Software Platform

Lecture 3

Emanuele Di Buccio

04/03/2024

Master Degree in Computer Engineering, A.A. 2023/2024

In the previous lecture . . .

Previous lecture topics

- Course topics
  - Microservices (what)
  - Frameworks for integration (Spring)
  - Cloud Computing
- More on Software Processes

Scale Cube

- Let us look again at the meaning of the axes
- The Akf Scale Cube
- Scale Cube on [Richardson, 2019]

Software Engineering

Process Paradigms

Process Paradigms

- General process models
- High-level, abstract descriptions of software processes
- Process frameworks that may be extended/adapted to create more specific software engineering processes
- Also called **Software Development Life Cycle models**
- Examples:
  - Waterfall model
  - Incremental development
  - Integration and configuration

Waterfall Model (1)

- Software development process as a number of stages
- Why *waterfall*? Because of the cascade from one phase to another
- Example of plan-driven
- Stages:
    1. requirements analysis and definition
    2. system and software design
    3. implementation and unit testing
    4. integration and system testing
    5. operation and maintenance

Waterfall Model (2)

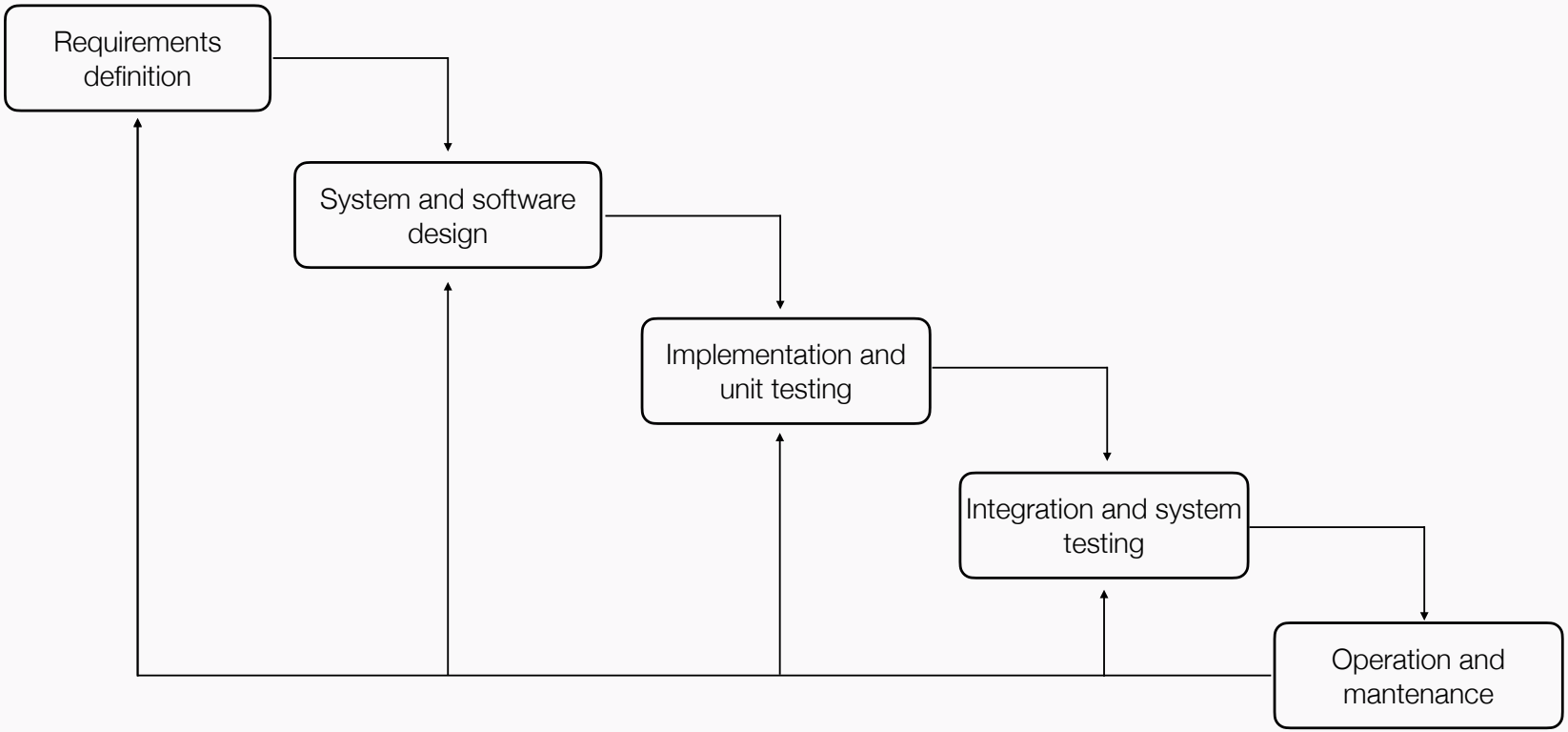


Figure from [Sommerville, 2016]

Waterfall Model (3)

- result of each phase (in principle) is one or more documents that are approved
- following phase should not start until the previous phase has finished
- how to handle new information emerges in a process stage?
  - modification of documents in previous stage
  - changes approval by the customer

Waterfall Model (4)

When the Waterfall model is beneficial

- Embedded systems
  - inflexibility of hardware does not allow decisions on the software's functionality to be delayed
- Critical systems
  - specification and design documents must be complete for extensive safety and security analysis
- Large software systems developed by several partner companies
  - easier to follow a common model for (hardware and) software
  - complete specifications to allow for the independent development of different subsystems

Incremental development (1)

Main idea

- developing an initial implementation
- getting feedback from users and others
- evolving the software through several versions

$\Rightarrow$ Specification, development, and validation are **interleaved**

Incremental development (2)

