

Software Development (UP Elaboration Phase)

Marek Rychlý

`rychly@fit.vutbr.cz`

Brno University of Technology
Faculty of Information Technology
Department of Information Systems

Information Systems Analysis and Design (AIS)
5 December 2019



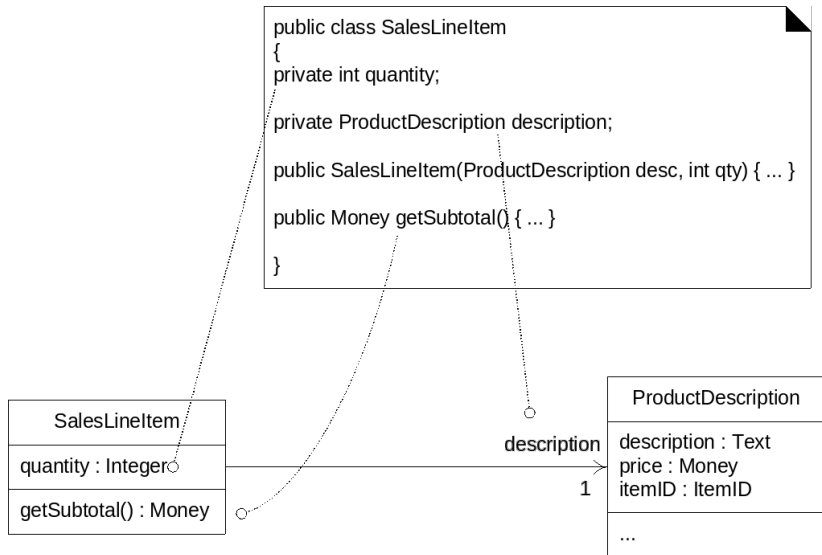
Outline

- 1 Software Development
 - From Design to Implementation
 - Test-Driven Development
 - Code Refactoring
- 2 Next Iterations is the UP Elaboration Phase
 - The 2nd Iteration
 - The 3rd Iteration
- 3 An Example: The Monopoly Game System

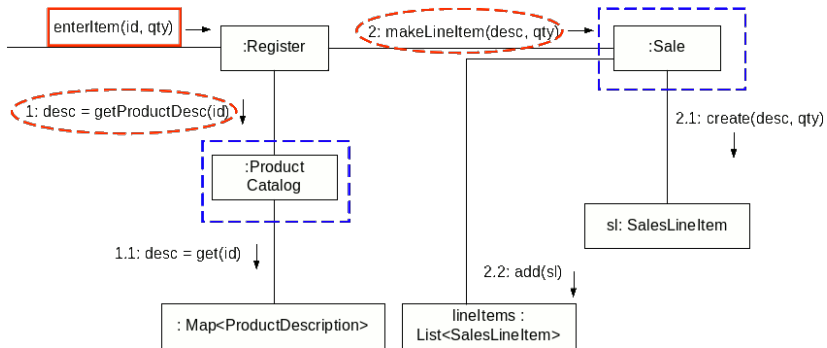
Software Design and Development

- To map design models to implementation models.
(e.g., class&interaction diagrams to source code, SQL scripts, GUI artefacts, etc.)
- It is not just a transformation of a design to implementation model.
(as it would be in the case of Model Driven Development/Architecture)
- Usually, there will be re-design by/during development/coding.
(due to technical problems and their solutions, integration issues, etc.)
- So the resulting design after/during the development may significantly differ from the original pre-development design.
- That is OK – design is just to outline/understand, not to document.
(the documentation will be generated from the code later, by re-engineering tools)

Design Classes and their Source Code



Implementation of Methods from Interaction Diagrams



(adopted from "Applying UML and Patterns" by Craig Larman)

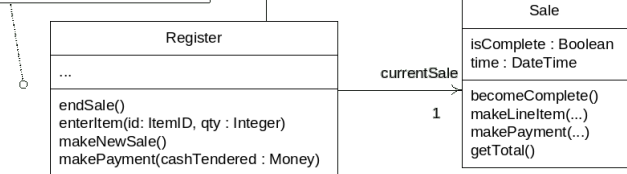
An Example: Register Class

```
public class Register
{
    private ProductCatalog catalog;
    private Sale currentSale;

    public Register(ProductCatalog pc) {...}

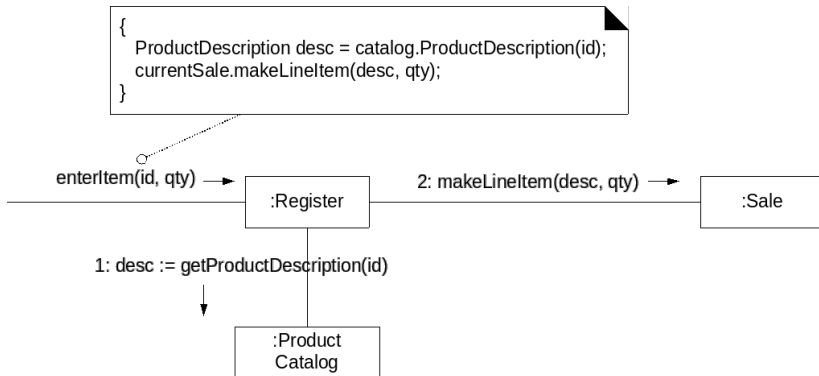
```

```
    public void endSale() {...}
    public void enterItem(ItemID id, int qty) {...}
    public void makeNewSale() {...}
    public void makePayment(Money cashTendered) {...}
}
```



(adopted from "Applying UML and Patterns" by Craig Larman)

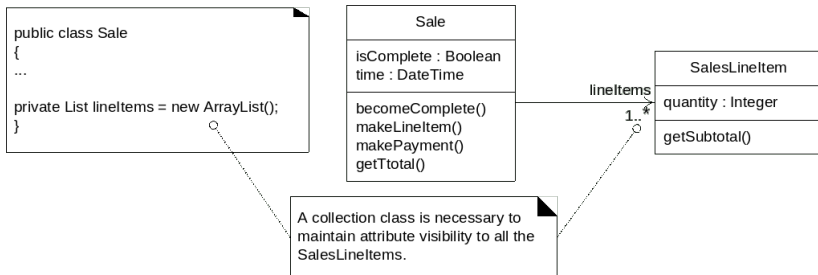
The Example: Register.enterItem Method



(adopted from "Applying UML and Patterns" by Craig Larman)

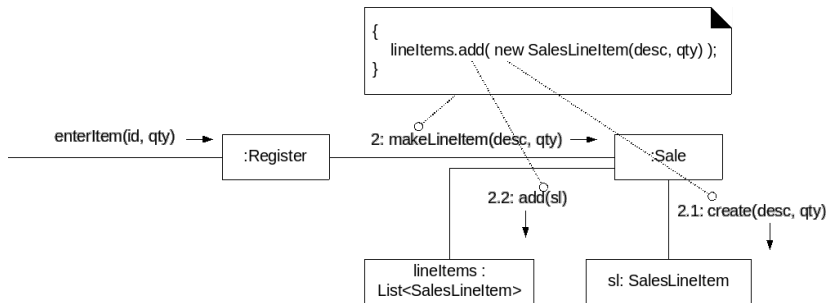
Collections Of Objects of Associated Classes

- The collections implement associations with *-cardinality.
- There are different implementations of several types of collections. (arrays, lists, maps, sets, etc.; as ordered, linked, hashed, mutable/immutable, etc.)
- If possible, use a generic implementation-independent interface. (e.g., use “List” instead of “ArrayList”, etc.)



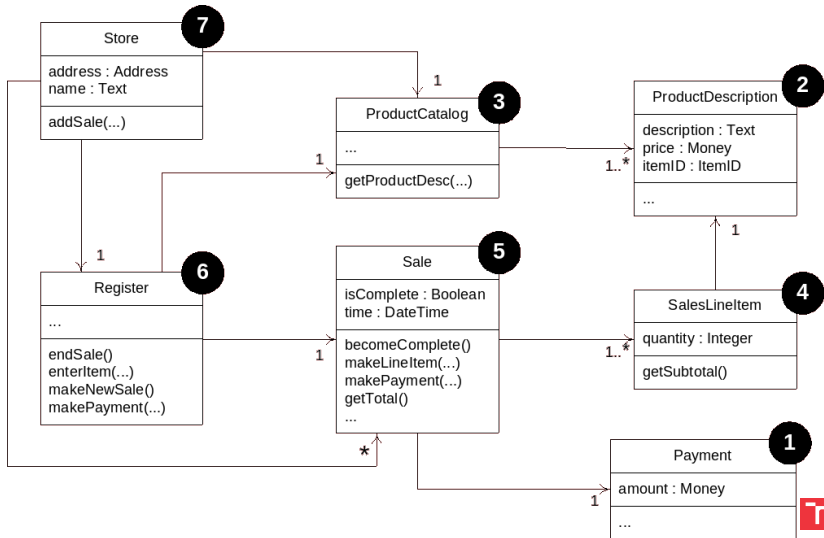
(adopted from “Applying UML and Patterns” by Craig Larman)

An Example: Sale.makeLineItem Method




(adopted from “Applying UML and Patterns” by Craig Larman)

Implementation Order By Dependencies



(adopted from "Applying UML and Patterns" by Craig Larman)

Test-Driven Development (TDD)

- Well-established in iterative/incremental processes.
(also in Unified Process and Agile dev.; to maintain a constant “cost of change”)
- Requirements are turned into very specific automated test cases.
(also known as “test-first development”; the acceptance tests)
- A SW unit is implemented/improved just to pass its unit tests.
(the tests pass when the implementation is according to the requirements)
- The unit is a black-box with well-defined interface and behaviour.
(so the interface and behaviour are stabilised before further development)
- The unit should be as small as possible.
(both by its definition and also in its implementation)
 - Its tests are easier to write, read, and to understand.
 - Test failures are easier to detect, tracking down errors, and fix.
- There are several frameworks for automated unit testing in TDD. 
(JUnit, NUnit, CppUnit; also languages with built-in unit testing support, e.g., Go)

An Example: Sale and SaleTest

Before Sale class implementation, we need to implement its unit test.

