Context Encapsulation

Three Stories, a Language, and Some Sequences

Kevlin Henney
kevlin@curbralan.com
kevlin@acm.org

June 2005

Abstract

The skill of writing is to create a context in which other people can think.

Edwin Schlossberg

This paper presents a small pattern language, CONTEXT ENCAPSULATION, based on dividing up the ENCAPSULATE CONTEXT pattern by Allan Kelly. Four patterns for encapsulating execution context in statically typed languages are described, allowing a component to be sufficiently decoupled from the code and assumptions in the environment in which it was written. It connects these patterns into the kernel of a potentially larger pattern language. It presents pattern stories that illustrate the language by example and offers pattern sequences that highlight more generally how the pattern language can be animated.

The historical background and motivation for the themes in the paper are also introduced. This paper concludes with a section of notes that readers should consider appendices, and should be read only if the subject matter appeals: some reflections on the wider pattern relationships the pattern language can draw on; and a brief and formal reflection on pattern sequences, pattern languages, and the nature of design.

Some Context

The devil is in the detail.

You have this system...

- *You try globals... well, probably you don't: the one thing you learned in school was* no globals.
- *You try for* SINGLETON*, it is in the book, it is good... but then you find you have these nasty ripples... then someone tells you it's a bad thing and it's obvious to you.*
- *So you try passing parameters: they overwhelm you.*
- *You refactor a bit (à la Fowler) and before you know it you've got* ENCAPSULATE CONTEXT*.*
- *You carry on down this path, you get more mileage here, but over time it starts to look like Foote's* BIG BALL OF MUD*.*

The solution is to reduce the coupling, improve the cohesion, but how?

Allan Kelly[1]

It all started a couple of years ago — the summer of 2002, to be more precise. Following a discussion thread on the ACCU's main list, Allan Kelly decided to document a pattern for addressing the problem of propagating context through a program that neither relied on the coupling of global variables and their kin nor on the unmanageability of long argument lists. I volunteered to shepherd the paper informally; Allan then submitted it to EuroPLoP 2003; it received further shepherding from Frank Buschmann; it was accepted and workshopped at the conference.

There was a recognition that the scope of the pattern was perhaps greater than could be contained conveniently within a single pattern. Most of the subtlety was in realizing the pattern effectively, and the options available and decisions that needed to be taken formed a long tail in the presentation of the pattern. Following its inclusion in the EuroPLoP proceedings [Kelly2003], the paper was also published in the ACCU's *Overload* magazine [Kelly2004], where it generated some heated discussion on the editorial review team [Overload2004] and the letters page [Overload2005]. Although ENCAPSULATE CONTEXT's description contained careful discussion of how to avoid having a context object turn into an uncohesive blob of code, this discussion sometimes appeared to be overlooked.

Recasting the pattern in terms of a pattern language rather than a single pattern allows the flow of design issues to be identified more explicitly, promoting each of the considerations and decisions as a response to the issues raised in another pattern. Instead of considering the coupling and cohesion issues as simply being implementation details within a single pattern, the design process is made more explicit by naming and connecting some of these design decisions. Pulling supporting patterns out of a larger root pattern helps to manage pattern scope, which can otherwise creep with each new consideration that is incorporated.

A common technique in pattern writing is to employ examples to motivate and illustrate patterns and pattern languages. In the case of pattern languages such pattern stories help to bind the separate patterns in the language together in the minds of both the reader and the author. Although there is a tendency to use single examples to illustrate lone patterns or whole languages, it is often just as important to use more than story to show explicitly

[1] In private correspondence, February 2005.

the diversity and range of a pattern or language. A language may admit many paths through it; stories are one way of directing readers along these paths. However, stories are walks along the paths rather than the paths themselves. The idea of pattern sequences has gained some attention in recent times, but beyond a more conscious use of pattern stories there has been relatively little documentation of pattern sequences in their own right.

All of which brings us to this paper.

A Guide for (and to) the Reader

The patterns in this paper began life in an email with no specific goal in mind, part of a discussion about the original ENCAPSULATE CONTEXT pattern. The writing of the paper also coincided with a continued interest in pattern presentation and concepts. However, this personal perspective may not match the interests and inclinations of the reader, so it is worth presenting the content and goals of this paper more explicitly so that the reader can pick and choose or peruse and contemplate, as appropriate:

- *For readers most interested in the problem being solved*: A tempting and popular solution for communicating pervasive context information to different parts of a program is to introduce some centralized and global dependency, whether obviously as a global variable or more surreptitiously as a SINGLETON object. This approach is not without its problems. This paper addresses the issues with a solution family presented in pattern form that is more loosely coupled than the global option. In addressing a common tension in layered architectures, the solution family emphasizes locality over globalness. Examples, in the form of pattern stories, are first used to motivate the patterns and illustrate how they may be combined. The patterns are presented as a connected form, a pattern language, and common usage sequences are also highlighted. Material beyond this basic scope can be considered supplementary.
- *For readers familiar with commonly publicized design patterns*: It perhaps goes without saying that SINGLETON is one of the most popular OO design patterns. And yet it troubles almost every system it appears in, whether its programmers recognize this or not. This paper offers a perspective on some of the problems and related patterns of this globalized approach; it offers a reasoned alternative based on existing, proven design practice. The concepts of pattern stories, pattern sequences, and pattern languages are introduced and may prove to be of interest, but the additional notes to the back end of the paper can be considered supplementary.
- *For readers with an interest in pattern concepts*: There is a strong emphasis in this paper in going not only beyond the use of individual patterns, but also in extending the typical application of pattern languages. The role of stories is prominent in illustrating how a language plays out and how it can be introduced. Stories can be further distilled into their essential pattern sequences. A number of pattern languages are published with stories against them, but few are published with a distillation of the sequences in their own right. The additional notes at the back end of the paper may prove to be of interest, but should be considered optional.
- *For readers with an (unhealthily) in-depth interest in pattern theory*: In addition to presenting the concepts of pattern stories, pattern sequences, and pattern languages from the ground up, this paper offers some further thoughts on presenting and thinking about pattern languages and sequences. Formally, the view is often taken that pattern languages are directed graphs with patterns at the nodes. This view is not without truth, but it does not include notions such as grammar, which can be seen as relevant to any form of language. A process-based view, drawn from process algebra, offers an alternative perspective that includes the concept of sequencing more explicitly. The ideas need further elaboration and discussion before they can be considered mature, and are presented here for precisely this reason.

Three Stories

After writing a story I was always empty and both sad and happy, as though I had made love, and I was sure this was a very good story although I would not know truly how good until I read it over the next day.

Ernest Hemingway

The step from individual patterns to pattern languages can be a large one, especially for a pattern author. But it is a step that is all too often encouraged without caution. As with any other form of design knowledge, the ability to combine patterns in a clear, sensible, and robust manner is by the accumulation of fragments, not the proposition or imposition of some grand meta-scheme.

Part of the appeal of many patterns and much pattern-related writing is the emphasis on illustration through practical examples. There is a lot about patterns that can be abstract; examples offer concretion. The same thinking applies to dealing with multiple patterns in concert: an example that brings them together will tend to be more credible and more useful than simple assurances or bald assertions that the patterns will (or should) work together.

On Pattern Stories

It makes sense to take smaller, surer steps towards a goal or in exploring a design than to generalize hastily and repent, rewrite, and rescope at leisure. This is the role that pattern stories can play for both pattern reader and pattern author. Pattern stories are illustrative, realistic, reflective, and pragmatic, as well as more modest in their aims than the patterns and languages they illustrate [Henney1999]:

> Where pattern languages are narrative frameworks, the development of a system can be considered a single narrative example, where design questions are asked and answered, structures assembled for specific reasons, and so on. We can view the progress of some designs as the progressive application of particular patterns.
>
> What we can term *pattern stories* are narrative structures, i.e. they exist as complete and in the real world. As with many stories, they capture the spirit and not necessarily the truth of the detail in what happens. It is rare that our design thinking will so readily submit itself to named classification and sequential arrangement, so there is a certain amount of retrospection and revisionism. However, as an educational tool, such an approach can reveal more than conventional case studies, as each example literally tells the story of a design. This is a valuable aid in studying design through the study of designs.

Thus, pattern stories can do for a pattern language what simpler examples do for individual patterns. Such stories can be woven into the presentation of patterns as running examples or they can be presented separately, ahead of the language or following it. Stories may be of systems already built, forecasts of systems to be built, or simply hypothetical illustrations of how systems could be built.

Although lone examples are useful and motivating, a common pitfall readers trip over is the assumption that there is little more depth or breadth to the pattern or pattern language than is shown in the example. Of course, this depends very much on the reader, the

examples, and the patterns in question, but using multiple examples makes the pitfall less likely and the pattern presentation correspondingly richer.

As an aside, there is also a pleasing resonance to a more everyday use of the term *pattern stories*, which are stories that include repeated and emphasized phrase structures and associated actions to involve listeners, particularly children.

Storytelling Style

It is often said that patterns help to establish a design vocabulary, with a pattern language offering a grammar. This metaphorical relationship continues with the notion of pattern stories: a tale told using the words according to the grammar. The storytelling device has been used across many pattern works, regardless of whether or not the notion of a narrative has been acknowledged explicitly. And, of course, there is as much variation in style as there is with real storytelling.

There is a brief example in *A Pattern Language* [Alexander+1977] that is more a piece of reportage that details what design unfolded rather than a story that describes why. By contrast, *The Oregon Experiment* [Alexander1975] is more a novel than a short story or review. In *Design Patterns* [Gamma+1995] the story is the document editor case study. In other papers and books it is an unfolding concrete example that illustrates the language in the way that an individual example might illustrate a lone pattern. *Organizational Patterns* [Coplien+2005] includes four stories that follow the reporting style found in *A Pattern Language*.

So, in some cases stories are presented more like news stories or story synopses; in others there is a stronger narrative backbone based on dramatic tension and resolution created by the key players in the design situation.

Dramatis Personae

The following three tales include a common cast of main characters, which are the focus of the kernel pattern language in the next section, plus some supporting characters, which would be included in any extended version of the pattern language. The patterns are named and introduced briefly here. Further pattern connections are discussed toward the end of this paper.

