Migrating to OO

EnergyPlus C++

Stuart G. Mentzer
Objexx Engineering, Inc.

Why OO?

Manage complexity via abstraction

Self-managing objects improve reliability

Interfaces give extensibility & pluggability

Component packages give modularity

OO C++: Good News

Enables powerful robust systems

- Modular
- Extensible
- Testable
- High performance

OO C++: Bad News

Takes a lot of work and code in C++

Abstractions slice application in new ways

- Poorly decomposed systems are inflexible
- Finding decoupling “pinch points” takes experience

Abstraction inhibits quick hacks (*not all bad*)

Learning robust idioms & patterns takes time

Not everyone can become proficient at OO

Getting to OO

Turning a procedural code into OO is a big job

Clean slate for crusty legacy components

Evolutionary migration is usually best

Clean Slate Approach

Need OO dev team

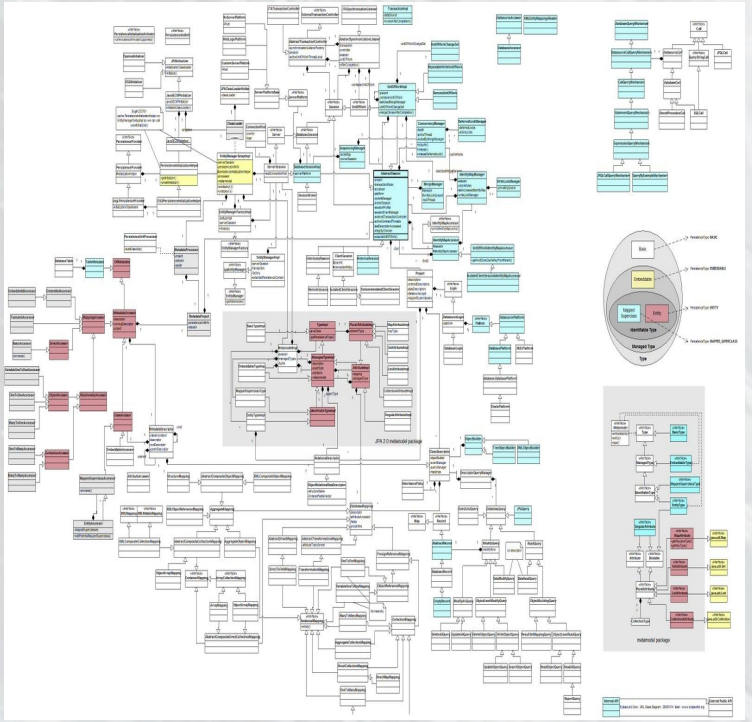
Architect high-level design

Build “mini” prototype 1st

Refine and embellish

Ambitious

Delayed benefits



Evolutionary Approach

Develop OO guidelines

Refactoring phases:

- Support developers

Architecture evolves

Smaller steps

Less ambitious

Can EnergyPlus Migrate to OO? (Yes)

You can introduce OO to a procedural code

Code can evolve towards an OO design

You don’t need to make everything OO

Various migration strategies...

Migration Sequencing Strategies

Most Useful: Biggest Bang theory

Top-Down:

- High-level parts are most important
- They provide the application framework

Bottom-Up: Low-hanging fruit

- Attack peripheral/utility classes early
- They are simpler & have fewer dependencies

Typically use all of these approaches

Migration Process

Choose a migration strategy

Gradually replace legacy components with OO

Aim for frequent *functional* milestones

Build tests as you go

OO Principles

Focus on reducing dependencies

- Good for extensibility and compile speed
- Acyclic package dependencies

Insulate *client* code from object internals

- Interface-based designs: Clients know *what* not *how*
- Factories localize dependence on concrete types
- Pluggable and fast compilation

