Applied Hierarchical Reuse

Capitalizing on Bloomberg's Foundation Libraries

John Lakos

Wednesday, May 15, 2013

Copyright Notice

Abstract

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

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

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

What’s The Problem?

What's The Problem?

Large-Scale C++ Software Design:

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

What's The Problem?

Large-Scale C++ Software Design:

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

What's The Problem?

Large-Scale C++ Software Design:

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

What's The Problem?

Large-Scale C++ Software Design:

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

Purpose of this Talk

There's lots to talk about:

Purpose of this Talk

There's lots to talk about:

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

Purpose of this Talk

There's lots to talk about:

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

- Present ***cohesive techniques*** we have found to address these problems.

Purpose of this Talk

There's lots to talk about:

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

Outline

0. Goals

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

1. Process & Architecture

   Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

   Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

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

4. Bloomberg Development Environment (BDE)

   Rendered as Fine-Grained *Hierarchically* *Reusable* Components.

Outline

0. Goals

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

1. Process & Architecture

   Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

   Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

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

4. Bloomberg Development Environment (BDE)

   Rendered as Fine-Grained *Hierarchically Reusable* Components.

What Are We Trying To Do?

What are the goals?

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

What Are We Trying To Do?

What are the goals?

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

What Are We Trying To Do?

Who are the intended clients?

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

What Are We Trying To Do?

Who are the intended clients?

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

Our Intended Clients

Our Intended Clients

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

Our Intended Clients

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

- Solving real-world industrial-strength problems.

Our Intended Clients

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

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

Our Intended Clients

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

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

Our Intended Clients

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

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

Our Intended Clients

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

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

Our Intended Clients

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

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

Our Intended Clients

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

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

Our Primary Goal

Our Primary Goal

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

Our Primary Goal

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

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

Our Primary Goal

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

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

Our Primary Goal

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

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

Our Primary Goal

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

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

Our Primary Goal

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

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

Our Primary Goal

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

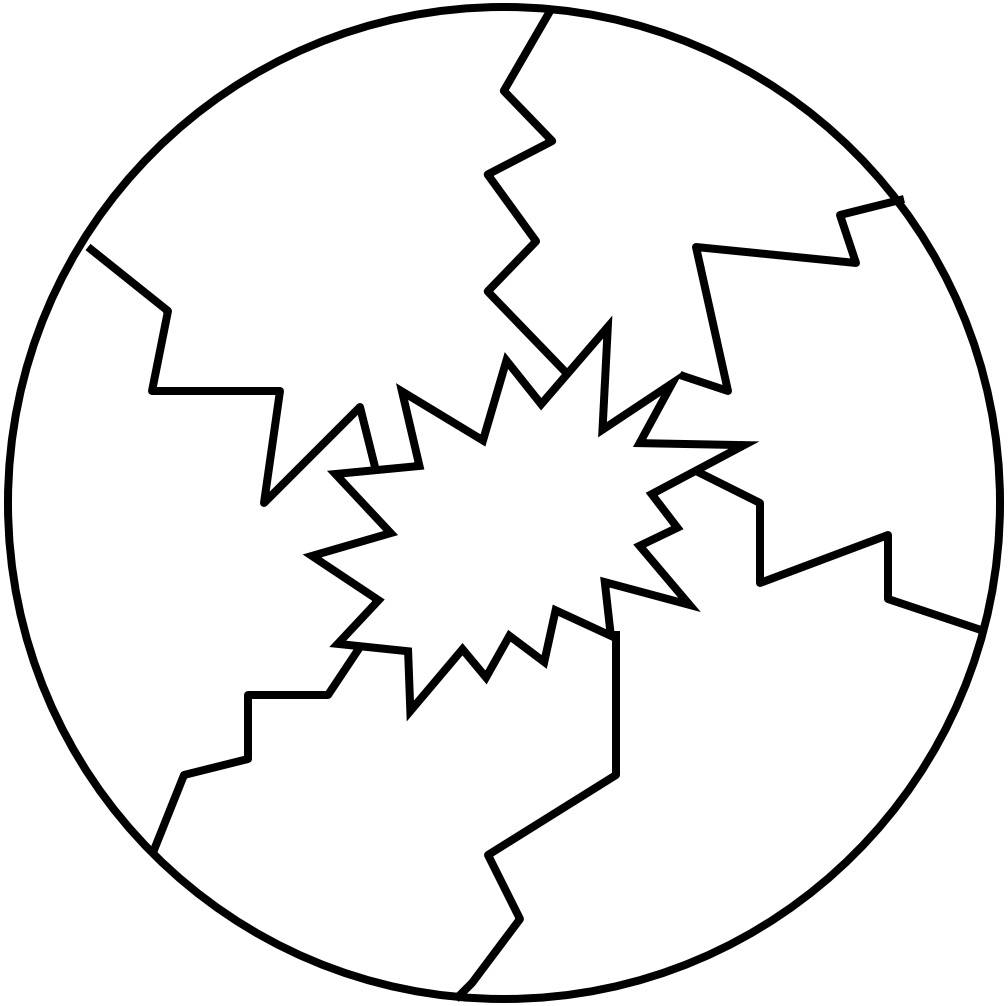
- Demonstrate exemplary methodology that scales to projects of all sizes.
- Provide a framework for developing real-world software effectively.
- Address specific problems that are relevant to our clients.
- Maintain stable solutions used by many versions of many products.
- Teach our clients a proven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

Our Primary Goal

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

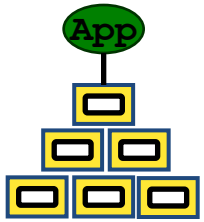
- Demonstrate exemplary methodology that scales to projects of all sizes.
- Provide a framework for developing real-world software effectively.
- Address specific problems that are relevant to our clients.
- Maintain stable solutions used by many versions of many products.
- Teach our clients a proven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

