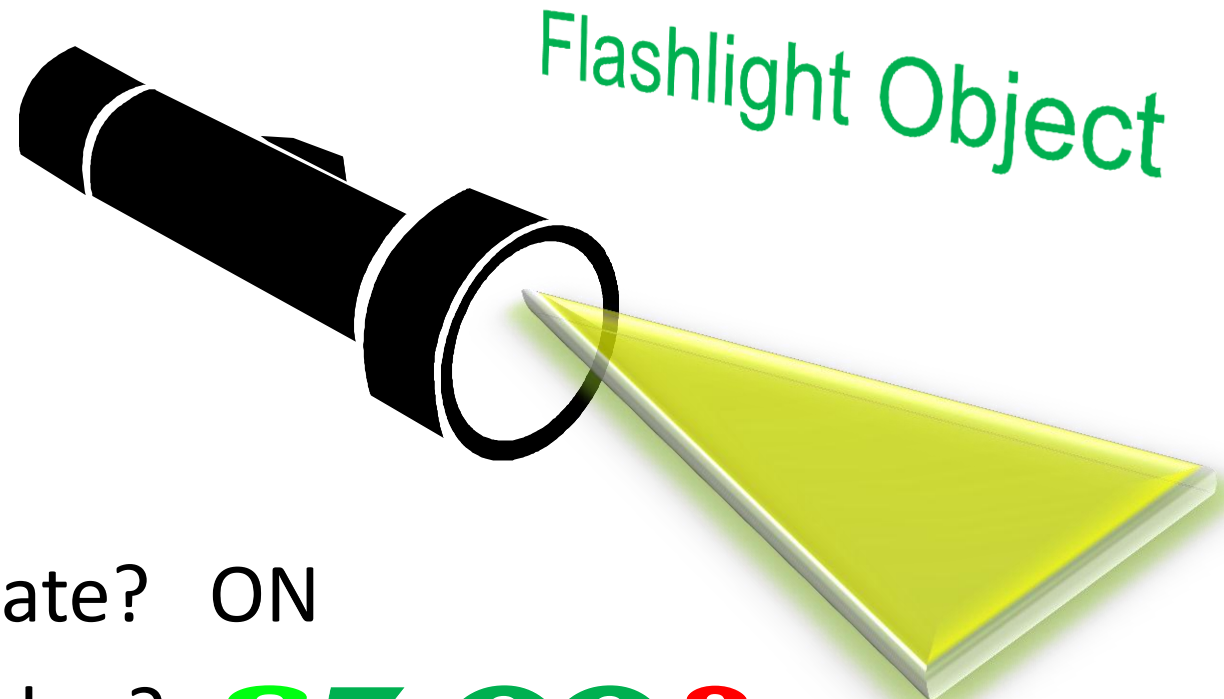
Does **state** *always* imply a "**value**"?



What is its state? ON

What is its value? ?

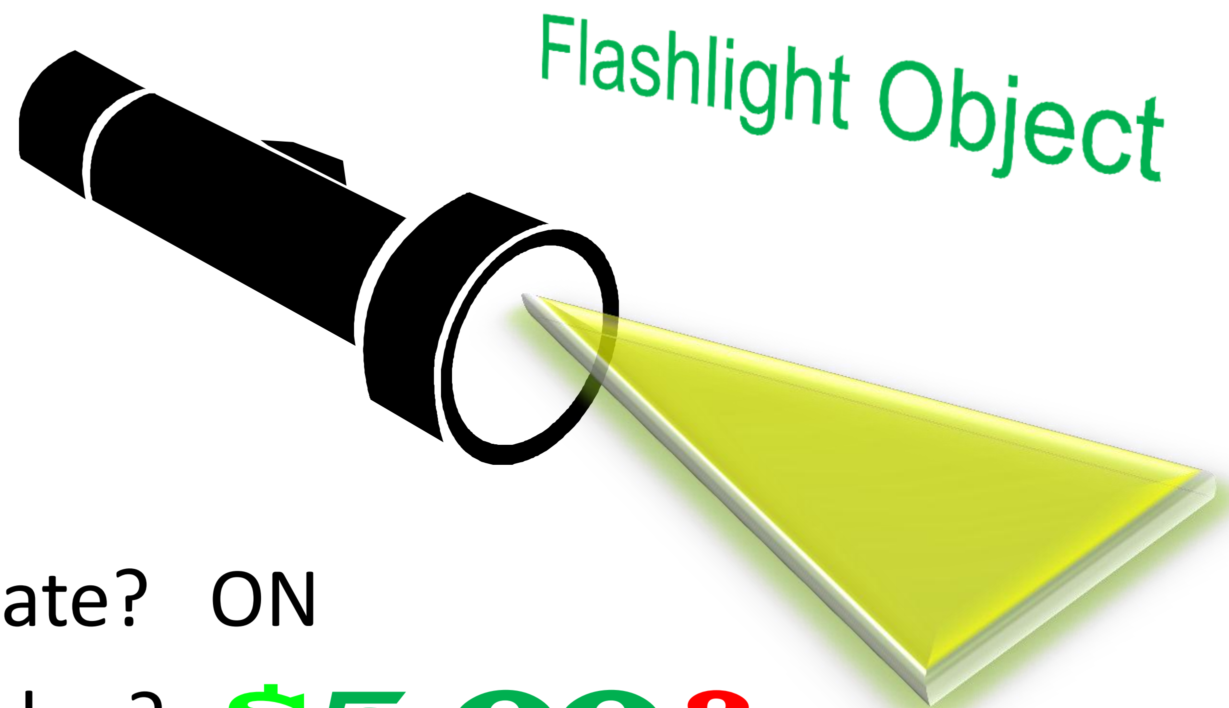
Does **state** *always* imply a "**value**"?



What is its state? ON

What is its value? £5.00 ?

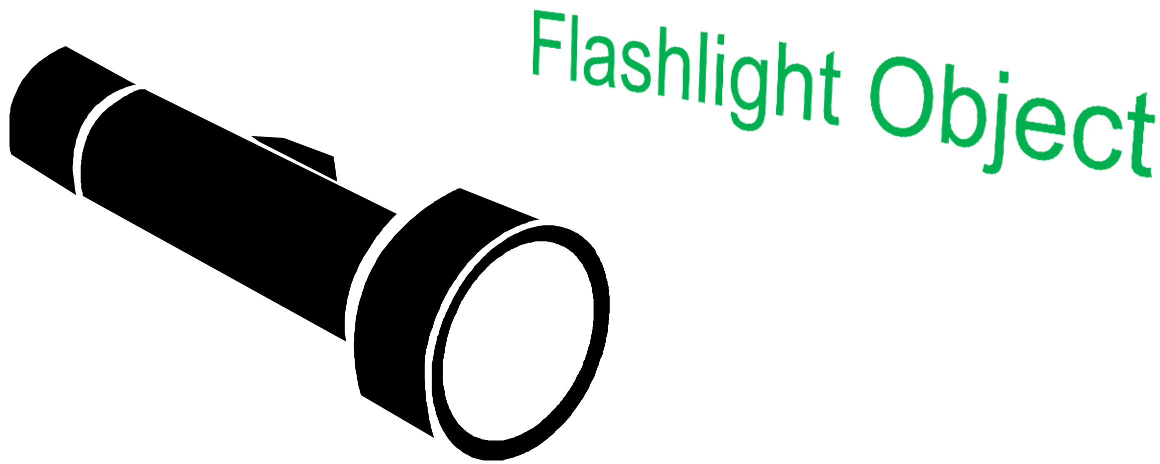
Does **state** *always* imply a "**value**"?



What is its state? ON

What is its value? $5.00 ?

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



What is its state? ON

What is its value? **?**

Any notion of “value” here would be artificial!

Does **state** *always* imply a "**value**"?

<u>Not</u> every ***stateful*** object has an ***obvious*** value.

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

Does **state** *always* imply a "**value**"?

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

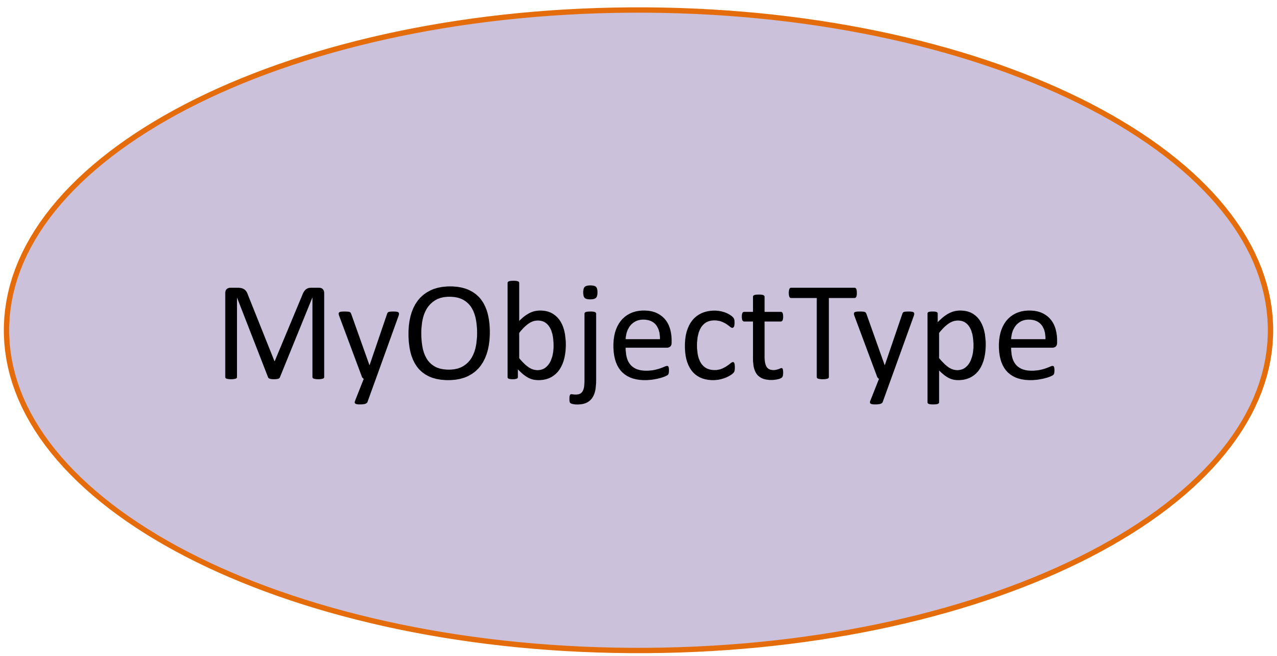
- TCP/IP Socket
- Thread Pool
- Condition Variable
- Mutex Lock
- Reader/Writer Lock
- Scoped Guard

- Base64 En(De)coder
- Expression Evaluator
- Language Parser
- Event Logger
- Object Persistor
- Widget Factory

Does **state** *always* imply a "**value**"?

We refer to *stateful* objects that do <u>not</u> represent a value as "Mechanisms".

Categorizing Object Types

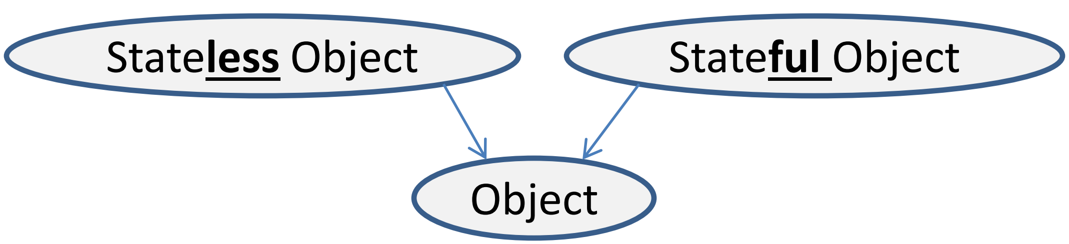


Categorizing Object Types

The first question: “Does it have state?”

