CS 619 Introduction to OO Design and Development

GRASP Patterns

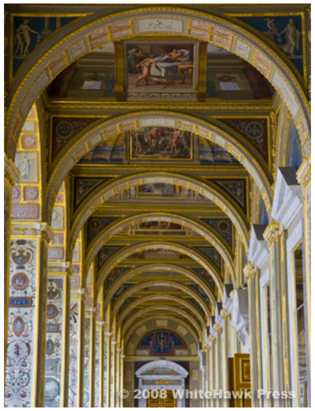
Fall 2012

- Learn about design patterns
- Learn how to apply five GRASP patterns

- You' ve learned about static class diagrams and dynamic interaction diagrams
- UML is just notation; now you need to learn how to make effective use of the notation
- UML modeling is an art, guided by principles

Design patterns in architecture

- A *pattern* is a **recurring** solution to a standard problem, in a **context**.
- Christopher Alexander, professor of architecture…
  - *Why is what a prof of architecture says relevant to software?*
  - "A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."



© 2008 WhiteHawk Press

Patterns in engineering

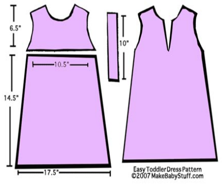
- *How do other engineers find and use patterns?*
  - Mature engineering disciplines have handbooks describing successful solutions to known problems
  - Automobile designers don't design cars from scratch using the laws of physics
  - Instead, they reuse standard designs with successful track records, learning from experience
- *Should software engineers make use of patterns? Why?*
  - Developing software from scratch is also expensive
  - Patterns support reuse of software architecture design

Definitions

- In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design.
- Larman: “In OO design, **a pattern is a named description of a problem and solution that can be applied in new contexts;** ideally, a pattern advises us on how to apply the solution in varying circumstances and considers the forces and trade-offs.”

Definitions

- Patterns have suggestive names:
  - Arched Columns Pattern, Easy Toddler Dress Pattern,Star and Plume Quilt etc.
- Why?
  - It supports chunking and incorporating that concept into our understanding and memory
  - It facilitates communication



Design Patterns

- A design pattern is not a finished design that can be transformed directly into code.
- It is a description or template for how to solve a problem that can be used in many different situations.
- OO design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

GRASP

- Name chosen to suggest the importance of **grasp**ing fundamental principles to successfully design object-oriented software
- Acronym for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
- Describe fundamental principles of object design and responsibility
- Expressed as patterns

Five GRASP patterns:

- Creator
- Information Expert
- Controller
- Low Coupling
- High Cohesion

GRASP Creator

Name: **Creator**

Problem: Who creates an instance of A?

Solution: Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):

- B contains or aggregates A (in a collection)
- B records A
- B closely uses A
- B has the initializing data for A

Who creates the Squares?

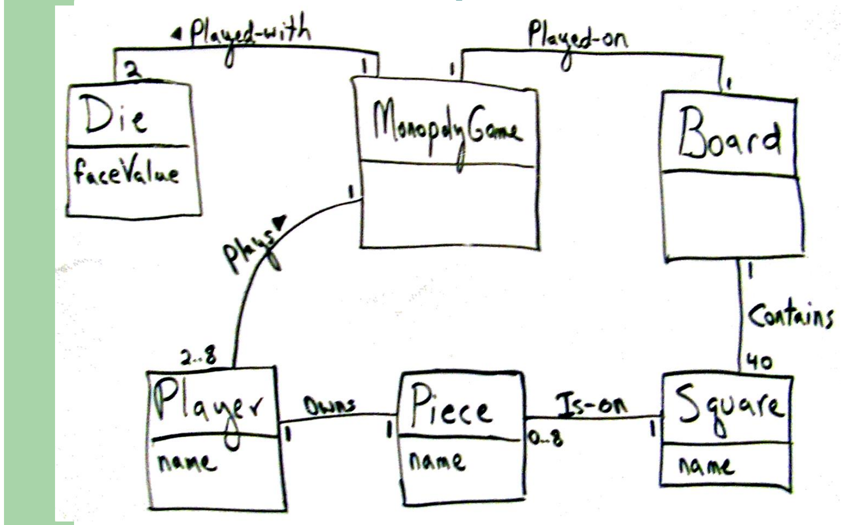


Figure 17.3, page 283

How does Create pattern lead to this partial Sequence diagram?

