#OSCON

**Building Evolutionary Architectures**

Support Constant Change

Mike Mason
@mikemasonca

Zhamak Dehghani
@zhamakd
OSCON Networking Opportunity
What is Software Architecture?
requirements

Time
accessibility
accountability
accuracy
adaptability
administrability
affordability
agility
auditability
autonomy
availability
compatibility
composability
configurability
correctness
credibility
customizability
debuggability
degradability
determinability
demonstrability
dependability
deployability
discoverability
distributability
durability
effectiveness
efficiency
extensibility
failure transparency
fault-tolerance
fidelity
flexibility
inspectability
installability
integrality
interchangeability
interoperability
learnability
maintainability
manageability
mobility
modifiability
modularity
operability
orthogonality
portability
precision
predictability
process capabilities
productivity
provability
recoverability
relevance
repeatability
reproducibility
resilience
responsiveness
reusability
robustness
safety
scalability
seamlessness
self-sustainability
serviceability
supportability
securability
simplicity
stability
standards compliance
survivability
sustainability	
tailorability
testability
timeliness
traceability
transparency
ubiquity
understandability
upgradability
usability

<table>
<thead>
<tr>
<th>Accessibility</th>
<th>Reliability</th>
<th>Reproducibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accountability</td>
<td>Extensibility</td>
<td>Resilience</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Failure transparency</td>
<td>Responsiveness</td>
</tr>
<tr>
<td>Adaptability</td>
<td>Fault-tolerance</td>
<td>Reusability</td>
</tr>
<tr>
<td>Administrability</td>
<td>Fidelity</td>
<td>Robustness</td>
</tr>
<tr>
<td>Affordability</td>
<td>Flexibility</td>
<td>Safety</td>
</tr>
<tr>
<td>Agility</td>
<td>Inspectability</td>
<td>Scalability</td>
</tr>
<tr>
<td>Auditability</td>
<td>Installability</td>
<td>Seamlessness</td>
</tr>
<tr>
<td>Autonomy</td>
<td>Integrity</td>
<td>Self-sustainability</td>
</tr>
<tr>
<td>Availability</td>
<td>Interchangeability</td>
<td>Serviceability</td>
</tr>
<tr>
<td>Compatibility</td>
<td>Interoperability</td>
<td>Supportability</td>
</tr>
<tr>
<td>Composability</td>
<td>Learnability</td>
<td>Securability</td>
</tr>
<tr>
<td>Configurability</td>
<td>Maintainability</td>
<td>Simplicity</td>
</tr>
<tr>
<td>Correctness</td>
<td>Manageability</td>
<td>Stability</td>
</tr>
<tr>
<td>Credibility</td>
<td>Mobility</td>
<td>Standards compliance</td>
</tr>
<tr>
<td>Customizability</td>
<td>Modifiability</td>
<td>Survivability</td>
</tr>
<tr>
<td>Debugability</td>
<td>Modularity</td>
<td>Sustainability</td>
</tr>
<tr>
<td>Degradability</td>
<td>Operability</td>
<td>Tailorability</td>
</tr>
<tr>
<td>Determinability</td>
<td>Orthogonality</td>
<td>Testability</td>
</tr>
<tr>
<td>Demonstrability</td>
<td>Portability</td>
<td>Timeliness</td>
</tr>
<tr>
<td>Dependability</td>
<td>Precision</td>
<td>Traceability</td>
</tr>
<tr>
<td>Deployability</td>
<td>Predictability</td>
<td>Transparency</td>
</tr>
<tr>
<td>Discoverability</td>
<td>Process capabilities</td>
<td>Ubiquity</td>
</tr>
<tr>
<td>Distributability</td>
<td>Producibility</td>
<td>Understandability</td>
</tr>
<tr>
<td>Durability</td>
<td>Provability</td>
<td>Upgradability</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>Recoverability</td>
<td>Usability</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Relevance</td>
<td></td>
</tr>
</tbody>
</table>

**Evolvability**
Change
Change

ecosystem

requirements
Business Change
Change

ecosystem

requirements
Dynamic Equilibrium
How is long term planning possible when things constantly change in unexpected ways?
Once I’ve built an architecture, how can I prevent it from gradually degrading over time?
In each group, choose a system that one of your members has worked on. Explain the architecture to the rest of the group — we’ll use this to anchor today’s workshop exercises.