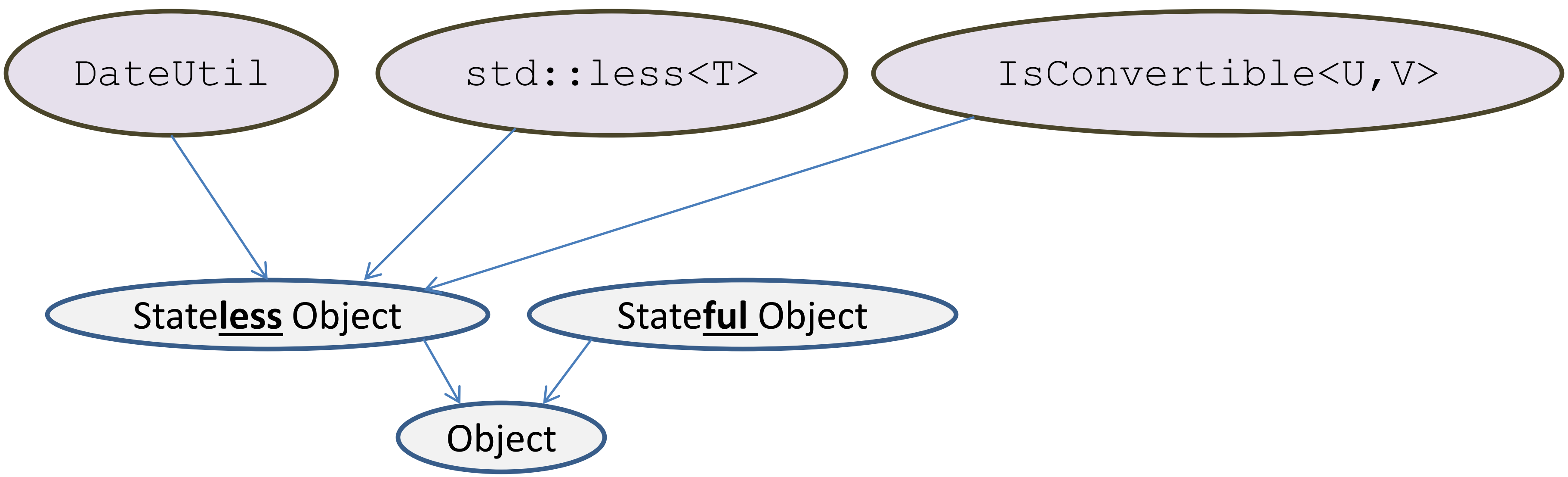
Categorizing Object Types

The first question: "Does it have state?"



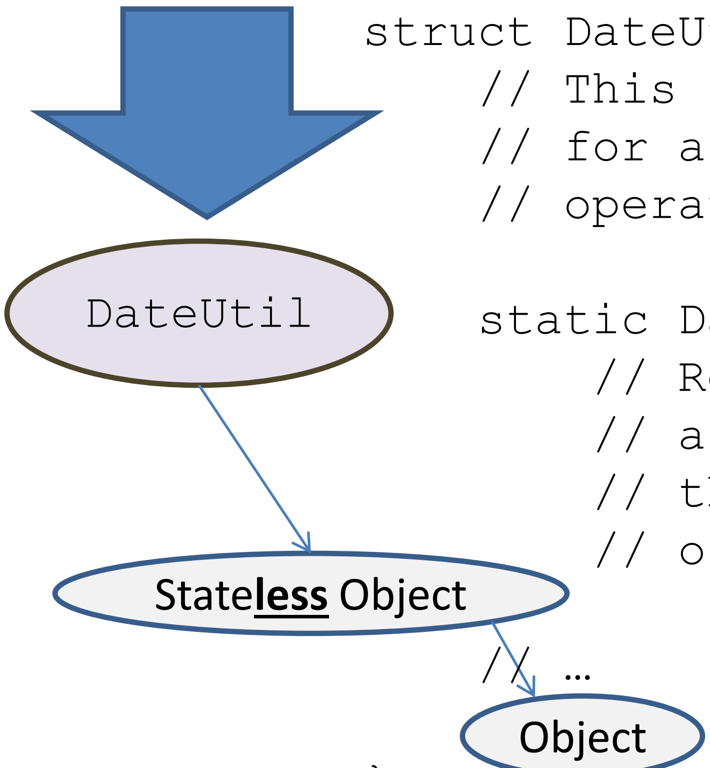
Categorizing Object Types

The first question: "Does it have state?"



Categorizing Object Types

The first question: “Does it have state?”

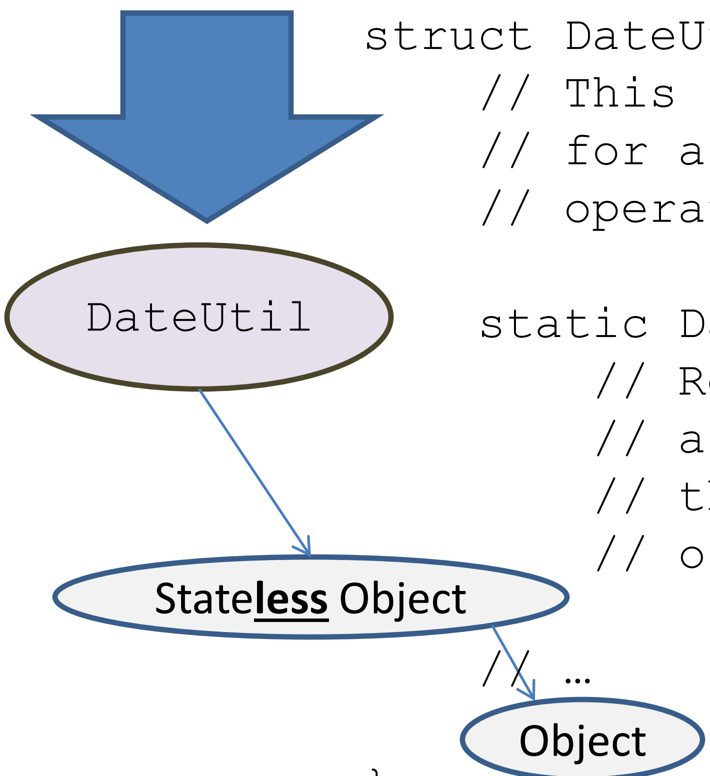



```
struct DateUtil {
    // This struct provides a namespace
    // for a suite of non-primitive functions
    // operating on Date objects.

    static Date lastDateInMonth(const Date& value);
        // Return the last date in the same month
        // as the specified date 'value'.  Note
        // that the particular day of the month
        // of 'value' is ignored.

    // ...
}
```

Categorizing Object Types

The first question: "Does it have state?"

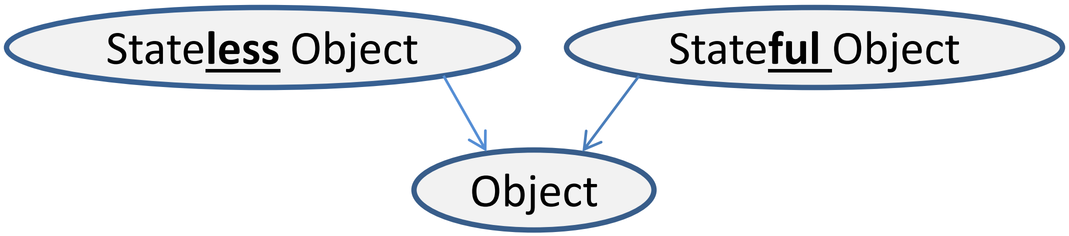
```
struct DateUtil {
    // This struct
    // for a suite
    // operating on

    static Date las                                    e);
        // Return th                                   n
        // as the sp
        // that the
        // of 'value

    // …
}
```



Utilities are an important class category!

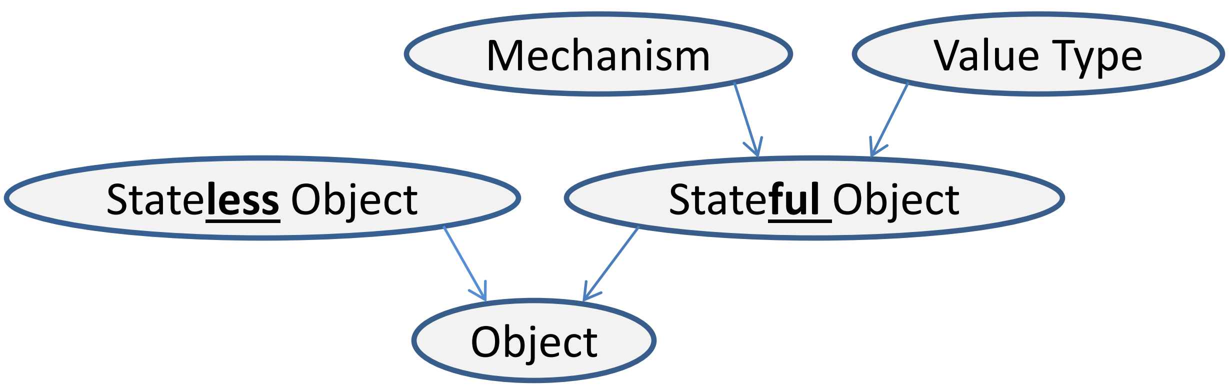
Categorizing Object Types

The second question: "Does it have value?"

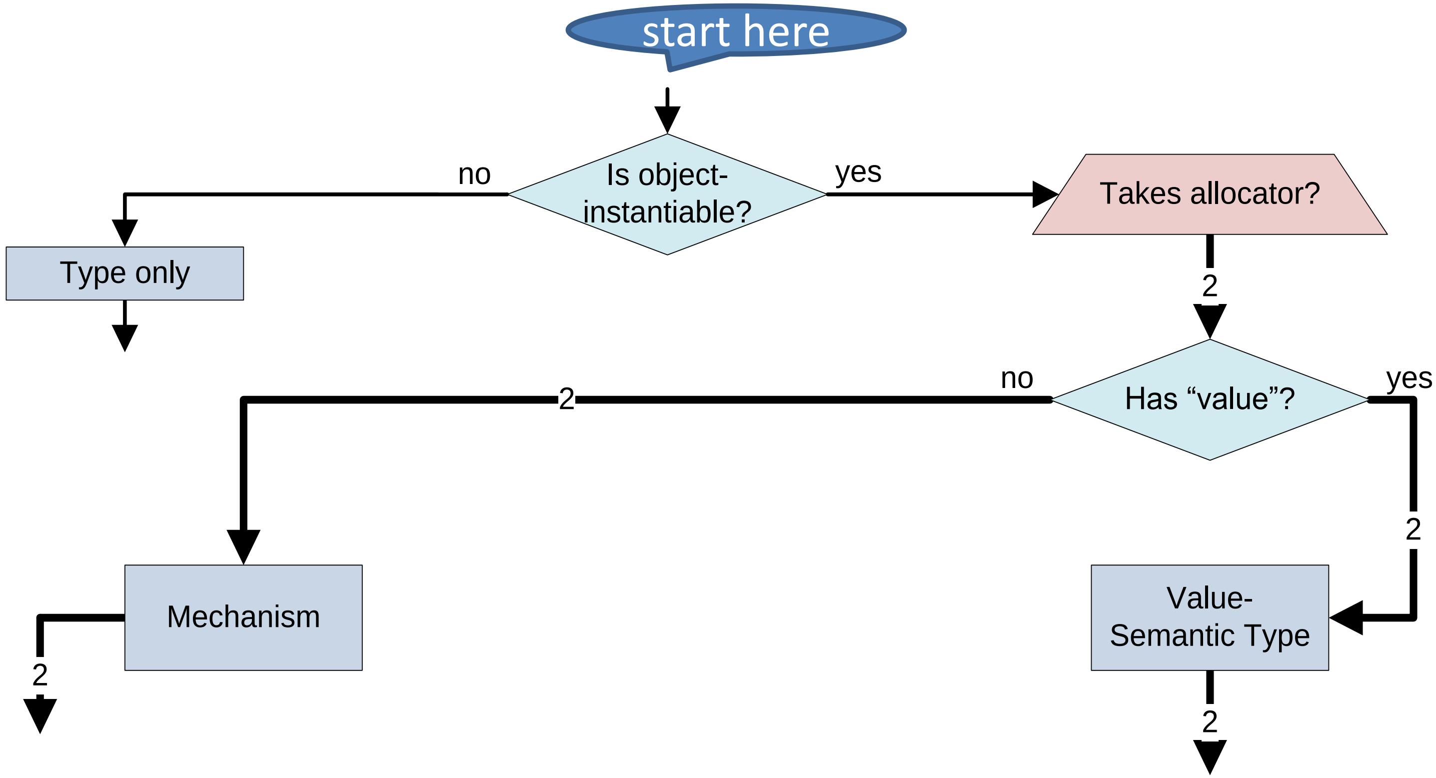


Categorizing Object Types

The second question: “Does it have value?”

