Capable, Productive and Satisfied

Patterns for Productivity

Paul Taylor,
Senior Consultant,
Simsion Bowles & Associates,
1 Collins Street, Melbourne, 3000, Australia.
http://www.sba.com.au
prt@sba.com.au

About Productivity

The greatest asset of a software development organisation is the productive, cohesive team. Quality software, architectures and systems can only be developed when productive, cohesive teams operate. This pattern language addresses team capability and productivity—the ways a capability to produce and evolve software can be created and maintained in a healthy state.

These patterns are primarily about people, process, and understanding, rather than products, production and artefact. Their solutions resolve forces to create more productive software development environments by drawing on products, production and artefacts.

The "growing alternative" metaphor (Brooks 95) strongly underlies these patterns; software is *grown* by a group of people who share a culture of understanding of the problem space, the domain, the system requirements, the implementation technology, and a shared perception of the potential futures of the system. A productive development team therefore *represents a development capability* which should be regarded as of *greater value to the business than architectures or other software artefacts*, which can be evolved while team activity is maintained.

About these Patterns

The following eleven patterns describe the establishment and maintenance of a team. Two patterns, *BOOTSTRAPPING* and *COHESIVE TEAM* address establishing a productive team. Four patterns, *ROUND PEGS FOR ROUND HOLES*, *PROJECT PULSE*, *DELIVERABLES TO GO*, and *PRODUCTION POTENTIAL* address the managing and scheduling of an established team. The remaining patterns, *SHOW AND TELL*, *PRODUCTIVITY GAMES*, *TEAM SPACE*, *EFFECTIVE HANDOVER* and *ARRANGING THE FURNITURE* address maintaining a healthy level of productivity as the project delivers.

A number of excellent organisational pattern languages have been written (for example, (Coplien 95)). If any distinction can be made, it is that the following patterns take a 'capability based' view of a development team, and address ongoing management of the capability in a business context. In Eric Raymond's terms (Raymond 98), they pragmatically recognise the reality of cathedral style development of medium and large business applications and systems, and address some of the inherent inflexibilities and shortcomings. Several patterns introduce 'bazaar' qualities in small doses.

These patterns are by no means complete; rather, they describe elements of the author's experience and observation. Alexander writes *"good patterns are good because to some extent each one of them reaches the quality without a name itself"*. Patterns themselves, like the systems in which they are embedded and repair, must be allowed to evolve in a piecemeal fashion. These patterns are offered primarily for their face value, but also as an incremental contribution to the piecemeal growth of organisational pattern literature.

Some Patterns for Establishing Productivity

1. BOOTSTRAPPING

A pattern for initiating productive work from a newly formed team. Also known as *PRIMING THE PUMPS*, *TEAM CONSTRUCTION*.

Context

You have a recently formed development team, where most team members are new to the problem domain, the application, and each other.

Gearing up for the creation and manipulation of intellectual property differs from physical production—the simple act of getting started presents new challenges.

Problem

Your newly formed team must become productive *quickly*.

Forces

An unproductive team costs the same to run as a productive one—the length of the initial phase of low productivity must be minimised.

An unproductive team is an unstable team, and is more easily driven apart by external influences.

Leadership is much more difficult before team dynamics, individual abilities, interests and roles have been established.

There is a danger that people will start building solutions before they understand the problem.

Solution

In the early days of a team's existence, people cope with their disorientation by looking for firm ground upon which to anchor their nascent understandings. If not responsibly guided, people run the risk of diverging, spending too much time on unnecessary detail, or looking for eye-catching decorations when a firm foundation is required.

The standard approach of allocating tasks and work areas to individuals addresses the mechanics of project planning *but not team formation*, which is best fostered by interaction and group problem solving. A useful approach is to focus on responsibilities—what has to be *known*, and what has to be *done*.

Therefore, *partition* some responsibilities, and *share* others.

Your developers should understand this approach, since a responsibility driven allocation of work reinforces one of the basic principles of object-orientation, although care should be taken not to overdo the analogy.

In order to partition responsibilities well, you must recognise that some responsibilities are best owned by individuals with specific capability and experience. You will need to draw on your experience in order to spot these responsibilities, and then your experience of your team members to correctly allocate them (this is where *ROUND PEGS FOR ROUND HOLES* comes in). Here are some candidate responsibilities which you might choose to *partition*:

- *abstractionist* roles—senior designers who seed the business model and software architecture;
- *business domain expert* roles—business representatives who produce appropriately detailed domain models, investigate critical domain issues, and connect with information sources;
- *development environment* roles—senior developers who understand software tool and build environments;
- *user liaison* roles—users, possibly outside the core development team, who translate business requirements to software architecture.

Not all responsibilities should be partitioned. There are many responsibilities of team membership which must be evenly shared. The mechanical (but important) tasks in building software must be shouldered by the team rather than by a few individuals. Here are some candidate responsibilities which you might choose to *share*:

- everyone must have an understanding of the system's high level architecture;
- everyone must have an understanding of the system's business or technical context;
- all developers must be fluent in the development language and basic development environment; for example, everyone must converge on naming, coding and layout conventions;
- all developers must be comfortable with source version control, debugging techniques, and software build processes.

Resulting Context

By partitioning and sharing development responsibilities with your team, you achieve a degree of understanding, openness and trust which can never be achieved by simply hanging names on a Gantt chart. Your people will additionally benefit from your openness and clarity in these assignments. They should also be reminded that the assignments are not fixed indefinitely.

