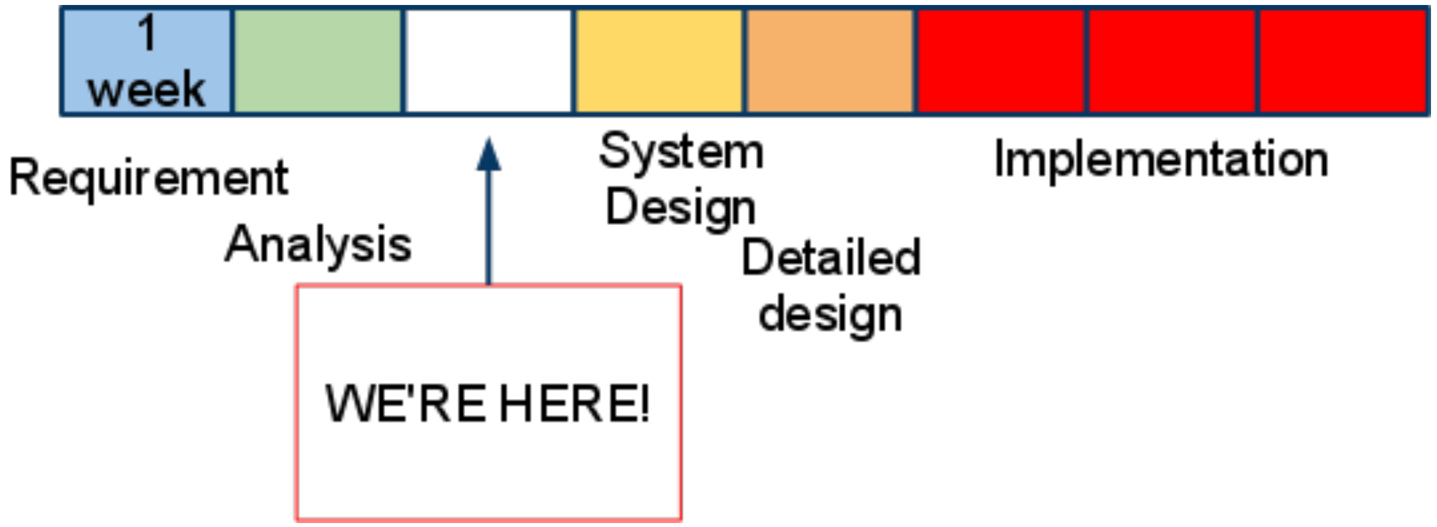
# System Design

Phase 3

# So far...

- Requirement elicitation, **done**
  - Have the use cases, preliminary functional/non-functional requirements, GUI
- Analysis, **done**
  - Have the preliminary analysis model
- First running increment, under way...



