Cranking Policies Up

Andrei Alexandrescu

Conclusions

- Eight years later, systematic design automation using policies is still elusive
- In spite of all the snazzy features acquired by languages

- Some progress has been made
- Yet too many designs require high implementation effort
- Nirvana = Static Introspection + Code generation

This Talk

- Review of Policy-Based Design
- Red Code, Green Code
- Red Data, Green Data
- Null Object, Black Holes, and White Holes
- Conclusions

Policy-Based Design Refresher

What is PBD? Simply Put:

1. Take a design offering various tradeoffs
2. Separate tradeoffs in *orthogonal* concerns
3. Encode each concern into a specialized type (policy) obeying a *static interface*
4. Write a *host class* assembling the policies

What is PBD? Simply Put:

1. Take a design offering various tradeoffs
2. Separate tradeoffs in *orthogonal* concerns
3. Encode each concern into a specialized type (policy) obeying a *static interface*
4. Write a *host class* assembling the policies
5. . . .
6. Profit!

PBD Success Stories

- Smart pointers
- Strings, arrays, containers
- Certain Pattern Automation: Factory, Visitor, Observer, . . .
- PBD does not “implement” patterns
- PBD does use and apply patterns in finding the right decomposition
- Recent work: “Red Code, Green Code” by Scott Meyers

PBD Success Stories

- Smart pointers
- Strings, arrays, containers
- Certain Pattern Automation: Factory, Visitor, Observer, . . .
- PBD does not “implement” patterns
- PBD does use and apply patterns in finding the right decomposition
- **Recent work: “Red Code, Green Code” by Scott Meyers**

Red Code, Green Code

Red Code, Green Code

- Express code features as types
- Portable code, thread-safe code, exception-safe code, reviewed code. . .
- Features are *transitive*

  Portable code should only call portable code
- Features are *combinable*

  Portable *and* exception-safe code
- Features are are *contravariant*

  More features $<:$ Less features

Defining Features

```
struct ThreadSafe {};
struct ExceptionSafe {};
struct Portable {};

typedef boost::mpl::vector<
ThreadSafe, ExceptionSafe
> TESafe;
template<class Features> struct MakeFeature
{
    ...
}
```

Transitive & Contravariant

- Say a function fulfills certain features
- It should only call functions fulfilling those features
- Plus possibly other, unrelated features
- Please note: this is exactly the *opposite* of inheritance
- Inheritance: you can pass a derived as a base
- Features: you can pass a base as a derived!

Combinable

- Arbitrary number of code features can be created and used
- They can be arbitrarily superposed
- Superposition result does not depend on order
- Please note: this is quite the *opposite* of templates
- Templates: order of arguments is essential
- Features: order of arguments is irrelevant

Using a Set of Features

```
void f(int x, double y,
MakeFeatures<TESafe>::type features)
{
   ...                           // no
}
```

Using a Set of Features (cont’d)

```
typedef boost::mpl::vector<
ThreadSafe, ExceptionSafe, Portable
> TEPSafe;

void g(MakeFeatures<TEPSafe>::type features
```

Enforcing Features' Structure $(+)$

```
void f(int x, double y,
MakeFeatures<TESafe>::type features)
{
    ...
    g(features);                        // fine
    ...
}
```

- g is thread-safe and exception-safe, and also portable
- *Contravariance* of code features

Enforcing Features’ Structure (—)

```
void g(MakeFeatures<TEPSafe>::type features
{
    int xVal, yVal;
    ...
    f(xVal, yVal, features);// error!
    ...
}
```

- `f` does not respect the Portable requirement
- Can’t call non-portable code from portable code

Actual Error Message (Literary Klingon)

```
articlecode.cpp: In function ’void g(
CodeFeatures::Features<
boost::mpl::v_item<
CodeFeatures::Portable
, boost::mpl::v_item<
CodeFeatures::ExceptionSafe
, boost::mpl::v_item<
CodeFeatures::ThreadSafe, boost::mpl::vector0<mpl_::na>
, 0
>, 0
>, 0
>
>
)’:
articlecode.cpp:32: error: conversion from ’CodeFeatures::Features<
boost::mpl::v_item<
CodeFeatures::Portable
, boost::mpl::v_item<
CodeFeatures::ExceptionSafe
, boost::mpl::v_item<
CodeFeatures::ThreadSafe, boost::mpl::vector0<mpl_::na>, 0
>, 0
>, 0
>
>’ to non-scalar type ’CodeFeatures::Features<
boost::mpl::v_item<
CodeFeatures::ExceptionSafe
, boost::mpl::v_item<
CodeFeatures::ThreadSafe, boost::mpl::vector0<mpl_::na>, 0
<, 0
>
>’ requested
```

Remark

If there’s any hope to automate designs, custom error messages are a must.

