C++ Modules and Large-scale Development

Bloomberg

Engineering

ACCU 2019 (Autumn)
November 12, 2019

John Lakos
Senior Architect

TechAtBloomberg.com

C++ Modules &
Large-Scale Development

John Lakos

Tuesday, November 12, 2019

This version is for ACCU'19 (Autumn), Belfast Ireland

Copyright Notice

Abstract

Much has been said about how the upcoming module feature in C++ will improve compilation speeds and reduce reliance on the C++ preprocessor. However, program architecture will see the biggest impact. This talk explains how modules will change how you develop, organize, and deploy your code. We will also cover the stable migration of a large code base to be consumable both as modules and as normal headers.

What’s The Problem?

What's The Problem?

Large-Scale C++ Software Design is Multi-Dimensional:

What's The Problem?

Large-Scale C++ Software Design is Multi-Dimensional:

- It involves many subtle *logical* and *physical* aspects.

What's The Problem?

Large-Scale C++ Software Design is Multi-Dimensional:

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

What's The Problem?

Large-Scale C++ Software Design is Multi-Dimensional:

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

What's The Problem?

Large-Scale C++ Software Design is Multi-Dimensional:

- It involves many subtle *logical* and *physical* aspects.
- It requires an ability to isolate and modularize logical functionality within discrete, fine-grained physical components.
- It requires the designer to delineate logical behavior precisely, while managing the physical dependencies on other subordinate components.
- The C++ language itself lacks a mechanism to characterize and render software at a sufficiently high level of logical and physical abstraction.

Purpose of this Talk

Purpose of this Talk

1. Review the basics of component-based design:

Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams

Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams
   - Implied Dependency and Level Numbers

Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams
   - Implied Dependency and Level Numbers
   - The Two Most Important Physical Design Rules

Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams
   - Implied Dependency and Level Numbers
   - The Two Most Important Physical Design Rules
   - Guidelines for Collocating Classes in a Component

Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams
   - Implied Dependency and Level Numbers
   - The Two Most Important Physical Design Rules
   - Guidelines for Collocating Classes in a Component
   - Logical *Encapsulation* Versus Physical *Insulation*

Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams
   - Implied Dependency and Level Numbers
   - The Two Most Important Physical Design Rules
   - Guidelines for Collocating Classes in a Component
   - Logical *Encapsulation* Versus Physical *Insulation*
   - When to `#include` a Header File in a Header

Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams
   - Implied Dependency and Level Numbers
   - The Two Most Important Physical Design Rules
   - Guidelines for Collocating Classes in a Component
   - Logical *Encapsulation* Versus Physical *Insulation*
   - When to `#include` a Header File in a Header
   - Our Three-Level Physical-Packaging Hierarchy

Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams
   - Implied Dependency and Level Numbers
   - The Two Most Important Physical Design Rules
   - Guidelines for Collocating Classes in a Component
   - Logical *Encapsulation* Versus Physical *Insulation*
   - When to `#include` a Header File in a Header
   - Our Three-Level Physical-Packaging Hierarchy
2. Introduce the notion of a new C++ language entity, `module`…

Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams
   - Implied Dependency and Level Numbers
   - The Two Most Important Physical Design Rules
   - Guidelines for Collocating Classes in a Component
   - Logical *Encapsulation* Versus Physical *Insulation*
   - When to `#include` a Header File in a Header
   - Our Three-Level Physical-Packaging Hierarchy
2. Introduce the notion of a new C++ language entity, `module`, and describe it in terms of the essential engineering requirements it must fulfill if it is to be readily adopted widely by industry.

Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Introduce the Notion of a `module` in C++

   Requirements: Comparison with Conventional Headers

3. Achieving Physical Aggregation in C++ Today

   Organizing Components into Packages and Package Groups

4. Packaging Libraries Using C++ Modules

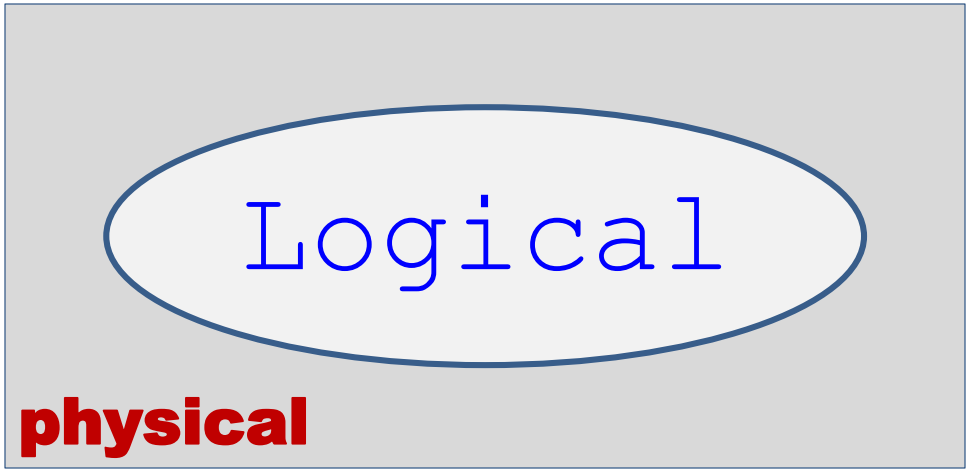
   Abstraction: Providing Refined Views on Existing Software

Outline

1. Review of Elementary Physical Design
   - Components, Modularity, Physical Dependencies
2. Introduce the Notion of a `module` in C++
   - Requirements: Comparison with Conventional Headers
3. Achieving Physical Aggregation in C++ Today
   - Organizing Components into Packages and Package Groups
4. Packaging Libraries Using C++ Modules
   - Abstraction: Providing Refined Views on Existing Software

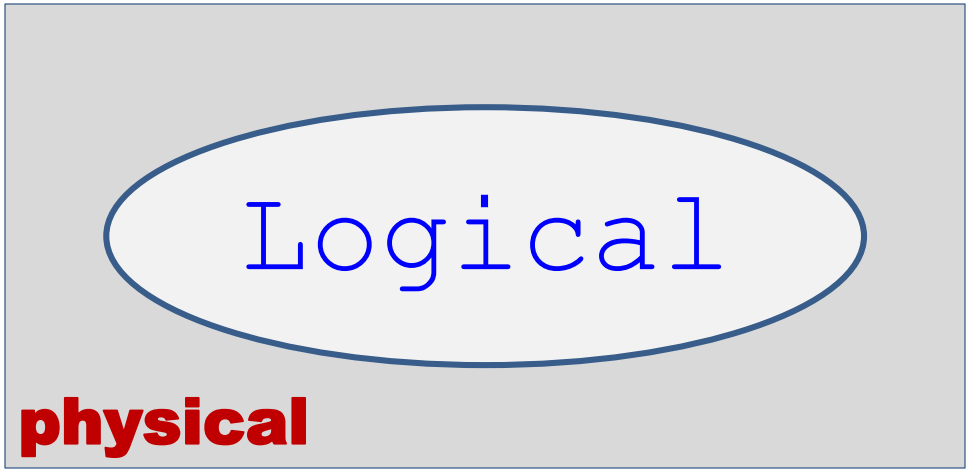
Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?



Logical versus Physical Design

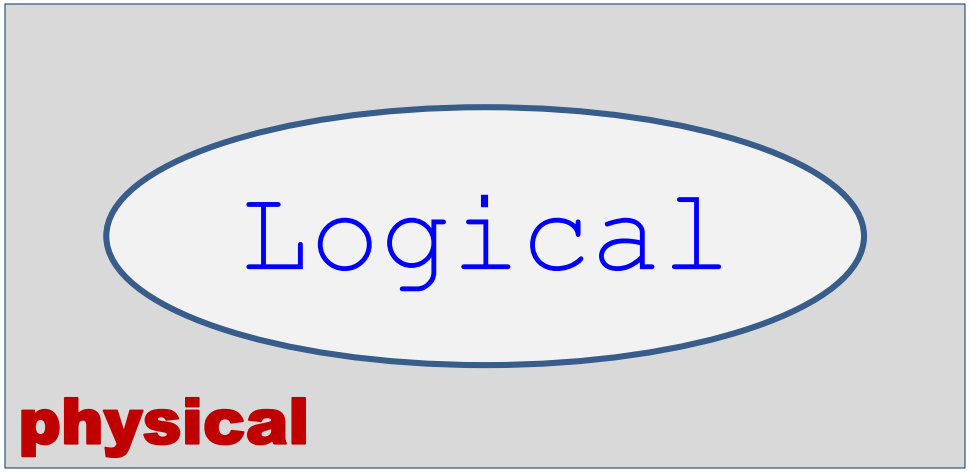
What distinguishes *Logical* from *Physical* Design?



Logical: Classes and Functions

Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?

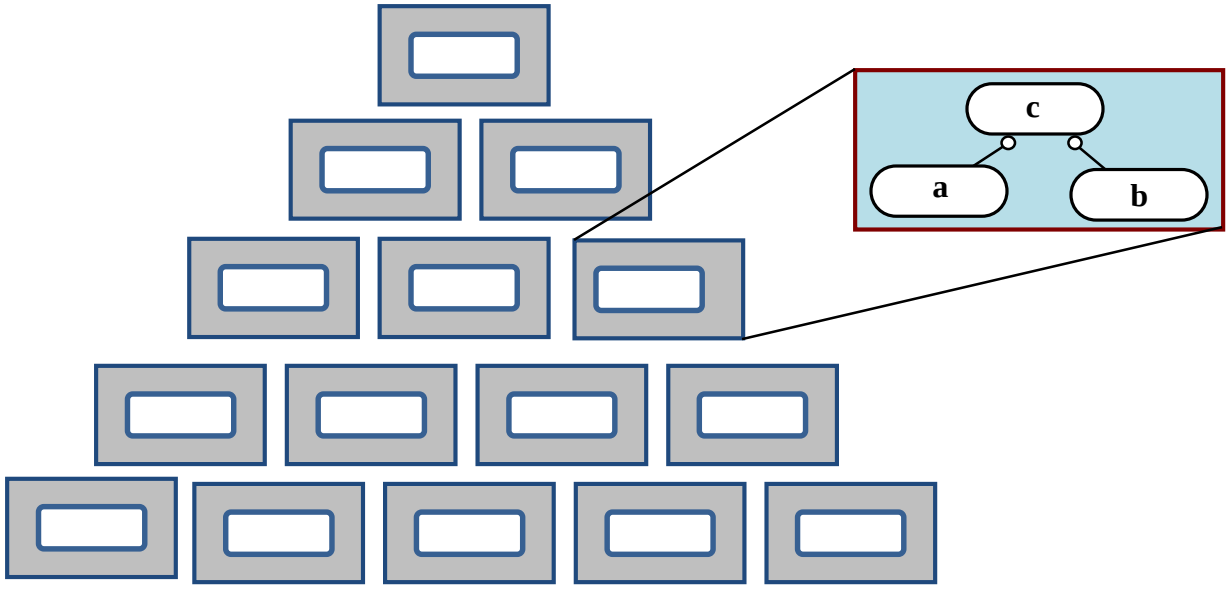


Logical: Classes and Functions

Physical: Files and Libraries

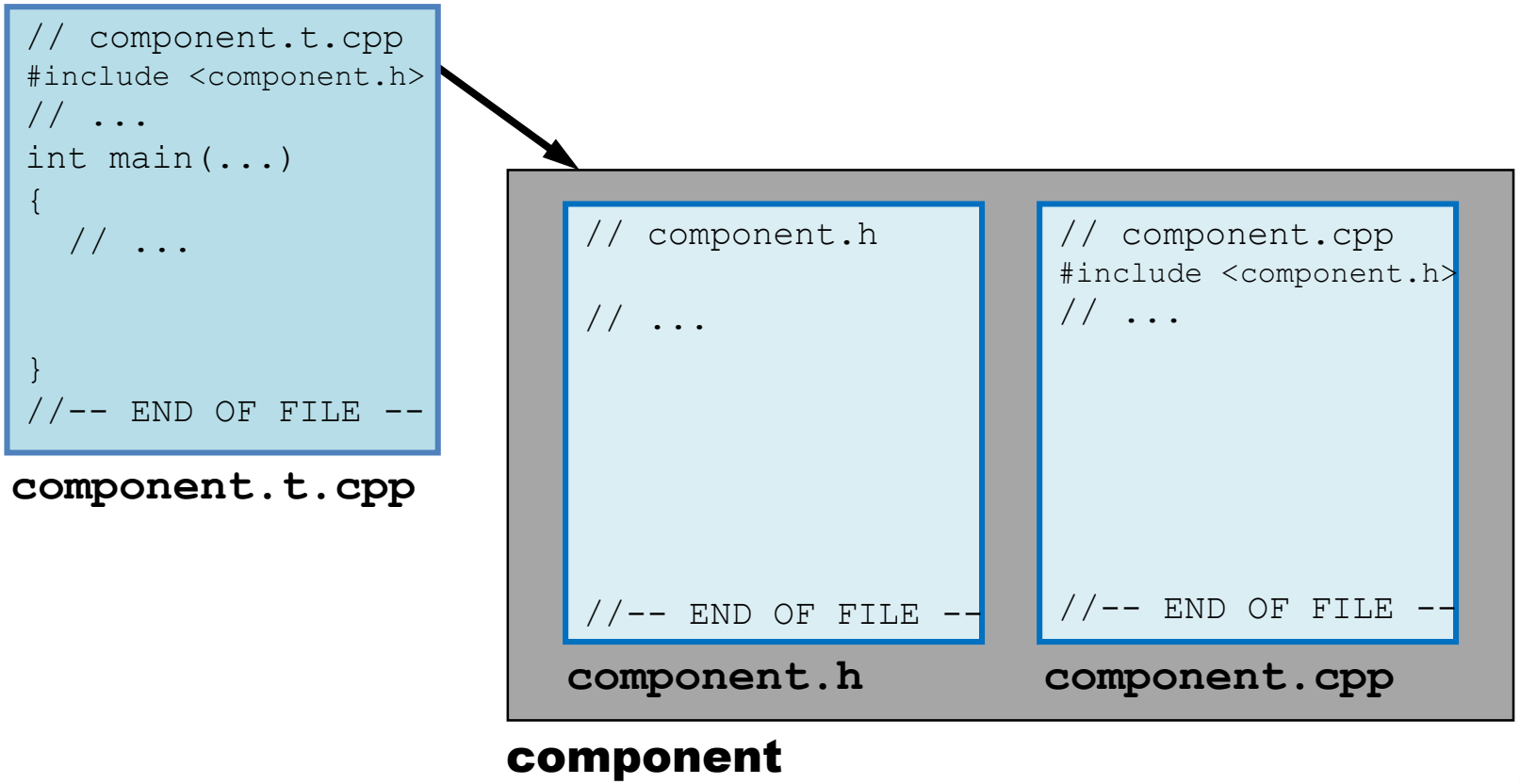
Logical versus Physical Design

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

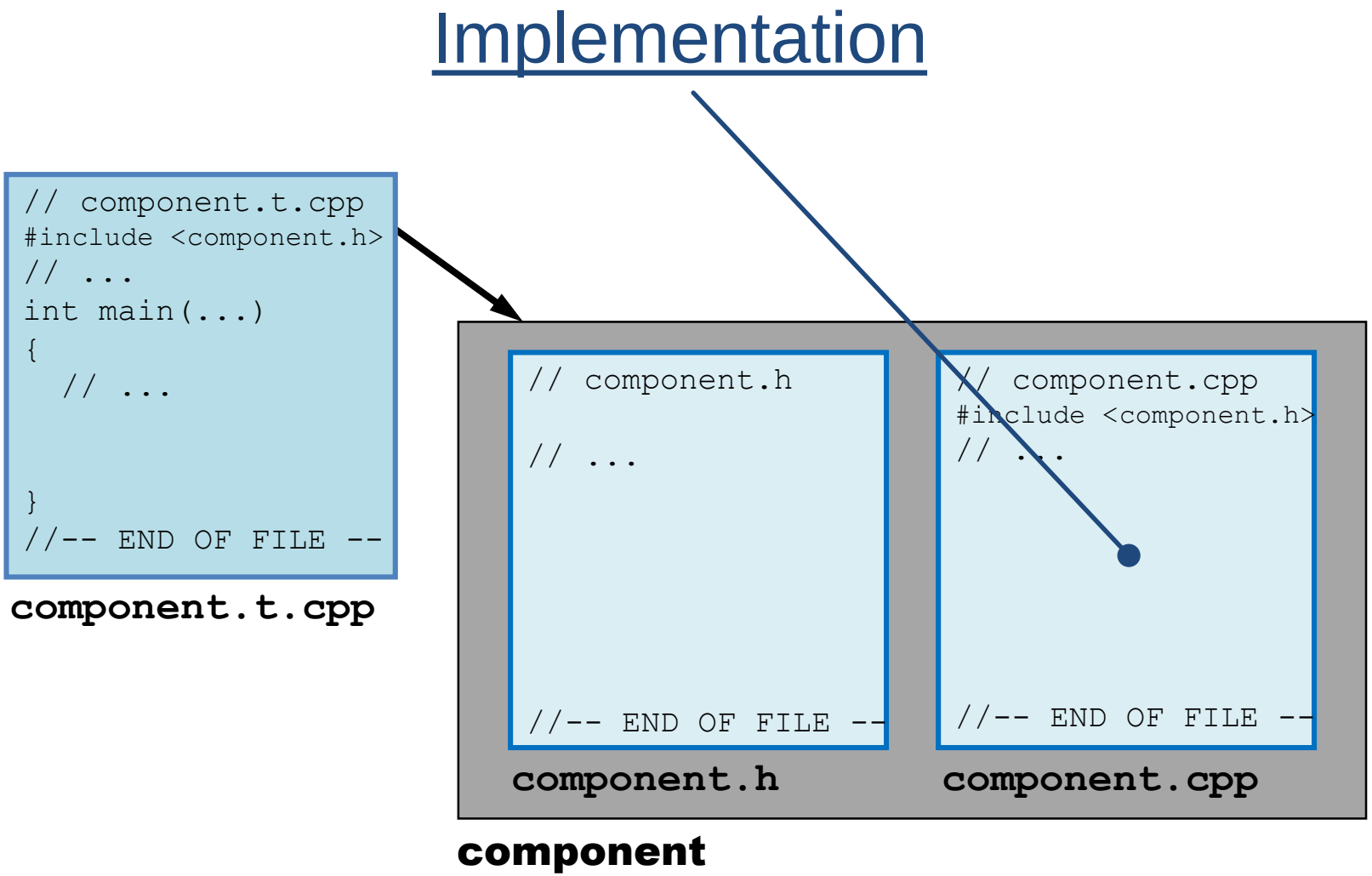


Component: Uniform Physical Structure

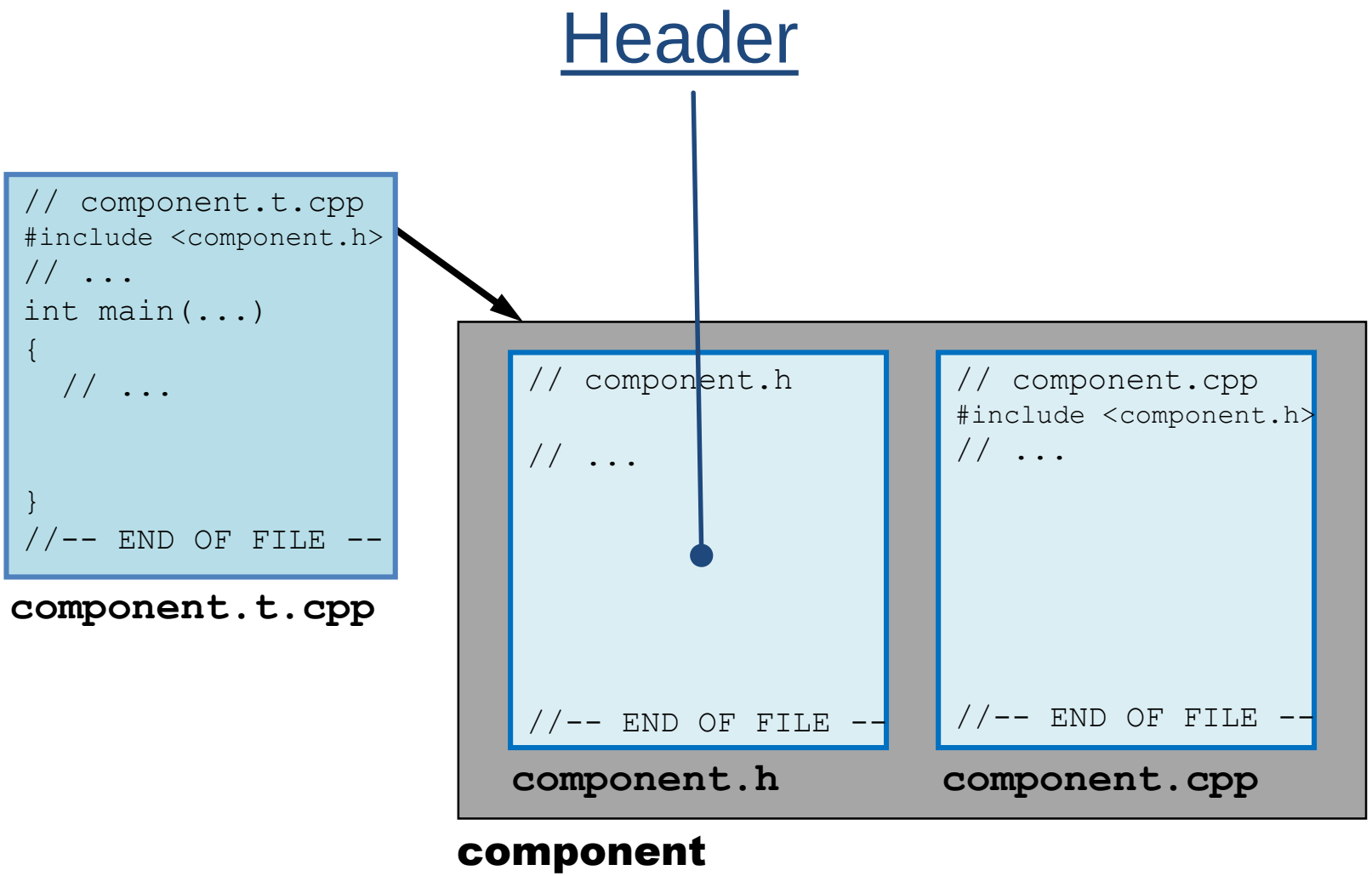
A Component Is Physical



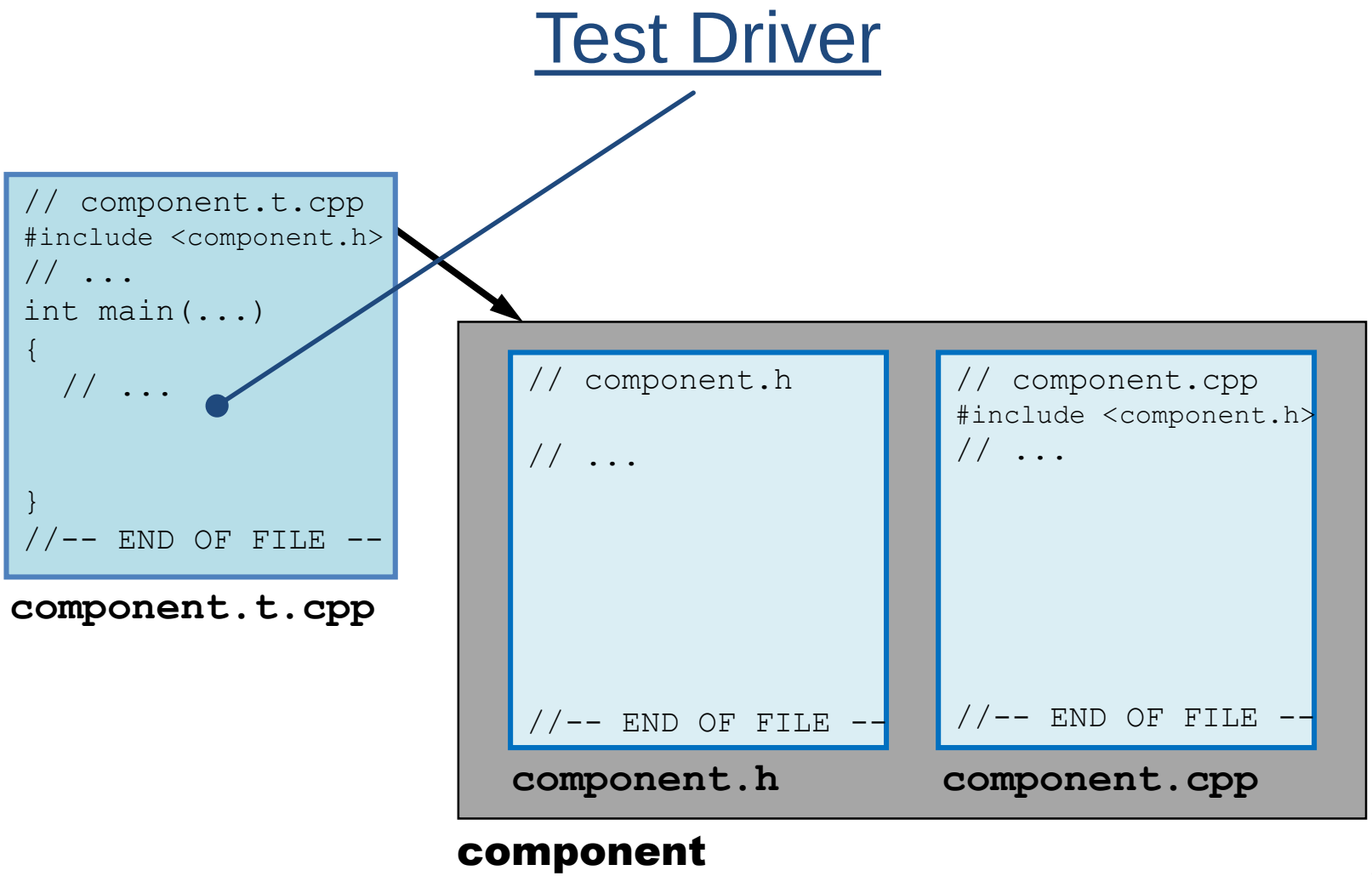
Component: Uniform Physical Structure



Component: Uniform Physical Structure

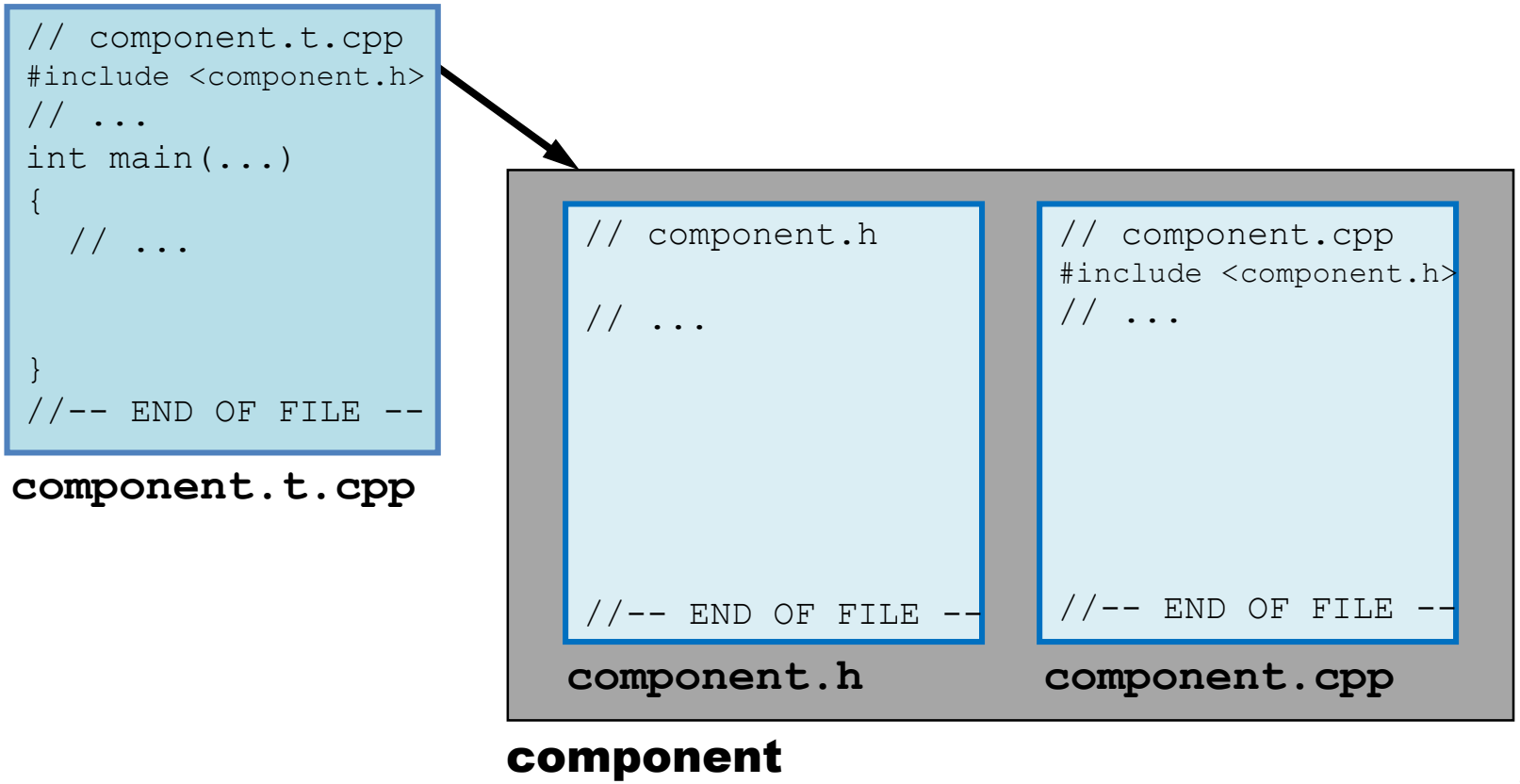


Component: Uniform Physical Structure

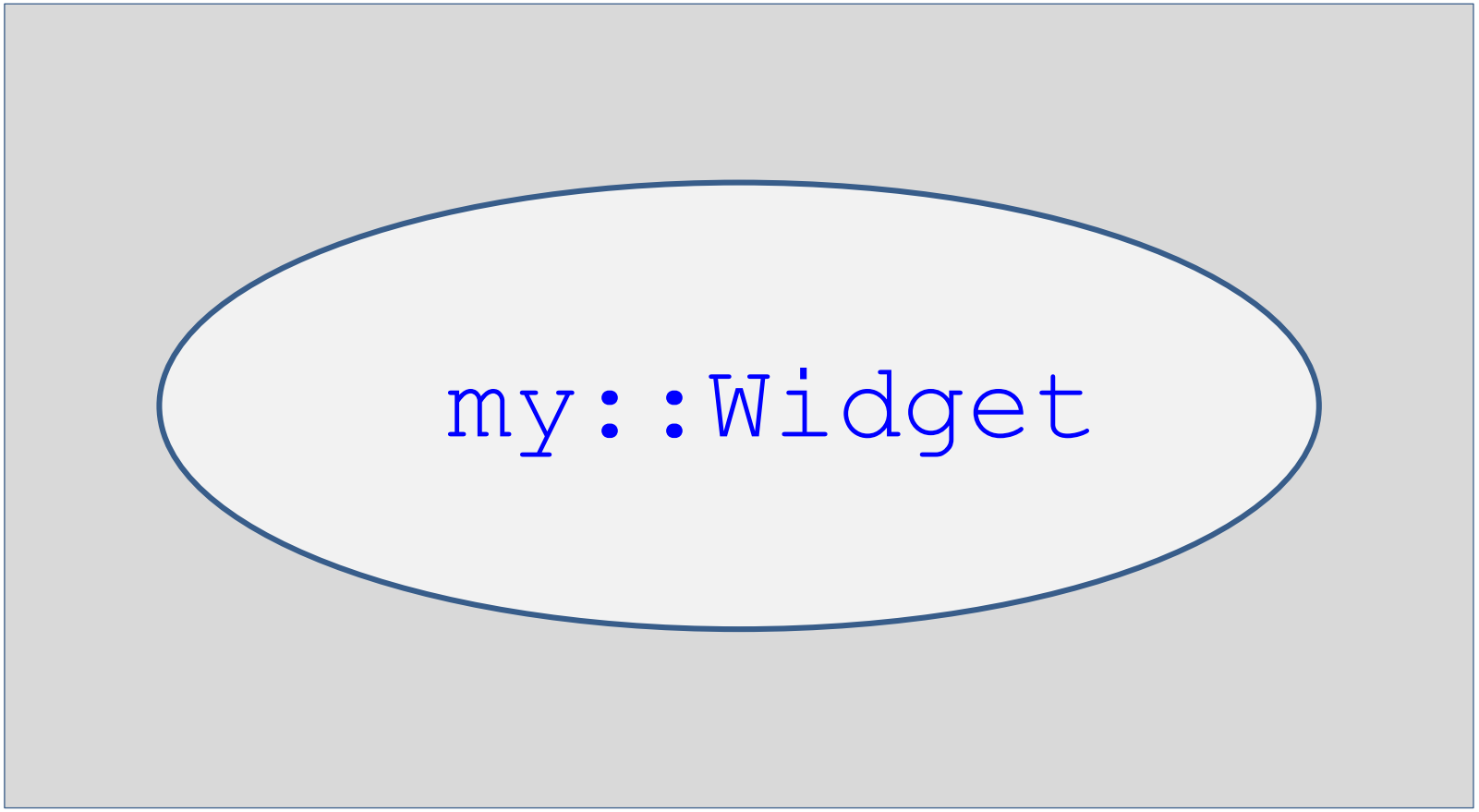


Component: Uniform Physical Structure

The Fundamental Unit of Design



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



`my_widget`

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

There are four Properties…

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

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

EVEN IF THE .CPP IS OTHERWISE EMPTY!

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

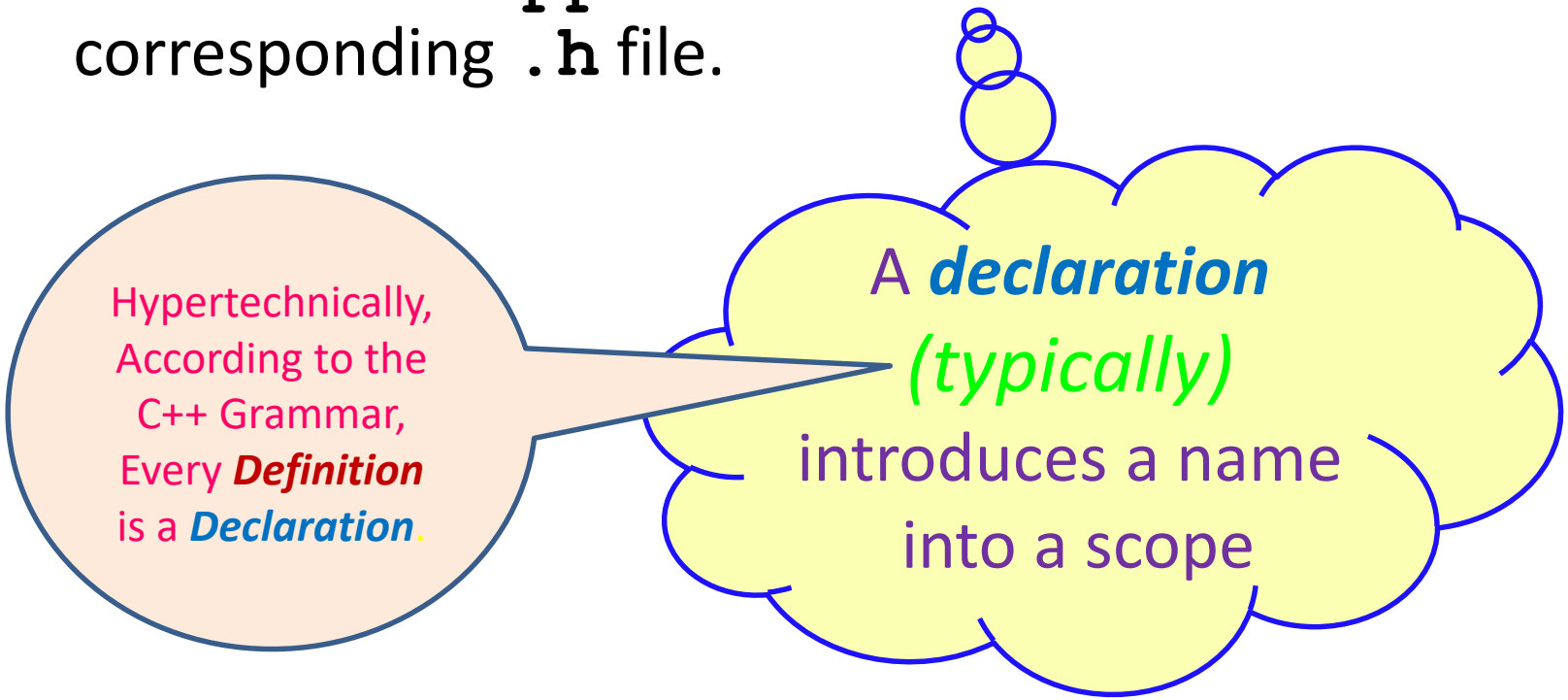
1.  The `.cpp` file includes its `.h` file as the first substantive line of code.
2.  All logical constructs having external *linkage* defined in a `.cpp` file are declared in the corresponding `.h` file.

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 having external *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 having external *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 having external *linkage* defined in a `.cpp` file are declared in the corresponding `.h` file.

For our purposes

A ***declaration*** introduces a name* into a scope

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 having external *linkage* defined in a `.cpp` file are declared in the corresponding `.h` file.

```
int a;                  // Declaration And Definition
extern int a;           // Declaration Only
extern int a = 0;       // Declaration And Definition
void f();               // Declaration Only
void f(){}              // Declaration And Definition
class Foo;              // Declaration Only
class Foo { // …        // Declaration And Definition
    static int d_s; // Declaration Only
} object;               // Declaration And Definition
int Foo::d_s;           // Definition Only
```

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 having external *linkage* defined in a `.cpp` file are declared in the corresponding `.h` file.

```
int a;                              // External Linkage
static int a;                       // Internal Linkage
void f(){};                         // External Linkage
static void f(){}                   // Internal Linkage
inline void f(){}                   // External Linkage
static inline void f(){}            // Internal Linkage
class Foo { // …                    // External Linkage
  static int d_s;                   // External Linkage
} object;                           // External Linkage
int Foo::d_s;                       // External Linkage
```

Declaration

Definition

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 having external *linkage* defined in a **`.cpp`** file are declared in the corresponding **`.h`** file.

```
namespace { class Foo {          // Internal Linkage
  static int d_s;                // Internal Linkage
  } object; }                    // Internal Linkage
int Foo::d_s;                    // Internal Linkage
namepsace ns {                   // None (External-ish)
  class Foo {                    // External Linkage
    static int d_s;              // External Linkage
    } object; }                  // External Linkage
int ns::Foo::d_s;                // External Linkage
int Foo::d_s;                    // Doesn't Compile!
```

Declaration

Definition

Declaration

Definition

Definition

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 having external *linkage* defined in a **`.cpp`** file are declared in the corresponding **`.h`** file.

```
namespace { class Foo {          // Internal Linkage
Declaration  static int d_s;     // Internal Linkage
           } object; }           // Internal Linkage

namepsace ns {                   // None (External-ish)
            class Foo {          // External Linkage
Declaration  static int d_s;     // External Linkage
           } object; }           // External Linkage
Definition int ns::Foo::d_s;     // External Linkage
Definition int Foo::d_s;         // Doesn't Compile!
```

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 having external *linkage* defined in a **`.cpp`** file are declared in the corresponding **`.h`** file.

```
namespace { class Foo {          // Internal Linkage
Declaration static int d_s;      // Internal Linkage
          } object; }            // Internal Linkage

namepsace ns {                   // None (External-ish)
            class Foo {          // External Linkage
Declaration   static int d_s;    // External Linkage
            } object; }          // External Linkage
Definition int ns::Foo::d_s;     // External Linkage
Definition int Foo::d_s;         // Internal Linkage
```

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 having external *linkage* defined in a **`.cpp`** file are declared in the corresponding **`.h`** file.

```
namepsace ns {                    // None (External-ish)
   class Foo {                    // External Linkage
   static int d_s;                // External Linkage
   } object; }                    // External Linkage
int ns::Foo::d_s;                 // External Linkage
int Foo::d_s;                     // Internal Linkage
```

Declaration

Definition

Definition

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 having external *linkage* defined in a **`.cpp`** file are declared in the corresponding **`.h`** file.

```
namepsace ns {                    // None (External-ish)
   class Foo {                    // External Linkage
   static int d_s;                // External Linkage
   } object; }                    // External Linkage
int ns::Foo::d_s;                 // External Linkage
int Foo::d_s;                     // Doesn't Compile!
```

Declaration

Definition

Definition

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 having external *linkage* defined in a `.cpp` file are declared in the corresponding `.h` file.
3.  All constructs having external or dual *bindage* declared in a `.h` file (if defined at all) are defined within the component.

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

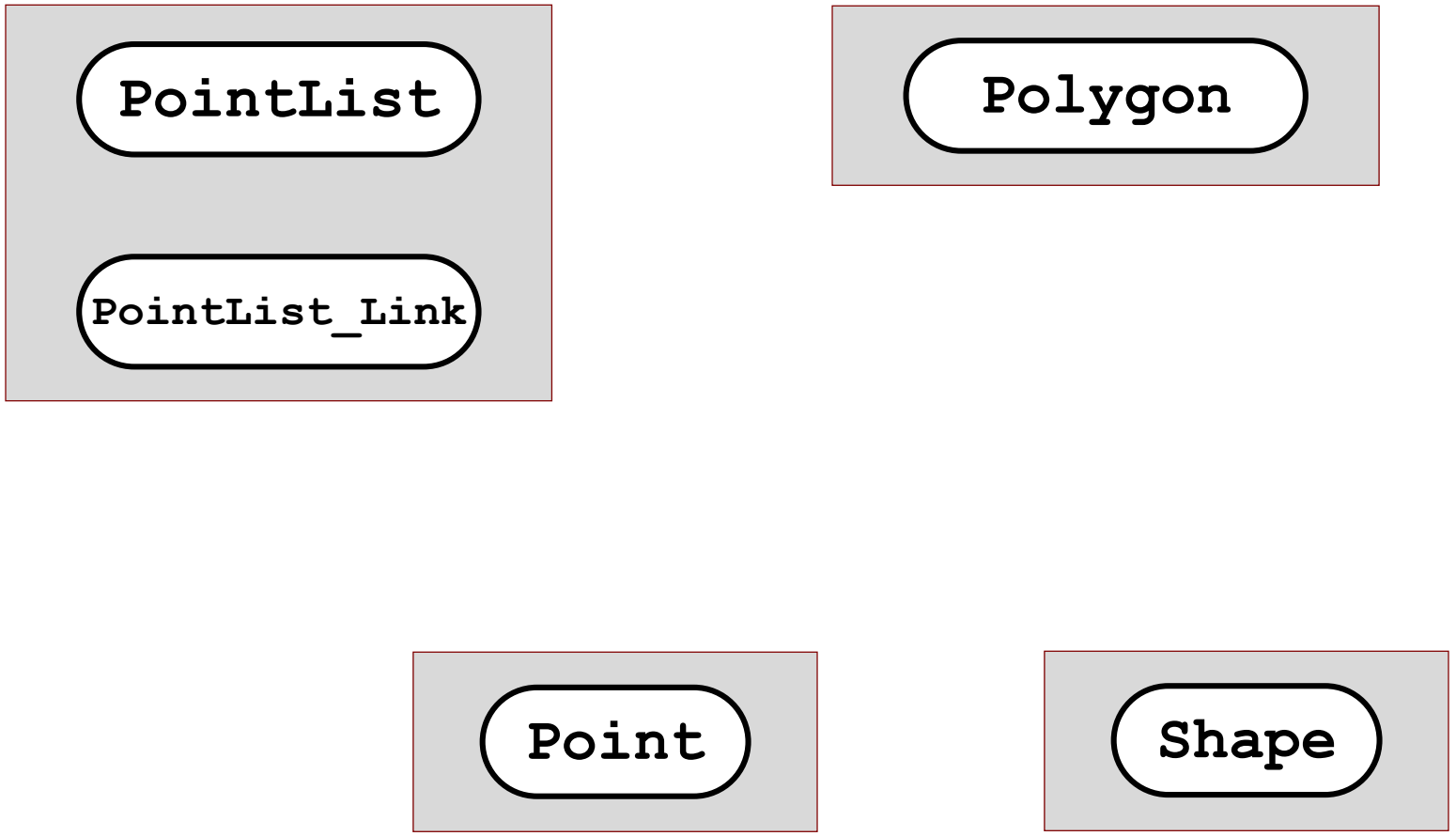
1.  The `.cpp` file includes its `.h` file as the first substantive line of code.
2.  All logical constructs having external *linkage* defined in a `.cpp` file are declared in the corresponding `.h` file.
3.  All constructs having external or dual *bindage* declared in a `.h` file (if defined at all)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