From an external management perspective, a rapidly bootstrapped team will be perceived as more cost-effective than one which bootstraps more slowly or haphazardly.

Example

A newly formed team of twelve developers was bootstrapped by *partitioning* two abstractionists, two business subject experts, and two tool-smiths. All developers *shared* design and review process adherence, coding practices, version control and personal build practices, and responsibility for learning and using an object-oriented database product. The individuals quickly adopted these responsibilities. After three months, the abstractionists transitioned into coding. In retrospect, the object database presented a significant and costly learning curve for all team members, which could have been softened by earlier partitioning some responsibilities for its efficient use.

Known Problems

If shareable responsibilities are narrowly partitioned, the individual will suffer from being excluded from the team's intellectual culture. It is a far worse mistake to share responsibilities which require critical skills or a singularly focussed approach, such as object-oriented domain analysis or requirements modeling—the results of inappropriate assignments on specialist tasks may haunt the project for some considerable time.

Many teams have a star performer, the brilliant and personable individual who seems like the ideal candidate for every partitioned responsibility. Resist the temptation to overload this individual, firstly for his or her sake, and secondly because you need to give other ‘rising stars’ the chance to grow into expertise and leadership.

Some responsibilities may fall somewhere between being partitionable and shareable. Partition these, and observe them carefully as the *PROJECT PULSE* begins to beat.

Related Patterns

BOOTSTRAPPING addresses the initialisation of a new team, rather than the incremental replacement of members in an established team. *BOOTSTRAPPING* sets the context for a *COHESIVE TEAM* to operate. Team turnover is addressed by *EFFECTIVE HANDOVER*. Initial *BOOTSTRAPPING* initiates the first phase in the *PROJECT PULSE*.

2. COHESIVE TEAM

A pattern for encouraging a team culture focussed on the problem space. Also known as *SELF-MOTIVATING TEAM, PULLING THE SAME ROPE, CONSTRAINT DISCOVERY.*

Context

You have an assigned development team, after *BOOTSTRAPPING*.

In the realm of physical work, team cohesion can be enforced by applying process standardisation and precision. Where the work is on intellectual property, team cohesion involves aligning people's motivations and interests.

Problem

You must now convert individual capabilities into a team productivity.

Forces

Teams consist of individuals, each with their own interests and agendas.

Intra-team working relationships must be allowed to establish and settle before people feel and become productive.

Software projects, particularly young ones, present a maze of possible activities and paths, many of which are unproductive, futile, or harmful.

Establishment of constraints within a project takes time and effort.

Solution

Your team's ramp-up period may be viewed as a time of *constraint discovery*, when people discover the constraints in the problem domain, the requirements, and the software architecture, which will shape the delivered solutions. Learning constraints goes hand in hand with learning the nature of the problem. Once the problem space and its constraints have been probed, a foundation has been formed upon which design and construction can commence.

The problem space is only half the picture—the *team* forms a space of its own, held together by relationships, navigated by communication, and constrained by the member's capabilities and attitudes. While each individual discovers the relevant problem space constraints in his or her own way, the interaction tests out the team space.

Team *cohesion* is about balancing these dependencies and individualities. Although a balance will always be achieved after time, elapsed unproductive time must be avoided. The catalyst to speed this reaction must be a common motivation, the lowest common denominator. Experience suggests that *the imperative to fully understand the problem* is such a catalyst.

Therefore, stimulate an interest in the *problem space*, and use it as the common motivation to build team cohesion.

Focussing the team on the problem space gives each member an appreciation of their part, a taste of the project's vocabulary, and a base of confidence to aid communication. Team members who, for whatever reason, never grasp the problem being solved by the software system often lack vision and commitment to go beyond short term tasks—they often want to be told what to do. While not everyone needs to lead, everyone needs to be able to find motivation.

To focus a team on the problem space, you can:

- have the system's potential users visit and educate the team,
- create some non technical tasks which involve both users an developers,
- use *excursions*, so that team members can experience the system's business context,
- tie architectural decisions back to the problem space during architecture and design reviews.

Learning the problem space is a means to an end (a *COHESIVE TEAM*) not an end in itself. The activity should be resourced accordingly.

Resulting Context

Individual team members with an understanding of the problem can better understand the reasons for their tasks and work orientation, and are well placed to make the right decisions in designing their own components of the solution. A collective understanding of the problem facilitates better and more accurate team communication. A *COHESIVE TEAM* is a team of people who work together to solve different but related parts of the same problem.

Example

A team of twelve developers was educated in the business domain using a range of the activities described above. A project sponsor (in this case more of a 'father figure'—a long serving business expert one year off retirement) paid the team regular visits to discuss progress and answer questions about the business domain. This growing knowledge informed frequent discussions about requirements which helped to clarify ambiguity and motivate people.

Known Problems

Some problem spaces are highly complex, and cannot be mastered by all team members in a short time. In this case, choose a critical but manageable part of the problem space.

Related Patterns

An appreciation of the problem is strengthened by *DELIVERABLES TO GO*, which brings developers and users together around deliverables.

Some Patterns for Managing Productivity

3. ROUND PEGS FOR ROUND HOLES

A pattern for effectively assigning tasks to people. Also known *as KNOW YOUR PEOPLE*.

Context

You must allocate tasks appropriately amongst team members during iterative software development.