Protagonist Patterns

ENCAPSULATED CONTEXT OBJECT: Pass execution context for a component, whether it is a layer or an individual object, as an object rather than as a long argument list of individual configuration parameters or implicitly as a global service. The execution context may include external configuration information and services such as logging.

DECOUPLED CONTEXT INTERFACE: Reduce the coupling of a component to the concrete type of the ENCAPSULATED CONTEXT OBJECT by defining its dependency in terms of an interface, whether `interface` or INTERFACE CLASS, rather than the underlying implementation type. This allows substitution of alternative implementations, including NULL OBJECTS and MOCK OBJECTS.

ROLE-PARTITIONED CONTEXT: Split uncohesive ENCAPSULATED CONTEXT OBJECT interfaces into smaller more cohesive context interfaces based on usage role, each expressed with a DECOUPLED CONTEXT INTERFACE or through a ROLE-SPECIFIC CONTEXT OBJECT.

ROLE-SPECIFIC CONTEXT OBJECT: Multiple context interfaces may be realized either in a single object at runtime or with an object per role. The latter option allows independent parts of a context to be more loosely coupled and separately parameterized.

Support Patterns

INTERFACE CLASS: In statically typed languages that support object orientation but do not have a built-in language mechanism for expressing pure object interfaces, e.g. `interface` in Java and C#, an interface can be expressed as a fully abstract class. In C++ an INTERFACE CLASS can be defined as a class that has only `public` pure `virtual` ordinary member functions, no `private` section, and whose destructor is either `public` and `virtual` or `protected` and optionally non-`virtual`.

LAYERS: A system can be separated into vertically stacked LAYERS [Buschmann+1996] that contain its implementation and interfaces. Each layer is a subsystem that depends only on LAYERS below. There are many criteria for separation, including abstraction, technology, rate of change, organizational structure, deployment target, etc. Layering may be local or global, i.e. within a package or across a whole system, and strict or not, i.e. dependencies are strictly between adjacent layers or not, but never cyclic.

MOCK OBJECT: To be practical, unit testing often requires the ability to isolation test certain components, which means that external dependencies such as databases, external files, devices, etc, must be decoupled — testing with them is integration testing rather than unit testing. A MOCK OBJECT [Beck2003, Mackinnon+2000] fulfills the role of the external dependency, but is fully internal and under the control of a given test, where it is either primed (if it is a source) or inspected (if it is a sink), or both.

NULL OBJECT: The art and implementation of doing nothing. A NULL OBJECT [Henney2002] supports a given object interface with no-op methods; for non-`void` methods, a suitable zero or null value is returned. NULL OBJECT simplifies use of references that might otherwise be `null` by ensuring they are non-`null` and can be programmed against.

PLUG-IN: In a PLUG-IN [Fowler2003, Marquardt1999] architecture subsystems are combined via a set of published interfaces, making the architecture orthogonal, loosely coupled, and highly configurable. The layer of external interfaces effectively represent an inversion layer: although the architecture is essentially layered, code dependencies are isolated so that the concepts of *higher* layer versus *lower* layer become a little blurred.

Antagonist Patterns

BIG BALL OF MUD: A commonly recurring architecture [Foote+2000]: "A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. We've all seen them. These systems show unmistakable signs of unregulated growth and repeated *expedient* repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated."

MONOSTATE: This pattern can be considered a salve for programmers who dislike the guilt by association of employing SINGLETON. It also plays the role of syntax sugaring for those want a less cumbersome usage syntax than SINGLETON. A MONOSTATE [Ball+1997] object looks like an ordinary object but shares its state `static`-ally with all other instances of the class, leading to "spooky action at a distance" and aliasing problems when the state changes. If SINGLETON is the problem, MONOSTATE as a treatment can be worse than the problem, although some developers mistake it for a cure. MONOSTATE is also known affectionately and revealingly as the BORG pattern.

SINGLETON: A creational pattern whose brief is given as follows [Gamma+1995]: "Ensure a class has only one instance, and provide a global point of access to it." However, it invariably causes problems with coupling and execution, and is more than a little overused, so a less favorable view is becoming common [Beck2003]: "How do you provide global variables in languages without global variables? Don't. Your programs will thank you for taking the time to think about design instead."

Pattern Story: A Tale of Pluggable Implementations in C#

Consider a PLUG-IN interface that can be used to extend a C# application or component with additional parts. The PLUG-IN interface is defined as a pure `interface`. The interface should also be independent of the application, which is the PLUG-IN user, and other PLUG-INS. Thus, the PLUG-IN user relies only on what is defined in the PLUG-IN interface and its dependencies, and likewise the PLUG-IN implementation can rely on the interface and its dependencies, but neither the implementation nor the user can depend on one another.

Once published, the PLUG-IN interface is expected and required to remain stable in terms of methods and method signatures. However, any other dependencies that appear in the PLUG-IN interface may evolve and are required to have at least source-code compatibility — preferably binary compatibility — for the PLUG-IN implementation.

How can the PLUG-IN implementation gain access to certain execution context information and various services that would be offered by the PLUG-IN user? There is no requirement or guarantee that the information or services correspond directly to the equivalents in the operating system, e.g. the user name for the application need not be the same as the user name for the current operating system session, and no certainty that there is only a single way of defining a service that will remain stable over time, e.g. how logging is realized is a matter for the provider of the PLUG-IN framework and not an assumption that a PLUG-IN can make. It is also possible that new context information and services will be added over time.

A naïve implementation would have PLUG-INS depend directly on some root application object, perhaps expressed as a SINGLETON that could be accessed globally throughout the whole executable:

```
public class Application
{
    public static Application Instance
    {
        get {...}
    }
    public string UserName
    {
        get {...}
    }
    public void WriteLog(string message) {...}
    ...
}
```

However, all that is required is certain services and information, not the whole application object interface. Furthermore, the requirement is that the PLUG-INS can be freely used by other components or systems that rely on nothing more than the PLUG-IN interface and its immediate dependencies: there may be no meaningful root application object or, if there is, it may derive from another framework.

On the other hand, while it makes sense to group the features of the execution context together, simply detaching it from a root application object and have it expressed as a global variable is not much of a solution. Of course, classic C-like globals cannot be expressed directly in C#, which requires all data to be bound to a scope. However, non-constant `public static` fields are to most intents and purposes the equivalent of global variables — `static` data is little more than global data with scope etiquette:

```
public class PlugInContext
{
    public static string UserName;
    ...
}
```

A common way to ease programmer conscience over globals is to employ SINGLETON objects:

```
public class PlugInContext
{
    public static PlugInContext Instance
    {
        get {...}
    }
    public string UserName
    {
        get {...}
    }
    public void WriteLog(string message) {...}
    ...
}
```

Thus, instead of writing `PlugInContext.UserName` inside a PLUG-IN implementation, the programmer writes `PlugInContext.Instance.UserName`. It is not altogether obvious that this qualifies as a big step forward.

SINGLETON's popularity is perhaps inversely proportional to its applicability. Although superficially simple, focusing on the design of one class, it is riddled with subtlety. SINGLETON inevitably invites the programmer to work around it — for example, ensuring that its initialization is thread safe requires skill, gymnastics, and a certain leap of faith. SINGLETON represents a source of high coupling and complications whose common purpose, it seems, is to (1) support a denial that globals are not being used and (2) save programmers from having to pass an object explicitly as an argument. This latter approach has much in common with traditional FORTRAN techniques for passing state around covertly — and sometimes mysteriously — through `COMMON` blocks rather than being explicit about the parameters that actually parameterize a subroutine. The sense of mystery can be heightened by using MONOSTATE objects instead of a SINGLETON.

Another problem with globalesque solutions is a matter of change: information is provided by the PLUG-IN user for PLUG-INs to read but not to modify. This means there is a need for some kind of additional initialization method or a family of setting methods or properties:

```
public class PlugInContext
{
    ...
    public string UserName
    {
        get {...}
        set {...}
    }
    ...
}
```

A `static` context object's constructor cannot be used for this purpose because it will already have executed before the point in the PLUG-IN user where the values can be set. However, adding extra methods for setting state also allows PLUG-INs to modify the state, which they are not supposed to do. Restriction of visibility of extra methods to assembly `internal` or `namespace` private does not resolve the problem because there is no requirement that a PLUG-IN user must be implemented in the same assembly or namespace as the PLUG-IN support code — indeed, the desire for non-invasiveness ensures that, if anything, there is a requirement against placing them together. One could also consider providing a separate interface or controller object that accommodated change. However, this interface separation or related object would have the same visibility to the PLUG-IN implementation as it would to the PLUG-IN user.

Therefore, pass an ENCAPSULATED CONTEXT OBJECT from the PLUG-IN user to the PLUG-IN implementation explicitly. The ENCAPSULATED CONTEXT OBJECT is passed explicitly as an argument to the PLUG-IN as part of its startup, and so a method to do this must appear in the PLUG-IN interface. The ENCAPSULATED CONTEXT OBJECT offers access to all the external information and services that a PLUG-IN can depend on with respect to its execution context.

The resulting code is non-intrusive on the PLUG-IN user:

```
public sealed class PlugInContext
{
    public PlugInContext(string userName, ...) {...}
    public string UserName
    {
        get {...}
    }
    public void WriteLog(string message) {...}
    ...
}
```

And offers a stable and minimal area of contact between PLUG-IN and user for communicating execution context:

```
public interface PlugIn
{
    void Initialize(PlugInContext context);
    ...
}
```

The PLUG-IN user is responsible for correctly initializing the ENCAPSULATED CONTEXT OBJECT passed to its dependent PLUG-IN implementations. A suitable constructor is therefore needed for the ENCAPSULATED CONTEXT OBJECT type, and this is the only point in the lifecycle that state can be set in an ENCAPSULATED CONTEXT OBJECT, a point in the lifecycle that is not accessible to the PLUG-IN implementation.

In principle there is no reason that the same ENCAPSULATED CONTEXT OBJECT need be used for all PLUG-INS. It may be appropriate to provide different execution contexts to different PLUG-INS; explicitly passing an object ensures that different uses and objects do not interfere with one another. Over time it is possible to add new information or services to the ENCAPSULATED CONTEXT OBJECT type and still retain signature compatibility. However, retiring features is not as simple.

