Software Project Howto

Thanks to Nick Parlante for much of this handout

Software Project Planning

1 — Requirements

In the traditional software engineering process, we research what the customers want and what we're going to do. This results in a "Requirements" document that identifies the features and capabilities that we're shooting for. The Requirements document might list core use cases and scenarios, and include 1st-pass screenshots, or those might be merged into the later design steps. For Bunny World, the assignment handout plays the role of the requirements document.

2 — Core Use Cases

Think of the common, mainstream core "use cases" for the project -- problems or situations the user should be able to solve with the project once it is working. Use cases do not need to be obscure or represent weird edge cases. The use cases can represent, in a sense, a by-example vision of what the project does. Use cases provide concrete examples to guide the discussion of the design. The use cases provide nice, concrete examples to guide subsequent discussions.

3 — Basic Design

Make a basic design with the whole team present. Identify the major classes required and capabilities those classes will need. For each class, identify its responsibilities, what it will need to store, what services it will need from other classes, and how, across all the classes, memory will be owned. I like a design in the form of a big diagram showing a typical arrangement of the objects, what they store, and what messages they respond to. For Bunny World (team project, hard deadline) the design should aim for simplicity and modularity, even if the structure is not that impressive or novel. (more material on what's the right amount of design below)

4 — Verify The Design

Verify the class design by thinking through the chronology of messages which will happen for each of the core use cases step by step. Draw a diagram showing a typical arrangement of the main objects for the program. Label what each object stores. Trace through the flow of messages for the core use cases. Verify that it's really going to work at every step. Do not skimp on this step. Really simulate your way through your design to see what it needs to handle. Don't just think about the **result** of a step, think about the **mechanism** (messages, data structure operations) that lead to the result.

Don't design to death. Establish a basically sound design, verify as best you can on paper that it's going to work against the core use cases, and then **get coding**. Some issues will not be revealed until you get into the implementation, and no amount of design is going to change that (IMHO).

5 — Code In Parallel

Divide up and make as much parallel progress as you can on components.

> Assign subproblems to sub-teams of size one or two. Make firm deadline commitments for when tested components will be available for integration. Create Java interfaces to define inter-module message prototypes.
>
> The subteams can spend most of their time designing, implementing, and testing their components independently. They will need to touch base with the other teams periodically to think through inter-component design issues as they come up.

The subteams may want to use throw away test-bed code to exercise their components if the other components do not yet exist.

6 — Initial Integration, New Plan

Try combining subcomponents to run together. Typically, this is more work than you would think. A basically sound design and some parallel testing can prevent pain here. Also, projects that integrate frequently keep the cost of any one integration low.

Issues which were not fully anticipated in the design will be revealed — this is not because you are a bad designer, this is just the way it always is with a complex system. Hopefully the design identified the major issues. It's expected that some design problems will be revealed and need to be resolved in integration.

Bugs in the components will be revealed now that they are running with the other real components. It's better to test components a little independently -- easier to deal with basic problems then than in integration when everyone is there looking at you. Do not do all the integration late in the project. Integration exposes too many bugs to be left until the last minute.

Observe What remains to be done? What component is proving to be the most difficult? Some components have not been written at all. All the components need more features, integration, or debugging. Some components will have turned out to be much more or much less work that you thought. Some components will blow their deadlines. Don't pretend the component made the deadline if it has not been tested. Observe and make a new plan. Some people will have proved they can work independently. Other people will need help. Don't accuse or feel bad. Do deal with the reality. If the data model keeps not working, your team needs to do whatever it takes to **make** it work. Reassign people in a way that will create the most progress.

New Assignments Assign subteams of one or two to work on specific problems — integrating/debugging, adding features, creating new components. Some work will need to be on the integrated version. Other tasks can proceed in parallel e.g. "We need a catalog. Go dig around in the docs to become a JTree expert, report back with a design on monday, implement and test it independently, and we can try to integrate it on Thursday." In your plans, identify what things can be done in parallel (efficient) vs. which tasks will need to be done on the integrated version. For new assignments, make firm, but realistic deadlines. The common error here is to make a deadline which reflects when you **wish** something were going to be done vs. when, realistically, it **can** be done.

