Software Engineering

CS305, Autumn 2020

Week 7

Class Progress…

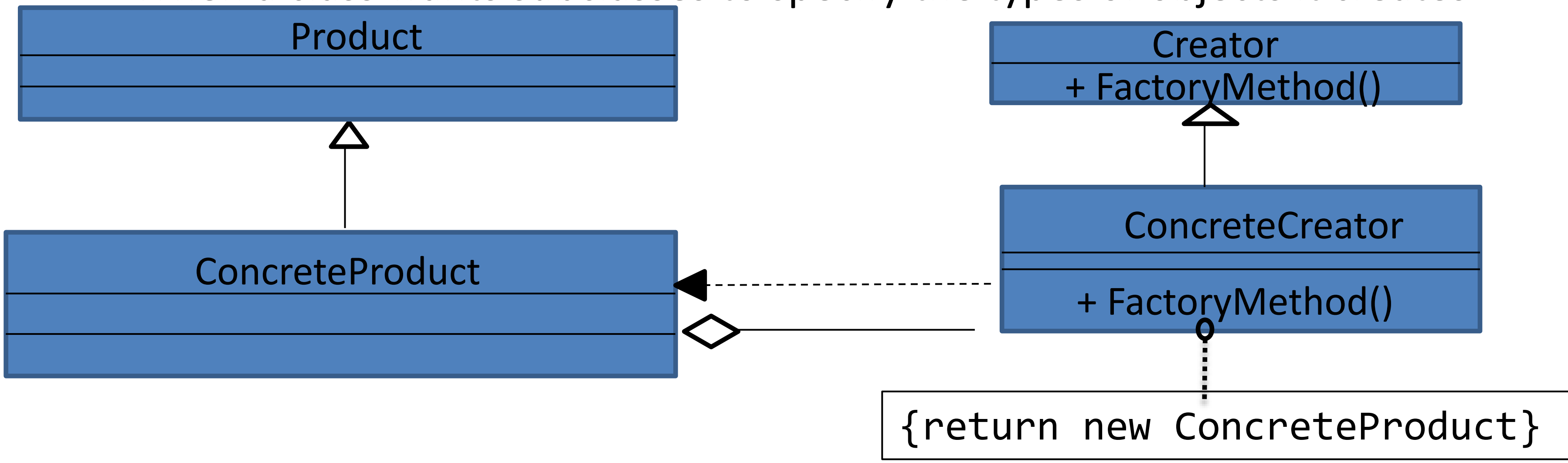
- Last week:
  - Architectural styles
    - Shared services and servers, repository, layered
  - Detailed design
    - Design patterns
    - Singleton

Class Progress…

- This week:
  - Design patterns, Design principles, Rational Unified Process

Factory Method Pattern

- Intent: define an interface for creating an object, and let applications decide which object type to create.
- Applicability
  - When the exact type of object to be created is known at runtime
  - When a class needs control over object creation
  - When a class wants subclasses to specify the types of objects it creates