- 1 The unit test of Sale will create its “fixture” instance,
- 2 add several sale items by calling the makeLineItem of the Sale,
(that is the method we are testing)
- 3 and check the resulting total amount.
(by getTotal and assertTrue methods of the Sale class and a unit test framework)

Then, we are ready to implement the Sale to pass the test.
(just barely good enough to pass, we can refactor later when necessary)

The Example: SaleTest Class (1)

```
public class SaleTest extends TestCase
{
    // ...

    // test the Sale.makeLineItem method
    public void testMakeLineItem()
    {
        // STEP 1: CREATE THE FIXTURE

        // -this is the object to test
        // -it is an idiom to name it 'fixture'
        // -it is often defined as an instance field rather than
        // a local variable
        Sale fixture = new Sale();

        // set up supporting objects for the test
        Money total = new Money( 7.5 );
        Money price = new Money( 2.5 );
        ItemID id = new ItemID( 1 );
        ProductDescription desc =
            new ProductDescription( id, price, "product 1" );
    }
}
```

(adopted from "Applying UML and Patterns" by Craig Larman)



The Example: SaleTest Class (2)

```
// STEP 2: EXECUTE THE METHOD TO TEST
```

```
// NOTE: We write this code **imagining** there  
// is a makeLineItem method. This act of imagination  
// as we write the test tends to improve or clarify  
// our understanding of the detailed interface to  
// to the object. Thus TDD has the side-benefit of  
// clarifying the detailed object design.
```

```
// test makeLineItem  
sale.makeLineItem( desc, 1 );  
sale.makeLineItem( desc, 2 );
```

(adopted from “Applying UML and Patterns” by Craig Larman)



The Example: SaleTest Class (3)

```
// STEP 3: EVALUATE THE RESULTS

// there could be many assertTrue statements
// for a complex evaluation

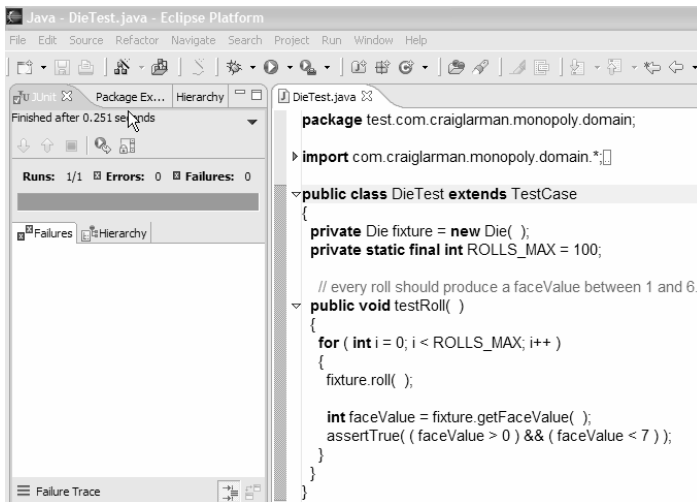
// verify the total is 7.5
assertTrue( sale.getTotal().equals( total ));
}
}
```

(adopted from "Applying UML and Patterns" by Craig Larman)

Development with Unit Tests

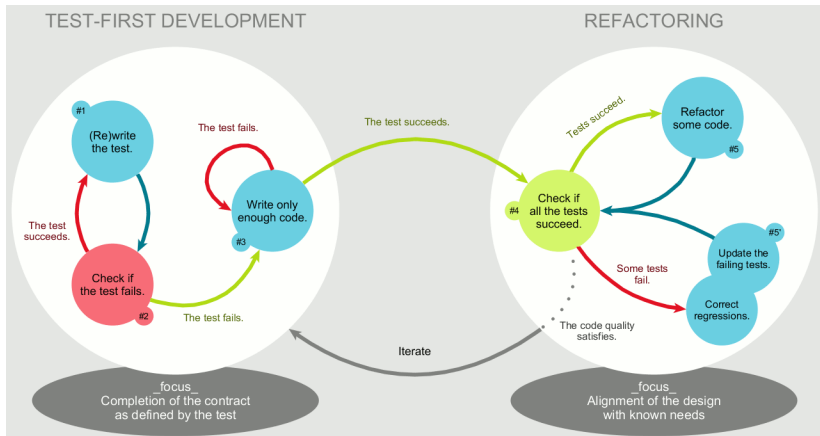
- ❶ Create a unit test class for a particular tested class.
(e.g., by extending a unit test framework class)
- ❷ In the unit test class, create a “fixture” instance of the tested class.
(the fixture is a test-specific class where methods typically represent tests to run)
- ❸ Repeat until the tested class is finished (has all required methods):
 - ❶ Create a test method for a particular tested method.
 - ❷ In the test method, call the tested method and check its results.
 - ❸ Finally, (re-)implemented the tested method until pass the test.
- ❹ Later, when refactoring the tested class, run test class methods.
(to rerun the tests and check possible regressions in interface or behaviour)

JUnit in Eclipse IDE



(adopted from "Applying UML and Patterns" by Craig Larman)

Test-First Development and Code Refactoring



(adopted from "TDD Global Lifecycle" by Wikipedia)

Code Refactoring

- The process of restructuring existing source code without changing its external behaviour.
(it does not change functionality, just non-functional attributes of the software)
- An iterative cycle of making a small program transformation, e.g.,
(to improve code structure, readability, complexity, to improve SW maintainability)
 - to remove duplicate code, to apply design patterns,
 - to shorten/rearrange long and complicated parts of the code,
 - to remove/extract “magic numbers”, etc.
- Each transformation is tested to ensure its correctness.
(to check that it did not change the external behaviour of the code)
- Continuous refactoring is common in XP and other agile/iterative development processes.



Code Smells (1)

- Duplicated code or Contrived complexity/Over-engineering
- Large class/God object, Lazy class/freeloader
(a class that does too much with a low cohesion, or does too little)
- Feature envy, Inappropriate intimacy
(an excessive usage of methods of another class or its implementation details)
- Refused bequest
(a class overriding a method of a base class so that the contract is not fulfilled)
- Excessive use of literals, Constant class
(magic literals that should be constants, or constants that should belong elsewhere)
- Cyclomatic complexity, Down-casting
(too many branches or loops, or a type cast which breaks the abstraction model)
- Data clump
(a group of variables often passed around together should form an object together)

Code Smells (2)

- Too many parameters in a method
(a long list of method parameters, hard to read, call, and test)
- Excessive return of data from a method
(a method returns more than what each of its callers needs)
- Excessively long method, Long line of code (God method/line)
(a method/line that has grown too large, low cohesion)
- Excessively short/long identifiers
(e.g., for variables, method names, class names, etc.)

And many others poor design/development practices.
(they will be discussed later in a lecture on anti-patterns)

Refactoring Transformations

- **Extract Method**

(You have a code fragment that can be grouped together. Turn the fragment into a method whose name explains the purpose of the method.)

- **Replace Magic Number with Symbolic Constant**

(You have a literal number with a particular meaning. Create a constant, name it after the meaning, and replace the number with it.)

- **Extract Variable**

(You have a complicated expression. Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.)

- **Replace Constructor with Factory Method**

(You want to do more than simple construction when you create an object. Replace the constructor with a factory method.)

There is a catalogue of the transformations on

<https://refactoring.com/catalog/>.



Extract Method

You have a code fragment that can be grouped together.

```
void printOwing() {  
    printBanner();  
    // print details  
    System.out.println("name:_" + _name);  
    System.out.println("amount_" + getOutstanding());  
}
```

Turn the fragment into a method whose name explains the purpose of the method.

```
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}  
  
void printDetails(double outstanding) {  
    System.out.println("name:_" + _name);  
    System.out.println("amount_" + outstanding);  
}
```

Replace Magic Number with Symbolic Constant

You have a literal number with a particular meaning.

```
double potentialEnergy(double mass, double height) {  
    return mass * height * 9.81;  
}
```

Create a constant, name it after the meaning, and replace the number with it.

```
static final double GRAVITATIONAL_CONSTANT = 9.81;  
double potentialEnergy(double mass, double height) {  
    return mass * GRAVITATIONAL_CONSTANT * height;  
}
```


Extract Variable

You have a complicated expression.

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&  
    (browser.toUpperCase().indexOf("IE") > -1) &&  
    wasInitialized() && (resize > 0)) { /* do something */ }
```

Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

```
final boolean isMacOs  
    = platform.toUpperCase().indexOf("MAC") > -1;  
final boolean isIEBrowser  
    = browser.toUpperCase().indexOf("IE") > -1;  
final boolean wasResized  
    = resize > 0;  
  
if (isMacOs && isIEBrowser && wasInitialized()  
    && wasResized) { /* do something */ }
```



Replace Constructor with Factory Method

You want to do more than simple construction when you create an object.

```
public Employee(int type) {  
    _type = type;  
}
```

Replace the constructor with a factory method.

