Topic 2
UML Revision, GRASP Principles and CRC Cards

Mastering OOD

- OOD is sometimes taught as the sequence of steps:
  - Identify requirements
  - Create a domain model
  - Add methods
  - Define the messages between objects
- In reality, deep principles and patterns must be used when deciding:
  - Where methods must be placed
  - How objects should interact
  - Today we will look at our second group of patterns called GRASP
- First however let’s quickly do some UML revision to revisit how to represent OO design

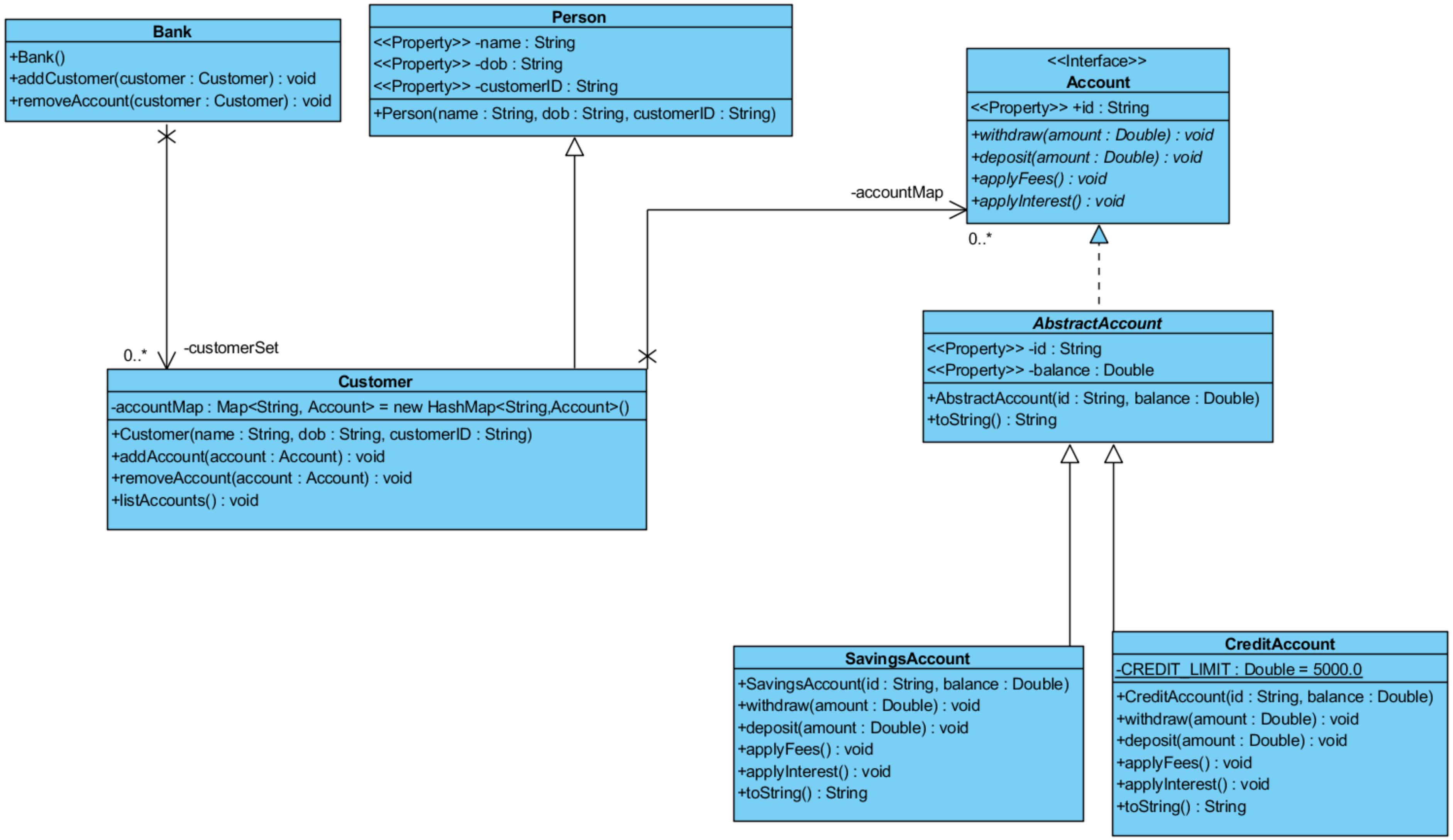
UML versus Design Principles

- UML is a standard visual modelling language
- However, UML does not guide the way objects must be designed
- It is misleading to call UML a “design tool”
- How to become a good designer?
  - Familiarity with design principles and patterns
  - Learnt through experience both positive and negative

UML Outputs (examples)

- Class diagrams
  - most widely used diagram type since classes are the most fundamental OO construct
- Use Case Diagrams
  - document requirements using domain concepts (user readable)
- Interaction/Sequence diagrams
  - useful where interaction is non trivial
- State Diagrams
  - useful where solution contains complex state
- Package diagrams
  - useful to organise large systems
- Deployment diagrams
  - useful for distributed systems or complex deployments
- Data Models
  - useful where non OO database is used (e.g. ER diagram .. non UML)
- UI Prototypes and Sketches
  - useful during prototyping (non UML)

UML Class Diagram


Bank
+Bank()
+addCustomer(customer : Customer) : void
+removeAccount(customer : Customer) : void
Person
<<Property>> -name : String
<<Property>> -dob : String
<<Property>> -customerID : String
+Person(name : String, dob : String, customerID : String)
<<Interface>>
Account
<<Property>> +id : String
+withdraw(amount : Double) : void
+deposit(amount : Double) : void
+applyFees() : void
+applyInterest() : void
-accountMap
0..*
0..*
-customerSet
Customer
-accountMap : Map<String, Account> = new HashMap<String,Account>()
+Customer(name : String, dob : String, customerID : String)
+addAccount(account : Account) : void
+removeAccount(account : Account) : void
+listAccounts() : void
AbstractAccount
<<Property>> -id : String
<<Property>> -balance : Double
+AbstractAccount(id : String, balance : Double)
+toString() : String
SavingsAccount
+SavingsAccount(id : String, balance : Double)
+withdraw(amount : Double) : void
+deposit(amount : Double) : void
+applyFees() : void
+applyInterest() : void
+toString() : String
CreditAccount
-CREDIT_LIMIT : Double = 5000.0
+CreditAccount(id : String, balance : Double)
+withdraw(amount : Double) : void
+deposit(amount : Double) : void
+applyFees() : void
+applyInterest() : void
+toString() : String

UML Class Diagram Relationships

Association – The most generic form of relationship. One class uses another class usually though an attribute level reference (since method parameter or local variable reference implies an even weaker **dependency** relationship).

Aggregation – Similar to association but only refers to attribute level access (not parameters or local variables). Also implies part/whole semantics but objects on both sides of the relationship have independent lifetimes.

Composition – Same as aggregation but enforces part/whole semantics i.e. if the whole is deleted then so are the parts.

Inheritance – An “is a” or extends relationship implying abstraction or superclass/subclass semantics.

NOTE: In many cases (especially at the design stage) association and inheritance are sufficient.

Non Inheritance Relationships – Details

Assume *A* has a directed (one way) association with *B* i.e. *A* “has a” (or uses) *B*

AGGREGATION: *A* requires *B* to operate and in particular *B* is initialized in (or passed to) the constructor of *A* (B is **part of** A)

COMPOSITION: As above but *B* follows the lifecycle of *A* or more specifically it is destroyed in the destructor of *A* (for languages that support this construct) or when *A* is finalised via garbage collection (A is **composed of** B)

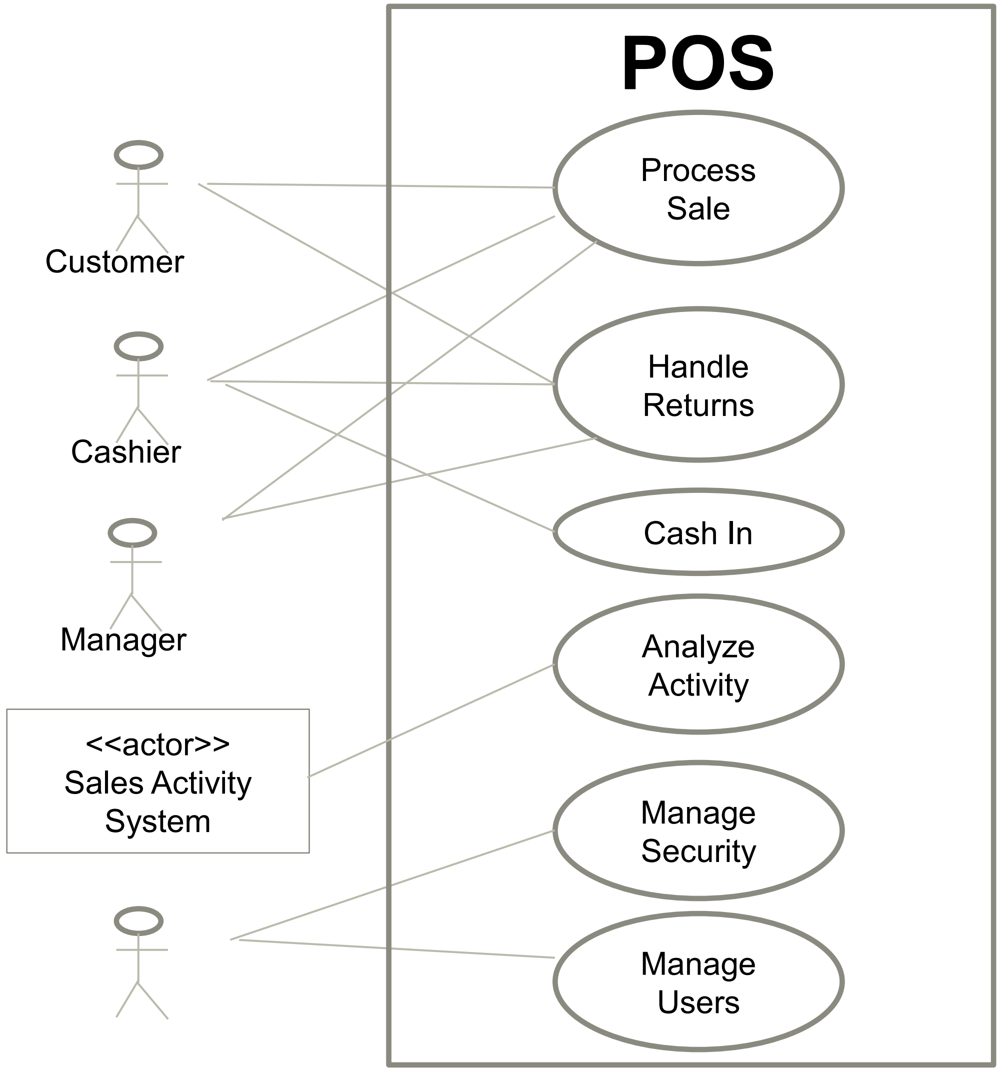
ASSOCIATION: Neither of the above applies so it is just a general class usage represented as an attribute and directed association (A **uses** B)

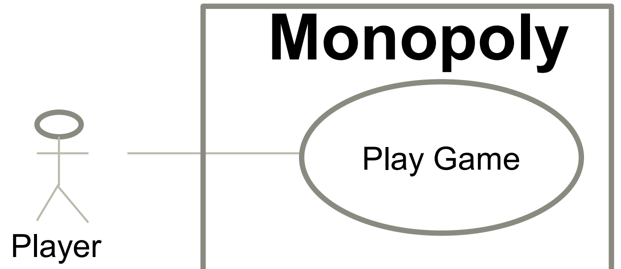
DEPENDENCY: *A* has a parameter or local variable reference to *B*. In practice this distinction from association is not always made at the design stage (A **uses** B)

