Software technology

07 – OOP Principles, Refactoring, Modeling

Dependency Injection

Lecture: Zoltán GERA / Practice: László GRAD-GYENGE

Editor & Presenter: Dr. Attila GLUDOVÁTZ

Online catalog – every week

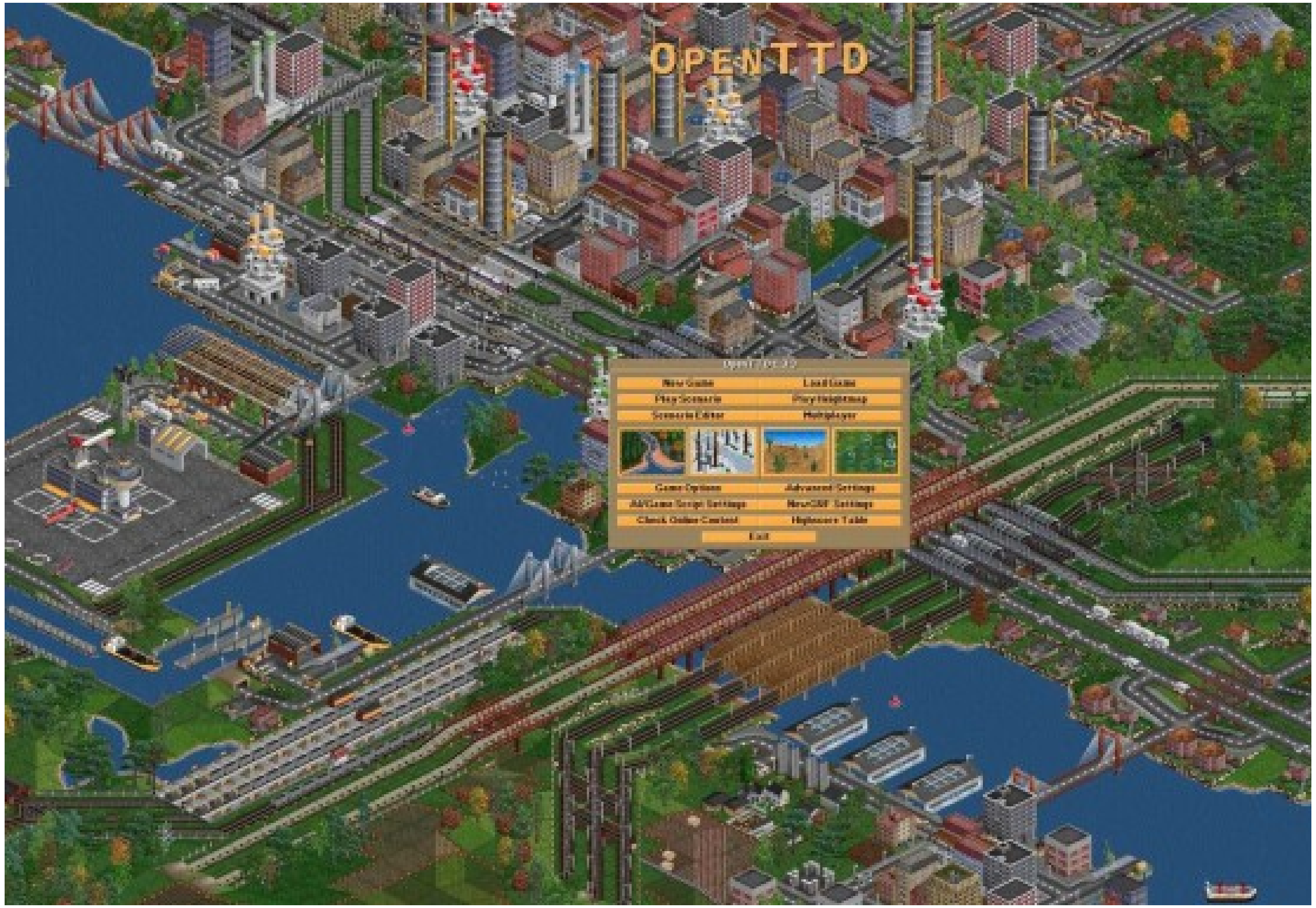
- https://catalog.inf.elte.hu/
- Log in
- Username: yourUsername (@inf.elte.hu)
- Password: your email password
- Captcha: I generate a number for you…
- Lecture attendance is ***not*** optional! Max 3 misses and you are out

Why OOP? (Object-oriented programming)

- What is it?
- Why are we using OOP?
- What are the tools for OOP?
- What is a good OOP design?
    - How can we evaluate?
    - How can we generate?

