Is Your Project Out of Control?

Introduction

This guide is intended to help senior-level project managers, especially those with limited experience in managing software development projects, recognize the symptoms of a project that is in trouble. The key goal is to identify the problem.

What Is Control?

It seems obvious that to know when you are “out of control,” you must first understand what “in control” means. Obvious or not, that’s a difficult question to answer, and it may not be all that relevant. The point is not to measure the project, the people, or the management against some yardstick. Rather, it is to recognize when the team has slipped from a state of healthy, natural chaos to a point where progress is hindered by a lack of control.

As Microsoft’s Chris Peters noted, “The default state of a project is out of control.” The natural tendency is toward unhealthy chaos. Only the management of the team, with the help of the team itself, can provide the control necessary to make the project a success. This makes it difficult if you are vigilantly watching for some transition from “in control” to “out of control,” because some teams are never in control in the first place.

This document focuses on recognizing the unhealthy signs of a project that needs corrective action. It’s much more a listing of symptoms than a cookbook for project management, and much less a how-to than a diagnostic tool.

This document is divided into the major areas where projects can get out of control. They are:

- Focus
- Product
- Team
- Communication

- Process
- Schedule

Each section of the paper describes the key aspects that make the category important, gives assistance in identifying problems within that category, and occasionally offers some ideas to deal with these problems.

Finally, the document ends with an appendix that divides this list of problems into a timetable. This can be used as a checklist for managers at various points in the project, helping to identify that state of “out of control.”

Focus

Probably the most important—and the most difficult to judge—characteristic of a well-functioning team is the possession of the focus and vision needed to create an outstanding product. Without this focus, the team cannot execute on the underlying tasks that make up the project.

Lack of a Shared, Crisp Vision

The one success factor most experts agree on is the need for a project team to have a clear vision for its project.
How do you determine whether a team has that level of shared understanding?

One method that seems to work well is to pick a few individuals from a variety of areas on the team and ask each one to briefly outline the vision for the project. If each person selected cannot immediately identify, in just a few sentences, the key goals and the key customers for the project, the project is in trouble. The response to “What are we building?” should be a prompt “The fastest spreadsheet in the world,” or something to that effect.

The breakdown—if there is one—may have occurred because the project does not have a crisp vision, the vision has not been well communicated to the team, or the team does not agree with or believe in the vision. Whatever the cause, however, the lack of a shared vision is a fundamental flaw that will prove fatal to the project. Without a vision for the project, the team will be unable to make difficult feature or bug tradeoffs, will be frustrated by communication problems among members of the team, and will inevitably make decisions that are misdirected because they have no foundation.

Lack of a Customer Focus

Another way to look at the problem of focus is to ask “Who will pay good money for this product?” If the customer is not readily identifiable and is not a key part of the goals and vision for the product, this is another major sign that the product is in trouble. When difficult decisions arise, make sure that a customer advocate is identified in the group and that that person is heard.
The corollary to this is to be sure that you have identified who is *not* the customer. It is often too easy to say “Anyone would want this,” thereby avoiding giving sufficient thought to who the customer specifically is and is not.

Customers are not homogeneous. Some groups of customers will definitely be outside your market. It’s vital to identify them too.

Focus on the Wrong Things

One trap that is very easy to fall into is the tendency to focus on things that are not directly related to shipping a product. Examples range from the apparently important to the obviously inane, from an inordinate focus on preparing for conferences or project reviews to angst over the color of the T-shirt.

If the team is getting T-shirts for almost completing milestone 2, you should be concerned. If the end goal in sight is not a product for your customers, but a conference for your peers, you should consider some actions to refocus the team.

Certainly some creative extracurricular activities are vital to maintaining team morale, and they can often help to focus the team. But if they seem the persistent focus of the group, if they occupy more than just a small window of time, or if they are the subject of controversy in the group, they are counter-productive.

Product

It's often easy to look at the product itself and judge whether the project has a good chance of success. This is where an objective, third-party view—that of another experienced project manager, perhaps—can help the most.

