RESHAPING AN OLD PIECE OF DESIGN

Author: Mark Radford.

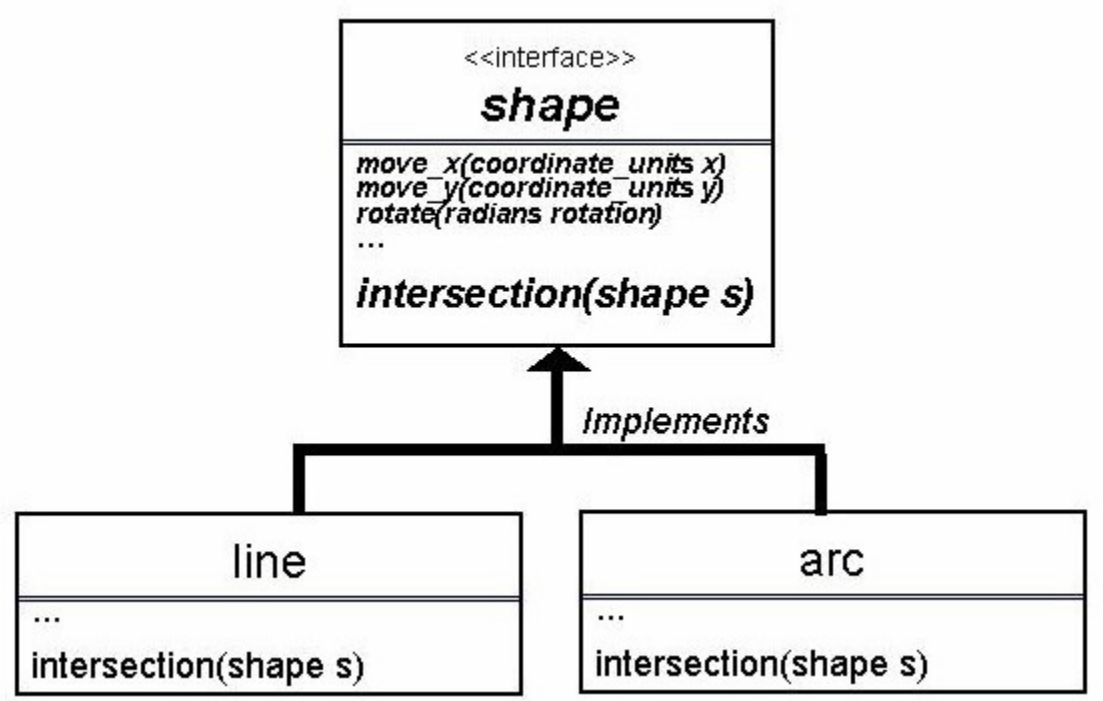
In C++, virtual functions are fundamental in supporting the capability to implement an object-oriented design. They allow a call to a member function made on a pointer/reference to a base class, to result in a member function of the object's concrete class being called. In doing so, they are the language's fundamental mechanism of run time polymorphism – the function actually called depends on the type of the object pointed to, as determined at *run time*.

Sometimes being able to select a function to call based on the run time type of *one* object is not enough. Sometimes there is a need to create the effect of a function being virtual with respect to two or more objects. Some languages (e.g. CLOS) have such a mechanism, and such functions are known as *multi-methods*. However C++ has no such feature, and where multi-methods are required in C++ the effect must be achieved using design and programming techniques.

In this article I will first describe a problem I once faced, that motivated me to take an interest in these techniques. I will describe the solution I chose (which unfortunately was not a good one) and the alternatives I considered, examining the tradeoffs they offer. Then I will go on to look at the solution I would choose if I faced the problem now, and explain why I would prefer it.

The Problem

A few years ago I was involved in the development of a package for producing two-dimensional technical drawings. The drawing program supported two basic shapes: straight lines, and semi-circular arcs, and it is easy to understand how the following hierarchy was fundamental to the design.



It is obvious that these `shape`s would need an interface capable of supporting the operations the user is certain to expect, such as being able to move the shapes around and rotate them. However, because the program was for producing drawings of a technical nature – essentially 2D CAD – an operation to calculate the *intersection* with another shape was also necessary. Unfortunately having available a shape abstraction is not good enough: the `intersection()` methods need to implement the intersection calculation formula, and implementing the formula requires the concrete type of both shapes. In passing, it was to my delight that I found Bjarne Stroustrup cites almost this very problem (in [D&E]) as an example of where multi-methods would be useful.

