Software Engineering Essentialized
Teaching material

Midterm Review

IDG

www.semat.org

The Exam

- Will be online on Moodle
- Will consist of 25 questions
- Will be open book, open notes, but not open other individuals or the internet
- You will be given 1 hour to take the exam

- The Exam will open on Wednesday at 5 pm
- The Exam will close on Friday at midnight

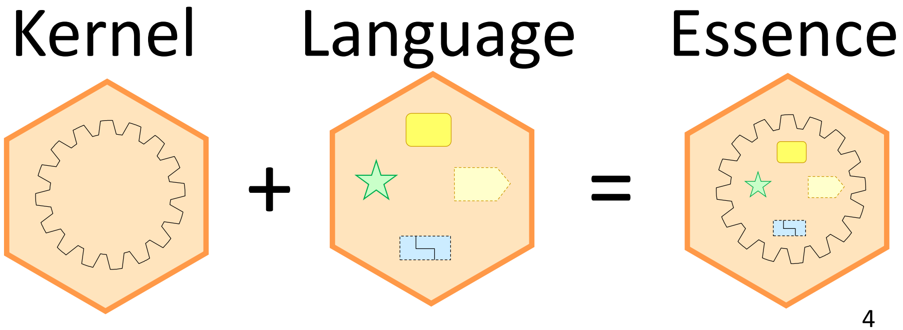
Software Engineering Essentialized
Teaching material

Topic Review

www.semat.org

Essence

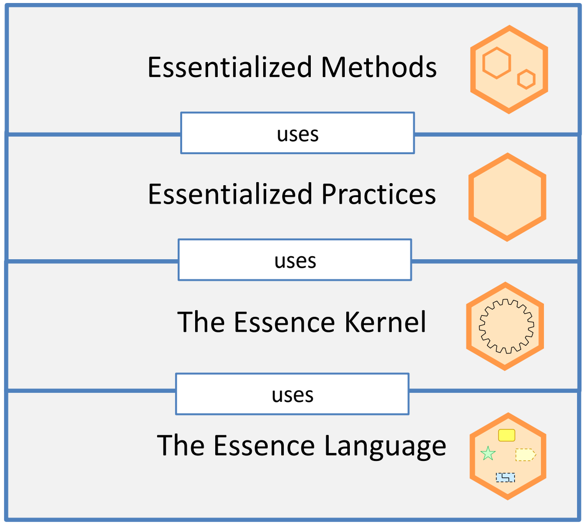
- **Essence is the kernel of a Software Engineering Theory as well as the language to describe such theory and the approach to describe methods and practices based on the theory**
- Essence is made of 2 parts:
  - Kernel
    - The set of elements that would always be found in all types of software system endeavours
  - Language
    - The Essence language is very simple, intuitive and practical
    - Utilized in describing the Essence kernel with the elements that constitute a common ground



Essence Method Architecture

- **Essentialized Methods** are composition of **Essentialized practices**

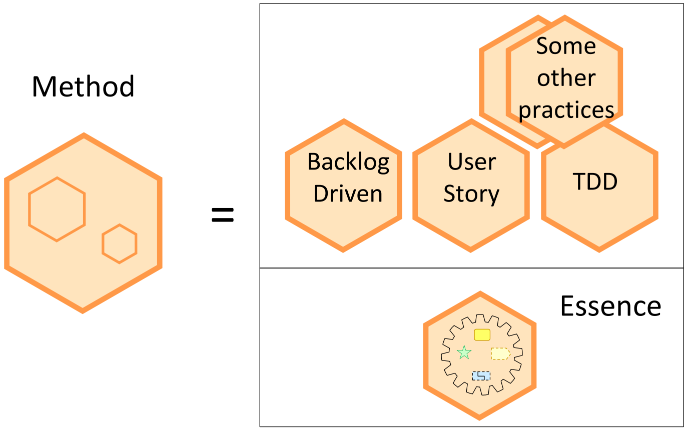
- Practices can be compositions of smaller practices.
  - Scrum for instance can be seen as a composition of three smaller practices:
    - Daily Stand-up,
    - Backlog-Driven Development
    - Retrospective.
  - Composition of practices is an operation merging two or more practices to form a method.



How to learn a Method

- Essence, beside text books, wants to provide team members with a new engaging and hands-on experience to learn the tailored method that each organization will define
  - A set of icons will help represent the elements
  - A set of cards will help describe and discuss the elements

- To build a method, a team start with the kernel and selects a number of practices and tools to make up its way-of-working



Areas of Concerns

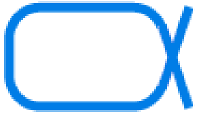
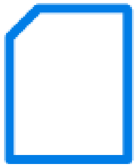
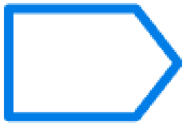
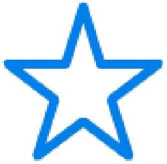
- The Essence kernel elements are organized around 3 areas of concerns, that we have already seen:

Customer – This area of concern contains everything to do with the actual use and exploitation of the software system to be produced.

Solution - This area of concern contains everything related to the specification and development of the software system.

Endeavor - This area of concern contains everything related to the development team and the way that they approach their work

Essence Language Element Types

Alpha		An essential element of the software engineering endeavor that is relevant to an assessment of the progress and health of the endeavor
Work Product		The tangible things that practitioners produce when conducting software engineering activities
Activity		Things which practitioners do
Competency		Encompasses the abilities, capabilities, attainments, knowledge, and skills necessary to do a certain kind of work.

- The Essence list is longer, but at this time we consider these elements as key and the first to learn

Alphas and Work Products

Def: ***Alphas* are subjects in a software endeavour whose evolution we want to understand, monitor, direct, and control**

- Alphas are the most important things you must attend to and progress in order to be successful in a software development endeavour
- An Alpha is not tangible by itself, but it is understood or evidenced by the work product(s) that are associated with it and thus describe a particular aspect of the alpha

Def: ***Work Products* are tangible things such as documents and reports**

- Work products may provide evidence to verify the achievement of alpha states.
- The fact that you have a document is not necessarily a sufficient condition to prove evidence of state achievement.
- Essence does not specify which work products are to be developed

Activity and Compentencies

Def: ***Activities* are things which practitioners do**

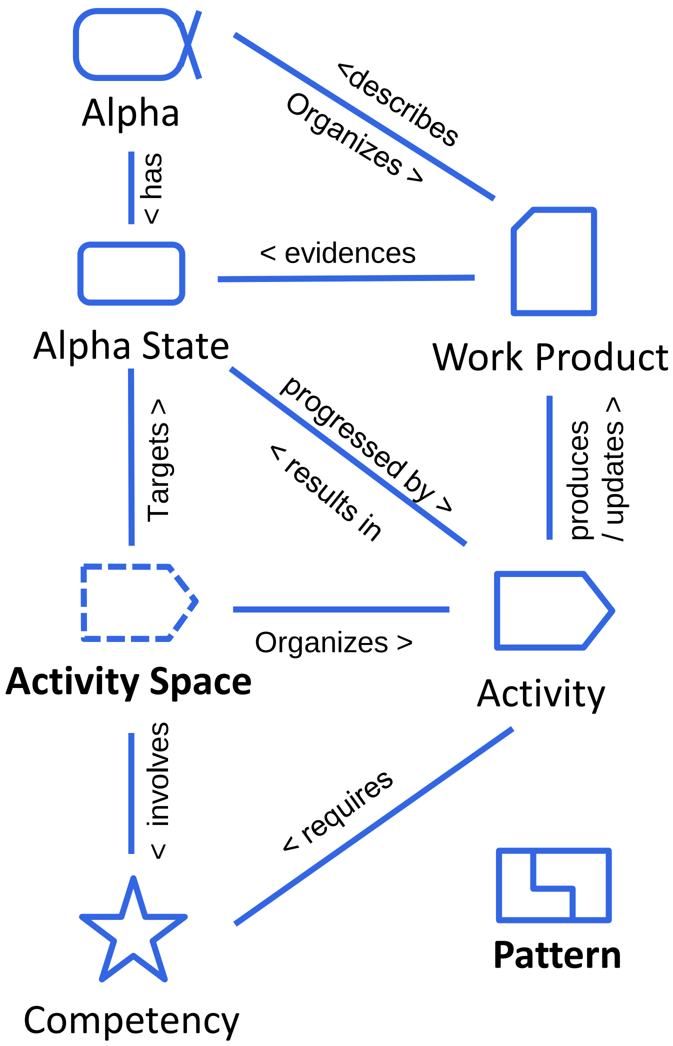
- Practitioners often struggle to determine the appropriate degree of detail or formality with an activity, or exactly how to go about conducting the activity.
- A practice may include several activities that are specific to the practice being described.

Def: ***Competencies* are the abilities needed when applying a practice**

- Often software teams struggle because they don’t have all the abilities needed for the task they have been given.

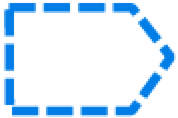
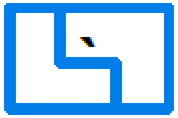
Essence Language

- These are the elements of ESSENCE LANGUAGE and their relationships
- Essentializing a Practice, means to describe a practice using the Essence language.

