Principles of Software Construction: Objects, Design, and Concurrency (Part 2: Designing (Sub-)Systems)

Assigning Responsibilities

Jonathan Aldrich Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

Learning Goals

- Apply GRASP patterns to assign responsibilities in designs
- Reason about tradeoffs among designs

Today’s topics

- Object-Oriented Design: “After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements.”
- But how?
  – How should concepts be implemented by classes?
  – What method belongs where?
  – How should the objects interact?
  – This is a critical, important, and non-trivial task

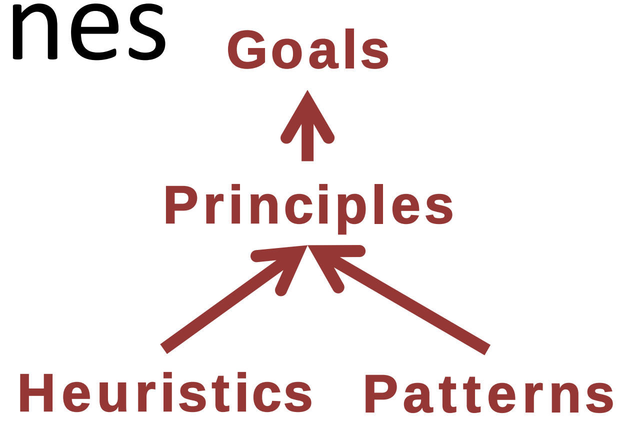
Responsibilities

- Responsibilities are related to the obligations of an object in terms of its behavior.
- Two types of responsibilities:
  - knowing
  - doing
- Doing responsibilities of an object include:
  - doing something itself, such as creating an object or doing a calculation
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- Knowing responsibilities of an object include:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

Design Goals, Principles, and Patterns

- Design Goals
  – Design for change, understanding, reuse, division of labor, …
- Design Principle
  – Low coupling, high cohesion
  – Low representational gap
  – Law of demeter
- Design Heuristics (GRASP)
  – Information expert
  – Creator
  – Controller

Goals, Principles, Guidelines

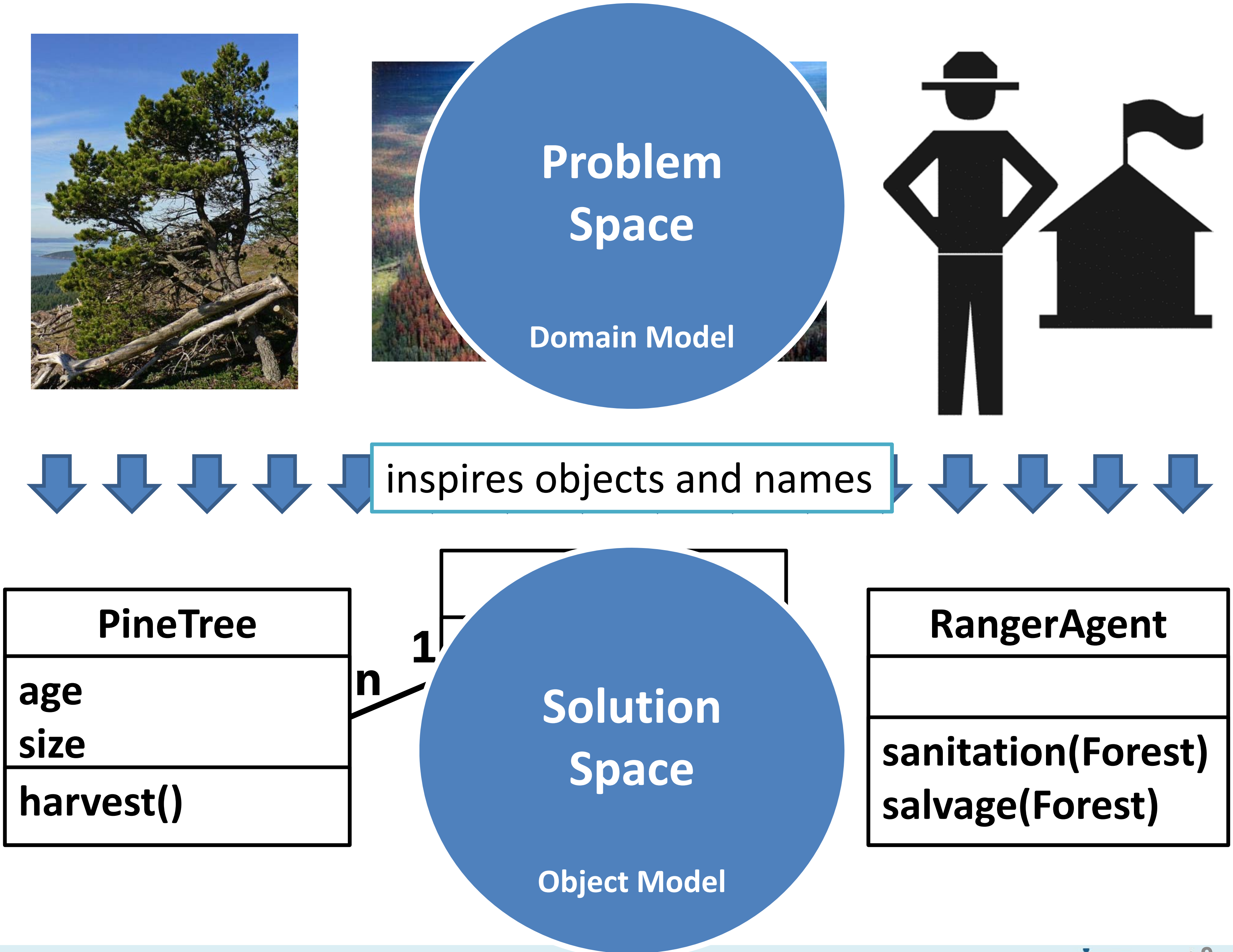


- Design Goals
  - Desired quality attributes of software
  - Driven by cost/benefit economics
  - Examples: design for change, understanding, reuse, …
- Design Principles
  - Guidelines for designing software
  - Support one or more design goals
  - Examples: Information hiding, low repr. gap, low coupling, high cohesion, …
- Design Heuristics
  - Rules of thumb for **low-level** design decisions
  - Promote design principles, and ultimately design goals
  - Example: Creator, Expert, Controller
- Design Patterns
  - General solutions to recurring design problems
  - Promote design goals, but may add complexity or involve tradeoffs
  - Examples: Decorator, Strategy, Template Method
- Goals, principles, heuristics, patterns may conflict
  - Use high-level goals of project to resolve

GRASP Patterns

- GRASP = General Responsibility Assignment Software Patterns
- Patterns of assigning **responsibilities**
  - reason about design trade-offs when assigning methods and fields to classes
- The GRASP patterns are a **learning aid** to
  - help one understand essential object design
  - apply design reasoning in a methodical, rational, explainable way
  - lower level and more local reasoning than most **design patterns**

