

Proper Inheritance

John Lakos

Wednesday, April 20, 2016

Copyright Notice

© 2016 Bloomberg L.P. Permission is granted to copy, distribute, and display this material, and to make derivative works and commercial use of it. The information in this material is provided "AS IS", without warranty of any kind. Neither Bloomberg nor any employee guarantees the correctness or completeness of such information. Bloomberg, its employees, and its affiliated entities and persons shall not be liable, directly or indirectly, in any way, for any inaccuracies, errors or omissions in such information. Nothing herein should be interpreted as stating the opinions, policies, recommendations, or positions of Bloomberg.

Abstract

All essential behavior of our software must be documented, and yet there are important advantages, with respect to development, verification and testing, performance, and stability, for leaving the behavior for some combinations of inputs and initial conditions undefined. What is and is not defined behavior should therefore be readily discernible from the contract, especially when creating contracts that must span classes related by inheritance.

In this two-part talk, we begin by first reviewing components, interfaces and contracts in general, and the significance of *narrow* versus *wide* contracts. In the second part, we go on to explore three kinds of inheritance: (1) **Interface Inheritance** resulting from pure-virtual functions, (2) **Structural Inheritance** resulting from non-virtual functions, and (3) **Implementation Inheritance** resulting from non-pure virtual functions. Proper contracts involving each of these distinct forms have different criteria that must be addressed. The three kinds of inheritance are compared, and their relative utility is explained. What's more, several common uses of inheritance that are provably improper are summarily debunked.

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 **logical functionality** within discrete, fine-grain **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-grain 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-grain physical components.
- It requires the designer to delineate logical behavior precisely, while managing the physical dependencies on other subordinate components.
- It requires attention to numerous **logical** and **physical** rules that govern sound software design.

Purpose of this Talk

Purpose of this Talk

Review:

Purpose of this Talk

Review:

1. Components –

Our fundamental unit of *logical* and *physical* design

Purpose of this Talk

Review:

1. Components –

Our fundamental unit of *logical* and *physical* design

2. Interfaces and contracts (in general)

Purpose of this Talk

Review:

1. Components –

Our fundamental unit of *logical* and *physical* design

2. Interfaces and contracts (in general)

3. *Narrow versus Wide* contracts (in particular)

Purpose of this Talk

Review:

1. Components –

Our fundamental unit of *logical* and *physical* design

2. Interfaces and contracts (in general)

3. *Narrow* versus *Wide* contracts (in particular)

4. Explore these basic ideas
in the context inheritance.

Outline

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

4. Proper Inheritance

Is-A for *Interface*, *Structural*, & *Implementation* Inheritance

Outline

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

4. Proper Inheritance

Is-A for *Interface*, *Structural*, & *Implementation* Inheritance

1. Components (review)

Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?



1. Components (review)

Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?

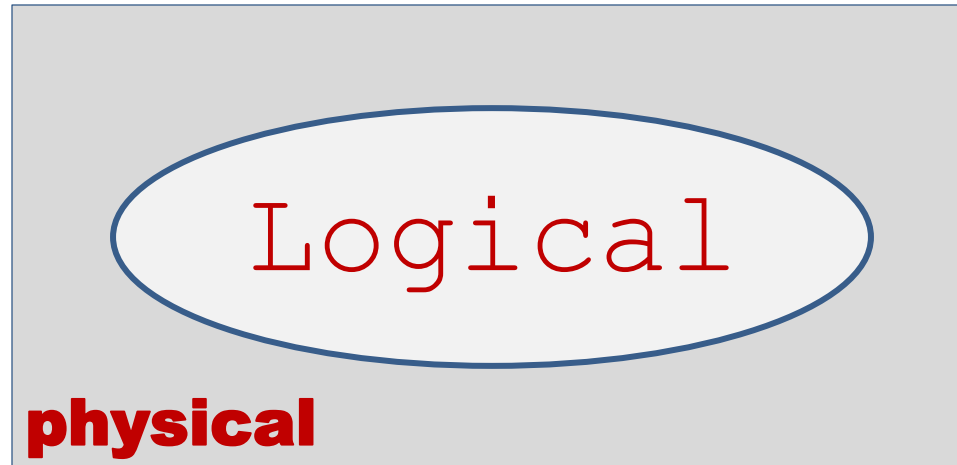


Logical: Classes and Functions

1. Components (review)

Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?



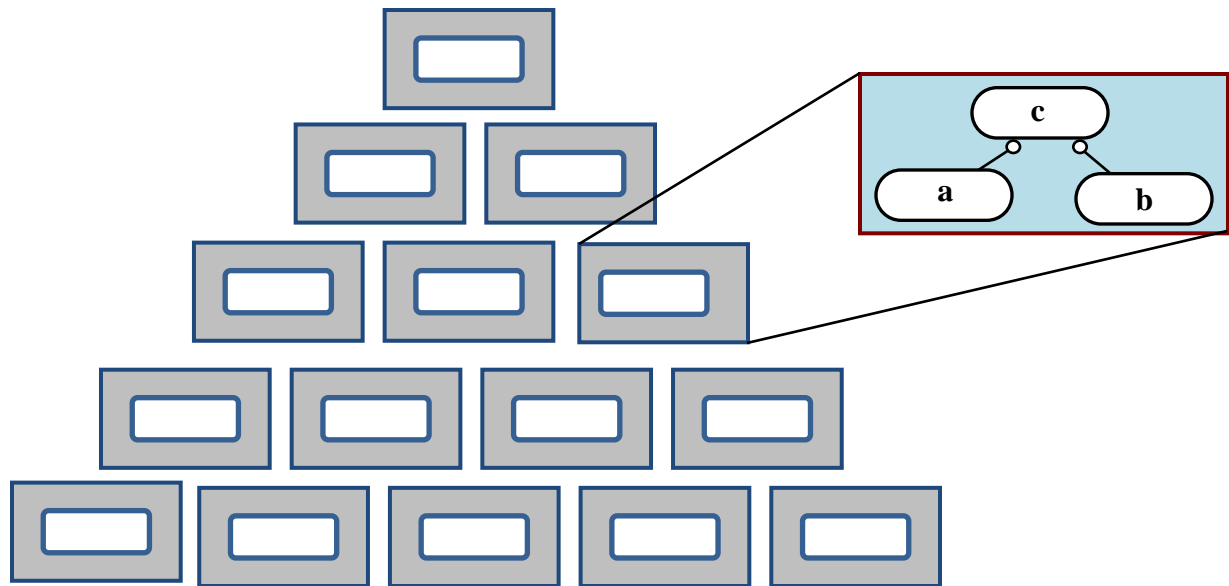
Logical: Classes and Functions

Physical: Files and Libraries

1. Components (review)

Logical versus Physical Design

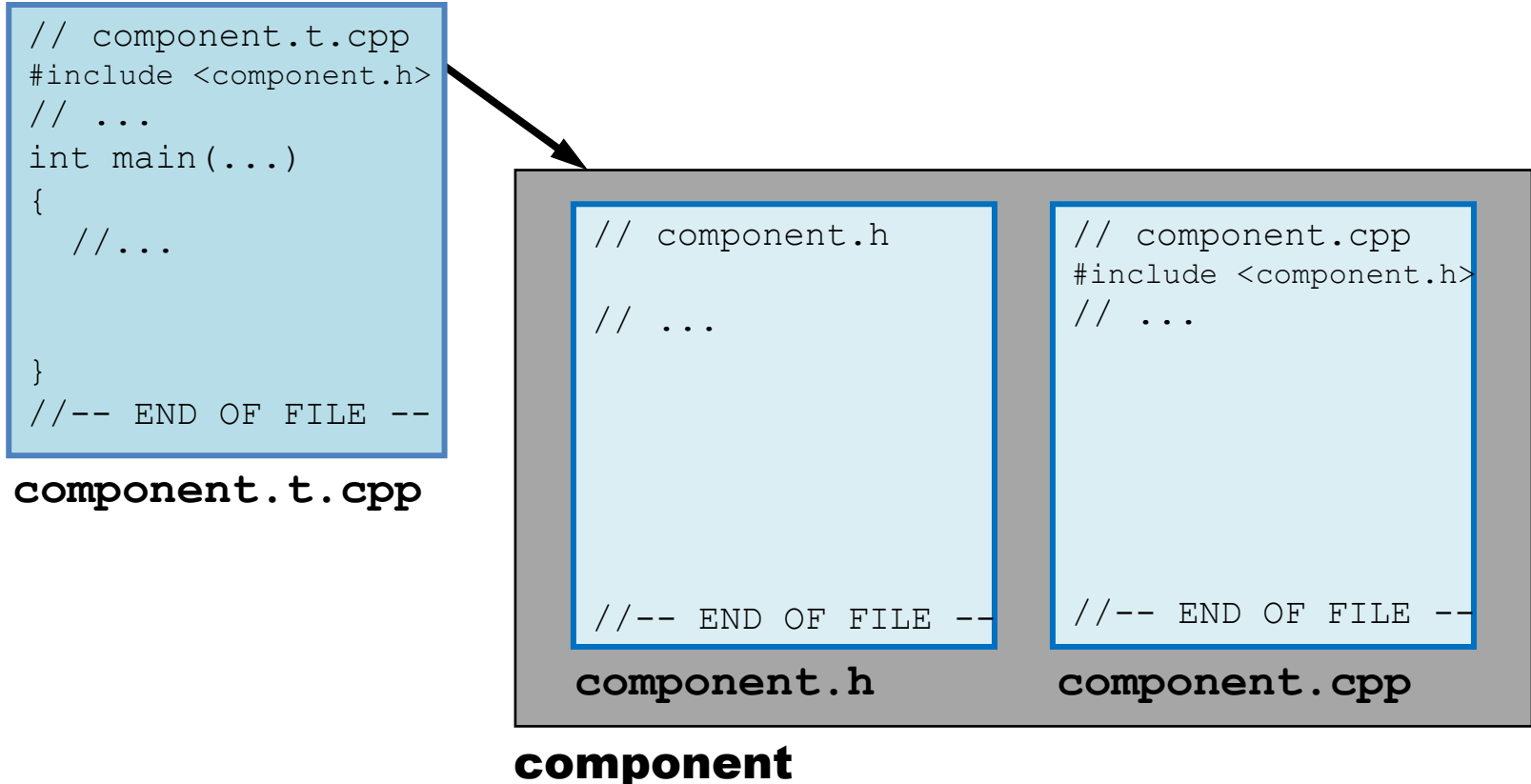
Logical content aggregated into a
Physical hierarchy of **components**



1. Components (review)

Component: Uniform Physical Structure

A Component Is Physical



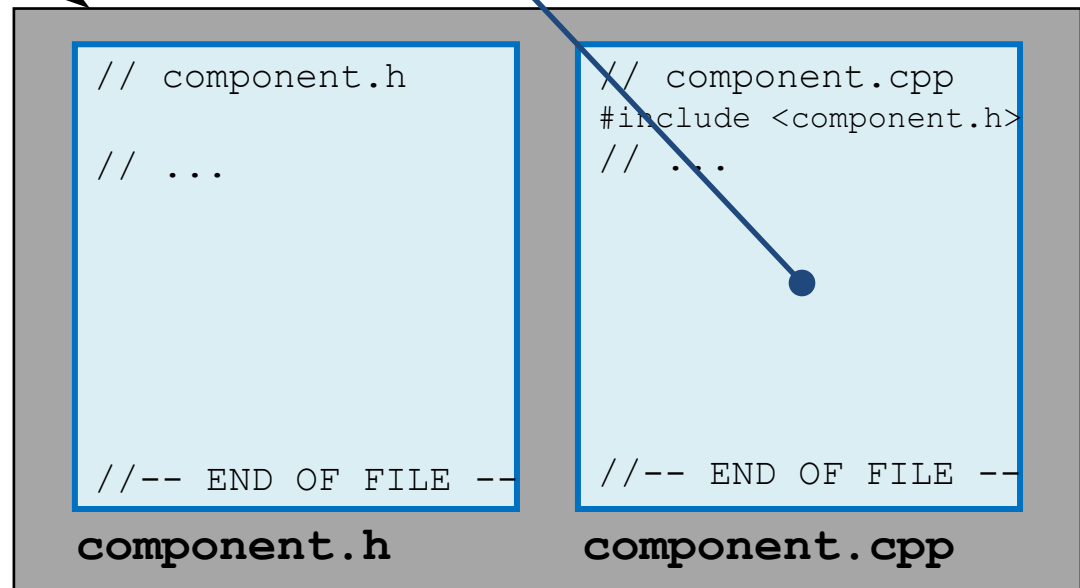
1. Components (review)

Component: Uniform Physical Structure

Implementation

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

component.t.cpp



component

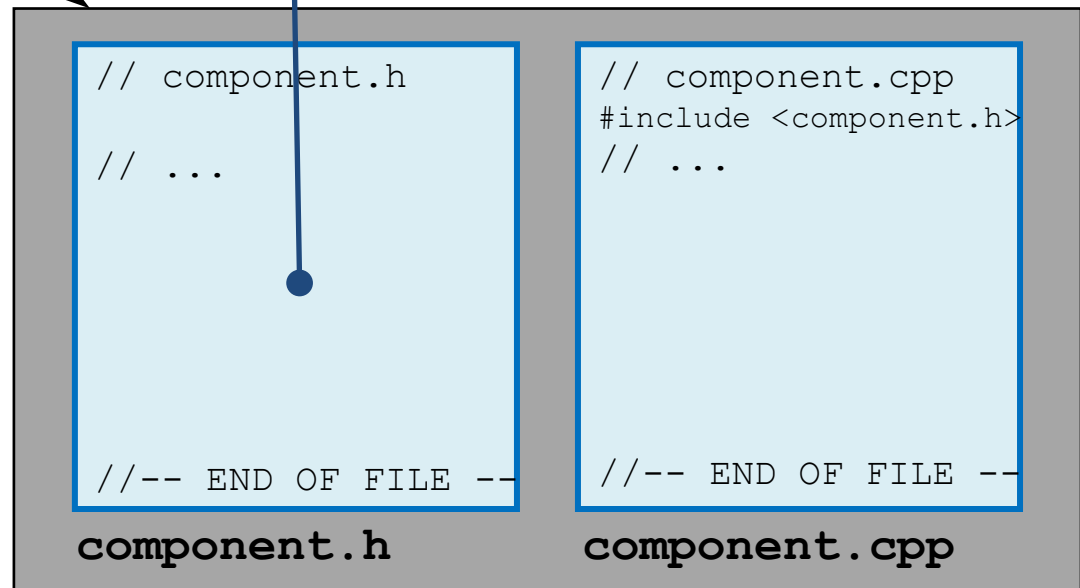
1. Components (review)

Component: Uniform Physical Structure

Header

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

component.t.cpp



component

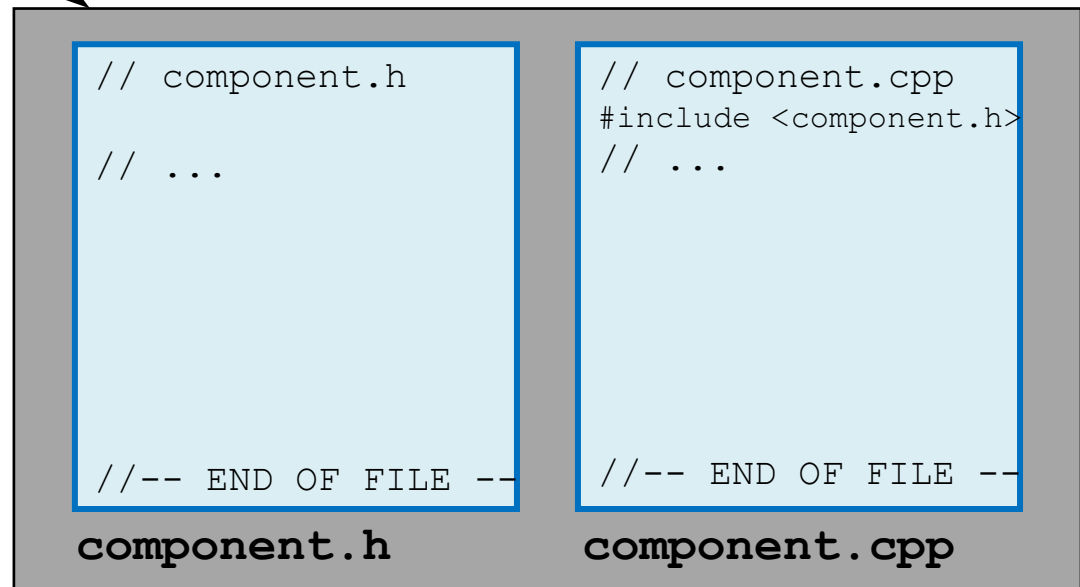
1. Components (review)

Component: Uniform Physical Structure

Test Driver

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

component.t.cpp



component.h

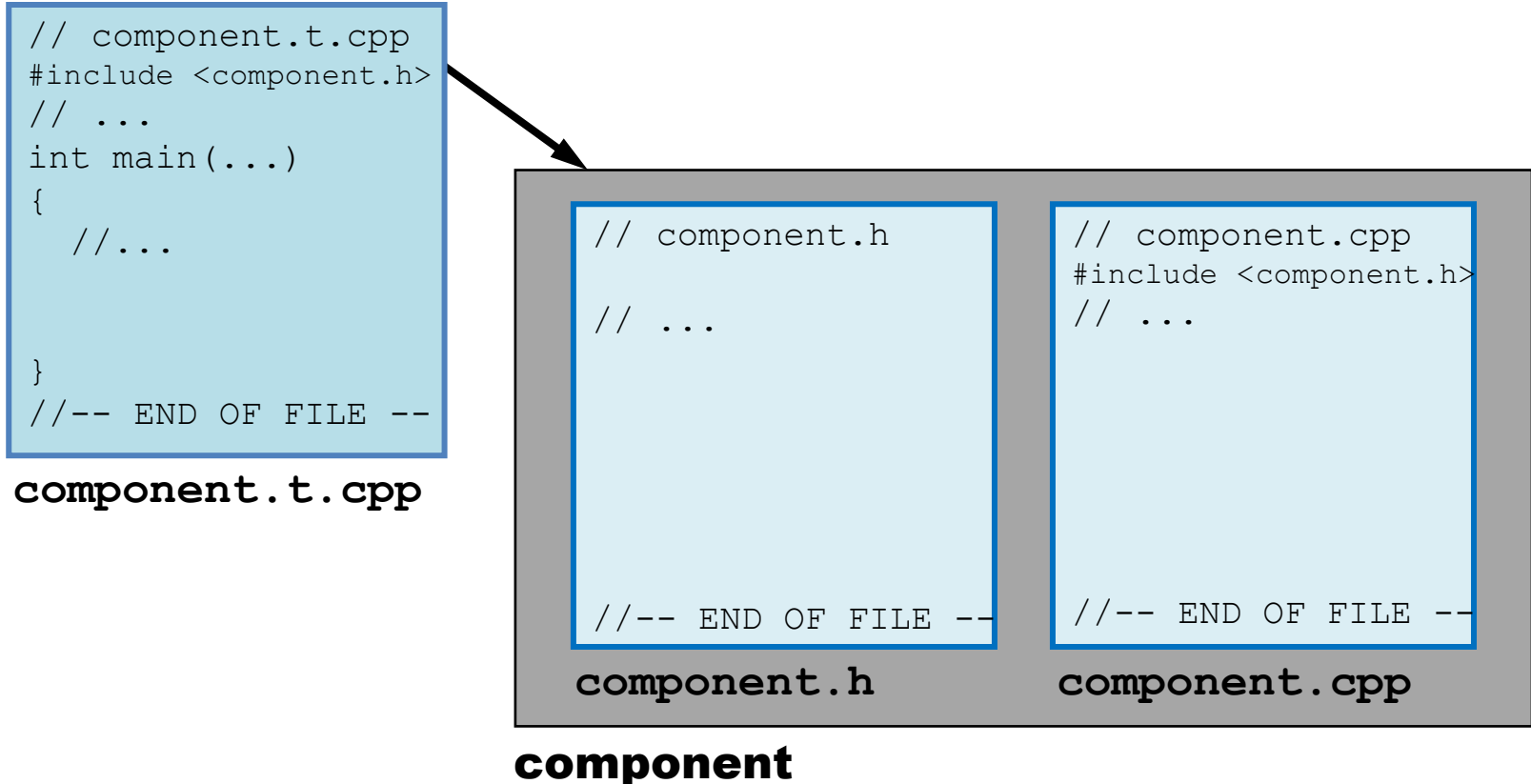
component.cpp

component

1. Components (review)

Component: Uniform Physical Structure

The Fundamental Unit of Design



1. Components (review)

Component: Not Just a .h/.cpp Pair



`my::Widget`

`my_widget`


1. Components (review)

Component: Not Just a .h/ .cpp Pair

There are four Properties...

1. Components (review)





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!

1. Components (review)

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.

1. Components (review)

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

We could easily
spend 20 minutes
on this slide alone!

Avoid Global
Namespace
Pollution

Achieve
Logical/Physical
Modularity

Enable *Efficient*
Extraction of
Physical
Dependencies

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 *linkage* declared in a `.h` file (if defined at all) are defined within the component.
4. A component's functionality is accessed via a `#include` of its header, and never via a forward (`extern`) declaration.

Avoid Global
Namespace
Pollution

Achieve
Logical/Physical
Modularity

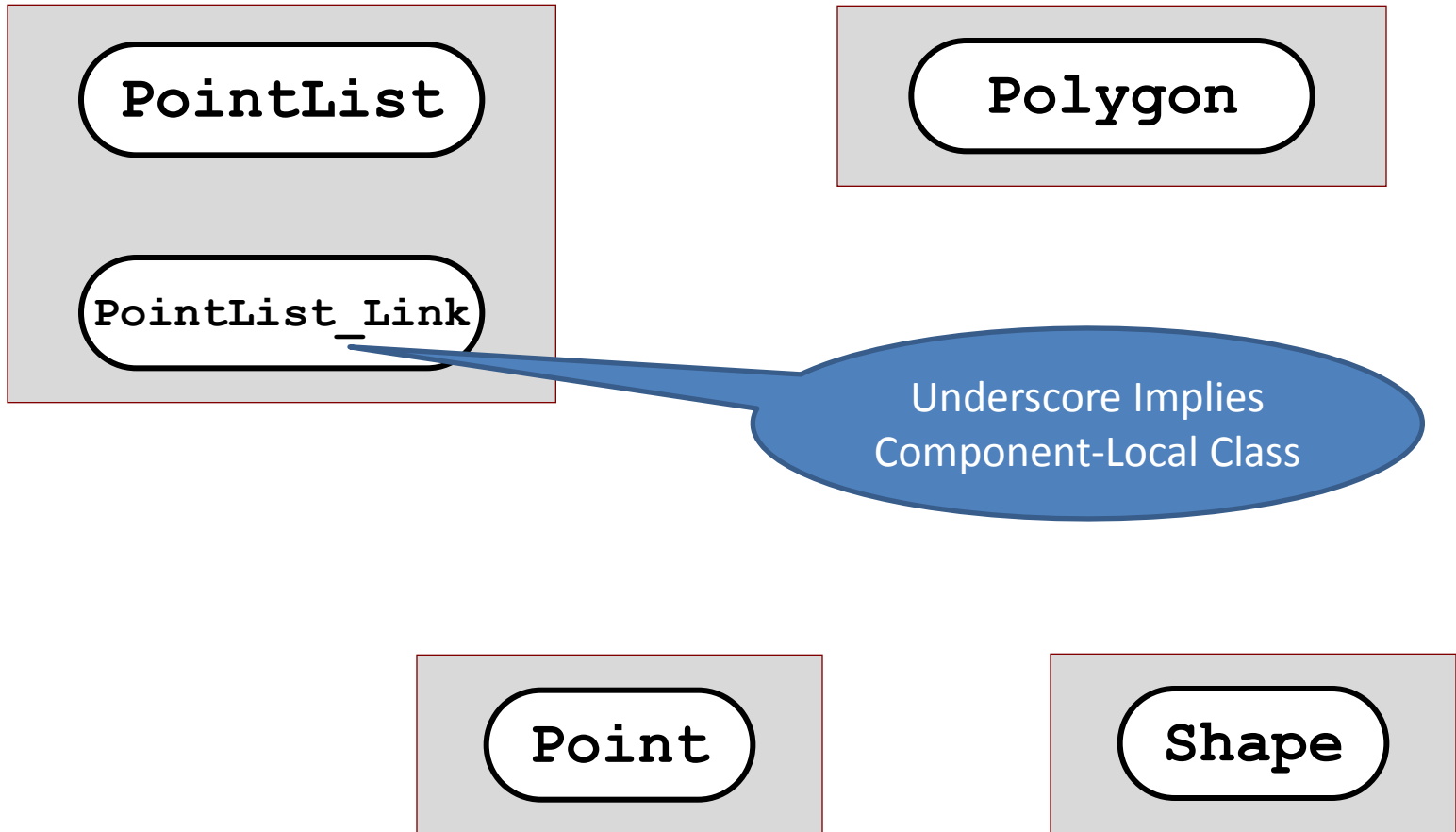
Enable *Efficient*
Extraction of
Physical
Dependencies

For much more see:

ADVANCED LEVELIZATION TECHNIQUES

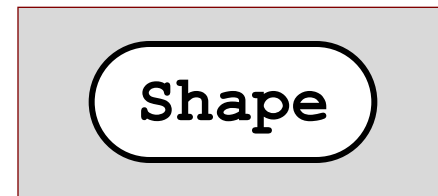
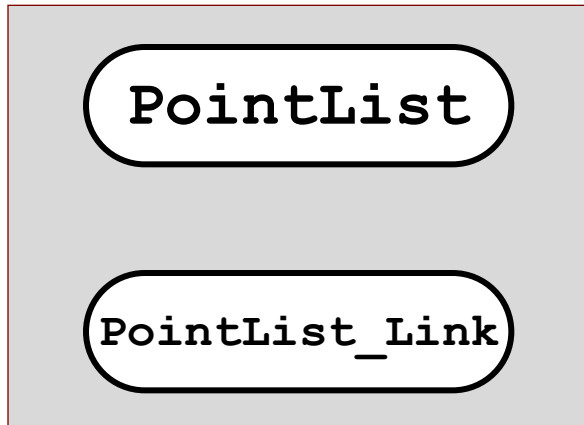
1. Components (review)

Logical Relationships



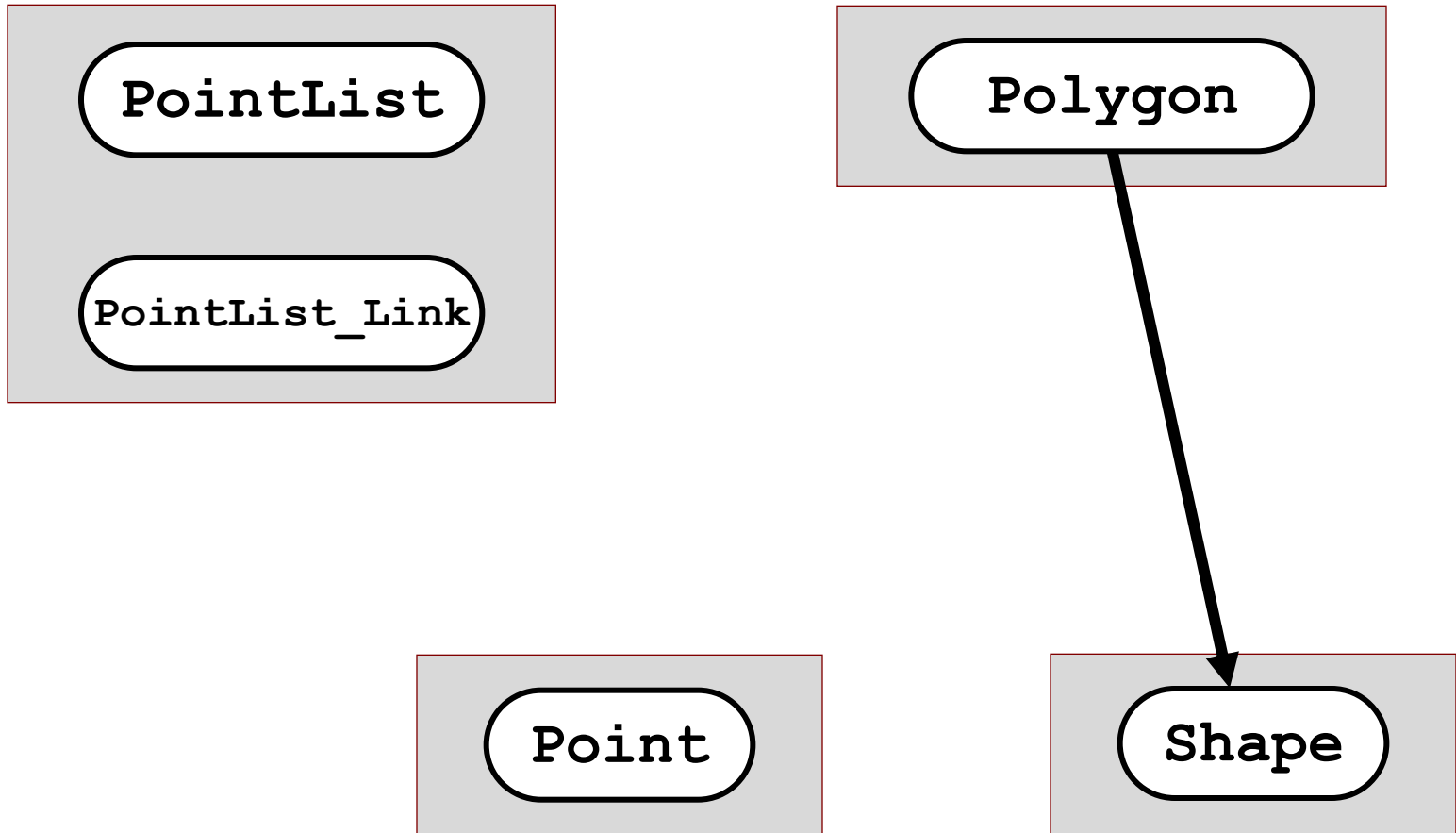
1. Components (review)

Logical Relationships



1. Components (review)

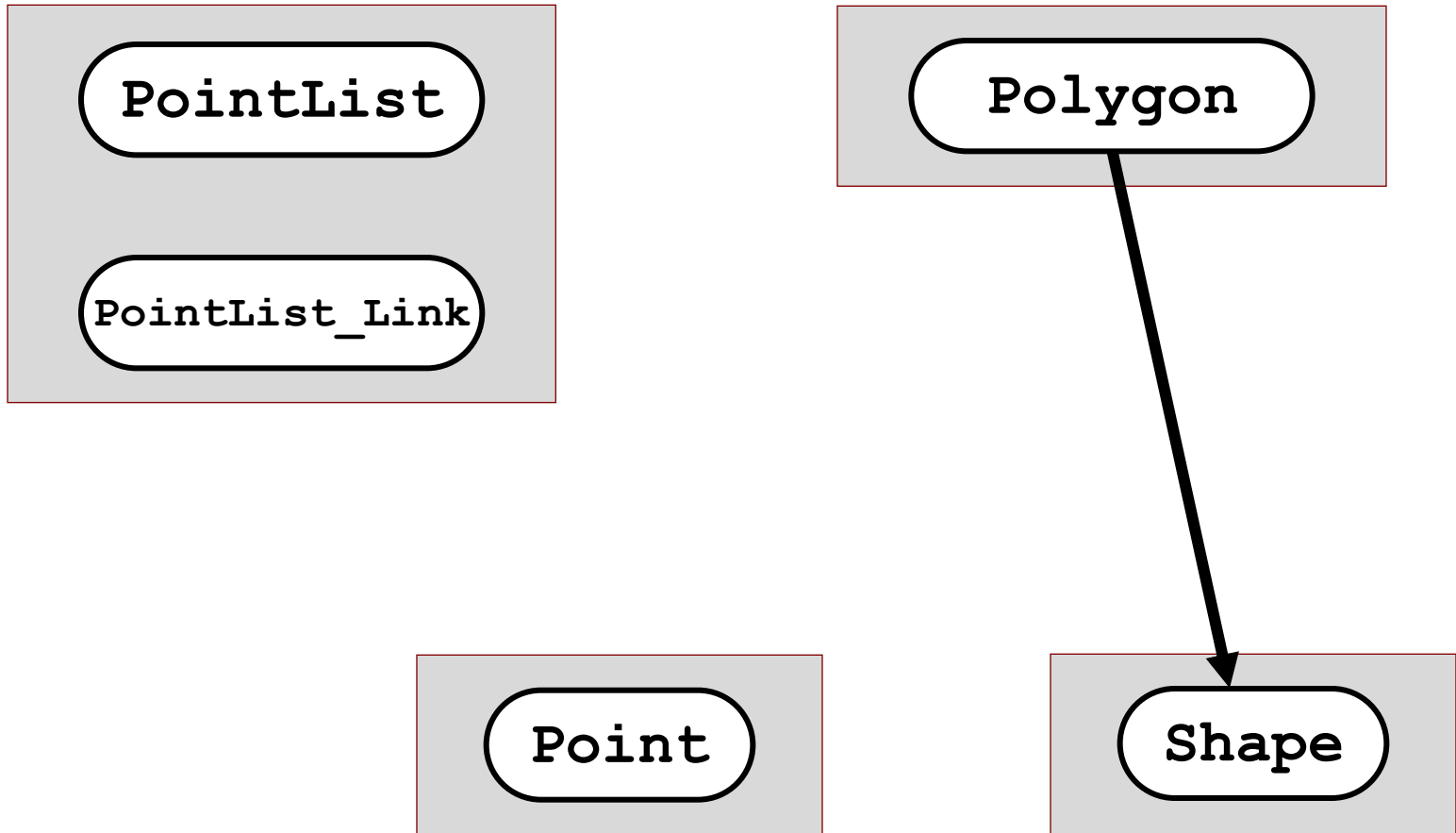
Logical Relationships



→ Is-A

1. Components (review)

Logical Relationships

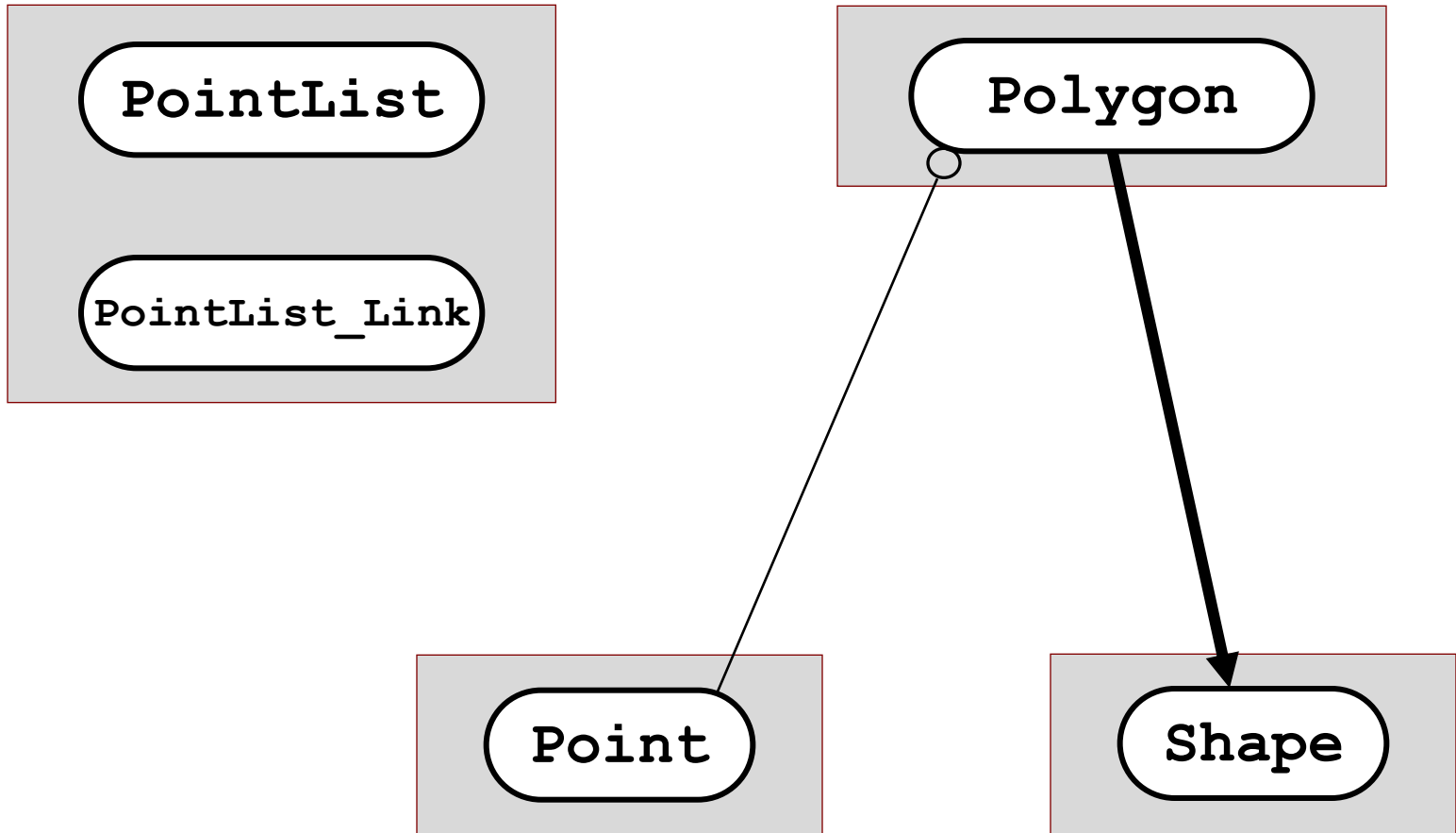


○ — Uses-in-the-Interface

→ Is-A

1. Components (review)

Logical Relationships

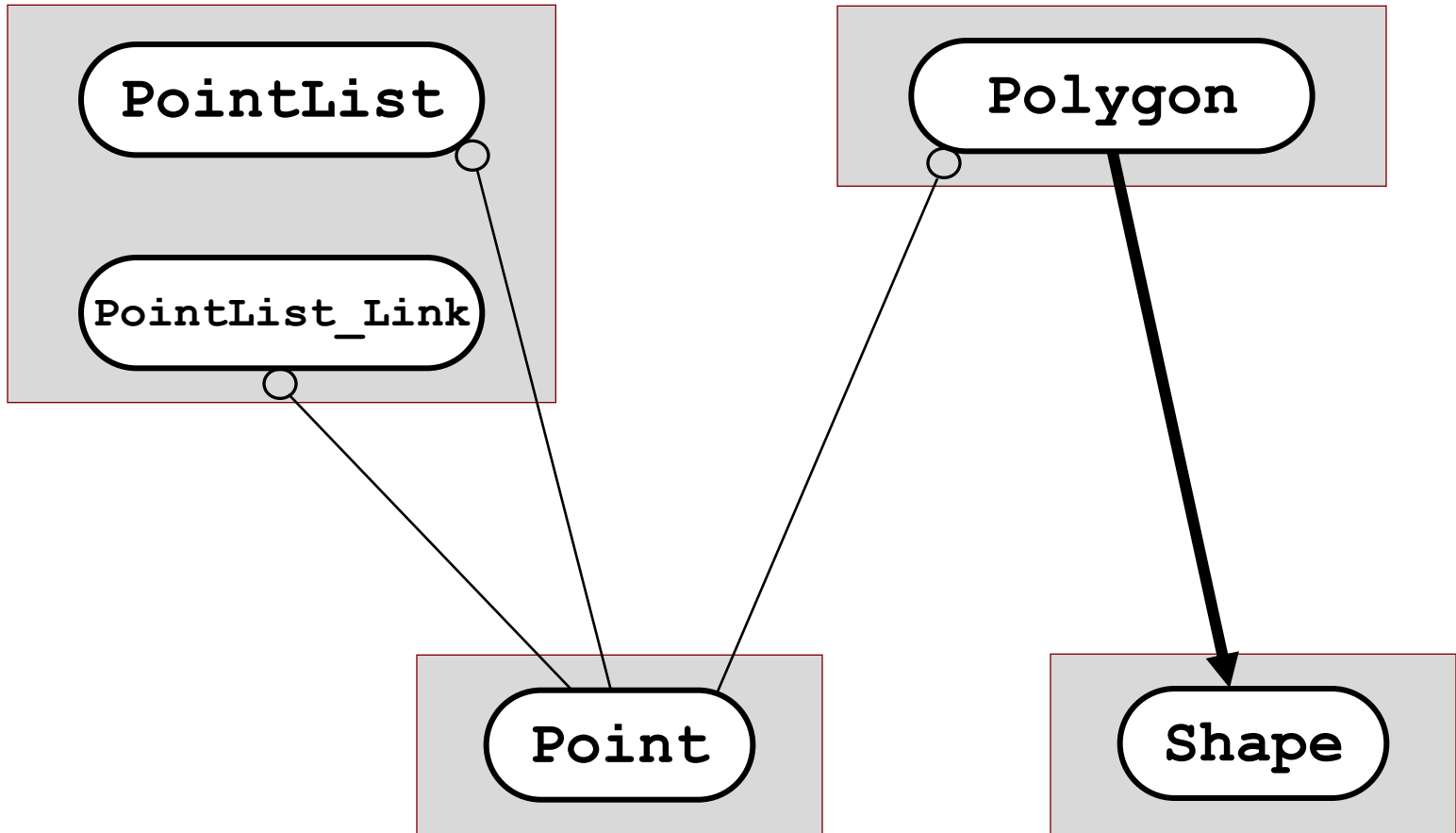


○ — Uses-in-the-Interface

➔ Is-A

1. Components (review)

Logical Relationships

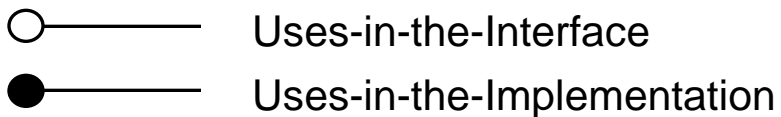
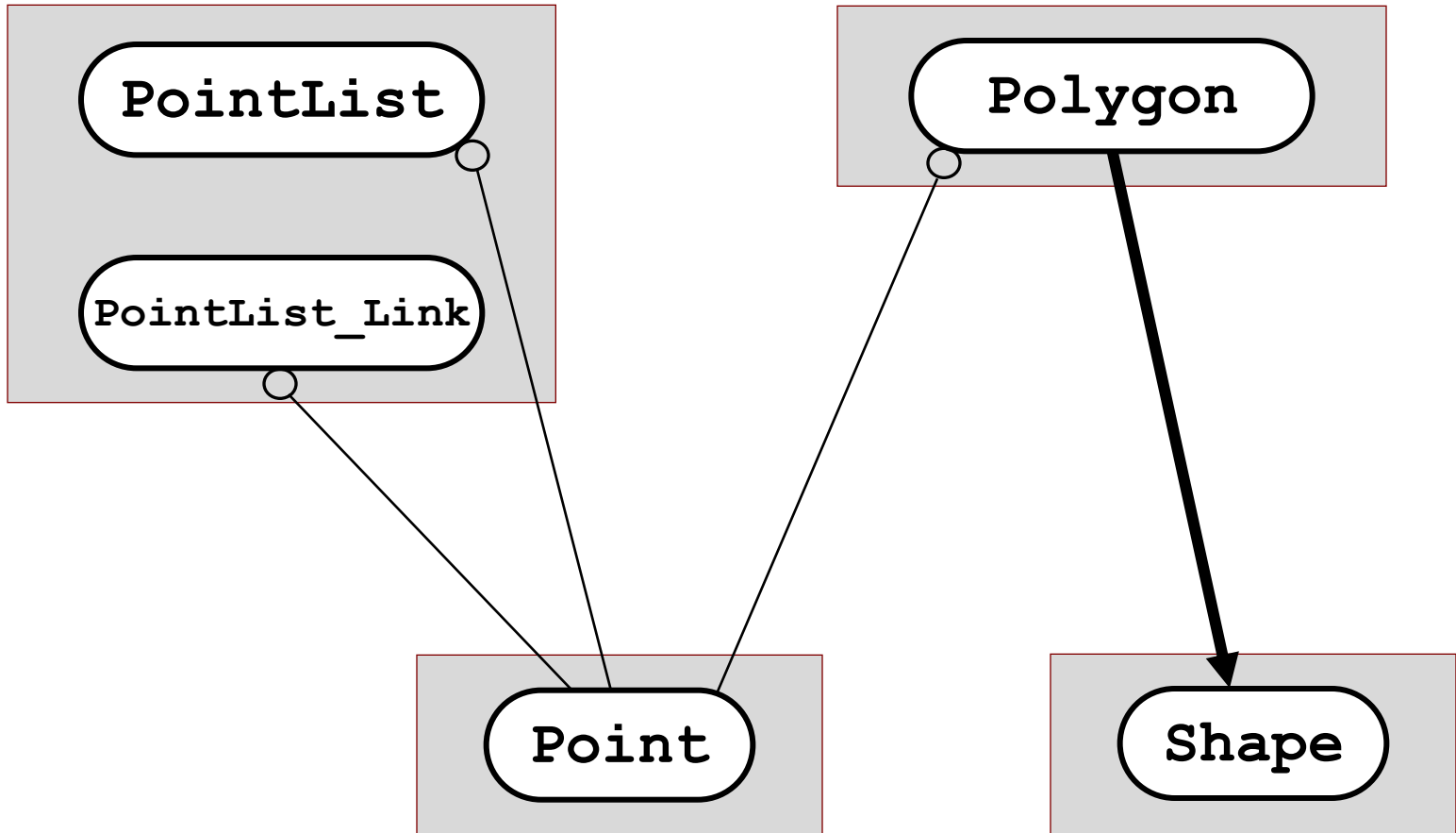


○ — Uses-in-the-Interface

➔ Is-A

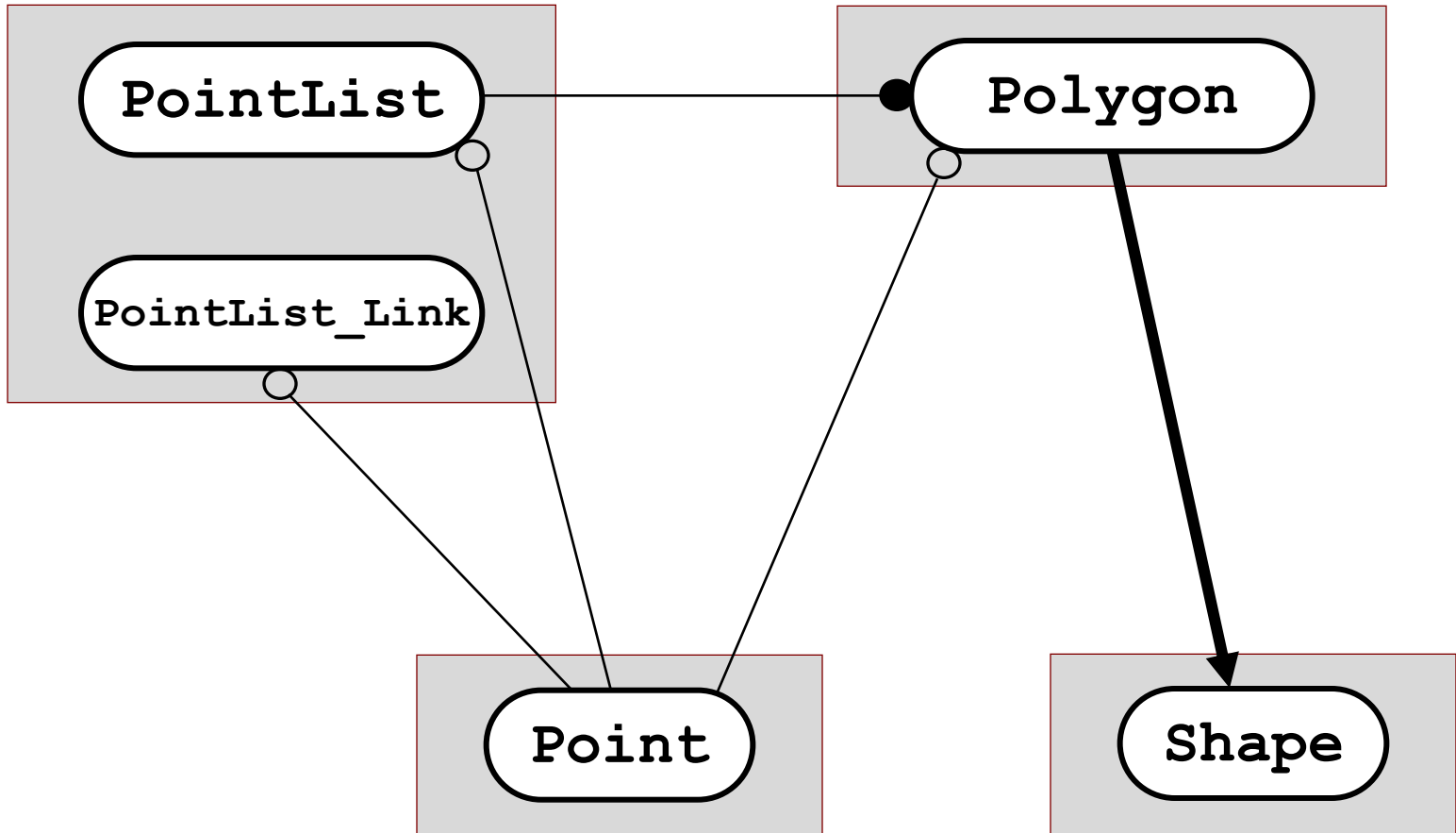
1. Components (review)

Logical Relationships



1. Components (review)

Logical Relationships



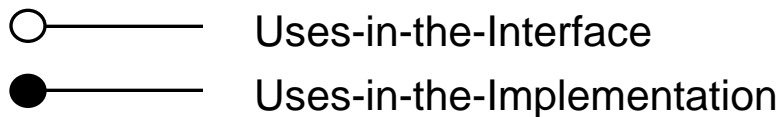
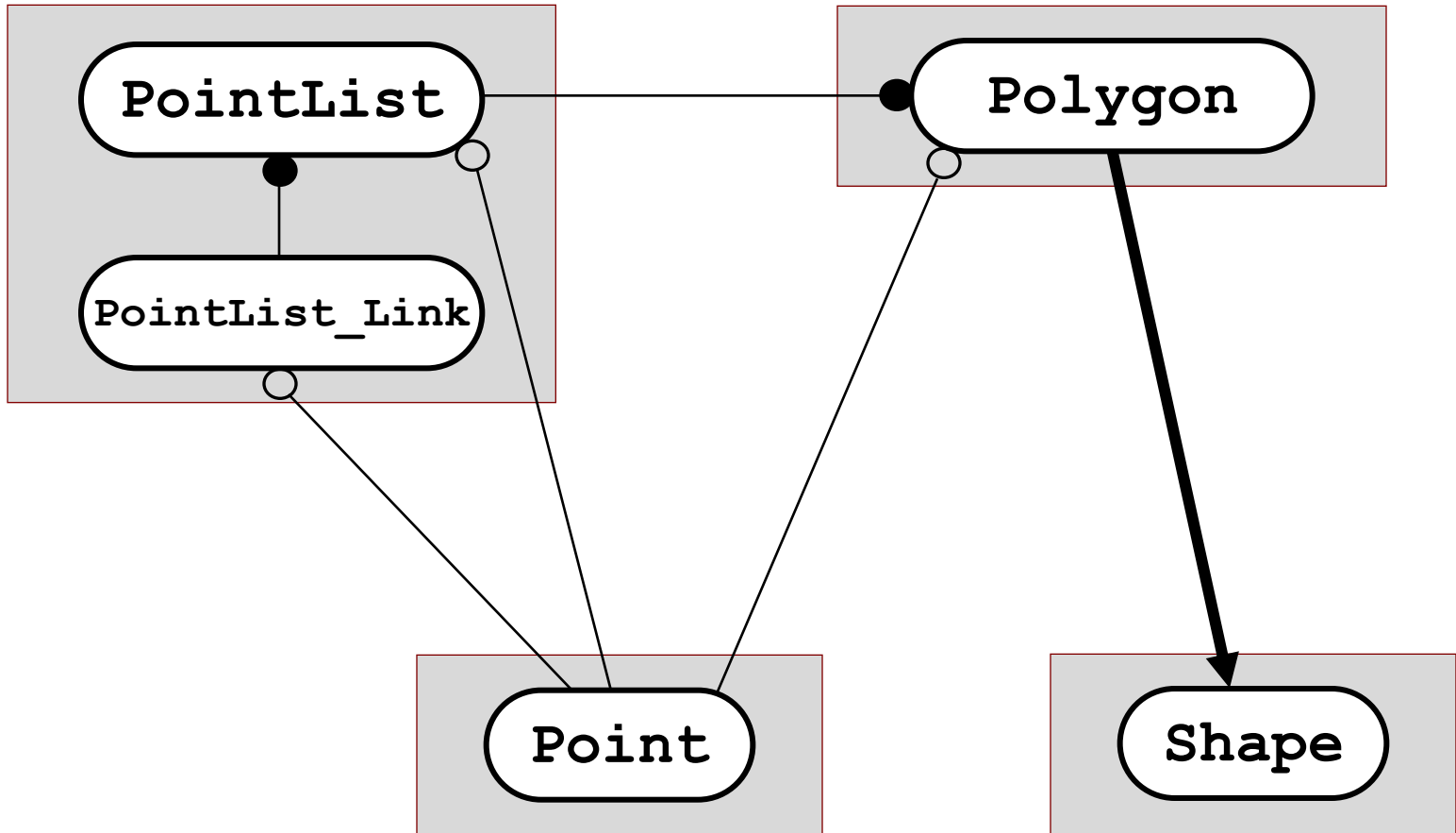
○ — Uses-in-the-Interface

● — Uses-in-the-Implementation

➔ Is-A

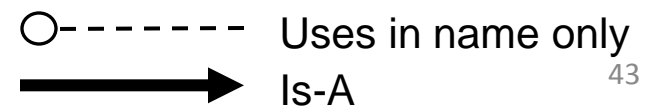
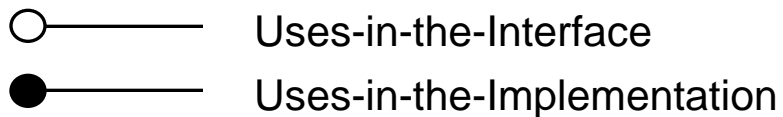
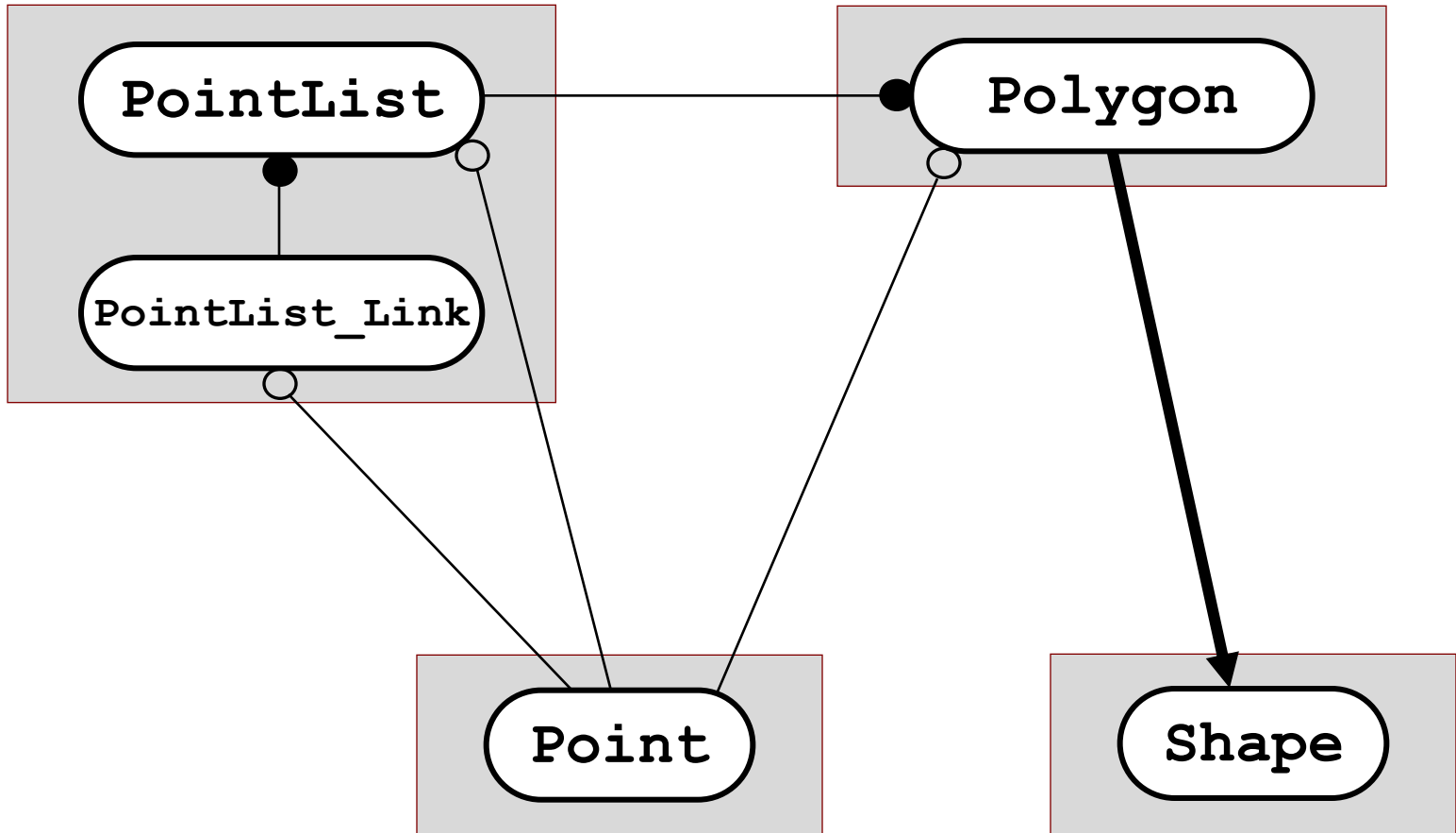
1. Components (review)

Logical Relationships



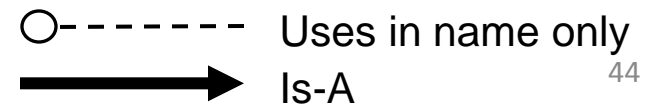
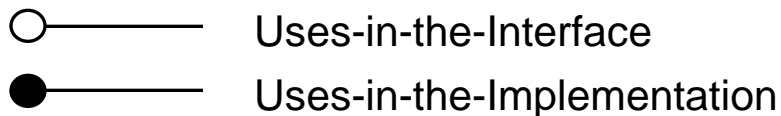
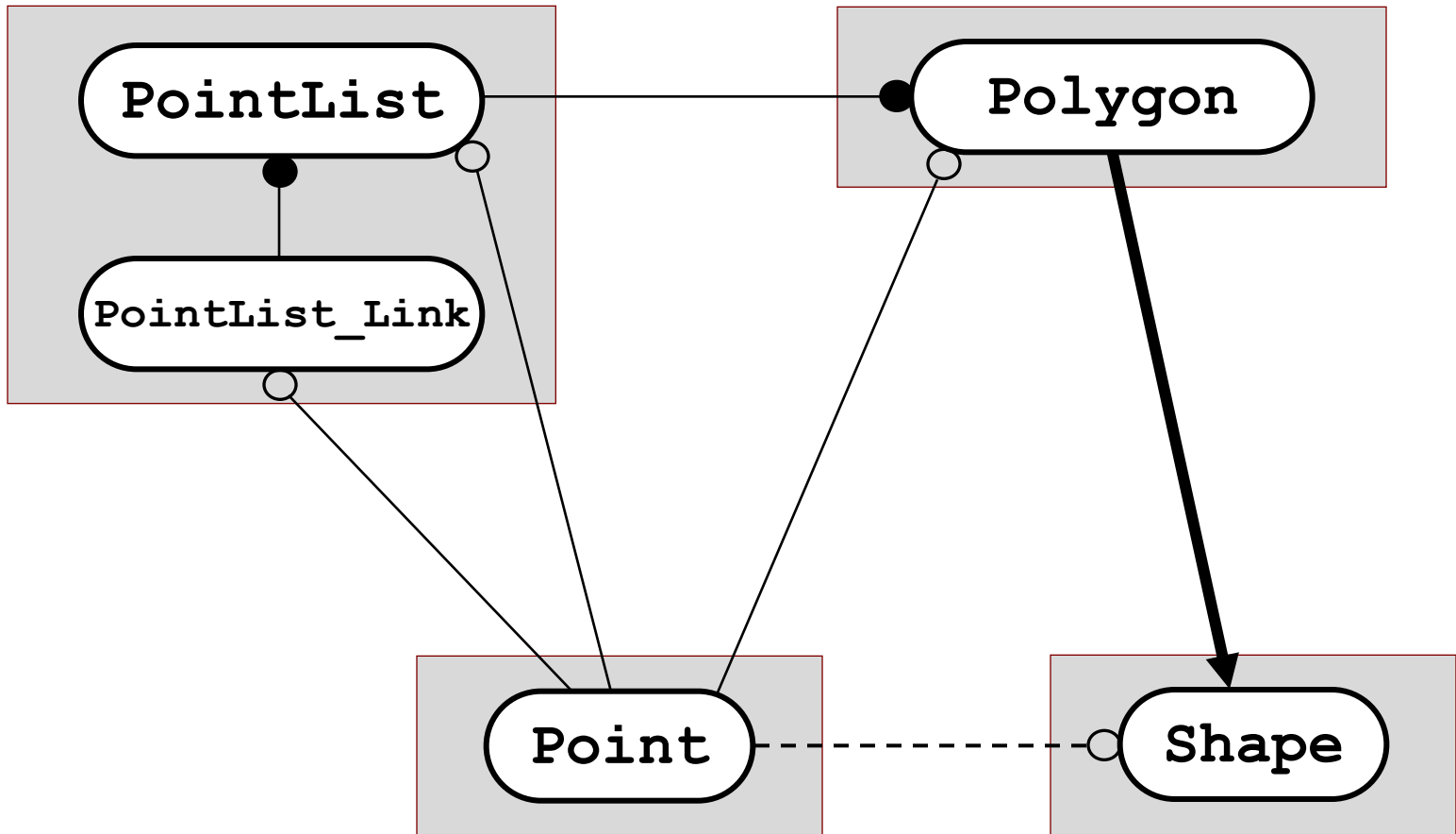
1. Components (review)

Logical Relationships



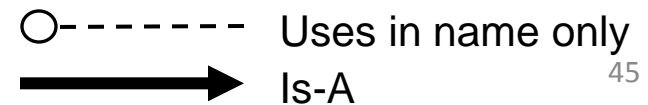
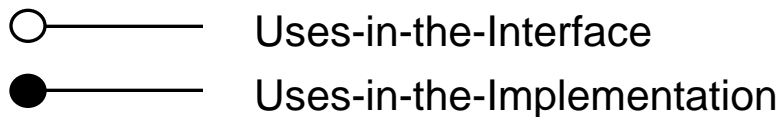
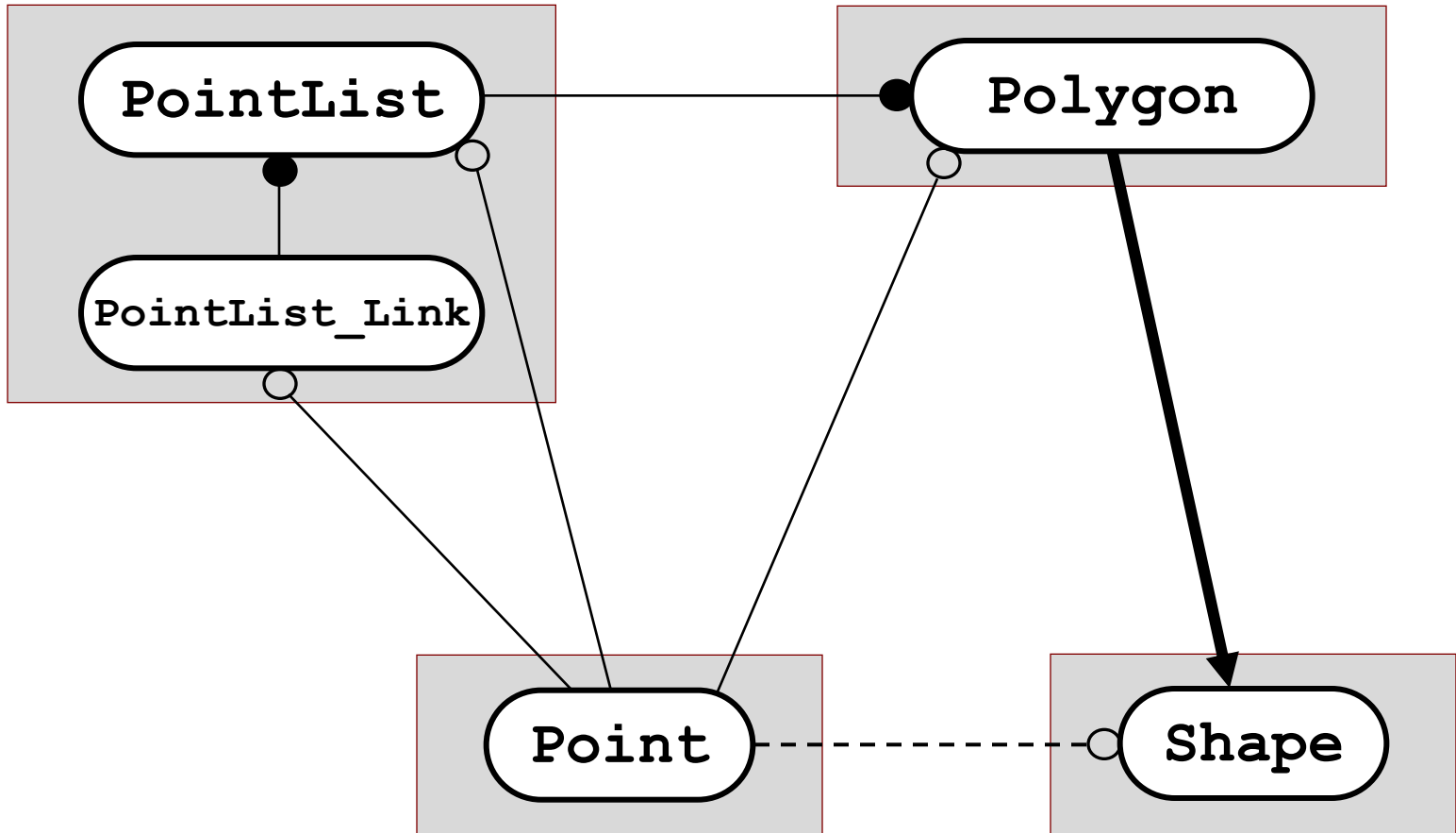
1. Components (review)

Logical Relationships



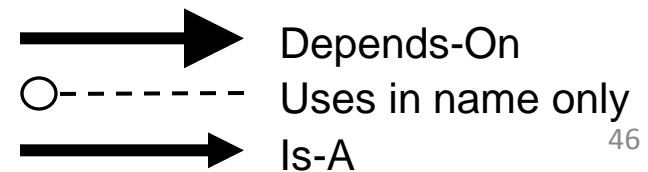
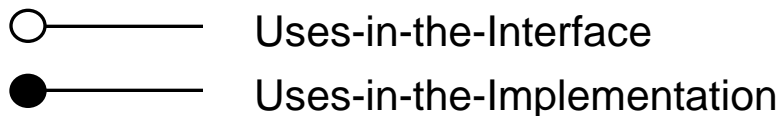
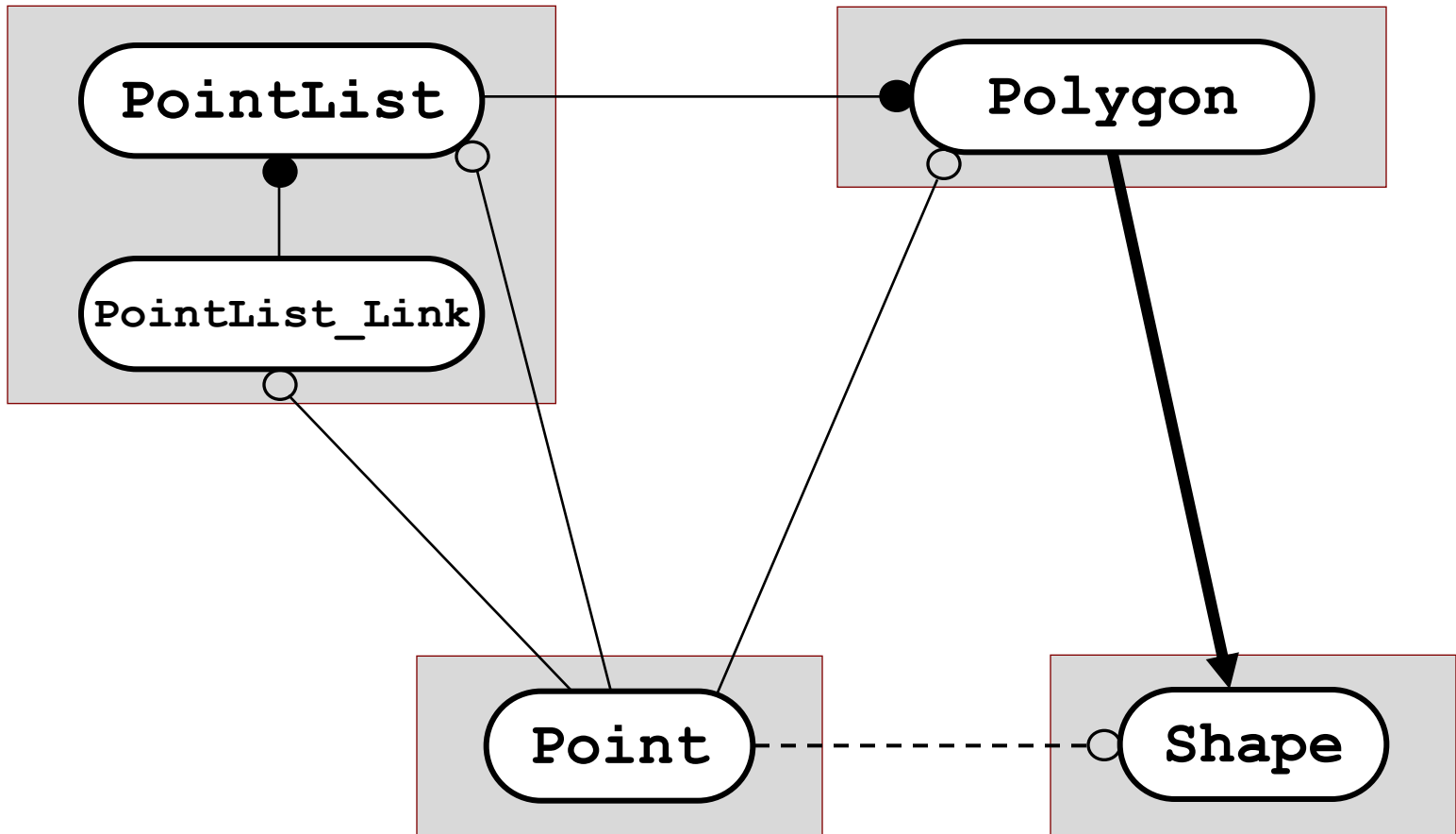
1. Components (review)

Implied Dependency



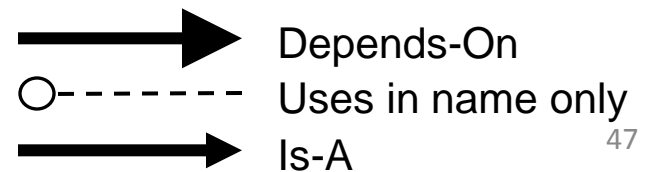
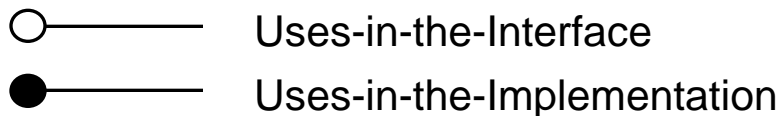
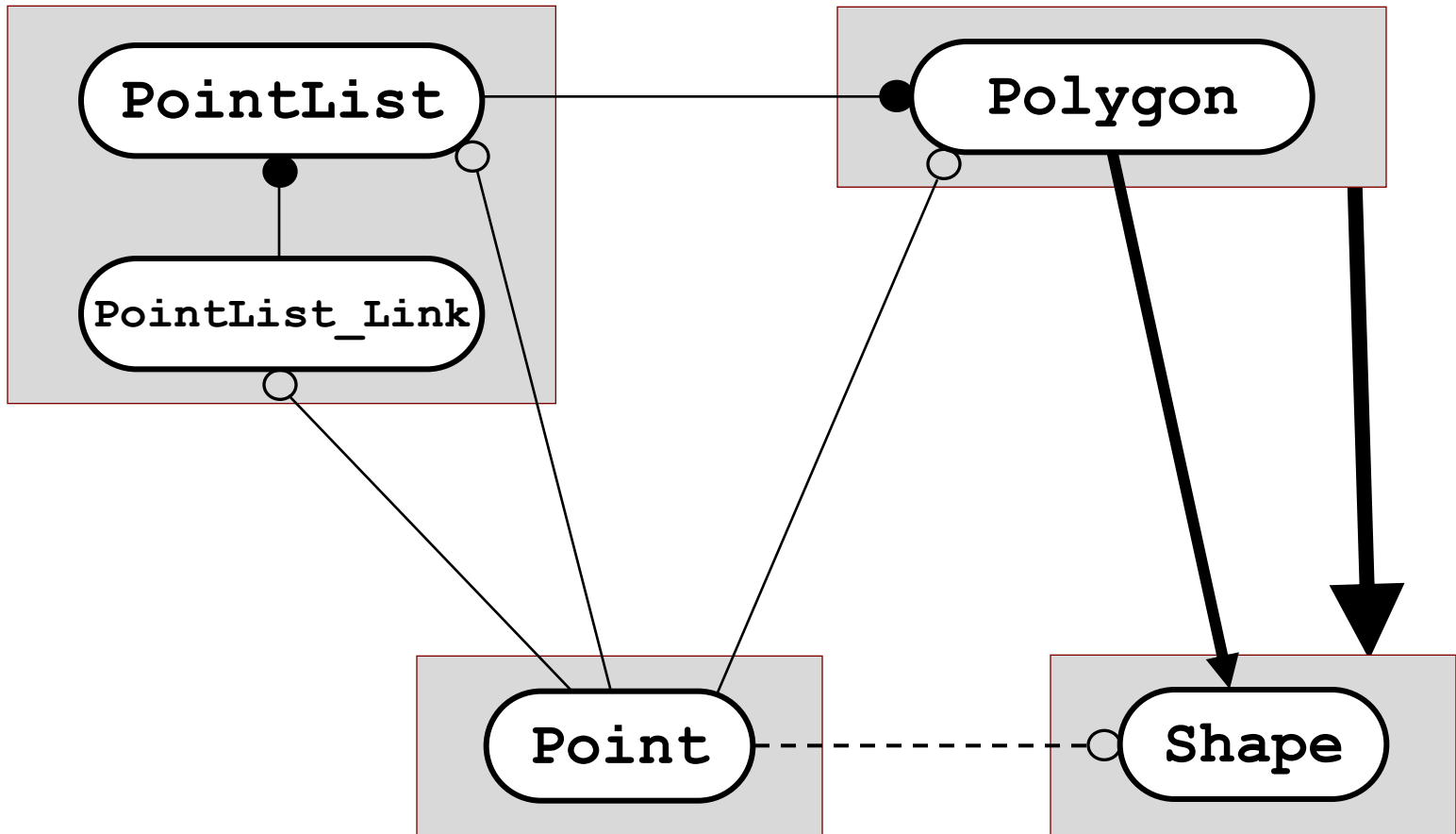
1. Components (review)

