Practical Tips For SOFTWARE-INTENSIVE STUDENT PROJECTS

3rd Edition

Damith C. Rajapakse

Contents

CHAPTER 1: INTRODUCTION

One principle problem of educating software engineers is that they will not use a new method until they believe it works and, more importantly, that they will not believe the method will work until they see it for themselves. --Humphrey, W.S., "The Personal Software Process"

This book is intended for students who have to do a software-intensive team project as part of their course of study. Most students find it hard to apply theories they previously learnt in the context of an actual project. This book is an attempt to help such students. It is a collection of bite-sized practical tips gathered from my software engineering experience (both as an educator and a practitioner) and from many excellent books and articles on the topic.

This book is **free for use** by anyone. Here is how you can use it:

> **Students**: Use this book to guide you in your project. It might not fit your project exactly, as project courses differ widely. If it does not, do let me know; I shall try my best to address the mismatch in a future version.
>
> **Educators**: This book can be used as supplementary text for your course. If you are an experienced educator, there are probably many tips you could add to this collection. In that case, please let me know by adding a comment [http://tinyurl.com/tipsbook-feedback] or emailing me. If you would like to co-author a customised version of this book to fit your own course or to translate it to a different language, please drop me an email.
>
> **Practitioners**: Although not specifically targeted at practitioners, this book can help newly recruited programmers. More importantly, as practising software engineers of today, you could help us educators do a better job of training tomorrow's practitioners by giving me feedback. Just by add a comment here or email me.

If you would like to have **a printable PDF version** of the entire book, please let me know using this page [http://tinyurl.com/requestpdf]. Usually, I can send you the pdf file within a day. If you want **to reproduce some of this content** elsewhere - that can be arranged too; please let me know in advance.

URL to add comments: http://tinyurl.com/tipsbook-feedback | **Author's email:** damith{at}comp.nus.edu.sg

Many helped to make this book what it is today. In particular, I wish to thank (in no particular order):

- For advice on various aspects of project courses : Associate Professor Stan Jarzabek, Associate Professor Khoo Siau Cheng and Dr Ben Leong from National University of Singapore, Professor John C. Knight from University of Virginia, Professor Scott Tilley from Florida Institute of Technology.
- Colleagues with whom I taught project courses: Bimlesh Wadhwa, Soo Yuen Jien.
- Dear past students who gave feedback on the book. I'm touched that you volunteered to help me when you are no longer studying under me: Ankit Tibrewal, Bryan Chen, Ho Man Boa, Roshan Ananad, Rushi Padhuman, Savitha R., Yuan Chin, Zi Vvq Huong.
- Brayan Chen (again!) for volunteering to do the cover design, which turned out to be great.

- For listing this book: Free Tech Books, Online Computer Books, PC Book Review, eBookBay, Subject Books
- Authors of the great books/articles referenced in this book; You made this task so much easier.
- Those of you who read this book, and recommend it to your students/friends.

Just for the record, this is the book's progress:

- May 2010: 3rd edition, renamed as "Practical Tips for Software-Intensive Student Projects"
- July 2008: 2nd edition, renamed as "Tips to Succeed in Software Engineering Student Projects"
- June 2008: 1st edition, named "A Student's Guide to Software Engineering Projects"

CHAPTER 2: PREPARING FOR THE COURSE

The best way to get a project done faster is to start sooner --Jim Highsmith

Here are some tips for those early birds who want to prepare for the course even before the term starts.

2.1 FIND OUT WHAT TO EXPECT

Talk to those who have taken the course before, preferably with the same course co-ordinator. Learn what the course *really* requires.

Where necessary, contact the instructor and verify your assumptions before you act on them. Just because it was conducted in a certain way last semester (or, for so many previous semesters), does not mean it will be the same next semester. For example, the course may have used C++ for last so many semesters; but before you jump into learning C++, check with the instructor whether it will be C++ again next semester.

Most lecturers upload materials to the course website in advance. Monitoring the website is especially useful if there is a "first come, first served" list of potential projects for you to pick.

2.2 CLIMB THE LEARNING CURVE

Spend some time revisiting course materials of software engineering theory courses you took before. You will not have time for this when the project is in full swing. After brushing up on your theory, you can look for some 'practice-based' materials. A good place to start is the book **The Pragmatic Programmer: From Journeyman to Master**.

If the course requires you to program in a certain programming language, start learning it now! Practise as you learn it, as the best way to learn a language is to use it.

Find out the "must use" tools for the course. Start learning them early. Before downloading the latest version of a tool, verify with the instructor whether you are required to use a specific version. Courses often force students to use an earlier, more stable version of a software tool.

2.3 BRACE FOR THE WORKLOAD

Plan for other courses based on the *real* workload of the project course. Often, project courses carry more credits than a regular course, resulting in higher workloads than regular courses. Even if there are no extra credits involved, project courses often imply heavier workload than regular courses. Ironically, some students imagine it to take less time than a regular course ("we can do it over the weekend") - a mistake they will realise only when it is too late.

2.4 FIND YOUR TEAMMATES

If the course allows you choose team members for your team, find suitable team members soon (see chapter 5 for more hints on choosing team members). While the instructor may downplay the issue by saying "it is not *required* to form groups early" or "you will be given enough time to form teams", it is in your best interest to find suitable team members as early as you can. They are a scarce commodity.

CHAPTER 3: MANAGING REQUIREMENTS

Some project courses give a product specification for students to implement while other courses let student choose what to implement. Some projects have actual external/internal clients while others let students develop a product for an assumed potential market. Whichever the case in your course, at least some of these tips can help you manage the requirements of your project.

3A. PROPOSING A PROJECT

This section is useful if your project course allows you to define your own project.

3.1 CHOOSE A WORTHY CAUSE

The hardships of the project will be much easier to bear (and the final result much more satisfying) if you build something that you all believe in. Do not choose something that will simply fulfil the course requirements with the least amount of effort. For example, if you are building a utility that will solve a problem for someone, you should be convinced that it is a real problem worth solving. If it is a game, you should feel like it is a fun game to play and many others will enjoy playing it.

For the same reason, do not opt to duplicate something that has been done before unless you plan to do it in a novel way or achieve a better outcome than the previous efforts.

Having chosen a product, continue to have faith in it. The battle is lost the moment you stop believing in your product; do not start a battle that is lost even before it is started.

3.2 KNOW YOUR MARKET

The trouble with most good ideas you have is that many others had the same idea long before you. If you have very little experience in the chosen product domain, it is helpful to do an extensive examination of all similar products in the market. Well, at least Google around a bit before you declare your idea 'ground breaking' or 'novel'. However, do not plagiarise things from existing products.

Note that the lack of or low level of competition can be the sign of a good opportunity or of a bad product idea. Sometimes, there is a reason why nobody else is doing it.

3.3 TARGET REAL USERS

Not many project courses allow students to build software for a specific customer, especially if the customer is a commercial entity. However, most will allow you to release the software to the public. In either case, having real people using your software and giving you feedback can be really motivating and useful at the same time.

If possible, choose to build something that you and your teammates can use so that you have some real users close at hand all the time. It is pretty hard to second guess user needs for a product that you cannot relate to, and the result would be rather inaccurate too.

While your course might not allow external clients, you might still be able to have internal clients from within the university. Check with professors in your university; most have software development needs that cannot be satisfied by their own research students.

3.4 DREAM 'GRAND', PROMISE 'DOABLE'

It is helpful to have a grand vision for your product. For example, rather than 'just another *foo*', you can go for something like 'the most streamlined *foo*', 'the most versatile *foo*', or 'the best *foo* on the platform *bar*'. The course duration may not be long enough to achieve this grand vision entirely, but it is still good to have such a vision. However, you should also promise to deliver a version of your product that makes significant progress towards your grand vision and yet is 'doable' within the given time constraints. If you are targeting real users, this version should be useful by itself to at least a sub-section of those users.

The idea here is that you should contribute something useful towards your grand vision before the course ends, and hopefully you will be motivated to continue the project towards your grand vision even after the course. Did you know that many successful products have roots in class projects?.

3.5 NAME IT WELL

Many students underestimate the benefits of a good product name. This is especially important if you plan a public release of the product. Here are some things to consider.

- The name should clearly convey what your product is, and if possible, what its strength are.
- The name should make sense to your target market, not just you.
- If you have long-term plans for the product, check if there is already another product by that name. (Did you know that Java was first named 'Oak'? They had to rename it because of an existing little-known language by the same name.) You can even check if the name fits well into a web domain name and that the web domain name still available. The '-' saved www.experts-exchange.com from having a totally inappropriate domain name; but you might not be so lucky.
- Be wary of using team member initials to form a product name. It won't mean much to potential users. The product is the main thing; not you.
- Be wary of difficult-to-spell/pronounce names, alterations of existing names, and names resulting from private jokes in your team; users may not find those amusing.
- Be aware that names starting with 'Z' will end up at the bottom of any alphabetical listing.

3.6 SELL THE IDEA

If the course requires you put in a proposal for approval, take note of the following:

- Make these things clear: What is the problem you are solving? Why is it worth solving? What is your solution (grand vision, as well as what you plan to deliver)? Why is it better than existing solutions?
- Do not make it too long. The evaluator should be able to make up his mind from the first 1-2 pages of the proposal. The rest is just for the sake of completeness.
- It helps to include a UI prototype, a high-level project plan, and a high-level architecture.
- If the product involves solving a difficult technical problem, include some preliminary ideas on how you plan to solve it (e.g. which algorithms you are planning to try out). It shows you know what you are getting into.

3B. DEALING WITH PRODUCT FEATURES

3.7 PUT VISION BEFORE FEATURES

Combining a multitude of good features does not automatically give you a good product, just as duct-taping a good camera to a good phone [http://tinyurl.com/cameraphonephoto] does not give you a good camera phone. Resist the temptation to make up a product by combining feature proposals from all team members. The vision for your product is the first thing you should settle because it dictates the features for your product. Defining a vision after deciding features is like shooting the arrow first and painting the target where it landed. It is very important that you are clear about the vision, scope and the target market for the product. These should be documented and made available to others in the team, instructor and any other stake holders.

3.8 FOCUS FEATURES

Given the tight resource constraints of a student project, you cannot afford to be distracted by any feature that is not central to your product's vision. Ditch features that do not match the product vision, no matter how 'cool' or 'useful' they are.

Do not provide features because you think they 'can be given at **zero cost**'. There is no such thing. Every feature has some cost. Include a feature only if there is a better justification than "it costs nothing".

Some teammates might try to squeeze in their '**pet features**' for their own enjoyment. Reject them gently but firmly; if 'password security' is not essential to your product vision, do not add it even if one team member claims he already has the code for it and it can be added in 'no time'.

Aim in **one direction**. Avoid having multiple 'main' features that pull in different directions. Choose other 'minor' features in a way that they strengthen this 'main' feature even more. One 'fully baked' feature is worth two 'half-baked' ones.

Do not feel compelled to implement all '**commonplace features**' of similar products before you move to those exciting features that will differentiate your product. A product can survive with some 'glaring omissions' of common features if it has a strong 'unique' feature. Take note that the successful iPhone did not have many 'common' features such as 'SMS forwarding' when it was first released.

3.9 CATEGORISE, PRIORITISE

Categorise and order requirements based on criteria such as priority, urgency, cost, complexity, utility, risk, type of function and type of user. You won't have time to implement them all. In fact, there will not be enough time to do even the ones you thought you could. It is easier to make optimal keep-or-drop decisions if you already have your features categorised and ordered. Besides, having such a list can earn you extra credit because it shows how systematic you were in selecting features to implement.

3.10 FOCUS USERS

Larger targets are easier to hit but harder to conquer. It is better to target your product for a well-identified small market (e.g. faculty members of own university) than a vaguely-defined but supposedly larger market (e.g. all office workers). To take this approach to an extreme, it is even acceptable to identify a single real user and cater for that person fully than trying to cater for an unknown market of an unknown number of users.

It is critically important that you use the users' perspective when deciding what features to include in a product. What is 'cool' to you may not be 'cool' to users. Do not choose 'technically challenging' yet 'harder to implement' features if there is no significant benefit to the user.

3.11 GET EARLY FEEDBACK

Get feedback from your client and from your supervisor. If you do not have a specific client, find a potential user who is willing to give you feedback. Force yourself to use the software; if you do not like it yourself, it is unlikely someone else will.

You can use requirements specifications or an early prototype as the basis for soliciting feedback. Another good strategy is to write the user manual very early and use that to get feedback.

Do not be afraid to change direction based on early feedback, when there is still enough time left. That is precisely why you need to get feedback *early*.

3.12 GENTLY EXCEED EXPECTATIONS

Plan to deliver a little bit more than what is generally expected or previously promised. This will delight the evaluator. Conversely, delivering even slightly less than previously promised creates a feeling of disappointment, even if what you managed to deliver is still good.

3C. MANAGING PRODUCT RELEASES

This section is especially useful if you are releasing your product to the public.

3.13 PUBLICISE YOUR PRODUCT

If you do a public release, try to spend some effort publicising your project among potential users. Note that most evaluators will not give extra marks for such efforts; but it is still worth doing. There are many websites, such as softpedia.com, cnet.com, and softsea.com that will host your software for others to download, rate your software, and even write reviews for your software. Here [http://tinyurl.com/synchsharpreview] is a sample review for a software product written by students. If your software is good, even popular tech blogs such as lifehacker.com and addictivetips.com will start publishing reviews, boosting your download count even more. Here [http://tinyurl.com/synchlessreview] is such a review for a software written by students. It is helpful to have a website for your product, and even your own domain name. You can easily buy a domain name for less than $10/year. There are many ways to create a website for free (e.g. using Google Sites). Project hosting environments such as Google Code Project Hosting (GCPH), too, will let you create a project website. For example, http://syncless.info is the home page of a sample student project hosted on GCPH.

Other ways to publicise your project:

- Create a Facebook fan page.
- Add a link to your product in your email signature.
- Post in forums (if allowed by the forum).

3.14 RELEASE GRADUALLY

It is not a good idea to throw your very first release at ALL your potential users. Releasing it gradually gives you more opportunities to refine your product to fit the market. For example, you can do an alpha release among your friends and get some friendly feedback. After that, refine the product and do a beta release to a small community such as posting in a public forum that you participate (but make sure the forum allows such posting, or you will be flamed for spamming). Next, refine it further before you go for a full public release. If time permits, and you are allowed by the course structure, do all these releases *before* the final project submission because the feedback you get from these releases will make your final submission all the better.

3.15 NURTURE EARLY ADOPTERS

Carefully nurturing early adopters is important to a new product as it is important for a budding leader to nurture early followers [http://tinyurl.com/shirtlessdancing]. First of all, make sure they have a way to reach you. You can either build a 'give feedback' feature into the product or use an indirect mechanism such as Google Groups.

If possible, create a mechanism for current users to be notified about new releases. This way, you get a second shot at those who did not like the early release they tried. For example, you can build a 'check for updates' mechanism into the product, use the 'follow us on facebook/twitter' technique, or collect their email addresses so that you can email them about updates.

3.16 ACT PROFESSIONAL

You are a developer of a real product; act like one. Get out of your 'still a student' mentality. All public-facing materials should be professional and should inspire confidence in potential users. In my opinion, declaring the project a school project does not inspire confidence in potential users. Yes, people will be impressed to see how much you have accomplished even as students; but they will hesitate to become a real user of your software. Therefore, think twice before putting references to course-related matters in public-facing project materials.

3.17 PLAN MAINTENANCE

It would be a pity to let your product die off after the course ends, especially if you already have some real users using it. Being 'one of the founders' of an existing product can be a real boost to your resumé. But more importantly, knowing that something you produced is being used by others is a really satisfying feeling. Therefore, try your best to keep the product alive after the course. However, note that the time and energy you can spend on the project will most likely be reduced after the course. Plan accordingly. Be wary of undertaking drastic changes to the product; rather, concentrate on sustaining and perfecting what you have. Also note that some project members might not continue with you and some may not be as available as before. This means there should be more than one person who can take care of a particular aspect at this stage. The project should not fall apart just because one of the team members dropped out of the project after the course.

CHAPTER 4: MANAGING PROJECT RISKS

Attack major risks early, and continuously, or they will attack you
--Tom Gilb (in Principles of Software Engineering Management)

4.1 ATTACK RISKS EARLY AND CONTINUOUSLY

While project courses usually have in-built risk mitigation measures, this should not stop you from applying your own. It is important that you identify risks early, analyse them and continuously try to mitigate them.

Things you will have to consider include:

- What can go wrong? i.e. the risk *identification*
- What is the likelihood that it would go wrong? i.e. the risk *probability*
- What are the consequences? i.e. the *impact*
- How can we avoid things going wrong? If things do go wrong, how do we minimise the impact? i.e. *mitigation* strategies
    - What are our alternative risk mitigation strategies?
    - What are the trade-offs between those alternatives?

This section offers tips on how to manage common project risks, explained in rather general terms. Note that there will be additional *project-specific* risks that you need to manage. Obviously, mitigating risks has its own rewards. With that said, some evaluators give additional credit for explicitly tackling risks.

4.2 DOCUMENT RISKS

It is advisable that you document your risk mitigation measures, and include them in the project documentation/presentation, even if it was not asked for. You can include the following information when documenting risks:

The risk: What problem do you anticipate? What is the probability, and the impact of the risk (you can use a rating system here, e.g. rate risks 1 to 5).
Mitigation strategy: How do you plan to mitigate this risk? Who will be in charge of monitoring this risk?
Actual response (to be filled in later): Did the anticipated problem actually materialise? How did you react to it? Why did your mitigation strategy fail?

'HOW TO' SIDEBAR 4A: HOW TO TACKLE COMMON PROJECT RISKS

4.3 BUILDING THE WRONG PRODUCT

Risk: You may not understand the project requirements correctly, and waste time developing the wrong product.

Mitigation: Early, and continuous *validation* is critically important. If a specifications document has been given to you, discuss doubtful parts of the specification with every team member who will be affected by how you interpret that part. If in doubt, do not assume, always clarify with the instructor. You would not believe how many bug explanations start with "Oh, we assumed ..."
If the project was defined by you, you need to establish a clear vision for the product, ensure this vision is shared by everyone in the team, and have a solid justification as to why this vision will produce a successful product. If the product is targeted for real users, you can validate the product concept by using small-scale early releases. If the product has a specific client, you should continuously get client feedback to ensure the validity of the product.

4.4 PROBLEMATIC TEAM MEMBERS

Risk: Some team members might fail to contribute as expected, or behave in other ways detrimental to the project.

Mitigation: Be aware of such potential behaviour, make the culprit aware of how his behaviour is hurting the team, and take remedial actions as soon as possible (see chapter 5 for more on this topic). Clearly assign responsibilities to each member (you can use the *guru* concept here - see tip 9.14). Implement ways of tracking each teammate's contribution. For example, you can use a GoogleDocs spreadsheet for everyone to enter project work they did and the estimated time it took.

4.5 LACK OF SKILLS

Risk: Some members might lack the skills required for the project.

Mitigation: While external resources (tutorials, books, etc.) and lectures/labs might help here, it is also important for the whole team to work together so that you can help each other when needed. For example, if you do your early coding together, you can help each other to master the basic usage of the programming language and other tools. You could also try pair programming [http://tinyurl.com/PairIntro] to get each other up-to-speed, to get used to each other's way of work, and learn tips and tricks from each other.

4.6 SUB-STANDARD SUB-GROUPS

Risk: One sub-group might do an inferior job, bringing the whole team down at the end.

Mitigation: Try to balance the sub-groups in terms of skills, commitment, etc. You can also enforce cross-testing (i.e. one sub-group does the system testing for the other sub-group).

4.7 MISSING THE QUALITY BOAT

Risk: You might not achieve the required quality levels.

Mitigation: First, be clear about which qualities are expected, and the relative importance of each quality. For example, correctness may be the most important quality, and performance could be the second most important. Second, try to build these qualities into the product from early on, rather than as an afterthought. For instance, allocate time/resources for correctness/performance testing from early stages of the project.

4.8 GOING DOWN THE WRONG PATH

Risk: Your design might not work as expected.

Mitigation: Try it out sooner rather than later. E.g. you can try out uncertain parts of the product (perhaps, on a smaller scale) during the prototyping stage.

4.9 OVERSHOOTING THE DEADLINE

Risk: You might overshoot the deadline, and fail to deliver a working product.

Mitigation: Have a working product at all times. This mitigation measure is already included in iterative software processes. If you overshoot a deadline while using an iterative process, you can always submit the most recent intermediate product, which should support at least some of the expected functionality. The effectiveness of this risk mitigation strategy depends on how short your iterations are. If you did weekly iterations, overshooting a deadline will mean at most you miss submitting the work scheduled for the last week, which should not be much.

4.10 UNEXPECTED DELAYS

Risk: Unforeseen events (such as a team member falling ill/ dropping out) could jeopardise your "perfect" project plan.

Mitigation: Build "buffers" into the project plan. Do not rely heavily on a particular team member. While a team member might have his own part of the system to work on, it is a good idea for at least one other member to have *some* idea about that part. If the original author is unavailable, the other member can fill in as required.

4.11 BAD ESTIMATES

Risk: Certain tasks may take longer than expected i.e. because of your unfamiliarity with tasks of that nature, you are not sure whether you allocated enough time for it.