## System design

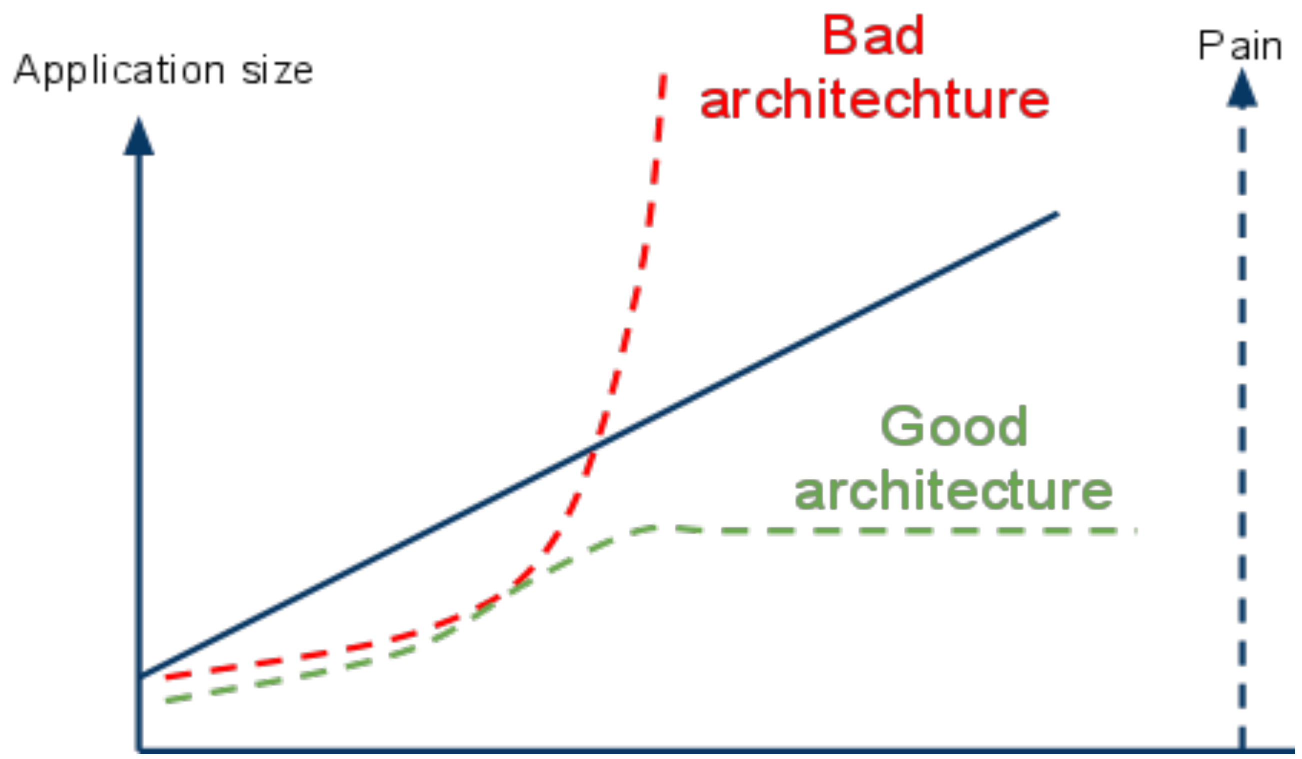
- During system design we try to create the overall structure
  - Partitioning (divide) the problem and the system to be built into discrete pieces
  - Create interfaces between these pieces
  - Manage overall structure and flow
  - Interface the system to its environment
  - What we get is the **system architecture** (general: Software architecture)
- This is **not** a well understood topic

*The distinction between design and architecture is blurred. In this course architecture is higher level than design*

## Software architecture

- The software architecture defines the non-functional requirements and the environment of the system
  - During the next phase, "detailed design", we define how to deliver the functional behavior **within the architectural rules**
- Architecture is important because it;
  - Controls complexity
  - Enforces best practices
  - Gives consistency and uniformity
  - Increases predictability
  - Enables re-use.

## The Impact of Architecture



# Points of Variation

- Try to identify what can (will probably) change
- Prepare for change but don't "over-engineer"
- Possible points
  - GUI
  - Rules (Business, Games?)
  - Input/Output
  - Data formats
  - ...

# Top-down or Bottom-up?

- Are we working top-down or bottom-up?
  - Analysis model is bottom-up (i.e starting with the small pieces, the details...)
- But shouldn't we think top down?
  - Starting with the over all picture (high abstraction)
- **Answer:** OOAD (Object oriented analysis and design) is both
  - From now (and for a while) we think top-down but...
  - ...have to move up/down between abstraction levels
    - If stuck high-level, imagine how you would like the code to look
    - If stuck low-level, imagine the overall model of this

## Some System Design Principles

- There are many, a few...
  - Keep it small and simple (KISS)
  - Everything has one well defined responsibility (Single responsibility principle)
  - Minimize side effects (low coupling)
  - Minimize dependencies (low coupling)
    - Subsystems as independent as possible
    - No circular dependencies
  - Keep complexity inside a module (high cohesion)
  - Keep high rates of information exchange inside a module (high cohesion)
  - State exists inside modules (low coupling)
  - Use open standards
- ...typically easy to state, hard to achieve