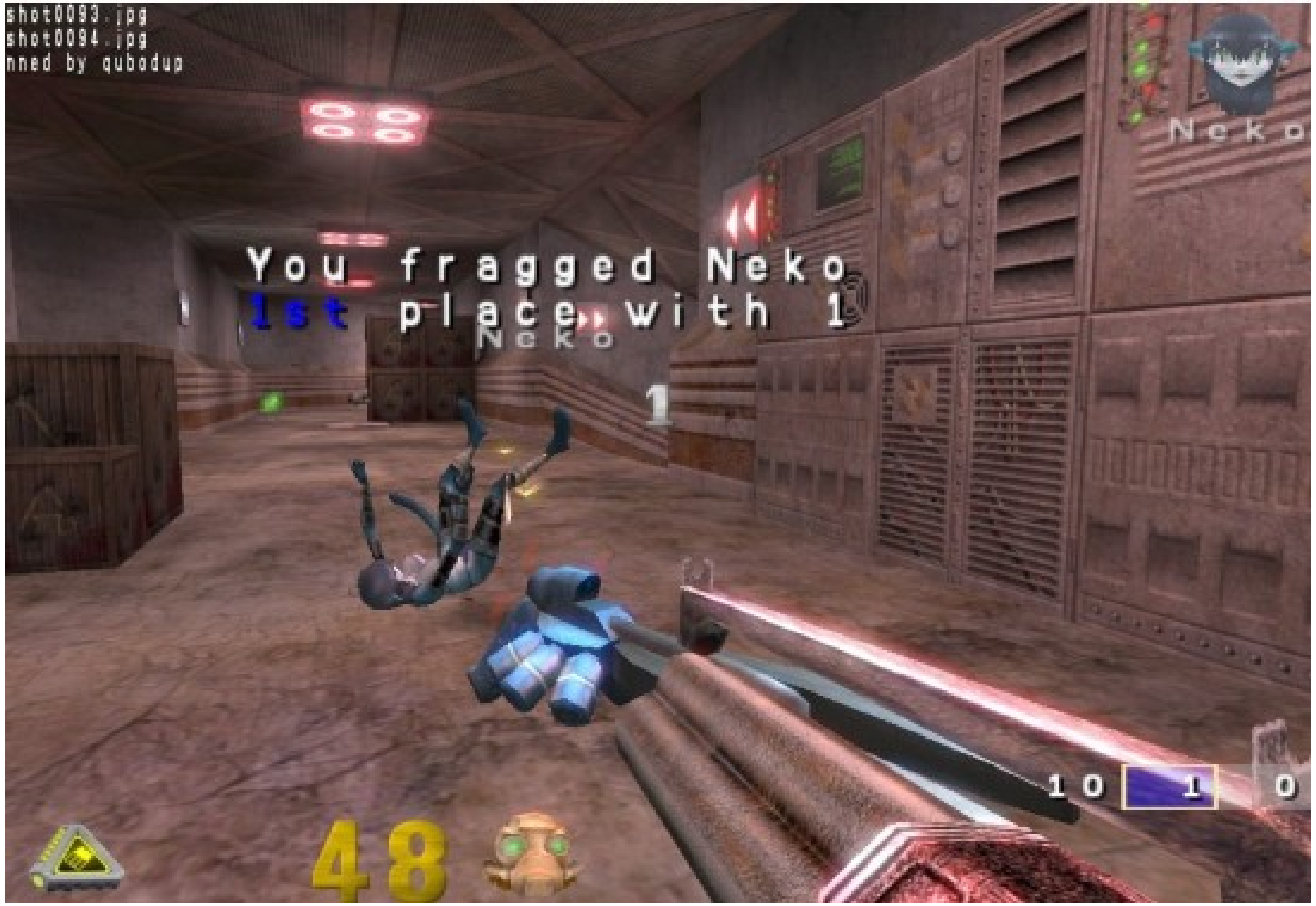
OOP Example

How would you sketch up the object model of this game?



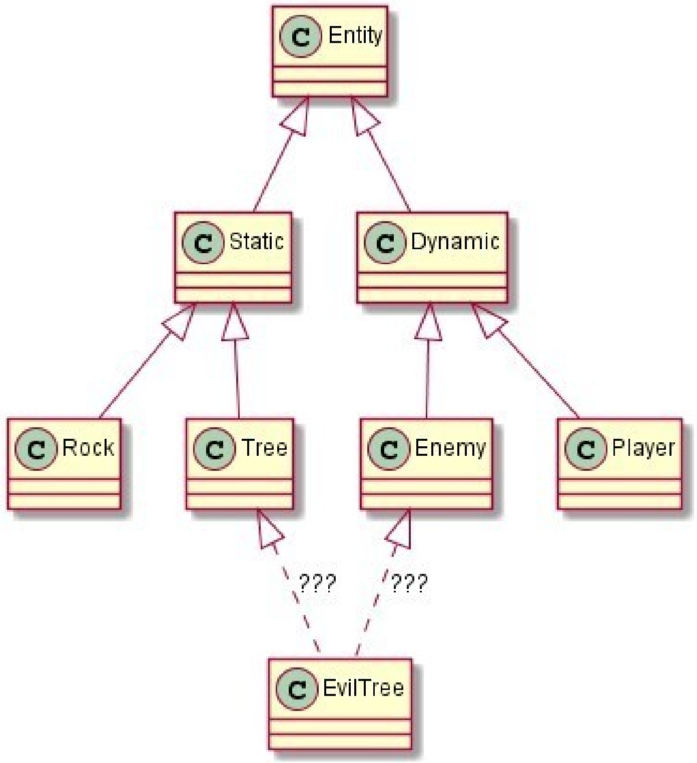
OOP Example

How would you sketch up the object model of this game?



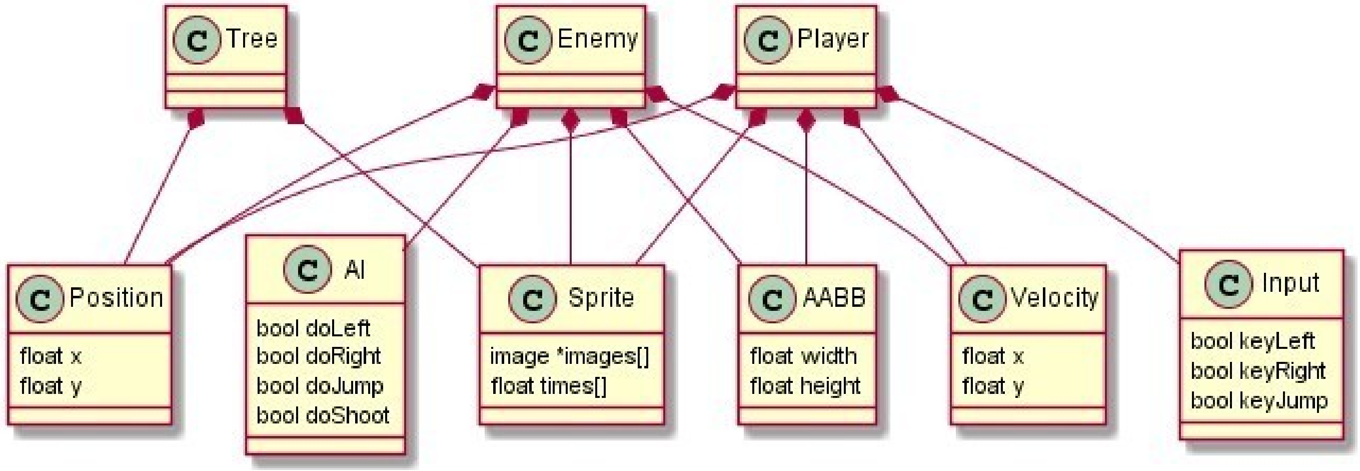
OOP Example

- Not everything can be solved by inheritance or classic OOP constructs (at least not in the classic way)



OOP Example

- Composition over inheritance → ECS (Entity-Component-System)
- ECS is a high-level compositional design pattern



OOP Tools

- Classes, Objects…
- Composition or inheritance or delegation?
- Dynamic dispatch (late binding) or message passing
  - What about static (parametric) polymorphism? (templates)
- Goals
  - Reusability
  - Maintainability
  - Support teamwork

OOP Tools

- Tools are not enough…
  - Design Patterns
  - Control flow vs Data flow
  - Responsibility-driven Design
  - Data-driven Design

OOP Tools

- Unlimited ways of code + data grouping
  → How to do encapsulation?
  → Which one is the best?
- Maybe: Think about
  - SW processes
  - Feature introduction
  - Agile
- *Just because I am using OOP Tools does not mean that my design is any good*