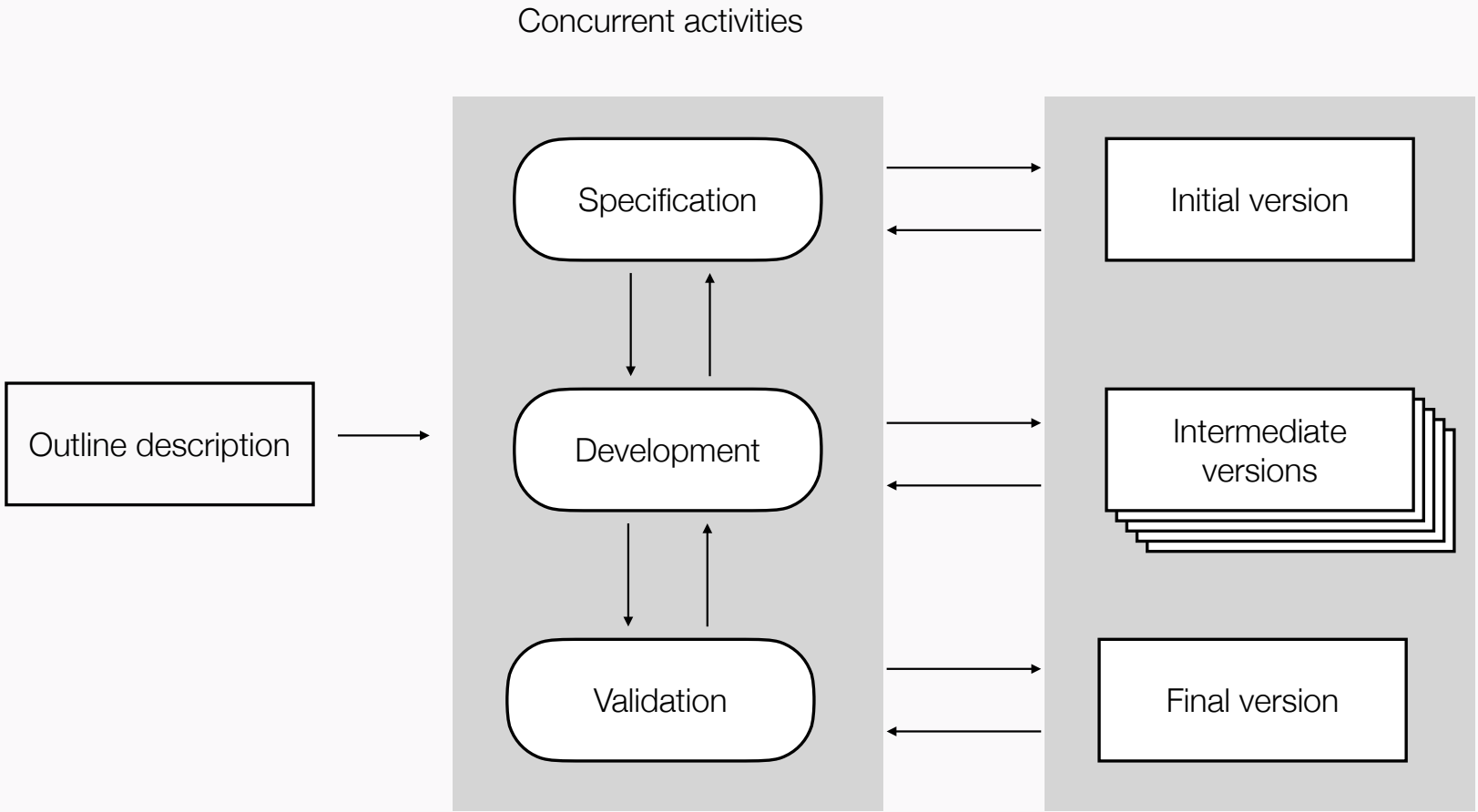


Figure from [Sommerville, 2016]

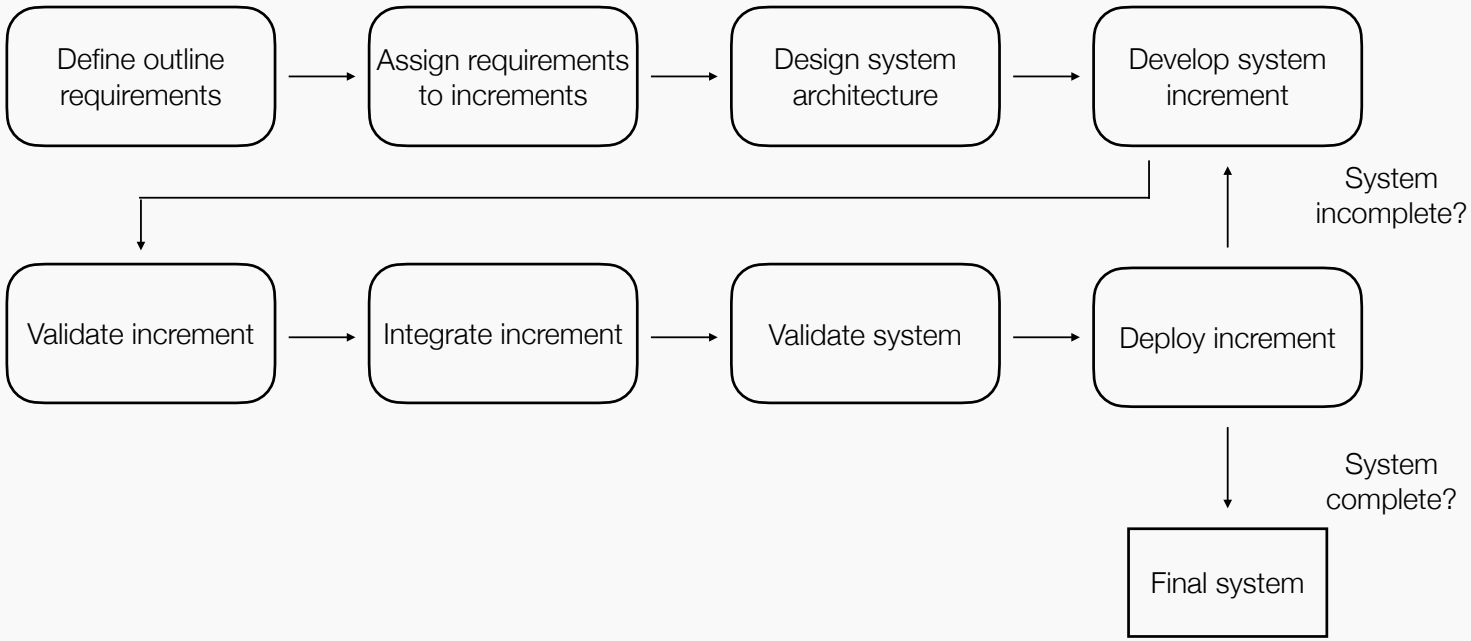
Incremental development (3)