Mitigation: Try to do such "unfamiliar" tasks earlier in the project, rather than later (it is said that when forced to eat frogs, one should eat the biggest frog first). At least, attempt a similar task on a smaller scale during prototyping stage so that you get a better idea of how much time to allocate to it.

CHAPTER 5: WORKING AS A TEAM

You can't have great software without a great team, and most software teams behave like dysfunctional families. --Jim McCarthy

Unlike a typical project team in the industry,

- Your team is a team of peers and a team of novices. While your abilities may vary, none of you is experienced software engineers or managers.
- The size of your team is usually fixed and small.
- You cannot usually hire and fire team members during the project.
- You cannot outsource work.
- You have no access to other expert groups such as QA team, Graphics team, Technical writers, Business Analysts, Legal team, etc.

Here are some tips to help you overcome those limitations.

5A. TEAMING UP

5.1 THE TEAM MATTERS. TAKE IT SERIOUSLY.

If you are allowed to choose your own team members, do not take it lightly. Try to assemble the best team you can - it certainly doesn't hurt to have a great team.

5.2 EXERCISE 'DUE DILIGENCE'

When someone offers to join your team, try to find out whether he has done a good job in previous course projects. Some even hold interviews or ask applicants to answer a questionnaire before selecting team members. You can also check for things such as the workload, extra curricular activities, and external commitments that person is planning to take. This is to ensure that he can commit enough time to the project. Schedule and location compatibility matter, too. For example, teamwork is easier when team members have similar schedules and live close by (e.g. in the same student residence). It is also ideal if all team members have similar levels of expectations on what they hope to achieve; things do not go well if one person is trying to score an A+ while for another a B- is 'more than enough'.

Be mindful that some students are committed, loyal and pleasant to work with, but lacks the 'hard' skills required for the project while some others may be 'top of the class' high fliers lacking in the 'soft' skills necessary for team work.

5.3 BALANCE SKILLS

Most projects require a good combination of various skills. Form your team accordingly. If the software you will build requires a good GUI, try to find someone with good GUI skills. If it is going to be a game, it will be good if at least one person in your team is a 'gamer'. In particular, you need NOT pick only those with good programming skills. Still, programming skills are critical to the project - try to pick at least one or two good programmers.

5.4 LOOK FOR TEAMMATES, NOT BUDDIES

A team out of your best buddies has many advantages, but there is a downside, too. For instance, you will find it hard to maintain a professional atmosphere in your project work. Furthermore, you may be forced to put up with low contributions from some of your team members for the fear of hurting the friendship.

Unfortunately, we have no advice on how to dodge friends who want to join your team for a free ride, except to say that they are probably not worthy of your friendship.

5.5 ASSIGN ROLES

A project needs expertise in different areas, including areas none of you are expert in. Assign at least one team member to become the 'guru' of each such area. The guru's role is to specialise in a given area: to learn the area in-depth, help other team members on matters related to that area, and ensure sufficient quality in your product with respect to that area. For example, the testing guru is responsible for tasks such as looking for good testing tools, defining testing policies, test planning, helping others in writing test cases, troubleshooting test tools, ensuring adequate testing is being done, etc. Put differently, it is the testing guru's responsibility to earn the team as many points for 'good testing' as possible. Other possible guru-type roles include SCM guru, Documentation guru, Build guru, Integration guru, Visual studio guru, C# guru, Performance optimiser, Design maintainer, Deadline watcher, etc.

Note that every team member should have basic knowledge of all aspects of the project, not just the area they are guru of. For example, the testing guru cannot limit himself to testing only.

The idea is to become a 'Jack of all trades, and a master of *at least one*', i.e. to acquire a so-called 'T-shaped' skill set. Here, the horizontal bar of the T represents the breadth of knowledge in many areas i.e. 'jack of all trades', and the vertical bar represents the depth of knowledge in some areas i.e. 'master of at least one'. Such a skill set, said to be in high demand in the job market, should greatly enhance your employability.

Note that all of you are novices to begin with, but everyone can become an expert in a given area if they spend enough time on it. Choosing an area you like will definitely make the job easier.

5B. LEADING A TEAM

5.6 HAVE A LEADER

The 'Everyone will share the leadership role' approach, without explicitly assigning the leadership role to someone, rarely works in a student project. It is highly recommended that someone assumes the role 'team leader'. The team leader role does not put the person holding it *above* others in the team. Treat it simply as another role one has to play in the team, albeit an important one. Having an assistant leader is another good step in this direction.

5.7 ROTATE LEADERSHIP

Leading a team of software engineers is no easy task. It has been likened to "herding cats" (in the book The Pragmatic Programmer).

Playing the team leader role is a valuable learning experience. Ideally, all students in a team should get this experience. If more than one person is willing to try being team leader, consider the following strategy: after completing each major milestone, the current leader steps down if another team member volunteers for the team leader role. This should not be considered 'challenging' the current leader; rather, it is saying: "that looks like something worth learning. I want to try it, too". Rotating leadership is especially useful for projects lasting more than 2-3 months. However, it is OK if you want to keep the same leader for the whole project.

5.8 DELEGATE AUTHORITY

Another way to lessen the burden of the team lead and spread the leadership experience around is to have *group* leaders for each sub-group. The *team* leader can be one of these group leaders, or a separate person altogether. If the latter option is chosen, the team leader will only make high level decisions that affect the whole team (e.g. decisions related to system integration), while he will be working under the respective group leader at other times.

5.9 STRIVE FOR CONSENSUS

A team/group leader is generally expected to consider opinions of the other team members and achieve a consensus before making decisions. However, there will be times when the leader has to make quick and hard decisions on his own, either due to time pressure, or due to irreconcilable differences of opinion between team members. A team/group leader can also consult the supervisor to get one more opinion before making decisions. Once a decision has been made, others should respect this decision and try their best to implement it.

'HOW TO' SIDEBAR 5C: HOW TO FIT INTO YOUR TEAM

Here are some tips on how to deal with "hey, am I part of the team the team or what?" situations.

5.10 MY TEAM OVERRULED ME!

Problem: Your team took a decision against your opinion.

Recommendation: As the saying goes, "there is more than one way to skin a cat"; while the team's decision may be different from what you believe to be the best way to proceed, it might still lead to a reasonable outcome. Do your best to help the current course of action succeed. Do not try to sabotage it or distance yourself from the work. You can also persuade the team to get the supervisor involved in evaluating alternatives. Finally, make sure that the decision point is well documented, as such decisions make an important part of project documentation.

5.11 MY TEAM CUT ME OUT!

Problem: You want to contribute, but the team appears to cut you out (e.g. when the other members have a history together, and you are the 'outsider').

Recommendation: Voice your concern to the team (e.g. "I worry that I'm not pulling my share of the load. Can I have more work?"). Alert the supervisor if the situation does not improve. Above all, do not delay. This problem surfaces in the early stages of the project and it should be solved in the earliest stage. It is your responsibility to notify the supervisor early if the the team does not respond to you positively. If not, you might get penalised for not contributing enough.

5.12 I AM A 'SPARE WHEEL'!

Problem: Your team is doing fine without you. They do not mind your lack of contribution. Their attitude is "stay out of the way, and we'll let you share our success for free". They give you good ratings during peer-evaluations although you did not do enough to deserve it.

Recommendation: If you go along, you are taking a BIG risk. There are ways other than peer-evaluations to gauge your contribution. (e.g. your SVN/CVS activity statistics). It is your responsibility to do a fair share of the work. Most evaluators will penalise you for being a spare wheel if you do not alert them early enough. Follow the recommendations in tip 5.11.

5.13 I LABOURED IN VAIN

Problem: Someone else has done (without telling you!) the work formally assigned to you.

Recommendation: If you did your part on time and to the required quality standards, you have a right to insist that it is used in the product. However, note that this is usually a symptom of other problems (e.g. others have lost confidence in your work). Alert the supervisor so that these problems can be sorted out.

5.14 LOST IN TRANSLATION

Problem: Others are from country X except you. Most team discussions are done in their own language.

Recommendation: This is unacceptable. Ask others to use a common language at all times. Alert the supervisor *immediately* if they do not agree. Not alerting the supervisor early enough and using this problem as an excuse (i.e. "I didn't do much work because I didn't understand team discussions") at the end of the project will not get you much sympathy.

5.15 I LAG BEHIND

Problem: You are slower or not as skilled as others in the team.

Recommendation: No team member is perfect. You may be weak in programming while you still may be good in some other aspect. Do not make your weakness an excuse for a free ride. If your teammates are much stronger than you, do not expect an equal share of the credit they earn for the team, even if you work as hard as them (it is not about how hard you work; rather, it is how much you contribute). Accept tasks that you can deliver and then, deliver to the best of your ability. As you learn more, their confidence in you will grow.

'HOW TO' SIDEBAR 5D: HOW TO HANDLE PROBLEMATIC TEAM MEMBERS

A 'jelled' team is a team functioning smoothly - like a well-oiled machine. Unfortunately, student teams do not jell that often. Problems within teams are common, and working them out is part of the learning experience. This means 'team problems' is not a valid excuse for making a mess of the project.

When a team member is causing problems within the team, we should consider that this may be unintentional. Therefore, it is important that he is made aware of the team's concerns. Keeping silent will not solve the problem. Rather, it will deprive this person of a chance to mend his ways. Do this at the earliest possible moment, but discuss the issue informally and gently. Firing off an 'official' email to everyone and CC'ing the supervisor should not be the first thing you do. However, if the situation does not improve within a short period (say, within a week), it may be time to alert the supervisor. While you cannot blame everything on bad team dynamics, it pays to keep your supervisor informed about problems within your team. This could prevent the whole team getting penalised for one person's actions and could earn extra credit for 'overcoming team management challenges'.

Given next are some undesirable personalities we sometimes see in project teams. Insist that everyone in the team read this list. Read it yourself and see whether you fit any of them. If you do, your team mates probably realise it too, and it is up to you to change for the better. Note that the uncharitable name we have given to each personality is just for ease of reference. Please do not use them to belittle your teammates.

5.16 ABDICATOR

Problem: A team member who delegates *everything*. He acts as the 'management', and treats others as employees!

Recommendation: This is unacceptable. Effective delegation is a part of being a leader; but everyone, including the leader, should do a fair share of the work in all aspects of the project. Alert the supervisor. If the leader continues to evade work, change the leader at the next milestone.

5.17 TALKER

Problem: A team member who talks a lot (especially, if the supervisor is present) and acts like the most active member of the team. But that is about all he does.

Recommendation: Active participation during discussions is good as long as it does not stifle/overshadow other team members. What is not so good is dominating team meetings just to cover up lack of real contributions. Everyone should do a fair share of the work in all aspects of the project. If not, try to sort out the issue with the offender. Alert the supervisor if the situation does not improve soon. Formally keeping track of contributions can deter this type of behaviour.

5.18 LONE RANGER

Problem: A team member who does not abide by collective decisions, or does things on his own way without bothering about getting the team's consensus first.

Recommendation: Let the supervisor know if reasoning with him does not work. However, note that working in a team does not mean every decision has to be made collectively. E.g. if a certain decision's impact is local to a component, let the person in charge of that component decide whether to make the decision individually or collectively. On a related note, you are not helping the team by blowing up trivial decisions (local to your work) into a team discussion.

5.19 DOMINATOR

Problem: A team member who tries to micro-manage everything. He dismisses others' ideas and insists on doing things his way.

Recommendation: If you do not feel comfortable showing your displeasure to the dominator directly, inform the supervisor about the situation and ask his help to balance things out. It is easy for a supervisor to contact such a person privately and 'encourage' him to let others have a more active role in the project.

5.20 SUFFERING GENIUS

Problem: A team member who thinks he is more 'experienced' than the rest and thinks he has no time to waste with 'amateurs'. While he is willing to do a major chunk of the work, he does not want to get involved in anything else such as team meetings. (E.g. "I did the component X, didn't I? That's the most difficult part; you guys sort the rest out and leave me alone!")

Recommendation: This is unacceptable, particularly from a person who thinks of himself as 'experienced'. This behaviour is going to hurt the team no matter how good the code he has produced. Such code usually makes many assumptions the rest do not know anything about (because there is little interaction between the team and this person), and is inflexible in ways that hinder the future progress of the project. Let the supervisor know if the team cannot get this person to co-operate. Teamwork is an essential part of a project course. Those who fail to work as a team should be penalised, no matter how much work they do alone.

5.21 BUSY BEE

Problem: A team member who cannot commit enough energy to the project because he has a busy schedule (extra curricular activities, other courses, competitions, etc.)

Recommendation: As fellow-students and teachers, we should do our best to support this person to succeed in all the things this person is engaged in. However, a team cannot jeopardise the project due to this. If possible, adjust schedules/plans, etc. to accommodate this member's availability. You can also allocate this person less work (as much as he is willing to take, and able to deliver) and keep the supervisor informed about this imbalance of workload. The grade needs to be adjusted accordingly.

5.22 PERSONAL ISSUE GUY

Problem: A team member who is hit with an unexpected personal issue (death of a loved one, financial/health/relationship problems, etc.) that prevents him from contributing equally to the project.

Recommendation: We should do our best to support this person in this moment of personal misfortune. If possible, adjust schedules/plans, etc. to accommodate this member. You can also allocate this person less work (as much as he is willing to take, and able to deliver) and keep the supervisor informed about this imbalance of workload. The final grades will be adjusted accordingly and as per the institute's guidelines.

5.23 UNWILLING LEADER

Problem: A team leader who just sits around and pretends he is just a team member instead of doing 'leader stuff'. This makes the whole project effort unco-ordinated.

Recommendation: Some people are not natural leaders. Change the leader at the next milestone.

5.24 GREENHORN

Problem: A team member who is inexperienced and lacks almost all skills required for the project. Helping this person appears to slow the team down.

Recommendation: No matter how 'green' this person appears to be, it is well worth spending time to help him learn the skills needed for the project. Always try to work in groups during the initial stages. Even coding can be done together if you use pair programming [http://tinyurl.com/PairIntro] (highly recommended technique to improve coding skills). It is a mistake to cut this person out, make him a 'spare wheel' [see tip 5.12] , or only assign trivial tasks to this person; you need all members of the team to contribute fully for the project to succeed.

5.25 WEAK LINK

Problem: A team member who is really weak/slow, and has a track record of ending up at the bottom of the class. This is different from inexperienced team members [see tip 5.24].

Recommendation: We have to learn to make the best use of 'below average' team members. Do not make them 'spare wheels' [see tip 5.12] or assign them trivial tasks. Instead, assign non-critical tasks that match their ability to deliver. Your peer-evaluation should reflect the correct level of contribution from this person. But remember to give him a chance to contribute to the best of his ability. By the way, it is a big mistake to appoint the weak member as the tester of the system.

5.26 CHEATER

Problem: A team member who suggests (or appears to be) using unscrupulous means to do project work, such as plagiarism or outsourcing.

Recommendation: RED ALERT. The whole team will be held responsible in the end. Inform the supervisor immediately.

5.27 SLIPSHOD

Problem: A team member who continues to deliver work of low quality.

Recommendation: Insist on better quality work and refuse to incorporate low quality work into the product. You can use cross-testing (i.e. components being tested by others in the team, in addition to testing by the authors) to reduce the risk of this low quality work affecting the overall quality of the product. You can also assign earlier deadlines to this person to allow enough time for rework if needed. If the quality does not improve over time, keep the supervisor informed so that grades can be adjusted accordingly. If the course uses peer-evaluations, you can use it to reflect the low contribution from this person (i.e. low-quality work should not be counted as contributions, in fact they are negative contributions). Cutting the person out of the team is not the answer [see tip 5.11].

5.28 HERO

Problem: A team member who insists that he will take care of a critical task. He is unwilling to discuss details of how it will be done or the current status of the task.

Recommendation: This may be OK if this team member has a track record of delivering on his promises. However, in general, this is a risky situation. If the project fail because one person promised something but never delivered, the whole team will be penalised (as they should be). Make status reporting compulsory. Insist on incremental delivery. Do not assign critical components to someone before testing out his delivery quality. If in doubt, make critical components joint-work rather than entrusting them to a single person.

5.29 FREELOADER

Problem: A team member who tries to exploit others' (often, those who are good friends of the offender) goodwill to let him share the credit without doing sufficient work. Often, such a person has some 'legitimate' reason for not being able to work, and wants the team to 'help' him out. Freeloading is also called 'social loafing'.

Recommendation: If the team members go along, they will have to lie about the share of the workload during peer-evaluations. The whole team will get into trouble when the imbalance of workload is noticed by the supervisor. Please give correct information about the contribution of each team member. You are being unjust to yourself and other course-mates if you do not.

5.30 OBSERVER

Problem: A team member who does not take an active interest in the project. He keeps silent during meetings and does not contribute any ideas. However, he does all tasks assigned to him.

Recommendation: This is not a big problem as long as you do not have too many such members. If this is due to shyness or for the fear of being overruled, you could entice a more active role from this person by assigning him more responsibility. You can also assign tasks that force this person to participate more actively. For example, assign him to "come up with an alternative design and describe it in the next meeting".
Adopt the practice of recording each person's contributions including ideas they contributed. This could coax the observer to be more active in the project. However, if this person is deliberately shirking responsibilities, then it is a serious matter akin to freeloading [see tip 5.29]. Do not blindly equate the observer's lack of involvement to his support for what the team is doing. The rest of the team should continue to seek any alternative opinions the observer might have.

5.31 LAID BACK JIM

Problem: A team member who is less committed to the project than the rest. While you are working very hard for an A+, this person is 'doing just enough to get a pass'.

Recommendation: While this is frustrating, there is no easy way to motivate this person to work as hard as you are doing. Assign as much work to this person as he is willing to take. However, it should be clear that his level of contribution will be reported in peer-evaluations. In addition, whatever this person does should be of the expected quality. Doing less is not the same as doing a shoddy job [see tip 5.27]; the latter is much more damaging.

'HOW TO' SIDEBAR 5E: HOW TO DEAL WITH TEAM DYNAMICS ISSUES

Here are some tips on how to deal with problems related to how the team works together.

5.32 TUG OF WAR

Problem: The team is highly polarised over a critical decision.

Recommendation: It is healthy to have alternate views, be passionate about your views, and be prepared to defend your views. But protracted arguments can actually hurt the team. Try to find ways to objectively evaluate alternative views (e.g. implement a prototype of both alternatives and measure which one works better). You can add the supervisor's opinion into the equation as well. No matter which alternative you choose, it is important that everyone tries their best to implement the choice made. Furthermore, do not forget to document the decision.

5.33 LIFELINE LEADER

Problem: The team relies on the leader (or another dominant member) to do all decision making ("You are the leader. You tell us what to do!").

Recommendation:

For the team members: You are in this as much as the leader. You should take responsibility for the success of the project as much as the leader. The leader might not see all the alternative paths; it is up to you to point them out.

For the leader: Delegate. Assign a role to each team member (see tip 5.5).

5.34 RANDOM TEAM

Problem: All team members are strangers to each other, preventing the team from achieving the flying start the project needs.

Recommendation: Well, the only possible remedy is the get to know each other, fast! See SIDEBAR 5F for some tips on how to promote team spirit.

5.35 NEGATIVE-SYNERGY TEAM

Problem: The team spends long hours working together, sometimes late into the night, yet accomplish very little. Time is often wasted on casual conversation, teasing, flirting, joking, or arguing. No one is willing to break away from this pattern for the fear of hurting the 'team spirit'.

Recommendation:

- Working together means working in close proximity so that one team member can help another when needed, and issues can be resolved quickly. It does not mean you spend a long time 'talking' together. Programming does not mix well with intermittent distractions. It takes 10-20 minutes of undisturbed work for a programmer to attain an optimum level of productivity (seasoned practitioners call this 'going to the zone'). A single distraction can snap the developer out of this zone. Next time you are about to disturb your team mates for 10 seconds with a joke, think about how much time you are actually wasting.
- Often, it is enough to work together in sub-teams, not the whole team together.
- Identify and deal with distracters [tip 7.14] in your team (gently, of course).
- Commit to an agenda. E.g. Let us work together for one hour and integrate the version V0.2.

5.36 CLIQUE

Problem: Several team members have formed a closed group (i.e. a 'clique'), alienating everyone else. For example, when one of the clique members gets into an argument with a non-clique team member, the whole clique comes to the defense of the clique member, although the discussed issue does not concern them.

Recommendation: If possible, let the clique become a sub-group, and let them work together on a sub-system. This should reduce friction between the clique and the rest. If the clique continues to undermine project harmony, get the supervisor involved to balance things out. For example, if you think the clique is forcing non-optimal decisions on the team, get your supervisor's opinion on the decision as well.

5.37 HERD OF CATS

Problem: Each team member does what he wants. No one follows the directions from the team leader, or there is no clear leadership.

Recommendation: Such behaviour results in problems such as duplication of work, integration problems, and in general, a messy product. In addition, most instructors will explicitly penalise such unprofessional behaviour. Appoint a strong leader and co-operate with him or you will all lose out in the end.

5F. INCREASING TEAM SYNERGY

Working *in* a team is not the same as working *as* a team. A synergistic team's output should be greater than the sum of what each member could have done on his own. Here are some tips to increase your team's synergy.

5.38 BUILD TEAM SPIRIT EARLY

As soon as the team is formed, make it a point to spend some time together. This lets you get to know your team members quickly. Even small adjustments like having lunch together or going for a movie together can help in this area.