The PLUG-IN can use or retain the passed ENCAPSULATED CONTEXT OBJECT as it sees fit. Indeed, there is no requirement that a reference-based object is used at all: a `struct` would serve just as well, still retaining source-level compatibility. Likewise, in this case an `interface` could be introduced in place of a `class` and still retain source-level compatibility in the PLUG-IN implementation. As an aside, this substitutability under refactoring highlights why the common use of the `I` prefix for interfaces in C# is not a wise naming convention. (Besides, it is difficult to resist feeling that `IPlugIn` ought to be complemented by `UPlugInUser`.)

There are certainly further refinements that could be made to loosen the coupling, but even a plain ENCAPSULATED CONTEXT OBJECT represents a significant improvement over the alternatives considered. The resulting architecture is based on the PLUG-IN user pushing the context onto the implementation rather than having it pulled; the pull of information or use of service is localized to an object rather than a whole application or component.

Pattern Story: A Tale of Alternative Implementations in C++

Consider a C++ application structured in conventional LAYERS. Different pieces of code in different LAYERS tend to need access to environmental values, such as the host name, the user name, the application command line, application-related configuration values that can be looked up by name, and so on. These will be handled differently depending on deployment options: under Microsoft Windows the centralized registry may hold the configuration values; conventional environment variables, accessed via the C standard `getenv` function, may be used, and would be a common solution under UNIX; a text file with a simple format would be another alternative, as would an XML file; and so on. Likewise, the means for accessing the other environmental values would vary with platform. It would be repetitive and clumsy (but not unknown) to litter the code that needed these values with an impenetrable patchwork of conditionally compiled code [Spencer+1992].

How can functions and objects in different LAYERS look up environmental values without their code becoming coupled to the underlying implementation? The values should not be modifiable and they may be held locally or looked up externally to the application. Not all functions and objects need access to environmental values — indeed such access should be localized and the dependency visible.

Such environmental values do not lend themselves to being expressed as individual or collection-based global variables. Unless declared `const`, such variables would be not only readable but also assignable from anywhere in the application. However, if declared `const`, there is no guarantee that it would be possible to initialize them to the right value: the point of variable initialization would be before the point in the application's lifecycle where they could be set.

This suggests that a function-based approach is more effective than one based on global variables. Because the implementation of how values are looked up varies consistently and together, it makes sense to think of such functions as being grouped together in a consolidated API rather than fragmented as lone functions across disparate header files. This also ties in with usage: programmers writing code dependent on environmental lookup would rather see the features grouped together as a cohesive whole than scattered around.

The notion of an API and the desire to make a dependency on environmental lookup visible suggests that the functions should be wrapped in a `namespace`:

```
namespace environment
{
    std::string host();
    std::string user();
    std::string command_line();
    std::vector<std::string> arguments();
    bool has_value(const std::string &name);
    std::string value_of(const std::string &name);
    ...
}
```

Dependency on the API can be seen both from header file inclusion and from the use of a `using` directive, `using` declaration, or, most clearly, a fully qualified name, e.g. `environment::host`. However, although this makes dependency clear in implementation files, it is not visible in header files. Classes may appear at first not to have a dependency on `environment` simply because it is not mentioned in the header files defining the classes and declaring their members.

There is also the question of setting state. Depending on the platform and implementation choices, some of these values may be looked up repeatedly and dynamically, whereas others may need to be set explicitly at application startup. This raises two issues: first, how to offer one or more initializing functions that can only be called during application startup

and not during the application's main lifecycle and not by the users of the `environment` API; second, how to accommodate the interface variability of initialization for different implementations.

For an API based on global functions (euphemistically called non-member functions by many C++ programmers) there is no simple way to define an explicit point of initialization for the application to use that cannot also be called by other parts of the application. Correct use rests on an understanding of the API contract and honoring the gentleman's agreement that goes with it. The variability of the initialization part of the API can be handled either by defining in function form the set of all possible setting options, disabling the ones that are not relevant for a particular platform at runtime, or by conditionally including different functions depending on the platform. Neither is particularly satisfactory: the former option presents a consistent function interface by appearance but not by action; the latter manicures the API at compile time depending on build options, which means that the API is not stable with respect to its apparent set of features.

Therefore, create an ENCAPSULATED CONTEXT OBJECT *that is passed down from the root of the application to the functions and objects in different* LAYERS *that depend on environmental lookup. The* ENCAPSULATED CONTEXT OBJECT *offers access to environmental values through query functions.*

The resulting API is now cohesive at the level of the object:

```
class environment
{
public:
    std::string host() const;
    std::string user() const;
    std::string command_line() const;
    std::vector<std::string> arguments() const;
    bool has_value(const std::string &name) const;
    std::string operator[](const std::string &name) const;
    ...
};
```

Any setting of values is handled at construction, so it is not possible to later modify these through the `const` function interface. Dependency on environmental values is now made explicit: to have access to the application environmental values, an `environment` object must be passed in as an argument, raising the dependency to the signature level.

Making the dependency a visible object relationship rather than a covert implementation relationship may suggest that `environment` will suddenly grace the signatures of nearly all functions in all LAYERS. However, not all functions and objects need access to environmental lookup, only a handful actually need this kind of access. If it appears that a handful becomes a bucket full, this is an indication that the responsibilities of each component are unclear — excess coupling and a lack of cohesion have been revealed. Thus, it is not the introduction of an ENCAPSULATED CONTEXT OBJECT that is at fault; it is the introduction of the ENCAPSULATED CONTEXT OBJECT that has highlighted the fault.

The flow of control and the flow of information are now similar: although environmental values are often considered to be a facility inhabiting the lower levels in a LAYERS architecture, matters of application combination, configuration, and initialization are the responsibility of the hub or root of the application — most simply, `main`.

How can implementation variation of ENCAPSULATED CONTEXT OBJECTs *be accommodated most simply and uniformly? Different implementations are possible, based on platform or preference. Usability, scalability, and configuration problems arise for both initialization and implementation, but perhaps most strikingly for initialization.*

Taking the preprocessor path can lead to a constructor that is almost artful in its clutter and audacious in its unreadability:

```
class environment
{
public:
    ...
    explicit environment(
        #ifndef NO_ENVIRONMENT_CTOR_ARGS
        #ifdef ENVIRONMENT_FROM_FILE
        const std::string &file,
        #else
        #ifdef HOST_FROM_ARG
        const std::string &host,
        #endif
        #ifdef USER_FROM_ARG
        const std::string &user,
        #endif
        #ifdef COMMAND_LINE_FROM_ARGS
        const std::string &command_line,
        const std::vector<std::string> &arguments,
        #endif
        #ifdef VALUES_FROM_ARG
        const std::map<std::string, std::string> &values,
        #endif
        ...
        #endif
    );
    ...
};
```

The non-preprocessor approach will be marginally clearer in declaration, but will contain the superset of all possible arguments, which are selectively ignored according to the implementation. So, a long argument list with subtle interactions between the arguments and the implementation. This is best characterized as tedious and error prone.

Furthermore, the question of implementation variation needs to be addressed. For example, in addition to the production versions it makes sense to have a version that can be used for testing. In particular, unit tests require that the component under test is isolated from externals, such as registries, the file system, environment variables, etc, so that test cases can be written and run without the need for external configuration, dependencies that would make them integration rather than unit tests. An implementation of `environment` for testing purposes would have all of its values set in a test case on construction and would hold them explicitly, without reference to external APIs:

```
class environment
{
    ...
private:
    #ifdef TESTING
    std::string host_name, user_name;
    std::vector<std::string> args;
    std::map<std::string, std::string> values;
    ...
    #else
    ...
    #endif
};
```

The preprocessor approach again clutters the header file and scales poorly for multiple implementations. In addition, `environment` and its users need to be recompiled and relinked whenever testing is needed. A slightly looser binding is afforded by accommodating link-time rather than compile-time substitutability. The following definition is binary compatible for different implementations:

```
class environment
{
    ...
private:
    struct representation;
    representation *body;
};
```

Instead of relying on the preprocessor, different versions can be implemented in their own, standalone implementation files. It becomes a link-time decision as to which `environment` implementation is linked in. For example, the testing version can be defined and compiled without preprocessor tricks or reference to other implementations, so the following definition would appear in a compiled source file rather than a header file:

```
struct environment::representation
{
    std::string host_name, user_name;
    std::vector<std::string> args;
    std::map<std::string, std::string> values;
    ...
};
```

This is a significant improvement over the previous situation, but still fails to resolve the construction problem. It also seems to be a `make`-clever way of emulating polymorphism, for which C++ already has a perfectly good mechanism.

Therefore, define a DECOUPLED CONTEXT INTERFACE for the ENCAPSULATED CONTEXT OBJECT that has no mention of implementation. Users of the ENCAPSULATED CONTEXT OBJECT depend only on this interface and only the creator of the ENCAPSULATED CONTEXT OBJECT depends on its concrete implementation; runtime polymorphism bridges the two views.

The DECOUPLED CONTEXT INTERFACE is implemented as an INTERFACE CLASS, which is a class whose public interface contains only pure `virtual` functions and which contains no data or ordinary function implementation:

```
class environment
{
public:
    virtual std::string host() const = 0;
    virtual std::string user() const = 0;
    virtual std::string command_line() const = 0;
    virtual std::vector<std::string> arguments() const = 0;
    virtual bool has_value(const std::string &name) const = 0;
    virtual std::string operator[](const std::string &name) const = 0;
    ...
protected:
    ~environment();
};
```

The `protected` destructor ensures that object deletion is not possible at the interface level. Construction is restricted to the concrete classes that implement the DECOUPLED CONTEXT INTERFACE, and therefore the constructor arguments are specific to the actual ENCAPSULATED CONTEXT OBJECT type rather than the outcome of a preprocessor workout. The following is a MOCK OBJECT definition for testing purposes:

```
class mock_environment : public environment
{
public:
    mock_environment(
        const std::string &host_name, const std::string &user_name,
        const std::vector<std::string> &args,
        const std::map<std::string, std::string> &values, ...);
    virtual std::string host() const;
    virtual std::string user() const;
```

```
    virtual std::string command_line() const;
    virtual std::vector<std::string> arguments() const;
    virtual bool has_value(const std::string &name) const;
    virtual std::string operator[](const std::string &name) const;
    ...
private:
    std::string host_name, user_name;
    std::vector<std::string> args;
    std::map<std::string, std::string> values;
    ...
};
```

In addition to the other benefits of this approach, it is possible to have not only multiple ENCAPSULATED CONTEXT OBJECTs current in a program, but also multiple ENCAPSULATED CONTEXT OBJECT implementations. The implementations are isolated from one another, requiring no build magic to avoid interference. For example, this would allow a program to have one thread running a service that was written to depend on environment variables and another running a service that assumes its values come from a configuration file.

