

# Excerpts From A Set Of Technical Leadership Patterns

Mark Prince, Andy Schneider

BJSS, 1<sup>st</sup> Floor Coronet House, Queens Street, Leeds LS1 4PW UK

[mark.prince@bjss.co.uk](mailto:mark.prince@bjss.co.uk)

[andrew.schneider@bjss.co.uk](mailto:andrew.schneider@bjss.co.uk)

**Abstract.** This paper presents four patterns; *InformationIsKing*, *CoreSampling*, *CodeSurfing*, and *TakeCareOfTheSmallChange*. *CodeSurfing* describes a mechanism for parsing changes in a code base to ensure expectations are met. *CoreSampling* describes a mechanism for detailed information analysis of a developers work. The paper also covers a management aspect of Technical Leadership -*TakeCareOfTheSmallChange*. This mixture was deliberately chosen to reflect the mix of activities a technical lead undertakes.

## Introduction

Traditionally developers travel three paths to Technical Leadership:

- Default appointment due to current incumbent moving
- Being asked to take charge of a team (not necessarily knowing *why* there is an opening)
- Observing that leadership and/or organisation is needed within a team and stepping up to the challenge. This often happens with more “natural” leaders.

It is very unlikely that the developer will be sent on a course and subsequently be given a mentor to ensure their technical leadership skills flourish. This is a significant problem because successful Team Leadership and Architecting require mastery of a diverse set of skills. For example, a Team Leader will probably need to combine the roles of Team Manager, Design Authority, Communicator and Mentor. He will need to organise the team, manage the technical direction of the development, liaise with the rest of the organisation and look after the health and growth of his team. The journey from good developer to good Team Leader can be a hard one given the lack of training and broad selection of skills required.

This paper represents part of a larger effort to capture a set of patterns<sup>1</sup> to fill the knowledge gap which confronts the inexperienced and aspiring Team Leader and Architect. As such the target audience for the pattern and proto-pattern work is IT Software professionals that have had between 2 and 5 years industrial experience.. Line Managers, Programme Managers and Project Managers are not the target audience for the patterns.

## Pattern Relationships

Figure 1 (below) shows the key relationships between the patterns described in this paper.

*InformationIsKing* describes the importance of information to a Lead and the need to critique information appropriately. The rest of the patterns in this paper provide mechanisms for gathering and analysing information. *CodeSurfing* captures one of the activities that successful Leads use to keep track of the evolution of the development. This technique can help Leads target defect reduction strategies so they are applied in a surgical manner. The paper then presents a related pattern, *CoreSampling*. *CoreSampling* captures the habit some successful leads have of pairing with a team member and having them walk through the code or design currently being worked on. This informal habit provides a mechanism for understanding the detail in isolated areas. Both *CoreSampling* and *CodeSurfing* provide advanced warning of problems such as quality issues and allow the Lead to determine whether activities such as mentoring, inspections etc. would benefit the developer. These patterns also work to provide early detection of micro-scope creep, a problem addressed by *TakeCareOfTheSmallChange*.

---

<sup>1</sup> Alongside other patterns work such as [1], [2], [3], [4] and [5].

Mark Prince, Andy Schneider

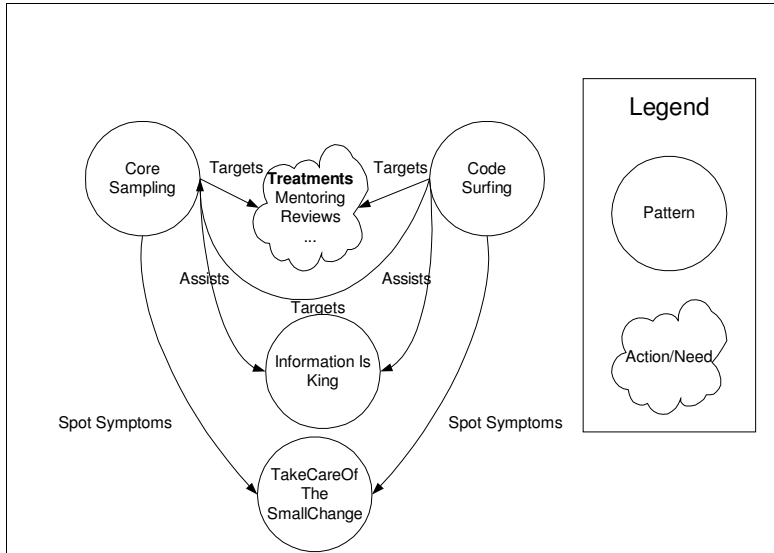


Figure 1: Pattern Relationships

When considering these patterns it is useful to appreciate that the Team Leader or Architect often has two roles:

- Technical Leader
- Project Leader (Team Leaders and Architects often run teams, which means they usually perform some project management).

The patterns presented in the paper describe choices and actions that are owned by one or both of the above roles. The ownership is described in the table below.

Pattern/Aspect	Technical Leadership	Project Leadership
InformationIsKing	Shared	Shared
CoreSampling	Primary	
CodeSurfing	Primary	
TakeCareOfTheSmallChange	Shared	Shared

## InformationIsKing

**To manage rather than react, a Lead needs to gather and analyse information flowing both within and without the team.**

The Lead needs to sift information and take appropriate action. This is analogous to a bush pilot on patrol who spots a plume of smoke. The pilot will radio back for an investigation rather than fly lower to see what’s causing the smoke. The results of the investigation need to be assessed and appropriate action taken – is the grain store on fire or has someone got a smoky barbeque?