Use Cases and Scenarios

- Use cases are ANALYSIS artefacts that can be represented as text or more formal diagrams (e.g. UML Use Case Diagram)
  - a **scenario** is a specific instance of a use case with specific inputs and outcomes
- Written from the perspective of the user using domain language (i.e. the language of the problem) not development constructs
- Describe a specific goal that the user wants to achieve, independent of implementation details e.g. select product, make reservation, make payment etc.
- A combination of actors (users or external systems), use cases (tasks/goals) and associations between the two

Use Case Diagrams





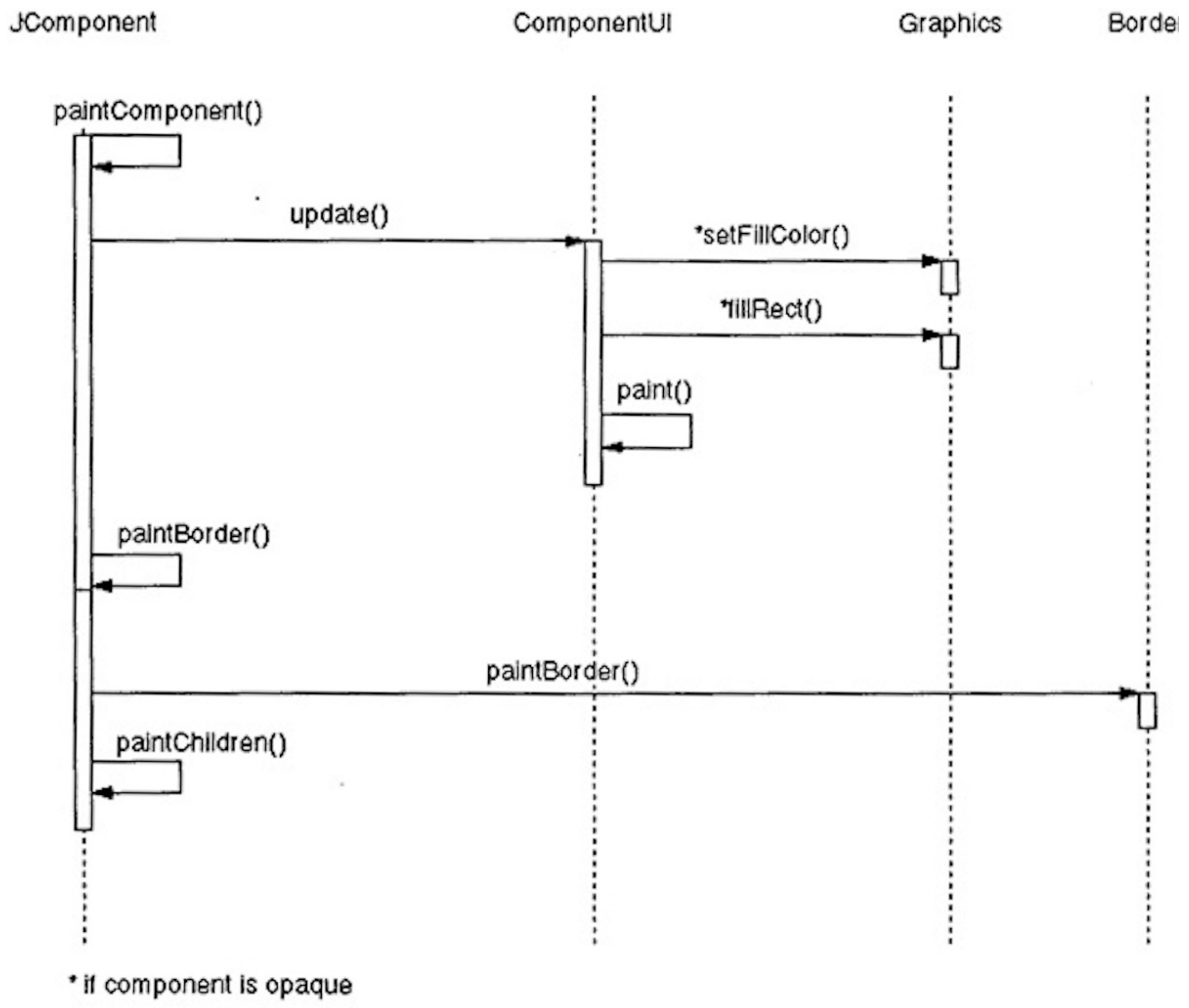
Use Cases and Scenarios (contd.)

- May link to (sub)systems and processes that address the problem but which are described in more detail in other diagrams
- Can annotate with storyboards, mock-ups etc. although these may come later
- Keep use cases simple so they describe the problem to be solved not HOW to solve it
- Use case diagrams are high level and are supplemented with textual descriptions (which contain the bulk of the relevant information)
- Use other diagrams such as sequence diagrams or state diagrams where you want to show detailed process or implementation details (can be linked from use case)

Sequence Diagrams

- Sequence diagrams show the interaction between objects at runtime
- Generally they represent messages being sent to and received from components
- In an OO language such as Java, they represent method calls/ parameters as well as return values
- Messages occur in a (usually numbered) sequence with time spanning vertically top to bottom
- The various objects involved in the interaction are spread horizontally
- UML supports notation for conditionals/branching as well as iteration/looping

- **NOTE**: Use only where the interaction is non trivial. e.g. a sequence diagram showing two objects (A and B) is often not very useful!
- See Java Swing example on next slide



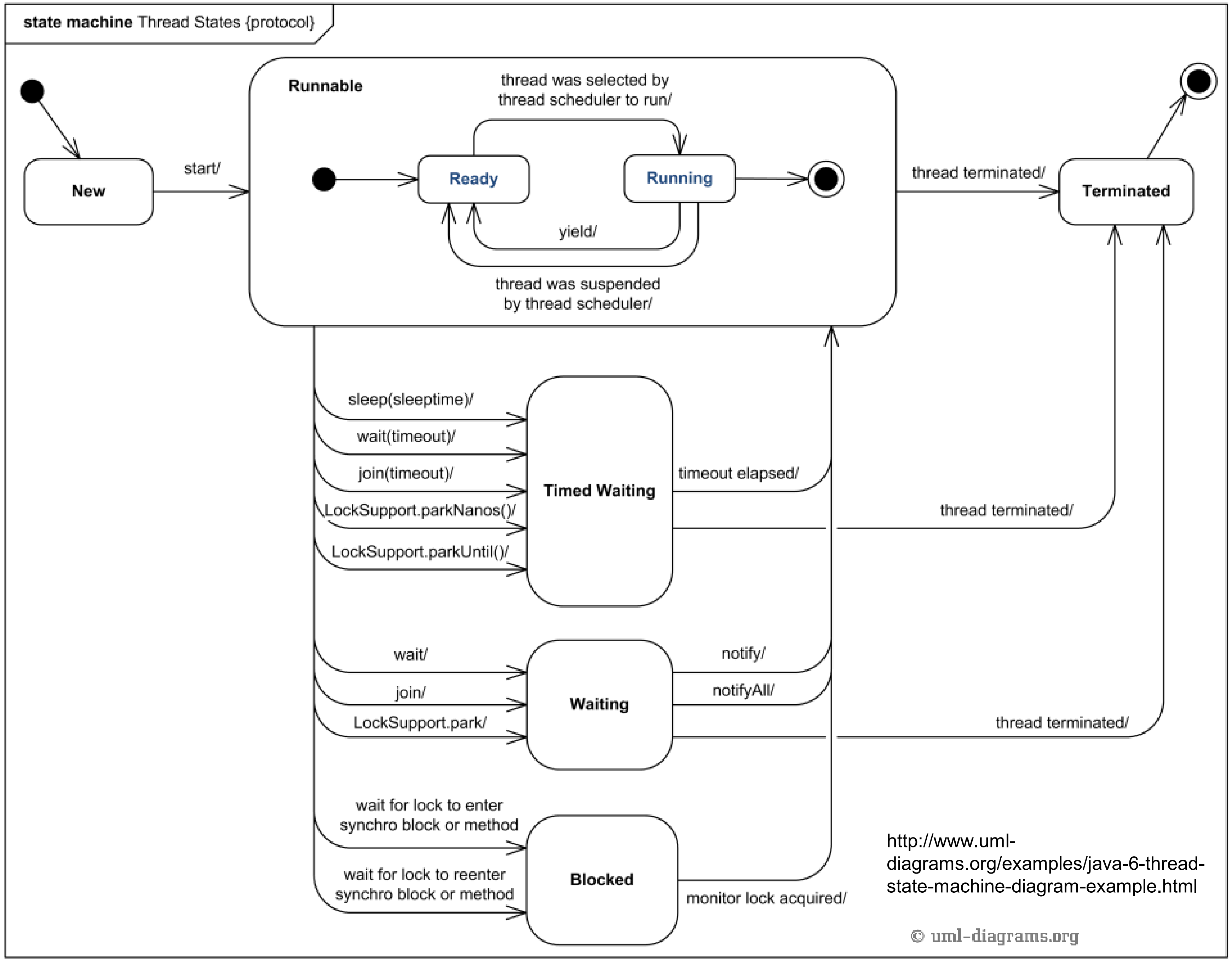
jaxax.swing.JComponent.paint() From: David M. Geary, Graphic Java 2: Swing, Prentice Hall, 1999

System Sequence Diagrams

- As described previously UML Sequence Diagrams are primarily used for showing the flow of control of method calls between objects
- However they are also useful for providing graphical representations of use case scenarios
- Visual Paradigm supports this usage by including actors in sequence diagrams
- Such usage is called a System Sequence Diagram because it shows actors interacting with the system
- Described in terms of actions and verbs from the problem domain NOT implementation constructs such as classes, objects and methods
- Model scenarios including the success case as well as complex fail cases
- Derived form and can be annotated with text from the associated use case

State Diagrams

- Modelled on the State Machine concept showing:
  - possible Object states
  - transitions between states
  - events that trigger state transitions
- Can be used at different stages of the SDLC
  - during Analysis and Use Case Modelling to show valid business rules or legal/valid system states
  - during design and implementation to show actual object and algorithm behaviour
- **NOTE**: Do not force usage or overuse, many software artefacts and data driven applications have very little interesting state
- Some good usage example are protocols (e.g. communication and security) and lifecycles (see Java Thread lifecycle on next slide)
- The game rules for your assignment are a good candidate!