- Advantages
  - reduced cost (VS waterfall) of implementing requirement changes
  - easier to get customer feedback on the development work (e.g., through software demos)
  - early delivery and deployment of useful software
- Disadvantages
  - process is not visible
  - system structure tends to degrade as new increments are added

Note: increment ! = release to the customer
(not necessarily incremental delivery)

Incremental Delivery (1)

- one of the methods for coping with change
- some of the developed increments are delivered to the customer and deployed for use in their working environment



Incremental Delivery (2)

Advantages

- Customers can use the early increments as prototypes and gain experience for later system increments
- Increments are part of the real system
- Customers do not have to wait until the entire system is delivered before they can gain value from it
- As the highest priority services are delivered first and later increments then integrated

  $\Rightarrow$ the most important system services receive the most testing

Incremental Delivery (3)

Problems with incremental delivery

- when the new system is intended to replace an existing system, users need all of the functionality of the old system
  - users unwilling to experiment with an incomplete new system
  - often impractical to use the old and the new systems alongside
- as requirements are not defined in detail until an increment is to be implemented, it can be hard to identify, if any,
  - set of basic facilities that are used by different parts of the system
  - are needed by all increments
- new form of contract, not based on the complete system specification

Integration and configuration (1)

- In software projects, often, some software reuse
- Reuse-oriented approaches rely on
  - reusable software components
  - integrating framework for composition
- Components commonly used:
  - stand-alone application systems configured for use in a particular environment
  - collections of objects developed as a component /package to be integrated with a component framework, e.g., Java Spring
  - Web services available for remote invocation

Integration and configuration (2)

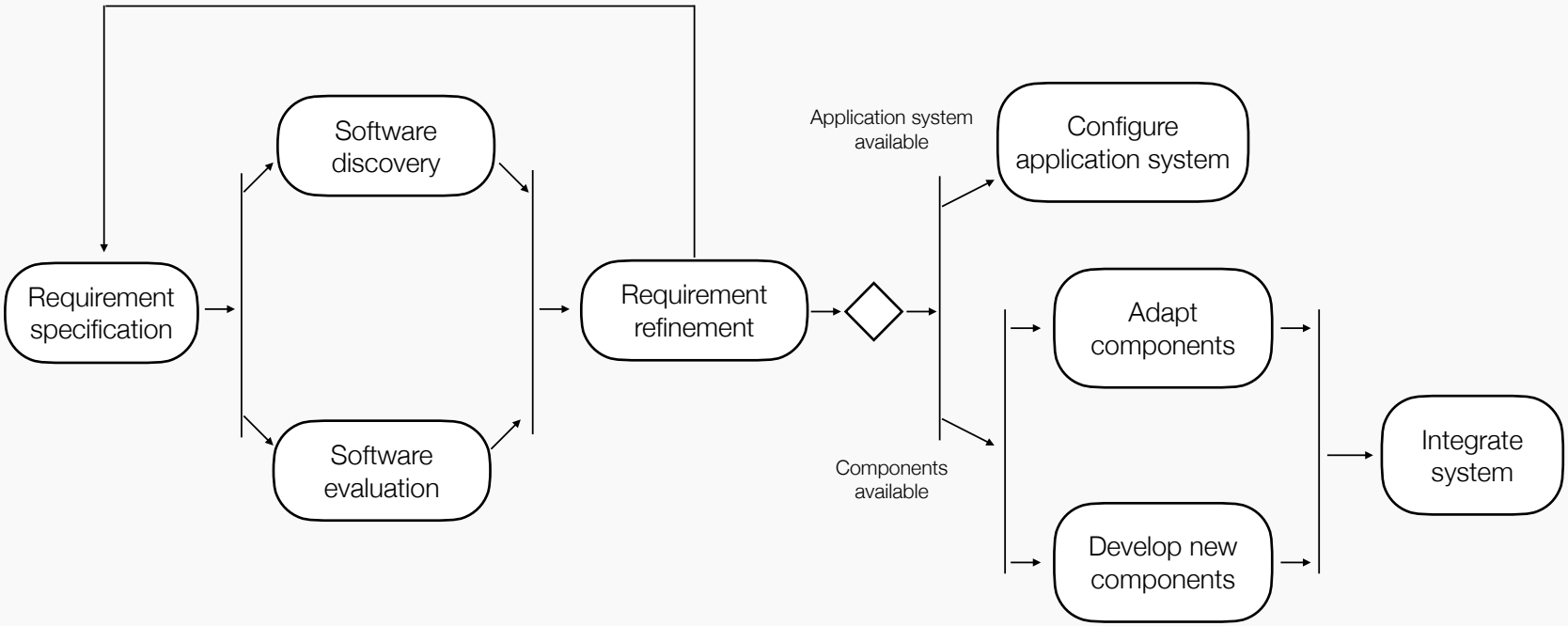


Figure from [Sommerville, 2016]

Integration and configuration (3)

Process phases:

1. **Requirements specification**:
   - proposal of the initial requirements (brief description)
2. **Software discovery and evaluation**:
   - search for components and systems that provide the functionality required
3. **Requirements refinement**
   - requirements refined using information about the discovered reusable components and applications.
   - if requirement changes are not possible, back to step 2

Integration and configuration (4)

4. **Application system configuration**
   - if off-the-shelf application system available, configuration if needed
5. **Component adaptation and integration**
   - no off-the-shelf system available
   - adaptation of the components, if possible
   - development of novel components, if needed

Integration and configuration (5)

- Advantages
  - reducing the amount of software to be developed
  - reducing cost
  - (usually) faster delivery
- Disadvantages
  - sometimes requirements compromises
  - if reusable components are not under the control of the organization, some control over the system evolution is lost (e.g., newer components)

Agile

Why Agile?