## Excerpts From A Set Of Technical Leadership Patterns

Some of the forces affecting the information gathering process are:

- **Context** – the Lead needs to understand the context within which information is gathered, constructed and provided. This is in much the same way as one ought to be aware of the political bias and business interests of a newspaper when reading it. The Lead must be aware not only of the context within which the information giver provides information but also of his or her own prejudices. Actively thinking about these details allows a more accurate view of the information to be formed.
- **Granularity** – the Lead needs information about the whole. Since very few people can hold the detail of an entire project in their heads the Lead needs to ensure that such information is of a systemic nature, rather than being detailed information about a part of the project. However, how is the Lead to be able to critique information received without a detailed understanding? If the Lead has little understanding of the detail then a natural response is either to accept or reject information with little thought. It is therefore key that the Lead gathers enough detailed information to allow other data to be assessed on its merits.
- **Validation** – writing a thing down does not make it a fact. The Lead must know when to validate, research and question information. [28] shows the importance of validation when considering information. Validation also requires an understanding of the area of interest. Whilst a Lead can always “Get a guru”[1], the Lead stills needs enough understanding to communicate with the Guru.
- **Networks** – all organisations have formal hierarchies and information channels. These are useful sources of information. However, be aware that many overlapping informal networks also exist. It is therefore important to build cross-team contacts and spend time looking after those relationships. Failure to do so will result in an inward rather than outward perspective and therefore the team is in danger of becoming detached from its customers and stakeholders.

There are many different types of information that need gathering. Some of these are:

- **Requirement changes** - The lead needs to ensure they are aware of changes in order to manage their impact, rather than react when the consequence of the change becomes apparent. This requires the Lead to understand not only the broad requirements, but the small changes that occur on a day by day basis.
- **Quality of the team’s output** – if defect rates are rising or it looks as if a developer lacks the appropriate skills to complete their job effectively then action needs to be taken. It is the responsibility of the lead to refactor with appropriate mechanisms (such as mentoring, training, more appropriate roles, etc.).
- **Progress** – How is the team doing? Who is struggling? Who is ahead of schedule? This sort of information allows the Lead to watch out for potential slips against the schedule and to help when a team member needs support.
- **Organisation** – There may be changes going on within the organisation. These could well impact the team and the Lead should make sure they are aware of pending changes before they occur.

A wide variety of information is required, at multiple levels of detail for the Lead to become effective and have enough data to manage rather than react to events as they occur or when it is too late.

*Therefore*

**Practise detailed technical leadership – gather both broad and detailed information. Understand enough about the project to make educated decisions rather than “shooting from the hip” or avoiding the decision altogether.**

Detailed technical leadership challenges the lead to:

- Understand the technologies well enough to have intelligent discussions with the developers.
- Understand the detail of the design and implementation to ensure appropriate decisions are made. However, the Lead cannot hope to understand it all. Design reviews, inspections, *CoreSampling* and *CodeSurfing* are strategies for handling this quandary.
- Ensure the Team understands why some of the Lead’s decisions have to take into account political considerations and factors outside the team. The team needs to understand that a decision may not necessarily be the best team-centric decision but may be better for the overall delivery.
- Understand the core competencies in the team in order to ensure everyone is utilised effectively, challenged and mentored.

### Mark Prince, Andy Schneider

- Analyse information so that appropriate conclusions can be drawn. A Lead may ask a developer about progress. The developer, being busy, may respond with “ok”. It is up to the Lead to understand whether this information should be accepted at face value or whether to probe deeper. In the latter case, be relaxed, ask leading questions. Do not interrogate, criticise or express disbelief. Be sensitive that your position alone may intimidate other members of the team - humour can break down the barriers and allow the team member to feel more comfortable in expressing themselves.
- Correctly balance the amount of detailed and broad information that is required.
- Know enough about the team and the information to delegate the decision into the team. The team can then present options to the Lead who can make the decision. This empowers the team and makes them part of the decision making process whilst allowing the lead to *Lead*.
- Actively manage small changes in scope (see *TakeCareOfTheSmallChange*) and schedule rather than waiting until the small problem becomes a large one.

### Incorrect Application

- Without understanding the detail of the information flowing through the team, the Lead will fall into a common trap: ignoring the detail and assuming that the solution is “not hard, just a technical issue”. This results in an over confident Lead who fails to detect problems until they are found in testing, future phases of work or have resulted in a significant project crisis.
- The Lead must be careful not to confuse Detailed Technical Leadership with Micromanagement. Detailed Technical Leadership is the application of deep understanding of a project to its management. Micromanagement is the management of team member’s tasks on a daily or hourly basis. Micromanagement is fuelled by mistrust, introduces disaffection in the team and does not scale.
- Over application means getting lost in the detail (“Analysis Paralysis”), and the Lead will be unable to take a systemic view. *Hover Shoes*[1] is a useful refactoring in this instance.
- Too much information and not enough time to process will bring on self-induced stress. *Don’tSpareTheDelegation*<sup>2</sup> is a useful refactoring here, and brings the team more into the decision making process.

### Related Practices

Management By Walking Around (MBWA) [16] is one mechanism for gathering information from within the team. So are conversations, meetings, e-mail, media, software inspections [8], daily scrums [21], walkthroughs [8] and inspections [10]. In all cases, apply thoughtfully and actively analyse the information gathered before applying it.

## CodeSurfing

### It is essential to get a feel of the terrain in order to ensure development finds the best way forward.

One of the challenges of Technical Leadership is to understand enough about the code and design base to manage the team, and by implication, its output.