SOLID

- 5 principles of Object-Oriented Programming and Design
- Principles to remove Code Smells
- **S** – Single Responsibility Principle
- **O** – Open/Closed Principle
- **L** – Liskov Substitution Principle
- **I** – Interface Segregation Principle
- **D** – Dependency Inversion Principle

SOLID

- **S** = A class should have one responsibility. Encapsulate along responsibility lines.
- **O** = Classes should be closed for modification (they are whole, complete, functioning units), but open for extension (structural extension example via polymorphism)
- **L** = Objects can be replaced by their subtype without ruining correctness.
- **I** = Large interfaces should be broken up. If I use an interface, I depend on it. I don’t want to depend on many. (Maximal separation)
- **D** = Depend on abstractions only, not concrete implementations

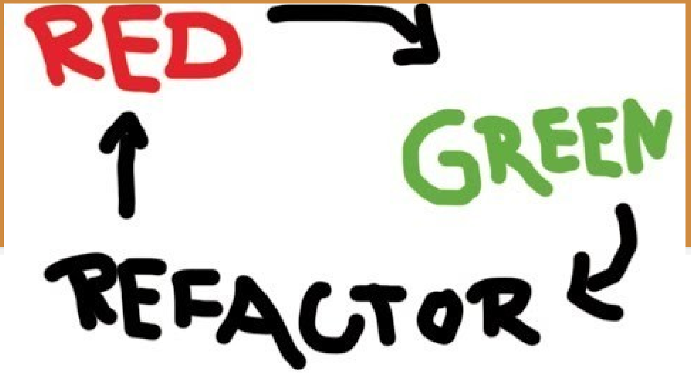
Architectural Improvements

1. Refactoring

2. Class Normalization

3. Design Patterns (and anti-patterns)

1. Refactoring



- No external behavior change, but
- Improve
  - Readability
  - Ease of understanding (lower complexity)
  - Extensibility
  - Maintainability

→ Improve all goals of OOP (teamwork)

1. Refactoring

- Program transformations
  - Rename (understanding – most important!!!)
  - Move
  - Break into components (new class or method)
    - Encapsulate
  - Generalize
  - Branching into Compound State or Polymorphic behavior
- Tools (…many IDEs)

refactor

engage ■ succeed ■ repeat

2. Class Normalization

- Comes from DB normalization
  - 1st object normal form (1ONF)
    - Encapsulate behavior of multiplicity >1
  - 2nd object normal form (2ONF)
    - Encapsulate any shared behavior
  - 3rd object normal form (3ONF)
    - Encapsulate one set of cohesive behavior per class
- Behavior = code, data or combination

3. Design Patterns

- *...coming soon! (next weeks)*
- Types
  - Creational
  - Structural
  - Behavioral
  - Concurrent
  - Architectural

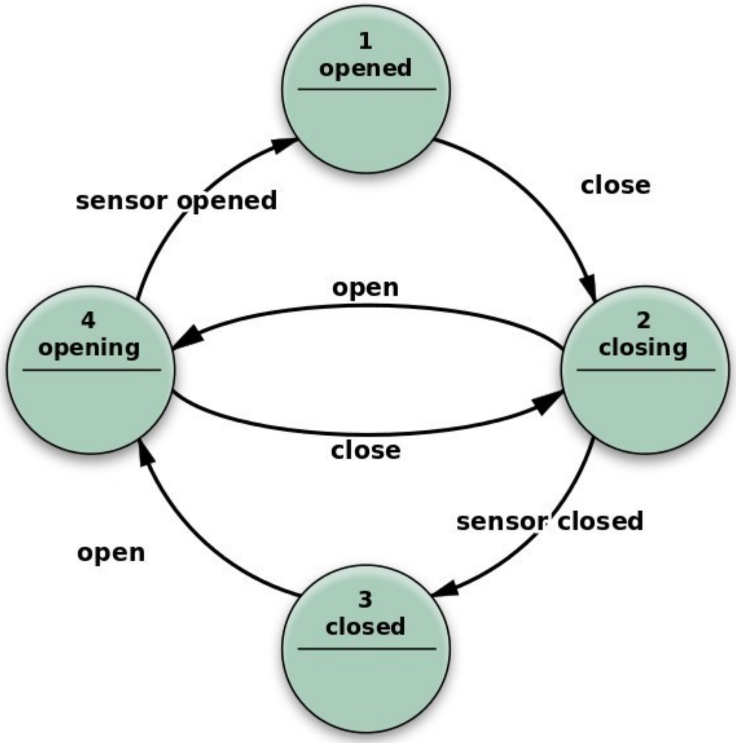
OOA (Object-Oriented Analysis)

- Structured analysis and design was something like
  - Sketch up system (to some level of detail)
  - Implement
  - Improve

- Instead do deeply precise analysis → OOA (Shlaer-Mellor Method)

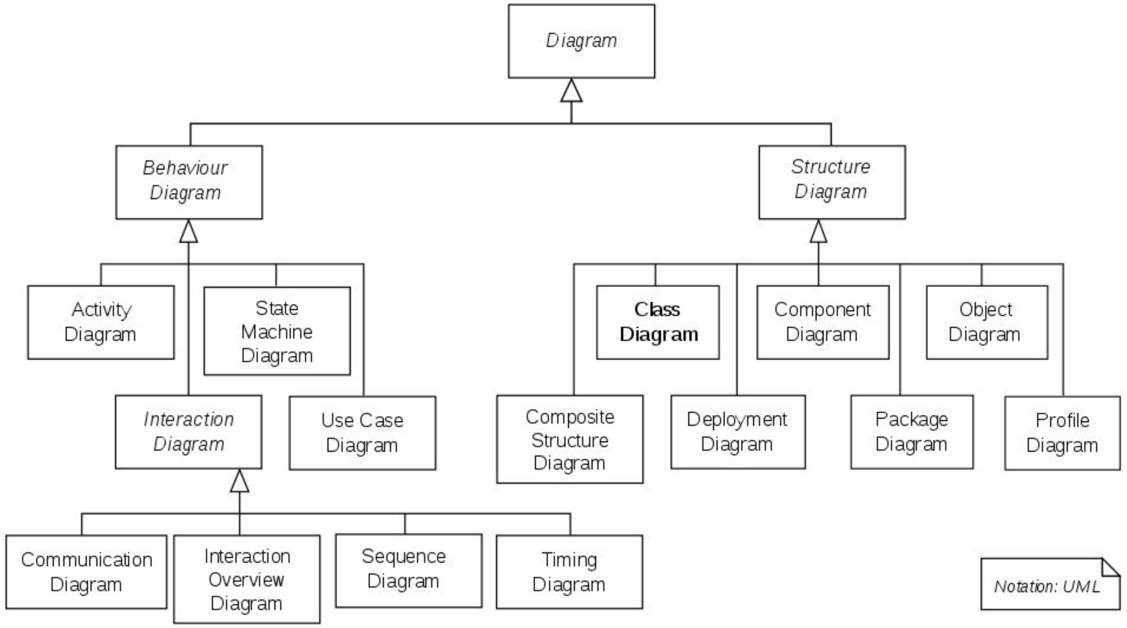
OOA (Object-Oriented Analysis)

