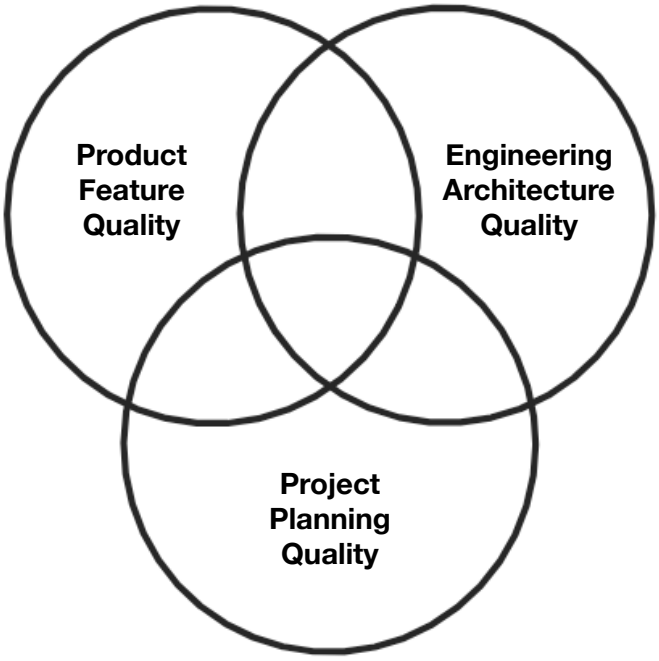
Quality Assurance Philosophy

(Note: This is a philosophy — a collection of ideas I believe have value. This is not a vision of how things must be. Each enterprise is unique, and even the projects within the enterprise can be unique. The value of this philosophy is to help bring focus to a specific vision for an enterprise.)

Quality Ownership

Product
Are we building the right thing?



Engineering
Are we building the thing right?

Program Management
Are we building at a sustainable pace?

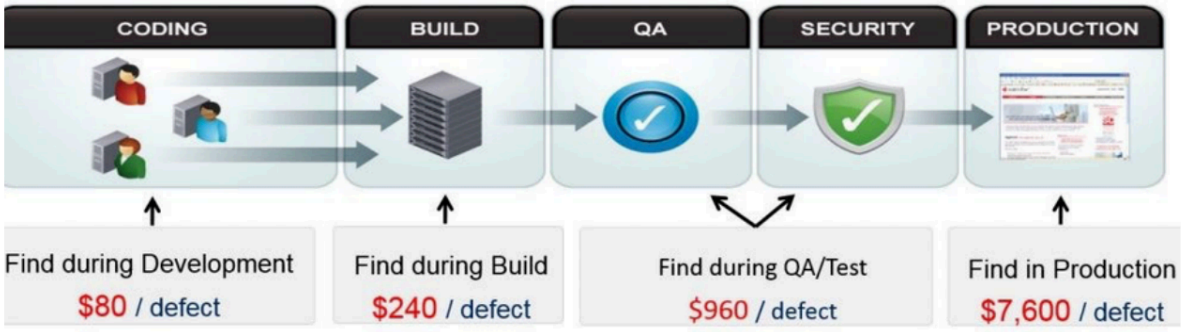
Shifting Quality Left

Benefits of Focusing on Quality Early

The primary benefit of shifting quality left Is the positive impact on those who use the software:

- It is the best overall experience because the software "just works," minimizing the number of bugs that make it into production.
- It results in a high level of trust in the integrity of our products.
- Customers will not be motivated to find another vendor or use a competitor's site.

Another significant benefit of shifting quality left is cost savings. Here are the varying costs of resolving issues based on when they are found (2017 research from The Ponemon Institute):



Shift Left — Product

- Well-defined, detailed acceptance criteria are included for each feature.
- Quality staff can assist with writing these in a natural language that a testing automation framework can interpret.

Shift Left — Program Management

- Analyzing historical feature workflows to improve future feature delivery scheduling.
- Coordinating scheduling across implementation teams during the Design and Analysis phase when interdependencies exist.