It may be tempting to try to add some default behavior into the INTERFACE CLASS, but this compromises its simplicity and implementation neutrality. There is no advantage in terms of inheritance because any common implementation can still be factored out into a middle level of the hierarchy; there is no need to lumber the otherwise pure root of the hierarchy with additional assumptions and implementation guesswork.

Likewise, introducing factory behavior into the root class in the form of a `static` function that can determine which implementation is required reintroduces many of the dependencies that the current loosely coupled design has eliminated. It closes the hierarchy to simple extension and makes the hierarchy "self-aware": a cyclic dependency is introduced from the implementation of the root of the hierarchy to its children. This increase in coupling is accompanied by a dilution in the root class's cohesion. In the context of the current design it adds only complexity, no benefit.

Pattern Story: A Tale of Orthogonal Implementations in Java

Consider a Java application that has been designed to be loosely coupled but where a number of classes still have a need for common services that can be considered to come from the execution context, such as logging, writing messages to a console, looking up configuration values, etc. These different services can be implemented and tested in a number of different ways.

How can objects in different parts of the application use reporting services and configuration lookup that are specific to the execution context? Application objects should not depend on the underlying mechanisms used and the use of a common context should not lead to the introduction of global dependencies, nor should passing context around result in unmanageably long argument lists.

The effect of global variables on software architecture is well understood, and the comparable effects of SINGLETON and MONOSTATE increasingly so. The initial gold rush to introduce a SINGLETON wherever there appeared to be a system-level object multiplicity of one has ebbed, and indeed the outgoing tide has revealed that much of what glittered was iron pyrite. There are some well-defined circumstances that require a hard constraint of only a single instance, such that more than one can be considered a genuine error, but most uses of globals (including non-`final public static`s), SINGLETONs, MONOSTATEs, etc, do not qualify. Furthermore, a creational constraint does not automatically imply global promiscuity.

Therefore, offer the reporting services and configuration lookup in an ENCAPSULATED CONTEXT OBJECT that is passed to application objects that need them. The object provides the housing for the context features and ordinary argument passing provides the mechanism for distribution.

The resulting design makes the use and propagation of execution context explicit and localized. The interface of the ENCAPSULATED CONTEXT OBJECT represents a one-stop shop for all the context-related services:

```
public class ExecutionContext
{
    public void writeLog(String message) ...
    public void writeConsole(String message) ...
    public boolean containsVariable(String name) ...
    public String valueOfVariable(String name) ...
    ...
}
```

The `ExecutionContext` class represents a stable container for what may be an evolving and expanding set of execution services. For example, access to the current date and the time of day could be added as new features without breaking existing users of `ExecutionContext`.

How can a user of an ENCAPSULATED CONTEXT OBJECT that uses only a specific, narrow aspect of its features gain access to only those features that it needs, without also finding itself dependent on a larger, potentially less stable interface? It is easy for an ENCAPSULATED CONTEXT OBJECT to degenerate into a BIG BALL OF MUD, a parameter version of a FORTRAN `COMMON` block.

The one-stop shop character of an ENCAPSULATED CONTEXT OBJECT is not always a benefit. The `ExecutionContext` class can easily grow to hold a slop bucket of methods, an incoherent aggregation of features. Application objects that need reporting services, such as `writeLog` and `writeConsole`, do not necessarily need to lookup configuration variables, and vice versa:

```
public void configure(ExecutionContext context)
{
    String serverName = context.valueOfVariable("server");
    ...
}
public void start(ExecutionContext context)
{
    context.writeLog("Preparing to start");
    try
    {
        ...
    }
    catch(RuntimeException caught)
    {
        context.writeLog("Failed to start: " + caught);
        context.writeConsole("Error: " + caught);
        throw caught;
    }
    context.writeLog("Started successfully");
}
```

Although being passed an ENCAPSULATED CONTEXT OBJECT is explicit, it is perhaps too explicit and overbearing for many uses.

Therefore, separate out ROLE-PARTITIONED CONTEXTs that target each usage role specifically. Dependent code now works in terms of more focused types without drawing in unrelated features or dependencies, or being necessarily subject to all evolutionary changes in the execution context needed for the application as a whole.

Application objects can now work in terms of narrow, more cohesive types:

```
public void configure(Configuration config)
{
    String serverName = config.valueOfVariable("server");
    ...
}
public void start(Reporting reporter)
```

```
{
    reporter.writeLog("Preparing to start");
    try
    {
        ...
    }
    catch(RuntimeException caught)
    {
        reporter.writeLog("Failed to start: " + caught);
        reporter.writeConsole("Error: " + caught);
        throw caught;
    }
    reporter.writeLog("Started successfully");
}
```

Of course, if ROLE-PARTITIONED CONTEXTS are segregated too finely, application objects that depend on many roles will end up with longer and less stable argument lists, which in turn would encourage reconsolidation away from ROLE-PARTITIONED CONTEXTs.

How should the ROLE-PARTITIONED CONTEXTs be defined? Certain specific usage views do not necessarily represent all the ways that the execution context might be used and they do not necessarily dictate the structure of the implementation.

There are many ways in which the different context aspects could be defined and combined. For example, the `Configuration` and the `Reporting` types could both be concrete classes, useable for distinct objects, but not for general combination by inheritance because of Java's single-subclassing constraint. On the other hand, one could be a `class` and others could be `interfaces`, allowing a lighter mix-in style. There is no obvious leading candidate to be the sole `class`. Application objects should not become coupled to execution context implementation specifics and assumptions, no matter how finely partitioned the context becomes.

Therefore, present each ROLE-PARTITIONED CONTEXT in terms of a DECOUPLED CONTEXT INTERFACE. Context users are now fully decoupled from any of the many implementation choices that an application may employ to express execution context. The dependency is solely on the usage contract.

This leads to the definition of a Java `interface` for each role:

```
public interface Reporting
{
    void writeLog(String message);
    void writeConsole(String message);
    ...
}
public interface Configuration
{
    boolean containsVariable(String name);
    String valueOfVariable(String name);
    ...
}
```

Any commonality can be factored out and introduced below the level of the root interfaces, leaving usage unaffected. It may be appropriate to implement a single ENCAPSULATED CONTEXT OBJECT for all ROLE-PARTITIONED CONTEXTs:

```
public class ExecutionContext implements Reporting, Configuration, ...
{
    ...
}
```

On the other hand, it may also be practical to consider this ripe for a DECOUPLED CONTEXT INTERFACE if the need is there:

```
public interface ExecutionContext extends Reporting, Configuration, ...
{
}
```

Stability at the interface roots does not necessitate stability at the implementation leaves — interfaces increase rather than decrease degrees of freedom.

How can independent implementations of ROLE-PARTITIONED CONTEXTS be defined and mixed freely? The implementation can vary widely according to both taste and test requirements.

For example, disabling reporting is a common need in both production and testing, and the need for configuration should not pull in external dependencies for unit tests. Although there may be a reasonable common implementation of the execution context used for most deployments, realizing all ROLE-PARTITIONED CONTEXTS in a single ENCAPSULATED CONTEXT OBJECT, this is not the case for all application and testing scenarios. Providing a concrete implementation for each of the combination of possible features leads to combinatorial explosion of implementation classes and more than a dash of duplicated code.

Therefore, provide ROLE-SPECIFIC CONTEXT OBJECTS, an ENCAPSULATED CONTEXT OBJECT for each ROLE-PARTITIONED CONTEXT. Assuming that clients depend only on the ROLE-PARTITIONED CONTEXTS, there will be no difference to usage code and freedom of choice in combination for the code defining how the application objects are to be employed.

For example, this means that a NULL OBJECT implementation can be provided for `Reporting` independently of how any other context aspects are implemented:

```
public class NullReporting implements Reporting
{
    public void writeLog(String message)
    {
    }
    public void writeConsole(String message);
    {
    }
    ...
}
```

And likewise, a MOCK OBJECT implementation of `Configuration` would not necessarily be coupled to any other choice:

```
public class InternalConfiguration implements Configuration
{
    public boolean containsVariable(String name) ...
    public String valueOfVariable(String name) ...
    public String setVariable(String name, String value) ...
    public boolean unsetVariable(String name) ...
    ...
    private Map<String, String> variables = new TreeMap<String, String>();
}
```

The same approach can be continued if the scope of the execution context expands, e.g. to take on querying date and time. System time can be considered a global variable that is asynchronously updated and accessed through static helpers (e.g. `System.currentTimeMillis`). As with other globals, this is not something that should be accessed freely in arbitrary pieces of code. To make clock dependency clearer and to allow MOCK OBJECTS for testing, time should be considered part of the execution context, being passed around and decoupled accordingly.

A Language

Words are but the signs of ideas.

Samuel Johnson

Where pattern stories are concrete and linear, pattern languages are more abstract and typically more richly interconnected. A pattern language defines a network of patterns that build on one another, so that one pattern can optionally or necessarily draw on another, elaborating a design in a particular way, responding to specific forces. A pattern story can be considered a path through a language with respect to a particular example.