Be ready to share at the end!
Building Evolutionary Architectures

#OSCON

Defining Evolutionary Architecture

Neal Ford, Rebecca Parsons & Patrick Kua
Evolutionary Architecture

An evolutionary architecture supports guided, incremental change across multiple dimensions.
Evolutionary Architecture

An evolutionary architecture supports guided incremental change across multiple dimensions.
*guided*

**evolutionary computing fitness function:** a particular type of objective function that is used to summarize...how close a given design solution is to achieving the set aims.
An architectural fitness function provides an objective integrity assessment of some architectural characteristic(s).
Evolutionary Architecture

An evolutionary architecture supports guided, incremental change across multiple dimensions.
incremental
Evolutionary Architecture

An evolutionary architecture supports guided, incremental change across multiple dimensions.
multiple dimensions

auditability

performance

security

requirements

Time

data

legality

scalability
Evolutionary Architecture

An evolutionary architecture supports guided, incremental change across multiple dimensions.
Why evolutionary?
Why evolutionary?

adaptable?
Building Evolutionary Architectures

Penultima↑e

#OSCON

Neal Ford, Rebecca Parsons & Patrick Kua
EA Spreadsheet

✔ definition

! verification

Penultima → e
EA Spreadsheet

✔ definition
✔ verification

Penultima e
Building Evolutionary Architectures

Fitness Functions: Categories and Examples

Neal Ford, Rebecca Parsons & Patrick Kua

#OSCON
Categories of Fitness Functions

atomic

run against a singular context and exercise one particular aspect of the architecture.

holistic

run against a shared context and exercise a combination of architectural aspects such as security and scalability.
Categories of Fitness Functions

run based on a particular event, such as a developer executing a unit test, a deployment pipeline running unit tests, or a QA person performing exploratory testing.

don’t run on a schedule, but instead execute constant verification of architectural aspect(s) such as transaction speed.
Categories of Fitness Functions

**Static**

have a fixed result, such as the binary pass/fail of a unit test.

**Dynamic**

rely on a shifting definition based on extra context. Some values may be contingent on circumstances, and most architects will accept lower performance metrics when operating at high scale.
Categories of Fitness Functions

tests and other verification mechanism that run without human interaction.

must involve at least one human.
Categories of Fitness Functions

architects may want to build a time component into assessing fitness

temporal

break on upgrade

overdue library update
Categories of Fitness Functions

Some architectures have specific concerns, such as special security or regulatory requirements.
Categories of Fitness Functions

architects will define most fitness functions at project inception as they elucidate the characteristics of the architecture...

...some fitness functions will emerge during development of the system
Fitness Function

- Triggered
- Atomic
- Holistic
- Continuous
Cyclic Dependency Function

```java
/**
 * Tests that a package dependency cycle does not exist for any of the analyzed packages.
 */
public void testAllPackages() {
    Collection packages = jdepend.analyze();
    assertEquals("Cycles exist",
                false, jdepend.containsCycles());
}
```
Consumer Driven Contracts

martinfowler.com/articles/consumerDrivenContracts.html
Fitness Function

- atomic
- holistic
- triggered
- continuous
Fitness Function

atomic

triggered

holistic

continuous
Holistic fitness functions must run in a specific (shared) context.
Fitness Function

- Atomic
- Holistic
- Triggered
- Continuous
atomic monitoring

continuous logging
Use synthetic transactions to test production systems.
Fitness Function

- Atomic
- Holistic
- Triggered
- Continuous
Fitness Function

atomic

triggered

continuous

holistic
System-wide Fitness Function
Building Evolutionary Architectures

Fitness Functions: Testing and Automation

#OSCON
Testable

- persistence
- web

packages/namespaces
Testable

packages/namespaces
public void testMatch() {
    DependencyConstraint constraint = new DependencyConstraint();