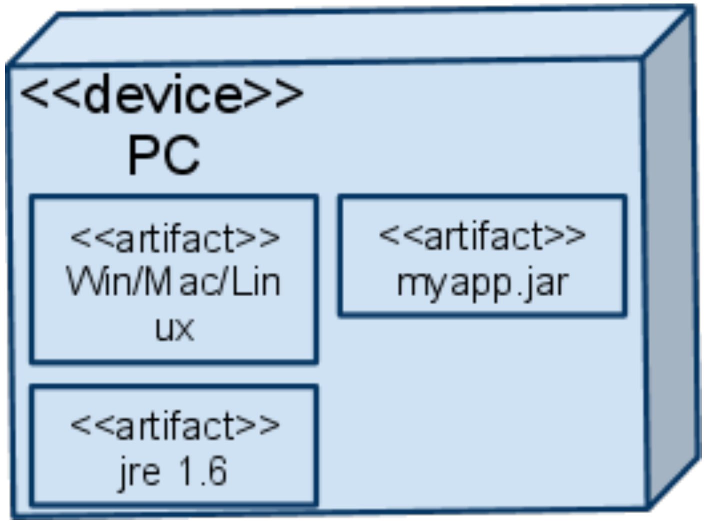
# Documenting System Design

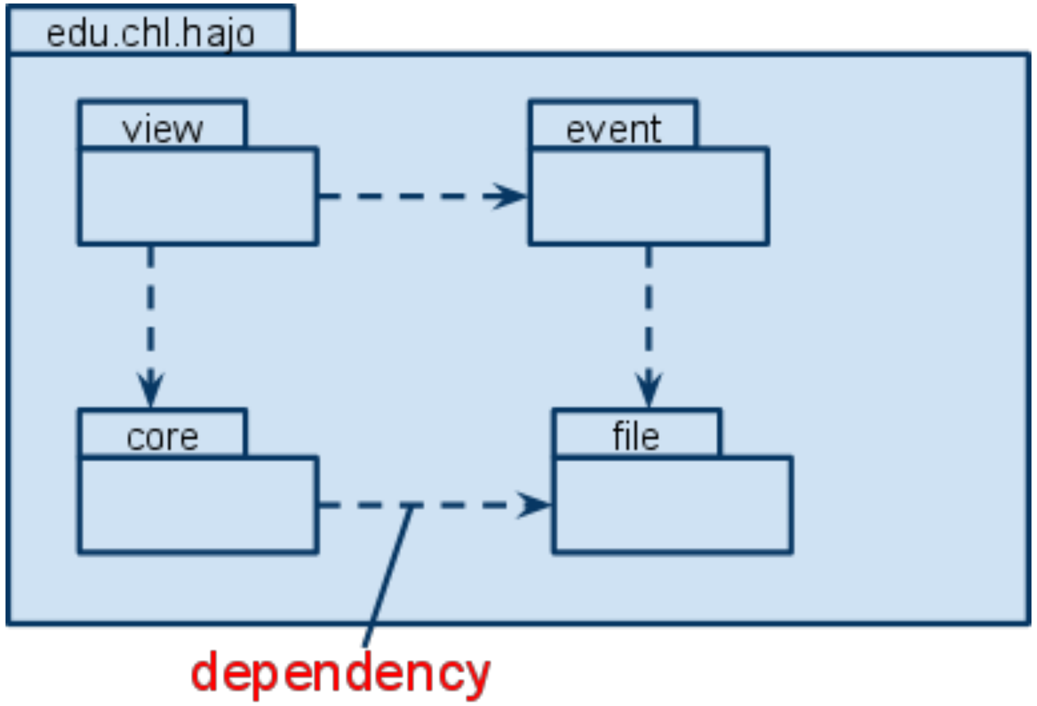
- The System Design Document (SDD)
    - SDD Template on course page
    - Again: Not everything is applicable,
        - If not put a "NA" in the section

