Team Lead 4 Presentation

- Team Café - Kyle
- The Middle Men- Don
- Bloonagins - Chase
- ScoSoft - Austin

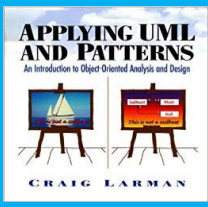
Overview

- Cohesion Vs. Coupling
- GRASP (First 5 Principles)
- Coding Standards
- Developer Manual
- Using the Gantt
- Gantt and Code Issues (The Roast)

Cohesion Vs. Coupling

Know these concepts for the oral exam and for the post mortem!

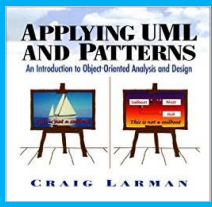
GRASP: General Responsibility Assignment Software Patterns



Jargon alert: terms and definitions

- **coupling:** the degree to which each program module relies on other modules
- **cohesion:** a measure of how well the operations in a module work together to provide a specific piece of functionality
- **encapsulation:** synonym for “information hiding” and restricting access.
- **reusability:** likelihood a segment of structured code can be used again to add new functionality with slight or no modification

Outline



- Modules
- Cohesion
- Coupling
- Encapsulation & ADTs
- Visability

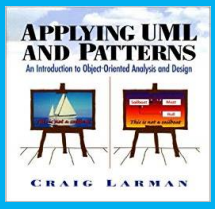
The Nine GRASP Patterns

1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

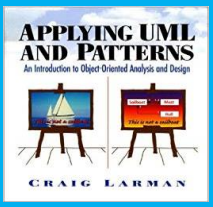
Modules

– "Lexically contiguous"
  • Connected in the code (not just related by control flow or data).
– "Boundary elements"
  • { ... }
  • begin ... end
– "Aggregate identifier"
  • A name for the entire module

- What is a module?
  - lexically contiguous sequence of program statements, bounded by boundary elements, with aggregate identifier
- Examples
  - procedures & functions in classical PLs
  - objects & methods within objects in OO PLs



Modularity is for your benefit not the computers

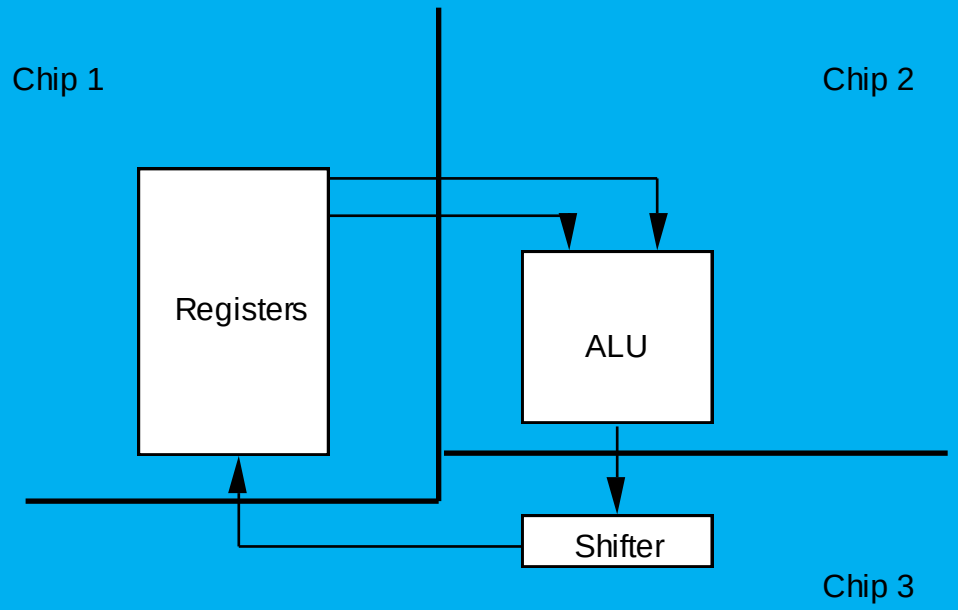


- Computers do not need code to be modularized.
  - In fact in order to run, a program must be translated into an imperative, low-level language: binary.
- People do need modularity in order to conceptually manage a large project.
  - Goal: Make the code read more like a story.
  - Ask yourself: Does this make the code easier to read and understand?
  - Does this make the logic more comprehensible or less?

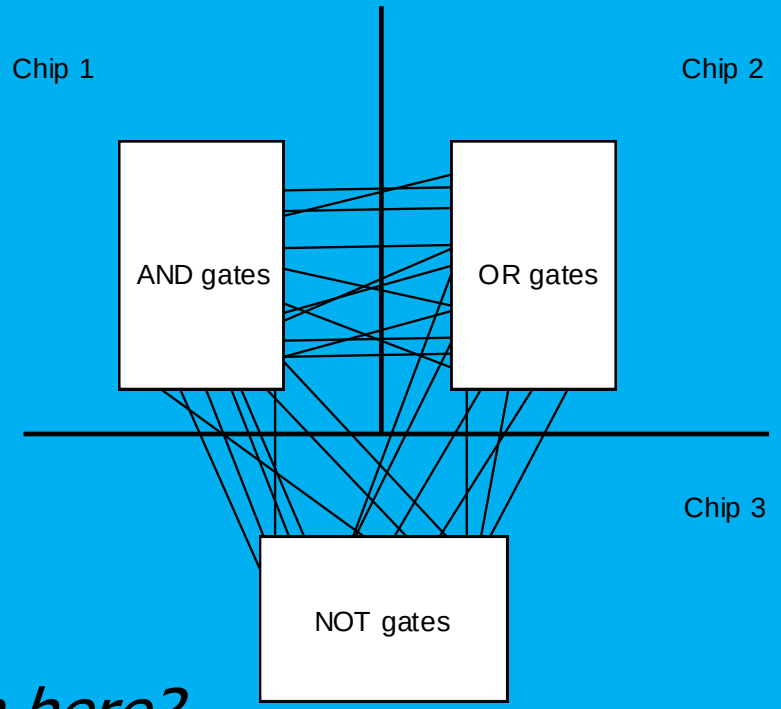
Good vs. Bad Modules

- Modules in themselves are not "good"
- Must design them to have good properties

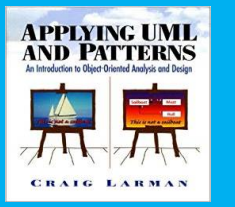
i) CPU using 3 chips (modules)



ii) CPU using 3 chips (modules)

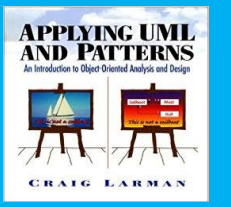


What is the problem here?



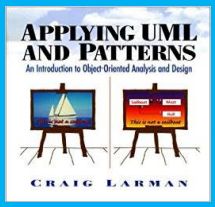
Good vs. Bad Modules

- Two designs functionally equivalent, but the 2nd is
  - hard to understand
  - hard to locate faults
  - difficult to extend or enhance
  - cannot be reused in another product. Implication?
  - -> expensive to perform maintenance
- Good modules must be like the 1st design
  - maximal relationships within modules (cohesion)
  - minimal relationships between modules (coupling)
  - this is the main contribution of structured design



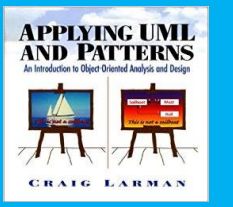
Don’t over-do it

- Don’t stifle creativity.
  - Make informed, deliberate decisions on when and where to apply modularization.
  - Ask: How will this impact the development environment?
- Modules that are too small.
  - One function modules lead to unnecessary overhead.



Don’t over-do it

- Refactoring for no reason
  - Refactor and modularize the code that is likely to change and needs to be understandable.
  - Some code almost NEVER changes. It is not worth refactoring.



Modules to Objects

Objects with high cohesion and low coupling

↑

Objects

↑

Information hiding

↑

Abstract data types

↑

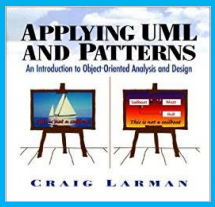
Data encapsulation

↑

Modules with high cohesion and low coupling

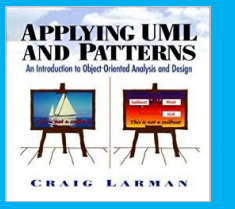
↑

Modules



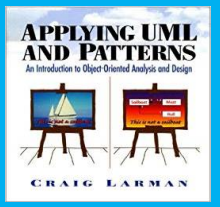
Cohesion

- Degree of interaction **within** module
- Seven levels of cohesion
  - 7. Functional | Informational (best)
  - 5. Communicational
  - 4. Procedural
  - 3. Temporal
  - 2. Logical
  - 1. Coincidental (worst)



1. Coincidental Cohesion

- Def.
  - module performs multiple, completely unrelated actions
- Example
  - module prints next line, reverses the characters of the 2nd argument, adds 7 to 3rd argument
- How could this happen?
  - hard organizational rules about module size
- Why is this bad?
  - degrades maintainability & modules are not reusable
- Easy to fix. How?
  - break into separate modules each performing one task

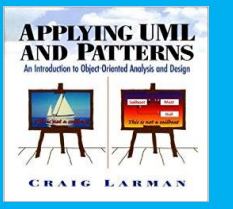


2. Logical Cohesion

- Def.
  - module performs series of related actions, one of which is selected by calling module
- Example

```
function code = 7;
new operation (op code, dummy 1, dummy 2, dummy 3);
// dummy 1, dummy 2, and dummy 3 are dummy variables,
// not used if function code is equal to 7
```

- Why is this bad?
  - interface difficult to understand
  - code for more than one action may be intertwined
  - difficult to reuse

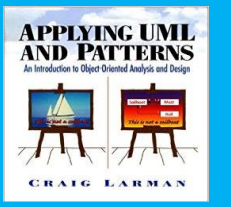


3. Temporal Cohesion

- Def.
  - module performs series of actions related in time
- Initialization example

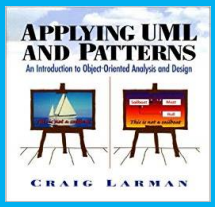
```
open old db, new db, transaction db, print db, initialize
    sales district table, read first transaction record,
    read first old db record
```

- Why is this bad?
  - actions weakly related to one another, but strongly related to actions in other modules
  - code spread out -> not maintainable or reusable
- Initialization example fix
  - define these initializers in the proper modules & then have an initialization module call each



4. Procedural Cohesion

- Def.
  - module performs series of actions related by procedure to be followed by product
- Example
  - update part number and update repair record in master db
- Why is this bad?
  - actions are still weakly related to one another
  - not reusable
- Solution
  - break up!

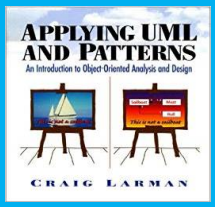


5. Communicational Cohesion

- Def.
  - module performs series of actions related by procedure to be followed by product, but in addition all the actions operate on same data
- Example 1

  `update record in db and write it to audit trail`
- Example 2

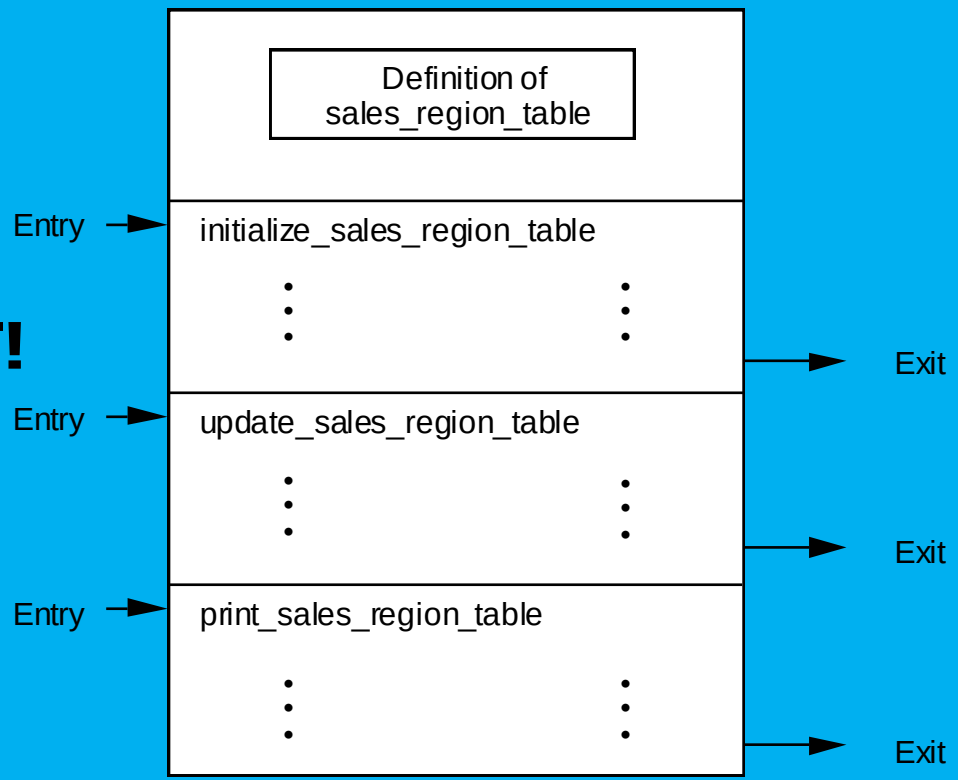
  `calculate new coordinates and send them to window`
- Why is this bad?
  - still leads to less reusability -> break it up



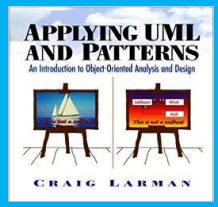
7. Informational Cohesion

- Def.
  - module performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure

This is an ADT!