Status

- C++: no
- D: yes (`static assert`)
- C++1x: yes (`static_assert`)

Superposition

- Define a *total ordering* over features (types)
- `MakeFeatures` sorts by that order

  ⇒ Initial order becomes irrelevant
- How to do that?

Scott’s Solution

```
namespace CodeFeatures {
      namespace mpl = boost::mpl;
      using mpl::_1;
      using mpl::_2;

      template<typename S, typename T>
      struct IndexOf:
         mpl::distance<typename mpl::begin<S>::type,
                              typename mpl::find<S, T>::type>
      {};

       template<typename Unordered>
       struct Order:
          mpl::sort<Unordered,
                         mpl::less<IndexOf<AllCodeFeatures, _1>,
                                        IndexOf<AllCodeFeatures, _2> > >
       {};

       template<typename CF>
       struct MakeFeatures {
             typedef
                Features<typename mpl::copy<typename Order<CF>::type,
                                                          mpl::back_inserter<mpl::vector0<> > >::type>
                type;
          };

    }
```

(You didn’t see the half of it.)

Scott's Solution, Summarized

- Define a type vector containing *all features*
- Sort by a feature's position in that vector
- Extremely coupled: new features must be added to "the registry"
- Mechanics-heavy

Total Ordering of Types

Three possibilities:

1. Manually associate integral IDs with properties; sort by ID
2. Sort by type name

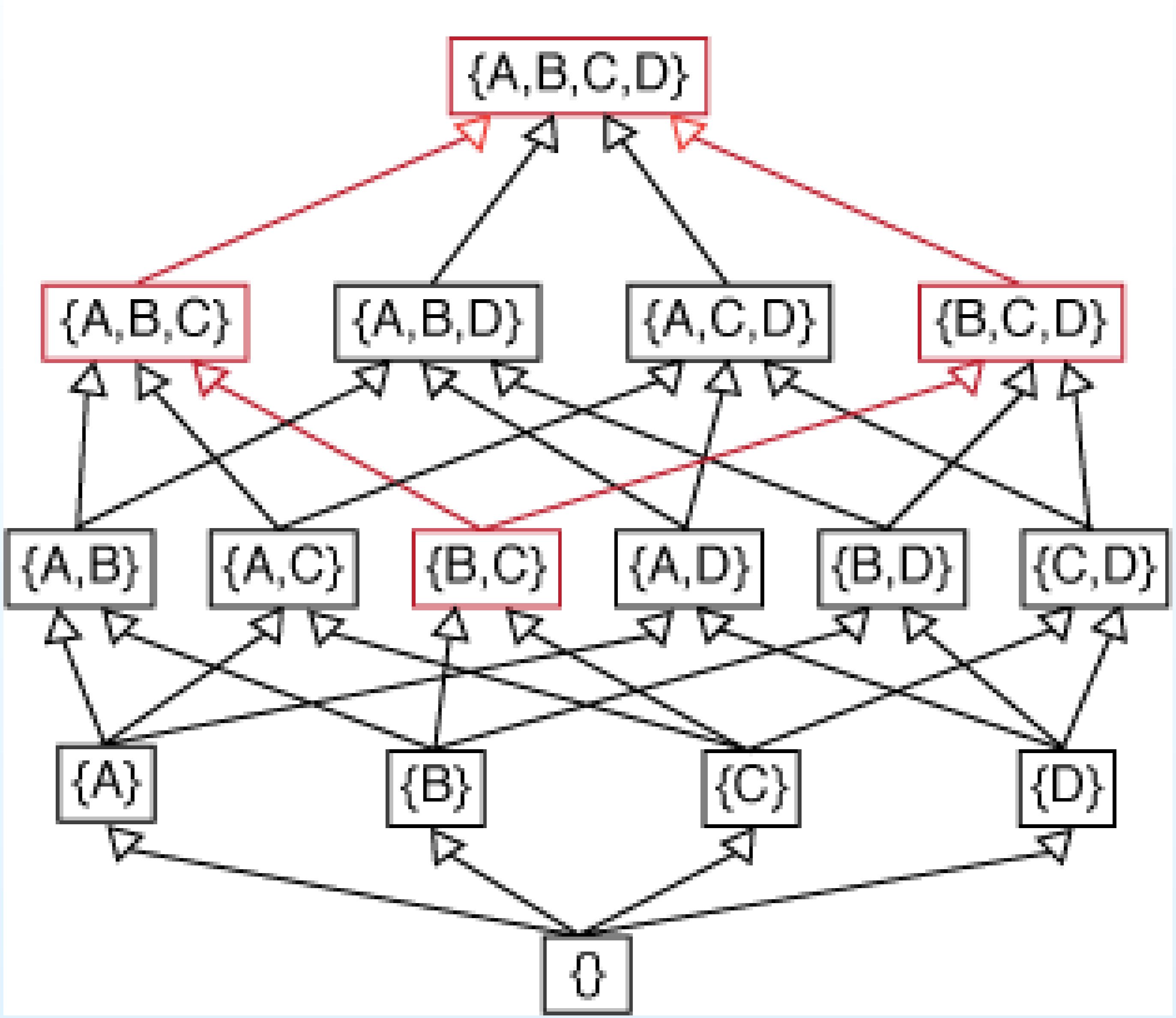
   C++ does not allow that

3. Express properties as strings

   D can do string manipulation statically

   C++ can too, if all strings are 8 characters

Contravariance



Remark

To automate design, compile-time string processing is a must.

