The
SOFTWARE
ESSENTIALIST

The Ideal Developer Workflow — How to Kick Off Feature Work Consistently & Effectively

essentialist.dev

Different ways to kick off a project & different ways to approach feature-development.

Based on The Metaphysics & The 12 Essentials, we now know a few things about this…

- *The Feedback Loop: It's more likely we'll introduce bugs and miss things than get everything right, so we should drive our work with tests.*
- *Value Identification: Nothing we're doing matters if it doesn't solve an actual User problem, so we need to clarify what that is — and build it for the Customer.*
- *Abstraction: The customer decides what we build (Acceptance Criteria), so we use Acceptance Tests and drive our work Outside-In*
- *The Walking Skeleton: It's better to first deploy a Vertical Slice of functionality to identify uncertainty*

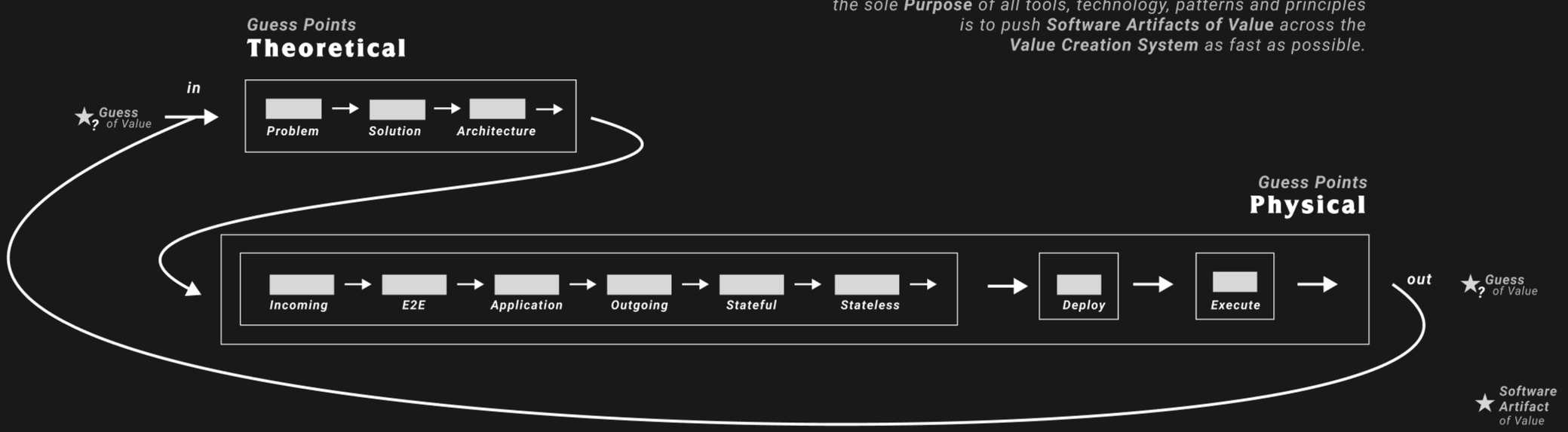
Based on The Metaphysics & The 12 Essentials, we now know a few things about this

- *Vertical Slicing: We operate on Vertical Slices; Features (customer-driven) or individual Operations (developer-driven).*
- *Horizontal Decoupling: Frontend, backend, desktop — every Vertical Slice has the same same stereotypical layers and concerns; these are mini systems.*
- *Subject Verification: We can place boundaries around the systems of our app & write tests for them.*
- *Deployment & Delivery: We need to embed tests into our Deployment Pipeline so that we can prevent Negative Value and achieve high quality code*

Putting that all together, we can conceptualize an Ideal Developer Workflow which works on any side of the stack

The Value Creation System

*As value-creating software developers, the sole **Purpose** of all tools, technology, patterns and principles is to push **Software Artifacts of Value** across the **Value Creation System** as fast as possible.*



THE SOFTWARE ESSENTIALIST

The 12 Essentials

of Software Design, Architecture, Testing & Strategy

Metaphysical Essential

Abstraction

Mediocre developers think about Code first.

Software Essentialists, excellent developers think about Value first.

The purpose of software is to solve problems — to achieve goals.

There is a Theoretical and a Physical aspect to all problems.

A high-level and a low-level.

Software Essentialists bring the Theoretical and the Physical together using the art of Abstraction.