examples of bindage

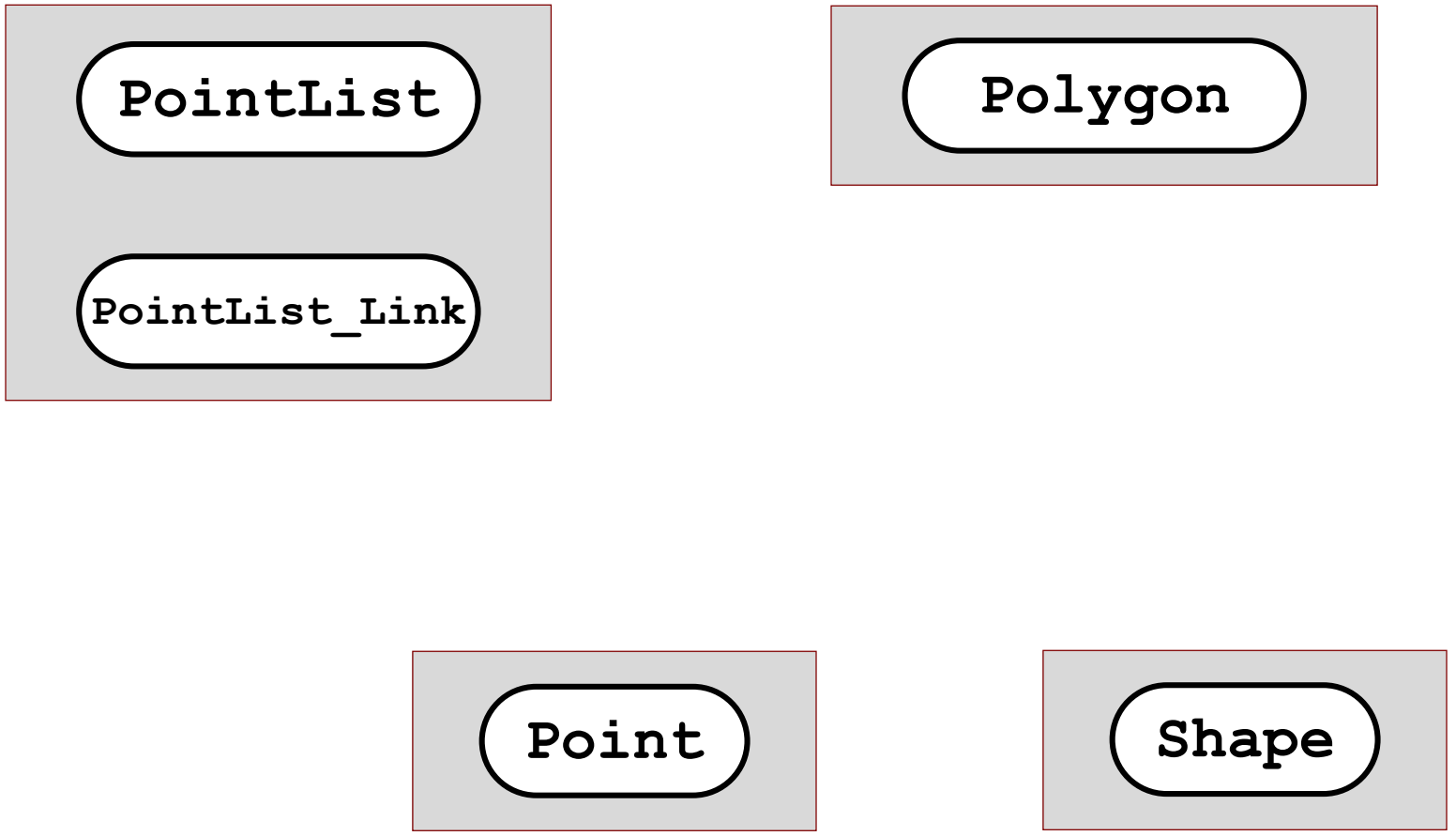
Component: Not Just a `.h`/`.cpp` Pair

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

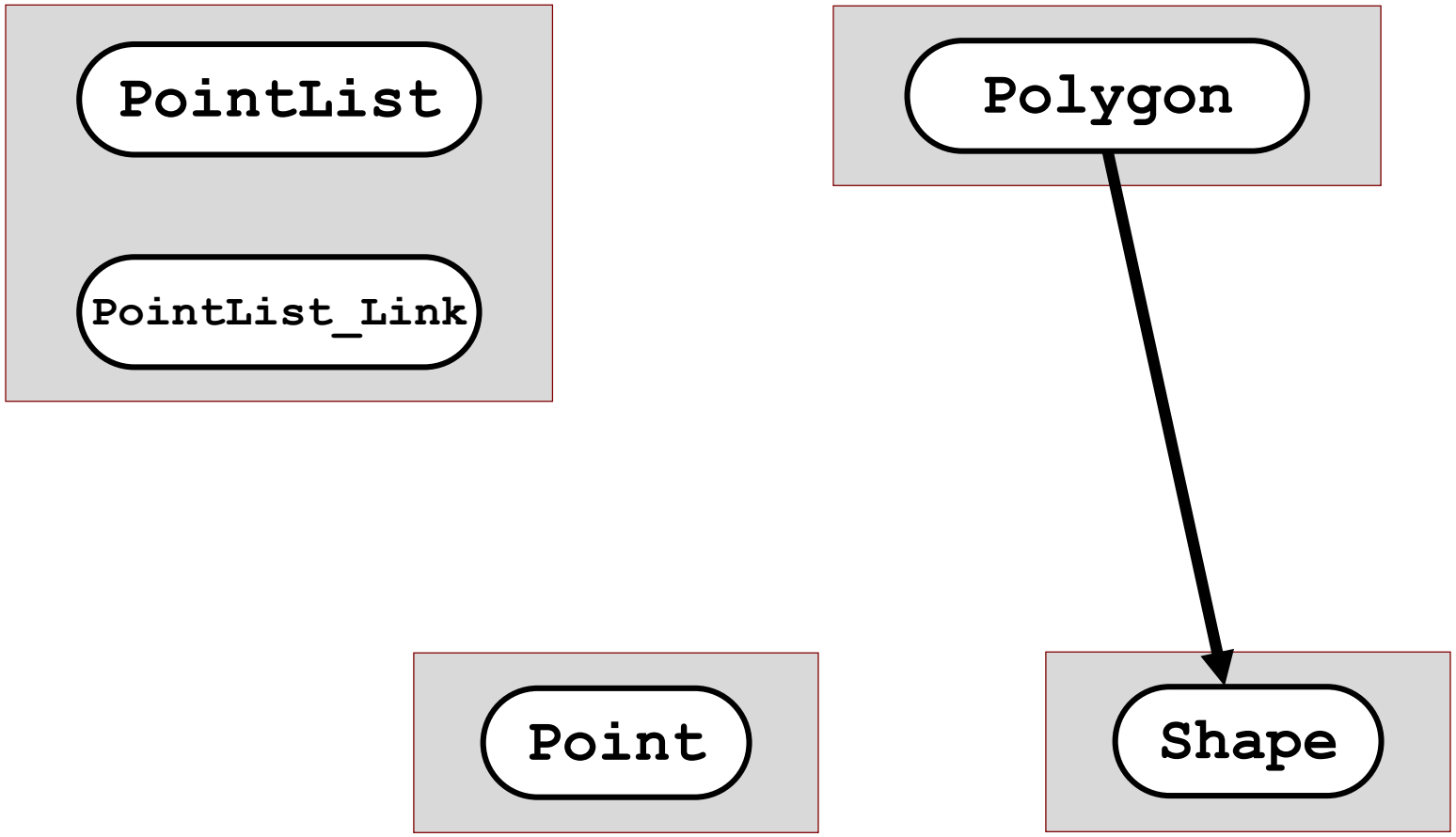
Logical Relationships



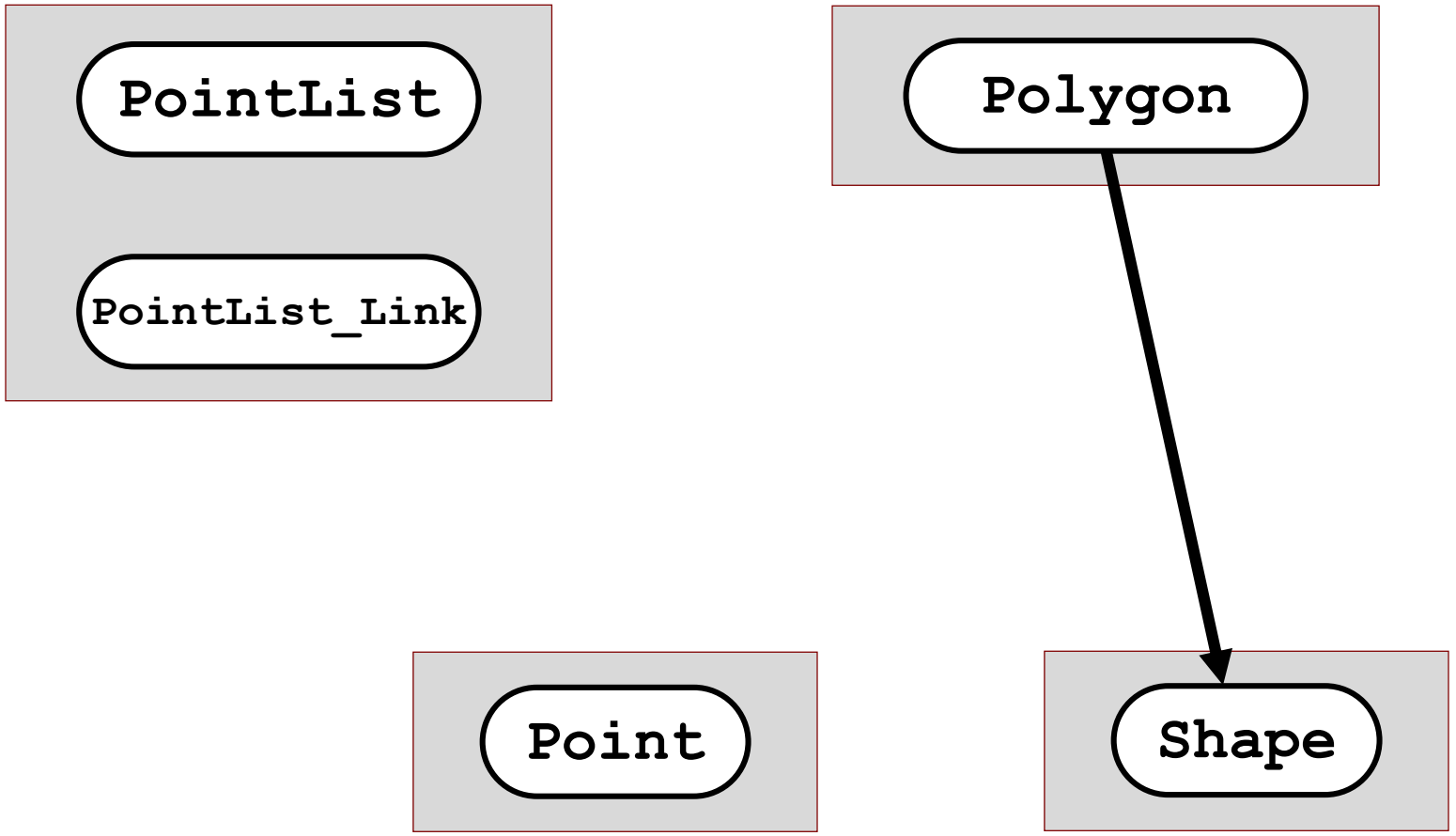
Logical Relationships



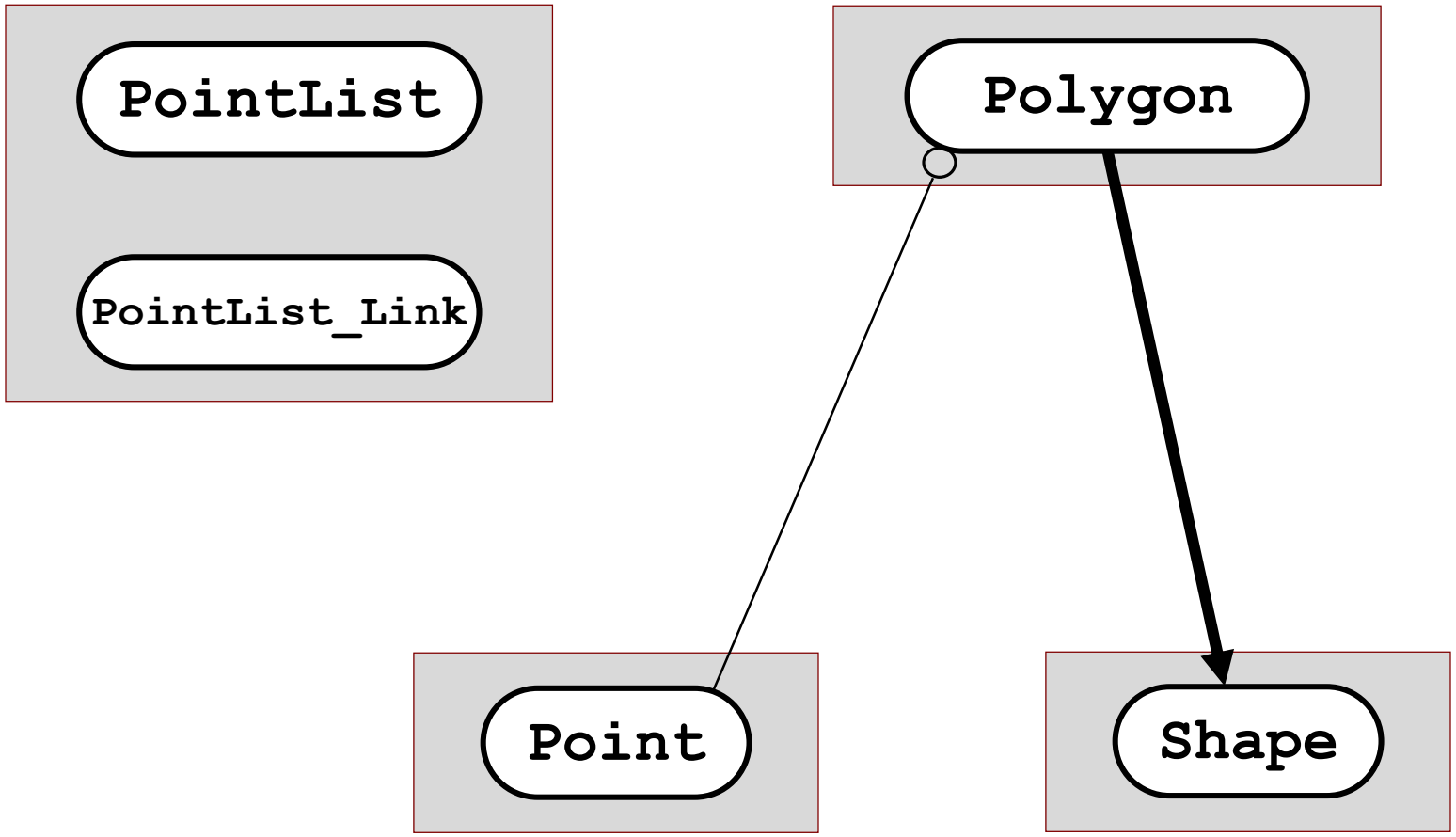
Logical Relationships



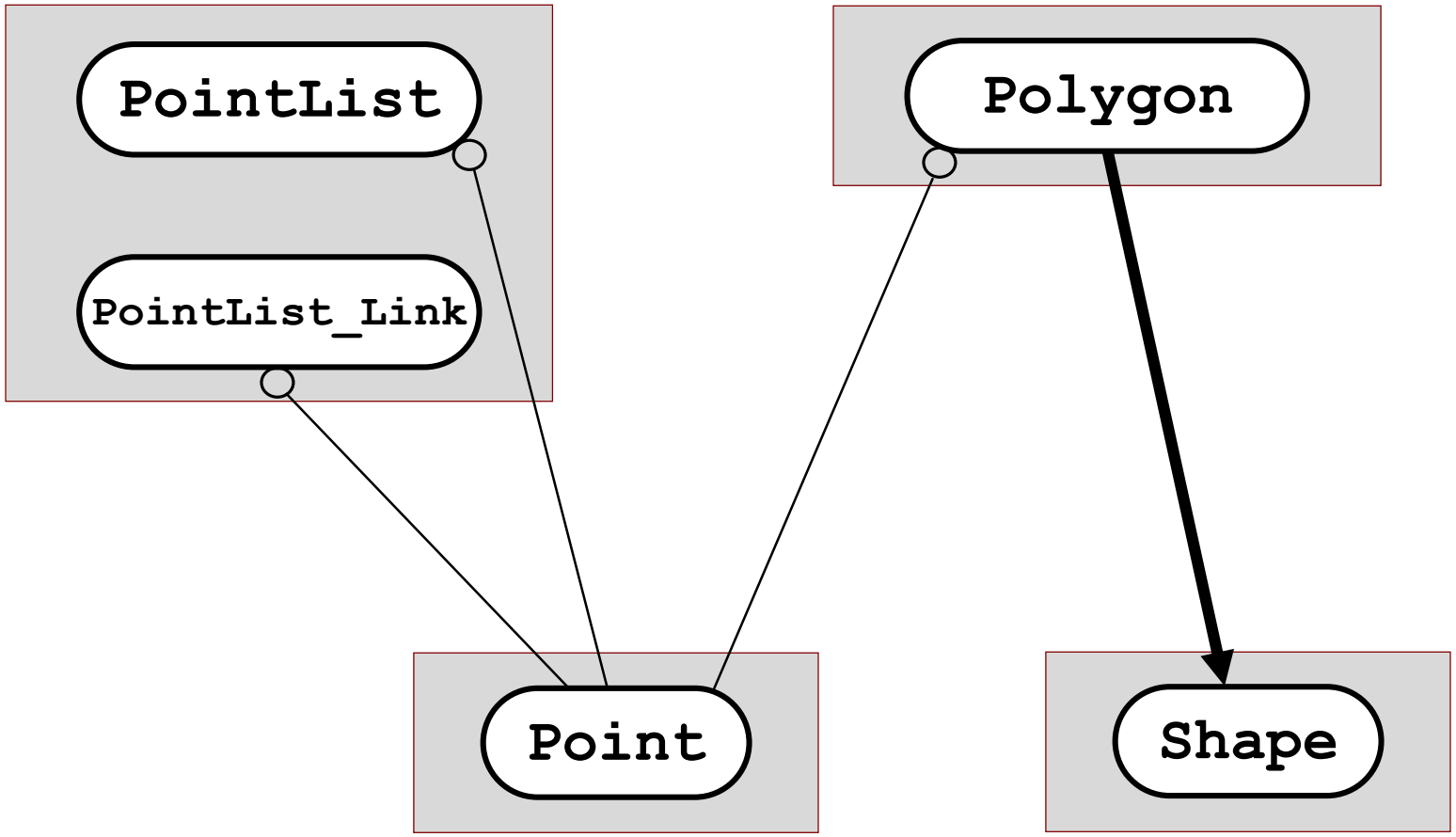
Logical Relationships



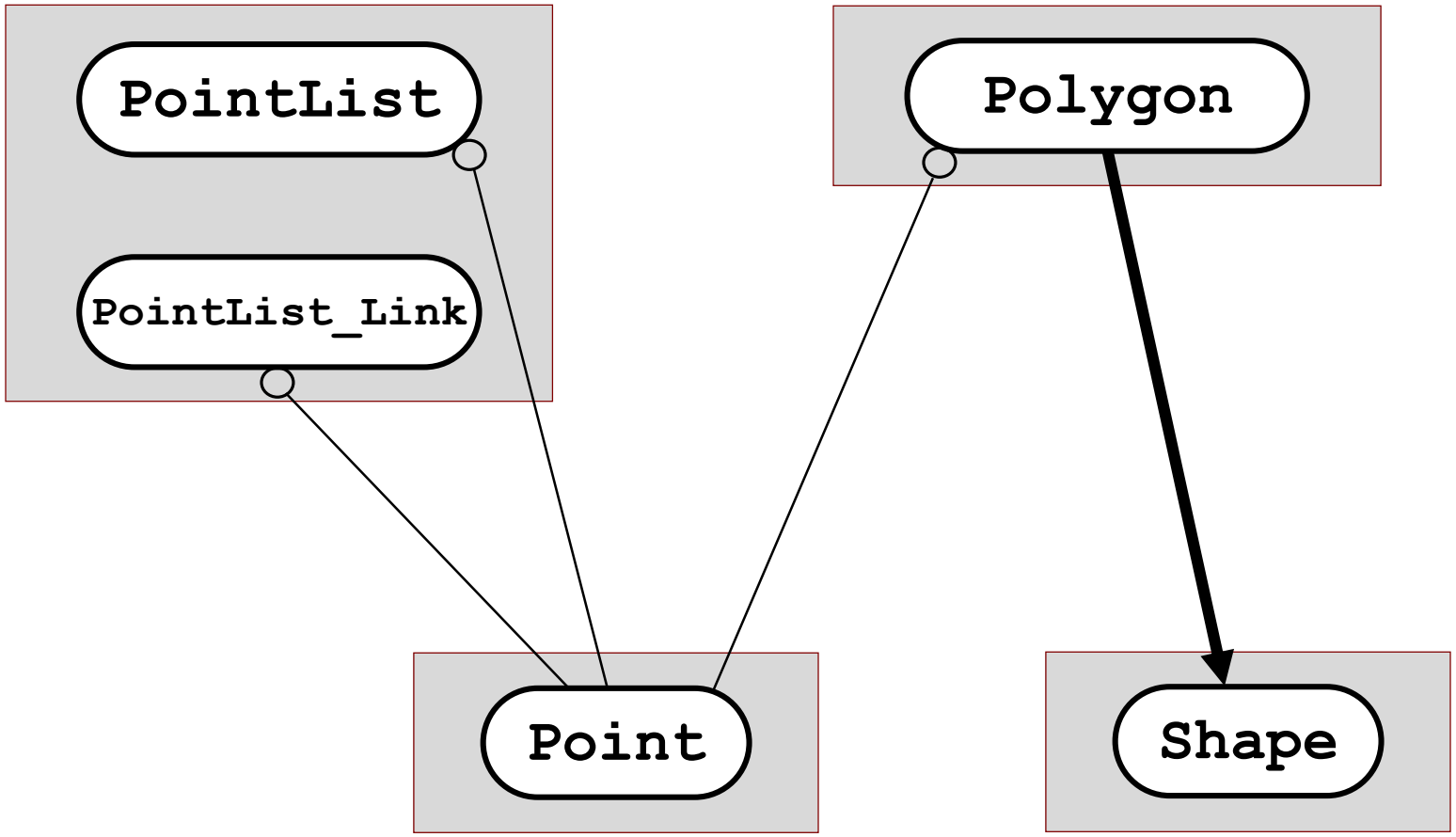
Logical Relationships



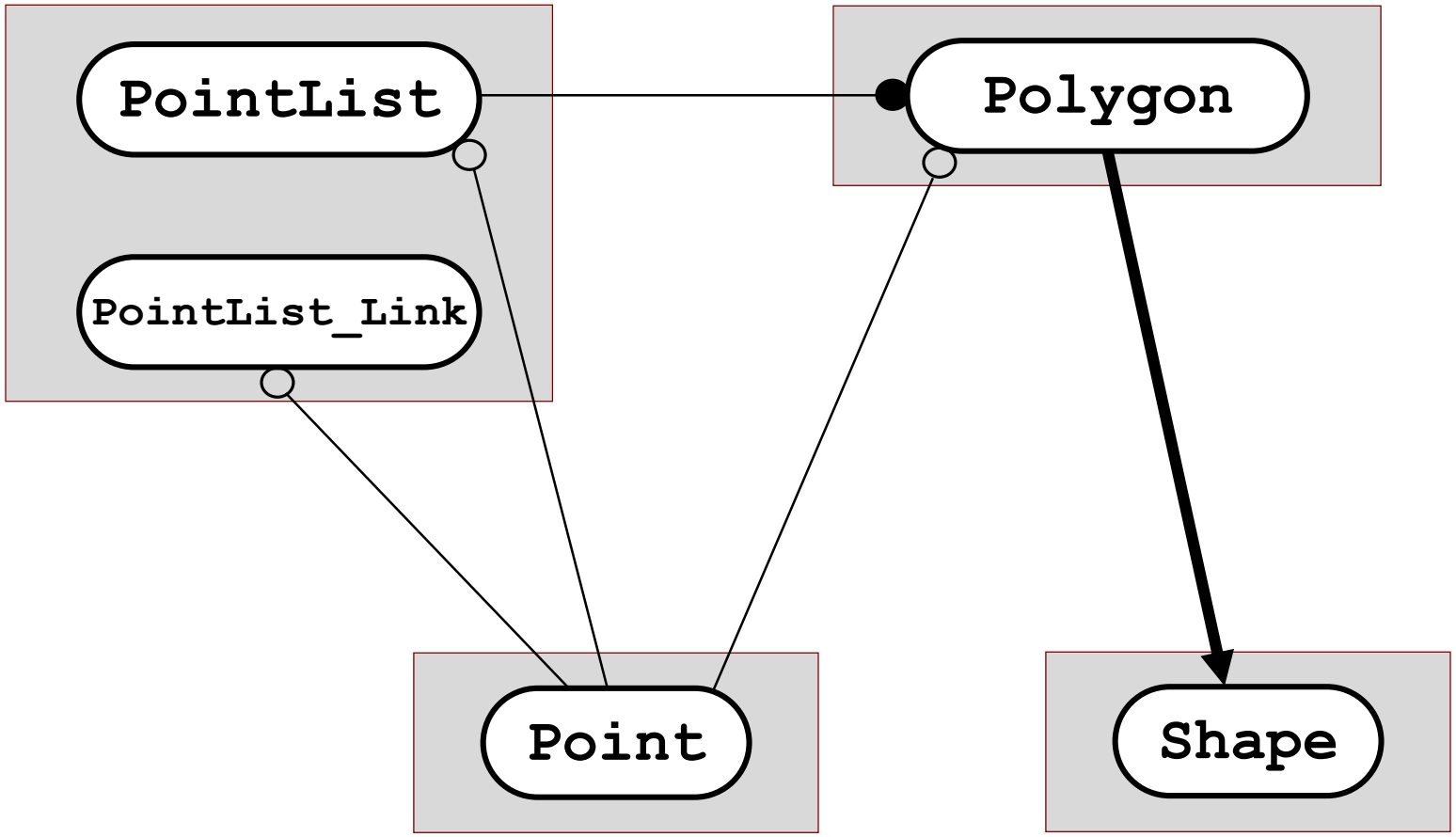
Logical Relationships



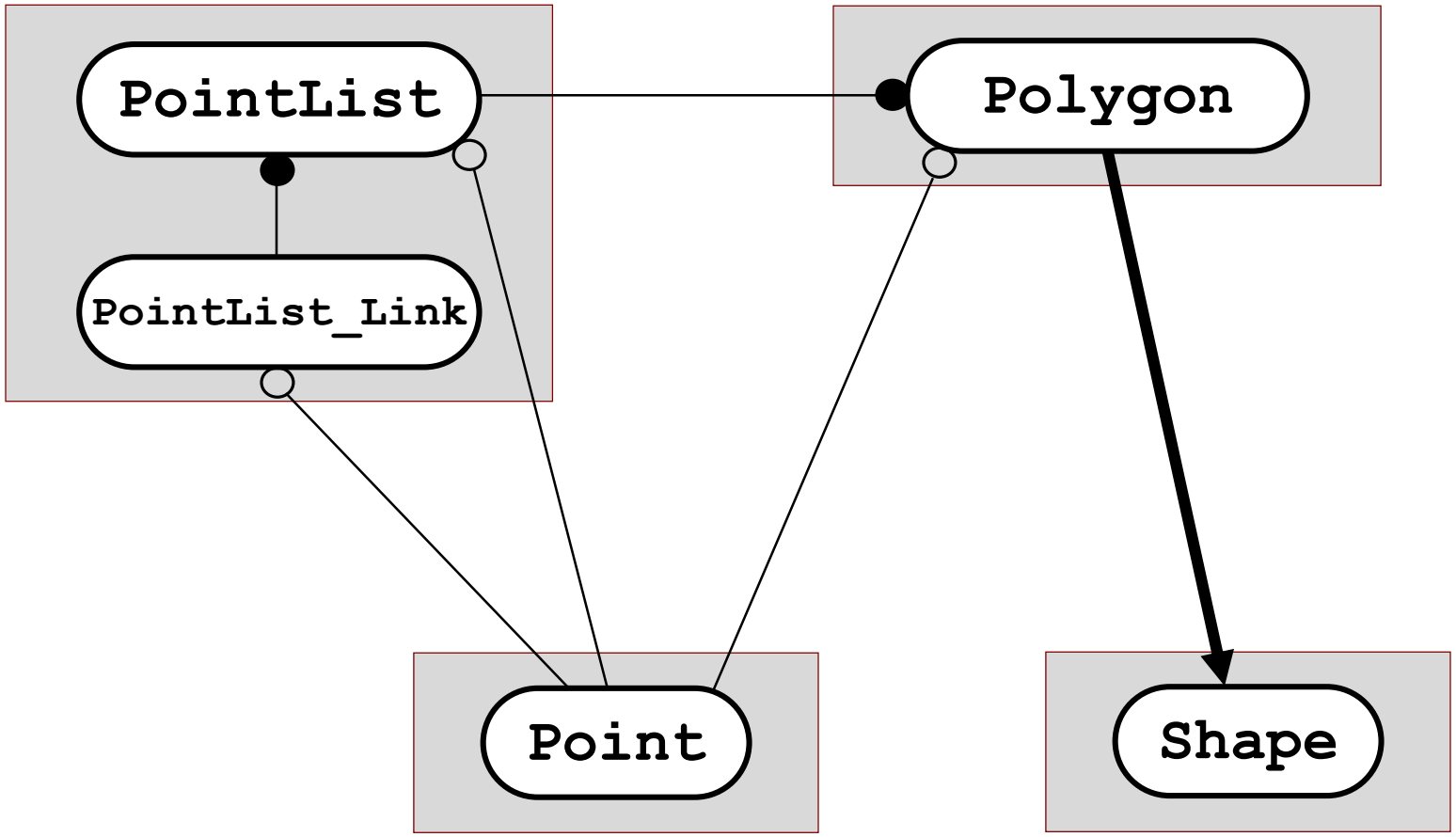
Logical Relationships



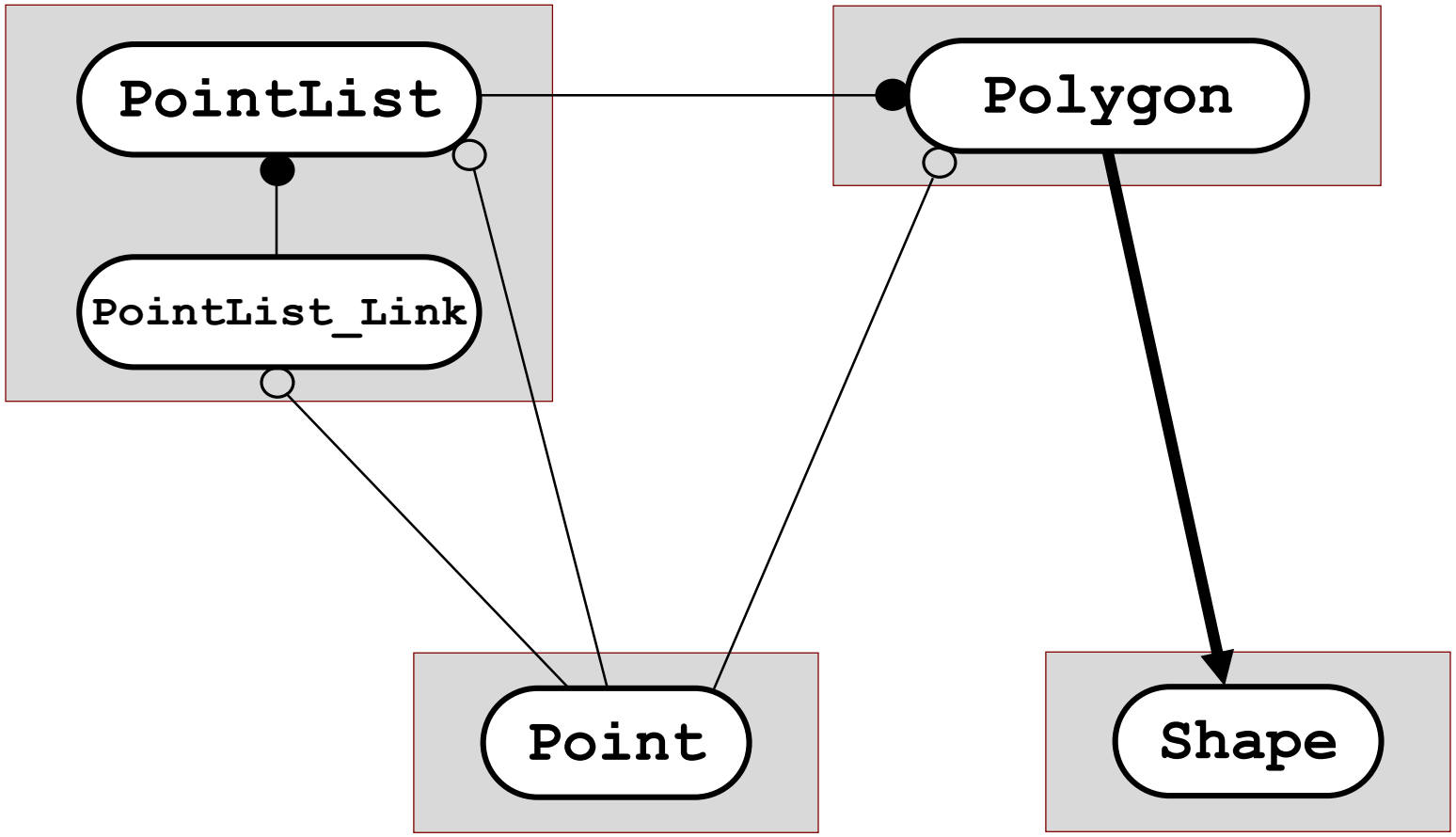
Logical Relationships



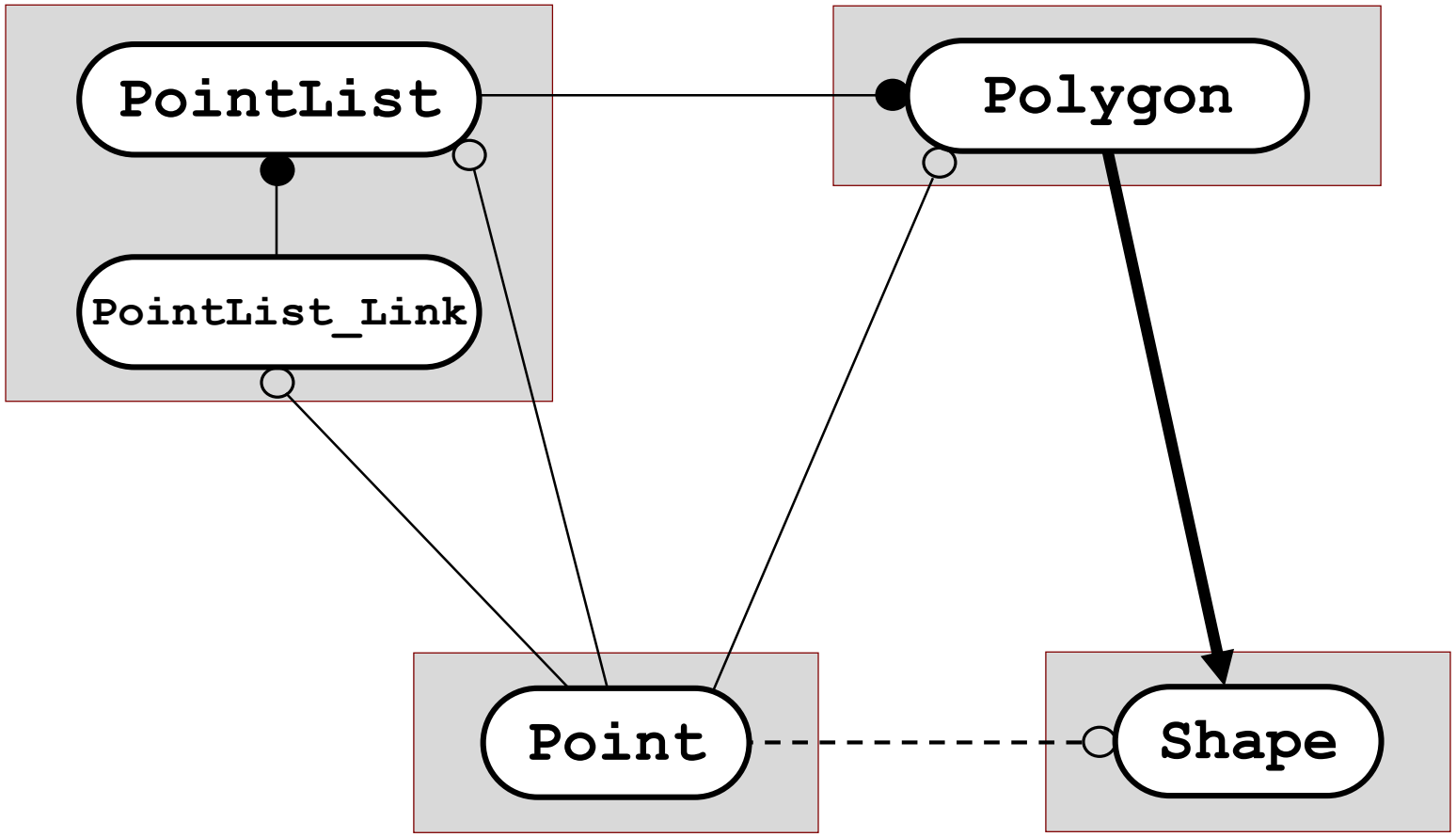
Logical Relationships



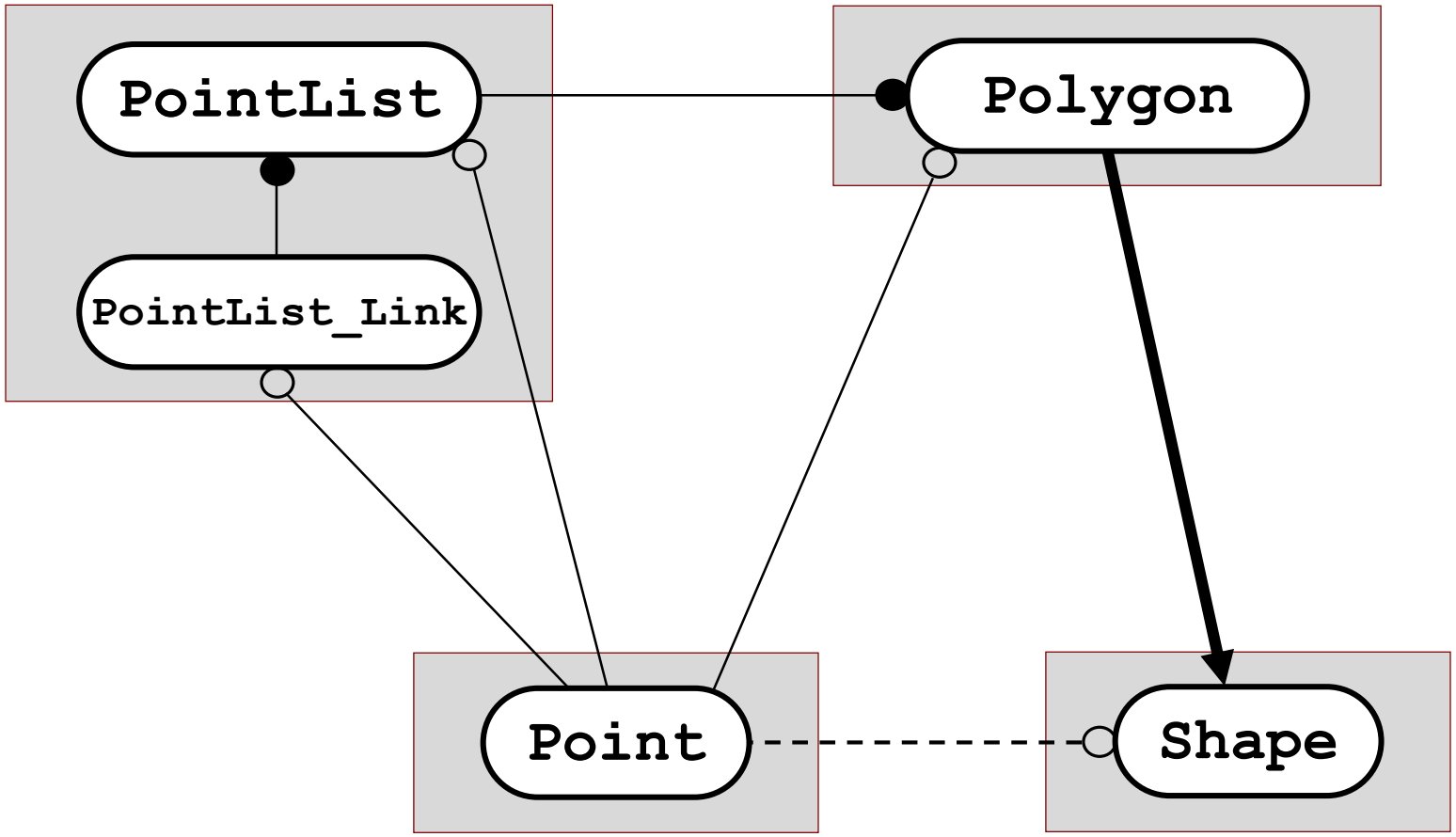
Logical Relationships



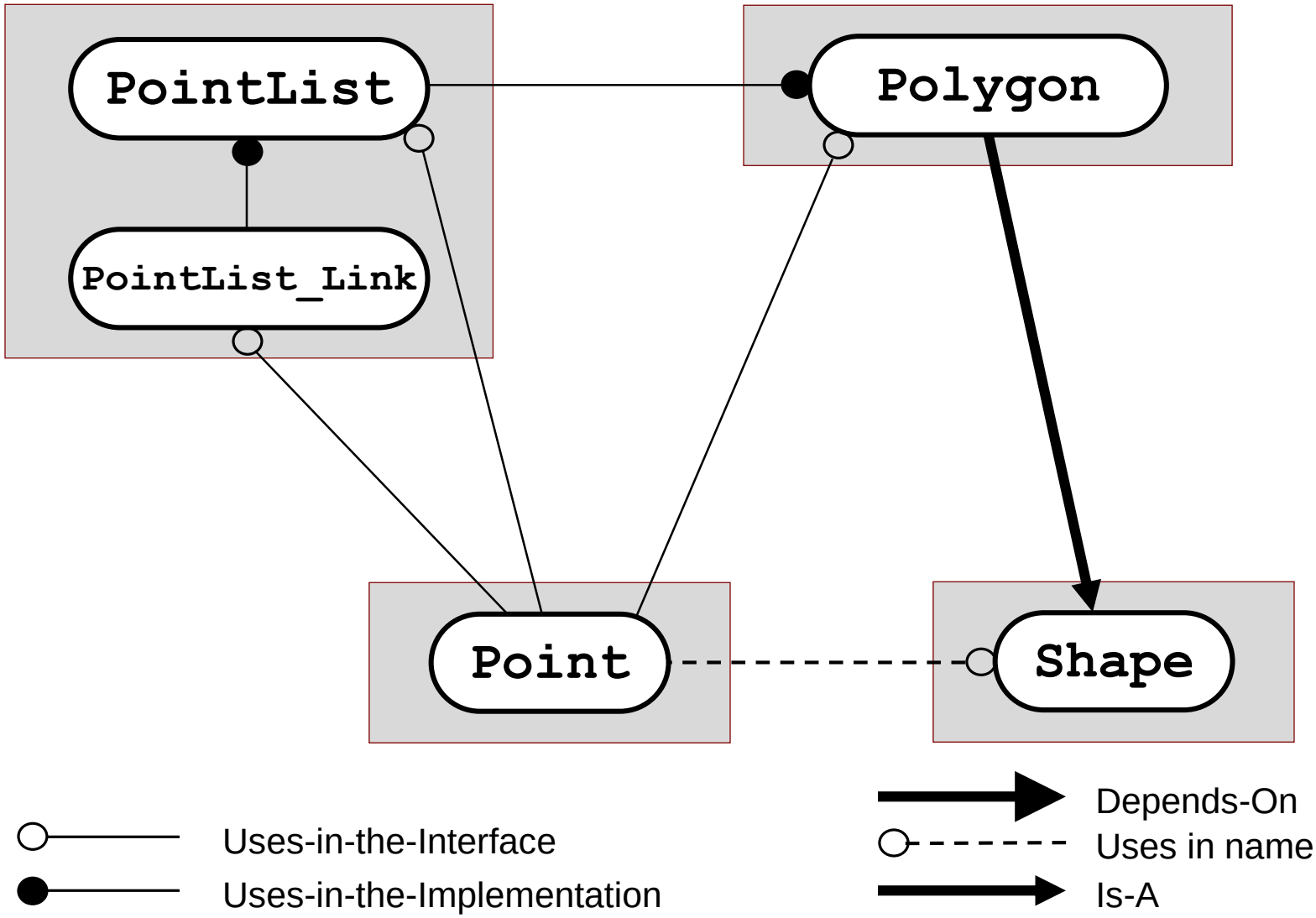
Logical Relationships



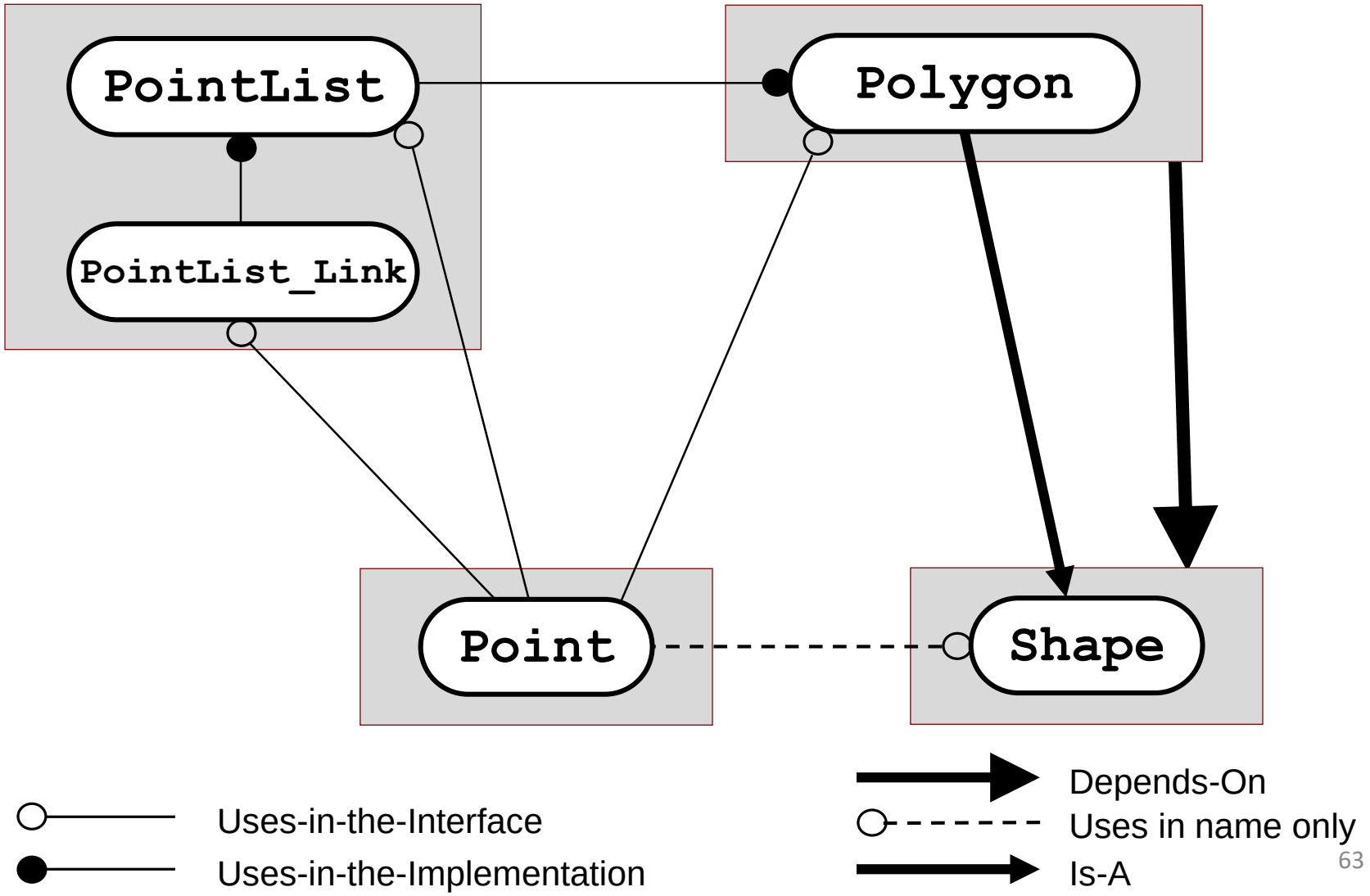
Implied Dependency



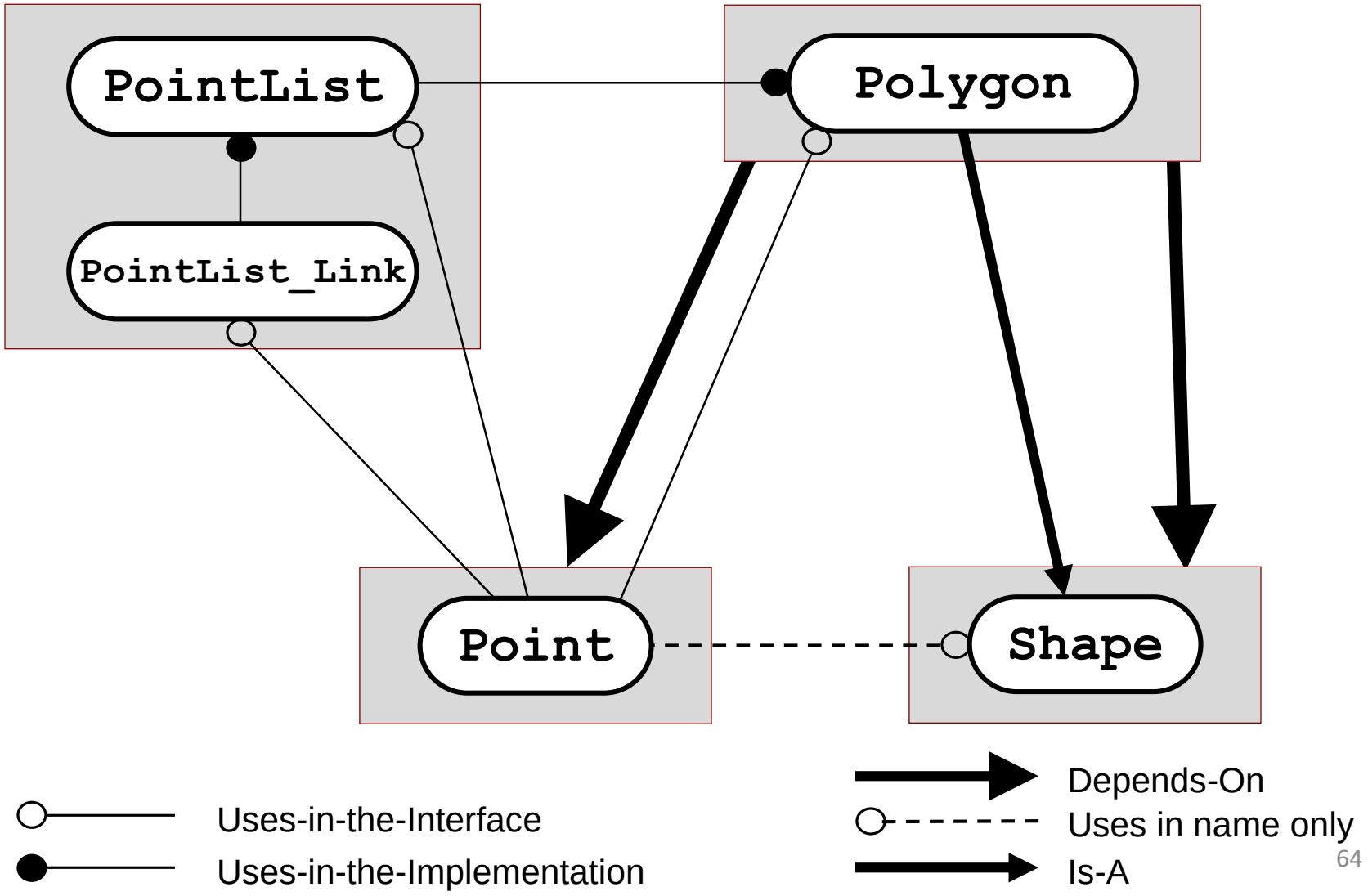
Implied Dependency



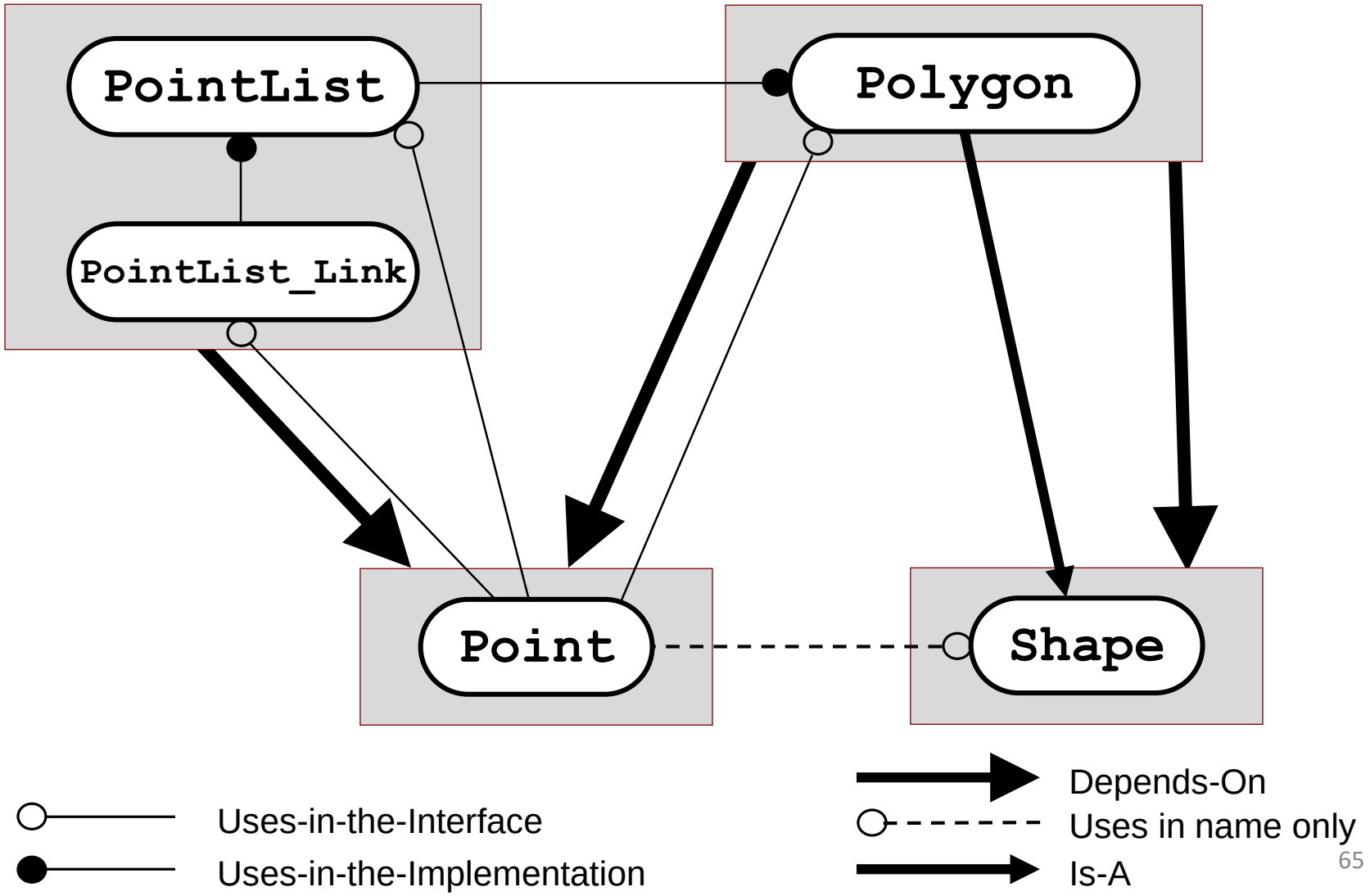
Implied Dependency



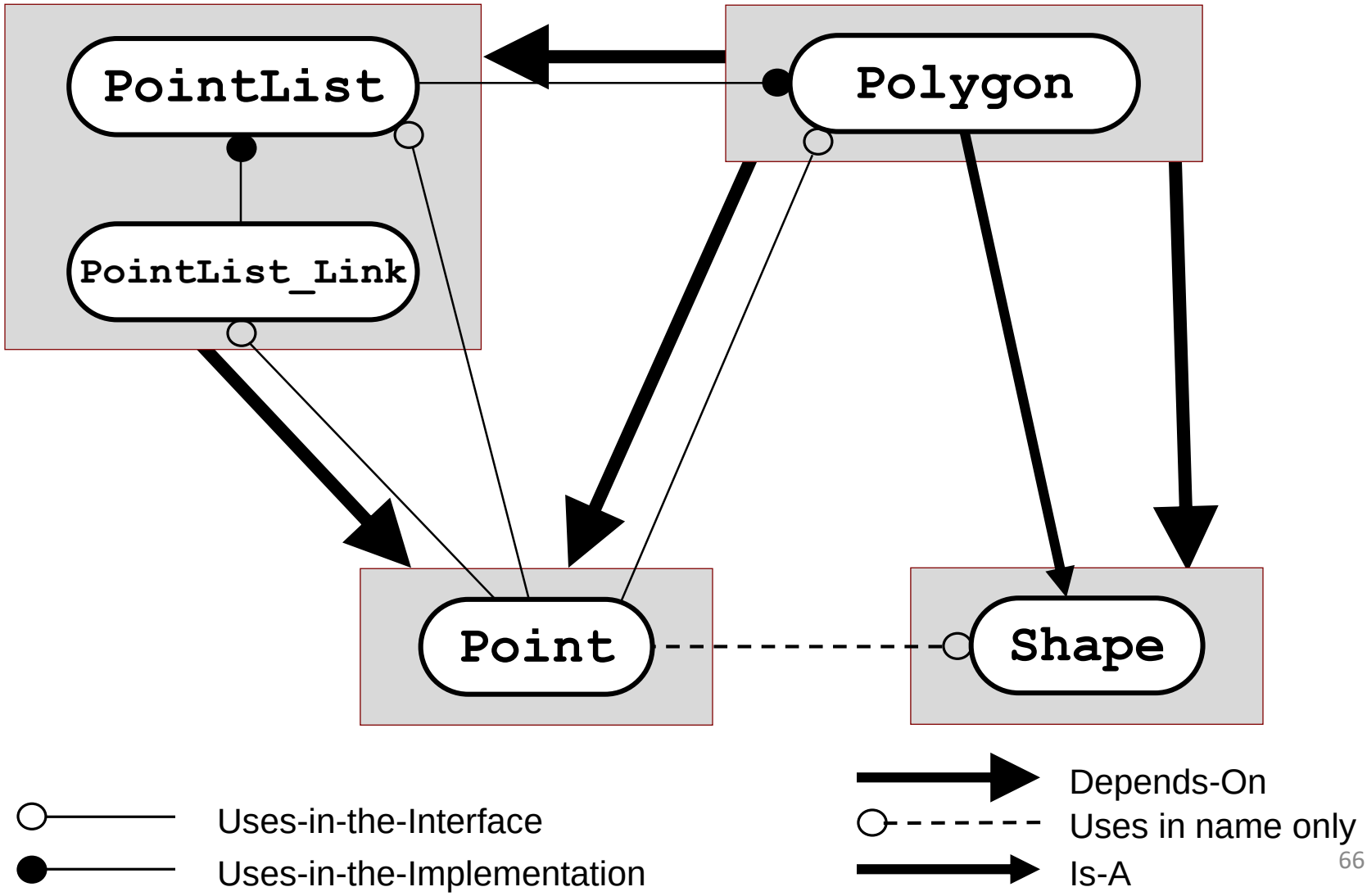
Implied Dependency



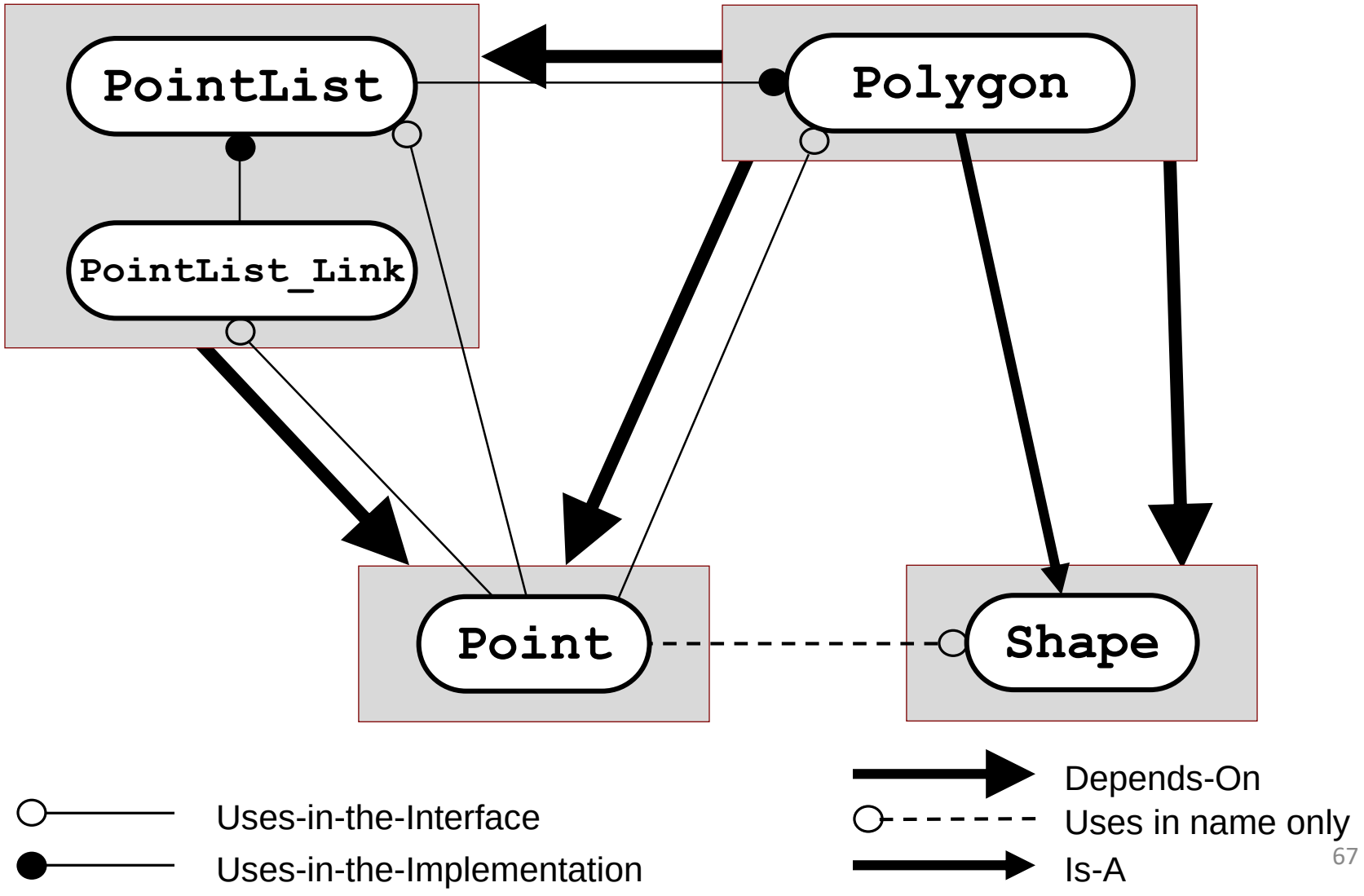
Implied Dependency



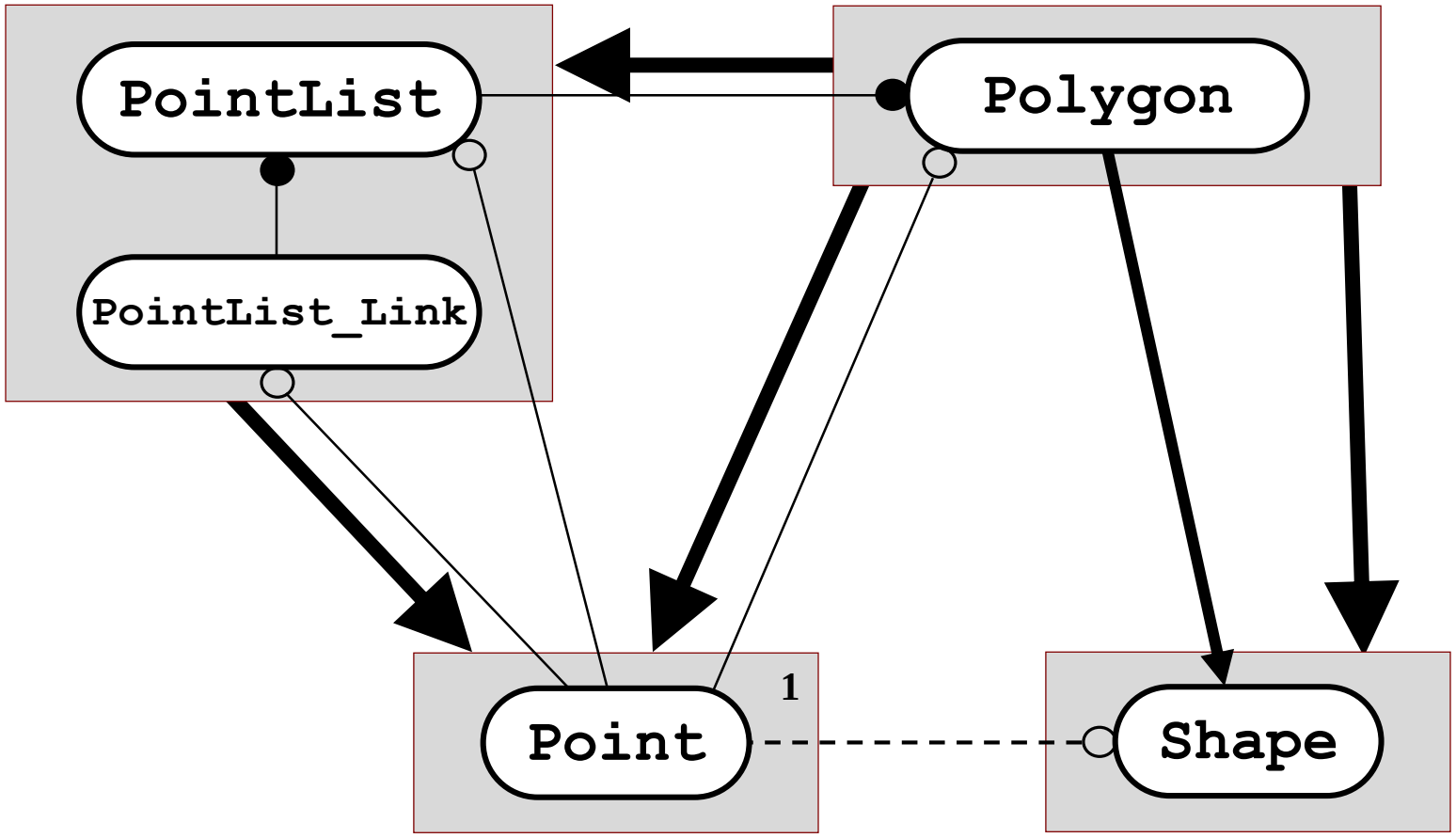
Implied Dependency



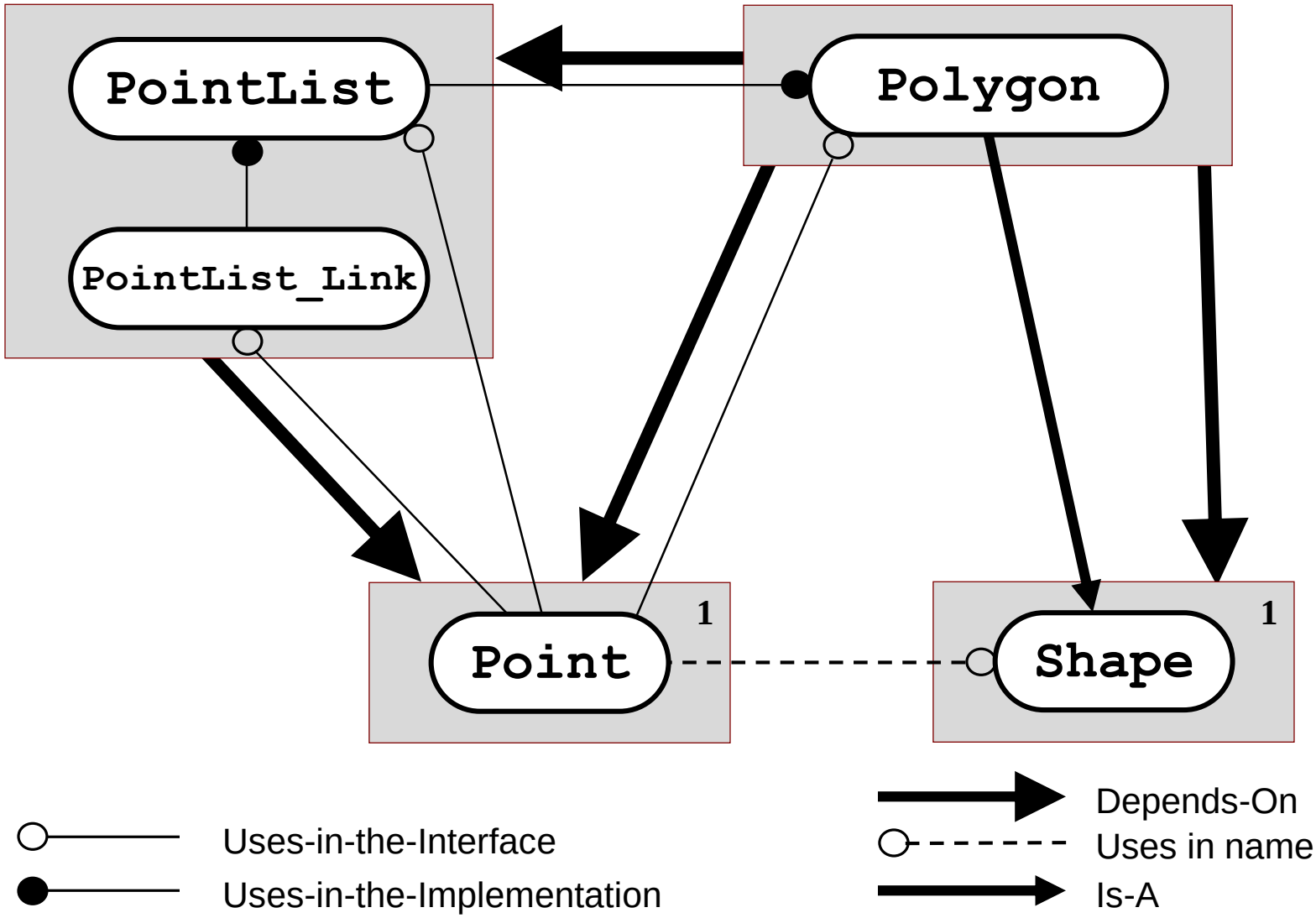
Level Numbers



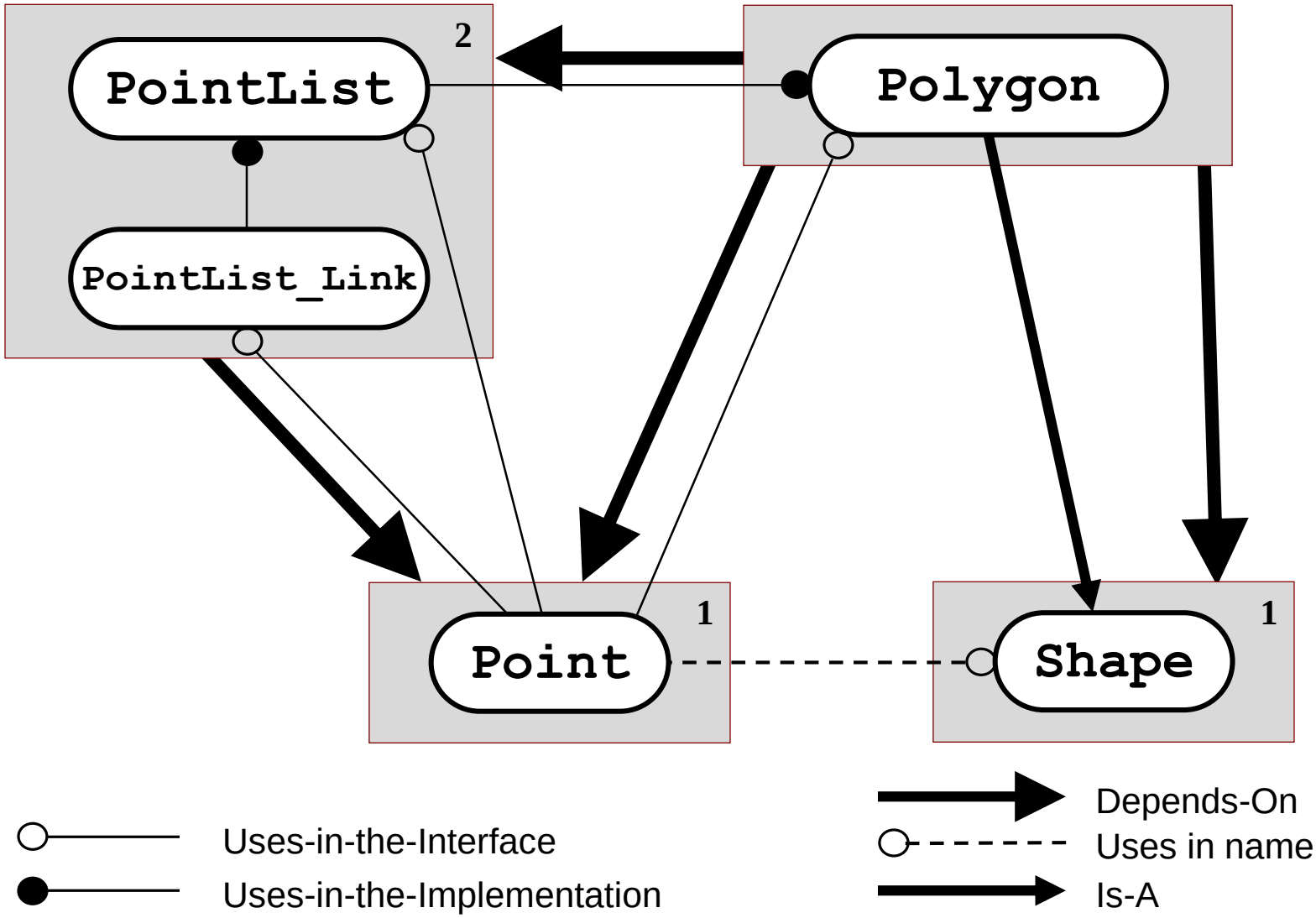
Level Numbers



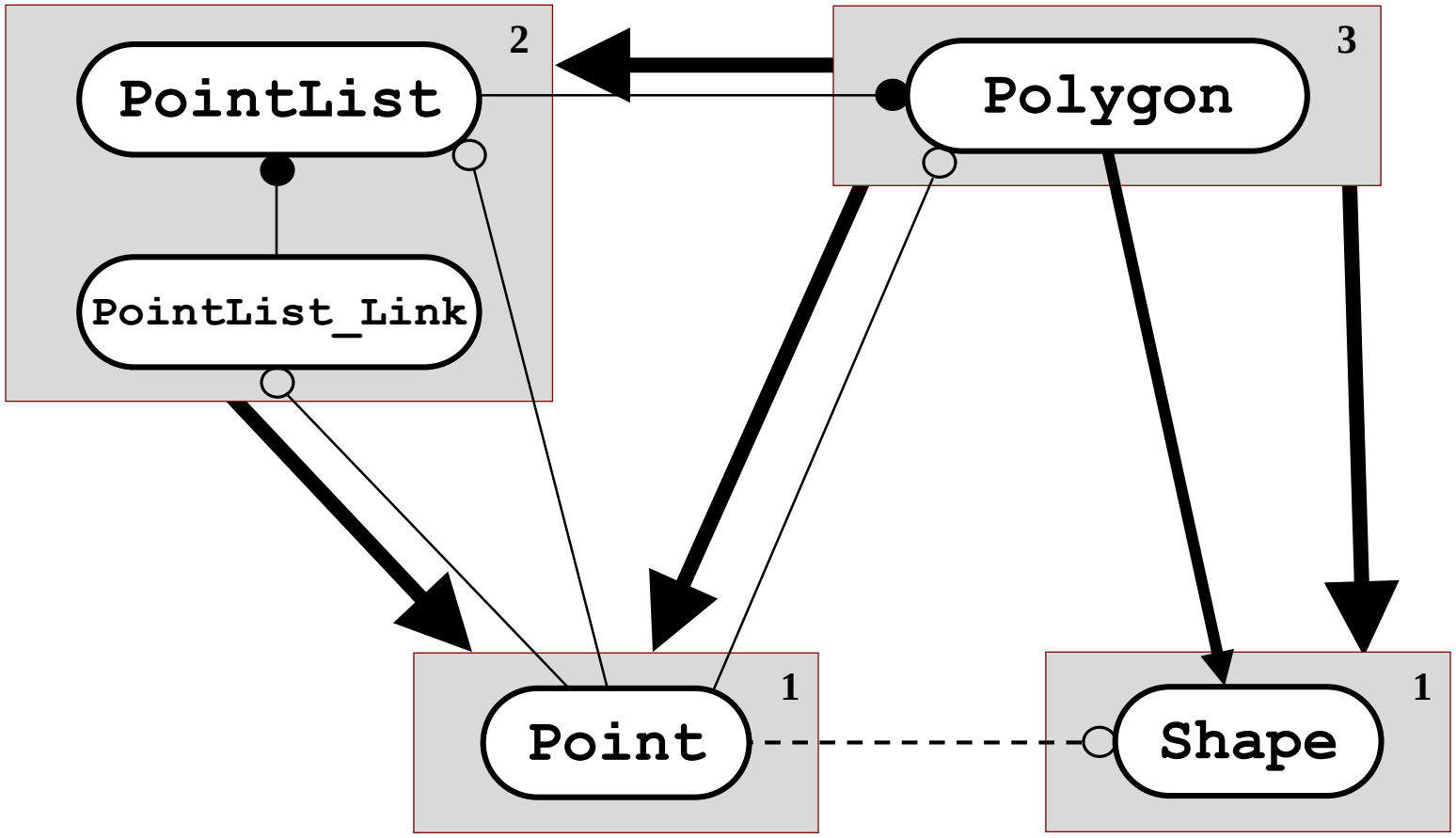
Level Numbers



Level Numbers



Level Numbers



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 Colocating “Public” Classes

Criteria for Colocating “Public” Classes

There are four:

Criteria for Colocating "Public" Classes

There are four:

1. Friendship.

Criteria for Colocating "Public" Classes

There are four:

1. Friendship.

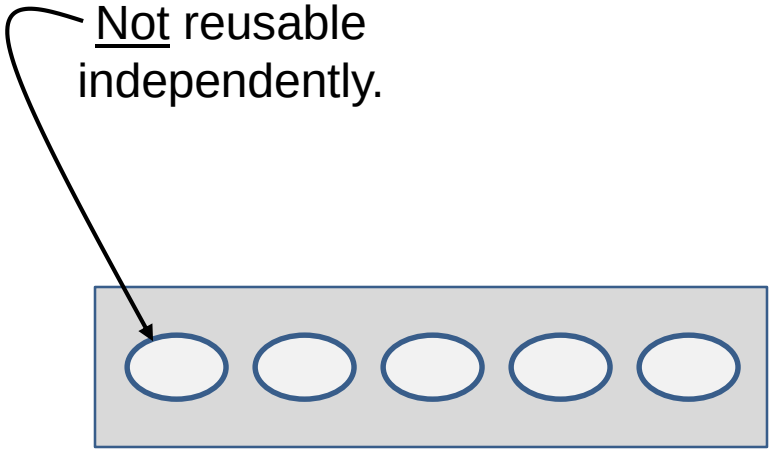
2. Cyclic Dependency.

Criteria for Colocating "Public" Classes

There are four:

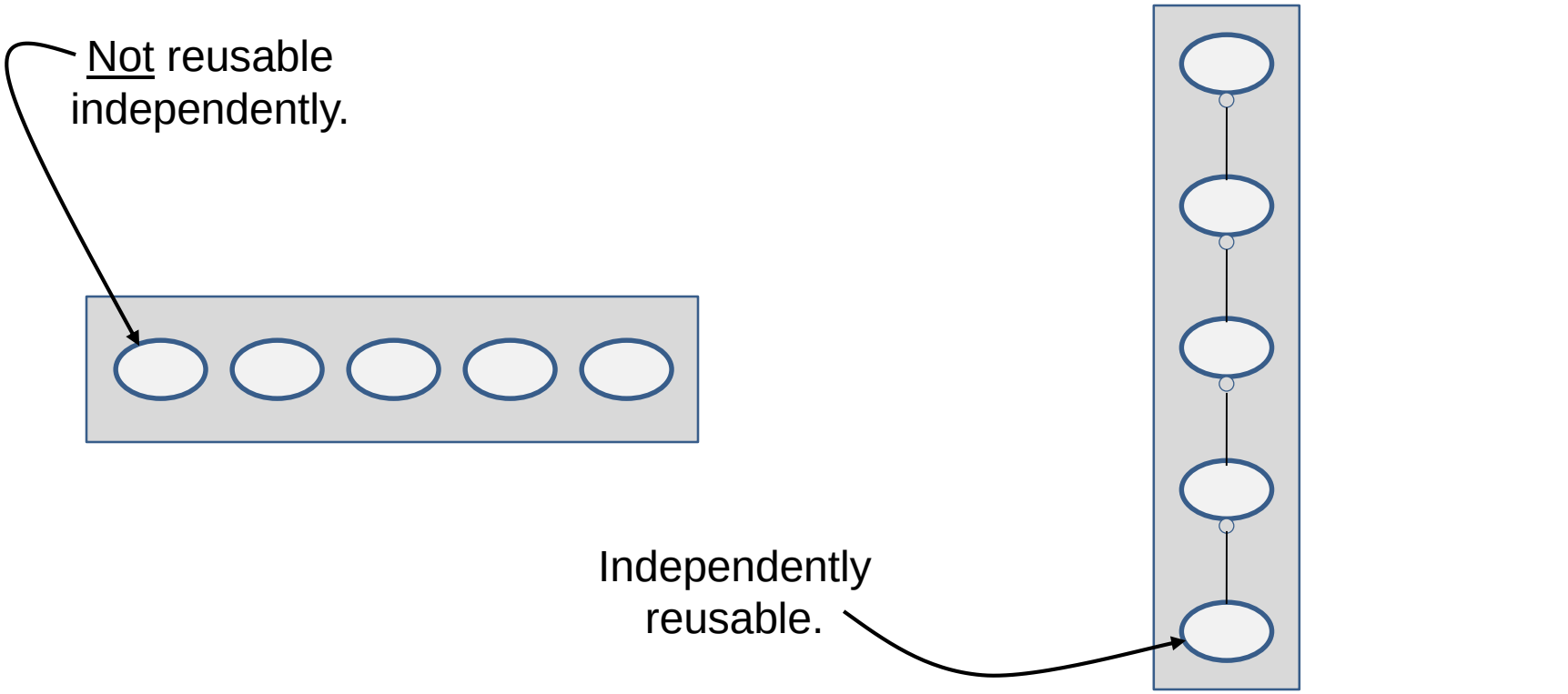
1. Friendship.
2. Cyclic Dependency.
3. Single Solution.

Criteria for Colocating "Public" Classes



Single Solution

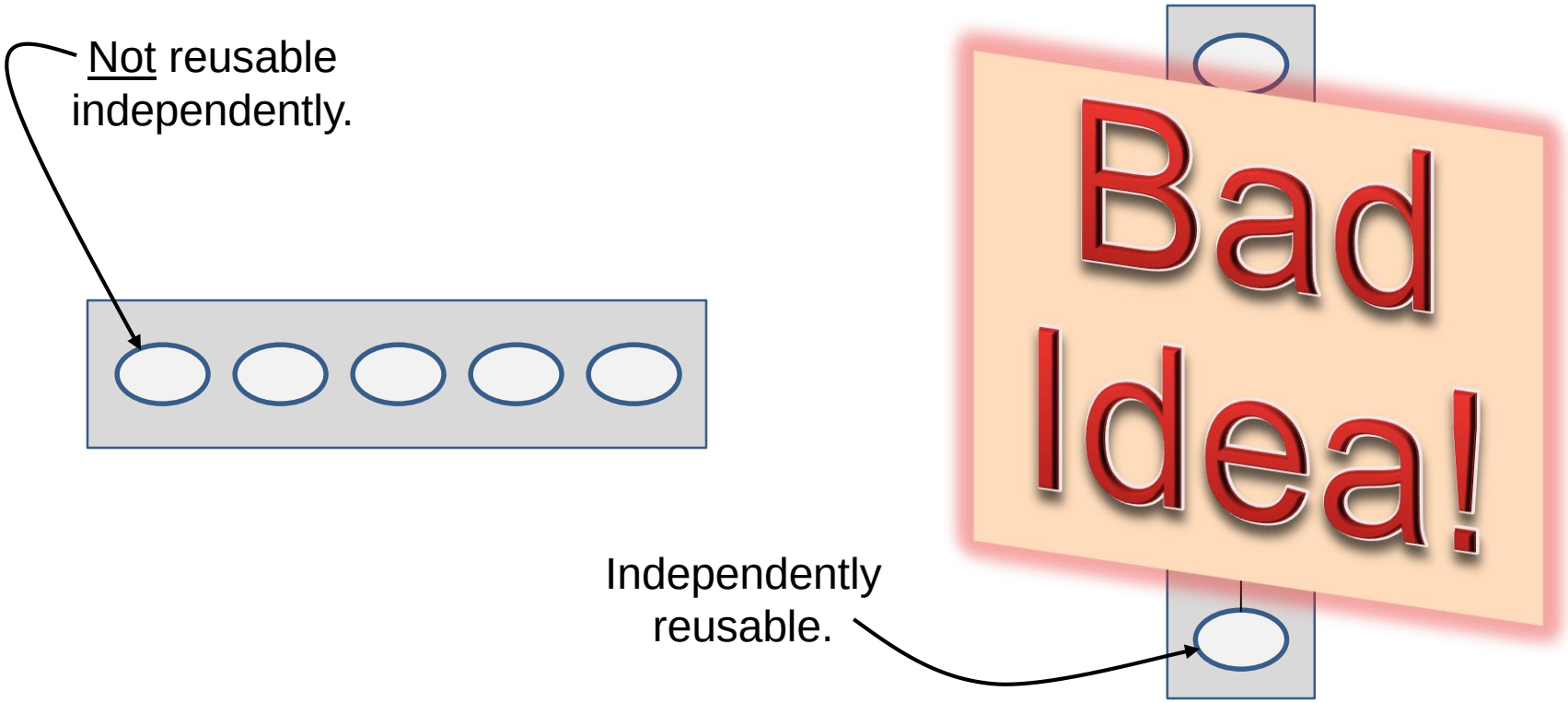
Criteria for Colocating "Public" Classes



Single Solution

Hierarchy of Solutions

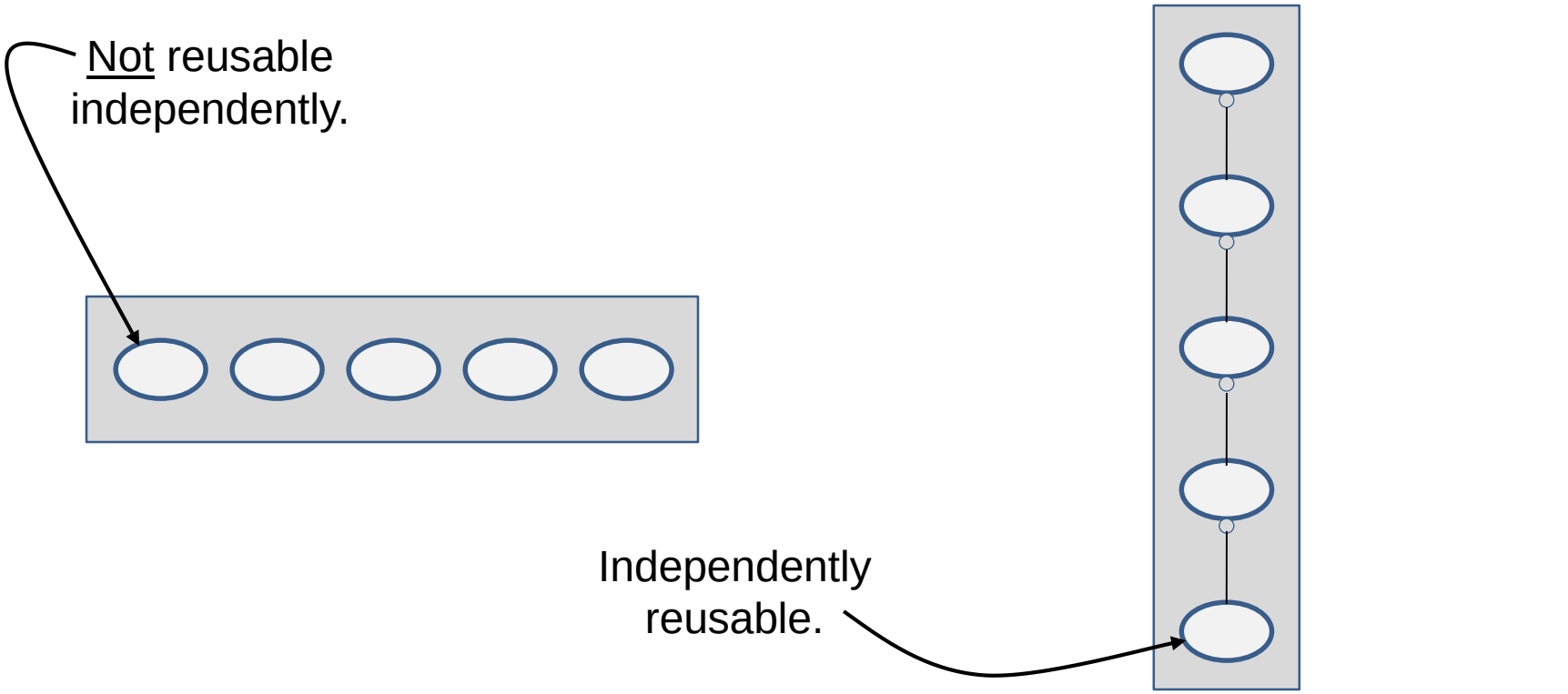
Criteria for Colocating "Public" Classes



Single Solution

Hierarchy of Solutions

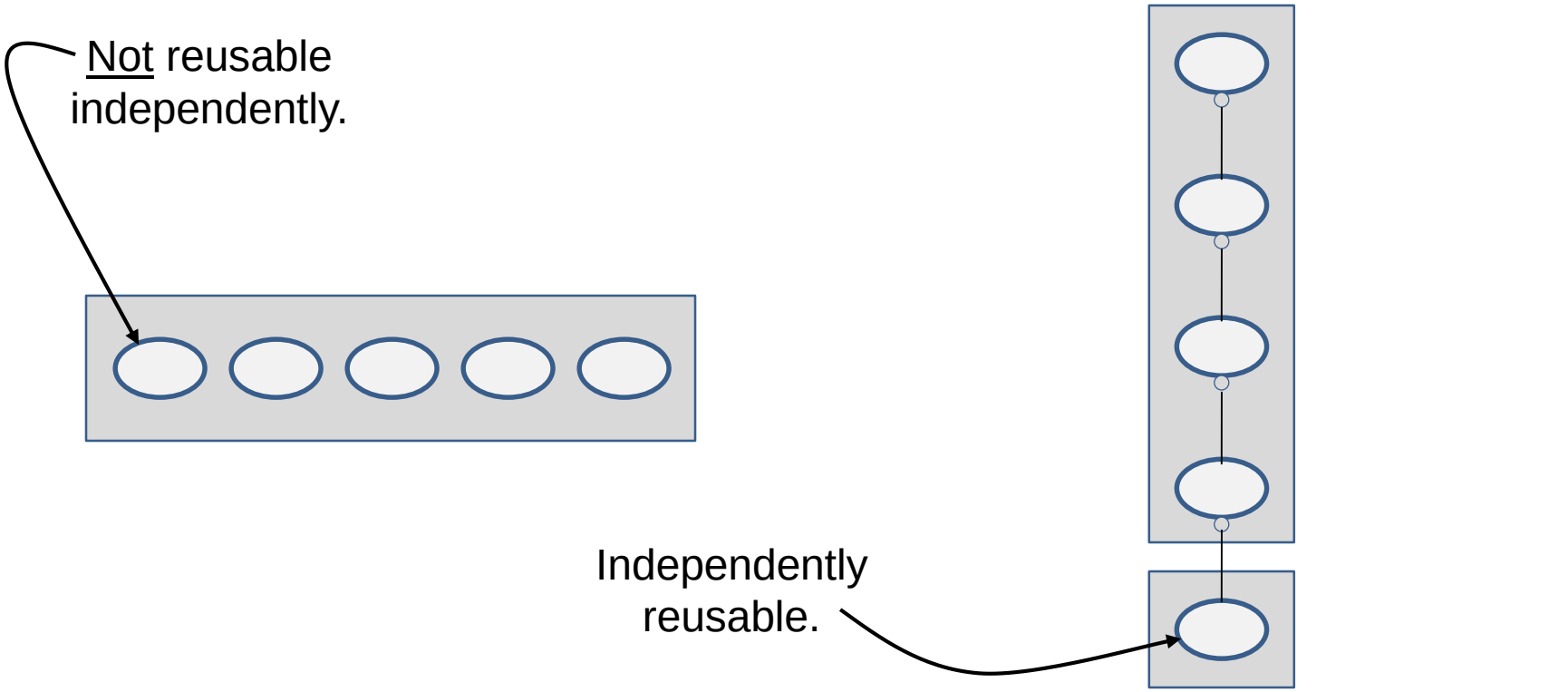
Criteria for Colocating "Public" Classes



Single Solution

Hierarchy of Solutions

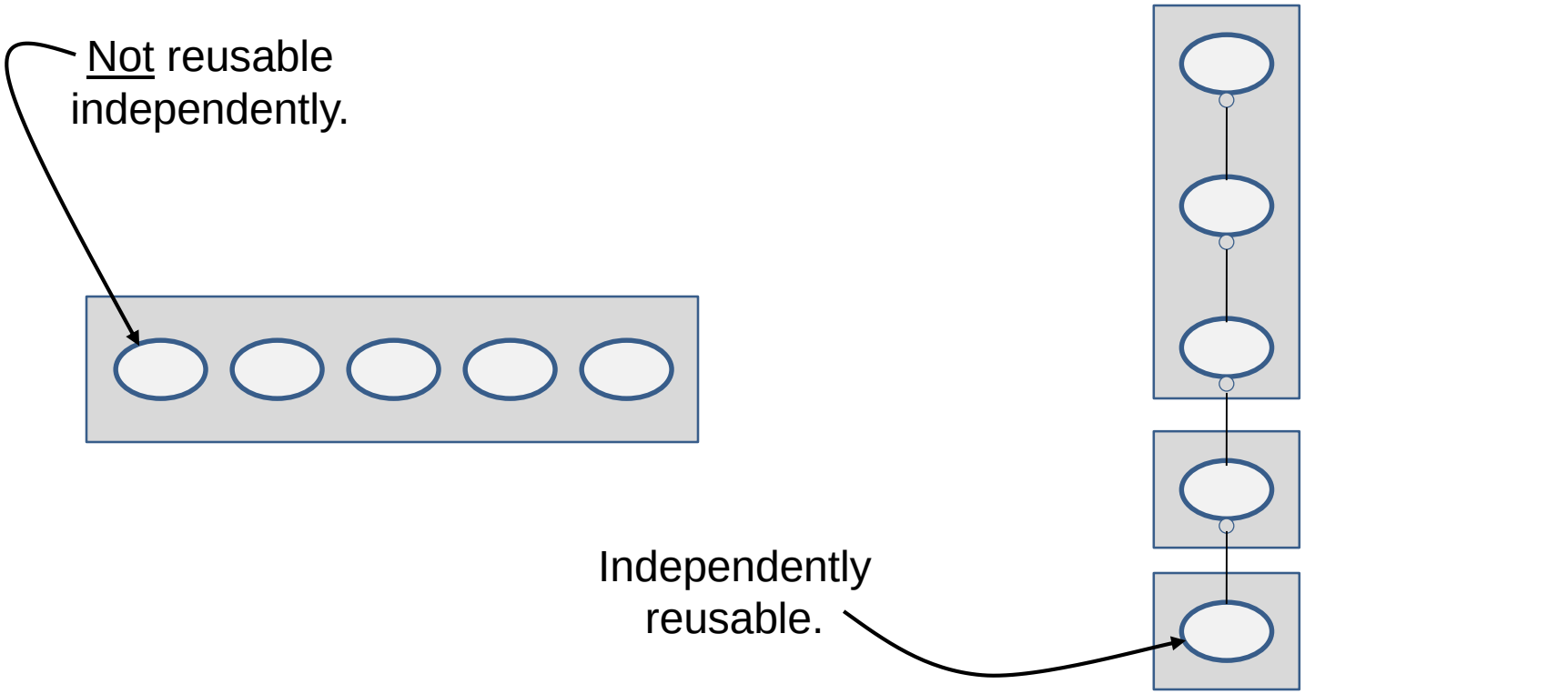
Criteria for Colocating "Public" Classes



Single Solution

Hierarchy of Solutions

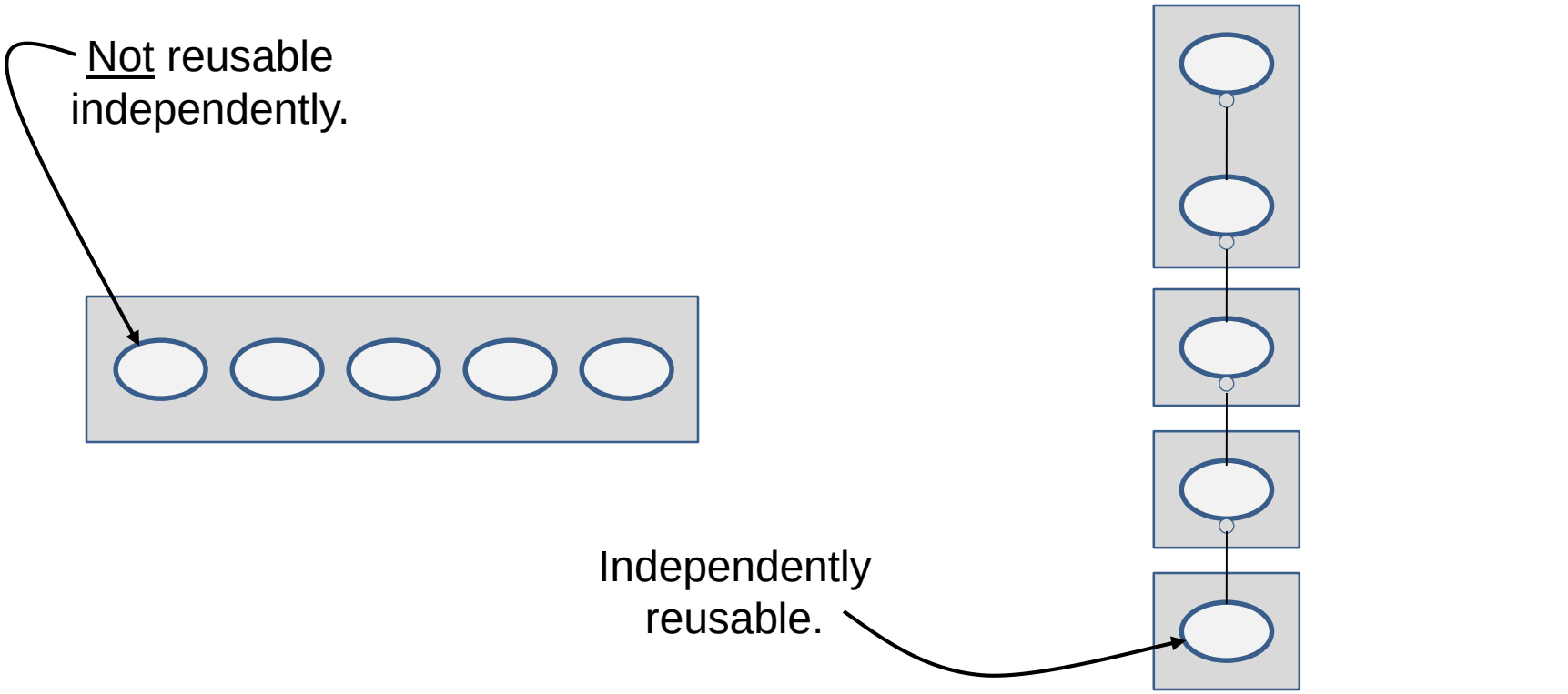
Criteria for Colocating "Public" Classes



Single Solution

Hierarchy of Solutions

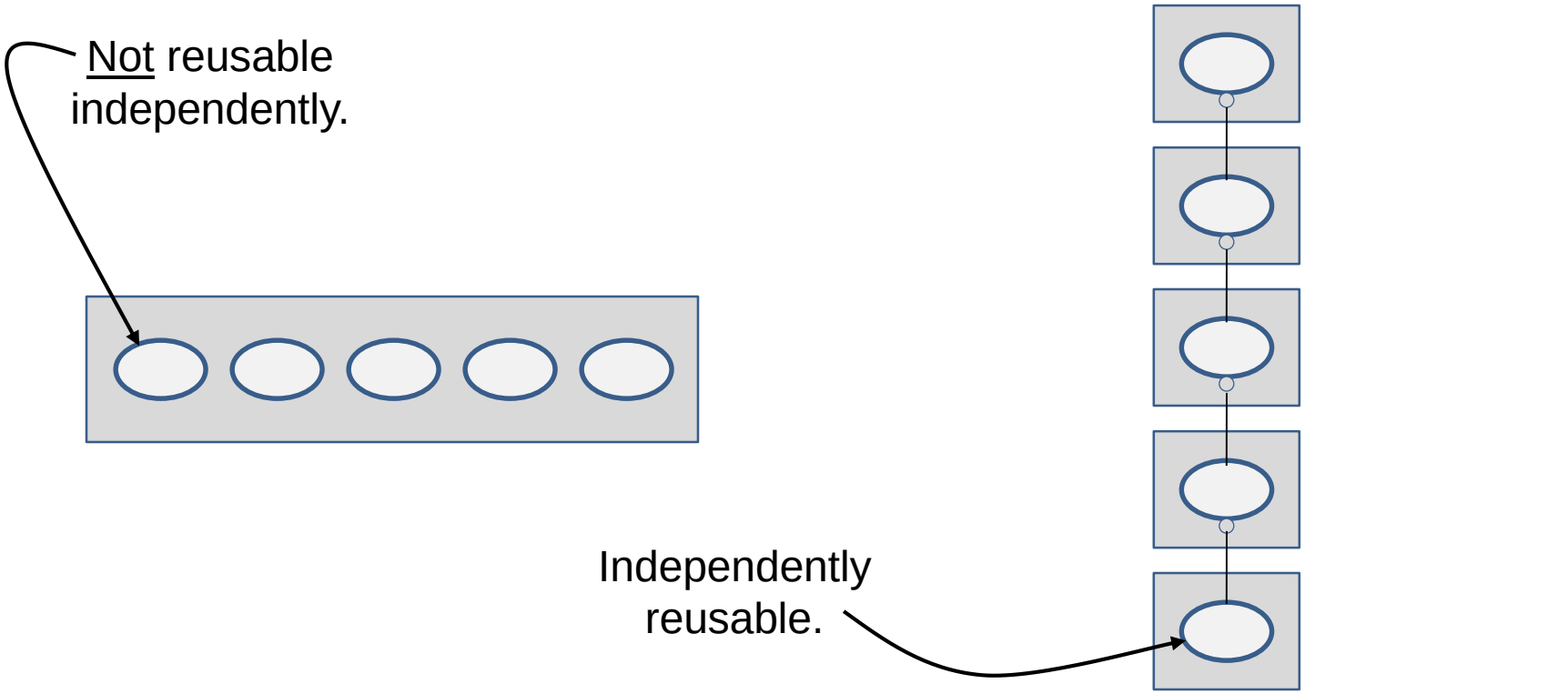
Criteria for Colocating "Public" Classes



Single Solution

Hierarchy of Solutions

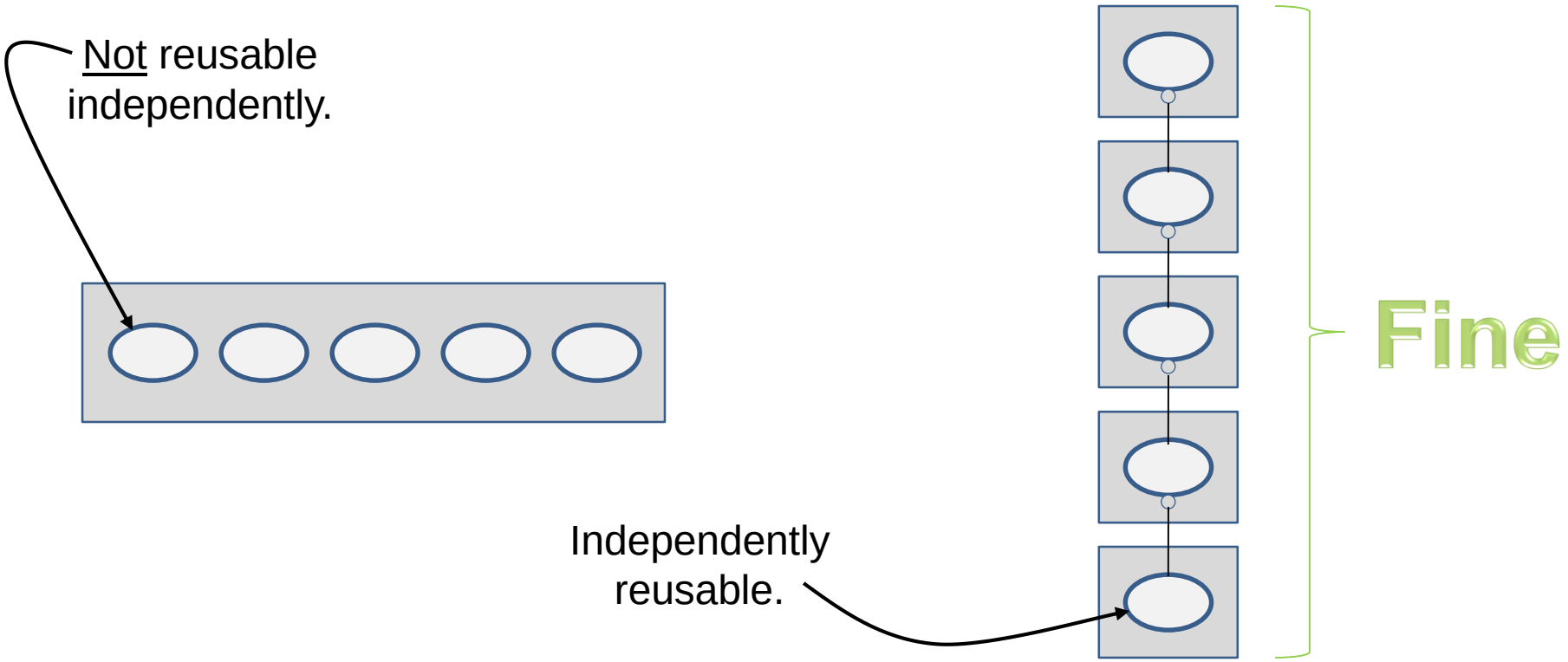
Criteria for Colocating "Public" Classes



Single Solution

Hierarchy of Solutions

Criteria for Colocating "Public" Classes

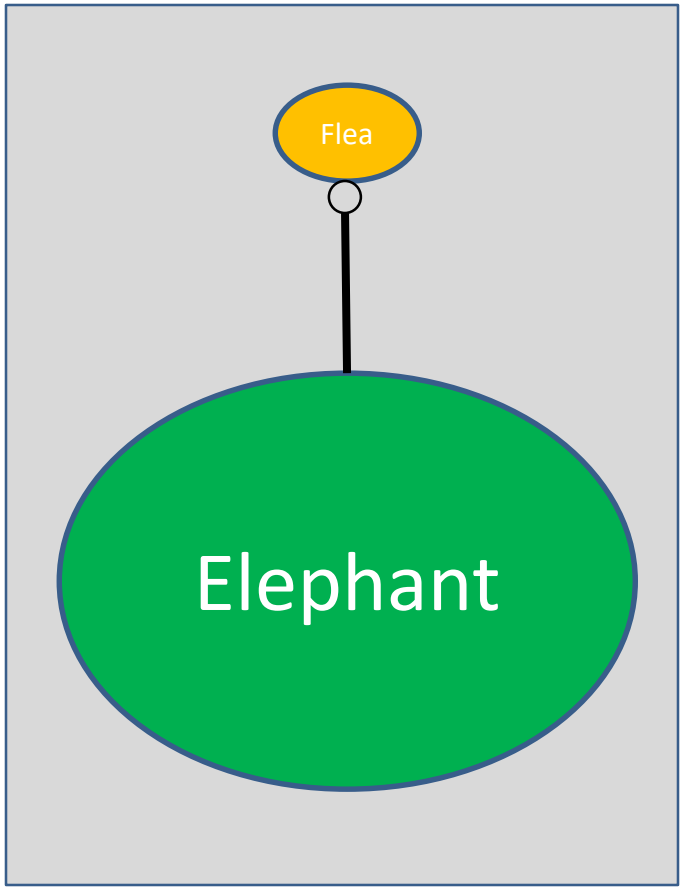


Criteria for Colocating "Public" Classes

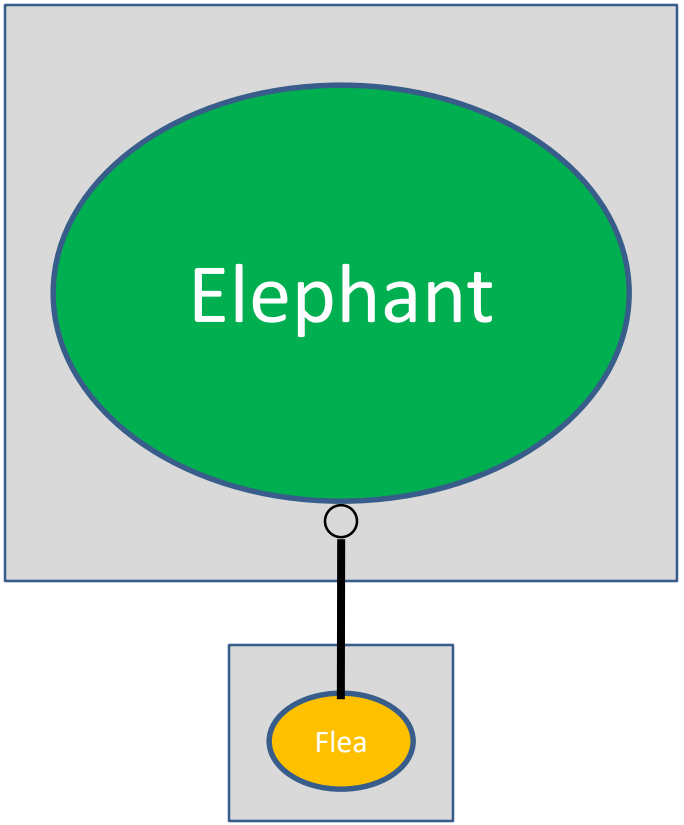
There are four:

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

Criteria for Colocating "Public" Classes

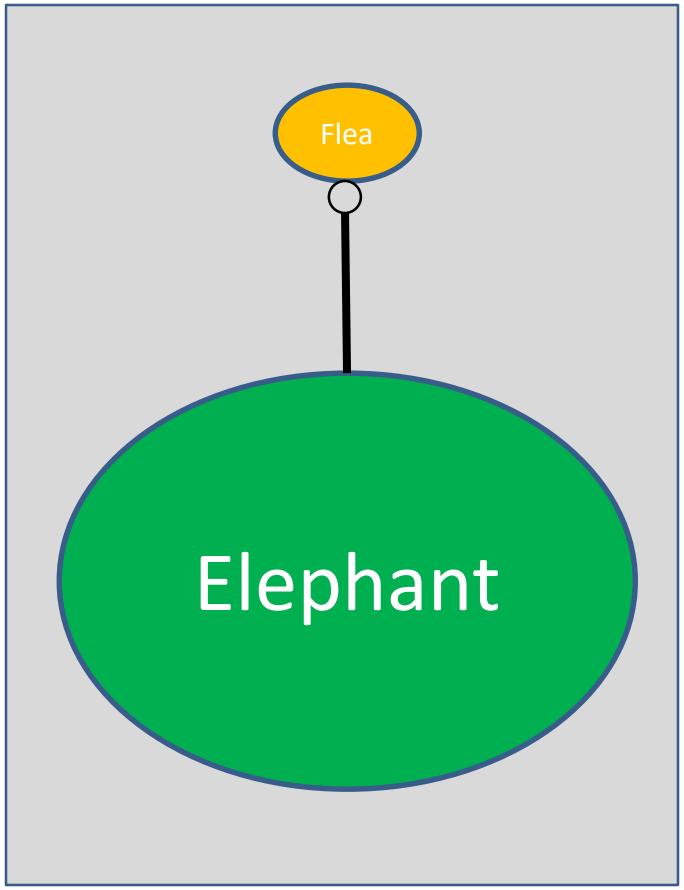


"Flea on an Elephant"

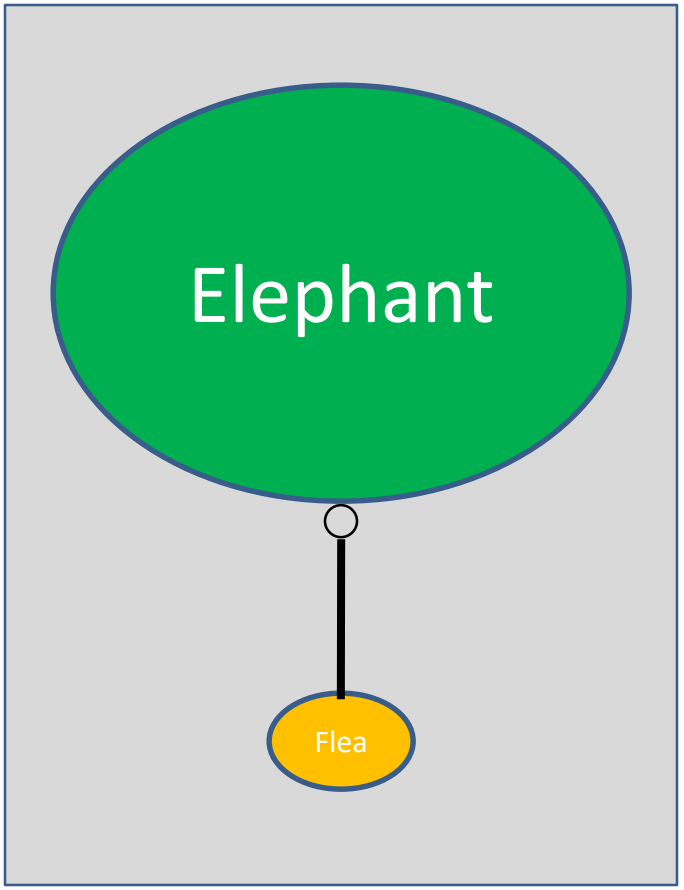


(Elephant on a Flea)

Criteria for Colocating "Public" Classes

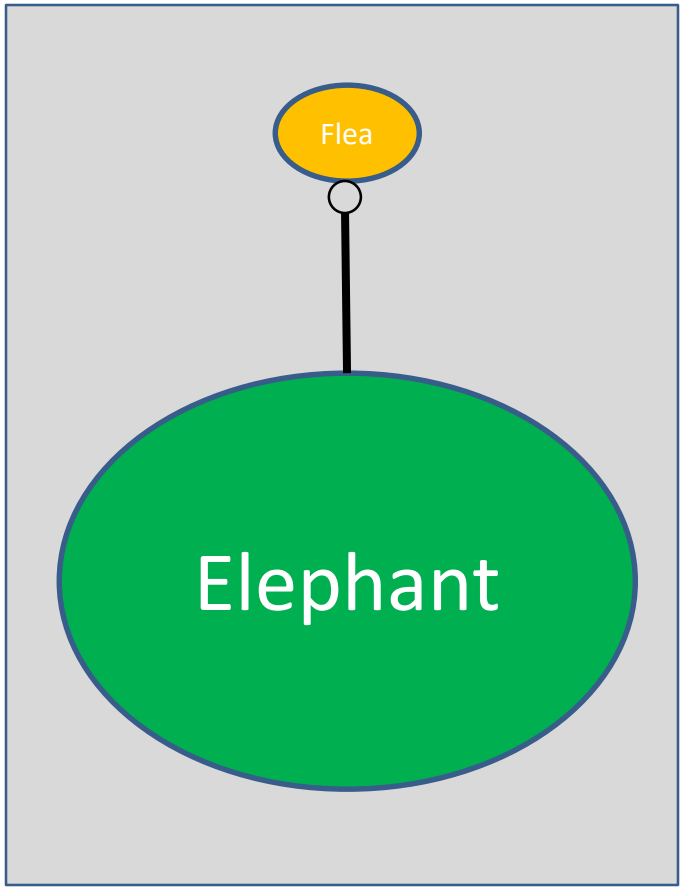


"Flea on an Elephant"



(Elephant on a Flea)

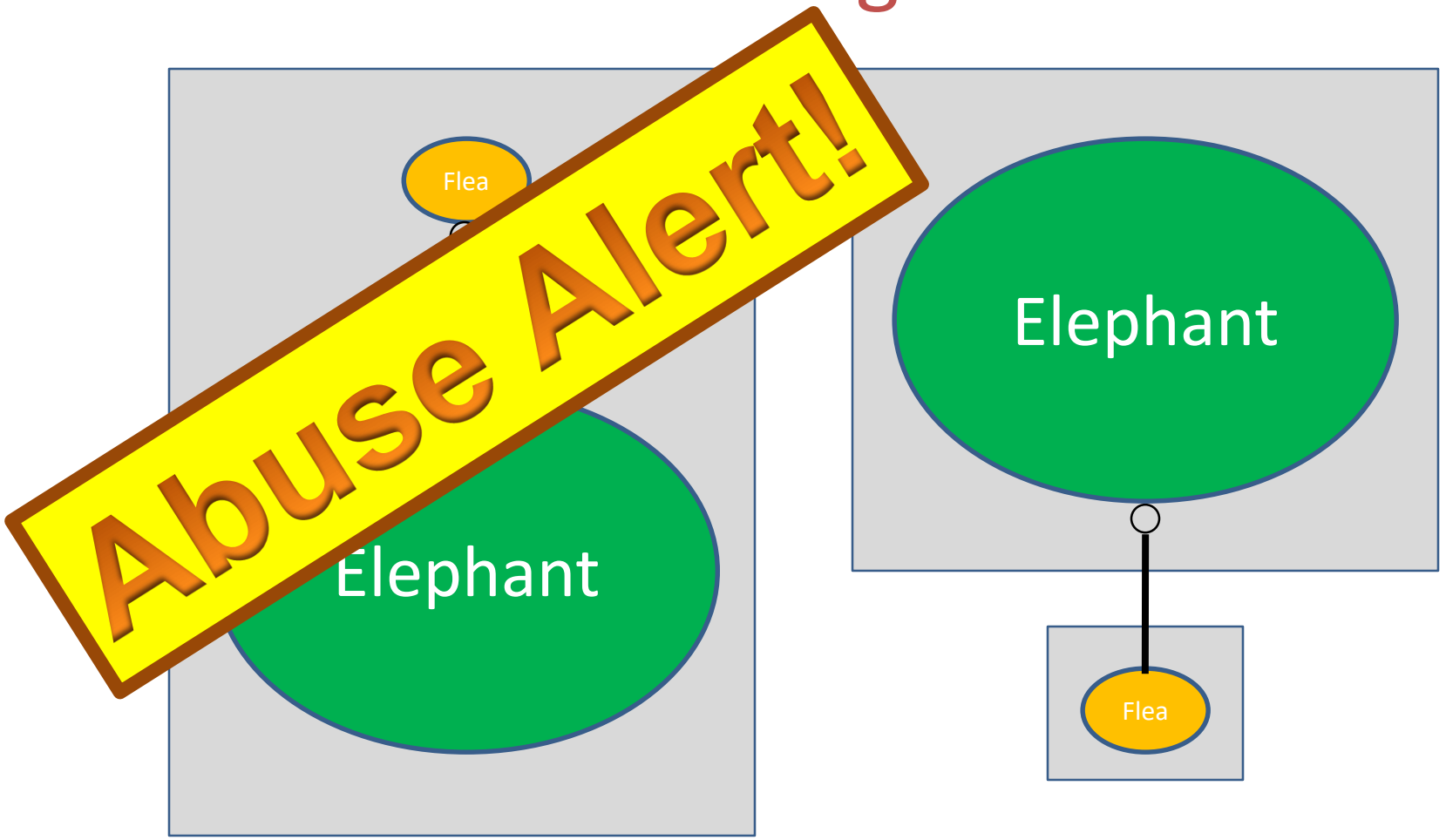
Criteria for Colocating "Public" Classes



"Flea on an Elephant"

(Elephant on a Flea)

Criteria for Colocating "Public" Classes



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

Insulation

Insulation

Logical ***encapsulation*** versus *physical* ***insulation***:

Insulation

Logical ***encapsulation*** versus *physical* ***insulation***:

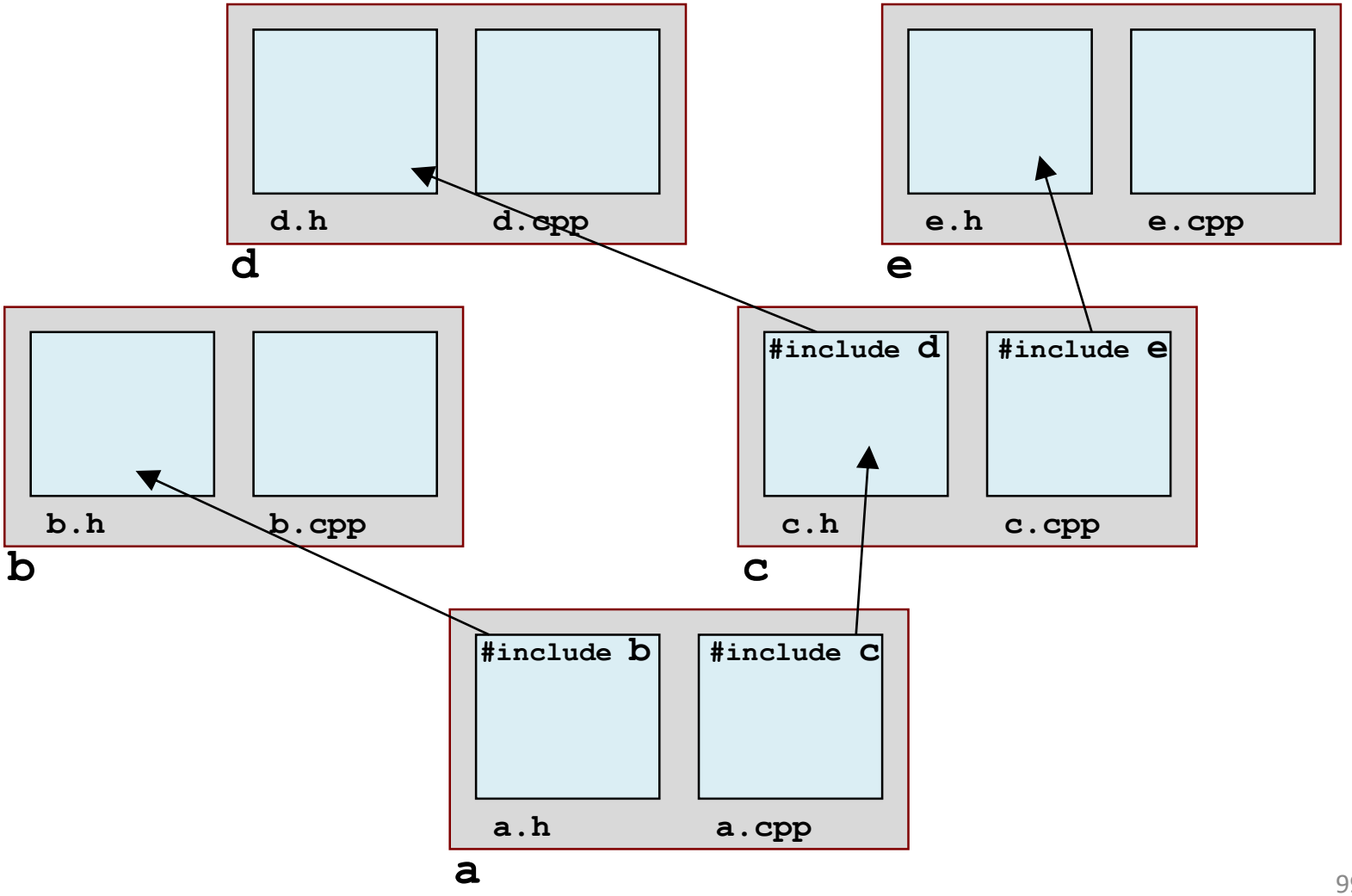
- An implementation detail of a component (type, data, or function) that can be altered, added, or removed *without* forcing clients to *rework their code* is said to be ***encapsulated***.

Insulation

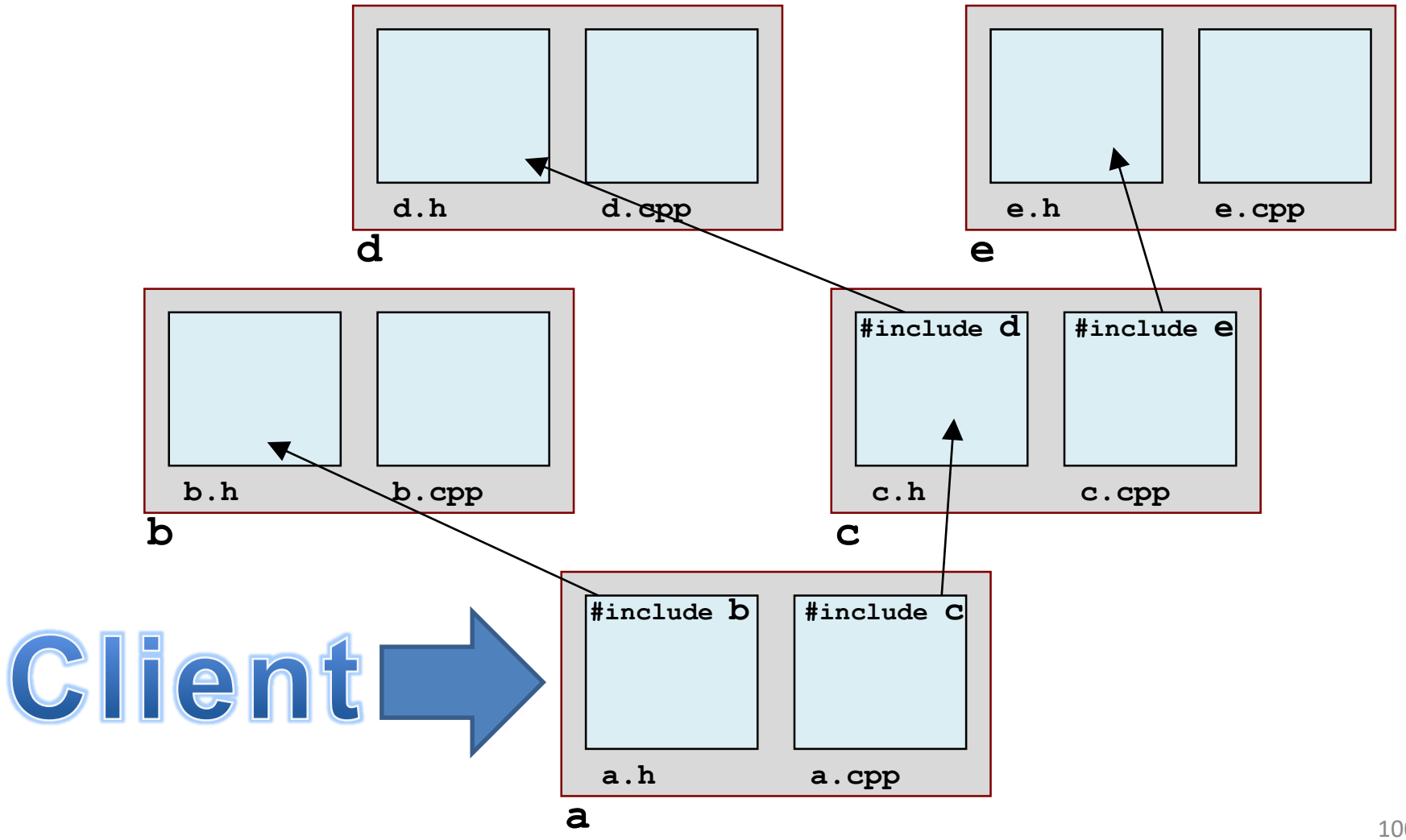
Logical ***encapsulation*** versus *physical* ***insulation***:

- An implementation detail of a component (type, data, or function) that can be altered, added, or removed *without* forcing clients to *rework their code* is said to be ***encapsulated***.
- An implementation detail of a component (type, data, or function) that can be altered, added, or removed *without* forcing clients to *recompile* is said to be ***insulated***.

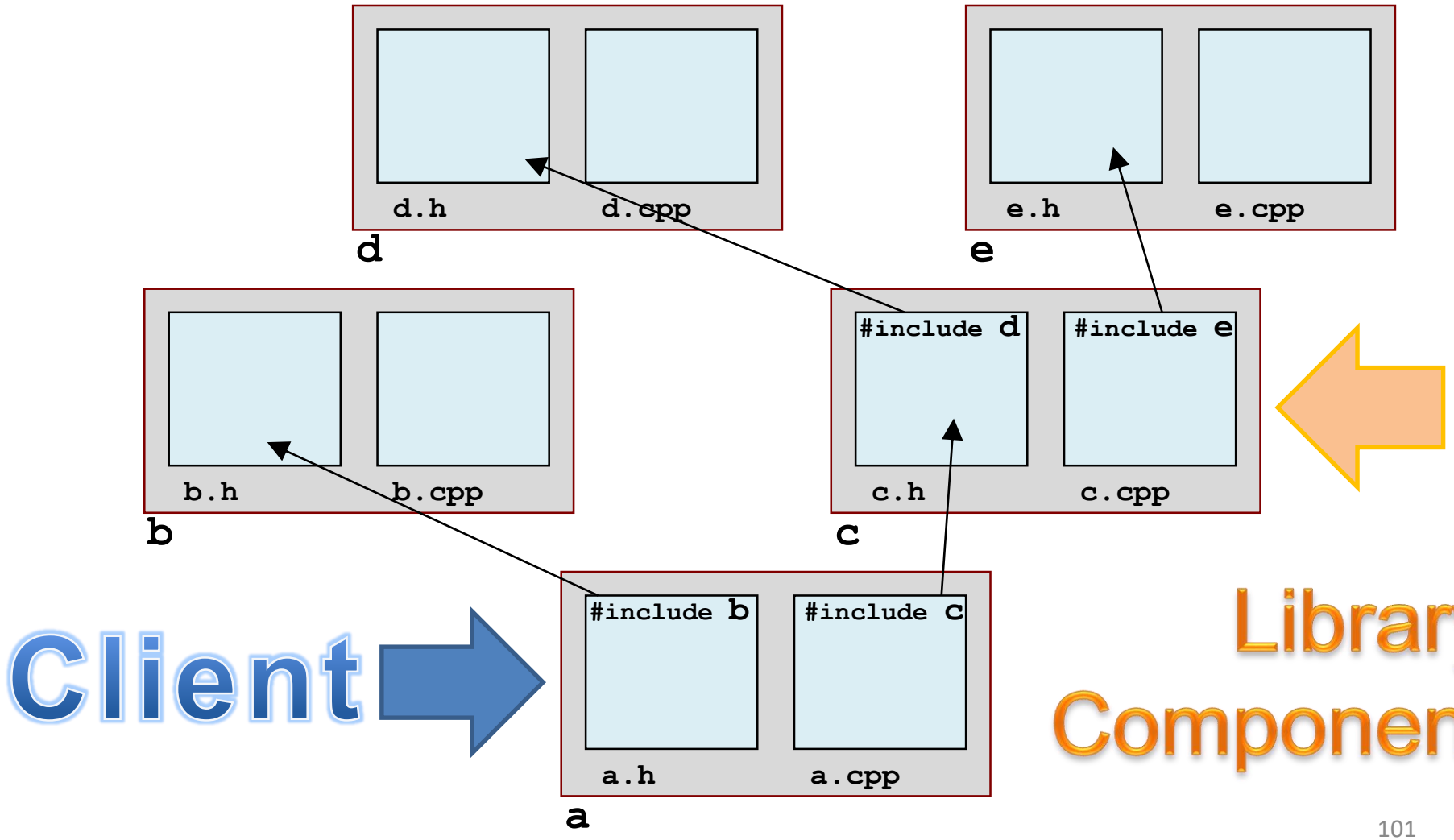
Insulation


d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
#include b
#include c
a.h
a.cpp
a

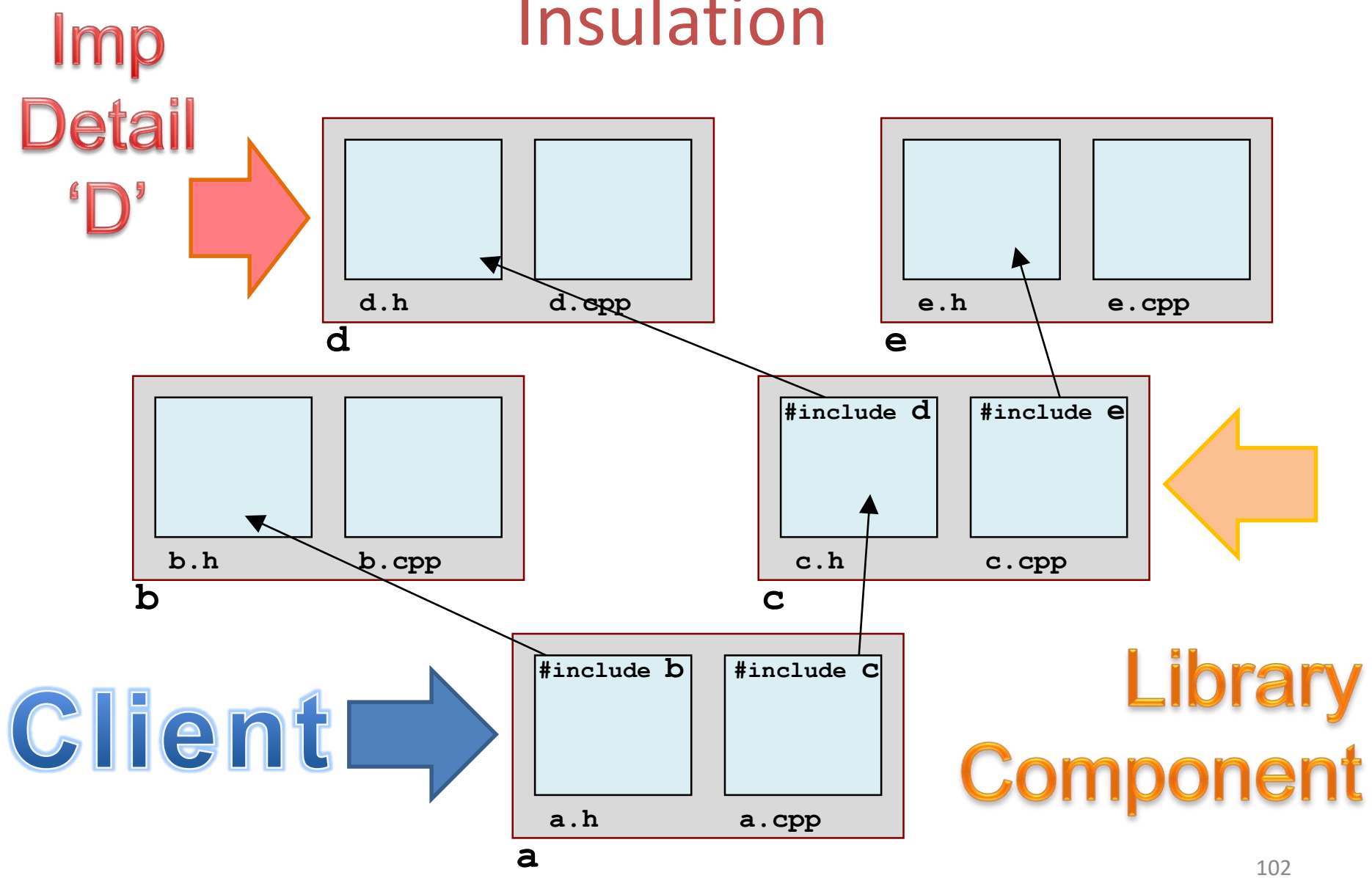
Insulation


d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a

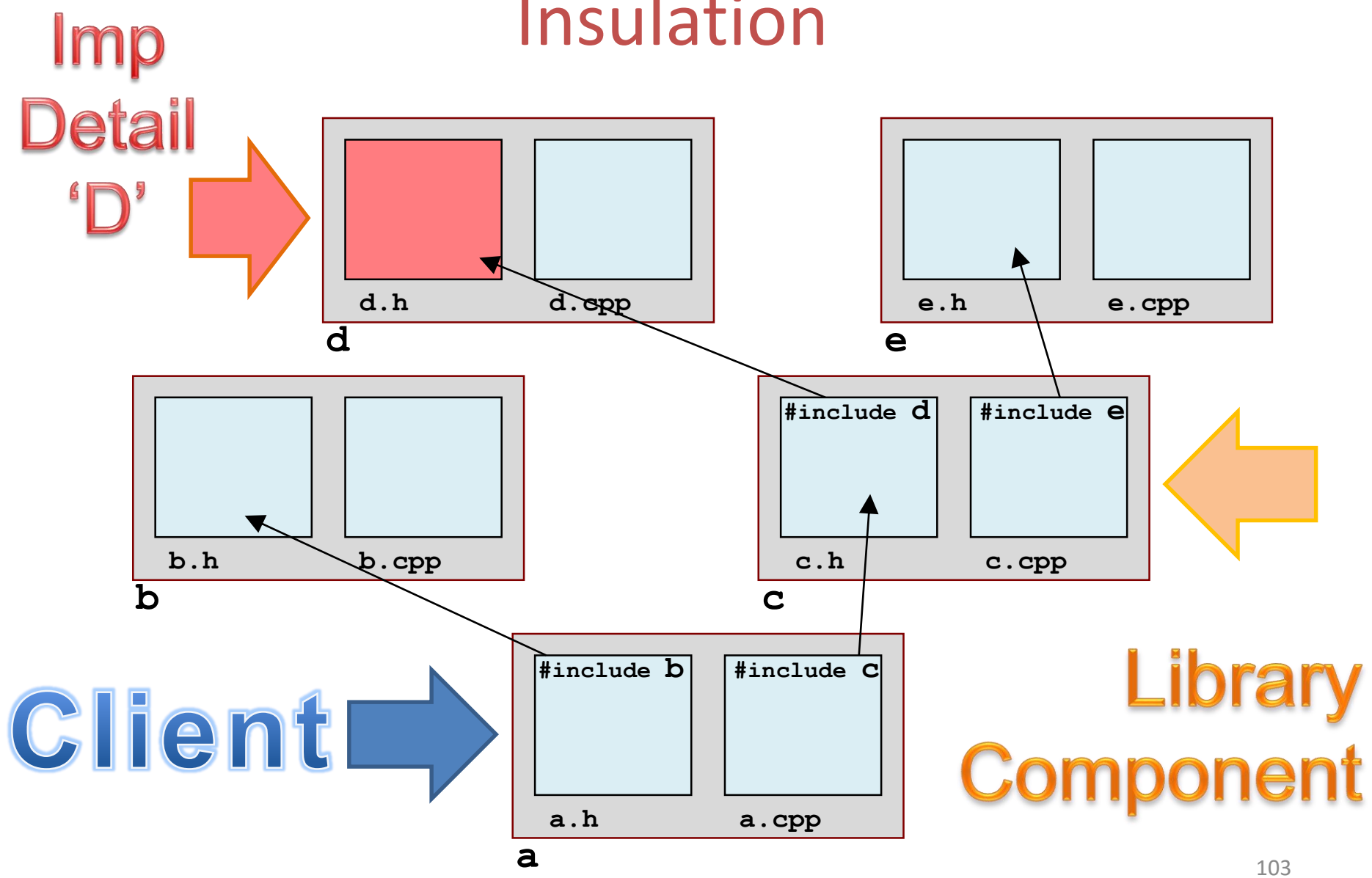
Insulation


d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
#include b
#include c
a.h
a.cpp
a
Client
Library
Component

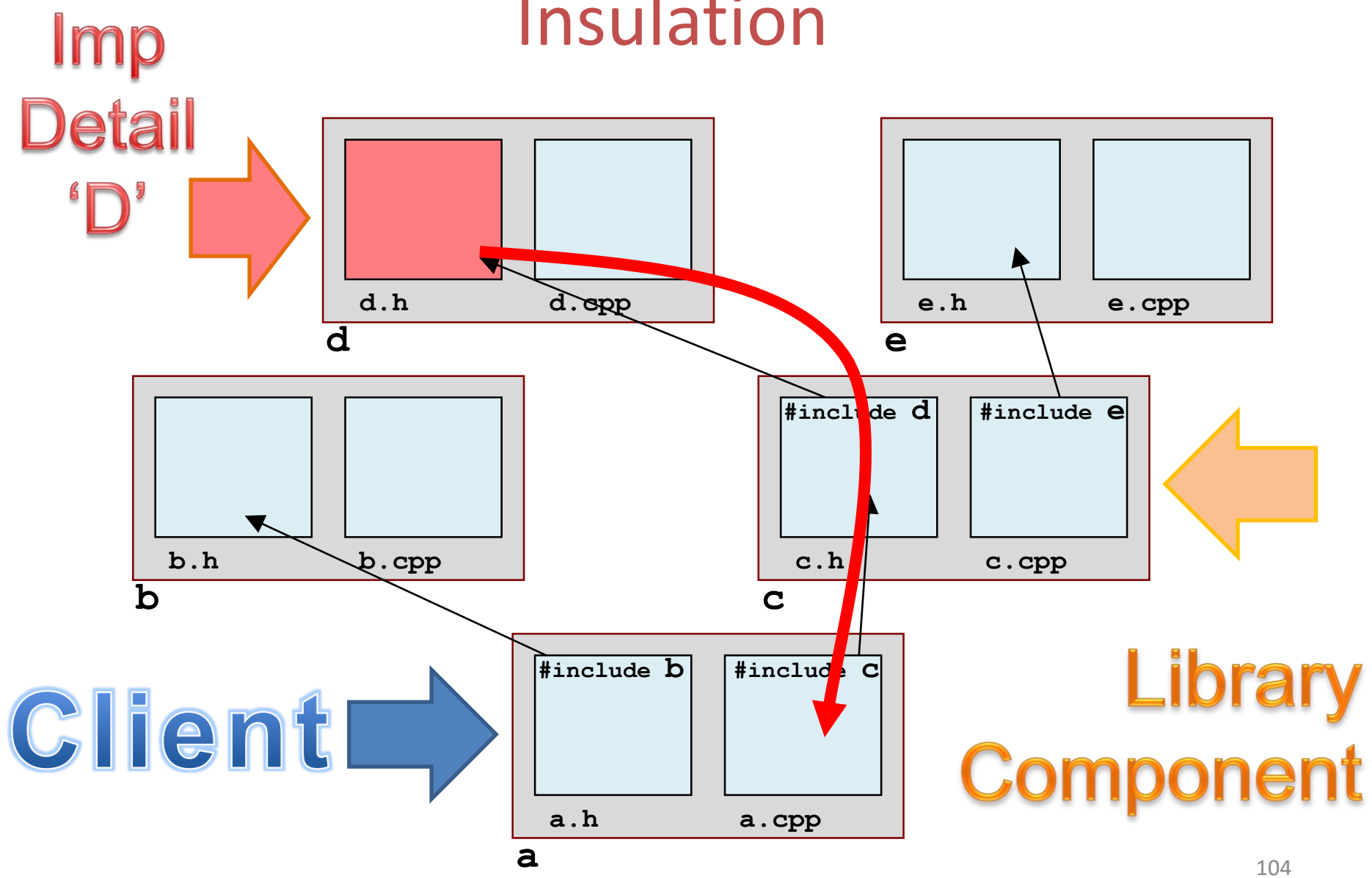
Insulation


Imp
Detail
'D'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
#include b
#include c
a.h
a.cpp
a
Client
Library
Component

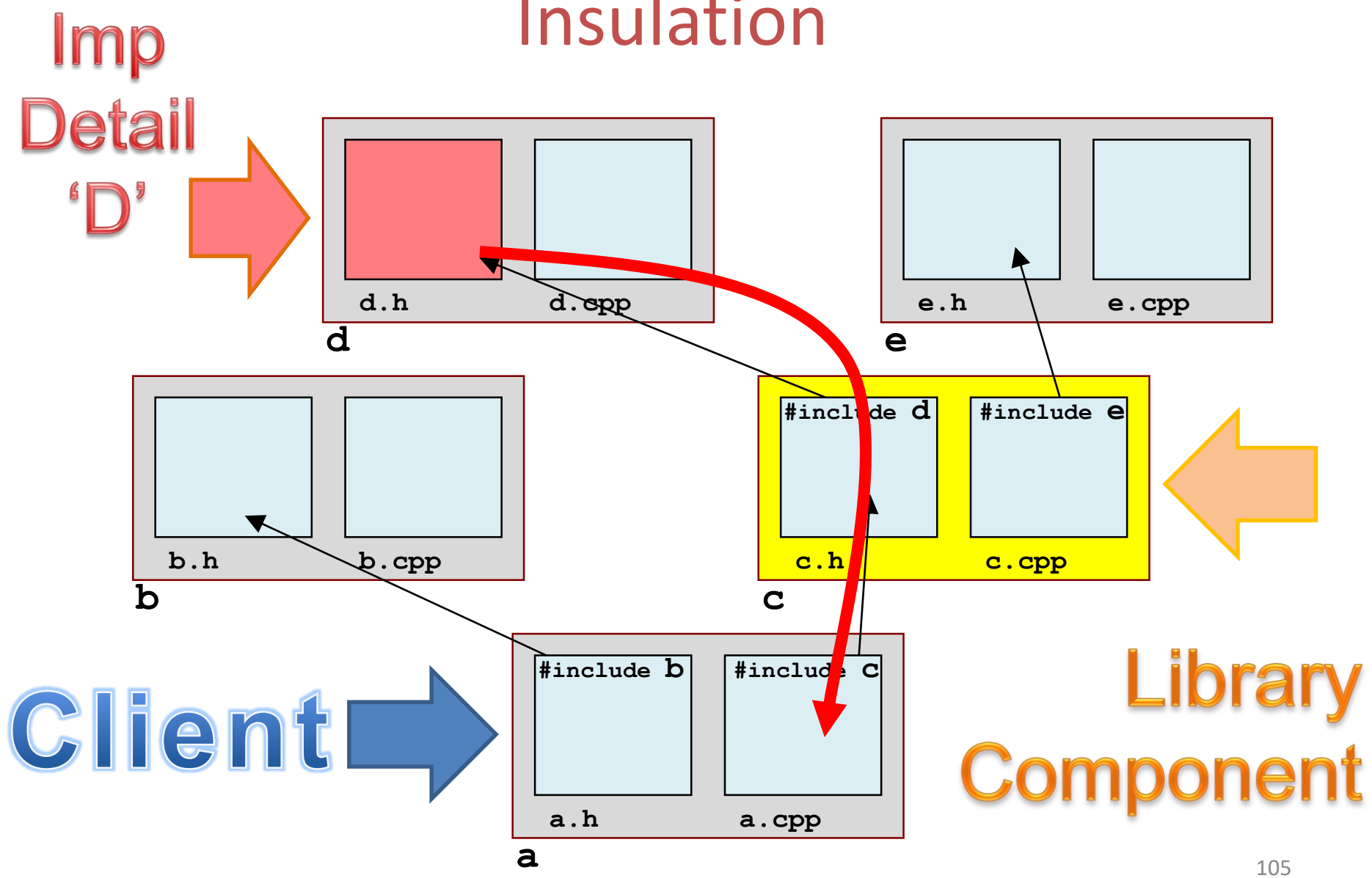
Insulation


Imp
Detail
'D'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

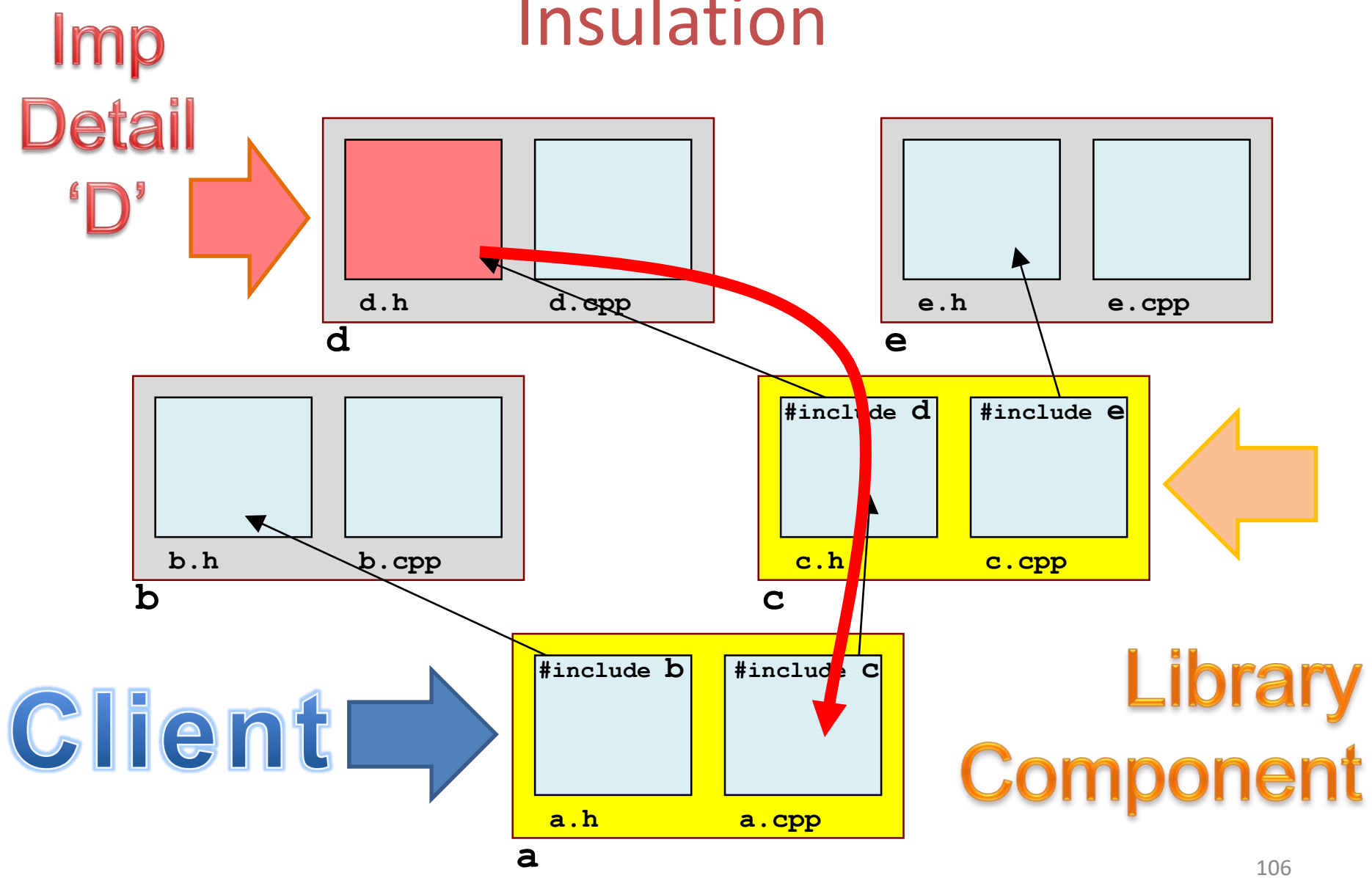
Insulation


Imp
Detail
'D'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
#include b
#include c
a.h
a.cpp
a
Client
Library
Component