Problem

By assigning a task to the wrong person, you can waste time, get inferior results, and discourage both the individual and the team.

Forces

Developers are not interchangeable.

Not all developers like, or are good at doing the same basic tasks.

Different business situations require different work approaches.

A complex toy can expose children's inability to organise themselves. The presence of an adult who knows the children well enough to appropriately assign roles to individuals can make the game run smoothly.

People are most productive when they are able to do what they do best.

People don't always recognise what tasks they are best at.

People need new challenges.

Solution

Experienced managers get to know their team members well enough to allocate tasks appropriately. Software developers differ widely—some like to work quickly to provide fast turnaround on small tasks, while others prefer to take more time and deliver quality and thoroughness. These are personality characteristics—you cannot determine them from resumes or interviews. Individual's capabilities and preferred work modes must be observed over time in their current team context.

Therefore, discover each individual's preferences for working, and allocate tasks accordingly.

This solution uses *observation*, then *action*. First, you must observe how your people work. Then you must apply these observations in your decisions about "who does what" in the mix of team tasks. You cannot short-cut this process of discovery, but you can hasten it by trying non-critical task allocations to test people's abilities and interests.

Take the demonstration of a product's user interface as an example. User interfaces are often demonstrated by the lead GUI developer, although better *presenters* may be available elsewhere in the team. A demonstration to users requires a smooth and focused run through the business tasks, rather than an deep knowledge of how the code is structured. By rotating the task around different team members, the right demonstrator can be determined for the prototypes' different audiences.

The "preferences for working" that you should learn are orthogonal to preferences for technologies and software layers. People with different work preferences are needed in all of the system's architectural layers and technologies. A looming software deadline illustrates this point. The known bugs which inhibit release are typically classified into those requiring simple (but tedious) fixes, to the really difficult ones which require complex testing with debugging processes or tools. This workscape demands not only different technical abilities, but different personal work modes; some bugs need a quick resolution which will hold up in 80% of cases, while others require a methodical approach and thorough application of the fix—an informed selection of the individual to do these tasks is critical. You can speed up the acquisition of this knowledge by trying different bug allocations to individuals as you approach the deadline.

Resulting Context

Knowledge of individual's working preferences pays off in several ways. Obviously, you can make better assignments of tasks to people. As well, team members will appreciate the personal attention evident in their assignments, and the breadth and depth of their skills may increase.

Example

In a project team of fifteen members, the project and technical managers worked together to trial task assignments or team members. They had periodic discussions about their observations of individual's preferences. One team member worked to the 80/20 rule, delivering 80% of functionality very quickly, but ignoring the outlying cases and execution paths. Others took more time but delivered code which rarely failed. These observations were used when project planning, and the team quickly settled into a productive rhythm.

Known Problems

Observations must be made over time, in the heat of battle, as individuals learn, advance and make ground, small steps at a time. Your observation work must not become 'micro managing' or stand-over supervision—remember to let people know what you are doing, and periodically discuss how assignments are going.

The knowledge you gain of individuals is valuable, and must *only* be used to make informed decisions about task allocation; other uses risk contravening the working relationship between you and your team.

Related Patterns

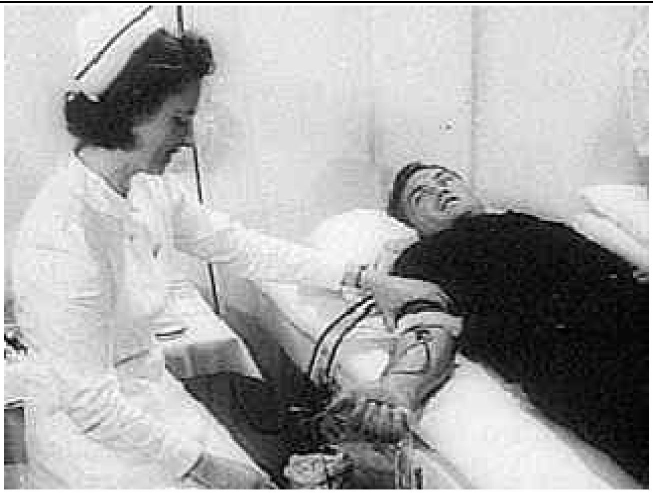
Times of lower *PROJECT PULSE* rates may be best for testing out an individual's performance on a new task.

4. PROJECT PULSE

A pattern for managing the peaks and troughs of team activity to ensure a high average level of productivity in the long term. Also known as *PACING YOURSELF*.

Context

Your team must develop and deliver a significant amount of the project's functionality over the coming months.



Pulse rate is the basic indicator of the body's work rate; it directly correlates with physical and emotional exertion as a result of external demands. Exercise increases the body's pulse to a predetermined level for a short period of more intense work. It then relaxes to its normal level.

Problem

Your team is capable of bursts of higher productivity to meet production demands, such as a software release—but you must handle the timing, length and frequency of these peaks with great care.

Forces

Increasing the delivery pressure on a team results in a short term productivity increase.

Teams cannot maintain peak productivity rates indefinitely.

Teams without delivery pressure do not produce to their capability.

Solution

Teams are usually exercised in a cycle of increased and decreased work rates dictated by project schedules and deliverables. Over time, the period of the cycle will emerge; once the team's work rate is observably cycling, the rhythm can be used to everyone's advantage.

Therefore, determine the team's *delivery rhythm* by putting the team through periods of higher pressure release cycles.