# UML for System Design

- Deployment diagram (left)
- Package diagram (right)
- Also: Class diagrams

myapp.jar





## System Design Overview

- Design goals
- Global design decisions
- Software decomposition (the pieces)
  - Tiers, subsystems, interfaces
- Layering
- Communication
- Dependency analysis
- Persistency, storing data, data formats
- Concurrency issues
- Security
- Boundary conditions; Start, stop, errors
- Selecting platform, **done**, (Java SE 1.6)

## Design Goals

- Input from RAD (non-functional)
- Remainder:
    - Reliability, NA (not applicable)
    - Fault tolerant, possible...
    - Security, possible "roles", ...
    - Modifiability, to some extent, costs time..
    - Performance, probably NA
    - Portability, NA
    - Usability, probably ...
    - Testability (high level, yes, low level later, yes)
    - ...

# Global design decisions

- Decisions affecting "everything"
    - Distributed application (optional)
    - Globally unique id's
    - Global data structures (accessible globally)
    - MVC model, **done**
    - Life cycles of objects
    - Interoperability requirements
    - Communication (also inside single application)
    - ...

# Design Goals and Global Decisions for MoPro

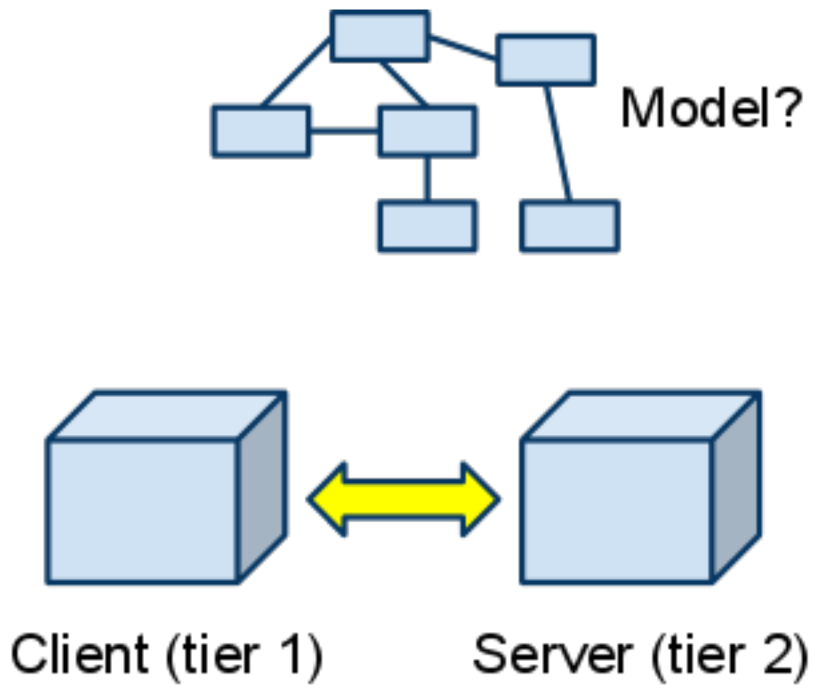
- Design goals
  - Inspect SDD section 1.1
- Some global decisions
  - Inspect monopoly-3.2.ep/doc
  - Global design decision: Spaces
    - Rationale: It's a static setup defined at program start. Easy to handle with all spaces in a single list (can build Spaces and GUI from datafile)

# Decomposition

- Finding the pieces
  - Distributed applications (optional)
  - MVC
  - Analysis model
  - Subsystems
- Interfaces!.. to be continued...