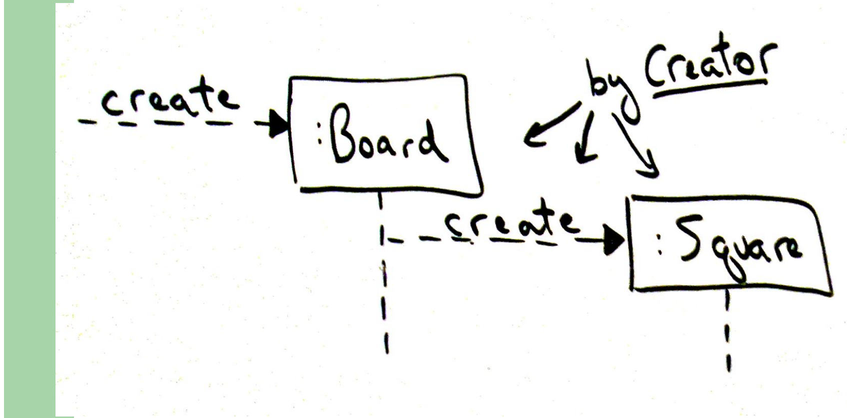


Figure 17.4, page 283

How does Create pattern develop this Design Class Diagram (DCD)?

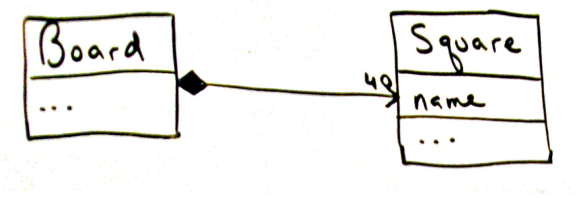
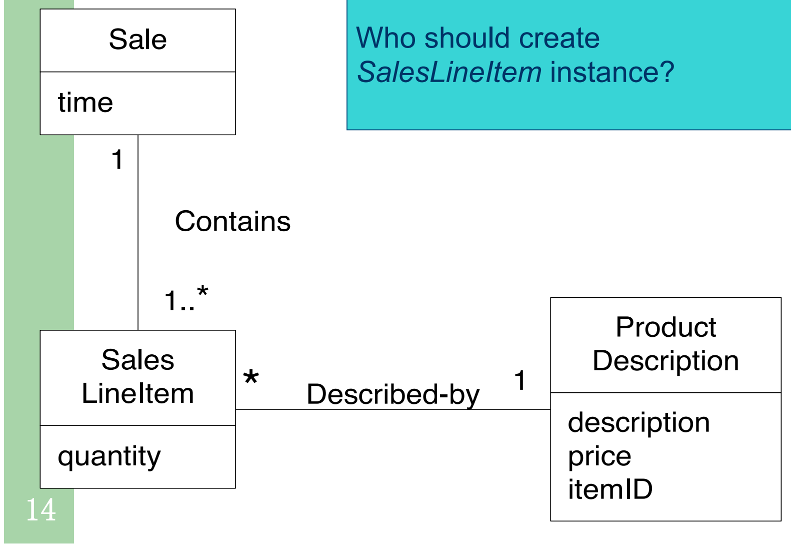


Figure 17.5 , page 283

Board has a composite aggregation relationship with *Square*
• I.e., Board contains a collection of Squares

