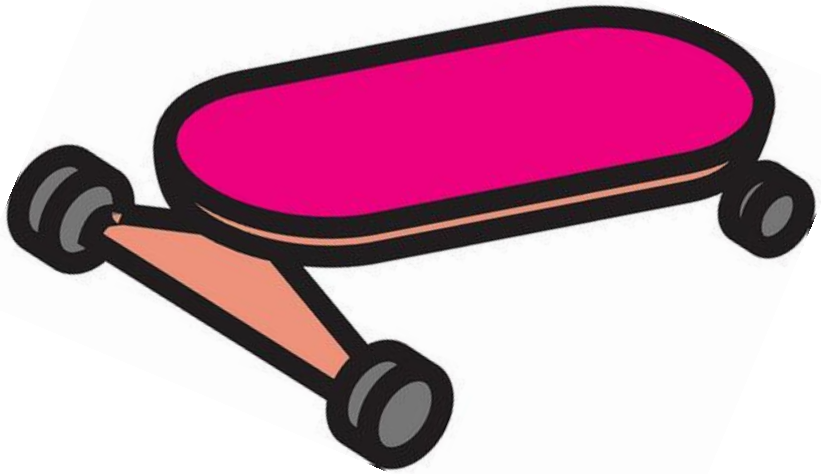
SWEN-383

Software Design Principles & Patterns

Introduction

Design(?)

