17-723: Designing Large-scale Software Systems

Software Design Process

Tobias Dürschmid

Carnegie Mellon University

This Lecture

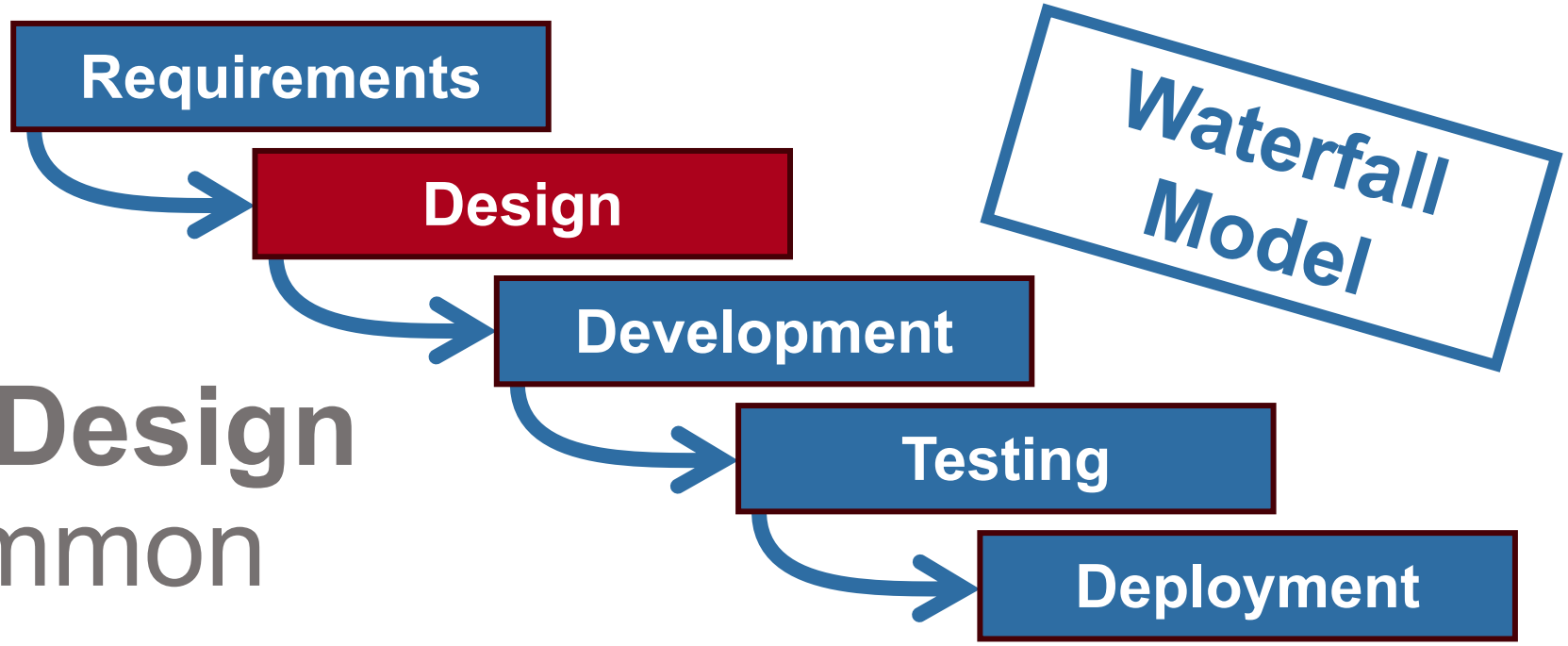
- How to Design in Agile Projects?
- How to Consider the Human Aspect of Software Design?
- How to Adjust the Design Process To Domain-Specific Needs?

Carnegie Mellon University

How to **Design in Agile Projects?**

Designing Large-scale Software Systems – Software Design Process

Carnegie Mellon University



In the Past **Big Upfront Design** was very Common

- ☹ Requirements might **change** during development
- ☹ Customer **feedback** is considered very **late** in the process
- ☹ Projects were **delayed** very often

Carnegie Mellon University

What implications does this have on software design?
What role should software design play in agile projects?

Alternative to Waterfall: **Agile Manifesto**

2001

We are uncovering better ways of developing software by doing it and helping others do it. Through this work **we have come to value**:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is **value in the items on the right**,
we **value the items on the left more**.

See https://agilemanifesto.org/

Carnegie Mellon University

Design-related Principles behind the Agile Manifesto

Working software is the primary measure of progress.
[…]
Continuous attention to technical excellence and **good design enhances agility**.
[…]
The best architectures, requirements, and designs emerge from **self-organizing teams**.