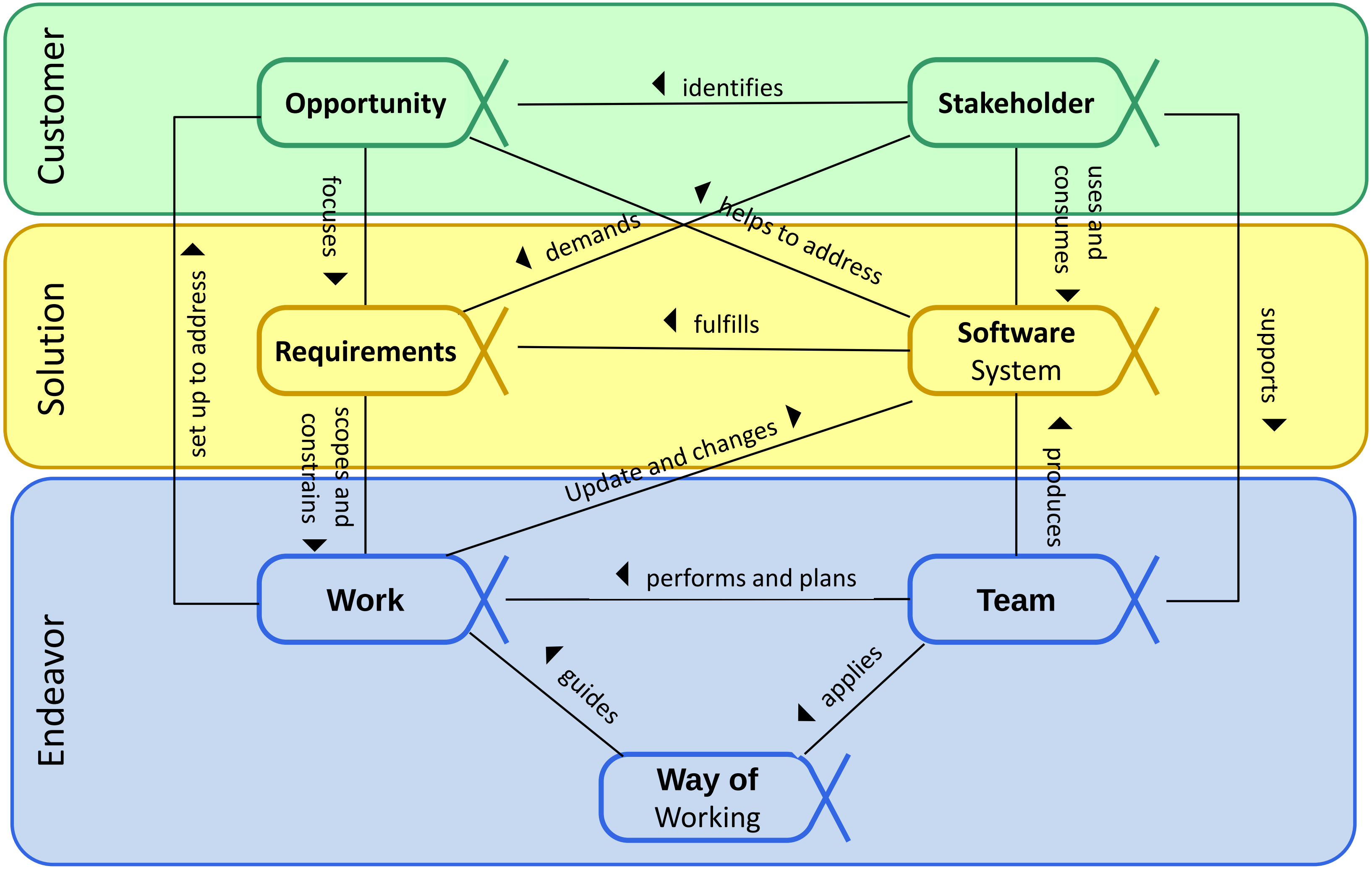


Additional Elements in Essence Language

- The Essence Language Element list contains two more elements

Element Type	Syntax	Meaning of element type
Activity Space		A placeholder for something to do in the software engineering endeavor. A placeholder may consist of zero to many activities.
Pattern		An arrangement of other elements represented in the language.

1. The alphas


Customer
Solution
Endeavor
Opportunity
Stakeholder
identifies
focuses
uses and consumes
demands
helps to address
set up to address
supports
Requirements
fulfills
Software System
scopes and constrains
Update and changes
produces
Work
performs and plans
Team
guides
applies
Way of Working

Customer

Customers: Users of our system or people that are purchasing this system for the users

Software Engineering is about providing value to customers

- Opportunity
  - An opportunity is a chance to do something to provide value to customers, including fixing an existing problem via this software system
- Stakeholders
  - Stakeholders are individuals, organizations or groups that have some interest or concern either in the system to be developed or in its development

Solution

The solution is the outcome of this endeavor

- Requirements
  - Requirements provide the stakeholder view of what they expect the software system to provide
    - They indicate what the software system must do, but do not explicitly express how it must do it
    - Among the biggest challenges software teams faces are changing requirements
- Software System
  - The primary outcome of a software endeavour is of course the software system itself.
  - 3 important characteristics of software systems
    - Functionality – Must serve some function
    - Quality – Reliability, Performance, Rich user experience, etc.
    - Extensibility – From version to version and platform to platform

Endeavors

An endeavor is any action that we take to achieve an objective

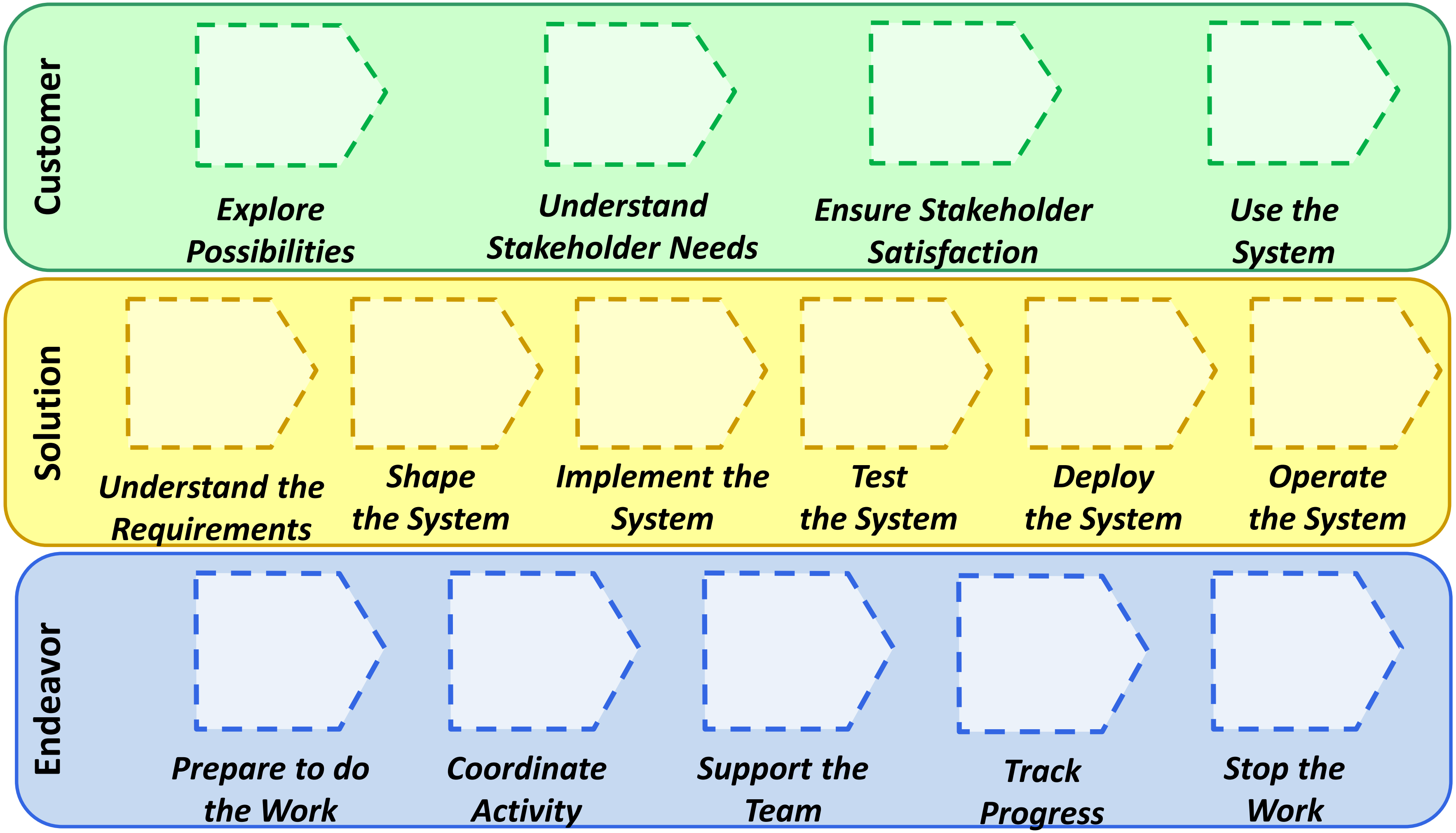
- Team
  - Team must have enough people (and not too much), with right skill mix, work collaboratively, and adapting to changing environments
  - Good team working is essential
- Work
  - The work of bringing the opportunity to reality
  - Effort and Time are the most important measures of the work
    - Effort and Time are limited
    - The idea is to get things done fast but with high quality
- Way of Working
  - Team members must agree on their way of working
    - The practice and tools that will be used
    - Used by all team members
    - Improved by the team when needed
  - One of the things we hope to achieve with Essence is simplifying the process of reaching a common agreement, that is always a major challenge

2. The Activities and Activity Spaces

- In every software development endeavour you carry out a number of activities.
  - **Essence does not define any activities**
    - how your team goes about capturing and communicating the requirements can be very different from team to team
  - **Essence defines a number of activity spaces.**
- **Def. Activity spaces are generic placeholders for specific activities**
  - Since the activity spaces are generic
    - They are method-independent
    - They could be standardized and are thus part of the Essence standard
    - Each activity space can be extended with concrete activities that progress one or more alphas
  - The activity spaces are packages used to group activities, which are related to one another
  - The activity spaces represent the essential things that have to be done to develop software

Activity Spaces in Kernel Standard

These are the Activity Space from Essence Standard



Customer

Explore Possibilities

Understand Stakeholder Needs

Ensure Stakeholder Satisfaction

Use the System

Solution

Understand the Requirements

Shape the System

Implement the System

Test the System

Deploy the System

Operate the System

Endeavor

Prepare to do the Work

Coordinate Activity

Support the Team

Track Progress

Stop the Work

Activity Spaces Essence Standard Desc.

