3350 Notes

Week 1

Agile intro:

Real World Projects:

- Immense in size
- Linux Kernel
    - 24,766,703 LOC
    - >15,000 developers across > 1500 companies
- RDR2:
    - >3000 Contributors
- Google (across all projects):
    - >2 billion lines

Building Software

- Needs to be organized
    - **Planning**
- Needs to be flexible
- Often grows **organically**
    - Start small
    - Build

Software Development lifeCycle

- A: Analysis/Requirements
- B: Design
- C: Implementation
- D: Test/Verification
- E: Maintenance
- F: Decommission

Waterfall Model

- Requirements
- → Design
- → → Implementation
- → → → Testing
- → → → → Ship
- → → → → → Maintenance
- **One Way** (First **all** the requirements, then **all** of the design…)
- What happens if theres a problem in the design?
    - Have to stop the waterfall and redo all of the steps again (Really bad)
- **Rigid**: Sequential application of SDLC
- Each phase is done by a **Different** team.
- Works ok when problems are discovered early
- A **Stable** environment but extremely **rigid**
    - Cant detect technology problems early
    - Difficult to plan and schedule
        - “It would be nice if it also did this..”
    - The customer doesn't see anything until the **end** of the model

Prototyping

- Build a small-scale sample of a system

- May be functional (or not, a mockup)
- One of two *dimensions*
    - Horizontal:
        - Lots of features
        - Not deep
        - Minimally working
    - Vertical:
        - One or two features working nearly completely
    - Can mix the two different methods
- Gives users a chance to see the product
    - Provide criticism of **vision**

Developing Software

- Involves **Exploration**
    - Rarely know **exactly** how / what you're going to build
    - Exploration: **Observation and experience**
    - Remember: Software grows organically, and this isnt a bad thing

Iterations

- Some design, some implementation, and then a review which loops back to the beginning
    - More focused
    - A prototype for each cycle
    - Can **pivot** and change the design or requirements

Non Technical Details

Developing Software

Involves *Exploration*

- Rarely know exactly how/ what you're going to build
- Exploration Observation and Experience
- Remember: Software Grows Organically

Agile Methodology

a. **Individuals and interactions** over processes and tools
b. **Working Software** over comprehensive documentation.
c. **Customer Collaboration** over contract negotiation.
d. **Responding to change** over following a plan.

Our Agile Methodology

- Just enough up-front documentation
- ***Iterative*** (repeated) and ***overlapped*** execution
    - Implementation + Testing **Together**
- Iterations are **time-limited**
    - **Working** product delivered at end of each iteration
- Working with customer *constantly* (... or at least as often as possible)

Non-Technical Planning Artifacts

Three Types of artifacts

a. Initial **vision** for the software
b. Set of **features** required to accomplish the vision
c. Set of **user stories** required to accomplish a feature

What roles exist in larger-scale software? Who creates which artifacts?

Who's involved?

- Development team (developers, project managers, product managers)
- Customers / Users (←**- most important**)
- Others as necessary

Vision Statement

- Describes the **purpose** of the project
- Defines **what,** not "**How**"
- Start with **one** sentence that describes the purpose of the project
- Then, one paragraph to three pages (one page is **ideal**):
    - What is the project going to accomplish?
    - Why is the project valuable? To who?
    - What are the **success** criteria? (measurable and objective)
- A document for **General Audience**
- A document written by **Customers/Users**
- **NO TECHNICAL DETAILS**

Features

- **Users** define main capabilities of the software
    - What does the software need to be able to accomplish the vision?
- Something that can be completely accomplished **within one interaction**.
- Participants:
    - **Customers**: Write feature and assign a **priority**
    - **Developers:** Estimate **cost** (time) of developing feature

What is a Feature?

Each feature contains:

a. Description of a scenario
b. Goal / Description
c. Relative importance of feature (priority)
d. Approximate cost (in **days**) to implement

Usually represent each feature as a **Card**

NO TECHNICAL DETAILS

Required Reading: User Stories: An Agile introduction by Scott Ambler

Initial User Stories (informal)

- User stories are one of the primary development artifacts for Scrum and Extreme Programming teams.
- A user story is a very high-level definition of a requirement, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it
- Good way to think of a use story is that it is a reminder to have a conversation with your customer/stakeholders
- User stories are small
- **Example:**

- Students can purchase monthly parking passes online.
- Parking passes can be paid via credit cards.
- Parking passes can be paid via PayPal.
- Professors can input student marks.
- Students can obtain their current seminar schedule.
- Students can order official transcripts.
- Students can only enroll in seminars for which they have prerequisites.
- Transcripts will be available online via a standard browser.

-

- **Each statement represents a single user story**

- **Important Considerations for writing user stories:**
  - **Stakeholders write user stories:**
    - Stakeholders write the user stories, not the developers. User stories are simple enough that people can learn to write them in a few minutes, so it makes sense that the domain experts (stakeholders) write them.
  - **Use the simplest tool**
    - User stories are often written on index cards. Index cards are very easy to work with and are therefore an *inclusive modeling technique*
  - **Remember non-functional requirements**
    - Stories can be used to describe a wide variety of requirement types.
      - *"Students can purchase parking passes online"* is a usage requirement similar to a use case
      - *"Transcripts will be available online via a standard browser"* is closer to a technical requirement

- **Indicate the estimated size**
    - Include an estimate for the effort to implement the user story
    - One way to estimate is to assign user story points to each card, a relative indication of how long it will take a pair of programmers to implement the story.
    - The team then knows that if it currently takes them on average 3 hours per point, a 5 point story will take around 15 hours to implement
- **Indicate the priority**
    - Requirements, including defects identified as part of your independent parallel testing activities or by your operations and support efforts, are prioritized by your stakeholder and added to the stack in the appropriate place.
    - You can easily maintain a stack of prioritized requirements by moving the cards around in the stack as appropriate.
    - User story cards include an indication of the priority (usually from 1-10 where 1 is the highest priority)
    - Other prioritization approaches are possible
        - Priorities of high/medium/low
        - Some people assign each card its own unique priority order number (e.g. 344, 345)
    - Important to indicate priority somehow incase you drop the deck of cards or if your using more sophisticated electronic tooling.
    - Priorities may change throughout the process. Can be used to motivate the team to move the card to another point in the stack
- **Optionally include a unique identifier**

        - The only reason to do this would be if you need to maintain some sort of traceability between the user story and other artifacts, in particular acceptance tests.

Initial User Stories (Formal)

- Mike Cohn suggests a more formal approach to writing user stories. He suggest the following format
    - **As a (Role) I want (Something) so that (Benefit).**
    - From the previous example for students buying parking passes, it can be rewritten as the following:
        - *As a student I want to purchase a parking pass so that i can drive to school*
    - This approach helps you think about who a certain feature is built for and why.

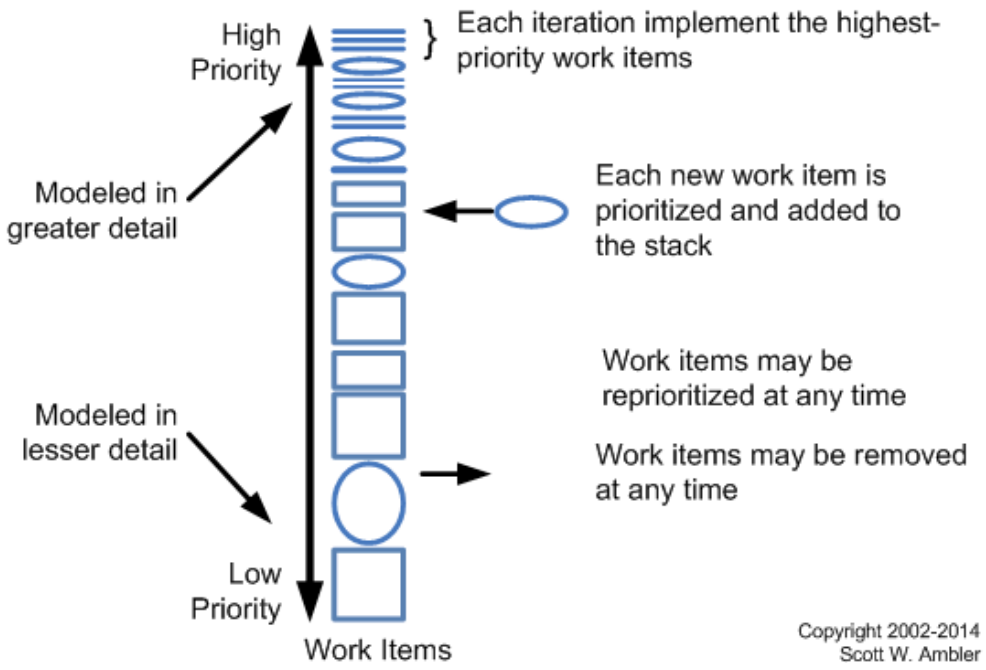
User Stories and planning

There are two areas where user stories affect the planning process on agile initiatives:

1. Scheduling:
    a. The priority assigned to a story affects when the work will be done to implement that requirement. Recall that stakeholders are responsible for prioritizing requirements. Stakeholders also have the right to define new requirements, change their minds about existing requirements, and even reprioritize requirements as they see fit. Stakeholders must be responsible for making decisions and providing information in a timely manner
2. Estimating

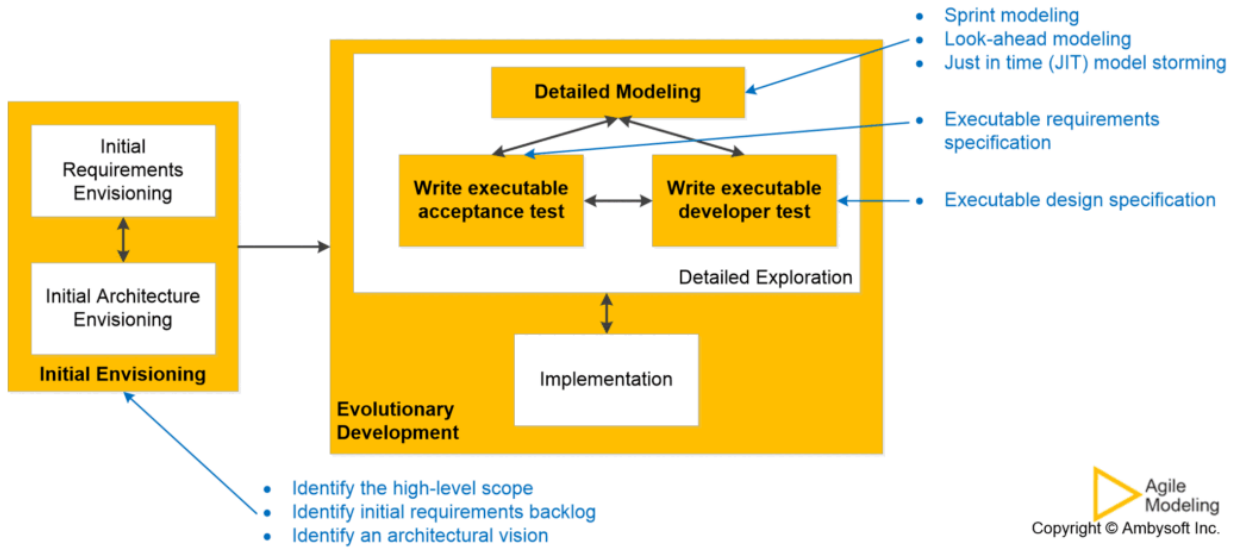
a. Developers are responsible for estimating the effort required to implement the things which they will work on, including stories.
b. The implication is that because you can only do so much work in an iteration, the size of the work items (including stories), affect when those work items will be addressed. Although you may fear that developers don't have the requisite estimating skills, and this is often true, the fact is that it doesn't take long for people to get pretty good at estimating when they know that they will have to live up to those estimates.
c. If you adapted a pair programming practice then a user story must be able to be implemented by two people in a single iteration/sprint. Therefore if you're working in one week iterations each user story must describe less than on week worth of work
    i. Large stories sometimes called **epics** would need to be broken up into smaller stories to meet this criteria.

Disciplined agile change management process depiction:



User Stories Throughout the Agile Lifecycle

AMDD Lifecycle Depiction:



There are three common times when stories will be worked on during an agile initiative:

1. **Inception**
    a. You often create a stack of user stories during Sprint 0, as part of your requirements envisioning activities to identify the scope of your system
2. **Construction**
    a. During construction iterations you will identify new stories, split existing stories when you realize that they're too large to be implemented in single iteration, reprioritize existing stories, or remove stories that are no longer considered to be in scope .
    b. The point is that your stories evolve over time just like other types of requirements models evolve.
    c. Enhancement requests may be identified by your support staff during the production phase and then forwarded to a development team as they are working on an upcoming release.
        i. These enhancement requests are effectively new stories
3. **Transition**
    a. Sometimes new stories will be identified during the Deploy phase, although this isn't very common as the focus of release is on hardening the system and not on new functionality. But it does happen, and these stories would be prioritized and placed on the stack in priority order just as you normally would

Detailing a User Story

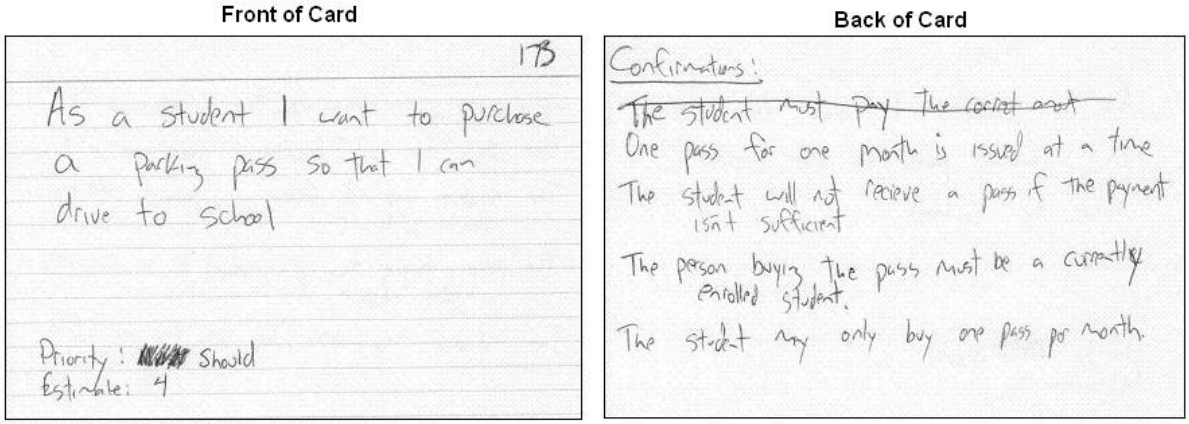
Since user stories contain little information you will need to flesh them out a bit when you first work with them. There are three common times when you would do this:

1. **During JIT analysis/model storming with stakeholders**

a. Remember that the early definition of user stories are that they are a reminder to have a conversation with your stakeholder
b. During this conversation you will explore the details behind that user story
    i. It is common to create screen sketches with stakeholders to explore what they want.
    ii. Also common to identify the ***acceptance criteria*** or confirmations which the stakeholders will use to validate that the user story has been implemented correctly.

2. **During iteration planning**
    a. As part of the estimation effort it is quite common to list programming tasks required to implement the user story
3. **During implementation**
    a. When you start to work on implementing the user story you may decide to create some rough sketches of what you're going to build, perhaps a flow chart or UML activity diagram representing the relevant business logic

User Story card (formal, with confirmations)



Epics

Epics are large user stories, typically ones which are too big to implement in a single iteration and therefore they need to be disaggregated into smaller user stories at some point

- Typically are lower priority user stories because once the epic works its way towards the top of the work item stack it is reorganized into smaller ones.
- Doesn't make sense to disaggregate a low-priority epic because you'd be investing time on something which you may never get to addressing, unless a portion of the epic is high priority and needs to be teased out.
    - Model on a just-in-time (JIT) basis, to increase your overall productivity

Themes

A theme is a collection of related user stories, for example for a university registration system there might be themes around students, course management, transcript generation, grade administration, financial processing.

Themes are often used to organize stories into releases or to organize them so that various subteams can work on them.

Required Reading: Agile Manifesto

Required Reading: Twelve Principles Behind Agile Manifesto

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
2. Welcome changing requirements, even late in development. Agile processes harness change for the customers competitive advantage
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Required Reading: Users Stories by Mike Cohn

Def User Story: a short and simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

Historically user stories were **deliberately kept informal**, written on index cards or sticky notes, stored in a shoe box, and arranged on walls or tables **to facilitate planning and discussion**

- **Their impermanence made it easy to tear them up, throw them away, and replace them** with new stories as more was learned about the product being developed.

Designed to strongly shift the focus from writing about features to discussing them. **These discussions are more important than whatever text is written.**

What is a good user story?

- **Card:** Written description of the story, used for planning and as a reminder
- **Conversation:** Conversations about the story serve to flesh out the details of the story
- **Confirmation:** Tests that convery and document details that can be used to determine when a story is complete

User stories have many advantages, but most important might be that every user story is **a placeholder for a future conversation**

How to write a user story

When writing a user story, remember that user stories follow a standard template:

- As a <Type of user>, I want <Some goal> so that <Some reason>

Examples:

- As a site member, I can fill out an application to become a Certified Scrum Trainer so that I can teach Certified Scrum Master (CSM) and Certified Scrum Product Owner (CSPO) courses and certify others.
- As a trainer, I want my profile to list my upcoming classes and include a link to a detailed page about each so that prospective attendees can find my courses.
- As a site visitor, I can access old news that is no longer on the home page, so I can access things I remember from the past or that others mention to me.
- As a site visitor, I can see a list of all upcoming "Certification Courses" and can page through them if there are a lot, so I can choose the best course for me.

Note that **you don't see any user story as a product owner**. The product owner is an essential stakeholder, but is not the end user/customer. **When creating user stories, its best to be as specific as possible about the type of user**

Does the product owner write user stores?

The product owner's responsibility is to make sure a product backlog of agile user stories exists, but that doesn't mean that the product owner is the one who writes them.

Over the course of a good agile project, you should expect to have some user stories written by each team member.

Note: **Who writes the user story is far less important than who is involved in the discussions of it.**

When are user stories written

Throughout the agile project. Usually a story writing workshop is held near the start of the agile project. Everyone on the team participates with the goal of creating a product backlog that fully describes the functionality to be added over the course of the project or a three-to-six-month release cycle within it.

Some of these agile user stories will undoubtedly be epics. Epics will later be decomposed into smaller stories that fit more readily into a single iteration. New stories can be written and added to the product backlog at any time and by anyone.

Examples of Epics

Large user stories are less detailed and are generally known as **epics**

Example:

- **As a user, I can backup my entire hard drive.**

Because an epic is generally too large for an agile team to complete in one iteration, it is split into multiple smaller user stories before it is worked on. The epic above could be split into dozens (or possibly hundreds), including these two example user stories:

- **As a power user, I can specify files or folders to backup based on file size, date created and date modified**
- **As a user, I can indicate folders not to backup so that my backup drive isn't filled up with things I don't need saved.**

How is detail added to user stories?

Detail can be added in two ways:

- By splitting a user story into multiple, smaller user stories.
- By adding conditions of satisfaction (acceptance criteria).

When a relatively large story is split into multiple, smaller agile user stories, it is natural to assume that detail has been added.

The conditions of satisfaction is simply a high-level acceptance test that will be true after the agile user story is complete.

Consider the following as another agile user story example:

- **As a vice president of marketing, I want to select a holiday season to be used when reviewing the performance of past advertising campaigns so that I can identify profitable ones.**

Detail could be added to that user story example by adding the following conditions of satisfaction:

- **Make sure it works with major retail holidays:** Christmas, Easter, President's Day, Mother's Day, Father's Day, Labor Day, New Year's Day.
- **Support holidays that span two calendar years** (none span three).
- **Holiday seasons can be set from one holiday to the next** (such as thanksgiving to christmas)
- **Holiday seasons can be set to be a number of days prior to the holiday.**

Required Reading: Product Vision by Joel Spolsky

Product Vision Statement example

A sample product vision statement:

“For a mid-sized company’s marketing and sales departments who need basic CRM functionality, the CRM-Innovator is a Web-based service that provides sales tracking, lead generation, and sales representative support features that improve customer relationships at critical touch points. Unlike other services or package software products, our product provides very capable services at a moderate cost.”

Vision Statement Fundamentals

This product vision model helps team members pass the elevator test — the ability to explain the project to someone within two minutes. It comes from Geoffrey Moore’s book Crossing the Chasm. It follows the form:

- **For** (target customer)
- **Who** (Statement of the need or opportunity)
- **The** (product name) **is a** (product category)
- **That** (Key benefit, compelling reason to buy)
- **Unlike** (Primary competitive alternative)
- **Our product** (Statement of primary differentiation)

For any project but particularly those with high uncertainty for which significant requirements changes are anticipated, creating a **product vision statement** *helps* teams **remain focused on the critical aspects of the product**, even when details are changing rapidly. It is very easy to

get focused on the short term issues associated with a 2-4 week development iteration and lose track of the overall product vision.