To gain control of the project pulse, you need to determine the optimum pressure and recuperation periods, then establish a project rhythm of production peaks and troughs, carefully aligned with the project plans and deliverables.

All projects experience external pressure to maintain high productivity for long periods. You can break long periods of high pulse rate by inserting space and time between two or more *real* releases:

- rotate team members through activities which are amenable to changing personnel, such as software support, testing, or training;
- provide additional support for pressured individuals within the work environment;
- provide support outside the project environment where appropriate, such as child-care, carefully managed time away, or other assistance.

Alternatively, there may at times be long periods of lower productivity, due to changing project goals, or dependencies on other groups or products. Avoid risking loss of project rhythm by breaking long recuperation cycles with an artificially created release (a *manufactured* pressure cycle), or with useful development work such as class and architecture generalisation, framework development, or work on other nice-to-have deliverables for consumption by the project team.

Resulting Context

Regular and managed productivity peaks allow team members to plan and anticipate work peaks, which can help in managing life's other commitments. In the reverse direction, deliverables can be planned to coincide with the team's delivery peaks, so that the team exerts pressure back on the business, shaping its future schedules and the timing of deliverables.

Example

On a software team delivering successive releases of a telecommunications application, the project cycle was determined after about a year's work with a stable team of eight core developers. Cycle peaks and troughs of eight to ten weeks were alternated in the project's work life. Productivity peaked close to the end of high periods, and increased at each successive cycle. The rhythm was finally broken after system delivery, when two key team members left the project.

You may have been unfortunate enough to have experienced the classic counter examples of this pattern. A team permanently pressured by impossible deadlines, moving targets or under-resourcing with its project pulse rate permanently just below seizure level is untenable—people will leave this situation, either physically or metaphorically. The counter example is the un-pressured team with no plans, no deadlines, and no direction—people will leave this team for different reasons. Both extremes are unhealthy, unsatisfying and ultimately unproductive.

Known Problems

Project pulse is more evident when the team and its focus are stable; adding or removing team members can significantly affect the team's delivery capability. When the team line-up changes, the rhythm must be re-established. Ideally, people should change at times of low project pulse.

Projects have dependents (users being an obvious example) who need to be considered when planning a peak in project pulse.

Related Patterns

A project's pulse rate can be steadily increased after *BOOTSTRAPPING*, and once a *COHESIVE TEAM* has formed. Peaks in pulse rate typically produce *DELIVERABLES TO GO*. Levels of productivity should be assessed by *PRODUCTION POTENTIAL* rather than the team's perceived activity levels. Periods of recuperation are best for an *EFFECTIVE HANDOVER*.

5. DELIVERABLES TO GO

A pattern for ensuring the team's deliverables are used. Also known as *CONNECT SOURCES TO SINKS*.

Context

Your team maintains (or will shortly maintain) released software.

Problem

You must scope and time your team's deliverables so that their recipients will use them.

Forces

Developers want their work to be appreciated.

Developers want the extent of their solutions to be explored and tested.

Some things are best consumed here and now. When this is not possible, they lose value.

Users receive the best support from developers when the work is fresh in the minds of both parties.

Unused or unwanted deliverables de-motivate developers.

Unused deliverables open the door to a lower standard of quality.

Requests for new functionality which are not responded to carefully and in reasonable time-frames de-motivate users.

Solution

Synchronising the timing and expectations between the development team and user community is critical to getting good results from both.

Therefore, only release when users are ready to consume.

This pattern commends an agreement between developers and users which works both ways. Firstly, developers must treat user acceptance and requests for new or additional functionality seriously. Secondly, users must recognise their responsibility to explore the deliverables they have requested via the system requirements process. This can only happen when developers have access to users, and can begin to integrate users into parts of the team culture. You can stimulate this integration by:

- inviting user involvement in team activities (for example, business or domain experts can be involved in reviews);
- encouraging the involvement of certain developers in user or business activities, such as business processes, tests and trials, system or prototype evaluations.

When this agreement matures, it should be possible for the development team to delay a planned deliverable until users are ready for it (with their agreement), without fear of recrimination or embarrassment.

Resulting Context

The relationship between developers and users should be a *truce*—users should exert pressure on developers to motivate their work and maintain quality—developers should exert pressure on users to test and explore their product. A healthy resolution of these forces increases communication and builds trust at the team's critical personal interfaces.

Example

This pattern is most readily illustrated by considering its absence. Most developers will be recall a software release which was delivered for no apparent reason. One example concerns a middleware development team which worked long hours to complete a first release. Management set a date in response to external pressure, but did not communicate its subsequent deferral. The release was delivered to a desk in the early hours of the final day, where it sat for the next month, a silent reminder of the futility of the exercise.

Related Patterns

Large releases typically get delivered at the end of a peak in *PROJECT PULSE* rate, and are followed by a carefully managed recuperation. A *PRODUCTION POTENTIAL* measure at the time of release should show full effort focused on the deliverable.

6. PRODUCTION POTENTIAL

A pattern for making a fair assessment of team progress. Also known as *TRANSPARENT PROGRESS, DESIGN VELOCITY*.

Context

You have a development team deeply involved in iterative development.

Problem

You observe a great deal of activity and communication, but little apparent progress on the software. You must translate this team activity into an accurate and fair assessment of team progress.

Forces

Team progress cannot be solely assessed from the state of the source files in the project directories.