Do not gloss over personnel problems -- if one person cannot deliver on something they are working on, be open to shifting responsibilities around. Do not make the project depend on one person completing something when the evidence suggests they are having problems with it -- that would be almost quixotic (it’s a good word to know go ahead..google it). By the same token, don't put energy into blaming people or feeling guilty. Move responsibilities around so everyone has an appropriate, useful area in which to contribute and get back to work.

7 — Final Integration

You can only build things in parallel up to a point. Some features and testing only make sense with all the significant components present.

People can work on specific classes independently, but the real final testing will need to be in the integrated version.

When things get intense, some people like to focus on a hard debugging problem alone. Other people debug best working with others.

For an industrial scale project you would expect to spend a fair amount of time in "beta" -- testing and debugging on the integrated version with all the final components and features present.

You need to stop adding features and switch to verify -that-it-works mode before the last minute. "Stop shaking the Jell-O".

Sometimes, CS108 projects get turned in with pretty obvious bugs in them that would have been easy to fix. Essentially, they needed 4 more hours to smoke out some obvious bugs added in the last day (or rather, they need to switch from "feature" to "bug-hunt" mode 4 hours sooner). This also happens because the stress of the final hours makes people much more error prone than they are normally. Ideally, be done 10 hours ahead of time, and spend the last time checking for bugs, but not messing with the code. Be very careful in that last stretch -- it seems like the bug-per-line rate goes way up because of the deadline stress.

Design/Integrate/Plan Issues

1. Core Use Cases

- Think about the main, obvious use cases for the program -- problems it needs to solve.
- Not weird, hard to imagine cases -- nice obvious cases.
- Use these to drive and test design ideas.

2. Just Enough Design

- Come up with an OOP design of the major classes.
- Identify responsibilities -- what they respond to, what they store
- i.e. services each class provides for the other classes.
- At this stage, you can keep the description simple and human -- "stores the image model objects" not "has an ArrayList of ...". Similarly, identify basically what the objects can do, but don't feel you need to nail down names for all the methods.
- The goal is not to have 100% foresight about future issues -- just get the major issues figured out so that things will fit together without too much grief later.
- Funny quote making fun of people who do zero design: "We don't have time to stop for gas, we're already late" (heard from Mike Cleron)

Nasty Surprise

- A "nasty surprise" is when, upon integration, you discover that the components have some deep design difference that makes them hard to integrate.
- If the methods names change a little, or a little method or ivar must be added to a class to make integration work -- that's fine. Some little issues like that will only become apparent when you are actually building and integrating thing.
- It is impossible to plan so accurately that there are zero integration issues
- It is possible to plan enough to avoid the nasty surprise.

Design Spectrum
Details vs. Prediction

- At one extreme, you can try to predict or forecast all the issues that will arise -- have a complete design before doing any code. (Waterfall method)
- At the other extreme, just jump in and start coding without much of a plan ("rapid prototyping").
- Obvious Middle ground

- Having some sort of plan is good -- allows us to make certain key, smart decisions that improve the whole project and allow parallel development.
- However, our ability to predict the details of how complex systems fit together is very poor.
- Therefore, have a basic structural plan, but do not try to predict every detail.
- Accept that some issues cannot be predicted, they can only be observed when you actually try to build it.
- We need to be flexible to account for issues as they come up.
- Once a basically sound design structure is agreed upon -- get coding!

Do Not Search For the "Perfect" Design

- It is often the case that there are multiple reasonable solutions, all of which will work ok.
- It is a tempting in a team to iterate, trying to isolate the "perfect" solution -- do not do this too much!
- Instead, pick one an acceptable, simple, functional solution and move on.
- The iterative discussion trying to identify the marginal better solution can be a time trap.

3. Object Diagram

- We are accustomed to creating class diagrams that show the classes and their subclassing relationship to each other.
- Another type of diagram will show the **objects** in a typical arrangement -- one canvas, a few shapes, a few shape models, ...
- For each object, identify what it stores and what it responds to.

4. Test Use Cases on Object Diagram

- For each core use case, trace through the sequence of message sends, object changes, etc. that happen to implement that use case.
- Do not skimp on this step!
- "The canvas searches its shape list for the target shape by name. It then sends a `foo()` message to that shape. This causes the shape to send a `bar()` message to each of its ...<snip>.. and so eventually the right Binky object is sent back to the canvas, and we're all done."
- It is so much easier to flesh out the major points of the design now vs. when it is actually in code.

