Design by Introspection
DConf 2017

Andrei Alexandrescu, Ph.D.

2017-05-06
Design Patterns Recap

- Inspired by Christopher Alexander’s (actual) architecture work
- Reusable solutions to reoccurring problems within given contexts
- Heyday in early 2000s
- Overselling and rebuttals predictably followed
- Left a lasting influence on design methodologies
Policy-Based Design

- Coined by “Modern C++ Design” in 2001
- Enjoys use in C++, D
- Inducted in Wikipedia’s “hall of fame” at http://en.wikipedia.org/wiki/Programming_paradigm (along with 75 others)
Approaches on the same core issue:
Design elements reoccur in response to typical problems

Patterns: programmer is the generator
Policy-Based Design: programmer controls the generator
“[...] the Design Patterns solution is to turn the programmer into a fancy macro processor.”

– Mark Dominus
Policy-Based Design (PBD)

- Def: Assembling a design by mixing components (policies) during compilation
- Nothing new:
  - Interface-based programming
  - Template Method pattern
- Yet:
  - Compile-time assembly offers extra static checking
  - “Frictionless abstraction” makes PBD suitable for good design of low-level components
Segue to Policies

- Semi-automated “macro” preprocessing

+ Better software reuse
+ Excellent static checking
+ Ultimate efficiency in time and space
  - Run-time rigid
  - No graceful degradation
  - Compile-time dependent
struct Widget(T, Prod, Error) {
    private T _frob;
    private Prod _producer;
    private Error _errPolicy;

    void doWork() {
        ... rely on implicit interfaces ...
    }
}
Design by Introspection
“What would happen if we could arrange the atoms one by one the way we want them?”

– Richard P. Feynman
Core Idea

- Patterns: programmer “expands” mental macros
  - Total plasticity, no code reuse
- PBD: programmer assembles rigid macros
  - No plasticity, good code reuse

- DbI: programmer *molds* macros that communicate with, and adapt to, one another
  - Good plasticity, good code reuse
DbI Prerequisites

- DbI Input
DbI Prerequisites

- DbI Input
  - Introspect types: “What are your methods?”
  - Variant: “Do you support method xyz?”
DbI Prerequisites

- DbI Input
  - Introspect types: “What are your methods?”
  - Variant: “Do you support method xyz?”
- DbI Processing
DbI Prerequisites

- DbI Input
  - Introspect types: “What are your methods?”
  - Variant: “Do you support method xyz?”
- DbI Processing
  - Arbitrary compile-time evaluation
DbI Prerequisites

- DbI Input
  - Introspect types: “What are your methods?”
  - Variant: “Do you support method xyz?”
- DbI Processing
  - Arbitrary compile-time evaluation
- DbI Output
DbI Prerequisites

- DbI Input
  - Introspect types: “What are your methods?”
  - Variant: “Do you support method xyz?”
- DbI Processing
  - Arbitrary compile-time evaluation
- DbI Output
  - Generate arbitrary code
How does D stack up?

- DbI Input
How does D stack up?

- DbI Input
  - `tupleof, __traits, ...`
How does D stack up?

- DbI Input
  - `tupleof, __traits, ...`
- DbI Processing
How does D stack up?

- DbI Input
  - `tupleof`, `__traits`, ...
- DbI Processing
  - `CTFE`, `static if`, ...
How does D stack up?

- DbI Input
  - `tupleof, __traits, ...`
- DbI Processing
  - `CTFE, static if, ...`
- DbI Output
How does D stack up?

- DbI Input
  - `tupleof`, `__traits`, ...
- DbI Processing
  - CTFE, `static if`, ...
- DbI Output
  - `template expansion`, `mixin`, ...
Optional Interfaces
Optional Interfaces

- A DbI component typically prescribes:
  - \( n_r \) required primitives (may be 0)
  - \( n_o \) optional primitives
- Introspection queries for optionals
- What’s missing as important as what’s present

- Up to \( 2^{n_o} \) possible interfaces, in compact form!
Optional Interfaces: Aftermath