Design The Box

One practice that I've found effective in getting teams to think about a product vision is the **Design-the-Box exercise**. This exercise is great to open up a session to initiate a project.

In this exercise the entire team, including users, breaks up into groups of four to six (this works best with cross-functional participation).

The team makes the assumption that the product will be sold in a shrink-wrapped box, and their task is to design the product box front and back. This involves coming up with a product name, a graphic, three to four key bullet points on the front to "sell" the product, a detailed feature description on the back, and operating requirements.

Coming up with 15 or 20 product features proves to be easy. It's figuring out which 3 or 4 would cause someone to buy the product that is difficult.

One thing that usually happens is an intense discussion about who the customers really are. Using a very simple product example, it always amazes me that the product vision boxes end up so disparate among three to five teams.

Presentations by each of the groups are then followed by discussing how the multiple focal points can be reduced to a few that everyone agrees upon. A lot of good information gets

generated by this exercise — and it's fun. Once this exercise has been completed, the group can then work on constructing their own version of Moore's product vision statement.

Finally, with several hours of active discussion recorded on flip-chart paper, the group can construct a good outline for a complete one- to three-page product vision document that might include:

- the mission statement,
- a project profile overview (scope, schedule, cost, defects) with priorities,
- pictures of the "boxes,"
- target customers and each of their needs,
- customer satisfaction measures,
- key technology and operational requirements,
- critical product constraints (performance, ease of use, volumes),
- key financial indicators.

For a four- to six-month project, this visioning exercise might take from a half-day to a full day, but it will pay big dividends. The more critical the delivery schedule and the more volatile the project, the more important it is that the team have a good vision of the final desired outcome. Minus this vision, iterative development projects are likely to become oscillating projects — going round and round in circles because everyone is looking at the minutia rather than the big picture.

Week 2

User Stories

User Stories are in the following shape:

As an [actor] I need to be able to do [action]

A user story is an *action* that a *user* must be able to do to use a feature

- Example: “As a **customer**, I need to be able to **view my order history**.”
- Hard Mode: “As a **customer**, I need to be able to **view my order history** because we need to be transparent.”

User stories *resemble* a feature

- But user stories are detailed and **actionable**.

User stories include:

- Time estimates → smaller, more precis (1-3) business days
- Priorities determined by customer
    - **High** = *needed* for this release
    - **Medium** = useful, but not *necessary*
    - **Low** = nice to have

All non-technical planning artifacts are about **what** this thing does, **not** how it will be implemented, or how it will look

What is the point of vision, feature, and user stories?

Breaking down a problem, this is functional decomposition

Release Planning

Delivery

Vision, Feature, and stories share *what* we want to build

- *When* are we going to do it?
- *When* are we going to deliver a *working* product?

Agile → iterative

- “Releases” at specified time intervals

Release planning

Each release is assigned a set of features (by priority and dependencies)

- Release is “**timeboxed**”
    - Time limited
    - Release what is **done** (*not* necessarily what you planned…)
- For us: Each iteration ends in release
    - Larger projects may have several iterations per release.

Release early, and release often

Goals:

- Deliver working software early
- Get feedback
- “Fail fast”

“The Planning Game”

A. Select a user story (or create one)
B. Developers estimate cost (time)
    a. Can assign **risk** to estimates
    b. Split/merge stories
C. Customers prioritize stories, use risk to adjust
D. Repeat until you’ve planned enough for timebox

Where are we?

- Focus on **what** we’re building (*just enough* up-front planning)
- Take a big problem and break it into small pieces
- What about **how** we’re going to build it? Have we put this off too long?

Architecture

Team needs to agree on how to *organize* our code

- Use an *architectural pattern*

Patterns

Software design pattern

- De facto standard, reusable *solution* to a commonly occurring problem.
- There are many architectural patterns

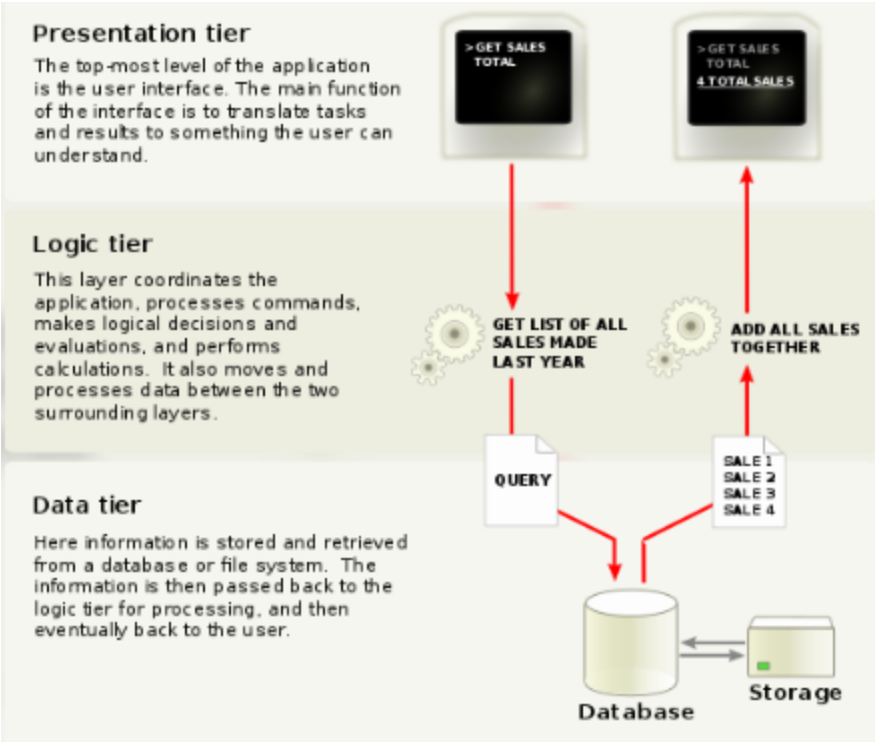
We are going to focus on **one** specific pattern: n-tire architecture

- Separate code into *tiers*/layers
- Each layer has **one** responsibility

Terms:

- Presentation/Interface
- Logic/Business
- Data/Persistence
- Domain-specific Objects

There is an *interface* between the layers



Architectures:

- Data model used across all tiers
- Tiers communicate with method calls
- Use *package* structure to match architecture

Iteration Planning

- We've agreed on what we're going to build
- We've agreed on when we're going to build it
- We've agreed on how we're going to organize it
- **How do we start writing code?**
    - Come up with *technical* developer tasks

Developer Tasks

Are a description of coding tasks required to build user story

- Estimate of time in **hours**

Number of hours <= 1 business day

The Process:

A. Brainstorm
    a. What do I need to do to build this story?
        i. What classes do I need to create?
        ii. What changes do I need to make to the database?
B. Estimate
    a. How long do I think this is going to take?
        i. **Estimate** based on what you know.
        ii. Time in *hours*
        iii. Task is too large (>1 business day)? Split it!
C. Sum
    a. Calculate total cost for **all** time estimates
    b. Possibly *revise* the release plan
        i. Move stories to next iteration
        ii. Split a story that's too complex

The Tools:

- Post it notes
- A wall (or some other flat surface)
- Ordering tasks:
    - Priorities

- Dependencies

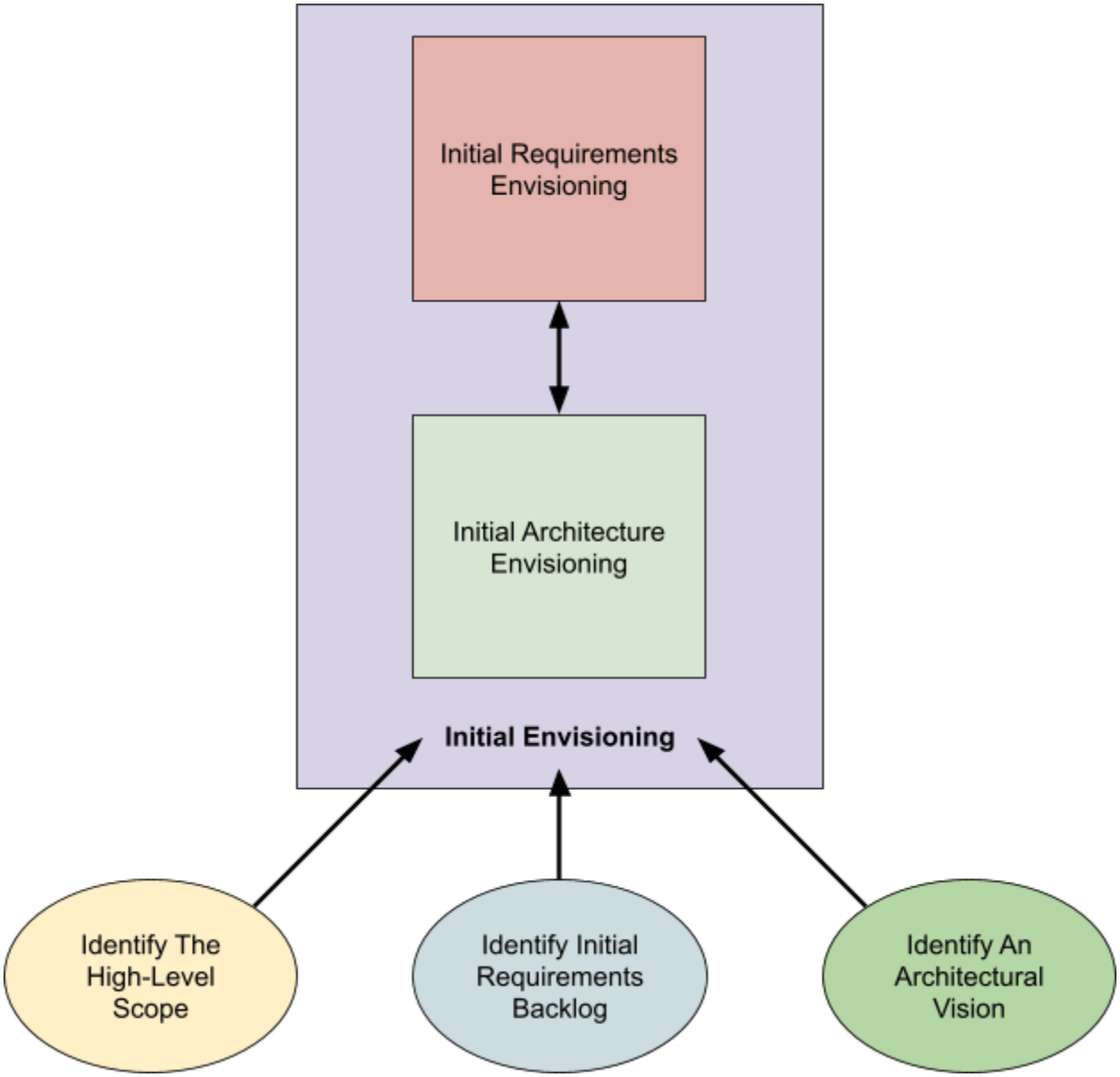
Required Reading Architecture Envisioning by Scott Ambler

Architecture envisioning is the agile practice to perform some high-level architectural modeling early in the life cycle.

- Helps foster a common vision regarding your technical strategy within the team and with critical stakeholders
- Goal at this point is to identify an architectural strategy, not write mounds of documentation.

When Should You Do Initial Agile Architecture Modeling?

Initial architecture modeling is particularly important for scaling agile software development techniques to large, complex, or globally distributed development (GDD) efforts.



Why Should You Do Some Initial Agile Architecture Modeling?

Doing some initial architectural modeling in an agile manner offers several benefits:

1. **Improved Productivity:**
    a. You can think through some of the critical technical issues facing your team and potentially avoid going down fruitless technical paths.

2. **Reduces Technical Risk:**
    a. Your team gains the advantage of having a guiding vision without the disadvantage of having to overbuild your system – just because you've modeled it doesn't mean you have to build it
3. **Reduced Development Time:**
    a. Initial agile architecture modeling enables you to make better cost and time estimates, two pieces of information which management will want.
4. **Improved Communication:**
    a. Having a high-level architecture model helps you to communicate what you think you're going to build and how you think that you'll build it, two more critical pieces of information desired by management.
5. **Scaling Agile Software Development:**
    a. Your initial architecture model will be a key work product in any "agile at scale" efforts because it provides the technical direction required by sub-teams to define and guide their efforts.
6. **Improved Team Organization:**
    a. Effective teams are organized around the architecture or line of business, not around job function.
    b. As you scale to larger and/or distributed teams the sub-teams should each be responsible for one or more subsystems
    c. you don't want to organize your sub-teams around a job function (e.g. an architecture team, a development team, a testing team,...) because that requires you to increase the documentation and bureaucracy overhead which in turn increases risk, cost, and time to value.

What Should You Model Initially?

Early in the initiative you need to have at least a general idea of how you're going to build the system.

- Is it a python application?
- A .Net application?
- Java?
- Something else?

To do this developers on the team will get together in a room, often around a whiteboard, discuss and then sketch out a potential architecture for the system.

This modeling work is based on, and performed in parallel to, your initial high-level requirements modeling efforts.

Your architecture will evolve over time so it does not need to be very detailed yet.

Try and focus on high-level free-form diagrams which overview how we think we'll build the system. General advice for what to focus for are:

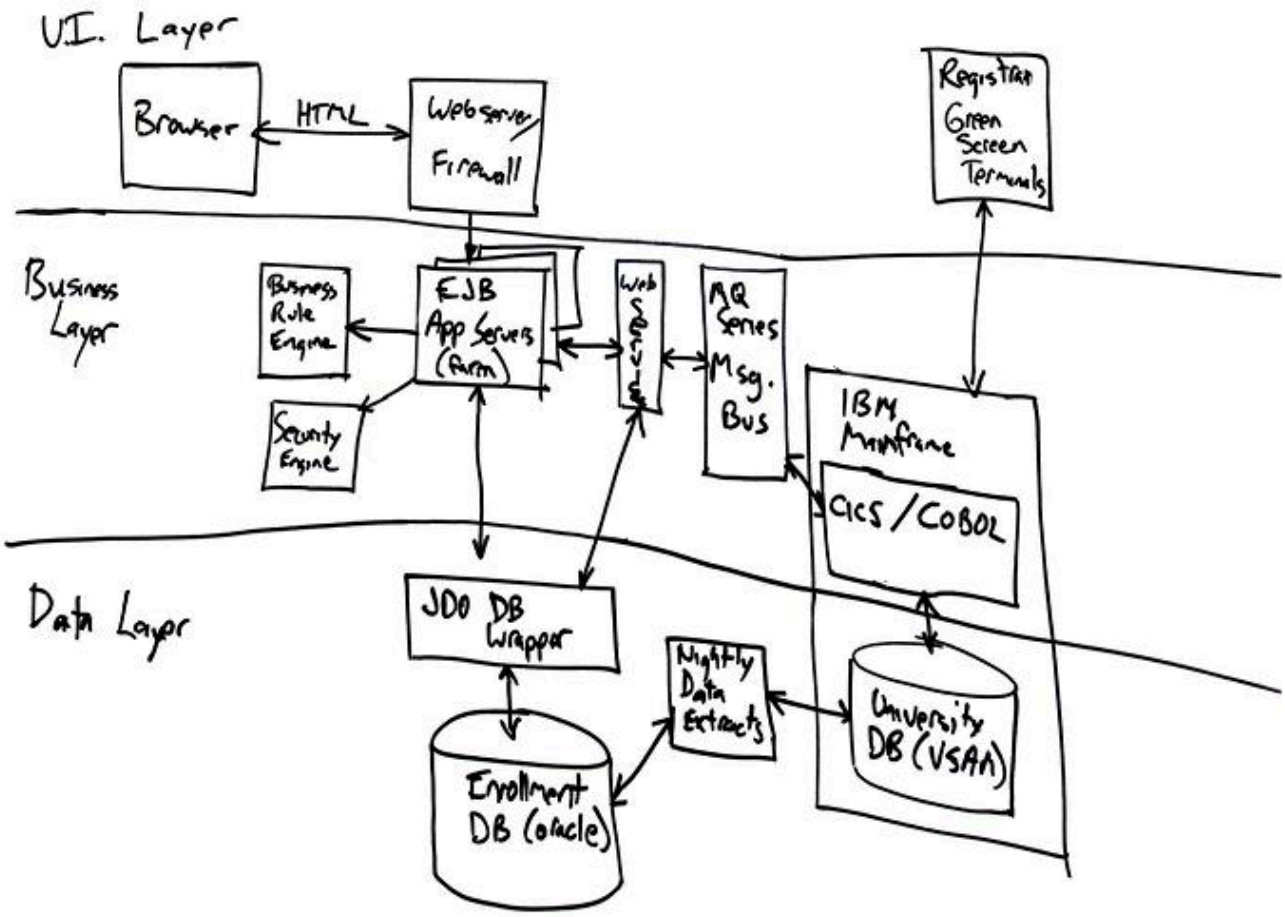
- Technology diagrams
- User interface (UI) flow
- Domain models
- Change cases

Agile Architecture Model: Technology Diagrams

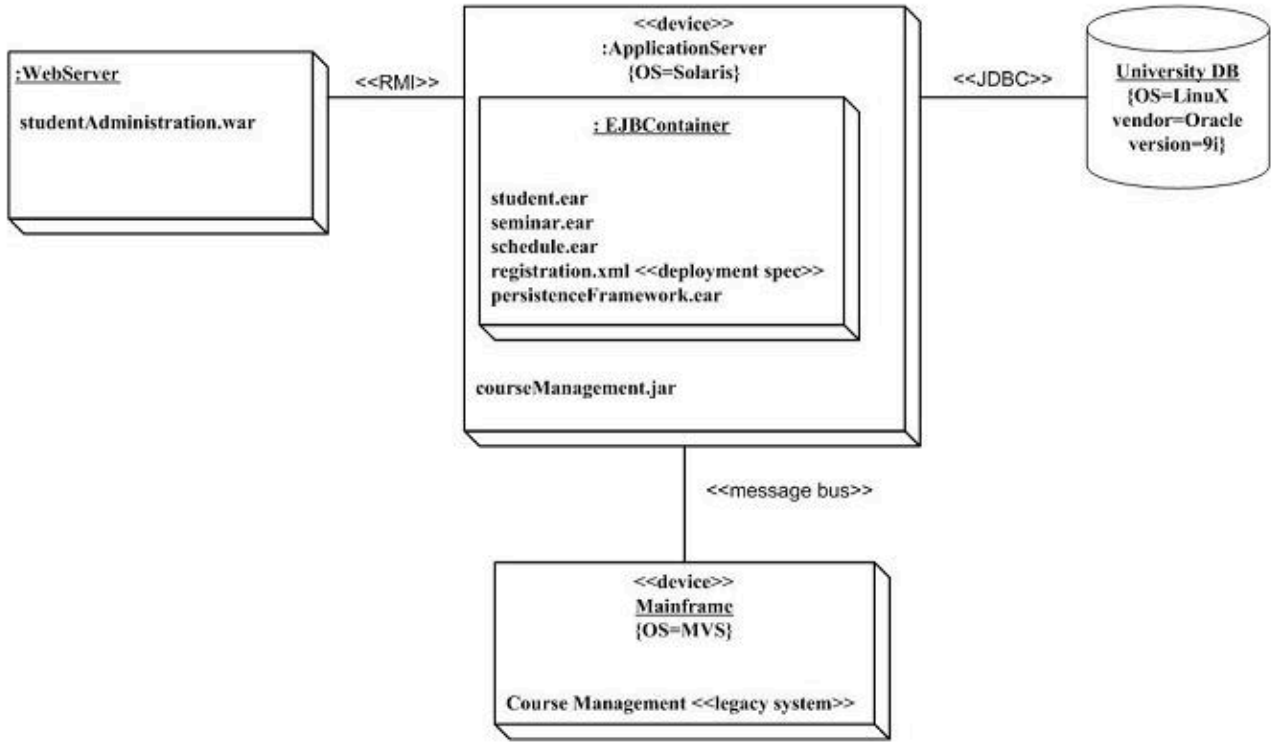
Often some form of technology stack diagram or deployment diagram will do. These diagrams are useful because they depict the major software and hardware components and how they interact at a high level. This includes legacy assets, including legacy databases and legacy systems, which may need to be analyzed in greater detail later in the initiative.

Right now your goal should be to identify these assets and perhaps discuss their viability as sources of functionality and/or data for your system

Example Technology Stack Diagram:



Example UML Deployment Diagram:



You will also be identifying technical constraints at this point in time.