- Businesses now operate in a **global**, **rapidly changing** environment
- Software has to be developed quickly to
  - take advantage of new opportunities
  - respond to competitive pressure
- Changing environment
  - impossible to derive a **complete** set of **stable** requirements
  - By the time the software is available for use, often the original reason for its procurement may have changed radically
  - Some real requirements may become clear after a system has been delivered

$\Rightarrow$ **Agile methods**

Agile Methods: Common Characteristics (1)

- specification, design and implementation are **interleaved**
- system is developed in a series of **increments**
- extensive **tool support** is used to support the development process

Agile Methods: Common Characteristics (2)

- specification, design, and implementation are **interleaved**
  - no detailed system specification
  - minimize documentation
  - “informal” documentation
  - when possible, generated automatically by the programming environment

Agile Methods: Common Characteristics (3)

- system is developed in a series of **increments**
  - End-users and other system stakeholders involved in specifying and evaluating each increment
  - users and stakeholders may propose changes to the software and new requirements, but implemented in later versions
  - small increments
  - new system releases every 2-3 weeks

Agile Methods: Common Characteristics (4)

- extensive **tool support** is used to support the development process
  - automated testing
  - support configuration management
  - system integration
  - automate User Interface (UI) production

Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Online version here

When use agile methods?

Particularly successful for two kinds of system development

- Product development where a software company is developing a small or medium-sized product for sale
- Custom system development within an organization
  - clear commitment from the customer to become involved in the development process
  - few external stakeholders
  - few regulations that affect the software

Agile development techniques (1)

- Underlying ideas developed around the same time by a number of different people in the 1990s
- For and historical perspective: **Clean Agile** [Martin, 2019]
- One of the most significant approaches: **Extreme Programming**

Agile development techniques (2)

Extreme Programming (XP):

- developed by pushing recognized good practice, such as iterative development, to “extreme” levels
- Several new versions of a system may be developed by different programmers, integrated, and tested in a day
- short time gap between releases of the system

Agile development techniques (3)

Extreme Programming (XP):

- requirements are expressed as scenarios (called **user stories**)
- stories implemented directly as a series of **task**s
- Programmers
  - **work in pairs**
  - **develop tests** for each task before writing the code
- All tests must be successfully executed when new code is integrated into the system

Agile development techniques (4)

Practices in XP reflect the principles of the agile manifesto

- Incremental development is supported through small, frequent releases
- Customer involvement is supported through the continuous engagement of the customer in the development team
- People, not process, are supported through pair programming, collective ownership of the code, and sustainable development
- Change embraced through regular system releases to customers, test-first development, refactoring to avoid code degeneration and continuous integration of new functionality
- Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system

Agile development techniques (5)

- XP is not only a “set of techniques”
- 5 values
  - **Communication**: everyone on a team works jointly at every stage of the project
  - **Simplicity**: Developers strive to write simple code bringing more value to a product, as it saves time and effort
  - **Feedback**: Team members deliver software frequently, get feedback, and improve it according to the new requirements
  - **Courage**: Programmers objectively evaluate their own results without making excuses and are ready to respond to changes
  - **Respect**: Every person assigned to a project contributes to a common goal

Agile Project Management (1)

Self-organizing teams, little documentation, planned development in very short cycles.

Problem: can we monitor the process in Agile?

Yes! Example: **Scrum**

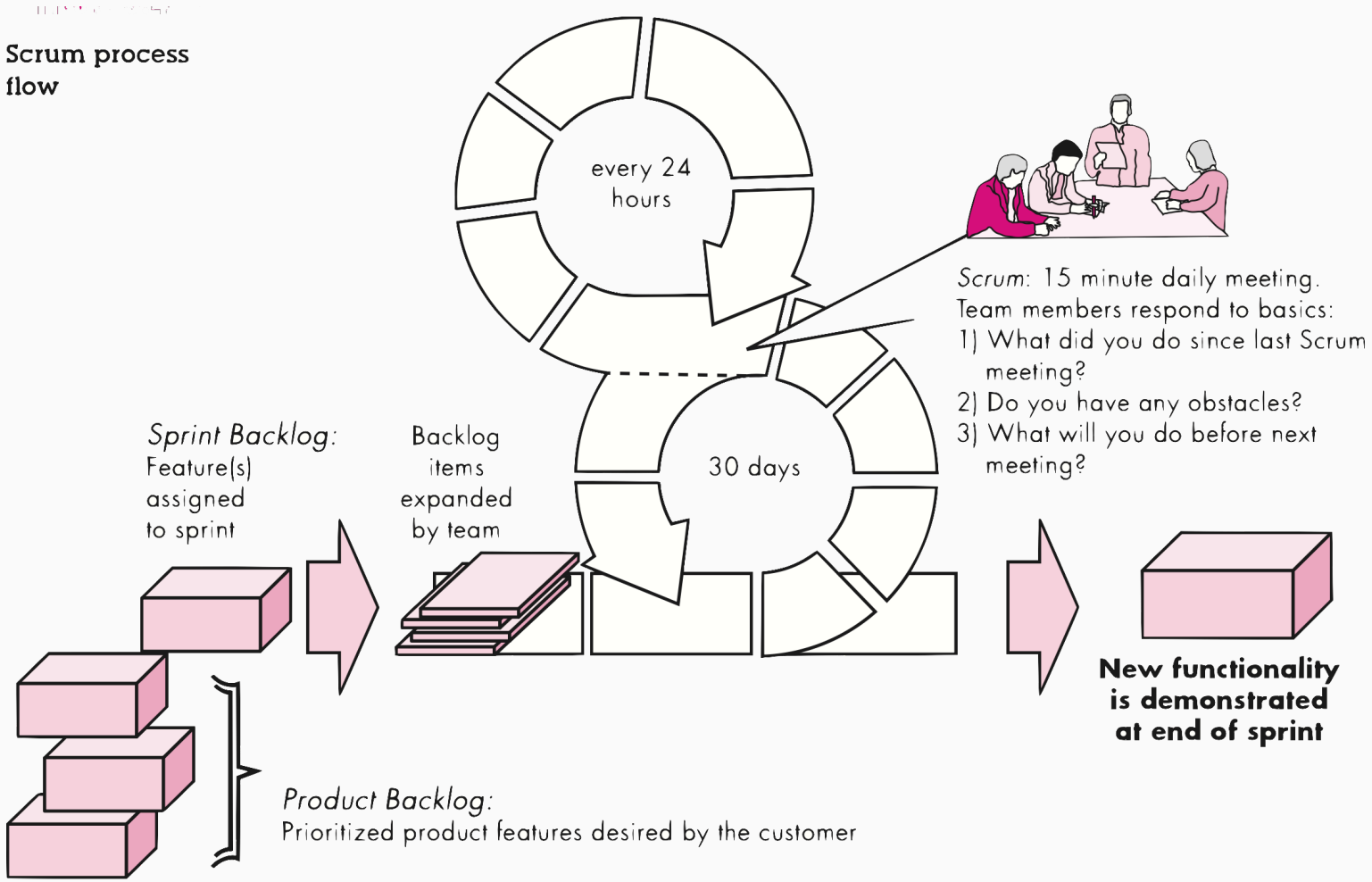
What is scrum? A **framework** to organize agile projects

