EPISODES:
A Pattern Language of Competitive Development
Part I

Ward Cunningham, ward@c2.com
IBM Consulting Group

Submitted to the Second International Conference on
Pattern Languages of Programs
Monticello, Illinois, 6-8 September, 1995

This pattern language describes a form of software development appropriate for an entrepreneurial organization. We assume the entrepreneur to work in a small team of bright and highly motivated people. We also assume time to market is highly valued as it often is where market windows close quickly and development dollars are in short supply. But, unlike some entrepreneurs, we also place high value in being able to get a second version out the door in a timely way; and a third version; and an Nth version; many years down the road. That is, we expect to be successful and have every intention of exploiting that success by continuing development for as long as our customer has desires.

These patterns describe how to develop software. They could be fairly described as *process* patterns though they don't actually describe a process the way a methodology document might. Nor do they describe *designs* or *organizations* a other patterns have. Being patterns, they do describe *things*, things that solve problems that occur in the process. The things can be physical like a document or meeting. Or they can be mental like a commitment or state of mind.

We are particularly interested in the sequence of mental states that lead to important decisions. We call the sequence an *episode*. An episode builds toward a climax where the decision is made. Before the decision, we find facts, share opinions, build concentration and generally prepare for an event that cannot be *known* in advance. After the climax, the decision is known, but the episode continues. In the tail of an episode we act on our decision, promulgate it, follow it through to its consequences. We also leave a trace of the episode behind in its products. It is from this trace that we must often pick up the pieces of thought in some future episode.

We won't be so naive as to suggest that the thoughts leading to a decision be written down. These thoughts are too complex and decisions too numerous for this to be practical. What we do suggest is that hints and pointers be placed in strategic locations so that preparation for subsequent episodes might go more smoothly. Of course they won't. That's because each episode to touch a given area does so with more expectation. We only hope to rise to the occasion. We will know that we have done so if our episodes remain well shaped: not too heavy on the front or the back, and not always getting longer.

There is an old saying that laments, *there is never time to do it right but always time to do it over*. We take this to be a fact of competitive life. We find ourselves unable under competitive pressure to make the kind of careful decisions we would like. These patterns tell what decisions can be made, in fact should be made, to maintain continuous forward motion through itterative development.

One does not have to compete to find these patterns useful. The developments they create are equally applicable for entrepreneurial groups within large organizations, or any other group that wants to develop code quickly and indefinitely.

The language addresses a wide variety of development issues. These have been organized into topic areas that could be described as top-down or chronological. Don't think that any real development is so structured or sequenced. In practice, these patterns will be applied over and over, in or out of order, sometimes by people whose job description says they should do so, and sometimes not. Chart 1. presents a map of the language with patterns positioned by task and agent. Here *task* implies the kind of work being done while *agent* implies the kind of person doing it. These aren't to be taken too seriously. Far more important are the relationships between patterns. Patterns are related when one leads to another. This happens when the strong forces bearing on a pattern are brought into balance its a solution. Such resolutions of strong forces inevitably expose weaker forces to which attention should then be applied. It is this shift of attention that we capture in the

