Software Development Methodologies

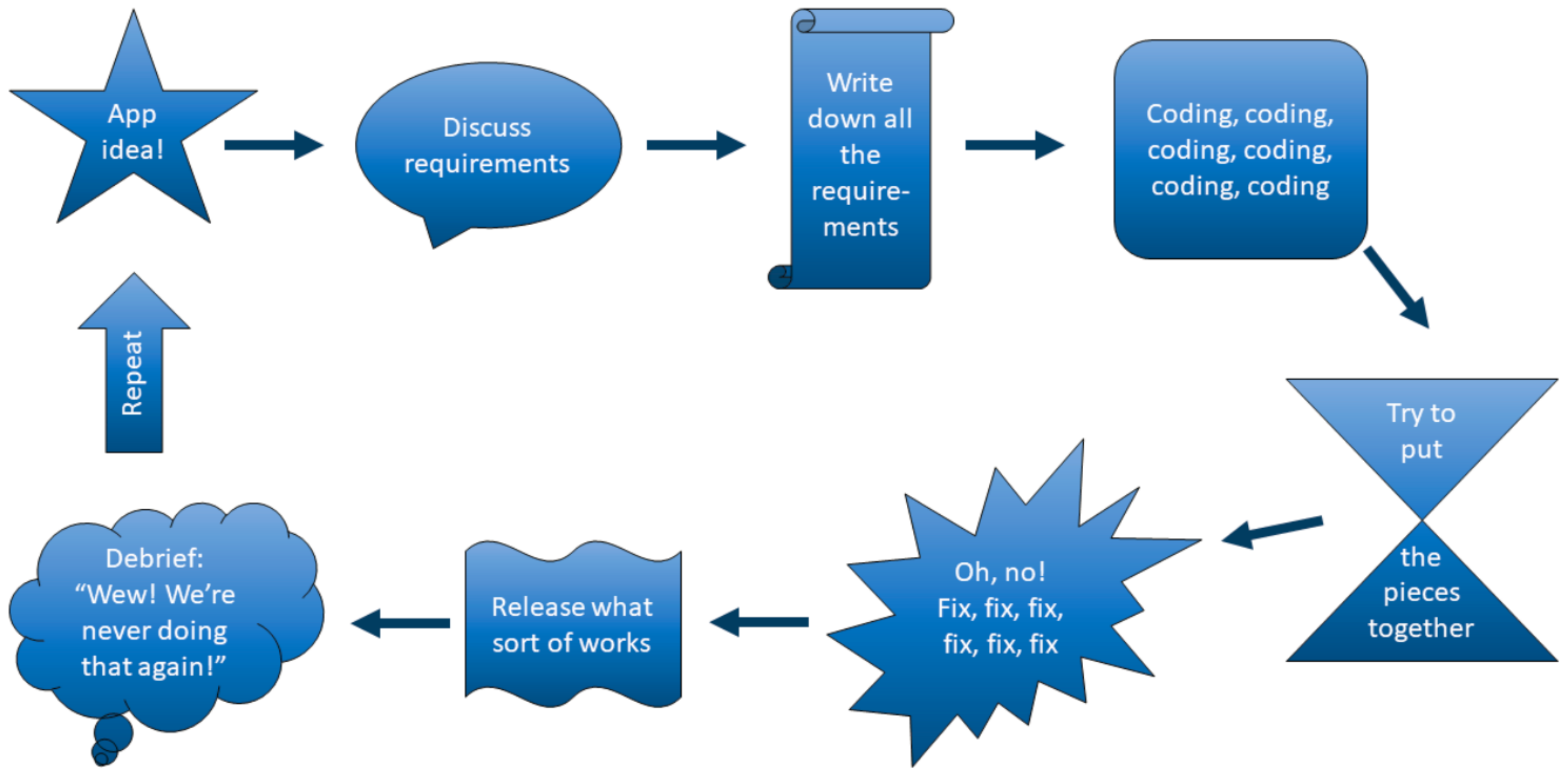
ELEC 376 (week 1)

TODO: Programing Experience and Teaming survey

- **TODO:** Programming and teamwork survey due **Today (11:59pm)**
  - On onQ: week 1 (activities and assessments)
  - You need to fill out the survey so you can be added to a group for the term project
- **Note:** start thinking about ideas for your project

Class Representative

What not to do


App idea!
Discuss requirements
Write down all the require-ments
Coding, coding, coding, coding, coding, coding
Try to put the pieces together
Oh, no! Fix, fix, fix, fix, fix, fix
Release what sort of works
Debrief: “Wew! We’re never doing that again!”
Repeat

Problems with such process

- Not fun
- Not fast
- Not profitable
- Not reliable
- Not mature
- Not great for the customers and users, either

Software engineering

- Software Engineering:
  - A structured approach with defined procedures to develop and maintain software.
- **Characteristics:**
  - **Systematic**
  - **Structured Procedures**
  - **Goal-Oriented** (remember ELEC 290, how do you define success?)

Software engineering

- **Examples:**

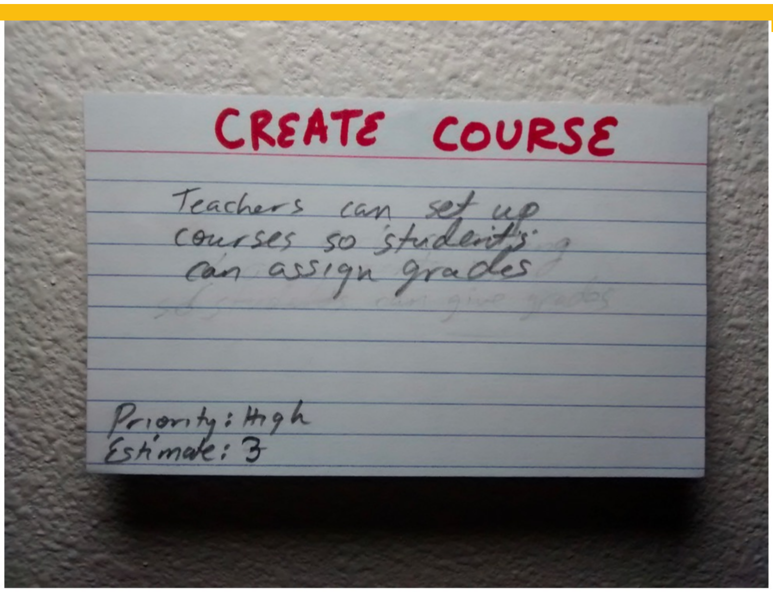
  - *User Stories:* Communicates and captures requirements effectively.

