The Business Value of a Good API

Sponsored by: The American East Const Association of America ®

Bob Steagall
Maryland C++ User Group
2020-01-08

KEWB
COMPUTING

Software Failures

- January 2018, Tricentis' *Software Fail Watch* documents 606 software failures
  - 3.6 billion people affected
  - $1.7 trillion lost revenue
  - Software failures resulted in 268 years of downtime
  - The number of reported failures was 10 percent higher in 2017 than in 2016
  - Retail and consumer technology industries experienced the most software failures of any industry analyzed

Expensive Software Failures

- Ariane 5 explodes at launch ($370m)
  - double to int16 conversion
- Mars climate orbiter burns up in space ($235m)
  - Imperial to metric conversion
- Knight Capital Loses $460m in 45 minutes
  - Dead code, test flag accidentally left enabled
- Radiation overdoses from Therac-25 cause death
  - Race condition

Warning! Warning! WARNING!

Unsupported opinions…

… and unjustified conjecture ahead!

What is an Interface?

- From Wikipedia:

  In computing, an *interface* is a shared boundary across which two or more separate components of a computer system *exchange information*. The exchange can be between software, computer hardware, peripheral devices, humans and combinations of these.

- Alternatively:

  A well-defined location for information exchange.

What is an API?

- **A**pplication **P**rogramming **I**nterface
  - A set of clearly defined methods of communication between components

- An API represents a **contract** between a component and its users
  - The API specifies expected behavior by both parties
  - The library follows the rules on the implementer side of the contract
  - The developer follows the rules on the user side of the contract

- An API can have multiple implementations
  - Consider the Standard C++ Library…
  - …Or many domain-specific libraries

What is an API?

- In C++, the terms “API” and “interface” are highly overloaded
  - The public interface exported by a **library**
  - The public interface exported by a module or subsystem
  - The public and protected interfaces provided by a class
  - The collection of function signatures from an overload resolution set
  - The interface from a collection of ordinary functions
  - The signature of a single function or macro
  - Messaging protocols
  - And so on, and so on, etc., etc., etc., …

What is an API?

- For the purposes of this talk, “API” will mean…

  The non-private interface exported by a library or a framework, as expressed in its headers, source, executables, link libraries, and documentation

- For the purposes of this talk, “interface” will mean…

  A discrete and well-defined subset of an API – function/template/macro signatures, constants, variables, and types

Reasons to Use/Create an API

- Higher quality code – using a good API
  - Hides implementation details
  - Supports implementation change and optimization
  - Promotes modularity
  - Reduces code duplication
  - Increases product longevity
- Promotes code reuse
- Promotes parallel development
- Promotes better products

Reasons to Avoid Using an API

- Domain mismatch
  - Doesn’t quite solve your problem; or, does so in an awkward way

- Poor/no documentation
  - Understanding becomes expensive

- Closed/proprietary source code (i.e., headers + binary libs + docs)
  - Prevents quick reactions to problems

- License restrictions
  - An API’s license restrictions may not conform to your requirements

- Doesn’t meet requirements
  - Incorrect, inefficient, poor support, wrong platforms/compilers, etc…

Creating an API is Different than Building an Application

- An API is designed for developers
  - Developers are **not** ordinary users…
- Multiple products may (will) share a given API
  - Useful APIs tend to have long life
- Backward compatibility must be a high priority
  - Breaking changes must be very carefully considered and managed
  - Robust change control is **critical** to maintaining quality (and your sanity)
- APIs have unique maintenance challenges
- Good documentation is ***extremely*** important
- Extensive testing, preferably automated, is ***extremely*** important

Creating an API is More **Dangerous** than Writing an Application

multiplier

multiplier

- Remember, APIs have a multiplier effect

multiplier

multiplier

The Business Value of APIs

Why are So Many APIs So Awful?

- **Bad APIs are Easy**
  - For every right way to do something, there are many more wrong ways
  - The wrong way is inevitably easier and faster
  - Less expensive to create (dollars / time)
  - Path of least resistance

- **Good APIs are Hard**
  - Finding the right way to do something can be time-consuming
  - The right way is usually more expensive to build (dollars / time)
  - More unpredictable schedules, especially green-field API development
  - More subject to managerial skepticism (or disdain)

Technical Debt

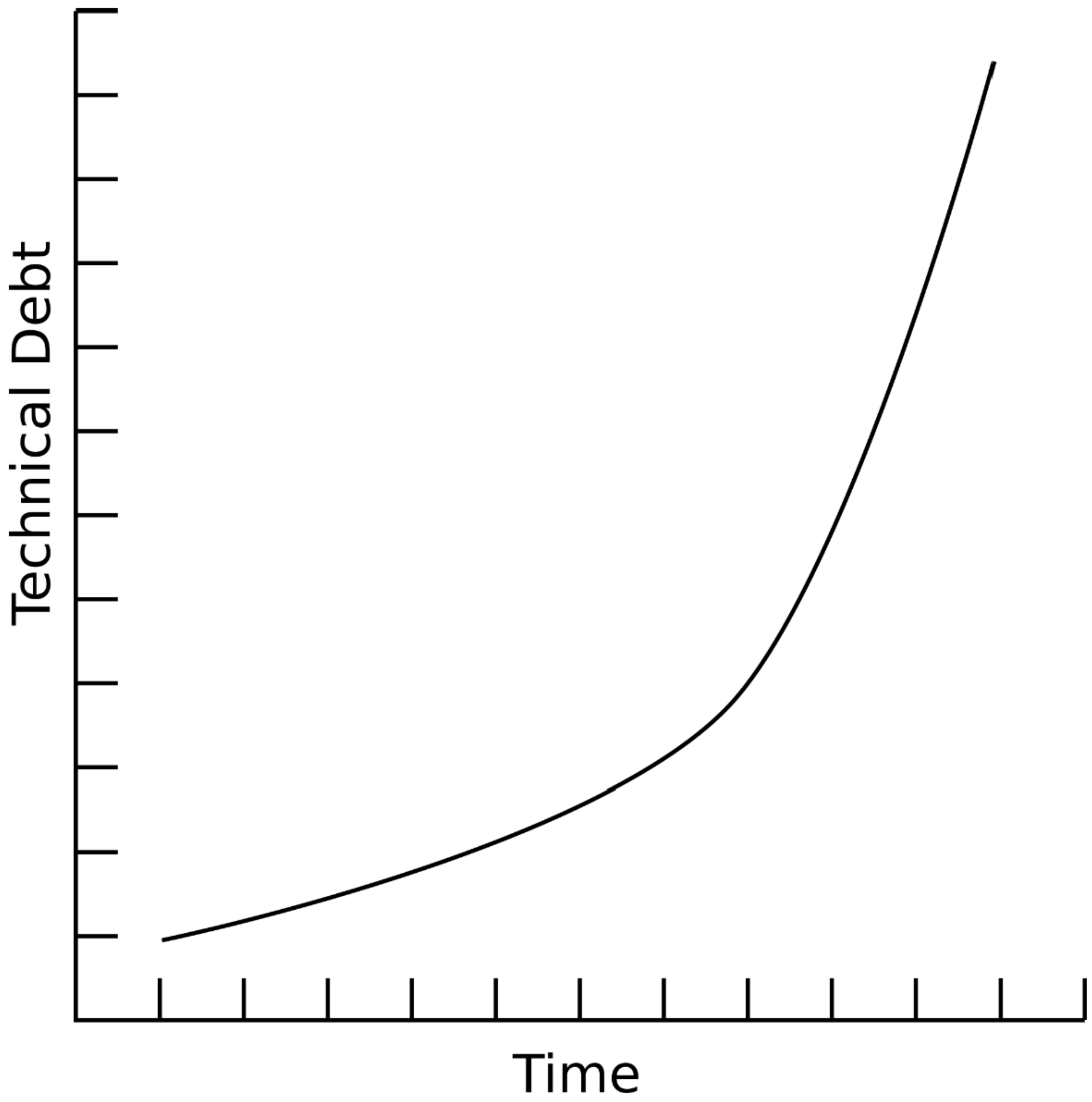
- What is it?

  The cost of additional future rework caused by taking an easy way out now instead of a better solution that would take longer.