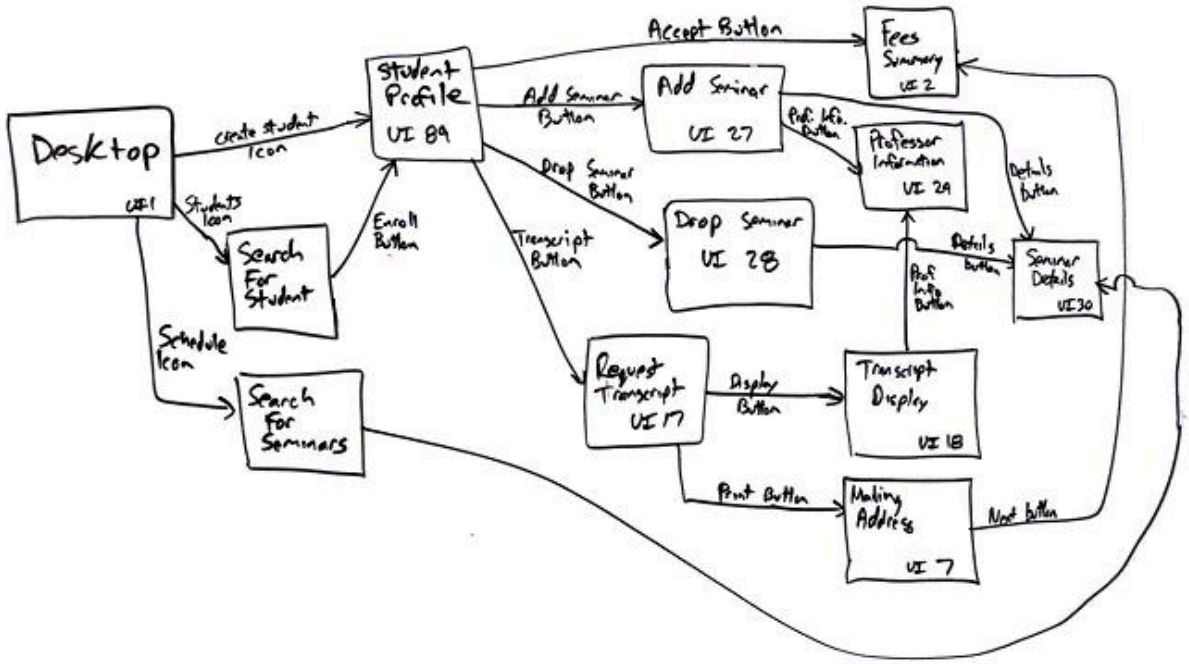
- For example, although you would love to work with the latest version of DB2, unfortunately your corporate database standard is Oracle, and you're therefore constrained in that architectural choice.

Agile Architecture Model: User Interface Flow Models

A common diagram to create is a user interface (UI) navigation or UI-flow diagram, to explore how you will architect the UI of your system by exploring the flow between major UI elements, including both screens/pages and reports.

This is critical to your systems success because the user interface is the system to your stakeholders. Not the technology, not the data, and not really the cool frameworks that you're working with. If you do not architect the user interface effectively you run the risk that you will build a system that your stakeholders aren't interested in working with.

Example UI Flow Diagram:



Agile Architecture Model: Domain Models

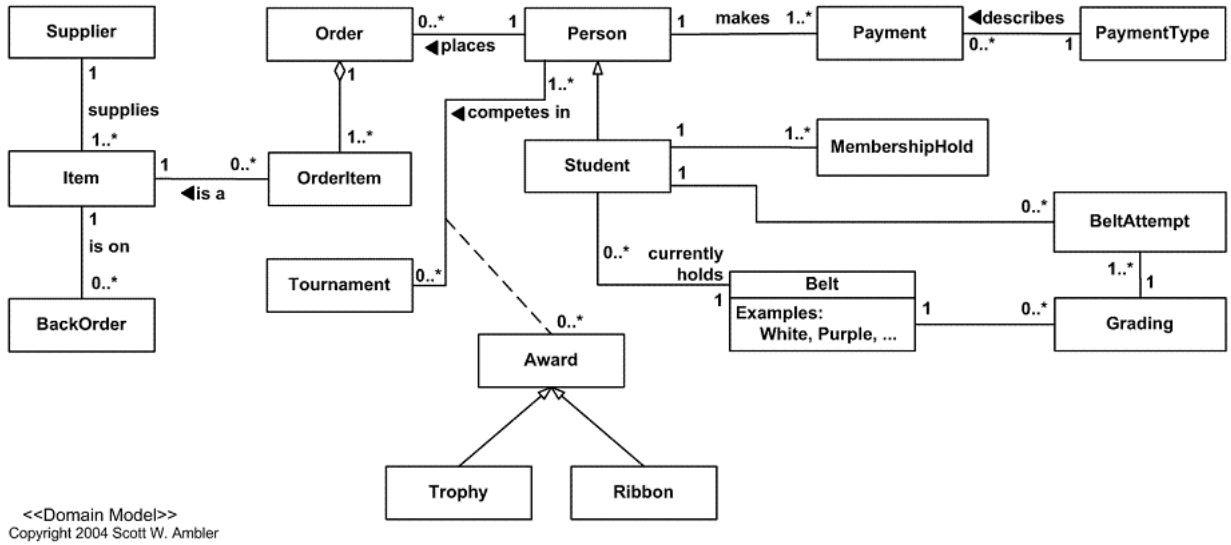
Part of your initial architectural modeling efforts, particularly for a business application, will likely include the development of high-level domain model.

This model should be very slim, capturing the main business entities and the relationships between them. Some people consider this type of model to be an initial requirements model

instead of an initial architecture model — it doesn't really matter because both of these initial modeling efforts are performed in parallel anyway.

The initial domain model will be used to help guide both the physical data model as well as the class design, potentially captured via a UML class diagram.

Example Initial Domain Model:



Agile Architecture Model: Change Cases

Another common architectural model that I sometimes consider capturing are change cases which are potential architecture-level requirements which your system may need to support.

Change cases allow you to test the long-term viability of your architecture without requiring you to overbuild your system because you can think through the impact of likely changes to ensure yourself that your system will still work.

Example Two Change Cases:

Change Case: Registration will occur completely via the internet.

Likelihood: Medium likelihood within two to three years, very likely within ten years.

Impact: Unknown. Although registration will be available online starting in September, we currently expect less than one quarter of registrations to be made via the Internet this year. Response time will be an issue during the peak use periods, which are the two weeks prior to the beginning of classes each term, as well as the first week of classes.

—----------

Change case: The university will open a new campus.

Likelihood: Certain. It has been announced that a new campus will be opened in two years across town.

Impact: Large. Students will be able to register in classes at either campus. Some instructors will teach at both campuses. Some departments, such as Computer Science and Philosophy, are slated to move their entire programs to the new campus. The likelihood is great that most students will want to schedule courses at only one of the two campuses, so we will need to make this easy to support.

What Modeling Tools Should You Use?

Agile modelers will use the simplest tool which will get the job done. Whiteboard diagrams typically suffice for initial architecture models. Your goal is to think through the critical technical issues at this point in the life cycle, not create pretty diagrams. There's nothing wrong with using more sophisticated drawing tools, just ensure that the investment you're putting into the effort is worth it.

It all depends on the situation that you're working in. Furthermore, paper-based tools are also quite common.

How Much Initial Agile Architecture Modeling Should You Do?

Let's consider several common situations that your team might find itself in, and then reflect upon how much architectural modeling would be appropriate in that situation. These scenarios are:

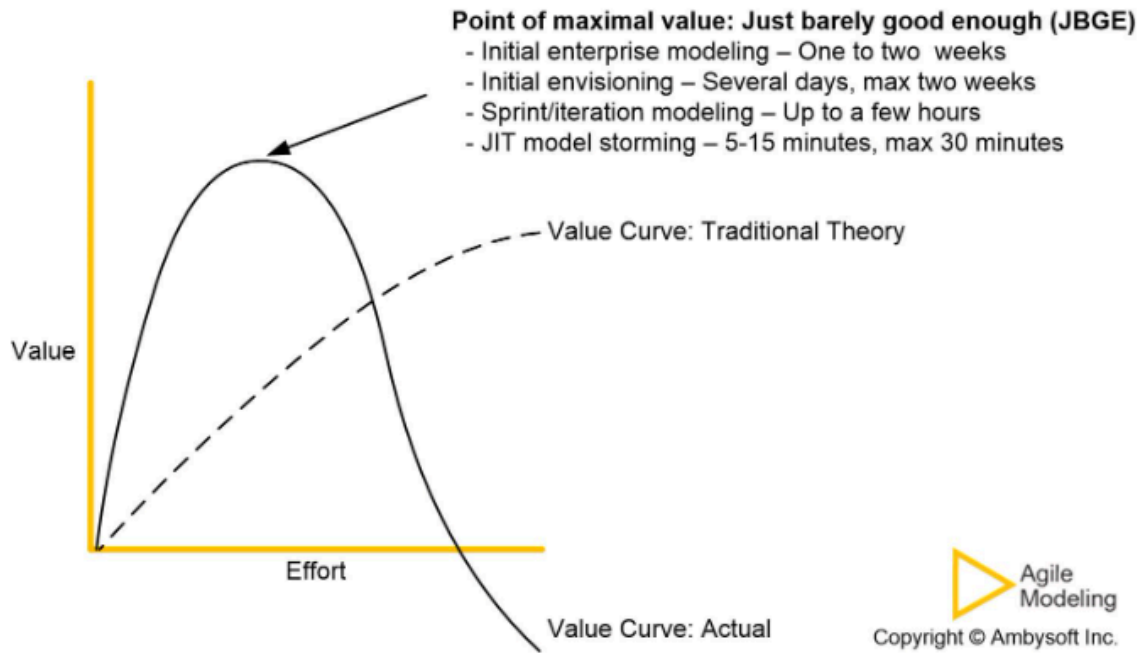
1. **Your team is working with known technologies that it has worked with before.**
    a. This is the simplest situation you can find yourself in and at most I suspect that you need to create a single whiteboard sketch to ensure that everyone is working to the same vision, and frankly you likely don't even need that. For example, if this is the tenth web-based application that your team has built using Websphere and DB2, how much architectural modeling do you really need to do?
2. **You have little or no experience with the technologies.**
    a. Traditionalists will often tell you that to reduce your risk that you should model everything in detail, but if you step back and think about it for a minute, this

actually increases your risk. Does it really make sense to do a lot of detailed modeling when you don't really know what you're doing? No, all that could possibly achieve is that you waste a lot of time, write a lot of documentation which will likely be ignored by the developers because they'll know pretty quickly that you don't know what you're talking about, it give you false confidence that you know what you're doing, and it will likely give management a false sense of confidence that the team has a solid architecture plan in place.

b. What you really want to do is do a little bit of modeling to help you identify what you don't know and, if possible, you want to invite people on to your team who do have the experience that you're missing. Then you need to gain some hands-on experience with the actual technologies themselves via architectural prototyping/spiking, and then based on that experience evolve your architectural models so that they reflect what actually works. In other words, let your architecture emerge.

3. **Somewhere in the middle.**
    a. This is where you typically find yourself: perhaps you have experience with some but not all of the technologies or perhaps your team is experienced with the technologies but simply hasn't used them all together at once.
    b. In middle ground situations such as this it makes sense to do a lot more modeling initially (perhaps even several days) so as to identify potential risks and likely strategies to mitigate them.
    c. Modeling makes sense in this situation because it enables you to define a shared technical vision within the team and to potentially think through major issues. In other words, some initial architectural modeling will offer value to your team.

Value Of Modeling:



Why Do You Need to Do Less Initial Agile Architecture Modeling Than Traditionalists Think?

Traditional modelers assume that it is desirable to model the problem and solution domain in detail at the beginning of an initiative. I refer to this as “big modeling up front (BMUF)“, something which encompasses “big requirements up front (BRUF)” where you create a detailed requirements specification early in the lifecycle and “big design up front (BDUF)” where you create a detailed requirements specification early in the lifecycle. These practices make sense when you’re a modeling specialist, and sure enough traditional modelers always seem to have good reasons for why BMUF is desirable.

Arguments, and Counter Arguments Against, Architectural BMUF:

Argument	**Counter-Argument**
We can think the technical details through up front	This may be true of very simple systems, but for even moderately complex systems we cannot hope to think through the details because of the various nuances of each technology, changing usage patterns of the components, and changing technologies themselves. It may appear that everything in your architectural model is going to work, but until you prove it with code and then act on what you learn you'll never know for sure. The best architectures evolve over time, they're not defined up front.
We need to identify the best technical strategy	Yes, you need to identify a good strategy but that doesn't mean you need to write a lot of detailed documentation. With a BDUF approach you're being asked to make serious technical decisions when you know the least about the problem domain, increasing the chance that you'll choose the wrong path. Lean/agile approaches instead advise you to make these sorts of decisions at the last possible moment when you can safely make them, instead of at the beginning.

We need to build the technical foundation first	You need a technical vision to work towards and good communication/collaboration between the people on the team. A detailed architecture/design motivates you to overbuild your system — it's in the design, so you'd better build it, whether the stakeholders really need it or not. How many times have you seen teams spend the first six months building great frameworks which will support all future needs, only to discover that a lot of the framework isn't needed at all? Unless you get the design perfectly right, which is very difficult to do in practice, you'll end up building far more than you need to.
We need to specify the design for the coders	Coders are really smart people. They may need some guidance and leadership, but they very likely don't need to be told in detail what to do. The fact is that you put a yourself at risk when you present a detailed architecture to coders. Some coders may choose not to follow the design in the belief that they know better (and perhaps they do). Worse yet, the more modeling and documentation that you do, the greater the chance that'll you'll

	commit to the documented approach because of the investment that you've made in the work. This occurs even when the design is known to be questionable because it's often too much effort to rework the design approach once you've started down a specific path.
Architecting software is just like architecting a bridge	The bridge analogy generally goes like this: "Building a bridge is a very complex thing, just like building software. Bridge builders create complex blueprints up front and then work to those plans, therefore we should do the same thing too." If someone doesn't have direct experience building bridges and building software then shouldn't we question their assumptions? I know some physical architects, and they work in a far more creative and evolutionary manner than you might think. To see a world-class architect in action, I highly suggest watching the documentary "The Sketches of Frank Gehry". I can very safely say that Gehry is an agile modeler, not a traditional one. So, perhaps the bridge analogy does hold, but not in the

	way that traditionalists think.
The architecture should be defined in detail just as the requirements are	Have you ever tried to fully specify your design when you are decorating a room in your home? No matter how much effort you put into it, you'll always discover that the couch is at the wrong angle, that the wall art doesn't work with the area rug, and so on. You always have to adjust your design and sometimes even make major changes. If you can't do something as simple as specify up front the decoration of a room, what makes you think you can do the same for a system which is many orders of magnitude more complex? People don't know what they want; at best they may understand their goals/intent. People are very good at saying what they don't like, and providing suggestions for possible improvements. This implies the need to take an evolutionary approach where you obtain feedback on a regular basis, not a serial, documentation-based approach. This assumes of course that you actually want to build a system which reflects the needs of its

	stakeholders. The bottom line is that requirements change throughout an initiative and you need to accept this and act accordingly.
Detailed documentation and models add value	This argument might be partly true, although writing comprehensive documentation early in the lifecycle is the wrong time to do it. Agile Modeling recommends that you write documentation only when the information being captured has stabilized AND if someone understands and is willing to cover the total cost of ownership (TCO) of that documentation. You don't want to document speculative things, and documenting requirements and proposed designs early in the lifecycle are clearly speculative, because speculative things have a tendency to change and thereby increase your TCO for that documentation.
There's no proof that evolutionary techniques work	Actually, yes there is. When people ask for "irrefutable evidence" what they're often saying is that they don't want to move away from their preferred way of working. Traditional approaches were adopted with far

	less evidence than what we have today supporting evolutionary approaches, and to be quite blunt I think that the traditional track record speaks for itself — the evidence, if you’re willing to accept it, clearly points away from BMUF.

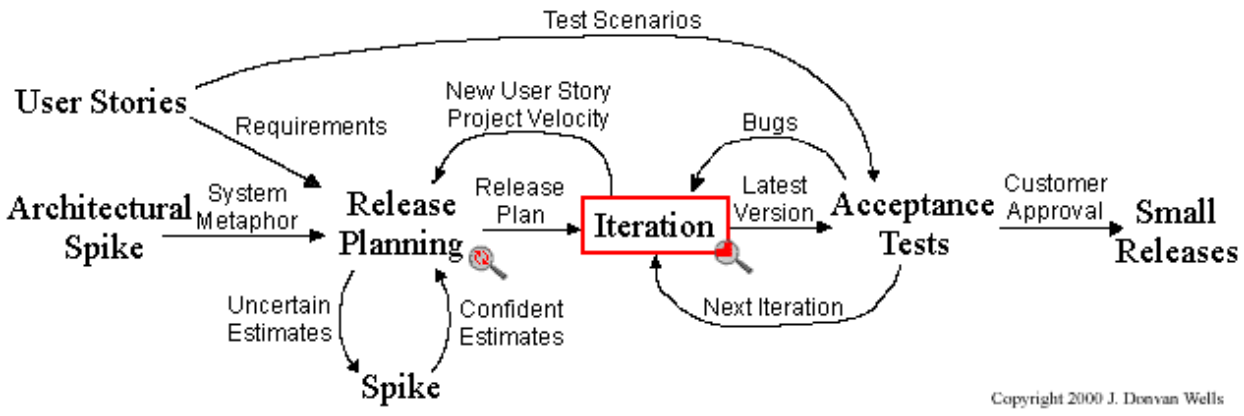
Required Reading Release Plan by Don Wells

After user stories have been written you can use a release planning meeting to create a release plan. The release plan specifies which user stories are going to be implemented for each system release and dates for those releases. This gives a set of user stories for customers to choose from during the iteration planning meeting to be implemented during the next iteration. These selected stories are then translated into individual programming tasks to be implemented during the iteration to complete the stories.

Stories are also translated into acceptance tests during the iteration. These acceptance tests are run during this iteration, and subsequent iterations to verify when the stories are finished correctly and continue to work correctly.

When the project velocity changes dramatically for a couple iterations or in any case after several iterations go ahead and schedule a release planning meeting with your customers and create a new release plan.

The release plan used to be called the commitment schedule. The name was changed to more accurately describe its purpose and be more consistent with the iteration plan.



Required Reading Iteration Planning by Don Wells

An iteration planning meeting is called at the beginning of each iteration to produce that iteration's plan of programming tasks.

Each iteration is 1 to 3 weeks long.

User stories are chosen for this iteration by the customer from the release plan in order of the most valuable to the customer first. **Failed acceptance tests to be fixed are also selected.** The customer selects user stories with estimates that total up to the project velocity from the last iteration.

The user stories and failed tests are broken down into the programming tasks that will support them. Tasks are written down on index cards like user stories. While user stories are in the customer's language, tasks are in the developer's language. Duplicate tasks can be removed. These task cards will be the detailed plan for the iteration.

Developers sign up to do the tasks and then estimate how long their own tasks will take to complete.

It is important for the developer who accepts a task to also be the one who estimates how long it will take to finish. **People are not interchangeable and the person who is going to do the task must estimate how long it will take.**

Each task should be estimated as 1, 2, or 3 (add 1/2 if you need to) ideal programming days in duration.

Ideal programming days are how long it would take you to complete the task if there were no distractions.

Tasks which are shorter than 1 day can be grouped together. **Tasks which are longer than 3 days should be broken down farther.**

Now the project velocity is used again to determine if the iteration is over booked or not. Total up the time estimates in ideal programming days of the tasks, this must not exceed the project velocity from the previous iteration. If the iteration has too much then the customer must choose user stories to be put off until a later iteration (**snow plowing**).

If the iteration has too little then another story can be accepted. The velocity in task days (iteration planning) overrides the velocity in story weeks (release planning) as it is more accurate.

It is often alarming to see user stories being snow plowed. Don't panic. Remember the importance of unit testing and refactoring. A debt in either of these areas will slow you down.