The quality of the design only matters if it is **observable**

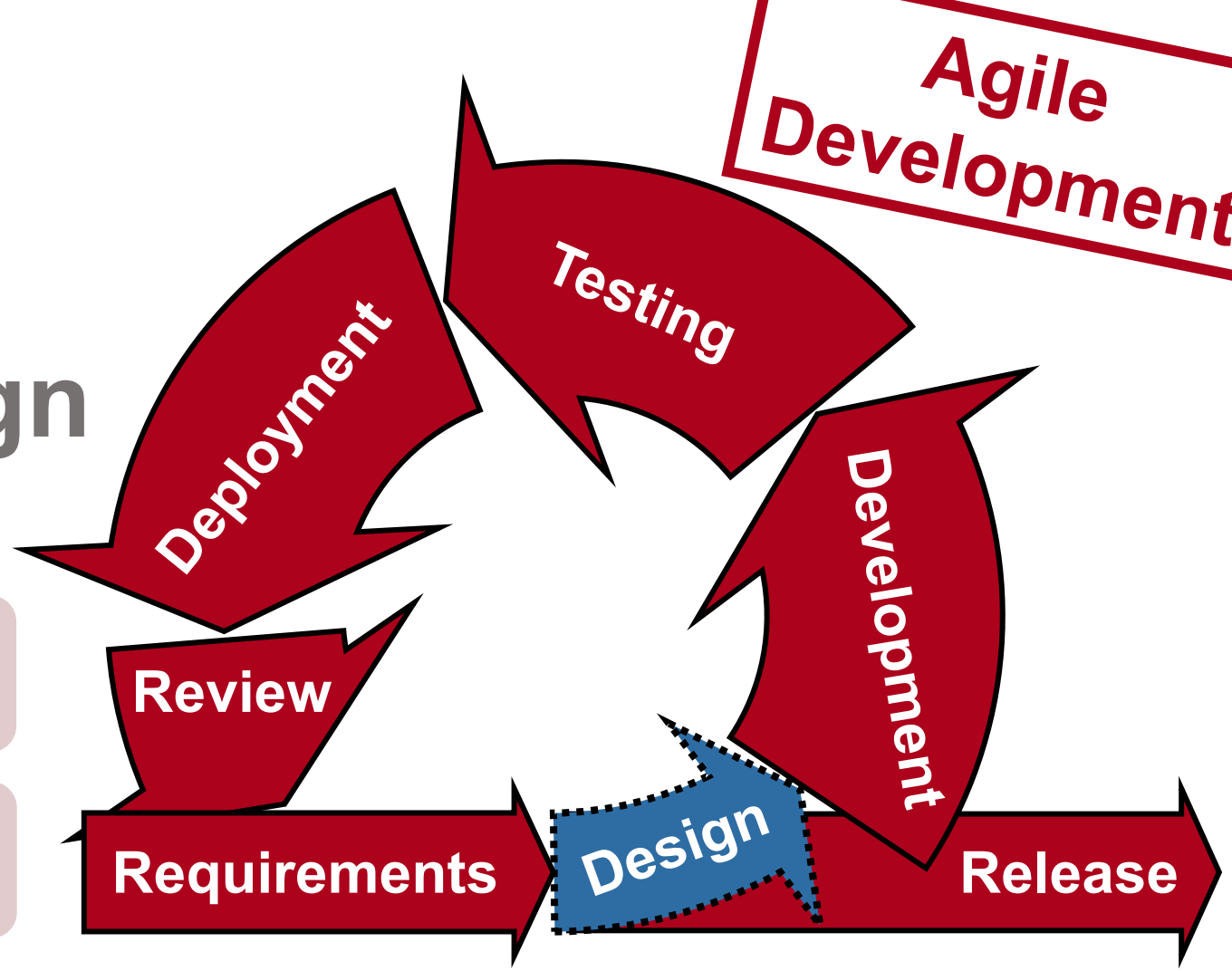
Design is not an initial phase but part of **every iteration**

There is no single architect or top-down design

See https://agilemanifesto.org/principles.html

Carnegie Mellon University

Today **No** / **Tiny Upfront Design** Is Common

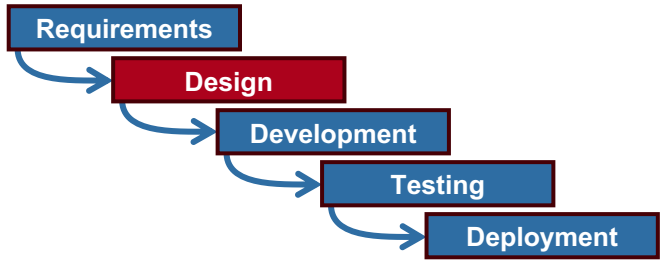


- ☹ Software is **hard to change**
- ☹ **Improving Quality Attributes** is hard
- ☹ Small **Bus Factor** (i.e., number of people who can leave a project (“hit by a bus”) before the project stalls. Measures shared knowledge & documentation)

Read more here: Dikert, Kim, Maria Paasivaara, and Casper Lassenius. "Challenges and success factors for large-scale agile transformations: A systematic literature review." *Journal of Systems and Software* (2016)

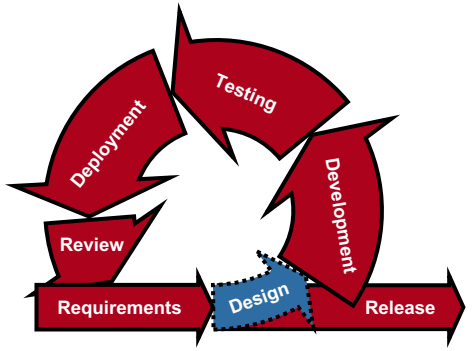
Carnegie Mellon University

What should we do instead?