DESIGN PRINCIPLE:
LOW REPRESENTATIONAL GAP


Problem
Space
Domain Model
inspires objects and names
PineTree
age
size
harvest()
n
1
Solution
Space
Object Model
RangerAgent
sanitation(Forest)
salvage(Forest)

Designs with Low Representational Gap

- Create software class for each domain class, create corresponding relationships
- Design goal: Design for change
- This is only a starting point!
  - Not all domain classes need software correspondence; pure fabrications might be needed
  - Other principles often more important

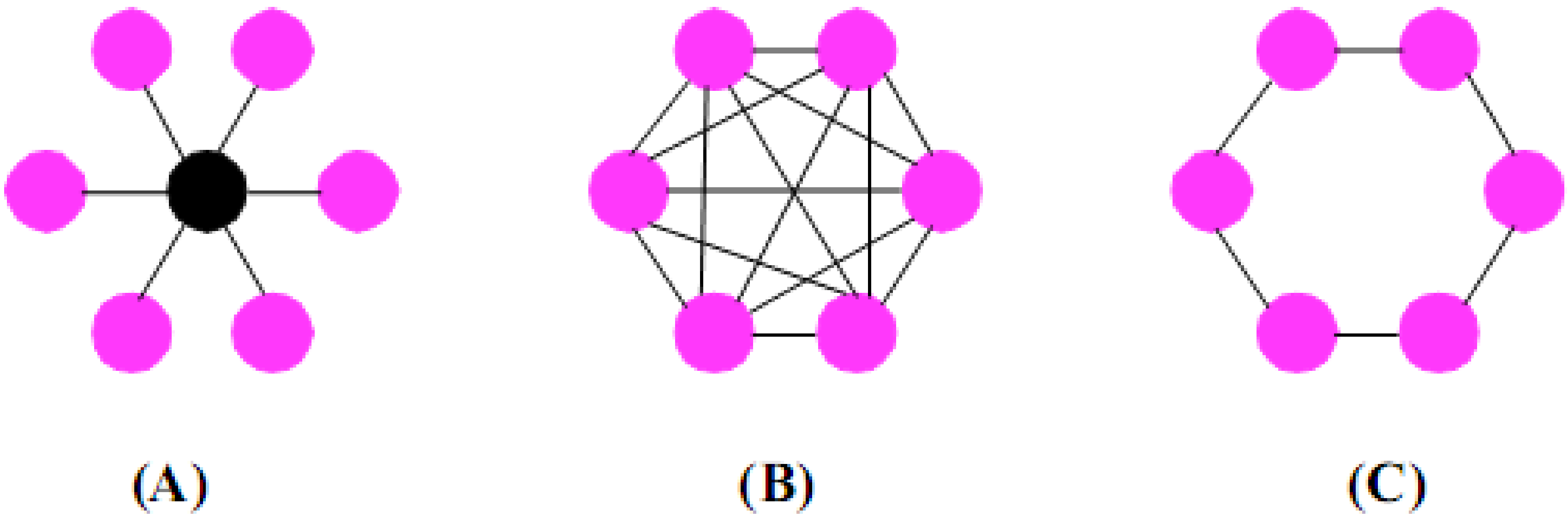
DESIGN PRINCIPLE: LOW COUPLING

Design Principle: Low Coupling

A module should depend on as few other modules as possible

- Enhances understandability (design for underst.)
  – Limited understanding of context, easier to understand in isolation
- Reduces the cost of change (design for change)
  – Little context necessary to make changes
  – When a module interface changes, few modules are affected (reduced rippling effects)
- Enhances reuse (design for reuse)
  – Fewer dependencies, easier to adapt to a new context

Topologies with different coupling



High Coupling is undesirable

- Element with low coupling depends on only few other elements (classes, subsystems, …)
  - "few" is context-dependent
- A class with high coupling relies on many other classes
  - Changes in related classes force local changes; changes in local class forces changes in related classes (brittle, rippling effects)
  - Harder to understand in isolation.
  - Harder to reuse because requires additional presence of other dependent classes
  - Difficult to extend – changes in many places

Which classes are coupled?
How can coupling be improved?

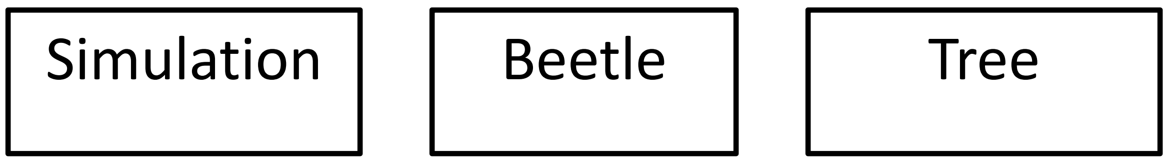
```
class Shipment {
        private List<Box> boxes;
        int getWeight() {
                int w=0;
                for (Box box: boxes)
                        for (Item item: box.getItems())
                                w += item.weight;
                return w;
}
class Box {
        private List<Item> items;
        Iterable<Item> getItems() { return items;}
}
class Item {
        Box containedIn;
        int weight;
}
```

A different design.
How can coupling be improved?

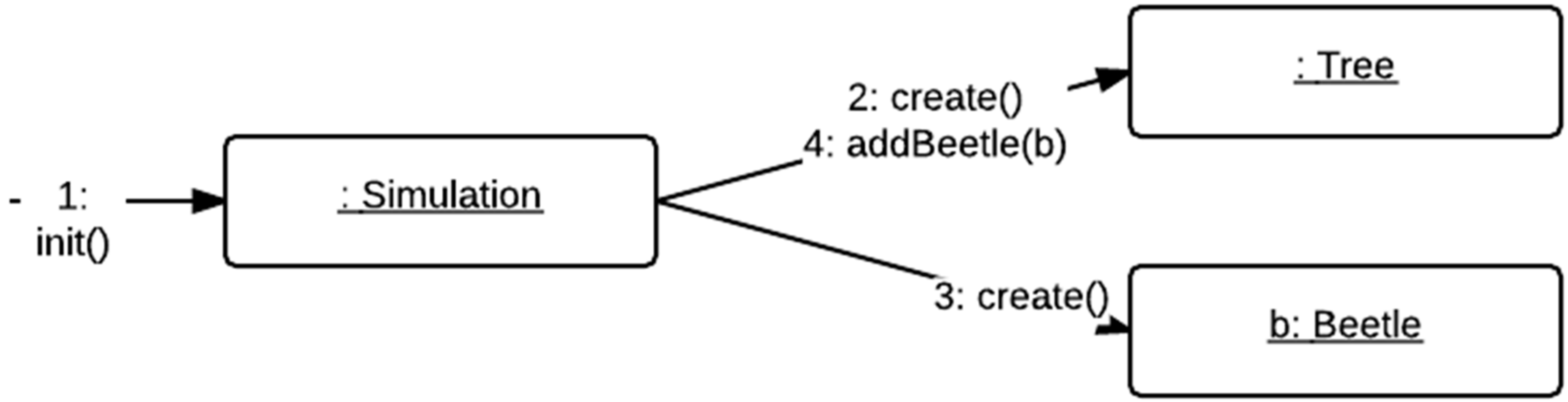
```
class Box {
        private List<Item> items;
        private Map<Item,Integer> weights;
        Iterable<Item> getItems() { return items;}
        int getWeight(Item item) { return weights.get(item);}
}
class Item {
        private Box containedIn;
        int getWeight() { return containedIn.getWeight(this);}
}
```