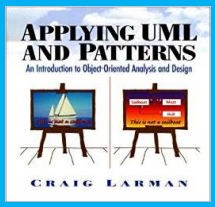


Example




```
9 references
public int getSpellDamage()
{
    return DAMAGE;
}

9 references
public float getSpellKnockBack()
{
    return KNOCK_BACK;
}

0 references
public float getSpellCoolDown()
{
    return COOL_DOWN;
}
```

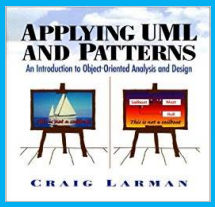
```
public class FireBall: Spells
{
    public float radius = 3f;

    0 references
    public FireBall()
    {
        speed = 20.0f;
        DAMAGE = 70;
        COOL_DOWN = 0.75f;
        KNOCK_BACK = 200.0f;

        if(checkUpgrades() != 0)
        {
            applyUpgrades();
        }
    }
```



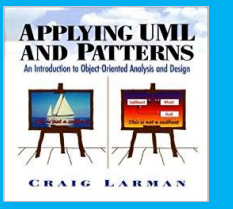
7. Functional Cohesion

- Def.
  - module performs exactly one action
- Examples
  - `get temperature of furnace`
  - `compute orbital of electron`
  - `calculate sales commission`
- Why is this good?
  - more reusable
  - corrective maintenance easier
    - fault isolation
    - reduced regression faults
  - easier to extend product

Coupling



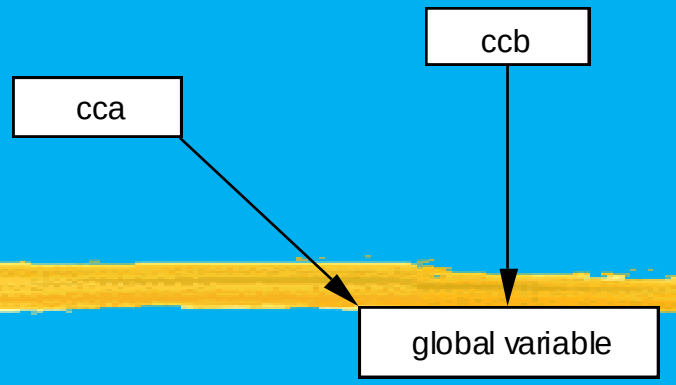
- Degree of interaction **between** two modules
- Five levels of coupling
  - 5. Data (best)
  - 4. Stamp
  - 3. Control
  - 2. Common
  - 1. Content (worst)

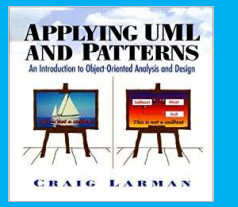


1. Content Coupling

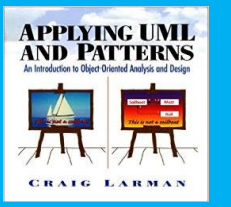
- Def.
  - one module directly references contents of the other
- Example
  - module **`a`** modifies statements of module **`b`**
  - module **`a`** refers to local data of module **`b`** in terms of some numerical displacement within **`b`**
  - module **`a`** branches into local label of module **`b`**
- Why is this bad?
  - almost any change to **`b`** requires changes to **`a`**

2. Common Coupling



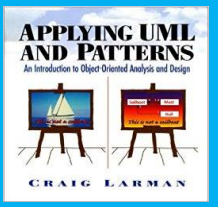


- Def.
  - two modules have write access to the same global data
- Example
  - two modules have access to same database, and can both read and write same record
- Why is this bad?
  - resulting code is unreadable
    - modules can have side-effects
    - must read entire module to understand
  - difficult to reuse
  - module exposed to more data than necessary



3. Control Coupling

- Def.
  - one module passes an element of control to the other
- Example
  - control-switch passed as an argument
- Why is this bad?
  - modules are not independent
    - module `b` must know the internal structure of module `a`
    - affects reusability

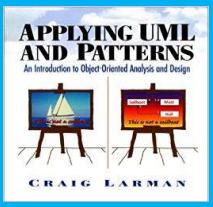


4. Stamp Coupling

- Def.
  - data structure is passed as parameter, but called module operates on only some of individual components
- Example

```
calculate withholding (employee record)
```

- Why is this bad?
  - affects understanding
    - not clear, without reading entire module, which fields of record are accessed or changed
  - unlikely to be reusable
    - other products have to use the same higher level data structures
  - passes more data than necessary
    - e.g., uncontrolled data access can lead to computer crime



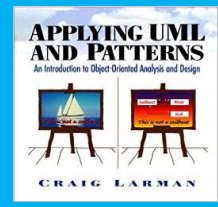
5. Data Coupling

- Def.
  - every argument is either a simple argument or a data structure in which all elements are used by the called module
- Example

```
display time of arrival (flight number)
get job with highest priority (job queue)
```

- Slippery slope between stamp & data (see queue)
- Why is this good?
  - maintenance is easier
  - good design has high cohesion & weak coupling

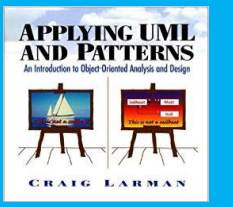
Example




```
public class Summon : MonoBehaviour
```

```
// if the square is a valid location, then attempt to place the summon, and if that succeeds, spend mana
if (Summon.attemptPlacement(summon, PlayerScript.getGridCursorPoint(), summonPosition))
{
    PlayerScript.spendMana(summon.GetComponent<Summon>().getCost());
}
```

Modules to Objects



Objects with high cohesion and low coupling

↑

Objects

↑

Information hiding

↑

Abstract data types

↑

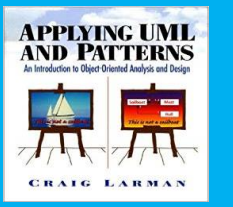
Data encapsulation

↑

Modules with high cohesion and low coupling

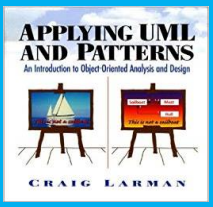
↑

Modules



Information Hiding

- Really "details hiding"
  - hiding implementation details, not information
- Example
  - design modules so that items likely to change are hidden (private)
    - future change localized
    - change cannot affect other modules
  - data abstraction
    - designer thinks at level of ADT



Not necessarily “duh”.

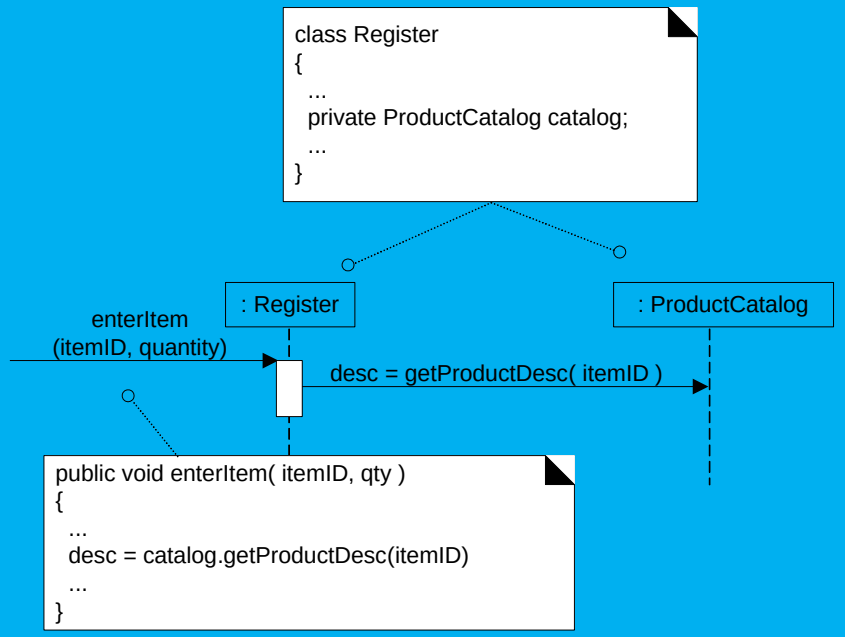
For an object to send a message to another object…

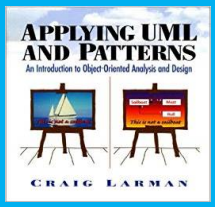
… it must have visibility to it.

More about visibility

- Definition:
  - Ability of one object to "see" or to "have reference" to another
- As messages are between objects, they can only be sent from an object that has a reference to another
- Visibility needs often emerge from interaction diagrams

Implies that instances of ProductCatalog are visible from Register.



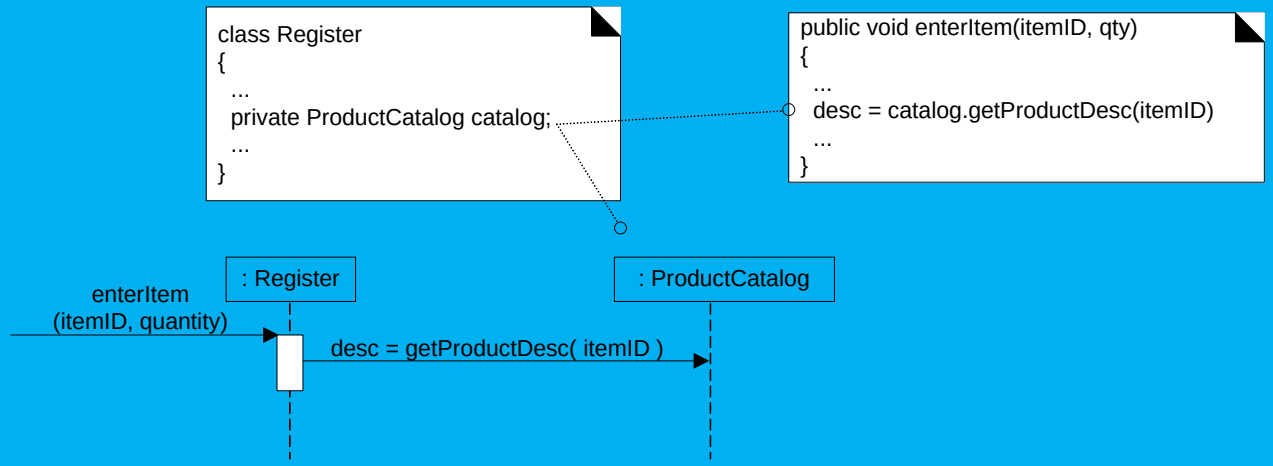


Kinds of visibility

- "Seeing" or "having reference" is usually a function of scope
  - That is, "Is object A within the scope of object B"
- Four kinds:
  - Attribute visibility
  - Parameter visibility
  - Local visibility
  - Global visibility
- So: for an object A to send a message to object B, B must be visible to A
- But what kind of visibility do we choose?