Shift Left — Engineering

- Instill Quality Ownership in Software Engineers:
  - Choose to make automated testing coverage a requirement for all new feature implementations.
  - Train new team members so they become Subject Matter Experts in the automated testing for each repository.
- Implement Behavioral Driven Development (BDD — see Agile Alliance Glossary)
  - aka Specification by Example
  - Ties into Shift Left for Product.
- Implement Test Driven Development (TDD — see Agile Alliance Glossary)
  - It is an engineering practice, not a testing technique.
  - It results in a fully automated suite of fast feedback tests.
  - It must focus on business logic to provide the highest value.
- Implement Continuous Delivery/Deployment (DevOps or Site Reliability Engineering):
  - Implement canary deployments. These will prevent disruptions to the deployment environment (production, staging, etc) due to failures.
  - Implement feature toggles:
    - Feature toggles allow for incremental development without exposing partially completed features to customers.
    - Facilitates A/B Testing

Benefits of BDD and TDD

The benefits of implementing BDD and TDD include:

- Finding issues earlier, when they cost significantly less to resolve.
- Delivering projects with higher quality from the start.
- Significantly reducing the risk of projects missing their delivery dates.
- Software developed is easier to maintain and enhance because of the craftsmanship improvements:
  - Modularity
  - Cohesiveness
  - Separation of Concerns
  - Abstraction/Information Hiding
  - Loose Coupling
- Living Documentation

Quality Assurance Principles

Quality staff advocate for quality!

They ensure quality gets addressed during the following:

- **Design** via *validation* (Are we building the right thing?)
  - Understanding the scope of the feature/enhancement
  - Identifying testing requirements across the Testing Pyramid levels
  - Collaborating on acceptance criteria
- **Implementation** via *verification* (Are we building the thing right?)
  - Were the acceptance criteria met?
  - Were the acceptance criteria protected by automated testing across the various levels of the Testing Pyramid?
- **Deployment** via *certification:*
  - Does everything in production still work as expected after *the deployment*?
  - Does this right thing perform at scale?
- **Release** via *attestation:*
  - Does everything in production still work as expected after the release?
  - Have we activated all of the functionality of the newly released feature?)

Quality staff advocate for our customers!

They bring focus on the customers during:

- **Design** by:
  - Surfacing testability requirements.
  - Including estimates for quality analysis to ensure the work will deliver high value from the Quality Pyramid. It will be:
    - Performant
    - Well crafted
    - An experience that is pleasing for the customer.
- **Implementation** by validating incremental deployments align with the published interfaces used by our customers.
- **Deployment** by:
  - Creating deployment testing plans.
  - Ensuring the impacted services meet defined standards and functionality (SLAs).
- **Release** by:
  - Creating release testing plans.
  - Ensuring releases meet the defined standards and functionality.
  - Tracking issues and their timely resolution.

Quality Assurance Practices

- **Issue Tracking and Resolution:**
  - Issue priorities are assigned based on the following:
    - Scope of impact on customers:
      - All
      - Portions of the customer base
      - None
    - Severity of impact on the system(s)
- **Performance Monitoring:**
  - Dashboards are created/modified to monitor for:
    - Latency
    - Time To Interactive (TTI)
    - Error rates
  - Monitoring and alerting on these is critical! They can all have an impact on SLAs.
  - Dashboards are reviewed during sprint planning to ensure issues are being worked on in their defined timeframes.
- **Monitoring and Reporting on Quality:**
  - Software Composition Analysis (in collaboration with Security Engineering):
    - Open source dependency vulnerabilities
    - License compliance
    - Static code analysis
  - Engineering Analysis:
    - Testing coverage analysis
    - Deployment analysis (in collaboration with Cloud Engineering)
    - Successful deployment rate
    - Deployment pipeline time analysis
- **Deployment Testing:**
  - Readiness Testing:
    - Management (Starts/Stops/Restarts gracefully)
    - Health checks: the service and its dependencies
    - Logging and Auditing: Is the service recording implementation and domain-level events?
    - Error Handling: Are retries and fallbacks behaving as expected?
  - Canary Deployments:
    - Slow rollout of new server instances
    - Monitors health of new servers, halting the deployment if there are issues.
  - System Performance Monitoring: Ensure new monitoring requirements are in place.
- **Release Testing:**
  - Feature Activation Testing
  - New feature(s) is(are) made available
  - Limitations on accessibility (phased rollouts)
  - System Performance Monitoring:
    - Observe new monitoring activity
    - Temporarily increase specific alerting frequency so issues are caught quickly.