Customer

- **Explore Possibilities**
  Explore the possibilities presented by the creation of a new or improved software system. This includes the analysis of the opportunity and the identification of the stakeholders.
- **Understand Stakeholder Needs**
  Engage with the stakeholders to understand their needs and ensure that the right results are produced. This includes identifying and working with the stakeholder representatives to progress the opportunity.
- **Ensure Stakeholder Satisfaction**
  Share the results of the development work with the stakeholders to gain their acceptance of the system produced and verify that the opportunity has been addressed.
- **Use the System**
  Observe the use the system in a live environment and how it benefits the stakeholders.

Solution

- **Understand the Requirements**
  Establish a shared understanding of what the system to be produced must do.
- **Shape the system**
  Shape the system so that it is easy to develop, change and maintain, and can cope with current and expected future demands. This includes the architecting and overall design of the system to be produced.
- **Implement the System**
  Build a system by implementing, testing and integrating one or more system elements. This includes bug fixing and unit testing.
- **Test the System**
  Verify that the system produced meets the stakeholders' requirements.
- **Deploy the System**
  Take the tested system and make it available for use outside the development team

Endeavour

- **Prepare to do the Work**
  Set up the team and its working environment. Understand and commit to completing the work.
- **Coordinate Activity**
  Co-ordinate and direct the team's work. This includes all ongoing planning and re-planning of the work, and re-shaping of the team.
- **Support the Team**
  Help the team members to help themselves, collaborate and improve their way of working.
- **Track Progress**
  Measure and assess the progress made by the team.
- **Stop the Work**
  Shut-down the software engineering endeavour and handover of the team's responsibilities.

3. The Competencies

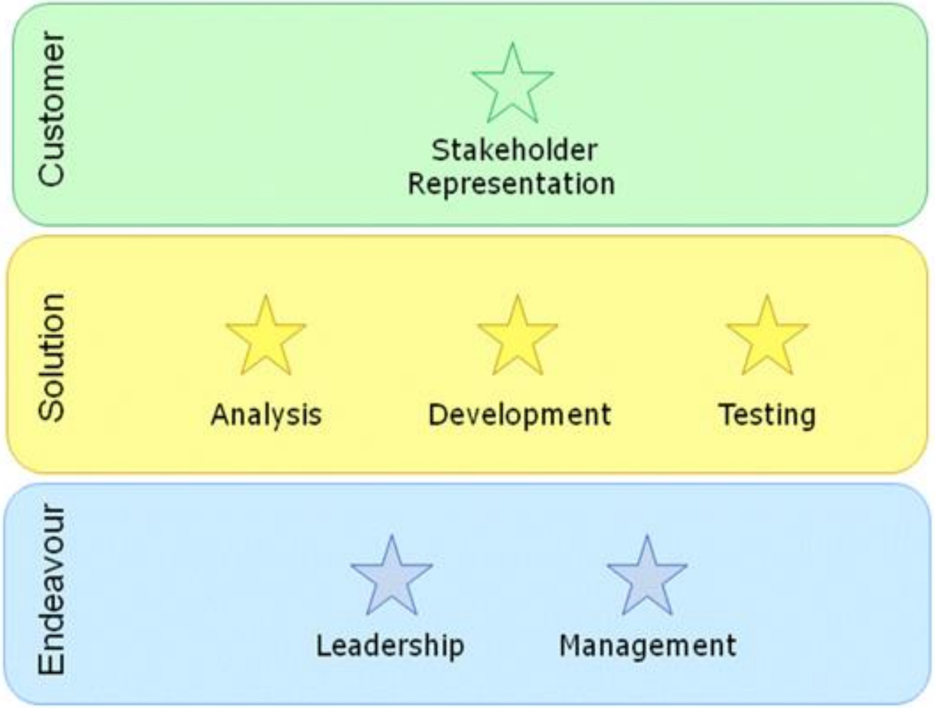
Def. *Competencies* are generic containers for specific skills

  - Specific skills, for example Java programming, are not part of the kernel because this skill is not essential on all software engineering endeavours.

- A common problem on software endeavours is not being aware of the gap between the competencies needed and the competencies available.

Competences in Essence Kernel Standard

- Competencies are aligned to the three focus areas
- Essence Kernel Standard competencies are needed for any Software Engineering Endeavour, independently then methods and techniques adopted



Competences Essence Standard Desc.

Customer

- **Stakeholder Representation**
  This competency encapsulates the ability to gather, communicate, and balance the needs of other stakeholders, and accurately represent their views.

Solution

- **Analysis**
  This competency encapsulates the ability to understand opportunities and their related stakeholder needs, and to transform them into an agreed upon and consistent set of requirements.
- **Development**
  This competency encapsulates the ability to design, program and code effective and efficient software systems following the standards and norms agreed upon by the team.
- **Testing**
  This competency encapsulates the ability to test a system, verify that it is usable and that it meets the requirements.

Endeavour

- **Leadership**
  This competency enables a person to inspire and motivate a group of people to achieve a successful conclusion to their work and to meet their objectives.
- **Management**
  This competency encapsulates the ability to coordinate, plan and track the work done by a team

Competency levels

Competency levels of achievement:

1. **Assists** – Demonstrates a basic understanding of the concepts and can follow instructions.
2. **Applies** – Able to apply the concepts in simple contexts by routinely applying the experience gained so far.
3. **Masters** – Able to apply the concepts in most contexts and has the experience to work without supervision.
4. **Adapts** – Able to apply judgment on when and how to apply the concepts to more complex contexts. Can enable others to apply the concepts.
5. **Innovates** – A recognized expert, able to extend the concepts to new contexts and inspire others.

4. Patterns

Def. *Patterns* are generic solutions to typical problems

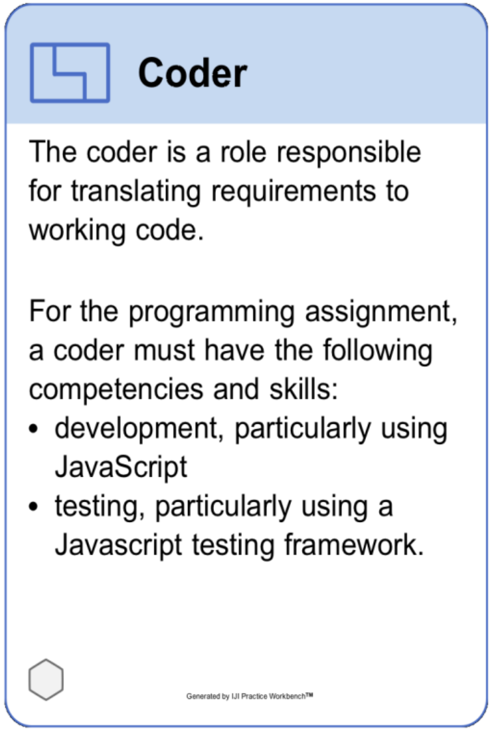
  - Patterns is the way *Essence* allows arrangements of elements to solve a specific problem

- Patterns are optional elements (not required element of a practice definition) that may be associated with any other language element.
- *Roles* are special type of *Patterns*

Roles: A Special kind of Pattern

- Roles represent the set of competencies needed to do a job effectively
  - Roles are a special kind of patter that apply to people
  - Example of Roles are Coder, Analyst, Tester

- Responsibilities to achieve a task are assigned to the task owner, that could be playing a role, but the responsibilities are not part of the role definition



Coder

The coder is a role responsible for translating requirements to working code.

For the programming assignment, a coder must have the following competencies and skills:

- development, particularly using JavaScript
- testing, particularly using a Javascript testing framework.

Generated by IJI Practice Workbench™

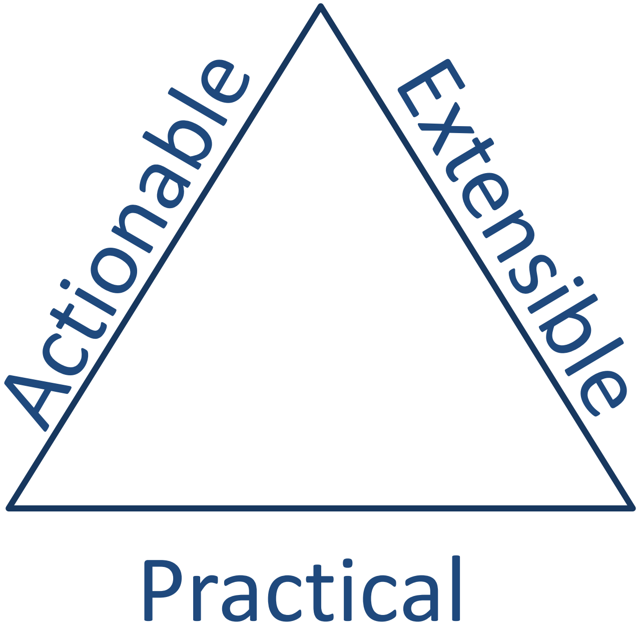
Pattern name

Brief pattern description

What makes Essence more than a conceptual framework?