Teams often make significant progress without touching source files.

Leaps in progress sometimes result in removal of source code, or seemingly trivial software changes.

In industry, the factory, plant and infrastructure are a tangible indicator of a capability to produce. In software development, the intellectual infrastructure exists in intangible forms: the experience and knowledge of the team, its key people, and their determination to deliver.

Solution

Where teams are concerned, a leap of progress may look like a free-fall from a great height to some observers. Teams make progress by learning, and learning is iterative. It can therefore be a mistake to judge progress by the state of a team's deliverables, particularly early in the project. Learning often results in the throwing away of software artefacts—classes, designs or micro-architectures—but these discards represent progress in the team's understanding of the problem, and a growing confidence in the mechanisms which will be used as parts of the solution. Conversely, a design shortcoming discovered late in the project may represent a significant backward step, which is not reflected in the software or artefacts on disk.

Therefore, measure team progress by a weighted mix of assessed progress in all spheres of work.

At any point in time, a team's progress is a function of the following quantities:

- the software architecture and modules, represented in source code;
- the design and architecture models and documentation (although these often lose some currency as indicators of progress late in the project);
- the collective understanding and degree of confidence of the team members;
- the collective degree of comfort with the solution of the team members.

The first two measures of progress are visible, while that final two are not. Hidden progress of this kind must not be undervalued through lack of acknowledgment, or lack of recognition. Intellectual progress increases a team's problem solving *capability* or *potential energy*. This potential energy will be converted into the kinetic energy of software production at a time when all significant preconditions have been met.

In the mean time, you should look for progress on many fronts, and relate this progress to the project phase. You can weight your individual assessments to form an overall view of progress. It is then possible to consider where progress shortfalls are occurring.

Resulting Context

This pattern's message underlies the entire pattern language, so it deserves to stated loudly:

> *A software development team's most valuable attribute is its potential to produce. Production potential is a function of experience, knowledge, understanding, established roles, and a supporting environment.*

The patterns in this language interwork to generate productive teams by creating an environment where this capacity to produce is realised. Viewing progress in terms of the team's production potential helps to recognise the value of work done which contributes only indirectly to deliverables.

Example

If you have worked on a non-trivial object-oriented software system, you will have observed at least one period when the maturity of the source and the team's progress towards a solution strongly contradicted each other. The use of object-oriented metrics in management and progress reporting often highlights this disconnect.

A team nearing completion of a large middleware software product worked at full strength and full pace for 3 months on resolving bugs and completing final functionality. During this time, weekly bug reports indicated zero or negative progress—for each new bug fixed, another one or two were detected by the test team. While management requested clarification of the apparent absence of progress, the project and technical managers held the team "steady as she goes", protecting and preserving the people and their processes from reactionary change. During this time, three developers were quietly refactoring a significant class cluster. Their progress in locating the seat of the bugs, and redesigning the architecture, was invisible. Once complete, the system stabilised rapidly, and as the bugs became easier to locate and resolve the arrival rate plummeted, and the system shipped.

Known Problems

Developers can be difficult people to extract accurate progress information from. It takes time to learn the right questions to ask of each individual.

Related Patterns

A team's *PRODUCTION POTENTIAL* typically translates into high productivity at times of peak *PROJECT PULSE*. *SHOW AND TELL* sessions can be useful in extracting hidden progress. *ROUND PEGS FOR ROUND HOLES* may help you to learn what kinds of questions to ask individual team members.

Some Patterns for *Maintaining* Productivity

7. SHOW AND TELL

A pattern which facilitates communication of localised solutions between team members. Also known as *SOLUTION SHARING*.

Context

Your developers are designing and coding their own localised areas of the architecture and software.

Problem

You must eliminate the re-invention and wasted effort which results from general design and code solutions which are not communicated.

Forces

Isolation decreases quality by creating uninformed and inferior designs.

Isolation decreases productivity by increasing re-invention.

The time taken to refine a solution can be greatly reduced by jointly tackling elements of the problem at appropriate times.

Solutions and idioms are improved with communication.

Solution

The ability to develop regional or localised software structures and solutions depends on an understanding of the overall architecture. Solutions, such as proven design and code mechanisms, must become idiomatic in the team culture.

Refining and sharing solutions is a fundamental pattern in the process of software development, and has many faces. The face which most directly hits team productivity is the duplicated investigation of a problem or a bug, and the unnecessary re-discovery of a solution. Because software developers often have to maintain an intense focus on a highly complex and localised problem, they sometimes fail to lift their eyes above their partitions for long periods, and are prone to such duplication of effort.

Therefore, conduct a periodic discussion session where individual's problems and solutions are reviewed and shared.

Fortunately, the solution is simple—give team members a forum to discuss their solutions in a positive and supportive way. It is critical to make the 'telling' activity fail safe (failure must be acceptable), since it involves revealing one's solution, design, or code to the rest of the team. When others raise arguably better solutions, the session must be arbitrated to reach the best result for the project without crushing egos or letting suggestions, ideas or good work go unrecognised.

Resulting Context

A *SHOW AND TELL* forum reduces re-invention, and strengthens the shared solutions through review and discussion. The forum additionally provides a powerful peer education function through the sharing of experience. Finally, team communication is increased and the visibility of individual's work is raised.