CodeSurfing is a pattern for taking a breadth-first view over the code and design base in order to identify any potential areas of worry. This is rather like taking an aeroplane up and scanning for plumes of smoke in the forest. Once the broad view has identified an area of concern, the Technical Lead can investigate further (such as *CoreSampling* or reviews) and ensure corrective action is taken. This pattern should be used to guide the Technical Lead to target improvement mechanisms. Some of the key forces in how and when to apply *CodeSurfing* are:

---

<sup>2</sup> See Proto-Patlets section

## Excerpts From A Set Of Technical Leadership Patterns

- **Code and Design Base Size** - The design or development effort of the team will reach a critical mass after which the Technical Lead can't understand it all or directly control the quality.
- **Freedom** - When a Lead delegates to the team and provides freedom, the team benefits. A good team has freedom to make decisions, take risks and deliver. However, a Lead stills needs to monitor quality and know when to provide a Guiding Hand[1].
- **Early Warnings** - The longer a problem exists, the greater the cost of fixing it [9].
- **Empowerment** - The Technical Lead needs to allow the team members to take ownership of the quality themselves. Failure to do this will lead to either:
  - micro-management<sup>3</sup>, where the Lead manages every aspect of the delivery or
  - a significantly increased risk of quality problems during the latter stages of the project, where the Lead fails to delegate responsibility yet does not fill the quality assurance role.
- **Team Experience Level** – for a team mainly comprised of inexperienced members, a feedback mechanism like this can consume too much of the Technical Lead's time. In this situation, it may be more productive to devote more time to mentoring. If mentoring consumes too much time, it may be appropriate to RaiseTheYellowFlag<sup>4</sup> and request another senior technical resource be drafted into the team.
- **Trust** - The technical lead needs to transfer from faith to trust in the team. Rather than the team relying on the Technical Lead for guidance and control, the Technical Lead can rely on the team for delivery. To make that leap the Lead needs to track the quality of the team deliverables.
- **Mentoring** - To grow and empower the team, mentoring (amongst other techniques) needs to be applied. However, it may not be clear where to best target mentoring time.
- **Bandwidth** - The Technical Lead can only dedicate a finite amount of time to maintaining the quality of the code and design base.
- **Churn** – A Technical Lead can be confident in the strengths and weaknesses of a stable team. However, if the Technical Lead has frequently to change teams or there are a lot of changes in the team, the Technical Lead will need to understand what is going on.
- **People** – some developers are sensitive to criticism and this may be an issue when corrective action is applied.
- **Requirements** – the Technical Lead needs to ensure requirements are being satisfied
- **A feel for the terrain** – is the architecture being implemented? Are there too many Bad Smells [27]? Are implementations too light for the perceived complexity of the problem domain?

*Therefore*

**Go CodeSurfing. It gives an understanding of what is happening and is an unobtrusive way of ensuring the broader quality picture is understood. It is not an alternative to inspection techniques but aims to give insight to where more detailed inspections may be applied.**

Allocate an amount of time and go trawling through the recent check-ins and look for problems – you are looking at the overall shape of things, rather than delving into the detail. Going through check-ins rather than work in progress is more representative of the quality of the artefacts being produced. Check-in based surfing also allows you to assess who is checking in frequently, and possibly who isn't checking in at all.

When surfing look for warning signs that indicate something may need addressing. Suggested points of interest are:

- Are standards being complied with?
- Are the team members repeatedly making the same mistakes?

---

<sup>3</sup> See CoreSampling for a brief discussion on the perils of micro-management.

<sup>4</sup> See Proto-Patletts

## Mark Prince, Andy Schneider

- Are appropriate language specific idioms being used?
- Are there *Bad Smells*? If so, have the developer refactor with the normal patterns.
- Are there any anti-patterns (such as *CodeForCrufts*<sup>5</sup> or *GoldPlating*[7] )?
- Does there seem to be a significant amount of development activity in one area of code that should be closed off – if so, is there cause for worry? For example, five different developers have all changed the same 20 lines of code in the same method over a two-week period; there may be broader issues that need to be addressed.
- Can the work be related back to a requirement - are significant requirements apparently not implemented? The converse is also true – the team may have discovered work that needs to be done that isn't captured as a requirement. If so, make it so, but pay heed to *TakeCareOfTheSmallChange*.
- Are there broader based issues emerging that aren't the fault of individuals, but will require intervention to ensure the team is pulling in the same direction? An example would be the lead using his holistic view to spot obvious duplications.

Once an issue has been identified the first step is to investigate further. Investigative techniques can range from a quick chat, through *CoreSampling* and pair programming to more formal inspections. Whilst *CodeSurfing* helps spot problems early, its primary use is to take the broad view. Defects can be spotted during *CodeSurfing*, but it is more likely that defects will be found after further investigation has taken place.

## Application Techniques

- Make sure everyone knows that you surf the code. Do not be apologetic but do be sensitive. You are not trying to make a point or be a pedant. It is your role to ensure the team fulfils its goals.
- When you're *CodeSurfing* and sitting with the team, be careful of the impression you project with your body-language.
- Avoid making your team dependent on you by providing answers to all the questions [26]. If there is an opportunity to mentor, don't approach the team member and say "you should do it like this", take a softer and definitely positive approach; "I see what you're trying to achieve here, and it's good for these reasons. Did you know you can do this..." List the reasons to show you have understood the team member's work and are trying to help them improve, rather than shoot from the hip. Alternatively, simply showing them an example of what you'd like can also work effectively. An advantage of the latter is that you can empower the team member by saying "take a look at that and let me know what you think". This puts control back into the developers' hands.
- Time box *CodeSurfing*. Initially apply the *CodeSurfing* regularly. When you start to become comfortable, you will be able to apply this less often to maintain the same comfort levels.
- Unless you see a pressing problem that requires immediate investigation, complete your surfing before addressing any investigative activity required. It enhances your leadership if you can spot the big picture issues rather than continuously drilling down into the detail with team members. They should be perfectly capable of dealing with detail issues using normal peer techniques (such as pair programming [22] or informal peer reviews).
- Keep it going – you may reach the stage where you're applying *CodeSurfing* on an ad-hoc basis, but that move should be a conscious decision. As the team settles down into a steady rhythm it will become clear what areas of the code are generally problematic and who needs more mentoring. This may result in less need for *CodeSurfing* on a regular basis. However, if a completely new developer arrives or significant different functionality is required then *CodeSurfing* may need to be more regular for a while. *CodeSurfing* should ebb and flow to match the needs of the project.
- Apply when the team is under deadline pressure and the temptation is greater to slip the quality. Prioritise any application of corrective action and apply it very delicately because everyone in the team will be under stress. This is a good opportunity for you to show you can rise above the stress and Lead effectively.
- You are not a perfectionist . Just because you'd do things differently may not be sufficient reason to have a team member change their work.
- When *CodeSurfing* indicates that you need to investigate further, apply any corrective action informally; go over and chat with the member in question. *CoreSampling* describes several informal techniques for approaching team members.

