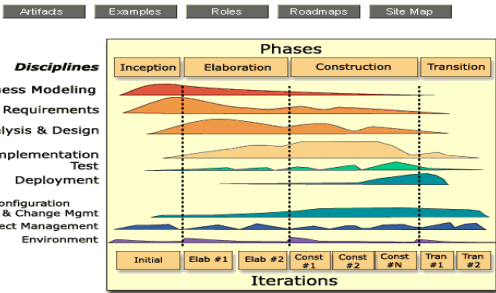


Announcements

Today: chapters 21-22

Next HW on Web by 3:00 today

Rational Unified Process: Overview



From One Iteration to the Next Iteration

- At the end of each iteration (except inception phase):
 - There is a working system
 - It has core functionality
 - (As opposed to peripheral functionality)
 - POS example
 - It can handle a sale
 - It cannot handle cashier login
- The program is also a foundation for further development
 - The analysis and design are also foundations for further development

GRASP Revisited

- Chapter 22 discusses
 - GRASP (again)
 - There are other GRASP patterns besides the ones we've seen
 - First, let's recap our previous examination of the topic

The GRASP Concept

- **GRASP** stands for
 - G**eneral
 - R**esponsibility
 - A**ssignment
 - S**oftware
 - P**atterns
- The mnemonic idea is, successful OOA/D requires "grasping" (understanding) GRASP
 - What do the parts mean?

GRASP Concept (cont.)

General Responsibility Assignment Software Patterns

- General = Abstract; **widely applicable**
- Responsibility = **Obligations, duties**
- Assignment = **Giving a responsibility to a module**
- Software = **Computer code**
- Patterns = Regularities, templates, **abstraction**
- How would you parse the GRASP phrase?
 - [General [Responsibility Assignment]] [Software Patterns]
- Let's look at these terms in more detail

GRASP Responsibilities: Knowing and Doing

- 1. Understand member data
 - 1. Perform a directly useful action
 - Change some data
 - (Assign, calculate, create an object,...)
- 2. Understand what you (an object) can do yourself
 - 2. Initiate actions in other objects
- 3. Understand related objects
 - 3. Control/coordinate other objects
 - They might change some data
 - If they don't, what do they do?
 - Analogy: top brass, middle brass, and everyone else

Match blue dots on right to blue dots on left!

Object Responsibilities

- Notice the relevance of ideas on last slide to
 - Designing classes, including
 - Their data
 - Their methods
 - Their interactions with (messages to) other objects

Responsibilities and Diagrams

- Review:
 - *Collaboration Diagrams*
 - *Sequence Diagrams*
 - *Interaction Diagrams*
 - Which is the superset?
 - Do the others give different information?
- Also recall:
 - *Class Diagrams*
 - *Domain models (domain class diagrams)*
 - *Design class diagrams*
 - What are the relationships among those?
- These diagrams help greatly in guiding responsibility assignment

GRASP: Software Patterns

- **Software Pattern:** is it more like
 - *a template or abstract design which you fill out the details of to get a specific design*
 - *a general solution strategy that helps guide design development*
- Patterns should have names
 - ...to make them easier to think/talk about
- A pattern should come with
 - Suggestions for how to apply it
 - Explanation of its trade-offs

A Few GRASP Patterns

- We've looked at:
 - High Cohesion pattern
 - Low Coupling pattern
 - Information Expert pattern
 - Creator pattern
 - Controller pattern

GRASP Patterns

- High cohesion pattern, low coupling pattern, information expert pattern, creator pattern, or controller pattern?
- Applying this pattern assigns responsibilities to minimize interaction

GRASP Patterns

- High cohesion pattern, low coupling pattern, information expert pattern, creator pattern, or controller pattern?
- Example: consider Sale, SalesLineItem, and ItemNumber classes
 - Which should be *assigned responsibility* for getting the grand total?
 - Asking/answering this question = applying which pattern?
- Which pattern says to assign computations to the class that has the needed input information?



GRASP Patterns

- High cohesion pattern, low coupling pattern, information expert pattern, creator pattern, or controller pattern?
- **Assigning certain responsibilities to objects:**
 - Example: Sale, SalesLineItem, and ItemNum classes
 - Which should construct **SalesLineItems**?
 - Why?
 - Which pattern applies here?



GRASP Patterns

- High cohesion pattern, low coupling pattern, information expert pattern, creator pattern, or controller pattern?
- **Assigns responsibility for handling input events to appropriate objects**
- Example: cashier login event
- Which class or object should handle that?
- Which pattern applies here?



GRASP Patterns

- High cohesion pattern, low coupling pattern, information expert pattern, creator pattern, or controller pattern?
- Applying this pattern produces objects whose parts are closely related and work together
 - Which pattern?



More GRASP Patterns

- (See chapter 22)
- Some of these are named:
 - **High cohesion pattern, low coupling pattern, information expert pattern, creator pattern, controller pattern, polymorphism pattern, indirection pattern, pure fabrication pattern, and protected variations pattern**



Polymorphism Pattern

- **polymorphism pattern**, indirection pattern, pure fabrication pattern, and protected variations pattern
- Polymorphism – having one name for several different things
- The things should be related
- Example: different constructors
- Example: Class and object



Polymorphism Pattern

- **polymorphism pattern**, indirection pattern, pure fabrication pattern, and protected variations pattern
- POS example: we don't know what 3rd party tax calculator will be used by the business that buys our POS system
- Multiple methods called, e.g., getTax() are needed
 - Pattern says to put each call in a different class
 - See Figure 22.1
 - Now, to deal with a new 3rd-party tax calculator...
 - Add a new class – minimizes need to change existing classes
- "Assign responsibility to the classes for which the behavior varies"

Pure Fabrication Pattern

- polymorphism pattern, indirection pattern, **pure fabrication pattern**, and protected variations pattern
- Support high cohesion, low coupling, etc., by grouping responsibilities into an “artificial” or “convenience” class
 - Artificial – does not reflect domain realities
 - Examples: logging class; database access class
 - Do they reflect design realities?
- How is this “artificial”? Supportive of high cohesion? Supportive of low coupling?

Indirection Pattern

- polymorphism pattern, **indirection pattern**, pure fabrication pattern, and protected variations pattern
- Avoid coupling between some objects by interposing a intermediate class
- Exampe 1: Suppose N classes must all interact
 - How many arcs? Coupling is high!
 - Now interpose an intermediate class
 - All talk only to it, it talks to all
 - How many arcs now?
- Example 2: 3rd-party sales tax calculators have their divergent APIs hidden
 - Just call <name-of-calculator>.getTax(int taxableTotal)
- Why “indirection”?

Protected Variations Pattern

- polymorphism pattern, indirection pattern, pure fabrication pattern, and **protected variations pattern**
- Example: the interface of Figure 22.1
 - It and its implementing classes exemplify the protected variations pattern
 - Sale is thus protected from worrying about
 - variations in the API of each 3rd-party calculator
- Idea of this pattern is:
 - hide technical details of
 - variations of
 - some core concept

Protected Variations Pattern

61. How to minimize the impact of change due to variations or instability of certain objects, subsystems and systems?

(<http://www.objectsbydesign.com/projects/umltest/cfw-questions-2.html>)

- A. Identify points of predicted variation or instability; assign responsibilities to create a stable interface (or protection mechanism) around them.
- B. Applying data encapsulation, interfaces, polymorphism and indirection design patterns
- C. Service Lookup is an example of protected variations because clients are protected from changes in the location of services using the lookup service.
- D. Externalizing properties in a property file.
- E. All of the above
- F. Only a and d
- *Answer is E*