Stable Intermediate Forms

A Foundation Pattern for Derisking the Process of Change

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

January 2005

Abstract

The universe is change; life is what thinking makes of it.

Marcus Aurelius

Change is often associated with risk, in particular the risk of failure or unwanted side effects. Whether such change is the move from one position on a rock face to another during a climb, or the progress of development through a software project, or the change of state within an object when it is being copied to, risk is ever present. In all cases, change is associated with questions of confidence, consequence, and certainty. In the event of any failure or unknowns, change can cause further failure and greater unknowns — a missed foothold, a missed deadline, a missed exception.

STABLE INTERMEDIATE FORMS is a general pattern for making progress with confidence and limiting the effect of failure. When undertaking a change or series of changes to move from one state to another, rather than carrying out a change suddenly and boldly, with a single large stride but uncertain footfall, each intermediate step in the process of change is itself a coherent state. This high-level pattern manifests itself in various domain-specific patterns, three of which are reported informally in this paper and used as examples: THREE POINTS OF CONTACT for rock climbing; ITERATIVE AND INCREMENTAL DEVELOPMENT for software development process; and COPY BEFORE RELEASE for exception safety in C++. It is a matter of preference and perspective as to whether this paper is considered to document one, three, or four patterns.

Thumbnail

Change typically involves risk. Any change from one state of affairs to another that cannot be characterized as atomic inevitably involves a number of steps, any one of which could fail for one reason or another, leaving the change incomplete and the circumstances uncertain. Therefore, ensure that each intermediate step in the process of change expresses a coherent state, one that in some meaningful way represents a whole rather than a partial state of affairs.

Example: Three Points of Contact

There you are, halfway up a cliff. Your face pressed against the rock face. Gravity is carefree but ever present. The ground is unforgiving and ever distant. The rock is your friend. And you hope that your friend, standing on the ground below you, is like a rock. In the event of a slip, his belaying should provide you with the failsafe that prevents you from joining him too rapidly.

To go up, you need to go right and then up. To do this you need to reach for what promises to be a good hold, but it is a little too far away to reach comfortably. There is no convenient ledge or outcrop to rest your whole weight on, and the rock is both sheer and wet. You recall that really cool bit at the beginning of *Mission Impossible 2* where free-climbing Tom Cruise, presented with a challenge, leaps from one rock face to another and manages, with a little grappling, to secure himself. Very cool... but not really an option: (1) Tom Cruise lose his grip and only just manages to avoid falling; (2) you are not in a movie; (3) you are the lead climber, so there is still a little way to fall and some significant rock to bounce off before the rope becomes taut and carries your weight; and (4) you are a relatively inexperienced climber and new to leading, so experience is less likely to be on your side.

Alternatively, you can try leaning over, with one foot jammed in the crack you are currently using as a handhold, and reaching for the new hold with your right hand as you angle to the right. There is nothing substantial that your left hand can hold once you have stretched to the right and the other foot would have to be free, providing balance but not support. Should work, if you have the distance right (being a little taller at this point would have helped...), but not exactly a sure thing.

However, there is yet another way. By dropping down a little, reversing some of your ascent, and then climbing across and then up, you could do it without either the dramatics or the uncertain leaning. There are enough handholds to afford you three points of contact most of the way, ensuring that you can shift your weight more easily across the rock face — two feet and one hand or two hands and one foot providing support at all times. Although it'll take another minute there is a much greater chance of success and much greater certainty that it will be only a minute.

Example: Iterative and Incremental Development

There are a number of different activities involved in taking a software development project from inception through to deployment, and a project will pass through a number of distinct phases. Whether it is the comprehension of what needs to be built, the formulation of an architectural vision, the writing of code, the running of tests, and so on, there is a question of how the activities should be played out over time, how they relate to one another, and how they relate to the planned, sequential phases of a project.

The manufacturing metaphor of software development has a certain simple attraction. Activities are strictly aligned with phases, so that comprehension of the problem domain is done during a phase dedicated to understanding the problem domain, coding is done during a phase of the project dedicated to writing code, and so on. Because each phase is carried out in sequence, each activity is also carried out in sequence, making the development a pipeline.