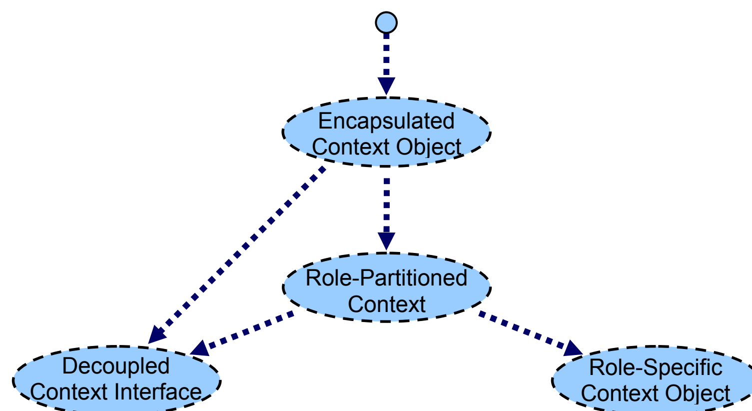
Context Encapsulation

The goal of the CONTEXT ENCAPSULATION pattern language is to present core practices for preserving loose coupling in software architectures based on static type systems. The pattern language could easily be adapted to include dynamically typed models, but that is not the focus here.

The language follows fairly classic pattern language form and substance: each pattern description presents a problem statement; each is elaborated with a sufficiently comprehensive set of forces; credible configurations address the forces; configurations may include other patterns for consideration and completion; the inclusion relationships form a small singly rooted directed acyclic graph.

The scope presented is narrowed to the core language. Although NULL OBJECT, MOCK OBJECT, and other patterns played valuable roles in the specific stories, they follow from the kernel of the language rather than defining it. A fuller presentation of the language would draw these other patterns into its presentation. There are further notes on pattern relationships within and outside the language at the end of this paper.

The *protagonist patterns* in the preceding stories form the set of patterns from which the essential CONTEXT ENCAPSULATION pattern language is formed. The basic connections in the language can be presented in a number of different ways, including visually. The following diagram uses a typical diagrammatic form:



However, diagrams are not the only approach available, and briefer, more formal approaches are also possible. Some of these are discussed toward the end of this paper.

Pattern: Encapsulated Context Object

In a system partitioned with respect to LAYERS, PLUG-INS, COMPONENTS, or other loosely coupled subsystems, there is a need to share common information and services that relate to the context of execution of the program across disparate parts of a program — externally configured values, logging, persistence, etc.

However, SINGLETONs and globals (and MONOSTATEs and `statics`) are too permissive and broad in scope. They provide uncontrolled access from all parts of a program, not just those specific parts that need it, and introduce unnecessary opportunities for coupling. Globals and SINGLETONs do not easily provide late binding, so that the implementation and instance they are based on is hardwired rather than runtime configurable. It is desirable to "program to interface, not an implementation" [Gamma+1995], but something like a SINGLETON leads to code that programs to an instance [Henney2003], which is even more restrictive and more tightly coupled than simply programming to a class implementation.

On the other hand, propagating many pieces of fine-grained information as individual variables and services as individual operations can lead to unmanageable, unmemorable, and unstable argument lists. Long argument lists are tedious, error prone, and troubled by an almost compulsive urge to grow longer.

Therefore, represent the information and services in an object that encapsulates the needed context and pass it explicitly via an argument to the client objects and operations that need the context.

Basic runtime substitutability is afforded by the standard argument passing mechanism. It is possible to modify the execution context of a component without necessarily having to modify the configuration or code of other parts of a system. It is also possible to run with multiple, different contexts within the same program, perhaps in different threads.

Over time, small changes to the information and services in the context are absorbed as the unit of stability is the ENCAPSULATED CONTEXT OBJECT and not its individual pieces of information and service operations, i.e. there is some slack in the stability of the context schema.

In statically typed languages, further decoupling, as is needed for endotesting, alternative production implementations, etc, can be offered through a DECOUPLED CONTEXT INTERFACE. Where the cohesion of the different information and services appears too loose, and each client does not — and should not — need all of the interface all of the time, the dependencies can be normalized using ROLE-PARTITIONED CONTEXTs.

An ENCAPSULATED CONTEXT OBJECT introduces an inversion of dependencies and responsibilities. It may also introduce an inversion of control flow, but this is not necessarily always the case: an informational context will tend to be more passive, but a more behavioral service context will generally result in more obvious inversion of control flow.

Pattern: Decoupled Context Interface

An ENCAPSULATED CONTEXT OBJECT allows common services and information to be propagated to disparate parts of a program through explicit argument passing. However, although more loosely coupled and precisely targeted than globals and their (a)moral equivalents, a context object still represents a given representation of the context. This manifests itself as a concrete class in the argument lists of any components dependent on the context.

Any change to the implementation of this type in a statically typed language can cause a rebuild (and redeployment) ripple effect. Although the binding to actual instance occurs at runtime, the type is hardwired, and alternative implementations, such as for testing, cannot be substituted easily or consistently.

For example, a production context object type may import externally set configuration values automatically via environment variables, but this would be unsuitable for unit testing, for which the test harness should be able to localize and control settings itself within the program, and does not accommodate alternative implementations, such key-based lookup of a centralized registry. Conditional compilation, as supported by the C preprocessor, is not available in all languages and, even when supported, represents at best a clumsy solution [Spencer+1992]. Conditional compilation hardwires the handling of variability into a single piece of code that grows with each variation, supports only a single variant at a time, and requires recompilation to switch between variants. The centralization of a potentially growing mass of dependencies can create an instability hotspot.

Therefore, define and present the interface that context clients will depend on separately from the implementation type. This named interface explicitly declares the operation signatures available to context clients; one or more concrete implementations can realize the interface independently.

Clients use and depend only on what they can see of the context object, which is its declared signatures. There is no dependency on a given implementation, so full runtime binding is possible. There is no rebuild ripple effect in changing the implementation of the actual context object class or substituting an alternative implementation class, whether a NULL OBJECT, a MOCK OBJECT, or a genuine alternative production variation. This separation supports product families, different platforms, experimental prototyping of implementation variations, testing, etc.

An uncohesive DECOUPLED CONTEXT INTERFACE can be further divided into ROLE-PARTITIONED CONTEXTs, giving rise to multiple DECOUPLED CONTEXT INTERFACEs. If appropriate, distinct DECOUPLED CONTEXT INTERFACEs can be composed into an additional interface that aggregates multiple interfaces by inheritance.

In Java, C#, and other languages with directly comparable type systems, the context interface is defined through `interface` types. In C++ and other statically typed languages that support full multiple inheritance, the context interface is defined as a fully abstract base class, an INTERFACE CLASS. In principle, compile-time polymorphism could be used, but for the common needs and benefits of ENCAPSULATED CONTEXT OBJECT, a runtime binding is more appropriate. In dynamically typed languages the implementation of ENCAPSULATED CONTEXT OBJECT can be said to rise to a DECOUPLED CONTEXT INTERFACE implicitly: the absence of a declared interface automatically provides the substitutability of alternative context implementations.

Pattern: Role-Partitioned Context

An ENCAPSULATED CONTEXT OBJECT combines commonly needed services and information into an object made available to specific clients. However, not all of the clients will need all of the interface all of the time, nor should they.

The services and information on offer may be at best coincidentally related, e.g. a logging service and access to user configuration values. This can make the interface, which may already be a DECOUPLED CONTEXT INTERFACE, appear uncohesive and a source of dependencies rather than a resolution of them. Although a client need draw no more from the context bucket than is strictly necessary, a broad interface still intrudes on the client's dependency set.

On the other hand, sometimes it is the bundling of services and information that is more stable than the specific services and information themselves, i.e. the aggregate can be more stable than its parts.

Therefore, partition the usage interface of a context object according to common usage, so that services and information are grouped and named according to the roles that they can play for a client.

Clients now declare the more specific role that they depend on rather than necessarily depending on the whole type. Each role type can typically be considered an interface in its own right, and can in turn be expressed as a DECOUPLED CONTEXT INTERFACE.

Where a broader context object interface is needed, the client can declare for it or accept an additional argument that specifies that specific role. This design balances the cohesion of the context propagated with the need to group different services and information together. The design is less intrusive that having a single broad context interface pervade the whole program, and more stable with respect to change in the detail of context and the requirements for it. There is also no need to implement only a single context object against multiple ROLE-PARTITIONED CONTEXTs: whether a single object realizes multiple interfaces or different interfaces are realized through separate ROLE-SPECIFIC CONTEXT OBJECTs is sufficiently transparent to the client.

Thus, a ROLE-PARTITIONED CONTEXT is expressed through multiple DECOUPLED CONTEXT INTERFACEs, or separate ROLE-SPECIFIC CONTEXT OBJECTs, or a combination of the two.

Pattern: Role-Specific Context Object

An ENCAPSULATED CONTEXT OBJECT gathers together services and information for use in disparate parts of a program. These services and pieces of information may not be collectively cohesive in terms of their use so that the broad interface can be divided into different ROLE-PARTITIONED CONTEXTs. However, although the interface may be partitioned cohesively, the implementation is still bundled as one.

The client no longer depends on the whole context available with a program, thanks to its role partitioning, but the resulting context object still gathers together all the different role aspects in a single implementation. Such an implementation may still be considered coincidentally cohesive.

There may be a need not only for using different context aspects independently, but also for providing alternative context aspect implementations. For example, the decision to provide a file-based instead of console-based log is conceptually independent from the decision to use registry-based or file-based application configuration details. These are orthogonal concerns, and so they should not be coupled in a single unit of code.

Within a program's runtime, different context roles may change at different rates. For example, one role may define information or services that remain unchanged over the whole duration of a program, whereas another may define a context that is stable and valid over a much shorter lifetime, such as a connection, a transaction, or a call out.

Therefore, define separate implementations for each ROLE-PARTITIONED CONTEXT, and provide a suitable instance to each context client depending on its usage of specific services and information and the implementation properties required for the program.

A client continues to depend on the declared interface and passed reference to the context it requires. The client is not inappropriately tied to other services and information that it may not or should not need, so it remains unaffected by any build-time issues that changes in these may bring. In addition, the implementation of a particular piece of context is also not bound to the implementation of other aspects of context that may be unrelated.

If each ROLE-SPECIFIC CONTEXT OBJECT implements a DECOUPLED CONTEXT INTERFACE, different pieces of context can be individually substituted according to need, e.g. NULL OBJECT, MOCK OBJECT, or proper alternative production implementations. If not, then only instance-level variation is possible. Either way, the use of ROLE-SPECIFIC CONTEXT OBJECTs can directly support context aspects that have different lifetimes and rates of change.

