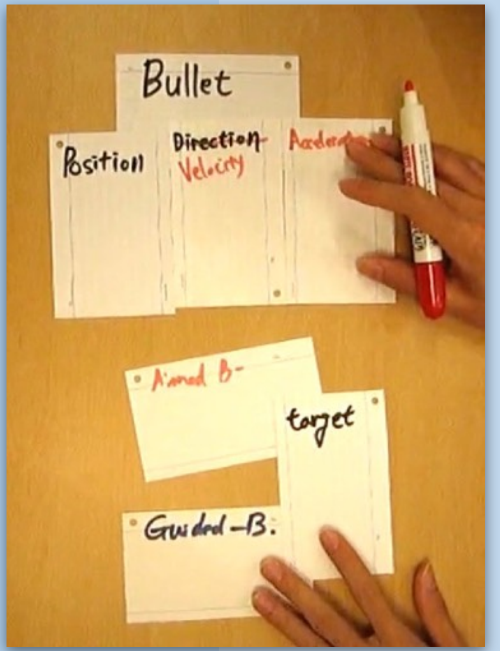
Introduction to Software Engineering

Responsibility-Driven Design

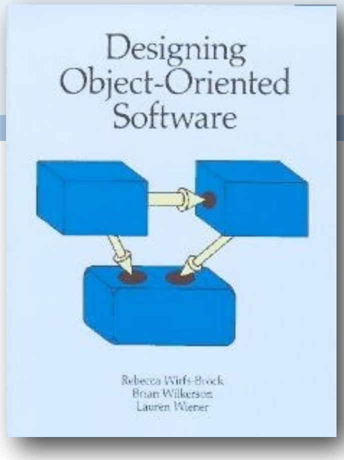


Roadmap

> Responsibility-Driven Design
  - Finding Classes
  - Class Selection Rationale
  - CRC sessions
  - Identifying Responsibilities
  - Finding Collaborations
  - Structuring Inheritance Hierarchies

> SOLID object-oriented design principles

Bibliography



- **Designing Object-Oriented Software, R. Wirfs-Brock, B. Wilkerson, L. Wiener, Prentice Hall, 1990.**
- Design Principles and Design Patterns, R.C. Martin, 2000
- Object-Oriented Design Heuristics, A.Riel, 2000

Responsibility-Driven Design (RDD) was developed by Rebecca Wirfs-Brock and colleagues in the mid 1980s as they made some of the first industrial experiences in applying object-oriented programming to real-world projects with Smalltalk.

The central idea is that in a “good” object-oriented design, every object has clear and well-defined responsibilities. Responsibilities are not concentrated centrally, but are well-distributed amongst the objects. Such a design can evolve gracefully since changes impact only objects responsible for the affected behaviour and their direct collaborators.

RDD, together with Design by Contract (DbC) are considered to be cornerstones of object-oriented development methods, and they have proven to work well with other process-oriented methodologies, such as agile development.

Roadmap

> **Responsibility-Driven Design**
  - Finding Classes
  - Class Selection Rationale
  - CRC sessions
  - Identifying Responsibilities
  - Finding Collaborations
  - Structuring Inheritance Hierarchies
> SOLID object-oriented design principles

Why Responsibility-driven Design?

Functional Decomposition:

Decompose according to the functions a system is supposed to perform.

- Good in a “waterfall” approach: stable requirements and one monolithic function

However

- Naive: Modern systems perform more than one function
- Maintainability: system functions evolve ⇒ redesign affect whole system
- Interoperability: interfacing with other system is difficult

Classical, top-down functional or procedural decomposition yields a system built out of procedures calling other procedures and manipulating shared data structures. This approach works very well for algorithmic applications with very stable requirements, but works less well for rapidly evolving application domains. The problem is that the *design focuses on the functions performed*, but these *may not be very stable*, leading to a lot of re-design over time.

Why Responsibility-driven Design?

Object-Oriented Decomposition:

Decompose according to the objects a system is supposed to manipulate.