Doing early project activities together strengthens your team spirit further and helps you gauge strengths and weaknesses in each other. Consider the task of picking a project topic; instead of emailing back and forth about potential topics, meet together and talk it over face-to-face. At early stages, pick small and easy activities for the whole team to do together quickly. This will boost your confidence in your team.

5.39 BUILD A TEAM IDENTITY

Building an identity for your team can help bring the team together. Here are some things to try:

- Take a 'funky' team photo (Examples: http://tinyurl.com/teamphoto1, http://tinyurl.com/teamphoto2).
- Set up a web page for your team (Examples: http://tinyurl.com/codedroids).
- Brainstorm for a catchy code name for your team/product.

5.40 BUILD POSITIVE GROUP NORMS

Groups norms are mutual understandings as to "the way we do things in our team". Try to build positive group norms. E.g. "we always come to the meeting on time", "we never submit code without unit testing first", "No texting during meetings", etc.

5.41 KEEP IN TOUCH

When a team member is 'unreachable' for an extended period of time, the usual excuse given is "oh, I don't check *that* email account" or "I was using my *other* phone that day". As a team member, it is your responsibility to keep the team informed about how to reach you.

However, it does not mean that you have to be reachable all the time. It is OK to be unavailable for project work for acceptable reasons as long as you keep your team informed of your unavailability. For example, "I don't check mail during 11pm-8am" seems quite reasonable. You can also 'take leave' from the project (e.g. to go on a trip) as long as you inform the team early and your absence is officially incorporated into the project schedule. In addition, try your best to minimise the impact of your absence (e.g. finish your assigned tasks *before* leaving, rather than postpone them).

5.42 WORK TOGETHER, WORK ALONE

Some things are better done as a team while some others are better done individually. Often, you can start something by working together and then split the work and carry on individually. But maintain a healthy interest in

what others are doing. It is also good to review work done by others at milestones, especially work done individually.

Be careful of how you tread on another member's work space. For example, be tactful when you refactor code written by others. It is not recommended to refactor others' code, without their consensus, to suit your coding style.

CHAPTER 6: WORKING WITH YOUR SUPERVISOR

Most project courses assign each team a supervisor, i.e. an instructor who oversees the team's progress, provides expert advice on the problem domain, and possibly takes part in grading the team. Project teams can meet their supervisors regularly for consultations. The supervisor plays a critical role in your project; not only is he meant to be your best source of advice and guidance, but he may also be able to influence your grade. Therefore, it is important that you know how to work with your supervisor in an optimal manner.

6.1 HAVE REALISTIC EXPECTATIONS

Most importantly, your supervisor is not an all-knowing oracle. While even the most experienced educators have differing views about various aspects of software engineering, the supervisor you get could be an inexperienced junior faculty member or a senior student. Therefore, do not expect your supervisor to have a sure-fire solution to all your problems. Even when you get an answer from him, do not take it as the best possible answer one can get.

6.2 ASK FOR OPINIONS, NOT DECISIONS

When faced with alternative ways of doing something, do not ask the supervisor to make the choice. Instead, ask what are the pros and cons of each alternative. You can make your final decision based on your own opinions, your supervisor's opinions, experimentation results (i.e. you can try out each alternative on a small scale to see which one is better), and other information you can gather on the subject.

6.3 PREFER WRITING TO CALLING

When you telephone the supervisor, you are effectively saying "I don't care what you are doing now. Drop it and attend to me right now!". When you send an email or a text message to him instead, you are saying, "Attend to me when you have time". Use the former option sparingly.

6.4 DO NOT DROP BY UNANNOUNCED

If you want to meet the supervisor, make a prior appointment or follow pre-determined consultation hours. Most supervisors are too nice to turn you away if you come unannounced, but they will not be happy about the disruption.

When you make an appointment with your supervisor, PLEASE be there on time. Plan to meet together 10 minutes before the appointment time and go to the supervisor's office as a team, rather than each member joining the meeting one at a time.

6.5 RESPECT SUPERVISOR'S TIME

You do not get credit for spending time in the supervisor's office or writing the most number of emails to the supervisor. While a supervisor might offer to meet you 'as many times as you want' and ask you to write 'as many emails as you want', it is your responsibility to limit both to reasonable levels. It shows forethought, care, and professionalism on your part. It makes the supervisor's life easier as having to switch many times between tasks could heavily disrupt his own work schedule.

Do your homework before you make contact with the supervisor. Discuss the questions with your team first. Try to find answers on your own before you approach the supervisor. Prepare your questions before you meet the supervisor. Better yet, email him the questions in advance. This helps to clear your own mind about the issue you want to discuss. Consolidate your questions into a single email rather than firing off an email the moment you encounter a problem. Keep meetings short: get in, do your business, get out. Do not start team discussions in the middle of a meeting with the supervisor.

Avoid involving the supervisor in the things you should do yourselves, such as debugging. At the same time, make sure you maintain an adequate level of interaction with the supervisor, even if he does not initiate contact himself. You do not want to be referred to as "oh, the team that never came to see me" when he sits down for grading.

6.6 ALLOW TIME TO RESPOND

Most supervisors cannot speed-read. Do not send him a document five minutes before a meeting and then expect feedback during the meeting. Be patient if you do not get immediate answers to your questions. Sometimes, supervisors have to consult other supervisors, a senior professor, or other resources before answering.

6.7 LET THE SUPERVISOR SET THE TONE

Some supervisors keep the supervisor-team relationship informal and relaxed while others want to maintain a certain level of formality. It is the supervisor's choice (or may be dictated by the course structure). You should not try to force a different level of formality into the relationship. For example, do not try to 'lighten the mood' by inserting jokes in to a discussion with the supervisor if the supervisor is clearly trying to emulate a formal project discussion.

6.8 KEEP IN THE LOOP

It is important that the supervisor is in touch with all the important developments in your project. For example you can give your supervisor access to your project discussion forum, code versioning system, mailing group, etc. Copying every internal project-related email to the supervisor is a definite 'no no'; sending a weekly summary or a progress report (even if he does not ask for such) is a better alternative.

If your team is facing difficulties, let the supervisor know; pretending otherwise will prevent the supervisor from helping you on time.

On a related note, think twice before you bypass your own supervisor and approaching someone above him, e.g. approach the course instructor instead of talking to the teaching assistant supervising your team. Doing so can sour the supervisor-team relationship.

6.9 DO NOT BLAME THE SUPERVISOR

If the project did not go well, resist the temptation to blame the supervisor. While the supervisor gave all advice in good faith, it was not his responsibility to make the project a success. Besides, blaming the supervisor for a failed project can hurt your grade further, in more ways than I care to mention here.

CHAPTER 7: PROJECT MEETINGS

The best meetings get real work done. When your people learn that your meetings actually accomplish something ,they will stop making excuses to be elsewhere.
--Larry Constantine [from the book Beyond Chaos by Larry L. Constantine]

A project course requires frequent project meetings. Most of these meetings can be very informal. However, periodic semi-formal project meetings are useful, too, especially at critical junctures of the project when input from all members is required. Some courses require the supervisor to attend such meetings as an observer, evaluator, and sometimes, as a facilitator. If that is the case in your course, how you conduct the meeting will form an impression in your supervisor's mind as to how professionally you go about doing the project, and this impression can affect your grade. Besides, everyone hates meetings that waste time. Introducing a little bit of structure and order into meetings, as suggested in tips in this section, can make your project meetings more productive.

7A. RUNNING A PROJECT MEETING

7.1 HAVE AN AGENDA

Having an agenda reduces aimless 'time waster' type meetings, and sets targets to achieve during the meeting. Decide the agenda for the meeting in advance, and communicate it to the team. If this is not possible, at least decide the agenda at the beginning of the meeting.

7.2 APPOINT A MODERATOR

It is a good idea to assign someone the responsibility of making meetings work. This person, let us call him the *moderator*, should steer the meeting along the correct path. The process need not be formal, for example, there is no need for the moderator to sit at the head of the table or stand up when talking. The team leader will often serve as the moderator, but not necessarily.

7.3 START ON TIME

Once the starting time is decided, it is everyone's responsibility to be there on time. If you are not sure of making it on time, let others know before finalising the start time, so that it can be rescheduled. Two prominent 'time wasters' during meetings are 1. waiting for someone to arrive, and 2. filling in someone who joined mid-meeting. Some teams even fine late-comers to discourage tardiness.

7.4 MAKE IT SHORT, BUT NOT TOO SHORT

We should minimise time wasted in meetings because every minute wasted in a meeting is a loss for all attendees. For example, 10 minutes wasted in a six-person meeting is effectively a 60-minute loss. But on the other hand, the time spent on getting to and getting back from a meeting is much higher in a student project than in an industry project. This is because, unlike in most industry projects, students do not work in cubicles located close to each other. Therefore, it does not make sense to have very short meetings either (imagine spending half an hour getting to a five-minute meeting). If you do not have enough items on the agenda to cover at least one hour, try using online meetings instead (for example, using instant messengers). Alternatively, you can schedule meetings and 'work together' sessions back-to-back so that all the time spent on getting to the meeting location is justified. In our experience, two-hour meetings work better than one-hour meetings.

7.5 TAKE NOTES

Appoint a 'scribe' to take notes during the meeting (i.e. to record minutes). These notes should be circulated among the team members, and perhaps CC'ed to the supervisor. This is particularly useful for the supervisor to keep track of decisions taken by your team (remember, he tracks multiple teams), and for members who did not attend the meeting. A digital camera can help greatly in capturing anything on the whiteboard.

7.6 SUMMARISE

Just before you break up the meeting, the scribe can recap (verbally) what was discussed during the meeting. This reiterates the 'takeaway points' of the meeting. This is also useful to gauge how much you accomplished during the meeting.

7.7 FIX FINISH TIME, FINISH ON TIME

Fix the finish time when you schedule the meeting. For example, do not schedule 'open-ended' meetings such as '5 pm onwards'. Instead, it should be something like '5-7 pm'. Do not let the meeting drag on beyond the finish time. When you aim to finish on time, there is an incentive to get on with the agenda.

7.8 FIRST THINGS FIRST

Higher-priority items should appear earlier in the agenda. If you run out of time, you can postpone unfinished lower-priority items to a later meeting.

7.9 SET ASIDE REGULAR MEETING SLOTS

Schedule your meetings on a regular fixed time slot (e.g. we meet every Friday 12-2 pm). Do this at the beginning of the course and inform team members to keep this time slot free. This reduces the chances of someone skipping a meeting due to an alleged timetable clash.

7.10 USE BRAINSTORMING

Brainstorming is a good tool to put the collective brain power of the team to good use. In a brainstorming session, everyone throws in any idea that comes into their mind no matter how outlandish it may sound. Proposed ideas are not debated or evaluated during the session. All ideas presented are recorded for further consideration.

7.11 SET GROUND RULES

Setting ground rules avoids certain unwelcome behaviour from team members without having to confront the culprit individually. Some example ground rules.

- No texting, web surfing, or other side activities during the meeting - you should give 100% attention to the meeting.
- No mini-meetings within the meeting (i.e. no separate side-discussions during the meeting).
- No use of abusive language.
- Only the moderator can interrupt when someone is speaking.
- No bringing up unrelated topics. No chit-chat.
- Use only English (or whatever the language common to all participants) during meetings.

7.12 AVOID WIN-LOSE SITUATIONS

Some suggestions to avoid frictions during meetings (and at other times):

- Avoid voting to decide things. If you must, use a secret ballot. An open vote creates 'winners' and 'losers'.
- Do not attach opinions to yourself. Rather than say "I think we should do X", say "Another way we can do this is X".
- When giving your opinion about someone else's suggestions, start by saying what you like about it before going on to what you dislike about it.
- Make sure you heard from everyone before you make the final decision. Force others to think of alternatives. Rather than say "That's what we should do. Anyone has a better idea?" (implies "anyone dares to challenge this idea?"), say "Looks like method X is one good way to do it, but before we make a decision, let's just see what the other options are..."

7.13 USE MEETINGS FOR MEETINGS

If something does not apply to the majority of those present, do not use the meeting to discuss it. Avoid putting on the table for discussion topics that are not suitable to discuss in a meeting, maybe because they require some preliminary research to be done first or because they require hard thinking and analysis at a slower pace. When the topic to be discussed is known beforehand (which should be the case most of the time), do your homework before the meeting and come prepared to discuss it. For example, create the test plan *before* the meeting and use the meeting to tweak it rather than trying to create the test plan from scratch in the middle of the meeting.

'HOW TO' SIDEBAR 7B: HOW TO HANDLE 'MEETING POOPERS'

Here are some tips to handle those who disrupt meetings in various ways.

7.14 THE DISTRACTOR

Problem: A team member who frequently sends the project discussion along an irrelevant tangent. Some distracters joke too much, some take pleasure in arguing about unimportant things, some bring up 'interesting' trivia (e.g. "By the way, you know what else is an interesting design pattern?"), some love to interrupt with chit chat ("Hey, is that a new phone. What happened to the the old one?"), and the list goes on.

Recommendation: This is often (but not always) unintentional. Set a ground rule to pre-empt this kind of behaviour (e.g. 'not too much joking during meetings').

7.15 NATIVE SPEAKER

Problem: A team member who uses his native language when talking to his compatriot teammates.

Recommendation: This is unacceptable. Set ground rules to pre-empt this behavior.

7.16 CHATTERBOX

Problem: A team member who loves to hear his own voice dominate the discussion.

Recommendation: The moderator should take action to give equal chance to other team members.

7.17 TOPIC JUMPER

Problem: A team member who constantly brings up topics related to his own part of the project even when the team is discussing something else.

Recommendation: It is the job of the moderator to redirect the discussion to the topic at hand. Another tactic is to allocate time in the agenda for each person to bring up issues.

7.18 MEETING ADDICT

Problem: A team member who uses the meeting as a surrogate social life. He insists on frequent meetings and uses various tactics to lengthen meetings.

Recommendation: A meeting is a costly tool (in terms of time spent). Question the need for the meeting that do not have a worthy agenda. During a meeting, the moderator should follow the agenda and limit the meeting to the duration previously agreed.

7.19 NO SHOW

Problem: A team member who often skips team meetings.

Recommendation: Count the time spent in meetings towards the contribution calculation for each team member.

7.20 MULTI-TASKER

Problem: A team member who is physically present in the meeting, but uses the meeting time to do other work, or frequently drifts away from the discussion to answer text messages or phone calls.

Recommendation: Set ground rules to preempt such behaviour (e.g. no texting in meetings)

CHAPTER 8: PLANNING AND TRACKING THE PROJECT

I have always found that plans are useless, but planning is indispensable.
--Dwight Eisenhower

Project planning and tracking is a vital part of any non-trivial project. Most students use ad hoc 'back of the napkin' 'inside your head' planning. They tend to avoid elaborate project planning or 'fake it' for the sake of appearances because of two reasons:

- **It is a frustrating activity**. Since things rarely go according to the plan, it seems like a waste of energy to draw up a plan.
- **They think they know all that is there to know about planning**. Since the meaning of 'planning' is well-known, students think there is nothing much more to learn about it.

In reality, a typical student project can get by without much elaborate planning because of the flexibility in available time and expected deliverables. For example, they can respond to a project crisis by skipping other course work to spend more time on the project, heroics such as overnight coding sessions, or simply not delivering what was initially promised.

However, project planning is integral to the learning experience of a project course.That is why sincere project planning is usually given extra credit. Dilligent students who want to use the project to learn the valuable skill of project planning will find the tips in this section very useful.

Why won't developers make schedules? Two reasons. One: it's a pain in the butt. Two: nobody believes the schedule is realistic. Why go to all the trouble of working on a schedule if it's not going to be right? -- JoelOnSoftware.com

8.1 HAVE A PLAN

The most important tip we can give about project planning is 'do it'. We may not get it right the first time; but our project planning skills will improve over time, and our plans will get closer to reality. There is much to be learned about planning (and related activities like estimation and scheduling), and much to be gained from practising it during your project. Here are some:

- Treating project planning as a distinct part of the project (rather than an implicit thing we do in our heads) forces us to think through what we are going to do in the project and how we are going to do it. This gives you a far better result than if you just 'winged it'.
- A plan helps us to foresee and mitigate risks. For example, planning might alert us to tasks that are in the *critical path*. i.e. any delay in their completion can directly affect the delivery date.
- A plan supports better decision making. For example, having a plan helps when you have to decide whether to keep working on a given feature or abandon it for the sake of a more important feature:

"According to the project plan, we should start feature 'foo' by today if we want to include in the final product. This means we cannot deliver 'foo' if we keep working on 'bar''.

- A plan helps us cross-check our understanding of the project. For example, instructors can use it to assess whether you have underestimated the work: "How come this team allocated five hours to feature 'foo' while all other teams allocated more than 20 hours?".
- It helps us to assess progress: "Hey, feature 'foo' is supposed to be done by 20th. At this rate, I don't think we are going to make it".

8.2 KNOW OBJECTIVES

What is the objective of a project plan? A project plan should help us answer at least some of the following questions:

- What (and when) are the project milestones?
- What are the deliverables at each project milestone (external and internal)?
- Who does what, in which order, by when?
- Which part of the product each team member will implement?
- At a given point of time, what is the assigned task for each team member?
- What are the roles played by each project member?

8.3 GET EVERYONE ON BOARD

Preferably, planning should be done as a collaborative activity. When the plan is built upon team consensus, your plan itself tends to be of better quality and others tend to follow the plan more willingly. Even if your team wants you - the team leader - to draw up the plan, do not simply allocate work to others at your discretion. Instead, present them with a draft plan and get their feedback before adopting it.

8.4 CHOOSE ITERATIVE

Any non-trivial student project should be done in iterative fashion. The waterfall process [http://tinyurl.com/wikipedia-waterfall] may suit certain kinds of projects, for example, when a team of seasoned programmers builds a product they have prior experience in. Most certainly your project does not fall into this category.

If you are serious about having a well-defined process for your project, you can choose from many available processes such as eXtreme programming, Unified Process, SCRUM, etc. However, smaller projects can survive just fine by simply using an iterative approach to developing the product. Elaborate process-specific practices advocated by those processes may turn out to be an overkill for small, short-term projects.

8.5 ORDER FEATURES

If we are following an iterative process (we should) and the product has many features, we have to figure out in which order we will implement them. While many factors influence this decision, two most important ones are the value and the risk. One strategy is to use the following order [adapted from the book *Agile Estimation and Planning by Mike Cohn*]: first, we implement *high risk, high value* features; then, *low risk, high value*; last, *low risk, low value*, leaving out *high risk, low value* features.

This order emphasises that 'critical' features - those that are difficult to implement (i.e. 'high risk') yet vital to the proper functioning of the product (i.e. 'high value') should be scheduled early in the project although the client/instructor does not demand it that way. This gives us enough time to recover if something goes wrong while implementing those features.

Another important factor we must consider is feature dependencies. This analysis might even discover a *critical path* of features in the project. A critical path is a tightly packed chain of features each one dependent on the previous one. A delay in one feature will cause a delay in the final delivery date.

8.6 KNOW WHEN TO DETAIL

"It'll be done when it's done!" is not a plan [quoted from JoelOnSoftware.com], neither is it a good attitude to have. "Design 4 weeks, implement next 6 weeks, test during the last 6 weeks" is a plan, but it is too high-level (and not a very good one either - for one thing, it follows the waterfall model). So what is the appropriate level of details for your project plan?

First, there is a limit to how detailed it can be. The element of uncertainty common to all software projects and our own lack of expertise in similar projects prevent us from knowing in advance every detail of how we will do the project.

Second, we may not want to detail out certain things upfront even if we could. For example, we may not want to be specific about which function we will be implementing on Monday of the 11th week from today. Instead, we might stop at specifying which set of features we will be implementing during weeks 8-12.

In general, we prefer to define major milestones in the high-level plan (we can call this the master plan) using features rather than components, functions, or tasks. This emphasises our focus on delivering values to the customer and not merely 'doing some project work'. Here is an outline of such a plan :

> iteration 1 [weeks 1-2] V0.0 : skeleton architecture, feature 1.
> iteration 2 [weeks 2-3] V0.1 : feature 2, feature 3.
> iteration 3 [weeks 3-4] V0.9 : feature 4.
> iterations 4-7 [2 months] V1.0: features 5,6,7 and 2 additional features from the 'nice to have' list.

This plan tells us which features we will deliver from each iteration. At the end of each iteration, we plan our next iteration in more detail (we can call this the *iteration plan*). Here, we will break down each feature in that iteration into fine-grain tasks, estimate the effort required, and schedule them appropriately.

8.7 ALIGN TO EXTERNAL MILESTONES

Align your iterations appropriately with external milestones imposed by the course instructor or the external client. For example, if your iteration ends one day before the external milestone, you have a buffer of one day to pit against any last minute delays.

8.8 CHOOSE A SIMPLE FORMAT

If we can use a simple table or a spreadsheet to document the project plan and still achieve all the objectives easily, there is no need for fancy templates, diagrams, and charts. See here [http://tinyurl.com/simplePP-doc] for a sample project plan, maintained as a simple MSWord document.