Status

- C++: no
- D: yes (`static assert`)
- C++1x: yes (`static_assert`)

Red Data, Green Data

Typing the Untypable

- Certain properties are difficult to encode as types

  Example: “array is sorted”

  Example: “array is normalized”

  Example: “array has no negative elements”

- Could define `SortedArray`, `NormArray`, `NonNegativeArray`

- How about `SortedNormArray`?

More Examples

- String has only capital letters
- Array of strings, each no more than 6 characters
- String is normalized
- String is *tainted*
- Object is *unaliased*

Properties

- These properties are seldom worth defining types for
- Yet applications routinely make various assumptions about data
- Properties may superpose $\Rightarrow$ combinatorial explosion
- Checking throughout is often not an option

Should `binary_search` call `is_sorted`?

Assume* Shims

```
template<class T>
struct AssumeNormalized {
   T& payload;
};
template <class T>
AssumeNormalized<T>
assumeNormalized(T & input);
```

Using an `Assume*` Shim

```
void
entropy(AssumeNormalized<vector<double>> v) {
    ... use v.payload ...
}
// call
vector<double> v;
normalize(v);
entropy(assumeNormalized(v));
```

Composing Properties

- How about a vector that’s positive and normalized?
- . . . positive, normalized, sorted?
- . . . positive, normalized, sorted, and not degenerate?

Composing Properties

- How about a vector that's positive and normalized?
- . . . positive, normalized, sorted?
- . . . positive, normalized, sorted, and not degenerate?

- Combinatorial explosion very similar to design choices in policy-based design!

Multiple Parameters to the Rescue (usage)

```
struct Normalized;
struct Positive;

// usage
vector<float> v;
fun(Assume<Normalized, Positive>::on(v));
```

Multiple Parameters to the Rescue (impl.)

```
template<typename T, typename... Ps>
struct AssumeImpl {
   typedef Typelist<Ps...> Properties;
   T& payload;
};

template<typename... Ps>
struct Assume {
   AssumeImpl<T, Ps...> on(T& v);
};
```

Multiple Parameters to the Rescue?

- `Assume<A, B>::on(v)` and `Assume<B, A>::on(v)` are distinct types!
- If a function expects them in the `A, B` order and you have them in `B, A` order, the call will not go through
- What to do?

Multiple Parameters to the Rescue?

- `Assume<A, B>::on(v)` and `Assume<B, A>::on(v)` are distinct types!
- If a function expects them in the `A, B` order and you have them in `B, A` order, the call will not go through
- What to do?
- We need to define a *total ordering* over possible properties

Designs That *Should*

Null Object Pattern

- Replace a null pointer with a valid object that does nothing
- Defined behavio(u)r: no more checking for null
- Useful in prototyping
- Useful for e.g. “null stream,” stub observers, terminators

Black Holes & White Holes

- One approach: “absorb” all calls without doing anything: *Black Hole*
- Another: reject every operation by throwing an exception *White Hole*
- Both are useful in applications
  - Standalone
  - Base classes (implement a fraction of an interface)

Implementation

```
interface Foobar {
   virtual void foo(int);
   virtual int bar(string);
}
class BlackHoleFoobar : Foobar {
   virtual void foo(int) {}
   virtual int bar(string) { return int{}; }
}
class WhiteHoleFoobar : Foobar {
   virtual void foo(int) {
      throw exception("Unimplemented: foo");
   }
   virtual int bar(string) {
      throw exception("Unimplemented: bar");
   }
}
```

That’s awful!

Automating Null Objects

- Such code should be automatically available

```
BlackHole<Foobar>

WhiteHole<Foobar>
```

- Should work with both interfaces and classes
- Should be as fast as the hand-written implementation
- What do we need to make it work?

We Need

- Compile-time introspection

  Enumerate members of a class/interface

- Compile-time code generation

  For each member found, generate customized code

- No popular compiled language offers both

We Need

- Compile-time introspection

  Enumerate members of a class/interface

- Compile-time code generation

  For each member found, generate customized code

- No popular compiled language offers both

To Conclude (again)

- Design Automation is still an elusive goal
- Like the STL: not designing the language for design automation $\Rightarrow$ no design automation will be achieved
- Progress has been made
- Much more is to be done