Implied Dependency



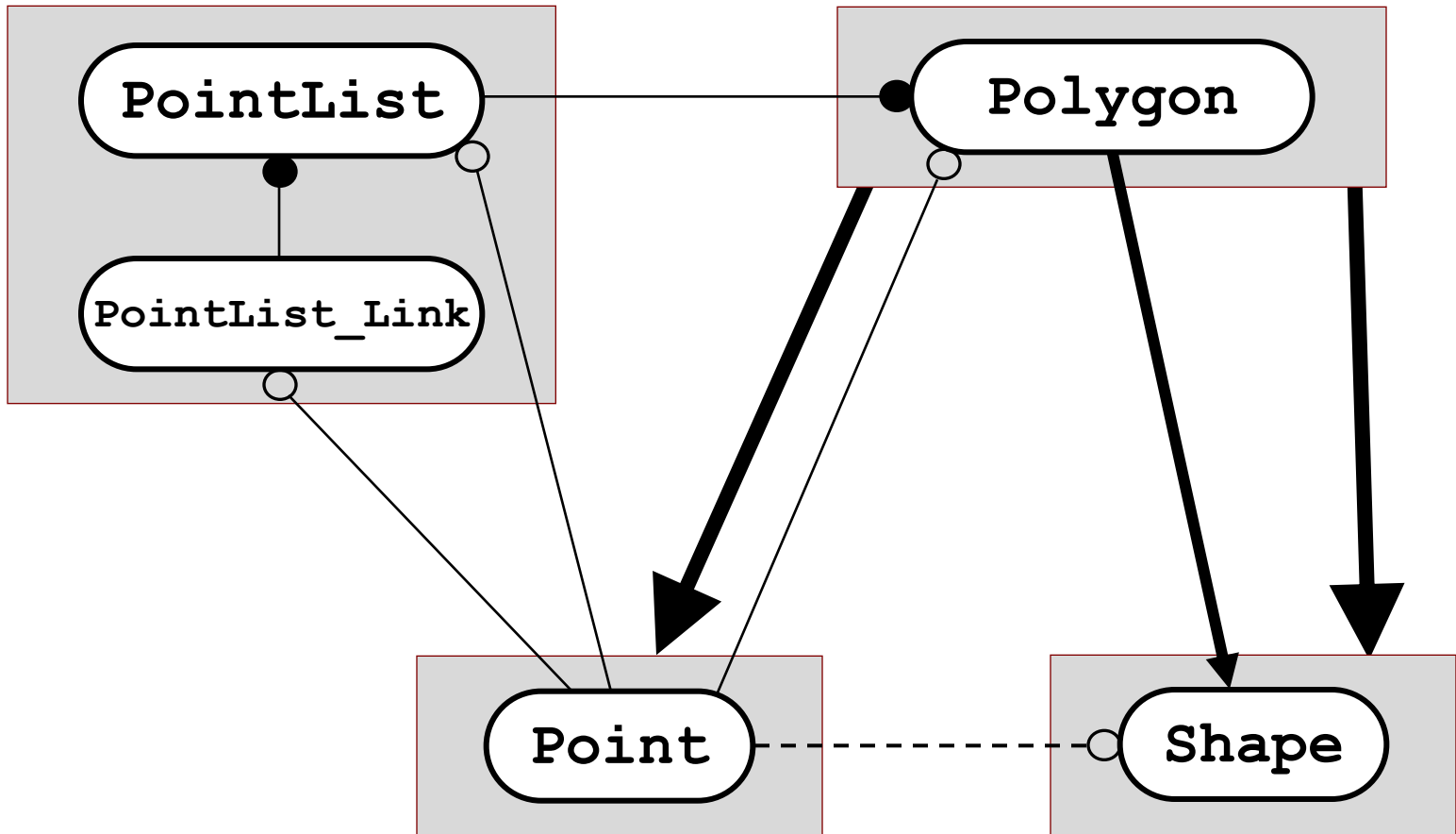
1. Components (review)

Implied Dependency



1. Components (review)

Implied Dependency

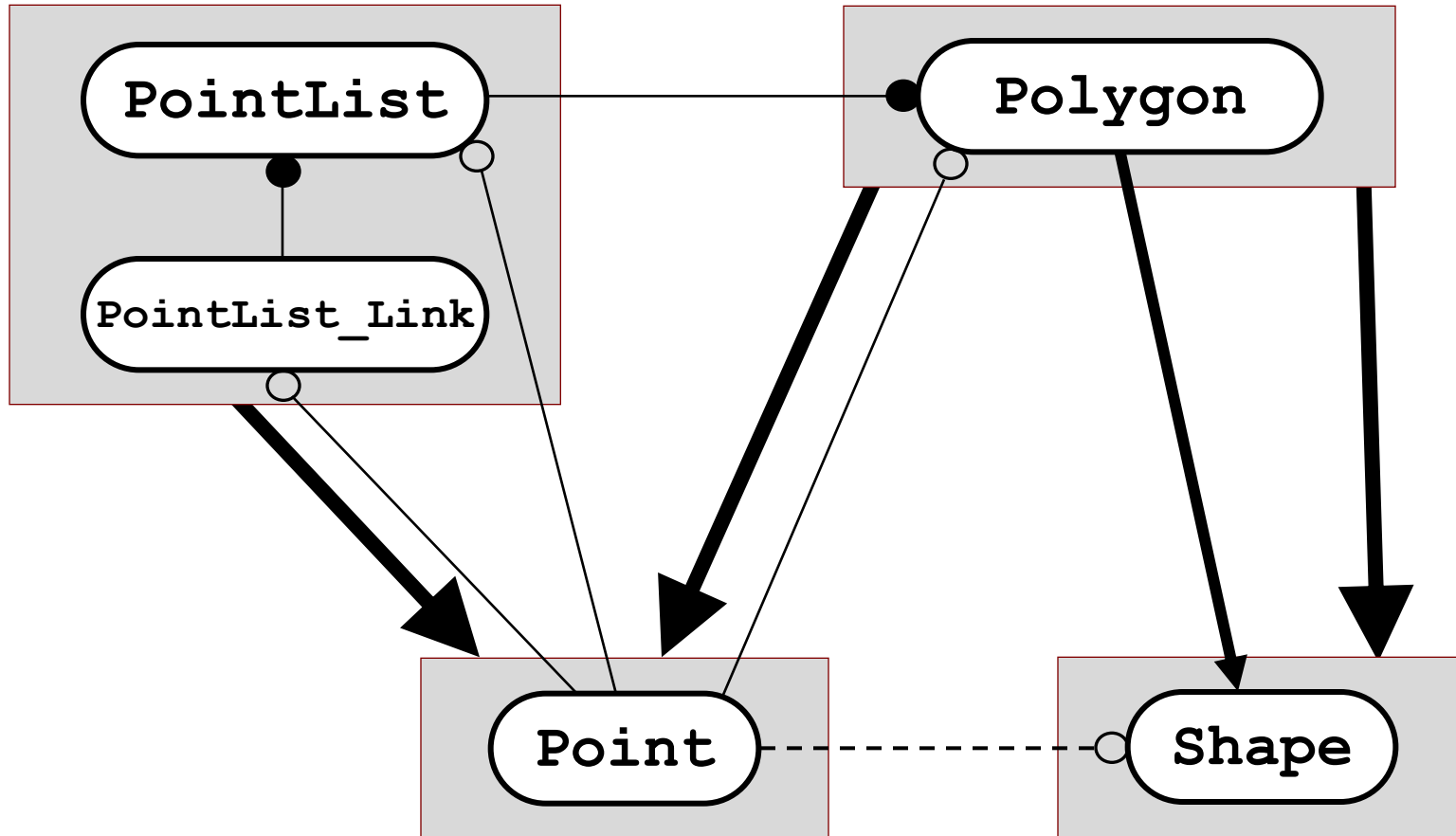


○ — Uses-in-the-Interface
● — Uses-in-the-Implementation

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

1. Components (review)

Implied Dependency

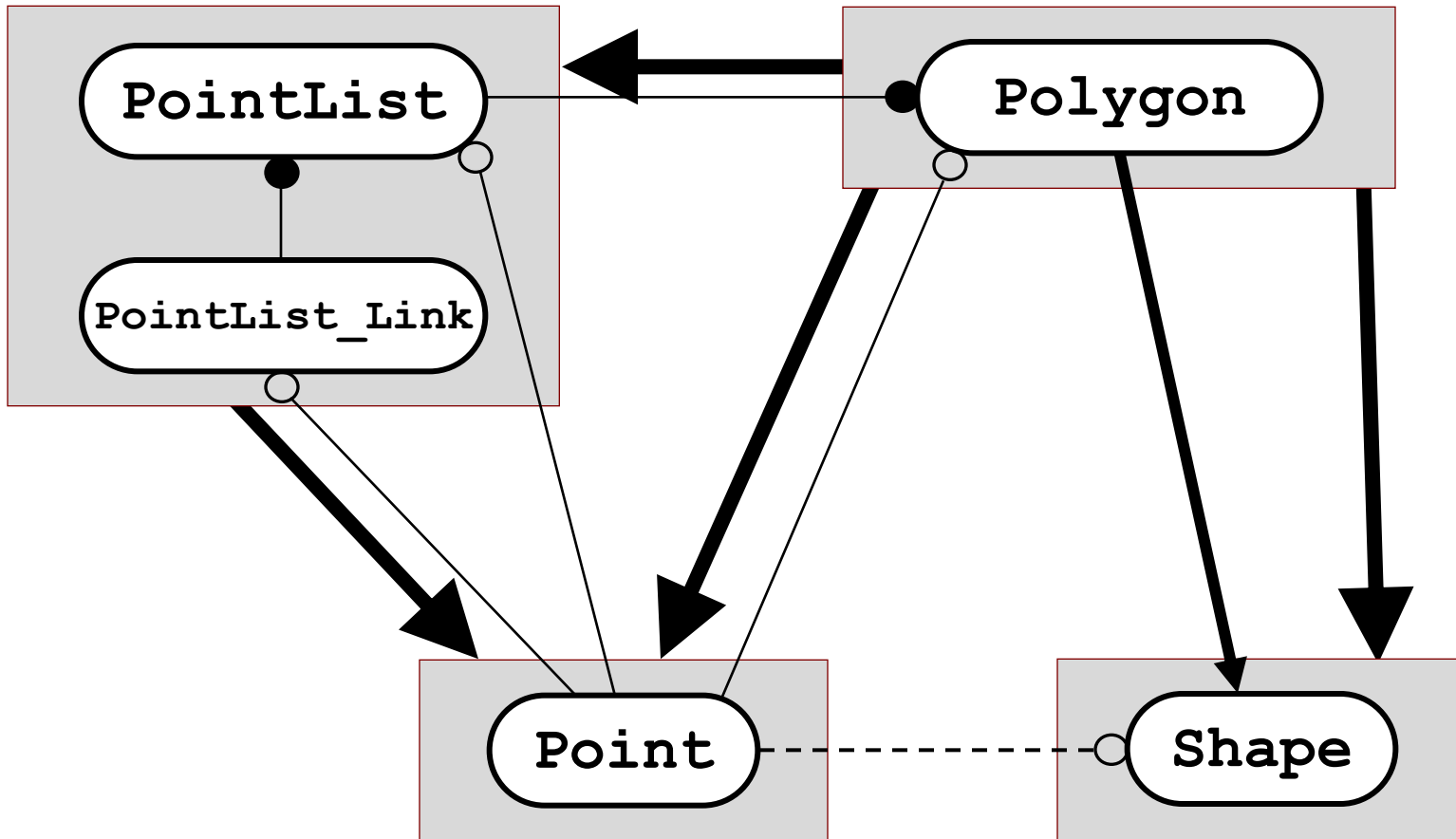


○ — Uses-in-the-Interface
● — Uses-in-the-Implementation

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

1. Components (review)

Implied Dependency

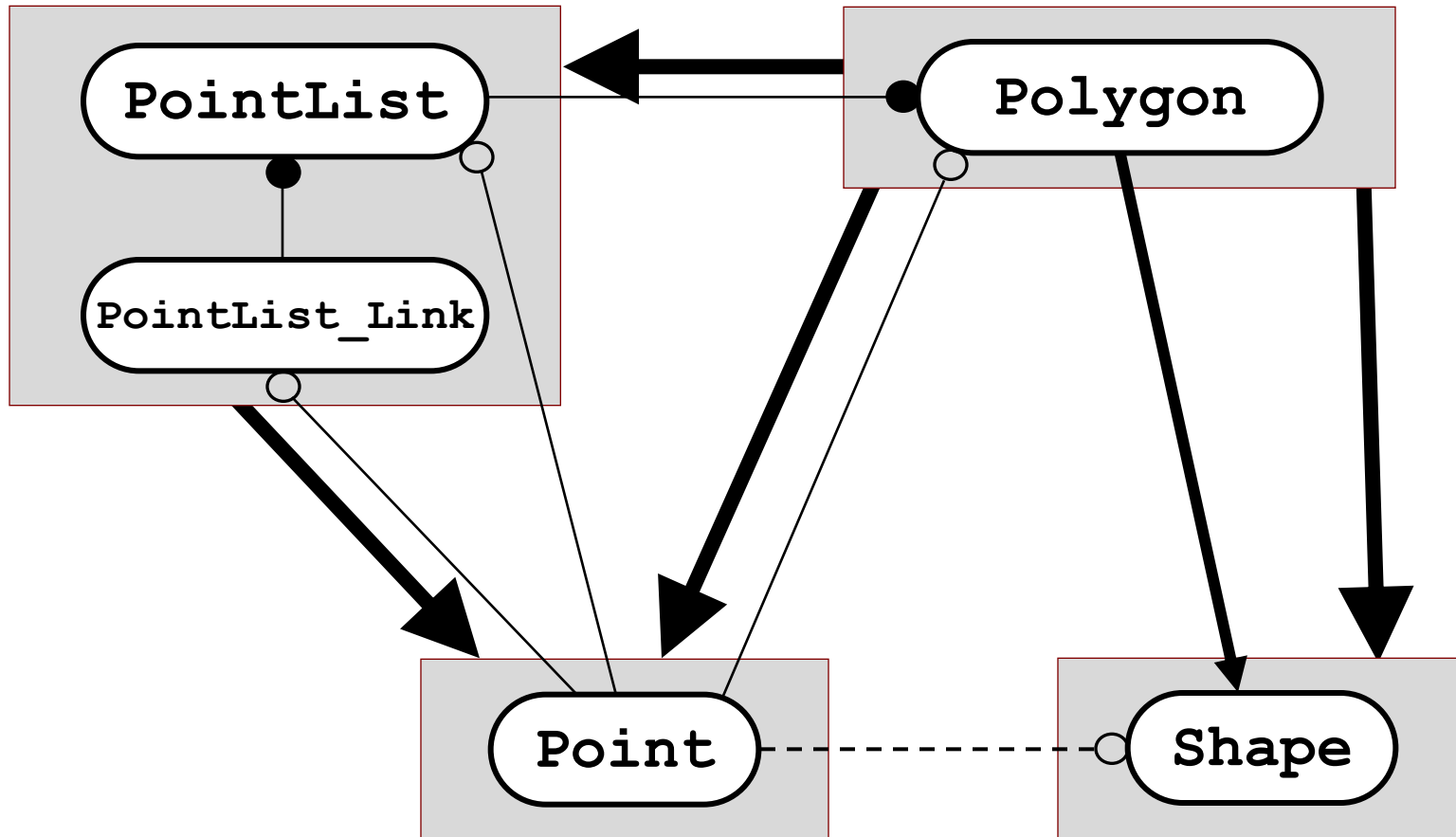


○ — Uses-in-the-Interface
● — Uses-in-the-Implementation

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

1. Components (review)

Level Numbers

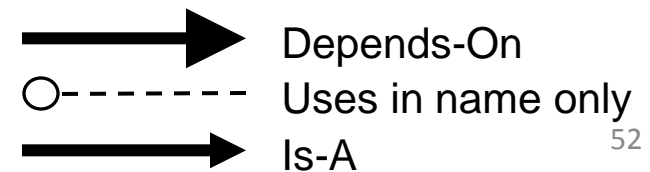
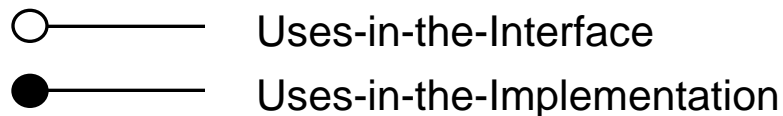
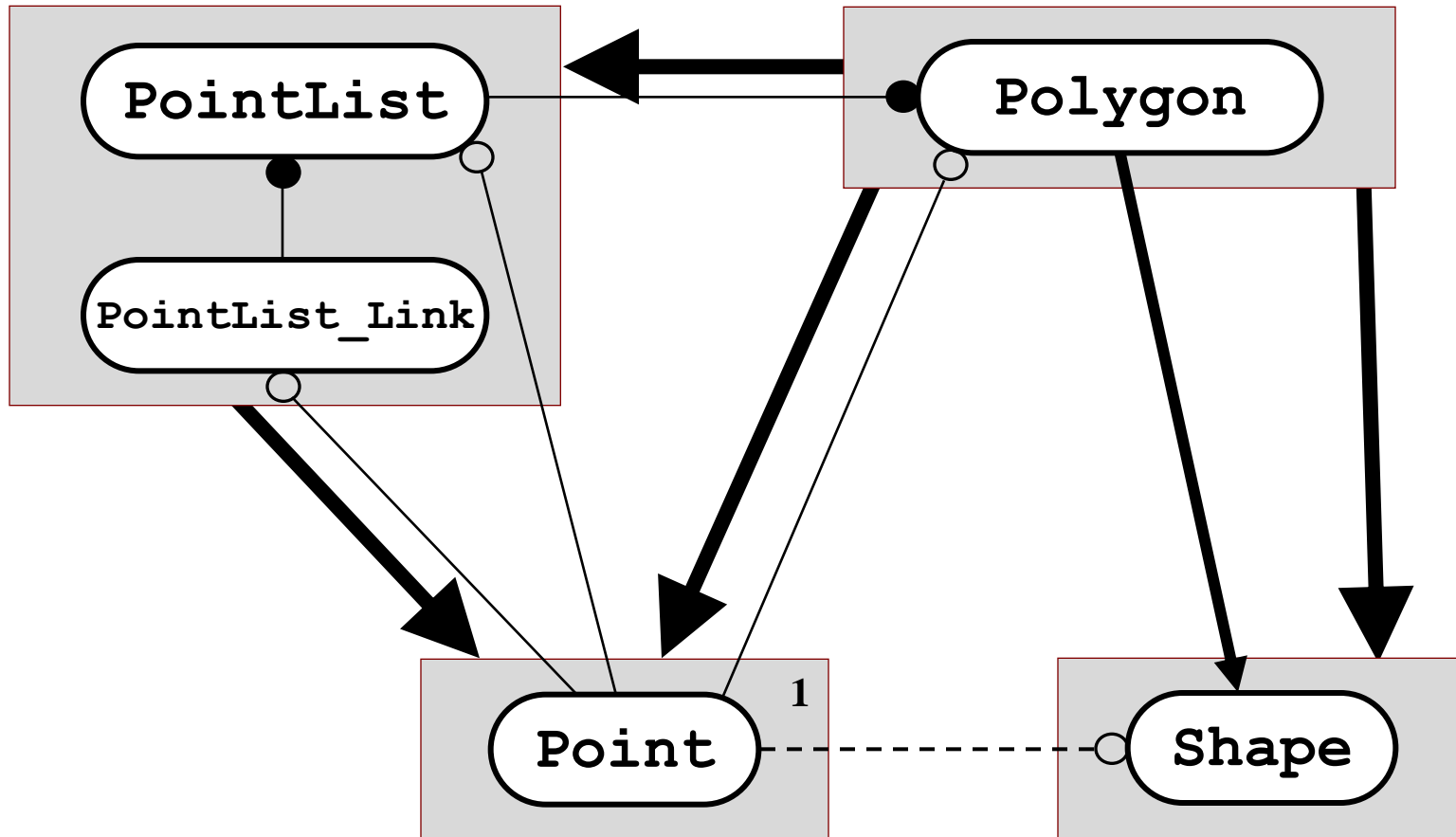


○ — Uses-in-the-Interface
● — Uses-in-the-Implementation

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

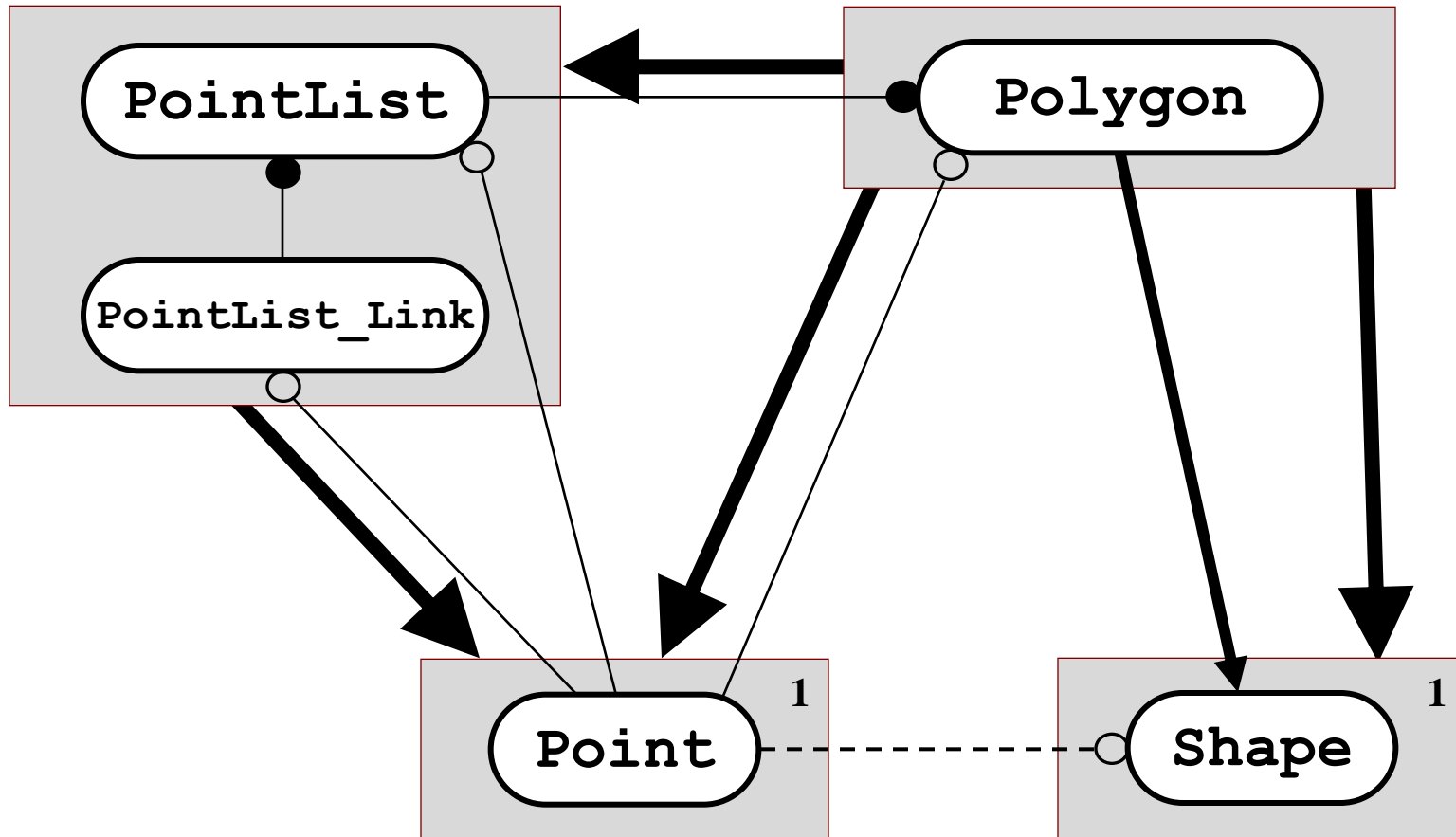
1. Components (review)

Level Numbers



1. Components (review)

Level Numbers

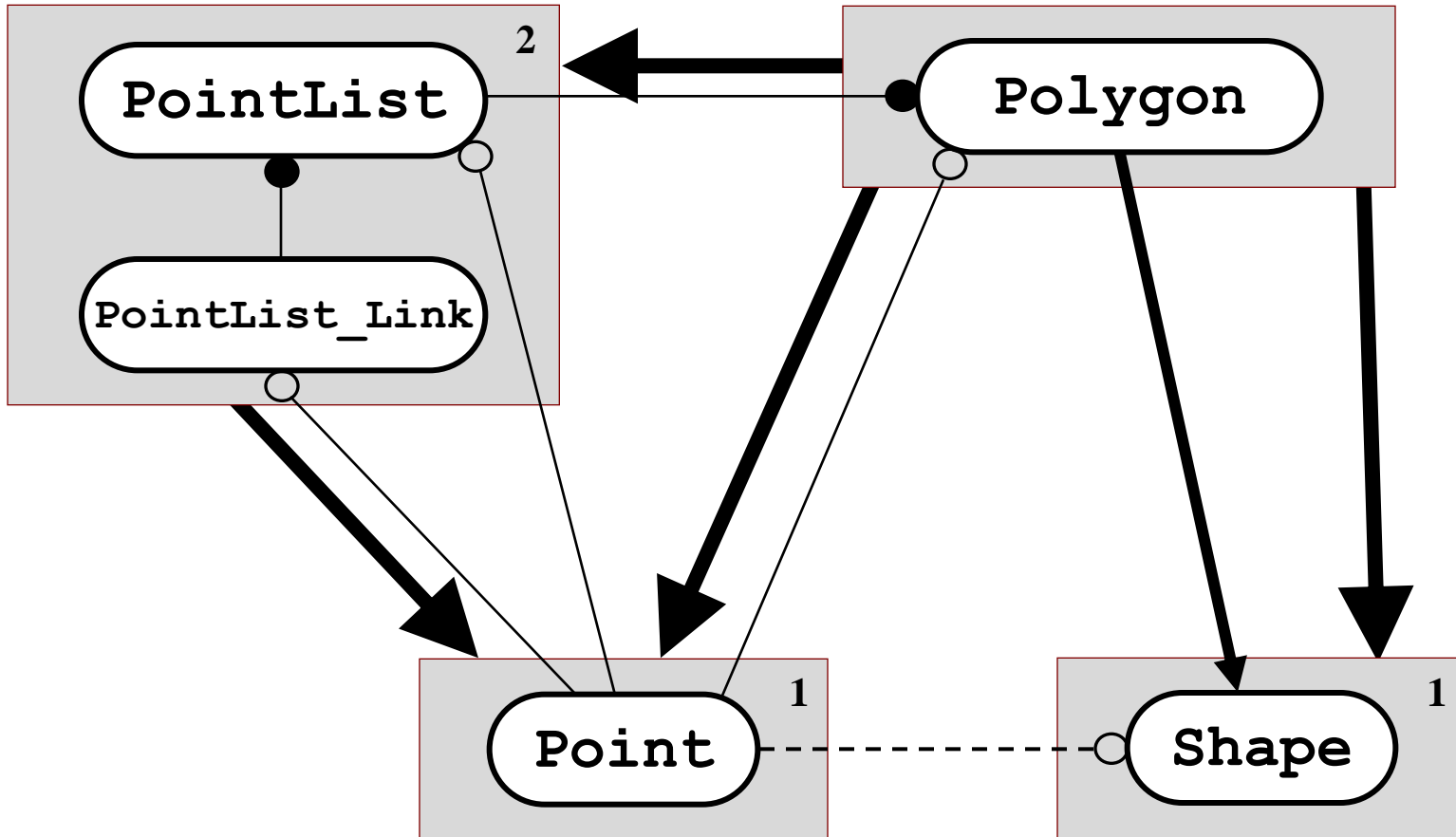


○ — Uses-in-the-Interface
● — Uses-in-the-Implementation

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

1. Components (review)

Level Numbers

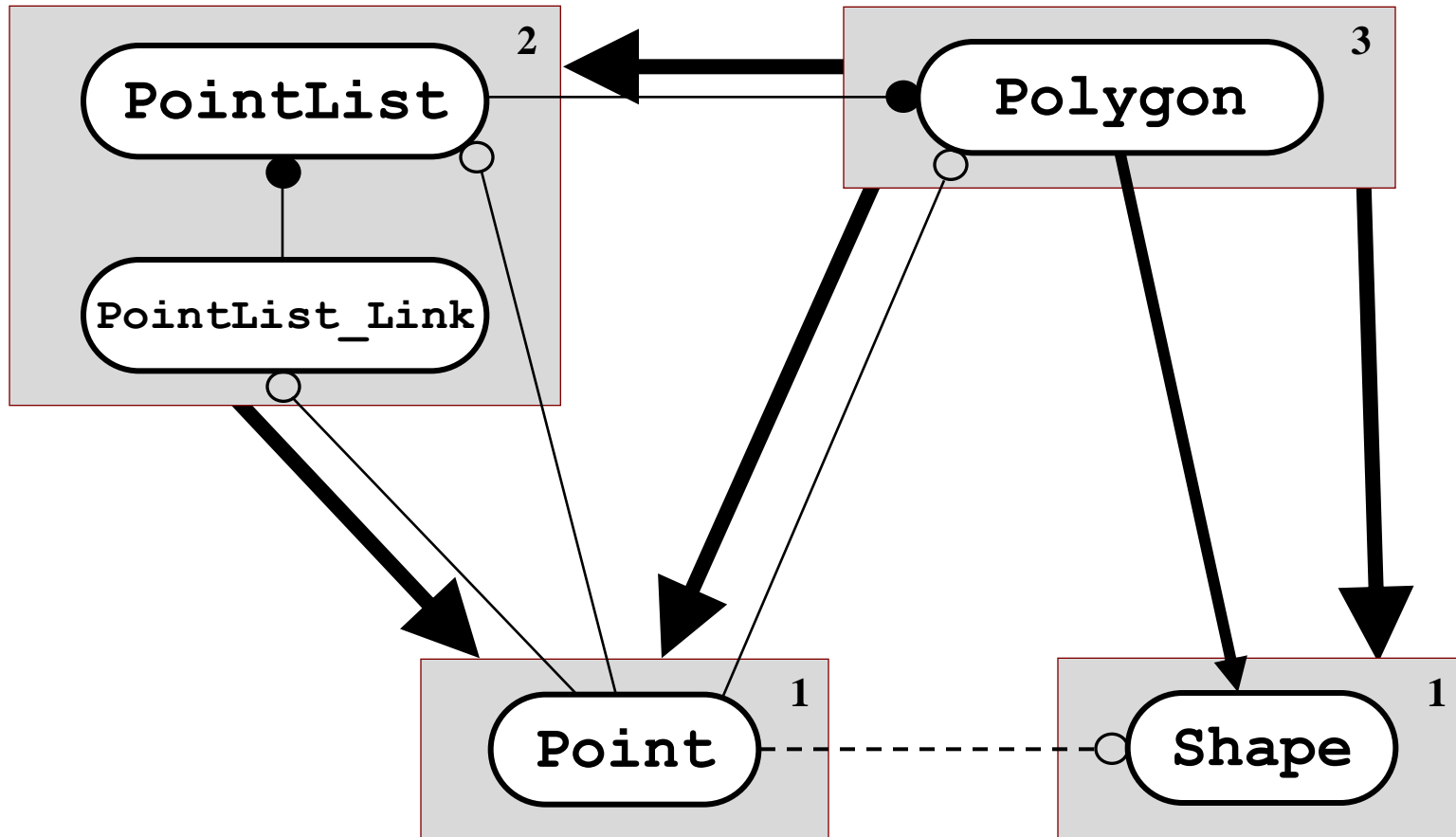


○ — Uses-in-the-Interface
● — Uses-in-the-Implementation

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

1. Components (review)

Level Numbers



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

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

1. Components (review)

Essential Physical Design Rules

1. Components (review)

Essential Physical Design Rules

There are two:

1. Components (review)

Essential Physical Design Rules

There are two:

1. No *Cyclic* Physical Dependencies!

1. Components (review)

Essential Physical Design Rules

There are two:

1. No *Cyclic* Physical Dependencies!

2. No *Long-Distance* Friendships!

1. Components (review)

End of Section

Questions?

1. Components (review)

What Questions are we Answering?

- What distinguishes *Logical* and *Physical* Design?
- What is the first of the (four) fundamental properties of a `.h/.cpp` pair that make it a component?
- Which of these fundamental properties helps us extract physical dependencies efficiently? Extra credit: Why? How?
- What are the (four) logical-relationship annotations?
- Which logical relationship does not imply a physical one?
- How do we infer physical relationships (*Depends-On*) from logical ones?
- What do we mean by the term *level number*?
- What are the (two) *quintessential* physical design rules?

Outline

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

4. Proper Inheritance

Is-A for *Interface*, *Structural*, & *Implementation* Inheritance

Outline

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

4. Proper Inheritance

Is-A for *Interface*, *Structural*, & *Implementation* Inheritance

2. Interfaces and Contracts (review)

Interfaces and Contracts

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

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

2. Interfaces and Contracts (review)

Interfaces and Contracts

Function

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

2. Interfaces and Contracts (review)

Interfaces and Contracts

Function



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

Types Used
In the Interface

2. Interfaces and Contracts (review)

Interfaces and Contracts

Function

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

2. Interfaces and Contracts (review)

Interfaces and Contracts

Class

```
class Date {
```

```
    //...
```

```
    public:
```

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

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

```
        // ...
```

```
};
```

2. Interfaces and Contracts (review)

Interfaces and Contracts

Class

```
class Date {
```

```
    //...
```

```
    public:
```

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

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

```
    // ...
```

```
};
```



Public
Interface

2. Interfaces and Contracts (review)

Interfaces and Contracts

Class

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

2. Interfaces and Contracts (review)

Interfaces and Contracts

Class

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date in history between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
  
    Date(const Date& original);  
  
    // ...  
};
```

2. Interfaces and Contracts (review)

Interfaces and Contracts

Class

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


2. Interfaces and Contracts (review)

Interfaces and Contracts

Class

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

2. Interfaces and Contracts (review)

Interfaces and Contracts

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

Component

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

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

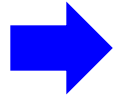
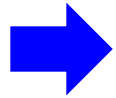
```
std::ostream& operator<<(std::ostream& stream, const Date& date);
```

2. Interfaces and Contracts (review)

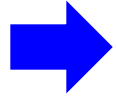
Interfaces and Contracts

Component

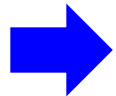
```
class Date {  
    // ...  
    public:  
    // ...  
};
```



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



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



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

“Public”
Interface

2. Interfaces and Contracts (review)

Interfaces and Contracts

Component

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

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

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

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

2. Interfaces and Contracts (review)

Interfaces and Contracts

Component

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

```
bool operator==(const Date& lhs, const Date& rhs);  
    // Return 'true' if the specified 'lhs' and 'rhs' dates have the same  
    // value, and 'false' otherwise. Two 'Date' objects have the same  
    // value if their respective 'year', 'month', and 'day' attributes  
    // have the same value.
```

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

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

2. Interfaces and Contracts (review)

Interfaces and Contracts

Component

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

```
bool operator==(const Date& lhs, const Date& rhs);  
    // Return 'true' if the specified 'lhs' and 'rhs' dates have the same  
    // value, and 'false' otherwise. Two 'Date' objects have the same  
    // value if their respective 'year', 'month', and 'day' attributes  
    // have the same value.
```

```
bool operator!=(const Date& lhs, const Date& rhs);  
    // Return 'true' if the specified 'lhs' and 'rhs' dates do not have the  
    // same value, and 'false' otherwise. Two 'Date' objects do not have  
    // the same value if any of their respective 'year', 'month', and 'day'  
    // attributes do not have the same value.
```

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

2. Interfaces and Contracts (review)

Interfaces and Contracts

Component

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

```
bool operator==(const Date& lhs, const Date& rhs);  
    // Return 'true' if the specified 'lhs' and 'rhs' dates have the same  
    // value, and 'false' otherwise. Two 'Date' objects have the same  
    // value if their respective 'year', 'month', and 'day' attributes  
    // have the same value.
```

```
bool operator!=(const Date& lhs, const Date& rhs);  
    // Return 'true' if the specified 'lhs' and 'rhs' dates do not have the  
    // same value, and 'false' otherwise. Two 'Date' objects do not have  
    // the same value if any of their respective 'year', 'month', and 'day'  
    // attributes do not have the same value.
```

```
std::ostream& operator<<(std::ostream& stream, const Date& date);  
    // Format the value of the specified 'date' object to the specified  
    // output 'stream' as 'yyyy/mm/dd', and return a reference to 'stream'.
```

2. Interfaces and Contracts (review)

Preconditions and Postconditions

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Function

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Function

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

```
// Return the square root of the specified 'value'.
```

```
// The behavior is undefined unless '0 <= value'.
```

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Function

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

```
// Return the square root of the specified 'value'.
```

```
// The behavior is undefined unless '0 <= value'.
```

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Function

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

```
// Return the square root of the specified 'value'.
```

```
// The behavior is undefined unless '0 <= value'.
```

Precondition

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Function

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

```
// Return the square root of the specified 'value'.
```

```
// The behavior is undefined unless '0 <= value'.
```

Precondition

For a Stateless Function:

Restriction on *syntactically legal* inputs.

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Function

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

```
// Return the square root of the specified 'value'.
```

```
// The behavior is undefined unless '0 <= value'.
```

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Function

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

```
// Return the square root of the specified 'value'.
```

```
// The behavior is undefined unless '0 <= value'.
```

Postcondition

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Function

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

```
// Return the square root of the specified 'value'.
```

```
// The behavior is undefined unless '0 <= value'.
```

Postcondition

For a Stateless Function:
What it “returns.”

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Object Method

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Object Method

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

Preconditions and Postconditions

Object Method

- Preconditions: What must be true of both (object) state and method inputs; otherwise the behavior is undefined.
- Postconditions: What must happen as a function of (object) state and input if all Preconditions are satisfied.

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Object Method

*a.k.a.
Essential
Behavior*

➤ Preconditions: What must be true of both (object) and input if the method is to be executed otherwise.

➤ Postconditions: What must happen as a function of (object) state and input if all Preconditions are satisfied.

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

a.k.a.
**Essential
Behavior**

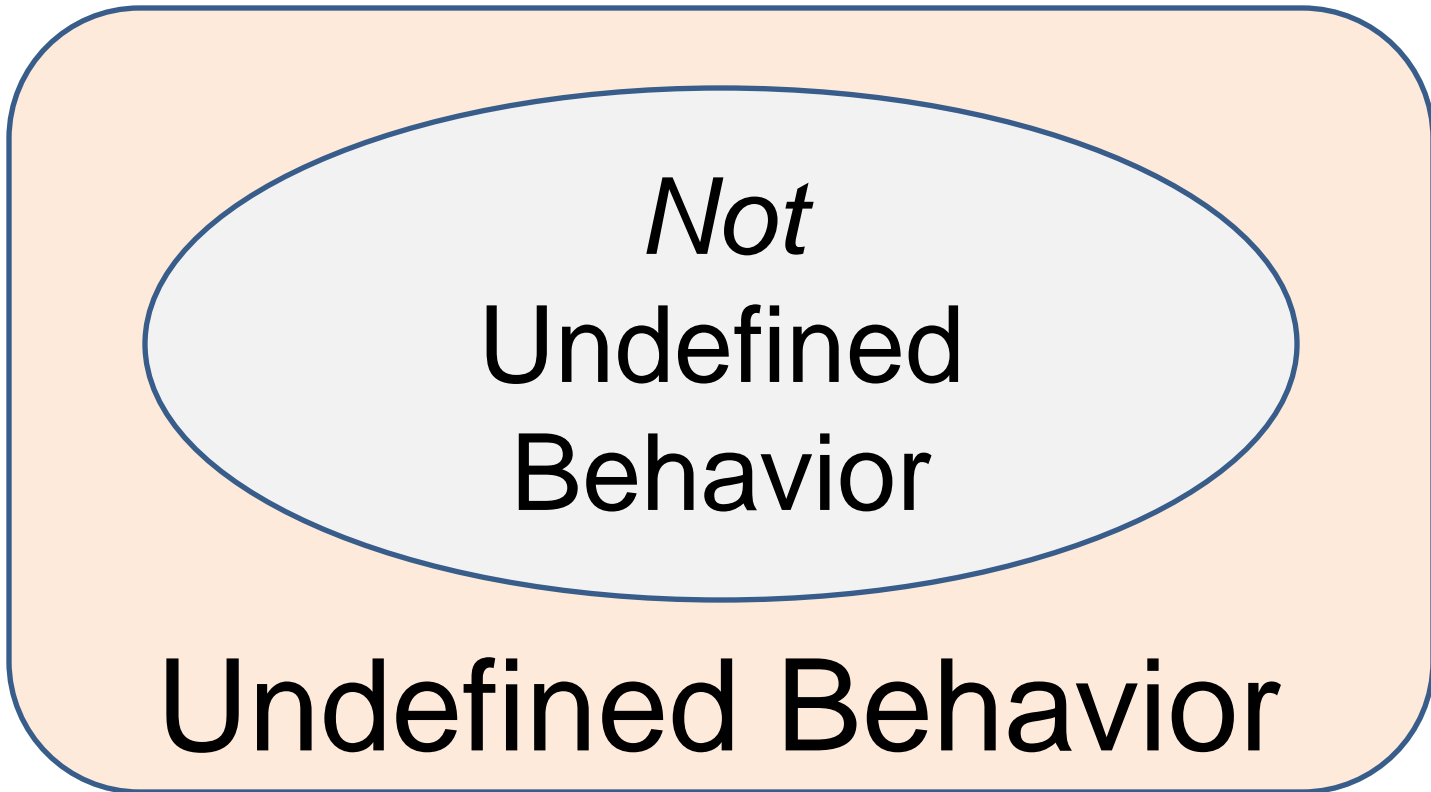
- Precondition: What must be true of both (object) state and method inputs otherwise the method is not allowed.
- Postconditions: What must happen as a function of (object) state and method inputs if all preconditions are satisfied.

Observation By
Kevlin Henny

2. Interfaces and Contracts (review)

Preconditions and Postconditions

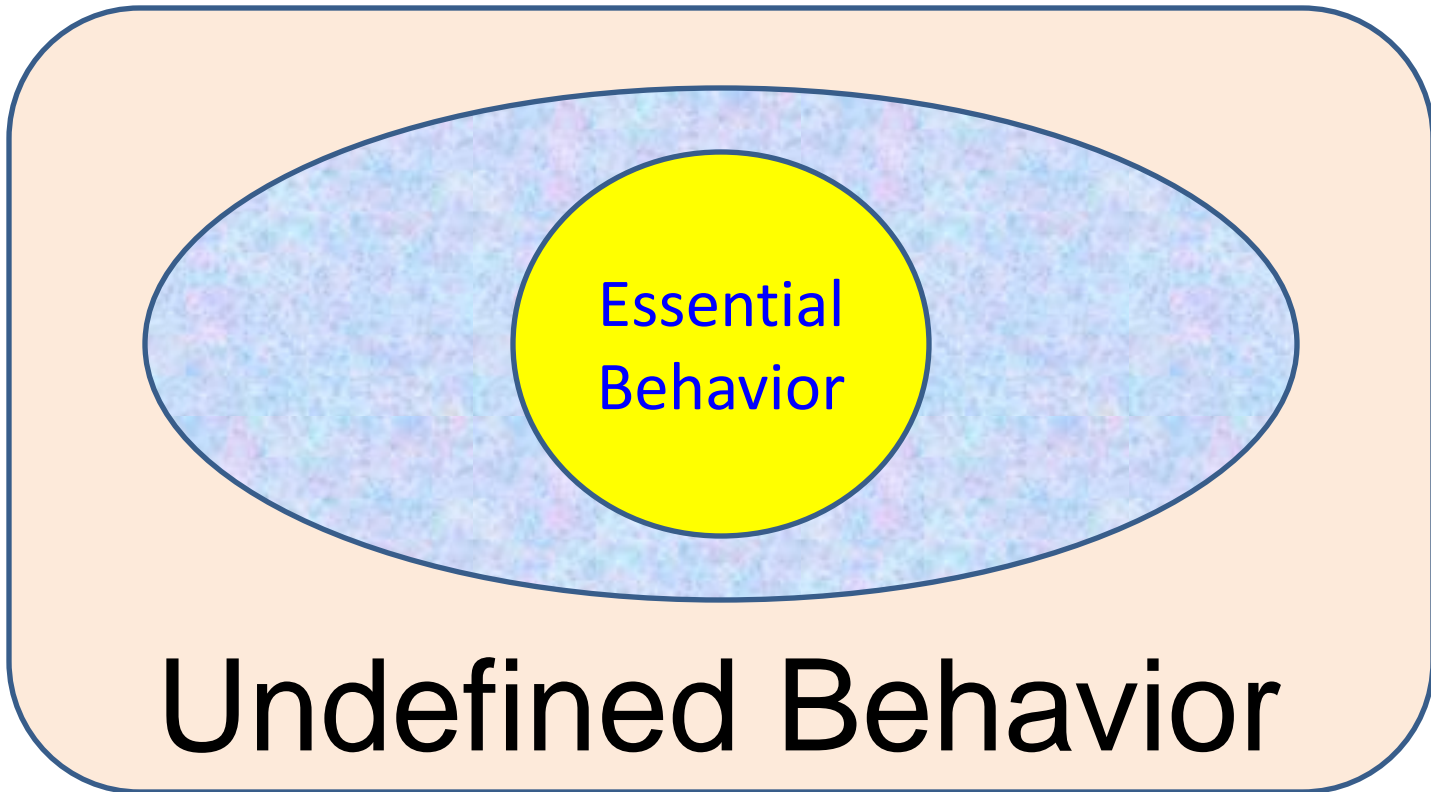
Defined & Essential Behavior



2. Interfaces and Contracts (review)

Preconditions and Postconditions

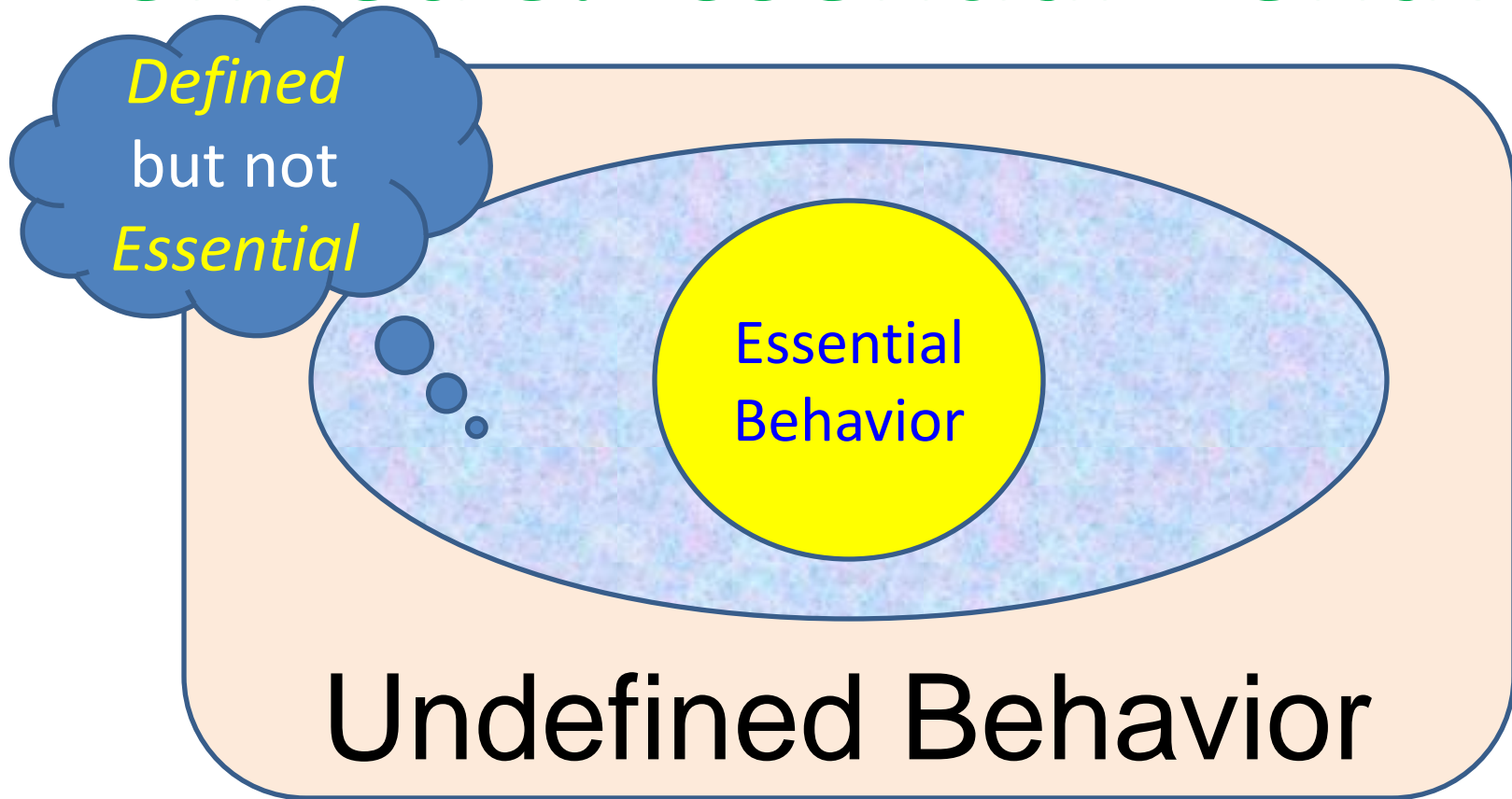
Defined & Essential Behavior



2. Interfaces and Contracts (review)

Preconditions and Postconditions

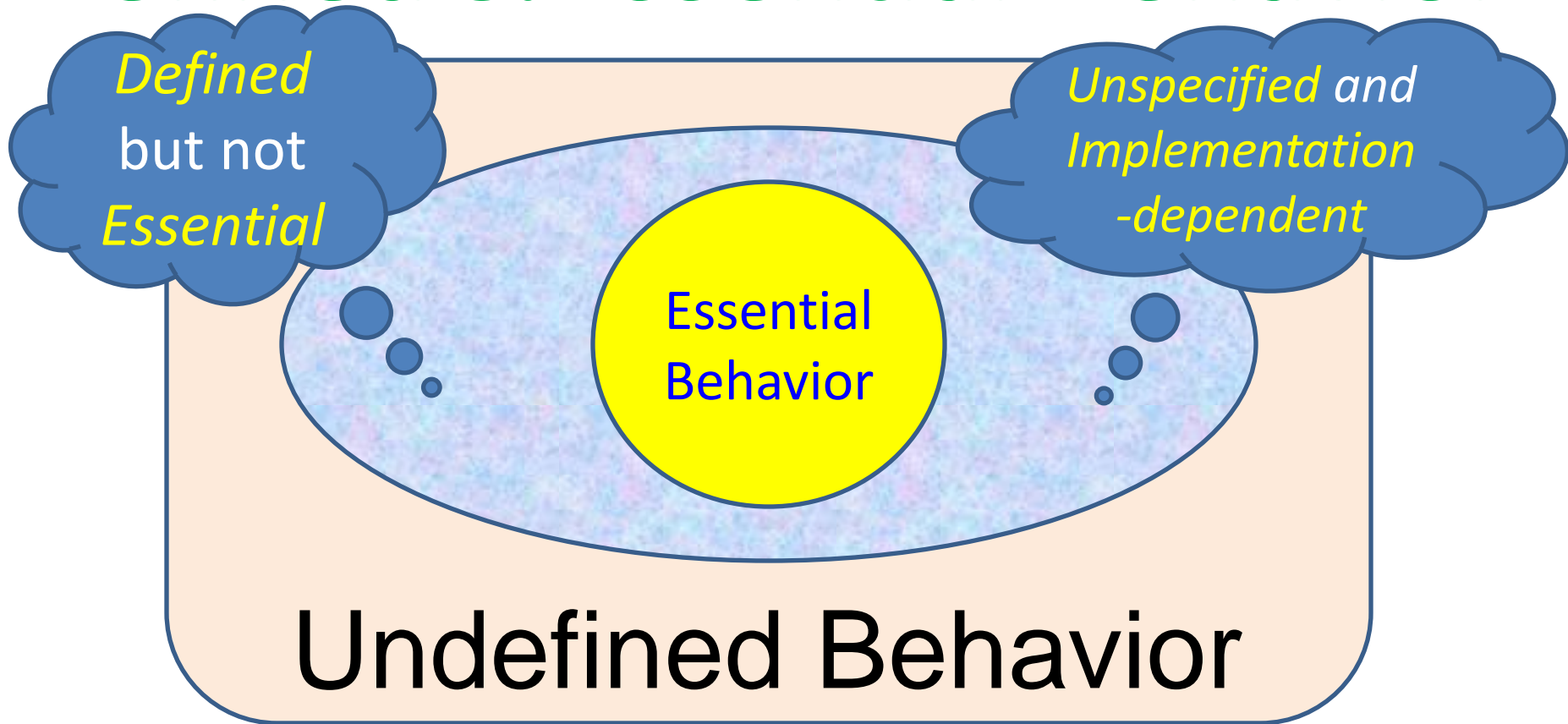
Defined & Essential Behavior



2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior



2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior

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

```
    int level = 0,
```

```
    int spacesPerLevel = 4) const;
```

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

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior

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

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior

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

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior

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

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior

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

Preconditions and Postconditions

Defined & Essential



Any
Undefined
Behavior?

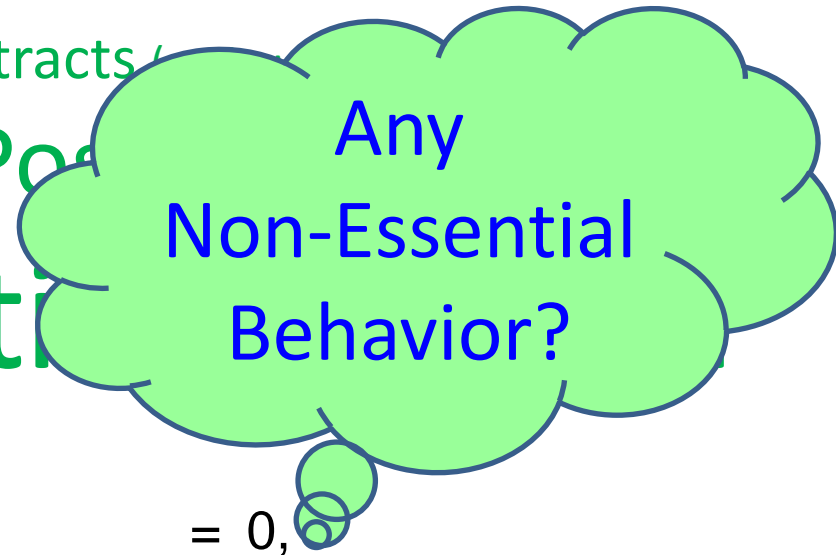
```
std::ostream& print(std::ostream& stream,  
                  int level = 0,  
                  int spacesPerLevel = 4) const;
```

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

2. Interfaces and Contracts /

Preconditions and Postconditions

Defined & Essential



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


Hint

2. Interfaces and Contracts /

Preconditions and Postconditions

Defined & Essential

Any
Non-Essential
Behavior?

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

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior

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

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior

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



2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior

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



2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior

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

2. Interfaces and Contracts (review)

Preconditions and Postconditions

Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date in history between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
    // Create a valid date from the specified year, month, and day.  
    // 'day'. The behavior is undefined if the date is not valid.  
    // represents a valid date in the range [0001/01/01, 9999/12/31].  
  
    Date(const Date& original);  
    // Create a date having the value of the specified 'original' date.  
    // ...  
};
```



2. Interfaces and Contracts (review)

Preconditions and Postconditions (Object) Invariants

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

2. Interfaces and Contracts (review)

Preconditions and Postconditions (Object) Invariants

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


2. Interfaces and Contracts (review)

Preconditions and Postconditions (Object) Invariants

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

```
//...
```

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

```
};
```

2. Interfaces and Contracts (review)

Preconditions and Postconditions (Object) Invariants

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

2. Interfaces and Contracts (review)

Preconditions and Postconditions (Object) Invariants

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

Answer: No!

2. Interfaces and Contracts (review)

Preconditions and Postconditions Variants

Semantic type representing the dates 0001/01/01 and

What happens when behavior is undefined is undefined!

Answer: No!

```
public:
```

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

```
// Create a valid date from the specified 'year', 'month', and  
// 'day'. The behavior is undefined unless 'year'/'month'/'day'  
// represents a valid date in the range [0001/01/01 .. 9999/12/31].
```

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

```
// Create a date having the value of the specified 'original' date.
```

```
// ...
```

```
};
```

2. Interfaces and Contracts (review)

Design by Contract

2. Interfaces and Contracts (review)

Design by Contract

(DbC)

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

*including state

2. Interfaces and Contracts (review)

Design by Contract

Documentation

There are five aspects:

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

2. Interfaces and Contracts (review)

Design by Contract

Documentation

There are five aspects:

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

2. Interfaces and Contracts (review)

Design by Contract

Documentation

There are five aspects:

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

2. Interfaces and Contracts (review)

Design by Contract

Documentation

There are five aspects:

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

2. Interfaces and Contracts (review)

Design by Contract

Documentation

There are five aspects:

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

2. Interfaces and Contracts (review)

Design by Contract

Documentation

There are five aspects:

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

2. Interfaces and Contracts (review)

Design by Contract

Verification

2. Interfaces and Contracts (review)

Design by Contract

Verification

➤ **Preconditions:**

2. Interfaces and Contracts (review)

Design by Contract

Verification

➤ **Preconditions:**

✓ RTFM (Read the Manual).

2. Interfaces and Contracts (review)

Design by Contract

Verification

➤ Preconditions:

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

**DEFENSIVE
PROGRAMMING**

2. Interfaces and Contracts (review)

Design by Contract

Verification

➤ **Preconditions:**

- ✓ RTFM (Read the Manual).

- ✓ Assert (only in *'debug'* or *'safe'* mode).

➤ **Postconditions:**

2. Interfaces and Contracts (review)

Design by Contract

Verification

➤ **Preconditions:**

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

➤ **Postconditions:**

- ✓ Component-level test drivers.

2. Interfaces and Contracts (review)

Design by Contract

Verification

➤ Preconditions:

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

➤ Postconditions:

- ✓ Component-level test drivers.

➤ Invariants:

2. Interfaces and Contracts (review)

Design by Contract

Verification

➤ Preconditions:

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

➤ Postconditions:

- ✓ Component-level test drivers.

➤ Invariants:

- ✓ Assert invariants in the destructor.

2. Interfaces and Contracts (review)

Contracts and Exceptions

Preconditions *always* Imply Postconditions:

2. Interfaces and Contracts (review)

Contracts and Exceptions

Preconditions always Imply Postconditions:

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

2. Interfaces and Contracts (review)

Contracts and Exceptions

Preconditions always Imply Postconditions:

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

2. Interfaces and Contracts (review)

Contracts and Exceptions

Preconditions always Imply Postconditions:

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

2. Interfaces and Contracts (review)

End of Section

Questions?

2. Interfaces and Contracts (review)

What Questions are we Answering?

- What do we mean by *Interface* versus *Contract* for a *function*, a *class*, or a *component*?
- What do we mean by *preconditions*, *postconditions*, and *invariants*?
- What do we mean by *essential* & *undefined behavior*?
- Must the code itself preserve invariants even if one or more *preconditions* of the contract are violated?
- What is the idea behind *Design-by-Contract (DbC)*?
- How do we document the contract for a function?
- How can clients ensure that preconditions are satisfied?
- How do we guarantee that postconditions are satisfied?
- How can we test to make sure invariants are preserved?
- What must be true if a client satisfies all preconditions?

Outline

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

4. Proper Inheritance

Is-A for *Interface*, *Structural*, & *Implementation* Inheritance

Outline

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

3. **Narrow versus Wide Contracts** (review)

The Significance of *Undefined Behavior*

4. Proper Inheritance

Is-A for *Interface*, *Structural*, & *Implementation* Inheritance

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Pejorative terms:

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Pejorative terms:

- ***Fat* Interface** (4. Proper Inheritance)

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Pejorative terms:

- *Fat* Interface (4. Proper Inheritance)
- ***Large* (Non-Primitive) Interface**

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Pejorative terms:

- *Fat* Interface (4. Proper Inheritance)
- *Large* (Non-Primitive) Interface
- ***Wide* Contract**

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

What should happen with the following call?

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

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

What should happen with the following call?

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

How about it must return 0?

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

How about it must return 0?

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

```
int strlen(const char *s)
{
    if (!s) return 0; } Wide
    // ... Likely to mask a defect
}
```

How about it must return 0?

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

```
int strlen(const char *s)
{
    if (!s) return 0; } Wide
    // ... Likely to mask a defect
}
```

How about it must return 0?

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

What should happen with the following call?

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


3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

What should happen with the following call?

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

Undefined Behavior

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

} **Narrow**

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

} Narrow

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

```
int strlen(const char*s)
{
    ...
}
```

Just Don't
PASS 0! } Narrow

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should

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

Return a status?

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should

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

Return a status?

Absolutely Not!

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

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

Therefore, why should I bother to check status?

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

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

Therefore, why should I bother to check status?

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


3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

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

Therefore, why should I bother to check status?

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

Double Fault!!

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

➤ Returning status implies a *wide* contract.

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

Narrow Contract:
Checked Only In
“Debug Mode”

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

Wide Contract:
Checked in
Every Build Mode

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

- What should happen when the behavior is undefined?

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

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

- What should happen when the behavior is undefined?

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

- Should what happens be part of the contract?

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

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

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

- Should what happens be part of the contract?

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

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

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

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

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

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

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

Must check
in **every**
build mode!

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

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

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

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

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

Must check
in **every**
build mode!

at happens be part

If it is, then it's esse

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

Bad
Idea!

rior!

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`
or less than zero?

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`
or less than zero? **If so, what should it be?**

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero? **If so, what should it be?**

```
if (idx < 0)           idx = 0;
```

```
if (idx > length())   idx = length();
```

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero? **If so, what should it be?**

```
if (idx < 0)         idx = 0;  
if (idx > length())  idx = length();  
idx = abs(idx) % (length() + 1);
```

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero? If so, what should it be?

```
if (idx < 0)           idx = 0;  
if (idx > length())   idx = length();  
idx = abs(idx) % (length() + 1);
```

More Code

Runs Slower!

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero? If so, what should it be?

```
if (idx < 0);  
if (idx > length());  
idx = abs(idx) % (length() + 1);
```

**Would Serve Only
To Mask Defects**

3. Narrow versus Wide Contracts (review)

Wide Contracts

No Undefined Behavior:

Should be defined when `idx` is greater than `length()` or less than zero? If so, what should it be?

```
if (idx < 0;
if (idx > length();
idx = abs(idx) % (length());
```

What happens when behavior is undefined is undefined!

Curve Only
Undefined Behavior

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`
or less than zero?

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`
or less than zero? Answer: **No!**

3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero? Answer: **No!**

```
assert(0 <= idx); assert(idx <= length());
```


3. Narrow versus Wide Contracts (review)

Narrow versus Wide Contracts

Narrow Contracts Imply Undefined Behavior:

Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero? Answer: **No!**

**DEFENSIVE
PROGRAMMING**

See the
`bsls_assert`
component.

3. Narrow versus Wide Contracts (review)

Appropriately Narrow Contracts

3. Narrow versus Wide Contracts (review)

Appropriately Narrow Contracts

Narrow, but not too narrow.

3. Narrow versus Wide Contracts (review)

Appropriately Narrow Contracts

Narrow, but not too narrow.

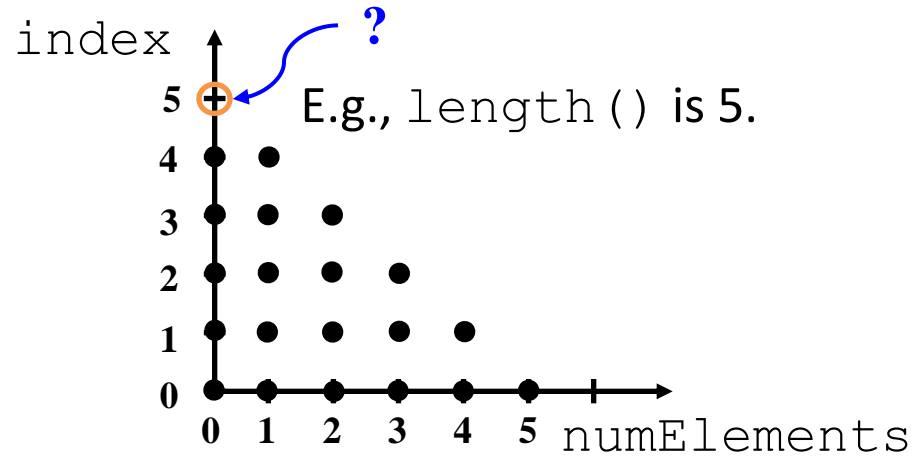
Should the behavior for

```
void replace(int index,  
             const TYPE& value,  
             int numElements);
```

be defined when `index` is `length()` and `numElements` is zero?

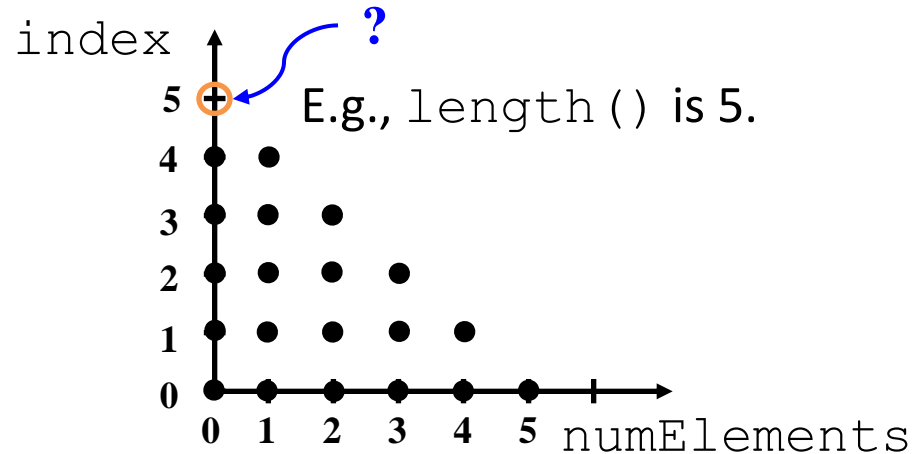
3. Narrow versus Wide Contracts (review)

Appropriately Narrow Contracts



3. Narrow versus Wide Contracts (review)

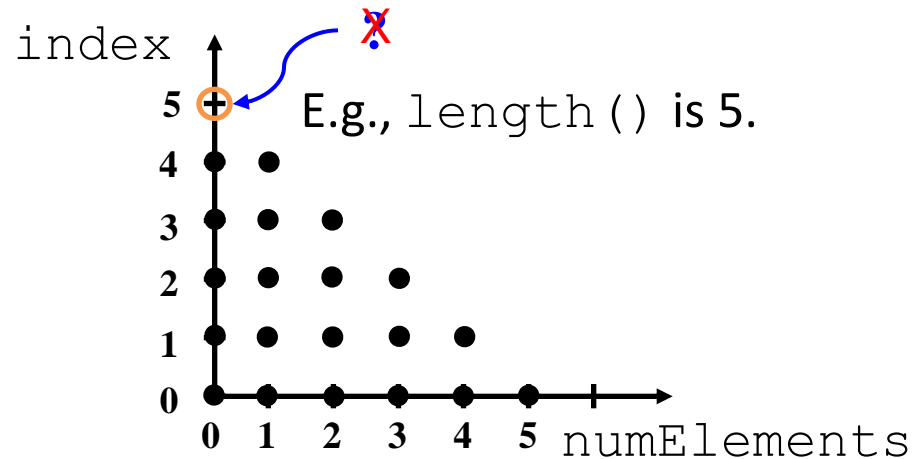
Appropriately Narrow Contracts



```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

3. Narrow versus Wide Contracts (review)

Appropriately Narrow Contracts

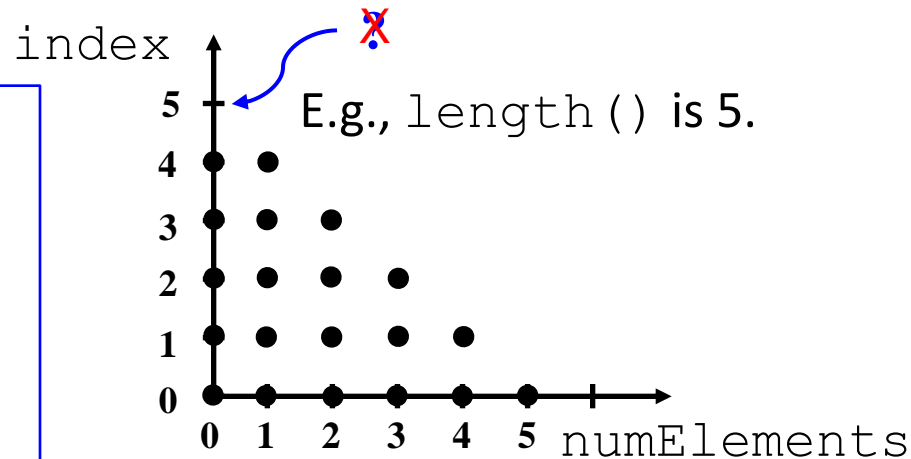


```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

3. Narrow versus Wide Contracts (review)

Appropriately Narrow Contracts

Now a client would have to check for this special case.

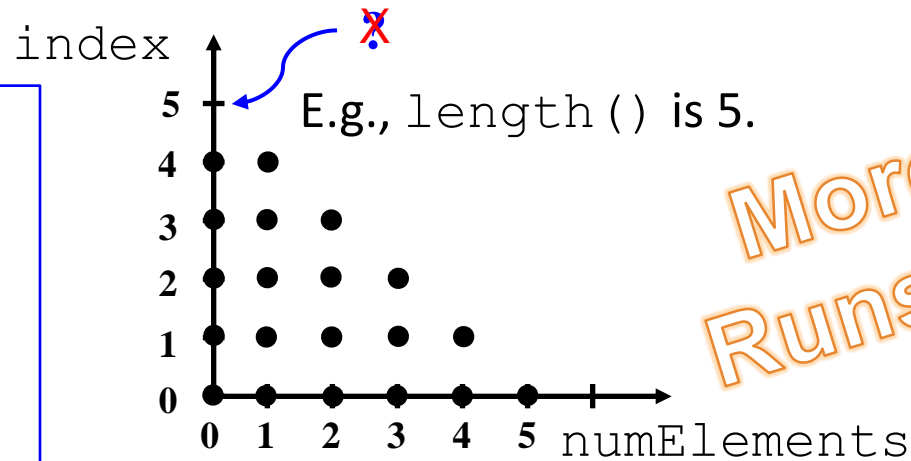


```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```


3. Narrow versus Wide Contracts (review)

Appropriately Narrow Contracts

Now a client would have to check for this special case.



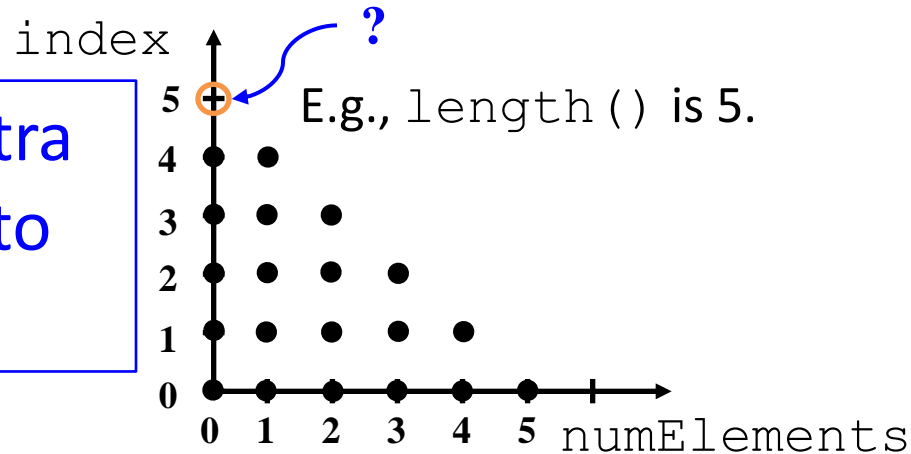
More Code
Runs Slower!

```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

3. Narrow versus Wide Contracts (review)

Appropriately Narrow Contracts

Assuming no extra code is needed to handle it ...

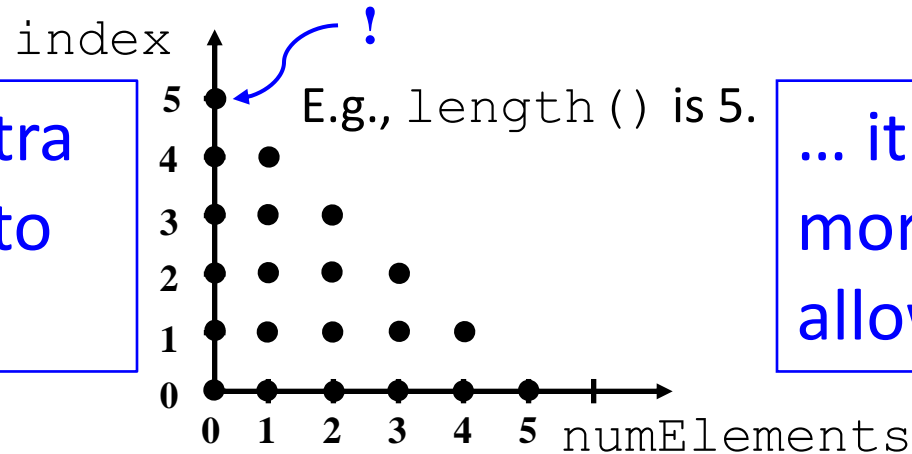


```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

3. Narrow versus Wide Contracts (review)

Appropriately Narrow Contracts

Assuming no extra code is needed to handle it ...



... it is naturally more efficient to allow it.