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

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

CRC Card

Class *Name*

Responsibilities

- **Interface (Services)**
- **Invariants (Promises)**

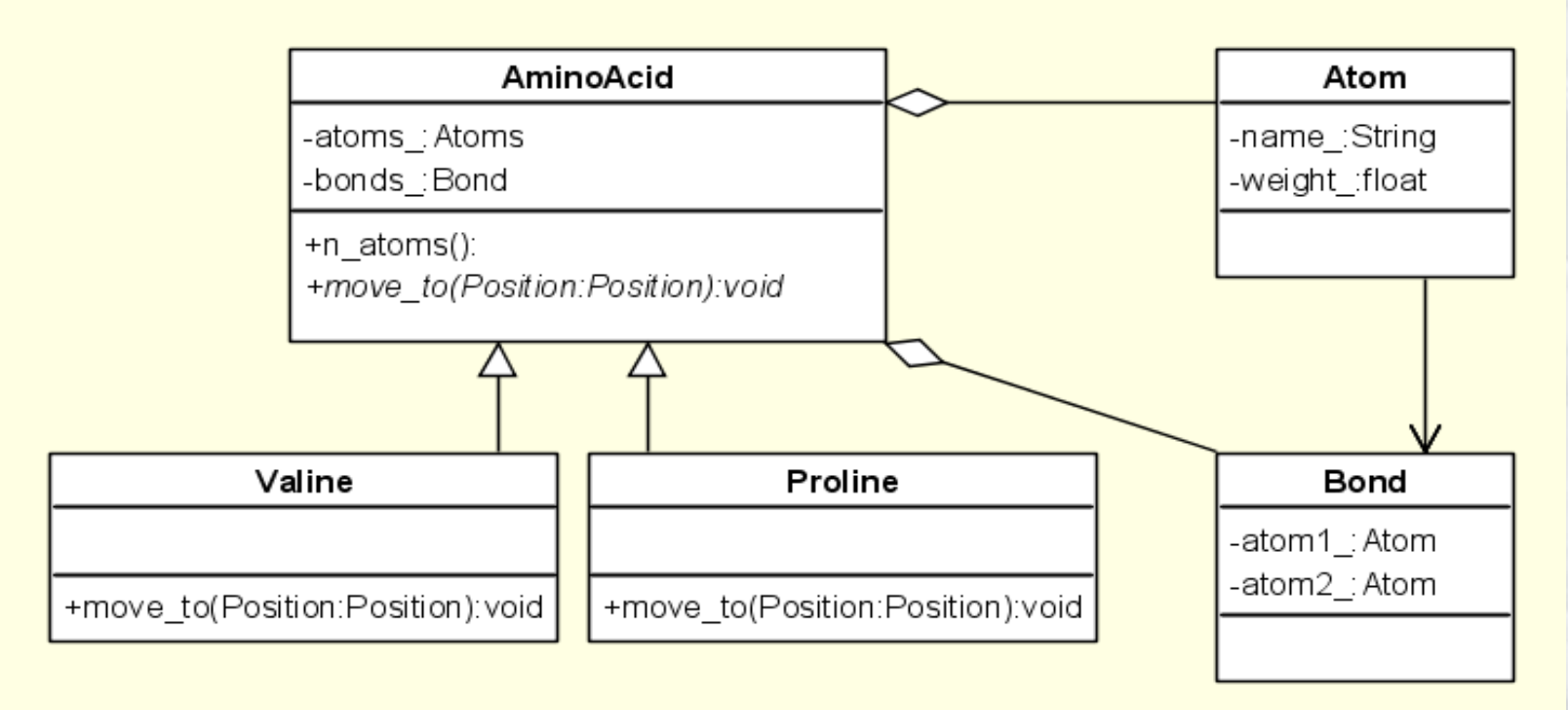
Collaborators

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

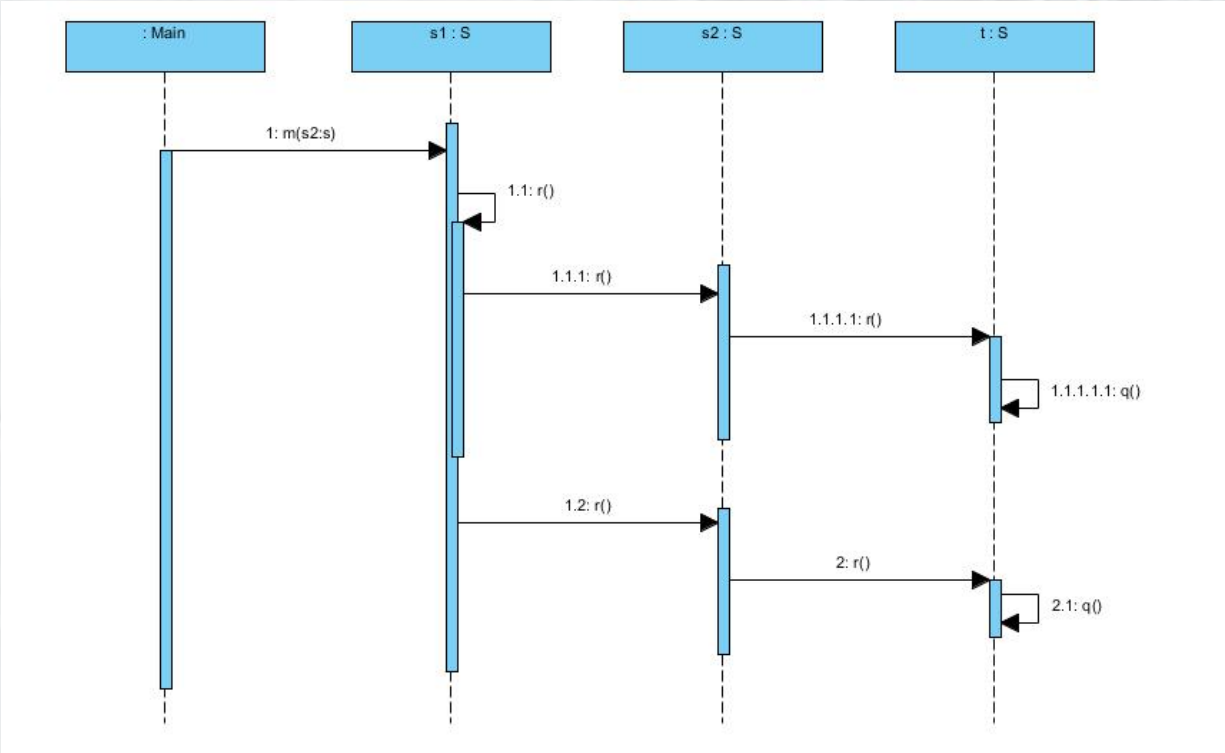
Class Profile

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



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!

Phase 1: Basics

structs -> classes: methods, private data, ...

Find hierarchies / Add base classes

Replace if blocks of type tests with virtual calls

Reduce global data

Phase 1: Payoffs

Reduce boilerplate code

Improve maintainability

Enable assert & unit testing

Insulate clients of types from some details

Phase 2: Extend Object Model

Improve object granularity

Find more objects

Bring more functions into classes

Modern C++ data structures and smart pointers

Phase 2: Finding Objects

Nouns: Building, Zone, Pump, Coil, …

Verbs: Iteration, Minimization, Controller alg.

- Strategy Pattern / Functors

Factories separate point of (concrete) object creation from (abstract) object users: Decouple

Struct Should be a Class If...

Code to manage/check state reappears:

- Principal angle in [0, 2π]
- Rotation matrix orthogonality
- Unit quaternion renormalization

Objects manage their own state