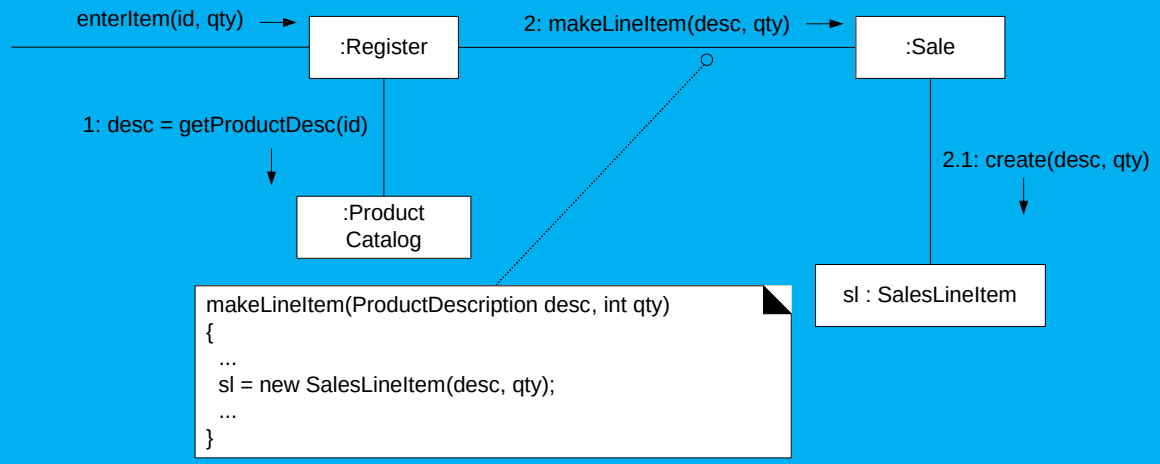
1. Attribute Visibility

- Relatively permanent form
  - That is, it exists as long as objects A and B exist
  - Declare a member variable (attribute) in the class.
- Perhaps the most common form



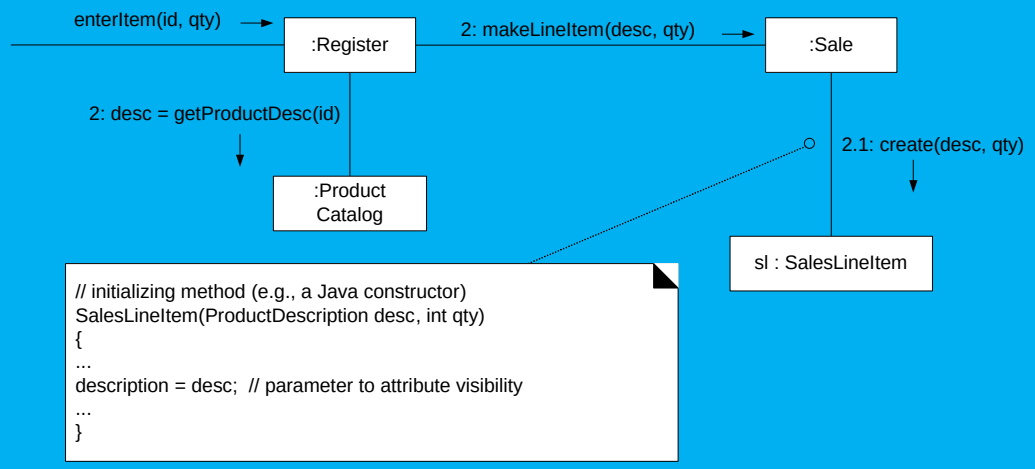
2. Parameter Visibility

- Exists when B is passed as a parameter to method of A
- Compared to Attribute visibility, this is relatively temporary
  - That is, persists only within scope of called method
- Can transform Parameter visibility to Attribute visibility



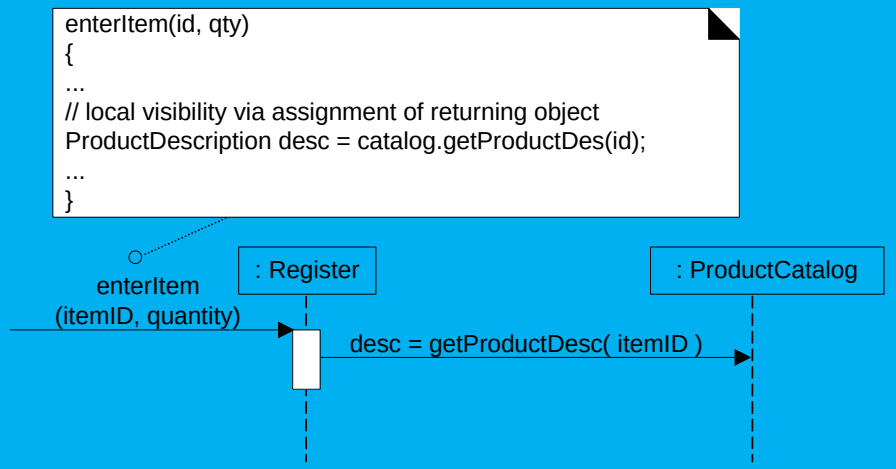
3. Local Visibility

- Exists when B is declared as a local object within a method of A
- Two ways to achieve this are by:
  - creating a new local instance and assigning it to a local variable
  - assigning the returning object from a method invocation to a local variable

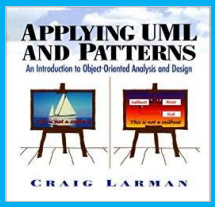


4. Global Visibility

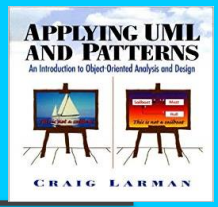
- Exists when B is global to A
- Relatively permanent visibility, but also the least common
- At least two techniques to achieved this:
  - ordinary global variables
  - Singleton pattern



Summary



- Good modules have strong cohesion and weak coupling
- Data encapsulation, information hiding, & objects ease maintenance & reuse.
- If you can't see it, you can't talk to it.

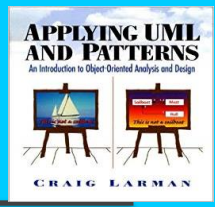


Designing Objects with Responsibilities

Using several specific design principles to guide OO design decisions.

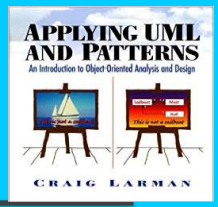
Larman: "The critical design tool for software development is a mind well educated in design principles."

Book Outline



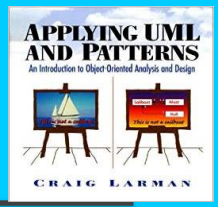
- ✧ Emphasis: There is no magic!
- ✧ The book explores use of GRASP with two case studies
- ✧ NextGen POS use cases
  - Startup
  - makeNewSale
  - enterItem
  - endSale
  - makePayment
- ✧ Monopoly
  - Startup
  - playGame

Key Point



The assignment of responsibilities and design of collaborations are very important and creative steps during design, both while diagramming and while coding.

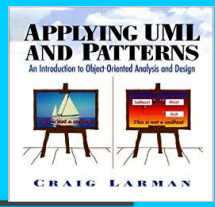
Mantra



Start assigning responsibilities by clearly stating the responsibilities.

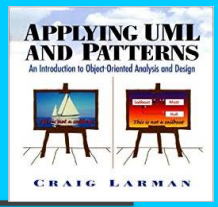
If this seems like hard work, it is because it is hard work. It relies on having enough energy, and the willingness, to be an independent thinker.

Challenge



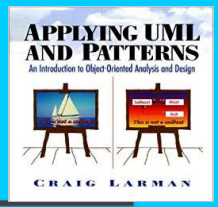
- ✧ Old-school advice on OOD
  - "After identifying your requirements and creating a domain model, then *add methods to the appropriate classes*, and *define the messages between the objects* to fulfill the requirements."
- ✧ New-school advice: there are consequences to:
  - where we place methods
  - they way objects interact in our design
  - specific principles can help us make these decisions
- ✧ Note: This is not UML! (Unified Modeling Language)
  - UML is a notation

Recall: The big picture



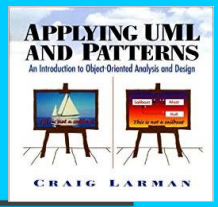
- Object design is part of the larger modeling effort
- Some inputs to modeling:
  - What's been done?
  - How do things relate?
  - How much design modeling to do, and how?
  - What is the output?
  - Domain models, SDDs (Software Design Description)
- Some outputs from modeling:
  - object design (UML diagrams – interaction, class, package)
  - UI sketches and prototypes
  - database models
  - sketches of reports and prototypes

Design of objects



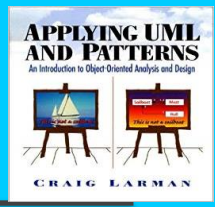
- ✧ Responsibility-driven design (RDD)
  - What are an object's *responsibilities*?
  - What are an object's *roles*?
  - What are an object's *collaborations*?
- ✧ The term is important:
  - We are initially trained to think of objects in terms of *data structures* and *algorithms* (attributes and operations)
  - RDD shifts this by treating objects as having *roles* and *responsibilities*
- ✧ Objects then become:
  - service providers
  - information holders
  - coordinators, controllers
  - interfaces, etc.

What is an object responsibility?



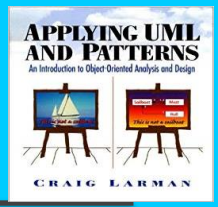
- ✧ Doing
  - doing something itself (creating an object; performing a calculation)
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- ✧ Knowing
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate
- ✧ Responsibilities vary in “granularity”
  - big (hundreds of classes); example: “provide access to relational databases”
  - small (one method); example: “create an instance of *Sale*”

What is an object collaboration?



- Assumption:
  - responsibilities are implemented by means of methods
  - the larger the granularity of responsibility, the larger the number of methods
  - this may entail an object interacting with itself or with other objects
- Therefore:
  - fulfilling responsibilities requires collaborations amongst different methods and objects

What is an object collaboration?

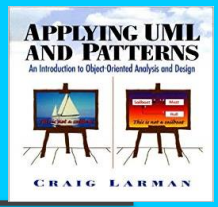


- “Low representational gap”:
  - RDD is a metaphor
  - “Objects” are modeled as similar to persons / systems with responsibilities
  - “Community of collaborating responsible objects”

"Notice that although this design class diagram is not the same as the domain model, some class names and content are similar. In this way, OO designs and languages can support a lower representational gap between the software components and our mental models of a domain. That improves comprehension."

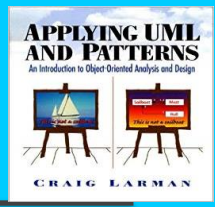
–Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Craig Larman (2004) (pg 11)

A couple of parenthetical comments



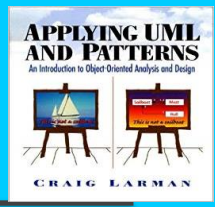
- "Responsibility" here implies "non-interference"
  - This is idealized (i.e., in the real world this is not necessarily true in boundary cases)
  - However it is suitable enough for object design
- Suitable for "programming-in-the-large"
  - As opposed to "programming-in-the-small"
  - This is a qualitative difference:
    - mass of details
    - amount of communication
    - this become overwhelming
- "Practical" as opposed to "theoretical"

GRASP



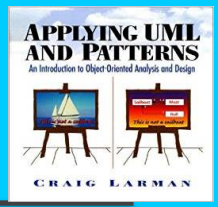
- Craig Larman's methodical approach to OO design
- GRASP stands for
  - General
  - Responsibility
  - Assignment
  - Software
  - Patterns

GRASP



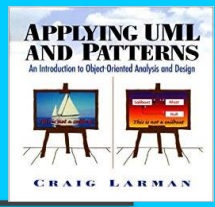
- In essence:
  - a tool to help apply responsibilities to OOD design
  - designed (and meant to be used as) methodical, rational, explainable techniques
- They are “patterns of assigning responsibilities”
  - Note: the use of “pattern” here is slightly different from some that intended by the GoF (Gang of Four) book
- Recognizes that “responsibility assignment” is something we already do:
  - UML interaction diagrams are about assigning responsibilities

A bit more about “pattern” terminology



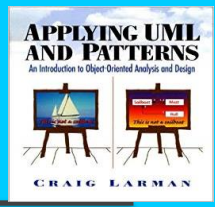
- “Patterns” as applied to architecture:
  - invented by Christopher Alexander
  - context that of a “pattern language” for designing spaces used by humans
- Classic design patterns for software
  - invented by the Gang of Four (GoF) (Gamma, Johnson, Helm, Vlissides)
- “Patterns” is now a rather “loose” term
  - Conferences exist to look at patterns
  - "Pattern Language of Programs" http://hillside.net/plop/2007/
- Larman has a special meaning for his GRASP patterns

What is a GRASP pattern?



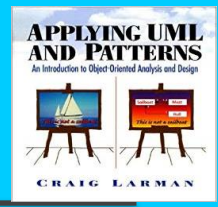
- A named and well-known problem/solution pair
- General enough to be applied to new contexts
- Specific enough to give advice on how to apply it
- Especially important for novel situations
- Also comes with a discussion of:
  - trade-offs
  - implementations
  - variations

Specifically GRASP



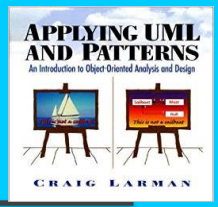
- Defines nine basic OO principles
  - basic building blocks for OO designs
- Meant to be “pragmatic”
- Meant to be a “learning aid”:
  - aids naming the group of ideas
  - aids in presenting insight of ideas
  - aids in remembering basic, classic design ideas
- Also intended for combination with:
  - a development process (UP)
  - an organizational metaphor (RDD)
  - a visual modeling notation (UML)

The Nine GRASP Patterns



1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

Guideline

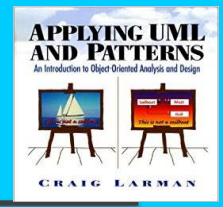


When coding, program at least some Start Up initialization first.

But during OO design modelling, consider the Start Up initialization design last, after you have discovered what really needs to be created and initialized.

