

# Game Production Practice

DR. ROBERT ZUBEK, SOMASIM LLC

EECS-397/497: GAME DEVELOPMENT STUDIO

WINTER QUARTER 2018

NORTHWESTERN UNIVERSITY



NORTHWESTERN  
UNIVERSITY

# AAA games are expensive to make

---

Let's pick on "reasonable AAA" titles

1995	Twisted Metal	\$ 800 k
1998	Thief	\$ 3 M
2001	Black & White	\$ 5.7 M
2004	CoD: Finest Hour	\$ 8.5 M
2006	Gears of War	\$ 10 M
2007	Bioshock	\$ 15 M
2010	God of War III	\$ 44 M
2012	Borderlands 2	\$ 35 M
2014	Watch Dogs	\$ 68 M
2015	Witcher 3	\$ 81 M

Even if 1/2 of that is marketing etc.  
that's a lot of money.

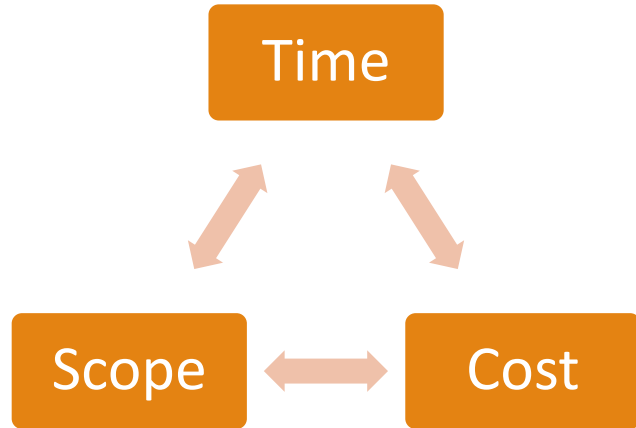
Cost doubles every ~5 years

→ Amount of work doubles every ~5 years

Teams of 100+ people not uncommon

# The project planning triangle

---



“What can you commit to?”  
“Time, scope, or cost: pick two.”

## Time

- When you have to be done

## Cost

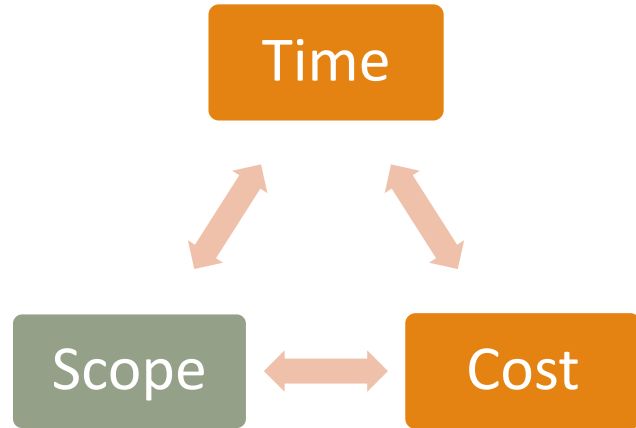
- Usually function of team size + external services

## Scope

- Which features / how many / what quality level

# The project planning triangle

---



“What can you commit to?”  
“Time, scope, or cost: pick two.”

## Time

- Shipping date usually fixed

## Cost

- Team size capped based on budget

## Scope

- This is the most flexible part!

(Sometimes people try to cheat: increase scope, and then introduce crunch to hide increased cost)

# So how do we plan?

---



# Plan out nothing?

---

1. Write some stuff
2. Oops, it didn't work
3. Fix it
4. Actually we needed different code
5. Rewrite old code
6. Go to step 2
7. Maybe write more stuff
8. Go to step 2

No methodology:

- Minimal planning
- Incremental implementation

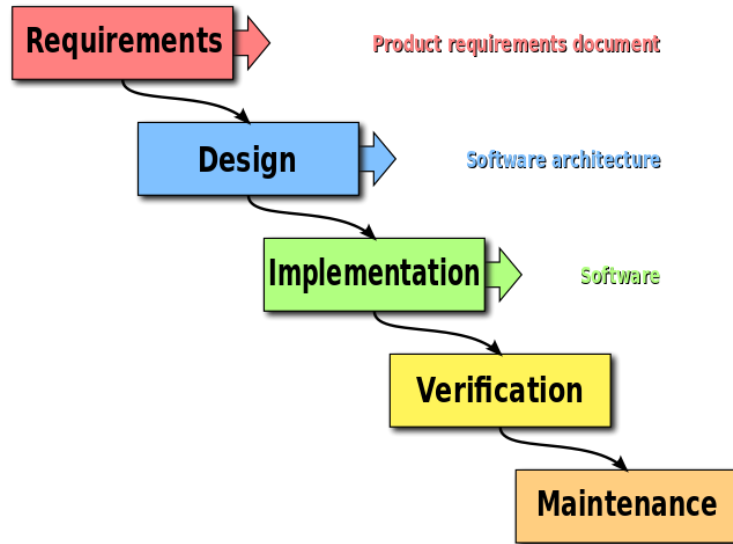
Problems

- Doesn't scale beyond a tiny team (and even then...)
- Wasted effort: a lot of previous code gets redone over time, as you discover missing key capabilities

... that said, that's how people often learn new things – by experimenting without a set plan

# Plan out everything?

---



## “Waterfall” model

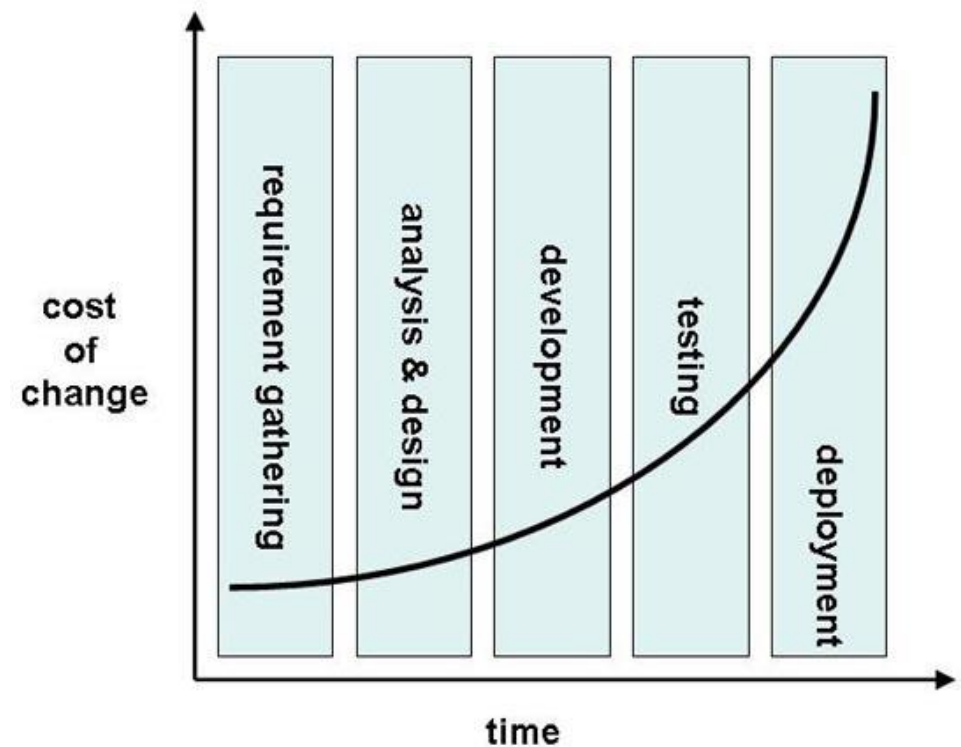
- First figure out what the product needs to be
- Then design the software architecture on paper
- Then code everything up, test, ship, & profit!