Coupling Example

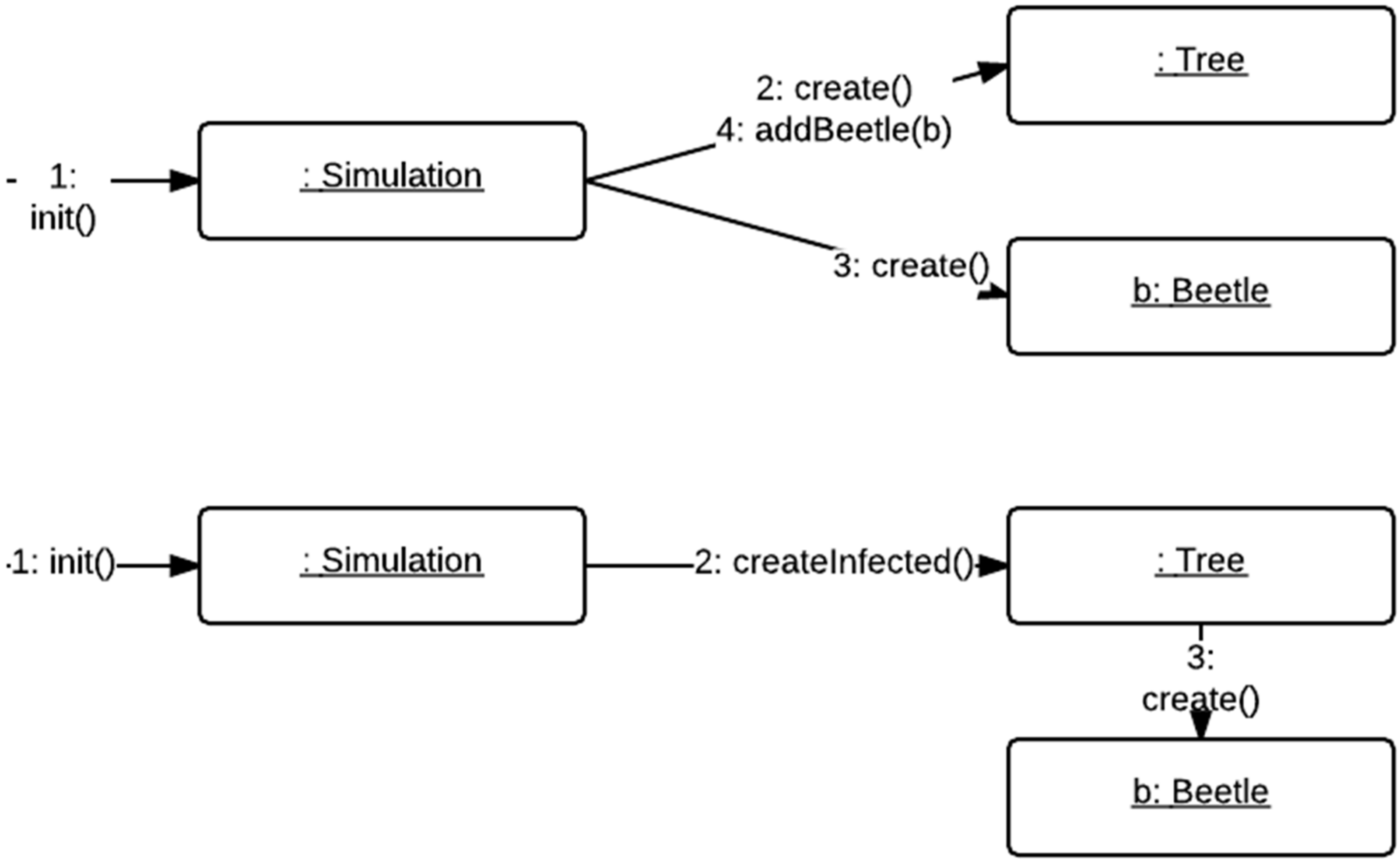
- Create a Tree and “infest” it with beetles



