The Type Safe Builder Pattern

Using templates to ensure correctly built objects

John Stracke, Ocient
jstracke@ocient.com

The Builder Pattern

- Very basic idea: instead of having a constructor, have a builder object which can be configured incrementally.
- Advantages:
  - Convenient when there are many parameters to specify, and/or the API may be extended to add more parameters.
    - Similar to named parameters.
- Disadvantages:
  - Hard to tell which parameters are mandatory, especially if it depends on what other parameters have been set.
    - This is the problem that type safe builders tackle.

Motivation

// Good (will compile):

```
Plane p = makeBuilder()
              .setPosition(5, 10)
              .addPassenger("Fred")
              .addPassenger("Wilma")
              .setSize(20, 5)
              .build();
```

// Bad (will not compile):

```
Plane p = makeBuilder()
              .setPosition(5, 10)
              .addPassenger("Fred")
              .addPassenger("Wilma")
              .build();
Plane p = makeBuilder()
              .setPosition(5, 10)
              .addPassenger("Fred")
              .addPassenger("Wilma")
              .setSize(20, 5)
              .setPosition(17, 23)
              .build();
```

Validation

- Commonly, builders validate their inputs only at construction time.
  - It is possible to do it earlier; for example, a build for the Plane class could record that position has already been set, and throw an exception if the caller sets it again.
  - But they definitely validate at runtime, not at compile time.
- I will present a way to do validation at compile time.

The wrong way

- The brute force way to tackle the problem is to write a class for each legal combination of {is position set, is size set, are there passengers}. For example, each class representing “yes, position is set” will not have a setPosition() method.
- That’s up to 2^3 classes, with up to 3 methods each, for 24 methods in all.
- But that is obviously not scalable. $O(n * 2^n)$. Also, there’d be lots of duplicate code.

Templates

- The scalable way to do it is to write a template with three Boolean parameters: HasPosition, HasSize, and HasPassenger.
- We use partial specialization so that, for example, setPosition() returns a builder whose HasPosition template argument is true.
- This gets us down to one builder template with three methods and three helper functions, each with one specialization: 6 methods, down from 24. O(n).

```
class Plane {
public:
        const int m_x;
        const int m_y;
        const int m_width;
        const int m_height;
        const std::vector<std::string> m_passengers;
};

Plane p = makeBuilder()
              .setPosition(5, 10)
              .addPassenger("Fred")
              .addPassenger("Wilma")
              .setSize(20, 5)
              .build();
```

```
template<bool HasPosition, bool HasSize, bool HasPassengers>
class builder_t_tmpl {
public:
        Plane build() const {
                return build_t<HasPosition, HasSize, HasPassengers>
                        ::build(*this);
        }
        builder_t_tmpl(int x, int y, int width, int height,
                        std::vector<std::string> passengers)
                : m_x(x), m_y(y), m_width(width), m_height(height), m_passengers(passengers)
        {}

        // …set methods on next slide
};
```

```
template<bool HasPosition, bool HasSize, bool HasPassengers>
class builder_t_tmpl {
public:
        builder_t_tmpl<true, HasSize, HasPassengers>
        setPosition(int x, int y) const {
                return setPosition_t<HasPosition, HasSize, HasPassengers>
                        ::setPosition(*this, x, y);
        }

        builder_t_tmpl<HasPosition, true, HasPassengers>
        setSize(int width, int height) const {
                return setSize_t<HasPosition, HasSize, HasPassengers>
                        ::setSize(*this, width, height);
        }

        builder_t_tmpl<HasPosition, HasSize, true>
        addPassenger(std::string_view passenger) const {
                return addPassenger_t<HasPosition, HasSize, HasPassengers>
                        ::addPassenger(*this, passenger);
        }
};
```

```
template<bool HasPosition, bool HasSize, bool HasPassengers> struct setPosition_t {
        static builder_t_tmpl<true, HasSize, HasPassengers>
        setPosition(
                const builder_t_tmpl<HasPosition, HasSize, HasPassengers>& b,
                int x, int y);
};

template<bool HasPosition, bool HasSize, bool HasPassengers> struct setSize_t {
        static builder_t_tmpl<HasPosition, true, HasPassengers>
        setSize(const builder_t_tmpl<HasPosition, HasSize, HasPassengers>& b,
                 int width, int height);
};

template<bool HasPosition, bool HasSize, bool HasPassengers> struct addPassenger_t {
        static builder_t_tmpl<HasPosition, HasSize, true>
        addPassenger(const builder_t_tmpl<HasPosition, HasSize, HasPassengers>& b,
                     std::string_view passenger) {
                std::vector<std::string> passengers{b.m_passengers};
                passengers.emplace_back(passenger);
                return builder_t_tmpl<HasPosition, HasSize, true>
                        {b.m_x, b.m_y, b.m_width, b.m_height, passengers};
        }
};
```

```
template<bool HasSize, bool HasPassengers>
struct setPosition_t<false, HasSize, HasPassengers> {
        static builder_t_tmpl<true, HasSize, HasPassengers>
        setPosition(
                const builder_t_tmpl<false, HasSize, HasPassengers>& b,
                int x, int y) {
                return builder_t_tmpl<true, HasSize, HasPassengers>{
                        x, y, b.m_width, b.m_height, b.m_passengers
                };
        }
};

template<bool HasPosition, bool HasPassengers>
struct setSize_t<HasPosition, false, HasPassengers> {
        static builder_t_tmpl<HasPosition, true, HasPassengers>
        setSize(
                const builder_t_tmpl<HasPosition, false, HasPassengers>& b,
                int width, int height) {
                return builder_t_tmpl<HasPosition, true, HasPassengers>{
                        b.m_x, b.m_y, width, height, b.m_passengers
                };
        }
};
```

```
template<bool HasPosition, bool SizeSpecified, bool HasPassengers>
struct build_t {
	static Plane build(
		const builder_t_tmpl<HasPosition, SizeSpecified, HasPassengers>& b);
};

template<>
struct build_t<true, true, true> {
	static Plane build(const builder_t_tmpl<true, true, true>& b) {
		return Plane {
			.m_x = b.m_x,
			.m_y = b.m_y,
			.m_width = b.m_width,
			.m_height = b.m_height,
			.m_passengers = b.m_passengers
		};
	}
};

using builder_t = builder_t_tmpl<false, false, false>;

inline builder_t makeBuilder() {
	return builder_t(0,0,0,0,{});
}
```

```
Plane p = makeBuilder()
              .setPosition(5, 10)
              .addPassenger("Fred")
              .addPassenger("Wilma")
              .setSize(20, 5)
              .build();
```