Avoid adding any functionality before it is scheduled. Just add what you need for today. Adding anything extra will slow you down. **Don't be tempted into changing your task and story estimates.**

The planning process relies on the cold reality of consistent estimates, **fudging them to be a little lower creates more problems.**

Keep an eye on your project velocity and snow plowing. You may need to re-estimate all the stories and re-negotiate the release plan every three to five iterations, this is normal. **So long as you always implement the most valuable stories first you will always be doing as much as possible for your customers and management.**

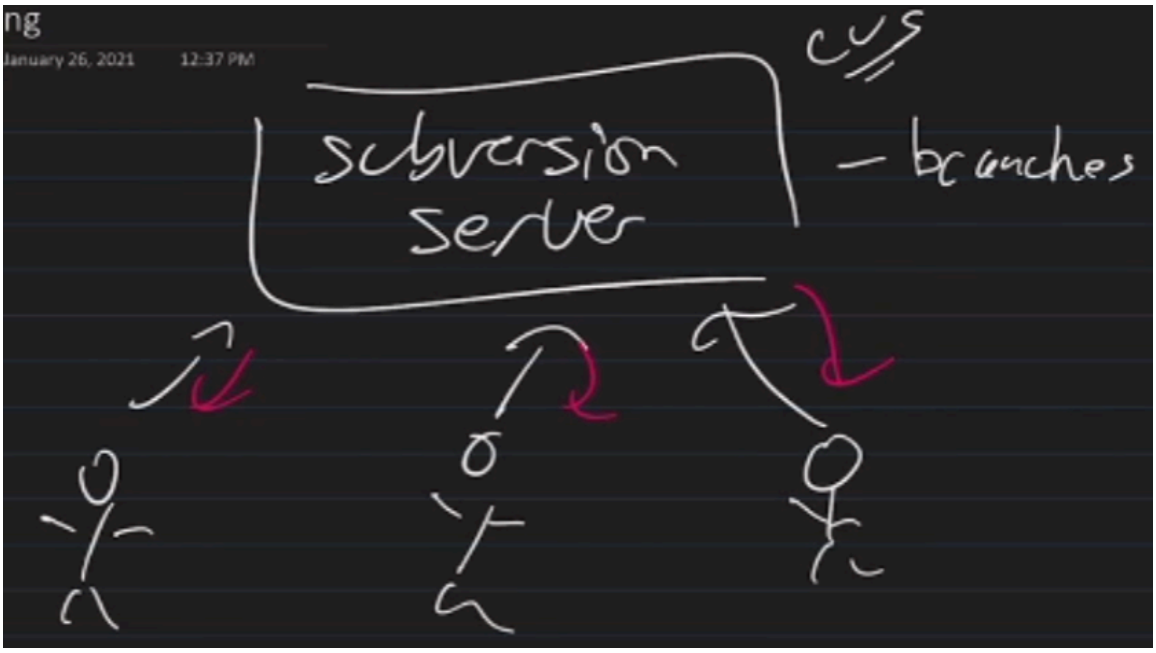
An iterative development style can add agility to your development process. **Try just in time planning by not planning specific programming tasks farther ahead than the current iteration.**

Week 3

Version Control

A. Storage for:
    a. Source Code
    b. Configuration Files
    c. Assets (images, 3d models, etc)
    d. NOT binaries / compiled files
B. Tracks *changes* to files over time
C. Facilitates collaboration

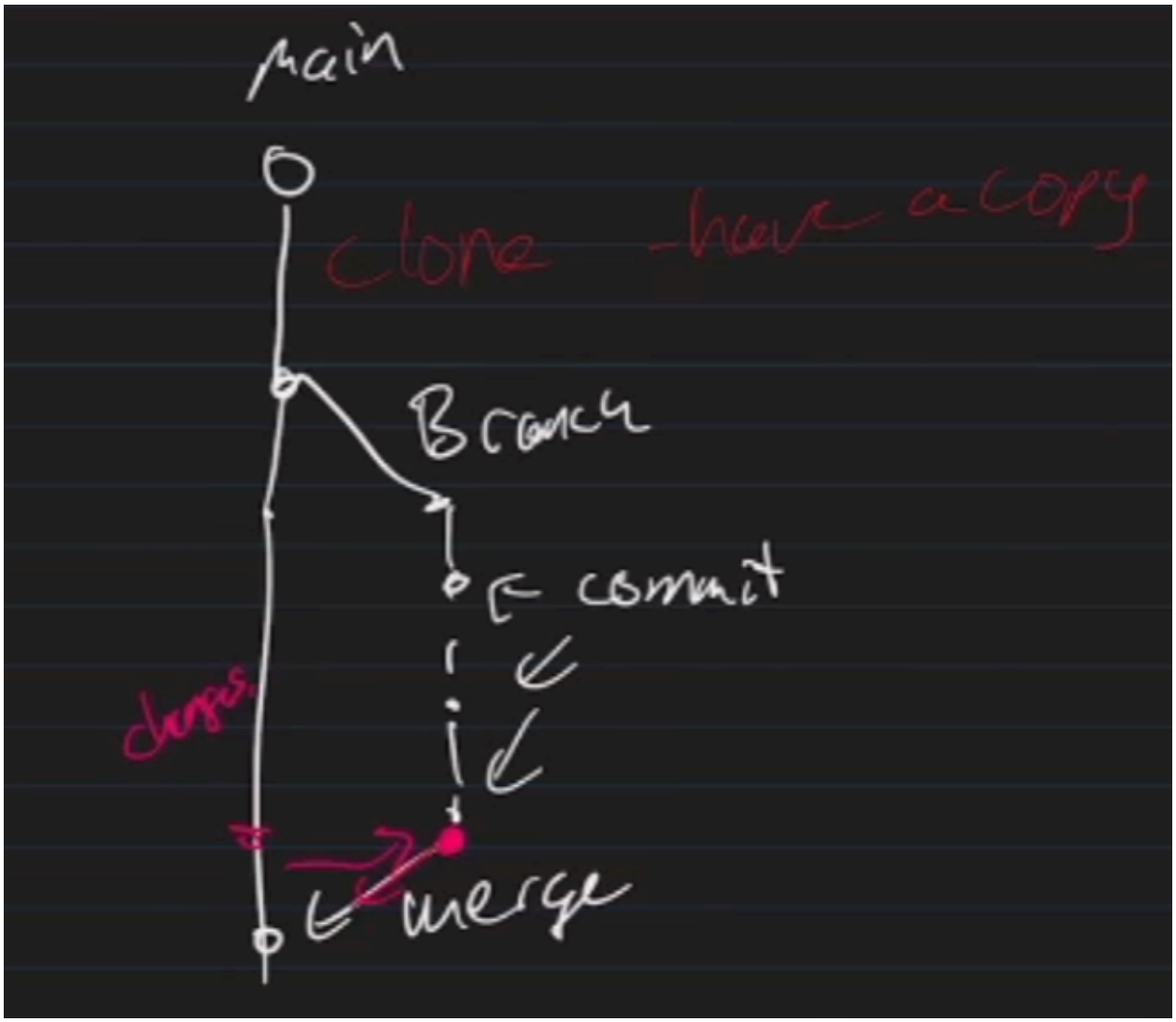
Types of Version Control

- Two major classifications:
    - **Centralized** version control
    - 

        - In the beginning
            - *Naming* files with version
            - Sharing floppy disks
        - Centralized *software*
            - At first, local only (70s, early 80s)
            - Then, client-server model (mid 80s → 2000s)
            - Files (and changes) stored *centrally*
            - Version identifiers managed by software
        - Centralized *problems*
            - Changes are tracked

        - Branching is easy
        - **Merging is hard**
        - **Can't submit work offline**
- **Distributed** version control
    - Concepts and terminology are *mostly* the same
    - *Everybody* gets a copy of the repository
    - Branching and merging are *easier*

Version Control Workflow



Typical:

A. Clone/initialize repository
B. Branch, then build feature/commit
C. Pull/merge/rebase most recent changes
    a. Address conflicts, if any
D. Push
E. Go back to B

Branching Strategies

A. Git flow
B. Github flow
C. Cactus

- No correct workflow
- No best workflow
    - There is a best workflow for your team
    - There is a best workflow for your team at a certain time

Strategy:

A. Choose one
B. Try it (for a while)
C. See what works
D. Keep what's good
E. Throw out whats bad

3350 Project

We will be using Git for our projects. It's a distributed system, it's good, but git is hard.

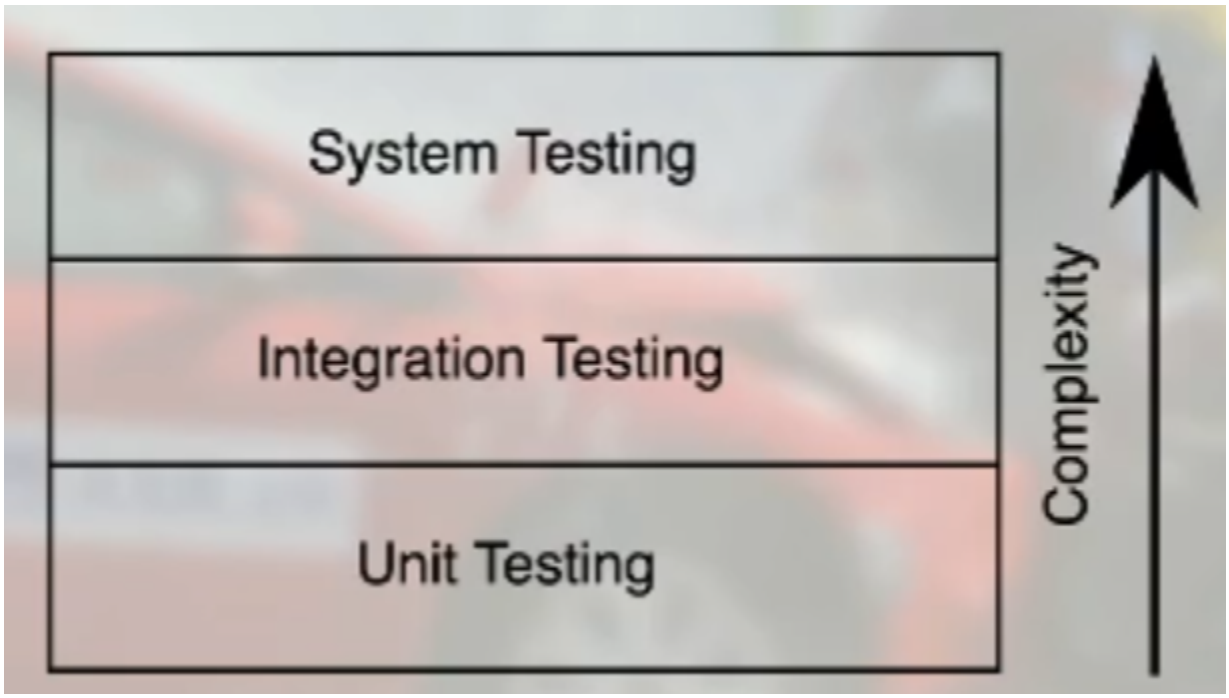
- Common problems and solutions: https://ohshitgit.com/

Testing (high level)

- NOT to find bugs
    - Help prevent introducing bugs with changes!
- Show that your product works
    - Ideally before deployment / delivery
- Goals:
    - Verification: Does it work?
    - Validation: Does it do what it needs to do?

Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead. - Martin Fowler

Testing Levels

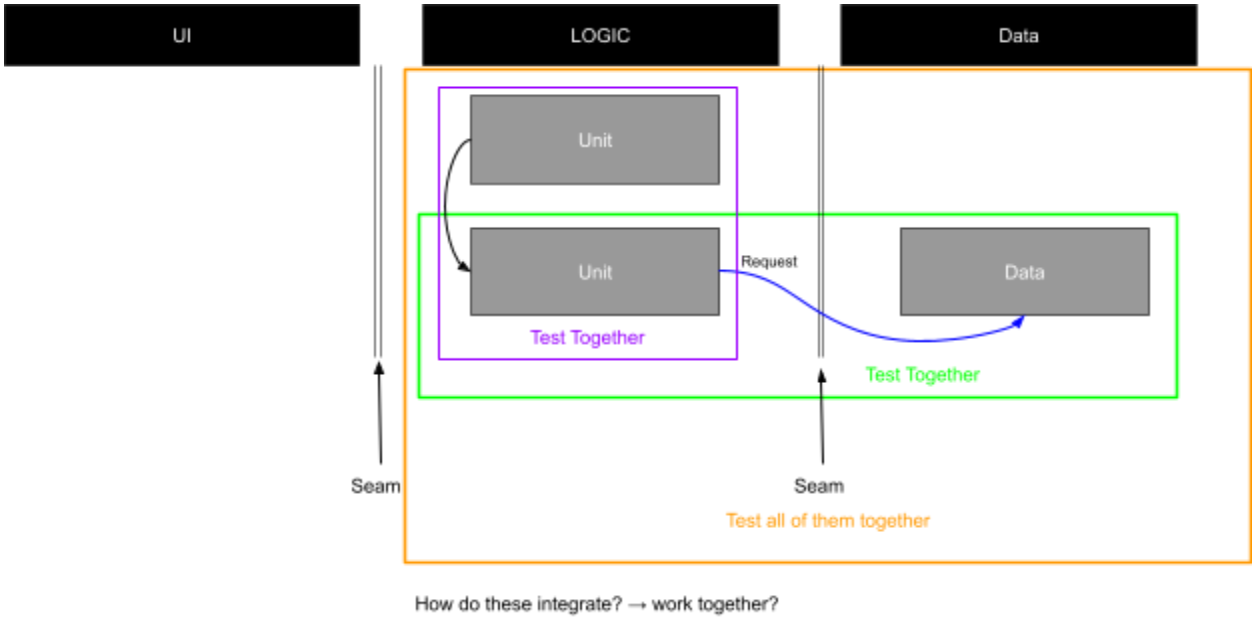


Unit Testing

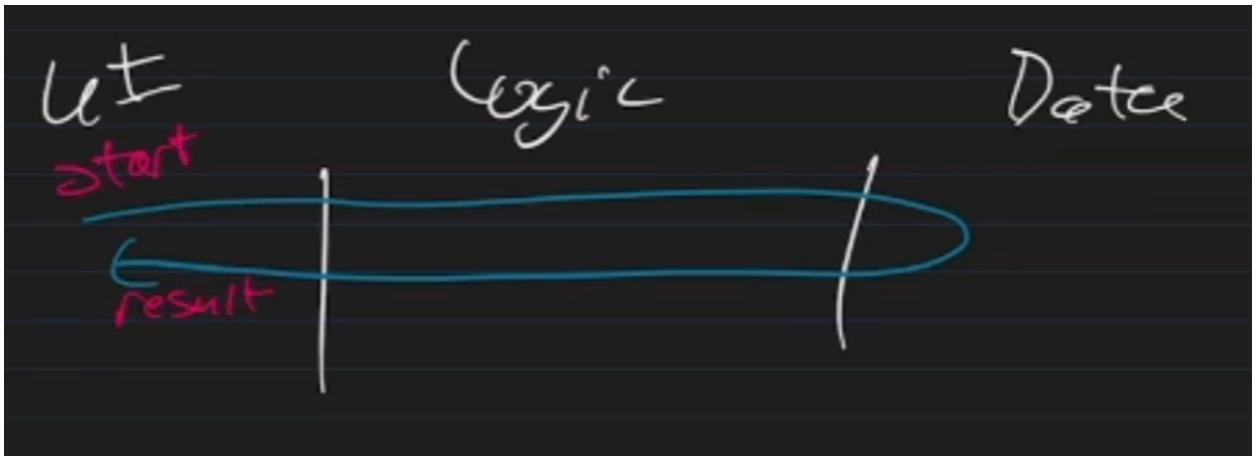
- Does a **unit of code** work by itself?
    - Unit: usually a method (but maybe a whole class)
    - Tested in **total isolation.**

Integration Testing

- Do units of code work together?
    - Testing communication between components
    - Testing **seams** between layers
- Iteration 2 task (more later!)



System Testing



- Does the entire application work?
    - Use real dependencies.
    - In the target environment?
- Iteration 3 task (much more later)

Types of unit testing

Two Strategies:

A. Black-box testing
B. White-box testing

Interface

```java
// MyInterface.java
public interface MyInterface {
    public Widget doThing(final Foo x, final Bar y);
}
```

```c
// myinterface.h
typedef unsigned long widget_id_t;
widget_id_t do_thing(Foo x, Bar y);
```

Declares methods that a class can do

- Specifies the behaviour that a class can do
    - No implementation

Implementation

```java
public class MyClass implements MyInterface {
    public Widget doThing(final Foo x, final Bar y) {
        // actually **do** thing
    }
}
```

- Defines the method implementations
    - Concrete class

Black Box Testing

- Testing just to the interface
    - Testing behaviour
    - No concept of how the code is implemented, just what it should do

Unit Testing

White box testing

- Test to the implementation
    - Testing code paths
        - I.e., cover both branches of an if statement
    - Able to see the how the code is implemented

Tests must be:
- Deterministic
- Isolated

Writing JUnit Tests

A. Static import for assertions:
    a.
```
import static org.junit.Assert.*
```
B. Annotate test methods with @Test
    a.
```
@Test
public void testDoThing() {
    // tests
}
```
C. Test class name should have a suffix Test

Assertions

- Use assert* to verify results:
    -
```
assertEquals( "a should equal b", a, b );
```
- Check out, for more assert methods
    - **http://junit.org/junit4/javadoc/latest/org/junit/Assert.html**
    - Always include a message!

- Helpful for:
    - Self documenting tests
    - Better understanding errors

Before and After

Methods run before/after each tests:

```
@Before
public void setUp() {}
@After
public void tearDown() {}
```

- @Before For creating instance class to test
- @After for releasing any acquired resources
- Supports tests independence
- Be DRY - Dont repeat yourself

Writing tests

- Tests should be minimal
    - One test method should test one thing.
    - One test method should test one piece of data.
    - Each implementation method may have many test methods.
- Complicated test imply complicated code
    - Complicated code should be refactored and simplified

Verifying Failure

- We've got a hierarchy of granular exceptions
    - Represent different types of invalid data

Expected exceptions

- Test invalid data by verifying an exception is thrown

```
@Test(expected=ExpiredCardException.class)
public void testExpiredCard() {
    // card num, expiry date, cvc
    cardProcessor.processCard("123456789", "01/01", "888");
}
```

-
- Test passes only if the exception is thrown
    - Test fails if the exception is not thrown.

- Use the most specific exception type

Optional Reading: A visual Guide to Version Control

Version control/ Revision Control/ Source Control: Lets you track your file over time

Large, fast-changing projects with many authors need a Version Control System to track changes and avoid general chaos. A good VCS does the following:

- **Backup and Restore:** Files are saved as they are edited, and you can jump to any moment in time.
- **Synchronization:** Lets people share files and stay up-to-date with the latest version.
- **Short-term Undo:** Monkeying with a file and messed it up? Throw away your changes and go back to the “last known good” version in the database.
- **Long-term Undo:** Sometimes we mess up badly. Suppose you made a change a year ago, and it had a bug. Jump back to the old version, and see what change was made that day.
- **Track Changes:** As files are updated you can leave messages explaining why the change happened (stored in the VCS, not the file). This makes it easy to see how a file is evolving over time, and why.
- **Track Ownership:** A VCS tags every change with the name of the person who made it. Helpful for giving credit.
- **Sandboxing:** making a big change? You can make temporary changes in an isolated area, test and work out the kinks before “checking in” your changes.
- **Branching and Merging:** A larger sandbox. You can Branch a copy of your code into a separate area and modify it in isolation. Later you can merge your work back into the common area.

Lingo

(LINGO) Basic Setup:

- **Repository (repo):** the database storing the files.
- **Server:** The computer storing the repo.
- **Client:** The computer connecting to the repo
- **Working Set/Working Copy:** Your local directory of files, where you make changes.
- **Trunk/Main:** The primary location for code in the repo. Think of code as a family tree – the trunk is the main line.

(LINGO) Basic Actions:

- **Add:** Put a file into the repo for the first time, i.e. begin tracking it with version control

- **Revision:** What version a file is on (v1, v2, v3, etc)
- **Head:** The latest revision in the repo.
- **Check out:** Download a file from a repo
- **Check in:** Upload a file to the repository (if it has changed). The file gets a new revision number, and people can “check out” the latest one.
- **Checkin Message:** A short message describing what was changed.
- **Changelog/History:** A list of changes made to a file since it was created.
- **Update/Sync:** Synchronize your files with the latest from the repository. This lets you grab the latest revisions of all files.
- **Revert:** Throw away your local changes and reload the latest version from the repository

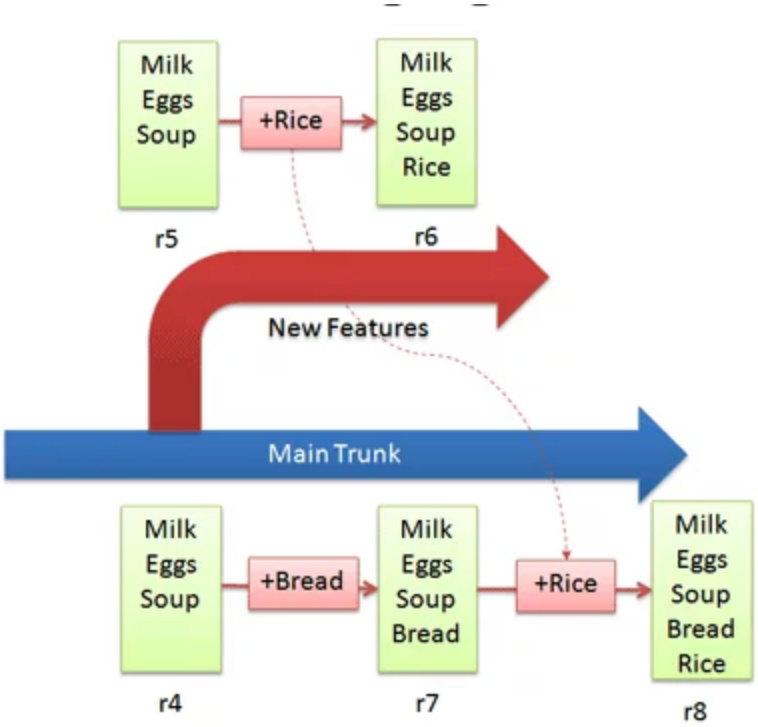
(LINGO) Advanced Actions

- **Branch:** Create a separate copy of a file/folder for private use (bug fixing, testing, etc). Branch is both a verb (“branch the code”) and a noun (“which branch is it in?”)
- **Diff/Change/Delta:** Finding the differences between two files. Useful for seeing what changed between revisions.
- **Merge (or patch):** Apply the changes from one file to another, to bring it up-to-date. For example you can merge features from one branch into another.
- **Conflict:** When pending changes to a file contradict each other (both changes cannot be applied)
- **Resolve:** Fixing the changes that contradict each other and checking in the correct version.
- **Locking:** Taking control of a file so nobody else can edit it until you unlock it. Some version control systems use this to avoid conflicts.
- **Breaking the lock:** Forcibly unlocking a file so you can edit it. It may be needed if someone locks a file and goes on vacation
- **Check out for edit:** Checking out an “editable” version of a file. Some VCSes have editable files by default, others require an explicit command.

