



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Object-oriented Programming Project

Analysis

Dr. Alex Gerdes

TDA367/DIT212 - HT 2018

- Representatives:
 - Alrik Kjellberg
 - Edvin Leidö
 - Pontus Lindblom
 - Carolina Larsson
 - Oskar Lyrstrand (GU)
 - David Weber Fors (GU)
- Contact details on course website
- Meeting after lecture (?)

- User Stories
 - Describe the requirements and the acceptance criteria
 - Can also hold information about the estimate and the priority
 - Should describe the value of the story
 - Can be updated continuously
 - INVEST criteria
 - In Requirements and Analysis Document (RAD)
- Backlog / sprintlog
 - Brake down in tasks
 - Vertical slices!

- Workflow:
 - Project idea: have this really clear
 - Define User Stories
 - Prioritise the User Stories
 - Break down the User Stories in tasks
 - Make rough estimation of User Stories (in person-days)
 - Make a selection for 2 to 3 weeks
 - Design (this and next lecture)
 - Implement (start with defining tests, TDD)
 - Check acceptance criteria
 - Reflect and iterate

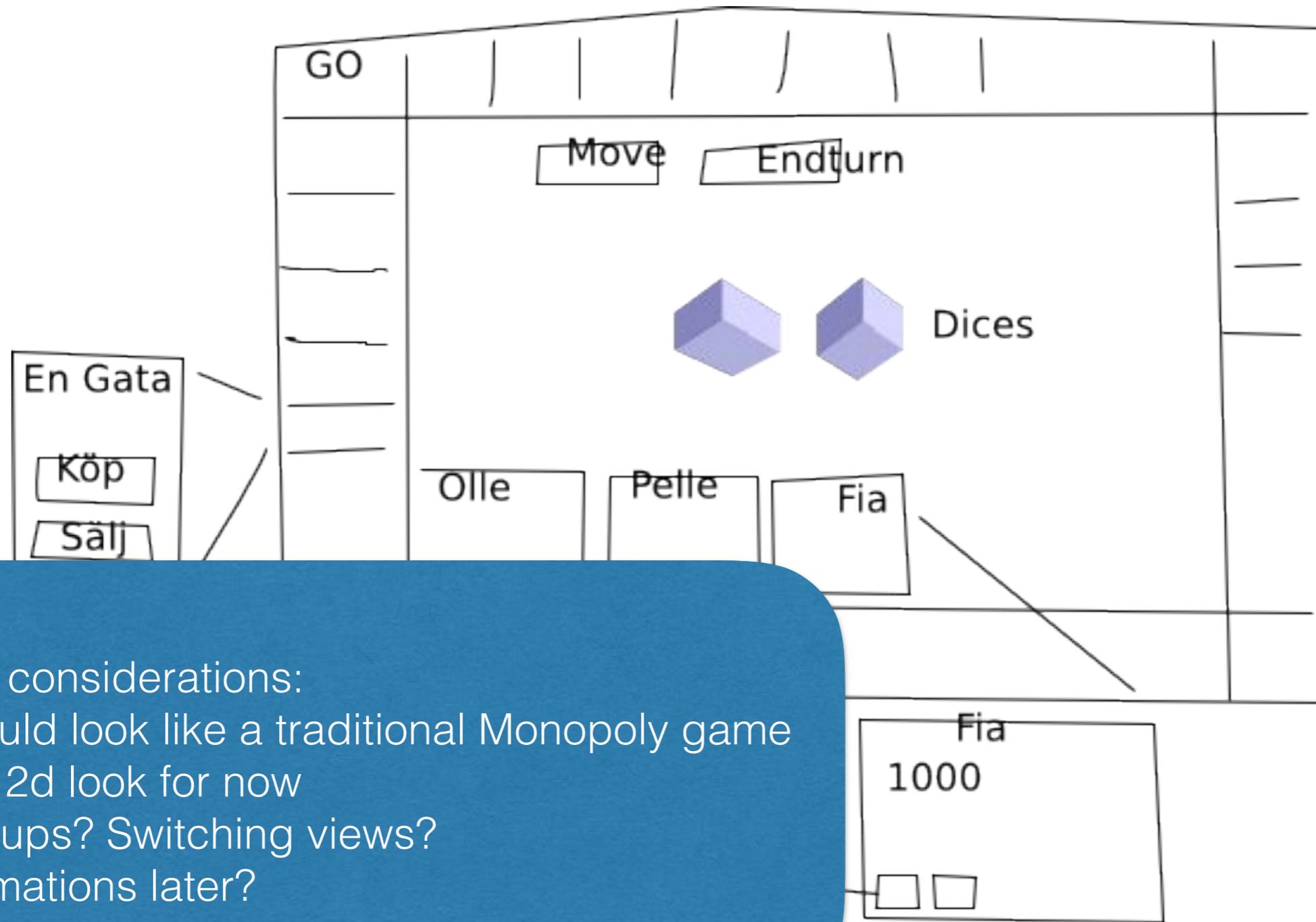
Running example: Monopoly

- *The project aims to create a computer based generic version of the well known board game Monopoly by Parker brothers. Generic in the sense that it's should be possible to adapt the game to different locations and more.*
- Some general characteristics:
 - The application will be turn based. The actual player must explicitly end his or her turn. The next player is chosen by the application from a preset ordering. The ordering is generated randomly by the application at start of the round.
 - There are no time constraints for a round.
 - The application will end according to the rules or possible be canceled.
 - If the game is canceled the player with most resources will be the winner.
 - The application will handle all of the bank's responsibilities.
 - The application will use a GUI very similar to the original game.
 - The application does not include a computer-player. It's impossible to play the game alone (a person can of course choose to play against herself).
 - The application does not save interrupted games or collect any statistics (high score or other).

Write as
introduction in
RAD

- To aid the definition of user stories we create a *preliminary* graphical user interface
 - A GUI can be referred to from a user story
 - Sketch a simple initial GUI
- A GUI sketch lets you:
 - Envision the system (important for customers)
 - Explore the problem space with your stakeholders
 - Explore the solution space of your system
 - Communicate the possible UI design(s) of your system

User interface sketch



Some considerations:

- Should look like a traditional Monopoly game
- Flat 2d look for now
- Popups? Switching views?
- Animations later?

As as: user
I want to: play the Monopoly game
so that: I can have fun

Acceptance:

- User can play game according to MP rules

As as: user
I want to: set up the game
so that: the game can commence

Acceptance:

- User can set the options
- After all options have been defined (number of players etc.), the game is ready to start

As as: player
I want to: take a turn
so that: I can try to win

Acceptance:

- User can roll dices
- User can make a move
- The state of the game is updated according to MP rules
- ...

Remember DoD

As as: user

I want to: choose the number of players
so that: I configure the game

Acceptance:

- Application can read input from user
- User can fill in number of players
- The game configuration is updated according to user input
- The game cannot start before the number of players is configured
- The number of players can not be changed after the game has started
-

And many more!

As as: player

I want to: roll the dices
so that: I can make a move

Acceptance:

- Player can start the roll of the dices
- All players can see the result
- After rolling the player can make a move
- The player can only roll the dices one time
- ...

Breakdown in tasks



As as: player

I want to: roll the dices

so that: I can make a move

Acceptance:

- Player can start the roll of the dices
- All players can see the result
- After rolling the player can make a move
- The player can only roll the dices once
- ...

Vertical slices!

- Show the board
- Show the players on the board
- Highlight the active player
- Allow the active player to roll the dices
- Show the resulting dice values
- Make the dices values available to other actions (next move tex)
- Change active player

Non-functional requirements

Portability

Ability to easily move the application to a different hardware platform, operating system or even database management system or network protocol

Personalisation

Ability to allow individual users to customise their view of the application (solution) (My Yahoo style)

Monitorability

Ability to access information on the applications behaviour

Performance

Throughput, system load, capacity, user volume, response times, transit delay, latency. Possibilities for scheduled processing vs real-time.