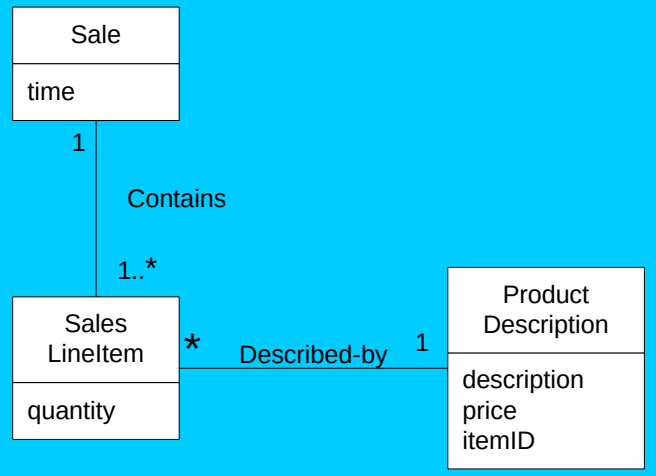
Then design the initialization to support the needs of other use case realizations.

1. GRASP Creator



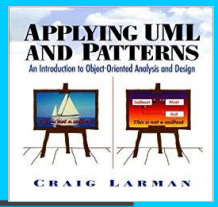
Problem: Who should be responsible for creating a new instance of some class?

Example: Who is responsible for creating "SalesLineItem"?



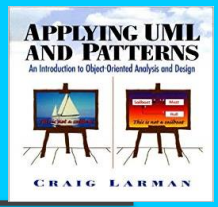
- Object creation is one of the most common OO activities
- "Creator" principle is meant to help us achieve:
  - low coupling
  - increased clarity
  - increased encapsulation
  - increased reusability

Jargon reminder:



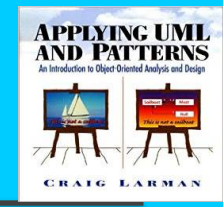
- **coupling:** the degree to which each program module relies on other modules
- **cohesion:** a measure of how well the operations in a module work together to provide a specific piece of functionality
- **encapsulation:** synonym for “information hiding”
- **reusability:** likelihood a segment of structured code can be used again to add new functionality with slight or no modification

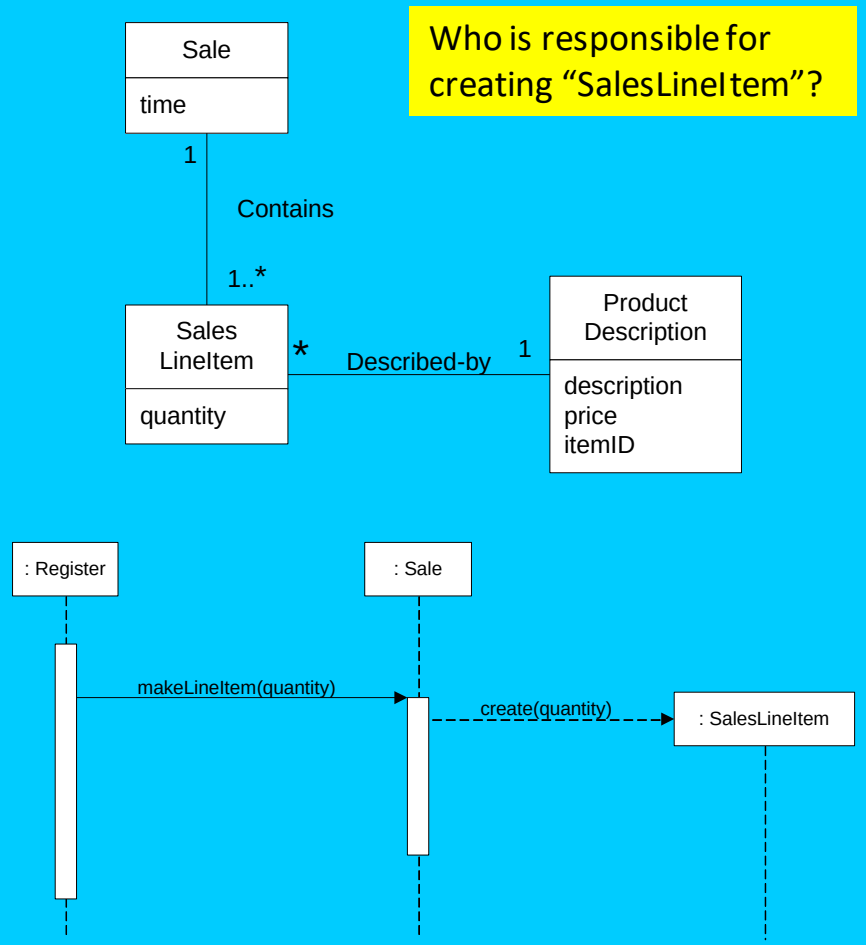
1. GRASP Creator



- Problem statement:
  - Who should be responsible for creating a new instance of some class?
- Solution:
  - Assign class B the responsibility to create an instance of class A if one of these is true (& the more the better)…
    i. B "contains" or compositely aggregates A.
    ii. B records A
    iii. B closely uses A.
    iv. B has initializing data for A (i.e., B is an *Expert* with respect to creating A).

1. GRASP Creator



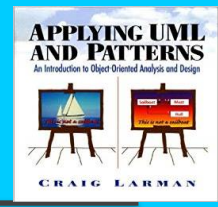


- Sale will contain many SalesLineItem objects
- Creator GRASP pattern suggests Sale is one object that could fulfill this responsibility.
- Consequences:

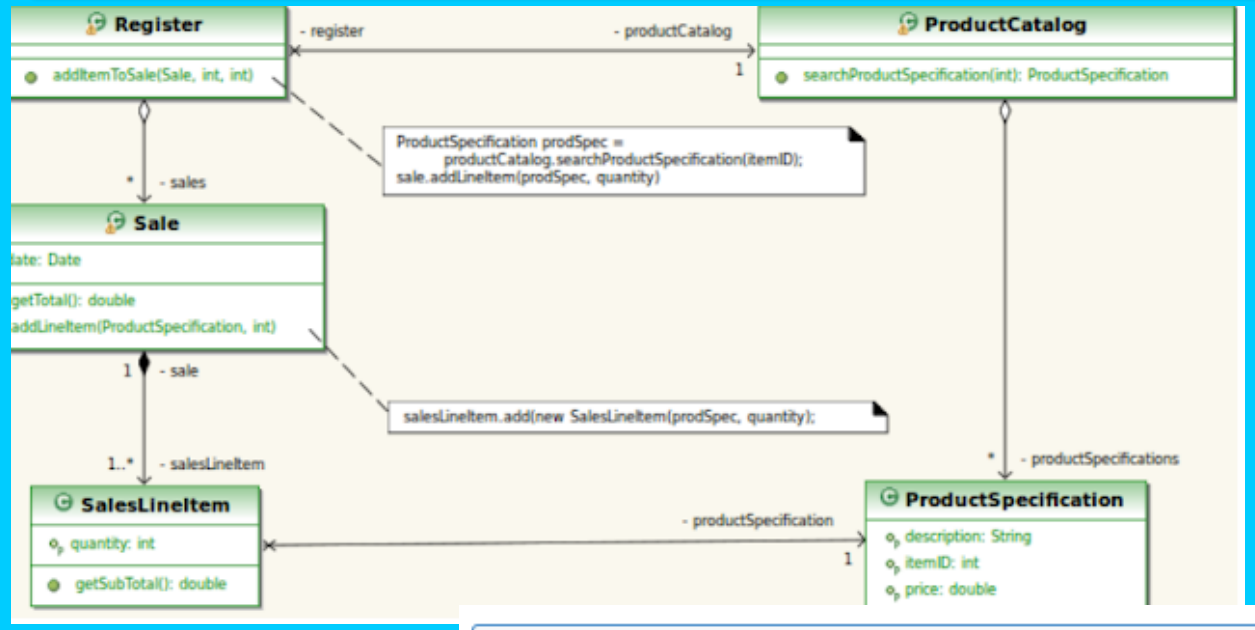
  makeLineItem becomes a method in Sale.

  we capture this decision in our UML design-model diagrams

https://www.sourcecodeexamples.net/2018/06/creator-grasp-pattern.html

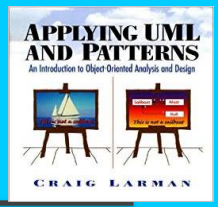


Another Look



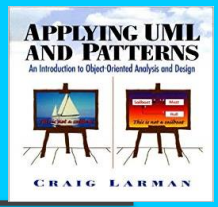

```
class Sale {
    List<SalesLineItem> salesLineItem
    = new ArrayList<SalesLineItem>();
    //...
    public void addLineItem(ProductSpecification prodSpec,int quantity) {
        salesLineItem.add(new SalesLineItem(prodSpec, quantity);
 return salesLineItem;
    }
}
```

GRASP patterns give you more…



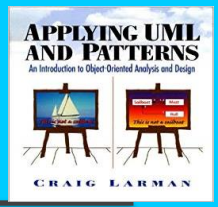
- Each of these patterns comes with:
  - a “discussion” section
  - a “contraindication” section
  - a “benefits” section
  - a “related patterns or principles” section
- Important!
  - these patterns provide advice on assignment of responsibilities
  - patterns may conflict in their advice
  - therefor we must exercise some judgment in applying these patterns
  - the extra sections help us with this
- This is not mechanical work!

1. GRASP Creator



- Discussion:
  - intent is to support low coupling (i.e., creator is found that already needs to be connected to created object)
  - initializing data is sometimes an indicator of a good Creator
    - some object constructors have complex signatures
    - which objects have the information needed to supply parameter values for such constructors?
- Contraindications
  - creation can be complex (recycled instances, creating an instance from a family of similar classes)
  - may wish to use Factory Method or Abstract Factory instead in these cases

1. GRASP Creator



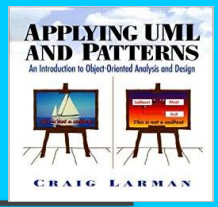
✧ Benefits:

- low coupling has already been mentioned
- why is this good?

✧ Related patterns or principles:

- Low Coupling
- Factory Method and Abstract Factory
- Whole-Part pattern: defines aggregate objects that support encapsulation of components

Guideline

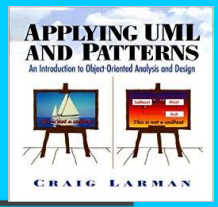


Do the initialization design last.

Choose as an initial domain object a class at or near the root of the containment or aggregation hierarchy of domain objects.

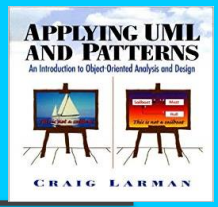
This might be a facade controller or some other object considered to maintain all or most other objects.

2. GRASP Information Expert



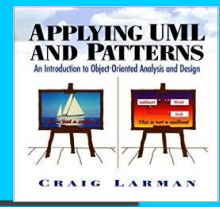
- Problem statement: What is a general principle of assigning responsibilities to objects?
  - Recall: design model may end up very large (hundreds of classes and methods)
  - If assignment of responsibilities is well chosen, result is system more easily understood than one with poorly chosen responsibility assignment.
- Solution:
  - Assign a responsibility to the information expert
  - This expert is the class that has the information needed to fulfill the responsibility

Key idea:

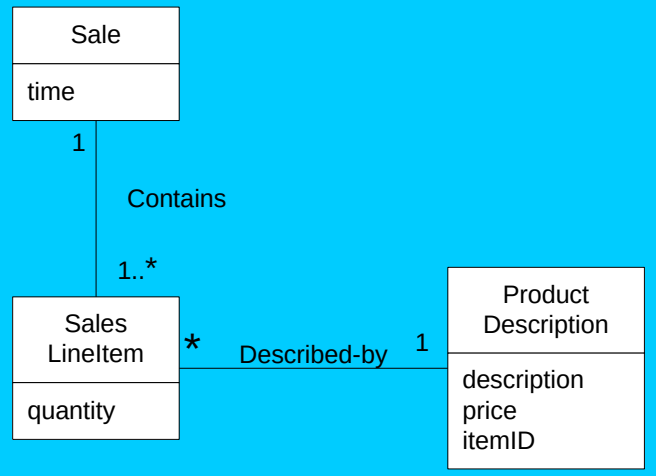


("Clear Statement" Rule): Start assigning responsibilities by clearly stating the responsibility.