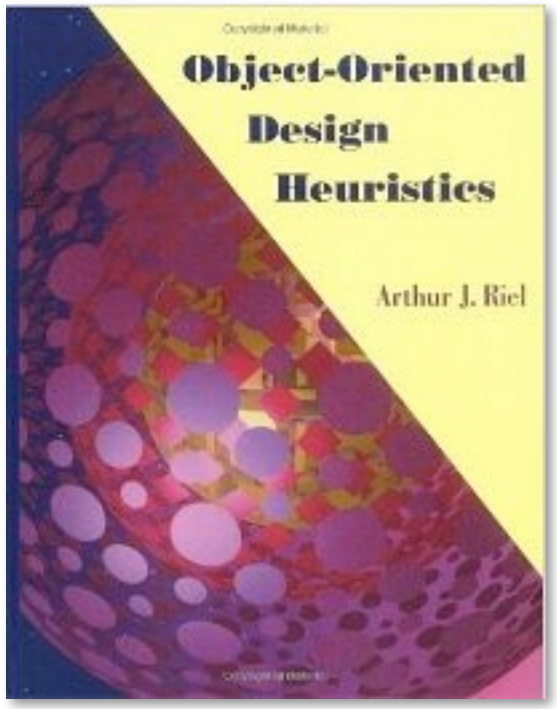
— Better for complex and evolving systems