The solution I came up with at the time was not very good and the irritating thing was that I knew I knew this – I just didn't know what else to do. I could think of other approaches, but they all seemed worse than the one I used. For example, some sources (e.g. [More Effective C++]) use the brute force approach of down casting in conjunction with RTTI. In hindsight though, the RTTI approach offered a better set of tradeoffs.

This problem has been in my mind (on and off) ever since, and years later, I have come up with what I think is a satisfactory approach.

Two Alternative Solutions

I considered two solutions at the time. One of them worked by finding out the run time types of the shapes using run time type information (RTTI); this could be described as a "brute force" approach. The alternative used an object-oriented approach, and I consider it to be a classic example of a solution being flawed while being unquestionably object-oriented.

Solution 1: The RTTI Approach

First consider what a fragment of the code to implement this approach would look like. Here, `dynamic_cast` is used to check for each possible type, and to provide the necessary downward conversion (or down cast).

```
void intersection(const line& l, const shape& s, intersection_points& where)
{
  if (const line* lp =
    dynamic_cast<const line*>(&s))
  {
    lines_intersection(l, *lp, where);
  }
  else if (const arc* ap =
    dynamic_cast<const arc*>(&s))
  {
    line_arc_intersection(l, *ap, where);
  }
  else
    //..
}

void intersection(const arc& a, const shape& s, intersection_points& where)
{ /* .. */ }
```

Now consider the consequences of adding a new specialisation of `shape`, e.g. an elliptical arc. This would mean two things:

(1) Adding a new `intersection()` function overload.

(2) Adding more code to the existing intersection functions.

In passing, note there is a historical twist to my rejection of this solution: neither `dynamic_cast`, nor any other form of RTTI for that matter (remember I said it was a few years ago), were implemented in the compiler used on the 2D CAD project! Therefore, this

approach would have required the manual implementation of some kind of an RTTI substitute (e.g. each class having an integer constant to identify it).

Solution 2: A Flawed Object Oriented Approach

This is the solution I implemented at the time. It employs an object-oriented mechanism of type recovery using virtual functions. The mechanism takes advantage of the fact that an object's concrete type is known within the member functions of the object's class.

Let's look at a C++ fragment showing relevant parts of the shape hierarchy's class definitions:

```
class shape
{
public:

  virtual ~shape();

  virtual void intersection(
      const shape& s, intersection_points& where) const = 0;

  virtual void intersection(
      const arc& s, intersection_points& where) const = 0;

  virtual void intersection(
      const line& s, intersection_points& where) const = 0;
//...
};

class arc : public shape
{
private:
 virtual void intersection(
      const shape& s, intersection_points& where) const;

  virtual void intersection(
      const arc& s, intersection_points& where) const;

  virtual void intersection(
      const line& s, intersection_points& where) const;
// ...
};

class line : public shape
{
private:
 virtual void intersection(
      const shape& s, intersection_points& where) const;

  virtual void intersection(
      const arc& s, intersection_points& where) const;

  virtual void intersection(
      const line& s, intersection_points& where) const;
// ...
};
```

The `shape` class provides the interface class heading up the hierarchy. Note that it has a virtual function overload taking shape as a parameter, as well as one for each of line and arc; if another type of shape (e.g. an elliptical arc) were ever to be added to the hierarchy, shape would need a further virtual function taking the new type as a parameter, and derived classes would need to implement it. Therefore, this design is awkward to extend because it would require changes to code in many of the files participating in the implementation of the shape hierarchy.

The next code fragment shows what happens during an attempt to find the intersection (if any) of objects of type `line` and `arc`:

```
shape* shape1 = new line(..);

shape* shape2 = new arc(..);

shape1->intersection(*shape2, where); // Calls line::intersection()

void line::intersection(
  const shape& s, intersection_points& where) const
{
  s.intersection(*this, where); // Call is re-dispatched...
}

void arc::intersection(
  const line& s,  intersection_points& where) const
{
  line_arc_intersection(
    s,       // ...and handled by the arc::intersection()
    *this,   // overload that handles lines
    where);
}
```

The first call is made on an object of concrete type `line`, so the first virtual function implementation entered is that of the overload `line::intersection(const shape&, ..)`. *Note*: the type of the pointer returned by `this` is `line*` (rather than of type `shape*`).