- Linear code for exponential behaviors
  - Includes state variations, too
  - *static if* the “magic design fork”
- No penalty for fat interfaces
- Graceful degradation
  - Old: Less capable components $\Rightarrow$ errors
  - New: Less capable components $\Rightarrow$ reduced features
Each use of `static if` doubles the design space covered
Realized Designs

- `std.experimental.allocator`: unbounded allocator designs in 12 KLOC
  - jemalloc: 1 allocator in 45 KLOC
- Collections: see talk by Eduard Stănileoiu
- `std.experimental.checkedint`: now
Checked Integrals

- +, +=, -, -=, ++, --, *, * may lose information
- Division by zero in /, /=
- -x.min negative for all signed types
- -1 == uint.max, -1 > 2u

- That’s pretty much it!
• Options that come at a runtime cost
  ◦ Integrate in the programming language
  ◦ Do away with fixed-size arithmetic altogether
• Have the programmer insert tests appropriately
  ◦ For an appropriate definition of “appropriately”
  ◦ Bulky, difficult to follow, fragile
Possible Designs (2/2)

- Designate “checked integral” types
- Hook all operations and insert checks
- User replaces primitive types with these
  - Selectively depending on safety/speed tradeoff
- Requires user-defined operator overloading
Design Challenges

- What gets checked: overflows? div0? negation? mixed-sign comparisons? conversions? some of the above—which?
- Type system integration: statically disallow some operators/conversions?
- Make it efficient (not easy!)

- Make it small
  - Proportional response
  - Not rocket surgery after all
Meta Design Challenges

- No trouble to implement any given behavior
- Much more difficult to allow behaviors that are as of yet unspecified
- Scaffolding scales poorly with behaviors
- “Sticker shock” of generic libraries
  - “You mean I need to use this 5 KLOC library coming with 20 pages of documentation to check a few overflows?”
Baselines (1/2)

- Mozilla’s CheckedInt for C++
  - 0.8 KLOC (without docs, unit tests)
  - Inefficient layout (“valid” bit with the integral)
  - Enforcement onus on user code
  - No configurability
  - Inefficient approach (checks separated from operations)
- Microsoft’s SafeInt for C++
  - 7 KLOC
  - Lavish documentation
  - The Death Star of checked integers
Baselines (2/2)

- **safe_numerics** for C++ by Robert Ramey
  - Policy-based Design in 5 KLOC (+ 5 KLOC tests)
  - Requires 6 Boost libs
- **checkedint** for D by T. S. Bockman
  - PbD in 5 KLOC, including docs
std.experimental.checkedint size

- 3 KLOC (code + unit tests + documentation)
- Code: 1200 LOC
- Tests: 900 LOC
- Documentation: 900 LOC

- Speed: comparable to hand-inserted checks
Overall Design

- “Shell with hooks” approach
- Shell: high-level language integration
- Hook: optional intercepts of ops/events
- Default hook: just abort on anything fishy

```c
struct Checked(T, Hook = Abort) if (isIntegral!T) {
   private T payload;
   Hook hook;
   ...
}
```
Stateless hook? No problem!

```c
struct Checked(T, Hook = Abort) if (isIntegral!T) {
    private T payload;
    static if (stateSize!Hook > 0) Hook hook;
    else alias hook = Hook;
    ...
}
```
Default should be configurable

- Good for "integers with NaN"

```cpp
struct Checked(T, Hook = Abort) {
    static if (hasMember!(Hook, "defaultValue"))
        private T payload = Hook.defaultValue!T;
    else
        private T payload;
    static if (stateSize!Hook > 0) Hook hook;
    else alias hook = Hook;
    ...
}
```
The Shell

- Factors all commonalities
- Handles qualifiers
- Drives hooks
- Type system integration (\texttt{bool}, \texttt{float} etc)
- Composition mediation

- Not needed/appropriate for all designs
Graceful Degradation

- Traditionally: insufficient capabilities ⇒ error
- New: Insufficient interface ⇒ less capabilities

Checked!(\texttt{int}, \texttt{void}) x;
// x behaves like vanilla int
...