- Translation instead of Elaboration
- Logic in Finite-State Machines
- Action Data Flow Diagram or Action Language
- Virtual Machine
- Cross language, Cross platform compilable
- Simulation
- Test

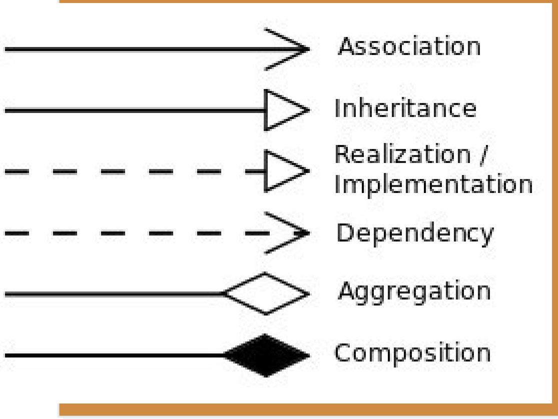


UML (Unified Modeling Language)

- Divided into 2 groups:
  1. Static (structural) view
  2. Dynamic (behavioral) view



UML
Class diagram



Association

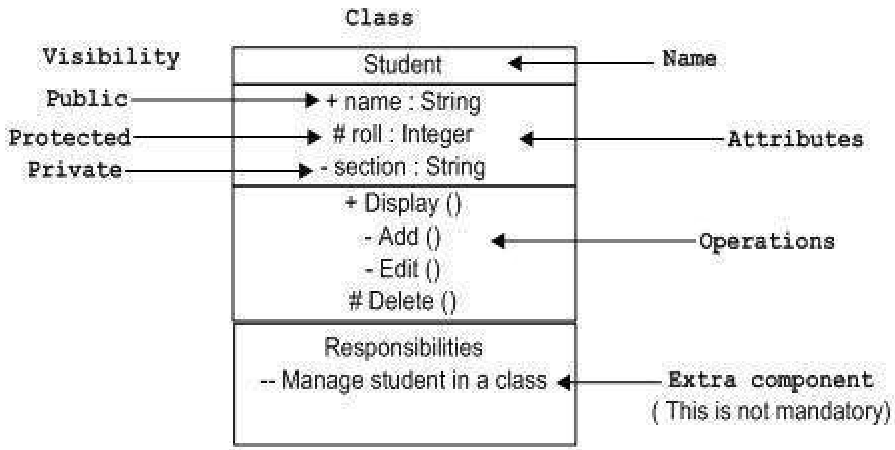
Inheritance

Realization / Implementation

Dependency

Aggregation

Composition



Class

Visibility

Student ← Name

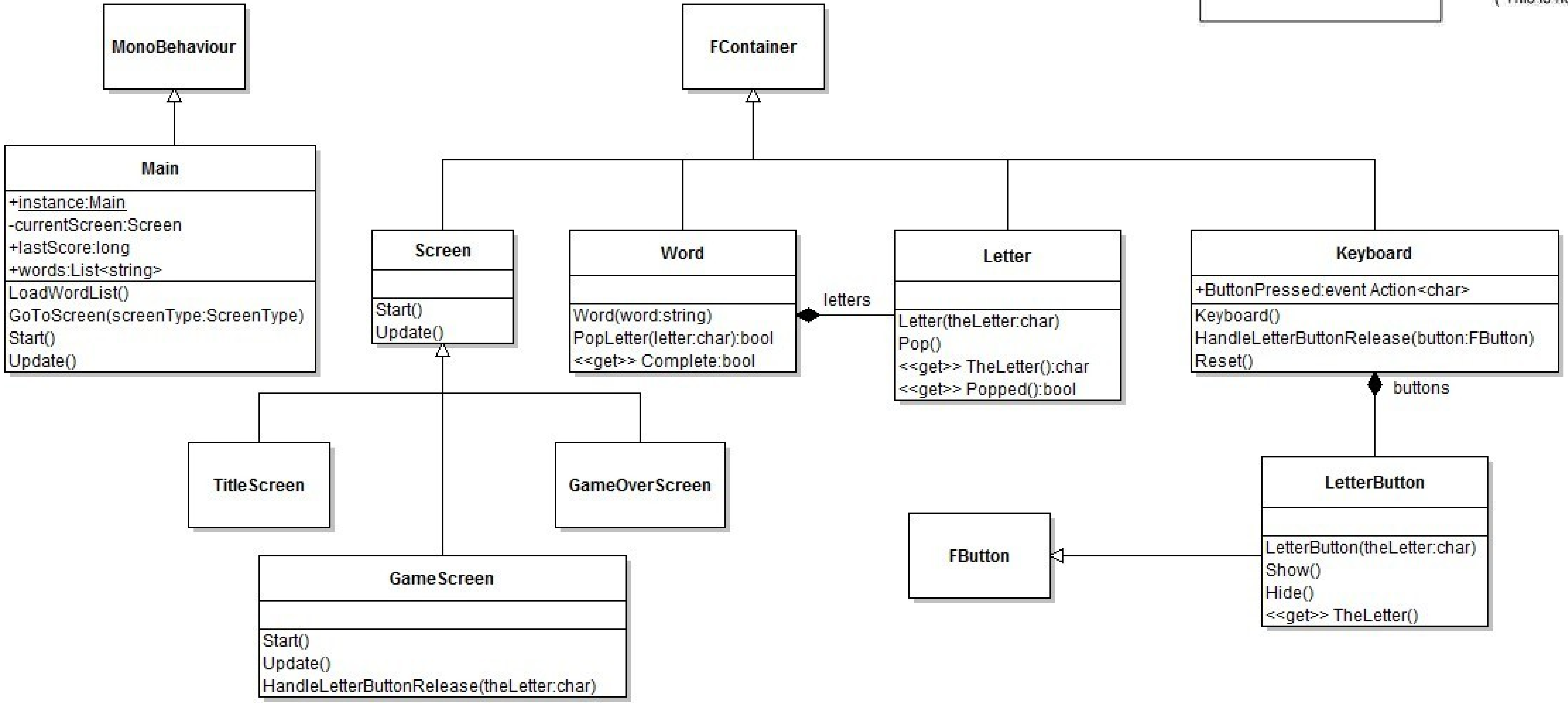
Public → + name : String

Protected → # roll : Integer ← Attributes

Private → - section : String

+ Display ()
- Add () ← Operations
- Edit ()
Delete ()