Factory Method Pattern

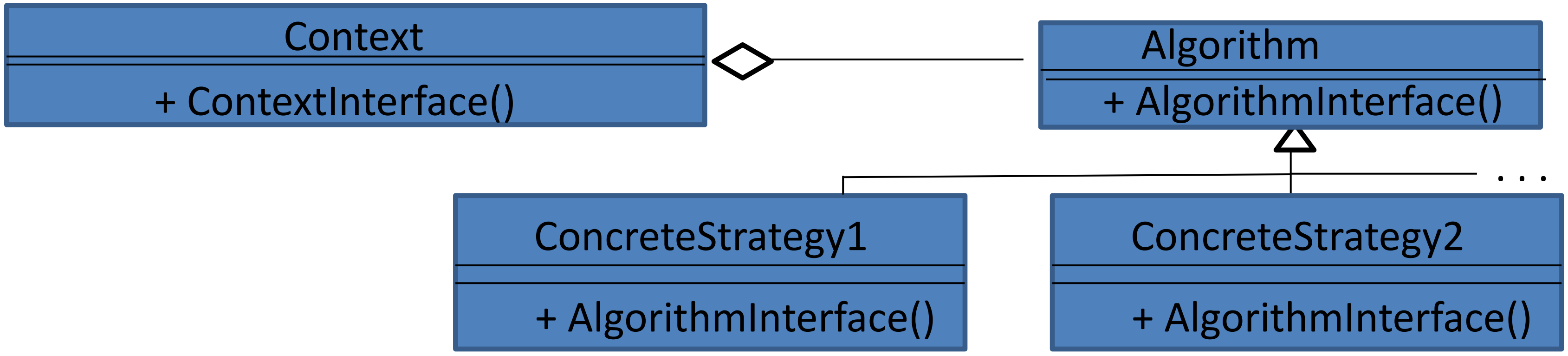
```
Vehicle* VehicleFactory(VehicleType type, Color c) {
        if(type == BUS)
                return new Bus(c);
        else if(type == CAR)
                return new Car(c);
        else
                return NULL;
}

int main() {
        Vehicle* redCar = VehicleFactory(CAR,RED);
        Vehicle* blueBus = VehicleFactory(BUS,BLUE);
}
```

Comment about the structure of `VehicleFactory`?

Strategy Pattern

- Intent: encapsulate each one of a family of algorithms in a separate class and make their usage agnostic
- Applicability



Some Commonly Used Patterns

- **Visitor** – separate the algorithm from the data structure on which it operates e.g. finding minimum in a binary tree, finding maximum in a binary tree, finding multiples of a given number in a binary tree.
- **Observer** - notify dependents when object changes
- **Iterator** – access elements of a collection without knowing about underlying representation
- **Proxy** – a surrogate controls access to an object

Choosing a Pattern

- Broad guidelines
  - Understand design context
  - Examine the patterns catalogue
  - Identify and study related patterns
  - Apply suitable pattern
- Avoid:
  - Overusing patterns

Design Principles

- **Performance vs. Maintainability tradeoff**
  - **Performance goal:** localize critical operations and minimize communications. Therefore, use coarse-grain rather than fine-grain components. *Coarse-grain components are difficult to maintain*
  - **Maintainability goal:** use fine-grain, replaceable components. *Fine-grain components localize communication*
- **Security vs. Availability tradeoff**
  - **Security goal:** secure critical assets in the inner layers when using a layered architecture.
  - **Availability goal:** include redundant components and mechanisms for fault tolerance. *Redundant components increase availability. However, security becomes difficult.*
- **Safety vs. Communication/Performance tradeoff**
  - **Safety goal:** localize safety-critical features in a small number of sub-systems. *Localizing means more communication and hence, degraded performance.*