    JavaPackage persistence = constraint.addPackage("com.xyz.persistence");
    JavaPackage web = constraint.addPackage("com.xyz.web");
    JavaPackage util = constraint.addPackage("com.xyz.util");

    persistence.dependsUpon(util);
    web.dependsUpon(util);

    jdepend.analyze();

    assertEquals("Dependency mismatch",
                true, jdepend.dependencyMatch(constraint));
}
Implementing architectural fitness functions using Gradle, JUnit and code-assert

Architectural fitness functions

Inspired by Neal Ford's presentation at our Change is the Only constant event I started experimenting with architectural fitness functions. An architectural fitness function provides an objective integrity assessment of some architectural characteristic(s).

If you want to take a deeper dive into evolutionary architectures including fitness functions take look at Neal's book: Building Evolutionary Architectures: Support Constant Change.

Neal's slides contained an example of verifying package dependencies from a Unit Test using JDepend.

Verifying code modularity

In this blog post we'll elaborate on that approach and create a Unit Test that verifies that our code conforms to the chosen coding strategy using an
Package by layer (horizontal slicing)

Package by feature (vertical slicing)
public class VerifyPackageByLayerTest {

    @Test
    public void verifyPackageByLayer() {

        // Create an analyzer config for the package we'd like to verify
        AnalyzerConfig analyzerConfig = GradleAnalyzerConfig.gradle().main("com.jdriven.fitness.packaging.by.layer");

        // Dependency rules for Packaging by Layer
        // NOTE: the classname should match the packagename
        class ComJdrivenFitnessPackagingByLayer extends DependencyRuler {

            // Rules for layer child packages
            // NOTE: they should match the name of the sub packages
            DependencyRule controller, service, repository;

            @Override
            public void defineRules() {
                // Our App classes depends on all subpackages because it constructs all of them
                base().mayUse(base().allSub());
                // Controllers may use Services
                controller.mayUse(service);
                // Services may use Repositories
                service.mayUse(repository);
            }
        }

        // All dependencies are forbidden, except the ones defined in ComJdrivenFitnessPackagingByLayer
        // java, org, net packages may be used freely
        DependencyRules rules = DependencyRules.denyAll()
            .withRelativeRules(new ComJdrivenFitnessPackagingByLayer())
            .withExternals("java.*", "org.*", "net.*");

        DependencyResult result = new DependencyAnalyzer(analyzerConfig).rules(rules).analyze();
        assertThat(result, matchesRulesExactly());
    }
}
public class ControllerA {
    private final ServiceA serviceA;
    private final RepositoryA repositoryA;

    public ControllerA(ServiceA serviceA, RepositoryA repositoryA) {
        this.serviceA = serviceA;
        this.repositoryA = repositoryA;
    }
}

java.lang.AssertionError:
Expected: Comply with rules
but: DENIED com.jdriven.fitness.packaging.by.layer.controller -&gt;
com.jdriven.fitness.packaging.by.layer.repository (by com.jdriven.fitness.packaging.by.layer.controller.ControllerA)
ArchUnit
https://www.archunit.org/

Unit test your Java architecture

Start enforcing your architecture within 30 minutes using the test setup you already have.

Start Now

ArchUnit is a free, simple and extensible library for checking the architecture of your Java code using any plain Java unit test framework. That is, ArchUnit can check dependencies between packages and classes, layers and slices, check for cyclic dependencies and more. It does so by analyzing given Java bytecode, importing all classes into a Java code structure. You can find examples for the current release at ArchUnit Examples and the sources on GitHub.
import static com.tngtech.archunit.lang.syntax.ArchRuleDefinition.noClasses;
import static com.tngtech.archunit.library.GeneralCodingRules.ACCESS_STANDARD_STREAMS;
import static com.tngtech.archunit.library.GeneralCodingRules.NO_CLASSES_SHOULD_ACCESS_STANDARD_STREAMS;
import static com.tngtech.archunit.library.GeneralCodingRules.NO_CLASSES_SHOULD_THROW_GENERIC_EXCEPTIONS;
import static com.tngtech.archunit.library.GeneralCodingRules.NO_CLASSES_SHOULD_USE_JAVA_UTIL_LOGGING;

public class CodingRulesTest {
    private JavaClasses classes;