Responsibilities
-- Manage student in a class ← Extra component
(This is not mandatory)



MonoBehaviour

Main
+instance:Main
-currentScreen:Screen
+lastScore:long
+words:List<string>
LoadWordList()
GoToScreen(screenType:ScreenType)
Start()
Update()

FContainer

Screen
Start()
Update()

Word
Word(word:string)
PopLetter(letter:char):bool
<<get>> Complete:bool

letters

Letter
Letter(theLetter:char)
Pop()
<<get>> TheLetter():char
<<get>> Popped():bool

Keyboard
+ButtonPressed:event Action<char>
Keyboard()
HandleLetterButtonRelease(button:FButton)
Reset()

buttons

TitleScreen

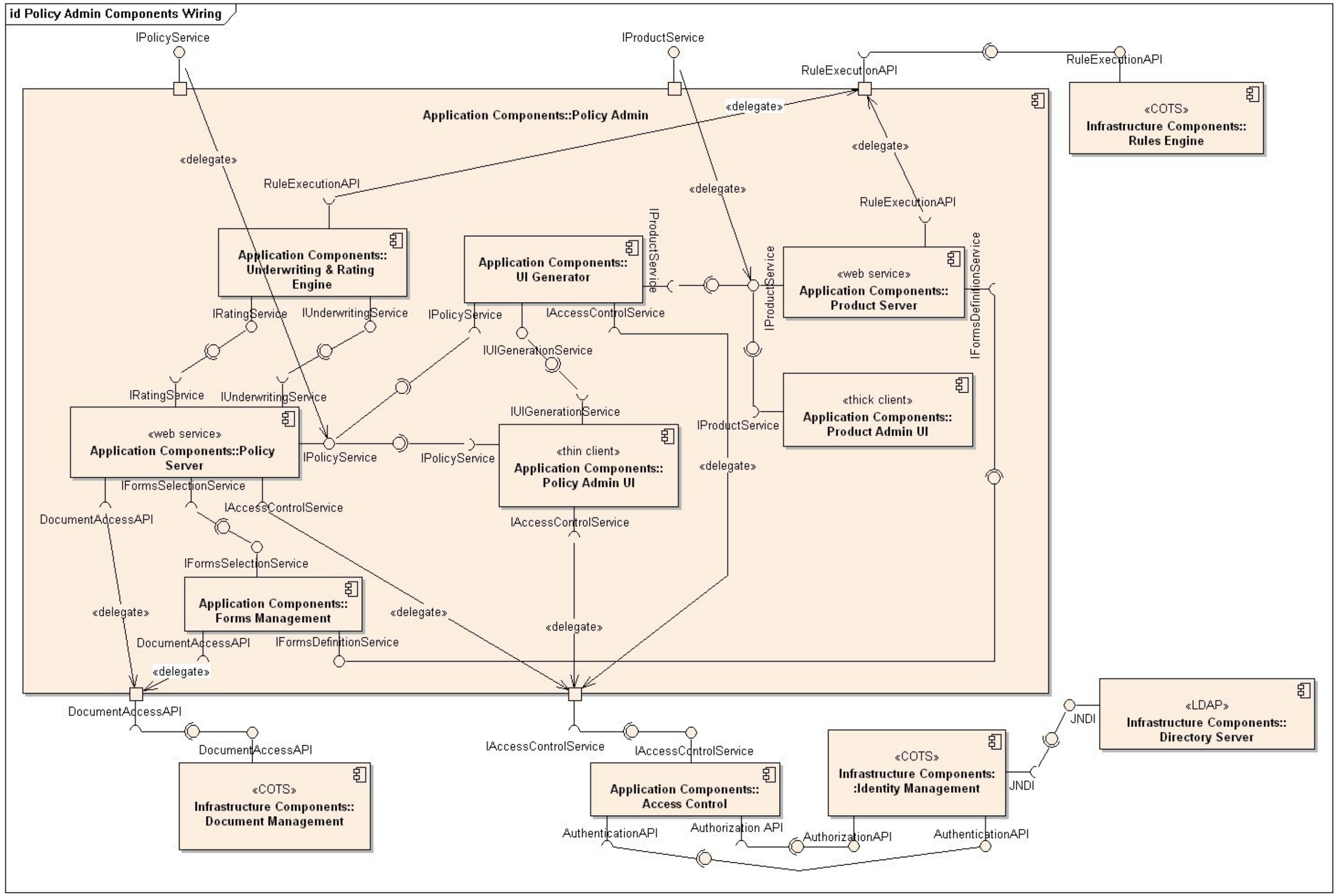
GameOverScreen

GameScreen
Start()
Update()
HandleLetterButtonRelease(theLetter:char)