Known Problems

More experienced team members must take a responsible attitude to *SHOW AND TELL* sessions by not dominating them. The sessions must not become a soap-box for preaching ideas, nor should it be allowed to stray off track. The importance of one design problem and its solution should not be over-emphasised over others. All team members should be equally encouraged to both review, and show and tell their own work.

Candidate solutions can be promoted to team idioms and practices before being thoroughly tested or understood, causing unnecessary rework when the full implications are understood.

A clever, appealing coding idiom may become fashionable, and may be over-used.

Example

Many organisations instantiate a form of this pattern as weekly or monthly lunchtime presentations to staff, by staff members. This variant addresses general technology education, whereas Show and Tell topics are highly team-specific and address narrow design, process or coding issues of interest to team members only.

SHOW AND TELL in some form is highly valuable during a period of debugging. A team working on a major software release found that once the obvious application bugs were located and removed, the next group of software bugs involved the application's use of an object database product, and much of the time spent in bug diagnosis involved learning nuances of the specific software release of the database. After several developers reported doing very similar lengthy investigations into specific database behaviours, a weekly *SHOW AND TELL* session was set up to discuss the database issues encountered during the week, and their solutions and work-arounds.

Related Patterns

Use this pattern for sharing the successful reification of design patterns to the system architecture and design. *SHOW AND TELL* sessions should peak in frequency with the *PROJECT PULSE*. *SHOW AND TELL* is one of the *PRODUCTIVITY GAMES* which your team may trial and adopt. The *TEAM SPACE* is the ideal location to host *SHOW AND TELL*.

8. PRODUCTIVITY GAMES

A pattern which recognises the contribution of certain types of 'play' to productivity. Also known as *GET A LIFE*.

Context

Your team is deeply involved in iterative development.

Problem

Individuals are so closely focussed on design and code that they are beginning to lose the shared culture of problem solving they developed earlier.

Forces

When a project hits its development peak, group problem solving gives way to individual problem solving.

Team problem solving skills learned in earlier work phases should not be neglected.

Game playing is a natural part of learning—it should be creatively included in adult learning and work.

Game playing encourages problem solving interaction on many levels.

Game playing reinforces shared values and team culture.

Solution

Teams develop a shared culture of communication over time as a result of collectively finding solutions in their area of productivity. Maximum productivity is not reached until a core of this culture is in place, so time to build this culture must be minimised. Once established, a team problem solving capability needs to be maintained. As team members rotate through the inwardly focussed activities of iterative development, it is easy for their group problem solving skills to atrophy.

Game playing is a fundamental technique in teaching concepts. People often take a different, more relaxed approach to solving game puzzles or logical problems, which can give rise to creative, lateral solutions which might otherwise not arise. Games also provide a continuing thread to entertain people as they do the hard work of software production.

Therefore, encourage people to *play games* with interesting and important concepts and ideas.

Productivity games are low priority, ongoing threads of discussion and thought with the following characteristics:

- they are inexpensive (in resources and time),
- they have a definite deliverable which can only be produced by interaction and group communication,
- the deliverable is relevant and useful to the project,
- the deliverable is achievable.

Worthwhile productivity games are opportunistic—they sometimes spin out of investigations into the business domain. For example, a team developing financial trading systems might decide to collectively design the best performing financial instrument. A team building network management software might design a network traversal algorithm to extract useful network performance heuristics. During times when people's heads need to be lifted from their screens for some basic interaction, simple productivity games (such as estimating when a bug count will drop below a certain level) can be enjoyable.

Productivity games should deliver a single solution which most of the team's members contribute to. While productivity games should produce something, the purpose of a game is interaction and maintenance of the problem solving culture—the actual deliverable is secondary.

Resulting Context

Productivity games, when used appropriately, help to maintain a healthy level of problem focussed team interaction. From this context, it becomes easier to rapidly change work activities and phases, as often happens during iterative development.

Known Problems

Taking time away from core development activities may seem like a luxury which projects cannot afford. In such cases, the productivity game may be taken out of work hours, or interworked within an existing task.

Example

One team turned code reviews into a productivity game. After doing extensive code reviews, they developed an elaborate culture to keep reviews happening after the initial burst of activity. Over time, the team owned, defined and regulated the entire process. All team members were involved in all review roles, any team member could call up any piece of code for review at any time, and team members could change established coding practices and idioms as a result of reviews. Another team used productivity games to maintain social interaction and judgement through a particularly long period of bug fixing. They took low-price bets on a diverse range of simple outcomes, such as the exact date and time a large database load would complete, and the number of function points which an independent team would count.

Related Patterns

PRODUCTIVITY GAMES act as a catalyst to speed the process of forming a *COHESIVE TEAM*. Productivity games should not be overlooked during *PROJECT PULSE* peaks, particularly when individuals are focussed on personal design and coding.

9. TEAM SPACE

A pattern of physical interaction which occurs when people are free to cycle through the phases of individual and collective problem solving. Also known as *TEAM LOUNGE, QUIET CORNER*.

Context

Your team is deeply involved in iterative development.

Problem

Your physical work environment should be arranged to maximise people's productive time.

Forces

People performing detailed, complex *thinking* work benefit from periodic short breaks from their work environment.

Developers need to feel that relaxing their concentration for short periods is acceptable behaviour.

Refreshment often flows from a change in physical surroundings.

If not provided, team members will find or create this space themselves.

People unintentionally distract others when their relaxation periods occur out of phase.

In the absence of a team space, teams may splinter into smaller cliques and allegiances, which meet in more inaccessible or private locations.