Moving from Technology to Product

Version 1.0 of any product is more inclined to be out of control, or at least to appear that way, because the target can move or be unclear. Much of the difficulty occurs in the transition from technology to product. This transition, during which a great piece of technology gets solidified into something a customer would pay money for, is a very difficult one.

The key flag for a product that is not successfully making this transition is a team that is in love with the technology.
When that happens, team members are so enamored of the technology that they can't clearly identify who the customer is, why the customer would want the technology, and how the technology fits into a business model. They often redirect questions in this area to demonstrations of the technology or to detailed discussions of how it works.

People in love with the technology also often have a difficult time discussing version 2.0 or 3.0 of the product. They've become so engrossed in what it is today that they haven't thought about how it will be a business in the long run.

Biting Off Too Much

If you review the project with an experienced project manager and his or her response is that the project seems very difficult to accomplish, consider that a danger sign. It isn't so much that every project should be easy, but each project should be doable.

There is a tendency, especially again with version 1.0 projects, to feel compelled to make a big hit with the product. This tendency can lead to over-ambitious projects that are destined to have difficult births. Often this pressure is caused by fear of a competitor. Teams worry that they'll make an inadequate response to the competition's latest release and that their product will be a failure in the marketplace. The solution is to make a project plan that covers several manageable releases. Plan version 1.0 to build a solid foundation and establish a position in the market. Leave for version 2.0 or 3.0 those features that will make the project late or that require time in the marketplace to jell.

It may be tempting to try to perfect a version 1.0 project. However, experienced managers will tell you that, after five years, they would rather be working on version 3.0 of a product than working on the fifth year of version 1.0.

Too Many Dependencies

Dependencies on other projects are an unavoidable fact of life and are likely to increase the larger a company gets. Dependencies have their benefits, since larger companies can leverage their size and allow groups to specialize.

However, there is danger any time a project team is dependent on another team for a key functionality, and the project team needs to manage that dependency very closely. A project with most of its functionality in the form of dependencies is a project likely to be in trouble.

Under-Detailed Specifications

No project should exist without some common point of reference, and a specification is a very good example of one. At the beginning of a project, before coding has begun, the specification is the only deliverable, and the only real sign of progress in the group. Once the team is well into a project, the product becomes a more visible indicator, and the team should be able to trust it far more than any document. But at the beginning, some documentation is a must.

Over-Detailed Specification

The converse to having no specifications is having an encyclopedic specification. Communicating the project to the team and its dependents is key, but an extremely detailed specification can be a problem for a couple of reasons.

An overly detailed specification is a warning sign because it can, itself, become the deliverable. The team can get overly focused on updating the specification, on filling in every little hole, or on producing regular updates. An overly detailed specification is an indicator that the team has lost its focus on the end product and has forgotten that the key deliverable is a product for the customer.
An overly detailed specification is also problematic because it can breed overconfidence. By possessing a seemingly bullet-proof specification, a team can fail to prepare for the inevitable calamity or change in direction. The team will schedule things with inadequate buffer, will be resistant to changing the specification for all but the most dire of causes, and will lose sensitivity to the market it is trying to serve.

So how much is too much? This varies with the product, but the warning signs are fairly easy to spot. Is the specification in numbered volumes? Has the specification been read, really read, by anyone other than the author? Is the specification the review objective for the program manager, as opposed to a good feature in the product? Is updating the specification someone’s full-time job? These are not necessarily fatal signs, but they often indicate a problem.

The Vision in the Wind

Often used as the excuse for a project’s problems, but only occasionally correct, is the problem of the project whose vision changes every few months. This has been the downfall of more than one project, and is usually blamed on “management.” The reality is more often that the project never had a clear vision in the first place, or the vision failed to take into account some crucial elements (the customer, for example).

Detecting this problem is rarely difficult. Teams will usually howl in pain with each change of the vision. Changes in the individual features or minor project goals are commonplace. The time to worry is when the fundamental direction of the project changes but the shipment deadline doesn’t.