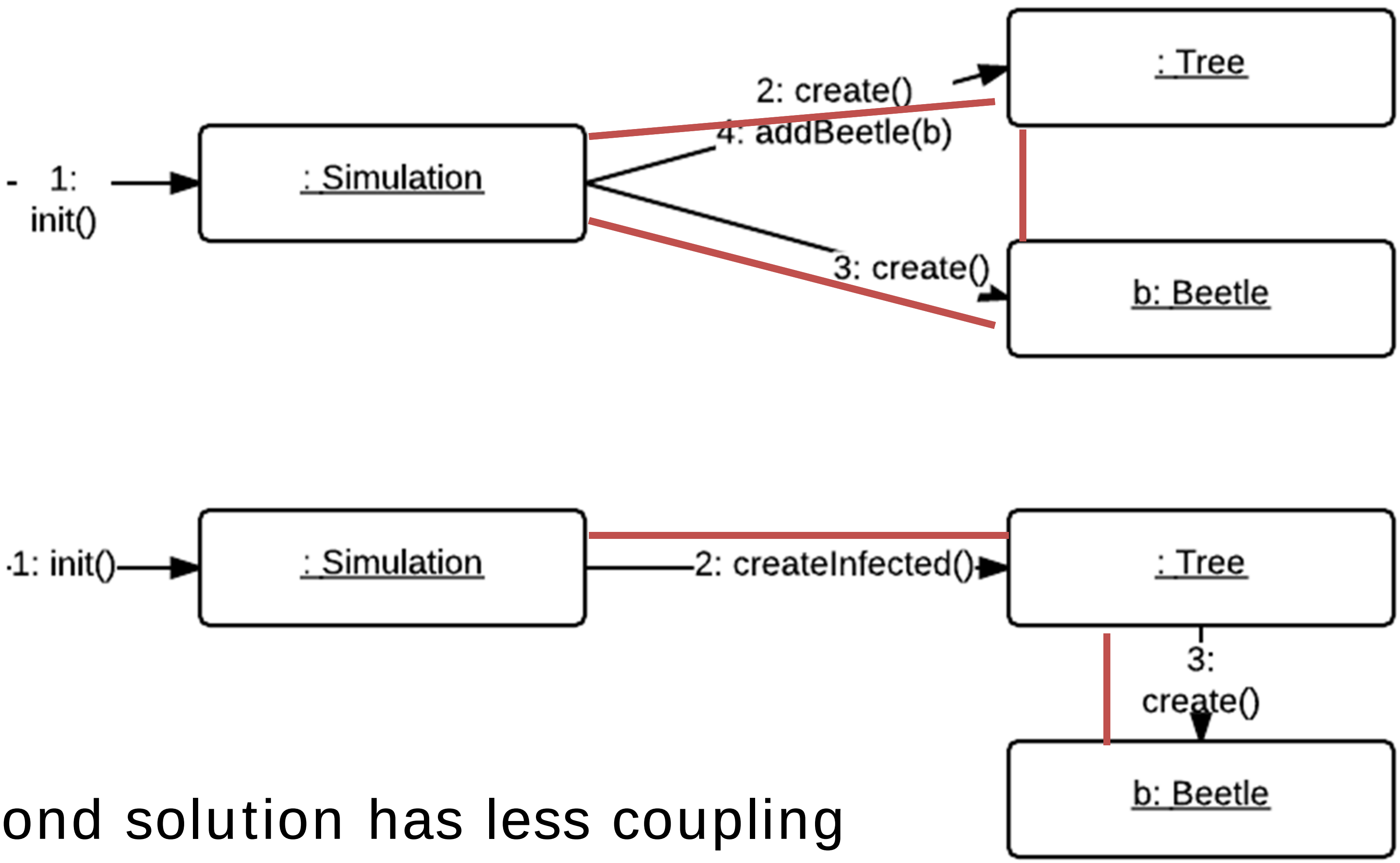
Coupling Example



Coupling Example



Coupling Example



Second solution has less coupling
Simulation does not know about Beetle class

Common Forms of Coupling in OO Languages

- Type X has a field of type Y
- Method m in type X refers to type Y
  - e.g. a method argument, return value, local variable, or static method call
- Type X is a direct or indirect subclass of Type Y
- Type Y is an interface, and Type X implements that interface

Low Coupling: Discussion

- Low Coupling 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.
- Low Coupling supports design of more independent classes; reduces the impact of change.
- Context-dependent; should be considered together with cohesion and other principles and patterns
- Prefer coupling to interfaces over coupling to implementations

Law of Demeter

- *Each module should have only limited knowledge about other units: only units "closely" related to the current unit*
- In particular: Don't talk to strangers!
- For instance, no a.getB().getC().foo()

```
for (Item i: shipment.getBox().getItems())
        i.getWeight() …
```

Coupling: Discussion

- Subclass/superclass coupling is particularly strong
  - protected fields and methods are visible
  - subclass is fragile to many superclass changes, e.g. change in method signatures, added abstract methods
  - *Guideline: prefer composition to inheritance, to reduce coupling*
- High coupling to very stable elements is usually not problematic
  - A stable interface is unlikely to change, and likely well-understood
  - *Prefer coupling to interfaces over coupling to implementations*
- Coupling is one principle among many
  - Consider cohesion, low repr. gap, and other principles

Coupling to “non-standards”

- Libraries or platforms may include non-standard features or extensions
- Example: JavaScript support across Browsers
  - <div id=“e1”>old content</div>

W3C-compliant DOM standard

- In JavaScript…
  - MSIE: e1.innerText = “new content”
  - Firefox: e1.textContent = “new content”

Design Goals

- Explain how low coupling supports
  - design for change
  - design for understandability
  - design for division of labor
  - design for reuse
  - …

