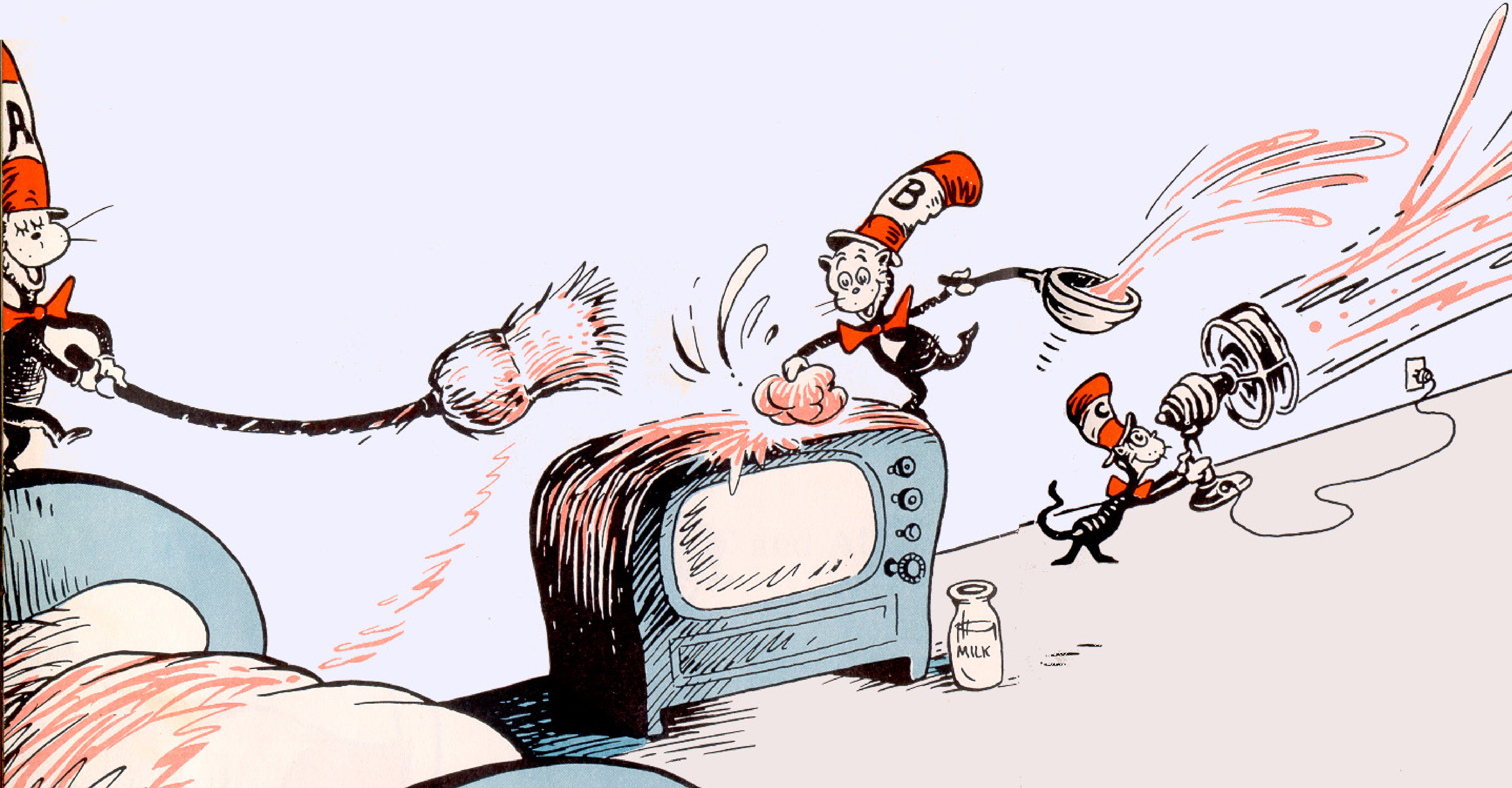
Migrating to a C++ Object-Oriented Design

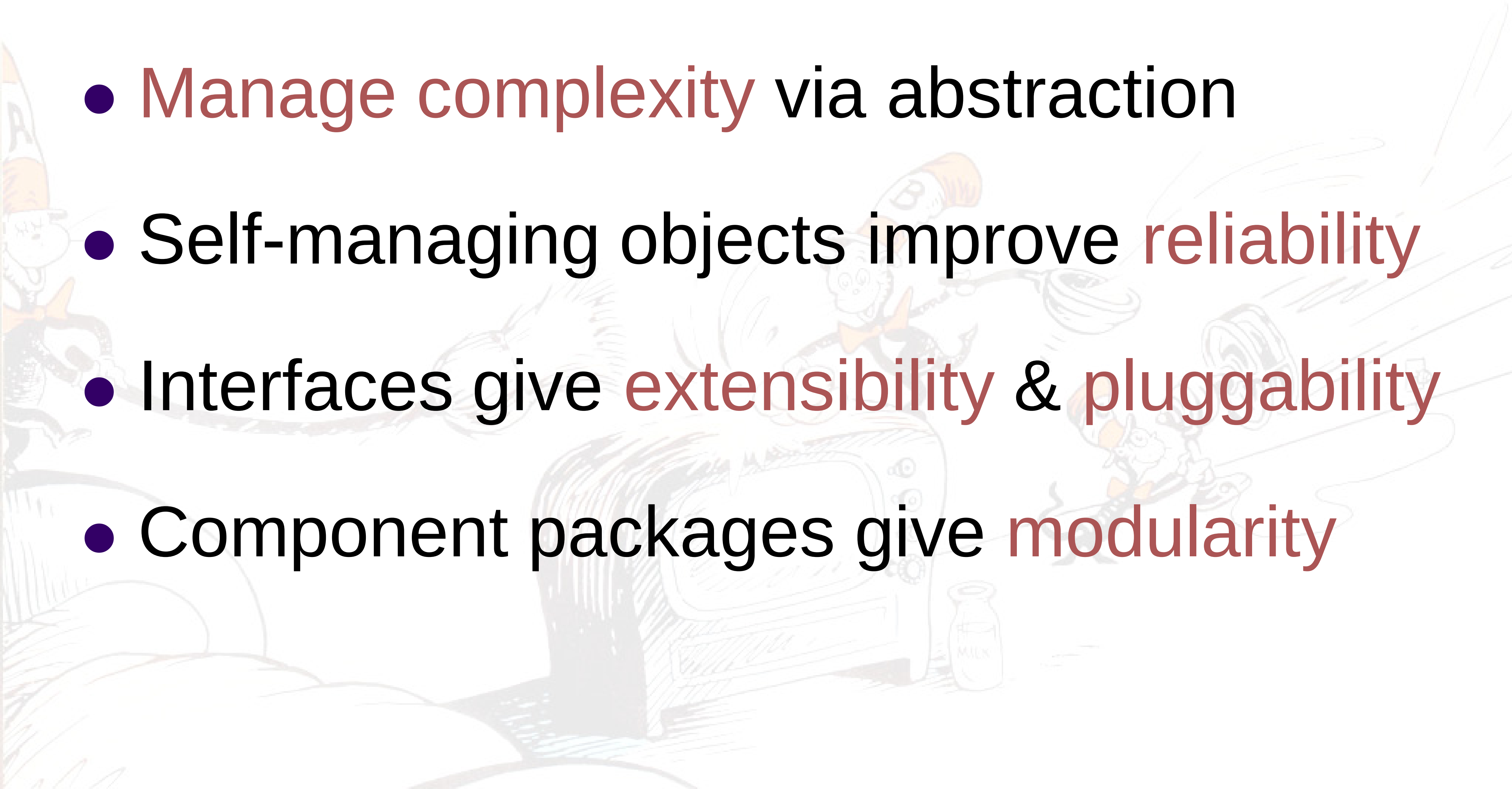
Strategies and Patterns

OBJEXX
ENGINEERING

Why OO?