The more fine-grained approach of ROLE-SPECIFIC CONTEXT OBJECTs is less intrusive and more discreet for many common uses, but can become fragmented and difficult to manage if carried out aggressively on larger contexts, leading to long, rambling argument lists.

Some Sequences

Separation of tasks is a good thing, on the other hand we have to tie the loose ends together again!

Edsger Dijkstra

Pattern sequences are an often overlooked but useful tool for presenting and walking through pattern languages. They are related to pattern stories in the same way that individual patterns are related to examples that illustrate or motivate those patterns.

In one sense, pattern sequences can be seen to have always been an implicit part of the pattern concept. It is the sense that a design unfolds through the paths laid out by a language, and that there are many paths that can be taken. A given pattern story recalls a walk down one path, and no two paths are ever the same and each stroll down the same path can be different. However, in another sense, a rigorous and clear notion of pattern sequences has been largely missing as an explicit part of the mainstream pattern concept, and has only recently been given this attention.

The three stories presented earlier in the paper highlighted three different sequences by example. To clarify terminology, a pattern sequence is not a pattern story: the story arises in the telling, but it may in part be characterized by a given pattern sequence. The same sequence may be applied in different situations giving rise to distinct stories — each different in its detail but each related through the same pattern sequence.

On Pattern Sequences

Although the word *sequence* is used in Christopher Alexander's original pattern writing [Alexander+1977, Alexander1979], its meaning is often ambiguous and neither its claim to being a distinct label nor its role within the pattern argot is always clear. Sometimes the notion of sequence does appear to be singled out as a first class role. But at other times, in the same context, the word is used in its common descriptive sense, simply denoting a succession of items, rather than as a label for an artifact with identity, something with the same first-order significance as patterns or pattern languages. And then there are times that the word is used loosely and with some ambiguity. Given the degree of imprecision and the different uses to which the word is put, relying on this material for a clear and definitive definition of *sequence* in the context of patterns is not without problems.

It is fairer to say that although the relationship between pattern languages and the process of applying them through sequences emerged at the same time as the concept of patterns and pattern languages, pattern sequences did not come of age as a more explicit term and a more crisply defined notion until more recently. While the process of applying patterns was considered important and some detail provided, some of the significant detail on how to do so effectively was missing [Porter+2004]:

> Pattern languages give only a weak notion of the order in which patterns should be introduced during the construction process. Larger patterns are applied before smaller patterns; larger patterns are shown higher in the language diagram than smaller patterns, and arrows run from larger to smaller patterns. Alexander further notes that adjacent patterns should be applied as close in sequence as possible, but neither Alexander's theory or language diagrams indicate how to order adjacent patterns.

Thus, a pattern language frames a possible set of descents which are played out as pattern sequences. Each pattern sequence describes a process of design that embraces some or all of the patterns in a language. However, judging the qualities or appropriateness of a given pattern sequence has been left as a private and informal matter. More recently, pattern sequences have gained fuller recognition and a more thorough treatment [Alexander2002, Coplien+2005, Porter+2004], although much of this treatment still tends more toward the abstract than the concrete.

A given pattern sequence can be considered a highly specific development process [Coplien+2005]:

> The process thus involves step-by-step adaptation with feedback. Simply following the pattern language doesn't give you a clue about how to handle the feedback.

Or, more accurately and practically, a pattern language includes its sequences, and the knowledge of how to handle feedback should be considered part of the scope and responsibility of a language rather than as some supplementary artifact. Either way, a given pattern sequence can be used as a guide to the reader for one way that a language has been, can be, or is to be used. When taken together, a number of sequences can be seen to provide guidance on the use of a given pattern language — or, alternatively, when taken together, a number of sequences can be used as the basis of a pattern language.

A common medium for illustrating sequences is stories, but sometimes the stories are too specific — in the way that a motivating example in a pattern is sometimes mistaken as the pattern, a pattern story may end up stealing the limelight from the pattern sequence it represents.

Thus, when made distinct, sequences have the potential to play a number of roles. However, with the exception of pattern stories, they are normally not made explicit as part of the presentation of a language. So there is generally more discussion about pattern sequences than actual cataloging or specific description. Given that different pattern sequences give rise to different common design fragments with different properties, useful in different situations, it seems worthwhile documenting some of these, even if briefly.

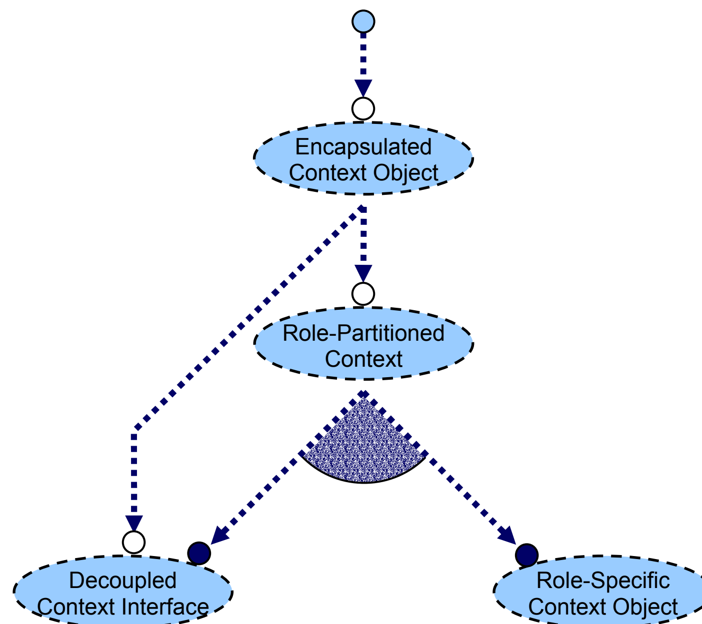
Pattern sequences also offer one way of making sense of the popular notion of compound patterns [Riehle1997, Vlissides1998]. From the perspective of pattern sequences compound patterns might be more accurately named *pattern compounds*. Thus, a pattern compound is an identifiable and common sequence of patterns that can be considered as a whole with respect to the problem it addresses and the design it achieves. This take on pattern compounds resolves the problem that some pattern commentators have had of where to place pattern compounds with respect to lone patterns and pattern languages. It also becomes clearer that there has been a convergence of quite distinct but complementary traditions: Christopher Alexander's work in the built environment; the use of stories to relate how patterns are combined in a particular design; the interest in recurring compounds of patterns that do not seem to have the ambition or breadth of pattern languages, and yet clearly have more to say than their constituent patterns. A practical view of sequences can be arrived by combining the relevant perspectives from each, leading to a more unified understanding of patterns in the context of software.

Sequences can be shown diagrammatically, or they can be presented as a list presented in prose — e.g. "apply ENCAPSULATED CONTEXT OBJECT and then DECOUPLED CONTEXT INTERFACE" or, less sequentially but with better prose style, "implement an ENCAPSULATED CONTEXT OBJECT with a DECOUPLED CONTEXT INTERFACE" — or in a terser, more formal notation — e.g. ⟨ECO, DCI⟩. Of course, as with individual patterns, pattern compounds, and pattern languages, there is also a good case for naming certain pattern sequences, particularly where the sequence becomes too long to describe conveniently.

Patterns Sequences: Context Encapsulations

A pattern story can be considered to retell an occurrence of a particular pattern sequence, and many sequences can be generated from a pattern language, depending on its structure. Which leads to the next point: the common diagrammatic presentation of pattern language connections shows geometry but not rules of combination. Not all sequences generated through a simple hierarchical view of the pattern language are necessarily valid.

For example, it is possible to apply ENCAPSULATED CONTEXT OBJECT on its own, or in conjunction with DECOUPLED CONTEXT INTERFACE, but not solely in conjunction with ROLE-PARTITIONED CONTEXT, which requires either one or both of DECOUPLED CONTEXT INTERFACE and ROLE-SPECIFIC CONTEXT OBJECT. Or, put more briefly, ⟨ECO⟩ and ⟨ECO, DCI⟩ are valid sequences, but ⟨ECO, RPC⟩ is not. In other words, what the language needs is grammar, and this is normally embedded in the pattern descriptions rather than being made visible at the level of a pattern language diagram. The following diagram includes some constraints on combination:



The constraints are based on feature modeling notation [Czarnecki+2000]: an open circle on a relationship end indicates an optional relationship, a filled circle a mandatory relationship, and a filled arc between mandatory relationships indicates that one or more of the relationships is required.

The basic traversal rules ensure that not only is a pattern sequence partially ordered as indicated by the basic connections tree, but that a sequence must start at the root, thus ⟨ECO, RPC, DCI, RSCO⟩ is a valid sequence but ⟨RPC, DCI, RSCO⟩ is not. There is more discussion of traces through a pattern language and of language grammar in the final section of this paper.

The following documented sequences present a subset of the traces possible through the CONTEXT ENCAPSULATION language along with notes on their applicability. Of course, there are also other ways of using these patterns that are not constrained by the topology and grammar of the language; what is being presented is how they can work together in the context of the CONTEXT ENCAPSULATION language.

Pattern Sequence ⟨⟩

Use no patterns from the language.

The empty sequence is a valid sequence from a pattern language. It is the degenerate case that corresponds to not applying the pattern language. It emphasizes that a pattern is not a universal, context-free piece of advice: in the general case it advises that applying patterns from a language is optional rather than mandatory.

Pattern Sequence ⟨ECO⟩

Introduce an ENCAPSULATED CONTEXT OBJECT.

In the limit, the root pattern represents a valid sequence within this language. Applying this sequence yields a design that is more loosely coupled than a corresponding design that relies on globals and their kindred spirits. It may be sufficiently cohesive and support loose enough coupling without further refinement. It offers instance-level substitutability of context. It can also offer type-level substitutability over the continued development lifetime of the system as the ENCAPSULATED CONTEXT OBJECT's interface evolves. However, if the context is broad and embraces many roles, the interface to the context object can be uncohesive and cumbersome.

This is the lonely pattern sequence behind the *Tale of Pluggable Implementations in C#* story.

Pattern Sequence ⟨ECO, DCI⟩

Introduce an ENCAPSULATED CONTEXT OBJECT and define a DECOUPLED CONTEXT INTERFACE.