However, it is an idealized pipeline, where the output of one activity becomes the input to the next, without a feedback loop, and the pipeline is assumed to be non-lossy and free from interference. These assumptions are somewhat difficult to replicate in the real world. The pipeline is shielded — or rather, blinkered — from the business of dealing with clients, responding to change and clarification, or working with and learning from colleagues. Risks, and the consequent element of surprise they bring to a schedule, tend to stack up rather than drop down as development proceeds. There is no adequate mechanism for handling additional requirements, clarification of requirements, changes in staff or organizational structure, shifts in objectives, as well as technical issues and unknowns. The inevitability of any one of these automatically jeopardizes the assumptions that might make a pipeline a valid and viable solution.

Pipeline approaches are exemplified by the conventional view of the Waterfall Development Lifecycle, a term that conjures up a majestic image, albeit one that ends with water crashing against rocks with great force. The predicted and steady progression from *analysis*, through *design*, into *implementation*, *testing*, and then finally *deployment*, has an undeniable charm and simplicity. A pipeline project plan looks great on paper. It is easy to understand. It is easy to explain. It is easy to track. It is easy to fall under its spell. The fact that it may bear little relation to how people work or the actual progress of a project may seem somehow less important when caught in its thrall.

There is an implicit assumption that when a project deviates from its pipelined schedule it is due to some concrete aspect of the development — the developers, the management, the optimism of the schedule, external factors — rather than a fault in the underlying development metaphor. The reaction is often to declare that "we'll do it right next time" and then throw all hands — and more — to the pump to push through the delivery — more staff, more money, more time, less testing, less attention to detail, less application of best practice. The innate human response is that lost time can be made up for, that continual requirements clarification and change is an external factor not intrinsic to the nature of software development, and that making the development process more formal "next time" will address the shortcomings of the present.

For it to be of practical use, a development macroprocess should give more than the illusion of order. However, the notion of making something ordered by strictly pigeonholing one activity to one phase is not a well-measured response. A development process should also actively seek to reduce the risks inherent in development rather than ignore or amplify them. Risk tends to accumulate quietly in the shadow of planned development pipelines, bursting at just the wrong moment — inevitably later rather than sooner.

Activities do not need to be aligned discretely and exclusively with phases. Instead, the lifecycle can be realigned so that the activities run throughout. This is not to say that the emphasis on each activity is identical and homogeneous throughout the development, just that each is not strictly compartmentalized. There is likely to be a stronger emphasis on understanding the problem domain early on in the development than later, and likewise a stronger emphasis on deployment and finalization towards the end that at the start, but these different concerns are not exclusive to the beginning and end.

Instead of aligning the development cycle with respect to a single deadline and deliverable at its end, structure it in terms of a series of smaller goals, each of which offers an opportunity for assessment and replanning. Each subgoal should be a clearly defined development increment. However, because usable functionality is not proportional to quantity of code, the completion of code artifacts (lines of code, classes, packages, layers) cannot meaningfully be used as a measure of progress. As indicators they are too introspective and are at best only weakly correlated with accessible functionality. Functionality, whether defined by usage scenarios or feature models, is a more visible indication of software completeness. A development increment defined in terms of functionality is a more identifiable concept for all stakeholders concerned, whether technical or non-technical, whereas code artifacts have meaning only for developers.

Breaking down development objectives into these smaller increments allows all stakeholders to make a more meaningful assessment of progress and provides more opportunities to refine or rearrange objectives for future increments. However, should increment deadlines be fixed or variable? In other words, should development on an increment stop when its initial deadline passes, regardless of the scope that has yet to be covered, or should the deadline be pushed back until the intended scope of the increment has been completed?

Spacing the increment deadlines regularly, whether one week or one month apart, establishes a rhythm that offers a useful indication of progress, answering the question of how much functionality can be covered in a fixed amount of time. The scope is therefore variable and functionality must be prioritized to indicate which features can be dropped if the time does not allow for all intended functionality to be completed. Fixing the scope rather than the time means that the deadline, and therefore the point of assessment or demonstration, is always unknown. It is difficult to schedule people, meetings, and subsequent development if dates are always the unknown. Any unexpected problems in the development of an increment can mean that the increment begins to drag on without apparent end, when it would perhaps be better cut short and subsequent iterations reconsidered. Therefore, fixed scope per increment decreases rather than increases the knowability of development progress, and is therefore a riskier option than establishing a steady pulse and measuring scope coverage to date.