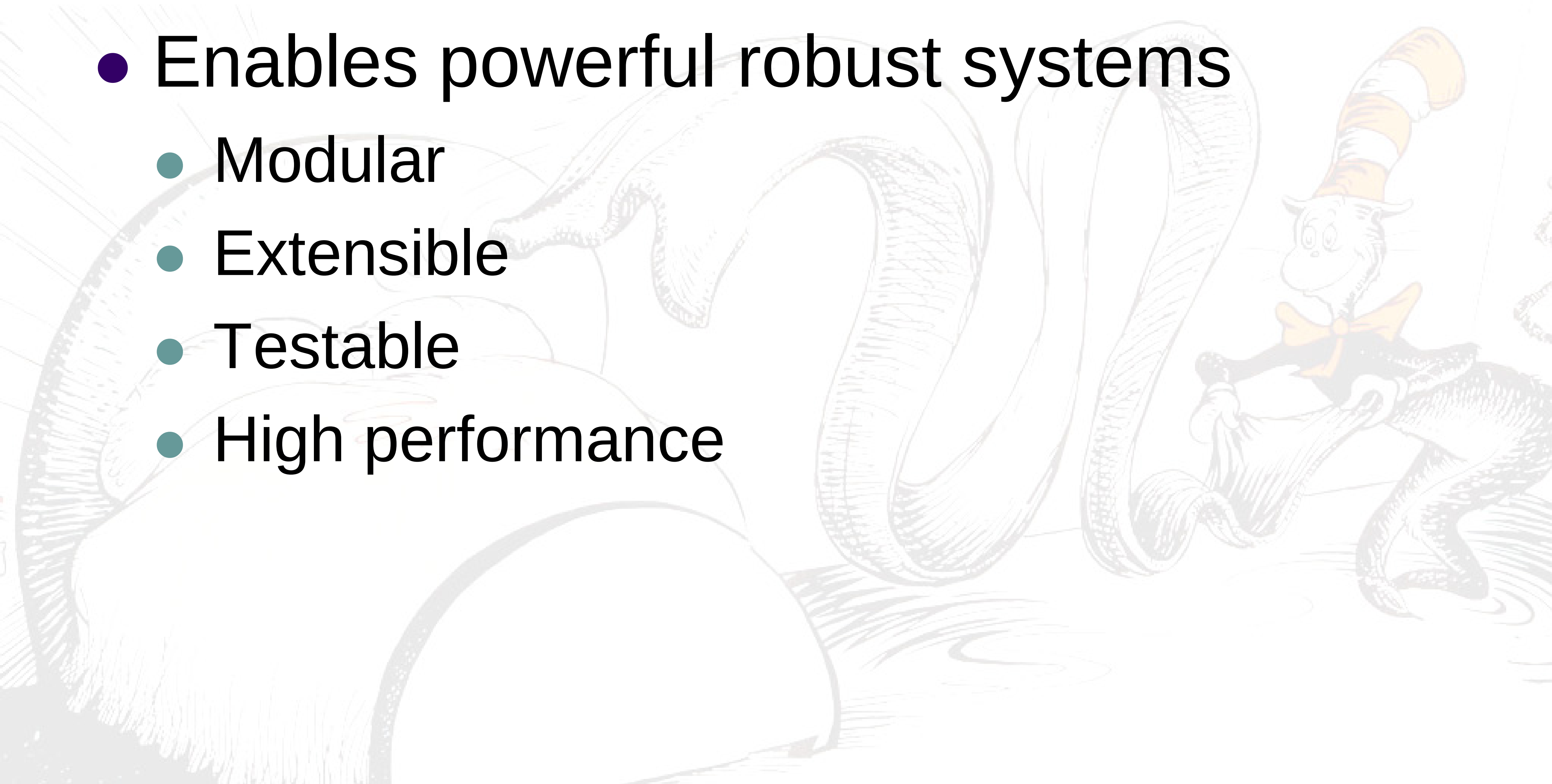
Why OO?



- Manage complexity via abstraction
- Self-managing objects improve reliability
- Interfaces give extensibility & pluggability
- Component packages give modularity