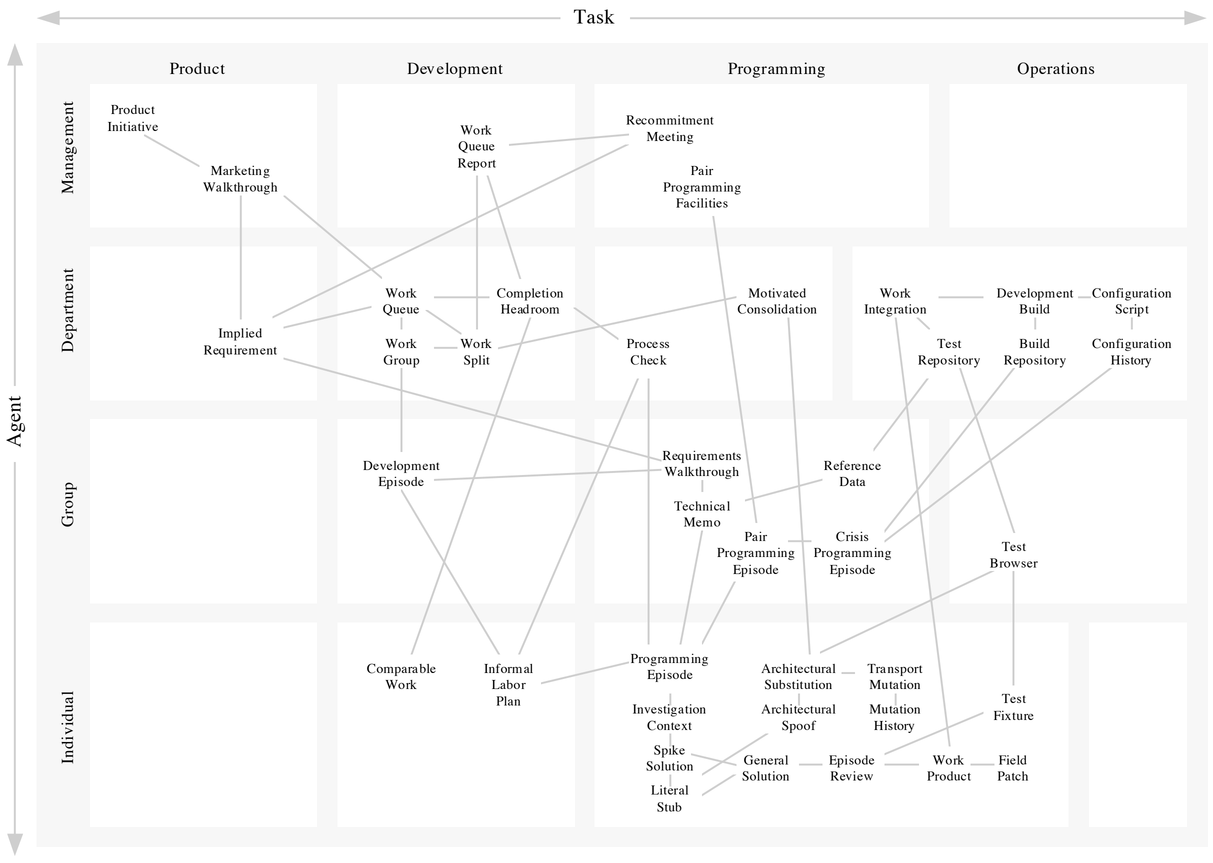

Figure 1. Map of EPISODE *patterns and their relations.*

relations between patterns. These relations turn a bunch of patterns into a pattern language; a system that, like a natural language, is employed without a great deal of conscious thought.

1. Product

We pick up the development process somewhere in the long middle, after the first few versions have shipped and well before the customers have lost interest in future enhancements. We start with the longest and largest of nested episodes, the Product Initiative.

1.1 Product Initiative

Market Conditions or Operating Conditions may indicate the need for increased or modified product functionality. Or, an in place Product Plan may call for specific functionality on a given date. Anyway, it is time to direct attention of multiple groups toward specific product goals.

Everyone involved with a product accumulates a wish-list of features and functions they would add given half a chance. A product designed to absorb an ever increasing complement of features is at particular risk of losing any sense of coherence or direction.

Therefore: Articulate clearly the most important direction for improvement of a product. Expect all members of all involved groups to be able to at least summarize the initiative and defend its business rationale. A new initiative need not align with previous initiatives. There is no intention to retreat on past accomplishments by withdrawing them from the product or withholding proper support. There is, however, every intention of reducing or displacing less than relevant thoughts in every mind within the initiative.

Expect an initiative to come from upper management where responsibility for major resource allocation rests. It is possible for a product initiative to boil up from the development ranks. What is important is that everyone accept the initiative as the direction of the moment, no matter how such agreement may be reached.

A clear initiative will Imply Requirements which should be talked through carefully in a Marketing Walkthrough. Definite schedule and resource goals further enhance the focus of an initiative. Throughout the initiative, management should track the Completion Windows of Work Groups on each requirement. Initiative slips result in over allocation of attention to one subject at the expense of good will and future initiatives. In the worst case Implied Requirements may need to be reviewed and possibly deferred or discarded in an organization wide Recommitment Meeting.