---

<sup>5</sup> See Proto-Patlets section.

## Excerpts From A Set Of Technical Leadership Patterns

- Remember, this is very much a people issue and needs to be handled carefully. As the Lead, you need to be aware of how different people respond. Choose techniques that work with each individual. Sometimes advice can meet resistance. This can be for many reasons. Weinberg [26] has excellent coverage of how to handle this in the section “What to do when they resist”.
- *CodeSurfing* does not lend itself to pairing (except in a mentoring role – see below). Pairing *CodeSurfing* has a number of problems:
  - Potential problems are discovered in a joint forum. If the issue is sensitive then it is harder to tackle in private.
  - *CodeSurfing* should be a time-boxed, concentrated process. Pairing involves communication, which may extend *CodeSurfing* into a mentoring session. If *CodeSurfing* is to be the forum for mentoring it should be a conscious decision and appropriate time should be planned in.
  - *CodeSurfing* is a combination of browsing and reading – this is not normally a paired activity and the authors have no information to suggest that *CodeSurfing* would benefit from pairing.

### Over Application

A badly run code inspection or insensitive mentoring can result in an atmosphere that is defensive rather than constructive. *CodeSurfing*, as a quality assurance mechanism carries the same risks. Mistakes to avoid are:

- Assuming the developer is in error. In the majority of cases there is a good reason why things are the way they are. It is important therefore to assume this and use an appropriate mechanism (mentoring, *CoreSampling*, reviews).
- Pointing out problems in code to people sitting next to you. Issues should be taken up with the developer in question.
- Using *CodeSurfing* as a control mechanism. *CodeSurfing* is an early warning system, a way of targeting positive mechanisms such as mentoring. Performing *CodeSurfing* and then haranguing a team member will result in the entire process being seen in a negative light.
- Acting on impulse.
- Exploring detail with *CodeSurfing*. If the Lead becomes embroiled in micro-issues then the surfing will take too long and it will be perceived as having little value. The Lead does not win respect by constantly pulling the team member up on every small detail.
- Using *CodeSurfing* primarily as a defect detection mechanism. Inspections are a far better mechanism for defect detection than time spent *CodeSurfing*.
- Using *CodeSurfing* as a threat, i.e. “I’ll be surfing your code later...”.

Over application may also consume more time than you have available. This pattern should be considered a tool for the Technical Lead rather than the sole activity required to lead the team.

### Related Practices

Practices such as the Lead Pair-Programming [22] provide a detailed view of what is happening within the team. However, the Lead normally does not have time to pair with each member of the team. *CodeSurfing* helps by providing a “big picture” view of the terrain. From this the Lead can decide to go *CoreSampling*, organise an inspection (see Gilb [8] for a detailed review of inspections), pair with the developer in question (for mentoring purposes) or use any other of a number of quality assurance mechanisms.

In many ways *CodeSurfing* can be seen as a specialisation of a code review, where the objective is to discover general problem areas, (that exhibit themselves at the macro level), rather than specific details.

**Mark Prince, Andy Schneider**

**Examples:**

**Detecting problems with responsibility allocations in classes.**

*CodeSurfing* revealed a business domain class that contained only *get* and *set* methods with little business logic. To resolve this, further investigation was needed. The lead practised *CoreSampling*. The opening gambit was to discuss the requirements at the coffee area and then walk back to the developer's desk. Having opened a conversation, a question could then be asked: "I noticed that the domain object didn't contain much business logic. What are its responsibilities?" This led to a discussion about what elements invoked the domain object and where the functionality was. The Lead followed with questions exploring the allocation of responsibilities. After a while it became apparent that there was a misunderstanding in the use of J2EE session beans. Once the Lead had explained, the code was refactored. The net result was mentoring, and the developer learnt about encapsulation. Meanwhile, the Lead had shown he was genuinely interested and willing to share knowledge. It also led to the earlier discovery of potentially significant problem that would have resulted in significant refactoring if it had been left to the end of the iteration, or worse still, was discovered when maintenance proved expensive.

**Finding problems in inherited code.**

A team was working on a COTS based project that involved code inherited from an external supplier. *CodeSurfing* the inherited code revealed far greater complexity than the problem being solved warranted. As a result of this broad impression the Lead organised a code inspection. This inspection discovered nine bugs in seven lines of code. Further detailed examination revealed the code base was months away from being production ready and the Lead had the opportunity to raise the issue with his manager. The end result was the inherited code was de-scoped from the first release. This gave time for the development to be refactored and ready for production. Without this discovery the system would have gone live and failed in days.

## **CoreSampling**

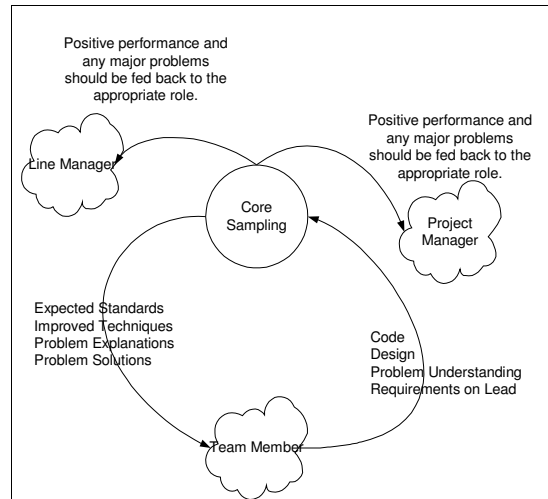
**In any project of reasonable size the Lead cannot understand the entire implementation in enough depth to feed appropriate feedback loops.**

With the best team in the world, a Technical Lead can give them a business case and play Asteroids until the delivery occurs. However, with all the other teams in the world, the Lead needs to apply a feedback loop. This feedback loop needs to positively reinforce a Team Member when expectations are being met and to allow for a GuidingHand[1] to nudge the Team Member back on track when the expectations are not being met.