    @Before
    public void setUp() throws Exception {
        classes = new ClassFileImporter().importPackagesOf(ClassViolatingCodingRules.class);
    }

    @Test
    public void classes_should_not_access_standard_streams_defined_by_hand() {
        noClasses().should(ACCESS_STANDARD_STREAMS).check(classes);
    }

    @Test
    public void classes_should_not_access_standard_streams_from_library() {
        NO_CLASSES_SHOULD_ACCESS_STANDARD_STREAMS.check(classes);
    }

    @Test
    public void classes_should_not_throw_generic_exceptions() {
        NO_CLASSES_SHOULD_THROW_GENERIC_EXCEPTIONS.check(classes);
    }

    @Test
    public void classes_should_not_use_java_util_logging() {
        NO_CLASSES_SHOULD_USE_JAVA_UTIL_LOGGING.check(classes);
    }
}
public class InterfaceRules {

    @Test
    public void interfaces_should_not_have_names_ending_with_the_word_interface() {
        JavaClasses classes = new ClassFileImporter().importClasses(
            SomeBusinessInterface.class,
            SomeDao.class
        );

        noClasses().that().areInterfaces().should().haveNameMatching(".*Interface").check(classes);
    }

    @Test
    public void interfaces_should_not_have_simple_class_names_ending_with_the_word_interface() {
        JavaClasses classes = new ClassFileImporter().importClasses(
            SomeBusinessInterface.class,
            SomeDao.class
        );

        noClasses().that().areInterfaces().should().haveSimpleNameContaining("Interface").check(classes);
    }

    @Test
    public void interfaces_must_not_be_placed_in_implementation_packages() {
        JavaClasses classes = new ClassFileImporter().importPackagesOf(SomeInterfacePlacedInTheWrongPackage.class);

        noClasses().that().resideInAPackage("..impl..`).should().beInterfaces().check(classes);
    }
}
public class LayerDependencyRulesTest {
    private JavaClasses classes;

    @Before
    public void setUp() throws Exception {
        classes = new ClassFileImporter().importPackagesOf(ClassViolatingCodingRules.class);
    }

    @Test
    public void services_should_not_access_controllers() {
        noClasses().that().resideInAPackage("..service..")
            .should().accessClassesThat().resideInAPackage("..controller..").check(classes);
    }

    @Test
    public void persistence_should_not_access_services() {
        noClasses().that().resideInAPackage("..persistence..")
            .should().accessClassesThat().resideInAPackage("..service..").check(classes);
    }

    @Test
    public void services_should_only_be_accessed_by_controllers_or_other_services() {
        classes().that().resideInAPackage("..service..")
            .should().onlyBeAccessed().byAnyPackage("..controller..", "..service..").check(classes);
    }
}
@Test
public void third_party_class_should_only_be_instantiated_via_workaround() {
    classes().should(notCreateProblematicClassesOutsideOfWorkaroundFactory()
        .as(THIRD_PARTY_CLASS_RULE_TEXT))
        .check(classes);
}

private ArchCondition<JavaClass> notCreateProblematicClassesOutsideOfWorkaroundFactory() {
    DescribedPredicate<JavaCall<?>> constructorCall0fThirdPartyClass =
        target(is(constructor())).and(targetOwner(is(assignableTo(ThirdPartyClassWithProblem.class))));

    DescribedPredicate<JavaCall<?>> notFromWithinThirdPartyClass =
        originOwner(is(not(assignableTo(ThirdPartyClassWithProblem.class)))).forSubType();

    DescribedPredicate<JavaCall<?>> notFromWorkaroundFactory =
        originOwner(is(not(equivalentTo(ThirdPartyClassWorkaroundFactory.class)))).forSubType();