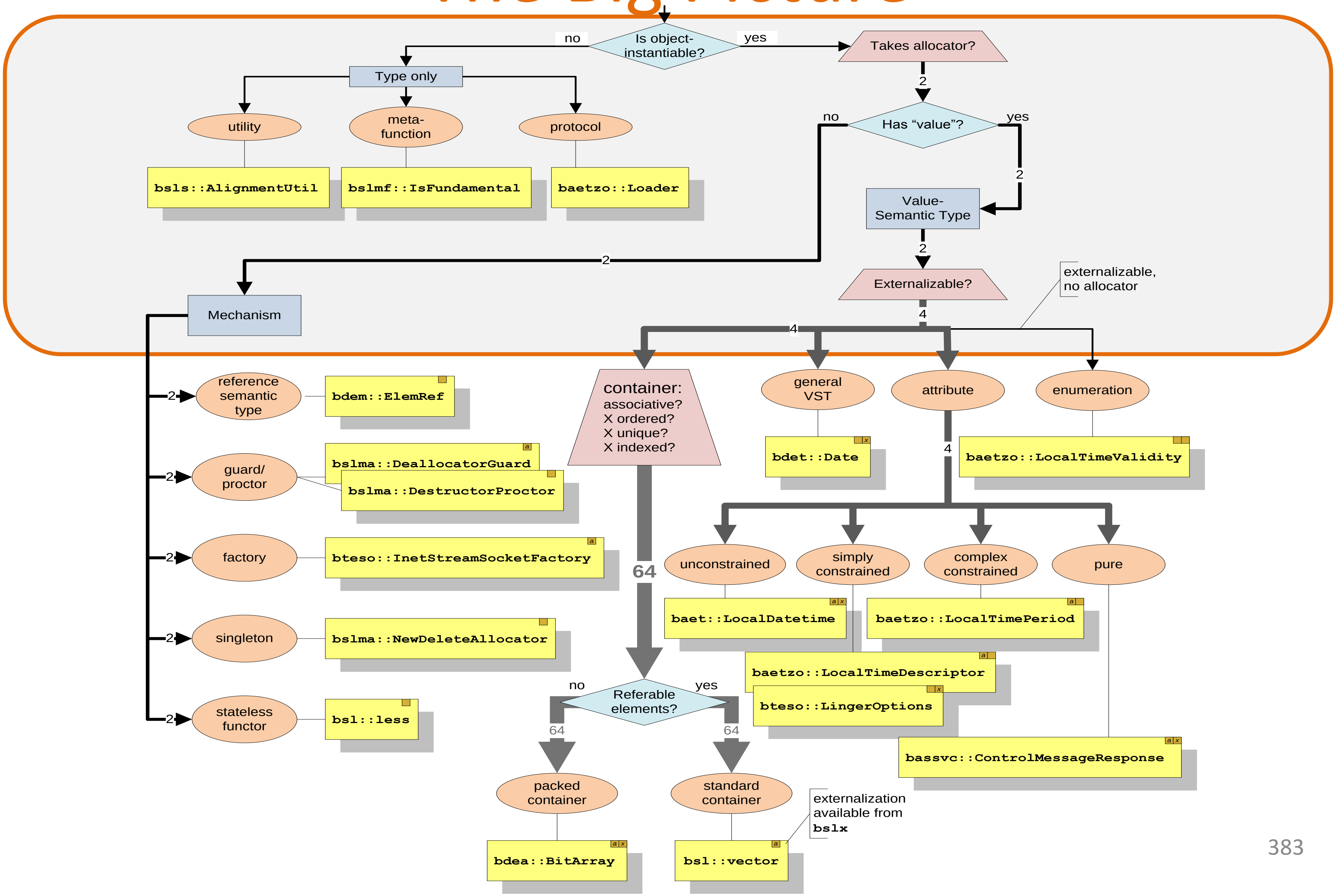


Top-Level Categorizations

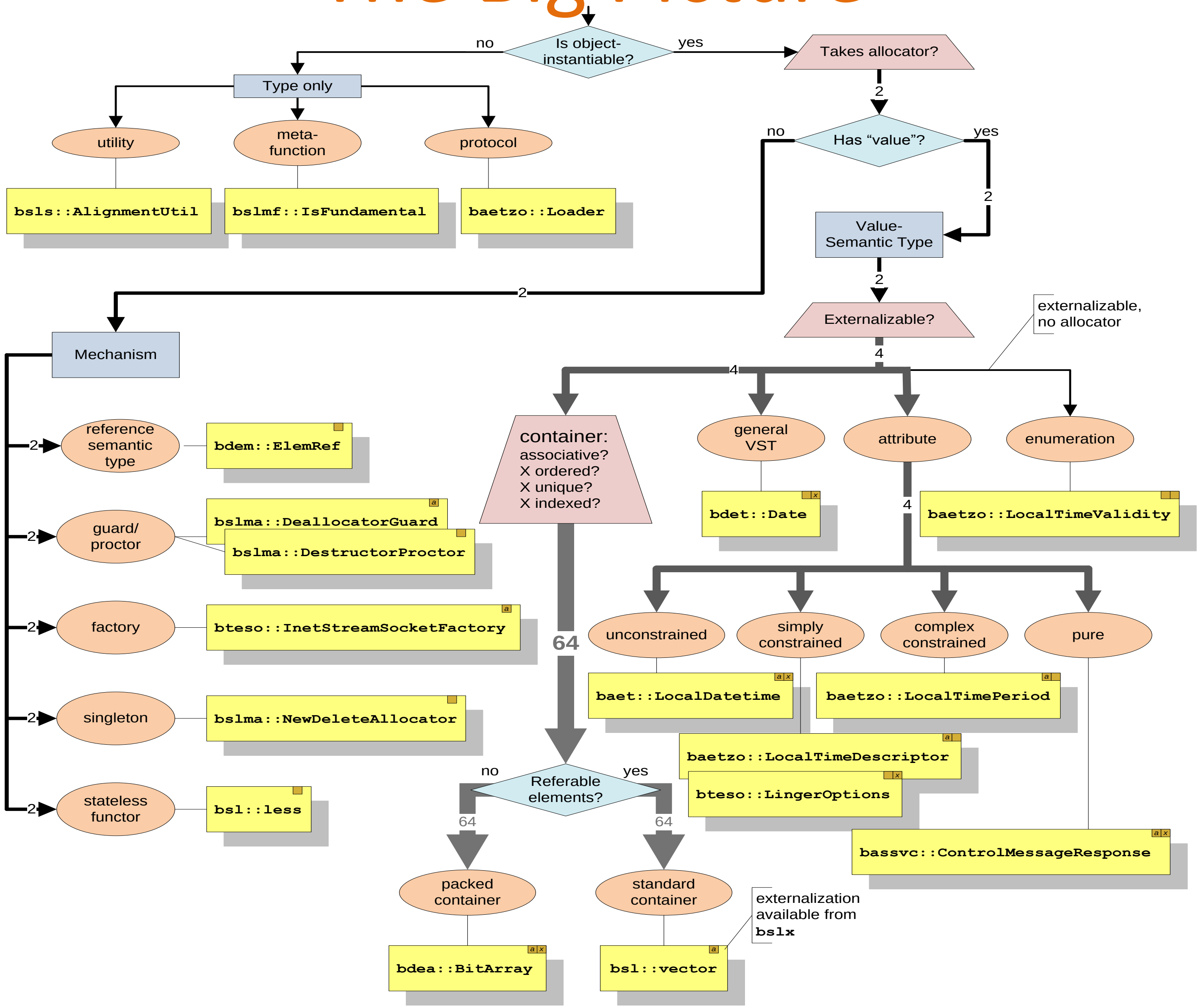


2. Design & Implementation

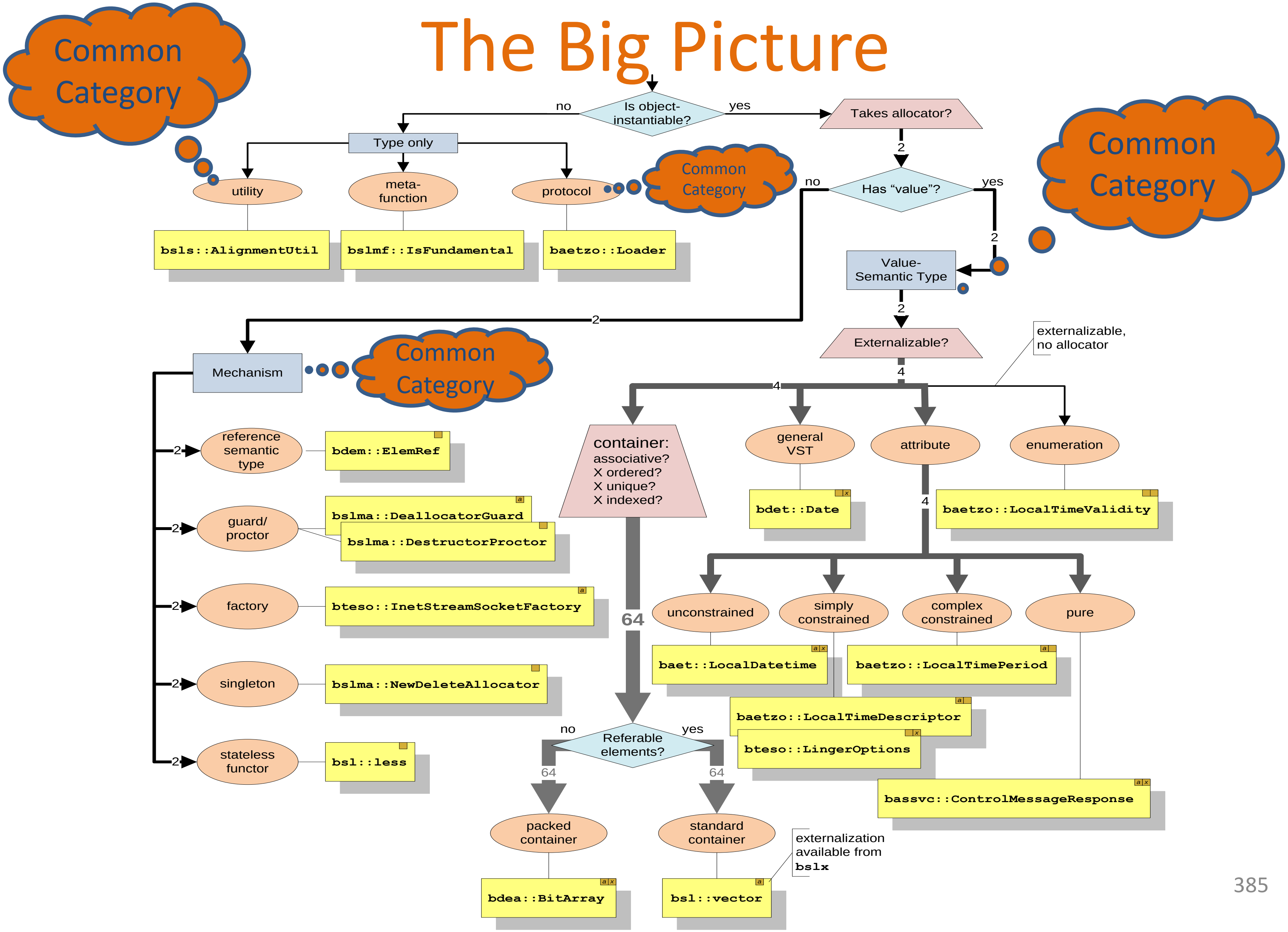
The Big Picture



The Big Picture



The Big Picture



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

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**.