Removes clutter from engineering code

Reduces bug exposure

You Need a Class Hierarchy If ...

```
if ( ControlTypeNum == 0 ) { // Uncontrolled
    ...
} else if ( ControlTypeNum == SingleHeatingSetPoint ) {
    ...
} else if ( ControlTypeNum == SingleCoolingSetPoint ) {
    ...
} else if ( ControlTypeNum == SingleHeatCoolSetPoint ) {
    ...
} else if ( ControlTypeNum == DualSetPointWithDeadBand ) {
    ...
}
```

Looks Like a Factory

```
if ( SameString( Alphas( 4 ), "CaseTemperatureMethod" ) ) {
   RefrigCase( CaseNum ).LatentEnergyCurveType = CaseTemperatureMethod;
} else if ( SameString( Alphas( 4 ), "RelativeHumidityMethod" ) ) {
   RefrigCase( CaseNum ).LatentEnergyCurveType = RHCubic;
} else if ( SameString( Alphas( 4 ), "DewpointMethod" ) ) {
   RefrigCase( CaseNum ).LatentEnergyCurveType = DPCubic;
} else {
   ShowSevereError( RoutineName + trim( CurrentModuleObject ) + …
   ErrorsFound = true;
}
```

Functors ‘R Us

```
if ( SELECT_CASE_var == CaseTemperatureMethod ) {
   DefCapModFrac = CurveValue( DefCapCurvePtr, TCase );
   DefrostRatio = max( 0.0, ( 1.0 - ( RatedAmbientRH - ZoneRHPercent ) …
} else if ( SELECT_CASE_var == RHCubic ) {
   DefrostRatio = max( 0.0, CurveValue( DefCapCurvePtr, ZoneRHPercent ) );
} else if ( SELECT_CASE_var == DPCubic ) {
   DefrostRatio = max( 0.0, CurveValue( DefCapCurvePtr, ZoneDewPoint ) );
} else if ( SELECT_CASE_var == None ) {
   DefrostRatio = 1.0;
}
```