- The Team Members need to positively reinforce the Lead when the Lead is meeting expectations and express their expectations when the Lead is not meeting them.
- The Technical Lead taps the feedback loop to grow his understanding of the technical detail.
- The Technical Lead feeds back timely information to the other management roles within the organisation.

A side effect of the established information flows is that information is gathered within iterations and not just at the end. This is important because problems discovered early in an iteration are fixed more easily than those found at the end.

## Excerpts From A Set Of Technical Leadership Patterns



**Figure 2: Feedback**

Constructing and maintaining the feedback loop requires a forum for the gathering of detailed information and the communication of expectations and ideas. However, in any project of a reasonable complexity, the size of the code base precludes understanding all of it. The code will need to be mined in order to get a detailed understanding of the quality of the delivery – architecture, design and code. Gathering information on the code base can be problematic because there are also non-technical forces in play.

Some of the key forces are:

- **Productivity** - The process must avoid interrupting developers when they are “in the zone”. DeMarco and Lister[15] describe the effect interruptions can have during this time. They show that an interruption loses not only the raw developer time, but also the time required for reimmersion. The nature of being “in the zone” is described well by Csikszentmihaly [14].
- **Participation** - The feedback loop requires the team to be involved. The developer and Lead should feel that the act of participation is worthwhile.
- **Growth** - The actions taken as a result of feedback should contribute to the growth of the developer, team and the Lead.
- **Micromanagement** - Executed badly, any review type mechanism can become a tool for tyranny. Constant interruptions, micro-management and supervision shows a lack of trust in the team and reduces opportunities for team growth. The end result will be a disengaged and disenfranchised team. The quality will drop, the delivery may run late or the project may even fail. A micro-managed team has a short shelf life.
- **Bandwidth** - Time spent gathering information must be in proportion to the time spent on other aspects of the Lead’s role. Each task must be kept in balance, according to factors such as urgency and importance.
- **Detachment** - “Architect Also Implements”[6] is an important practice but a Lead may not always have time to do this<sup>6</sup>. CoreSampling complements “Architect Also Implements” and, in the case when the Lead does not have time to implement, addresses the “architect becomes detached” problem.

*Therefore*

### **Take core samples and extrapolate.**

The art of *CoreSampling* is to pick a thread of development and follow it from top to bottom with the developer. Gain impressions from a number of these investigations to extrapolate and measure the health of the entire code base.

<sup>6</sup> Not always practical on large projects or projects where other aspects of leadership mean time is not available for coding.

## Mark Prince, Andy Schneider

*CoreSampling* is less expensive in time than a structured walkthrough or inspection (see Related Practices below)., However it should still be used in a manner that does not consume unsustainable amounts of the Lead or team's time. If *CoreSampling* is happening once a day, it is too often (see Over Application below).

When following a sample through with a developer, be careful to avoid forcing a context switch on them. Make sure an appropriate time is picked to pair with the team member in question. This time doesn't have to be booked. In fact, a degree of spontaneity provides greater flexibility. It would be inappropriate to interrupt a team member when they are absorbed in a problem or discussing an issue with someone else. Instead, be aware of the environment and the ebb and flow of work. Look for signs that the person you wish to work with is free, as you normally would before interrupting someone.. People tend to work better at particular times of the day so try and avoid interrupting people during these periods. Look for relevant openings in the ongoing conversation. For example, you may be discussing a technical issue over coffee. Once the discussion has finished an opening gambit could be "I was wondering how you've gone about structuring the blah, can you walk me through it?" The person may say "no, not at the moment, but I'm free in an hour/ after 6". This is fine, you have an agreed time.

Remember the aim is to understand a slice through the code so that a view can be formed of the current implementation. The activity needs to be treated as exploration, where information is freely exchanged, techniques swapped and expectations confirmed or reinforced. Once you have started to work through the sample with the team member, ask leading questions in a humble fashion, e.g. "I don't understand that - how does that work?" and dig into a small piece of detail. In addition, keep it informal, the idea is to create an open environment for learning about the design or code.

Suggested conversation devices :