- A term originally coined by Ward Cunningham
  - Intended as a metaphor to capture trade-offs for short-term progress to be repaid over the long term by refactoring

- Very much like monetary debt
  - Grows non-linearly, almost exponentially - accumulating “interest”
  - Increases time to add new features
  - Eventually, the cost of adding features is greater than the cost of a rewrite
  - Leads to **software entropy**

Technical Debt Grows Quickly Over Time



From *Building Software Capital*
David Sankel
CppCon 2016

Technical Debt – the Kitchen Metaphor – Green Field

Technical Debt – the Kitchen Metaphor – Release 1

Technical Debt – the Kitchen Metaphor – Release 2

Technical Debt – the Kitchen Metaphor – Release N

Causes of Technical Debt

- Insufficient up-front definition
- Last-minute specification changes
- Business pressures
- Changes in business needs
- Lack of process
- Lack of understanding
- Tightly-coupled components
- No testing or test suite
- Little or no documentation
- Little or no collaboration
- Little or no version control
- Delayed refactoring
- Lack of alignment to standards
- Lack of ownership
- Poor technical leadership
- Poor business leadership

Software Entropy

- What is it?

  The second law of thermodynamics, in principle, states that a closed system's disorder cannot be reduced, it *can* only remain unchanged or increase. A measure of this disorder is **entropy**. This law also seems plausible for software systems; as a system is modified, its disorder, or entropy, tends to increase. This is known as **software entropy**.

- Alternatively,

  Software entropy is the tendency for software, over time, to become difficult and costly to maintain. A software system that undergoes continuous change, such as having new functionality added to its original design, will eventually become more complex and can become disorganized as it grows, losing its original design structure.

Software Entropy Symptoms

- Feature Creep

  Ongoing expansion or addition of new features in a product that extend beyond the basic function of the product.

- Gold Plating

  The addition of any feature not considered in the original project scope or product description at any point of the project; adding unnecessary, frivolous features.

- Brittleness

  The increased difficulty in fixing older software that may appear reliable, but fails badly when presented with unusual data or is altered in a seemingly minor way.

Software Entropy Symptoms

- Bloat

  The process whereby successive versions of a computer program become perceptibly slower, use more memory, disk space or processing power, or have higher hardware requirements than the previous version.

- Bit Rot

  The slow deterioration of software performance over time, or its diminishing responsiveness, that will eventually lead to that software becoming faulty, unusable, and in need of upgrade.

Software Capital

- What is it?

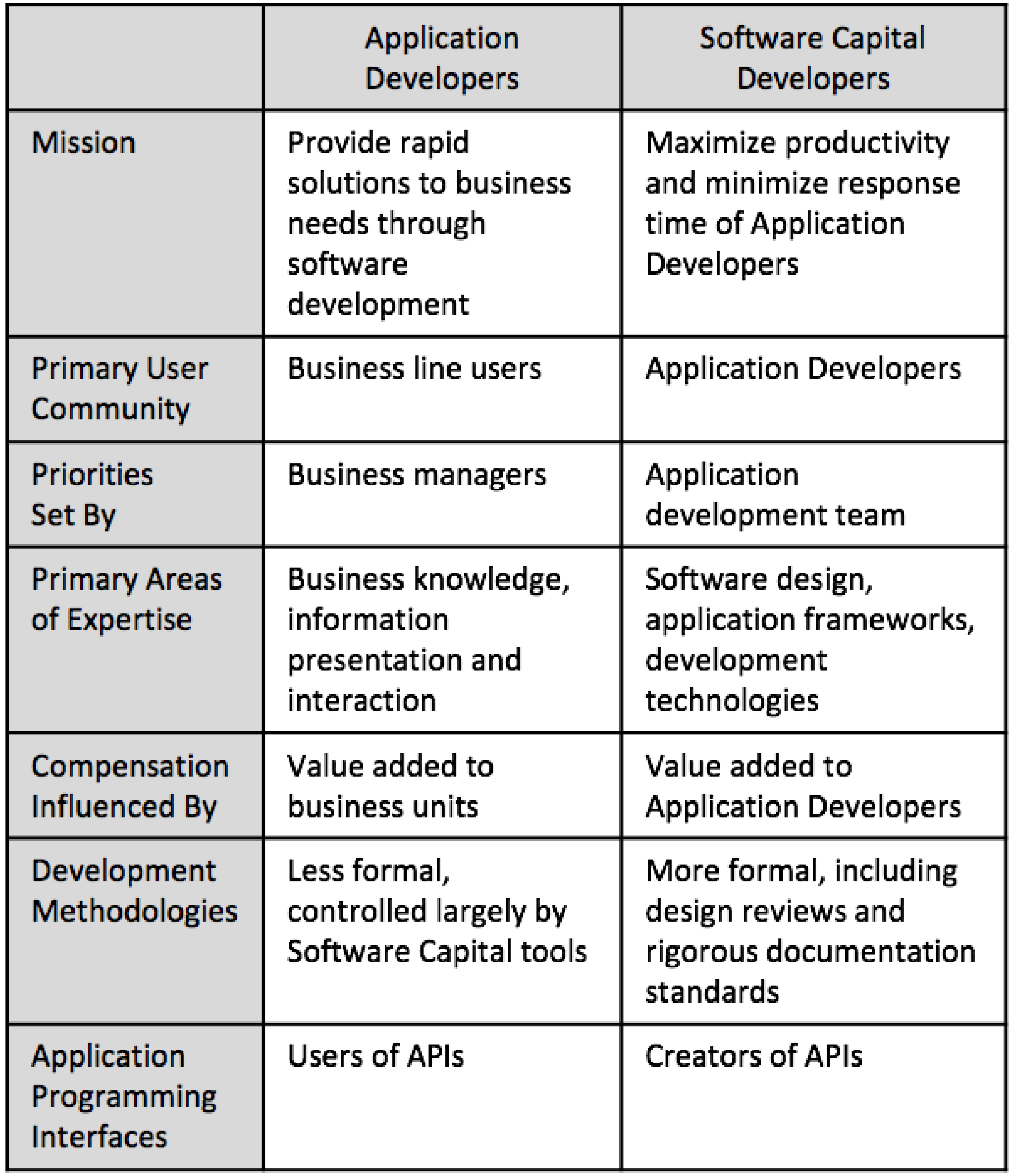
  The cumulative technology that can be re-deployed to new situations.

- A term coined by Dean Zarras
  - Intended to reflect software as an asset rather than a liability

- Done correctly, software **capital**
  - Becomes a toolset for application developers to more rapidly respond to business needs
  - Can raise the starting point from which a solution begins
  - Reduce time to market
  - Pays “dividends” rather than costing “interest”
  - **Is the opposite of technical debt**