# What makes waterfall appealing?

---

Cost of changes increases over time

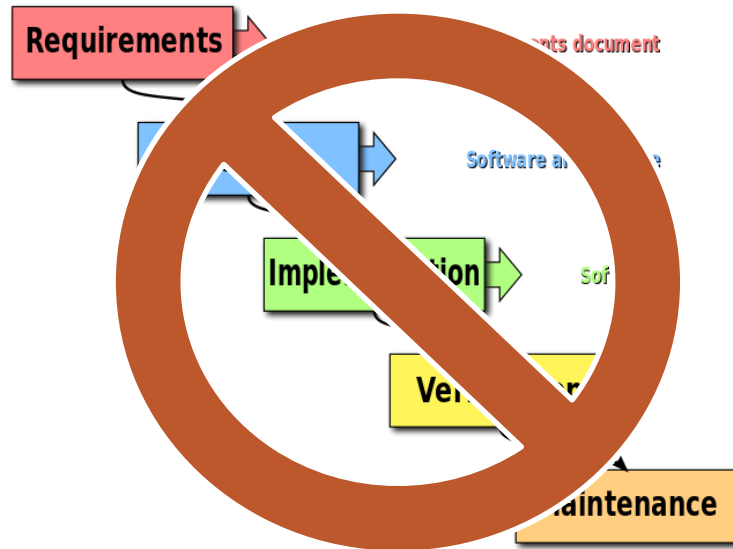
Front-load all decisions!





# Plan out everything?

---



Waterfall is a “strawman” model.  
Actual designers don’t work like this.

- We don’t know what the goal is when we start
- We don’t know the decision space
  - It’s too large: we discover it as we make decisions
- We can’t evaluate individual decisions, only complete designs that bring them all together
  - Experience matters: prunes entire branches off of the tree
- Desiderata can change
- Constraints can change

# Solution: staged “spiral” methodology

Split production into stages

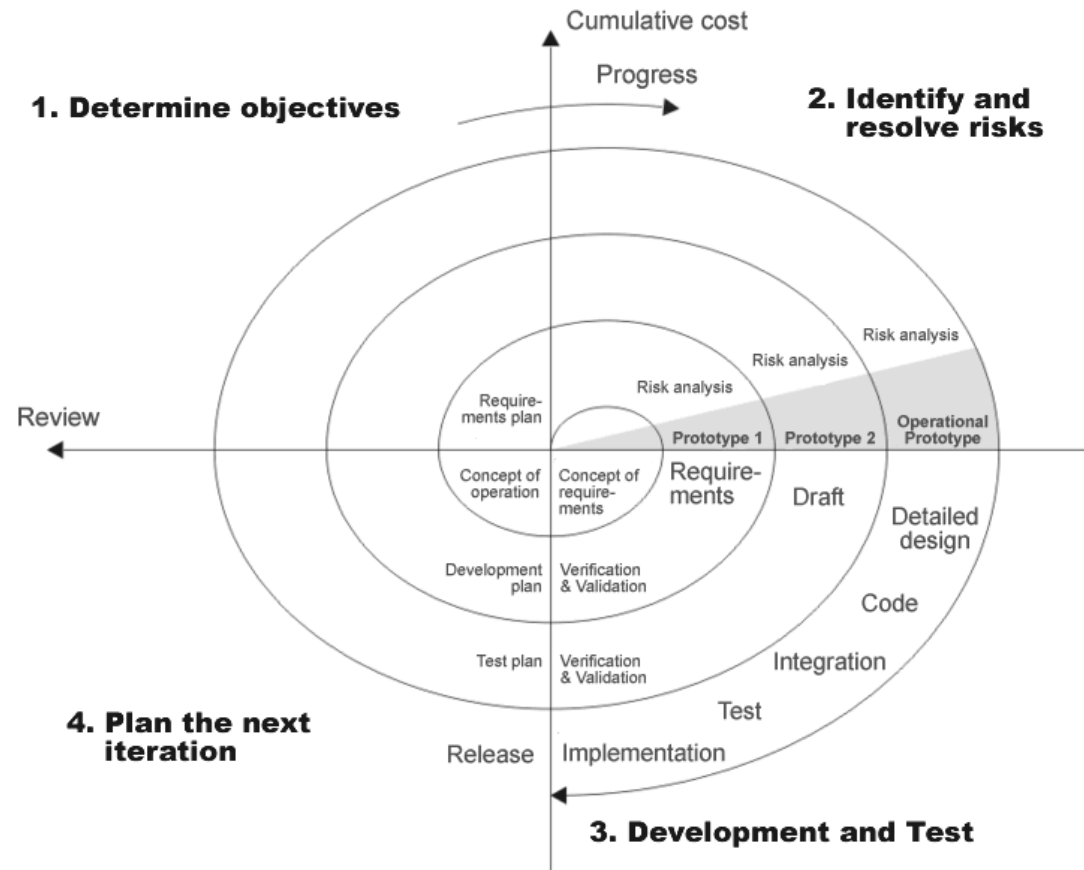
Start out with a blank slate and a tiny team

At intermediate stages:

- Figure out more about the game
- Decrease decision space
- Increase resources

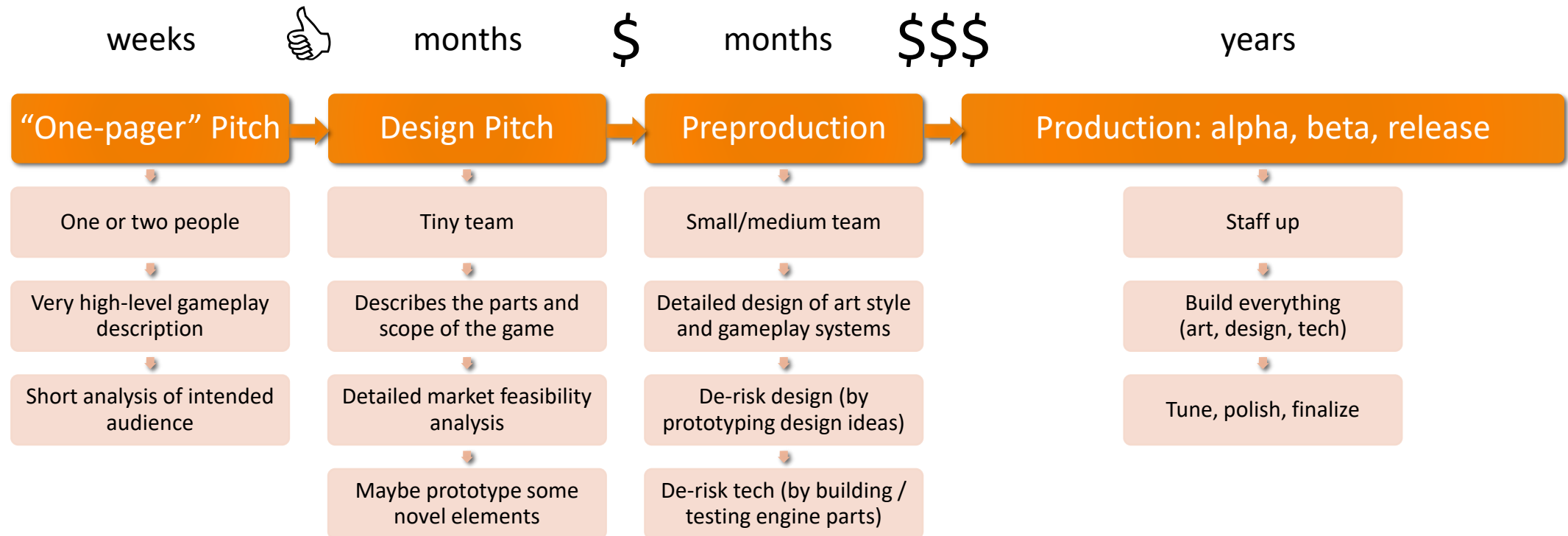
Final stage:

- Decide what the game is in detail
- Fund it and build it



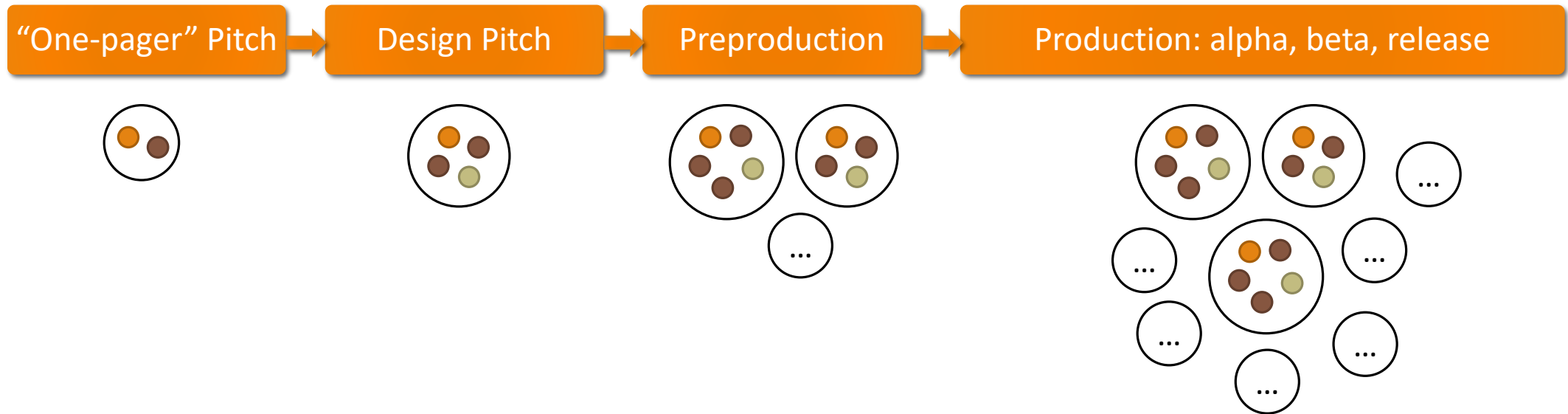
Boehm's Spiral Model

# AAA production process



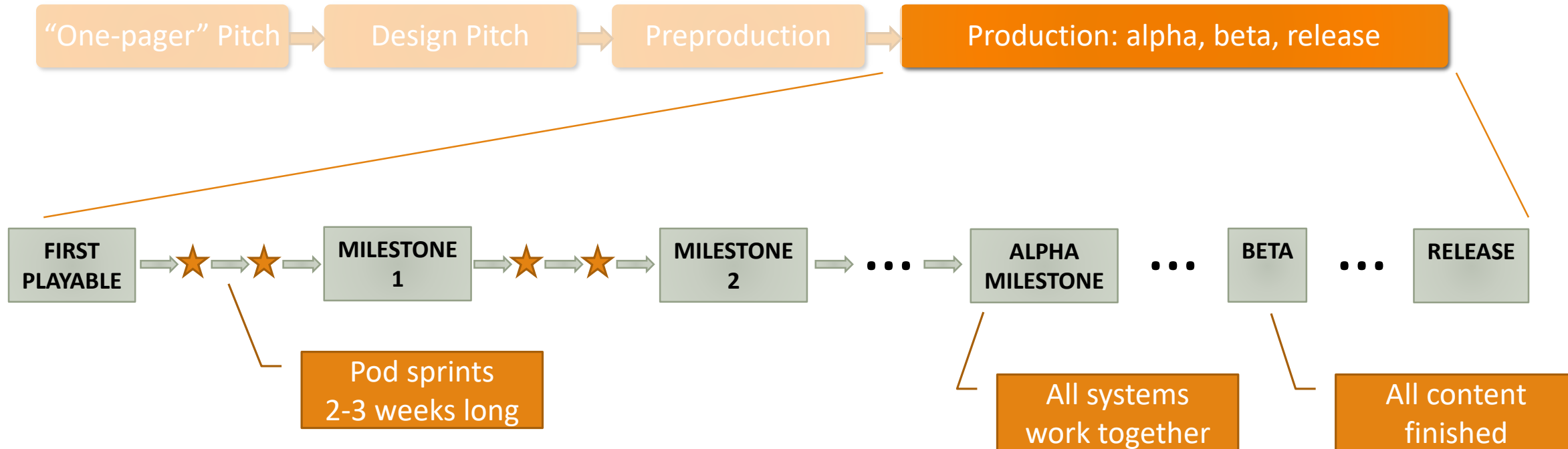
# AAA production process

---



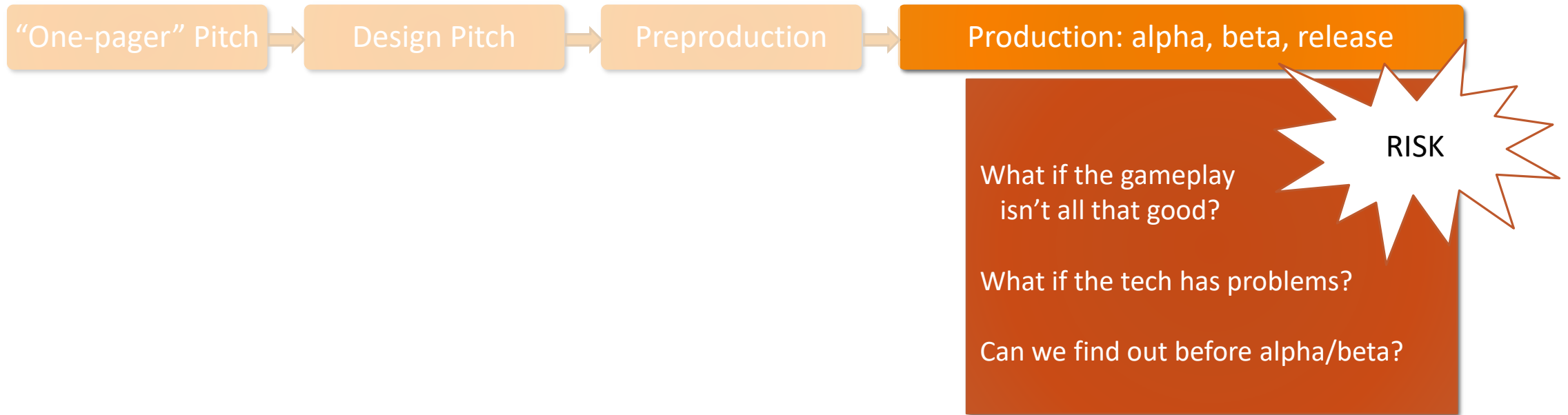
# AAA production process

---



# Quick aside on risk management

---

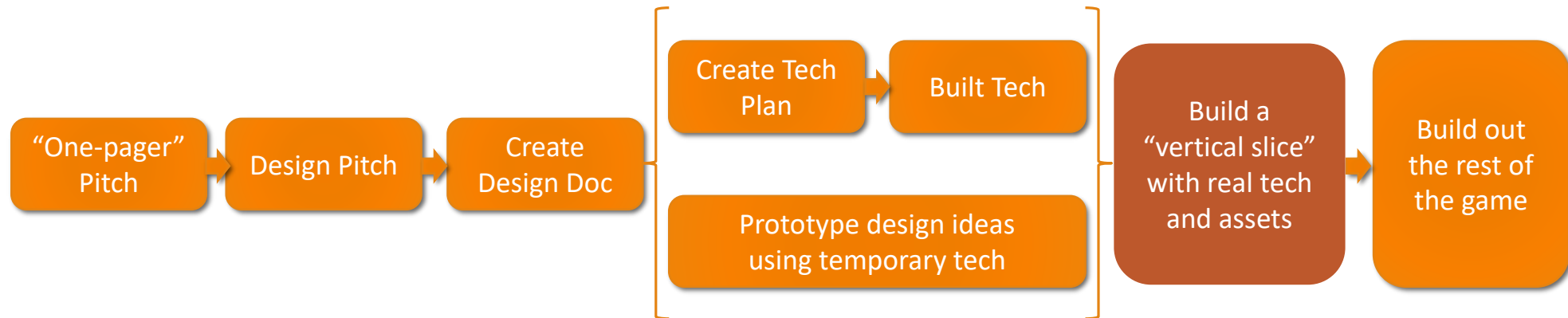


# “Cerny Method” for AAA

---

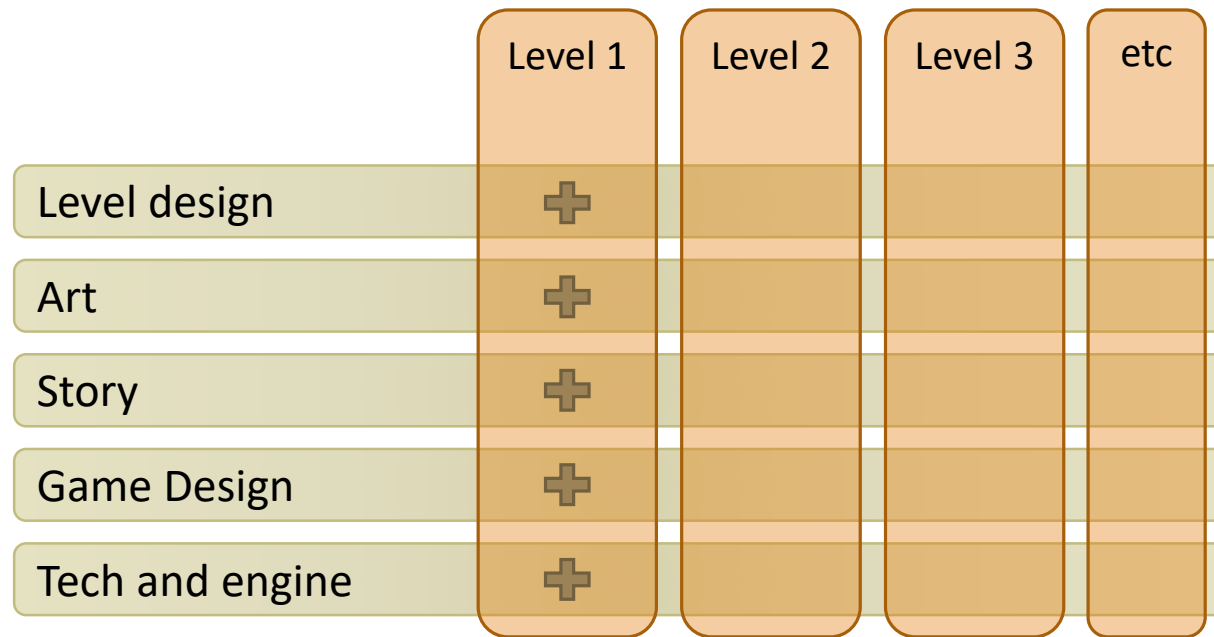
Goal: validate that all parts work together as a game, ASAP

“Vertical slice” – build a shippable-quality level of the game



# Vertical slice

---



## Idea:

- Build a shippable-quality level that shows off all major parts
- Quickly iterate on any failures
- Fund production only if the vertical slice is good enough

## Problem:

- 1 level  $\sim$  10% of content, but  $\sim$ 70% of tech and design work!
- Only works for some games



# From AAA to smaller teams

---

Process for large games tends to be more rigid

- “Spiral” early on, but lock in as the team grows
- Optimizes **efficiency**, but limits **flexibility**

Small teams can be much more flexible

- Keep the iteration spiral going through development
- Adjust plans as you learn more about the game