Merging Example:

Lets say we want to get the “Rice" feature from our experimental branch into the mainline. How would we do this? Diff r6 and r7 and apply that to the main line?

We only want to apply the **changes that happened in the branch**. That means we diff r5 and r6, and apply that to the main trunk:



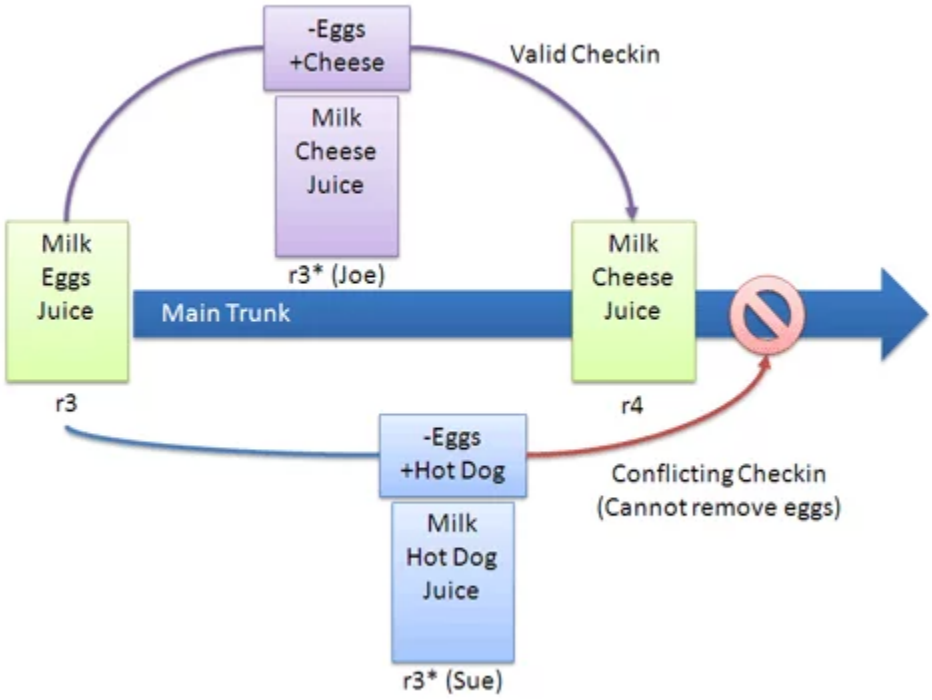
If we diffed r6 and r7, we would lose the “bread” feature that was in main. This is a subtle point – imagine “peeling off” the changes from the experimental branch (+Rice) and adding that to main. Main may have had other changes, which is ok – we just want to insert the Rice feature.

In subversion, merging is very close to diffing. Inside the main trunk, run the command:

```
svn merge -r5:6 http://path/to/branch
```

This command diffs r5-r6 in the experimental branch and applies it to the current location. Unfortunately, subversion doesn't have an easy way to keep track of what merges have been applied, so if you’re not careful you may apply the same changes twice.

Conflicts Example:



At this point it's a race: if Joe checks in first, that's the change that goes through (and Sue can't make her change).

When changes overlap and contradict like this, the VCS may report a **conflict** and not let you check in – it's up to you to check in a newer version that **resolves** this dilemma. A few approaches:

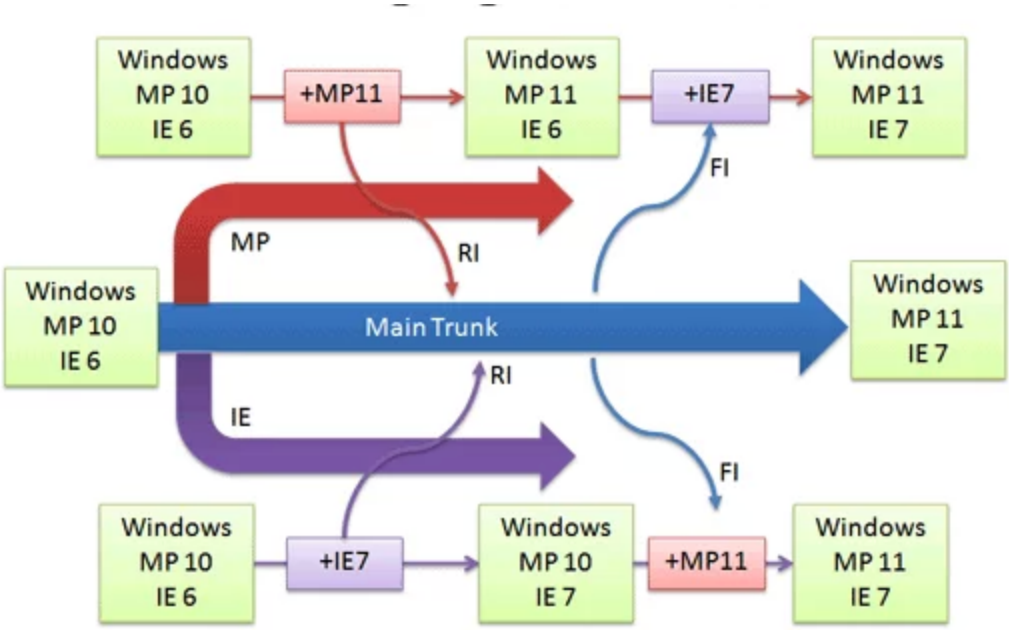
- **Re-apply your changes:** Sync to the latest version (r4) and re-apply your changes to this file: Add hot dog to the list that already has cheese.
- **Override their changes with yours:** Check out the latest version (r4), copy over your version, and check your version in. In effect, this removes cheese and replaces it with hot dog.

Real Life Example: Managing Windows Source Code

We guessed that Windows was managed out of a shared folder, but its not the case. So how is it done?

- There's a **Main Line** with stable builds of windows
- Each group (Networking, User Interface, Media Player, etc). **Has its own branch** to develop new features. These are under development and less stable than main.

You develop new features in your branch and “Reverse Integrate (RI)” to get them into main. Later, you “Forward Integrate” to bring the latest changes from Main into your branch:



Lets say we're at Media Player 10 and IE 6. The Media Player team makes version 11 in their own branch. When it's ready and tested, there's a patch from 10-11 which is applied to Main (just like the "Rice" example, but a tad more complicated). This is a **reverse integration**, from the branch to the trunk. The IE team can do the same thing.

Later the, Media Player team can pick up the latest code from other teams, like IE. In this case, Media Player **forward integrates** and gets the latest patches from main into their branch.

Optional Reading: Intro to Distributed Version Control

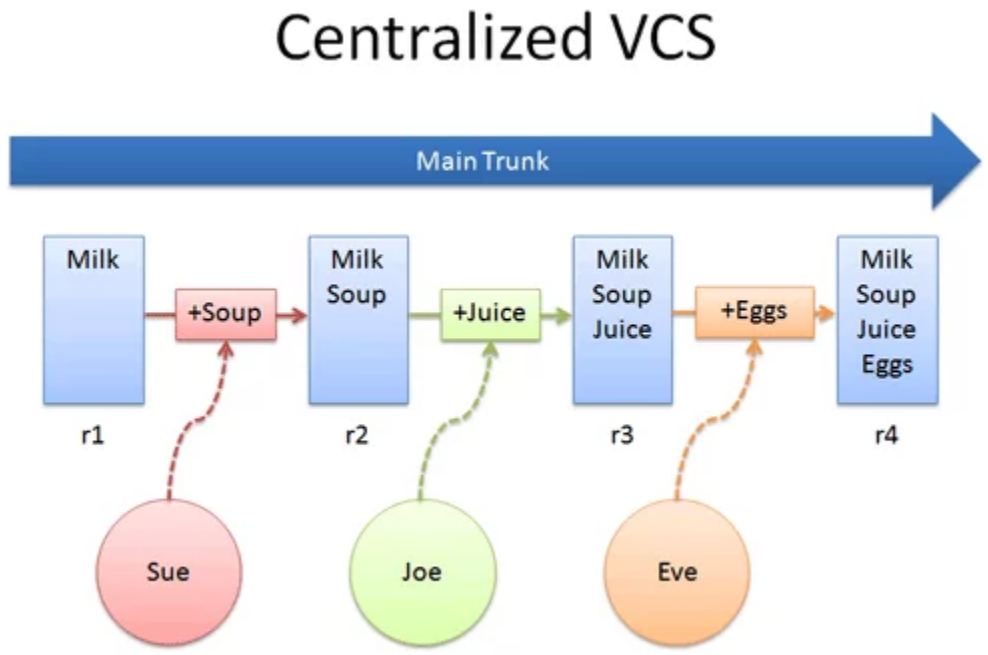
Traditional version control helps you **backup, track and synchronize files**. **Distributed** version control **makes it easy to share changes**. Done right, you can get the best of both worlds: simple merging and centralized releases.

Distributed? What's wrong with regular version control?

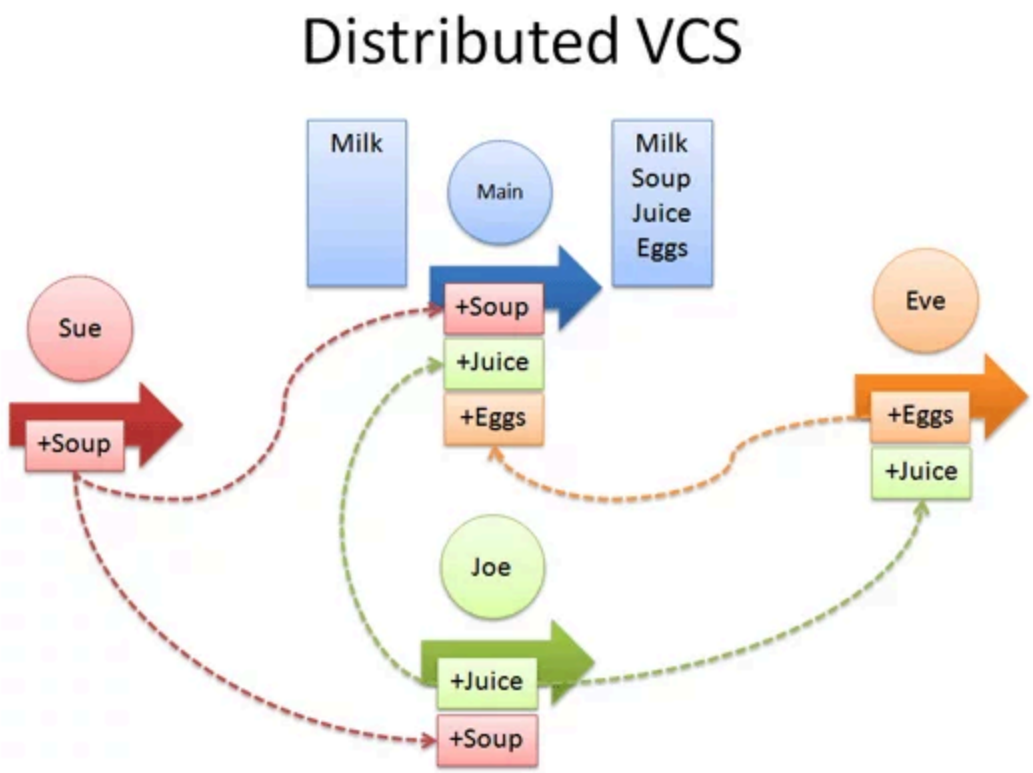
Centralized is simple, and what you'd first invent: **a single place everyone can check in and check out**. It's like a library where you get to scribble in the books.

This model works for backup, undo and synchronization but isn't great for merging and branching changes people make. As projects grow, you want to split features into chunks, developing and testing in isolation and slowly merging changes into the main line. **In reality, branching is cumbersome, so new features may come as a giant checkin, making changes difficult to manage and untangle if they go awry.**

Sure, **merging is always “Possible” in a centralized system, but it's not easy**: **you often need to track the merge yourself to avoid making the same change twice.** Distributed systems make branching and merging painless because they rely on it.



Everyone syncs and checks into the main trunk



In a distributed model, every developer has their own repo. Sue's changes live in her local repo, which she can share with Joe or Eve.

If desired everyone can push changes into a common repo.

Key Concepts

Core Concepts:

- Centralized version control focuses on **synchronizing, tracking, and backing up files**
- Distributed version control focuses on **sharing changes**; every change has a guid or unique id.
- **Recording/Downloading** and **applying** a change are separate steps (in a centralized system, they happen together).
- **Distributed systems have no forced structure**. You can create "centrally administered" locations or keep everyone as peers.

New Terminology

- **Push:** Send a change to another repository (may require permission)
- **Pull:** Grab a change from a repository

Key Advantages

- **Everyone has a local sandbox:** You can make changes and roll back, all on your local machine. No more giant checkins; your incremental history is in your repo
- **It works offline:** You only need to be online to share changes. Otherwise, you can happily stay on your local machine, checking in and undoing, no matter if the “server” is down or you’re on an airplane.
- **It’s fast:** Diffs, commits and reverts are all done locally. There’s no shaky network or server to ask for old revisions from a year ago.
- **It handles changes well:** Distributed version control systems were built around sharing changes. Every change has a guid which makes it easy to track
- **Branching and merging is easy:** Because every developer “has their own branch”, every shared change is like reverse integration. But the guids make it easy to automatically combine changes and avoid duplicates.
- **Less management:** Distributed VCSes are easy to get running; There’s no “always-running” server software to install. Also, DVCSes may not require you to “add” new users’ you just pick what URLs to pull from. This can avoid political headaches in large projects.

Key Disadvantages:

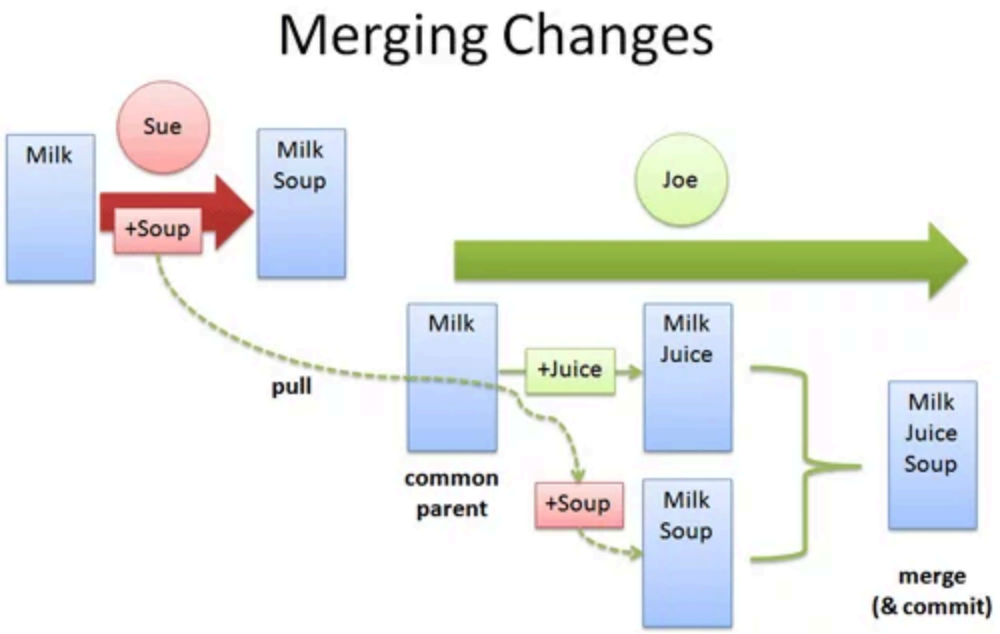
- **You still need a backup:** Some claim your “backup” is the other machines that have your changes. What if they’re offline and you have new changes? With a DVCS, you still want a machine to push changes to “just in case”.
- **There's not really a “latest version”:** If there’s no central location, you don't immediately know whether to see Sue, Joe, or Eve for the latest version. Again, a central location helps clarify what the latest “stable” release is.
- **There aren’t really revision numbers:** Every repo has its own revision numbers depending on the changes. Instead, people refer to change numbers, thankfully you can tag releases with meaningful names.

Understanding Updates and Merging

- **Getting** the change into a repo (pushing or pulling)
- **Applying** the change to the files (update or merge)
- **Saving** the new version (commit)

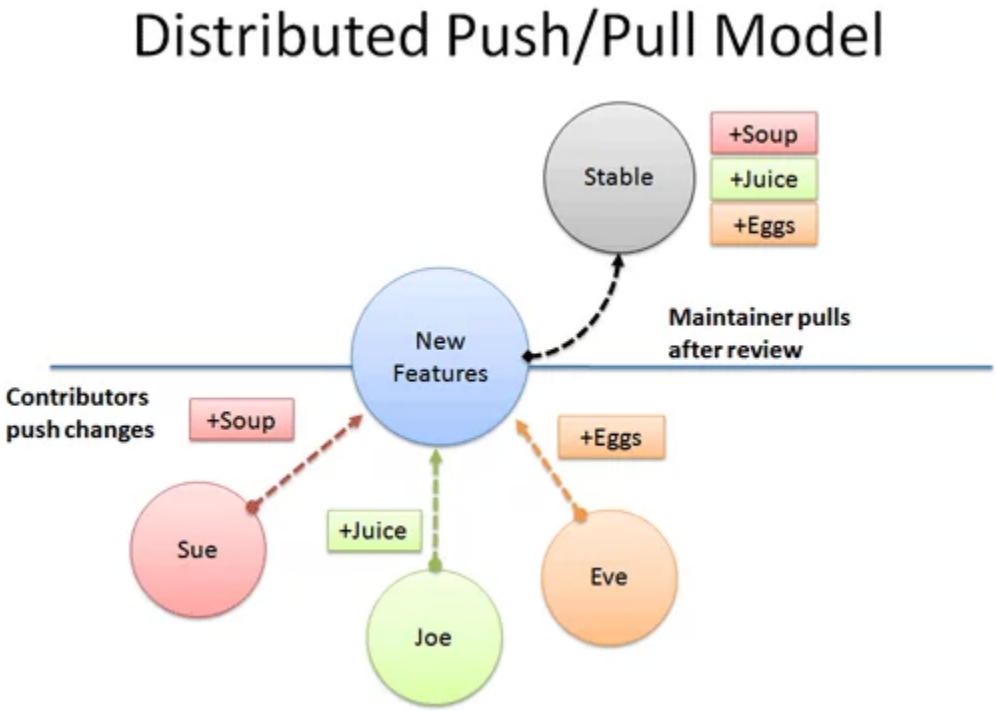
Depending on the change, you can update or merge:

- **Updates** happen when there’s no ambiguity. For example, I pull changes to a file that only you’ve been editing. The file just jumps to the latest revision, since there's no overlapping changes.
- **Merges** are needed when we have conflicting changes. If we both edit a file, we end up with two “branches” (i.e. alternate universes). One world has my changes, the other world has yours. In this case we (probably) want to merge the changes together into a single universe



In this case, a merge is needed because (+Soup) and (+Juice) are changes to a common parent: the list with just “Milk”. After Joe merges the files, Sue can do a regular “pull and update” to get the combined file from Joe. She doesn’t have to merge again on her own.

Organizing a Distributed Project



Sue, Joe and Eve check changes into a common branch. They can trade patches with each other to do simple “buddy builds”: Hey buddy, can you try out these patches? I need to see if it works before I push to the experimental branch.

Later, a maintainer can review and pull changes from the experimental branch into a stable branch, which has the latest release. A distributed VCS helps isolate changes but still provide the “single source” of a centralized system. There are many models of development, from “pull only” (where maintainers decide what to take, and is used when developing Linux) to “shared push” (which acts like a centralized system). A distributed VCS gives you flexibility in how a project is maintained.

Optional Reading: Git Tutorial on GitHub

https://docs.github.com/en/get-started/quickstart/set-up-git

Required Reading: How to Write a Git Commit Message by Chris Beams

Introduction: Why good commit messages matter

If you browse the log of any random Git repository, you will probably find its commit messages are more or less a mess. For example, take a look at these gems from my early days committing to Spring:

```
$ git log --oneline -5 --author cbeams --before "Fri Mar 26 2009"

e5f4b49 Re-adding ConfigurationPostProcessorTests after its brief removal in r814. @Ignore-ing the
testCglibClassesAreLoadedJustInTimeForEnhancement() method as it turns out this was one of the culprits in the
recent build breakage. The classloader hacking causes subtle downstream effects, breaking unrelated tests. The
test method is still useful, but should only be run on a manual basis to ensure CGLIB is not prematurely
classloaded, and should not be run as part of the automated build.

2db0f12 fixed two build-breaking issues: + reverted ClassMetadataReadingVisitor to revision 794 + eliminated
ConfigurationPostProcessorTests until further investigation determines why it causes downstream tests to fail
(such as the seemingly unrelated ClassPathXmlApplicationContextTests)

147709f Tweaks to package-info.java files

22b25e0 Consolidated Util and MutableAnnotationUtils classes into existing AsmUtils

7f96f57 polishing
```

Compare that with these more recent commits from the same repository:

```
$ git log --oneline -5 --author pwebb --before "Sat Aug 30 2014"

5ba3db6 Fix failing CompositePropertySourceTests

84564a0 Rework @PropertySource early parsing logic

e142fd1 Add tests for ImportSelector meta-data

887815f Update docbook dependency and generate epub

ac8326d Polish mockito usage
```

The former varies in **length and form**; the latter is **concise and consistent**.