- Ask the developer to provide an explanation of how he is addressing the problems presented by the current requirements. Sometimes developers suffer from "analysis paralysis" or are "*GoldPlating* [7]" the solution. A focussed discussion around the existing requirement can spot this. A Lead often has the best holistic view of the requirements. Discussing the implementation from a requirements perspective allows information around the "big picture" to flow within the pair. This is also the opportunity to check the coverage of unit tests.
- Ask the developer how his work integrates with another software element being built. This approach can elicit interesting information on the coupling between software elements. It is a good place to catch problems caused by people not talking to each other enough and just assuming elements interface in a certain way. This is also a good opportunity to bring the Lead's holistic view to the table. The Lead knows what is going on across the team, and can therefore spot opportunities for collaboration or pairing that the developer may not be aware of.
- If possible, have the code executed in front of you – a good idea when GUI development is being done. Single out an interesting feature and have the developer walk through the code to gain an understanding of how it was done. This may be a learning experience for the Lead. Use this technique to see if the code complexity matches the requirement. This type of "show and tell" often occurs at the end of an iteration. However, finding a problem within an iteration is better than finding one at the end.
- If part of the solution has to be complex, have the developer talk through how he did it – this allows the fragility to be assessed. Discussing the code or design is a learning experience. You learn about the design or code and the developer learns more about your expectations whilst swapping information on requirements and good development practices. Remember, it is important to ask open questions. The aim is to get the developer to open up and explore his solution and its ramifications. Questions that can be answered with a simple yes or no should be avoided as this can make the developer feel they are being grilled for information. When discussing the code or design it is important to bear the following in mind:
  - Distinguish between advice that must result in a change to the code or design and advice that is meant for the longer term.
  - The team must be made to feel safe. Safe to describe problems, safe to challenge the Lead, safe to make decisions and take risks. Olsen & Stimmler [1] describe this in the EnoughRope pattern. McCarthy and McCarthy also discuss the importance of establishing safety in the section on Alignment Patterns in [2].
  - Be cognisant of the developers' workload and prioritise any explicit changes required.
  - Avoid judgemental statements. They will result in defensive behaviour. Instead explore the ramifications of any problem. Lead the developer to work out what the problem is.
  - Do not criticise. Problems will probably be due to lack of developer knowledge in a particular area or external influences such as you not executing parts of *your* job well enough. However, if there is a real problem do not be afraid to address it; but address it in a private meeting room, *after* you have had time to formulate your views and consider all aspects of the problem.
  - Avoid commenting on the minutiae. Whether the developer puts spaces between method names and brackets is irrelevant<sup>7</sup>.

---

<sup>7</sup> If it really is important then use a pretty printer that is triggered on check-in.

### Excerpts From A Set Of Technical Leadership Patterns

- Practise effective listening. Playback your understanding, “so, what you mean is....”. This way you can be sure you accurately understand what you are hearing.. See [17] for more information on effective listening.
- Give advice freely. Gabriel [25] discusses how important this is. Dikel et al [13] touch on this issue in the Reciprocity pattern. They explain how a “fair and proactive exchange of value” can assist in building co-operative relationships. Whilst Dikel et al are concerned with cross team relationships, the same applies within the team.
- Use terms such as “I feel that this isn’t right” rather than “this isn’t right”. No-one can argue about your feelings, and it avoids statements sounding accusatory.
- The developer will have been through a discovery process to bring the code (or design) to the point that it is. Therefore, understand the history before drawing any conclusions about the development.
- Respond positively to good practices. The point is to reinforce what is needed.
- Weinberg [26] points out that simply providing solutions to problems makes the receiver more dependent. Instead, work to improve the developers’ problem solving skills. One technique is to lead the developer into a discussion about the ramifications of the decisions that have been taken. Questions such as “what would happen in this case?” can open avenues of exploration and learning that would not exist if the developer was just told “do it like this”.
- When problems are noted, consider what systemic solutions can be applied within the team. Try to react strategically as well as tactically. Feeding these remedies into the Lead related activities is an important part of the process.

Successfully applied, *CoreSampling* will enable the lead to manage the team more effectively by:

- Providing a guiding hand early in the project to ensure the team is operating within acceptable parameters
- Spotting problems early. One example is scope creep. If micro-changes in scope are occurring then apply *TakeCareOfTheSmallChange*.
- Providing pointers to areas of code and design that will benefit most from more formal reviews.
- Providing opportunities for mentoring, training or the provision of a greater challenge.
- Providing opportunities to spot synergies between team members that the members may not have spotted.
- Mentoring the team, and providing encouragement when a team member has done a good thing.
- Spotting a struggling team member. This can be fixed with mentoring, training, relieving their anxiety, migration to a different role or, if all else fails, removal of the person from the team. See *RottenFruit* [1].
- Increasing the Lead’s impression of the code base, even though it may be so large that overall comprehension is difficult.
- Ensuring regular interaction on a detailed technical basis with the team. This helps the team to gel and helps goals to be aligned.
- Allowing the health of the code base to be extrapolated in cases where a broad view is difficult due to its size.

### Over application of CoreSampling

Like all review mechanisms, *CoreSampling* can be over-applied. The Lead’s role is to assist in the delivery of the software development, within certain explicit and implicit constraints. *CoreSampling* is an enabler, not a goal in itself. Over application will result in:

- The team members not feeling trusted because they are constantly being questioned on their design and implementation.
- Productivity dropping due to constant interruptions.
- The Lead spending all his or her time *CoreSampling* rather than leading the team.
- The Lead being so far into the detail that he cannot “see the wood for the trees”.

Mark Prince, Andy Schneider

### Related Practices

Peters [16] advocated “Management by Walking Around” (MBWA) as a means of addressing the problem that management are often remote from the detail and out of touch with their people and customers. Peters described a number of practices, such as reserving time to walk around the office talking to the team. Leads who practice *CoreSampling* are acknowledging the benefits of such an approach. However, they are blending this with aspects of informal Code Walkthroughs and Pair Programming [22]. The informal walkthroughs are not as structured as those described in Yourdon’s work [10], but they do provide the information needed to form a view as to whether the code needs additional investigation or remedial work. Pair Programming provides some of the benefits of *CoreSampling* but a Lead often does not have enough time for serious Pair Programming with all members of the team.