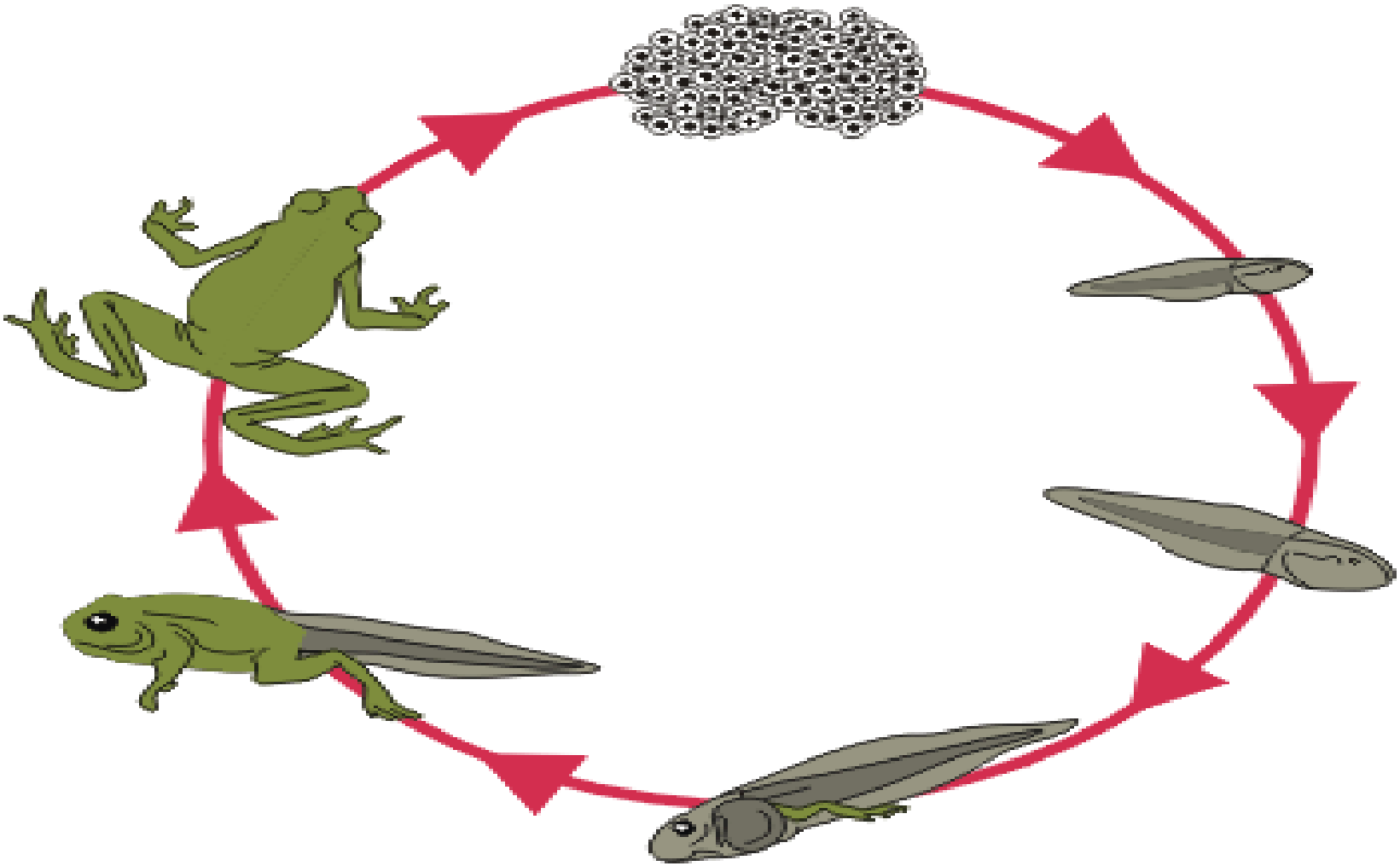


Design

"Designs are not **good** or **bad**, they are more or less ***useful***."

- *Kent Beck (Well Known SW Developer)*

Life-Cycle



Process

Software Development Life-Cycle

ALL software development projects, big and