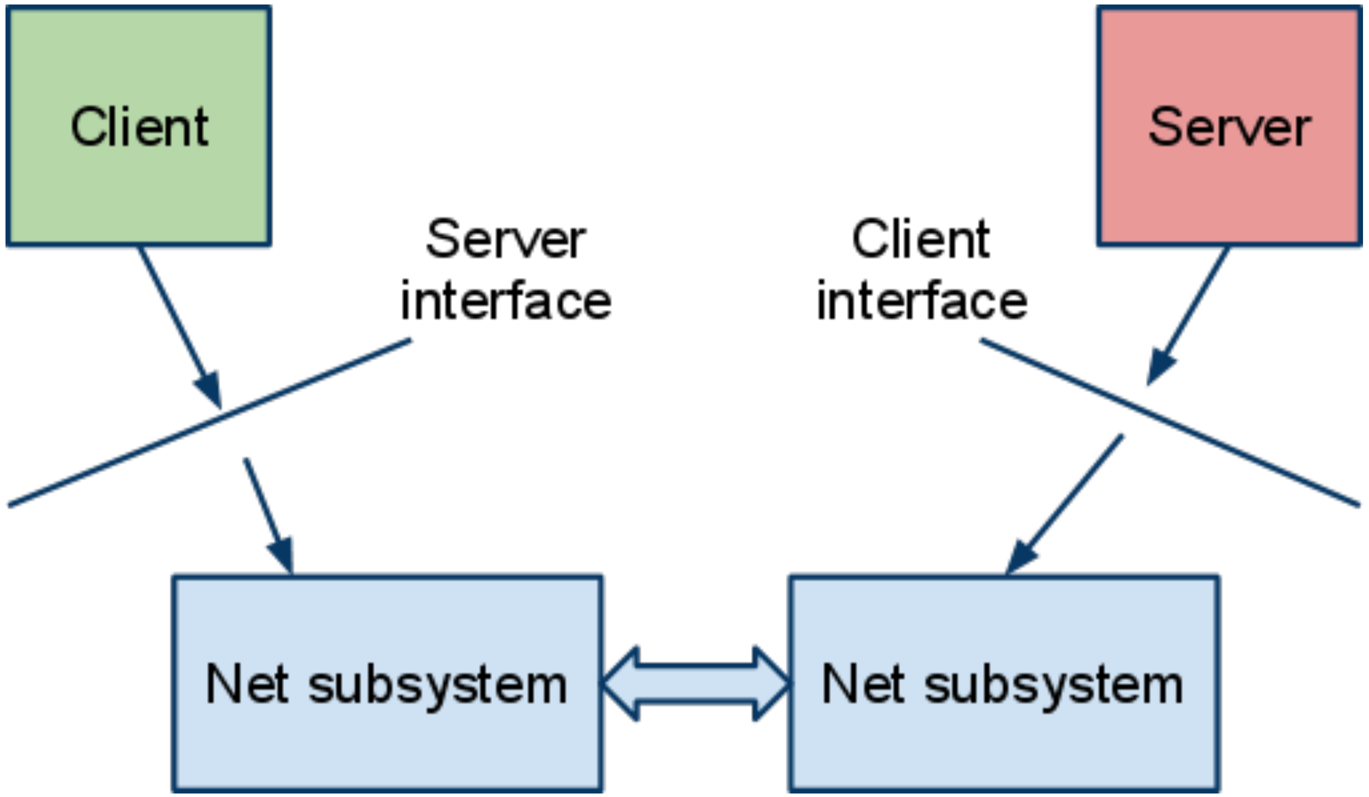
# Distributed applications (1) (Optional)

- Partitioning Into Tiers (separate applications)
  - Typically client/server
- Parts of analysis model probably shared
  - Where to put it?



# Distributed applications (2) (Optional)

- Interfaces
  - The important design decisions! What would we like to do?
  - Exact implementation is a **detail**, keep it open...



# MVC

- Have the pieces
  - Packages in Java
- The interfaces
  - View implements some observer interface
  - Controls often use a common simple interface, more later...
  - Model...?? ..next slide...