```
private Employee(int type) {  
    _type = type;  
}  
public static Employee create(int type) {  
    return new Employee(type);  
}
```

Why Should You Refactor?

according to “Refactoring: Improving the Design of Existing Code” by Martin Fowler

- Refactoring improves the design of software.
(without refactoring, the design of the program will decay)
- Refactoring makes software easier to understand.
(refactor the code to better reflect your understanding)
- Refactoring helps you find bugs.
(understanding the code also help to spot bugs; “I’m not a great programmer; I’m just a good programmer with great habits” by Kent Beck)
- Refactoring helps you program faster.
(a good design is essential for rapid software development)

When Should You Refactor?

according to “Refactoring: Improving the Design of Existing Code” by Martin Fowler

- The Rule of Three.
(the first time you do something, you just do it; the second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway; the third time you do something similar, you refactor)
- Refactor when you add function.
(if a design does not help to add a feature easily; to understand a code to modify)
- Refactor when you need to fix a bug.
(because the code was not clear enough to see there was a bug)
- Refactor as you do a code review¹
(to help to review someone else's code; refactoring in active code reviews/XP)

¹Kent Beck's Two Hats: divide your time between two distinct activities: adding function/coding and review/refactoring.

Best Practices in Refactoring

according to “Refactoring: Improving the Design of Existing Code” by Martin Fowler

- Firstly, build a solid set of automatic tests for code to refactor.
(refactoring must not introduce any bugs, the change needs to be inexpensive)
- Backtrack – refactoring changes the programs in small steps.
(if you make a mistake, it is easy to find the bug)
- Get used to picking a goal.
(somewhere your code smells bad, resolve to get rid of the problem)
- Stop when you are unsure.
(if the code is already better, stop and release; throw away the changes otherwise)
- Work in pairs, refactor with someone.
(he/she see things you do not see and know things you do not know)



Problems with Refactoring

according to “Refactoring: Improving the Design of Existing Code” by Martin Fowler

- Databases are difficult to change.
(refactoring persistent classes may affect a db. schema and cause data migration;
use an abstraction to implement a db. layer, the implementation can be unstable)
- Changing interfaces may not be possible.
(a refactoring may change an interface, which is not possible for stable/published
interfaces; do not publish interfaces prematurely)
- Design changes can be difficult to refactor.
(if there wont be a simple way to refactor, you must put more effort into the design)

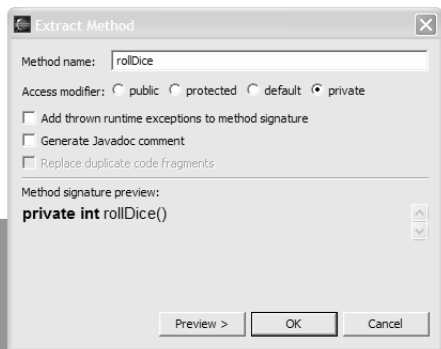
Code Refactoring in Eclipse IDE (1)

```
public Player(String name, Die[] dice, Board board)
{
    this.name = name;
    this.dice = dice;
    this.board = board;
    piece = new Piece(board.getStartSquare());
}
```

```
public void takeTurn()
```

```
{
    // roll dice
    int rollTotal = 0;
    for (int i = 0; i < dice.length; i++)
    {
        dice[i].roll();
        rollTotal += dice[i].getFaceValue();
    }
}
```

```
Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
piece.setLocation(newLoc);
}
```



(adopted from “Applying UML and Patterns” by Craig Larman)

Code Refactoring in Eclipse IDE (2)

```
▼ public void takeTurn()
{
    int rollTotal = rollDice();

    Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
    piece.setLocation(newLoc);
}