8.9 TIME-BOX ITERATIONS

A floating deadline (i.e. "Let's finish our tasks and email each when we are done") is a sure way to slow down the project. Estimate how long the iteration will take, and fix a deadline based on the estimates. Deadlines are a way to get a commitment from all, and push the project tasks up their priority list ("I have to finish this by Friday, or the team will let me have it!"). Of course if you find yourself finishing the work before the deadline, set a shorter deadline next time.

8.10 START WITH SHORT ITERATIONS

Start with one-week iterations. If things go according to the plan, you can try longer iterations; if they do not, make the next iteration even shorter. If you can gather all team members together for a couple of hours, consider starting with a mini-iteration that lasts only a couple of hours.

8.11 USE SAME SIZE ITERATIONS

Once you have found an iteration size that fits you, try to stick to it. Try to schedule milestones on the same day of the week. Doing so helps your team to 'get into a rhythm' and makes subsequent iterations easier to plan. This especially makes sense since most of your work from other courses also follows a similar pattern.

Parkinson's law: Work expands to fill the time available.

8.12 ESTIMATE SIZE, DERIVE DURATION

Answering question such as "How long do you need to do task X?" in fact requires answering two questions: "How much effort will it take?" and "How long will it take you do that amount of work?". The first question is primarily about estimating the size of a task. The answer to the second question depends on the person who does it and when it will be done.

... my mantra of estimation and planning: estimate size, derive duration [adapted from the book Agile Estimation and Planning by Mike Cohn]

8.13 DEFINE VERIFIABLE TASKS

When you need to define tasks, make sure they are verifiable. 'Complete the parser' is a verifiable task. Either the parser is complete, or it is not. 'Add more functionality to the parser' or 'Finish 50% of the parser' are not verifiable. Always plan an iteration using verifiable tasks. How else will you decide when the task is done?

8.14 RESIST 'GUESSTIMATING' EFFORT

While you may not have a lot of past data/experience to help in estimating, and while you are not usually required to use sophisticated estimation techniques, do resist the temptation to give the first number that pops into your head. Take some time to consider the following factors:

- Try to break the feature/task into smaller (but not ridiculously small) parts.
- If it is a coding task, estimate how many LOC it will require. Use your LOC/hour rate to calculate an initial estimate.
- Does it require learning new technologies/tools? If it does, you need to allocate time for it.
- Does it require you to work with other team members? The more team members you have to work with, the slower the progress will be.
- Have you done any similar work before? If you have, how much effort did it take?

In rare cases when you have absolutely no idea of the size of a task, you may want to try out a similar task at a MUCH smaller scale before venturing an estimate. Such a tryout is called a *spike solution*. While this delays the planning process a bit, it is much better than planning based on guesstimates.

8.15 ESTIMATE COLLABORATIVELY

We can estimate the task size as a team or as a sub-group of teammates related to the task. You can ask the person who is likely to do the task for an estimate. Then, others can add their opinion, especially if they think the task is being over/underestimated by a big margin.

If you feel that some team members are padding estimates to evade work, here is a bit more elaborate and fun way to estimate task size (this approach is called *planning poker* [http://www.planningpoker.com/]):

- Everyone who takes part in estimation meets together.
- One person outlines the task that needs to be estimated.
- Each person mentally calculates an estimate (but does not tell others).
- After everyone is done estimating, all reveal their estimate at the same moment. The easiest way to do this is to use your fingers: Fold the right number of fingers in your hand to indicate your estimate, but keep them out of view from others (under the table or behind your back) until it is time to reveal your estimate.
- If the estimates are different, each one justifies his estimate. This should lead to a useful discussion about the best way to do the task.
- With the new knowledge gathered from the discussion, the whole group takes another shot at estimating the task. This goes on until some level of consensus is achieved about the estimate.
- We repeat the process for all tasks that need estimation.

8.16 HAVE 'FLOATING' TASKS

Floating tasks are the tasks not in the critical path. Their completion time is generally flexible. E.g. 'looking for a good format for the user manual' can be done any time, as long as it is done before starting the user manual. Identify floating task for idle team members (those who have completed their tasks, and waiting for others to finish) to do.

8.17 THINK IN HOURS, NOT DAYS

Unlike industry practitioners, your commitment to the project is not full-time (because you are taking other courses at the same time). This means the number of hours you spend on the project is different from one day to another. Therefore, estimate task duration using person hours, not person days.
E.g. If you think a task has a size of five hours, and the time you can devote for the project is Monday - two hours, Tuesday - zero hours, Wednesday four hours, you should ask for time until the end of Wednesday to finish the task.

8.18 DO NOT FAKE PRECISION

Let's face it: estimates can never be 100% accurate. The larger the task is, the less precise the estimate will be. First, break larger tasks into smaller ones. Second, do not give seemingly precise values to largely uncertain estimates. Using scales such as 1,2,3,5,8 (Fibonacci series) or 1,2,4,8 reflects that when the task is bigger, our estimates are less precise [adapted from the book *Agile Estimation and Planning by Mike Cohn*]. For example, using the first scale means when a task is bigger than five hours, we put it into the eight hours bucket instead of choosing between six, seven, and eight hours.

8.19 IT TAKES LONGER THAN YOU MIGHT THINK

According to Ed Yourdon's book *Death March*, one reason for project failure is the naïve 'we can do it over the weekend' optimism. An overconfident student might think he can develop *any* system over a weekend, and things such as documentation, error-handling, and testing are so 'minor' that they do not need much time.

We should keep accurate measures of the time we spend on each task, compare it with our initial estimates, and adjust our future estimates accordingly. Do not be surprised if you found your estimates were less than half of the actual time you took.

The process is called estimation, not exactimation.
-- Phillip Armour (as quoted in the book Software Estimation by Steve McConnell)

8.20 INCLUDE THINGS TAKEN FOR GRANTED

Things like unit testing, adding comments, and routine refactorings are usually considered integral parts of coding and, therefore, should not generally be shown as separate activities in the project plan. However, it is better for student teams to include them as separate activities [adapted from the book *Agile Estimation and Planning by Mike Cohn*], at least until they become a habit. This practice will help us to consciously allocate time for these important activities that may be overlooked otherwise. Also, remember to allocate time for fixing bugs (found in features delivered in previous iterations).

Do not forget to include meeting times in the project plan, too. However, we should not include things that do not add value to the project (e.g. answering emails).

No battle plan survives contact with the enemy. -- Helmuth von Moltke

8.21 PLAN SINCERELY

Some teams allocate one member to 'do the project plan'. This member creates a nice project plan and submits it, while others never get to see it, let alone follow it. Instead, each team member should have an intimate knowledge of the project plan, at least the part that applies to him. One should always be able to answer the following question: "According to the project plan, what should you be doing today?"

It is not enough to know the plan. You should also try to *follow* the plan. Always refer to the plan when choosing what to do next.

Even the best planning is not so omniscient as to get it right the first time. -- Fred Brooks

8.22 TRACK VISIBLY

Each iteration should start by drawing up a detailed plan for that particular iteration. For increased visibility, you can send a copy of the plan to the supervisor at this stage itself.

You can also maintain the project plan as a GoogleDoc (or some other online format), and give access to the supervisor. In this case, it is easy for the team members to update the document as and when required. For example, a team member can mark a task as 'done', as soon as it is done. GoogleDocs also keeps track of the past versions of the document, and shows who made which change. You can also consider more specialised project tracking tools mentioned in Chapter 9.

8.23 REFINE REGULARLY

Planning is not over until the project is over. 'Revise plan' should appear frequently in the project plan. It is also advisable to put someone in charge of this aspect.

Do a post-mortem analysis at the end of an iteration. What did not go according to the plan, and why? Change the master plan as necessary.

When there is a mismatch between what you actually did and what the plan says, it means the plan needs adjusting. But remember, you should not adjust the past plan to match what you did up to now, but you should adjust the future part of the plan to match what you now think you can do (as opposed to what you originally thought).

Evaluators know that a project plan never revised at least once is most likely a fake.

8.24 ADD BUFFERS

Do not assume that everything will go according to the plan. Insert buffers to strategic locations of the task chain. If task *B* depends on the completion of task *A*, try not to schedule task *A* immediately before task *B*. Scheduling task A a little earlier creates a buffer between *A* and *B*. That way, a delay in *A* is less likely to affect *B*.

Note that buffers should be explicitly shown in the plan. Inflating all estimates by 10% is not the same as adding buffers.

8.25 DROP FEATURES WHEN BEHIND

Consider the following true story: A team claims to have implemented all the features of the product, but did not have enough time to hook up the functionality with the GUI, and therefore, had no way of demonstrating those features. They offer to take the evaluator through the code to prove their claims, an offer he politely declines. The team receives a very low grade, although they supposedly fell only a small step short of the complete product.

They could have got a much higher grade if they had omitted some features and used that time to hook up the remaining features to the GUI.
The moral of the story: When short of time, you have to sacrifice some features so that at least the other features survive. If you realise you are behind schedule, and if late submissions are heavily penalised (they should be!), you should not carry on with the current plan as if nothing is wrong. Choose wisely what you will include in the product, and what you will have to omit. Do not blindly re-allocate time from one task to another (e.g. from documentation to coding) - the decision should depend on the amount of credit you stand to gain from each. Do not skimp on testing no matter what - it is better to have 90% of the features 100% tested (i.e. a good product that lacks some features) than 100% of the features 90% tested (i.e. a lousy product. Period).

Similarly, when you cannot make the iteration deadline (an internal deadline set by the team), drop/postpone features rather than extend the deadline. Changing deadlines disrupt the rhythm of your team.

8.26 DO NOT SLACK WHEN AHEAD

If you are ahead of the schedule, do not 'take it easy' until the plan catches up with you, and do not use the extra time for 'gold plating' (i.e. adding unnecessary adornments to the product). It simply means the plan was wrong, fortunately, in a good way. Do what you would do if you were behind schedule; consider the current status, revise the schedule accordingly, and carry on.

CHAPTER 9: USING PROJECT TOOLS

Tools amplify your talent -- from the book The Pragmatic Programmer

Any decent software developer will use a repertoire of useful tools. The better your tools, and the better you know how to use them, the more productive you *can* be. Of course, most student projects can be done without using too many special tools, but such teams fall behind teams that do use appropriate tools.

'HOW TO' SIDEBAR 9A: HOW TO USE TOOLS FOR DIFFERENT PROJECT TASKS

9.1 CODING

Using an Integrated Development Environment (IDE) such as Visual Studio, NetBeans, and Eclipse is a must for today's programmers. Nobody codes in notepad any more! There was a time not long ago when notepad-like text editors could show more productivity than cumbersome IDEs, but those days are gone. Especially for a novice like yourself, using an IDE is highly recommended. Today's IDEs are loaded with features (and some of them are actually useful!). Keep looking for smart ways to use your IDE - a good way to start is to learn the keyboard shortcuts for frequently used features.

A very much under-used IDE feature is the debugger. Learn how to stop execution at a pre-defined 'break point', how to 'step through' the code one statement at a time, inspect intermediate values of a variable at various points, etc. Using the debugger to do all this is a much superior way to inserting ad hoc *print* statements.

Modern IDEs already have some automatic refactoring and code analysis capabilities. But you may also wish to look at more powerful refactoring add-ons such as Re-sharper (a commercial plugin for Visual Studio. Free education licenses are available).

Some tools can check style conformance, auto-format code, and do other types of quality assurance of the source code. E.g. FXCop can be used to enforce coding standards in .NET assemblies.

9.2 PROFILING

Profilers are indispensable for effective performance optimizations. They help you pin point *actual* performance bottlenecks in your program. IDEs such as the latest Visual Studio have in-built profilers. Other examples: CLR Profiler, DotTrace

9.3 METRICS

Some IDEs such as the latest Visual Studio and other speicalised tools such as Microsoft LOC counter help you gather various metrics about the code, such as the LOC written by each team member.

9.4 CONFIGURATION MANAGEMENT

Software Configuration Management (SCM) is another important aspect of a non-trivial project. At a minimum, you should use a source code revision control tool such as Subversion (a Windows client for Subversion is TortoiseSVN) to keep track of code versions. If you are open-sourcing your project, you can use the SVN server provided for free by Google Code Project Hosting.

Institute check-in policies to enforce proper commenting, tagging, and to keep the build unbroken.

9.5 DOCUMENTATION

Another under-used tool is the word processor. If you find yourself doing some of the below, it is an indication that you are not using the word processor optimally.

- When you insert a figure/heading in the middle of the document, you have to manually increment all figure/heading numbers that appear below it.
- When you want to change the appearance of all chapter headings, you have to do this manually, one heading at a time.
- You insert blank lines to push a stranded heading (i.e. stuck at the bottom of a page) to the next page.
- You modify some part of the document, and write an email to describe what you did to other team mates. You attach both the modified and the original document, in case the team decides to reverse your modification.

Modern word processors can use 'styles' to apply consistent formatting to a document; auto-generate table of contents, list of figures, and indexes; auto-update cross references to figures/tables when you insert a figure/table in the middle and automatically keep track of changes you did to a document.

Tools such as MSVisio (commercial) and ArgoUML (free) can help you with drawing diagrams. Some IDEs let you generate UML diagrams from the source code.

9.6 TESTING

You must automate as much of testing as possible. Unit testing frameworks such as JUnit and NUnit can help in unit testing as well as other types of testing (integration testing, regression testing, system testing).

For most products, it is well worth writing your own test driver for specialised system testing.

9.7 COLLABORATION

Several types of tools can help you to collaborate easily with your team members. Some simple examples include GoogleDocs, Wikis, and mailing groups.

9.8 PROJECT MANAGEMENT

Tools such as MSProject help you in project planning and tracking. Trac is an example of a server-based project management tool.

Free project hosting services such as Google Code Project Hosting (GCPH) or Sourceforge can be used to host and manage your project online. These services usually come with tools for issue tracking, code versioning, discussion forums, etc. but impose some restrictions, for example, only open-source projects are allowed on GCPH and Sourecforge. Alternatively, you can use free-for-non-commercial-use accounts provided by commercial project hosting services such as Pivotal Tracker and Zoho.

9.9 BUILDING

Most modern IDEs have in-built support for automated project building (look for a 'build' button on your favourite IDE). Specialised build tools include Make and Ant.

9.10 ISSUE TRACKING

Bugs (and other issues such as feature requests) go through a life cycle. For example, a bug can be in different life cycle stages such as 'raised', 'assigned', 'fixed', 'reopened', etc. Tools such as Bugzilla and Trac help is in tracking bugs (and other similar issues).

It is good to enforce the use of an issue tracker to record bugs you discover in others' code because it pressurises everyone to produce better quality code (because no one wants to see many bugs recorded against their name). Some issue trackers can be used to record project tasks (not just issues), too, resulting in an automatic record of what each person contributed to the project.

9.11 LOGGING

There are tools/libraries to help your code produce an audit trail (a log file) while in operation. Such tools can be easily configured to control the level of details produced. For example, when you suspect a bug in the code that affects the operation in subtle ways, you could increase the level of details in the log file to try and catch the bug.

9B. USING TOOLS

9.12 DO NOT EXPECT SILVER BULLETS

Using a plethora of tools does not guarantee an increase in productivity or a successful project. If you want to maximise the potential of tools, you should use the right tools and integrate them into your work in such a way that it flows with your team's dynamics. On the other hand, tools cannot help you much in decisions such as how

to partition your code into packages, in which directory to place documentation, who does the integration, how to resolve bugs during system testing, etc.

9.13 LEARN TOOLS EARLY

The best time for you to learn the tools is during the initial stages. Unfortunately, most students will find tools to be more of a hindrance than a help during this period. There are two reasons:

- *Each tool has a high learning curve.* Because you are not familiar with the tools, you will find it tedious and cumbersome to use them initially. Please do not give up! Once you cross the initial hurdle, you will find the tools to be indispensable.
- *The project is less complex and smaller in the initial stages.* You will probably find the tools to be overkill during the first iteration because most of the tools are meant for large projects. However, you will realise the benefits of the tools as you move to later stages and when your product starts to get larger and more complex.

9.14 APPOINT TOOL 'GURUS'

While each student must learn basic usage of each tool selected, it is unrealistic to expect everyone to learn all of them in depth. Instead, become a 'Jack of all tools, a master of at least one'. A way to formalise this approach is to appoint one team member as the 'guru' for each tool (see tip 5.5 for more about gurus).

9.15 LOOK FOR BETTER TOOLS

Your instructor will not specify all the tools you can use. It is up to you to find tools that enhance your productivity. If the thought "there must be a smarter way of doing this!" ever crosses your mind during a project task, it is time to look for a tool.

If you find yourself writing code to do a common task (e.g. file handling, sorting, etc.), look for a library that provides that function.

9.16 PREFER ESTABLISHED SOFTWARE

Be wary of adopting any old tool/library you find on the Internet. Check the license first; it may not be free. Be extra careful if you are developing the product for an external client who intends to use it for commercial purposes.

While it may be OK to use evaluation versions of tools (as long as the evaluation period covers the project duration), note that your time spent on learning the tool will be lost unless you buy the tool later. On the other hand, incorporating time-limited evaluation versions of libraries into your product is not recommended; for one thing, it will prevent the archiving of your product for future reference. Be wary of libraries that require installing,

as this might cause problems if the product demo is to be done on a different PC. If in doubt, check with the instructor first.

It is safer to stick with stable and established software (such as those released by Apache Software Foundation). Do not believe in all the claims made by the tool/library authors. Check the forum/buglist of the software. See how many outstanding issues remain, how fast the author has responded to user queries, and how long the software has been around.

Be wary of bleeding edge tools (e.g. beta releases) - as you cannot afford to spend energy on battling an unstable tool. By the same token, be wary of using any old code sample you find from the Internet as they could very well contain bugs.

9.17 CHOOSE A COMMON SET OF TOOLS

Whenever possible, choose a common set of tools for the whole team to use. This increases the in-house expertise of each tool - something very much scarce in a student team.

9.18 AUTOMATE

Most tools are about automating things. In general, try to automate as much of the project work as possible.

Imagine you want to change a method name. You simply right click the method name and specify the new name, and all the references to that method are updated automatically. Then you press another button, which automatically compiles the code, links the code, builds the executable, runs the tests, updates the documentation, packages the product in to a distributable installer, uploads the installer to the website, updates the SCM system, updates the project tracking tool, and emails all users about the update. Now that's automation!

9.19 BE A TOOLSMITH

Sometimes you have to create your own tools. A specialised test driver is one example of such a tool. But do not reinvent the wheel.

9.20 DOCUMENT USAGE

In addition to the productivity gained by using tools, you can usually earn extra credit for being a good tool user if you document your use of tools.

Things you could document include:

- Which tools did you adopt in the project? For each tool, give a 2-3 line description of how you use each tool, and list (in point form) the features your 'guru' recommended for the team to use.
- Which other tools did you consider but did not adopt? Why?

- What other aspects of your project can benefit from more tools?

CHAPTER 10: DESIGNING THE PRODUCT

In a room full of top software designers, if two agree on the same thing, that's a majority. -- Bill Curtis

Most hard-core project courses put emphasis on the design aspect. Here are some things to keep in mind when designing your system.

10A. GENERAL DESIGN TIPS

10.1 ABSTRACTION IS YOUR FRIEND

A software design is a complex thing, especially, if you do not use abstraction to good effect. The design should be done at various levels of abstraction. At higher levels, you visualise the system as a small number of big components while abstracting away the details of what is inside each component. Once you have a good definition of what each of those components represent and how they interact with each other, you can move to the next lower level of abstraction. That is, you take each one of those components and design it using a small number of smaller components. This can go on until you reach a lower level of abstraction that is concrete enough to transform to an implementation.

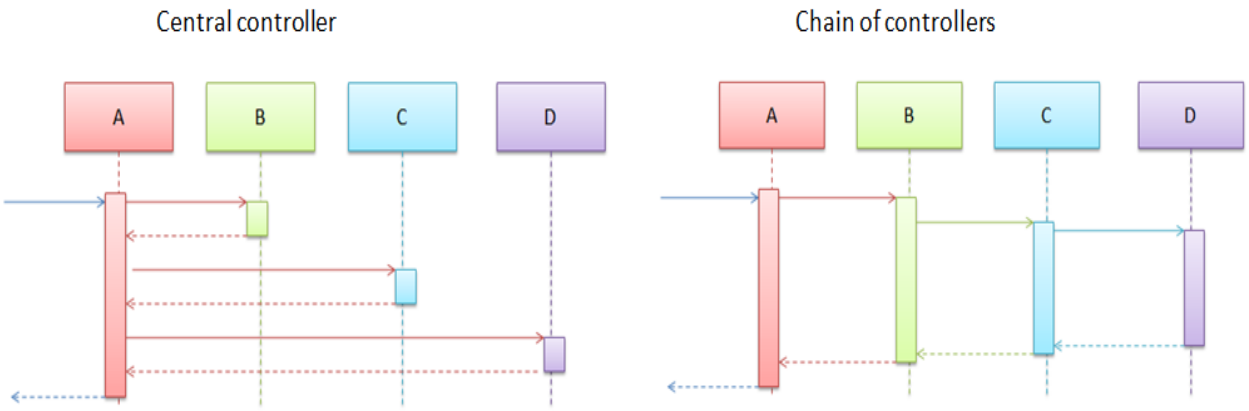
10.2 SEPARATE CONCERNS

We should try to keep different 'concerns' of the system as separated from each other as possible. For example, parsing (and things related to the 'parsing' concern) should be done by the *parser* component, and everything that has to do with sorting should be done by the *sorter* component.