Creator example


Sale

time

1

Contains

1..*

Sales LineItem

quantity

*

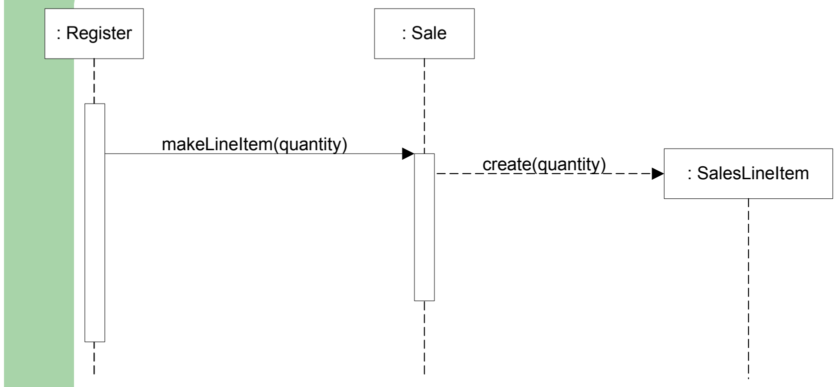
Described-by

1

Product Description

description
price
itemID

Who should create *SalesLineItem* instance?


: Register

: Sale

makeLineItem(quantity)

create(quantity)

: SalesLineItem

Discussion of Creator pattern

- Responsibilities for object creation are common
- Connect an object to its creator when:
  - Aggregator aggregates Part
  - Container contains Content
  - Recorder records
  - Initializing data passed in during creation

Contraindications or caveats

- Creation may require significant complexity:
  - recycling instances for performance reasons
  - conditionally creating instances from a family of similar classes
- In these instances, other patterns are available…
  - We'll learn about Factory and other patterns later…

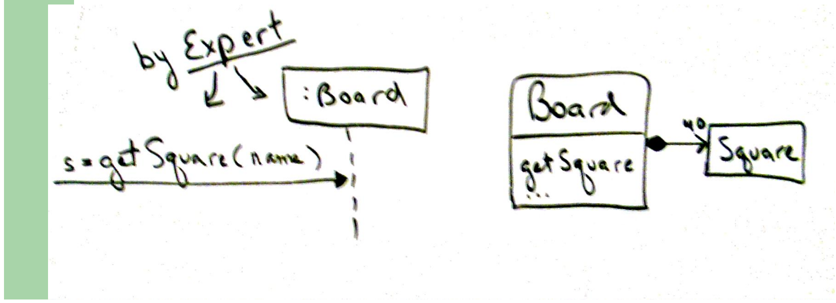
GRASP: Information Expert Pattern

Name: Information Expert

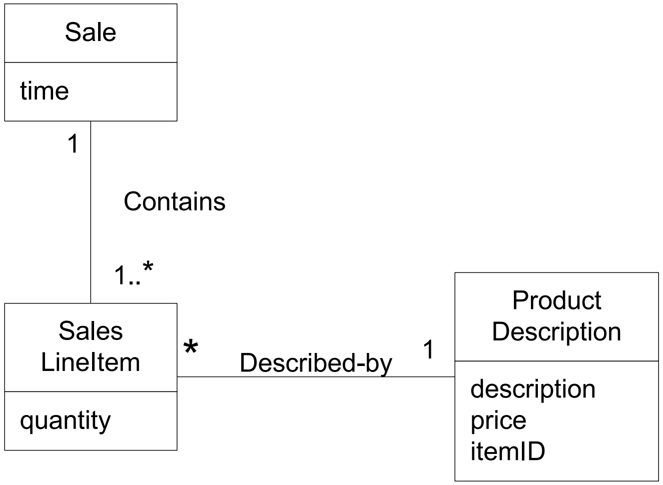
Problem: How to assign responsibilities to objects?

Solution: Assign responsibility to the class that has the information needed to fulfill it.