# Interface(s) to Analysis Model

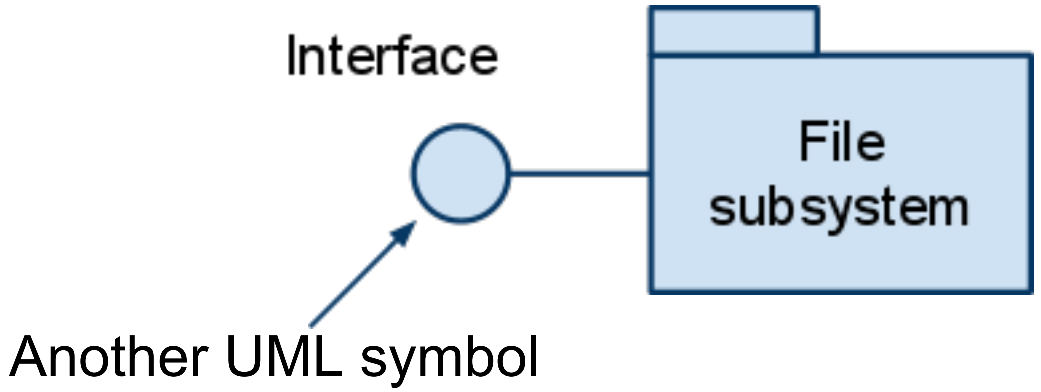
- Anemic model
  - Analysis classes act as data, used as parameters and/or return values, normally no interfaces for model object
  - Controls uses simple interface(s) to retrieve model objects and possible act on
- Fat model
  - Model objects with much functionality
  - Expose functionality through interfaces
  - Keep as much as possible inside model, just expose what's needed!
- Again: Possible use a mixture

## MoPro Interfaces to Analysis Model

- Comparing monopoly-3.2.ep (anemic) and monopoly-3.2-DDD.ep (fat)
- Anemic
    - One simple interface IGame
- Fat
    - Objects exposing functionality to GUI have interfaces, IGame, IPlayer, IRules (other interfaces for technical reasons used by Visitor-pattern, possible more later...)
    - The data-parts (the pure data) of the model objects as "value objects"
        - Sent from/to GUI

# Partition Into Subsystems

- Which are the high level subsystems?
  - Vague name code smell => low cohesion

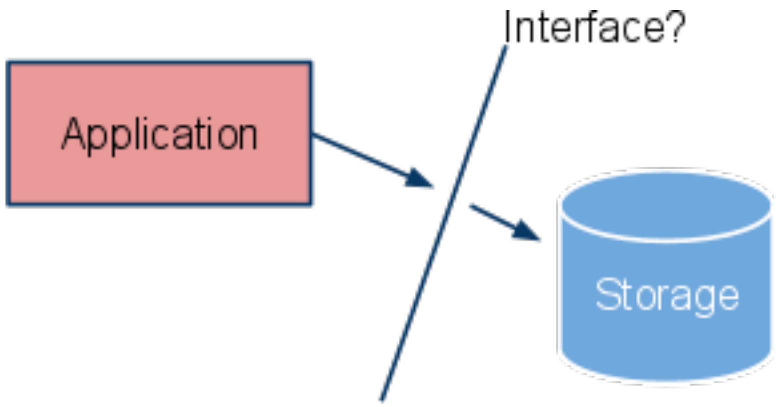


## Some Typical Subsystem

- Persistency
- Printing
- Communication
- Rule systems (business/game rules)
- Engines, simulation engine
- Processors (text  formatter, spell checker)
- Security, authorization module
- Mappers, mapping between formats