Next, a call `s.intersection(*this, ..)` is made, and results in a call to the `intersection()` overload taking a `line` as a parameter. Given that the pointer passed in (i.e. `shape2`) points to an object of concrete type `arc`, the result is a call to `arc::intersection(const line&, ..)`. Now the concrete types of both objects is known.

Sadly this solution is flawed because, in a nutshell, it renders derived classes intrusive not only on each other, but also on the base class. It must be remembered that calculating intersection points is only *one* aspect of shape functionality, yet providing it needs three virtual functions in the interface of each class in the hierarchy.

Towards A Better Solution (?)

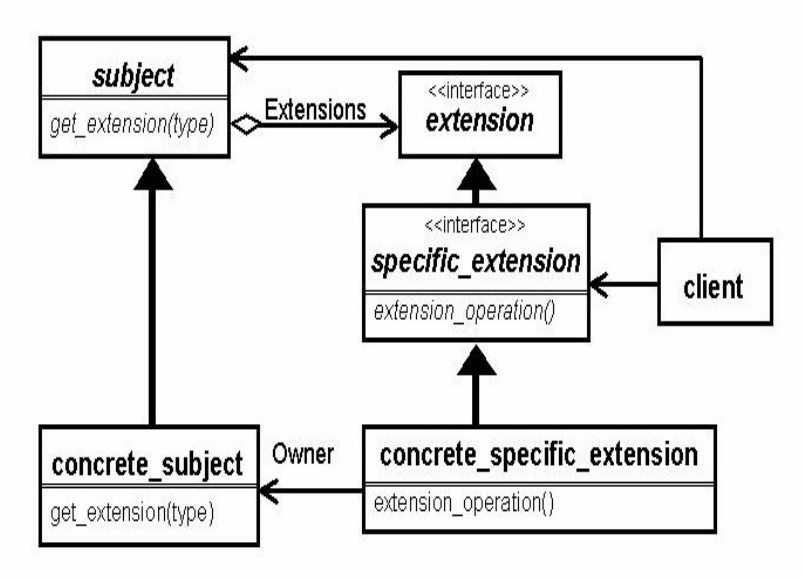
In seeking a better solution, I'm going to start by asserting that the flawed object oriented solution would actually have been quite reasonable but for one thing: classes are *intrusive* on each other. My point is that this intrusiveness would not be such a problem if it could be compartmentalised and therefore its impact limited. To this end I will recruit the help of the EXTENSION OBJECT design pattern (originally documented by Erich Gamma – see [PLoPD3] for the full write-up). What follows is only a brief and slightly C++ centric summary of the pattern, but the description (below) of how it is used to implement a better solution should complete the picture.

Pattern Extension Object.

Context, problem and forces Different clients will have different requirements of an object's interface. The precise interface that will be required by each client cannot always be anticipated at design time. Also, it is often unacceptable to trade provision for them against the interface bloat that would result. In C++ this problem can be addressed to some extent by an approach using freestanding functions. However this does not solve all the (potential) problems (for example, freestanding functions cannot be virtual).

Solution Support the additional interfaces using separate objects and give the Subject an interface for returning *Extension Objects*.

Configuration The extensions hierarchy is headed up by the `extension` interface, while the facilities the extension offers to clients are made available through the interface `specific_extension`.



The `extension` interface does not support the operations required by the client, because different extensions will offer different operations. Therefore `client` obtains access to extensions via `get_extension()`, to which it passes *type*, where *type* is simply some kind of indication of the extension type being requested.

Consequences It can be seen that this pattern offers benefits in terms of flexible extensibility, but there are some drawbacks, for example:

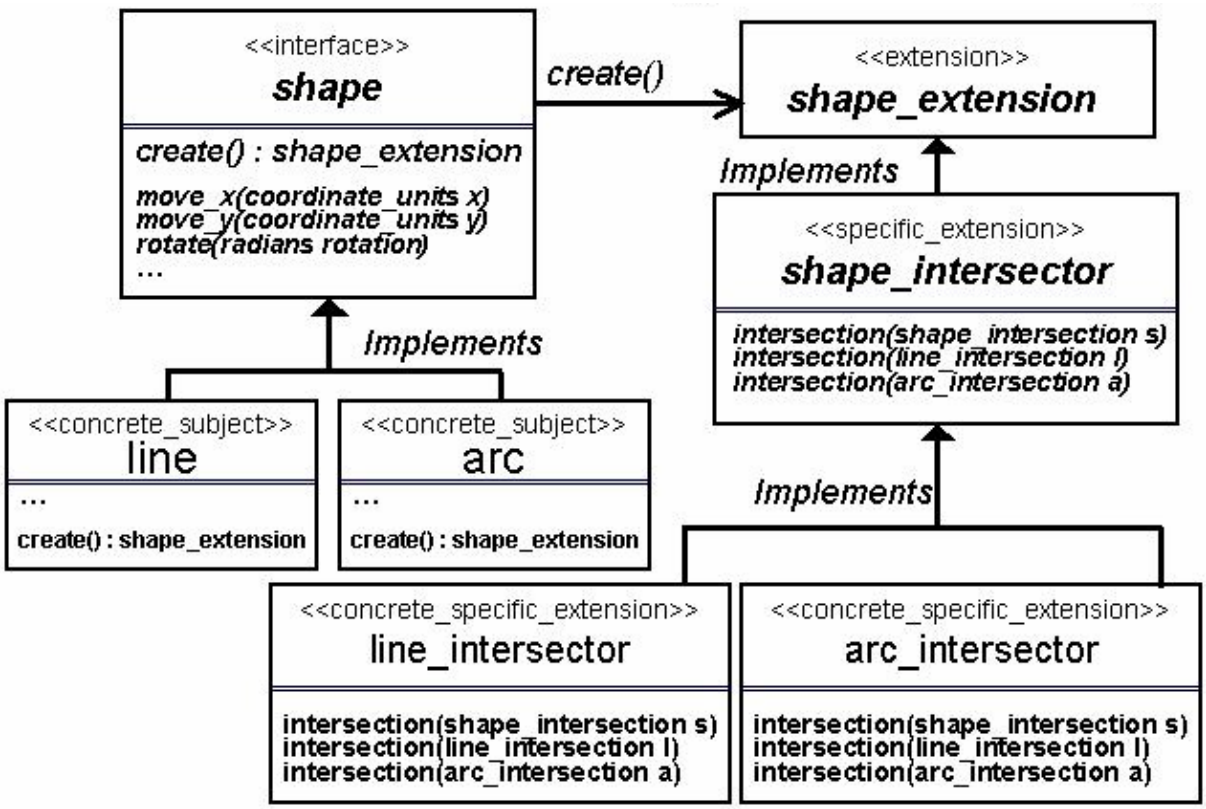
(1) Some of the behaviour of `subject` is moved out of it, so `subject` no longer expresses all the behaviour that clients can perceive it as having (whether this is a good or bad thing depends on the actual behaviour).

(2) The client code will need to recover the `specific_extension` type. A typical method of doing so in

C++ is by using `dynamic_cast`. Therefore, clients become more complex in the face of the "machinery" needed to use the extensions. This machinery can be encapsulated, but the issue still needs to be kept in mind.

Solution Using Extension Objects

The solution presented as a flawed object-oriented solution was in some ways an attractive one, exhibiting the benefits of object-oriented design, keeping code performing a function together and separate from code performing other functions. It was only flawed as a consequence of making classes within the `shape` hierarchy intrusive on each other, and the interface clutter caused (three virtual functions were needed in each class' interface). Introducing EXTENSION OBJECT allows the same mechanisms to be deployed while keeping the intrusiveness and interface clutter out of the `shape` hierarchy. The design now looks like this.



In this design, the following mappings from the Extension Object configuration (above) are used:

- `shape`'s `create()` method takes over from `subject`'s `get_extension()` method. This is because of a C++ object lifecycle issue that will soon become clear.
- `shape_extension` and `shape_intersector` assume the roles of extension and `specific_extension`, respectively.
- `line_intersector` and `arc_intersector` are the `concrete_specific_entensions`.

As an aid to understanding these mappings, the names from the configuration are used as *stereotypes* in the above exposition in UML.

Implementation

The mechanics of recovering the types and working out the intersection points are the same as in the flawed solution – the only difference is that this time the participants are `shape_intersector`, `arc_intersector`, `line_intersector` and the additional `shape_extension`.