1.2 Market Walkthrough

A Product Initiative will be expressed in market or business terms. A product is more than a program or any other piece of technology. Davidow makes this clear in his book *Marketing High-Technology*. It is a marketing function that makes one from the other. And marketing must have good contact with both sides of its operations, the product's customers and the program's developers. Likewise, development must understand the customer needs served by an initiative and have the confidence and resources to peruse market questions as they arise.

Therefore: Begin every initiative with a walkthrough of program and product concepts involving most of the development and marketing staffs. Understand an initiative from the buyer's and user's perspective and from developments point of view too. Should an initiative come from or involve contract terms, now is a good time to review them. Finally, all should agree on basic terminology, such as that used as Implied Requirements.

Example:

A trading software company is responding to growing fear of derivative contracts by adding improved pricing models and related analytic. The marketing department has selected key customers with derivative portfolios and a willingness to work with development. In a market walkthrough the company president outlines changes in the derivatives market, the New York region customer representative summarizes the newest pricing models popular on "the street", and the staff domain specialist outlines a vision for incorporating similar function in the companies product. The walkthrough ends with a long question and answer period in which marketing and development begin to match customer needs with implementation possibilities.

1.3 Implied Requirement

A Product Initiative has identified the direction for further development and a Marketing Walkthrough has explored the customer motivation and developmental possibilities behind it. We expect positions and attitudes to be understood but have yet to make any commitments beyond everyone's general commitment to do a good job by the company.

A commitment implies an agreement between people. Development commitments generally obligate developers to meet some customer need in a timely and satisfactory way. The tension here is to define a need in sufficient detail that commitments have meaning without exhausting up-front analysis or over constraining a solution.

Therefore: Select and name chunks of functionality. Use names that would have meaning to customers consistent with the Product Initiative. Allow these names to imply customer requirements without actually enumerating requirements in the traditional sense.

Examples:

Year-End Tax Reports
Dollar Denominated Japanese Bonds
High-Quality Printing
Disconnected Operation on Lap-Tops

These names will fill in the blank in the recurring questions like: Who's handling the programming (or specification, or customer contact, or manual update, or release notes) for _____.

2. Development

The following patterns generate development team activity leading to frequent and regular releases of increasingly functional programs. An important idea is the simultaneous development of requirements, specification, design and implementation. Where these responsibilities fall to single individuals, they are assumed to be able to wear the *appropriate hat at the appropriate time. Similarly, should different people fill each role, they are assumed to be able to coordinate their activities such that each is productive and benefits from the other's work.*

The patterns also generate a schedule of sorts while preserving considerable latitude to do what makes the most sense at the moment. Our general strategy is to develop to a few fixed target dates. When a delivery date arrives, we would like to look back over the development period and say with confidence that we used every available minute wisely.

2.1 Work Queue

Implied Requirements suggest deliverable program enhancements which will have various necessities, dependencies, risks and rewards. Deliverables may be ill-defined being represented more by a vision or desire than anything concrete or measurable.

If we were to work up a conventional schedule we would probably begin with a block of requirements analysis for each item. From these would be hung blocks of specification, design, implementation and eventually integration and testing. Add to this some wild guesses and a few ordering constraints and, presto, thirty feet of diagram saying what will be finished when and by whom. Such a document takes on a life of its own striking fear in developer's hearts and generally distracting everyone else from the real scheduling task which is to get better input, not larger output.

Therefore: Produce a schedule with less output than you have input. Use the list of Implied Requirements (really just names) as a starting point and order them into a likely implementation order favoring the more urgent or higher priority items. When work can be factored from two or more entries, go ahead and do so giving the common element a name that establishes its worth and implies its implementation precedence.

Example:

1. Settlement-Date Positions
2. Settlement-Date Based Tax Reports
3 Trade vs. Settlement Accounting Preference by Portfolio

Be prepared to reorder this list as unforeseen interactions surface or business realities demand new priorities. Remove work from the list as it is completed. Observed defects is not enough to return completed work to the list. However, independently scheduled repair activity may uncover omissions that are more appropriately removed from defect tracking and scheduled in competition with all of the other work on the Work Queue.