Not Managing Product Performance and Size

Features can be debatable in terms of which should be in and which should be out. But the areas of product functionality that are constant and most frequently overlooked are performance and size. Failure to pay heed to these issues will always come back to haunt the team and can be an early warning sign of a product doomed to be out of control. The product team must clearly specify performance and size goals for every significant part of the product. If the team is not managing these issues by monitoring progress toward these goals with the daily builds, it is going to be surprised at the end of the project. If the team has no time in the schedule to fix performance and size problems, the schedule will likely slip. If the team has no people to worry about this as their primary job function, the team is likely to ship a bad product. In any of these cases, the team is not in control.

Team

The caliber of the team is often raised as a key success factor. While this is relatively obvious, the signs of a problem in this area are less so.

Lack of Shipping Experience

Shipping software is a complicated art. It requires a combination of skills and experience that is fundamentally impossible to attain elsewhere. The skills can range from technical to political, and the experience can range from the smallest products to major releases. But there is no replacement for people who've lived through it. If the team does not have people—particularly senior leaders—who have shipped software in at least half of the key positions (development manager, test manager, group program manager, user education manager, and/or product or business unit manager), it is likely to have a very difficult time.

Lack of Technical Skills

An obvious source of problems is a team that does not have, or has lost, the technologists to produce the product. Key warning signs include black box technologies ("We don't want to change that code. No one knows how it works") or overly sensitive technologies ("We'll do anything to avoid touching that code. It's very delicate").

Lack of Ownership

Each core technology should have an individual who can always be turned to in that area. That person should "own" the technology or feature, and his or her ownership should be well known by the whole team and all dependents.

The owner doesn't have to be a lead or manager. He or she is often a star member of the team who understands the details and communicates them well. He or she usually feels a strong vested interest in its success, and is often the cheerleader and staunchest supporter of at least the feature, and often the product as a whole. The owner wears the label as "the person to go to" on any issue relating to that area.

Inadequate Staffing

An obvious warning sign about the health of a team is open or inadequate headcount in key areas. A team with key positions (the business/product unit manager, development manager, test manager, group program manager, or user education manager) left unfilled for a long period of time will find it impossible to set and maintain direction. A team with more than one of these roles unfilled, even for a short period of time, is in jeopardy.

A team with no testers, or even one with substantially fewer testers than developers, is likely to have a difficult time producing a quality product. A team with developers writing specifications as their key job, or with program managers writing code as theirs, is understaffed or misdirected in a serious way. Even a team with a high percentage of temporaries (greater than 20 percent) is putting a great deal of the intellectual horsepower in a volatile resource. All of these require reallocation of resources to succeed.

Hostages

Much like a team without key people, a team with key contributors who want to leave, but are being held against their will, is destined for disaster. It's often better to just admit the problem and let them go, so at least they won't drag the rest of the team down into their bad morale pit.

Communication

Communication, both inside and outside the team, is key. During a postmortem, people often say, "There was a lack of communication." It's hard to know what that really means. Here are some clues.

The Reality Distortion Field

A key communication problem within the team is fondly known as the "reality distortion field." This phenomenon occurs when a team, engrossed in its own magnificence, convinces itself that impossible dates can be met, that enormously complex technical problems are nothing to worry about, and that naysayers just aren't committed to shipping on time.

The reality distortion field occurs in varying degrees, which often makes it difficult to detect. It shows its true colors when challenges to the schedule, business model, or technology are not carefully considered but rather are discarded as the ranting of a non-believer. Often the retort is “you just don’t understand.”

The appropriate reaction is to build understanding, but if this is not forthcoming, there may be a need for some cold, hard reality. A thorough project review with a skeptical audience can do wonders toward dissipating the field.

Lack of Communication Among Dependents

The most important part of working with groups that you are dependent on, or who are dependent on you, is communication. It’s easy to understand the importance of getting the programmatic interfaces correct, or to see why planning the dates is vital, but none of this will actually happen without a huge investment in communication.