First, you clarify the Theoretical — the Who, the Why, the What.

Then you move to the Physical — the How, the Code.

Software Essentialists write less code, have the end goal in mind, and create the most impact.

Metaphysical Essential

The Feedback Loop

Every single aspect of software development is a Guess.

Do our API calls work?

Are we missing any edge cases?

Are we saving to the database properly?

Will this architecture serve us in the long-term?

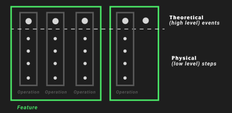
Does this feature do what the Customer wants?

Will users actually use this?

It's all a Guess.

The only way to progress is 3 Ways: Feedforward, Feedback, and Refinement.

Declare the end result, take a leap forward, use the feedback to refine.



Physical Essential

Vertical Slicing

If you can't "see your features" from the folder structure alone, there's a good chance maintainability is under pressure.

Developers primarily work against Features.

Discovering, understanding, adding, changing, removing, testing, debugging features.

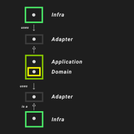
Because of this, you want to chunk your work, your modules, your features, your operations - into Vertical Slices of functionality.

There are two types of Slices: Features & Operations.

Features are high-level slices. These are valuable for the Customer. You build & Acceptance Test User Stories, Scenarios & Concrete Examples against these.

Operations are low-level slices. These are your commands & queries. Your API calls. They are what come together in sequence to make a feature.

Focus on thinking, planning, organizing, structuring, testing & building vertical slices into your testing & architecture.



Physical Essential

Horizontal Decoupling

When we don't separate concerns, code can turn to mush.

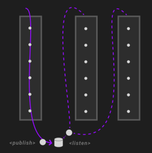
Not only is your codebase hard to test, but it can feel like you're maintaining a ton of small scripts.

Without a clear place for abstractions to go, you introduce duplication & complexity over time.

Horizontal Decoupling EoS involves reorganizing concerns into well-known layers of Abstraction (infra, adapter, app, domain).

It also involves learning to use dependency inversion to decouple, invert & isolate the layers you want to test or swap out.

With a clear picture of where abstractions belong, how they interact, you gain testing options, less duplication and more maintainable code, flexible code.



Physical Essential

Temporal Decoupling

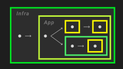
Requirements are temporal.

There's always an element of "time" involved.

They always seem to come in the form of "after this", "before that", or "only when this happens".

Instead of trying to slot new operations inside of, before, or after existing ones - coupling them together, we use Temporal Decoupling: decouple using events.

By publishing & subscribing to events, we can chain features & operations, chaining functionality in a scalable way over time.



Physical Essential

Composition

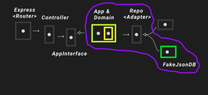
Software is made from a lot of different parts, but many developers wire these parts up without much thought.

It matters. A lot.

Without conscious thought, you can create a coupled codebase where abstractions are difficult to use, understand, test, and maintain.

Composition is about the way you consciously create a network of collaborating abstractions.

When done well, your application is configurable from a single location and it is easy to set up dynamic runtime behavior for different environments: dev, prod, testing, ci, and so on.



Physical Essential

Subject / System Verification

Your application is comprised of many different input-output systems.

Subject/system verification gives us the ability to isolate the systems (subjects) we want to test, then using one of 3 techniques.

Result, State & Communication Verification.

Result Verification verifies the input-output of a system.

State Verification verifies the internal state of a system.

Communication Verification verifies the communications that a system has with other systems.

Using these 3 forms of verification, we can perform all sorts of testing strategies using some or none of stubs, mocks, spies and the like.

Physical Essential

Deployment & Delivery

We really don't want to ship negative value to our users.

And if we're working with others, we don't want to tolerate low quality code either.

Deployment & delivery is about creating a deployment pipeline through which our software must pass before it lands in the hands of users.

This is done by introducing quality phases which run our tests.

While we can introduce any number of phases, the minimum ones we need are the Commit & Deploy ones.

The Acceptance Phase is also important, as it runs our tests in a production-like environment and gives us the most confidence that our code does what the customer asked for.

We can also add other phases over time, optimize, tune and refine the pipeline to be more stable, fast, and improve the overall quality of the software we ship.

Theoretical Essential

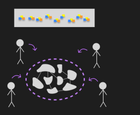
Value Identification

The worst thing to do is to start writing code before we know who it's for, why they want it, and what they should be able to do.