While many repositories' logs look like the former, there are exceptions. The *Linux kernel* and *Git* itself are great examples. Look at *Spring Boot*, or any repository managed by Tim Pope.

The contributors to these repositories know that a well-crafted Git commit message is the best way to communicate context about a change to fellow developers (and indeed to their future selves). A diff will tell you what changed, but only the commit message can properly tell you why.

Re-establishing the context of a piece of code is wasteful. We can't avoid it completely, so our efforts should go to reducing it [as much] as possible. Commit messages can do exactly that and as a result, a commit message shows whether a developer is a good collaborator. - Peter Hutterer

A well-cared for log is a useful thing. Git blame , revert , rebase , log , shortlog and other subcommands come to life. Reviewing other's commits and pull requests becomes something worth doing, and suddenly can be done independently. Understanding why something happened months or years ago becomes not only possible but efficient.

A project's long-term success rests (among other things) on its maintainability, and a maintainer has few tools more powerful than his project's log. It's worth taking the time to learn how to care for one properly.

In order to create **a useful revision history**, teams should first **agree on a commit message convention** that defines at least the following three things:

1. **Style**: Markup syntax, wrap margins, grammar, capitalization, punctuation. Spell these things out, remove the guesswork, and make it all as simple as possible. The end result

will be a remarkably consistent log that's not only a pleasure to read but that actually **does get read** on a regular basis

2. **Content**: What kind of information should the body of the commit message (if any) contain? What should it **not contain?**
3. **Metadata:** How should issue tracking IDs, pull request numbers, etc. be referenced?

The Seven Rules of a Great Commit Message

1. ***Separate subject from body with a blank line***
    a. Not every commit requires both a subject and a body. Sometimes a single line is fine, especially when the change is so simple that no ruther context is necessary
        i. Eg. *Fix typo in introduction to user guide*
            1. Nothing more needs to be said, if the reader wants to know what the type was she can look at the change itself
                a. I.e. use `git show` or `git diff` or `git log -p`
    b. If a commit merits a bit of explanation and context, you need to write a body
    c. Separation from the body pays off when browsing the log. Here's an **example:**
        i.
```
$ git log
commit 42e769bdf4894310333942ffc5a15151222a87be
Author: Kevin Flynn <kevin@flynnsarcade.com>
Date:   Fri Jan 01 00:00:00 1982 -0200

 Derezz the master control program

 MCP turned out to be evil and had become intent on world domination.
 This commit throws Tron's disc into MCP (causing its deresolution)
 and turns it back into a chess game.
```
2. ***Limit the subject line to 50 characters***
    a. 50 characters is not a hard limit, just a rule of thumb. Keeping subject lines at this length ensures that they are readable, and forces the author to think for a moment about the most concise way to explain what's going on.
        i. GitHubs UI is aware of these conventions and will warn you if you exceed 50 characters
        ii. Any subject line exceeding 72 characters will be truncated
3. ***Capitalize the subject line***
    a. Self explanatory
4. ***Do not end the subject line with a period***
    a. Trailing punctuation is unnecessary in subject lines. Besides, space is precious when you're trying to keep them to 50 chars or less
5. ***Use the imperative mood in the subject line***
    a. *Imperative mood* just means "spoken or written as if giving a command or instruction", **A few examples:**

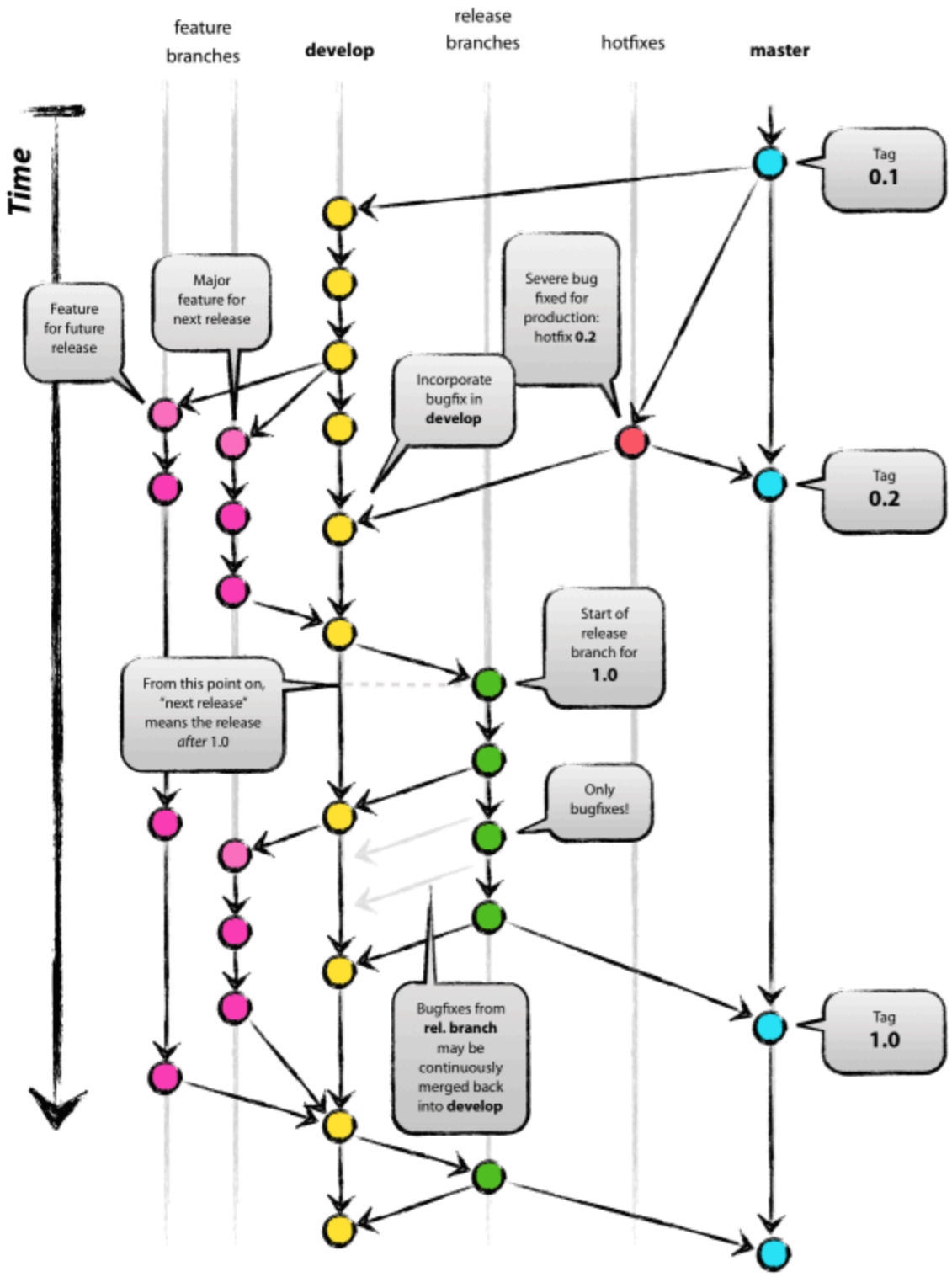
        i. Clean your room
        ii. Close the door
        iii. Take out the trash
    b. Imperative may seem rude but its perfect for Git commit subject lines. One reason for this is that Git itself uses the imperative whenever it creates a commit on your behalf.
    c. When you write your commit messages in the imperative, you're following Git's own built-in conventions. For **example:**
        i. Refactor subsystem X for readability
        ii. Update getting started documentation
        iii. Remove deprecated methods
        iv. Release version 1.0.0
    d. A properly formed Git commit subject line should always be able to complete the following sentence:
        i. If applied, this commit will *your subject line here*
        ii. Example:
            1. If applied, this commit will **refactor subsystem X for readability**

6. Wrap the body at 72 characters
    a. Git never wraps text automatically. When you write the body of the commit message, you must mind its right margin, and wrap text manually.
7. Use the body to explain what and why vs. how
    a. Focus on making clear the reasons why you made the changes in the first place – the way things worked before the change (and what was wrong with that), the way they work now, and why you decided to solve it the way you did.

Example:

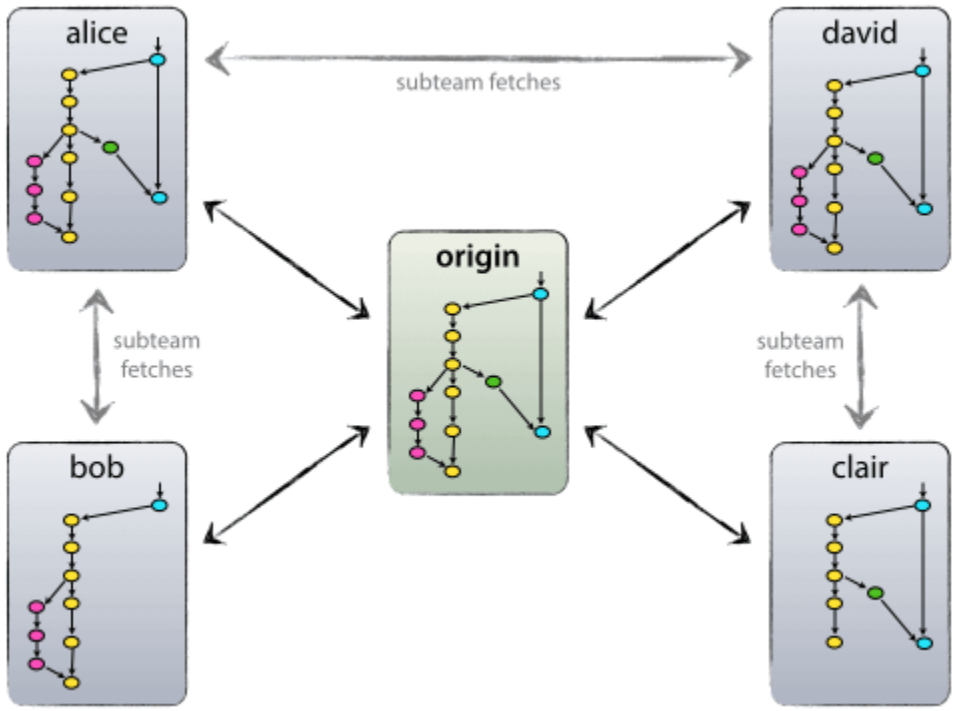
```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

Required Reading: A Successful Git Branching Model by Vincent Driessen


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Decentralized But Centralized

The repository setup we use and that works well with this branching model, is that with a **central "truth" repo.** Note *this repo is only considered to be the central one* (since Git is a DVCS, there is no such thing as a central repo at a technical level). We will refer to this repo as **origin**.



Each developer pulls and pushes to the origin. But besides the centralized push-pull relationships, each developer may also pull changes from other peers to form sub teams.

For example, it might be useful to work together with two or more developers on a bigg new feature, before pushing the work in progress to origin prematurely. In the figure above, there are subteams of Alic and Bob, Alice and David, and Clair and David.

Technically, this means nothing more than that Alice has defined a Git remote, named bob, pointing to Bob's repository, and vice versa.

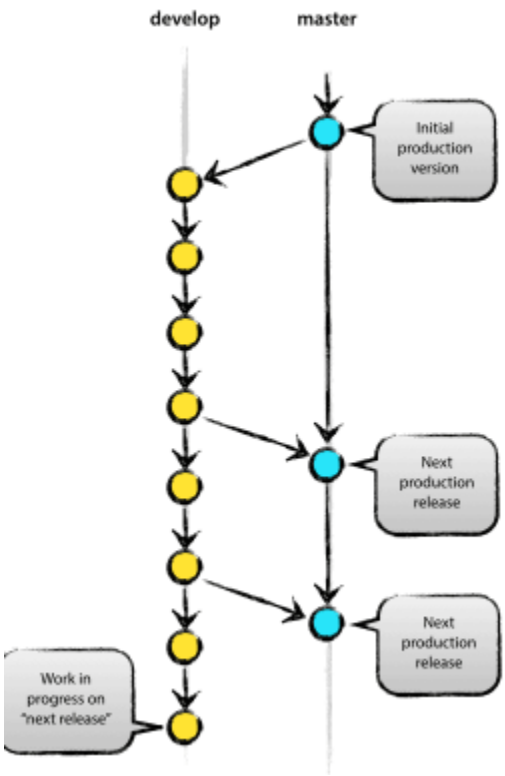
The Main Branches

The Central Repo holds two main branches with an infinite lifetime:

- **Master**
    - The main branch where the source code of **HEAD** always reflects a **production-ready state**
- **Develop**

- The main branch where the source code of **HEAD** always reflects a state with the latest delivered development changes for the next release. Some would call this the “integration branch”. This is where any automatic nightly builds are built from.
- The source code reaches a **stable point** and is ready to be released, all of the changes should be merged back into master somehow and then tagged with a release number.

Each time when changes are merged back into master, this is a new production release by definition. We tend to be very strict at this, so that theoretically, we could use a Git hook script to automatically build and roll-out our software to our production servers every time there was a commit on Master.



Supporting Branches

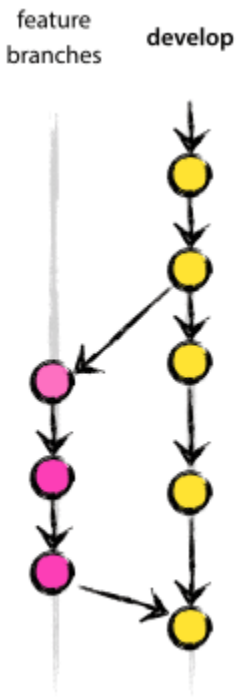
Next to the main branches are a variety of supporting branches to aid parallel development between team members, ease tracking of features, prepare for production releases and to assist in quickly fixing live production problems.

Unlike the main branches, these branches always have a limited life time, since they will be removed eventually.

Different types of branches we may use:

- Feature Branches
- Release Branches
- Hotfix Branches

Each of these branches have a specific purpose and are bound to strict rules as to which branches may be their originating branch and which branches must be their merge targets.



Feature Branches

- May **branch off** from develop
- Must **merge back** into develop
- Branch **naming convention** is anything except master, develop, release-*, or hotfix-*

Feature branches are used to develop new features for the upcoming or distant future release.
When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into develop.

Creating a feature branch

When starting work on a new feature, branch off from the `develop` branch.

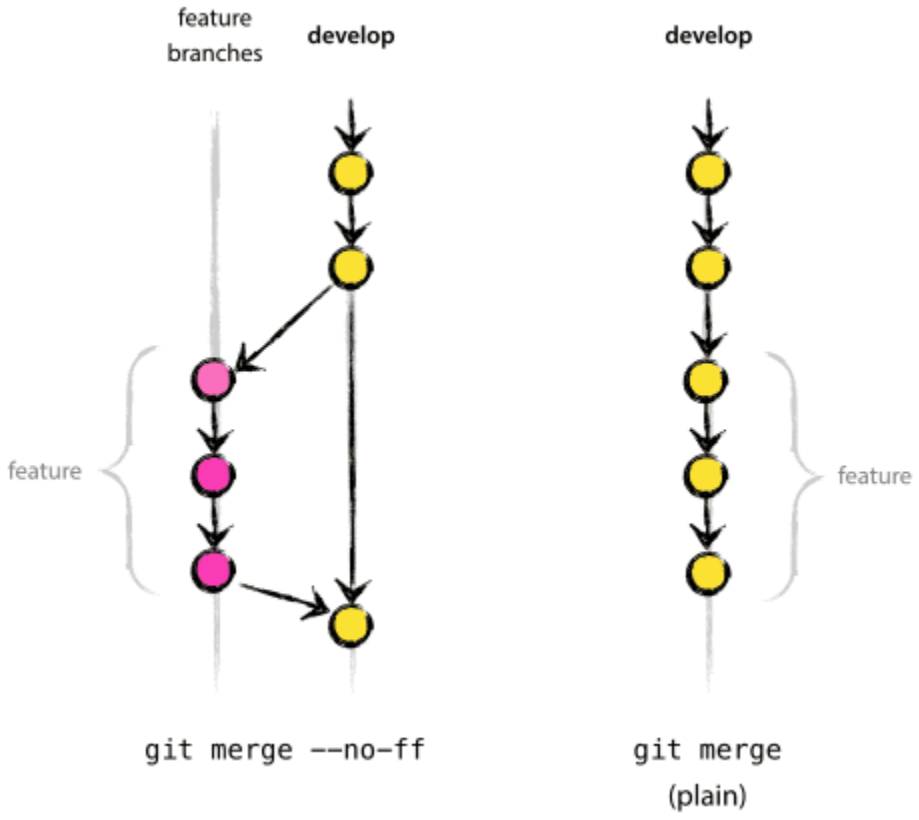
```
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
```

Incorporating a finished feature on develop

Finished features may be merged into the `develop` branch to definitely add them to the upcoming release:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

The –no-ff flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoid losing information about the historical existence of a feature branch and groups together all commits that added the feature.



In the latter case, it is impossible to see from the Git history which of the commit objects together have implemented a feature—you would have to manually read all the log messages. Reverting a whole feature (i.e. a group of commits), is a true headache in the latter situation, whereas it is easily done if the --no-ff flag was used.

Yes, it will create a few more (empty) commit objects, but the gain is much bigger than the cost.

Release Branches

May branch off from `develop`
Must merge back into `develop` and `master`
Branch naming convention: `release-*`

Release branches support preparation of a new production release. They allow for last-minute dotting of i's and crossing t's. Furthermore, they allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the develop branch is cleared to receive features for the next big release.

The key moment to branch off a new release branch from develop is when develop (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to develop at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number—not any earlier. Up until that moment, the develop branch reflected changes for the "next release", but it is unclear whether that "next release" will eventually become 0.3 or 1.0, until the release branch is started. That decision is made on the start of the release branch and is carried out by the project's rules on version number bumping.

Creating a release branch

Release branches are created from the **`develop`** branch.

Example:

- Say version 1.1.5 is the current production release and we have a big release coming up. The state of **`develop`** is ready for the "next release" and we have decided that his will become version 1.2. So we branch off and give the release branch a name reflecting this new version number:

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"
$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.
$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

- After creating a new branch and switching to it, we bump the version number. Here, bump-version.sh is a fictional shell script that changes some files in the working copy to reflect the new version (this can of course be a manual change – the point being that some files change). Then the bumped version number is committed.
- This new branch may exist for a while, until the release may be rolled out definitely. During that time, bug fixes may be applied in this branch (rather than on the **develop branch).** Adding large new features here is strictly prohibited. They must be merged into **develop ,** and therefore, wait for the next big release.

Finishing a release branch

When the state of the release branch is ready to become a real release, some actions need to be carried out.

First, the release branch is merged into **master** (since every commit on **master** is a new release by definition).

Next, that commit on **master** must be tagged for easy future reference to this historical version.

Finally, the changes made on the release branch need to be merged back into **develop**, so that future releases also contain these bug fixes.

The first two steps in Git:

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2
```

The release is now done, and tagged for future reference.

To keep the changes made in the release branch, we need to merge those back into **develop**, though.

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
```

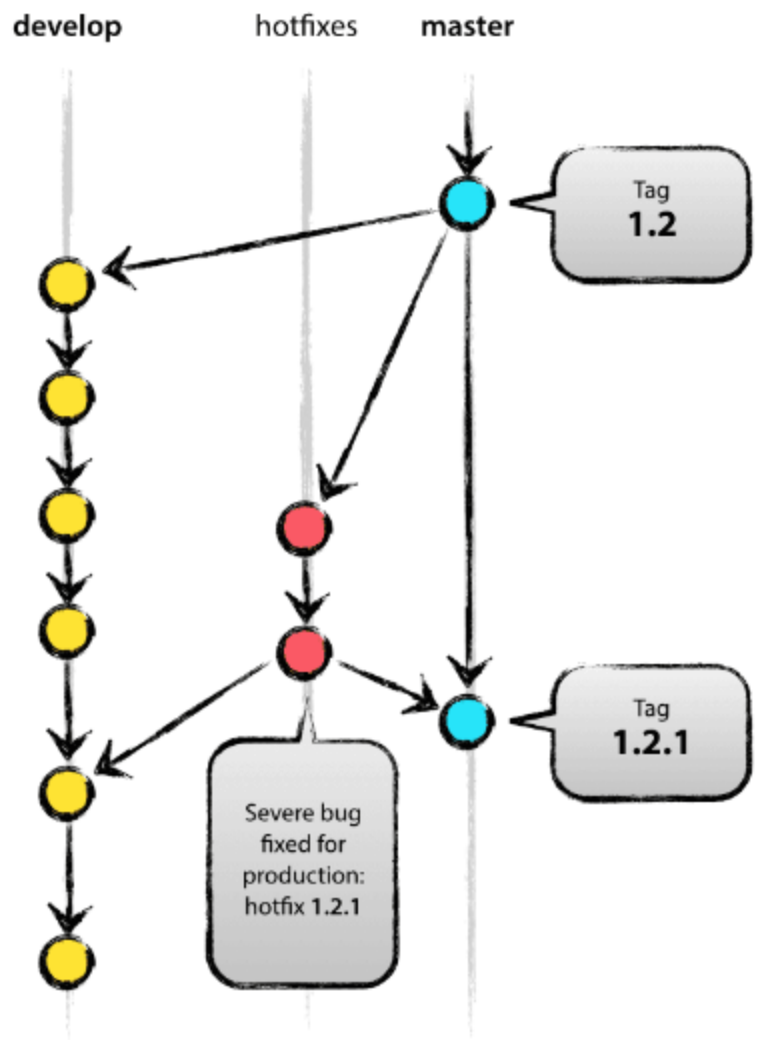
This step may well lead to a merge conflict (probably even, since we have changed the version number). If so fix it and commit.

Now we are really done and the release branch may be removed, since we dont need it anymore:

```
$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe).
```

Hotfix Branches

May branch off from **master**,
Must merge back into **develop and master**
Branch naming convention **hotfix-***



Hot fix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned.

They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

The essence is that work of team members (on the **develop** branch) can continue, while another person is preparing a quick production fix.

Creating the hotfix branch

Hotfix branches are created from the **master** branch.