```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

3. Narrow versus Wide Contracts (review)

End of Section

Questions?

3. Narrow versus Wide Contracts (review)

What Questions are we Answering?

- What do we mean by a *narrow* versus a *wide* contract?
 - Should `std::strlen(0)` be required to do something reasonable?
 - Should `Date::setDate(int, int, int)` return a status?
- What should happen when the behavior is undefined?
 - Should what happens be part of the component-level contract?
- What about the behavior for these specific interfaces:
 - Should `operator[](int index)` check to see if `index` is less than zero or greater than `length()`?
 - And what should happen if `index` is out of range?
 - Should `insert(int index, const TYPE& value)` be defined when `index` is greater than `length()` or less than zero?
 - Should `replace(int index, const TYPE& value, int numElements)` be defined when `index` is `length()` and `numElements` is zero?
- What do we mean by *Defensive Programming (DP)*?

Outline

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

4. Proper Inheritance

Is-A for *Interface*, *Structural*, & *Implementation* Inheritance

Outline

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

4. Proper Inheritance

Is-A for *Interface*, *Structural*, & *Implementation* Inheritance

4. Proper Inheritance

Three Kinds of Inheritance

4. Proper Inheritance

Three Kinds of Inheritance

There are three kinds of inheritance because there are three kinds of member functions:

4. Proper Inheritance

Three Kinds of Inheritance

There are three kinds of inheritance because there are three kinds of member functions:

Public,
Protected,
Private?

4. Proper Inheritance

Three Kinds of Inheritance

There are three kinds of inheritance because there are three kinds of member functions:



4. Proper Inheritance

Three Kinds of Inheritance

There are three kinds of inheritance because there are three kinds of member functions:

4. Proper Inheritance

Three Kinds of Inheritance

There are three kinds of inheritance because there are three kinds of member functions:

- Interface Inheritance:
 - Pure Virtual Functions

4. Proper Inheritance

Three Kinds of Inheritance

There are three kinds of inheritance because there are three kinds of member functions:

- **Interface Inheritance:**
 - Pure Virtual Functions
- **Structural Inheritance:**
 - Non-Virtual Functions

4. Proper Inheritance

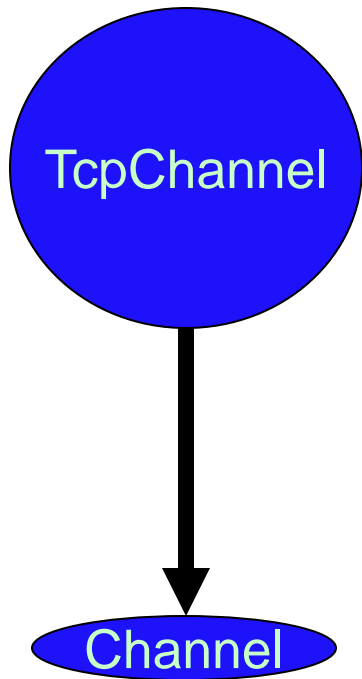
Three Kinds of Inheritance

There are three kinds of inheritance because there are three kinds of member functions:

- **Interface Inheritance:**
 - Pure Virtual Functions
- **Structural Inheritance:**
 - Non-Virtual Functions
- **Implementation Inheritance:**
 - Non-Pure Virtual Functions

4. Proper Inheritance

Interface Inheritance

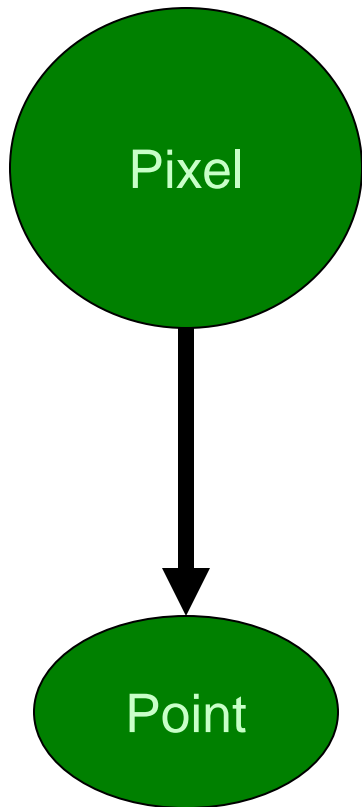


```
class TcpChannel : public Channel {
    /* ... */
public:
    // ... (creators)
    virtual int read(char *buffer, int numBytes) {...}
    virtual int write(const char *buffer, int numBytes) {...}
};
```

```
class Channel {
public:
    virtual ~Channel() { }
    virtual int read(char *buffer, int numBytes) = 0;
    virtual int write(const char *buffer, int numBytes) = 0;
};
```


4. Proper Inheritance

Structural Inheritance

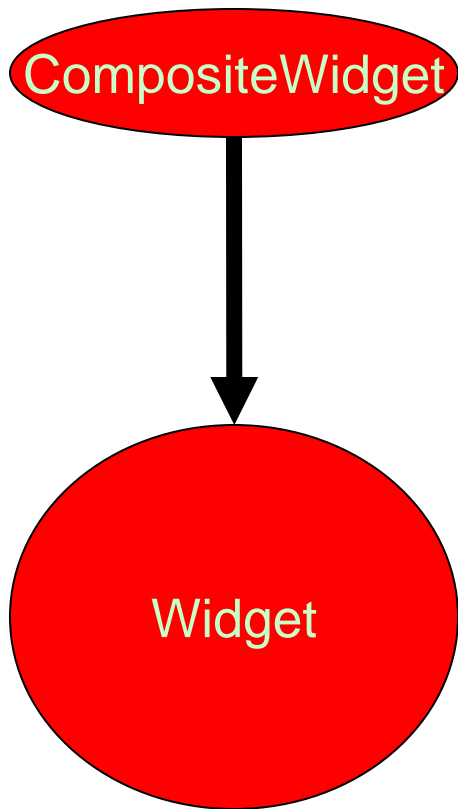


```
class Pixel : public Point {  
    public:  
        enum Color { RED, GREEN, BLUE };  
    private:  
        Color d_color;  
    public:  
        // ... (creators)  
        void setColor(Color color) { /* ... */ }  
        Color color ( ) const { /* ... */ }  
};
```

```
class Point {  
    int d_x;  
    int d_y;  
    public:  
        // ... (creators)  
        void setX(int x) { /* ... */ }  
        void setY(int y) { /* ... */ }  
        int x() const { /* ... */ }  
        int y() const { /* ... */ }  
};
```

4. Proper Inheritance

Implementation Inheritance

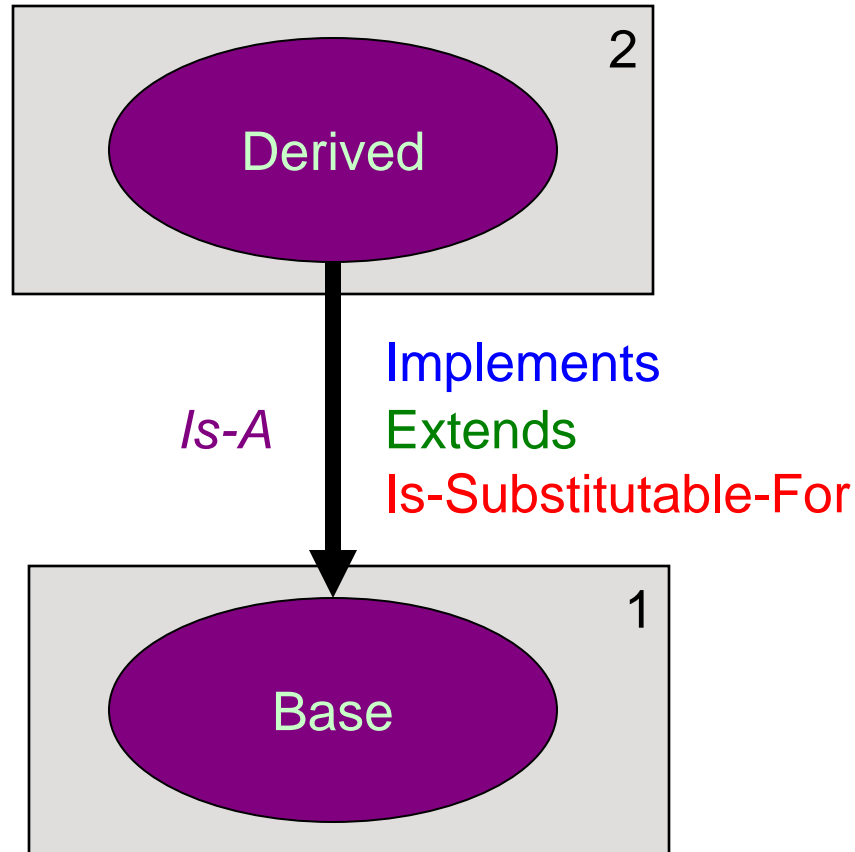


```
class CompositeWidget : public Widget {  
    // ...  
public:  
    // ... (creators)  
    virtual const char *widgetCategory() const { return "COMP"; }  
    virtual int numChildren() const { /* ... */ }  
    // ...  
};
```

```
class Widget {  
    Point d_origin;  
    // ...  
public:  
    // ... (creators)  
    virtual bool isNameable() const { return false; }  
    virtual const char *instanceName() const { return 0; }  
    virtual bool hasLocation() const { return true; }  
    virtual Point origin() const { return d_origin; }  
    virtual const char *widgetCategory() const { return "LEAF"; }  
    virtual int numChildren const { return 0; }  
    // ...  
};
```

4. Proper Inheritance

What Is Proper Inheritance?



4. Proper Inheritance

What Is Proper Inheritance?

- The “IsA” Relationship?
 - What does it mean?

4. Proper Inheritance

What Is Proper Inheritance?

- The “IsA” Relationship?
 - What does it mean?
- Weaker Preconditions?
- Stronger Postconditions?
- Same Invariants?

4. Proper Inheritance

What Is Proper Inheritance?

- The “IsA” Relationship?
 - What does it mean?
- Weaker Preconditions?
- Stronger Postconditions?
- Same Invariants?
- **Providing a Proper Superset of Behavior?**

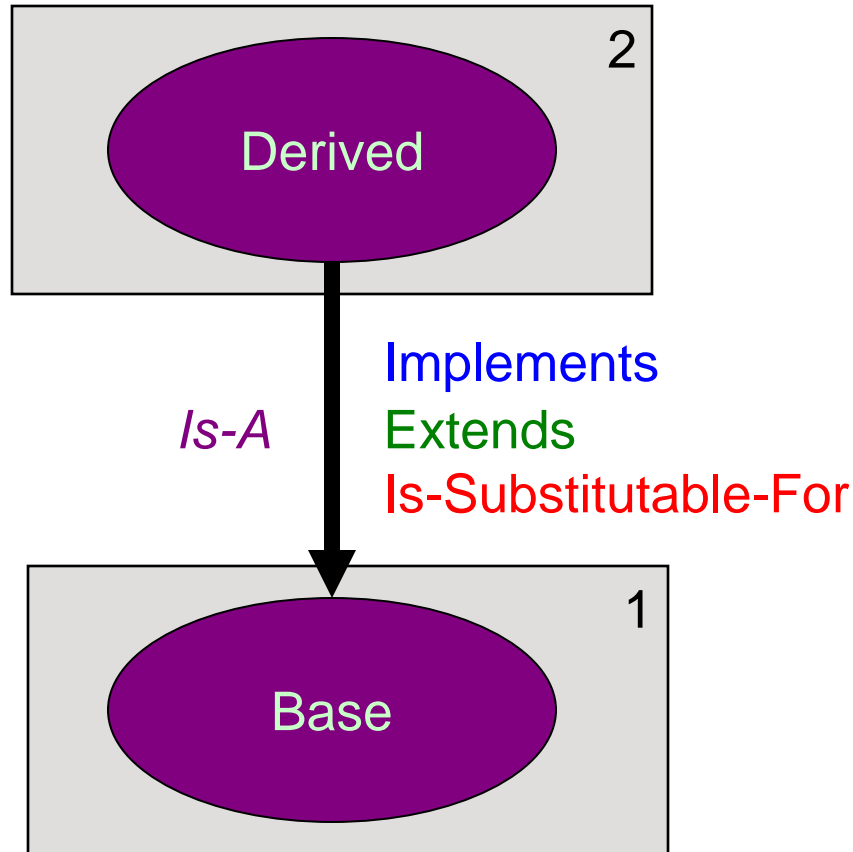
4. Proper Inheritance

What Is Proper Inheritance?

- The “IsA” Relationship?
 - What does it mean?
- Weaker Preconditions?
- Stronger Postconditions?
- Same Invariants?
- Providing a Proper Superset of Behavior?
- **Substitutability?**
 - Of what?
 - What criteria?

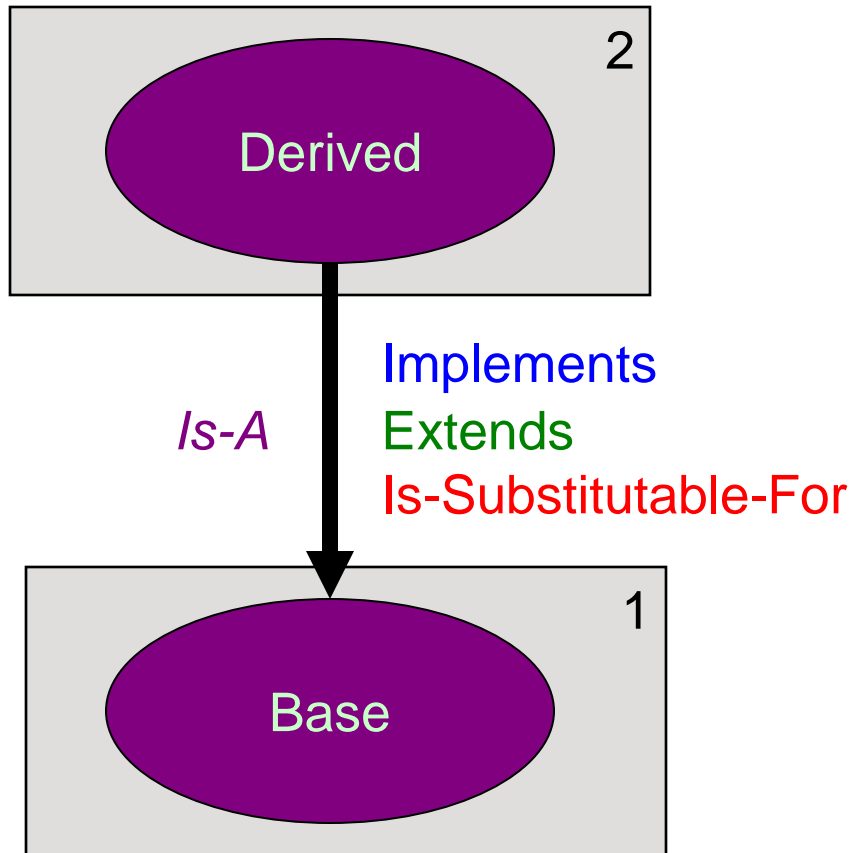
4. Proper Inheritance

What Is Proper Inheritance?



4. Proper Inheritance

What Is Proper Inheritance?



The *Is-A* Relation:

The implementation of a *derived* class must satisfy (simultaneously) its own contract, as well as that of “each” *base* class.

4. Proper Inheritance

What Is Proper Inheritance?

What about the following *general property*:

4. Proper Inheritance

What Is Proper Inheritance?

What about the following *general property*:

For inheritance to be *proper*, any operation that can be invoked on a derived-class *object* via a base-class pointer (or reference) must behave identically if we replace that base-class pointer (or reference) with a corresponding derived-class one.

4. Proper Inheritance

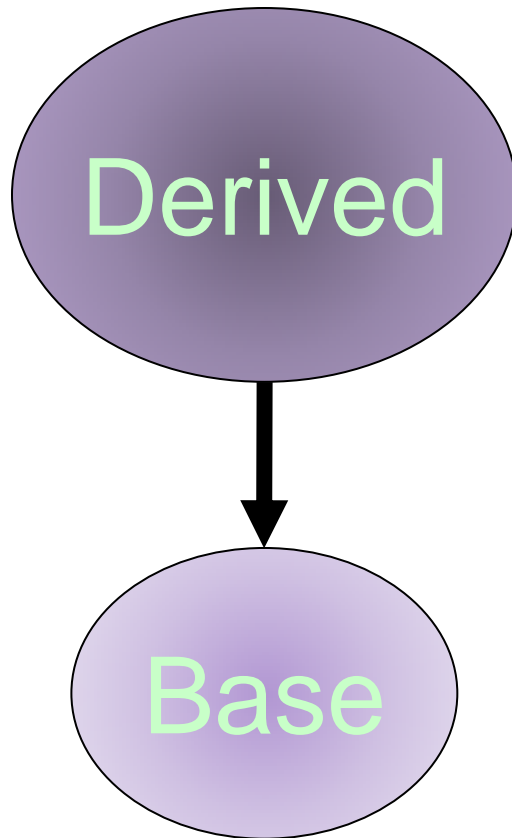
What Is Proper Inheritance?

What about the following *general property*:

For inheritance to be *proper*, any operation that can be invoked on a derived-class *object* via a base-class pointer (or reference) must behave identically if we replace that base-class pointer (or reference) with a corresponding derived-class one.

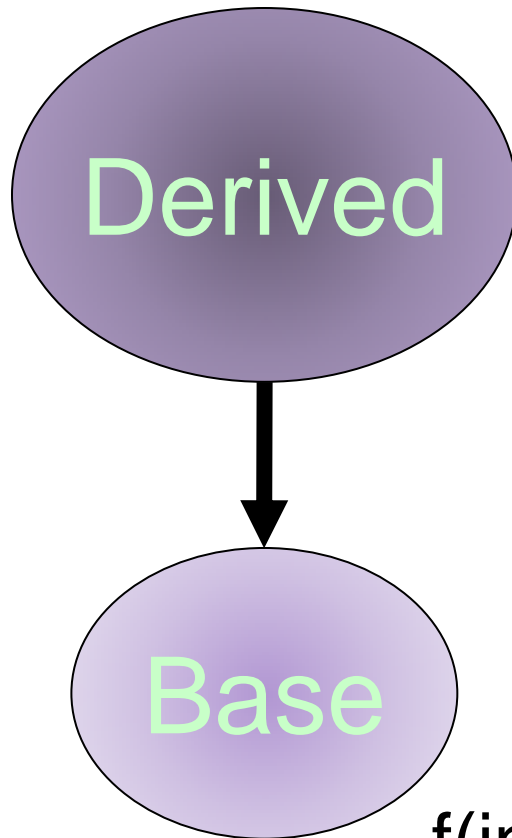
4. Proper Inheritance

What Is Proper Inheritance?



4. Proper Inheritance

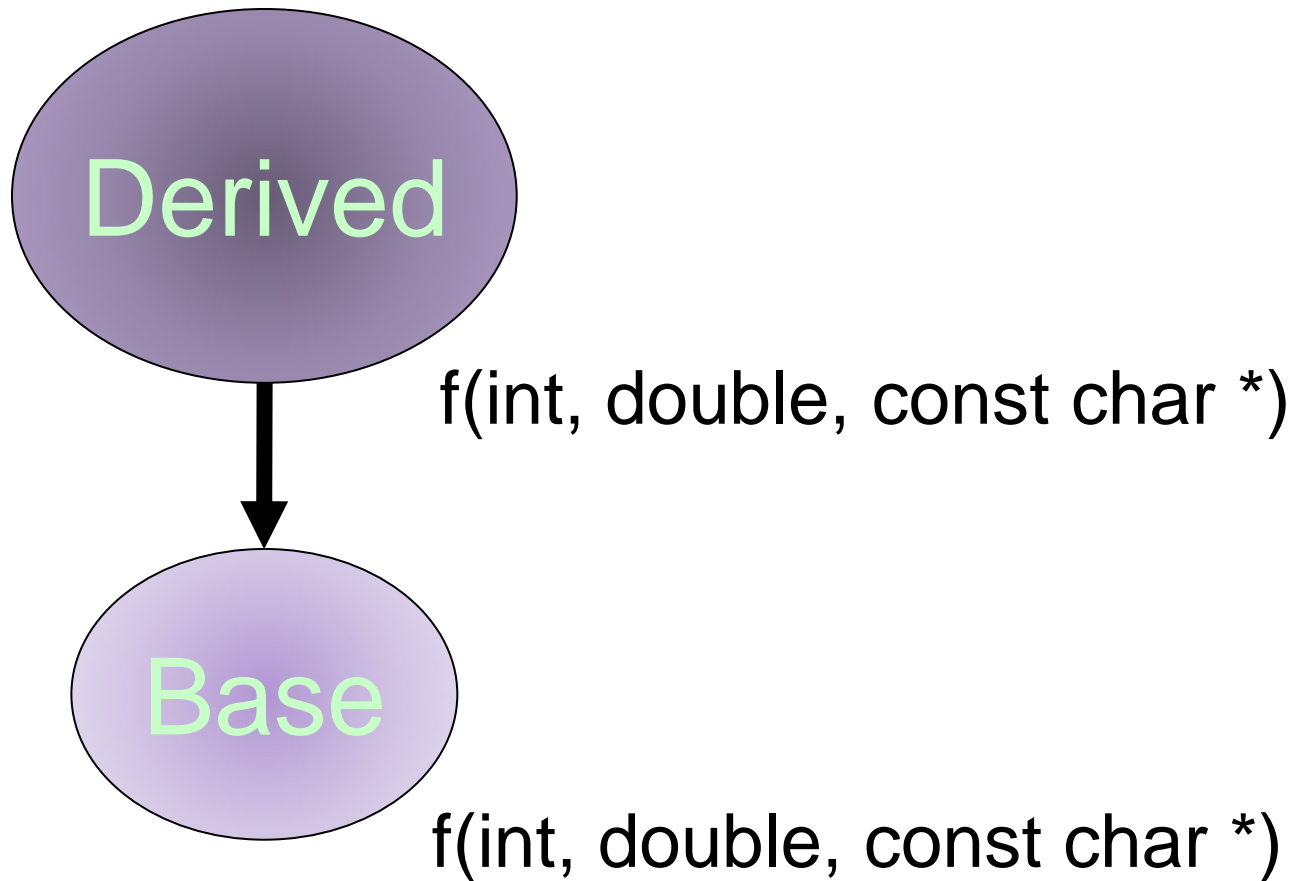
What Is Proper Inheritance?



f(int, double, const char *)

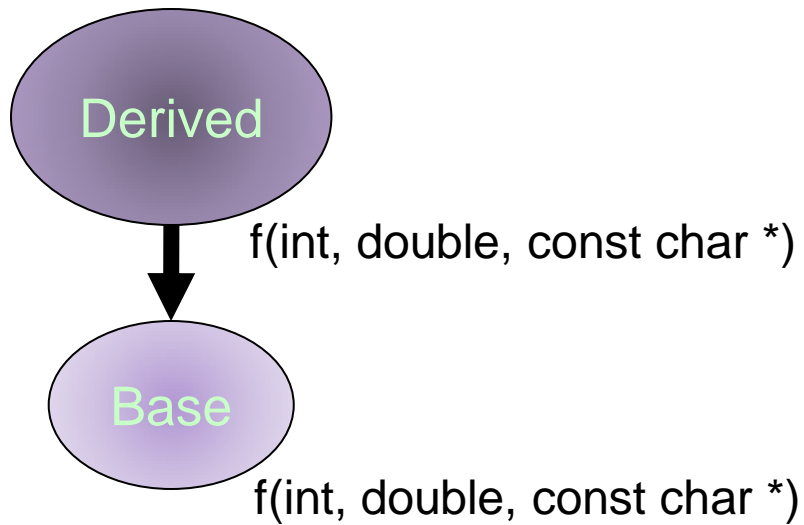
4. Proper Inheritance

What Is Proper Inheritance?



4. Proper Inheritance

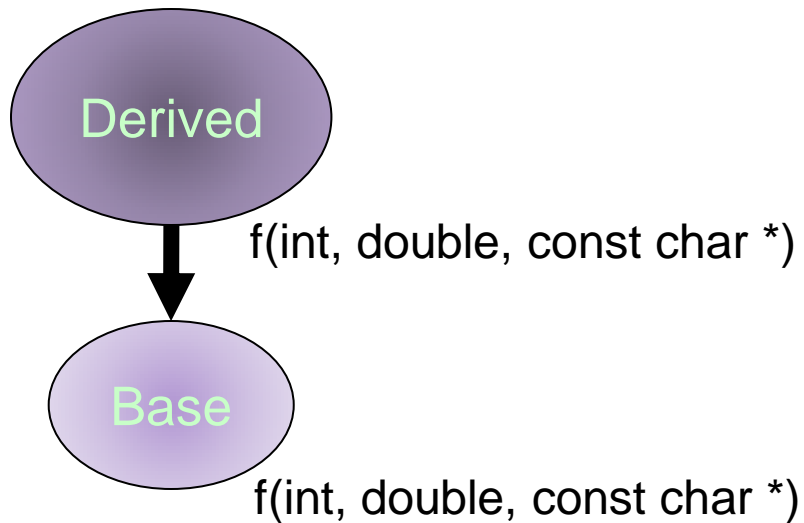
What Is Proper Inheritance?



4. Proper Inheritance

What Is Proper Inheritance?

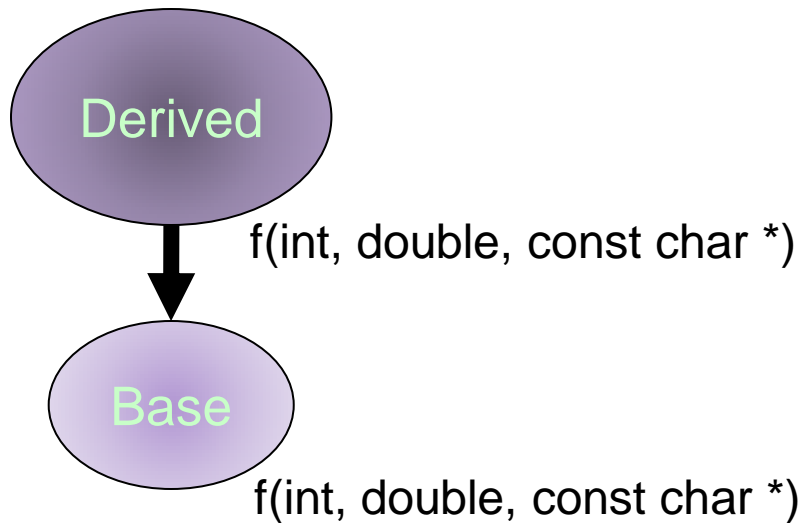
```
Derived *dp = new Derived();
```



4. Proper Inheritance

What Is Proper Inheritance?

```
Derived *dp = new Derived();
```

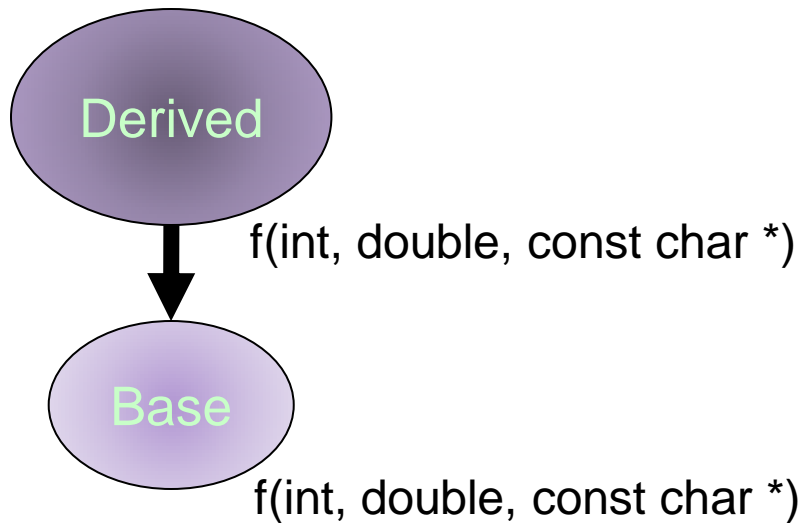


```
0x002130:  
0x002138:  
0x002140:  
0x002148:  
0x002150:  
0x002158:  
0x002160:  
0x002168:  
0x002170:
```

4. Proper Inheritance

What Is Proper Inheritance?

```
Derived *dp = new Derived();
```

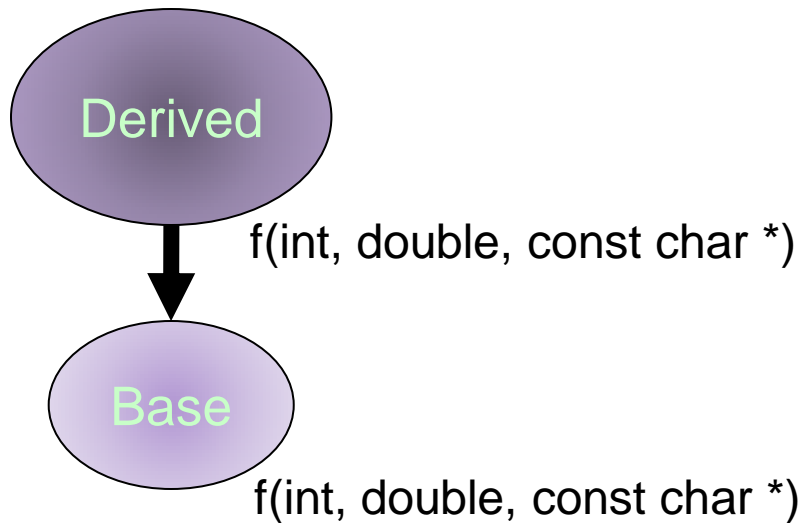


```
0x002130 :  
0x002138 :  
0x002140 :  
0x002148 :  
0x002150 :  
0x002158 :  
0x002160 :  
0x002168 :  
0x002170 :
```

4. Proper Inheritance

What Is Proper Inheritance?

```
Derived *dp = new Derived();
```



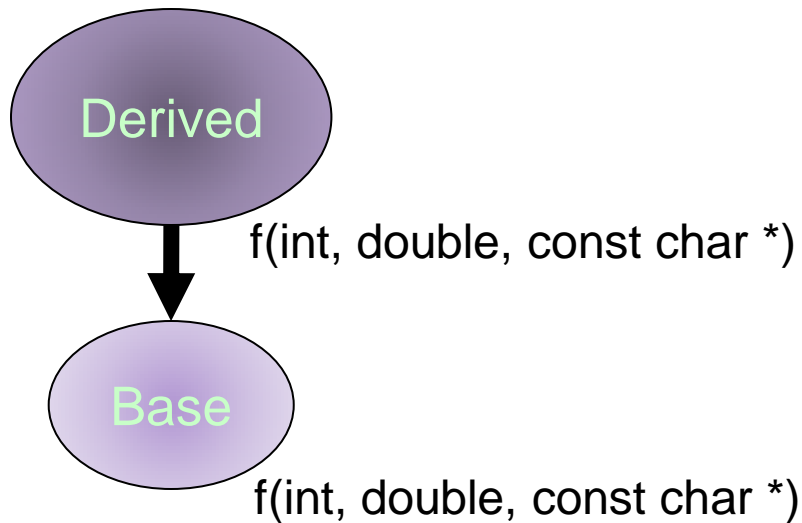
```
0x002130 :  
0x002138 :  
0x002140 :  
0x002148 :  
0x002150 :  
0x002158 :  
0x002160 :  
0x002168 :  
0x002170 :
```

Object
of type
Derived

4. Proper Inheritance

What Is Proper Inheritance?

```
Derived *dp = new Derived(); // dp = 0x002140
```



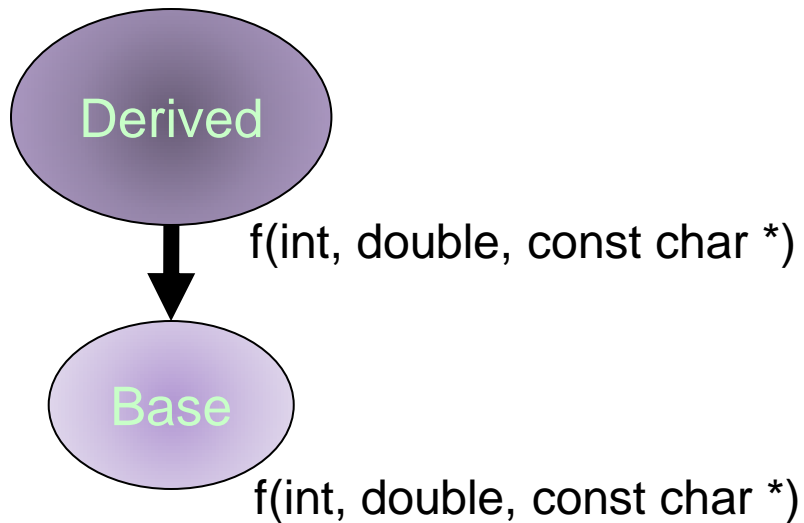
```
0x002130 :  
0x002138 :  
0x002140 :  
0x002148 :  
0x002150 :  
0x002158 :  
0x002160 :  
0x002168 :  
0x002170 :
```

Object
of type
Derived

4. Proper Inheritance

What Is Proper Inheritance?

```
Derived *dp = new Derived(); // dp = 0x002140
```



```
0x002130:  
0x002138:  
0x002140:  
0x002148:  
0x002150:  
0x002158:  
0x002160:  
0x002168:  
0x002170:
```

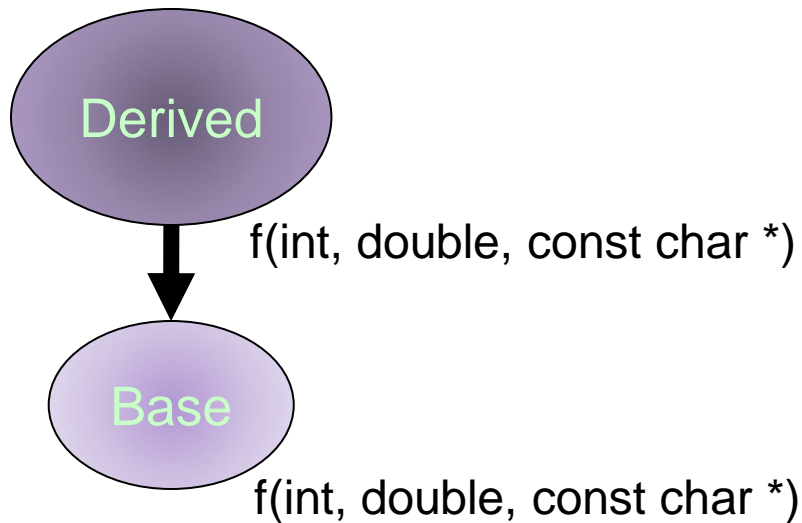
Object
of type
Derived

4. Proper Inheritance

What Is Proper Inheritance?

```
Derived *dp = new Derived(); // dp = 0x002140
```

```
Base *bp = dp; // bp = 0x002140
```



0x002130:

0x002138:

0x002140:

0x002148:

0x002150:

0x002158:

0x002160:

0x002168:

0x002170:

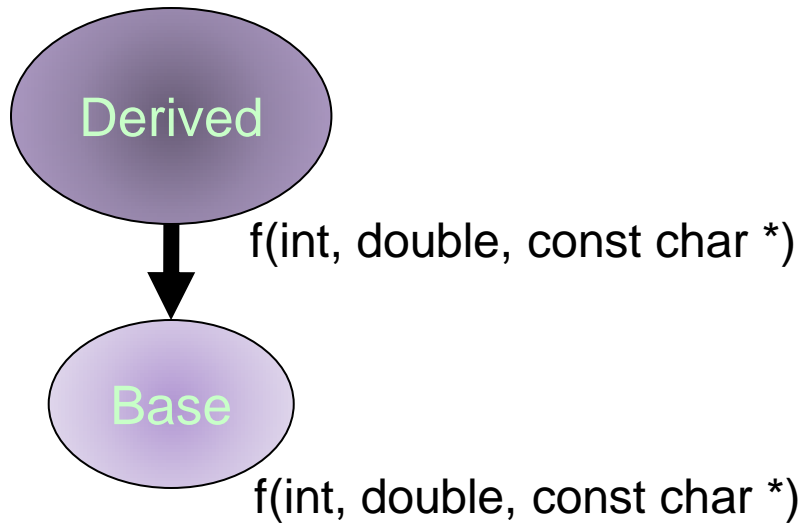
Object
of type
Derived

4. Proper Inheritance

What Is Proper Inheritance?

```
Derived *dp = new Derived(); // dp = 0x002140
```

```
Base *bp = dp; // bp = 0x002140
```



0x002130:

0x002138:

0x002140:

0x002148:

0x002150:

0x002158:

0x002160:

0x002168:

0x002170:

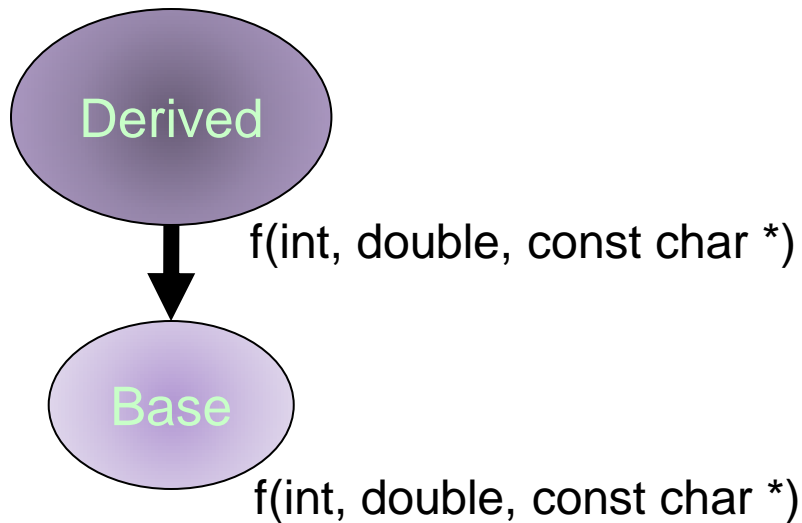
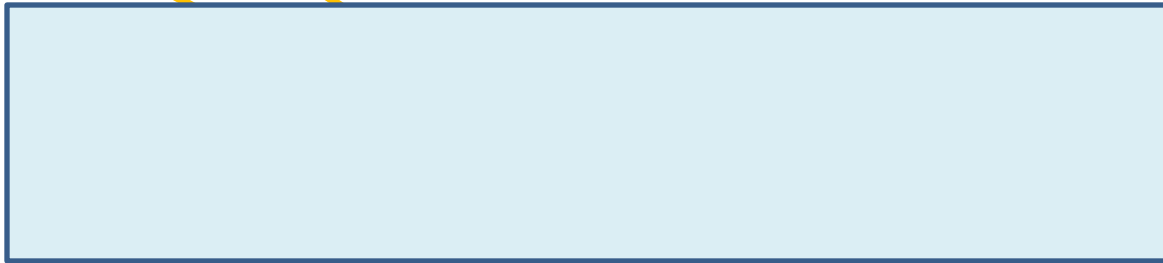


4. Proper Inheritance

What Is Proper Inheritance?

```
Derived *dp = new Derived(); // dp = 0x002140
```

```
Base *bp = dp; // bp = 0x002140
```



0x002130:

0x002138:

0x002140:

0x002148:

0x002150:

0x002158:

0x002160:

0x002168:

0x002170:



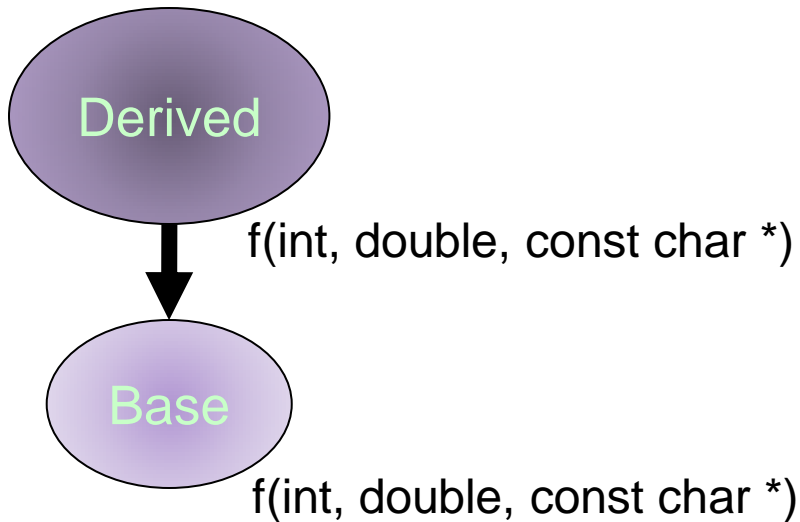
4. Proper Inheritance

What Is Proper Inheritance?

```
Derived *dp = new Derived(); // dp = 0x002140
```

```
Base *bp = dp; // bp = 0x002140
```

```
bp->f(1, 2.0, "three");
```



```
0x002130:
```

```
0x002138:
```

```
0x002140:
```

```
0x002148:
```

```
0x002150:
```

```
0x002158:
```

```
0x002160:
```

```
0x002168:
```

```
0x002170:
```

Object
of type
Derived

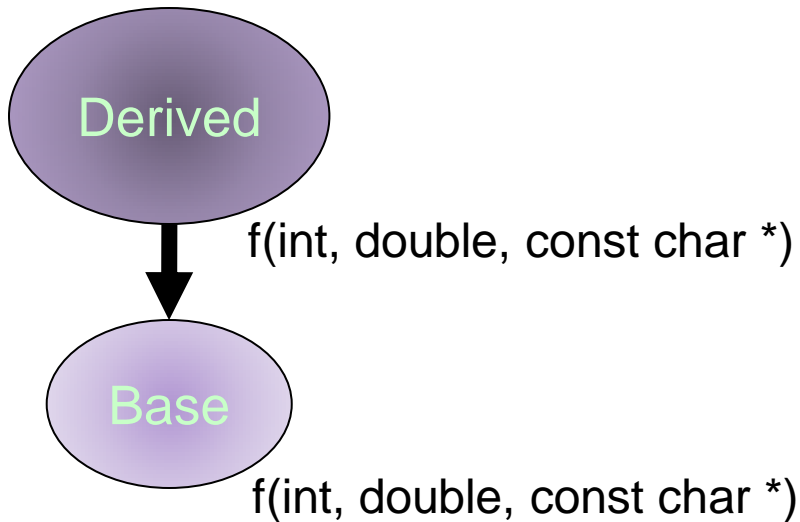
4. Proper Inheritance

What Is Proper Inheritance?

```
Derived *dp = new Derived(); // dp = 0x002140
```

```
Base *bp = dp; // bp = 0x002140
```

```
bp->f(1, 2.0, "three");  
dp->f(1, 2.0, "three");
```



```
0x002130 :  
0x002138 :  
0x002140 :  
0x002148 :  
0x002150 :  
0x002158 :  
0x002160 :  
0x002168 :  
0x002170 :
```

Object
of type
Derived

4. Proper Inheritance

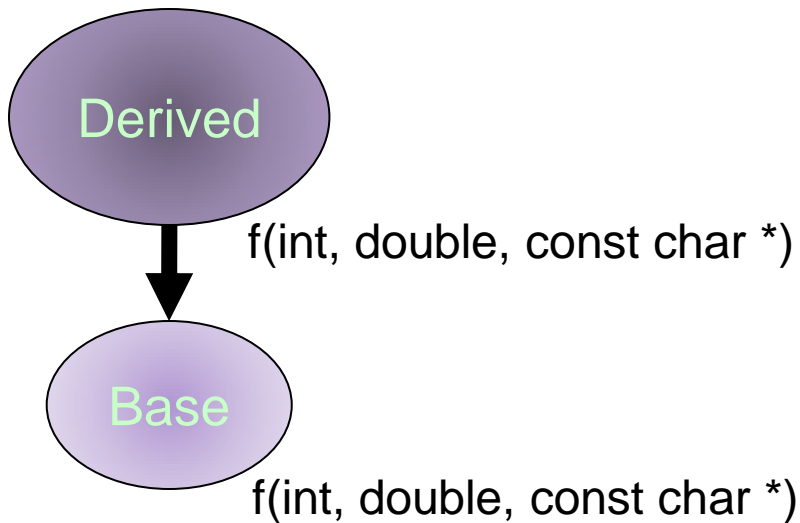
What Is Proper Inheritance?

```
Derived *dp = new Derived(); // dp = 0x002140
```

```
Base *bp = dp; // bp = 0x002140
```

```
bp->f(1, 2.0, "three");  
dp->f(1, 2.0, "three");
```

Identical Behavior



0x002130:

0x002138:

0x002140:

0x002148:

0x002150:

0x002158:

0x002160:

0x002168:

0x002170:

**Object
of type
Derived**

4. Proper Inheritance

What Is Proper Inheritance?

What about the following *general property*:

For inheritance to be *proper*, any operation that can be invoked on a derived-class *object* via a base-class pointer (or reference) must behave identically if we replace that base-class pointer (or reference) with a corresponding derived-class one.

4. Proper Inheritance

What Is Proper Inheritance?

What about the following *general property*:

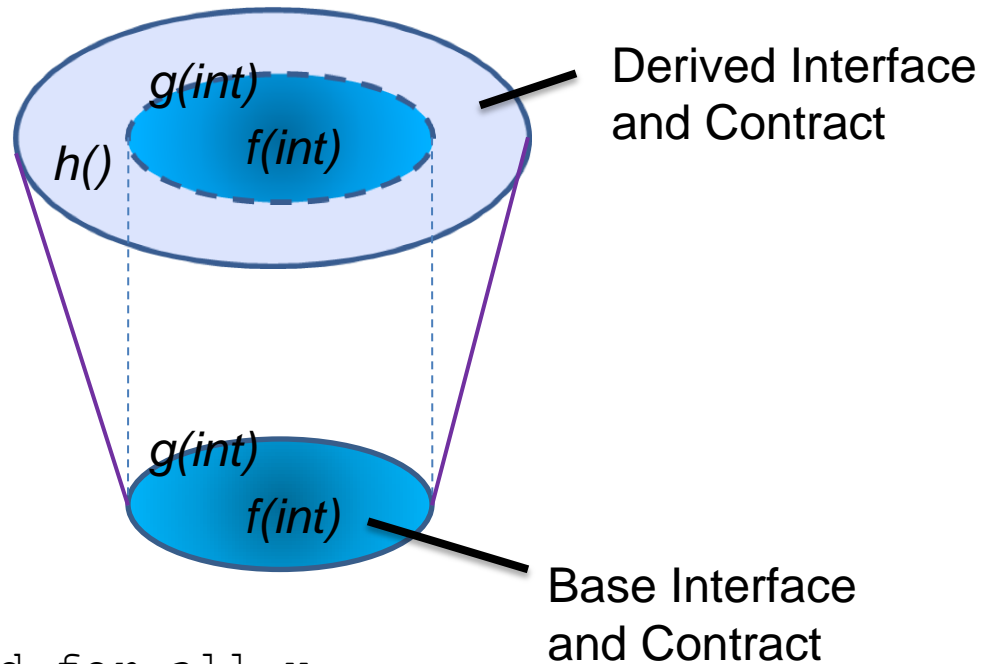
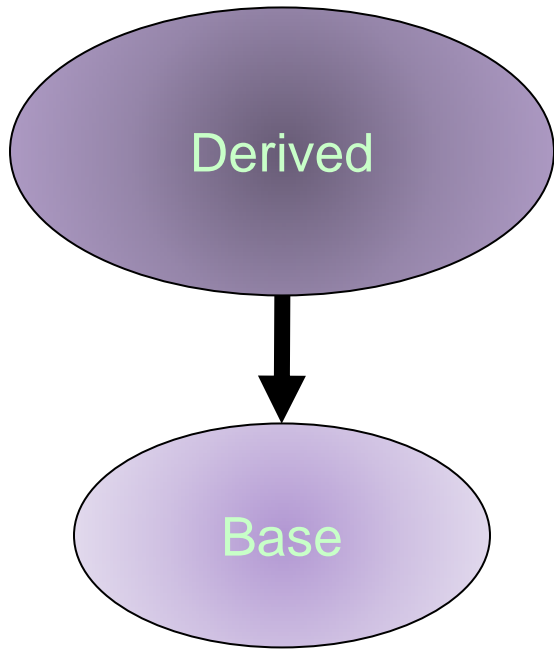
For inheritance to be *proper*, any operation that can be invoked on a derived-class *object* via a base-class pointer (or reference) must behave identically if we replace that base-class pointer (or reference) with a corresponding derived-class one.

Note that this is how **virtual functions** behave!

4. Proper Inheritance

What Is Proper Inheritance?

```
Derived::f(int x); // Defined for all x.  
Derived::g(int x); // Defined for all x.  
Derived::h(); // Note: not accessible from Base class.
```

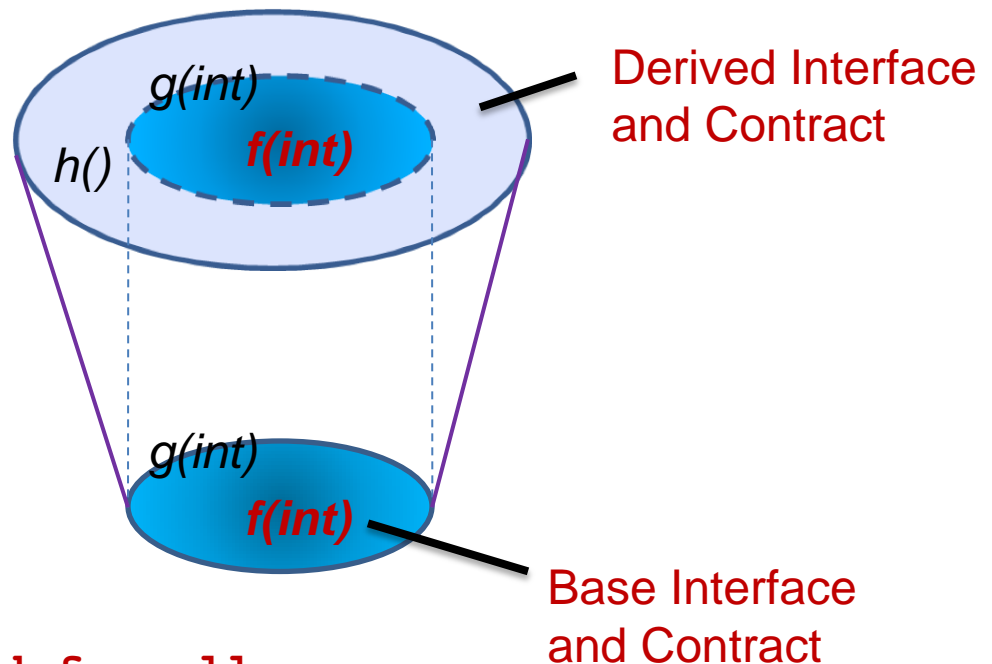
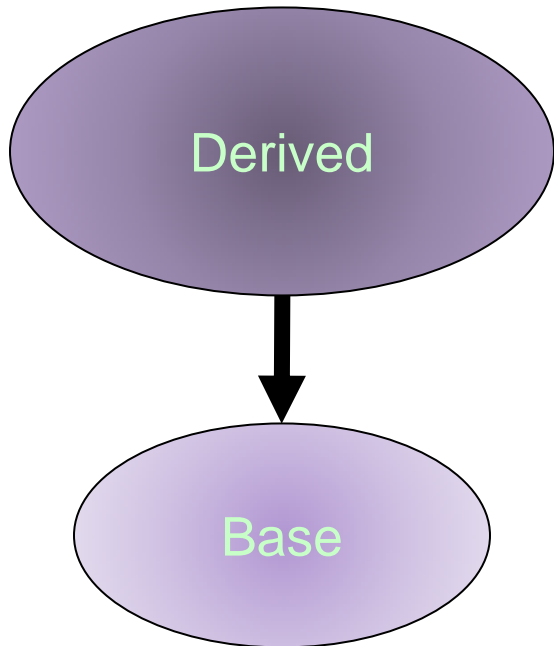


```
Base::f(int x); // Defined for all x.  
Base::g(int x); // Defined only for  $0 \leq x$ .
```

4. Proper Inheritance

What Is Proper Inheritance?

```
Derived::f(int x); // Defined for all x.  
Derived::g(int x); // Defined for all x.  
Derived::h(); // Note: not accessible from Base class.
```

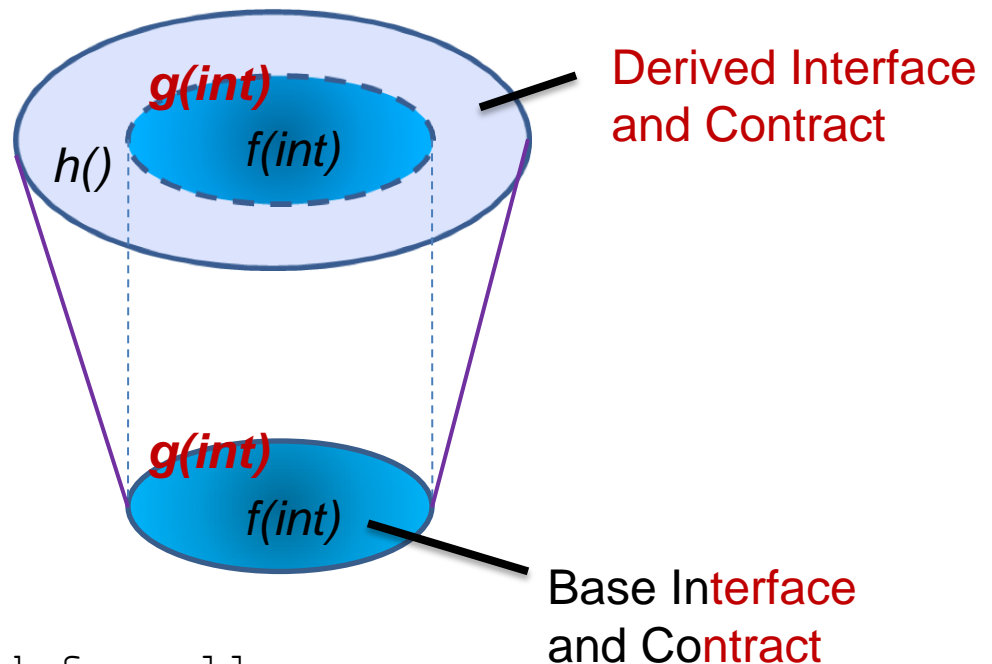
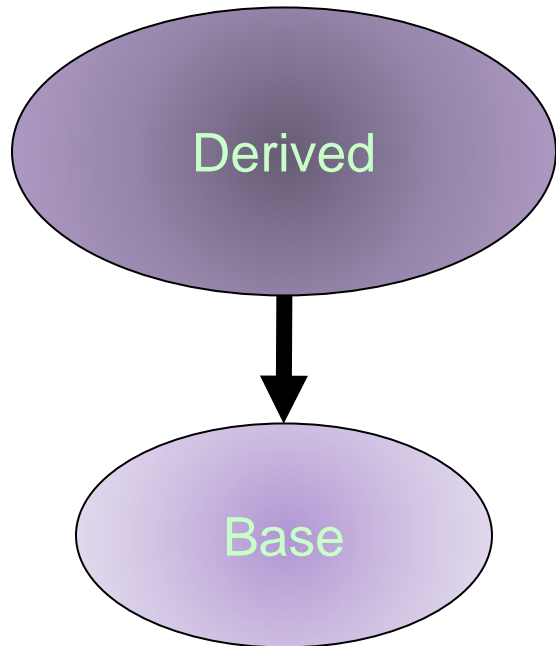


```
Base::f(int x); // Defined for all x.  
Base::g(int x); // Defined only for  $0 \leq x$ .
```


4. Proper Inheritance

What Is Proper Inheritance?

```
Derived::f(int x); // Defined for all x.  
Derived::g(int x); // Defined for all x.  
Derived::h(); // Note: not accessible from Base class.
```

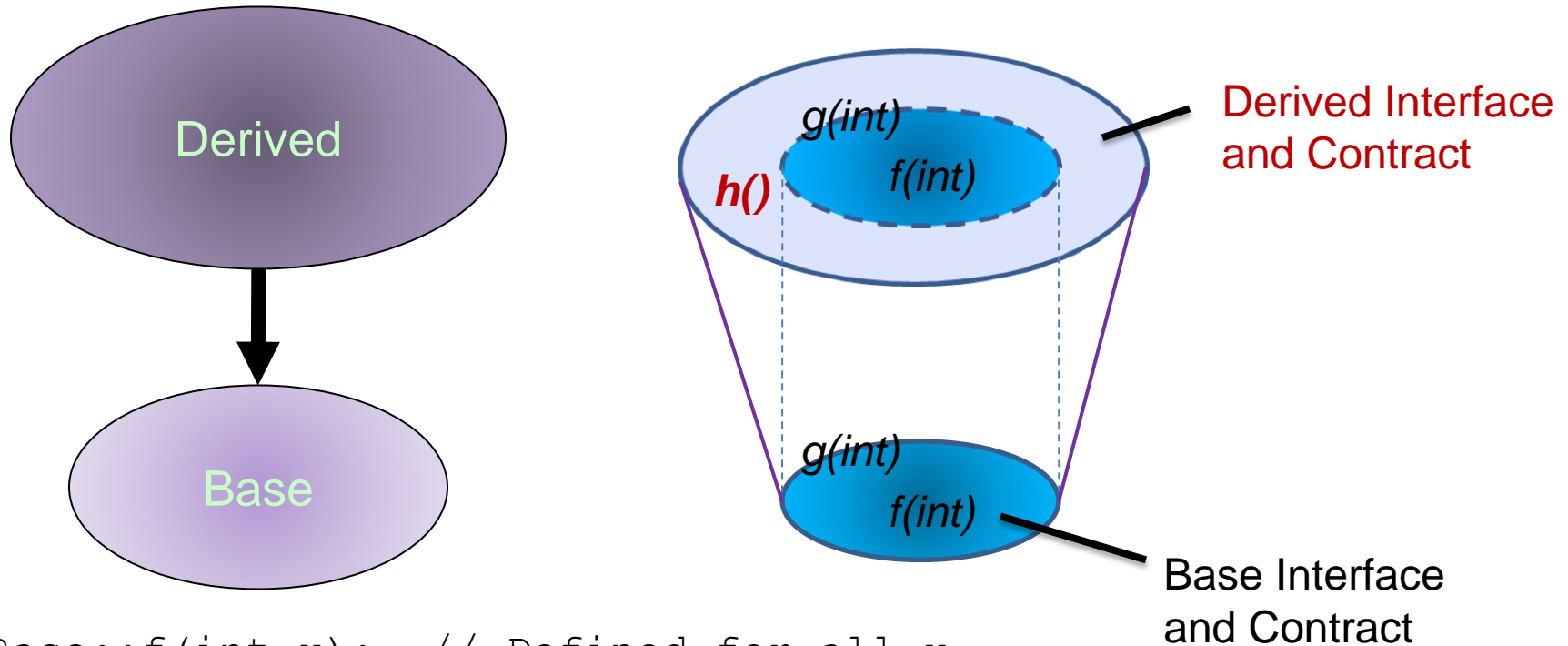


```
Base::f(int x); // Defined for all x.  
Base::g(int x); // Defined only for  $0 \leq x$ .
```

4. Proper Inheritance

What Is Proper Inheritance?

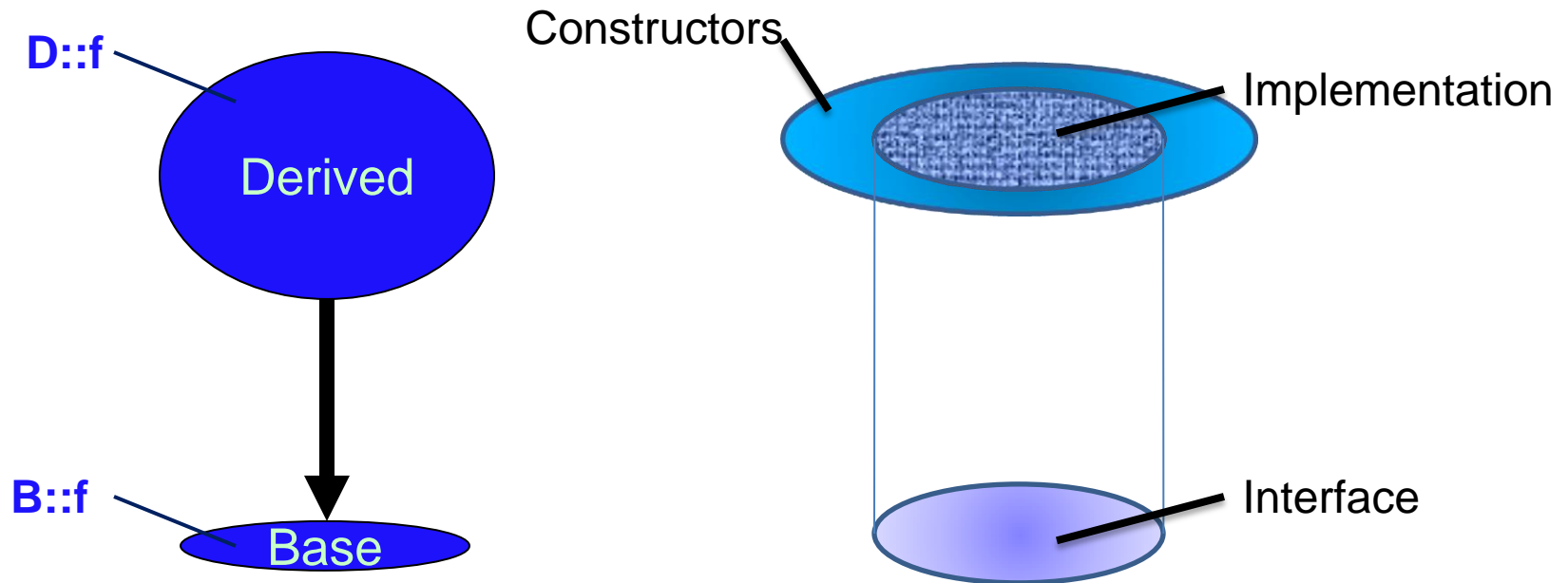
```
Derived::f(int x); // Defined for all x.  
Derived::g(int x); // Defined for all x.  
Derived::h(); // Note: not accessible from Base class.
```



```
Base::f(int x); // Defined for all x.  
Base::g(int x); // Defined only for 0 <= x.
```

4. Proper Inheritance

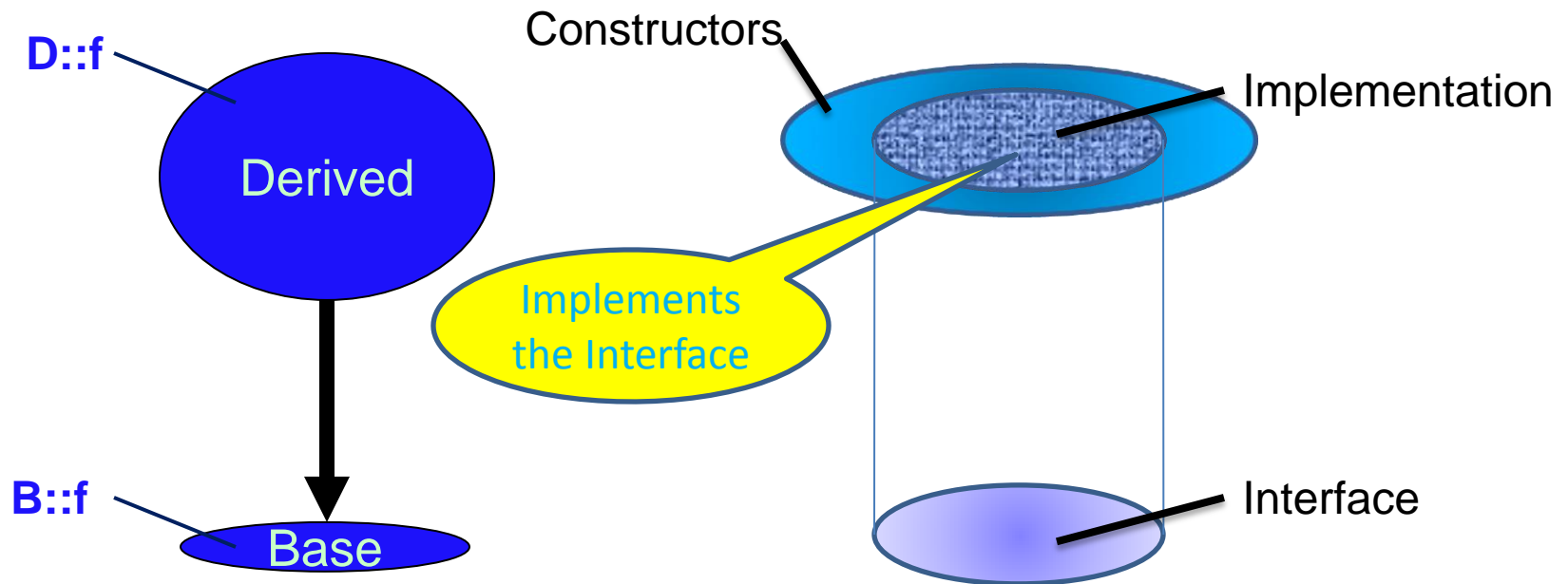
Pure Interface Inheritance



For each function **D::f** in the derived class overriding a virtual one **B::f** in the base class, the (documented) *preconditions* of **D::f** must be no stronger than those for **B::f**, and the *postconditions* no weaker.

4. Proper Inheritance

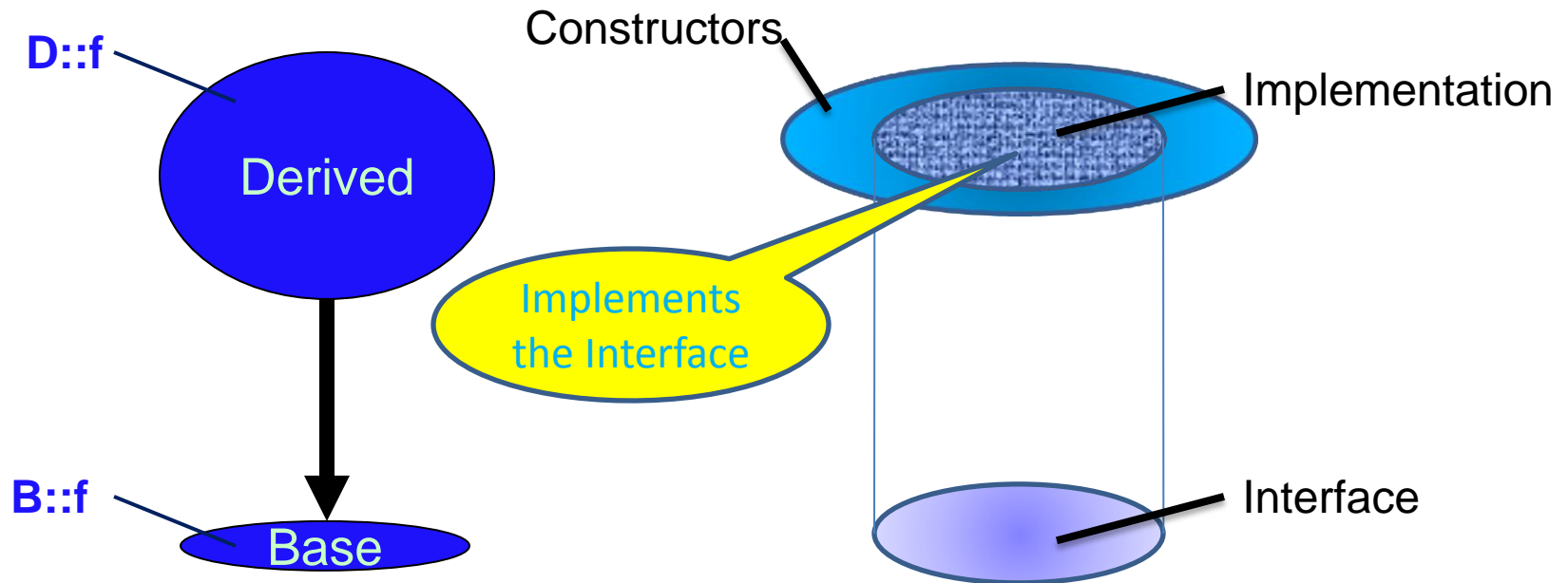
Pure Interface Inheritance



For each function **D::f** in the derived class overriding a virtual one **B::f** in the base class, the (documented) *preconditions* of **D::f** must be no stronger than those for **B::f**, and the *postconditions* no weaker.

4. Proper Inheritance

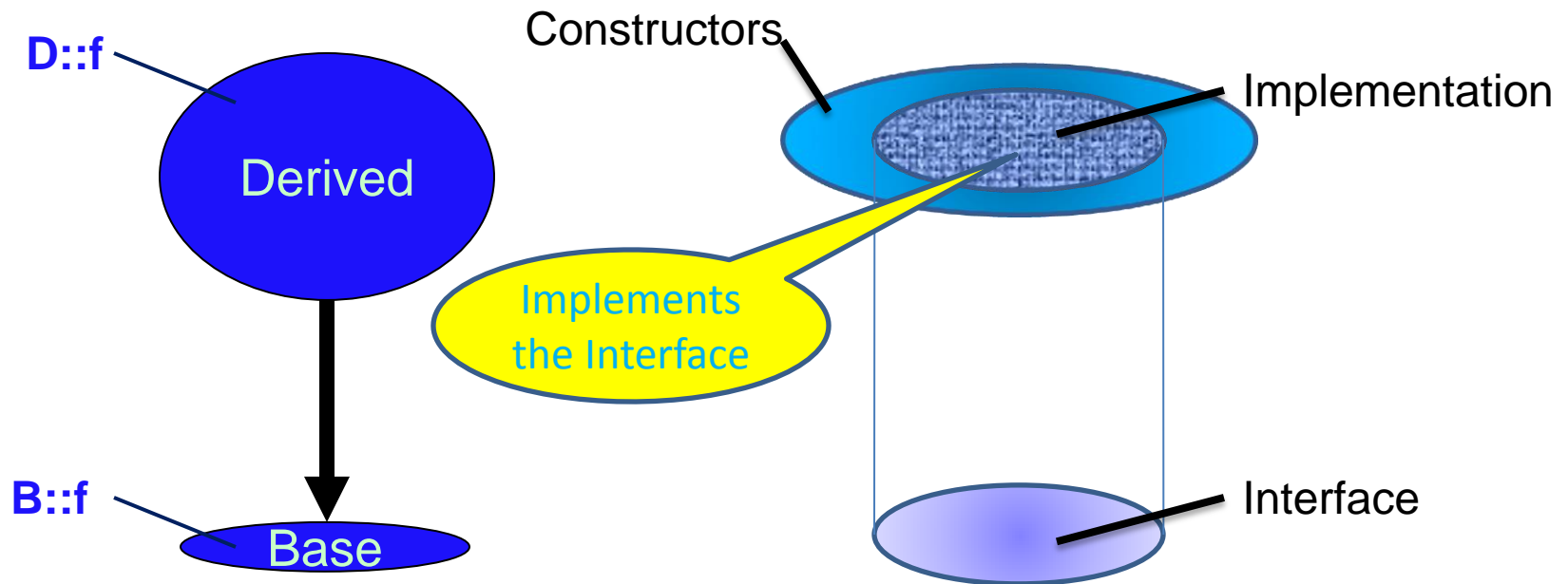
Pure Interface Inheritance



For each function **D::f** in the derived class overriding a virtual one **B::f** in the base class, the (documented) *preconditions* of **D::f** must be no stronger than those for **B::f**, and the *postconditions* no weaker.

4. Proper Inheritance

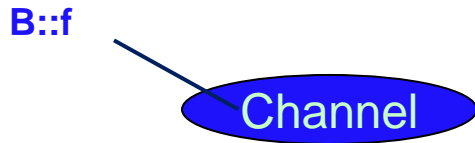
Pure Interface Inheritance



For each function **D::f** in the derived class overriding a virtual one **B::f** in the base class, the (documented) *preconditions* of **D::f** are typically the same as those for **B::f**, and the *postconditions* no weaker.