- Useful for:
  - Validate approach through “dry run”
  - Control design through versioning
  - Cover a larger design space!
Example

```csharp
ref Checked opUnary(string op)() return
if (op == "++" || op == "--") {
    static if (hasMember!(hook, "hookOpUnary"))
        hook.hookOpUnary!op(payload);
...
```
Example (cont’d)

```c
else static if (hasMember!(Hook, "onOverflow")) {
    static if (op == "++") {
        if (payload == max.payload)
            payload = hook.onOverflow!"++"(payload);
        else
            ++payload;
    }
    else {
        if (payload == min.payload)
            payload = hook.onOverflow!"--"(payload);
        else
            --payload;
    }
} else {
    mixin(op ~ "payload;");
    return this;
}
```
Defined Hook Primitives

- **Statics:** `defaultValue`, `min`, `max`
- **Event handling:** `onBadCast`, `onOverflow`, `onLowerBound`, `onUpperBound`
Defined Hooks

- Abort
- Throw
- Warn: output issues to stderr
- ProperCompare: fix comparisons on the fly
- WithNaN: Reserve “not a number” value
- Saturate: sticky saturation instead of overflowing

- Your own
  - Average length: 50 lines
Hook Example

- No Pesky Comparisons

```c
struct NoPeskyCmps {
    static int hookOpCmp(Lhs, Rhs)(Lhs lhs, Rhs rhs) {
        const result = (lhs > rhs) - (lhs < rhs);
        if (result > 0 && lhs < 0 && rhs >= 0 ||
            result < 0 && lhs >= 0 && rhs < 0) {
            assert(0, "Mixed-signed comparison failed.");
        }
        return result;
    }
    alias MyInt = Checked!(int, NoPeskyCmps);
}
```
Flexibility

- No Pesky Comparisons—EVAR!

```c
struct NoPeskyCmpsEver {
    static int hookOpCmp(Lhs, Rhs)(Lhs lhs, Rhs rhs) {
        static if (lhs.min < 0 && rhs.min >= 0 &&
                   lhs.max < rhs.max || rhs.min < 0 &&
                   lhs.min >= 0 && rhs.max < lhs.max) {
            static assert(0, "Mixed-sign comparison of " ~
                           Lhs.stringof ~ " and " ~ Rhs.stringof ~
                           " disallowed. Cast one of the operands.");
        }
        return (lhs > rhs) - (lhs < rhs);
    }
    alias MyInt = Checked!(int, NoPeskyCmpsEver);
}
```
Composition
• Traditionally: Checked works with integrals

```c
struct Checked(T, Hook = Abort)
if (isIntegral!T) {
 ...
}
```

• New: Checked works with integrals or itself

```c
struct Checked(T, Hook = Abort)
if (isIntegral!T || is(T == Checked!(U, H), U, H)) {
 ...
}
```

• Unique opportunities, but also challenges
Reflexive Composition: Examples

- “The Pit of Success”
- Checked!(Checked!(int, ProperCompare))
  - Fix comparisons, abort on everything else
- Checked!(Checked!(int, ProperCompare), WithNaN)
  - Has NaN, fix comparison for non-NaNs
Nonworking Combos

- **Nonsensical:**
  - Abort, Throw, Warn
  - Abort/Throw before ProperCompare, WithNaN, Saturate

- **Inefficient/ambiguous:**
  - Warn, then fix comparisons:
    Checked!(Checked!(int, ProperCompare), Warn)
  - Fix comparisons, then warn for all others:
    Checked!(Checked!(int, Warn), ProperCompare)
  - Warn, then fix:
    Checked!(Checked!(Checked!(int, ProperCompare), Saturate), Warn)
Saturate operations, abort on bad casts

```cpp
struct MyHook {
    alias onBadCast = Abort.onBadCast,
    onLowerBound = Saturate.onLowerBound,
    onUpperBound = Saturate.onUpperBound,
    onOverflow = Saturate.onOverflow,
    hookOpEquals = Abort.hookOpEquals,
    hookOpCmp = Abort.hookOpCmp;
}
alias MyInt = Checked!(int, MyHook);
```
Design by Introspection

- Assembly with plastic, adaptable components
- Optional Interfaces
- Automatic/semi-automatic composition
- Exponential coverage with linear code
- Graceful degradation
Destructionize!