10.3 DON'T TALK TO STRANGERS

Otherwise known as the *Law of Demeter*, the 'don't talk to strangers' principle advocates keeping unrelated things independent of each other. For example, if the *parser* component can function without any knowledge of the *sorter* component, then the *sorter* is a stranger to the *parser*. That means the *parser* should not have any reference to the *sorter* (e.g. can you compile the *parser* without compiling the *sorter*?)

A classic example where this principle applies is when choosing between a *central controller* model and *chain of controllers* model. The former lets you keep strangers as strangers, while the latter forces strangers to talk to each other. Notice how one design below lets *B* and *C* remain strangers while the other forces them to know each other.



Along the same vein, minimise communication between components. Avoid cyclic dependencies (e.g. *A* calls *B*, *B* calls *C* and *C* calls *A*)

10.4 ENCAPSULATE

A component should reveal as little as possible about itself. This is also known as *information hiding*. For example, other components that interact with your component should not know in what format your component stores data and they should not be allowed to manipulate those data directly.

10.5 PRESERVE CONCEPTUAL INTEGRITY

Fred Brooks contends that (in the book The Mythical Man-Month)

> *... the conceptual integrity is the most important consideration in system design. It is better for a system to omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.*

A student team is a team of peers that usually does not have a designated architect to dictate a design for others to follow. Everyone may want to have their say in the design. However, after discussing all alternative designs proposed, you should still choose one of them to follow, rather than devise a design that combines everyone's ideas. Combining ideas into one design has its merits, but do not do it just for the sake of achieving a compromise between competing ideas. If in doubt, get your supervisor's opinion as well.

10.6 STANDARDISE SOLUTIONS

Similar problems should be solved in a similar fashion. Sometimes, it pays to solve even slightly different yet largely similar problems in exactly the same way. It makes programs easier to understand. In other words, do not go out of your way to customise a solution to fit a problem *precisely*. It may be better to preserve the similarity of the solution instead.

10.7 USE PATTERNS

Patterns (design patterns [http://tinyurl.com/wikipedia-patterns], as well as other types of patterns such as analysis patterns, testing patterns, etc.) embody tried-and-tested solutions to common problems. Learn patterns and use them where applicable.

10.8 DO NOT OVERUSE PATTERNS

Most patterns come with extra baggage. Do not use patterns for the sake of using them. For example, there is no need to apply the Singleton pattern to *every* class that will have only one instance in the system; use it when there is a danger of someone creating multiple instances of such a class.

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies,and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*
> *--C.A.R. Hoare*

10.9 VALUE SIMPLICITY

Try to make the design as simple as possible (but no simpler!). Simple yet elegant designs are much better than complex solutions: The former is much harder to achieve however, but that is what you should strive for. Given any design, try to see whether you can simplify it. Resist any change that makes it more complex.

10.10 INCREASE *FAN-IN*, DECREASE *FAN-OUT*

When many other components use a given component (i.e. the component has high *fan-in*), this is a good thing because it increases reuse. When a given component uses many other components (i.e. it has high *fan-out*), this is not a good thing because it increases coupling.

10.11 GIVE BRUTE FORCE A CHANCE

Some problems have an obvious and simple brute force solution. Do not dismiss this solution too quickly in your haste to look for a smarter solution. If you can afford it, give this brute force solution a chance; it may be all you need.

10.12 LESS IS MORE

Trim all parts of the design that are not immediately useful to the system. It does not matter how elegant they are, how proud you are of dreaming them up, and how hard you worked at building them. The same applies to code.

10.13 DO NOT FORGET NON-FUNCTIONAL QUALITIES

Some non-functional qualities need to be incorporated from the design stage. One non-functional quality rarely mentioned in the specification and often forgotten in the design is the testability. Improving testability improves many other qualities of the design.

10.14 BEWARE OF PREMATURE OPTIMISATION

Hoare wasn't kidding when he said "Premature optimization is the root of all evil in programming". Opt for a simple design. If it is fast enough, stick with it. If it is not, find the real bottlenecks (profiling tools [Chapter 9] can be used for this purpose) and optimise accordingly.

Caveat: This does not mean that you should start with a stupid design. Some designs are obviously inefficient and should be discarded immediately. Start with a design that is, in Einstein's words "as simple as possible, but no simpler".

10.15 AVOID *ANALYSIS PARALYSIS*

During analysis and design, consider all the known facts but do not fret too much about unknowns. If you are given a concrete and stable specification (e.g. writing a parser for a given language) it would be stupid to start with a design that does not take the entire specification into account. Such short-sighted designs will eventually require change, causing rework that could have been avoided. On the other hand, if you are defining a first-of-a-kind exploratory system for an unspecified user base, go for a design with a reasonable degree of flexibility; do not worry about all the nitty-gritty issues that it might or might not have to face later.

10.16 POLISH DOCUMENTS LATER

Documenting design often requires wrestling with UML editors and other diagramming tools. Therefore, it is very frustrating when we have to modify those documents as the design changes over time.

While designs should be documented as they are done, there is no need to start creating well-polished design documents right away. You can keep the documentation as low-maintenance rough sketches (with none of the important points missing) until the design is sufficiently stable. For example, you can take a photo of the whiteboard on which you drew the initial design, print it out, do your (minor) modifications on the hard copy, and convert it to a digitised UML diagram much later.

10.17 REMEMBER THE REASON

As students, you are not expected to choose the best design in the first try. Designs often evolve over time. Be sure to document such changes so that you get credit for the effort spent in the process. Make sure your design has a rationale. Often evaluators ask "why did you choose this design (over another)?" No design is perfect. You

may be able to score extra credit by critically evaluating your final design, comparing it with an alternative design, and discussing ways of improving it further.

10B. UI DESIGN TIPS

The design of the UI is important because even the best functionality will not compensate for a poor user interface,and people will not use the system unless they can successfully interact with it
[adapted from the book Automated Defect Prevention: Best Practices in Software Management]

10.18 DESIGN IT FOR THE USER

The UI is for the user; design it for the user. It should be easy to learn, understand, remember, and navigate *for the user* (not you, the developer). A UI that makes the user wonders "where do I start?" the first time he looks at it is not an intuitive UI.

...user's aren't stupid; they just simply are't programmers. Users are all smart and knowledgeable in their own way. They are doctors and lawyers and accountants, who are struggling to use the computer because programmers are too 'stupid' to understand law and medicine and accounting [http://tinyurl.com/stupidprogrammers].

10.19 USABILITY IS KING

Do your best to improve the usability of the UI. Here are some things to consider:

- First, make it look presentable. If you do not have any graphics expertise in the team, just sticking to standard UI elements without trying anything fancy is a good idea. For example, this page shows different UIs developed by students for 14 software doing largely similar tasks.
- Minimise work for users.
    - Minimise clicks. If something can be done in one click, do not force the user to use two clicks. Stay away from 'cool' but useless UI adornments such as animated buttons.
    - Minimise pop-up windows. Some messages can be conveyed to the user without requiring him to click a pop-up dialogue.
    - Minimise unnecessary choices. Do not distract the user by forcing him to think about things not directly related to the task at hand.
- Minimise chances of user error. For example, place the 'delete permanently' button in a place that is unlikely to be clicked by mistake. If a user is not supposed to do a certain action at a certain stage, simply disable/remove that action from that stage. Double confirm with the user before the software obeys a user command that is irreversible.
- Where possible, give an avenue to recover from mistakes and errors. In addition to an understandable error message, try to suggest possible remedial actions.

- Minimise having to switch between the mouse and the keyboard frequently. Minimise travel distance for the mouse by placing all related 'click locations' close to each other.
- The UI should use terms and metaphors familiar to users, not terms invented by developers. Do not use different terms/icons to mean the same thing.
- Make common things easy, rare things possible. Do not bother everyone with things that only a few will care about. For example, if you think most users will be OK with the default install location, do not force everyone to click 'Yes, install here'; instead, provide a button to 'Change install location' that is optional to click.
- Make the UI consistent. It may be a good idea to let one person design the whole UI.
- The UI should leave no room for the user to wonder about the current system status. For example, the user should not wonder "Is it still saving or is it safe to close the application now?"
- Minimise things the user has to remember. For example, the user should not wonder "What save location did I choose in the previous screen?"
- A good UI should require very little user documentation, if any at all. On the other hand, "it is explained in the user guide" is not an excuse to produce unintuitive UIs.
- Think as a user when designing the UI. Even naming matters. Did you know that in one case, renaming a button increased revenue by $300 million [http://tinyurl.com/millionbutton]?

10.20 DO A PROTOTYPE

Do an early UI prototype/mockup and get feedback from teammates/instructor/client before implementing the whole system. UI prototypes are great in answering the "are we building the right system?" question. Note that a UI prototype need not have the final 'polished' look, and it need not be complete either. It can be drawn on paper, done using PowerPoint, drag-and-drop UI designers that come with IDEs such as Visual Studio, or using special prototyping tools such as Balsamiq.

10.21 BE DIFFERENT, IF YOU MUST

There is no need to follow what everyone one else does, but do not do things differently just for the heck of it. Deviating from familiar interaction mechanisms might confuse the user. For example, most Windows users are familiar with the 'left-click to activate, right-click for context menu' way of interacting, and will be irritated if you use a different approach.

10C. API DESIGN TIPS

An API that isn't comprehensible isn't usable. --James Gosling

An API (Application Programming Interface) is a set of operations a software component (i.e. a system, a sub-system, class) provides to its clients. For example, here [http://tinyurl.com/stringAPI] is the API of the Java *String* class.

A well-defined API is not only easy to use, but it gives the implementers (of that component) flexibility in implementation. As clients of the component write code against the API (i.e. they only know about the API), implementers have the freedom to change internal implementation details. As long as the API remains stable, client code will not have to change. For example, as clients of the *String* class, we only know its API, but we can use it without knowing how it is implemented internally.

When we design a system, we should use APIs to achieve a similar degree of freedom between components so that developers of those components can work independently of each other. For example, let us assume our system has two sub-systems: *front-end* and *back-end*. Let us also assume that the *front-end* uses the *back-end*. In this case we must first define the *back-end* API. Once that is defined, both the *front-end* team and the *back-end* team can start work in parallel instead of the *front-end* team waiting until the completed *back-end* is available. That is because the *front-end* team is assured that the *back-end* will support the agreed API irrespective of how it will be implemented. Therefore, defining APIs is an integral part of designing software.

10.22 TALK TO API CLIENTS

To define the API of a component, we must clearly understand the needs of the component's clients. The best way to discover an API of a component, therefore, is to talk to those who will use the component. For example, clients for the *back-end* include *front-end*. Therefore, the *back-end* API should be defined based on a discussion between the *front-end* team and the *back-end* team. Similarly, if you are writing a utility class to be used by many other classes in the system, you must define the API of your class based on the requirement of those client classes.

10.23 CHOOSE DESCRIPTIVE NAMES

API operation names and parameter names should have intuitive meanings. E.g. *setParent(parent, child)* is more meaningful than *setPrt(p,c).* The chapter on 'implementation' has some more tips about naming.

10.24 NAME CONSISTENTLY

If you have two operations *addVariable()* and *removeVar()*, you are not naming them consistently. Name the second operation *removeVariable()*, or the first operation *addVar()*. When using a consistently named API, users can simply deduce the correct operation to call without actually checking the API documentation.

10.25 HIDE INTERNAL DETAILS

The API should not reveal clues about the internal structures. For example, *addToVarHashtable()* reveals that we are using a *Hashtable* to store variables; what if later we want to change it to a *Vector*?

10.26 COMPLETE PRIMITIVE OPERATIONS

Make sure the component provides a complete set of primitive operations. You can add more specialised operations as necessary and implement them using the primitive operations. Even if you do not supply those specialised operations, clients will be able to accomplish such specialised tasks using the primitive operations. This avoids polluting the API with highly-specialised yet rarely-used operations.

10.27 AVOID BLOAT

If the API is becoming too long, may be the component is doing more than it should; consider splitting the component if appropriate. Or else, the component may be supporting too many operations to achieve the same thing. Strive to achieve a minimal yet complete API (refer tip 10.26).

10.28 AVOID 'SWISS ARMY KNIFE' OPERATIONS

Having multi-purpose methods causes unnecessary complexities, performance penalties, and misunderstandings. Make each operation do one thing and only one thing. Rather than have a *writeToFile(string text, boolean isAppend)* where the second parameter indicates whether to overwrite or append, have two methods *appendToFile(string text)* and *overwriteFile(string text)*.

Caveat: An over-enthusiastic application of this guideline could lead to a bloated API [see tip 10.27]

10.29 MAKE COMMON THINGS EASY, RARE THINGS POSSIBLE

This is another way to look at the previous two points. On the one hand the API should provide direct and easy-to-use ways to accomplish common tasks. On the other hand, rarely-used highly-specialised tasks need not be directly supported; instead, we can let clients implement these operations themselves, using a combination of existing primitive operations.

10.30 PROMISE LESS

An API is a contract (between the implementer of the component, and users of the component); just like any contract, promising less makes it easier to deliver. E.g. If an operation is for calculating the root of positive integers, it should be *findRoot(PositiveInt i)*, not *findRoot(int i)*

10.31 SPECIFY ABNORMAL BEHAVIOURS

An operation should throw exceptions to indicate exceptional situations (e.g. when an input file was not found). These should be clearly specified in the API. A common mistake is to specify only the typical behaviour.

10.32 CHOOSE ATOMIC OVER MULTI-STEP

When a set of steps is always done together in the same sequence, capture that as one atomic operation instead of having separate operations for each step. For example, imagine that your component has to parse a file that contains some program code, and create the corresponding abstract syntax tree (AST).

- Option 1: Have one operation that takes the file name as the parameter and returns the resultant AST *parseFile(String fileName): AST*
- Option 2: Have three operations *setFileName(String fileName)*, *parse()*, *getAST(): AST*

Clearly the first option is better; it makes the API shorter and reduces chances for error (what if we call *parse* method without calling the *setFileName* method first?).

10.33 MATCH THE ABSTRACTION

Operations in the API should be in tune with the abstraction the API represents. For example, a Parser API should use terms related to Parsing. This is also applicable to the exceptions thrown by the API. For example, a Parser API should throw exceptions that are related to a Parser.

10.34 KEEP TO THE SAME LEVEL OF ABSTRACTION

Try to keep all operations of an API on the same level of abstraction. An API that has a mix of high-level and low-level operations is harder to understand. For example, the File API should not mix high-level operations such as open(), close(), append(String) with low-level operations such as those for direct manipulation of bits inside the file.

10.35 EVOLVE AS NECESSARY

While we try to keep APIs stable, it is not uncommon for changing client requirements to force changes in existing APIs. If you have multiple clients, make sure that a change requested by one client does not adversely affect the other clients.

Furthermore, we sometimes evolve the API as our understanding of the client requirements improves.

It is OK to specify the data types in the operation header using symbolic names, and later refine them into concrete types. For example, the return type can be specified as *list-of-TreeNodes* and later refined to *TreeNode[]* or *ArrayList<TreeNode>.*

10.36 DOCUMENT THE API

The API documentation should be precise, unambiguous, concise and complete. Do not leave room for the client to make assumptions. Instead, explicitly specify what a method does. You can use this API description of the Java *String* class as a guiding example when documenting APIs, although yours need not be as elaborate.

10.37 GENERATE DOCUMENTATION FROM CODE

Find a way to generate the API documentation from the code itself and comments therein. An example of this strategy in action is the Java API documentation generated by Javadoc comments embedded in the code. Doxygen is another tool that can help here.

CHAPTER 11: IMPLEMENTATION

Any fool can write code that a computer can understand. Good programmers write code that humans can understand. --Martin Fowler [in the book Refactoring: Improving the Design of Existing Code]

A bad job of coding is a sure path to a low grade, but a good job of coding does not necessarily mean a good grade. Here are some tips to ease your implementation pains.

11.1 REFACTOR OFTEN

Since you are rather inexperienced in programming, no one expects you to write quality code in the first attempt. However, if you decide to live with messy code you produce, it will get messier as time goes on (have you heard about the 'broken windows theory' [http://tinyurl.com/refactoringstory] ?). The remedy is refactoring [http://tinyurl.com/code-refactoring]. Refactor often to improve your code structure.

Refactoring code regularly is as important as washing regularly to keep your privates clean. Neglecting either will result in a big stink. Analogies aside, any time spent on refactoring will not only reduce your debugging time, the quality improvement it adds to your code could even earn you extra credit.

11.2 NAME WELL

Proper naming improves the code quality immensely. It also reduces bugs caused by misunderstandings about what a variable/method does. Here are some things to consider:

- Related things should be named similarly, while unrelated things should NOT be named similarly.
- Use nouns for classes/variables and verbs for methods.
- A name should completely and accurately describe what it names. For example, processStuff() is a bad choice; what 'stuff'? what 'process'?. removeWhiteSpaceFromInput() is a better choice. Other bad examples include flag, temp, i (unless used as the control variable of a loop).
- It is preferable not to have 'too long' names, but it is OK if unavoidable. 'Too short' names are worse.
- Use correct spelling in names (and even in comments). Avoid texting-style spelling.
- If you must abbreviate or use acronyms, do it consistently, and explain their full meaning in an easy-to-find location.
- Avoid misleading names (e.g. those with double meaning), similar sounding names, hard to read ones (e.g. avoid having to ask "is that simple l or capital I or number 1?", or "is that number 0 or letter O?"), almost similar names, foreign language names, slang, and names that can only be explained using private jokes.
- Minimise using numbers or case to distinguish names (value1 vs value2, Value vs value).

- Distinguish clearly between single valued and multivalued variables (e.g. Person student, ArrayList<Person> student**s**).

11.3 BE OBVIOUS

We can improve understandability of the code by explicitly specifying certain things even if the language syntax allows them to be implicit. Here are some examples:

- Use explicit type conversion instead of implicit type conversion.
- Use parentheses to show grouping even when you can skip them.
- Use enumerations when a certain variable can take only a small number of finite values. For example, instead of declaring the variable 'status' as an integer and using values 0,1,2 to denote statuses 'starting', 'enabled', and 'disabled' respectively, declare 'status' as type SYSTEM_STATUS and define an enumeration called SYSTEM_STATUS that has values 'STARTING', 'ENABLED', and 'DISABLED'.
- Use comments to identify the end of a long block with deep nesting (better yet, avoid writing such blocks altogether).
- When statements should follow a particular order, try to make it obvious (with appropriate naming, or at least comments). For example, if you name two functions 'phaseOne()' and 'phaseTwo()', it is obvious which one should be called first.

11.4 NO MISUSE OF SYNTAX

It is safer to use language constructs in the way they are meant to be used, even if the language allows shortcuts. Here are some examples:

- Use the 'default' option of a case statement for a real default and not simply to execute the last option. If there is no default action, you can use the 'default' branch to detect errors (i.e. if execution reached the 'default' branch, throw an exception). This also applies to the final 'else' of an if-else construct. That is, the final 'else' should mean 'everything else', not the final option. Do not use 'else' to catch an option that can be specified more specifically, unless that is the absolutely the only other possibility.

Not recommended	if (red) print "red"; else print "blue";

Better	if (red) print "red";
	else if (blue) print "blue";
	else error("incorrect input");

- Use one variable for one purpose. i.e. do not reuse a variable for a different purpose than for what it was declared.
- Do not reuse parameters received as local variables inside the method.
- Avoid data flow anomalies such as variable being used before initialization and assigning values to variables but changing it later without using the value.

11.5 AVOID ERROR-PRONE PRACTICES

Some coding practices are notorious as sources of bugs, but nevertheless common. Know them and avoid them like the plague. Here are some examples:

- Never write a case statement without the 'default' branch.
- Never write an empty 'catch' statement.
- All 'opened' resources must be closed explicitly. If it was opened inside a 'try' block, it should be closed inside the 'finally' block.
- If several methods have similar parameters, use the same parameter order.
- Do not include unused parameters in the method signature.
- Whenever you do an arithmetic calculation, check for 'divide by zero' problems and overflows.

11.6 USE CONVENTIONS

When working in a team, it is very important for everyone to follow a consistent coding style. Appoint someone to oversee the code quality and consistency in coding style (i.e. a code quality guru). He can choose a coding standard to adopt. You can simply choose one from many coding standards floating around for any given language. Prune it if it is too lengthy - circulating a 50-page coding standard will not help. But note that a coding standard does not specify everything. You still have to come up with some more conventions for your team e.g. naming conventions, how to organise the source code, which 'error-prone' coding practices to avoid, how to do integration testing, etc.

Conventions work only if the whole team follows them. You can also look for tools that help you enforce coding conventions.

11.7 MINIMISE GLOBAL VARIABLES

Global variables may be the most convenient way to pass information around, but they do create implicit links between code segments that use the global variable. Avoid them as much as possible.

11.8 AVOID MAGIC NUMBERS

Avoid indiscriminate use of numbers with special meanings (e.g. 3.1415 to mean the value of mathematical ratio PI) all over the code. Define them as constants, preferably in a central location.

```
[Bad]    return 3.14236;
[Better] return PI;
```

Along the same vein, make calculation logic behind a number explicit rather than giving the final result.

```
[Bad]    return 9;
[Better] return MAX_SIZE-1;
```

Imagine going to the doctor's and saying "My nipple1 is swollen"! Minimise the use of numbers to distinguish between related entities such as variables, methods and components.

```
[Bad] value1, value2
[Better] originalValue, finalValue
```

A similar logic applies to string literals with special meanings (e.g. "Error 1432").