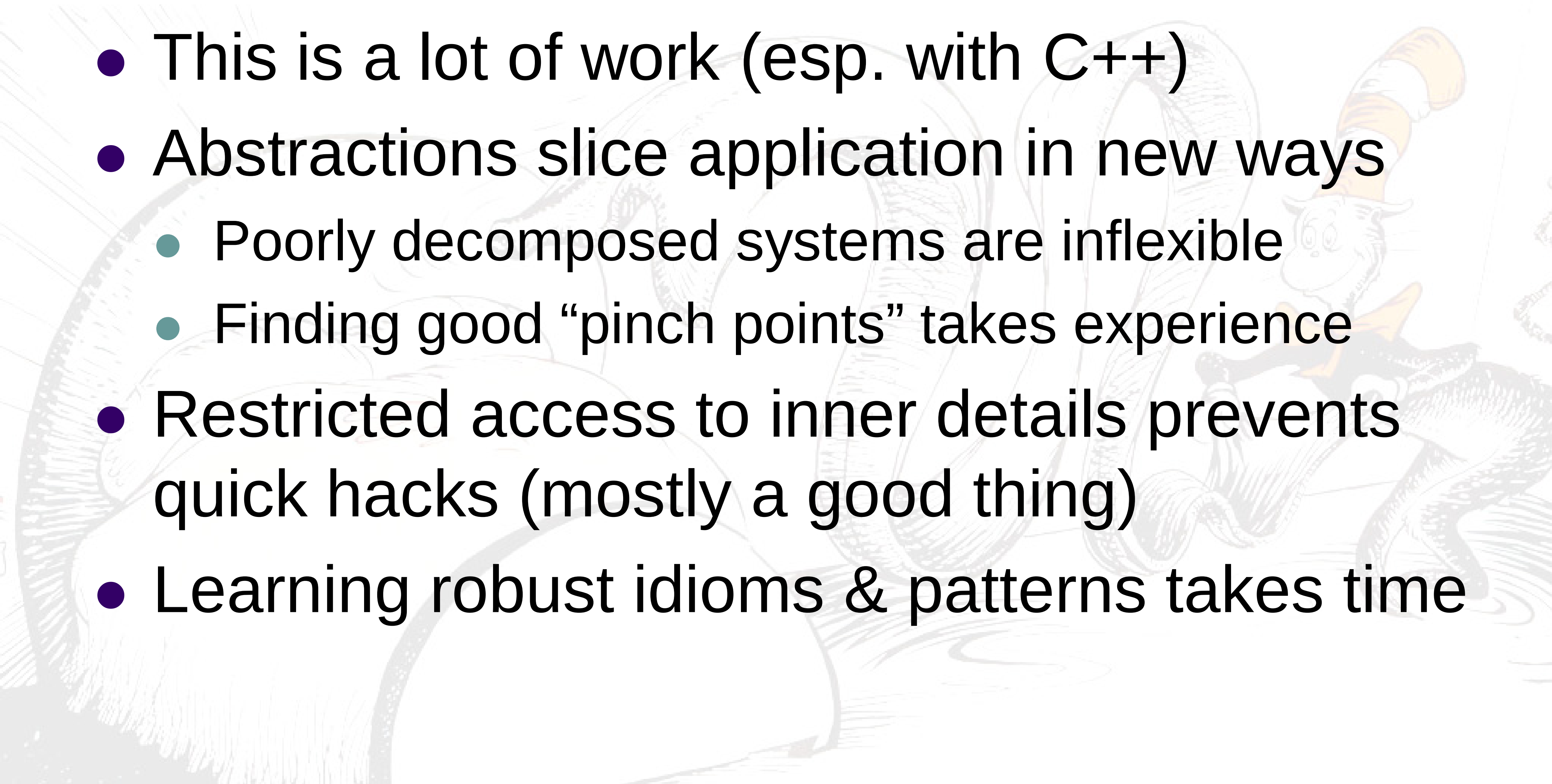
OO C++ Good News / Bad News

OO C++: Good News



- Enables powerful robust systems
    - Modular
    - Extensible
    - Testable
    - High performance

OO C++: Bad News



- This is a lot of work (esp. with C++)
- Abstractions slice application in new ways
  - Poorly decomposed systems are inflexible
  - Finding good “pinch points” takes experience
- Restricted access to inner details prevents quick hacks (mostly a good thing)
- Learning robust idioms & patterns takes time

Signs that You May Need OO

- System is hard to change (correctly)
  - Adding arguments down long call trees
  - Fixing bugs often introduces new ones
  - Code is “exceptional”: lots of arcane cases
- Repeated if/switch blocks
- Behavioral distinctions dominate computations
- Alternate algorithms are desired

OO C++ Rules of Thumb

- Use type safety to make bugs compile-time
- Use forward declarations maximally
- Use “live” assertion testing liberally (DBC)
- Use unit testing: It can be pretty painless
- Make source self-documenting & clear
- Source docs to clarify the subtleties

OO Principles

- Focus on reducing dependencies
  - Good for extensibility and compile speed
- Insulate *client* code from object internals
  - Interface-based designs: Clients know what not how
  - Factories localize dependence on concrete types
  - Pluggable and fast compilation
- Make package dependencies acyclic

More OO Principles

- Classes should have clearly defined, limited scopes of responsibility
  - Testable and less buggy
  - Allows self-management of invariants and pre- and post-conditions

Can you Migrate to OO? (Yes)

- You can introduce OO to a procedural app
- Apps can evolve towards an OO design
- You don’t need to make everything OO
- Various strategies to choose from
  - Most useful first: biggest bang theory
  - Top-down: High-level parts are easier to replace
  - Bottom-up: Low-level mess permeates the code

Migration Impacts

- Choose a migration strategy
- Sequence development to gradually replace legacy components with OO components
- Aim for frequent functional milestones
- Building tests in parallel is vital

Application Domain Modeling

- Finding the objects
  - Start with natural application domain nouns/entities
- Algorithm variants
- Rough out a high-level design
  - CRC cards
  - Use cases
  - Class diagrams
  - Sequence diagrams
- Agile process: Build this!

CRC Card

Class *Name*

Responsibilities

- Interface (Services)
- Invariants (Promises)