▼ private int rollDice()
{
    // roll dice
    int rollTotal = 0;
    for (int i = 0; i < dice.length; i++)
    {
        dice[i].roll();
        rollTotal += dice[i].getFaceValue();
    }
    return rollTotal;
}
```

(adopted from "Applying UML and Patterns" by Craig Larman)

Forward, Reverse, and Round-Trip Engineering

To switch between code and design models, usually in an IDE.

- Forward – code generation from design models.
(usually, the code is just a skeleton and must be completed later)
- Reverse – a design model generation from the code.
(useful to better understanding of the code)
- Round-Trip – both the model and the code synchronisation.
(loose-less transformations of the code to design models and back)

By the End of the 1st Iteration

- Tests of the first iteration software are finished.
(unit, integration, acceptance, load, etc. tests; UP requires early, realistic, and continuous quality checks and validation tests)
- Stakeholders are participating in the evaluation.
(to get feedback, to enable progress monitoring, etc.)
- The resulting system is fully integrated and stable.
(the iteration resulted into a baseline, i.e., the first internal version of the systems)

Transition to the 2nd Iteration

- There will be a planning meeting for the next iteration.
- The 1st iteration results will be reverse-engineered to diagrams.
(these diagrams will be further utilised in the design for the next iteration)
- There is ongoing
 - a feasibility study,
(we need to check the feasibility continuously, to manage emerging risks)
 - development of a user interface,
(we need to design the UI according to the incoming requirements)
 - modelling and implementation of a database.
(we need to update a database schema according to the newest features)
- There will be another requirement workshops.
(to analyse and specify use cases addressed in this iteration)
 - Both new reqs. and passing reqs. from the previous iteration.
(developers from the previous iteration need to participate)
 - We need 80% done and 10% implemented in the elaboration phase.

The 2nd Iteration in the NexGen POS Example

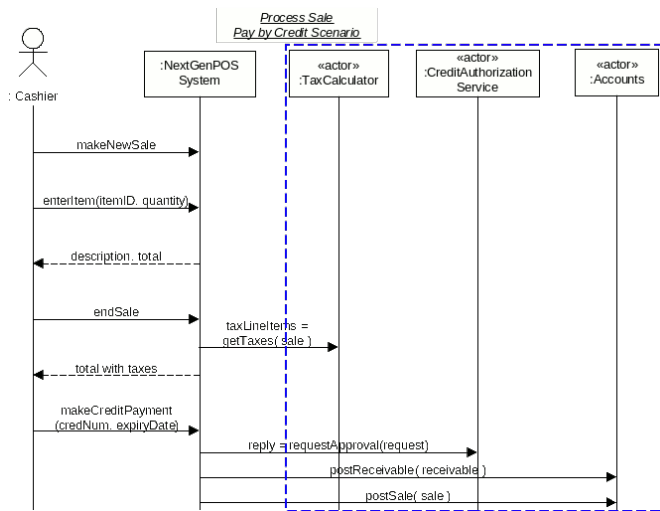
- Focus on requirements related to the Process Sale use case.
 - Connection to various external services.
(tax calculators, accounting systems, payment services, etc.)
 - Rules for the price calculation (discounts, etc.).
 - The update of a total price display on a new sale item.
- The assigning of these requirements to this iteration is for demonstration purposes only. In a real-world case study, we should **continue with the highest impact/risk use cases**.
- The Process Sale use case can be implemented
 - in one iteration,
(that is all its scenarios in the single 2nd iteration)
 - or in multiple iterations by individual scenarios.
(anyway, each scenario must be in the exactly one iteration, it cannot be split)



Process Sale in the 2nd Iteration

- The use case specification has been finished in the 1st iteration.
- Now, in the 2nd iteration, it will be revised and extended.
- The system sequence diagram (SSD) will be extended.
(there will be new system/internal operations with another objects)
- The domain model will be particularised.
(missing operations will be defined, parameters will be revised, etc.)

Process Sale SSD in the 2nd Iteration



(adopted from "Applying UML and Patterns" by Craig Larman)

GRASP and Design Patterns in the 2nd Iteration

To design/implement the Process Sale use-case requirements.

- Connection to various external services.
(e.g., Adapter, Factory, and Abstract Factory design patterns)
- Rules for the price calculation (discounts, etc.).
(e.g., Strategy and Composite design patterns)
- The update of a total price display on a new sale item.
(e.g., Observer and Publish-Subscribe design patterns)

For the examples, see the respective principles/patterns in the previous lectures.

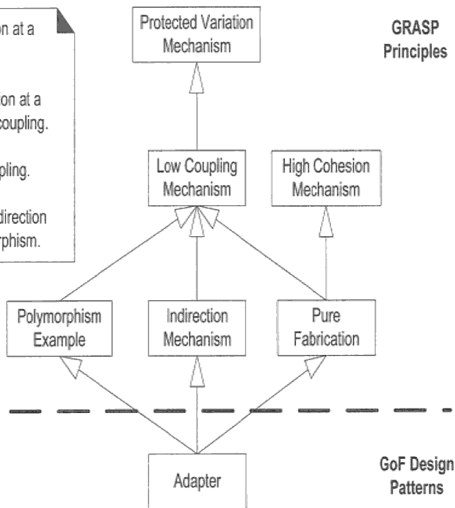
GRASP and GoF Adapter Design Pattern

Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.



(adopted from "Applying UML and Patterns" by Craig Larman)

The 3rd Iteration in the NexGen POS Example

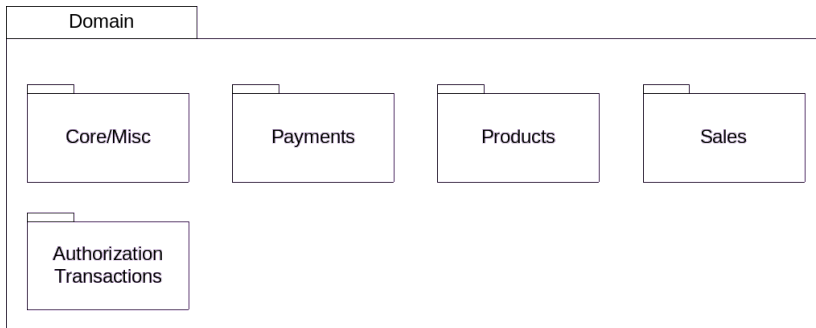
- Another requirements related to the Process Sale use case.
 - The robustness of local services.
(their availability on connection breaks/unavailability of remote services; e.g., by utilising local cached object when trying to access a remote database)
 - An integration of POS hardware connected to a register.
(e.g., a cash drawer)
 - A card payment authorisation.
 - Persistence of (domain) objects.
- To address these requirements, we will revise/create artefacts:
 - an object persistence framework,
(developed in-house or acquired from a third-party)
 - the system architecture documentation,
 - UML models of participating processes,
 - generalisation and specialisation of existing classes,
 - design of new packages, revision of the existing packages.



Packages and Domain Model Objects

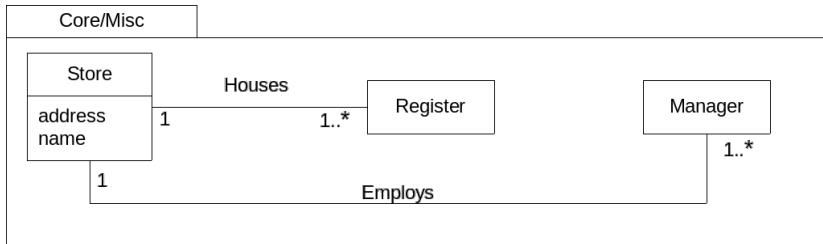
- A growing domain model is useful to split into several packages.
- The splitting should respect a low coupling and high cohesion.
- The low coupling and high cohesion means the encapsulation of
 - classes of the same concept or responsibilities,
 - in the same class hierarchy,
 - participating in the same use cases,
 - with a high number of associations between them.

The NextGen POS: Domain Model in Packages



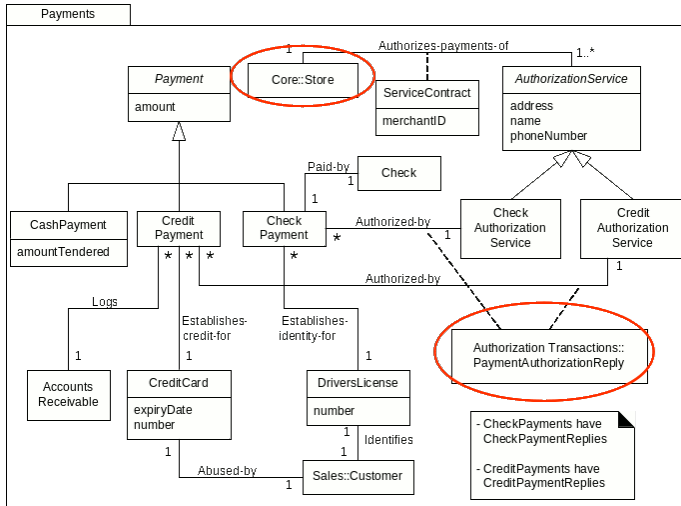
(adopted from "Applying UML and Patterns" by Craig Larman)

The NextGen POS: Core Package



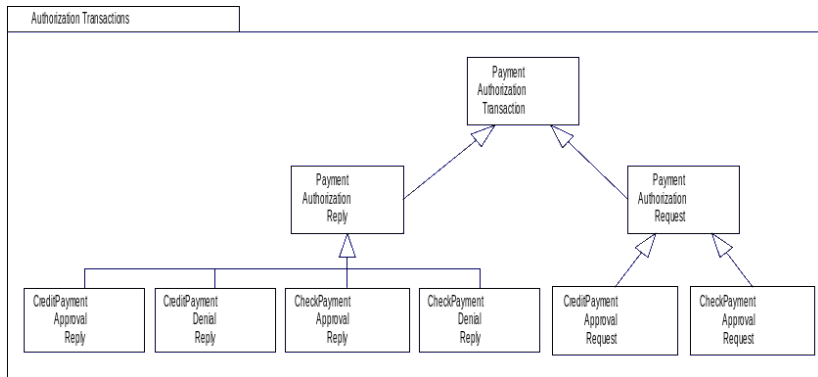
(adopted from "Applying UML and Patterns" by Craig Larman)

The NextGen POS: Payments Package



(adopted from "Applying UML and Patterns" by Craig Larman)

The NextGen POS: Authorisation Transaction Package



(adopted from "Applying UML and Patterns" by Craig Larman)

Architecture Design

- Affected by important requirements, both non- and functional.
(such requirements are analysed first, in the beginning of the Elaboration phase)
- The analysis and design of the architecture should start ASAP.
(even before the 1st iteration, before the development of a baseline starts)
- An architect should identify architectural patterns.
(from the requirements analysis, as described in a supplementary specification)
- If there are any high-impact factors, possible alternatives of the architecture need to be analysed and the factors need to be addressed in the resulting architecture.

An Example of Factors of the Architecture Design (1)