Maintainability

Amount of effort required to maintain (and enhance) application/solution in production

Authorisation

Security requirements to ensure users can access only certain functions within the application (by use case, subsystem, web page, business rule, field level etc)

Localisation

Support for multiple languages on entry/query screens in data fields; on reports; multi-byte character requirements and units of measure or currencies

Testability

1. Introduction 
2. Requirements 
 - 2.1. User Stories
 - Functional Requirements
 - Non-functional Requirements
 - 2.2. GUI 
 - Sketch
3. Domain model 

- During this phase you should start out technical prototyping
 - GUI
 - Services (file handling, sound, graphics, Android, etc....)
 - Hard code, mock anything you need

```
public void initMaterials() {
    wall_mat = new Material(assetManager,
        "Common/MatDefs/Misc/Unshaded.j3md");
    TextureKey key = new
        TextureKey("Textures/Terrain/BrickWall/BrickWall.jpg");
    key.setGenerateMips(true);
    Texture tex = assetManager.loadTexture(key);
    wall_mat.setTexture("ColorMap", tex);
}

private JPanel createCardsPanel() {
    int size = board.size();
    JButton[] wall_buttons = new JButton[size];
    JPanel pnl = new JPanel();
    pnl.setLayout(new GridLayout(size, size));
    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            JButton b = new JButton();
            b.setBackground(cardBack);
            b.addActionListener(this);
            b.setName(row + ":" + col); // Use this as Lookup Later,
            // see actionPerformed
            b.setPreferredSize(new Dimension(WIDTH / size, HEIGHT /
            size));
            pnl.add(b); // Add to panel
            cardButtons[row][col] = b; // Store so we can access later
        }
    }
    return pnl;
}
```

Analysis

Need for design

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE

Overarching goal:

***use abstraction to keep the design of
your application manageable***

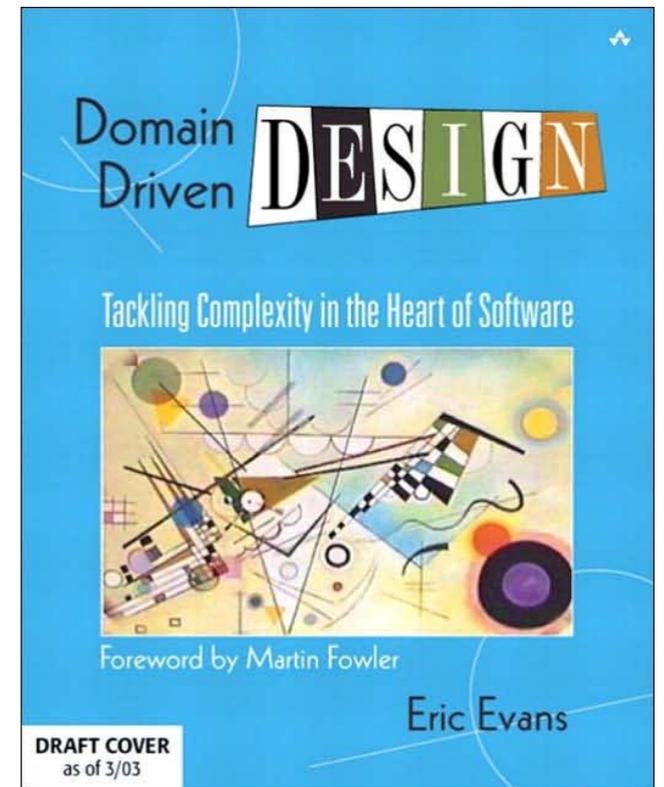


- Analysis is the second phase in the process
 - *During analysis we try to create a model of the problem domain as a collection of interacting objects*
- The Domain model
 - Is the core of our application (domain modelling)
 - The model is an abstraction of some problem
 - Is input for the design model
 - Should be kept in sync!
- Based on User Stories and idea, have to find:
 - Objects and how they are related (associations)
 - Classes for the objects
 - To a lesser degree: attributes, behaviour (methods)
 - Avoid too many details (inheritance, ...)

Domain-driven design

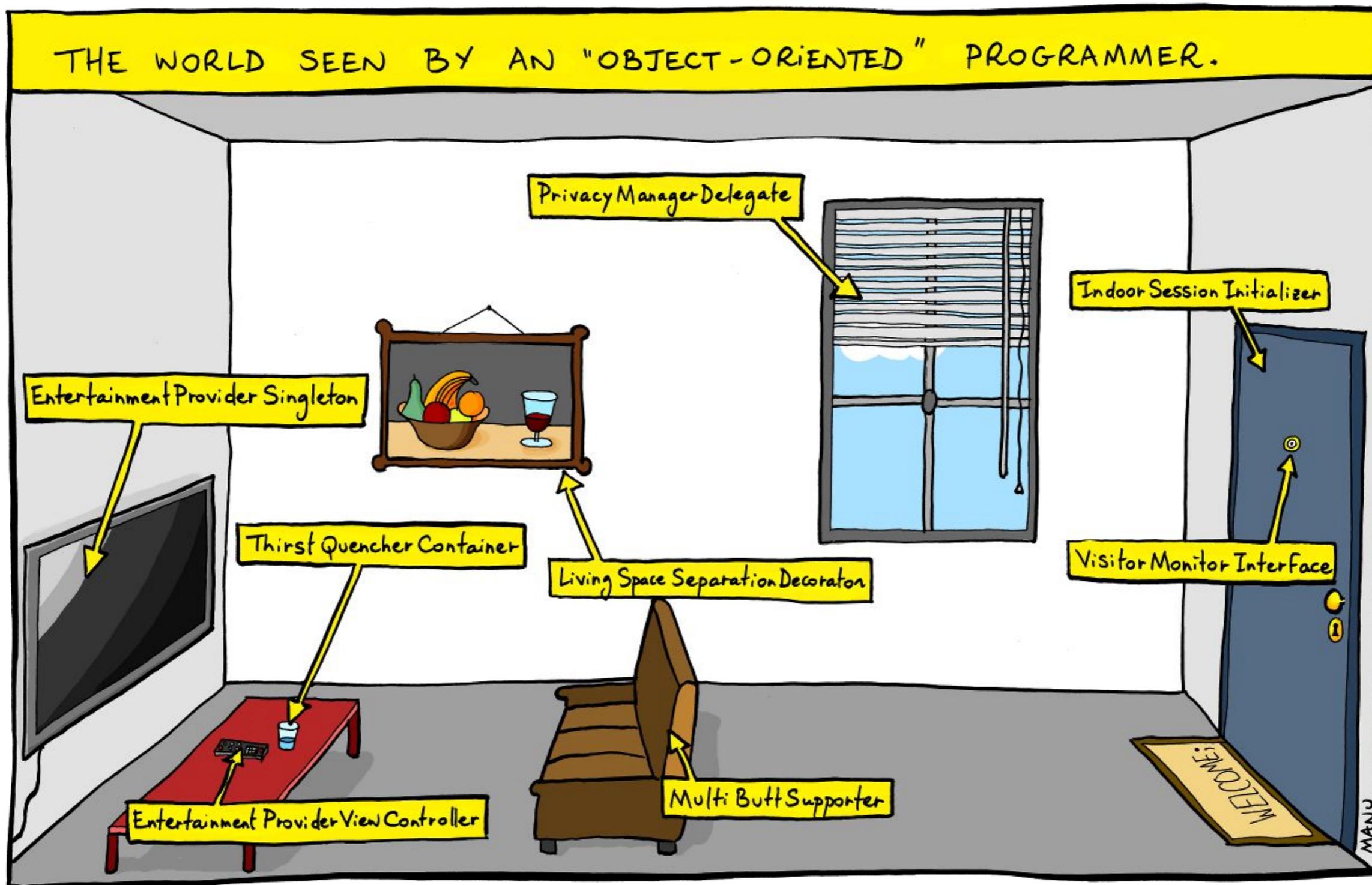


- Domain-Driven Design by Eric Evans
- The book addresses the analysis and design of software based on domain knowledge
- Pretty advanced, but useful
- Free compact version available, link on course website



- During this phase we adhere to domain driven design
 - focus on the core domain and domain logic
 - Explore models in a creative collaboration of domain practitioners and software practitioners.
 - Using the (ubiquitous) language of the domain
- Solution to the problem lies in the domain model (implies fat classes)
- Design application based on model of the domain

No 'technobabble'



- A common language between the domain experts and the developers
- The Domain model should be based heavily on the Ubiquitous Language
- We have a (sub)section in the RAD on this
- The common language connects the different models:
 - Domain Model -> Design Model -> Implementation

A central model



- Different roles in a project: domain experts, designers, developers, users
- Communication is difficult but essential:
 - Use common language
 - Have a central model (the domain model)
- Domain model facilitates discussion
- Iteratively develop the domain model
- Reflect and keep the domain model up-to-date
 - Do not allow domain model, design model and implementation to diverge

- To maintain the correspondence between model and implementation there are specific techniques that Eric Evans suggests.
 - Isolate the domain using a layered architecture
 - Domain layer techniques
 - ▶ Use associations wisely
 - ▶ Use appropriate model elements
 - ▶ Utilize Modules

Domain Layer Building Blocks



- Associations
- Three patterns of model elements
 - Entities
 - An object that represents something with continuity and identity – something that is tracked through different states or even across different implementations
 - Value Objects
 - Attribute that describes the state of a particular object aspect
 - Services
 - Actions or operations
- Modules
 - “The ideas of high cohesion and low coupling, often thought of as technical metrics, can be applied to the concepts themselves. In a MODEL-DRIVEN DESIGN, MODULES are part of the model, and they should reflect concepts in the domain (pg. 82).”