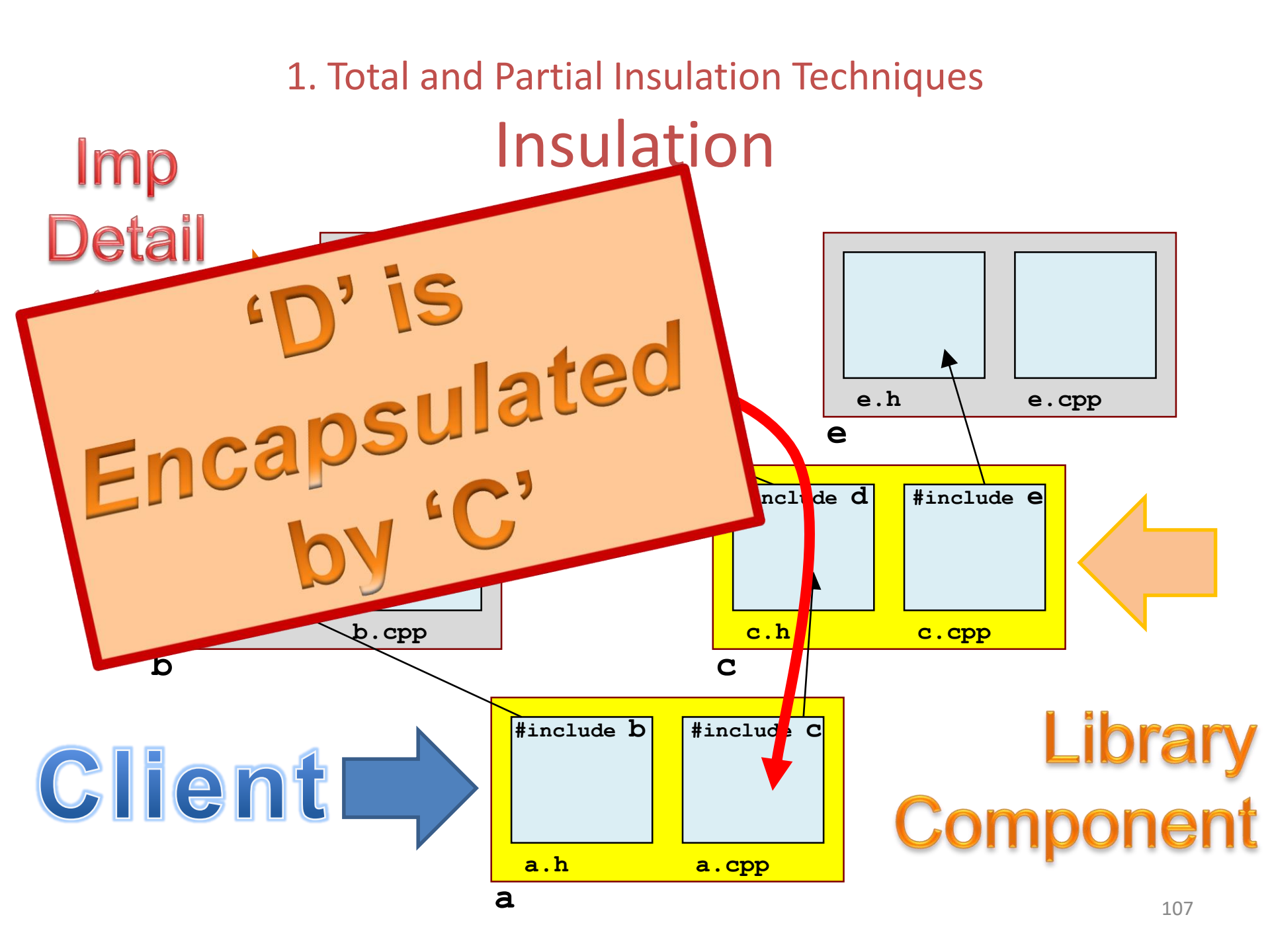
Insulation


Imp
Detail
'D'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

Insulation


Imp
Detail
'D'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
#include b
#include c
a.h
a.cpp
a
Client
Library
Component

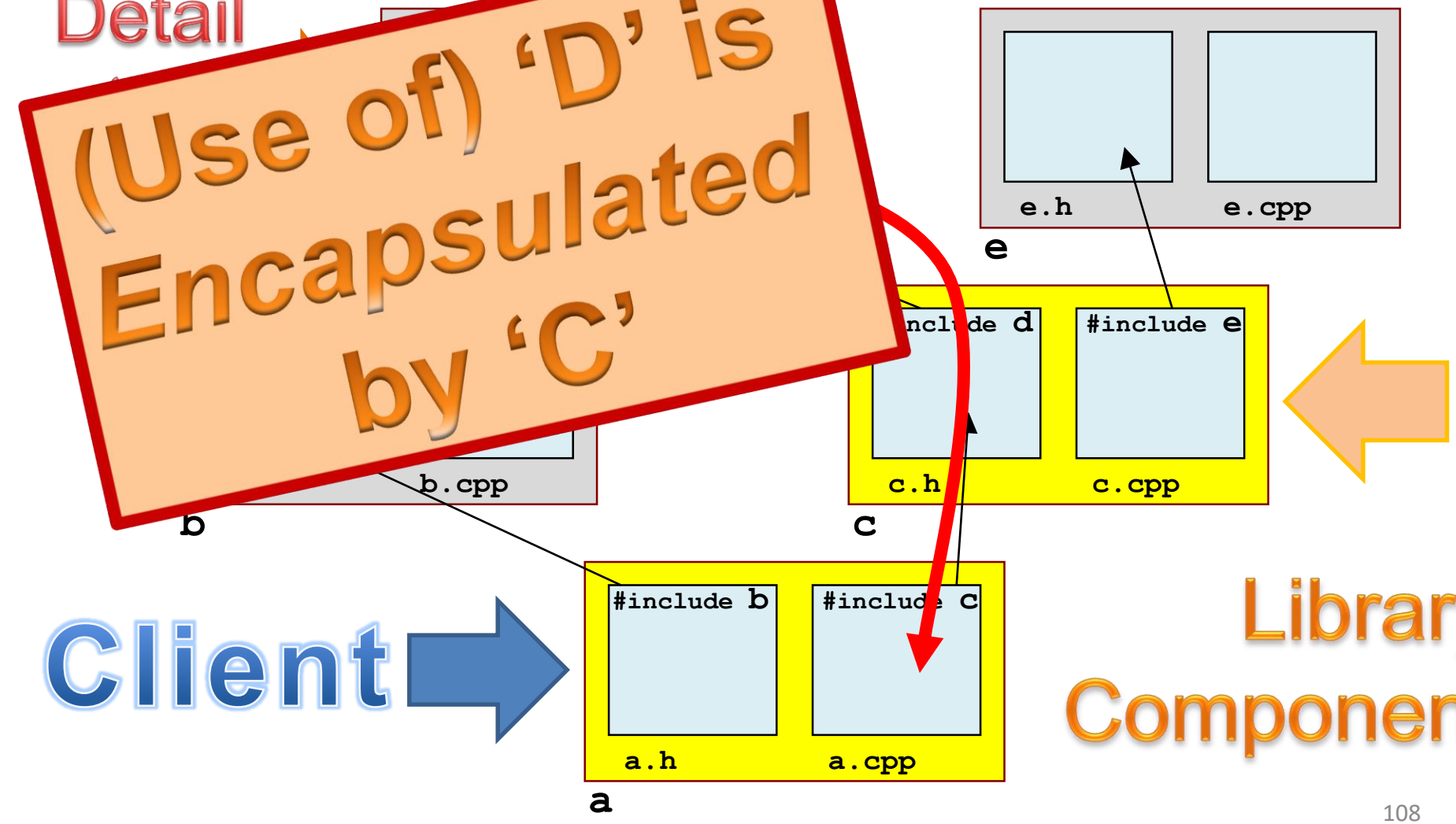
Insulation


Imp
Detail
'D' is
Encapsulated
by 'C'
e.h
e.cpp
e
#include d
#include e
c.h
c.cpp
c
b.cpp
b
#include b
#include c
a.h
a.cpp
a
Client
Library
Component

Insulation

Imp Detail

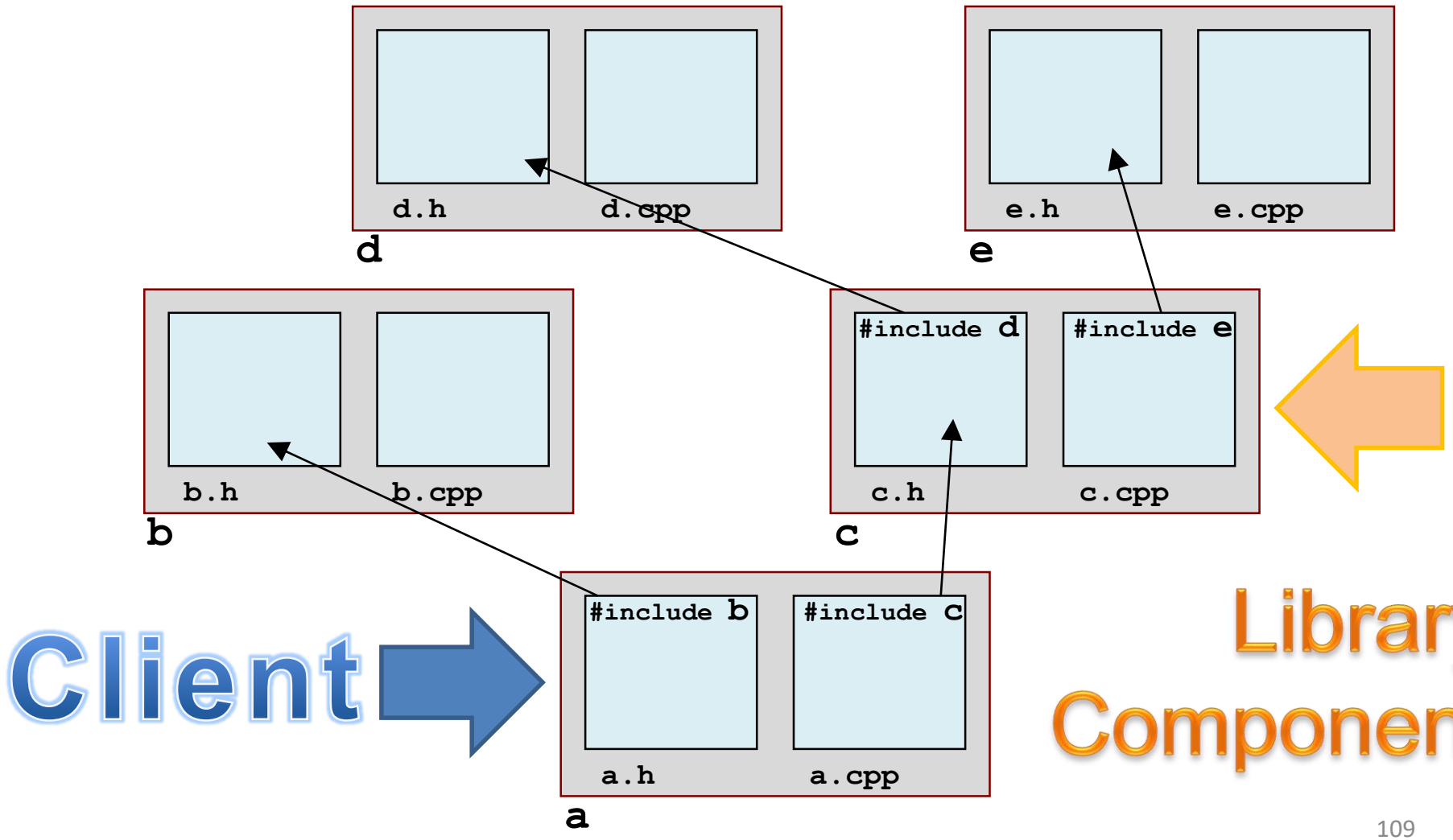
(Use of) 'D' is Encapsulated by 'C'



Client

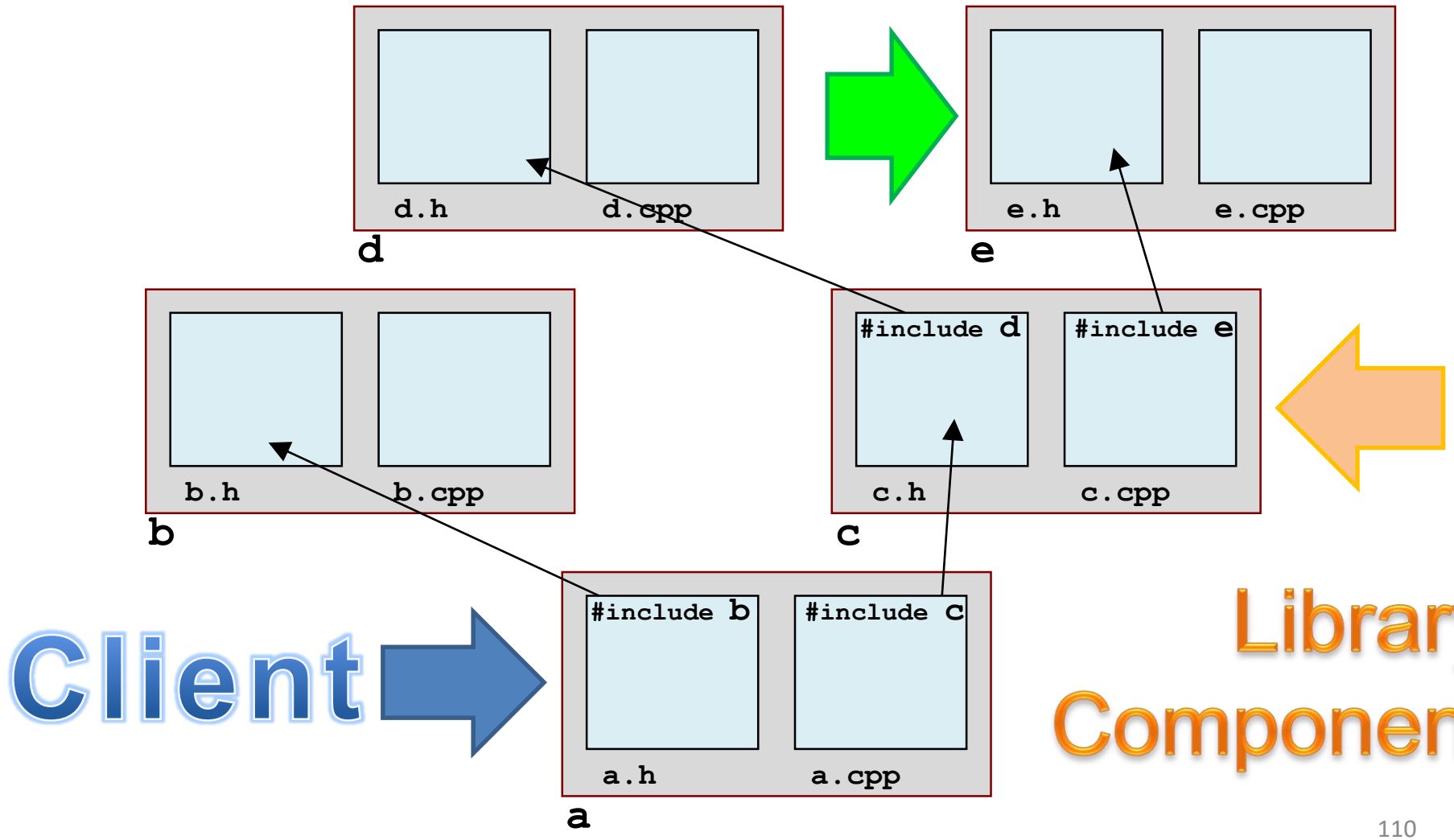
Library Component

Insulation


d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
#include b
#include c
a.h
a.cpp
a
Client
Library
Component

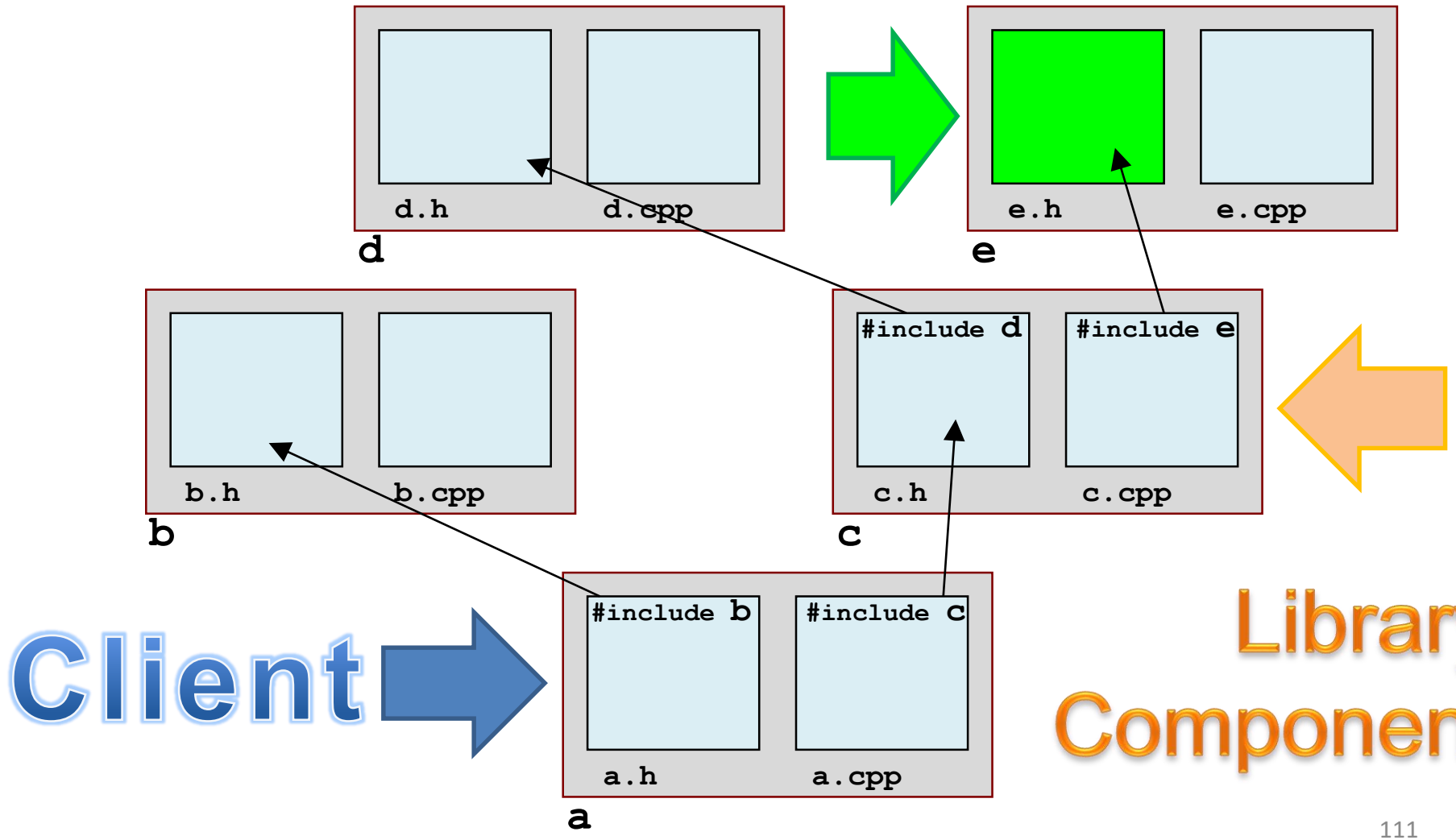
Insulation

Imp
Detail 'E'



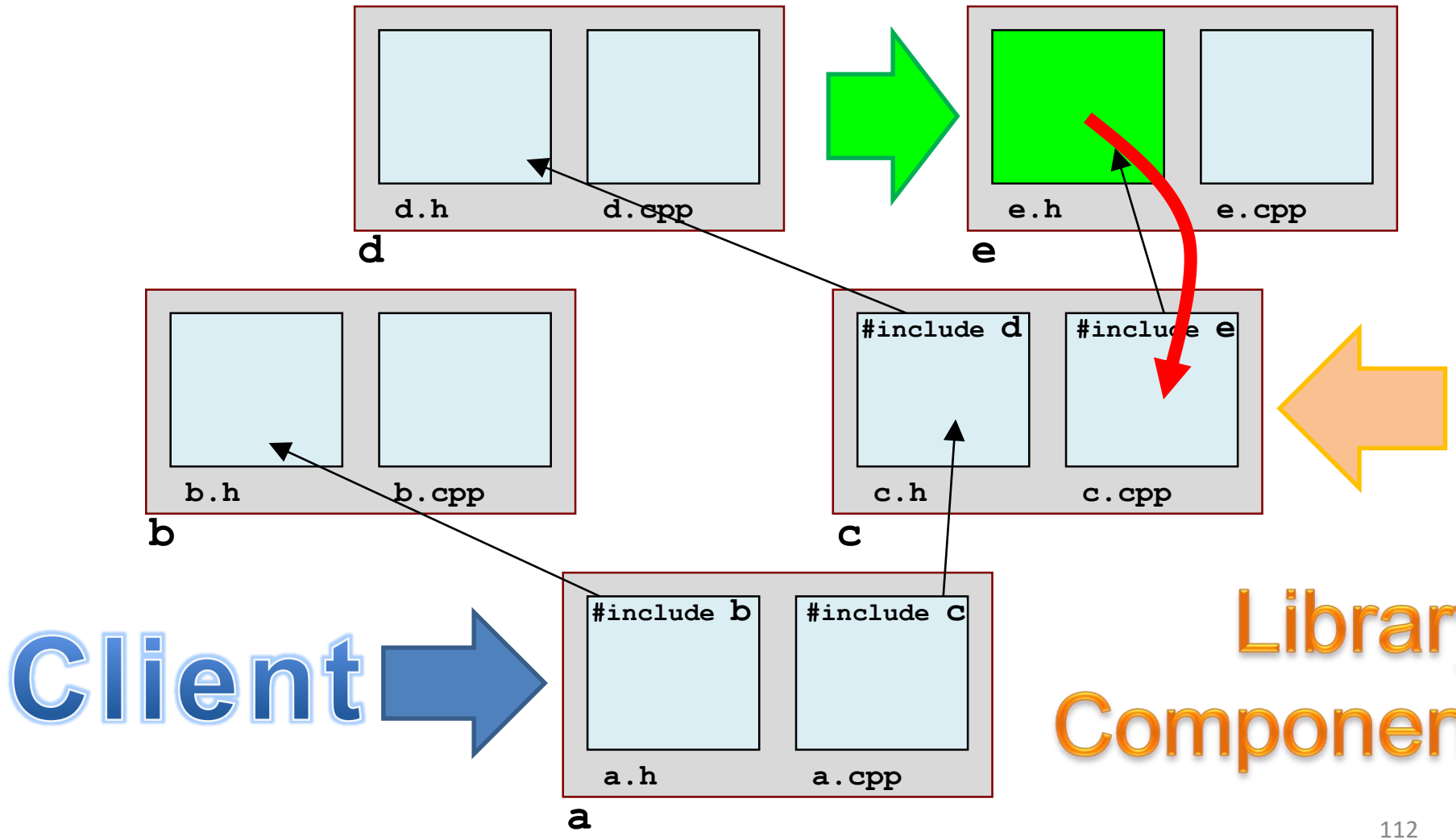
Insulation

Imp
Detail 'E'



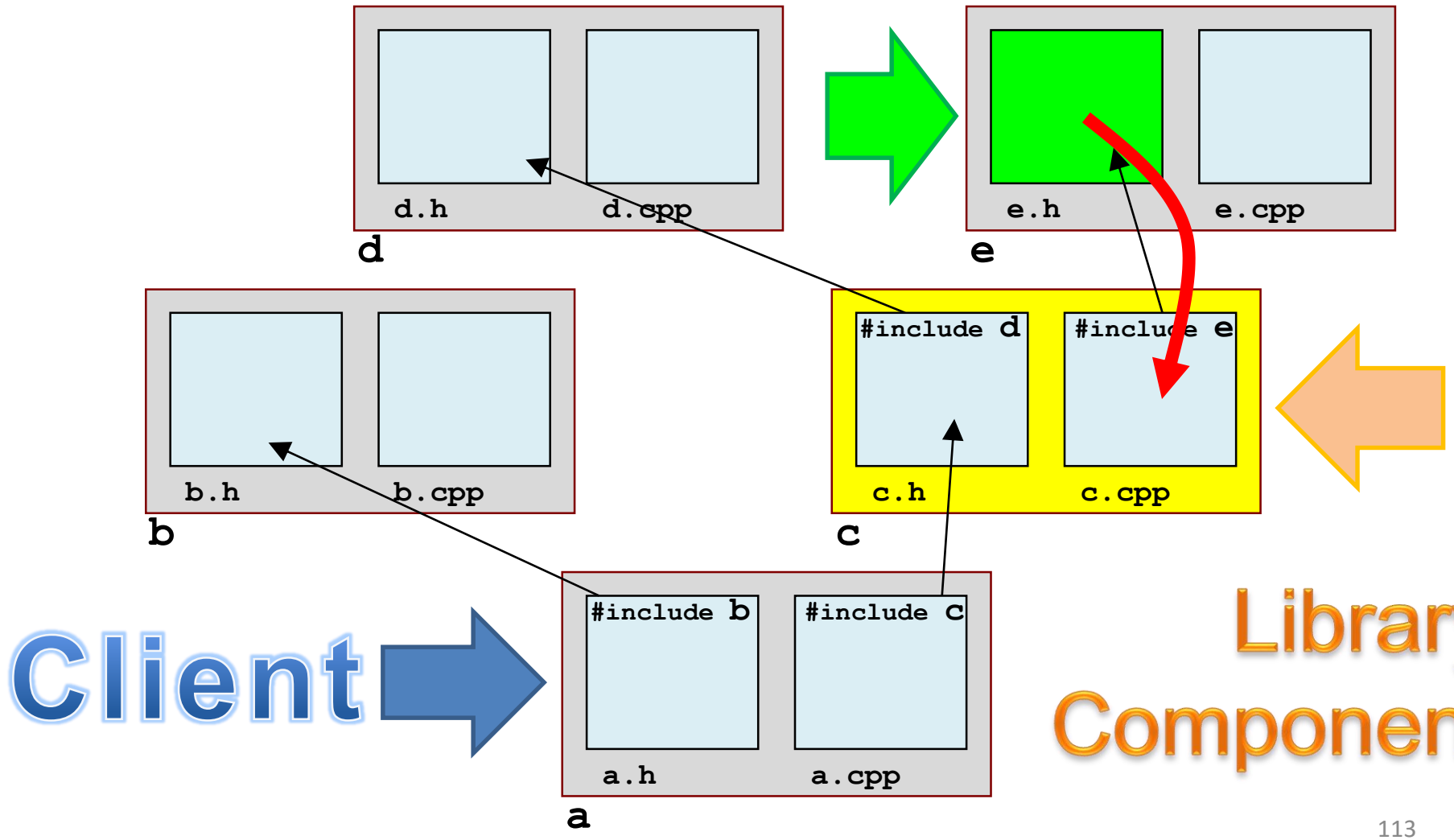
Insulation

Imp
Detail 'E'

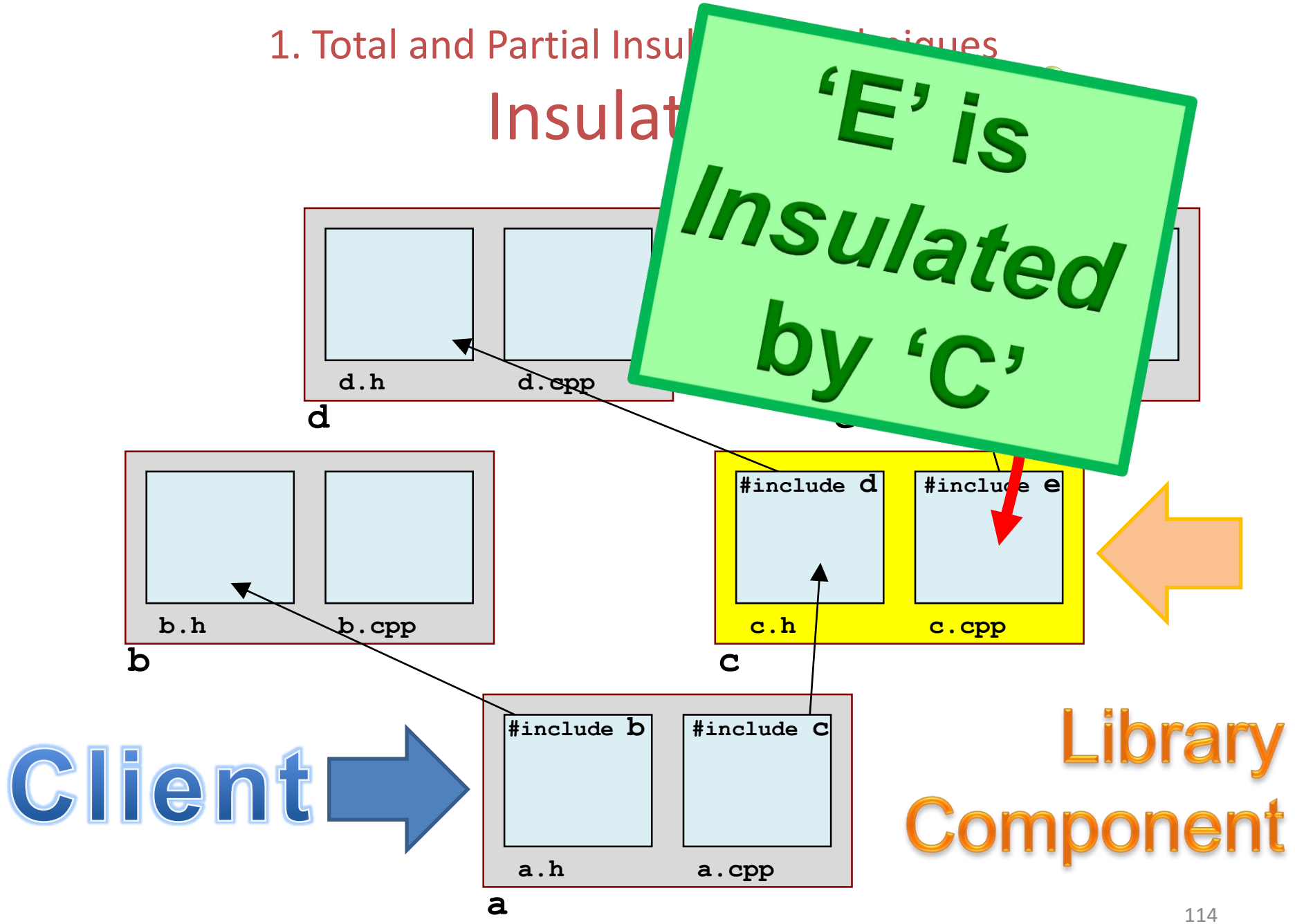


Insulation

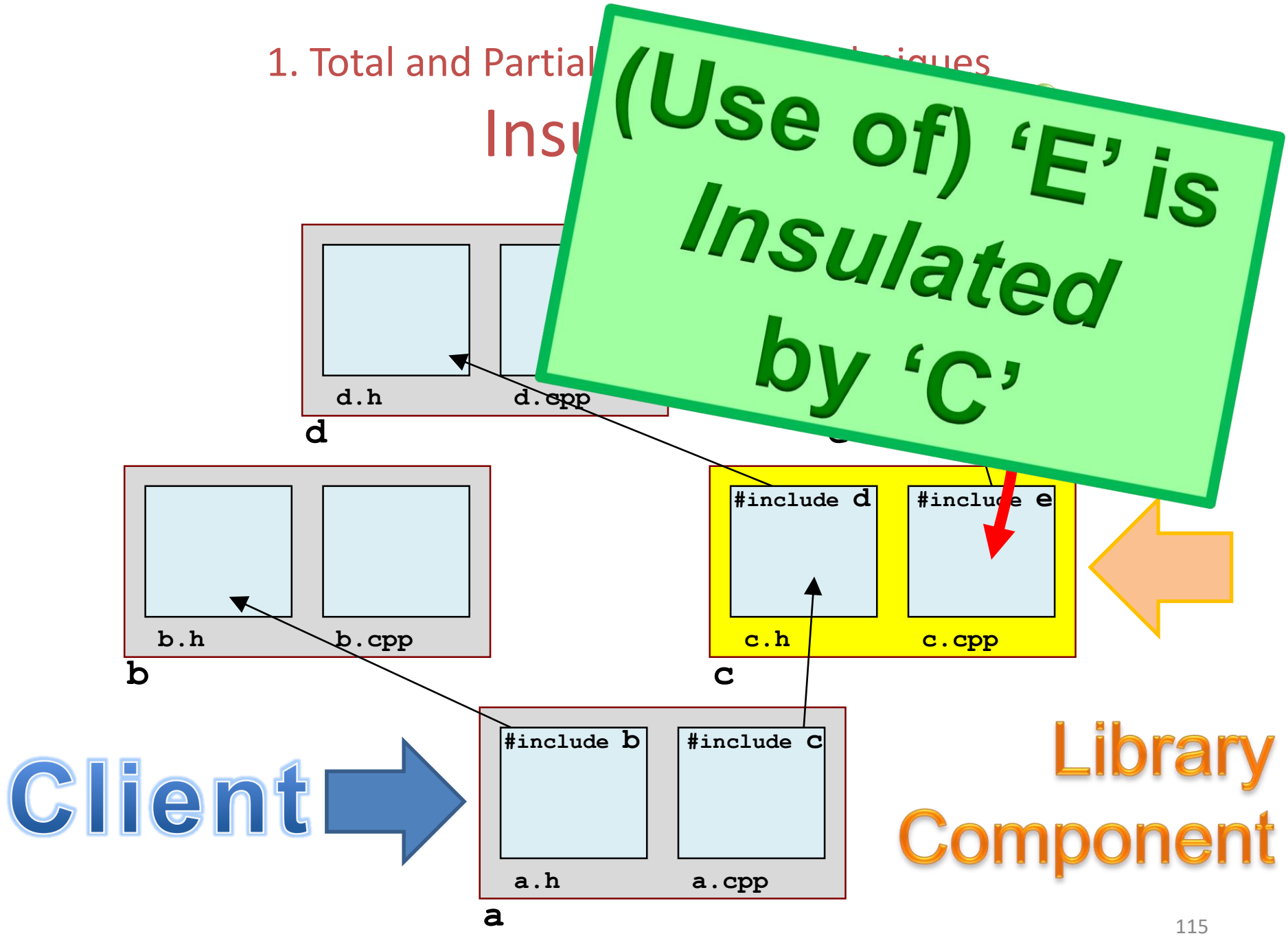
Imp
Detail 'E'



Insulat


'E' is Insulated by 'C'
d.h
d.cpp
d
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
#include b
#include c
a.h
a.cpp
a
Client
Library Component

Insu...


(Use of) 'E' is Insulated by 'C'
d.h
d.cpp
d
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
#include b
#include c
a.h
a.cpp
a
Client
Library Component

Criteria for having a `#include` in a `.h` File

Criteria for having a `#include` in a `.h` File

Recall that:

A header file must be "self-sufficient" w.r.t. compilation.

Criteria for having a `#include` in a `.h` File

There are five:

Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*

Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

} *But not Uses !*

Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

} *But not Uses !*

```
#include <point.h>
Point appendVertex(int                  index,
                   const Point& vertex);
```

Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

} *But not Uses !*

```
#include <point.h>
Point appendVertex(int                  index,
                   const Point&         vertex);
```

Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

} *But not Uses !*

```
class Point;
Point appendVertex(int                  index,
                   const Point&         vertex);
```

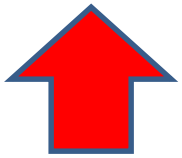
Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

} *But not Uses !*

```
#include <point.h>
Point appendVertex(int                  index,
                   const Point& vertex);
```



Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

} *But not Uses !*

```
class Point;
Point appendVertex(int                  index,
                   const Point& vertex);
```

Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

Criteria for having a `#include` in a `.h` File

There are five:

1. Is-A

2. Has-A

`3.inline` (used in function body)

Criteria for having a `#include` in a `.h` File

There are five:

1. Is-A

2. Has-A

`3.inline` (used in function body)

`4.enum`

Criteria for having a `#include` in a `.h` File

There are five:

1. Is-A

2. Has-A

`3.inline` (used in function body)

`4.enum`

`5.typedef` (e.g., template specialization)

Criteria for having a `#include` in a `.h` File

There are five:

1. Is-A

2. Has-A

`3.inline` (used in function body)

`4.enum`

`5.typedef` (e.g., template specialization)

Note: Covariant return types is another edge case.

End of Section

Questions?

What Questions are we Answering?

- What distinguishes *Logical* from *Physical* Design?
- What are the fundamental properties of a component?
- How do we infer dependencies from logical relationships?
- What are *level numbers*, and how do we determine them?
- How do we extract component dependencies efficiently?
- What essential physical design rules must be followed?
- What are the criteria for collocating classes & functions?
- What do we mean by the term *Insulation*?
- How does *Insulation* compare with *Encapsulation*?
- Why/when would we put a `#include` directive in a `.h` file?
- What cost/*benefit* is generally associated with insulation

What Questions are we Answering?

- What distinguishes *Logical* from *Physical* Design?
- What are the fundamental properties of a component?
- How do we infer dependencies from logical relationships?
- What are *level numbers*, and how do we determine them?
- How do we extract component dependencies efficiently?
- What essential physical design rules must be followed?
- What are the criteria for collocating classes & functions?
- What do we mean by the term *Insulation*?
- How does *Insulation* compare with *Encapsulation*?
- Why/when would we put a `#include` directive in a `.h` file?
- What cost/*benefit* is generally associated with insulation

What Questions are we Answering?

- What distinguishes *Logical* from *Physical* Design?
- What are the fundamental properties of a component?
- How do we infer dependencies from logical relationships?
- What are *level numbers*, and how do we determine them?
- How do we extract component dependencies efficiently?
- What essential physical design rules must be followed?
- What are the criteria for collocating classes & functions?
- What do we mean by the term *Insulation*?
- How does *Insulation* compare with *Encapsulation*?
- Why/when would we put a `#include` directive in a `.h` file?
- What cost/*benefit* is generally associated with insulation

What Questions are we Answering?

- What distinguishes *Logical* from *Physical* Design?
- What are the fundamental properties of a component?
- How do we infer dependencies from logical relationships?
- What are *level numbers*, and how do we determine them?
- How do we extract component dependencies efficiently?
- What essential physical design rules must be followed?
- What are the criteria for collocating classes & functions?
- What do we mean by the term *Insulation*?
- How does *Insulation* compare with *Encapsulation*?
- Why/when would we put a `#include` directive in a `.h` file?
- What cost/*benefit* is generally associated with insulation

Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Introduce the Notion of a `module` in C++

   Requirements: Comparison with Conventional Headers

3. Achieving Physical Aggregation in C++ Today

   Organizing Components into Packages and Package Groups

4. Packaging Libraries Using C++ Modules

   Abstraction: Providing Refined Views on Existing Software

Outline

1. Review of Elementary Physical Design
   - Components, Modularity, Physical Dependencies
2. Introduce the Notion of a `module` in C++
   - Requirements: Comparison with Conventional Headers
3. Achieving Physical Aggregation in C++ Today
   - Organizing Components into Packages and Package Groups
4. Packaging Libraries Using C++ Modules
   - Abstraction: Providing Refined Views on Existing Software

2. Introduce the Notion of a module in C++

Business Requirements for Modules

1. Introduction and Purpose

Modules are considered to be a **critically needed language feature** by many C++ developers, but the reasons for the urgency vary considerably from one engineer to the next. Some are looking, primarily, to **reduce protracted build times** for template-ladened header files (e.g., with **build artifacts**). Others want to use modules as a vehicle to **clean up impure vestiges of the language, such as macros**, that leak out into client code. Still others are looking to **"modernize" the way we view C++** rendering completely — even if it means forking the language. These are all very different motivations, and they may or may not be entirely compatible, but **if the agreed-upon implementation of modules does not take into account established code bases**, such as Bloomberg's, **they will surly fall far short of wide-spread adoption by industry.**

2. Introduce the Notion of a module in C++

Business Requirements for Modules

The primary purpose of this paper is to serve as a proxy for discussion regarding critically important requirements for substantial software organizations, such as Bloomberg, that have very specific architectural needs, yet also have vast amounts of legacy source code that cannot reasonably be migrated to a new syntax in any bounded amount of time.

Business Requirements for Modules

2. Current Situation

Some of the **strategies require existing code bases to change** before they can take advantage of modules. **Significant work has gone into tooling** that converts existing code bases to become "modularized**", replacing conventional `.h`/`.cpp` pairs with the equivalent in module syntax**, import statements in place of **`#include`** directives, etc. For companies, like Bloomberg, that have an enormous sprawling code base along with numerous disparate clients at every level of the software's physical hierarchy**, any approach that requires transforming the entire codebase along with all the clients is a non-starter.**

2. Introduce the Notion of a module in C++

Business Requirements for Modules

Don Knuth asserted that premature optimization is the root of all evil. Any sensible implementation of modules will enable the kind of compile-time optimizations we are all looking for, but the converse is not true. If we come up with an **optimization-oriented implementation of modules and release it first**, it will be **impossible to graft on the necessary architecture-oriented features** that would make modules realize their potential value for large-scale C++ software designers and architects. If we are to be truly successful, we must start with a fully-baked design; only after that should we attempt to optimize it.

2. Introduce the Notion of a module in C++

Business Requirements for Modules

In order for any new module technology to have a plausibly successful path to adoption, its **integration must be (purely) additive, hierarchical, incremental, and interoperable, but not necessarily backward compatible** with traditional rendering (e.g., `.h`/`.cpp` pairs). By **(purely) additive**, we mean that providing a module-style interface to existing **code does not require that code to be modified** (in any way whatsoever). By **hierarchical**, we mean that what we **add to an existing code base** to provide module interfaces depends on that code base (and never vice versa). By **incremental**, we mean that adding a module interface to one part of the code **base never implies adding it to some other, disparate part** of the code base. Finally, by **interoperable**, we mean that a **C++ construct consumed through both a module interface and a (conventional) header-file interface** is understood by the client's compiler to be the same construct **without violating the ODR**.

2. Introduce the Notion of a module in C++

Business Requirements for Modules

3. High-Level Requirements

Modules will realize their full potential as an important new feature of C++ only if:

I. Modules deliver effective support for a **larger, more powerful unit of logical and physical architectural abstraction**, beyond what is currently realizable using conventional `.h`/`.cpp` pairs to form components compiled as separate translation units.

a. **Logical versus physical encapsulation**. Today, if I have a private data member, my client needs to see the definition of that data member. Modules should allow that definition to be exported to the client's compiler, but not to the client, for arbitrary reuse. In this way, modules fix an important and **pervasive problem: transitive includes.**

2. Introduce the Notion of a module in C++

Business Requirements for Modules

I.

a.

b. **Modules should be atomic with respect to compilation** for all of the elements they comprise. That is, **if I build a module containing templates and inline functions at a given level of contract assertions, the client will see that level**, rather than the level at which the client was build. While this is just an example, it should apply to any and all build options.

2. Introduce the Notion of a module in C++

Business Requirements for Modules

I.

a.

b.

c. **Modules can be used as views on existing software subsystems consisting of arbitrary numbers of `.h` and `.cpp` files**. That is, without changing an existing, conventionally implemented subsystem, **one can create a module interface (purely additively) that provides an arbitrary subset of the logical entities that the module comprises**. Ideally, but not necessarily initially, the level of filtering will enable one to drop below global entities to incorporate (or not) nested entities such as **individual member functions**. In this way a module does not encapsulate the original definition of the legacy code, but rather its use through this module interface. Finally it should be possible for **multiple modules to wrap the same conventional software as views aimed ad distinct clients** that converge to a single main. All of the entities exported should be **known to be the same with no ODR violation**.

2. Introduce the Notion of a module in C++

Business Requirements for Modules

I.

a.

b.

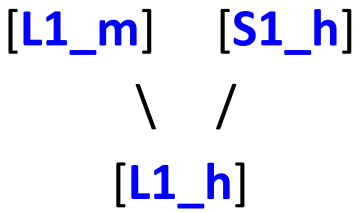
c.

d. **Modules that act as views should behave similarly to C procedural interfaces**. (See Lakos'96, section 6.5.1, pp. 425-445.) What I mean by that is that if a conventional TU is exposed in parallel with a modular view of that TU, then **a client importing entities from both will get the union of access**, and overlapping entities will be considered by the client's compiler as being the **same entity (without violating the ODR)**.

Business Requirements for Modules

II. **There exists a well-considered, viable adoption strategy that does NOT require existing software to be altered in any way** in order to begin to make use of the new features to allow new clients to consume legacy software.

a. Let's take a look at a real-world scenario. Suppose we have a **library, `L1_h`,** implemented as `.h` / `.cpp` pairs. Suppose further that we have a **subsystem, `S1`,** that depends on, and traffics in types defined in `L1_h` in its interface. Now suppose we want to add, hierarchically, a module interface for `L1_h`, which we'll call **`L1_m`**. The current state of affairs now looks roughly like this:

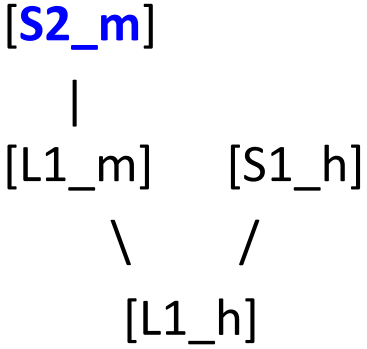


Business Requirements for Modules

II.

a.

b. Now suppose that we get another client subsystem written entirely in module speak, **`S2_m`**. This client has no legacy implementation and none of its sub-components are consumable by conventional renderings (which is "fine" because it is new code and no old code currently depends on it):



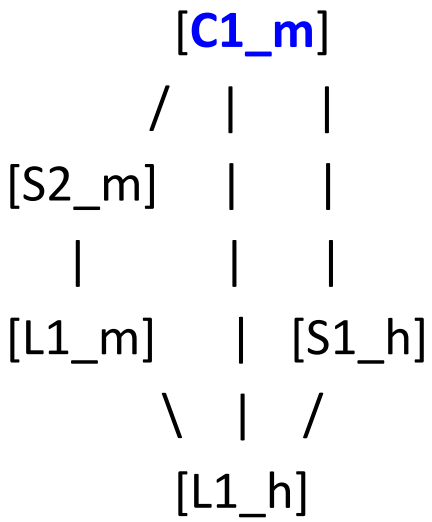
Business Requirements for Modules

II.

a.

b.

c. Finally a client, **`C1_m`** comes along and wants to use both `S2_m` and `S1_h`, both of which make use in their respective interfaces of types defined in `L1_h`:



Business Requirements for Modules

II.

a.

b.

c.

d. Types defined in **`L1_h`** and consumed from both **`S2_m`** and **`S1_h`** need to refer to the same entities. In this way, we can **keep our current code base while continuously evolving towards the "more modern" module only approach**. At some later point, `S1_m` may be created at which point `C1_m` **may or may not may want to convert to use it instead**, but now **all new code will benefit from using the more powerful, more modern, more efficient `S1_m`** rendering.

Business Requirements for Modules

III. **The implementation chosen does not require centralized repositories** or other known-to-be brittle techniques that would render important software processes such as distributed development or interaction with source-code control systems significantly more problematics than they already are.

a. **The Google approach seems to me to rely heavily on a module cache** which, from what I recall with template repositories from the 1990s was **sufficiently problematic that it ushered in the current linker technology** where template instantiations are duplicated locally in each translation unit in which they are used. (By "repository" here, I mean a cache of binary template instantiations that can be reused across translation units.)

Business Requirements for Modules

IV. **Once we have addressed I, II, and III,** it is assumed and expected that compile-times – especially for template-ladened interfaces – **will realize dramatic improvements over always fully reparsing source text in every translation unit.**

Business Requirements for Modules

4. Conclusion

There are many different competing ideas surrounding the design and implementation of modules in C++. There are many ways to realize modules in ways that address the requirements elucidated in this paper. It is hard for me to know, from what I have read, if and to what extent all of these requirements are addressed by the current proposal. It **is my intention that this paper serve as a proxy for a discussion** to learn more about where are currently, and where we need to be to move forward.

Review: Why Modules?

Some typical motivations

- Reduce compilation time.
- Eradicate macros.
- Change look and feel of C++.

Yet must not ignore a serious, real-world concern:

Large, legacy code bases!!

2. Introduce the Notion of a module in C++

Properties for Legacy Code

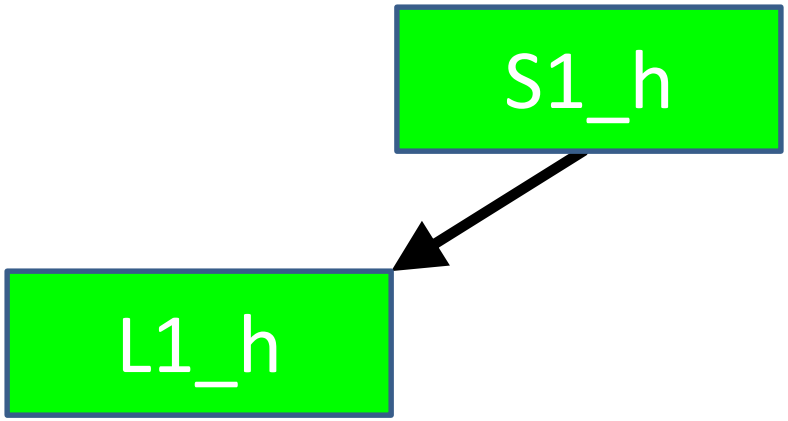
Property	Description
(Purely) Additive	Adding module interfaces need not require changes to existing code ***at all***.
Hierarchical	Added interfaces depend on the existing code, never *vice versa*.
Incremental	Module interfaces can be added individually, as needed (without requiring it of others).
Interoperable	A C++ construct consumed via a module is no different (w.r.t. ODR) from that same construct consumed via header file.

Enhancing C++ via Modules

- Fix the transitive **`#include`** problem. Provide private symbols for compilation, but not arbitrary reuse by clients.
- Contract-assertion level set by the module builder, not the builder of the client.
- ***Future:*** Modules could provide multiple views of a code base without violating the ODR.
  - Clients w/multiple views get the ***union*** those views.

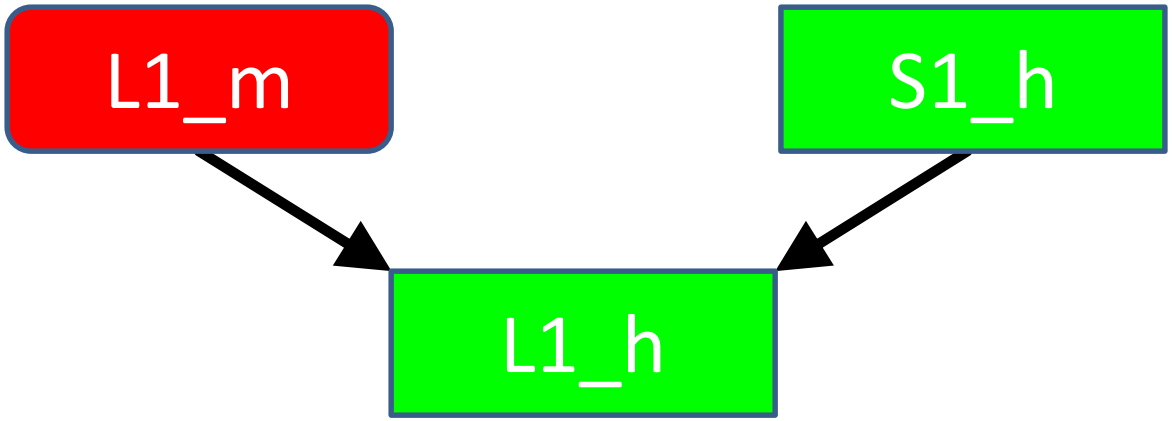
2. Introduce the Notion of a module in C++

Adoption Strategy



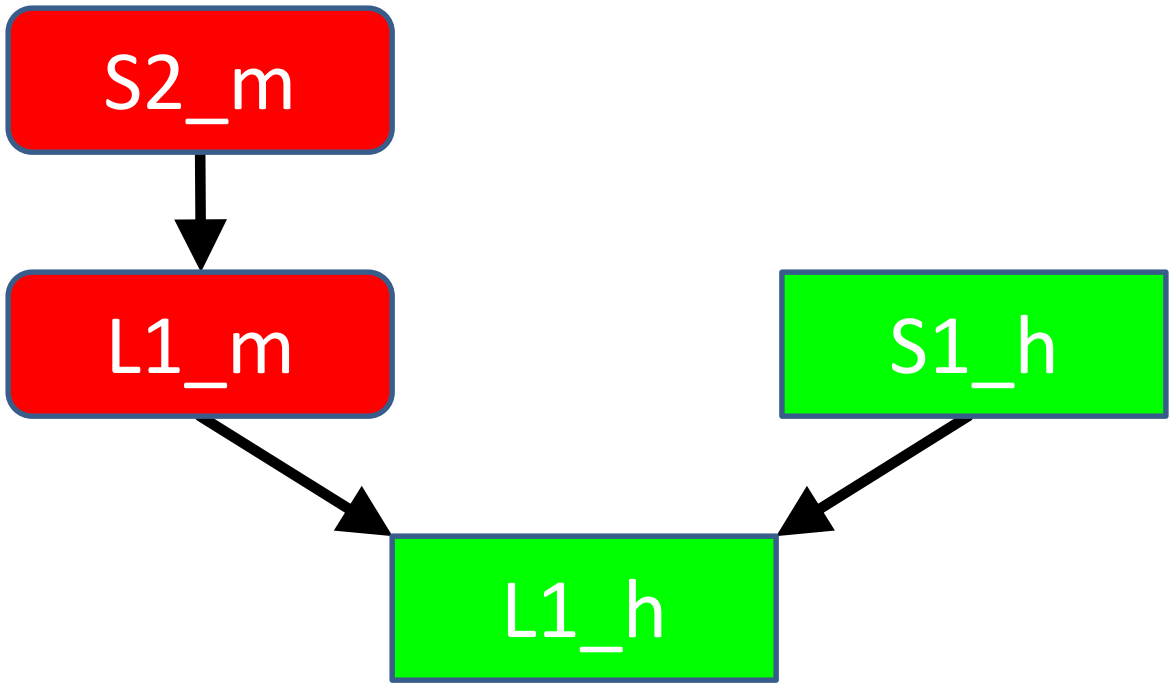
2. Introduce the Notion of a module in C++

Adoption Strategy



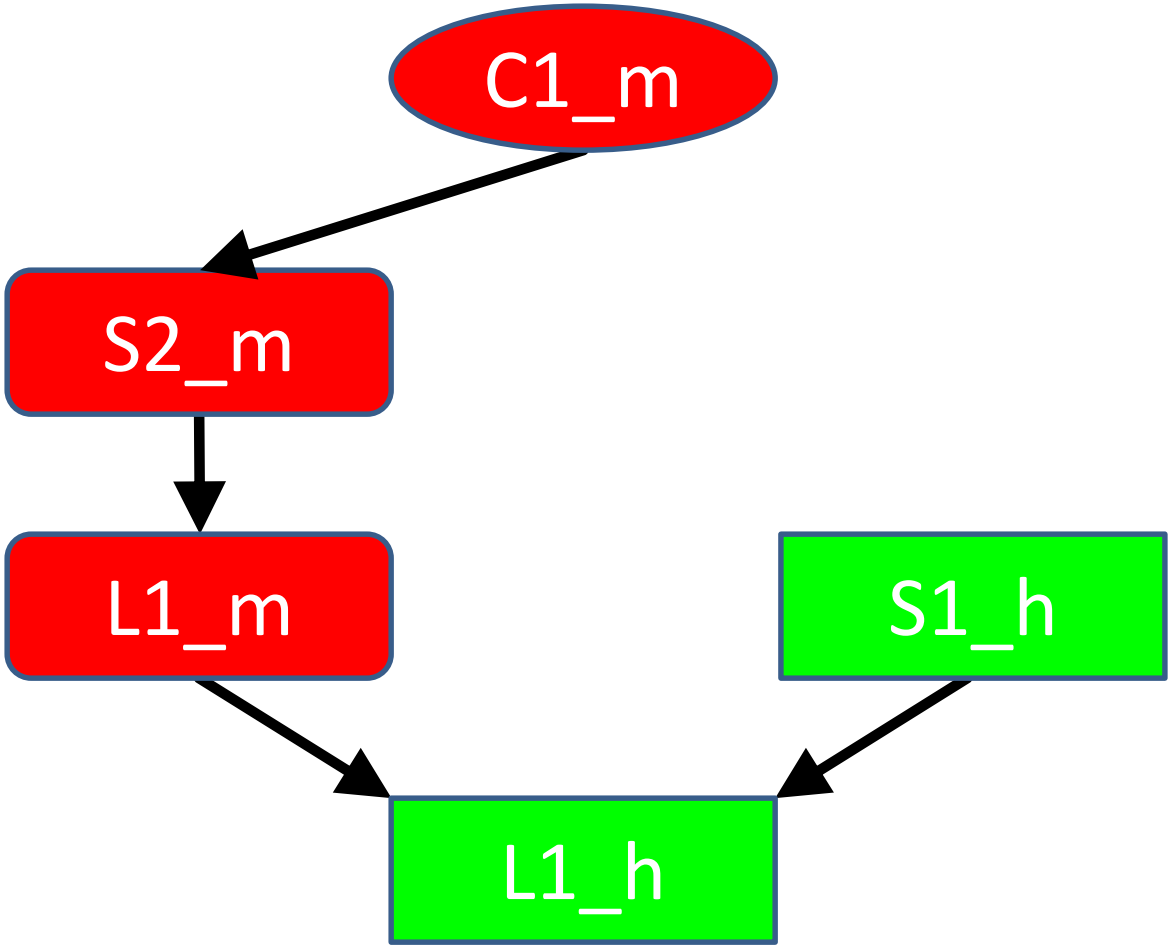
2. Introduce the Notion of a module in C++

Adoption Strategy



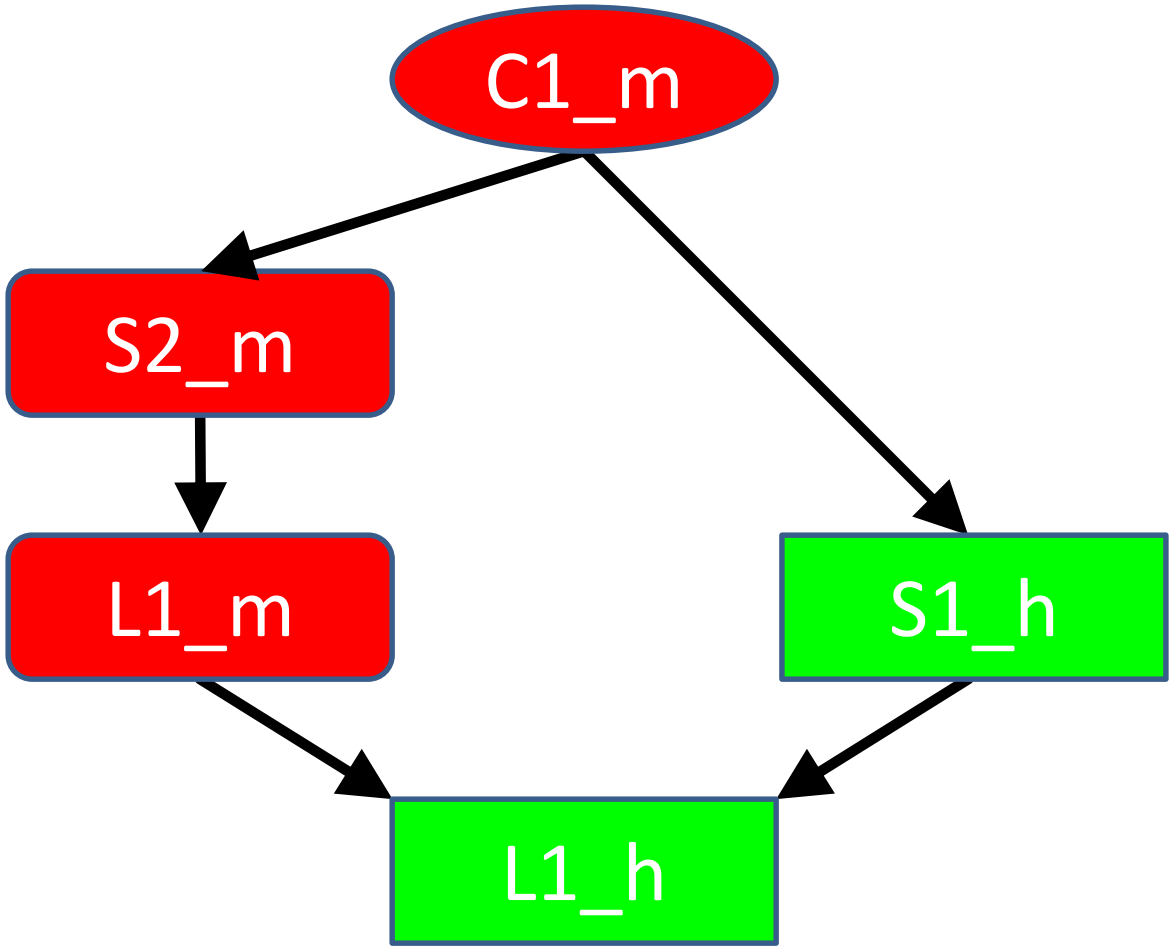
2. Introduce the Notion of a module in C++

Adoption Strategy



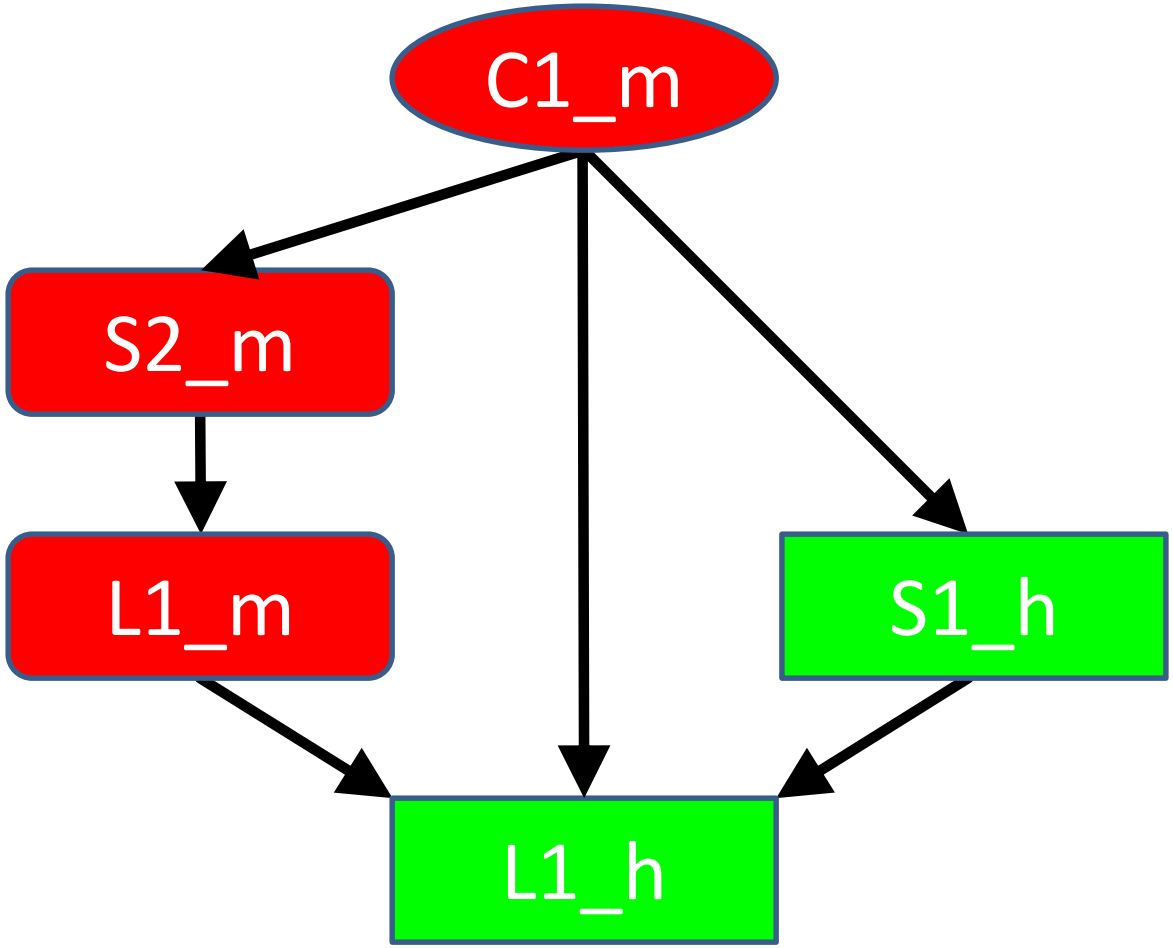
2. Introduce the Notion of a module in C++

Adoption Strategy



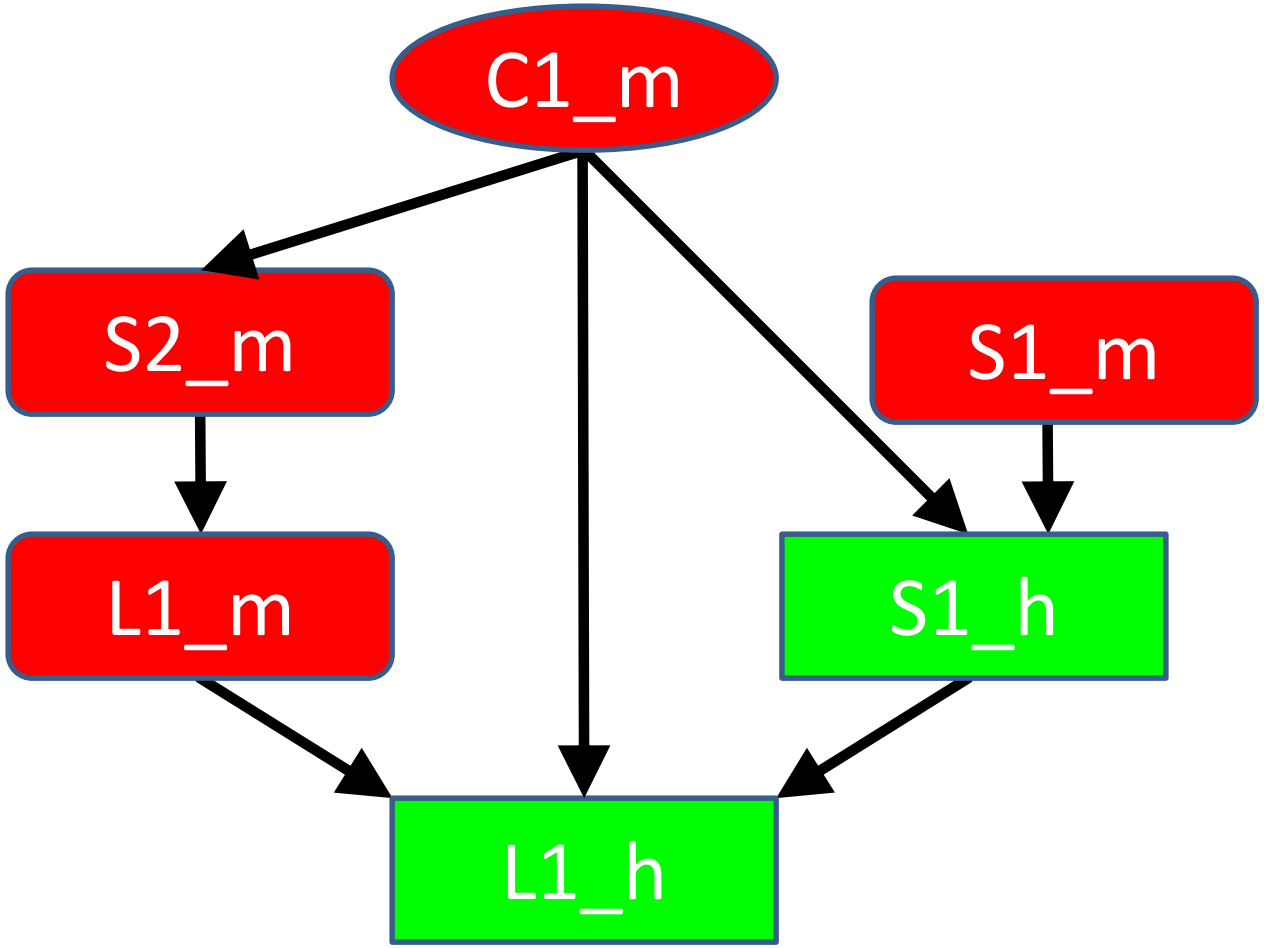
2. Introduce the Notion of a module in C++

Adoption Strategy



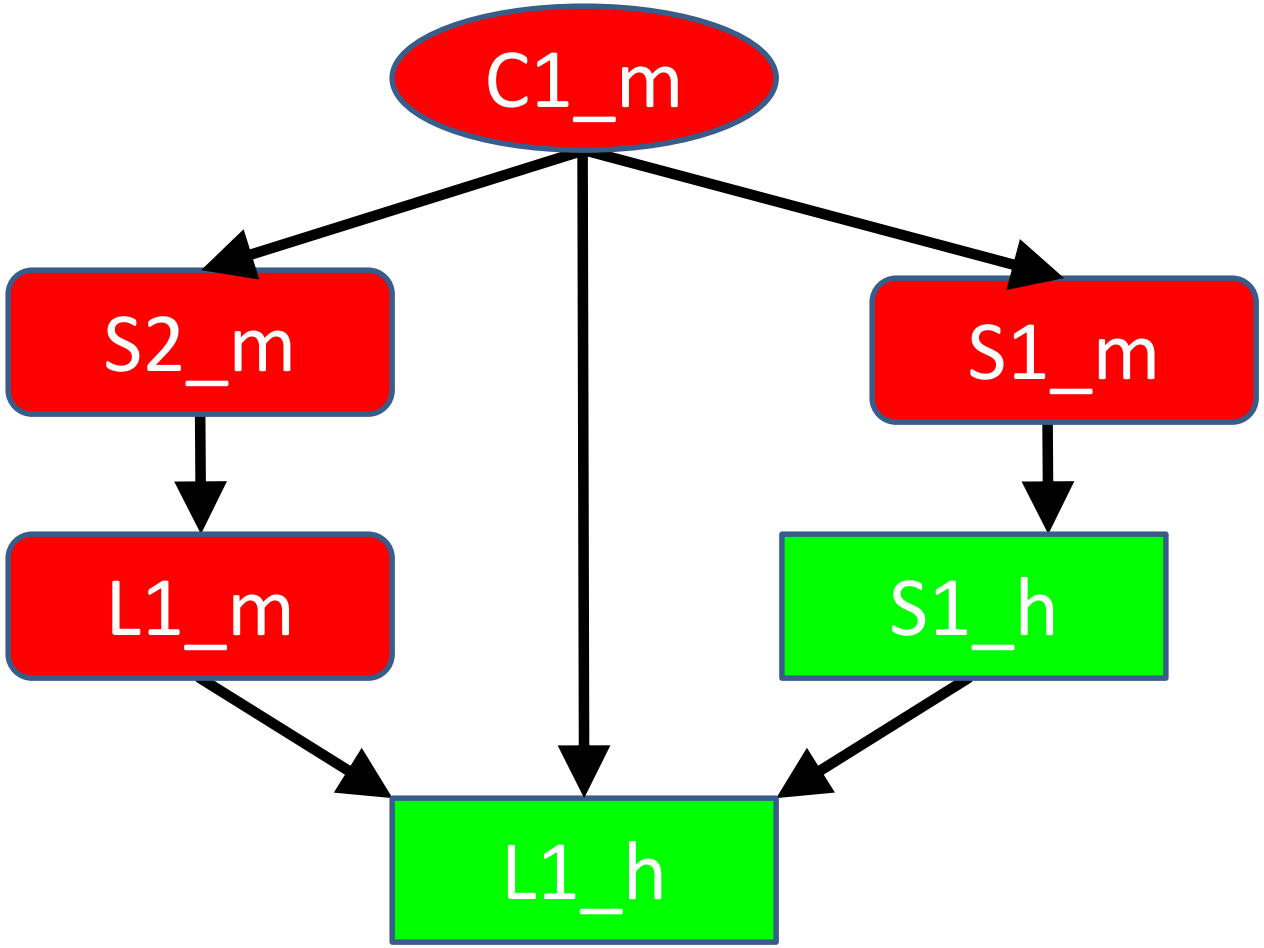
2. Introduce the Notion of a module in C++

Adoption Strategy



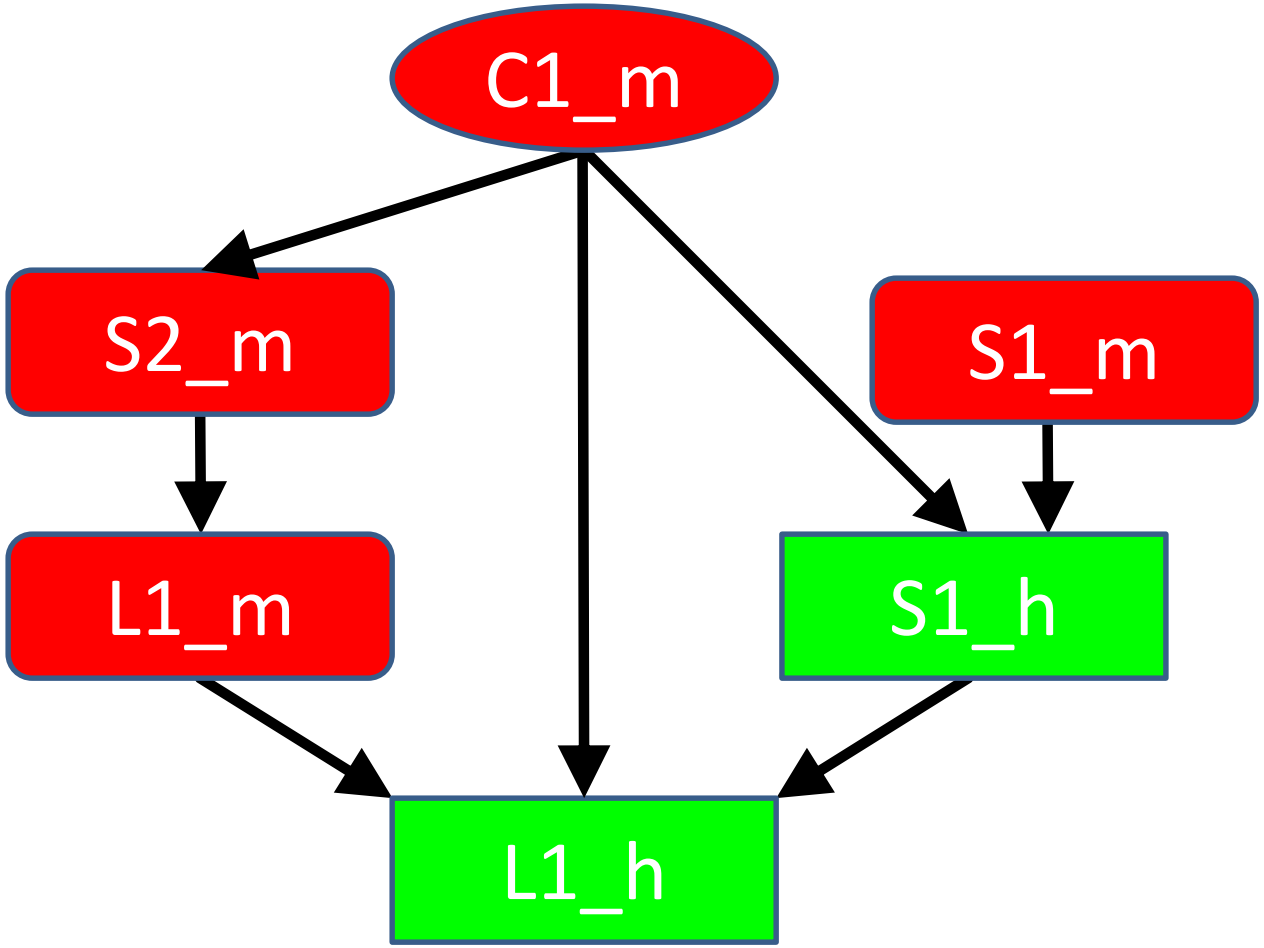
2. Introduce the Notion of a module in C++

Adoption Strategy



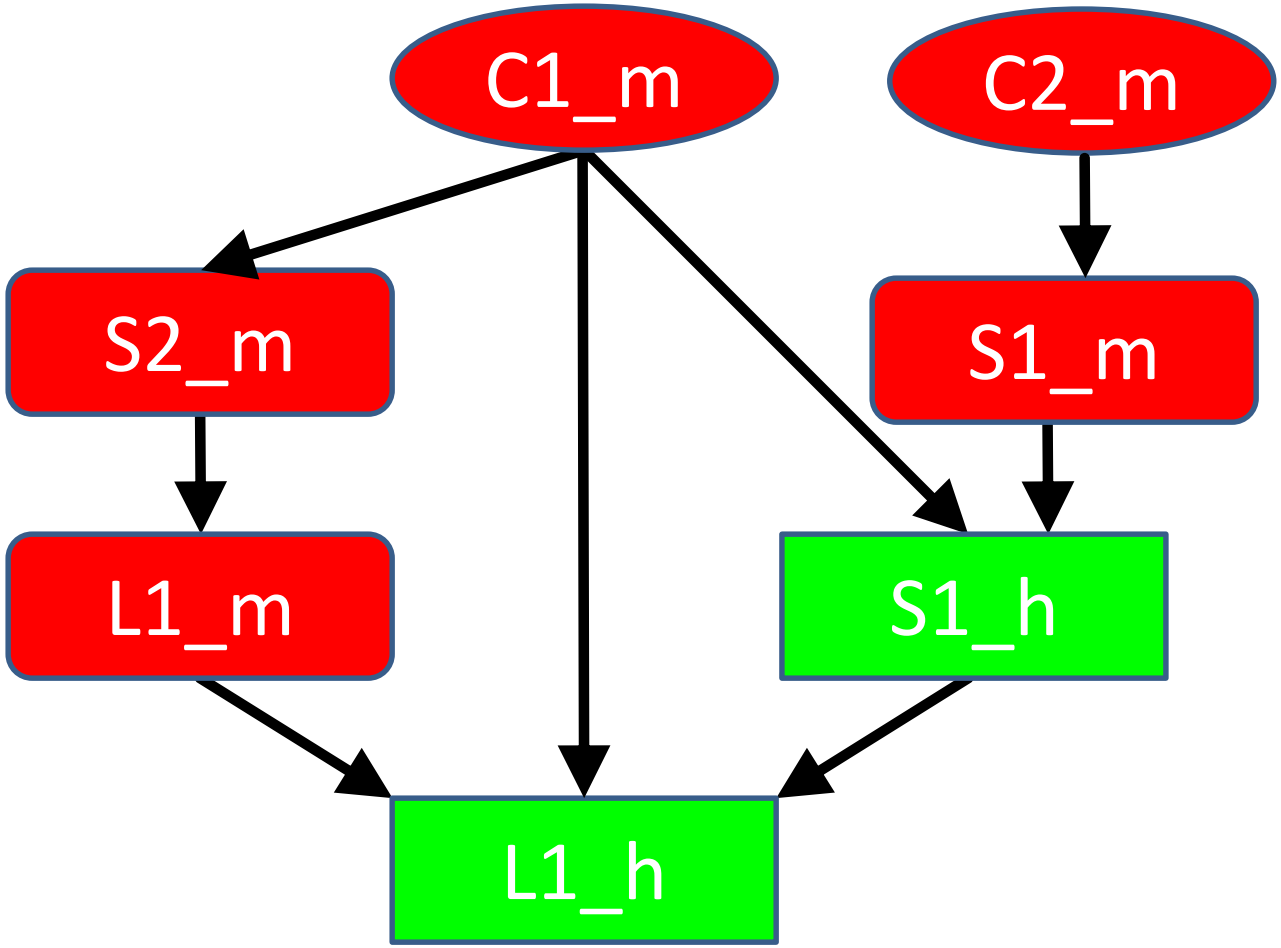
2. Introduce the Notion of a module in C++

Adoption Strategy



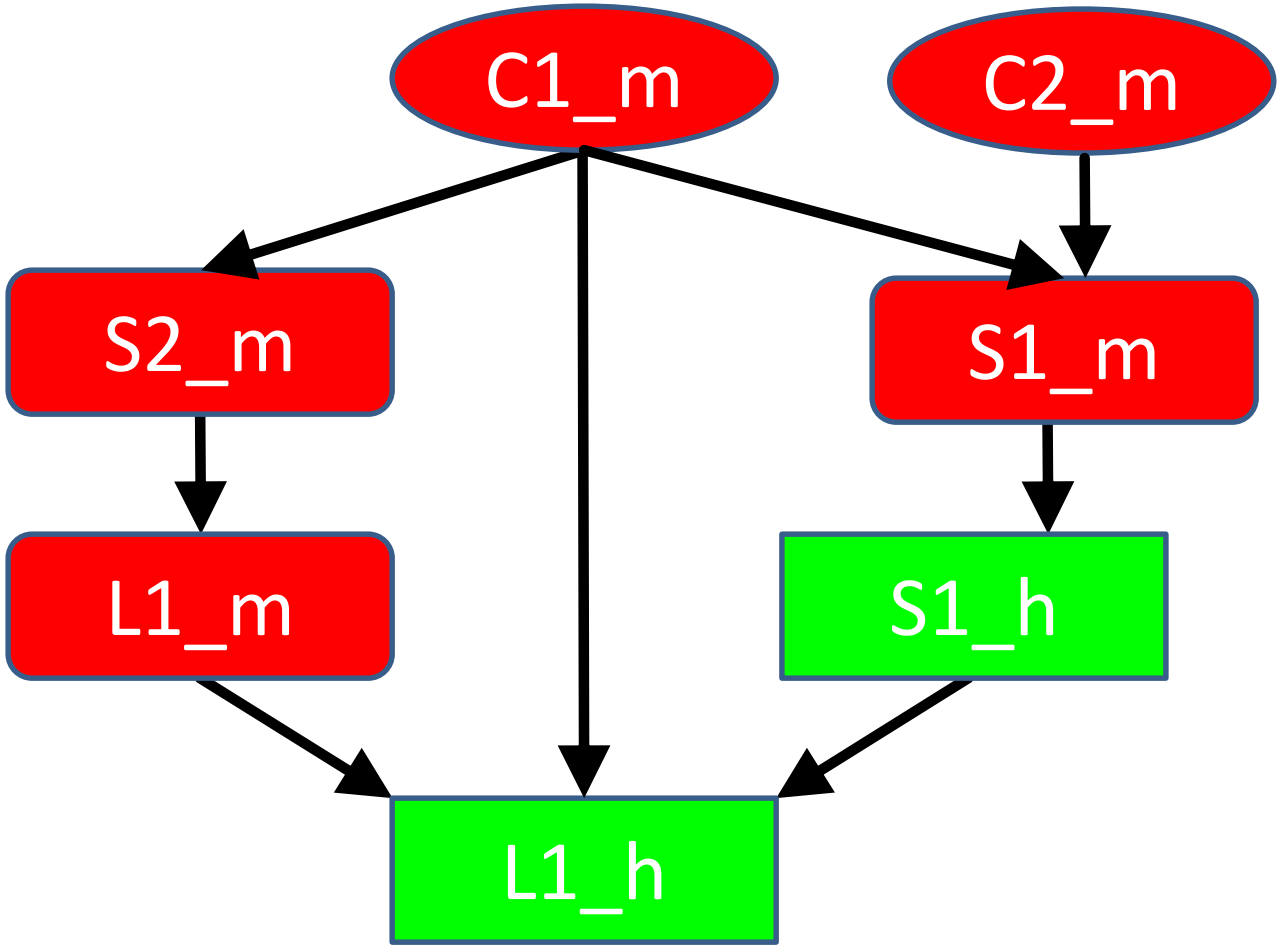
2. Introduce the Notion of a module in C++

Adoption Strategy



2. Introduce the Notion of a module in C++

Adoption Strategy



Additional Concerns

- Focus on compiler optimization (prematurely) might preclude needed architectural features.
- Implementations requiring centralized repositories (for faster builds) might impede distributed software development:
  - I.e., We need to be able to build any translation unit – independently of any other – directly from source code.
- **To soon** to commit to a specific module design.

End of Section

Questions?

2. Introduce the Notion of a module in C++

What Questions are we Answering?

- What are the engineering requirements for C++ Modules?

2. Introduce the Notion of a module in C++

What Questions are we Answering?

- What are the engineering requirements for C++ Modules?

Outline

1. Review of Elementary Physical Design
   - Components, Modularity, Physical Dependencies
2. Introduce the Notion of a `module` in C++
   - Requirements: Comparison with Conventional Headers
3. Achieving Physical Aggregation in C++ Today
   - Organizing Components into Packages and Package Groups
4. Packaging Libraries Using C++ Modules
   - Abstraction: Providing Refined Views on Existing Software

Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.

Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.
- **No** *cyclic* dependencies/*long-distance* friendships.

Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.
- **No** *cyclic* dependencies/*long-distance* friendships.
- Colocate logical constructs only with good reason:
  i.e., friendship; cycles; parts-of-whole; flea-on-elephant.

Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.
- **No** *cyclic* dependencies/*long-distance* friendships.
- Colocate logical constructs only with good reason:
  i.e., friendship; cycles; parts-of-whole; flea-on-elephant.
- Put a `#include` in a header only with good reason:
  i.e., *Is-A*, *Has-A*, `inline`, `enum`, `typedef`-to-template.

Conclusion

The End

We are hiring!

https://www.bloomberg.com/careers

Questions?

Bloomberg

Engineering

TechAtBloomberg.com

Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Introduce the Notion of a `module` in C++

   Requirements: Comparison with Conventional Headers

3. Achieving Physical Aggregation in C++ Today

   Organizing Components into Packages and Package Groups

4. Packaging Libraries Using C++ Modules

   Abstraction: Providing Refined Views on Existing Software

Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Introduce the Notion of a `module` in C++

   Requirements: Comparison with Conventional Headers

3. Achieving Physical Aggregation in C++ Today

   Organizing Components into Packages and Package Groups

4. Packaging Libraries Using C++ Modules

   Abstraction: Providing Refined Views on Existing Software

Physical Dependency

Five levels of **physical dependency**:

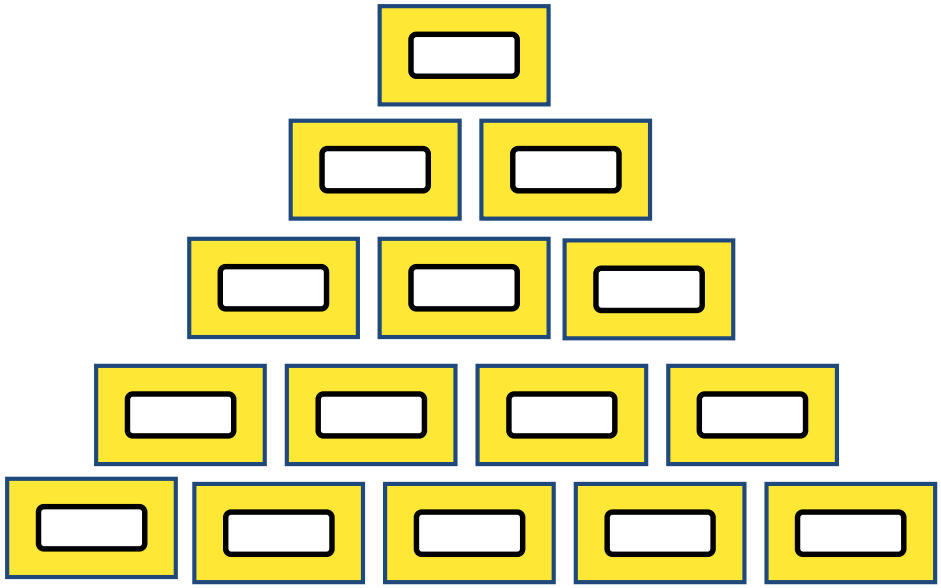
Level 5:

Level 4:

Level 3:

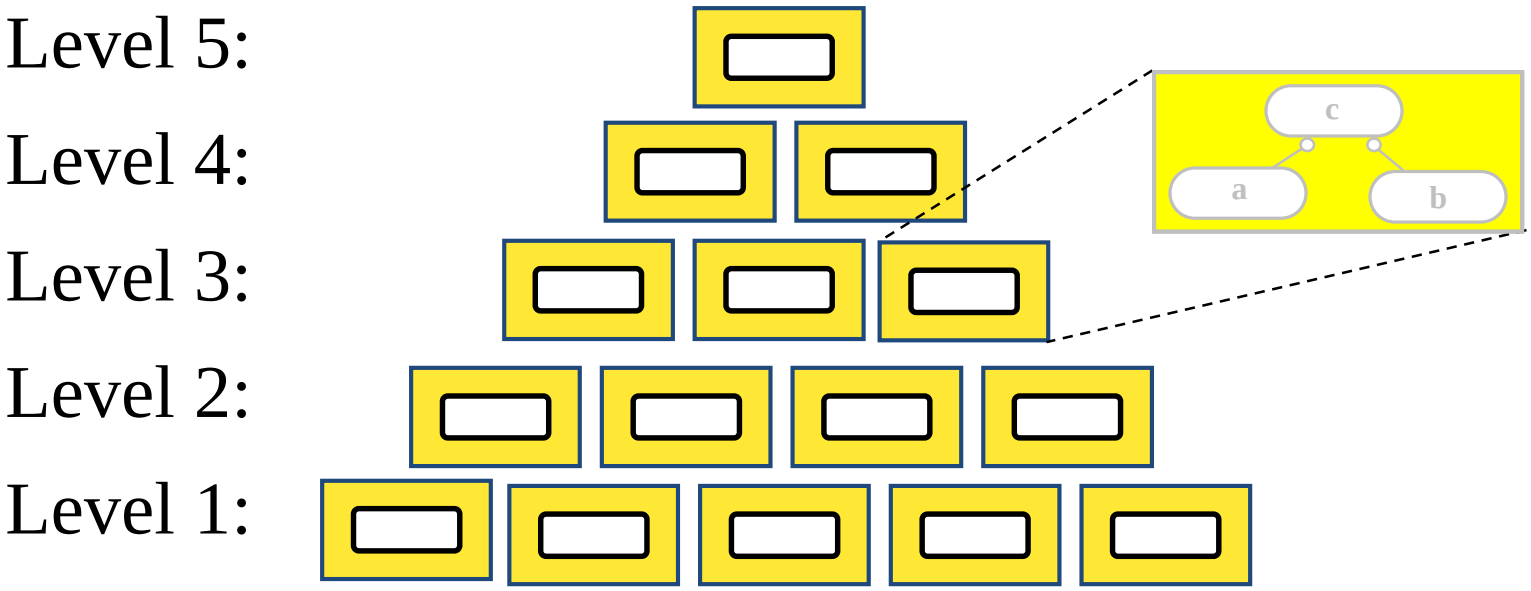
Level 2:

Level 1:



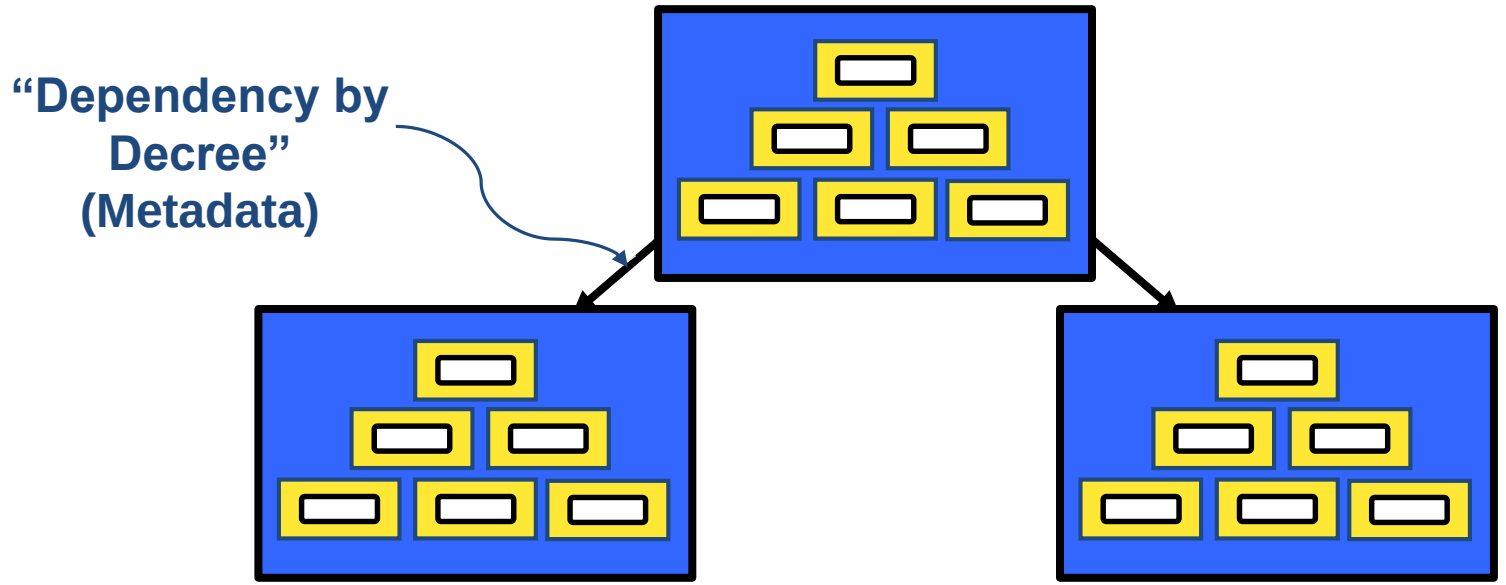
Physical Aggregation

Only one level of **physical aggregation**:



The Package

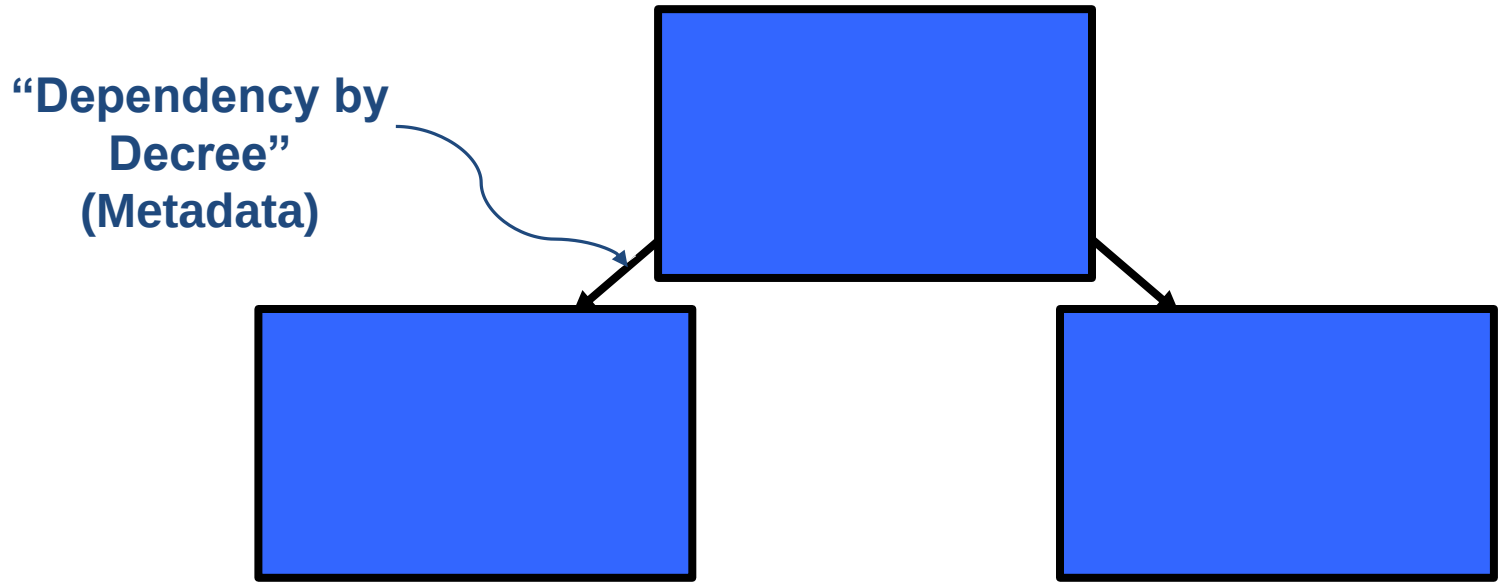
Two levels of physical aggregation:



"A Hierarchy of Component Hierarchies"

The Package

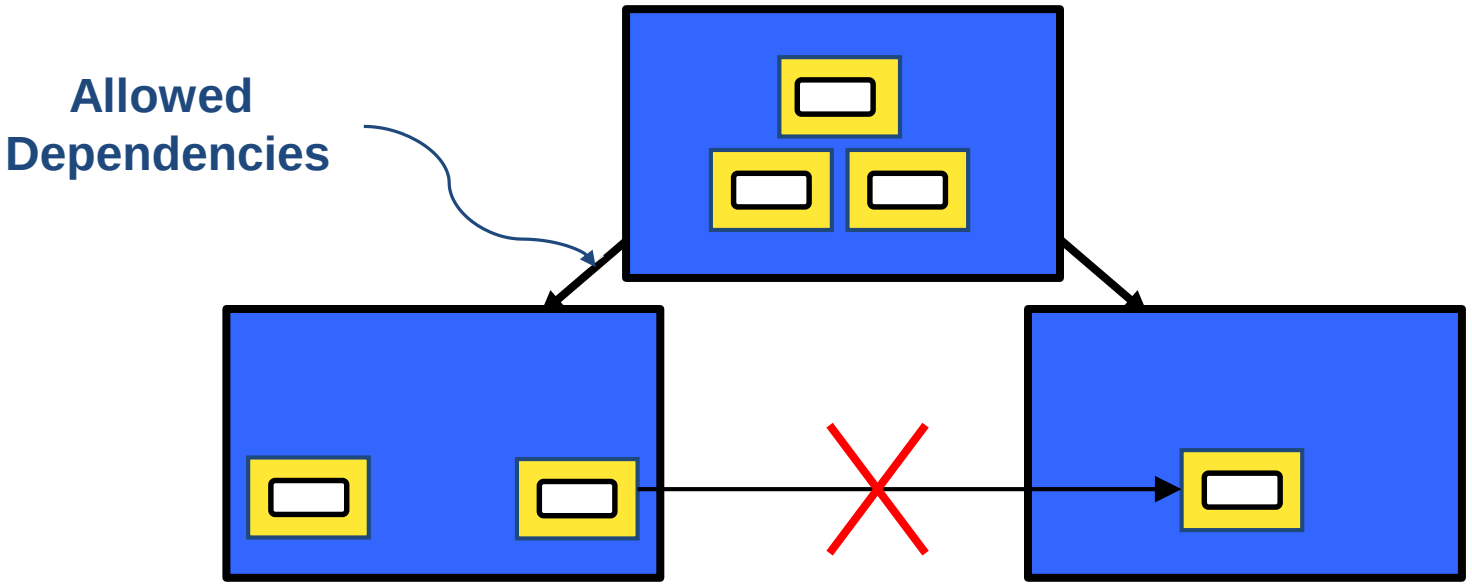
Two levels of physical aggregation:



Metadata governs, even absent of any components!