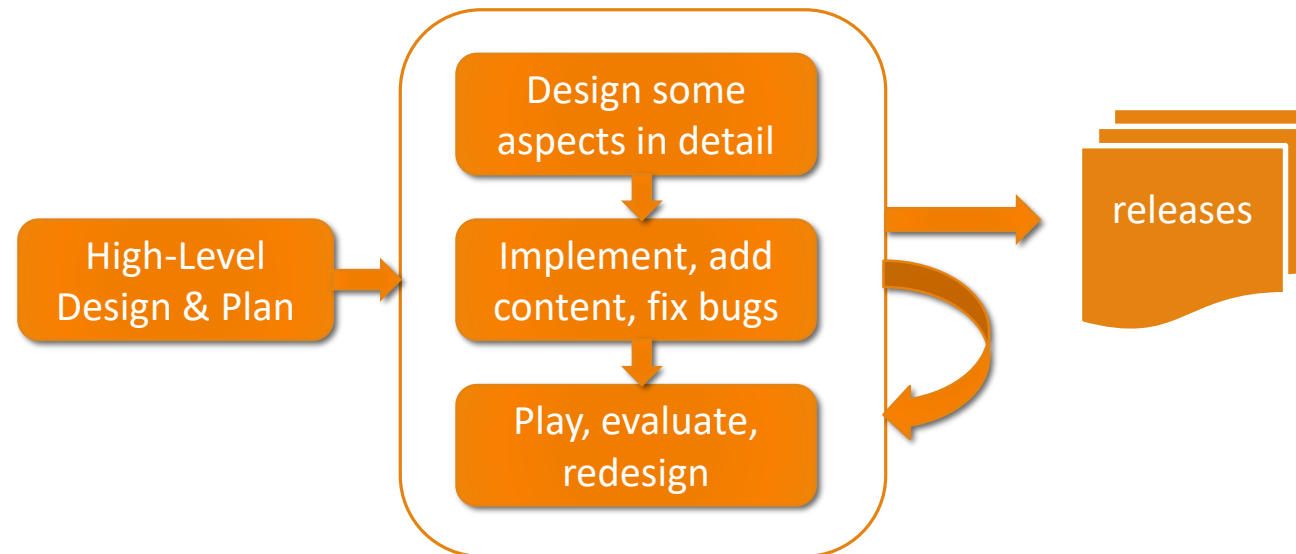
Both large and small teams use some variant of “Agile” – we’ll be using it too! 😊

# Small team / “indie” process

---

With a small team, you can get away with being more iterative

(Big teams would like that too, but it doesn't scale)



# Gamedev version of Agile

---

Project → multiple milestones

- Rough division of what needs to be done when

Milestone → multiple sprints

- Each sprint with a major theme

Sprint = predefined short period

- Sprint planning: team agrees what will be done
- Devs do their work, with daily check-ins
- Sprint ends on schedule. **No extensions!**



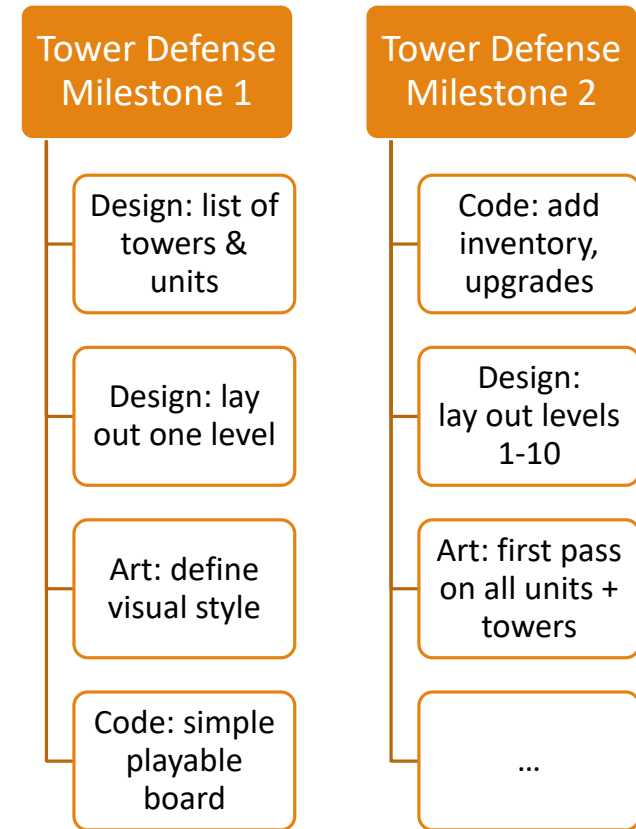
# High-level schedule

---

AKA “Backlog”

High level schedule contains all major pieces of the game

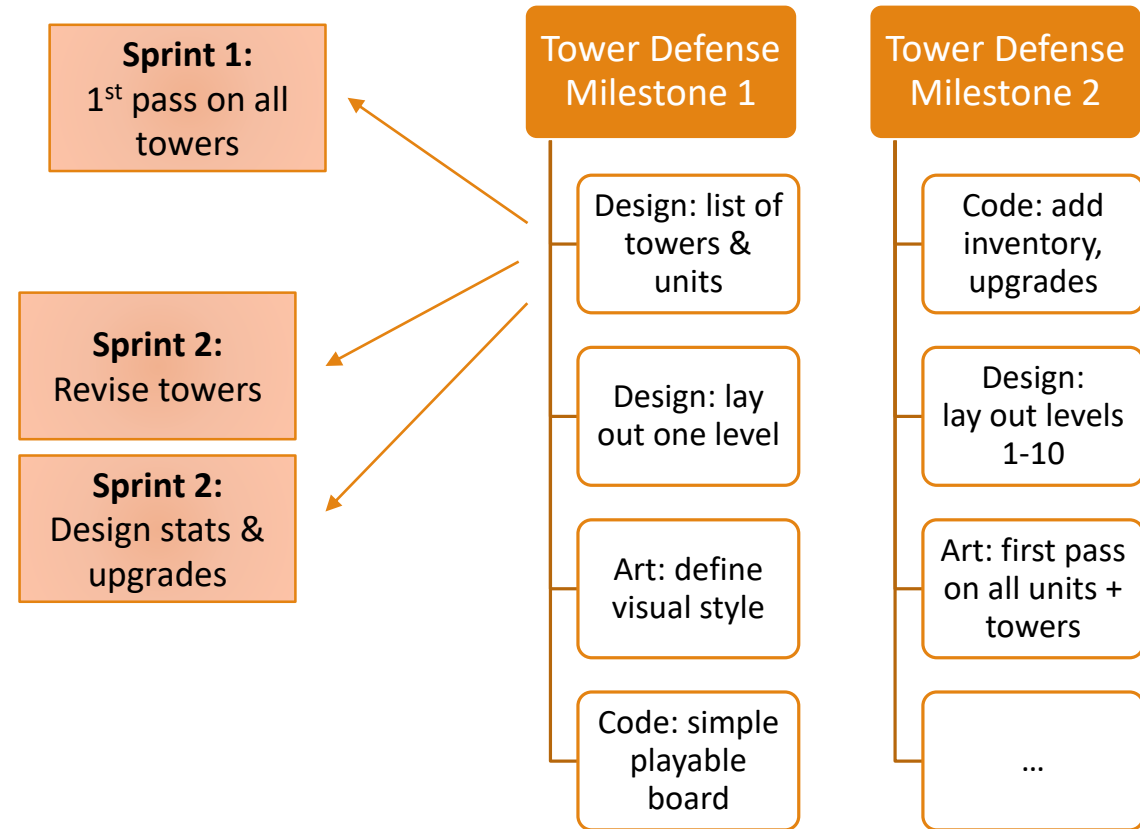
- Typically fairly coarse level of detail
  - So everyone needs to agree on what they mean :)
- Each item assigned to some milestone
- Scope is a guess (not enough detail yet)
  - Generously padded! Nothing ever works as expected.