Software Capital

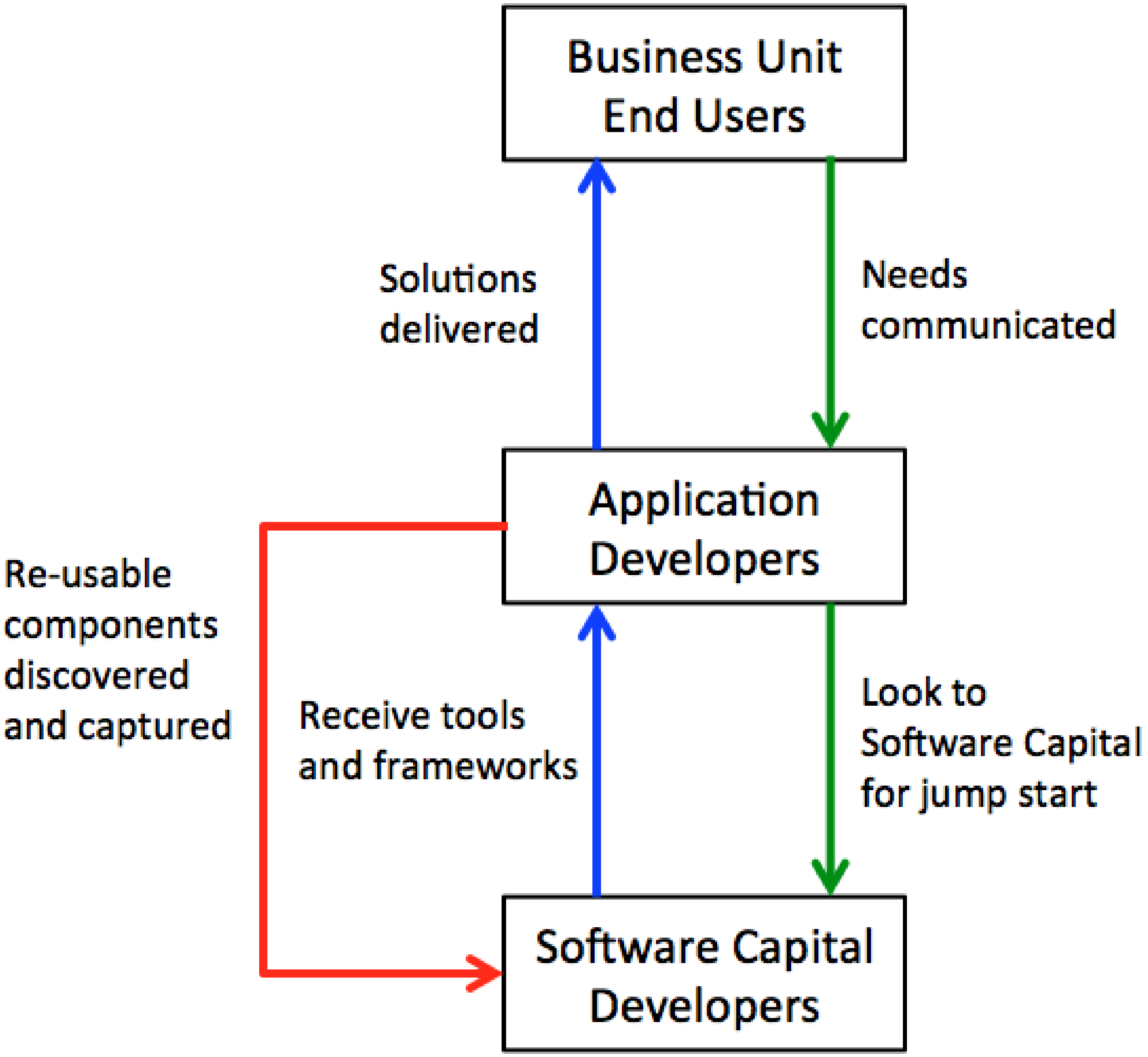
- Two separate development teams
  - Application developers and software capital developers
- Application developers serve the business units directly
- Software capital developers provide the foundation used by the application developers
  - Enhance product quality and developer productivity
  - Promote enterprise reuse
- Both sides leverage their strengths
  - Application developers – business rules and domain knowledge
  - Software capital developers – good software design and engineering knowledge



	Application Developers	Software Capital Developers
Mission	Provide rapid solutions to business needs through software development	Maximize productivity and minimize response time of Application Developers
Primary User Community	Business line users	Application Developers
Priorities Set By	Business managers	Application development team
Primary Areas of Expertise	Business knowledge, information presentation and interaction	Software design, application frameworks, development technologies
Compensation Influenced By	Value added to business units	Value added to Application Developers
Development Methodologies	Less formal, controlled largely by Software Capital tools	More formal, including design reviews and rigorous documentation standards
Application Programming Interfaces	Users of APIs	Creators of APIs

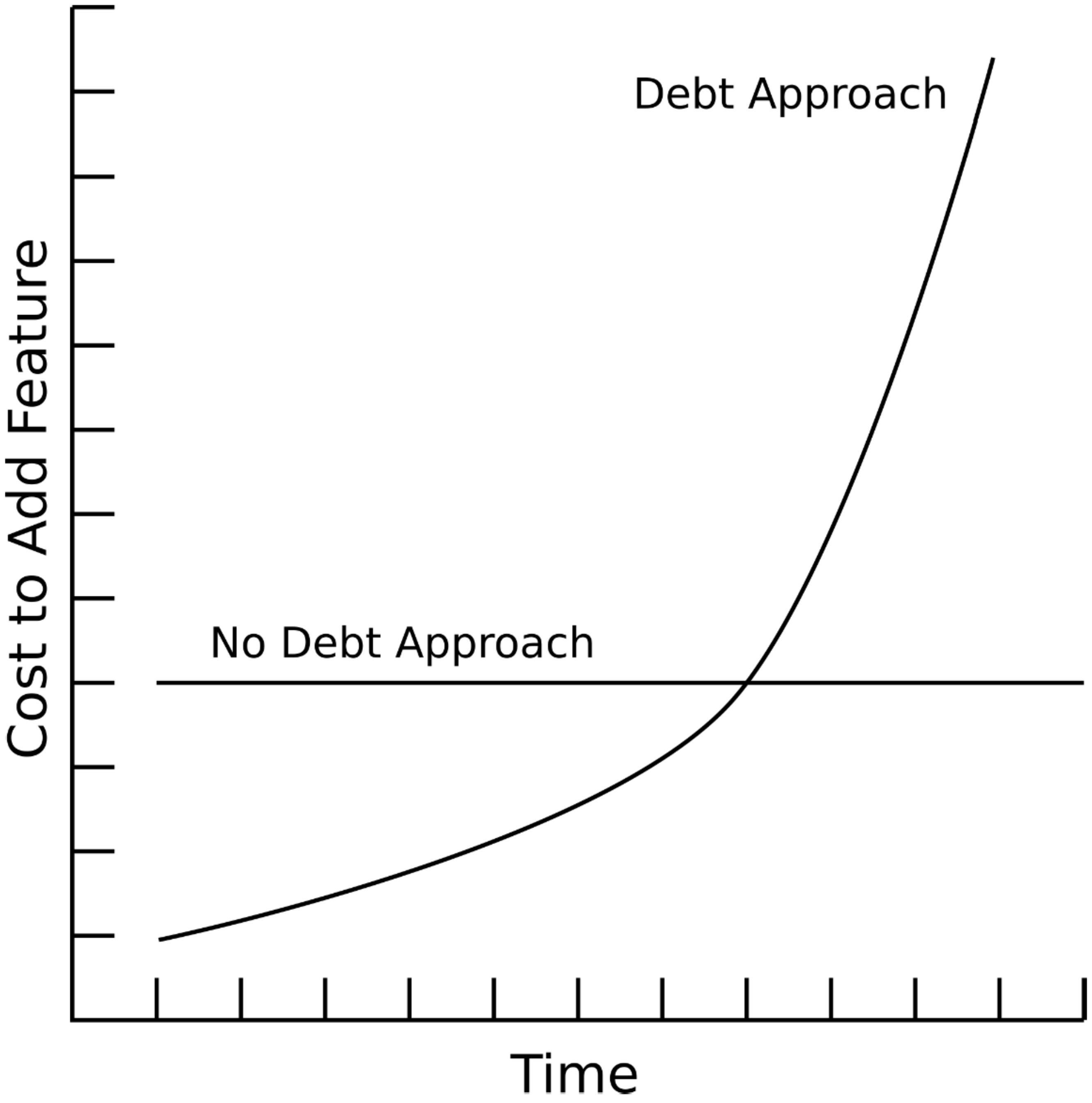
From *Software Capital — Achievement and Leverage*
Dean Zarras
Hackernoon 2016

Software Capital in Action


Business Unit End Users
Solutions delivered
Needs communicated
Application Developers
Re-usable components discovered and captured
Receive tools and frameworks
Look to Software Capital for jump start
Software Capital Developers

From *Software Capital — Achievement and Leverage*
Dean Zarras
Hackernoon 2016

Technical Debt -vs- Software Capital Over Time



From *Building Software Capital*
David Sankel
CppCon 2016

Summary: Technical Debt -vs- Software Capital

Technical Debt (Bad API)	**Software Capital (Good API)**
Easy to create	Difficult to create
Cheap to create	Expensive to create
Reused “under the gun”	Voluntarily Reused
Narrow focus	Wide Focus
Ugly	Beautiful
Incomplete	Complete
Increases time to market	Decreases time to market
Increases costs in the long term	Reduces costs in the long term

Adapted from *Building Software Capital*
David Sankel
CppCon 2016

Evaluating an API

On Clean Code

I like my code to be **elegant** and **efficient**. The logic should be **straightforward** to make it hard for bugs to hide, the dependencies **minimal** to ease maintenance, error handling **complete** according to an **articulated strategy**, and performance close to **optimal** so as not to tempt people to make the code messy with unprincipled optimizations. **Clean code does one thing well**.

Bjarne Stroustrup

Clean Code

API Quality Goals

- Readability
- Understandability
- Conceptual Integrity
- Mnemonic Consistency
- Correctness
- Testability
- Performance
- Increased **software capital**

Assessing the Quality of an API

- How do we know if an API is good or bad?