## Subsystem Interfaces

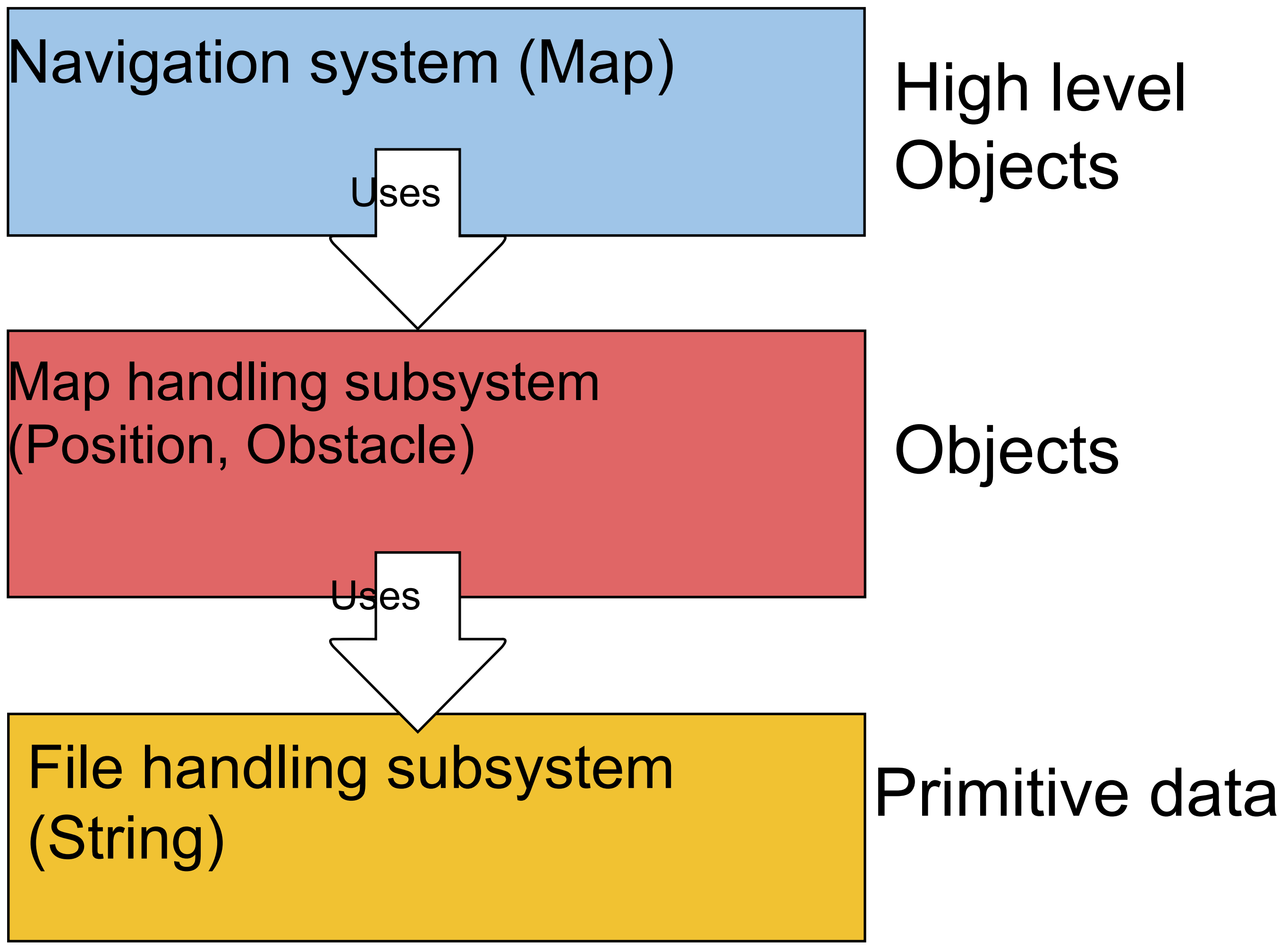
- Once again: The interfaces are the important design decisions, exact implementation is a **detail**!.
- Example: IPersistency
    - Interface to storage system
        1. What would you like to do?
        2. Implementation: Flat files, serialization, XML, real database