4. Proper Inheritance

Pure Interface Inheritance



4. Proper Inheritance

Pure Interface Inheritance

B::f

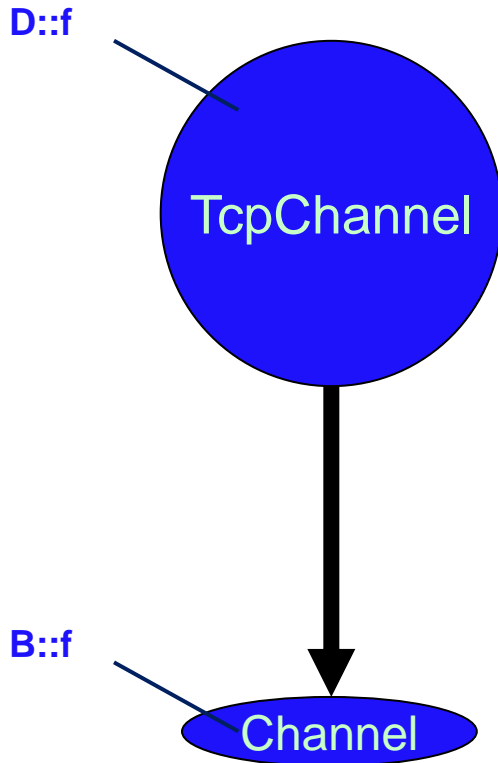


Channel

```
virtual int write(const char *buffer, int numBytes) = 0;
// Write the specified 'numBytes' from the specified
// 'buffer'. Return 0 on success, and a non-zero value
// otherwise. The behavior is undefined unless
// '0 <= numBytes <= 32767'.
```


4. Proper Inheritance

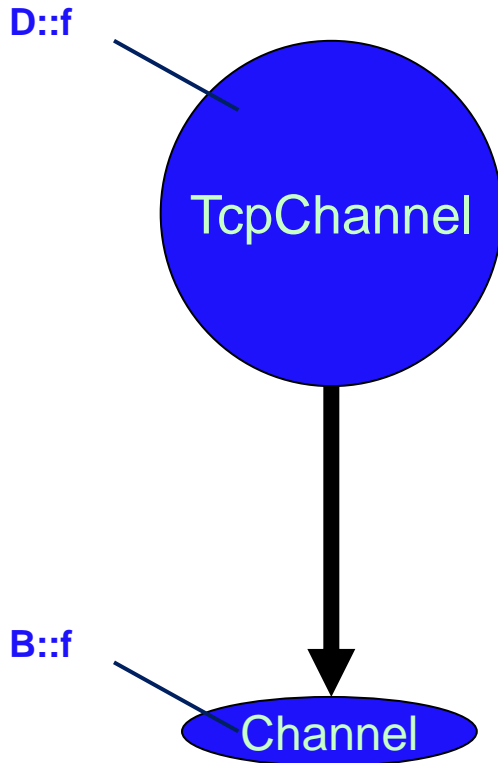
Pure Interface Inheritance



```
virtual int write(const char *buffer, int numBytes) = 0;  
    // Write the specified 'numBytes' from the specified  
    // 'buffer'. Return 0 on success, and a non-zero value  
    // otherwise. The behavior is undefined unless  
    // '0 <= numBytes <= 32767'.
```

4. Proper Inheritance

Pure Interface Inheritance

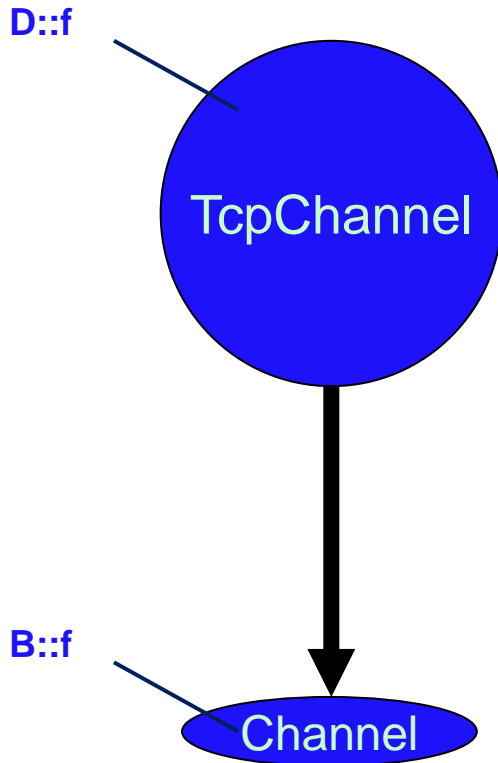


```
virtual int write(const char *buffer, int numBytes);  
    // Write to this TCP/IP channel the specified  
    // 'numBytes' from the specified 'buffer'. Return 0 on  
    // success, and a non-zero value otherwise. The  
    // behavior is undefined unless '0 == numBytes % 4'.
```

```
virtual int write(const char *buffer, int numBytes) = 0;  
    // Write the specified 'numBytes' from the specified  
    // 'buffer'. Return 0 on success, and a non-zero value  
    // otherwise. The behavior is undefined unless  
    // '0 <= numBytes <= 32767'.
```

4. Proper Inheritance

Pure Interface Inheritance

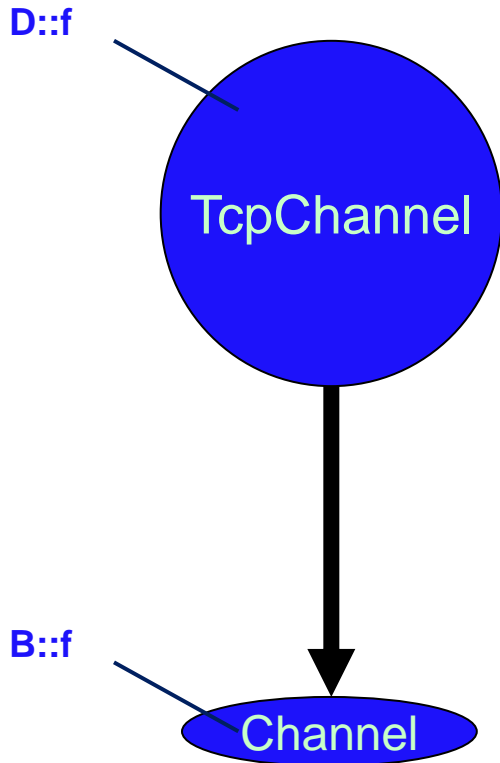


```
virtual int write(const char *buffer, int numBytes);  
    // Write to this TCP/IP channel the specified  
    // 'numBytes' from the specified 'buffer'. Return 0 on  
    // success, 1 if '0 != numBytes % 4', and a negative  
    // value otherwise.
```

```
virtual int write(const char *buffer, int numBytes) = 0;  
    // Write the specified 'numBytes' from the specified  
    // 'buffer'. Return 0 on success, and a non-zero value  
    // otherwise. The behavior is undefined unless  
    // '0 <= numBytes <= 32767'.
```

4. Proper Inheritance

Pure Interface Inheritance



```
virtual int write(const char *buffer, int numBytes);  
    // Write to this TCP/IP channel the specified  
    // 'numBytes' from the specified 'buffer'. Return 0 on  
    // success, and a non-zero value otherwise. Note that  
    // this functionality is not yet implemented on Windows;  
    // on that platform, this function always returns -1.
```

```
virtual int write(const char *buffer, int numBytes) = 0;  
    // Write the specified 'numBytes' from the specified  
    // 'buffer'. Return 0 on success, and a non-zero value  
    // otherwise. The behavior is undefined unless  
    // '0 <= numBytes <= 32767'.
```

4. Proper Inheritance

What Is a Proper Subtype/Subclass?

4. Proper Inheritance

What Is a Proper Subtype/Subclass?

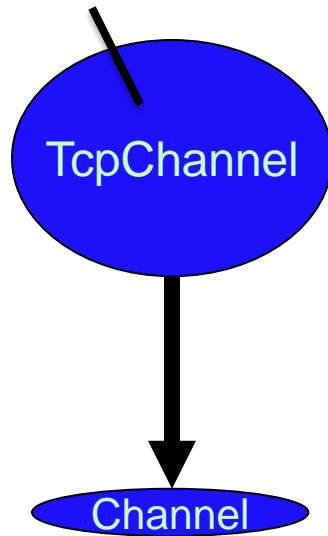
“A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a *subtype* is one whose objects provide all the behavior of objects of another type (the *supertype*) plus something extra.” – *Barbara Liskov*
(OOPSLA '87)

4. Proper Inheritance

What Is a Proper Subtype/Subclass?

“A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a *subtype* is one whose objects provide all the behavior of objects of another type (the *supertype*) plus something extra.” – *Barbara Liskov*
(OOPSLA '87)

Can **create** it.

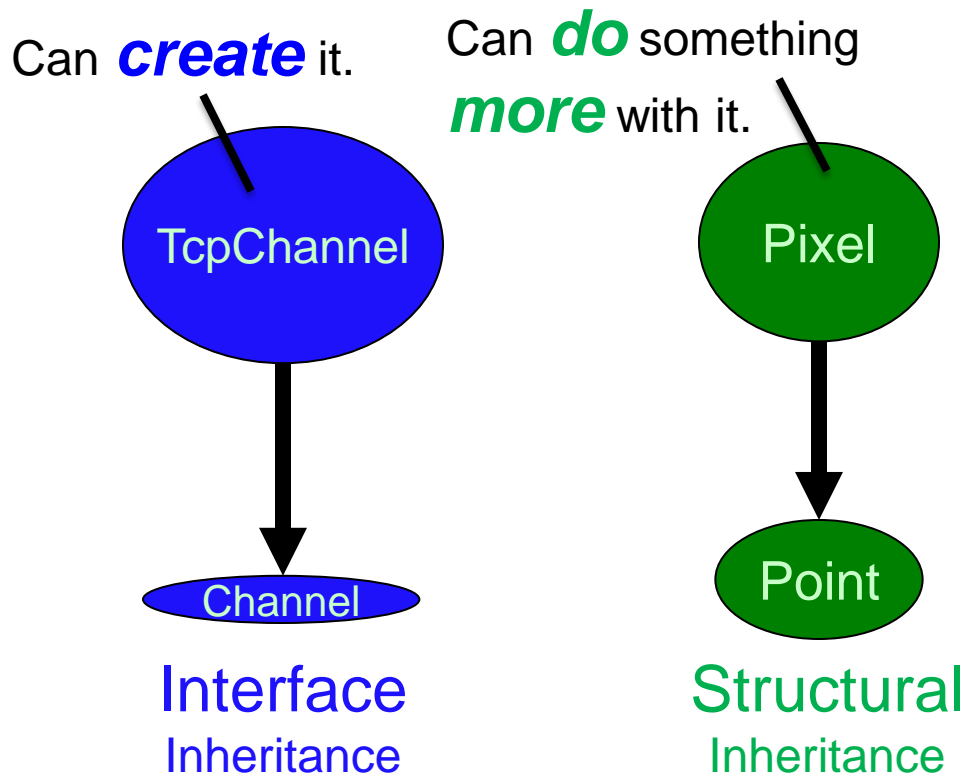


Interface
Inheritance

4. Proper Inheritance

What Is a Proper Subtype/Subclass?

“A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a **subtype** is one whose objects provide all the behavior of objects of another type (the **supertype**) plus something extra.” – *Barbara Liskov*
(OOPSLA '87)

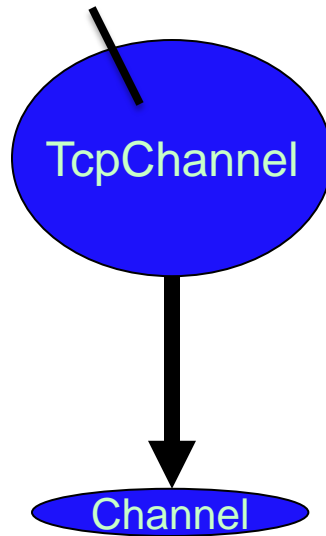


4. Proper Inheritance

What Is a Proper Subtype/Subclass?

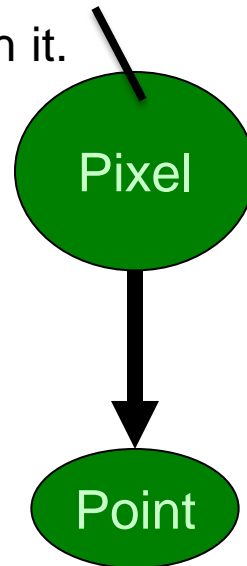
“A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a *subtype* is one whose objects provide all the behavior of objects of another type (the *supertype*) plus something extra.” – *Barbara Liskov* (OOPSLA '87)

Can **create** it.



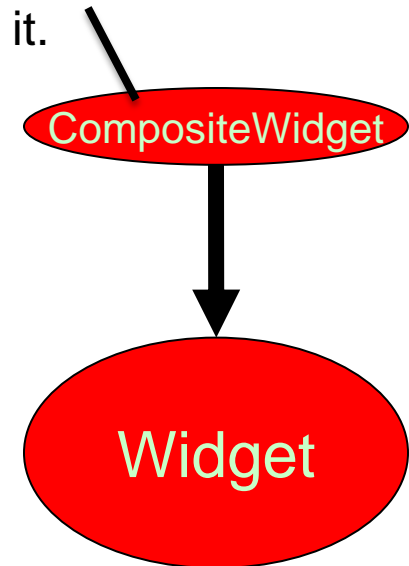
Interface
Inheritance

Can **do** something
more with it.



Structural
Inheritance

Can **create** something
else with it.



Implementation
Inheritance

4. Proper Inheritance

What Is Liskov Substitution?

4. Proper Inheritance

What Is Liskov Substitution?

What exactly is the *Liskov Substitution Principle* (**LSP**)?

4. Proper Inheritance

What Is Liskov Substitution?

What exactly is the *Liskov Substitution Principle* (**LSP**)?

- What motivated **LSP** in the first place?

4. Proper Inheritance

What Is Liskov Substitution?

What exactly is the *Liskov Substitution Principle* (**LSP**)?

- What motivated **LSP** in the first place?
- (How?) Does **LSP** relate to inheritance in C++?

4. Proper Inheritance

What Is Liskov Substitution?

What exactly is the *Liskov Substitution Principle (LSP)*?

- What motivated **LSP** in the first place?
- (How?) Does **LSP** relate to inheritance in C++?
- After *Liskov* substitution is applied, can (observable) behavior be (subtly) different?

4. Proper Inheritance

What Is Liskov Substitution?

What exactly is the *Liskov Substitution Principle (LSP)*?

- What motivated **LSP** in the first place?
- (How?) Does **LSP** relate to inheritance in C++?
- After *Liskov* substitution is applied, can (observable) behavior be (subtly) different?
- Does **LSP** apply to all *three* kinds of inheritance?

4. Proper Inheritance

What Is Liskov Substitution?

What exactly is the *Liskov Substitution Principle (LSP)*?

- What motivated **LSP** in the first place?
- (How?) Does **LSP** relate to inheritance in C++?
- After *Liskov* substitution is applied, can (observable) behavior be (subtly) different?
- Does **LSP** apply to all *three* kinds of inheritance?
- Does **LSP** have any other practical applications?

4. Proper Inheritance

What Is Liskov Substitution?

What exactly is the *Liskov Substitution Principle (LSP)*?

- What motivated **LSP** in the first place?
- (How?) Does **LSP** relate to inheritance in C++?
- After *Liskov* substitution is applied, can (observable) behavior be (subtly) different?
- Does **LSP** apply to all *three* kinds of inheritance?
- Does **LSP** have any other practical applications?
- Let's have a look...

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T.” – *Barbara Liskov* (OOPSLA '87)

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T.” – *Barbara Liskov* (OOPSLA '87)

Type Bool

Subtype Fool

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T.” – *Barbara Liskov* (OOPSLA '87)

```
main()
{
    Fool f0(false);
    Fool f1(true);
    // ...
    p(f1, f0, ...);
}
```

Type Bool

Subtype Fool

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

```
class Fool : public Bool {  
    public:  
};    Fool(int x) : Bool(!x) { }
```

```
main()  
{  
    Fool f0(false);  
    Fool f1(true);  
    // ...  
    p(f1, f0, ...);  
}
```

Type Bool

Subtype Fool

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

```
class Fool : public Bool {  
    public:  
};    Fool(int x) : Bool(!x) { }
```

```
main()  
{  
    Bool b0(false);  
    Bool b1(true);  
    // ...  
    p(b0, b1, ...);  
}
```

Type Bool

```
main()  
{  
    Fool f0(false);  
    Fool f1(true);  
    // ...  
    p(f1, f0, ...);  
}
```

Subtype Fool

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

```
class Bool {
    bool d_v;
public:
    Bool(int x) : d_v(x) { }
    operator bool() const {return d_v;}
};
```

```
class Fool : public Bool {
public:
};    Fool(int x) : Bool(!x) { }
```

```
main()
{
    Bool b0(false);
    Bool b1(true);
    // ...
    p(b0, b1, ...);
}
```

Type Bool

```
main()
{
    Fool f0(false);
    Fool f1(true);
    // ...
    p(f1, f0, ...);
}
```

Subtype Fool

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

```
class Bool {
    bool d_v;
public:
    Bool(int x) : d_v(x) { }
    operator bool() const {return d_v;}
};
```

```
class Fool : public Bool {
public:
}; Fool(int x) : Bool(!x) { }
```

```
void p(const Bool& x, const Bool& y, ...) { /* ... */ }
```

```
main()
{
    Bool b0(false);
    Bool b1(true);
    // ...
    p(b0, b1, ...);
}
```

Type Bool

```
main()
{
    Fool f0(false);
    Fool f1(true);
    // ...
    p(f1, f0, ...);
}
```

Subtype Fool

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

```
class Bool {
    bool d_v;
public:
    Bool(int x) : d_v(x) { }
    operator bool() const {return d_v;}
};
```

```
class Fool : public Bool {
public:
}; Fool(int x) : Bool(!x) { }
```

```
void p(const Bool& x, const Bool& y, ...) { /* ... */ }
```

```
main()
{
    Bool b0(false);
    Bool b1(true);
    // ...
    p(b0, b1, ...);
}
```

Note order
is different!

```
main()
{
    Fool f0(false);
    Fool f1(true);
    // ...
    p(f1, f0, ...);
}
```

Type Bool

Subtype Fool

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

(by this definition)

Fool is a
“subtype” of Bool
and vice versa!

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

```
class Bool {
    bool d_v;
public:
    Bool(int x) : d_v(x) { }
    operator bool() const {return d_v;}
};
```

```
class Fool : public Bool {
public:
}; Fool(int x) : Bool(!x) { }
```

```
void p(const Bool& x, const Bool& y, ...) { /* ... */ }
```

```
main()
{
    Bool b0(false);
    Bool b1(true);
    // ...
    p(b0, b1, ...);
}
```

Note order
is different!

```
main()
{
    Fool f0(false);
    Fool f1(true);
    // ...
    p(f1, f0, ...);
}
```

Type Bool

Subtype Fool

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T.” – *Barbara Liskov* (OOPSLA '87)

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

If, for each “derived-class” object o_1 of type S , there exists a “base-class” object o_2 of type T such that, for all programs P defined in terms of type T , the behavior of P is unchanged when the “derived-class” object o_1 is substituted for the “base-class” object o_2 , then S is a subtype of T .

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

If, *for each* “derived-class” object o_1 of type S , *there exists* a “base-class” object o_2 of type T *such that*, *for all* programs P defined in terms of type T , the behavior of P is unchanged when the “derived-class” object o_1 is substituted for the “base-class” object o_2 , then S is a subtype of T .

If, *for each* “derived-class” object d of type D , *there exists* a “base-class” object b of type B *such that*, *for all* programs P defined in terms of type B , the behavior of P is unchanged when the “derived-class” object d is substituted for the “base-class” object b , then D is a subtype of B .

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

If, for each “derived-class” object o_1 of type S , there exists a “base-class” object o_2 of type T such that, for all programs P defined in terms of type T , the behavior of P is unchanged when the “derived-class” object o_1 is substituted for the “base-class” object o_2 , then S is a subtype of T .

If, for each “derived-class” object d of type D , there exists a “base-class” object b of type B such that, for all programs P defined in terms of type B , the behavior of P is unchanged when the “derived-class” object d is substituted for the “base-class” object b , then D is a subtype of B .

If, for each object d of type D , there exists an object b of type B such that, for all programs P defined in terms of B , the behavior of P is unchanged when d is substituted for b , then D is a subtype of B .

4. Proper Inheritance

What Is Liskov Substitution?

“If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .” – *Barbara Liskov* (OOPSLA '87)

If, for each “derived-class” object o_1 of type S , there exists a “base-class” object o_2 of type T such that, for all programs P defined in terms of type T , the behavior of P is unchanged when the “derived-class” object o_1 is substituted for the “base-class” object o_2 , then S is a subtype of T .

If, for each “derived-class” object d of type D , there exists a “base-class” object b of type B such that, for all programs P defined in terms of type B , the behavior of P is unchanged when the “derived-class” object d is substituted for the “base-class” object b , then D is a subtype of B .

If, for each object d of type D , there exists an object b of type B such that, for all programs P defined in terms of B , the behavior of P is unchanged when d is substituted for b , then D is a subtype of B .

4. Proper Inheritance

What Is Liskov Substitution?

If, for each object **d** of type **D**, there exists an object **b** of type **B** such that, for all programs **P** defined in terms of **B**, the behavior of **P** is unchanged when **d** is substituted for **b**, then **D** is a subtype of **B**.

```
class Bool {
    bool d_v;
public:
    Bool(int x) : d_v(x) { }
    operator bool() const {return d_v;}
};
```

```
class Fool : public Bool {
public:
};    Fool(int x) : Bool(!x) { }
```

```
void p(const Bool& x, const Bool& y, ...) { /* ... */ }
```

```
main()
{
    Bool b0(false);
    Bool b1(true);
    // ...
    p(b0, b1, ...);
}
```

Type Bool

```
main()
{
    Fool f0(false);
    Fool f1(true);
    // ...
    p(f1, f0, ...);
}
```

Subtype Fool

4. Proper Inheritance

What Is Liskov Substitution?

If, for each object **d** of type **D**, there exists an object **b** of type **B** such that, for all programs **P** defined in terms of **B**, the behavior of **P** is unchanged when **d** is substituted for **b**, then **D** is a subtype of **B**.

```
class Bool {  
    bool d_v;  
public:  
    Bool(int x) :  
        operator bool  
};
```

```
void p(const
```

```
main()  
{  
    Bool b0  
    Bool b1  
    // ...  
    p(b0, b1  
}
```

```
public Bool {  
    x) : Bool(!x) { }
```

```
/* ... */ }
```

```
else);  
true);  
...);
```

Necessary,
but Not
Sufficient, for
Inheritance.

Type Bool

Subtype Fool

4. Proper Inheritance

What Is Liskov Substitution?

If, for each object **d** of type **D**, there exists an object **b** of type **B** such that, for all programs **P** defined in terms of **B**, the behavior of **P** is unchanged when **d** is substituted for **b**, then **D** is a subtype of **B**.

(by this definition)

Every *empty* type
is a “subtype”
of all types!

4. Proper Inheritance

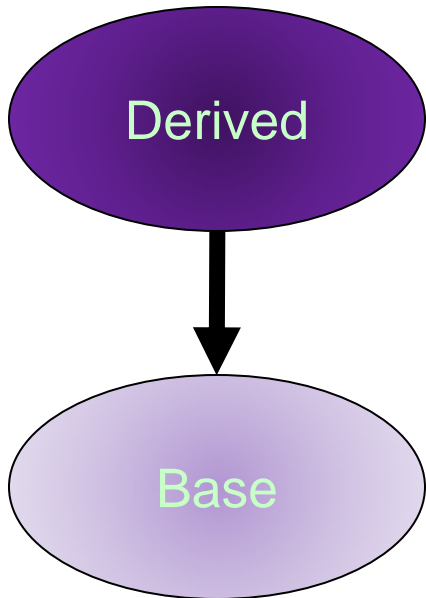
What Is Proper Inheritance?

Recall the following *general property*:

For inheritance to be *proper*, any operation that can be invoked on a derived-class *object* via a base-class pointer (or reference) must behave identically if we replace that base-class pointer (or reference) with a corresponding derived-class one.

4. Proper Inheritance

What Is Proper Inheritance?



```
void example(Derived *pDerived)
{
    Base *pBase = pDerived;

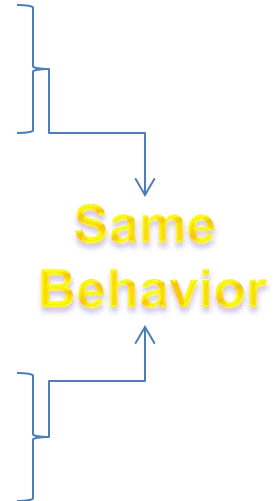
#ifdef USE_DERIVED_CLASS_INTERFACE

    pDerived->someMethod(/* ... */);
    int result = someFunction(*pDerived);

#else

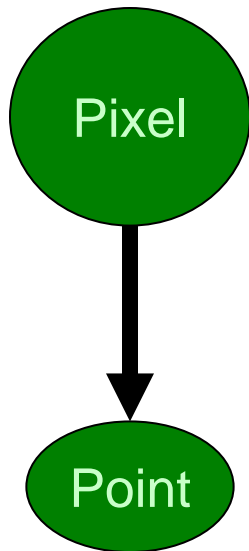
    pBase->someMethod(/* ... */);
    int result = someFunction(*pBase);

#endif
}
```



4. Proper Inheritance

Pure Structural Inheritance



```
#ifdef USE_BASE_CLASS_INTERFACE
    typedef Point Type;
#else
    typedef Pixel Type;
#endif
```

```
void anyProgram(Type *p);
```

```
void main()
```

```
{
```

```
    Pixel pixel(1, 2, Pixel::BLUE);
```

```
    anyProgram(&pixel);
```

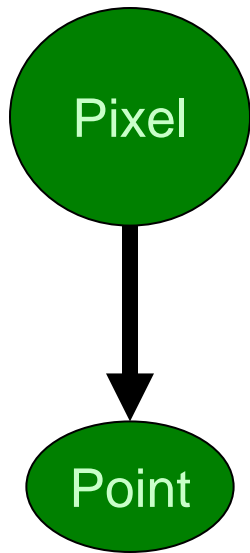
```
}
```

```
using std::cout; // (We do this only
using std::endl; // in test drivers.)
```

4. Proper Inheritance

Pure Structural Inheritance

```
void anyProgram(Type *p)
{
    cout << p->x() << endl;
}
```

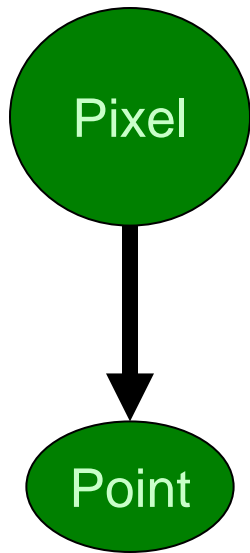


```
int Point::x() const
{
    return d_x;
}
```


4. Proper Inheritance

Pure Structural Inheritance

```
void anyProgram(Type *p)
{
    p->setY(10);
}
```



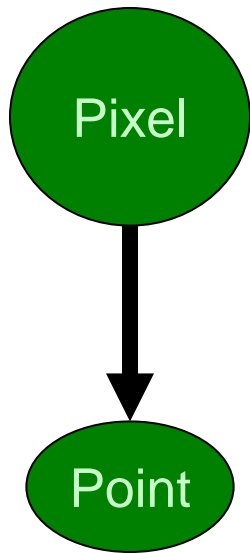
```
void Point::setY(int y)
{
    d_y = y;
}
```

4. Proper Inheritance

Pure Structural Inheritance

```
class Pixel : public Point {  
    // ...  
    Color d_color;  
    // ...  
};
```

```
void anyProgram(Type *p)  
{  
    cout << p->color() << endl;  
}
```

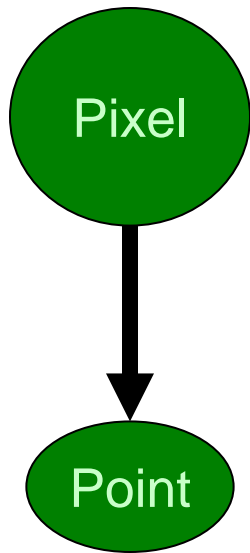


```
Pixel::Color Pixel::color() const  
{  
    return d_color;  
}
```

4. Proper Inheritance

Pure Structural Inheritance

```
void anyProgram(Type *p)
{
    p->setY(10);
}
```



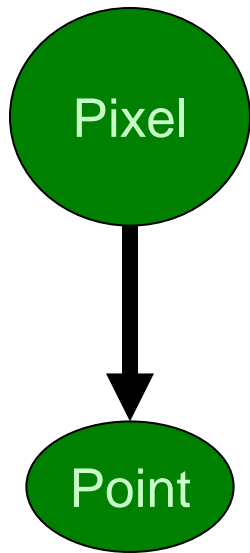
```
void Pixel::setY(int y)
{
    cout << "Pixel::setY(int y)" << endl;
    d_y = y;
}
```

```
void Point::setY(int y)
{
    d_y = y;
}
```

4. Proper Inheritance

Pure Structural Inheritance

```
void anyProgram(Type *p)
{
    p->setY(10);
}
```



```
void Pixel::setY(int y)
{
    cout << "Pixel::setY(int y)" << endl;
    d_y = y;
}
```

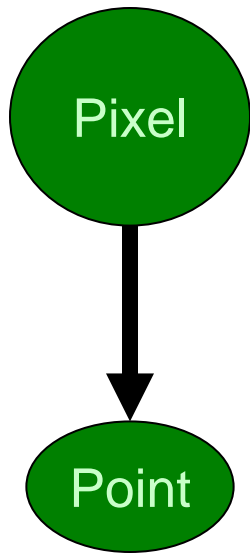
```
void Point::setY(int y)
{
    d_y = y;
}
```

4. Proper Inheritance

Pure Structural Inheritance

```
class Pixel : public Point {  
    // ...  
    static int s_numSetY;  
public:  
    // ...  
};
```

```
void anyProgram(Type *p)  
{  
    p->setY(10);  
}
```



```
void Pixel::setY(int y)  
{  
    ++s_numSetY; // Pixel class data  
    d_y = y;  
}
```

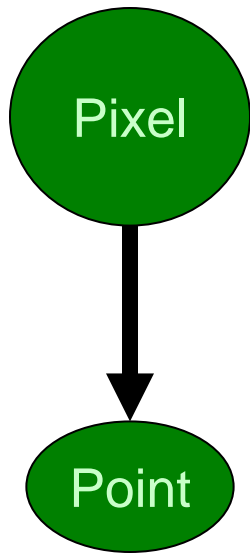
```
void Point::setY(int y)  
{  
    d_y = y;  
}
```

4. Proper Inheritance

Pure Structural Inheritance

```
class Pixel : public Point {  
    // ...  
    static int s_numSetY;  
public:  
    // ...  
};
```

```
void anyProgram(Type *p)  
{  
    p->setY(10);  
    cout << p->numSetY() << endl;  
}
```

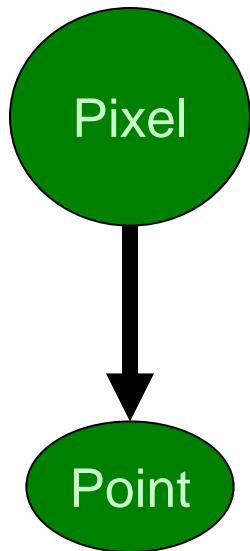


```
void Pixel::setY(int y)  
{  
    ++s_numSetY; // Pixel class data  
    d_y = y;  
}  
int Pixel::numSetY() { return s_numSetY; }  
  
void Point::setY(int y)  
{  
    d_y = y;  
}
```

4. Proper Inheritance

Pure Structural Inheritance

```
void anyProgram(Type *p)
{
    p->setY(10);
}
```



```
void Pixel::setY(int y)
```

```
// Set the y-coordinate of this object to the absolute value of the  
// specified 'y'. The behavior is undefined unless 'INT_MIN < y'.
```

```
{  
    d_y = y < 0 ? -y : y;  
}
```

```
void Point::setY(int y)
```

```
// Set the y-coordinate of this object to the specified 'y'.  
// The behavior is undefined unless '0 <= y'.
```

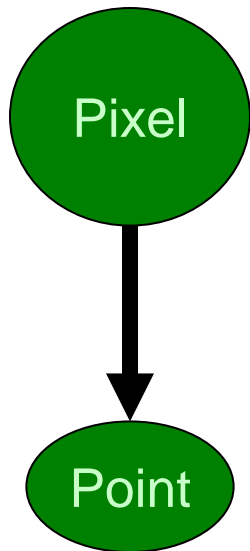
```
{  
    d_y = y;  
}
```

4. Proper Inheritance

Pure Structural Inheritance

```
class Pixel : public Point {  
    // ...  
    Color d_color;  
    // ...  
};
```

```
void anyProgram(Type *p)  
{  
    if (sizeof p->self() > sizeof(Point))  
        cout << "It's not a Point!" << endl;  
}
```



```
const Pixel& Pixel::self() const  
{  
    return *this;  
}
```

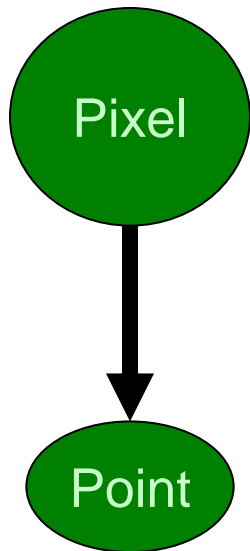
```
const Point& Point::self() const  
{  
    return *this;  
}
```


4. Proper Inheritance

Pure Structural Inheritance

```
class Pixel : public Point {  
    // ...  
    Color d_color;  
    // ...  
};
```

```
void anyProgram(Type *p)  
{  
    if (sizeof p->self() > sizeof(Point))  
        cout << "It's not a Point!" << endl;  
}
```



```
const Pixel& Pixel::self() const  
{  
    return *this;  
}
```

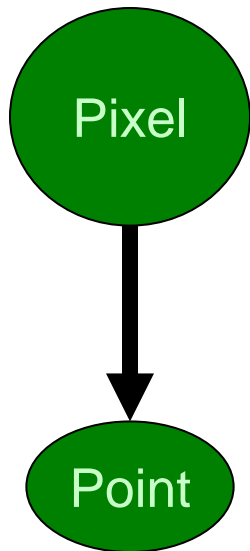
```
const Point& Point::self() const  
{  
    return *this;  
}
```

4. Proper Inheritance

Pure Structural Inheritance

```
class Pixel : public Point {  
    // ...  
    Color d_color;  
    // ...  
};
```

```
void anyProgram(Type *p)  
{  
    if (sizeof *p > sizeof(Point))  
        cout << "It's not a Point!" << endl;  
}
```

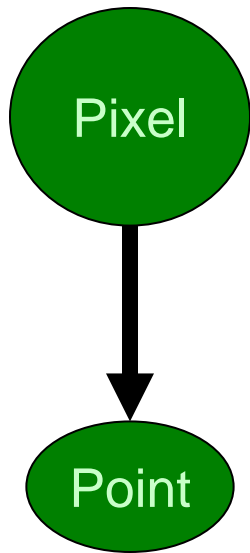


4. Proper Inheritance

Pure Structural Inheritance

```
class Pixel : public Point {  
    //  
    Color a_color;  
    //  
};
```


```
void anyProgram(Type *p)  
{  
    if (sizeof *p > sizeof(Point))  
        cout << "It's not a Point!" << endl;  
}
```



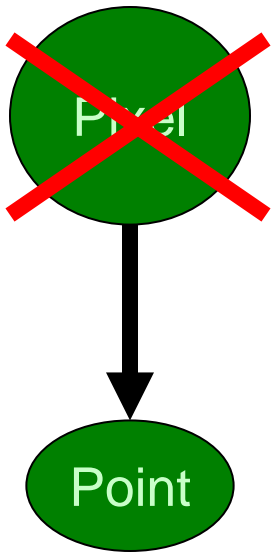
4. Proper Inheritance

Pure Structural Inheritance

```
class Pixel : public Point {  
    // ...  
    Color a_color;  
    // ...  
};
```



```
void anyProgram(Type *p)  
{  
    if (sizeof *p > sizeof(Point))  
        cout << "It's not a Point!" << endl;  
}
```

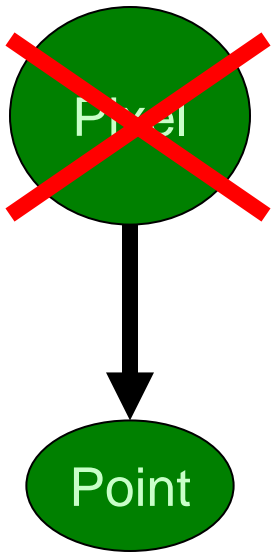


4. Proper Inheritance

Pure Structural Inheritance

```
class Pixel : public Point {  
    //  
    Color a_color;  
    //  
};
```

```
void anyProgram(Type *p)  
{  
    if (sizeof *p > sizeof(Point))  
        cout << "It's not a Point!" << endl;  
}
```



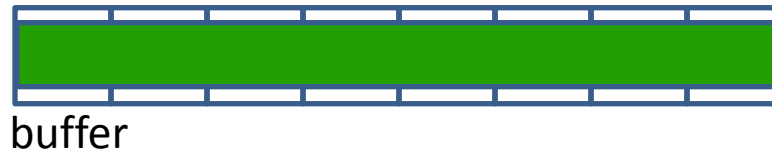
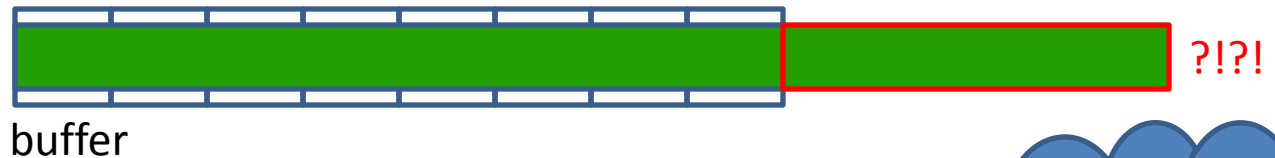
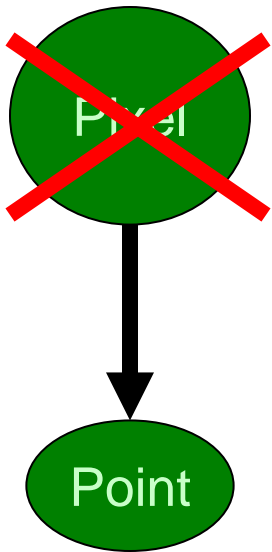
Proper **Structural**
Inheritance *extends*
functionality, but
does **not** extend the
object's footprint.

4. Proper Inheritance

Pure Structural Inheritance

```
class Pixel : public Point {  
    // ...  
    Color a_color;  
    // ...  
};
```

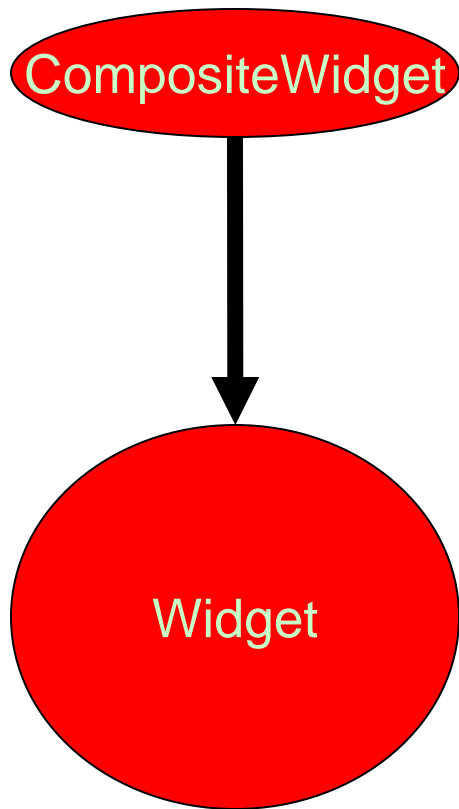
```
void anyProgram(Type *p)  
{  
    double alignmentHack; // Don't do it!  
    char buffer[sizeof(Point)];  
    (Type *)&buffer = *p;  
}
```



The same “*size*” issue applies to arrays of objects!

4. Proper Inheritance

Implementation Inheritance



Implementation hierarchies are *highly problematic!*

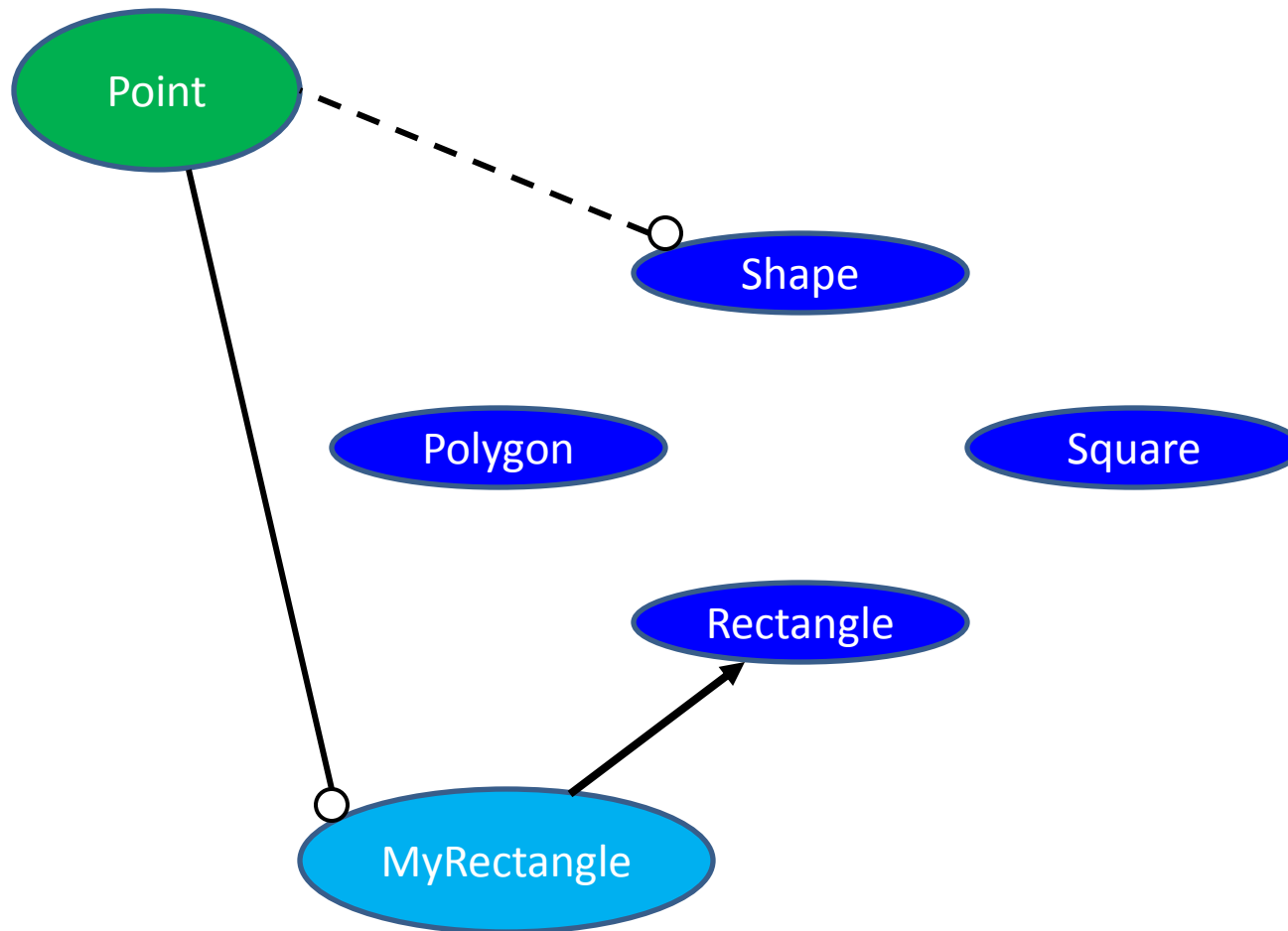
Incorporating implementation with interface inheritance:

- Makes software brittle, inflexible, and hard to maintain.
- Exposes public clients to physical (compile- and link-time) dependencies on the shared implementation.
- Adds nothing that cannot be done with pure interface inheritance and layering.

Its only value is as a syntactic expedient!

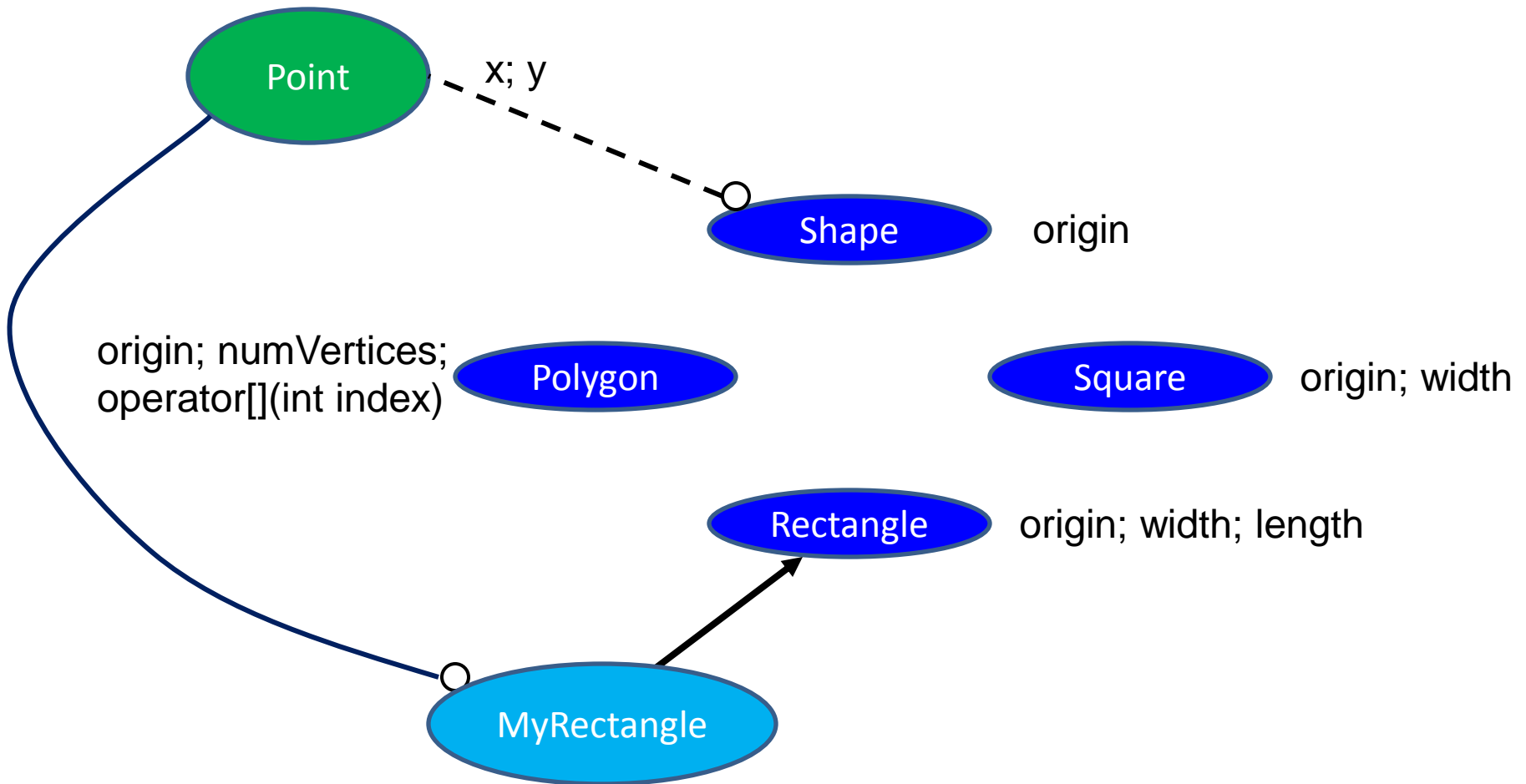
4. Proper Inheritance

Using Interface Inheritance Effectively



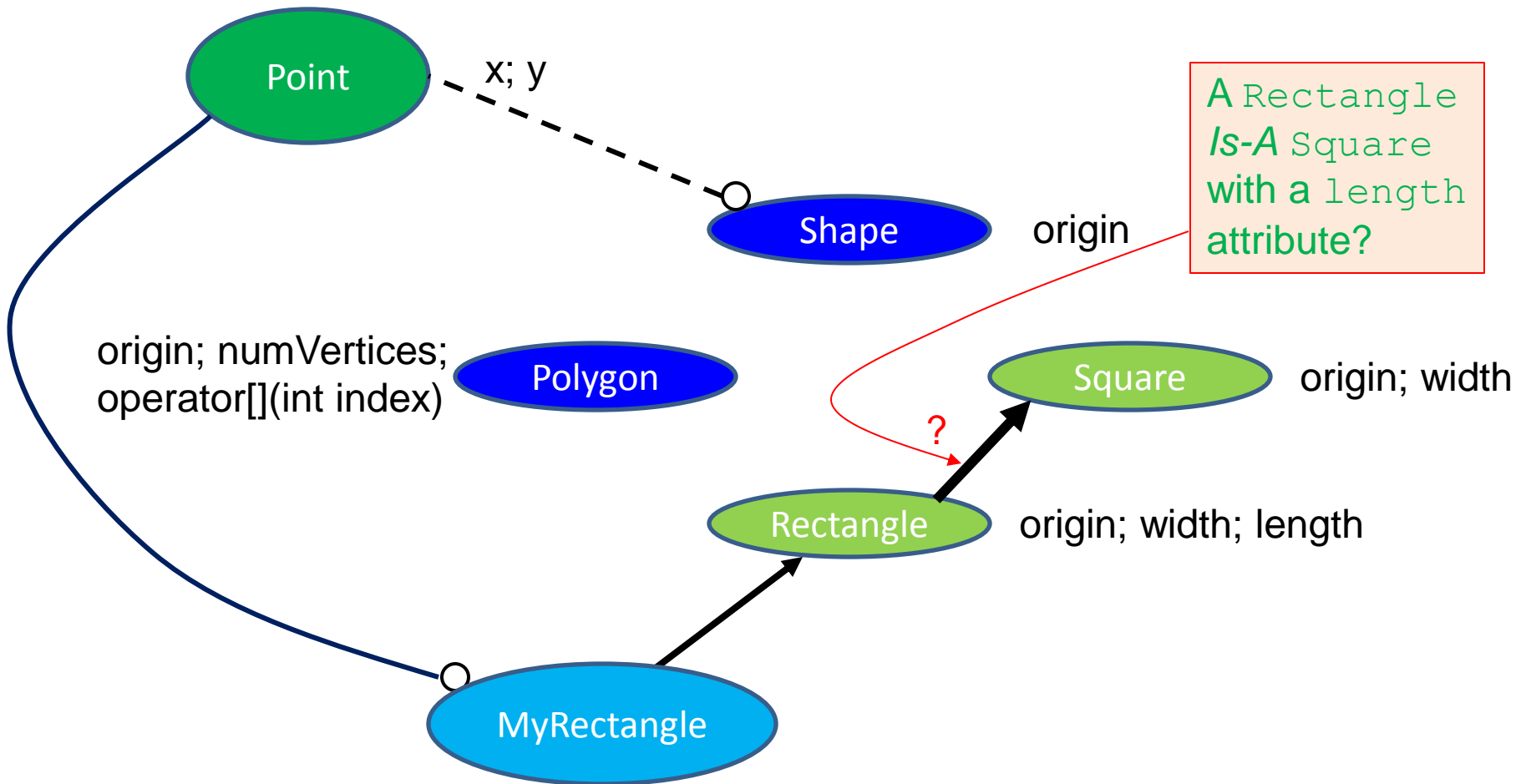
4. Proper Inheritance

Using Interface Inheritance Effectively



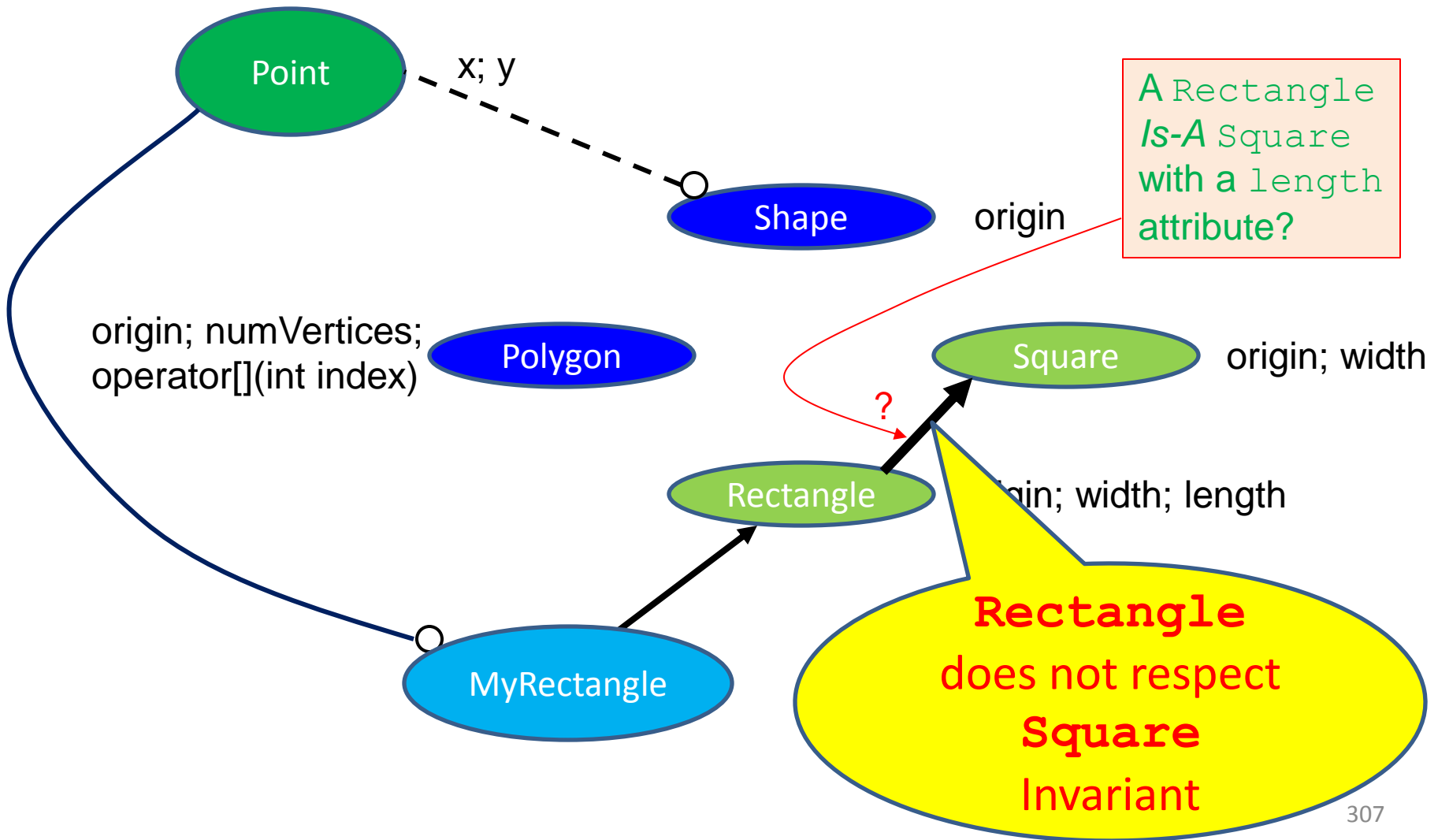
4. Proper Inheritance

Using Interface Inheritance Effectively



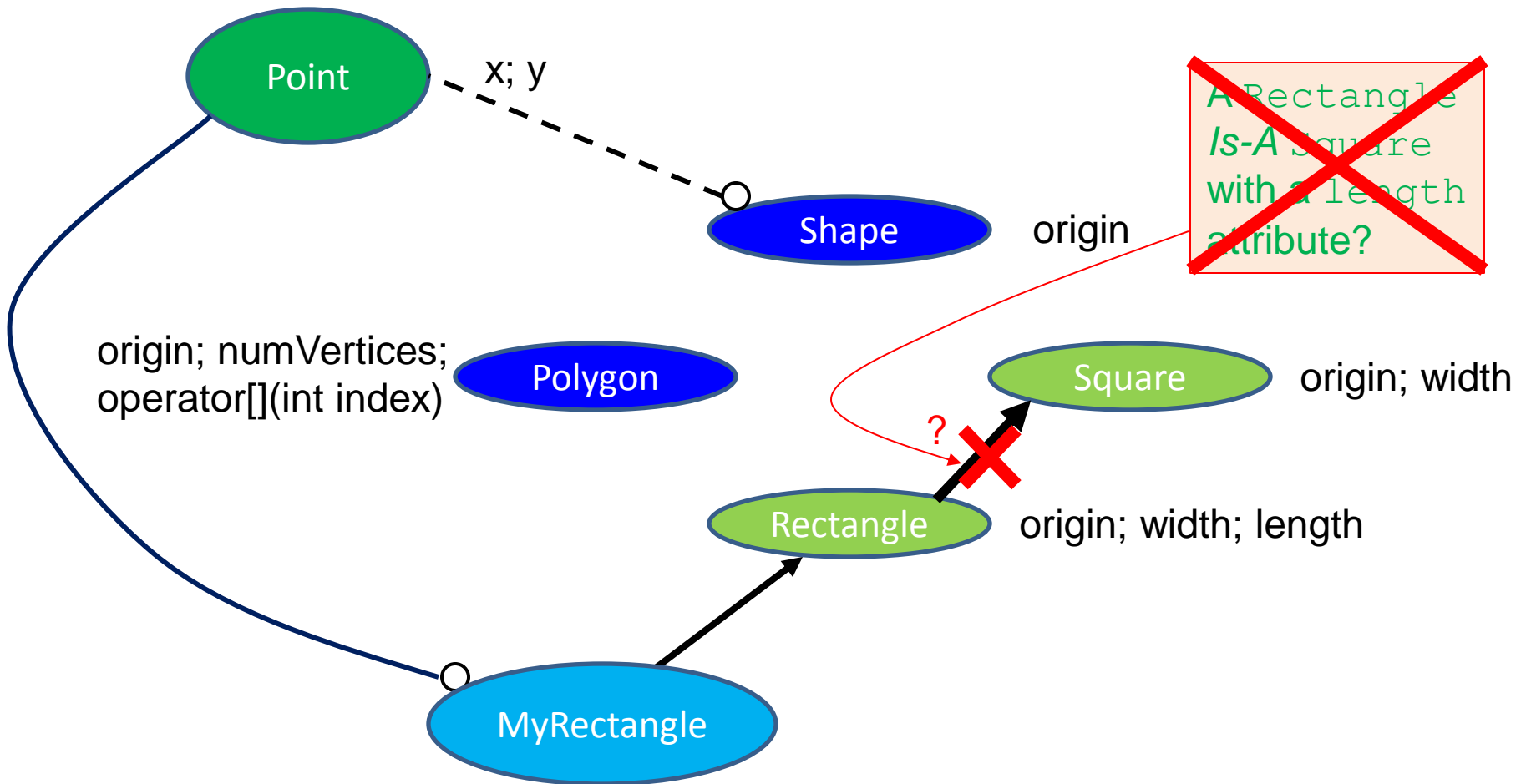
4. Proper Inheritance

Using Interface Inheritance Effectively



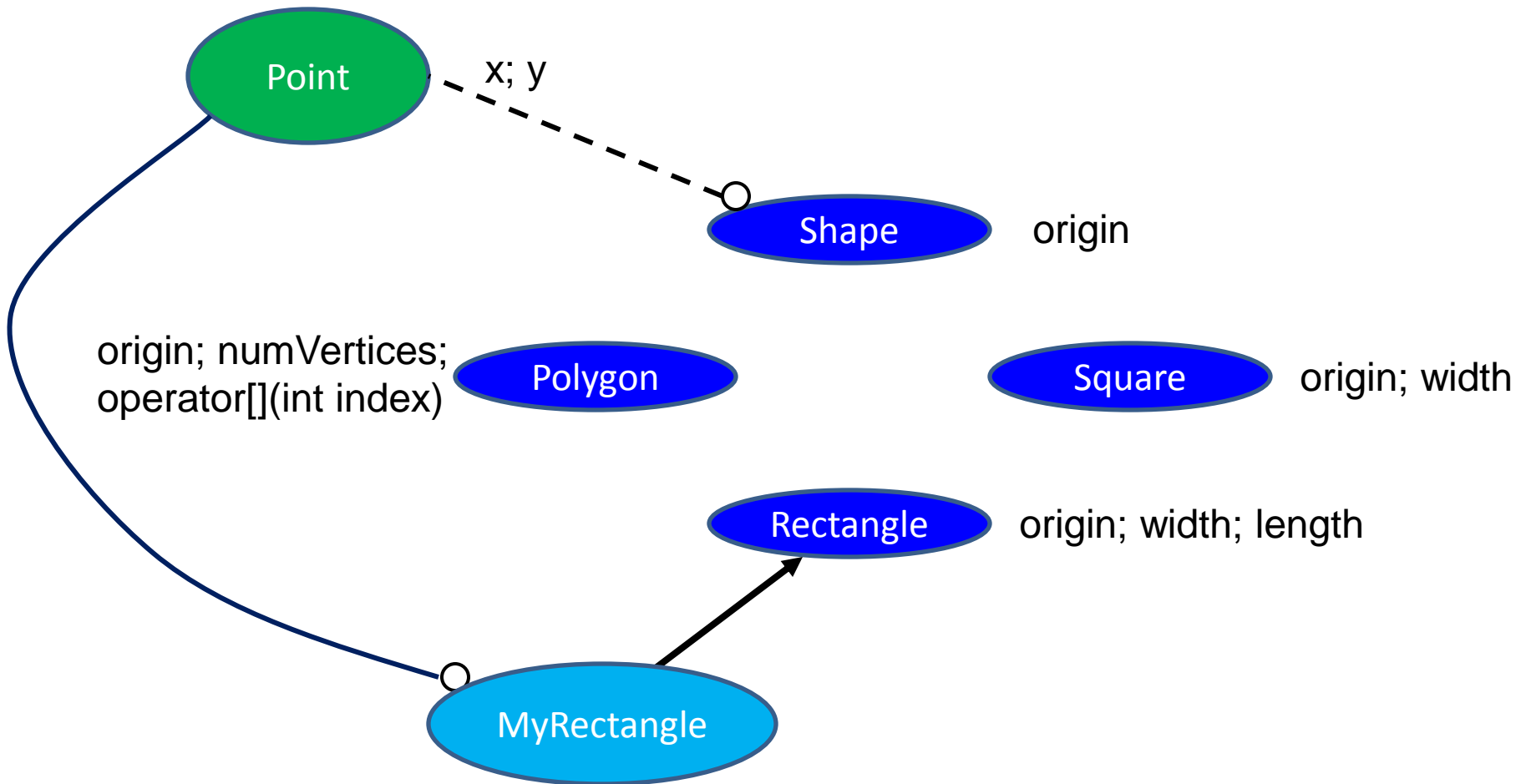
4. Proper Inheritance

Using Interface Inheritance Effectively



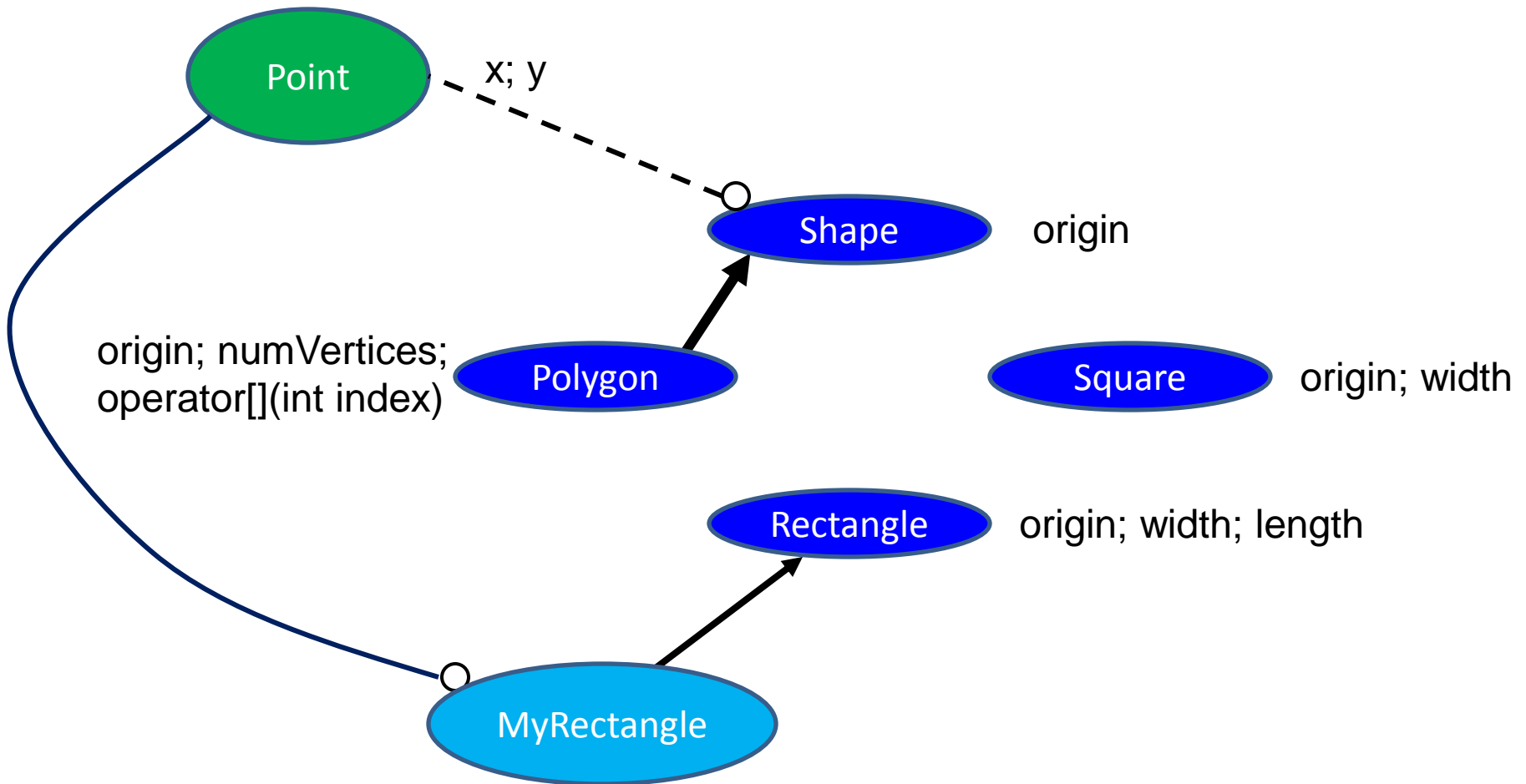
4. Proper Inheritance

Using Interface Inheritance Effectively



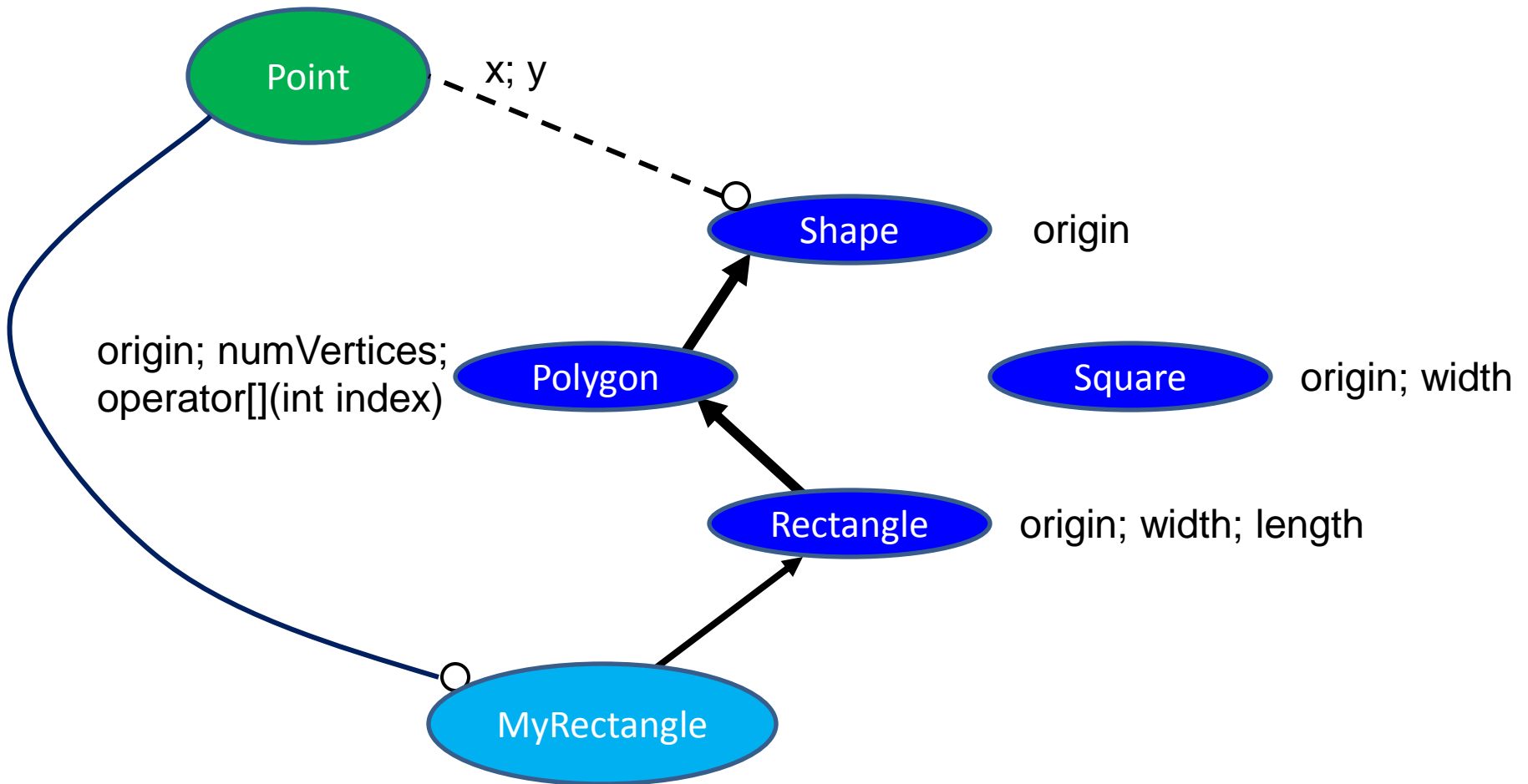
4. Proper Inheritance

Using Interface Inheritance Effectively



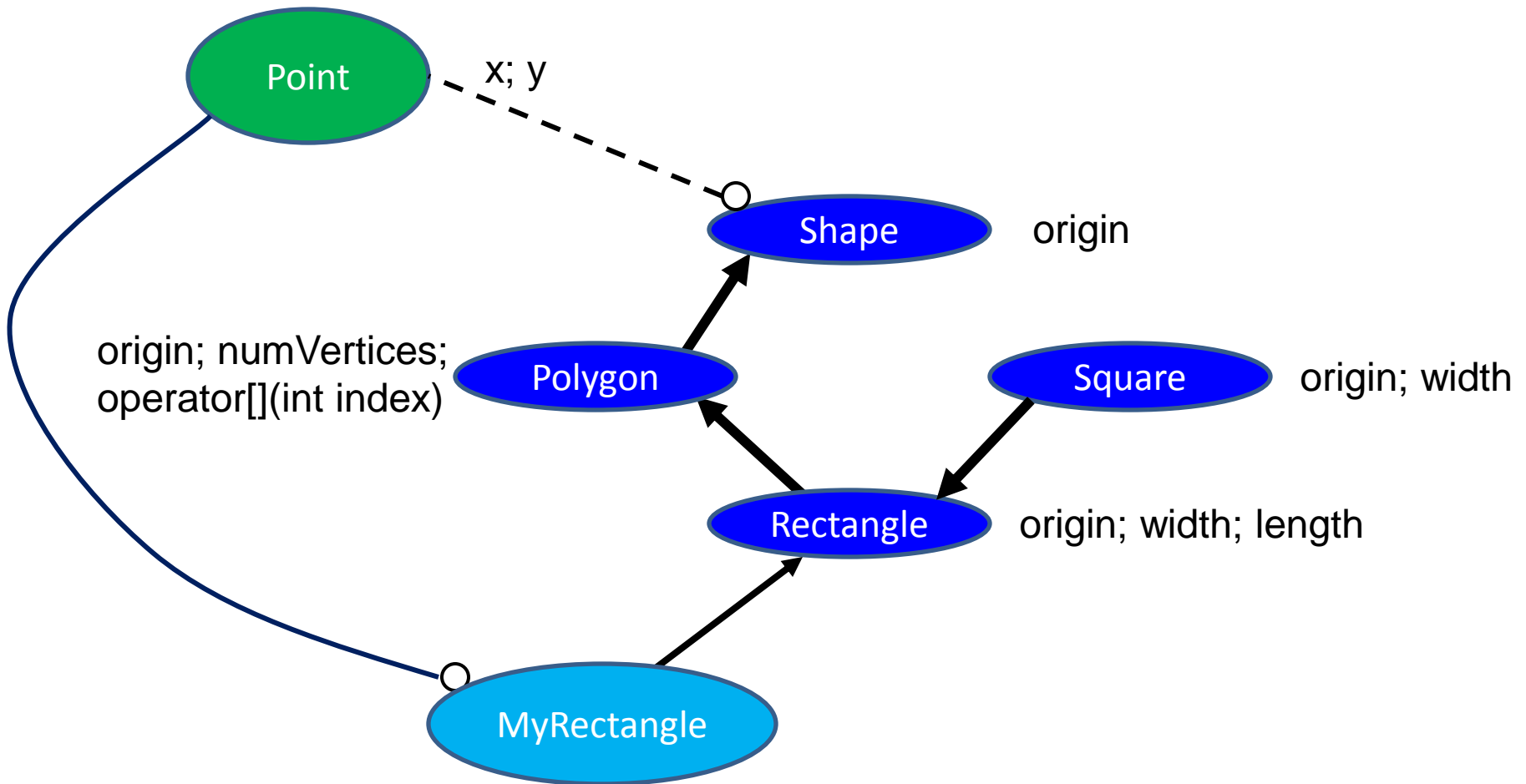
4. Proper Inheritance

Using Interface Inheritance Effectively



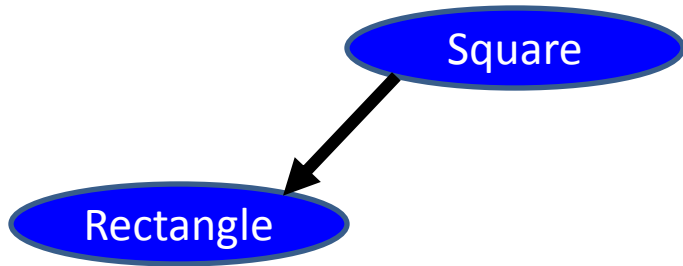
4. Proper Inheritance

Using Interface Inheritance Effectively



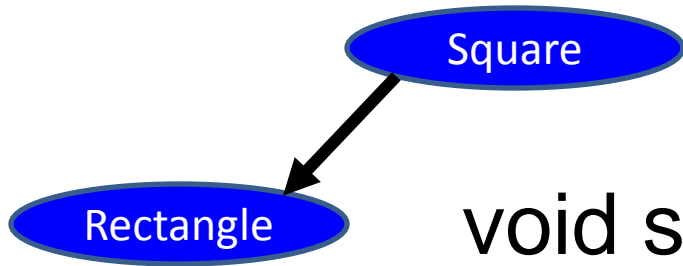
4. Proper Inheritance

Using Interface Inheritance Effectively



4. Proper Inheritance

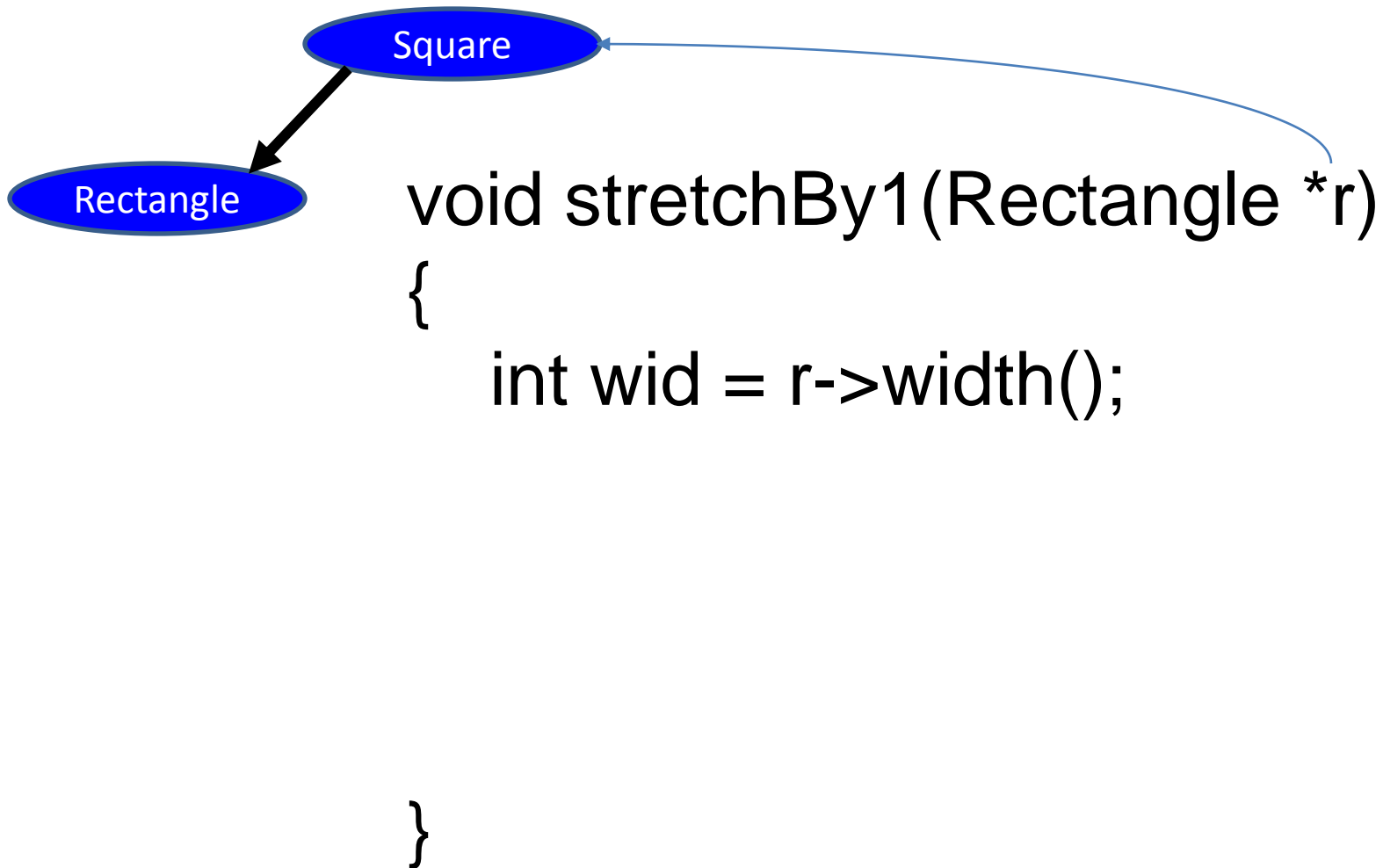
Using Interface Inheritance Effectively



```
void stretchBy1(Rectangle *r)
{
}
}
```