small, follow a process that defines the activities which are performed in each of the project's life-cycle phases leading to product release.

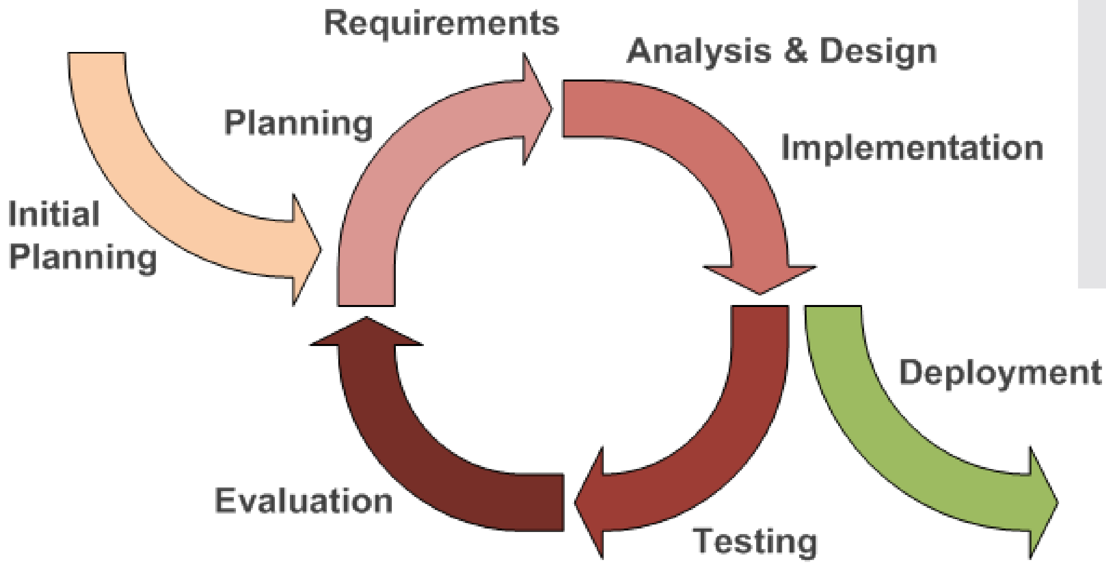
(If you choose not to decide on a process, then you still have made a choice.)

