Architectural Code Analysis

Using it in building Microservices

NYC Cloud Expo 2017 (June 6-8)
Agenda

- Intro to Structural Analysis
- Challenges addressed during traditional software development
- The new world of Cloud and Microservices – new set of challenges
- Designing and building new applications
- Making use of Microservices
- Refactoring monolith applications
Quick Intro

- Contact:
  - Email: P.Rabinovich@castsoftware.com
  - Twitter: itmetricsguy
  - LinkedIn: linkedin.com/in/prabinovich

Background:

- 18 years of building and delivering enterprise level solutions for Fortune 1000 companies and federal government agencies
- Various roles in software development, IT, delivery and solution design organizations
- For the past 7 years have been helping customers to incorporate Architectural Analysis into their development process
Why is Structural Analysis is Important

- **Architecturally Complex Violations** are structural flaws involving interactions among components that reside in different application layers. Although they constitute only 8% of the vulnerabilities in an application, they represent:
  - 48% of the repair effort and require 20 times more changes to fix!
  - 8 times more likely to escape into testing and 6 times more likely to escape into operations.

**Effective Software Risk Prevention:**

- Focus on critical violations that matter
- Focus resources on area of highest impact not highest volume

“Tracking programming practices at the Unit Level alone may not translate into the anticipated business impact, ...most devastating defects can only be detected at the System Level.” - OMG
Recent SEI Study on Impact of Decisions

What Kinds of Decisions Matter?

Survey results from over 1,800 developers
Look Horizontally Across Application Tiers

Look horizontally across layers to see how they are interacting
Tracing Vertically Though Application Call Stack

Review vertical slice of the application from user action (or API call) through the call stack all the way down to the backend components.
Example: System-Level Defects visible through Transactions

**Good Quality?**
- **Ascx Source File**
  - WCSFinancialData.ascx
- **public C# Class**
  - wcsfinancialdata_ascx_CastGenerated_741880ac
- **override public C# Method**
  - InitMe
- **private C# Method**
  - PopulateFinancialGrid
- **public C# Method**
  - GetFinancialData
- **proc_SearchWCSFinancial**
  - UsrStgModScrn
- **StoModScrnID** int
- **StoPartCovID** int
- **StoContractID** int

**Poor Quality**
- **Ascx Source File**
  - WCSFinancialData.ascx
- **public C# Class**
  - wcsfinancialdata_ascx_CastGenerated_741880ac
- **override public C# Method**
  - InitMe
- **private C# Method**
  - PopulateFinancialGrid
- **public C# Method**
  - GetFinancialData
- **proc_SearchWCSFinancial**
  - UsrStgModScrn
- **StoModScrnID** int
- **StoPartCovID** int
- **StoContractID** int

**Website UI**
- **Remote Calls in Loop**
- **Business Logic**
- **Avoid using SQL queries inside a loop**
- **SELECT * FROM customers WHERE username = "John Doe"**
- **SQL query on an XXL tables without indexes**

**Data Access & DB**
- **Transaction**
What does the new world look like?

MICROSERVICES DEFINED
What exactly are Microservices?

No concrete definition exists…

But… have consensus on principles
- They are small
- Narrowly focused
- Independently deployable
The new world of Microservices

UI
Business Logic
Data Access

Database

UI
Microservice

Microservice

Microservice

Data
Data
Data
Microservices – The Promise

- Faster time to market
- Independent release cycles
- Use any technology to develop
- Isolate failures
Microservices – why do it?

It is all about scaling!

- Scaling of the development organization
- Scaling of applications that to meet increasing demand
- Increase velocity of getting features into production
Foundation Required to Support Microservices

- Mature CI CD process
- Good understanding of the functional domain
- Organization is ready to embrace and restructure
- Finally, this is right design pattern – not a given!!
Not a Silver Bullet

- A lots of things to keep in mind
- In effect this is a "distributed" app design
- A LOT harder to build!!!
- As such, keep in mind all “Fallacies of Distributed Computing” when utilizing Microservices architecture
  - The **network** is reliable.
  - **Latency** is zero.
  - **Bandwidth** is infinite.
  - The network is **secure**.
  - **Topology** doesn't change.
  - There is one **administrator**.
  - Transport cost is zero.
  - The network is homogeneous.
The entire notion of Microservices is built on is minimizing dependencies.
Building new Microservices
GREENFIELD PROJECTS
City Planner vs. Architect

- City Planner is a better comparison to the role of a Software Architect vs. Civil Engineer
- A system, just like a city, needs to be a happy place to cohabitate developers, operations people, and of course end users
- “Zoning” Mentality – need to worry much less about what happens inside the zone than what happens between the zones. worry about what happens between the zones vs. inside the zones.”