Essence Guiding principles

- Alphas helps assess & drive progress and health of project
- Each state has a checklist
  - Criteria needed to reach the state
- Alphas are method and practice independent



- Practices are distinct, separate, modular units
- Kernel allow create or tailor and compose practices to new methods
- Additional Alphas can be added

- Tangible through the cards
  - Cards provide concise reminders
- Practical through Checklists and Prompts
  - Utilizable on a daily basis helping making decisions

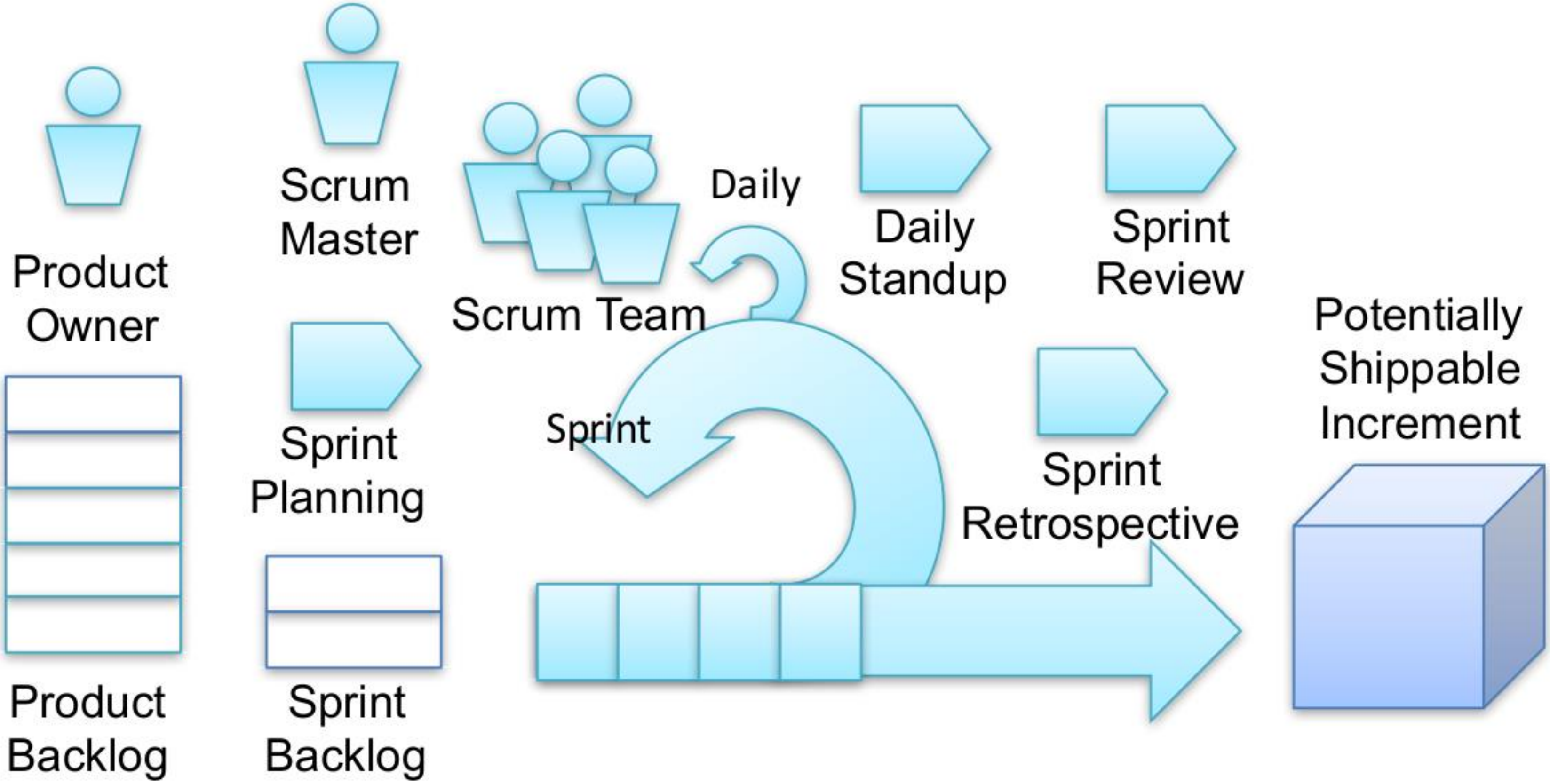
What is a Theory?

- Most theories share three characteristics
  - they attempt to generalize local observations and data into more abstract and universal knowledge
  - they generally have an interest in causality (cause and effect)
  - they often aim to explain or predict a phenomenon.
- Gregor[REF] proposes 4 goals for a theory:
  1. Describe the studied phenomenon
     - Function Point and SWEBOK could serve as an example.
  2. Explain the how, why, and when of the topic
     - theory of cognition is aimed at explaining the human mind’s limitations
  3. Beside explaining what has already happened also predict what will happen next
     - Cocomo attempts to predict the cost of software projects
  4. Prescribe how to act based on predictions
     - Alan Davis’s 201 principles exemplify this goal[REF]

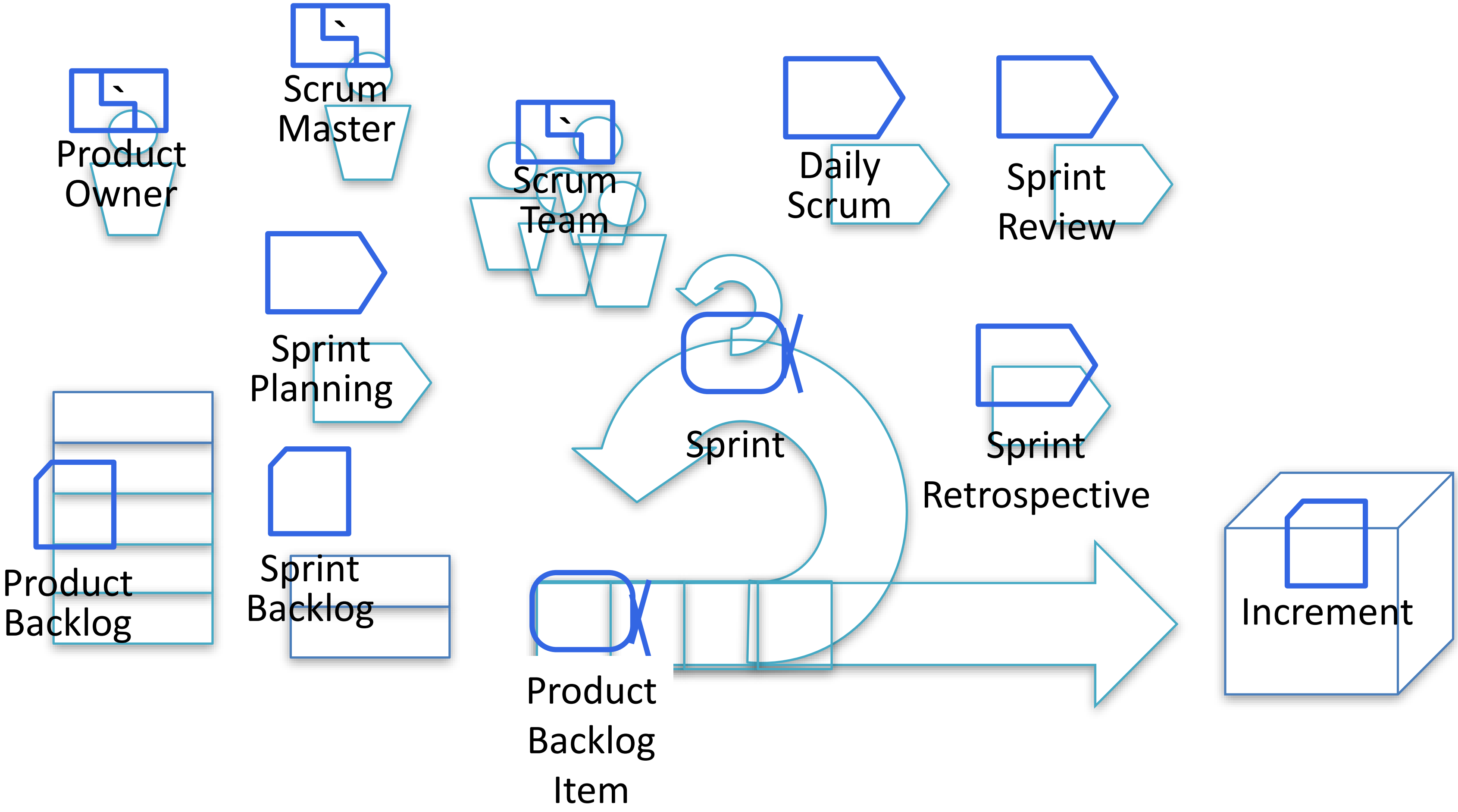
From Practices to Method

- A Method is made by selecting practices
  - Each practice is composed on top of the Kernel
  - There could be overlaps and conflicts to be resolved
- Overlaps
  - You have an overlap if two or more practices share the same kernel element for instance a work product.
  - Then you have to look at all "competing" instances of that work product and if needed replace them with one that works for all of them.
- Conflicts
  - You have a conflict if two or more elements are identical (or could be made identical) but they have been given different names.
  - Then you have to do another merge and replace these elements with one that they can all share.