# Product backlog → sprint backlog

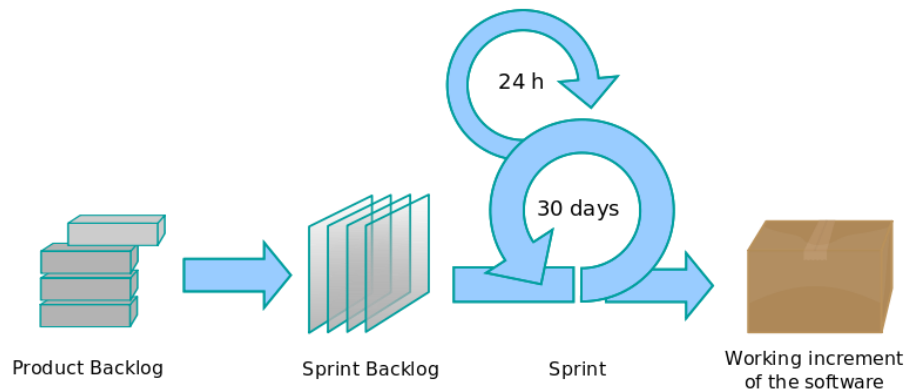
Start of milestone:

- Pick item from backlog
- Split into smaller pieces
- Assign priority, and assign to sprints



# Sprint

---



## Timeboxed

- 1-4 weeks

## Starts with sprint planning

- Pull items from the backlog
- Devs do work estimates and discuss in group
  - Make sure this fits into the sprint!
  - Otherwise split it up into 2 work items, revise backlog

## Ends with a new “product release”

- Features implemented to spec or not
- But not half-done or buggy
  - Don't merge it in unless it's done

# Sprint deliverables

---

## Commitment to Software Quality

Each sprint = software release

- Slowly growing the feature set
- No tech debt! “Shippable” code

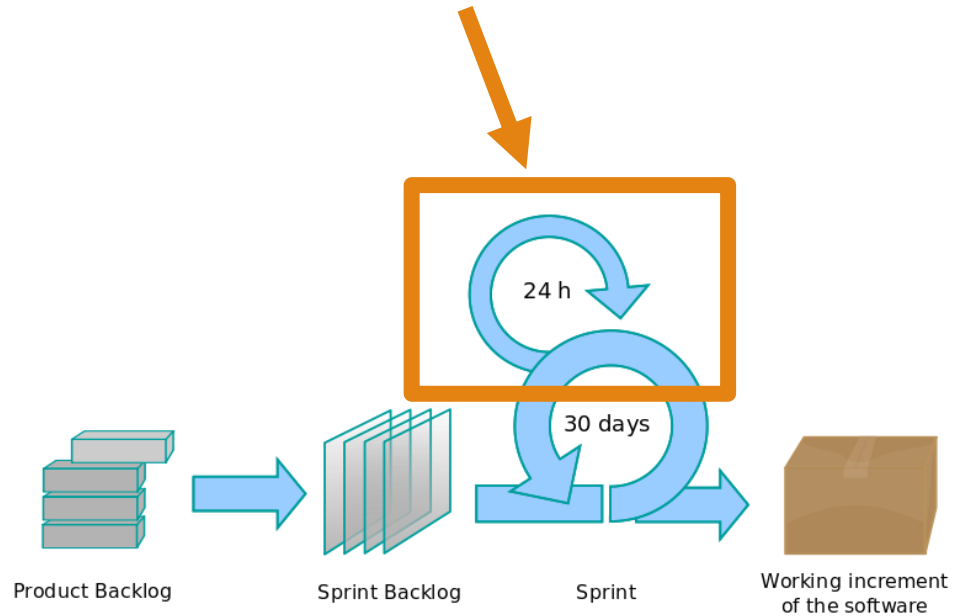
## Bugs

- Fix as you go, don't procrastinate
- Goal to have zero P1 bugs end of sprint



# Daily scrum

---



## Timeboxed daily meeting

- Short: ~15 minutes
- Starts on time whether you're there or not

## Every member states

- What you did yesterday
- What you're working on today
- What problems or blockers you're running into

## "Stand up" meeting

- Hold it in front of the task board
- Force people to stand, to keep it short :)
















# Task board

Tracks the status of features

- In backlog / to be done
- Being worked on
- Completed
- Blocked!

Shows at a glance

- What remains to be done
- Who is working on what
- How we're tracking towards goals