Solution

If productivity is to be maintained over time, a holistic, healthy work environment must recognise the natural rhythm of human work. It is widely recognised that productivity peaks interleave with non-productive periods during an average working day. When resting or reflecting between periods of higher productivity, developers need to change their surroundings, change their posture, and casually interact.

Therefore, create a physical space for casual, unplanned interactions between team members.

Your team space should be small enough to feel something like a domestic lounge room, but large enough to comfortably house the entire team without individuals feeling physically constrained. It should preferably be close to the team's work space, and sufficiently private so as to be inconspicuous to visitors. Its partitions, walls, and contents should be as moveable, replaceable, and as flexible as possible, to allow redefinition as the team size, its members, and its activities change.

This pattern differs from the others in this pattern language because its solution involves *physical* space, in which a social context welcomes people by allowing them to physically relax and mentally change gear. A social context can be created using couches, video equipment, posters, and icons collected during the team's ongoing life. Let the members decorate and arrange the space themselves.

Many of Alexander's architectural patterns for domestic living are applicable if the project has the option to design a space. For example, a team space should have *ALCOVES (179)* and might follow *STRUCTURE FOLLOWS SOCIAL SPACES (205)* (Alexander 1979) so that concurrent activities can occur, and so that longer term activities, such as ongoing discussions, diagrams, white boards, or games can be left undisturbed over days, weeks or even months.

Resulting Context

A team space exists for the team members, and should be owned by the team members. The team space can be used for social events, like welcoming a new member, or fare-welling a departing member. It can be used for discussing and making decisions where consensus is required.

Known Problems

Activities in the team space may become a distraction at times from pressing work. This may be addressed by taking the work into the team space for a period. Most team members will understand and respect the motivation for providing the facility.

A team space is not a marketplace, a cafeteria, or a sporting complex—rather, it is a space for the team only, and should be designed to be used briefly and frequently during periods of productive work. Company sponsored gymnasiums, health centers or sporting facilities are *not* examples of this pattern, because they typically inhibit work-related interactions, and usually demand a prolonged break from the team environment.

Examples

The team space need not be in the office. A team located in a conventional open plan office tower above a large retail complex had no suitable space, so they created their own by regularly visiting one of several coffee lounges in the retail complex. Visits were made at certain quiet periods during the day, and each visit was announced to all present team members. Work related matters were frequently discussed. The team's management encouraged this use of coffee lounges by attending.

Related Patterns

The team space makes an excellent venue for *SHOW AND TELL* sessions, and may also hold the context of some *PRODUCTIVITY GAMES*. Alexander's patterns for domestic spaces are directly relevant to the design of the physical team space.

10. EFFECTIVE HANDOVER

A pattern which resolves the risk of lost productivity when an experienced team member is replaced.

Context

Your team is running at peak productivity, or nearing a critical delivery phase.

Problem

A critical team member is departing, so you must minimise the lost productivity that will inevitably result.

Forces

The departing developer has the most detailed knowledge of a critical piece of the software system.

The replacement has development skills but no application, architecture or domain knowledge.

The departing developer has unfinished work which must be done before departure.

Most professions must cope with staff rotation and handovers. In health care, a rushed or inaccurate handover can cost a patient's life, while in manufacturing, the safe operation of a production process.

A full handover would tie up both developers for several weeks, killing the departing developer's productivity.

Solution

In software development, we do handovers notoriously badly. In development, the equivalent of a 'shift' is a very long period, sometimes several years, and the depth of contextual knowledge required before useful work can be done is considerable. Knowledge of the application, its business domain, and project history which account for critical architecture and technology decisions may have taken months or years to accrue. In most project situations, when a key developer leaves, it is nearly impossible for the departing developer to effect any significant degree of transfer of knowledge to the replacement.

Therefore, the departing developer should hand over to an existing developer, setting the replacement free to do *new* work.

It make much more sense to:

- start the new team member on solving a different problem,
- leave the departing developer as much uninterrupted time as possible to work on resolving unfinished work, bugs, documentation and incomplete work, and
- give an existing team member the task of picking up the departing developer's software.

An effective handover preserves the team's ability to work and support a large and complex software architecture by re-distributing responsibilities around the team's proven 'pillars'. By handing over to an experienced team member, the difficult task of picking up the departing expert's area now becomes an exercise in identifying what the continuing team member *additionally* needs to know. When compared to the base level of project expertise, individual expertise often shrinks to manageable proportions.

Resulting Context

This solution recognises that even the most expert team member builds his or her expertise on a common base of project experience, history and context. It is this context which takes the time to learn and assimilate. The resolution removes a demanding and typically unachievable requirement for the new team member to “fill the expert’s shoes”. Instead, the newcomer can learn at a more manageable pace and be free to migrate towards the areas and work activities which they would naturally choose. This gives the newcomer “freedom to succeed” in their own space.

Known Problems

The continuing developer may become overloaded. If this happens, use the departure as an opportunity to shuffle and re-allocate everyone’s tasks and responsibilities.

There may not be a suitable or significant piece of system behaviour ready for the newcomer to commence. There are many alternative tasks which a new team member can usefully do to learn a software system while making a useful contribution—code re-factoring, code reviewing, reverse engineering of design documentation, updating design or user documentation, testing and test scripting are all practical examples.