The Package

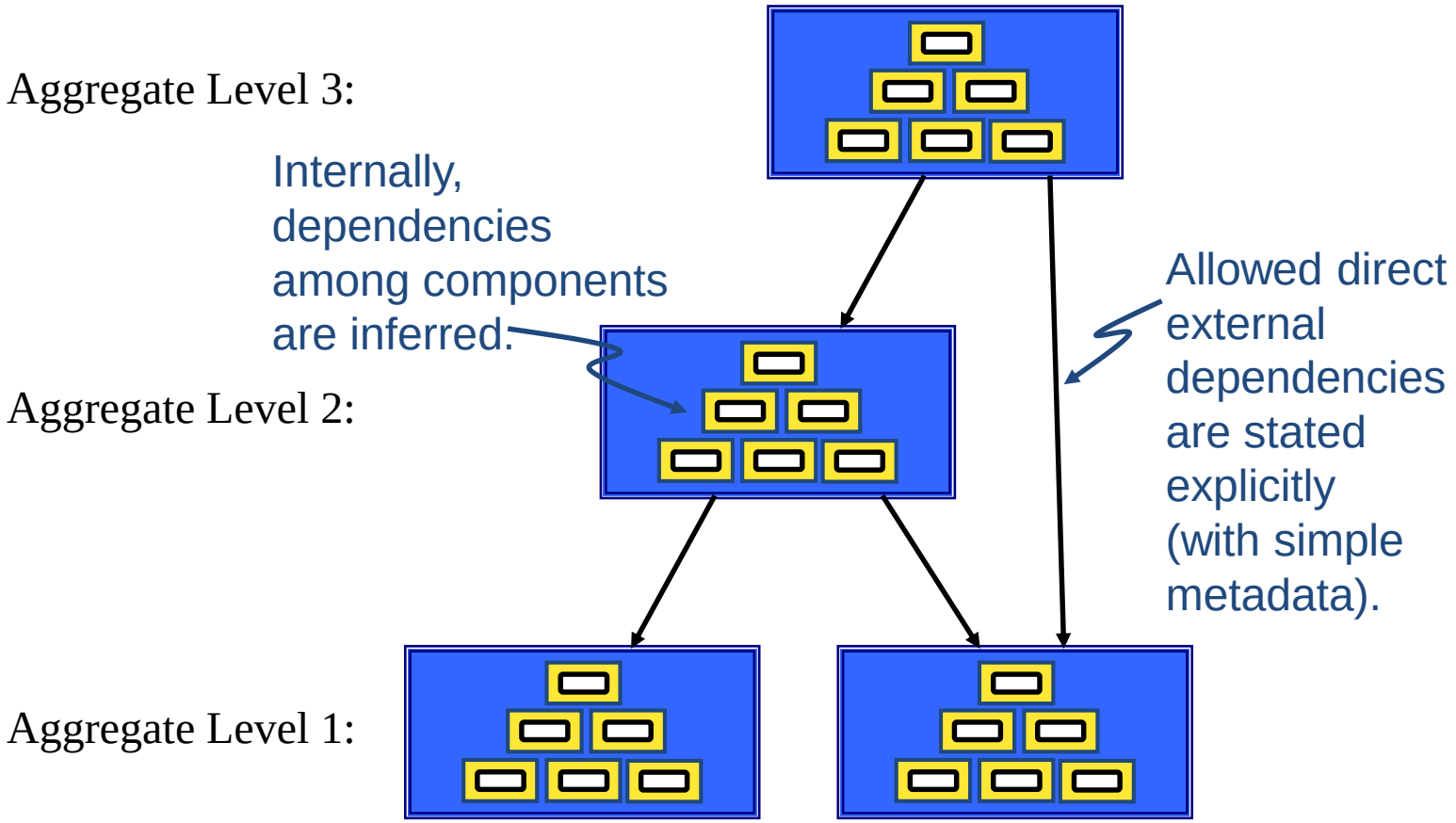
Two levels of physical aggregation:



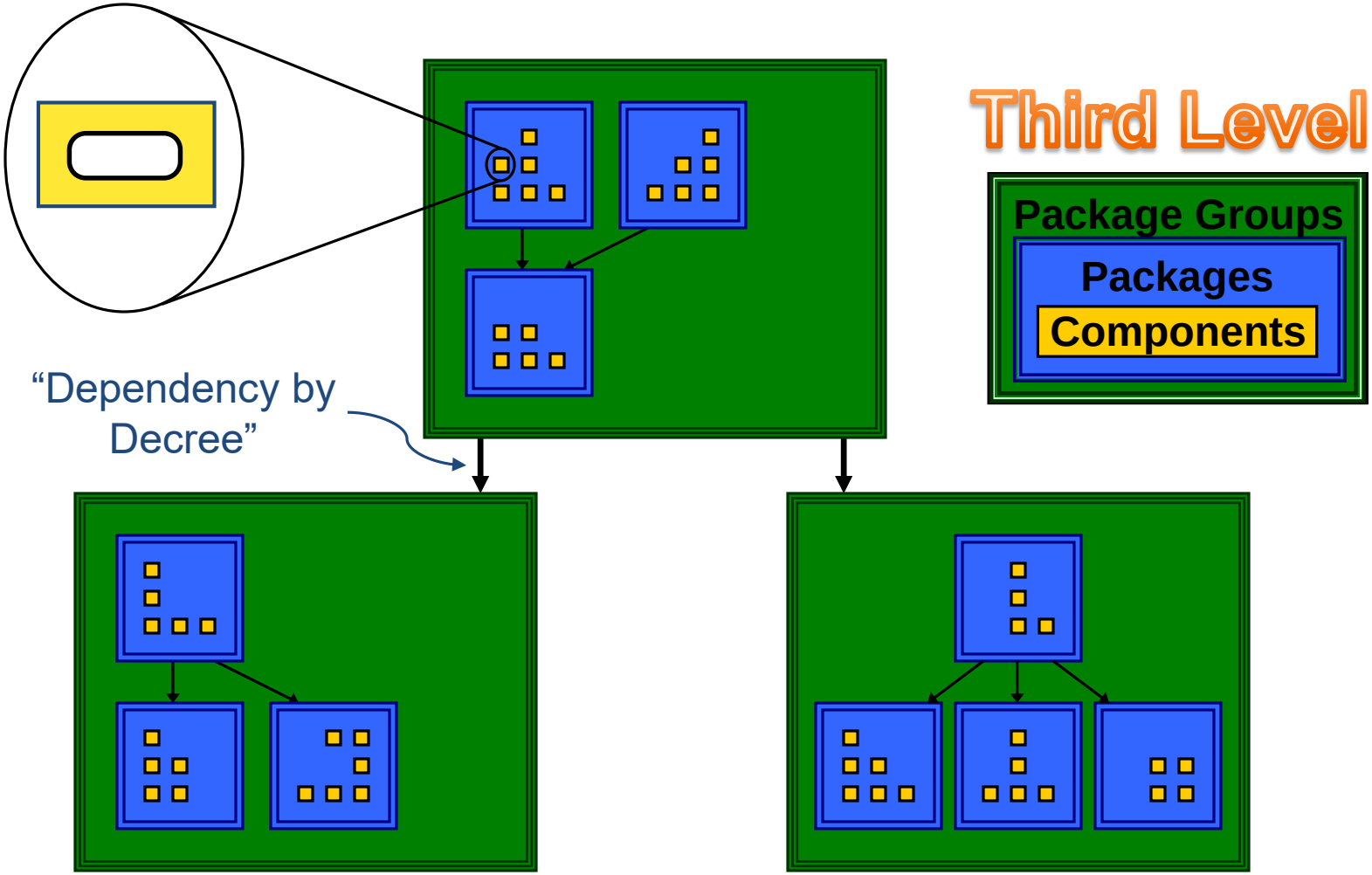
Metadata governs allowed dependencies.

Package Dependencies

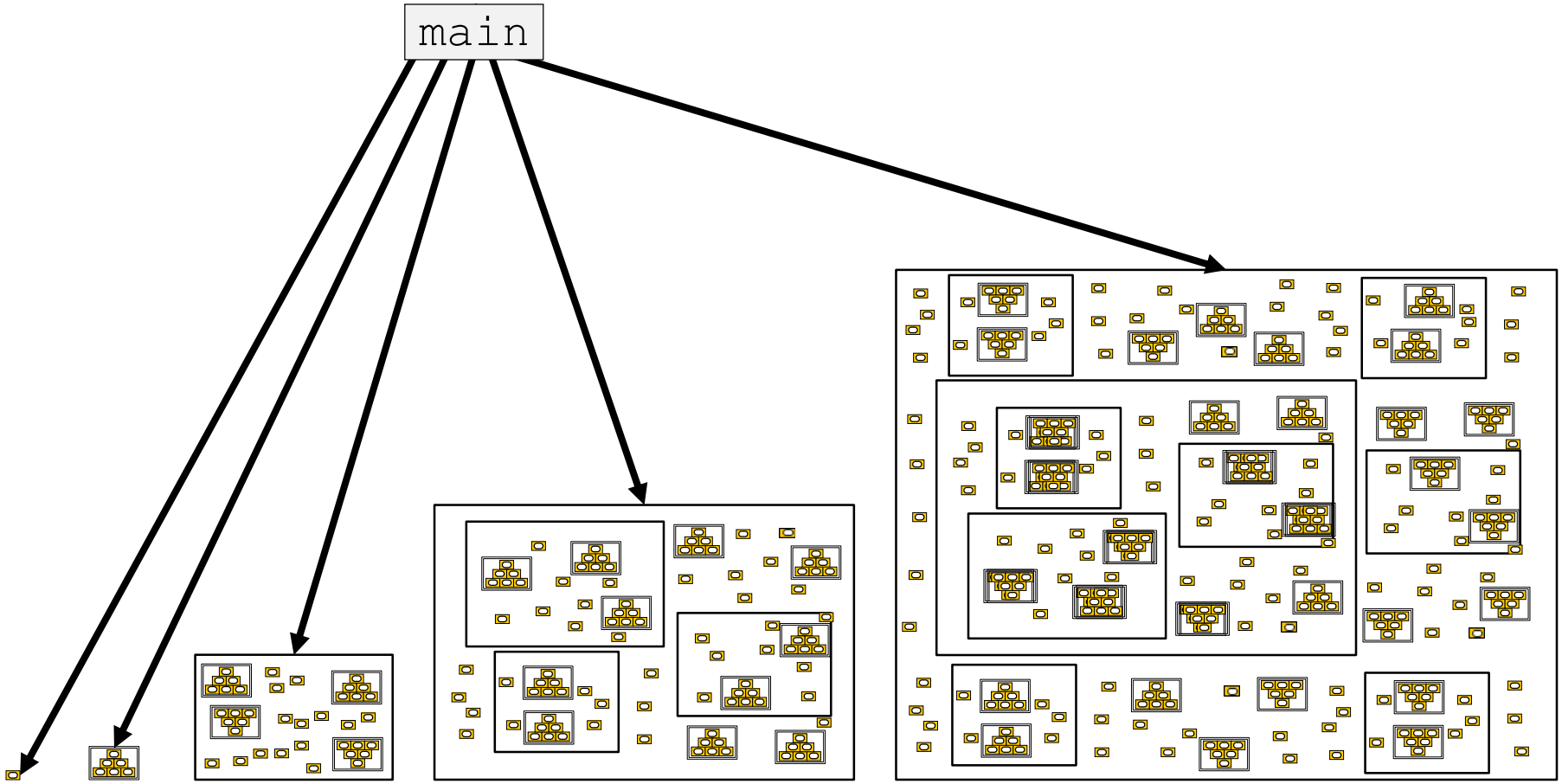
Aggregate dependencies:



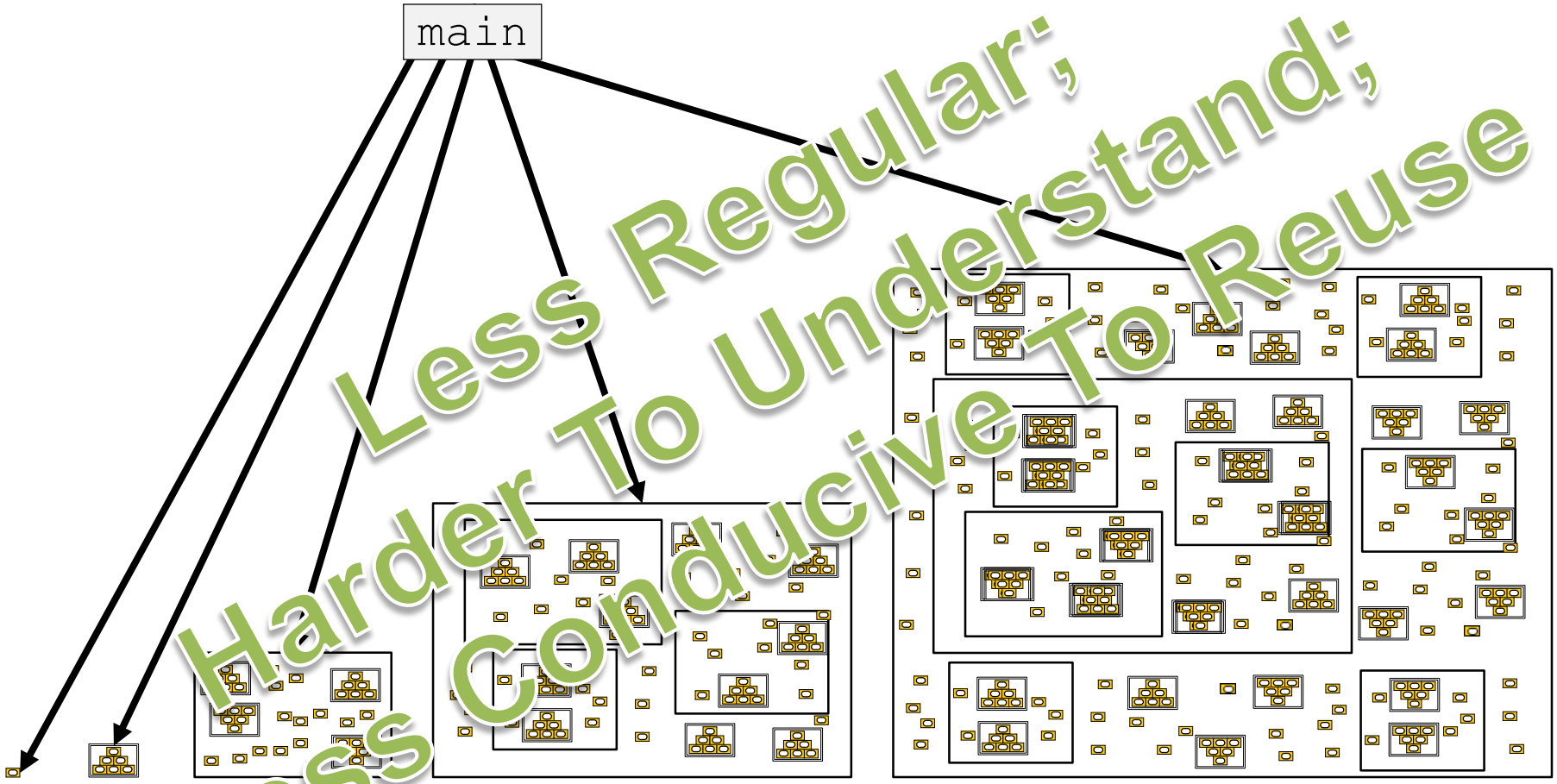
The Package Group



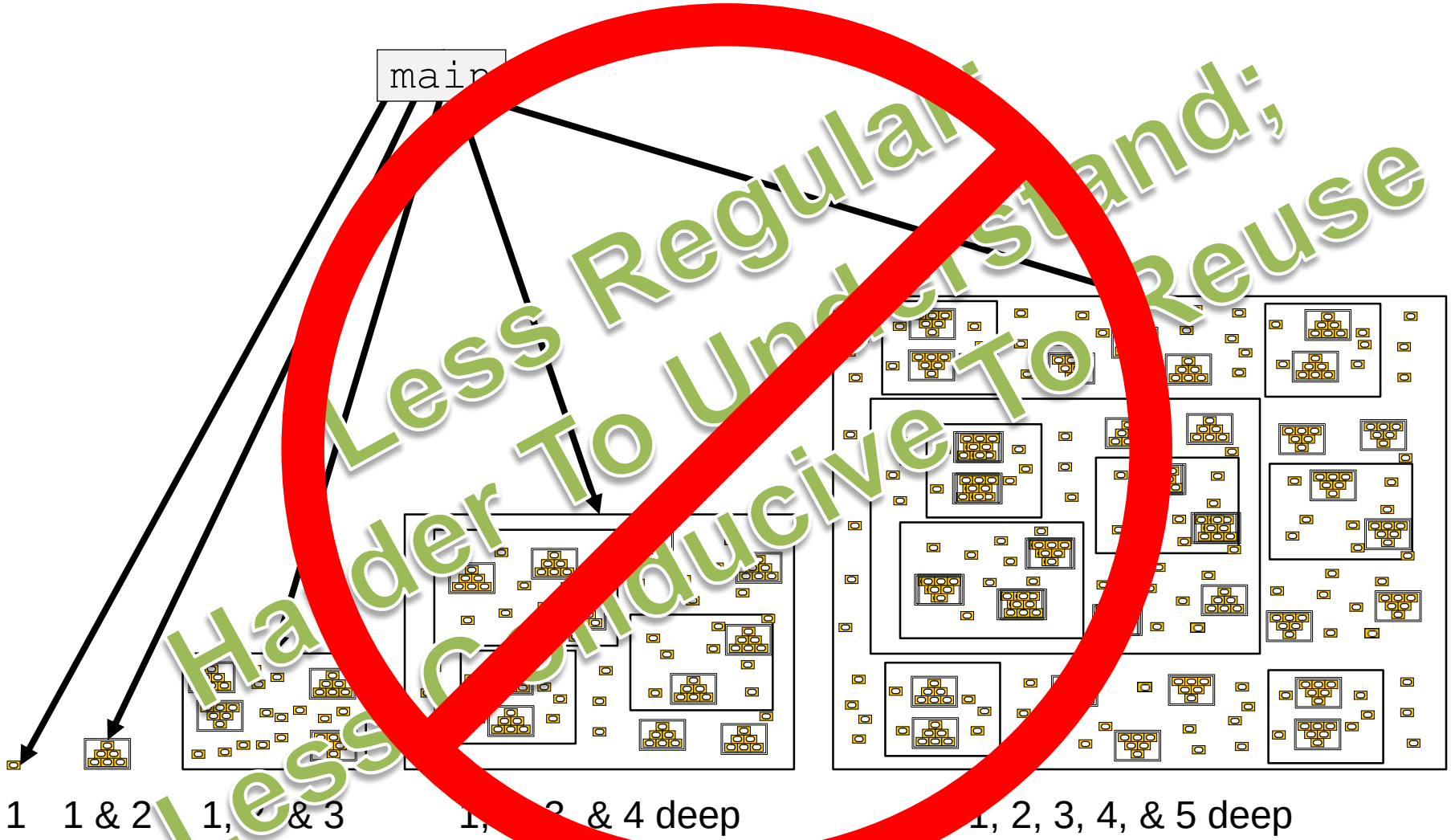
Non-Uniform Physical-Aggregation Depth



Non-Uniform Physical-Aggregation Depth



Non-Uniform Physical-Aggregation Depth



Uniform Depth of Physical Aggregation

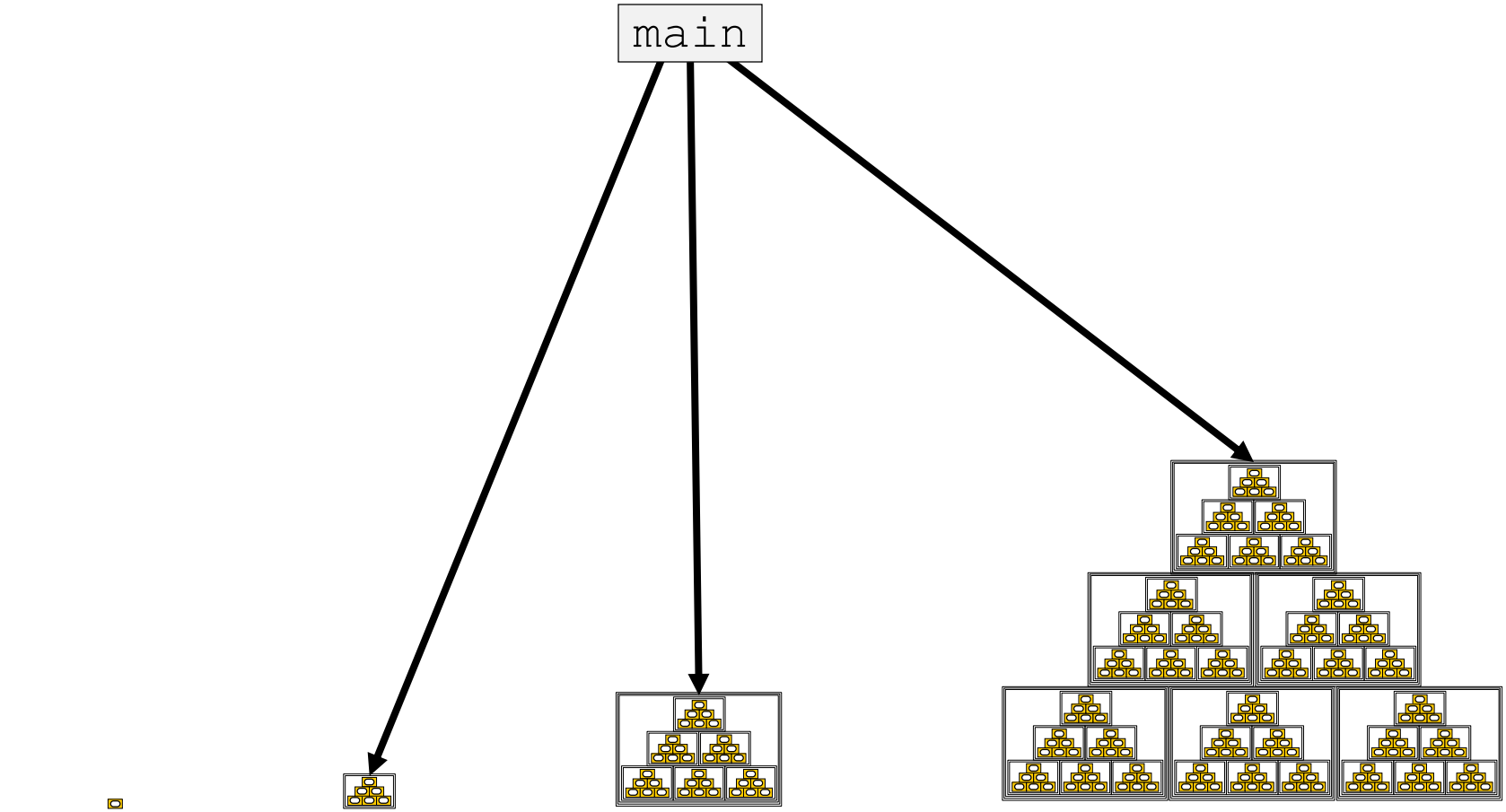
Component Package Package Group

Uniform Depth of Physical Aggregation

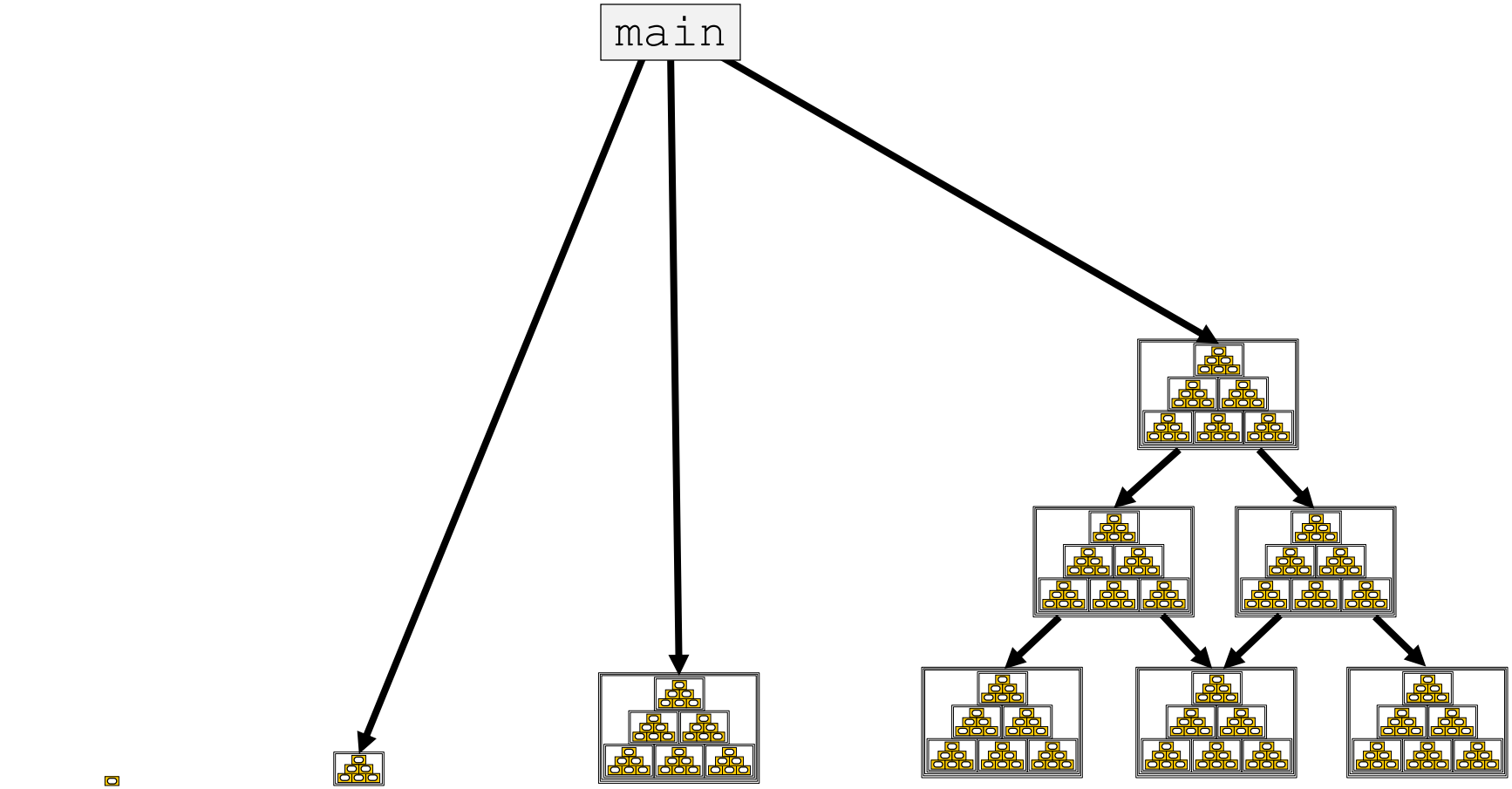
Exactly Three Levels of Physical Aggregation

Component Package Package Group

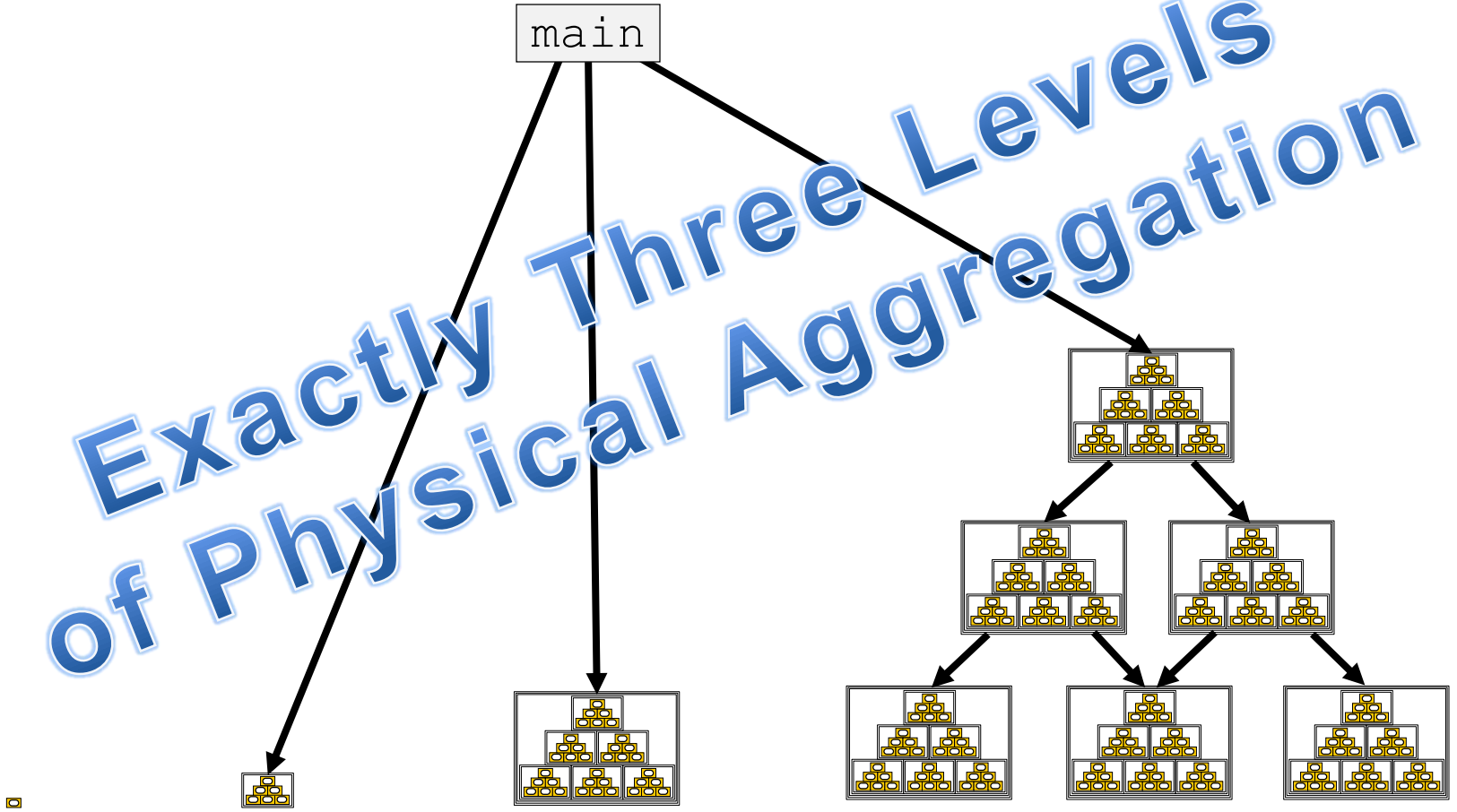
Uniform Depth of Physical Aggregation



Uniform Depth of Physical Aggregation

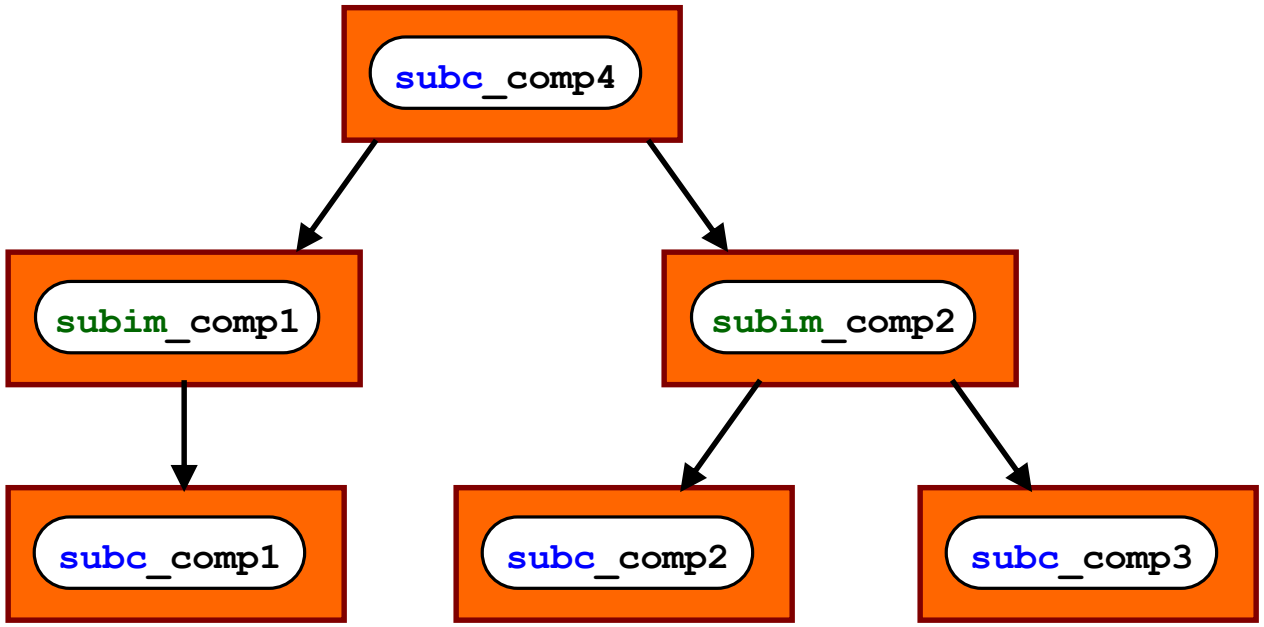


Uniform Depth of Physical Aggregation



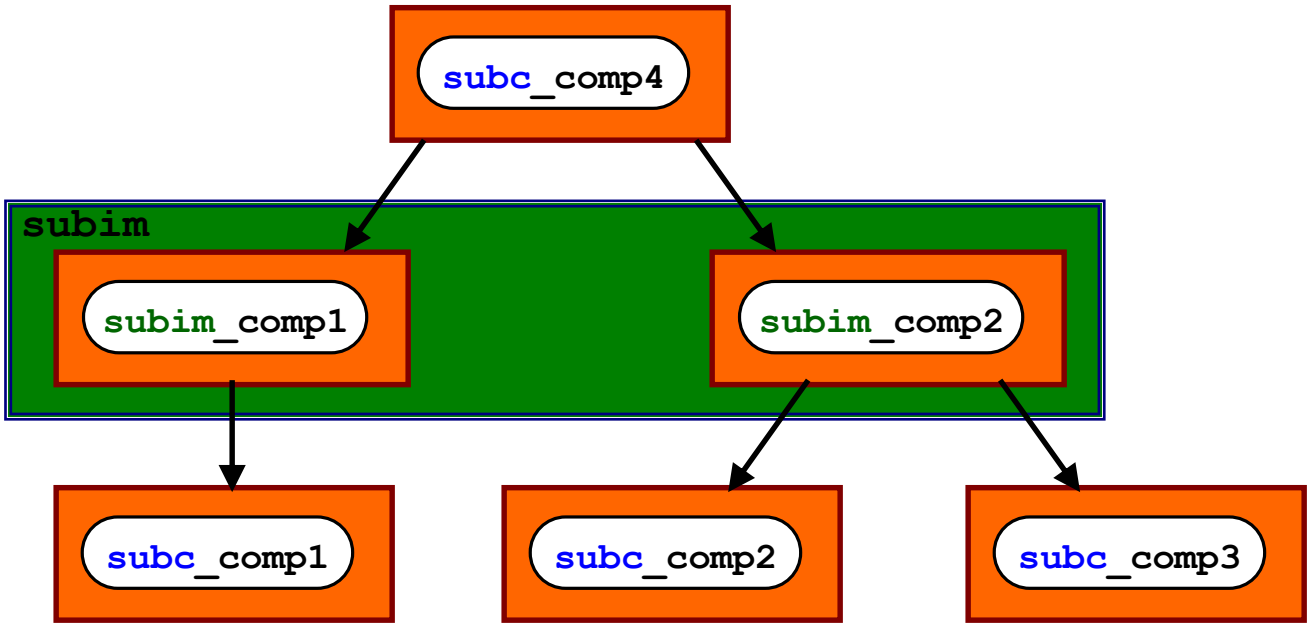
Escalating Encapsulation

Package naming is more than just a convention:



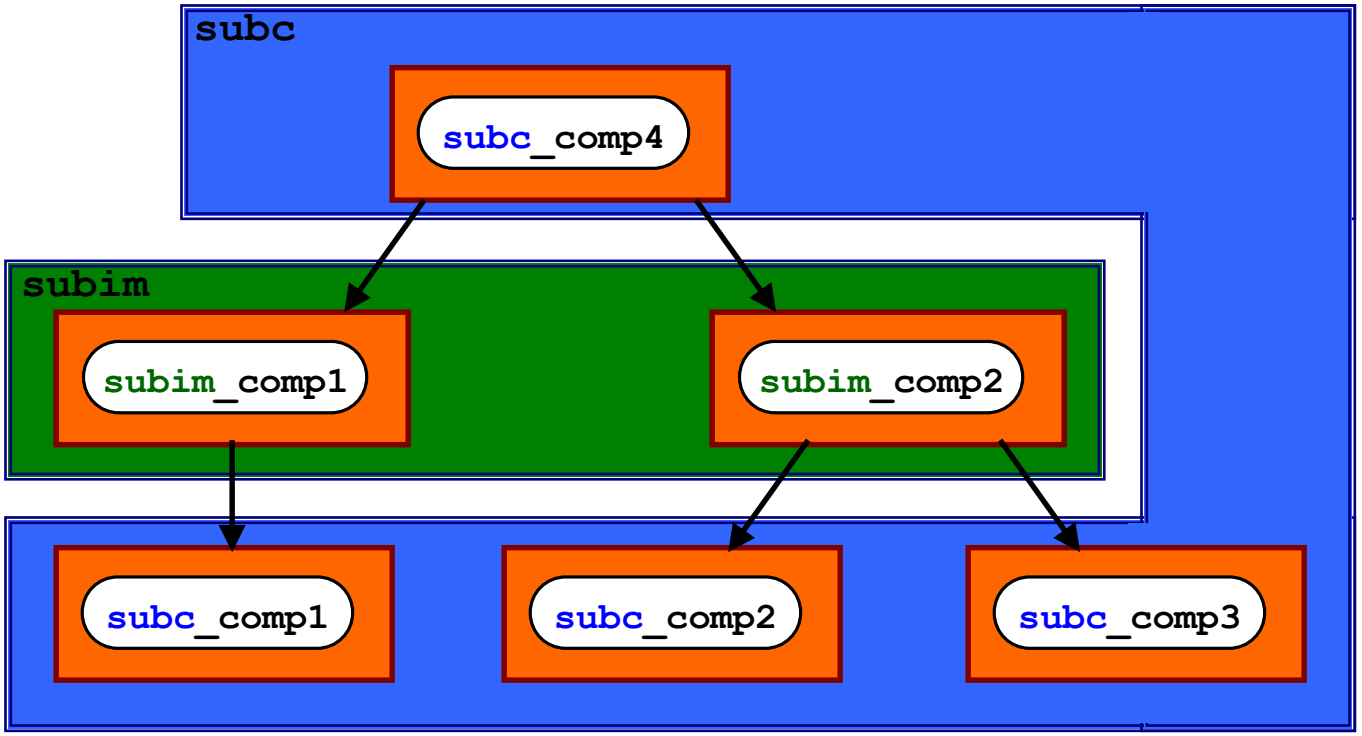
Escalating Encapsulation

Package naming is more than just a convention:



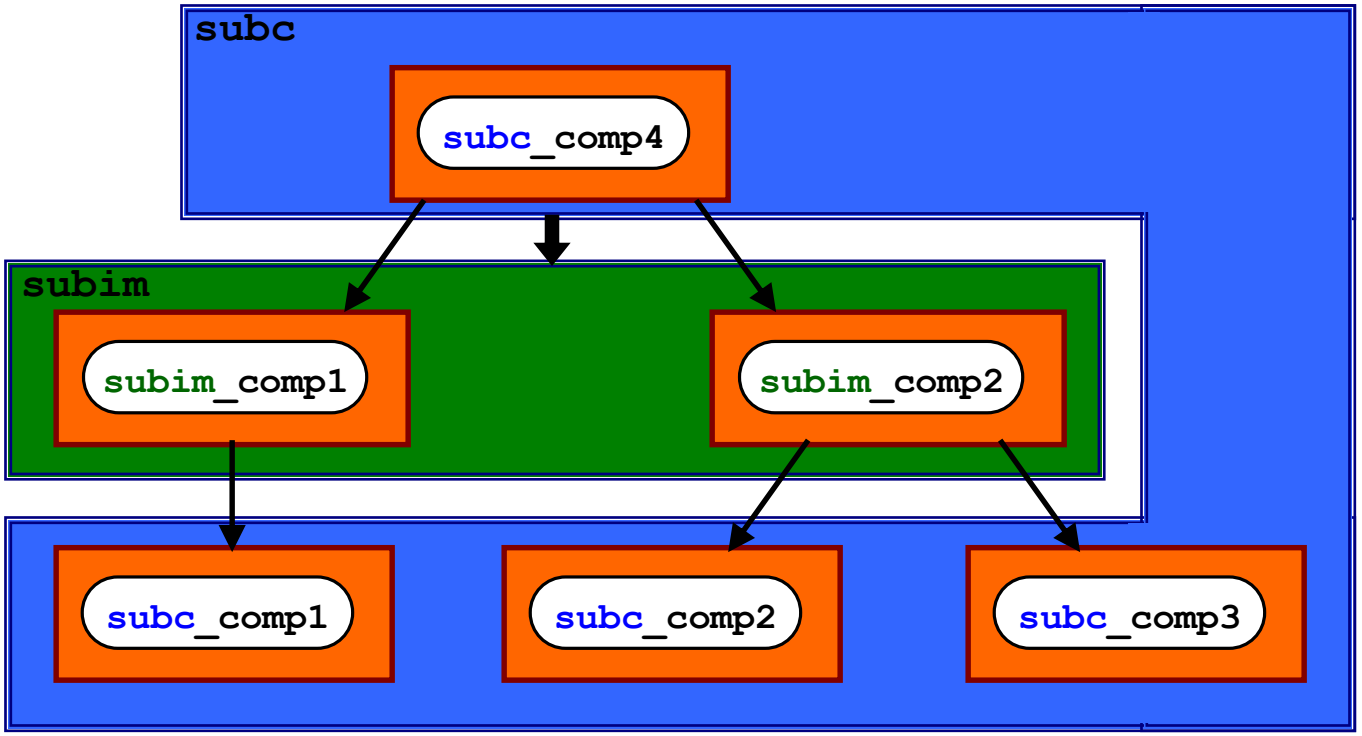
Escalating Encapsulation

Package naming is more than just a convention:



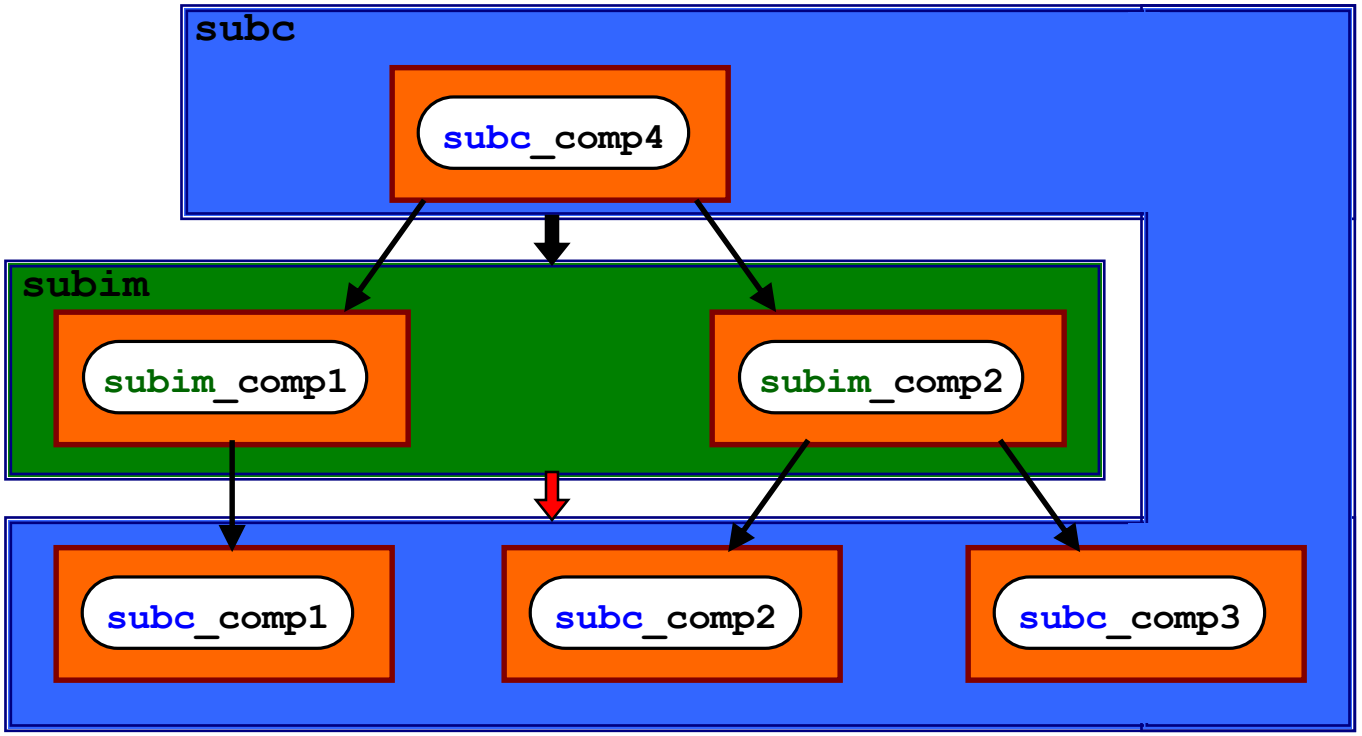
Escalating Encapsulation

Package naming is more than just a convention:



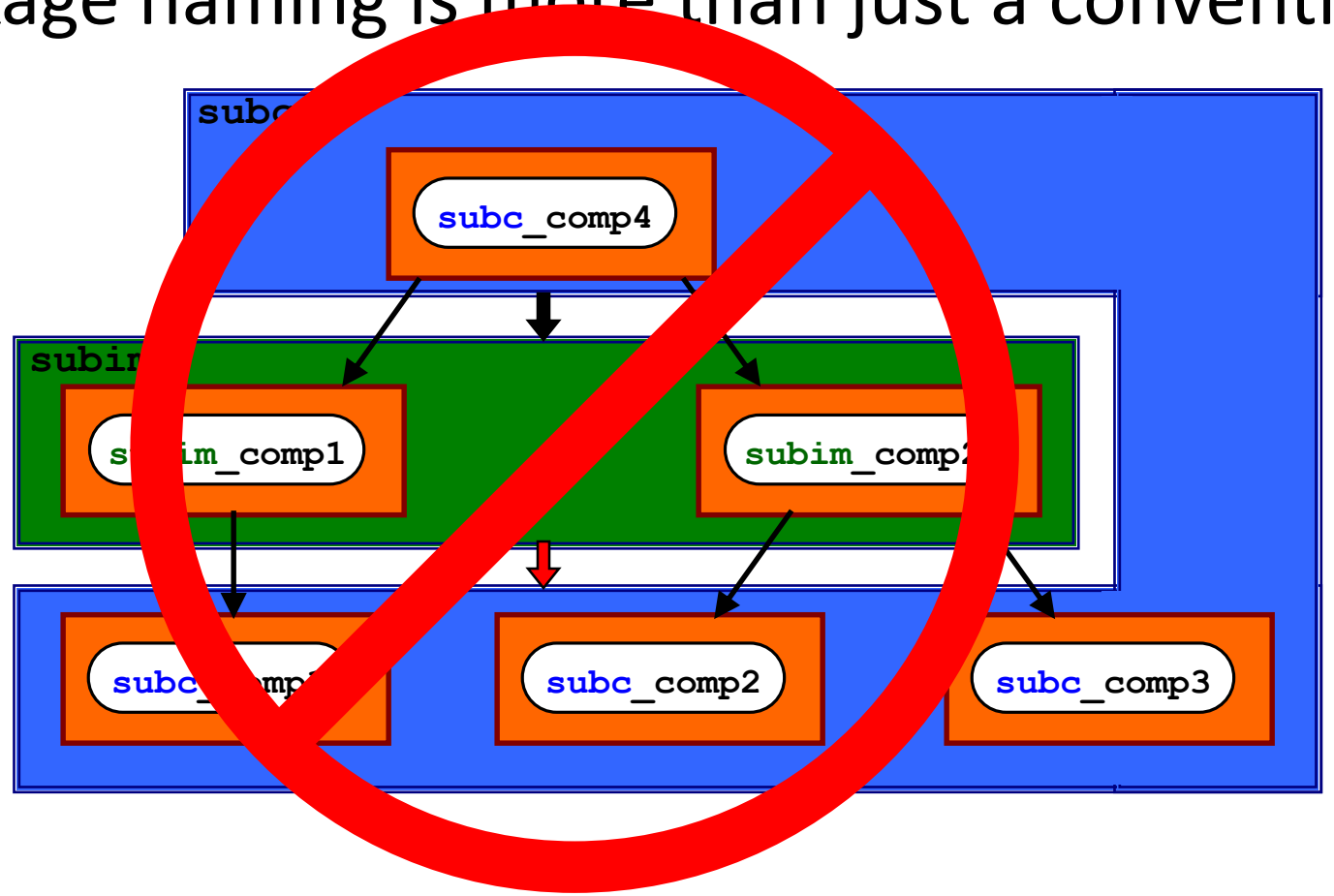
Escalating Encapsulation

Package naming is more than just a convention:



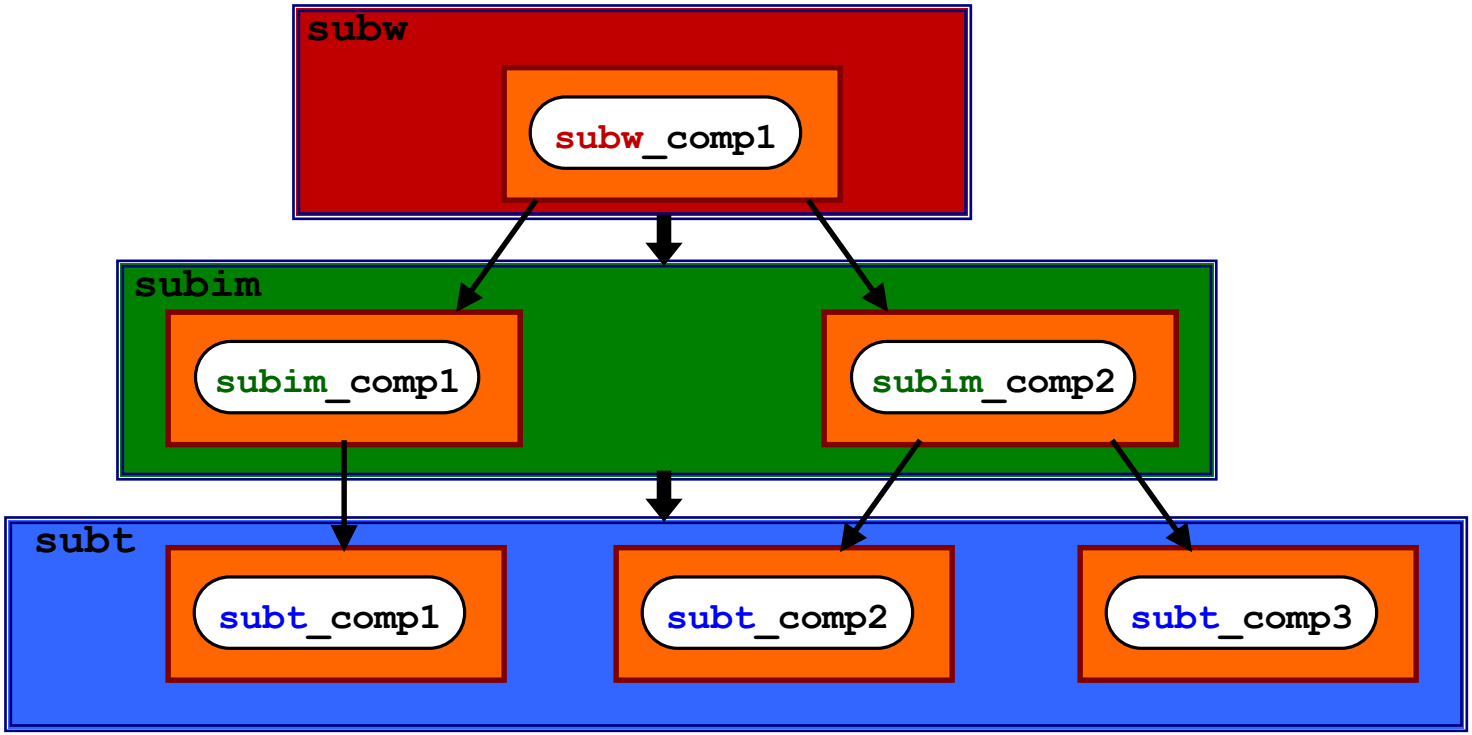
Escalating Encapsulation

Package naming is more than just a convention:

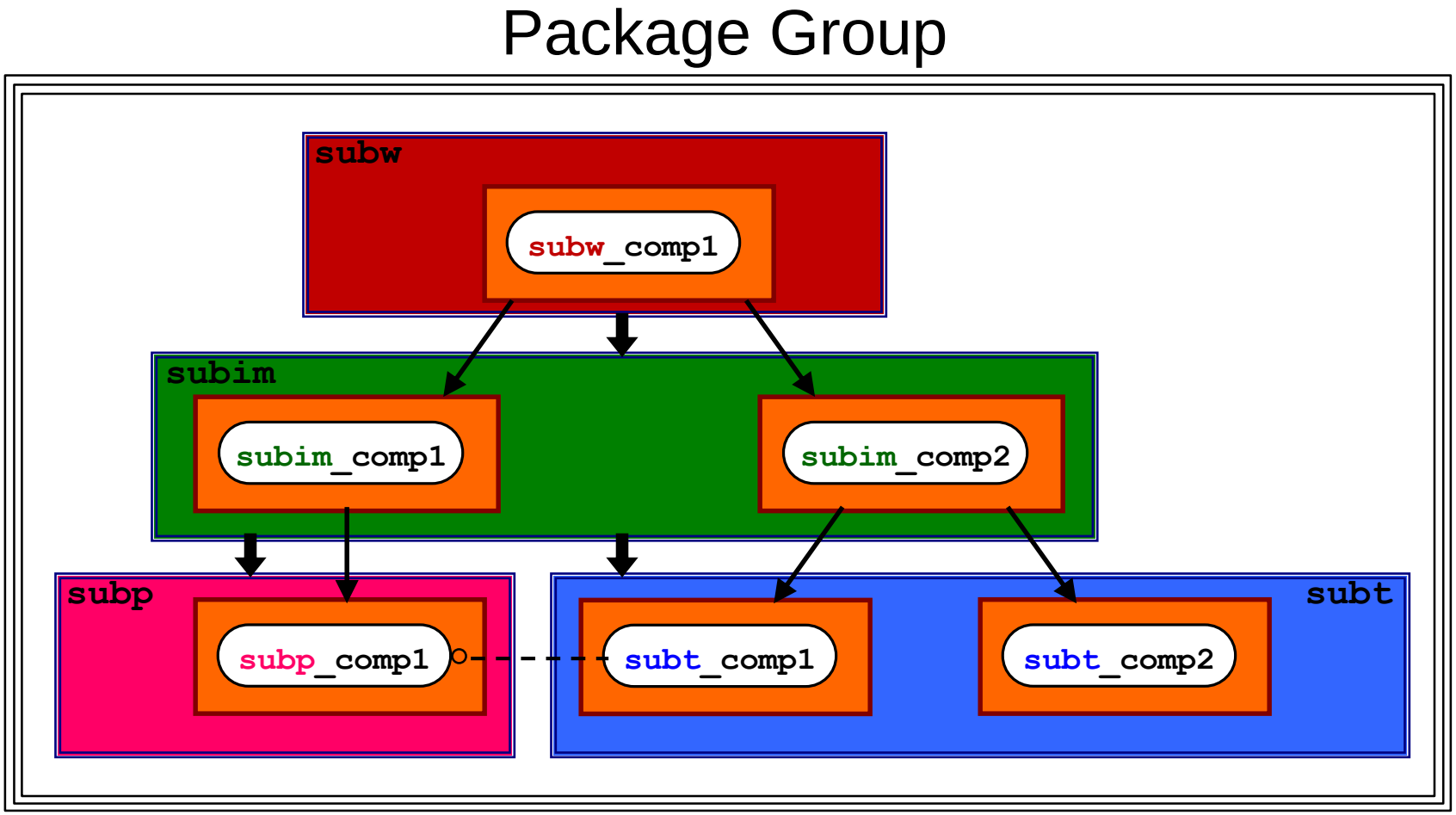


Escalating Encapsulation

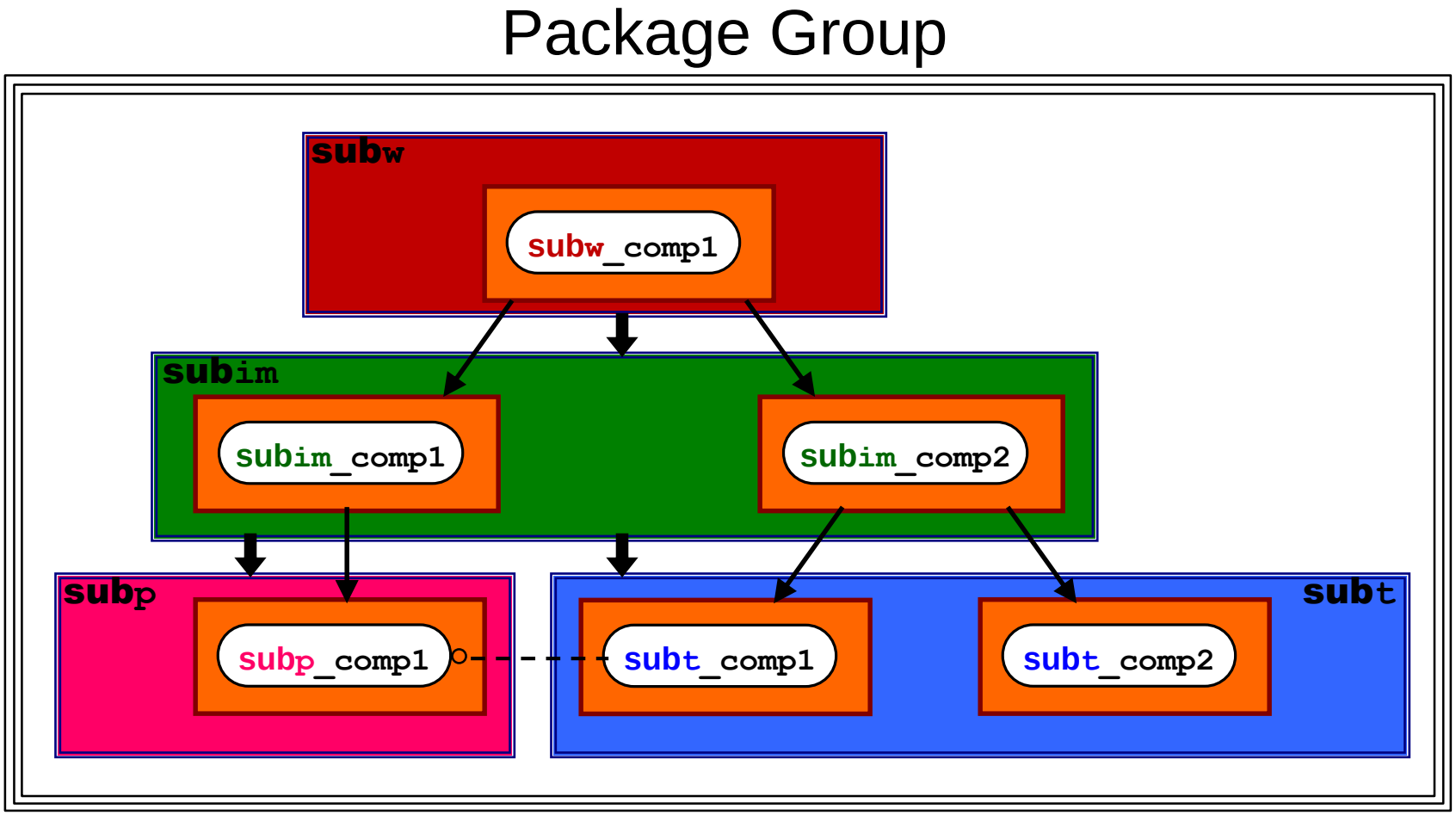
Package naming is more than just a convention:



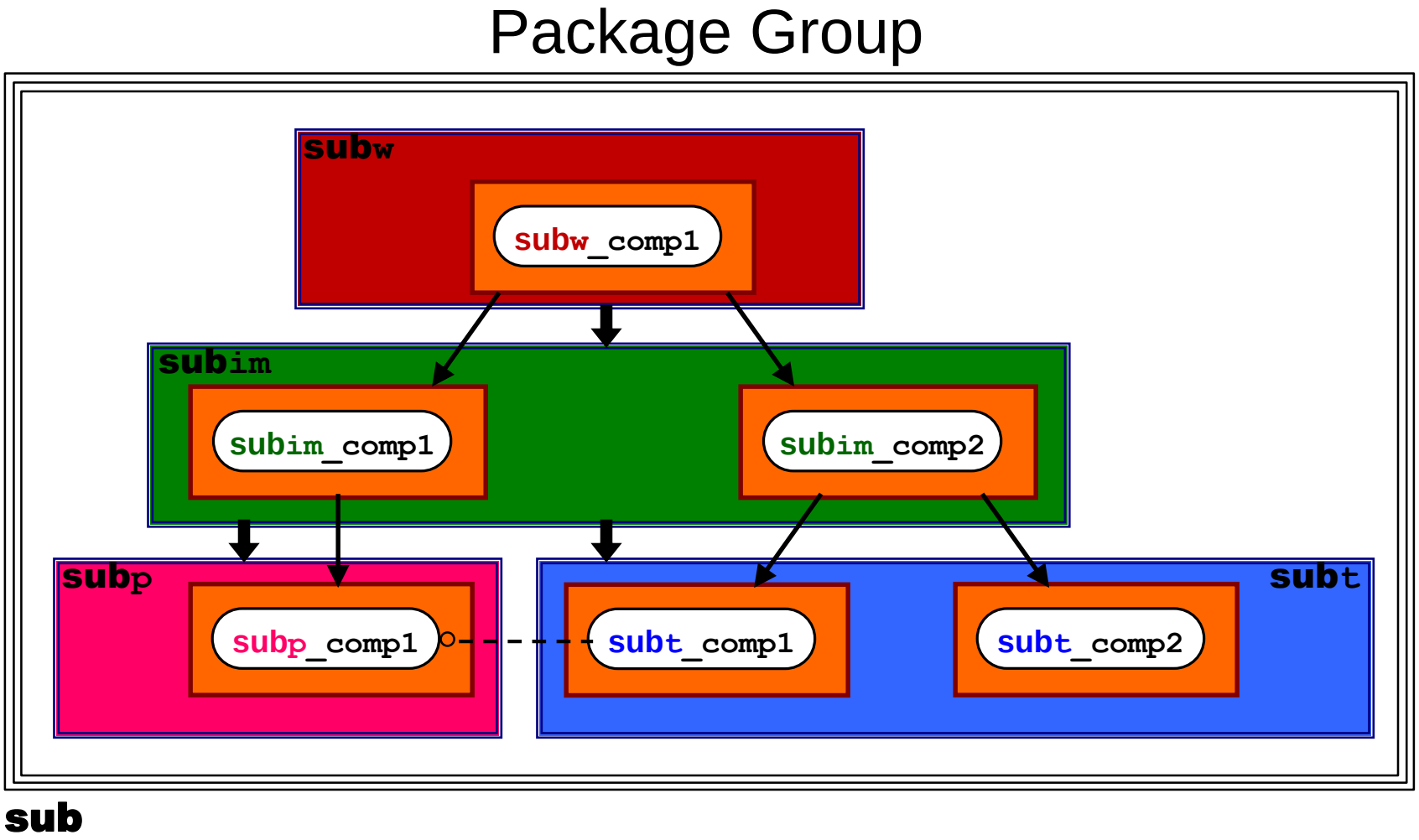
Escalating Encapsulation



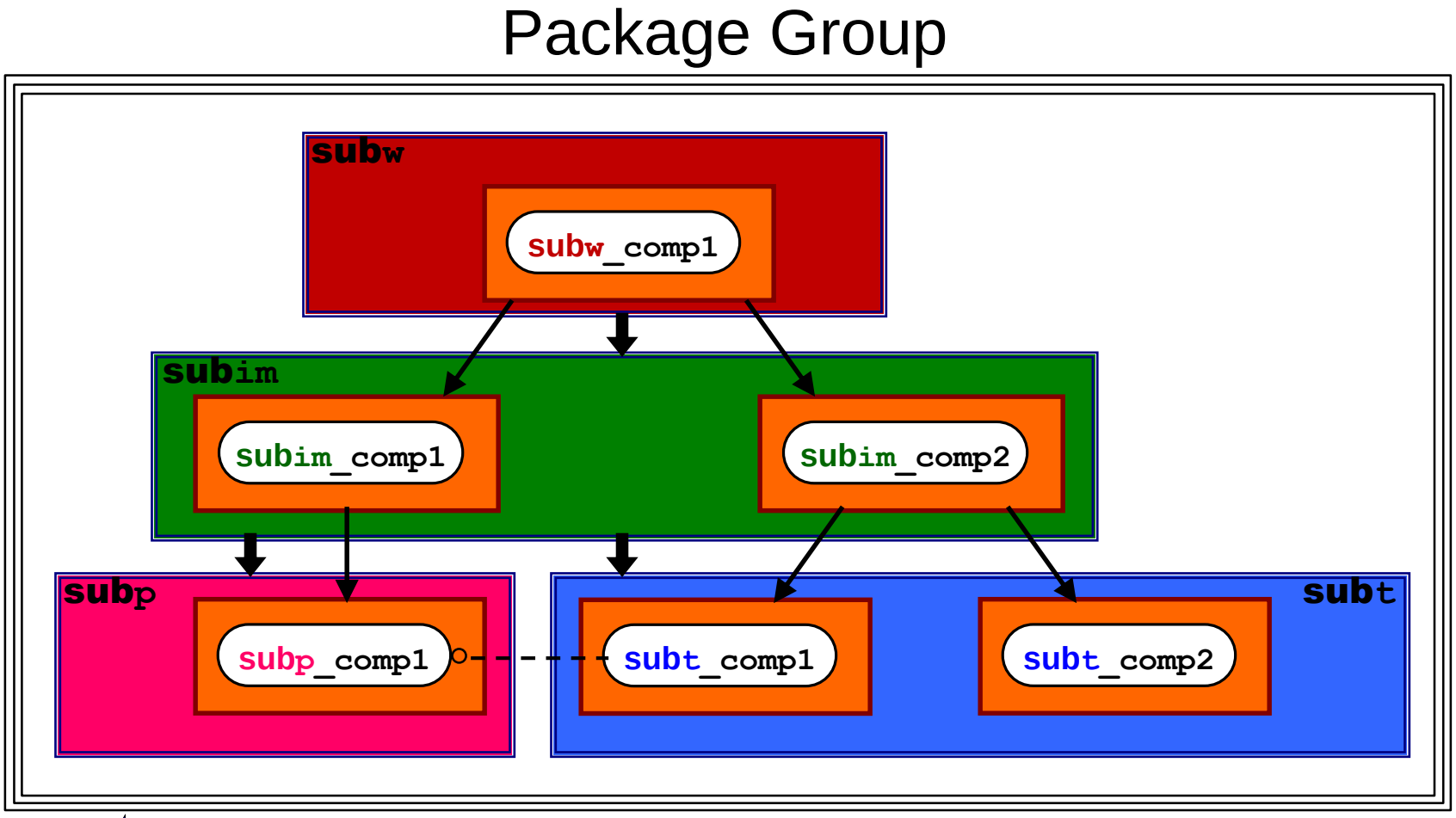
Escalating Encapsulation



Escalating Encapsulation



Escalating Encapsulation



Introduction

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

Introduction

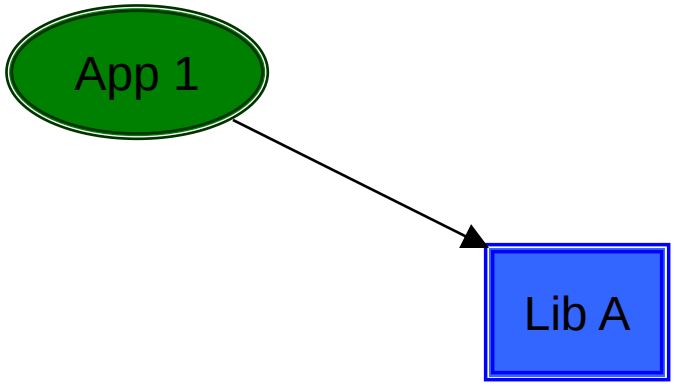
All of the software we write is governed by a common overarching set of

Organizing Principles.

Among the most central of which is achieving

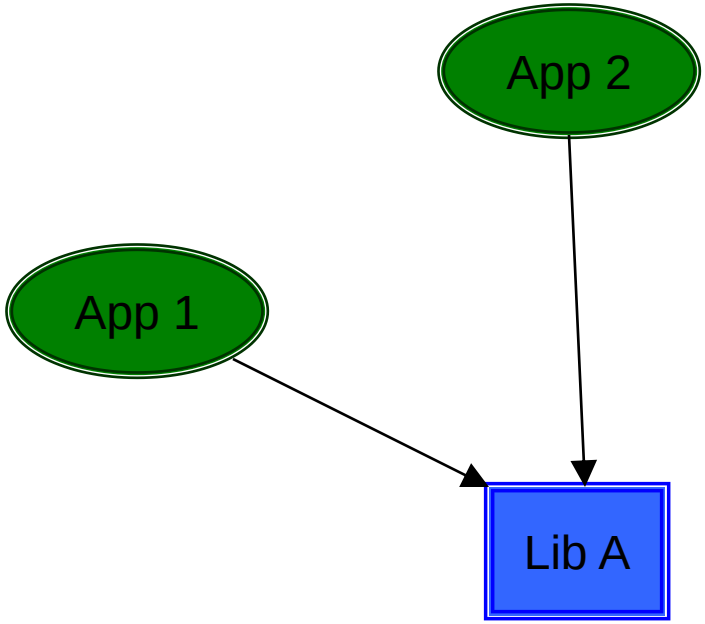
Sound Physical Design.

Creating a Big Ball of Mud



Creating a Big Ball of Mud

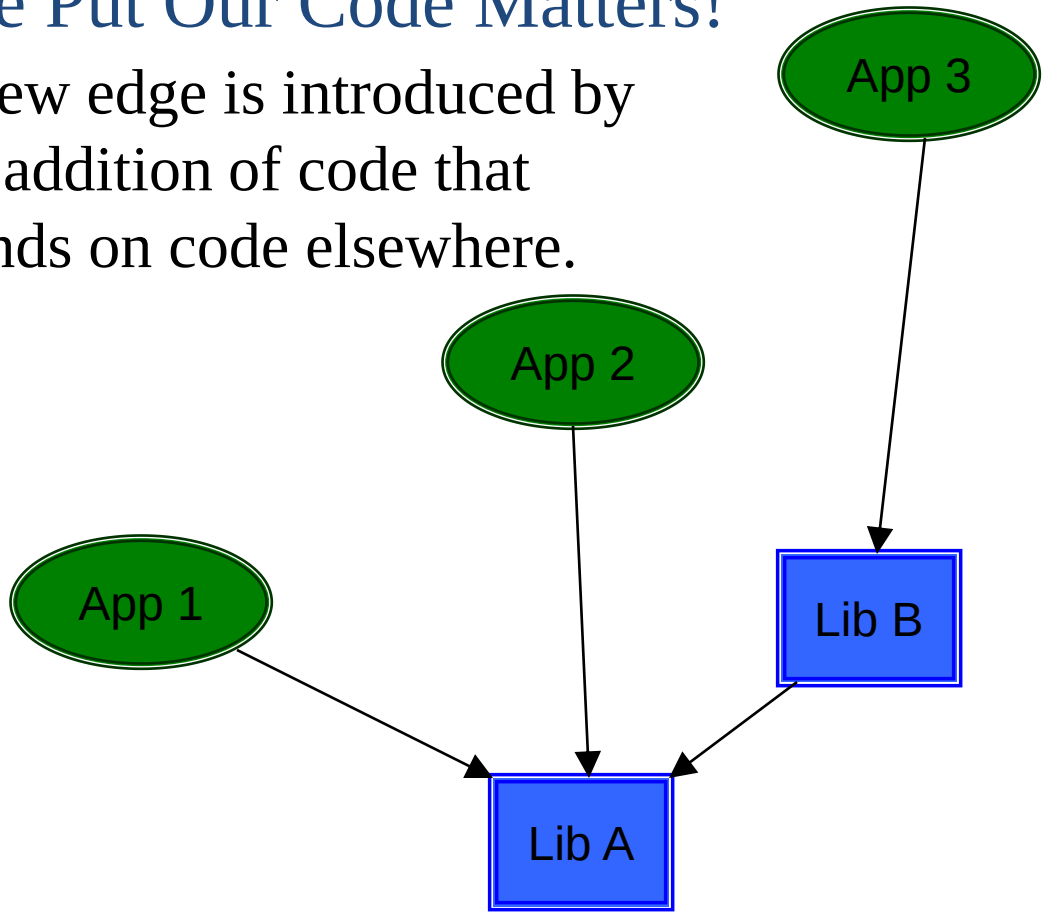
Where We Put Our Code Matters!



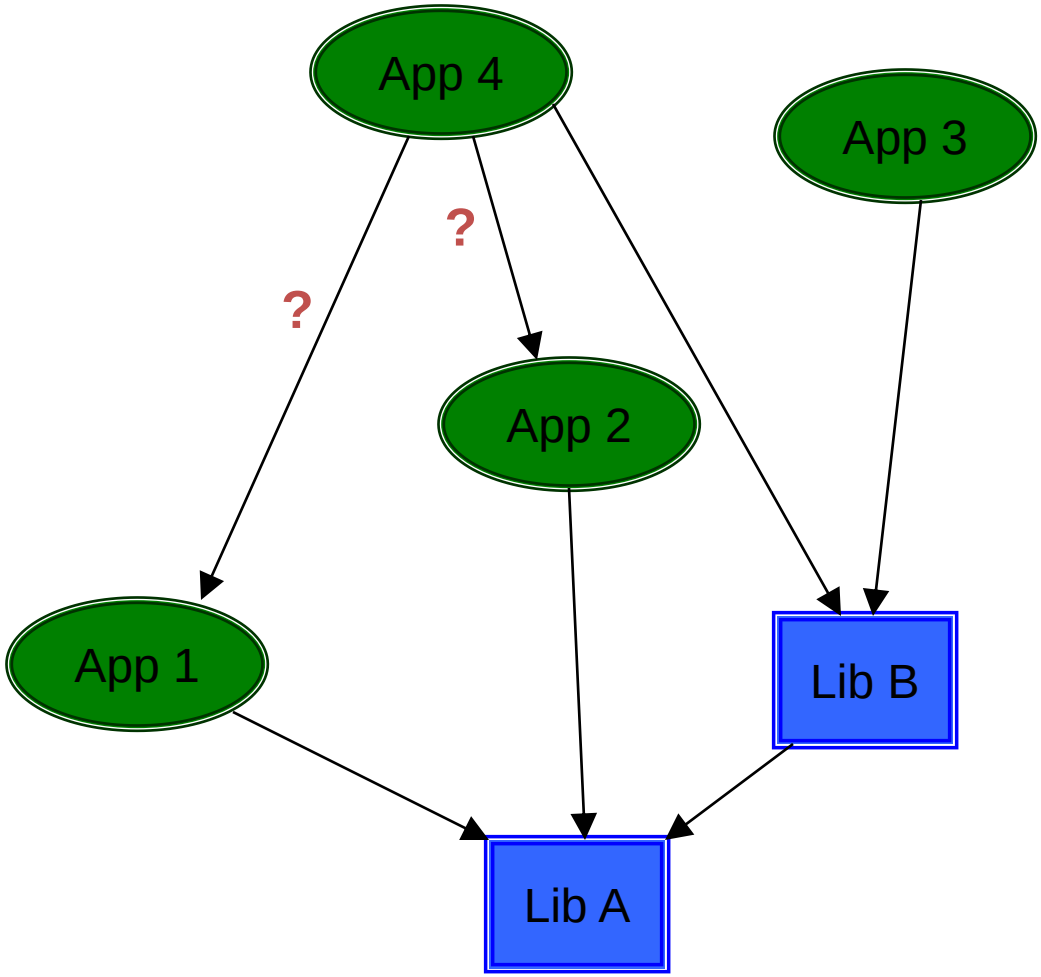
Creating a Big Ball of Mud

Where We Put Our Code Matters!

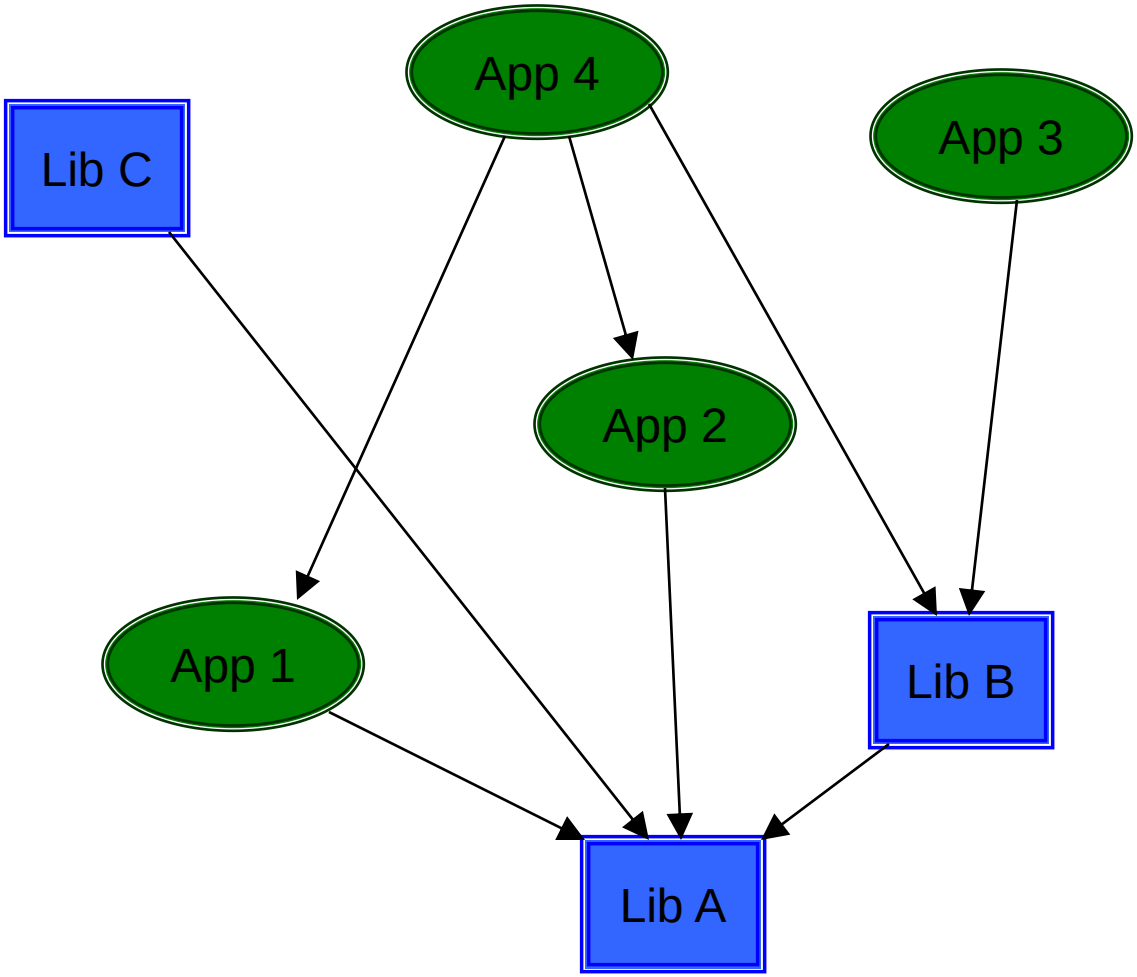
Each new edge is introduced by the addition of code that depends on code elsewhere.



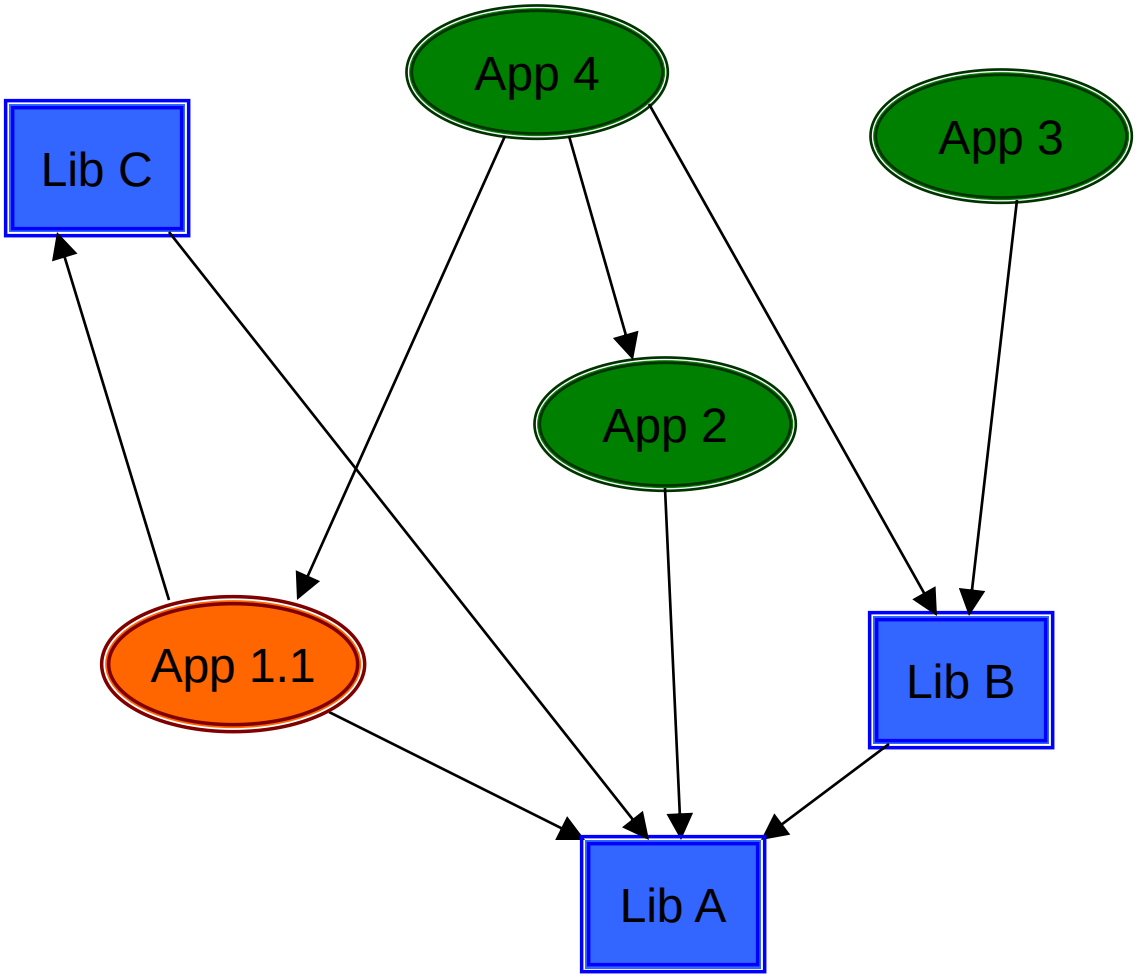
Creating a Big Ball of Mud



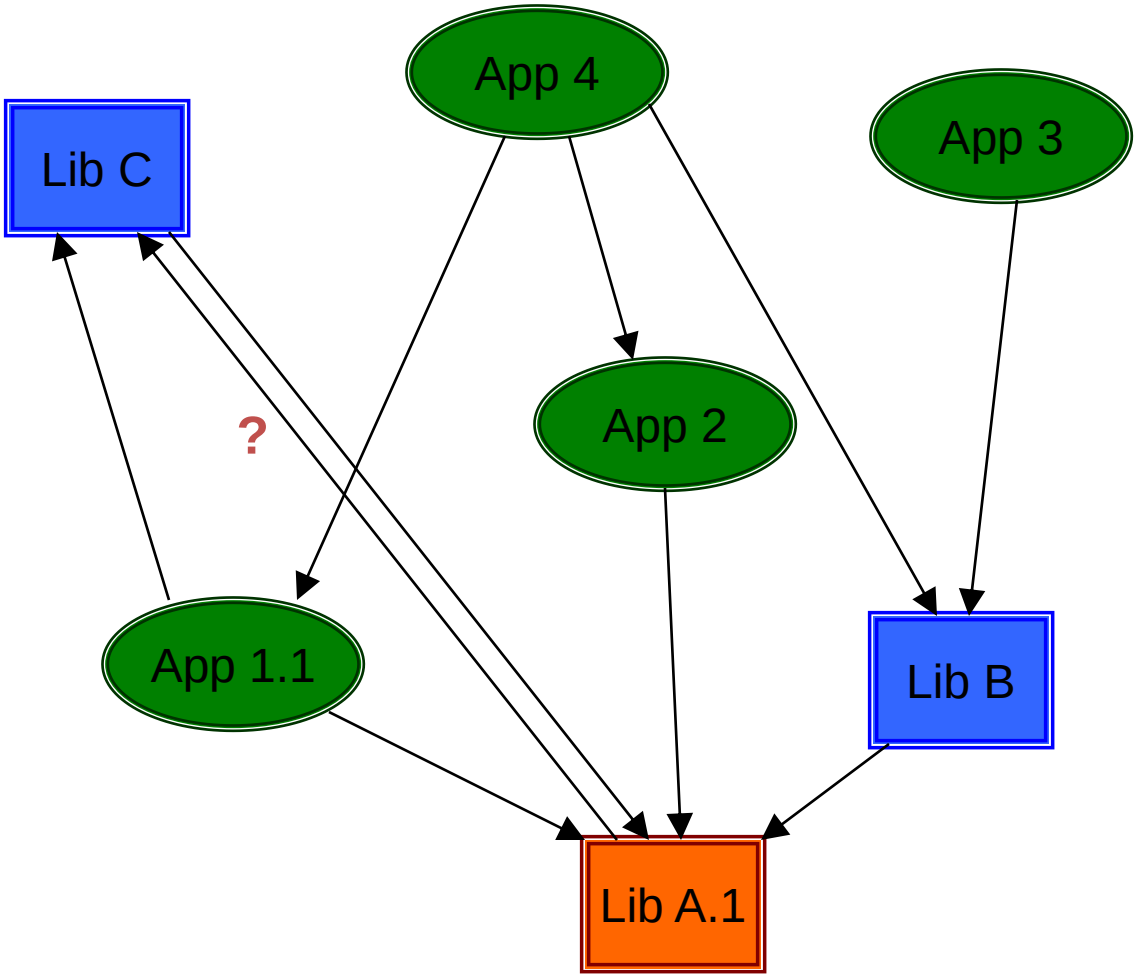
Creating a Big Ball of Mud



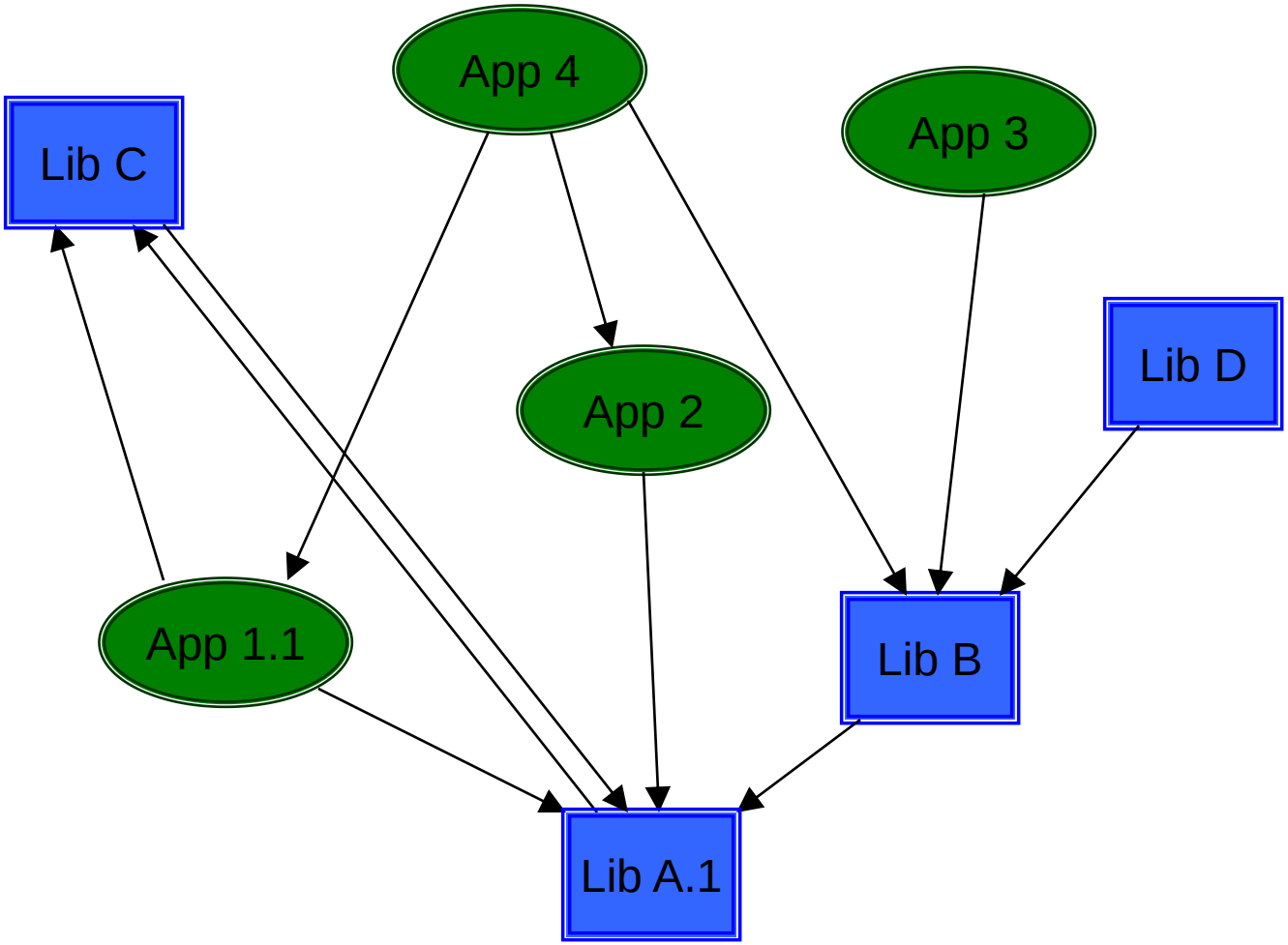
Creating a Big Ball of Mud



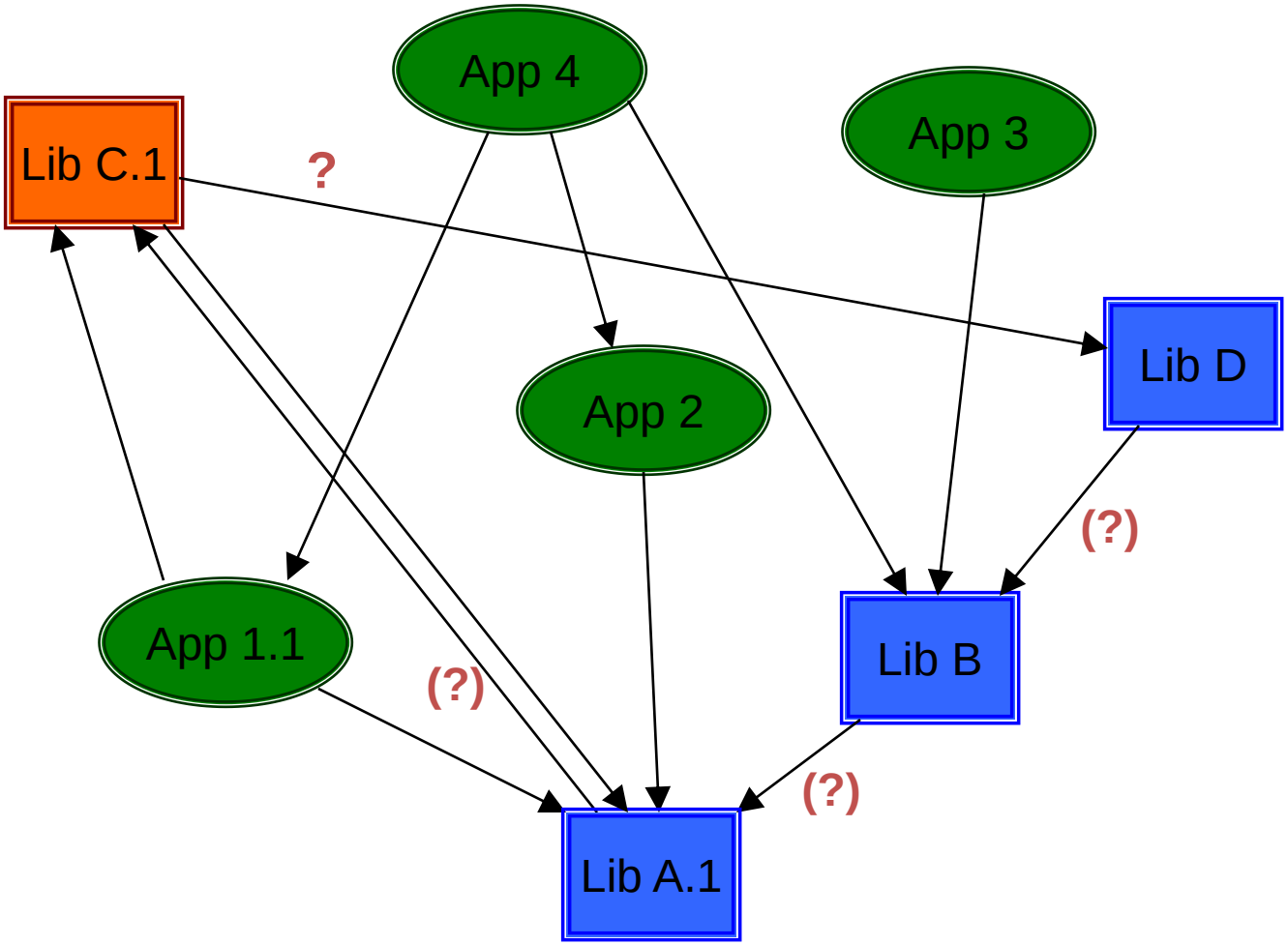
Creating a Big Ball of Mud



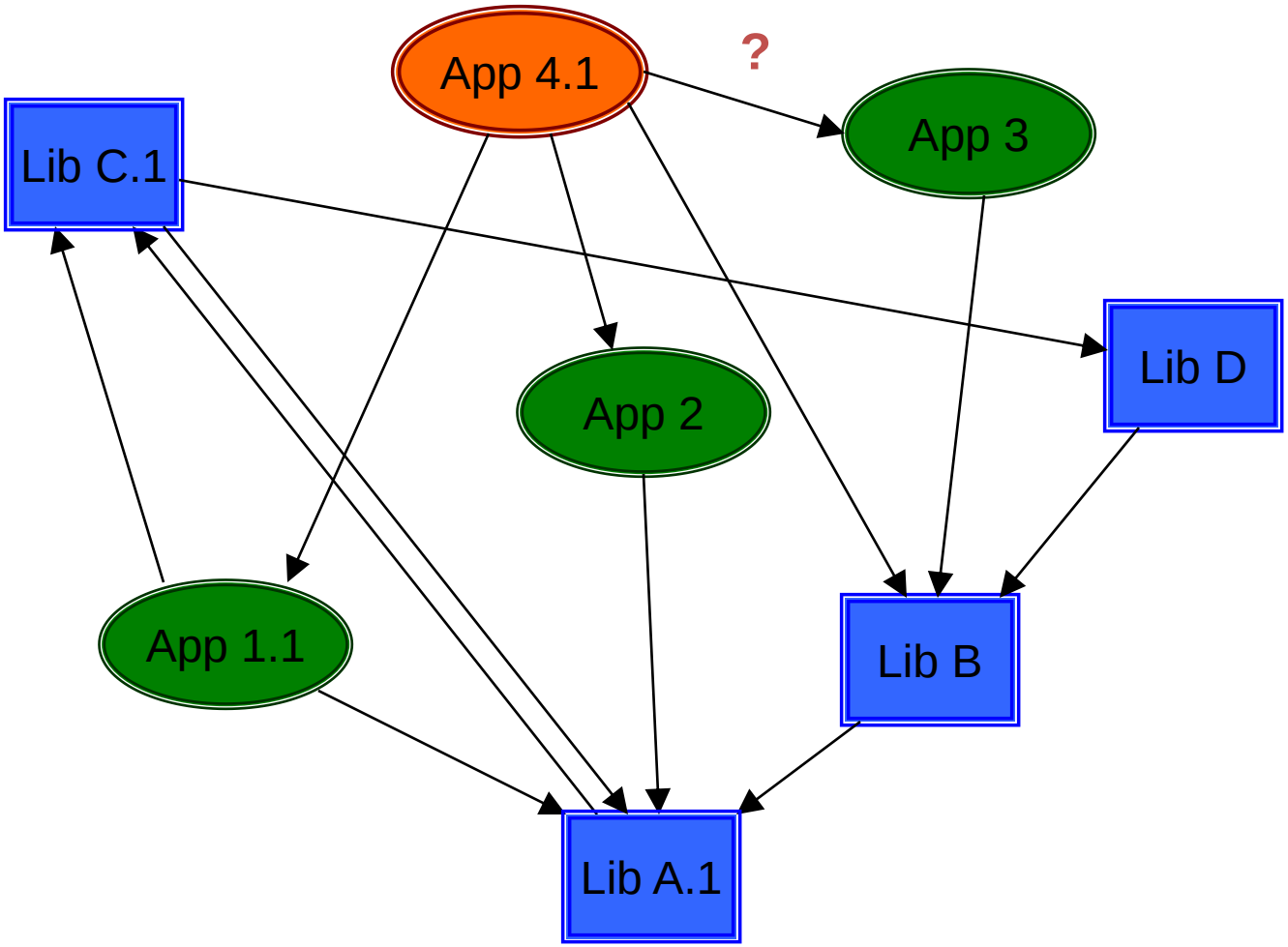
Creating a Big Ball of Mud



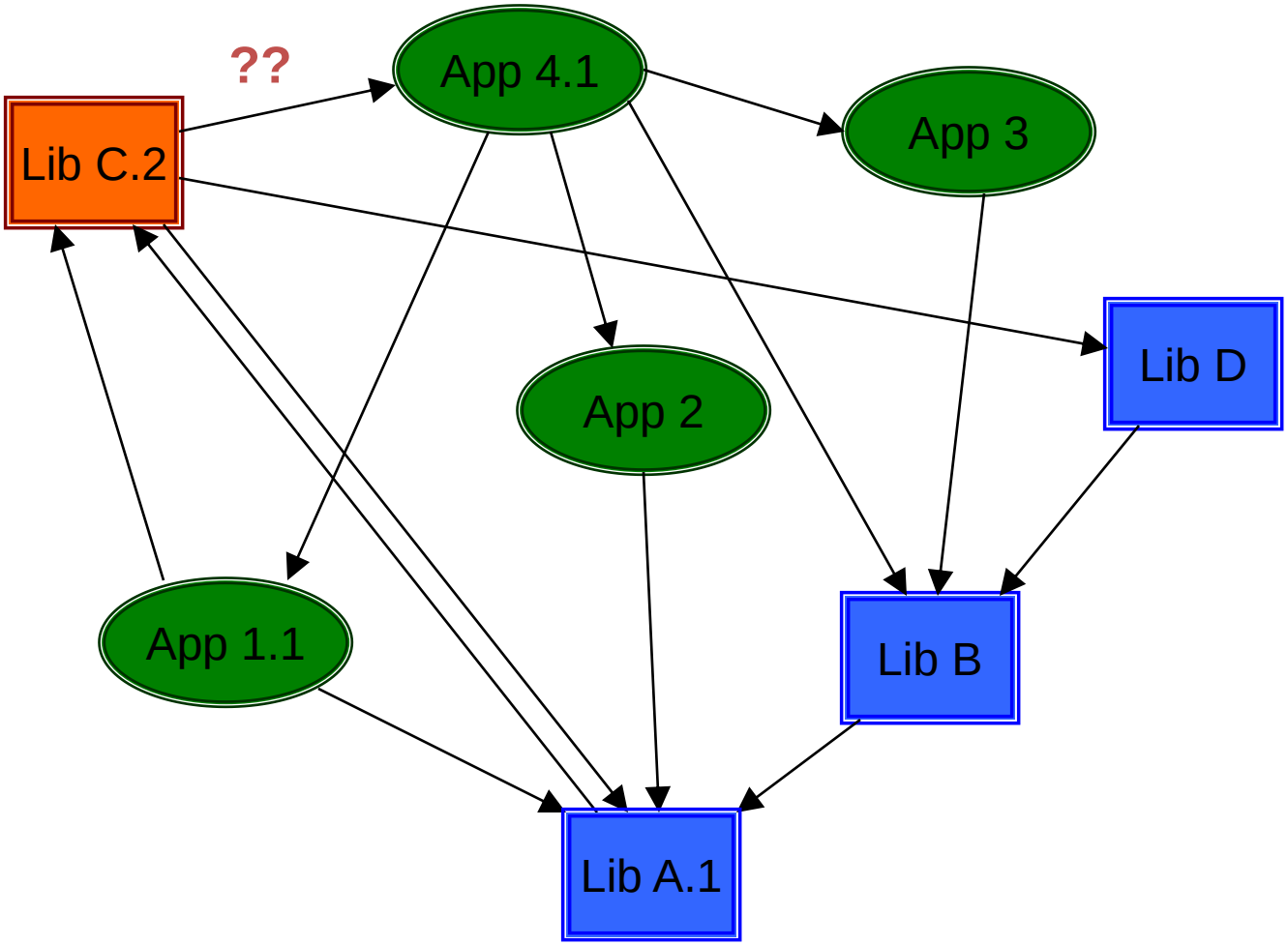
Creating a Big Ball of Mud



Creating a Big Ball of Mud

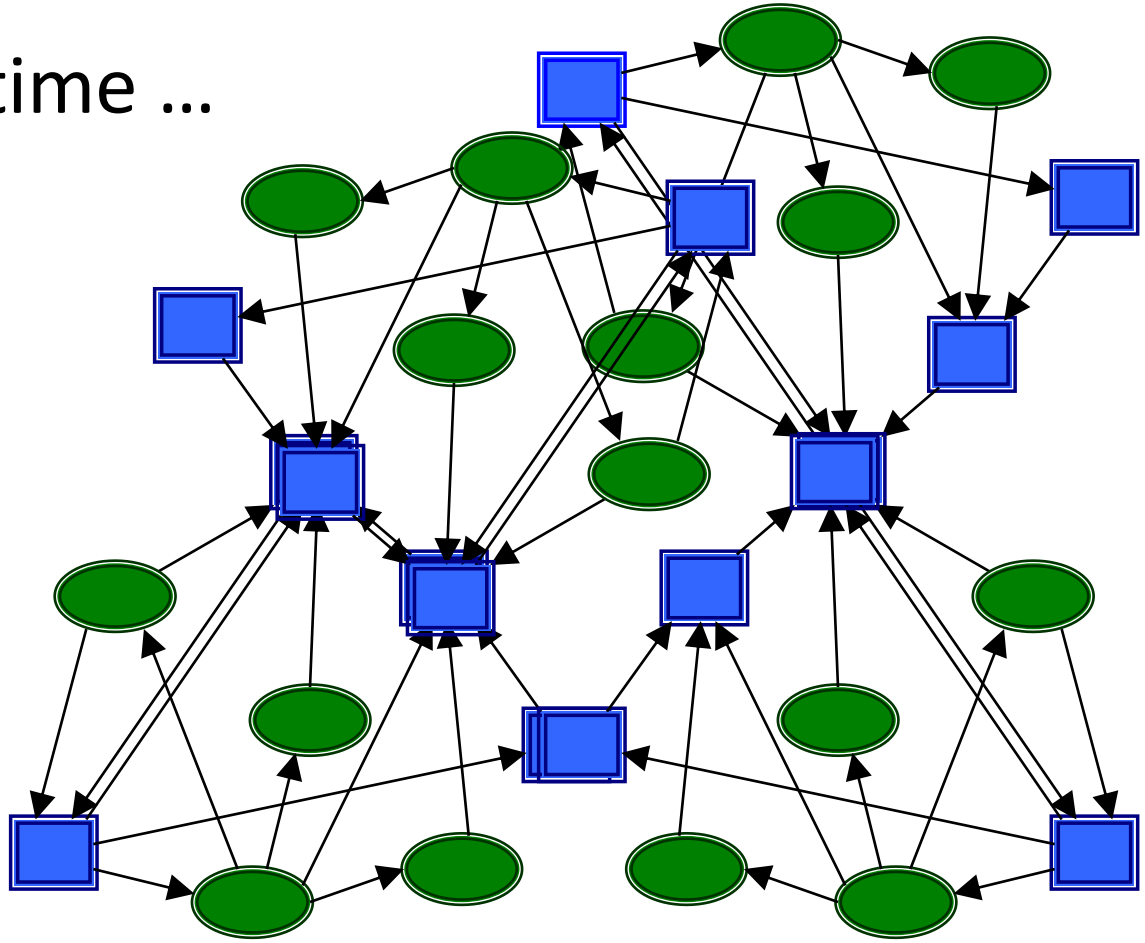


Creating a Big Ball of Mud



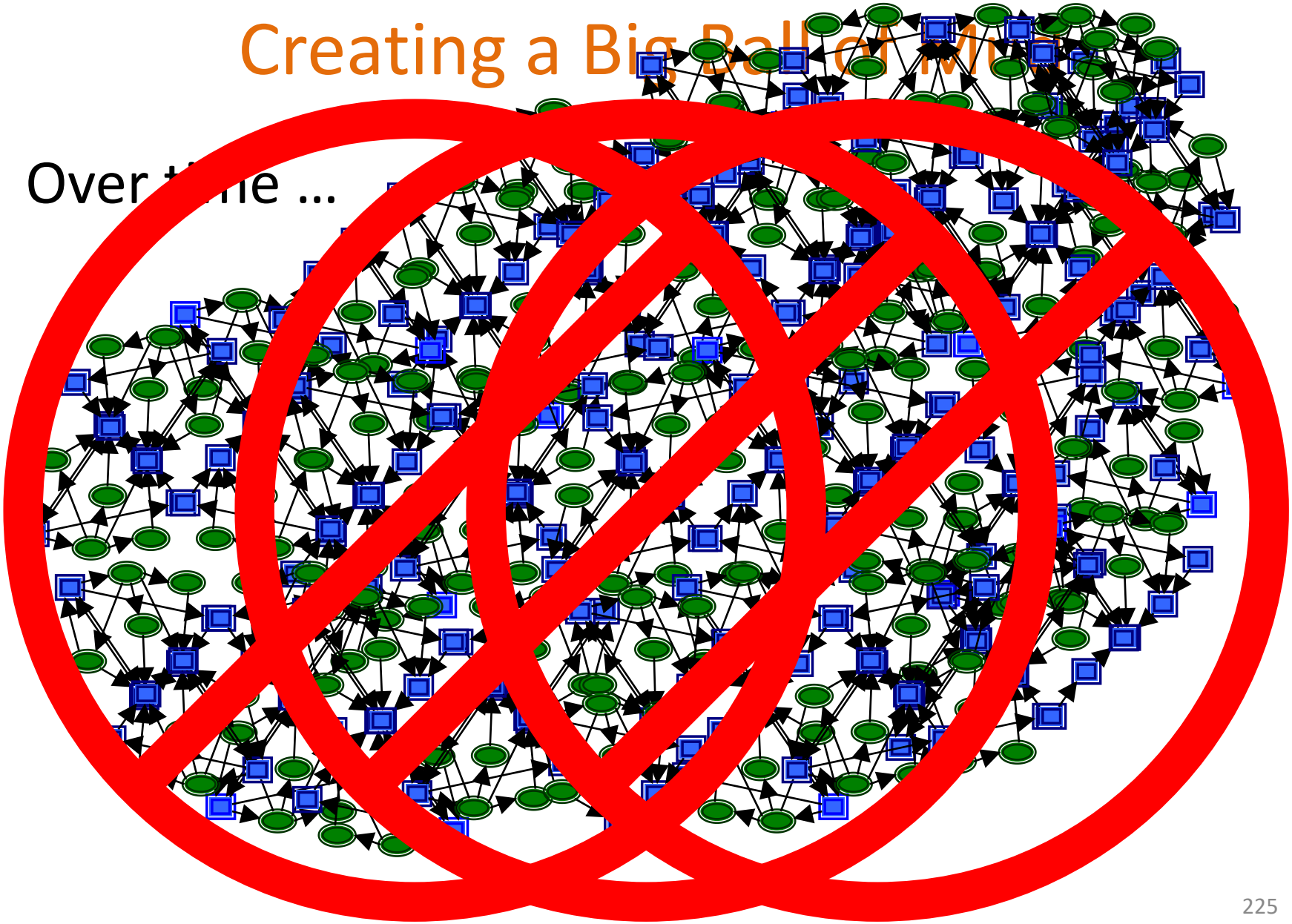
Creating a Big Ball of Mud

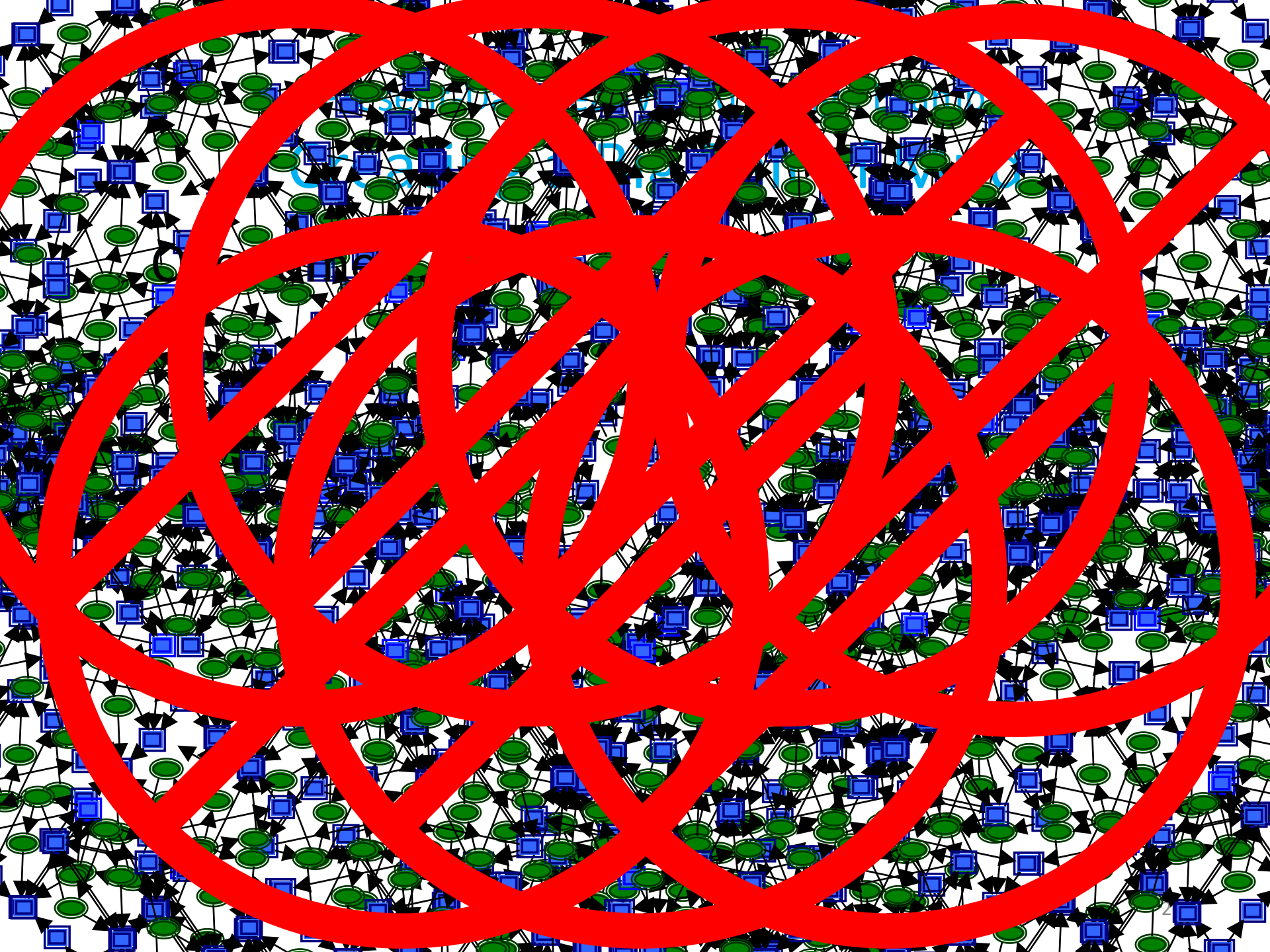
Over time …



Creating a Big Ball of Mud

Over time …





3. Present-Day, Real-World Design Examples

Large-Scale Physical Design

Large-Scale Physical Design

- Good physical design is an **engineering discipline**, **not** an afterthought.