Agile Project Management (2)

- Principles consistent with the Agile Manifesto
- Activities included: requirements, analysis, design, evolution, delivery
- Based on a process pattern called **Sprint**
- Number of sprints depends on the complexity and the size of the “product”
- The work in a sprint is **adaptable** and can be modified

Agile Project Management (3)

Figure from [Pressman, 2009]



Agile Project Management (4)

Main concepts/actions (from [Pressman, 2009])

- **Backlog**
- **Sprint**
- **Scrum** (meeting)
- **Demos**

Agile Project Management (5)

- **Backlog**: a prioritized list of project requirements or features that provide business value for the customer.
  - Items can be added to the backlog anytime (this is how changes are introduced)
  - The product manager assesses the backlog and updates priorities as required
- **Sprints**: work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time box (typically 30 days).
  - changes are not introduced during the sprint

Agile Project Management (6)

- **Scrum meetings**: short (typically 15 minutes) meetings held daily by the Scrum team.
  Three key questions asked and answered by all team member
  - What did you do since the last team meeting?
  - What obstacles are you encountering?
  - What do you plan to accomplish by the next team meeting?

  Meeting goal: uncover potential problems as early as possible

- **Demos**: deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer

Patterns

What is a *pattern*? (1)

- *"The regular way in which something happens or is done"* (from the *Oxford Dictionary*)
- *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution of that problem, in such away that you can use this solution a million times over, without ever doing ti the same way twice."* [Alexander et al., 1977]

What is a *pattern*? (2)

"Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience." from the The Hillside Group Website

What is a *pattern*? (3)

"Design patterns capture solutions that have developed and evolved over time [. . .] They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form." [Gamma et al., 1995]

What is a *pattern*? (4)

A pattern is a named description of a problem and a solution that can be applied to new contexts; ideally, a pattern advises us on how to apply its solution in varying circumstances and considers the forces and trade-off.

Most simply, a pattern is a named problem/solution pair that can be applied in new context, with advice on how to apply it in novel situations and discussion of its trade-offs.

[Larman, 2004]

OOA/D and RDD (1)

- OOA/D: Object-Oriented Analysis and Design
- Analysis
  - emphasizes an investigation of the problem and requirements rather than a solution
  - OOA: emphasis on finding and describing the objects or the concepts in the problem domain
- Design
  - emphasizes a conceptual solution that fulfills the requirements rather than its implementation
  - OOD: emphasis on defining objects and how they collaborate to fulfill the requirements

"Analysis and design have been summarized in the phase do the right thing (analysis), and do the thing right (design)." [Larman, 2004]

OOA/D and RDD (2)

- Many skills/guidelines to learn
- Suppose you must choose one skill. What would it be?

$\Rightarrow$ *"A critical ability in OO development is to skillfully assign responsibilities to software objects"* [Larman, 2004]

OOA/D and RDD (3)

- Main idea:
  - thinking about the design of software objects and larger scale components in terms of responsibilities, roles, and collaborations
  - think of software objects as having responsibilities (abstraction of what they do)

  $\Rightarrow$ **Responsibility-Driven Design (RDD)**

- 9 principles for RDD: GRASP Patterns

OOA/D and RDD (4)

RDD as a metaphor

- software objects as similar to people with responsibilities
- people collaborate with other people to get the work done

$\Rightarrow$ OOD as a community of collaborative responsible objects

OOA/D and RDD (5)

Often OOD results in something like:

After identifying your requirements and creating a domain model, then add methods to the software classes and define the messaging between the objects to fulfill the requirements. [Larman, 2004]

Problem: too vague! We need to decide:

- what methods belong where
- how the objects should interact

We will use GRASP, patterns of assigning responsibilities.

Case Study: Monopoly (1)

- players take turns moving their token on the board according to the result of the roll of two dice (die)
- players purchase land and develop it with houses and hotels; they collect rents from players whose pawn stops at the property
- the purpose is to bankrupt everyone else and remain the last player

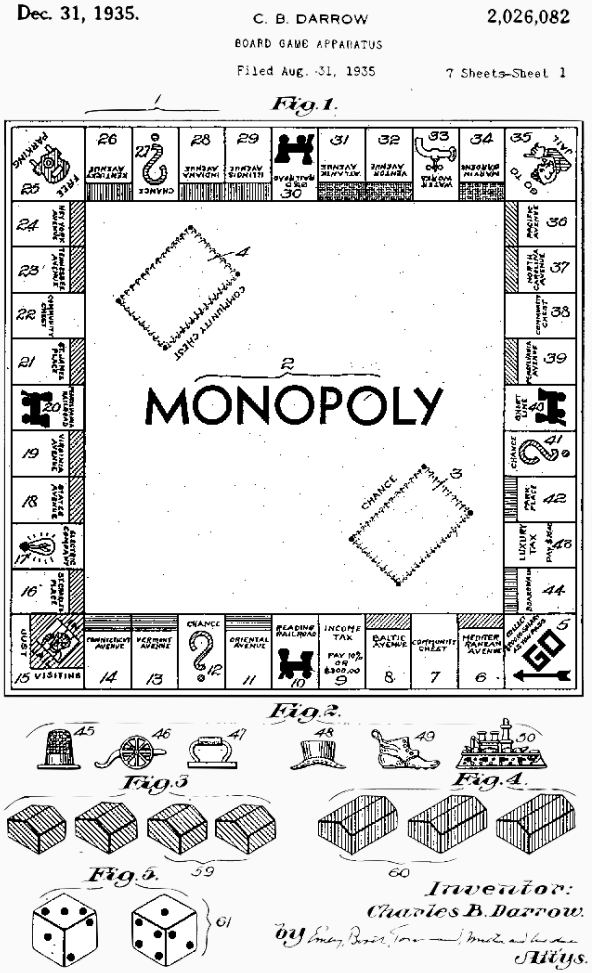


Image from https://it.wikipedia.org/wiki/Monopoly

Case Study: Monopoly (2)