	TODO	IN PROGRESS	DONE	BLOCKED
Alice				
Bob				
Carol				
Dave				

# Task board

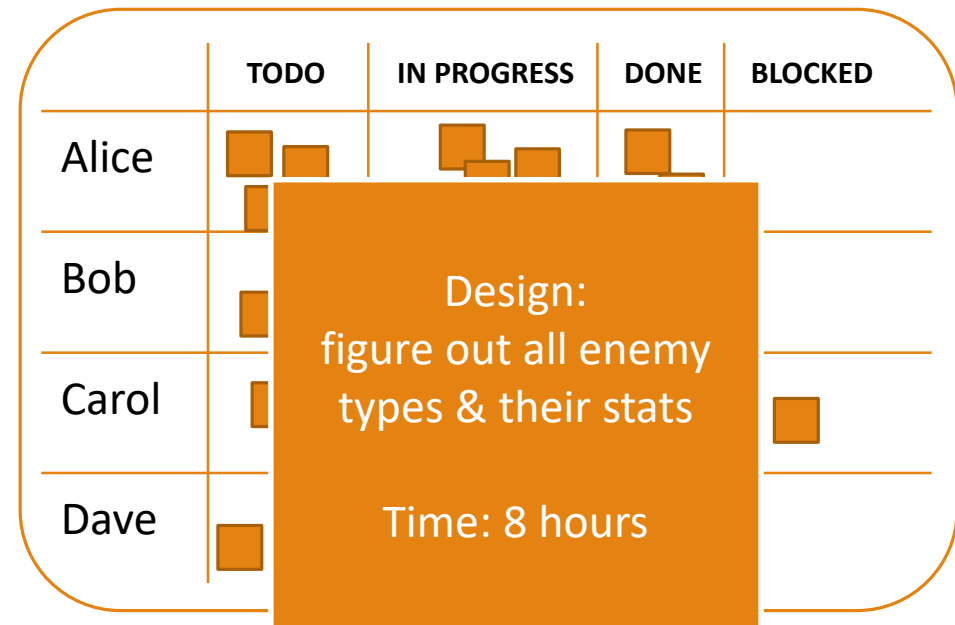
Ideally, each item has a time estimate

Used to see if it all fits into sprint schedule

- How many work hours assigned per dev
- If anyone is falling behind

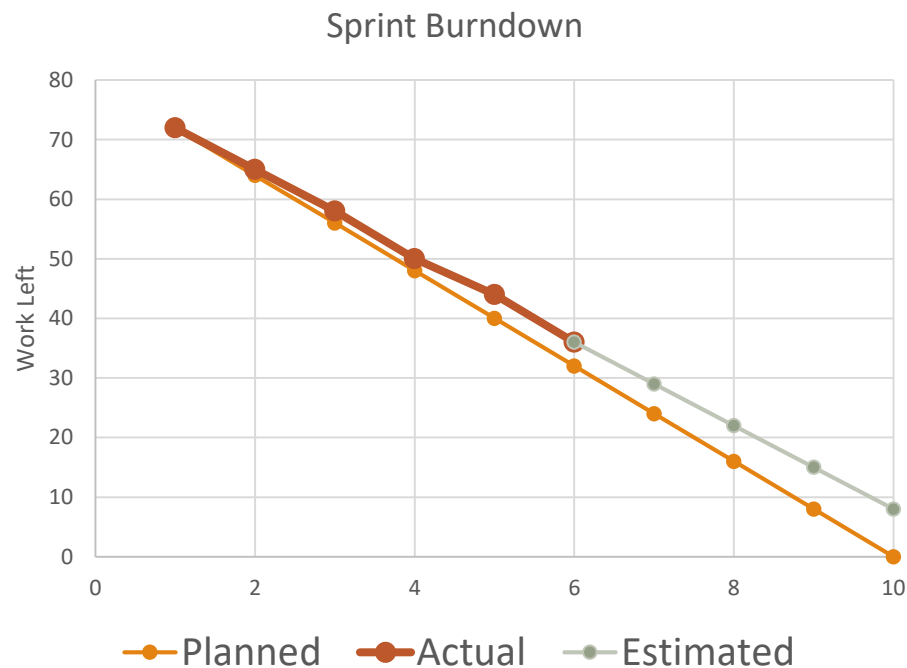
Typically done in software these days

- Trello
- JIRA
- etc.



# Burndown chart

---



Different way to track if you're on track

1. Starts with N hours of work left to do in D days
2. Every day should decrease it by  $N/D$  hours
3. Until you reach 0 hours left by end of sprint

Graph planned vs actual completed work to easily extrapolate to end of sprint

# Tests

---

## Automated

- Unit tests  
test each component in separation  
to make sure it hasn't changed
- Integration tests  
make sure different components  
work together as planned

Usually run by the build system

## Manual

- Smoke test  
load the game up, load up a level, make  
sure you can do the basics (run around)
- Full test  
QA prepares a checklist of everything that  
needs to be checked based on features

Usually done manually either by developers or  
by dedicated QA, when merging to mainline

# Build systems

---

Small projects: rebuild after each commit

Large projects: rebuild nightly

What's in a build:

- Check out latest version of the project
- Compile all source code for all platforms
- Compile all assets for all platforms
- Run automated tests
- Send hate mail if anything fails



# Jenkins

(...or some other build server)

# Builds

---

Typically two types of builds:

## Debug build

- Used by devs for testing
- Asserts enabled
- Optimizations optional
- Debugging enabled



Debug build is slower,  
but does error checking and reporting

## Release build

- Used by QA, artists – “final” build
- Asserts disabled
- All optimizations applied
- No debug symbols

Release build is optimized for performance  
and hides any errors from the player

(Might send error reports back to mothership)

# Source control

---

## Perforce at large studios

- It's expensive
- Good workflow for large teams

## Git at smaller studios/teams

- It's free! And quite good
- Mostly optimized for text files, though
- Good luck with binary assets

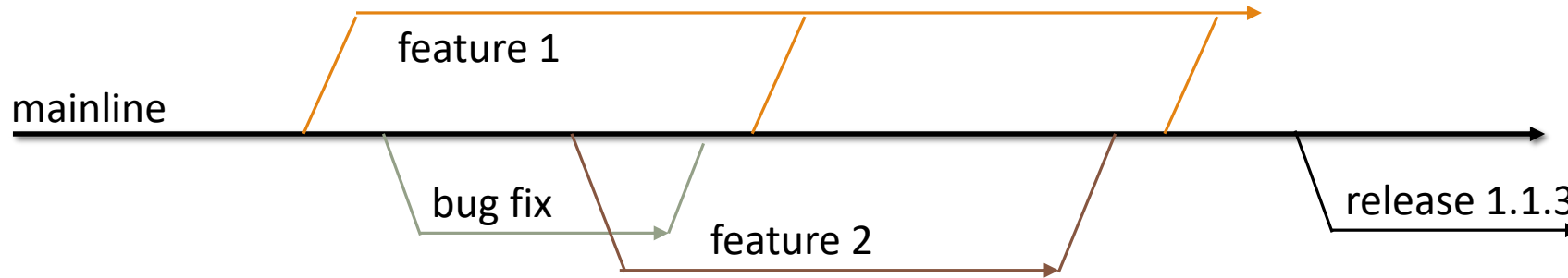
SVN and others rarely used anymore

## Some typical problems:

- Uploading / downloading large binary files
  - Git downloads a complete history unless you tweak it
- Binary assets are not merge-able
  - What if two artists work on the same texture or character model at the same time?
  - Perforce supports team-wide file locking
  - Git laughs at you and says "tough luck, sucker!"

# Source branching

---



## Types of branches:

- Mainline
- Large feature branches
- Small feature / bugfix branches
- Release branches (or tags or workspaces)

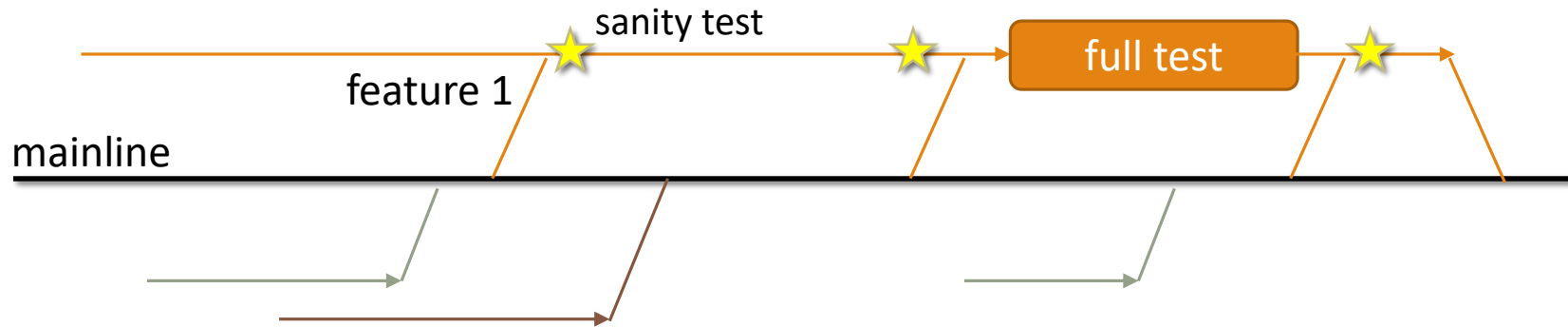
## Branching allows devs to work in parallel

- How easy or hard this is depends hugely on the source control system



# Source branching

---



Failures usually happen at merge time

Common failure types:

- Code merge
- Binary asset merge (this is a bad one)
- Conflicting architecture or functionality

After merging from mainline, do a sanity test

Before merging back to mainline

- Run a full test (might involve QA)
- If mainline changes in the meantime, do one more merge and a quick sanity test

# Releases

---

## Packaged game

- Alpha, beta, RC1, RC2, ..., gold master, DONE!!!
- The world isn't like that anymore :)

## Standalone game (eg. Steam)

- Alpha, beta, 1.0 RC1, 1.0 RC2, ..., 1.0 final
- Patches: 1.0.1, 1.0.2
- Enhancements: 1.1, 1.2
- Ensuring update safety is a hassle
  - What if someone upgrades from 1.0 straight to 1.99?
  - Will their save files still work? How can you be sure?