Costs of Type-Selection If Blocks

Adding types is a large task

Maintaining all the if blocks is a big bug risk

Concrete type set dependency everywhere

Factories localize concrete dependency

- Reduces build time and code clutter

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;
};

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

Stage 2: Migrate Behavior

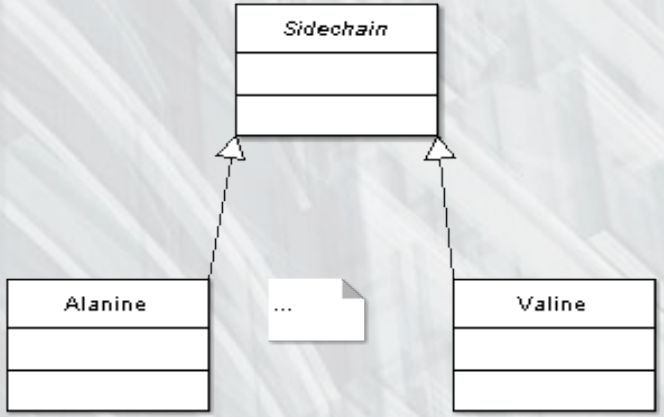
```
struct Position
{
  // Default Constructor
  Position() :
    x( 0.0 ),
    y( 0.0 ),
    z( 0.0 )
  {}

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

  // Length
  double
  length() const;

  // Normalize
  Position &
  normalize();

  // Data
  double x, y, z;

};
```

Stage 3: Hidden Data

```
class 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)
  {}

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

  // X coordinate
  double &
  x()
  { return x_; }
  …
private: // Data

  double x_, y_, z_;

};
```