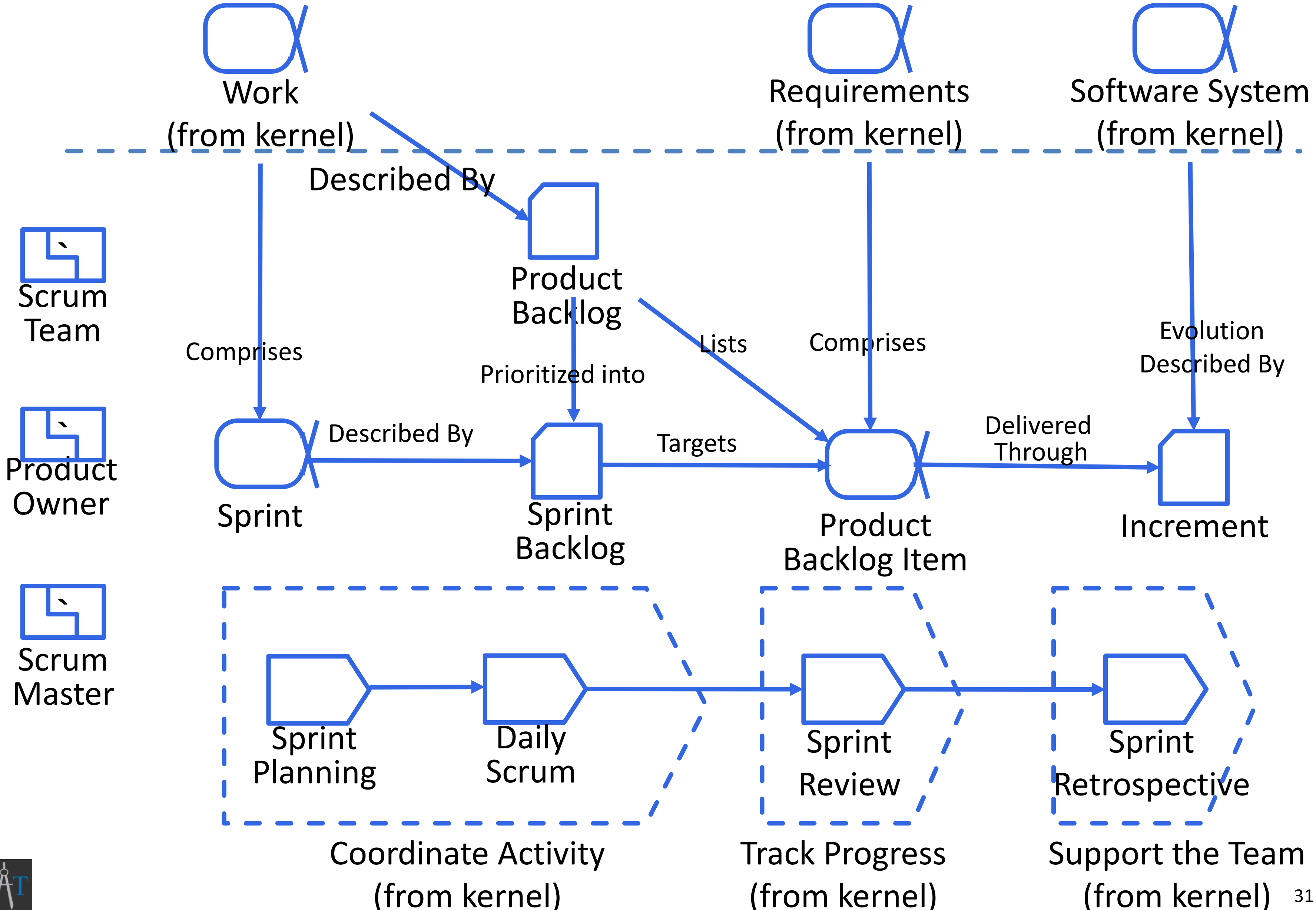
Scrum Overview



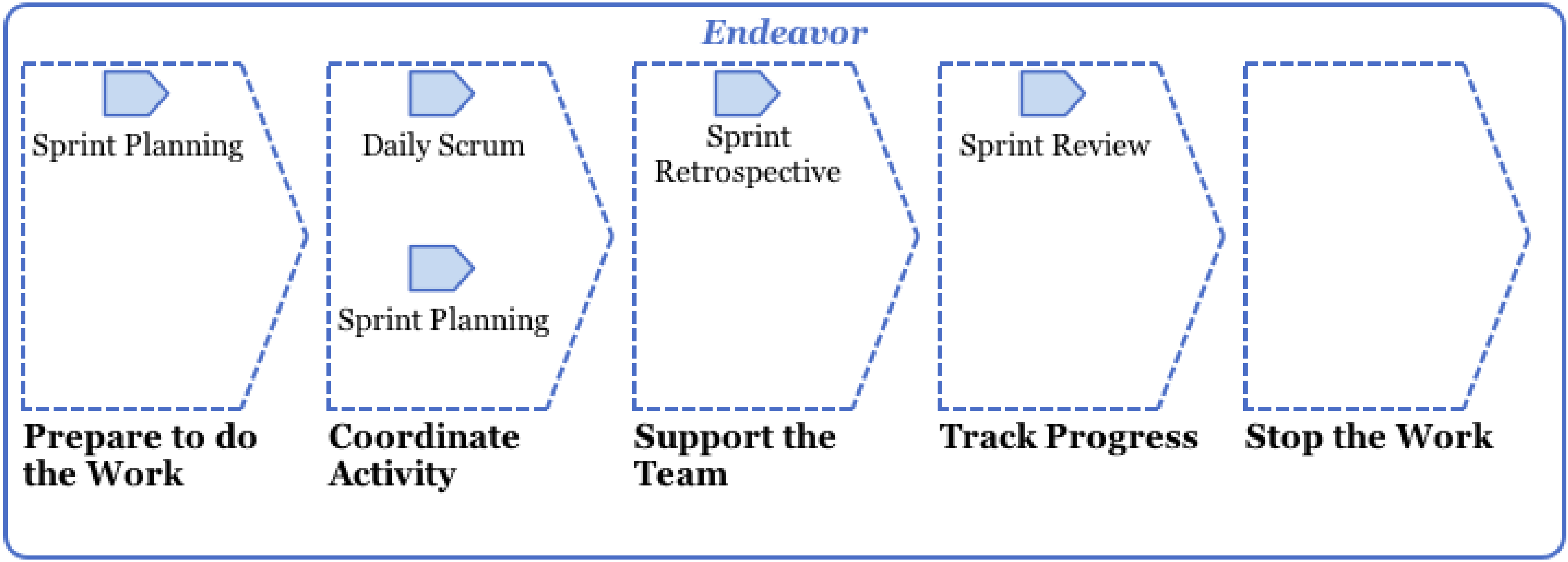
Essence elements within Scrum



Scrum Lite Practice Expressed in the Essence Language



SL Essentialized: impact



Introduction to User Story concept

- The User Story practice is a popular practice, in particular for small teams.
  - User stories have the benefit of getting the team to think, enquire and understand the value of what they do from the point of view of their users.
- A user story describes functionality in the system we are building that is valuable to a user of a system.
- A user story includes a written description that is used when discussing the story along with tests to help communicate what is needed to complete the story.
  - The idea of user stories is to provide a way to facilitate discussion to help clarify who (i.e. a role) a piece of functionality is for and how it benefits the role.

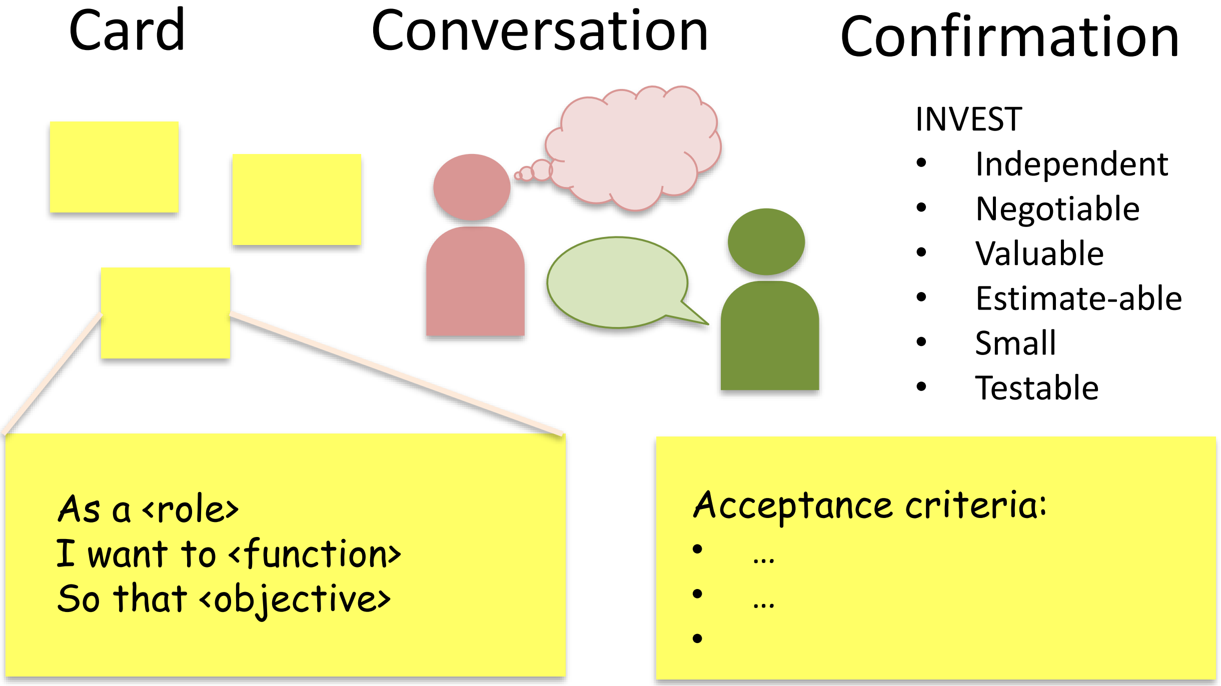
Capturing User Stories

- A user story is often captured on a 5 x 3 index card with a very concise format
  - As a <role, or type of user>,
  - I want to <list here the function you want the system to do>,
  - so that <list here the objective you want to achieve>

- An example could be:
  - As a bank customer
  - I want to have a direct deposit capability
  - so that my employer can electronically send me my paycheck.

- Who – will get the value?
- What – do we need to achieve?
- Why – are we doing it?