Achieving Reuse



Achieving Reuse

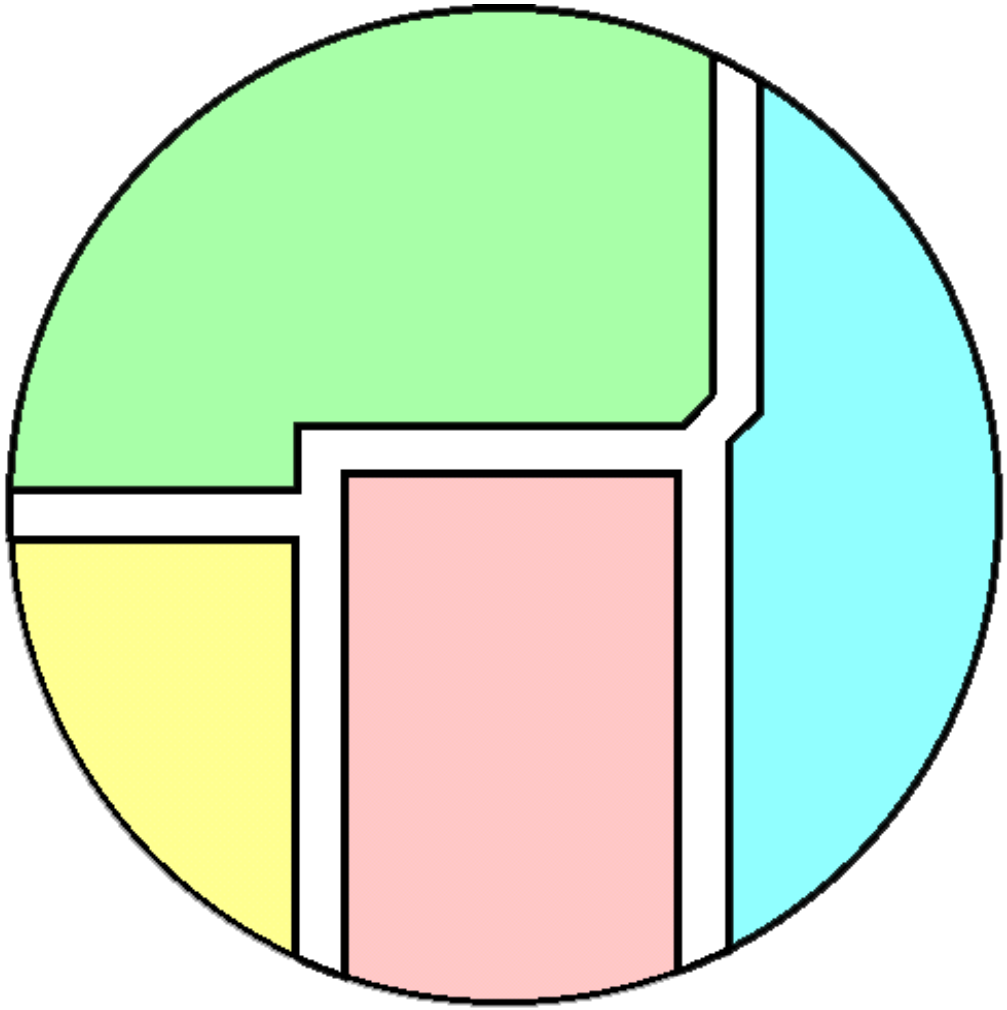
Achieving Reuse



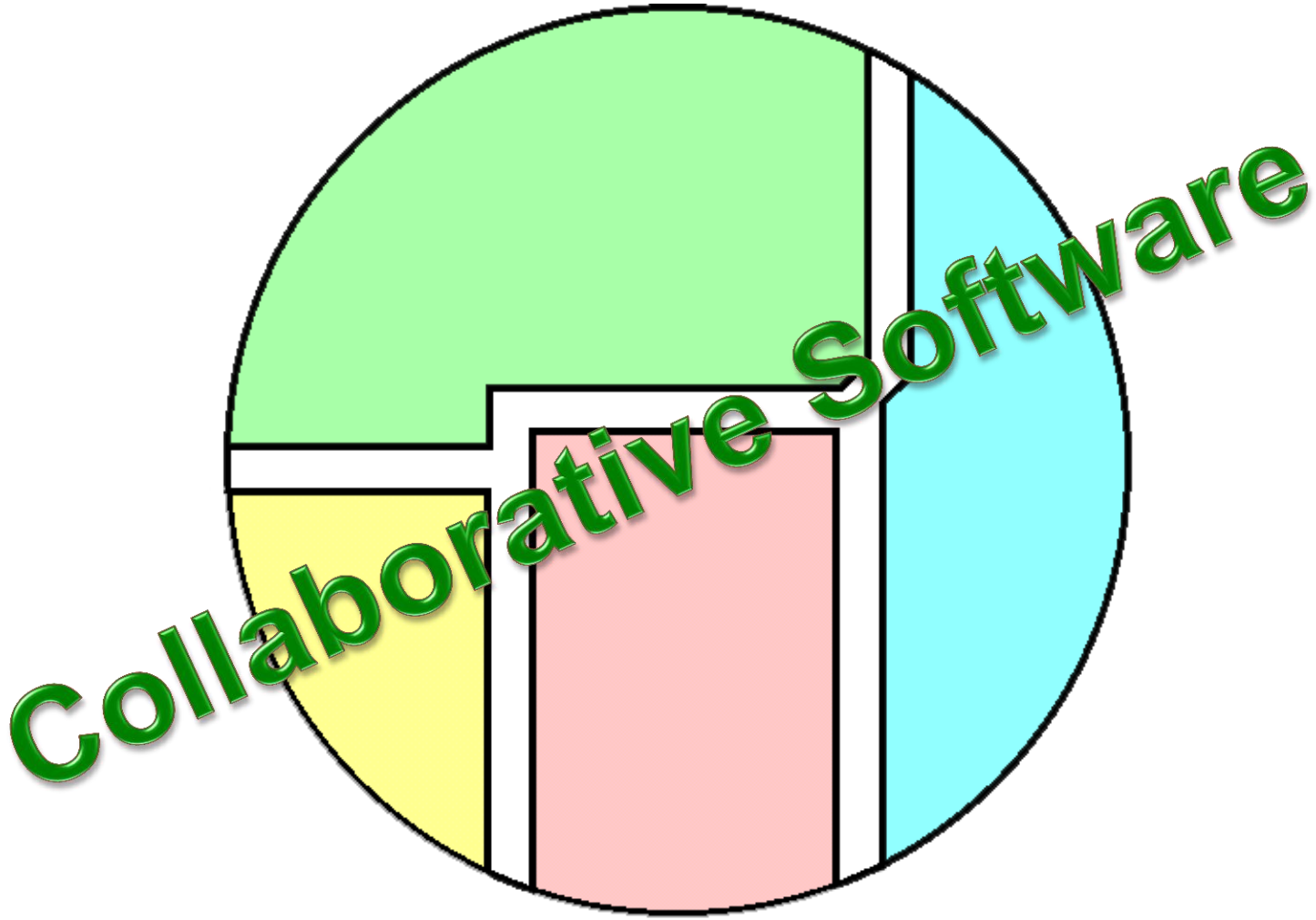
Brittle

Within just one version of a single App

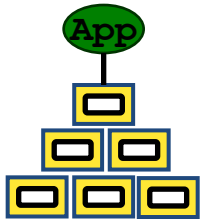
Achieving Reuse



Achieving Reuse

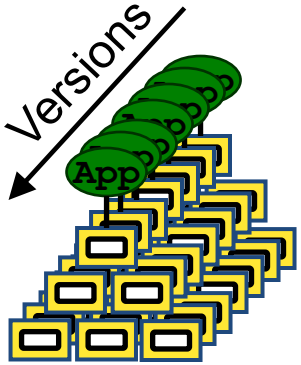


Achieving Reuse



Brittle

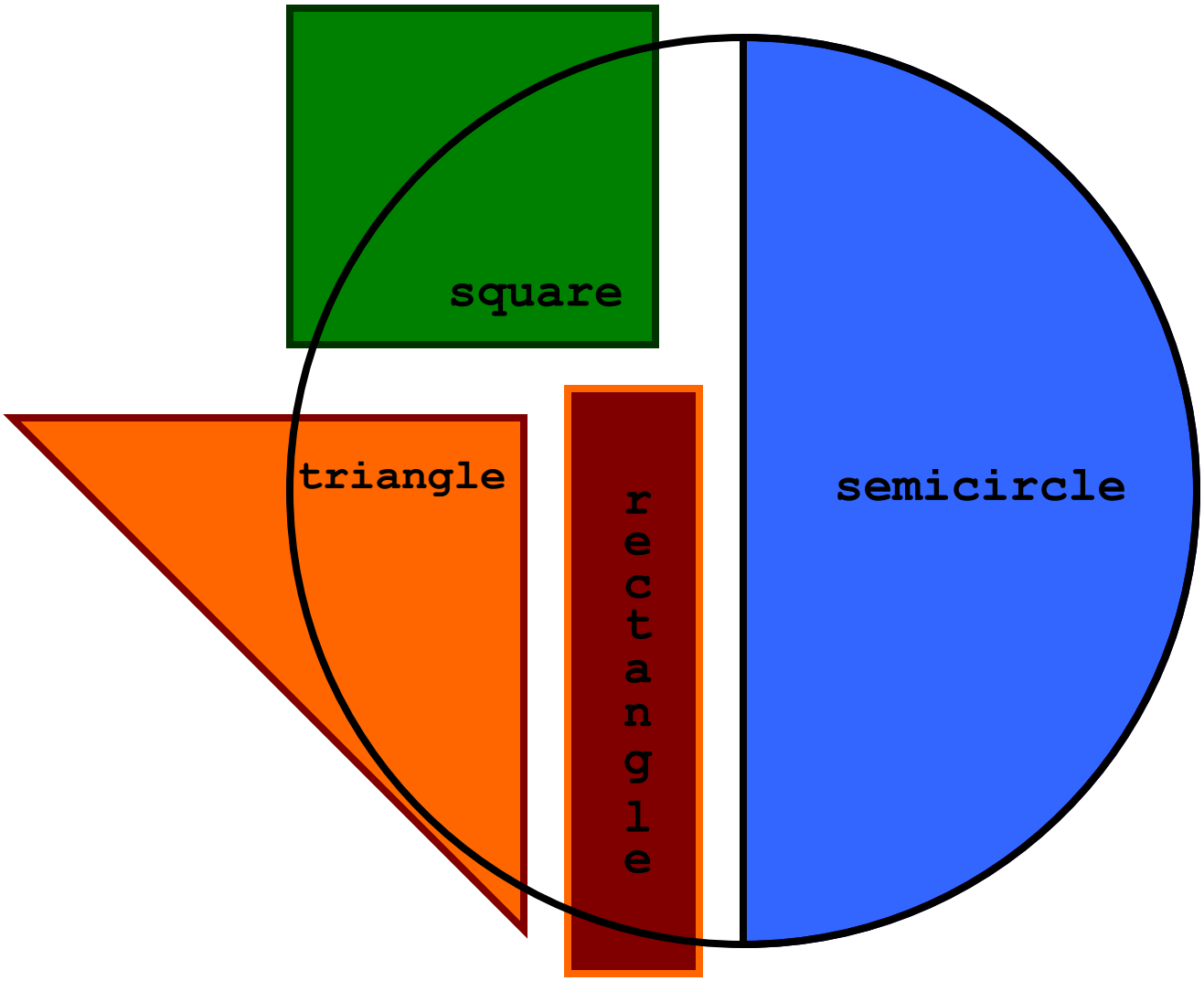
Within just one version of a single App



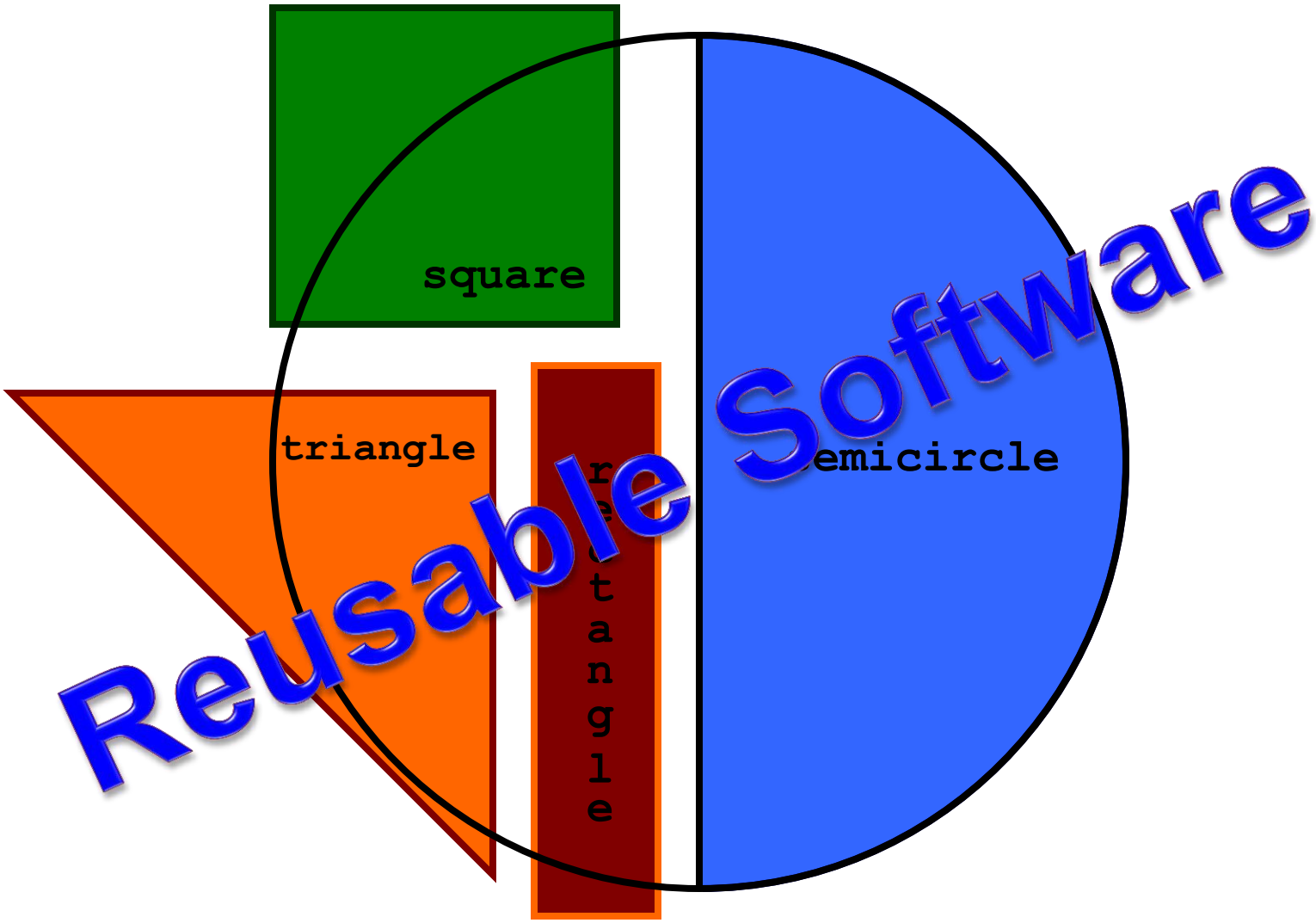
Collaborative

Across several versions of a single App

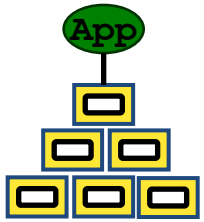
Achieving Reuse



Achieving Reuse

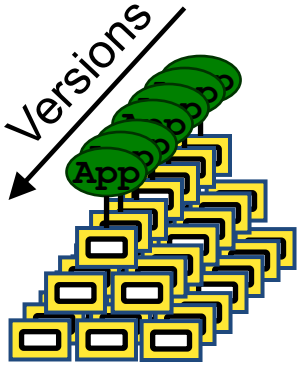


Achieving Reuse



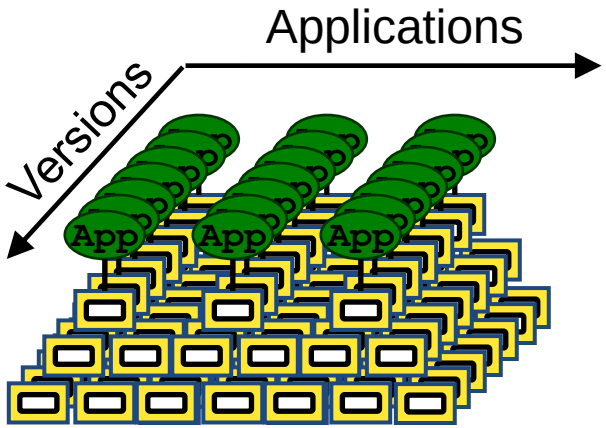
Brittle

Within just one version of a single App



Collaborative

Across several versions of a single App



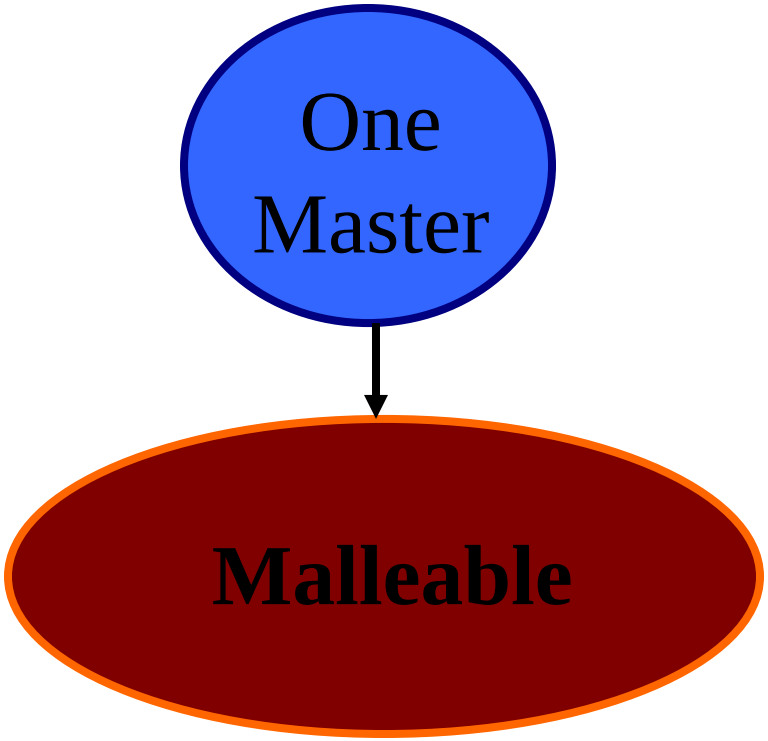
Reusable

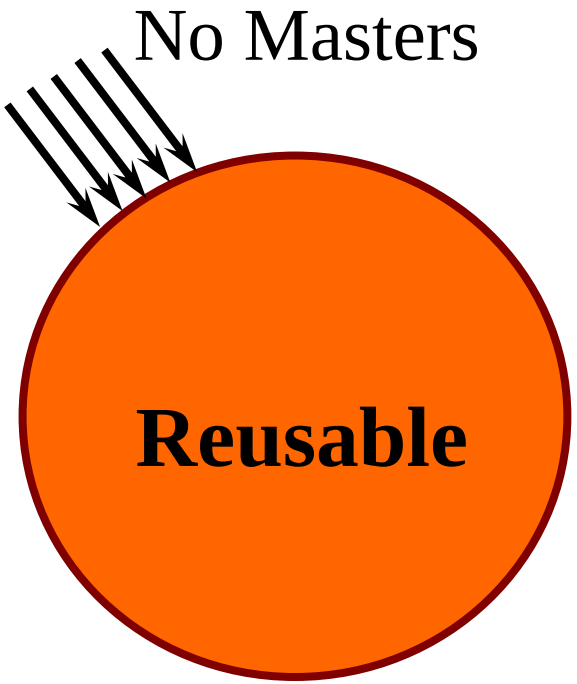
Across several versions of many distinct applications and products

Achieving Reuse

Good applications are *malleable*…

…but reusable software is ***stable***!





Achieving Fine-Grained Reuse

Fundamental properties of modular software:

Achieving Fine-Grained Reuse

Fundamental properties of modular software:

- Fine-Grained Physical Modularity

Achieving Fine-Grained Reuse

Fundamental properties of modular software:

- Fine-Grained Physical Modularity
- Logical/Physical Coherence

Achieving Fine-Grained Reuse

Fundamental properties of modular software:

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

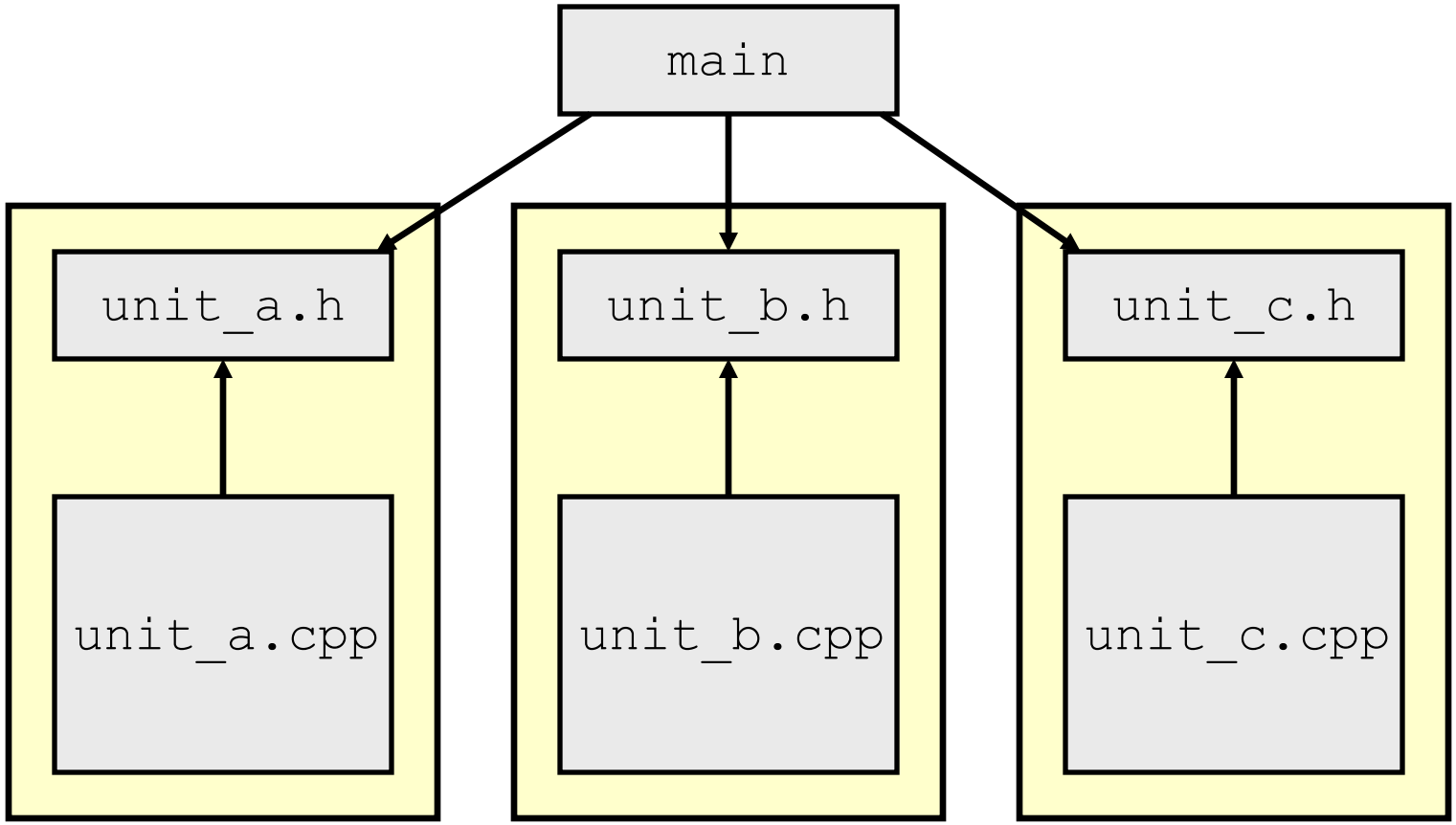
Achieving Fine-Grained Reuse

Fundamental properties of modular software:

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

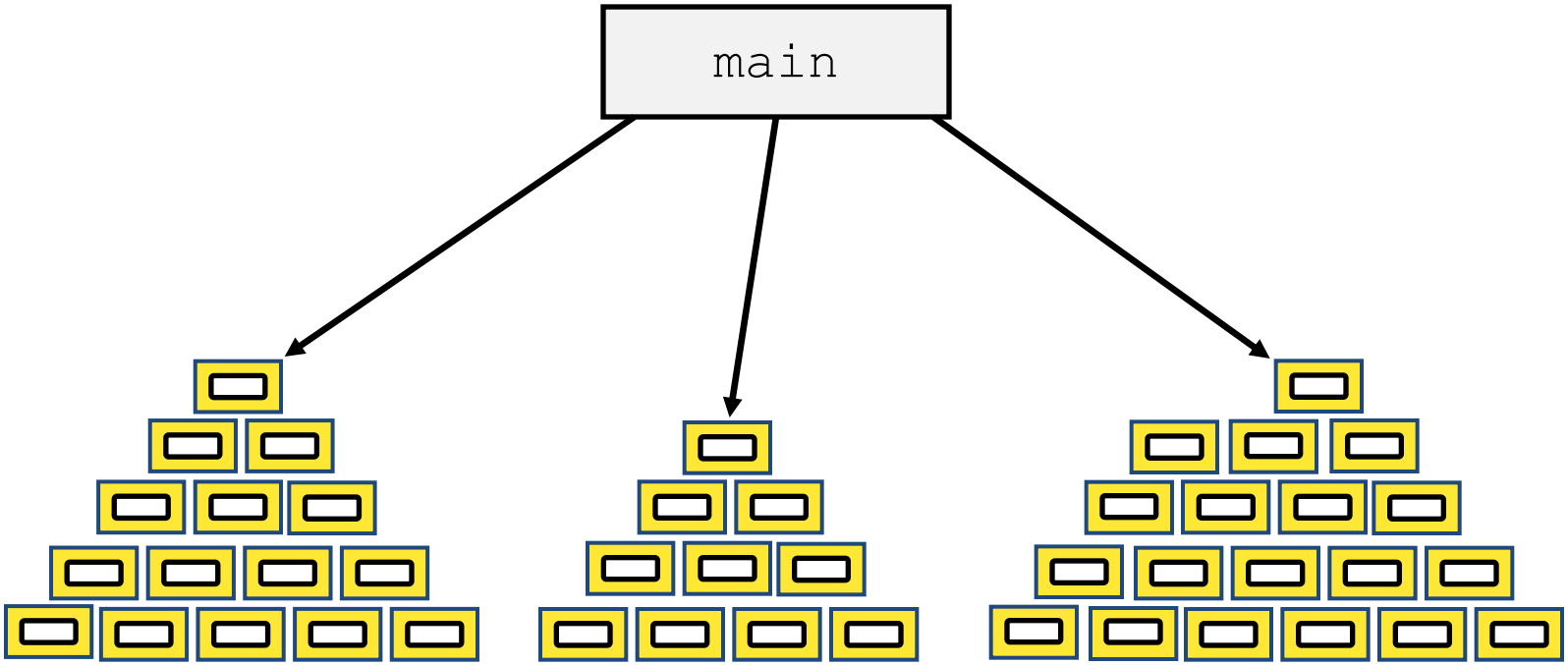
Achieving Fine-Grained Reuse

Coarse-Grained Physical Modularity



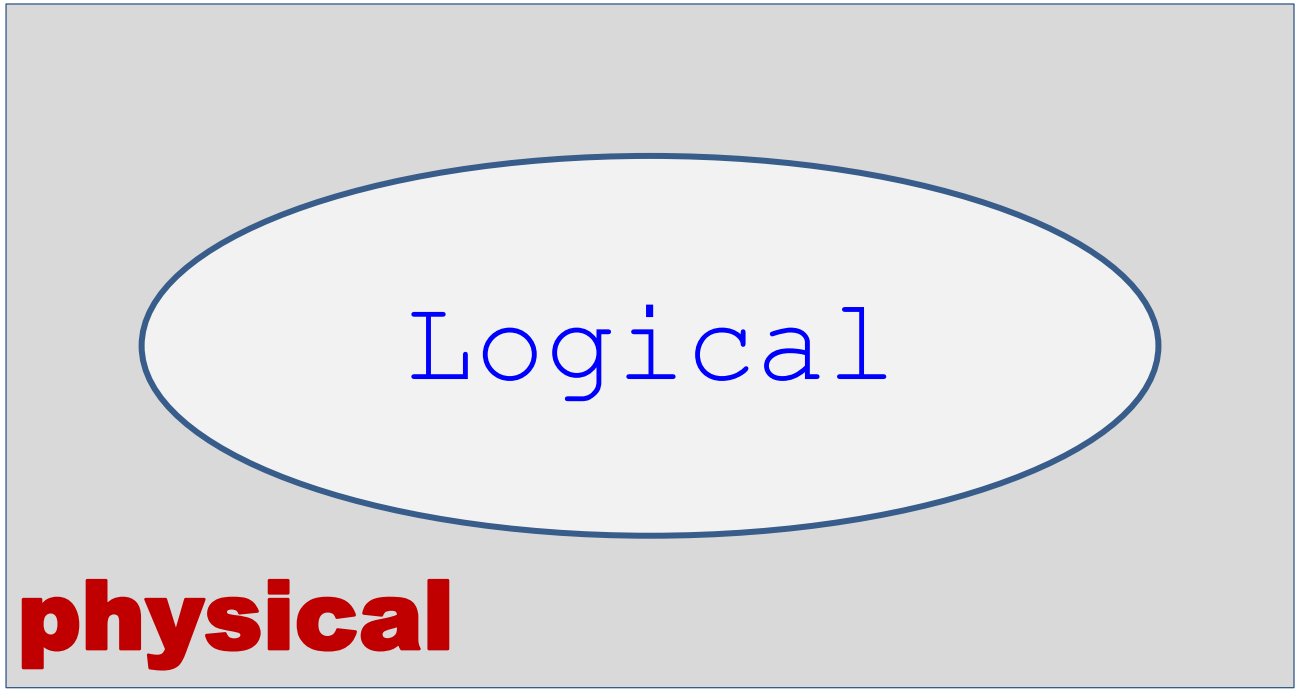
Achieving Fine-Grained Reuse

Fine-Grained Physical Modularity



Achieving Fine-Grained Reuse

Logical/Physical Coherence



Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

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

intset.h

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

stack.h

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

intset.cpp

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

// ...
```

stack.cpp

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

main.cpp

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

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

intset.h

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

stack.h

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

intset.cpp

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

// ...
```

stack.cpp

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

main.cpp

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

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

intset.h

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

stack.h

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

intset.cpp

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

// ...
```

?

stack.cpp

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

main.cpp

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

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

intset.h

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

stack.h

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

intset.cpp

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

// ...
```

?

stack.cpp

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

main.cpp

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

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

intset.h

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

stack.h

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

intset.cpp

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

// ...
```

?

stack.cpp

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

main.cpp

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

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

intset.h

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

stack.h

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

intset.cpp

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

// ...
```

?