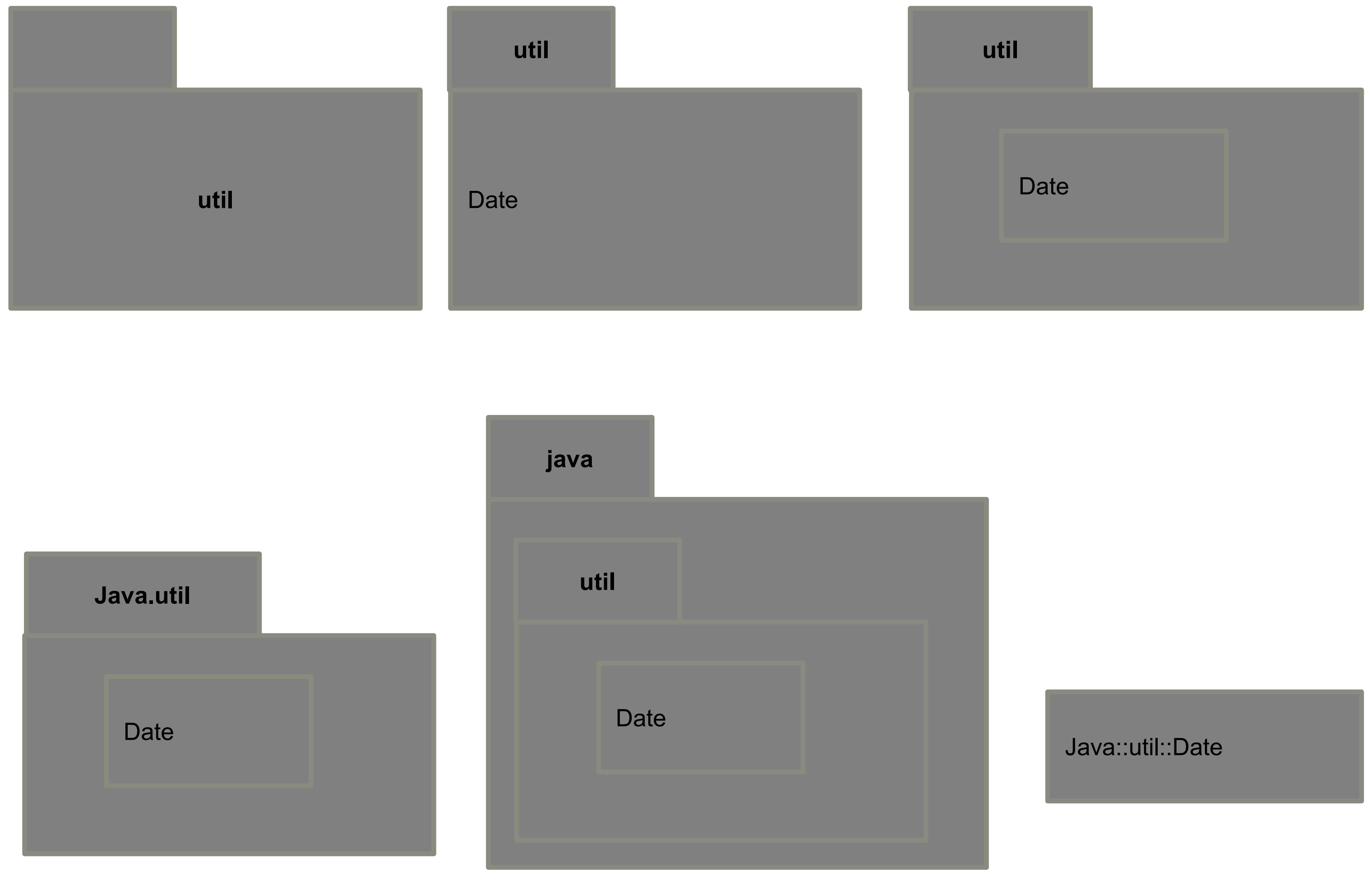


http://www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html

Package Diagrams

- A package is a grouping construct that allows you to take any construct in the UML and group its elements together into higher-level units.
- Commonly used for grouping classes. Each class in UML (and Java) is a member of only one package.
- A package can be a member of other packages allowing hierarchical packaging.
- A package can show only package name, or package name and contents. Even a class diagram can be shown within a package.
- When two different classes have the same name (but in different packages) fully qualified names can be used.
- In diagrams packages are shown as a tabbed folder.

Package Diagrams: Different ways

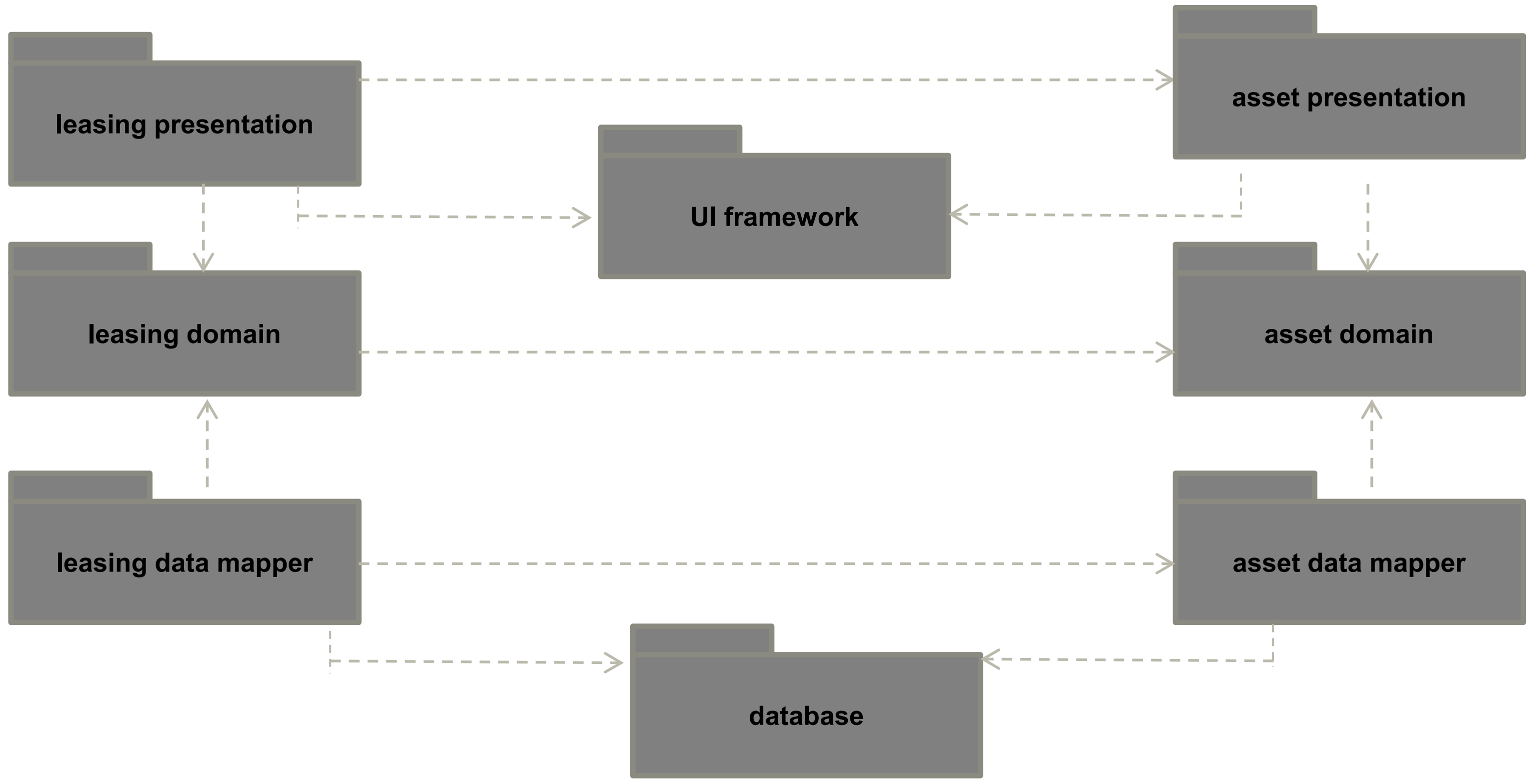


Structuring Packages and Classes

- UML allows classes in packages to be public or private
- A public class is part of the interface of the package and can be used by classes in other packages
- Private classes are hidden
- Different programming environments have different rules for package visibility – may require bending UML rules.
- How to group classes into packages?
  - Common Closure Principle says classes in a package should need changing for similar reasons
  - Common Reuse Principle says classes in a package should all be reused together
  - Subpackages may be chosen for organisational convenience since the expectation is that they will be reused together with super package

Packages and Dependencies

- A package diagram that shows dependencies allows the structure in a medium to large system to be easily visualised.

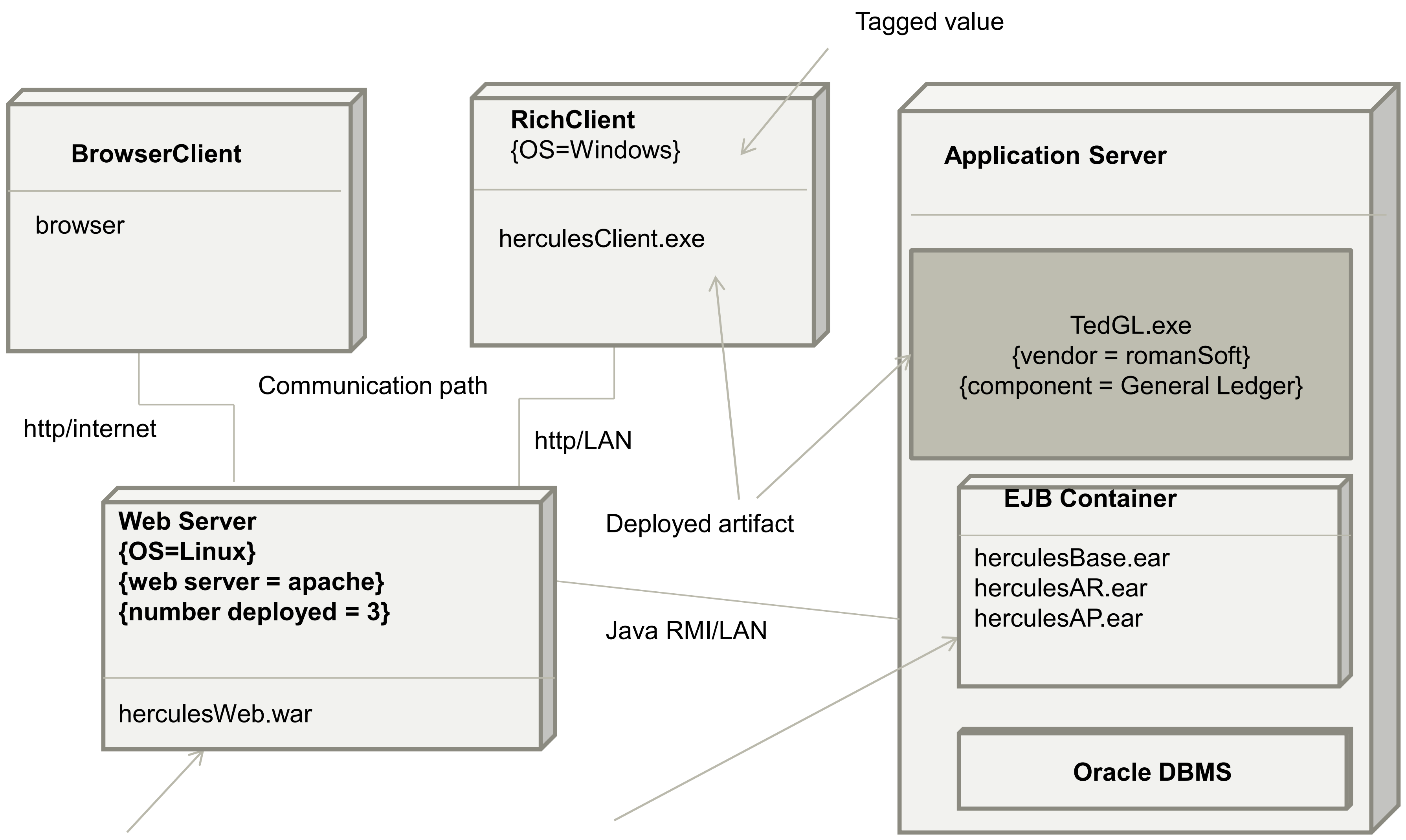


Deployment Diagram

- Deployment diagrams represent a system’s physical layout showing the software components and processes mounted/running on various devices
- Main items in DD are nodes connected by communication paths. Communication paths can show the protocols used
- Nodes can be either
  - A device that hosts software directly
  - An execution environment that hosts software indirectly (web/applicaiton servers, VMs etc.)
- Nodes contain software artefacts such as executable files, jar files, DLLs, databases etc.
- When multiple physical nodes carry out the same tasks “number deployed” tag can be used to indicate the number
  - e.g. load balancing, caching, redundancy

NOTE: Not needed for the assignment which is a standalone Java application executing on a single node

Deployment Diagram



GRASP (General Responsibility Assignment Software Patterns)

- From:
  - *Larman, Craig (2005) Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.). Prentice Hall.*
- Grasp Principles and Patterns provide a methodical way to reason about assigning responsibilities to objects
  - where/when are objects created?
  - where should methods go?
  - what is the flow of control?
- GRASP Patterns:

– Creator	-Information Expert
– Low Coupling (*)	-High Cohesion (*)
– Controller	-Polymorphism (*)
– Pure Fabrication	-Indirection
– Protected Variations	

* Some of these are probably already quire familiar!