- E.g., Board information needed to get a Square



Information Expert Example

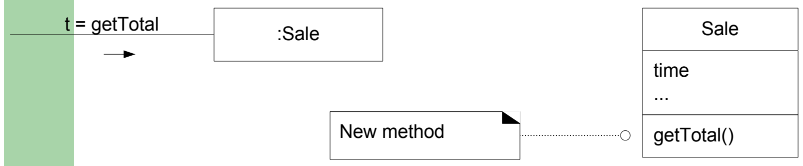


Information Expert Example

- *Who should be responsible for knowing the grand total of a sale?*
- By Information Expert, we should look for that class of objects that has the information needed to determine the total.
- *Do we look in the Domain Model or the Design Model to analyze the classes that have the information needed?* Answer:
  - If there are relevant classes in the Design Model, look there first.
  - Otherwise, look in the Domain Model, and attempt to use (or expand) its representations to inspire the creation of corresponding design classes

Information Expert Example

- What information do we need to determine the grand total?
- We need to know about all the *SalesLineItem* instances of a sale and the sum of their subtotals.
- A Sale instance contains these; therefore, by the guideline of Information Expert, *Sale* is a suitable class of object for this responsibility; it is an information expert for the work.

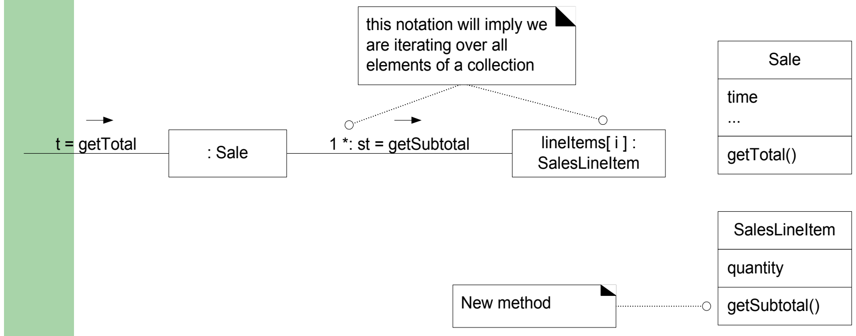


Information Expert

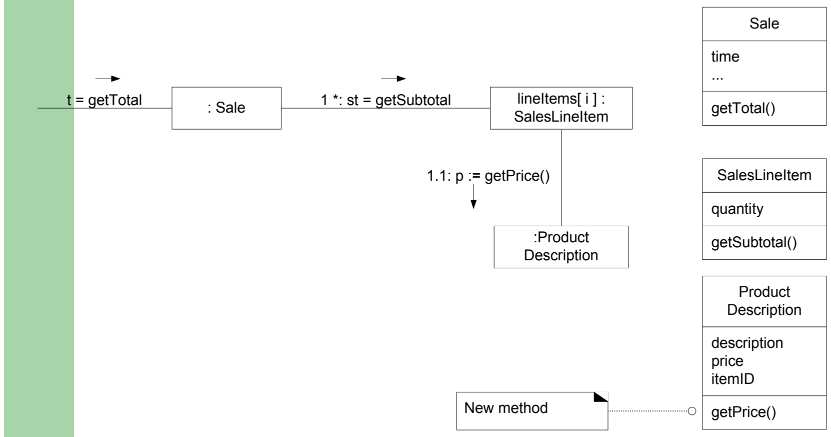
- Not done yet!
- What information do we need to determine the line item subtotal?
  - SalesLineItem.quantity and ProductDescription.price.
- The *SalesLineItem* knows its quantity and its associated *ProductDescription*; therefore, by Expert, *SalesLineItem* should determine the subtotal; it is the information expert.
- This means that

  the *Sale* should send *getSubtotal* messages to each of the *SalesLineItems* and sum the results.
- This design is shown on next page.

Information Expert

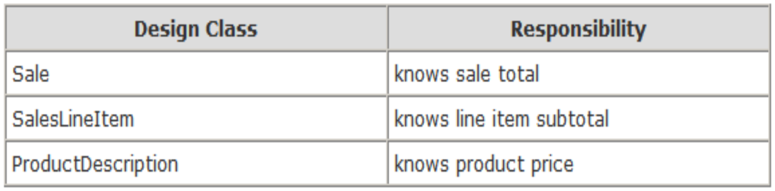


Information Expert



Information Expert

- In conclusion, to fulfill the responsibility of knowing and answering the sale's total, we assigned three responsibilities to three design classes of objects as follows.



Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductDescription	knows product price

- We considered and decided on these responsibilities in the context of drawing an interaction diagram.

Benefits and Contraindications

- Facilitates information encapsulation: *why?*
  - Classes use their own info to fulfill tasks
- Encourages cohesive, lightweight class definitions

But:

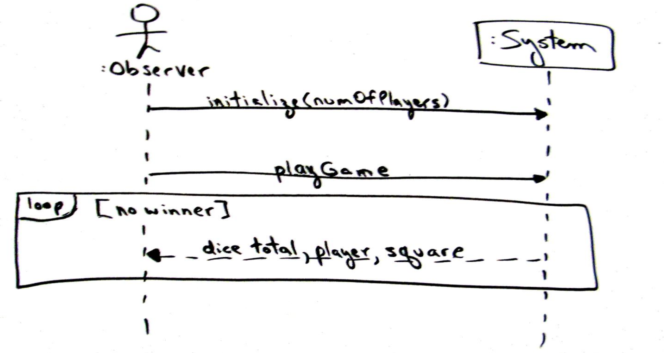
- Information expert may contradict patterns of Low Coupling and High Cohesion
- Remember separation of concerns principle for large sub-systems
- I.e., keep “business” or application logic in one place, user interface in other place, database access in another place, etc.

GRASP: Controller Pattern

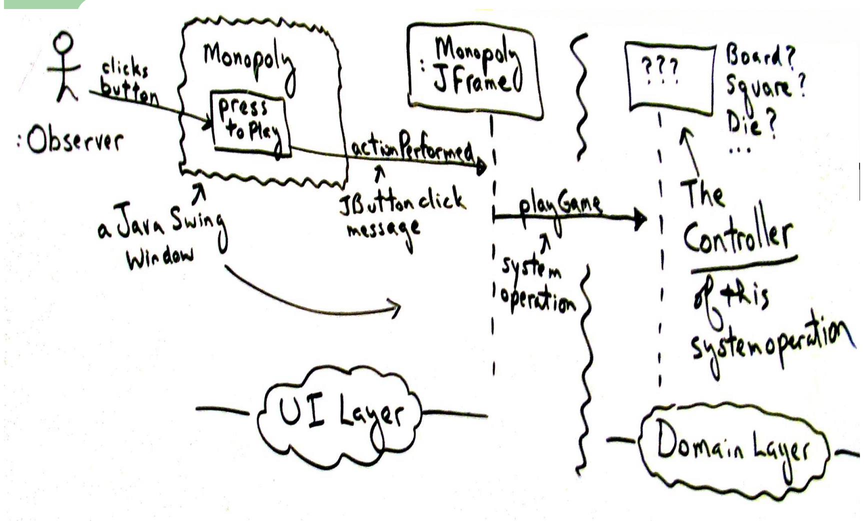
- A simple layered architecture has a UI layer and a domain layer, among others.
- From the Model-View Separation Principle, we know the UI objects should *not* contain application or "business" logic such as calculating a player's move.
- Therefore, once the UI objects pick up the mouse event for example, they need to delegate (forward the task to another object) the request to domain objects in the domain layer.

GRASP: Controller

- The Controller pattern answers this simple question: *What first object after or beyond the UI layer should receive the message from the UI layer?*



GRASP: Controller



GRASP: Controller

- Controller deals with a basic question in OO design: How to connect the UI layer to the application logic layer?
  - Should the Board be the first object to receive the playGame message from the UI layer? Or something else?