2. GRASP Information Expert

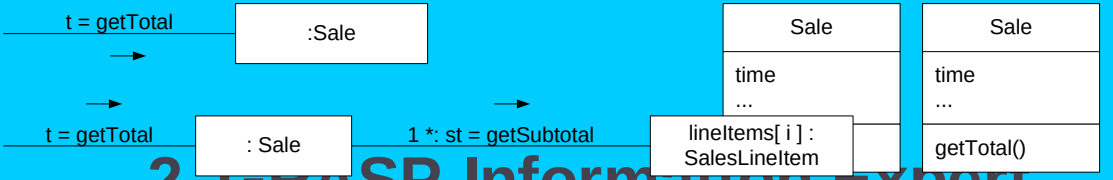


If there are relevant classes in the design model, then look there first for classes…

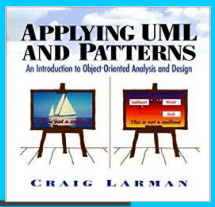


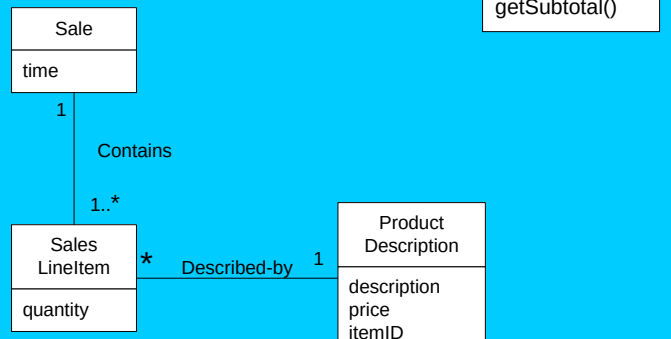
… otherwise look into the Domain Model in order to use or expand its representations to suggest the creation of appropriate classes

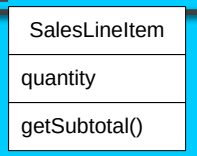
- NextGEN POS will end up with lots of functionality
  - One small part of this is a class that knows the grand total of a sale
- "Clear Statement" rule could transform this into:
  - "Who should be responsible for knowing the grand total of a sale?"
- Which class of objects has the information needed to determine the total?



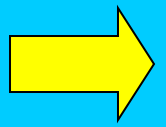
2. GRASP Information Expert



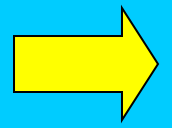




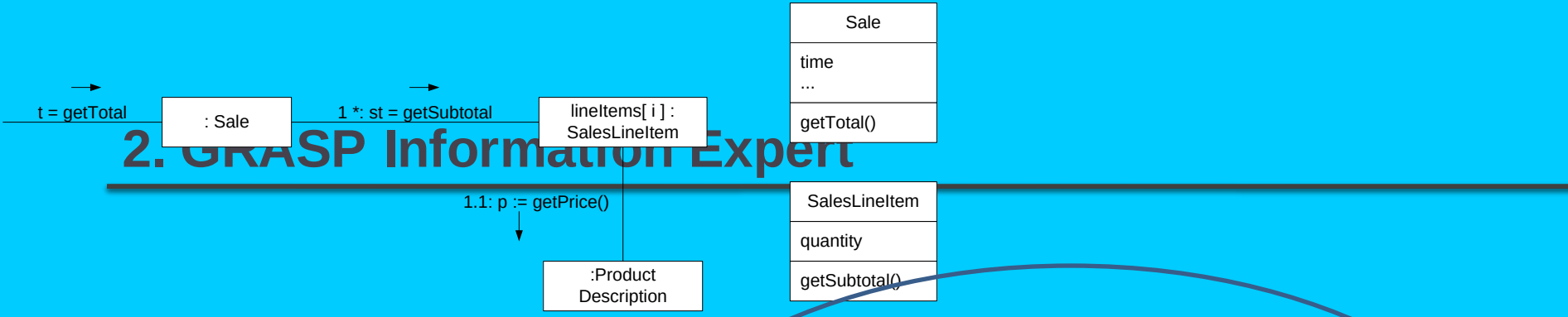
new method

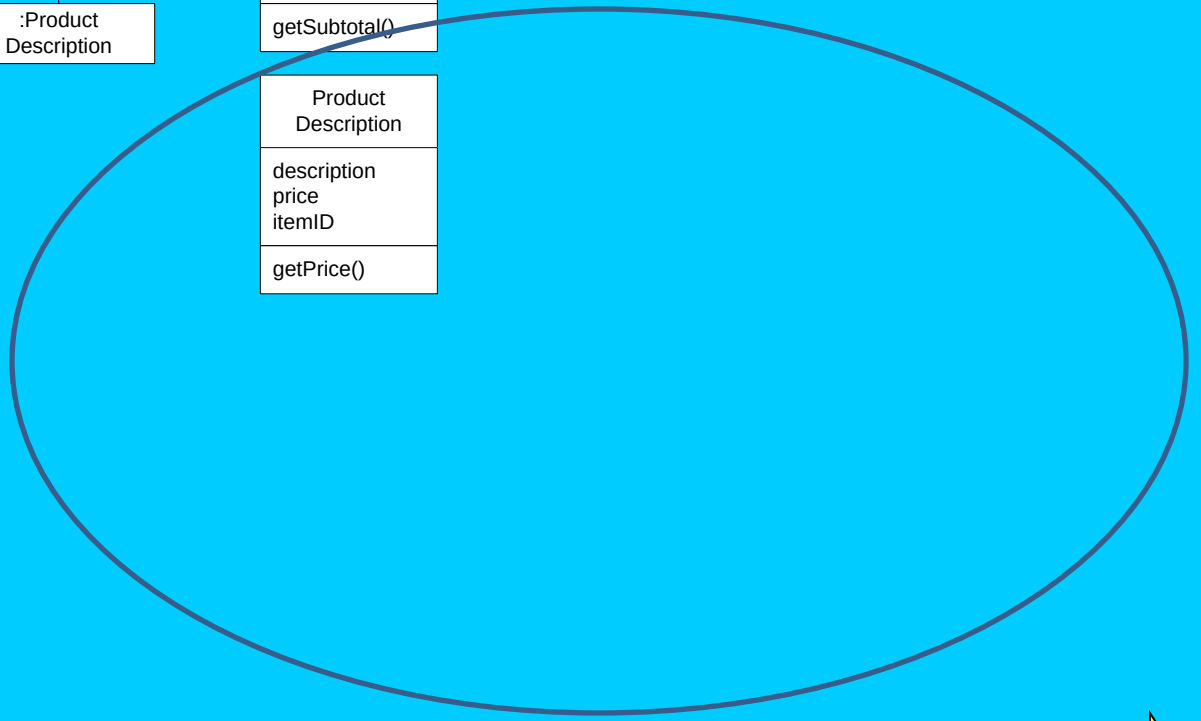


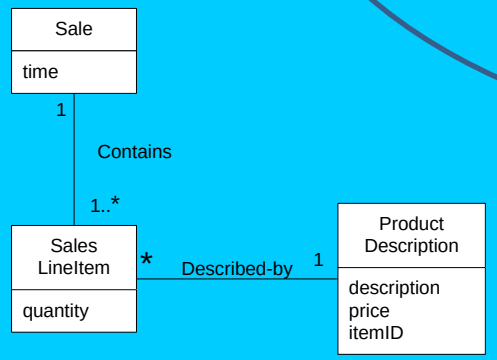
new method



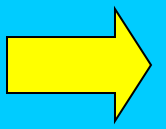
2. GRASP Information Expert

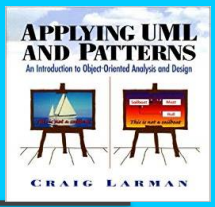




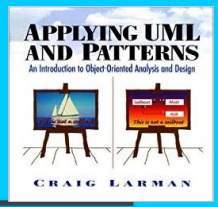


new method



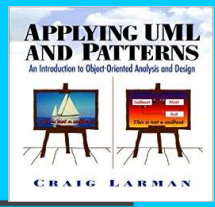


Comment



- Notice how the initial assignment of responsibilities drove the design:
  - Result is the identification of several new methods
  - Interaction diagram helped us capture the result
  - Class model entities are also modified
- Also notice the degree of delegation
  - “getSubtotal” (vs. “getPrice” and “getQuantity” to supply a multiplication operation)

3. GRASP Low Coupling



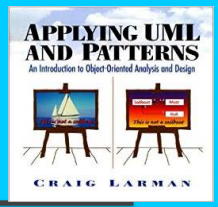
✧ Problem:

- How can our design support
  - low dependency?
  - low change impact?
  - increased reuse?

✧ Solution:

- **Assign a responsibility so that coupling remains low**.
- Use this principle to evaluate alternative assignment of responsibilities.

Key idea:



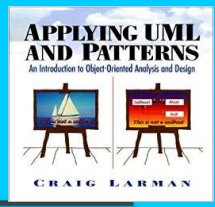
Coupling affects global understanding, but decisions about placing responsibility must be made more locally, usually by choosing from amongst alternatives.

: Register

: Payment

:Sale

Example

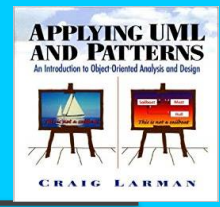


- NextGen POS: dealing with Payments
- "What object should be responsible for associating Payments with Sales?"
- Three objects appear to be involved

  Real-world domain: Register records a Payment (Creator?)

  Also in real-world: A Register also records Sales.

Example

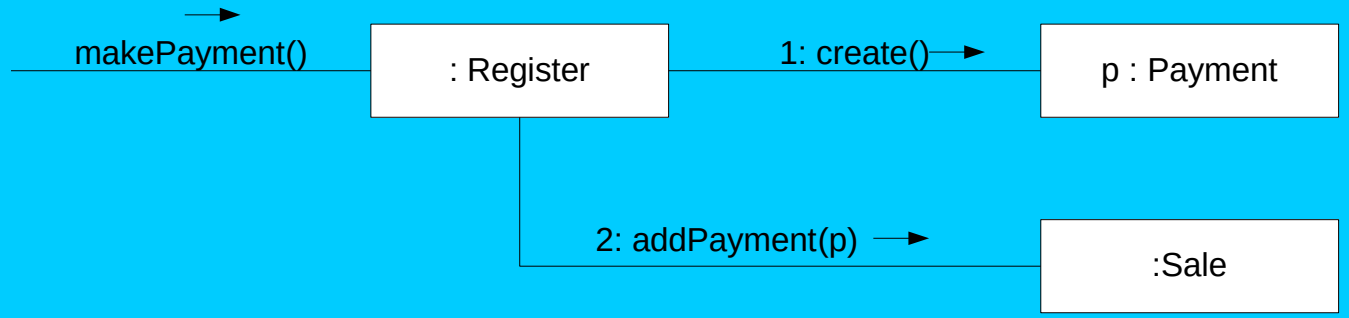


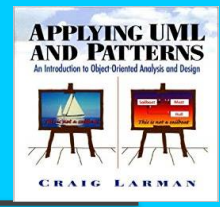
- Suppose we apply our two previous GRASP patterns like so:

  (Creator) Suggests Register as a candidate for creating Payments.

  (Information Expert) Register knows about Payments, and therefore associates these with Sales

- Resulting interaction diagram appears below

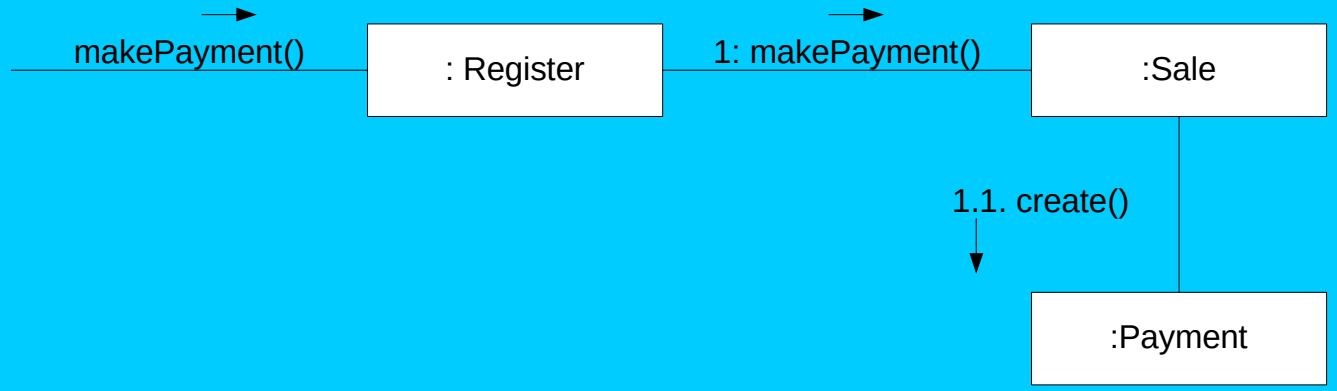




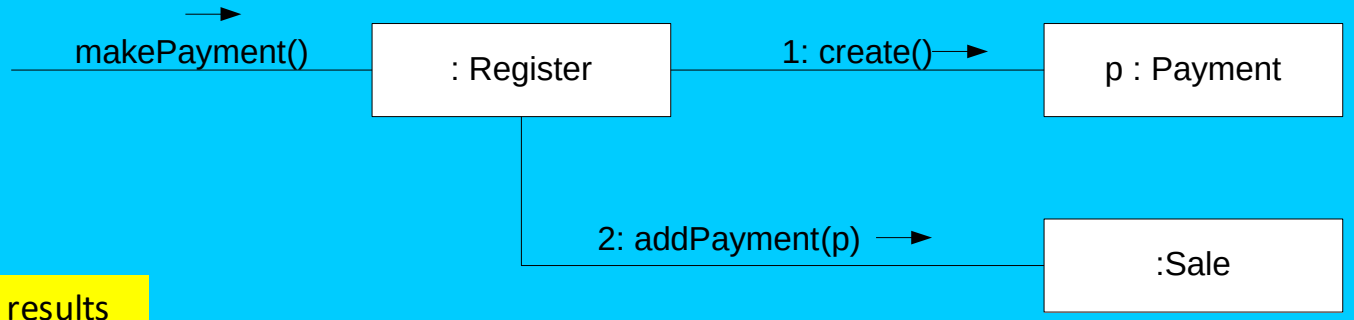
Example

- Recall: Our GRASP pattern is to be applied to alternatives.
- Another approach:

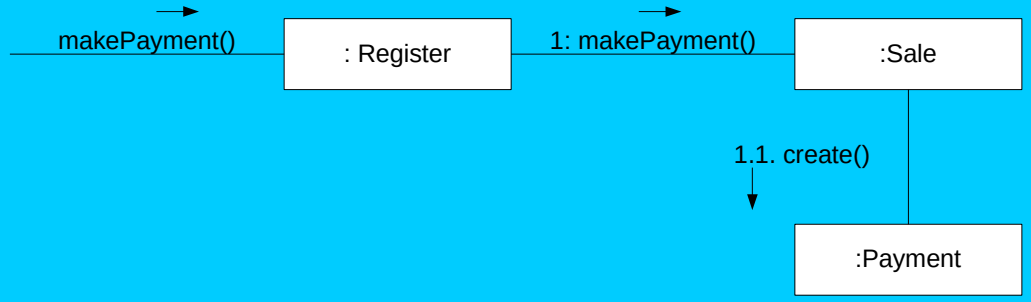
  (Creator) Suggests Sales creates Payments



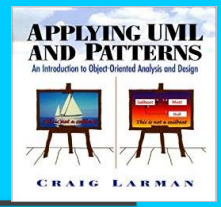
Evaluating Alternatives



Which design choice results in lower coupling?

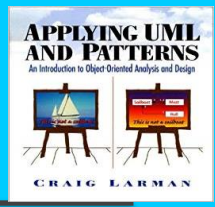


Kinds of Coupling



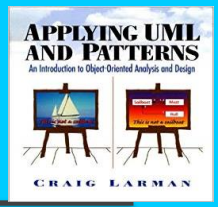
- Often applied to “types” or “classes”
  - Class X has an attribute referring to Class Y instance
  - Class X objects call on the services of a Class Y object
  - Class X has methods referencing instances of Class Y (i.e., parameters, local variables of Class Y)
  - Class X is a direct or indirect subclass of Class Y
  - Y is an interface, and Class X implements it
- Note:
  - All of these are needed at some point
  - We are not trying to eliminate coupling, but rather make careful choices

Important contra-indicator



- There is a least one set of objects and classes to which we must allow our code to be highly-coupled
  - Java libraries (i.e., javax.swing, java.util) etc.
  - C libraries
  - .NET libraries
- Key features permitting this:
  - these are stable
  - they are widespread

4. GRASP Controller



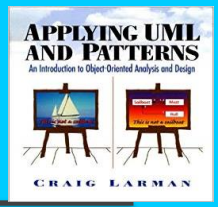
✧ Problem:

- What first object beyond the UI layer receives and coordinates (“controls”) a system operation?

✧ Solution:

- Assign responsibility to a class based on one of the following:
  - class is “root object” for overall system (or major subsystem)
  - a new class based on use case name

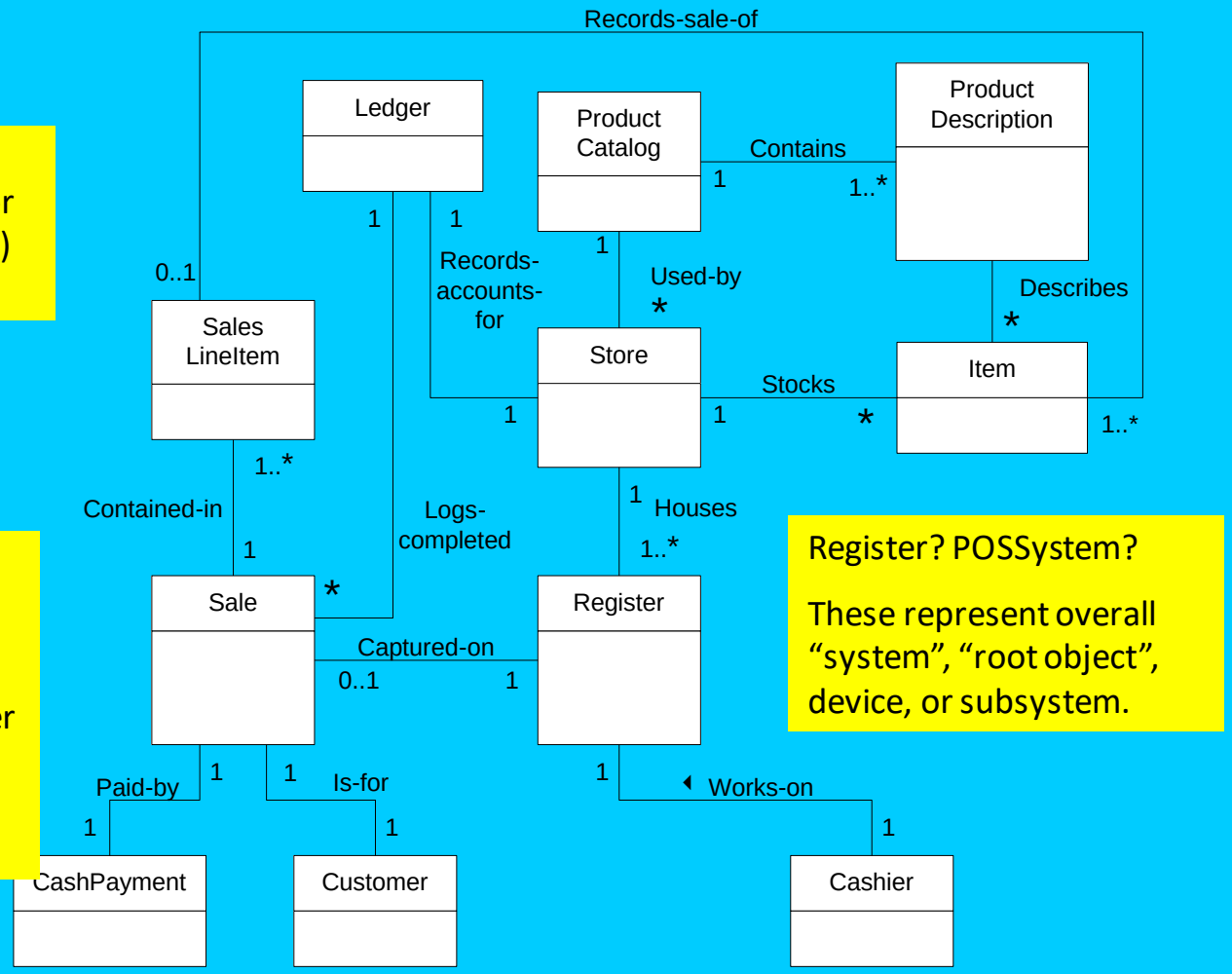
Context



- This usually is applied to the actions recorded in our SSDs (system sequence diagrams)
  - system events generated by some GUI
  - we do not write to the GUI just yet, but assume some UI layer
- Therefore we do not have “window”, “view” or “document” classes
  - Be very careful here as we are so tied to GUIs when visualizing applications
  - GUI is meant to “delegate” tasks to system

Example (NextGen POS)

Which class of object should be responsible for receiving the enterItem() system event message?



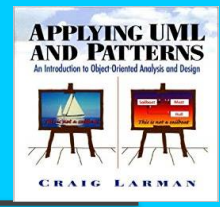
ProcessSaleHandler?
ProcessSaleSession?
(these would be new)

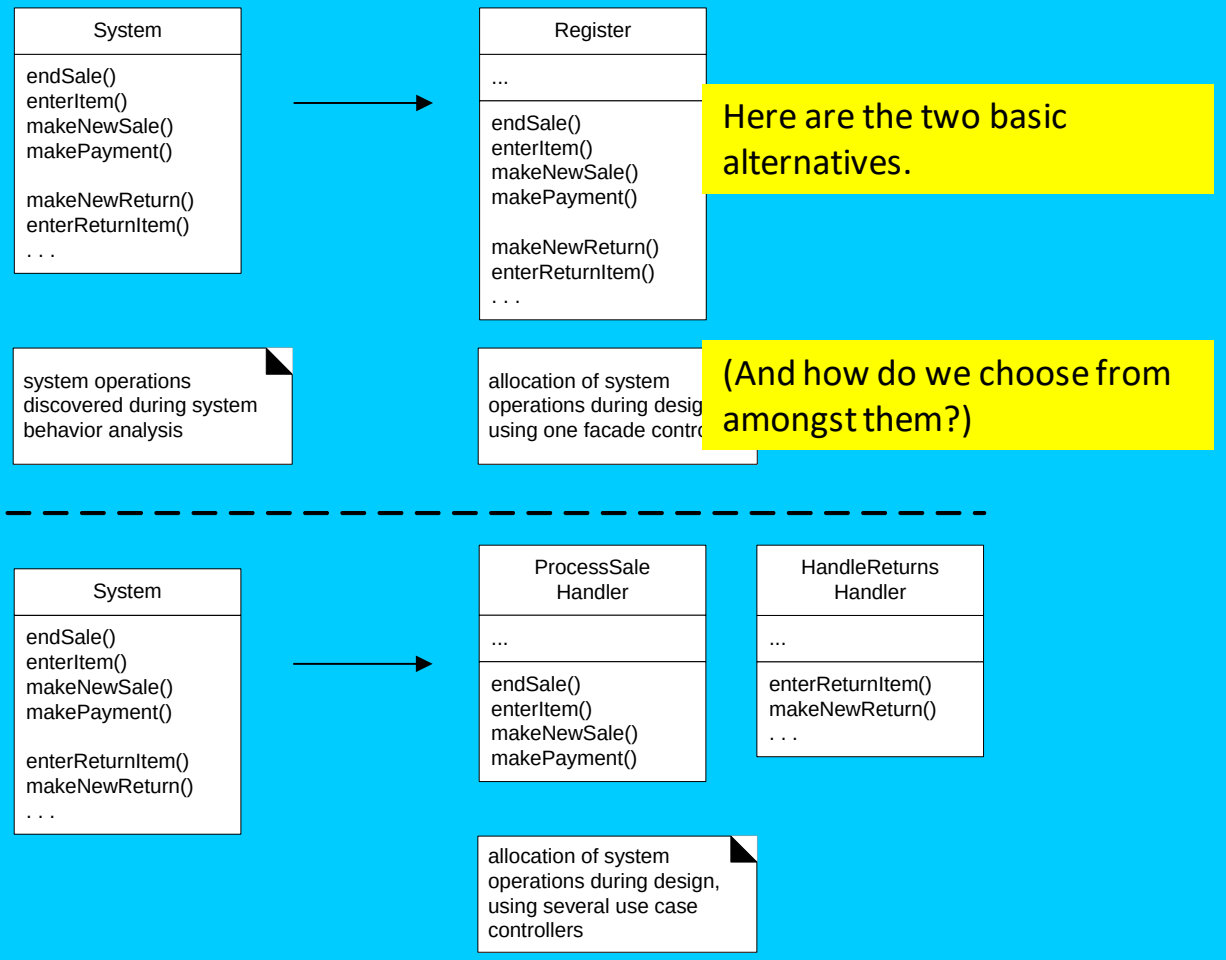
These represent a receiver or handler of all system events of a use case scenario.

Register? POSSystem?

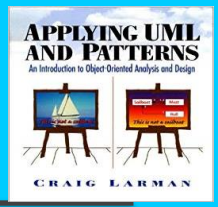
These represent overall "system", "root object", device, or subsystem.

Example NextGen POS



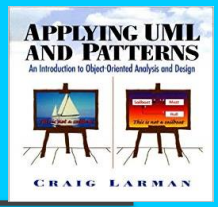


Caveats



- A balance must be striken between having too many controllers and too few
  - More common problem: too few (i.e., “bloated controllers”)
- Example:
  - single controller classes for all events (façade)
  - controller performs many of the events itself rather than delegate work
  - controller maintains a lot of state information about the system (e.g., variables and attributes)

5. GRASP High Cohesion



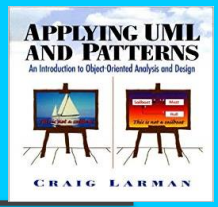
✧ Problem:

- How can we keep objects in our design
  - focused?
  - understandable?
  - manageable?