Value identification is the first essential we should practice at the start of a project or a feature.

Before we get into the How (the code), we also clarify Who-Why-What.

This helps us ensure what we build will have impact for both the User & the Customer.



Theoretical Essential

Domain Discovery

The language you use matters.

As abstraction designers, it's your responsibility to ensure all developers are working against the same mental models - the same conceptual understanding of the domain.

Domain Discovery involves using a number of DDD & BDD techniques to build a shared understanding of the domain, the boundaries of the problem, how we could theoretically solve it.

The benefit is a rich, stable domain model through which everything relies that we embed in the core of our codebase.

This produces way more maintainable code in the long term.

Theoretical Essential

Strategic Design

Strategic design is making smart decisions about large scale architectural components.

But there's more to architecture than just libraries, tools & frameworks

How do you know which components to use?
How do you know what to build yourself?
How do you integrate the component together?
What do you do when there isn't a component you need?

All developers need to know how to do 4 things:

1. Stereotype components
2. Develop components
3. Integrate components
4. Prioritize components

Physical Essential

The Walking Skeleton

There are a lot of snags and trouble spots that will occur when shipping anything of value.

The best time to find those snags and deal with them is at the start of a project.

The Walking Skeleton is about building the smallest slice of useful functionality and deploying it as quickly as possible.

You want to Draw a broad stroke architecture and then Build, Test & Deploy the a tiny slice of E2E functionality.

Expect lots of scripting, problems, and weirdness to occur.

Once this is done, you have a testable architecture and the foundation upon which to write quality code.

essentialist.dev

What is the Ideal Developer Workflow?

- *An ideal for how to approach your work based on all of the 12 Essentials and the teachings from the Phronimos Developers over the last 40 years*
- *Clarity; you won't always be able to do each of the steps, but some of them are non-negotiable*

What are the steps?

- *1 | Clarify The Theoretical Guesses*
- *2 | Broad Stroke Architecture*
- *3 | Clarify Testing Strategy*
- *4 | Build, Test, Deploy Walking Skeleton*
- *5 | Feature Work:*
  - *Outer-Inner Loop Tests Based on Testing Strategy*
  - *Additional Testing*
  - *Done! Next*

What are the steps?

- *1 | Clarify The Theoretical Guesses*
- *2 | Broad Stroke Architecture*
- *3 | Clarify Testing Strategy*
- *4 | Build, Test, Deploy Walking Skeleton*
- *5 | Feature Work:*
  - *Outer-Inner Loop Tests Based on Testing Strategy*
  - *Additional Testing*
  - *Done! Next*

What are the steps?

- *1 | Clarify The Theoretical Guesses*
- *2 | Broad Stroke Architecture*
- *3 | Clarify Testing Strategy*
- *4 | Build, Test, Deploy Walking Skeleton*
- *5 | Feature Work:*
  - *Outer-Inner Loop Tests Based on Testing Strategy*
  - *Additional Testing*
  - *Done! Next*

What are the steps?

- *1 | Clarify The Theoretical Guesses*
- *2 | Broad Stroke Architecture*
- *3 | Clarify Testing Strategy*
- *4 | Build, Test, Deploy Walking Skeleton*
- *5 | Feature Work:*
  - *Outer-Inner Loop Tests Based on Testing Strategy*
  - *Additional Testing*
  - *Done! Next*

What are the steps?

- *1 | Clarify The Theoretical Guesses*
- *2 | Broad Stroke Architecture*
- *3 | Clarify Testing Strategy*
- *4 | Build, Test, Deploy Walking Skeleton*
- *5 | Feature Work:*
  - *Outer-Inner Loop Tests Based on Testing Strategy*
  - *Additional Testing*
  - *Done! Next*

What are the steps?