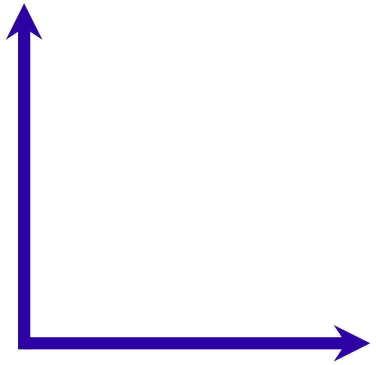
- Are “good” and “bad” reasonable and/or relevant adjectives?
  - Effective / ineffective?
  - Desirable / undesirable?

- For convenience, I will use the terms “good” and “bad”
  - **Good** – an API that mostly increases **software capital**
  - **Bad** – an API that mostly increases **technical debt**

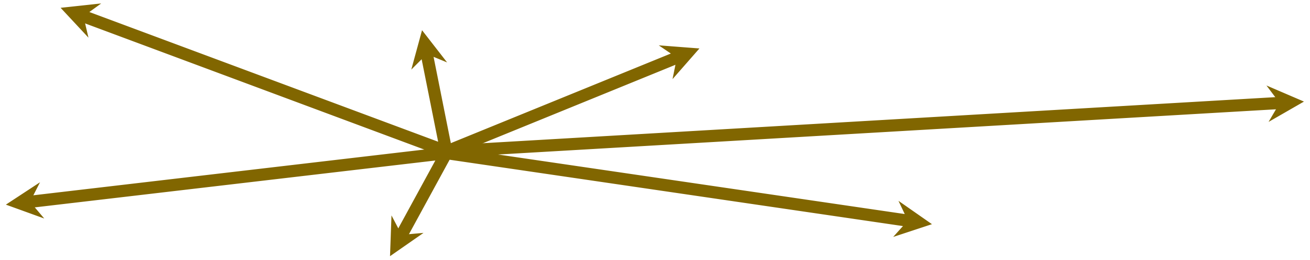
Warning! Warning! Warning!

- The characterizations that follow are somewhat intuitive, but…
  - They are **not** orthogonal
  - They are **not** quantitative
  - They are **not** exact

What I'd like to give you…



…What you're going to get



Characteristics of a Bad API – General

- Is incomplete – it doesn't do its job
- Is unstable / untested / undocumented
- Has inconsistent design, structure, or naming
- Is unnecessarily complex

Characteristics of a Bad API – General

- Is unnecessarily difficult to learn
- Is written from the perspective of the implementer
    - Inconveniences the user to benefit the implementer
    - Requires repetitive boilerplate code to be used
- Is indecisive (passes the buck)
    - Example: using enumerations instead of types to specify behavior
- Unnecessarily exposes its implementation

Characteristics of a Bad API – Design Smells

- Missing abstraction
  - Clumps of data or encoded strings are used instead of creating an abstraction
- Multifaceted abstraction
  - When an abstraction has multiple responsibilities assigned to it
- Deficient encapsulation
  - When the declared accessibility of one or more members of an abstraction is more permissive than what is actually required
- Unexploited encapsulation
  - When client code uses explicit type checks instead of exploiting the variation in types already encapsulated within a hierarchy

Characteristics of a Bad API – Design Smells

- Duplicate abstraction
  - When two or more (different) abstractions have confusingly similar names
- Duplicate implementation
  - When two or more abstractions with different names have identical implementations
- Broken modularization
  - When data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions
- Insufficient modularization
  - When an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both

Characteristics of a Bad API – Design Smells

- Cyclic dependencies
  - When two or more abstractions depend on each other directly or indirectly
- Unfactored hierarchy
  - When there is unnecessary duplication among types in a hierarchy
- Broken hierarchy
  - When a base type and its derived type do not share an “IS-SUBSTITUTABLE-FOR” relationship, resulting in broken substitutability
- Cyclic hierarchy
  - When a base type in a hierarchy depends on any of its derived types

Characteristics of a Good API - General

- Solves the problem correctly
- Provides satisfactory performance
- Composed of manageable pieces
- Built with common idioms and patterns
- Decisive (does **not** pass the buck)

Characteristics of a Good API - General

- Designed from the perspective of the user
- Behaves consistently, follows the principle of least surprise
- Explicitly states and observes the contracts to which it will adhere
- General-purpose APIs should be “policy-free”
- Special-purpose APIs should be “policy-rich”

Characteristics of a Good API – Usability

- **Stable**, **documented**, and **tested**
  - Behaves consistently at both compile and run time
  - Documentation is complete, especially with regard to error handling
  - Reasonable code coverage, test result transparency

- Minimally Complete
  - Doesn’t over-promise
  - Uses both types of polymorphism wisely
  - Doesn’t impose inconvenience on the user
  - Has just the right amount of convenience interfaces

Characteristics of a Good API – Usability

- Ergonomic
  - Easy to read, understand, learn, and memorize
  - Discoverable (mnemonic consistency)
  - Easy to use correctly, difficult to use incorrectly
  - Consistent in structure, design, naming, usage
  - Orthogonal (mostly – convenience interfaces reduce orthogonality)
  - Helps the caller/user
  - Provides required extensibility
  - Provides robust resource management
  - Platform independent

Characteristics of a Good API – Structure

- Loosely coupled
  - Modular and layered
  - Coupling is by name only
  - Redundancy is intentional and minimal
  - Provides manager classes, callbacks, observers, and notifications as appropriate

- Hides implementation details judiciously
  - Uses physical hiding – declaration versus definition
  - Uses logical hiding – encapsulation
  - Hides implementation – member data / functions / classes

A Simple Process for API Design

Guiding Principles – The Four C’s

- Clarity of Purpose
  - Understand the goals and why you’re seeking to achieve them

- Clarity of Thought
  - Understand the problem and your solution to it

- Clarity of Expression
  - Clearly describe the problem and your solution to others

- Neatness Counts!
  - The presentation of your solution is just as important as the solution itself
  - A well-presented solution promotes understandability, which leads to everything

On Users

- I see three categories of users that have different expectations:
- **Casual users** want maximum ease of use, minimal learning curve, good experience and immediate productivity right out of the box
- **Power users** want ease of use combined with some customizability in search of higher performance
- **Expert users** want maximum customizability to solve highly specialized problems and/or highest performance
- Understand which categories users the API is targeting!