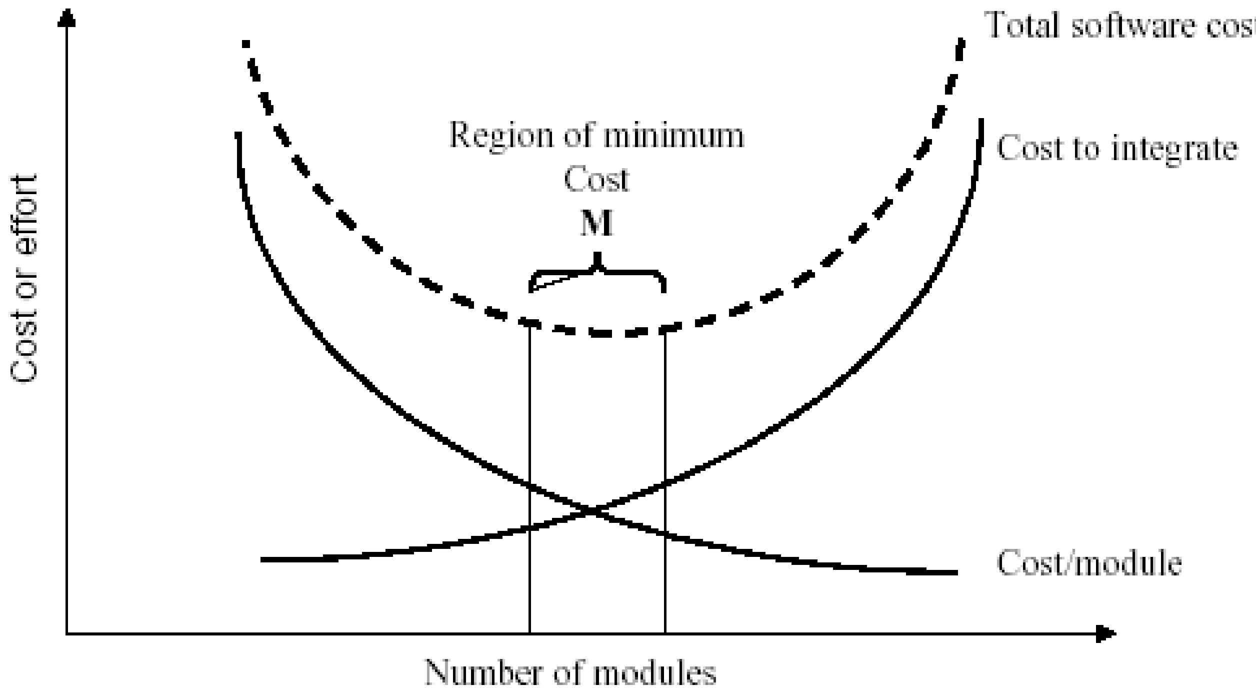
Design Principles

- Balance coupling and cohesion with non-functional requirements
- Consider information hiding
  - Provide abstraction / refinement
- Document design decisions *(design rationale)*

Design Principles

- *Coupling vs. Cohesion*
  - Recall that coupling is the extent to which two components depend on each other for successful execution. *Low coupling is good*
  - *Recall that Cohesion* is the extent to which a component has a single purpose or function. *High cohesion is good*

Modularity and Software Cost



- consider low coupling/high cohesion
  - module should be ‘stand alone’, errors contained as much as possible
- consider requirements
  - change in requirements should minimize number of modules affected

Design Decisions - dimensions

- Architecture level:
  - Choose from repository, service, layered, …
- Component level:
  - identify components
- Connector level: determine control model
  - Choose from centralized, event-driven, …
- Subsystem level:
  - Choose from behavioral, object, … models

Design Principles - SOLID

- Single-responsibility Principle
  - *"A class should have single responsibility"* – to prevent from side-effects resulting from future requirements changes
- Open-Closed Principle
  - *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."* – should be able to add new functionality without modifying existing code

Design Principles - SOLID

- Liskov-Substitution Principle
  - objects of a superclass shall be replaceable with objects of its subclasses without breaking the application
  - Pretty similar to Bertrand Meyer's *design-by-contract* principle
- Interface Segregation principle
  - *"Clients should not be forced to depend upon interfaces that they do not use."*
- Dependency Inversion Principle
  1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  2. Abstractions should not depend on details. Details should depend on abstractions.

Further reading: https://stackify.com/solid-design-principles/

- *So far:*
  - *Requirements Modelling, Analysis, and Design in detail and little bit of Coding and Functional Testing*

- *Next:*
  - *a software process model that binds these activities and other ones in SDLC*

Unified Software Process

- The starting point was in 1997 when Rational proposed 6 best practices in modern software engineering
- It is a generic framework rather than a process
  - Rational Unified Process (RUP) is a refinement and the best known example of Unified Software Process
  - OpenUP, Agile Unified Process are other examples

6 Best Practices in Software Engineering

1. Develop iteratively with risk as the primary driver for the iteration
2. Manage requirements - updating and maintaining traceability information that associates requirements with other artifacts
3. Employ a component-based architecture – high-level design involving components and their interactions.
4. Model software visually - use visual diagrams e.g. UML diagrams so that the artifacts can be easier to understand and agreed upon among stakeholders
5. Continuously verify quality throughout development process
6. Control changes using change management tools