  - *Unit Testing:* Ensures quality and functionality of code.

User stories

Keys to a good user story...

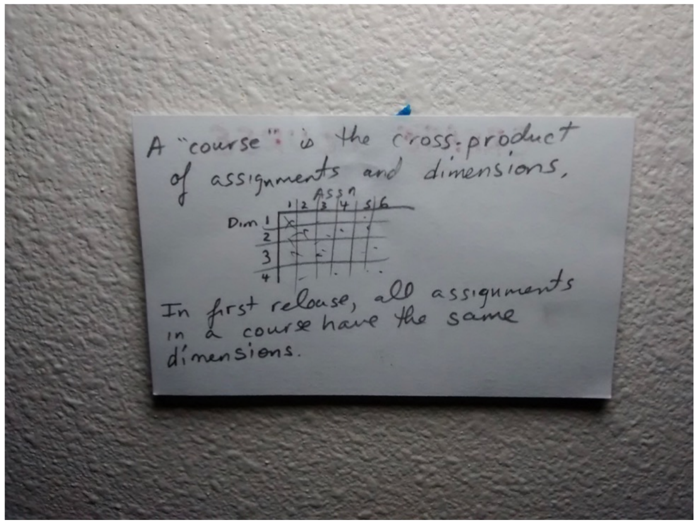
- **Concise**
- **Clear**
- **Separable**
- **Success criteria**



CREATE COURSE

Teachers can set up courses so students can assign grades

Priority: High
Estimate: 3



A "course" is the cross-product of assignments and dimensions,

Assn
1
2
3
4
5
6
Dim
1
2
3
4

In first release, all assignments in a course have the same dimensions.

Example: Peer grading system

- A system for students to grade one another

Example: Peer grading system

- A system for students to grade one another
- **User stories:**
  - Log in (Student, Teacher)
  - Create course (T)
  - Form team (S)
  - Enter grades (S)
  - View histograms (T)

Example: Peer grading system

- A system for students to grade one another
- **User stories:**
  - Log in (Student, Teacher)
  - Create course (T)
  - Form team (S)
  - Enter grades (S)
  - View histograms (T)
  - Authorize TA (T)
  - Download grades (T)
  - Freeze grades (T)
  - View my grades (S)

When writing user stories

- Think about who the users are
- On each card, write one sentence about what a user could do with your software

  Mention: Who needs to be able to do what
- Give each card a name
- On the back, write any detailed notes needed to clarify
- Priority: high, medium, low
- Estimated time? 3h? 7h?

  Concise? Clear? Separable?

Three dominant families of software development methods

Agile

- You can trust your team to deal with whatever life throws at them.

Waterfall

- You can get things right on the first try, if you plan carefully enough.

Spiral

- Things are confusing and risky, so take time to consider your options.

Agile Characteristics

- **Interacting Individuals:**
  - Prioritizes face-to-face communication and collaboration among team members.
  - You must meet at least twice a week with your team (one of them during tutorial)
    - You will need evidence of this for your sprint reports (Week 5, Week 8)

Agile Characteristics

- **Interacting Individuals**
  - Prioritizes face-to-face communication
- **Iterative cycles of working software**

Agile Characteristics

- **Customer Collaboration:**
  - Engages customers actively throughout the development process to ensure alignment with their needs.

Agile characteristic

- **Customer Collaboration:**
  - Engages customers actively throughout the development process to ensure alignment with their needs.
- **Responding to Change:**
  - Embraces changes in requirements, adapting and adjusting throughout development.

Consequent Characteristics

- **Limited Reliance on Formalized Processes:**

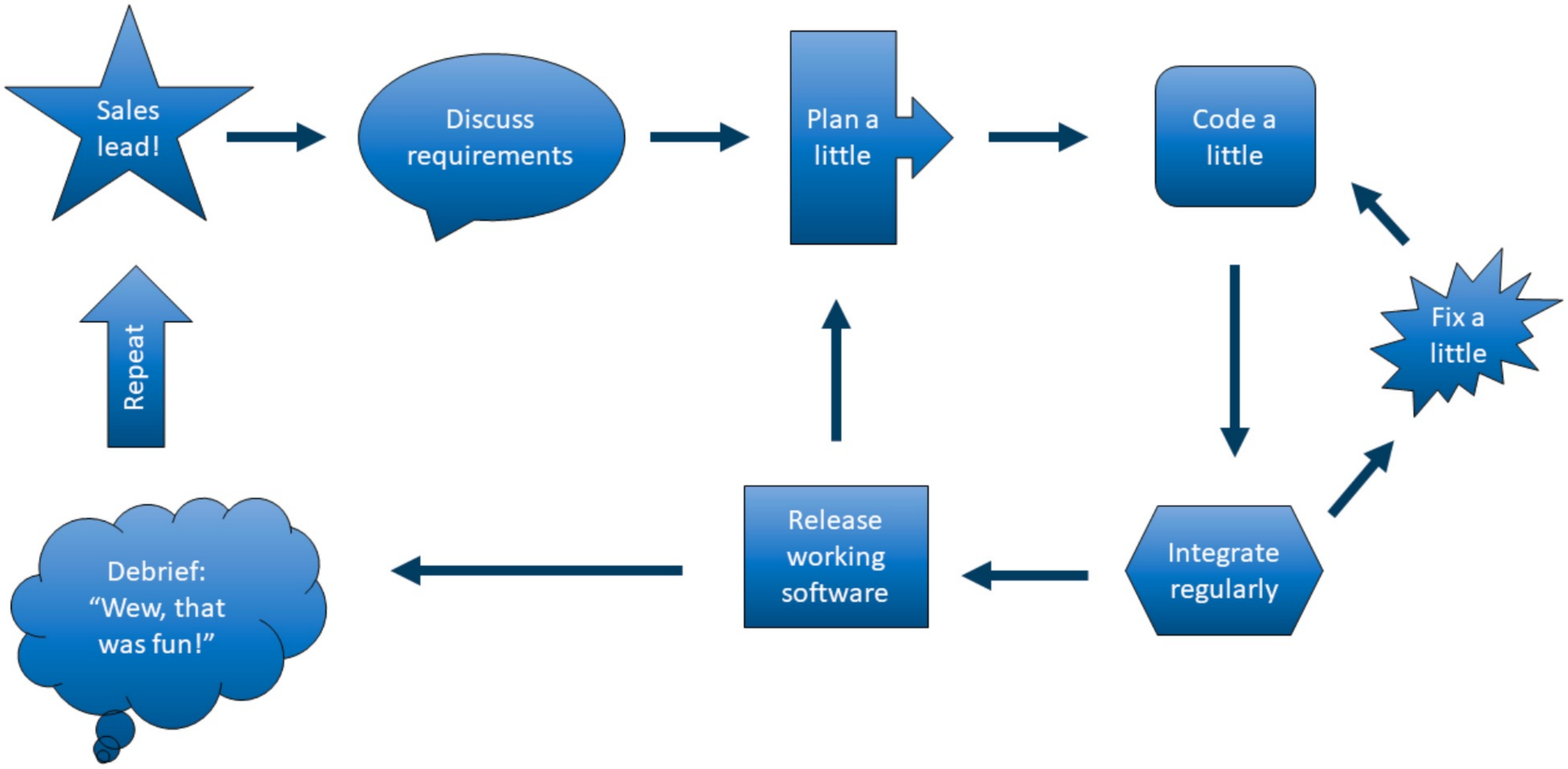
- **Continuous Integration and frequent release:**