This sequence improves on the basic application of ENCAPSULATED CONTEXT OBJECT by allowing type-level substitutability within the lifetime of an executing program, not just the evolution of its code. It separates concerns more fully, minimizing dependencies that clients have on the context, and clarifying the contract for the ENCAPSULATED CONTEXT OBJECT's implementation in the form of a DECOUPLED CONTEXT INTERFACE.

This is the core pattern sequence behind the *Tale of Alternative Implementations in C++* story.

Pattern Sequence ⟨ECO, RPC, DCI, RSCO⟩

Introduce an ENCAPSULATED CONTEXT OBJECT, divide its use into ROLE-PARTITIONED CONTEXTs expressed as DECOUPLED CONTEXT INTERFACEs, and then divide its implementation into ROLE-SPECIFIC CONTEXT OBJECTS.

This sequence provides interface–implementation decoupling and focuses on context cohesion, partitioning it according to role so that a client does not depend on much more than they need. With this division of context interface in place, it becomes possible to split the context object as well. However, overapplication of this sequence can lead to fragmentation rather than loose coupling, with the context being divided so finely as to be unwieldy and incomprehensible.

This is the core pattern sequence behind the *Tale of Orthogonal Implementations in Java* story.

Pattern Sequence ⟨ECO, DCI, RPC, RSCO⟩

Introduce an ENCAPSULATED CONTEXT OBJECT, define a DECOUPLED CONTEXT INTERFACE, and then divide its use into ROLE-PARTITIONED CONTEXTs implemented as ROLE-SPECIFIC CONTEXT OBJECTS.

This sequence has the same pattern alphabet as the previous sequence, ⟨ECO, RPC, DCI, RSCO⟩, but applies them in a different order, focusing first on decoupling the interface before partitioning it.

Although the end result may be the same, the emphasis and properties of this sequence are subtly different concerning how the sequence can play out over time. It is possible to break the previous sequence, ⟨ECO, RPC, DCI, RSCO⟩, into the following discrete steps: ⟨ECO⟩ followed by ⟨RPC, DCI⟩ followed by ⟨RSCO⟩, with each stage yielding a working system. It is possible to break the current sequence, ⟨ECO, DCI, RPC, RSCO⟩, into the following steps, with arbitrary pauses between them: ⟨ECO⟩ followed by ⟨DCI⟩ followed by ⟨RPC, RSCO⟩. The STABLE INTERMEDIATE FORMS [Henney2004] that can be followed in each sequence are slightly different.

Pattern Sequence ⟨ECO, DCI, RPC, DCI, RPC, DCI⟩

Introduce an ENCAPSULATED CONTEXT OBJECT with a DECOUPLED CONTEXT INTERFACE, divide its use with ROLE-PARTITIONED CONTEXTs expressed in turn as DECOUPLED CONTEXT INTERFACEs. Later, based on feedback from use, revisit the partitioning to further rearrange and divide ROLE-PARTITIONED CONTEXTs as DECOUPLED CONTEXT INTERFACEs.

Another, formal way of writing this pattern trace might be ⟨ECO, DCI⟩^⟨RPC, DCI⟩^{2}, which hints at what is going on: repetition and revision, handling feedback and refactoring accordingly. And once you have one power in there, in principle there is no reason you cannot have another, suggesting that sequences are not quite as bounded and unrepetitive in practice as they might first appear in theory.

A Few Notes

A witty saying proves nothing.

Voltaire

Notes on the Pattern Language

There are many patterns, and not just ENCAPSULATED CONTEXT OBJECT, that improve the adaptability of a piece of code with respect to its environment, whether large or small, with the typical effect of reducing coupling and the not infrequent effect of inverting the flow of control, e.g. PLUGGABLE BEHAVIOR [Beck2003] (more commonly, but less reasonably, known as STRATEGY [Gamma+1995]), MOCK OBJECT, DEPENDENCY INVERSION [Martin+1996a], DEPENDENCY INJECTION [Fowler2004], PLUG-IN, MICROKERNEL [Buschmann+1996], CONTAINER [Völter+2002], and others. Underpinning all of these can be said to be an inversion of responsibilities that has been characterized as the PARAMETERIZE FROM ABOVE pattern:

> Within a layered system, some commonly used complex objects or simple values, used by different layers or by many different parts of a given layer, may find themselves expressed in global form, e.g. as SINGLETON or MONOSTATE objects. This parameterizes components from below, but hardwires their dependencies, increasing the coupling of the component and making alternatives difficult or impossible to substitute. Instead, invert the relationship, so that these objects are passed in from above, i.e. so that the calling or owning component in the layer above passes in the appropriate instance.

Whether in terms of individual parameterizing integers or larger-scale, architecturally significant objects, to PARAMETERIZE FROM ABOVE represents a common reaction to unnecessary system-wide hardwiring of certain assumptions and facilities, and hence it is often employed in reaction to global variables, SINGLETONs, etc. Thus, an ENCAPSULATED CONTEXT OBJECT is an application of PARAMETERIZE FROM ABOVE, made more specific with respect to its context (sic) of application and its decomposition in terms of other patterns.

One side of ENCAPSULATED CONTEXT OBJECT's character is about separation, placing dependencies upstream rather than downstream from a piece of code; the other side is about aggregation, drawing together different parts into one or more wholes. Thus, an ENCAPSULATED CONTEXT OBJECT is also an ARGUMENT OBJECT [Noble1997] and can arise as a result of INTRODUCE PARAMETER OBJECT [Fowler1999]. Where the reaction to globals leads to greater flexibility, aggregation targets the question of stability and usability.

However, such aggregation can lead to a natural tension in a system, where lowering coupling may not necessarily lead to a corresponding increase in cohesion: a single, lumpen mass of coincidentally related features passed around like that somehow necessary but undrunk bottle of Liebfraumilch that used to attend many parties. It is this tension that is released by the other patterns in the CONTEXT ENCAPSULATION language, ROLE-PARTITIONED CONTEXT in particular.

A ROLE-PARTITIONED CONTEXT provides the counterbalance to the common tendency to have just *a* context object rather than a possibly plurality of them, subdivided according to common need. It is a specific application of INTERFACE SEGREGATION [Martin1996b] or ROLE DECOUPLING [D'Souza+1999] within the CONTEXT ENCAPSULATION language. It is most

often supported with a DECOUPLED CONTEXT INTERFACE, itself an application of an EXPLICIT INTERFACE [Buschmann+2003] within the CONTEXT ENCAPSULATION language.

The modularity and separation offered by CONTEXT ENCAPSULATION is a developmental mechanism that can also offer load-time and runtime configuration flexibility. A platform that offers support for dynamic loading based on paths or manifests allows a program to take advantage of alternative implementations of a DECOUPLED CONTEXT INTERFACE when it is configured to run. Explicit dynamic loading — such as the Win32 `LoadLibrary` function, the standard UNIX `dlopen` function, or the Java `Class.forName` methods — allows a programmer to extend this flexibility into a program's runtime.

The CONTEXT ENCAPSULATION language can be considered to be a sublanguage of a larger whole, a language whose narrative imperative is partitioning for stability and loose coupling. This language has two dominant patterns: LAYERS, whether applied locally or globally, introduce a separation, grouping elements according to abstraction criteria, stability, technology, etc; PARAMETERIZE FROM ABOVE for keeping the connections between layered elements supple. This architectural style can be applied both locally and globally. As a micro-architectural style it gives rise to common techniques such as ENCAPSULATED CONTEXT OBJECT and PLUGGABLE BEHAVIOR. When applied more strategically as a macro-architectural style, it can be seen as the foundation of PLUG-IN, CONTAINER, MICROKERNEL, and other architectures.

Regardless of scale, the effect is to keep the dependency horizon of any given element close, making any region of dependencies local relative to the system as a whole. When applied consistently across a system or group of components, *inversion layers* emerge. These are layers made up of interfaces, with perhaps a few supporting value and exception types, on which higher layers can depend but across which the code of higher layers depend on almost nothing — they rely on the existence of an implementation, but not directly on the implementation code. This makes code-to-code layering much more local than is commonly assumed: the PowerPoint architectural view still shows the top-to-bottom dependencies all the way through the technology layers, and the runtime is stacked like this, but this larger structure is invisible to the code itself. Each seam in the system is lined with interfaces, which are typically also packaged apart from the implementations as SEPARATED INTERFACES [Fowler2003]. Based on its properties, this can be said to be an *orthogonal architecture*.

Visually there are many ways of representing this architectural style: the inversion can be emphasized by drawing the layers in their conventional top-to-bottom ordering, the inversion layer representing a point on the diagram where dependencies flip from being downward pointing to being upward pointing; the orthogonality of implementations of an inversion can be emphasized either by drawing all implementations out to one side at right angles or all implementations arranged radially around a core; preserving a downward-only direction for inter-component dependencies leads to a view that emphasizes parameterization from above, with the implementations appearing off to one side but at the same level as the application or service root itself.

In addition to separation, along with its interesting geometric interpretations, another significant property of this general architectural style is stability. Grouping many unstable things together can result in a whole that is more stable than the individual parts. This view accounts for software development as a learning activity that incorporates learning [Henney2005]: something that is dynamic rather than static. An unchanging system has no need for design practices that buffer against change, but a changing system needs slack.

Notes on Pattern Sequences and Languages

In contrast to many existing pattern publications that reference sequences, but focus on specific stories, this paper makes the explicit point of presenting sequences within a language, and not just stories associated with sequences. There are a number of observations can be taken from documenting sequences and considering their relationship to pattern languages. Although these observations are not the primary goal of the paper, they are nonetheless worth including in brief appendix form because they have the potential to offer a deeper understanding of patterns as processes than appears to have been presented to date. There is no intention to present a calculus of patterns, but formalisms are drawn in as necessary to help draw parallels and shed light on the relationship between pattern sequences and languages.

The common diagrammatic presentation of connections in a pattern language invariably suggests that formal descriptions of pattern language structure can offer some insight into their application, even though the full concept and value of patterns remains inherently beyond the reach of mathematical tools. Ronald Porter et al [Porter+2004] offer a simple mathematical presentation of some of the ideas that relate sequences to languages, but do not follow through to the process-based view described here.