    DescribedPredicate<JavaCall<?>> targetIsIllegalConstructorOfThirdPartyClass =
        constructorCallOfThirdPartyClass.
            and(notFromWithinThirdPartyClass).
            and(notFromWorkaroundFactory);

    return never(codeUnitWhere(targetIsIllegalConstructorOfThirdPartyClass));
}
Fitness Function Katas

http://evolutionaryarchitecture.com/ffkatas/
Building Evolutionary Architectures

AFTERNOON TEA

Mike Mason  
@mikemasonca

Zhamak Dehghani  
@zhamakd

Neal Ford, Rebecca Parsons & Patrick Kua
Building Evolutionary Architectures

HYPOTHESIS AND DATA-DRIVEN DEVELOPMENT

Neal Ford, Rebecca Parsons & Patrick Kua
Experiments to Perform

- More Listings
- Better Structure
- Better Prioritization
vision, strategy, business goals

ideation

selected experiments:

portfolio of experiments

pivot
fold
double down
Building Evolutionary Architectures

MOVE FAST AND FIX THINGS

O'REILLY

Building Evolutionary Architectures

SUPPORT CONSTANT CHANGE

Neal Ford, Rebecca Parsons & Patrick Kua

GitHub Engineering

Move Fast and Fix Things

Anyone who has worked on a large enough codebase knows that technical debt is an inescapable reality. The more rapidly an application grows in size and complexity, the more technical debt is accrued. With GitHub's growth over the last 7 years, we have found plenty of nooks and crannies in our codebase that are inevitably below our very best engineering standards. But we've also found effective and efficient ways of paying down that technical debt, even in the most active parts of our systems.

At GitHub we try not to brag about the "shortcuts" we've taken over the years to scale our web application to more than 12 million users. In fact, we do quite the opposite: we make a conscious effort to study our codebase looking for systems that can be rewritten to be clearer, simpler and more efficient, and we develop tools and workflows that allow us to perform these rewrites efficiently and reliably.

As an example, two weeks ago we replaced one of the most critical code paths in our infrastructure: the code that performs merges when you press the Merge button in a Pull Request. This code had been an evergreen cause of regressions and outages over the years, and we finally got around to finding a better way.

At the end of the day, technical debt is always a choice. The choices we make to pay it down are what truly defines the quality of our engineering work.
def create_merge_commit(base, head, author, commit_message)
    base =resolve_commit(base)
    head = resolve_commit(head)
    commit_message = Rugged.pretty_message(commit_message)
    merge_base = rugged.merge_base(base, head)
    return [nil, "already merged"] if merge_base == head
    ancestor_tree = merge_base & Rugged.Commit.lookup(rugged, merge_base).tree
    merge_options = {
        fail_on_conflict => true,
        oscket => true,
        use_rebase => true
        }
    index = base.tree.merge(head.tree, ancestor_tree, merge_options)
    return [nil, "merge conflicts"] if (index.nil? || index.conflicts)
    options = {
        message => commit_message,
        comitter => author,
        author => author,
        base => [base, head],
        }.
    tree = Index.write_tree(rugged)
    (Rugged::Commit.create(rugged, options, nil)

def create_merge_commit(author, base, head, options = {})
    commit_message = options[commit_message] || "Merge #{head} into #{base}"
    now = Time.current
    science "create_merge_commit" do
        e.context :issue => base.to_s, :ref => head.to_s, :remote => repository.remote
        e.use { create_merge_commit_git(author, now, base, head, commit_message) }
        e.try { create_merge_commit_rugged(author, now, base, head, commit_message) }
    end
end

https://github.com/github/scientist
- It decides whether or not to run the try block,
- Randomizes the order in which use and try blocks are run,
- Measures the durations of all behaviors,
- Compares the result of try to the result of use,
- Swallows (but records) any exceptions raised in the try block
- Publishes all this information.
Accuracy

The number of times that the candidate and the control agree or disagree. View mismatches
The number of incorrect/ignored only.
Building Evolutionary Architectures