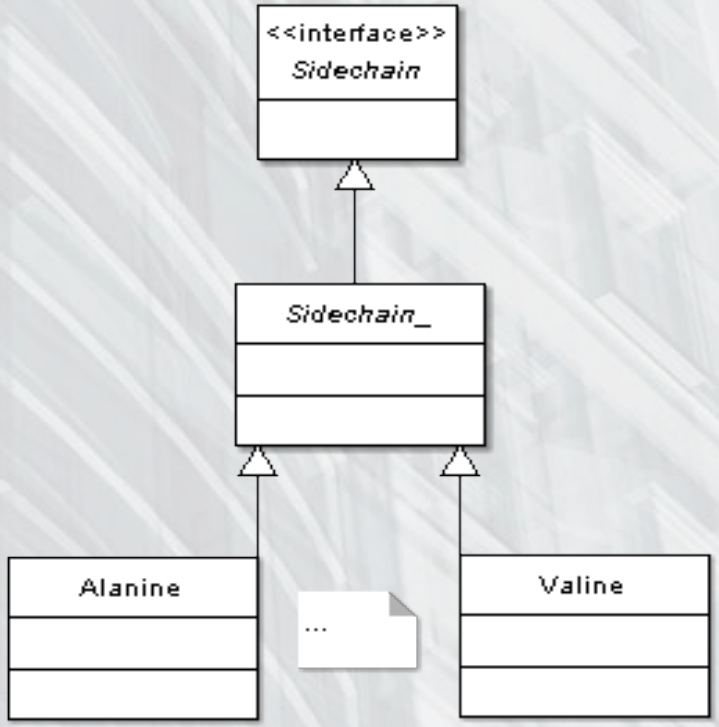
Stage 4: Polymorphism

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



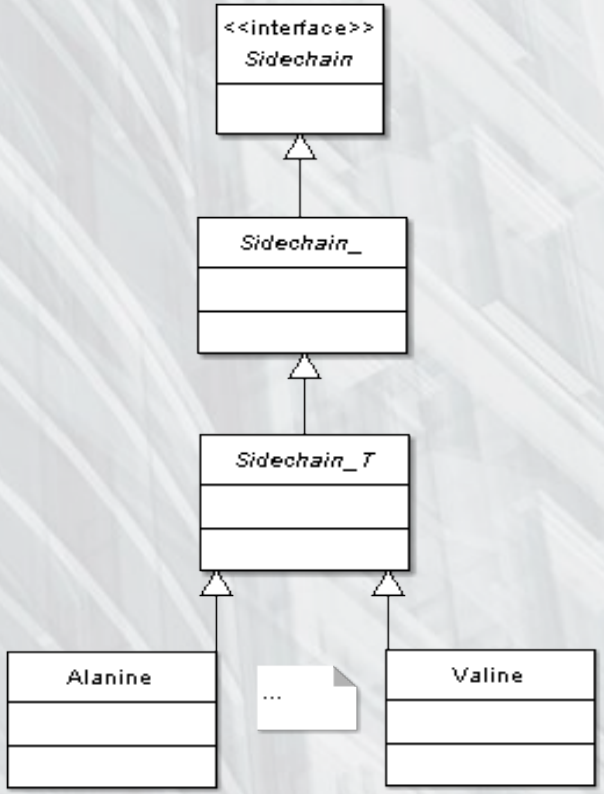
Stage 5: Interfaces

- Observation: Users dependency on shared data or methods is bad
- Solution: Interface root with only pure virtual functions and no data



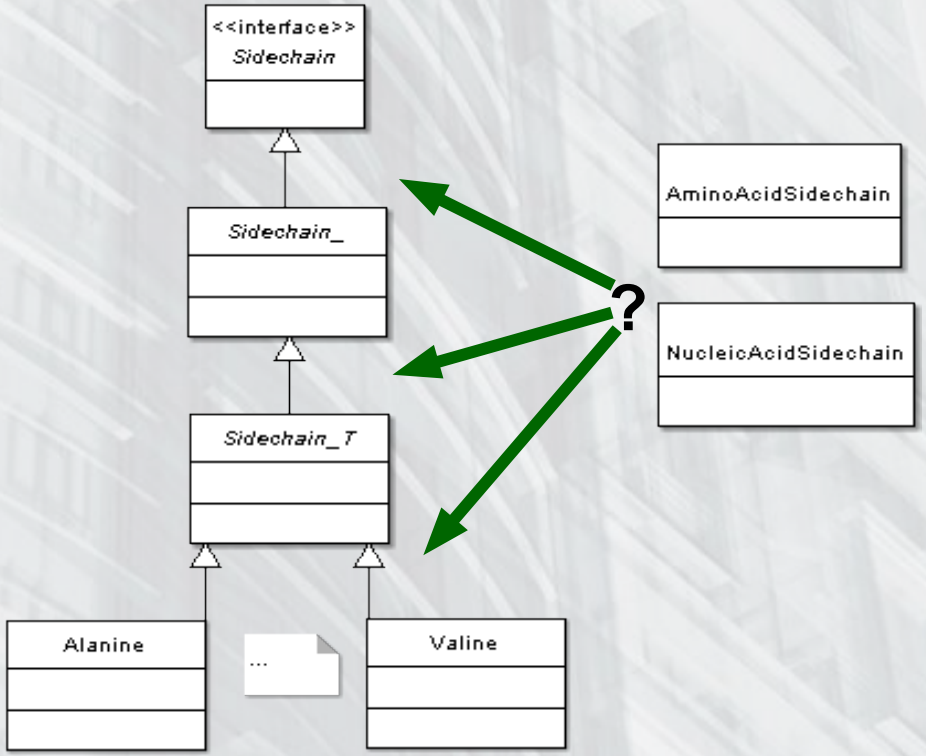
Stage 6: Templates

- Same behavior diff. types
- Behavior sep. as Policy
- Templates bring:
  - Code reduction: GOOD
  - Dependencies: BAD
- CRTP: Template base class is a template argument unique