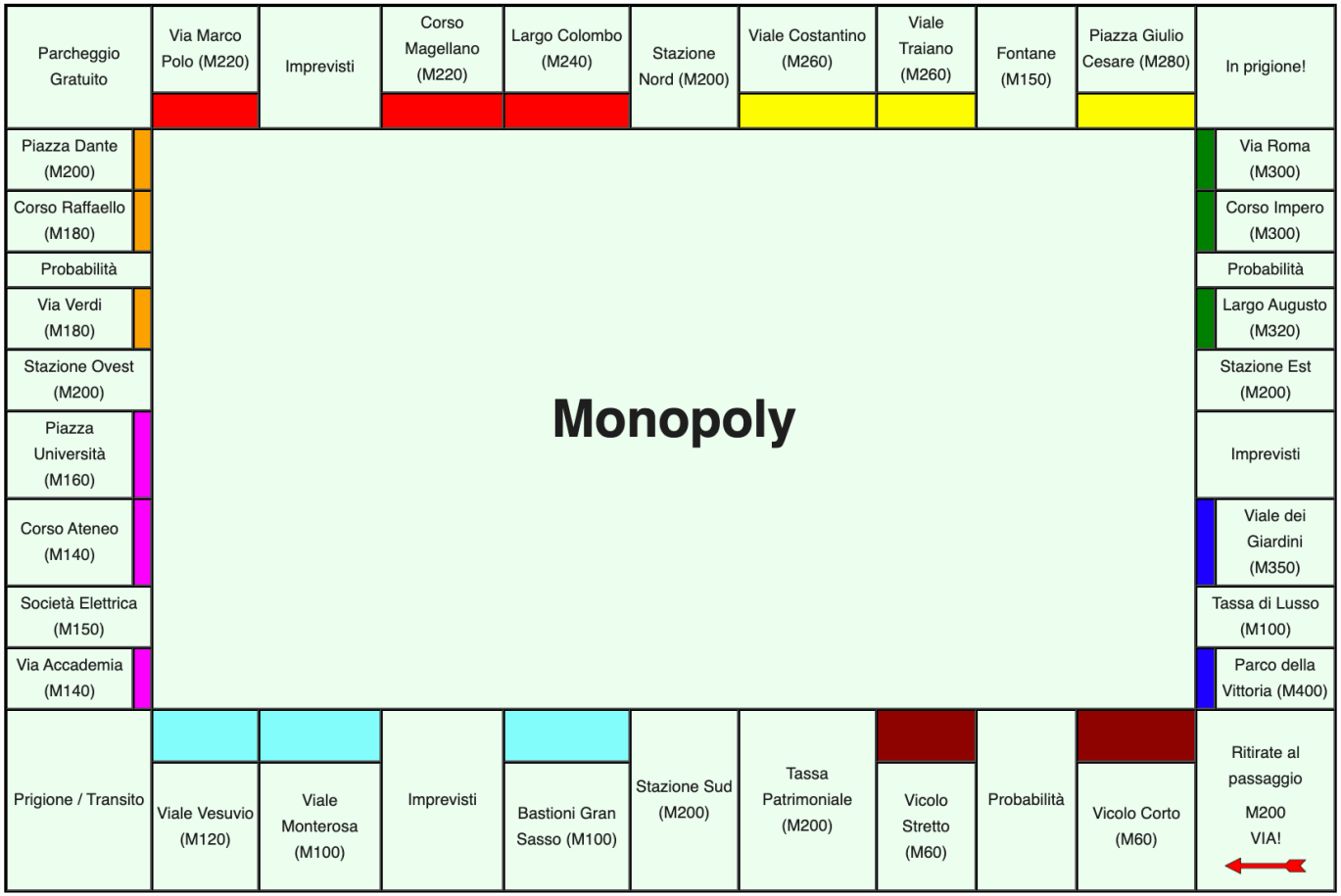


Image from https://it.wikipedia.org/wiki/Monopoly

GRASP Patterns (1)

GRASP: General Responsibility Assignment Software Patterns

- Creator
- Information Expert
- Low coupling
- Controller
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

GRASP Patterns (2)

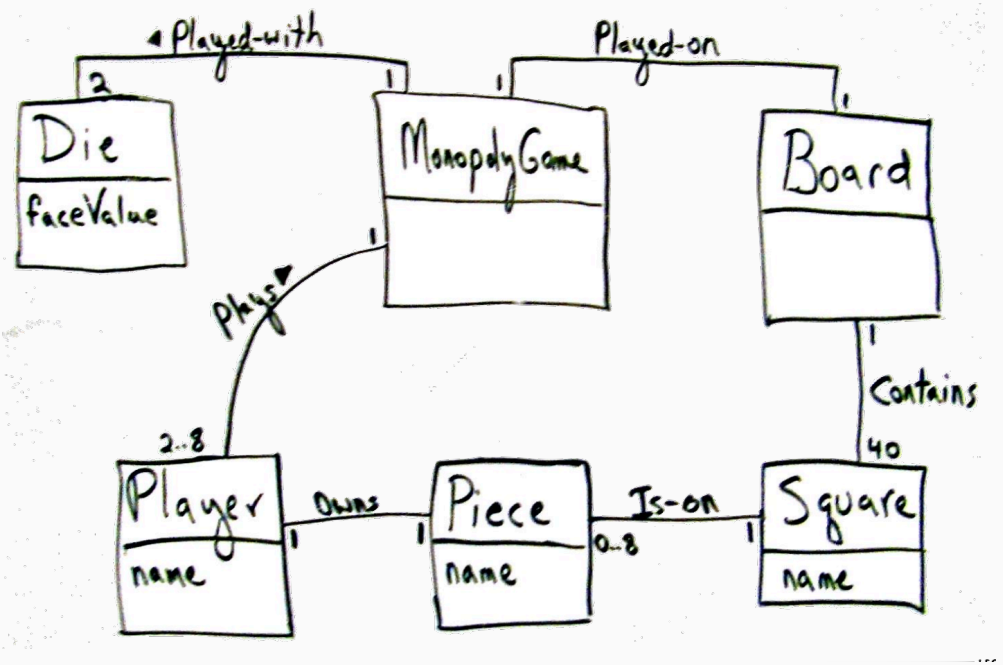
Creator Pattern

- **Nome**: Creator
- **Problem**: Who creates A?
- **Solution**: Assign class B the responsibility to create an instance of class A if one of these is true (the more, the better):
  - B “contains” or compositely aggregates A
  - B records A
  - B closely uses A
  - B has the initializing data for A

GRASP Patterns (3)

Creator Pattern - Example

- **Problem**: Who creates the *Square* object?