Discovering whether you have a communication problem with your dependents is actually quite easy.

Among the key questions to ask:

- When is the next drop? The answer should be unequivocal and the date should be before your next drop.
- Who in our group attends their staff meetings? The answer should be at least one of your program managers, and maybe a developer or two as well.
- How many bugs do we have on their bug list? The answer isn’t important, but someone on your team should know it.
- How do they prioritize our bugs? The answer should be “like their own.”

In any case, your project is in jeopardy if you don’t manage these communication issues very well.

Inordinate Emphasis on Secrecy

One of the clues to a team in trouble is if the team tries to hide things. Clearly some details of the technology can or should remain confidential, but a team that is overly protective of its specifications or schedule is hiding something. And it’s not likely to be something good.

Ask for a look at the specification. You don’t have to read it (but you probably should). It’s just a test to see if the team will give it to you. Ask to go over the schedule. If the team will let you, it’s probably OK. If it won’t, you should be alarmed. If you’re not on the team, maybe there are good reasons not to give you access. But it’s time to really be alarmed if you’re on the same team and you can’t get access to things. Something is wrong.

Status Reports That Blame Others

If every month the status report looks bleaker and bleaker, yet the problems are always someone else’s fault, there’s a problem. There’s little question that a team can be derailed by the failings of its dependencies, but if that is offered as a reason month after month, there is a problem. The team should be making alternate plans, falling back on its contingency strategies, doing something. But throwing up its collective hands and blaming its woes on others is a clear sign of a team that is out of control.

Process

Much of the wisdom relating to the development process is well known, but it deserves repeating. Often the difference between in control and out of control is in the details.

Failing to Use Milestones and Milestone Reviews

Most project teams divide their projects into milestones and use these milestones to track the progress of development. However, many teams also cheat on their milestones, which of course only causes problems down the line.

There are several common milestone cheats:

- Fudging the end date (“We didn’t really say the 3rd. We said August.”).
- Declaring victory (“Well, most of the team made it.”).
- Not having definitive quality objectives (“So what if we have 1,500 bugs? The code is complete.”).
- Moving things that didn’t make it into the next milestone without careful evaluation and without a complete reschedule of the rest of the project.

Teams also commonly fail to come to closure on milestones by not holding milestone reviews. These reviews, also referred to as milestone postmortems, are extremely important sources of corrective action for the next milestone. Failure to hold them, or to reset the schedule during them, is a serious warning sign.

Lack of Regular (Daily) Builds

Time has shown the daily build of the product to be one of the most important diagnostic tools for the health of the product. Having a daily build does not ensure the health of the project, but few well-run projects survive without one.

Key to the daily build are the daily sniff tests (or smoke tests). Stolen from the electronics industry where people would plug in a board and see what smoked (or smelled bad), sniff tests are a vital part of the build.

These tests ensure the basic stability of the product by checking daily that nothing has been checked in that jeopardizes the basic product. If something is uncovered, it is repaired immediately. This provides a foundation for the new work to be added each day and ensures that the project doesn't stray too far from a stable working version.
Build sniff tests to be run by the developers before checking in code. This prevents breaking the daily build and also encourages developers to take more responsibility for the stability of the product.
The importance of a regular (preferably daily) build cannot be underestimated. A good way of thinking about this is to consider how much time it would take to recover from a fatal flaw in a check-in. If you build weekly, and someone checks in erroneous information on Monday that you don't find until you build on Friday, you might have to repair a week of work before health is restored. You've lost two weeks. If you built daily, you would lose only two days.

It's also important to start daily builds very early in the process, preferably from day one on new versions of existing projects. Think of it as getting the project to a known state and ensuring that it stays there.

Failing to Eat Your Own Dog Food

Eating your own dog food—that is making your team run, live, and breathe its own product—has been shown to be a vital method of ensuring the health of a project. Some people claim that this is only viable with applications that people use in their daily lives (such as the operating system, or a compiler), but the reality is that you can make daily and frequent use of your product a key goal for every team member.