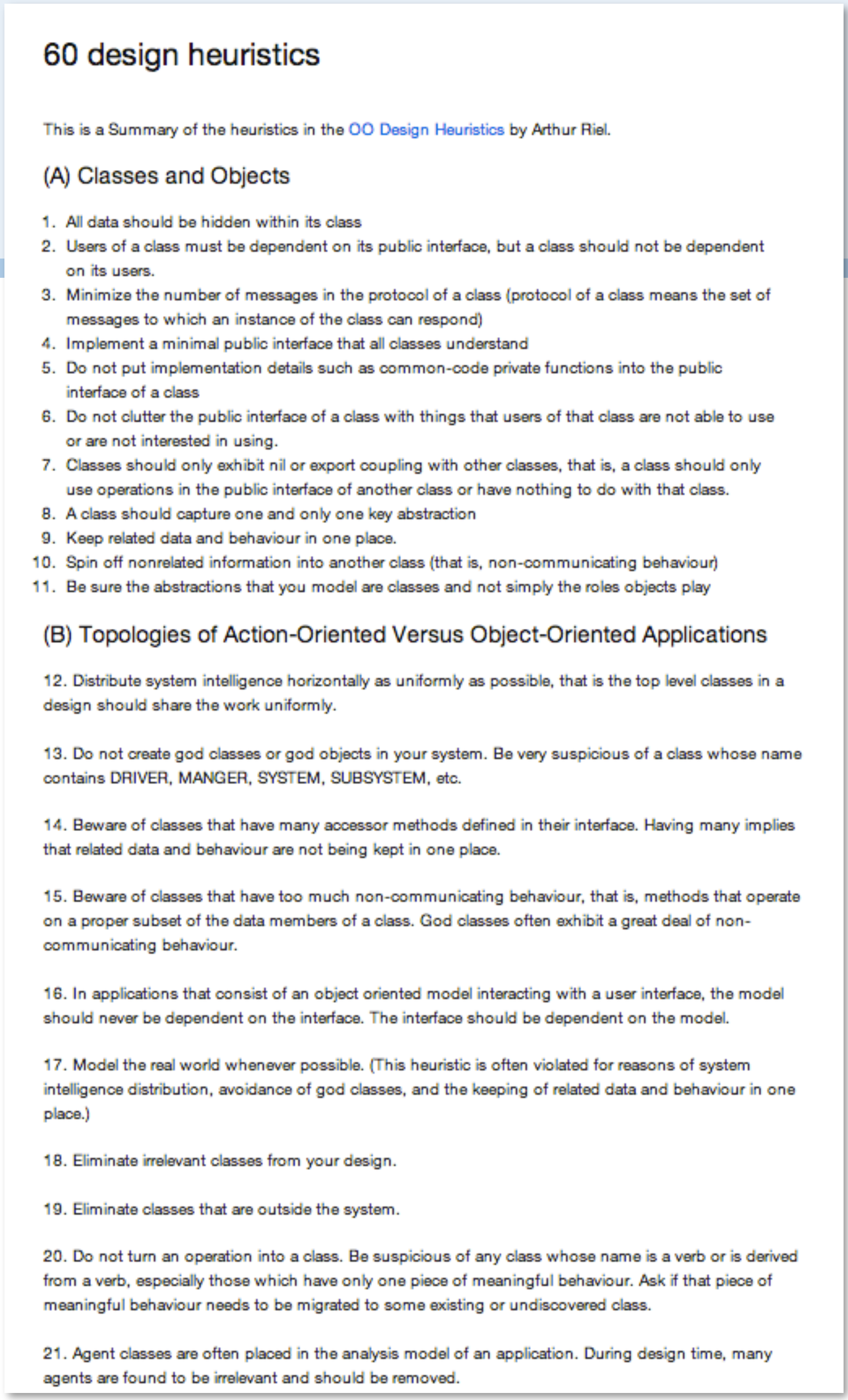
However

— How to find the objects?

An object-oriented design is centred around the “objects”, i.e., the *domain concepts*, of a particular application. Even in a rapidly evolving application, *the underlying domain concepts tend to be stable*. As a consequence, a good object-oriented design tends to be more robust over time than a functional design.

Design is not algorithmic





60 design heuristics

This is a Summary of the heuristics in the OO Design Heuristics by Arthur Riel.

(A) Classes and Objects

1. All data should be hidden within its class
2. Users of a class must be dependent on its public interface, but a class should not be dependent on its users.
3. Minimize the number of messages in the protocol of a class (protocol of a class means the set of messages to which an instance of the class can respond)
4. Implement a minimal public interface that all classes understand
5. Do not put implementation details such as common-code private functions into the public interface of a class
6. Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.
7. Classes should only exhibit nil or export coupling with other classes, that is, a class should only use operations in the public interface of another class or have nothing to do with that class.
8. A class should capture one and only one key abstraction
9. Keep related data and behaviour in one place.
10. Spin off nonrelated information into another class (that is, non-communicating behaviour)
11. Be sure the abstractions that you model are classes and not simply the roles objects play

(B) Topologies of Action-Oriented Versus Object-Oriented Applications

12. Distribute system intelligence horizontally as uniformly as possible, that is the top level classes in a design should share the work uniformly.

13. Do not create god classes or god objects in your system. Be very suspicious of a class whose name contains DRIVER, MANGER, SYSTEM, SUBSYSTEM, etc.

14. Beware of classes that have many accessor methods defined in their interface. Having many implies that related data and behaviour are not being kept in one place.

15. Beware of classes that have too much non-communicating behaviour, that is, methods that operate on a proper subset of the data members of a class. God classes often exhibit a great deal of non-communicating behaviour.

16. In applications that consist of an object oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.

17. Model the real world whenever possible. (This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behaviour in one place.)

18. Eliminate irrelevant classes from your design.

19. Eliminate classes that are outside the system.

20. Do not turn an operation into a class. Be suspicious of any class whose name is a verb or is derived from a verb, especially those which have only one piece of meaningful behaviour. Ask if that piece of meaningful behaviour needs to be migrated to some existing or undiscovered class.

21. Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

Engineering is about best practices, as opposed to science, which is about empirically discovering knowledge.

Accordingly, a design method provides guidelines, not fixed rules for designing software. This classic book by Arthur Riel describes a set of such guidelines formulated as “heuristics”. A summary can be found online:

http://scg.unibe.ch/teaching/ese/60-design-heuristics

Expertise matters

A good *sense of style* often helps produce clean, elegant designs

Experienced software designers know intuitively what makes a design good or bad.

Good designs are easy to understand, maintain and adapt; poor designs make further development painful.

RDD Steps

1. Find the *classes* in your system
2. Determine the *responsibilities* of each class
3. Determine how objects *collaborate* with each other to fulfill their responsibilities
4. *Factor* common responsibilities to build class hierarchies

RDD leads to software designs with well-distributed responsibilities.