The Quality Pyramids

Quality Pyramid

(Based on Hee-Meng Foo's *You Can't Fix Quality Just By Catching Bugs* article)

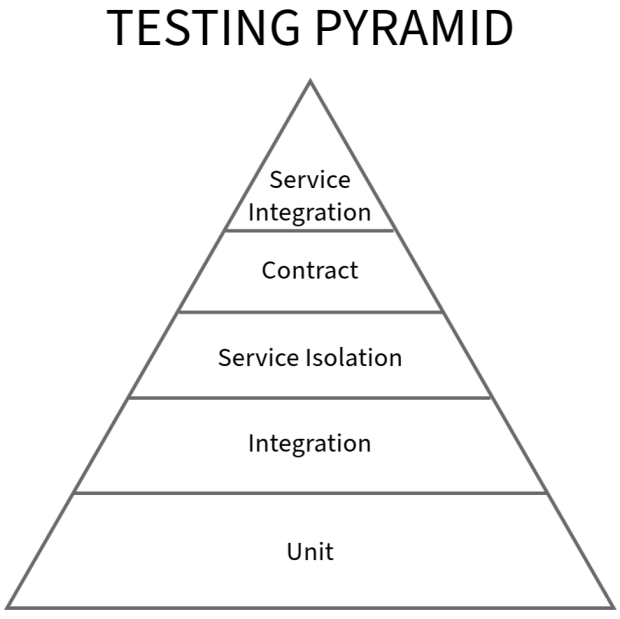
- **Basic Hygiene** (does the app/service crash or become unusable frequently?):
  - Infrastructure
  - Error handling
- **Value Provision** (does the functionality work as designed?):
  - Major functionality (Mission critical)
  - Minor functionality (Better-to-have than not)
- **Customer Performant** (is the customer's time wasted unnecessarily?):
  - "Are you really sure?" delete modals
  - Requiring customers to click-to-clear on success messages
- **Engineering Craftsmanship** (can teams deploy multiple times per day with a near-perfect success ratio?):
  - At this level, the culture of "delivering the feature at all costs" needs to change to "delivering quality software in a timely manner without burning down part of the house."
  - What is the cost of deploying one modified line of code?
  - Manual testing should NOT be required.
- **Customer Pleasure** (do customers derive pleasure from using the product?)

(from Hee-Meng-Foo's *You Can't Fix Quality Just By Catching Bugs* article)

Testing Pyramid for Microservices

(From Tariq King's *Testing Microservices* training)

- **Unit:**
  - Class as the Unit (Isolation)
  - Focused on business logic
  - Mock internal and external dependencies
- **Integration:**
  - Mock external dependencies only
- **Service Isolation:**
  - Microservice as the Unit
  - Mock are limited to external services
- **Contract:**
  - Public interface testing
  - Can provide insights into how the public interface is being used
- **Service Integration:**
  - Test Ordering Integration
  - Pairwise Integration
  - Neighborhood Integration
  - Production Mirror Integration



(From Tariq King's *Testing Microservices* training)

Glossary

Behavioral Driven Development (BDD): Defining the specifications for a product via scenarios. These scenarios communicate the desired outcomes that should result from an action or event performed under specific conditions. These specifications provide living documentation of the product.

Customers: Entities interacting with the software. These can be:

- Customers (B2C)
- Customers of our customers
- Users (non-paying customers)
- Internal engineering teams and their software
- External API Partners (B2B)

Deployment vs. Release: These two concepts must be considered separate and distinct. In a CI/CD paradigm, the code changes are being deployed to the production environment daily, possibly multiple times per day. Dozens of these deployments will likely occur before the feature is ready for customers. Once feature-ready, the business can choose when and how to release it, making it available to the customer base.

Neighborhood Integration: A "Neighborhood" is a set of services one edge away from the given service. These edge services may be either immediate predecessor or immediate successor services of the given service. In this scenario, I am only concerned with my service's neighborhood. Each of my neighbors is, in turn, responsible for their own immediate neighbors. NOTE: The difficulty of fault isolation increases as the neighborhood size increases.

Pairwise Integration: Since we cannot exhaustively test every combination of system inputs, we limit testing to each edge-pair of services in the system. This results in one integration test for each edge in a call graph.

Production Mirror Integration: A duplicate of the production environment is used for integration testing. In this scenario, there is no need to create mocks or stubs of anything.

Release vs. Deployment: See Deployment vs. Release

Test Driven Development (TDD): A software development practice that places a high value on the testability of code. Just as a regime of good personal hygiene, diet, and exercise improves a person's health, TDD improves the code base's health. Tests validating the code's functionality are written before writing the code, thus driving the development. This process is done incrementally — the tests and the code mature together until the requested functionality is fully developed. After development, these tests continue to protect the health of this functionality by validating that the results remain unchanged when new code is added anywhere in the codebase.

Test Ordering Integration: Seeks to find the "optimal" order for integration testing by reducing the cost of creating stubs (both in number and overall complexity). Typically, the ordering starts with services with no dependencies and works out from there.