Big Upfront Design

Risk-Driven Design



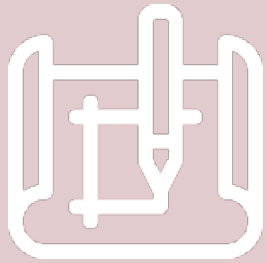
Tiny Upfront Design

Carnegie Mellon University

Risk-Driven Design

Identify **biggest risks** of the software and focus design on these risks.

The amount of risk involved in the project determines the **amount of upfront design.**



Read more here: Fairbanks, George. *Just enough software architecture: a risk-driven approach*. Marshall & Brainerd, 2010.

Carnegie Mellon University

Risks are Decisions that are **hard to change**

Example Risks

- Programming Languages
- Target Platforms
- Component Architectures & Connectors
- Interfaces
- Quality Attributes

Carnegie Mellon University

What Risks are Most Important in These Domains?

Security **Privacy** **Changeability**

Online Shops

Platforms **Usability** **Performance**

Games

Reliability **Robustness** **Testability**

Medical Software

Collaborative Risk Identification Technique: **Risk Storming**

Step 1: Model

Model your software design as diagrams

Step 2: Think

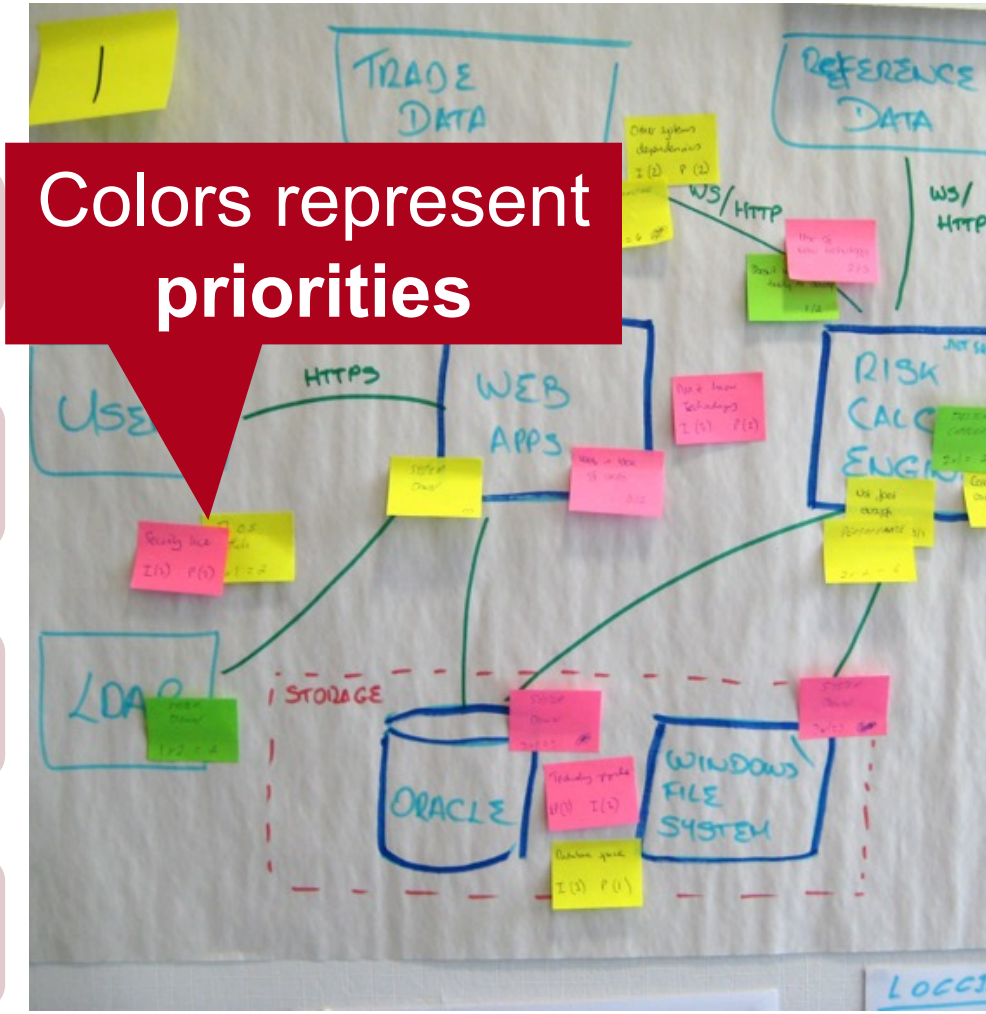
Identify the risks silently on post-its

Step 3: Share

Add post-its to the diagram

Step 4: Review

Discuss risks and summarize



Read more here: https://riskstorming.com/

by Simon Brown

Carnegie Mellon University

Identify and Mitigate Highest-Priority Risks

- Make big design decisions early
- Defer all small-scale decisions until later.
- High **cost of change** → **upfront design**
- Low cost of change → lean design

Carnegie Mellon University

> Responding to change over following a plan
> – Agile Manifesto

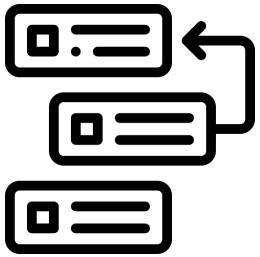
Changeability in Agile Projects

- A good architecture allows you to **make decisions late**
- A good architect maximizes the number of decisions **not made**
  - Information Hiding
  - SOLID Principles
  - Low Coupling
  - High Cohesion
  - Separate A software from T software