You can eat dog food in a variety of ways. Have a database? Use it to track your bugs. Have a spreadsheet? Use it for your schedule. Have a game? Make sure people are using it daily, taking it home, and sharing with their friends.

Eating your own dog food is so important that some teams can actually judge the ship date from the day they first start seriously doing it. For example, operating systems, some feel, are a year from release when dogfooding begins. Whatever your metric, if you're not doing it, you're in trouble.

Failing to Track Metrics

It's easy to get carried away with tracking bug counts, lines of code, person-hours per bug fixed, and on and on, but some attention to metrics can help a great deal.

The team should pick some metrics to track and should widely publish them regularly, at least monthly, but preferably much more frequently. This can be bug arrival rates, bug closure rates, or whatever, but tracking the process is essential.
The key to metrics is to track something meaningful.

To do this, you should:

- Decide the overall goal that you want to accomplish. This could be as straightforward as "cutting priority 1 bugs," or as complex as "increasing customer satisfaction."
- Determine a metric that will display whether you are meeting your objective. Define what it means if the number goes up, what it means if the number goes down, and what you might do about either case.
- Track the metric, report it on a regular basis, and review your performance using it.
- Change how you are working if you aren't meeting your objective. If you can't envision this happening, you're tracking the wrong metric.

You want to avoid tracking metrics for their own sake or tracking things that don't optimize your team's behavior or measure your progress toward shipping. Tracking the wrong thing can be worse than tracking nothing, because you may lure yourself into a false sense of accomplishment.

Missed Transitions of Responsibility

There is a "center of gravity" for projects that can be observed fairly easily, and the program management team should be following that center as it moves over time. This center begins in program management as the team designs the project and designs a specification for it. It then moves to development as developers code the project, and then to test as testers help to polish it for shipment.

The PMs on the team should be making the transitions with the center. As the focus moves to the coding team, program management should be working to close down issues, triaging bugs, and responding to difficulties the development team has in implementing the specification. As the focus changes to test, program management should focus on triaging bugs and preparing the product for shipment. If you find the PMs worrying about new features after code complete, or planning the next version instead of closing down bugs, they are not helping to ship the product. Either they need to be part of a separate team focused on the next release or they should be more closely working on the current release.

Handling Process

Few teams truly handle all aspects of their process well. This is not a fatal flaw, but it can be used as an indicator of the maturity of the development team. There are many fine points that should not be overlooked. Here are some examples, not that your team *must* follow, but that rather are used by many teams:

- After code complete, all check-ins must have a raid number.
- After some point in the process, every check-in must be reviewed by at least one other person (some teams require that it be a lead).
- No bug is dismissed as “non-reproducible” without consent of the test manager/lead.
- No bugs are dismissed en masse. Every bug gets a fair and honest hearing. Teams that blow off huge numbers of bugs often regret it, or read about it in the trade press.
- After some point, all feature/interface changes must be approved by a “change control group.”
- If the number of bugs exceeds the number that the development team can fix in a reasonable period (for example, 10 days), stop work on new features while the development and test teams work on fixing and closing bugs.

These are just examples, but your team should have a list of these kinds of rules, and it should be following them.

Schedule

Non-Bottom-up Scheduling

One of the more reliable predictors of the health of a project can be to determine where the schedule originated. If the schedule came from anywhere other than the people who will actually do the work (with appropriate buffering applied), it is wrong. Period.
Common places for a bad schedule to originate include:

- Marketing, when it is allowed to determine when a product “has to ship.” While marketing offers valuable input into the product development process, a date-driven milestone is always the source of problems. If it is coupled with explicit features, as it usually is, it is a recipe for disaster. The only three variables are time, resources, and product. You cannot restrict all three variables of the equation.
- Management, when it overrules the team and decides what the schedule “should be.” While management is another valuable source of input into the process, it cannot successfully dictate the schedule. Management *can* greatly assist by helping to adjust resources, priorities, and other aspects of the process, but it should not force a schedule on the team.
- Program management, when it moves beyond its charter for removing obstacles to shipment and being responsible for much of the shipping process, and when it mistakes its role for that of schedule-setter. Unfortunately, as with other people who are not actually performing the task, program management’s estimation skills usually fall short.
- Anywhere else. Even if the schedule was prepared by the responsible managers and leads, it’s suspect. The schedule must represent the commitments of the people who are going to do the actual work.