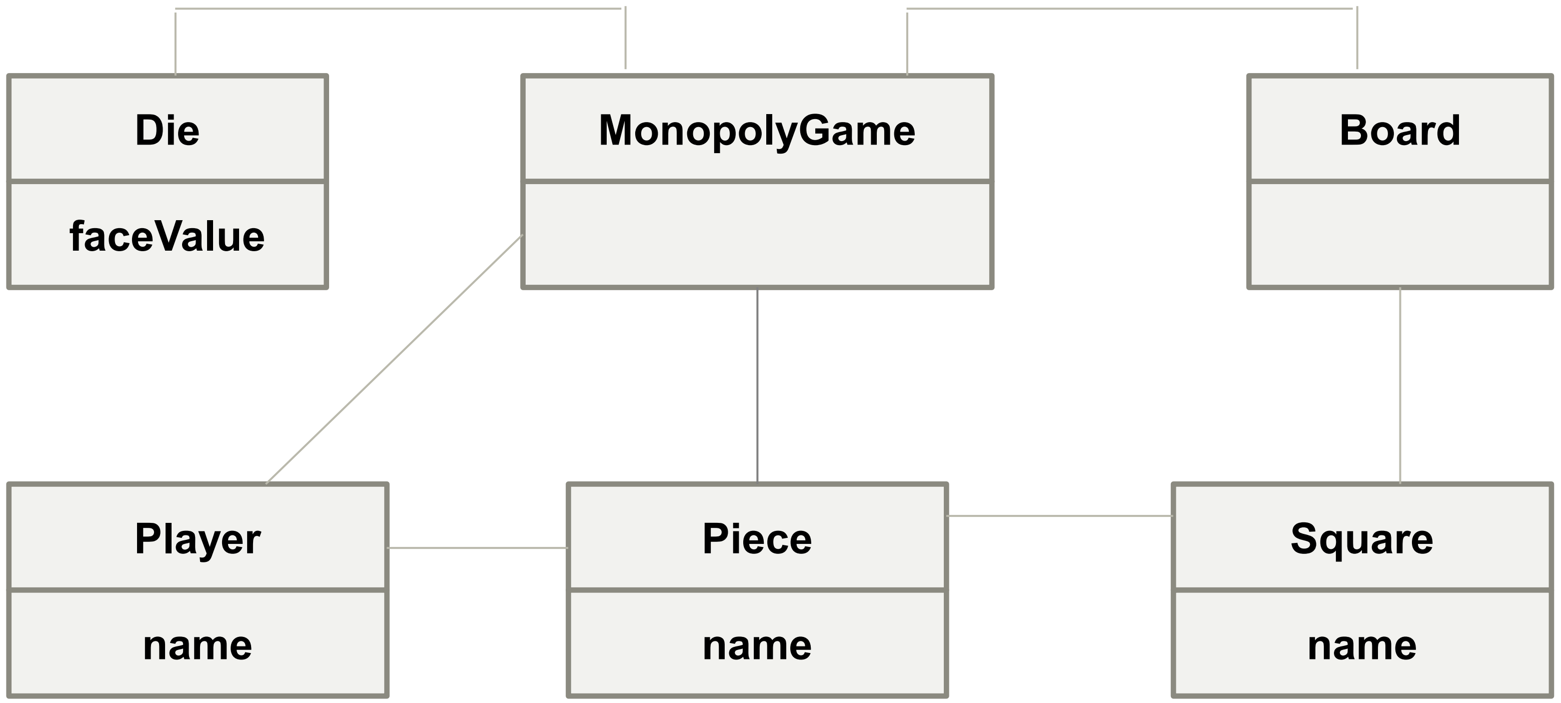
Creator pattern (definition)

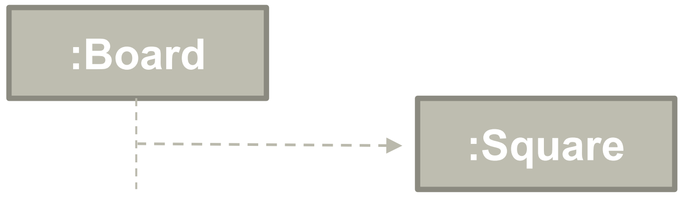
Name	Creator
Problem:	Who creates an A?
Solution:	Assign class B the responsibility to create an instance of class A if one of these is true B aggregates/contains A B records A B closely uses A B has the initializing data for A (information expert)

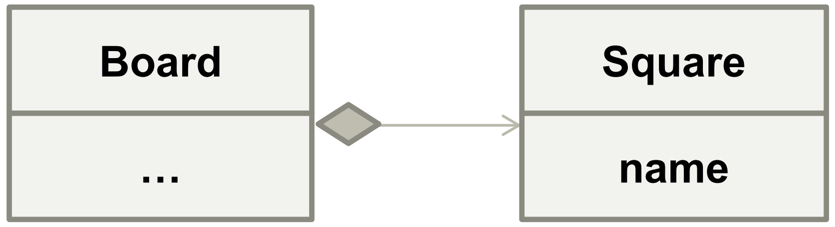
Note A and B refer to Software objects not domain model objects

Creator (Example)

- One of the first problems in OO design is:
  - Who creates object X?
- e.g. Who creates the Square object in Monopoly?
  - The Player object?
  - The Board object?
  - The Die object?
  - The Game object?
- A low representational gap (LRG) between domain model and software objects suggests “Board object creates the Square objects”
- It reflects the intuition “Containers” should create the things “contained therein”

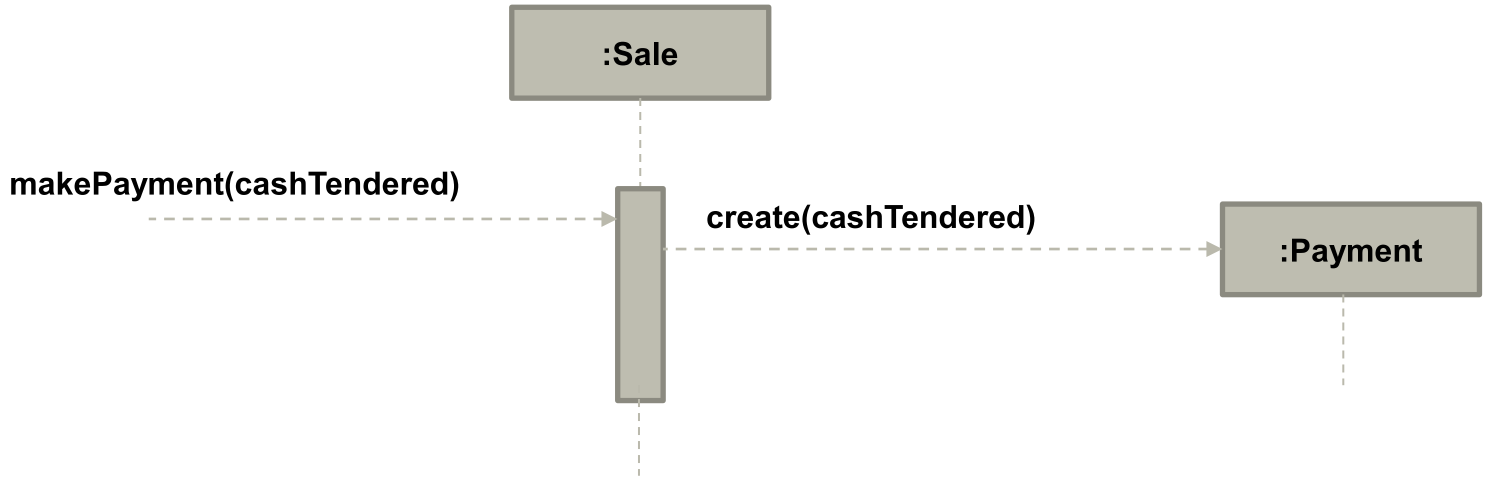

Die
faceValue
MonopolyGame
Board
Player
name
Piece
name
Square
name


:Board
:Square


Board
…
Square
name

Responsibilities, GRASP and UML

- In a POS System GRASP may suggest that the responsibility of creating a payment be assigned to Sale object



- Hence a makePayment message is sent to **Sale** which is handled by **makePayment** method.
- Fulfilment of this responsibility requires the collaboration between Sale (sending a message) and Payment (invoking constructor)

Creator: Discussion

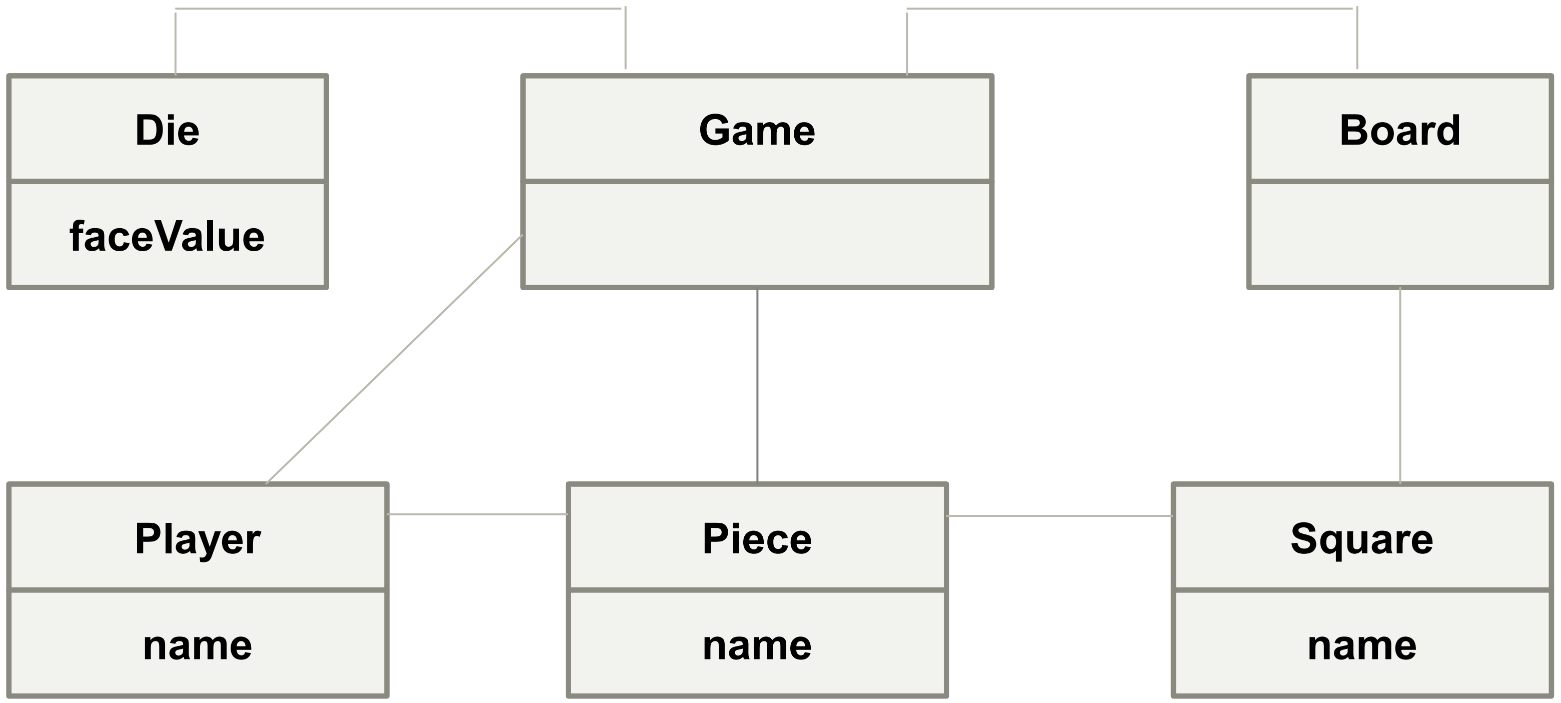
- Reduces coupling since the creating class is already coupled to the created class through containment, use etc.
- Often, creation involves significant complexity, such as using recycled instances (caching/pooling) for performance reasons.
- In such cases it is better for cohesion to delegate the creation to a separate Factory class (even though this increases coupling)
  - More details on Factories later in the semester.