- **Emphasis on Adaptability:**

Consequent Characteristics

- **Front-loading of Coding as Requirements Develop:**
  - Encourages early development based on evolving requirements, allowing for agile responses to changes.

Sketch of an agile process



Agile pros and cons

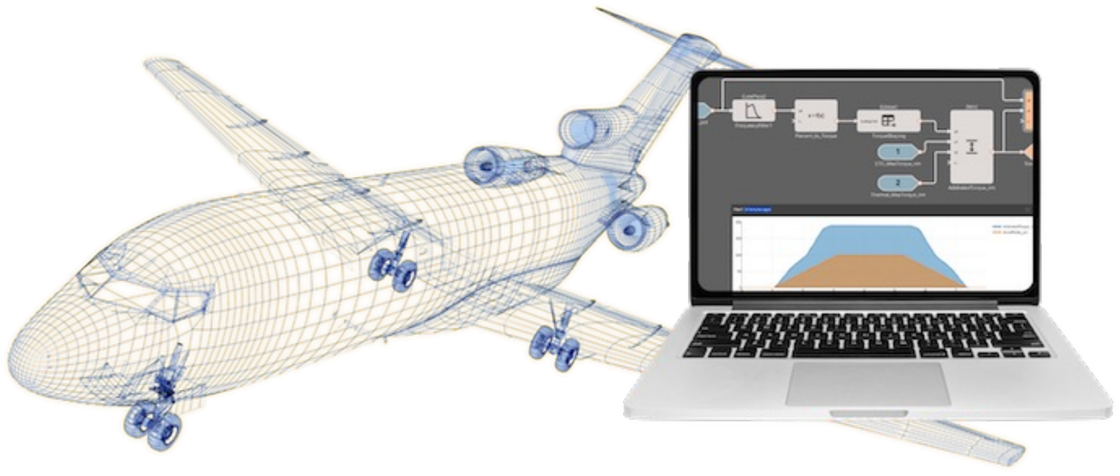
Agile is great when…

- You can identify a place to start
- And mistakes aren't a problem

Agile pros and cons

Agile is a poor choice when...

- Extensive planning is needed and/or mistakes are problematic



Examples of Agile Process

- **Scrum**
  - 2 - 4-week cycles called “sprints”, narrowly defined roles
- **Kanban**
  - Very similar to scrum; has additional rules about how project gets managed
  - Emphasizes continual delivery, flexibility, and efficiency by **visualizing workflow** on a board, allowing teams to manage tasks in a just-in-time manner

Examples of Agile process

- **eXtreme Programming**
  - Less similar to scrum; has additional rules about how implementation gets done
  - Improving **software quality** and responsiveness using practices like **pair programming**, frequent releases, continuous testing, and customer collaboration.
- **Feature-driven development**
  - Less similar to scrum; also has additional rules about modeling/diagramming
  - Building features incrementally, with a focus on **domain object modeling**, iterative development, and regular client input to deliver high-quality software

Scrum

Your brief project reports will mimic sprints:

- **Sprint 1 (Week 3 - Week 5 inclusive)**
    - A good number of user stories implemented , your system is alive
- **Sprint 2 (Week 6 – Week 8 inclusive)**
    - Many user stories implemented, your system is useful
- **Sprint 3 (Week 9 – To presentation)**
    - Most user stories, your system is desirable

Why cover Waterfall and Spiral?

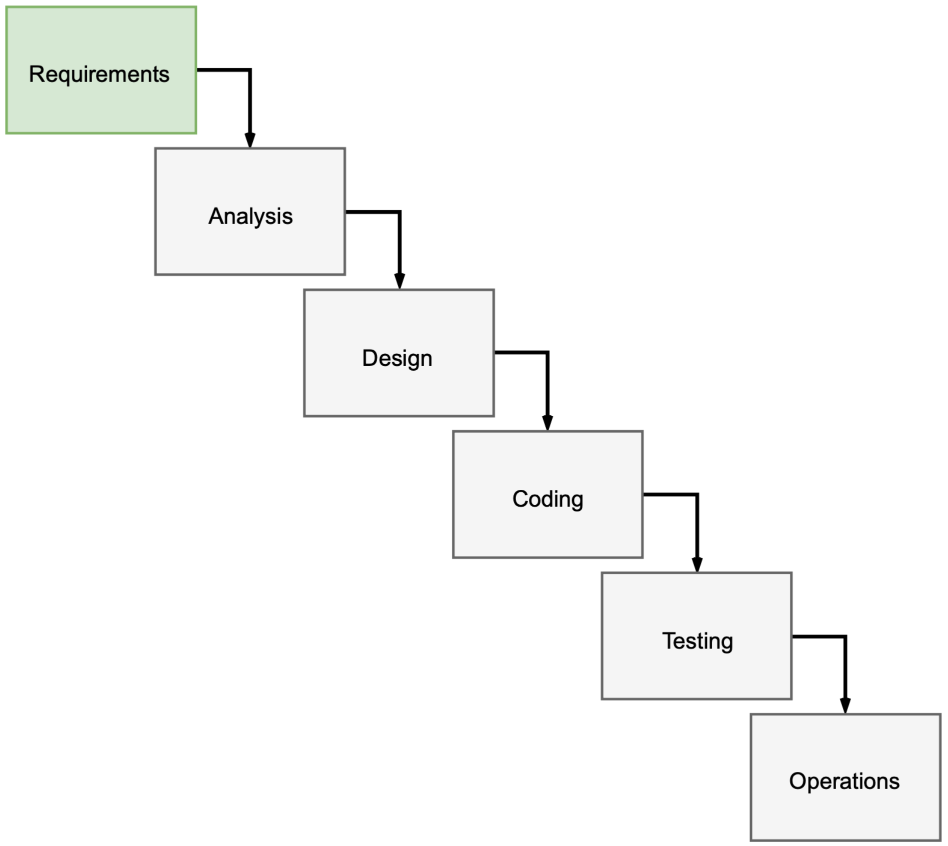
Companies rarely use pure agile

- Because those methods are sometimes useful

Why cover Waterfall and Spiral?