## Layering

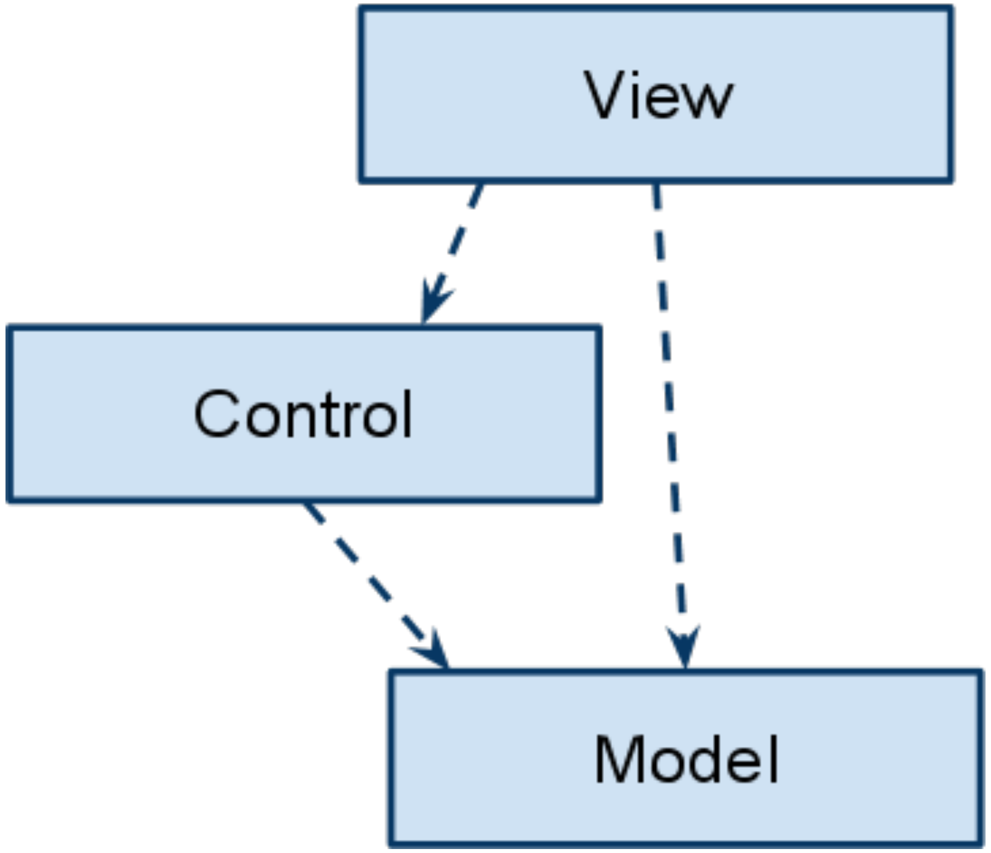
- High level abstraction subsystem uses low level
  - Dependencies going towards lower level
  - Lowest abstraction layer often uses primitive data
- Layers calls each other via interfaces
- Layering inside subsystem possible

# Layering Example


Navigation system (Map)
Uses
High level Objects
Map handling subsystem (Position, Obstacle)
Uses
Objects
File handling subsystem (String)
Primitive data

## Layering and MVC

- Dependencies!



## Communication

- Between applications (client/server)
- Inside application
- Synchronous
  - Blocking method call, like a telephone call
- Asynchronous
  - Method starts thread, and returns (aka messaging, like sending a letter)

# Communication in MoPro

- State Changes
  - The state of the model has changed
  - Example: A player buys a street, set owner for street => model state changed
- Events
  - Something happens
  - Example: Player gets equal dices. No state change, an "event" the game must handle
- Example : The EventBus
  - Rationale: Very many state changes/events, observers connected in many ways also dynamic. Centralize it.
  - See monopoly-3.2.ep/doc, event.EventBus, class Player and GUIBuilder for use

# Dependency Analysis

- Low coupling is a central quality aspect of software
- Must keep dependencies under control
    - At system design: Inspect UML or other to get an initial view
    - Later: Use tools continuously (JDepend, others...) to check.
        - If needed re-factor.

# Persistency And Concurrency

- Persistent data
  - Data that outlives the application execution
    - Example: Highscore list
    - Possible file format?
    - Possible have to translate between objects and other format
- Concurrency
  - Distributed applications inherently concurrent
  - Should the subsystem be thread safe?
  - Note: ConcurrentHashMap in MoPro (EventBus)

# Security And Boundary Conditions

- Security
  - Should user login?
  - Roles?
- Boundary conditions
  - How to start and stop
    - Usually a shell script or bat file
    - Stop, what will happen...(probably trivial)
  - Exception handling

## Leftovers

- Some classes will not fit in a subsystems
  - Helper classes
  - Possible utils-package

# System Design for MoPro

- A final look at the SDD
- And a quick demorun of monopoly-2.0.ep
  - Buy/Sell property use cases
  - EventBus
  - More GUI, disable/enable

## Hmm...

Our Disaster Recovery Plan Goes Something Like This...
HELP! HELP!