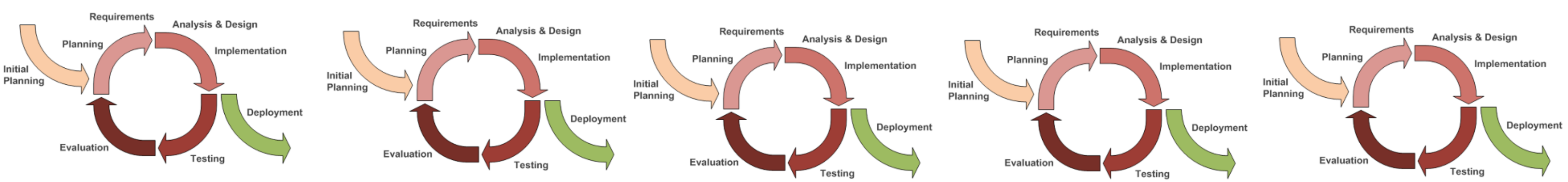
- "Freewill", Rush – Permanent Waves, 1980

Product Life Cycle Phases

- Requirements
- Analysis
- Design
- Implementation/Construction (code)
- Test / Evaluation
- Deployment & Maintenance

Iterative & Incremental Process





R1 R2 R3 . . . Rn

Upstream vs Downstream

The “flow” of development activities from early to later activities:

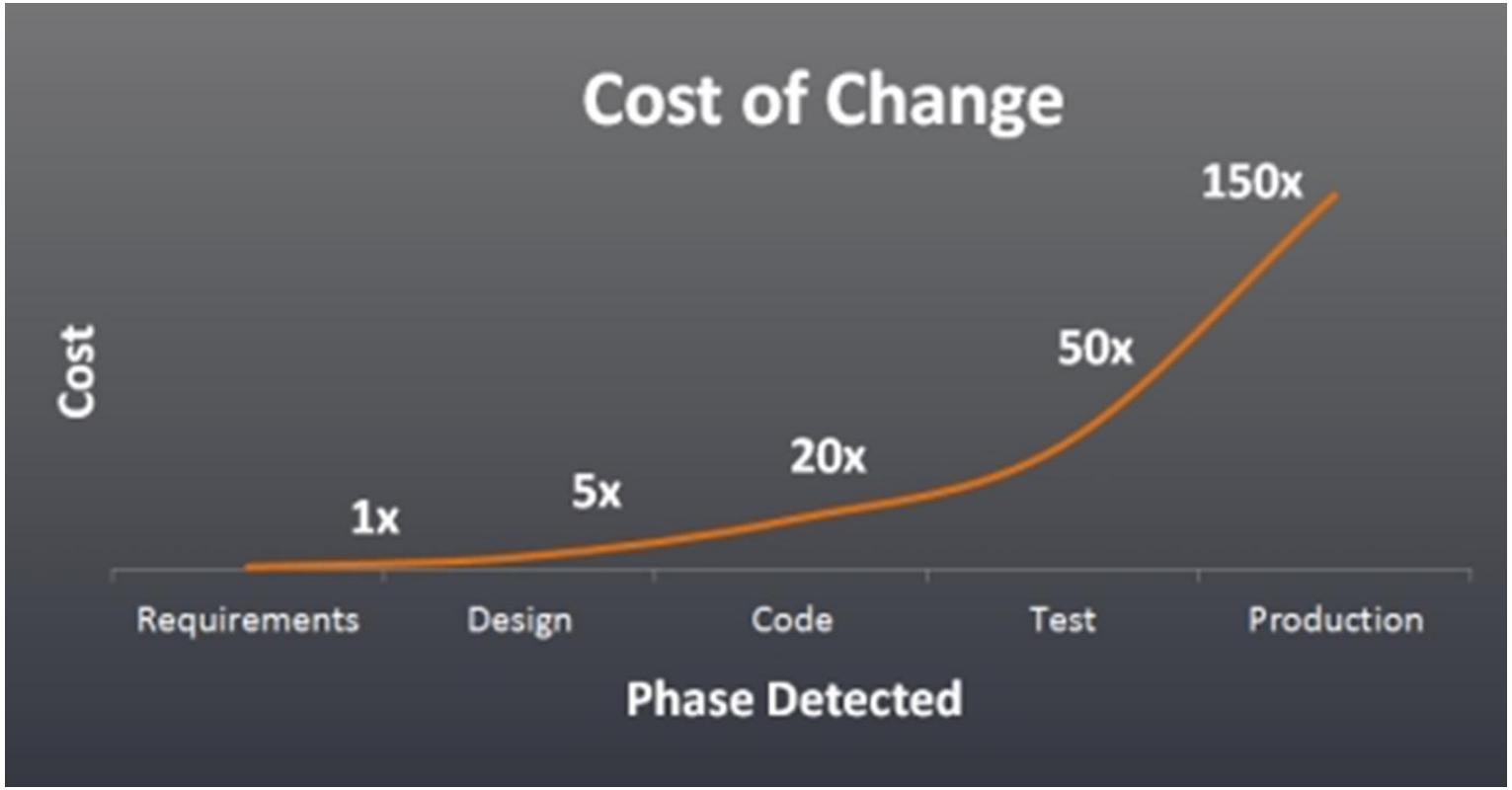
- Increment of Product Life-Cycle:
  - R1 -> R2 -> … Major Release
- Iteration of Life-Cycle:
  - Req -> Design -> Code -> Test -> Deploy