Consistency

- With the whole team looking at the design, you may think you all understand it in the same way.
- However, stepping through the core use cases can reveal details where each team member had their own interpretation. This is the time to make sure that everyone has the same understanding of the overall OOP design.

Implementation Mechanism vs. Results

- The classic error for use case testing, is to just visualize the result -- e.g. "to create a new shape ... well a new shape appears on screen."
- You must think in terms of implementation mechanism: what sequence of messages and object changes causes the result, ultimately, to show up on screen.
- Don't think pixels, think of the code that results in the pixels.

Java Interfaces

- Java interfaces are a handy way to spec out the messages that each class will support -- they form the point of agreement between the class and its clients. The class then implements the interface.
- Interfaces are a more lightweight and compact file to share among the team compared to a whole class file.

5. Parallel Development / Test Bed

- With a basic design structure in hand, you can make progress in parallel -- most productive use of your whole team.
- Course Correction
  - Mostly work independently, but contact module owners to fix up (hopefully minor) issues as they come up.
- Test Bed Code
  - If necessary, can test a module a little using some sort of hacked-together test bed.
  - e.g. Test `JTree` by installing it into `Whiteboard`, feeding it the same 6 strings every time, and using `println` to test what messages it sends.

Local Expert

- Divide up the problem so that one person can deal with, say, the `JTree` class, and nobody else needs to. The OOP design isolate the problem so just one person deals with it.

6. Integration

- Expect some problems when fitting modules together.
- Some issues can only be seen or tested with the real modules executing with each other.
- "Drive to release" idea -- see where the project is by really combining the code components.

7. Make A New Plan

- What is the integrated version telling you about the progress of the project?
- Try to avoid denial of problems / an optimistic bias in observation.
- Some things will be easier than you thought, and things that were not even on radar will pop up to create problems.

New Modules

- As things are finished, can "fork off" new projects for team-members, taking advantage of parallelism.

Problems

- If a module is proving unreliable -- it keeps causing problems in integration, or it just doesn't work, then that module needs help.
- Assign an additional teammate to help with it.
- If it is not working, then the new plan must take action. You cannot just say, "let's keep doing what we're doing, and hopefully it will work next week."

Switching Responsibilities Around

- If a person is not getting their module done, then you need to switch things around -- give them something else to do.
- The team, overall, needs to produce a working solution with the people it has. If a person cannot get a module done, then it is in everyone's interest to switch responsibilities around.
- This is not about fairness, or getting irritated, or feeling guilty. It's about setting up the responsibilities so that everyone on the team is contributing.
- If a person is having problems with their module for a week, don't just hope that next week it will be better. Accept the observed and make a new plan to best use the people you have.
- Like sports, it's not how the sled team was organized or the details of their preparation; it's what they produced on race day.

8. Final Integration

- As the deadline looms, there is a tradeoff between adding new features and testing that things basically work.

- In particular, for every new feature, there is a danger that it broke an existing feature.
- 1. As the deadline approaches, avoid source code changes that affect many parts of the program. Obviously, a modular design makes it easier to makes it easier to make changes in one place without introducing problems.
- 2. Reserve some time at the end, to stop adding features, and really go through and test everything.
- The idea that, at some point, you have to stop adding features and really work on testing, to make sure things still work is known as "Stop shaking the Jell-O."

Misc Design, Modularity, and Simplicity

What is the Right Amount of Design?

- At one extreme, you can try to predict or forecast all the issues that will arise -- have a complete design before doing any code. (Waterfall method)
- At the other extreme, just jump in and start coding without much of a plan ("rapid prototyping").
- Obvious Middle ground (Nick's software philosophy basically)
  - Having some sort of plan is good -- allows us to make certain key, smart decisions that improve the whole project and allow parallel development. Some decisions are hard to change on a project after it has half coded. There's value in making some time to think things through early on.
  - However, our ability to predict the details of how complex systems fit together is poor.
  - Therefore, have a basic structural plan, but do not try to predict every detail.
  - Accept that some issues cannot be predicted, they can only be learned when you actually try to build it.
  - We need to be flexible to account for issues as they come up.
  - Once a basically sound design structure is agreed upon -- get coding! Don't linger in the planning stage endlessly trying to nail down things which are actually unknowable.