- **Solution**: the board (contains the squares)

GRASP Patterns (4)

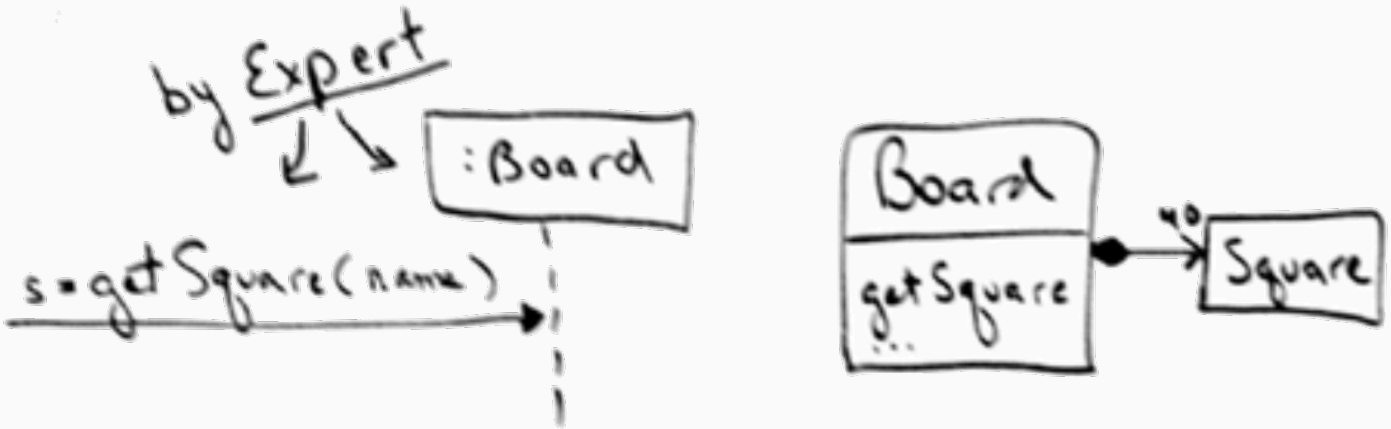
Information Expert Pattern

- **Nome**: Information Expert
- **Problem**: What is the basic principle by which to assign responsibilities to objects?
- **Solution**: Assign the responsibility to the information expert, the class with the information needed to fulfill the task/responsibility.

  "[. . .] expresses the common 'intuition' that objects do things related to the information they have." [Larman, 2004]

Information Expert Pattern - Example

- Who knows about a Square object, given a key? (identifier of the square)
- Since the Board is the object that aggregates all the Square objects, the Board has the information necessary to fulfill this responsibility



GRASP Patterns (6)

Low Coupling Pattern

- **Nome**: Low Coupling
- **Problem**: How to reduce the impact of change?
- **Solution**: Assign the responsibilities so that (unnecessary) *coupling* remains low. Use this principle to evaluate alternatives.

  What is coupling?

  Coupling is a measure of how strongly one element is connected to, has knowledge of, or depends on other elements.

GRASP Patterns (7)

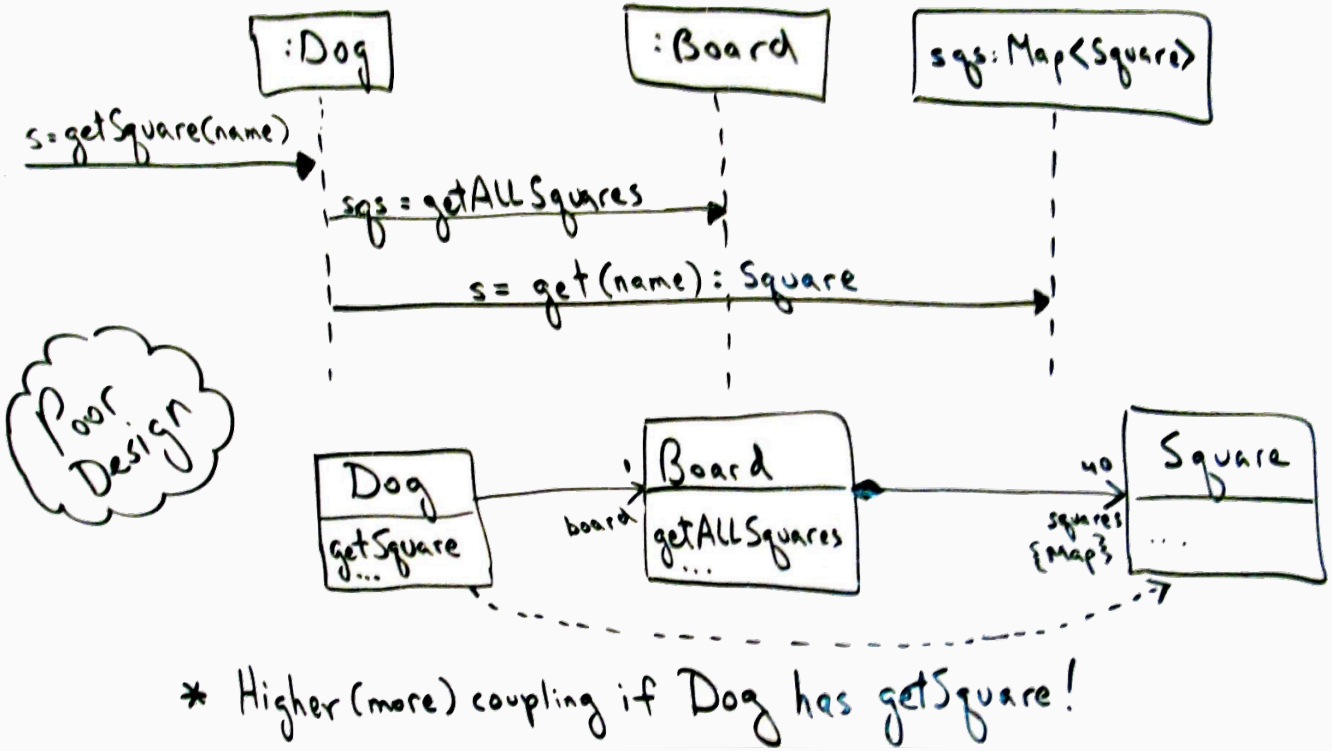
Problems with **high coupling**:

- forced local changes because of changes in related classes
- harder to understand in isolation
- harder to reuse because it requires other classes

Low Coupling Pattern - Example

Give the responsibility to know a particular square by name.