Regardless of who creates the schedule, if the source is not the team (the developers, testers, writers, etc.), the schedule is wrong.
How do you determine the source of the schedule? Ask several line team members what they think of the schedule, one on one. If they agree with it, and have honestly bought into it, they most likely had a hand in creating it. If they scoff, roll their eyes, or refuse to discuss it, the schedule is invalid.

Magic Dates

A warning sign of a bogus schedule is magic dates. Such dates tend be in two forms, the “when do you need it” form, and the “would you look at that” form.

The “when do you need it” form of magic date is the schedule date that miraculously meets the date management said it needed it. To be fair, the team could have carefully manipulated the resources, features, and teamwork to make the date work out. But even the best laid plans result in a schedule that only closely approximates the actual “needed by” date. More than likely what has happened is that the team has felt the need to acquiesce to what it feels is a management-driven date, and it has thrown its estimates away.

The “would you look at that” form of magic date is the schedule for several disparate components that all miraculously converge to be completed on some date. Realistically, even under the best of circumstances, convergence of a number of components will happen at only roughly the same date. If you find yourself looking at a schedule where the components all magically make the same date, you should be very suspicious.

Schedule Not Detailed Enough

If the schedule is in increments longer than a week (five work days) per person, the schedule is not detailed enough. Letting people run without a schedule checkpoint weekly is risking their being off by as much as a week. Compound that by all the people on the project, and the schedule is far too fuzzy.

This is such a key issue that some people insist on even smaller increments (2 to 3 days). Whatever granularity you choose, the key is to ensure that the schedule is accurate to the amount of time you’re willing to slip.

Schedule Churn

Somewhat the converse of a team that schedules too little is one that is overly fixated on the schedule. Clearly, some attention to the schedule is very important. However, do not lose sight focus of the schedule’s purpose in facilitating product completion and ultimately product shipping.

The team that over schedules tends to release new versions of the schedule very frequently (two or more times per week). The schedule tends to be too detailed, listing people’s tasks by the hour or some other fine granularity. The team tends to make adjustments to the entire project schedule that are in very small increments, such as days. In all, this indicates a lack of focus on the product and too much focus on the process.

The other serious danger of schedule churn is “crying wolf.” Frequent date changes get the team, the managers, the customers even, all believing that you are out of control. They won’t believe any date you give, and they are justified.

Lots of Items Assigned to TBH or TBD

Assigning tasks to someone TBH (to be hired) or TBD (to be determined) are key warning signs that a schedule is fictitious. Hiring takes time, so assigning a task to someone TBH is risky. Unless some sort of magic occurs, that someone will be hired and gotten up to speed too late to be useful.

TBD is a sign of one of several problems: inadequate resources, inadequate planning, too much product, or not enough willpower to cut features. In all cases, this is a dangerous warning sign.

How do you know when this is a problem? Are key features or bugs assigned to TBH/TBD? (One of the more common areas that gets this treatment is Setup, the first thing every customer sees.) Is more than a tiny percentage (less than about 1 percent) of the project unassigned? Is TBH/TBD used for more than a month as placeholders? These are signs that the project may be in trouble.

Two Sets of Books

Some managers feel the need to lie to their teams about the “real” date. The manager has in his or her head the date he or she expects the work to be done, and may even tell the boss this date, but will tell the team some earlier date. Managers excuse this blatant abuse of trust by insisting that it makes the team work harder. “They wouldn’t come in on weekends or really bust it if I told them the real date.”

This is simply horrible management. You can’t trust the team with the truth, but you’re willing to trust the team with the product? It makes no sense.