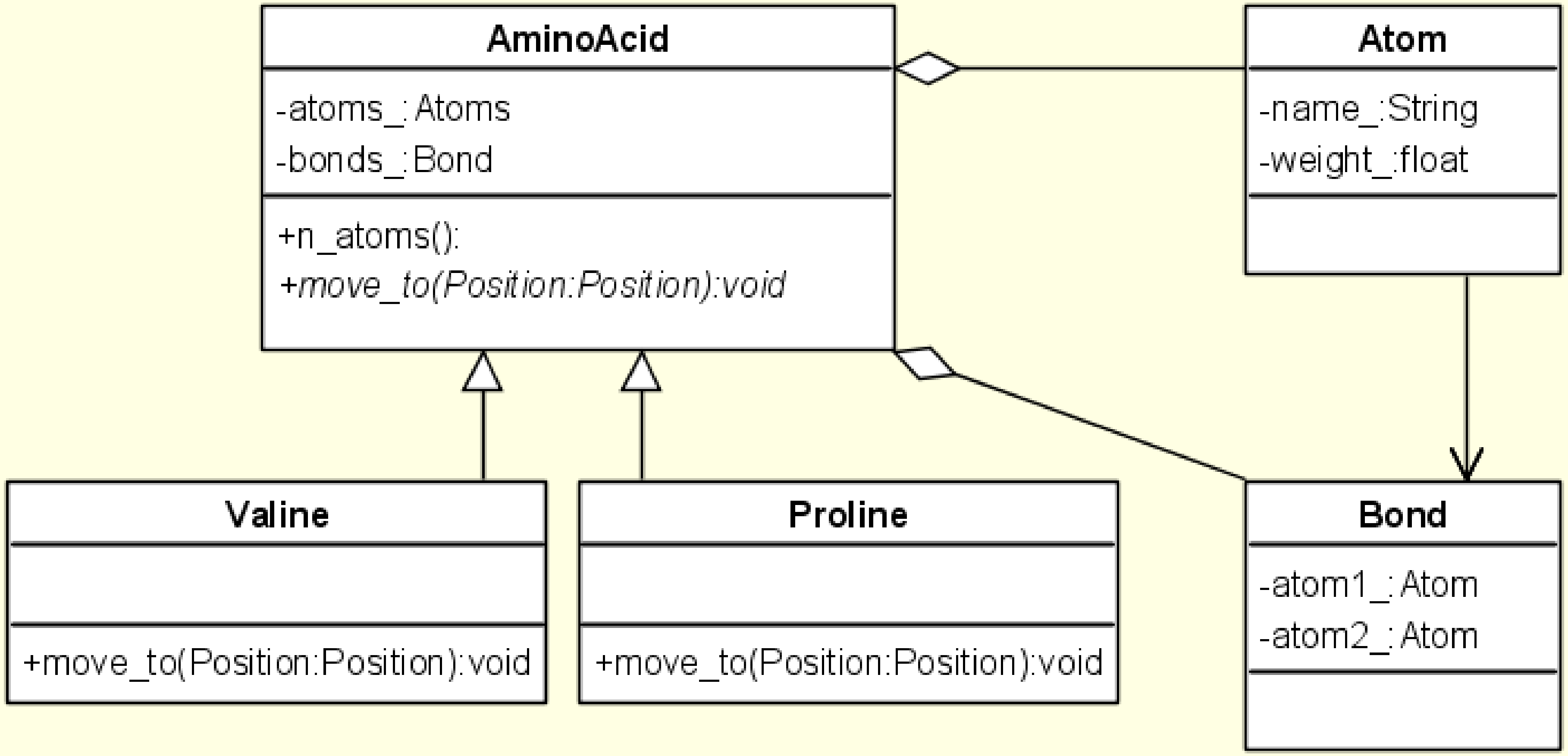
Collaborators

- Who does it interact with
- Does it use or own them

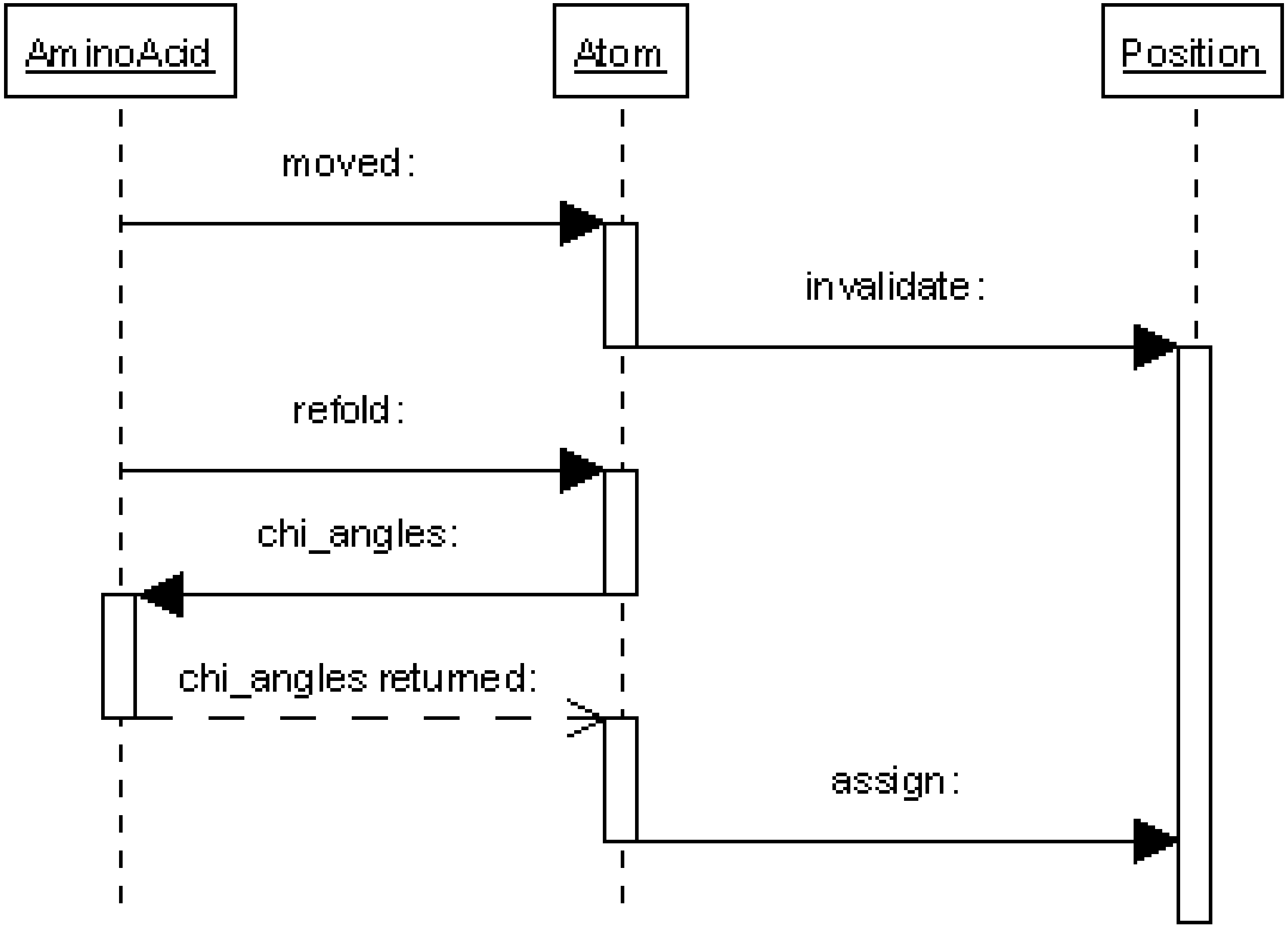
Class Profile

Name: **AminoAcid**	
Description: **An AminoAcid**	
What it Does	Who it Works With
move	**Position**
score	**Atom, Bond**
What it is Made Of	Description
Atom	
Bond	
ChiAngle	
Invariants	Description

UML Class Diagrams



UML Sequence Diagrams



Agile Processes

- No heavyweight up-front design process
- Build the application domain model
- Evolve it gradually
- Small iterations between working systems
- Test in parallel: Find problems early
- Evolve requirements in parallel
- Decoupled, interface-based designs help

Design Modeling & Implementation

- Fill out design as real implementations added
- Add “glue” and helper classes/functions
- Discover finer granularity design
- Refine package design for low coupling
- Agile: Build this as you go!

Stage 1: Data Bundling

- Combine data for objects into structs
- This may require cutting up arrays
- Reduces function argument lists

```
double x[N], y[N], z[N];
```

might become:

```
struct Position
{
  double x, y, z;
};

vector< Position > p(N);
```

Stage 2: Migrate Behavior

```
struct Position
{
 // Default Constructor
 Position() :
  x( 0.0 ),
  y( 0.0 ),
  z( 0.0 )
 {}

 // Coord Constructor
 Position(
  double x_,
  double y_,
  double z_
 ) :
  x( x_ ),
  y( y_ ),
  z( z_ )
 {}

 // Length
 double
 length() const
 {
  return sqrt(x*x+y*y+z*z);
 }

 // Normalize to unit length
 void
 normalize()
 {
  double const l( length() );
  assert( l > 0.0 );
  double const li( 1.0 / l );
  x *= li;
  y *= li;
  z *= li;
 }

 // Data
 double x, y, z;
};
```