ARCHITECTURAL CHARACTERISTICS

#OSCON
Architecture Characteristics

- reliability
- scalability
- performance
- availability
Architecture Characteristics

accessibility  evolvability  reproducibility
accountability extensibility resilience
accuracy fault transparency responsiveness
adaptability fidelity reusability
administrability flexibility robustness
affordability installability safety
agility integrality scalability
auditability interchangeability self-sustainability
autonomy interoperability serviceability
availability learnability supportability
compatibility maintainability secularity
composability manageability simplicity
configurability mobility stability
correctness modifiability standards compliance
credibility modularity survivability
customizability modularity sustainability
debugability modularity tailorability
degradability modularity testability
determinability operability timeliness
demonstrability orthogonality traceability
dependability portability understandability
deployability precision upgradability
discoverability predictability usability
distributability process capabilities

durability producibility

effectiveness provability

efficiency recoverability

erasibility relevance

Architecture Characteristics

For each of the following business challenges, decide on the appropriate “ilities” that would be necessary in the system you learned about in the icebreaker. For each characteristic, how would you measure it in the example system?
Architecture Characteristics

“our business is constantly changing to meet new demands of the marketplace”

extensibility, maintainability, agility, modularity
Architecture Characteristics

“due to new regulatory requirements, it is imperative that we complete end-of-day processing in time”

performance, scalability, availability, reliability
Architectural Characteristics

“we need faster time to market to remain competitive”

maintainability, agility, modularity, deployability, testability
Architecture Characteristics

“our plan is to engage heavily in mergers and acquisitions in the next three years”

scalability, extensibility, openness, standards-based, agility, modularity
Architecture Characteristics

“we have a very tight timeframe and budget for this project”

feasibility
architecture patterns help define the basic characteristics and behavior of the application
Domain Driven Design
Bounded Context
Architectural Quantum

![Diagram of architectural quantum components]
Microservices Quantum
An architectural quantum is an independently deployable component with high functional cohesion, which includes all the structural elements required for the system to function properly.
Architectural Quantum

An architectural quantum is an independently deployable component with high functional cohesion, which includes all the structural elements required for the system to function properly.
For Each Pattern:

- incremental change
- guided change via fitness functions
- appropriate coupling
- architectural quantum
Big Ball of Mud
Big Ball of Mud

Rippling side effects for any change

difficult because no clearly defined partitioning exists

Epitome of inappropriate coupling

the entire system
Monoliths
Unstructured Monoliths

Diagram showing a user interface and various classes interconnected.
Unstructured Monoliths

the entire system

Large quantum size hinders incremental change because high coupling requires deploying large chunks of the application.

difficult but not impossible

functionally almost as bad as the Big Ball of Mud
Layered Monolith

presentation

business rules

database

components
Layered Monolith

presentation
business rules
component
component
component

database
component
component
component

Selectively easy based on partitioning

easier because structure is more apparent

— good technical architecture partitioning
Modular Monoliths
Modular Monoliths

the entire system, with selective better granularity

the degree of deployability of the components determines the rate of incremental change.

easier to design and implement in this architecture because of good separation of components

Each component is functionally cohesive, with good interfaces between them and low coupling.
Microservices

client requests

client requests

client requests

API layer

checkout module

inventory module

listing module

accounts module

bulk txn module

service module

routing module

database

database

database

database

database

database

database
Microservices

extremely fine grained

deployment pipeline considered standard

21st century DevOps practices

robust testing & fitness function culture

extremely decoupled fine-grained quanta with well defined boundaries
“Serverless” Architecture

FaaS

Function as a Service

API Gateway

Amazon Lambda

DynamoDB
"Serverless" Architecture

- N/A
- critical
- deployment pipeline integration challenging
- redeploy code
#OSCON

Building Evolutionary Architectures

BUILDING EVOLVABLE ARCHITECTURES: MECHANICS

Neal Ford, Rebecca Parsons & Patrick Kua
1. Identify dimensions affect by evolution
1. Identify dimensions affected by evolution:

- Auditability
- Performance
- Security
- Requirements
- Data
- Legality
- Scalability
1. Identify dimensions affected by evolution