```
$ git checkout -b hotfix-1.2.1 master
```

```
Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1
Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

Dont forget to bump the version number after branching off!

Then, fix the bug and commit the fix in one or more separate commits.

```
$ git commit -m "Fixed severe production problem"
[hotfix-1.2.1 abbe5d6] Fixed severe production problem
5 files changed, 32 insertions(+), 17 deletions(-)
```

Finishing a hotfix branch

When finished, the bugfix needs to be merged back into **master**, but also needs to be merge back into **develop**, in order to safeguard that the bugfix is included in the next release as well.

- This is completely similar to how release branches are finished.

*First, update **master** and tag the release:*

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2.1
```

*Next, include the bugfix in **develop** too:*

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
```

The one exception to the rule here is that, **when a release branch currently exists, the hotfix changes need to be merged into that release branch, instead of develop**.

- Back-merging the bugfix into the release branch will eventually result in the bugfix being merged into **develop** too, when the release branch is finished.

- (if work in **develop** immediately requires this bugfix and cannot wait for the release branch to be finished, you may safely merge the bugfix into develop now already as well.)

Finally remove the temporary branch:

```
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).
```

Required Reading: A successful Git branching model considered harmful

Week 4

Test-Driven Development

- Write tests before implementation
- Process:
    - Write an interface and method signatures
    - Write one simple test, that fails.
    - Write just enough code to pass the test.
    - Clean up/improve the code (refactor)
    - Repeat until product is complete
    - *Ensures complete test coverage (white box)*
- These are unit tests - but test the interface
    - Still need to write Unit tests

Technical Debt

- Implementing a feature, two ways to do it
    - The quick, but messy way
    - The slower, but cleaner way

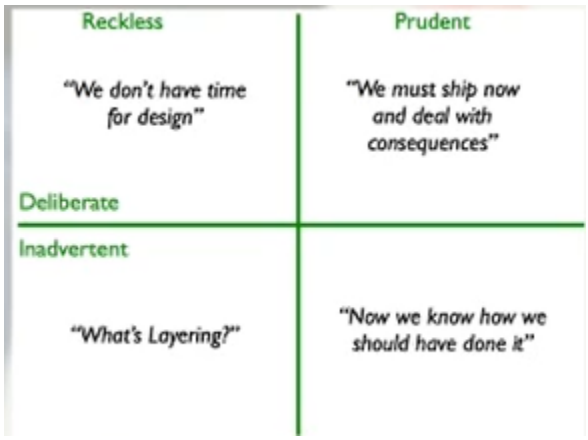
The quick, but messy way:

- Faster delivery
- Harder to make changes going forward

The slow, but cleaner way

- Slower delivery
- Easier to make changes going forward

Technical debt quadrants



Tadeoffs

- Analysis paralysis
    - Spending too much time designing it
    - Too much time trying not to go into debt
    - **End up not moving on at all**
- Agile: deliver working software
    - Just enough documentation
    - Get into some inadvertent debt
    - Go slightly into debt but not deeply

Technical debt != bad
- Credit card debt != bad

Required Reading: Technical Debt - Martin Fowler

- Think of technical debt like financial debt, the extra effort that it takes to add new features is the interest paid on the debt.

Cruft: deficiencies in internal quality that make it harder than it would ideally be to modify and extend the system further.

***Thinking of technical debt**:*

- Imagine I have a confusing module structure in my code base. I need to add a new feature. If the module structure was clear, then it would take me four days to add the feature but with this cruft, it takes me six days. The two day difference is the interest on the debt.

- What most appeals to me about the debt metaphor is how it frames how I think about how to deal with this cruft. I could take five days to clean up the modular structure, removing that cruft, metaphorically paying off the principal. If I only do it for this one feature, that's no gain, as I'd take nine days instead of six. But if I have two more similar features coming up, then I'll end up faster by removing the cruft first.

- Usually the best route is to do what we usually do with financial debts
    - Pay the principal off gradually.
        - On the first feature ill spend an extra couple of days to remove some of the cruft
            - May be enough to reduce the interest rate on future enhancements to a single day.
            - Still extra time but by removing the cruft I'm making it cheaper for future changes to this code.
        - The great benefit of gradual improvement is that it naturally means we spend more time on removing cruft in those areas we modify frequently, which are exactly those areas of the code base where we most need the cruft to be removed.

Required Reading: Technical Debt Quadrant - martin fowler

Technical debt should be reserved for cases when people have made a considered decision to adopt a design strategy that isnt sustainable in the longer term, but yields a short term benefit, such as making a release.

- The point is that the debt yields value sooner, but needs to be paid off as soon as possible.
- A particular benefit of the debt metaphor is that its very handy for communicating to non-technical people.

Required Reading: The Tragedy of 100% Code Coverage

Where is the tragedy?

Developers are wasting time writing tests that are pointless, and that will need to be maintained by future generations. The tragedy is that instead of using the correct tool for the job, we decide to keep plugging away with the wrong ones, for no particular good reason.

The tragedy is that once a good practice becomes mainstream we seem to forget how it came to be, what its benefits are, and most importantly, what the cost of using it is.

Author believes that everyone should achieve 100% code coverage in one project, to understand what both extremes are (0 unit tests - 100% code coverage with tests)

Required Reading: SOLID Principles: The Software Developers Framework to Robust & Maintainable Code by Khalil Stemmler

Writing code can be like pouring concrete. If what you've written in the past doesnt suit your needs today, it can be expensive to change. And change is inevitable. So if we're writing code, we need to write code that can be changed.

Making things work the first time was **easy**, but adding features to existing code, making it work the second time, the third time and the fourth time, multiple iterations, can be hard to do without introducing bugs.

Iterations and incremental improvements are what agile is all about.

When we first start out coding, we often have little regard for software design. After we achieve dexterity with a programming language, we focus our efforts on fighting undefined errors.

In order to go fast, we need to go well.

What are the SOLID Principles?

The solid principles are a set of software design principles that teach us how we can structure our functions and classes in order to be as **robust**, **maintainable** and **flexible** as possible.

1) **S: Single Responsibility Principle**
2) **O: Open-Closed Principle**
3) **L: Liskov-Substitution Principle**
4) **I: Interface Segregation Principle**

5) **D: Dependency Inversion Principle**

The primary benefits of becoming well acquainted with the SOLID principles are that you'll learn how to:

- Write code thats testable
- Write code thats easily understood
- Write code where things are where they're expected to be
- Write code where classes narrowly do what they were intended to do
- Write code that can be adjusted and extended quickly
- Write code that can be adjusted and extended quickly without producing bugs
- Write code that separates the *policy (rules)* from the *details (implementation*)
- Write code that allows for implementations to be swapped out (think swapping out Email API's, ORMs or web server frameworks)

S: Single Responsibility Principle

" A class or function should only have one reason to change"

Example:

If we had an HR department, an Accounting department and an IT department in an enterprise application that calculates pay, and reports and saves hours, we'd better make sure that we've split up (or abstracted) the operations most likely to change for **each department**.

Here's an SRP violation:

```
class Employee {
  public calculatePay (): number {
    // implement algorithm for hr, accounting and it
  }
  public reportHours (): number {
    // implement algorithm for hr, accounting and it
  }

  public save (): Promise<any> {
    // implement algorithm for hr, accounting and it
  }
}
```

This is bad. The algorithm for each department is located in the same class. If one department were to request a change to one of their respective algorithms, it has the increased potential to ripple into another department's algorithm.

Making this work would require some nasty "if or switch" statements.

Design Tip: if we see lots of code in switch statements, that should be a signal to us of a potential refactoring from a switch statement to several classes.

A solution:

```
abstract class Employee {
  // This needs to be implemented
  abstract calculatePay (): number;
  // This needs to be implemented
  abstract reportHours (): number;
  // let's assume THIS is going to be the
  // same algorithm for each employee- it can
  // be shared here.
  protected save (): Promise<any> {
    // common save algorithm
  }
}

class HR extends Employee {
  calculatePay (): number {
    // implement own algorithm
  }
  reportHours (): number {
    // implement own algorithm
  }
}

class Accounting extends Employee {
  calculatePay (): number {
    // implement own algorithm
  }
  reportHours (): number {
    // implement own algorithm
  }

}

class IT extends Employee {
  ...
}
```

Each employee in this social structure has a single place where we can go to in order to adjust their respective algorithm that is most likely to change.

The key thing is to separate responsibility based on the social structure of the users using the application

O: Open-Closed Principle

" A software artifact should be open for extension but closed for modification"

Generally this principle is all about writing your code in such a way so that when you need to add new functionality, it shouldn't require changing the existing code.

To do this, we **write interfaces and abstract classes** in order to dictated the **higher-level policy** that needs to be implemented, and then we **implement that policy** using concrete classes.

Example:

Lets say our boss told us that he wanted to use SendGrid for sending emails.

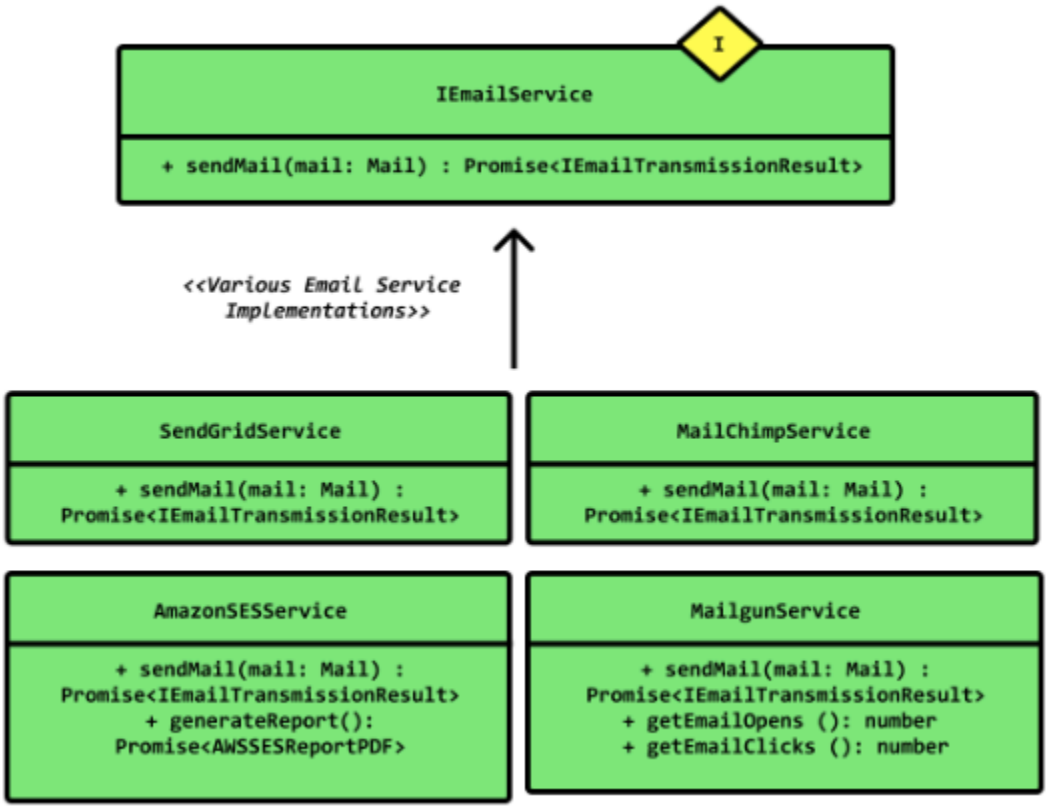
So we went and coded out a concrete sendGridService class, connecting to the Sendgrid API.

```
class SendGridService {
  constructor (sendgridInstance) {
    this.sg = sendgridInstance;
  }

  sendMail (from, to, body) {
    // format the mail object to the sendgrid api shape
    // send it
    // create a result object
    // return the result (success, failure, bounded, etc)
  }
}
```

3 months later he tells us he wants to use MailChimp instead because SendGrid is too expensive

So we go and code a concrete MailChimpService. But in order to hook it up, we're going to have to change and potentially break a lot of code.

Following OCP, we could define an interface that specifies what a mail service can do, and leave the actual implementation to be figured out separately.



The main idea is to <u>keep the policy separate from the detail</u> in order to enable loose coupling.

"Higher level-components are protected from changes to lower level components."

L: Liskov-Substitution Principle

The easiest way to explain it is that we should be able to swap one implementation for another.

Think of the Mail example we had:

Since we've defined an IMailService interface:

```typescript
type TransmissionResult = 'Success' | 'Failure' | 'Bounced'

interface IEmailTransmissionResult {
  result: TransmissionResult;
  message?: string;
}

interface IMailService {
  sendMail(email: IMail): Promise<IEmailTransmissionResult>
}
```

We can implement various email services, as long as they implement the IMailService interface and the required sendMail(email: Mail) method.

```typescript
class SendGridEmailService implements IMailService {
  sendMail(email: IMail): Promise<IEmailTransmissionResult> {
    // algorithm
  }
}

class MailChimpEmailService implements IMailService {
  sendMail(email: IMail): Promise<IEmailTransmissionResult> {
    // algorithm
  }
}

class MailGunEmailService implements IMailService {
  sendMail(email: IMail): Promise<IEmailTransmissionResult> {
    // algorithm
  }
}
...
```

And then we can “dependency inject” it into our classes, making sure we refer to the **interface** it belongs to rather than one of the concrete implementations.

```
class CreateUserController extends BaseController {
  private emailService: IEmailService;
  constructor (emailService: IEmailService) { // like this 😇
    this.emailService = emailService;
  }

  protected executeImpl (): void {
    // handle request

    // send mail
    const mail = new Mail(...)
    this.emailService.sendMail(mail);
  }
}
```

Now all of these are valid.

```
const mailGunService = new MailGunEmailService();
const mailchimpService = new MailChimpEmailService();
const sendgridService = new SendGridEmailService();

// any of these are valid
const createUserController = new CreateUserController(mailGunService);
// or
const createUserController = new CreateUserController(mailchimpService);
// or
const createUserController = new CreateUserController(sendgridService);
```

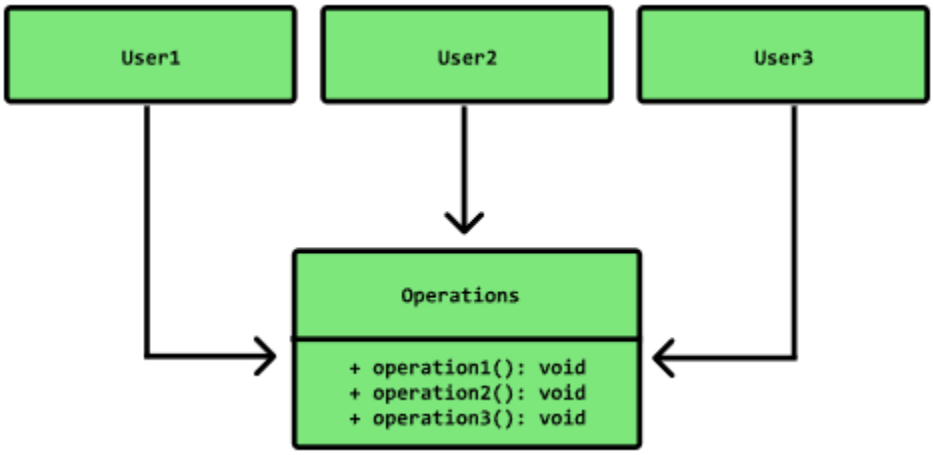
I : Interface Segregation Principle

"Prevent classes from relying on things that they dont need"

In order to prevent this, we should make sure to really split up the unique functionality into interfaces

Example:

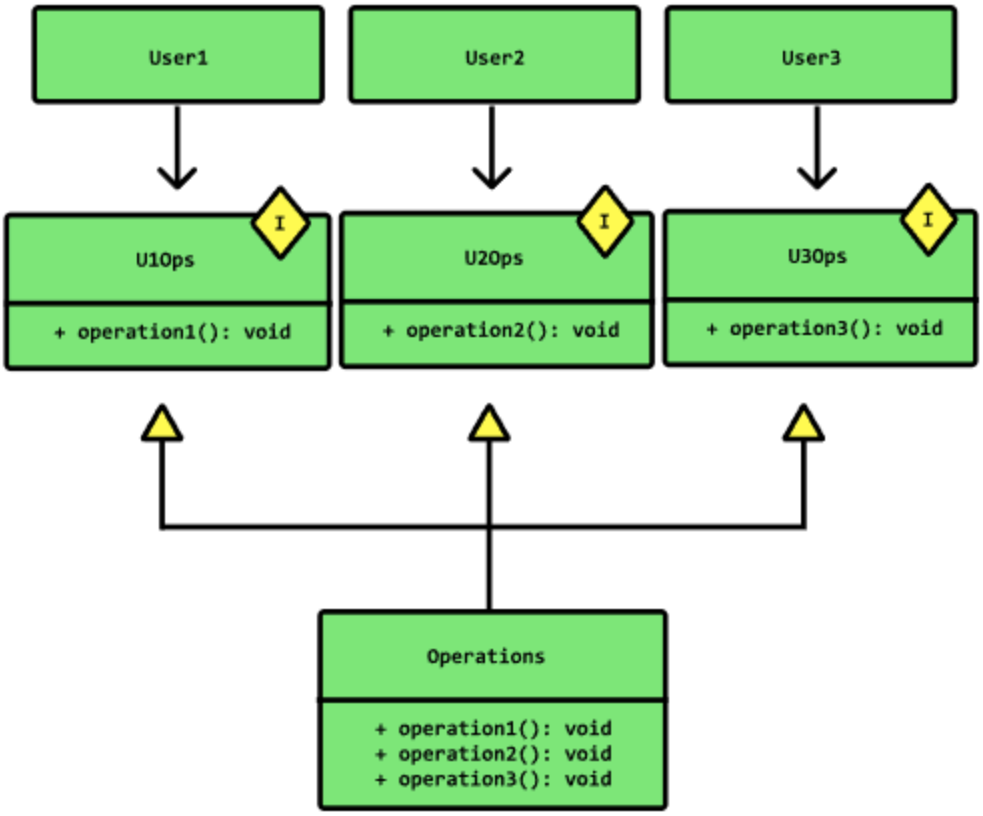
Lets say we have 3 different User classes, that use 3 different methods on the same *Operations* class, and for each *User* class, we're depending on 2 additional operations that we dont need.



For this, it might not seem like a big deal, but it could be a big deal if the constructor for the *operations* class required us to inject all different kinds of dependencies in order to satisfy the other 2 operations that we didnt need:

```
class Operations implements U1Ops, U2Ops, U3Ops {
  constructor (
    userRepo: IUserRepo,
    emailService: IEmailService,
    authService: IAuthService,
    redisService: IRedisService,
    ... // and more
  )

  ...
}
```

To fix it we can start by making sure that we rely on abstractions to the things that we need.



And then, if it were cumbersome to create the *Operations* class because it depended on so many things, taking *User1* for example, we would be able to create a class that ONLY implements *U1ops*

```
class User1Operations implements U1Ops {
  constructor (userRepo: IUserRepo) {
    ...
  }
  ...
}
```

Now *User1* only needs *User1Operations* and not all the other junk from *Operations*

D: Dependency Inversion Principle

"Abstractions should not depend on details. Details should depend on abstractions."

What's an abstraction? An interface or abstract class.

What's a detail? A concrete class.