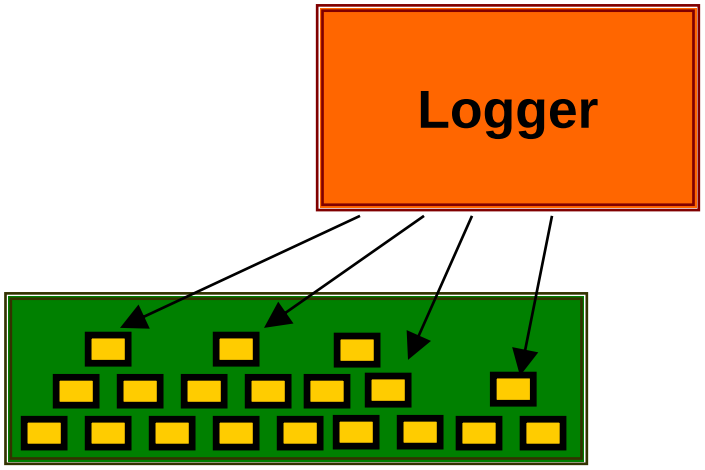
Large-Scale Physical Design

- Good physical design is an **engineering discipline**, **not** an afterthought.
- Good physical design must be introduced from the **inception** of an application.

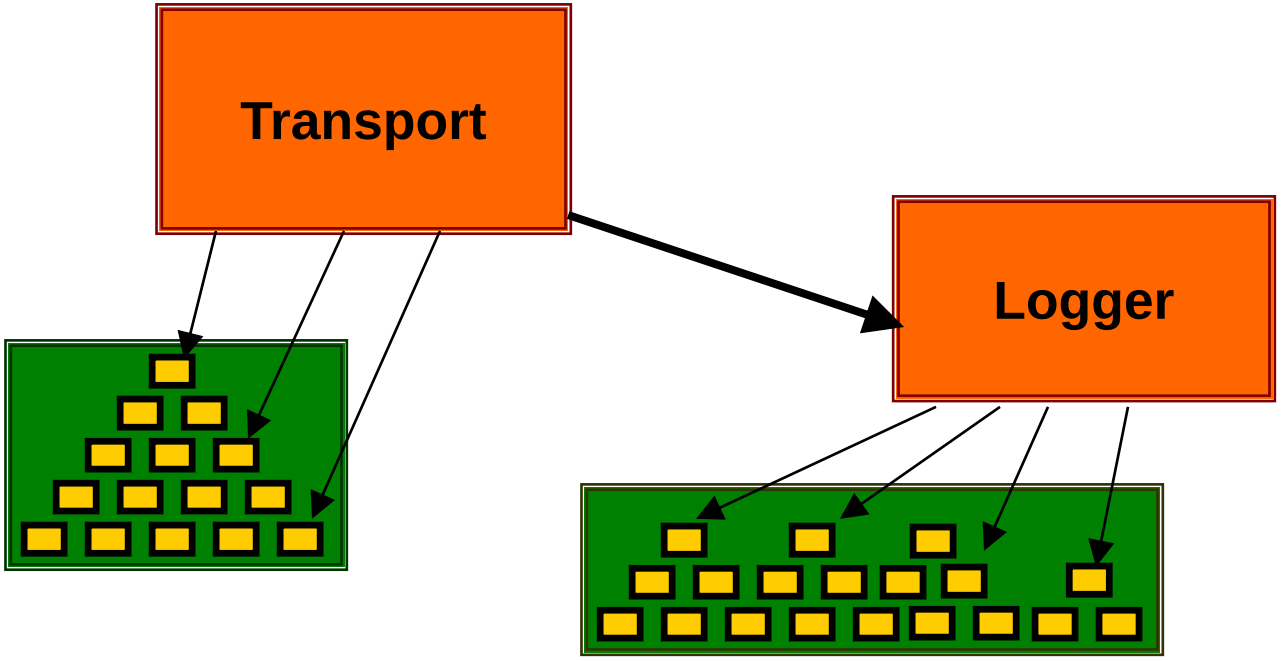
Large-Scale Physical Design

- Good physical design is an **engineering discipline**, **not** an afterthought.
- Good physical design must be introduced from the **inception** of an application.
- The physical design of our proprietary libraries should be **coherent** across the firm.

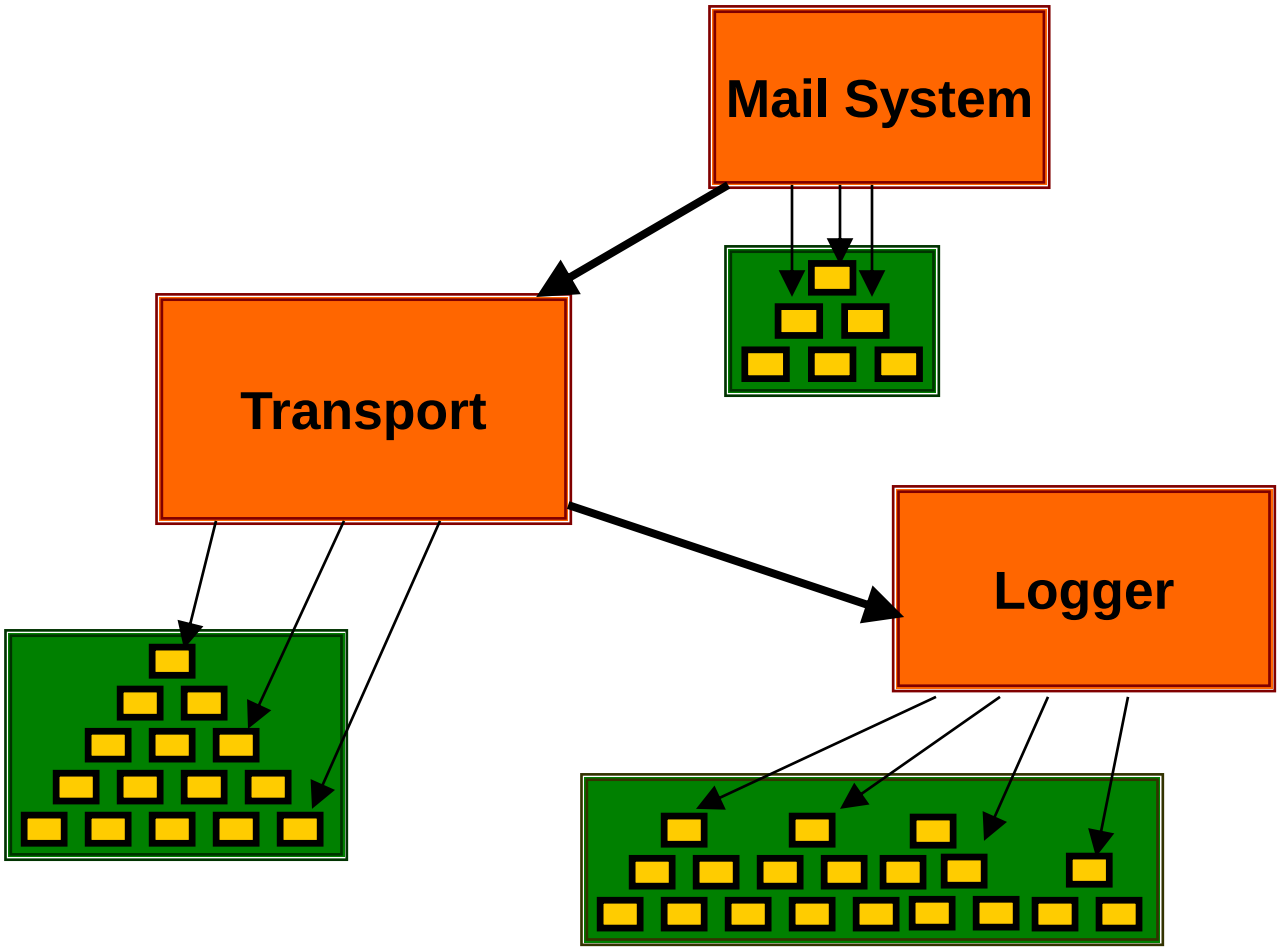
Cyclic Link-time Dependency



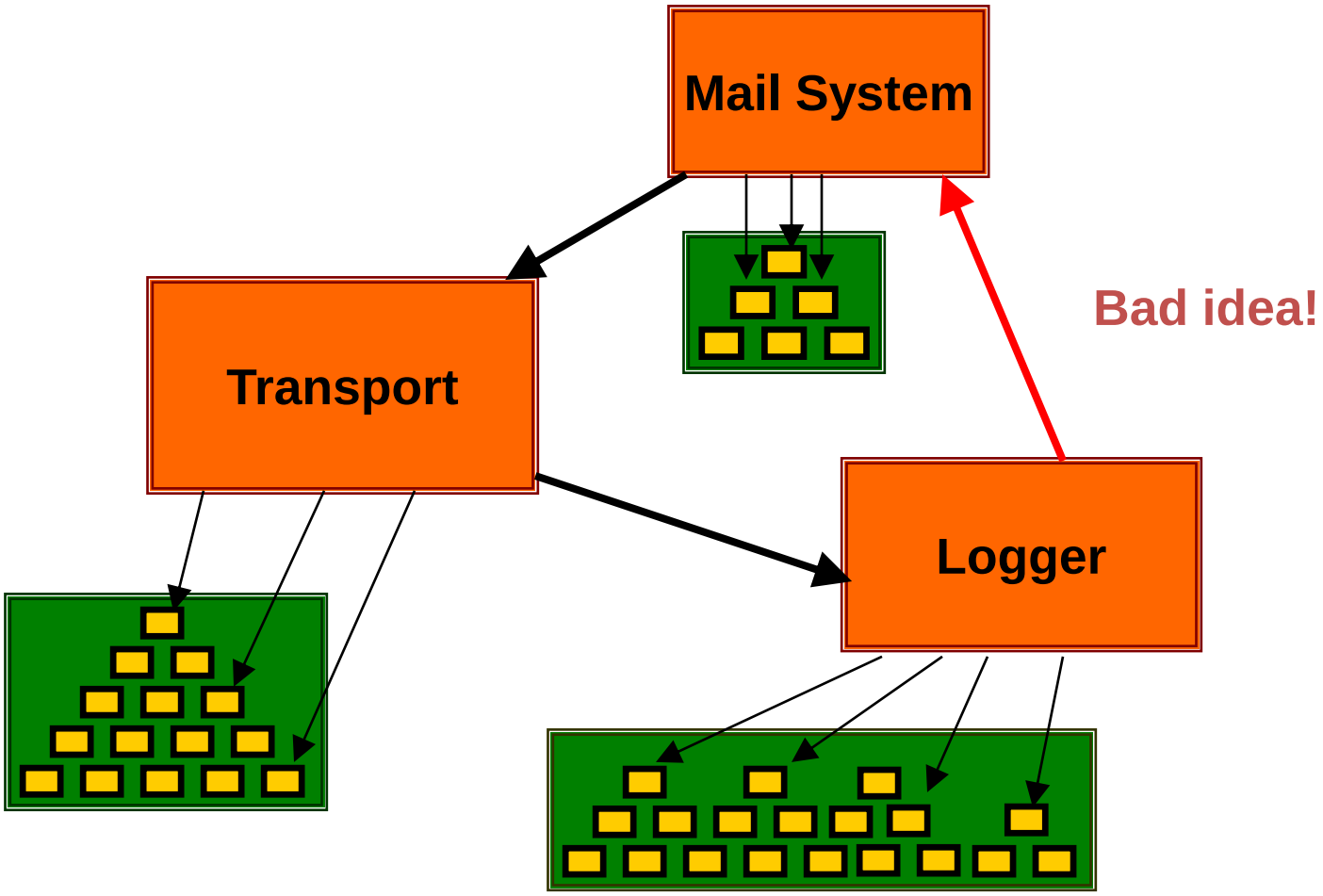
Cyclic Link-time Dependency



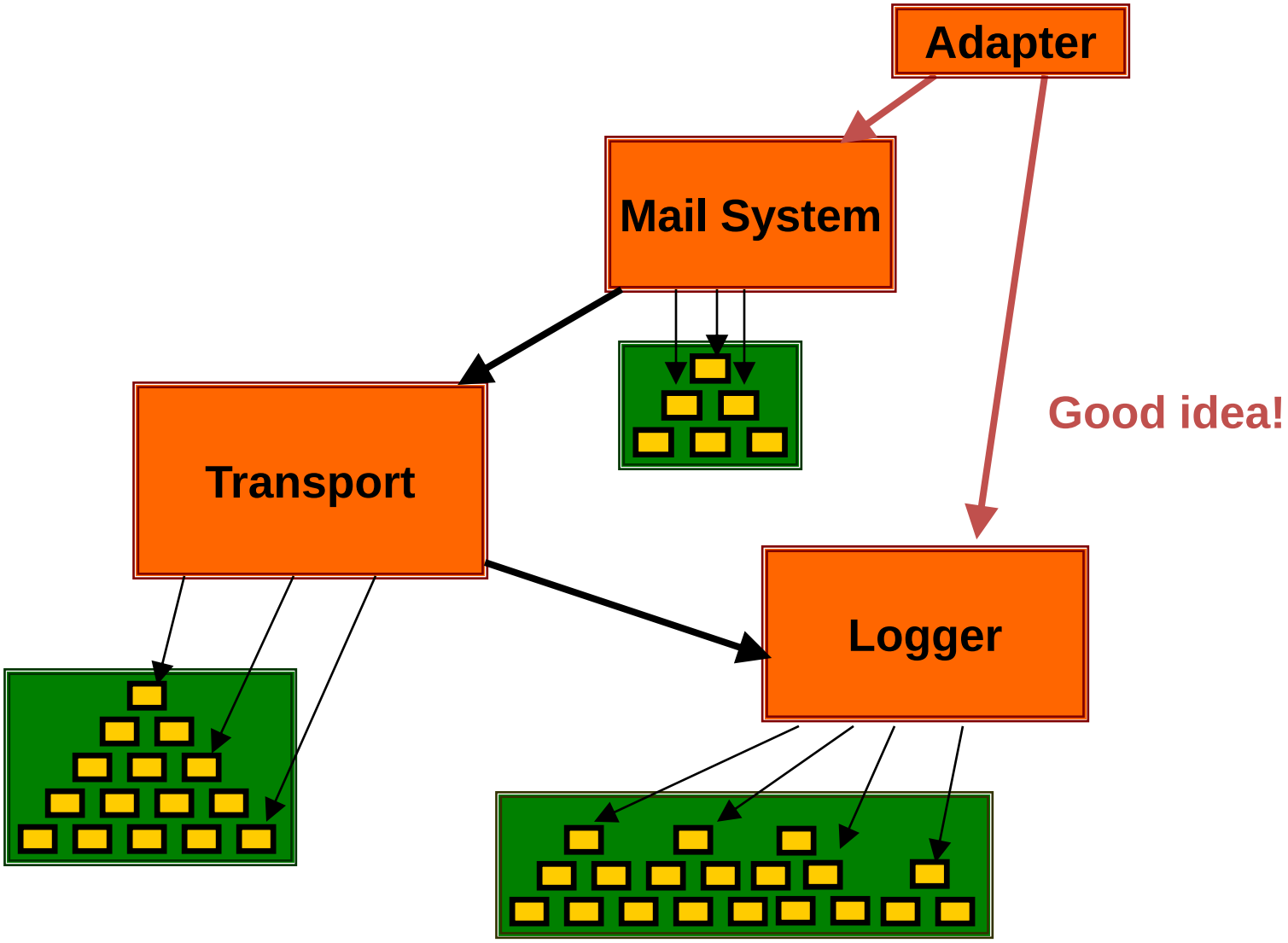
Cyclic Link-time Dependency



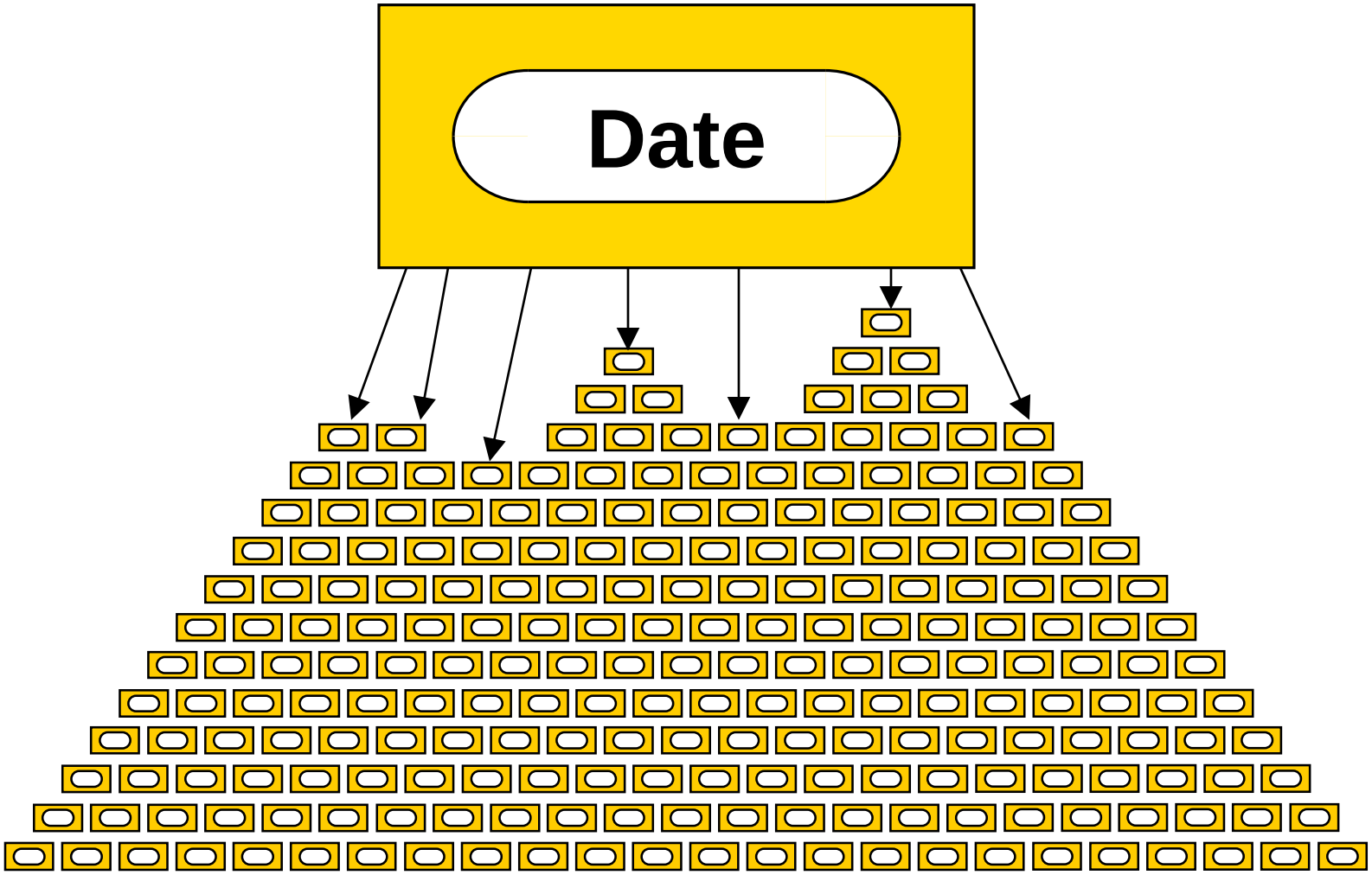
Cyclic Link-time Dependency



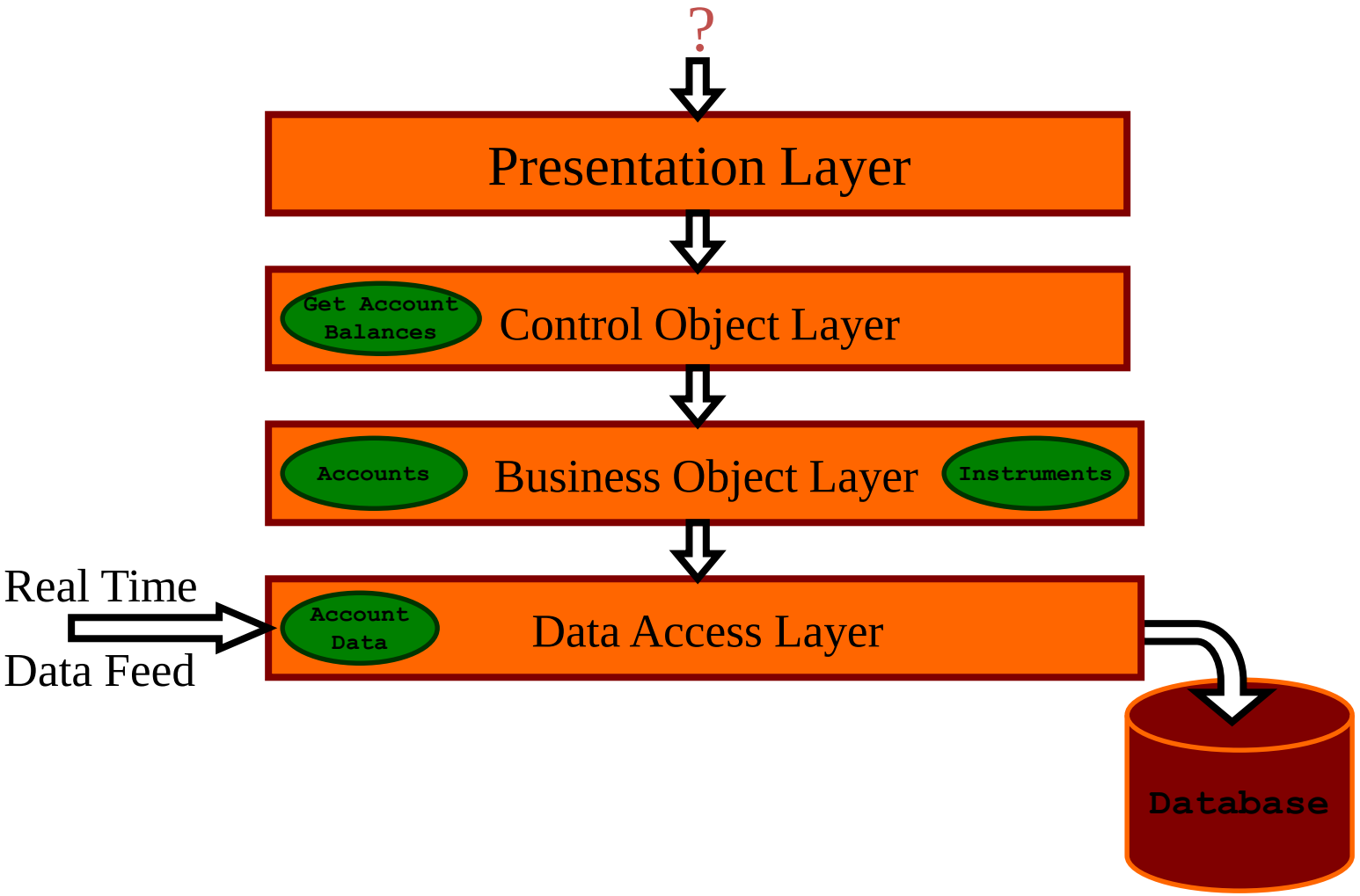
Cyclic Link-time Dependency



Excessive Link-time Dependency



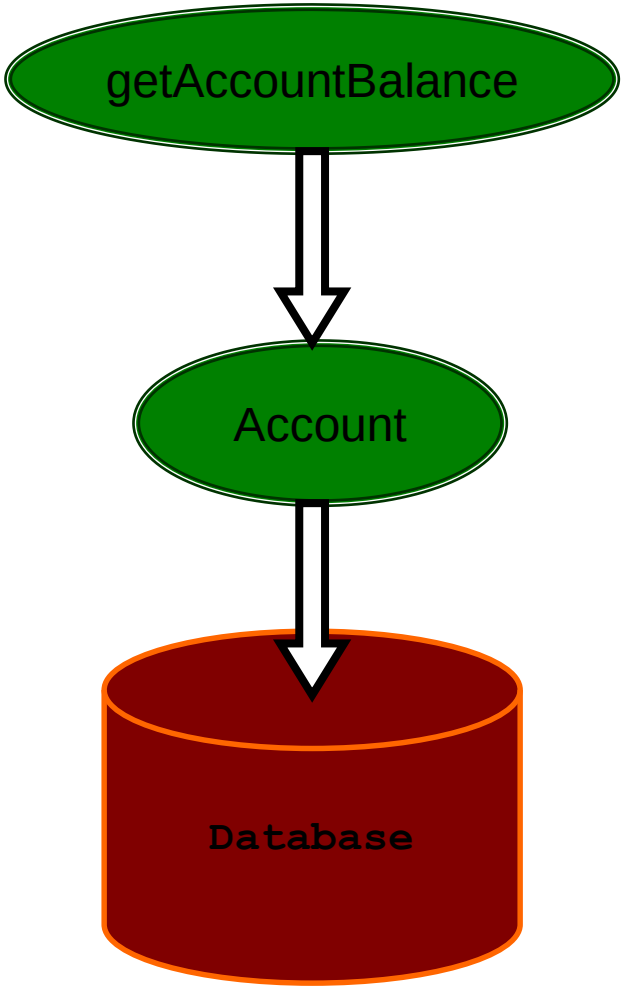
Classical Layered Architecture



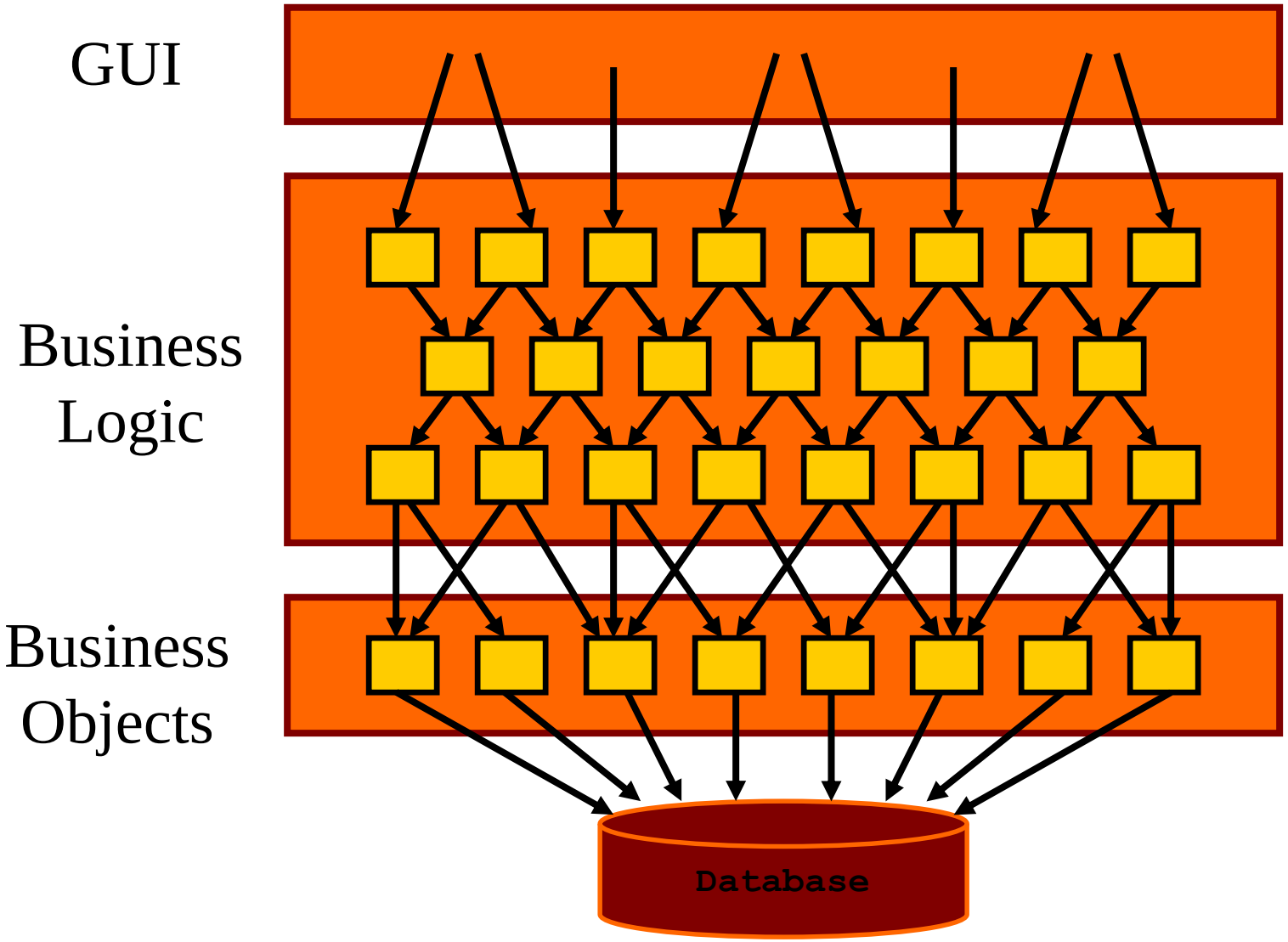
What Does `Account` Depend On?

```
class Account {
    // ...
  public:
    Account(int accountNumber);
        // Create an account
        // corresponding to the
        // specified 'accountNumber'
        // in the database.
    // ...
};
```

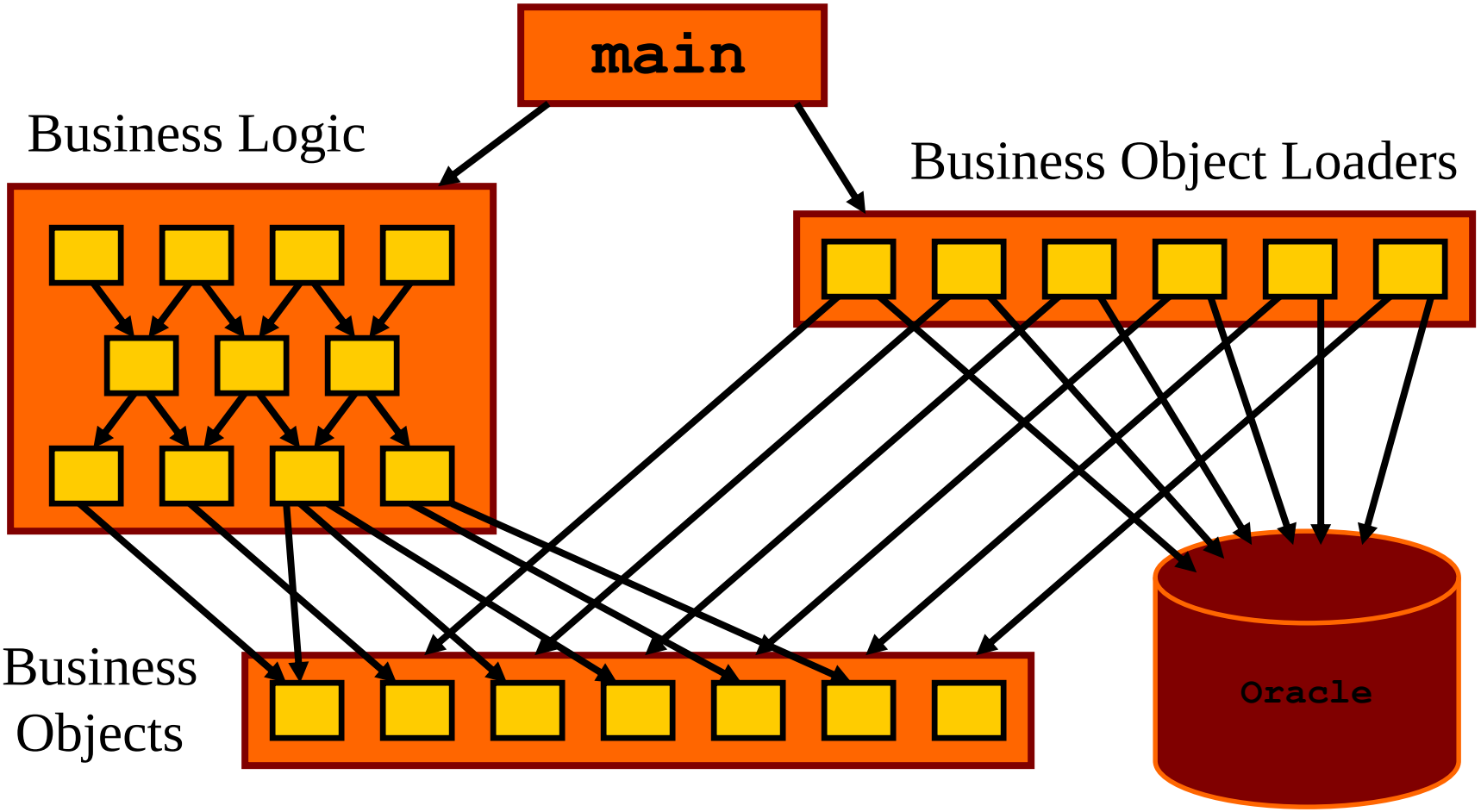
On the Database!



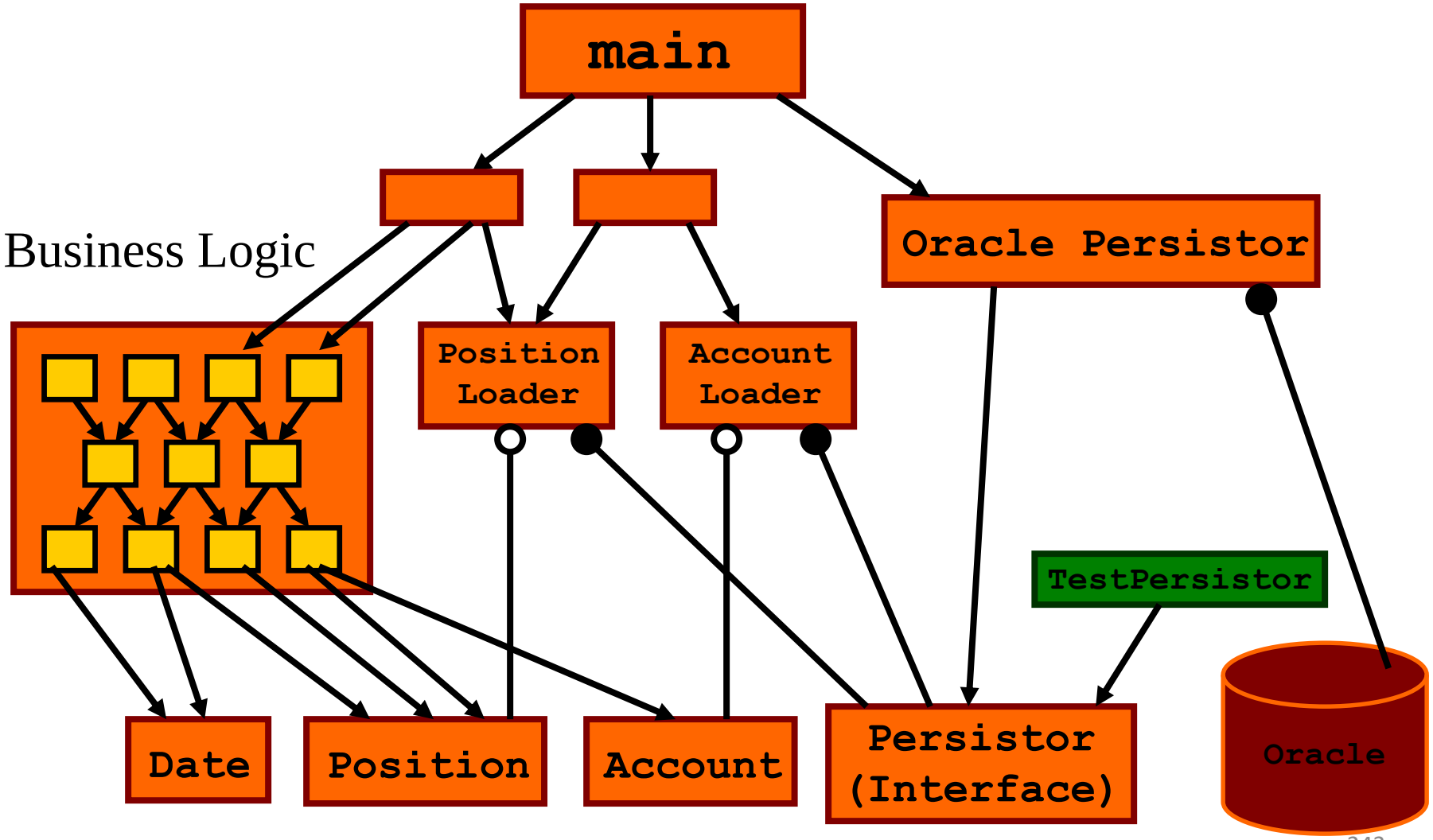
Everything Depends on the Database!



Escalating Heavy-Weight Dependencies



Breaking Dependencies Via Interfaces



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* its physical dependencies
- Are we 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?

(Shameless)

Advertisement

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

Let's learn about some of these techniques now!

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

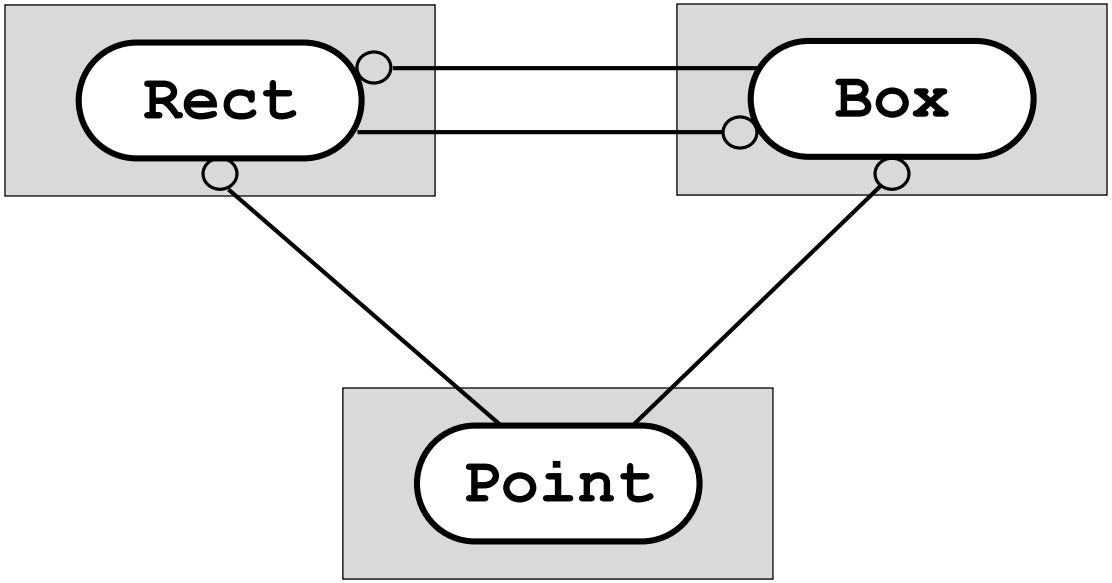
Later, we'll consider them in the context of C++ Modules.

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.***

Escalation

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

Escalation



Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  Point d_origin;
  int   d_width;
  int   d_length;
 public:
  // …
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  Point d_lowerLeft;
  Point d_upperRight;

 public:
  // …
  Box(const Rect& r);
  // …
};
```

Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  Point d_origin;
  int   d_width;
  int   d_length;
 public:
  // …
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  Point d_lowerLeft;
  Point d_upperRight;

 public:
  // …
  Box(const Rect& r);
  // …
};
```

Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& ll,
       int w, int h);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

Implicit Conversions

Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

Escalation

```
// rect.h
#include <point.h>
class Box;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

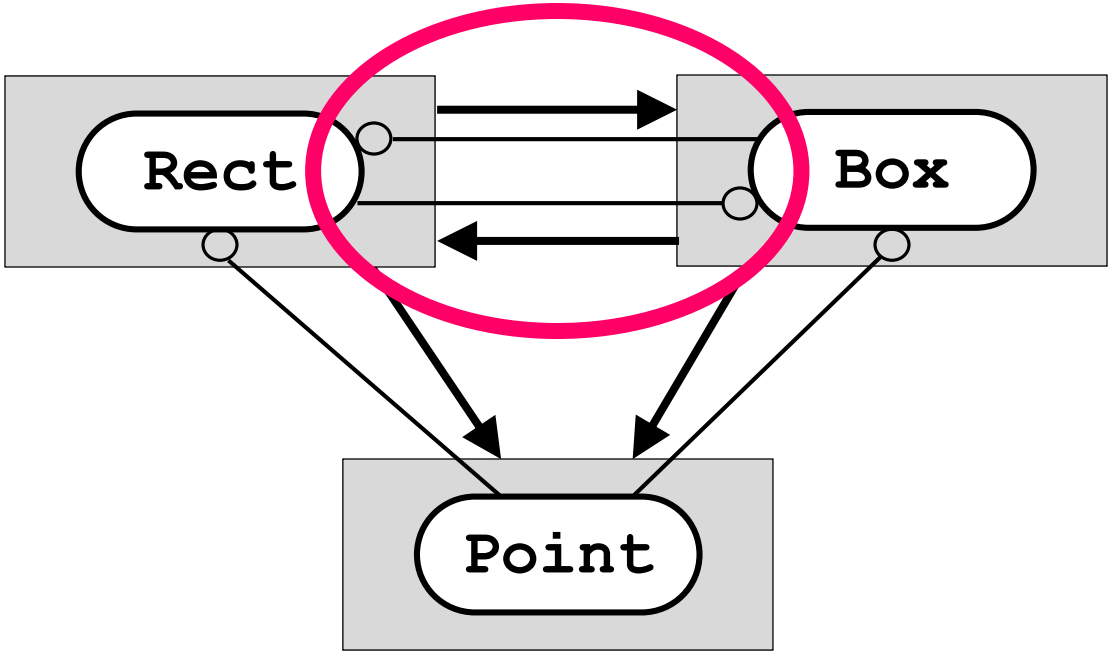
```
// box.h
#include <point.h>
class Rect;
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

Escalation

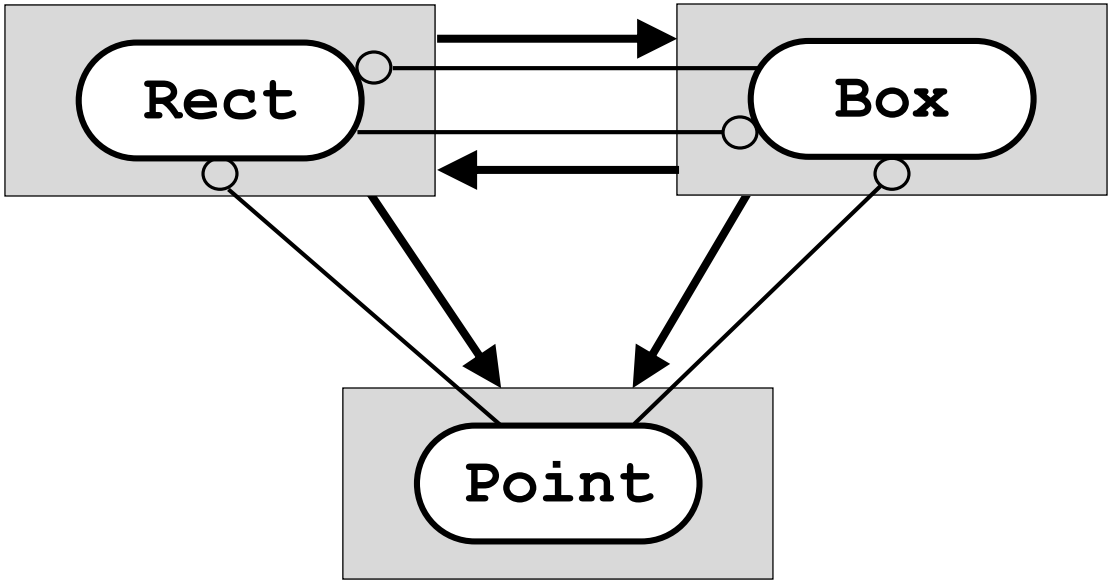
```
// rect.h
#include <point.h>
class Box;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
class Rect;
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

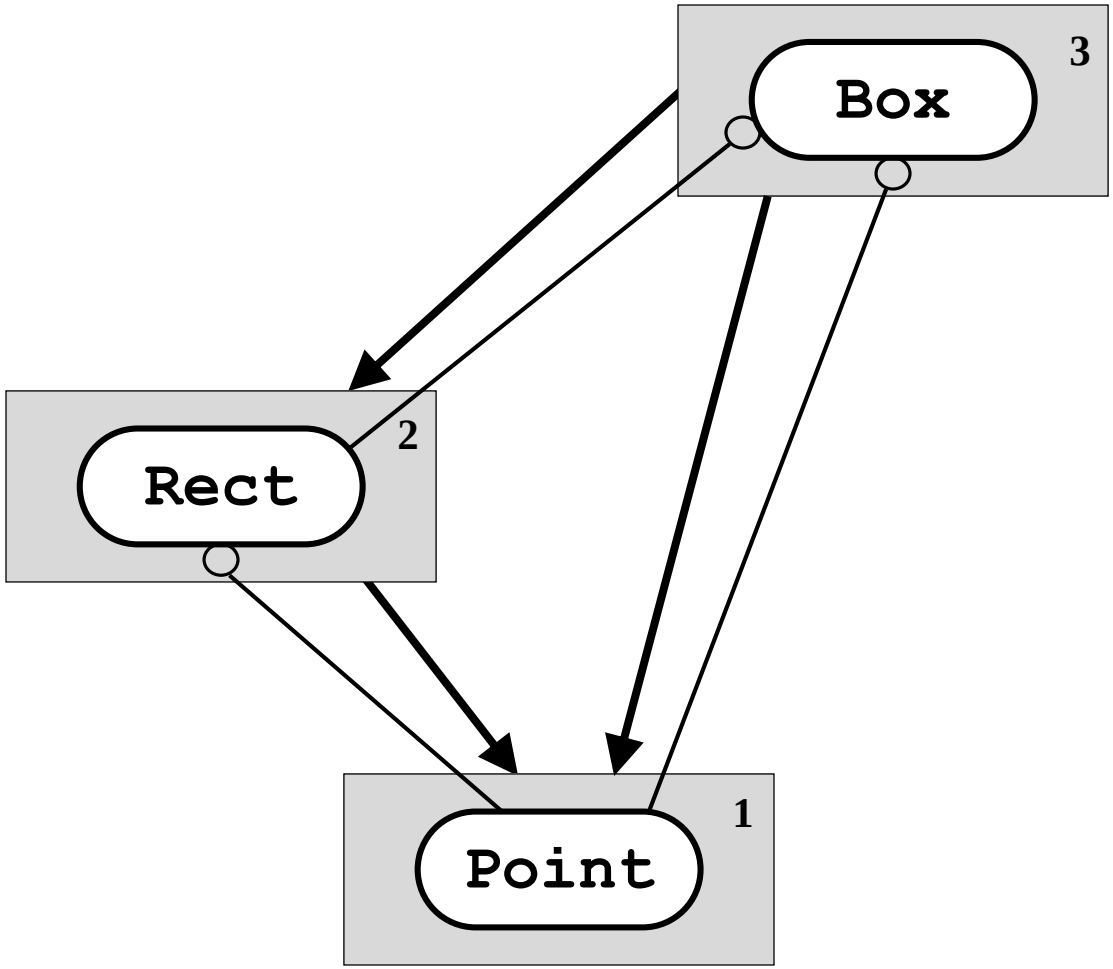
Escalation



Escalation



Escalation



Escalation

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

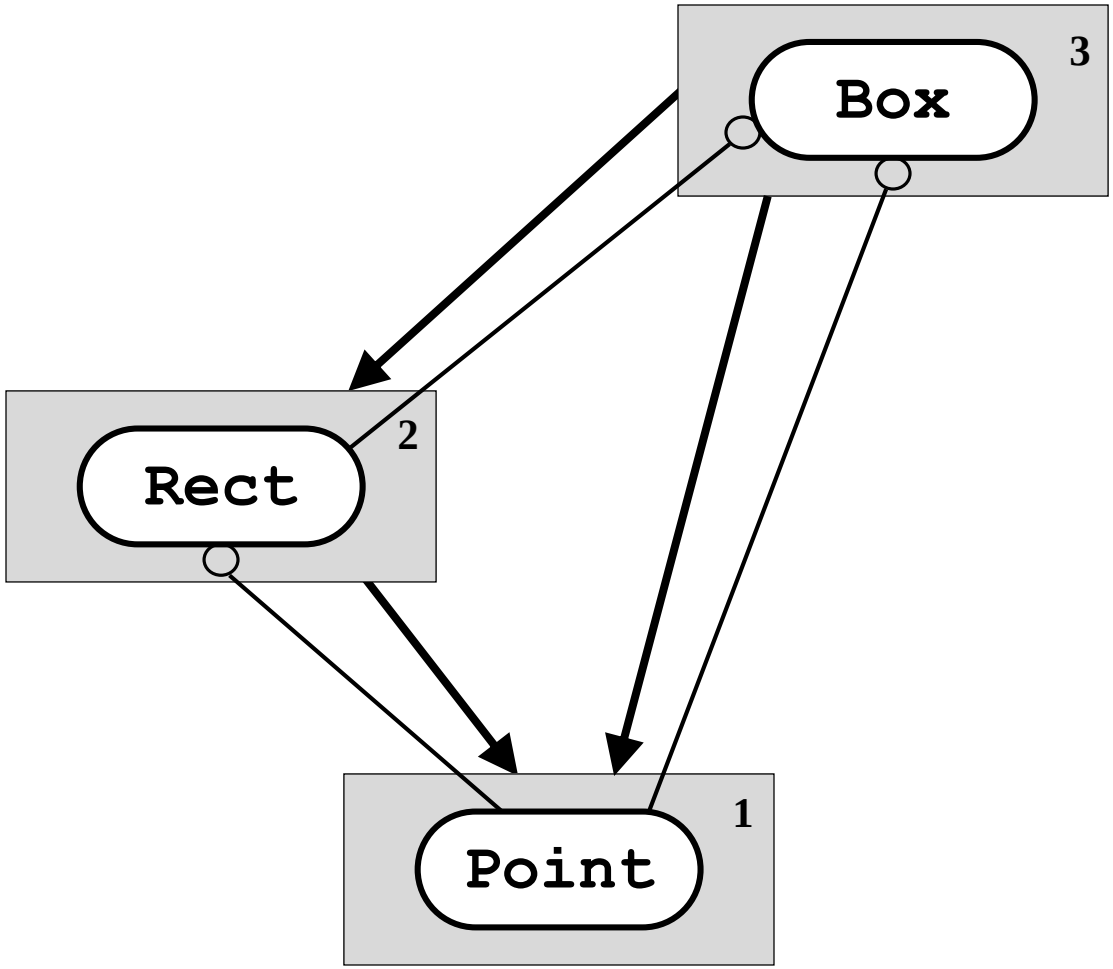
```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(Rect& r);
  // …
  operator Rect() const;
  // …
};
```

Escalation

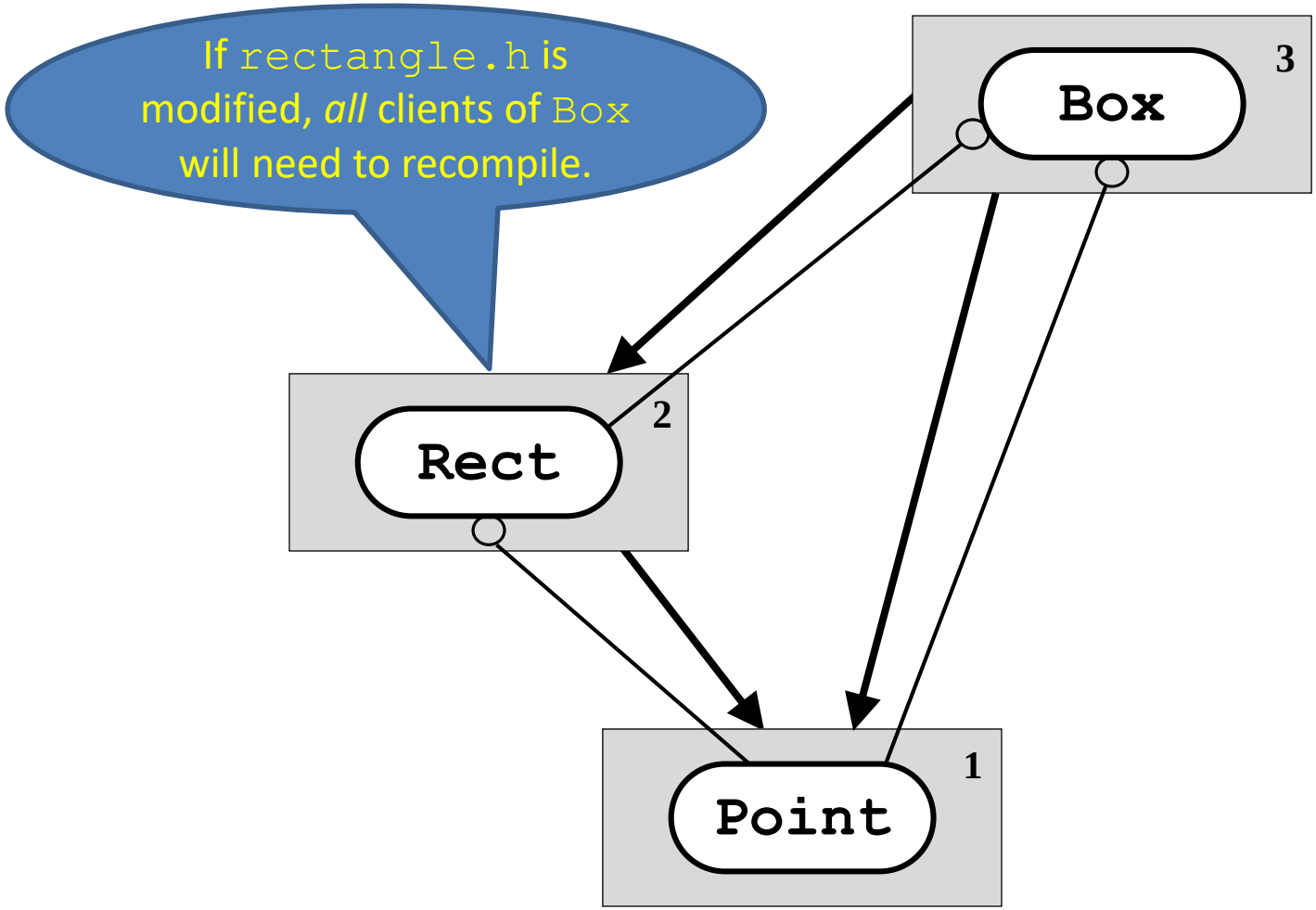
```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(Rect& r);
  // …
  operator Rect() const;
  // …
};
```

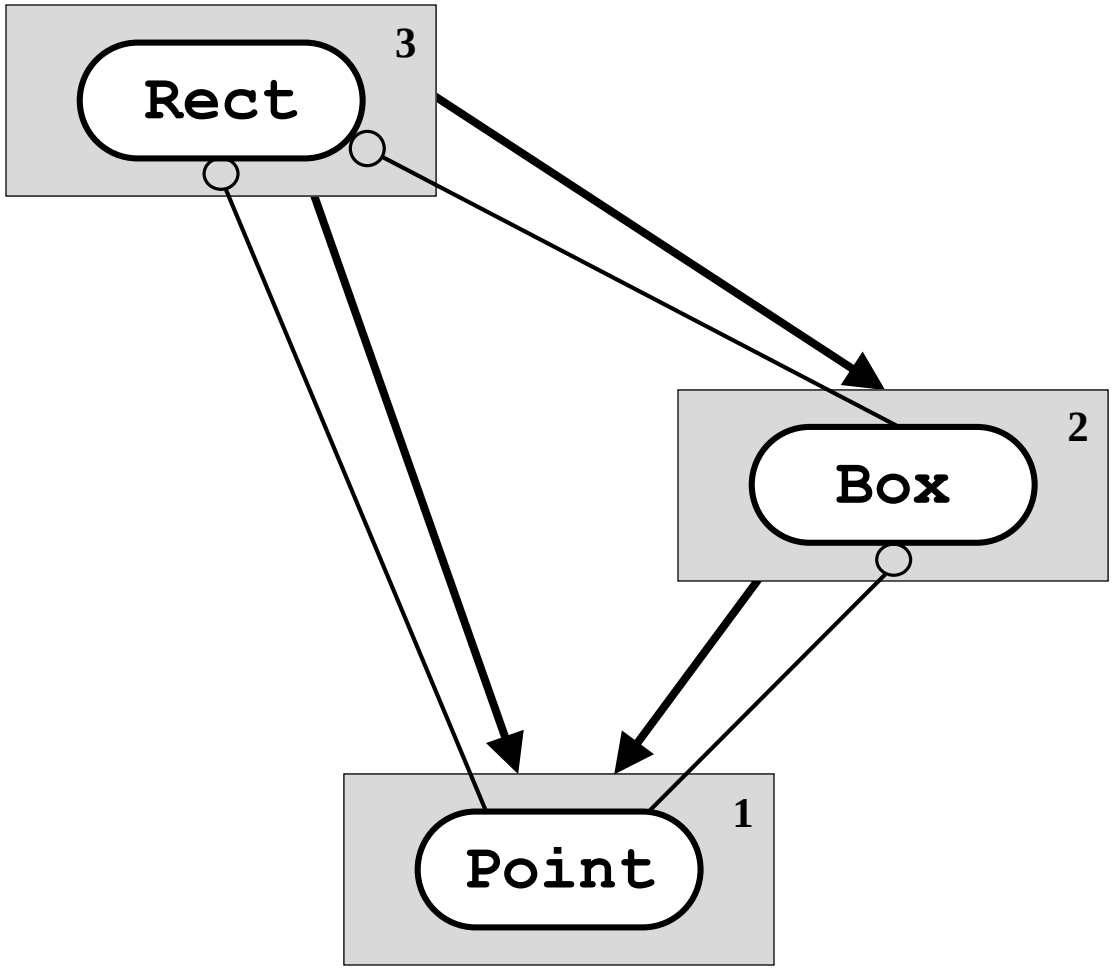
Escalation



Escalation


If rectangle.h is modified, all clients of Box will need to recompile.
Box
3
Rect
2
Point
1

Escalation



Escalation

```
// rect.h
#include <point.h>
class Box;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
  operator Box() const;
  // …
};
```

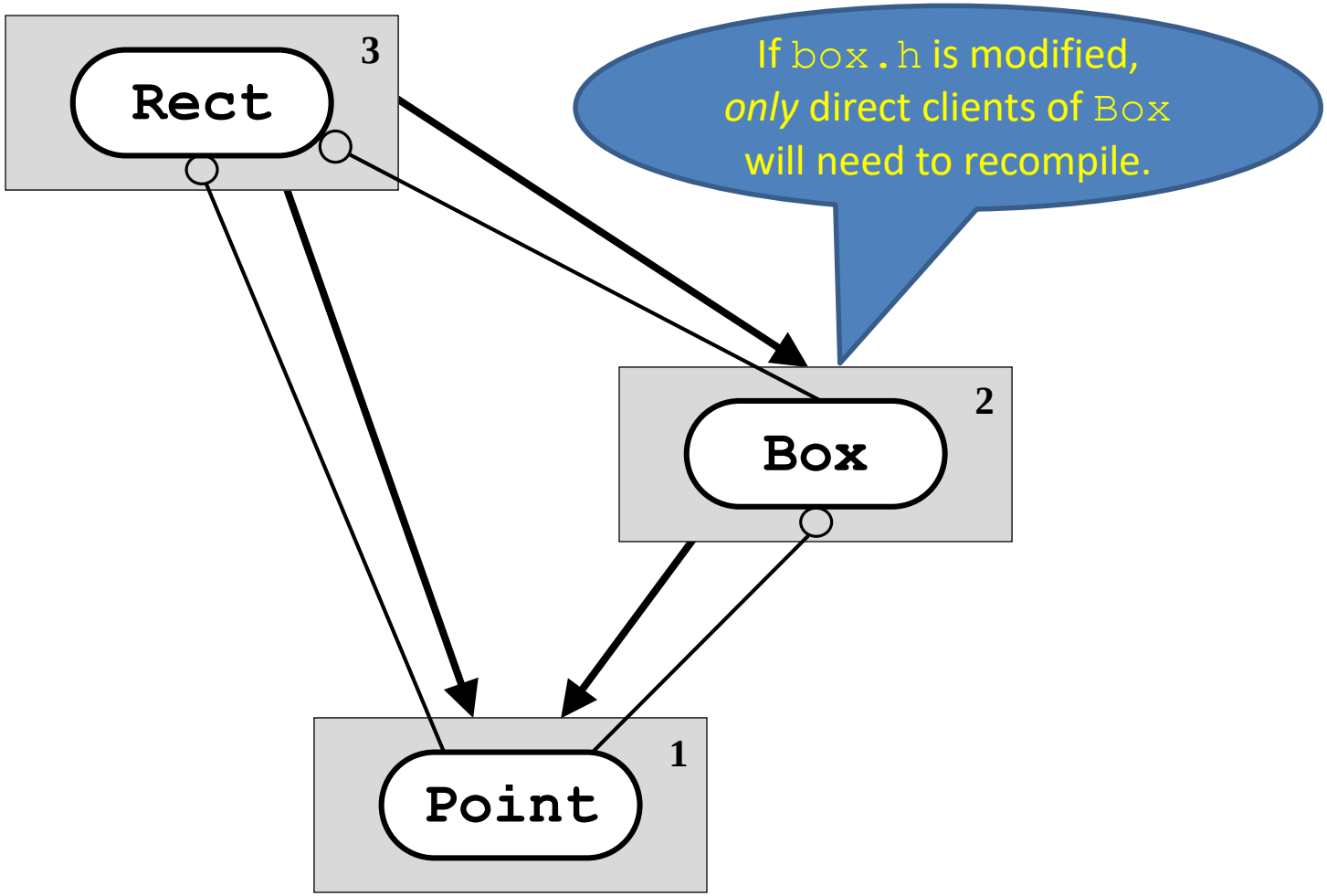
```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

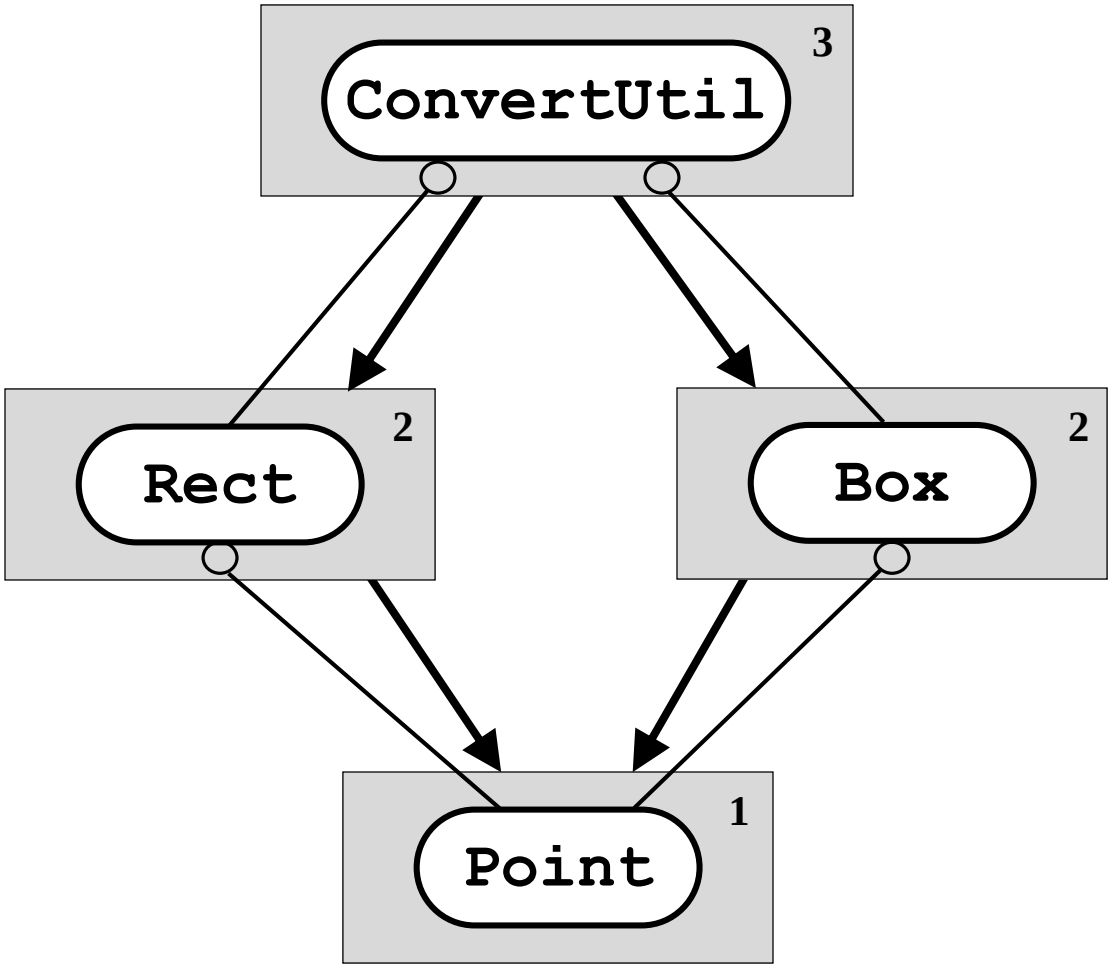
```
// rect.h
#include <point.h>
class Box;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
  operator Box() const;
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation


Rect
3
Box
2
Point
1
If box.h is modified,
only direct clients of Box
will need to recompile.

Escalation



Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

in order to support *inline functions*

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

in order to support *inline functions*

more on this later

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct convertutil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

in order to avoid *transitive includes*

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect
  static Rect rectFromBox
  // …
};
```

in order to avoid *transitive includes*

Presumes `inline` implementations of *(elided)* functions using `Point`.

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect
  static Rect rectFromBox
  // …
};

inline
ConvertUtil::boxFromRectangle(
                          const Rect& r)
{
  int length = r.ur.x() - r.ll.x();
  int width  = r.ur.y() - r.ll.y();
  Point origin(r.ll.x() + length/2,
               r.ll.y() + width/2);
  return Box(origin, length, width);
}
```

in order to avoid *transitive includes*

Presumes `inline` implementations of *(elided)* functions using `Point`.

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect
  static Rect rectFromBox
  // …
};
```

in order to avoid *transitive includes*

Presumes `inline` implementations of *(elided)* functions using `Point`.

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>

struct convertutil {
  static Box boxFromRect
  static Rect rectFromBox
  // …
};
```

in order to avoid *transitive includes*

Presumes `inline` implementations of *(elided)* functions using `Point`.

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>

struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>

struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

```
// rect.h
class Point;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
class Point;
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>

class convertutil {
  Box boxFromRect(const Rect& r);
  Rect rectFromBox(const Box& b);
};
```

```
// rect.h
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

convertutil.h No Longer Compiles!

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect
  static Rect rectFromBox
  // …
};
```

in order to avoid *transitive includes*

Presumes `inline` implementations of *(elided)* functions using `Point`.

```
// rect.h
class Point;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
class Point;
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

in order to avoid *transitive includes*

```
// rect.h
class Point;
class Rect {
  //
 pub
  Re
  Re

  //
};
```

```
// box.h
class Point;
class Box {
```

convertutil.h Again Compiles!

Escalation

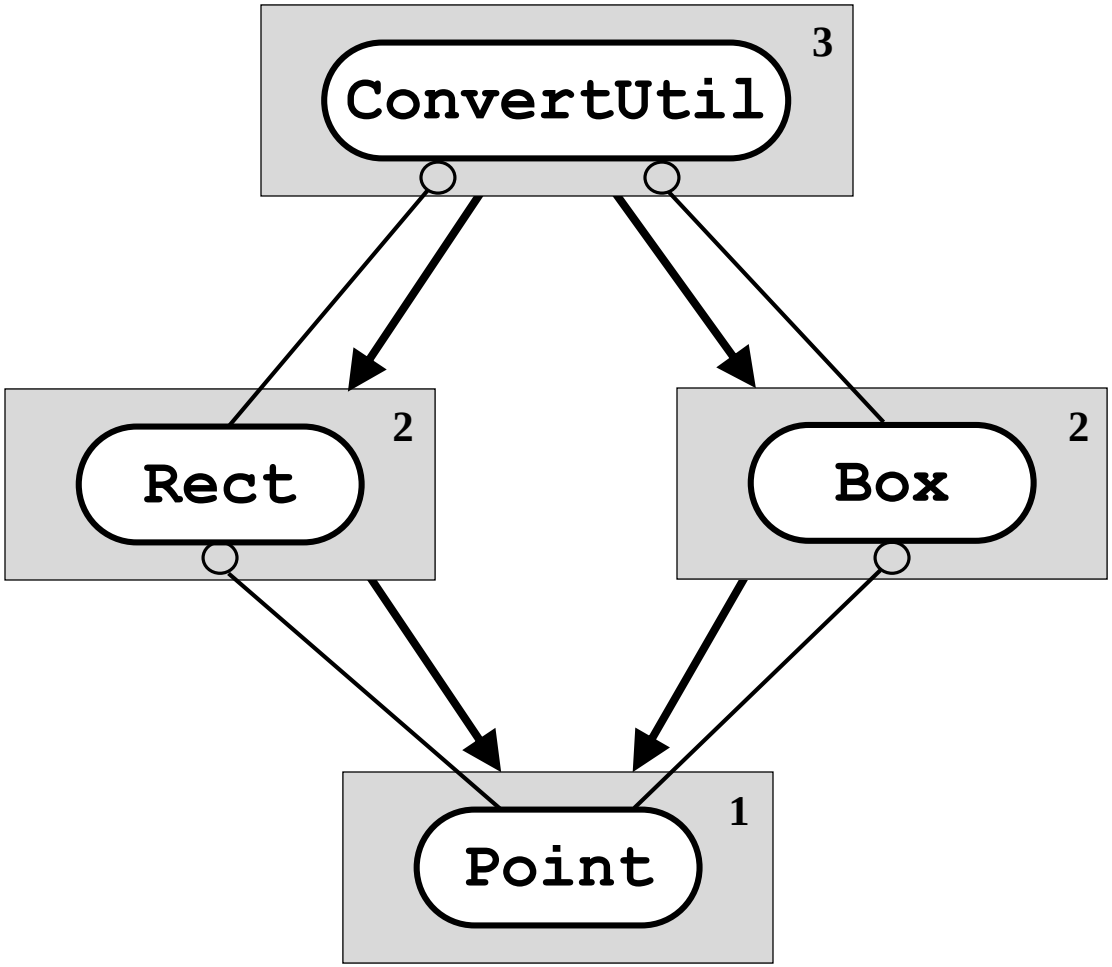
```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

in order to avoid *transitive includes*

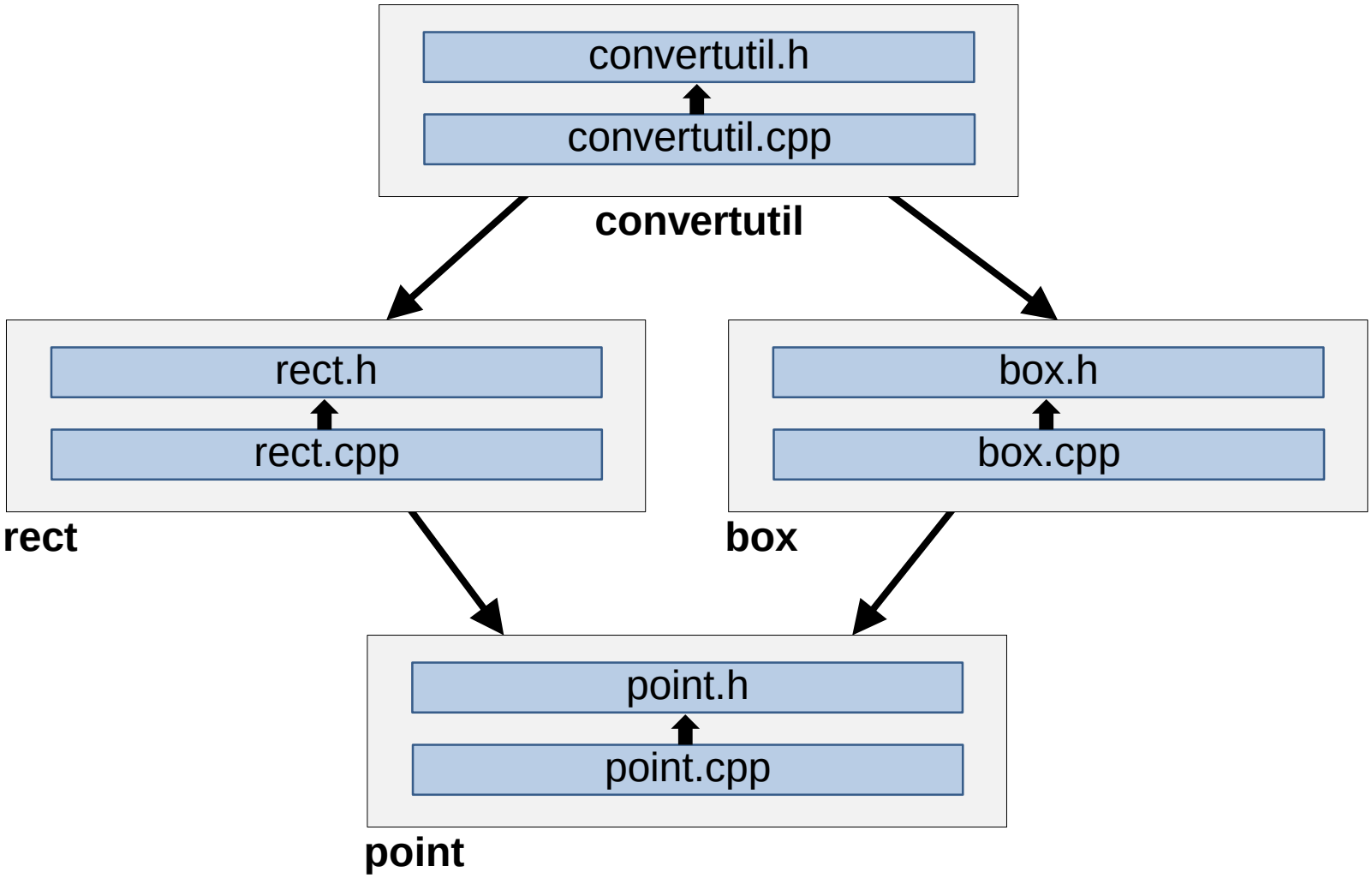
```
// rect.h
class Point;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
class Point;
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

Escalation



Escalation


convertutil.h
convertutil.cpp
convertutil
rect.h
rect.cpp
rect
box.h
box.cpp
box
point.h
point.cpp
point

Escalation

Discussion?

Demotion

Demotion – Moving common functionality lower in the physical hierarchy.

Opaque Pointers

Opaque Pointers **–**

Having an object use another *in name only*.

Dumb Data

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

Redundancy

Redundancy **–**

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

Callbacks

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

Callbacks

There are several flavors:

1. DATA (Effectively *Demotion*)
2. FUNCTION (Stateless/Stateful)
3. FUNCTOR (Function Object)
4. PROTOCOL (Abstract Interface)
5. CONCEPT (Structural Interface)

Factoring

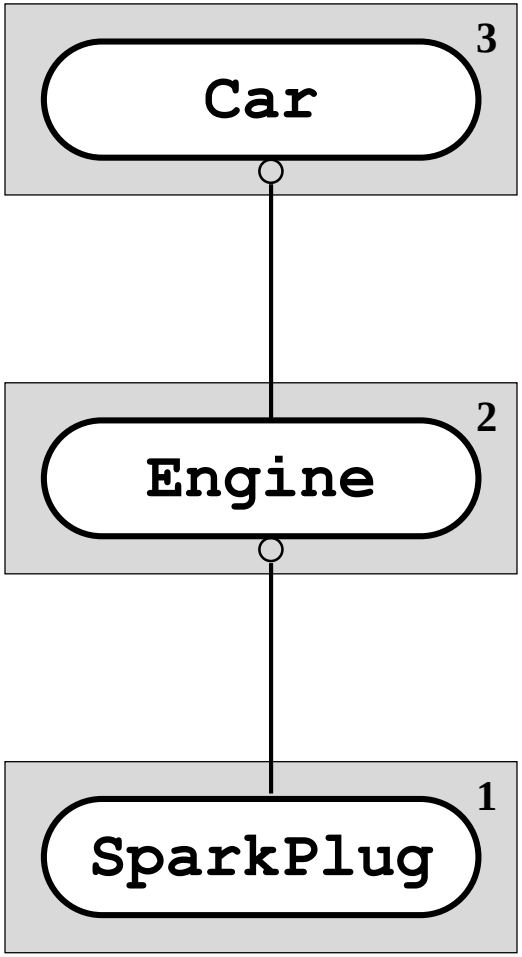
Factoring – Moving independently testable sub-behavior out of the implementation of a complex component involved in excessive physical coupling.