The development of each increment has its own internal lifecycle that is made up of the activities that run through the whole development. The repetition of this mini-lifecycle leads to an iterative process that is repeated across the whole macro-lifecycle. Each iteration has an associated set-up and tear-down cost, but with regularity the effort involved in kick-off and increment finalization is reduced through familiarity.

Iterative and incremental development derisks the overall development process by distributing the risk over the whole lifecycle, rather than towards the end, and making the progress more visible. The difficult-to-obtain determinism of knowing all of the problem before all of the solution, of writing all of the code before all of the tests, and so on, is traded for scheduling determinism and a more certain and empirical exploration of the scope.

Example: Copy before Release

Coding in the presence of exceptions is a very different context to coding in their absence. Many comfortable assumptions about sequential code have the rug pulled from beneath them. Writing exception-safe code requires more thought than writing exception-unsafe code. It is possible, if care is not taken, that an object may be left in an indeterminate state and therefore, by implication, the whole program put into an inappropriate state.

When an exception is thrown it is important to leave the object from which it was thrown in a consistent and stable state:

- Ideally any intermediate changes should be rolled back.
- Alternatively the object may be left in a partial state that is still meaningful.
- At the very least the object should be marked and detected as being in a bad way.

The consequence of not taking responsive and responsible action can lead to instability. This is both unfortunate and ironic: exception handling is supposed to achieve quite the opposite!

Exception safety can be attained via one of three paths:

- *Exception-aware code*: Code may be scaffolded explicitly with exception handling constructs to ensure that a restabilizing action is taken in the event of an exception, e.g. a `finally` block in Java or C#.
- *Exception-neutral code*: Code can be written to work in the presence of exceptions, but does not require any explicit exception-handling apparatus to do so, e.g. EXECUTE-AROUND OBJECT in C++ [Henney2000a] or EXECUTE-AROUND METHOD in Smalltalk [Beck1997] or Java [Henney2001].
- *Exception-free code*: Guaranteeing the absence of exceptions automatically buys code exception safety.

Exception-aware code is tempting because it is explicit, and therefore has the appeal of safety by diligence. It makes it look as if measures are being taken to deal with exceptions. However, such explicit policing of the flow is not always the best mechanism for keeping the peace. Exception-free code sounds ideal, but this path makes sense only when there is either no change of state to be made or the change is in some way trivial, e.g. assignment to an `int` or setting a pointer to null. A forced and uncritical attempt to cleanse code of exceptions leads to control flow that is far more complex than it would be with exceptions — a lot of C code bears witness to the twisted logic and opaque flow involved in playing return-code football. In some senses, forcing exception freedom on code leads to the same growth in supporting logic as the "be vigilant, behave" exception-aware doctrine. As with Goldilocks, when she decided to crash the Three Bears' residence and then crash on their beds, we find that exception-aware code can be too hard, exception-free code can be too soft, but exception-neutral code can be just right.

Of the three paths, it is exception-neutral code that is most often the path of enlightenment. Consider the following C++ example. The code fragment sketches a HANDLE–BODY [Coplien1992, Gamma+1995] arrangement for a value-object type, a class that among other features supports assignment:

```
class handle
{
public:
    ~handle();
    handle &operator=(const handle &);
    ....
private:
    class representation
    {
        ....             // actual implementation detail of class instances
    };
    representation *body; // pointer to the actual instance representation
};
```

The memory for the body must be deallocated at the end of the handle's lifetime in its destructor, either by replacing the plain pointer with a suitable smart-pointer type, such as `std::auto_ptr`, or explicitly, as follows:

```
handle::~handle()
{
    delete body;
}
```

The default semantics for the assignment operator are shallow copying, whereas a handle-to-handle assignment needs to deep copy the body. The traditional model for writing an assignment operator is found in the ORTHODOX CANONICAL CLASS FORM [Coplien1992]:

```
handle &handle::operator=(const handle &rhs)
{
    if(this != &rhs)                         // guard against self-assignment
    {
        delete body;                         // dispose of old body
        body = new representation(*rhs.body); // copy dereferenced RHS body
    }
    return *this;
}
```

An initial inspection suggests that this implementation has addressed the core needs of correct assignment: there is a check to prevent corruption in the event of self-assignment; the previous state is deallocated; new state is allocated; `*this` is returned as in any normal assignment operator. The construction of the nested body is assumed to be self-contained, but what if the `representation` constructor were to throw an exception? Or if `new` were to fail and throw `std::bad_alloc`? Here be dragons!