Information Expert (definition)

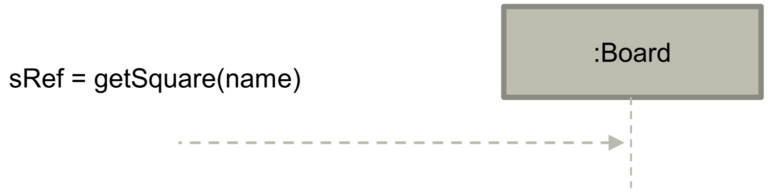
Name	Information Expert
Problem:	What is a basic principle by which to assign responsibilities to objects?
Solution:	Assign responsibility to the class that has the information needed to fulfil it.

Information Expert

- One of the most fundamental principles in assigning responsibility
  - Question: Which class/object should be responsible for a given function/behaviour?
  - Answer: The class/object that has the information necessary to perform the operation.
- Who should be responsible to get the reference to square given its name?
  - The Player object?
  - The Board object?
  - The Dice object?
  - The Game object?
- Information expert suggests “Board object should be responsible”
  - since the “Board” has the information necessary because it already aggregates/contains all the Square objects

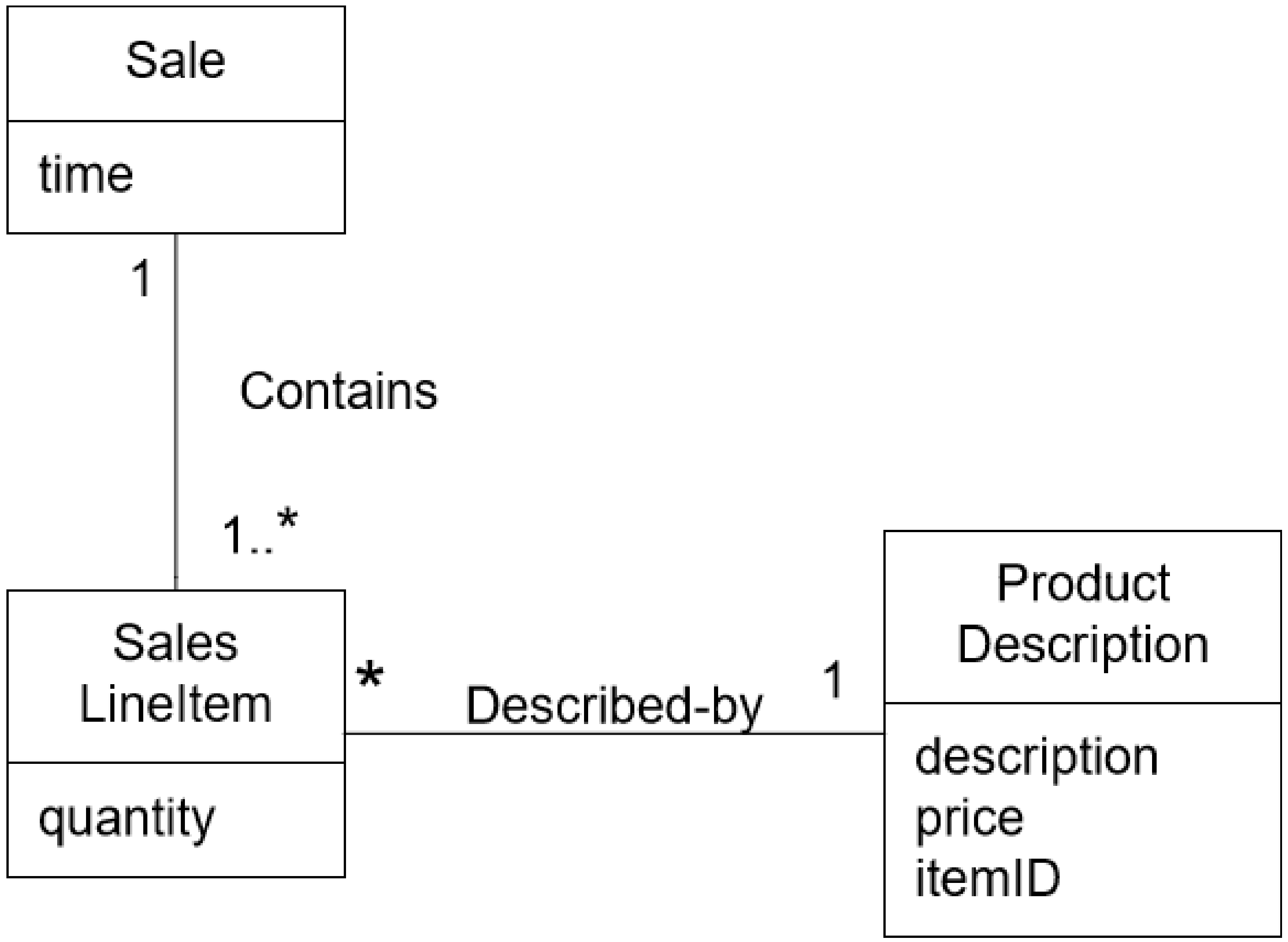


By InformationExpert



Information Expert: Discussion

- **Information Expert** has real world analogies: responsibilities are given to individuals who posses the necessary information.
  - Q: Who should be responsible for creating profit and loss accounts in a business?
  - A: Chief Financial Officer (CFO), Accountant etc.
- Implementing a use case will often be a collaboration of multiple information experts (according to cohesion)
- In some situations the use of **Information Expert** principle is inappropriate. For example, IE suggests that Sale class should be responsible for saving its details to a database. Similarly for Payment, Product etc.
- This however causes conflicts with other principles, namely cohesion and coupling:
  - Cohesion because each of these classes must contain logic related to database handling such as SQL and JDBC access
  - Coupling because each of these classes must be coupled with classes providing Database services.
- **EXAMPLE**: Consider the case of calculating a sale total using classes *Sale*, *LineItem* and *Product* (see next slide)



EXERCISE: Show a sequence diagram for calculating a sale total using the above classes (add methods as appropriate)

Low Coupling (definition)

Name	Low Coupling
Problem:	How to reduce the impact of change
Solution:	Assign responsibilities such that coupling remains low. Use this principle to evaluate alternatives.

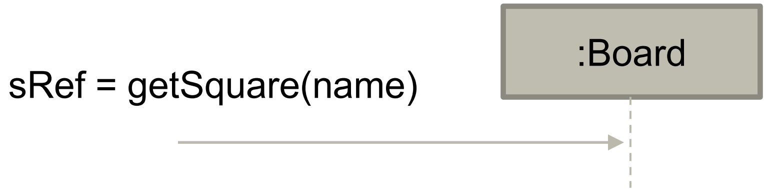
Low Coupling

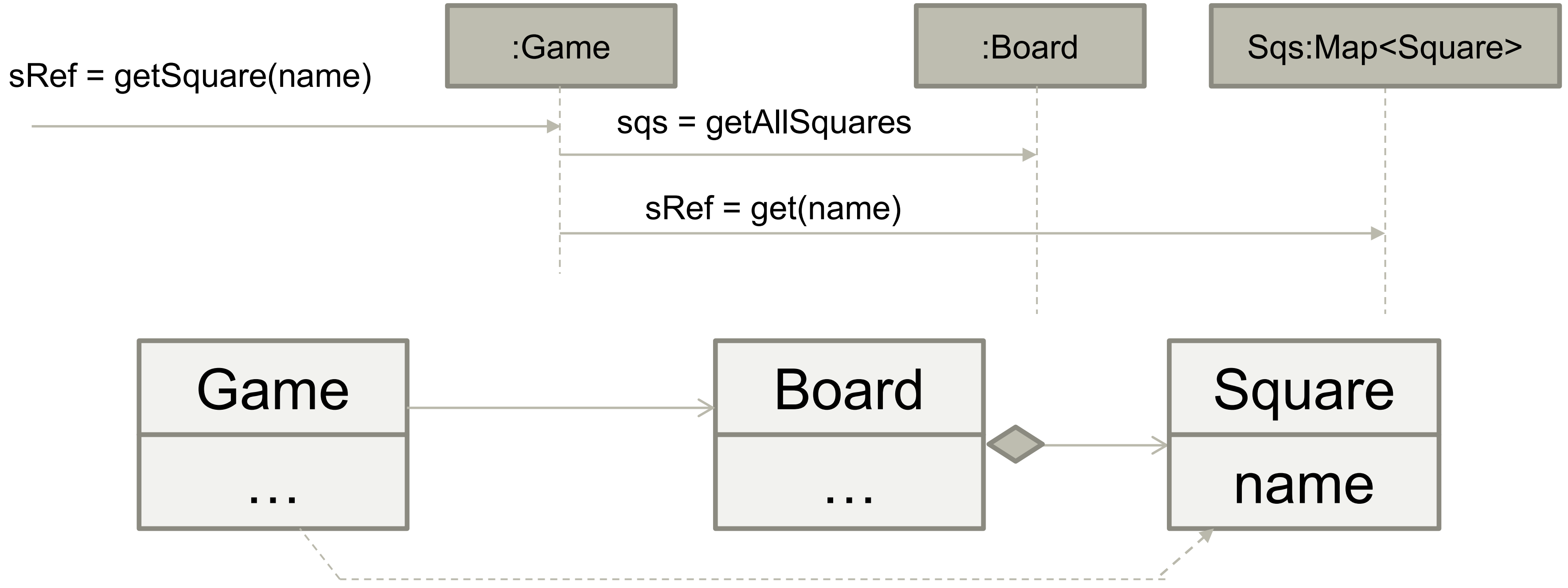
- Information Expert principle guided us to assign the responsibility of retrieving a Square reference to the Board - as it knows all about the squares.
- The basis for this principle is Low Coupling.
- Coupling measures the extent to which one object is dependent or has knowledge of other objects.
- Coupling makes it necessary to change all dependent elements when depended-upon element changes.

Impact of High Coupling

- Highly coupled classes are sensitive to ***change***
  - changing dependencies may break the coupled class
  - coupling to stable elements such as system services or APIs is less of an issue but the couples can still be isolated
- Highly coupled classes are harder to ***reuse***
  - you must reuse all the coupled classes which themselves may be further coupled
- Harder to ***understand*** in isolation
  - you need to investigate and understand all the dependencies
  - in some cases source code or documentation may be unavailable

Evaluating Alternatives





option 2 has higher coupling

Low Coupling: Discussion

- Common Forms of Coupling from TypeX to TypeY in OO languages
  - TypeX has an attribute (or collection) of TypeY instance
  - TypeX has a method that has a TypeY parameter or a TypeY local variable
  - TypeX calls method on TypeY (requires at least one of the former two)
  - TypeX implements TypeY interface
  - TypeX is a subclass of TypeY
- Coupling through an interfaces (i.e. X is an interface) is a lower form of coupling than through a class since it provides more scope for extensibility without changing dependencies
- The extreme case of low coupling is no-coupling between classes which results in bloated complex objects that do all the work. Hence, as discussed previously coupling and cohesion are a trade-off.
- Coupling will often change as you increment/iterate and your design develops! i.e. as you add new features you revise (hopefully improve!) your coupling decisions
- **TIP**: "Choose your battles" i.e. focus effort on reducing coupling at known/suspect change points (e.g. is a different UI or persistence mechanism likely, will business logic change etc.)