Escalating Encapsulation

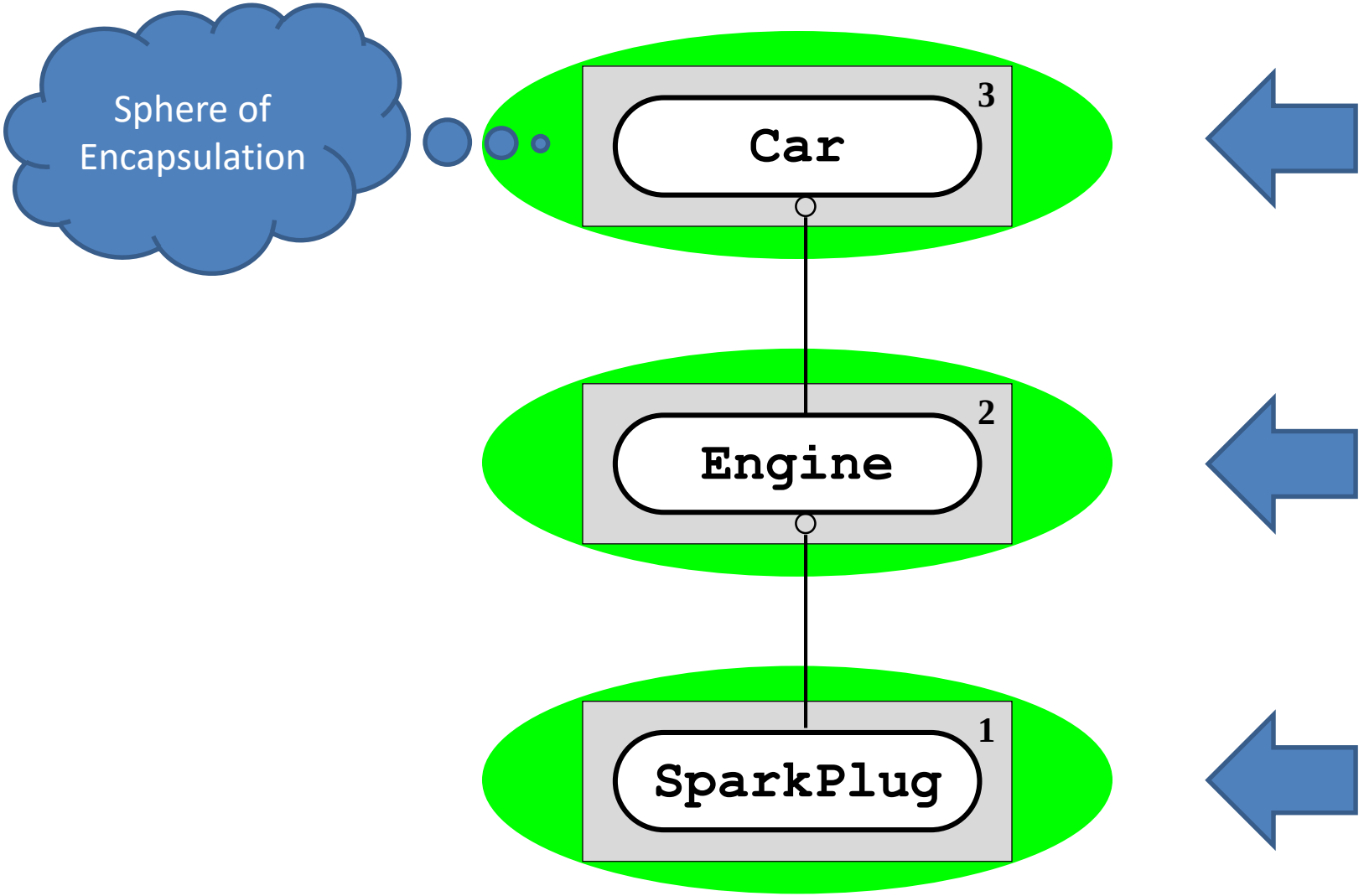
Escalating Encapsulation

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

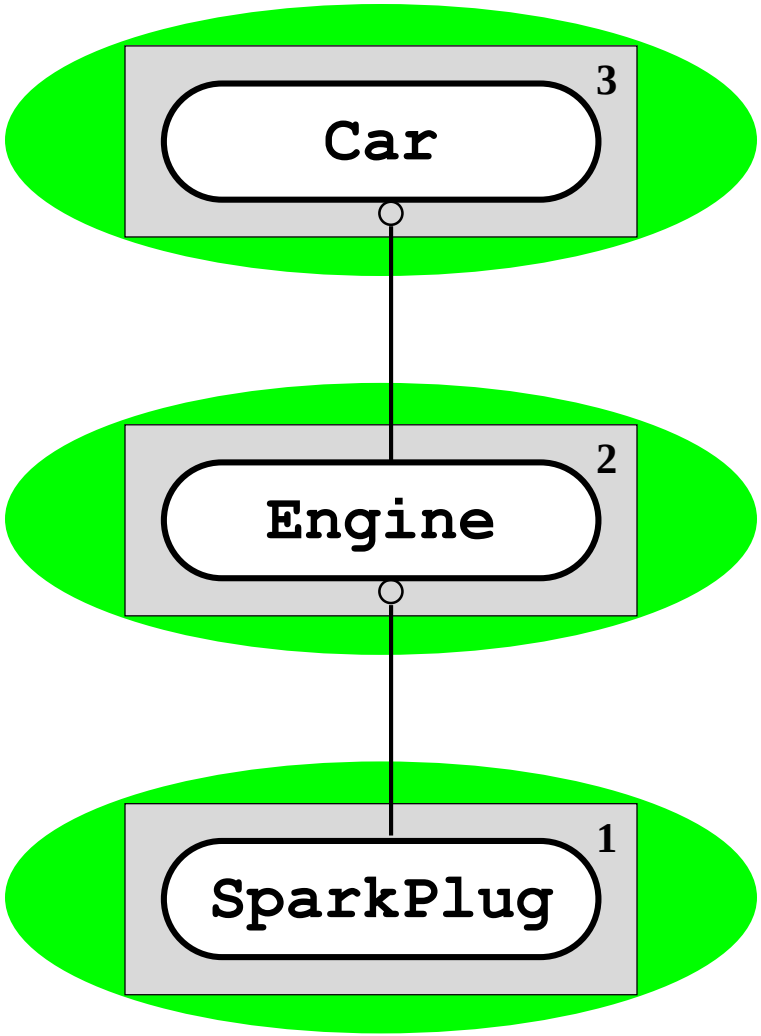
Escalating Encapsulation



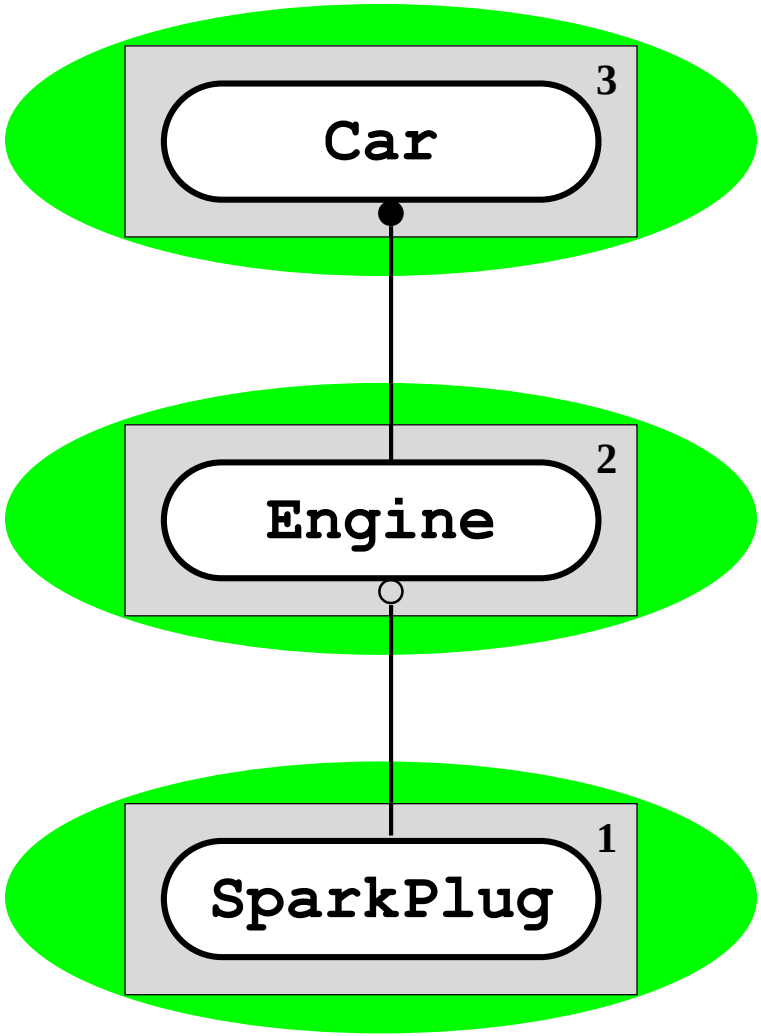
Escalating Encapsulation



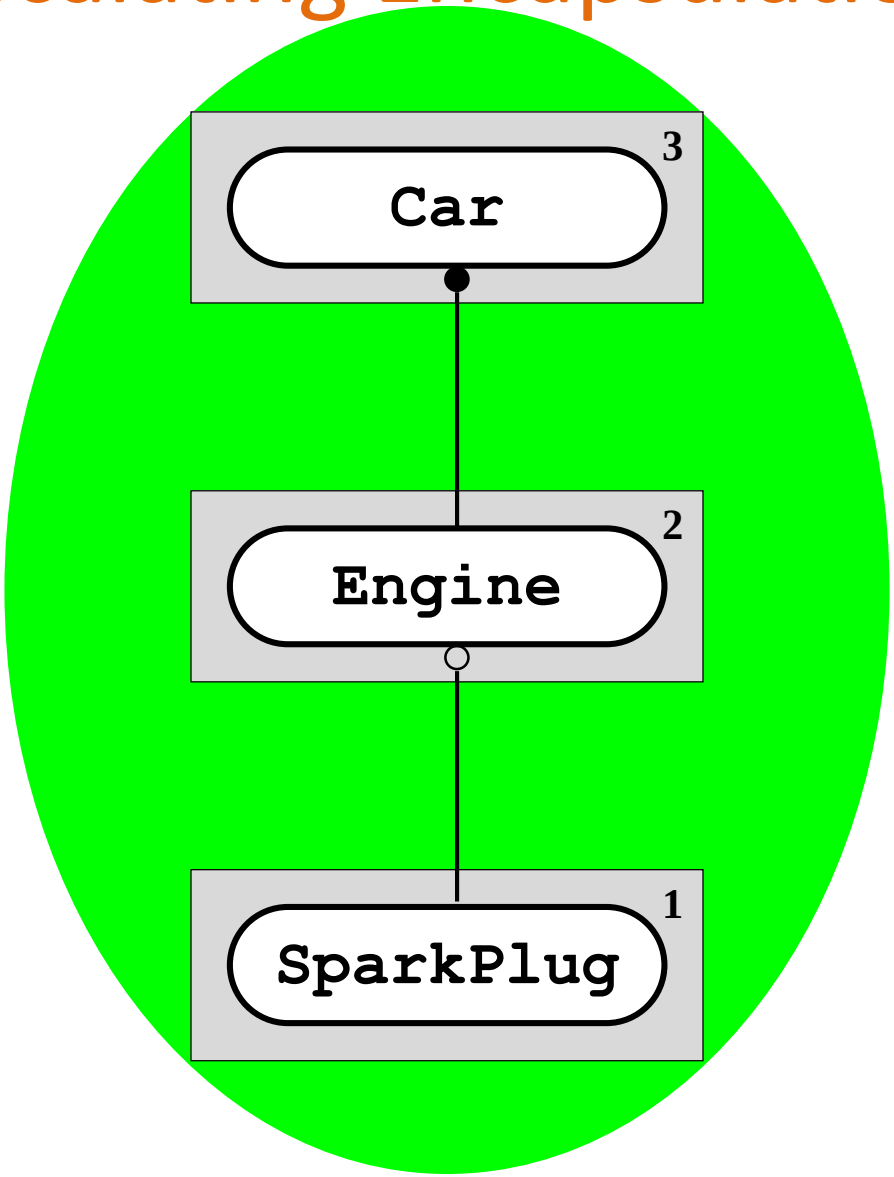
Escalating Encapsulation



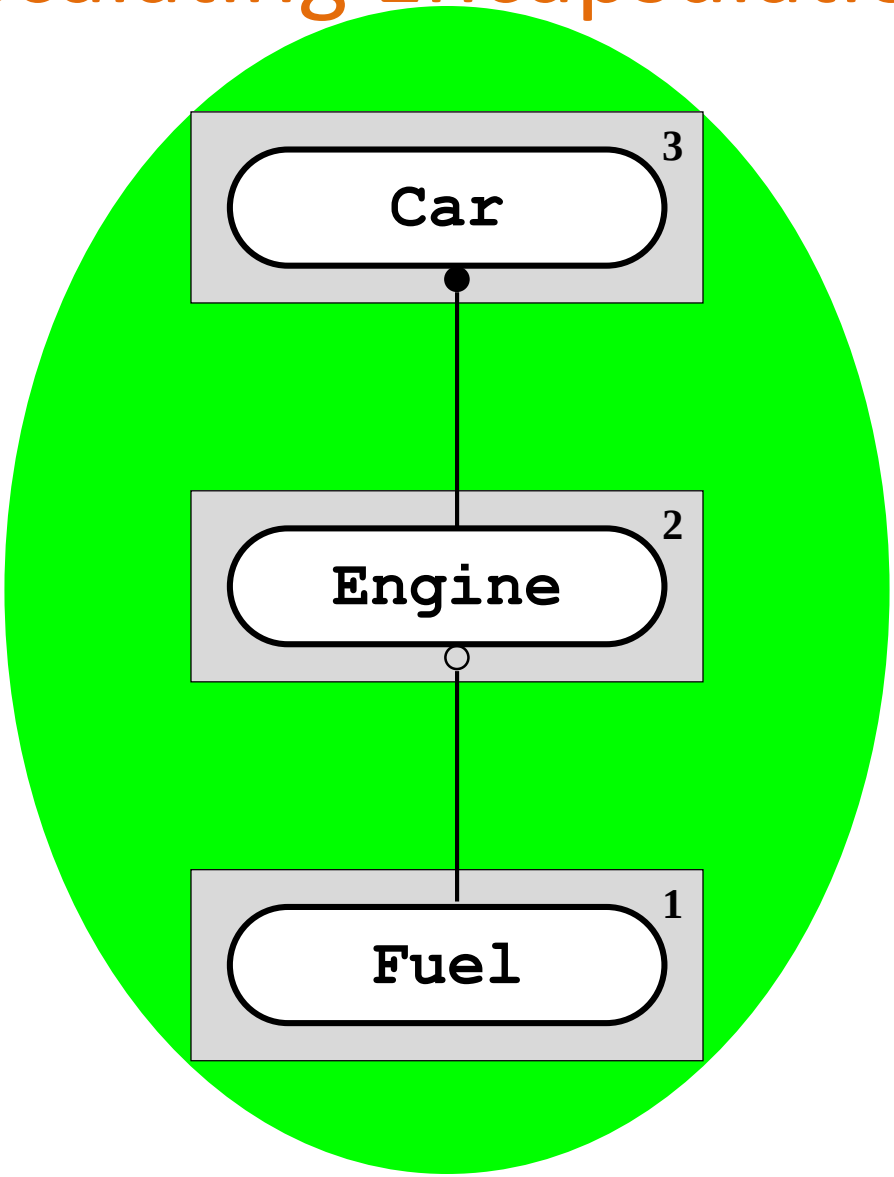
Escalating Encapsulation



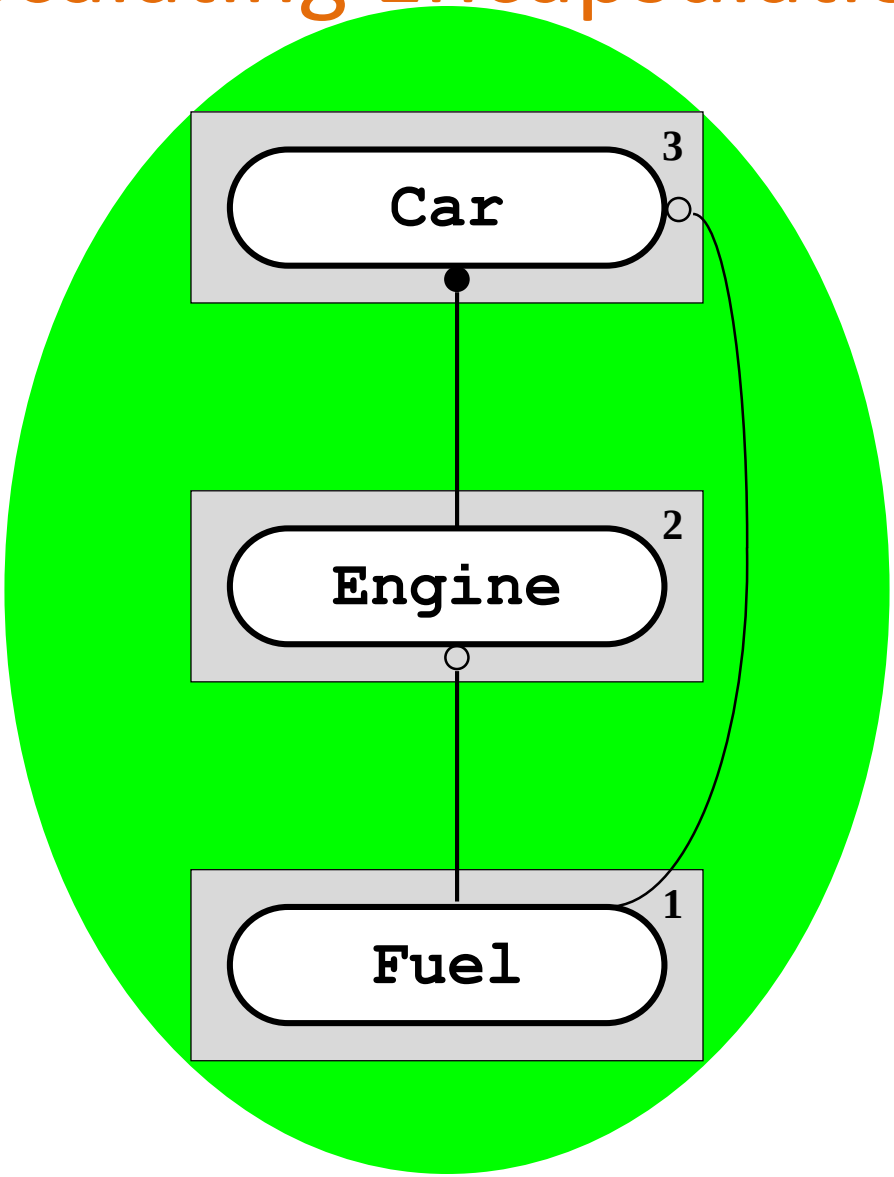
Escalating Encapsulation



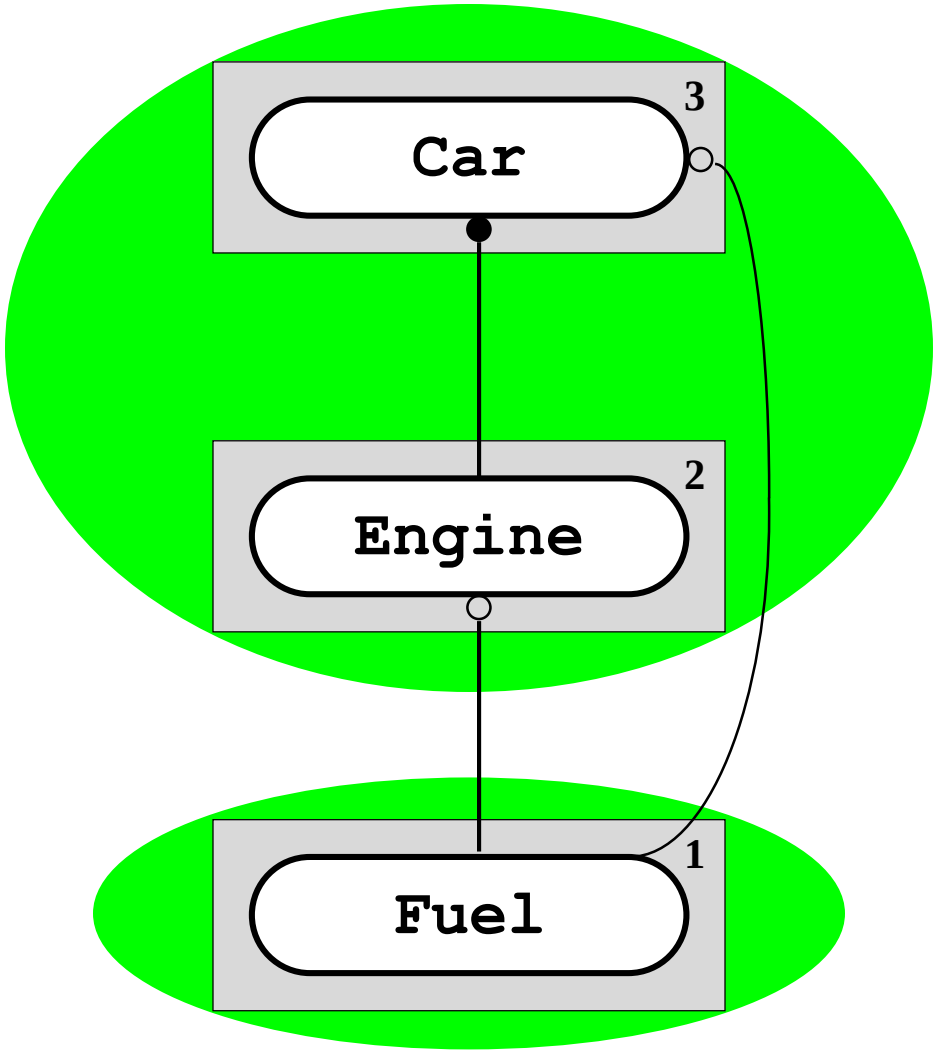
Escalating Encapsulation



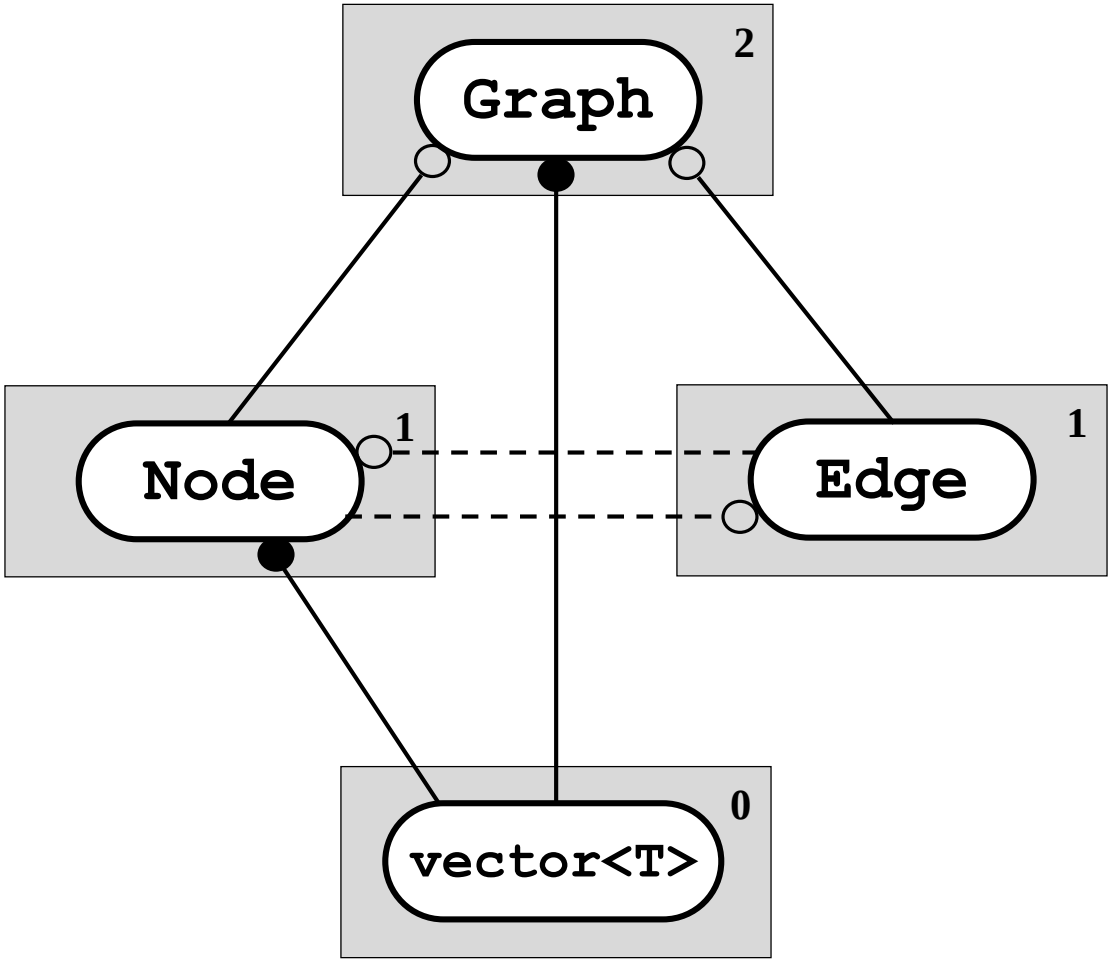
Escalating Encapsulation



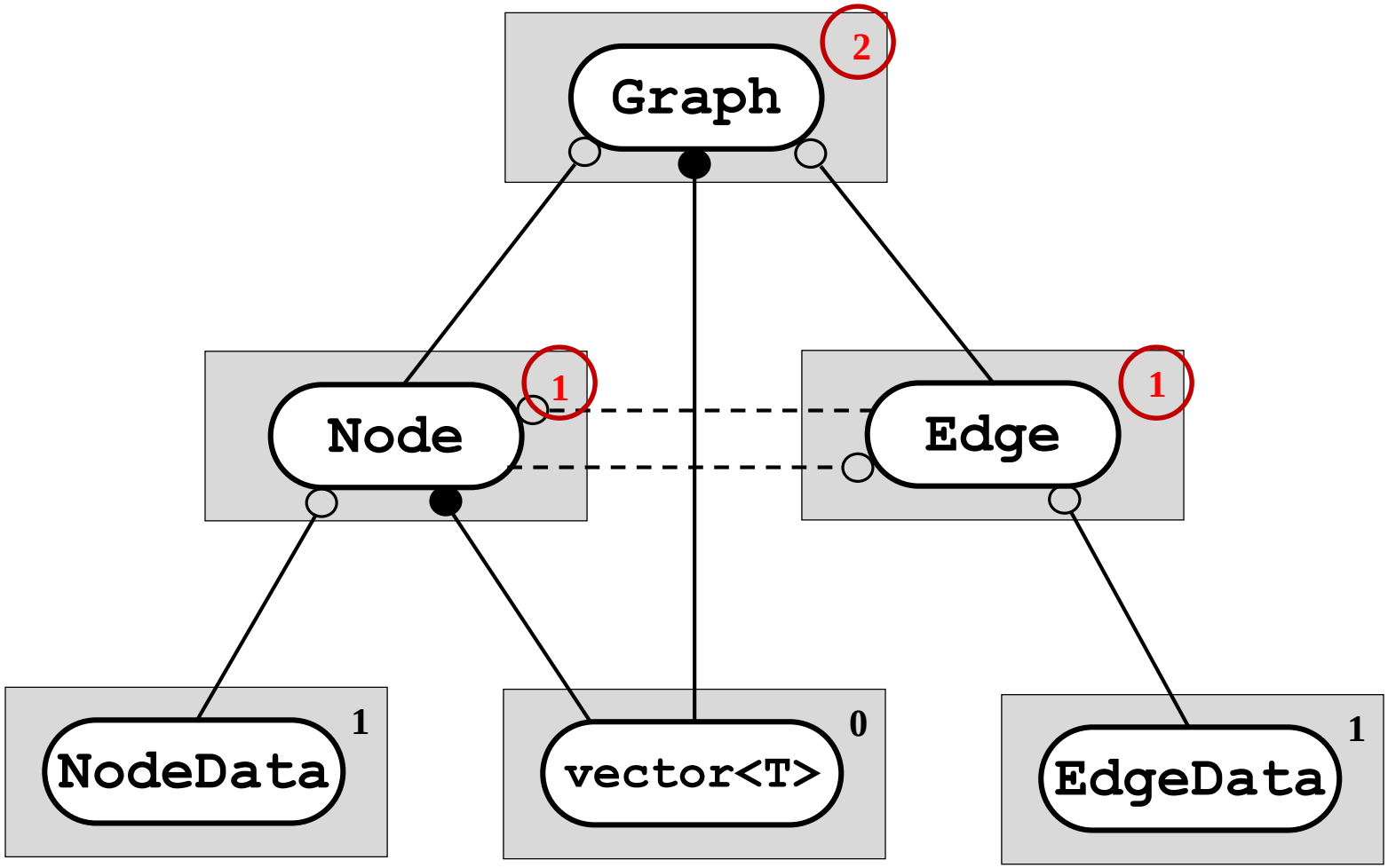
Escalating Encapsulation



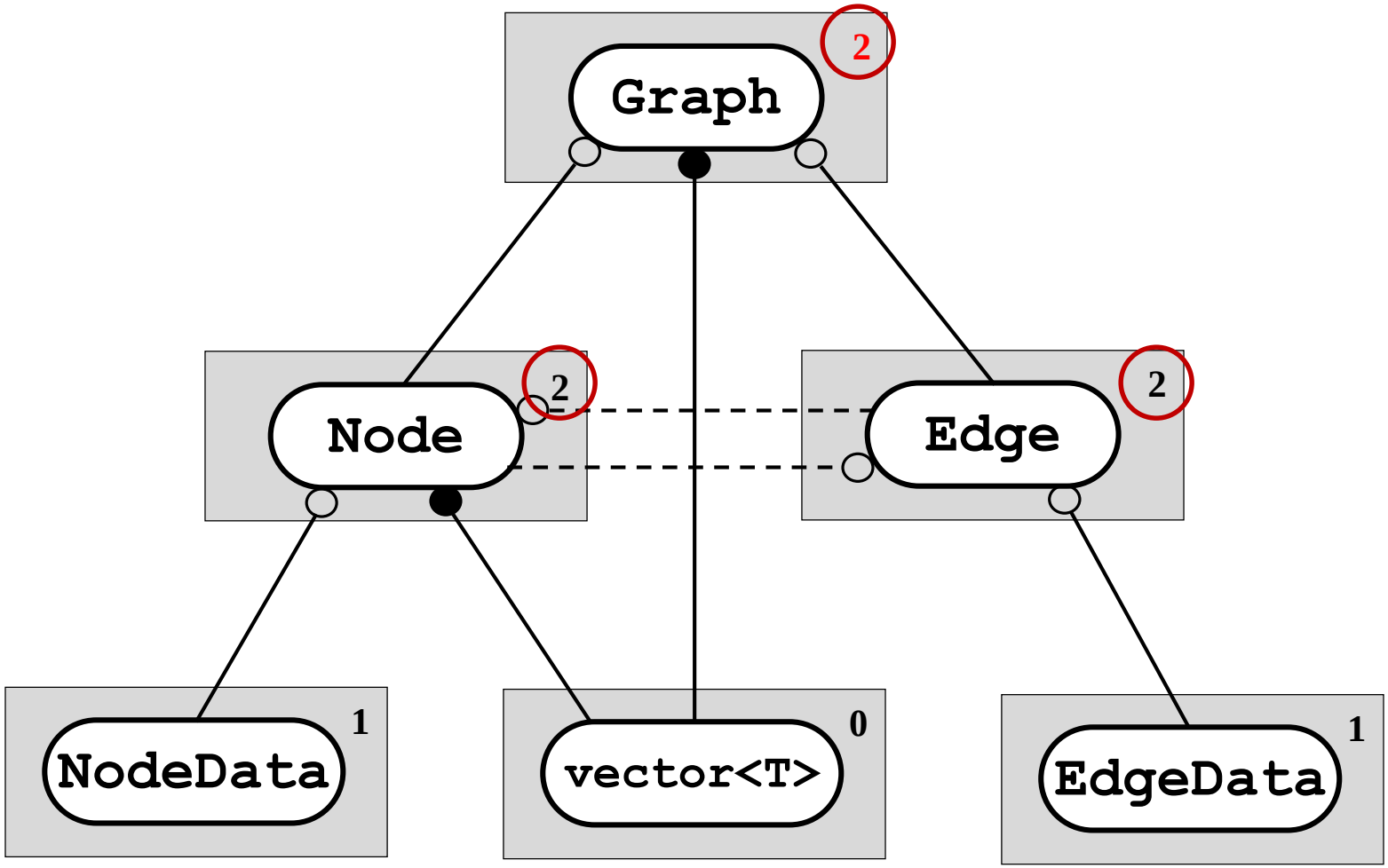
Escalating Encapsulation



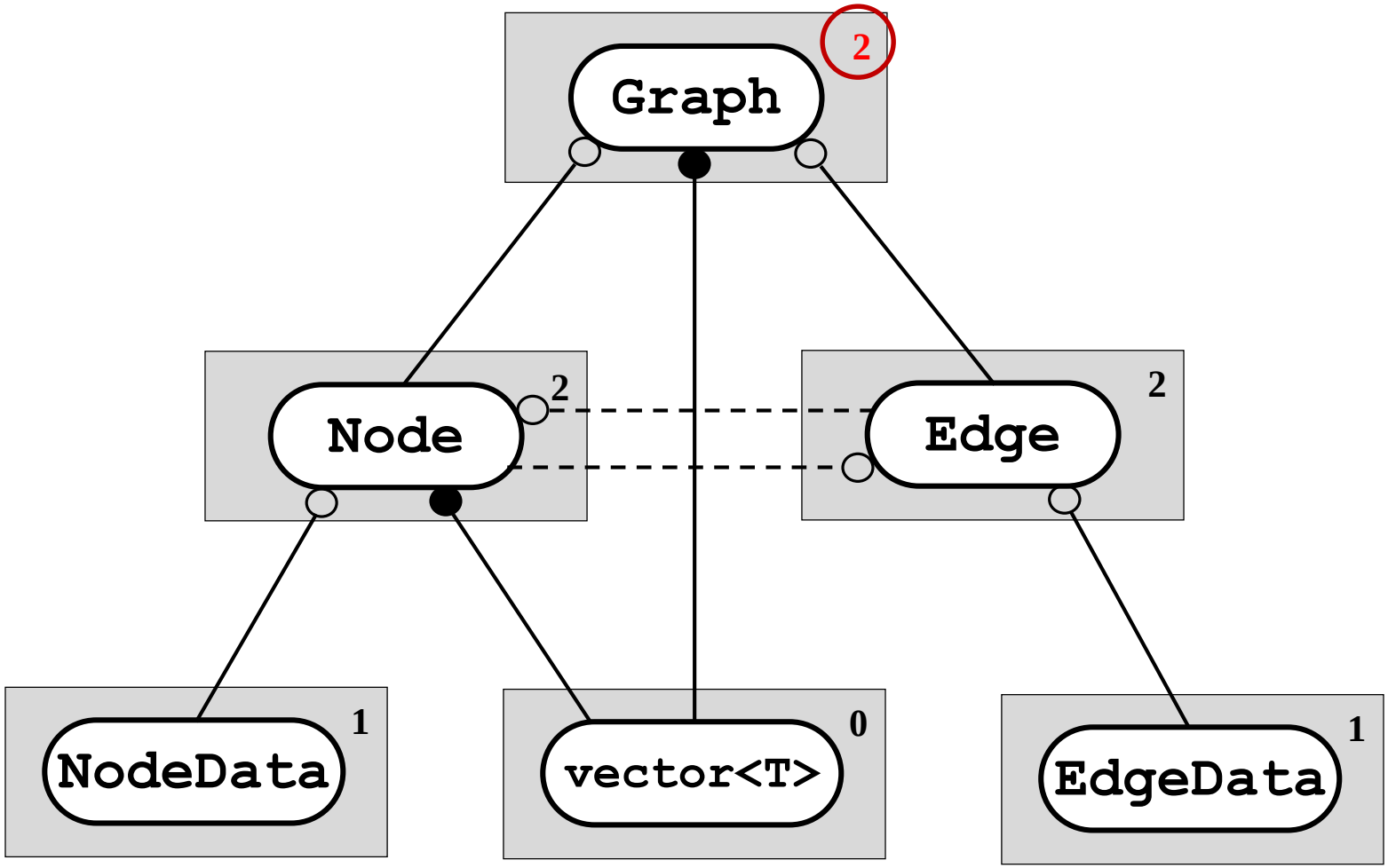
Escalating Encapsulation



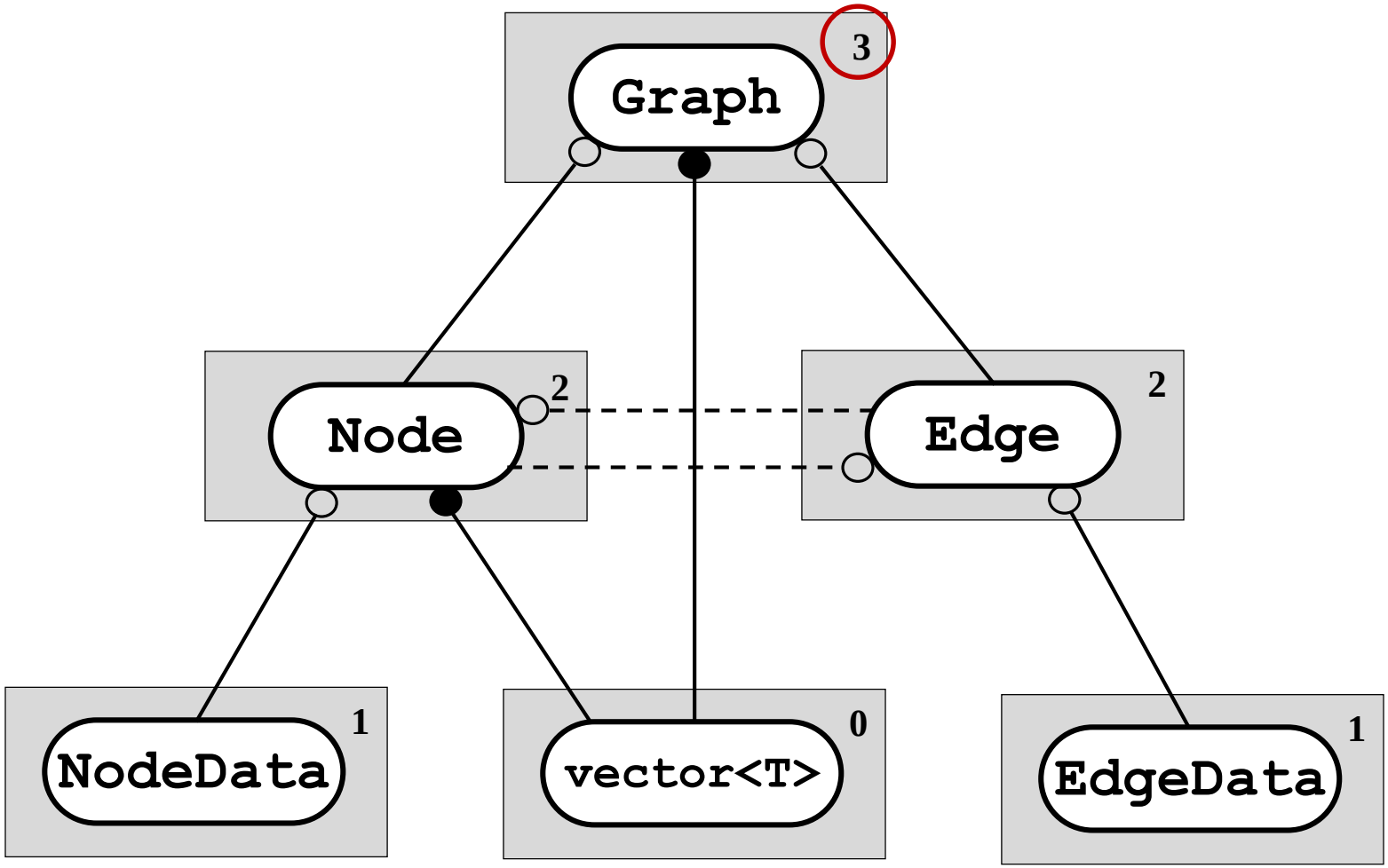
Escalating Encapsulation



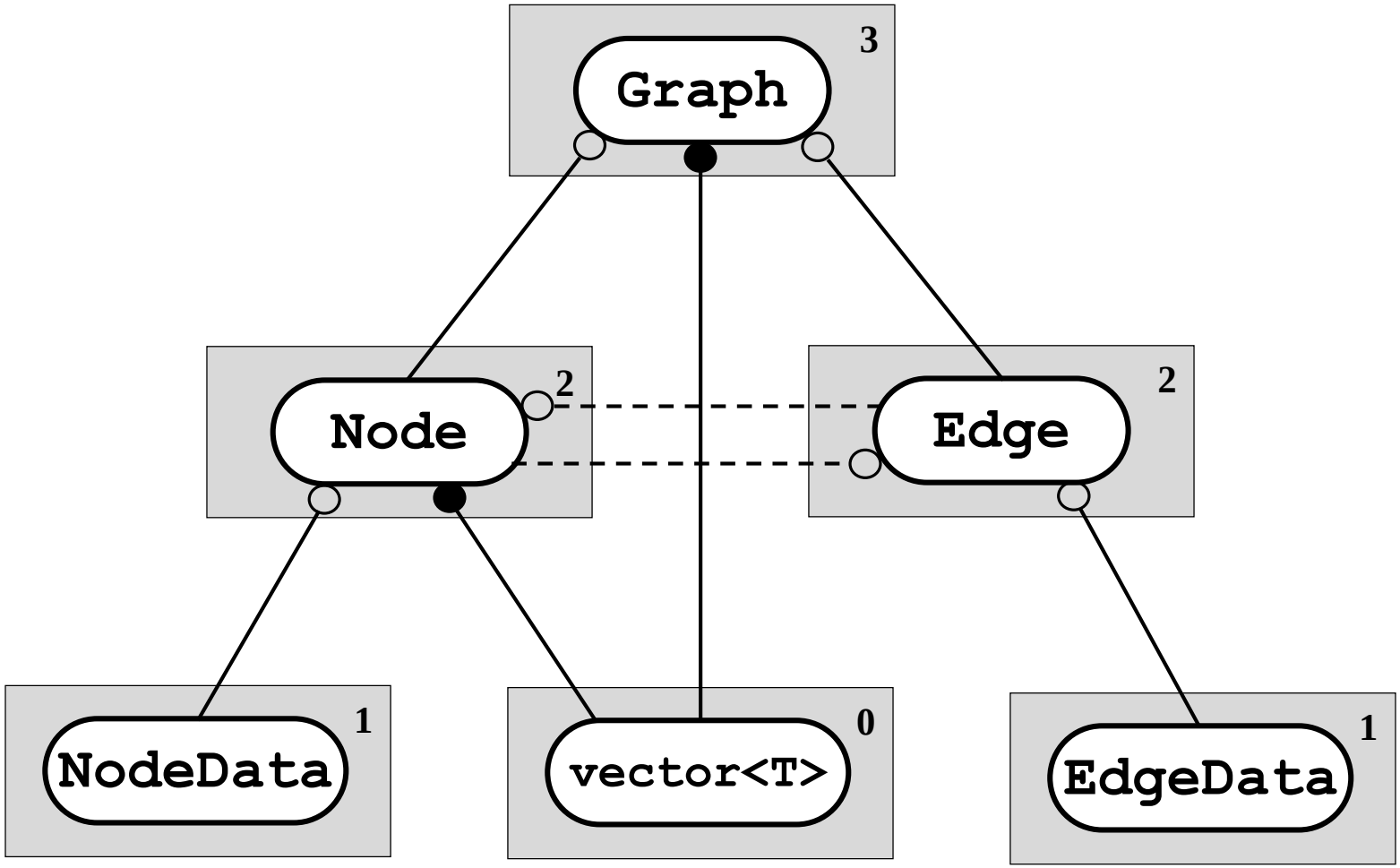
Escalating Encapsulation



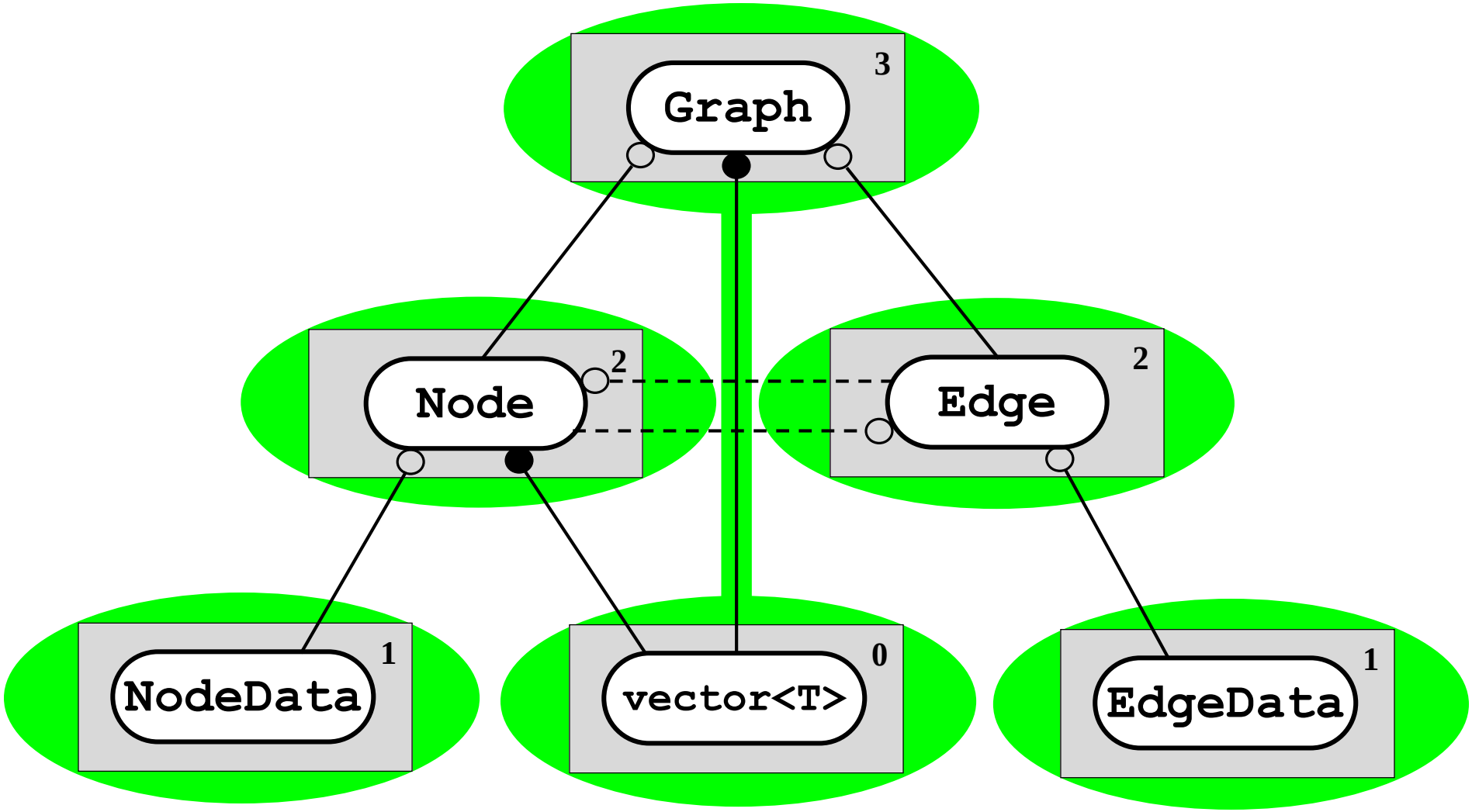
Escalating Encapsulation



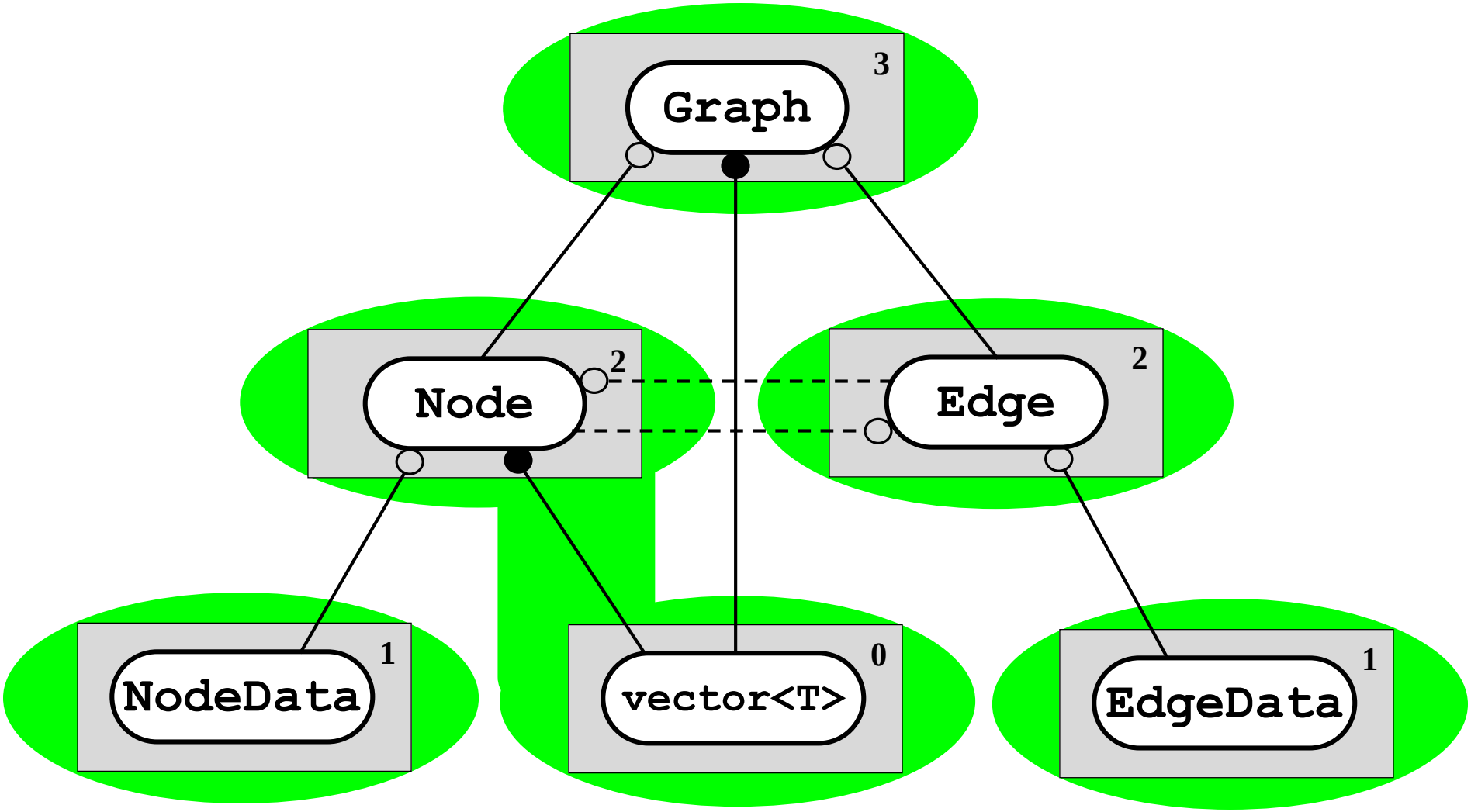
Escalating Encapsulation



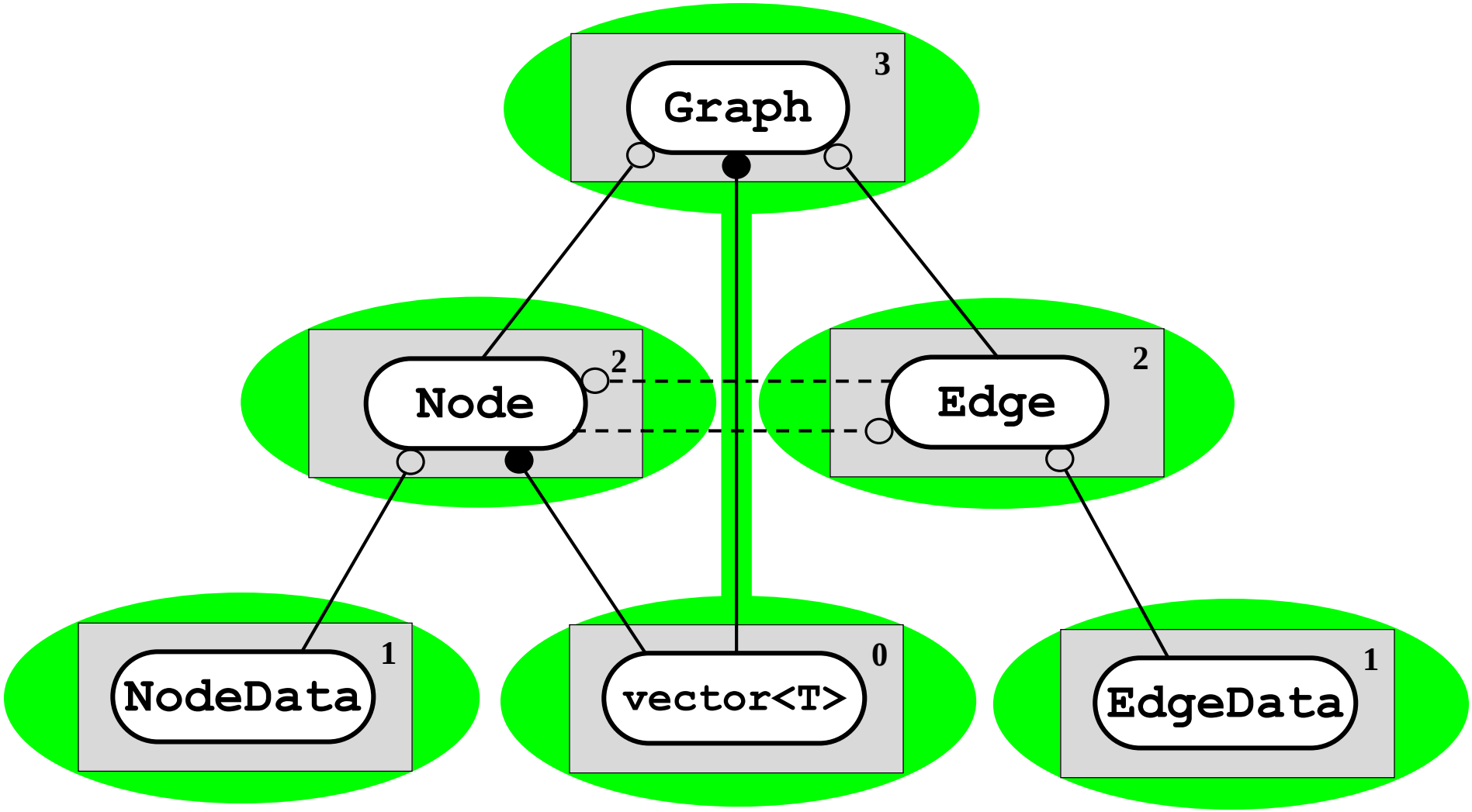
Escalating Encapsulation



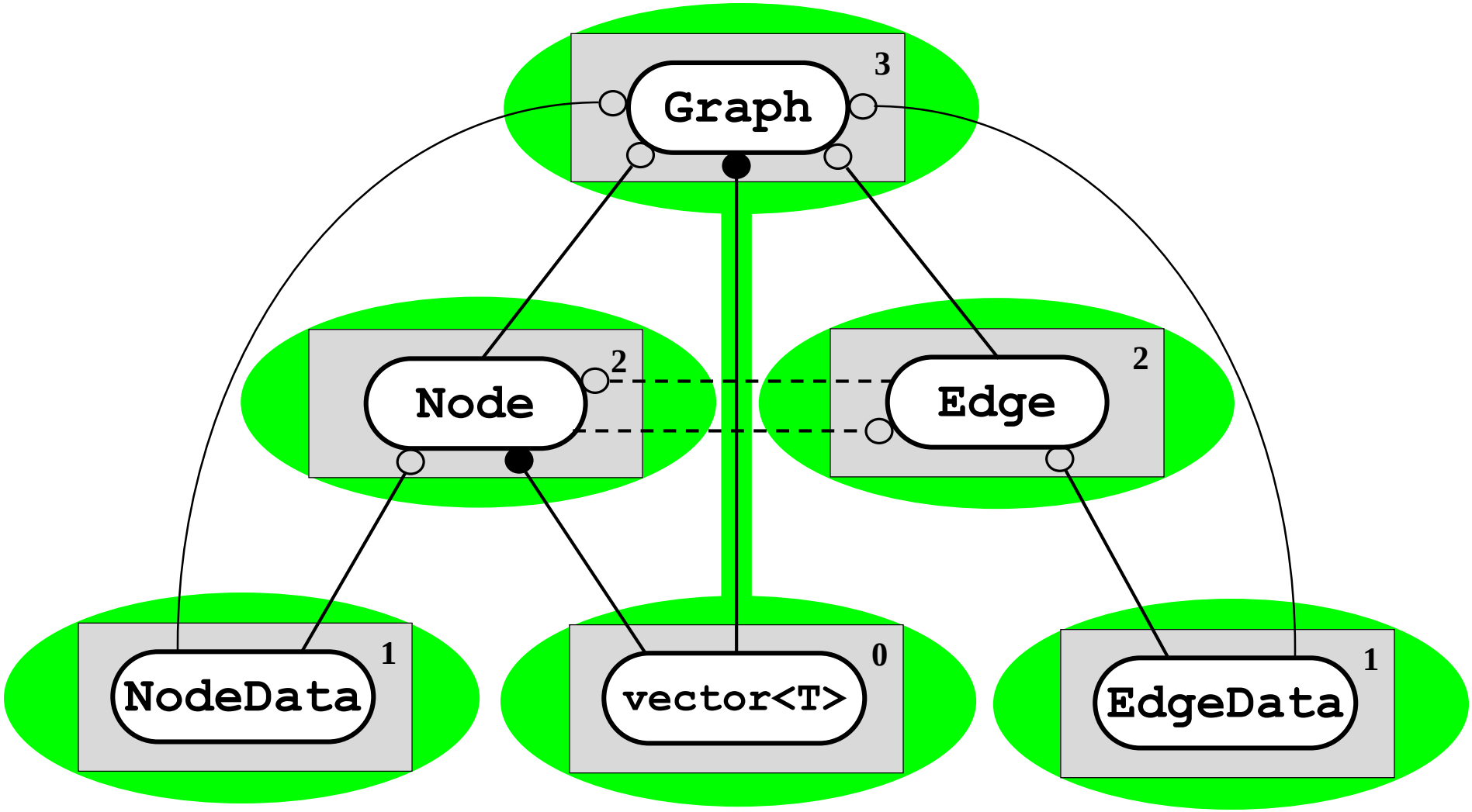
Escalating Encapsulation



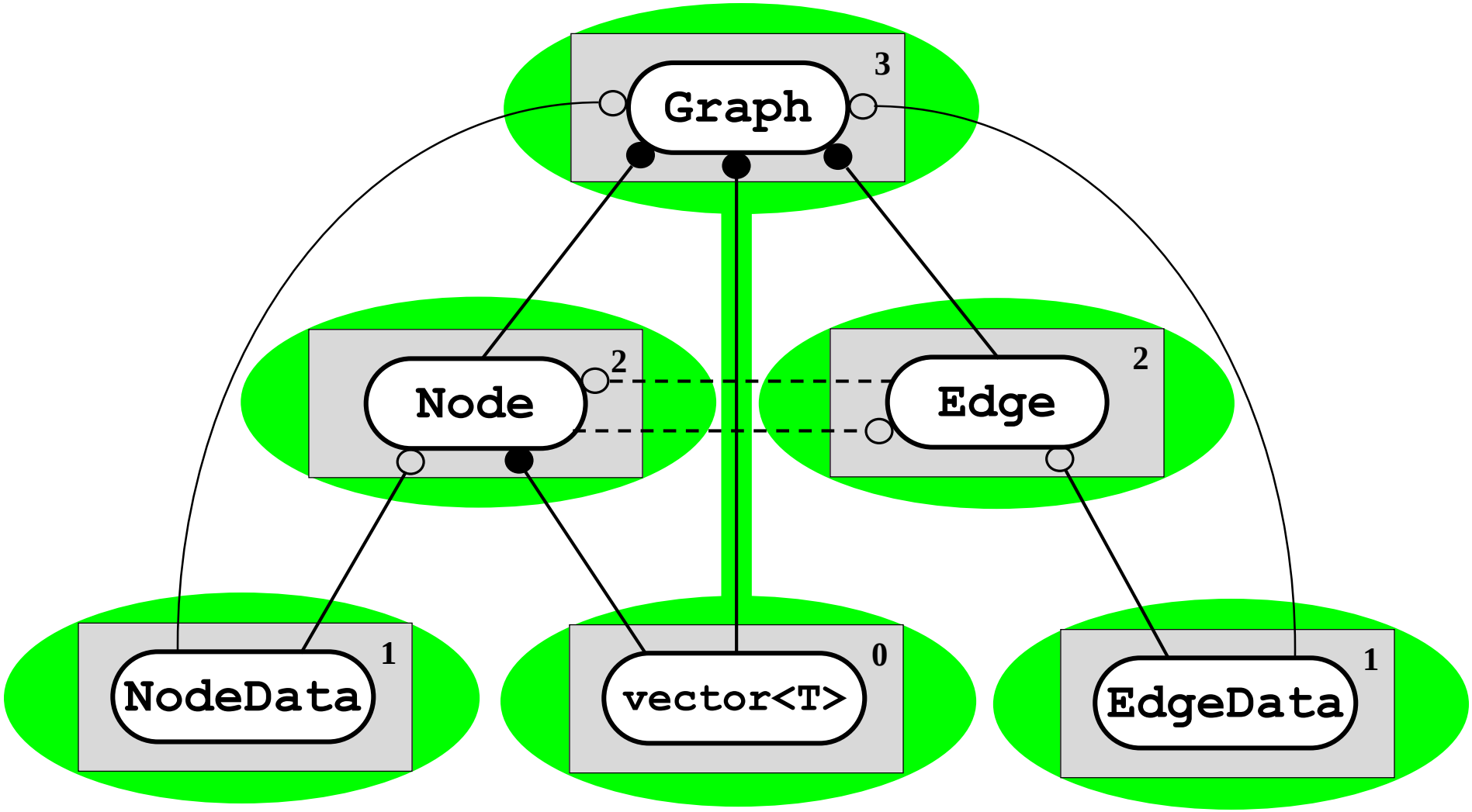
Escalating Encapsulation



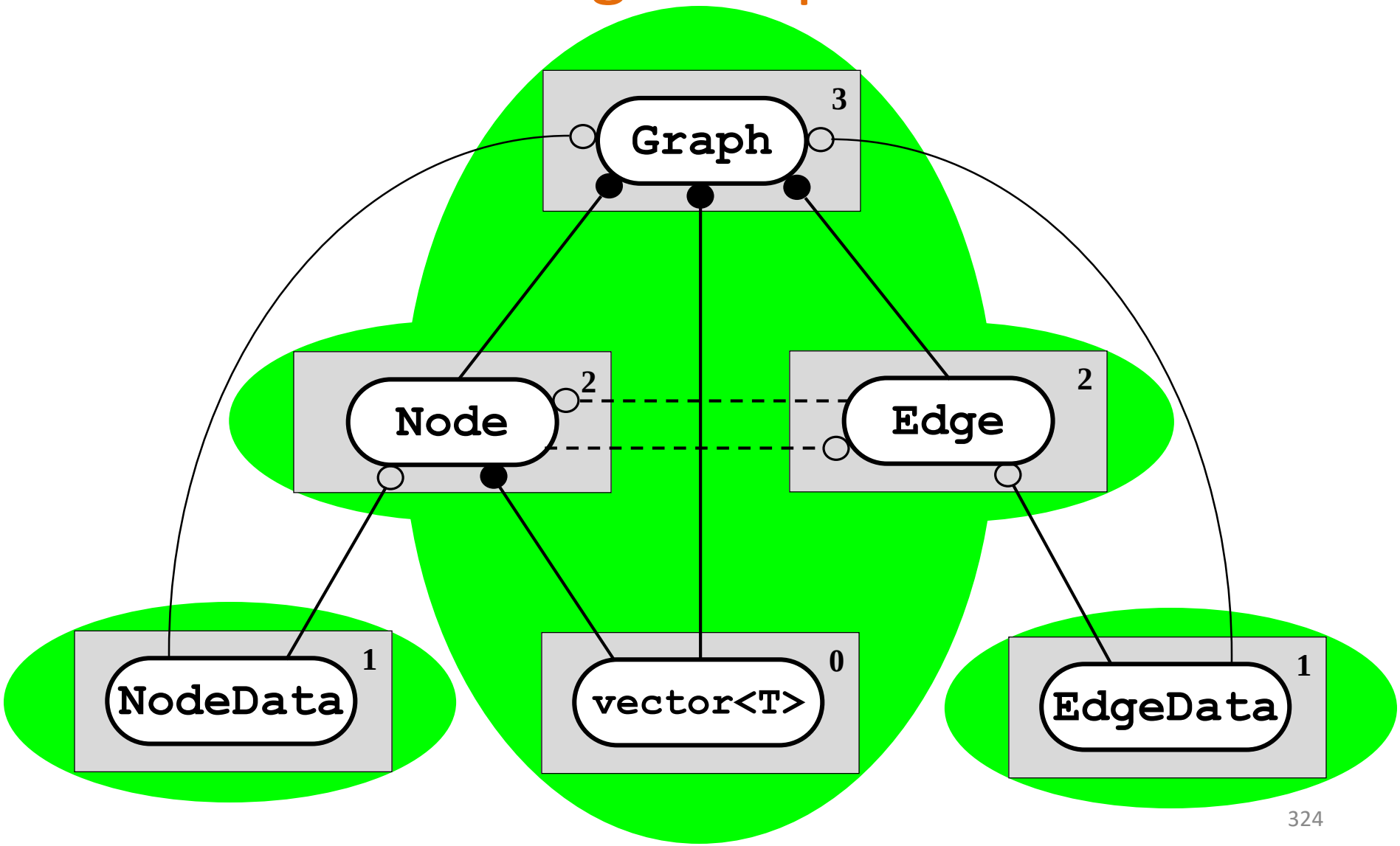
Escalating Encapsulation



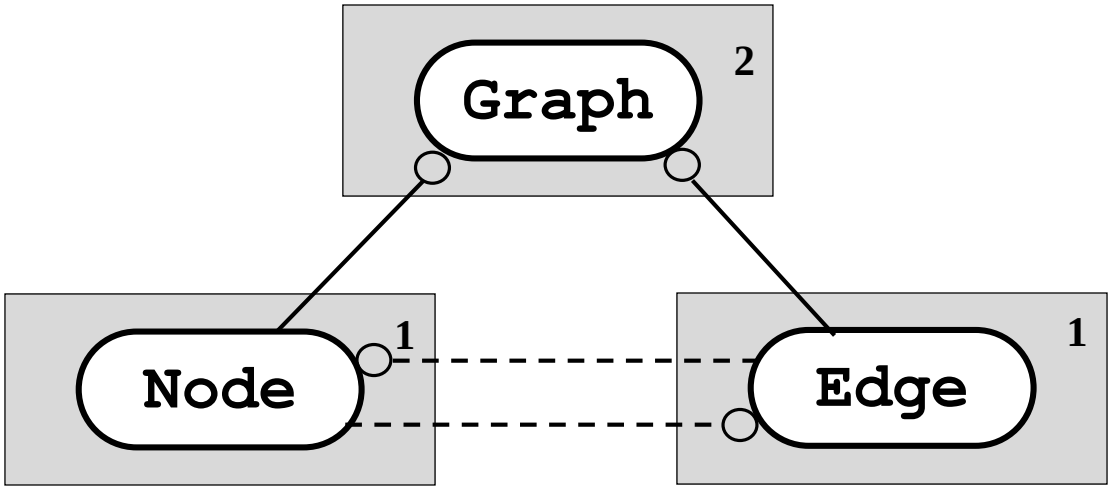
Escalating Encapsulation



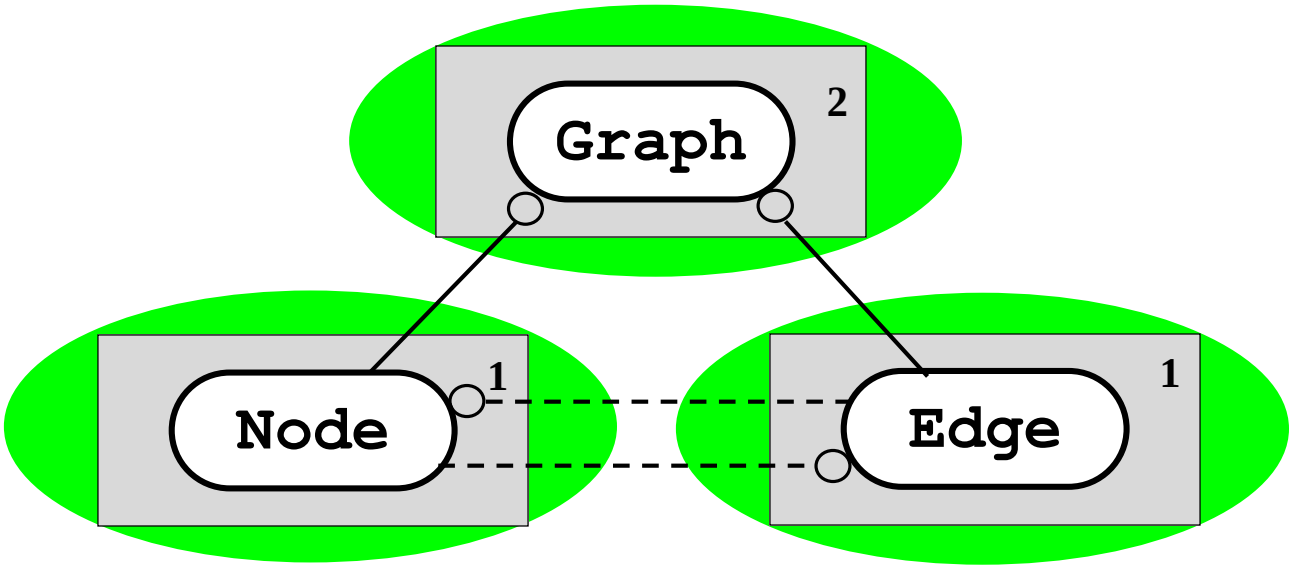
Escalating Encapsulation



Escalating Encapsulation

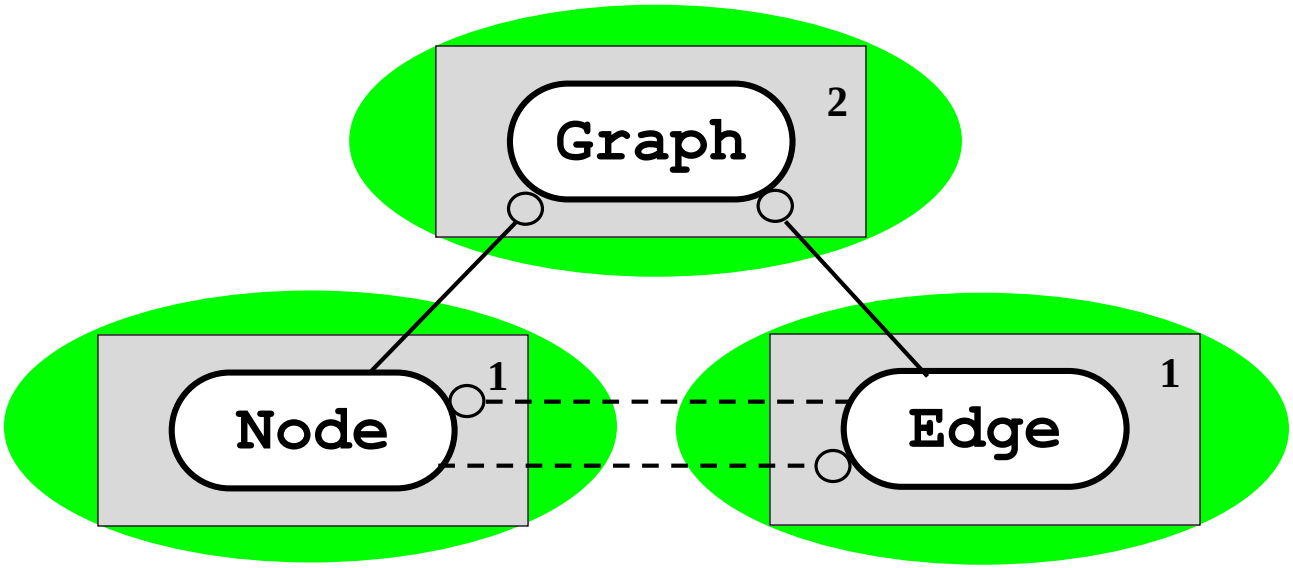


Escalating Encapsulation



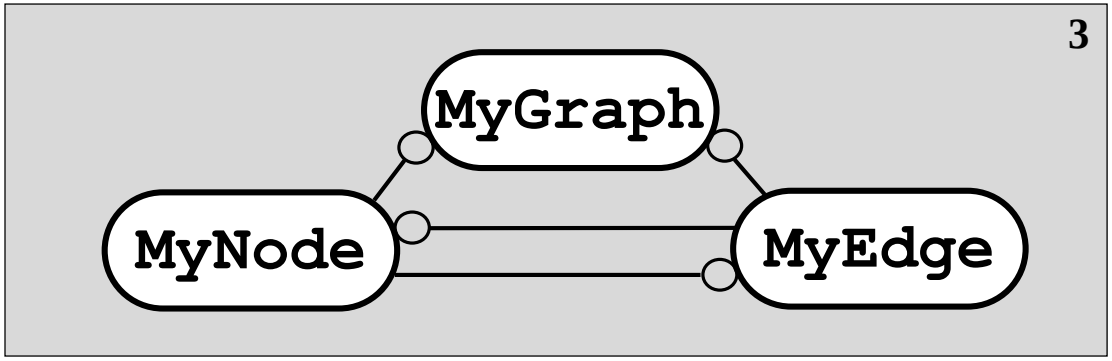
Escalating Encapsulation

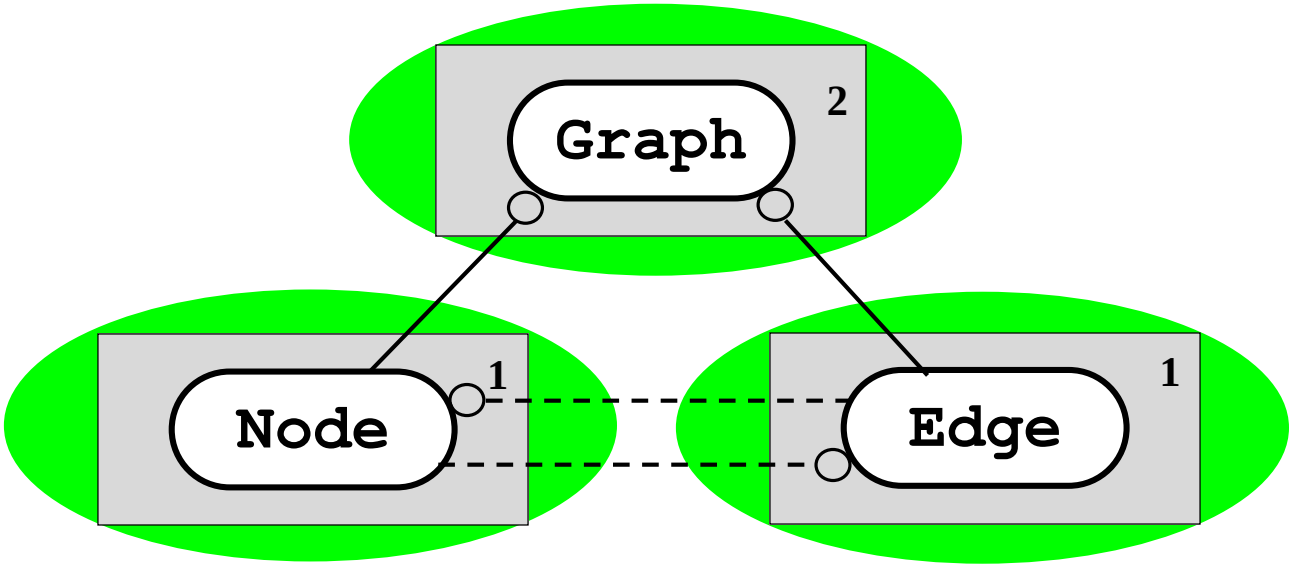
Single-Component Wrapper



Escalating Encapsulation

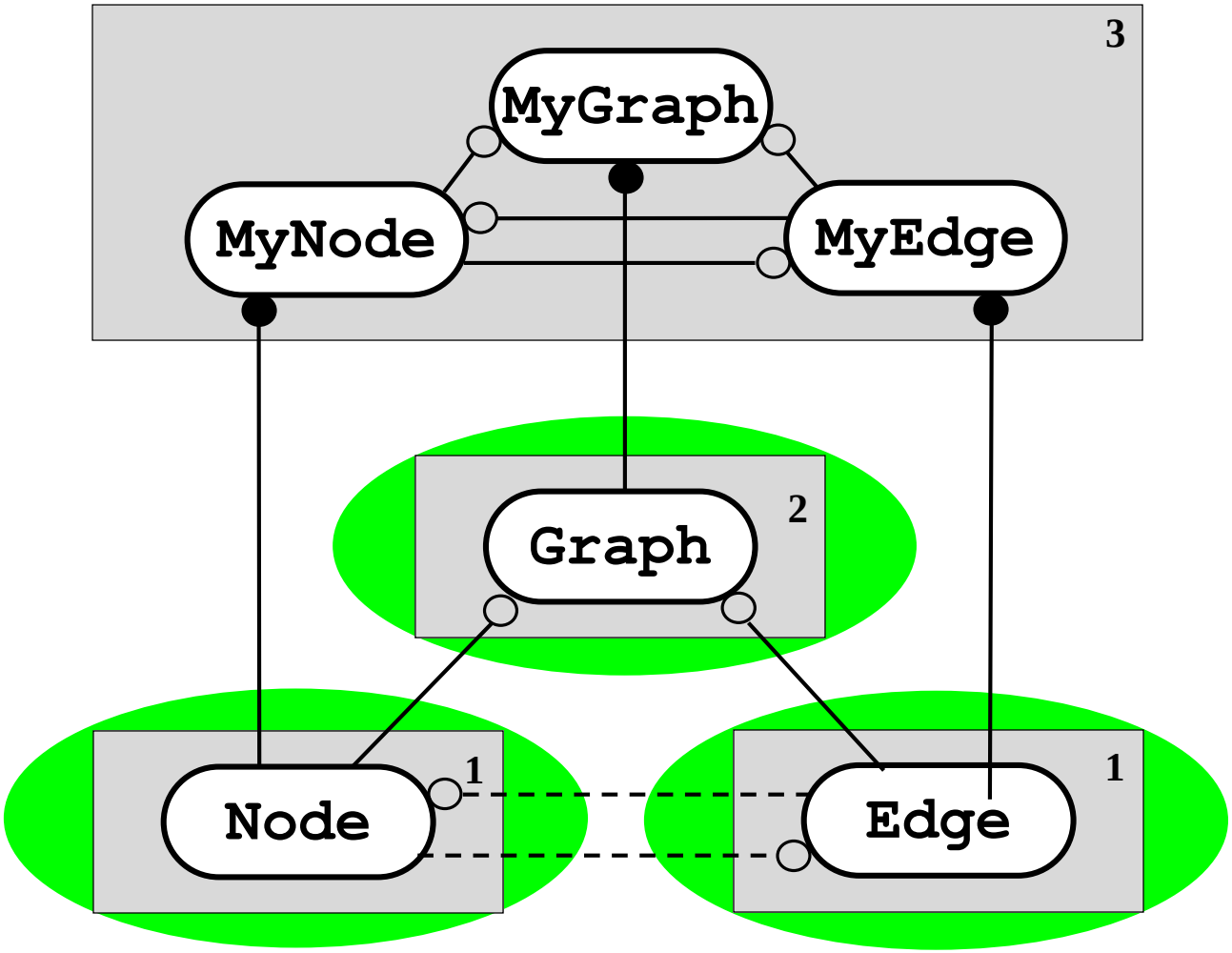
Single-Component Wrapper





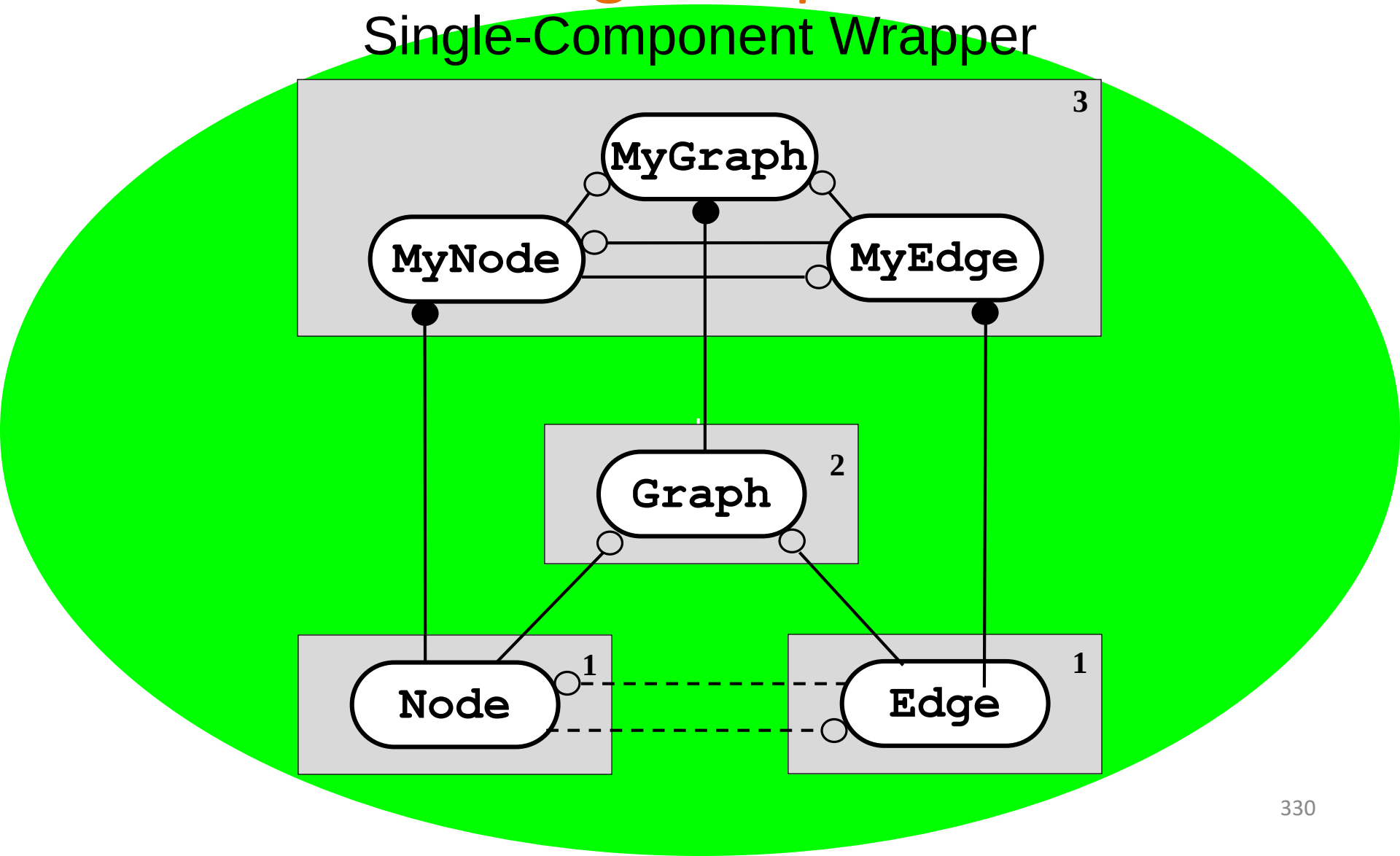
Escalating Encapsulation

Single-Component Wrapper



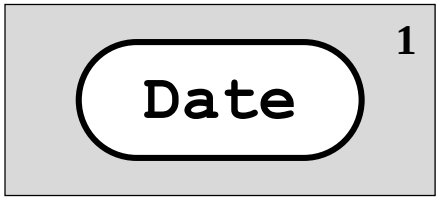
Escalating Encapsulation

Single-Component Wrapper



Escalating Encapsulation

Multi-Component Wrapper

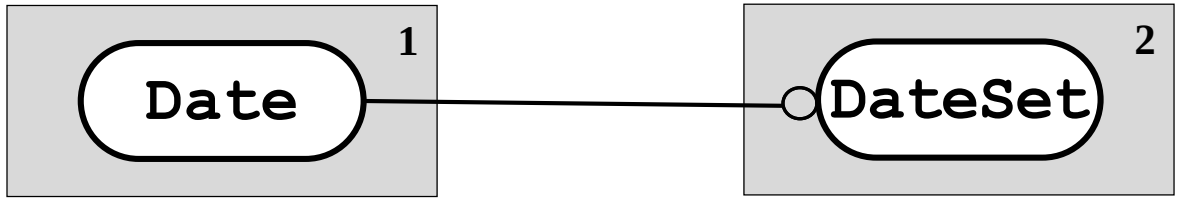


Escalating Encapsulation

Multi-Component Wrapper

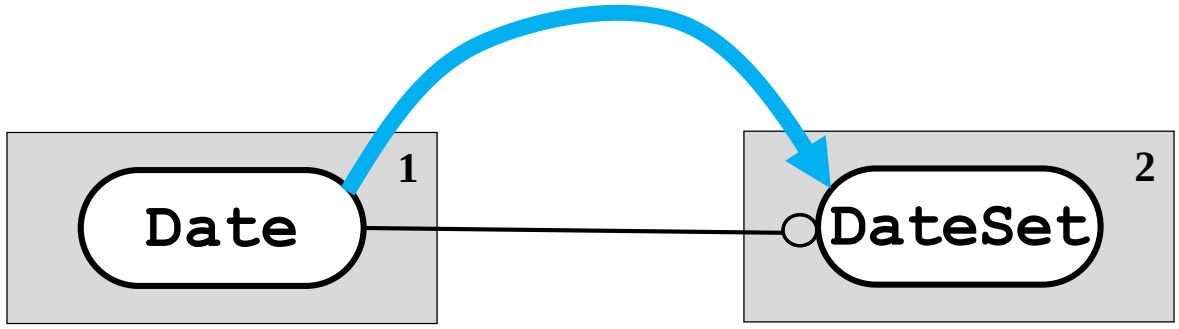
Escalating Encapsulation

Multi-Component Wrapper



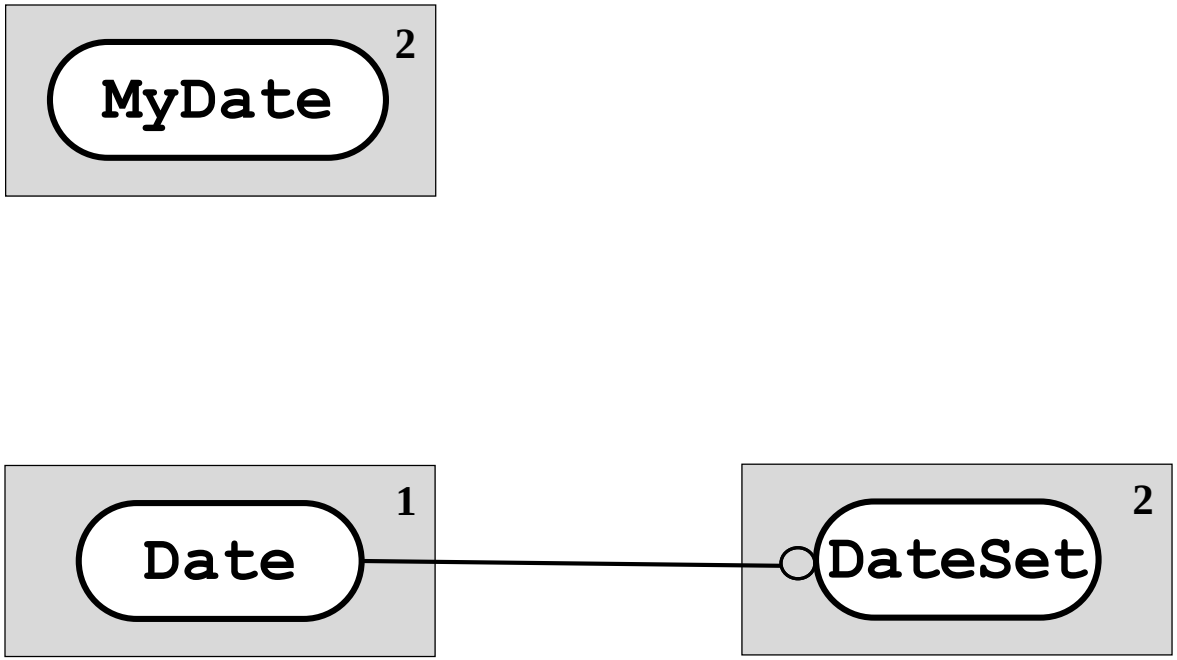
Escalating Encapsulation

Multi-Component Wrapper



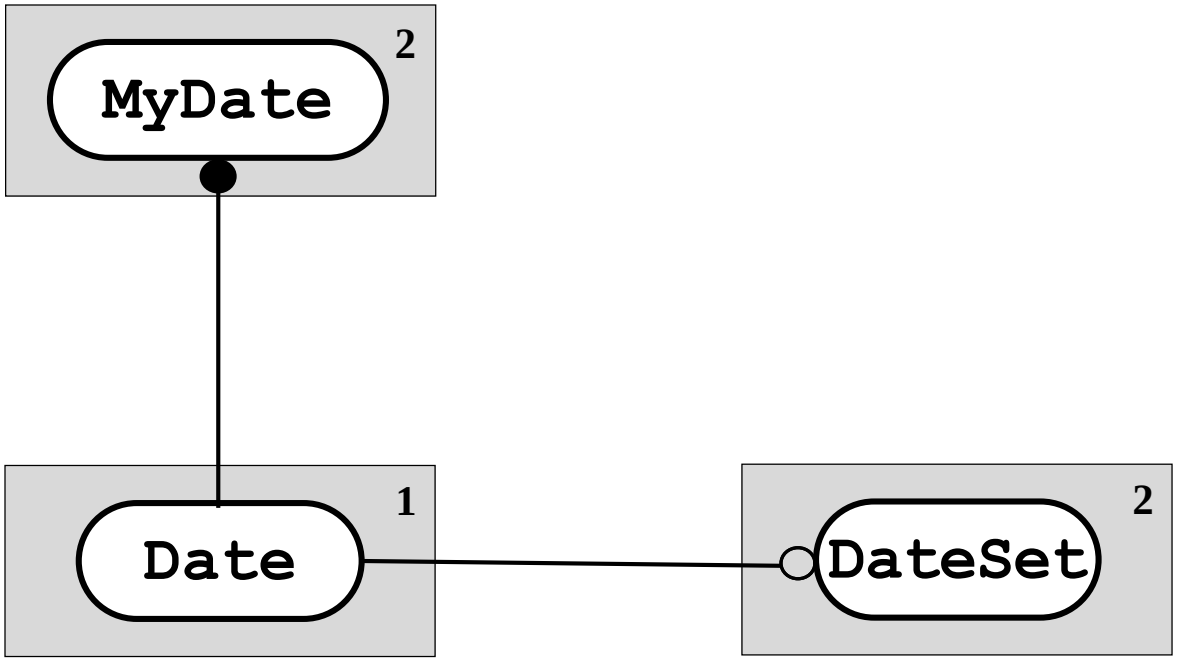
Escalating Encapsulation

Multi-Component Wrapper



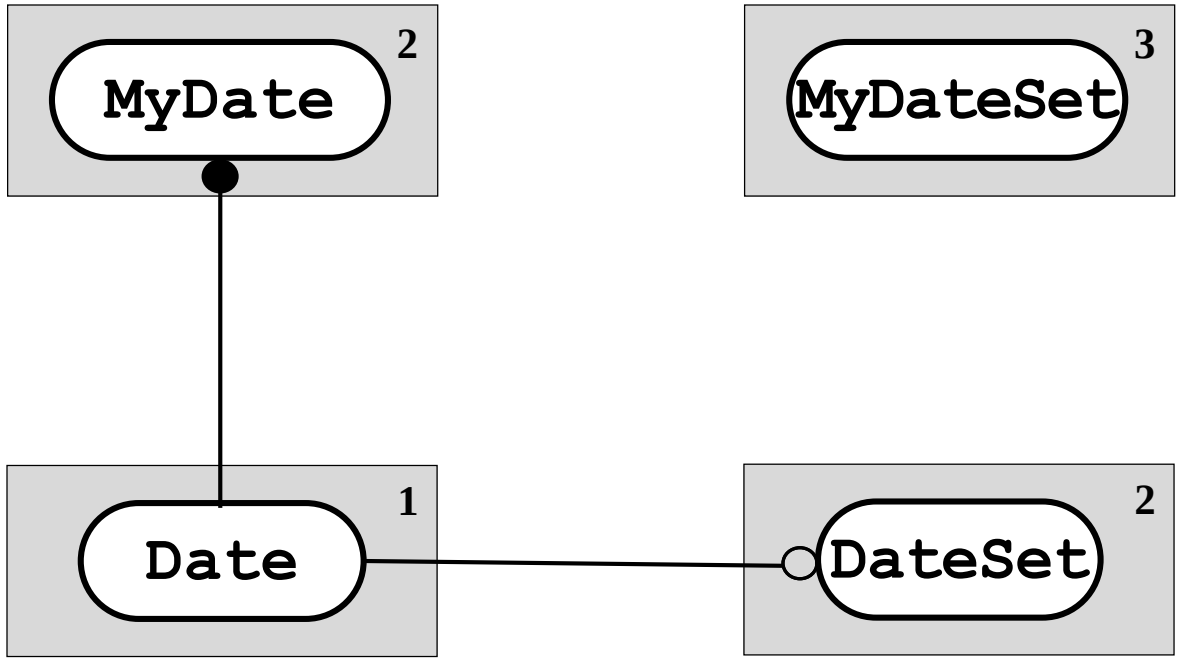
Escalating Encapsulation

Multi-Component Wrapper



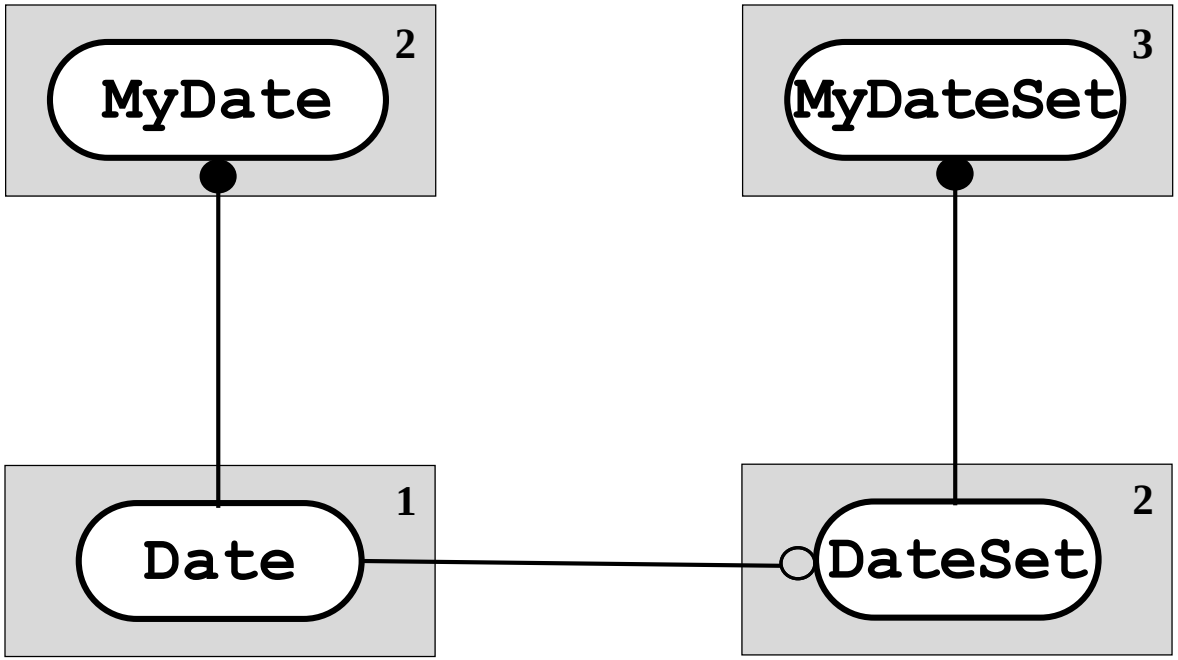
Escalating Encapsulation

Multi-Component Wrapper



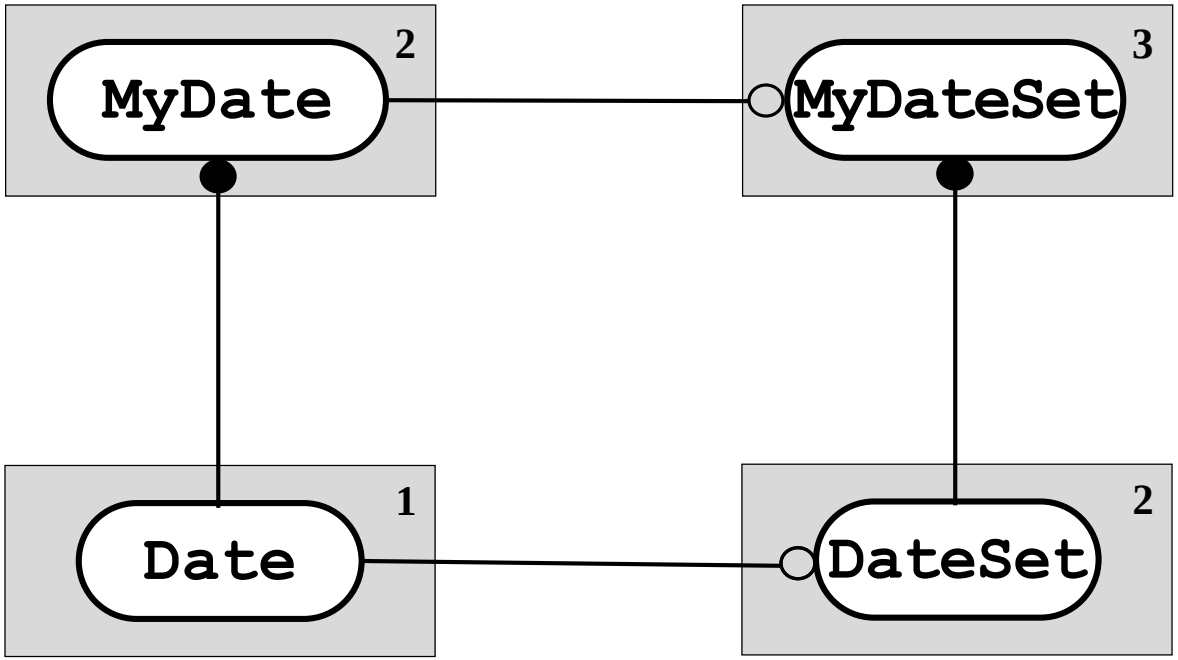
Escalating Encapsulation

Multi-Component Wrapper



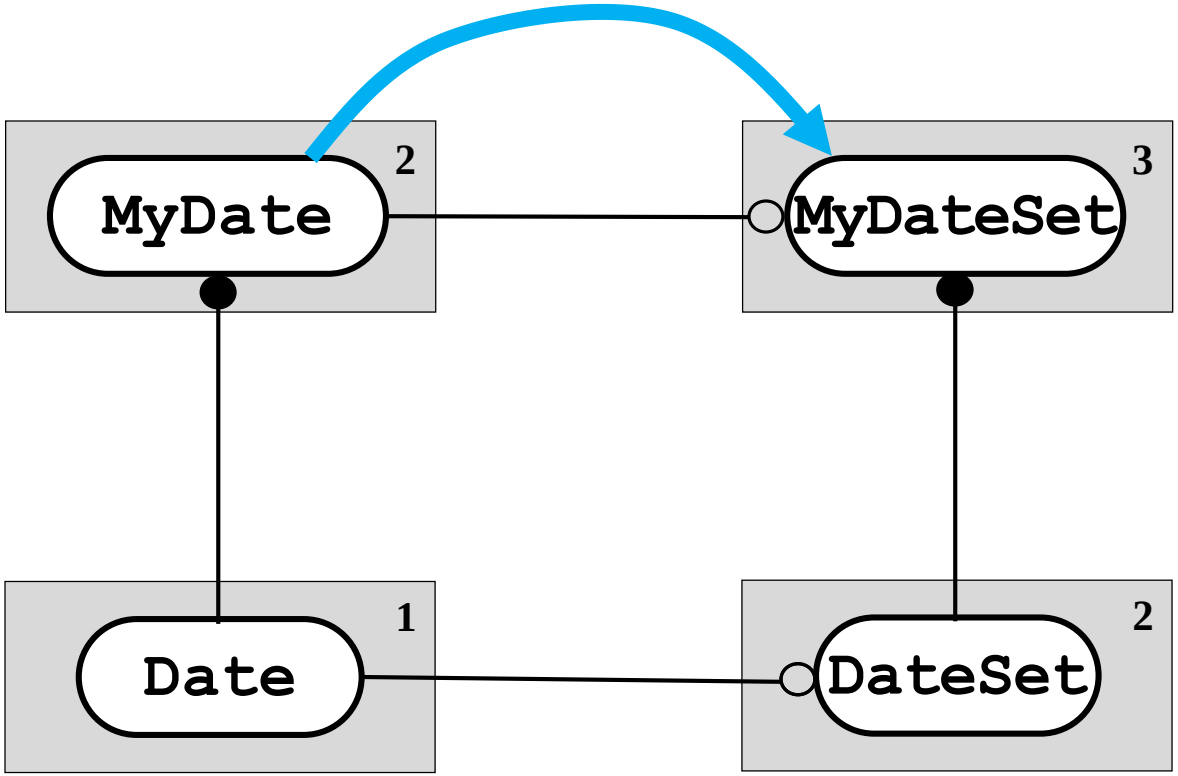
Escalating Encapsulation

Multi-Component Wrapper



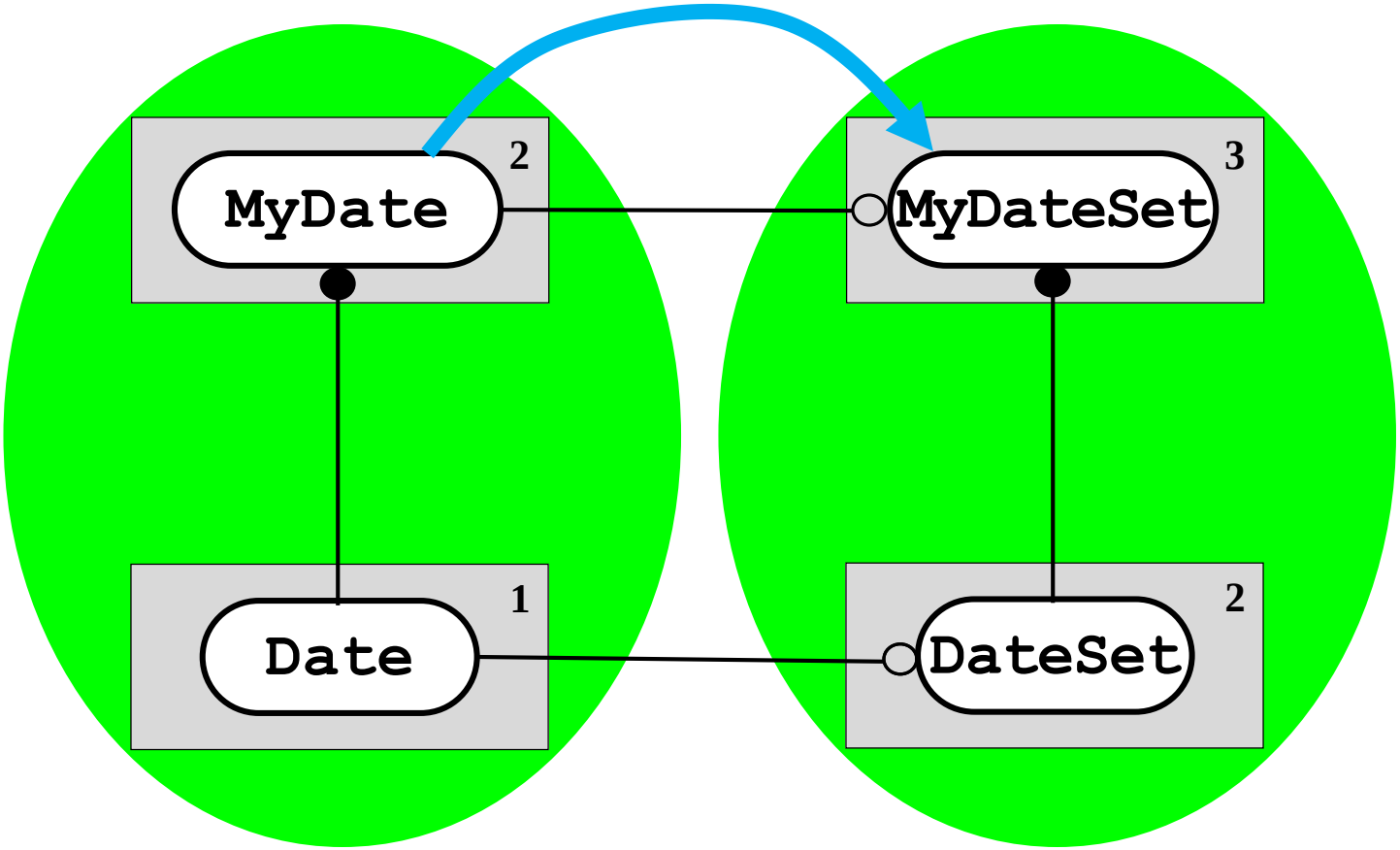
Escalating Encapsulation

Multi-Component Wrapper



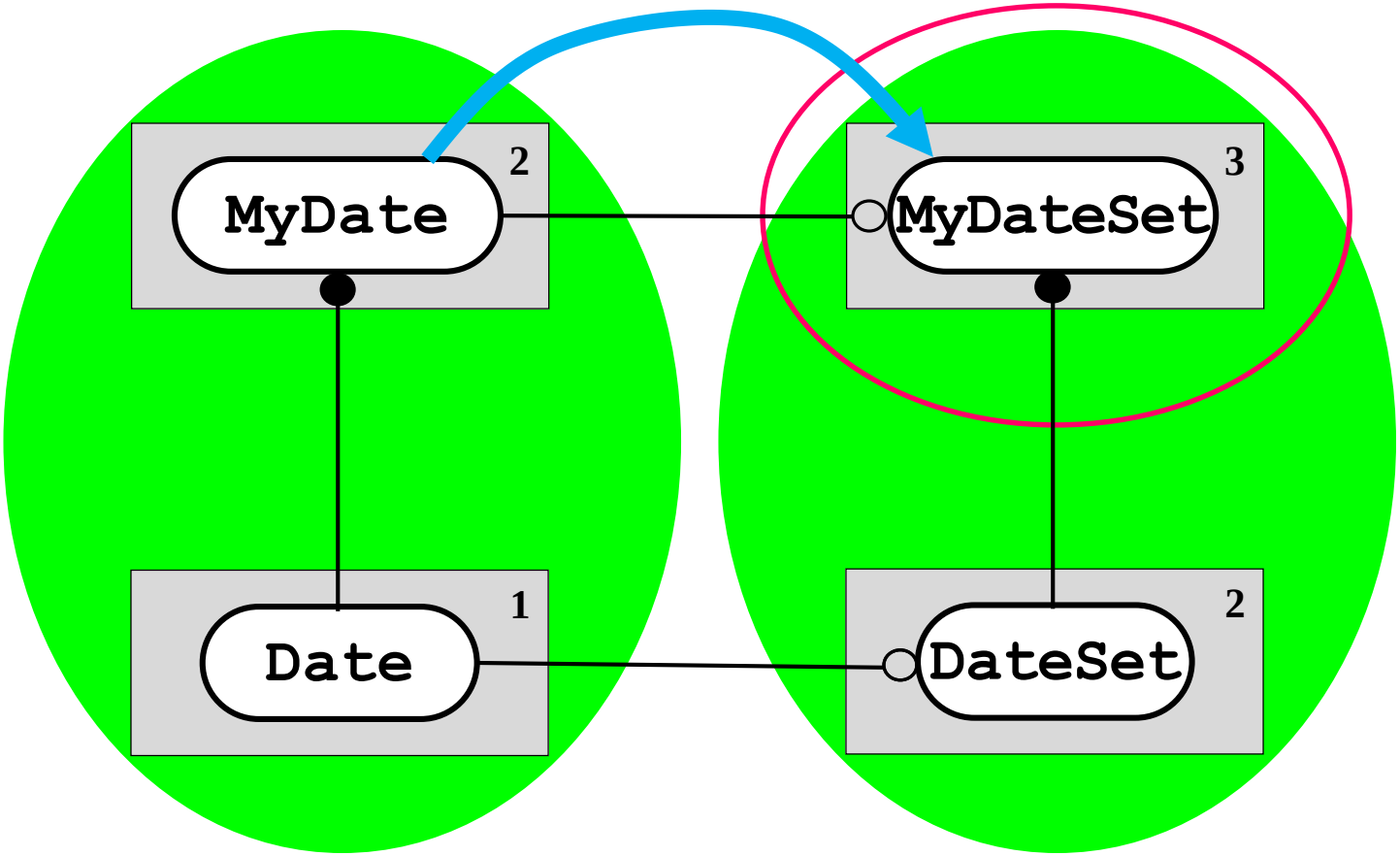
Escalating Encapsulation

Multi-Component Wrapper



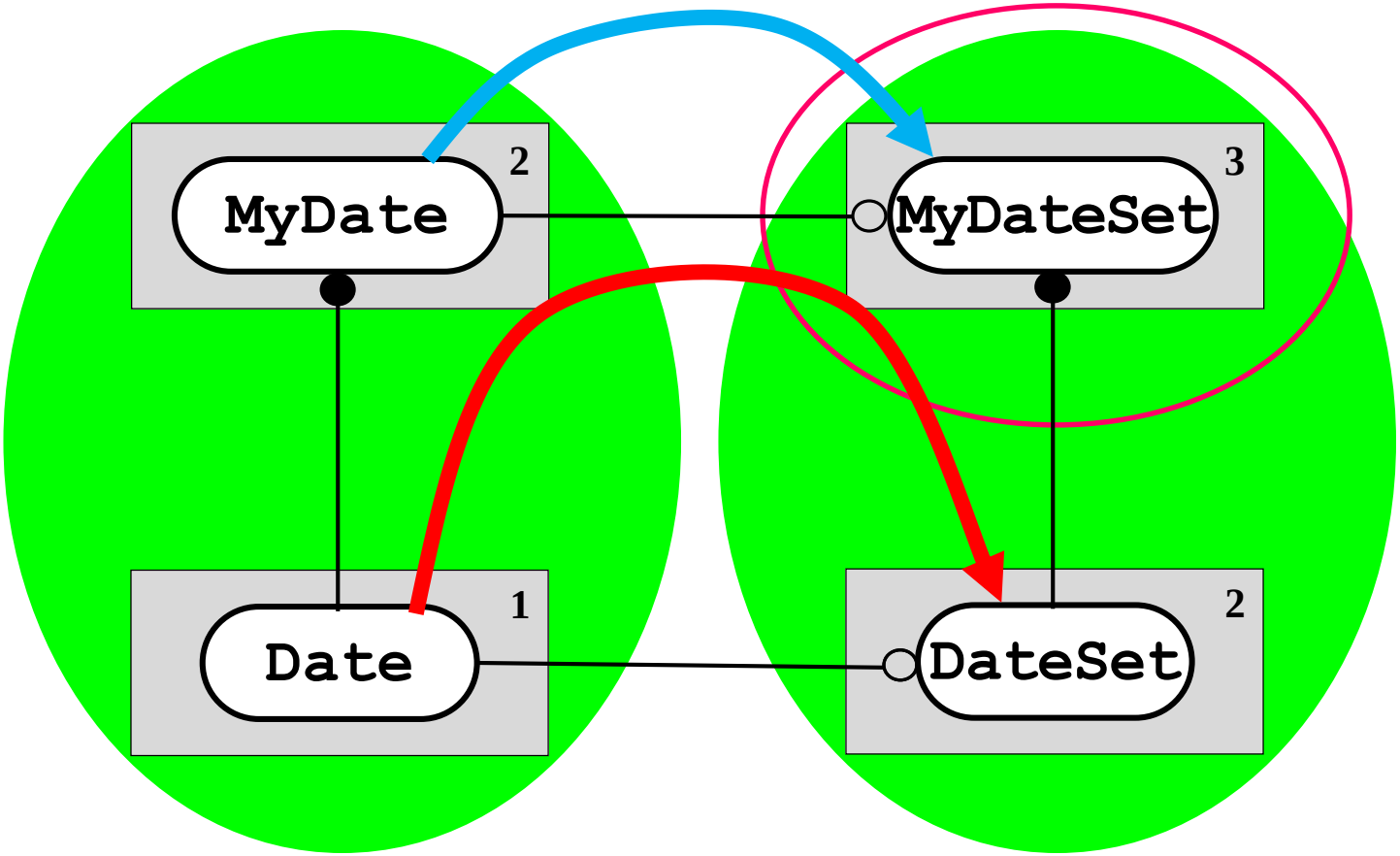