A failed creation will result in an exception: control will leave the function, leaving `body` pointing to a deleted object. This stale pointer cannot then be dereferenced safely: the object is unstable and unsafe. When the handle object itself is destroyed — for instance, automatically at the end of a block — the attempt to `delete body` will likely wreak havoc in the application and let slip the dogs of core.

It is tempting to construct a `try catch` block to guard against exception instability. This exception-aware approach normally leads to quite elaborate and intricate code; the solution often looks worse than the problem:

```
handle &handle::operator=(const handle &rhs)
{
    if(this != &rhs)
    {
        representation *old_body = body;
        try
        {
            body = new representation(*rhs.body);
            delete old_body;
        }
        catch(...)           // catch all exceptions
        {
            body = old_body; // rollback to pre-assignment state
            throw;           // rethrow current exception
        }
    }
    return *this;
}
```

It is worth pausing a moment to consider the rococo splendor of this code [Hoare1980]:

There are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other is to make it so complicated that there are no *obvious* deficiencies.

Nonetheless, in the spite of the ornate opaqueness, exception awareness for safety is still a strongly tempting path to follow. Such a solution reflects a way of thinking about the problem that is too close to the problem: a haphazard solution grown by grafting onto the original exception-unsafe code. A far simpler and exception-neutral solution is found in careful ordering of actions, so that the object's state passes through valid intermediate forms, stable no matter what rude interruption befalls the object:

```
handle &handle::operator=(const handle &rhs)
{
    representation *old_body = body;
    body = new representation(*rhs.body);
    delete old_body;
    return *this;
}
```

The essential flow expressed in the COPY BEFORE RELEASE C++ idiom [Henney1998] can be characterized simply:

1. Remember the old state.
2. Copy the new state.
3. Release the old state.

The same sequence of actions can be expressed using another idiomatic C++ form that relies on the handle's copy constructor:

```
handle::handle(const handle &other)
  : body(new representation(*other.body))
{
}
```

And on a non-throwing `swap` function, which allows two objects to exchange their state without the possibility of generating an exception [Sutter2000]. The copy of the object takes the role of an EXECUTE-AROUND OBJECT:

```
handle &handle::operator=(const handle &rhs)
{
    handle copy(rhs);
    std::swap(body, copy.body); // use standard library swap on pointers
    return *this;
}
```

The assignment operator takes a copy of the right-hand side. This may throw an exception, but if it does the current object is unaffected. The representation of the current object and the copy are exchanged. The assigned object now has its new state and the copy now has custody of its previous state, which is cleaned up on destruction at the end of the function.

The general form of COPY BEFORE RELEASE can be found repeated across other idioms for exception safety: checkpoint the current state; perform state-changing actions that could raise exceptions; commit the changes; perform any necessary clean up.

Problem

Change typically involves risk. Any change from one state of affairs to another that cannot be characterized as atomic inevitably involves a number of steps, any one of which could fail. The effect of successful completion is understood, as is the effect of failure at the start, but what of partial failure? Partial failure can lead to instability and further descent into total failure.

The most direct path from one state to another is most often the one with the strongest appeal and the greatest appearance of efficiency. What needs to be done is clear, allowing a single-minded focus on the path of change free of other distractions. All other things being equal, it will be the shortest and most direct path. However, when all other things are not equal the simple view supported by a direct change becomes a simplistic view lacking in care and foresight. The goal of minimizing overheads trades priorities with the safety net of contingency planning and derisking.