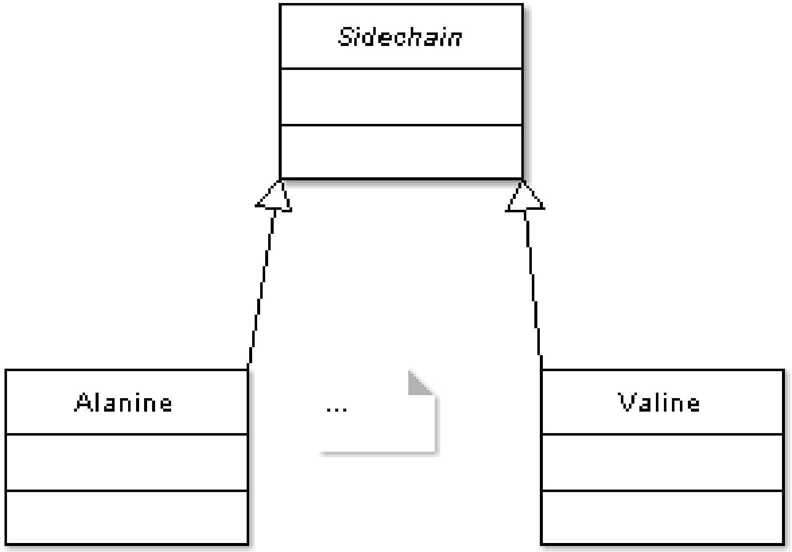
Stage 3: Hidden Data

```
class Position
{
public:

  // Default Constructor
  Position() :
    x_( 0.0 ),
    y_( 0.0 ),
    z_( 0.0 )
  {}

  // Coord Constructor
  Position(
    double x,
    double y,
    double z
  ) :
    x_( x ),
    y_( y ),
    z_( z )
  {}
```

```
  // X coordinate
  double
  x() const
  {
    return x_;
  }

  // X coordinate
  double & // Lose control
  x()
  {
    return x_;
  }

  // X Assigment
  void // Keep control
  x( double x_a )
  {
    // Control invariants
    x_ = x_a;
  }

private: // Data
  double x_, y_, z_;
};
```

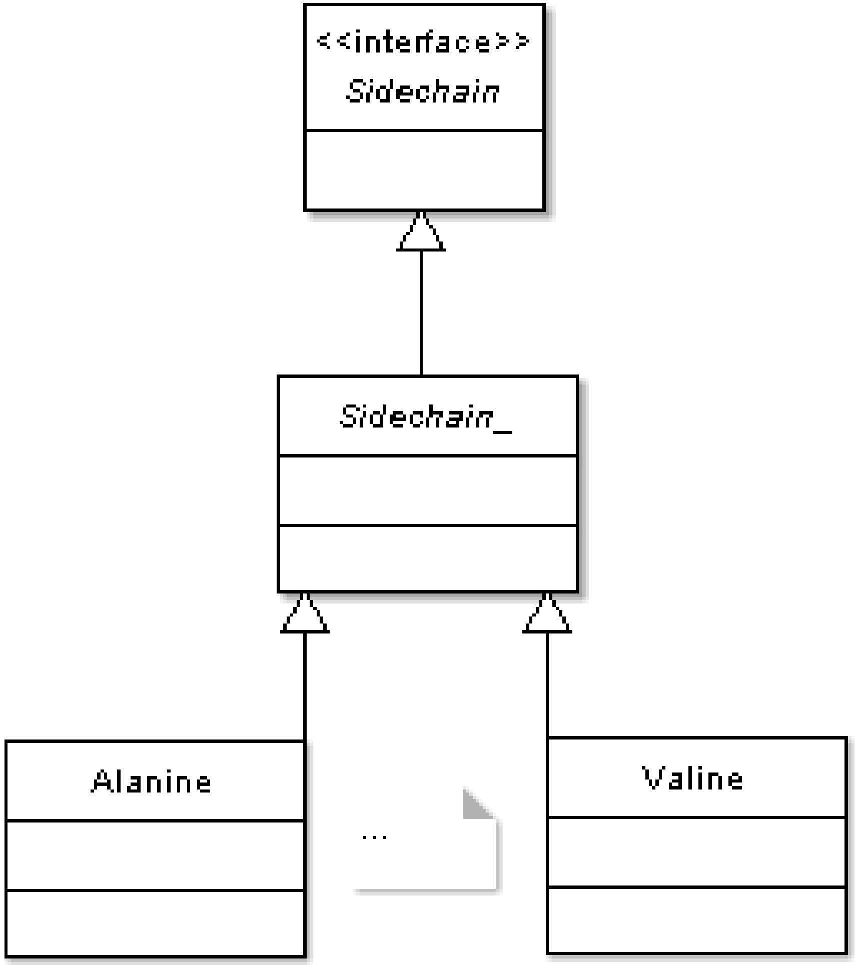
Stage 4: Polymorphism

- Insulates *Sidechain* users from concrete details
- Pluggable
- More maintainable
- Faster compiles
- But abstract Sidechain has shared implementation



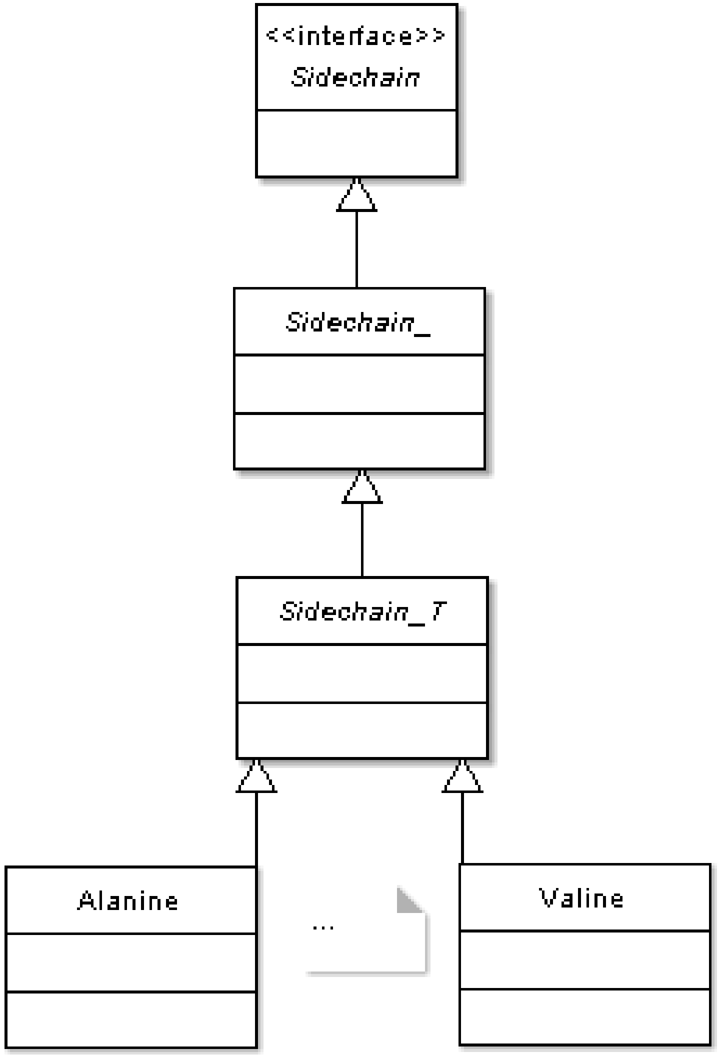
Stage 5: Interfaces

- Observation: Users don’t need to see shared data or implementations
- Solution: Interface root with only pure virtual functions and no data