Why Square over Dog?



GRASP Patterns (9)

Low Coupling Pattern - Example

- Following Information Expert pattern results in a choice that supports Low Coupling
- Putting responsibility anywhere else results in higher coupling
  - more information or objects must be shared from their original source
  - e.g., squares in the Map shared with Dog, away from their “home” that is the Board

GRASP Patterns (10)

High Cohesion Pattern

- **Nome**: High Cohesion
- **Problem**: How to keep objects focused, understandable, and manageable, and as a side effect support Low Coupling?
- **Solution**: Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.

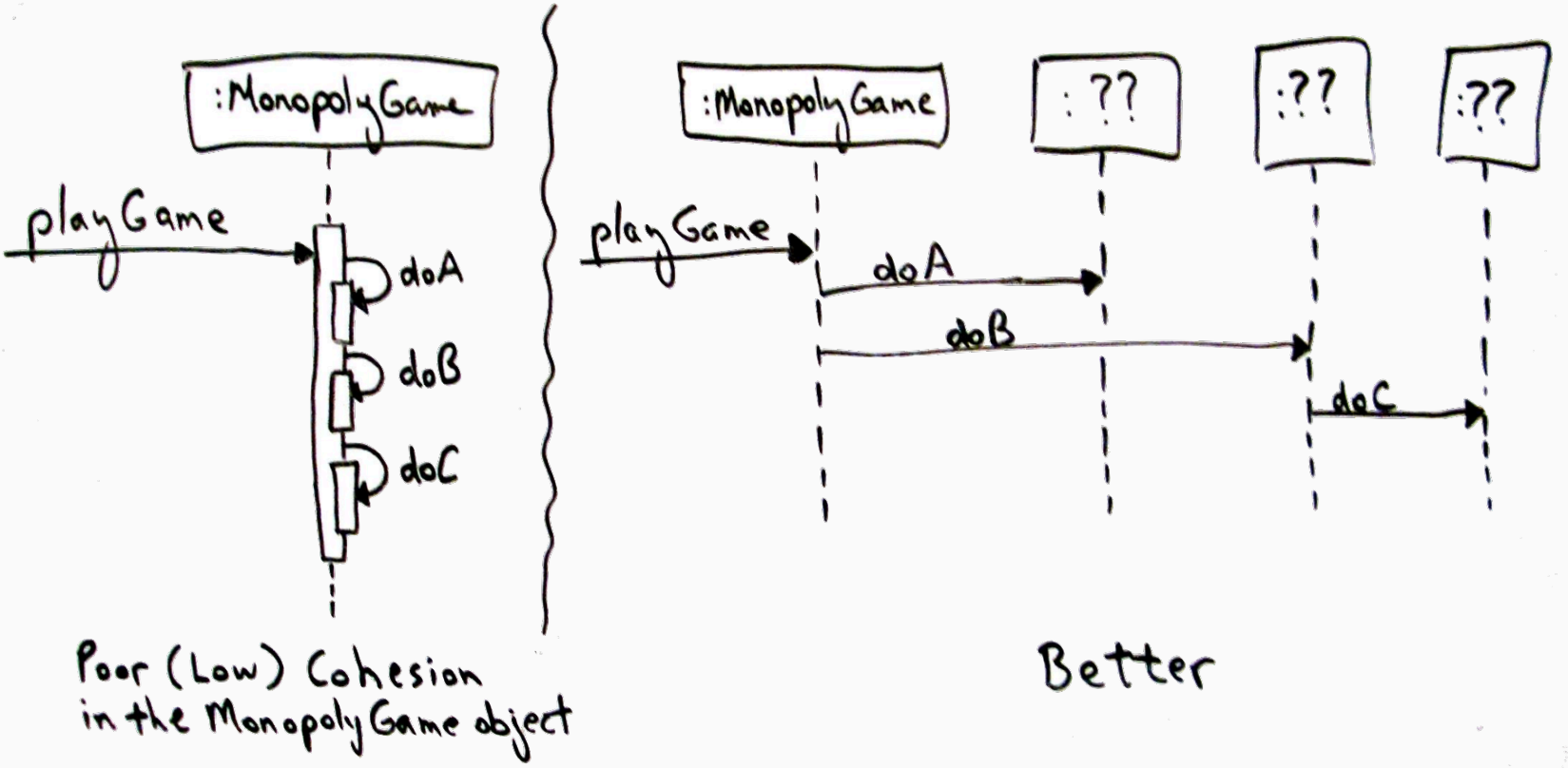
GRASP Patterns (11)

High Cohesion Pattern

- **Cohesion**: a measure of how strongly related the responsibilities of an element are
- A class with low cohesion
  - does many unrelated things
  - hard to understand, hard to reuse, hard to maintain
- A class with high cohesion
  - has a relatively small number of highly related methods
  - it collaborates with other objects

  $\Rightarrow$ Modular design

High Cohesion Pattern - Example



References

References (1)

Alexander, C., Ishikawa, S., and Silverstein, M. (1977).

A Pattern Language: Towns, Buildings, Construction.

Oxford University Press.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995).

Design Patterns: Elements of Reusable Object-Oriented Software.

Addison-Wesley Longman Publishing Co., Inc., USA.

Larman, C. (2004).

Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).

Prentice Hall PTR, USA.

References (2)


Martin, R. C. (2019).

Clean Agile.

Robert C. Martin Series. Pearson Education.


Pressman, R. (2009).

Software Engineering: A Practitioner's Approach.

McGraw-Hill, Inc., USA, 7 edition.


Richardson, C. (2019).

Microservices Patterns: With examples in Java.

Manning.


Sommerville, I. (2016).

Software Engineering.

Pearson, 10th - global edition edition.

Questions?