The first step is to identify *potential classes* that correspond to domain concepts. We will explore some heuristics for identifying such candidate classes.

The next task is to identify *responsibilities* for these classes. “Responsibilities” are *conceptual*, and do not refer to specific methods or interfaces, for example, “compute possible meeting dates for a committee”.

Collaborations refer to the other objects needed to fulfil a responsibility.

Finally, responsibilities should be factored to build conceptual clean class hierarchies.

Roadmap

- Responsibility-Driven Design
  - **Finding Classes**
  - Class Selection Rationale
  - CRC sessions
  - Identifying Responsibilities
  - Finding Collaborations
  - Structuring Inheritance Hierarchies
- SOLID object-oriented design principles

Finding Classes

> Start with requirements specification:
  - What are the *goals* of the system being designed, its expected *inputs* and desired *responses*?

Drawing Editor Requirements Specification

The drawing editor is an interactive graphics editor. With it, users can create and edit drawings composed of lines, rectangles, ellipses and text.

Tools control the mode of operation of the editor. Exactly one tool is active at any given time.

Two kinds of tools exist: the selection tool and creation tools. When the selection tool is active, existing drawing elements can be selected with the cursor. One or more drawing elements can be selected and manipulated; if several drawing elements are selected, they can be manipulated as if they were a single element. Elements that have been selected in this way are referred to as the current selection. The current selection is indicated visually by displaying the control points for the element. Clicking on and dragging a control point modifies the element with which the control point is associated.

When a creation tool is active, the current selection is empty. The cursor changes in different ways according to the specific creation tool, and the user can create an element of the selected kind. After the element is created, the selection tool is made active and the newly created element becomes the current selection.

The text creation tool changes the shape of the cursor to that of an I-beam. The position of the first character of text is determined by where the user clicks the mouse button. The creation tool is no longer active when the user clicks the mouse button outside the text element. The control points for a text element are the four corners of the region within which the text is formatted. Dragging the control points changes this region. The other creation tools allow the creation of lines, rectangles and ellipses. They change the shape of the cursor to that of a crosshair. The appropriate element starts to be created when the mouse button is pressed, and is completed when the mouse button is released. These two events create the start point and the stop point.

The line creation tool creates a line from the start point to the stop point. These are the control points of a line. Dragging a control point changes the end point.

The rectangle creation tool creates a rectangle such that these points are diagonally opposite corners. These points and the other corners are the control points. Dragging a control point changes the associated corner.

The ellipse creation tool creates an ellipse fitting within the rectangle defined by the two points described above. The major radius is one half the width of the rectangle, and the minor radius is one half the height of the rectangle. The control points are at the corners of the bounding rectangle. Dragging control points changes the associated corner.

This example is taken from the RDD book. It consists of a typical textual description of a system to be built. We will deconstruct this description to identify domain concepts that can serve as candidate classes for our design.

Finding Classes ...

1. Look for *noun phrases*
   — separate into obvious classes, uncertain candidates, and nonsense

2. Refine to a list of candidate classes
   — apply heuristics to select the best candidates

Drawing Editor: noun phrases

The drawing editor is an interactive graphics editor. With it, users can create and edit drawings composed of lines, rectangles, ellipses and text.

Tools control the mode of operation of the editor. Exactly one tool is active at any given time.

Two kinds of tools exist: the selection tool and creation tools. When the selection tool is active, existing drawing elements can be selected with the cursor. One or more drawing elements can be selected and manipulated; if several drawing elements are selected, they can be manipulated as if they were a single element. Elements that have been selected in this way are referred to as the current selection. The current selection is indicated visually by displaying the control points for the element. Clicking on and dragging a control point modifies the element with which the control point is associated.

…

As a first step we simply highlight all the nouns and noun phrases. Many of these will be eliminated as we apply our heuristics.

When a creation tool is active, the current selection is empty. The cursor changes in different ways according to the specific creation tool, and the user can create an element of the selected kind. After the element is created, the selection tool is made active and the newly created element becomes the current selection.