2.2 Work Group

We have a Work Queue that describes Product Initiative relevant work, is ordered by urgency, and that shifts up as completed work is removed from near the top. We must now allocate staff without overwhelming or under-utilizing any individual.

Therefore: Allocate staff to roughly two month's worth of work at the head of the Work Queue. Seek their commitment to work together as needed to understand the real and Implied Requirements, develop suitable specifications, complete or extend a design, assemble tests and implement all aspects of the deliverable. Expect individuals to apply themselves to their most urgent assigned work. Allow some latitude to compensate for impossible to forecast dependencies between projects and individuals, Expect the work of any given item to be performed with the usual rise and fall of concentration revolving around a burst of decision making that marks the center of a Development Episode.

2.3 Work Queue Report

With the usual mix of analysts, designers and implementers, one can assume that a proportional amount of analysis, design and implementation gets done every week. This can be confirmed in a weekly status meeting where every attendee is given five minutes to describe his or her specialty. However, it can be amazingly difficult to detect a slipping schedule in the status meeting venue.

Therefore: Collect status in regular personal interviews conducted at weekly intervals. Solicit days of remaining effort estimates using contrasts with Comparable Work.

Example:

"I put two full days into the new tax calculations, and one day with Joe on his U/I."

"How many uninterrupted days do you think you need to finish the calculations?"

"Oh, say two, It's no different from the accruals."

"And, working with Joe?"

"Well, we didn't get to the real work. I had three down last week? Must still be three days."

Use these estimates along with individual dilution factors (how many uninterrupted days of development does the individual have access to a week) to predict elapsed days to completion for each assigned deliverable. Compute and publish Completion Headroom from this data. Include a cover page with a few sentences explaining numbers that might have shifted in an interesting way.

2.4 Comparable Work

Developers are surprisingly bad estimators when it comes to dates. On the other hand, they have a good memory of what circumstances lead to what problems on just about every project they ever worked on and have a sixth sense for the same

Example Work Queue Report:

Work in Progress	*Uninterrupted Days to Completion*						*Earliest*	*Head*
(ordered by priority)	*Sue*	*Bill*	*Joe*	*Kay*	*Ed*	*Ray*	*Possible*	*Room*
availability:	*60%*	*70%*	*25%*	*70%*	*10%*	*75%*	*Finish*	
Settlement-Date Positions		3	2				June 5	12
Settlement-Date Tax Reports		2				2	June 8	9
Trade vs. Settle Preferences			1	4			June 7	10
etc.	5				5		June 21	8
etc.				9		3	June 25	4

circumstances in a new project.

Therefore: Let developers estimate effort by selecting comparable work. A job that is 2/3 as complex as some previous job will probably take about 2/3 as long. Comparable estimates are usually accurate even for ill-defined projects unless there is hidden complexity not taken into account when selecting comparable. Hidden complexity usually shows within a few days of actually starting work. It's OK to challenge an estimate that is not taken seriously but don't try to hold developers to last week's estimate when they've uncovered hidden complexity. Take heart, there is such a thing as hidden simplicity that does surface on occasion.

As an aid to memory, record uninterrupted days applied to current efforts as was done in Figure 3. This data will be a handy reference when today's projects become tomorrow's comparable. Do not expect a week on the job to yield more than two or three full days of development. Also, don't try to use this data for performance evaluation. To do so will destroy the frank relationship required for good estimating. Besides, it's not clear whether bigger or smaller numbers indicate improved performance. It will be necessary to prorate accumulated effort data should a project undergo a Work Split. Some ratio will suggest itself. Just don't count days twice.

2.5 Completion Headroom

Every project must commit to delivery on a few hard and fast dates. This is actually fortunate because it is about the only way to get out of work that is going poorly. A Work Split provides the graceful exit by allowing one to defer the portion of work that is not understood or going poorly while saving the part that does work or will save face. A Work Split does require some advance notice since some portion of the work must still be completed before deadline.