stack.cpp

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

main.cpp

Achieving Fine-Grained Reuse

No Logical/Physical Incoherence

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

intset.h

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

stack.h

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

intset.cpp

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

// ...
```

?

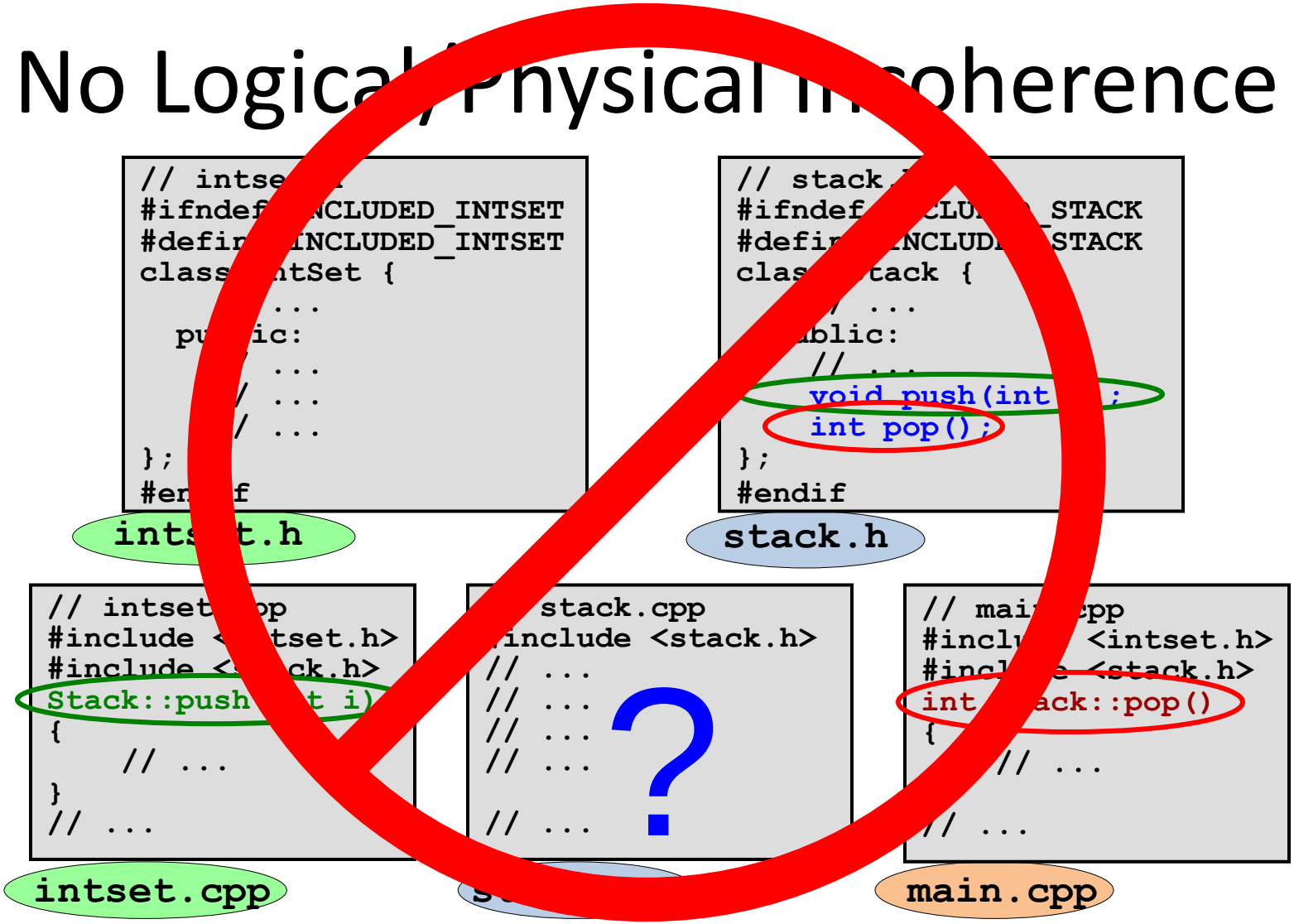
stack.cpp

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

main.cpp

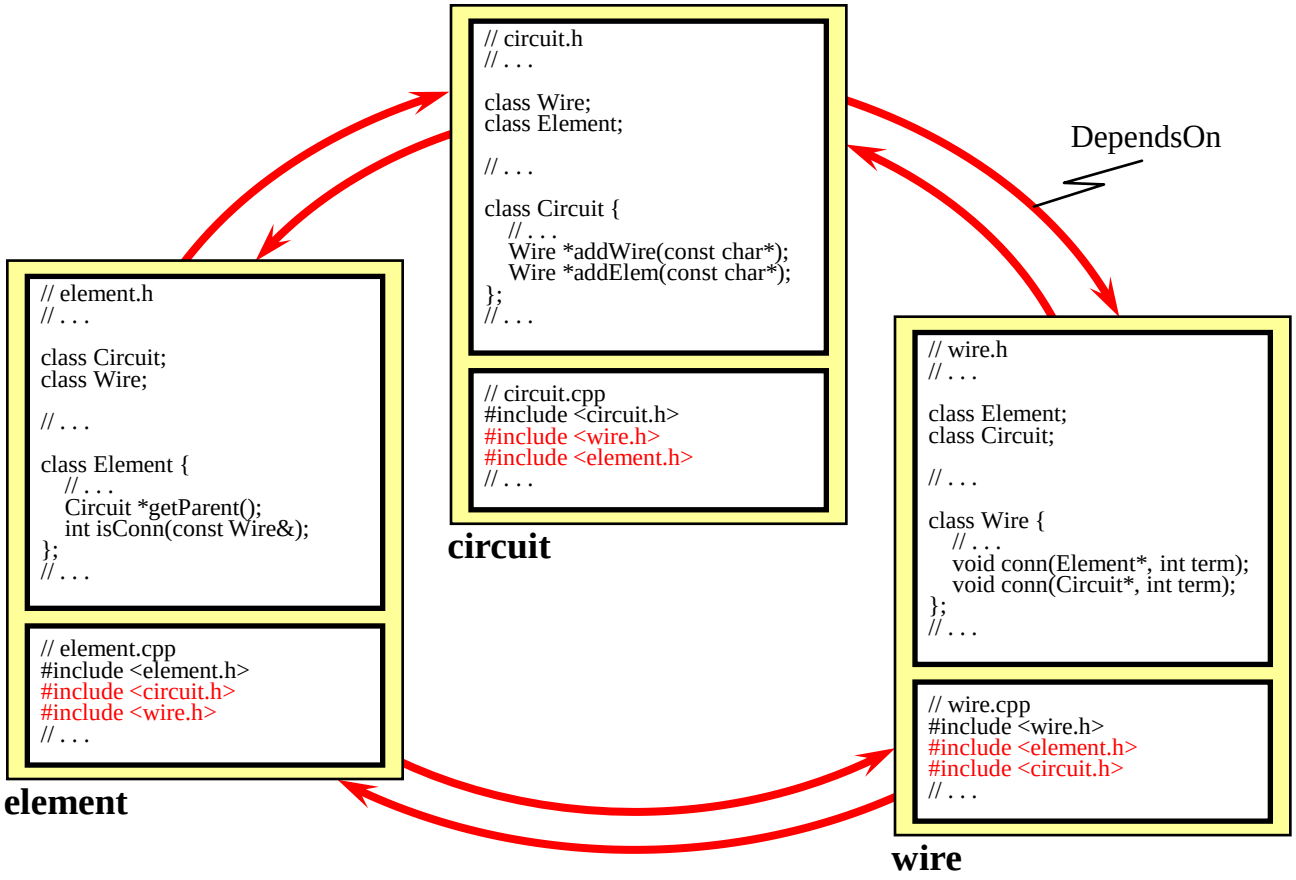
Achieving Fine-Grained Reuse

No Logical/Physical Incoherence



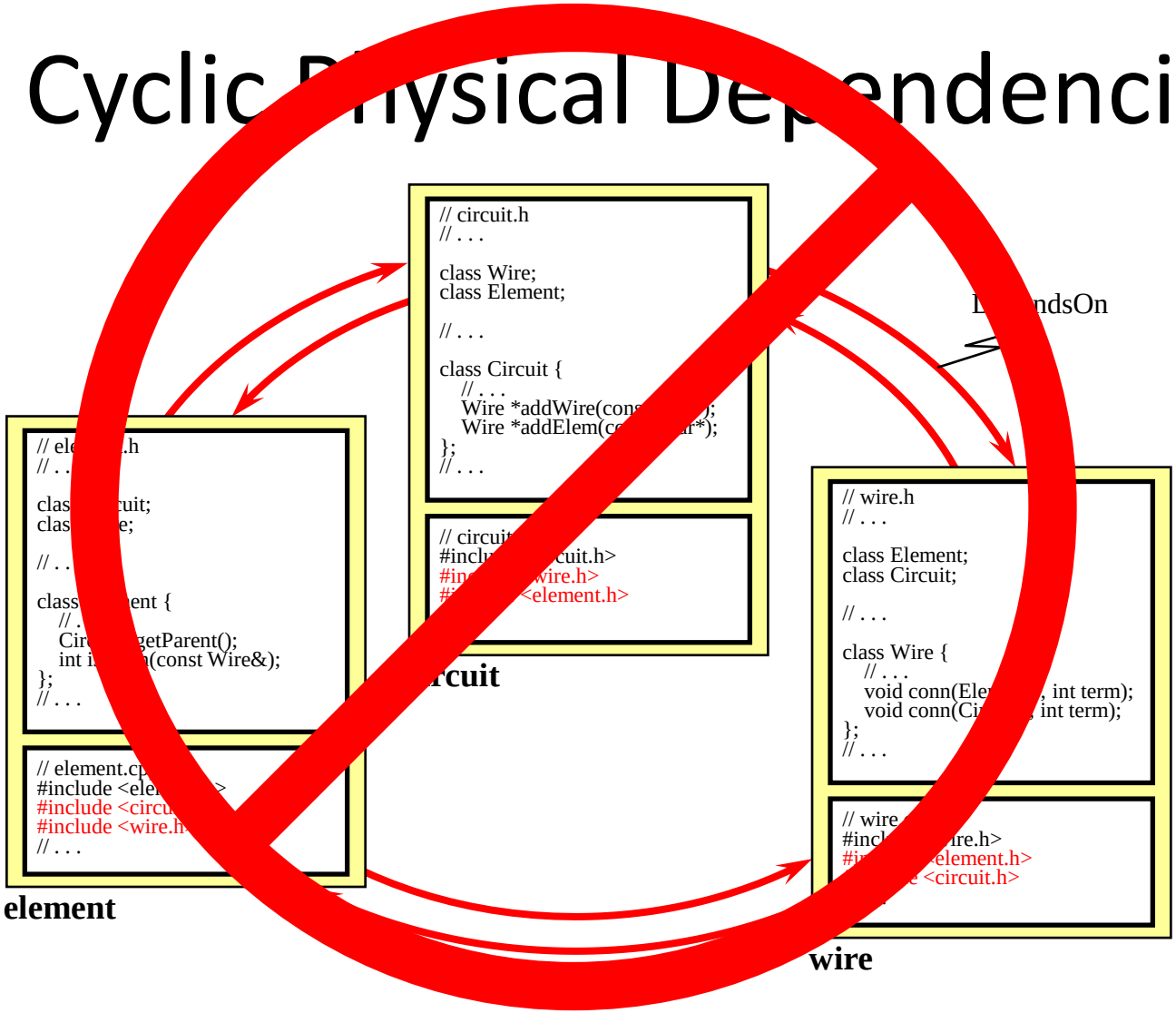
Achieving Fine-Grained Reuse

No Cyclic Physical Dependencies



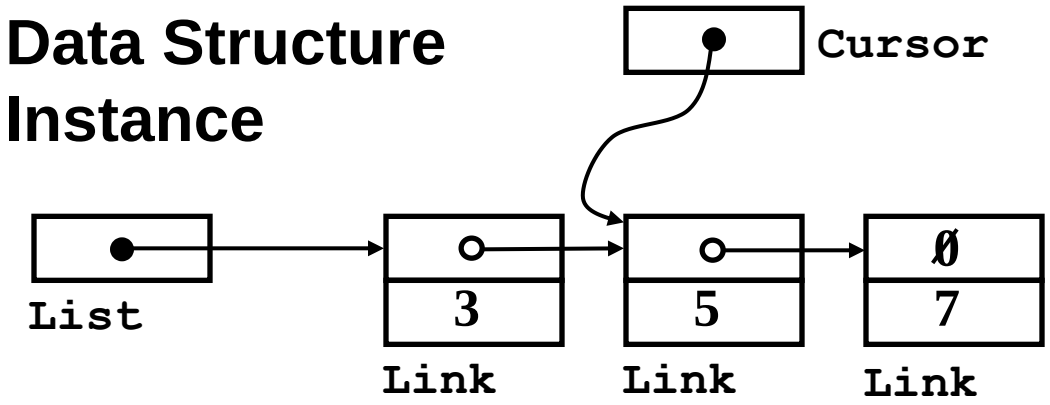
Achieving Fine-Grained Reuse

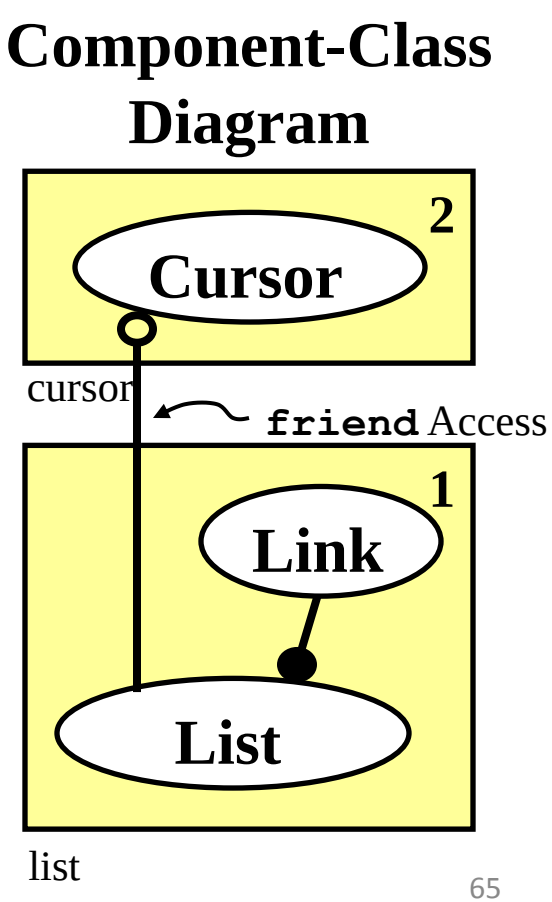
No Cyclic Physical Dependencies



Achieving Fine-Grained Reuse

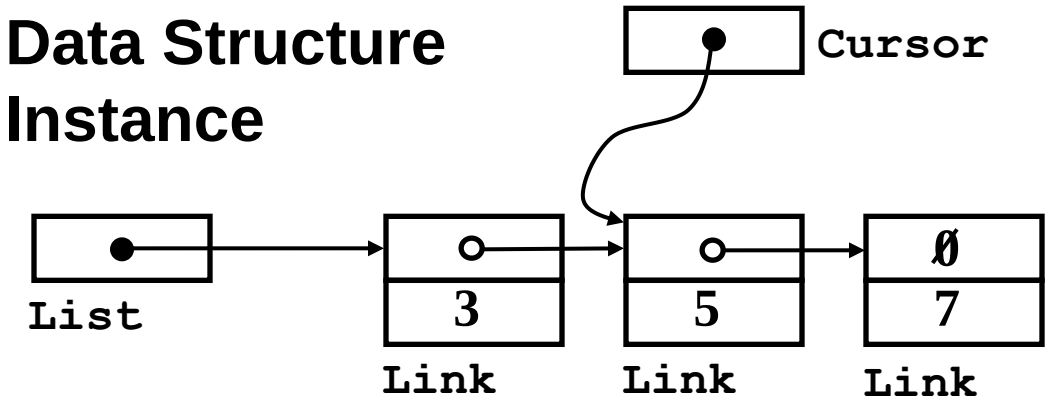
No Private "Back Door" Access

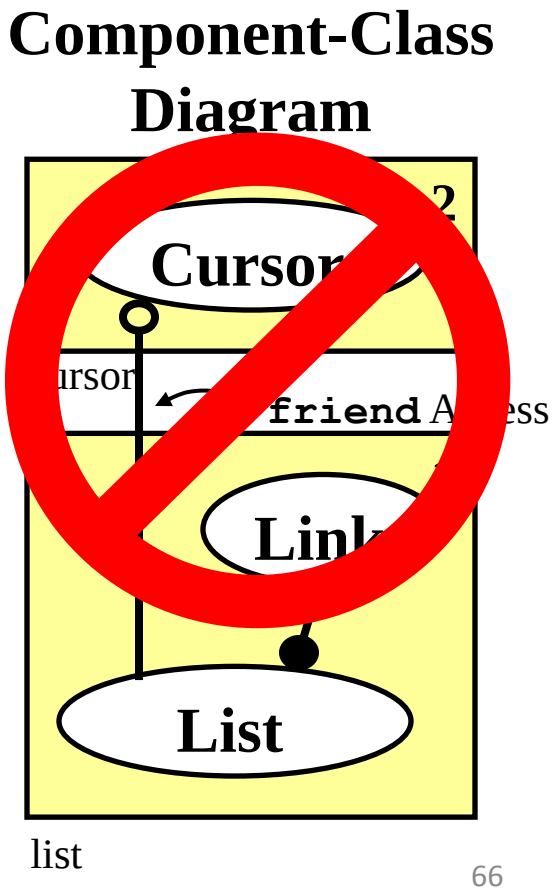




Achieving Fine-Grained Reuse

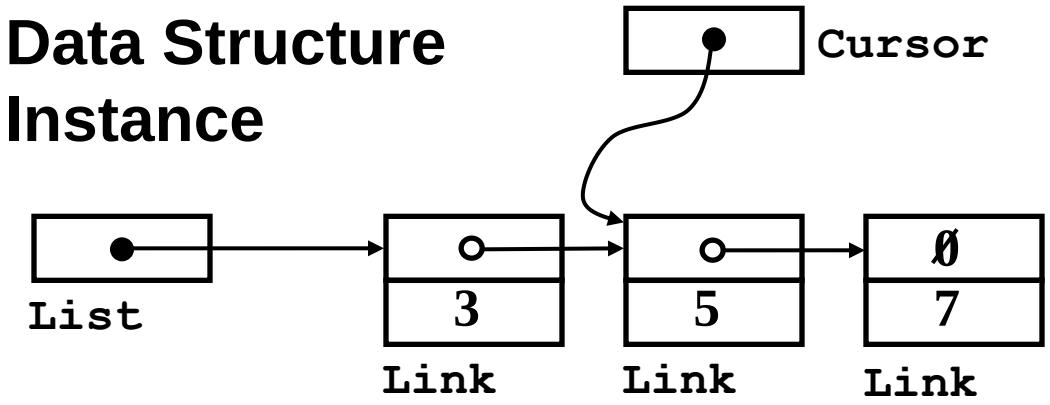
No Private "Back Door" Access

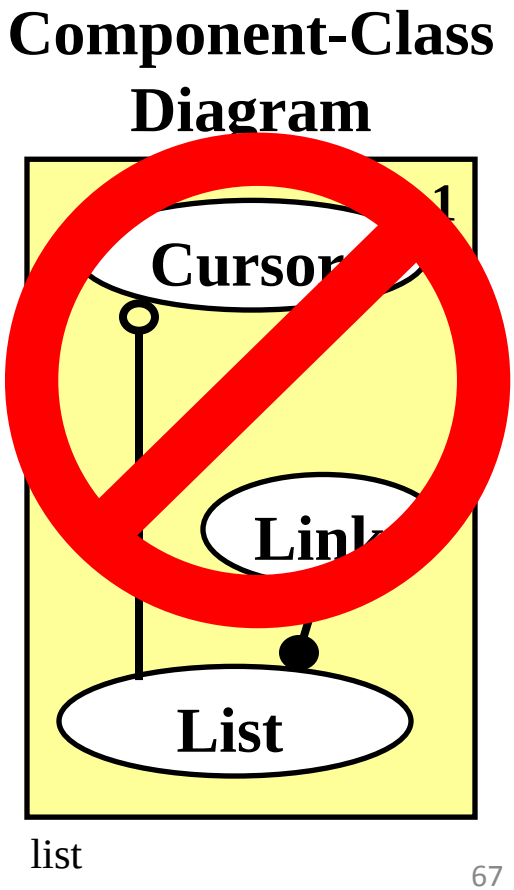




Achieving Fine-Grained Reuse

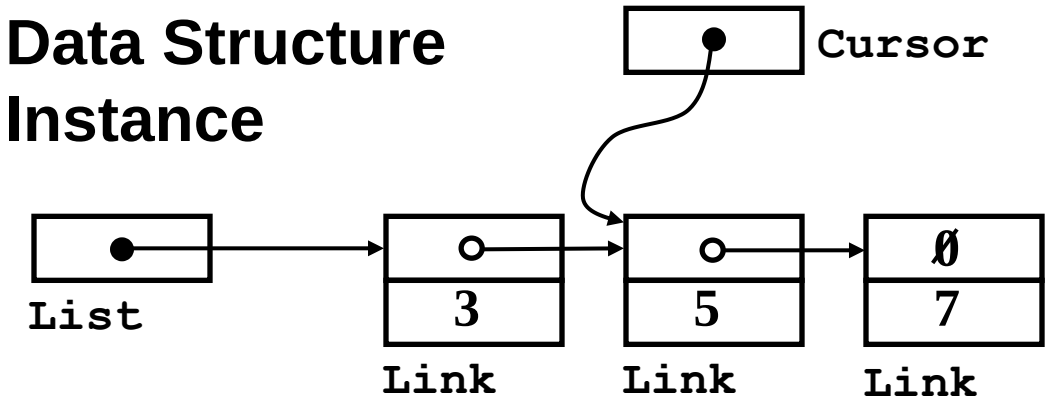
No Private "Back Door" Access

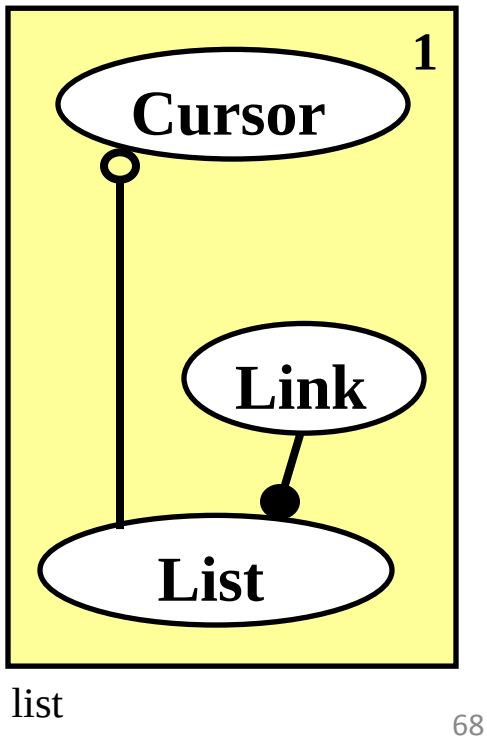




Achieving Fine-Grained Reuse

No Private "Back Door" Access



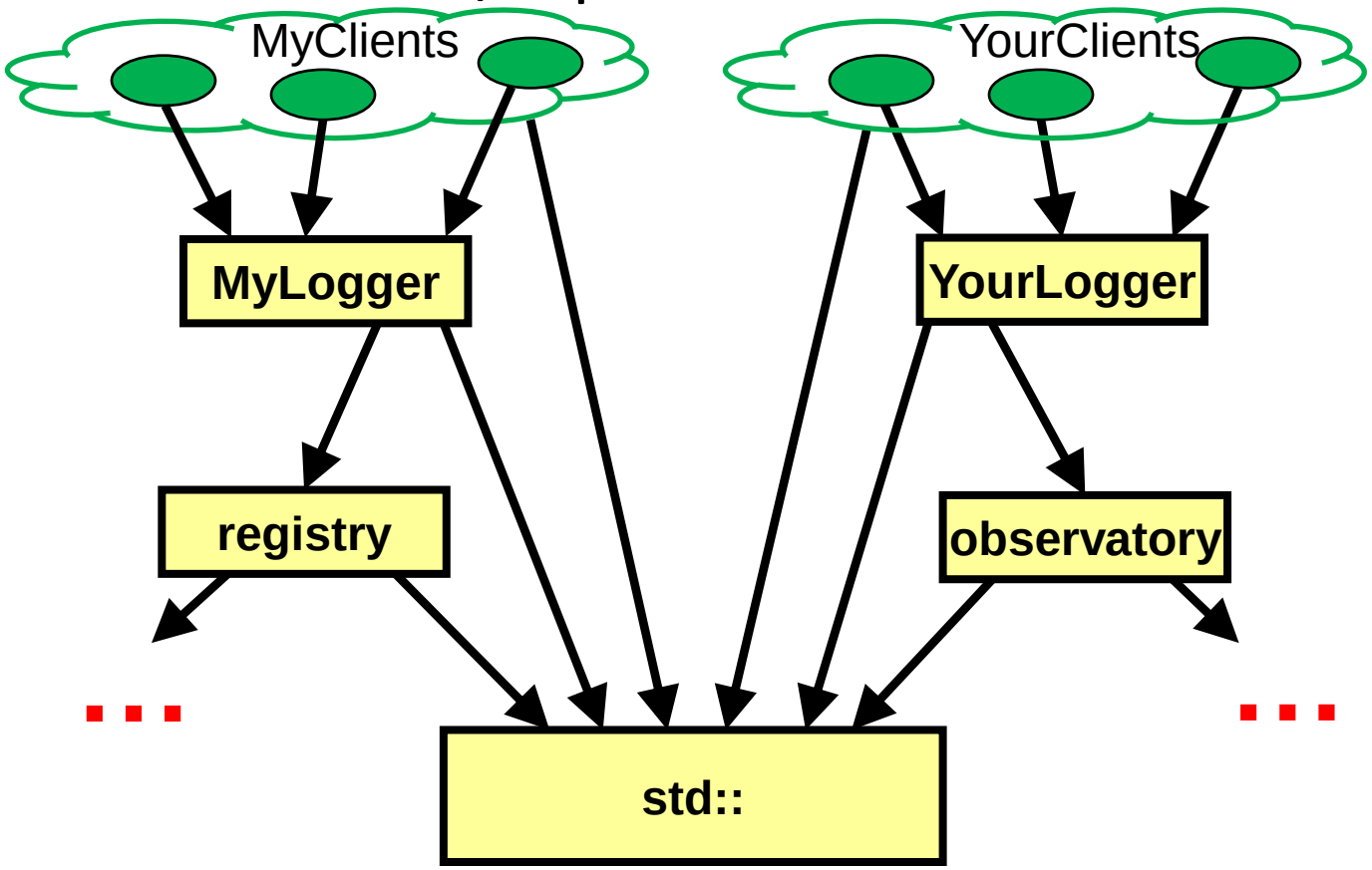


Achieving Hierarchical Reuse

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

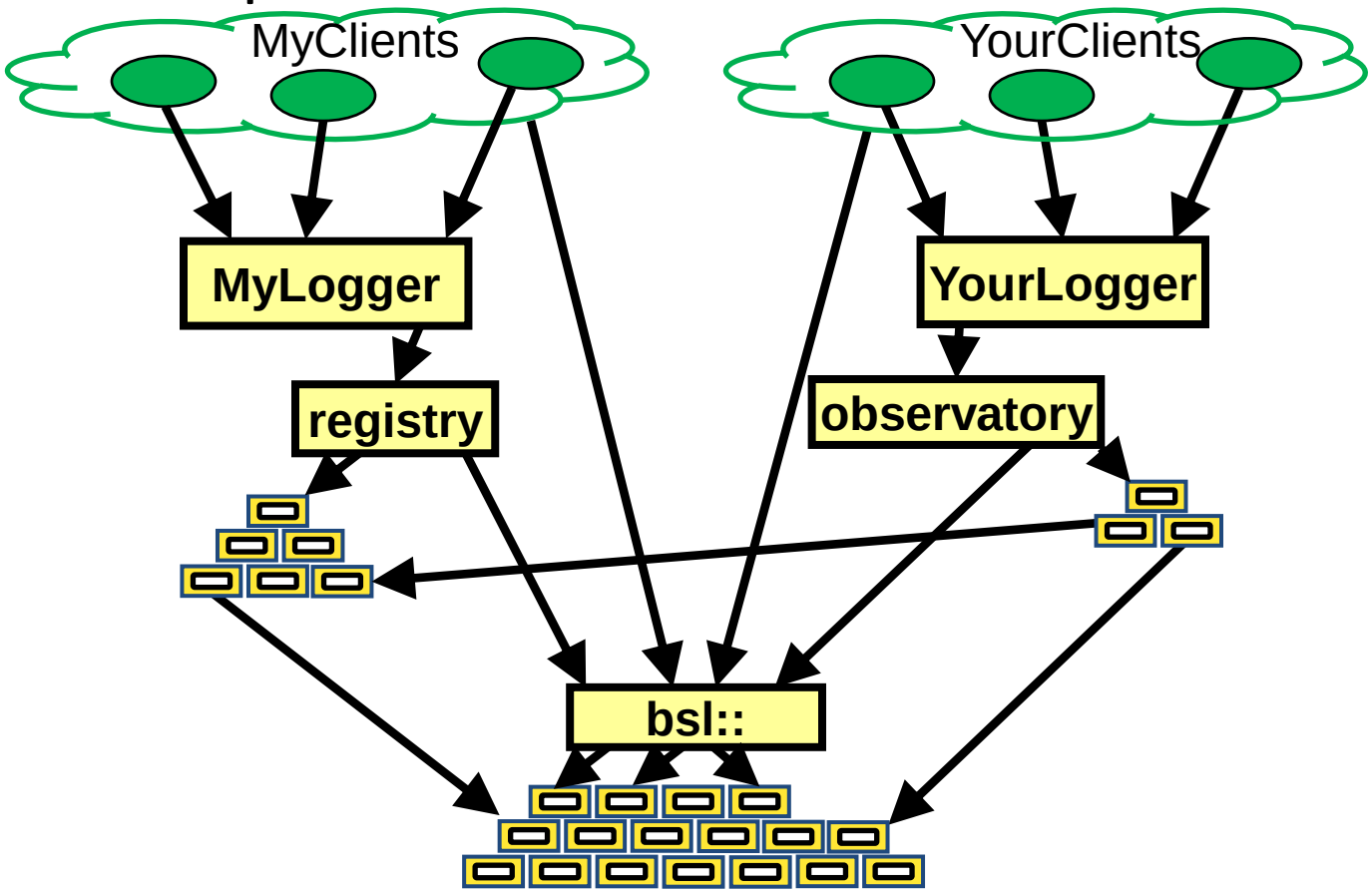
Achieving Hierarchical Reuse

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



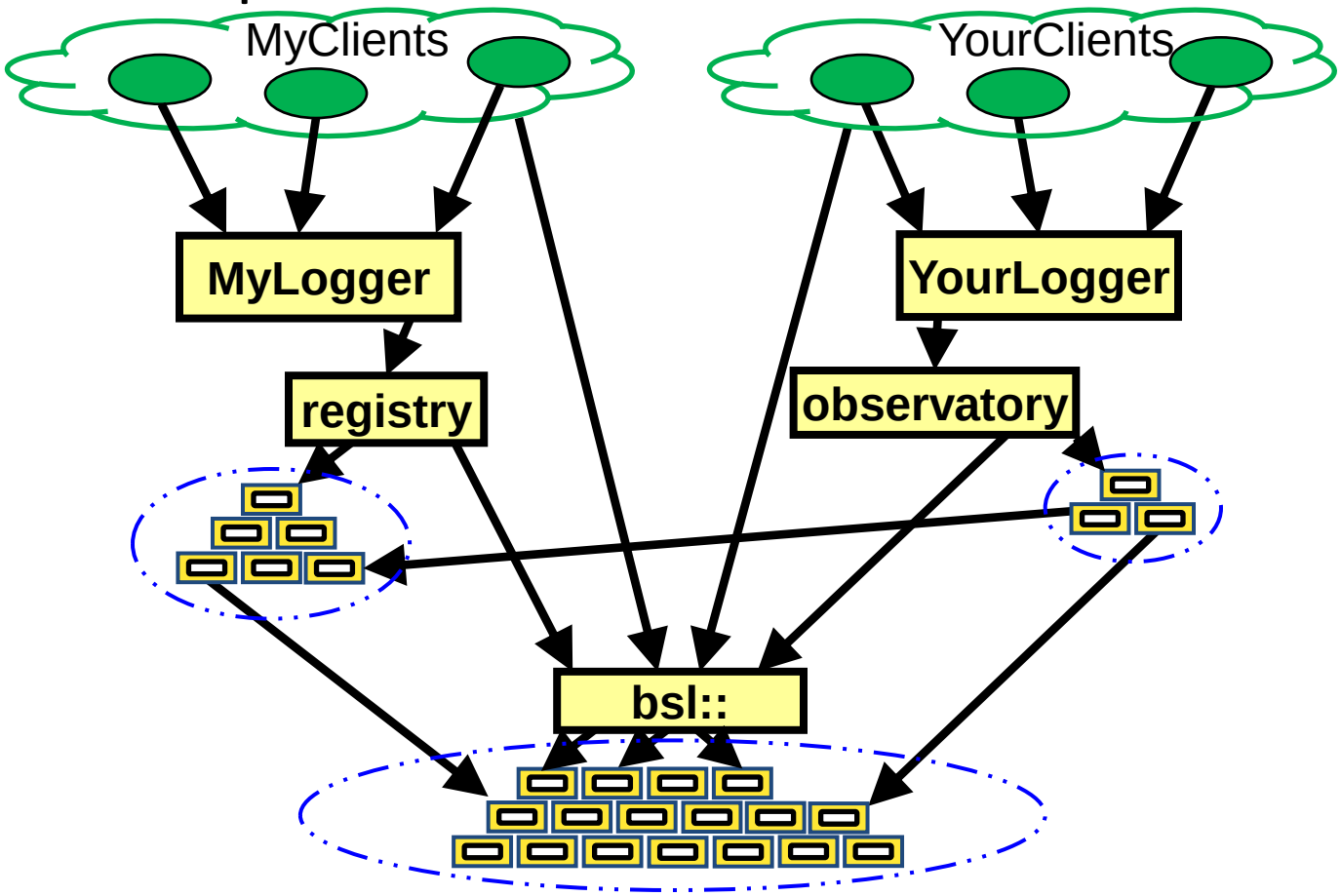
Achieving Hierarchical Reuse