Factor	Measures and quality scenarios	Variability (current flexibility and future evolution)	Impact of factor (and its variability) on stakeholders, architecture and other factors	Priority for Success	Difficulty or Risk
Reliability—Recoverability					
Recovery from remote service failure	When a remote service fails, reestablish connectivity with it within 1 minute of its detected re-availability, under normal store load in a production environment.	current flexibility - our SME says local client-side simplified services are acceptable (and desirable) until reconnection is possible. evolution - within 2 years, some retailers may be willing to pay for full local replication of remote services (such as the tax calculator). Probability? High.	High impact on the large-scale design. Retailers really dislike it when remote services fail, as it prevents them from using a POS to make sales.	H	M
Recovery from remote product database failure	as above	current flexibility - our SME says local client-side use of cached “most common” product info is acceptable (and desirable) until reconnection is possible. evolution - within 3 years, client-side mass storage and replication solutions will be cheap and effective, allowing permanent complete replication and thus local usage. Probability? High.	as above	H	M
Supportability - Adaptability					

(adopted from “Applying UML and Patterns” by Craig Larman)

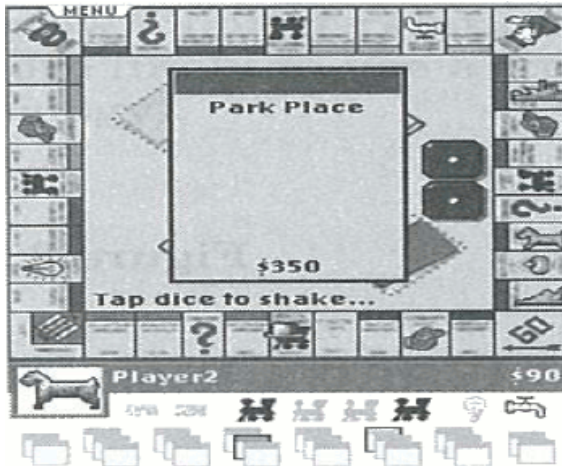
An Example of Factors of the Architecture Design (2)

Factor	Measures and quality scenarios	Variability (current flexibility and future evolution)	Impact of factor (and its variability) on stakeholders, architecture and other factors	Priority for Success	Difficulty or Risk
Support many third-party services (tax calculator, inventory, HR, accounting). They will vary at each installation.	When a new third-party system must be integrated, it can be, and within 10 person days of effort.	current flexibility - as described by factor evolution - none	Required for product acceptance. Small impact on design.	H	L
Support wireless PDA terminals for the POS client?	When support is added, it does not require a change to the design of the non-UI layers of the architecture.	current flexibility - not required at present evolution - within 3 years, we think the probability is very high that wireless "PDA" POS clients will be desired by the market.	High design impact in terms of protected variation from many elements. For example, the operating systems and UIs are different on small devices.	L	H
Other - Legal					
Current tax rules must be applied.	When the auditor evaluates conformance, 100% conformance will be found. When tax rules change, they will be operational within the period allowed by government.	current flexibility - conformance is inflexible, but tax rules can change almost weekly because of the many rules and levels of government taxation (national, state, ...) evolution - none	Failure to comply is a criminal offense. Impacts tax calculation services. Difficult to write our own service--complex rules, constant change, need to track all levels of government. But, easy/low risk if buy a package.	H	L

(adopted from "Applying UML and Patterns" by Craig Larman)

An Example: The Monopoly Game System

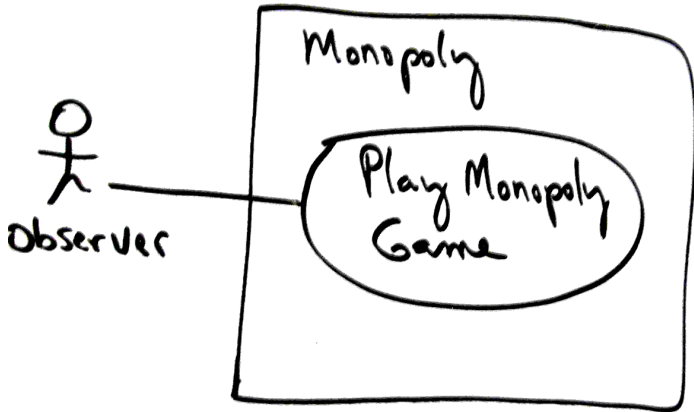
It is just a a computer simulation simply watched (not played) by a user.



(adopted from "Applying UML and Patterns" by Craig Larman)

The Inception Phase in the Monopoly Game

The only significant use case is Play Monopoly Game.
(even if it does not pass the Boss Test, the user is an observer, not a player)



(adopted from "Applying UML and Patterns" by Craig Larman)

The Monopoly Play Game Use Case

Use Case UC1: Play Monopoly Game

Scope: Monopoly application

Level: user goal

Primary Actor: Observer

Stakeholders and Interests:

– Observer: Wants to easily observe the output of the game simulation.

Main Success Scenario:

1. Observer requests new game initialization, enters number of players.
2. Observer starts play.

3. System displays game trace for next player move (see domain rules, and “game trace” in glossary for trace details).

Repeat step 3 until a winner or Observer cancels.

Extensions:

*a. At any time, System fails:

(To support recovery, System logs after each completed move)

1. Observer restarts System.
2. System detects prior failure, reconstructs state, and prompts to continue.
3. Observer chooses to continue (from last completed player turn).

Special Requirements:

– Provide both graphical and text trace modes.

(adopted from “Applying UML and Patterns” by Craig Larman)

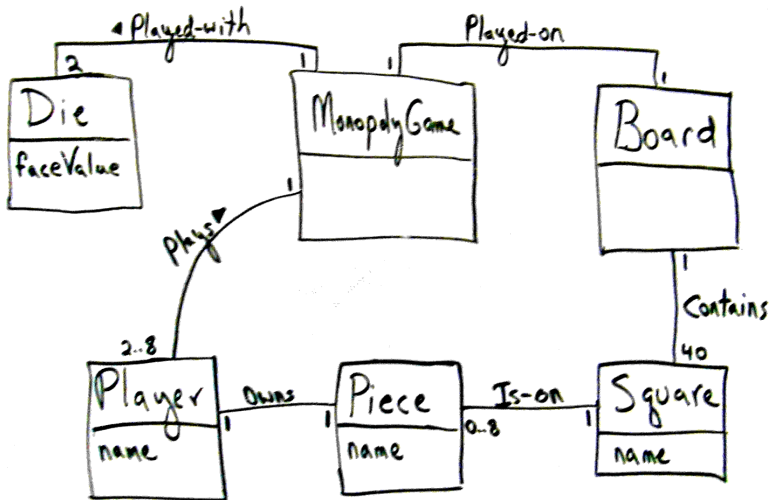
The Elaboration Phase in the Monopoly Game

The requirements specification/analysis:

- Implementation of the use case Play Monopoly Game: players are moving pieces around a gaming board.
- Implementation of the initialisation use case Start Up.
- The game simulated for 2 to 8 players, this is the only input parameter needed from a user/observer to start the game.
- Players are simulated to take turns:
each rolls two dice, moves around the board the total number of squares.
(the game is limited to 20 rounds in the 1st iteration²)
- The systems shows a name of a player in turn.
- Each square has a name – there is “Go” square where all pieces start and “Square1” to “Square39” squares.

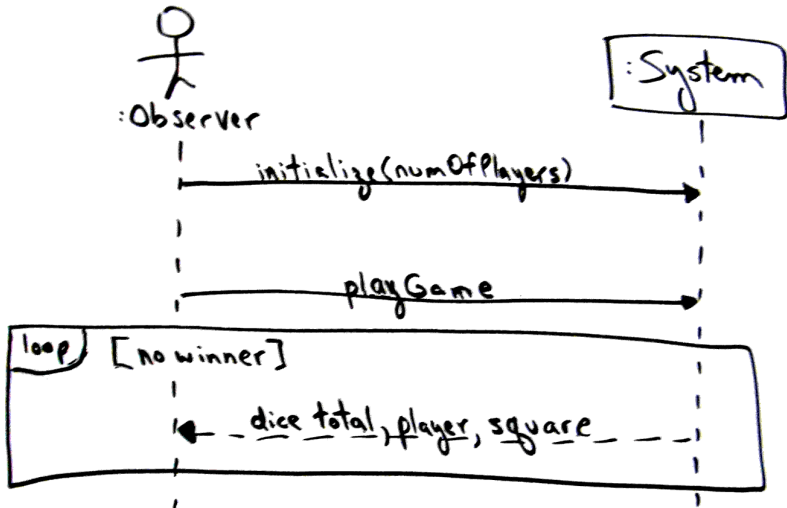
²there are also no winner/looser, no money, no opportunities to buy/rent, and no special squares in the 1st iteration

Domain Model of the Monopoly Game



(adopted from "Applying UML and Patterns" by Craig Larman)

System Sequence Diagram of the Monopoly Game



(adopted from "Applying UML and Patterns" by Craig Larman)

Object-Oriented Design: Controller

According to the GRASP Controller principle, there are two options.

1a a controller for an overall system:

`MonopolyGameSystem`,

(suitable for an embedded system where the controller handle UI events)

1b a controller as a root object:

`MonopolyGame`,

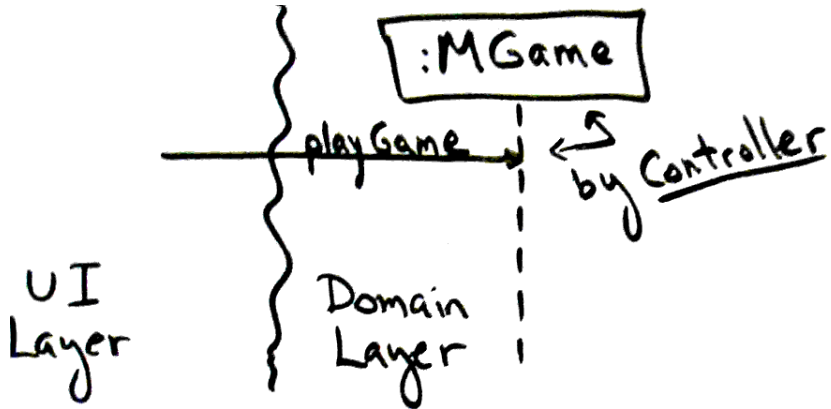
(suitable for a central implementation of various system operations)

2 a controller for the scenario/a session:

`PlayMonopolyGameHandler`, `PlayMonopolyGameSession`.

(suitable for handling a use case/scenario messages/operations)

Controller in the Monopoly Game



(adopted from "Applying UML and Patterns" by Craig Larman)

The Game-Loop Algorithm

(adopted from “Monopoly Use Case” by Alex Thomo)

- A domain dictionary contains the following terms:
 - Turn** is when a player rolling the dice and moving the piece.
 - Round** is when all the players taking one turn in the game.
- The system implements the following game-loop algorithm:

```
def playGame(numberOfRounds):  
    for n in range(0, numberOfRounds):  
        playRound()  
  
def playRound():  
    for p in players:  
        p.takeTurn()
```

Controlling the Game Loop in the Monopoly Game

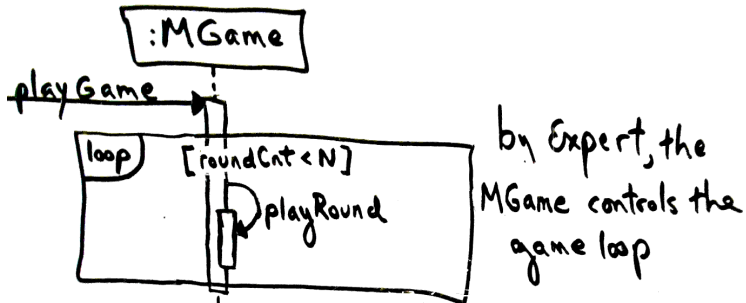
The responsibility for the game loop – according to the GRASP Information Expert principle, it should be done by object(s) having required information.

Information Needed	Who Has the Information?
the current round count	No object has it yet, but by LRG, assigning this to the <i>MonopolyGame</i> object is justifiable.
all the players (so that each can be used in taking a turn)	Taking inspiration from the domain model, <i>MonopolyGame</i> is a good candidate.

(adopted from “Applying UML and Patterns” by Craig Larman)

MonopolyGame is Responsible to the Game Loop

- There will be an internal/private helper operation playRound.
- It performs one round (all players take turns) – high cohesion.
- The name of the operation is consistent with a domain dictionary.
(all stakeholders will understand the semantics of this operation)



(adopted from "Applying UML and Patterns" by Craig Larman)

Taking Turns in the Monopoly Game

The responsibility for the take turn action – according to the GRASP Information Expert principle, the object(s) having required information.

Information Needed	Who Has the Information?
current location of the player (to know the start-point of a move)	Taking inspiration from the domain model, a <i>Piece</i> knows its <i>Square</i> and a <i>Player</i> knows its <i>Piece</i> . Therefore, a <i>Player</i> software object could know its location by LRG.
the two <i>Die</i> objects (to roll them and calculate their total)	Taking inspiration from the domain model, <i>MonopolyGame</i> is a candidate since we think of the dice as being part of the game.
all the squares—the square organization (to be able to move to the correct new square)	By LRG, <i>Board</i> is a good candidate.

(adopted from “Applying UML and Patterns” by Craig Larman)

There are **three candidates**: Player, MonopolyGame, and Board.
(all of them are partial information experts for this responsibility)

Taking Turn Responsibility: A Dominant Inf. Expert

(adopted from “Monopoly Use Case” by Alex Thomo)

When there are multiple partial information experts to choose from place the responsibility in the dominant information expert.

- The dominant IE is an object with majority of the information.
- This tends to best support a low coupling.
- Unfortunately, in this case, are all rather equal.
(each with about one-third of information)
- We cannot decide → no dominant expert.

Taking Turn Responsibility: Coupling and Cohesion

(adopted from “Monopoly Use Case” by Alex Thomo)

When there are alternative design choices, consider the coupling and cohesion impact of each, and choose the best.

- MonopolyGame is already doing some work, so giving it more work impacts its cohesion.
(in contrast with a Player and Board objects, which are not doing anything yet)
- However, we still have a two-way tie with Player and Board object.
- There is a high coupling → it is not optimal solution.

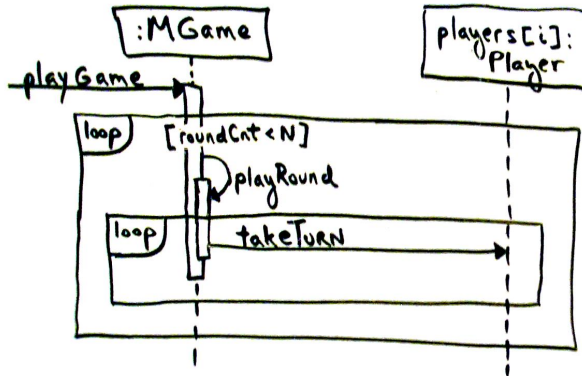
Taking Turn Responsibility: Future Evolution

(adopted from “Monopoly Use Case” by Alex Thomo)

Consider probable future evolution of the software objects and the impact in terms of Information Expert, cohesion, and coupling.

- In the 1st iteration, taking a turn does not involve much info.
- However, taking a turn in a later iteration with the complete set of game rules is not only about rolling the dice and moving the piece.
- A player will be able to buy a property that its piece lands on, with enough money or the colour fitting in with his/her “colour strategy”.
- What object would be expected to know
 - a player’s cash total? ... a Player object
 - a player’s colour strategy? ... a Player object

A Player Object Takes the Turn



MGame

roundCnt

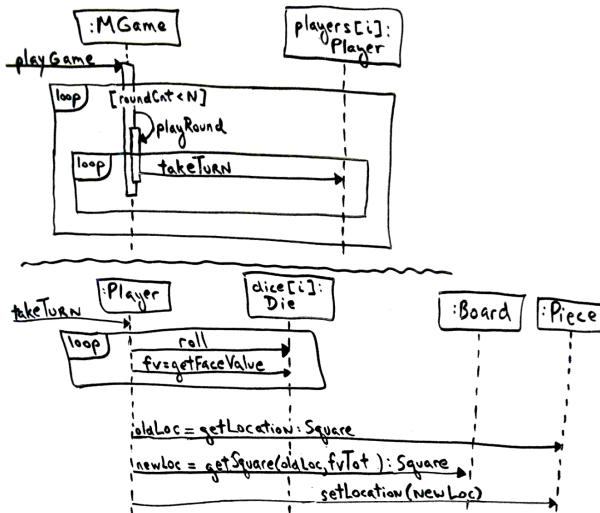
+ playGame

- playRound

Taking Turn (and Rolling Dice)

- Taking a turn means
 - 1 calculating a random number total between 2 and 12 (two dice),
 - 2 calculating the new square location,
 - 3 moving the player's piece from an old to a new square location.
- The calculating a random number is a responsibility of Die object.
(GRASP IE: it is able to randomly generate/set and read/know its faceValue)
- The calculating the new square location is resp. of Board object.
(GRASP IE: based on an old location & offset, Board knows a new square location)
- The moving the player's piece is responsibility of Piece objects.
(GRASP IE: it knows where it stands on the board and to set its new location)
- A coordinator of this action/responsible will be Player object.
- A Player object needs to know Die, Board, and its Piece objects.
(these will be set during the system initialisation, see 56)

The Final Interaction Diagram



(adopted from "Applying UML and Patterns" by Craig Larman)

Command-Query Separation Principle

- Each operation should be just one of
 - 1 a command method that performs an action,
(often has side effects such as changing the state of objects, and it is void)
 - 2 a query that returns data to the caller and has no side effects.
(it should not change the state of any objects)
- Mixing of these two operations leads to unclear responsibilities and unexpected side effects of the operations.
(an interface should behave exactly as the client thinks it behaves)
- This is also known as “Principle Of Least Surprise” (PLS) or “Principle of Least Astonishment” (PLA).

// wrong

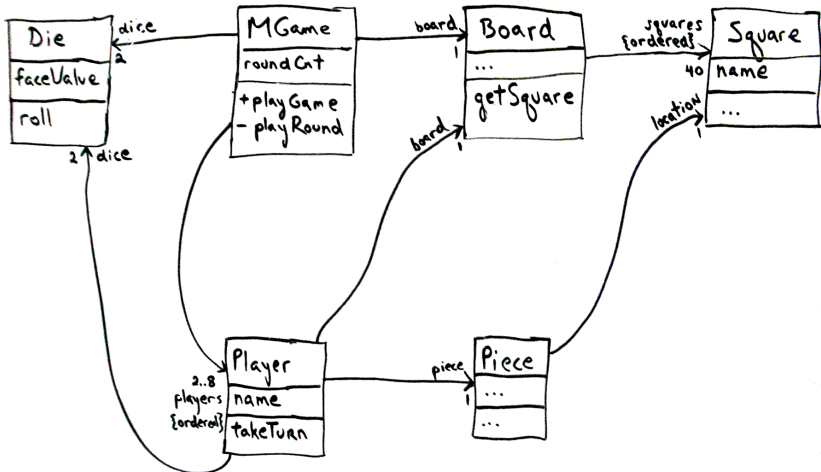
```
public int roll() { faceValue = 2+(int) (Math.random()*10);  
                return faceValue; }
```

// correct