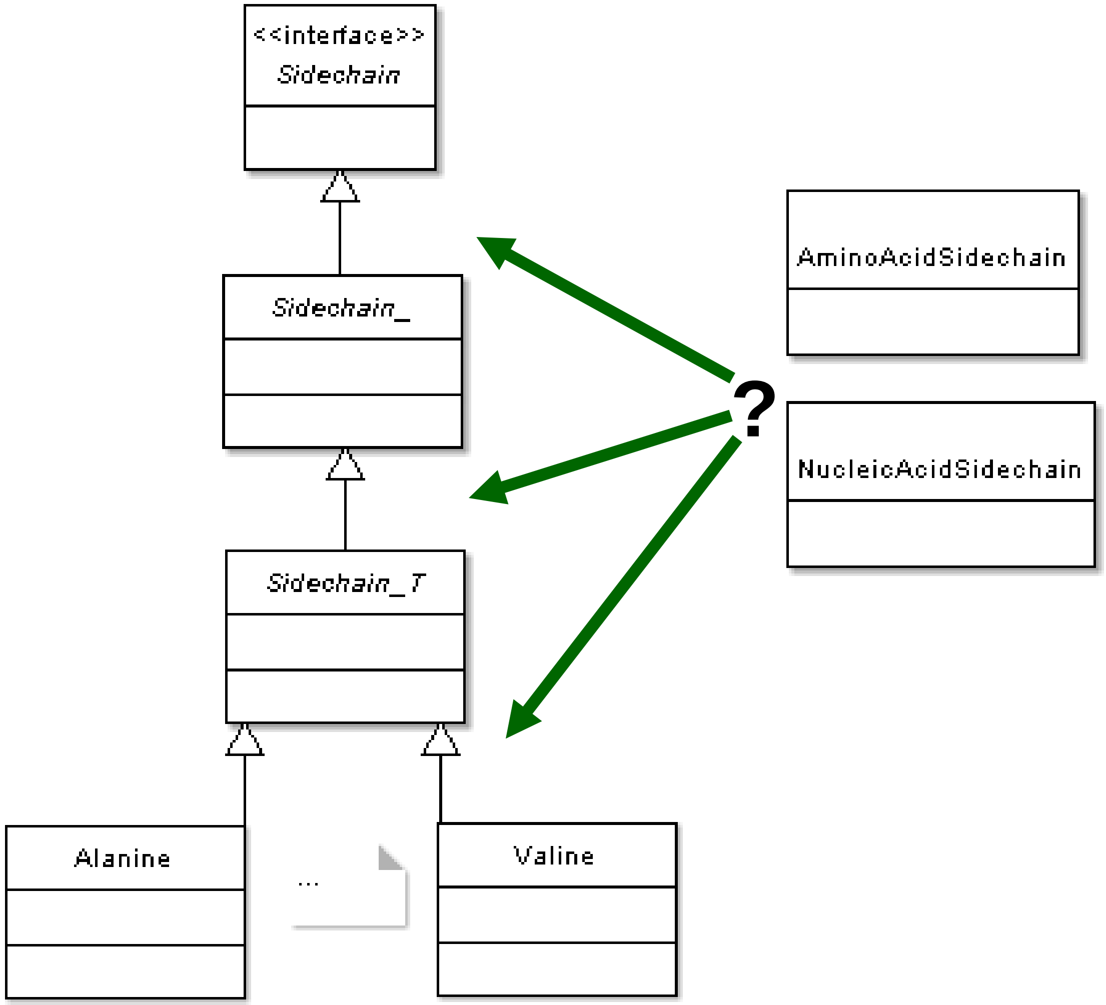
Stage 6: Templates

- A lot of shared data and functions that depend on the concrete type or another template argument
- Add a template layer
  - Code sharing: GOOD
  - Dependencies: BAD
- CRTP: Template base is unique to each concrete



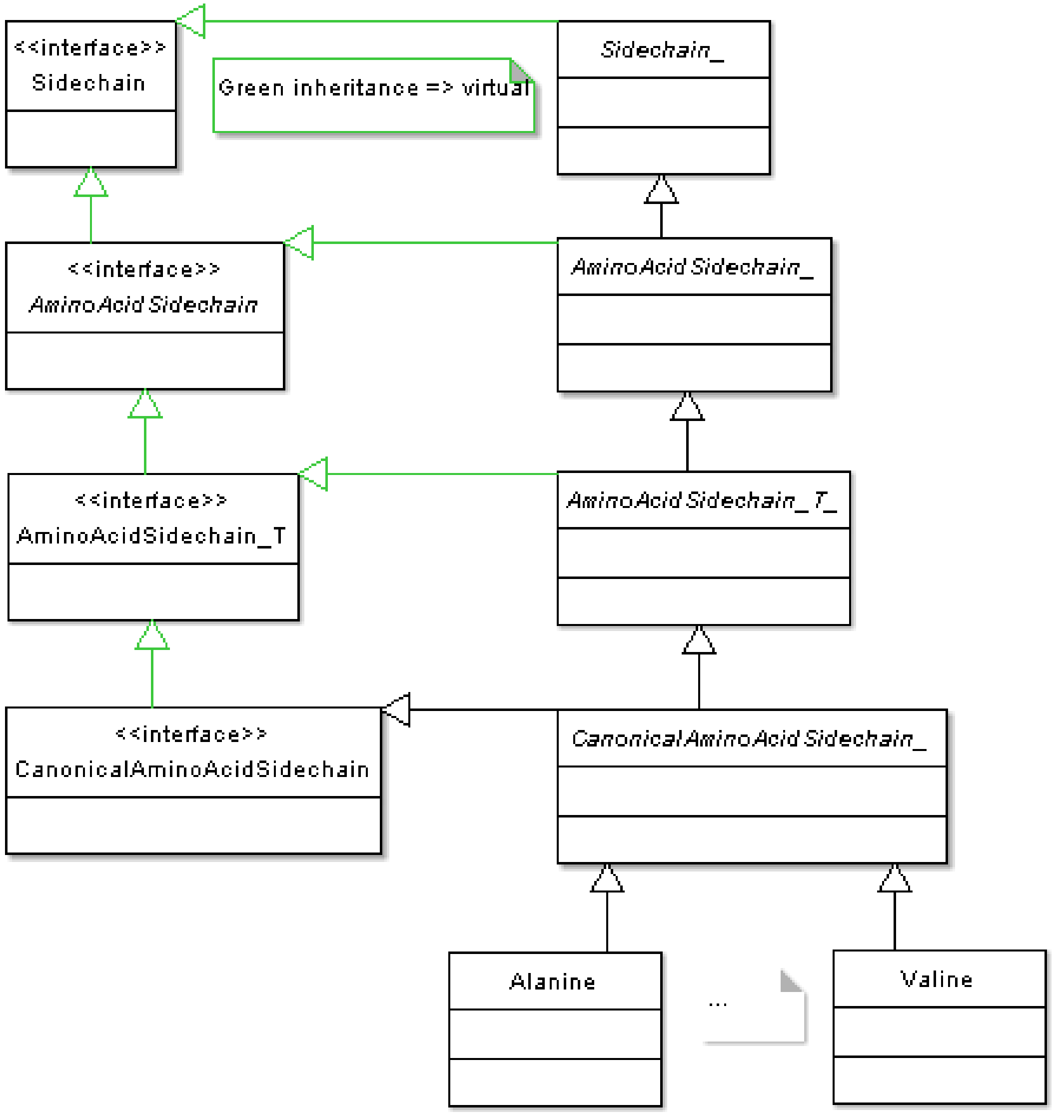
Are We Done?