High Cohesion

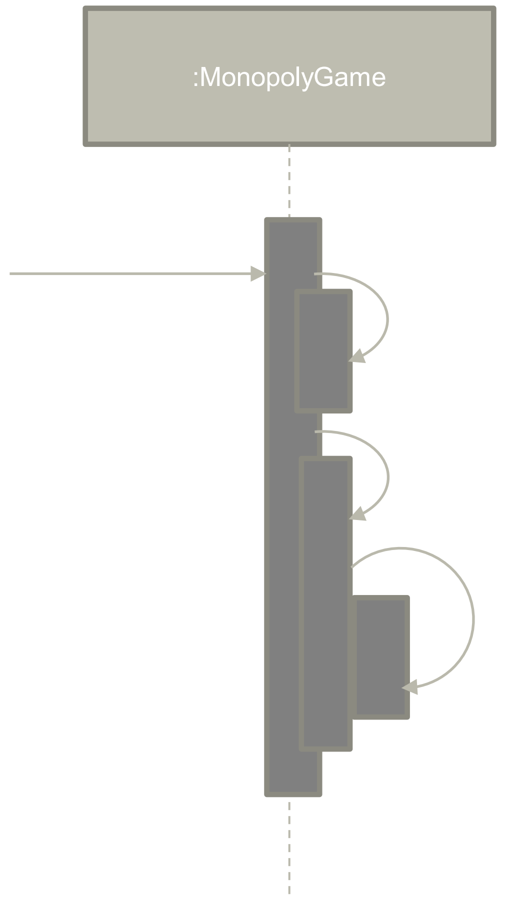
Name	High Cohesion
Problem:	How to keep objects focused and manageable and as a side effect support low coupling*?
Solution:	Assign responsibilities such that cohesion remains high. Use this principles to evaluate alternatives.

* This might seem unintuitive but how can high cohesion reduce coupling (since in general it can increase (i.e. is a trade-off with) coupling)

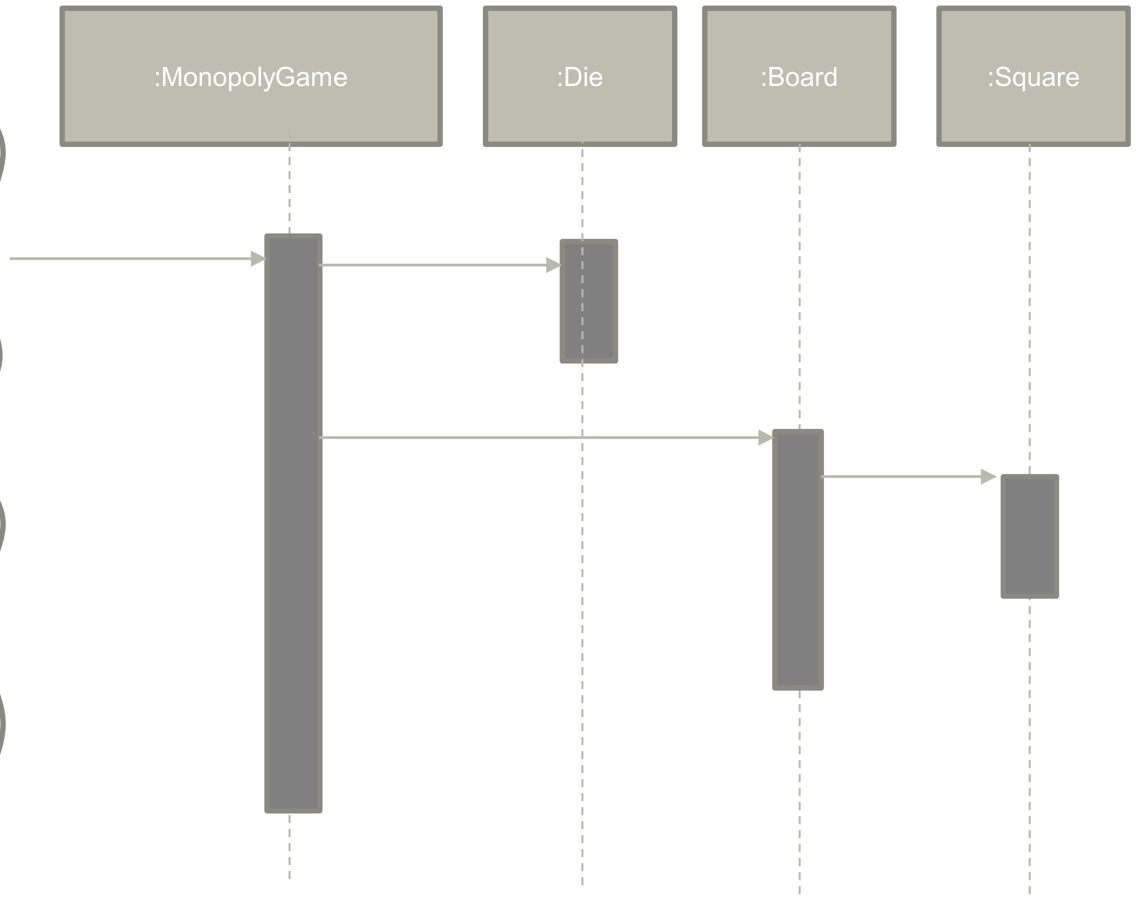
High Cohesion

- A crude approximation of cohesion can be related to how much work it is doing (lines of code)
  - consider a class *Big* with 200 methods and 5000 lines of code, compared to *Small* with 10 methods and 200 lines of code
  - it is likely that *Big* covers more different areas of responsibilities and is thus less cohesive than *Small*.
- However more correctly: *cohesion is the measure of the extent to which elements of a code module (usually a class) perform a clear delineated function*
- This requires semantic (i.e. conceptual) judgment and domain knowledge to determine the relatedness (i.e. cohesion) of attributes and methods
- However a good measureable indicator is the extent to which methods of a class use the attributes
  - i.e. if attributes are only used by a few methods they are candidates to split into a separate class, whereas if all methods use all attributes they are more likely to be related

Low cohesion



High cohesion



High Cohesion: Discussion

- Cohesion promotes comprehensibility/understandability (and thus maintainability) and reusability
- A class with high cohesion has a relatively small number of methods, with highly related functionality and collaborates with other objects to accomplish large tasks
- Real-world analogy: A person who takes on too many unrelated responsibilities without delegating them.

Controller

Name	Controller
Problem:	What first object beyond UI layer receives and coordinates a system operation?
Solution:	Assign responsibility to an object representing one of these • Represent the overall “system”, a “root object”, a device that the software is running within, or a major subsystem. • Represent a use case scenario within which the system operation occurs.

Controller (A GRASP pattern related to MVC)

- A basic design principle is that UI functionality should be separate from domain functionality.
- In a system with a GUI, Actors (human) generate UI events (e.g. button clicks) which trigger specific functionality
- Actors need not be human and interaction events can come from various sensors, other system components etc.
- MVC (Model View Controller pattern) specifies UI objects should not contain “business logic”. Therefore these tasks must be delegated to a *controller*

Model View Controller

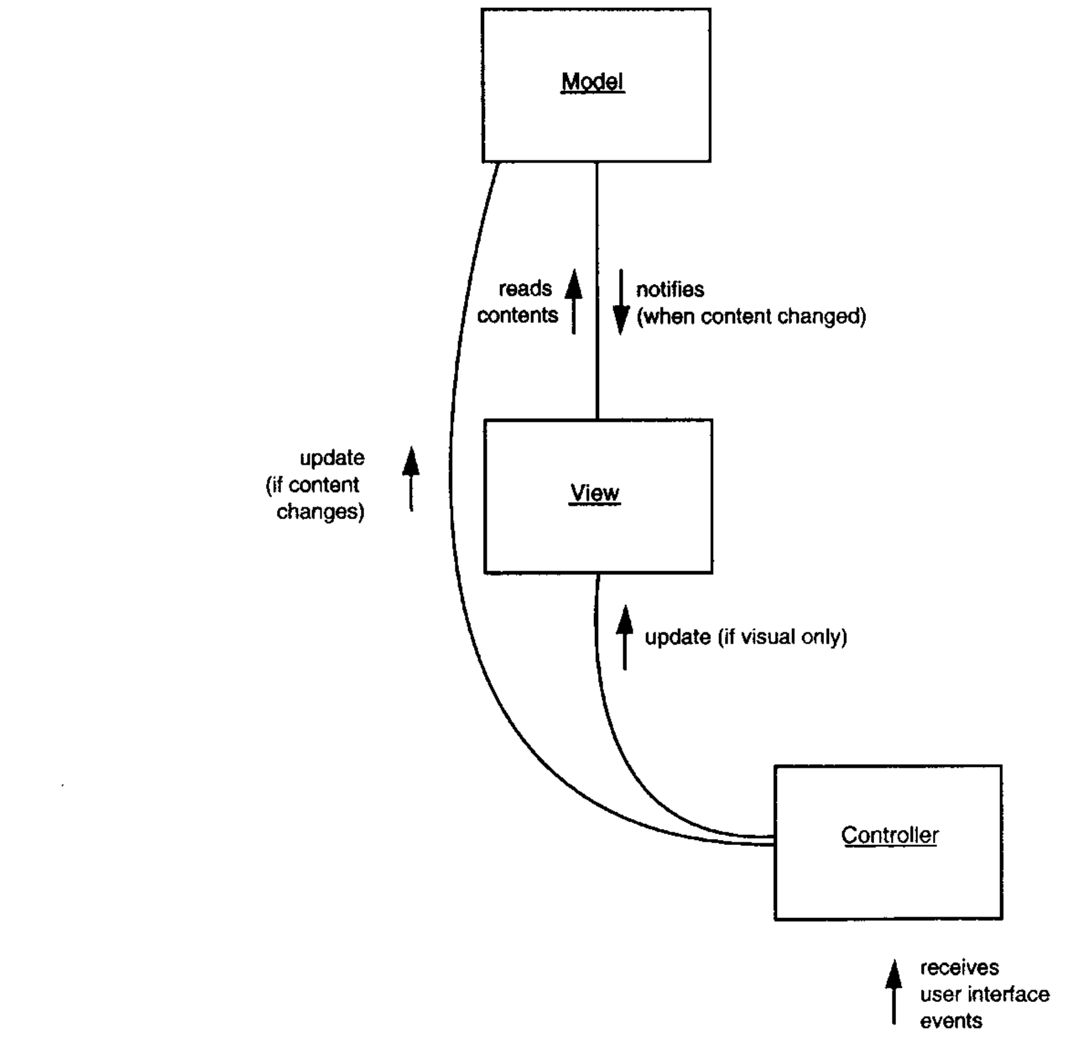

Model
reads contents
notifies (when content changed)
update (if content changes)
View
update (if visual only)
Controller
receives user interface events

Figure 9–3: Interactions between model, view, and controller objects

Model View Controller

Model: contains application code such as:

- data structures
- accessor methods
- I/O routines
- calculations etc.