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



Achieving Hierarchical Reuse

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



Achieving Hierarchical Reuse

Application Software:

Library Software:

Solutions

Sub-Solutions

Sub-Sub-Solutions

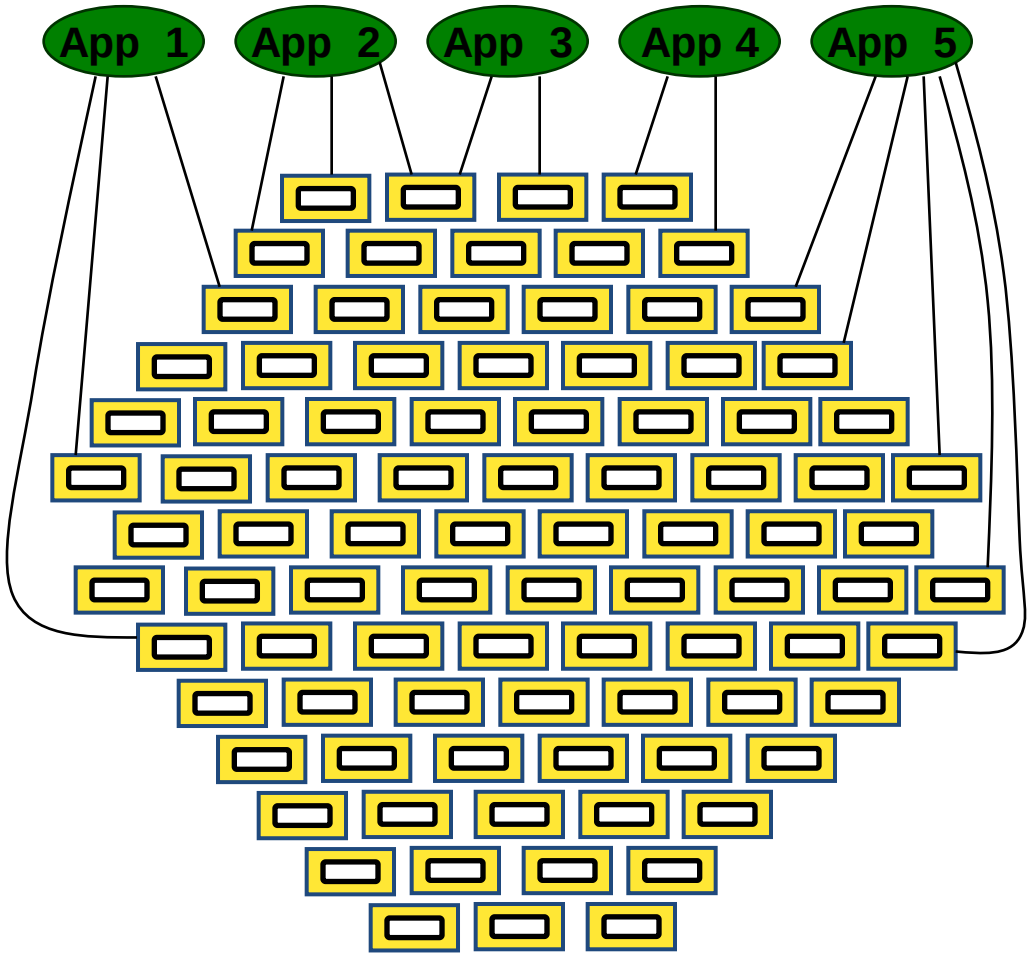
Vocabulary-Type Utilities

Vocabulary Types

Implementation Utilities

Low-Level Interfaces

Platform Adapters



Achieving Wide-Spread Reuse

As library developers, we must

Achieving Wide-Spread Reuse

As library developers, we must:

- Draw complexity inward; push simplicity outward.

Achieving Wide-Spread Reuse

As library developers, we must:

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

Achieving Wide-Spread Reuse

As library developers, we must:

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

Achieving Wide-Spread Reuse

As library developers, we must:

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

Achieving Wide-Spread Reuse

To the maximum extent practicable…

…every software component we write must be:

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

Achieving Wide-Spread Reuse

1. Easy to Understand

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

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use

- Effective usage model.
- Intuitive interface.
- Appropriate level of safety.
- Minimal physical dependencies.

Achieving Wide-Spread Reuse

1. Easy to Understand
2. Easy to Use
3. High Performance

- Execution (i.e., wall and CPU) run time.
- Process (i.e., in-core memory) size.
- Compile time (or the degree of compile-time coupling).
- Link time (or the extent of link-time dependency).