26

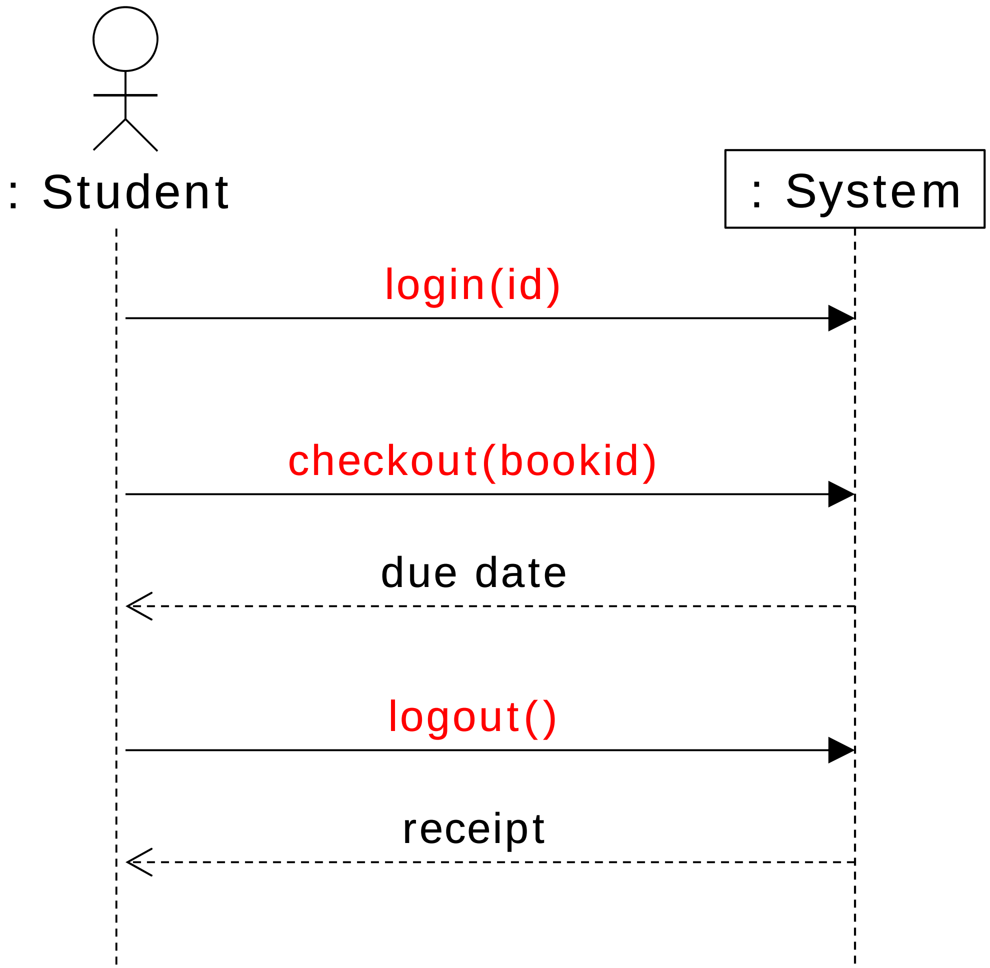
Design Goals

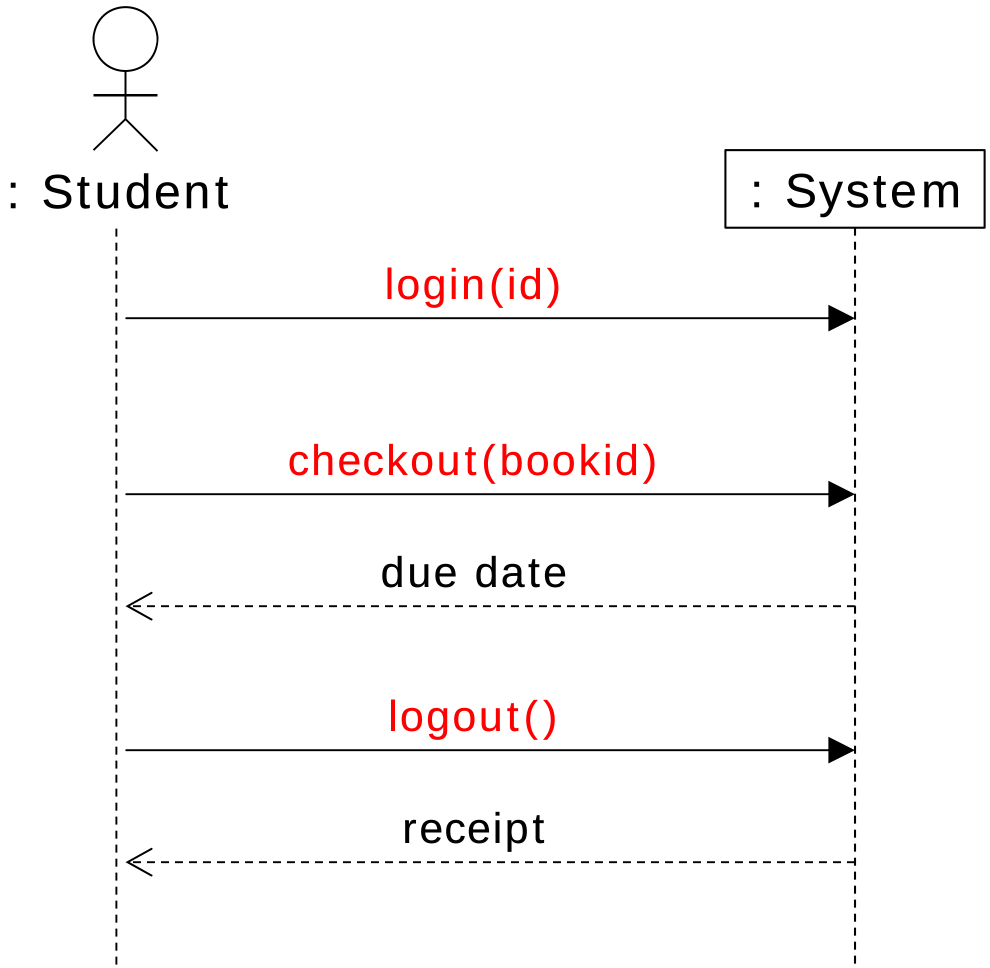
- design for change
  - changes easier because fewer dependencies on fewer other objects
  - changes are less likely to have rippling effects
- design for understandability
  - fewer dependencies to understand (e.g., a.getB().getC().foo())
- design for division of labor
  - smaller interfaces, easier to divide
- design for reuse
  - easier to reuse without complicated dependencies

GRASP PATTERN: CONTROLLER
DESIGN PATTERN: FAÇADE

Controller (GRASP)

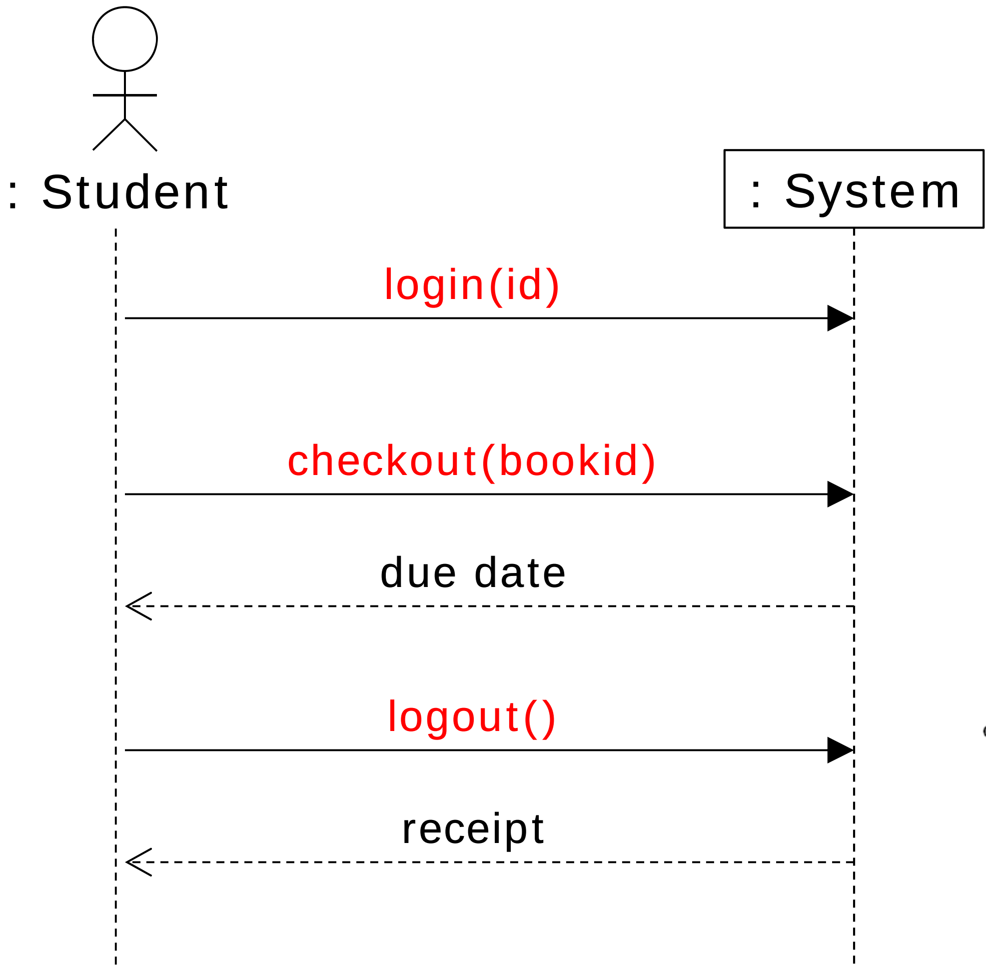
- Problem: What object receives and coordinates a system operation (event)?
- Solution: Assign the responsibility to an object representing
  - the overall system, device, or subsystem (façade controller), or
  - a use case scenario within which the system event occurs (use case controller)