RUP – Key features

- Process model
  - Describes ordered set of phases and when to transition from one phase to another
- Component based
  - Components are the building blocks and well-defined interfaces must exist to enable inter-component communication
- Tightly related to UML
  - Relies extensively on UML diagrams and notation
- Distinguishing features
  - Use-case driven,
  - architecture-centric, and
  - iterative and incremental

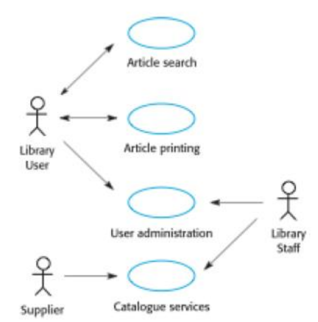
RUP – Distinguishing features

- Use case driven
  - Use cases are central elements of the entire RUP lifecycle
  - A software system performs some sequence of actions in response to user inputs
  - Recall that use cases capture these interactions and for each user

Use cases

- UML's scenario-based technique
  - actors and interactions
- Should describe all possible interactions with the system
- Sequence diagrams may be used to add details to use-cases

LIBSYS use cases

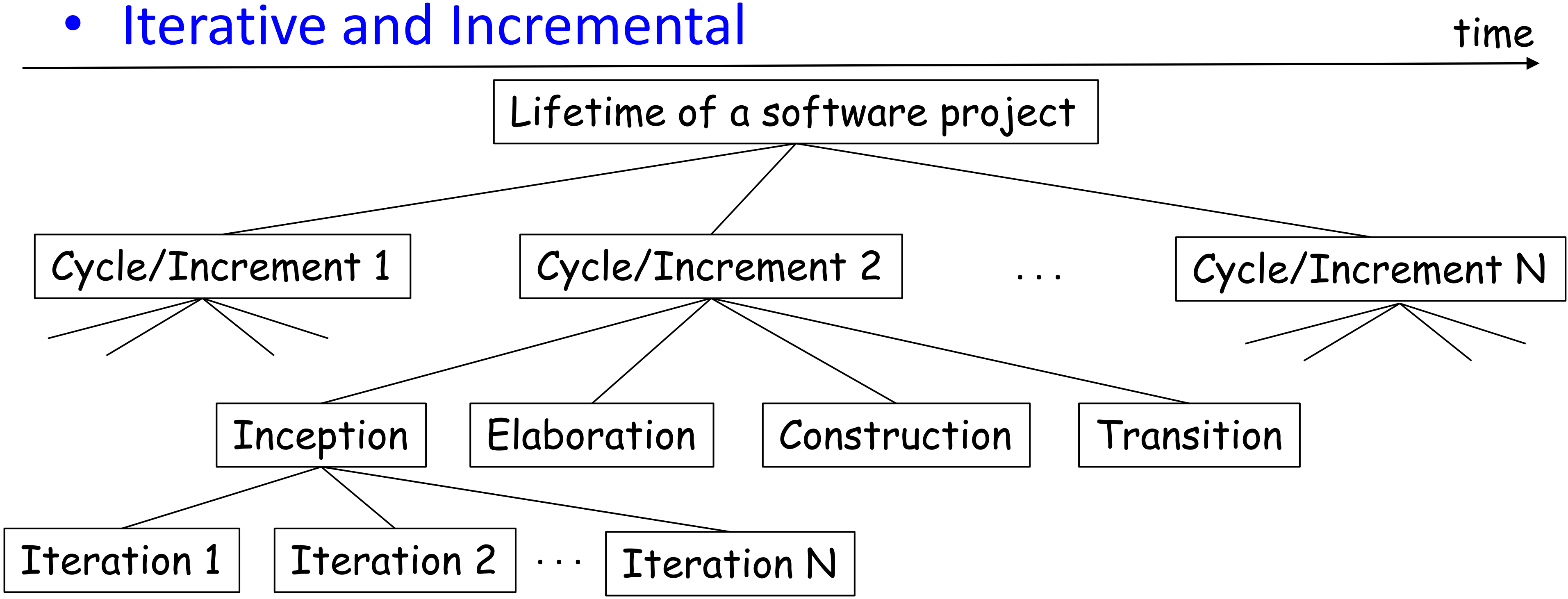


RUP – Distinguishing features

- Architecture centric
  - Architecture is the **high-level view of the principal design decisions** that you make
  - Use cases define **functionality,** whereas architecture defines **form** i.e. how the software must be structured to provide that functionality
  - Focuses on an incremental / iterative approach:
    - First **prepare a rough outline** of the system e.g. what platform to run on? What styles to choose from? etc.
    - Next **pick a key use case** and draw the model e.g. withdrawal feature in Banking system
    - Then **refine the architecture** by adding additional use cases