- Executable (i.e., on-disk) code size.

Achieving Wide-Spread Reuse

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

- Builds on all supported platforms.
- Runs on all supported platforms.
- Produces the same results on all supported platforms.
- Achieves "reasonable" performance on all supported platforms.

Achieving Wide-Spread Reuse

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

Achieving Wide-Spread Reuse

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

- No core dumps.

0. Goals

Achieving Wide-Spread Reuse

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

- No core dumps.
- No memory leaks.

Achieving Wide-Spread Reuse

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

- No core dumps.
- No memory leaks.
- No incorrect results.

Achieving Wide-Spread Reuse

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

- No core dumps.
- No memory leaks.
- No incorrect results.
- No bugs!

Achieving Wide-Spread Reuse

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

- No core dumps.
- No memory leaks.
- No incorrect results.
- No bugs!
- No, we're not kidding.

Achieving Wide-Spread Reuse

Wait a minute…

Just how good does software need to be?

Achieving Wide-Spread Reuse

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

Achieving Wide-Spread Reuse

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

- It must adequately perform its function.

Achieving Wide-Spread Reuse

Achieving Wide-Spread Reuse

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

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

0. Goals

Achieving Wide-Spread Reuse

Achieving Wide-Spread Reuse

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

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

0. Goals

Achieving Wide-Spread Reuse

0. Goals

Achieving Wide-Spread Reuse

Achieving Wide-Spread Reuse

0. Goals

Achieving Wide-Spread Reuse

Achieving Wide-Spread Reuse

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

Achieving Wide-Spread Reuse

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

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

– Lakos1x

Achieving Wide-Spread Reuse

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

—Van Snyder (Brooks95)

Achieving Wide-Spread Reuse

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

—Van Snyder (Brooks95)

Achieving Wide-Spread Reuse

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

—Van Snyder (Brooks95)

Achieving Wide-Spread Reuse

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

—Van Snyder (Brooks95)

Achieving Wide-Spread Reuse

Reusable library software:

Achieving Wide-Spread Reuse

Reusable library software:

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

Achieving Wide-Spread Reuse

Reusable library software:

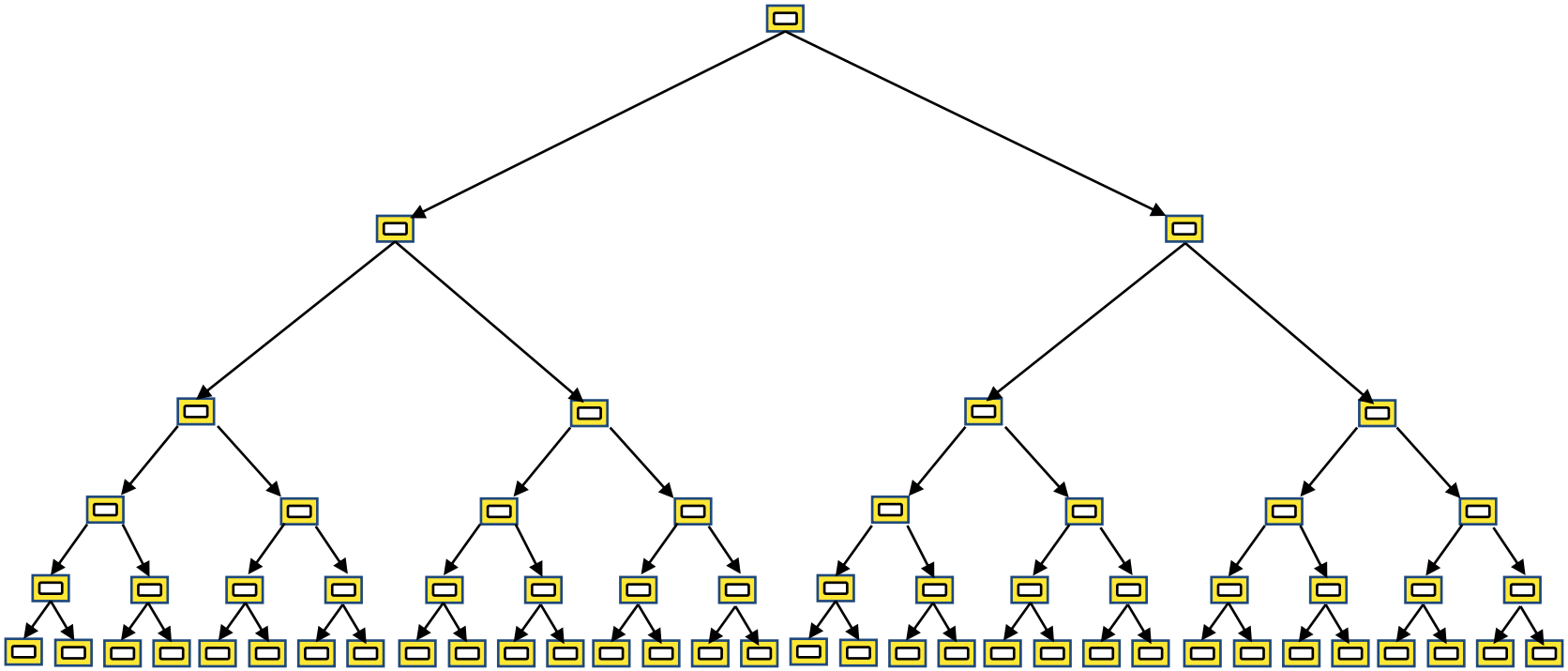
- Must be perceived as ***far*** better than what a prospective client (or anyone else) could do in any practical time frame.
- Unlike a house (or an App), can be consumed by many different (kinds of) clients.

Achieving Wide-Spread Reuse

Reusable library software:

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

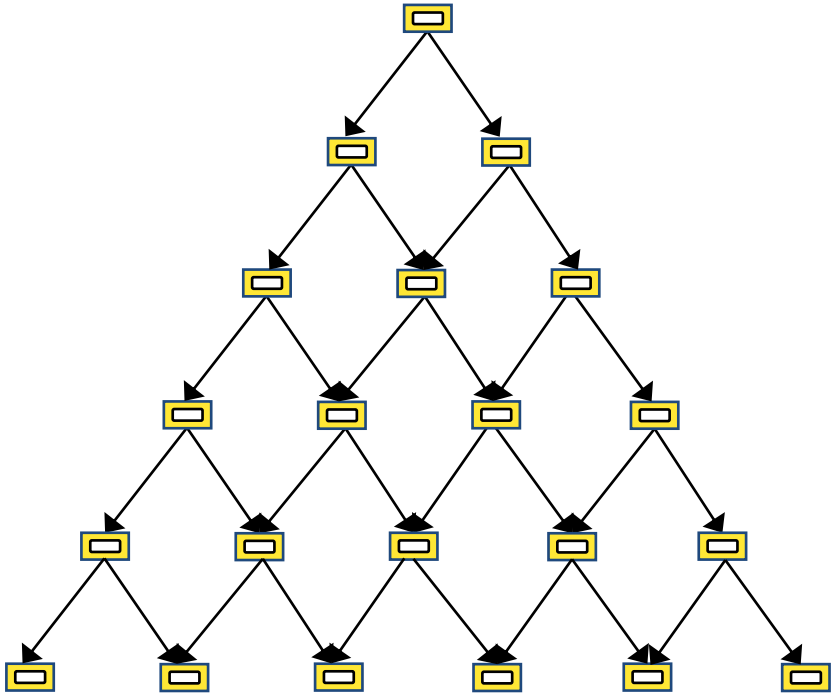
Achieving Wide-Spread Reuse



No Re-convergence

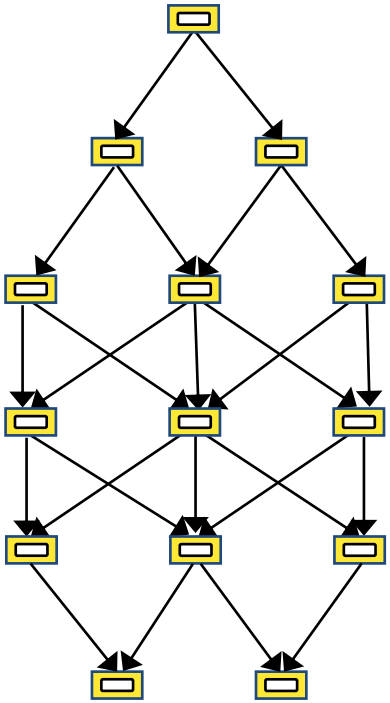
0. Goals

Achieving Wide-Spread Reuse



Significant Re-convergence

Achieving Wide-Spread Reuse



Maximal Re-convergence