## Mobile game

- At least Steam forces updates automatically
- On mobile, people can refuse to update
- This sucks if you have server-side logic
  - What if the client is running 1.0 from two years ago?
  - Detect and force them to update? Or try to be compatible?

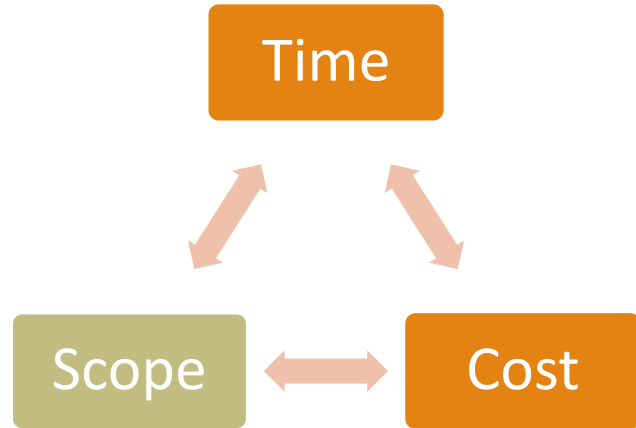
... a game's test and release plan grows increasingly complex as it gets older, and as it gets released on more platforms

Now let's talk about \*your\* projects...

---

# Your class project

---



## Time

- Deadline: end of quarter

## Cost

- Your team size is set

## Scope

- That's up to you!

# How we'll run the project

---

## 1-week sprints

### Standups

- Twice a week at least
- Before / after class, or however you arrange

### Weekly sprint reports (one pagers)

- Result of last sprint
- Goals for next sprint
- Backlog for next sprint with effort estimates
- Dates of standups + any notes
- Due each week EOD Sunday

## Tools

### Task board

- Use either Google Docs / Spreadsheets, or Trello

### Source control

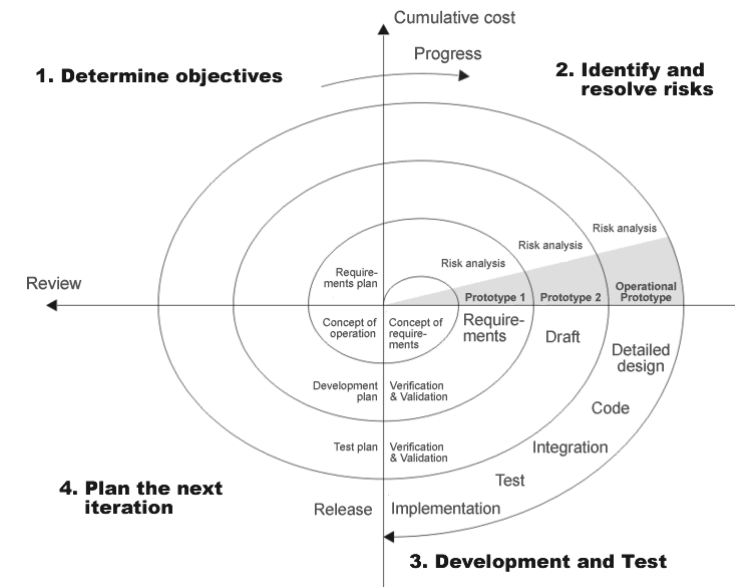
- Up to you
- BitBucket is good and free for small teams
- GitHub is also popular (new academic pricing?)

# Weekly sprints and playtests

Each week you will:

- Do a sprint with some specific goals
- Playtest whatever you're building in class
  - Get feedback from others
  - Give feedback on their work
- Integrate feedback into your planning for next sprint

→ Useful tactic: find the fun, and follow it



# The perils of prototypes

---

Beware the “it’s **only a prototype**” rathole

- It’s common to cut corners
- And say “we’ll fix it for the final version”

A project’s **technical debt** is the sum of all those little fixes you have to do before you can ship

- All the workarounds and other extra effort you have to do because of it is like paying interest on the debt

“The **danger** occurs when the debt is **not repaid**. Every minute spent on not-quite-right code counts as **interest** on that debt. Entire engineering organizations can be brought to a stand-still under the **debt load** of an unconsolidated implementation”

Ward Cunningham (1992)

# Example

---

You do a quick prototype of the editor for your game

And so you don't put in checks for invalid inputs

- It's just a prototype, right?
- You'll fix it later – **technical debt**

You still have to fix it sooner or later

- **Repay the debt**

In the meantime, it periodically crashes on your designers

- Losing their work – **interest on the debt**

The longer you wait to repay the debt, the more interest your organization pays

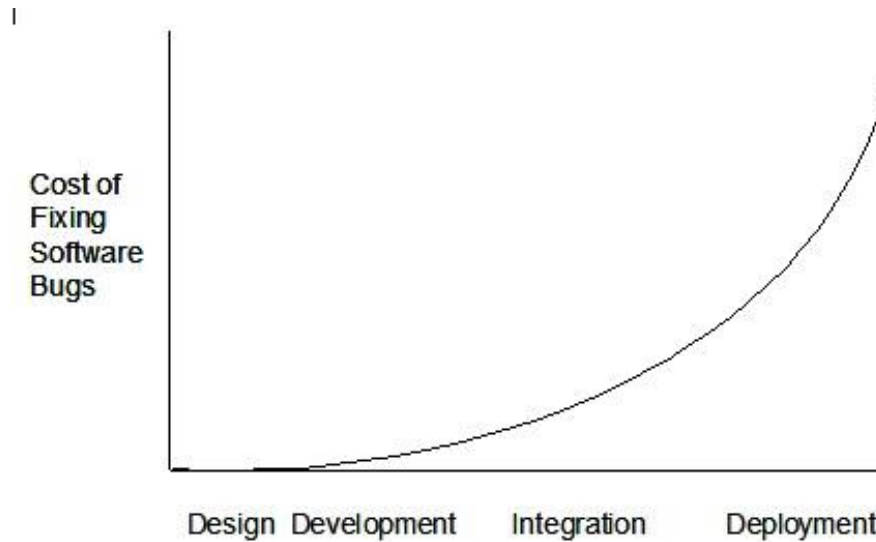


# Plus...

---

Remember about **late design changes**

The later they are, the **more expensive** they are



# Homework

---

Submit a high level project plan (details on next slide)

This is **not a binding estimate!**

It's a planning exercise for you,  
so that you know if your game is over-scoped.



# Homework

---

Submit a high level project plan (one per team)

- Assume 6 weekly sprints starting this past Monday
- Describe the team's goals for each sprint
  - What do you want to have done at the end of each week?
  - When will you have something playable to iterate on?
- Verify: do these steps add up to what you described in your project proposal?
  - If not, that's okay. Describe how you're revising your project.

Due EOD Sunday 2/4



# Afterwards: weekly status updates

---

These will start next week

One pager that states:

1. Result of last sprint
2. Dates of last sprint standups
3. Playtesting results (triaged feedback list)
4. Task items for next sprint with effort estimates

We'll be doing these every week.

Qs?

---