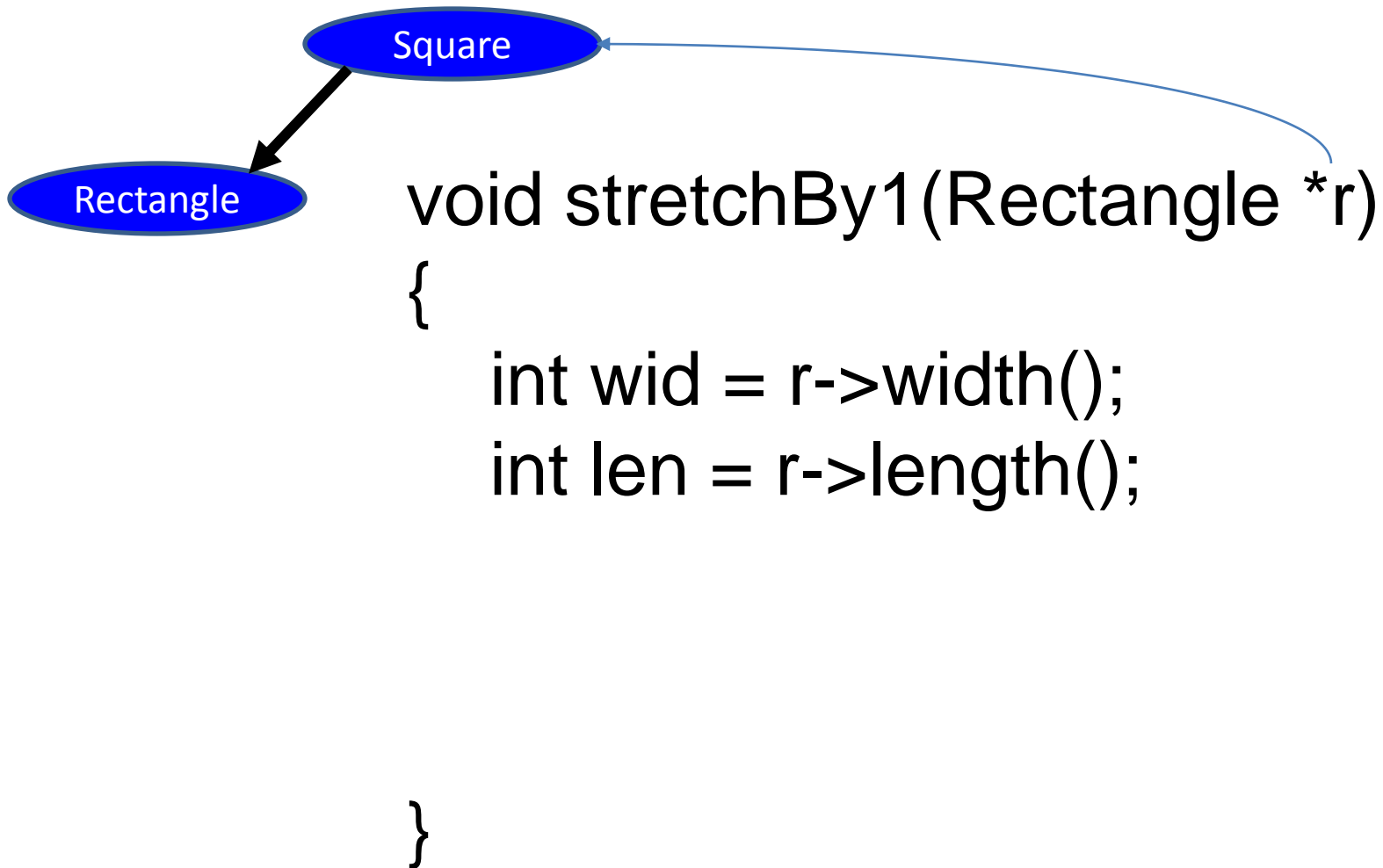

4. Proper Inheritance

Using Interface Inheritance Effectively



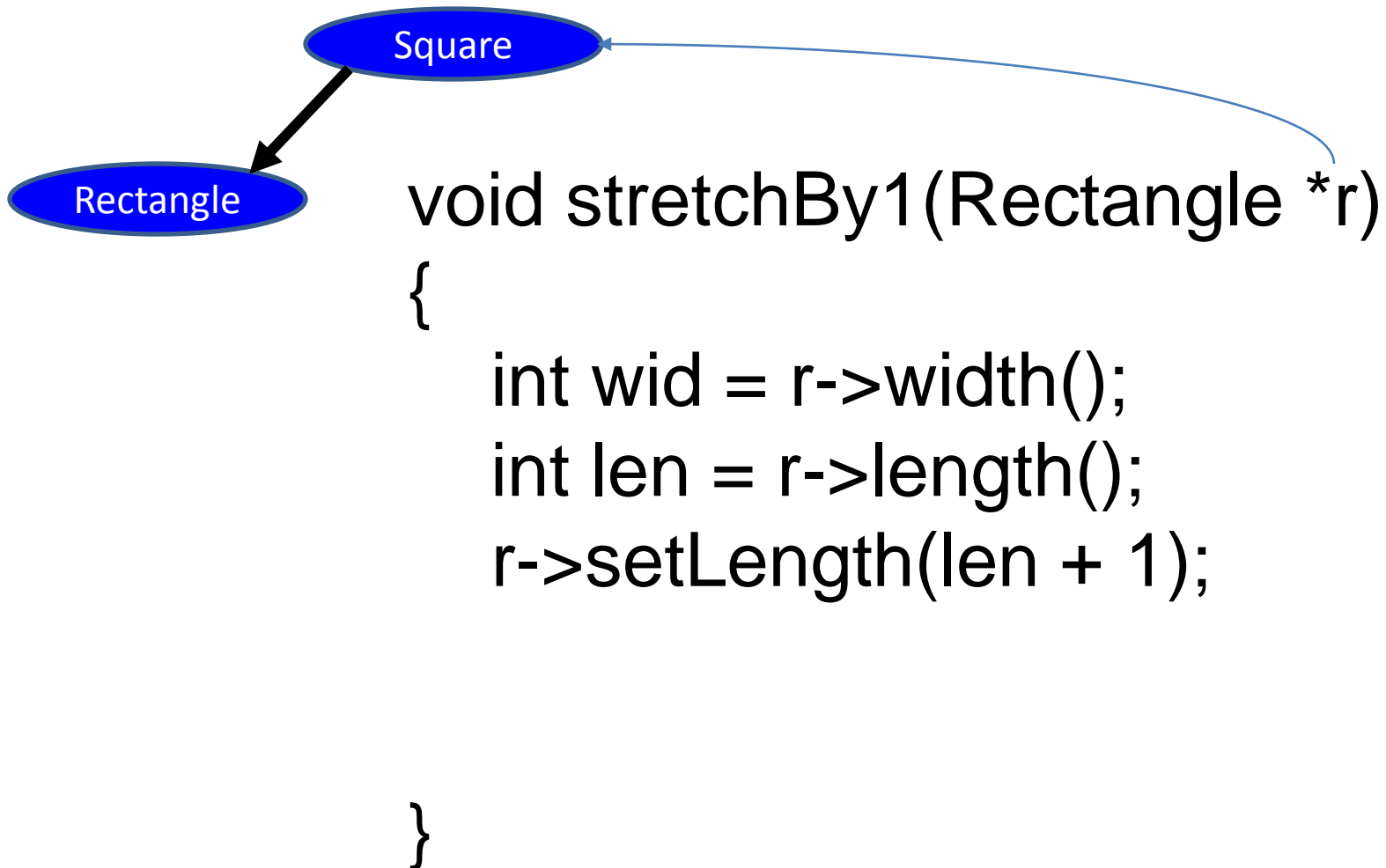
4. Proper Inheritance

Using Interface Inheritance Effectively



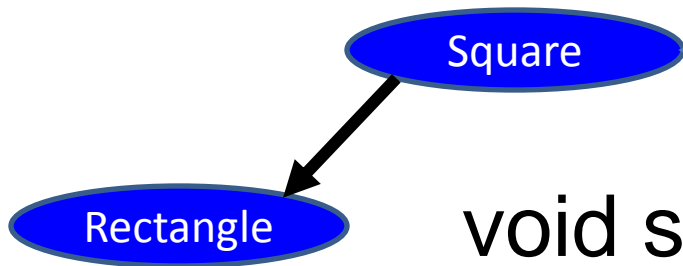
4. Proper Inheritance

Using Interface Inheritance Effectively



4. Proper Inheritance

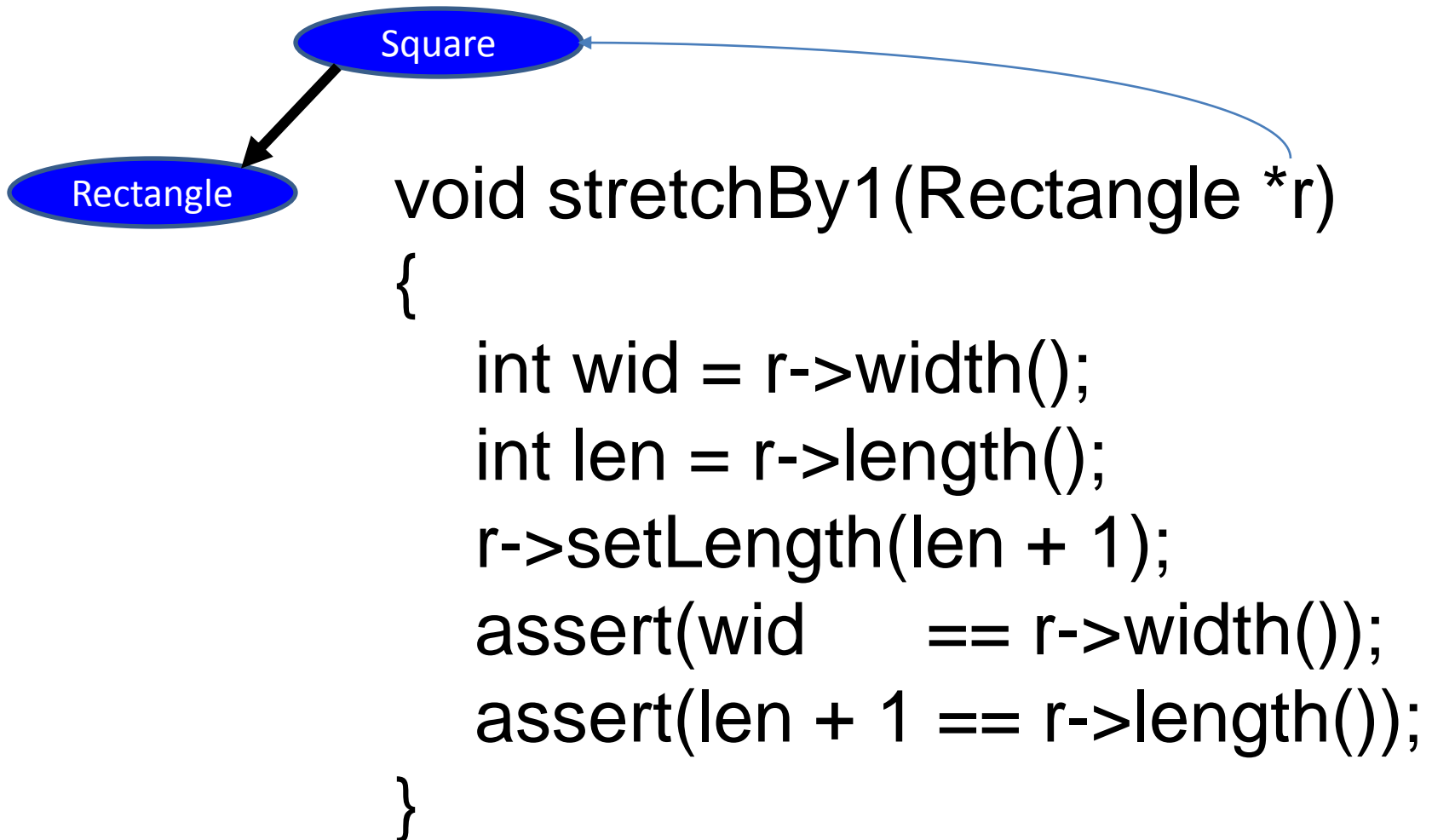
Using Interface Inheritance Effectively



```
void stretchBy1(Rectangle *r)
{
    int wid = r->width();
    int len = r->length();
    r->setLength(len + 1);
    assert(wid == r->width());
}
```

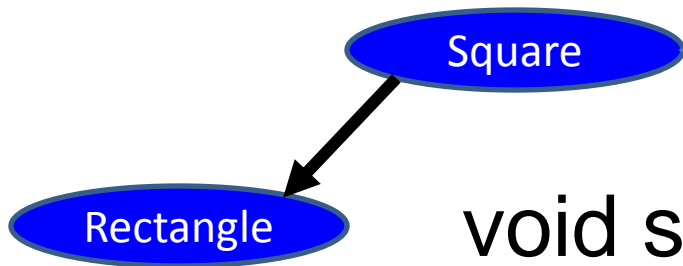
4. Proper Inheritance

Using Interface Inheritance Effectively



4. Proper Inheritance

Using Interface Inheritance Effectively



```
void stretchBy1(Rectangle *r)
```

```
{
```

```
    int wid = r->width();
```

```
    int len = r->length();
```

```
    r->setLength(len + 1);
```

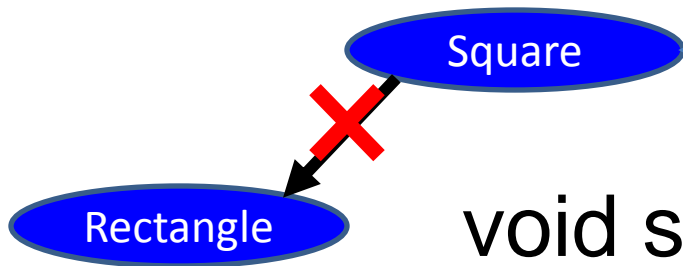
```
    { assert(wid == r->width());  
      assert(len + 1 == r->length());
```

```
}
```

Either **Assert**
or **No Longer**
Square

4. Proper Inheritance

Using Interface Inheritance Effectively



```
void stretchBy1(Rectangle *r)
```

```
{
```

```
    int wid = r->width();
```

```
    int len = r->length();
```

```
    r->setLength(len + 1);
```

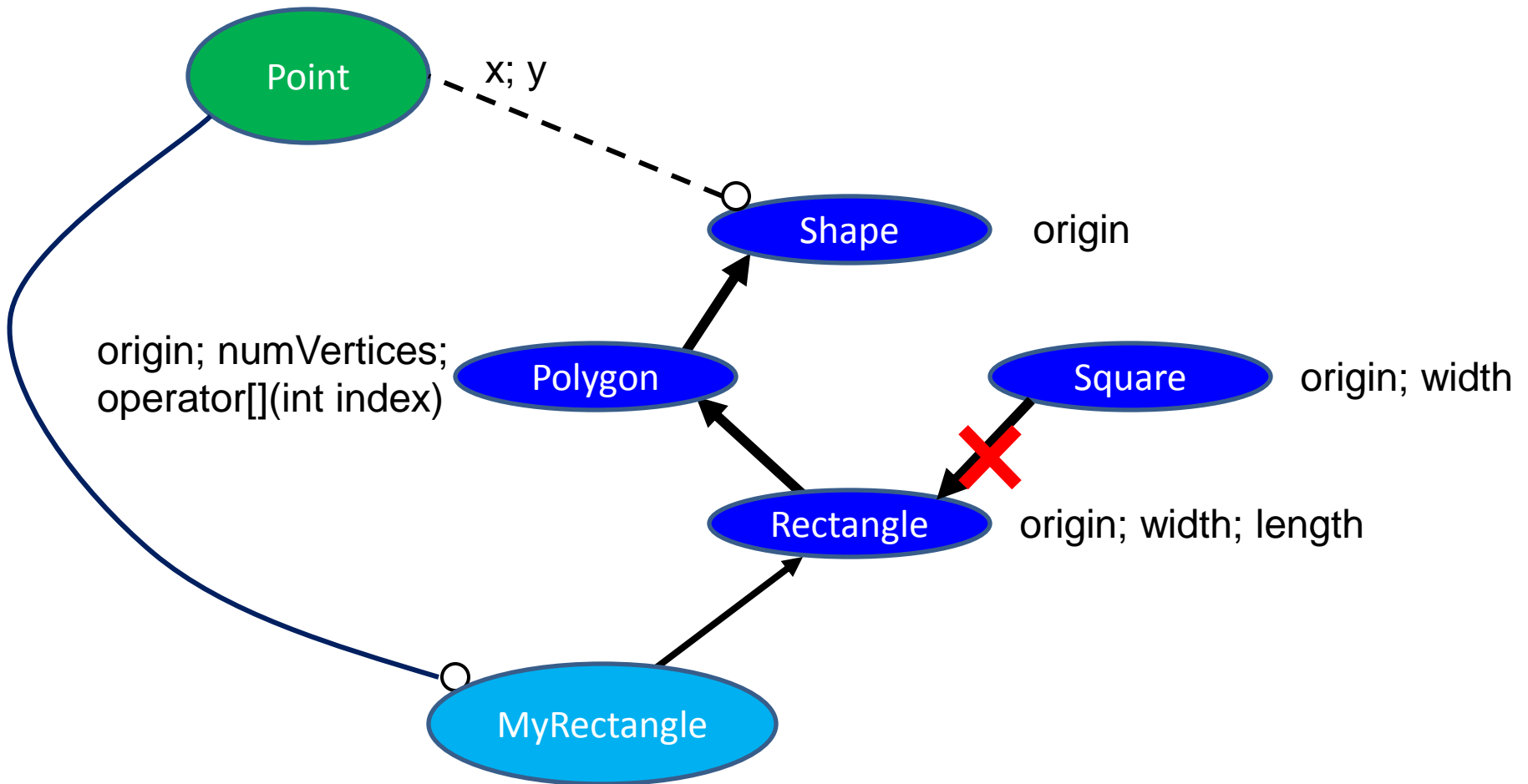
```
    { assert(wid == r->width());  
      assert(len + 1 == r->length());
```

```
}
```

Either **Assert**
or **No Longer**
Square

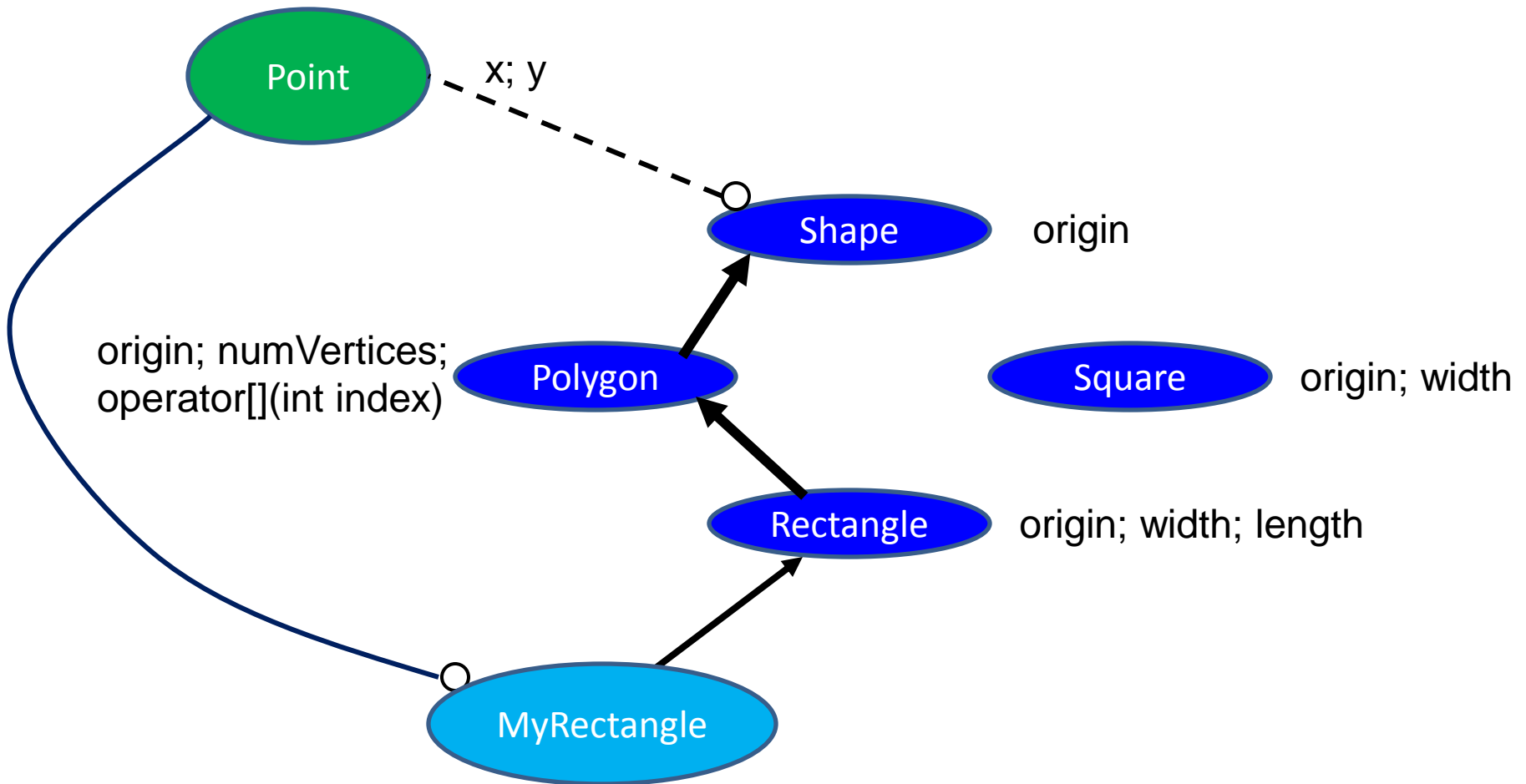
4. Proper Inheritance

Using Interface Inheritance Effectively



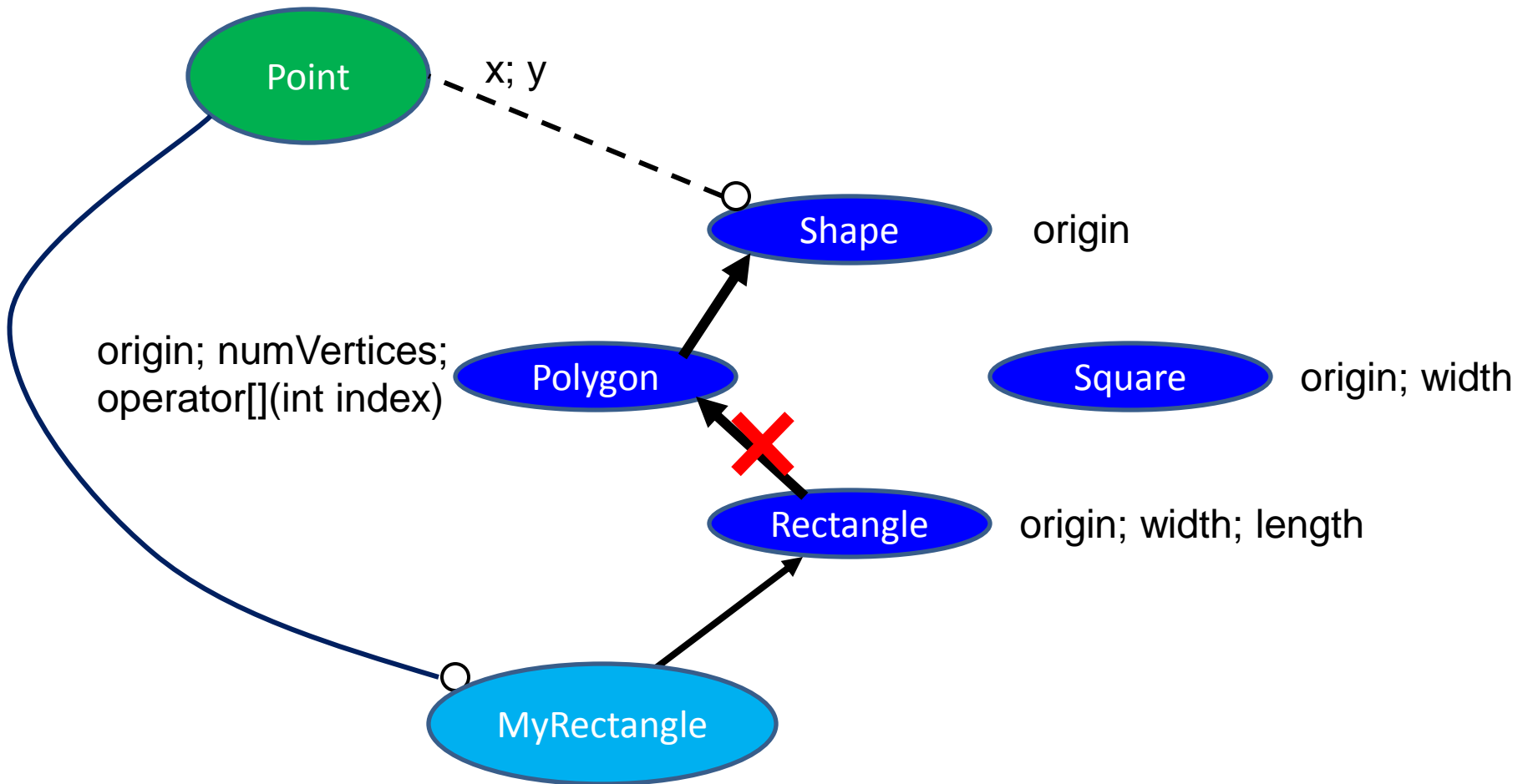
4. Proper Inheritance

Using Interface Inheritance Effectively



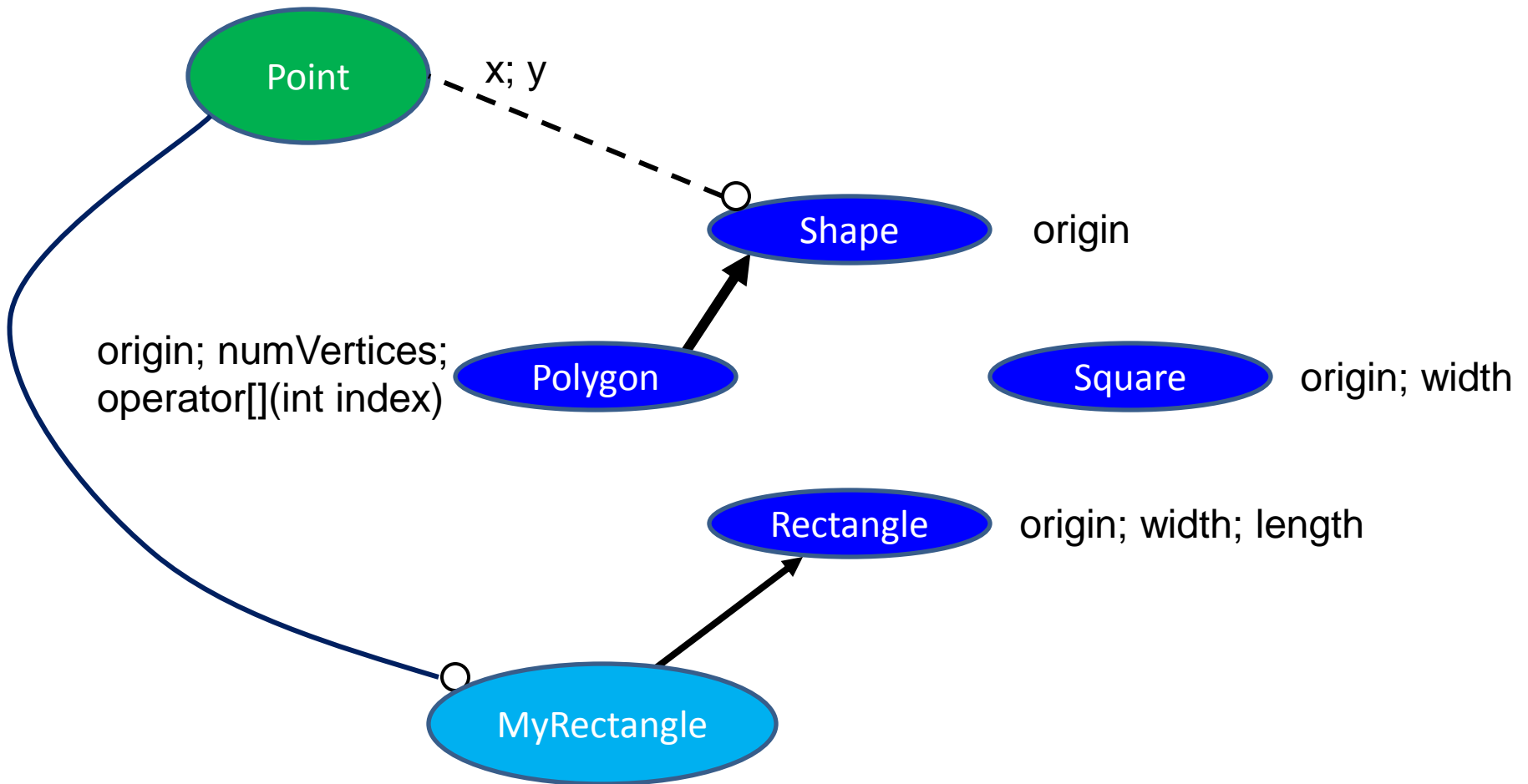
4. Proper Inheritance

Using Interface Inheritance Effectively



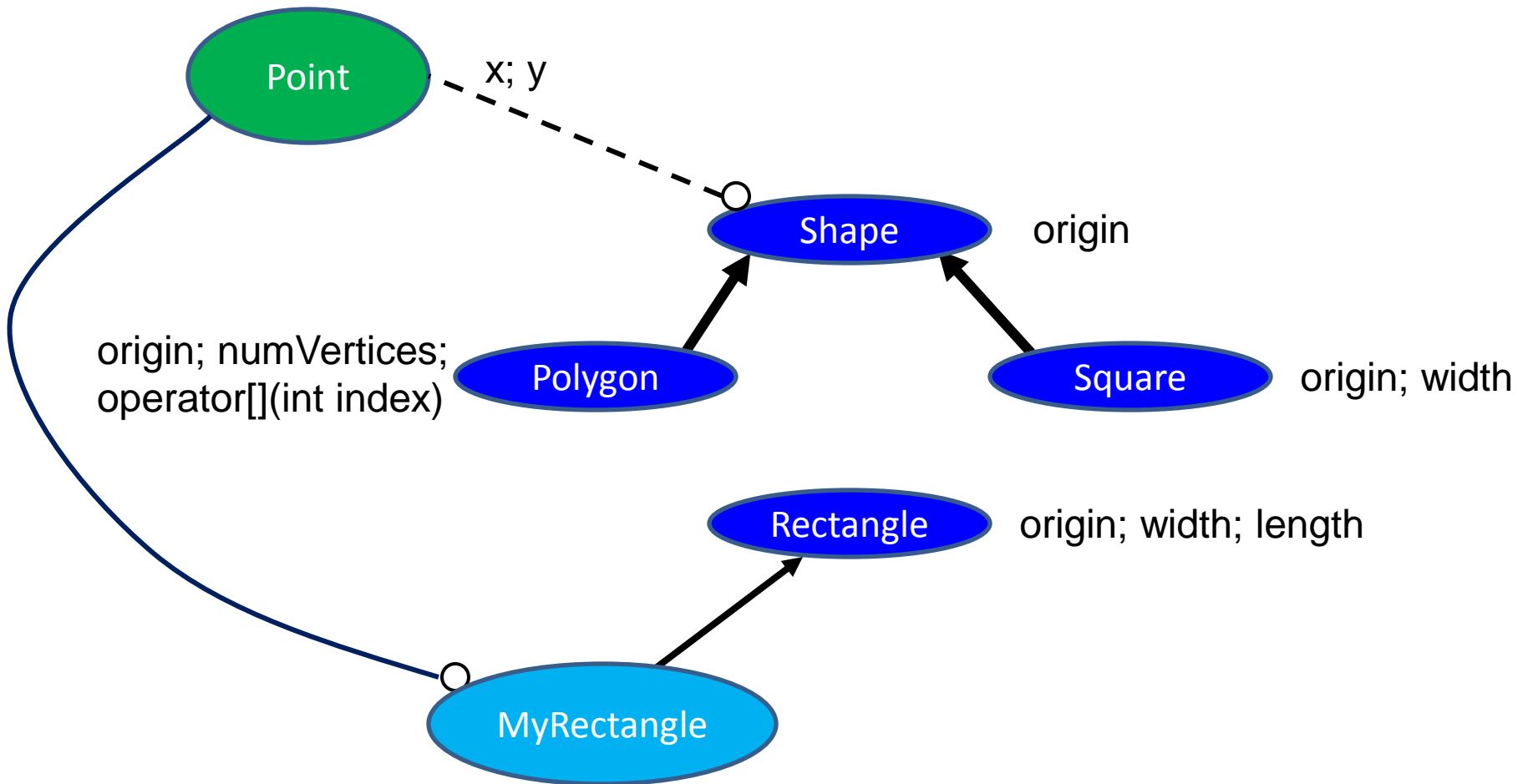
4. Proper Inheritance

Using Interface Inheritance Effectively



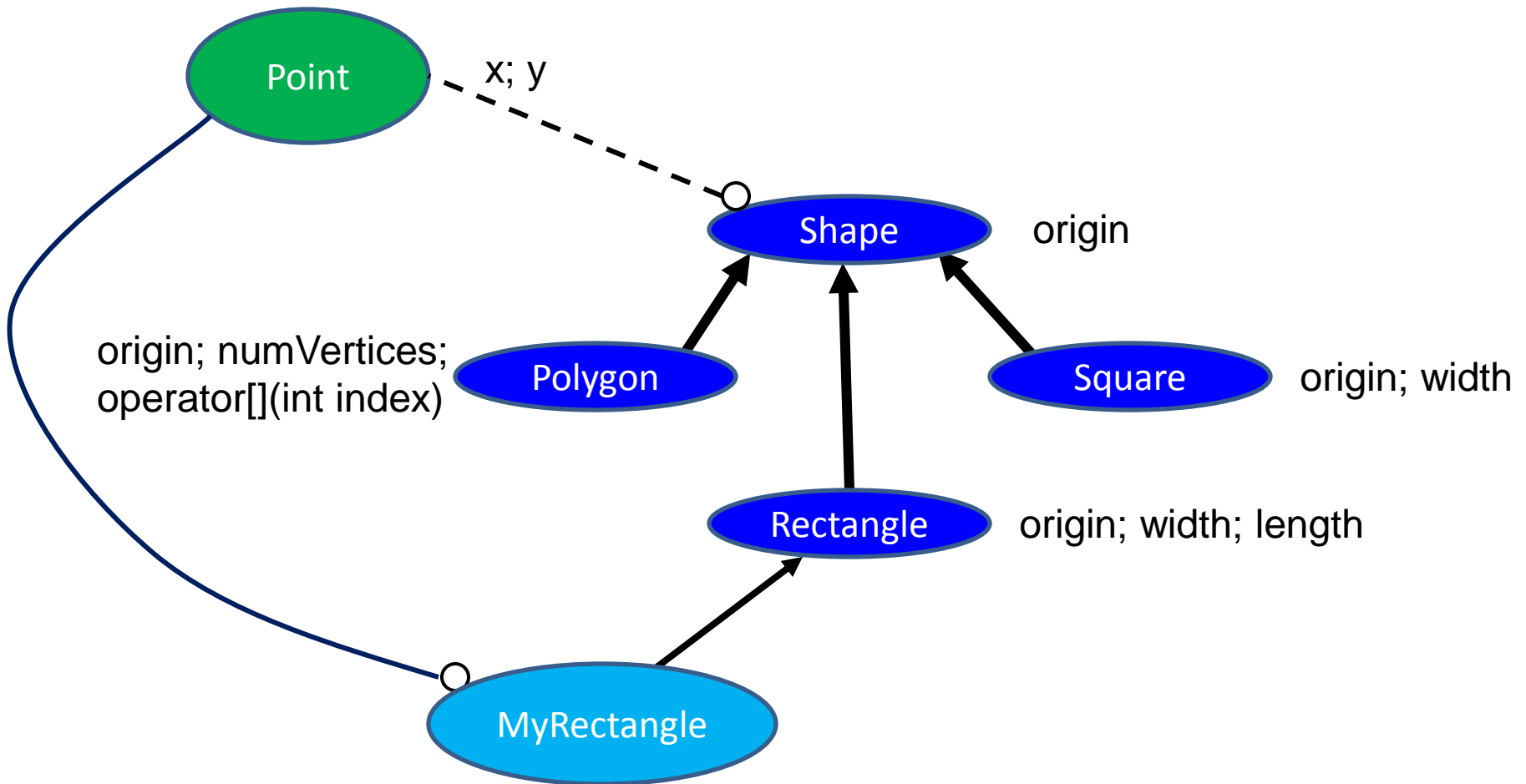
4. Proper Inheritance

Using Interface Inheritance Effectively



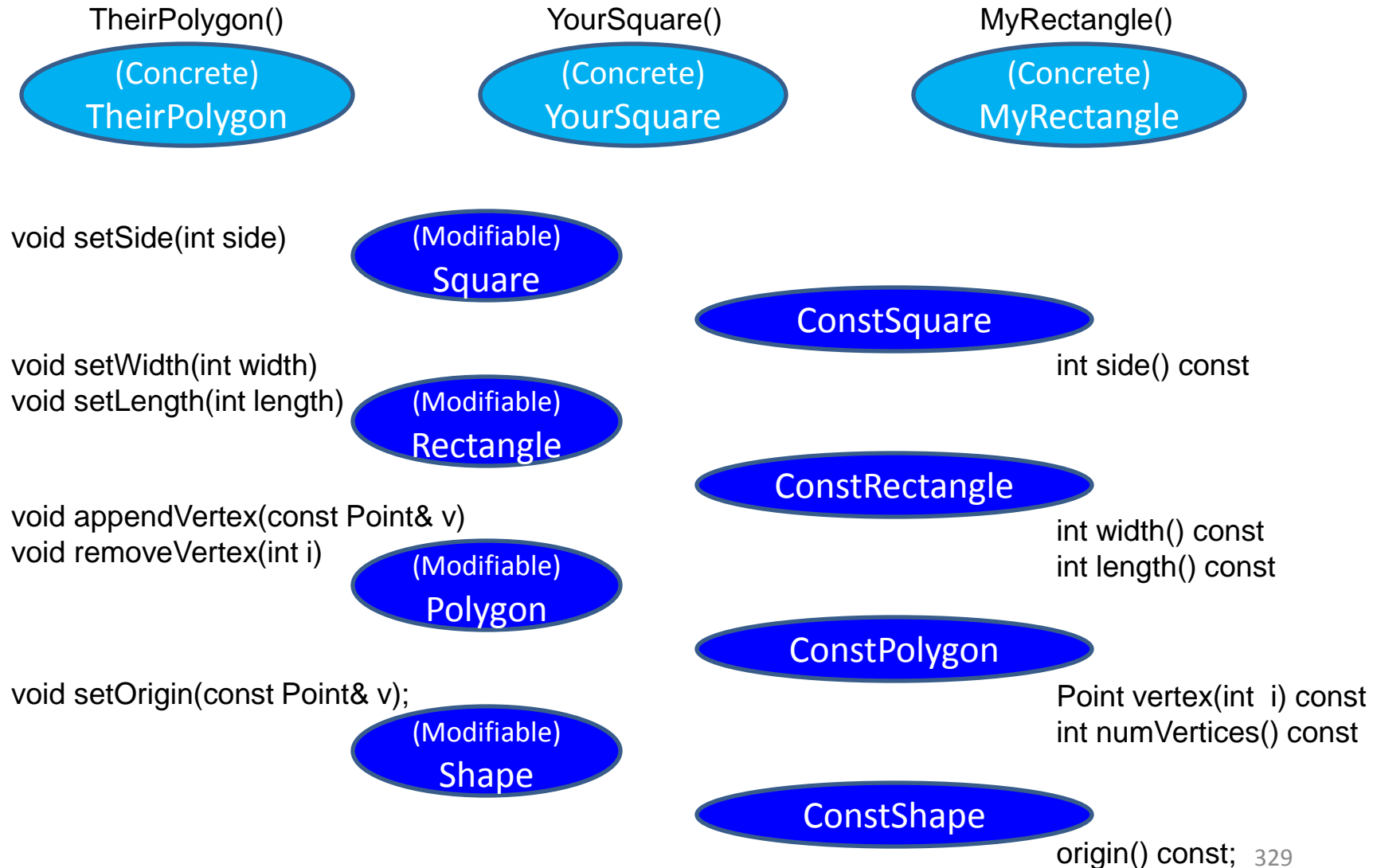
4. Proper Inheritance

Using Interface Inheritance Effectively



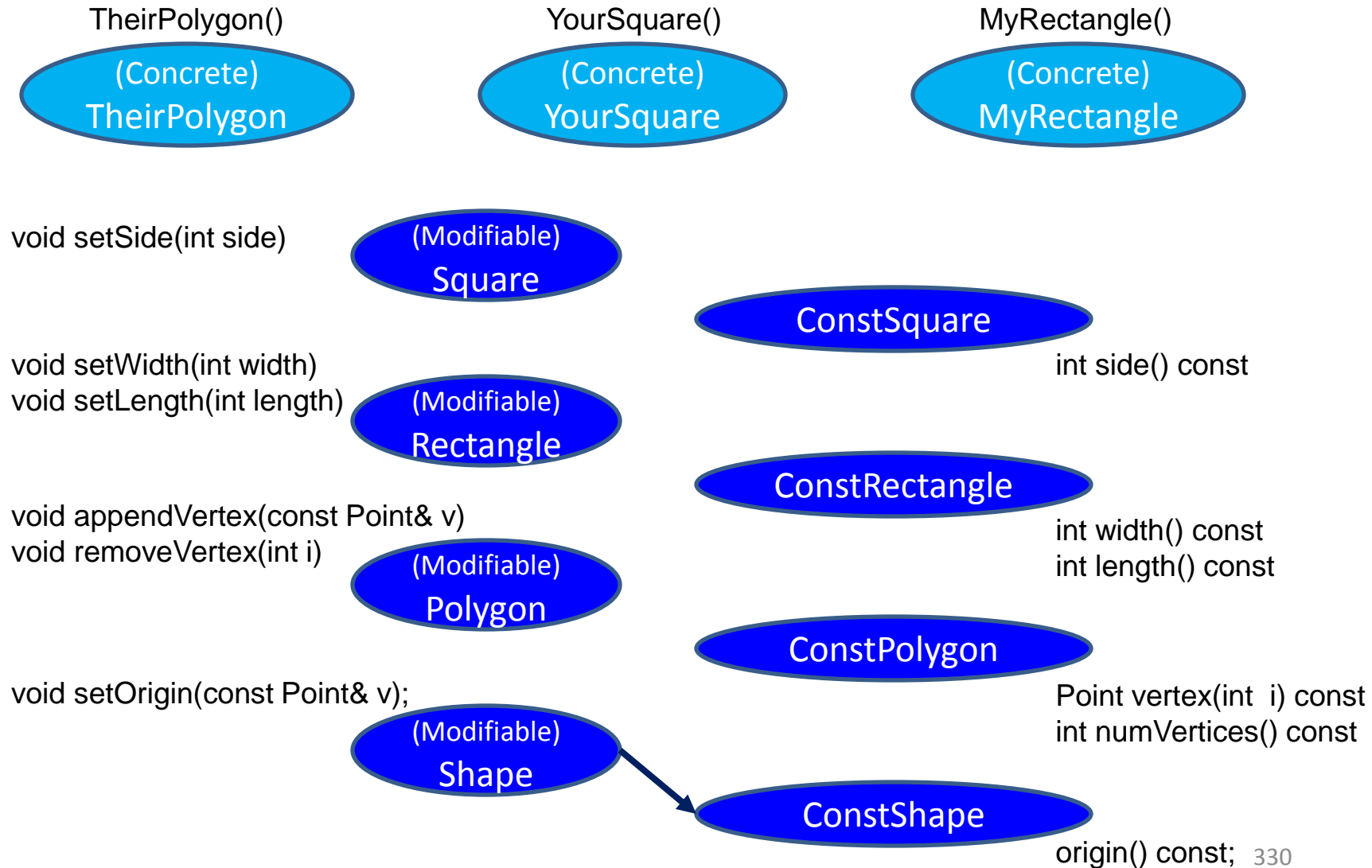
4. Proper Inheritance

Using Interface Inheritance Effectively



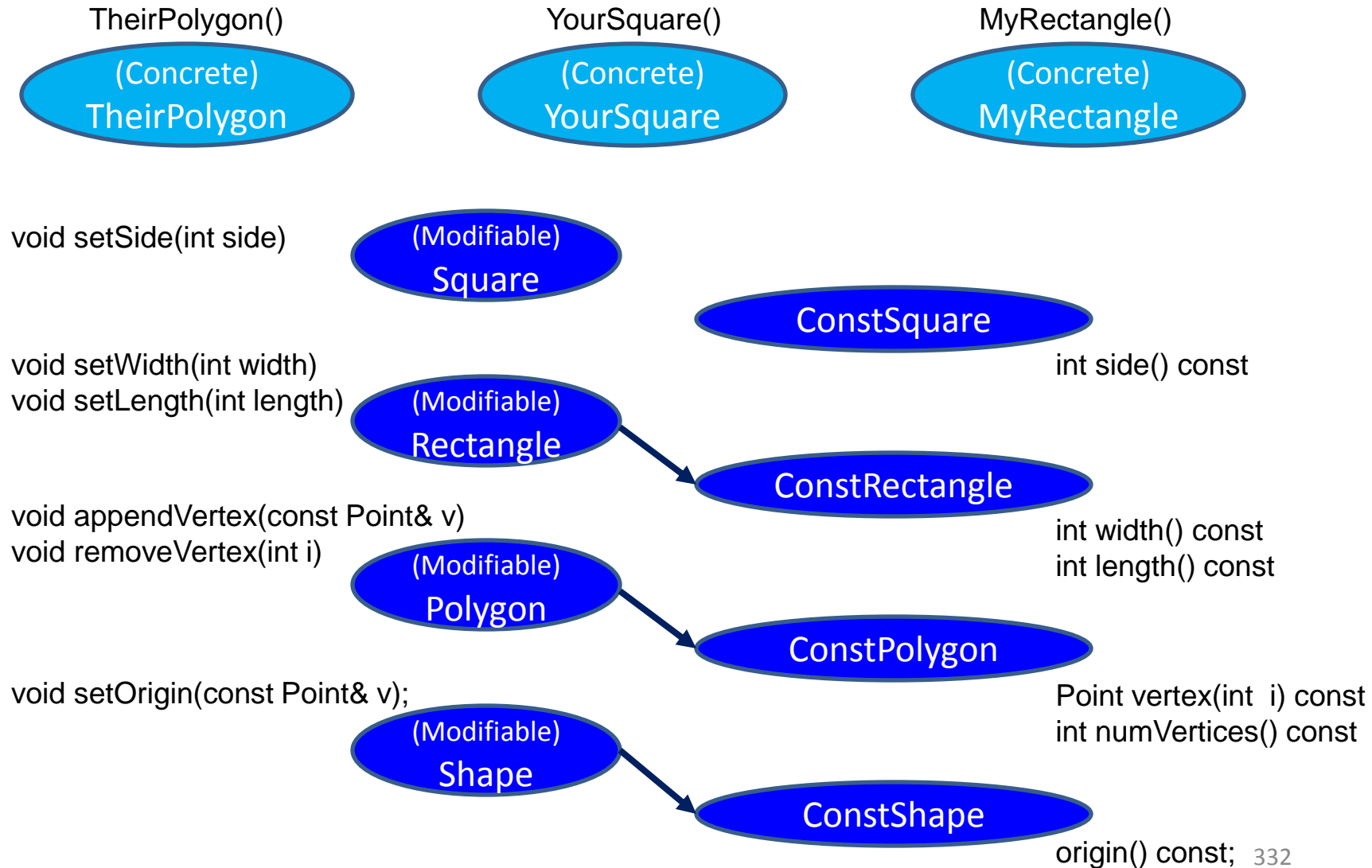
4. Proper Inheritance

Using Interface Inheritance Effectively



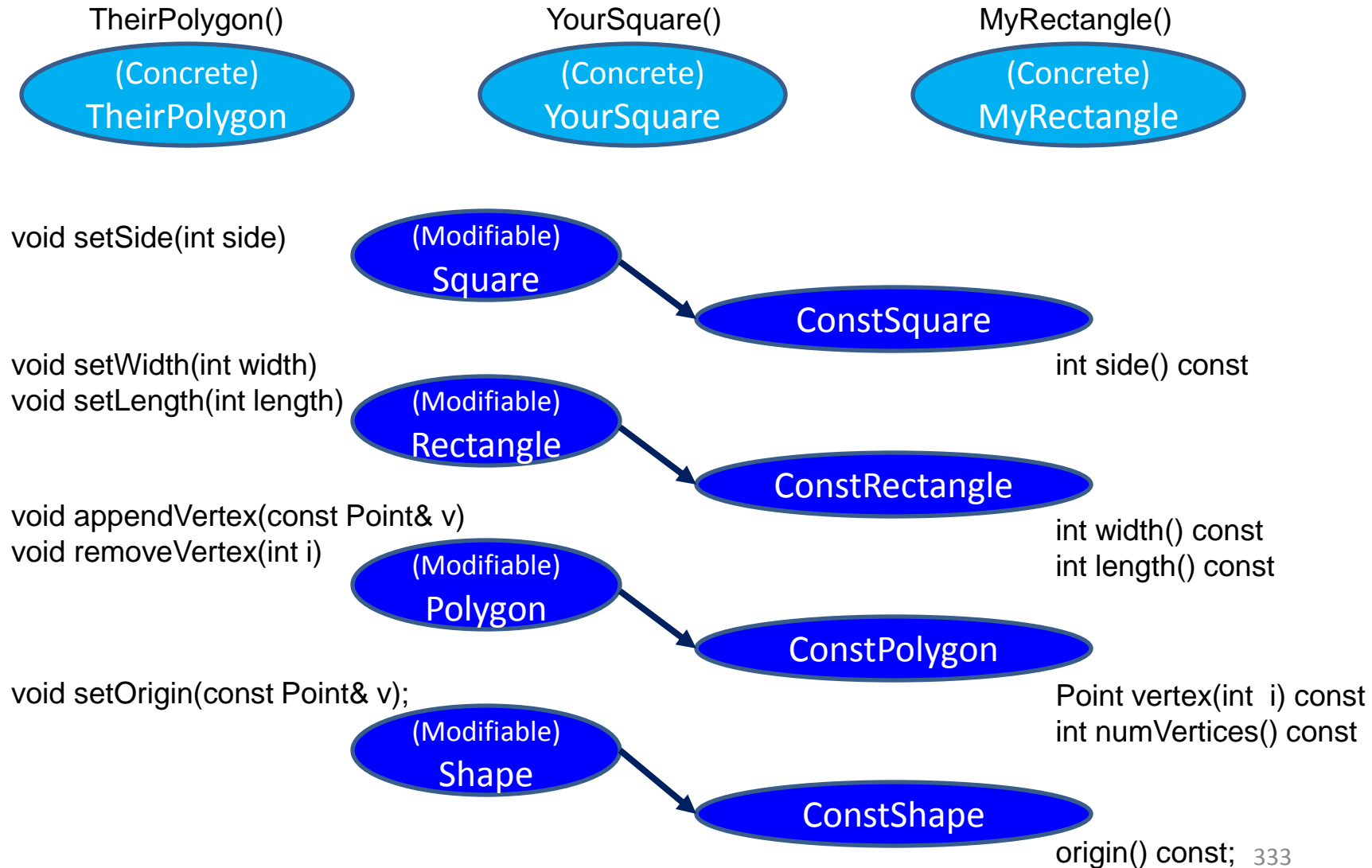
4. Proper Inheritance

Using Interface Inheritance Effectively



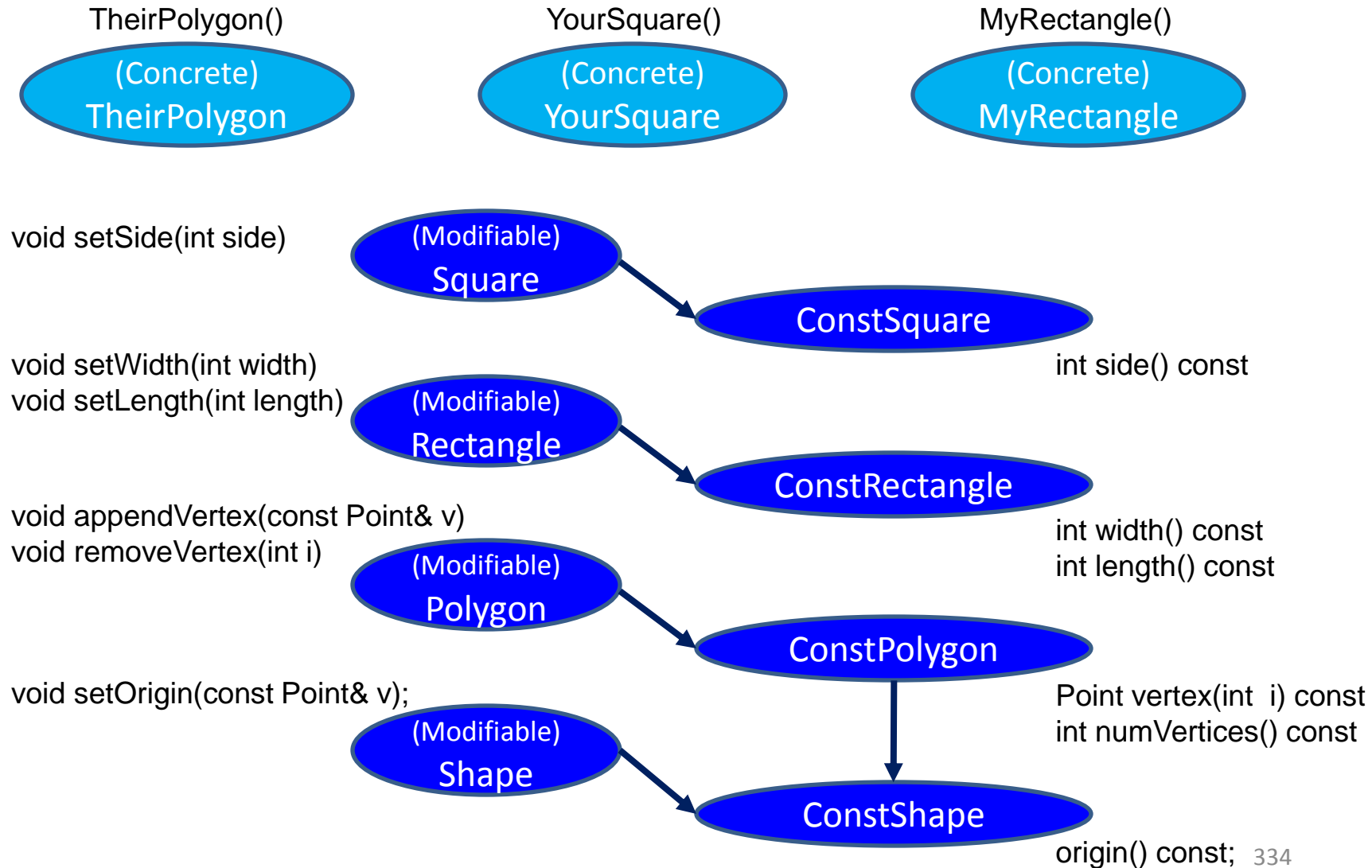
4. Proper Inheritance

Using Interface Inheritance Effectively



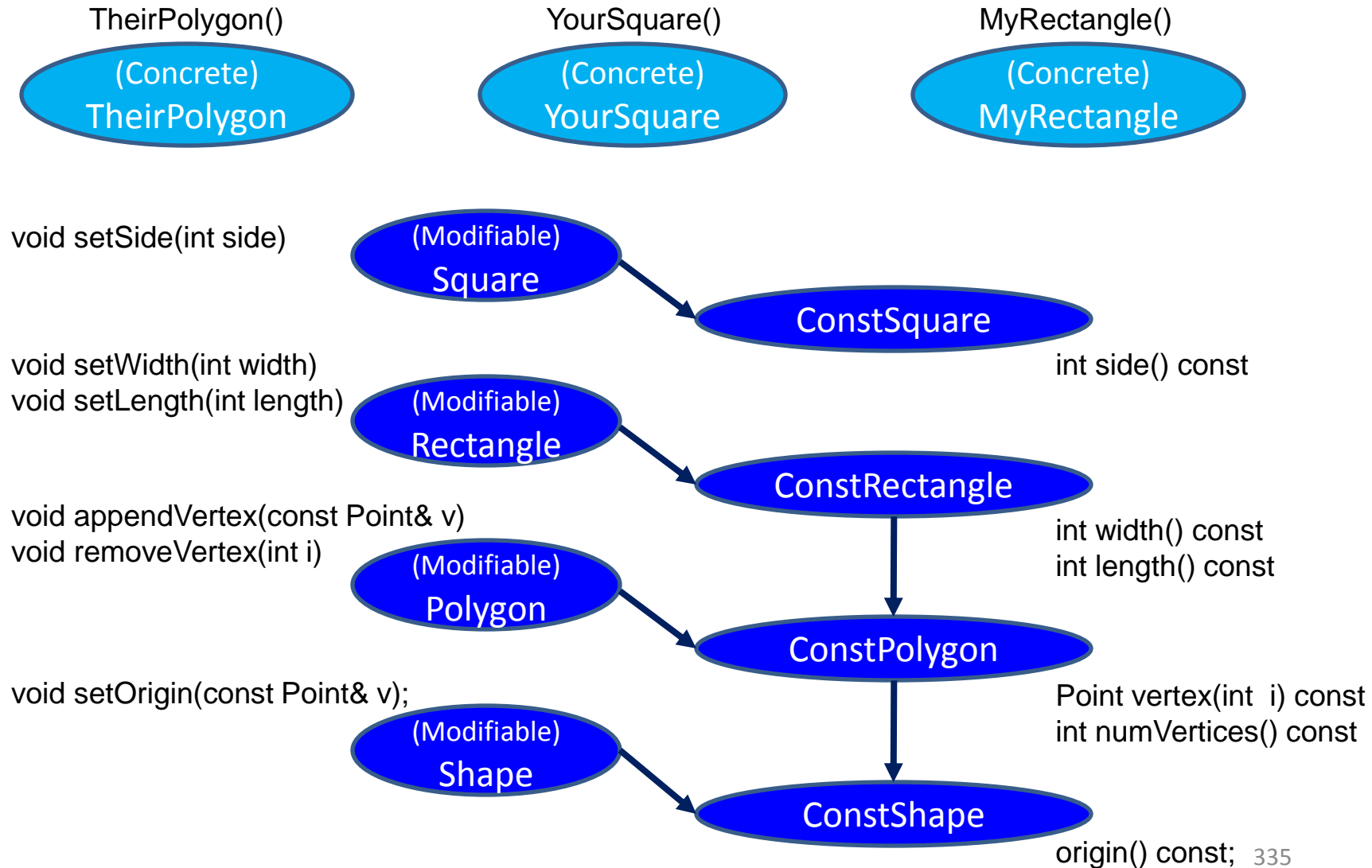
4. Proper Inheritance

Using Interface Inheritance Effectively



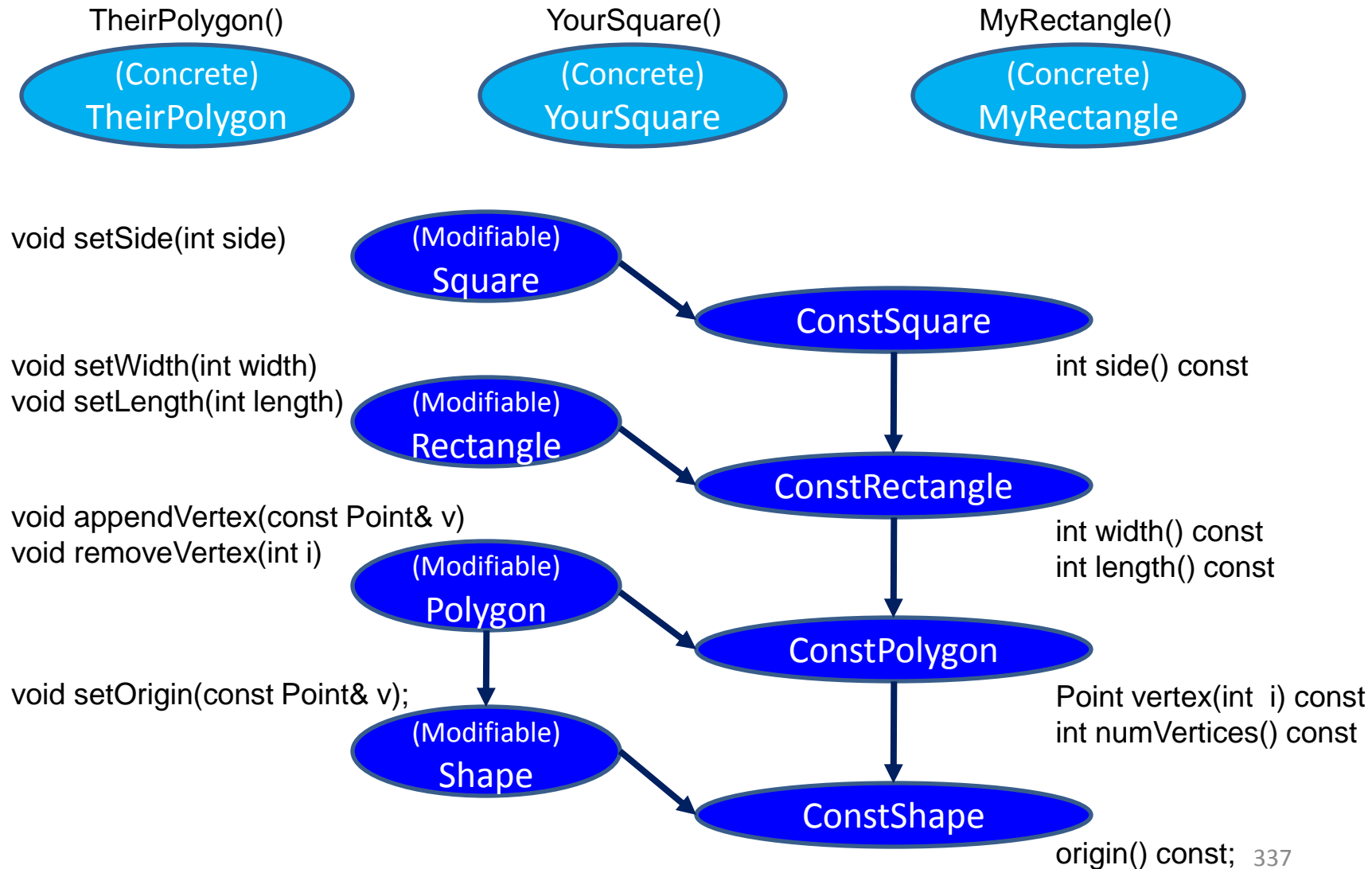
4. Proper Inheritance

Using Interface Inheritance Effectively



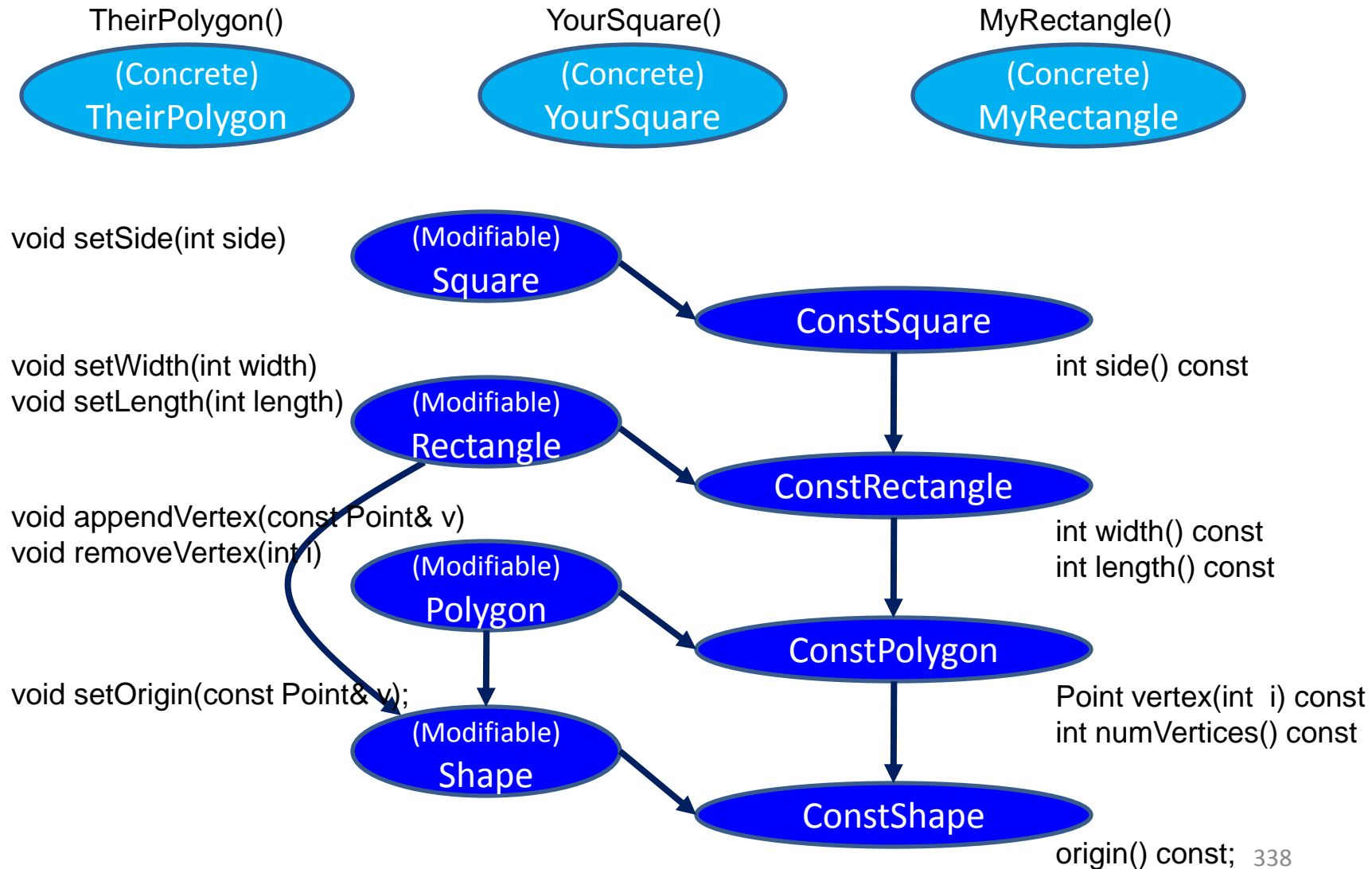
4. Proper Inheritance

Using Interface Inheritance Effectively



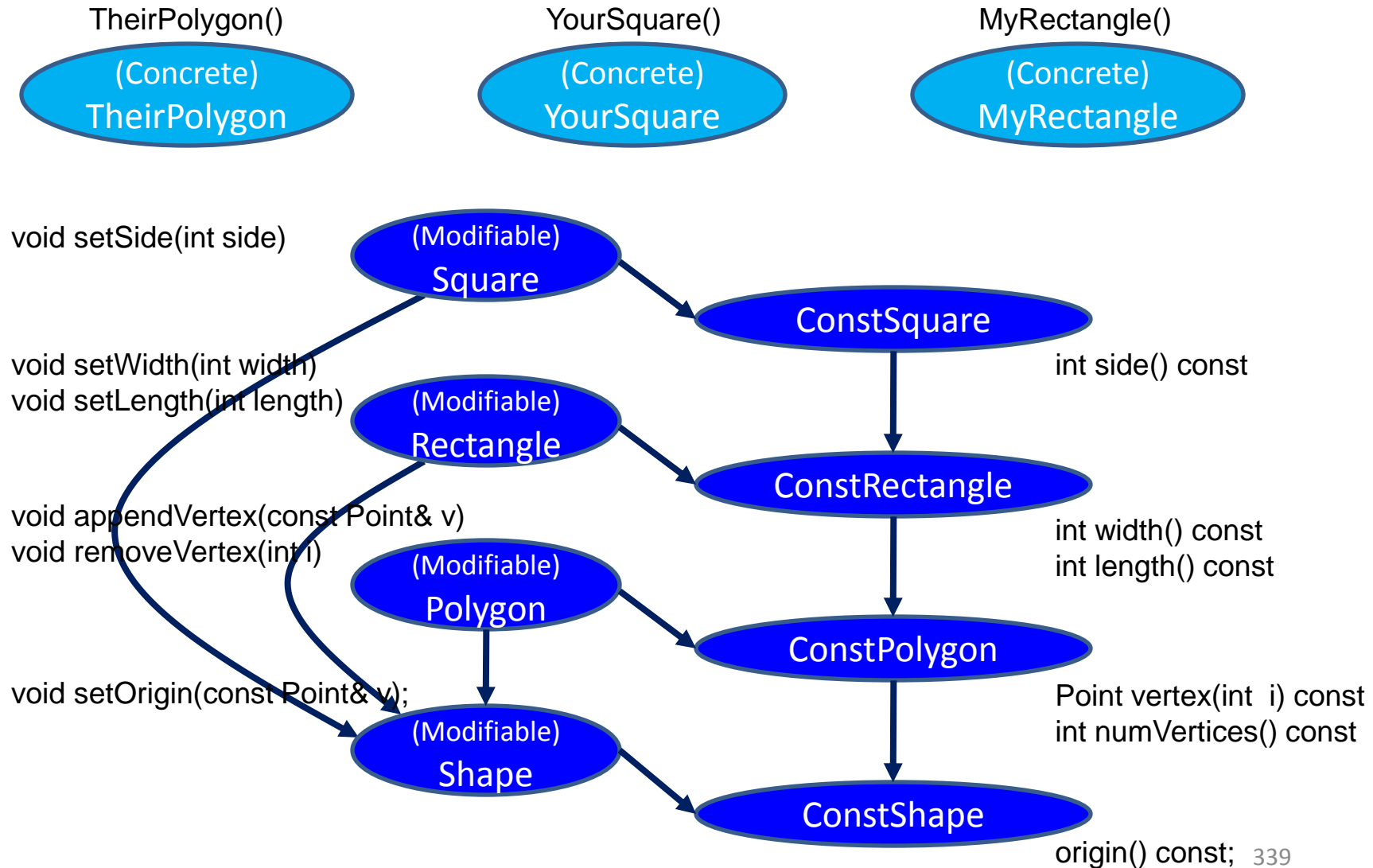
4. Proper Inheritance

Using Interface Inheritance Effectively



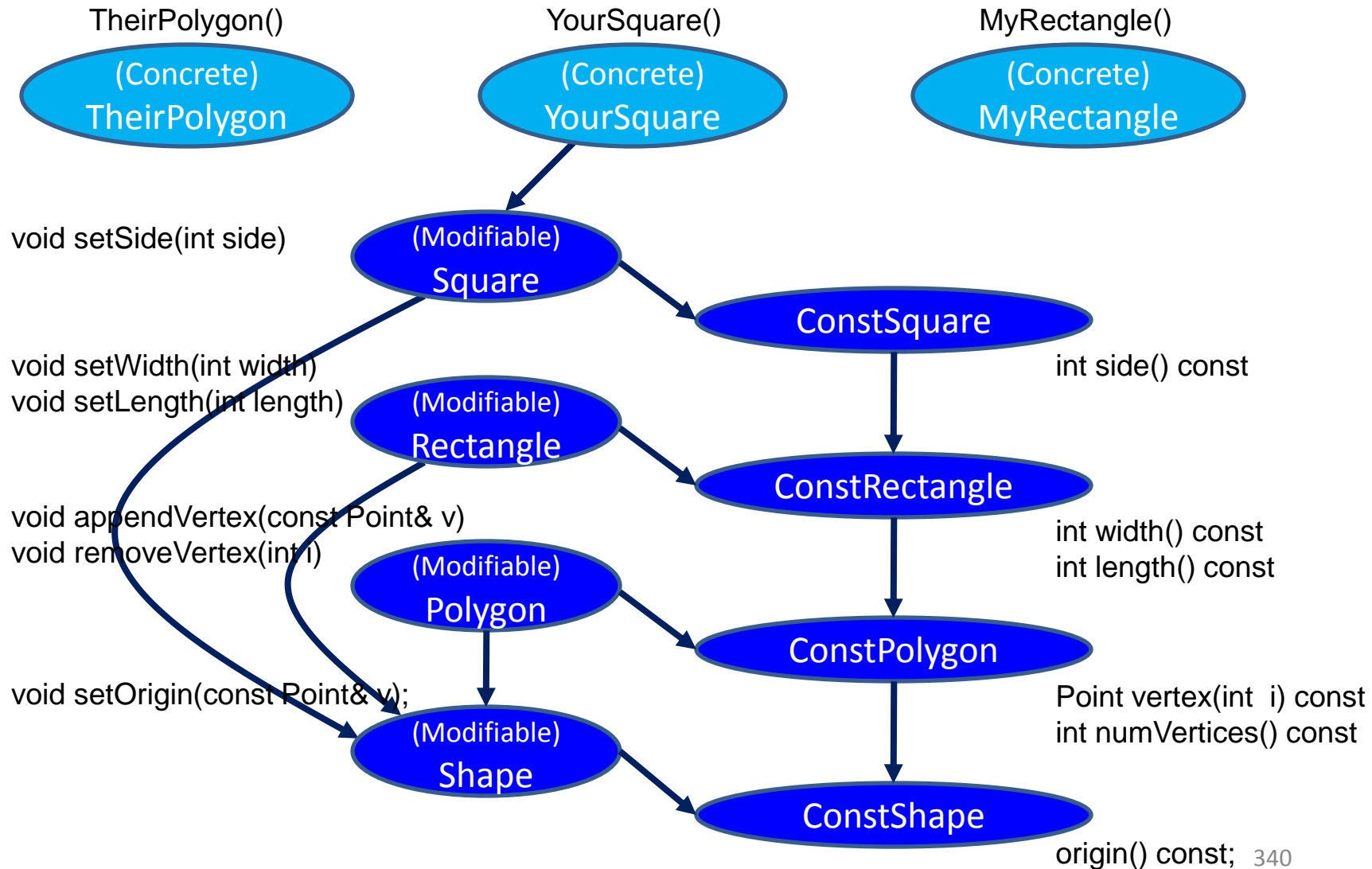
4. Proper Inheritance

Using Interface Inheritance Effectively



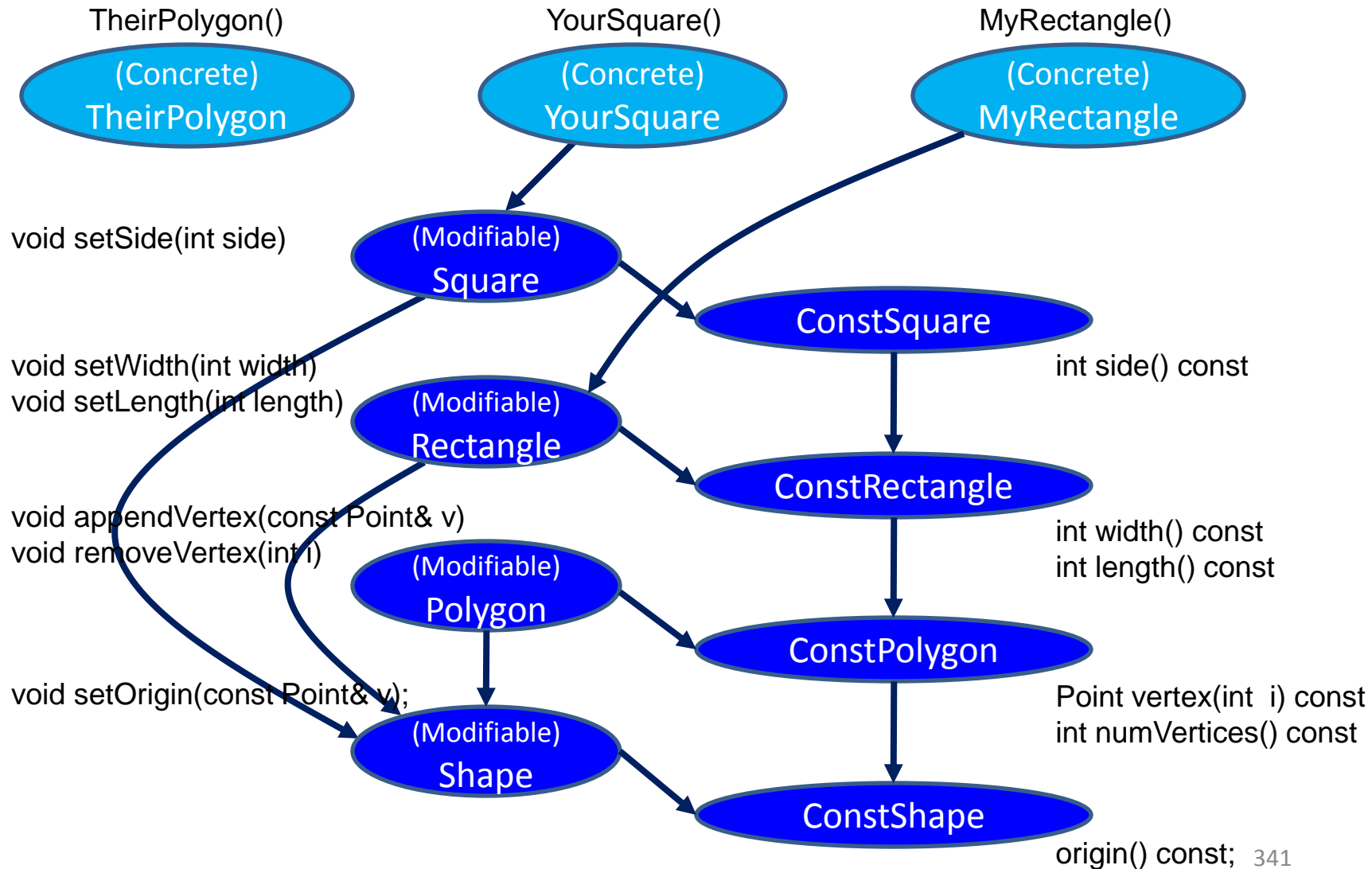
4. Proper Inheritance

Using Interface Inheritance Effectively



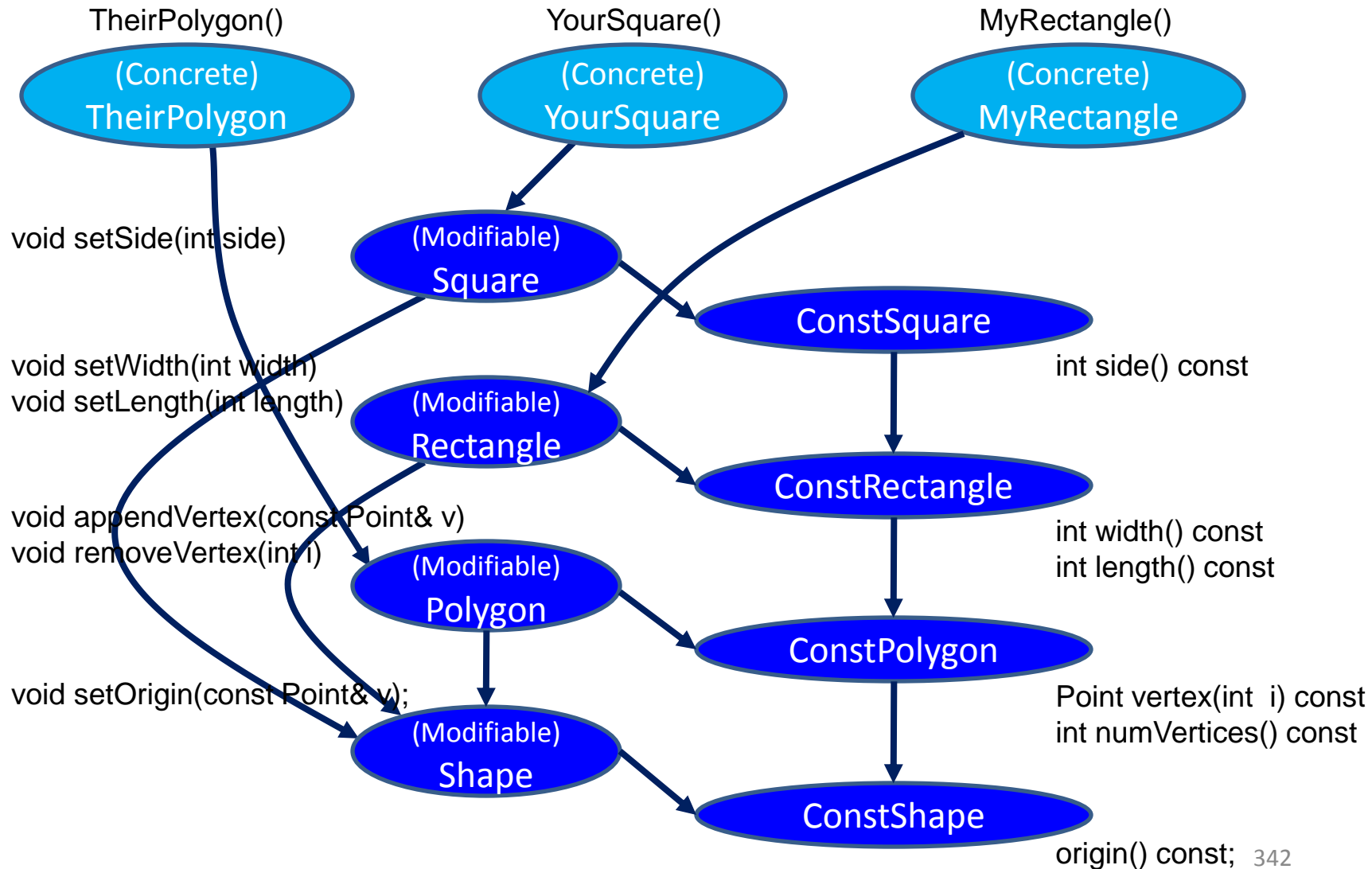
4. Proper Inheritance

Using Interface Inheritance Effectively



4. Proper Inheritance

Using Interface Inheritance Effectively



4. Proper Inheritance

Using Interface Inheritance Effectively

The principal clients of
Interface Inheritance
are **both** the
PUBLIC CLIENT
and the
DERIVED-CLASS AUTHOR.