User Story Lite Practice Big Picture



Card

Conversation

Confirmation

INVEST

- Independent
- Negotiable
- Valuable
- Estimate-able
- Small
- Testable

As a <role>
I want to <function>
So that <objective>

Acceptance criteria:

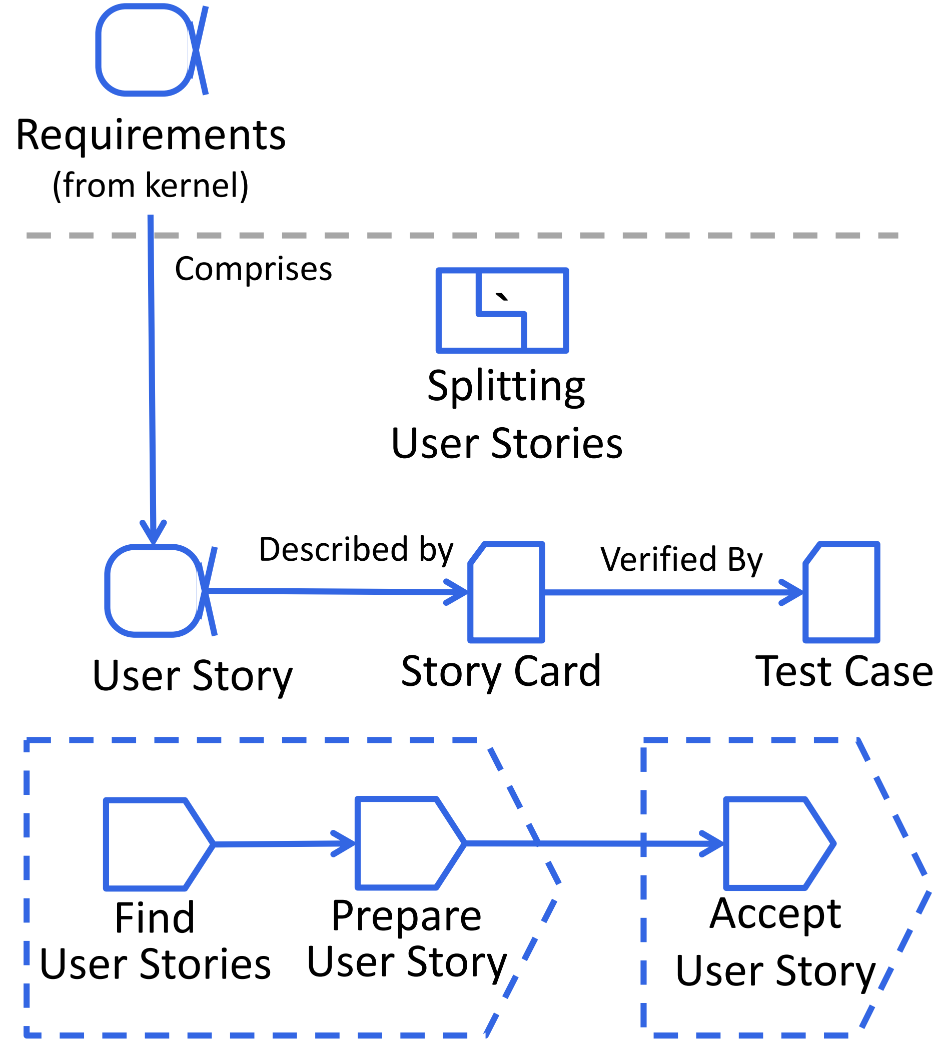
- …
- …
-

- **Card** – a succinct headline description, as captured on a Story Card
- **Conversation** – the actual users of the proposed system and developers discuss what is needed to converge on the best solution
- **Confirmation** – acceptance criteria, captured as bullet-point statements, which can be captured on the back of the Story Card.

How to write good User Stories: INVEST

1. **Independent** - *independent* of each
2. **Negotiable** – written to promote negotiation between the user and developers
3. **Valuable** - A user story should be *valuable* to the user.
4. **Estimatable** - enough details to allow the developer to estimate the work effort required to implement the story
5. **Small** - to fit within a given iteration
6. **Testable** - when completed it has to be *testable*.

User Stories Lite Practice expressed in Essence



Requirements
(from kernel)

Comprises

Splitting
User Stories

Described by

Verified By

User Story

Story Card

Test Case

Find
User Stories

Prepare
User Story

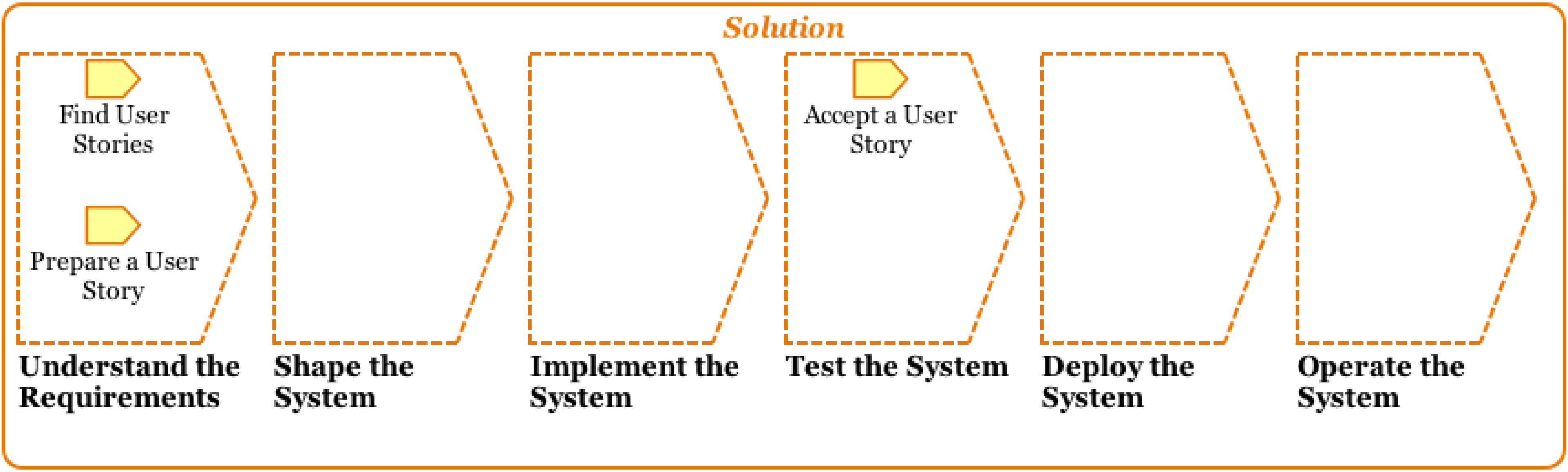
Accept
User Story

Understand the Requirements
(from kernel)`

Test the System
(from kernel)

SEMAT

Impact of User Stories Lite practice for the team



Solution

Find User Stories

Prepare a User Story

Accept a User Story

Understand the Requirements

Shape the System

Implement the System

Test the System

Deploy the System

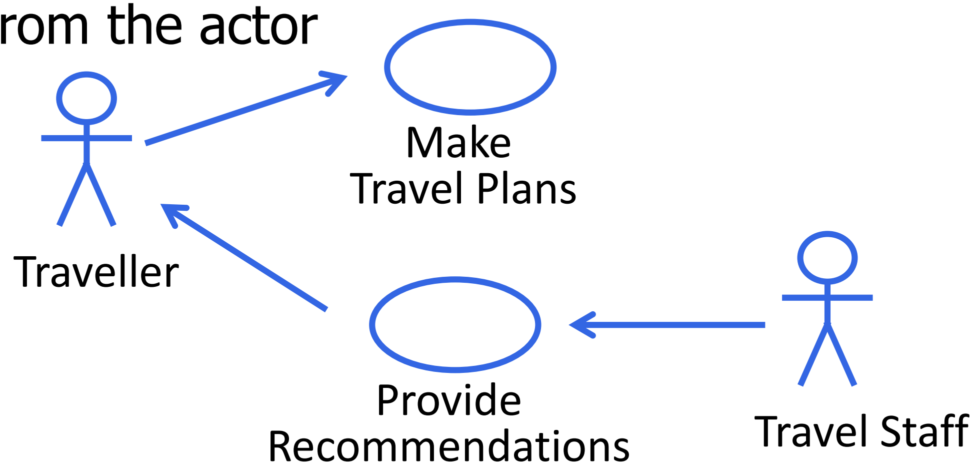
Operate the System

Use Case Explained

- A use case is all the ways of using a system to achieve a particular goal for a particular user
- How do they compare with User Stories?
  - User stories represent stories (each one is a single scenario or example of story) of using the system
  - Use Stories help flesh out missing requirements by encouraging informal discussion between developers and users, yet:
    - Many User stories … sometime are too many
    - It is not clear how these user stories make up something complete
    - Lack of structure
  - Epics can help consolidate many stories and provide structure
- Use cases give requirements a structure, or a systematic way to organize requirements
  - This structure makes it easier for teams to conduct analysis, user interface (UI) design, service design, implementation, tests and so on

Use Case Models

- In the Unified Modeling Language (UML) the relationships between users and use cases are represented in what is referred to as a Use-Case Model
- Use cases include the actual functionality and behaviour of the system.
  - Each use case is described in a Use-Case Narrative.
  - A use case narrative provides a textual description of the sequence of interactions between the actor and the system.
  - It also describes what the system does as a response to each message from the actor



Use Cases narrative

- The use case narrative is usually separated into two parts referred to as the basic flow and the alternate flows.
- The **basic flow** describes a normal or a “basic” use of the described use case often called the “happy day scenario”.
  - The basic flow is worded in a way you would test and verify the behaviour of the functionality.
  - It is a sequence of steps you would expect when using or testing the system.
- The **alternate flows** are variations of the basic flow to deal with more specific cases
  - These variations can be enhancements, special cases, etc.
  - There are multiple alternate flows

Use Case Narrative Examples

UC **Make Travel Plans**

- **Basic Flow**:
  1. Traveller provides travel details (travel dates and destination)
  2. System searches and displays hotels for the travel dates and destination.
  3. Traveller selects a hotel and room type.
  4. System computes and display.
  5. System makes a tentative reservation for the traveler on the selected hotel.
- **Alternate Flows**:

  A1. Travel plan with multiple destinations

  A2. Travel plan having a single destination but non-consecutive dates

  A3. Travel plan with non-consecutive dates and multiple destinations

UC **Provide Travel Recommendations**

- **Basic Flow**:
  1. Traveller verifies travel details (travel dates and destination)
  2. Traveller requests recommendations
  3. System provides list of recommendations
  4. Traveller browse recommendations
  5. Traveller selects and view recommendation.
- **Alternate Flows**:

  A1. Recommendations of different entities
  - a. Hotel, b. Place of Interest

  A2. Recommendations
  - Recommendations based on (a) popularity rating, (b) on pricing,