Therefore: Project Work Group completion dates from remaining effort estimates in the Work Queue Report. Take the largest of the earliest completion dates for each work group and compare it to any hard delivery date that may apply. The difference is your Completion Headroom. Headroom will often jitter plus or minus a day or two from week to week. But steady evaporation of headroom for any Work Group is a sure indicator for management attention. You have at your disposal reordering the Work Queue, possibly deferring whole items to later release, the Work Split already mentioned, or the public embarrassment of a Recommitment Meeting.

2.6 Development Episode

Members of a Work Group have been selected based on needs inferred from the Implied Requirement. Each member brings specific skills which will be important at some point in the development. For this we can be thankful. However, if we overemphasize a member’s specific strength, we diminish everyone's general abilities, unnecessarily narrow the members focus to applying just that specialty, risk creating ambiguity as to who is responsible for non-specialized tasks, and discourage the learning of new skills.

Therefore: Approach all development as a group activity as if no one had anything else to do. Expect the activity to follow the usual course of an episode where energy builds to a decision-making climax and then dissipates. At the height of the episode, purpose should be clear, terminology well understood, knowns well explored and unknowns identified. It is at exactly this point that individual strengths merge into a sort of common consciousness. Landmark decisions come easy. Breakthroughs are common. A creative act will have been shared.

Besides yielding better decisions, the collective episode has very positive effects on the participants. Looking back, people often have trouble identifying the actual source of key ideas. Non-specialists gain invaluable insight into the thought processes of the specialist. A specialty is demystified, shared, spread throughout the group. A master of a specialty will realize that this sharing will not diminish one's own status within the group. As insight wells up in the master, he will delay slightly, expecting others to be close to the same insight, and knowing that their actual recognition experience will be of tremendous value to them and a small loss to himself. Seymour Papert called this an "Ah Ha" and admonished instructors not to "Steal the Ah Ha" (Mindstorms).

2.7 Informal Labor Plan

The Development Episode presents an ideal that must be worked into the lives of people trying to get a big job done quickly. Developers will often find themselves obligated to more than one in-progress Development Episode at a time. The Work Queue offers one prioritization, though one that ignores the many small tradeoffs possible when the work is at hand.

Therefore: Let individuals devise their own short-term plans. Accept that much of the group activity implied in a Development Episode will take place pair-wise between group members that find the time to tackle some issue together. Avoid the temptation to call a meeting where a developmental climax is intended to happen. It won't. Instead let individuals express interests and make commitments to each other. And let them revise these intentions on a moment's notice when the energy of some episode reaches an irresistible level.

A Development Episode is actually composed of a series of Programming Episodes, some of which must take place in (at least) pairs if any approximation of group consciousness is to form. An individual's labor plan is his tool to make these connections happen. Pair Programming Facilities are configurations of the physical environment that can reduce this planning to an occasional promise in the hallway.

2.8 Work Split

A Work Group commits to resolve and deliver Implied Requirements in the most timely and satisfactory way they can find. They are not committed to specific dates. But they do have an obligation to make their efforts visible through what becomes the ultimate trouble signal, low Completion Headroom. Headroom disappears when developmental activities fail to match those of Comparable Work. A common problem is the well-meaning escalation of requirements by people too close to a problem.

Therefore: Divide a task into an urgent and deferred component such that no more than half of the developmental work is in the urgent half. Defer more if required to acquire sufficient Completion Headroom. Defer analysis and design of parts that won’t be implemented. This advise runs counter to conventional wisdom. Often a split is just a way to get back to the basic work that had been originally planned. Trust Architectural Substitution to cover for omissions and inconveniences caused by incomplete "up-front" work. Both halves of the split will appear in the Work Queue with distinctly different urgency.

2.9 Recommitment Meeting

Work Splits offer a mechanism to keep to a schedule that can be initiated from within a development group. If a Product Initiative is in jeprody because Implied Requirements cannot be met through schedule and Work Queue adjustments, then it is unlikely any other development initiated activity will help. Management up to at least the level that began the initiative will suddenly take interest in all circumstances leading up to the current situation. Some of this is natural and appropriate. But it won't be a time of high productivity and shouldn't be allowed to continue too long.