The class definition contains very little.

```
class shape_extension
{
public:
  virtual ~shape_extension();

// ...
};
```

It has a virtual destructor, but that needs no explanation.

```
namespace intersections
{
  class shape_intersector
  {
  public:
    virtual ~shape_intersector();

    virtual void intersection(
      const shape_intersector& obj, intersection_points& where)
    const = 0;

    virtual void intersection(
      const line_intersector& obj, intersection_points& where)
    const = 0;

    virtual void intersection(
      const arc_intersector& obj, intersection_points& where)
    const = 0;

  // ...
  };

  boost::shared_ptr<shape_intersector>
    down_cast(boost::shared_ptr<shape_extension> obj);
}
```

The `shape_intersector` class is the first one in the hierarchy to have an interface of any substance. It declares `intersection()` member function overloads in much the same way as `shape` did in the flawed object oriented solution – the difference here being that these overloads take `line_intersector` and `arc_intersector` parameters, in place of `line` and `arc` parameters respectively.

Another declaration of interest is that of the `down_cast()` function: not a member of `shape_intersector` but provided within the `intersections` namespace. To understand its role, first we need to look at `shape`.

```
class shape
{
public:

  virtual void move_x(coordinate_units x) = 0;
  virtual void move_y(coordinate_units y) = 0;
  virtual void rotate(radians rotation) = 0;
// ...

  virtual boost::shared_ptr<shape_extension>
    create(const std::type_info& type) const = 0;
};
```

The `shape` interface class provides (besides the functional interface supporting user operations) a virtual member factory function `create()` that returns a `shape_extension` instance. Here there is a deviation from the canonical EXTENSION OBJECT configuration, because `concrete_subject` (`line` or `arc`, omitted from the UML diagram) is designated as its owner, which is not quite the case here. The design in this example uses the C++ idiom of using a smart pointer to manage memory acquisition and release, to avoid running into problems with object lifetimes.

Returning to `down_cast()`: in order to use the `shape_intersector` interface, the `shared_ptr<shape_extension>` instance returned from `shape::create()` must be converted to type `shared_ptr<shape_intersector>` (remember this was listed as a consequence of the EXTENSION OBJECT design pattern). A custom mechanism in the form of `down_cast()` is provided to achieve this, because unfortunately the use of a smart pointer cuts across the natural approach of using `dynamic_cast`.

The definitions of classes `line` and `arc` are self-explanatory: they just provide implementations of `shape`'s virtual member functions `move_x()`, `move_y`, `rotate()` etc. I'm not going to list them here because I don't believe they will actually add anything to the illustration. Instead I'm going to move on to `intersection()`, another freestanding function declared within the `intersections` namespace.

```
namespace intersections
{
  intersection_points intersection(const shape& s1, const shape& s2) // 1
  {
    intersection_points where;

    boost::shared_ptr<shape_intersector> first =
      down_cast(s1.create(typeid(shape_intersector)));  // 2

    boost::shared_ptr<shape_intersector> second =
      down_cast(s2.create(typeid(shape_intersector)));  // 3

    first->intersection(*second, where); // 4

    return where;
  }
}
```

Before looking at `intersection()`'s implementation, I feel it is worth digressing briefly to look at a trade-off that has been made. It was observed that as a *consequence* of the EXTENSION OBJECT design pattern, the machinery for obtaining `extension` (`shape_extension`) instances and down casting them to `specific_extension` (`shape_intersector`) adds complexity to clients. It was also observed that one way to address this complexity is to *encapsulate* it, and this is the approach taken here: i.e. it's all wrapped up in the `intersection()` function. This encapsulation introduces a tension with the design decision to create `shape_extension` instances on the heap (instead of the originating object owning them): there is no way to preserve these instances between calls to

`intersection()`. Thus efficiency is traded for simplicity of usage (and tidiness of exposition in an article ☺).

Getting back to `intersection()`'s implementation...

The function takes two shape instances (by reference so they exhibit run time polymorphism), `s1` and `s2`, as its parameters (statement 1). Statements 2 and 3 create `first` and `second`, these being the `shape_intersector` instances, and here two things should be observed:

- The `shape::create()` function is called within the call to `down_cast()` so the instances, although present, never appear explicitly as type `shape_extension`.
- In the calls to `shape::create()`, the arguments are in both cases `typeid(shape_intersector)`, i.e. *not* the `typeid` of the most derived classes. This is because the concrete classes `line` and `arc` know they must create `line_intersector` and `arc_intersector` respectively – they only need to be told they are creating `extension`s to a type `shape_intersector`, as opposed to any other type of `extension`.