Design for Change

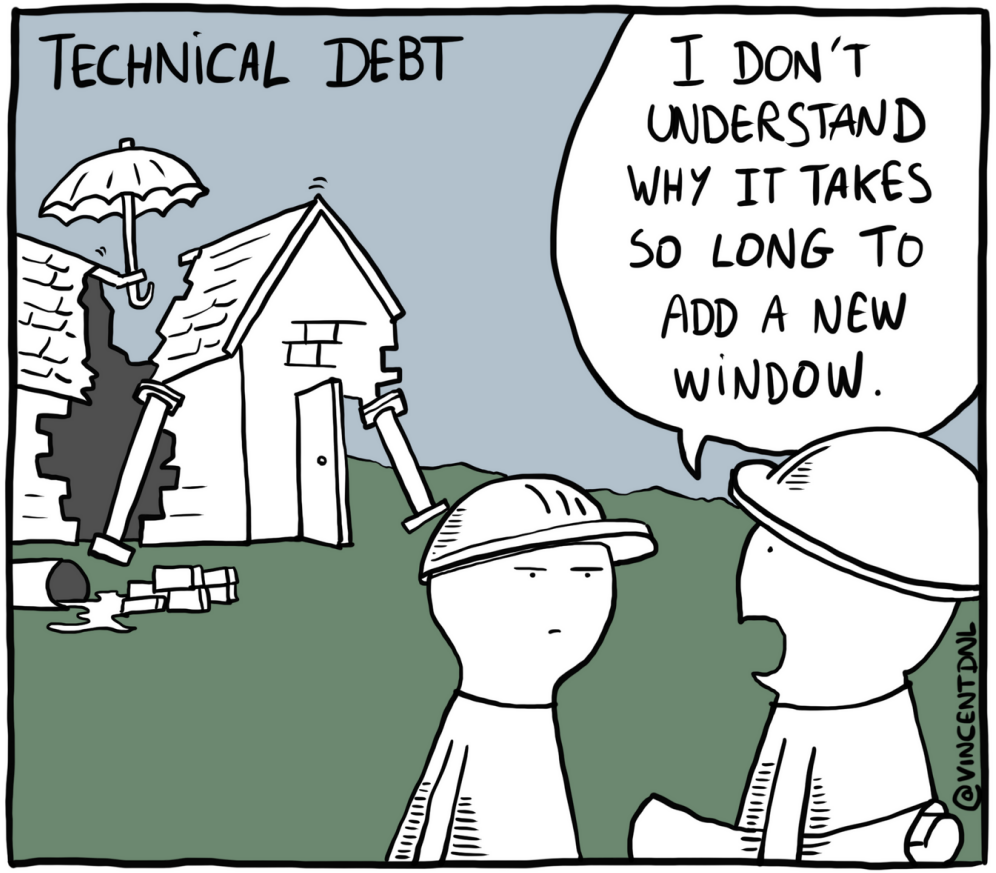
Carnegie Mellon University

Technical debt is the result of short-term-oriented decisions that make future changes more costly or impractical.



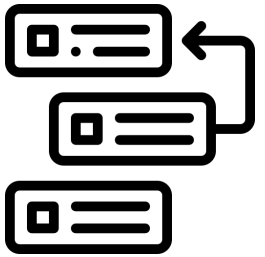
Feature Backlog vs Technical Debt Backlog

- User stories capture functional requirements in the feature backlog
- Maintain **a technical debt backlog** with issues **that improve software design** by refactoring & building abstractions



Read more here: https://agilewaters.com/technical-debt-and-product-backlog/

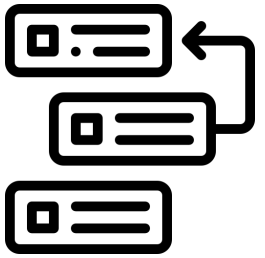
Carnegie Mellon University



Examples of Technical Dept Issues

- Fix **code smells** (e.g., duplicate code, high coupling, low cohesion, complex interfaces, …)
- Improve **documentation**
- **Architectural changes** to support performance, scalability, …

Carnegie Mellon University



Integrating Technical Dept Issues in Agile Processes

- Having a special role of an **architect** who **maintains the technical debt backlog** can be good option
- Either include some technical **dept issues in every sprint**, or dedicating **one sprint to only reducing technical debt**

Carnegie Mellon University

How to **Consider the Human Aspect of Software Design?**

Designing Large-scale Software Systems – Software Design Process

Carnegie Mellon University

Don't Design In an Isolated **Ivory Tower**

Ivory Tower Architects are:

- Not involved in the activities of software construction
- Ignoring input from other team members