11.9 THROW OUT GARBAGE

We all feel reluctant to delete code we wrote ourselves, even if we do not use that code any more ("I spent a lot of time writing that code; what if we need it again?"). Consider all code as baggage you have to carry; get rid of unused code the moment it becomes redundant. Should you eventually need that code again, you can simply recover it from the code versioning tool you are using (you are using one, are you not?). Deleting code you wrote previously is a sign that you are improving.

11.10 MINIMISE DUPLICATION

The Pragmatic Programmer book calls this the DRY (Don't Repeat Yourself) principle. Code duplication, especially when you copy-paste-modify code, often indicates a poor quality implementation. While it is not possible to have zero duplication, always think twice before duplicating code; most often there is a better alternative.

11.11 MAKE COMMENTS UNNECESSARY

> *Good code is its own best documentation. As you're about to add a comment, ask yourself, 'How can I improve the code so that this comment isn't needed?' Improve the code and then document it to make it even clearer. --Steve McConnell*

Some students think commenting heavily increases the 'code quality'. This is not so. Avoid writing comments to explain bad code. Try to refactor the code to make it self-explanatory.

Do not use comments to repeat what is already obvious from the code. If the parameter name already clearly indicates what it is, there is no need to repeat the same thing in a comment just for the sake of writing 'well documented' code. However, you might have to write comments occasionally to help someone to help understand the code more easily. Do not write such comments as if they are private notes to self. Instead, write them well

enough to be understandable to another programmer. One type of code that is almost always useful is the header comment that you write for a file, class, or an operation to explain its purpose.

When you write comments, use them to explain 'why' aspect of the code rather than the 'how' aspect. The former is not apparent in the code and writing it down adds value to the code. The latter should already be apparent from the code (if not, refactor the code until it is).

The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility,and among other things he avoids clever tricks like the plague. --Edsger Dijkstra

11.12 BE SIMPLE

Often, simple code runs faster. In addition, simple code is less error-prone and more maintainable. Do not dismiss the brute force yet simple solution too soon and jump into a complicated solution for the sake of performance, unless you have proof that the latter solution has a sufficient performance edge. As the old adage goes, KISS (keep it simple, stupid - don't try to write clever code).

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. --Brian W. Kernighan

Programs must be written for people to read, and only incidentally for machines to execute. --Abelson and Sussman

11.13 CODE FOR HUMANS

In particular, always assume anyone who reads the code you write is dumber than you (duh). This means you need to make the code understandable to such a person. The smarter you think you are compared to your teammates, the more effort you need to make your code understandable to them. You might ask why we need to care about others reading our code because, in the short span of a class project, you can always be there to handle your own code. Here are some good reasons:

- Evaluators might read the code to assess the code quality. Yes, they are not dumber than you (right?), but the more readable you make it, the easier it will be for them to understand it. No evaluator will give you extra credit for writing code they have a hard time figuring out.
- We all become 'strangers' to our own code after we spend some time away from it. In a student project, you *will* have to spend time away from the project due to other courses you are taking.
- It is always an advantage to let other team members review or modify your code.
- Finally, you should code the class project as if you are coding a real project in the industry.

If you think the above reasons are good enough to try and write human-readable code, here are some tips that can help you there:

- Avoid long methods. Be wary when a method is longer than the computer screen, and take corrective action when it goes beyond 200LOC. The bigger the haystack, the harder it is to find a needle.
- Limit the depth of nesting. (how many levels is too many? This is what *Linux 1.3.53 CodingStyle documentation* says: *If you need more than 3 levels of indentation, you're screwed anyway, and should fix your program*.)

- Have only one statement per line.

- Avoid complicated expressions, especially those having many negations and nested parentheses. Sure, the computer can deal with complicated expression; but humans cannot. If you must evaluate complicated expressions, do it in steps (i.e. calculate some intermediate values first and use them to calculate the final value).
- The default path of execution (i.e. the path taken if everything goes well) should be clear and prominent in your code. Someone reading the code should not get distracted by alternative paths taken when error conditions happen. One technique that could help in this regard is the use of *guard clauses* [http://tinyurl.com/guardclause].

11.14 TELL A GOOD STORY

Lay out the code to follow the logical structure of the code. The code should read like a story. Just like we use section breaks, chapters and paragraphs to organise a story, use classes, methods, indentation and line spacing in your code to group related segments of the code. For example, you can use blank lines to group related statements together.

Sometimes, the correctness of your code does not depend on the order in which you perform certain intermediary steps. Nevertheless, this order may affect the clarity of the story you are trying to tell. Choose the order that makes the story most readable.

A sure way to ruin a good story is to tell details not relevant to the story. Do not vary the level of abstraction too much within a piece of code.

[Bad]
```
readData();
salary = basic*rise+1000;
tax = (taxable?salary*0.07:0);
displayResult();
```

[Better]
```
readData();
processData();
displayResult();
```

11.15 DO NOT ASSUME. CLARIFY. ASSERT.

When in doubt about what the specification or the design means, always clarify with the relevant party. Do not simply assume the most likely interpretation and implement that in code. Furthermore, document the clarification or the assumption (if there was no choice but to make one) in the most *visible* manner possible, for example, using an assertion [http://tinyurl.com/wikipedia-assertion] rather than a comment. You can use assertions to document parameter preconditions, input/output ranges, invariants, resource open/close states, or any other assumptions.

11.16 CHOOSE AN APPROACH AND STICK WITH IT

Choose the *Design-by-Contract (DbC)* [http://tinyurl.com/wikipedia-dbc] approach or the *defensive programming* [http://tinyurl.com/wikipedia-dp] approach and stick to it throughout the code:

- DbC assumes that users of our component will meet all stated pre-conditions before using our component. If the programming language does not have in-built support for DbC, we can use assertions to enforce pre-conditions. The system will halt immediately if our component is used without meeting preconditions.
- Defensive programming is like defensive driving (i.e. since we are never sure what other drivers will do, we should drive in a way to maximise our own safety and not depend on other drivers to do the right thing). When we write a component, we assume we are the author of this unit only. We do not depend on others to use our component correctly. Instead, we write the component to survive even if others tried to use it in incorrect ways. That means our component has to respond to inappropriate usage with a suitable error recovery technique.

11.17 DO NOT RELEASE TEMPORARY CODE

We all get into situations where we need to write some code 'for the time being' that we hope to dispose of or replace with better code later. Mark all such code in some obvious way (e.g. using an embarrassing print message) so that you do not forget about their temporary nature later. It is irresponsible to release such code for others' use. Whatever you release to others should be worthy of the quality standards you live by. If the code is important enough to release, it is important enough to be of production quality.

11.18 HEED THE COMPILER

Instead of switching off compiler warnings, turn it to the highest level. When the compiler gives you a warning, act on it rather than ignore it. A compiler is better at detecting code anomalies than the most of us. It is wise to listen to it when it wants to help us.

11.19 STRATEGISE ERROR HANDLING

The error handling strategy is often too important to leave to individual developers' discretion. Rather, it deserves some pre-planning and standardisation at team level.

Response to an error should match the nature of the error. Handle local errors locally, rather than throwing every error back to the caller. Some systems can benefit from centralised error handling where all errors are passed to a single error handler. This technique is especially useful when errors are to be handled in a standard yet non-trivial manner (e.g. by writing an error message to a network socket).

11.20 HAVE SUFFICIENT VERBOSITY

A sufficiently verbose/interactive application avoids the doubt "is the application stuck or simply busy processing some data?" For example, if your application does an internal calculation that could take more than a couple of seconds, produce some output at short intervals during such a long process (e.g. show the number of records processed at each stage).

11.21 FLY WITH A BLACK BOX

Some bugs crops up only occasionally and they are hard to reproduce under test conditions. If such a bug crashes your system, there should be a way to figure out what went wrong. If your application writes internal state information to a log file at different points of operation, you can simply look at that log file for clues. This is the same reason why an aircraft carries a *black box* (aka Flight Data Recorder).

11.22 GREET THE USER, BUT BRIEFLY

Printing a welcome message at startup (or a splash screen, if your application has a GUI) - showing some useful data like the version number - is not just good manners; it also makes our application look more professional and helps us catch problems of 'releasing an outdated version'. However, such preliminaries should not take too long and should not annoy the user.

11.23 SHUTDOWN PROPERLY

The ability to do a clean shutdown is an important part of a system functionality most students seem to forget, especially, if there are some tasks to be done during the shutdown, such as saving user preferences or closing network connections.

11.24 STICK TO THE DOMAIN TERMS

Most students invent their own set of terminology to refer to things in the problem domain. This terminology eventually makes it to the program code, documentation, and the UI. Instead, stick to the terms used in the

problem domain. For example, if the specification calls a component the 'projector', do not refer to the same component as 'the visualiser' in your code. While the latter may be a better match for what that component actually does, it could cause unnecessary confusion and much annoyance for the evaluator/supervisor/customer.

Programs should be written and polished until they acquire publication quality.
--Niklaus Wirth

11.25 THROW AWAY THE PROTOTYPE

A 'throw-away prototype' (not to be confused with an 'evolutionary prototype') [http://tinyurl.com/wikipedia-prototyping] is not meant to be of production quality. It is a quick-and-dirty system you slapped together to verify certain things such as the architectural viability of a proposed product. It is meant to be thrown away and you are supposed to keep only the knowledge you gained from it. Having said that, most of you will find it hard to throw the prototype away after spending so much time on it (because of your inexperience, you probably spent more time on the prototype than it really required). If you really insist on building the real system on top of the prototype (i.e. starting to treat a throw-away prototype as an evolutionary prototype), devote the next iteration to converting the prototype to a production quality system. This will require adding unit tests, error-handling code, documentation, logging code [see tip 11.21], and quite a bit of refactoring [see tip 11.1].

11.26 FIX BEFORE YOU ADD

Most bugs are like needles in a haystack. Piling on more hay will not make them easier to find. It is always preferable to find and fix as many bugs as you can before you add more code.

11.27 NEVER CHECK-IN BROKEN CODE

Before you release your own code, check-out the latest code from others, do a clean build, and run all the test cases (having automated test cases will be a great help here). The biggest offense you can do while coding is releasing broken code to the code base.

Even if it is a tiny change and it cannot have possibly broken anything, do all of the above anyway, just to be sure.

11.28 DO NOT PROGRAM BY COINCIDENCE

You code a little, try it, and it seems to work. You code some more, try it, and it still seems to work. After many rounds of coding like this, the program suddenly stops working, and after hours of trying to fix it, you still cannot figure out why. This approach is what some call 'programming by coincidence' [http://tinyurl.com/prog-by-coin]; your code worked up to now by coincidence, not because it was 100% correct.

Instead, code a little, test it thoroughly, only then you code some more. See chapter 13 for more tips on testing.

... nearly everybody is convinced that every style but their own is ugly and unreadable. Leave out the "but their own" and they're probably right... --Jerry Coffin (on indentation)

11.29 READ GOOD CODE

How can you write a great novel, if you have not even read any great novels? How can you write great code if you have not even seen any great code? Take a look at 'good code' written by someone else. This easiest way to find a good code sample is to dig into a well-established open source project. Even if you cannot understand what that code does, at least you will know what well-written production code looks like.

11.30 DO CODE REVIEWS

Code reviews promote consistent coding styles within the team, give better programmers a chance to mentor weaker ones, and motivate everyone to write better code.

Hold at least one code-review session at an early stage of the project. Use this session to go over each other's code. Discuss the different styles each one has adopted, and choose one for the whole team to follow.

11.31 STOP WHEN 'GOOD ENOUGH'

There is no such thing as 'perfect' code. While you should not release sloppy software, do not hold up progress by insisting on yet another minor improvement to an already 'good enough' code.

11.32 CREATE INSTALLERS IF YOU MUST

Having an installer gives you several advantages, one of which is creating a more 'professional' impression of your software. There are several tools available today that can create an installer for your software very easily. But note that one major disadvantage of installers is some potential users might resist installing 'untested' software for the fear of 'slowing down the computer' or 'corrupting the registry'. If your software can do without a installer, then it is best not to have one.

11.33 SIGN YOUR WORK

To quote from book The pragmatic programmer: We want to see pride of ownership. "I wrote this, and I stand behind my work."

By making you put your name against the code your wrote (e.g. as a comment at the beginning of the code), we encourage you to write code that you are proud to call your own work. This is also useful during grading the quality of work by individual team members.

11.34 RESPECT COPYRIGHTS

This is especially important if you release your product to the public. Whenever you reuse anything in your product, make sure the owner has explicitly granted others the right to reuse it. Just because you found it floating on the Internet does not mean you have the right to reuse it. This applies to, but not limited to, images, videos, sound tracks, code samples, libraries, and even terminology.

11.35 HAVE FUN

To most of us, coding is fun. If it is not fun for you, you are probably not doing it right.

'HOW TO' SIDEBAR 11A: HOW TO ACHIEVE NON-FUNCTIONAL QUALITIES

The bitterness of poor quality remains long after the sweetness of meeting the schedule has been forgotten. --Anonymous

Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little. --Bertrand Meyer

Functional correctness is usually the most important quality of a system. To achieve functional correctness, we should understand the specifications well (we need to know what is the 'correct' behaviour expected in the first place).

In addition, a system is required to have many *non-functional qualities* (i.e. qualities that are not directly related to its functionality), some explicitly mentioned, some only implied. The nature of the system decides which qualities are more important. Find out the relative importance of each quality, as you might have to sacrifice one to gain another. There are many to choose from such as Analysability, Adaptability, Changeability, Compatibility, Configurability, Conformance, Efficiency, Fault tolerance, Installability/uninstallability, Interoperability Learnability, Localizability, Maintainability, Portability, Performance, Replaceablity, Reliability, Reusability, Security, Scalability, Stability, Supportability, Testability, Understandability, Usability.

Most grading schemes have some marks allocated for certain NF qualities. Note that some NF qualities are not easily visible during a product demo. Be sure to talk about them during the presentation, and in the report. Given next are some common NF qualities and some tips on achieving each.

The unavoidable price of reliability is simplicity. -- C.A.R. Hoare
Simplicity is prerequisite for reliability --Edsger W.Dijkstra

11.36 RELIABILITY

You need to show that your system can perform without failure for the specified operating environment. If the system is expected to run for a long period, you should ensure that it indeed can. For example, monitor the memory usage while the system is in operation; if it keeps increasing, your system might have a memory leak, which will crash it eventually.

Software and cathedrals are much the same - first we build them, then we pray. --Anonymous

11.37 ROBUSTNESS

Can your system withstand incorrect input? Will it gracefully terminate when pushed beyond the operating conditions for which it was designed? Does it warn the user when approaching the breaking point or simply crash without warning? If you can crash your system by providing wrong input values, then it is not a very robust system.

Always code as if the person who ends up maintaining your code will be a violent psychopath who knows where you live. -- Anonymous

11.38 MAINTAINABILITY

Do not think maintainability is unimportant because this is just a class project or because the instructor did not mention it specifically. The truth is that you will 'maintain' your code from the moment you write it. You will really regret writing unmaintainable code especially during bug-fixing and modifying existing features towards the end of the project.

Keeping the design and eventually, its implementation, simple and easy-to-understand can really boost the maintainability of your system. A comprehensive suite of automated test cases can detect undesirable ripple effects of a change on the rest of the system. Having such tests indirectly helps maintainability.

Maintainability is of utmost importance if you plan to release the product to the public and keep it alive after the course.

11.39 EFFICIENCY

As software engineers, we should always give due consideration to the efficiency of what we are building. It becomes especially important if multiple teams are building similar products and more efficient products stand to gain extra marks. Here are some guidelines you can consider adopting.

- **Define.** Find out which kind of efficiency you should be focussing on. It could be the space efficiency (how efficiently you use storage space) that you should worry about. It could also be the time efficiency (how

fast the system can work). In some cases, it can be a more specific aspect such as the time it takes to process a certain input.

- **Do not speculate; find out.**
  - If you try to optimise the parts that you *think* might be slow, you could end up optimizing the wrong part, especially, given your inexperience in the field. While you are busy looking for a super-fast sorting algorithm, your system could be slow simply because you use *Strings* instead of *StringBuffers*.
  - Use appropriate tools (e.g. profilers such as CLRProfiler and DotTrace) to find out where in the system you need to optimise for performance. Note that black-box performance testing is different from profiling. Remember to stay focussed on the specific aspect you want to profile. Do not get distracted by different kinds of data produced by the profiling tool. Sometimes, it is good to use multiple tools to confirm a finding, especially if the findings appear counter-intuitive at first.
  - You can use profiling to test hypotheses too, but if you limit yourself to hypothesis testing only, you will miss the chance to discover other interesting things you did not even think of when formulating your hypotheses.
  - If you want to test how the system responds to one parameter, you should vary only that parameter while keeping other parameters fixed.
  - Document your findings. Clearly state under which circumstances and using which tools the profiling was done. Include a short summary of the findings. Dumping a lot of data is not very helpful by itself.
  - Show evidence of your findings/conclusions. Including a screenshot adds to the credibility (i.e. it proves that you did not simply make up the numbers).
- **Prove your claims**. Provide results of performance testing to show how efficient your system is. If your system should be able to handle a million data items, provide evidence of how well it handles a million data items.
- **Benchmark.** You can also consider benchmarking your system against other existing alternatives. You can also benchmark against previous versions of your own system to show how you improved the efficiency of your system over time.

11.40 SECURITY

Security is a prime concern for most systems, especially for multi-user systems with open access (such as Internet-based applications). If security is a top priority for your system, make sure you have it built into the system from the very beginning. It is much harder to secure a system built based on an unsecure architecture.

Before software can be reusable it first has to be usable. --Ralph Johnson

11.41 REUSABILITY

Under this aspect, you can address the following questions:

- How well have you reused code in the part you wrote? For example, did you simply copy-paste-modify functions instead of writing generic reusable functions?
- How well did you reuse code across different parts of your system (written by different team members)?
- How reusable is your code in other similar systems or future versions of this system?

If you have a lot of duplicate code in your system (some code analysers can easily find this out for you), you have not reused code well.

If you can't test it, don't build it. If you don't test it, rip it out.
--Boris Beizer, in the book 'Software testing techniques'

11.42 TESTABILITY

Testability does not come by default; some designs are more testable than others. To give a simple example, methods that return a value are easier to test than those that do not. If you give up testing some parts of your system because it is too difficult to test, you have a testability problem. If you have good testability, it is possible to test each part of your system in isolation and with reasonable ease. During integration, a system with good testability can be tested at each stage of the integration, integrating one component at a time.

11.43 SCALABILITY

How well can your system handle bigger loads (bigger input/output, more concurrent users, etc.)? Does the drop in performance (if any) reasonably reflect the increase in workload?

Every program has (at least) two purposes: the one for which it was written,
and another for which it wasn't. --Alan J. Perlis

11.44 FLEXIBILITY/EXTENSIBILITY

How easily can you change a certain aspect of your system? For example, can you change the database vendor or the storage medium (say, from a relational database to a file-based system) without affecting the rest of the system? Can you replace a certain algorithm in your system by replacing a component?

How easily can you extend your system (to handle other types of input, do other types of processing, etc.)? For example, can you add functionality by simply plugging in more code, or do you need extensive changes to existing code to add functionality?

We have to stop optimizing for programmers and start optimizing for users. --Jeff Atwood

11.45 USABILITY

Does the usability of your system match the target user base? Is the operation intuitive to the target user base? Have you produced good user manuals? Does your system let users accomplish things with a minimal number of steps? Tip 10.19 can help you further in this aspect.

CHAPTER 12: TESTING

Testing is the process of comparing the invisible to the ambiguous, so as to avoid the unthinkable happening to the anonymous. --James Bach

Adequate testing is a critical success factor for your project. Often, good teams produce inferior products due to inadequate testing.

12A. TEST PLANNING

12.1 KNOW HOW MUCH TO TEST

A frequent question from students: "How many test cases should we write?"; Answer: "As many as you need". There is no need to write a single test case if you are sure that your system works perfectly. In practice, it is not unusual to have more testing code (i.e. test cases) than functional code. On the other hand, the number of test cases by itself does not guarantee a good product. It is the quality of the test cases that matters. In fact, it is the quality of the product that matters. Good testing is just one means of getting there. However, evaluators rarely have enough time to measure the quality of your product rigorously and might use the quality of test cases as a measure of the quality of the product. That is one more reason to write good test cases.

You should adjust your level of testing based on the following:

- **The type of product you are building**: If you are building a proof-of-concept prototype, concentrate on testing functions related to the concept you are trying to prove. If you are building a product that will be used by a client or released to the public, you need much more testing, including testing for invalid inputs and improper use.
- **How it will be evaluated**: If the product is evaluated based on a demo and you will be in control of the demo, test the functionality you will use during the demo. If you may not be in control of the demo (that is, the evaluator will direct the flow of the demo), also test alternate paths of the use cases you will demo.
- **How much the product quality will affect the grade**: Which aspects of the product you should test (performance, accuracy, error handling, usability, etc.) and how much you should test each depends on what percentage of your grade depends on it. Even if concrete percentage figures are not given, try to find out what aspects the evaluator values most.