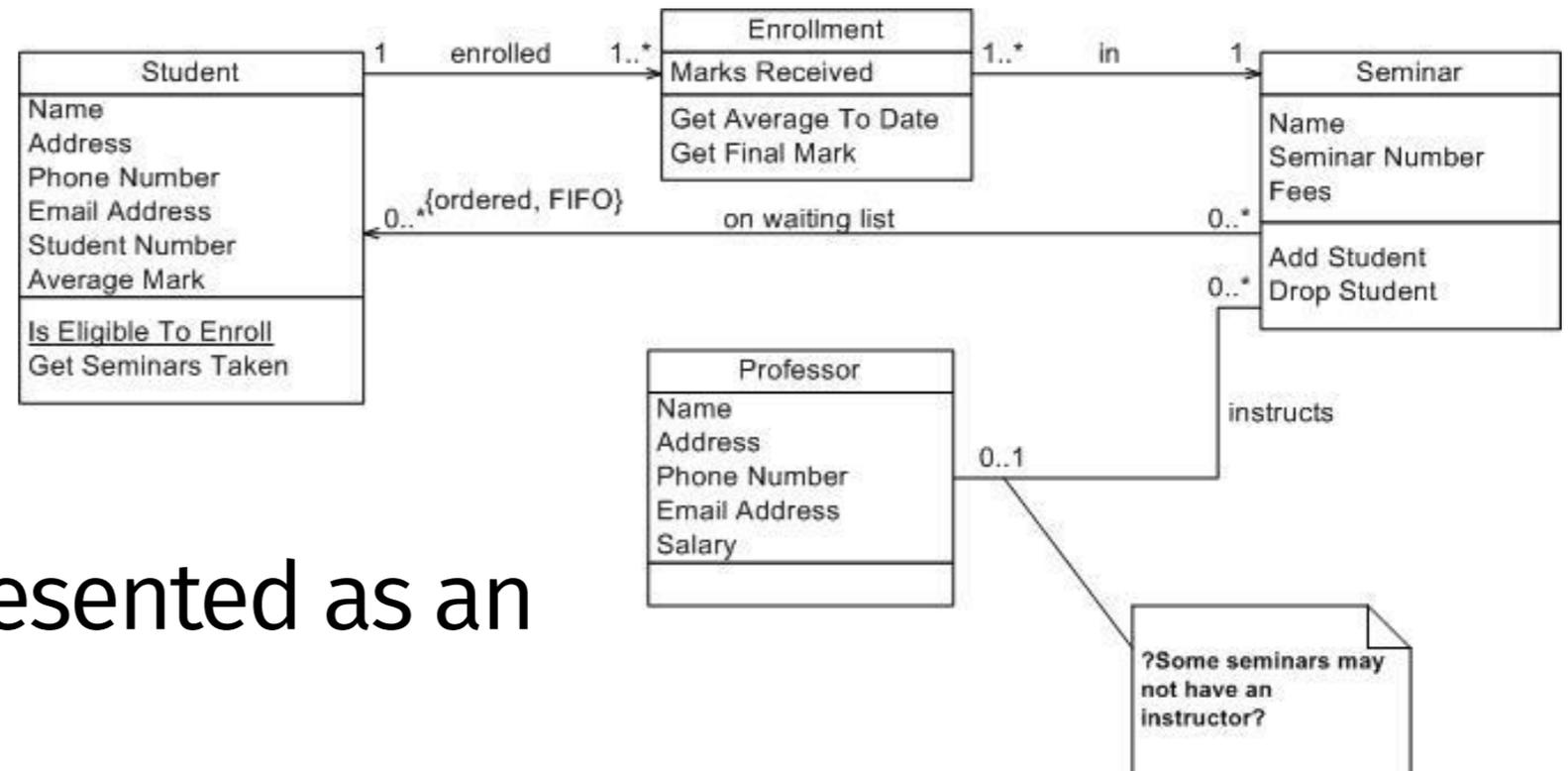
- Based on User Stories from the RAD start with the following simple method:
 - Underline nouns in use cases, will become classes
 - Underline verbs in use cases, will become methods
 - Sometimes hard to know which method belongs to which class,
 - put them in any that seems sensible, improve in next iteration
 - or leave out for now, will show up later!
 - Include as much as possible
 - Easy to skip later

Monopoly classes



- Monopoly
- Dice
- Piece
- Board
- Space (Street, Electricity, etc. ...)
- Jail
- Card
- Rent
- Player
- Balance
- Building
- Bank

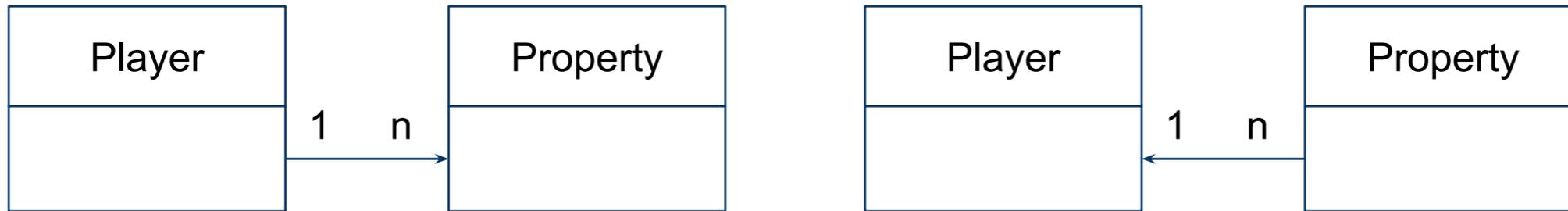
UML class diagram



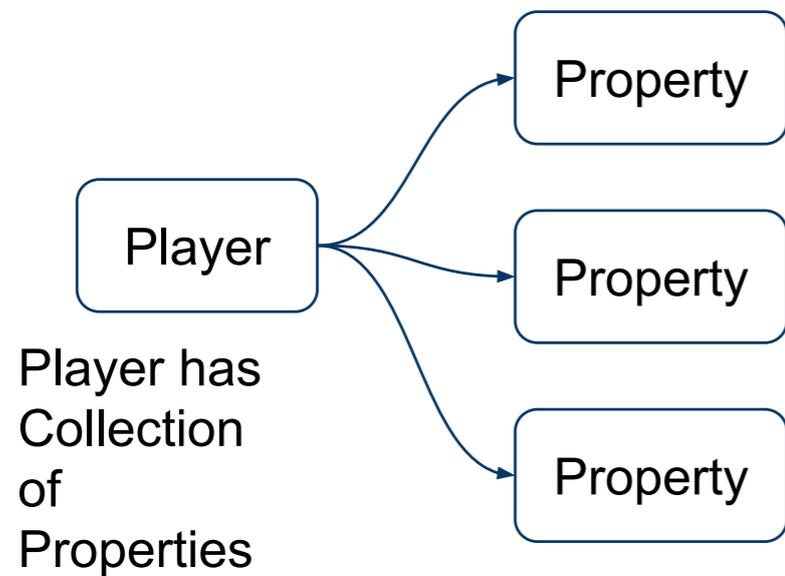
- Domain Model represented as an UML class diagram
 - Leave out many details
 - A static view
 - NOTE: Associations and multiplicity is between objects
 - Use standard notation!
- The diagram has a meaning.
 - Symbols, notations etc. should end up as runnable code!

- A model typically has many associations which can make implementation and maintenance complicated (especially many-to-many associations)
- Making associations more tractable
 - Impose a traversal direction
 - Add a qualifier
 - Eliminate nonessential associations
- This makes associations more expressive of the model as well as more tractable
- Again, use User Stories as a source of inspiration