: Student
: System
login(id)
checkout(bookid)
due date
logout()
receipt

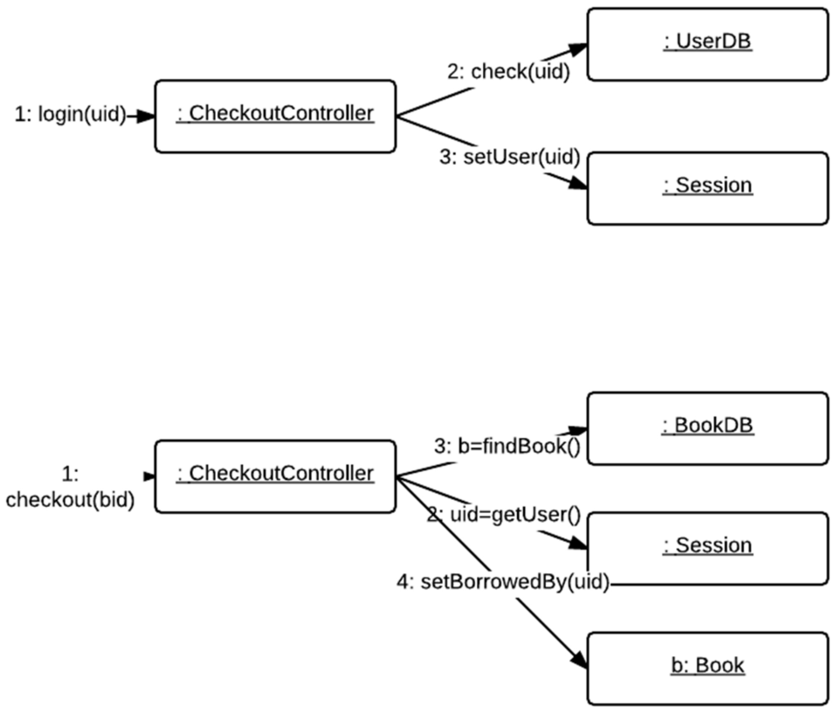


CheckoutController

login(id: Int)
checkout(bid: Int)
logout()



: Student
: System
login(id)
checkout(bookid)
due date
logout()
receipt



1: login(uid)
: CheckoutController
2: check(uid)
: UserDB
3: setUser(uid)
: Session
1: checkout(bid)
: CheckoutController
3: b=findBook()
: BookDB
2: uid=getUser()
: Session
4: setBorrowedBy(uid)
b: Book

Controller: Discussion

- A Controller is a coordinator
  - does not do much work itself
  - delegates to other objects
- Façade controllers suitable when not "too many" system events
  - -> one overall controller for the system
- Use case controller suitable when façade controller "bloated" with excessive responsibilities (low cohesion, high coupling)
  - -> several smaller controllers for specific tasks

- Closely related to Façade design pattern (future lecture)

Controller: Discussion of Design Goals/Strategies

- Decrease coupling
  - User interface and domain logic are decoupled from each other
    - Understandability: can understand these in isolation, leading to:
    - Evolvability: both the UI and domain logic are easier to change
  - Both are coupled to the controller, which serves as a mediator, but this coupling is less harmful
    - The controller is a smaller and more stable interface
    - Changes to the domain logic affect the controller, not the UI
    - The UI can be changed without knowing the domain logic design
- Support reuse
  - Controller serves as an interface to the domain logic
  - Smaller, explicit interfaces support evolvability
- But, bloated controllers increase coupling and decrease cohesion; split if applicable