- You should make selective use of those methods in your projects
  - Example: Use cases
    - Can be handy when a user story is too vague
    - Doesn't really stop you from being agile
  - Example: System Design Document (SDD)
    - Comprehensive specification of requirements and design

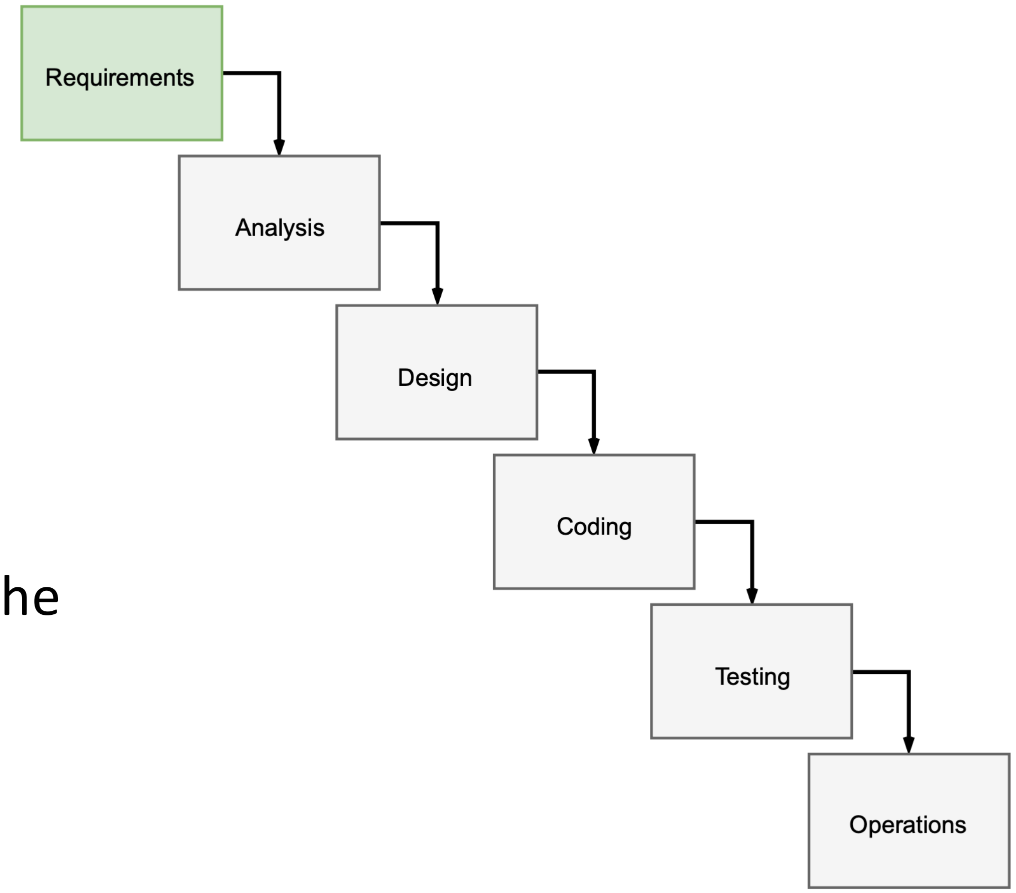
Waterfall in a visual



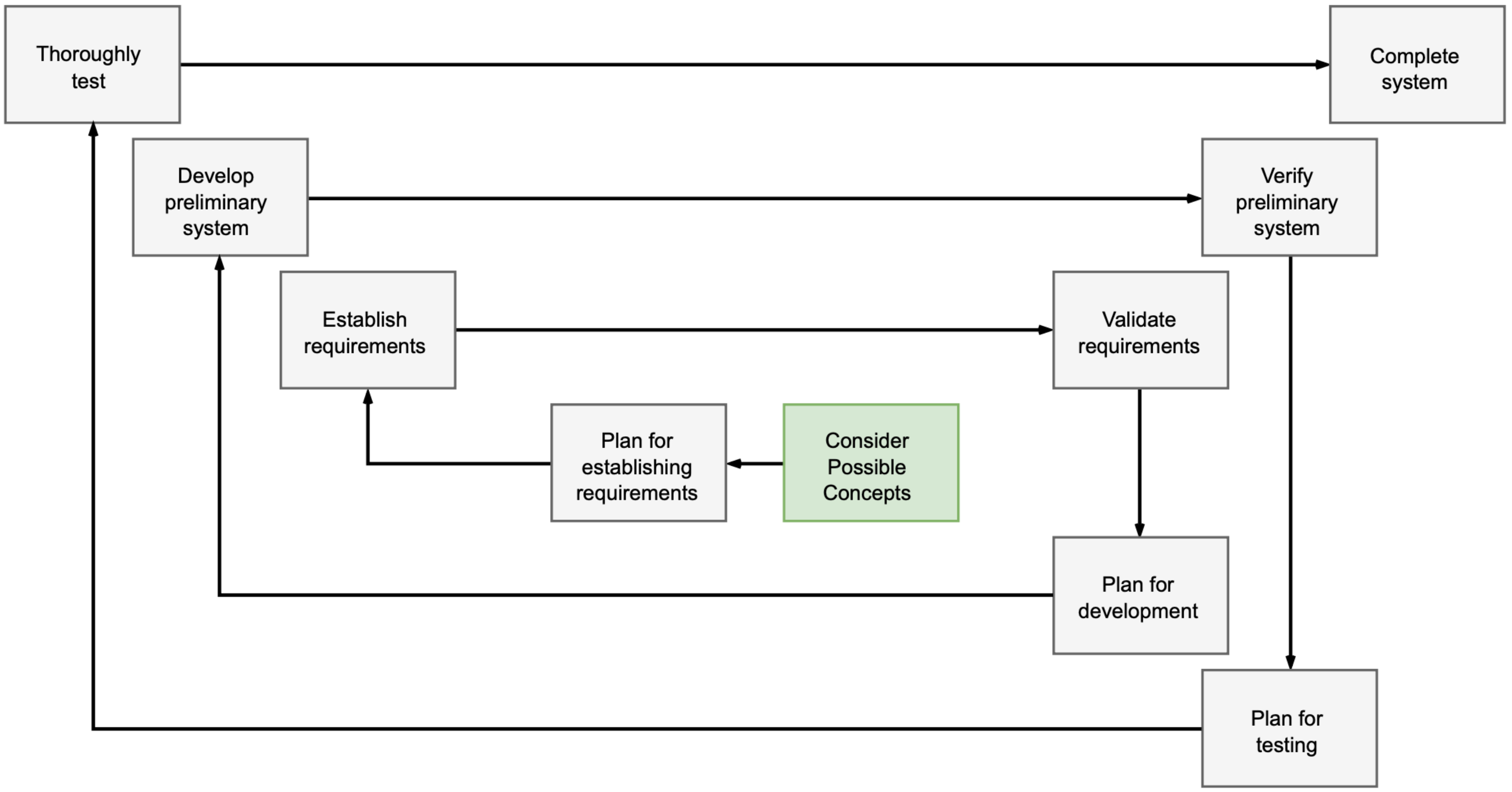
Waterfall in a visual