✧ Solution:

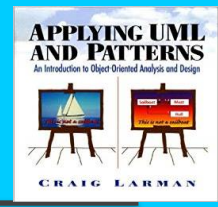
- Assign a responsibility so that cohesion remains high.
- As with low coupling, use this evaluate alternatives to placing responsibilities in objects

Signs of Low Cohesion

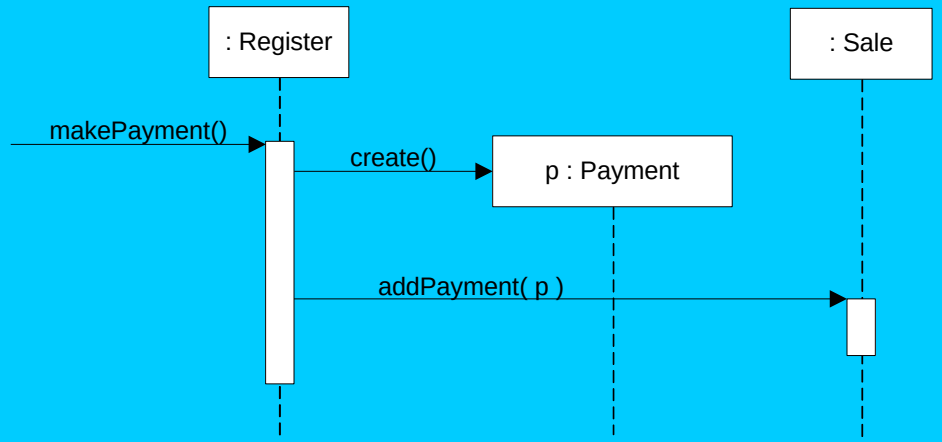


- ✧ Classes that do too many **unrelated** things or which do too much work
- ✧ Such classes are:
  - hard to comprehend
  - hard to reuse
  - hard to maintain
  - delicate in that any change elsewhere in the system requires many changes in this class
- ✧ In general:
  - the greater the granularity of classes, the lower the cohesion
  - but we do not want to go to the other extreme (i.e., lots and lots of classes, each doing one trivial thing)

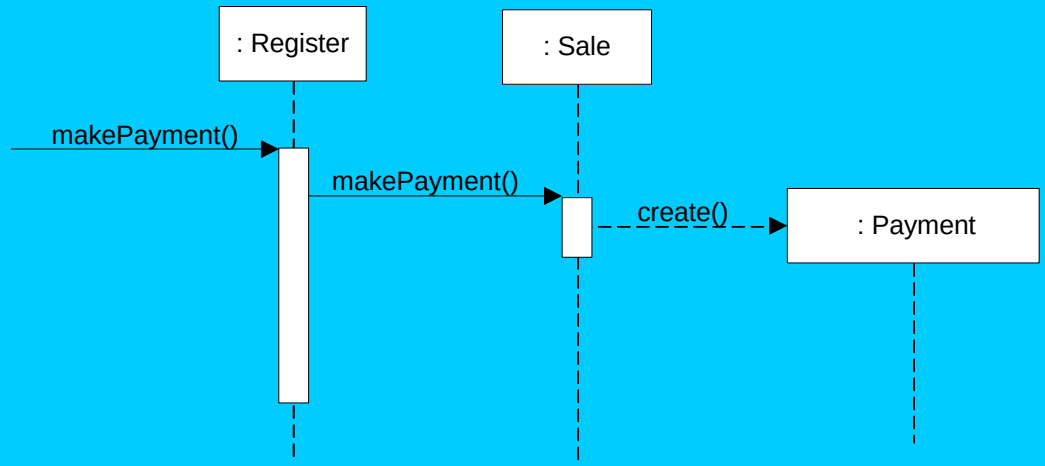
Compare and contrast



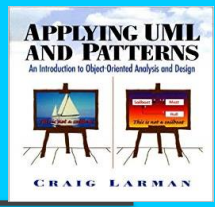
(Yet again) What class should be responsible for creating and class instance and associating it with a Sale?



We have two options. Which shows both high cohesion and low coupling?

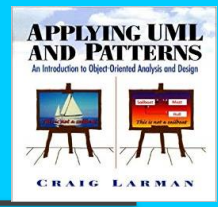


Contra-indicators



- ✧ Simple people-management issues
  - Large class maintained by a single programmer
- ✧ Performance
  - Distributed systems
  - Any place where the overhead of invoking an operation is a significant proportion of performing the operation

Guideline

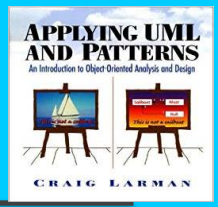


When there are alternative design choices, taking a closer look at the cohesion and coupling implications of alternatives.

You can also try to think ahead to future evolution pressures on the alternatives.

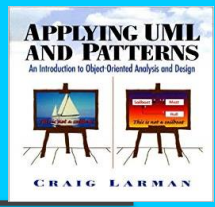
Choose an alternative with good cohesion, coupling, and stability in the presence of likely future changes.

Summary



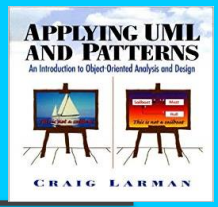
- ✧ High Cohesion and Low Coupling are the underlying theme of all patterns.
  - The Creator pattern supports low coupling by choosing a creator that already needs to be connected to created object.
  - Information Expert assigns responsibilities to the objects that have the information needed to complete the task.
    - Cohesion is maintained by delegating.
  - Sometimes patterns will give conflicting answers.
    - This is not an exact science, but be able to justify your choices.

Summary



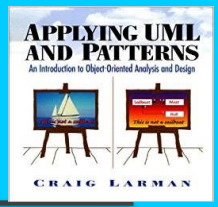
- ✧ No coupling is also undesirable
  - A central metaphor of object technology is a system of connected objects that communicate via messages.
  - Low Coupling is taken to excess leads to a few incohesive, bloated, and complex active objects that do all the work, with many very passive zero-coupled objects that act as simple data repositories.

Summary

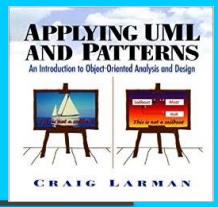


- ✧ Libraries are Highly Coupled. This is permitted because:
  - these are stable
  - they are widespread/proven
- ✧ A subclass is by definition Highly coupled to its superclass.
  - Have a good reason for sub classing between modules/team members.
  - If you have a superclass for another team member, ensure that it takes on the properties of a library.
    - Well documented
    - Infrequent changes
    - You will be held to a higher standard!

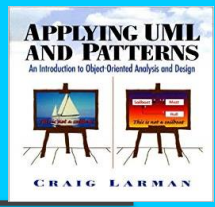
Summary



- ✧ Have a single controller that accepts messages (or brake it into several based on message type if it becomes too bloated.)
- ✧ Your GUI is NOT your controller.
  - Do not tie your business logic with your user interface because this limits future changes.



Coding Standards, Developer Manual, and Using the Gantt

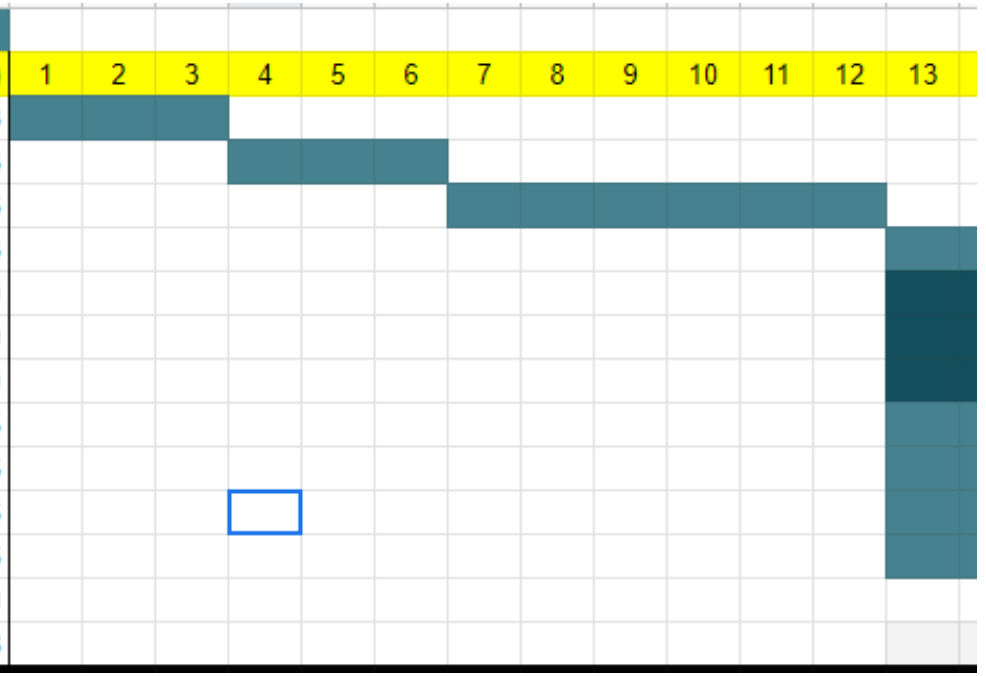

APPLYING UML
AND PATTERNS
An Introduction to Object-Oriented Analysis and Design
CRAIG LARMAN

The Roast

Roast of Bloonagins

Where are software patterns on your Gantt?

David		Unfinished	Finished
Enemy		Predicted Time (hrs)	Actual Time (hrs)
	Enemy Design	3	3
	Enemy SuperClass/Methods	3	3
	GoblinGrunt Subclass/Methods	5	6
	GoblinWarrior Subclass/Methods	5	3
	GoblinAssassin Subclass/Methods	5	0
	GoblinArcher Subclass/Methods	8	0
	GoblinGiant Subclass/Methods	5	0
	Enemy Targeting	10	5
	Enemy Interaction	10	6
	Enemy Movement/Pathfinding	8	6
	Enemy Boundary/Stress Tests	5	6
	Game Demo Mode	15	0
	Total	82	38



Roast of Bloonagins

Why are these individuals' hours low?

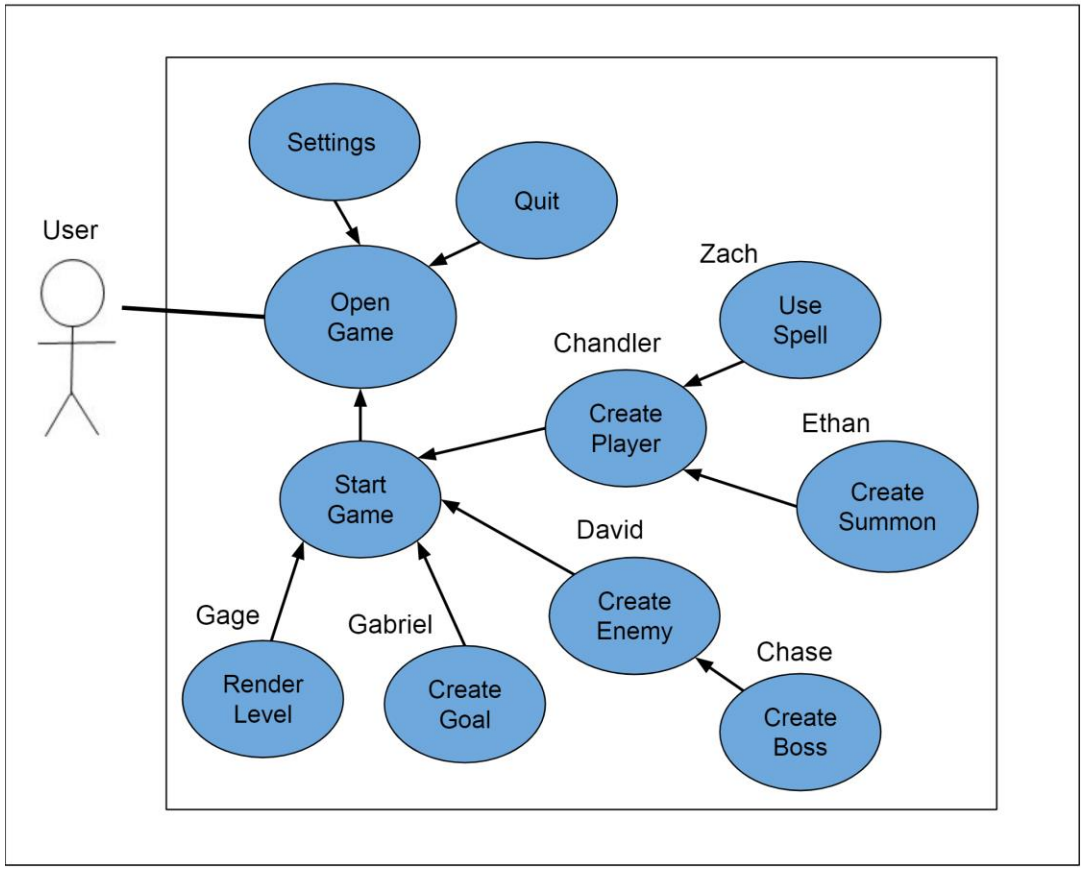
Gabriel	Key:	Unfinished	Finished
Menu/UI/HUD Integration		**Predicted Time (hrs)**	**Actual Time (hrs)**
	Menu Creation	3	4
	UI Creation	3	5
	Goal Manager Script Coding	4	
	Goal Superclass	5	
	Goal Subclass	6	
	Testing Balancing	7	
	HUD Elements	6	
	Integration Checks	5	
	Total	39	9