Planning vs. Clairvoyance

- Ignorant of the future
  - Obviously, we are not able to predict all the ways that complex systems will act and interact before they are built.
  - You are trying to make a plan about a software system, where its details will not be known for some time.
- 100% Foresight is Naive
  - "Unexpected problems" are not necessarily due to poor design -- The systems involved are sufficiently complex that a strategy that depends on 100% foresight is naive.

A Little Planning

- Make a basic plan of how the software system parts will fit together in the future. There is a plan, but it leaves unpredictable areas of the design sketchy.
- We acknowledge our ignorance of the future, and try to make a reasonable plan working with what we are confident about.
- Build extra time in the schedule for unexpected problems and changes.
- Avoid fragile designs that depend an unknowable future detail.
- Costs: we won't know exactly how long things will take, or how exactly it's going to work, or how far we will be by a particular date. Of course a plan that claims to know those things is probably kidding itself anyway!
- This generic advice can provide only a rough goal -- create a plan with moderate detail, but don't get sucked in too far.
- Clairvoyant Wish
  - Some design methodologies wish to predict enough of the future to plan for all of it. I think that's a bad strategy. I will promote a more flexible but sloppy strategy -- not knowing the future, and trying to deal with things as they occur.

Modularity and Ownership

- Stanford CS puts a lot of stress on modularity, and Bunny World is the sort of project where it can really pay off.
- You want the writers of modules and X and module Y to...
  - Not depend on implementation details of each other
  - Not be burdened with even knowing how the other modules works
  - expose simple, client-oriented interfaces to provide services for other modules
- The modularity allows the project to best make parallel progress
- The modularity makes it likely that changes, feature additions, etc. in one module will not mess up other modules.
- On some projects, modularity also maps to teammate ownership -- one person "owns" the `Foo` module and knows its implementation. Everyone else just uses the `Foo` interface.

Embrace the Ignorance!

- With modularity on a team project .. you can really appreciate not knowing things.
- e.g. you call things like model.write(file) or code.checkScript(). How does it work? Not my problem!

Generality -- Maybe Not

- Programmers and mathematicians always prefer the more general solution.
- For code that's often great
- But at some point, you really are trying to solve the problem at hand, and writing code very specific to the problem at hand is appropriate.
- In this case, we are writing Bunny World code. The classes will have all sorts of Bunny World specific structure, methods, etc. and that's fine. We are not designing classes that will also support some future, non-Bunny use.
- No: you can call this `getNextClause()` method and you pass it this data structure that specifies the lexical structure you want, so it can parse C or Java or anything.
- Yes: you can call this `getNextBunnyClause()`. It knows about Bunny Script syntax and has all sorts of BunnyScript specific messages and logic hard coded in its interface.
- Generality is great in cases here the code will go on to be used in many different contexts, such as the java Collection classes. We stress this sort of generality throughout CS.
- However, in this case, generality is not worth much. It's better to do something simple that solves exactly the Bunny problem at hand.
- Modularity is still a great deal, but generality is probably not.

e.g. Listener Generality

- In CS108, we have explored general solutions, such as using the "listener" paradigm to connect models to multiple views. That's a nice use of generality, since it can be used in almost any model-view situation.
- However, do not feel like you need to produce that level of generality throughout your code. The Swing listener interface needs to be general, since it is in the Swing library and must be flexible enough to work with many types of client code.
- Your code needs to work with .. your code. It's not some library that's going to be used by thousands of other programmers.

Simplicity

- Most projects fail because the pieces don't quite fit together. Have simple pieces where possible.
- Few projects fail because a module is not capable enough or is not fast enough or not elegant enough.
- Avoid doing complex things because they're neat or attractive. Prefer modules that are easy for the other modules to understand.
- The simplest solution that is just sufficient to solve the problem at hand.

- Extreme Programming quote: "Do The Simplest Thing That Could Possibly Work". Avoid complexity unless it is really necessary. Your inner nerd is attracted to complex, sophisticated solutions, so this can require some self-discipline.