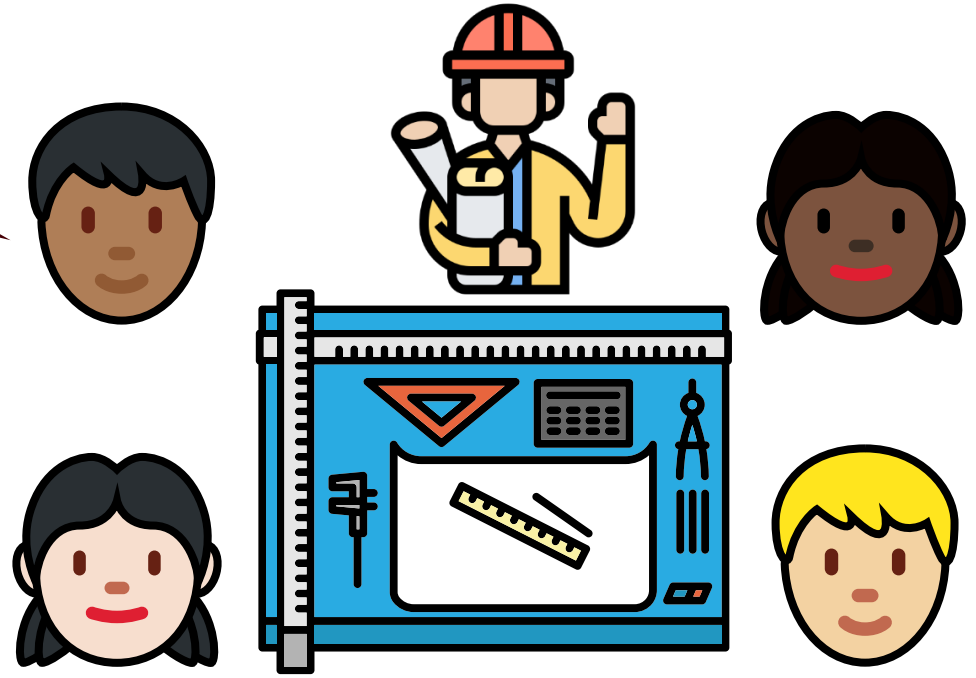
Ivory Tower Designs are:

- Elegant, beautiful, well-documented
- Only work in theory

Read more here: https://techcommunity.microsoft.com/t5/azure-architecture-blog/armchair-architects-architects-vs-the-ivory-tower/ba-p/3711703

Carnegie Mellon University

Diverse teams make better decisions



Lesson Learned: **Design is a Collaborative, Hands-on Activity**

- Include **developers** in important discussions to ensure **realism** of design
- Consult **domain experts** to take advantage of their **experience**
- Encourage other group members to present **design alternatives**
- Stay in touch with the current state of the **codebase**

Read more here: Smrithi Rekha V, Muccini, Henry. "Group decision-making in software architecture: A study on industrial practices." Information and software technology 101 (2018): 51-63.

Carnegie Mellon University

Rational Vs. Intuitive Decision Making

Rational Decision Making

(explicitly identifying, evaluating, and ranking design options via logical reasoning)

Can access **only explicit knowledge**

Documentation of rationale helps **revisiting** decisions

Helps **gain** explicit **knowledge** and **experience**

Guides **non-experts** to **better design** [1]

Intuitive Decision Making

(unconscious decisions relying on "gut feeling")

Hard to **communicate** / justify it

Challenging for **group decision making**

Prone to **cognitive biases**

(e.g., anchoring, confirmation bias, …)

Can access **all** implicit **knowledge** and **experience**

Helps experts with many years of experience to make **better decisions**

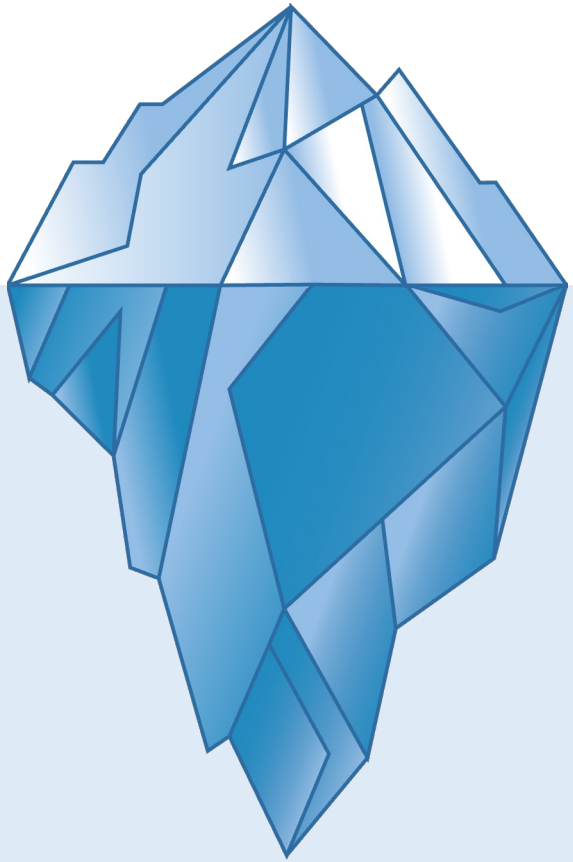
Can lead to **faster** decision making

[1] Tang, Antony, et al. "Design reasoning improves software design quality." *International Conference on the Quality of Software Architectures*. Springer. 2008.