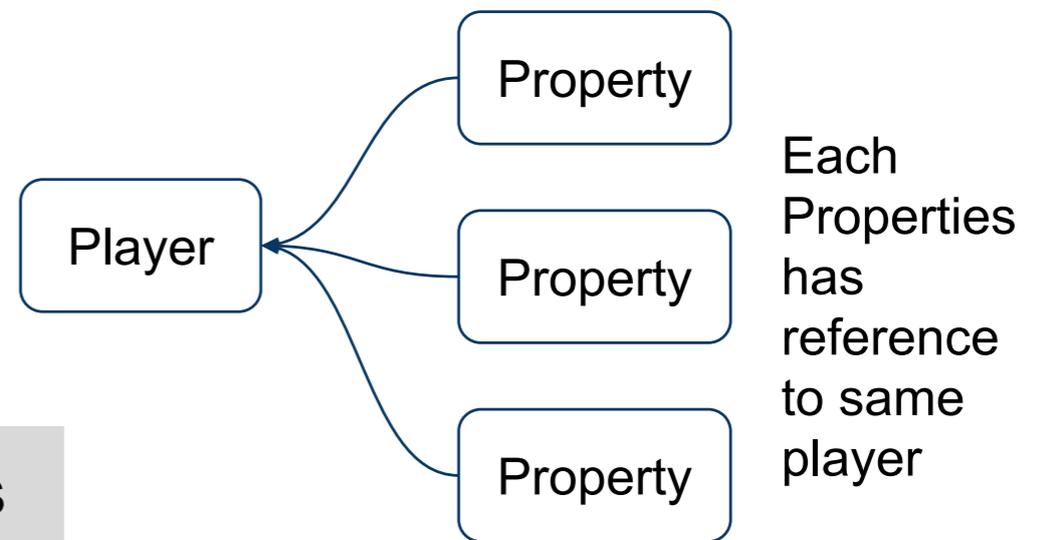
Multiplicity and direction



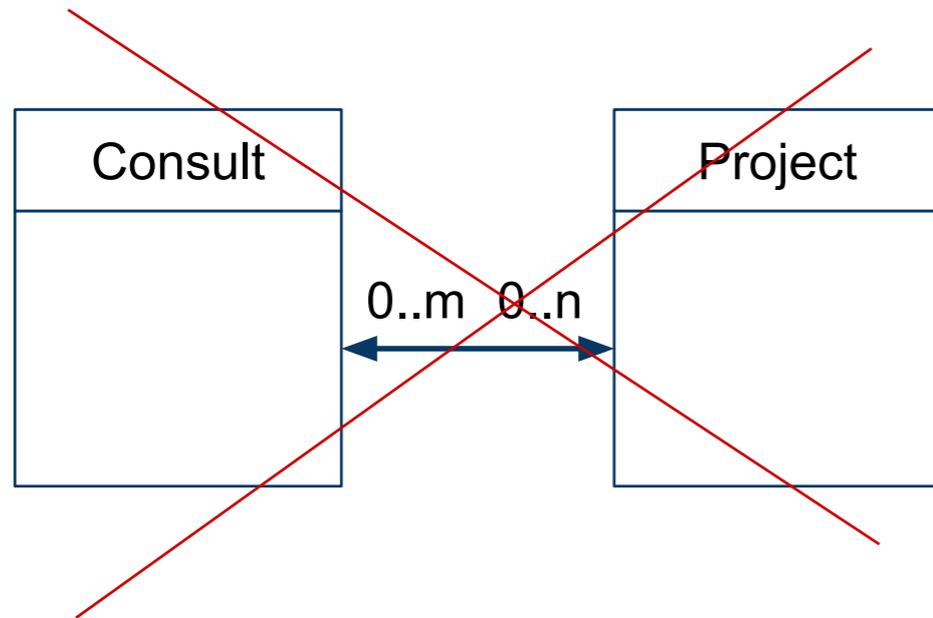
Same multiplicity



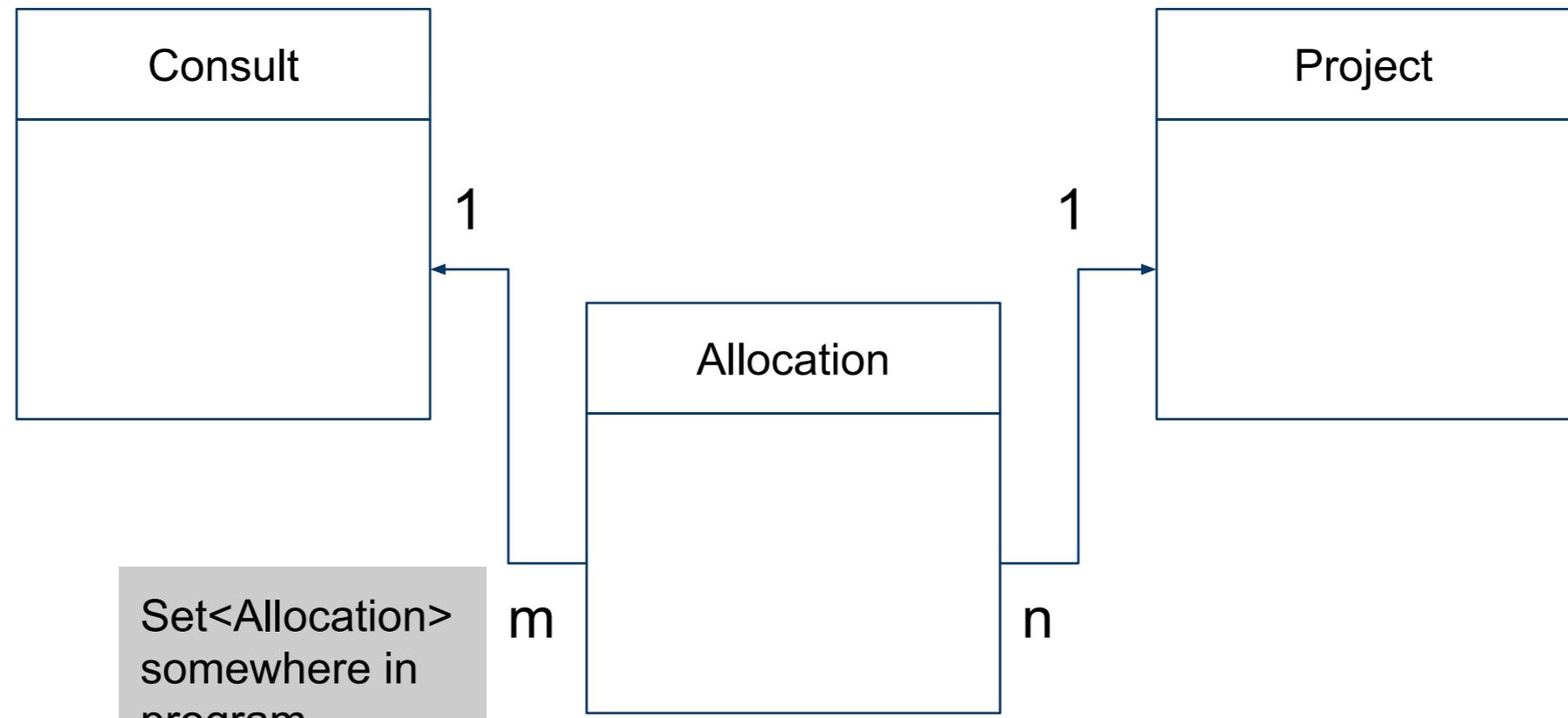
Objects



Association class



M:N
bidirectional

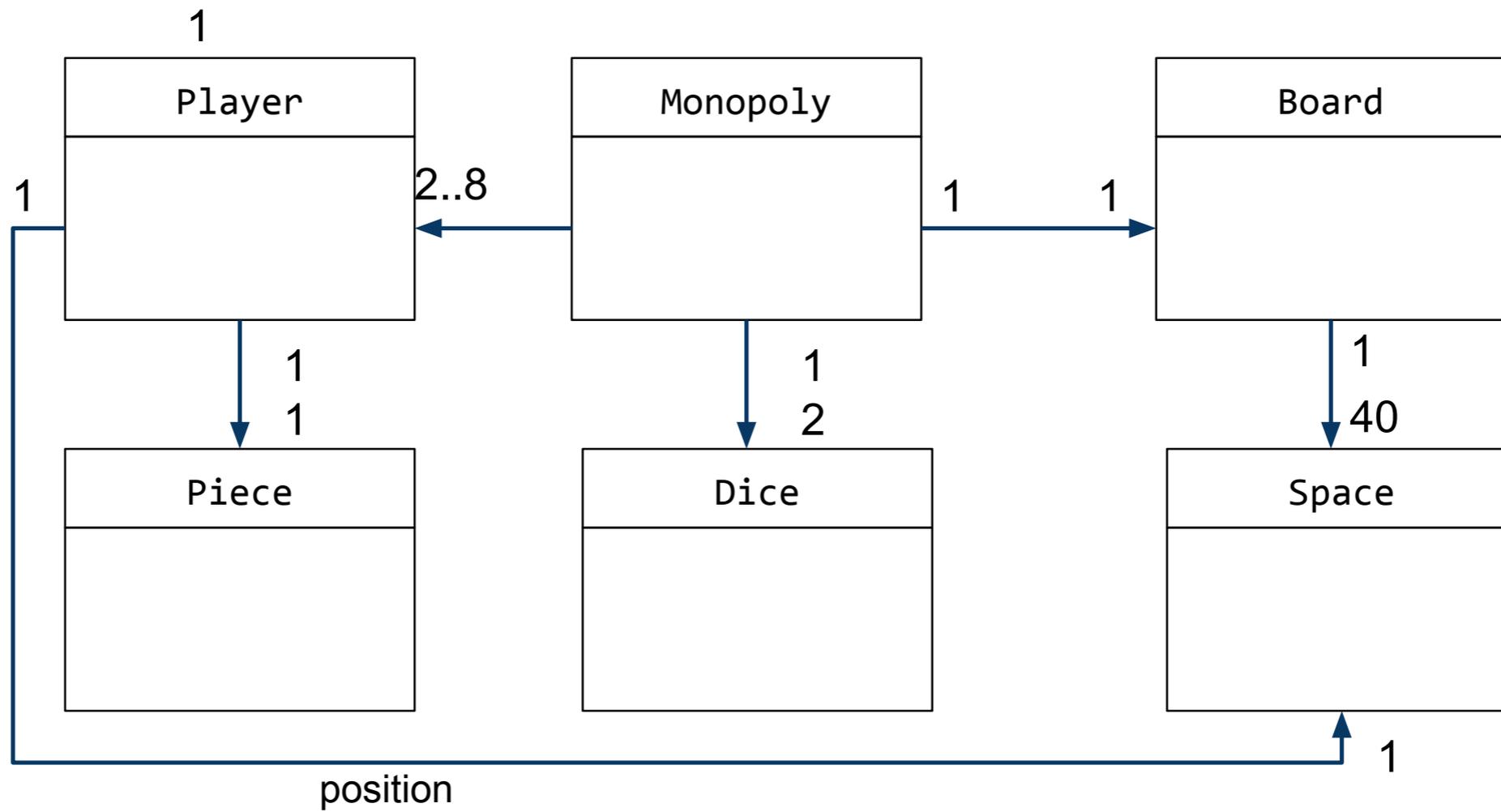


Set<Allocation>
somewhere in
program

- Mutual (bidirectional) associations should be avoided
 - Must keep two object in sync (reference each other) i.e. if new owner have to change 2 references
 - Domino effects (change one, affect other)
 - Classes not understood in separation
- Select association that seems to be used most, remove other