2. Define Fitness Function(s) for Each Dimension
2. Define Fitness Function(s) for Each Dimension
1. Identify dimensions affect by evolution

2. Define Fitness Function(s) for Each Dimension

3. Use Deployment Pipelines to Automate Fitness Functions
3. Use Deployment Pipelines to Automate Fitness Functions

commit/unit test → functional test → atomic fitness functions → UAT → holistic fitness functions

01001001010101
01010101010101
00101010010010
00100100010001
01010010100110
01000100010011
00101000100011

unit tested code
functionally tested code
architecturally tested code
deployed quantum
integration environment
1. Identify dimensions affect by evolution

2. Define Fitness Function(s) for Each Dimension

3. Use Deployment Pipelines to Automate Fitness Functions
1. Identify dimensions affected by evolution
2. Define Fitness Function(s) for Each Dimension
3. Use Deployment Pipelines to Automate Fitness Functions
#OSCON

Building Evolutionary Architectures

Building Evolvable Architectures: Greenfield & Brownfield Architectures

Neal Ford, Rebecca Parsons & Patrick Kua
Greenfield Projects
Greenfield Projects

– implement incremental change at project inception
– fitness functions definition easier before implementation
– architects don’t have to untangle legacy coupling points
– protective fitness functions from project outset
– choose architecture patterns that support evolution
Brownfield Projects
Appropriate Coupling & Cohesion
Understand the business problem before choosing an architecture.
Improve Engineering Practices

- **Release It! Second Edition**
  - Design and Deploy Production-Ready Software

- **Continuous Delivery**
  - Release Software Faster through Build, Test, and Deployment Automation
  - By Jez Humble and David Farley
  - Forwarded by Martin Fowler

- **Infrastructure as Code**
  - Managing Services in the Cloud
  - By Kief Morris
What About COTS?*

– quantum size: the package

– incremental change: generally scores poorly

– appropriate coupling: generally scores poorly

– fitness functions: generally scores poorly

*Commercial Off-the-shelf Software
Work diligently to hold integration points to your level of maturity...
...if that isn’t possible, realize that some parts of the system will be easier for developers to evolve than others.
#OSCON

Building Evolutionary Architectures

Putting Evolutionary Architecture into Practice
Organizational Factors
Incidentally Coupled Teams

user interface

database

server-side

DBA
Autonomous Teams

Orders

Inverse Conway Maneuver

Shipping

Catalog
Autonomous Teams...

...Organized around Business Capability

Orders

Shipping

Catalog
Product over Project
Autonomous Teams...

Orders

Catalog

Shipping
Product over Project
Product over Project

– Projects are ephemeral
– Projects isolate developers from operational aspects
– Products live forever
– Products have owners
– Products consist of cross-functional teams
Product over Project

Long-term company buy-in
Team Coupling Characteristics
Culture

– Does everyone on the team know what fitness functions are and consider the impact of new tool or product choices on the ability to evolve new fitness functions?

– Are teams measuring how well their system meets their defined fitness functions?

– Do engineers understand cohesion and coupling?

– Are there conversations about what domain and technical concepts belong together?

– Do teams choose solutions not based on what technology they want to learn, but based on its ability to make changes?

– How are teams responding to business changes?
Culture of Experimentation

- Bring ideas from outside
- Encouraging explicit improvement
- Spike and stabilize
- Creating innovation time
- Following set-based development
- Connecting engineers with end-users
Building Enterprise Fitness Functions

commit/unit test → functional test → atomic fitness functions → UAT → holistic fitness functions

0100100101010101
0101010101010101
0010101001001010
0010010001000100
0010010001000100

unit tested code → functionally tested code → architecturally tested code

deployed quantum

integration environment
commit/unit test -> functional test -> atomic fitness functions -> UAT -> holistic fitness functions

- unit tested code
- functionally tested code
- architecturally tested code
- deployed quantum
- integration environment
Building Evolutionary Architectures

For more information:
http://evolutionaryarchitecture.com