Vocabulary Types

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

- We achieve interoperability by the ubiquitous use of:

Vocabulary Types

2. Design & Implementation

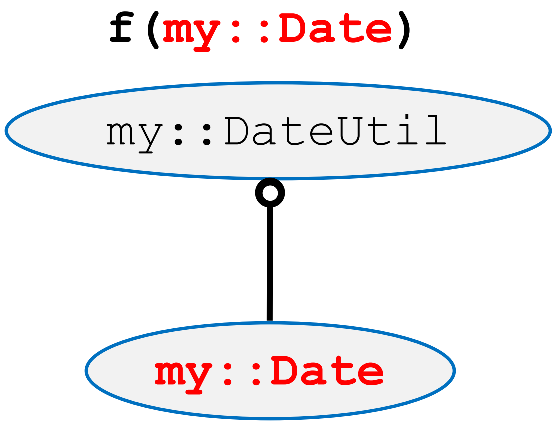
Vocabulary Types

(An Example)

`my::Date`

Vocabulary Types

(An Example)

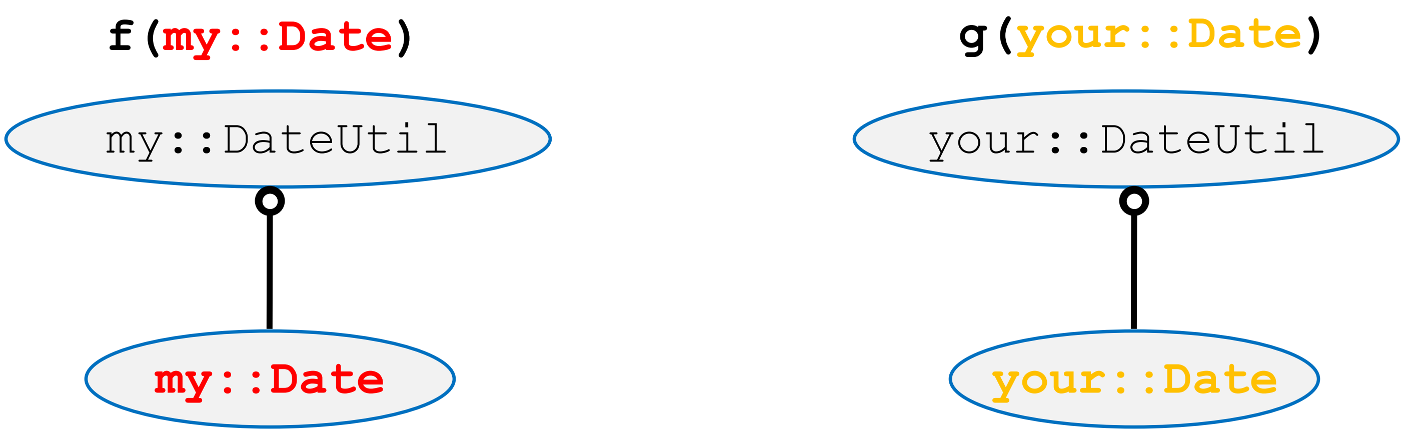


Vocabulary Types

(An Example)

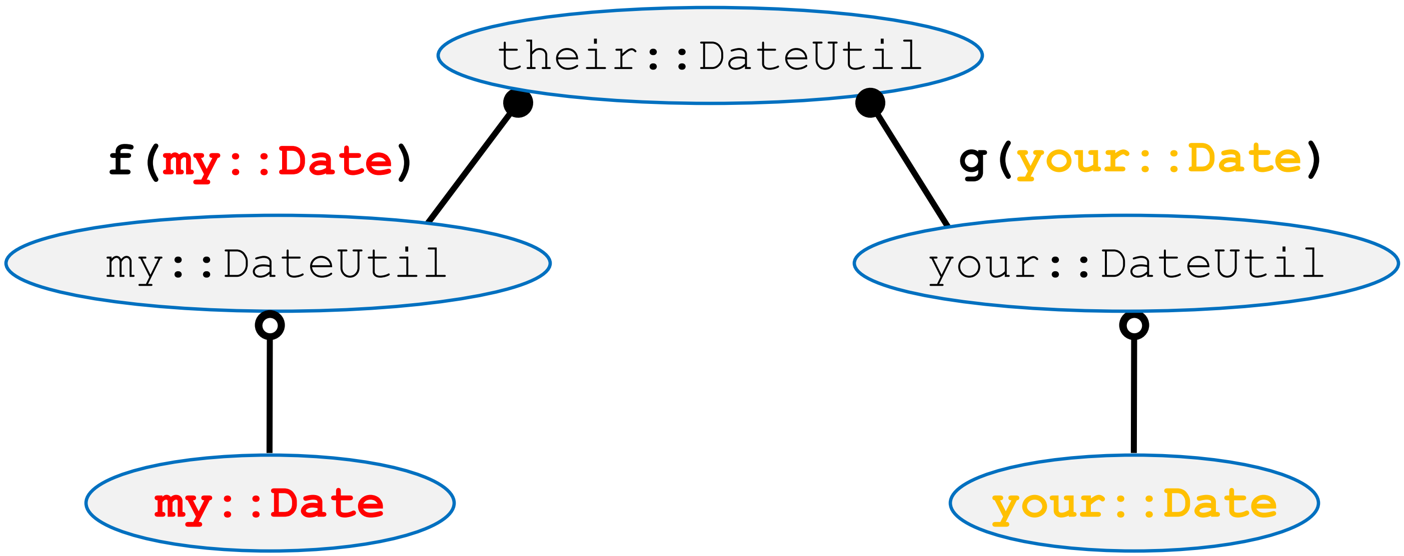
Vocabulary Types

(An Example)



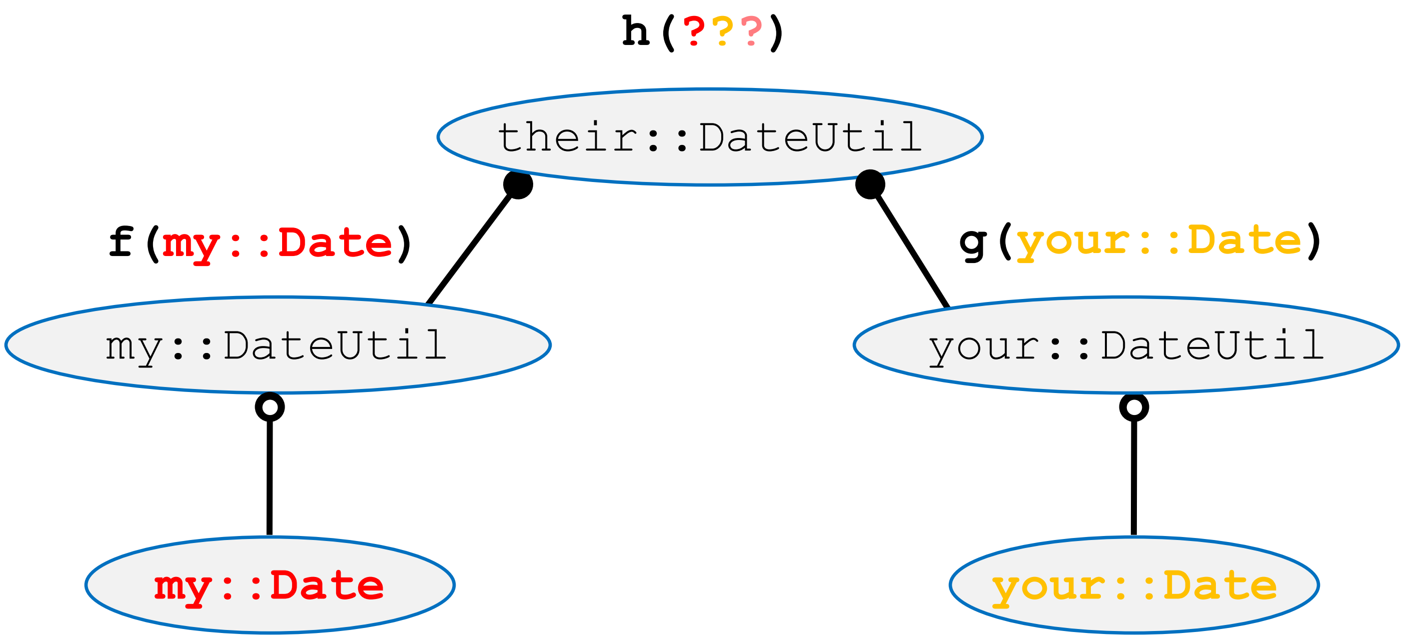
Vocabulary Types

(An Example)



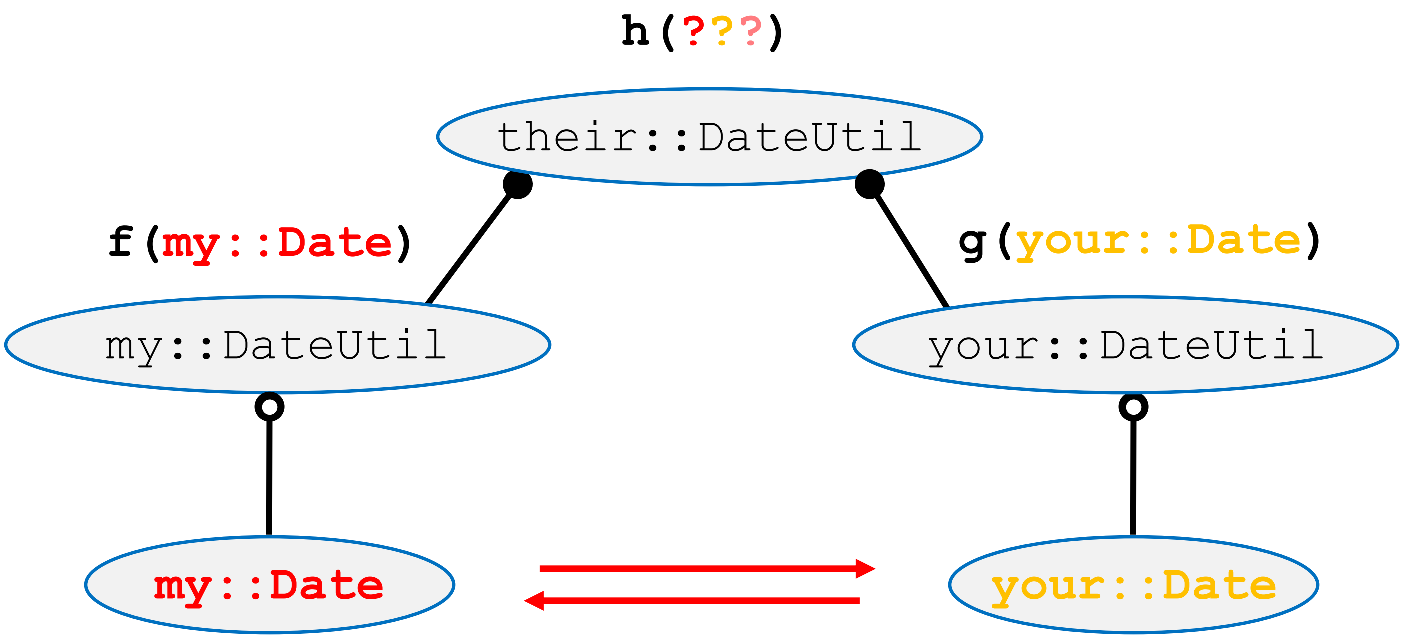
Vocabulary Types

(An Example)