Therefore: Assemble a meeting of interested management and key development people. Allow the meeting to review history until all present agree simple adjustments (like working weekends, or adding staff) won't help. Eventually a solution appears, usually expressed as a question of the form: What is the least amount of work required to do X? X is one person's idea of the most important part of the initiative. The question should be answered quickly and confidently by consulting a recent Work Queue Report. The process may repeat for plans Y and Z. Ultimately a plan will be selected. Then the remainder of the meeting is devoted to talking through implications of the decision and getting all parties commitment to the new plan and/or schedule.

This, of course, is another form of episode. The decisions are ones of allocating business resources and belong in upper management. However, all present can contribute, and should do so in a frank, honest, non-defensive and constructive way.

3. Programming

Programming is the act of making and encoding decisions about future behavior. Encoding requires the careful consideration of the basis and consequence of every decision. Often decisions are found incomplete and new questions raised. In this section we consider decision making in the presence of incomplete, obscure or questionable facts. We include patterns for the assembling of knowledge in artifacts and individuals, and for limiting decision making when knowledge requirements exceed that immediately available.

3.1 Requirement Walkthrough

Not all members of a Work Group will start to consider the Implied Requirements of a piece of work at the same time. Unpredictable circumstances lead individuals to work through their portions of the Work Queue at different rates. Although any members efforts should be considered a contribution, the first to the problem may seem to others to have inappropriate influence over decisions effecting others.

Therefore: Assemble the whole Work Group as soon as one member begins to consider any part of the Implied Requirement. Consider the expressed needs and desires, the individuals who hold them, and likely strategies to meet them. This is the beginning of the Development Episode and is a good time to sketch the first Informal Work Plan. This can lead to adjustments in group membership. It is also a good place for CRC level design.

3.2 Technical Memo

A Development Episode may intertwine with other activities demanding the attention of the Work Group. Further, some concepts may require quite contemplation to absorb or may involve sufficient detail that they cannot be recalled without aid.

Therefore: Maintain a series of page-printable technical memoranda addressing subjects not easily expressed in the program under development. Focus each memo on a single subject and keep the text short and to the point. Carefully selected and well written memos can easily substitute for traditional comprehensive design documentation. The latter rarely shines except in isolated spots. Elevate those spots to technical memos and forget about the rest.

3.3 Reference Data

The Requirement Walkthrough will identify relevant information sources which will be retrieved, reviewed and absorbed as the Development Episode begins. The various data may require transformations before they are easily interpretable. Such activities build the awareness that makes for the intensity characteristic of a decision making climax. However, after a climax, the focus is on the recent decisions and their implications; the data and processing that contributed to the insights are easily forgotten.

Therefore: Collect examples, test cases and customer data as machine readable examples. Use a spread-sheet program to organize and transform the data as appropriate. Leave notes and observations as text annotations to the sheets so that key observations won't be forgotten. Keep this handy throughout development and make sure it is easily imported into the development environment when useful. Check the program against this data throughout development, possibly by incorporating it in Test Suites developed for the application.

3.4 Programming Episode

Programming is the act of deciding now what will happen in the future. A programming language offers an operationally precise way to encode decisions through a process called simply *coding*. Programmers reason about future behavior by interpreting previously coded decisions and integrating these with their own decisions and their interpretations of other sources like Technical Memos and domain experts. The depth, quality and value of programming decisions will be limited by the programmers ability to concentrate.

Therefore: Develop a program in discrete episodes. Select appropriate deliverables for an episode and commit sufficient mind share to deliver them. Be aware of the rise in concentration as the episode progresses. Consider each source (above) and consciously include or exclude its recommendations.

Use the fear that often accompanies a decision not-yet-made as a motivation. Try to compare your position within an episode to similar points in previously successful episodes.

Example:

"I feel like we've been around twice now on the possible ways we can bind the six terms of this bond analytic to the four calculation classes we have in our library."

"Yea, right now I'd be happy if we could place the four primary terms, look at the error cases, and see if that gives us a hint how to proceed after lunch."