4. Proper Inheritance

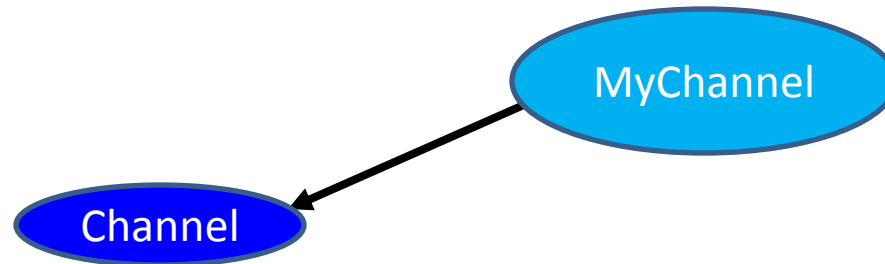
Using Interface Inheritance Effectively

Channel

```
int write(const char *b, int n);  
int read(char *b, int n);
```


4. Proper Inheritance

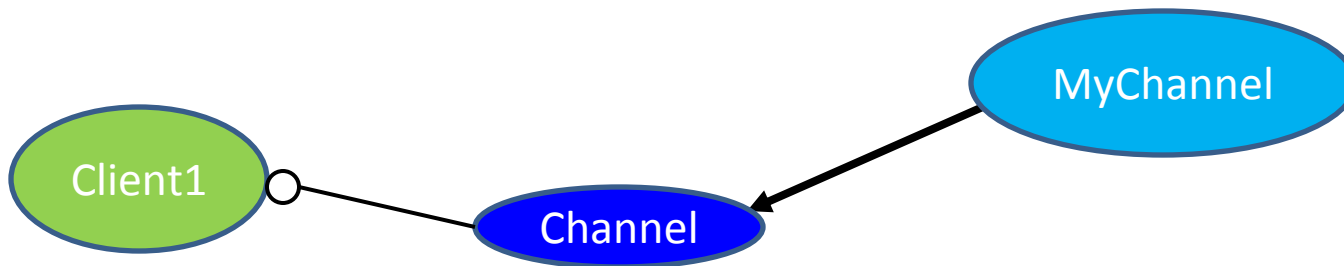
Using Interface Inheritance Effectively



```
int write(const char *b, int n);  
int read(char *b, int n);
```

4. Proper Inheritance

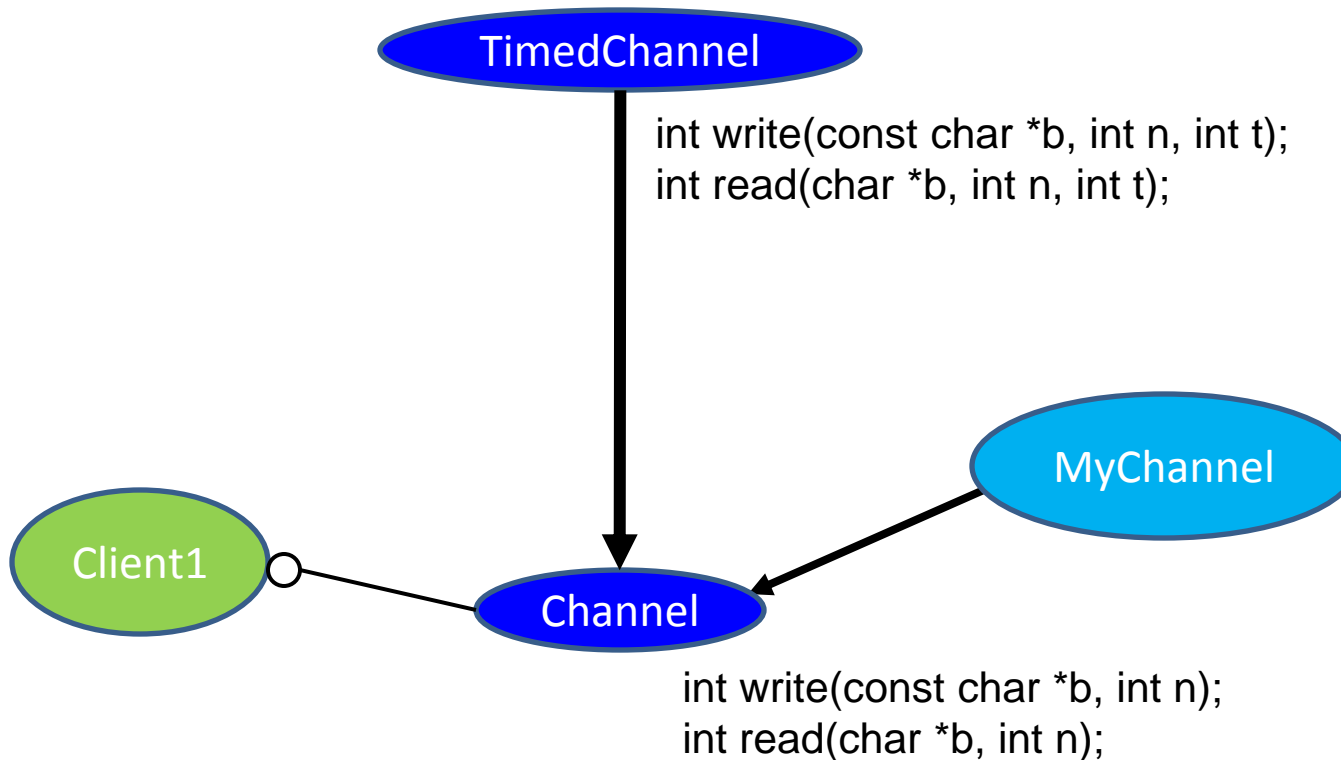
Using Interface Inheritance Effectively



```
int write(const char *b, int n);  
int read(char *b, int n);
```

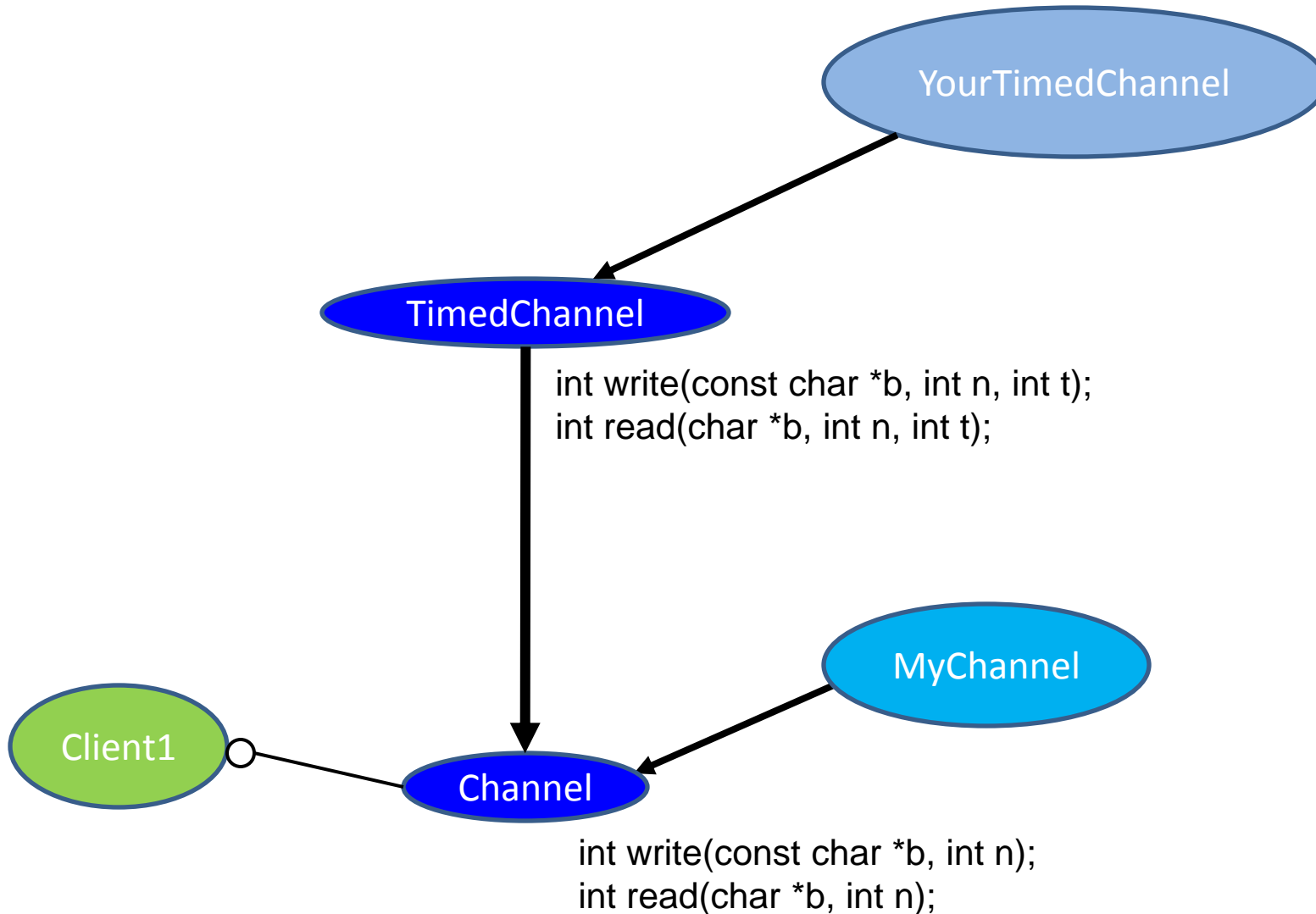
4. Proper Inheritance

Using Interface Inheritance Effectively



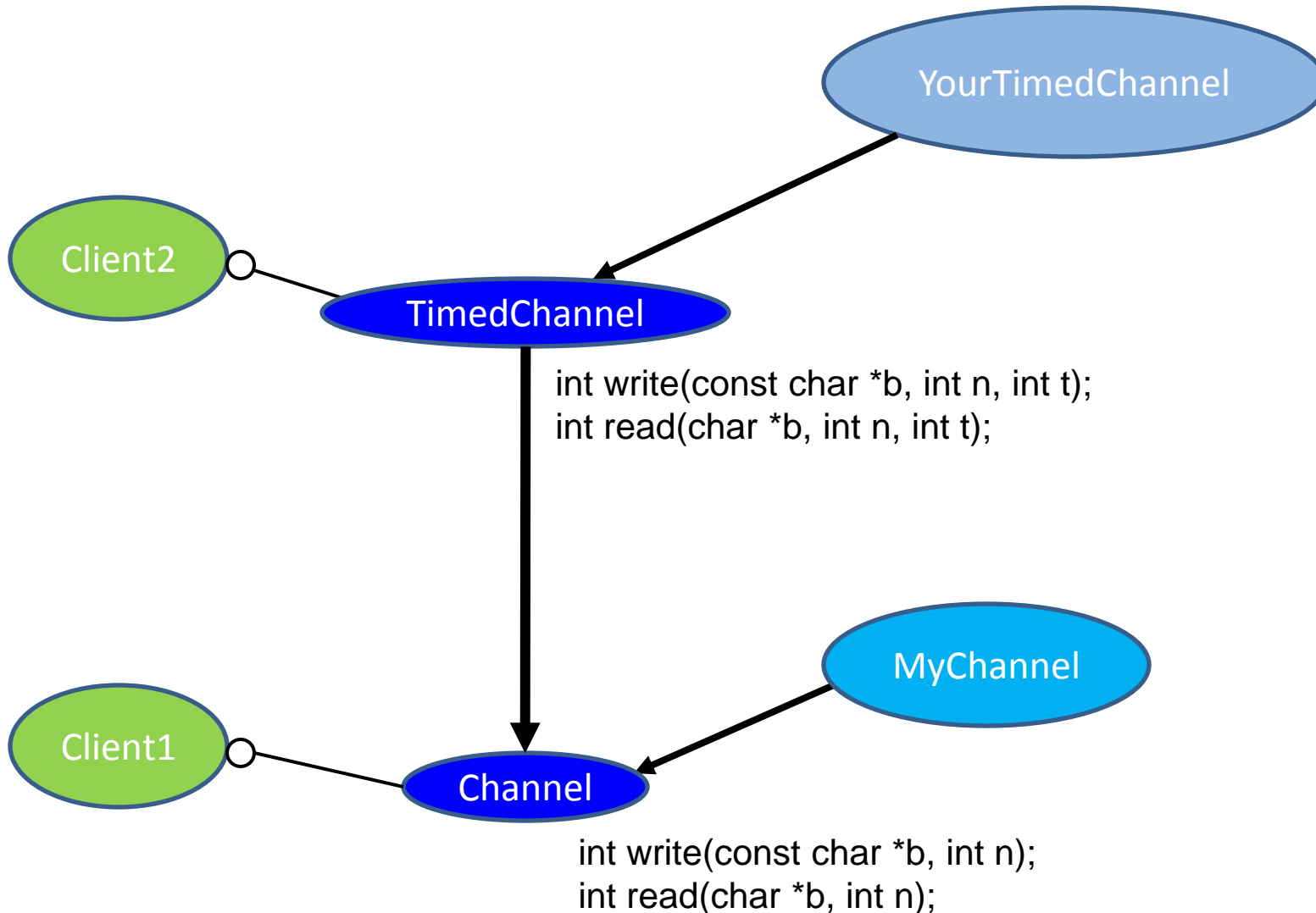
4. Proper Inheritance

Using Interface Inheritance Effectively



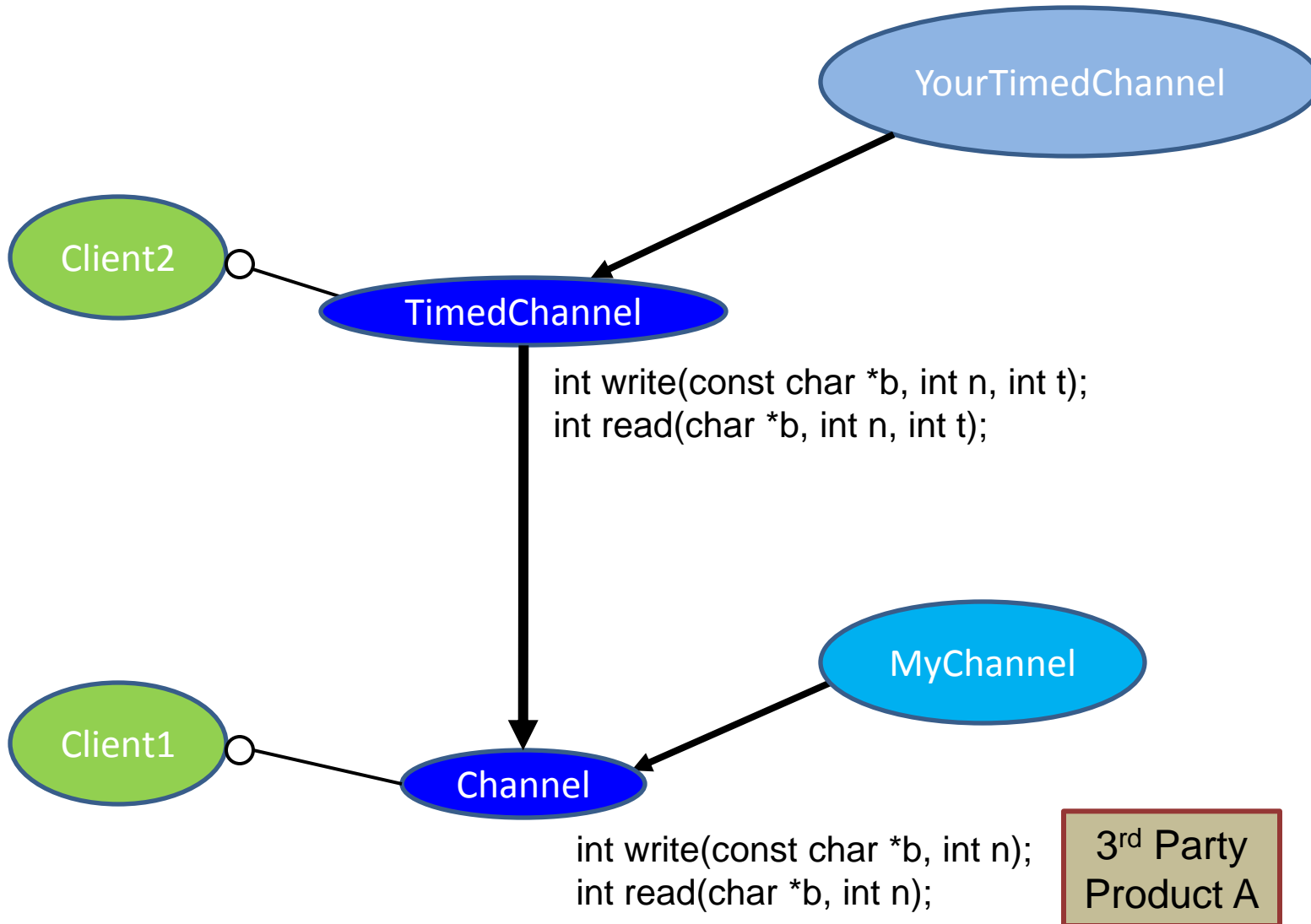
4. Proper Inheritance

Using Interface Inheritance Effectively



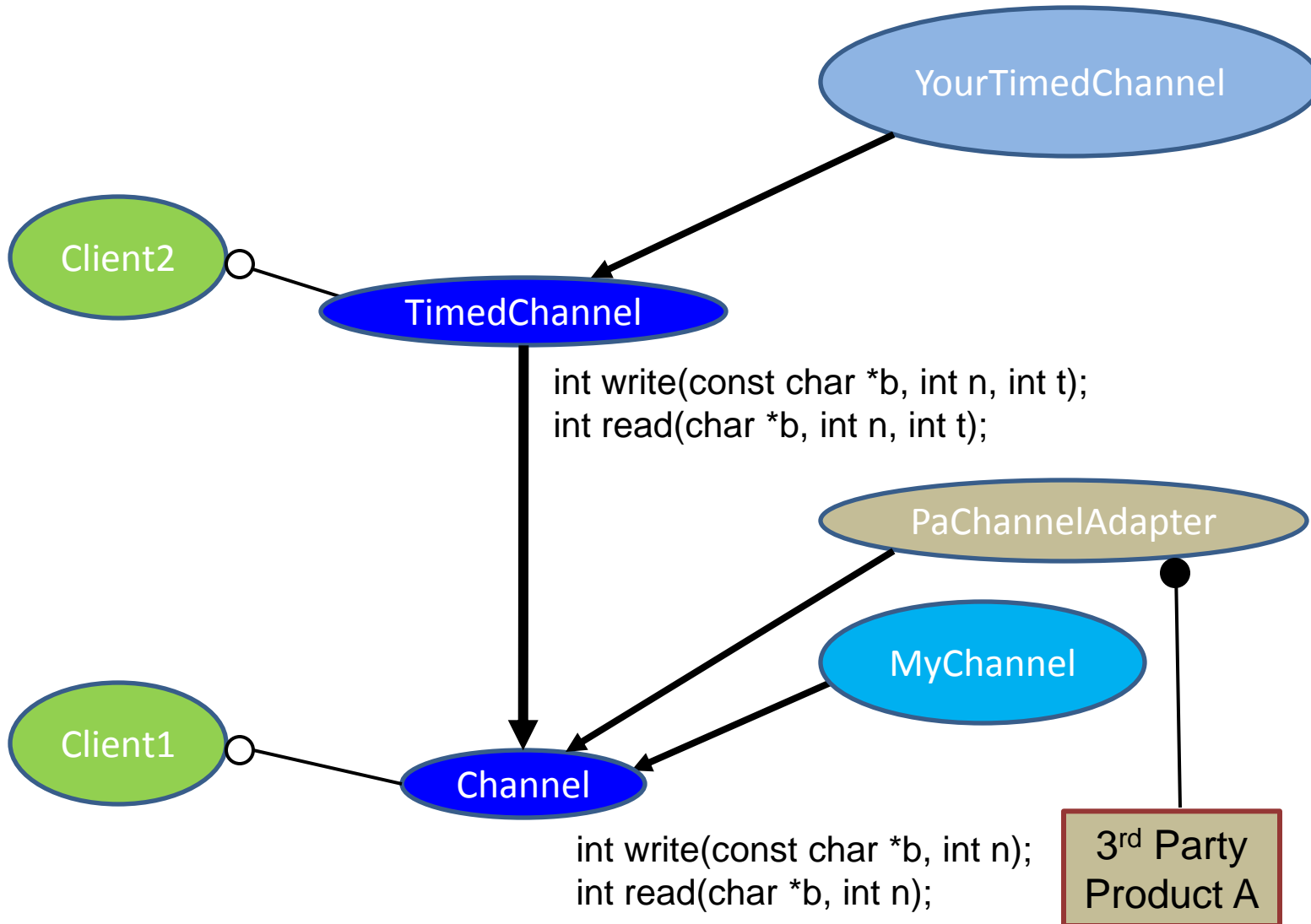
4. Proper Inheritance

Using Interface Inheritance Effectively



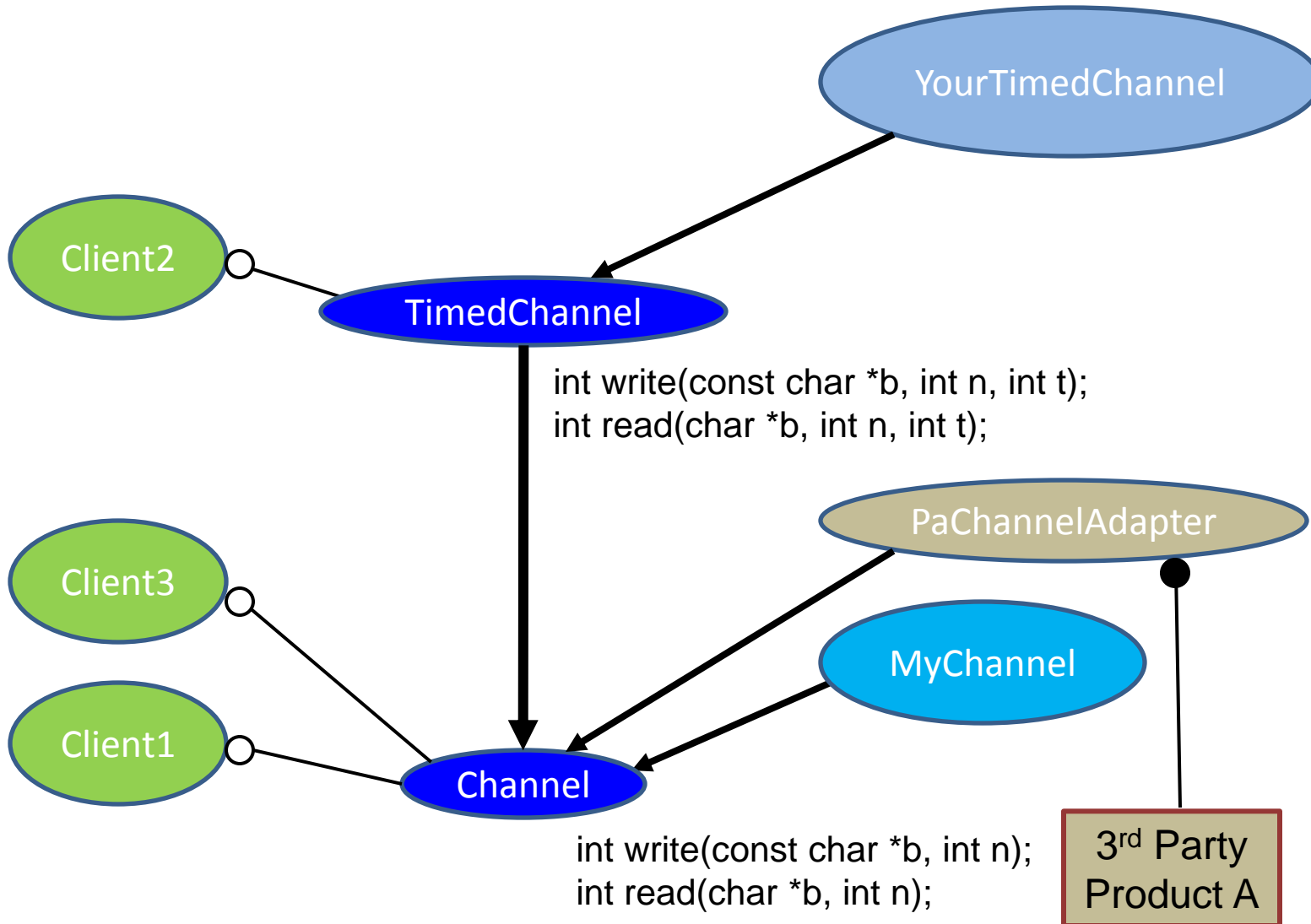
4. Proper Inheritance

Using Interface Inheritance Effectively



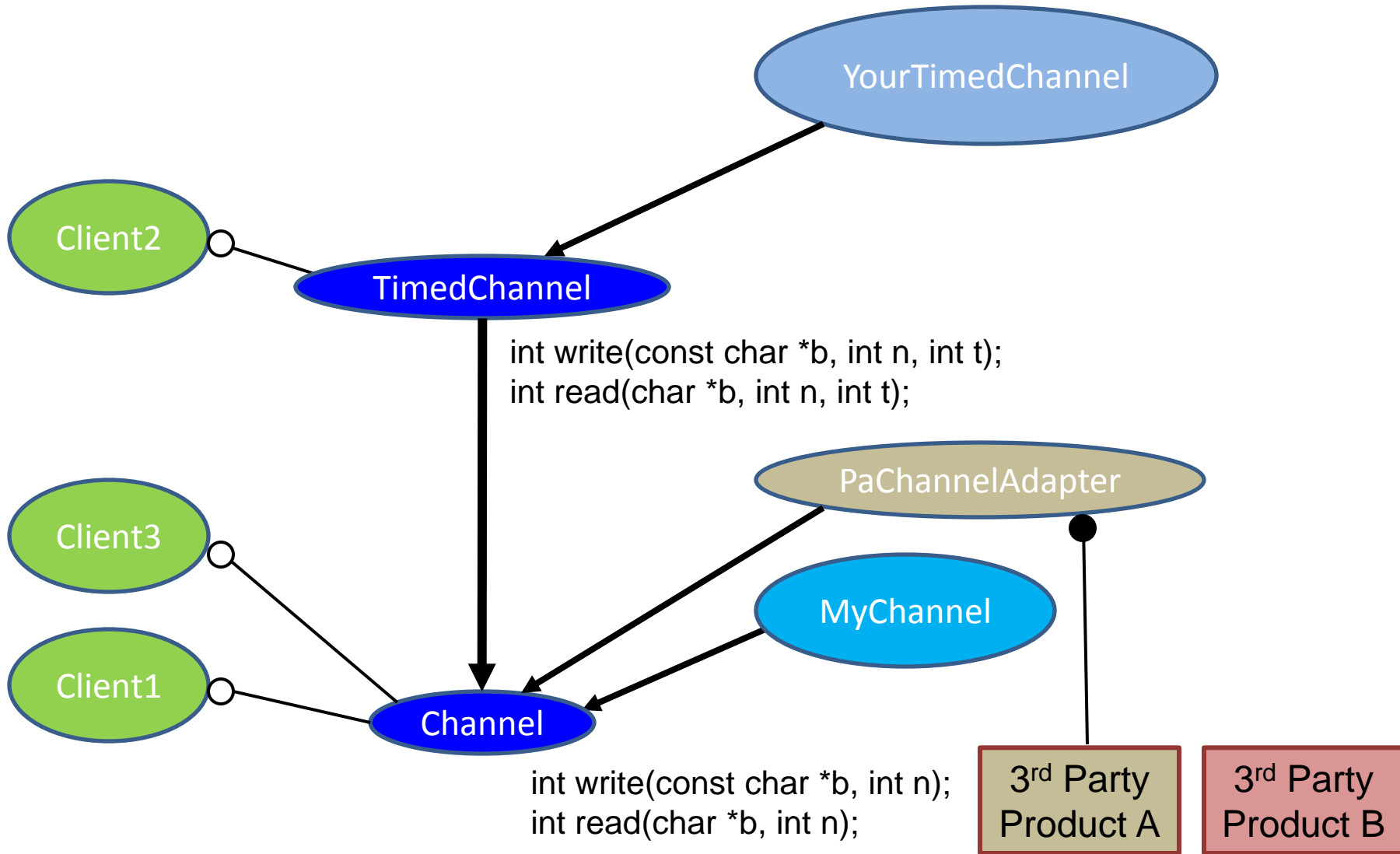
4. Proper Inheritance

Using Interface Inheritance Effectively



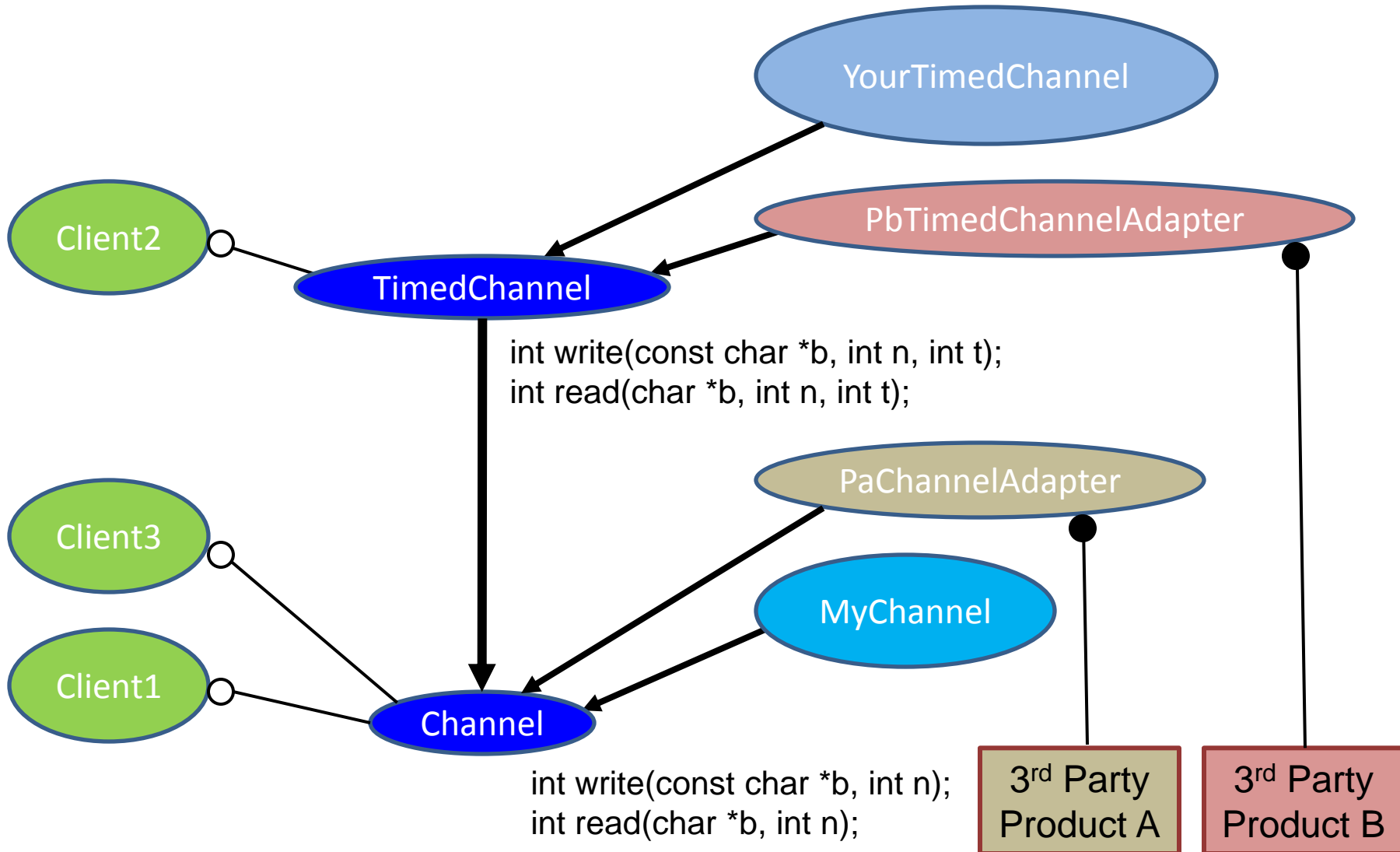
4. Proper Inheritance

Using Interface Inheritance Effectively



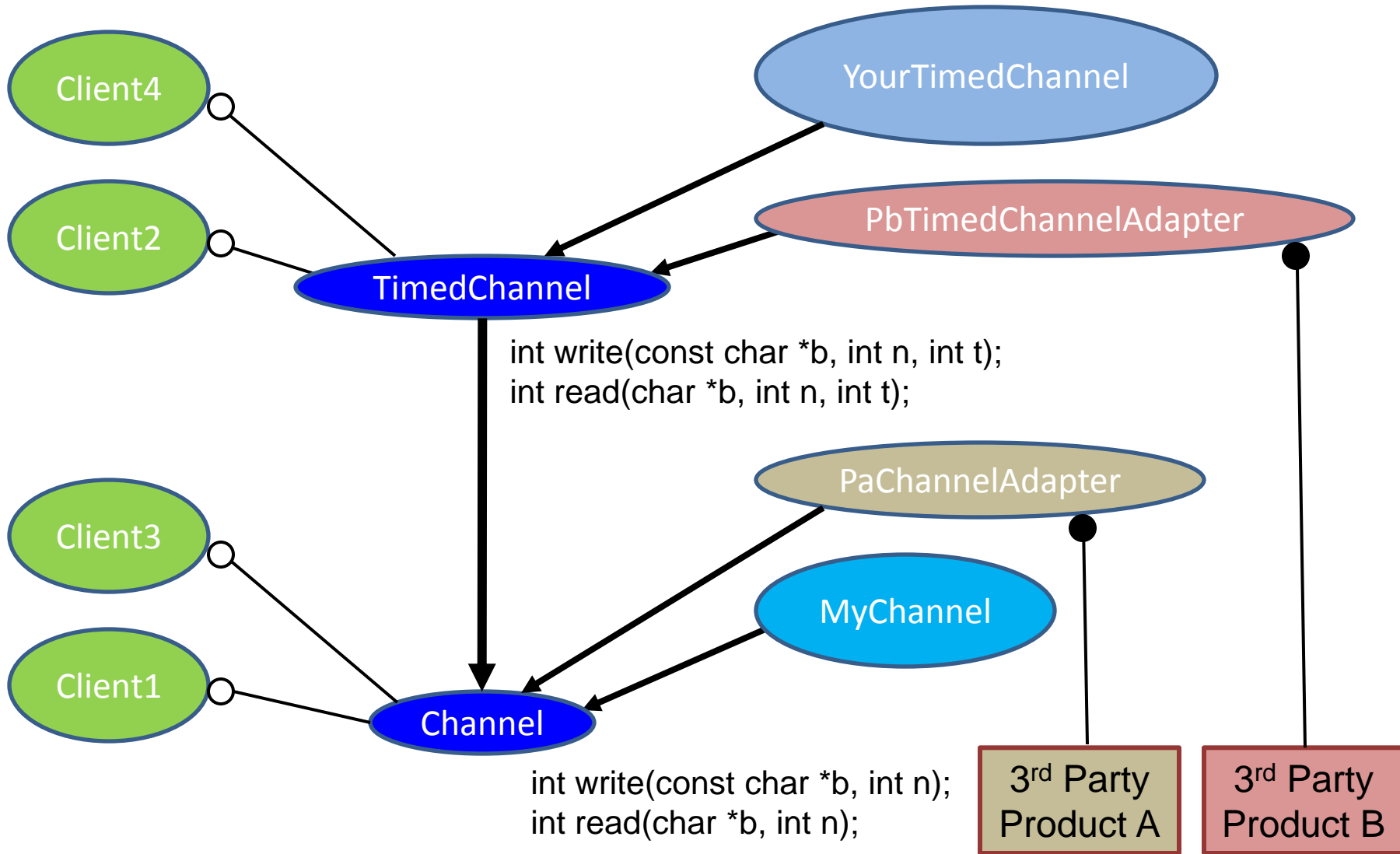
4. Proper Inheritance

Using Interface Inheritance Effectively



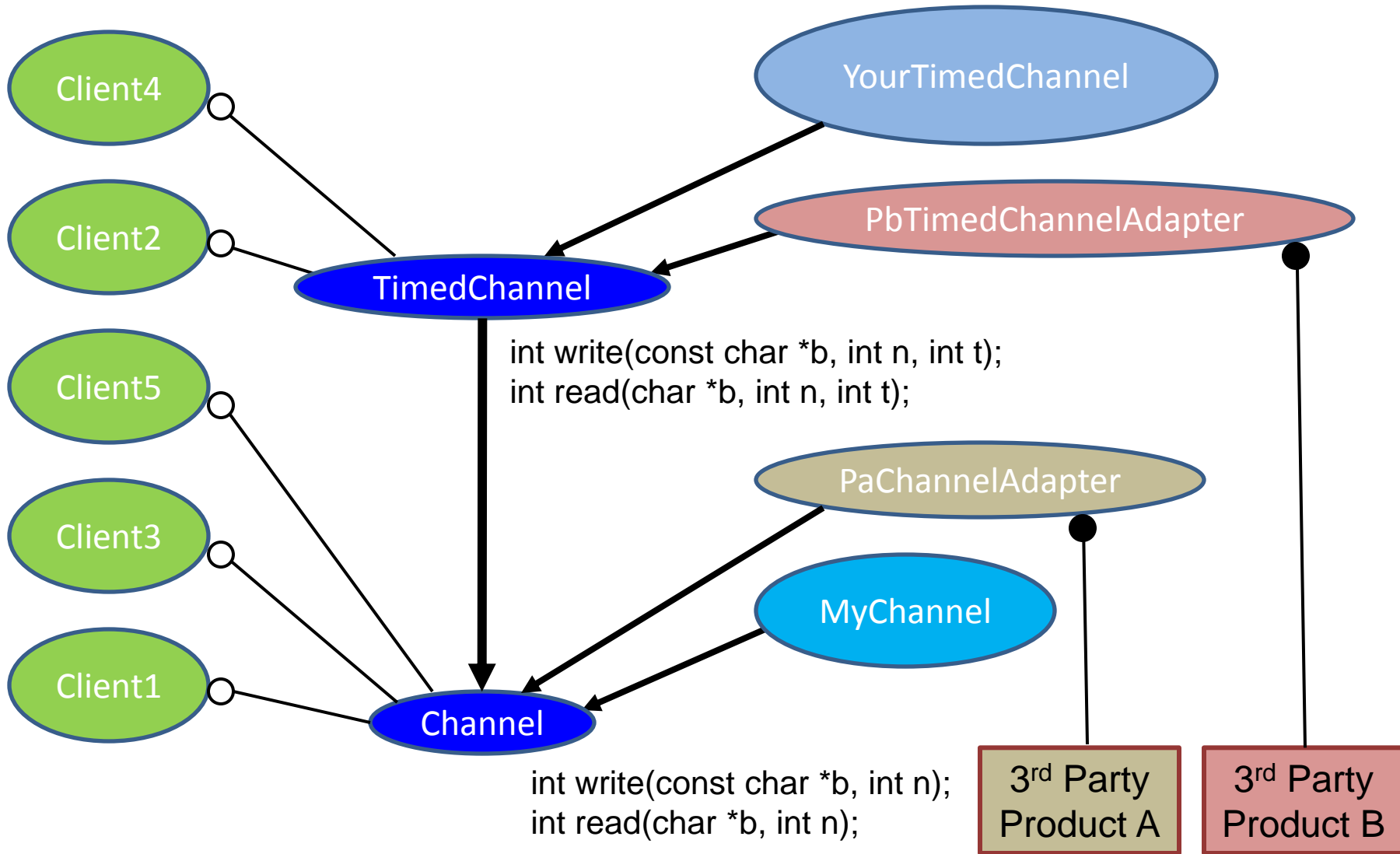
4. Proper Inheritance

Using Interface Inheritance Effectively



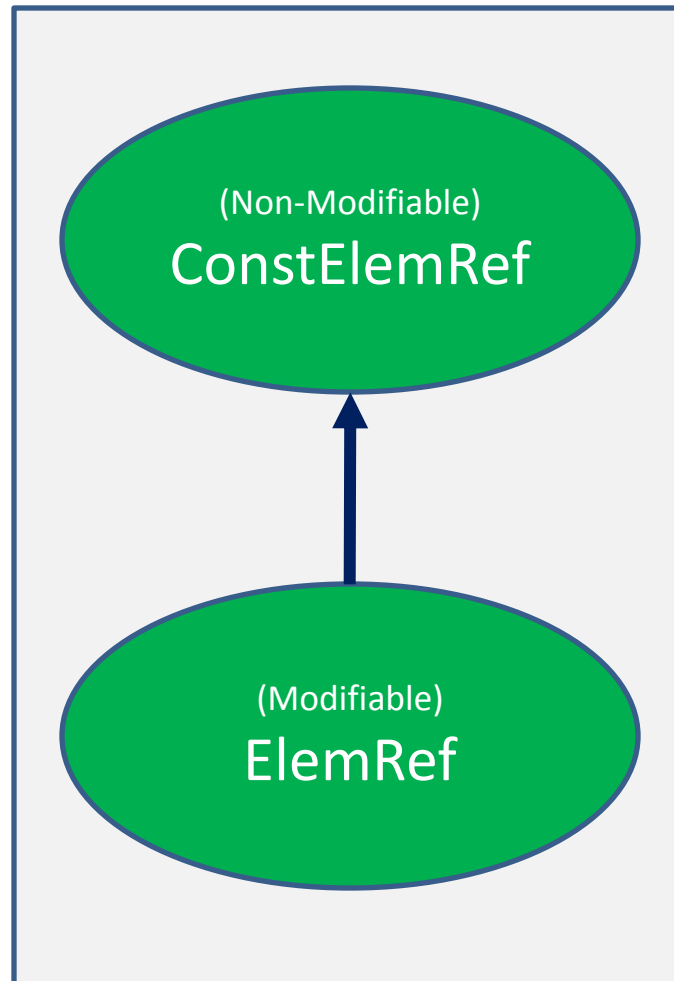
4. Proper Inheritance

Using Interface Inheritance Effectively



4. Proper Inheritance

Using Structural Inheritance Effectively



4. Proper Inheritance

Using Structural Inheritance Effectively

(Non-Modifiable)
ConstElemRef

elem() ;

```
template <class TYPE>
class ConstElemRef<TYPE> {
    const TYPE *d_elem_p;

    friend class ElemRef<TYPE>; // ← Note friendship

private: // Not Implemented. TBD
    ConstElemRef& operator=(Const ConstElemRef&);

public:
    // CREATORS
    ConstElemRef(const TYPE *elem);
    ConstElemRef(const ConstElemRef& ref);
    ~ConstElemRef();

    // ACCESSORS
    const TYPE& elem() const;
};
```

4. Proper Inheritance

Using Structural Inheritance Effectively

(Non-Modifiable)
ConstElemRef

elem() ;

```
template <class TYPE>
class ConstElemRef<TYPE> {
    const TYPE *d_elem_p;

    friend class ElemRef<TYPE>; // ← Note friendship

private: // Not Implemented. TBD
    ConstElemRef& operator=(Const ConstElemRef&);

public:
    // CREATORS
    ConstElemRef(const TYPE *elem);
    ConstElemRef(const ConstElemRef& ref);
    ~ConstElemRef();

    // ACCESSORS
    const TYPE& elem() const;
};
```

Single Pointer
Data Member

4. Proper Inheritance

Using Structural Inheritance Effectively

(Non-Modifiable)
ConstElemRef

elem() ;

```
template <class TYPE>
class ConstElemRef<TYPE> {
    const TYPE *d_elem_p;

    friend class ElemRef<TYPE>; // ← Note friendship

private: // Not Implemented. TBD
    ConstElemRef& operator=(Const ConstElemRef&);

public:
    // CREATORS
    ConstElemRef(const TYPE *elem);
    ConstElemRef(const ConstElemRef& ref);
    ~ConstElemRef();

    // ACCESSORS
    const TYPE& elem() const;
};
```

Derived Class
Declared Friend

← Note friendship

TBD

4. Proper Inheritance

Using Structural Inheritance Effectively

(Non-Modifiable)
ConstElemRef

elem() ;

```
template <class TYPE>
class ConstElemRef<TYPE> {
    const TYPE *d_elem_p;

    friend class ElemRef<TYPE>; // Note friendship

private: // Not Implemented. TBD
    ConstElemRef& operator=(Const ConstElemRef&);

public:
    // CREATORS
    ConstElemRef(const TYPE *elem);
    ConstElemRef(const ConstElemRef& ref);
    ~ConstElemRef();

    // ACCESSORS
    const TYPE& elem() const;
};
```

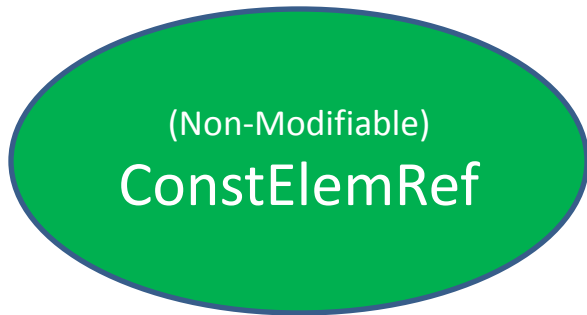
Copy Assignment
Not Implemented

Note friendship

TBD

4. Proper Inheritance

Using Structural Inheritance Effectively



elem() ;



```
template <class TYPE>
class ConstElemRef<TYPE> {
    const TYPE *d_elem_p;

    friend class ElemRef<TYPE>; // ← Note friendship

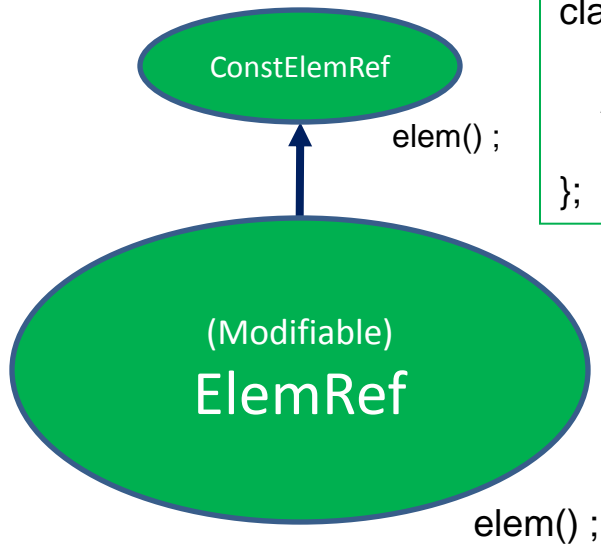
private: // Not Implemented. TBD
    ConstElemRef& operator=(Const ConstElemRef&);

public:
    // CREATORS
    ConstElemRef(const TYPE *elem);
    ConstElemRef(const ConstElemRef& ref);
    ~ConstElemRef();

    // ACCESSORS
    const TYPE& elem() const;
};
```

4. Proper Inheritance

Using Structural Inheritance Effectively

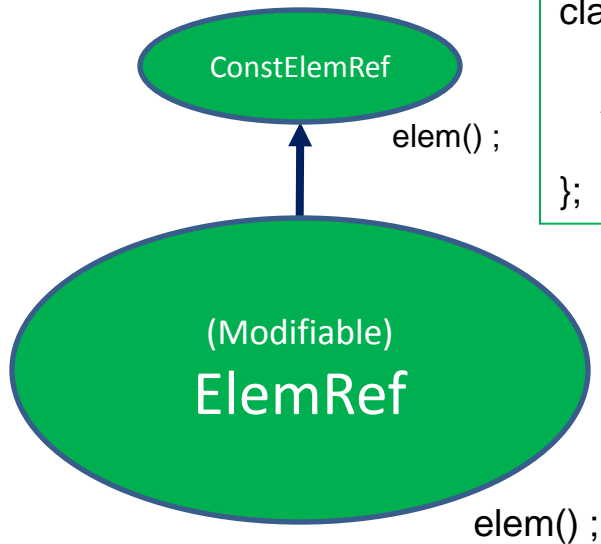


```
class ConstElemRef<TYPE> {  
    const TYPE *d_elem_p;  
    // ...  
    const TYPE& elem() const;  
};
```

```
template <class TYPE>  
class ElemRef<TYPE> : public ConstElemRef<TYPE> {  
public:  
    // CREATORS  
    ElemRef(TYPE *elem);  
    ElemRef(const ElemRef& ref);  
    ~ElemRef();  
  
    // MANIPULATORS  
    ElemRef& operator=(const ElemRef&); // Fine. TBD  
  
    // ACCESSORS  
    TYPE& elem() const;  
};
```

4. Proper Inheritance

Using Structural Inheritance Effectively



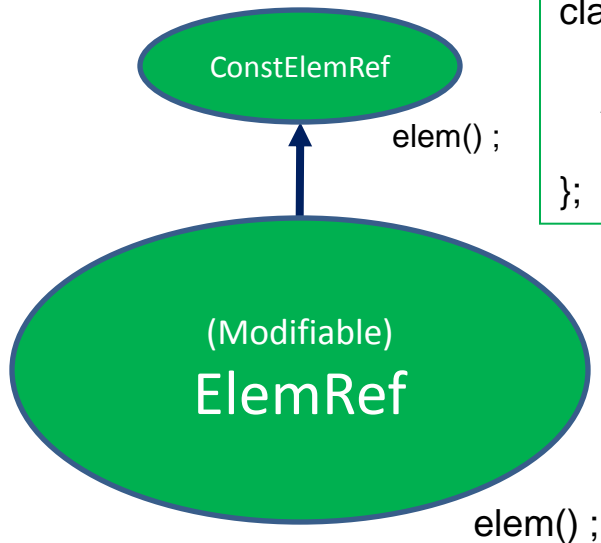
```
class ConstElemRef<TYPE> {  
    const TYPE *d_elem_p;  
    // ...  
    const TYPE& elem() const;  
};
```

Public
Structural
Inheritance

```
template <class TYPE>  
class ElemRef<TYPE> : public ConstElemRef<TYPE> {  
public:  
    // CREATORS  
    ElemRef(TYPE *elem);  
    ElemRef(const ElemRef& ref);  
    ~ElemRef();  
  
    // MANIPULATORS  
    ElemRef& operator=(const ElemRef&); // Fine. TBD  
  
    // ACCESSORS  
    TYPE& elem() const;  
};
```

4. Proper Inheritance

Using Structural Inheritance Effectively



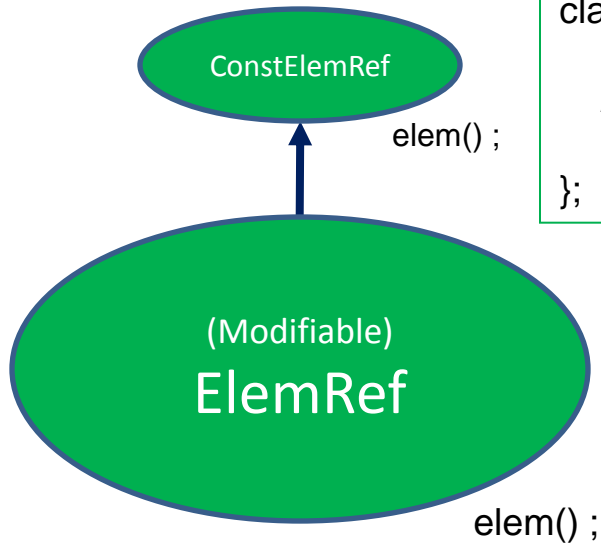
```
class ConstElemRef<TYPE> {  
    const TYPE *d_elem_p;  
    // ...  
    const TYPE& elem() const;  
};
```

No Additional
Member Data

```
template <class TYPE>  
class ElemRef<TYPE> : public ConstElemRef<TYPE> {  
public:  
    // CREATORS  
    ElemRef(TYPE *elem);  
    ElemRef(const ElemRef& ref);  
    ~ElemRef();  
  
    // MANIPULATORS  
    ElemRef& operator=(const ElemRef&); // Fine. TBD  
  
    // ACCESSORS  
    TYPE& elem() const;  
};
```

4. Proper Inheritance

Using Structural Inheritance Effectively



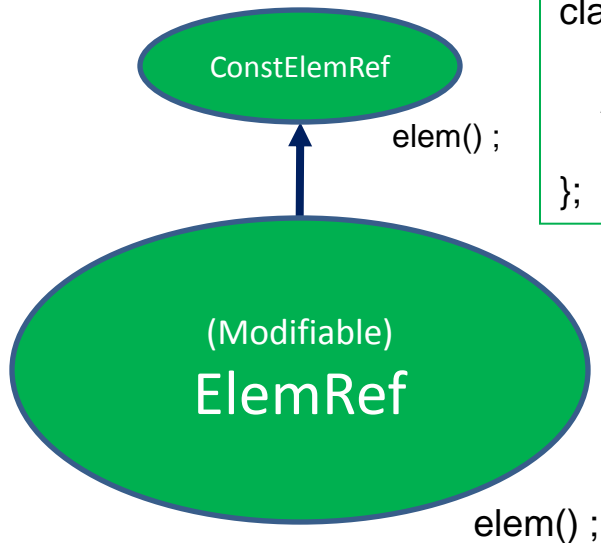
```
class ConstElemRef<TYPE> {  
    const TYPE *d_elem_p;  
    // ...  
    const TYPE& elem() const;  
};
```

```
template <class TYPE>  
class ElemRef<TYPE> : public ConstElemRef<TYPE> {  
public:  
    // CREATORS  
    ElemRef(TYPE *elem);  
    ElemRef(const ElemRef& ref);  
    ~ElemRef();  
  
    // MANIPULATORS  
    ElemRef& operator=(const ElemRef&); // Fine. TBD  
  
    // ACCESSORS  
    TYPE& elem() const;  
};
```

Copy Assignment
Implemented

4. Proper Inheritance

Using Structural Inheritance Effectively



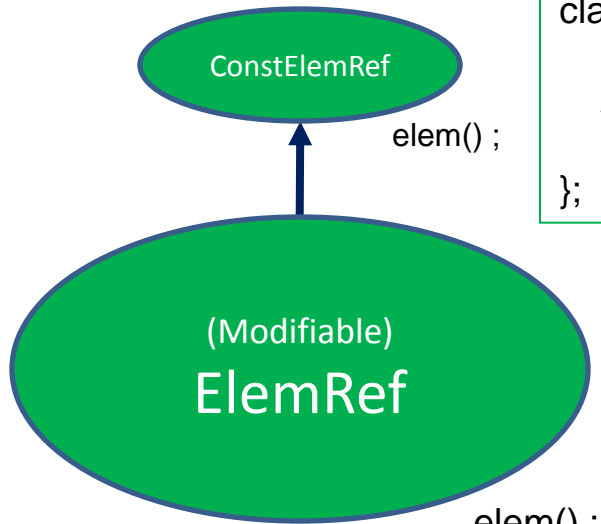
```
class ConstElemRef<TYPE> {  
    const TYPE *d_elem_p;  
    // ...  
    const TYPE& elem() const;  
};
```

```
template <class TYPE>  
class ElemRef<TYPE> : public ConstElemRef<TYPE> {  
public:  
    // CREATORS  
    ElemRef(TYPE *elem);  
    ElemRef(const ElemRef& ref);  
    ~ElemRef();  
  
    // MANIPULATORS  
    ElemRef& operator=(const ElemRef&); // Fine. TBD  
  
    // ACCESSORS  
    TYPE& elem() const;  
};
```

Read-Write
Access

4. Proper Inheritance

Using Structural Inheritance Effectively



```
class ConstElemRef<TYPE> {  
    const TYPE *d_elem_p;  
    // ...  
    const TYPE& elem() const;  
};
```

```
template <class T>  
class ElemRef<T> {  
public:  
    // CREATORS  
    ElemRef(T& t): d_elem_p(&t) {}  
    ElemRef(const T& t): d_elem_p(&t) {}  
    ~ElemRef();  
  
    // MANIPULATOR  
    ElemRef& operator=(const T& t);  
    ElemRef& operator=(const ElemRef& e);  
  
    // ACCESSORS  
    T& elem() const;  
};
```

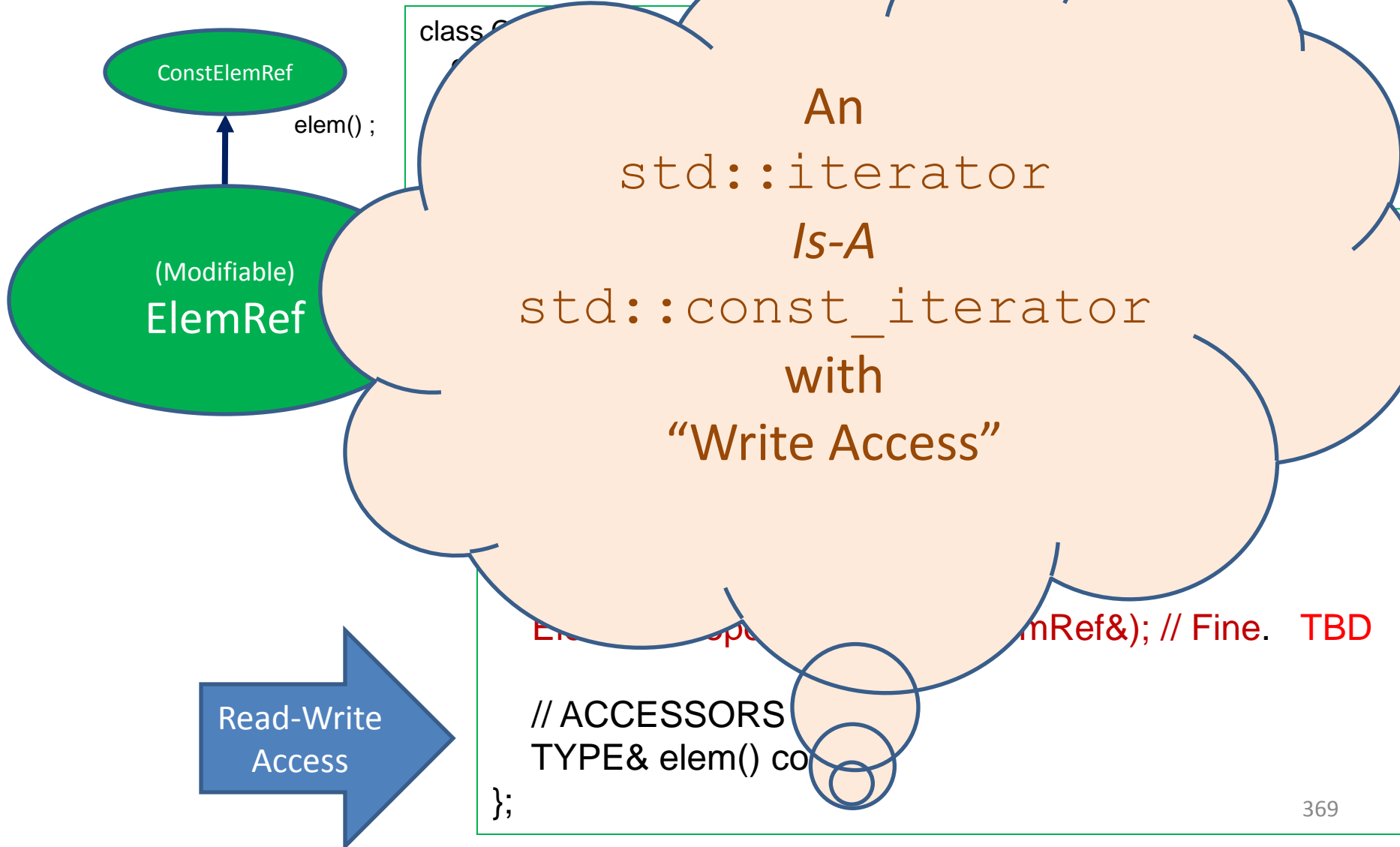
An
ElemRef
Is-A
ConstElemRef
with
"Write Access"

Read-Write
Access

ElemRef& operator=(const ElemRef& e); Fine. TBD

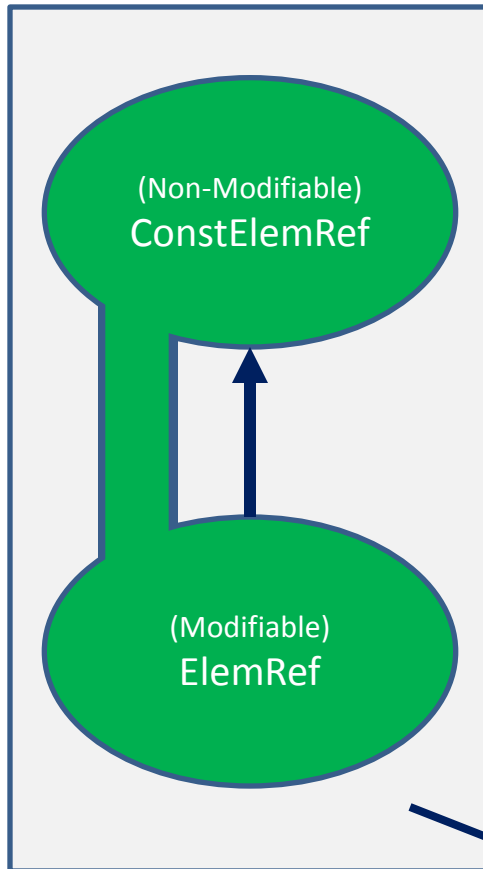
4. Proper Inheritance

Using Structural Inheritance Effectively



4. Proper Inheritance

Using Structural Inheritance Effectively



```
const TYPE& ConstElemRef::elem() const
{
    return *d_elem_p;
}
```

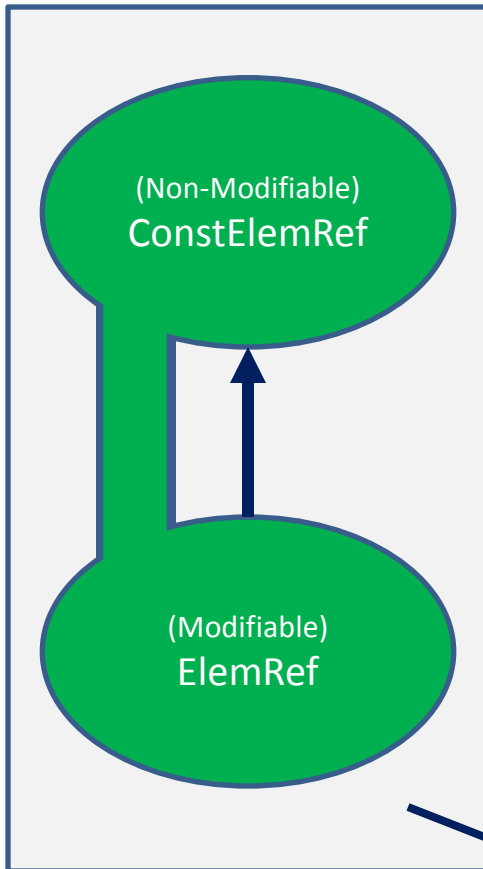
```
TYPE& ElemRef::elem() const
{
    return *const_cast<TYPE *>(d_elem_p);
}

// Note use of friendship as well.
```

Note: same component due to friendship.

4. Proper Inheritance

Using Structural Inheritance Effectively



```
const TYPE& ConstElemRef::elem() const  
{  
    return *d_elem_p;  
}
```

Note we are casting-away **const**

```
TYPE& ElemRef::elem() const  
{  
    return *const_cast<TYPE*>(d_elem_p);  
}
```

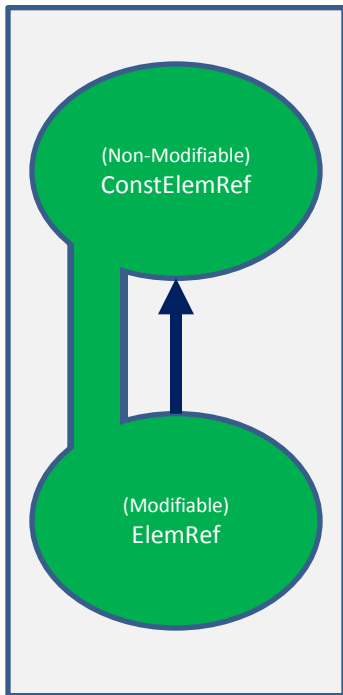
// Note use of friendship as well.