Waterfall is great when

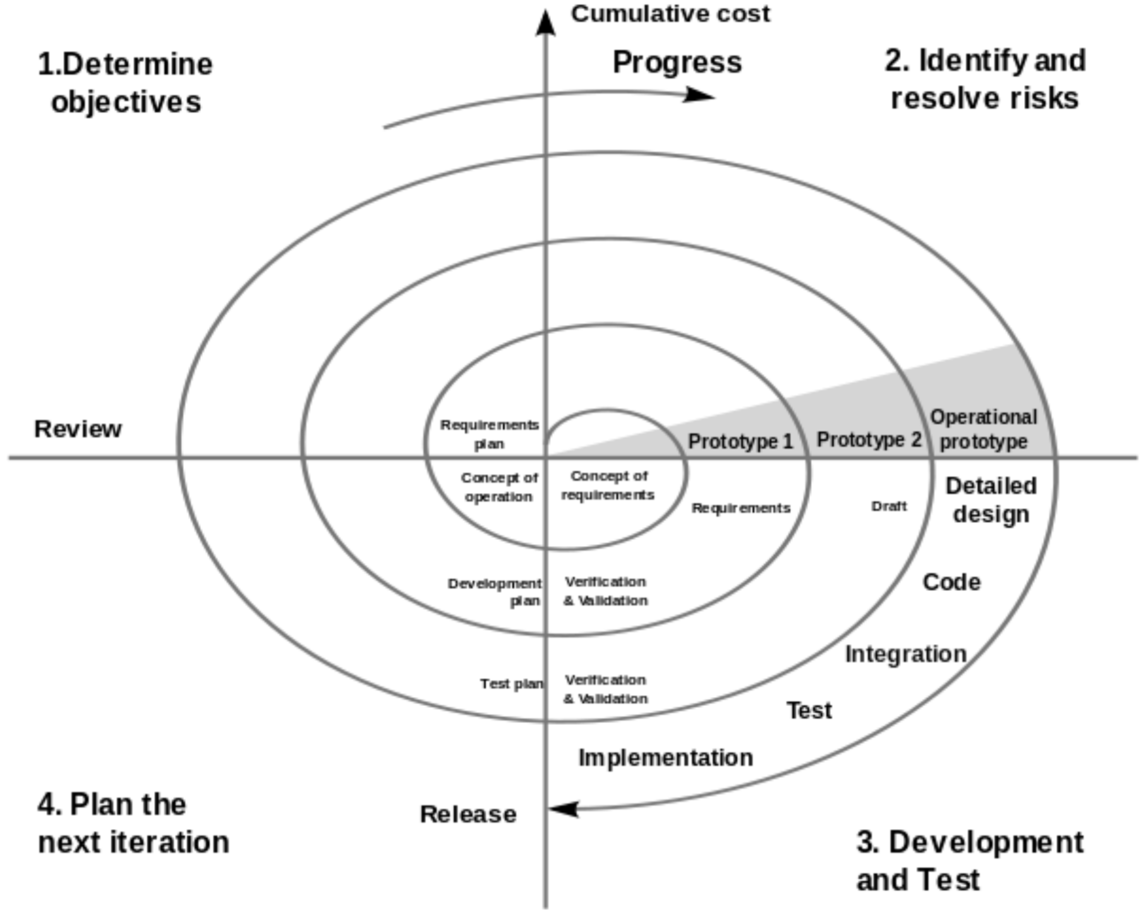
- Mistakes are costly and problematic
- And/or you can actually get the right answer by planning



Spiral in a visual


Thoroughly test
Complete system
Develop preliminary system
Verify preliminary system
Establish requirements
Validate requirements
Plan for establishing requirements
Consider Possible Concepts
Plan for development
Plan for testing