The text creation tool changes the shape of the cursor to that of an I-beam. The position of the first character of text is determined by where the user clicks the mouse button. The creation tool is no longer active when the user clicks the mouse button outside the text element. The control points for a text element are the four corners of the region within which the text is formatted. Dragging the control points changes this region. The other creation tools allow the creation of lines, rectangles and ellipses. They change the shape of the cursor to that of a crosshair. The appropriate element starts to be created when the mouse button is pressed, and is completed when the mouse button is released. These two events create the start point and the stop point.

...

The line creation tool creates a line from the start point to the stop point. These are the control points of a line. Dragging a control point changes the end point.

The rectangle creation tool creates a rectangle such that these points are diagonally opposite corners. These points and the other corners are the control points. Dragging a control point changes the associated corner.

The ellipse creation tool creates an ellipse fitting within the rectangle defined by the two points described above. The major radius is one half the width of the rectangle, and the minor radius is one half the height of the rectangle. The control points are at the corners of the bounding rectangle. Dragging control points changes the associated corner.

Roadmap

> Responsibility-Driven Design
  - Finding Classes
  - **Class Selection Rationale**
  - CRC sessions
  - Identifying Responsibilities
  - Finding Collaborations
  - Structuring Inheritance Hierarchies

> SOLID object-oriented design principles

Class Selection Rationale

Model physical objects:

- ~~mouse button~~ [event or attribute]

Model conceptual entities:

- ellipse, line, rectangle
- Drawing, Drawing Element
- Tool, Creation Tool, Ellipse Creation Tool, Line Creation Tool, Rectangle Creation Tool, Selection Tool, Text Creation Tool
- text, Character
- Current Selection

Not all nouns or noun phrases will be candidate classes.

Physical or real-world entities are often important domain concepts. In this case there are none, since the application domain is a technical one. The only physical entity is a mouse, but the requirements description actual refers to "clicking" the mouse button, so it is the event that is important, not the object. We scratch it out.

Another heuristic is to model "*conceptual entities*", or *domain concepts*. Since the application is a drawing editor, the important domain concepts are drawings and drawing elements.

Class Selection Rationale ...

Choose one word for one concept:

- Drawing Editor ⇒ ~~editor~~, ~~interactive graphics editor~~
- Drawing Element ⇒ ~~element~~
- Text Element ⇒ ~~text~~
- Ellipse Element, Line Element, Rectangle Element ⇒ ~~ellipse~~, ~~line~~, ~~rectangle~~

Beware of synonyms. It is normal to vary ones descriptions in natural language to enrich the text, but this may lead to confusion in the exercise of identifying domain concepts. An important task is therefore to search for such synonyms and to introduce a single, normalized term for each concept.

Class Selection Rationale ...

Be wary of adjectives:

- Ellipse Creation Tool, Line Creation Tool, Rectangle Creation Tool, Selection Tool, Text Creation Tool
  - all have different requirements
- ~~bounding rectangle~~, ~~rectangle~~, ~~region~~ ⇒ Rectangle
  - common meaning, but different from Rectangle Element
- Point ⇒ ~~end point~~, ~~start point~~, ~~stop point~~
- Control Point
  - more than just a coordinate
- corner ⇒ ~~associated corner~~, ~~diagonally opposite corner~~
  - no new behaviour

Adjectives sometimes introduce a specialized domain concept. A "Line Creation Tool" is a special kind of Tool just for creating lines. A Control Point is a special kind of point for controlling the shape of a drawing element.

In other cases adjectives may just indicate a special state of a given entity, or its relation to other entities. An end point and a start point are both Point objects, just occurring at different ends of a Line Element.

Class Selection Rationale ...

Be wary of sentences with missing or misleading subjects:

- "The current selection is indicated visually by displaying the control points for the element."
  - by what? Assume Drawing Editor ...

Model categories:

- Tool, Creation Tool

Model interfaces to the system: — no good candidates here ...

- ~~user~~ — *don't need to model user explicitly*
- ~~cursor~~ — *cursor motion handled by operating system*

Sentences written in the passive voice are missing a subject. That subject might be an important, but unstated domain concept.

Categories of domain concepts are important to capture. In a later phase we might organise them into a specialization hierarchy, but at this stage it is enough to capture them. There are several different kinds of tools. Choose a unique name for each of them.