Escalating Encapsulation

Multi-Component Wrapper



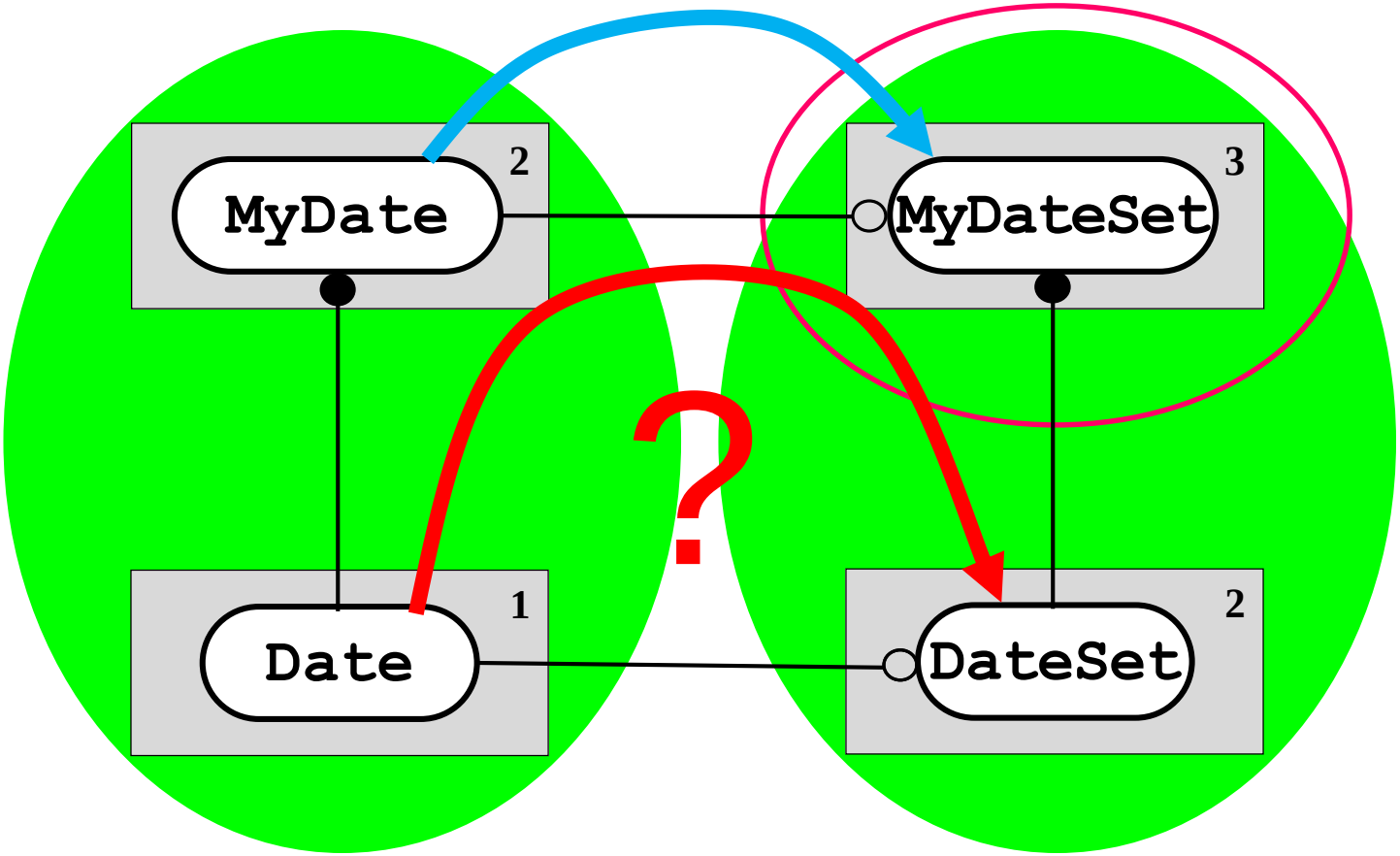
Escalating Encapsulation

Multi-Component Wrapper



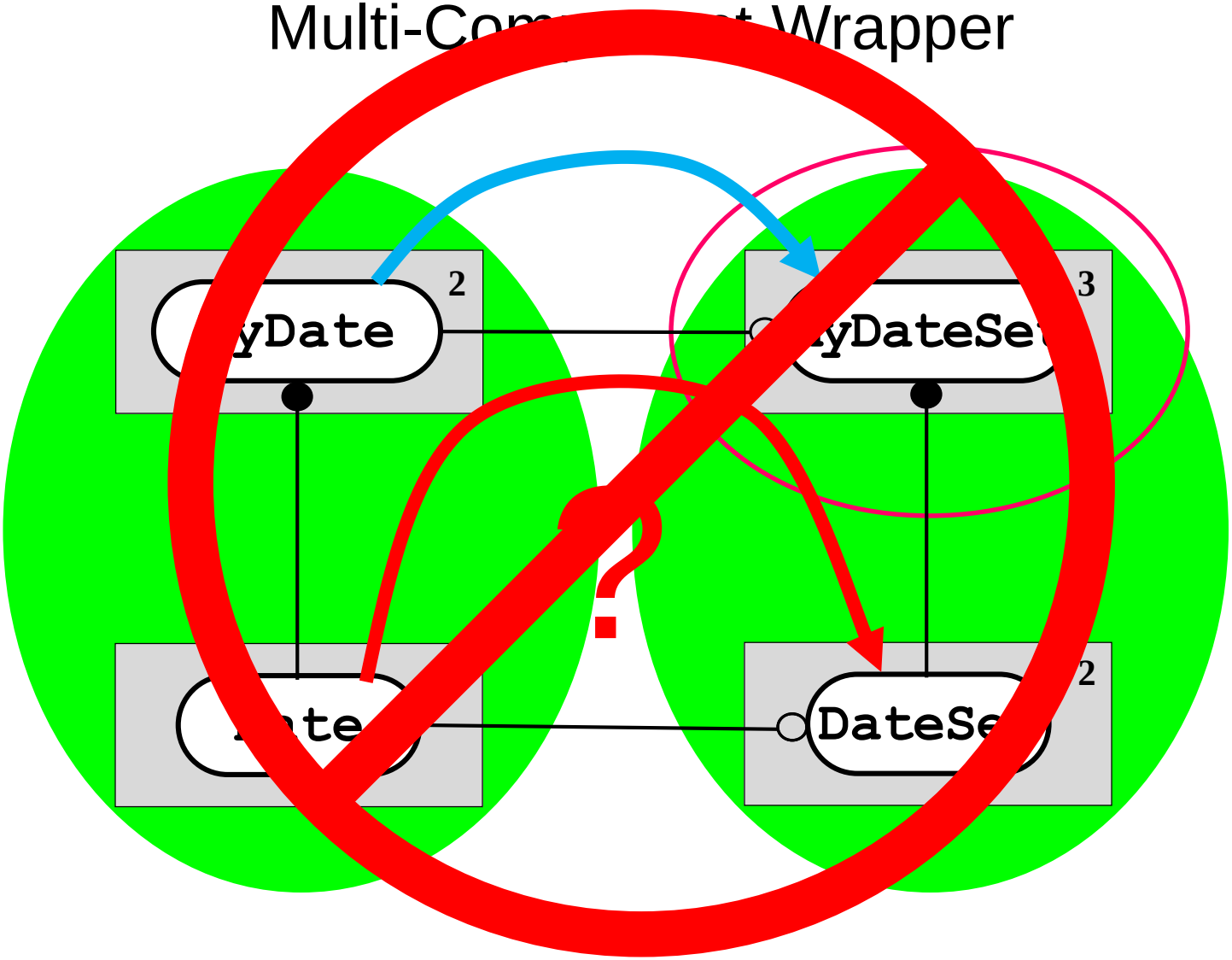
Escalating Encapsulation

Multi-Component Wrapper



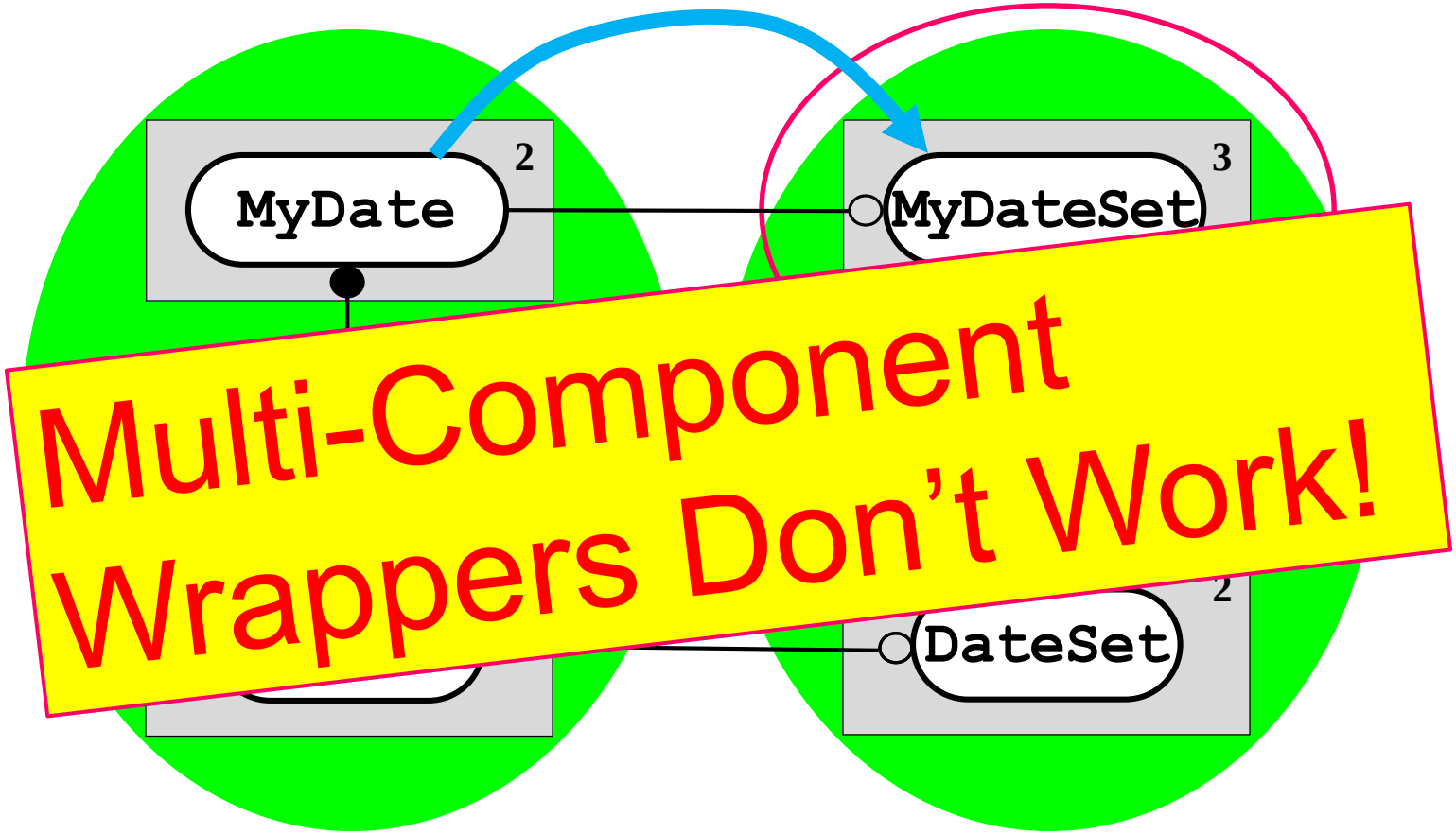
Escalating Encapsulation

Multi-Com Wrapper



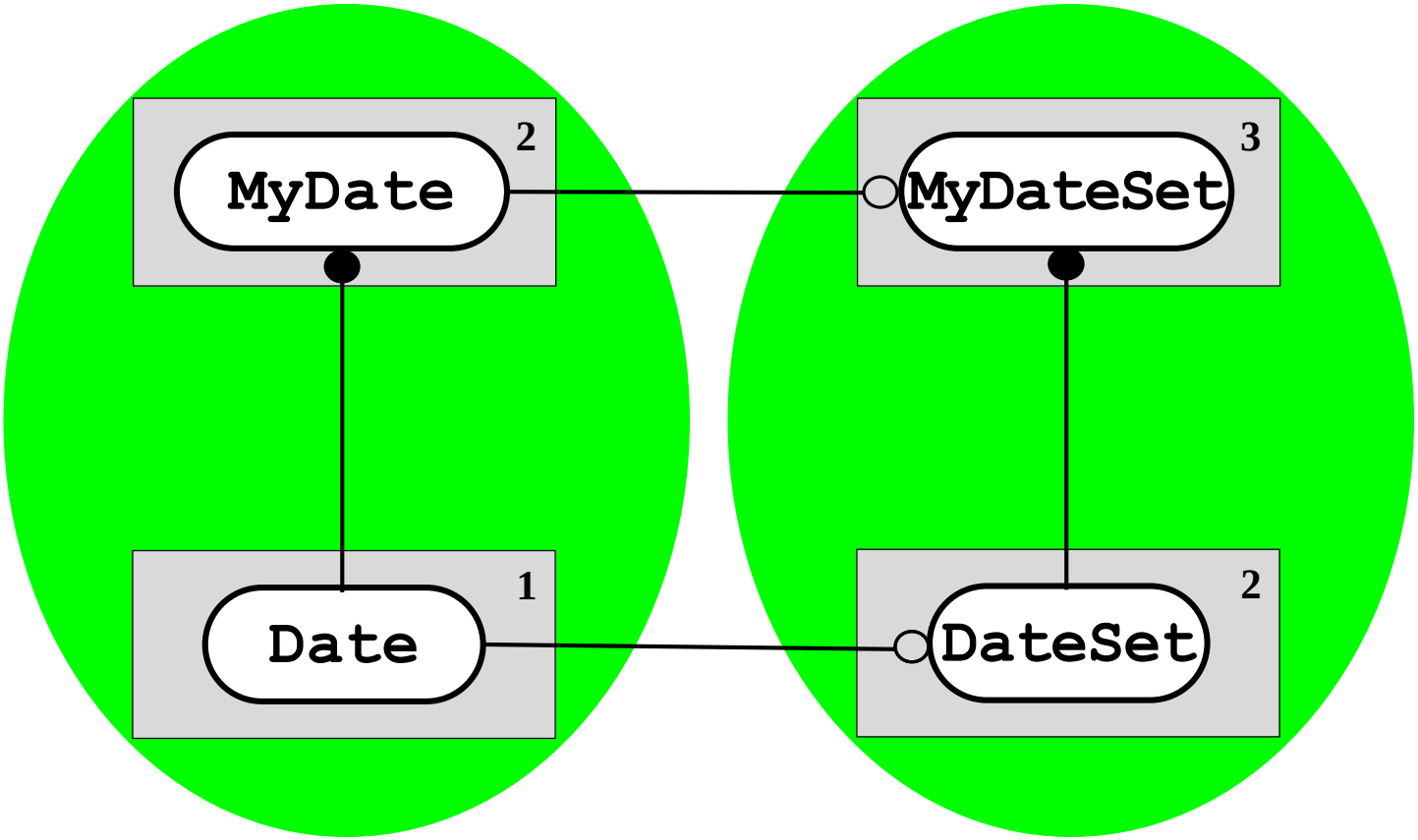
Escalating Encapsulation

Multi-Component Wrapper



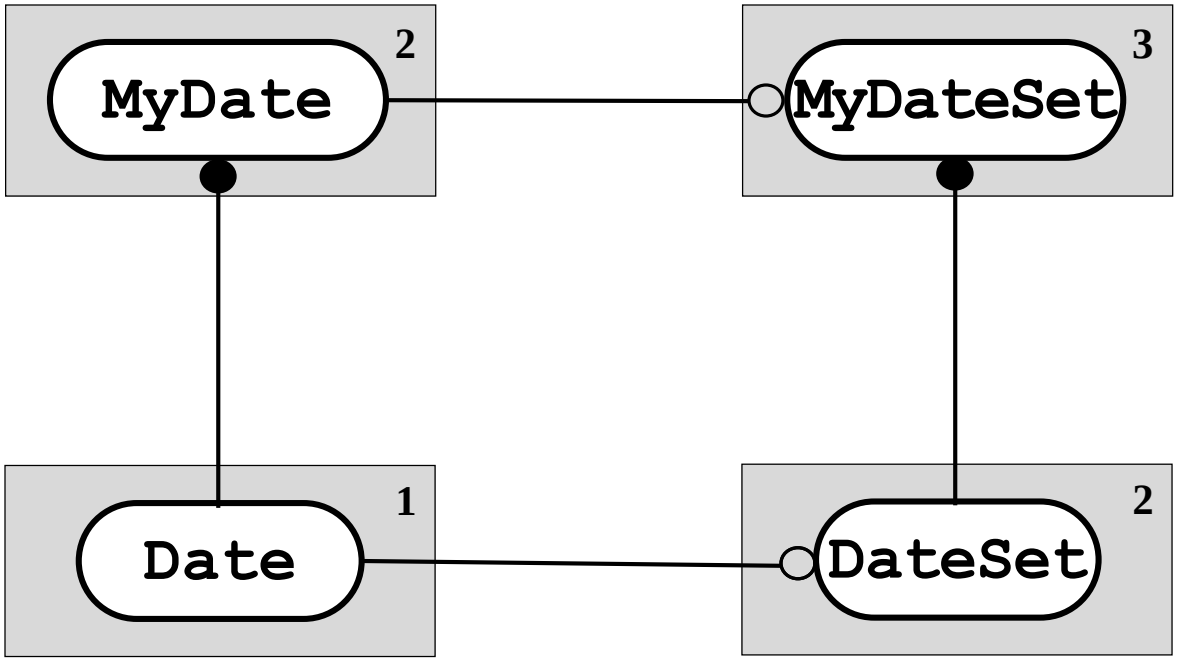
Escalating Encapsulation

Multi-Component Wrapper



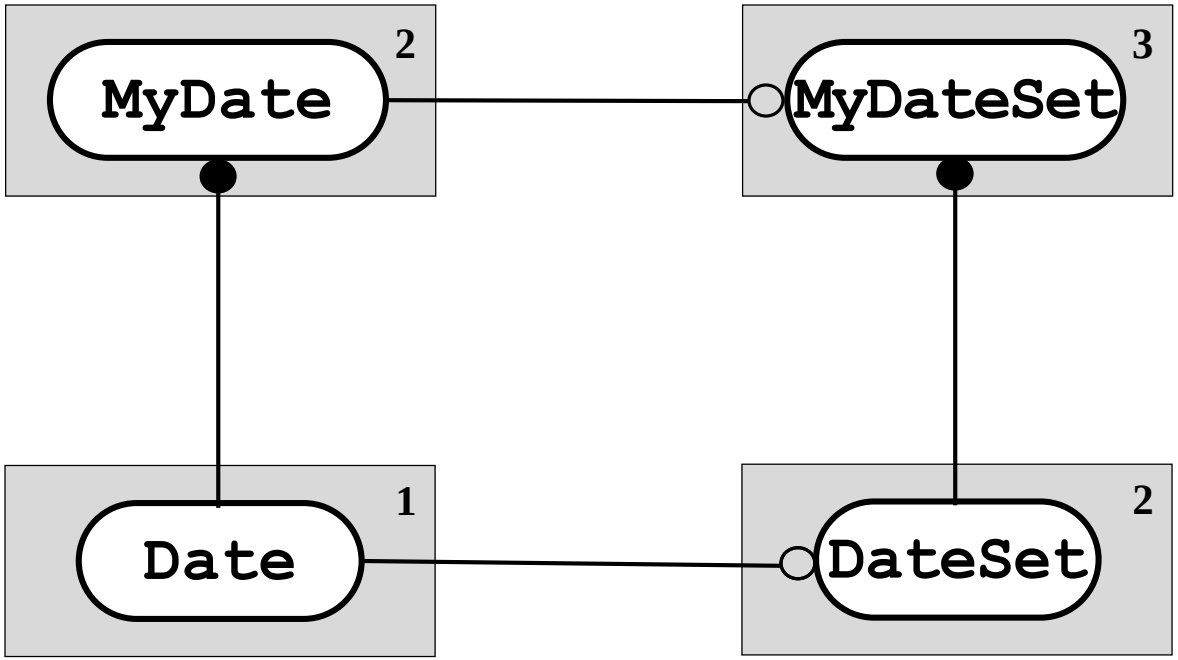
Escalating Encapsulation

Multi-Component Wrapper



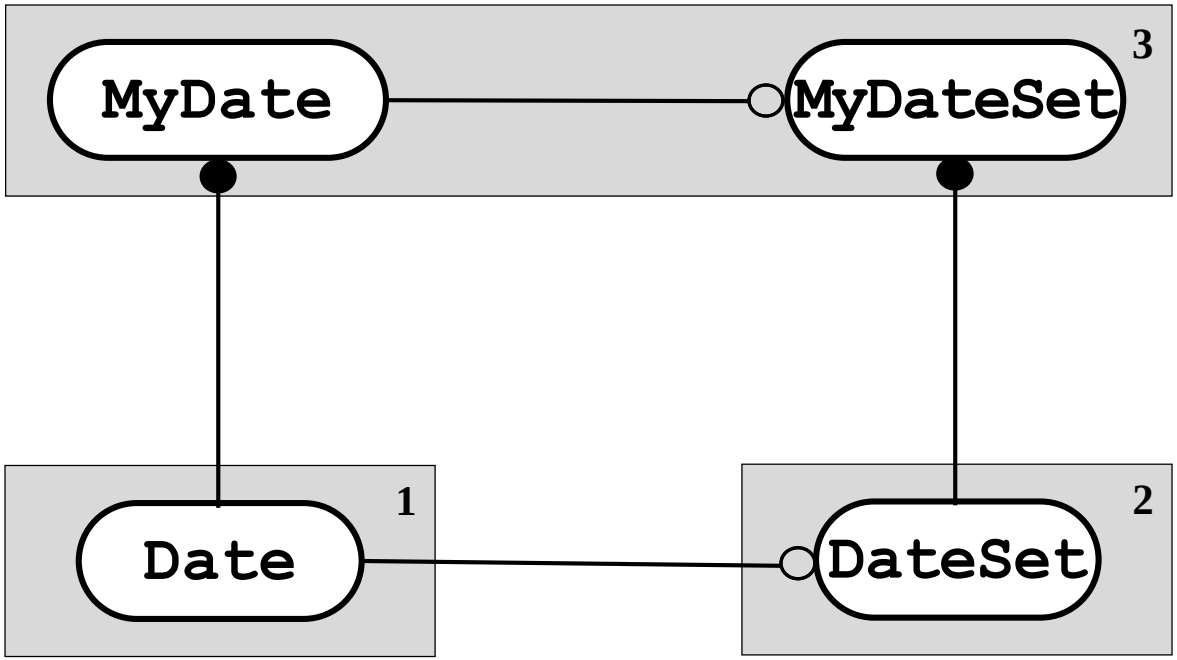
Escalating Encapsulation

Single-Component Wrapper



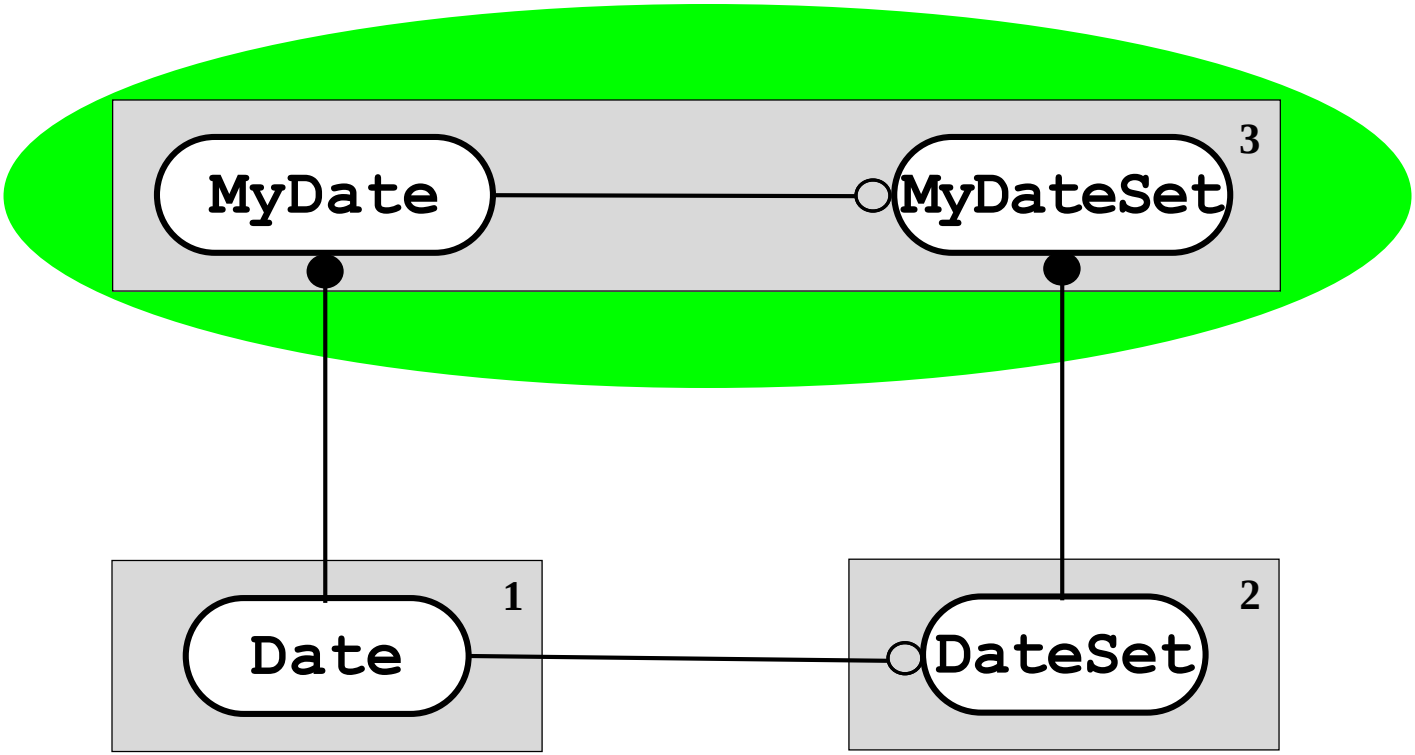
Escalating Encapsulation

Single-Component Wrapper



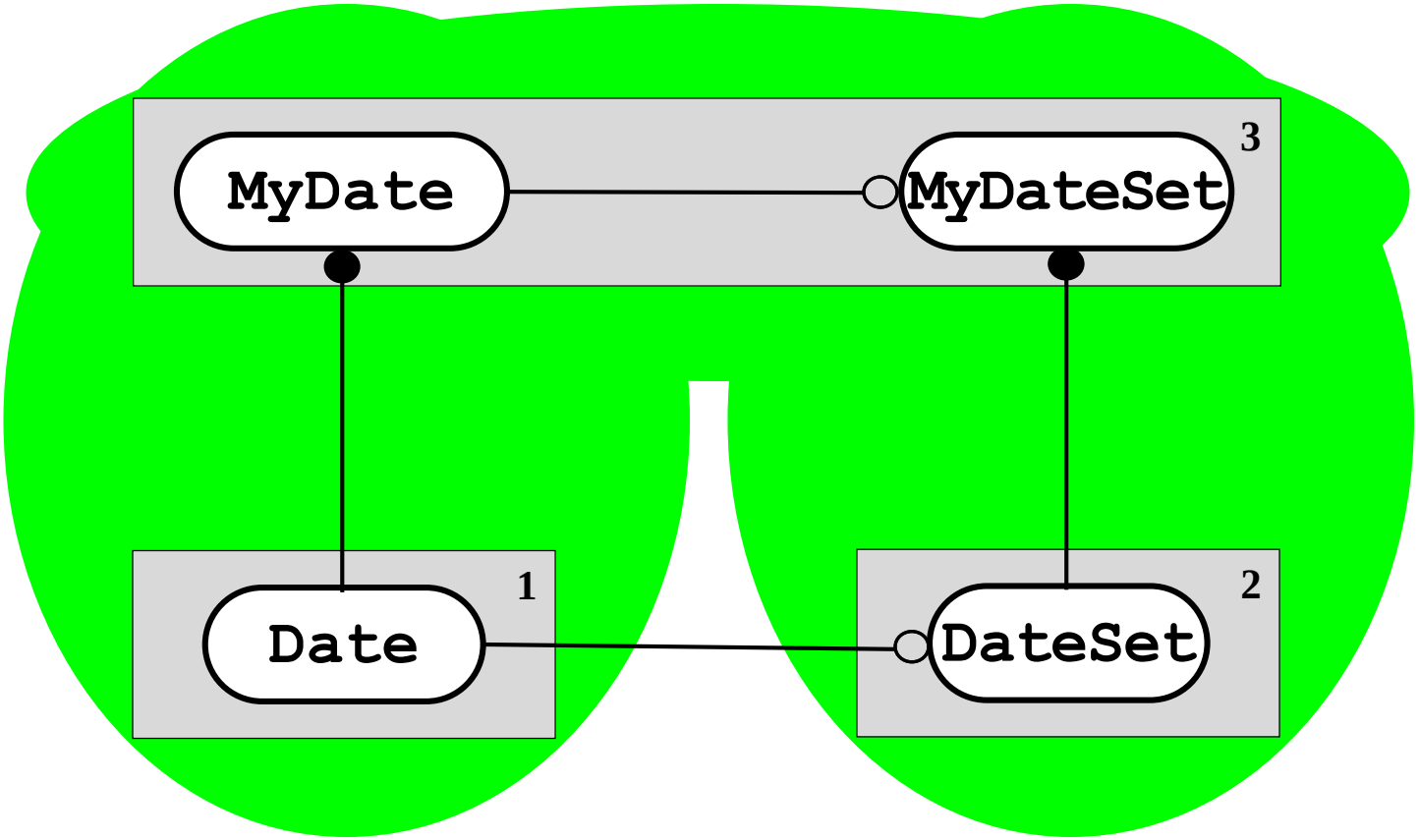
Escalating Encapsulation

Single-Component Wrapper



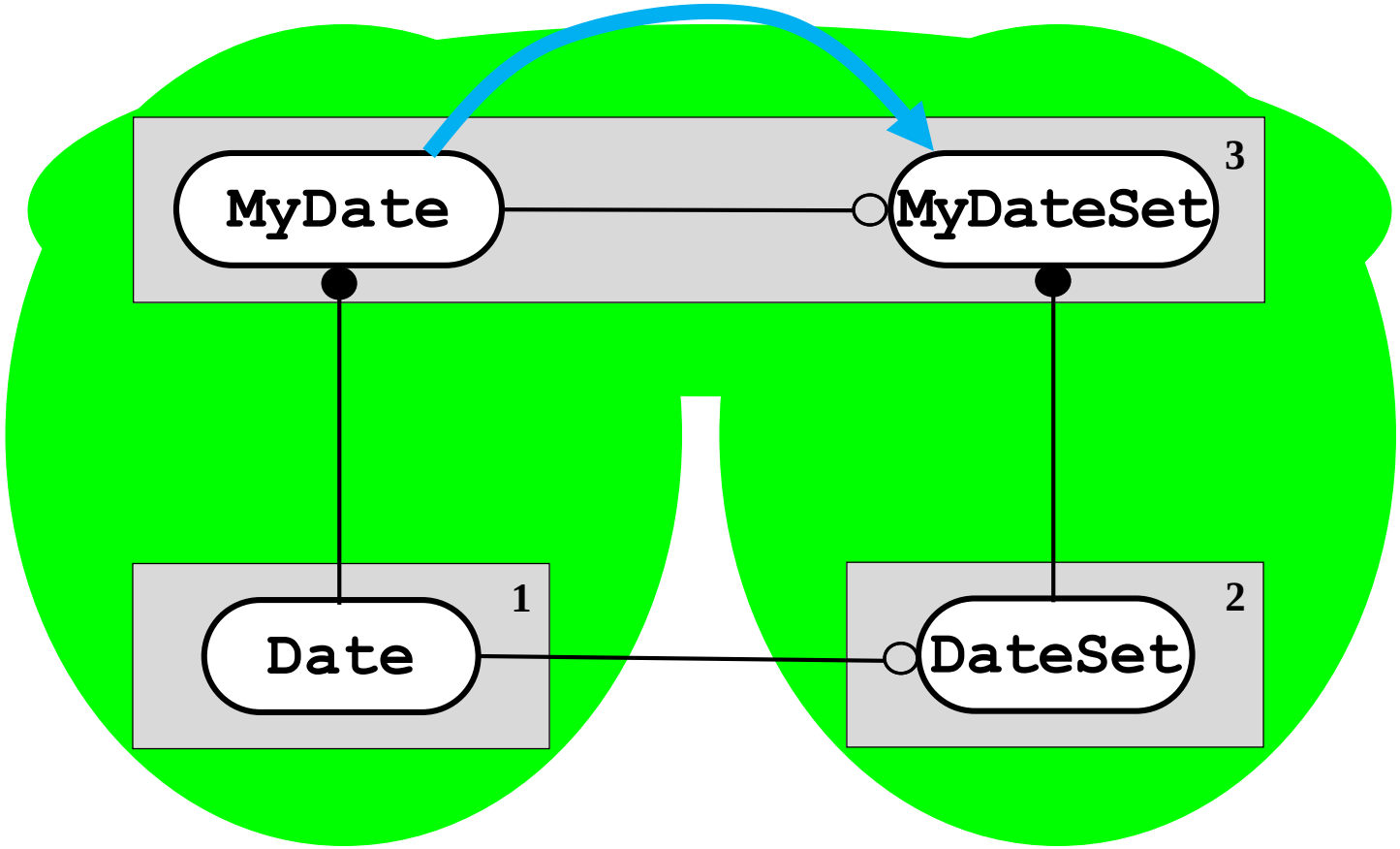
Escalating Encapsulation

Single-Component Wrapper



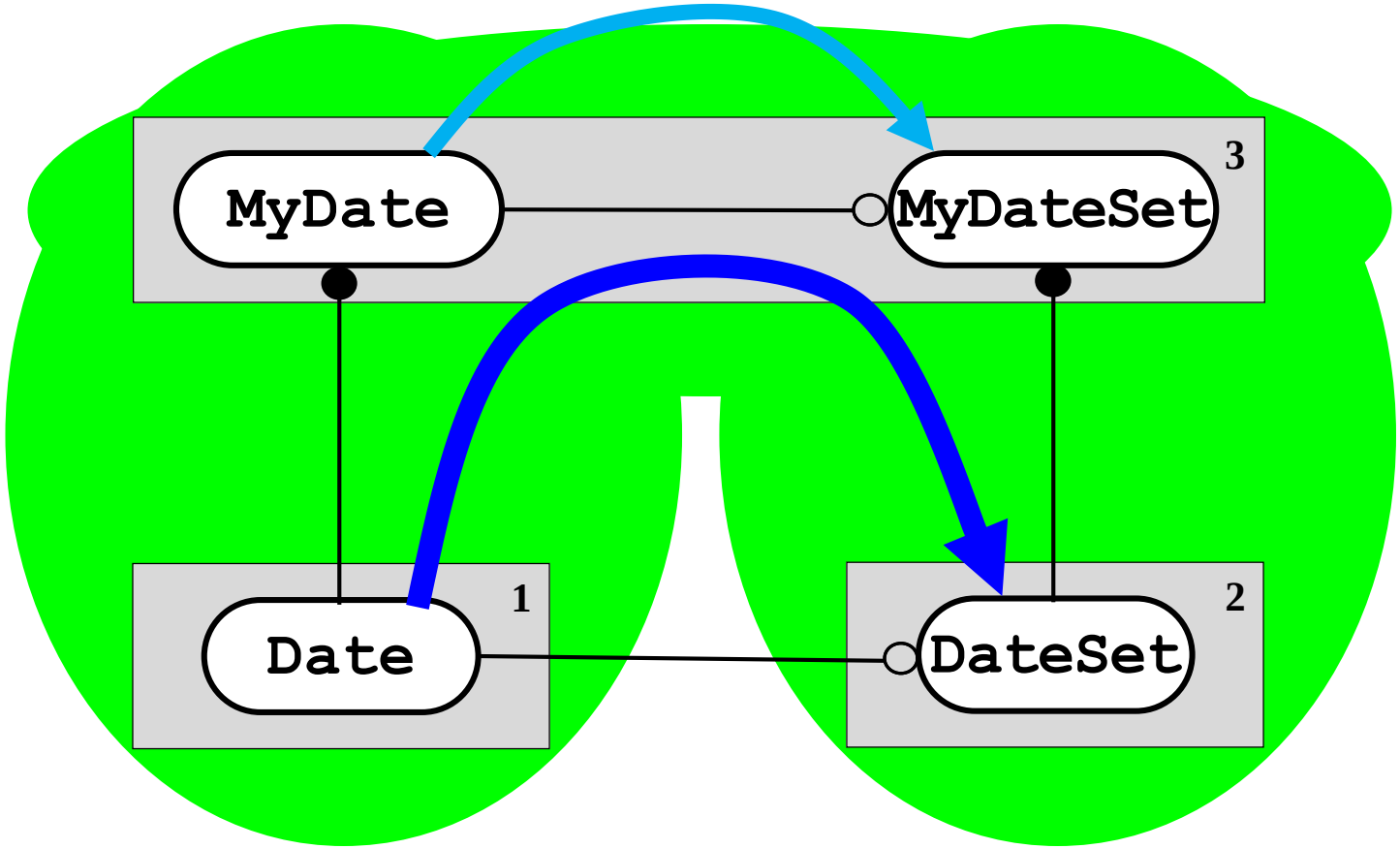
Escalating Encapsulation

Single-Component Wrapper



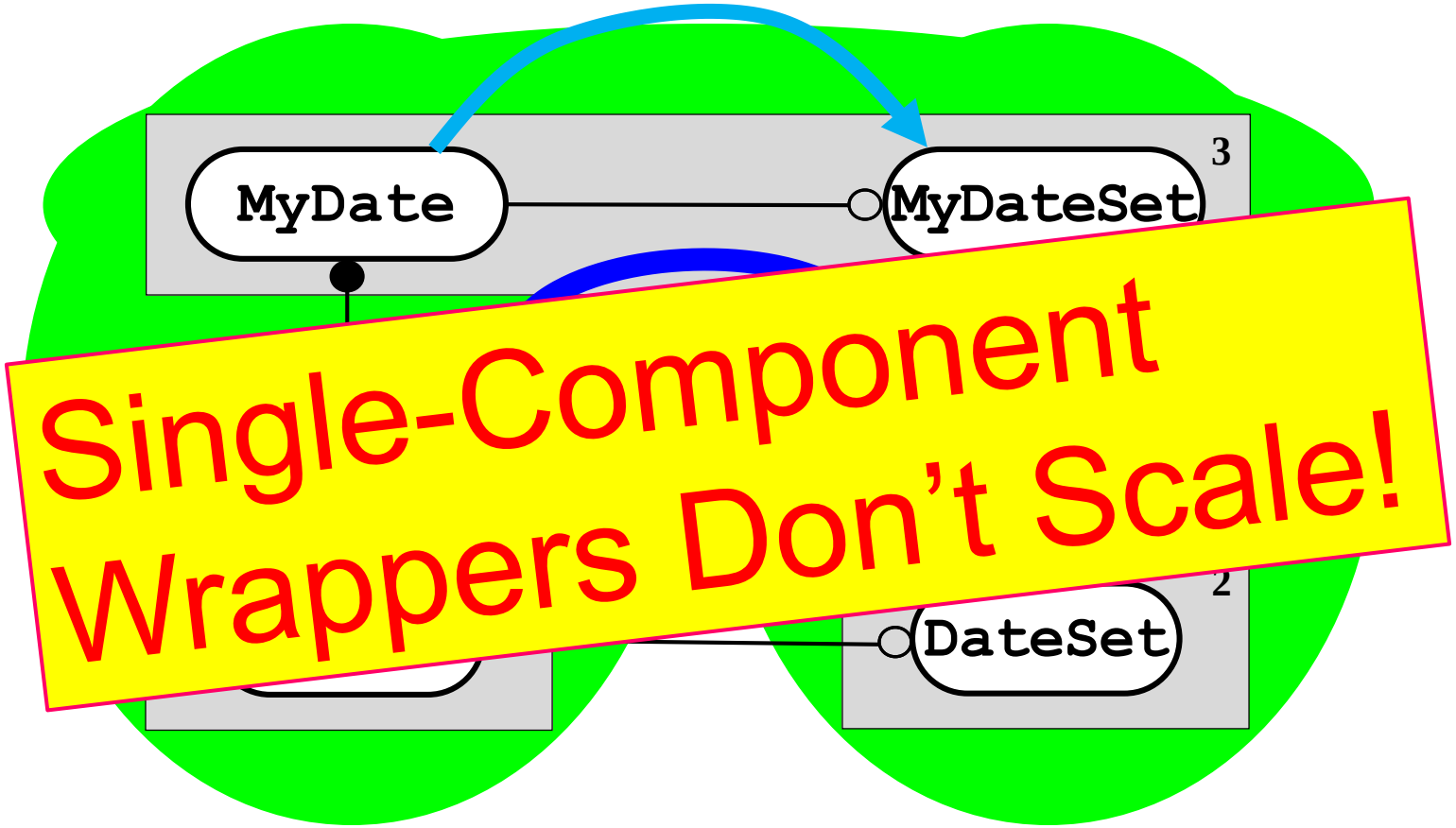
Escalating Encapsulation

Single-Component Wrapper



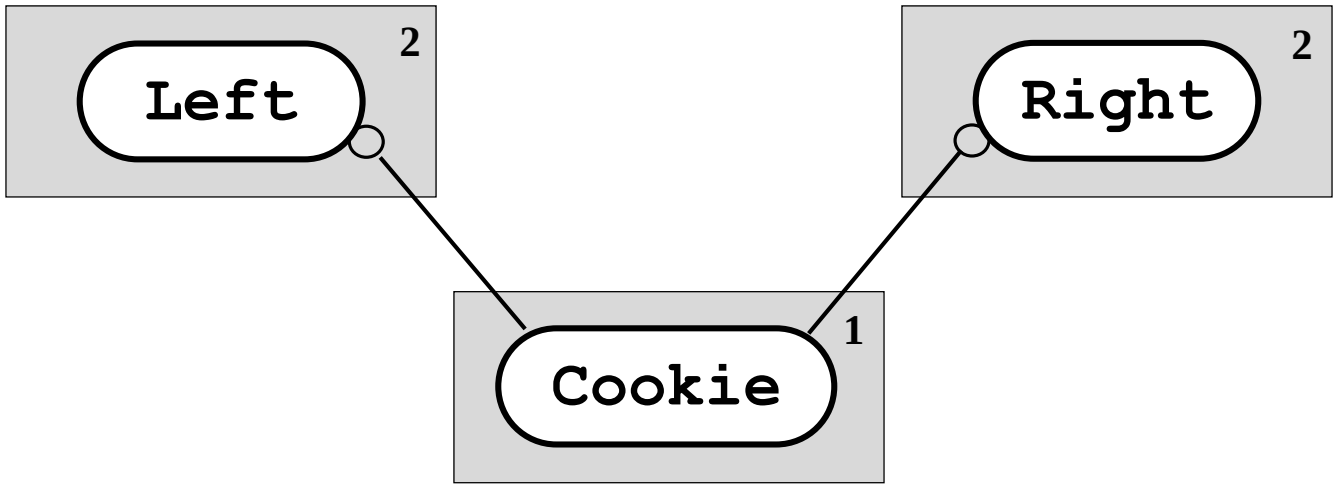
Escalating Encapsulation

Single-Component Wrapper



Escalating Encapsulation

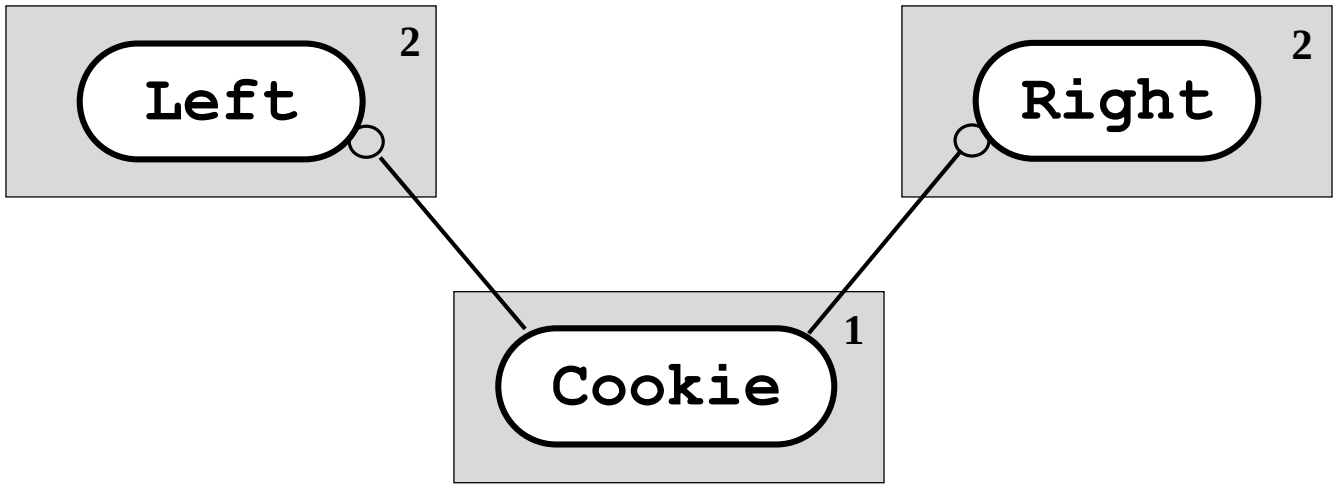
Hiding Header Files



Escalating Encapsulation

Hiding Header Files

```
// left.h
#include <cookie.h>
class Left {
  // …
  void setC(const Cookie& c);
  // …
};
```

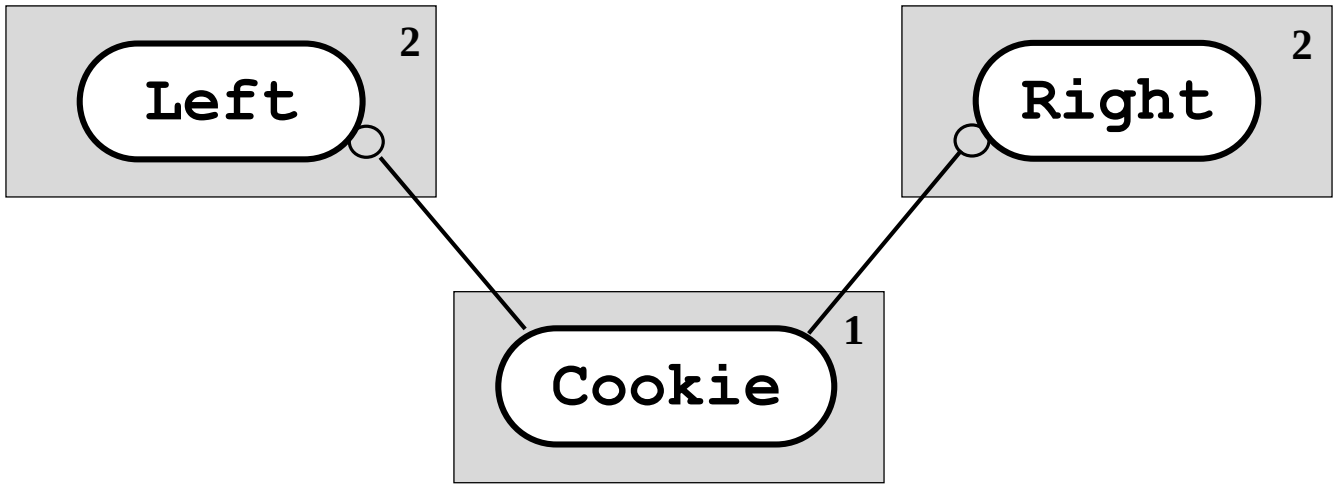


Escalating Encapsulation

Hiding Header Files

```
// left.h
#include <cookie.h>
class Left {
  // …
  void setC(const Cookie& c);
  // …
};
```

```
// right.h
#include <cookie.h>
class Right {
  // …
  const Cookie& getC() const;
  // …
};
```

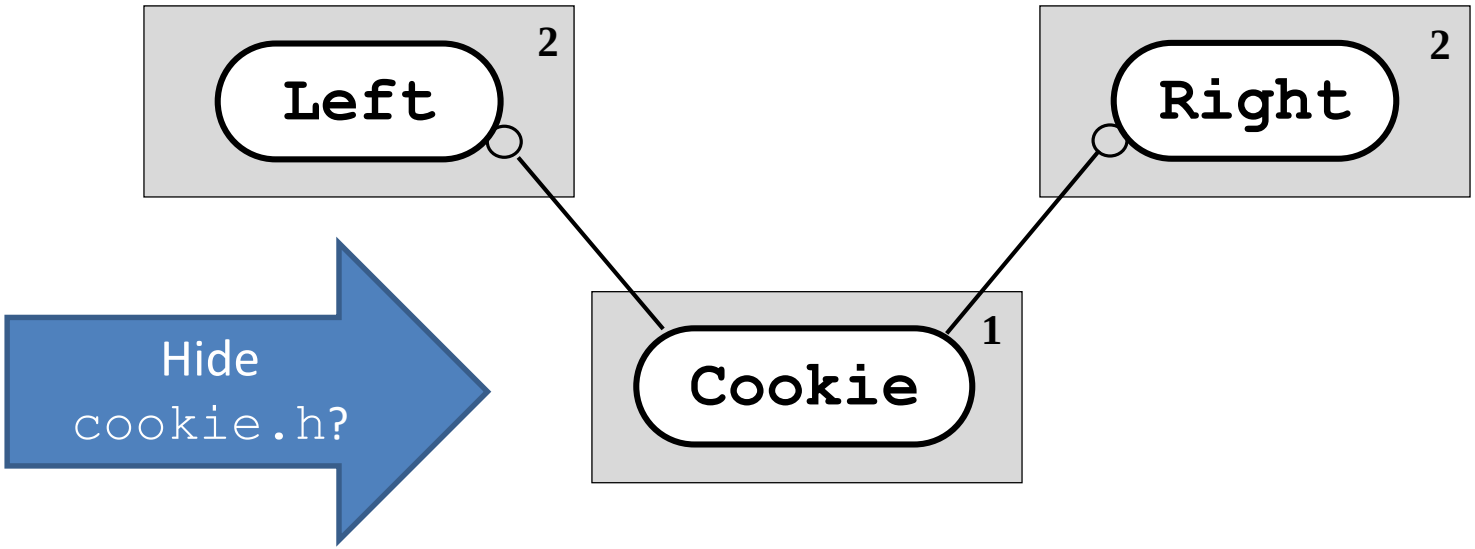


Escalating Encapsulation

Hiding Header Files

```
// left.h
#include <cookie.h>
class Left {
  // …
  void setC(const Cookie& c);
  // …
};
```

```
// right.h
#include <cookie.h>
class Right {
  // …
  const Cookie& getC() const;
  // …
};
```

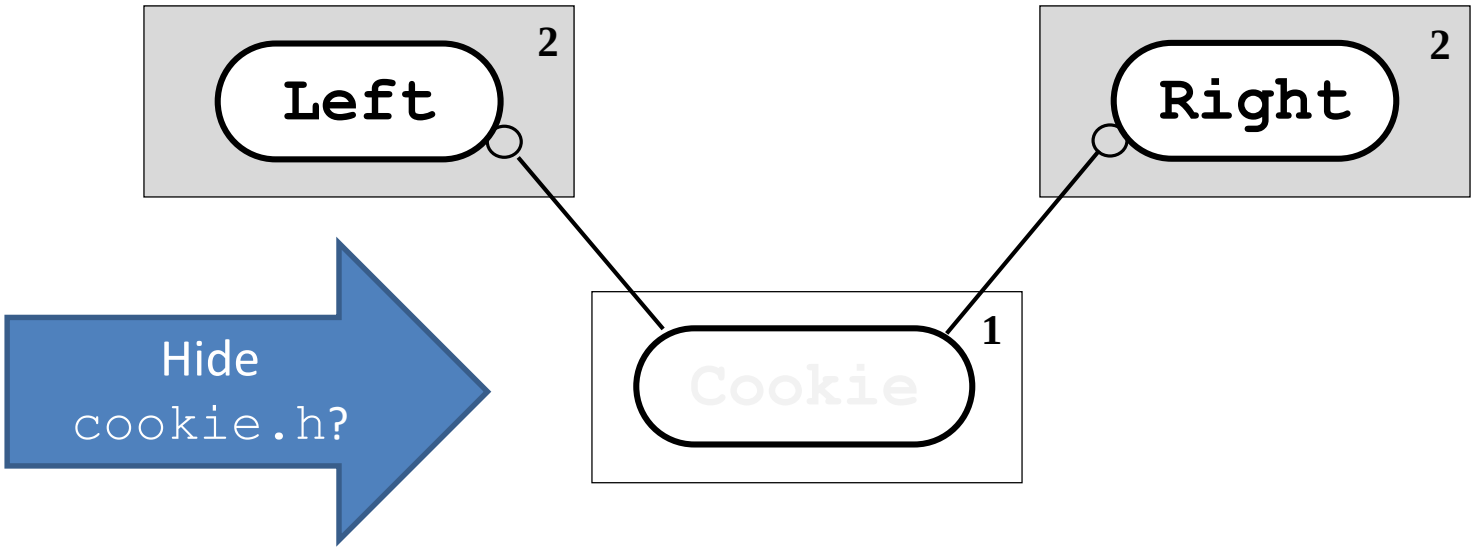


Escalating Encapsulation

Hiding Header Files

```
// left.h
#include <cookie.h>
class Left {
  // …
  void setC(const Cookie& c);
  // …
};
```

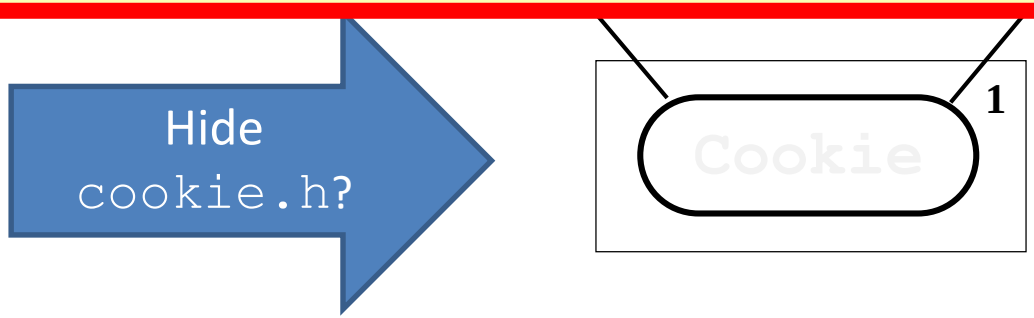
```
// right.h
#include <cookie.h>
class Right {
  // …
  const Cookie& getC() const;
  // …
};
```



Escalating Encapsulation

Hiding Header Files

Bad Idea:

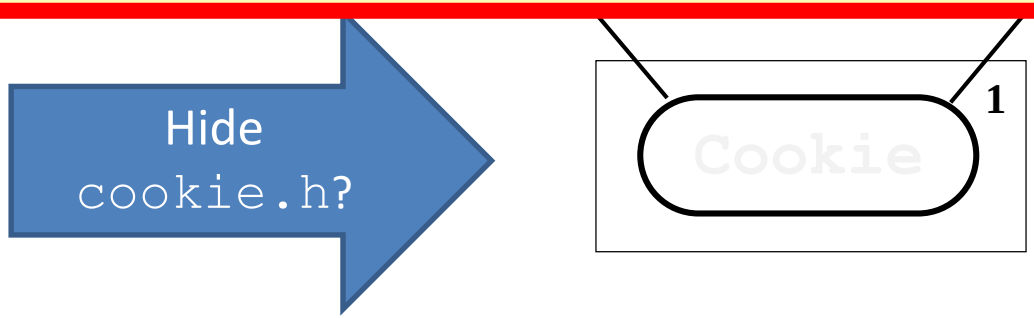


Escalating Encapsulation

Hiding Header Files

Bad Idea:

(1) Convolves architecture with deployment.



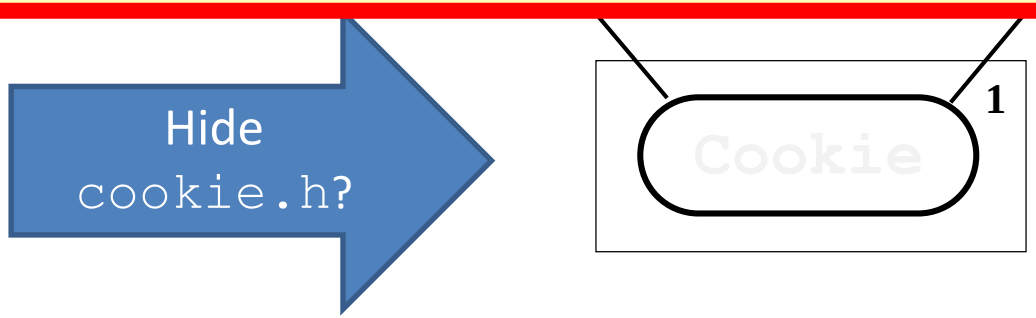
Escalating Encapsulation

Hiding Header Files

Bad Idea:

(1) Convolves architecture with deployment.

(2) Inhibits side-by-side reuse of the "hidden" component.



Escalating Encapsulation

Hiding Header Files

Bad Idea:

(1) Convolves architecture with deployment.

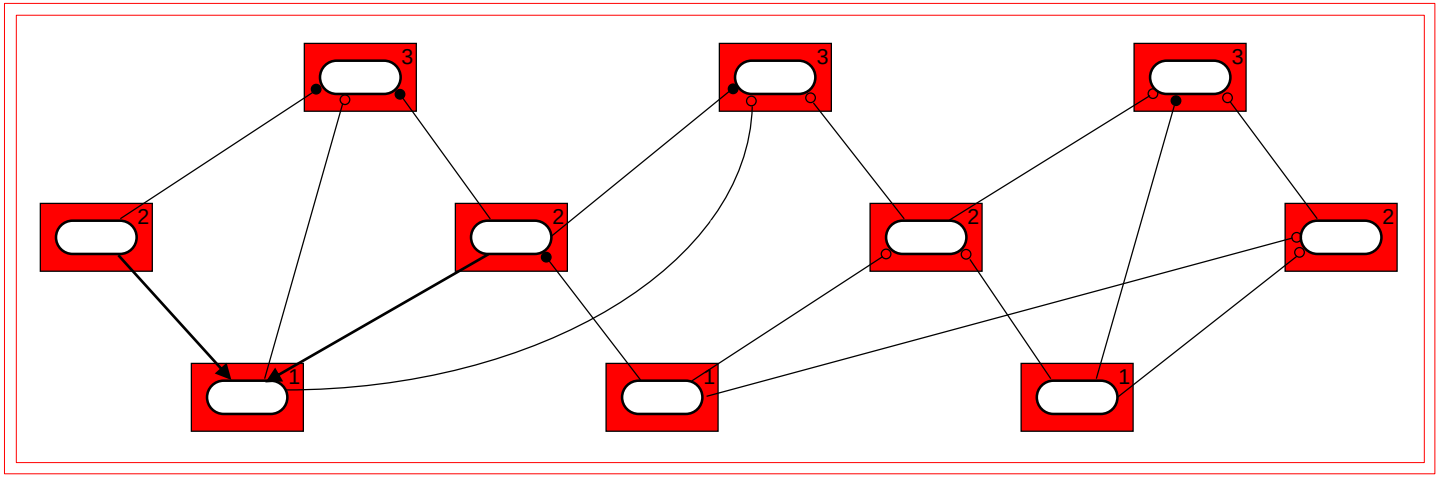
(2) Inhibits side-by-side reuse of the "hidden" component.