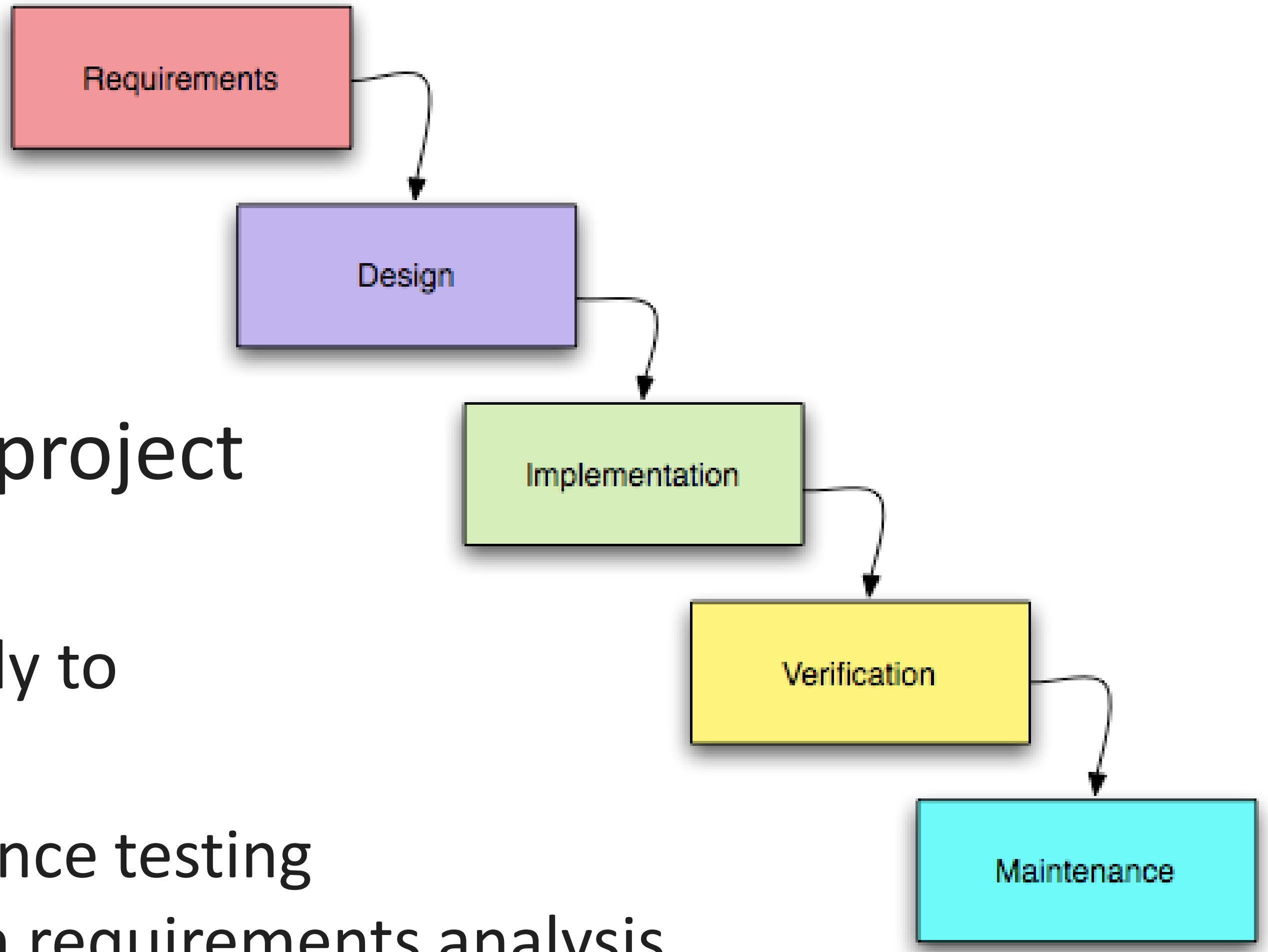
On Process

- A good process increases productivity
  - Uses objective metrics to track project status
  - Helps control the complexity of products and projects
  - Selection of tools and technologies
  - Defines how, when, and by whom development activities are carried out

- A good process improves software quality
  - Uses recognized best practices
  - Promotes software asset and tool standardization, leading to re-use
  - Enforces consistent and transparent change management

The Classic Waterfall Process

- Logical and simple
- Appropriate for small or trivial projects
- Suffers from a fatal flaw for any non-trivial project
  - Delays the recognition and mitigation of risks
  - As project proceeds, it becomes increasingly costly to correct defects from earlier phases
  - A requirements defect discovered during acceptance testing costs many times more to fix than if it is caught in requirements analysis



Requirements

Design

Implementation

Verification

Maintenance

Iterative Process Overview

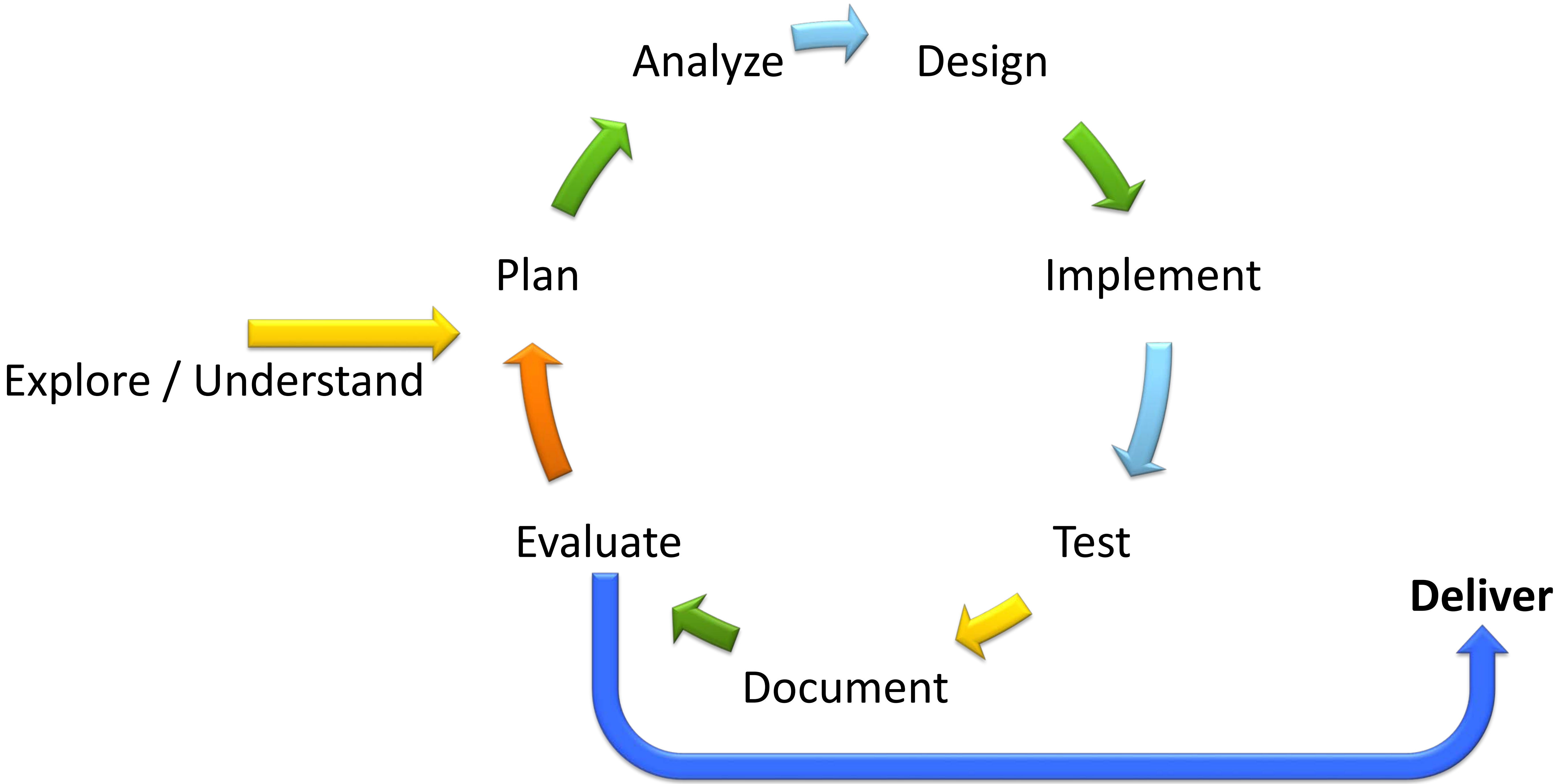
- So how do you improve the Waterfall model?

- Make it iterative and incremental
  - Add a subset of desired new capabilities in each iteration
  - Review, adjust, correct requirements multiple times
  - Review, adjust, correct design multiple times
  - Address highest risk items first
  - Build, integrate, test, and document continuously over lifecycle
  - Identify opportunities for component re-use

Steps in the Process – Creative Acts

- Gather and analyze requirements
- Design/refine some subset of the API
- Implement/refine the code for that subset
- Write/refine unit tests for that subset
    - Run all the unit tests and make sure they **all** pass
- Create/refine documentation for that subset

The Simple API Design Process as a Picture



The Simple API Design Process As Code – In Theory

```
void MyApi::Build(VersionType version)
{
    ExploreAndUnderstandTheProblem();

    while (!VersionComplete())
    {
        DoSomeIterationPlanning();
        AnalyzeReqs();
        RefineDesign();
        WriteCode();
        WriteAndRunTests();
        WriteDocs();
        ReviewAndEvaluate();
    }

    Deliver(version);
}
```

Steps in the Process – Creative Acts

- Gather and analyze requirements
- Design/refine some subset of the API
- Implement/refine the code for that subset
- Write/refine units tests for that subset
- Run all the unit tests and make sure they **all** pass
- Create/refine documentation for that subset

In real projects, these activities are almost always concurrent!