Push for the decisions that can be made. Don't abandon an episode; that will leave you feeling defeated and unable to achieve even the same level of concentration at a future time. Make the decisions that seem possible. Code the decisions. Then review the code to be sure that the extent of your decisions and your confidence in them is apparent in the code. Coding occurs on the down-hill side of a programming episode. Coding is the most direct way to promulgate programming decisions.

Appendix A: Supplemental Patterns in Brief

Space does not allow for the inclusion of all EPISODE patterns. This table lists patterns that will be included in part II of the EPISODES pattern language

Pattern	*Solution*
Pair Programming Episode	Add reflective articulation, subliminal Process Check, pattern propagation, search space pruning and general good will to the Programming Episode.
Pair Programming Facilities	Arrange the furniture. Adjust the fonts. Stretch the cables. Use a private office with an open door and two chairs. Abandon authorship/ownership.
Crisis Programming Episode	Assemble a possibly larger than necessary Work Group. Let members drop off as a solution becomes clear.
Investigation Context	Use inspection and cross-referencing tools to assemble a mental model of the as-is program
Spike Solution	Patch, debug and/or code a minimal solution of representative function in the task. Code protocol before variables using Literal Stubs to make progress.
Literal Stub	Return the value that you know you will eventually compute in the current Investigation Context.
Generalized Solution	Code for skipped/missing parts of the solution. Stub or UninterpretablyImplement still remaining function.
Process Check	What are you working on? Why are you working on that? Don't work on that anymore.
Architectural Substitution	Create a new architecture for function that does not fit well on existing architecture. Develop that architecture in isolation if necessary to understand mechanisms required by the new function.
Architectural Spoof	Add sufficient compatibility protocol to an obsolete architecture and/or it's substitution so that both can coexist. Reuse names so that Modification Lists will include both architectures.
Motivated Consolidation	Clean only that part of the architecture necessary to support a motivated (funded) set of functionality.
Episode Review	Examine the Episode Product from a solution-sharing rather than a problem-solving perspective.

Pattern	*Solution*
Work Product	Create a transmittable, integrateable summary of changes necessary to duplicate the solution.
Field Patch	Bundle the Work Product in a form that it can be efficiently distributed to field locations in an emergency. Integrate this patches as you would any other Work Product. Patches can induce Sweep Mutations, but these must be considered part of one continuous development. Where truly one-off modifications are required, use a subclass and script it out of standard configurations.
Sweep Mutation	Sweeping is the process of traversing clusters of application data as it is exported from or imported to the runtime memory environment. Expect to find any possible version of any object in the external world. Mutate these to the most current form as they are read. Write the most current form as new or modified objects are written.
Mutation History	Sweep Mutation requires that we remain familiar with all previous versions of any given object. Keep that history in an "evaluatable comment" in the mutating sweep method of the effected classes.
Work Integration	Assemble recent Work Products. Check for completeness and conflicts. Edit as appropriate.
Developmental Build	Reassemble an image from complete sources. Run regression tests and save results in a distributible image. Post this image to the Build Repository with the Work Products uniquely included. Mechanically assemble Episode Products. Supervise conflict resolution.
Build Repository	Collect Developmental Builds with auxiliary information used to construct and test them. Cull non-released builds when they are covered by a release and all open forks have been closed.
Test Suite Repository	Collect tests from Reference Data, Programming Episodes and normal test development. Distribute tests to Programming Episodes and Developmental Builds. Preserve and protect as if it were code. Make flexibility.
Test Suite Browser	Drill down from build statistics to suites to cases to variables to inspectors and debuggers. Import, enter, and compute expected values. Visualize failure distributions (systematic vs. sporadic).

Pattern	*Solution*
Test Fixture	Construct or retrieve objects suitable for performing a test. If a test case is an interrogation, then the fixture is the interrogator who calls up the interrogated and starts asking questions. Long term fixture maintenance is a development responsibility. Make sure the fixtures appear in a Development Episode's Investigation Context.
Configuration Script	Develop in a full configuration. Deploy by applying scripts that remove or disable unwanted function.
Configuration History	Accumulate a history of Configuration Scripts and Field Patches applied to a particular product. Make sure the history is available in Crisis Programming Episodes so that debuggable versions of faulty configurations can be reconstructed.