View: the representation of the data i.e. how it is presented to the user

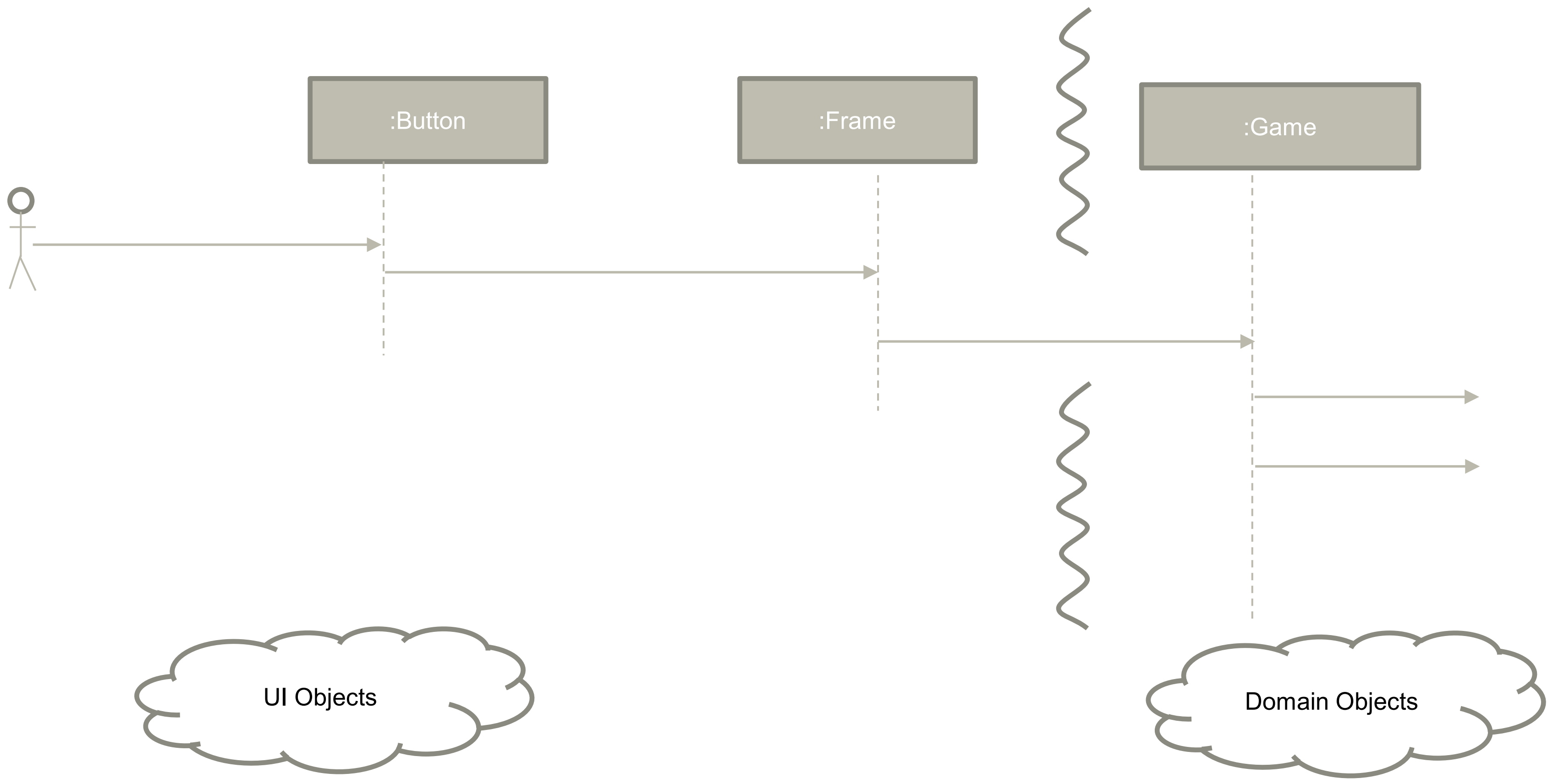
- (may be devices other than simple display terminal e.g. accessibility devices for disabled persons)
- multiple views of the same data (e.g. a spreadsheet has cell and chart views)

Controller: handles user interaction and mediates between the Model (data) and The View (representation)

- e.g user drags mouse in a drawing program. The controller modifies the series of points or vectors (data structures) in the model as the mouse is moved (user interaction) and instructs the view to redraw the data in the model (alternatively the model may automatically instruct it's registered views to redraw its data - example of Observer pattern [Design Patterns, Gamma et al. 1995])

- ***Consider Microsoft Excel as an Example***

Controller Layer



Controller Discussion

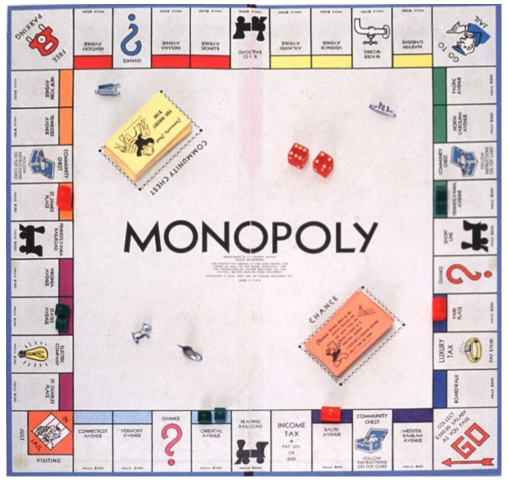
- The controller itself should do minimal work and delegate execution to the appropriate domain classes (model)
- A single controller (i.e. Façade or “Front Controller”) can be suitable when there are few system events to be handled
  - or where there is an overhead in issuing requests (e.g. across a slow network connection)
- For a local controller (e.g. listeners with a Swing/JavaFX interface) let cohesion guide you to use many small controllers
- A “use case controller” or session controller is useful where a request has a number of steps that may be interrupted (e.g. online shopping)
  - here the controller manages session state but still forwards work to the model
  - state machine implementation can be useful (covered later in semester)
  - may choose to forward state persistence to the model

Polymorphism

- How to handle alternative behaviour based upon type (interface or class in Java)
- How to create pluggable software components (i.e. For example, can one server component be replaced with another?)
- The functional programming mechanism for handling type-based alternatives is with conditionals:

Consider the Monopoly Game:

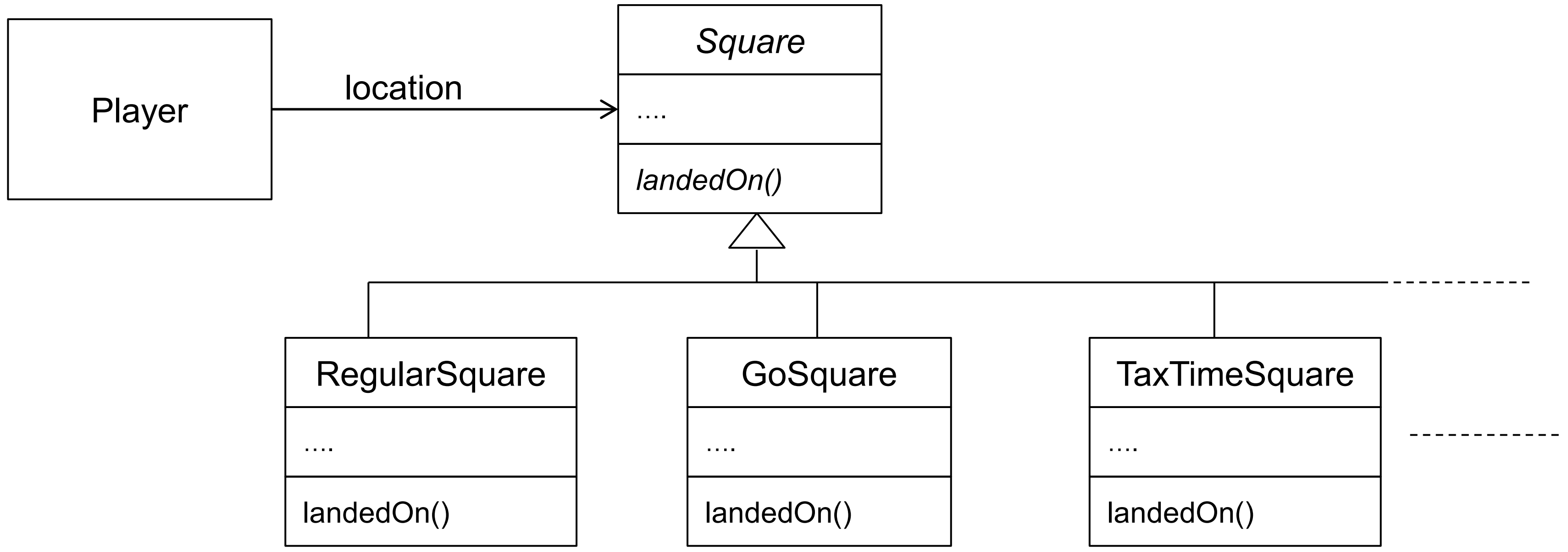
```
Switch (square.type)
{
   case GoSquare: …. // receive $200
   case RegularSquare: … // nothing
   case TaxTimeSquare: … // deduct tax
```



Poor cohesion and breaks information expert!

Polymorphism

- The OO approach is to create a polymorphic operation for each type for which the behavior varies.
- The behavior varies when a player lands on different types of squares
- A polymorphic operation can therefore be named *landedOn*
- *This agrees with both Information Expert and Cohesion patterns*

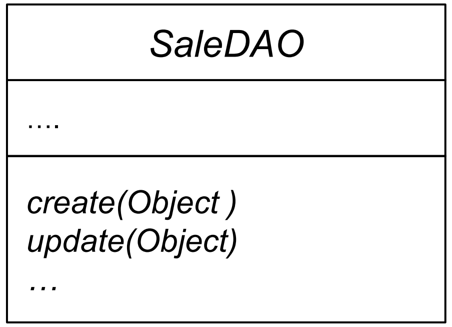


Pure Fabrication

- Many classes represent things in the problem domain, and so are derived/named from the former
- Sometimes it is useful to group methods by a behavior or algorithm, even if the resulting class doesn’t have a real-world representation
  - domain class has too many responsibilities
  - especially related to non-functional requirements or other cross cutting concerns (e.g. security, reliability, persistence etc.)
- In GRASP this is called ***Pure Fabrication*** (since we create a class that does not have a real-world corollary/representation)
- Assign a cohesive set of responsibilities to a convenience class that does not represent a domain object, but which supports high cohesion, low coupling, and reuse.
- **NOTE:** Be careful not to decouple the fabrications from their dependent information (i.e. the opposite of information expert!)

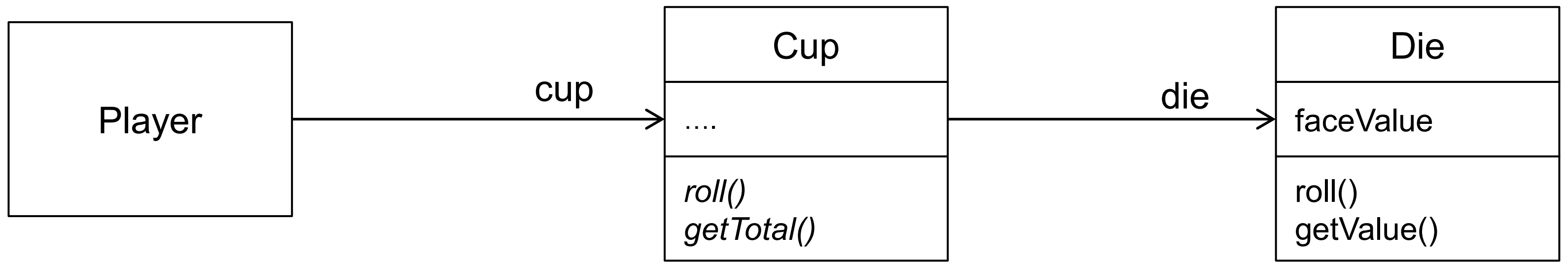
Pure Fabrication

- Consider Persistence or Database operations:
- Saving a *Sale* object might, by Information Expert, belong in the Sale class
- But the task requires a relatively large number of database-oriented operations, none related to the concept of sale-ness.
- Using a "fabricated" Data Access Object (DAO) class increases the cohesion in *Sale* and delegates complex coupling to the persistence engine (though it is still coupled to DAO)
- A DAO is not a domain concept. A businessperson or store attendant may have no idea what a DAO is.