```
public void roll() { faceValue = 2+(int) (Math.random()*10); }  
public int getFaceValue() { return faceValue; }
```



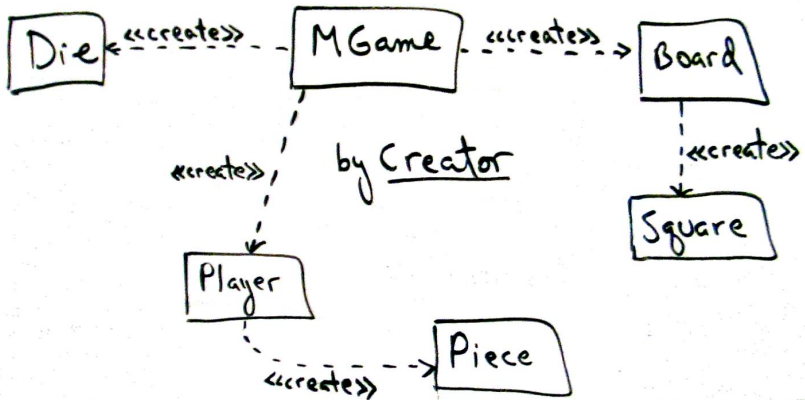
The Final Class Diagram



(adopted from "Applying UML and Patterns" by Craig Larman)

The Final Initialisation Use Case

MonopolyGame object will be responsible for the initialisation.



(adopted from “Applying UML and Patterns” by Craig Larman)

Demo cvičení na návrhové vzory

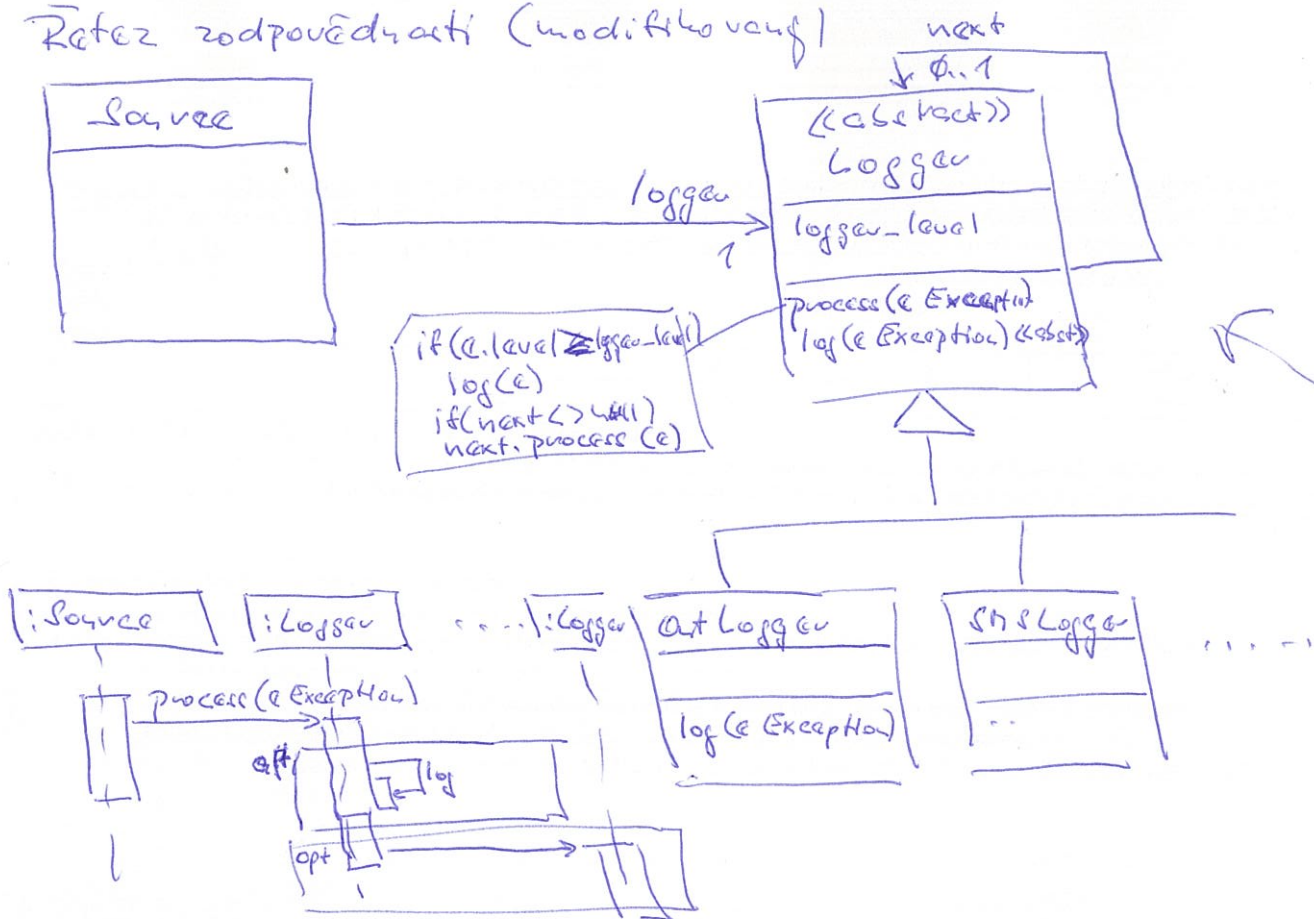
1. Uvažujte, že jste členem týmu, který vyvíjí část informačního systému firmy, která bude poskytovat podporu pro řízení kritických technologických procesů firmy. Vaším úkolem je navrhnout řešení související se zpracováním hlášení o výjimečných stavech procesů. Předpokládejte, že potřebné informace o takovém stavu jsou zapouzdřeny v objektu třídy *Error*. Pro jednoduchost předpokládejte, že část systému, ve které mohou být tyto výjimečné stavy detekovány, je reprezentována objektem třídy *Detector*. Ta tedy při detekci vytvoří instanci třídy *Error* a předá ji ke zpracování skupině tříd, kterou máte navrhnout vy. Výjimečné stavy lze klasifikovat do několika úrovní významnosti (předpokládejte odpovídající atribut třídy *Error* s názvem *level*). Může existovat řada způsobů hlášení podle významnosti výjimečného stavu, např. výpis na konzolu, poslání SMS, uložení detailní informace do logovacího souboru apod. Je požadováno, aby navržené řešení bylo snadno rozšiřitelné i pro případné další typy hlášení (minimální dopad na kód ostatní části systému). Uvažujte, že zpracování může být složitější a je proto žádoucí, aby byl každý typ zpracování zapouzdřen v samostatné třídě. Navíc to, jestli daný typ zpracování proběhne, je dáno úrovní pro zpracování daného typu – proběhne pouze, je-li větší nebo rovna hodnotě *level* objektu třídy *Error*.

Při vzniku výjimečného stavu se musí objekt třídy *Error* vytvořený objektem třídy *Detector* dostat ke každému objektu, který zodpovídá za zpracování, a ten podle významnosti stavu a své úrovně zpracování požadavek zpracuje nebo ne.

Nakreslete diagram tříd a diagram sekvence nebo komunikace a komunikaci stručně popište. V diagramu tříd musí být uvedeny operace, které navrhujete pro realizaci požadovaného chování. Tyto operace popište pseudokódem nebo slovně (kde by byl popis složitý). Pokud vaše řešení vychází z nějakého návrhového vzoru, napište jeho název a popište podstatu. Pokud ne, diskutujte, proč se žádný ze vzorů probíraných v předmětu AIS nehodí.

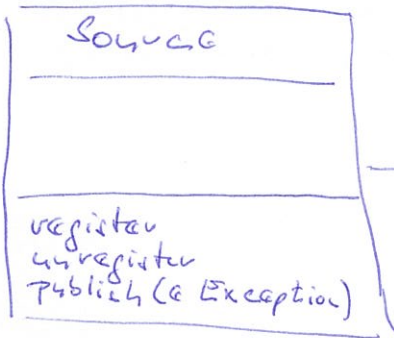
18 bodů

Řetez zodpovědnosti (modifikování)

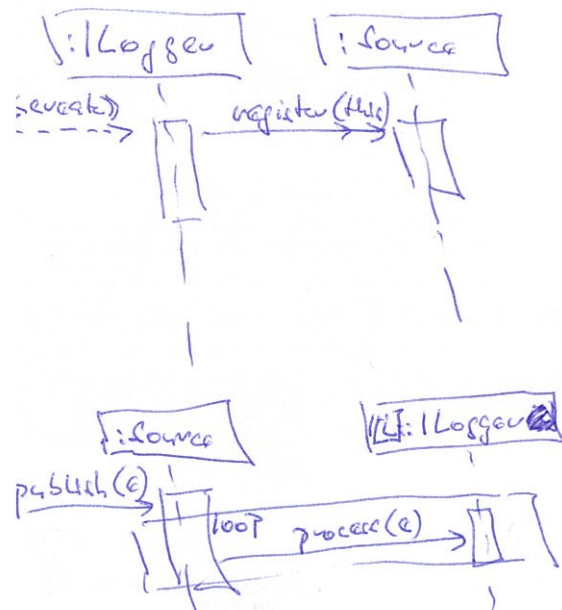


Publikace

okolo

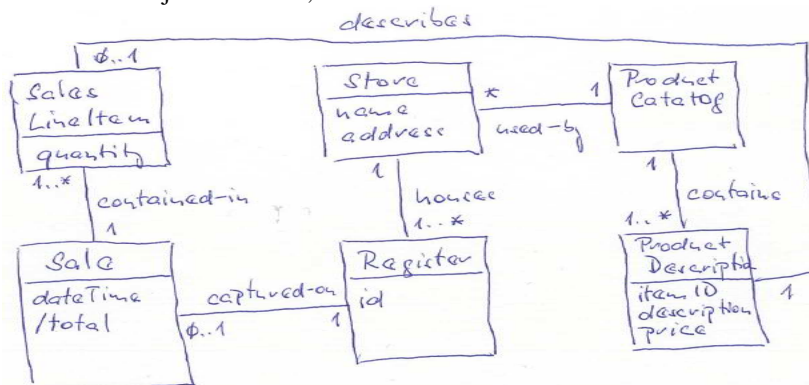


nebo abstr.
trída
s logger_level

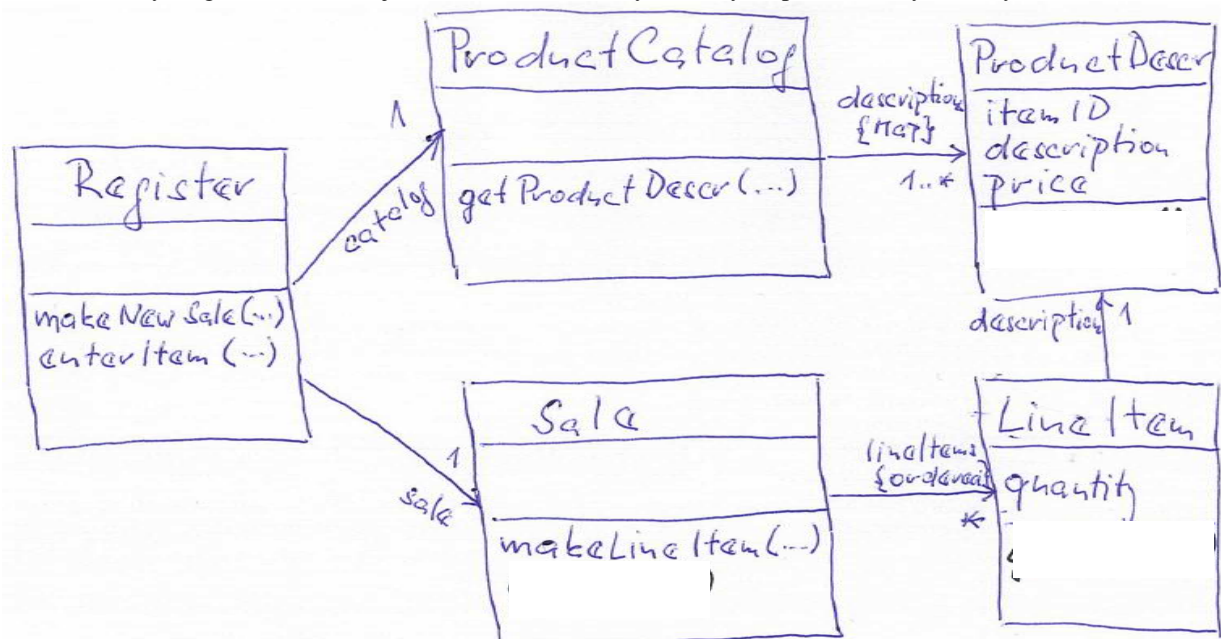


if (e.level > logger_level)
zašli(e)

1. Předpokládejte, že vyvíjíte software pro software *NextGenPOS*, jak je definován v případové studii použité na přednáškách. Jste členem vývojového týmu a daný software vyvíjíte v iteracích. Jste v 1. iteraci a navrhujete třídy vrstvy domény pro realizaci hlavního scénáře případu použití *Process Sale*, který zpracovává obsah nákupního košíku. Část realizace scénáře (zahájení nového prodeje, a část zpracování položky košíku) už byla navržena. Momentálně řešíte návrh operace *getTotal()*, která spočítá aktuální cenu dosud zpracovaného obsahu košíku. Tato operace není systémovou operací, využije se pro zobrazení průběžné celkové ceny nákupu. Vy zobrazení neřešíte, jen řešíte zajištění odpovídající hodnoty, kterou funkce vrací. Znáte model domény pro *NextGenPOS*, jehož relevantní část je na obrázku,



znáte návrhový diagram tříd obsahující dosud navržené třídy, atributy a operace vrstvy domény (viz níže)



a víte, že bylo rozhodnuto, že třída *Register* slouží jako řadič. Již navržená operace *makeNewSale()* představuje zahájení nového prodeje (zpracování obsahu košíku). V rámci ní se vytvoří instance třídy *Sale*, včetně kolekce položek *lineItems*. Instance třídy *Register*, *ProductCatalog* a naplněná kolekce popisů zboží *descriptions* byly vytvořeny ještě dříve. Operace *enterItem(...)* představuje zpracování jedné položky obsahu košíku. Řadič deleguje provedení třídě *Sale* (operace *makeLineItem(...)*). Ta vytvoří instanci třídy *LineItem*, nastaví u ní hodnotu atributu *quantity* udávajícího počet kusů a vytvoří vazbu na odpovídající popis zboží (instanci třídy *ProductDescr*) a vloží ji do kolekce *lineItems*. Operace *getProductDescr(ID)* třídy *ProductCatalog* získá instanci třídy *ProductDescr* s daným *ID*, atribut *price* ve třídě *ProductDescr* udává jednotkovou cenu daného zboží a atribut *description* jeho popis.