That one pattern comes 'above' another in a pattern language can be captured in an ordering relation [Porter+2004], and this can be used to define a rule that governs the partial ordering of members of the pattern set:

$$\text{ECO} < \text{DCI} \wedge \text{ECO} < \text{RPC} \wedge \text{RPC} < \text{DCI} \wedge \text{RPC} < \text{RSCO}$$

As a directed acyclic graph, the language's geometry can also be presented as a partially ordered set (a poset), which can be expressed as a set of tuples that define the edges of the graph:

$$\{(\text{ECO}, \text{DCI}), (\text{ECO}, \text{RPC}), (\text{RPC}, \text{DCI}), (\text{RPC}, \text{RSCO})\}$$

However, although we can use various formalisms, it is important to be aware of their scope of applicability. In particular, although it has been claimed that a "pattern language is a poset" [Porter+2004], this is not quite the case. It may be true that the topology of a pattern language is a poset, but there is somewhat more to pattern languages than just topology.

A defining characteristic of any language is its grammar, and topology is not grammar. It is from grammar that we can generate sequences. We can view pattern languages as containing, among other things, a vocabulary and a grammar, from which we can derive both its spatial connections (its topology) and its temporal connections (its set of sequences), as well as other properties. Of course, there is also more to language than grammar, but a discussion of language without consideration of grammar is likely to be unnecessarily limited.

We can also view pattern languages as processes for design, and it is this view of processes that suggests a different and perhaps more constructive perspective for understanding pattern sequences in a formal light. We are concerned with the idea of generative processes that progressively transform the designed state of a system, so it seems not unnatural to look at other process models. Key elements of the CSP model [Hoare1985] have found application both within and beyond the description of communicating sequential processes in concurrent systems [Hoare+1999]. There are a number of parallels which can be drawn that are useful: the patterns within a pattern language form the *alphabet* of the development process; the application of a pattern constitutes an *event*; the possible pattern sequences through a language forms the set of *traces* of the process.

The definition of a language as a graph offers a simple and powerful pictorial view of a pattern language's connections, but it lacks some of the rules of combination necessary for understanding the composition of pattern sequences. We can consider the process definition to be like the grammar of a language, in which case the grammar of CONTEXT ENCAPSULATION can be presented as follows:

ECO →
  ((DCI → (RPC → (RSCO | □) | □))
  | (RPC → (DCI → (RSCO | □) | RSCO → (DCI | □)))
  | □)
| □

Where → indicates sequential composition, | alternation, and □ completion in this improvised notation. A variation on this notation would focus on a starting state, ∅, rather than a state of completion, and the notion of optional sequential composition, $\rightarrow_\circ$:

∅ $\rightarrow_\circ$ ECO $\rightarrow_\circ$
  ((DCI $\rightarrow_\circ$ RPC $\rightarrow_\circ$ RSCO)
  | (RPC → (DCI $\rightarrow_\circ$ RSCO | RSCO $\rightarrow_\circ$ DCI)))

Note that, in principle, any pattern language must admit an initial state of completion: applying a given pattern language is an option not a law. A BNF-derived notation would be yet another way to present the grammar shown, as would a prose description:

> ENCAPSULATED CONTEXT OBJECT may optionally be applied. It may optionally be followed by DECOUPLED CONTEXT INTERFACE, which may optionally be followed by ROLE-PARTITIONED CONTEXT, which may optionally be followed by ROLE-SPECIFIC CONTEXT OBJECT. Alternatively, ENCAPSULATED CONTEXT OBJECT may optionally be followed by ROLE-PARTITIONED CONTEXT which must be followed by DECOUPLED CONTEXT INTERFACE, or ROLE-SPECIFIC CONTEXT OBJECT, or both, in either order.

It is easy to see from even this simple example that either a diagram or an enumeration of common sequences can offer a more effective presentation than either a process specification or a prose description, although formulae for process specification can offer a more useful medium for reasoning against, and a prose form can help to boost an author's word count, if nothing else.

The grammar described for CONTEXT ENCAPSULATION is simplified in that it does not include recursion. Although pattern languages are often presented as hierarchical and without cycles, this is a somewhat simplified — even naïve — view of software design, although it is certainly a convenient one. Revisiting and refining is inherently part of the design process and, in a domain as aphysical and scale free as software, the notion of physical levels lacks a firm foothold — it is useful as metaphor, but not as identity. Design is inevitably more truly recursive in software than, for instance, design in the built environment, where scale differentiation is more obvious and fundamental, although there are often similarities between different levels of granularity in the real world. The same can also be said of the notion of designing in sequences: it is a useful discipline, but it is not necessarily a realistic portrayal of design or the designer's mind. The world — including both given and designed domains — is inherently concurrent. Any sequences that are apparent arise from intentional or unintentional abstraction and the focused nature of human perception.

Some Acknowledgments

I would like to thank Allan Kelly, without whom there would have been no context for the paper, correspondence that led to the first draft of these patterns being hatched, encouragement to proceed with the patterns as part of a larger paper, or feedback on this paper. I would also like to thank Charles Weir for his shepherding and patience, and Frank Buschmann for his additional comments.

A Few References

[Alexander1975] Christopher Alexander, *The Oregon Experiment*, Oxford University Press, 1975.

[Alexander+1977] Christopher Alexander, Sara Ishikawa, Murray Silverstein, et al, *A Pattern Language*, Oxford University Press, 1977.

[Alexander1979] Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.

[Alexander2002] Christopher Alexander, *The Nature of Order, Book 2: The Process of Creating Life*, Center for Environmental Structure, 2002.

[Ball+1997] Steve Ball and John Crawford, "Monostate Classes", *C++ Report* 9(5), SIGS, May 1997.

[Beck2003] Kent Beck, *Test-Driven Development by Example*, Addison-Wesley, 2003.

[Buschmann+1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.

[Buschmann+2003] Frank Buschmann and Kevlin Henney, "Explicit Interface and Object Manager", *Proceedings of the 8th European Conference on Pattern Languages*, UVK, 2003.

[Coplien+2005] James O Coplien and Neil B Harrison, *Organizational Patterns of Agile Software Development*, Prentice Hall, 2005.

[Czarnecki+2000] Krzysztof Czarnecki and Ulrich W Eisenecker, *Generative Programming*, Addison-Wesley, 2000.

[D'Souza+1999] Desmond D'Souza and Alan Cameron Wills, *Objects, Components, and Frameworks with UML*, Addison-Wesley, 1999.

[Foote+2000] Brian Foote and Joseph Yoder, "Big Ball of Mud", *Pattern Languages of Program Design 4*, edited by Neil Harrison, Brian Foote, and Hans Rohnert, Addison-Wesley, 2000, `http://www.laputan.org/mud/`.

[Fowler1999] Martin Fowler, *Refactoring*, Addison-Wesley, 1999.

[Fowler2003] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[Fowler2004] Martin Fowler, "Inversion of Control Containers and the Dependency Injection pattern", January 2004, `http://www.martinfowler.com/articles/injection.html`.

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, 1995.

[Henney1999] Kevlin Henney, "Substitutability: Principles, Idioms and Techniques", *JaCC*, September 1999, available from `http://www.curbralan.com`.

[Henney2002] Kevlin Henney, "Null Object", *Proceedings of the 7th European Conference on Pattern Languages of Programs*, UVK, 2002, also available from `http://www.curbralan.com`.

[Henney2003] Kevlin Henney, "Eins oder Viele?", *JavaSPEKTRUM*, September 2003, available in English as "One or Many?" from `http://www.curbralan.com`.

[Henney2004] Kevlin Henney, "Stable Intermediate Forms", *EuroPLoP 2004*, July 2004, available from `http://www.curbralan.com`.

[Henney2005] Kevlin Henney, "Learning Curve", *Application Development Advisor*, March 2005, will be available from `http://www.curbralan.com`.

[Hoare1985] C A R Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[Hoare+1999] C A R Hoare and He Jifeng, "A trace model for pointers and objects", *ECOOP '99 Proceedings*, Springer, 1999, available from `http://research.microsoft.com/~thoare/`.

[Kelly2003] Allan Kelly, "Encapsulate Context", *Proceedings of the 8th European Conference on Pattern Languages*, UVK, 2003.

[Kelly2004] Allan Kelly, "Encapsulate Context", *Overload 63*, October 2004, `http://www.allankelly.net/patterns/encapsulatecontext.pdf`.

[Mackinnon+2000] Tim Mackinnon, Steve Freeman, and Philip Craig, "Endo-Testing: Unit Testing with Mock Objects", *XP2000*.

[Marquardt1999] Klaus Marquardt, "Patterns for Plug-Ins", *Proceedings of the 4th European Conference on Pattern Languages*, UVK, 1999.

[Martin1996a] Robert Martin, "The Dependency Inversion Principle", *C++ Report*, May 1996.

[Martin1996b] Robert Martin, "The Interface Segregation Principle", *C++ Report*, August 1996.

[Noble1997] James Noble, "Arguments and Results", *The Computer Journal*, 1997, `http://citeseer.nj.nec.com/107777.html`.

[O'Callaghan1999] Alan O'Callaghan, "Patterns for Change", *Proceedings of the 4th European Conference on Pattern Languages*, UVK, 1999.

[Overload2004] "Editorial", *Overload 64*, December 2004.

[Overload2005] "Letters to the Editor", *Overload 65*, February 2005.

[Porter+2004] Ronald Porter, James O Coplien, and Tiffany Winn, "Sequences as a Basis for Pattern Language Composition", *Science of Computer Programming*, Elsevier, 2004.

[Riehle1997] Dirk Riehle, "Composite Design Patterns", *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, 1997, `http://www.riehle.org/computer-science/research/1997/oopsla-1997.html`.

[Spencer+1992] Henry Spencer and Geoff Collyer, "`#ifdef` Considered Harmful, or Portability Experience with C News", *USENIX*, June 1992, `http://www.literateprogramming.com/ifdefs.pdf`.

[Vlissides1998] John Vlissides, "Pluggable Factory, Part I", *C++ Report*, November 1998, `http://www.research.ibm.com/designpatterns/pubs/ph-nov-dec98.pdf`.

[Völter+2002] Markus Völter, Alexander Schmid, and Eberhard Wolff, *Server Component Patterns*, Wiley, 2002.