Simplicity and Sophistication

- "Simple" has positive associations for most people. What if I used the word "unsophisticated" instead? Simple solutions, which are often a very good idea, may not appear very sophisticated.

Misc. Code Wisdom / People Skills

Ugly Code Tax

- Code tends to get uglier if is built and debugged in a hurry
- Ugly code imposes a tax on all the development that follows -- don't let ugly code stay that way in week one!
- With ugly code, you are a little unsure of editing your own code because it feels a little fragile and capricious.
- With ugly code, there are lines in the code added at one time or another which may or may not be necessary. These create new bugs and obscure the true operation of the code, making future work more difficult.
- Try to maintain a basic level of code cleanliness with each feature addition.
- If the code descends to being really ugly, it's hard to clean it up. It's easier to try to keep it clean in little steps all the time.
- There is a school of thought that if the code every gets very messy, it's almost irreversible.

Furious Activity is not Progress

- It's easy to get in a situation like "I'm lost now, but I'm going to sit here and bang my head against this problem so long and so hard that it's going to work."
- There's an admirable level energy and commitment, but they are aimed in the wrong direction.
- Such activity may be mostly social -- demonstrating your commitment to your teammates -- rather than actually making progress.
- Right:
  - Get out a sheet of paper and a sharp pencil
  - Have a clean, clear design of how things fit together. Take your time. It's not like banging your head against the problem was going to take less time.
  - The code is short, it works, and you know why it works. When it doesn't work, you figure out why it doesn't work and fix it.
  - Get help from a teammate or TA.

Ego vs. Team

- Avoid Ego
  - Don't bring stuff up just to prove you are right or smart. School trains you to exactly that, but in a time-pressure team project, it's just trouble.
- You are not your position
  - do not identify with a design position too strongly. Attacking the position is not attacking you, and neither is it useful for you to defend "your" position ruthlessly.
  - The team needs to pick a reasonably good solution and move on.
- "Consistent" vs. Perfect
  - Often, the team needs to pick some consistent solution approach among the many available. It is not so important that the single best "perfect" solution be found.

- There's no prize for being right.
  - School can get you in the habit that "being right" is the goal, but in reality the only measure of success is producing solutions.
  - Figure the best strategy possible for the hand you have been dealt (planned or unplanned, just or unjust...)
  - Success depends on forging a team which produces solutions which work.

Perfect Decisions -- Alpha Nerd

- Pick a reasonable solution and move on. Don't have a contest to see who's the alpha nerd, and don't spend an unlimited amount of time trying to identify the "perfect" solution. (see "perfect" point above)
- Pick a solution
  - Make sure that its going to work for the core use cases
  - Make sure everyone understands their responsibilities for the chosen solution.

"Rat Hole"

- Sometimes a decision goes back and forth and back and forth and back a forth -- a rat hole. Make some decision and get out.

Meeting Action Items

- Come out of meetings with clear "action items" which associate responsibilities and deadlines with people:
  - Bob, you're going to learn `JTree` inside and out and report what we should do on Wed.
  - Jane, you own the canvas and shape code until Wed and we're hoping you can get rid of the blinking bug and generally make sure the visual code is perfect.
  - Monica, run your data model against a copy of current HI code and figure out why it crashes on picture paste if it kills you.

It's easy to criticize

- It's very easy to enumerate and criticize the flaws of a complex system.
- Positive feedback
  - Complex systems have flaws. Give positive feedback about what works in addition to the negative feedback about what doesn't work.
  - Or from the other side: when you have built something, having someone discuss its flaws is discouraging and enraging, even if they are right.