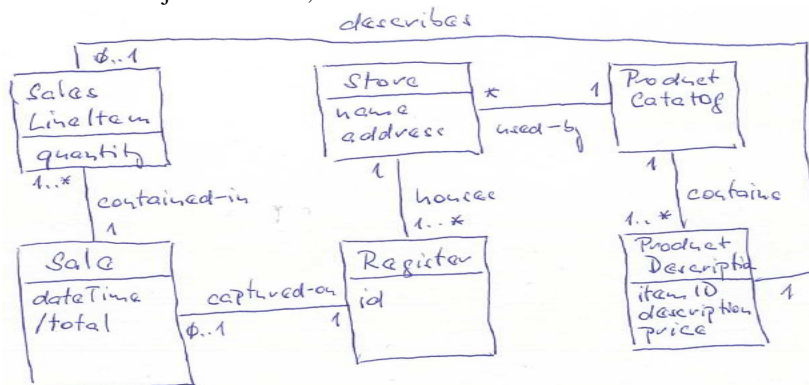
Při návrhu realizace operace *getTotal()* postupujte tak, že strukturovaným způsobem podobným CRC štítkům rozhodnete o přiřazení zodpovědnosti třídám a o spolupracujících třídách (opět přiřazení zodpovědností a jim odpovídajících operací). Každé přiřazení zodpovědnosti zdůvodněte (PROČ jste přiřadil danou zodpovědnost právě jí). Pokud použijete některý princip GRASP nestačí uvést jenom jeho název, ale i tady musí být zřejmé, proč. Použijte pro tyto účely tabulku níže.

Zodpovědnost, operace	Třída	Zdůvodnění	Spolupracující třídy

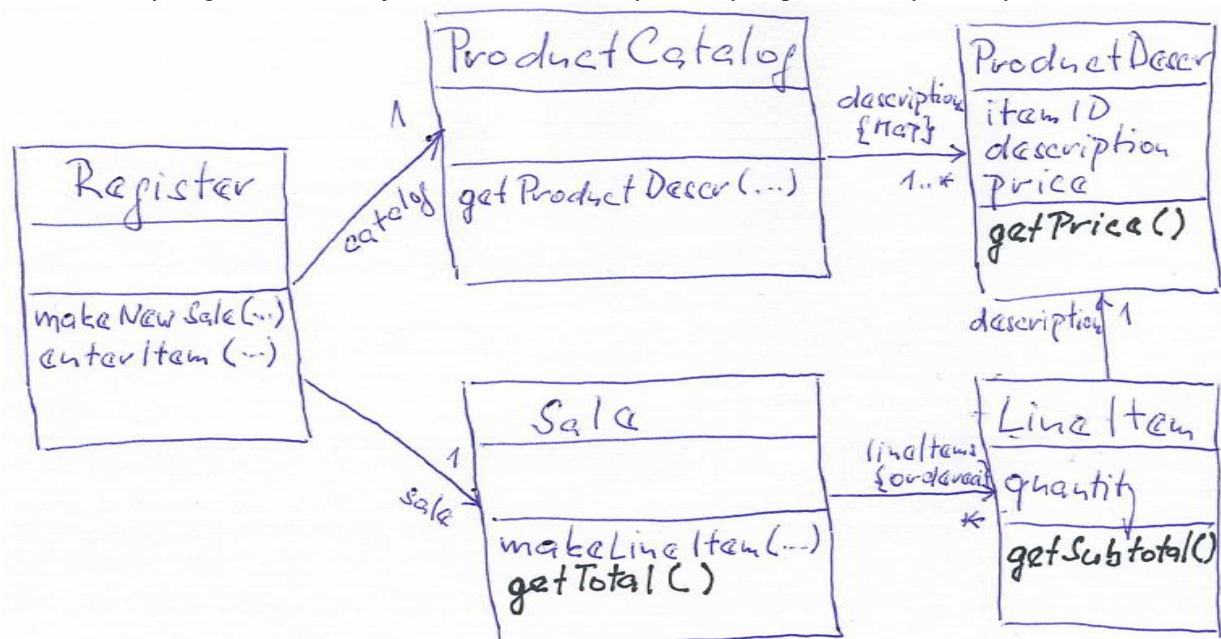
Do návrhového diagramu tříd výše doplňte vámi navržené třídy, operace a atributy a nakreslete diagram sekvence operace `getTotal()`, který bude ukazovat interakci spolupracujících tříd.

11 bodů

1. Předpokládejte, že vyvíjíte software pro software *NextGenPOS*, jak je definován v případové studii použité na přednáškách. Jste členem vývojového týmu a daný software vyvíjíte v iteracích. Jste v 1. iteraci a navrhujete třídy vrstvy domény pro realizaci hlavního scénáře případu použití *Process Sale*, který zpracovává obsah nákupního košíku. Část realizace scénáře (zahájení nového prodeje, a část zpracování položky košíku) už byla navržena. Momentálně řešíte návrh operace *getTotal()*, která spočítá aktuální cenu dosud zpracovaného obsahu košíku. Tato operace není systémovou operací, využije se pro zobrazení průběžné celkové ceny nákupu. Vy zobrazení neřešíte, jen řešíte zajištění odpovídající hodnoty, kterou funkce vrací. Znáte model domény pro *NextGenPOS*, jehož relevantní část je na obrázku,



znáte návrhový diagram tříd obsahující dosud navržené třídy, atributy a operace vrstvy domény (viz níže)

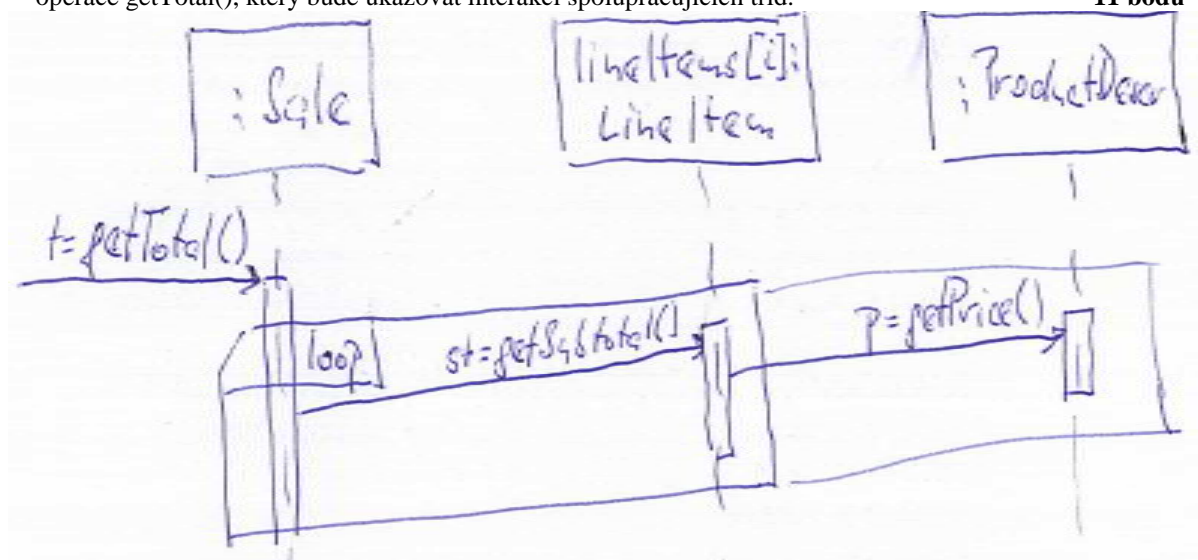


a víte, že bylo rozhodnuto, že třída *Register* slouží jako řadič. Již navržená operace *makeNewSale()* představuje zahájení nového prodeje (zpracování obsahu košíku). V rámci ní se vytvoří instance třídy *Sale*, včetně kolekce položek *lineItems*. Instance třídy *Register*, *ProductCatalog* a naplněná kolekce popisů zboží *descriptions* byly vytvořeny ještě dříve. Operace *enterItem(...)* představuje zpracování jedné položky obsahu košíku. Řadič deleguje provedení třídě *Sale* (operace *makeLineItem(...)*). Ta vytvoří instanci třídy *LineItem*, nastaví u ní hodnotu atributu *quantity* udávajícího počet kusů a vytvoří vazbu na odpovídající popis zboží (instanci třídy *ProductDescr*) a vloží ji do kolekce *lineItems*. Operace *getProductDescr(ID)* třídy *ProductCatalog* získá instanci třídy *ProductDescr* s daným *ID*, atribut *price* ve třídě *ProductDescr* udává jednotkovou cenu daného zboží a atribut *description* jeho popis.

Při návrhu realizace operace *getTotal()* postupujte tak, že strukturovaným způsobem podobným CRC štítkům rozhodnete o přiřazení zodpovědnosti třídám a o spolupracujících třídách (opět přiřazení zodpovědností a jim odpovídajících operací). Každé přiřazení zodpovědnosti zdůvodněte (PROČ jste přiřadil danou zodpovědnost právě jí). Pokud použijete některý princip GRASP nestačí uvést jenom jeho název, ale i tady musí být zřejmé, proč. Použijte pro tyto účely tabulku níže.

Zodpovědnost, operace	Třída	Zdůvodnění	Spolupracující třídy
vypočet ceny (ind kupa) getTotal()	Sale	IE - je schopna získat všechny potřebné informace (ind dostupnou kolekci položek)	LineItem + kolekce
získání ceny jedné položky getSubtotal()	LineItem	IE - získá počet ks a je schopna získat jednotkovou cenu	ProductDescr
získání jednotkové ceny getPrice()	ProductDescr	IE - ind atribut s touto hodnotou	—

Do návrhového diagramu tříd výše doplňte vámi navržené třídy, operace a atributy a nakreslete diagram sekvence operace getTotal(), který bude ukazovat interakci spolupracujících tříd. **11 bodů**



Thank you for your attention!

Marek Rychlý
<rychly@fit.vutbr.cz>