Boehm's spiral model Proposed in 1985



Spiral compared to others

Spiral vs waterfall

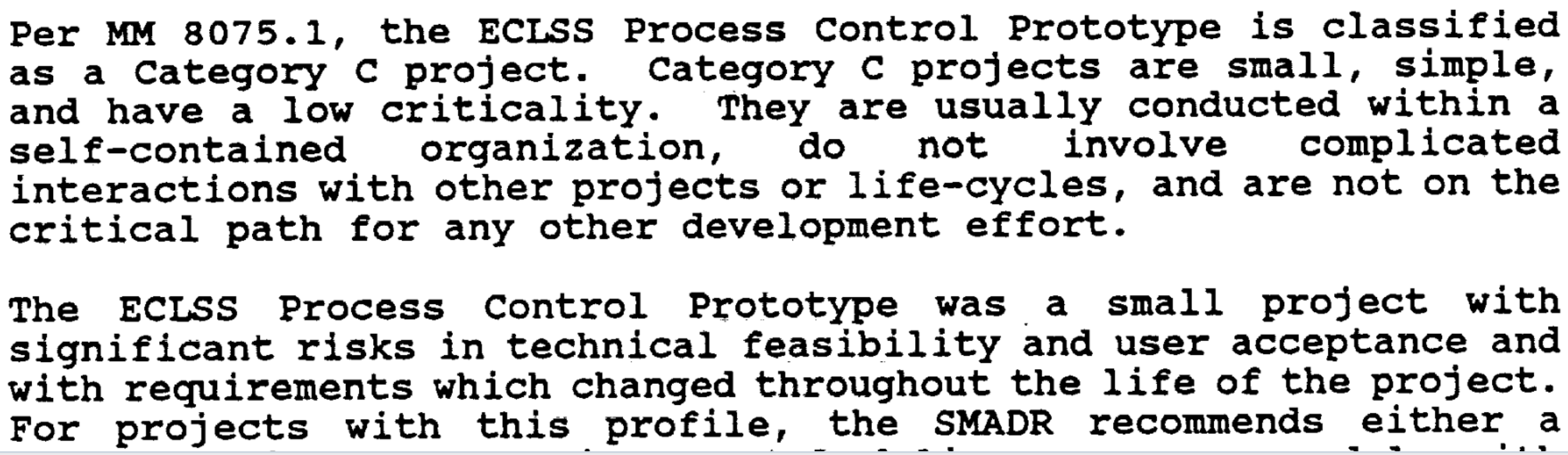
- Spiral takes waterfall one step further by explicitly incorporating notion of **risk management** through iterative refinement
- You are still planning, but it is more like **"plan and re-plan"** repeatedly
- The plan is much more open to negotiation than in waterfall

Spiral compared to others

Spiral vs Agile

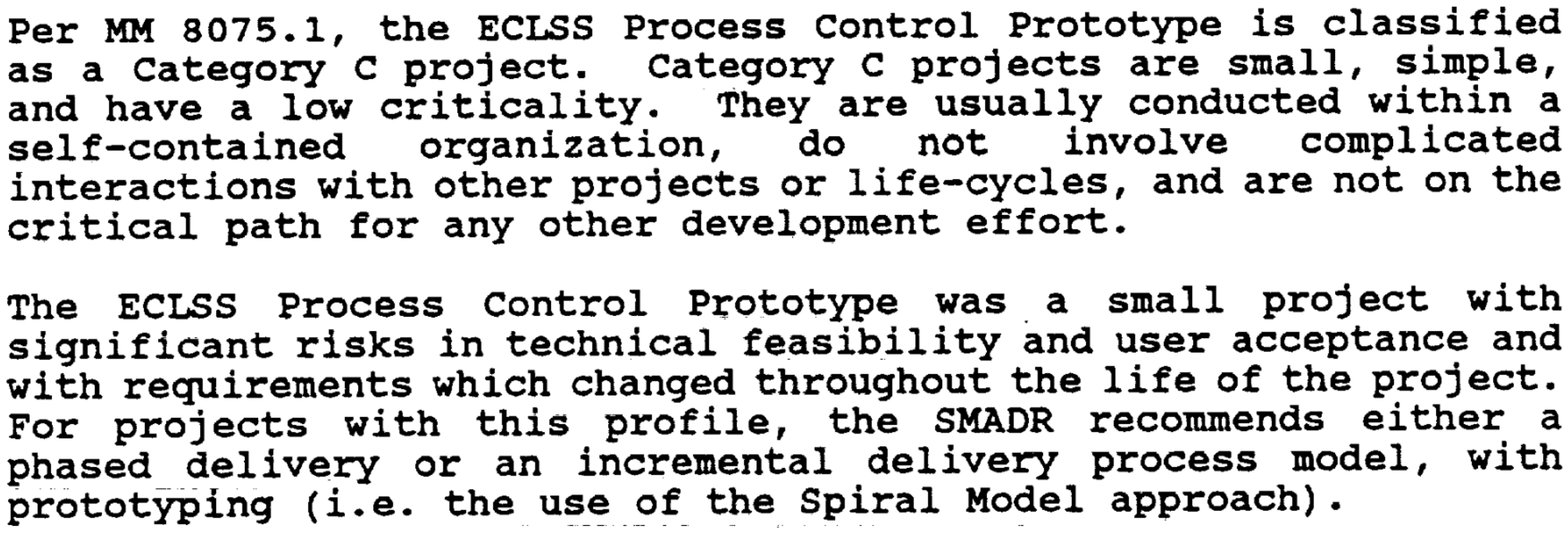
- Spiral invests lots of time and energy **in avoiding mistakes**
- You are **looking ahead** very carefully before every single step
- Spiral projects are often bigger, costlier, longer than in agile

Agile, Waterfall, or Spiral?



Per MM 8075.1, the ECLSS Process Control Prototype is classified as a Category C project. Category C projects are small, simple, and have a low criticality. They are usually conducted within a self-contained organization, do not involve complicated interactions with other projects or life-cycles, and are not on the critical path for any other development effort.