Gage	Key:	Unfinished	Finished
Tile Mapping/Levels		**Predicted Time (hrs)**	**Actual Time (hrs)**
	Develop rough draft of levels	8	4
	Level 1 map design/implementation	3	7
	Implemented testing to level 1	2	5
	Level 2 map design/implementation	5	
	Level 3 map design/implementation	5	
	Level 4 map design/implementation	5	
	Level 5 map design/implementation	5	
	Level 6 map design/implementation	5	
	Level 7 map design/implementation	5	
	Level 8 map design/implementation	5	
	Programming Level Manager	13	4
	Programming Wave Spawner	10	3
	Mobile compatibility	3	
	VR compatibility	2	
	Total	76	23

Chandler	Key:	Unfinished	Finished
Player Controls		**Predicted Time (hrs)**	**Actual Time (hrs)**
	Create Player Object	2	0.2
	Make Player Take Damage on Hit	4	2
	Make Player Lose Mana When Summoning	2	1
	Create Movement Controls	4	4
	Make Dash Button	2	1
	Make Build/Cast Mode Switching Controls	2	2
	Make Summon/Spell Selection Controls	4	2
	Make Casting / Summoning Controls	2	1
	Make Pause Button	1	0.1
	Make Summon Targeting Controls	1	0.1 so far
	Make Destroy Summon Button	1	0.2
	Make Ready Up Button	1	0.1 so far
	Make Wave Info Button	1	0.1
	Make Camera Follow Player	1	0.1
	Make Camera Shift with Cursor Movement	5	4
	Implement Pets?	16	
	Implement Potions?	12	
	Total	61	17.7

Roast of Bloonagins

- One teammate owns the superclass of another's subclass
  - Is this level of coupling going to be a problem?
  - How are you going to manage it?



Roast of ScoSoft

Where is Kadence's pattern information?

category	austin (h)	hayden (h)	james (h)	kadence (h)	rodney (h)	zach (h)	total (h)
training	1.0		1.0			5.0	**7.0**
software specialists presentation prep	6.0	15.0	8.0			25.0	**54.0**
inividual team lead presentation prep & deliverable	2.0	5.0	12.0			22.0	**41.0**
oral exam prep							**0.0**
post mortum presentation prep							**0.0**
choosing software patterns	0.5	1.0	1.0		1.0	0.1	**3.6**
total (h)	**9.5**	**21.0**	**22.0**	**0.0**	**1.0**	**52.1**	**102.0**

Roast of ScoSoft

- Why are these individuals' hours so low?

	name	role	predicted (h)	actual (h)	delta (h)	progress
3	austin	project manager	39.0	12.0	8.0	51%
4						
5	task	description	prereq	predicted (h)	actual (h)	status
6	1	launch menu graphic design		1.0	4.0	complete
7	2	pause menu graphic design	1	1.0		planned
8	3	settings menu graphic design	1	2.0		active
9	4	launch menu programming	1,3	3.0	2.0	complete
10	5	pause menu programming	1,2	3.0		planned
11	6	settings menu programming	1,4	5.0		planned
12	7	ainigma puzzle graphic design		4.0	1.0	active
13	8	odyssey puzzle graphic design		2.0	1.0	active
14	9	ainigma puzzle programming	7	4.0		planned
15	10	odyssey puzzle programming (dynamic binding)	8	7.0	2.0	active
16	11	testing	4,5,6,7,8,9,10	3.0	2.0	active
17	12	documentation	11	4.0		planned

name	role	predicted (h)	actual (h)	delta (h)	progress
kadence	version control manager	50.0	6.0	1.0	14%

task	description	prereq	predicted (h)	actual (h)	status
1	Requirements Definition	-	2.0	2.0	complete
2	Obstacle Design	1	5.0	4.0	complete
3	Obstacle 1 Graphic Design	2	7.0		active
4	Obstacle 1 Programming	3	7.0		planned
5	Obstacle 2 Graphic Design	2	5.0		planned
6	Obstacle 2 Programming	5	5.0		planned
7	Obstacle 3 Graphic Design	2	4.0		planned
8	Obstacle 3 Programming	7	4.0		planned
9	Testing	4,6,8	5.0		planned
10	Documentation	9	5.0		planned
11	Installation	9	1.0		planned

name	role	predicted (h)	actual (h)	delta (h)	progress
hayden	QA manager	85.0	17.0	28.0	53%

task		prereq	predicted (h)	actual (h)	status
1	Create flashlight class		20.0	10.0	complete
2	Find 3D assets		5.0	0.5	active
3	Integrate Flashlight with Player	1	5.0		planned
4	Create Enemy Superclass	1	5.0	5.0	complete
5	Create Light Enemy Subclass	4	25.0		planned
6	Create Heavy Enemy Subclass	5	5.0	0.5	active
7	Integrate Enemy Spawning with L	4,5,6	5.0		planned
8	Integrate Enemy Classes with We	4,5,6	5.0		planned
9	Testing		10.0	1.0	active

- James' powerups are only created when Zach calls them.

```
/*
* Scoin powerup
* Contains payload and overrides.
*
* Member variables: none
*/
public class Scoin : PowerUp
{
```

```
39    */
      10 references
40    public class Room : MonoBehaviour {
          0 references | 7 references | 6 references | 6 references
41        public GameObject m_enemy, m_pickup, m_wall, m_wallDoor;
          4 references
42        public GameObject[] m_wallList = new GameObject[4];
43
```

Roast of the Middle Men

Where are patterns and dynamic binding on Gantt?

Overhead

	Holly	Lily	Heath	Donald	Kaleb	Seth
Training	1	1	1	1	1	1
SA Preesentation Prep	4	3	4	4	4	4
Software Specialist Presentation Prep	12	10				
Team Lead Presentation Prep	3	7				
Oral Exam Prep						
Post Mortum Presentation Prep						
Total	20	21	5	5	5	5

Gantt Example

Holly	
Requirements Collection	2
Repo + Git Setup	3
Installation	1
Tutorials	3
Main Menu Design	1
Pause Menu Design	
Inventory Design	
Design Instructions & Settings Scree	3
Link Menus	1
Documentation	
Testing	11
Adding Patterns and Other Require	
totals	25

Roast of the Middle Men

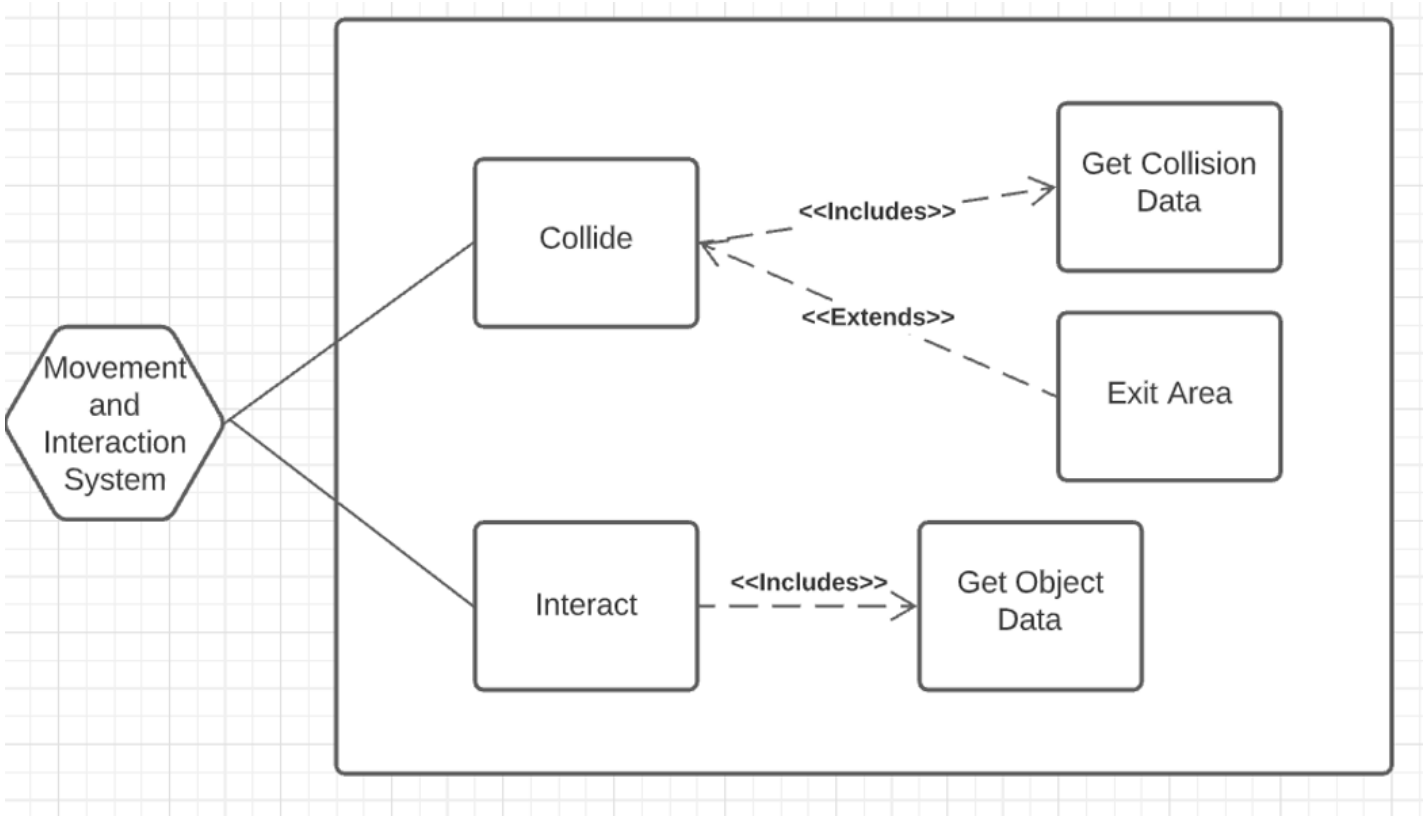
Why are these individuals' hours low?

Seth	
Overworld Player Movement	4
Player controls (Overworld)	1
Player/World Collisions	
Player/Object interaction	
Player/NPC interaction	
Dr BC Mode Implementation	
Testing	5
totals	10

Kaleb	
Player Controls	4
Organizational Troubles	3
Player Moves	1
Interface	
NPC AI	
NPC Moves	
Transitions	
Move Animations	
Hit/Block Interactions	
Dr-BC Mode	
Special Moves and Meter	
Autoplay	
Testing	4
totals	12

Roast of the Middle Men

Is there too much coupling between Seth and Donald's systems?



Roast of Team Cafés

Where are patterns/dynamic binding not in your Gantt?

Riley		
Create Battle Management System	9	8
Implement player actions	6	
Implement Player Victory	3	
Implement death case	3	
Implement Battle Rewards	3	
Implement different weapons	6	
Create Battle Help Menu	3	
Implement case for Boss battles	9	6
Create and implement testing system	9	2
Create bounds tests for battle system	6	6
total	57	22

Roast of Team Cafés

- Why are these individuals' hours so low?

Kyle		
Add items to inventory	1	2
Remove items from inventory	1	1
Static Testing	4	2
Dynamic Testing	6	
Create sprites	3	
Animate sprites	5	
Create pop-up for inventory	2	2
Exchange items in inventory	1	
Equip items to player	3	
Stack items	2	
Display combat stats in inventory	3	
Display shop inventory sub-class	9	
Sell items to shop	6	
total	46	7
Trevor		
Demo Mode World AI (AI Specialist Role)	9	
Enemy AI	8	
Demo Mode Battle AI	4	
Get Player Values & Enemy Values	6	
totals	27	

	Predicted Time (hrs)	Actual Time (hrs)
Jon		
Implement Quest system	6	9
Implement Simple Dialoge System	6	
Map out main storyline	6	
Create element to display NPC text	3	
Add essential NPC's to main story	6	
Map out several different side quests	9	
Add side quest NPC's	9	
total	33	
Ross		
Create Main world map	5	3
Movement (including accessing other places)	5	5
Create barries/traps	5	2
Create Interactable player objects	3	3
Testing	9	
Create dungeons/ other areas	5	
total	32	13

Roast of the Team Cafés

Isn't the Manage Player information going to create a high degree of coupling?