- *1 | Clarify The Theoretical Guesses*
- *2 | Broad Stroke Architecture*
- *3 | Clarify Testing Strategy*
- *4 | Build, Test, Deploy Walking Skeleton*
- *5 | Feature Work*
  - *Outer-Inner Loop Tests Based on Testing Strategy*
  - *Additional Testing*
  - *Done! Next*

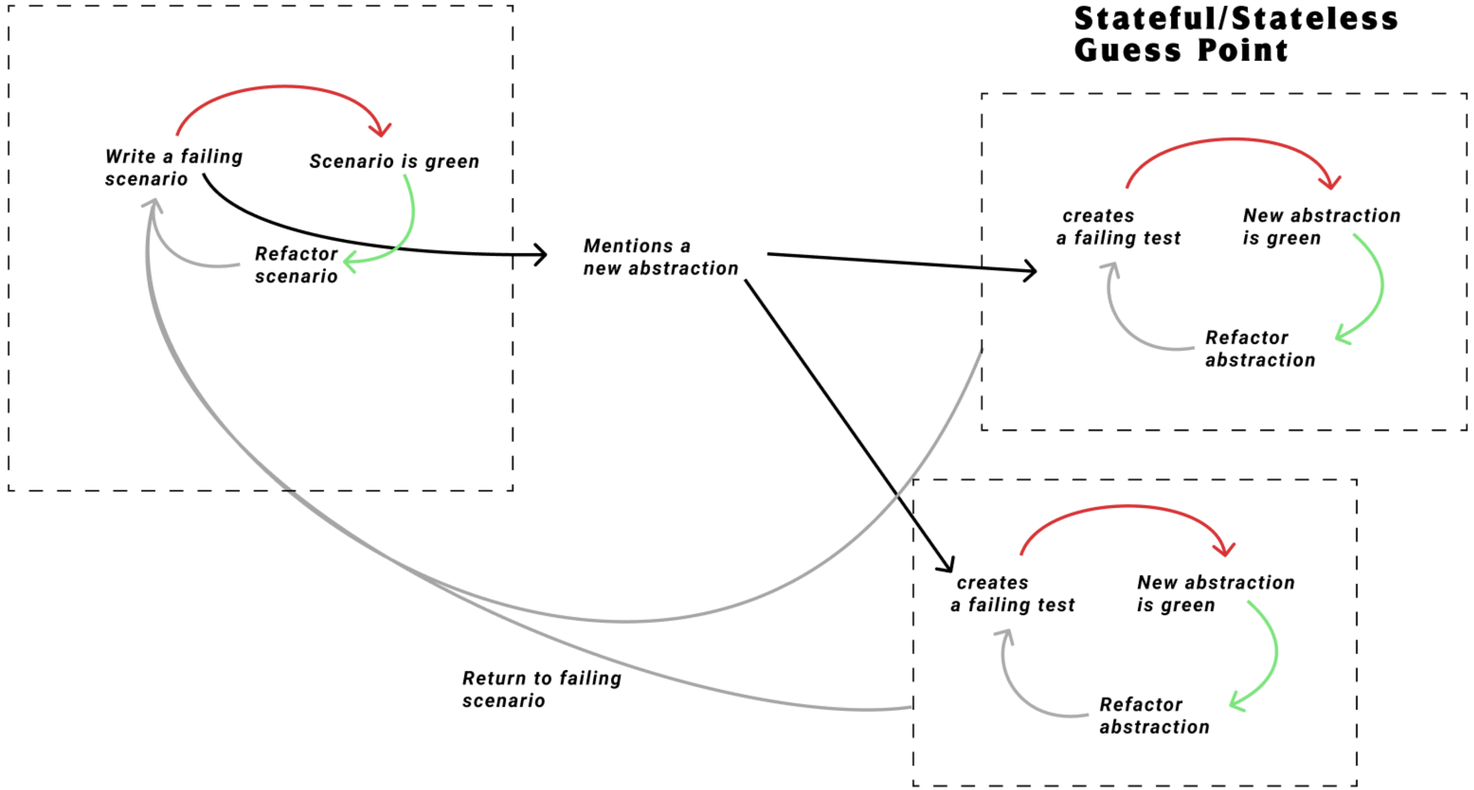
For example, if we were implement a 'CreateUser' use case, when we Feedback Loop from the Application Guess Point, we have to expect to DROP DOWN into lower level loops to create domain objects (UserEmail, UserPassword) and Adapters (EmailService, UserRepo).

Feedback Loop Level 3

Application Guess Point

Feedback Loop Level 1 & 2

Stateful/Stateless Guess Point



Feedback Loop Level 4

Outgoing Adapter Guess Point

** This is a higher Guess Point, but we use dependency inversion to flip the collaboration.*

Why is it useful?

- *Consistency*
- *Knowing how far off you are*
- *Integration of the 12 Essentials*

Where we'll learn more