It might be important to model interfaces to a system, that is, where information enters or leaves the system. This application is pretty much self-contained, so there are no good examples.

Class Selection Rationale ...

Model values of attributes, not attributes themselves:

- ~~height of the rectangle~~, ~~width of the rectangle~~
- ~~major radius~~, ~~minor radius~~
- ~~position~~ — *of first text character; probably Point attribute*
- ~~mode of operation~~ — *attribute of Drawing Editor*
- ~~shape of the cursor~~, ~~I-beam, crosshair~~ — *attributes of Cursor*
- ~~corner~~ — *attribute of Rectangle*
- ~~time~~ — *an implicit attribute of the system*

Many nouns will just refer to attributes of certain entities. Be careful to distinguish the role of an attribute from the type of value it is. Simple values do not need to be modeled as separate classes.

The height and width of a rectangle, and the major and minor radius of an ellipse are simple value attributes of rectangles and ellipses. We do not need to model them.

A corner of a Rectangle is probably a Point. Once we model Rectangle and Point, we do not need to model "corner" as a concept, but only as an attribute of a rectangle.

Candidate Classes

Preliminary analysis yields the following candidates:

Character	Line Element
Control Point	Point
Creation Tool	Rectangle
Current Selection	Rectangle Creation Tool
Drawing	Rectangle Element
Drawing Editor	Selection Tool
Drawing Element	Text Creation Tool
Ellipse Creation Tool	Text Element
Ellipse Element	Tool
Line Creation Tool	

Expect the list to evolve as design progresses.

Roadmap

- > Responsibility-Driven Design
  - —Finding Classes
  - —Class Selection Rationale
  - —**CRC sessions**
  - —Identifying Responsibilities
  - —Finding Collaborations
  - —Structuring Inheritance Hierarchies
- > SOLID object-oriented design principles

CRC Cards

Use CRC cards to record candidate classes:

Text Creation Tool	*subclass of Tool*
Editing Text	Text Element

Record the candidate *Class Name* and *superclass* (if known)

Record each *Responsibility* and the *Collaborating classes*

—compact, easy to manipulate, easy to modify or discard!

—easy to arrange, reorganize

—easy to retrieve discarded classes

CRC (Class-Responsibility-Collaboration) cards are a lightweight tool to keep track of candidate classes. Just use index cards (or cut up regular A4 sheets into 1/4 pieces).

On each card write the name of the candidate class at the top. For each class, list the high-level responsibilities in a column (there should not be too many of them). For each responsibility, list any collaborating classes to the right.

You will then use the cards to elaborate and validate your design.

https://en.wikipedia.org/wiki/Class-responsibility-collaboration_card

NB: You don't have to use index cards, but they have the advantage of being easy to add, change, move around and throw away. You don't need a high-tech tool.