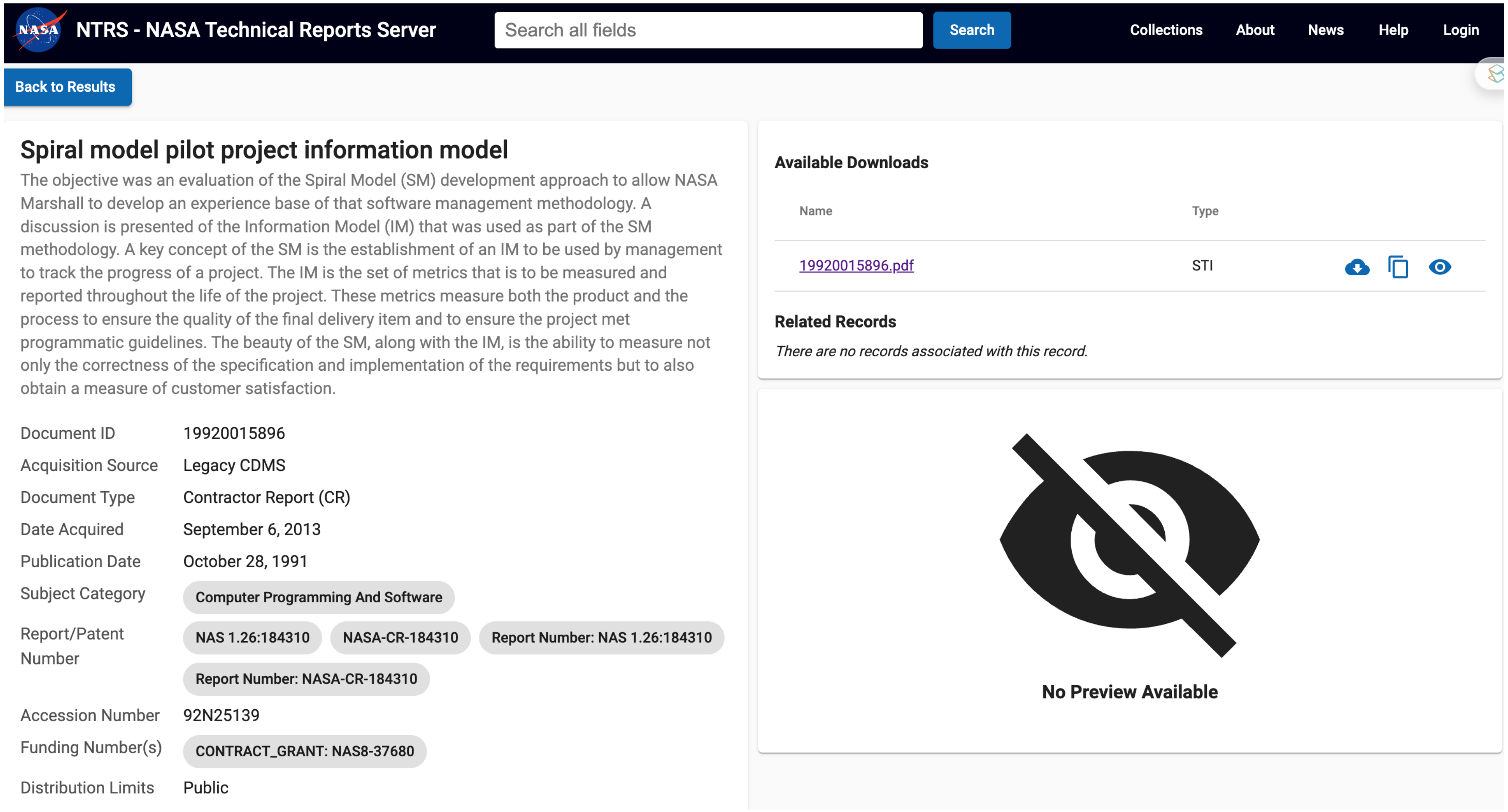
The ECLSS Process Control Prototype was a small project with significant risks in technical feasibility and user acceptance and with requirements which changed throughout the life of the project. For projects with this profile, the SMADR recommends either a



Per MM 8075.1, the ECLSS Process Control Prototype is classified as a Category C project. Category C projects are small, simple, and have a low criticality. They are usually conducted within a self-contained organization, do not involve complicated interactions with other projects or life-cycles, and are not on the critical path for any other development effort.

The ECLSS Process Control Prototype was a small project with significant risks in technical feasibility and user acceptance and with requirements which changed throughout the life of the project. For projects with this profile, the SMADR recommends either a phased delivery or an incremental delivery process model, with prototyping (i.e. the use of the Spiral Model approach).

https://ntrs.nasa.gov/api/citations/19920015896/downloads/19920015896.pdf



Spiral model pilot project information model

The objective was an evaluation of the Spiral Model (SM) development approach to allow NASA Marshall to develop an experience base of that software management methodology. A discussion is presented of the Information Model (IM) that was used as part of the SM methodology. A key concept of the SM is the establishment of an IM to be used by management to track the progress of a project. The IM is the set of metrics that is to be measured and reported throughout the life of the project. These metrics measure both the product and the process to ensure the quality of the final delivery item and to ensure the project met programmatic guidelines. The beauty of the SM, along with the IM, is the ability to measure not only the correctness of the specification and implementation of the requirements but to also obtain a measure of customer satisfaction.

Document ID	19920015896
Acquisition Source	Legacy CDMS
Document Type	Contractor Report (CR)
Date Acquired	September 6, 2013
Publication Date	October 28, 1991
Subject Category	Computer Programming And Software
Report/Patent Number	NAS 1.26:184310; NASA-CR-184310; Report Number: NAS 1.26:184310; Report Number: NASA-CR-184310
Accession Number	92N25139
Funding Number(s)	CONTRACT_GRANT: NAS8-37680
Distribution Limits	Public

Available Downloads

Name	Type
19920015896.pdf	STI

Related Records

There are no records associated with this record.

No Preview Available

Variable types

- How many types can you have in C? Not including structs

Variable types

- How many types can you have in C? Not including structs

- How many types can you have in C++?

Variable types

- How many types can you have in C? Not including structs
  - You can have the atomic types, pointers and arrays of these types
- How many types can you have in C++?
  - Infinitely many!

Variable declaration

- C++ is declarative and statically typed.

- Initialization is good practice.

Type categories in C++

- Fundamental or “Atomic” types:
  - **bool**, **char**, integer types and floating-point types
- Enumerated types.
- The **void** type.
- Pointer types (like **int***)
- Array types (like **int[]**)
- Reference types (like **int&**) – alias
- Data structures (like **vector**) and classes

Bool type

- **bool** literal values: **true** and **false**
- C programmers don’t worry too much about Boolean types since they require an #include or a #define.
- In C++, you have a native bool type – use it!
- This makes your code more readable!

Types of constants

- Literal constants
- Declared or defined constants
- Enumerated constants
- Constant expressions C++11

Declared or defined constants

```
// Integer constant
const int MONTHS = 12;

// Floating-point constant
const double GRAVITY = 9.8;
```

```
// Define a constant for the value of PI
#define PI 3.14159
```

Enumerated constants

```
enum Days {

     Sunday, // 0
     Monday, // 1
     Tuesday, // 2
     Wednesday, // 3
     Thursday, // 4
     Friday, // 5
     Saturday // 6
};
```

Constant expressions [C++11s]

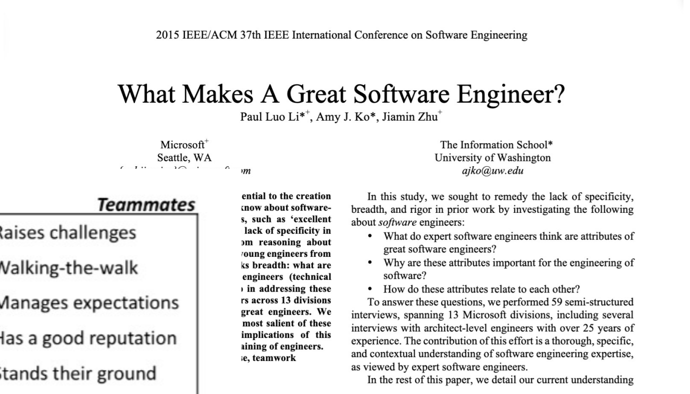
- Saves runtime performance

```
constexpr int add(int a,
int b) {
     return a + b;
}
```

```
constexpr int result = add(3, 4);
cout << result << endl;
```

https://en.cppreference.com/w/cpp/language/constexpr

Interesting read



2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

What Makes A Great Software Engineer?