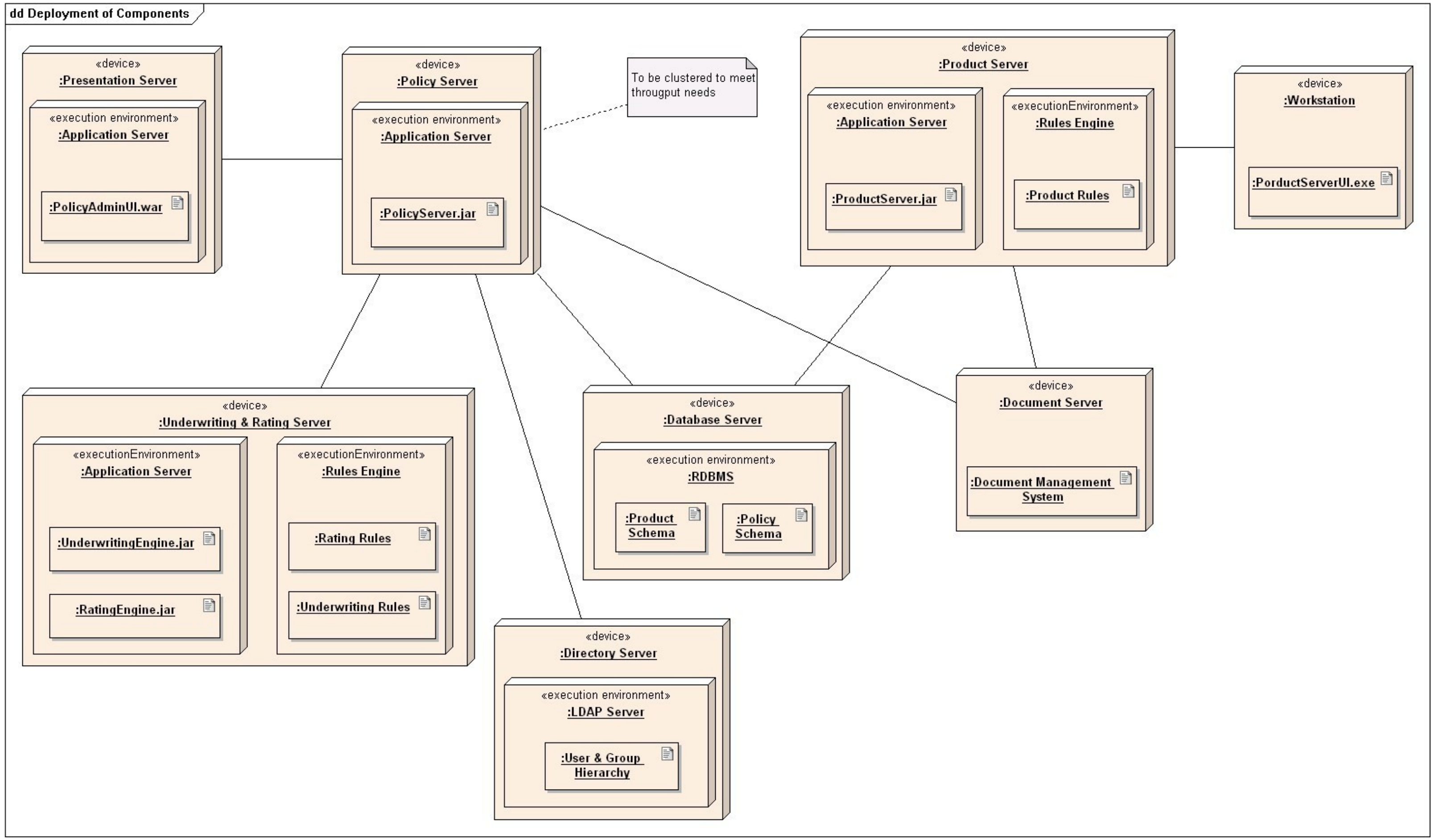
FButton

LetterButton
LetterButton(theLetter:char)
Show()
Hide()
<<get>> TheLetter()