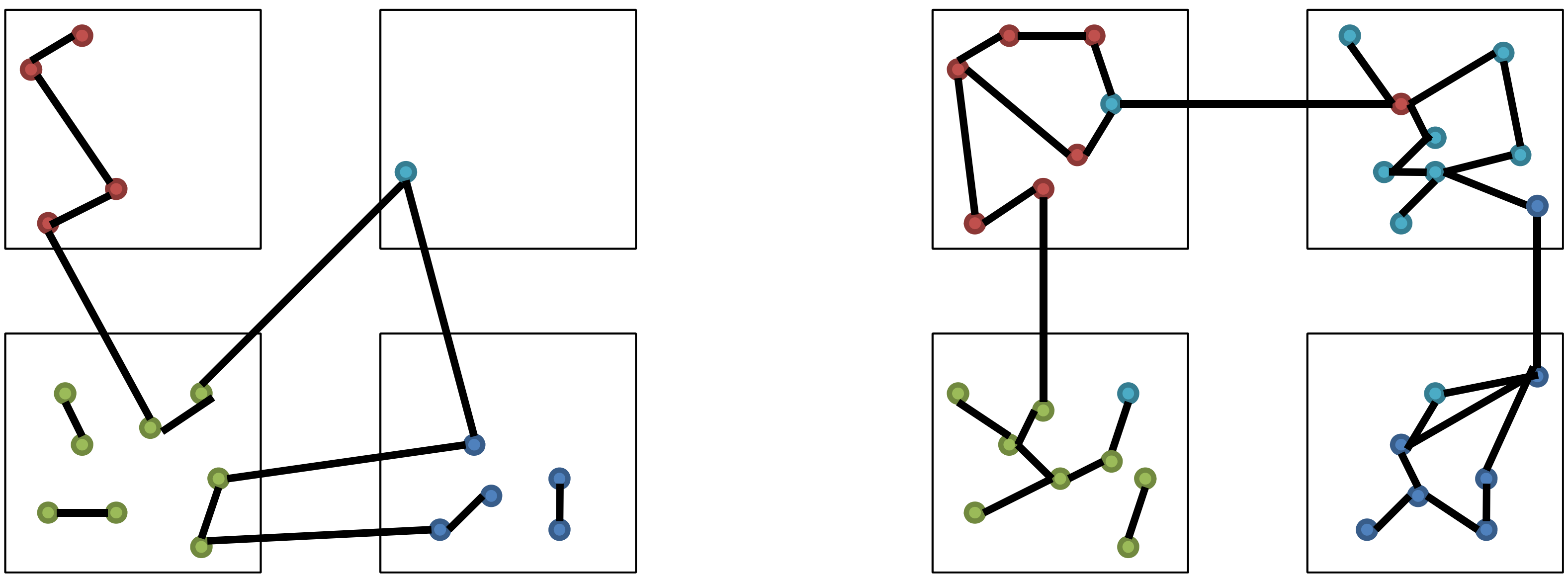
DESIGN PRINCIPLE: HIGH COHESION

35

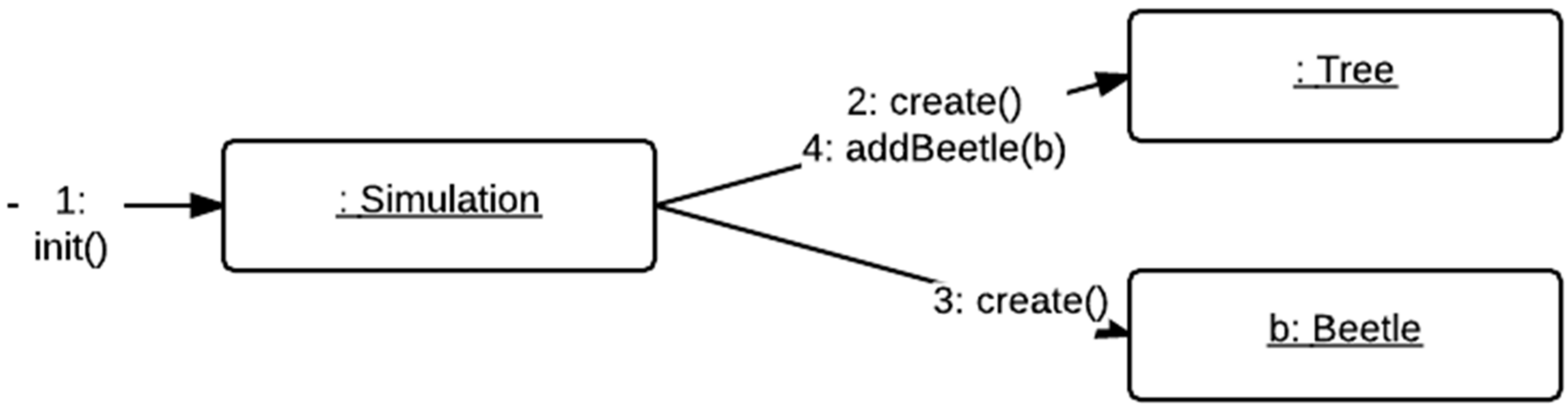
Design Principle: Cohesion

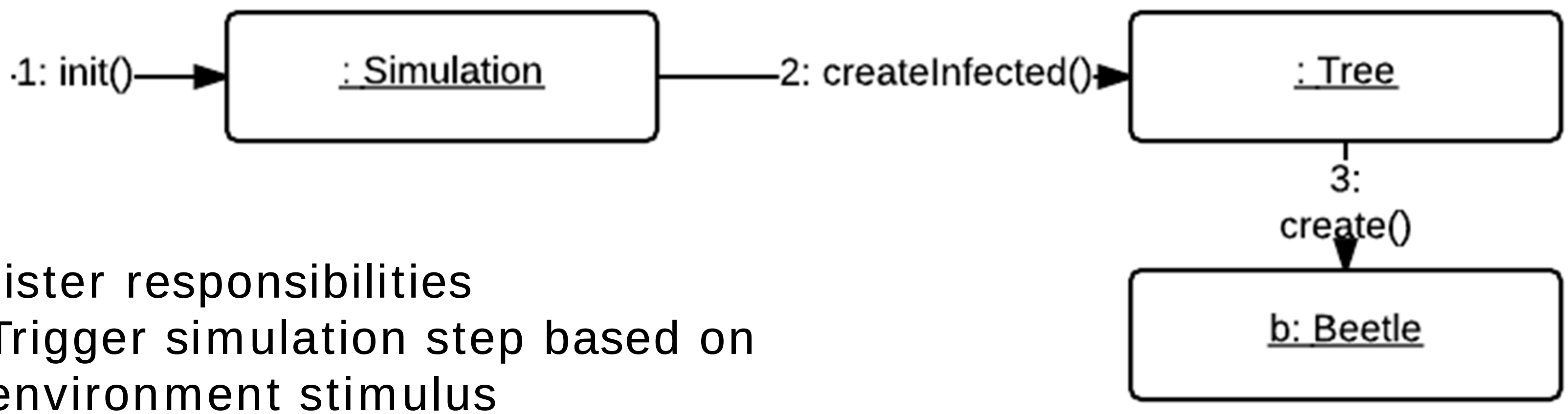
A module should have a small set of related responsibilities

- Enhances understandability (design for understandability)
  - A small set of responsibilities is easier to understand
- Enhances reuse (design for reuse)
  - A cohesive set of responsibilities is more likely to recur in another application



Cohesion in Simulation Example





Register responsibilities

- Trigger simulation step based on environment stimulus
- Coordinate creation of domain objects

```
class DatabaseApplication
        //... database fields
        //... Logging Stream
        //... Cache Status
        public void authorizeOrder(Data data, User currentUser, ...){
                // check authorization
                // lock objects for synchronization
                // validate buffer
                // log start of operation
                // perform operation
                // log end of operation
                // release lock on objects
        }
        public void startShipping(OtherData data, User currentUser, ...){
                // check authorization
                // lock objects for synchronization
                // validate buffer
                // log start of operation
                // perform operation
                // log end of operation
                // release lock on objects
        }
}
```

Cohesion in Graph Implementations

```
class Graph {
        Node[] nodes;
        boolean[] isVisited;
}
class Algorithm {
        int shortestPath(Graph g, Node n, Node m) {
                for (int i; …)
                        if (!g.isVisited[i]) {
                                …
                                g.isVisited[i] = true;
                        }
                }
                return v;
        }
}
```

Graph is tasked with not just data, but also algorithmic responsibilities

Monopoly Example

Which design has higher cohesion?

```
class Player {
        Board board;
        /* in code somewhere… */ getSquare(n);
        Square getSquare(String name) {
                for (Square s: board.getSquares())
                        if (s.getName().equals(name))
                                return s;
                return null;
}}
```

```
class Player {
        Board board;
        /* in code somewhere… */ board.getSquare(n);
}
class Board{
        List<Square> squares;
        Square getSquare(String name) {
                for (Square s: squares)
                        if (s.getName().equals(name))
                                return s;
                return null;
}}
```

Hints for Identifying Cohesion

- Use one color per concept
- Highlight all code of that concept with the color
- => Classes/ methods should have few colors

Hints for Identifying Cohesion

- There is no clear definition of what is a "concept"
- Concepts can be split into smaller concepts
  - Graph with search vs. Basic Graph + Search Algorithm vs. Basic Graph + Search Framework + Concrete Search Algorithm etc
- Requires engineering judgment

Cohesion: Discussion

- 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
- Advantages of high cohesion
  - Classes are easier to maintain
  - Easier to understand
  - Often support low coupling
  - Supports reuse because of fine grained responsibility
- Rule of thumb: a class with high cohesion has relatively few methods of highly related functionality; does not do too much work

Coupling vs Cohesion (Extreme cases)

Think about extreme cases:

- Very low coupling?
- Very high cohesion?

```
class Graph {
        Node[] nodes;
        boolean[] isVisited;
}
class Algorithm {
        int shortestPath(Graph g, Node n, Node m) {
                for (int i; …)
                        if (!g.isVisited[i]) {
                                …
                                g.isVisited[i] = true;
                        }
                }
                return v;
        }
}
```

Coupling vs Cohesion (Extreme cases)

- All code in one class/method
  - very low coupling, but very low cohesion
- Every statement separated
  - very high cohesion, but very high coupling

- Find good tradeoff; consider also other principles, e.g., low representational gap

GRASP PATTERN: INFORMATION EXPERT

Information Expert (GRASP Pattern/Design Heuristic)

- Heuristic**: Assign a responsibility to the class that has the information necessary to fulfill the responsibility**
- Start assigning responsibilities by clearly stating responsibilities!
- Typically follows common intuition
- Software classes instead of Domain Model classes
  - If software classes do not yet exist, look in Domain Model for fitting abstractions (-> correspondence)

Which class has all the information to compute the shipment's weight?

```
class Shipment {
        private List<Box> boxes;
        int getWeight() {
                int w=0;
                for (Box box: boxes)
                        for (Item item: box.getItems())
                                w += item.weight;
                return w;
}
class Box {
        private List<Item> items;
        Iterable<Item> getItems() { return items;}
}
class Item {
        Box containedIn;
        int weight;
}
```

Information Expert -> "Do It Myself Strategy"

- Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real-world thing it represents
  - a sale does not tell you its total; it is an inanimate thing
- In OO design, all software objects are "alive" or "animated," and they can take on responsibilities and do things.
- They do things related to the information they know.