Vocabulary Types

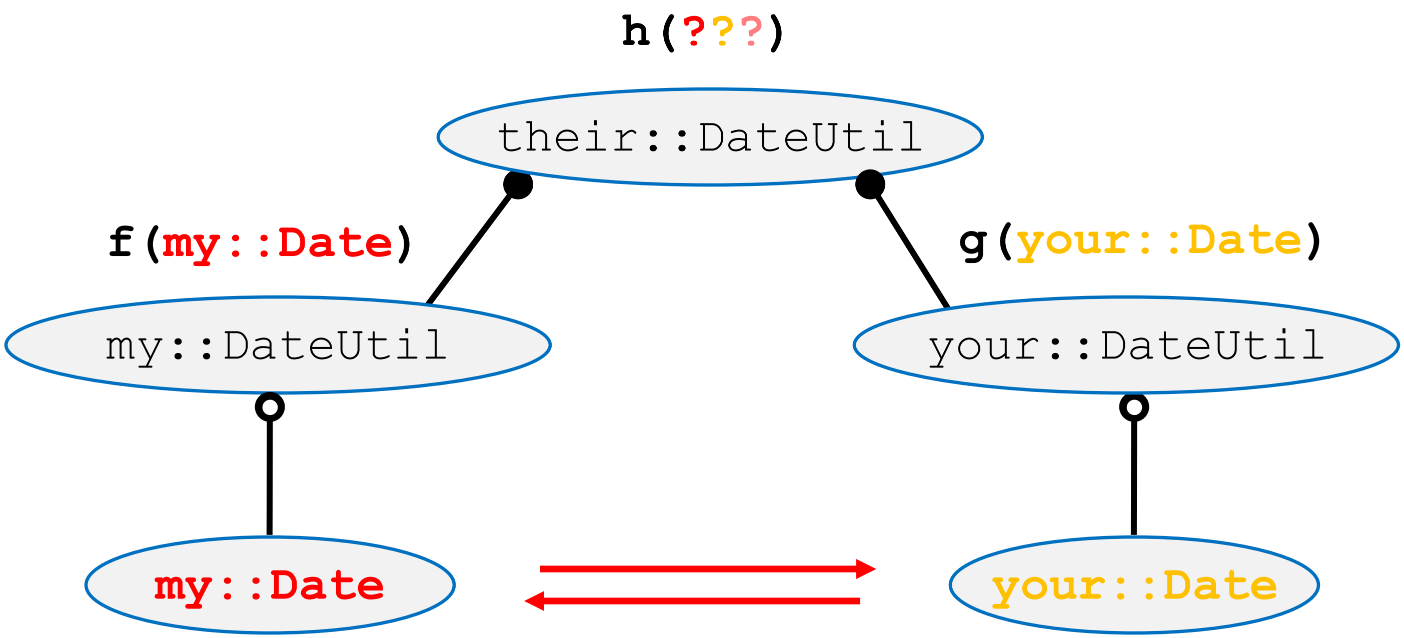
(An Example)



Interoperability Problem!

Vocabulary Types

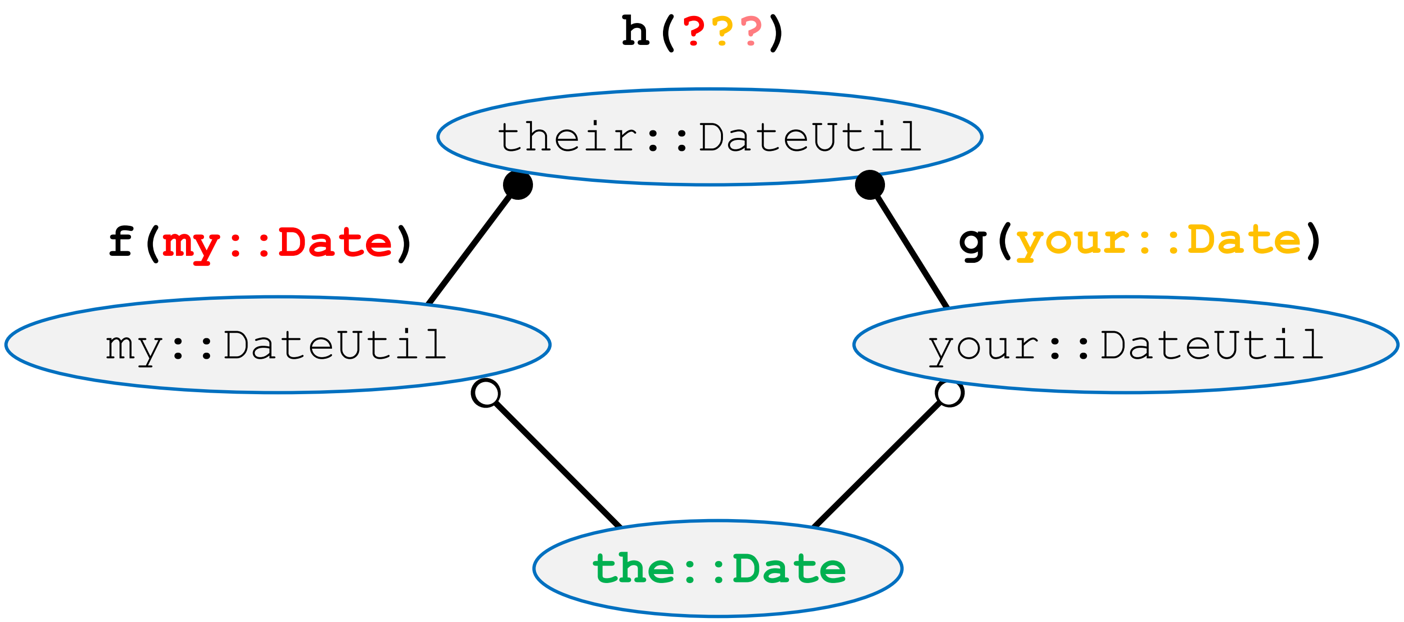
(An Example)



What should we do?

Vocabulary Types

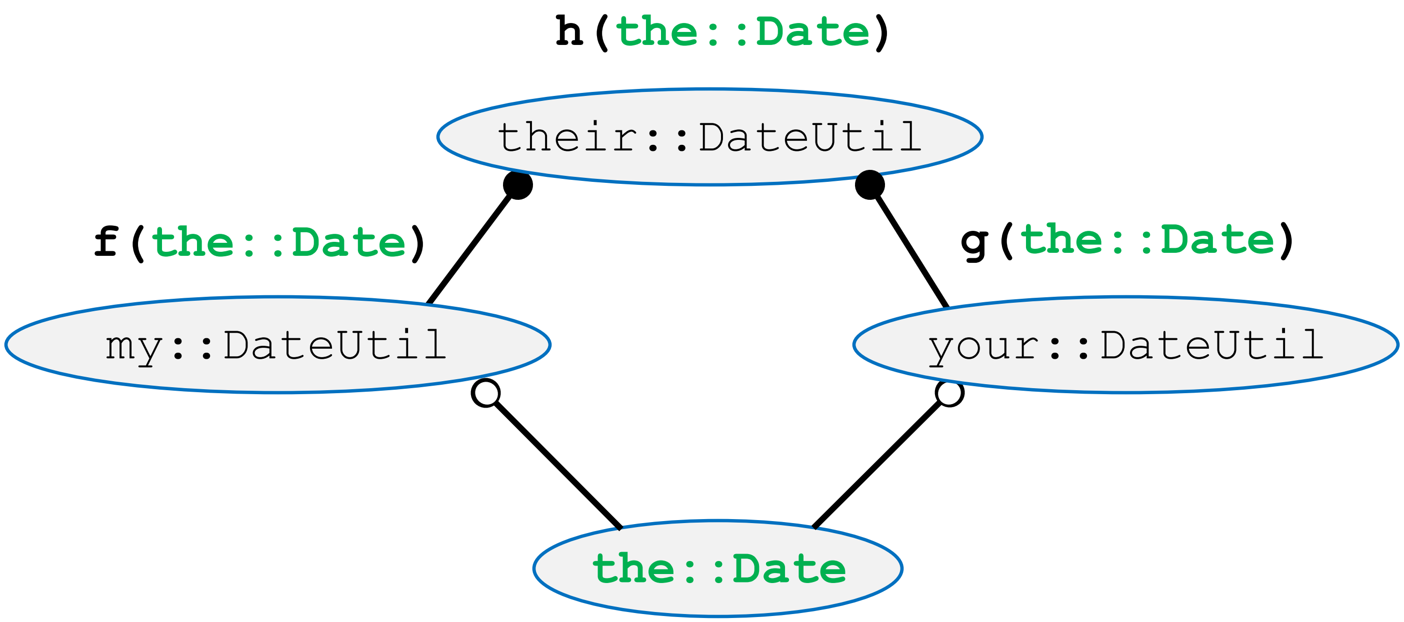
(An Example)



What should we do?

Vocabulary Types

(An Example)

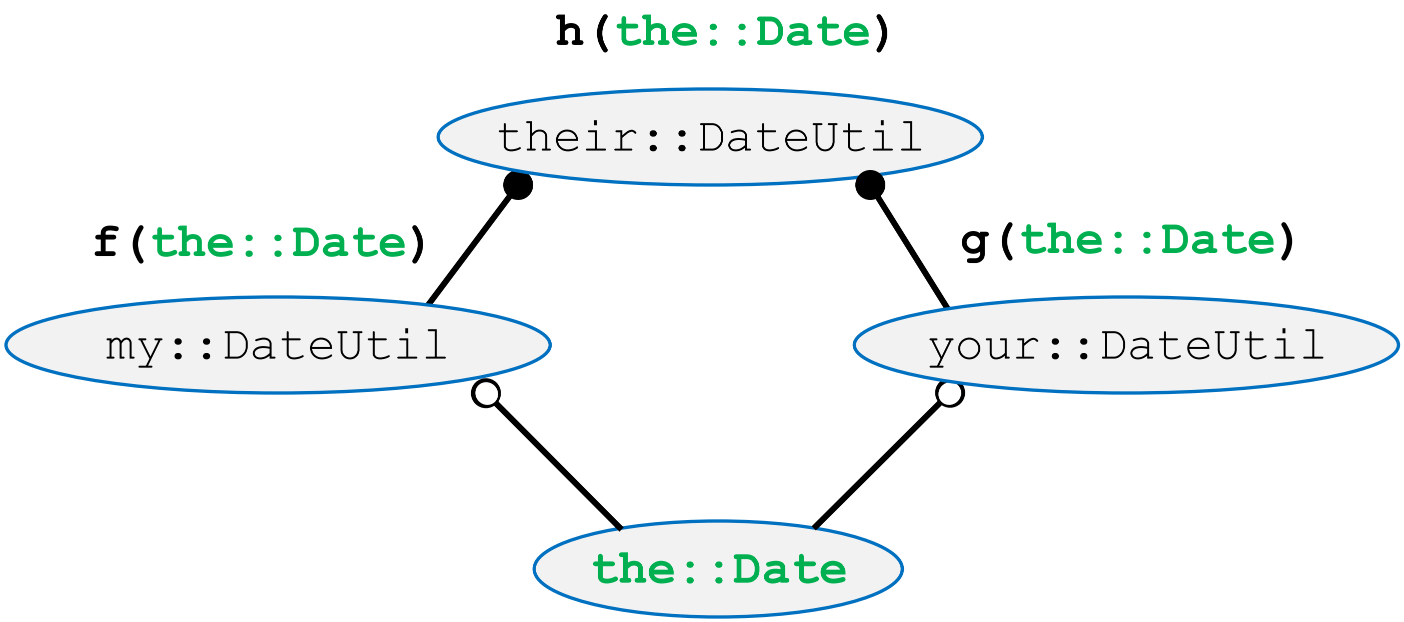


What should we do?

2. Design & Implementation

Vocabulary Types

(An Example)



No Interoperability Problem!

Vocabulary Types

On the other hand…

Distinct algebraic structures deserve distinct C++ types.

Vocabulary Types

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

Vocabulary Types

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

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

Vocabulary Types

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

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

Vocabulary Types

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

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

++x:
```

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.

Vocabulary Types

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

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

++x: 20080332
++y:
```

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.

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 <u>own</u> C++ type!

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;
```

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;
```

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

Vocabulary Types

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

integer	***string***
• **Age**	• Text
• **Shoe Size**	• Word
• **Account Number**	• Username
• **Year**	• Filename
• **Day of Month**	• Password
• **Number of Significant Digits**	• Regular Expression

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**

Template Policies

TEMPLATES CAN PRESENT A VOCABULARY PROBLEM

Template Policies

Template parameters can be partitioned into three basic categories:

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.

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.

Template Policies

Essential Parameters

- Are necessary for basic operation.
- Typically do <u>not</u> have reasonable defaults.

Template Policies

Essential Parameters

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

Example:

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

Template Policies

Essential Parameters

- Are necessary for basic operation.
- Typically do … sonable defaults.

Essential Parameter

Example:

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

Template Policies

Essential Parameters

- Are necessary for basic operation.
- Typically do <u>not</u> have reasonable defaults.