Paul Luo Li*[+], Amy J. Ko*, Jiamin Zhu[+]

Microsoft[+]
Seattle, WA

The Information School*
University of Washington
ajko@uw.edu

ential to the creation
know about software-
s, such as 'excellent
lack of specificity in
om reasoning about
young engineers from
ks breadth: what are
engineers (technical
in addressing these
rs across 13 divisions
great engineers. We
most salient of these
implications of this
aining of engineers.
e, teamwork

In this study, we sought to remedy the lack of specificity, breadth, and rigor in prior work by investigating the following about *software* engineers:

- What do expert software engineers think are attributes of great software engineers?
- Why are these attributes important for the engineering of software?
- How do these attributes relate to each other?

To answer these questions, we performed 59 semi-structured interviews, spanning 13 Microsoft divisions, including several interviews with architect-level engineers with over 25 years of experience. The contribution of this effort is a thorough, specific, and contextual understanding of software engineering expertise, as viewed by expert software engineers.

In the rest of this paper, we detail our current understanding

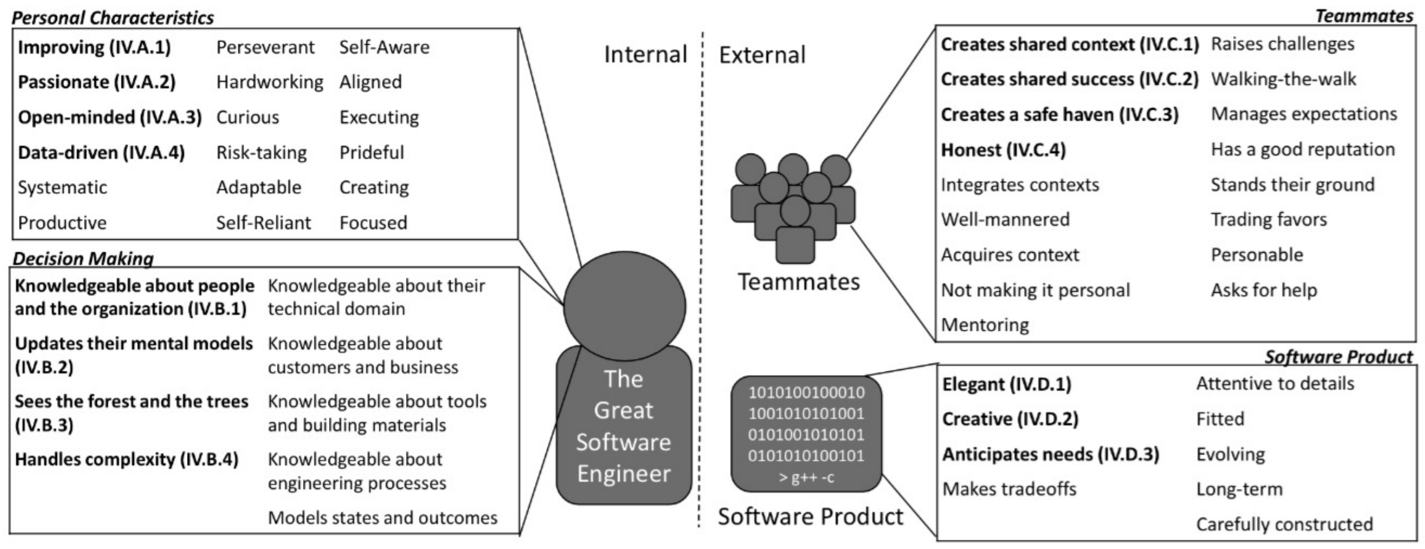

Personal Characteristics
Improving (IV.A.1) Perseverant Self-Aware
Passionate (IV.A.2) Hardworking Aligned
Open-minded (IV.A.3) Curious Executing
Data-driven (IV.A.4) Risk-taking Prideful
Systematic Adaptable Creating
Productive Self-Reliant Focused
Decision Making
Knowledgeable about people and the organization (IV.B.1)
Knowledgeable about their technical domain
Updates their mental models (IV.B.2)
Knowledgeable about customers and business
Sees the forest and the trees (IV.B.3)
Knowledgeable about tools and building materials
Handles complexity (IV.B.4)
Knowledgeable about engineering processes
Models states and outcomes
Internal
External
The Great Software Engineer
Teammates
1010100100010
1001010101001
0101001010101
0101010100101
> g++ -c
Software Product
Teammates
Creates shared context (IV.C.1) Raises challenges
Creates shared success (IV.C.2) Walking-the-walk
Creates a safe haven (IV.C.3) Manages expectations
Honest (IV.C.4) Has a good reputation
Integrates contexts Stands their ground
Well-mannered Trading favors
Acquires context Personable
Not making it personal Asks for help
Mentoring
Software Product
Elegant (IV.D.1) Attentive to details
Creative (IV.D.2) Fitted
Anticipates needs (IV.D.3) Evolving
Makes tradeoffs Long-term
Carefully constructed

Fig. 1. Model of attributes of great software engineers, with attributes we discuss in detailed in bold.

Creates shared context

"Most compellingly relate the value of that abstraction as it goes to non-abstract to very abstract to each person... empathize with your audience... get them to get it." -SDE2, Windows

Li, Paul Luo, Amy J. Ko, and Jiamin Zhu. "What makes a great software engineer?." 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 1. IEEE, 2015

WE'RE GOING TO TRY SOMETHING CALLED AGILE PROGRAMMING.
THAT MEANS NO MORE PLANNING AND NO MORE DOCUMENTATION. JUST START WRITING CODE AND COMPLAINING.
I'M GLAD IT HAS A NAME.
THAT WAS YOUR TRAINING.
www.dilbert.com scottadams@aol.com
11-26-07 ©2007 Scott Adams, Inc./Dist. by UFS, Inc.