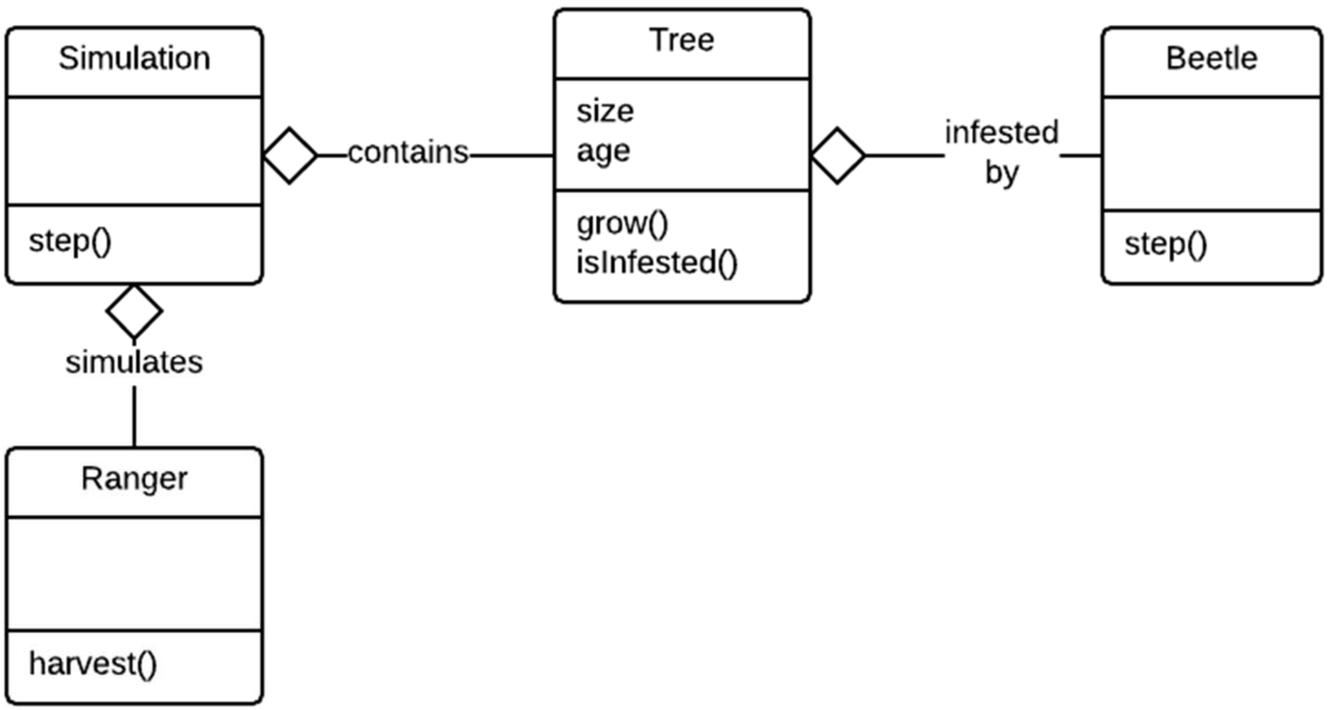
GRASP PATTERN: CREATOR

Creator (GRASP Pattern/Design Heuristic)

- Problem: Who creates an A?
- Solution: **Assign class responsibility of creating instance of class A to B if**
  - B aggregates A objects
  - B contains A objects
  - B records instances of A objects
  - B closely uses A objects
  - B has the initializing data for creating A objects
- the more the better; where there is a choice, prefer
  - B aggregates or contains A objects
- Key idea: Creator needs to keep reference anyway and will frequently use the created object

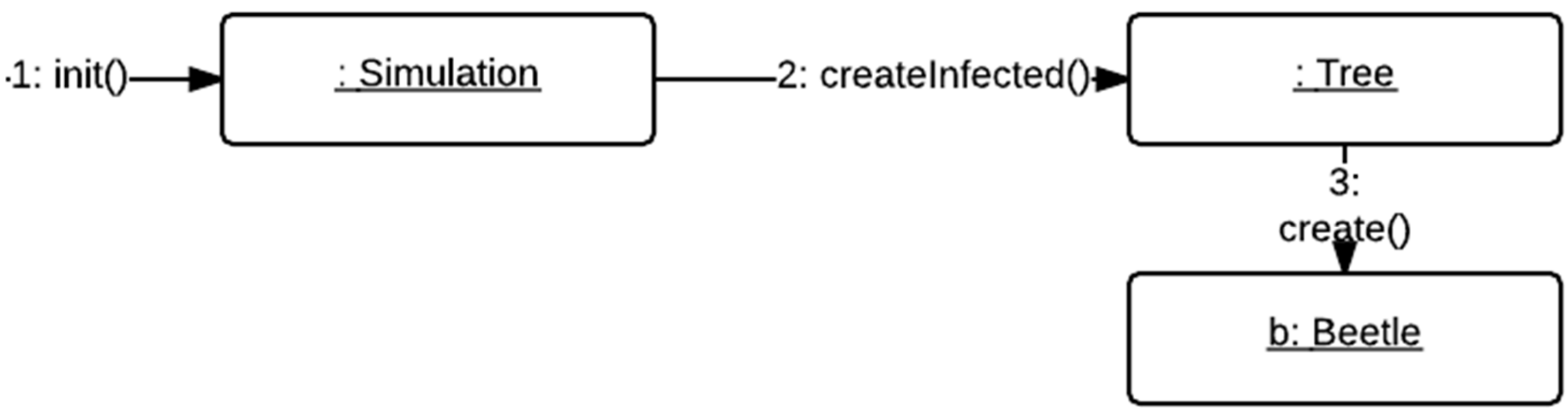
Creator (GRASP)

- Who is responsible for **creating** Beetle objects? Tree objects?



Creator : Example

- Who is responsible for creating Beetle objects?
  – Creator pattern suggests Tree
- Interaction diagram:



Creator (GRASP)

- Problem: Assigning responsibilities for creating objects
  - Who creates Nodes in a Graph?
  - Who creates instances of SalesItem?
  - Who creates Children in a simulation?
  - Who creates Tiles in a Monopoly game?
    - AI? Player? Main class? Board? Meeple (Dog)?

Creator: Discussion of Design Goals/Principles

- Promotes **low coupling**, **high cohesion**
  - class responsible for creating objects it needs to reference
  - creating the objects themselves avoids depending on another class to create the object
- Promotes **evolvability** (design for change)
  - Object creation is hidden, can be replaced locally

- Contra: sometimes objects must be created in special ways
  - complex initialization
  - instantiate different classes in different circumstances
  - *then **cohesion** suggests putting creation in a different object*
    - see *design patterns* such as builder, factory method

Take-Home Messages

- Design is driven by quality attributes
  – Evolvability, separate development, reuse, performance, …
- Design principles provide guidance on achieving qualities
  – Low coupling, high cohesion, high correspondence, …
- GRASP design heuristics promote these principles
  – Creator, Expert, Controller, …

Which design is better? Argue with design goals, principles, heuristics, and patterns that you know