Car in for repair?

Managed Change

Economics of SW Development

Cost ($$) = Resources(people) x Time



Software Product Requirements

- **Functional** Requirements – "what" the system needs to do.
  - *"Customer can withdraw funds from ATM"*
- **Non-Functional** Requirements – "how" the system satisfies a requirement.
  - *"Customer transactions occur securely"*

Non-Functional Requirements

- User/Product perspective:
  - Usability, Performance, Security, Reliability
- Developer/Product perspective:
  - Maintainability, Extendibility, Scalability, Testability
- Non-functional requirements = “ilaties”

Non-functional requirements drive design.

Design

"Designs are not **good** or **bad**, they are more or less ***useful***."

- *Kent Beck (Well Known SW Developer)*

A ***useful design*** allows developers to efficiently (cost effective) address product ***changes***, especially in downstream activities – i.e. after deployment.

Terminology Soup = concepts that give it flavor

WeatherStation

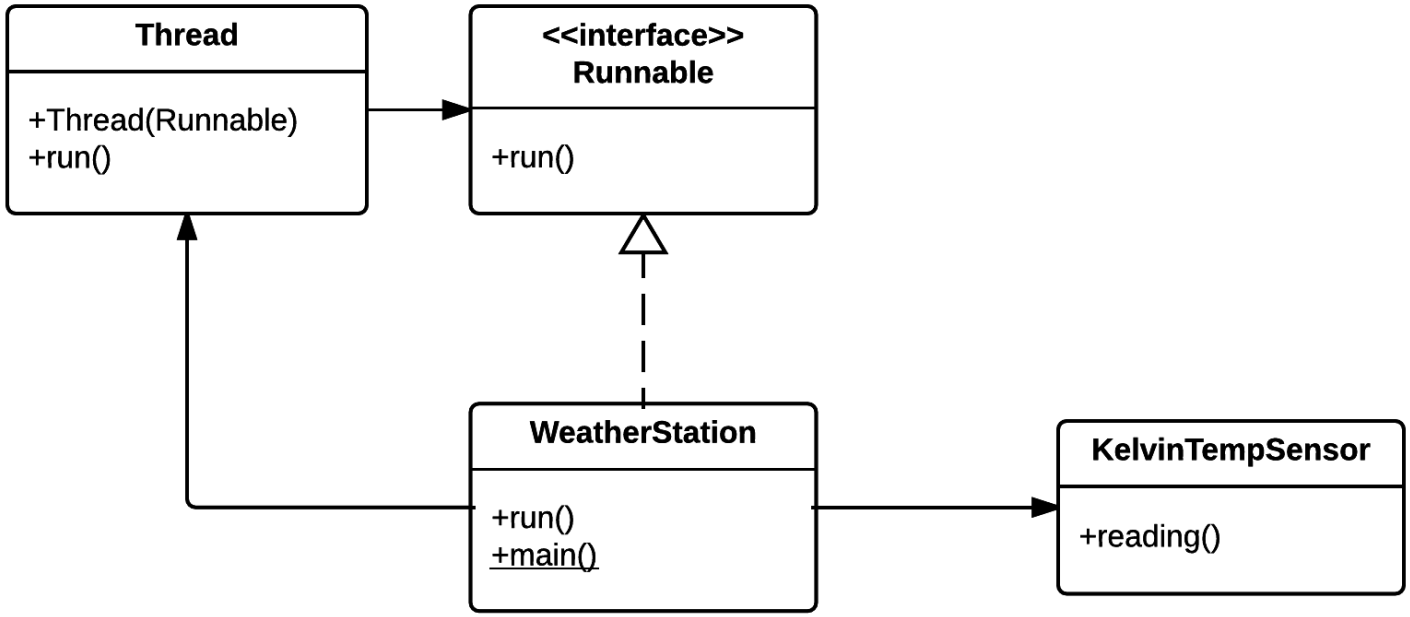
- Example to guide us through practice

REFACTORING

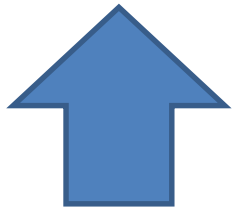
- When SHOULD we do it?
- When DO we do it?
- What are the benefits?

TECHNICAL DEBT

MODELing → Abstraction

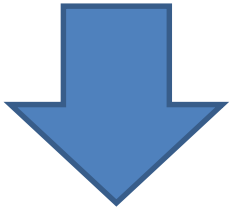

Thread
+Thread(Runnable)
+run()
<<interface>>
Runnable
+run()
WeatherStation
+run()
+main()
KelvinTempSensor
+reading()