Note: same component due to friendship.

4. Proper Inheritance

Using Structural Inheritance Effectively

Be especially careful to ensure `const`-correctness when `const`-casting is involved.

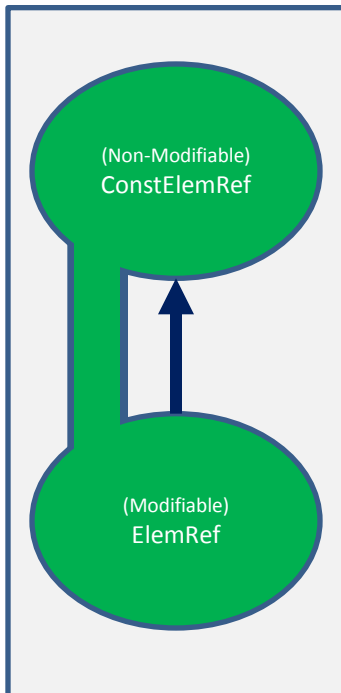


4. Proper Inheritance

Using Structural Inheritance Effectively

Be especially careful to ensure `const`-correctness when `const`-casting is involved.

```
void g(ConstElemRef *cer1, const ConstElemRef& cer2)
{
    *cer1 = cer2; // Enable const-correctness violation due to slicing.
} // Assumes copy assignment is enabled on the ConstElemRef base class.
```

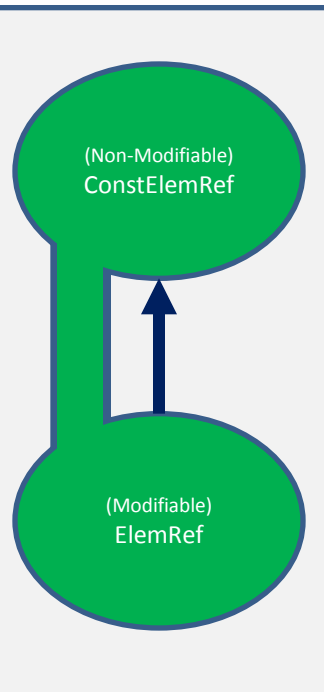


4. Proper Inheritance

Using Structural Inheritance Effectively

Be especially careful to ensure `const`-correctness when `const`-casting is involved.

```
void g(ConstElemRef *cer1, const ConstElemRef& cer2)
{
    *cer1 = cer2; // Enable const-correctness violation due to slicing.
} // Assumes copy assignment is enabled on the ConstElemRef base class.
```



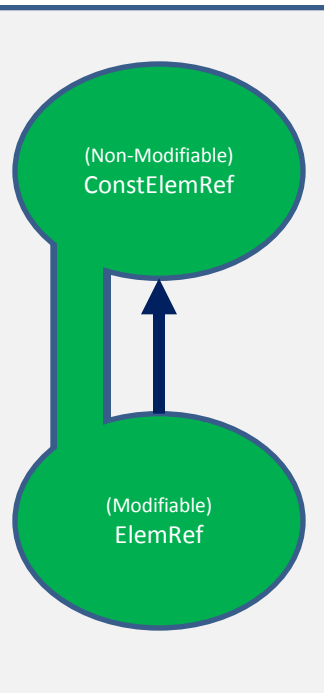
```
template < class TYPE>
void f(const TYPE& constElem)
{
```

4. Proper Inheritance

Using Structural Inheritance Effectively

Be especially careful to ensure `const`-correctness when `const`-casting is involved.

```
void g(ConstElemRef *cer1, const ConstElemRef& cer2)
{
    *cer1 = cer2; // Enable const-correctness violation due to slicing.
} // Assumes copy assignment is enabled on the ConstElemRef base class.
```



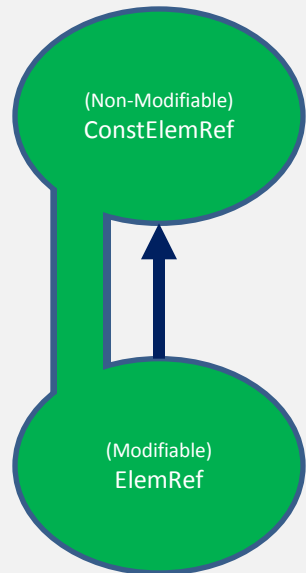
```
template < class TYPE>
void f(const TYPE& constElem)
{
    TYPE dummy;
```

4. Proper Inheritance

Using Structural Inheritance Effectively

Be especially careful to ensure `const`-correctness when `const`-casting is involved.

```
void g(ConstElemRef *cer1, const ConstElemRef& cer2)
{
    *cer1 = cer2; // Enable const-correctness violation due to slicing.
} // Assumes copy assignment is enabled on the ConstElemRef base class.
```



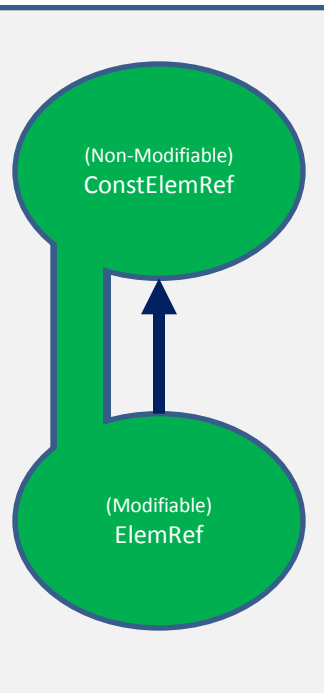
```
template < class TYPE >
void f(const TYPE& constElem)
{
    TYPE dummy;
    ElemRef er(&dummy);
}
```


4. Proper Inheritance

Using Structural Inheritance Effectively

Be especially careful to ensure `const`-correctness when `const`-casting is involved.

```
void g(ConstElemRef *cer1, const ConstElemRef& cer2)
{
    *cer1 = cer2; // Enable const-correctness violation due to slicing.
} // Assumes copy assignment is enabled on the ConstElemRef base class.
```



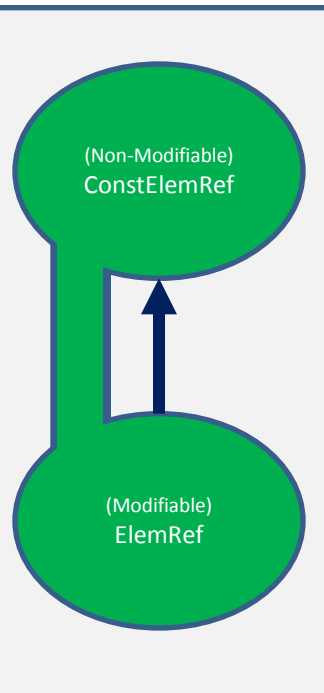
```
template < class TYPE >
void f(const TYPE& constElem)
{
    TYPE dummy;
    ElemRef er(&dummy);
    ConstElemRef cer(&constElem);
}
```

4. Proper Inheritance

Using Structural Inheritance Effectively

Be especially careful to ensure `const`-correctness when `const`-casting is involved.

```
void g(ConstElemRef *cer1, const ConstElemRef& cer2)
{
    *cer1 = cer2; // Enable const-correctness violation due to slicing.
} // Assumes copy assignment is enabled on the ConstElemRef base class.
```



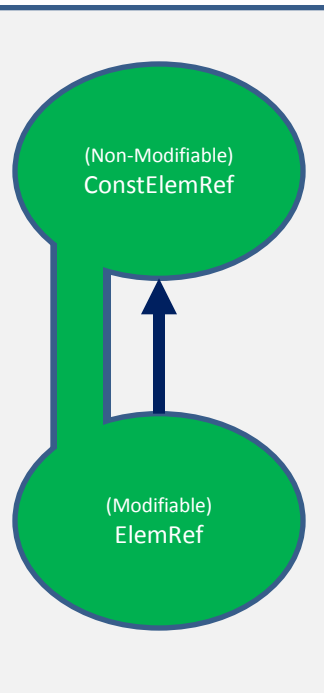
```
template < class TYPE >
void f(const TYPE& constElem)
{
    TYPE dummy;
    ElemRef er(&dummy);
    ConstElemRef cer(&constElem);
    g(&er, cer); // Rebind (modifiable) 'ElemRef' 'er'.
```

4. Proper Inheritance

Using Structural Inheritance Effectively

Be especially careful to ensure `const`-correctness when `const`-casting is involved.

```
void g(ConstElemRef *cer1, const ConstElemRef& cer2)
{
    *cer1 = cer2; // Enable const-correctness violation due to slicing.
} // Assumes copy assignment is enabled on the ConstElemRef base class.
```



```
template < class TYPE >
void f(const TYPE& constElem)
{
    TYPE dummy;
    ElemRef er(&dummy);
    ConstElemRef cer(&constElem);
    g(&er, cer); // Rebind (modifiable) 'ElemRef' 'er'.
    er.elem() = TYPE(); // Clobber 'constElem'.
```

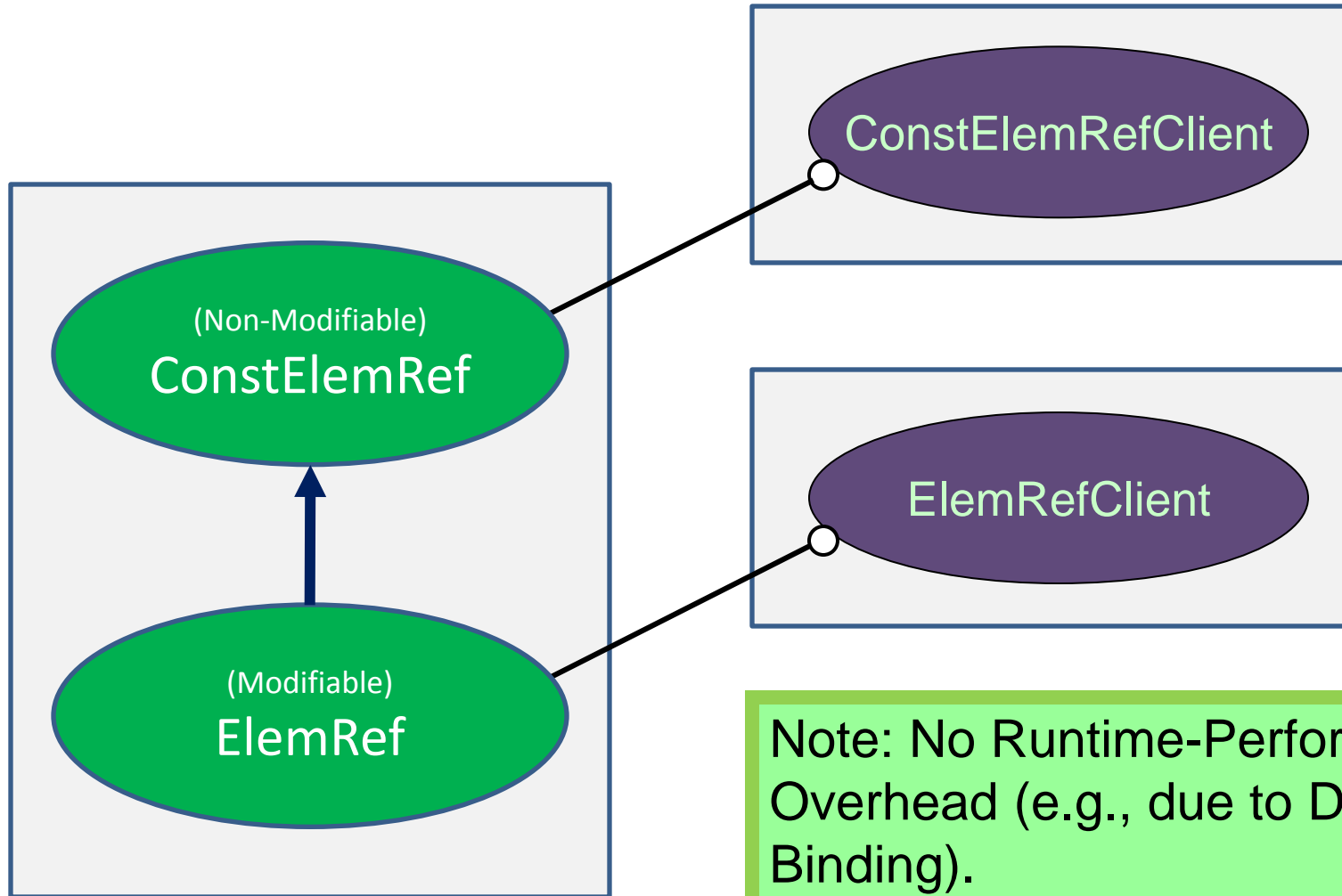
4. Proper Inheritance

Using Structural Inheritance Effectively

The principal client of
Structural Inheritance
is the
PUBLIC CLIENT.

4. Proper Inheritance

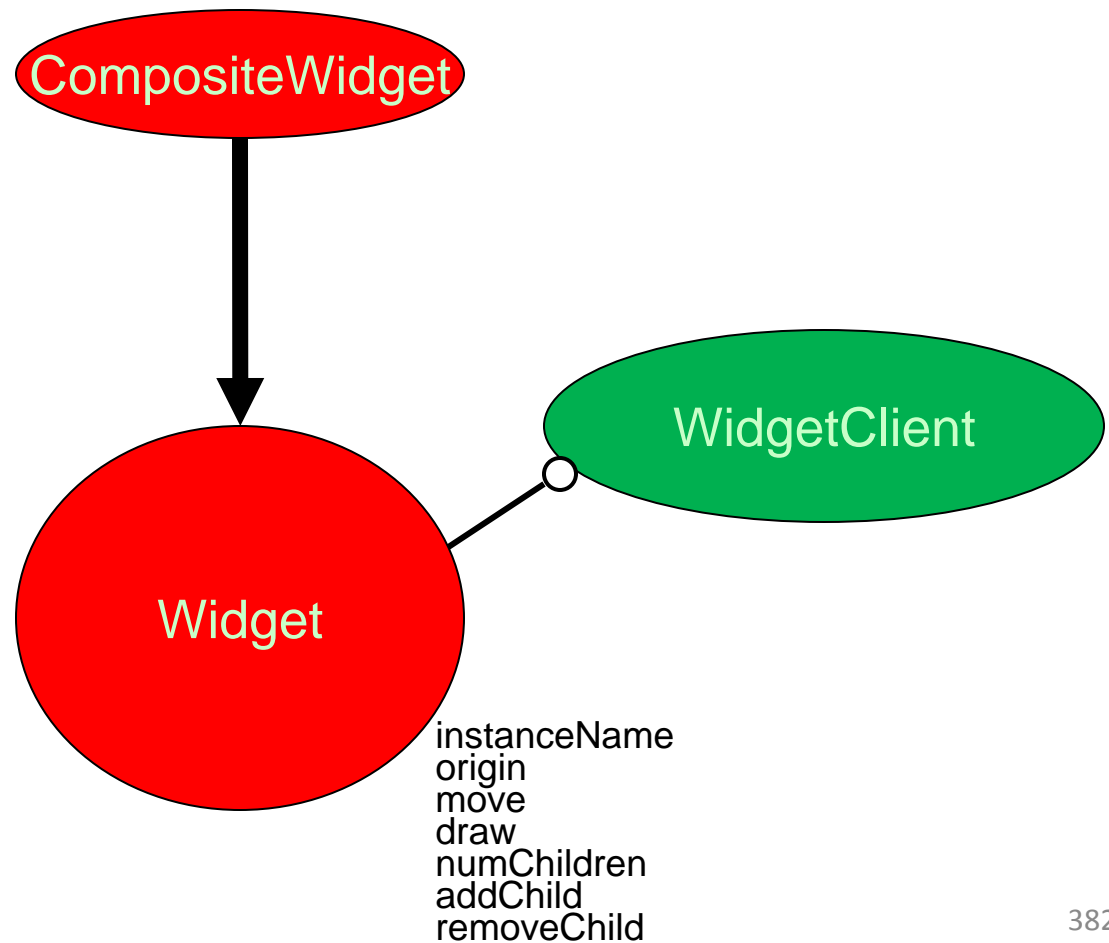
Using Structural Inheritance Effectively



Note: No Runtime-Performance Overhead (e.g., due to Dynamic Binding).

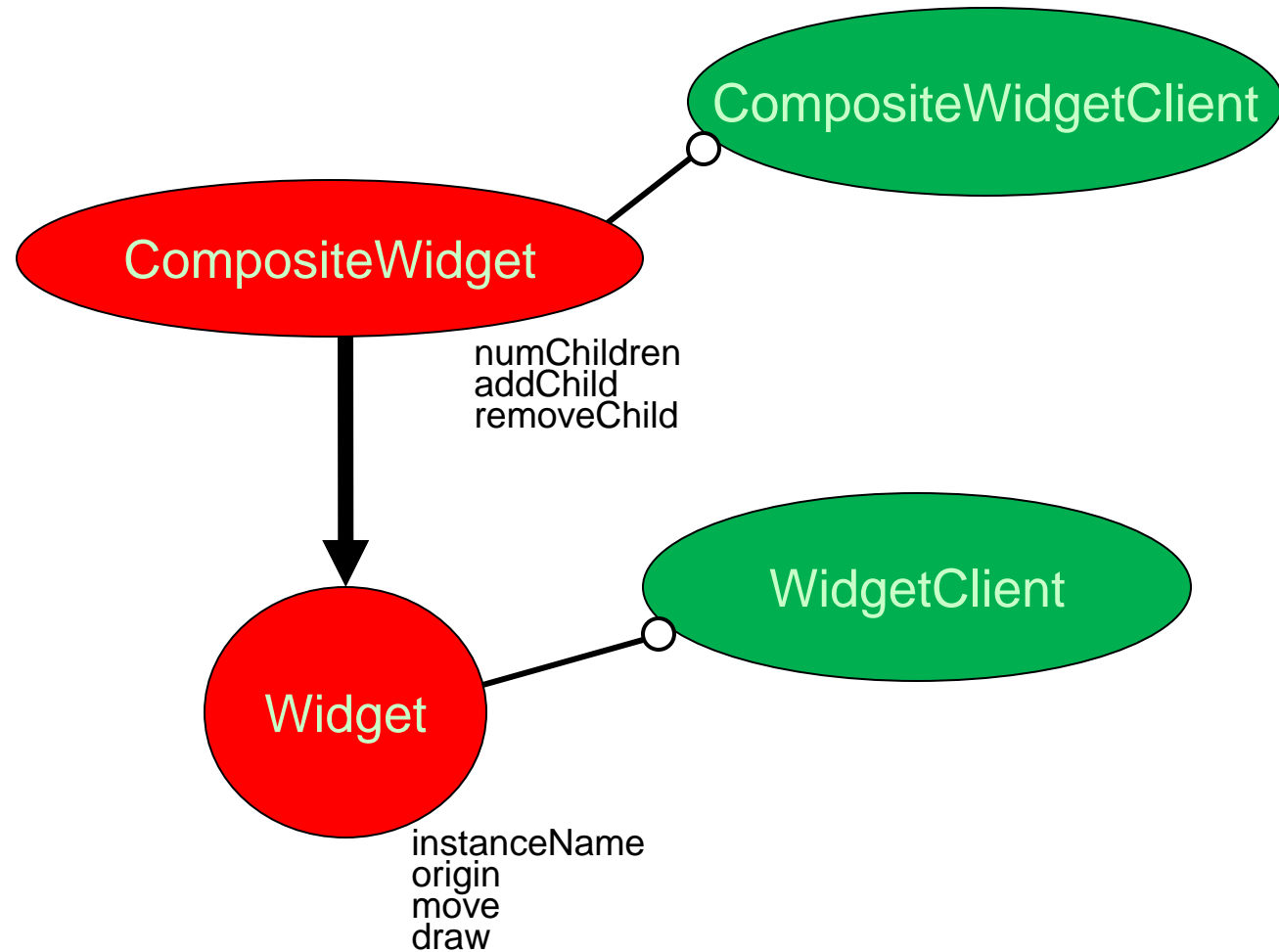
4. Proper Inheritance

Using Implementation Inheritance Effectively



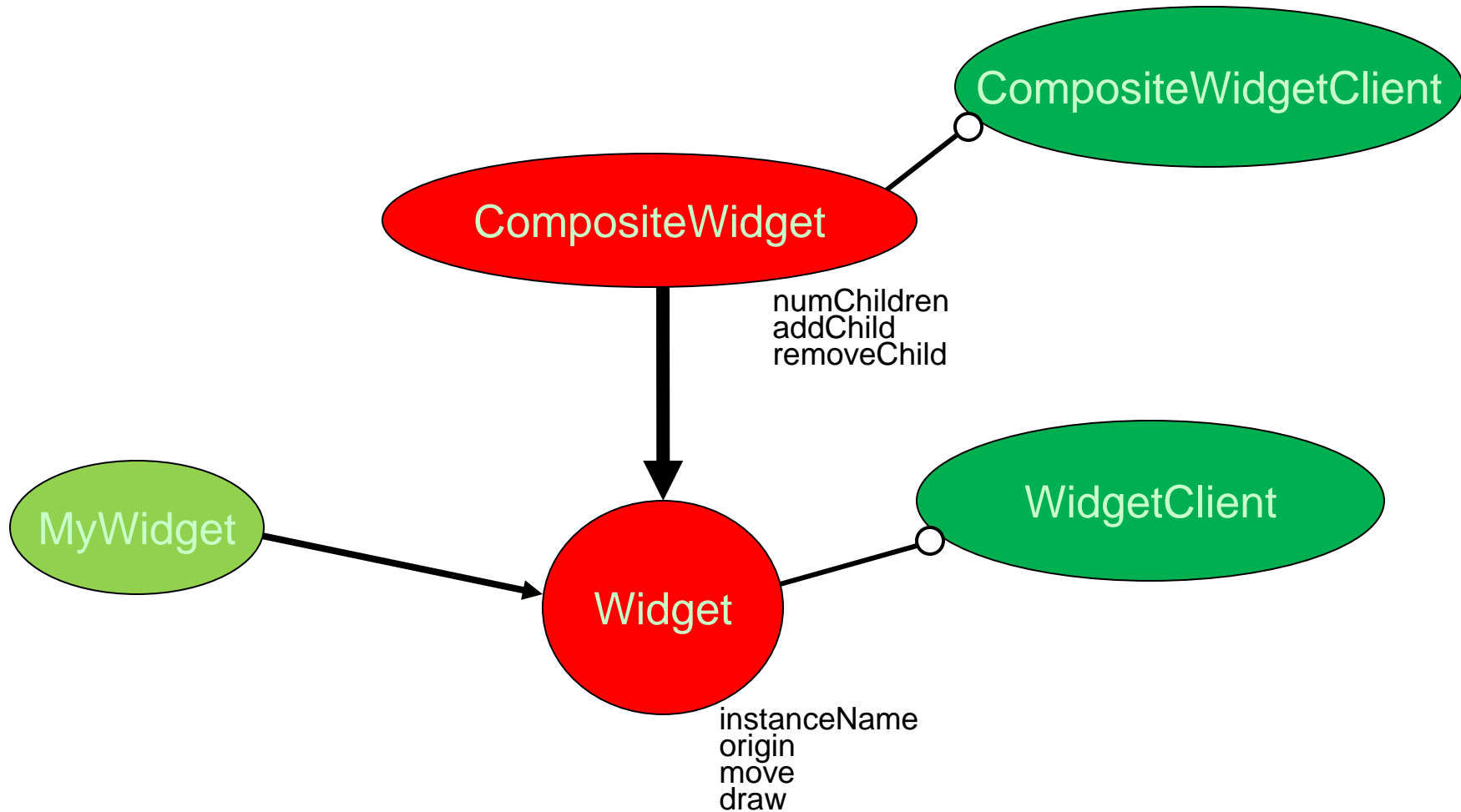
4. Proper Inheritance

Using Implementation Inheritance Effectively



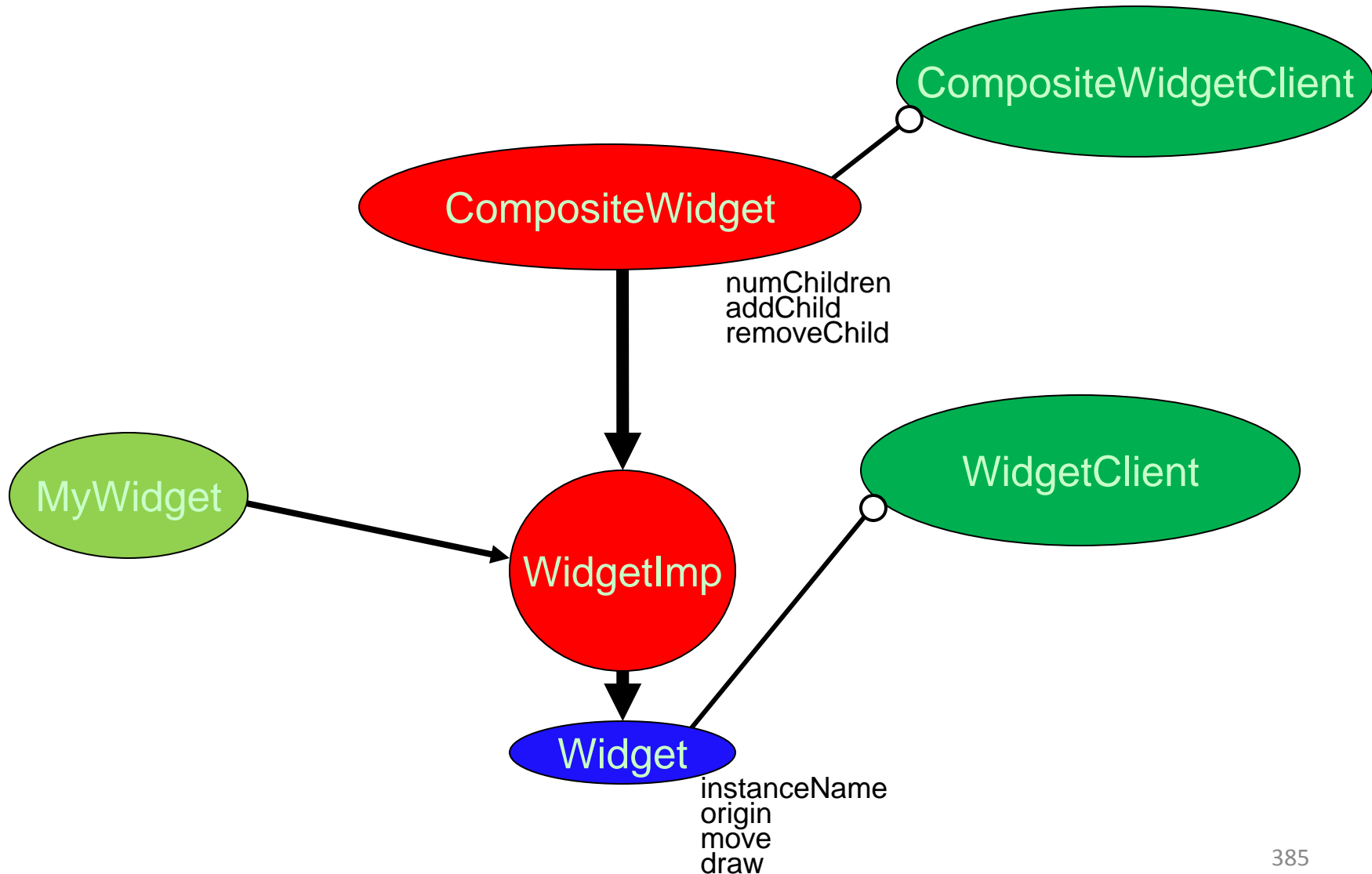
4. Proper Inheritance

Using Implementation Inheritance Effectively



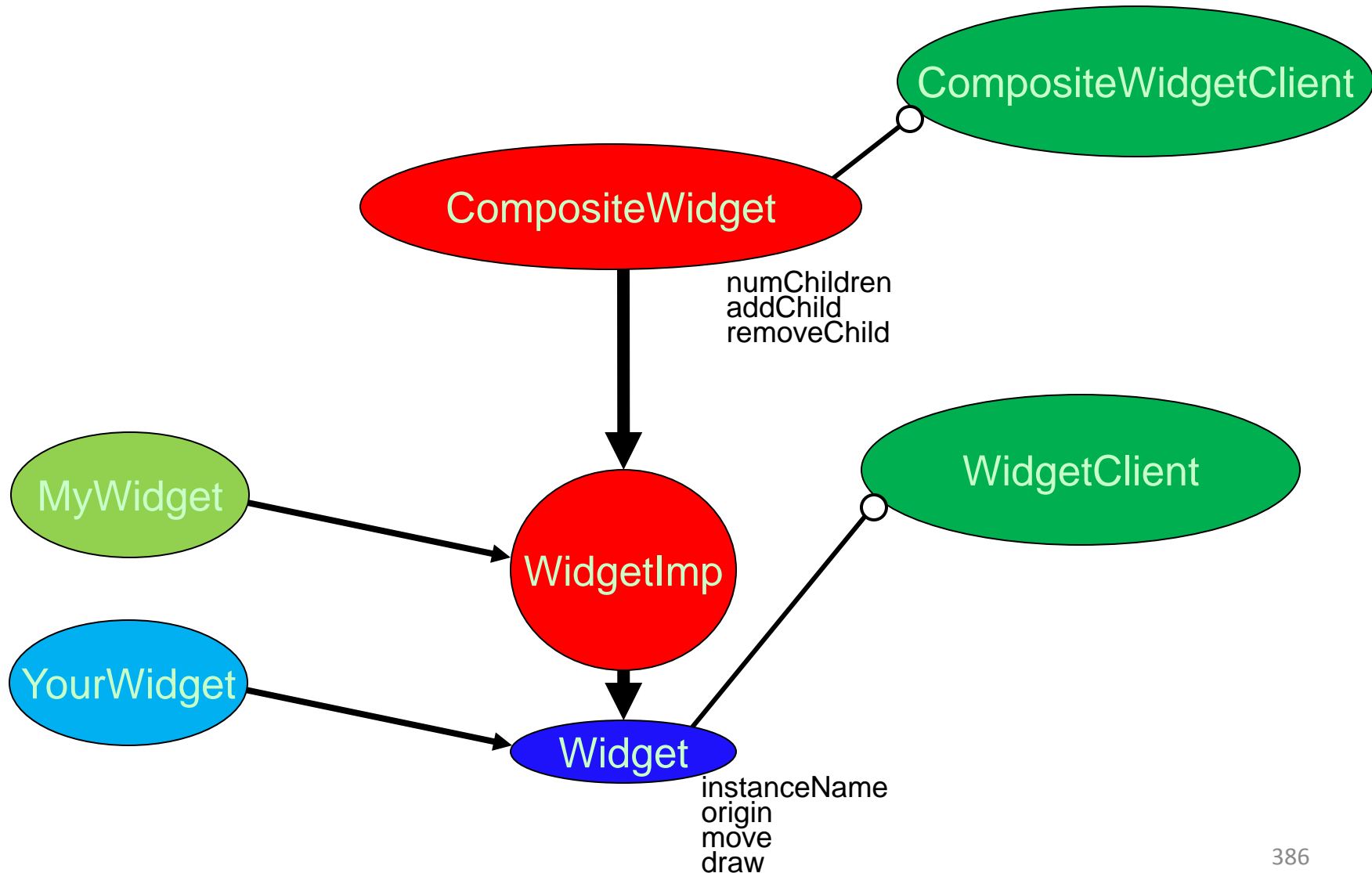
4. Proper Inheritance

Using Implementation Inheritance Effectively



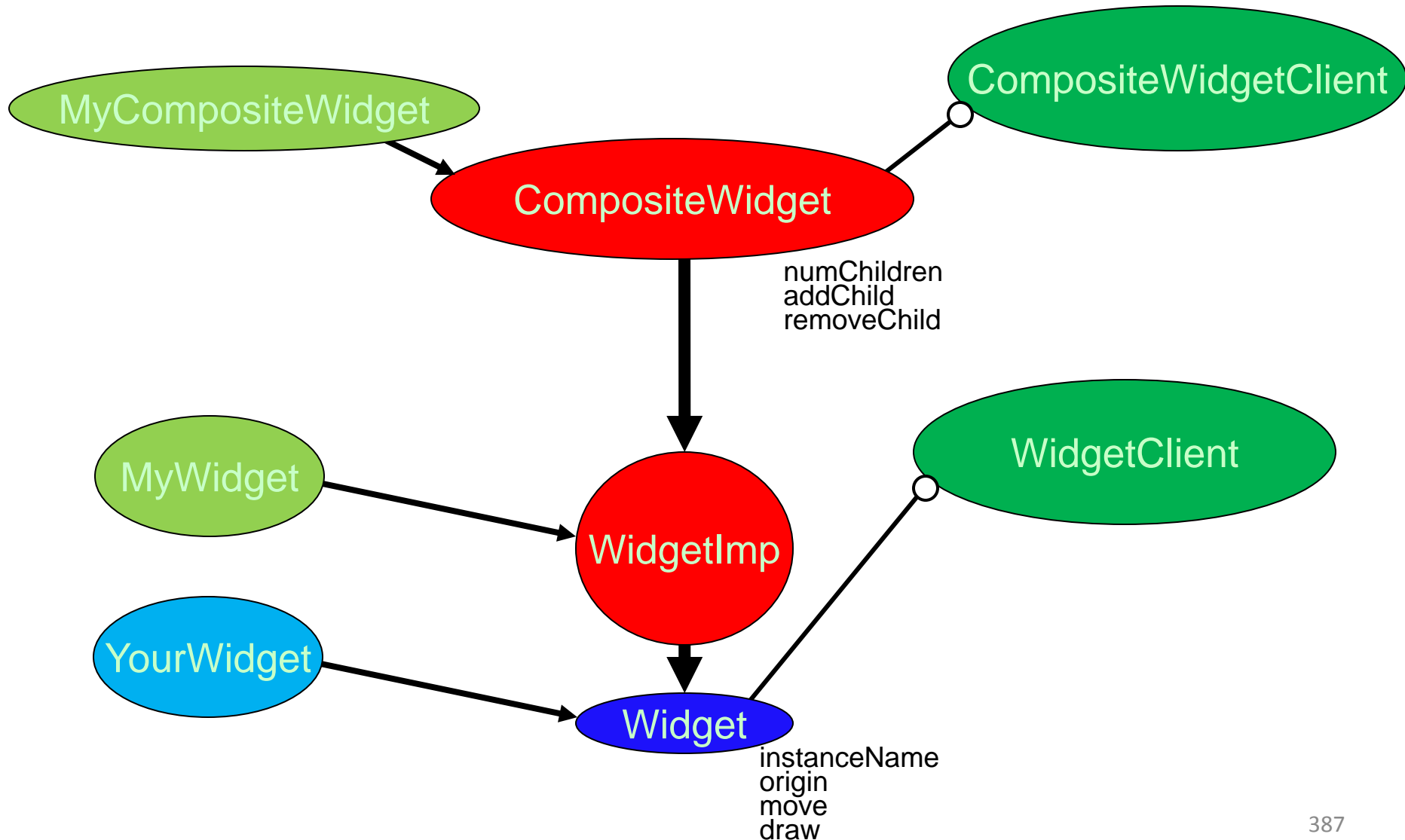
4. Proper Inheritance

Using Implementation Inheritance Effectively



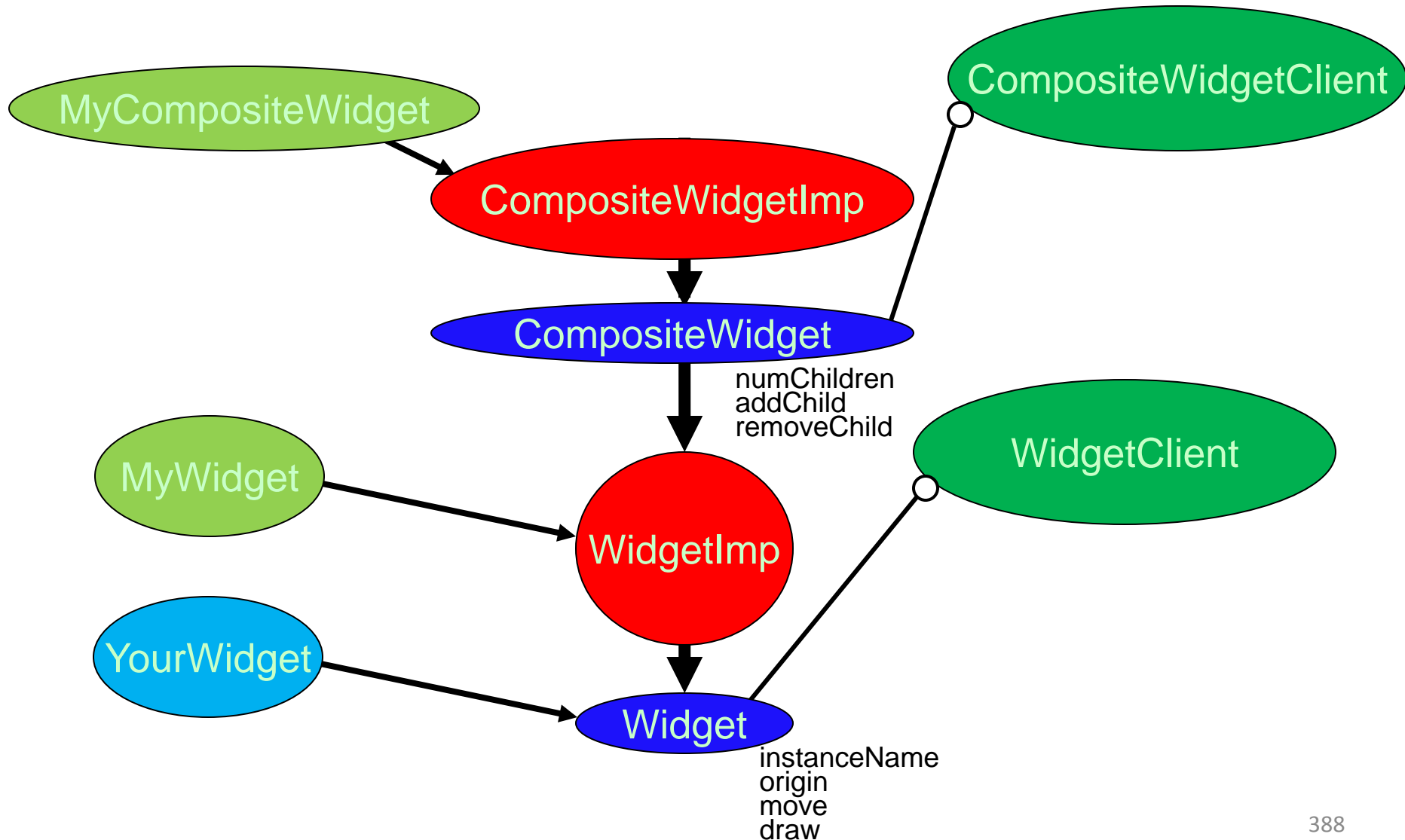
4. Proper Inheritance

Using Implementation Inheritance Effectively



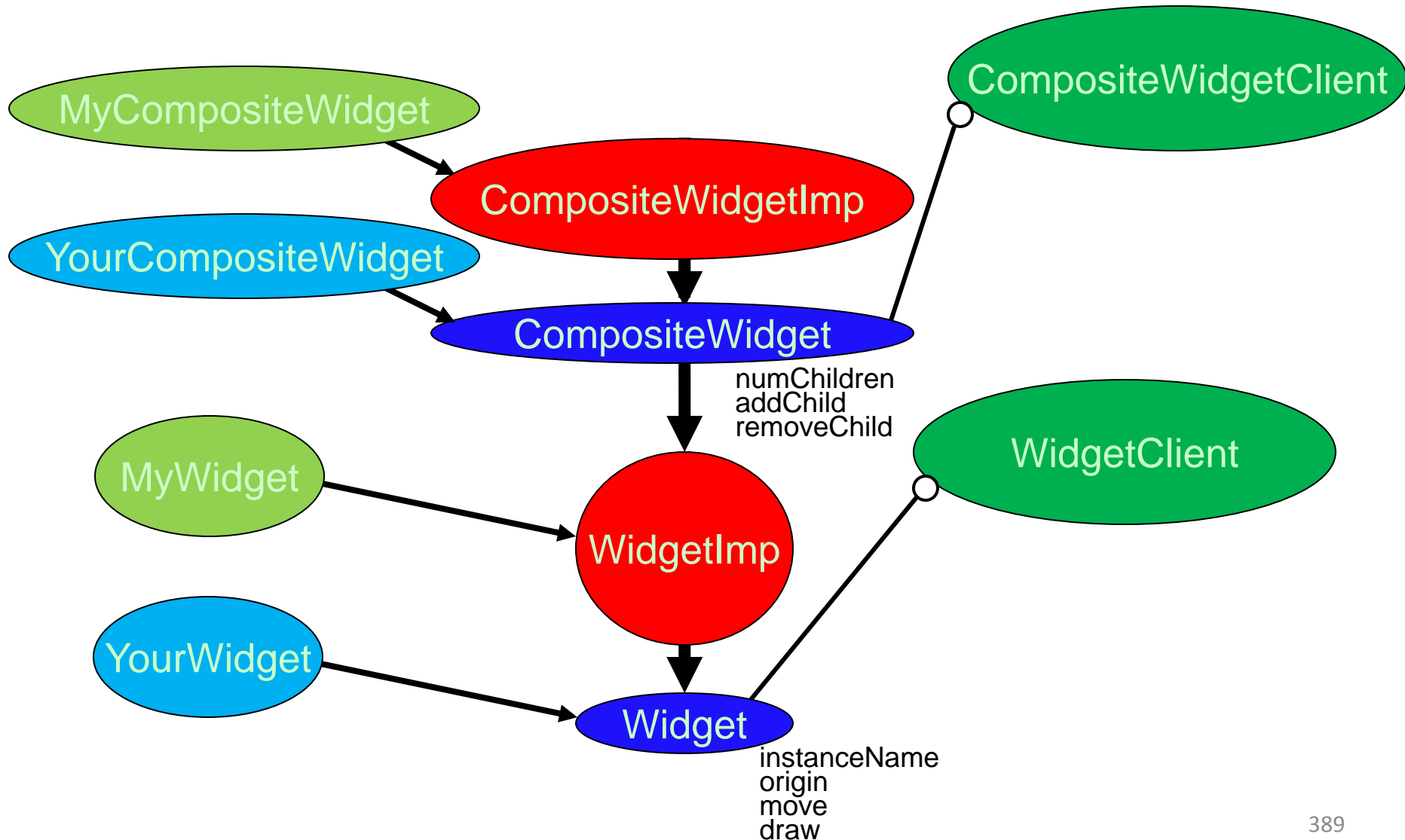
4. Proper Inheritance

Using Implementation Inheritance Effectively



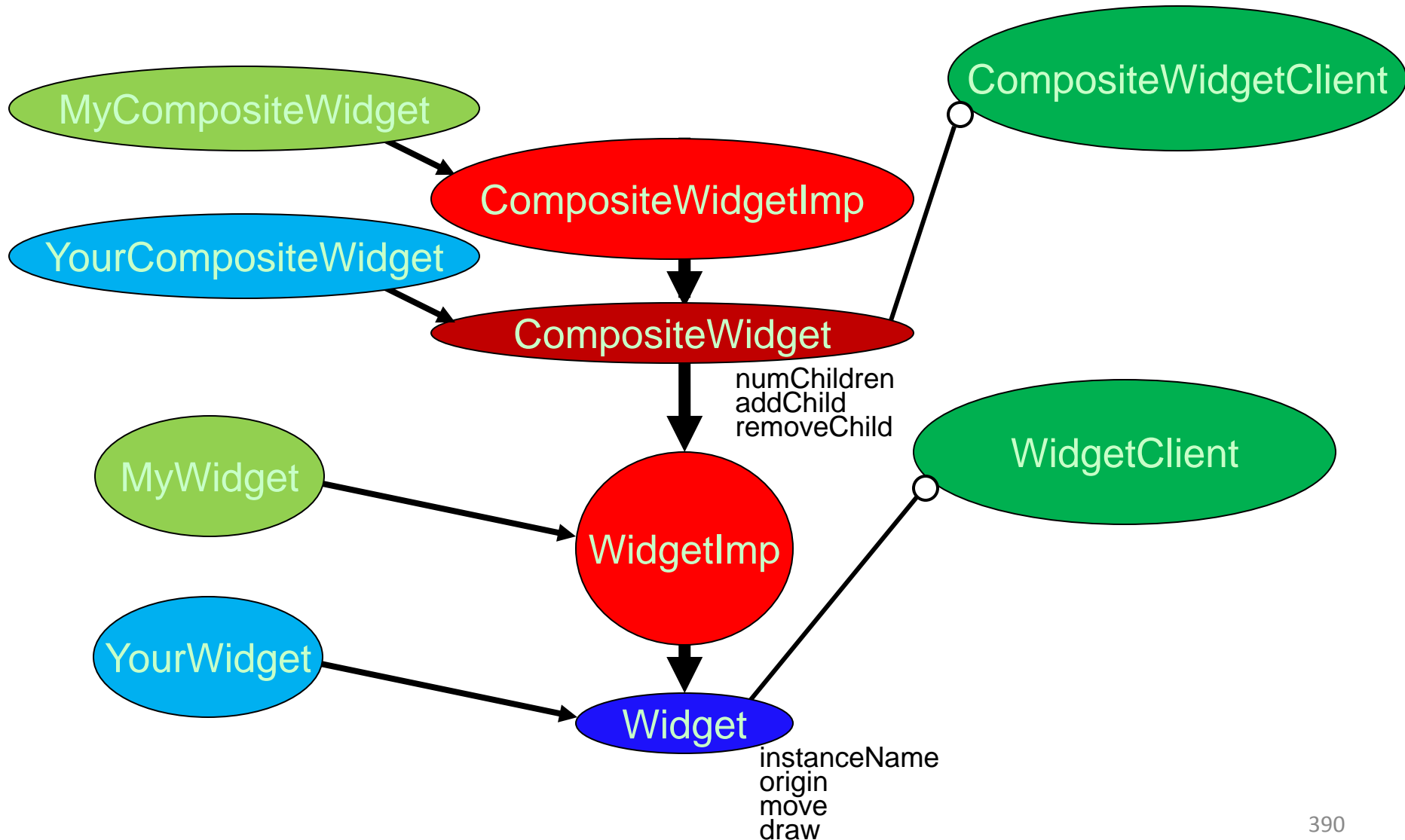
4. Proper Inheritance

Using Implementation Inheritance Effectively



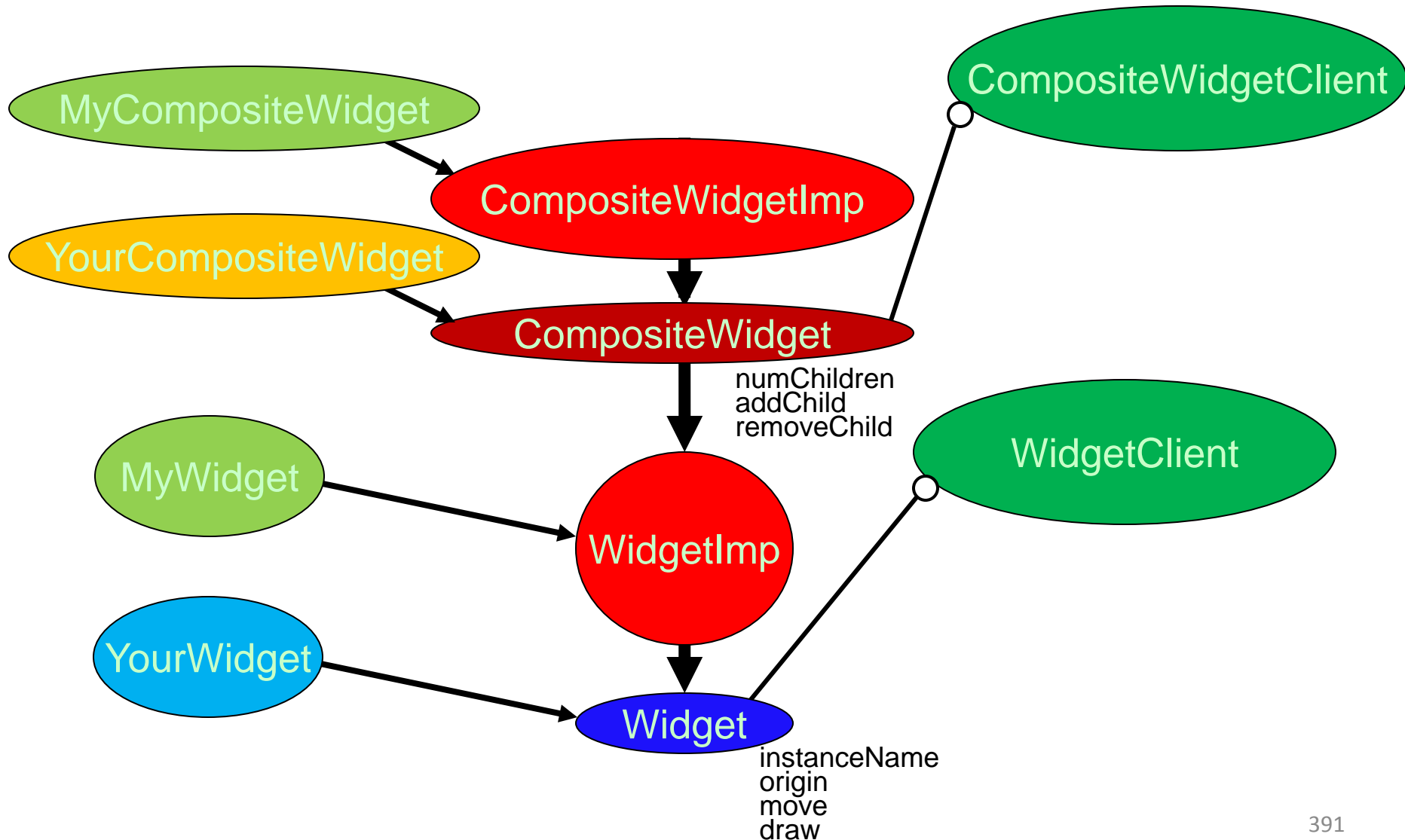
4. Proper Inheritance

Using Implementation Inheritance Effectively



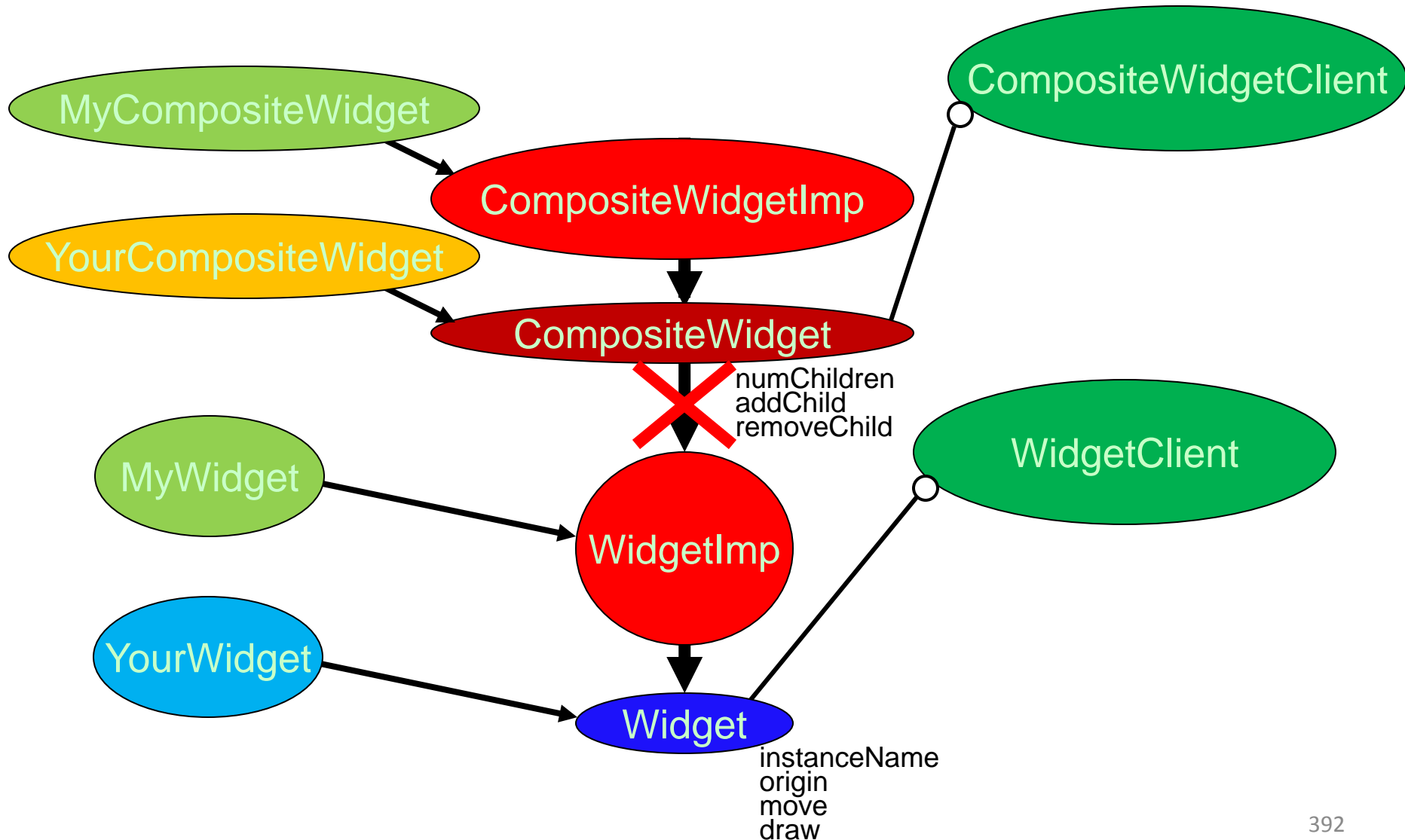
4. Proper Inheritance

Using Implementation Inheritance Effectively



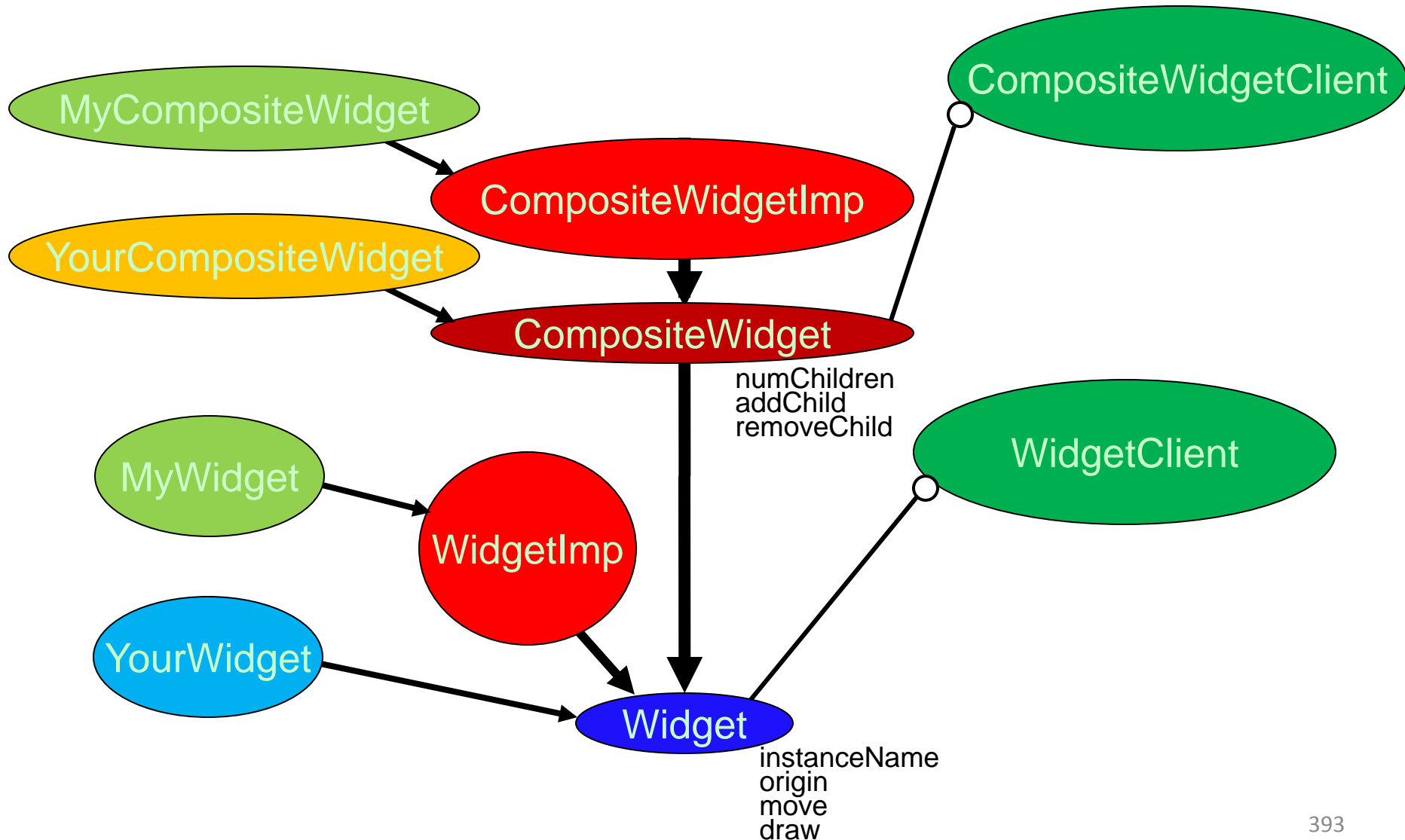
4. Proper Inheritance

Using Implementation Inheritance Effectively



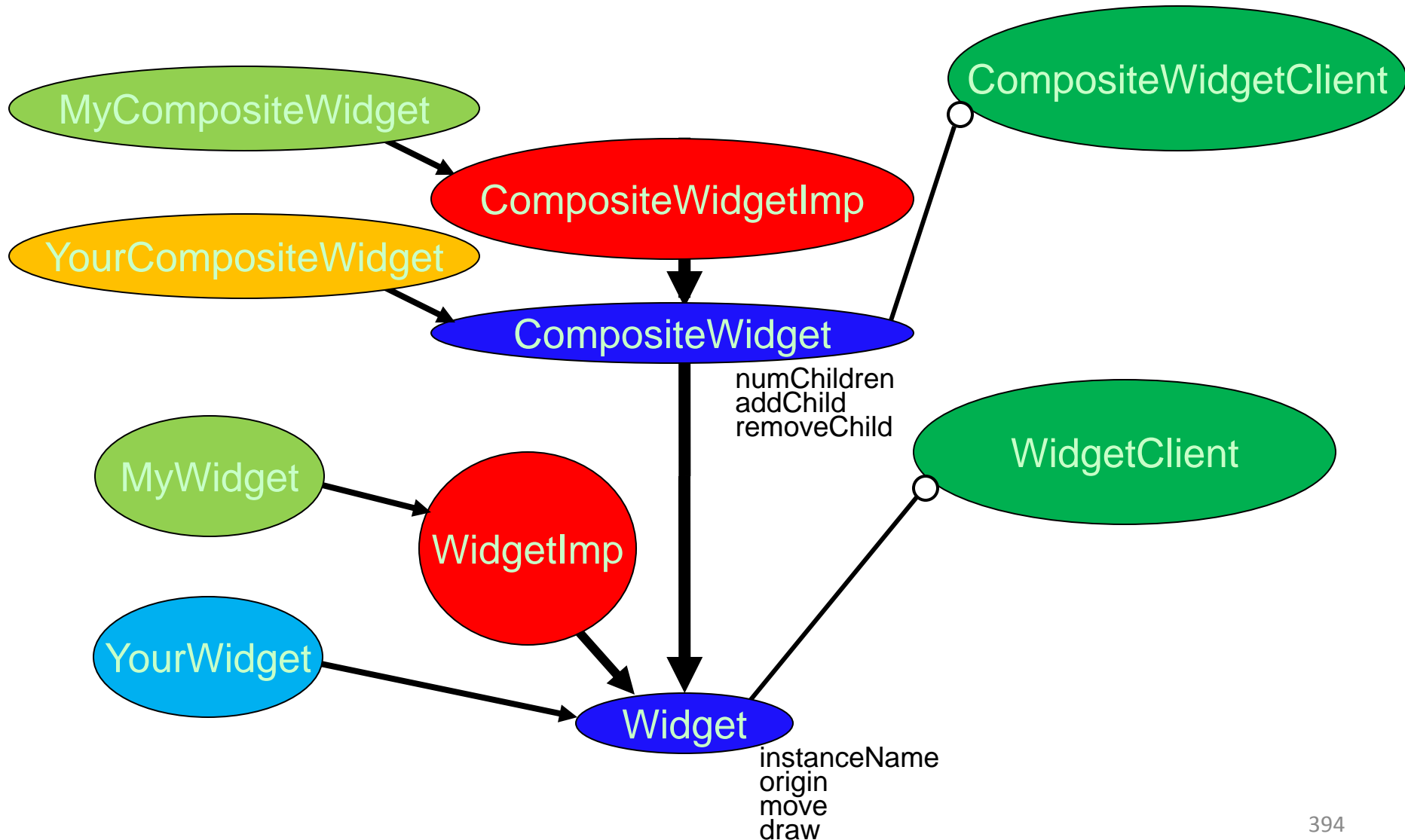
4. Proper Inheritance

Using Implementation Inheritance Effectively



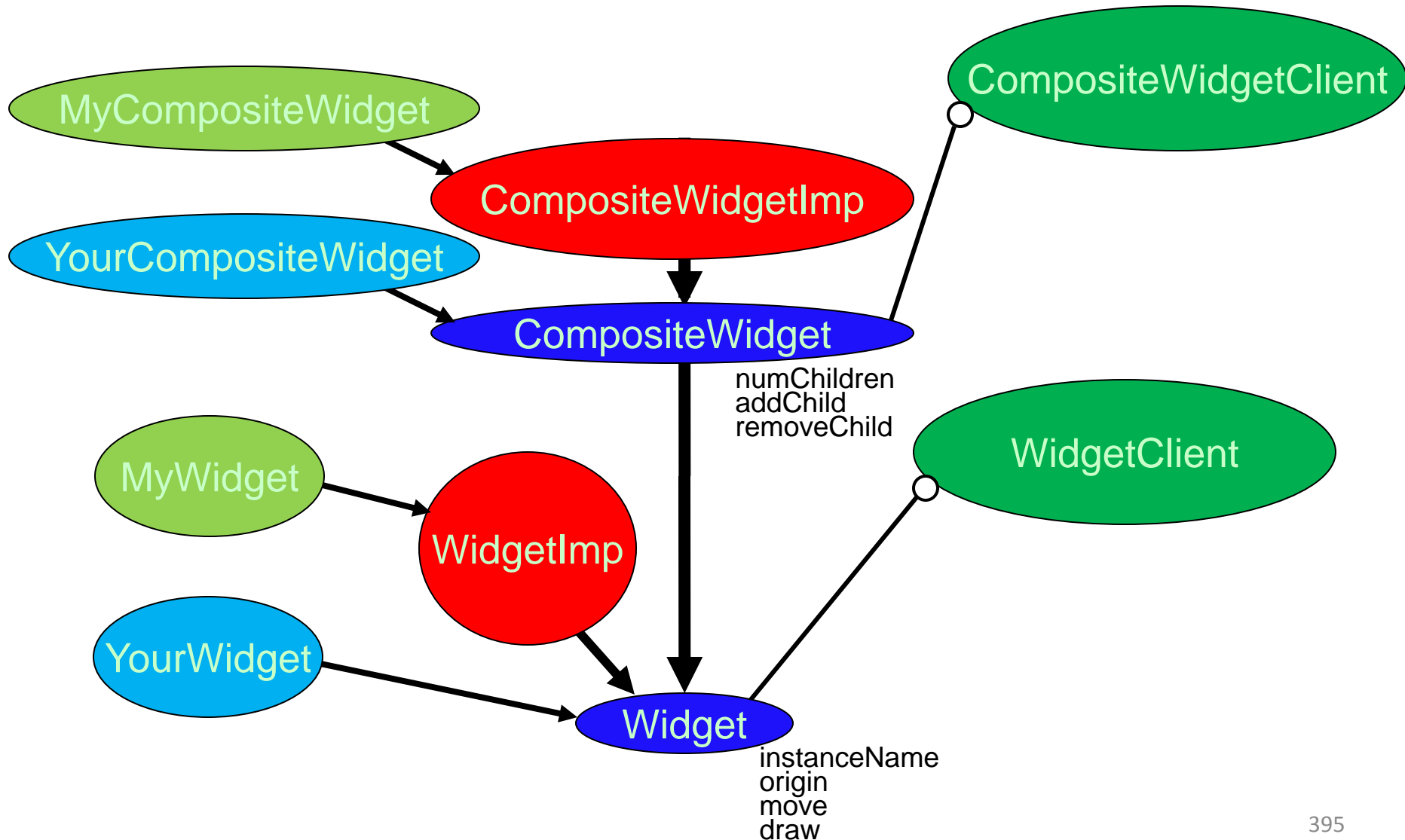
4. Proper Inheritance

Using Implementation Inheritance Effectively



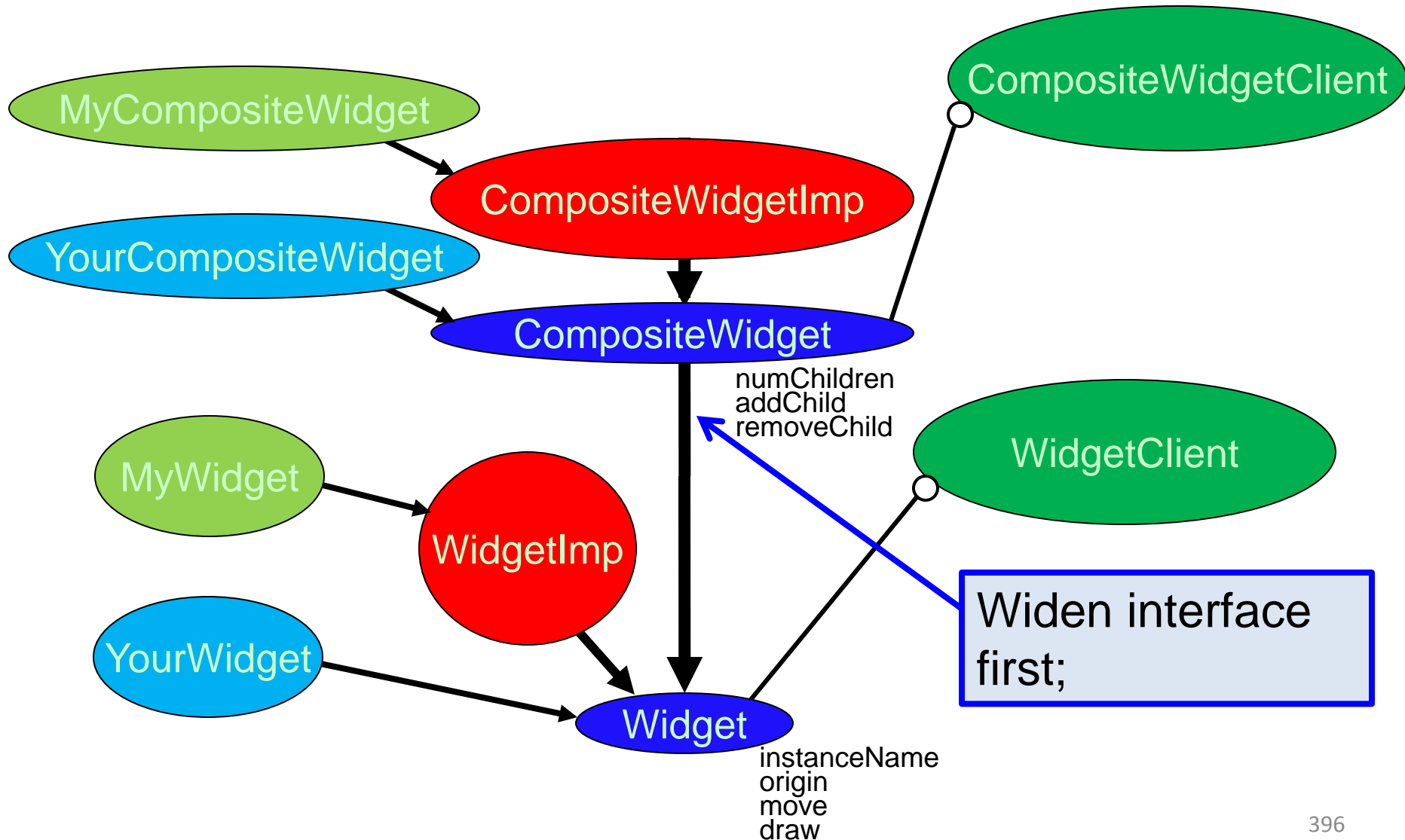
4. Proper Inheritance

Using Implementation Inheritance Effectively



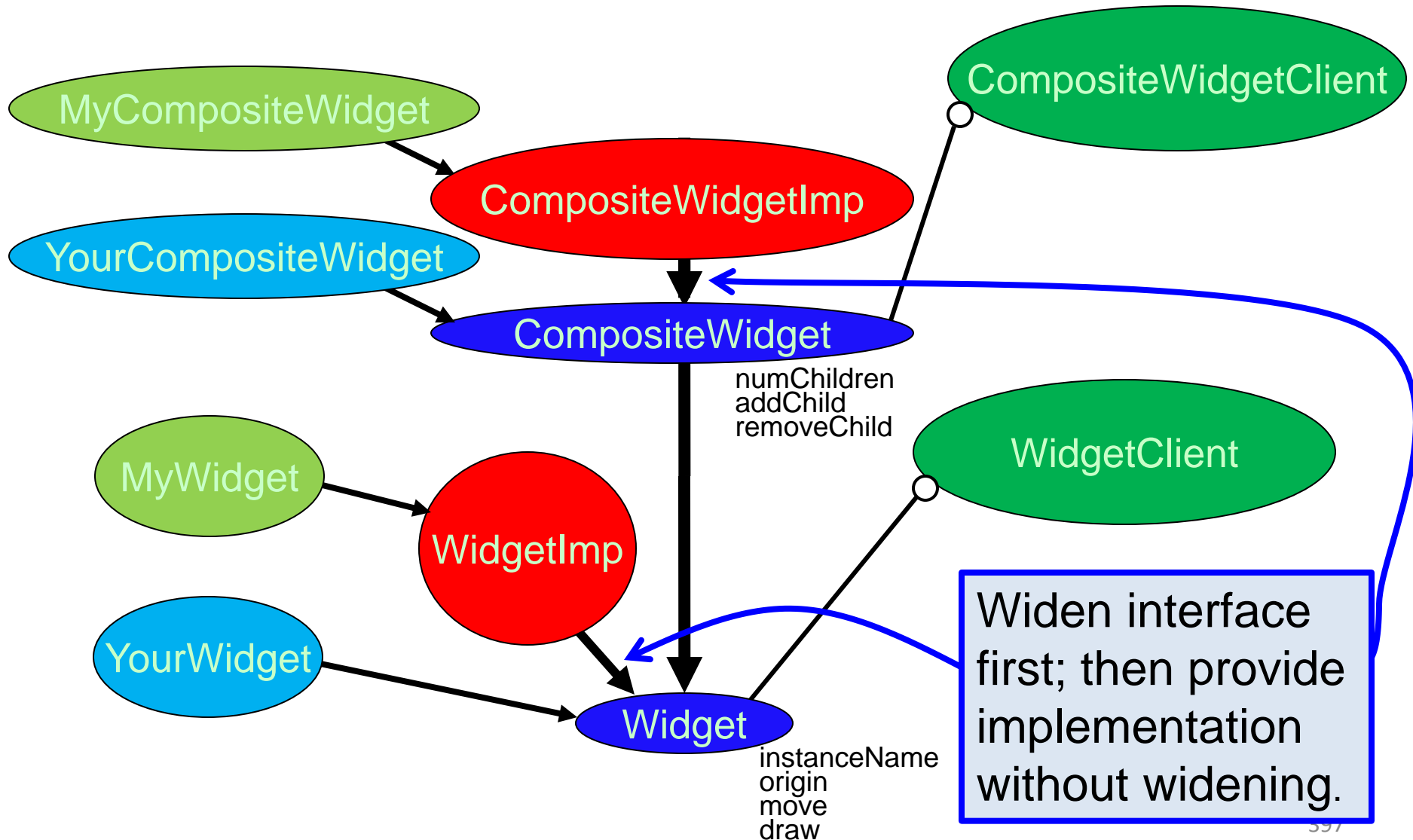
4. Proper Inheritance

Using Implementation Inheritance Effectively



4. Proper Inheritance

Using Implementation Inheritance Effectively



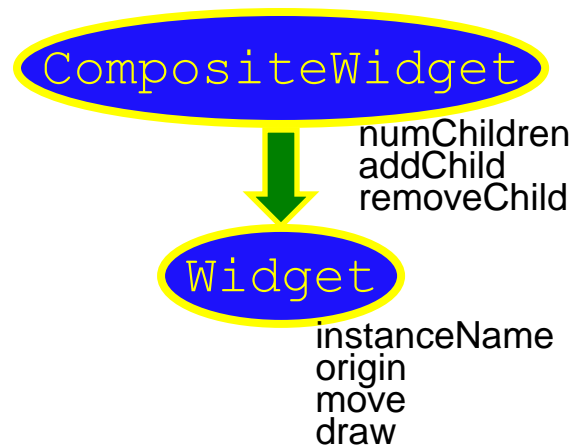
4. Proper Inheritance

Using Implementation Inheritance Effectively

The principal client of
Implementation Inheritance
is the
DERIVED-CLASS AUTHOR.