Focusing on one goal to the exclusion of others can weaken rather than strengthen control over risk, undermining rather than underpinning certainty. The polymath Herbert Simon coined the term *satisfice* to describe a form of behavioral satisfaction and sufficiency that accommodates multiple goals [Wikipedia]:

> In economics, satisficing is behaviour which attempts to achieve at least some minimum level of a particular variable, but which does not strive to achieve its maximum possible value. The most common application of the concept in economics is in the behavioural theory of the firm, which, unlike traditional accounts, postulates that producers treat profit not as a goal to be maximized, but as a constraint. Under these theories, although at least a critical level of profit must be achieved by firms, thereafter priority is attached to the attainment of other goals.

However, it is not necessarily the case that there is more control or predictability with respect to change simply because progress does not become the sole concern of a change of state. The wrong variables can be taken into account or an overcautious stance can lead to wasted effort.

In an effort to combat the uncertainty of the future, overdesign is a common response [Gibson2000]:

> That which is overdesigned, too highly specific, anticipates outcome; the anticipation of outcome guarantees, if not failure, the absence of grace.

Thus, a total and explicit approach to design or planning can become a measure that is overvalued and overused to the exclusion of other quantities and qualities that are potentially more valuable and balancing.

A big-bang approach to change proposes a discontinuous change from one state to another, essentially an abrupt shift in observable phenomena. Although it is easy to identify the point of change, there is significant risk if the change does not go according to plan. By contrast an approach that is chaotic rather than catastrophic emphasizes concurrency of change rather than a single point of change. However, as with free-threaded approaches to concurrency in software, change becomes difficult to trace, track, and reason about. A more bounded and constrained approach to concurrency compartmentalizes this uncertainty.

Solution

Ensure that each intermediate step in the process of change represents a coherent state of play. The risk and consequences of catastrophic failure are mitigated by ensuring that each mini-change making up the intended change is itself a well-defined and stable step that represents a smaller risk. The result of any failure will be a fall back to a coherent state rather than a chaotic unknown from which little or nothing can be recovered. Each step is perceived as atomic and each intermediate state as stable.

This sequencing of STABLE INTERMEDIATE FORMS underpins rock climbing, exception safety, and many successful software development strategies and tactics, as well as other disciplines of thought and movement, for example, T'ai Chi. The term itself is taken from the following observation by Herbert Simon (as quoted by Grady Booch [Booch1994]):

> Complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not.

The implication of stable intermediate forms is that change is realized as a gradual, observable, and risk-averse process — secure the current situation; prepare for a small change; commit to it; repeat as necessary. Not only does it break change down into more than a single, sudden step, but the steps in between have a stability and natural balance of their own. They are visible and discrete, which means that they can be used and measured without necessarily implementing further change. But although discrete, there is enough continuity to ensure that change can not only be paused, it can also take a new direction, even a reversal, without requiring disproportionate effort to replan and put into effect.

Failure does not unduly upset the system or require disproportionate recovery effort to restore or move it to a sensible state should the step to the next stable form falter. A missed handhold need not result in a fall. A thrown exception need not corrupt the state of the throwing object, nor does it require disproportionate effort to rollback — rollback happens by default.

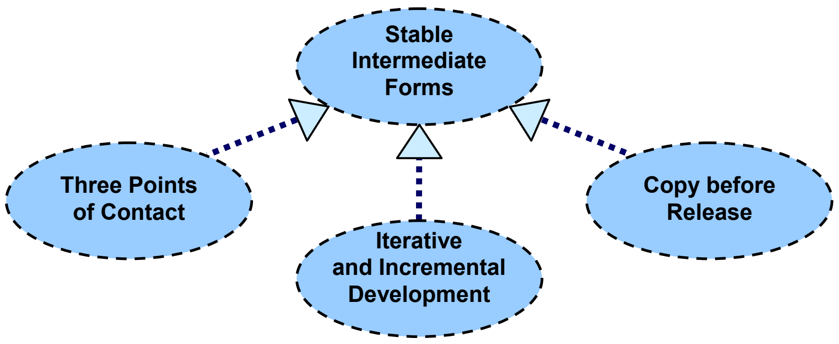
However, it is not necessarily the case that *stability* should simply be seen as a synonym for *good* without understanding its potential liabilities and context of applicability. There is a cautiousness, thoroughness, and deliberation about STABLE INTERMEDIATE FORMS that is not necessarily appropriate for all systems and objectives. Over a short enough timeframe the progression of small steps may be slower than taking a single leap and, more importantly, slower than is needed to achieve the end goal. For example, with at least one foot on the ground at any one time, walking is an inherently stable and leisurely activity. Running, on the other hand, is inherently unstable, trading raw speed for stability and effort. If speed is the goal a hundred meters is better covered with a run than a walk.