It has been proved time and again that teams can and do respond well to liberal doses of the truth. Explain the details of the schedule, define the pressures on the team, involve team members in the decisions, and track the progress daily with them. Every team dealt with in this way responds like the collection of professionals it is.

Buffer Misuse

Buffers are put in place to allow time to fix bugs or for performance and size problems (which are just a class of bug), not for completing features that are running late. Using up the buffer for feature work, and then saying “we made the milestone,” is just lying to yourself (see the previous discussion of milestones). When will the bugs in those features be fixed?

The team should readjust the schedule when new items are added or items take more time than allotted. They should not just eat up the buffer. Again, the buffer is for bugs and issues only.

Failure to Schedule Busy Work

Forgetting to schedule such things as holidays and vacations can be a real problem. Even important tasks like documentation reviews, code reviews, and group meetings can get left off the schedule.
People will often pass over these things as “just part of the buffer,” or something to that effect. That’s not a good idea. There will be enough things that just “come up” to require that you schedule anything you can anticipate.

We’ll Make It Up

This is a comment commonly heard at or near the completion of a milestone. It’s usually preceded by “That’s OK…” “Making it up” has never happened in recorded history (at least not when “it” is anything substantial). There’s no reason to believe your team will be the first. Spend the time to completely re-evaluate the schedule, from the bottom up, and come up with a whole new schedule. No other remedy is safe.

Fix—If There Is Time

A parallel problem to “we’ll make it up” is bugs assigned to “fix—if time.” The bugs will just sit on the list, there will never be time, and they will eat away at you, adding to that massive bug count that is making the team members sick to their stomachs. Be honest with yourself. Decide to fix the problem or decide to postpone it. Make a decision.

Sometimes this problem occurs because a team has not decided what the quality bar is and can’t decide whether the bug is something that needs to be fixed. Setting this bar early helps provide a consistent tone to the whole project and increases the chances of success.

Summary

Few teams will exhibit all of these symptoms, and the exhibition of one or two does not necessarily spell doom for the project. The way to use this list is to be as objective as possible, and use it as a diagnostic tool to determine the health of your team.

If you feel that the team is out of control, you have many options. You can and should take prompt action, either on your own or with assistance. The key thing is to do something. Remember, entropy favors the state of “out of control.”

Out of Control Checklist

This list is the same as the previous list, but presented in chronological order.

Specification Complete Milestone

Especially important when coding is about to begin are:

- Lack of a shared, crisp vision.
- Lack of a customer focus.
- Moving from technology to product.
- Biting off too much.
- Too many dependencies.
- Under-detailed specifications.
- Over-detailed specifications.
- Not managing product performance and size.
- Lack of shipping experience.
- Lack of technical skills.
- Inadequate staffing.
- The reality distortion field.
- Failing to use milestones and milestone reviews.
- Non-bottom-up scheduling.
- Magic dates.
- Schedule not detailed enough.
- Lots of items assigned to TBH or TBD.
- Failure to schedule busy work.

Coding or Code Complete Milestone

During coding, at coding milestones, or at code complete, watch for:

- Focus on extracurricular activities.
- The vision in the wind.
- Biting off too much.
- Too many dependencies.
- Not managing product performance and size.
- Lack of ownership.

- Hostages.
- Lack of communication among dependents.
- Inordinate emphasis on secrecy.
- Status reports that blame others.
- Failing to use milestones and milestone reviews.
- Lack of regular (daily) builds.
- Failing to eat your own dog food.
- Failing to track metrics.
- Handling process.
- Schedule churn.
- Lots of items assigned to TBH or TBD.
- Two sets of books.
- Buffer misuse.
- "We'll make it up."
- "Fix—if there is time."

Beta or Release Candidate Milestone

In the final stages, be especially sensitive to:

- Lack of a customer focus.
- Focus on the wrong things.
- Not managing product performance and size.
- Hostages.
- Lack of communication among dependents.
- Failing to track metrics.
- Handling process.
- "Fix—if there is time."