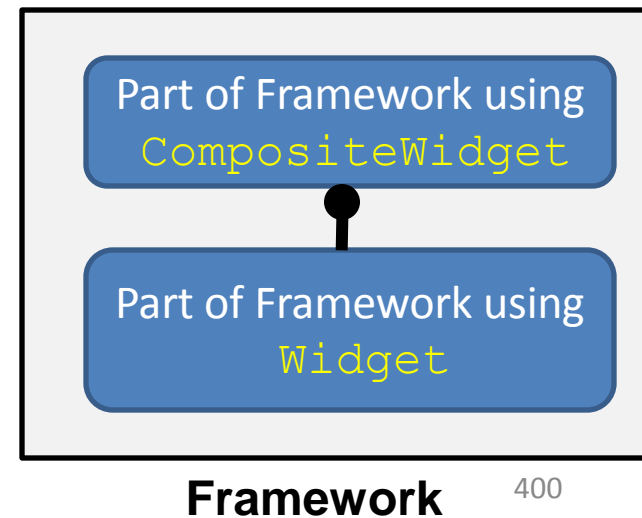
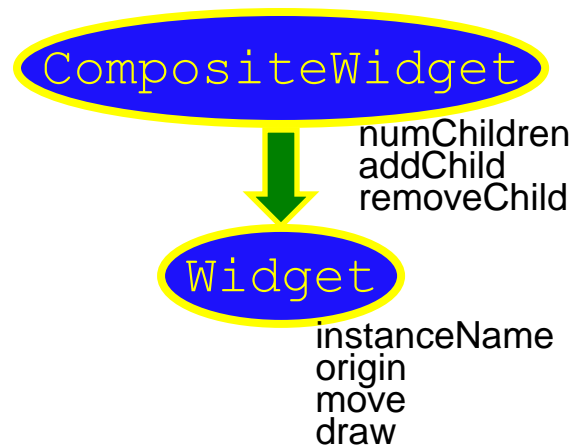
4. Proper Inheritance

Using Implementation Inheritance Effectively



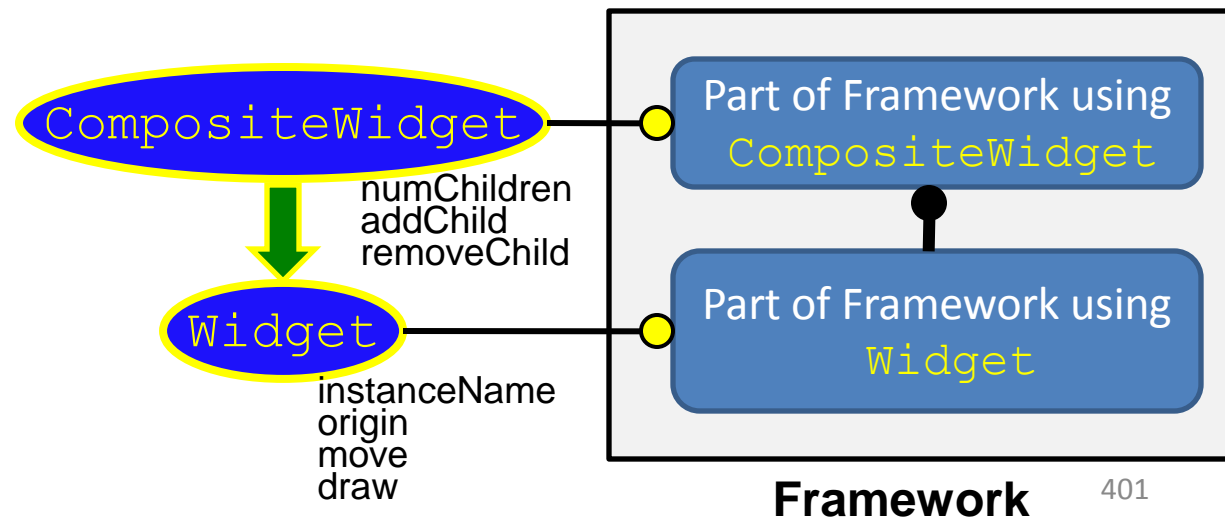
4. Proper Inheritance

Using Implementation Inheritance Effectively



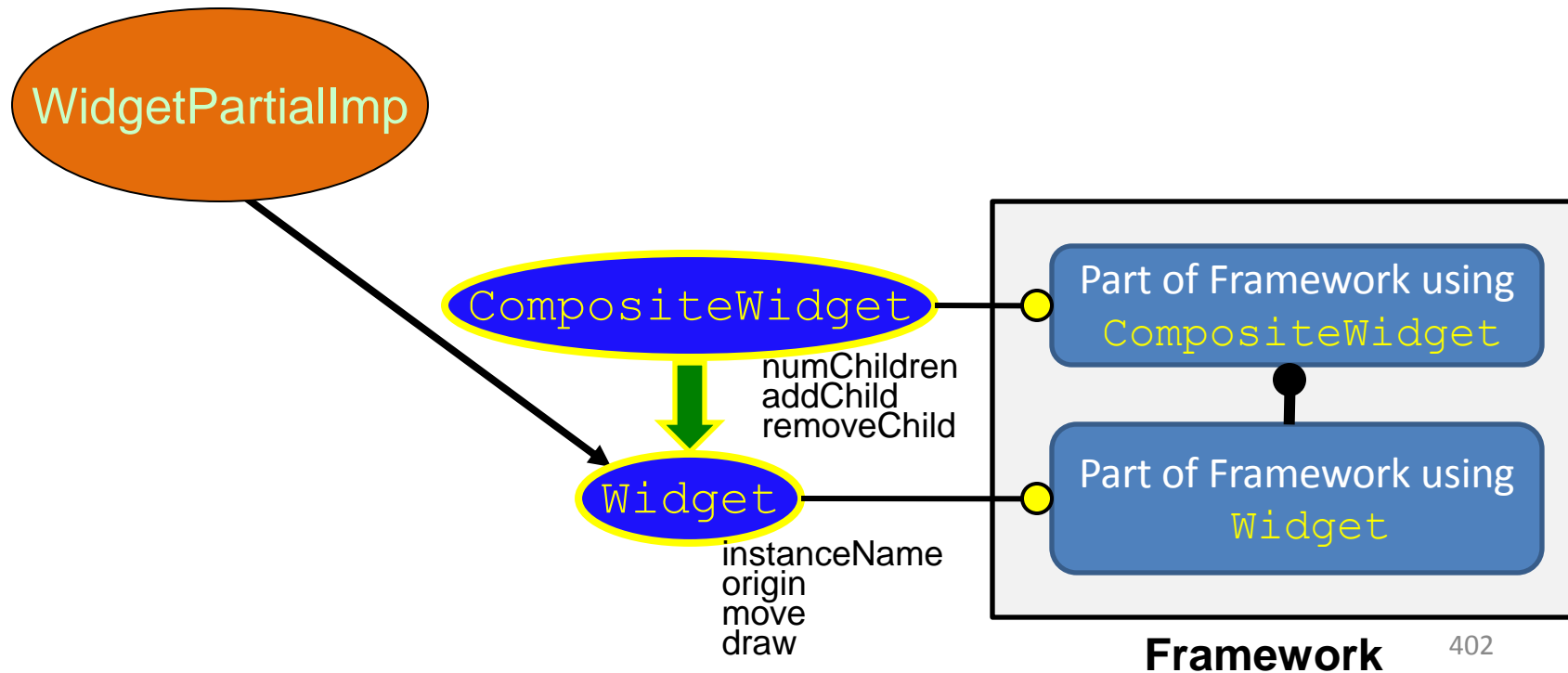
4. Proper Inheritance

Using Implementation Inheritance Effectively



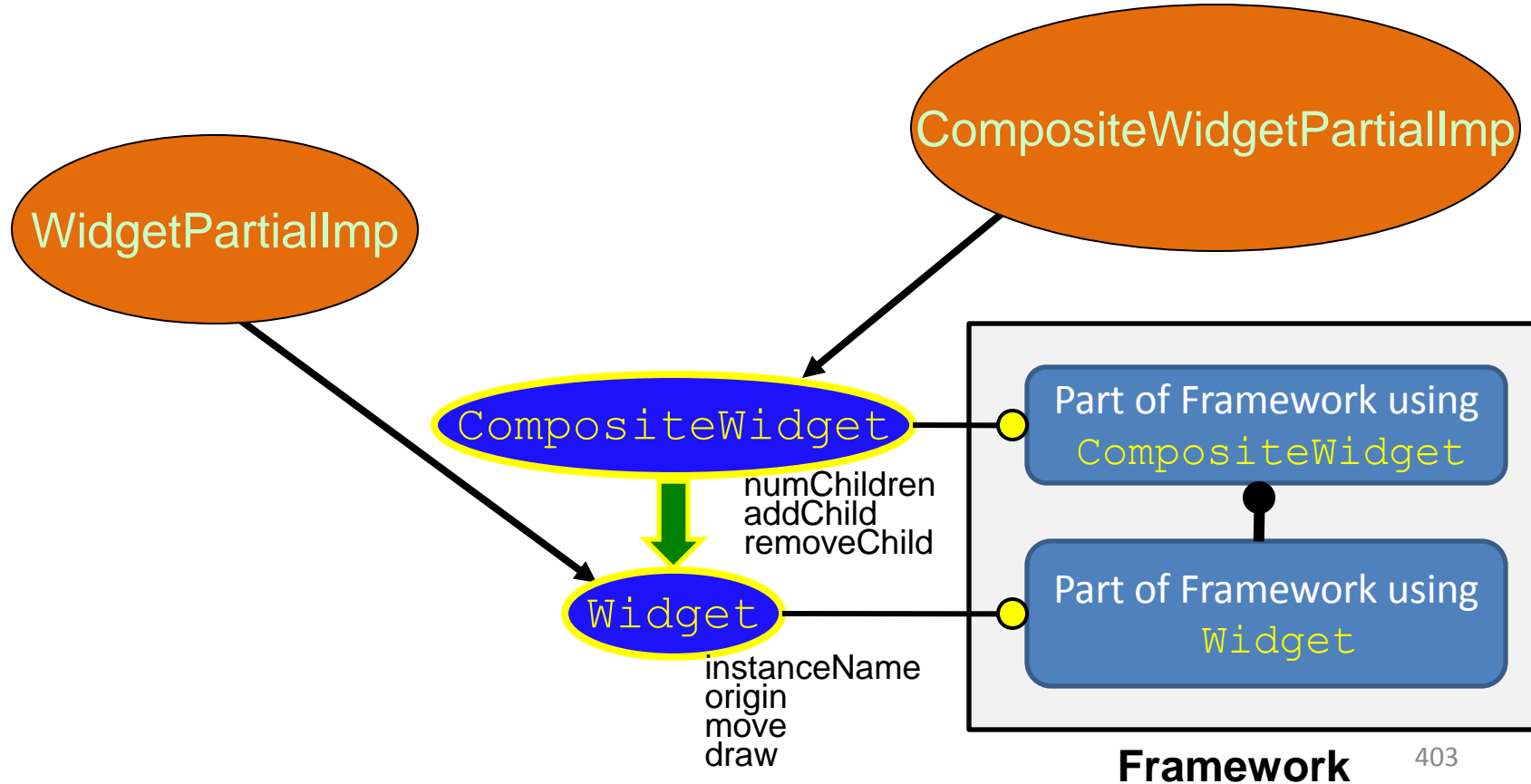
4. Proper Inheritance

Using Implementation Inheritance Effectively



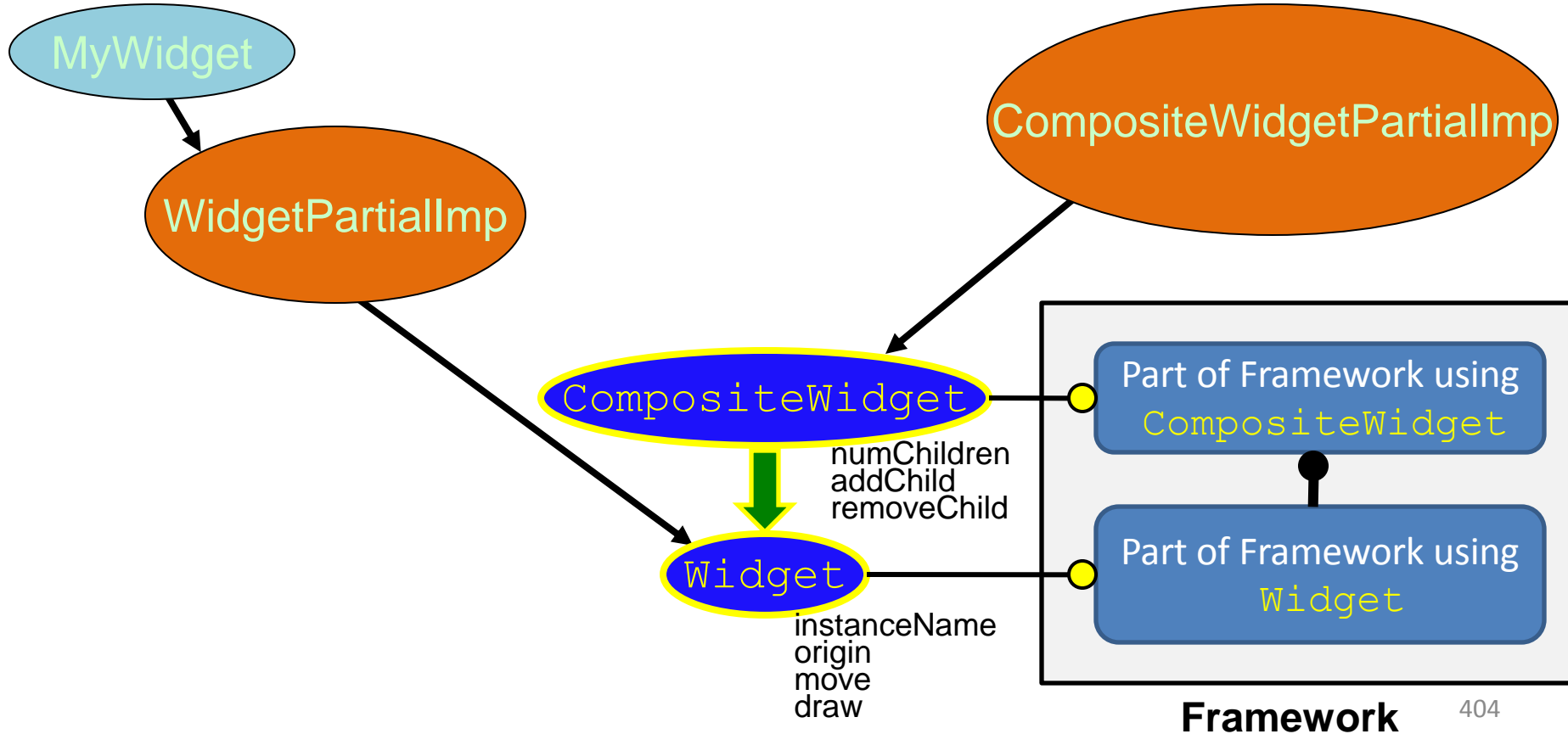
4. Proper Inheritance

Using Implementation Inheritance Effectively



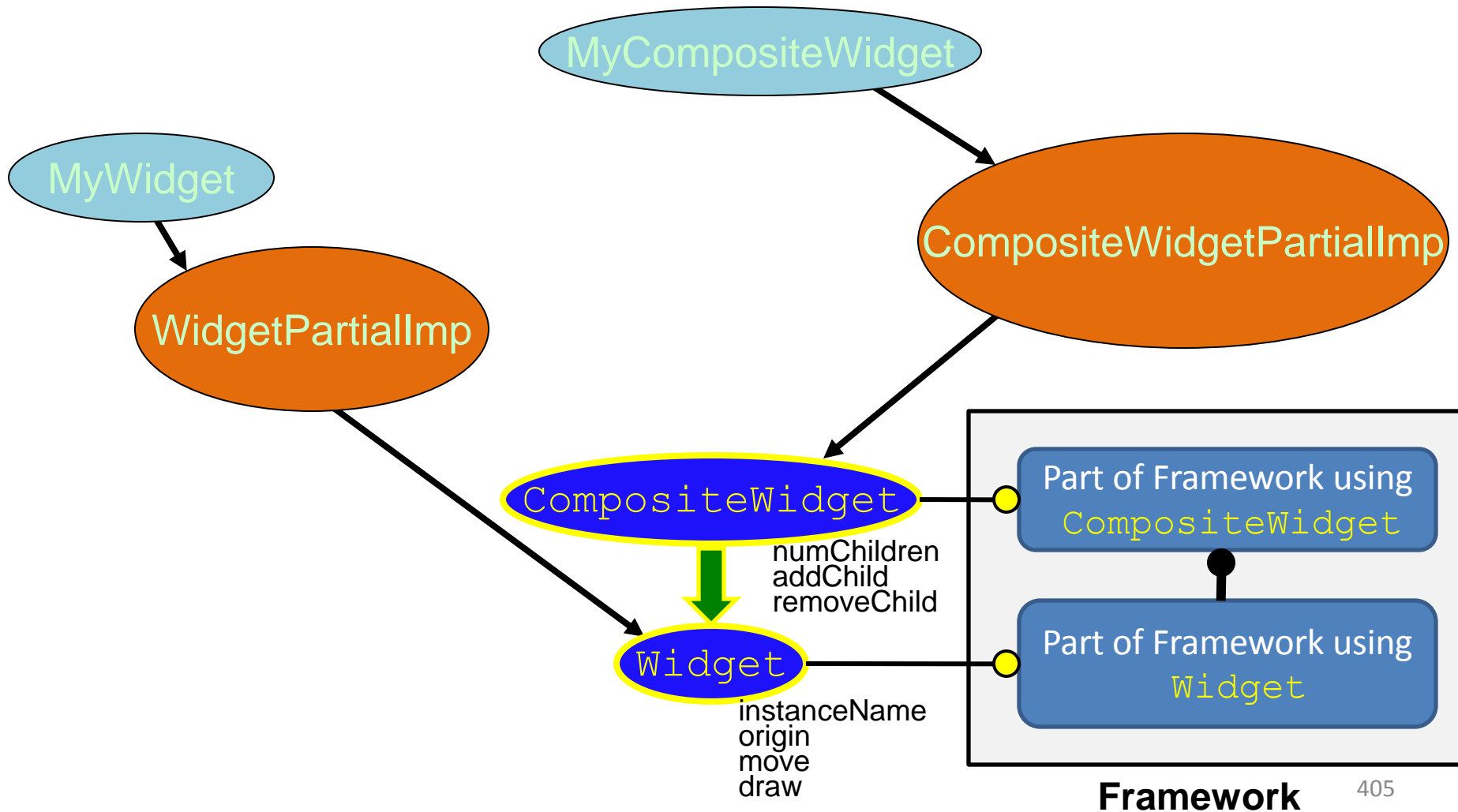
4. Proper Inheritance

Using Implementation Inheritance Effectively



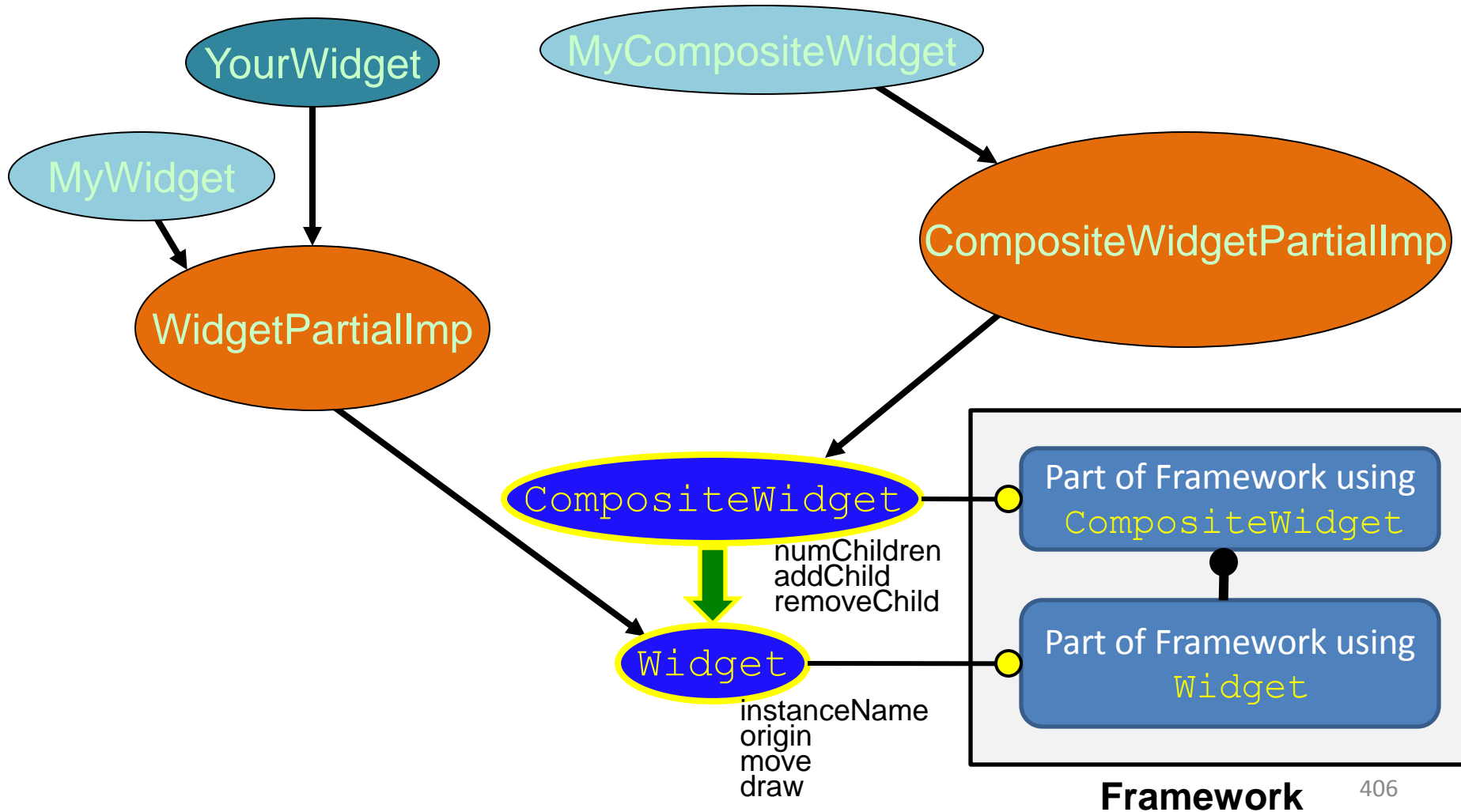
4. Proper Inheritance

Using Implementation Inheritance Effectively



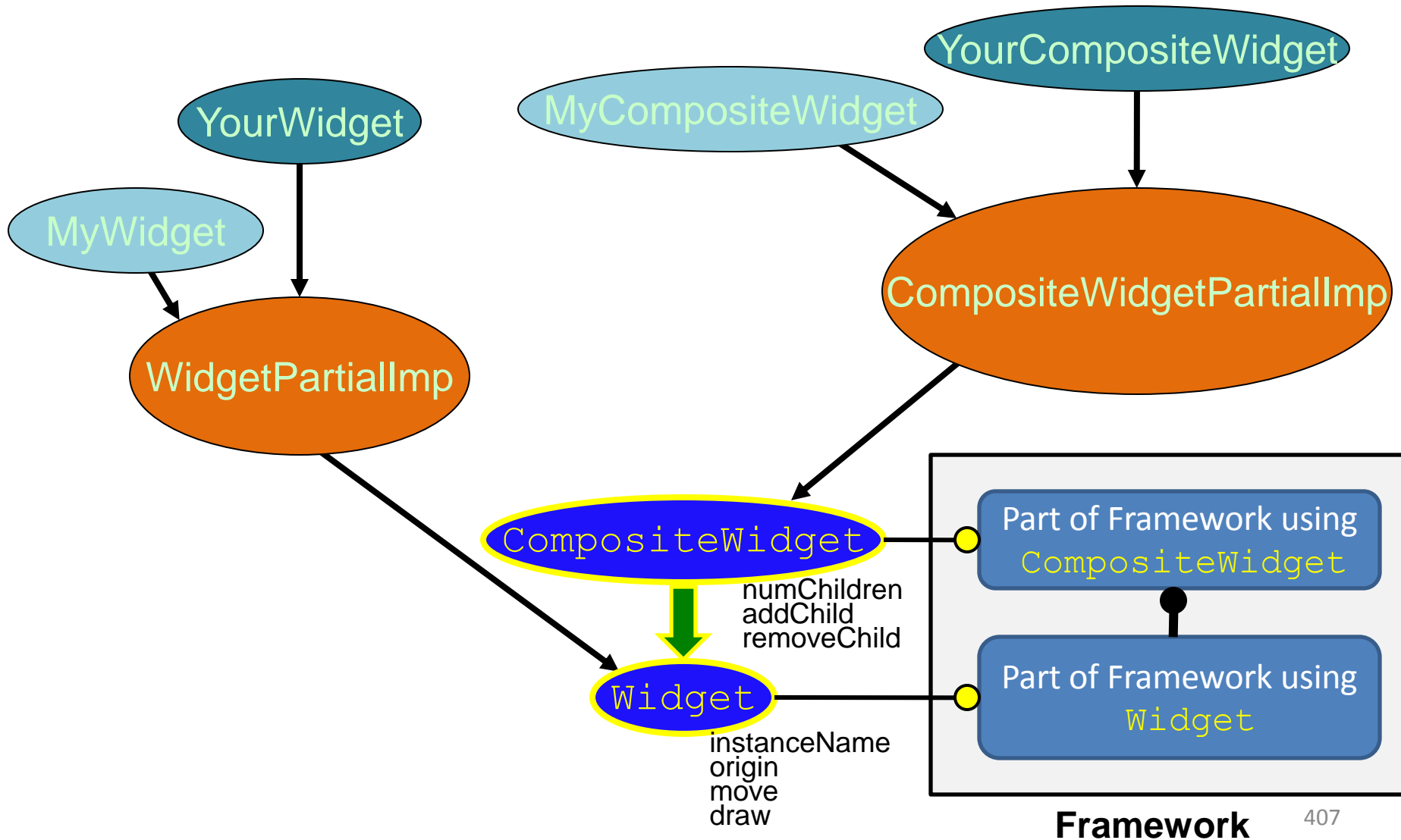
4. Proper Inheritance

Using Implementation Inheritance Effectively



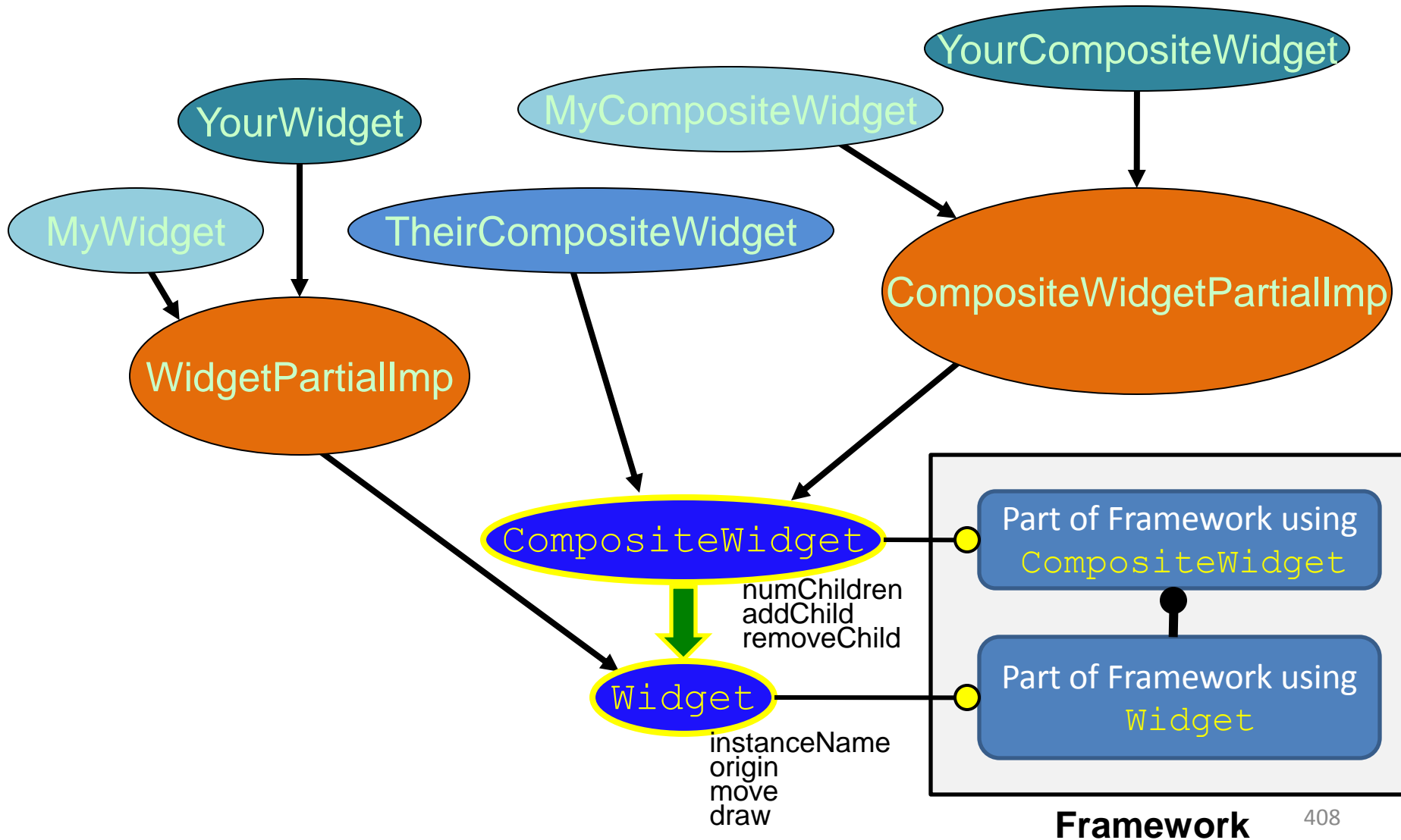
4. Proper Inheritance

Using Implementation Inheritance Effectively



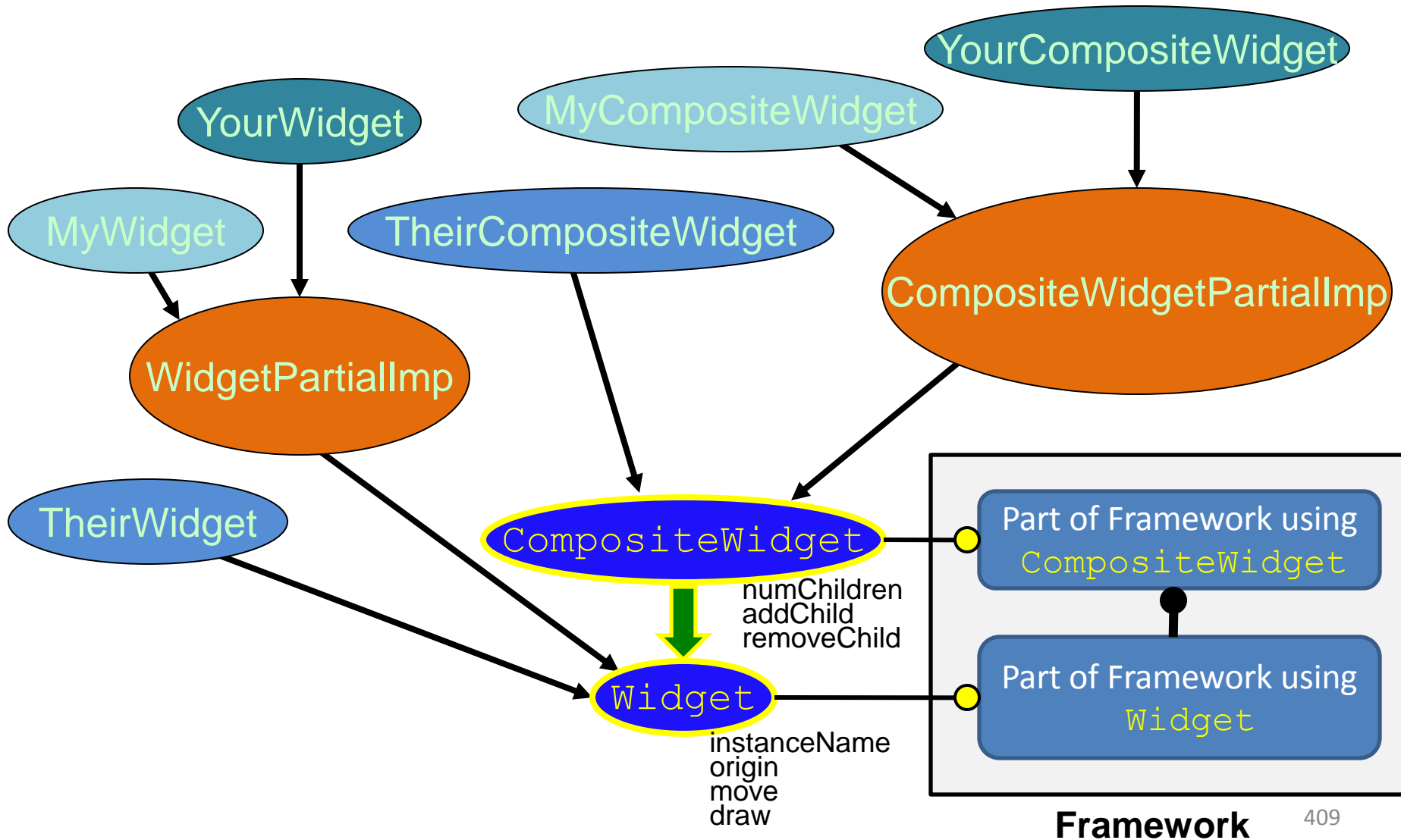
4. Proper Inheritance

Using Implementation Inheritance Effectively



4. Proper Inheritance

Using Implementation Inheritance Effectively



4. Proper Inheritance

Combining Kinds of Inheritance

4. Proper Inheritance

Combining Kinds of Inheritance

- **Structural & Interface**

4. Proper Inheritance

Combining Kinds of Inheritance

- Structural & Interface
 - Typically for Efficiency and Syntactic Sugar.

4. Proper Inheritance

Combining Kinds of Inheritance

- **Structural & Interface**
 - Typically for Efficiency and Syntactic Sugar.
- **Interface & Implementation**

4. Proper Inheritance

Combining Kinds of Inheritance

- **Structural & Interface**
 - Typically for Efficiency and Syntactic Sugar.
- **Interface & Implementation**
 - Interface inheritance (widening) first; then implementation inheritance (no widening) .

4. Proper Inheritance

Combining Kinds of Inheritance

- **Structural & Interface**
 - Typically for Efficiency and Syntactic Sugar.
- **Interface & Implementation**
 - Interface inheritance (widening) first; then implementation inheritance (no widening) .
- **Implementation & Structural**

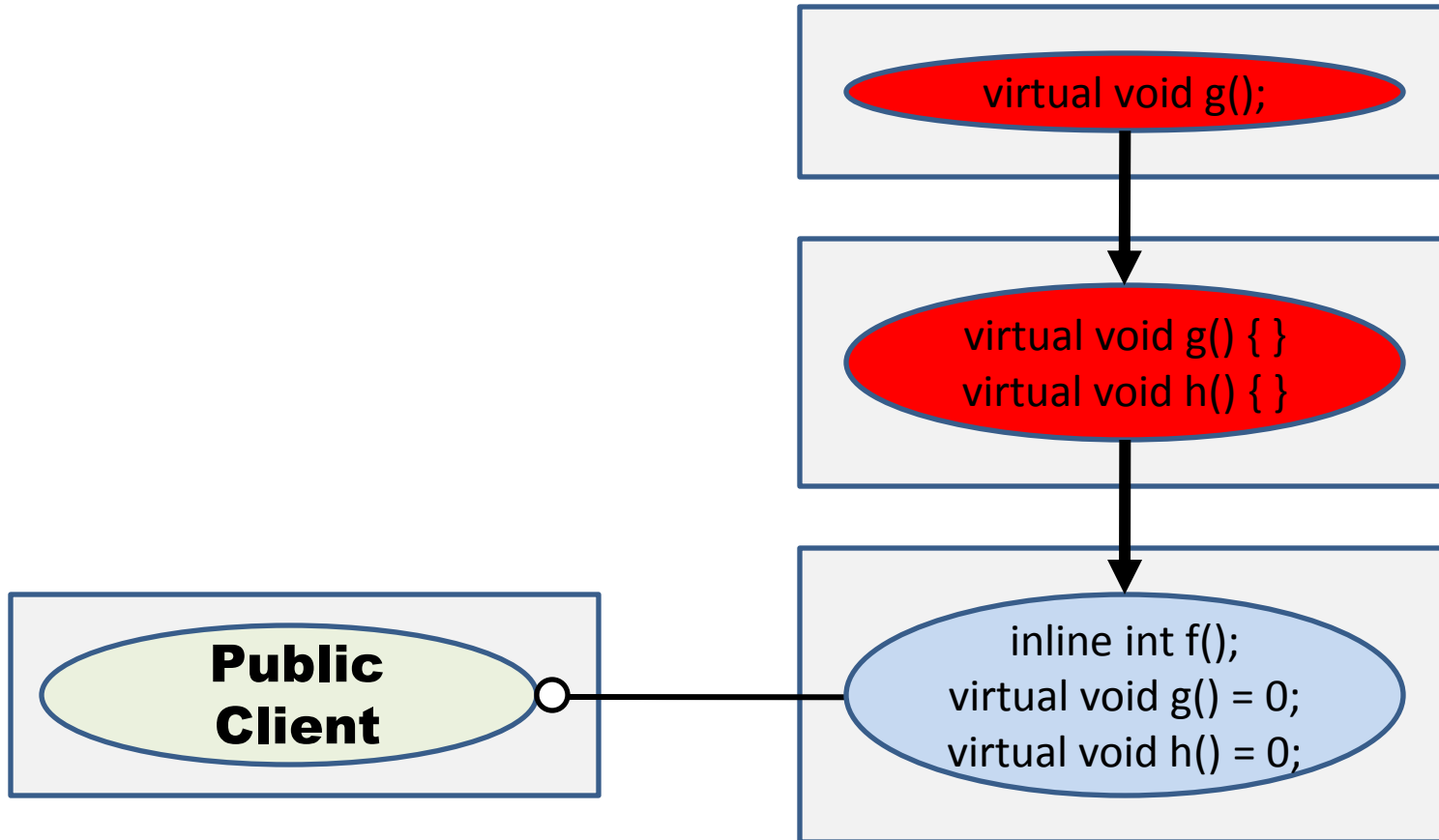
4. Proper Inheritance

Combining Kinds of Inheritance

- **Structural & Interface**
 - Typically for Efficiency and Syntactic Sugar.
- **Interface & Implementation**
 - Interface inheritance (widening) first; then implementation inheritance (no widening).
- **Implementation & Structural**
 - Bad Idea: Unnecessarily addresses the needs of derived class authors and public clients in the same physical component.

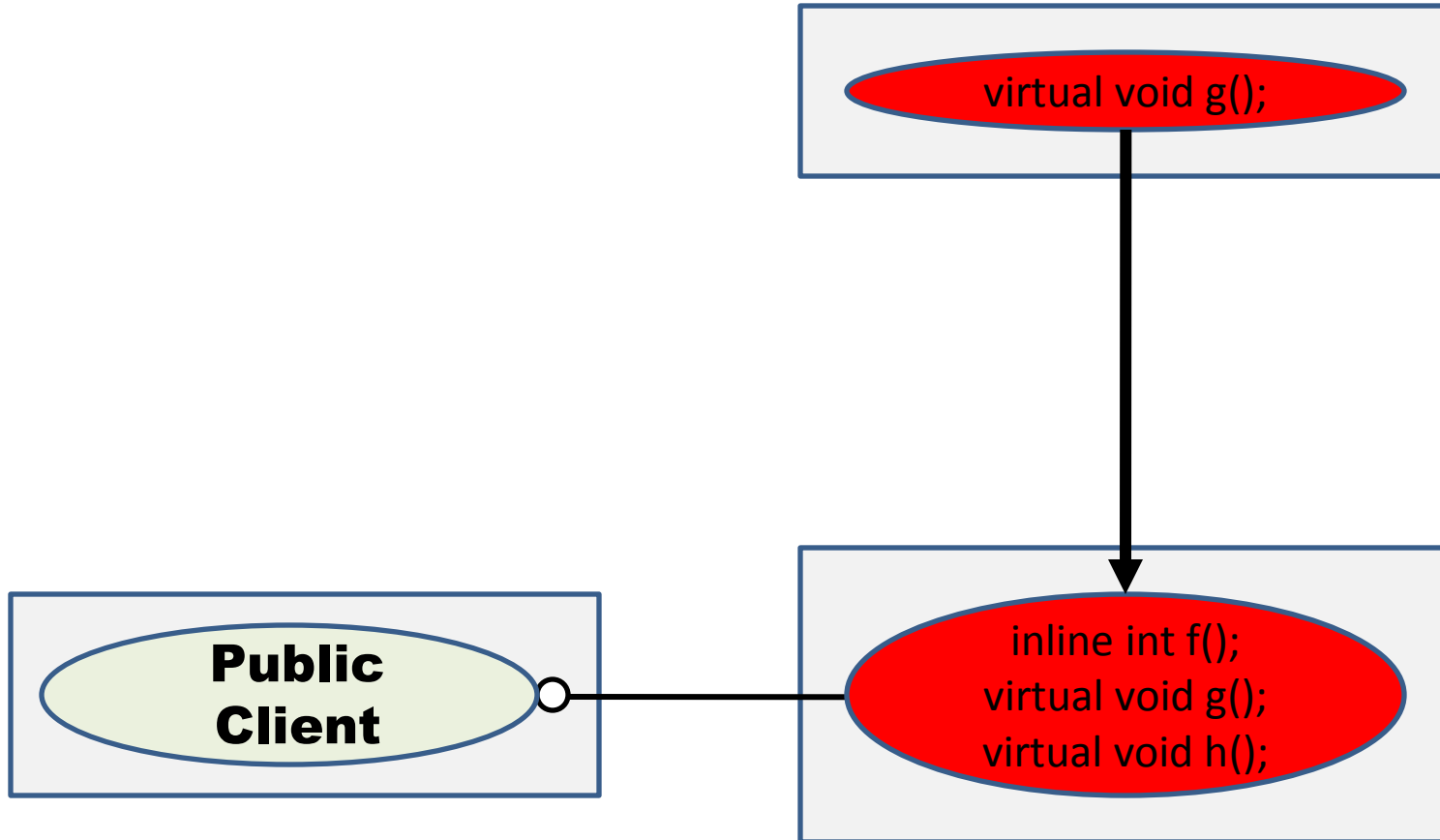
4. Proper Inheritance

Combining Kinds of Inheritance



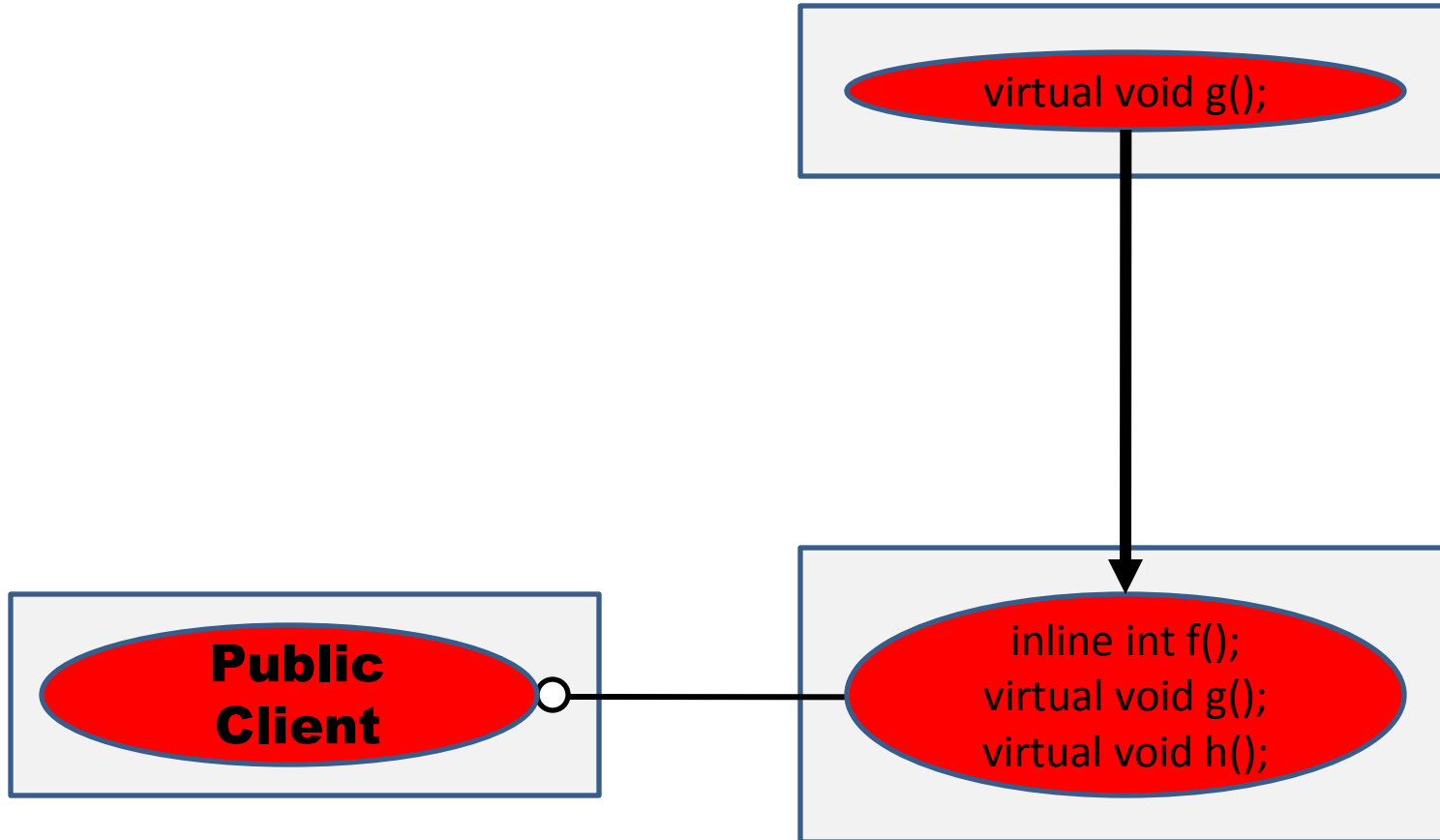
4. Proper Inheritance

Combining Kinds of Inheritance



4. Proper Inheritance

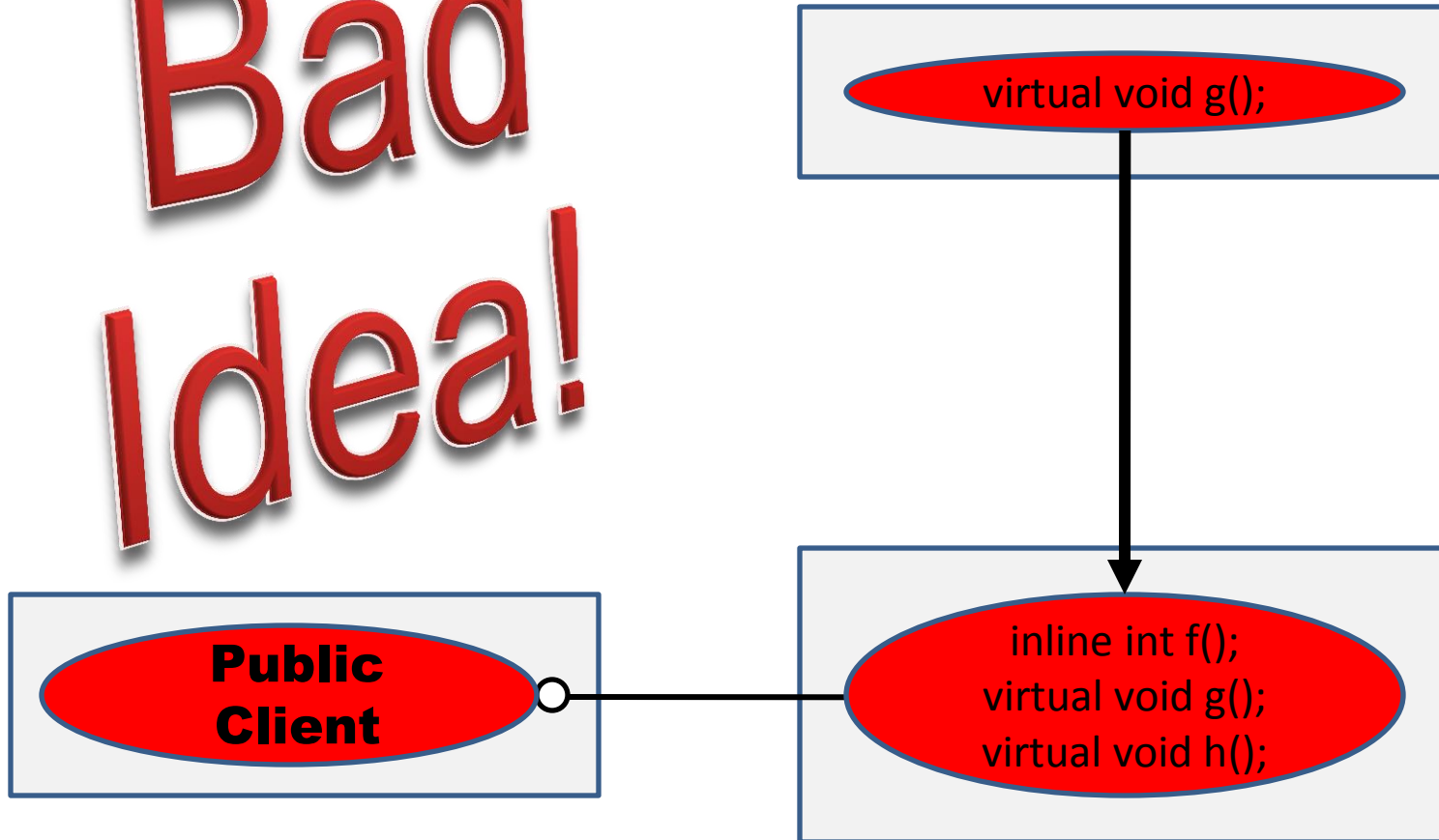
Combining Kinds of Inheritance



4. Proper Inheritance

Combining Kinds of Inheritance

**Bad
Idea!**



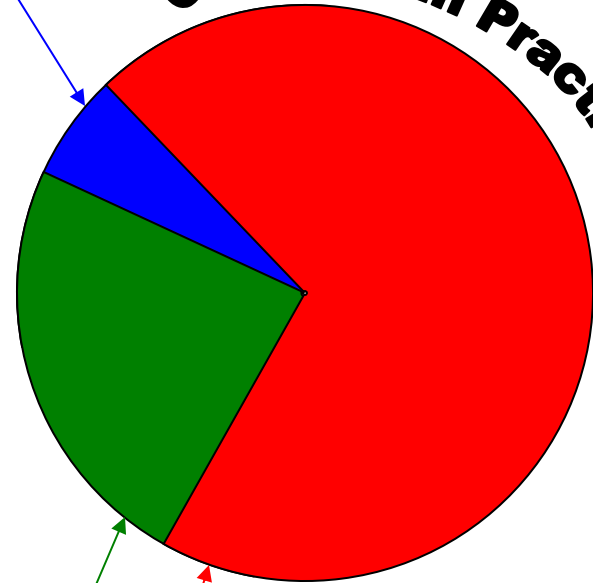
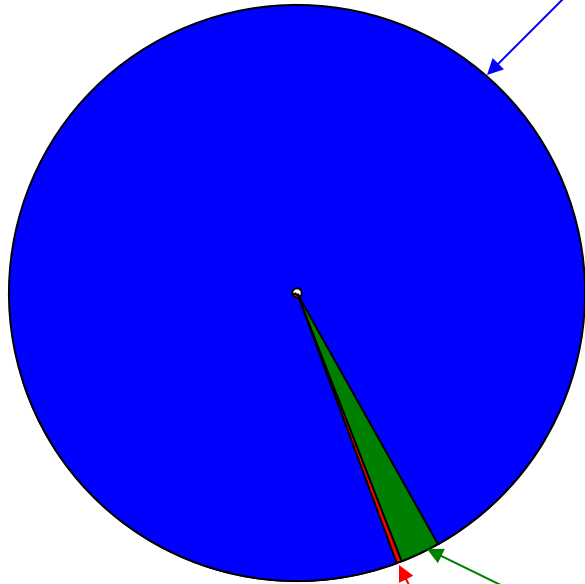
4. Proper Inheritance

Relative Utility

Proper in Theory!

Interface Inheritance

Common in Practice

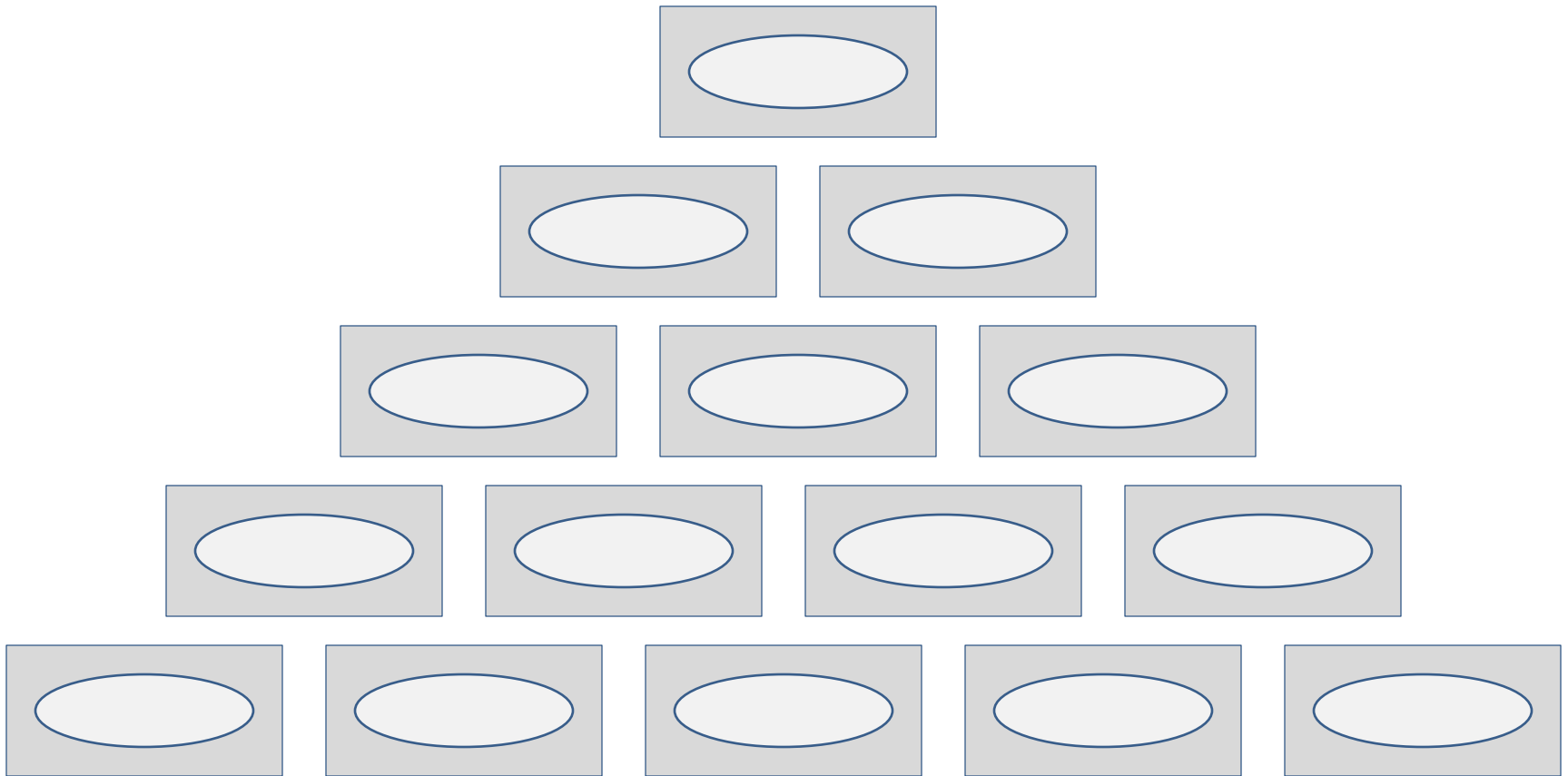


Structural Inheritance

Implementation Inheritance

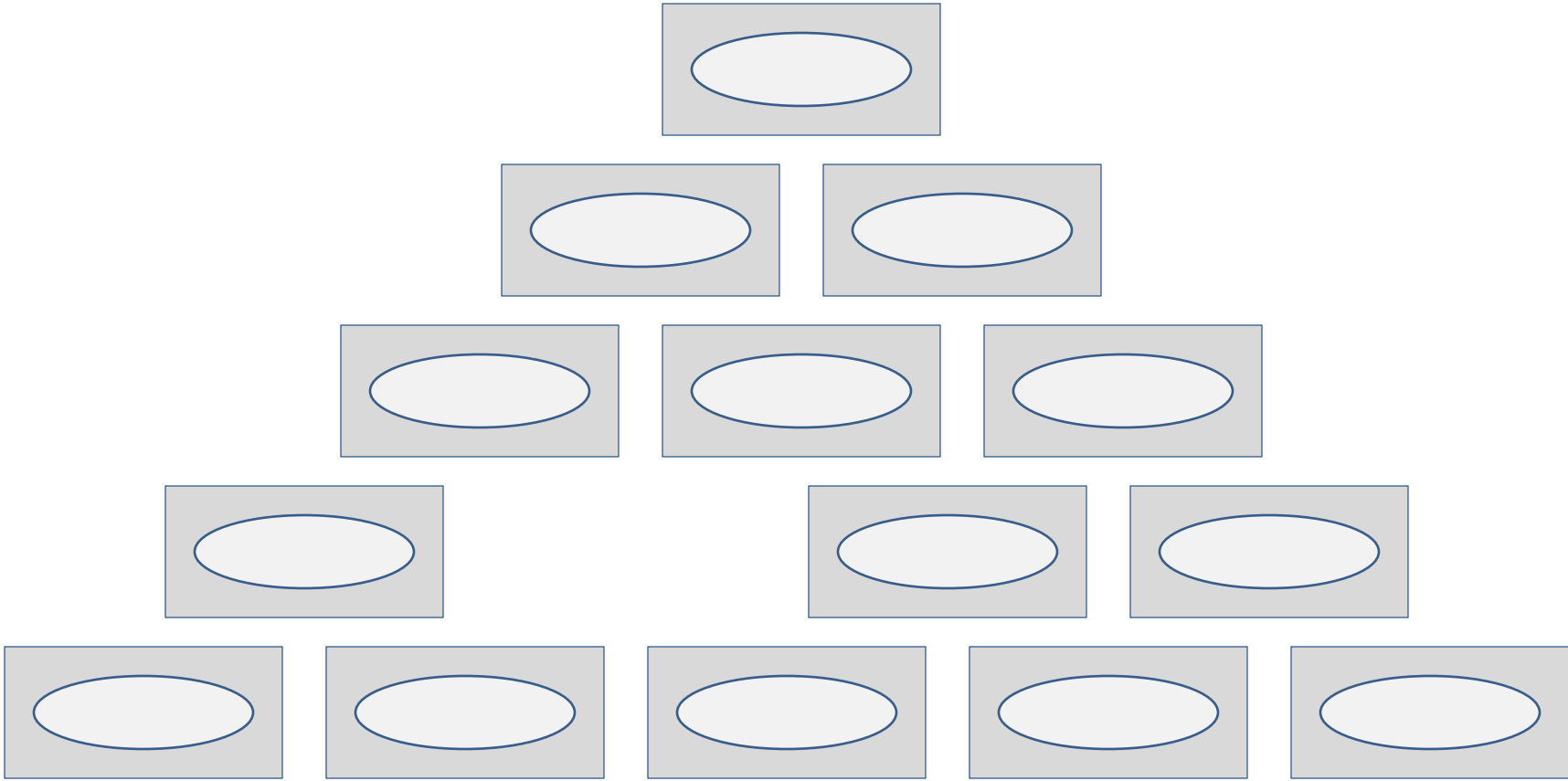
4. Proper Inheritance

Physical Substitutability



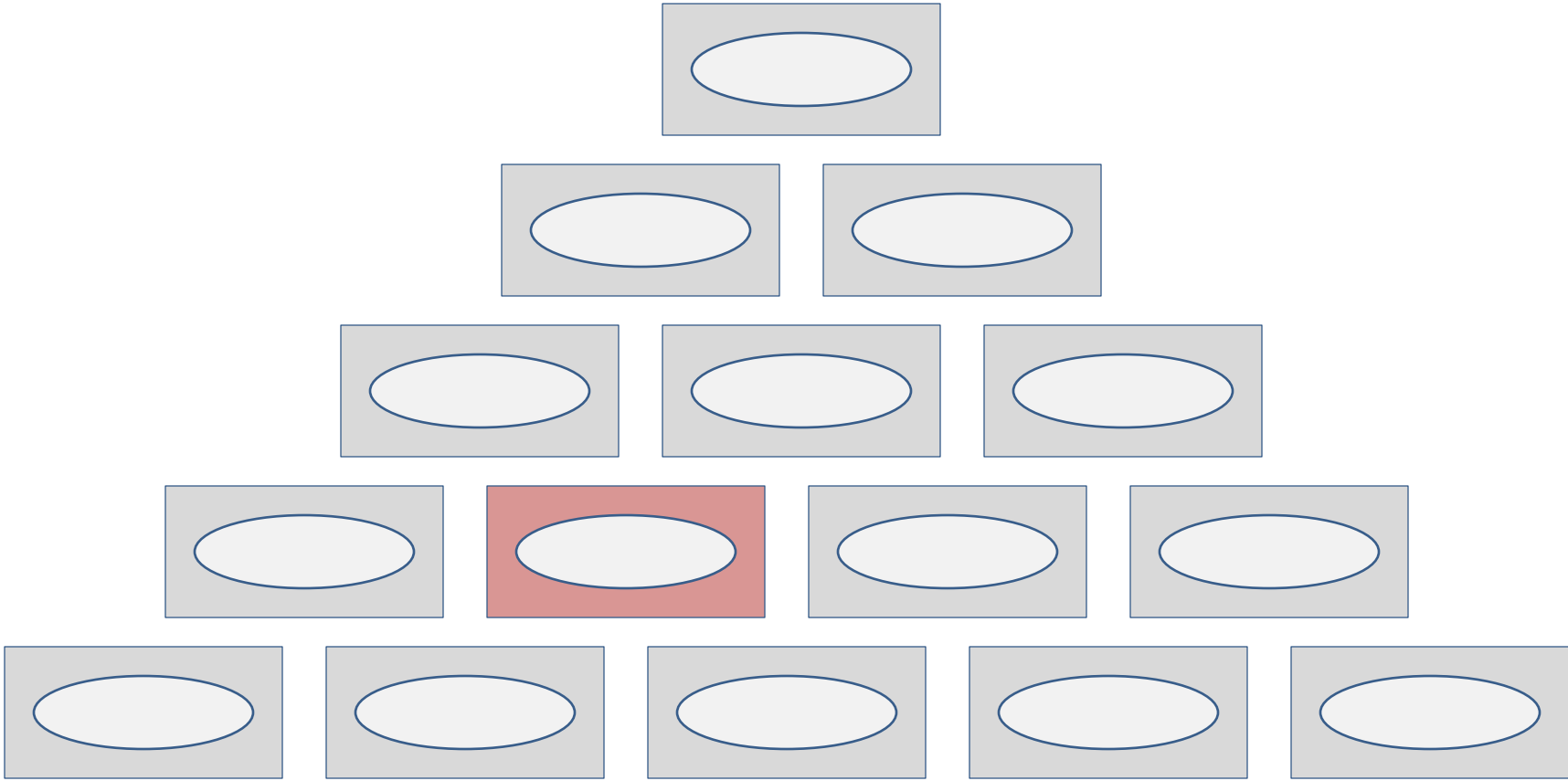
4. Proper Inheritance

Physical Substitutability



4. Proper Inheritance

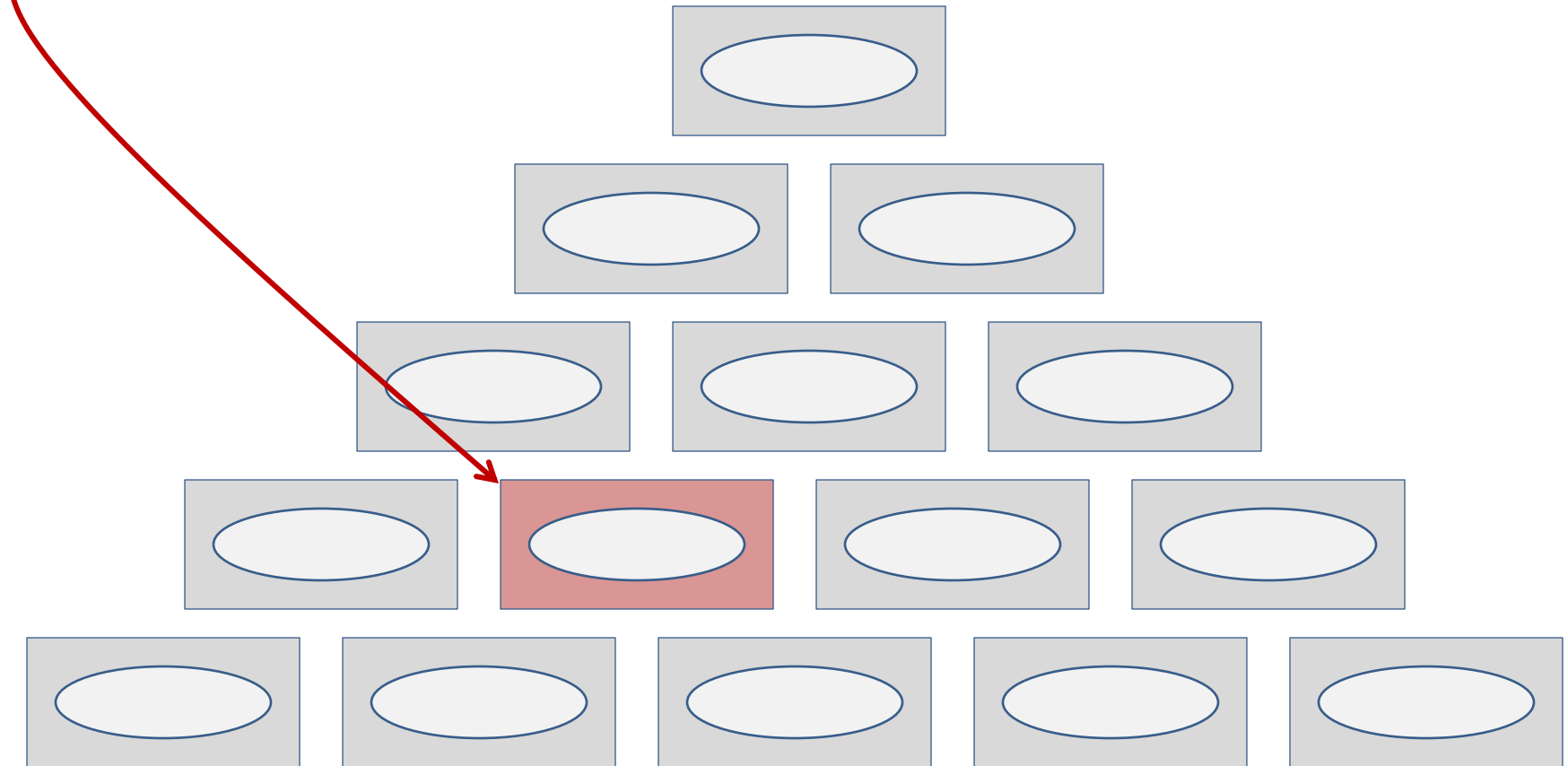
Physical Substitutability



4. Proper Inheritance

Physical Substitutability

What Criteria Must Be Satisfied?



4. Proper Inheritance

Physical Substitutability

The new component's logical behavior:

4. Proper Inheritance

Physical Substitutability

The new component's logical behavior:

- Preconditions needed for *defined behavior* can be made weaker, but no stronger.

4. Proper Inheritance

Physical Substitutability

The new component's logical behavior:

- Preconditions needed for *defined behavior* can be made weaker, but no stronger.
- Pre-existing *essential behavior* of the component must remain unchanged.

4. Proper Inheritance

Physical Substitutability

The new component's logical behavior:

- Preconditions needed for *defined behavior* can be made weaker, but no stronger.
- Pre-existing *essential behavior* of the component must remain unchanged.
- New behaviors may be defined, and essential ones extended, so long as the component is backward compatible with pre-existing clients.

4. Proper Inheritance

Physical Substitutability

The new component's physical characteristics:

4. Proper Inheritance

Physical Substitutability

The new component's physical characteristics:

- Physical dependencies cannot increase (much).

4. Proper Inheritance

Physical Substitutability

The new component's physical characteristics:

- Physical dependencies cannot increase (much).
- Compile-time cannot increase substantially.

4. Proper Inheritance

Physical Substitutability

The new component's physical characteristics:

- Physical dependencies cannot increase (much).
- Compile-time cannot increase substantially.
- Size (footprint) cannot increase (much).

4. Proper Inheritance

Physical Substitutability

The new component's physical characteristics:

- Physical dependencies cannot increase (much).
- Compile-time cannot increase substantially.
- Size (footprint) cannot increase (much).
- **Dynamic memory usage can't increase (much).**

4. Proper Inheritance

Physical Substitutability

The new component's physical characteristics:

- Physical dependencies cannot increase (much).
- Compile-time cannot increase substantially.
- Size (footprint) cannot increase (much).
- Dynamic memory usage can't increase (much).
- **Can't introduce dynamic memory allocation.**

4. Proper Inheritance

Physical Substitutability

The new component's physical characteristics:

- Physical dependencies cannot increase (much).
- Compile-time cannot increase substantially.
- Size (footprint) cannot increase (much).
- Dynamic memory usage can't increase (much).
- Can't introduce dynamic memory allocation.
- **Runtime must not be increased significantly for important (relevant) use-cases.**

4. Proper Inheritance
End of Section

Questions?

4. Proper Inheritance

What Questions are we Answering?

- What distinguishes *Interface*, *Structural*, and *Implementation* inheritance?
- What do we mean by the *Is-A* relationship, & how does *proper inheritance* vary from one form to the next.
 - What does **LSP** (*Liskov Substitution Principle*) have to do with it?
- How are each of the three inheritances used effectively?
 - Who is the principal client of each kind of inheritance?
 - How are interface and implementation inheritance *ordered*?
 - Does it make sense to combine two (or all three) inheritances?
 - What is the relative utility of the three forms of inheritance?
- How are *structural inheritance*, (logical) *substitutability*, & *backward compatibility* of (physical) components related?

Outline

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

4. Proper Inheritance

Is-A for *Interface*, *Structural*, & *Implementation* Inheritance

Conclusion

1. Components *(review)*

Modularity, Logical/Physical Dependencies, & Level numbers

Conclusion

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

- A ***Component*** – a **`.h/ .cpp`** pair satisfying four essential properties – is our **fundamental unit** of both ***logical*** and ***physical*** software design.

Conclusion

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

- A ***Component*** – a **`.h/ .cpp`** pair satisfying four essential properties – is our **fundamental unit** of both ***logical*** and ***physical*** software design.
- Logical relationships, such as ***Is-A*** and ***Uses*** between classes, imply ***physical dependencies*** among the *components* that defined them.

Conclusion

1. Components (review)

Modularity, Logical/Physical Dependencies, & Level numbers

- A ***Component*** – a **`.h/ .cpp`** pair satisfying four essential properties – is our **fundamental unit** of both ***logical*** and ***physical*** software design.
- Logical relationships, such as ***Is-A*** and ***Uses*** between classes, imply ***physical dependencies*** among the *components* that defined them.
- **No** ***cyclic dependencies/long-distance*** friendships!

Conclusion

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

Conclusion

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

- An ***interface*** is *syntactic*; a ***contract*** is *semantic*.

Conclusion

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

- An ***interface*** is *syntactic*; a ***contract*** is *semantic*.
- A *contract* defines both ***pre-*** & ***postconditions***.

Conclusion

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

- An ***interface*** is *syntactic*; a ***contract*** is *semantic*.
- A *contract* defines both ***pre-*** & ***postconditions***.
- ***Undefined Behavior*** if a precondition isn't met.

Conclusion

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

- An ***interface*** is *syntactic*; a ***contract*** is *semantic*.
- A *contract* defines both ***pre-*** & ***postconditions***.
- ***Undefined Behavior*** if a precondition isn't met.
- What *undefined behavior* does **is undefined!**

Conclusion

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

- An ***interface*** is *syntactic*; a ***contract*** is *semantic*.
- A *contract* defines both ***pre-*** & ***postconditions***.
- ***Undefined Behavior*** if a precondition isn't met.
- What *undefined behavior* does **is undefined!**
- Documented ***essential behavior*** **must not change!**

Conclusion

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

- An ***interface*** is *syntactic*; a ***contract*** is *semantic*.
- A *contract* defines both ***pre-*** & ***postconditions***.
- ***Undefined Behavior*** if a precondition isn't met.
- What *undefined behavior* does **is undefined!**
- Documented ***essential behavior*** must not change!
- ***Test drivers*** must verify all *essential behavior*.

Conclusion

2. Interfaces and Contracts (review)

Syntax versus Semantics & *Essential Behavior*

- An ***interface*** is *syntactic*; a ***contract*** is *semantic*.
- A *contract* defines both ***pre-*** & ***postconditions***.
- ***Undefined Behavior*** if a precondition isn't met.
- What *undefined behavior* does **is undefined!**
- Documented ***essential behavior*** must not change!
- ***Test drivers*** must verify all *essential behavior*.
- *Assertions* in *destructors* help verify ***invariants***.

Conclusion

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

Conclusion

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

- *Narrow contracts admit undefined behavior.*

Conclusion

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

- *Narrow contracts admit undefined behavior.*
- Appropriately narrow contracts are GOOD:

Conclusion

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

- *Narrow contracts admit undefined behavior.*
- Appropriately narrow contracts are GOOD:
 - Reduce costs associated with development/testing

Conclusion

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

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

Conclusion

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

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

Conclusion

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

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

Conclusion

3. Narrow versus Wide Contracts (review)

The Significance of *Undefined Behavior*

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

Conclusion

4. Proper Inheritance

Is-A for Interface, Structural, & Implementation Inheritance

Conclusion

4. Proper Inheritance

Is-A for Interface, Structural, & Implementation Inheritance

- The derived class must adhere to **both** contracts.

Conclusion

4. Proper Inheritance

Is-A for Interface, Structural, & Implementation Inheritance

- The derived class must adhere to **both** contracts.
- The **static type** of the pointer/reference should make **no** difference in programmatic behavior.

Conclusion

4. Proper Inheritance

Is-A for Interface, Structural, & Implementation Inheritance

- The derived class must adhere to **both** contracts.
- The **static type** of the pointer/reference should make **no** difference in programmatic behavior.
- *Interface inheritance is (virtually :-)* all we need!

Conclusion

4. Proper Inheritance

Is-A for Interface, Structural, & Implementation Inheritance

- The derived class must adhere to **both** contracts.
- The **static type** of the pointer/reference should make **no** difference in programmatic behavior.
- *Interface inheritance is (virtually :-)* all we need!
- *Backward **compatibility** for components is a whole lot like **proper structural inheritance.***

Conclusion

The End