Statement 4 is where the intersections (if any) are calculated. The rest of how this works is very similar to the way in which the flawed object-oriented solution worked.

```
shape* shape1 = new line();
shape* shape2 = new arc();
```

And their intersections calculated…

```
intersection_points where = intersection(*shape1, *shape2);
```

The workings of the `intersection()` function were explained above, so we now need to look at how `line_intersector::intersection()` and `arc_intersector::intersection()` work. When `intersection()` is called with `shape1` and `shape2` as arguments, statement 4 in its implementation results in a call to the `line_intersector::intersection()` overload taking a `shape_intersector` parameter.

```
void line_intersector::intersection(
  const shape_intersector& s, intersection_points& where) const
{
  s.intersection(*this, where); // Call is re-dispatched...
}
```

Remember `shape2` has concrete type `arc`, so re-dispatching the call results in a call to `arc_intersector::intersection()` – specifically, the overload that takes a `line_intersector` as a parameter.

```
void arc_intersector::intersection(
  const line_intersector& s, intersection_points& where) const
{
  line_arc_intersection(
    s,
    *this,
    where);
}
```

That's it. At this point the concrete types of both `shape_intersector`s are known, and the calculation (details of which we are not concerned with here) can be performed.

Phew!

Tradeoffs – In Favour

Intersection logic is non-intrusive with respect to the `shape` hierarchy. In the case of the flawed object oriented solution, the problem was that derived classes were intrusive on the

base class, and on each other. In the case of the solution that uses *Extension Objects*, classes derived from `shape_intersector` are also intrusive on each other, but there is a very important difference: there is no intrusiveness on the `shape` hierarchy. For example: if another shape is added, only the classes in the multi-methods hierarchy are affected.

Note that, in the case of the example of adding another type of shape (an elliptical arc for example), the bodies of existing `shape_intersector` member functions will not need their implementations changing. This is a consequence of virtual functions being used to automate the control flow by placing it in the hands of the C++ language. By contrast, in the case of the RTTI solution, the control flow is implemented directly in the code, and as a consequence adding the code for a new type of shape means modifying existing code. In the former case, the absence of a need to change existing code means that the chance of introducing an error into it is reduced.

Tradeoffs – Against

The `shape` and `shape_intersector` hierarchies have parallel corresponding classes. Working with and maintaining such parallel hierarchies always creates a balancing act of design.

The most obvious burden is the extra types that now inhabit the design, and these must be managed – not just in physical terms but also in addressing the communication issues that arise (more documentation will be needed).

More subtle is the lack of any direct mention of intersections in the `shape` interface, and in the interfaces of classes derived from it. Here, a consequence associated with applying the EXTENSION OBJECT design pattern haunts the design.

In Conclusion

Using the object-oriented paradigm does not automatically make a design superior. In the past object orientation has been adopted in the hope that it would be the silver bullet that would solve all software development problems. Of course, history now records that nothing was further from the truth. There were many factors involved, one being the lack of understanding of object orientation itself. Another critical factor however, was the assumption that being object oriented automatically made a design a good one. The flawed object oriented solution presented earlier is an excellent counter example.

An important lesson is that even good OOD has its costs. It comes back to the fact that when solving problems with any level of complexity, there is no such thing as a solution per-se – there are options and tradeoffs.

Finally, the BSI C++ panel are currently discussing a proposal by Julian Smith to add multi-methods to the language – therefore this feature may or may not be present in the language when the next edition of the standard appears. Full details of the proposal can be found at Julian's web site (see [Multi-Methods Proposal]).

References

[Boost] The Boost library (see www.boost.org)

[D&E], Bjarne Stroustrup, The Design and Evolution of C++, Addison-Wesley, 1994

[More Effective C++], Scott Meyers, More Effective C++: 35 New Ways to Improve Your Programs and Designs, Addison-Wesley, 1996.

[Multi-Methods Proposal] Julian Smith's proposal for adding multi-methods to C++ (www.op59.net/cmm/readme.html)

[PLoPD3], Robert Martin, Dirk Riehle and Frank Buschmann (Editors), "Pattern Languages of Program Design 3", Addison-Wesley, 1998.