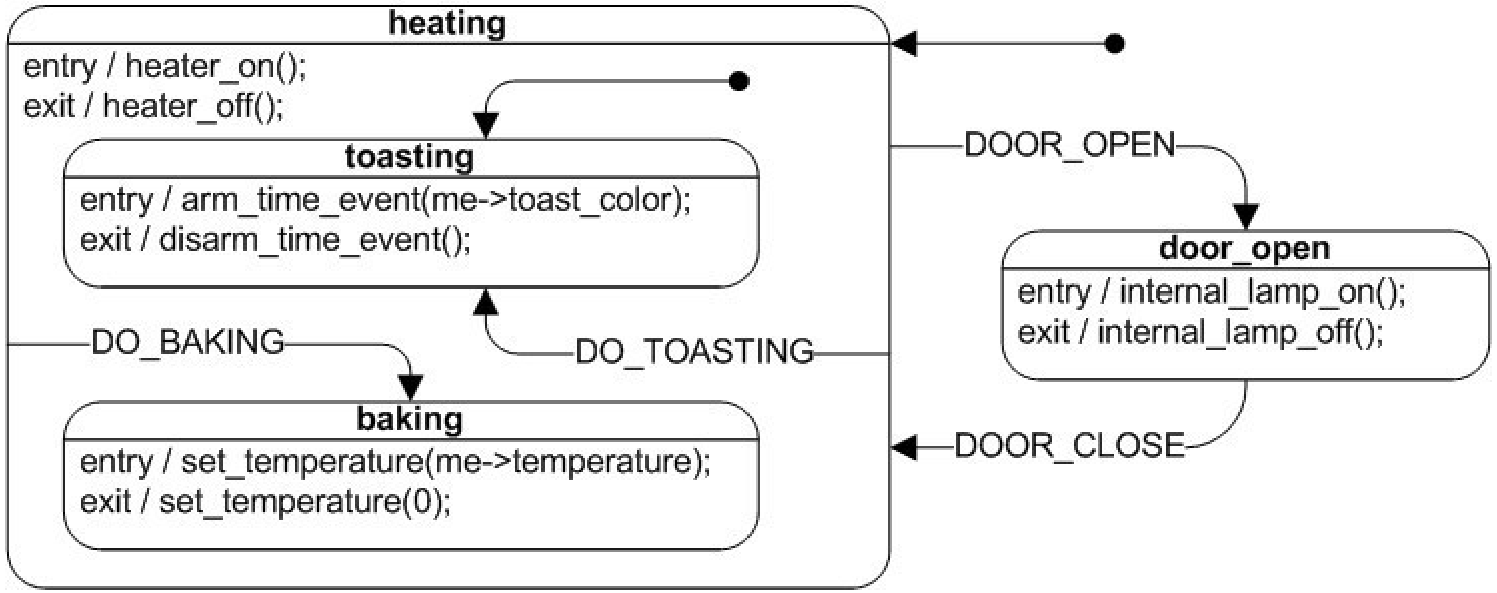
UML Component diagram



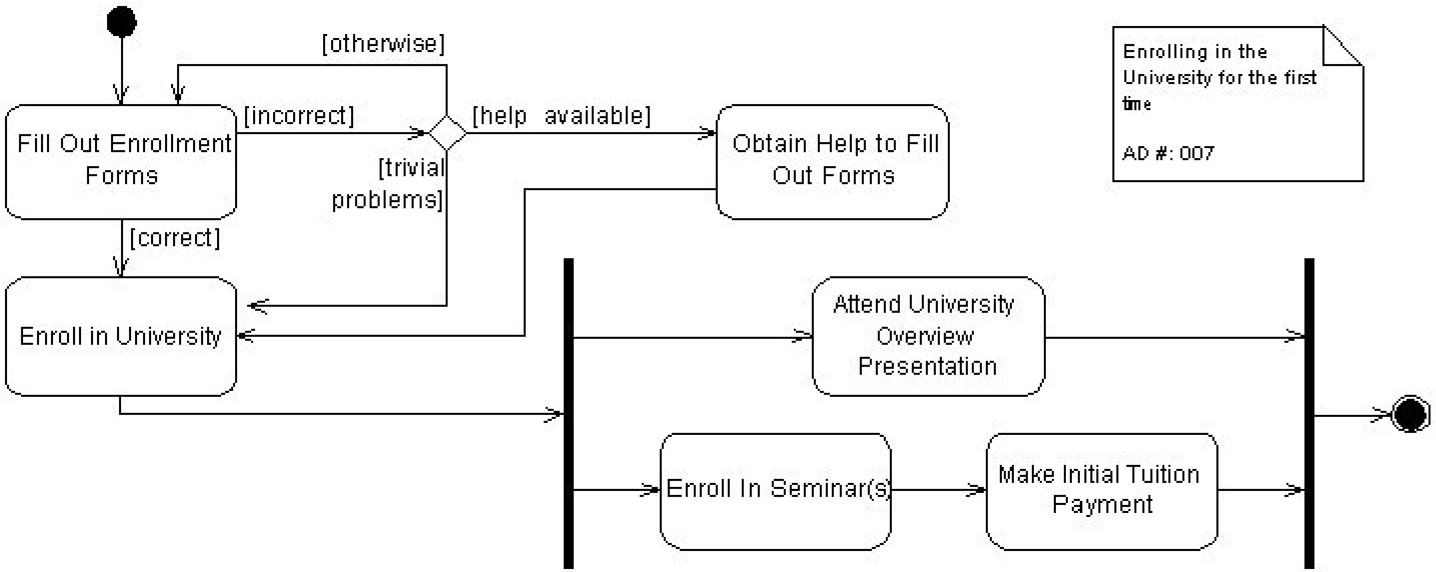
UML Deployment diagram



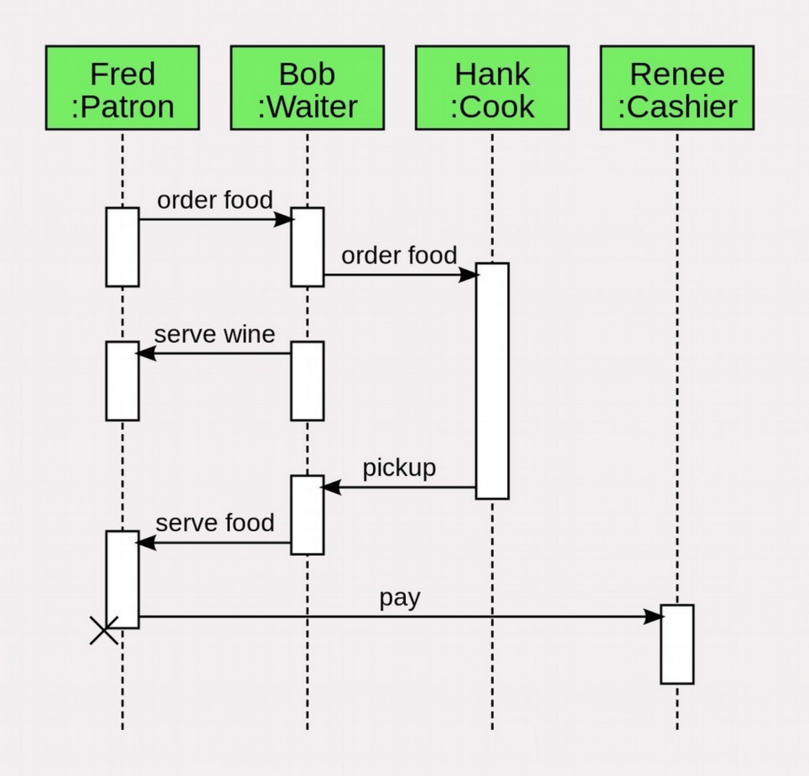
UML State Machine diagram


heating
entry / heater_on();
exit / heater_off();
toasting
entry / arm_time_event(me->toast_color);
exit / disarm_time_event();
DO_BAKING
DO_TOASTING
baking
entry / set_temperature(me->temperature);
exit / set_temperature(0);
DOOR_OPEN
door_open
entry / internal_lamp_on();
exit / internal_lamp_off();
DOOR_CLOSE

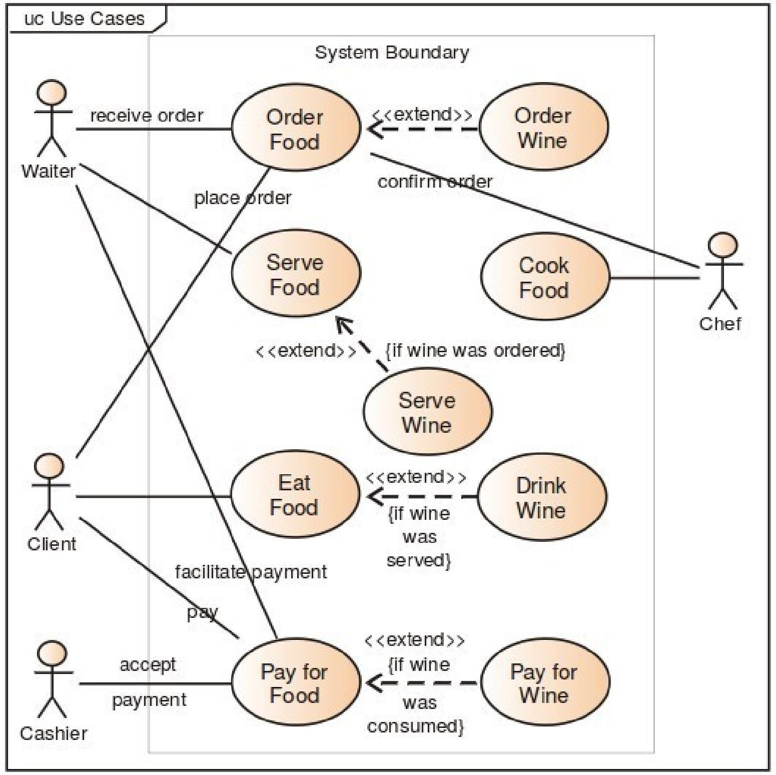
UML Activity diagram


[otherwise]
[incorrect]
[help available]
Fill Out Enrollment Forms
Obtain Help to Fill Out Forms
[trivial problems]
[correct]
Enroll in University
Attend University Overview Presentation
Enroll In Seminar(s)
Make Initial Tuition Payment
Enrolling in the University for the first time
AD #: 007

UML Sequence diagram



UML Use Case diagram



UML Criticism

- Good to visualize and present, but…
  - Nobody wants to program this way (creating diagrams)
  - Complex diagrams cannot be overseen
  - Simple diagrams are useless
  - Only program stub is generated
  - No round-trip editing

xtUML (eXecutable Unified Modeling Language)

- UML subsets (to make xtUML fully supported)
- Action Language
- Virtual Machine
- Testing, debugging (including state visualization), measurements are possible on original model without compilation
- Model Compilation
  - Into any language
  - On any platform
  - Possible optimization to target language / platform

MDD (Model-driven Development)

- Model-driven architecture design is useful for further development
- Model-driven Testing can be used independent of platform
- Model-driven Testing can give proof

Questions?