*CoreSampling* is just another technique in a successful Leaders toolbox, it is not a replacement for inspections, structured walkthroughs or testing. It is related to *CodeSurfing*, in that the latter may result in the former. The following table outlines key differences between the two:

Aspect	<i>CodeSurfing</i>	<i>CoreSampling</i>
<b>Time</b>	<b>Lead:</b> 30 minutes or an hour, initially each day (or every other day). Reduces over time (as confidence and trust are built) to maybe once per week. <b>Developer:</b> None, i.e. developer is not involved in <i>CodeSurfing</i> .	<b>Lead:</b> 30-90 minutes, depending on issues raised, on an ad-hoc basis. <b>Developer:</b> Same time as lead.
<b>Depth of information.</b>	Broad but shallow information. Provides information on key problems – but information is not detailed.	Detailed information on a small section of the development.
<b>Context</b>	As <i>CodeSurfing</i> is a lone activity, much of the context (i.e. why the design/code the way it is) is not captured. As a result, what can seem like a major problem may in fact be the best solution given the constraints. <i>CodeSurfing</i> cannot (without further investigation) validate a design choice against the context.	Working with the developer builds a rich context within which discussions of the form “why is it like this?” can take place.
<b>Mentoring</b>	<i>CodeSurfing</i> may indicate that mentoring is required but it does not provide mentoring itself.	<i>CoreSampling</i> provides a forum in which both the lead and the developer can learn from each other.
<b>Direction action.</b>	<i>CodeSurfing</i> is likely to result in further investigation, but is less likely to result in a direct change. This is because of the nature of the activity – there is a lack of context and detailed information.	<i>CoreSampling</i> can often mutate into a brief pair programming session. In this way <i>CoreSampling</i> is far more likely to involve direct action and feedback than <i>CodeSurfing</i> .

### TakeCareOfTheSmallChange

**Minor scope changes can cause the development schedule to slip undetected, but rejecting change stifles product development.**

The following is taken from a recent project:

The users decide they want some minor internationalisation of the static HTML. This can be done trivially, using Apache content negotiation. The project team agrees; it’s so simple, it’s not worth putting in the plan. The following week the users decide they need to index their preferences by name. This is also easy. Just add a column in the preferences table with the name (key) of the preference set. The estimates do not need to be re-visited since the solution is so simple. The users then decide they need to change the colour of the browser background. This just requires a developer to change the style for the background. It will take no more than a minute.

## Excerpts From A Set Of Technical Leadership Patterns

However, this could cause a context switch if it is requested immediately. None of the above changes constitute a significant departure from the plan. However, the sum of these changes is not negligible. Each change results in a micro-slip. Slowly changes accrue to become concerning. If the requirements are not actively managed then the project will slip. Testing compounds the problem. If minor changes are simply "done" (because they are so small) then formal acceptance test plans may not get updated or formal acceptance testing may balloon as the minor changes impact the test definitions. Either way, it is not just development that has been affected, but other parts of the lifecycle.

Jones [19] indicates that the average project experiences a 25% change in requirements over its lifetime. Jones also shows that many projects that fail to manage changing requirements are more likely to suffer from increased schedule compression. One reaction is to refuse all changes. However, resisting all change results in an inflexible development that is not likely to satisfy the customer. In addition, using a requirements document to enforce a particular set of requirements, against the users wishes, can lead to development moving elsewhere – as discussed by Yourdon in [20].

*Therefore:*

### **Take care of small change. Make sure all small changes in requirements are analysed and managed.**

Users can and will change their requirements. However small, these changes should be fed back into the plan (story cards, MS Project, Excel, the medium is not important) as scope needs to be controlled. Time boxing a project is acceptable, but if the resulting project no longer provides the required business value then the development was pointless. Welcome change, but treat with caution. Apply the “Must, Should, Could, Would” rules [12] to the change requests. Be sure to follow through on the impacts by asking questions such as:

- Does this change affect testing?
- Does this change affect any data migration activity?
- Does this change affect any non-functional requirements?
- Does this change affect deployment?
- Does this affect a support team (such as Operations)?

Allocate ‘Should’ or ‘Could’ requirements as late in the plan as possible, preferably in the spare iteration<sup>8</sup>. ‘Would’ requirements will not get done in this project, use them as input to the next. If not allocated to the spare iteration then ensure that space is made in the target iteration by moving content from the target iteration.

Irrespective of where requirements are located, any change must result in a change to the plan or stories. If the plan or stories are not changed then the result will be unmanaged requirements and therefore:

- Possible variance against the plan.
- As the project slips, unimportant changes can push out important functionality from a later iteration.
- Incomplete implementation (as changes are forgotten).
- Dissatisfied management. Scope problems are not communicated to management in a timely fashion.
- Schedule compression as additional functionality is squeezed into the project timeline.
- Lower quality due to function pushing out quality assurance activities.

Ensure the prioritisation is performed with the users, domain experts or whoever is the source of the requirements. Communicate the cost of minor changes carefully, so that all understand the impacts. This is important because often user representatives do not understand the lifecycle wide implications of certain changes. This in turn allows a feedback loop to be established with the user representatives. The representatives will become aware of the need for prioritisation and, with guidance, will learn how to apply prioritisation before the changes are passed on to the project team. A combination of feedback and joint prioritisation keeps the representatives engaged, presents the issue of scope control as a problem to be jointly solved and allows expectations to be managed. Failure to establish this type of collaborative working will result in a “them and us” situation, and the user representatives becoming disenfranchised.

---

<sup>8</sup> Some iterative developments plan in a spare iteration. One approach is to give Marketing the spare iteration for late breaking requirements. Once they’ve used the iteration they are done. Spare iterations work best when it is understood by all that the time in the spare iteration can be “drawn down” if items in other iterations take longer than expected.

**Mark Prince, Andy Schneider**

## **Related Practices**

Other complementary practices are:

- De-scoping – the practice of dropping functionality to meet deadlines. McCarthy[23] describes the successful application of de-scoping in Microsoft.
- Change boards – for formal control of changing requirements. Jones [19] has shown their effectiveness.
- Short release cycles – to elicit greater customer input. Many iterative developments such as Scrum[21], RUP[6] and XP[22] use this model.
- Prototyping (and JAD) provide a mechanism for specifying functionality more accurately by involving the users in more collaborative ways. Again, Jones[19] shows these techniques significantly reduce the likelihood of requirements churn later in the project.