Example:

```
template <class T> class vector;

template <class Iter>
void sort(Iter begin, Iter end);
```

Template Policies

Essential Parameters

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

Example:

```
template <class T> cl          or;

template <class Iter>
void sort(Iter begin, Iter end);
```

Essential Parameter

Template Policies

Interface Policies

- Affect intended “logical” behavior.
- Typically <u>do</u> have reasonable defaults.

Template Policies

Interface Policies

- Affect intended "logical" behavior.
- Typically do have reasonable defaults.

Example:

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

Template Policies

Interface Policies

- Affect intended “logical” behavior.
- Typically do … able defaults.

Essential Parameter

Example:

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

Template Policies

Interface Policies

- Affect intended “logical” behavior.
- Typically do… …able def…

Essential Parameter

Interface Policy

Example:

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

Template Policies

Implementation Policies

- **DO NOT** affect intended "logical" behavior.
- Typically do have reasonable defaults.

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;
```

Template Policies

Implementation Policies

- **DO NOT** affect intended “logical” behavior.
- Ty… e reasonable defaults.

Essential Parameter

Example:

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

Template Policies

Implementation Policies

- **DO NOT** affect intended “logical” behavior.
- Ty… e r…

Essential Parameter

Implementation Policy

Example:

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

Template Policies

Implementation Policies

- **DO NOT** affect intended “logical” behavior.
- Ty… e r…

Example:

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

Essential Parameter

Implementation Policy

Implementation Policy

Template Policies

Implementation Policies

- **<u>DO NOT</u>** affect intended "logical" behavior.
- Typically <u>do</u> have reasonable defaults.

Example:

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

Template Policies

Implementation Policies

- **<u>DO NOT</u>** affect intended “logical” behavior.
- Ty[illegible]e reasonable defaults.

Essential Parameter

Example:

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

Template Policies

Implementation Policies

- **DO NOT** ...ffect intended "logical" behavior.
- Ty... e reasonable defaults.

Example:

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

Essential Parameter

Implementation Policy

Template Policies

Problem!

Template Parameters Affect Object Type.

Template Policies

Essential Parameters:

```
vector<int> a;
vector<int> b;
a = b;
```

Template Policies

Essential Parameters:

```
vector<int> a;
vector<double> b;
a = b;
```

Compiler Error

FINE!

Template Policies

Interface Policies:

```
OrderedSet<int> a;
OrderedSet<int> b;
if (a == b) {
    // …
```

Template Policies

Interface Policies:

```
OrderedSet<int> a;
OrderedSet<int, MyLess> b;
if (a == b) {
    // ...
```

Compiler Error

Template Policies

Implementation Policies:

```
void f(Queue<double> *queue);

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

Template Policies

Implementation Policies:

```
void f(Queue<double> *queue);

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

Compiler Error

Template Policies

Implementation Poli

```
template<class T>
void f(T *queue);

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

Compiles Fine

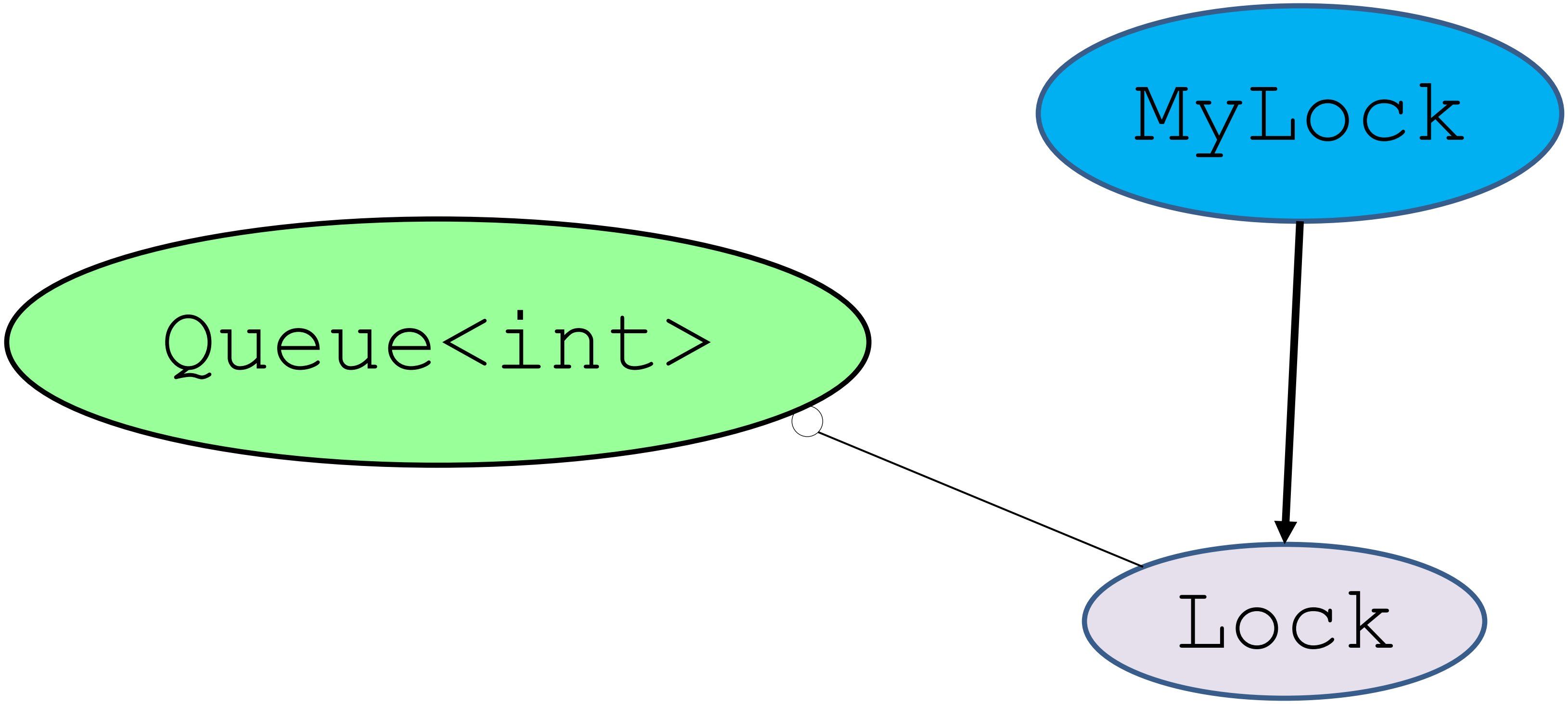
The *Entire* Implementation Must Now Reside In the Header File

Template Policies

<u>Solution!</u>

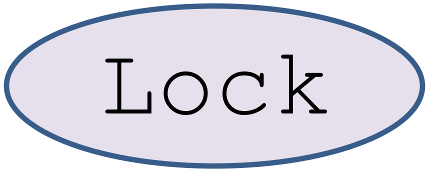
Runtime Implementation Policies

Runtime Implementation Policies



2. Design & Implementation

Runtime Implementation Policies



Runtime Implementation Policies

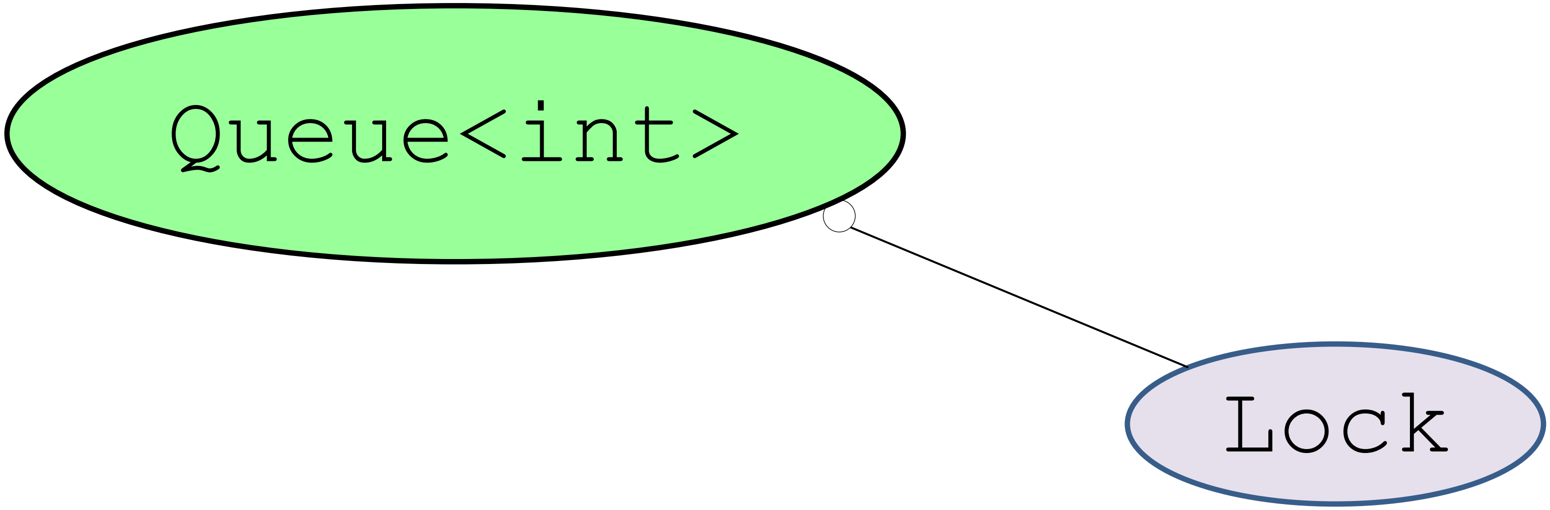
```
class Lock {
    // Pure abstract (protocol) class.
  public:
    virtual ~Lock();

    virtual void lock() = 0;
    virtual void unlock() = 0;
};
```

Runtime Implementation Policies

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

    virtual void lock(
    virtual void unloc        0;
};
```

Common Class Category

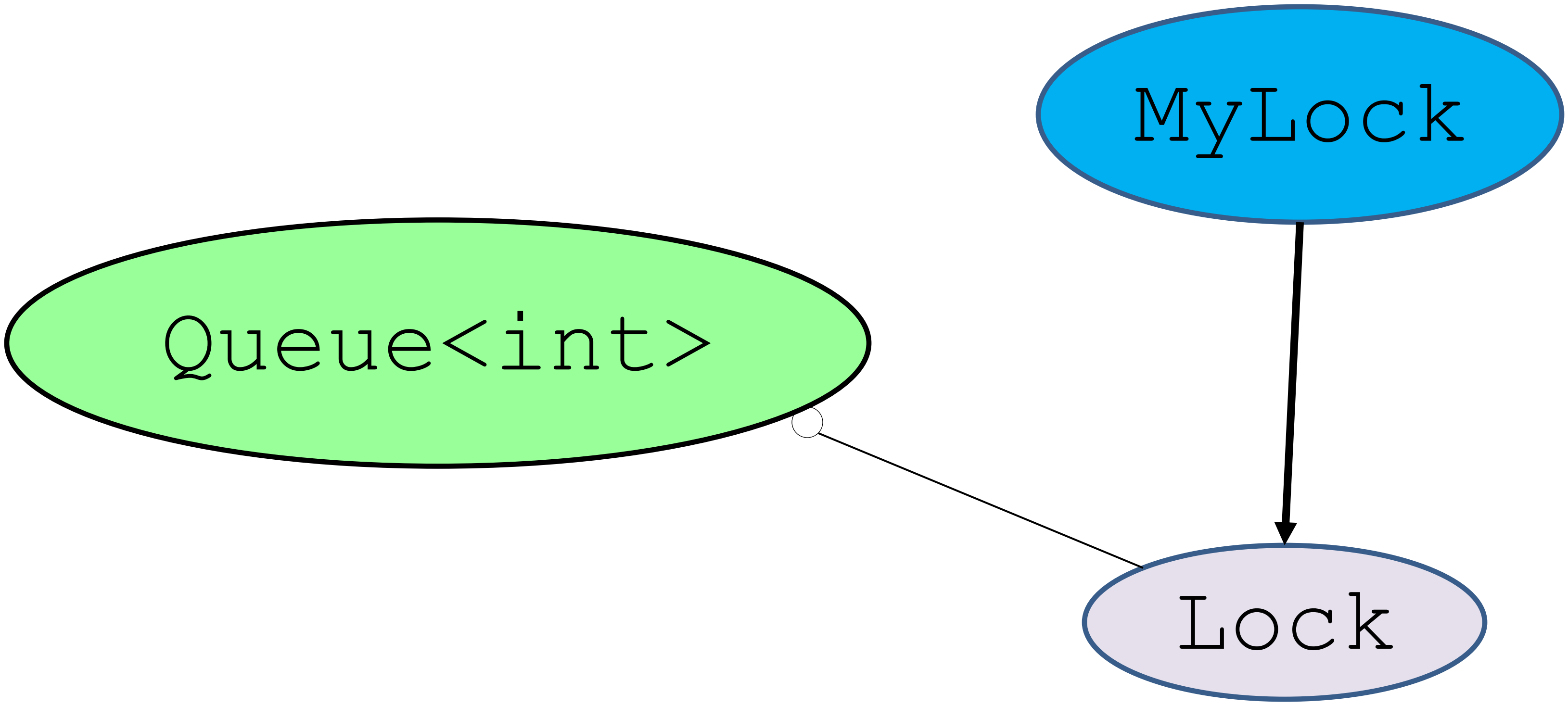
Runtime Implementation Policies



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);
    // …
};
```

