LECTURE 5

20.05.21

OUTLINE

1) Architectural style
   1.1) Model view controller
   1.2) Repository
   1.3) Graphs with components and connectors
2) Concurrency
3) Hardware software mapping
4) Global resource handling
5) Software control
6) Boundary conditions

1.1) Model view controller (MVC) architectural style

Problem: In systems with high coupling → any change to boundary objects (user interface) often forces changes to entity objects (data)

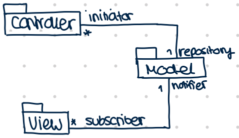
→ Hard to re-implement user interface
→ Hard to reorganize entity objects

Solution: **Decoupling:** model view controler decouples data access (entity objects) and data (boundary objects)

✓ **MODEL** process and store application domain data (entity objects)
✓ **VIEW** display information to user (boundary objects)
✓ **CONTROLLER** interact with user and update model

comprise the user interface

MVC vs. 3 tier architectural style



- MVC = nonhierarchical (triangular)
  - view sends updates to controller
  - controller updates model
  - view updated from model
- 3 tier = hierarchical (linear)
  - presentation layer never communicates directly with data layer (opaque)
  - all communication must pass through the middleware layer

Instantiating the MVC

- Software architecture instance of an architectural style
- Many possibilities to instantiate MVC
- 2 choices for notification

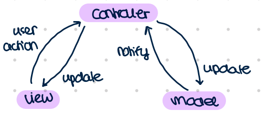
① PULL notification variant

=> view and controller obtain data from the model

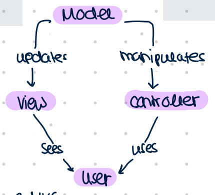
② PUSH notification variant

=> model sends the changed state to view and controller

Common instances of MVC

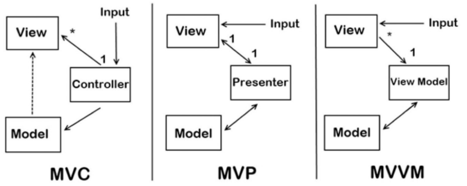


- MVC in ios
- CocoaTouch decouples view & model
- Controller as mediator



- MVC
- similar to pull variant but without updating the controller

Other variants:



MVC benefits and challenges

⊕

- multiple syncronized views on same model
- Pluggable views and controllers
- Exchangeability of look and feel
- Framework potential

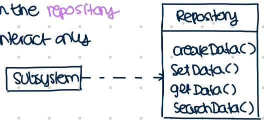
⊖

- Increased complexity
- Potential of excessive number of updates
- close connection between view & controller
- close coupling of views & controller to model

1.2) Repository architectural style

=> supports a collection of independent programs (called subsystems) that work cooperatively on a common data structure called repository

- subsystems access and modify data from the repository
- subsystems are loosely coupled (they interact only through the repo)



1.3) Graphs with components and connectors

Components (subsystems)

→ computational units with a specified interface

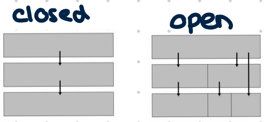
Connectors (communication)

→ Interaction between the components (subsystems)

Example Layered architectural style

components: Group of subtasks which implement an abstraction at some layer in the hierarchy

connectors: Protocols that define how the layers interact



2) Concurrency

- Two objects are inherently concurrent if they can receive events at the same time without interacting
  ↳ can be assigned to separate threads of control

Thread of controle = path through a set of states where only one object is active at any time

Thread splitting: non-blocking sending of an event to another object, so it does not wait

- concurrent threads can lead to race conditions
- Race condition = design flaw → output of process depends on a specific sequence of other events

Kostenlos heruntergeladen von Studydrive

Implement concurrency

1) Physical concurrency
→ Threads are provided by hardware (multi processors / core and networks)

2) Logical concurrency:
→ Threads are provided by software (usually provided by the operating system)

PROBLEMS: race conditions, deadlocks, starvation, fairness....

3) Hardware Software mapping

2 questions:
→ 1. Shall we realize the subsystem with hardware or software?
→ 2. How do we map the object model to hardware / software?
a) Mapping the objects: cpu, memory, IO-devices
b) Mapping the associations: network connections

a) Mapping the objects

Control objects → processor
- computation rate too demanding for one single cpu?
- improve by distributing objects across several processors?
- How many cpus to maintain steady state load?

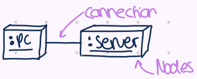
Entity objects → memory
- Enough memory?

Boundary objects → I/O-Devices
- Need of extra hardware?
- good response time? good communication?