Achieving Wide-Spread Reuse

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

Achieving Wide-Spread Reuse

Achieving Wide-Spread Reuse

Nothing Succeeds Like Excess!

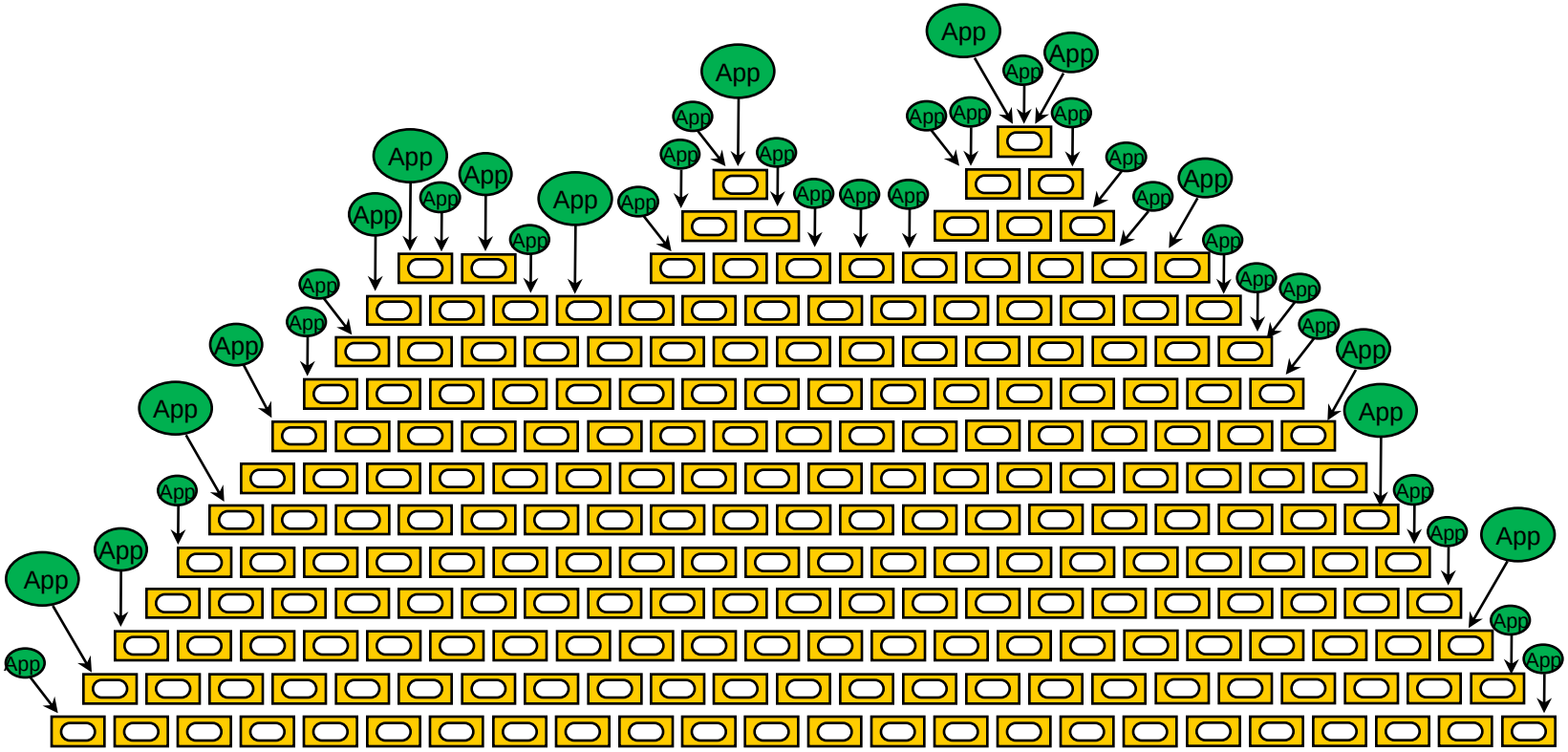
Achieving Wide-Spread Reuse

Nothing Succeeds Like Excess!

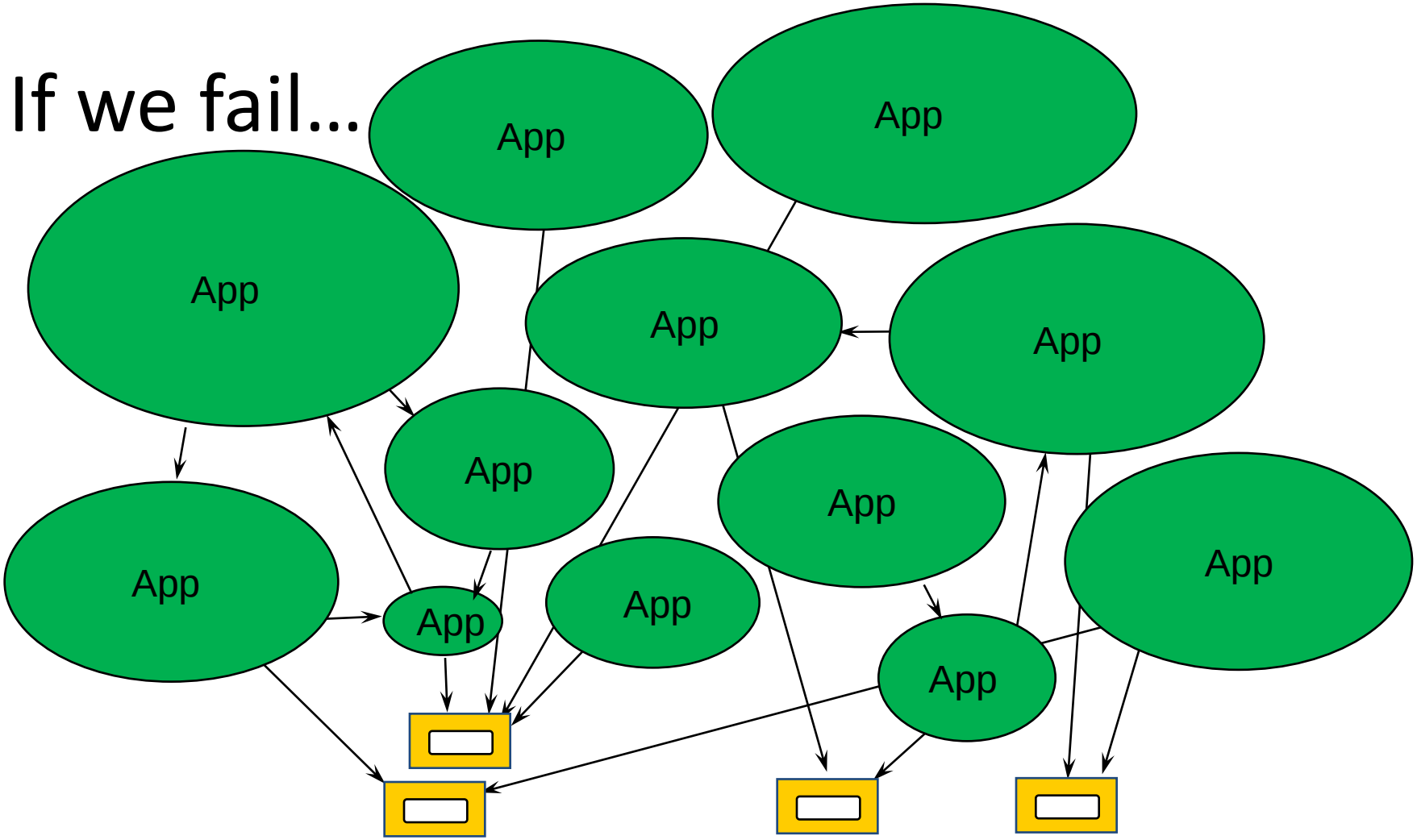
(If it's worth doing, it's worth overdoing.)

Achieving Wide-Spread Reuse

If we succeed…

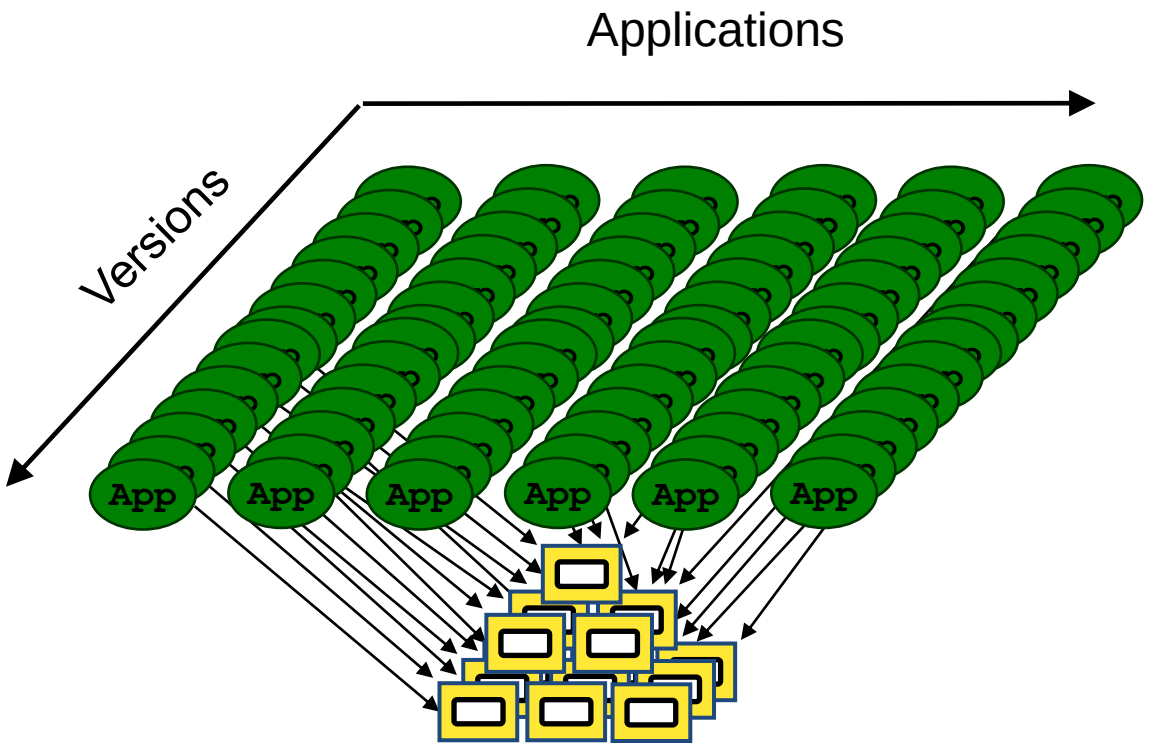


Achieving Wide-Spread Reuse

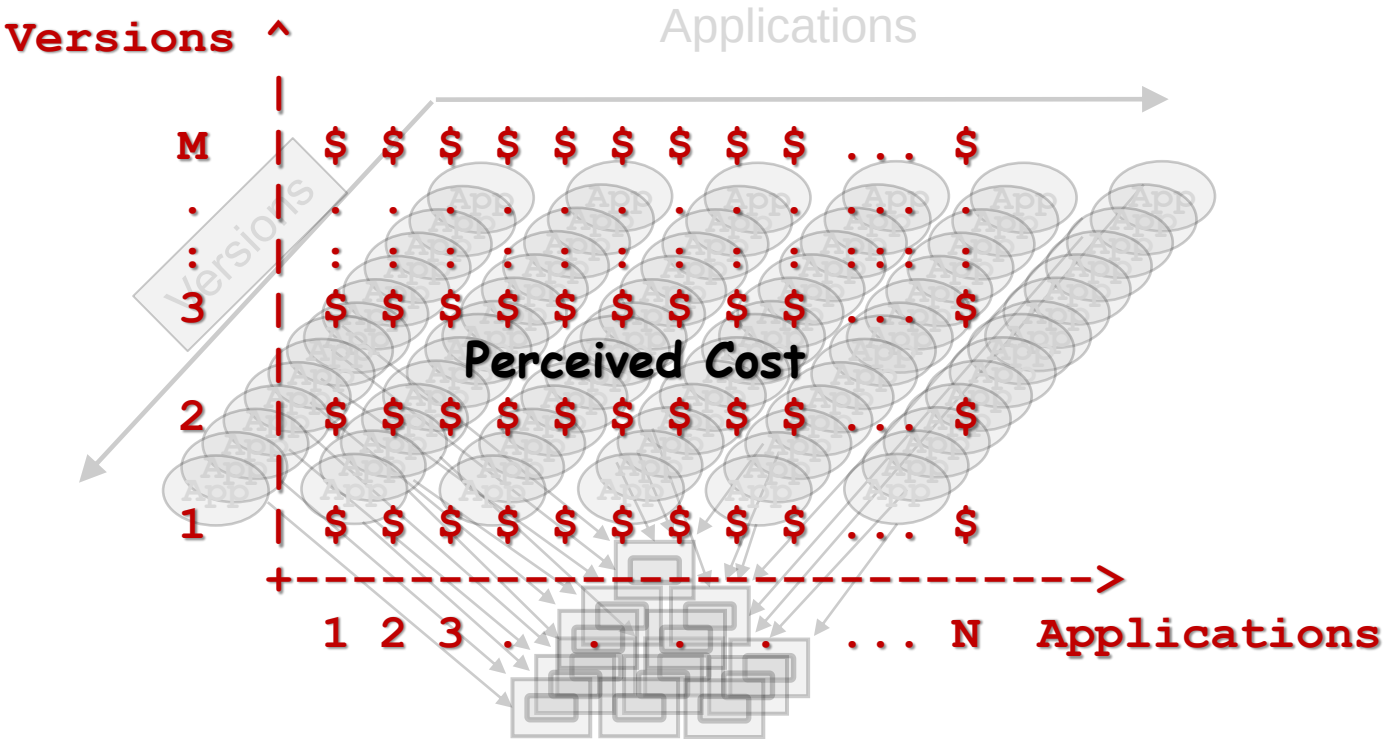


Achieving Wide-Spread Reuse

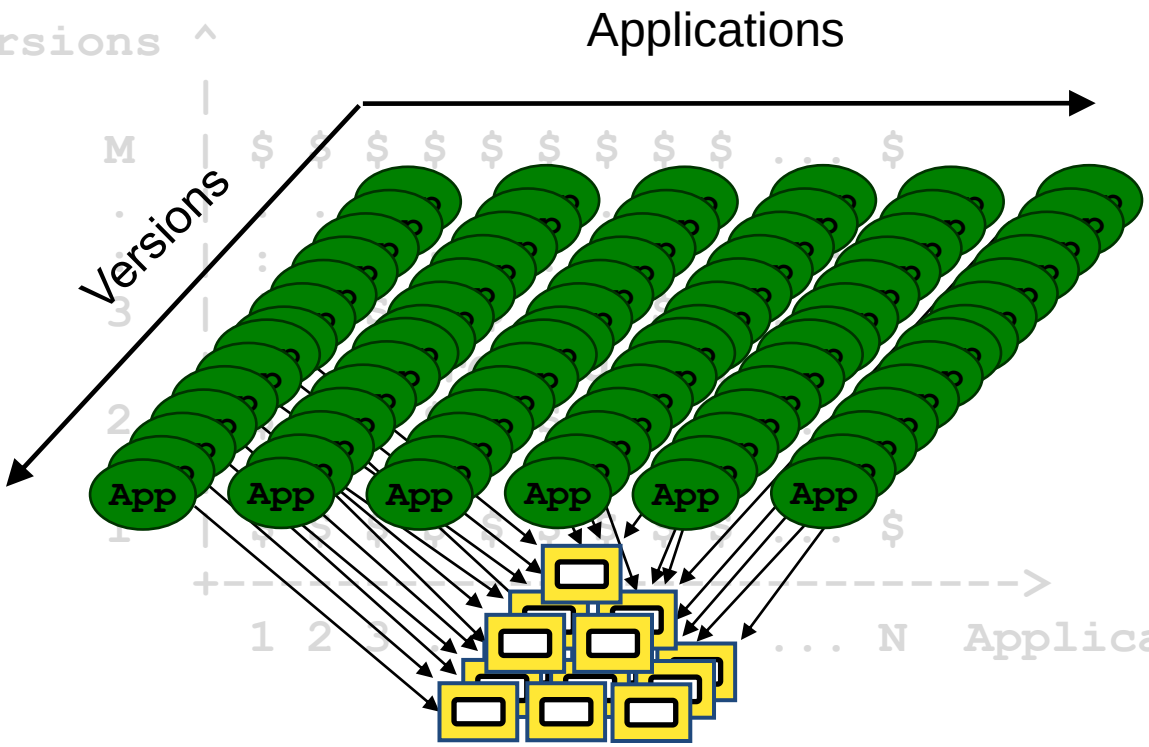
Fortunately, we can amortize the *perceived* cost over many **products X versions**:



Achieving Wide-Spread Reuse

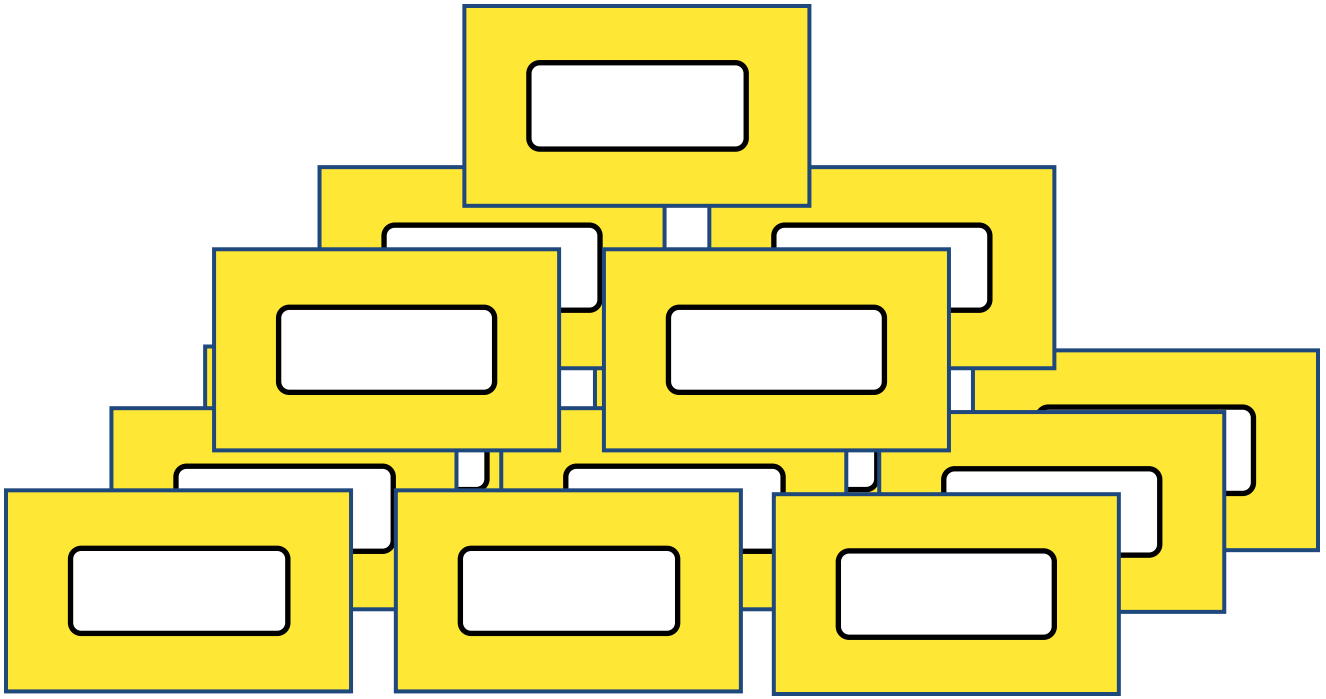


Achieving Wide-Spread Reuse



Achieving Wide-Spread Reuse

Hierarchically Reusable Software



Achieving Wide-Spread Reuse

Hierarchically Reusable Software

a.k.a.:

Outline

0. Goals

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

1. Process & Architecture

   Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

   Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

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

4. Bloomberg Development Environment

   Rendered as Fine-Grained *Hierarchically* *Reusable* Components.

Outline

0. Goals

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

1. Process & Architecture

   Organizing Software as Components, Packages, & Package Groups.

2. Design & Implementation

   Using Class Categories, Value Semantics, & Vocabulary Types.

3. Verification & Testing

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

4. Bloomberg Development Environment