Escalating Encapsulation

Wrapper Package

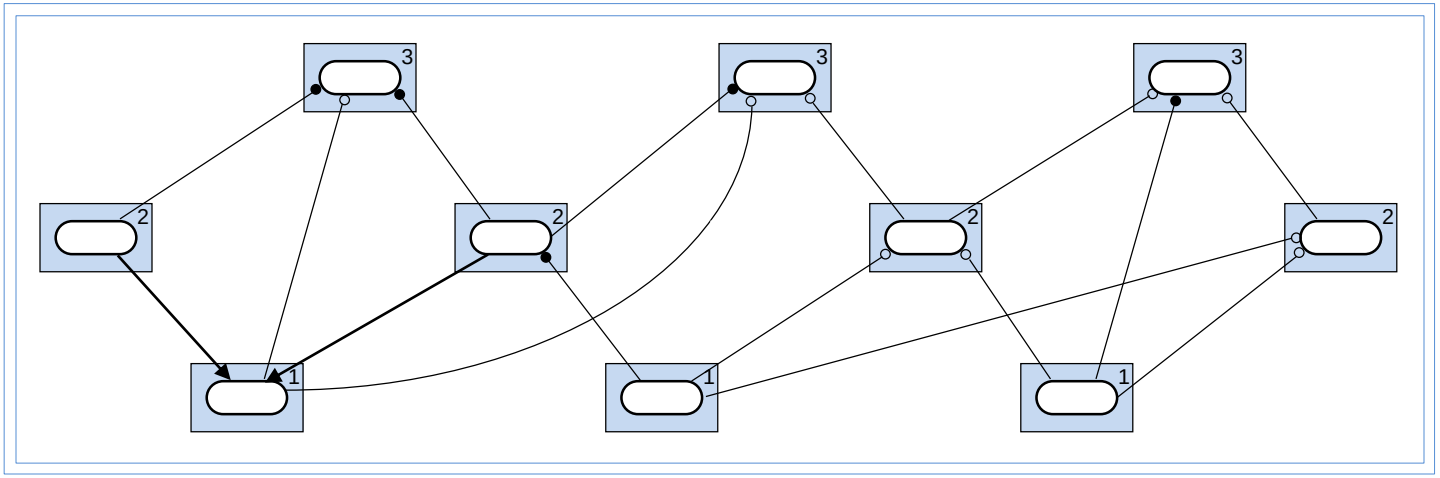
Escalating Encapsulation

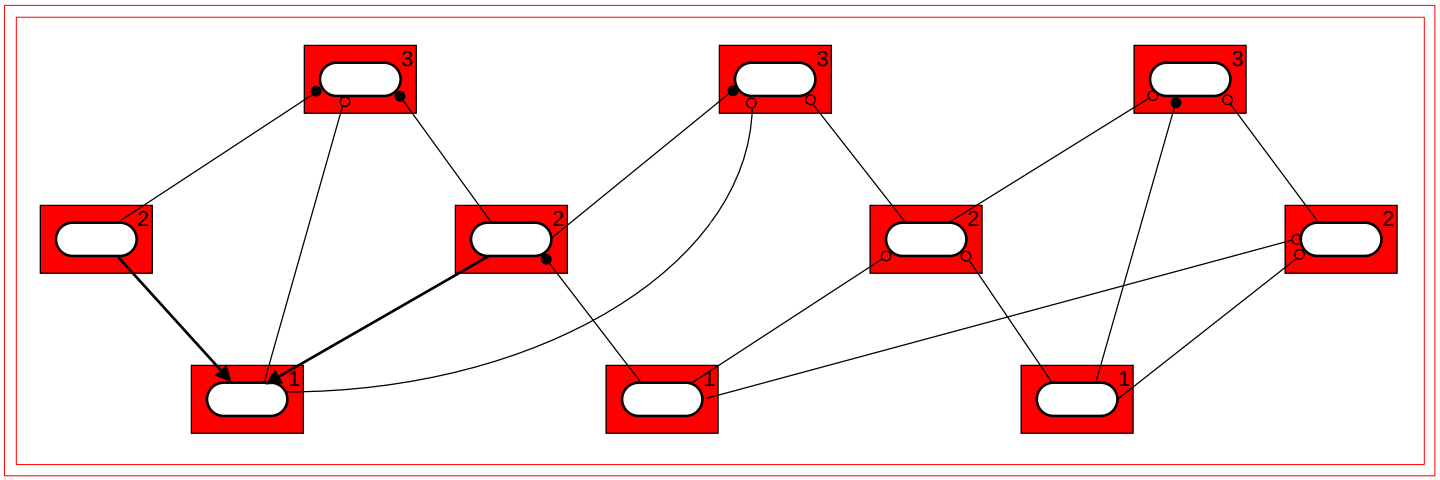
Wrapper Package



Escalating Encapsulation

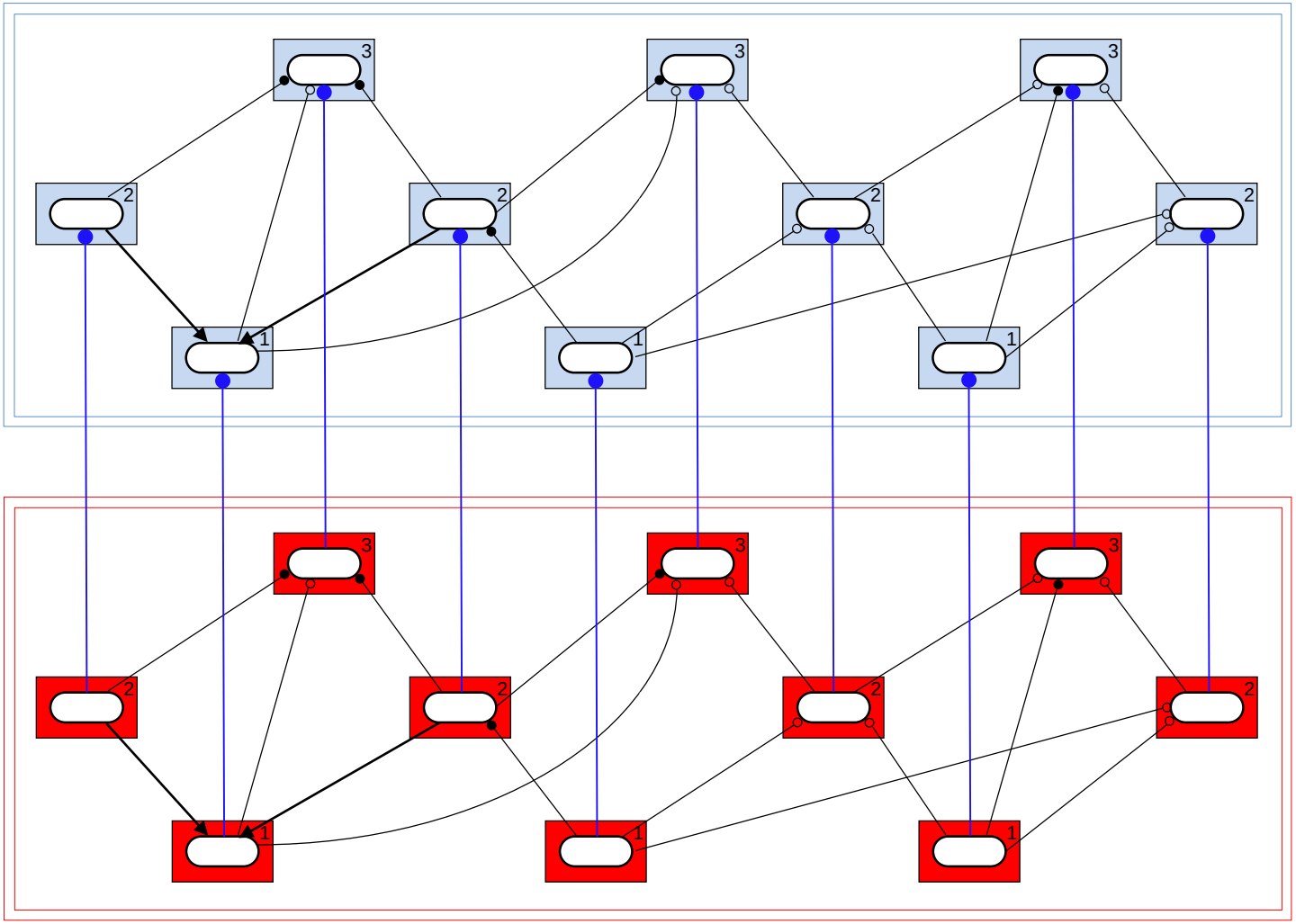
Wrapper Package





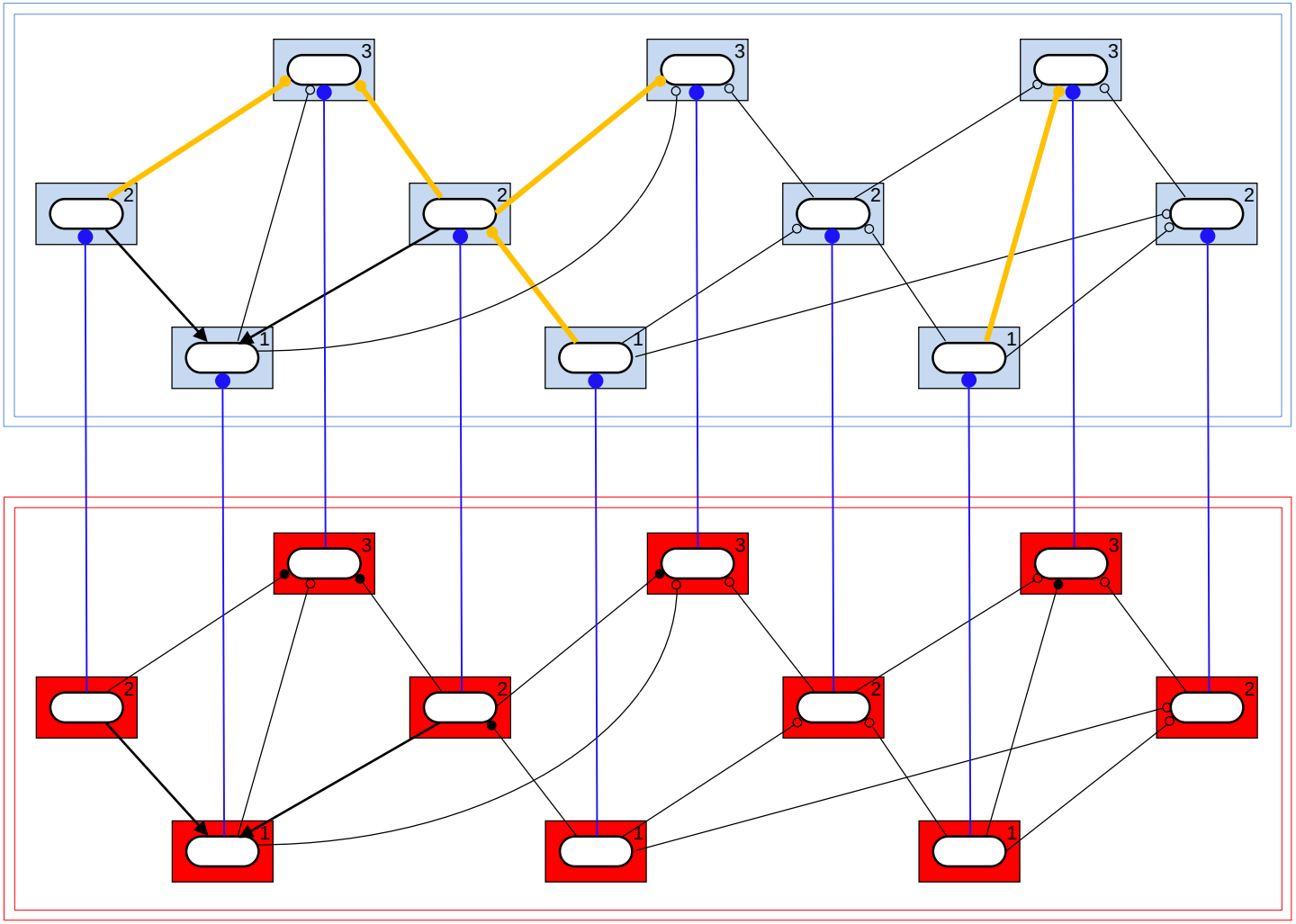
Escalating Encapsulation

Wrapper Package



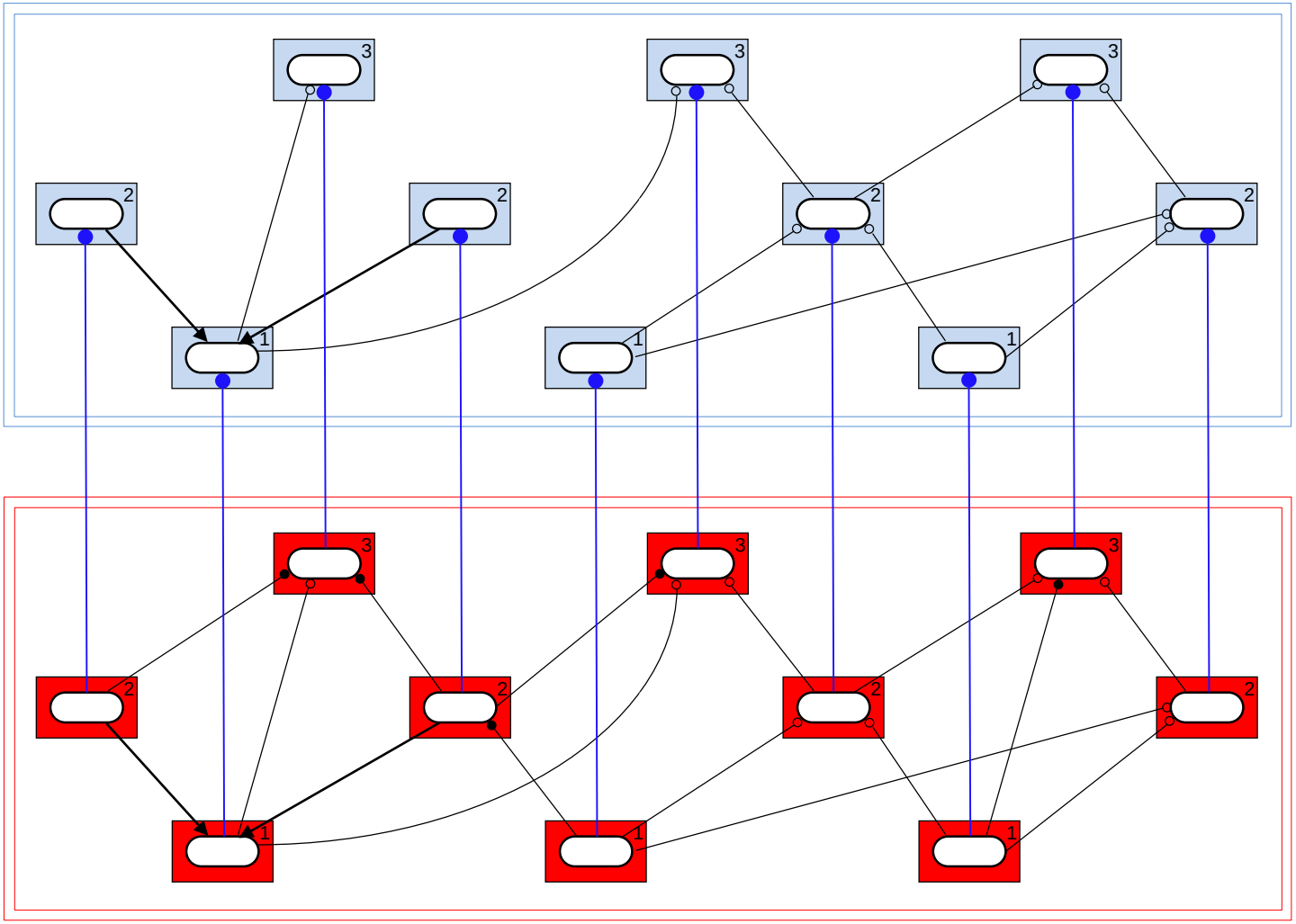
Escalating Encapsulation

Wrapper Package



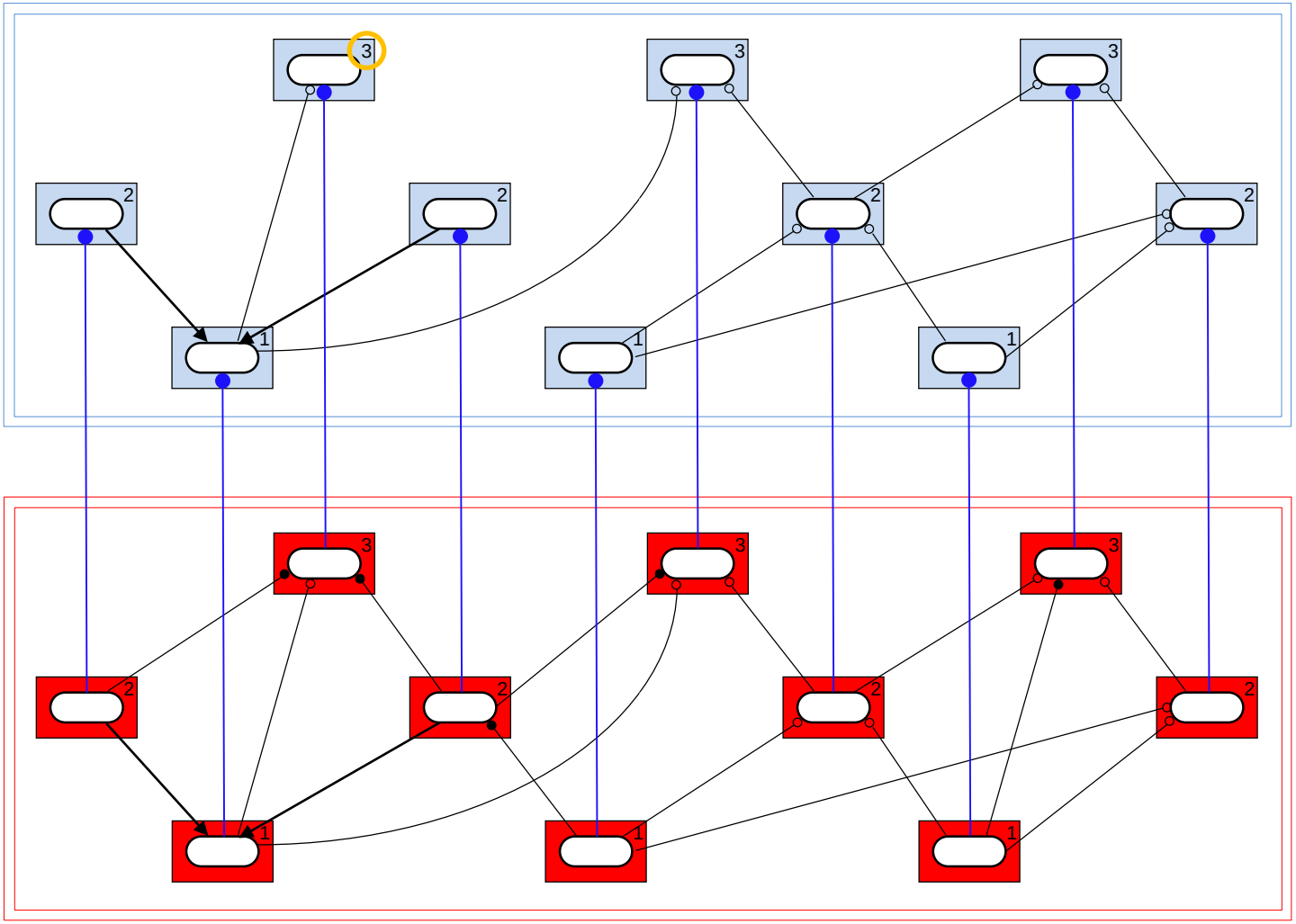
Escalating Encapsulation

Wrapper Package



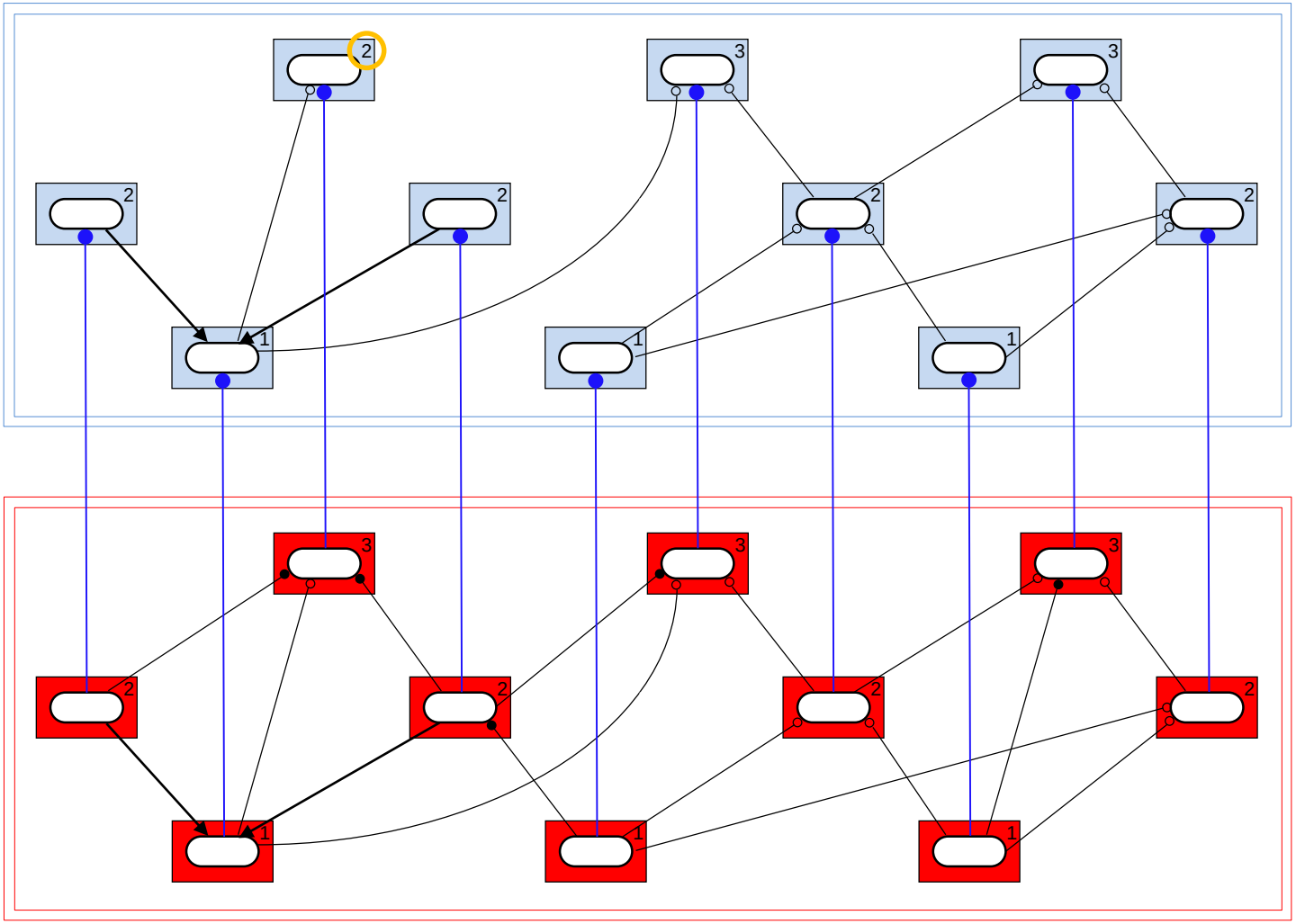
Escalating Encapsulation

Wrapper Package



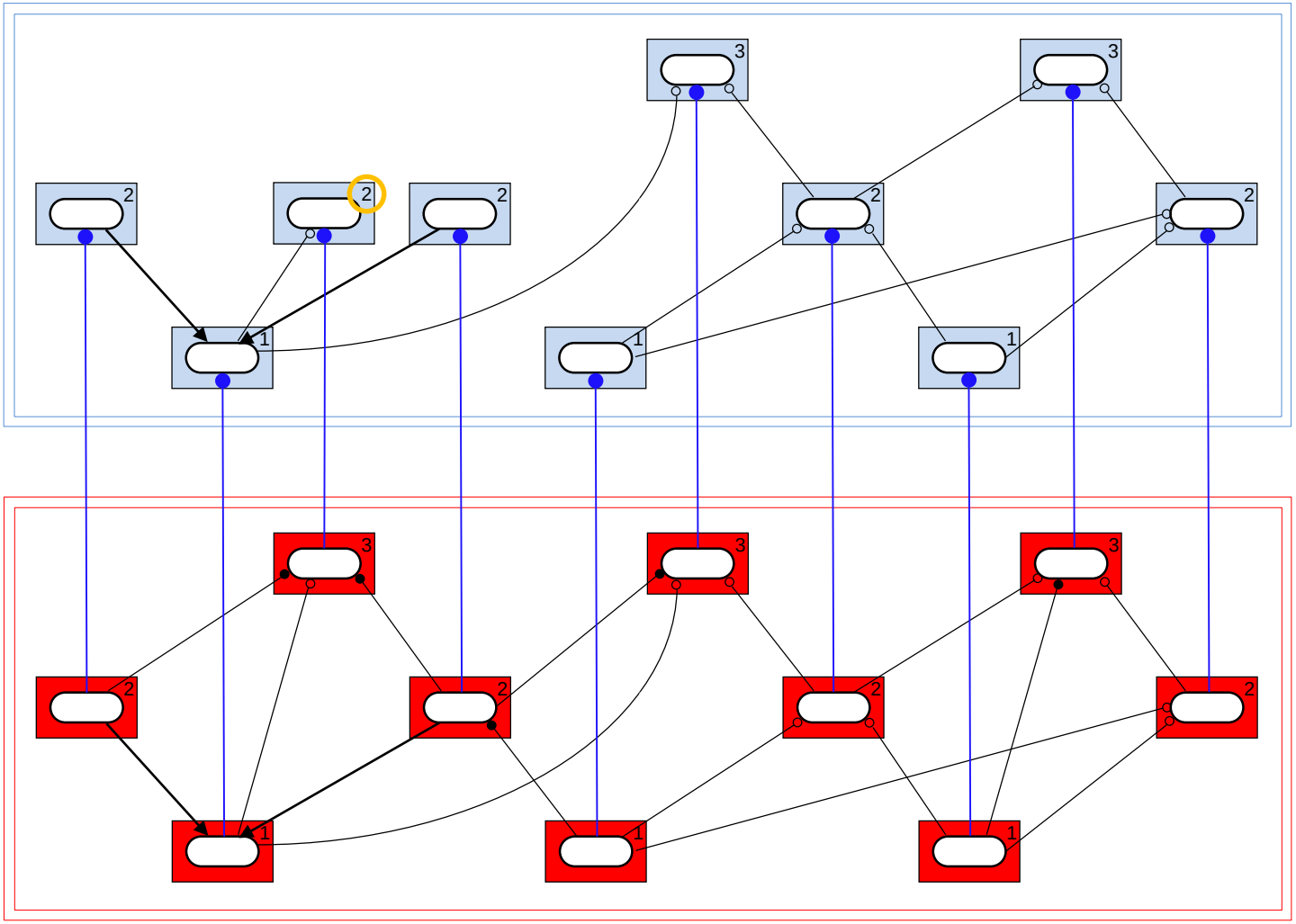
Escalating Encapsulation

Wrapper Package



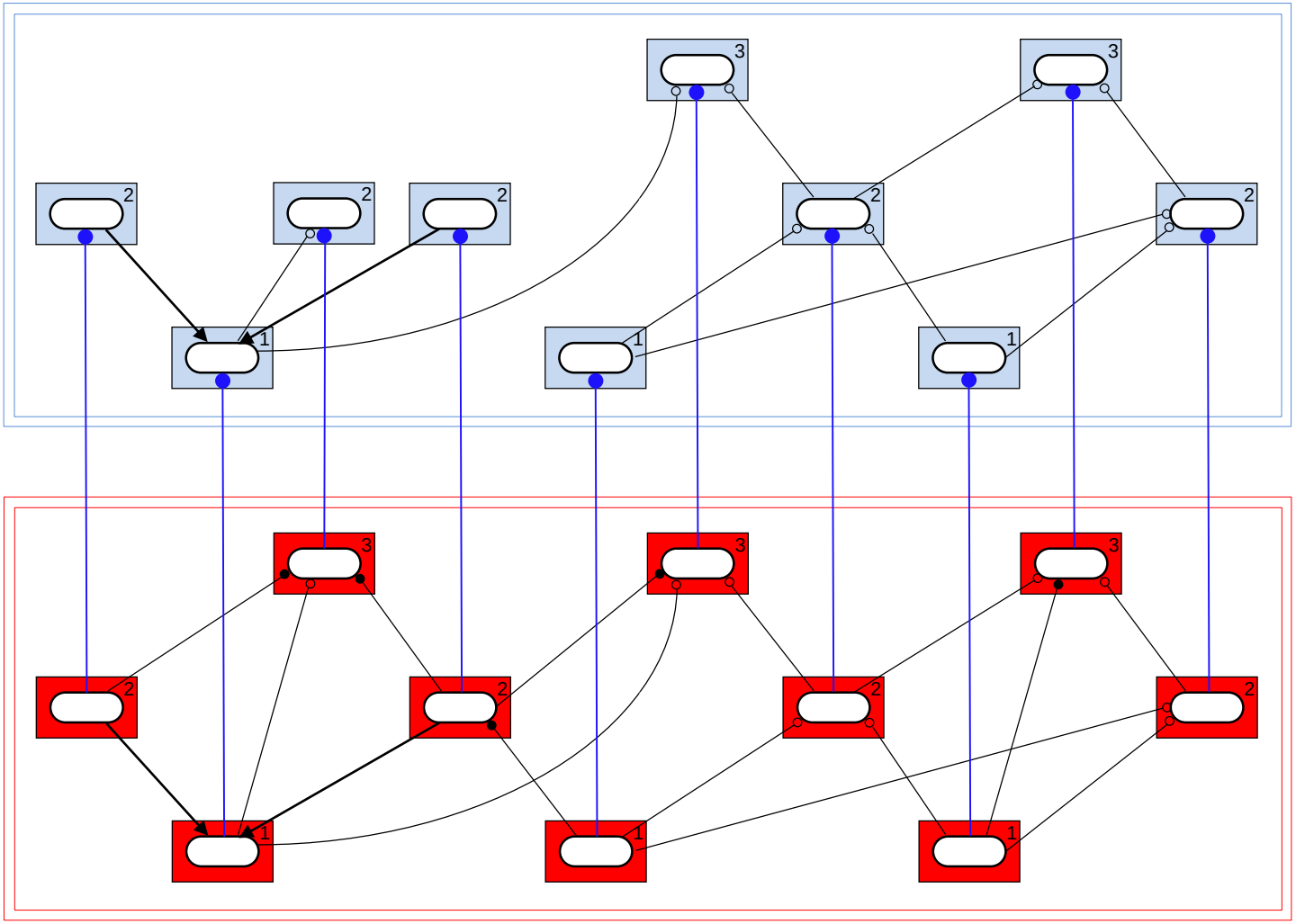
Escalating Encapsulation

Wrapper Package



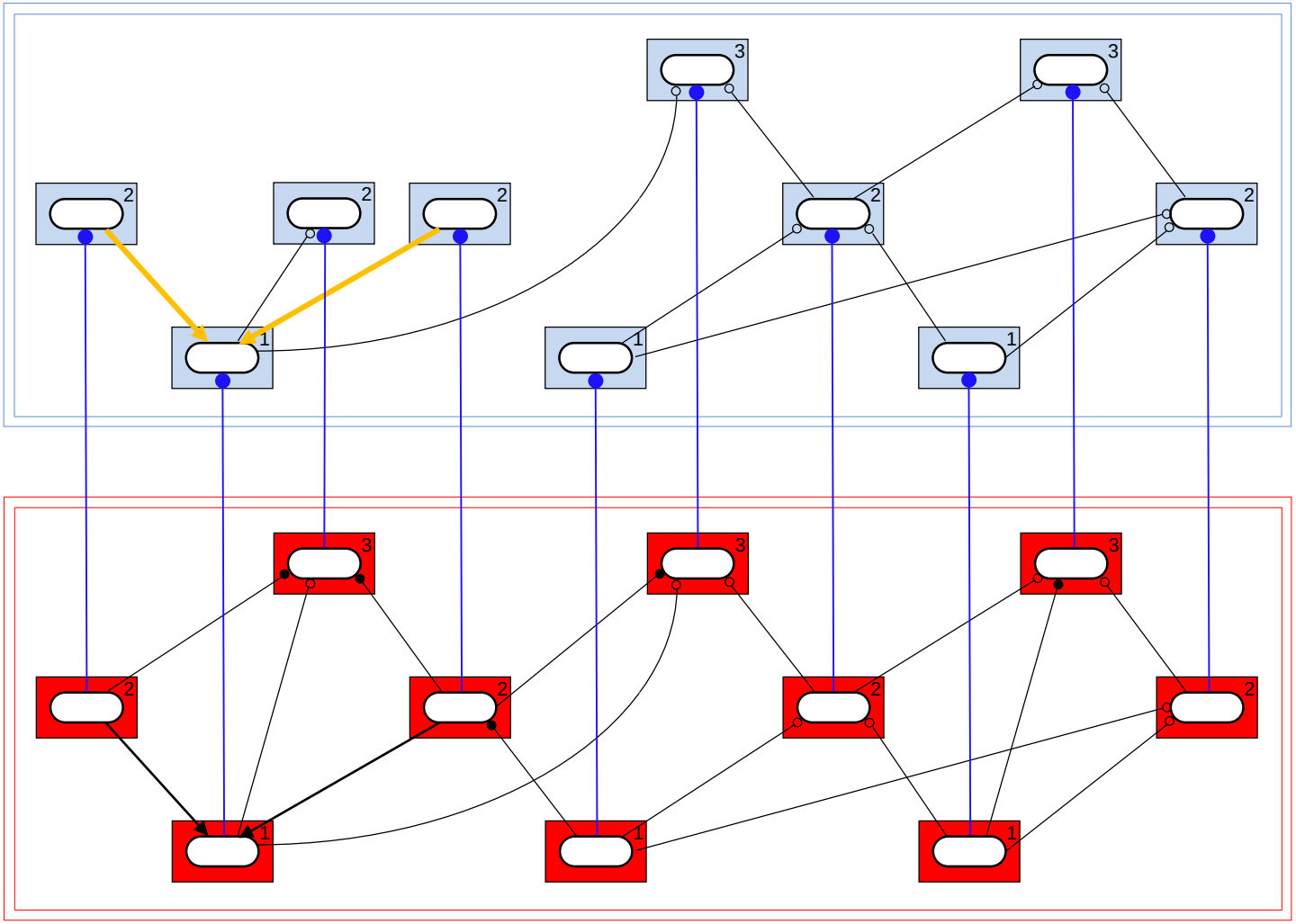
Escalating Encapsulation

Wrapper Package



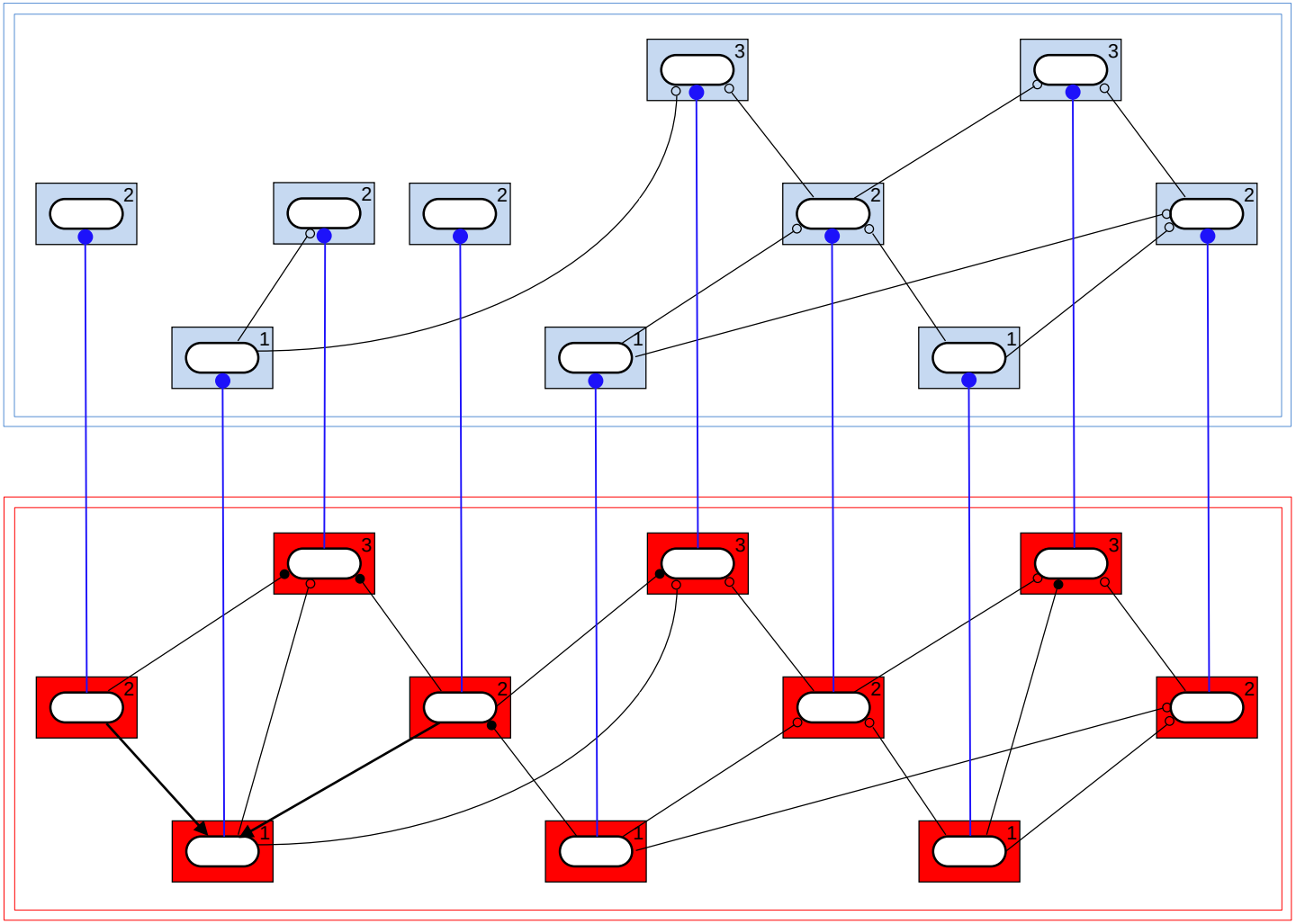
Escalating Encapsulation

Wrapper Package



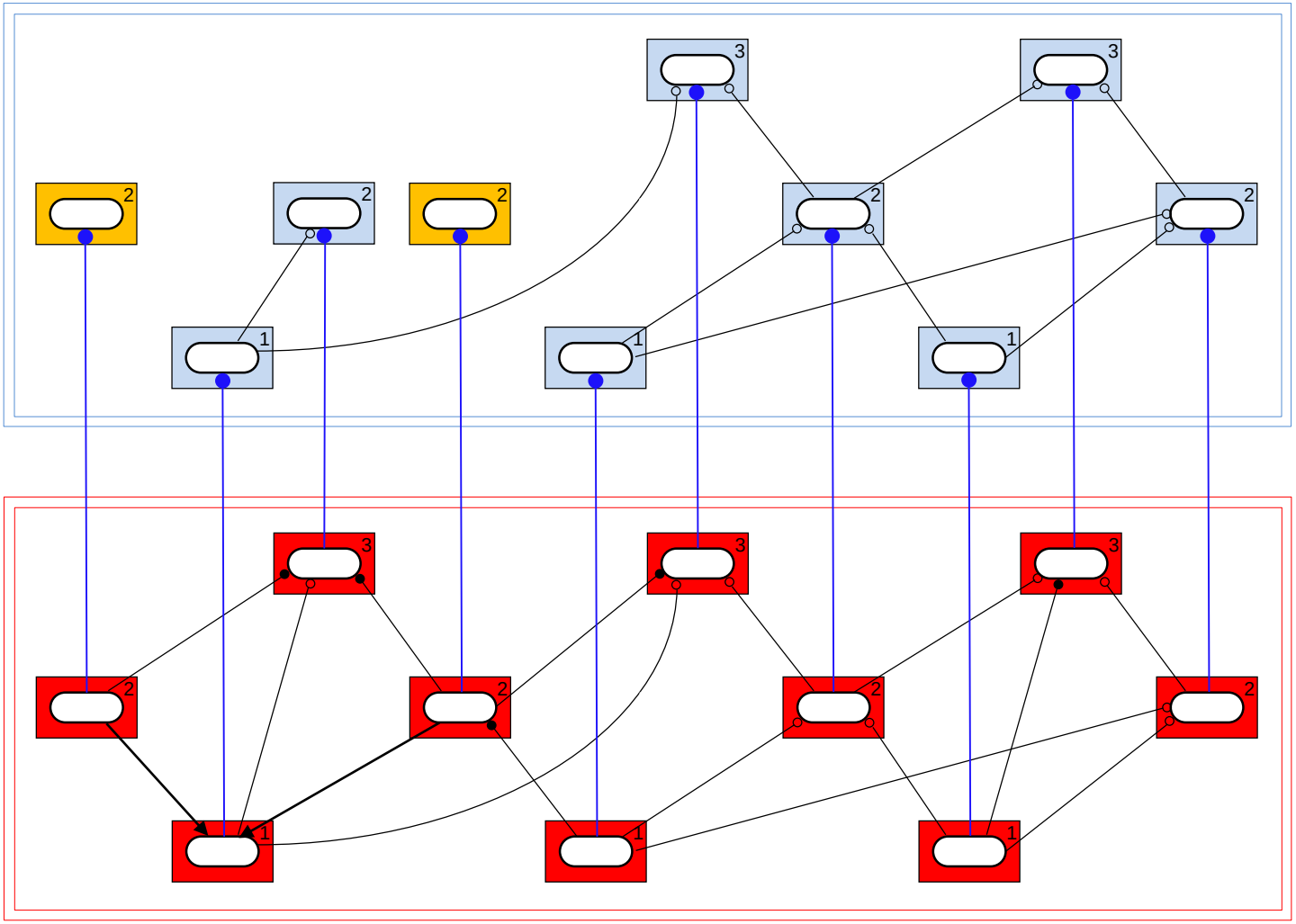
Escalating Encapsulation

Wrapper Package



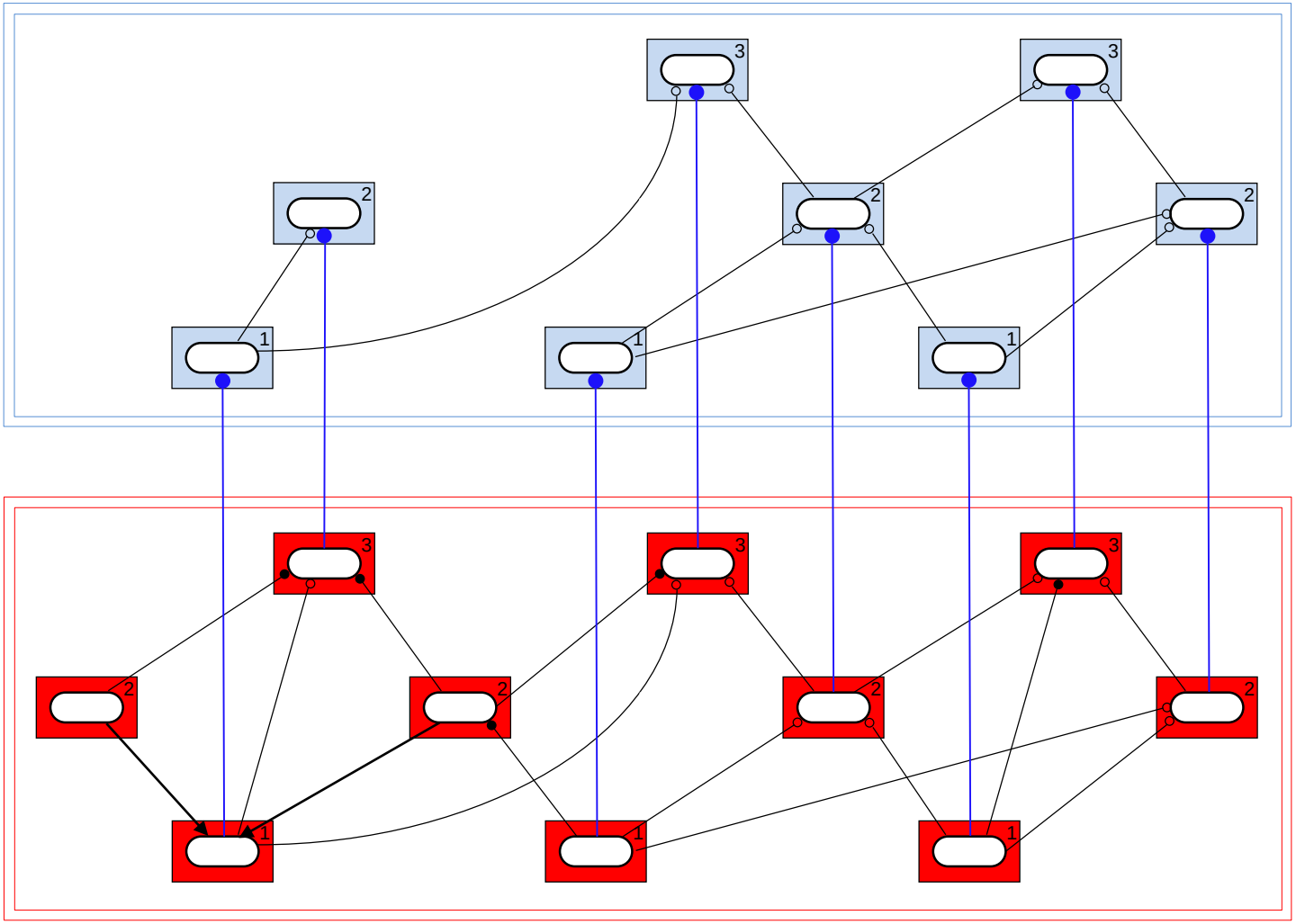
Escalating Encapsulation

Wrapper Package



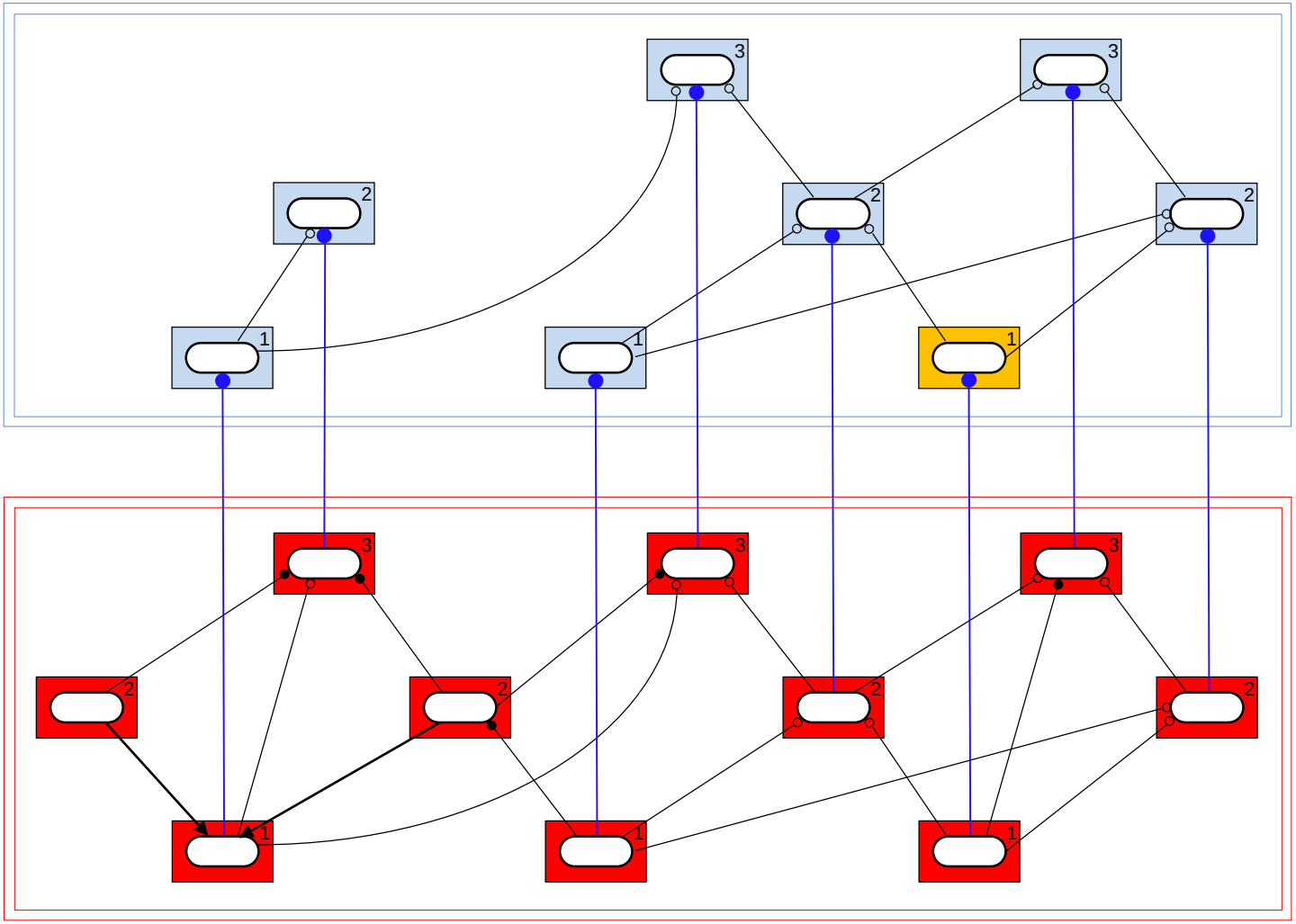
Escalating Encapsulation

Wrapper Package



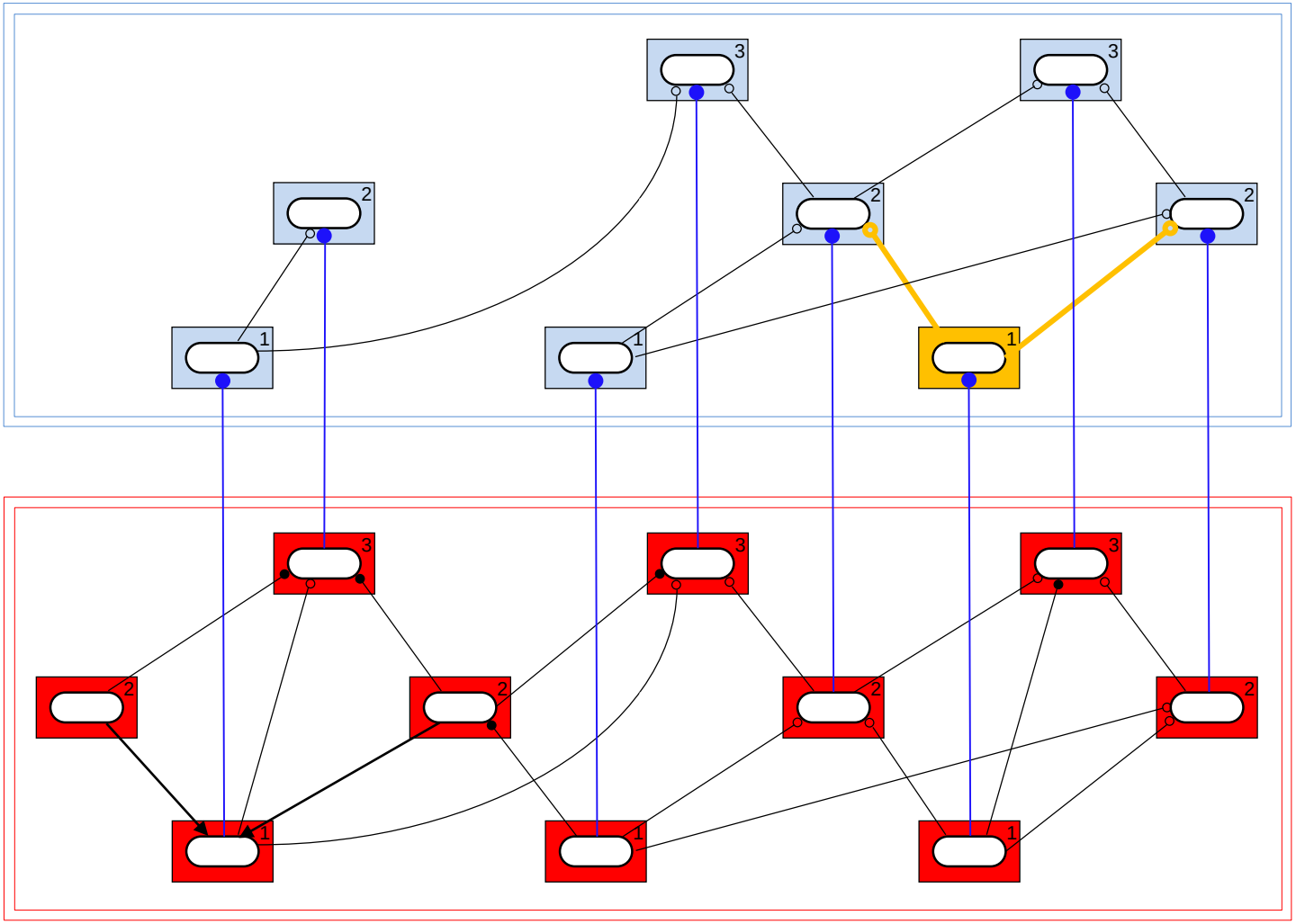
Escalating Encapsulation

Wrapper Package



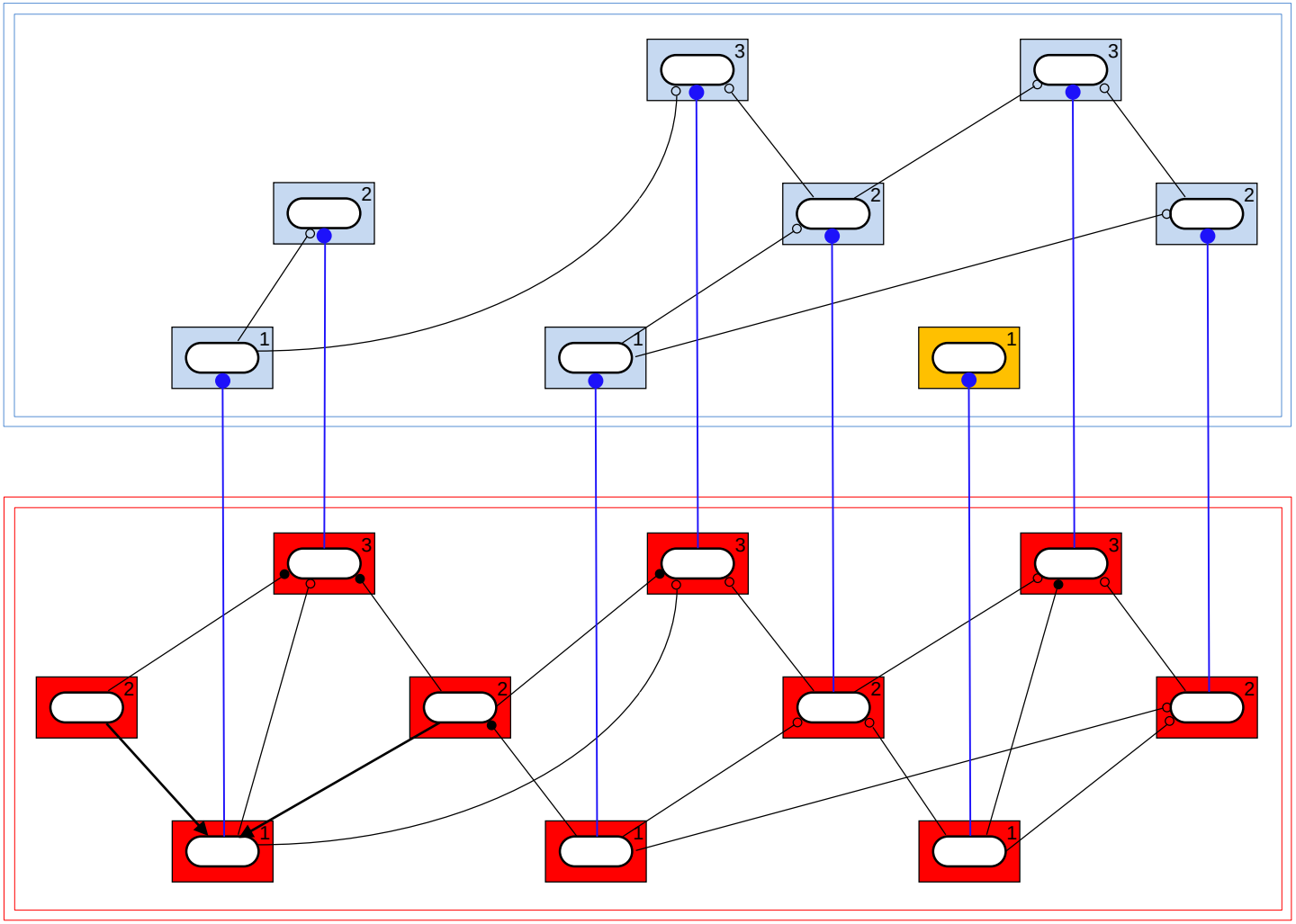
Escalating Encapsulation

Wrapper Package



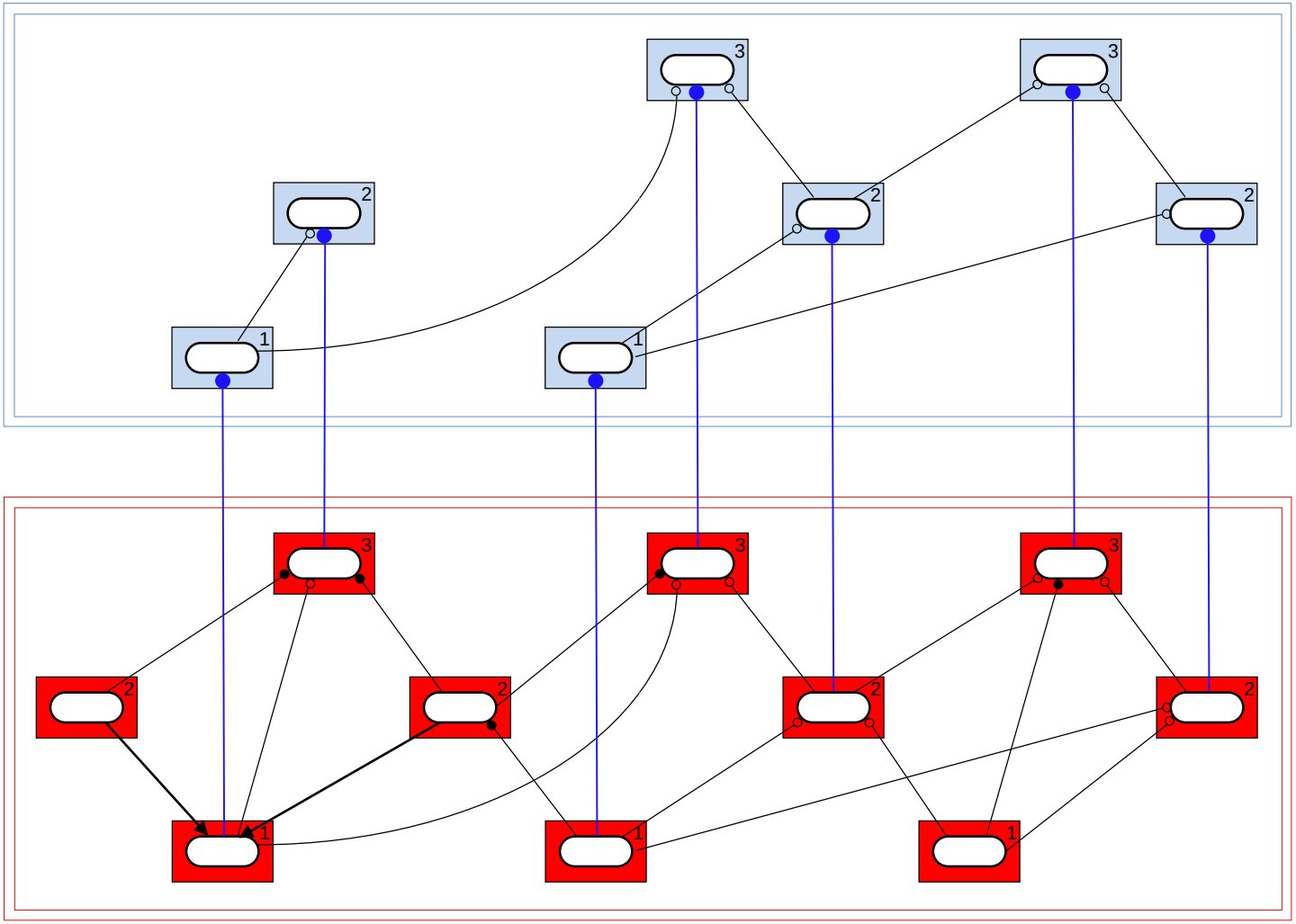
Escalating Encapsulation

Wrapper Package



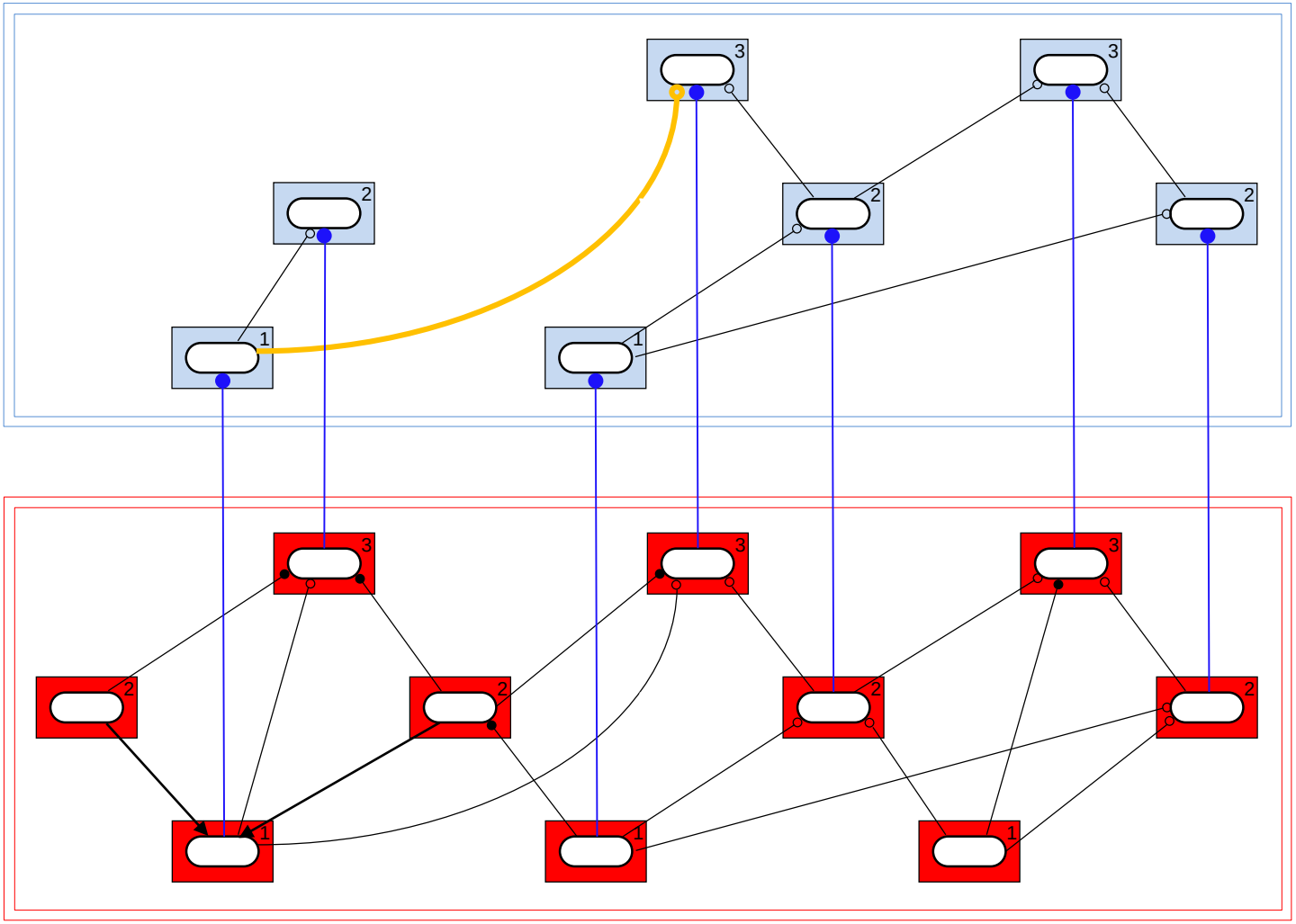
Escalating Encapsulation

Wrapper Package



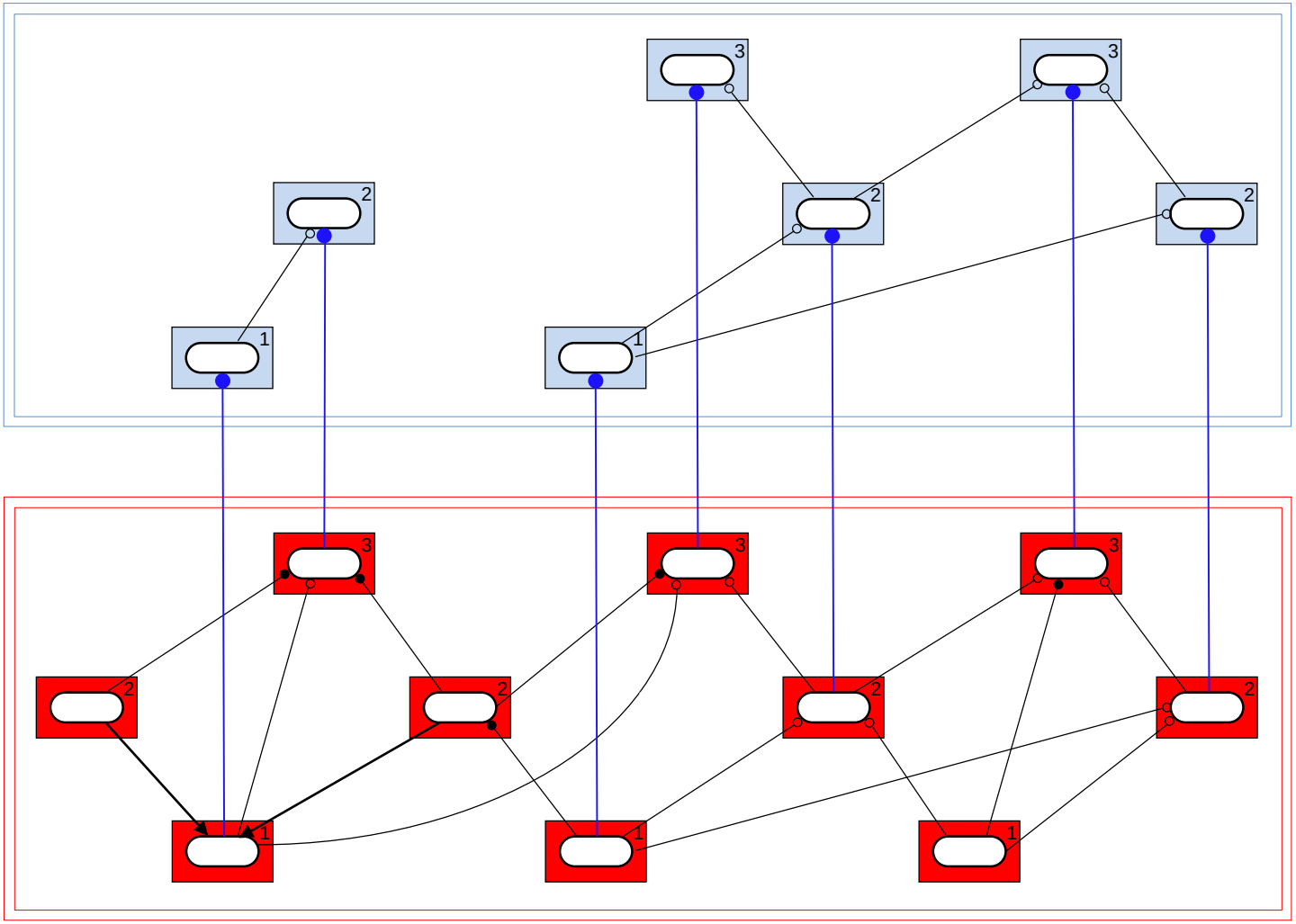
Escalating Encapsulation

Wrapper Package



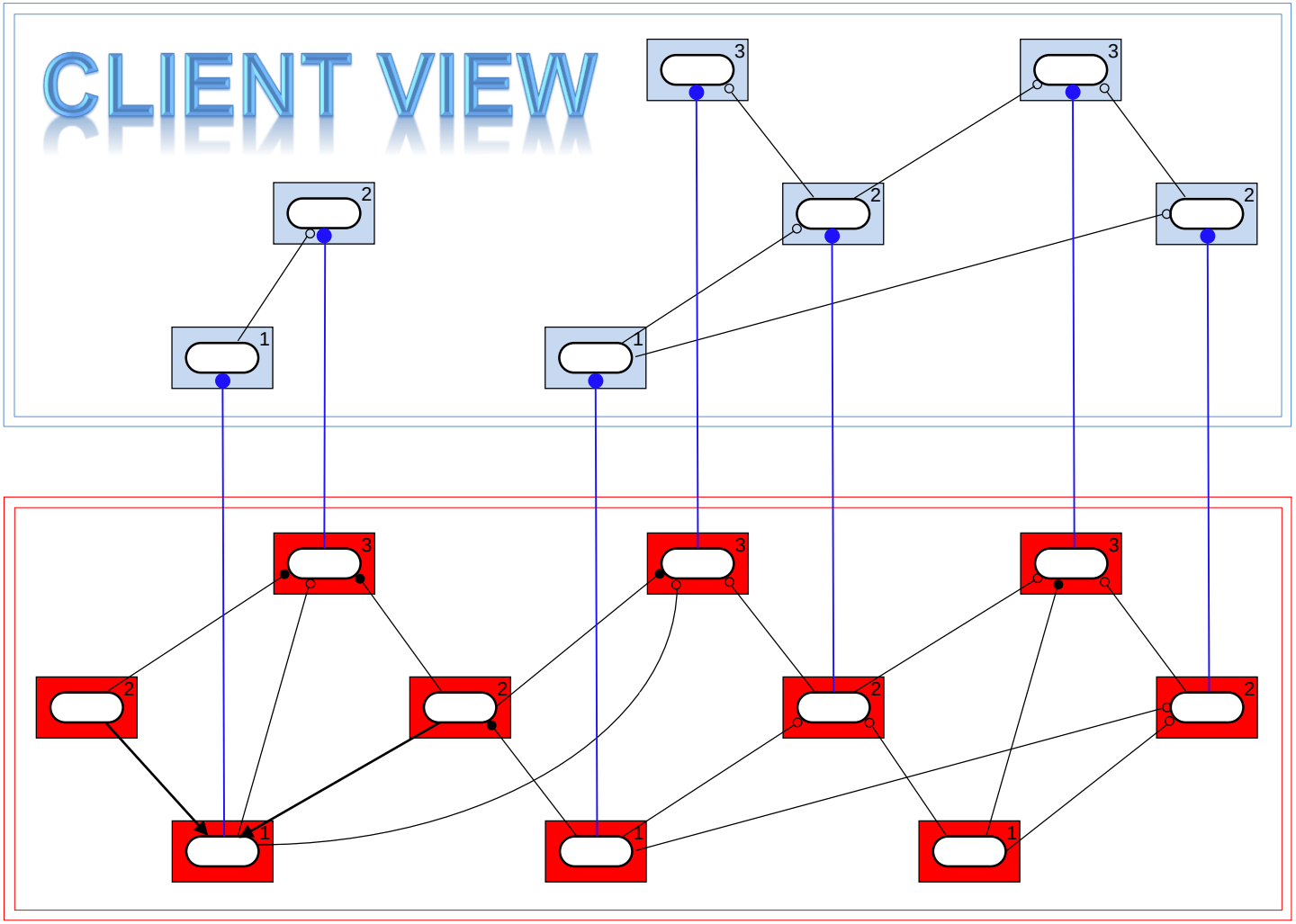
Escalating Encapsulation

Wrapper Package



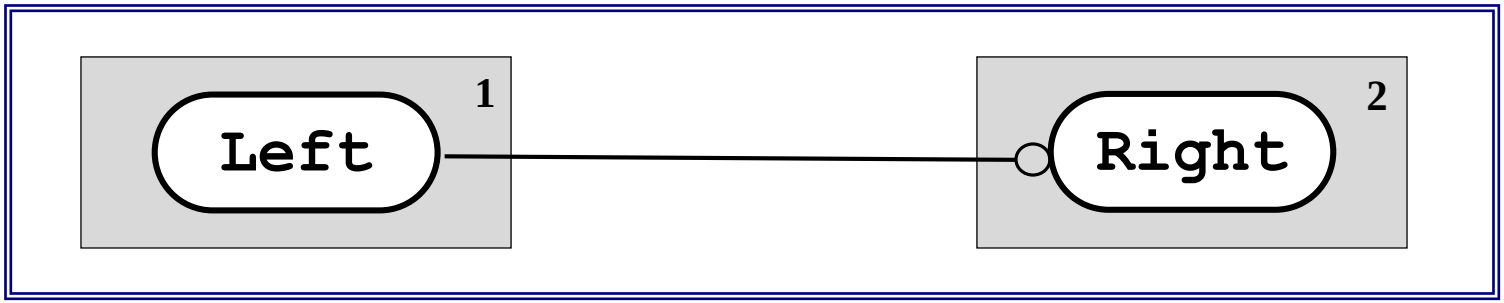
Escalating Encapsulation

Wrapper Package



Escalating Encapsulation

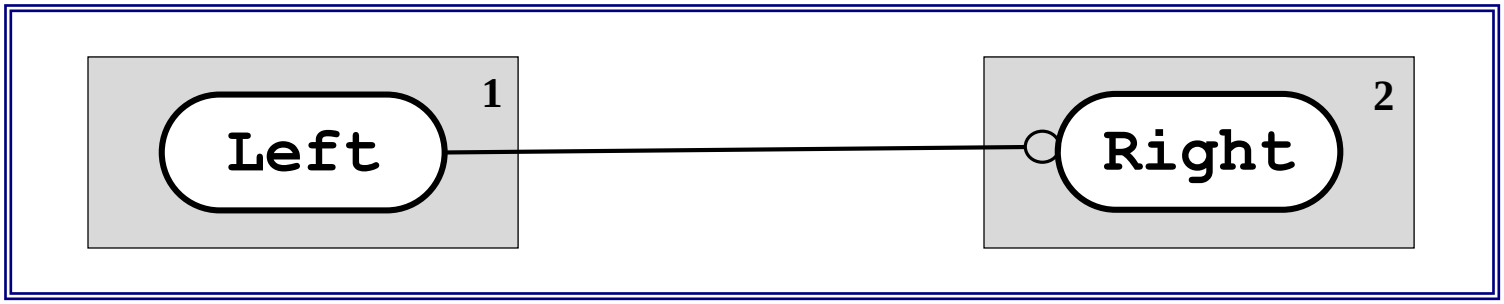
Wrapper Package



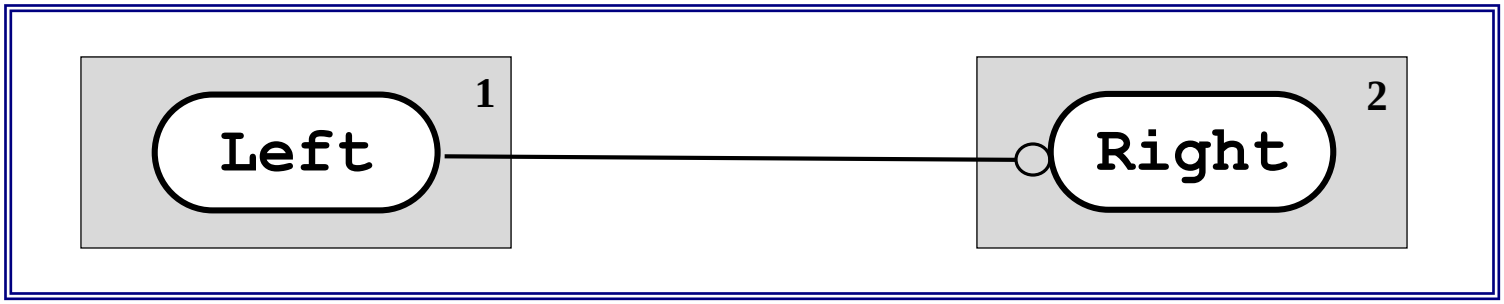
syst

Escalating Encapsulation

Wrapper Package



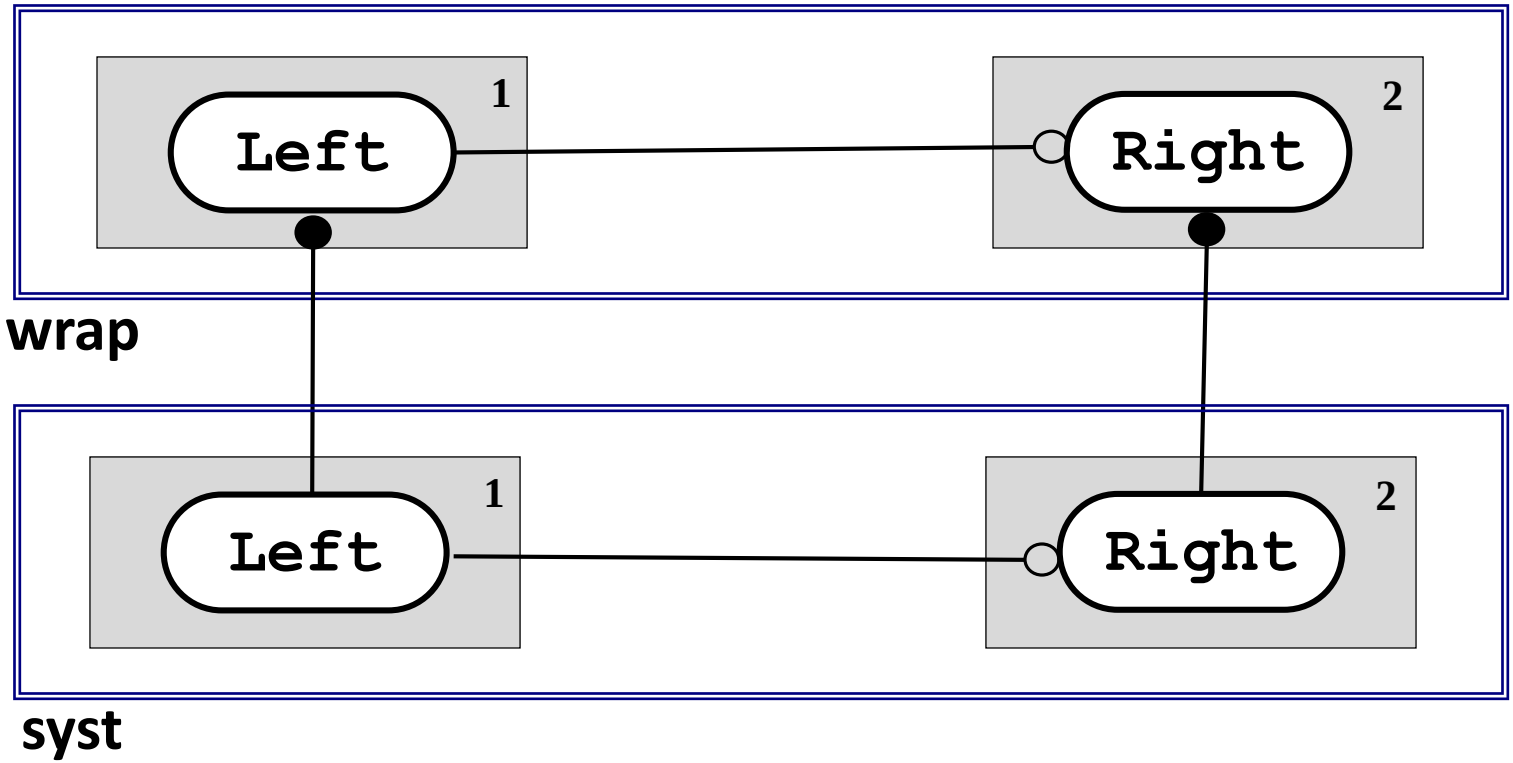
wrap



syst

Escalating Encapsulation

Wrapper Package

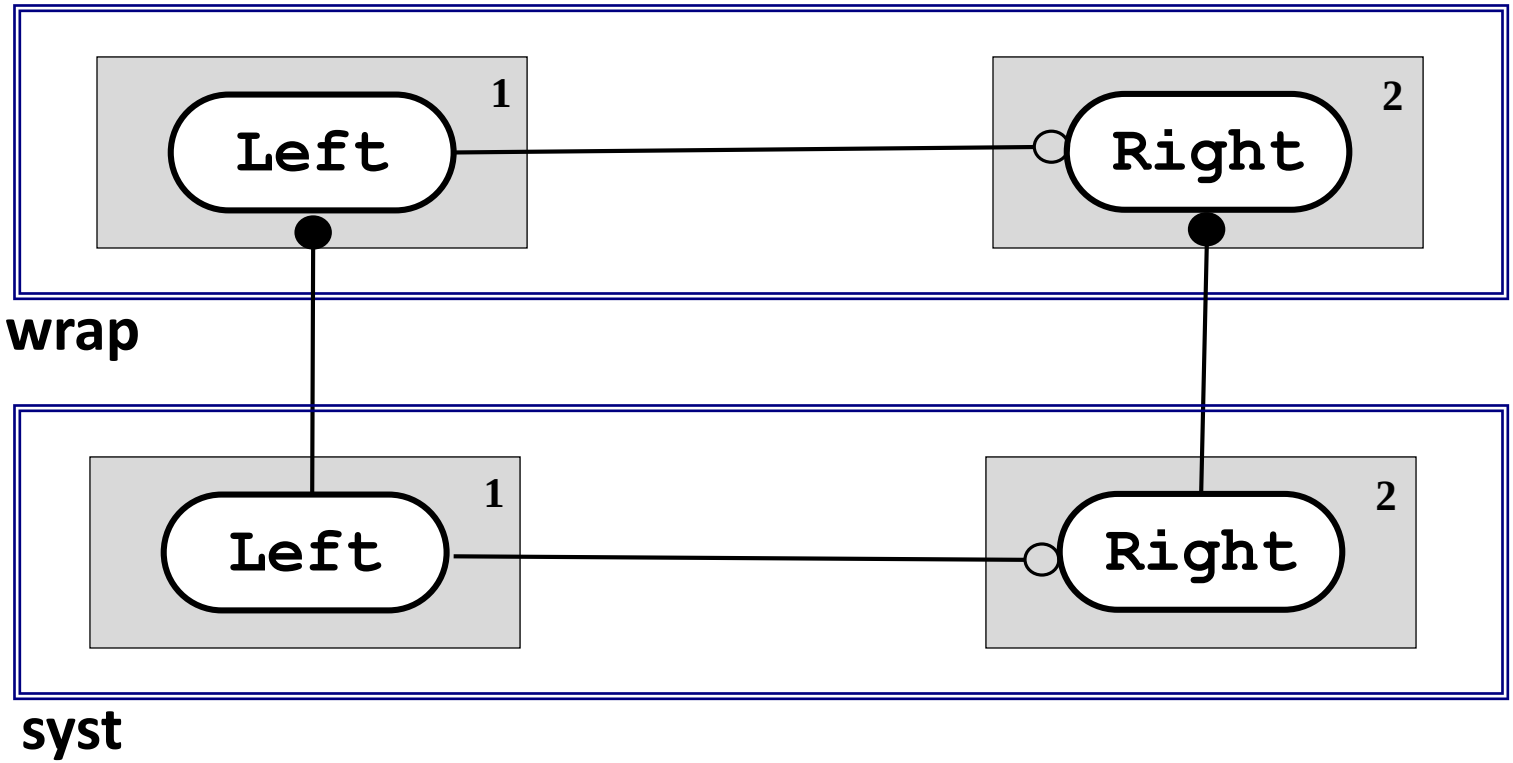


Escalating Encapsulation

Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

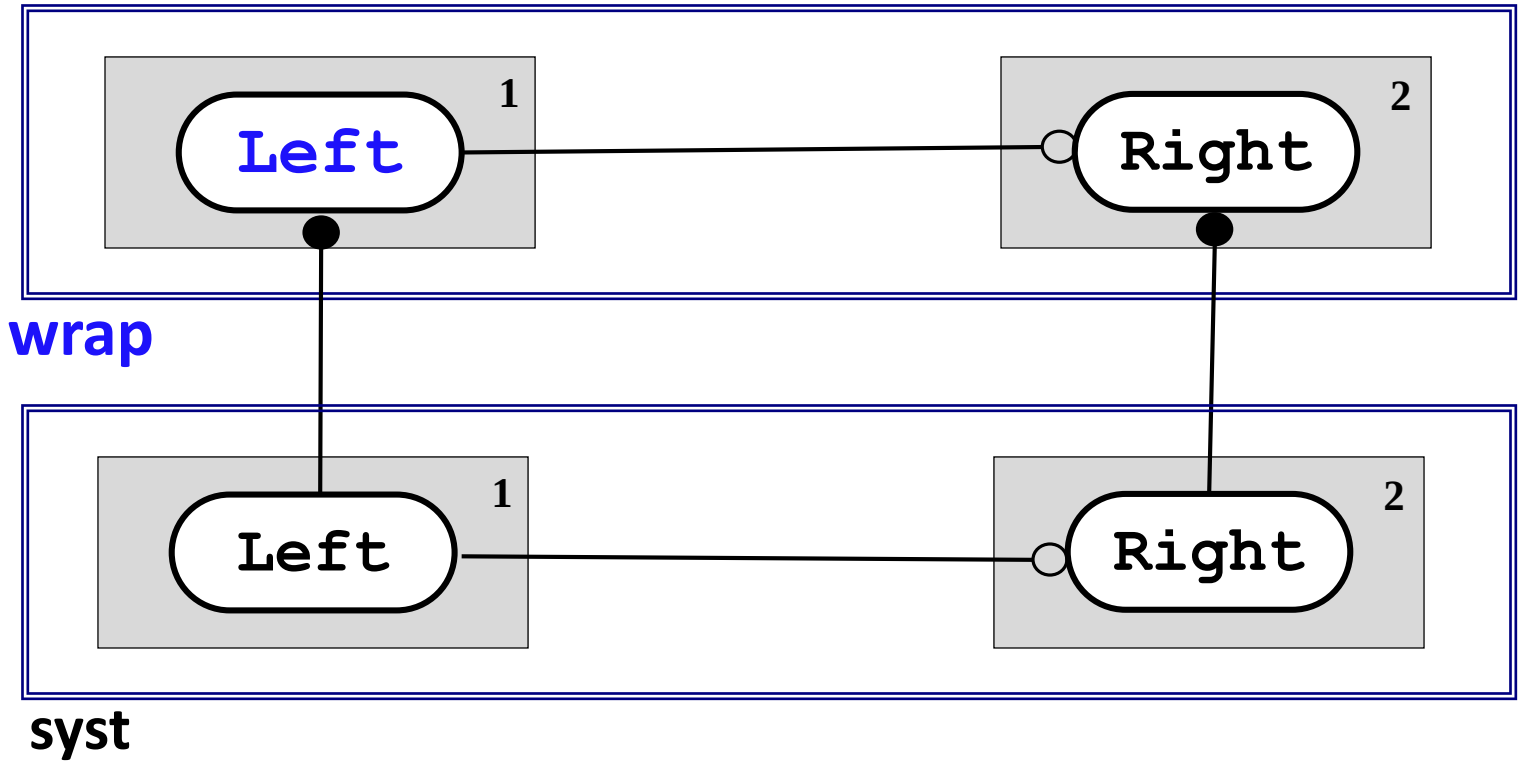


Escalating Encapsulation

Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

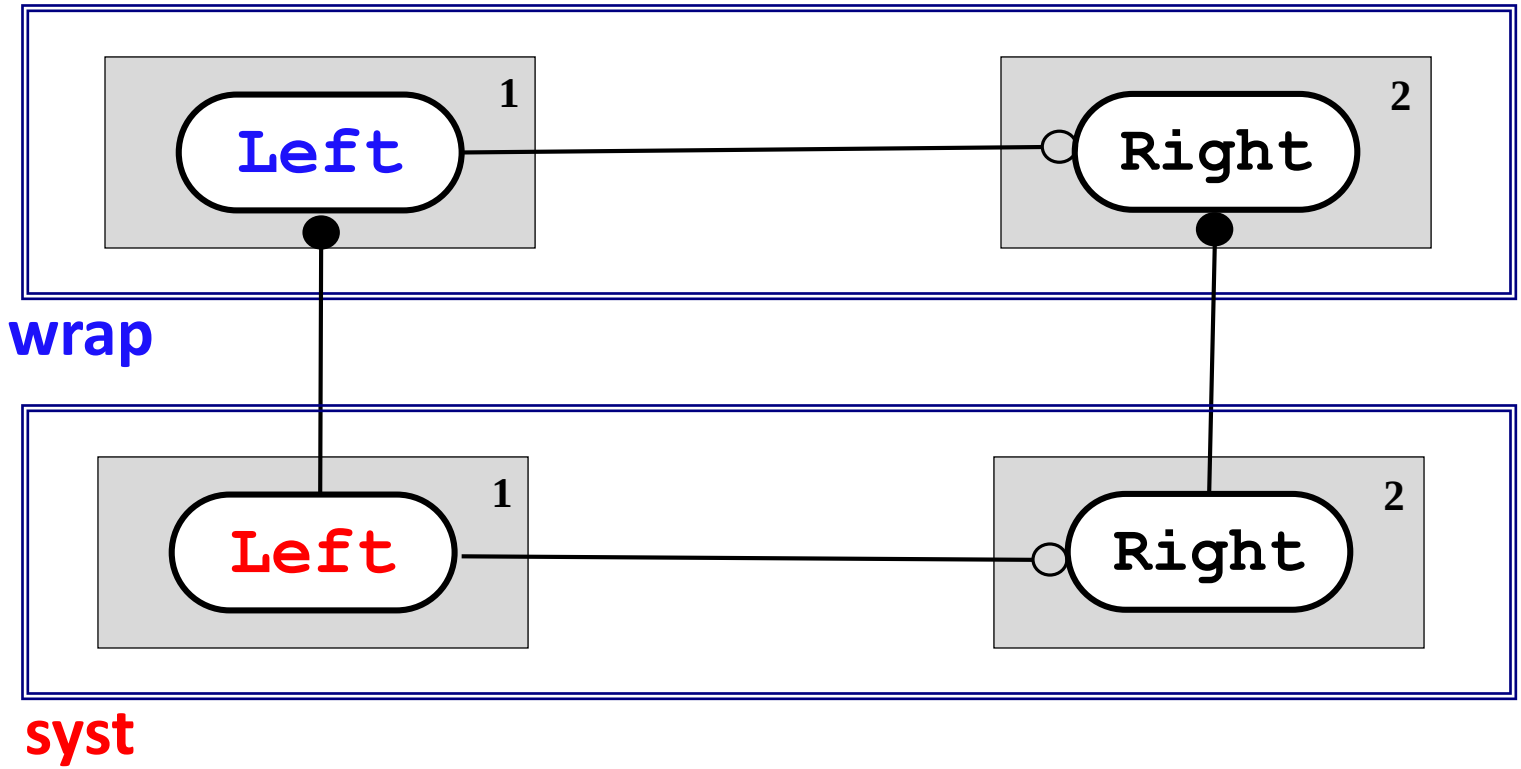


Escalating Encapsulation

Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

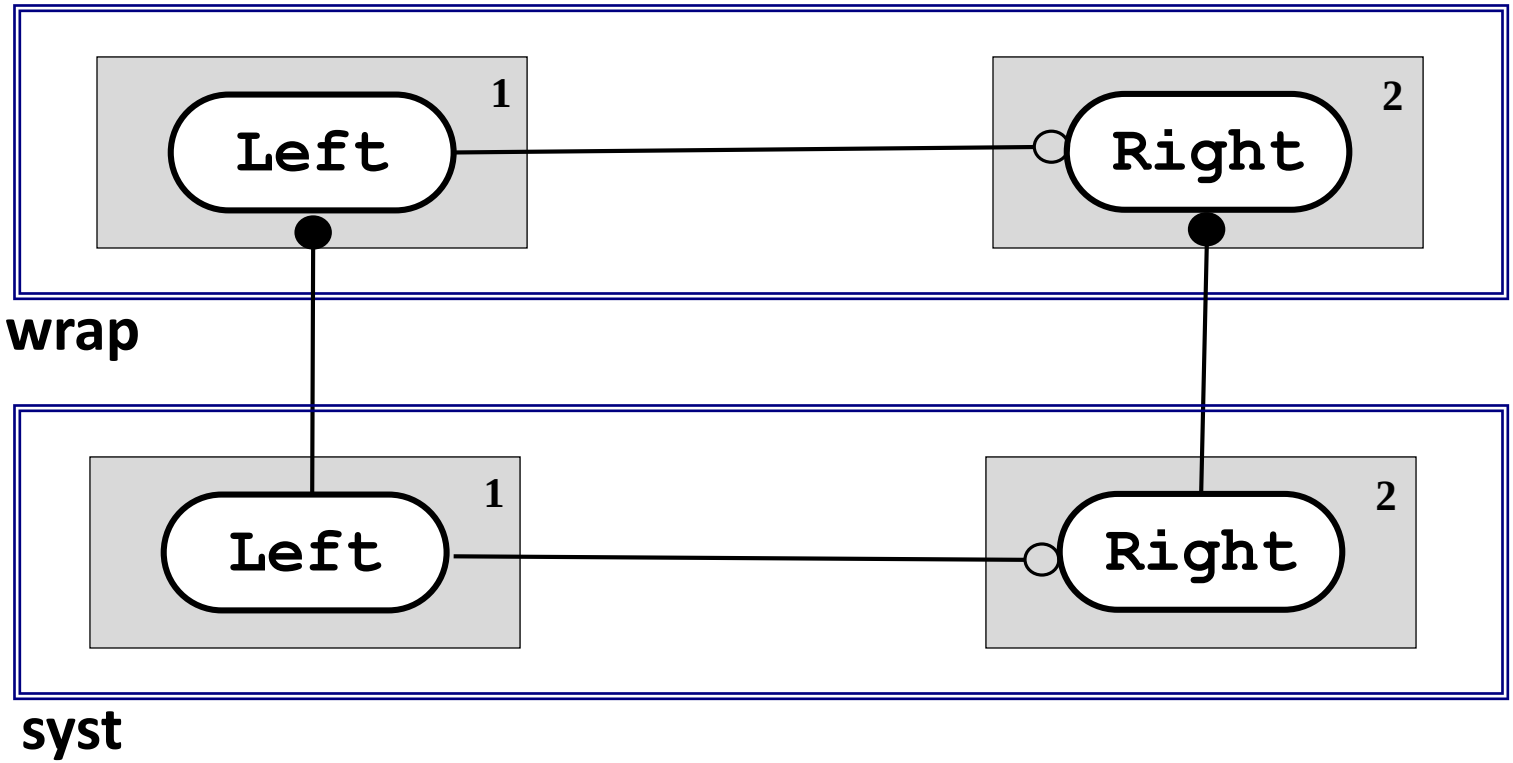


Escalating Encapsulation

Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

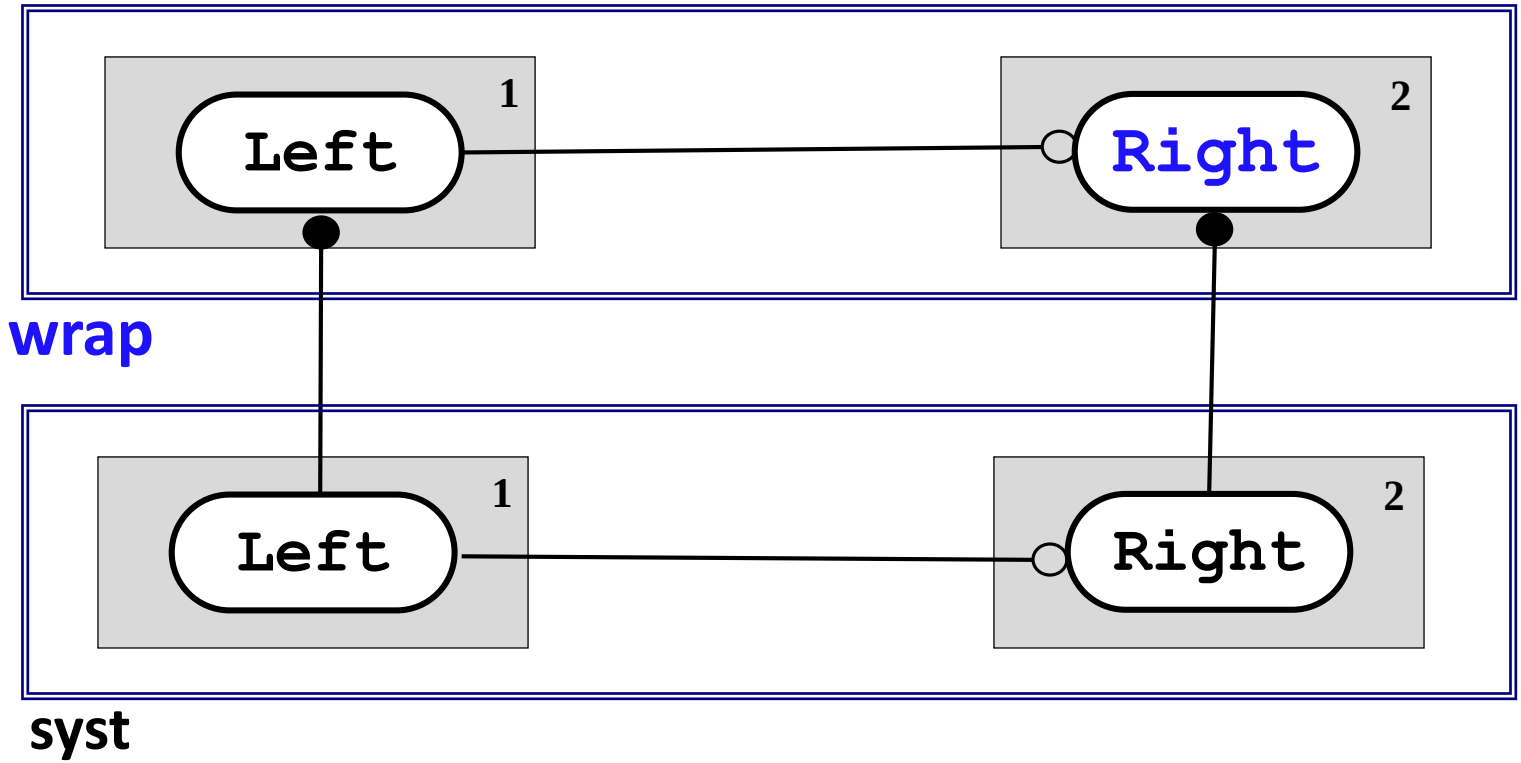


Escalating Encapsulation

Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

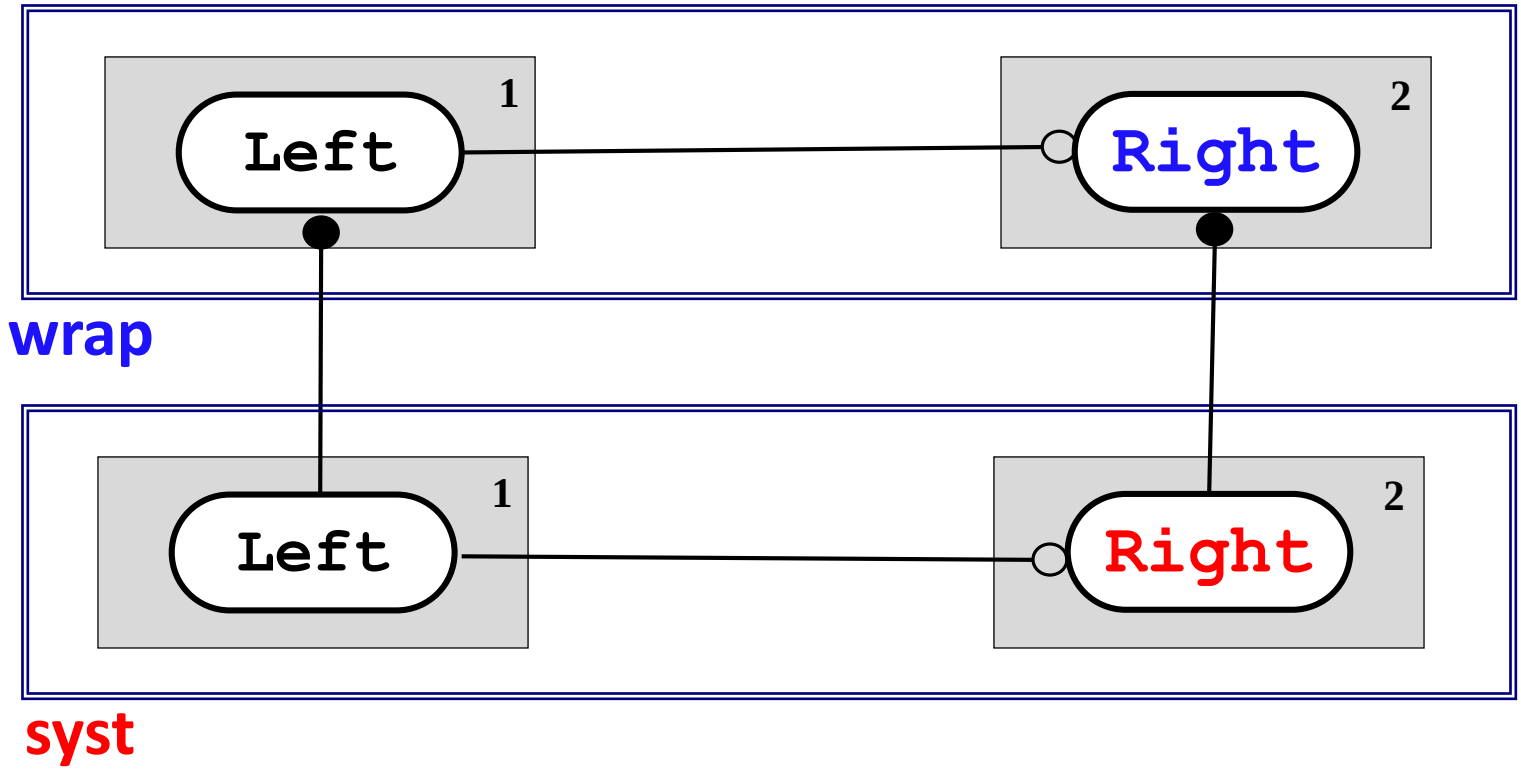


Escalating Encapsulation

Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

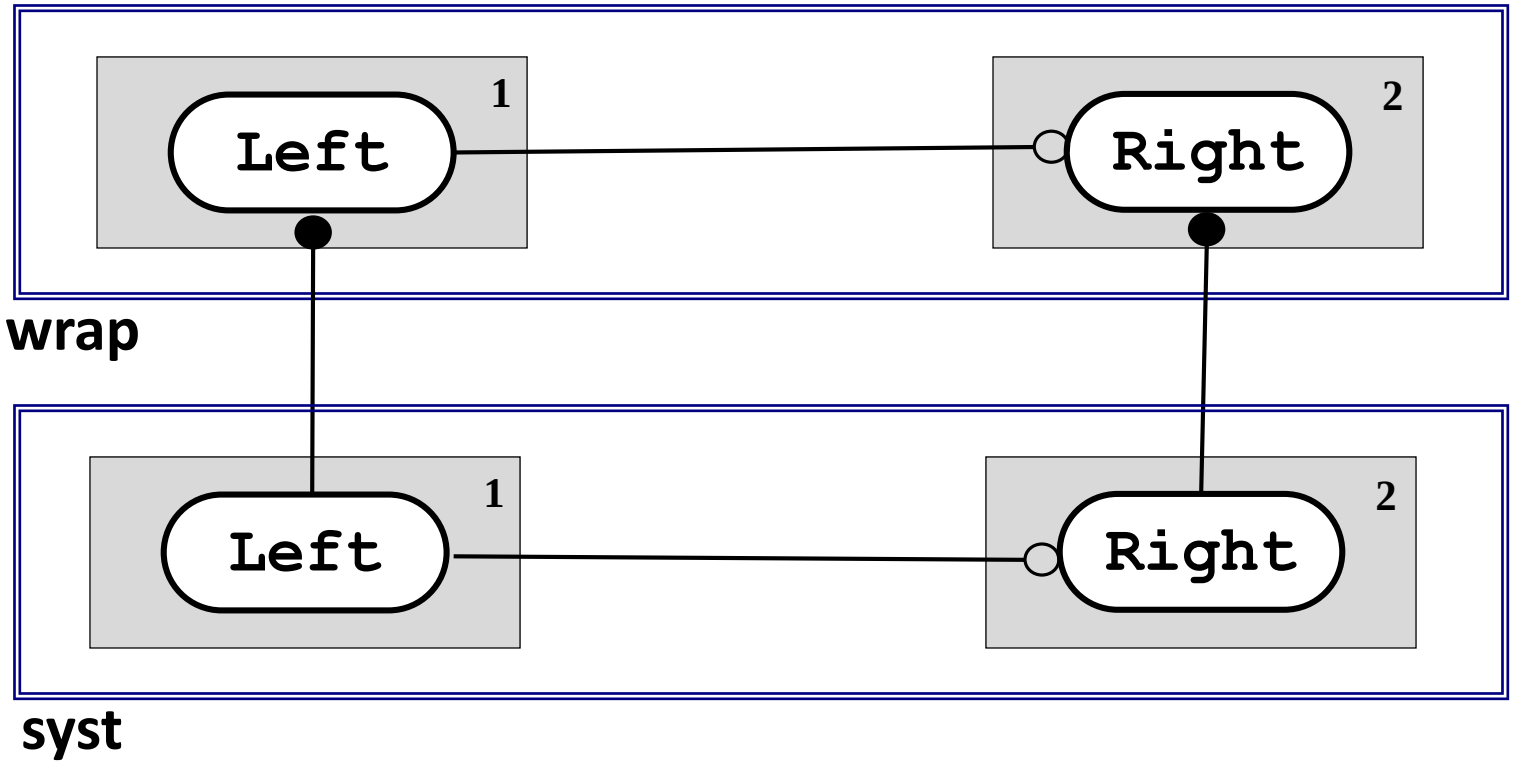


Escalating Encapsulation

Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

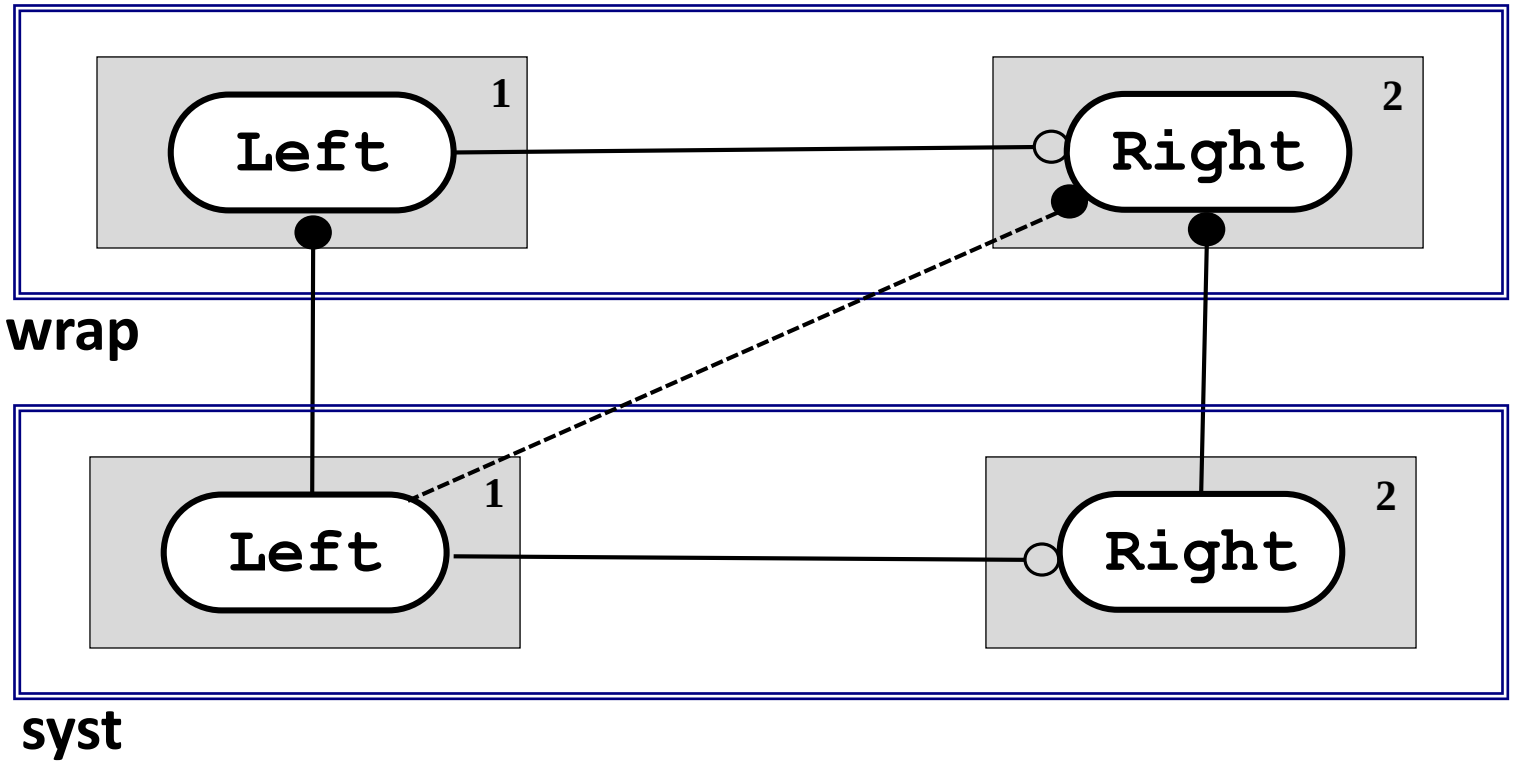


Escalating Encapsulation

Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

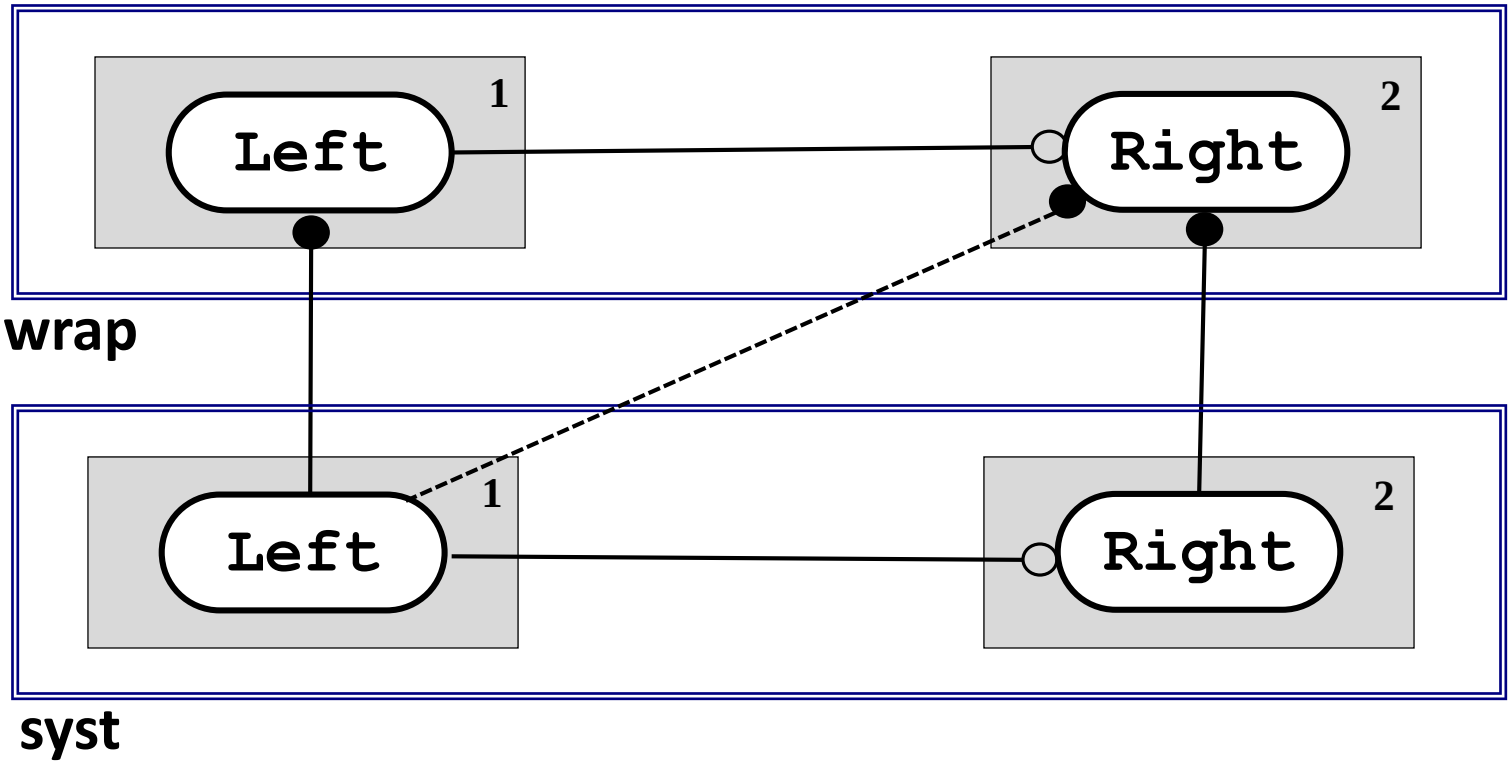
```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```



Escalating Encapsulation

Wrapper Package

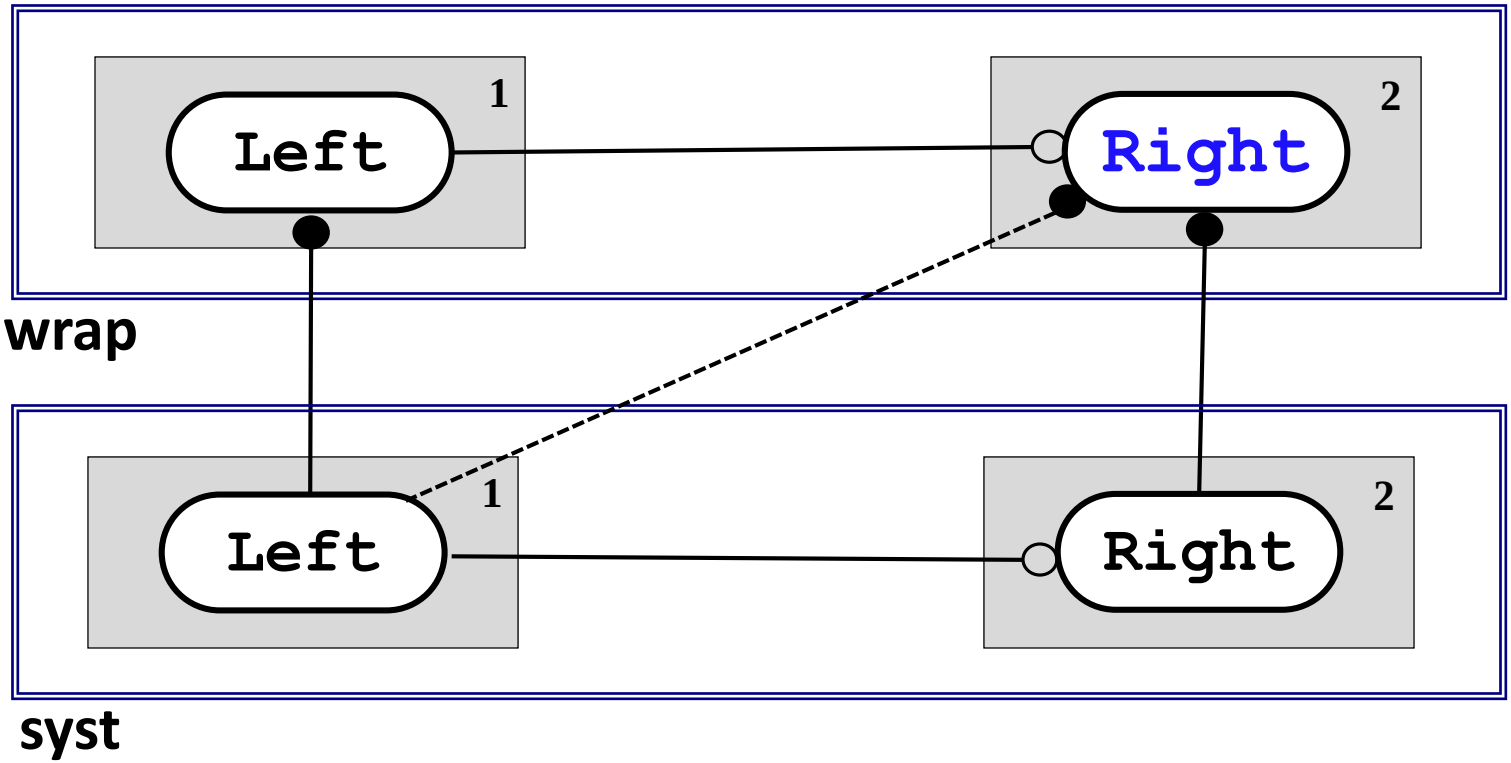
```
// wrap_right.cpp
// …
void wrap_Right::someFunction(const wrap_Left& v) {
  const syst_Left& vImp = *reinterpret_cast<syst_Left *>(&v);
  d_imp.someFunction(vImp);
```



Escalating Encapsulation

Wrapper Package

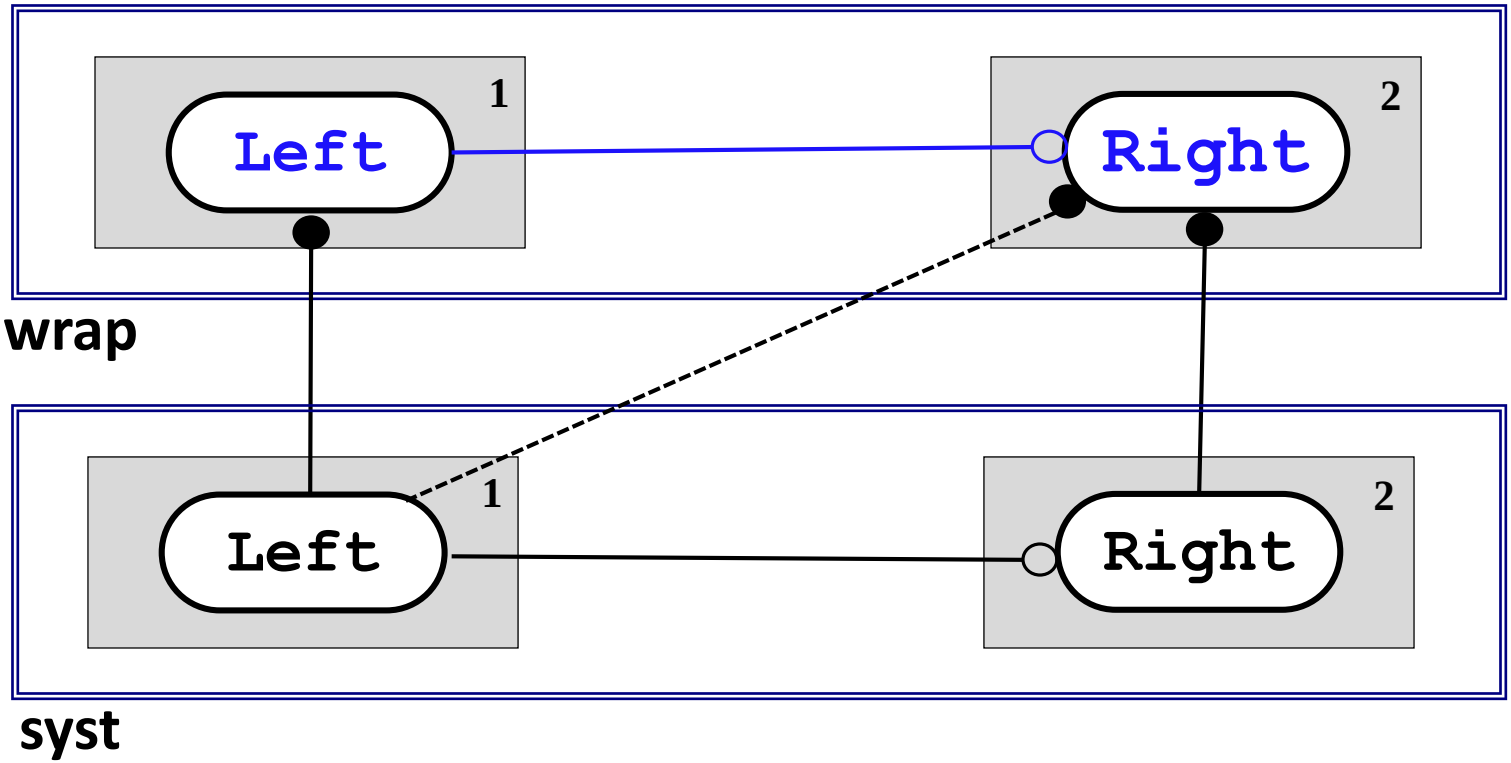
```
// wrap_right.cpp
// …
void wrap_Right::someFunction(const wrap_Left& v) {
  const syst_Left& vImp = *reinterpret_cast<syst_Left *>(&v);
  d_imp.someFunction(vImp);
```



Escalating Encapsulation

Wrapper Package

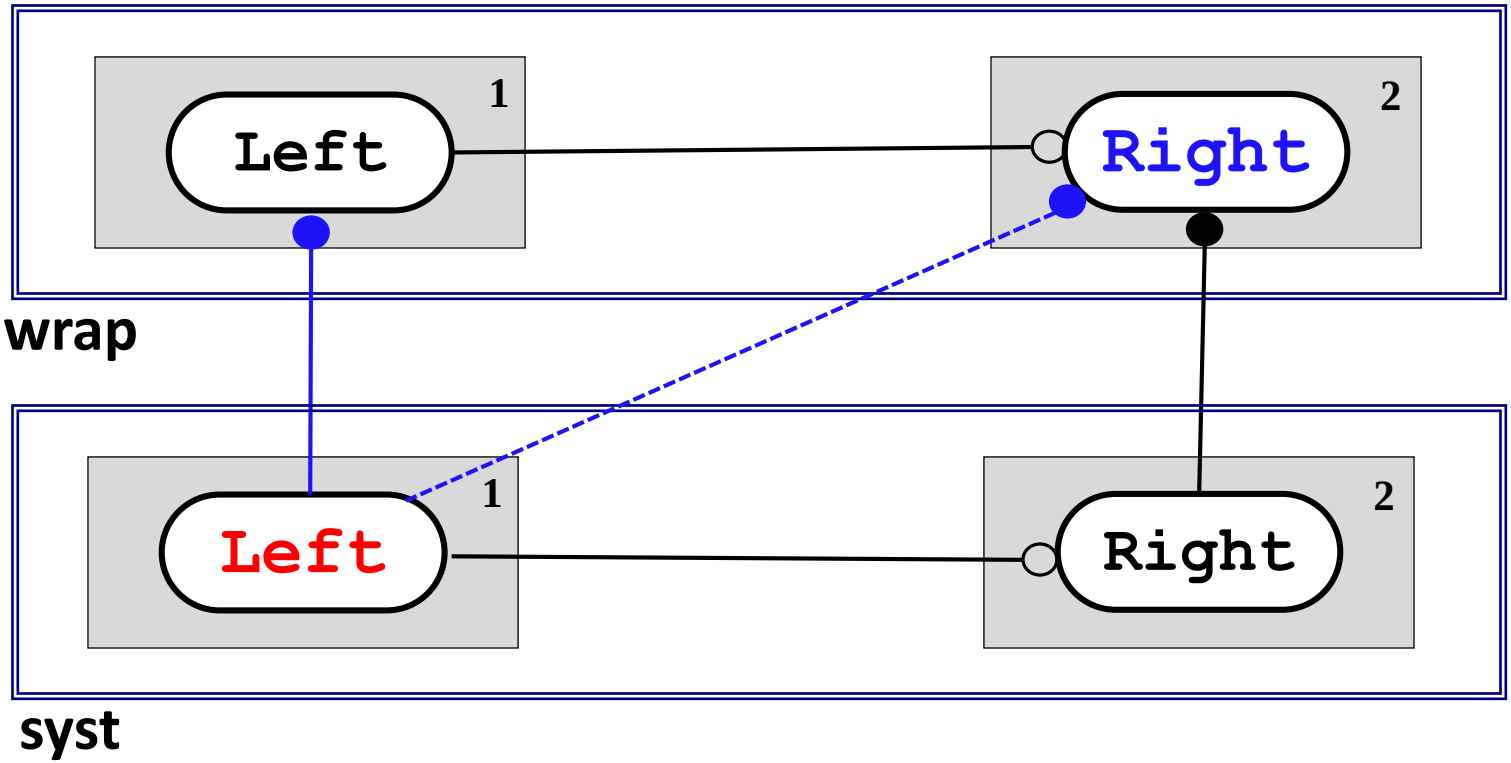
```
// wrap_right.cpp
// …
void wrap_Right::someFunction(const wrap_Left& v) {
  const syst_Left& vImp = *reinterpret_cast<syst_Left *>(&v);
  d_imp.someFunction(vImp);
```



Escalating Encapsulation

Wrapper Package

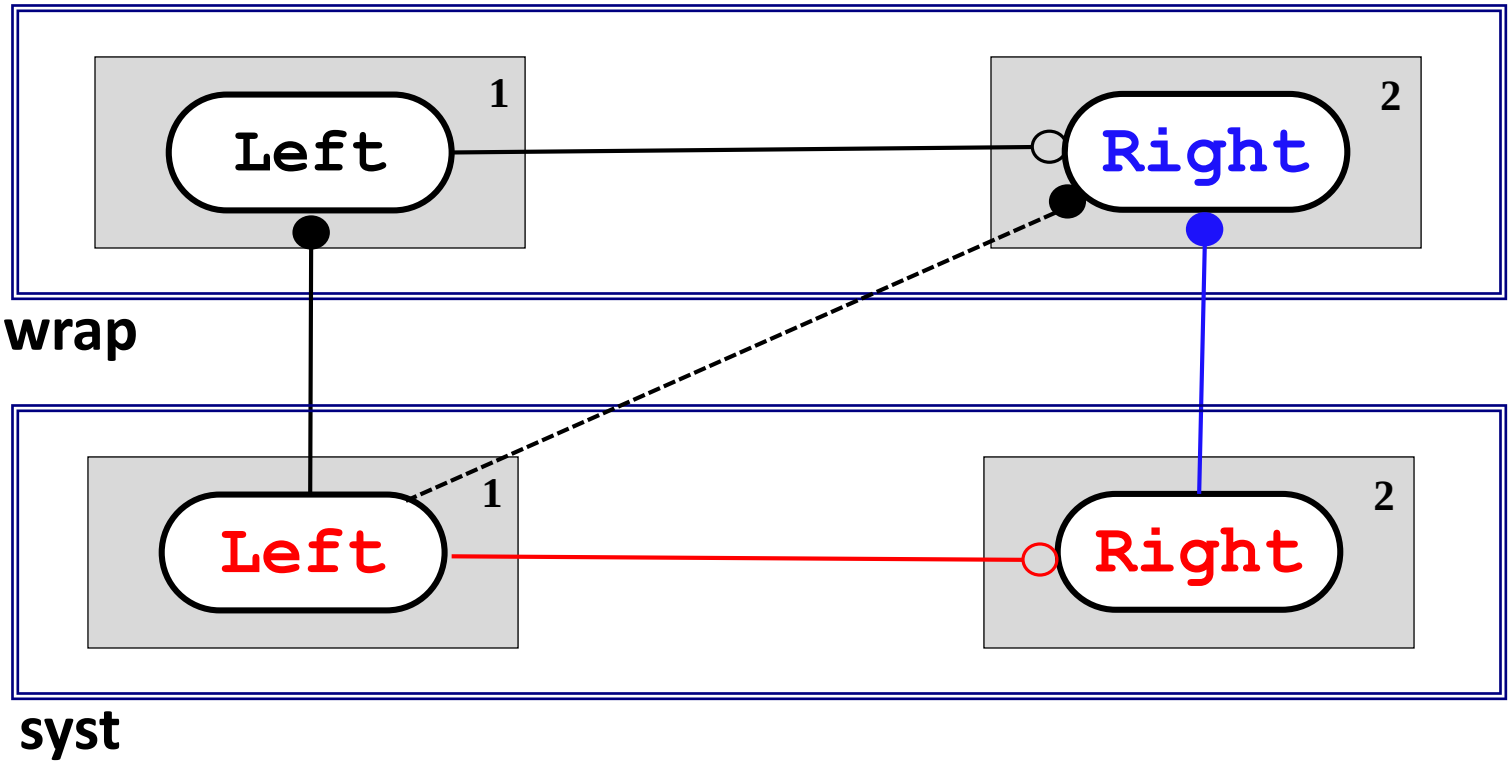
```
// wrap_right.cpp
// …
void wrap_Right::someFunction(const wrap_Left& v) {
  const syst_Left& vImp = *reinterpret_cast<syst_Left *>(&v);
  d_imp.someFunction(vImp);
```



Escalating Encapsulation

Wrapper Package

```
// wrap_right.cpp
// …
void wrap_Right::someFunction(const wrap_Left& v) {
  const syst_Left& vImp = *reinterpret_cast<syst_Left *>(&v);
  d_imp.someFunction(vImp);
```



Escalating Encapsulation

Discussion?

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/data 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 a 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.

End of Section

Questions?

What Questions are we Answering?

- How are components aggregated into larger physical units?
- How many levels of physical aggregation do we employ?
- How are component package names restricted physically?
- What do *levelize*, *levelizable*, and *levelization* mean?
- What does the *escalation* levelization technique involve?
- What does *multi-component wrapper* (MCW) delineate?
- Why is MCW difficult to achieve properly in classical C++?
- What specific MCW goals would we want C++ to support?

What Questions are we Answering?

- How are components aggregated into larger physical units?
- How many levels of physical aggregation do we employ?
- How are component package names restricted physically?
- What do *levelize*, *levelizable*, and *levelization* mean?
- What does the *escalation* levelization technique involve?
- What does *multi-component wrapper* (MCW) delineate?
- Why is MCW difficult to achieve properly in classical C++?
- What specific MCW goals would we want C++ to support?

What Questions are we Answering?

- How are components aggregated into larger physical units?
- How many levels of physical aggregation do we employ?
- How are component package names restricted physically?
- What do *levelize*, *levelizable*, and *levelization* mean?
- What does the *escalation* levelization technique involve?
- What does *multi-component wrapper* (MCW) delineate?
- Why is MCW difficult to achieve properly in classical C++?
- What specific MCW goals would we want C++ to support?

Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Introduce the Notion of a `module` in C++

   Requirements: Comparison with Conventional Headers

3. Achieving Physical Aggregation in C++ Today

   Organizing Components into Packages and Package Groups

4. Packaging Libraries Using C++ Modules

   Abstraction: Providing Refined Views on Existing Software

Outline

1. Review of Elementary Physical Design
   - Components, Modularity, Physical Dependencies
2. Introduce the Notion of a `module` in C++
   - Requirements: Comparison with Conventional Headers
3. Achieving Physical Aggregation in C++ Today
   - Organizing Components into Packages and Package Groups
4. Packaging Libraries Using C++ Modules
   - Abstraction: Providing Refined Views on Existing Software

Introduction

(Effective Use of Fine-Grained Filtering)

Under Construction

End of Section

Questions?

What Questions are we Answering?

- How do modules help us to better package our software?

What Questions are we Answering?

- How do modules help us to better package our software?

Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Introduce the Notion of a `module` in C++

   Requirements: Comparison with Conventional Headers

3. Achieving Physical Aggregation in C++ Today

   Organizing Components into Packages and Package Groups

4. Packaging Libraries Using C++ Modules

   Abstraction: Providing Refined Views on Existing Software

Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.

Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.
- **No** *cyclic* dependencies/*long-distance* friendships.

Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.
- **No** *cyclic* dependencies/*long-distance* friendships.
- Colocate logical constructs only with good reason:
  i.e., friendship; cycles; parts-of-whole; flea-on-elephant.

Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.
- **No** *cyclic* dependencies/*long-distance* friendships.
- Colocate logical constructs only with good reason:
  i.e., friendship; cycles; parts-of-whole; flea-on-elephant.
- Put a `#include` in a header only with good reason:
  i.e., *Is-A*, *Has-A*, `inline`, `enum`, `typedef`-to-template.

Conclusion

The End