GRASP: Controller

- Name: Controller
- Problem: What first object beyond the UI layer receives and coordinates ("controls") a system operation?
- Solution: (advice)

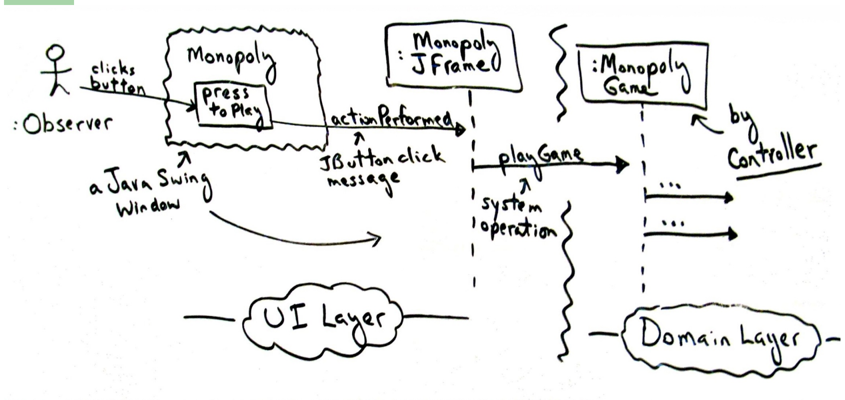
  Assign the responsibility to an object representing one of these choices:
  - Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem.
  - Represents a use case scenario within which the system operation occurs (a use case or session controller)

GRASP: Controller

Let's consider these options:

- Option: Represents the overall "system," or a "root object“ such as an object called MonopolyGame.
- Option: Represents a device that the software is running within.
  - This option appertains to specialized hardware devices such as a phone or a bank cash machine (e.g., software class Phone or BankCashMachine); it doesn't apply in this case.
- Option : Represents the use case or session. The use case that the playGame system operation occurs within is called Play Monopoly Game. Thus, a software class such as *PlayMonopolyGameHandler* (appending "…Handler" or "…Session" is an idiom in OO design when this version is used).

GRASP: Controller



GRASP: Low Coupling

Problem

- How to support low dependency, low change impact, and increased reuse?
- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
- An element with low (or weak) coupling is not dependent on too many other elements.

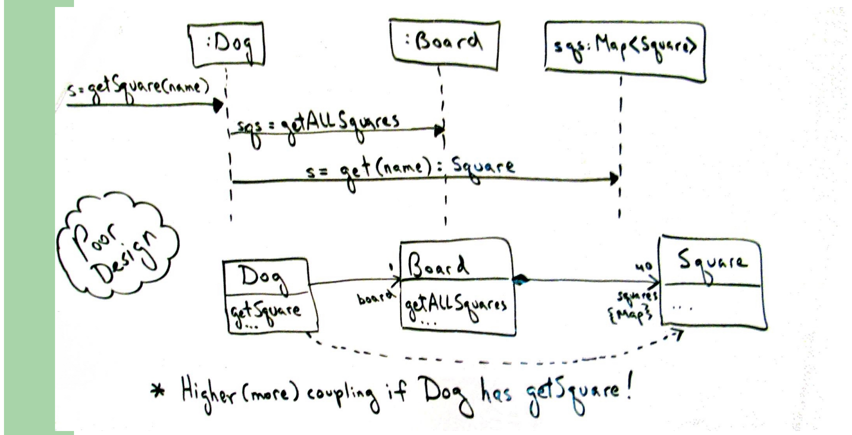
Low Coupling

- A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems:
  - Forced local changes because of changes in related classes.
  - Harder to understand in isolation.
  - Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

Solution

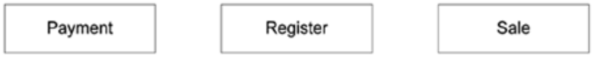
- Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

Why does the following design violate Low Coupling?



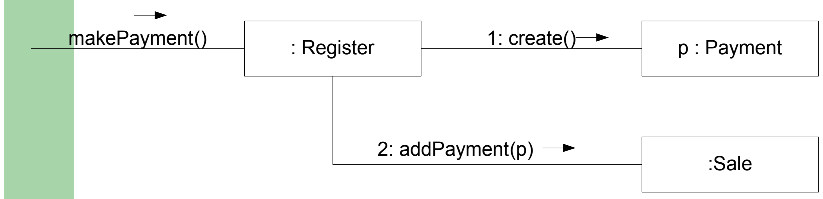
Why is a better idea to leave getSquare responsibility in Board?