Most student teams underestimate the testing effort. Testing coupled with subsequent debugging and bug fixing will take the biggest bite out of your project time. However, most students start by allocating to testing a much smaller share of resources than they allocate for development.

When correctness is essential, at least ~25-35% of the schedule should be for system level testing, not counting *developer testing*. Another good rule of thumb (mentioned in the book *UML Distilled* by Martin Fowler) is that unit tests should be at least as the same size as the production code.

12.2 HAVE A PLAN

"We test everything" is not a test plan. "We test from 13th to 19th" is not detailed enough. A test plan is not a list of test cases either. A test plan includes what will be tested in what order by whom, when will you start/finish testing, and what techniques and tools will be used.

12.3 PUT SOMEONE IN CHARGE

Testing is so important that you should put someone in charge of it even if you do not have anyone in charge of other aspects of the project. However, this does not mean only that person will do testing. Doing so will create a single point of failure for your project; if the so called 'tester' does not do his job well, all the good work done by others will be in vain.

12.4 INSIST ON ITERATIVE

If you have to move 100 bricks from point A to B within 10 hours, which method do you prefer: carry all 100 bricks and run from A to B during the last 10 minutes, or walk 10 times from A to B carrying 10 bricks at a time? If you prefer the latter, insist that the team follows an iterative development process; it will make the testing feel like walking with 10 bricks rather than running with 100 bricks.

12.5 CONSIDER TEST-DRIVEN

Code Complete (the book by Steve Mcconnell) says "... test-first programming is one of the most beneficial software practices to emerge during the past decade and is a good general approach". Test-Driven Development (TDD), referred to as 'test-first programming' in this quote, advocates writing test cases before writing the code. While TDD has its share of detractors, it is considered an exciting way to develop code, and its benefits outweigh the drawbacks (if any). It is certainly suitable for student projects. It might feel a bit counter-intuitive at first, but it feels quite natural once you have got used to it.

12.6 HAVE TESTING POLICIES

Define testing policies for your project (e.g. how to name test classes, the level of testing expected from each member, etc.). This can be done by the testing guru [see tip 5.5]. Here are some reasonable test policies you could adopt (examples only):

- For large components, its users should write black-box tests in TDD fashion. For smaller components, its developer should write white-box tests (TDD optional).

- Standalone utilities likely to be reused should be tested more than onetime-use components. Use assertions to protect them against misuse/overuse.
- An API exposed to other team members should be tested more than those we write for our own use.
- When unit-testing a higher-level component *H* that depends on a lower-level component *L*: if *L* is 'trusted' (i.e. already unit-tested), we do not replace *L* with a stub when unit testing *H*. However, if *H* is a highly critical component, we insist unit-testing it in 100% isolation from *L*.

12.7 INSIST ON DEVELOPER TESTING

Students often ask "Isn't it enough just to do system testing; if the *whole* works, then *parts* must surely work, right?"

- Bugs start like small fish in a small pool (pool = the component in which the bug resides). You should catch them with little nets of unit testing. Let them loose in the ocean (i.e. if you integrate the buggy component into the system), they multiply and hide, and become very hard to catch.
- If you catch your own bug using unit testing, you can fix it without anybody else knowing. If you wait till the system testing, and the bug will be counted against your name - oh, the shame!
- In general, the later you encounter a bug, the costlier it will be to find/fix.

12.8 CONSIDER CROSS TESTING

Cross-testing means you let a teammate test a component you developed. This does not mean you do not test it yourself; cross-testing is done in addition to your own testing. Cross-testing is additional work, delays the project, and is against the spirit of "being responsible for the quality of your own work". You should use it only when there is a question of 'low quality work' by a team member or when the component in question is a critical component. Any bug found during cross-testing should go on the record, and should be counted against the author of the code.

12.9 CHOOSE COMPETENT TESTERS

Everyone must unit-test their own code, and do a share of the integration/system testing as well. If your course allows choosing dedicated testers, choose competent testers. While you may or may not choose your strongest team members as testers, testing is too important to entrust to the weakest ones either.

12.10 AUTOMATE TESTING

While the instructor might not insist on fully automated testing, note the following:

- Yes, creating automated tests is harder than manual testing; but the effort will pay off as you can repeat the same tests many times later on, without any extra effort.
- The more automated testing you have, the better the quality of your eventual system (because all latent changes will be put through the same battery of tests to prevent any regressions).

- The more automated testing you have, the more credit you can claim for 'applying good software engineering practices'.
- Less automated testing often implies a system with low testability, which in turn implies room for improvement in the design (a good design results in a testable system).

Furthermore, it is natural to automate unit and integration testing as lower level components cannot be tested manually anyway because they do not have a user interface.

12.11 SMOKE TEST

A 'smoke test' is a set of very fundamental test cases that you frequently run to make sure at least the very basic functionality of the product remains solid. It does not prove software as release-worthy. But failing a smoke test proves the software is definitely NOT release worthy (as in, 'seeing smoke when you power up an appliance is a sure sign it is busted'). Since you want to do the smoke test frequently, it makes sense to automate it, if possible.

12B. INCREASING TESTABILITY

Testability depends on factors such as controllability, observability, availability (of executables, information), simplicity, stability, separation of components, and availability of oracles. This section gives some tips on increasing testability.

12.12 OPT FOR SIMPLICITY

Simpler designs are often easier to test. That is one more reason to choose simplicity over complexity. Besides, we are more likely to mess up when we try to be clever.

12.13 USE ASSERTIONS, USE EXCEPTIONS

Use exceptions where appropriate. Use assertions liberally. They are not the same (find out the difference), and the two are not interchangeable.

It has been claimed that MSOffice 2007 has about 250,000 assertions (about 1%). Various past studies have shown up to 6% of code as assertions in various software programs. Microsoft found that code with assertions has a lesser defect density (imagine the situation if they did not use assertions!).

12.14 PROVIDE A TESTING API

Testability does not come for free. You may have to write more code to increase testability. Some examples:

- Void methods are hard to test. You can add other methods to check the effect of void methods. For example, if the resetStatus() method is void, you will need a getStatus() method to test the resetStatus() method.
- Sometimes, it is troublesome to verify whether a retuned object is the same as the one we expected. We can override the *toString* method so that objects can be converted to strings that can then be compared easily. Another option is to override the *compare* method or overload the '==' operator.
- A component can collaborate with other components during execution. During testing, we might want to replace those collaborating objects with stubs or mock objects. This is called *dependency injection* [http://tinyurl.com/wikipedia-di]. Consider providing an API for such dependency injection for components you develop.
- To test certain functionalities, you first have to bring the system to a certain state. Providing a method to easily bring the system to a given state helps in such cases.

12.15 USE BUILT-IN SELF-TESTS

For correctness-critical parts, you can develop the same function using multiple algorithms and include them all in the system. During runtime, the system will use an internal voting mechanism to make sure all algorithms give the same result. If results differ, it should fire an assertion.

12.16 DECOUPLE THE UI

User interface testing is harder to automate. Decoupling the UI from the system allows us to test the rest of the system without UI getting in the way. Here are some hints:

- Avoid interacting with the user from a non-UI layer. For example, the business logic layer should not print a message for the user. Instead, the message should be passed to the UI layer to print.
- When passing data from the UI to a lower layer, try to pass primitive or simple objects rather than pass them as complex domain objects.
- If input values are supposed to be validated by the UI, put assertions in the receiving layer to ensure they really are.

12.17 MAINTAIN AUDIT LOG

If your system periodically writes to a log file, this file can be a valuable resource for testing and debugging. Sometimes, you can simply use the log file to verify the system behaviour during a test case.

12C. DEVELOPER TESTING

12.18 DO IT

Make sure you do enough developer testing (unit and integration testing) before you release the code to others. Note that debugging is not developer testing.

12.19 FOLLOW THE IMPLEMENTATION STRATEGY

In the *design by contract* [http://tinyurl.com/wikipedia-dbc] type coding, users of your component are responsible for the validity of the input values passed to your component. Your component does not guarantee anything if input values are wrong. That means you do not have to test the component for invalid input. If the language does not have in-built support for DbC, you can use assertions to enforce validity of input values.

In *defensive* coding [http://tinyurl.com/wikipedia-dp], you do not assume that others will use your code the way it is supposed to be used; you actively prevent others from misusing it. Testing should follow the same philosophy. Test the component to see how it handles invalid inputs.

12D. SYSTEM TESTING

12.20 WHEN SYSTEM TESTING, BE A SYSTEM TESTER

System testing is when you test the system as a whole against the system specification. System testing requires a different mindset than integration testing. That is why system tests are usually done by a separate QA team. Student projects often do not have a QA team. However, try to change your mindset when you transition from integration testing to system testing. Another trick that could help here is to plan system testing so that each team member tests functionalities implemented by someone else in the team.

12.21 MINIMISE RISK

Here are some tactics to mitigate the risk of critical bugs sneaking through:

- Duplicate testing: Let the same functionality be tested by more than one person
- Get test cases reviewed: Test cases can be wrong too. Get your test cases reviewed by another team member.

12.22 GAUGE TEST COVERAGE

After a developer says he is done writing test cases for a component, you can purposely insert subtle bugs into the component (this is called *error seeding*) and rerun the test cases to see how the test cases respond. If no test case

fails, you do not have enough test cases. You can make it a fun game in which you get points for sneaking in a bug that is not caught by the test cases.

12.23 KEEP THE FOREST IN VIEW

While testing needs to be meticulous and thorough, there is no point being fanatical about it. If you get bogged down by trying to achieve high coverage over one part of the system, you might end up not testing some other parts at all.

Test broadly before you go deep. Check all parts of the program quickly before focussing. Start with core/primitive functions. You may have to fix those before you test the rest.

Start with obvious and simple tests. If any such case fails, developers will want to know it sooner rather than later. While you need to check invalid and unexpected input, check valid and expected input first.

12.24 START BY WRITING CONCEPTUAL TEST CASES

Before you implement test cases in code, you may want to define them at a conceptual level. These are easier to cross-check against the specifications. Doing this early will help you in test planning and estimating the testing effort.

Here are some example test cases specified at a conceptual level:

- Test the scroll bar of the browser for long web pages.
- Test the scroll bar for displaying non-HTML objects, such as images.

As you can see from the above examples, these can later go into the documentation of test cases.

12.25 LOOK IN THE RIGHT PLACES

Hunt where the birds are, fish where the fish are, test where bugs are likely to be [adapted from the book *Pragmatic Software Testing*]. Be smart about where you focus your testing energy. Do not test areas randomly. Do not test operations in the order they appear in the API. Examples of places where bugs are likely to be:

- Parts recently modified.
- Where you found the last bug. More bugs found could mean more bugs hidden (bugs like to hang out together).
- More complicated parts of the system (metrics like cyclomatic complexity or function points can help here).
- Parts done by weaker programmers of your team.

12.26 REPRODUCE ERRORS

Before you can report the bug or start debugging yourself, you should be able to reproduce the error. If you are doing *scripted testing* (i.e. test cases are pre-determined), it is easier to reproduce the error. If you are doing *exploratory testing* (i.e. making up test cases as you go), the audit log can tell you what you were doing when the error was noticed. If the system does not produce an audit log (it should), you have to either memorise or record down everything you do. Alternatively, you can keep a screen recorder running to automatically record all steps performed on the UI. Having a screen recording is immensely useful if the bug falls in the dreaded category of 'intermittent faults' (i.e. happens only now and then) as it gives you concrete proof of the bug.

12.27 ISOLATE ERRORS

After you reproduce the error, try to isolate the error by removing steps which are not 'must have' to reproduce the error. For example, it may be an 'edit-save-edit-save' sequence that brought the error to your attention but you may be able to narrow the error as resulting from a 'save-save' sequence. i.e. the 'edit' steps do not have any bearing on the error.

12.28 MAKE BUG REPORTS WORK

If your project team uses a proper issue-tracking system to track bugs, you will have to write bug reports for bugs you find in others' code. Writing proper bug reports helps to get them fixed quicker, gives you a chance to practise the important skill of writing good bug reports, and shows evaluators how professionally you have done your project.

Your bug report should not stop at saying, "hey, I found a bug"; it should help your colleague as much as it can to locate the bug. When a test case fails, investigate further. See if similar failures occur elsewhere in the system. Figure out the minimum required steps that can reproduce the failure. Try to characterise the failure in a more general manner. Make the report self-explanatory to the developer who will eventually handle it or he will pester you for more details after you have forgotten all about it. It is good to give at least one example situation. Take care to word the report so that it will not offend the developer.

Here are some examples:

- "Test case 1103 failed", "Registration feature doesn't work" [not very helpful]
- "System fails when I enter the text '100% coverage expected <line break> terms fixed' into the description field" [better than the above]
- "System fails when I enter symbols such as '%' into any text field in the 'new policy' UI. This does not happen with other UIs" [much more helpful]
- "Validation mechanism is lousy" [likely to offend the developer]

Some useful policies for bug reporting:

- When you create a bug report, always assign it to someone. If you are not sure to whom, just pick the most likely candidate. Reason: if you leave it unassigned, it will not get picked up for fixing.
- A bug should be assigned to a single person. Reason: this makes it clear who is in charge of handling the bug.
- A bug assigned to you should be accepted or reassigned ASAP (say, within 24 hours?). Reason: to avoid someone sitting on a bug for a long period and then throwing it to someone else.
- Only the person who filed the bug report can close it. Reason: this will make sure the issue was actually handled.

12.29 TEST NON-FUNCTIONAL QUALITIES

The system may need to be checked for performance, usability, scalability, installability, uninstallability, portability, etc.

12.30 TEST ON CLEAN MACHINES

It is not enough to test the software by running it from the IDE. You should also test it by running the executable produced.
It is not enough to test the 'debug' version of the executable; you should also test the 'release' version of the software.
It is not enough to test on machines used for developing the software; before your release, you should also test it on a 'clean' machine (i.e. a machine that did not have your software before).

12E. SOME GENERAL TESTING TIPS

12.31 BE SYSTEMATIC

Writing random test cases that just 'feel right' is not good enough if you are serious about the quality of your software.

12.32 ACCEPT YOUR FALLIBILITY

Do not expect a piece of code to be perfect just because you wrote it yourself. There is no such thing as perfect code).

12.33 'TRIVIAL' IS NOT IMMUNE

Very small and simple code segments can introduce errors. Even if that code segment is too trivial to have bugs in it, it can still break some other part of the system. Such bugs are so easy to overlook precisely because you do not expect such code to have bugs.

12.34 DO NOT TEST GENTLY

Being the author of the code, you tend to treat it gingerly and test it only using test cases that you (subconsciously) know to work. Instead, good testing requires you to *try to break the code* by doing all sorts of nasty things to it, not *try to prove that it works*. The latter is impossible in any case.

Program testing can be used to show the presence of bugs,
but never to show their absence! --Edsger Dijkstra.

12.35 WRITE SELF-DOCUMENTING TESTS

It is unlikely that you will write a document that systematically describes every test case you have. But you could still make your test code self-documenting. Add on comments (or some other form of external documentation) for information that is not already apparent from the code.

For example, the following two test cases (written in xUnit fashion) execute the same test, but the second one is more self-documenting than the first because it contains more information about the test case.

Test case 1	assertEquals(p.getTotalSalary("1/1/2007", "1/1/2006"), 0);
Test case 2	print ("testing getTotalSalary(startDate, endDate) of Payroll class"); assertEquals(p.getTotalSalary("1/1/2007", "1/1/2006"), 0, "Case 347 failed: system does not return 0 when end date is earlier than start date");

12.36 ONE INVALID VALUE PER TEST CASE

When you are testing how a system responds to invalid inputs, each test case should have no more than one invalid input. For example, the following test case uses two invalid values at the same time, one for startDate and one for endDate.

```
print ("testing getTotalSalary(startDate, endDate) of Payroll class");
assertEquals(p.getTotalSalary("40/40/1036", "40/40/1037"), 0,
"Case 142: system does not return -1 when the date format is wrong");
```

If we wrote two test cases like this instead, we get to learn about the error handling for startDate as well as endDate.

```
print ("testing getTotalSalary(startDate, endDate) of Payroll class");
assertEquals(p.getTotalSalary("40/40/1036", "1/1/2007"), 0,
"Case 142: system does not return -1 when the startDate format is wrong");
assertEquals(p.getTotalSalary("1/1/2007", "40/40/1036"), 0,
"Case 143: system does not return -1 when the endDate format is wrong");
```

Note that if we want to test the error handling of each component of the date (i.e. day, month, and year), we have to write more test cases (yes, that is right; that is why testing is hard work :-)

We must design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort --Roger Pressman

12.37 MAKE EVERY TEST CASE COUNT

Testing must be both efficient (i.e. do not use more test cases than necessary) and effective (i.e. maximise the number of bugs found).You can use techniques such as *equivalence partitioning* and *boundary value analysis* to increase the effectiveness and effectiveness of testing.

Every test case you write must strive to find something about the system that the rest of the existing test cases do not tell you). That is the 'objective' of the test case. Document it somewhere. This can also deter students trying to boost their LOC count by duplicating existing test cases with minor modifications. For example, if one test case tests a component for a typical input value, there is no point having another test case that tests the same component for another typical input value. That is because objectives for the two cases are the same. Our time is better spent writing a test case with a different objective, for example, a test case that tests for a boundary of the input range.

12.38 HAVE VARIETY IN TEST INPUTS

Consider these two test cases for the getTotalSalary(startDate, endDate). Note how they repeat the same input values in multiple test cases.

id	startDate	endDate	Objective and expected result
345a	1/1/2007	40/40/1000	Tests error handling for endDate, error message expected
345b	40/40/1000	1/1/2007	Tests error handling for startDate, error message expected

Now, note how we can increase the variety of our test cases by not repeating the same input value. This increases the likelihood of discovering something new without increasing the test case count.

id	startDate	endDate	Objective and expected result
345a	1/1/2007	40/40/1000	Tests error handling for endDate, error message expected
345b	-1/0/20000	8/18/2007	Tests error handling for startDate, error message expected

I find that writing unit tests actually increases my programming speed --Martin Fowler, in 'UML Distilled'

12.39 KNOW HOW TO USE 'UNVERIFIED' TEST CASES

A test case has an *expected output*. When writing a test case, the *proper* way is to have the *expected output* calculated manually, or by some means other than using the system itself to do it. But this is hard work. What if you use the system being tested to generate the *expected output*? Those test cases - let us call them *unverified* test cases - are not as useful as *verified* test cases because they pass trivially, as the *expected output* is exactly the same as the *actual output* (duh!). However, they are not entirely useless either. Keep running them after each refactoring you do to the code. If a certain refactoring broke one of those unverified test cases, you know immediately that the behaviour of the system changed when you did not intend it to! That is how you can still make use of unverified test cases to keep the behaviour of the system unchanged. But make sure that you have plenty of verified test cases as well.

12.40 SUBMIT YOUR TESTS

Submit your bank of test cases as part of the deliverable. Add a couple of sample test cases to your report. Be sure to pick those that show your commitment to each type of testing. It is best to showcase those interesting and challenging test cases you managed to conquer, not those obvious and trivial cases. You can use a format similar to the following:

Test Purpose: Explain what you intend to test in this test case.
Required Test Inputs: What input must be fed to this test case.
Expected Test Results: Specify the results expected when you run this test case.
Any Other Requirements: Describe any other requirements for running this test case; for example, to run a test case for a program component in isolation from the rest of the system, you may need to implement stubs/drivers.
Sample Code: Illustrate how you implemented the test case.
Comments: Why do you think this case is noteworthy?

12F. DEBUGGING

12.41 PLAN FOR DEBUGGING

Do not forget to allocate enough time for debugging. It is often the most unpredictable project task.

12.42 BE SYSTEMATIC

When you notice erroneous behaviour in your software, it is OK to venture one or two guesses as to what is causing the problem. If a quick check invalidates your guesses, it is time to start a proper debugging session.

Debugging is not a random series of 'poke here, tweak there' speculative experiments. It is a systematic and disciplined process of starting from the 'effect' (i.e. the erroneous behaviour noticed during testing) and tracking down the 'cause' (i.e. the first erroneous behaviour that set off the chain of events that caused the 'effect') by following possible execution paths that could cause the effect.

Note that the cause and the effect can be separated by location as well as time. For example, the 'cause' could be an error in saving the data in the first shutdown of the software while the 'effect' could be a system freeze during the start up of the 11th run (It could happen. For example, if the software keeps history data for the last 10 runs of the software and tries to overwrite the history data for the first run when starting up for the 11th run).

12.43 USE THE DEBUGGER

Use the debugger of your IDE when trying to figure out where things go wrong. This is a far superior technique than inserting print statements all over the code.

12.44 WRITE THE TEST BEFORE THE FIX

When a system/integration test failure is traced to your component, it usually means one thing: you have not done enough developer testing! There is at least one unit/integration test case you should have written, but did not.

- So, the first thing to do is go ahead and write that test case.
- Now run that test case - it should fail (if it does not, you failed to write the correct test case, again!).
- Now debug the code to find where the bug is.
- Fix the bug. Run the test cases again to make sure all cases pass.
- Why did this bug come about? Think about it - there might be a lesson to be learnt here.
- Note that if you are using a bug-tracking tool (you should), the bug should be tracked using the tool.