   Rendered as Fine-Grained *Hierarchically Reusable* Components.

Introduction

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

Introduction

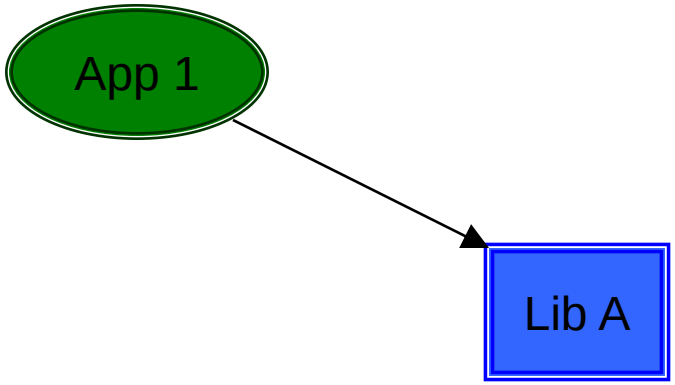
All of the software we write is governed by a common overarching set of

Organizing Principles.

Among the most central of which is achieving

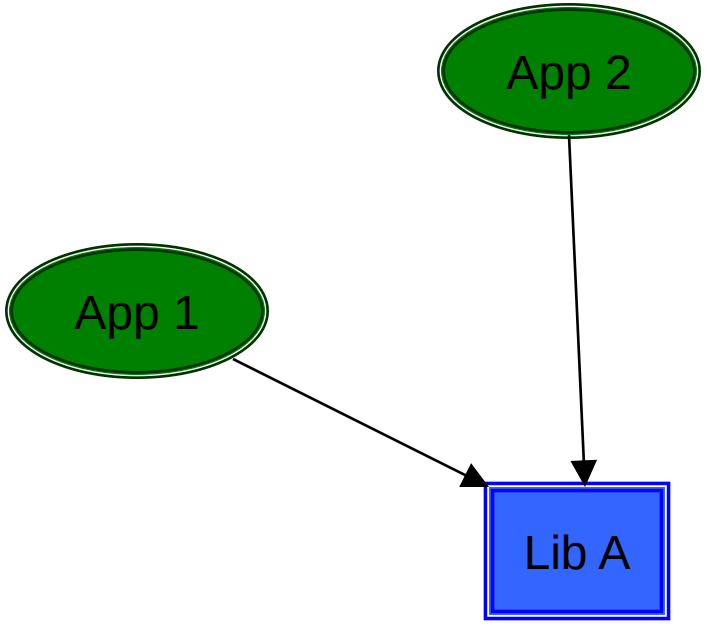
Sound Physical Design.

Creating a Big Ball of Mud



Creating a Big Ball of Mud

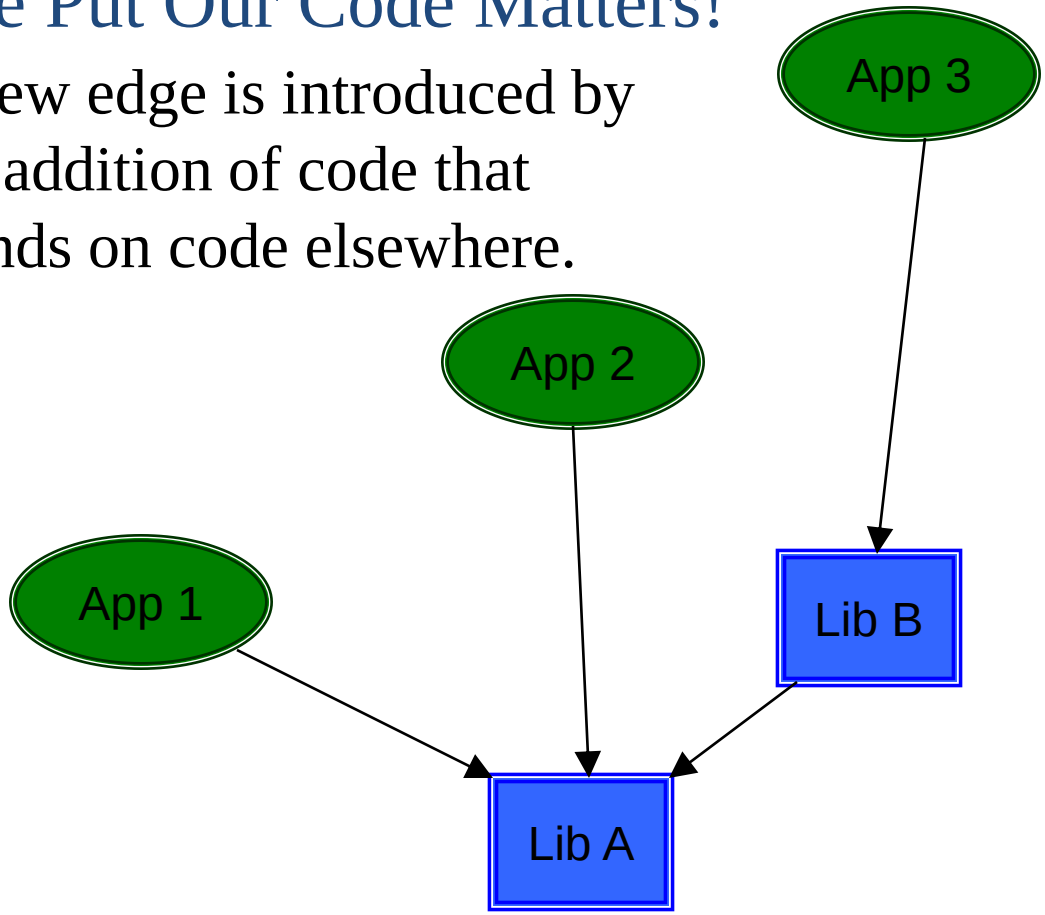
Where We Put Our Code Matters!



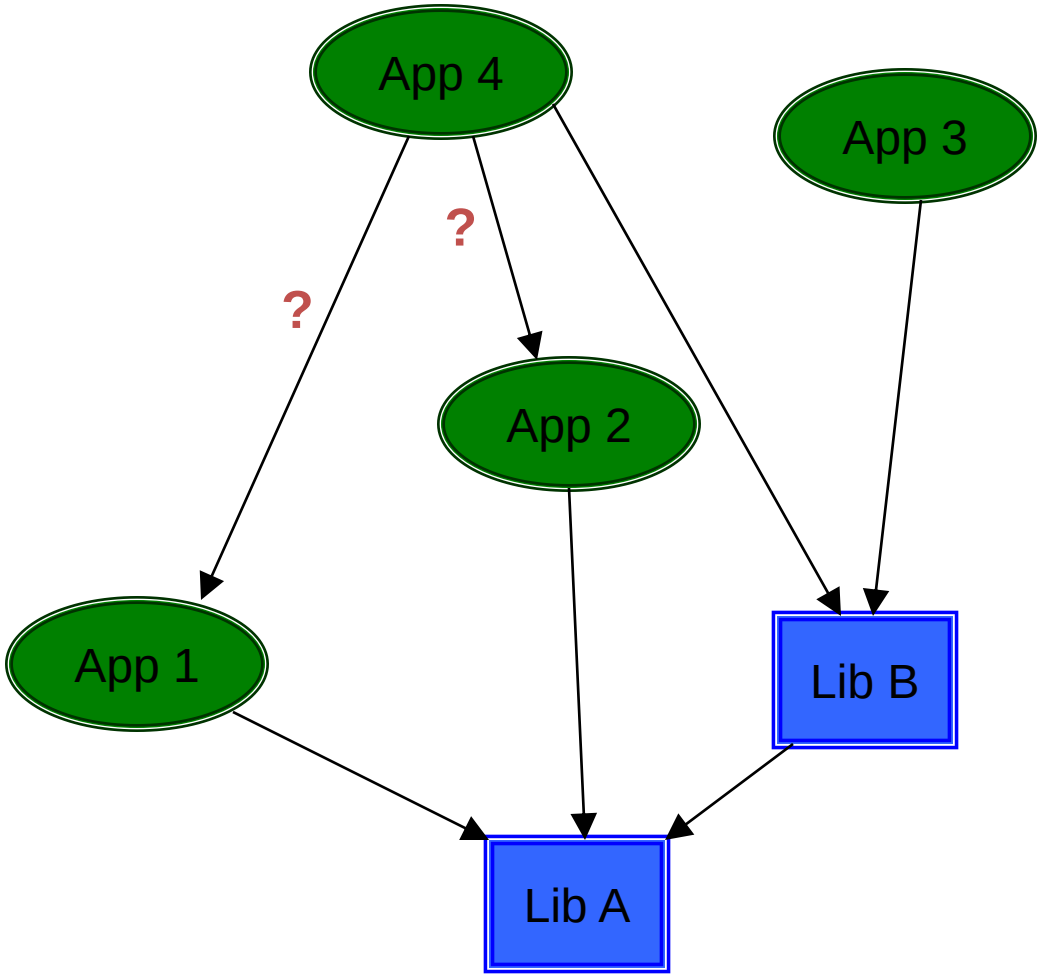
Creating a Big Ball of Mud

Where We Put Our Code Matters!

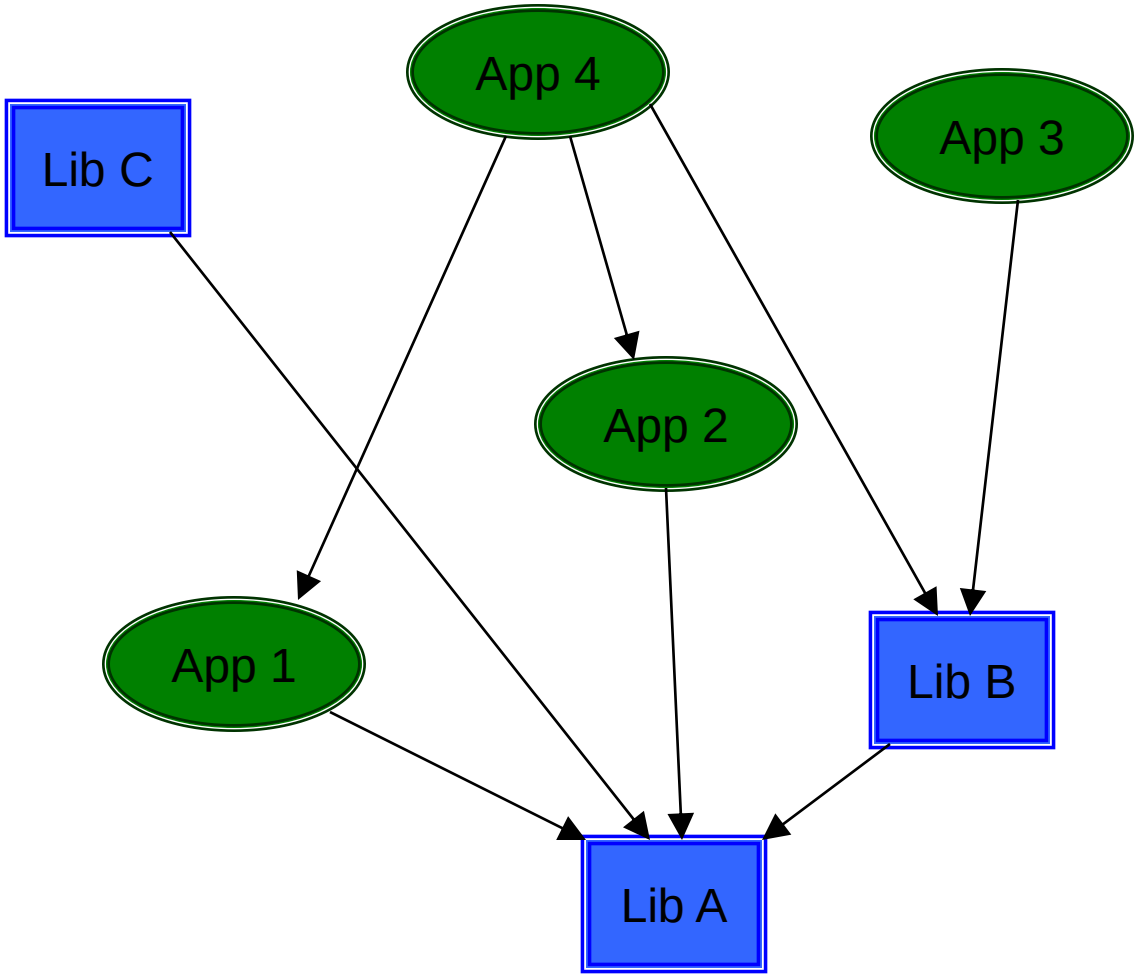
Each new edge is introduced by the addition of code that depends on code elsewhere.



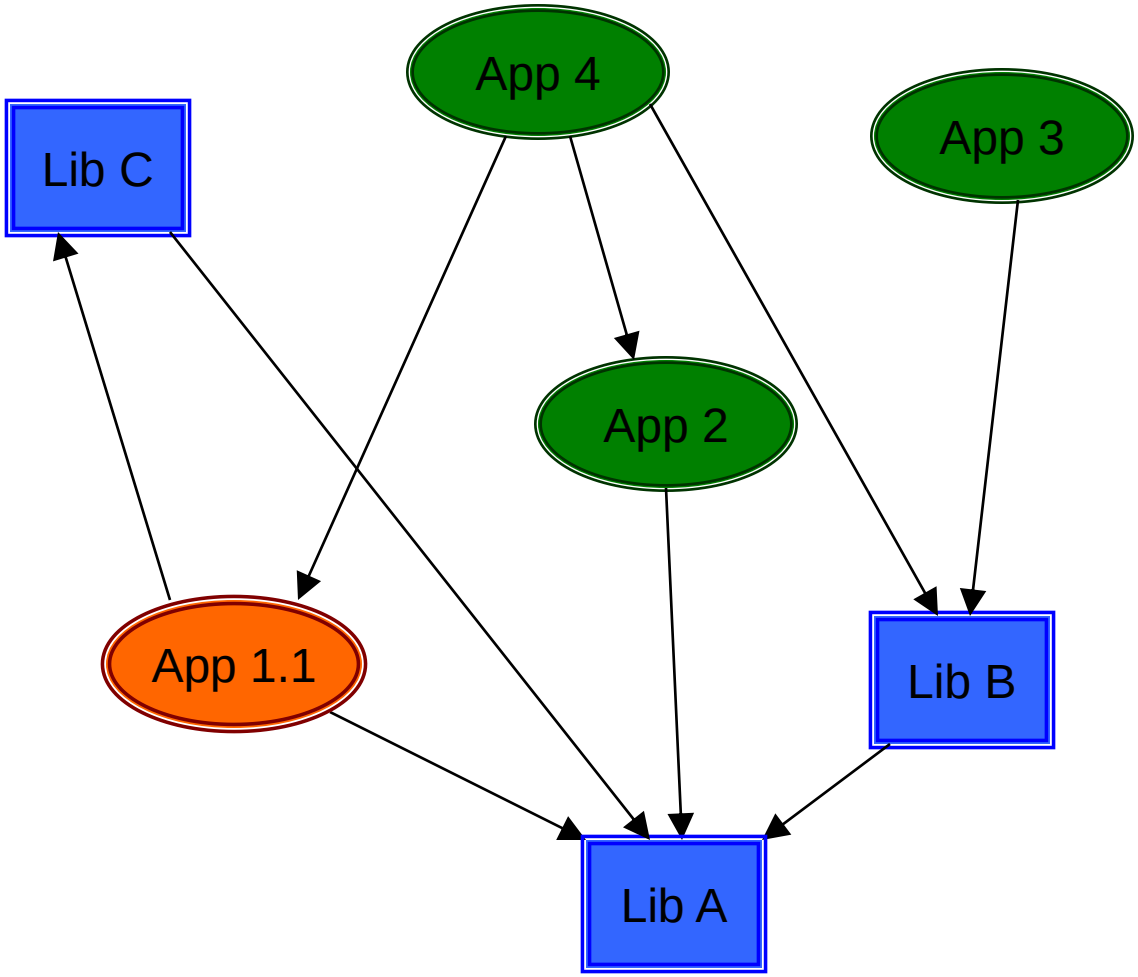
Creating a Big Ball of Mud



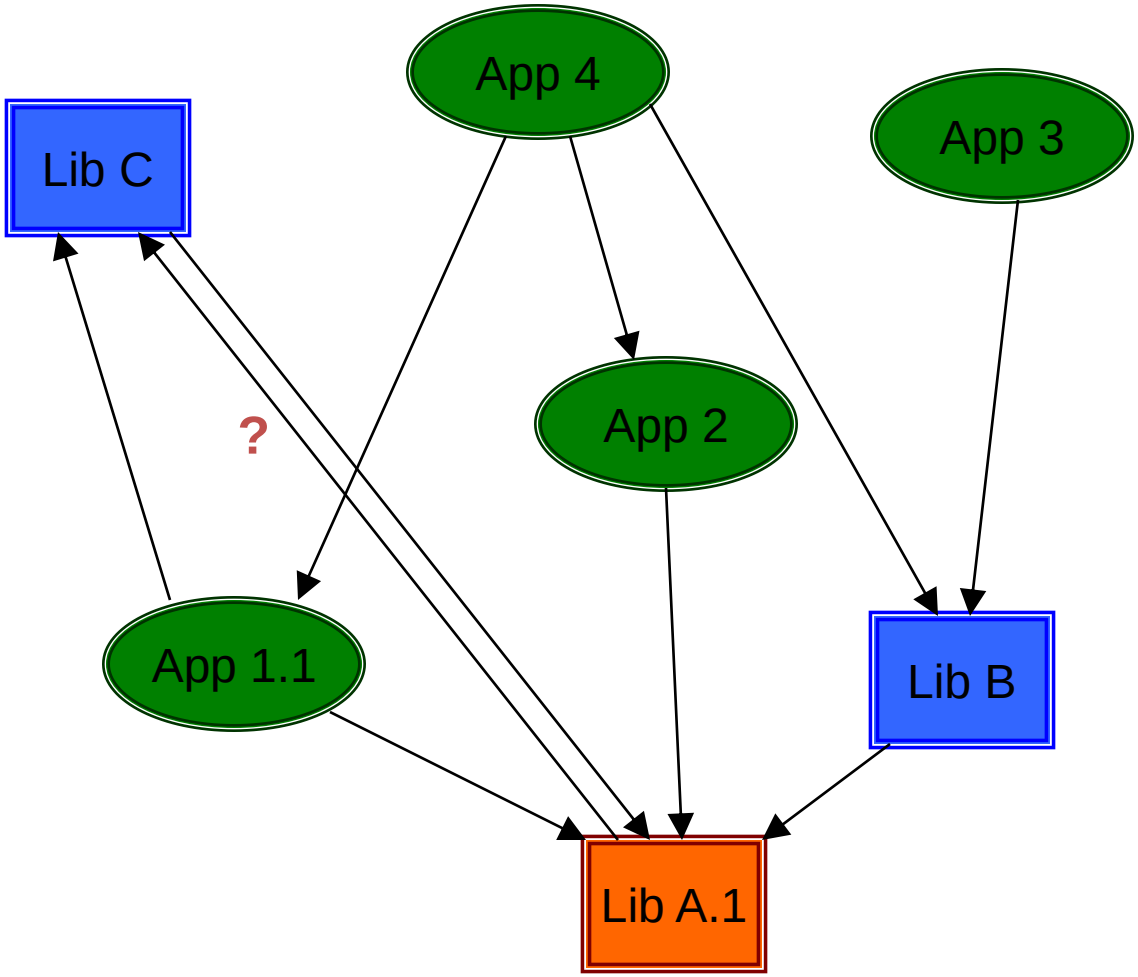
Creating a Big Ball of Mud



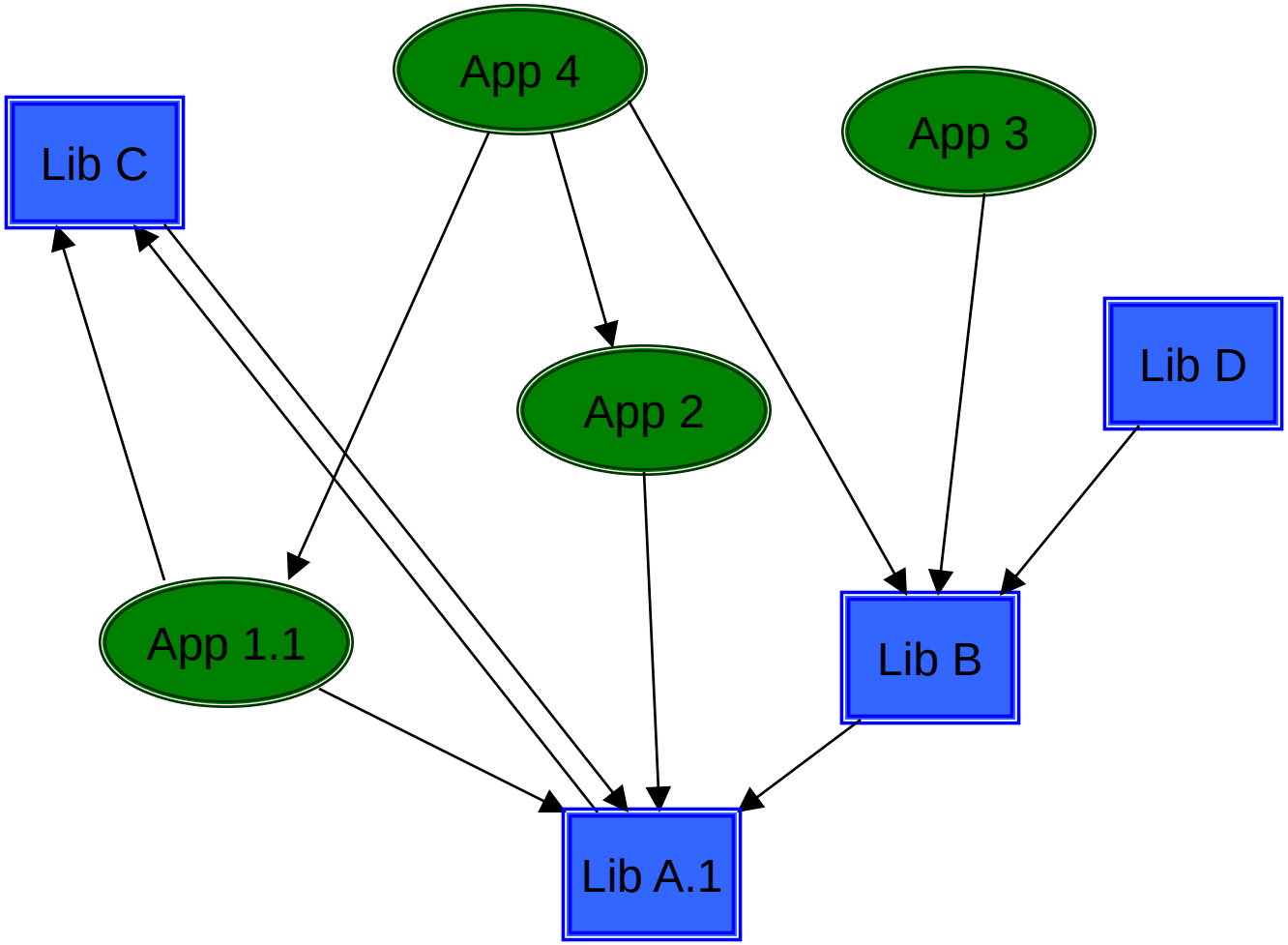
Creating a Big Ball of Mud



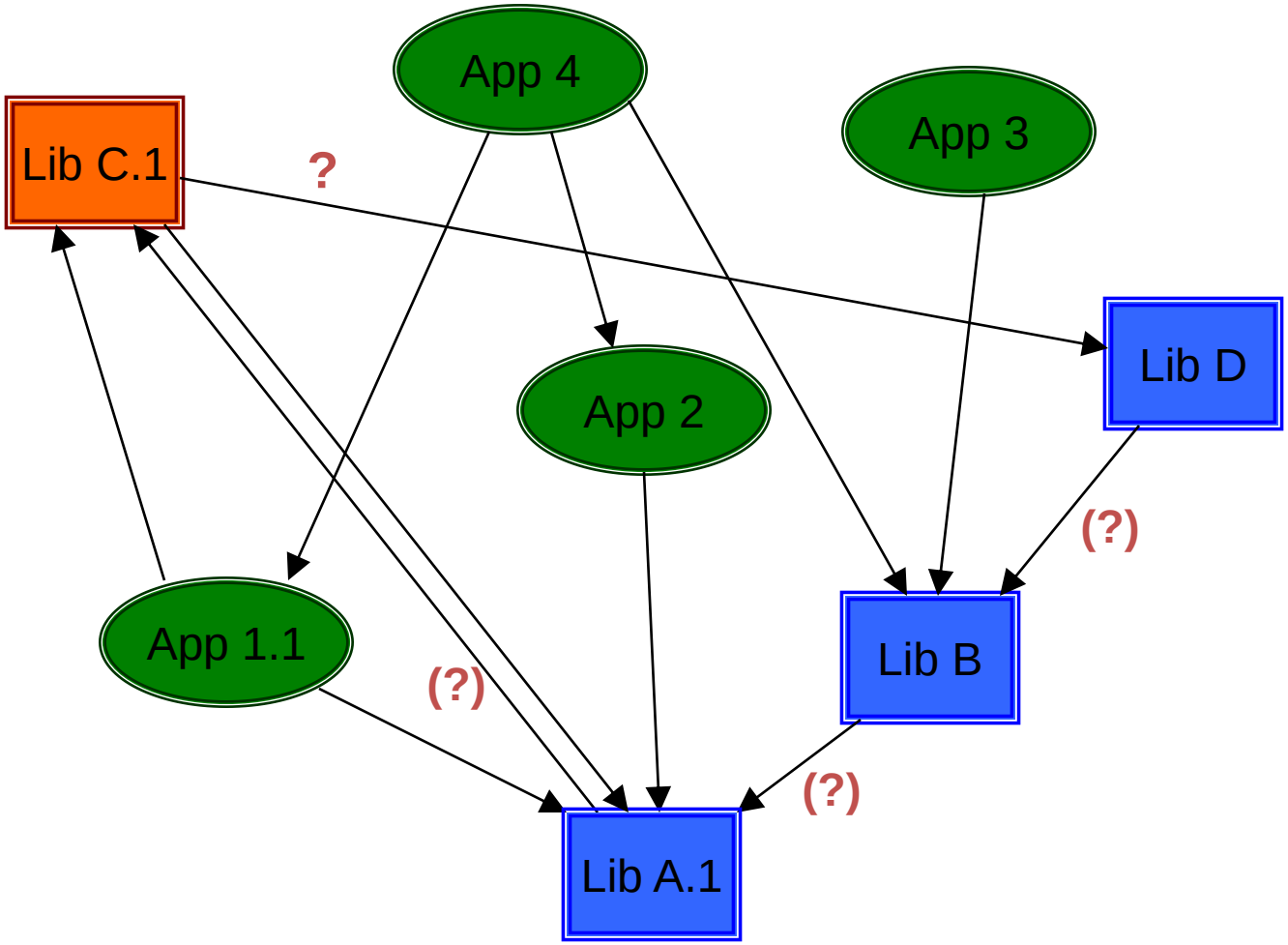
Creating a Big Ball of Mud



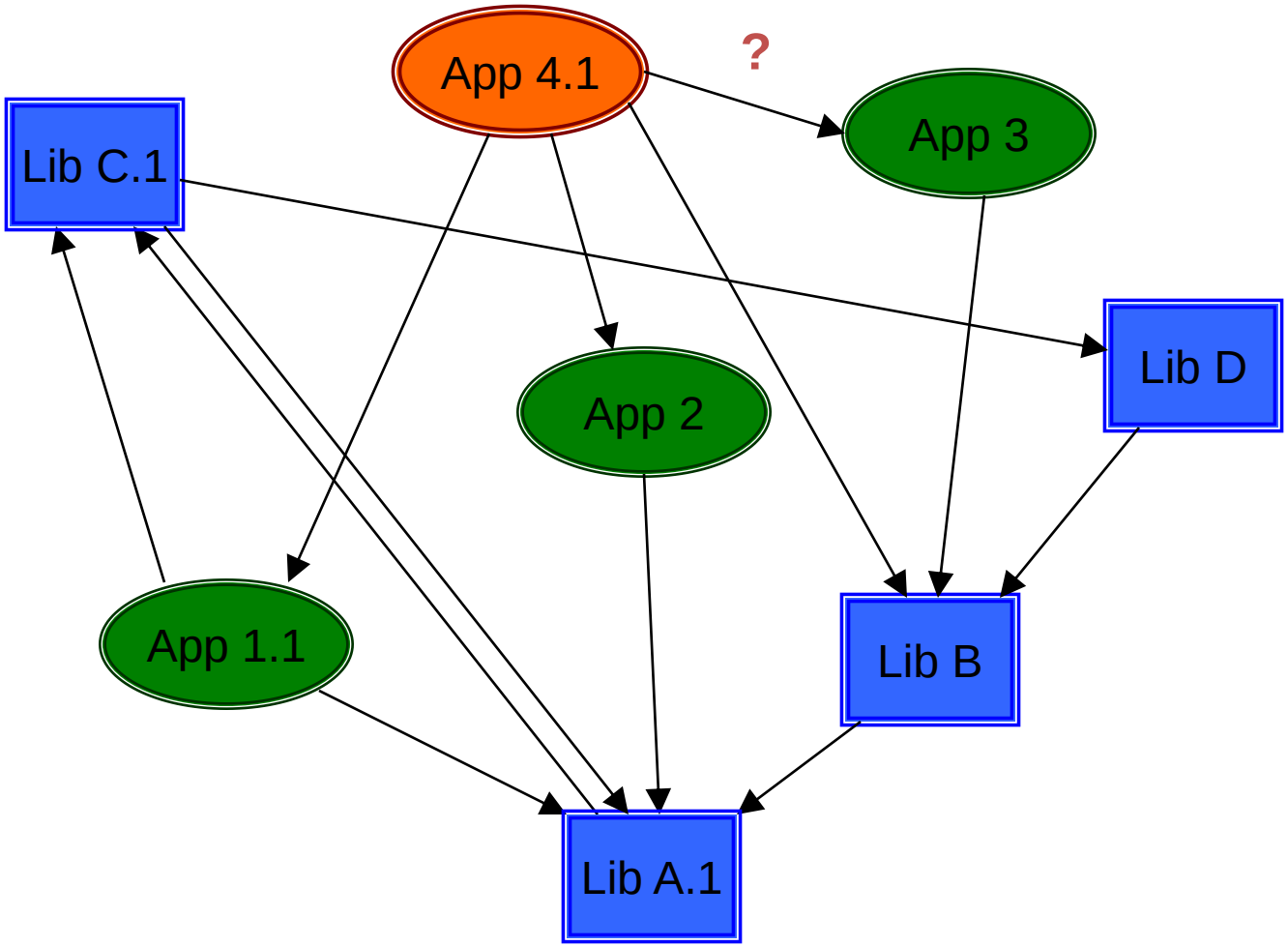
Creating a Big Ball of Mud



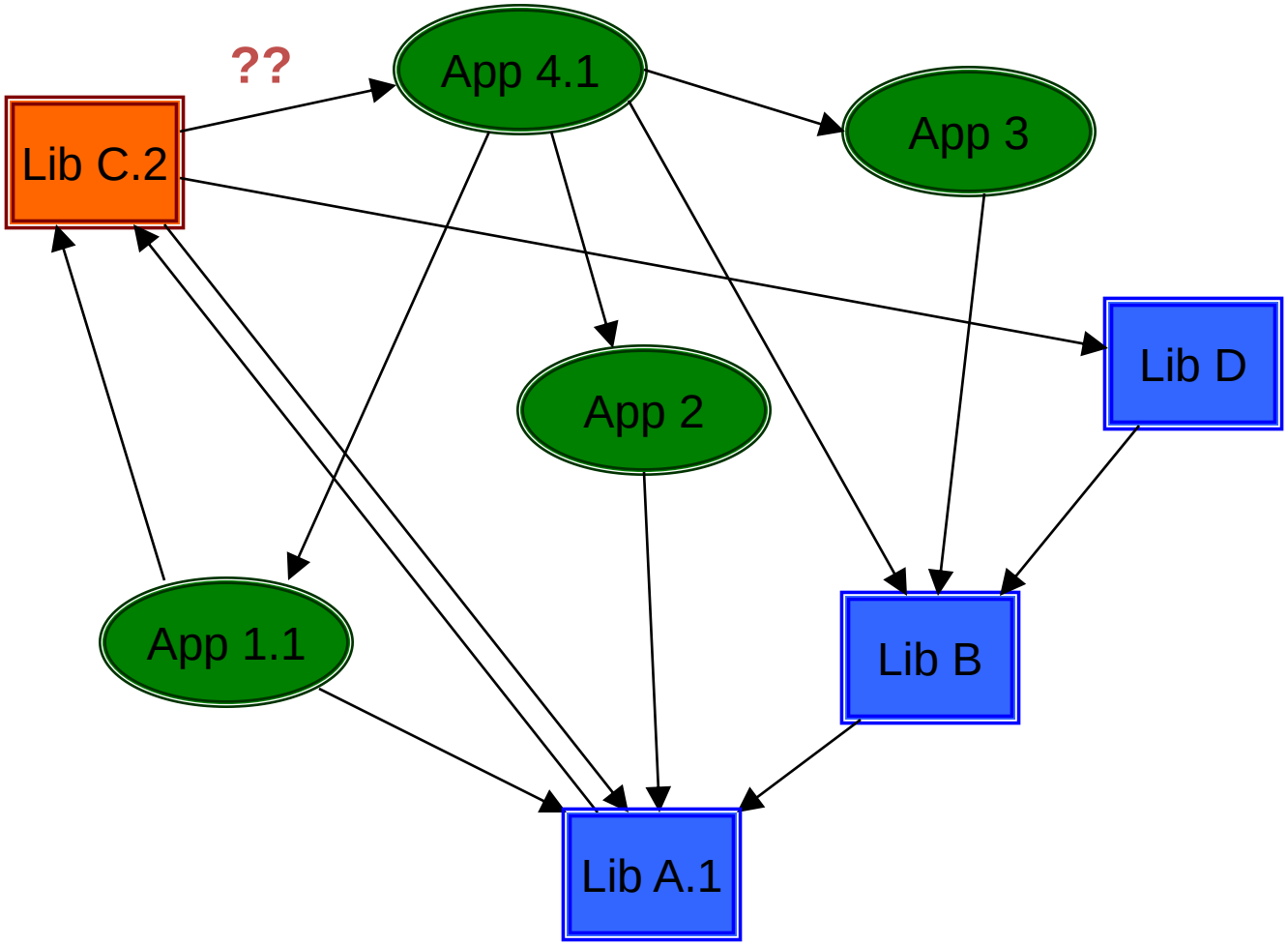
Creating a Big Ball of Mud



Creating a Big Ball of Mud

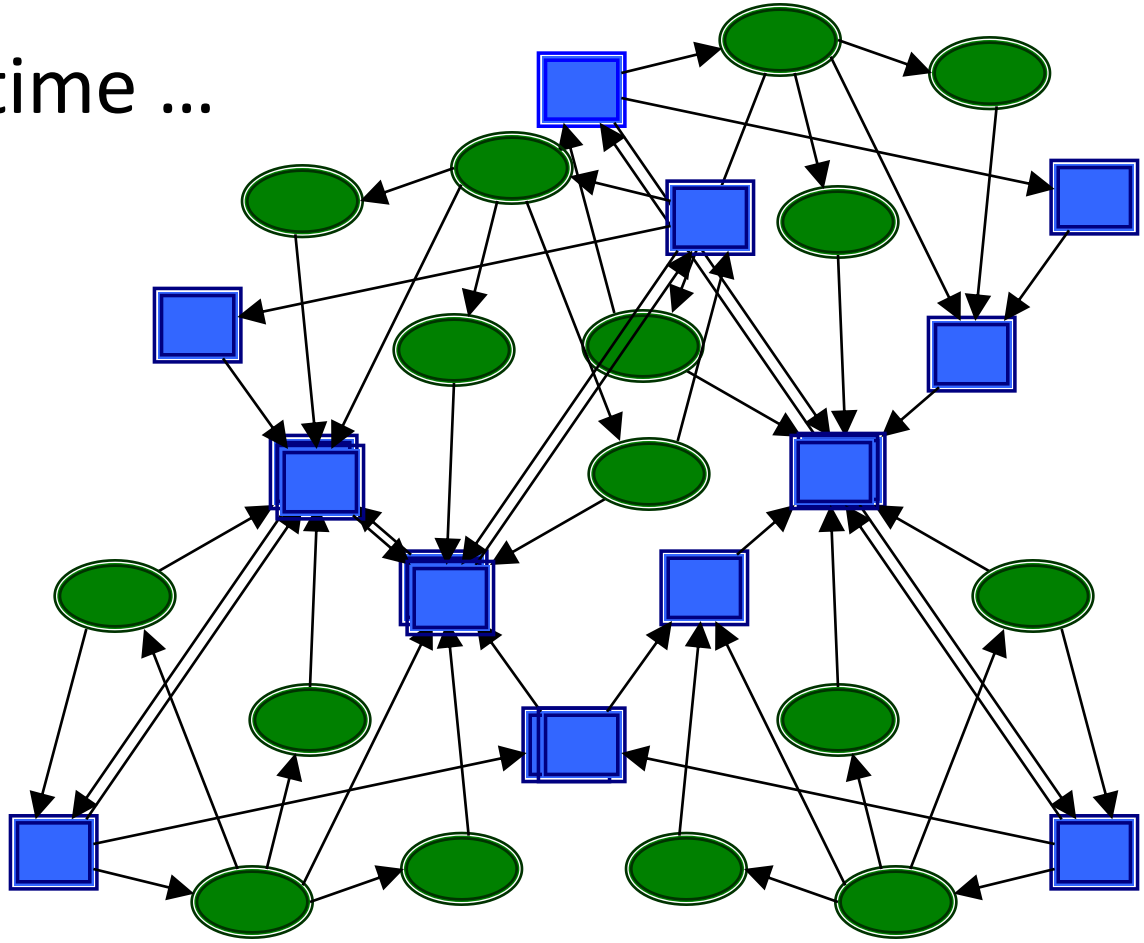


Creating a Big Ball of Mud



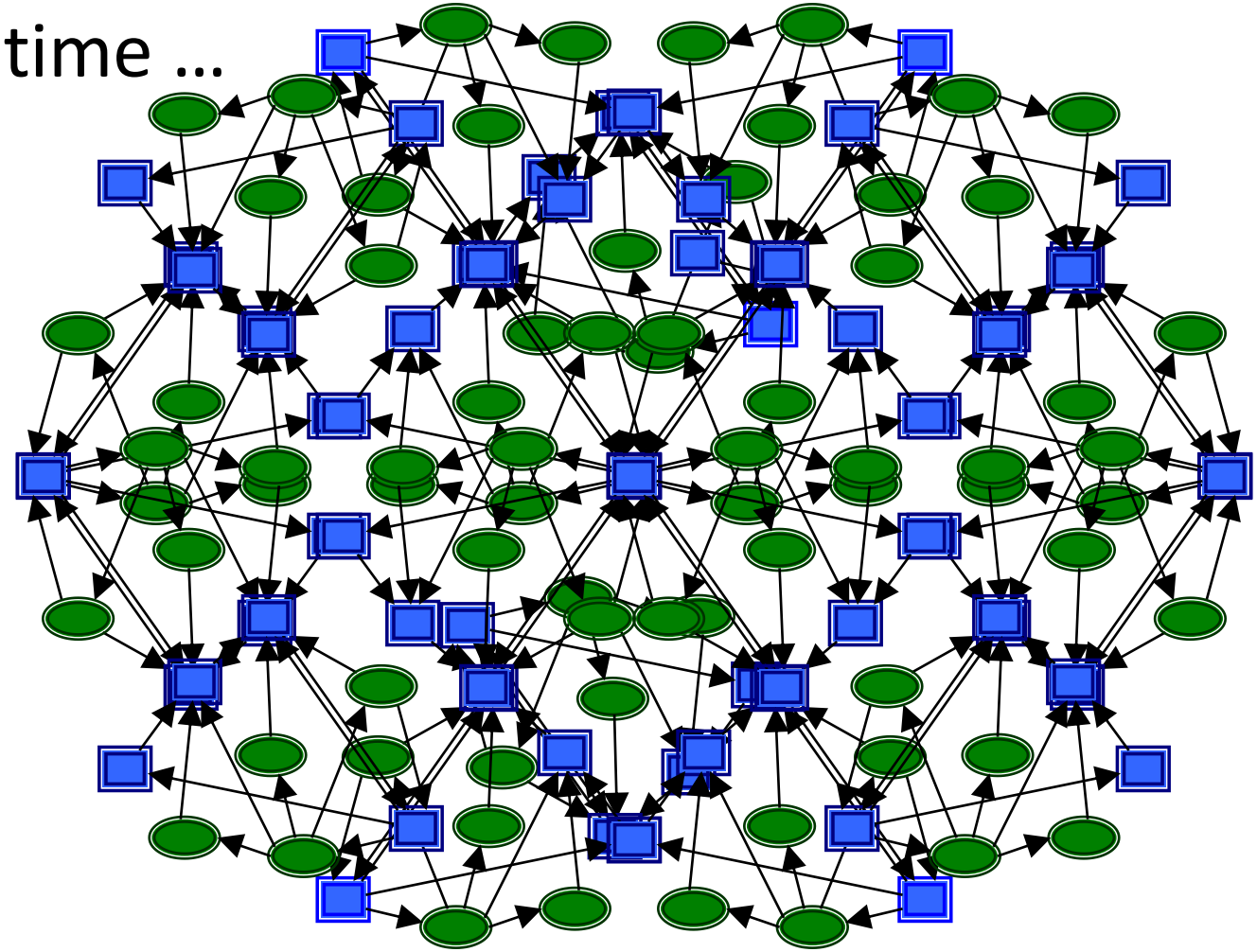
Creating a Big Ball of Mud

Over time …



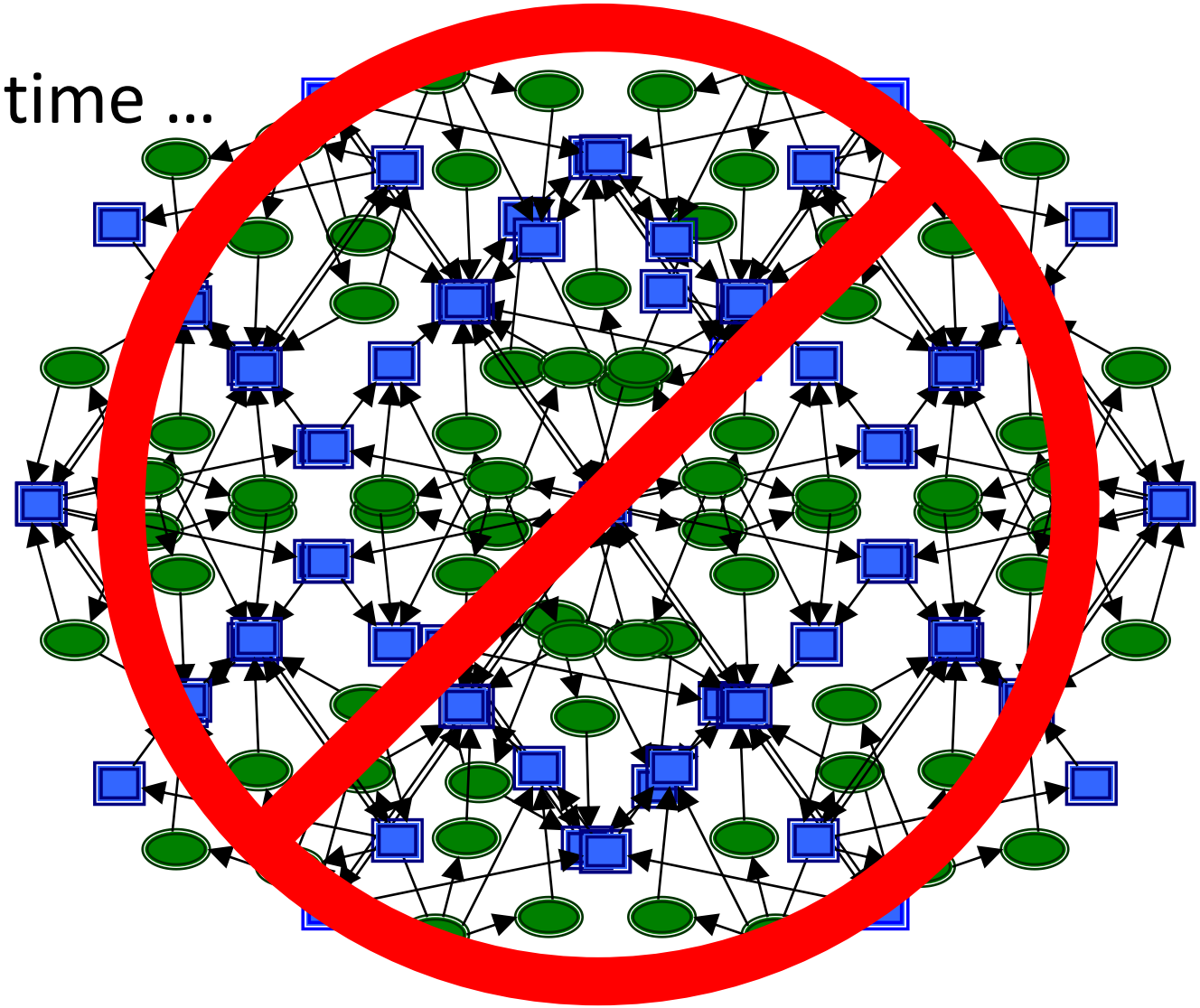
Creating a Big Ball of Mud

Over time …



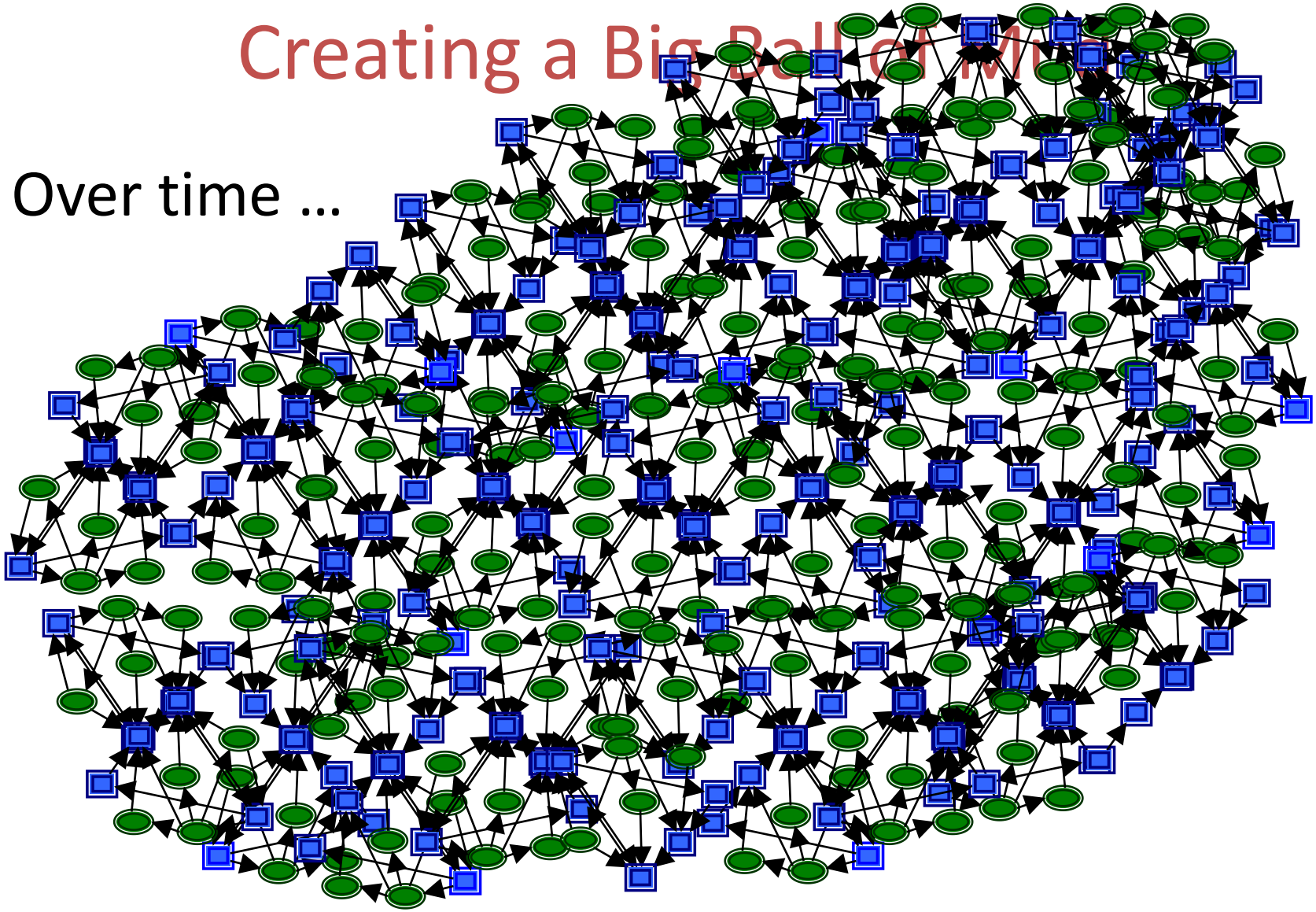
Creating a Big Ball of Mud

Over time …



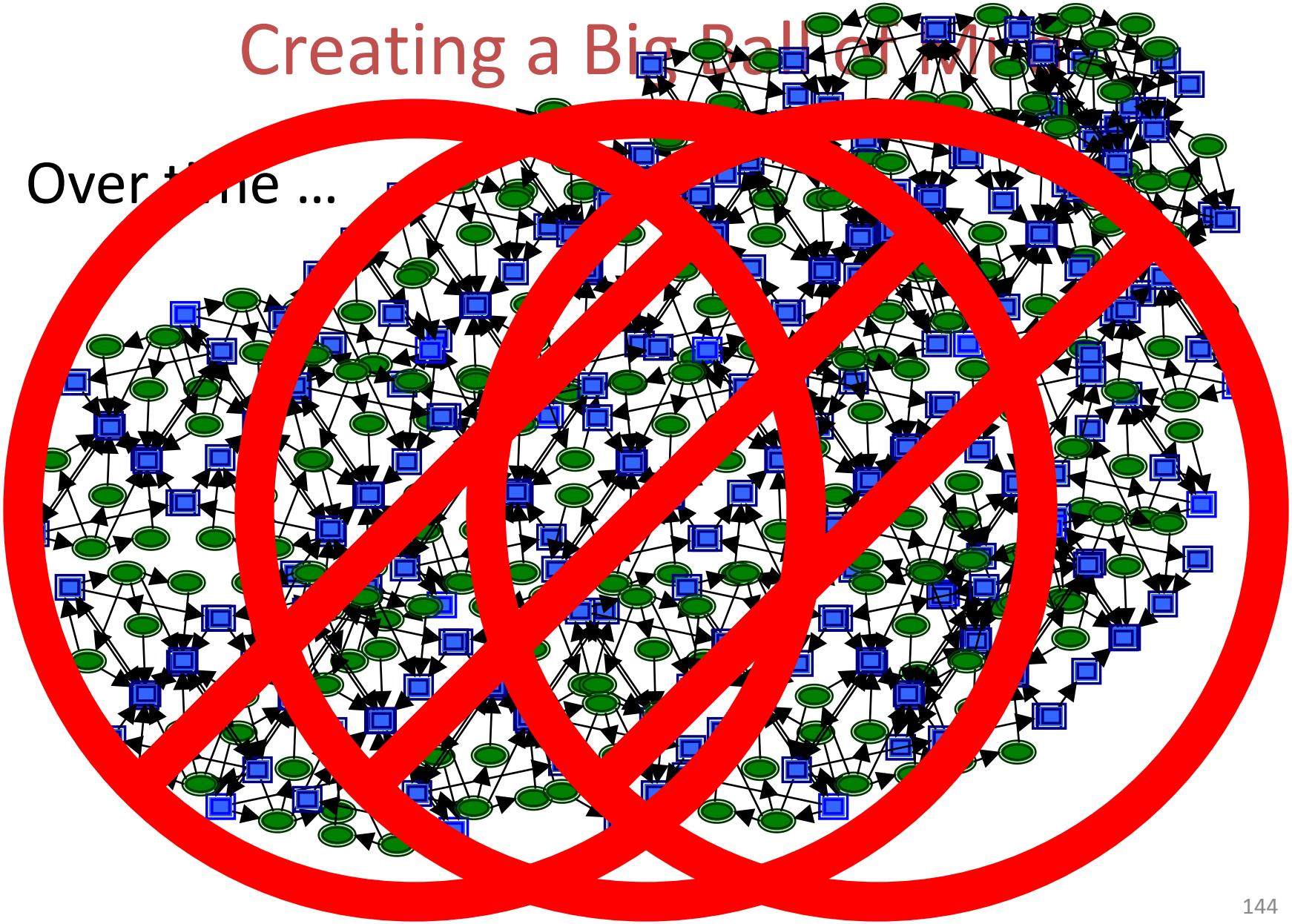
Creating a Big Ball of Mud

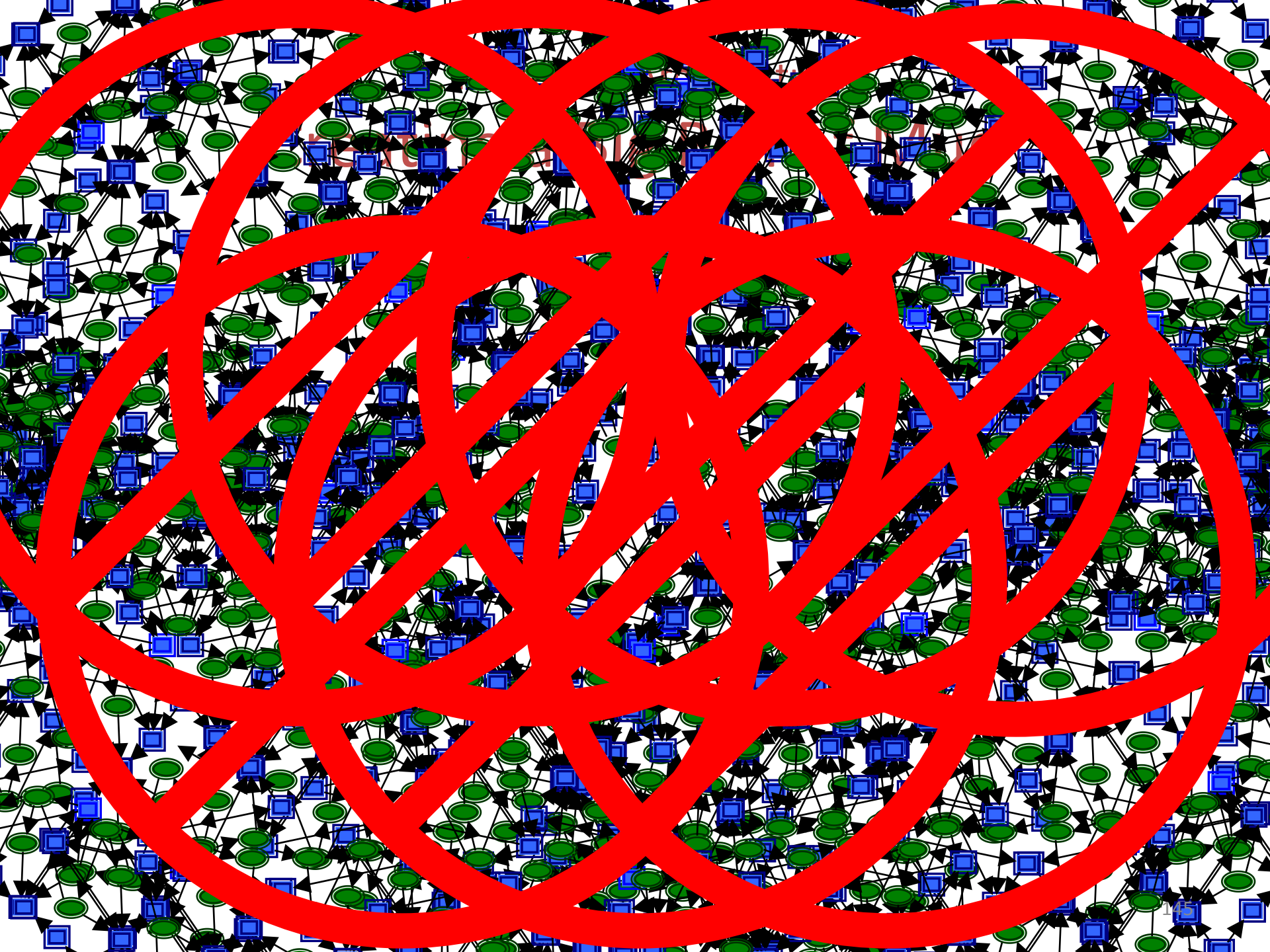
Over time …



Creating a Big Ball of Mud

Over time …




145