Carnegie Mellon University Lesson Learned: **Combine Both Processes**

Rational Decision Making

(explicitly identifying, evaluating, and ranking design options via logical reasoning)



Appropriate Context:

- Justification is Needed
- Well-structured problem
- Optimal decision is needed

Intuitive Decision Making

(unconscious decisions relying on “gut feeling”)
aka. *Naturalistic Decision Making*

Appropriate Context:

- Time pressure
- Experienced decision makers
- Lack of information
- Hard-to-define problem
- Uncertainty
- “Good-enough” is sufficient

Read more here: Power, Ken, and Rebecca Wirfs-Brock. "An exploratory study of naturalistic decision making in complex software architecture environments." *European Conference on Software Architecture* 2019.
and Tang, Antony, et al. "Human aspects in software architecture decision making." *2017 IEEE International Conference on Software Architecture (ICSA).*
and Pretorius, Carianne, et al. "Combined intuition and rationality increases software feature novelty for female software designers." *IEEE Software* 38.2 (2020): 64-69.

Carnegie Mellon University

How do these insights impact our approach to software design?

Bounded Rationality

- The rationality of our design decisions is **limited by our cognitive capabilities**
- Realistically, we **cannot consider all possible** design options to achieve an optimal design
- Designers often **retroactively rationalize** decisions

Read more here: Tang, Antony, et al. "Human aspects in software architecture decision making." *2017 IEEE International Conference on Software Architecture (ICSA).*
and Tang, Antony, and Hans van Vliet. "Software designers satisfice." *Software Architecture: 9th European Conference, ECSA 2015,*

Carnegie Mellon University

How to **Adjust the Design Process To Domain-Specific Needs?**

Designing Large-scale Software Systems – Software Design Process

Carnegie Mellon University

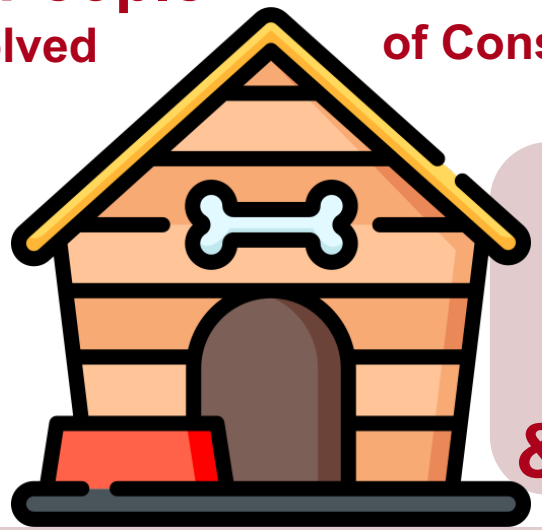
How do these insights apply to software engineering?

How does the **Design Process Differ** for **Doghouses** and **Skyscrapers**?

Fewer People involved

Short Process of Construction

Lower Risk of Failure



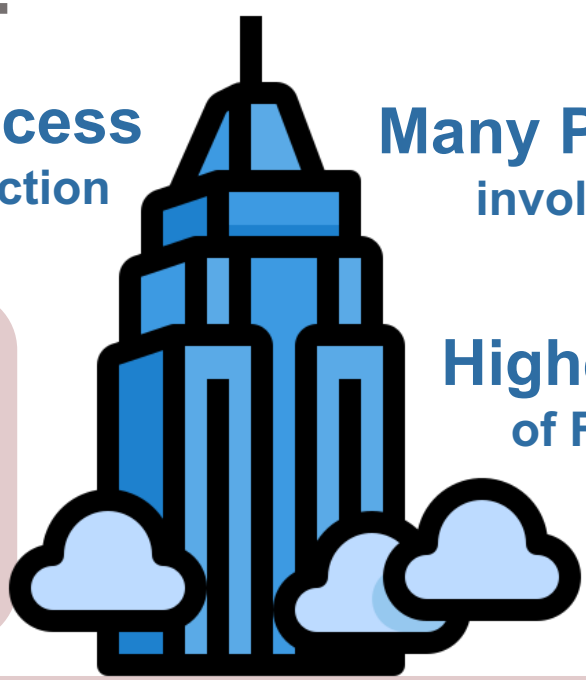
Less Upfront Design & Models

Higher Percentage of **Intuitive Decision Making**

Long Process of Construction

Many People involved

Higher Risk of Failure



More Upfront Design & Models

Higher Percentage of **Rational Decision Making**

Read more here: Software Architecture in Practice 3rd Edition Chapter 15

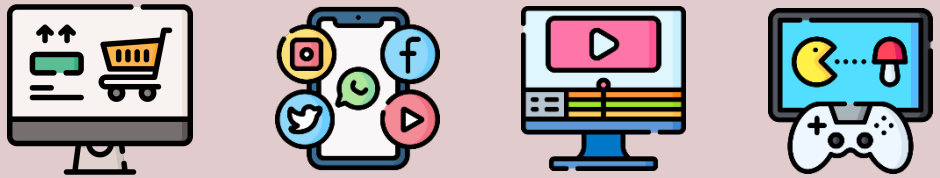
Carnegie Mellon University

Lesson Learned: **Adjust the Design Process to the Specific Domain**

Higher Risk Domains

- More Upfront Design
- Detailed Design Documents
- Rigorous Design Evaluation

Lower Risk Domains



- Some Upfront Design
- **Focusing on Highest Risks**
- Designing while Coding

Carnegie Mellon University

"Go fast and break things" – Mark Zuckerberg
(CEO of Facebook / Meta)

Web-based Social Media Apps

Risks

Usability: Easily change UI
Changeability: Easily add new features
Scalability: Support growth of userbase

Amount of Upfront Design

Small to Medium. Mostly limited to technology choices, client-server interfaces, component structures, and data models

Process Changes

Agile Process: Iterative development. "Perpetual development" (i.e., no predefined final objective). Frequent releases. Testing and peer review instead of design review. Responding to usage metrics, public opinion, and competitors.
In practice, a large portion of decisions are made intuitively, due to rapid development cycle.
We recommend to deliberately think of hard-to-change design decisions!