Hardware software mapping difficulties

=> addressing externally-imposes hardware and software constraints
- certain tasks have to be performed at specific locations (ATM)
- Some HW-components have to be used from a specific manufacturer

DEPLOYMENT DIAGRAM = useful for showing design after system design decisions have been made
-> subsystem decomposition
-> concurrency
-> Hardware Software mapping



Graph of nodes & connections („communication associations")
-> can be connected by lollipops and sockets

UML component vs. deployment diagram

- Illustrates dependencies between components at design time, compile and run time
- uses components and connectors (interfaces) to show the dependencies

- Illustrates the distribution of components on concurrent processes at run time
- uses nodes and connections to depict the physical resources in the system

b) Mapping the associations

-> describe the physical connectivity
-> describe the logical connectivity (subsystem association)
-> Informal connectivity drawings often contain both types of connectivity

4) Persistent data management

→ All objects of type entity in the system model need to be persistent

✓ Persistency: values have a lifetime beyond a single execution of the system

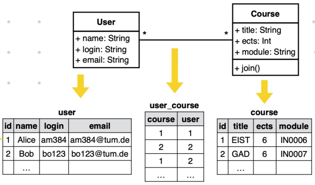
persistent class can be realized with following mechanisms
✓ FILE SYSTEM: data is used by multiple readers but a single writer
✓ DATABASE SYSTEM: data used by concurrent writers and readers

Mapping an object model to a database

- UML object models can be mapped to relational database
- Object relational mapping (ORM)

Example of ORM

- class – table
- attribute – column
- instance – row
→ methods are not mapped



user

id	name	login	email
1	Alice	am384	am384@tum.de
2	Bob	bo123	bo123@tum.de
	...		

user_course

course	user
1	1
2	2
1	2
...	...

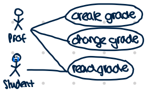
course

id	title	ects	module
1	EIST	6	IN0006
2	GAD	6	IN0007
	...		

5) Global resource handling

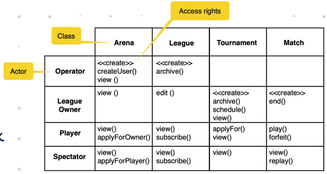
defining access control

→ different user usually have different access to functions and data
→ How do we model access rights?
=> During analysis: by associating with use cases
=> During system design: Determination which objects are shared



ACCESS MATRIX:

=> models access of actors on classes
→ rows – actors
→ columns – classes
→ Access right: entry in matrix


Access rights
Class
Actor

	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		

League Owner	view ()	edit ()	<<create>>
			archive()
			schedule()
			view()
				<<create>>
				end()

Player	view()
	applyForOwner()
		view()
		subscribe()
			applyFor()
			view()
				play()
				forfeit()

Spectator	view()
	applyForPlayer()
		view()
		subscribe()
			view()	view()
				replay()

3 different implementations:

1) Global access table
2) Access control list
3) Capabilities

1) Global access table

=> represents every non empty cell in the matrix as a triple

(actor, class, operation)

2) Access control list

=> associates list of pairs (actor, operation) with class being accessed

- every time an instance is accessed, the access list is checked for the corresponding actor and operation

3) Capability

=> associates a pair (class, operation) with an actor

- allows an actor to gain control access to an object of the class by calling one of the class operations described in the capability

Kostenlos heruntergeladen von Studydrive

6) Software control

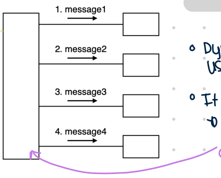
2 system design choices
- 1) Implicit software control
  - Rule based system
  - Logic programming
- 2) Explicit software control
  - centralized control
  - Decentralized control

Communication Diagrams:

→ can determine the decentralization of a system
→ The structure of the com. diagram helps us to determine how decentralized the system is

- FORK DIAGRAM
- STAIR DIAGRAM

FORK DIAGRAM

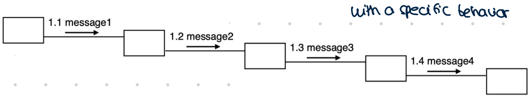


- Dynamic behavior is placed in a single object, usually control object
- It knows all the other objects
  → direct questions and demands

control object

STAIR DIAGRAM

- Dynamic behavior is distributed
- Each object delegates responsibilities to other objects
- Each object knows only a few other objects and knows which objects can help with a specific behavior



EXPLICIT SOFTWARE CONTROL: CENTRALIZED vs. DECENTRALIZED

centralized control

- Procedure-driven:
  → control resides within the program code
- Event-driven
  → control resides within a dispatcher calling the other functions via so called callbacks

⇒ One control object or subsystem („spider") controls everything

PRO: easy change in control structure

CON: single control object: possible performance bottleneck

Decentralized control

- control resides in several dependent objects

→ promises a possible speedup by mapping the objects on different processors, but requires increased communication overhead

⇒ Not 1 single object is in control, control distributed, more than 1 control object

PRO: Fits nicely into object oriented development

CON: Additional communication overhead

7) Boundary condition

- Initialization: system is brought from a non-initialized state to steady state
- Termination: resources are cleaned up and other systems are notified upon termination
- Failure: bugs, errors, external problems

⟹ good system design foresees fatal failures and provides mechanisms to deal with them

MODELING BOUNDARY CONDITIONS

→ best modeled as use cases
→ „boundary use cases" or „administrative use cases"
→ ACTOR: system administrator
→ INTERESTING: start up of subsystem / full system, Termination, Error handling

Ty for studying with me~ ♡
good luck

Kostenlos heruntergeladen von Studydrive