Runtime Implementation Policies



Runtime Implementation Policies

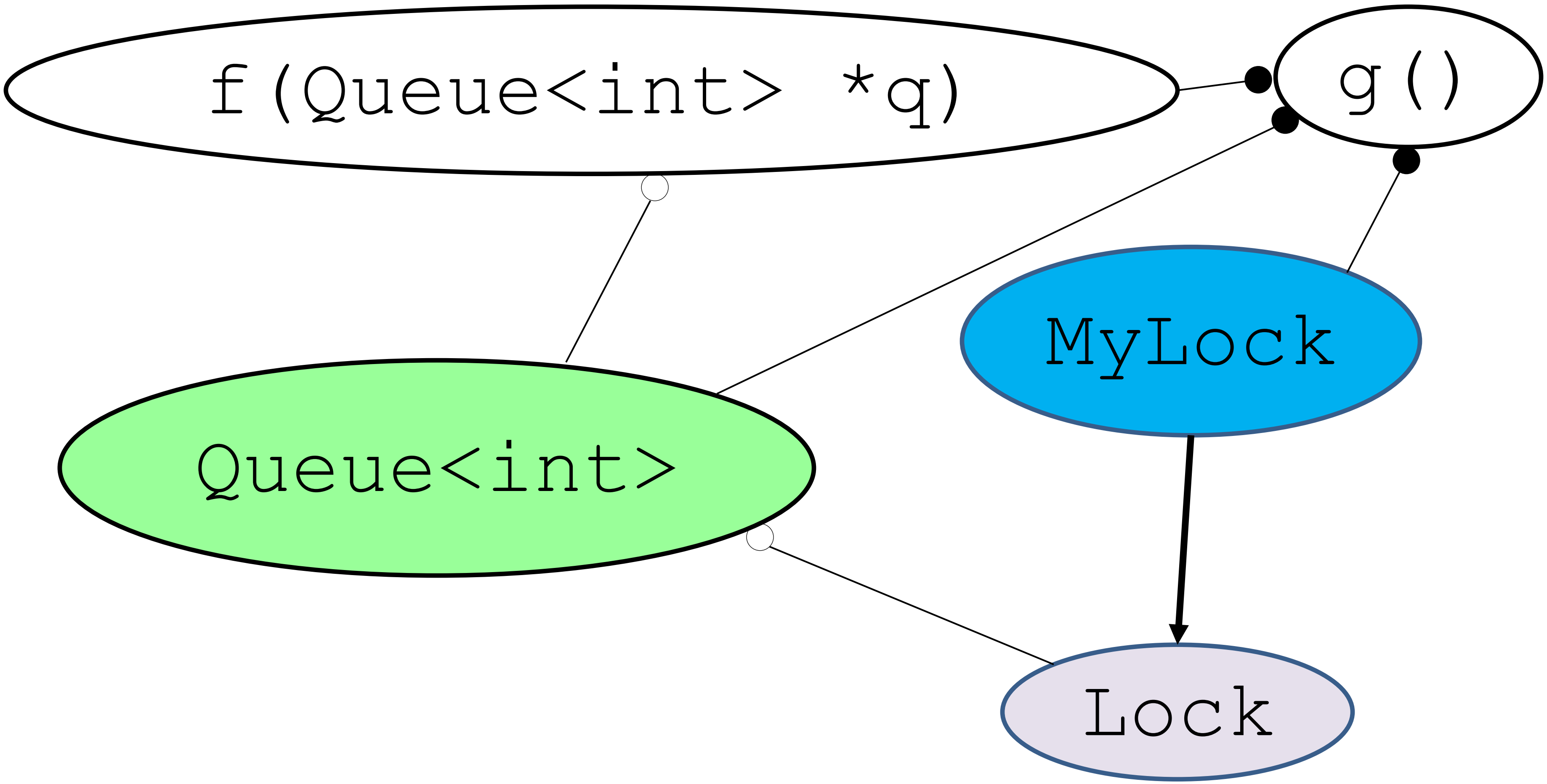
```
class MyLock : public Lock {
    // … Concrete mechanism.
  private:
    MyLock(const MyLock&);
    MyLock& operator=(const MyLock&);
  public:
    MyLock();
    virtual ~MyLock();
    virtual void lock();
    virtual void unlock();
};
```

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

Runtime Implementation Policies



Runtime Implementation Policies

```
void f(Queue<double> *q);

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

Runtime Implementation Policies

```
void f(Queue<double> *q);

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

Runtime Implementation Policies

```
void f(Queue<double> *q);

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

Runtime Implementation Policies

```
void f(Queue<double> *q);

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

Runtime Implementation Policies

```
void f(Queue<double> *q);

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

Runtime Implementation Policies

```
void f(Queue<doubl

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?

Memory Allocators

What is a memory allocator?

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

Memory Allocators

What is a memory allocator?

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

Memory Allocators

What is a memory allocator?

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

Memory Allocators

What is a memory allocator?

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

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*.

Polymorphic Memory Allocators

What is a memory allocator?

It should look like a "Lock" or any other *abstract mechanism*.

Polymorphic Memory Allocators

What is a memory allocator?

It should look like a "Lock" or any other *abstract mechanism.*

Vocabulary Type

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;
}
```

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;
}
```

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.

(**Halpern**-13: *Polymorphic Allocators*, **N3525**)

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

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

Interfaces and Contracts

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

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

Interfaces and Contracts

Function

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

Interfaces and Contracts

Function

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

Types Used
In the Interface

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.
```

Interfaces and Contracts

Class

```
class Date {

    //…
  public:
      Date(int year, int month, int day);

      Date(const Date& original);

      // …
};
```

Interfaces and Contracts

Class

```
class Date {

    //…

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

      Date(const Date& original);

      // …
};
```

Public Interface

Interfaces and Contracts

Class

```
class Date {

    //…
  public:
      Date(int year, int month, int day);

      Date(const Date& original);

      // …
};
```

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);


    // …
};
```

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.

    // …
};
```

Interfaces and Contracts

Component

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

```
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);
```

Interfaces and Contracts

Component

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

Interfaces and Contracts

Component

```
class Date {
   // …
  public:
   // …
};
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);
```

Interfaces and Contracts

Component

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

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'.
```

Preconditions and Postconditions

Preconditions and Postconditions

Function

Preconditions and Postconditions

Function

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

Preconditions and Postconditions

Function

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

Preconditions and Postconditions

Function

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

Precondition

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.

Preconditions and Postconditions

Function

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

Preconditions and Postconditions

Function

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

Postcondition

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

Preconditions and Postconditions

Object Method

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

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.

Preconditions and Postconditions

Object Method

a.k.a.
Essential
Behavior

- Precond… e of both (object) … otherwis… **ed**.

- 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.**

- Precond… …e of both (object)… otherwis… …ed.