12.45 MINIMISE MESS, CLEAN UP AFTER

Debugging should not involve extensive modifications to the code. Using a proper debugging tool can minimise the need for such modifications. In any case, do not change code like a bull in a china shop. If you experiment with your code to find a bug (or, to find the best way to fix a bug), use your SCM tool [see tip 9.4] to rollback those experimental modifications that did not work.

12.46 DO NOT PATCH THE SYMPTOM

When the system is not behaving as expected, do not patch the symptoms. For example, if the output string seems to have an extra space at the end of it, do not simply trim it away; fix the problem that caused the extra space in the first place.

12.47 SUMMON BACKUP

Sometimes, we see only what we want to see. If you cannot figure out the bug after a reasonable effort, try getting someone else to have a look. The reason is, you may be too involved in the code in concern (may be because you wrote it yourself) and be 'blind' to the flaw that someone else with a more detached perspective can detect easily.

12.48 FIX IT

Bugs do not go away by themselves. If you did not fix it, it is still there even if you cannot seem reproduce it. If you cannot fix it, at least report it as a known bug.

12.49 LEAVE SOME ALONE, IF YOU MUST

If there is no time to fix all bugs, choose some (but choose wisely) to be simply reported as 'known bugs'. All non-trivial software has a list of known bugs.

CHAPTER 13: DOCUMENTING

Documentation is the castor oil of programming. Managers think it is good for programmers and programmers hate it!. --Gerald Weinberg

Writing technical documents is an essential part of a software engineer's life. Most of us would rather be coding, but we still have to document things. That is why documentation is often a main deliverable in a project course. As you will soon realise, producing good project documents is hard work (sometimes, harder than coding) and we may not get it right in one shot. That is why an iterative approach to documentation can work better. i.e. show the intermediate versions to the supervisor (if allowed), and get his feedback on your writing style and level of details, etc. and improve subsequent versions of the documents based on this.

Project documents feature heavily in project grading. If reading it turned out to be an unpleasant experience, it is surely going to affect how the evaluator will look at the rest of your project. When the course does not have a system to rigorously evaluate your product (e.g. by way of an extensive test suite), project documents become even more important. All the hard work you put into the product can be undermined by the bad impression given by your project documents.

First, let us remind ourselves of the main objectives of different project documents:

- **Developer documentation**: Explains to fellow developers the design and implementation of the software you produced.
- **User documentation**: Explains to users how to use the software you produced.
- **Project report**: Explains to evaluators what software engineering techniques/tools/lessons you learned and used in the project; shows how well you did and tries to convince the evaluators why you deserve a good grade.

Note that in smaller projects you might be asked to produce a single document that combines all three aspects.

Given next are some tips to help you write better project documents:

13A. GENERAL TIPS ON DOCUMENTING

13.1 GIVE A GOOD FIRST IMPRESSION

Yes, evaluators will read your project documents, but there is no guarantee it will be read beginning-to-end with the same intensity. That is why it should give a good first impression. Pay particular attention to the beginning part of the document. A top-notch beginning will make the evaluator read the rest with a positive frame of mind. If your first chapter is lousy, your chances of a good grade take an immediate nose dive. By the same logic, pay special attention to the beginning part of each chapter as well. Of course a lousy document will not get you a good grade just because you slapped a spruced-up opening chapter on it.

While we are on the topic of making good impressions, note that a well-formatted crisp-looking document always has an edge over one that is not. Your project documents do not require publication quality formatting. But it does not hurt to use a clean tidy layout and easy-to-read font.

13.2 WRITE TO THE TARGET AUDIENCE

This goes without saying for any writing. For example, what is the right level of detail for a developer document? A safe bet is to write as if it is meant for another student who will be joining your team in the near future. Not too detailed, since the new guy will eventually look at the code. Not too high level, as you want the new guy to really understand how your software works.

13.3 READ IT

All too often we write documents but do not read them later. If you cannot force yourself to read it, how can you expect someone else to read it? Read whatever you wrote to see how it appears to your reader. If you let some time lapse before reading it, it will be easier to switch from a writer mindset to a reader mindset.

13.4 DESCRIBE TOP-DOWN

When writing project documents, a top-down breadth-first explanation is easier to understand than a bottom-up one. Let us say you are explaining a system called *SystemFoo* with two sub-systems: front-end and back-end. Start by describing the system at the highest level of abstraction, and progressively drill down to lower level details. Given below is an outline for such a description.

[First, explain what the system is, in a black box fashion (no internal details, only the outside view)]

> *SystemFoo* is a ....

[Next, explain the high-level architecture of *SystemFoo*, referring to its major components only]

> *SystemFoo* consists of two major components: *front-end* and *back-end*
>
> *front-end*'s job is to ... *back-end*'s job is to ...
>
> And this is how *front-end* and *back-end* work together ...

[Now we can drill down to *front-end*'s details]

> *front-end* consists of three major components: *A*, *B*, *C*
>
> *A*'s job is to ... *B*'s job is to... *C*'s job is to...
>
> And this is how the three components work together ...

[At this point you can further drill down to the internal workings of each component. A reader who is not interested in knowing these nitty-gritty details can skip ahead to the section on *back-end*.]

...

[Here, we can drill down to *back*-end's details.]

...

13.5 'ACCURATE AND COMPLETE' IS NOT ENOUGH

If you take the attitude "it is good enough to be accurate and complete", your documentation is not going to be very effective. It is in your best interest to make the document as easy and pleasant to read as possible. Some things that could help here are.

- Use plenty of diagrams: It is not enough to explain something in words; complement it with visual illustrations (e.g. a UML diagram).
- Use plenty of examples: Do not stop at explaining your algorithms in words. Show a running example to illustrate every step of the algorithm, parallel to your explanations.
- Use simple direct explanations: Convoluted explanations and fancy words will annoy (not impress) the evaluator. Avoid long sentences.
- Get rid of statements that do not add value. E.g 'We made sure our system works perfectly' (who didn't?), 'Component X has its own responsibility' (of course it has!).

13.6 EDIT COLLABORATIVELY

Most of your project documents will need input from multiple teammates. You can use tools like GoogleDocs to work on the same document in parallel. Since such tools often track revision history, it will even help the supervisor to know who did which part of the document.

13.7 DO NOT DUPLICATE TEXT CHUNKS

When you have to describe several similar algorithms/designs/APIs, etc., do not simply duplicate large chunks of text. Instead, have the similarity described in one place and emphasise only the differences in other places. It is very annoying to see pages and pages of similar text without any indication as to how they differ from each other.

13.8 MAKE IT AS SHORT AS POSSIBLE

Aim for 'just enough' documentation. If it can be said in 29 pages, do not use 30 pages even if the page limit for the document is 30. Evaluators like it when a document is short. However, do not skip important details.

13.9 MENTION THE EXPECTED, EMPHASISE THE UNIQUE

It is enough to very briefly mention the stuff you are usually expected to do anyway. If you do not mention them at all, there is a small risk of the evaluator assuming that you did not do them. But if you think some aspect your project is special, make sure you emphasise that aspect sufficiently.

13.10 SUPPORT YOUR CLAIMS

If you make a claim, make sure you support it sufficiently or else you will not get credit for it.

For example, simply claiming that: "We made our system faster" will not earn you marks. You should support it by answering the following questions: What improved the speed? Why did you think that change would improve the speed (provide analytical arguments)? Why do you think the change actually made the system faster (provide performance data)?

13.11 MAKE IT COHESIVE AND CONSISTENT

Different parts of a project document may be written by different team members. Still, it is important that the end result is a single cohesive document. If you have not been given a template for your document, creating such a template yourself should increase the cohesiveness of the final document.

However, you still have to ensure consistency across the whole document:

- To begin with, make formatting consistent. This can be done easily by using 'styles' feature of MSWord (other word processors are likely have an equivalent feature).
- It is more difficult, and perhaps more important, to keep other aspects consistent. For example, the use of upper/lower case (it is rather annoying to see ad hoc use of UPPER and lower case for the same word. e.g. BOOL vs bool vs Bool), use of terminology, the level of details, and the style of explanations.
- It helps to appoint one team member to oversee the quality of a project document (i.e. a documentation guru - see here to learn more about gurus). Of course, you should also allocate time for this person to do his job. i.e. to collate various parts done by different team members, weed out inconsistencies, and improve the cohesiveness of the document.

13.12 USE SLIDES TO GUIDE EXPLANATIONS

Describing an algorithm in words is hard work. Here is a small trick that is really useful, especially if the project presentation too includes a description of the same algorithm:

Step one: Instead of writing it as a document, create a PowerPoint presentation to describe the same. For each slide, use the 'notes' feature of PowerPoint to write down what you are going to say during the presentation. Do a mock presentation to your teammates using the slides and the accompanying notes. Ask your teammates to

interrupt you and ask questions when things are not clear. Refine slides/notes so that such questions are no longer needed. Iterate until you arrive at a near perfect presentation.

Step two: Convert the presentation into a document. Slide-notes and slide-text will be the text in your document. You can transfer diagrams in slides by embedding slides directly in a word document. If the presentation was understood well by the audience, the document will be understood well by the reader too. As an additional bonus, now you already have slides for the project presentation and you know how to present them, too!

13.13 DO NOT LEAVE IT TO THE END

If you plan to 'finish the coding first and write the documents afterwards', you are heading for trouble. First, coding rarely finishes on schedule. Second, documentation takes more effort than you think at first (if you want to write a good document, that is). As a result, you are likely to find yourself squeezed between a project running behind schedule and a looming deadline.

13B. WRITING USER DOCUMENTATION

13.14 POLISH IT

Grammar errors and spelling mistakes in user documentation may be interpreted by potential users to mean the product is of equally low quality. Avoid such mistakes as much as possible, although this may be hard for non-native speakers of the language. At least check for things like consistent capitalization, etc. Dialogue boxes, error messages, and other text produced from within the software itself are parts of the user documentation and should be written with due care. In fact, it is a good idea to put all such text in one file for easier proof reading.

13.15 HONOUR THE CONTRACT

User documentation is a contract with the user. Please make sure the software behaves exactly as described in the software. It is very common for the software to evolve without the user guide catching up.

13.16 WRITE TO THE USER

Make the user guide as simple to read/follow as possible. In particular, use users' language, not developer jargon. It is a good idea to have both a 'quick start' guide written in tutorial style and a more detailed user manual written in reference style.

13C. WRITING DEVELOPER DOCUMENTATION

13.17 EXPLAIN THINGS, NOT LIST DIAGRAMS

The developer guide exists to help other developers learn just enough to start working on a particular part of the software. After giving a high-level view of the system, you can systematically drill down to more details of each component, explaining their external behaviour (i.e. the API), the internal structure, and how the internal structure accomplish the required behavior.

Note that unless it is required by the instructor, it is not a good idea to have separate sections for each type of artifact, such as 'use cases', 'sequence diagrams', 'activity diagrams', etc. Such a structure indicates that you do not understand the purpose of documenting and is simply dumping diagrams without justifying why you need them.

13D. WRITING THE PROJECT REPORT

Experience doesn't necessarily teach anything.
--Gerald M. Weinberg, 'Understanding the Professional Programmer'

13.18 IF YOU DID NOT WRITE IT DOWN, IT DID NOT HAPPEN

Many things that happened during the project is not readily visible from the final product. The downside of this is some of those things could earn you extra marks if only you communicated them to the evaluator in the right light. How will the evaluator know that you went through two alternative designs before you settled on the final one, unless you have documented it somewhere? Unfortunately, most of things that happened during the project is forgotten by the time you come to the final evaluation. The remedy is to keep brief notes of all significant things the project goes through. At the end, you can select which ones can earn you more credit and should be explained in more details.

13.19 SHOW HOW YOU LEARNED FROM EXPERIENCE

A good project course is more about 'learning something' than it is about 'producing something'. At the end, you are trying to prove that you did the project well, but also, and more importantly, you can do it even better in the future (that is, you have learned something from this experience that made you a better software engineer). How well you learn from experience is a gauge of your maturity, and deserves a lot of extra credit. Unfortunately, most lessons are learnt and used without even realizing it. Please make an extra effort to record them somewhere and present them to the evaluators in some form.

A project course expects you to learn something at every step of the project. When things go wrong, not only should you correct the mistakes, but you should learn why it went wrong, and should reflect on how to avoid the same mistake in the future. When things go well, you still try to figure out why, and how to repeat the same

success in the future. A good practice is to have a small 'post-mortem' session after each iteration to figure how you can do better in the next iteration.

Providing details such as given below will demonstrate how well you learn from experience:
What was the problem faced? What were the alternative solutions you considered? Which alternative did you choose? Why did you choose it? Which ones did you try and discard? Did your solution work? Looking back, would you have chosen a different alternative? What lesson did you learn from facing the problem? Could you have avoided the problem altogether?

Of course, there is not enough time to make every mistake possible and learn from it. A smarter way to learn is to learn from other people's experience. Learn, and use, famous quotes related to software engineering. A quote contains a gem of wisdom that has stood the test of time. They are easy to remember, and impressive to use. Read books/articles that are based on software engineering *experiences* (not those thick theory books). Most of them contain lessons learned from decades of real software engineering experience. They contain interesting anecdotes, and are often written in an easy-to-read story-book style. While learning from these sources has its own benefits, showing that you did can earn you more marks.

CHAPTER 14: PROJECT PRESENTATION

One: demonstrations always crash. And two: the probability of them crashing goes up exponentially with the number of people watching. --Steve Jobs

The presentation is for explaining your project - both the product and the process - to the evaluators. The presentation complements the project documentation and the product demo (if any). It gives evaluators a chance to clear up doubts by asking questions on the spot, for example.

While most evaluators are supposed to read your project documentation, there is no guarantee that they will read it cover-to-cover. But they are guaranteed to be present at the project presentation and it is up to you to make the best of it. Some evaluators prefer to attend the presentation first and read the report later. In such a case, the presentation creates the first impression of your project in the mind of the evaluator. In some cases, the whole evaluation is based solely on the presentation. Whichever the case in your course, treat the presentation as a key determinant of your grade. Given below are some tips to make the best of your project presentation.

14.1 PRACTICE, PRACTICE, PRACTICE

Project presentations are usually short. The shorter the presentation, the more the amount of practice you need because there is less time for error. Do as many practice runs as it takes to perfect your delivery. When you practise, practise aloud. Sometimes, things you intend to say sound clever in your head, but come out lousy when you actually say the words aloud.

14.2 ARRIVE BEFORE TIME

For some reason, road traffic is especially slow on the presentation day and many students arrive late because they were 'stuck in traffic'. Please take extra pain to arrive at the presentation location with plenty of time to spare. The worst way to start a presentation is to arrive late. Arriving 'just in time' and starting the presentation breathless is not good either.

14.3 END ON TIME

Project presentations are usually scheduled back-to-back, all of which are attended by the same panel of evaluators. Needless to say evaluators get irritated when you run over the time limit. It shows gross negligence on your part; if you did a rehearsal, you should have realised the problem earlier.

If the presentation duration is 30 minutes, target to finish in 25 minutes. The actual run almost always takes longer.

14.4 BE READY FOR QUESTIONS

Be prepared for questions you can anticipate. Even if you have done all the right things in your project, evaluators get suspicious if you cannot explain the rationale behind your decisions. During practice, get your team mates ask questions that are likely to be asked by evaluators and make sure you can handle them fluently.

14.5 FOCUS ON THE IMPORTANT STUFF

Do not waste time on what is obvious. For example, do not waste a slide to explain what is unit testing; evaluators know what it is (you might inadvertently display your own lack of understanding of it! I have seen that happen) Mention them in passing if you must. Instead, focus on what is unique in your project.

14.6 AVOID OVERSELLING

If you think a top-notch sales pitch can cover up a lousy project, you are mistaken. Give a balanced view of what happened. Talk about things that went right, as well as things that went wrong (and why they went wrong, what lessons were learned, etc.). Experienced evaluators are quick to see through the marketing-oriented presentation that oversells the project.

14.7 CONTENT BEFORE SLIDES

After you put your content into PowerPoint slides, it is often hard to move content around. Use a different medium such as a text file to organise the contents. After your content is stable and the flow is clear, you can transfer the text to slides.

14.8 NEVER TALK FASTER TO CATCH UP

If you feel you are running out of time during the presentation, NEVER increase your talking speed to catch up. Doing so is the same as saying "'in case you didn't notice, I screwed up the delivery of my presentation". Instead, keep cool, leave out less important details (you can say things like "This slide is about XYZ, I'm not going to explain this slide in detail. We can come back to it during Q&A time if you need clarifications"), and finish on time.

14.9 BE REFRESHINGLY DIFFERENT

Do not feel compelled to follow exactly the same structure everyone uses. Evaluators will be happy to see a 'refreshingly different' presentation, as long as you do not stray too far from the objectives of the presentation.

Do not read directly off the slide. Evaluators can read, too. Instead use the slide as a backdrop to help you convey the point you are trying to make.

14.10 BANISH MURPHY FROM THE DEMO

Murphy's Law often interferes with product demos. Steps of your product demo should be pre-determined and well rehearsed. If appropriate, part of the demo can be a pre-recorded screen cast of how the software works, or some of the product functionality can be explained using screen shots. You can always justify not doing a live demo as 'in the interest of saving time'. If you plan to do a product demo, rehearse it in the actual environment you would be using; for example, a low-resolution projector can really take the edge off your product demo that looked wonderful in your own computer. On a related note, make sure your slides still look good when projected and videos/links still work in the environment the demo/presentation will be done.

Some students are 'compulsive clickers' who keep clicking things and moving the mouse around in a very irritating fashion while they demo software. Avoid allocating the product demo to a compulsive clicker.

14.11 DRESS SUITABLY

Do not dress shabbily even if there is no dress code. On the other hand, you do not want to be the only person wearing a full suit that day either.

14.12 AVOID IRRITANTS

It is safe to avoid certain behaviour ticks that might irritate the audience. Here are some examples:

- Saying "next slide please" aloud (work out a less intrusive signal to let your teammate know it is time to change the slide. A wireless mouse is another alternative).
- Excessive use of "OK?" "right?", audible sighing, air quotes (yes, especially air quotes).
- Excessive use of animations that take time, yet do not add any value (especially, those used for slide transitions)
- Saying "it's in the report" in a way that gives the impression "didn't you idiots read the report?" or "I don't care enough to explain it; you can read it yourself if you want".
- Phrases such as "stuff like that", "blah blah blah". Hearing a student say "we used design patterns and *stuff like that*" does not exactly impress an evaluator.

14.13 THE AUDIENCE IS NOT YOU

As you prepare the presentation, consider it from the audience' point of view. In our case, the audience is the evaluators. A common mistake is to assume the audience knows something just because you know it. Here is a useful test you can do during preparation. For each slide, verify that every significant term or concept that will be used to explain that slide is guaranteed to be known to the audience beforehand or has already been explained in previous slides. This test is a great way to identify kinks in the presentation flow.

14.14 STOP AT STATIONS

A train is not much use if it speeds through stations without picking up passengers; the same applies to presentations. Having all the right content organised in the best possible order is necessary but not sufficient to the success of your presentation. The truth is not all content is equally important to the audience. Slow down at key points, wait for impact, clarify doubts, and make sure everyone is 'on board' before proceeding.

14.15 MINIMISE TEXT

A presentation should be as graphical (less textual) as possible. Use screen shots, graphics, animations, videos, gestures and verbal explanations in place of lengthy text. Remember, it is a project presentation, not a lecture. Take your cues from Steve Jobs' presentations [http://tinyurl.com/presentlikesteve], not programming 101 lectures.

14.16 SHOW CONFIDENCE

Do not ruin your chances by betraying a lack of confidence in your own product. Avoid phrases such as "hopefully, it will start when I click this button". Rehearsing the demo beforehand will help you act confident while doing it.

14.17 BE CAREFUL WITH 'IMAGINARY' FEATURES

When you explain 'possible' features that are non-existent at the current time, be sure to explain them as things 'you could do in future' rather than 'things you wanted to do but couldn't'. Avoid phrases such as "by right, there should be an XYZ here, and it should work like ABC". Instead, say "we can extend the product by adding an XYZ here to do ABC".

On a related note, do not show what you do not want the evaluators to see. For example, you should not be saying things like "oh, ignore that; It's just a debugging message".

14.18 KNOW THE VALUE OF Q&A

If you run out of time, do not cut into the time allocated for Q&A time. Q&A time is for the evaluators to clarify things. Not giving them the chance to do so will not work in your favour. A question not asked is a question not answered.

Have backup slides to address questions. Evaluators will be impressed to see how you anticipated questions. When questioned, never act defensive. While it is OK to have a designated Q&A person (perhaps, the person who can think on his feet the best), be ready to take questions if that person is not conversant with the area under scrutiny. If you are cornered, do not try to bluff your way out. Being sincere and humble is more important than trying to score off every question. A better way out of a question that you cannot answer is to accept that you did not think of it, thank the evaluator for bringing it up, promise to think about it, and perhaps, use a light joke to break the tension.

14.19 SHOW TEAM SPIRIT

If the course does not require all team members to take part in the presentation, it makes sense to let team members who are good at presentations to do the presentation. However, others should try to show the appropriate level of involvement without getting in the way of the presenters. Sitting in the back of the room checking email while your teammates present is definitely not a good idea.

INDEX OF ALL TIPS