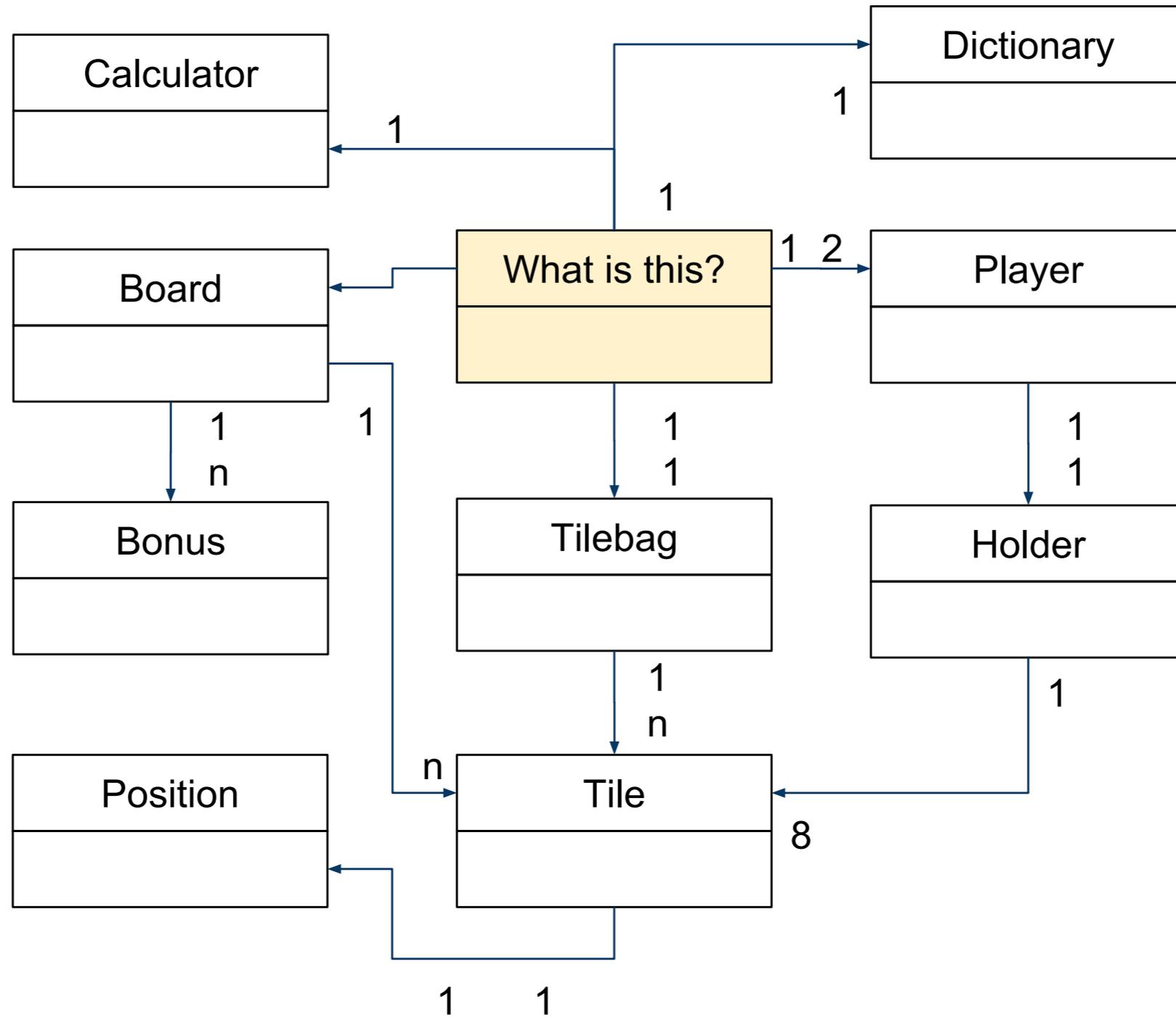


Monopoly domain model

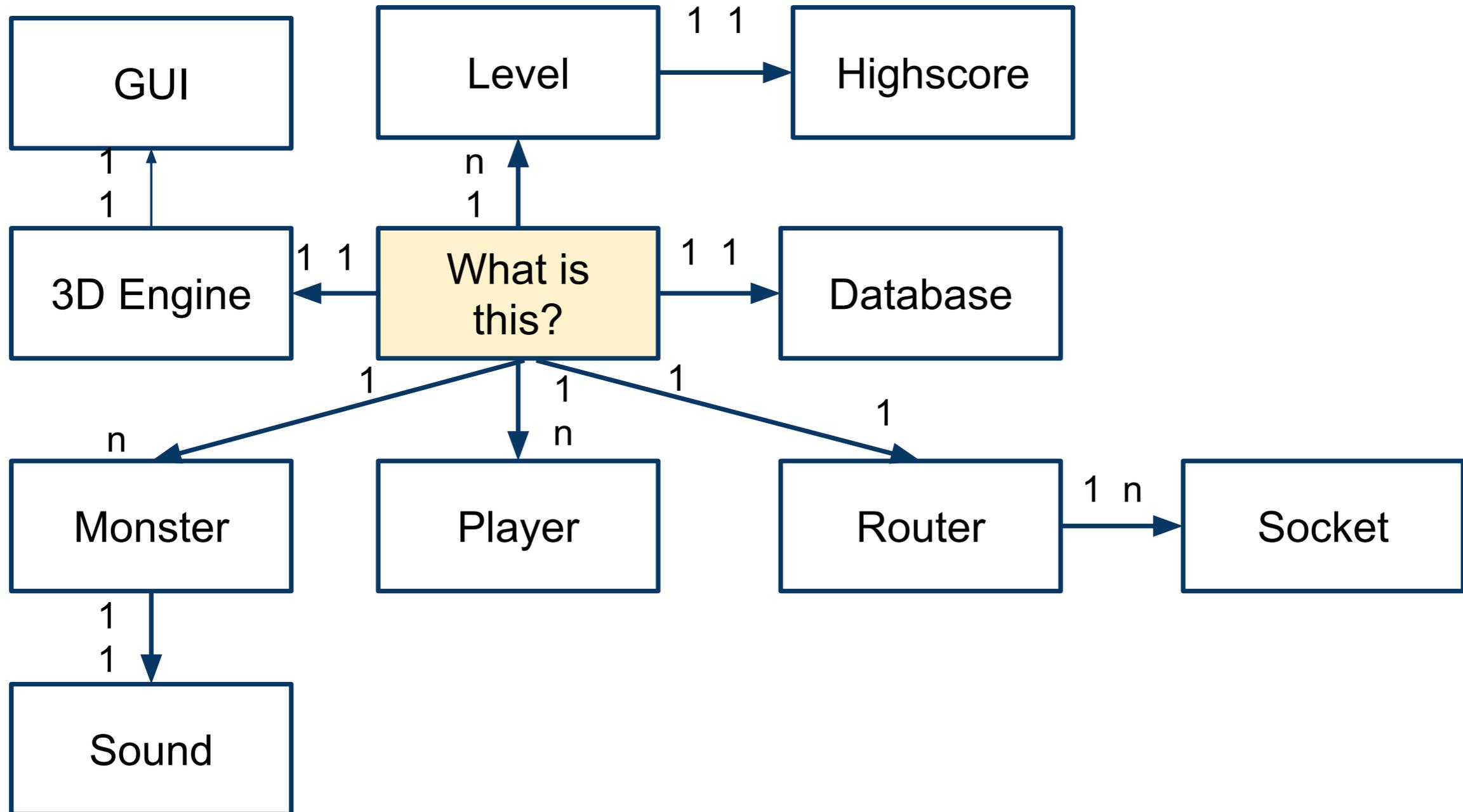


First iteration!