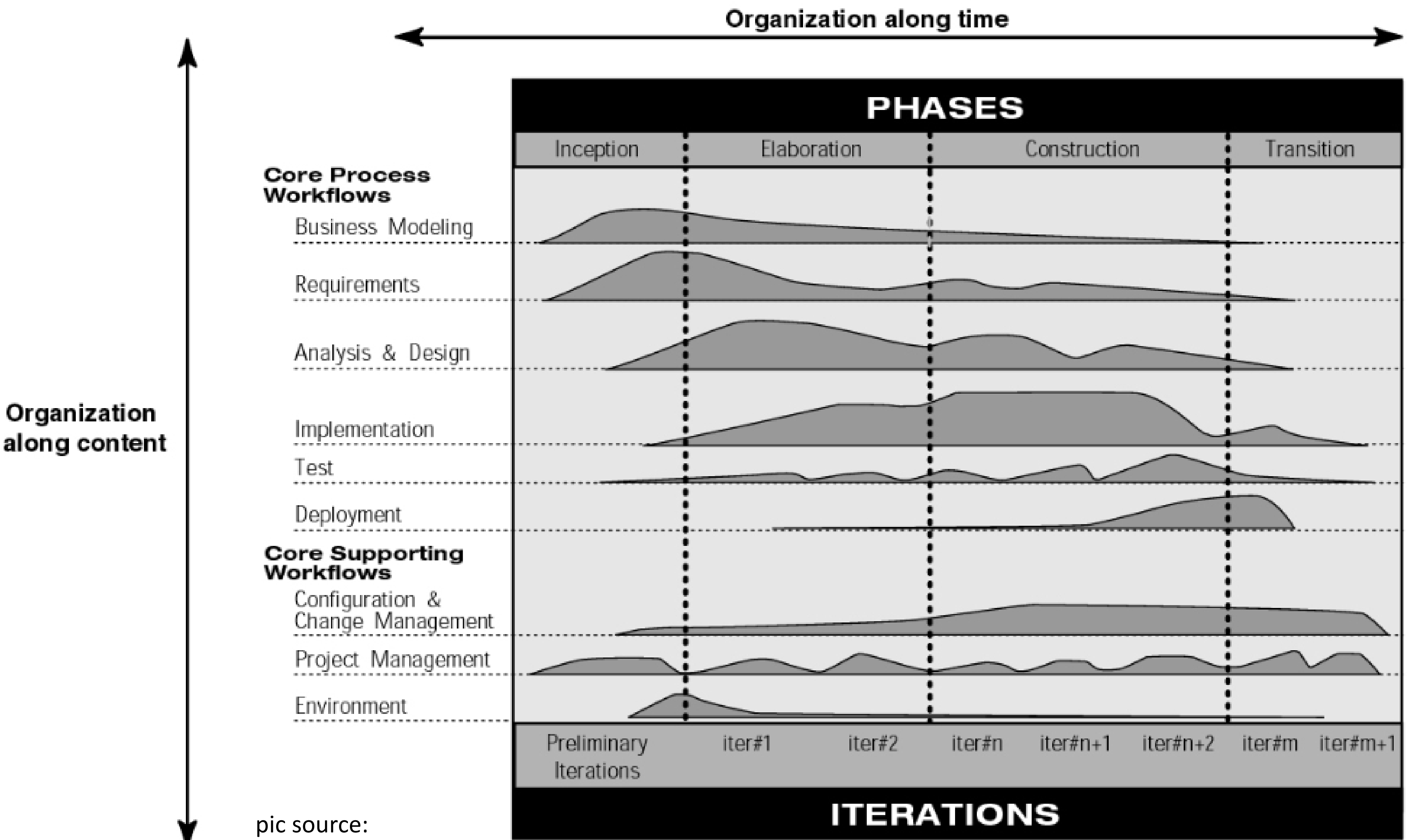
RUP – Distinguishing features

- Iterative and Incremental



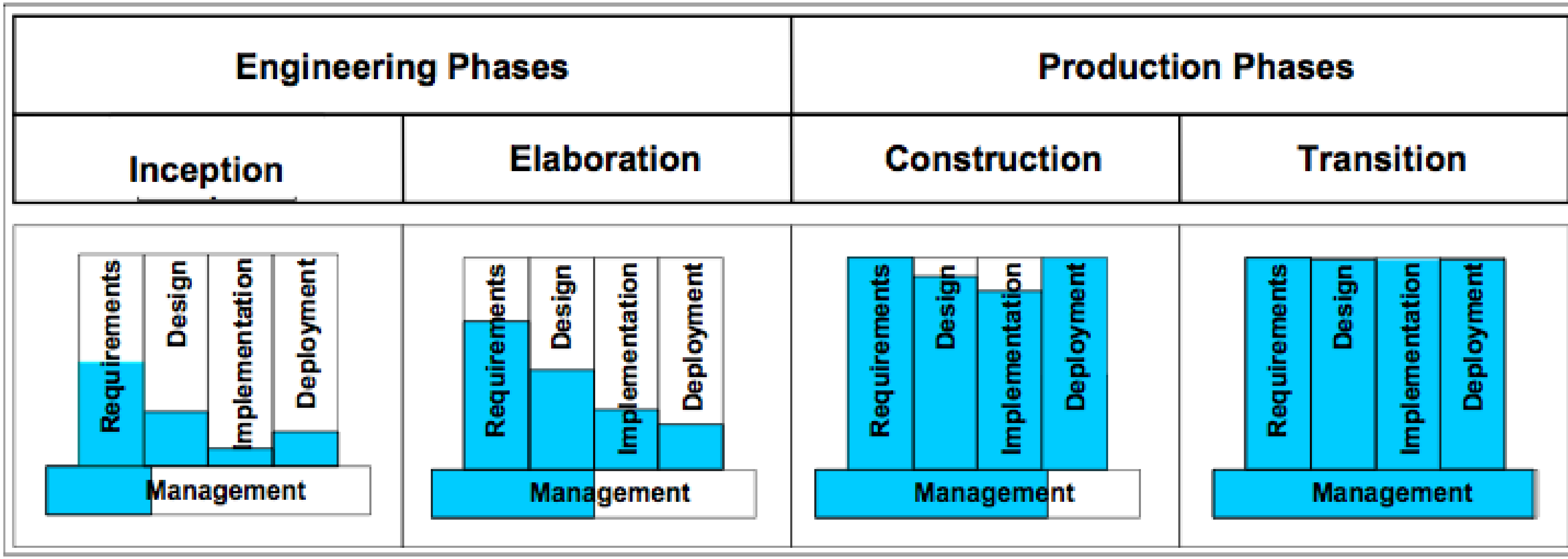
- An RUP life cycle is broken down into multiple cycles (also called as increments)
- Each cycle / increment includes all phases (inception, elaboration, construction, transition) of the process. Hence, results in a product release (internal / external)
- Each phase involves multiple iterations that identify a use case

RUP Phases



pic source:
https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

RUP Phases – a different perspective



pic source: Alex Orso, CS3300 and material from Ian Sommerville and Spencer Rugaber

RUP Iterations

- What happens in an iteration?

1. Define uses cases - which pieces of functionality this iteration represents
2. Followed by design that is guided by the chosen arch. (use cases + architecture = design)
3. Then implement the design that results in software components
4. Then verify components against use cases - testing or other
5. Then release - most often the release is for internal / stakeholders to get feedback

   Release contains requirements spec, code, manuals, use cases, non-functional specs, test cases,