- What if clients need subinterfaces? Want a chain of interfaces
  - How to avoid dependencies?
- Solution: Lattice hierarchy



Stage 7: Lattice Hierarchy

- Interface chain
- Multiple inheritance
- Not simple



<<interface>>
Sidechain

Green inheritance => virtual

*Sidechain_*

<<interface>>
AminoAcidSidechain

*AminoAcidSidechain_*

<<interface>>
AminoAcidSidechain_T

*AminoAcidSidechain_T_*

<<interface>>
CanonicalAminoAcidSidechain

*CanonicalAminoAcidSidechain_*

Alanine

...

Valine

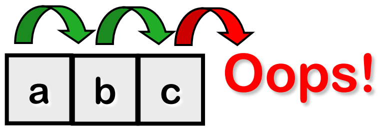
Proper C++ Classes

- RAII: Obtain resources in constructors and release them in destructor
  - Don’t hand out pointers hoping user will/won’t delete
- Rule of 3: If class owns heap resources write a copy constructor, assignment, and destructor
- Base class destructors must be virtual
  - Make them pure if no other pure functions
- Abstract class assignment is usually protected
  - Prevents slicing of data-incompatible subtypes
  - Virtual assignment idiom when appropriate

What C++ Adds Automatically

- Default constructor if no constructors specified
- Copy constructor unless suppressed or have reference data members
- Assignment unless have reference or const data members
- Automatic copy constructor and assignment do memberwise shallow copying

C++ Best Practices



- No C-style arrays & strings (leaks & overruns)
- No C-style i/o (unless performance dictates)
- RAII / Avoid manual heap use in client code
- Private class data members
- Pass by reference unless small built-in types
- Prefer exposing nothing or iterators to class containers
- Establish project guidelines for lifetime management, naming, style, …
- Assert invariants and pre-/post-conditions
- Code reviews (pair or group)

Useful Idioms

- Virtual construction: clone() and create()
- Named constructors: Avoid flag arguments
- Named keys (no “magic” numbers)
- Smart pointers: Manage ownership/lifetime so your code can focus on membership
  - Intrusive smart pointers are faster and safer
  - Beware of thread-safety issues
  - Keep pointer graph acyclic
  - auto_ptr is not usually what you want!

Useful Patterns

- Factories: Create correct type when specified at run-time by name, etc.
  - Localize dependency on concrete types
  - Pluggable Factory! Zero-maintenance & Zero-dependency: Very cool!
- Strategy: Hierarchies of algorithms
- Observer: Objects talk behind the scene
- GOF Book is good after initial stages

Code Duplication: Problem

Multiple copies of near-same code:

- Hard to maintain
- Get out of synch

FRANKENSOURCE: Lots of stuff bolted on:

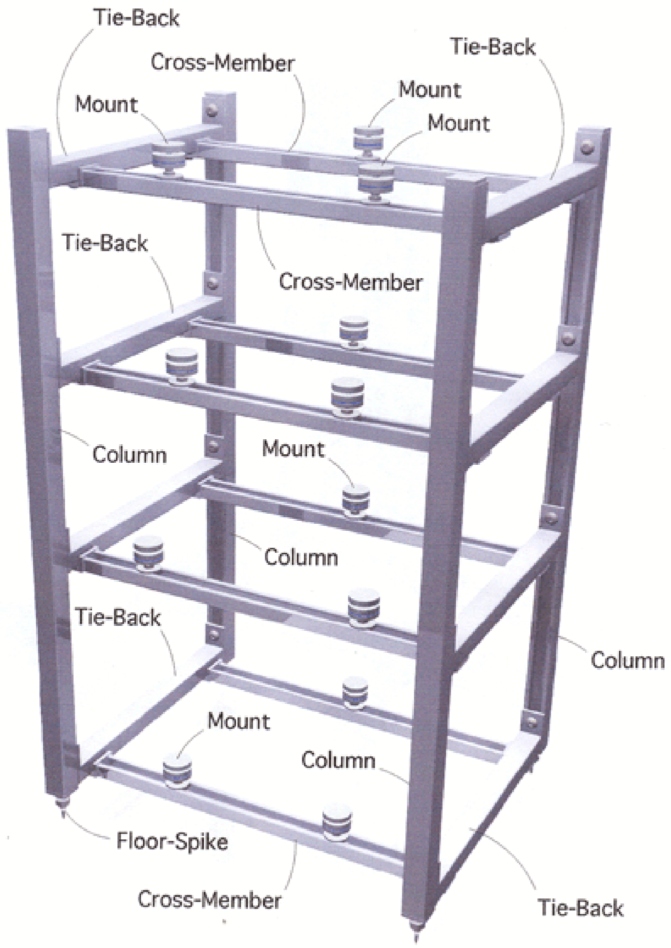
- Can’t understand science intent
- Can’t safely extend algorithms

Code Duplication: Solution

Scaffolding functions that call functors or virtual methods for the parts that vary[1]

Factories create correct functors

```
scaffolding()
{
   doA();
   ...
   doB();
   ...
   doC();
   ...
   doD();
}
```



---

[1] Template Method pattern

State and Copying

- Types of State
  - Identity state: Unique for that object: Never copy
  - Context state: About object associations: Copy parts in construction that apply to new object
  - Value state: Copy in constructor and assignment
- Often easiest to disallow copying but then can’t hold object in STL containers (but can hold pointers to them) or pass them by value

True Exception Safety is Hard

- Hard to get this right
- Has a run-time cost if you do
- Can’t just sprinkle this around an app
- Limit it to recoverable problems

Thread Safety is Hard

- Hard to get this right
- Has a run-time cost if you do
- Portability issues (until C++0X)
- There are multiple levels of thread safety
  - Weak / reentrant
  - Strong: Needs a lot of mutex locking
- Design to the lowest level you can
- Avoid non-const globals and class static data
- Consider processes instead of threads