Another domain model



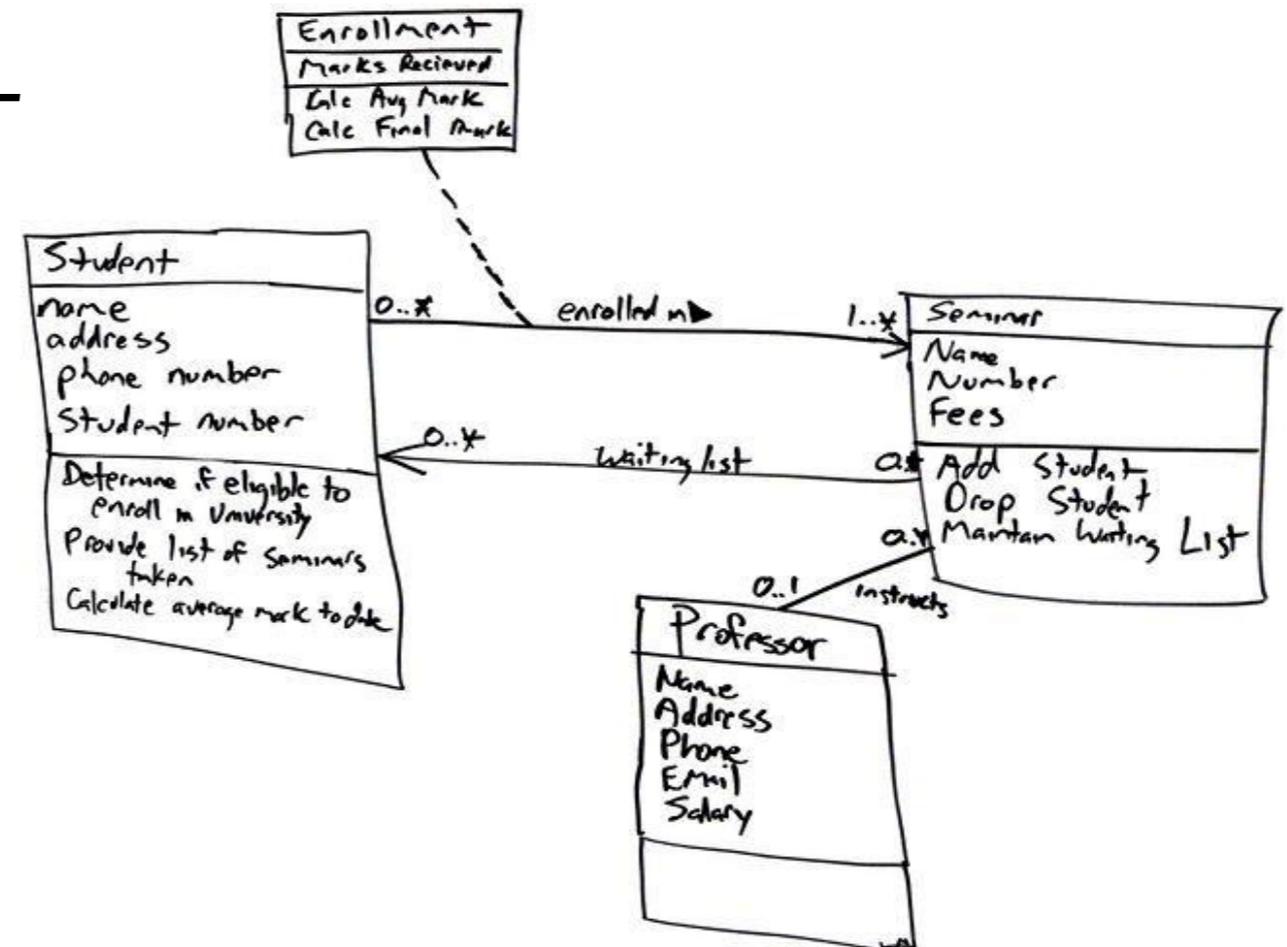
Yet another domain model



Efficient modelling

- Optimal is to first draw on whiteboard!
 - Very fast drawing
 - Very fast communication, everyone can participate
 - Use phone/camera to document

- Later use tools to draw UML



Important Object Characteristics



- Unique identity?
- Equality?
- Immutable?
- Persistence?
 - Will any objects survive the execution of the program?
- Lifecycle
 - When is object created?
 - How long does it exist?
 - When destroyed?
- ...

```
public class MyClass {  
  
    private ... data;  
    private ... moreData;  
    private ... yetMoreData;  
  
    public ... setData { ...}  
  
    public ... getData { ...}  
  
    public ... setMoreData { ...}  
  
    public ... getMoreData { ...}  
  
    public ... setYetMoreData { ...}  
  
    public ... getYetMoreData { ...}  
  
}
```

No
behaviour!

```
public class Board {
    private final List<Card> cards;

    public List<Card> unSelectPair() {
        List<Card> s = new ArrayList<>(selected);
        selected.clear();
        return s;
    }

    public List<Card> removeSelected() {
        List<Card> s = new ArrayList<>(selected);
        cards.removeAll(selected);
        selected.clear();
        return s;
    }

    public boolean hasMatchingPair() {
        return selected.size() == 2 &&
            selected.get(0).equalsByName(selected.get(1));
    }
}
```

Data and
behaviour!

- 3. Domain Model 
- 3.1. Class responsibilities 

Best practices for modelling (Evans)

Isolating the domain



User Interface Layer

- A.k.a. Presentation Layer
- Show Information
- Interpret commands

Application Layer

- Thin layer, directs UI commands to jobs in the Domain Layer
- Should not contain Business Rules or Knowledge
- No business “state”, may have progress “state”

Domain Layer

- Business objects, their rules, and their state
- The majority of the book focuses here

Infrastructure Layer

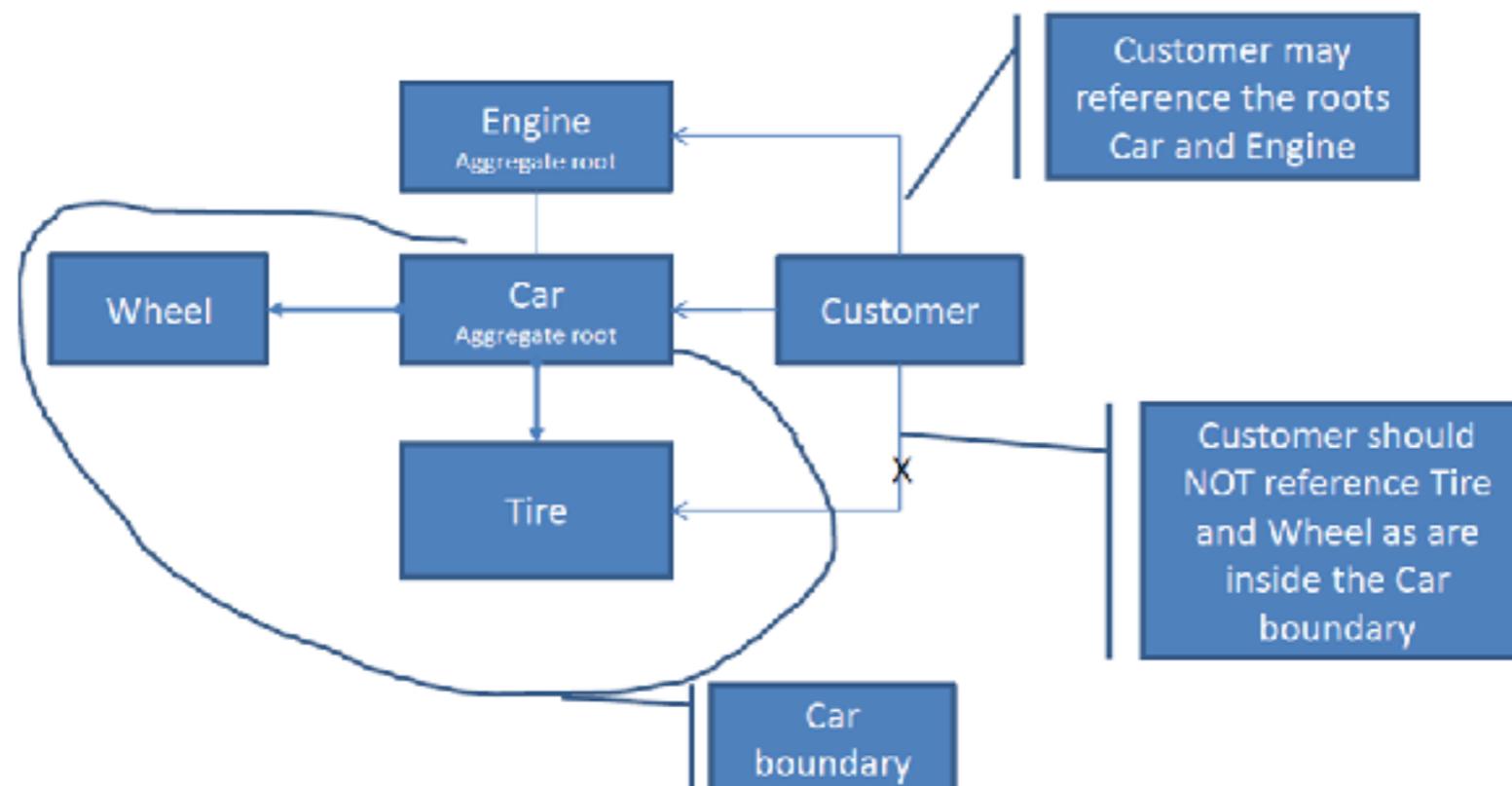
- Generic technical capabilities to support the higher layers
- Message sending, persistence
- Supports the interactions between topmost patterns