Low Coupling example



- Assume we need to create a Payment instance and associate it with the Sale. What class should be responsible for this?
- Since a Register "records" a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment.
- The Register instance could then send an *addPayment* message to the Sale, passing along the new Payment as a parameter.

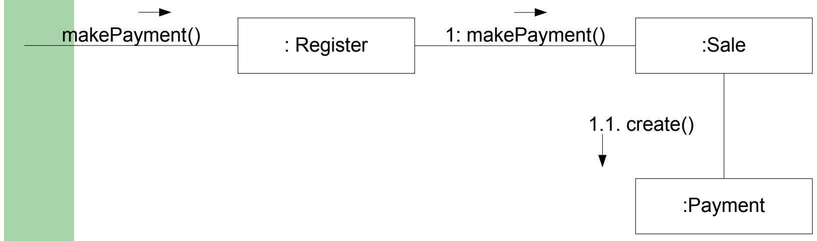
Low Coupling



- This assignment of responsibilities couples the Register class to knowledge of the Payment class.

Low Coupling

- An alternative



Low Coupling

- In both cases we assume the Sale must eventually be coupled to knowledge of a Payment.
- Design 1, in which the Register creates the *Payment*, adds coupling of Register to Payment;
- Design 2, in which the Sale does the creation of a Payment, does not increase the coupling.
- Purely from the point of view of coupling, prefer Design 2 because it maintains overall lower coupling.
- This example illustrates how two patterns - Low Coupling and Creator- may suggest different solutions.

Low Coupling

- In object-oriented languages such as C++, Java, and C#, common forms of coupling from TypeX to TypeY include the following:
  - TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
  - A TypeX object calls on services of a TypeY object.
  - TypeX has a method that references an instance of TypeY, or TypeY itself, by any means.
    - These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
  - TypeX is a direct or indirect subclass of TypeY.
  - TypeY is an interface, and TypeX implements that interface.

Low Coupling

- Low Coupling encourages you to assign a responsibility so that its placement does not increase the coupling to a level that leads to the negative results that high coupling can produce.
- Low Coupling supports the design of classes that are more independent, which reduces the impact of change.

- Low Coupling is a principle to keep in mind during all design decisions;
- It is an evaluative principle that you apply while evaluating all design decisions.

GRASP: High Cohesion

Problem

- How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?

- In terms of object design, cohesion (or more specifically, functional cohesion) is *a measure of how strongly related and focused the responsibilities of an element are.*

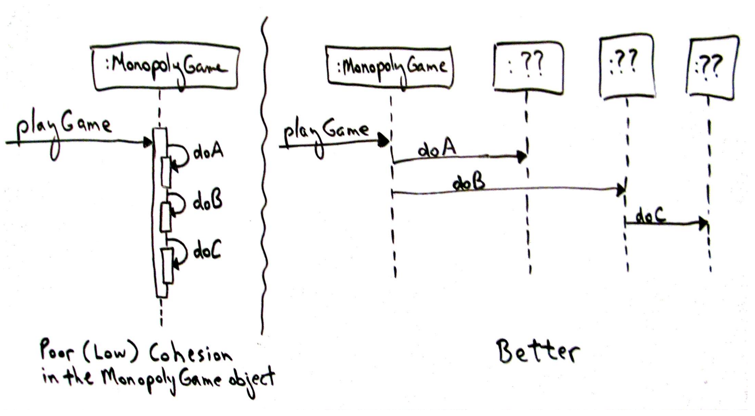
- An element with highly related responsibilities that does not do a tremendous amount of work has high cohesion. (These elements include classes, subsystems, and so on.)