For more details see: Feitelson, Dror G., Eitan Frachtenberg, and Kent L. Beck. "Development and Deployment at Facebook." *IEEE Internet Computing* 17.4 (2013): 8-17.

Carnegie Mellon University

Case Study: Design Decision Making at Google

Decision Makers

Decisions are made by **self-organized, autonomous teams** based on rational persuasion and data. Tech Lead approves the design.

Design Artifacts

Informal Design Docs (goals, non-goals, context diagrams, interface descriptions, data models, alternative options, and justification for chosen design).

Just like your project reports!

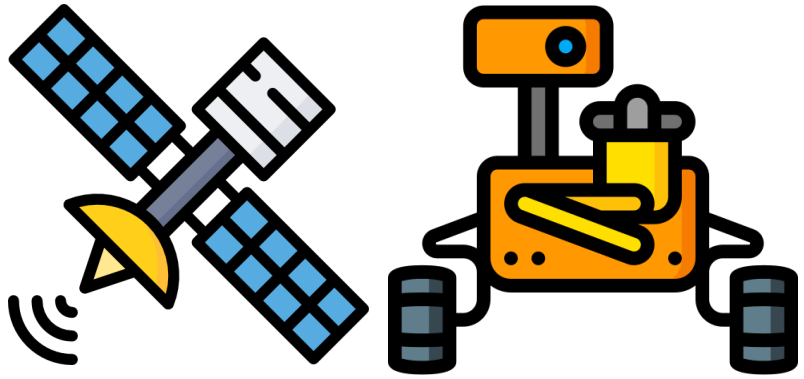
Design Process

Creating design docs **before** implementing **major decisions**. Discussion & review mostly via comments. Some teams have weekly design review meetings.

Read more here: https://www.industrialempathy.com/posts/design-docs-at-google/ and https://open.lib.umn.edu/organizationalbehavior/chapter/11-1-decision-making-culture-the-case-of-google

Carnegie Mellon University

“Failure is not an option” – Gene Kranz
(NASA Flight Director of Apollo 13)



Spacecraft Software

Risks

Robustness: Operate reliably in uncertain environments without human interference

Testability: Detecting faults on Earth is hard

Amount of Upfront Design

A lot! Many models & formal design reviews. Mission-critical elements are analyzed very rigorously.

Process Changes

“Waterfall-like” Process: Limited benefits of full-cycle iterations, due to single launch date.

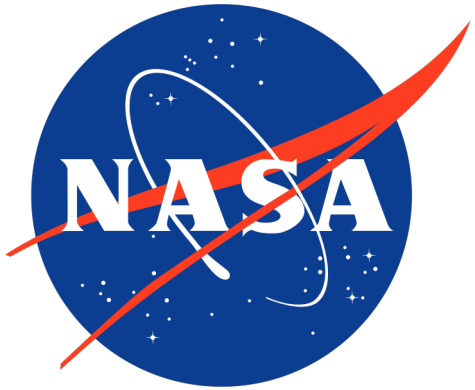
Avoiding intuitive decision making to extensively document & review design decisions.

Formal validation & verification of important components due to high cost of failure.

External reuse is very un-common, due to extreme reliability requirements. NASA even re-built their own Linux kernel. Internal reuse is very common.

For more details see: Markosian, Lawrence Z., et al. "Program model checking using Design-for-Verification: NASA flight software case study." 2007 IEEE Aerospace Conference. IEEE, 2007.

Carnegie Mellon University



Case Study: Design Decision Making at NASA

Decision Makers

Project managers develop, record, and maintain software design documents are reviewed based on **detailed checklists.**

Design Artifacts

Detailed Design Documents (very long documents outlining every aspect of the structure, behavior, and quality attributes of the design)

Design Process

Top-Down: System Definition Review -> Preliminary Design Rev. -> Critical Design Rev. -> System Integration Rev. -> Test Readiness Rev. -> System Acceptance Rev.

Read more here: https://swehb.nasa.gov/display/SWEHBVD/SWE-058+-+Detailed+Design

SADESIGN CHECKLIST

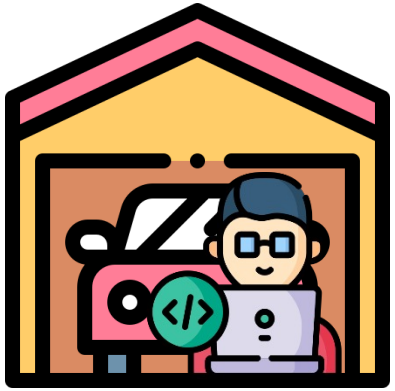
Project:	Organization Examined:	Revision Date: PAT05/31/2022
Date(s):	Assessor(s):	Page: 1 of 3

This checklist is designed for verifying that best design practices have been followed and that the design has considered all portions of the design for NASA mission software. It contains sample questions that might be used to formally inspect and analyze the resulting design. This checklist may be modified to address the needs of the project or organization.