  A3. Recommendation request trigger
  - (a) User initiated, (b) System triggered

  A4. Sorting of recommendations
  - (a) Sorting based on prices

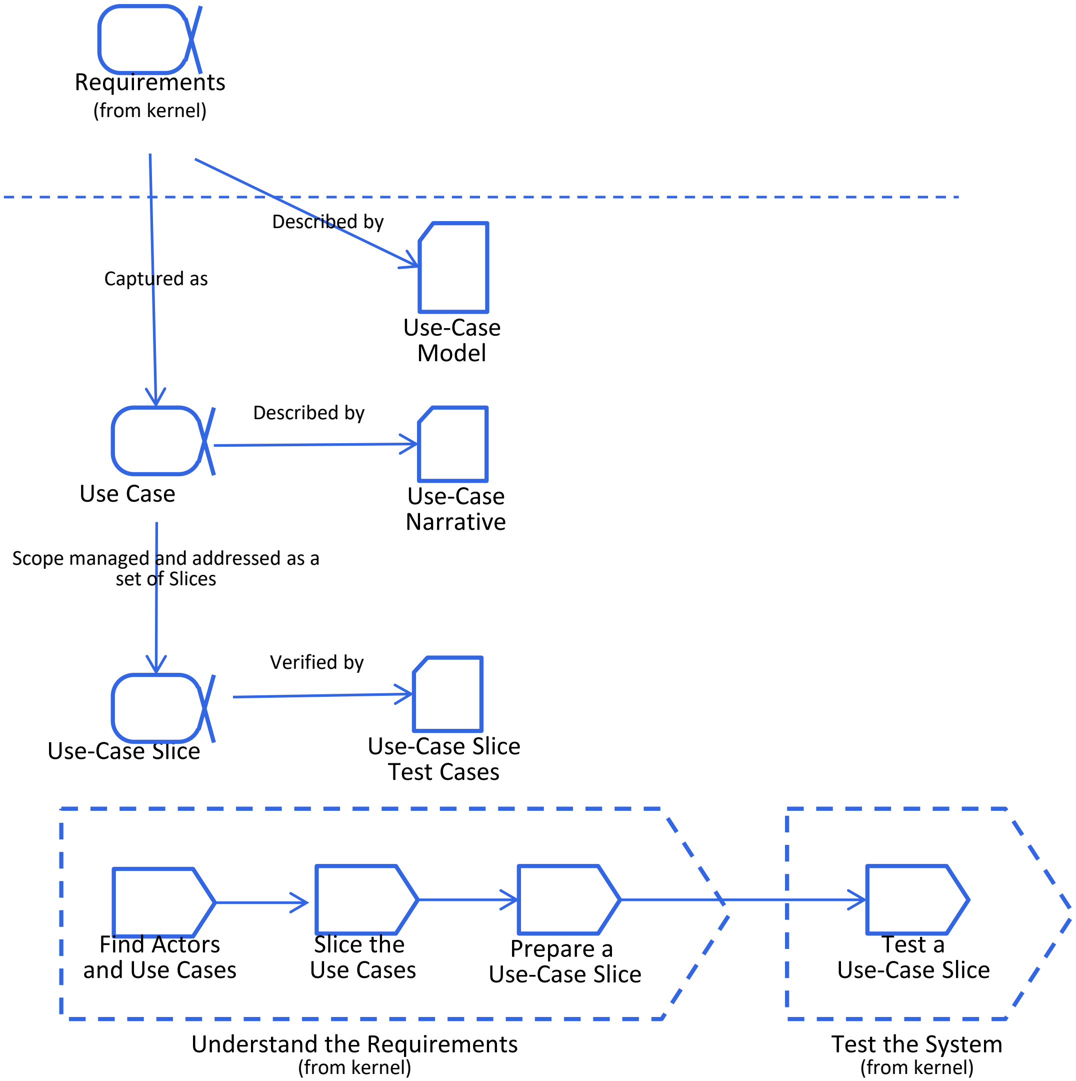
Use Case Considerations

1. The use cases help you see the big picture through the use-case model
2. The use case approach provides structure through the separation of basic and alternate flows
3. A use case often contains too much functionality to be developed in one iteration, such as a single sprint when using Scrum.

Use Case Slices

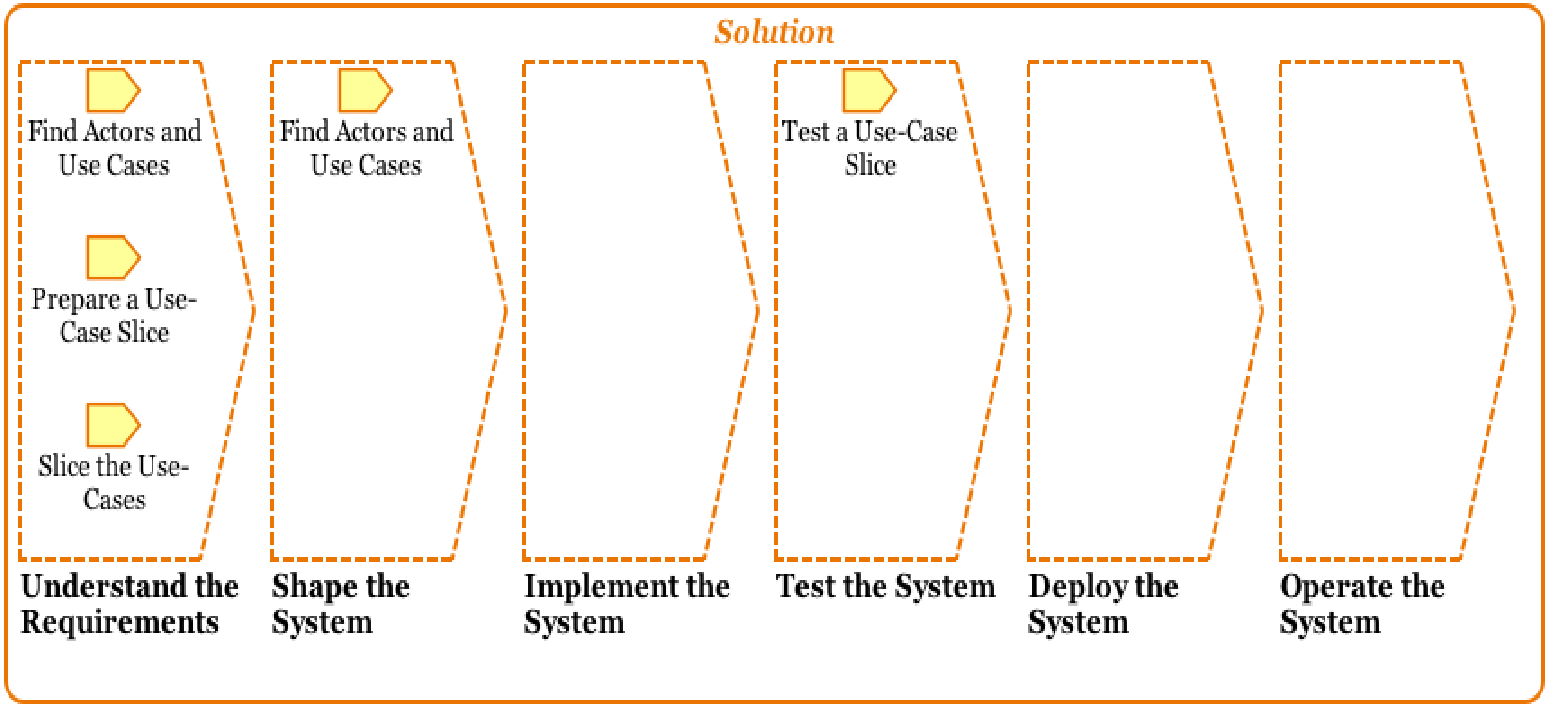
- A **use-case slice is a slice through a use case that is meaningful to describe, design, implement and test in one go**.
  - It doesn't need to by itself give value to a user, but together with all other slices of the same use case, the value is achieved.
  - For example, the basic flow of a use case is a good candidate to become an early use-case slice.
  - Additional slices can then be added to complete the whole use case later.
- The slicing mechanism enables you to create slices as big or small as you need **to drive your development**.
  - The use-case slices include more than just the requirements.
  - They also slice through all the other aspects of the system (e.g. user experience (user interface), architecture, design, code, test) and its documentation.

Use Case Lite Practice expressed in Essence Language



Impact of use case for the team

- Use cases helps the team found they could see the big picture of the system better through their use case diagrams and use case model