GRASP: High Cohesion

- In software design a basic quality known as *cohesion* informally measures how functionally related the operations of a software element are, and also measures how much work a software element is doing.
  - As a simple contrasting example, an object Big with 100 methods and 2,000 source lines of code (SLOC) is doing a lot more than an object Small with 10 methods and 200 source lines.
  - If the 100 methods of Big are covering many different areas of responsibility (such as database access and random number generation), *then Big has less focus or functional cohesion than Small*.
- In summary, *both the amount of code and the relatedness of the code are an indicator of an object's cohesion.*

GRASP: High Cohesion

- Two design choices after deciding on controller



High Cohesion

Solution

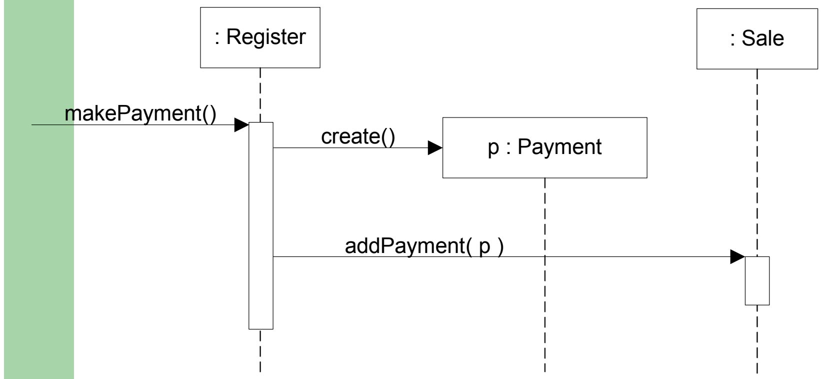
- Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

- A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:
  - hard to comprehend
  - hard to reuse
  - hard to maintain
  - delicate; constantly affected by change

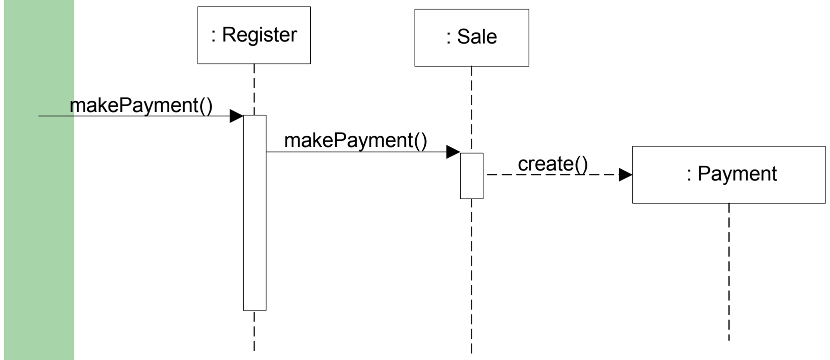
High Cohesion: Example

- Let's take another look at the example problem used in the Low Coupling pattern and analyze it for High Cohesion.
- Assume we have a need to create a (cash) Payment instance and associate it with the Sale.
- What class should be responsible for this?
- Since Register records a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment.
- The Register instance could then send an *addPayment* message to the Sale, passing along the new Payment as a parameter

High Cohesion



High Cohesion



High Cohesion

Discussion

- Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider.
- It is an evaluative principle that a designer applies while evaluating all design decisions.
- High functional cohesion exists *when the elements of a component (such as a class) "all work together to provide some well-bounded behavior".*

High Cohesion

- Here are some scenarios that illustrate varying degrees of functional cohesion:
  - Very low cohesion
    - A class is solely responsible for many things in very different functional areas.
  - Low cohesion
    - A class has sole responsibility for a complex task in one functional area.
  - High cohesion
    - A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.
  - Moderate cohesion
    - A class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept but not to each other.

High Cohesion

- Rule of thumb: *a class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work. It collaborates with other objects to share the effort if the task is large.*
- A class with high cohesion is advantageous because it is relatively easy to maintain, understand, and reuse.
  - The high degree of related functionality, combined with a small number of operations, also simplifies maintenance and enhancements. The fine grain of highly related functionality also supports increased reuse potential.

High Cohesion

- The High Cohesion pattern - like many things in object technology - has a real-world analogy.
- It is a common observation that

  if a person takes on too many unrelated responsibilities - especially ones that should properly be delegated to others - then the person is not effective.