#	Description	Y/N	Comments
Sub-section Title			
a.	Has the software design been developed at a low enough level for coding?		
b.	Is the design complete and does it cover all the approved requirements?		
c.	Have complex algorithms been correctly derived, provide the needed behavior under off-nominal conditions and assumed conditions, and is the derivation approach known and understood to support future maintenance?		
d.	Examine the design to ensure that it does not introduce any undesirable behaviors or any capabilities, not in the requirements?		
e.	Have all requirements sources been considered when developing the design (e.g., system requirements, interface requirements, databases, etc.)?		
f.	Have the interfaces with COTS, MOTS, GOTS, and Open Source been designed (e.g., APIs, .dlls)		
g.	Have all internal and external software interfaces been designed for all (in-scope) interfaces with hardware, user, operator, software, and other systems and are they detailed enough to enable the development of software components that implement the interfaces?		
h.	Are all safety features in the design e.g., (mitigations, controls, barriers, must-work requirements, must-not-work requirements		
i.	Does the design provide the dependability/reliability and fault tolerance/Fault Detection and Recovery (FDIR) required by the software, and is the design capable of controlling identified hazards? Does the design create any hazardous conditions?		
j.	Does the design adequately address the identified security requirements both for the software and security risks, including the integration with external components as well as information and data utilized, stored, and transmitted through the software?		
k.	Does the design prevent, control, or mitigate any identified security threats, weaknesses and vulnerabilities? Are any unmitigated weaknesses and vulnerabilities documented as risks and addressed as part of the software and software operations?		
l.	Have operational scenarios have been considered in the design (for example, use of multiple individual programs to obtain one particular result may not be operationally efficient or reasonable; transfers of data from one program to another should be electronic, etc.).		
	Have users/operators been consulted during design to identify any potential operational issues?		
	Maintainability: Has maintainability been considered? Is the design modular? Is the design easily extensible? Is it designed to allow for the addition of new capabilities and functionality?		
	Portability: Has portability been considered? Are environmental variables used? Can the software be moved to other environments quickly?		
	Is the design easy to understand?		
	Is the design unnecessarily complicated?		
	Is the design adequately documented for usability and maintainability?		
	Does the design address error handling?		
	Has software performance been considered during design? Has the software design been optimized for efficiency to reduce system load, run-time length/speed, etc.?		
	Has the level of coupling (interactivity between modules) been kept to a minimum?		
	Has software planned for reuse and OTS software in the system been examined to determine if it meets the requirements and performs appropriately within the required limits for this system? Has the software been evaluated for security vulnerabilities and weaknesses?		
	Does this software introduce any undesirable capabilities or behaviors?		
	Has the software design been peer reviewed?		
	Are components referenced by more than one application, file, module, components, functions, subroutines, classes, etc. stored in a common area such as a library, class, or package?		
	Summary of Analysis		

"I'm a long-term kind of person." – Steve Jobs
(Founder of Apple)

"Fake it until you make it"



Software Startups

Risks

Extensibility: Quickly respond to new customer needs

Time-to-Market: Quickly start breaking even

Amount of Upfront Design

None to Small. Most design happens implicitly while coding or after the first release. Decisions are driven by short-term needs.

Process Changes

Lean & Agile Process: Rapid prototyping & taking shortcuts to quickly get to the *minimum viable product (MVP)*. Relying as much on reuse as possible can speed up development. After reaching the MVP and/or breaking-even point, paying more attention to clean code and clean architecture supports future growth, onboarding of new developers, extensibility, and scalability to build a robust foundation for long-term success. But: **Avoid over-engineering**

For more details see: Tegegne, Esubalew Workineh, Pertti Seppänen, and Muhammad Ovais Ahmad. "Software development methodologies and practices in start-ups." IET Software 13.6 (2019)

Carnegie Mellon University

Stop Upfront Design When…

- ☑ You understand the **significant architectural drivers**
- ☑ You understand the **context** and **scope** of what you are building
- ☑ You understand the **significant design decisions**
- ☑ You have a way to **communicate** your **technical vision** to others
- ☑ You are confident that the **design satisfies key architectural drivers**
- ☑ You have **identified** and are **comfortable** with the project's **risks**

For more details see: The lost art of software design by Simon Brown

Carnegie Mellon University

Course Policy-Reminder

"The use of **generative AI** has to be explicitly marked as such, including your prompts and a screenshot of the result. While we encourage you to critically engage with generative AI and use it for idea generation, the submitted solution has to be your own and differ significantly from responses of generative AI. Generative AI is not allowed on exams or exit tickets."

Carnegie Mellon University

Please Complete the Exit Ticket in Canvas!

Question 1 1 pts

Please describe **what role software design plays in agile software projects.**

Question 2 1 pts

Please describe what **adjustments to the design process** would need to be made when developing **self-driving cars**.

Question 3 1 pts

Please leave any questions that you have about today's materials and things that are still unclear or confusing to you (if none, simply write N/A).

Carnegie Mellon University

Summary

- Identify **biggest risks** of your software and focus design on these
- **Design for Change** to support flexibility of changing requirements
- **Delay important decisions** to a later point in time and design the architecture to hide this decision in a single module
- Maintain a **Technical Debt Backlog** additionally to the Feature Backlog to maintain issues that improve software design
- **Adjust the design process** based on the domain's needs