Sam Newman
Building Microservices: Designing Fine-Grained Systems
Consistency through Architectural Standards

- Use of Standards in a distributed system is even more important than in Monolith
- A lot more moving parts so need standardization in order for those systems to come together
- Architectural standards vs. coding standards
Domain Driven Design

- Core concept of Microservices Architecture
- Break down the system into **Bounded Contexts**

**Image from MartinFowler.com**

**Diagram:**

- Sales Context:
  - Opportunity
  - Pipeline
  - Territory
  - Sales Person
- Support Context:
  - Customer
  - Ticket
  - Defect
  - Product Version

**Image from MartinFowler.com**
Approach for Defining Domains

- **Challenge – not so easy**
  - Need to have very good understanding of the problem space
  - Getting boundaries wrong can be very costly
  - Doing a lot of upfront design is not practical

- **Caution – do not rush into it**
  - Sam Newman – prematurely decomposing systems into Microservices can be costly, especially if new to domain
  - Martin Flower suggests that Monolith-First approach might be better way to go

- **Approach – build and refactor**
  - Architectural Analysis can provide a number of indicators to determine when to refactor and for ensuring the boundaries have been drawn up correctly
Architectural Analysis Metrics

- Cohesion is a degree to which elements inside a module belong together determined by strength of their relationships.
- Coupling is the degree of interdependence between modules.
Principle of Maintaining a Single Purpose

- Use **Cohesion Metrics** to determine when service starts providing too much unrelated functionality
- Perform continuous analysis and when gets above certain threshold put on the refactor backlog
Principle of Avoiding Chatty Services

- **Coupling Metrics** will provide indication on the level of dependencies between metric being developed and external metrics it consumes.
- Iterative calls to an external service is a potential indicator of excessive network traffic.
Cyclomatic Complexity Metric

- Cyclomatic Complexity is determined by the logical paths through the component’s source code.
Avoid CRUD-type Services that are too Small

- Use dataflow to identify the path traveled by the data elements from where they are stored to where they are shared out by the service.

Evaluate the amount of “logic” (cyclomatic complexity) that is performed on the data to ensure internal implementation details are hidden and the service provide functionality beyond simple CRUD operations.
Principle of Hiding Implementation Details

- Large portion of the data elements that is implemented by the service must be internal to the interworking of a service
- Use Architectural analysis to evaluate the ratio of total data elements to the number of elements exposed by the services interfaces
Principles of Observability and Traceability

- Observability is a level of transparency provided on what is happening within the service
- Log when transaction starts, critical points and when it exists
- Include Correlation ID to allow tie relevant events together
- Compare cyclomatic complexity of a transaction to the level of logging that is done within its boundaries
Principle of Isolation of Failure

- Use structural analysis to analyze the flow around calls to external services to ensure it is consistent and provides service resiliency.

- Calls to external services should not expect to be consistently dependable.

- Ensure calls to external services are properly decorated by setting appropriate timeout thresholds, recognition of failed conditions, retries, etc.

- Synchronous vs. Asynchronous
Principle of Being Conservative with Ingress Data

- Postel’s Law – Be conservative in what you send and liberal in what you accept
- Only extract data from payload that is being used by the service
- Use of Architectural Analysis to identify data not being used
Favor Orchestration vs. Choreography

- Avoid “god” services that are responsible for coordinating activities
- Avoid dumb CRUD services that are controlled by orchestration services
- Total number of outgoing calls and the “Integration Complexity” metrics are indicators of a “god” service
Use Pre-Built Tools

- Zipkin – distributed tracing system
- Linker-d and Envoy – proxies for inter-services communication
- Handle a lot of hard stuff that have to deal with when building distributed systems:
  - Discovery
  - Health checking
  - Load balancing
  - Observability
  - Failure handling
- Provides a way for services implemented in different technologies to communicate with one another
- Use Architectural Analysis to ensure these systems are used correctly, consistently and uniformly
Refactoring Monolith Systems to Microservices

MODERNIZATION PROJECTS
Principles

- Do it gradually – no big bang approach
- Start with least impactful parts of the application
- Use of “Stranger Pattern” to replace functionality
- It is all about Bounded Context and Seams

All these principles can be driven using **Architectural Analysis** through the **metrics** and **insight** that it provides about **interworkings** of a monolith application.
Identify Bounded Context and Seams

- **Cohesion** and **coupling** metrics can naturally help point out which components are part of the same domain and belong together and where to make the separation from the system.
Use of Transactions to “Peel” Vertical Layers

- Review which components support delivering of functionality for specific bounded context by reviewing transactions
- Identify components that create dependencies that will need to be decoupled
Use of Architectural Dependency Mappings

- Review dependencies that will need to be taken care of while decoupling components
- The level of interconnections will determine the effort and risks associated with refactoring efforts
Strangled Pattern in Action

- Martin Fowler introduced the “Strangle Pattern” as a way to refactor older applications.
- Architectural Analysis provides insight to help manage that process.

![Diagram of Strangled Pattern in Action]
Wrap-Up

- In the new world of Microservices it is all about **architecture**
- Has a lot of potential, but adds much **complexity**
- Don’t forget old-school OO concepts as they are still applicable in the new world
- Determine **If** use of Microservices architecture is **appropriate**
- Use Architectural Analysis for:
  - Refactoring monolith to identify bounded context and gradually “strangle” out functionality
  - Building new Microservices based application to drive consistency and adherence to best practices across an organization
  - Ensuring you are getting most out of prepackaged proxy or tracing system being used by the application
Thank you

- Contact:
  - Email: P.Rabinovich@castsoftware.com
  - Twitter: itmetricsguy
  - LinkedIn: linkedin.com/in/prabinovich