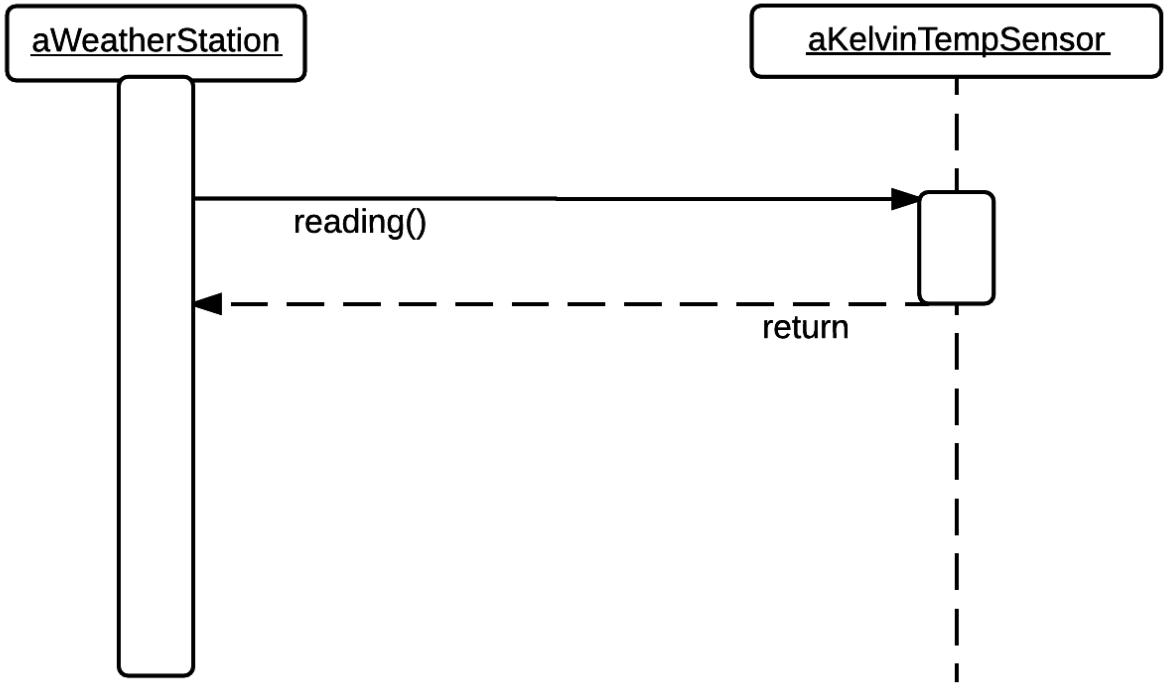
Measures of good Design

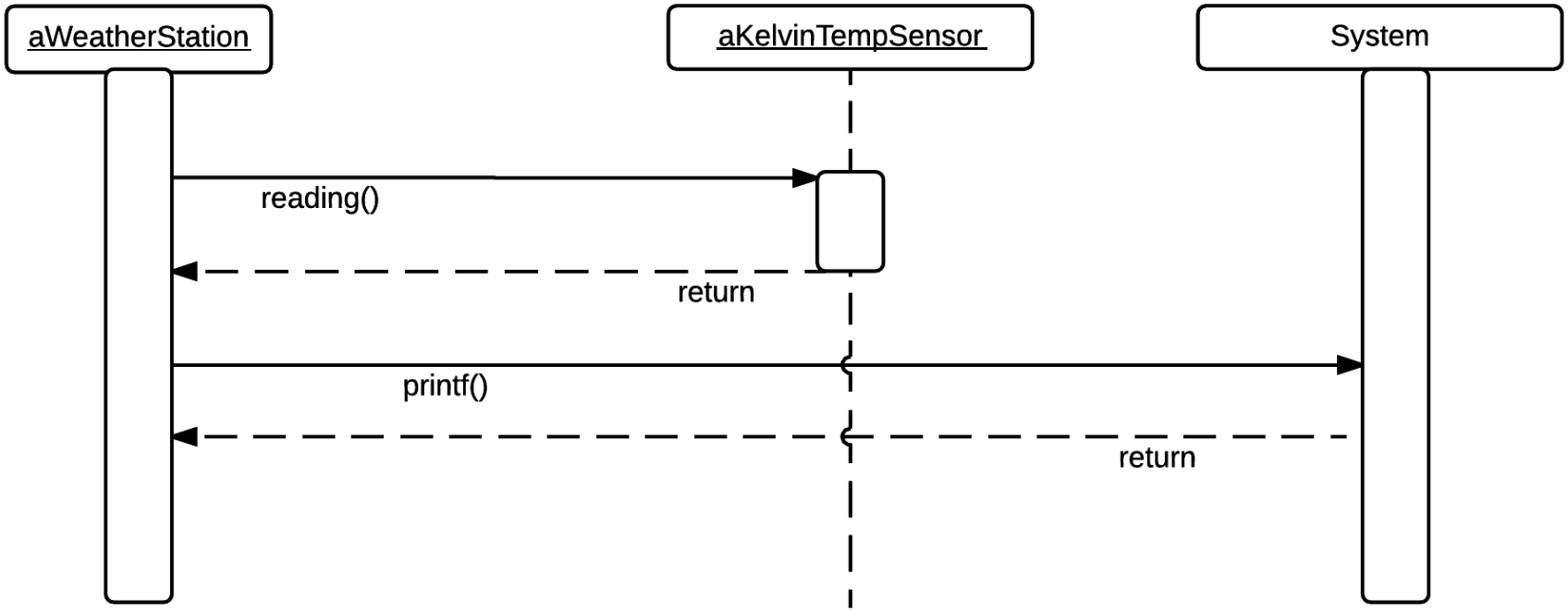


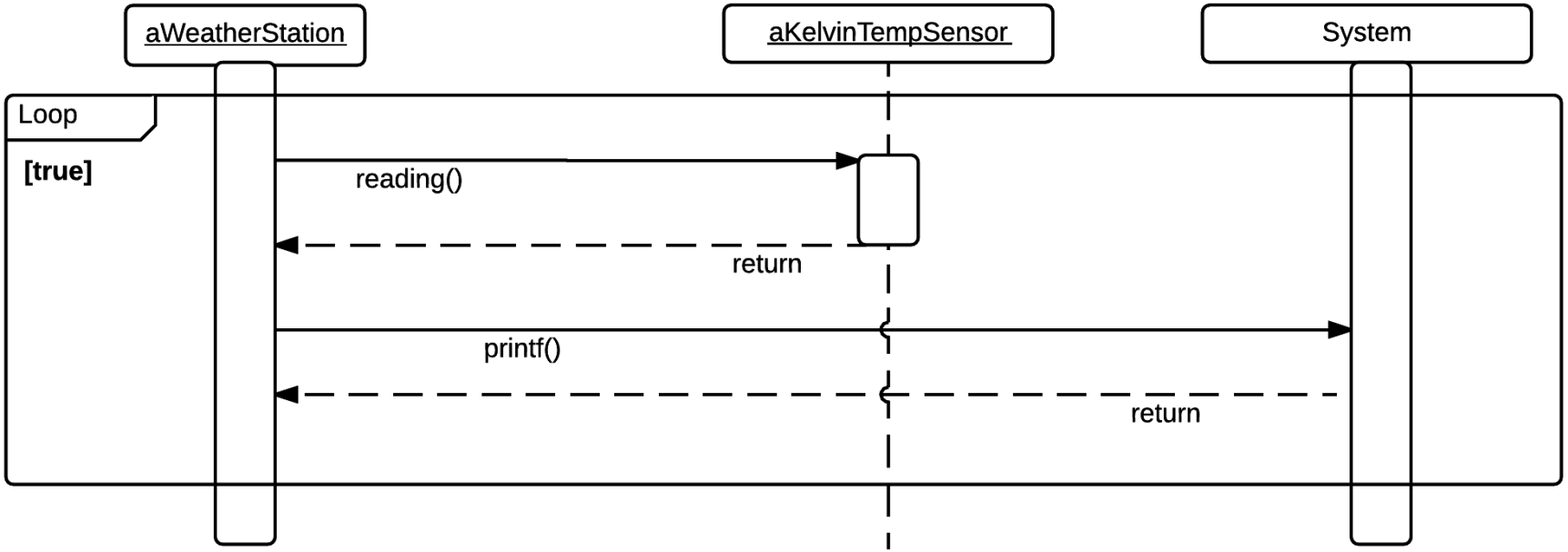
Cohesion

Coupling




aWeatherStation
aKelvinTempSensor
reading()
return


aWeatherStation
aKelvinTempSensor
System
reading()
return
printf()
return


aWeatherStation
aKelvinTempSensor
System
Loop
[true]
reading()
return
printf()
return

Software Design Principles & Patterns

- Design
  - Design principles and trade-offs
  - Design pattern catalog
    - Intent, structure, behavior, implementation, trade-offs
  - Design notation
    - UML class diagrams
    - UML object diagrams
    - UML sequence diagrams
  - Design quality
    - Technical debt, code smells, anti-patterns, refactoring
- Problem-based learning
  - Team design problem
  - Version control

Design Principles

- High cohesion
- Loose coupling
- Don’t Repeat Yourself
- Encapsulation
- Separation of Concerns
- Single Responsibility Principle
- Law of Demeter
- Favor composition over inheritance
- Delegation
- Program to an interface, not the implementation
- etc.

Design Patterns

- Reusable solution to a recurring design problem
  - Balance the design principles
  - Vocabulary for design discourse