- From Evans:
 - An object defined primarily by its identity is called an entity
 - What is the identity?
 - ▶ Consider two person objects: same name, same date of birth etc.
 - ▶ Generate an identifier
 - Their class definitions, responsibilities, attributes, and associations should revolve around who they are, rather than the particular attributes they carry

- Could make all objects entities...
 - "Software design is a constant battle with complexity. We must make distinctions so that special handling is applied only where necessary (pg. 98)."
 - Only use entities where necessary
- An object that represents a descriptive aspect of the domain with no conceptual identity
- It is recommended that value objects be *immutable*
- "[I]nstantiated to represent elements of the design that we care about only for what they are, not who or which they are (pg. 98)."
- Examples of possible Value objects:
 - Money/Currency class
 - Point class in a drawing application

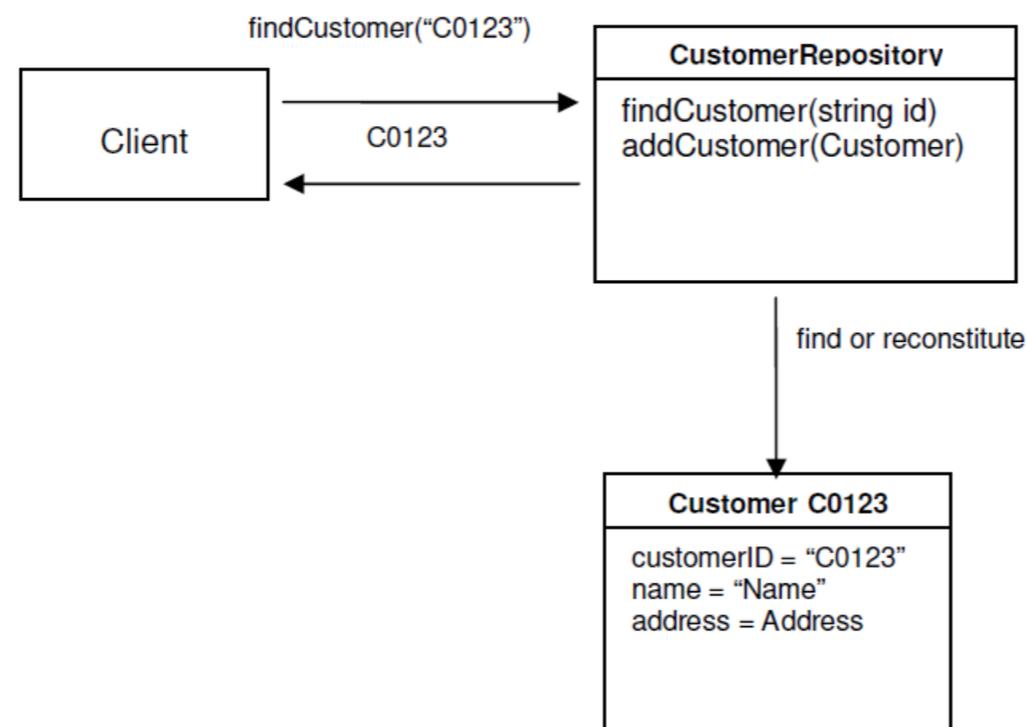
- Some aspects of the domain don't map easily to objects
- A Service is some behaviour, that is important to the domain, but does not “belong” to an Entity or Value object
- Example: Account Transfer
- Encapsulate an important domain concept
- Operation names should come from the UBIQUITOUS LANGUAGE
- Parameters and results should be domain objects, the operation in itself is stateless
- Note: There is a distinction between services discussed here that are used in the domain layer and those of other layers. Technical services lack business meaning.

- A group of associated objects which are considered as a unit with regard to data changes
- An aggregate should have one root
- The root is an entity object
- Outside objects can reference root, but not the other members of the aggregate



- Encapsulate the information necessary for object creation
 - Includes logic for all creating all the members of an aggregate
 - Allows us to enforce invariants during creation
 - Related GoF Design Patterns
 - ▶ Factory Method
 - ▶ Abstract Factory
 - Designing the Factory Interface
 - ▶ Each operation must be atomic
 - ▶ The Factory will be coupled to its arguments

- Encapsulates logic to obtain object references
- Provides a mechanism to persist/retrieve an object
 - Keeps persistence code out of the domain layer
- Repository interface should be driven by the domain model
- Repository implementation will be closely linked to the infrastructure



- “MODULES give people two views of the model: They can look at detail within a *MODULE* without being overwhelmed by the whole, or they can look at relationships between *MODULES* in views that exclude interior detail (pg. 109).”
- The *MODULES* in the domain layer should emerge as a meaningful part of the model, telling the story of the domain on a larger scale (pg. 109).”
- *MODULES* can be dangerous since the cost of refactoring *MODULES* can be prohibitive
- “If your model is telling a story, the *MODULES* are chapters (pg. 110).”
- “Give the *MODULES* names that become part of the *UBIQUITOUS LANGUAGE* (pg. 111).”

Intention revealing interfaces

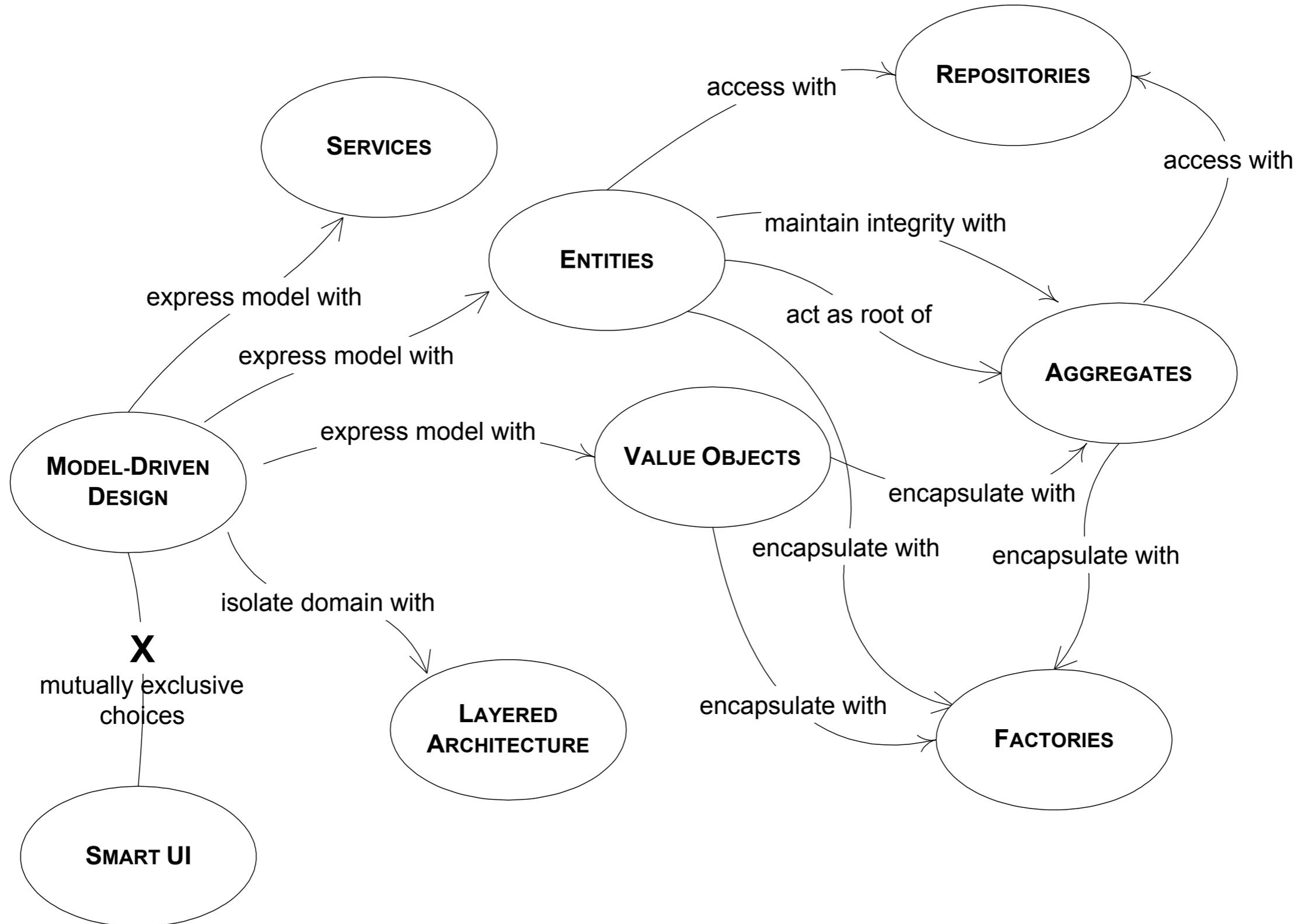


- “Type names, method names, and argument names all combine to form an INTENTION-REVEALING INTERFACE (pg. 247).”
- “Name classes and operations to describe their effect and purpose, without reference to the means by which they do what they promise (pg. 247).”
- “Write a test for a behaviour before creating it, to force your thinking into client developer mode (pg. 247).”