Abstractions should never depend on details. Ok, so that means, try not to do this:

```
interface IMailService {
  // refering to concrete "PrettyEmail" and "ShortEmailTransmissionResu
  sendMail(email: PrettyEmail): Promise<ShortEmailTransmissionResult>
}
```

But instead do this:

```
class SendGridEmailService implements IMailService {
  // concrete class relies on abstractions
  sendMail(email: IMail): Promise<IEmailTransmissionResult> {
  }
}
```

We can reference concrete classes:

```
// user/http/router/index.js
import * as express from 'express'
import { Router } from 'express'
import { CreateUserController } from '../controllers'
import { MailGunEmailService } from '../services'

const mailGunService = new MailGunEmailService();
const createUserController = new CreateUserController(mailGunService);

const userRouter = Router();

userRouter.post('/new', (req, res) => createUserController.exec(req, res

export {
  userRouter
}
```

We need to reference those concrete *CreateUserController* and *MailGunEmailService* classes in order to hook them up.

But we only do this to hook them up. We shouldnt refer to concrete classes from another concrete class *directly*.

Therefore do this:

```
class CreateUserController extends BaseController {
  private emailService: IEmailService; // <- abstraction
  constructor (emailService: IEmailService) { // <- abstraction
    this.emailService = emailService;
  }

  protected executeImpl (): void {
    // handle request

    // send mail
    const mail = new Mail(...)
    this.emailService.sendMail(mail);
  }
}
```

Dont do this:

```
class CreateUserController extends BaseController {
  // we're limiting ourselves to a particlar concrete class.
  private emailService: SendGridService; // <- concretion
  constructor (emailService: SendGridService) { // <- concretion
    this.emailService = emailService;
  }

  protected executeImpl (): void {
    // handle request

    // send mail
    const mail = new Mail(...)
    this.emailService.sendMail(mail);
  }
}
```

Definitely dont do this:

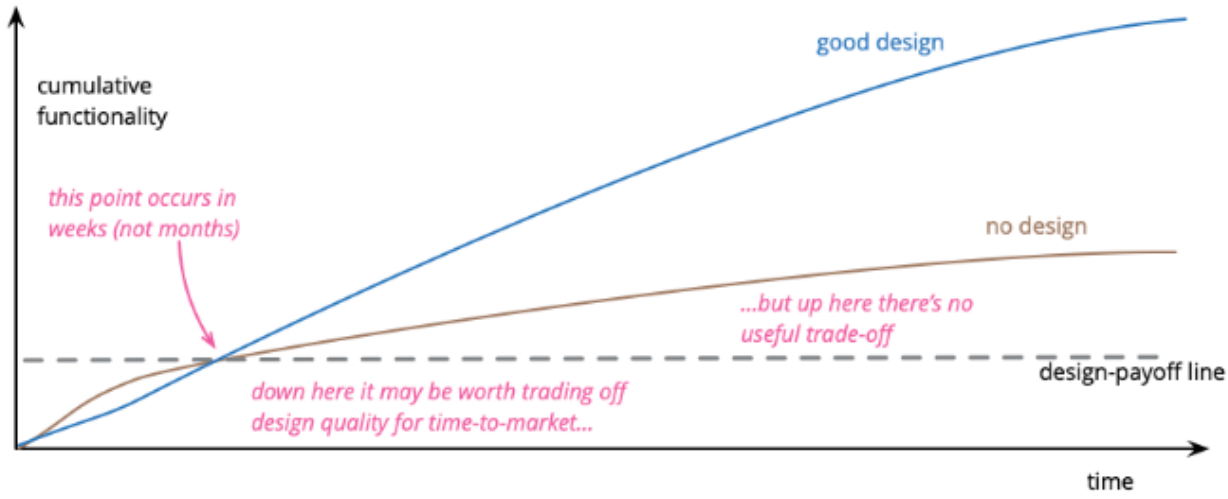
```
import { SendGridService } from '../../services'; // <- source code depe

class CreateUserController extends BaseController {
  // impossible to mock for tests
  private emailService: SendGridService = new SendGridService(); // <- c
  constructor () {
  }

  protected executeImpl (): void {
    // handle request

    // send mail
    const mail = new Mail(...)
    this.emailService.sendMail(mail);
  }
}
```

Required Reading: Design Stamina Hypothesis by Martin Fowler

Is it worth the effort to design software well?

There is a notion that design is something you can trade off for greater speed. Design activities certainly do take up time and effort, but they payoff because they make it easier to evolve the software into the future. You can save short-term time by neglecting design, but this accumulates TechnicalDebt which will slow your productivity later.

Putting effort into the design of your software improves the stamina of your project, allowing you to go faster for longer.

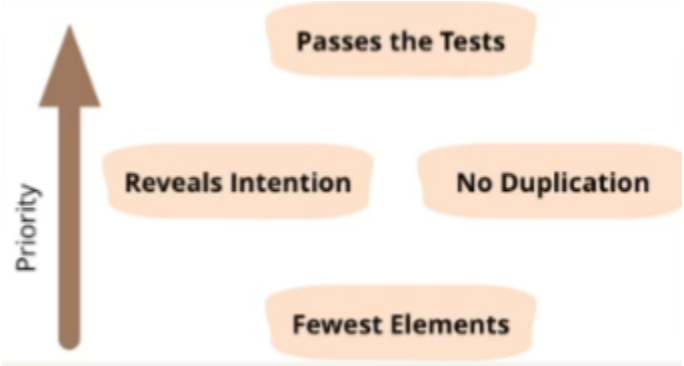


The problem with no design, is that by not putting effort into the design, the code base deteriorates and becomes harder to modify, which lowers the productivity, which is the gradient of the line.

Good design keeps its productivity more constant so at some point it overtakes the cumulative functionality of the no-design project and will continue to do better.

Week 5

Design Principles

- Principles of simple design
    - Passes tests
    - Reveals Intention
    - No Duplication
    - Fewest Elements
- 

Coupling and Cohesion

- Measurements of relationships *between (coupling)* and *within (cohesion)* units
- Software design rules and principles affect coupling and cohesion

Coupling

- Measuring **dependencies between** components.
- Making changes in one layer shouldn't cascade into other layers
- Low coupling == good
- Highly coupled to an interface is fine

Cohesion

- Measuring relationships **within** a component
    - UNIX philosophy: Do one thing and do it well.
    - String.java
        - Well written
        - Does one thing and does it well
    - High cohesion == good

Summary

Cohesion: Looking at it
Coupling: Looking at what it **communicates with**

Code Quality

Acronym Soup

- General Software design principles/rules
    - Slightly more tangible versions of principles for simple design
- Tools to manage coupling and cohesion

DRY:

- **Don't Repeat Yourself**
    - Code or behaviour
    - E.g., validation in two places.
    - Observe and resolve during *refactoring*
    - Helps reduce coupling

LOA:

- **Levels of Abstraction**

        - Define an **interface** (what your unit can do without specifying how)
        - Hide implementation details from the caller
        - E.g., I/O in UNIX
        - Helps reduce coupling

SOC:

- **Separation of Concerns**
    - UNIX philosophy: Do one thing and do it well.
    - E.g., Application logic should not know about presentation
    - Helps increase cohesion and reduce coupling

POLK:

- **Principle of Least Knowledge**
    - Also: Law of Demeter
    - Each unit knows about itself and interfaces it depends on
    - E.g., Walking a dog
        - If we're walking dog, we have a leash, we don't need to know what kind of a dog is on the leash, we just know that it will follow us and the leash
    - Helps reduce coupling

POLA:

- **Principle of Least Astonishment**
    - What do you expect this thing to do?
    - Answer should not differ from what **actually happens**
    - What does this javascript code do?
        - ['10','10','10'].map(parseInt);
            - => [10,NaN,2]
                - ???
                - Answer is completely different than what we expected

Refactoring

- What is refactoring?
    - Change **implementation** without changing *behaviour*
- Goals:
    - Maintainability, readability
    - Remove code smells

Smells

A surface indication that usually corresponds to a deeper problem in the system.

- No comprehensive list, no rigid definition
    - E.g., too long methods – what is too long?
- A smell doesnt **always** indicate a problem
    - Usually a violation of **SOLID** principles

Refactoring and SOLID

- There are many named refactorings
    - Simple data-level refactoring (renaming, named constants)
    - Statement-level refactoring (decomposing complex Boolean expressions)
    - Routine-level refactoring (decompose methods, convert method to class)
    - Class implementation refactoring (moving methods)
    - Class interface refactoring (move methods, separate classes)

SOLID: Five key indicating smells:

- S:
    - Our class should be responsible for **one** thing
    - Single responsibility
- O:
    - Open-closed principle
    - If you have a class and you want to make a modification you shouldnt have to modify that class
        - It should be open for extension and closed for modification
        - You should be making a subclass and **extending** to add functionality
- L:
    - Liskov substitution
    - If you inherited from an interface and someone is using that interface, you should be able to use any implementation based on that interface, or any subclass, you shouldnt change the meaning of what that class does.
- I:
    - Dont put a thousand things into an interface
    - Keep the interface small
- D:
    - Depend on the interface
    - Dont depend on a class, you should depend on the interface

Often when you violate one of these, you've violated more than one

Dependency Injection AKA passing arguments

Really simple but really powerful.

- Imagine we want to be able to use more than one database in our project.

```
class MyLogicLayer{
    private DBLayer db;
    public MyLogicLayer(){
        this.db = new DBLayer();
    }
    [....]
}
```

This is a DIP violation.

- Instead of depending on an interface we are depending on the DB layer which is the actual implementation of the db layer.
- We want to depend on the interface

How do we fix it?

- Instance variable becomes the interface type
- Simply pass the database we want to use

```
class MyLogicLayer{
    private IDBLayer db;
    public MyLogicLayer(IDBLayer db){
        this.db = db;
    }
    [....]
}
```

We can also use a default:

```
class MyLogicLayer{
    private IDBLayer db;
    public MyLogicLayer(IDBLayer db){
        this.db = db;
    }
    public MyLogicLayer(){
        this.db = new DBLayer();
    }
    [....]
}
```

- Overload the constructor

Why?

- Now we can use any database
    - Unit tests can use fake implementations, program can use real implementations.

Can be used everywhere

- Command-line arguments
- Which config file to use

Required Reading: Cohesion and Coupling: The Difference by Vliadimi Khorikov

Difference between Cohesion and Coupling

Cohesion: Represents the degree to which a part of a code base forms a logically single, atomic unit

- Can also be put as the number of connections inside some code unit.
- If the number is low, then the boundaries for the unit are probably chosen badly, the code inside the unit is not logically related
- A unit is not necessarily a class. It mighty be a method, a class, a group of classes, or even a module or an assembly

Coupling: Represents the degree to which a single unit is independent from others.

- In other words it is the number of connections between two or more units. The fewer the number, the lower the coupling.

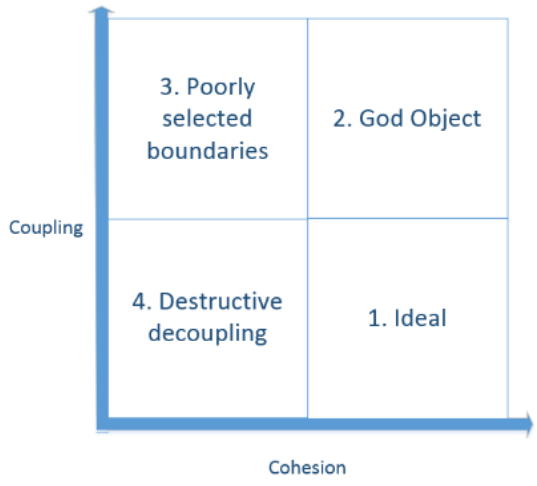
High Cohesion, Low Coupling Guideline

High Cohesion means keeping parts of a code base that are related to each other in a single place.

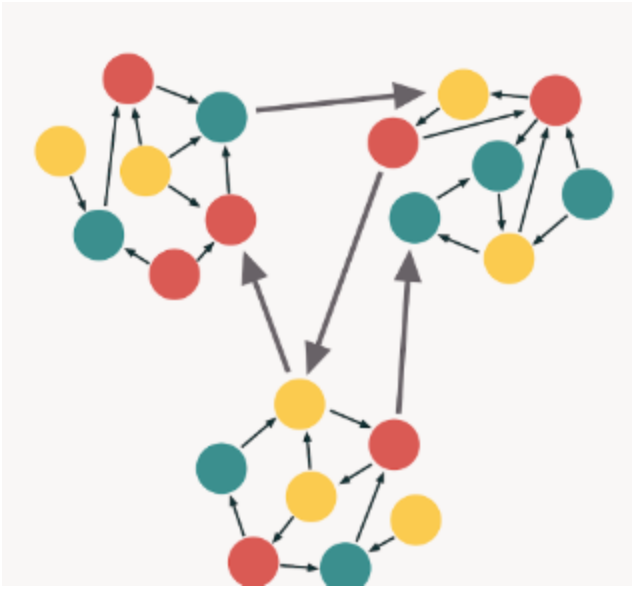
Low Coupling means separating unrelated parts of the code base as much as possible

Types of code from a cohesion and coupling perspective

Besides the code which is both highly cohesive and loosely coupled, there are at least three types that fall into other parts of the spectrum.



1. **Ideal** is the code that follows the guideline. It is loosely coupled and highly cohesive.
   a.
2. **God Object** is a result of introducing high cohesion and high coupling. It is an anti pattern and basically stands for a single piece of code that does all the work at once:
   a.
3. **Poorly Selected Boundaries** unlike the god object, code of this type does have boundaries. The problem here is that they are selected improperly and often do not reflect the actual semantics of the domain. Such code quite often violates the **single responsibility principle.**

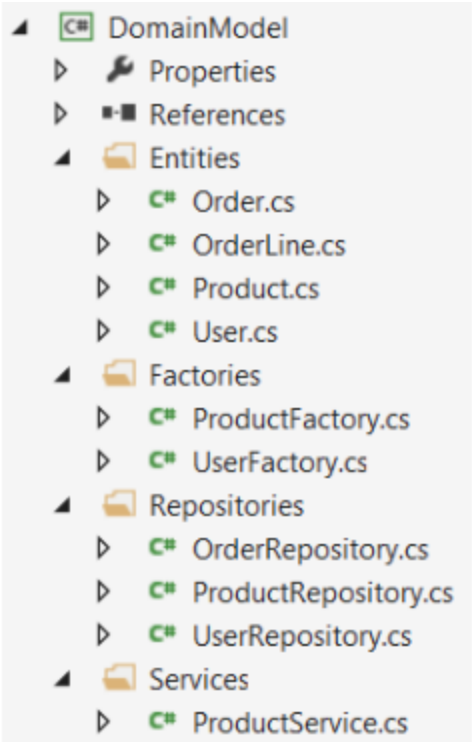
a. 

4. **Destructive Decoupling** sometimes occurs when a programmer tries to decouple a code base so much that the code completely loses its focus:

a.

Cohesion and coupling on different levels

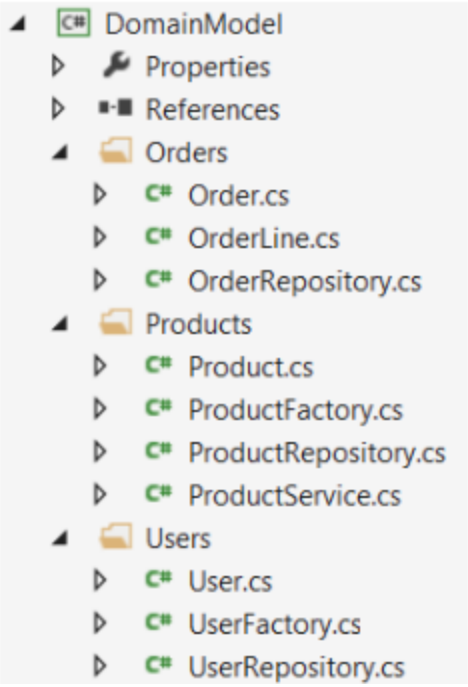
Cohesion and coupling can be applied on different levels. The class level is the most obvious, but its not the only one. An example here would be a folder structure inside a project:



At first glance, the project is well-organized: there are separate folders for entities, factories, and so on. However, it lacks cohesion.

It falls into the 3rd category in our diagram: poorly selected boundaries. While the internals of the project are indeed loosely coupled, their boundaries dont reflect their semantics.

A highly cohesive and loosely coupled version would be the following:



That way we keep the related classes together. Moreover, the folders in the project are now structured by the domain model semantics, not by the utility purpose.

Cohesion and SRP

The notion of cohesion is akin to the single responsibility principle. SRP states that a class should have a single responsibility, which is similar to what highly cohesive code does.

The difference here is that while high cohesion does imply code has similar responsibilities, it doesn't necessarily mean the code should have only one. SRP is more restrictive in that sense.

Required Reading: Beck Design Rules by Martin Fowler

Kent Beck came up with his four rules of simple design, which can be expressed as:

1. Passes the tests
2. Reveals intention
3. No duplication
4. Fewest elements

The rules are in priority order, so “passes the tests” takes priority over “reveals intention”

The most important of the rules is “passes the tests”. The point is that whatever else you do with software, the primary aim is that it works as intended and tests are there to ensure that happens.

The last rule tells us that anything that doesn't serve the three prior rules should be removed.

Week 6

Alternative to Inheritance

- Inheritance models “is a “ relationships
- We want to model **behaviors**
    - Language independent: Composition, aggregation, and delegation
    - Language dependent: Mixins

Composition, aggregation, and delegation

- Idea: Extract and model **behaviour** from an object.
    - Delegate responsibility of implementing behaviour to another object
- Composition ~= aggregation
    - **Composition**: Add behavior to a class by delegating responsibility of implementing that behavior to another class.
        - Embed many classes as instance variables
    - **Aggregation** Add behavior to a class by delegating responsibility of implementing that behavior to another class.
        - Embed many classes as instance variables
- What is the difference?
    - Composition implies *ownership* of instance variables
        - When owning object is destroyed, embedded object is also destroyed
    - Aggregation does not imply ownership of instance variables
        - When object is destroyed, embedded objects are not destroyed
    - **Example: Modeling Car Ownership**
        - Say Rob disappears
            - Rob has a car
                - Does his car disappear too?
                - No
                - Car could be owned by many people
                - When rob disappears the car will remain
                - What about insurance?
                - When rob disappears the insurance will be destroyed

Mixins

- A class with methods that can be injected into other classes.
- Idea: Get the behavior of a parent class without the “is a “ relationship
- Limited to language support

Cross-Cutting Concerns

What about functionality that we want executed before, after, or during method execution.

- Logging
- Validation
- Security / Authentication

This is called aspect oriented programming

Aspects

Aspects add functionality without modifying code.

- Define a “pointcut” - a description/pattern of which code should have that functionality added.

Software Design Patterns

- Composition and AOP are design patterns
    - A reusable solution for solving a problem
    - Patterns for code reuse without inheritance

Required Reading: Composition over Inheritance by Mattias Petter

Inheritance Example:

```
Robot
 .drive()

    CleaningRobot
     .clean()

    MurderRobot
     .kill()

Animal
 .poop()

    Dog
     .bark()

    Cat
     .meow()
```

Say that a project manager now needs a MurderRobotDog. It needs to be able to .kill(), .drive(), .bark(), but it cannot poop().

Now we’re screwed. We simply cannot fit the MurderRobotDog nicely into this inheritance hierarchy. We could create a new parent object, where you put all the functionality that is shared:

```
GameObject
  .bark()

    Robot
     .drive()

        CleaningRobot
          .clean()

        MurderRobot
          .kill()

          MurderRobotDog

    Animal
     .poop()

        Dog

        Cat
         .meow()
```

But that means that your objects will have a ton of functionality that they dont use, so you end up with a gorilla/banana problem – you request a banana, but you end up with a gorilla holding the banana and the entire jungle with it.

The other bad route that you can take is to duplicate the functionality:

```
Robot
 .drive()

    CleaningRobot
      .clean()

    MurderRobot
      .kill()

      MurderRobotDog
        .bark()

Animal
 .poop()

    Dog
      .bark()

    Cat
     .meow()
```

Time to use composition. Again, Inheritance is when you design your types around **what they are**, and **composition is when you design types around what they do**.

So using composition, we would do this instead:

```
dog            = pooper + barker
cat            = pooper + meower
cleaningRobot  = driver + cleaner
murderRobot    = driver + killer
murderRobotDog = driver + killer + barker
```

To do this using javascript. We are going to use normal functions and normal object literals.

```
const barker = (state) => ({
  bark: () => console.log('Woof, I am ' + state.name)
})

const driver = (state) => ({
  drive: () => state.position = state.position + state.speed
})
```

These are the definitions of barker and driver.

```
const murderRobotDog = (name)  => {
  let state = {
    name,
    speed: 100,
    position: 0
  }
  return Object.assign(
        {},
        barker(state),
        driver(state),
        killer(state)
    )
}

const bruno =  murderRobotDog('bruno')
bruno.bark() // "Woof, I am Bruno"
```

So here we have the murderRobotDog factory function. First it creates a state object. It assigns some default values to it – speed and position, and also assigns the name from the function argument. It then uses something called Object.assign – this is a function that is built into ES6, but there are equivalent methods in all javascript libraries. What it does is that it takes an object, in this case a new, empty object, and assigns the properties from other objects to it.

The really big problem with inheritance is that you're encouraged to predict the future. Inheritance encourages you to build this taxonomy of objects very early on in your project, and you are most likely going to make design mistakes doing that, because humans cannot predict the future, and getting out of these inheritance taxonomies is a lot harder than getting out of them.