Critical Path

- A feature is on the critical path : if the feature does not work, nothing works.
- Realize what features are on the critical path.
- Design them the most carefully. Work on them first

Always Be Near Something Shippable

- Avoid an all-or-nothing structure which is worthless if some part doesn't quite work.
- e.g. House Building
  - Suppose you are building a house with the constraint "at some point at random, we will tell you that whatever you have done in one week will be the final product" -- you build the house in smaller stages, each of which is somewhat useful.
  - “Never risk the necessary in search of the superfluous.” -- Len Deighton (Blood Tears and Folly) paraphrasing Pushkin
- e.g. ETOPS rules for aircraft design
  - Aircraft not allowed to be further from an airport than it is known to be able to fly with one engine not running.
  - An aircraft is rated for a number of hours it can fly with one engine not running. This in turn determines the radius of how far the plane is allowed to stray from the closest airport.
  - "Engines Turn Or Passengers Swim"

No "Big Bang" Projects

- At a larger scale, a big project is divided into shorter (3-6 Month) partially functional projects, rather than one "big bang" delivery of the whole thing.
- With a partial project in hand, planning the next step can work more rationally -- accept not mapping out the whole thing ahead of time.
- This is in contrast to trying to plan the whole thing in advance.
- Going in stages (evolutionary) does not reduce cost, but it does reduce the risk of the infamous huge-project-catastrophe.

Blocking

- "Blocking on X" = : I want to work on Y, but Y depends on X and X is not working so I'm not making progress.
- Example X's: A code component is not effectively functional. A sample file is not effectively functional. Some documentation is not available. The compiler is not working
- Passive Mode
  - Classic Error: there is a tendency to just go into "passive" mode when you block on something which is not your responsibility
- Work around
  - Work on something else
  - Finesse the dependency
- Someone who "get things done"
  - On teams, when hiring, etc. a great candidate is often described as "they get things done." I think this roughly means that they don't block -- they find some way to keep making progress. This may be the ultimate life skill.

"*Make Mistakes Quickly*"

- Imperfect Planning
  - Decisions and predictions will be imperfect, development will go into blind alleys, and that's ok.
- Recognize and fix
  - When something is wrong, recognize that fact as soon as possible, and put in the course correction. "*The first step to getting out of a deep hole is to stop digging*."
  - Quote from an army general discussing the inevitable logistical and planning snafu's "*Thank god what we have to fight is another army*."

Drive To Release / Integrate

- Drive to release -- integrate and combine the current code to a runnable version
  - Makes the state of the current development more obvious and concrete
  - Forces latent inter-module problems to the surface
- Investment
  - Driving to release imposes some costs in the short term
  - The more paranoid/uncertain you are about what you are building, the more frequently it should be driven to release -- integrated so its condition is apparent.
- Meetings / Reports
  - Could use memos, discussions, estimates to think about how far along the project is. This doesn't work all that well.
- Double-Clickable Application
  - Instead, spend time on integration periodically to build something that anyone can run and judge for themselves.
  - Avoid some of the problems o f denial.
- Investment Cost
  - The more uncertain things are, the more frequently you need to drive to release to see where you really are (so you can make good decisions)

- Widely accepted
  - Many modern programming disciplines include the notion of regular "drive to release" as part of the overall plan
- No 1 year deadline
  - Some argue: on an industrial scale project, never have a deadline more than 6 months in the future. All long term projects are a series of 6 month drive-to-release cycles at most. (unless you are trying to invent something truly innovative -- innovating and shipping by a deadline are two different types of work.)

The 24 Hour Rush

- In the last 24 hours, there can appear to be tremendous progress. Many things start working in quick succession.
- I'm not sure why this is -- is it that the adrenaline is so thick in the last 24 hours that you just get a lot done. Or is it that your subconscious has allowed just enough time to make the deadline.
- Or more often your subconscious has allowed almost but not quite enough time to make it: "We've been working on this for weeks, and now we are just almost but not quite going to make it. We could really use just an extra four hours."
- The problem is that at the end, there's not quite enough time left for proper integration -- the Jell-O rule.
- Point: Try to have the big kick of productivity a little before the deadline (maybe 12-24 hours before for something the scale of Bunny World). Then everyone can go home and one person can work on testing and bug-fixing.

Stop Shaking the Jell-O

- Adding new features, and testing and bug-fixing are somewhat incompatible.
- Each code change can create bugs in seemingly unrelated parts of the code.
- Be paranoid about changing the code as the deadline approaches.
- At some point, you need to stop adding new features and just test and fix. Testing which happens **before** code change is not complete.
- It's like testing the water supply, and then adding some more stuff to it.
- There needs to be some testing that happens **after** all the code changes.
- Point: stop adding features a while before the deadline and go make sure that all the old features still work.