Ascending a familiar route on a climbing wall is derisked with respect to both experience — it is familiar, and therefore not an unknown — and environment — an indoor climbing center has limitations and facilities that make it a more controlled and comfortable experience than being stuck on a real rock face in the middle of nowhere in bad weather. Other sources of risk mitigation, such as a no-throw exception guarantee or familiarity with a particular climb, can moderate the need for further derisking through STABLE INTERMEDIATE FORMS.

On the other hand, where a prime objective is to embrace rather than mitigate risk — the thrill of a hard climb, the adrenalin rush of a sprint, the intellectual high of tackling a hard problem by immersion in detail, caffeine, and the small hours — STABLE INTERMEDIATE FORMS "spoils the fun" and may be seen as inappropriate. Of course, if this prime objective is not shared by others, a failure to progress through stable intermediate forms may be perceived as subjective fancy rather than objective sensibility.

Discussion

As a pattern, STABLE INTERMEDIATE FORMS may be considered general and abstract, a high-level — even meta-level — embodiment of a principle that recurs in other more concrete, domain-specific patterns where the context of applicability is explicitly and intentionally narrowed to address more specific forces, solution structures, and consequences. The three examples in this paper are realizations of STABLE INTERMEDIATE FORMS:


Stable Intermediate Forms
Three Points of Contact
Iterative and Incremental Development
Copy before Release

Evolving through stable intermediate forms to move from one state to another, allowing adjustment and adaptation en route, can be similar to a hill-climbing approach, where the overall goal is to reach the brow but each step is taken based on local terrain. However, with hill climbing it is possible to get caught in suboptimal local maxima. The ITERATIVE AND INCREMENTAL DEVELOPMENT of a system will ensure that its architecture addresses the needs to date, but it may find itself unable to meet a new need without significant reworking. STABLE INTERMEDIATE FORMS also allows a steady descent from local peaks, as well as the risk of ascending them in the first place. A clear vision not only of the ultimate goal but also of the general lay of the land ahead can help to mitigate the risk of getting into such situations — for example, a shared notion of a stable baseline architecture and the use of prototyping support architectural stability and evolution.

To stabilize an intermediate form is, in many respects, an act of completion. To put something into a finalized state, such as finishing a development increment, involves more effort in the short term than simply continuing without such a reflective and visible checkpoint. For development that is well-bounded in scope, occurs in a well-known domain, and is carried out with an experienced and successful team, a non-checkpointed strategy may sometimes be seen as appropriate, but otherwise measures must be taken to avoid letting other development variables and realities interfere, either by inserting STABLE INTERMEDIATE FORMS or by shoring up other variables through other guidance and feedback mechanisms, whether formal or informal.

STABLE INTERMEDIATE FORMS offers a path for change that offers similar consequences for open-ended and continued development as well as for more carefully scoped and goal-driven objectives. This applies as much to software development, for example the evolution of typical Open Source Software [Raymond2000], as it does to other domains, such as building construction [Brand1994].

In moving from a state that is inherently complete to another state there is also the implication that there is some rework involved, so that not all of the effort invested will be used in making externally visible progress, the remainder being used in revision or considered redundant. In some cases this can represent overhead and overcautiousness, particularly where the cost–benefit of a more reticent and punctuated approach is not immediately clear. In others the return on investment in effort comes over time. Revision is not always an overhead: sometimes it is a necessary part of the creative process. The body renews itself over time to ensure both growth and the removal of damage. In STABLE INTERMEDIATE FORMS, stepping from one state to another allows consolidation as well as a simple change of state. In refactoring the same process of removing development friction over time can be seen [Fowler1999]:

> **refactoring** (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
>
> **refactor** (verb): to restructure software by applying a series of refactorings without changing the observable behavior of the software.