Related Patterns

Performing effective handovers when team members are replaced preserves the current *PROJECT PULSE* rate, as much as possible. The newcomer should immediately be assessed for work preferences (*ROUND PEGS FOR ROUND HOLES*). The team member picking up the departing developer’s design and code may spend some initial time *ARRANGING THE FURNITURE*.

11. ARRANGING THE FURNITURE

A pattern which aids the task of learning to inhabit (settling in to) someone else's software. Also known as *MARKING TERRITORY*, *HABITATION*.

Context

Your established team is entering a time of transition, as original members are replaced by newcomers.

Problem

Your newcomers must quickly come to grips with large and complex software modules.

Forces

People are 'territorial animals', and need a degree of ownership in order to be productive.

A new maintainer needs to gain confidence and familiarity with someone else's code before being productive.

Placement of furniture and selection of decorations allow us to impose personality on a space, building a sense of ownership.

For a software module to be *alive*, it must be actively inhabited on an almost daily basis.

Familiarity and confidence is achieved incrementally—in software, one edit at a time.

Many people only feel ownership after they have contributed to a product.

People develop confidence and familiarity more quickly when they see they have control over their environment.

Solution

People, like cats, can be territorial. Adult male cats spray on trees, bushes, fence posts and doors to mark their territory, and then only begin to relax after they have explored and understood their immediate surroundings. To some extent, software developers need to mark their intellectual territory before a personal feeling of ownership can be established.

Source code can be changed to increase familiarity without changing its performance or behaviour, and without violating project coding conventions. Many small changes can be made to enhance code familiarity in the eyes of the reader.

Therefore, an adopter should be encouraged to 'move in' by cosmetically arranging code.

Taking up residence in a physical space involves arranging the furniture, hanging some new pictures on the wall, and placing frequently used things in convenient locations. Software equivalents include comments, documentation approaches, and names. Settling in to someone else's source affords the opportunity to converge it towards project standards, which may involve anything from simple layout re-arrangement (inter-symbol spacings, control block justifications, and the like) to minor rewriting.

To settle into a new software module, you may choose to:

- correct departures from project coding guidelines and conventions,
- add comments, to express your understanding of the source in your own words,
- add tracing, so that you will be able to follow control flow when the source executes,
- add assertions within methods to test your understanding of how class attributes are used,
- add defensive code, to guard your module against bad arguments,
- change (locally scoped) names or other symbols which are ambiguous or not meaningful.

ARRANGING THE FURNITURE is intended to be an incremental activity. As each new ‘room’ is visited, explored and understood, the explorer leaves the room subtly re-arranged, so that it is more understandable and recognisable when subsequently re-visited. It is important that this task remains a background one, and does not become a systematic pass through dozens of source files, unless of course the state of the software dictates this action.

Resulting Context

ARRANGING THE FURNITURE is about cosmetic change, not structural or functional change. When you are very familiar with a software module, you may begin re-structuring (breaking up long methods or eliminating code or attribute duplication), and re-factoring (changing class boundaries and relationships to evolve the abstractions, design and architecture).

Known Problems

ARRANGING THE FURNITURE must not be used as an excuse to “trash the backyard”. Habitation affords the freedom to change style, not structure or long established project norms. The software architecture, class and module interfaces, and project coding standards must not be broken.

New graduates, full of idealism and undisciplined energy, may need their ‘furniture arranging’ to be supervised by a team member with a few year’s experience who can discern re-arrangements which improve habitation from those which only stamp personality.

Example

An existing team member adopted some code with extremely long-winded variable names from a departing developer. The first ‘furniture arrangement’ task was to replace the hierarchically constructed names with short names which made the code easier to read, and eliminated many unnecessarily long source lines.

Related Patterns

Arranging the furniture usually occurs after an *EFFECTIVE HANDOVER*. *ARRANGING THE FURNITURE* can still be done at times of high *PROJECT PULSE*, if the arrangements are made at the site of current work. Some arrangements might become *SHOW AND TELL* topics.

Acknowledgments

These patterns have been mined from the author’s experience of five software development projects over a 12 year period working in teams in the following application domains—distributed transaction switches, communication protocol engines, client-server middleware, legacy application gateways, and telecommunications fault management systems. Many discussions on the themes of these patterns have taken place with Darren Allen, Umberto Bonollo, Martin Dick, David Lawson, Alan McPharlane, Christine Mingins and Inji Wijegunaratne.

This pattern language owes the most to its PLoP’98 shepherd Don Olson, who waded through five early versions, spotting the occasional glimmer of truth amidst piles of opinion, observation and memories. His suggestions have been invaluable.

The images used herein were obtained from IMSI’s Master Photos Collection, 1895 Francisco Blvd. East, San Rafael, CA 94901-5506, USA.

References

Alexander, C. 1977. “A Pattern Language”, Center for Environmental Structure, Berkeley, University of California. Oxford Press.

Alexander, C. 1979. “The Timeless Way of Building”, Center for Environmental Structure, Berkeley, University of California. Oxford Press.

Brooks, F.P. 1995. “The Mythical Man-Month (Anniversary Edition)”, Addison-Wesley, Boston, MA.

Coplien, J.O. 1995. “Patterns for Software Development Organisations”, Proceedings of the First Conference of Pattern Languages of Program Design (PLoP1), Addison Wesley.

Raymond, E. 1998. “The Cathedral and the Bazaar”, http://www.redhat.com/redhat/cathedral-bazaar/cathedral-bazaar.html, accessed 28 April 1998.