- ...
- Or write me an email to gla@inf.elte.hu

Dependency Injection

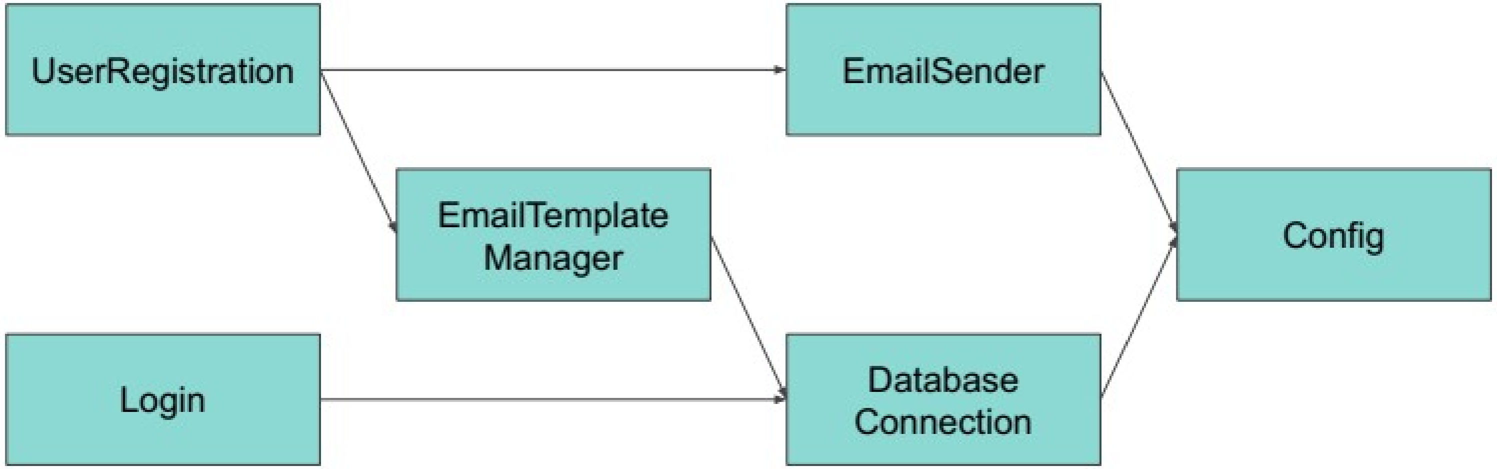
Class diagram

- During the semester, we work with the OOP and the imperative paradigm
- One of the basic principles of OOP is the class diagram, see:
  - https://en.wikipedia.org/wiki/Class_diagram
- Designing a program basically means to design the classes and the relations between the program operates with
- You basically define instance and class level relationships

Example
Who creates the Instances?

```
public void main(String[] args) {
   ConfigInterface config = new Config(“config.json”);
   EmailSenderInterface emailSender = new EmailSender();
   emailSender.setConfig(config);
   EmailTemplateInterface emailTemplate = new EmailTemplate();
   UserRegistratorInterface userRegistrator = new UserRegistrator();
   userRegistrator.setEmailSender(emailSender);
   userRegistrator.setEmailTemplateManager(emailTemplateManager);
   ...
}
```

Example on Class References



Manual Instantiation

- A lot of (centralized) code. Decentralization?
- Maybe a subtree is sufficient. No need for all the instances. Optimization?
- Redundancy (which should be eliminated in general, as it indicates a design flaw)
  - You define the new class
  - You define the new field that points to it
  - You create the instance
  - You set up a pointer
- Elimination of redundancy?

Dependency Injection

- Why not have a smart container which creates the instances on demand?
- The classes and the relationships are defined
- No instance is created manually
- The classes are annotated as injectables
- Some fields are annotated as inject targets
- The Context Dependency Injection framework creates the instances and also sets the references
  - See: https://en.wikipedia.org/wiki/Inversion_of_control

Annotations

- Classes annotated as **@Named** are to be created automatically
- Fields annotated with **@Inject** will be initialized automatically
- The method annotated with **@PostConstruct** is the initializer of the class. A constructor without parameters is necessary, so that the framework is able to create the instances. The parameters are passed as fields
- The context the instance is created in can be refined by the **@SessionScoped** and **@ApplicationScoped** (and similar) annotations
- You may take a look at:
  - https://en.wikipedia.org/wiki/Service_statelessness_principle

EmailService

```
@Named
public class EmailService implements EmailServiceInterface {
   public void sendEmail(String fromName, String fromEmail,
      String    recipientName, String recipientEmail, String subject,
      String html, String text) {

      <code that actually sends the email>

   }

   …
}
```

UserRegistration

```
@Named
public class UserRegistration implements UserRegistrationInterface {
    @Inject
    private EmailServiceInterface emailService;
    …
}
```

EntityManager

```
@Named
public class UserDao implements UserDaoInterface {
    @PersistenceContext(unitName = "<ProductName>EntityManager")
    protected EntityManager entityManager;
    protected CriteriaBuilder criteriaBuilder;
    @PostConstruct
    public void postConstruct() {
        criteriaBuilder = entityManager.getCriteriaBuilder();
    } …
}
```

ManagedExecutor

```
@Named
public class UserDao implements UserDaoInterface {
    // Do not create threads
    // Use the executor service instead
    @Resource(name = "DefaultManagedExecutorService")
    protected ManagedExecutorService executor;
    …
}
```

Scheduling

```
@Singleton
public class Worker {
   @Schedule(second = "*/10", minute = "*", hour = "*", persistent = false)
   public void run() {
   executor.submit(new FutureTask<Boolean>(new Callable<Boolean>() {
   @Override
   public Boolean call() throws Exception {
   // Do the job
   }
   }));
   }
}
```

Dependency Injection Dependency

- Advanced CDI features can be found in the API

```
<dependency>
<groupId>javax.enterprise</groupId>
<artifactId>cdi-api</artifactId>
<version>2.0-EDR1</version>
</dependency>
```

- In the case you would need to resolve an instance by code
- For example, the class / interface is **determined runtime**

Resolving Instances Runtime

- In the case you would need to resolve an instance by code
- For example, the class / interface is determined runtime

```
public static <T> T resolveByClass(Class<?> clazz)
{
  return (T) CDI.current().select(clazz).get();
}
```

Questions?

- ...
- Or write me an email to gla@inf.elte.hu