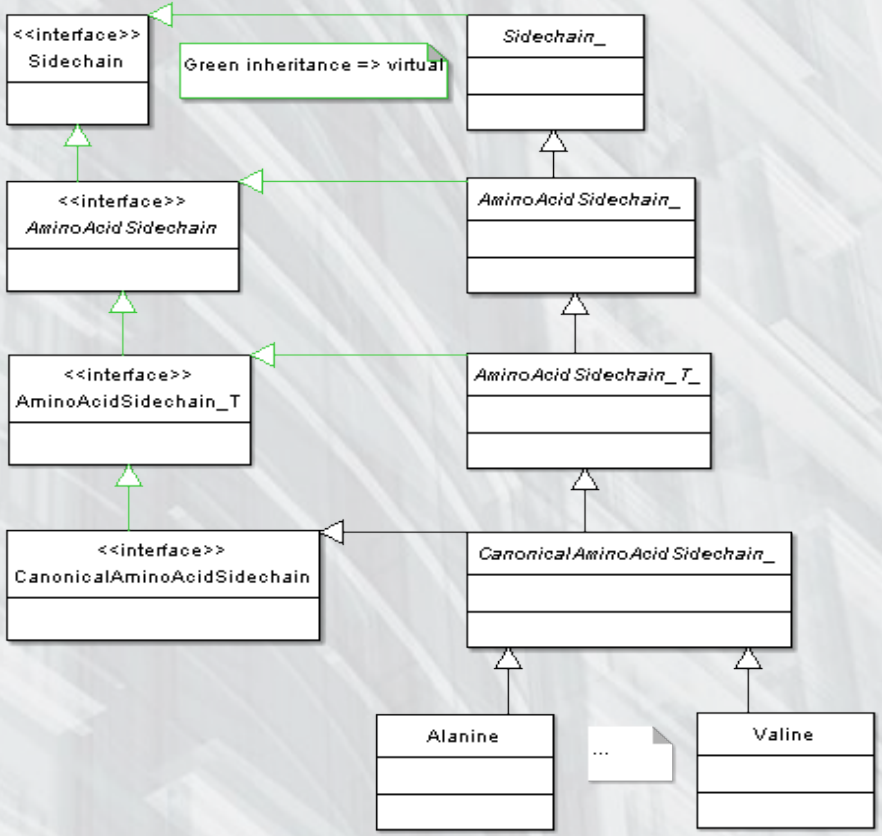
Are We Done?

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



Stage 7: Lattice Hierarchy

- Interface chain
- Multiple inheritance
- Not simple



Composite Objects: Structure

Contains objects of (usually dynamic) types

```
class Building
  std::set< Zone > zones; // No good if Zone abstract
```

These work:

```
  std::set< Zone * > zones; // Must manage lifetimes
  std::set< std::shared_ptr< Zone > > zones;
```

Composite Objects: Operation

Delegate varying behaviors to objects:

```
for ( auto & zone : bldg.zones )
   zone->run_cycle(); // Depends on zone
```

Require multiple dispatch patterns if varying behavior depends on multiple types

Code Duplication: Problem

Multiple copies of near-same code:

- Hard to maintain
- Get out of synch

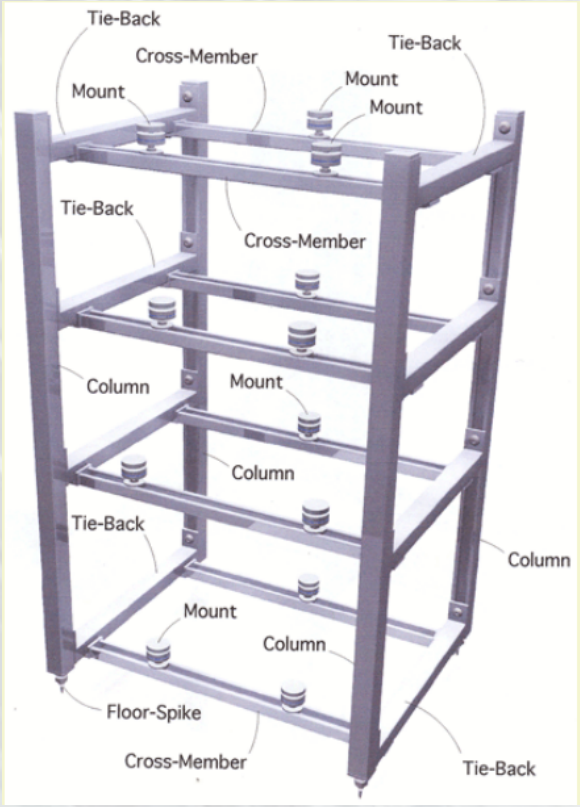
Frankensource:

- Keep bolting on more code to add subtypes
- Can’t understand intent
- Can’t safely extend algorithms

Code Duplication: Macro Solution

Scaffolding functions that call functors for the variable parts[1]

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



[1] Template Method pattern

Code Duplication: Micro Solution

Non-member scaffolding function:

- Function-like objects pick the right behavior
- Actual objects are set elsewhere so scaffold is stable

Scaffolding function is member of base class:

- Calls virtual functions to do the work

Result: Scaffolding code is stable while behavior is extensible: Don’t need to copy & paste

Phase 3: Advanced OO

Higher-level idioms/patterns/techniques to capture complex relationships & behavior

- Composite pattern for nested containment
- Compile-time type flexibility via templates
- Modular type flexibility via Pluggable Factories
- Functor hierarchies for pluggable algorithms (Strategy)
- Inheritance + templates: mixin/policy designs

Phase 3: Advanced OO

Application domain modeling

Select appropriate idioms/patterns

- Loose coupling
- Low dependency
- Avoid class explosion

Implementation Tasks

Utility and “glue” classes reduce code repetition

Support functions: numeric, array, string, ...

Layered namespace scoping to avoid conflicts

Unit and component-level tests

Design Patterns

Core OO patterns => Better solutions

Strategy

Observer

Factories (various)

Visitor / Multiple dispatch

CRTP

Proper C++ Classes

- RAII: Get resources in ctor and release in dtor
  - Don’t hand out pointers hoping user will/won’t delete
  - Smart pointers are an alternative
- 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 reference or const data
- Automatic copy constructor and assignment do memberwise shallow copying

OO C++: Rules of Thumb

Use type safety to make bugs compile-time

Use forward declarations maximally

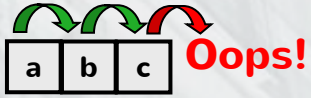
Use “live” assertion testing liberally (DBC)

Use unit testing: It can be pretty painless

Make source self-documenting & clear

External docs/wiki to clarify the subtleties

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
- Project guidelines for lifetime management, naming, style, …
- Assert invariants and pre-/post-conditions
- Code reviews (pair or group)

OO C++ Development is Painful

- Discipline needed to avoid buggy code
- Header inclusion is primitive
- Compile/link/test cycle is slow
- Error messages are horrid
- Static typing => Lots of code (but gives speed)
- Templates are powerful but make compiles slower and error message worse

Goals for New/OO Code

Reliability: Fewer points of potential failure

Clarity: Science not obscured by bookkeeping

Natural Data Structures: Clean & efficient

Modularity: Small self-contained subsystems

Decoupling: Subsystems, libs, platforms

Layering: Dependency management

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`

Architecture

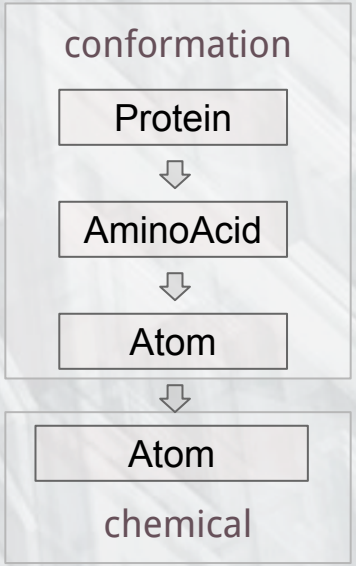
Dependency management

- Directional layered design
- Modular & faster builds

Decoupling via abstraction

Finding the right abstractions

Clean interfaces and responsibilities



Management Aspects

A strong OO architectural vision is essential

Support for non-OO developers

Scrums / Pair programming / Agile *buzzword*

Code review to assure OO best practices

Useful Idioms

- Virtual construction: clone() and create()
- Named constructors: Avoid flag arguments
- Named keys (no “magic” numbers)
- Smart pointers: Manage ownership/lifetime
  - Intrusive smart pointers are faster and safer
  - Beware of thread-safety issues
  - Keep pointer graph acyclic

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

Useful C++ Libraries

Boost

Loki (Modern C++ Design: Fun with Templates)

Blitz++ (Array expressions)

Armadillo | Boost uBLAS (Linear Algebra)

Qt | wxWidgets (GUI)

OpenSceneGraph | Ogre3D (3D visualization)

C++/OO Resources

Books

Accelerated C++

The C++ Programming Language

Effective C++, More…

The C++ Standard Library

Effective STL

Large-Scale C++ Software Design

Design Patterns

Web

C++ Best Practices [Meyers, ...]

Boost library

C++ FAQ / LITE / FQA

C++ containers, iterators, algorithms

Performance-critical design

Questions