## **Summary**

This paper has presented three patterns. These patterns provide mechanisms for detecting problems ahead of time and providing support for establishing feedback loops between the Lead and developers and the Lead and the source of requirements change. Woven through the patterns are some key themes:

- Understanding enough<sup>9</sup> detail so that the Lead can manage the team by planning ahead rather than reacting is a key skill.
- The team needs freedom (EnoughRope[1]), but the Lead always needs to ensure that this freedom does not jeopardise delivery, nor place developers in a situation where they become out of their depth. There always needs to be a safety net: understanding the detail of the project and pre-empting problems is part of that safety net.
- Everything the Lead says and does, including the body language used, has an effect on the team. The Lead needs to be cognisant of this, empathise with the team and ensure that interactions are positive and productive.
- The Lead must not shy away from Quality Assurance activities. Ensuring the output of the team is of an appropriate quality is key to providing a safety net and therefore enables both a successful delivery and an empowered and effective team<sup>10</sup>.
- Use the correct information gathering tools for the job in hand. Ensure there is a mix of “broad view” tools such as *CodeSurfing* and “detail” tools such as code inspections. Target “detail” tools and mentoring using information from the “broad view”.
- Communication between the Lead and the team is a two way street. Ensure that expectations are communicated clearly and feedback loops are in place to allow the team to respond.

Leads who utilise the above practices along with the other broad range of techniques and skills required are more likely to be successful.

## **Proto-Patlets**

This section provides thumbnails for proto-patterns described elsewhere in the paper.

*TransparentProject* – Use techniques such as BigVisibleCharts [24], clear and timely status reporting and unambiguous metrics to ensure that all stake holders can obtain the information needed to understand the project status.

*RaiseTheYellowFlag* – When you know there’s a problem, raise the flag to let everyone know – from the Motor Racing protocol to warn drivers that there is a problem ahead even though they can’t see it yet.

*CodeForCrufts* – Repeatedly grooming a piece of code that is already fit for purpose. Crufts is a British National Dog Show.

---

<sup>9</sup> The Leads needs enough detail, not too little (everything looks great from 20,000ft) and not too much (the Lead will be swamped with information and the team will be being micro-managed). See HoverShoes[1] for a complementary description of the trade the Lead has to make between detail and perspective.

<sup>10</sup> As the safety net allows delegation and freedom within the team.

## Excerpts From A Set Of Technical Leadership Patterns

*Don't Spare The Delegation* – Ensuring that as many appropriate tasks as possible are delegated to the team. Failure to do this results in an overloaded Lead and an under utilised team.

## References

- [1] Olsen, Don Sherwood & Stimmel Carol L.. *The Manager Pool (Patterns for Radical Leadership)*, Addison Wesley 2001.
- [2] McCarthy, Jim & McCarthy, Michele. *Software For Your Head*, Addison Wesley 2002.
- [3] Marquardt. *Diagnoses from Software Organizations*, EuroPLOP 2002.
- [4] Alistair Cockburn et al. *Risk Management Catalog*, <http://members.aol.com/acockburn/riskcata/risktoc.htm>
- [5] James Coplien. *A Generative Development-Process Pattern Language*. Pattern Languages of Program Design, Addison Wesley.
- [6] James Coplien. *Organizational Patterns*, <http://i44pc48.info.uni-karlsruhe.de/cgi-bin/OrgPatterns> (was <http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns>)
- [7] Ivar, Jacobson, Booch, Grady and Rumbaugh, James. *The Unified Software Development Process*, Addison Wesley 1999.
- [8] URL. <http://c2.com/cgi/wiki?GoldPlating>
- [9] Gilb, Tom & Graham, Dorothy. *Software Inspection*, Addison-Wesley 1993.
- [10] Boehm, Barry W & Papaccio, Philip N. *Understanding and Controlling Software Costs*, IEEE Transactions on Software Engineering, October 1998.
- [11] Yourdon, Edward. *Structured Walkthroughs*, Yourdon Press 1989.
- [12] Fagan, Michael . *Design and code inspections to reduce errors in program development*, IBM Systems Journal, Vol. 15, No. 3 1976.
- [13] DSDM Consortium. *MoSCoW Rules*, <http://www.dsdm.org/en/about/moscow.asp>
- [14] Dikel, David M., Kane, David & Wilson, James R. *Software Architecture (Organizational Principles and Patterns)*, Prentice Hall 2001.
- [15] Csikszentmihaly, Mihaly. *Finding Flow : The Psychology of Engagement With Everyday Life*.
- [16] DeMarco, Tom & Lister, Timothy. *Peopleware*, Dorset House 1987.
- [17] Peters, Thomas , Waterman, Robert H. *In Search of Excellence: Lessons from America's Best-Run Companies*, Warner Books 1988.
- [18] Foster, *101 Ways To Generate Great Ideas*, 2001.
- [19] Whitehead, Richard. *Leading A Software Development Team*, Addison-Wesley 2001.
- [20] Jones, Capers. *Assessment and Control of Software Risks*, Yourdon Press 1994.
- [21] Yourdon, Edward, *Decline & Fall of the American Programmer*, Yourdon Press 1992.
- [22] Schwaber, Ken & Beedle, Mike. *Agile Software Development with Scrum*, Prentice Hall 2001
- [23] Beck, Kent. *Extreme Programming Explained*, Addison Wesley 2000
- [24] McCarthy. *Managing Software Milestones at Microsoft*, American Programmer, Feb 1995.
- [25] Beck, Kent & Fowler, Martin. *Bad Smells In Code*, <http://home.sprintmail.com/~wconrad/refactoring-live/smells.html>
- [26] <http://c2.com/cgi/wiki?BigVisibleChart>
- [27] Gabriel, Richard P. *Writers' Workshops and the Work of Making Things*, Addison Wesley 2002
- [28] Weinberg, Gerald M. *The Secrets of Consulting (A guide to giving and getting advice successfully)*, Dorset House
- [29] Beck, Kent & Fowler, Martin. <http://home.sprintmail.com/~wconrad/refactoring-live/smells.html>
- [30] IEEE reference to come.