Another Example

- In Monopoly, one could have a *Cup* object that has properties such as the number of dice and methods to roll them and retrieve the total
- If you're creating games, a cup is generally useful, so this class could be reused



BUT IS THIS REALLY PURE FABRICATION?

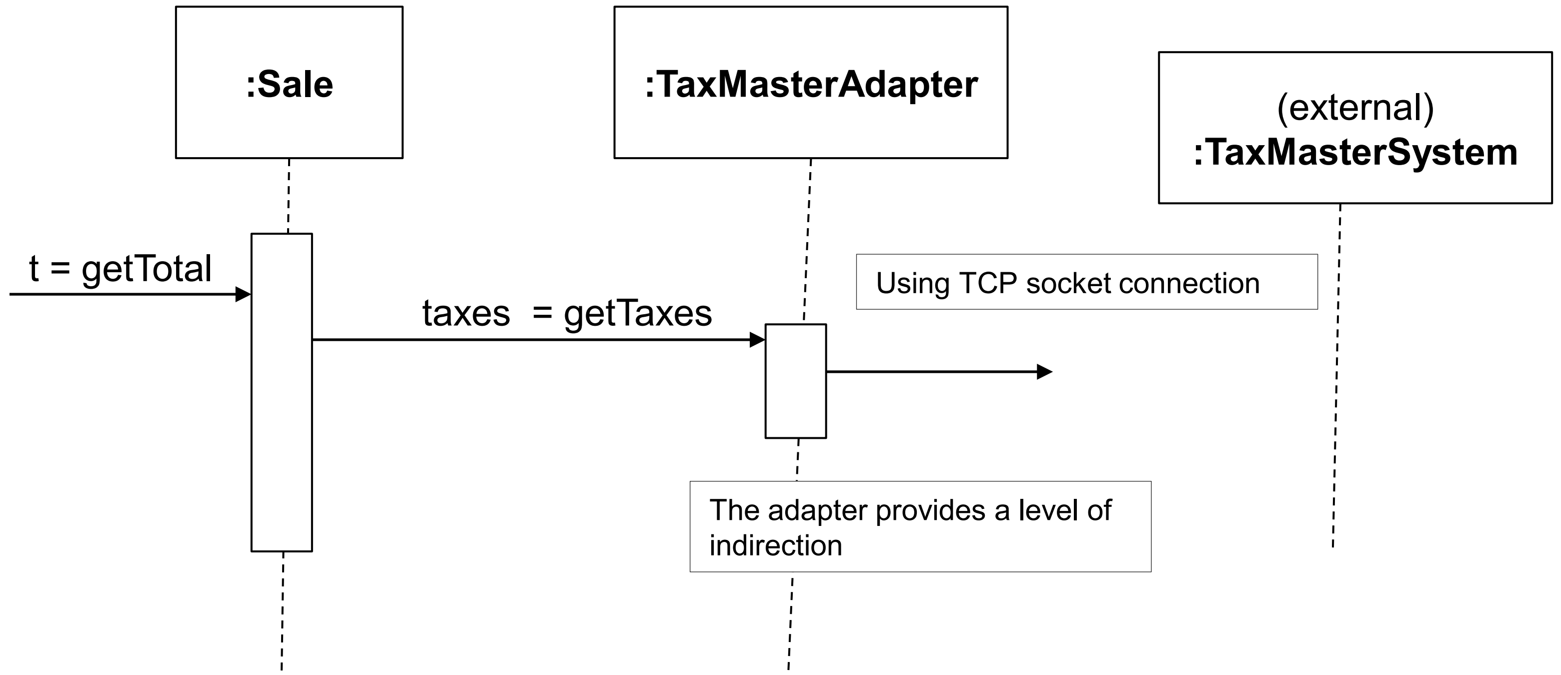
- Many existing design patterns are examples of Pure Fabrication (these GoF examples are covered later in the semester):
  - Adapter
  - Strategy
  - Command

Indirection

- Problem is where to assign responsibility to avoid direct coupling between two or more things?
- How to de-couple objects so that low coupling is supported and the chance of reuse is increased?
- Useful if you start to find that everything seems coupled to everything else!
- Solution is to assign the responsibility to an intermediate object to mediate between other components and services so that they are not directly coupled.
- The intermediary creates an indirection between the other components.
- The previous SaleDAO fabrication is also an example of indirection since it provides an interface (i.e level of indirection) to the datastore.
- The GoF *Mediator* pattern covered later in the semester is another good example of indirection

Indirection using an Adapter

- Consider a Sale object relying on external services to compute the relevant taxes (GST, payroll, …) – which changes with new legislation.
- An Adapter class can protect the inner design of methods against variations in external interfaces.
- UI classes with interdependent widgets that are enabled/disabled dynamically are good candidates for an Indirection class to handle the coupling



Protected Variations

- Problem : How to design systems, subsystems and objects so that the variations or instability in these elements does not have an undesirable impact on other elements
- Solution: identify points of predicted variation or instability; assign responsibilities to create a stable point of extension for them
- In the external tax calculator problem the point of instability or variation is the different APIs for external tax calculators
- Mechanisms motivated by Protected Variations:
- **Data Driven Designs**
  - place data in external representations that can be readily changed
  - e.g. style sheets, XML, language translation files, media/design artifacts, configuration and properties files, file paths etc.

Protected Variations

- **Naming/Service Lookup**
  - instead of referring to concrete components or class instances directly, refer to them via a reference to a name in a director/lookup service
  - LDAP, RMI and web services use this approach
- **Interpreter driven designs**
  - The system contains some form of rule engine or scripting support that can execute external scripts that change in response to behavioral/requirements change
  - anything from simple regular expression or arithmetic operations to complex embedded languages such as Python or Lua
  - workflow engines
- **Virtual machines**
  - this goes beyond programming/design and is more related to packaging and deployment

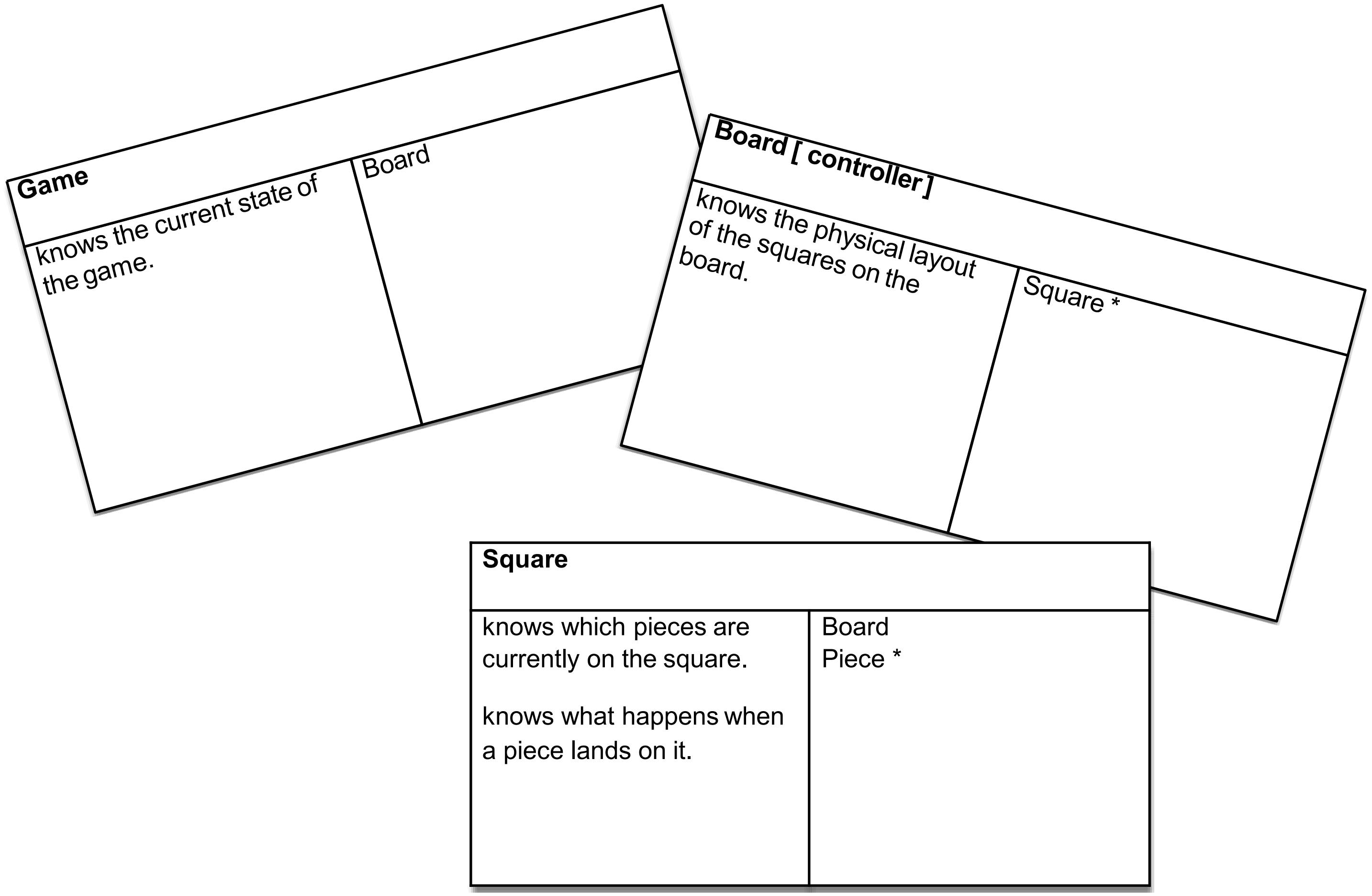
CRC Cards

Class – Responsibility – Collaboration
OO Iterative Design Brainstorming

CRC Cards

- **Class**
  - Name the class to reflect the responsibility of objects of this class.
  - e.g. Customer [Information Expert]
- **Responsibility**
  - Describe the responsibility in detail **in English**
  - e.g. remembers the name, address and telephone
- **Collaboration**
  - Refer (by name) to other classes that this class collaborates with
  - e.g.
    - Calls a method of the other class
    - Contains objects of the other class
    - Is contained in the other class.
    - Associated to objects of the other class
  - Multiplicity (for relationships)
  - Role (for method calls)
- (**back of the card)**
  - If you like to design implementation details in advance
  - List some candidate attributes and types

CRC Cards


Game
knows the current state of the game.
Board
Board [controller]
knows the physical layout of the squares on the board.
Square *
Square
knows which pieces are currently on the square.
knows what happens when a piece lands on it.
Board
Piece *

Summary

- How to become a good designer?
  - Familiarity with design principles and patterns
    - ❑ ***SOLID principles***
    - ❑ ***GRASP***
    - ❑ Refactoring Patterns
    - ❑ Design by Contract
    - ❑ GoF Patterns (5 weeks)
- Learnt through experience both positive and negative
    - ❑ through a major group assignment
  - ❑ Other nuggets of wisdom:
    - ❑ Empathy – code for others to read, reuse and modify with ease
    - ❑ Futurist – design for tomorrow and implement for today