To clarify, it is the functional behavior that remains unchanged with respect to a refactoring: the operational behavior, e.g. performance or resource usage, may well change. Refactoring needs to be informed by other practices that provide confidence that a change is indeed stable and defect free, e.g. refactoring as an integral practice in Test-Driven Development (TDD) [Beck2003] or as complemented by some form of code review, whether a simple desk check with a colleague or a full walkthrough in a larger meeting.

Considering a test-driven microprocess as a variant of ITERATIVE AND INCREMENTAL DEVELOPMENT raises the question of step size. How large should the span between one intermediate form and another be? In the case of TDD the granularity of change is visible to an individual developer over a minute-to-hour timeframe, for continuous integration stable changes are visible to a development team over an hour-to-day timeframe, and for NAMED STABLE BASES [Coplien+2005] in an ITERATIVE AND INCREMENTAL DEVELOPMENT the visibility of change to all stakeholders occurs on the order of a day-to-week timeframe.

Class invariants present a non-process example of STABLE INTERMEDIATE FORMS where the process of change is taken over an object's lifetime and the step size is defined in terms of public methods on an object [Meyer1997]:

> An invariant for a class C is a set of assertions that every instance of C will satisfy at all "stable" times. Stable times are those in which the instance is in an observable state: On instance creation... [and] before and after every [external] call... to a routine....

A method may be too fine grained to establish a simple invariant simply [Henney2003], in which case a larger step size is needed. Granular methods that are used together can be merged into a COMBINED METHOD [Henney2000b] so that the footfalls between observable stable forms are further apart.

In situations where there is no need for revision, the balancing efforts required to support STABLE INTERMEDIATE FORMS may seem superfluous. Repeatedly achieving temporary stability may seem to incur additional effort and redundancy that has reduced return on investment. The context determines whether such an approach is genuinely wasteful or not. For example, in COPY BEFORE RELEASE a replacement representation object is always created and the current one discarded even in the event of self assignment. Although the self assignment is safe, the extra allocation–deallocation cycle might be seen as overhead. However, self assignment is not a typical mode of use, so this cost tends toward zero.

Acknowledgments

I would like to thank James Noble for his patience and shepherding of this paper for EuroPLoP 2004, Klaus Marquardt for being a party to this and a foil for the humor and foot dragging of both shepherd and sheep, and the members of the EuroPLoP workshop for their feedback: Paris Avgeriou, Frank Buschmann, Kasper von Gunten, Arno Haase, Wolfgang Herzner, Asa MacWilliams, Juha Pärssinen, Symeon Retalis, Andreas Rüping, Aimilia Tzanavari, and Dimitrios Vogiatzis.

The first notes on this pattern were scribbled during a presentation at a patterns seminar in 1997. I would like to thank Alan O'Callaghan for the session that provided the inspiration, but I would also like to apologize for then paying less attention to his slides than I did to my scribbles.

References

[Brand1994] Stewart Brand, *How Buildings Learn*, Phoenix, 1994.

[Booch1994] Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2nd edition, Addison-Wesley, 1994.

[Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

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

[Fowler1999] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Gibson2000] William Gibson, *All Tomorrow's Parties*, Penguin Books, 2000.

[Henney1998] Kevlin Henney, "Creating Stable Assignments", *C++ Report* 10(6), June 1998, `http://www.curbralan.com`.

[Henney2000a] Kevlin Henney, "C++ Patterns: Executing Around Sequences", *EuroPLoP 2000*, July 2000, `http://www.curbralan.com`.

[Henney2000b] Kevlin Henney, "A Tale of Two Patterns", *Java Report* 5(12), December 2000, `http://www.curbralan.com`.

[Henney2001] Kevlin Henney, "Another Tale of Two Patterns", *Java Report* 6(3), March 2001, `http://www.curbralan.com`.

[Hoare1980] Charles Anthony Richard Hoare, "The Emperor's Old Clothes", *Turing Award Lecture*, 1980.

[Meyer1997] Bertrand Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.

[Raymond2000] Eric S Raymond, *The Cathedral and the Bazaar*, O'Reilly, 2000, `http://www.tuxedo.org/~esr/writings/cathedral-bazaar/`.

[Sutter2000] Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000.

[Wikipedia] `http://en.wikipedia.org/wiki/Satisficing`, definition of *satisficing*, June 2000.