Large Scale Physical Design

- Decompose into packages/namespaces
  - Low coupling between packages: Interfaces
  - Consider testing dependencies: Mocks/Stubs
- Acyclic dependencies between packages
  - Prefer downward, then sibling dependencies
- Minimize **`#include`** coupling
  - Fine grained: ***`Class`*****`.hh`**/**`.cc`** + **`all.hh`** + **`pkg.hh`**
  - Forward declaration headers ***`Class`*****`.fwd.hh`**

C++ Unit Testing

- Test class method and component behavior
  - Normal & edge cases
  - Invariants & pre/post-conditions → assertions
- Use mocks for externals: network resources, …
- Most UT frameworks take too much work
  - Should only touch one place to add/change a test
  - CxxTest and UnitTest++ meet this
- Run tests automatically at checkin to CI

C++ “Live” Testing

- Assert for bugs (programming errors)
  - Class invariants (DBC)
  - Function pre- and post-conditions (DBC)
  - Mid-function state
- Exceptions or `if` blocks for unexpected inputs or system state (out of memory, network error, …)

C++ Unit Testing

```
#include <UnitTest++.h>
#include <ObjexxFCL/byte.hh>

SUITE(ObjexxFCL_byte)
{
  using namespace UnitTest;
  using namespace ObjexxFCL;

  TEST(Construction)
  {
    byte b( 22 );
    CHECK_EQUAL( b, byte(22) );
    CHECK_EQUAL( b, 22 );
  }

  TEST(Assignment)
  {
    byte b( 55 );
    CHECK_EQUAL( b, 55 );
    CHECK_EQUAL( b += 2, 57 );
  }
}
```

The main source file:

```
#include <UnitTest++.h>

int main()
{
  return UnitTest::RunAllTests();
}
```

```
$ make
```

```
$ ObjexxFCL.unit
Success: 130 tests passed.
Test time: 0.02 seconds.
```

Failures show files/lines

C++ Tools

- Unit Testing
  - CxxTest
  - UnitTest++
- Dynamic Testing
  - valgrind (memory)
  - mpatrol
  - Electric Fence
  - Insure++ ($$)
- SCM
  - Subversion
  - Mercurial
  - Git
- Bug/Issue Tracking
  - Trac
  - Bugzilla
  - Mantis

OO C++ Development is Painful

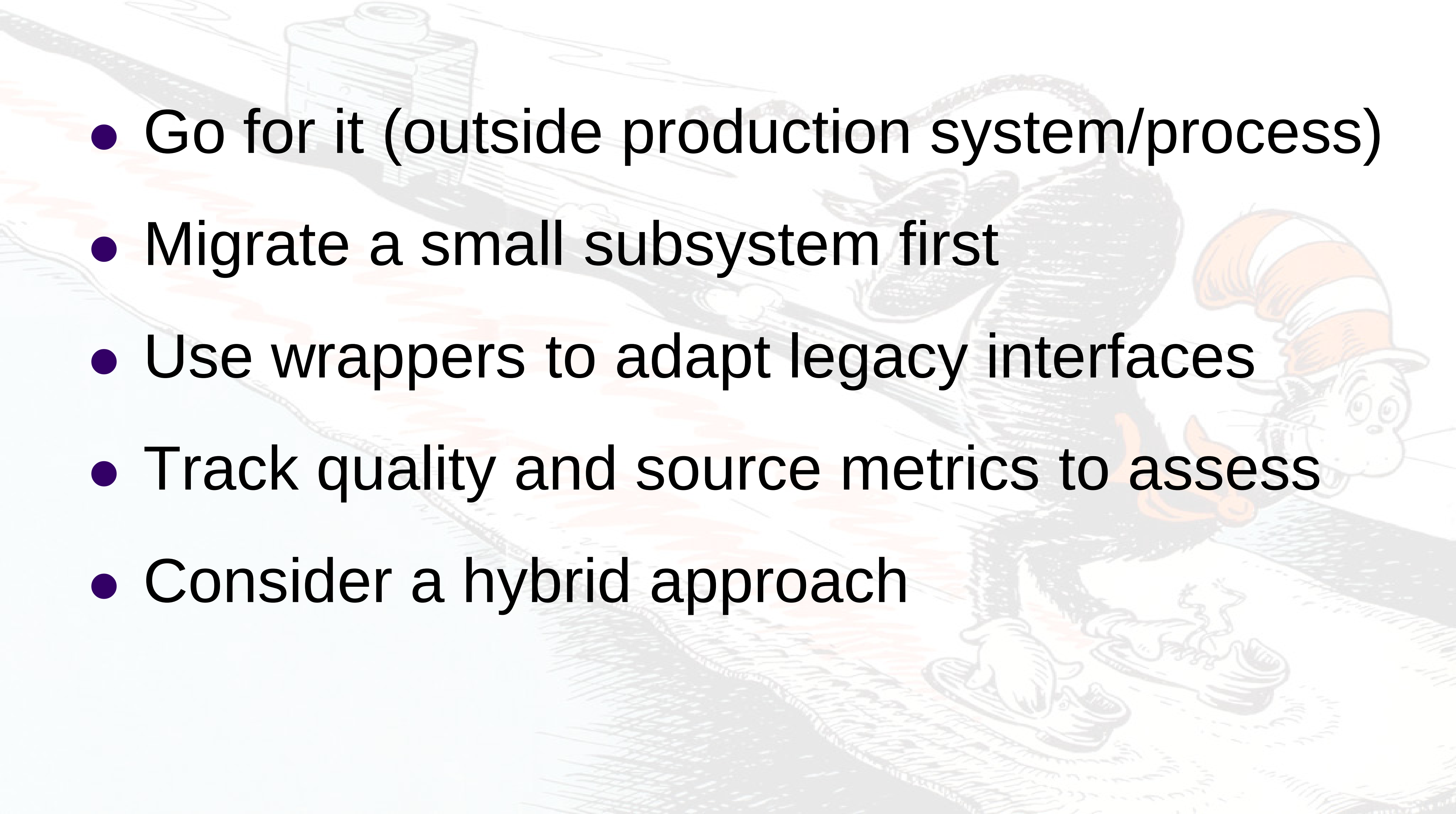
- Discipline needed to avoid buggy code
- Tasks take a lot of code to accomplish
- Header inclusion is primitive
- Compile/link/test cycle is slow
- Error messages are horrid
- Nifty template tricks are powerful but make compiles slower and error message worse
- Static types cause a lot of the pain (and speed)

Hybrid Systems

- Hybrid approach is becoming popular
- Good OO scripting languages (Python, Ruby, …)
- These are much easier and more productive (RAD)
- Can get back speed by writing the hot-spots in C/C++
  - Boost.Python to create interfaces
- Many GUI libs have Python & Ruby bindings
  - PyQt, QtRuby, PyGTK, RubyTGK2, wxPython, …
- Can treat scripts as RAD prototype for mig. to C++

Where to Go With OO

Where to Go With OO



- Go for it (outside production system/process)
- Migrate a small subsystem first
- Use wrappers to adapt legacy interfaces
- Track quality and source metrics to assess
- Consider a hybrid approach