The Simple API Design Process As Code – The Reality

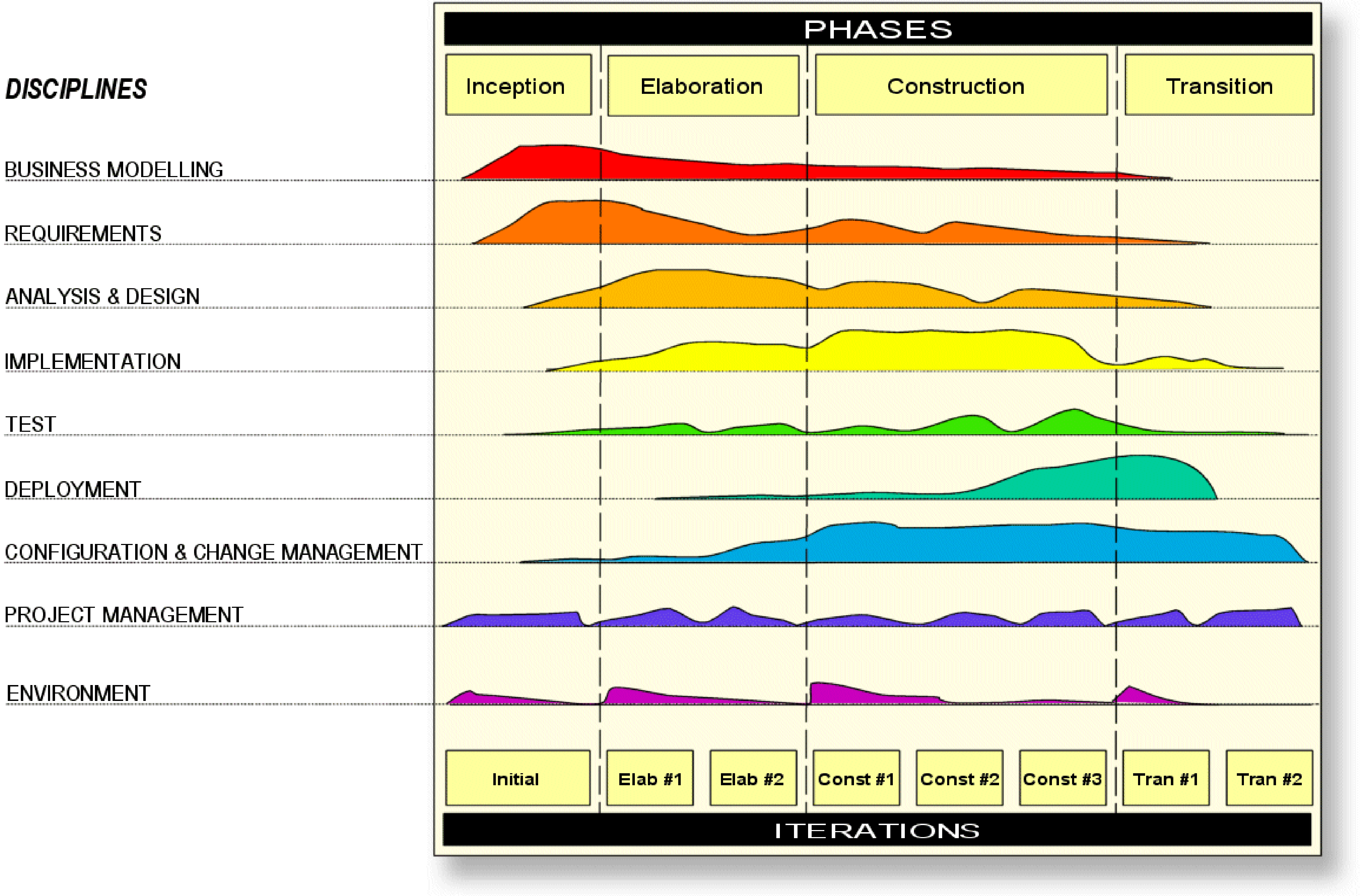
```
bool MyApi::Build(VersionType version)
{
    BeginIncurringCosts();
    ExploreAndUnderstandTheProblem();

    while (CostStillAcceptable() && !VersionComplete())
    {
        DoSomeIterationPlanning();

        auto  req = std::async(std::launch::async, &MyApi::AnalyzeReqs, this);
        auto  dsn = std::async(std::launch::async, &MyApi::RefineDesign, this);
        auto  cod = std::async(std::launch::async, &MyApi::WriteCode, this);
        auto  tst = std::async(std::launch::async, &MyApi::WriteAndRunTests, this);
        auto  doc = std::async(std::launch::async, &MyApi::WriteDocs, this);

        WaitFor(req, dsn, cod, tst, doc);
        ReviewAndEvaluate();
    }
    return (CostStillAcceptable() && OkToShip()) ? Deliver(version), true
                                                 : UpdateResume(), false;
}
```

A Quick Aside – The Unified Process



Steps in the Process – Explore and Understand

- Understand the problem
  - Describe the problem in writing and get someone to read it

- Define the scope of the API
  - Specify what aspects of the problem the API will address

- Define the completion criteria for this version
  - Specify what “done” means
  - Specify quality metrics and targets

- Define the tools and conventions that will be used
  - Development environment
  - Coding standards and guidelines

Steps in the Process – Gathering Requirements

- **F**unctional requirements
  - Features, capabilities, reusability, security
- **U**sability requirements
  - Aesthetics, learnability, consistency, documentation
- **R**eliability requirements
  - Availability, predictability, accuracy, failure rates
- **P**erformance requirements
  - Speed, efficiency, resource usage, throughput, capacity, scalability
- **S**upportability requirements
  - Testability, installability, maintainability

Steps in the Process – Gathering Requirements

- Some questions to ask stakeholders about the problem domain
  - What is the problem domain terminology?
  - What, if any, are the important conventions in the problem domain?
  - What mental models do they already have that describe the problem domain?
  - What tasks from the problem domain do they wish to perform with the API?
  - What is an optimal/typical/desired workflow to perform those tasks?

Steps in the Process – Gathering Requirements

- Some questions to ask stakeholders about the product (API)
  - What are all the valid inputs?
  - What are all the expected outputs?
  - What data formats / protocols must be supported?
  - What compilers are expected to be used?
  - Are there any restrictions on language features?
  - What are the development environment which the API will be used?
  - What is the expected end user computing environment?
  - What are the expected resource limits?

Steps in the Process – Gathering Requirements with Use Cases

- Use cases describe API behavior based on interactions with external entities, such as other software or end users
  - They capture who does what, and in what sequence
  - They help describe requirements from the perspective of the user

- Traditional requirements describe a system's capabilities

- Use cases describe how the system interacts with the outside world

- UML analogy
  - static (traditional ←→ class diagram)

    -vs-

    dynamic (use cases ←→ state/sequence diagram)

Steps in the Process – Writing Good Requirements

- Use domain terminology
  - Describe each requirement in the user’s language
- Each requirement should be
  - Unambiguous
  - Consistent with other requirements
  - Complete
- Don’t specify design in requirements
- Use the requirements and use cases to drive development of test assets
- Expect to revise the requirements several times
- Don’t expect to have complete/perfect requirements coverage

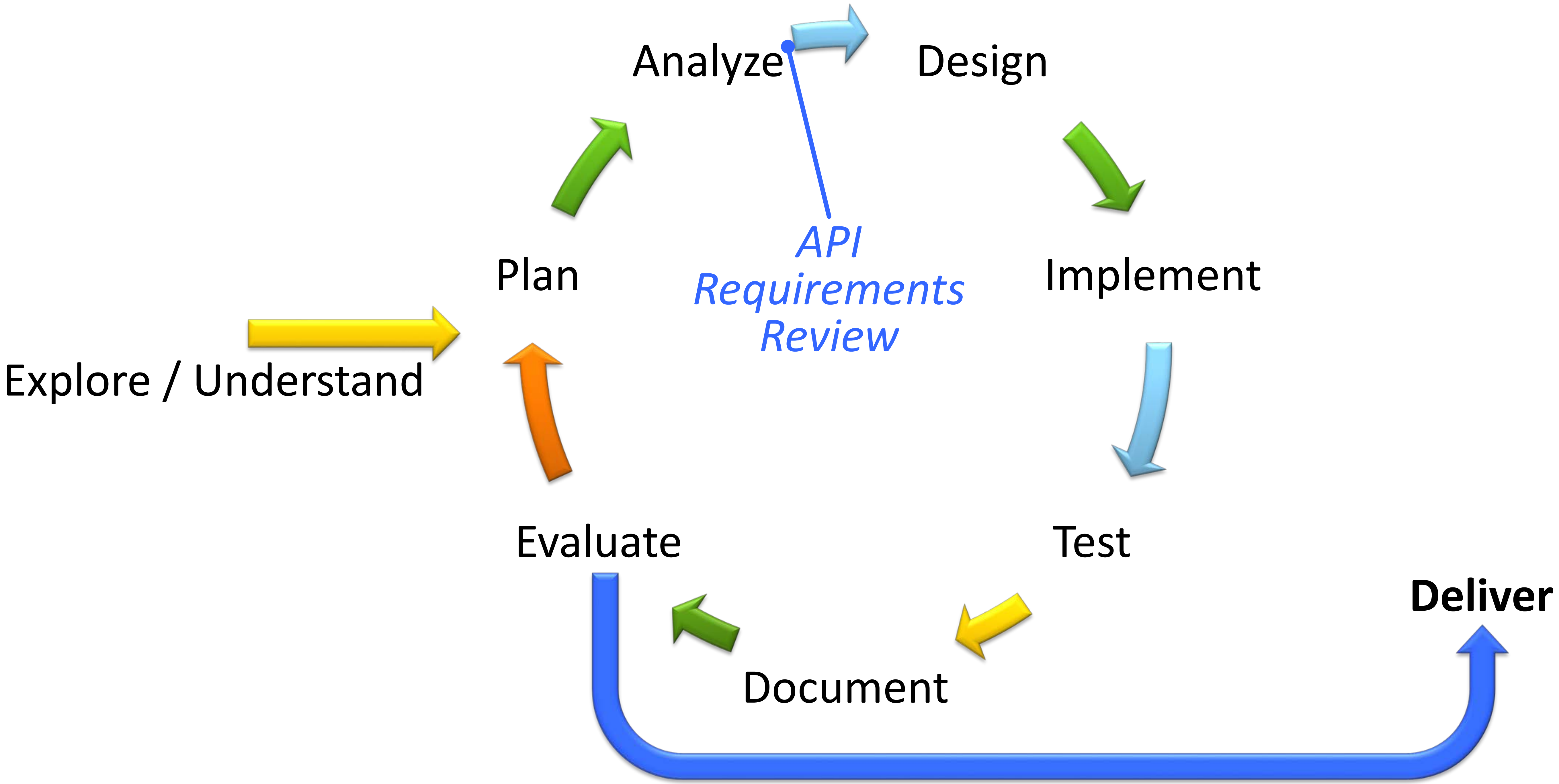
Steps in the Process – Review the Requirements