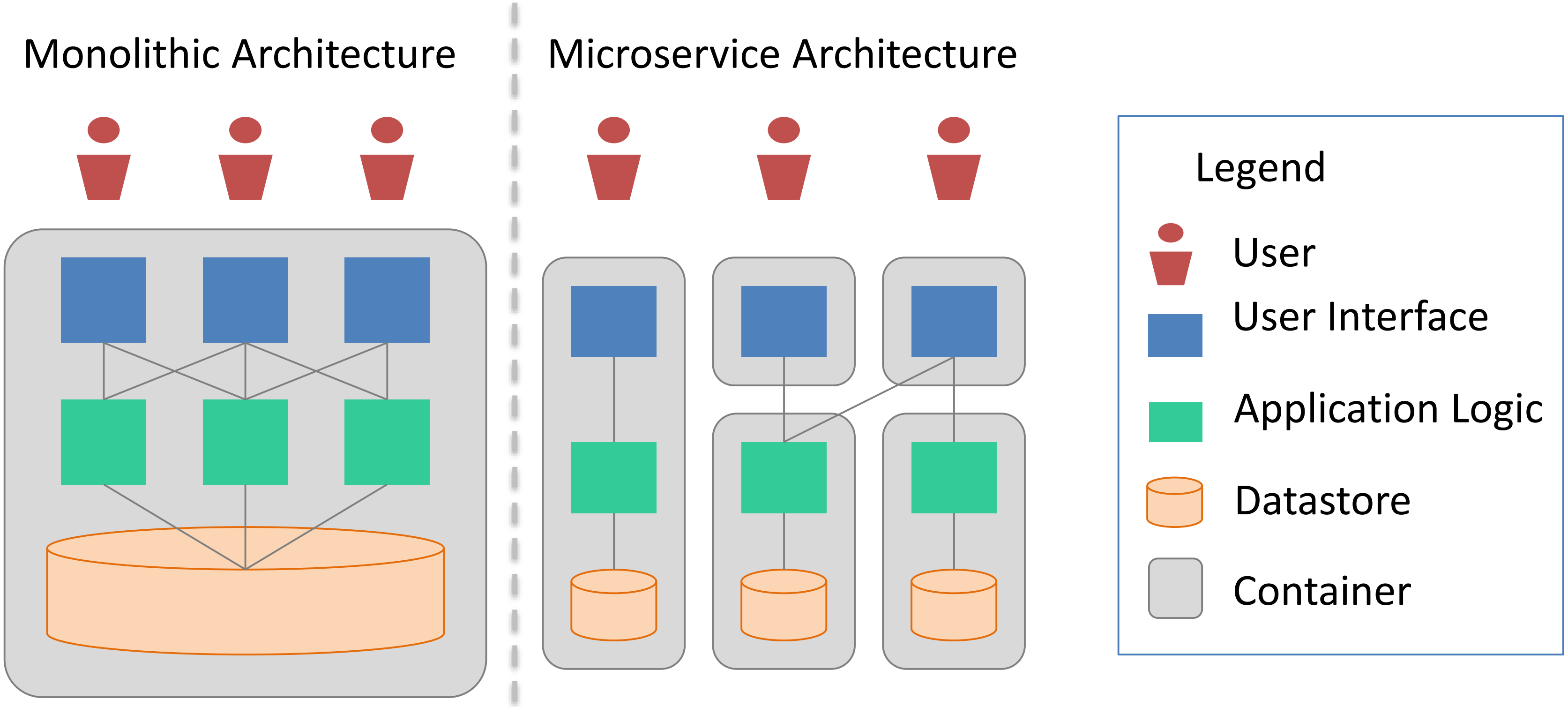
Introduction to Microservices concept

- Developing software with **microservices is an evolution of designing software with components**
  - which facilitates a modular approach to building the software system.
- A **component** can be described as an element containing code and data.
  - Only the code "inside" the component (object) can modify the data inside the object and it does that when another component sends a message with a request to do that.
  - This idea is known as "**data hiding**" (data is hidden to other components) and is an accepted best practice in developing maintainable software.
- What Microservices has added is support for components all the way from design, code to deployment

Components vs Microservices

- Like components
  - microservices are interconnected via interfaces over which messages are sent to allow communication.
  - each microservice can evolve separately from other microservices, thus making it easy to introduce new functionality
- In a software system built from microservice, each microservice runs a unique executing process.
  - There may be several such executions or instances of the same program running in parallel.
- What microservices bring to software beyond what components already did is the **ability to also deploy the microservices independently without stopping the execution of the entire software system**.

Microservices Big Picture

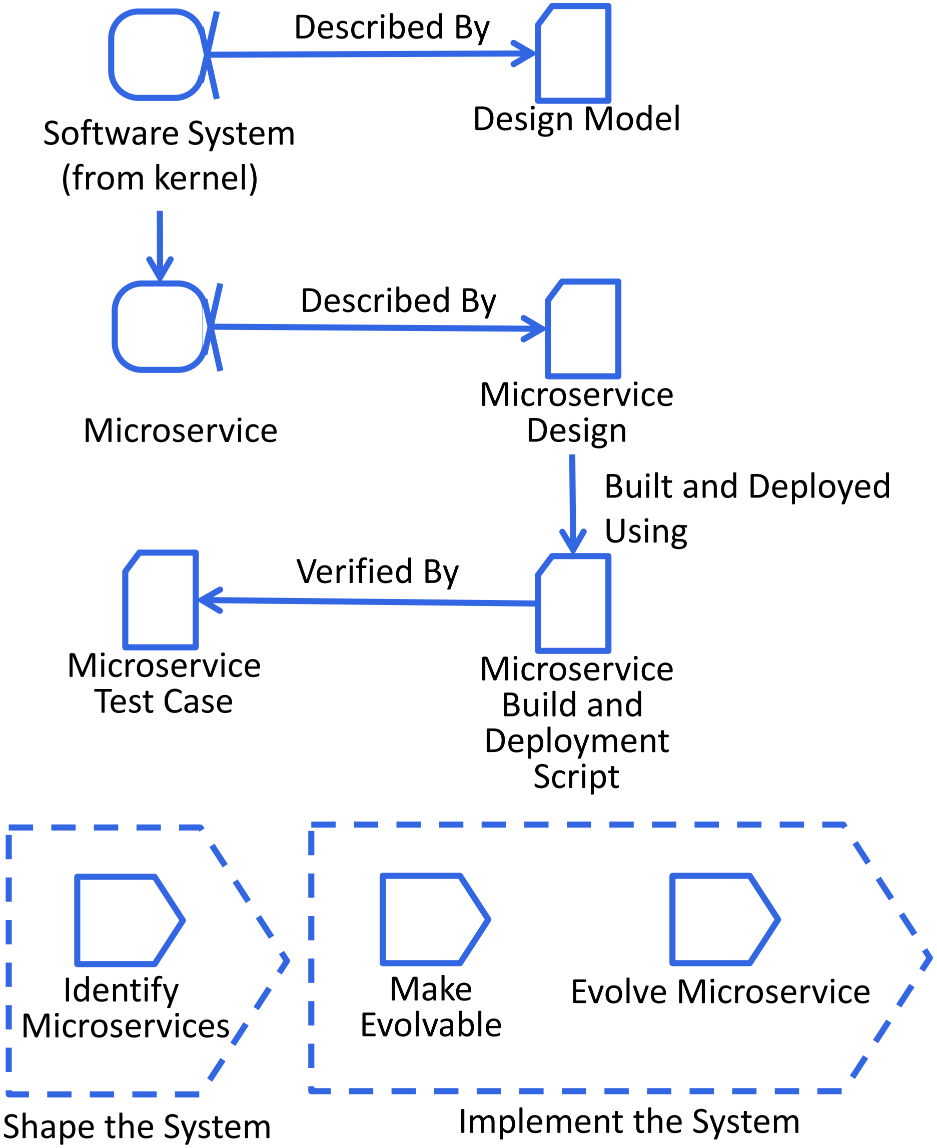


- **User Interface** – A user interface is the part of a software system that users interact with. It is the screens, and buttons, and so on.
- **Application Logic** – The code behind the user interface that performs computation, move data around, etc.
- **Data Store** – The data retrievable by the application logic lives in a data store.
- **Containers** – Containers are components of a software system that can be managed separately (i.e. started, stopped, upgraded, and so on)

Advantages of Microservices

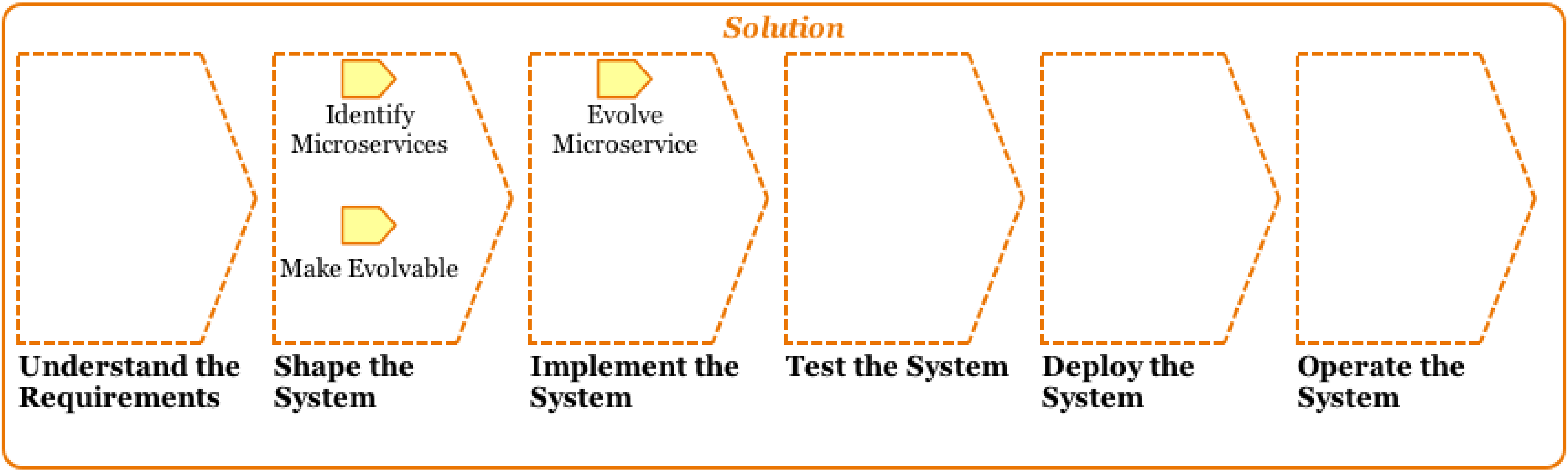
- Each microservice runs
  - as a separate process,
  - possibly in its own container or virtual machine
  - it has its own programming language, user interface, application logic and data store.
- This architecture allow developers to upgrade each microservice independently
  - For example you can upgraded a microservices from Java 8 to Java 9 or a data store without impacting other microservices.
  - If however, code for different logical software element were to run in the same process or virtual machine, an upgrade of one element may inadvertently impact another element.
  - Thus, **enhancing the functionality of an existing microservice is easier** than that of a monolithic software system.

Microservices Lite Practice expressed in Essence



Impact of Microservices practice to the team

- The Microservice Lite practice addresses implementation guidance
  - Providing guidance on Shape the System and Implement the System activity spaces



Microservices Considerations

- Having many microservices each running separately raises other problems such as:
  - how to manage and coordinate their execution
  - how to propagate the change of data from one microservice to other microservices
  - how to manage the security of each microservice
  - and so on.

- Cloud providers offer standard mechanisms to solve them.
  - This allows developers to focus on the application, realizing user requirements, push out functionality to the users quickly, get user feedback, and innovate
    - Rather than the low-level infrastructure plumbing which now happens behind the scenes
    - Such rapid cycles are ultimately the value of microservices