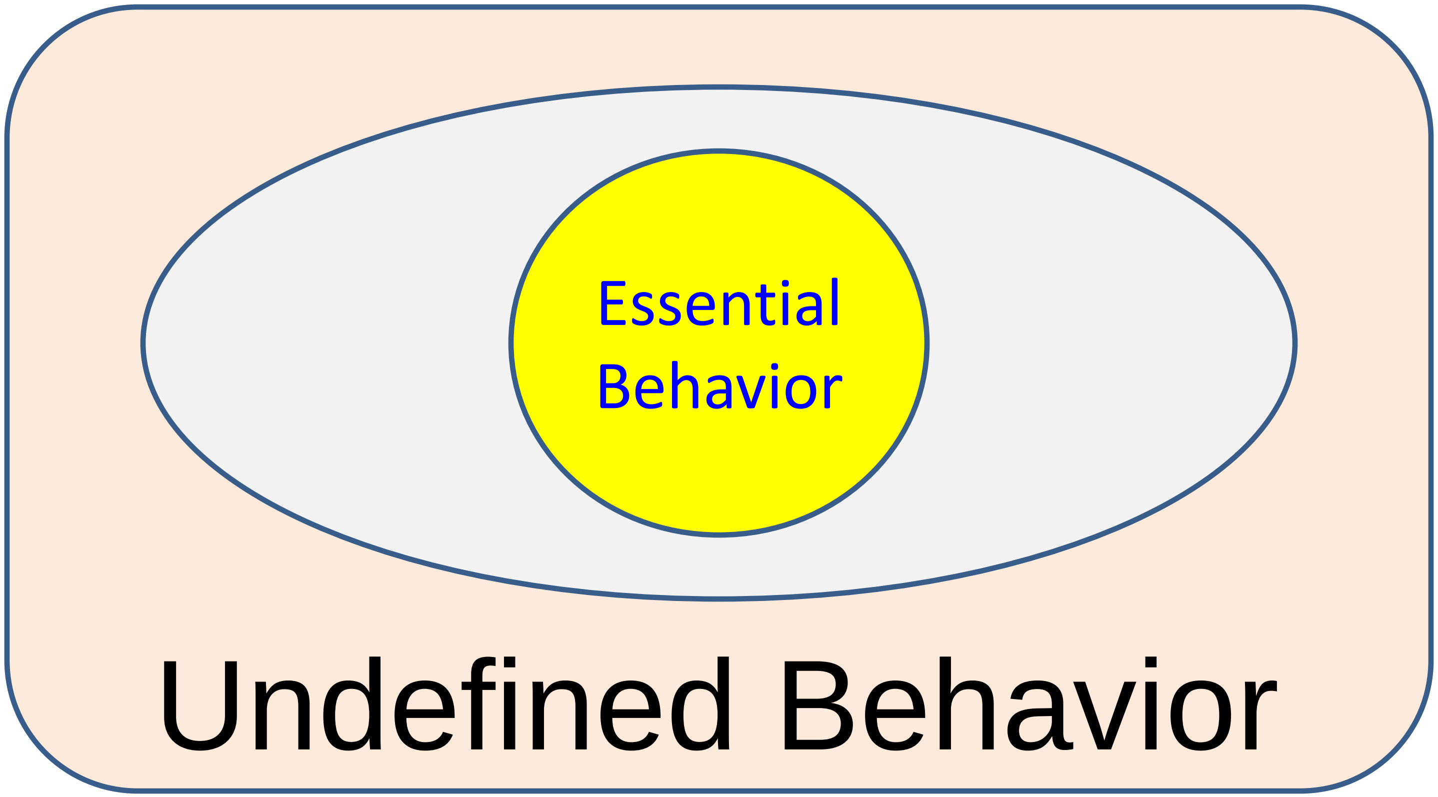
a.k.a. Essential Behavior

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

Observation By
Kevlin Henny

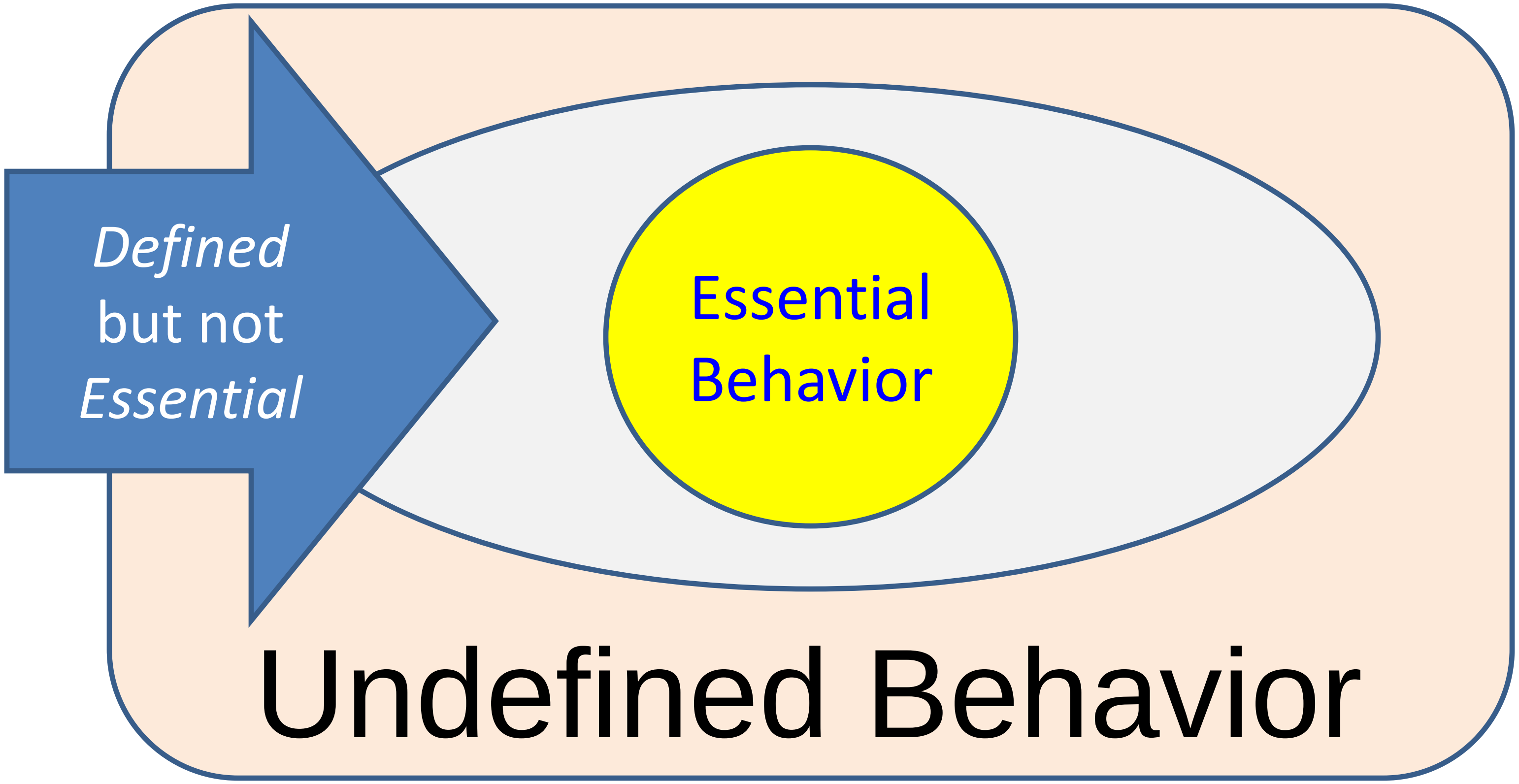
Preconditions and Postconditions

Defined & Essential Behavior



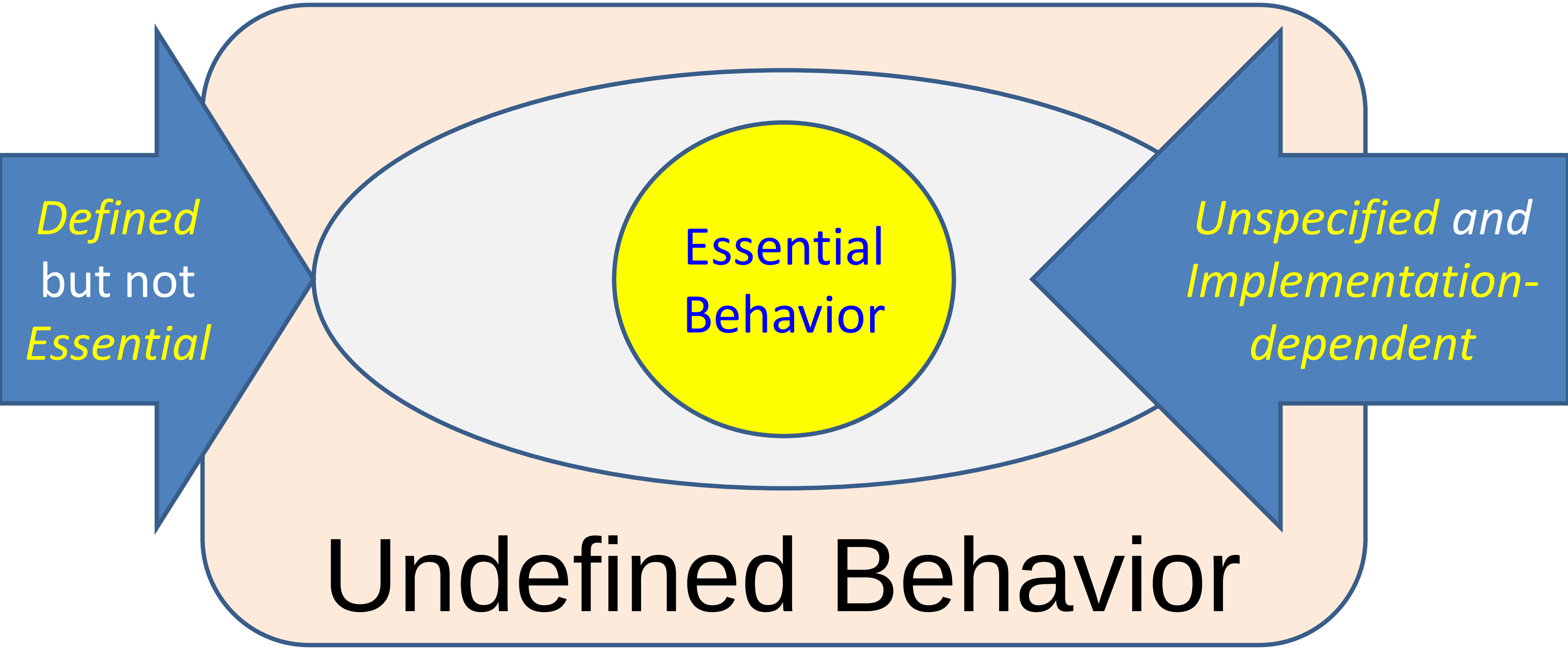
Preconditions and Postconditions

Defined & Essential Behavior



Preconditions and Postconditions

Defined & Essential Behavior



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.
```

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.
```

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.
```

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.
```

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.
```

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.
```

Preconditions and Po

Defined & Essenti

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.
```

Preconditions and Po

Defined & Essenti

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.
```

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.
      // …
};
```

Preconditions and Postconditions

Defined & Essential Behavior

```
class Date {
    // This class implements a value-seman
    // a valid date between the dates 000
    // 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.
      // …
};
```

Any Undefined Behavior?

Preconditions and Postconditions

Defined & Essential Behavior


```
class Date {
    // This class implements a value-seman
    // a valid date between the dates 000
    // 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.
      // …
};
```

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.
      // …
};
```

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 sp
        // 'day'.  The behavior is undefined u
        // represents a valid date in the range [0001/0           99/12/31].

    Date(const Date& original);
        // Create a date having the value of the specified 'original' date.
    // …
};
```

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.
      // …
};
```

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.
      // …
};
```

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?

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.
      // …
};
```

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!

…nd Postconditions

…ariants

…emantic type representing
…001/01/01 and

What happens when behavior is undefined is *undefined!*

Answer: No!

```
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.
    // …
};
```

Design by Contract

(DbC)

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

*including state

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…

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…

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…

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…

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…

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

Design by Contract

Verification

- **Preconditions:**

Design by Contract

Verification

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

Design by Contract

Verification

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

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.

Design by Contract

Verification

- **Preconditions:**
  - RTFM (Read the Manual).
  - Assert (only in '*debug*' or '*safe*' mode).
- **Postconditions:**

Design by Contract

Verification

- **Preconditions:**
  - RTFM (Read the Manual).
  - Assert (only in '*debug*' or '*safe*' mode).
- **Postconditions:**
  - Component-level test drivers.

Design by Contract

Verification

- **Preconditions:**
  - RTFM (Read the Manual).
  - Assert (only in '*debug*' or '*safe*' mode).
- **Postconditions:**
  - Component-level test drivers.
- **Invariants:**

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.

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

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

Defensive Programming

(DP)

- What is it?

Defensive Programming

(DP)

- What is it?

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

Defensive Programming

(DP)

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

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.

Defensive Programming (DP)

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

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?

Defensive Programming

What are we defending against?

Defensive Programming

What are we defending against?

- Bugs in software that we use in our implementation?

Defensive Programming

What are we defending against?

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

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?

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?

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?

Defensive Programming

What are we defending against?

MISUSE BY OUR CLIENTS

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

What should happen with the following call?

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

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?

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?

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?

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

More Code Runs Slower!

How about it must return 0?

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

What should happen with the following call?

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

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

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

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

} Narrow

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

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

} Narrow

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

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

} Narrow

Just Don't Pass 0!

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

Should

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

Return a status?

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

Should

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

Return a status?

Absolutely Not!

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?

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);
```

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);
```

Double Fault!!

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

- Returning status implies a ***wide*** interface contract.

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.

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;
}
```

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 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"

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;
}
```

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

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

- What should happen when the behavior is undefined?

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

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);
```

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);
```

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);
```

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);
```

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

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

Must check in **every** build mode!

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);
```

- …at happens be part … If it is, then it's defi… …or!

```
…PE& vector<TYPE>::at(i…
```

Must check in **every** build mode!

Bad Idea!

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

CRASH!

- What sh… …or is und… …mode.

```
TY… …rator[](int idx);
```

Must check in **every** build mode!

- …at happens be part … If it is, then it's defi… …or!

```
…PE& vector<TYPE>::at(i…
```

Bad Idea!

Narrow versus Wide Contracts

Narrow Contracts Admit *Undefined Behavior*:

- What sh… …or is unde… …d mode.

```
TYP… …[](int idx);
```

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

- …at happens be part… If it is, then it's defi… …or!

```
…PE& vector<TYPE>::at(i…
```

Must check in **every** build mode!

Bad Idea!

Contracts and Exceptions

Preconditions *always* Implply Postconditions:

Contracts and Exceptions

Preconditions *always* Imply Postconditions:

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

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).**

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).*

Appropriately Narrow Contracts

Narrow contracts admit *undefined behavior*.

Appropriately Narrow Contracts

Narrow contracts admit *undefined behavior*.

- Appropriately narrow contracts are <u>GOOD</u>:

Appropriately Narrow Contracts

Narrow contracts admit *undefined behavior*.

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

Appropriately Narrow Contracts

Narrow contracts admit *undefined behavior*.

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

Appropriately Narrow Contracts

Narrow contracts admit *undefined behavior*.

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

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*.

Appropriately Narrow Contracts

Narrow contracts admit *undefined behavior*.

- Appropriately narrow contracts are <u>GOOD</u>:
  - 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!

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

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

An Overriding Customer Focus

(1) Capture Practical Usage Example(s):

An Overriding Customer Focus

(1) Capture Practical Usage Example(s):

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

An Overriding Customer Focus

(1) Capture Practical Usage Example(s):

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

…its *raison d’être*.

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.

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

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);
//..
```

An Overriding Customer Focus

(2) Canonical Organization:

An Overriding Customer Focus

(2) Canonical Organization:

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

An Overriding Customer Focus

(2) Canonical Organization:

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

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** ...

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.

An Overriding Customer Focus

(2) Canonical Organization:

- Th

Easy to Find and Use

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

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

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

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

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`**

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…

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**.

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…

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***.

An Overriding Customer Focus

(3) Consistent, Useful Rendering:

- Make it look lik

Helpful Visual Cues