- “Interactions of multiple rules or compositions of calculations become extremely difficult to predict (pg. 250.)”
- To make code easier to use, separate calculations and state change into different operations.

- “Assertions make side effects explicit and easier to deal with (pg. 255).”
- “State post-conditions of operations and invariants of classes and AGGREGATES. If ASSERTIONS cannot be coded directly in your programming language, write automated unit tests for them (pg. 256).”

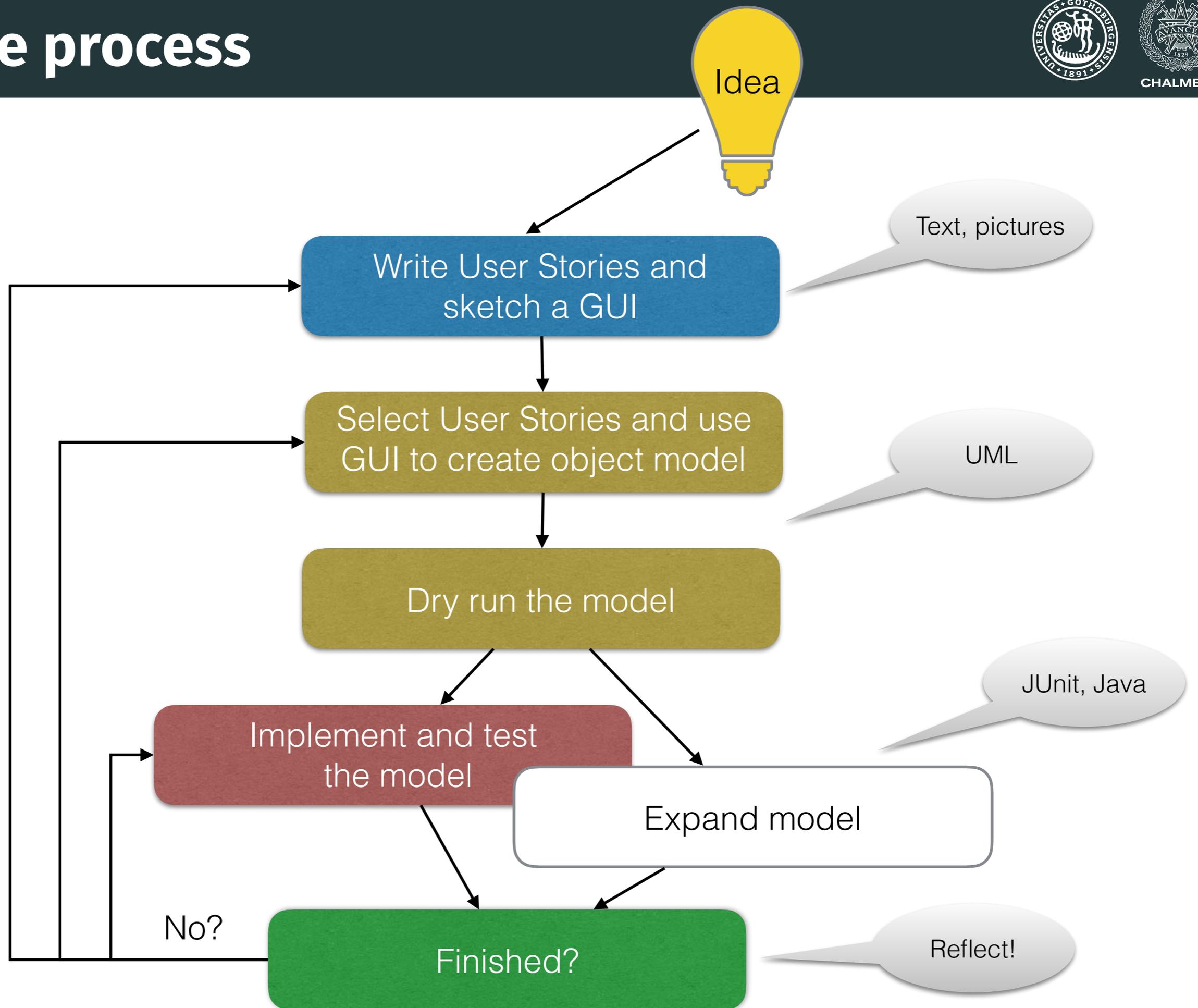
Navigation map



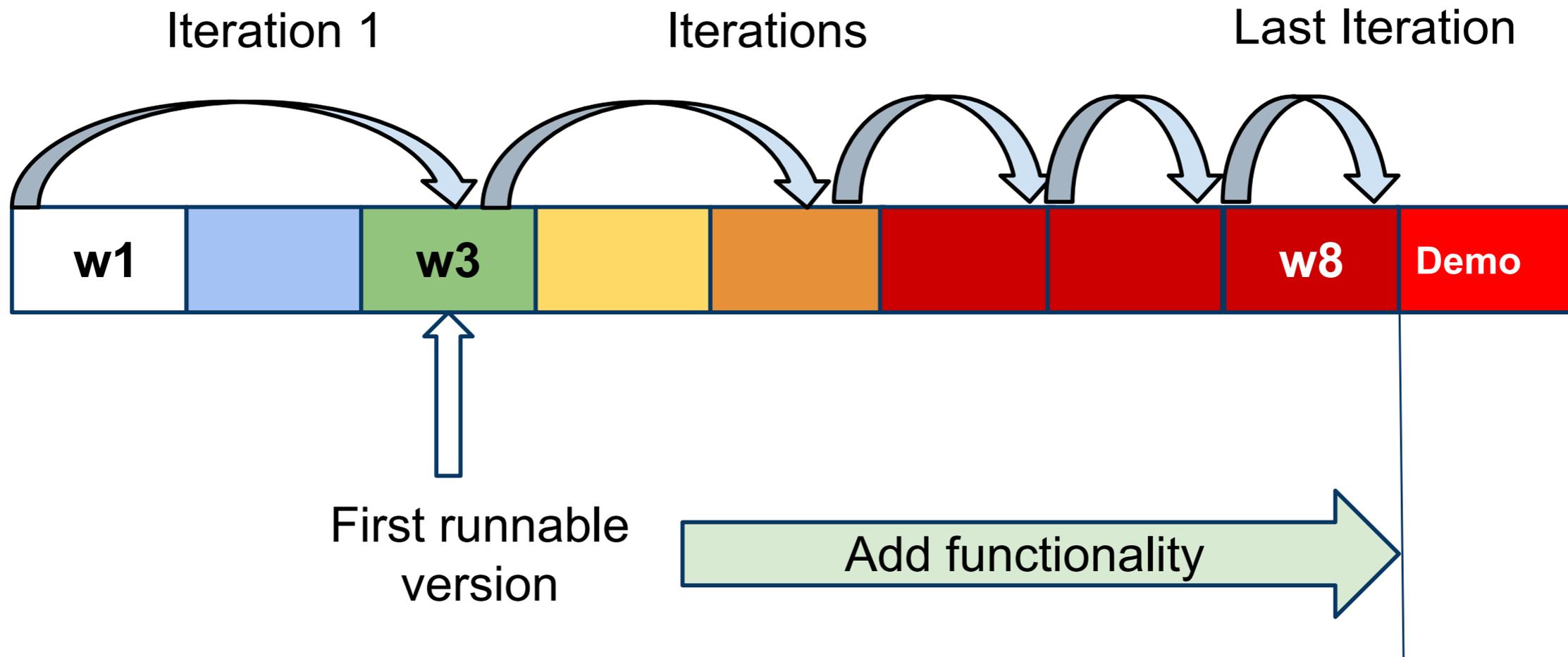
Summary

- Analysis focuses on building a **domain model**
 - We used the requirements from RAD (user stories) to extract the domain model
 - We expressed the model as an UML-class diagram
 - We documented model in RAD
- Next: From domain model to first implementation

Course process



Iteration planning



- <https://www.cs.colorado.edu/~kena/classes/5448/f12/presentation-materials/roads.pdf>
- selab.netlab.uky.edu/homepage/CS618-DDD-Foundations.pdf
- <https://domainlanguage.com/ddd/reference/>