- It is important to re-assess the requirements at the start of each iteration
  - Stakeholders’ understanding changes; and…
  - External constraints change; meaning…
  - Requirements change!

- Review ensures that the understanding of work to be done is in sync with the stakeholders’ expectations, external constraints, and the requirements

- In a requirements review, try to assess
  - Coverage and content
  - Overall quality
  - Individual quality

The Simple API Design Process as a Picture



Analyze

Design

Implement

Test

Document

Evaluate

Plan

Explore / Understand

API Requirements Review

Deliver

Steps in the Process – Refine the Design

- Design traditionally consists of two major activities
  - Architectural design
  - Detailed design

- Architectural design
  - Creating a description of the top-level structure and organization of the API
  - Very important for composing documentation

- Detailed design
  - Creating a description of individual components in sufficient detail such that they can be implemented
  - Can be useful…. or not, depending on the size of the system

Steps in the Process – Developing an Architecture

- Software architecture describes at a high level the structure of the system
  - The top-level objects and their relationships to each other

- There is rarely a single right architecture for a given problem
  - N architects will very likely find K*N different solutions (where K > 1)

- Objective is to create an API design that meets the quality objectives while fulfilling the requirements and resolving conflicts.

- In general, you will
  - Analyze the requirements that affect the architecture
  - Identify and account for architectural constraints
  - Invent the most important objects in the system and their relationships
  - Document and communicate the architecture

Steps in the Process – Understand Architectural Constraints

- There will always be factors that constrain the resulting architecture

- Organizational
  - Budget, schedule, team size, development process, management's priorities, …

- Environmental
  - Hardware, platform, development tools, protocols, 3rd-party APIs, …

- Operational
  - Performance, resource limits, reliability, concurrency, I18N, …

- Many, but not all, constraints come from the requirements

Steps in the Process – Techniques for Inventing the Key Objects

- Use natural language
  - Nouns tend to represent objects
  - Verbs tend to represent functions
  - Adjectives and possessive nouns tend to represent attributes

- Group by properties
  - Grouping objects that have similar properties or qualities
  - Try to find categories such that an object either belongs or does not belong

- Group by behaviors
  - Try to determine all the behaviors in the system, then group into subsystems
  - Derive the set of objects by finding the initiators and recipients of each behavior

Steps in the Process – Consider Architectural Patterns

- Like software design patterns at the component level, there are higher-level patterns for large-scale design, for example
    - Structural patterns – layers, pipes and filters, blackboard
    - Distributed systems – client/server, 3-tier, peer-to-peer, broker
    - Adaptable system – micro-kernel, reflection
    - Interactive systems – model/view/controller, model/view/presenter

- One important aspect of these patterns is the lack of cyclic dependencies

- Another important aspect is real-world experience

Steps in the Process – Scenario-Based Design

- Many frameworks often service a small set of common *usage scenarios* that employ only a subset of the entire framework
- Critical to investigate the most common scenarios and optimize them from the API user's perspective
- After identifying the major nouns and verbs in the important scenarios:
  - First, write pseudo-code to quickly sketch out usage
  - Then write **real code**, the same code that your API users would write
  - Think critically about that code and ask yourself if it makes sense
- Scenario explorations can (should!) feed back into the overall design

Steps in the Process – Communicating the Architecture

- Documenting the architecture is critical for understandability
  - Provides a roadmap for implementers (important for remote teams)
  - Promotes peer review of candidate architectures
  - Forms a basis for good user documentation

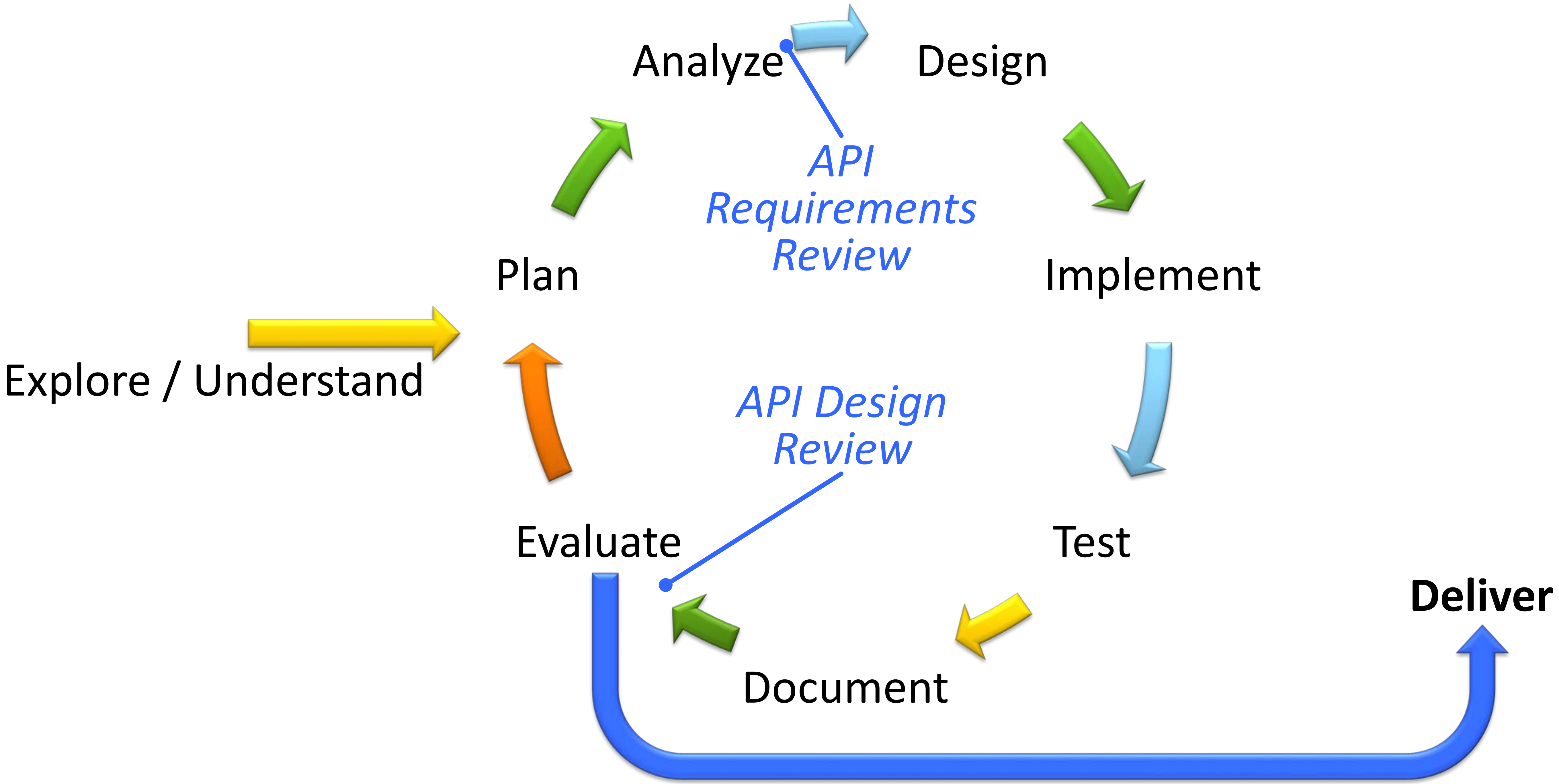
- Always describe your API's architecture and design rationale

- Agile processes de-emphasize documents, but…
  - The high-level requirements and architectural specifications are important enough to be captured somewhere
  - Capture lower-level requirements and design if time and budget allow, or if required

Steps in the Process – Review the Design

- It is important to re-assess the API's design near the end of each iteration
- Design review ensures that the understanding of work to be done is in sync with the stakeholders' expectations, external constraints, and the requirements
- In a review, try to gauge
  - Adherence to specified design considerations
  - Requirements traceability
  - Consistency
  - Performance
  - Maintainability
  - Compliance
  - Understandability

The Simple API Design Process as a Picture



Steps in the Process – Evaluate and Plan

- Evaluate progress
  - Review the API
  - Does the API provide the promised functionality?
  - Has the API has met its quality goals?
  - Are there any aspects of the API that should be refined in order to maintain or improve customer satisfaction (within reason)?

- Plan the next iteration
  - Is there any unfinished work to be completed?
  - Are there any major requirements changes?
  - Are there any new constraints on the project?
  - What is the scope of the next iteration?

Fitting API Design into Your Overall Process

- Discuss with customers and colleagues
  - If you can’t convince your customers or colleagues that your API idea is a good one, it may be time for a re-think (or maybe not)
  - If you **can** convince your customers **and** your colleagues, then you **may** be on the right track
- Make the business case to your manager / management
  - Adapt your pitch according to their pain points and customer feedback
  - Clue them in to technical debt and software capital
  - Provide realistic estimates of up-front costs (dev time, cap-ex, etc.)
  - Provide realistic estimates of risks (schedule slip, cost overruns, etc.)
  - Provide realistic estimates of benefits (reduced costs, sales growth, etc.)
- Be prepared for disappointment – **but keep trying**!

Fitting API Design into Your Overall Process

- Adapt the simple API design process to your employer's process
  - Try to fit the iterative spirit into your company's practices
  - Try to create written SWE artifacts, even if only for your usage
  - Try to ask the important questions at every stage

- Solicit feedback from stakeholders and colleagues
  - Accept feedback graciously, especially negative feedback
    - If reviewers are "difficult", consider Crocker's Rules
  - Consider all feedback carefully and be prepared to change your API if the feedback raises an important point
  - At the same time, remember the goals of design consistency, conceptual integrity, and mnemonic consistency

Process Recommendations

Integrate continuously

Always make sure that you’ve merged all changes from your master branch into your development workspace before creating a pull request for (or merging) your changes.

Leave things better than you found them

Before merging, always make sure that all of your unit tests pass and that the overall pass rate is equal to, or better, than that of the master branch.

Process Recommendations

Always design for change

Try to anticipate the ways in which requirements might change over an appropriate time horizon, and try to build in reasonable facilities to accommodate those changes.

Remember the Four C’s

If you can’t explain it or justify it in a clear and concise way, consistent with the rest of the API, don’t do it.

Don’t take it personally

People are what they are…

Process Recommendations

Use scenario-based design

Early in the product lifecycle, after you understand the problem and are starting to think about a design, write pseudo-code as a way to help you quickly formulate a design and understand the user's point of view. Then write real code, the same code that your API users would have to write.

Use your own API

The quickest way to uncover the shortcomings in your API is to use it yourself. **If it irritates you, imagine what it will do to others…**

Process Recommendations

Evaluate continuously

Try to internalize the foregoing principles, then frequently ask yourself:

- Does my API exhibit any of the bad characteristics?
  - If so, how do I fix them?
- Does my API exhibit any of the good characteristics?
  - If so, are they good enough, or is there some further improvement?
- Viewed from a high level, is my API meeting the stated API Design Goals?
- Am I observing the Four C’s?

Requirements Elicitation, Requirements Review, and Design Review Checklists

API Requirements Review Checklist

- Requirements content / coverage
  - Are all the tasks the API will perform specified?
  - Are the non-functional requirements stated clearly?
  - Are all the inputs to the API specified, including their accuracy, range of values, frequency, and format?
  - Are all the outputs from the API specified, including their destination, accuracy, range of values, frequency, and format?
  - Are all the external hardware and software interfaces specified?
  - Are all the communication interfaces specified?
  - Are the robustness requirements specified?
  - Are the internationalization and localization requirements specified?

API Requirements Review Checklist

- Requirements content / coverage
  - Are the processing speed and memory usage requirements specified?
  - Are the response time or latency requirements specified?
  - Are the throughput requirements specified?
  - Are the data volume requirements specified?
  - Are the peak or short-term load requirements specified?
  - Are the security/privacy requirements specified?
  - Are the safety requirements specified?
  - Are the supported configurations specified?
  - Are the compatibility requirements specified?

API Requirements Review Checklist

- Overall requirements quality
  - Are the requirements minimally complete?
  - Are the requirements consistent?
  - Are the requirements sufficient?
  - Are the requirements feasible?
  - Are the requirements clearly and appropriately prioritized?
  - Do the requirements avoid specifying the design?
  - Do the requirements adequately address all appropriate exception conditions?
  - Does the set of requirements adequately address known boundary conditions?
  - Can the requirements be implemented within known constraints?
  - Have all functional and non-functional requirements been considered?

API Requirements Review Checklist

- Individual requirements quality
  - Is the requirement precise and unambiguous?
  - Is the requirement stated in as simple or atomic a form as possible?
  - Is the requirement testable and/or verifiable?
  - Is the requirement correct?
  - Is the requirement as modifiable as possible?
  - Is the requirement written using the customer's terminology?
  - Is the requirement acceptable to all stakeholders?
  - Is the requirement a statement of stakeholder need, not a solution?
  - Is the requirement traceable?
  - Is the requirement necessary?

API Design Review Checklist

- General
  - Does the API design support its stated goals?
  - Is the design feasible from a technology, cost, and schedule standpoint?
  - Have known design risks been identified, planned for, or mitigated?
  - Are the methods used to create and capture the design appropriate?
  - Does the level of formality match the API project size and goals?
  - Has the design been refined based on prototyping or implementation feedback?
  - If possible, were proven past designs reused?
  - Does the design support continuing to the next development step?

API Design Review Checklist

- Design considerations
  - Does the design have conceptual integrity?
  - Does the design use standard techniques?
  - Does the design emphasize simplicity over cleverness?
  - Is the design "as simple as possible, but no simpler"?
  - Is the design lean? (i.e., all parts are strictly necessary?)
  - Is the design intellectually manageable?
  - Is the design robust?
  - Does the design create reusable components where appropriate?
  - If appropriate, will the design be easy to port to another environment?

API Design Review Checklist

- Requirements traceability
  - Does the design address all issues from the requirements?
  - Does the design add functionality which was not specified by the requirements?
  - If appropriate, has a requirements traceability matrix been created?
- Completeness
  - Are all of the assumptions, constraints, and dependencies documented?
  - Has a risk plan been made for the parts of the design that may not be feasible?
  - Have any assumptions made due to missing information been documented?
  - Have reasonable alternative designs been considered?

API Design Review Checklist

- Consistency
  - Is the design consistent with its upstream and downstream artifacts?
  - Does the design make sense both from the top down and the bottom up?
  - Is the design consistent with related artifacts?
  - Is the design consistent with the development and operating environments?
  - Are the components comprising the design consistent with themselves?

- Performance
  - Are all performance requirements defined and addressed by the design?
  - If appropriate, are there justifications for design performance?

API Design Review Checklist

- Maintainability
  - Does the design allow for ease of maintenance?
  - Does the design account for future extensions?
  - Will the design resist erosion in its correctness over time?

- Compliance
  - Does the design follow all required standards? (security/privacy/regulatory/etc.)

- Understandability
  - Are the headers (and source, if shipping source) readable? Understandable?
  - Can you effectively describe the design to a customer (or proxy)?

Thank You for Attending!

Talk: https://github.com/BobSteagall

Blog: https://bobsteagall.com

KEWB
COMPUTING