CRC Sessions

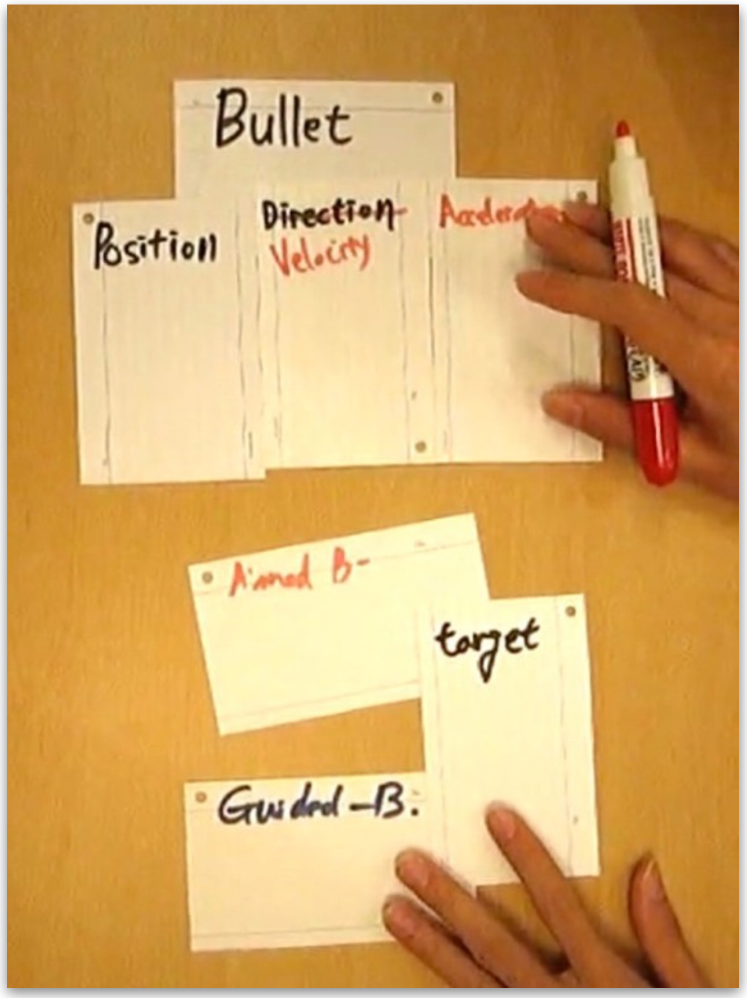
CRC cards are *not* a specification of a design.

They are a tool to *explore* possible designs.

- Prepare a CRC card for *each candidate class*
- Get a team of developers to *sit around a table* and distribute the cards to the team
- The team *walks through scenarios*, playing the roles of the classes.

This exercise will uncover:

- *unneeded* classes and responsibilities
- *missing* classes and responsibilities



CRC cards can be used to explore possible design in “CRC sessions”. The idea is to play a game in which team members each take a few of the cards, and then try to play through a concrete scenario (such as “schedule a committee meeting”).

At each step of the scenario, some object must assume responsibility for the task at hand. The game will expose missing responsibilities, unneeded responsibilities, missing classes, missing collaborations.

Roadmap

> Responsibility-Driven Design
  - Finding Classes
  - Class Selection Rationale
  - CRC sessions
  - **Identifying Responsibilities**
  - Finding Collaborations
  - Structuring Inheritance Hierarchies

> SOLID object-oriented design principles

Responsibilities

What are responsibilities?

—the *knowledge* an object maintains and provides

—the *actions* it can perform

Responsibilities represent the *public services* an object may provide to clients (but not the way in which those services may be implemented)

—specify *what* an object does, not *how* it does it

—don’t describe the interface yet, only *conceptual responsibilities*

Responsibilities are high-level specifications of what instances of a class know and do. These will later translate to state (instance variables) and behavior (methods), but at this stage they should just be specified in as general way as possible.

Examples: “keep track of committee members”, “schedule meetings”

Identifying Responsibilities

> Study the requirements specification:
  - highlight *verbs* and determine which represent responsibilities
  - perform a *walk-through* of the system
    - *explore as many scenarios as possible*
    - *identify actions resulting from input to the system*

> Study the candidate classes:
  - class names ⇒ roles ⇒ responsibilities
  - recorded purposes on class cards ⇒ responsibilities

Just as we identified candidate classes by highlighting nouns and noun phrases, we can identify potential responsibilities by highlighting verbs and verb phrases. Any *action* appearing in the requirements document must be the responsibility of some person or thing. If it is part of the system, figure out what class should be responsible for it.

Playing through CRC sessions can be helpful for assigning responsibilities. New responsibilities should be assigned to existing classes that have the needed knowledge.

(If a Committee class is responsible for keeping track of committee members, then adding a new member should also be the responsibility of that class.)

Some further hints follow …

How to assign responsibilities?

Assigning Responsibilities: Be lazy

“Don’t do anything you can push off to someone else.”

(Pelrine)

If you have a job to do but don't have all the needed resources yourself, figure out who does and pass the job on.

This is the object-oriented way of distributing responsibilities.

In a centralised design it is more common to gather all the needed resources and then handle tasks locally. This leads to a lot of needless interaction and a fragile design. The object-oriented way leads to a more robust and distributed design.

Assigning Responsibilities: Be tough

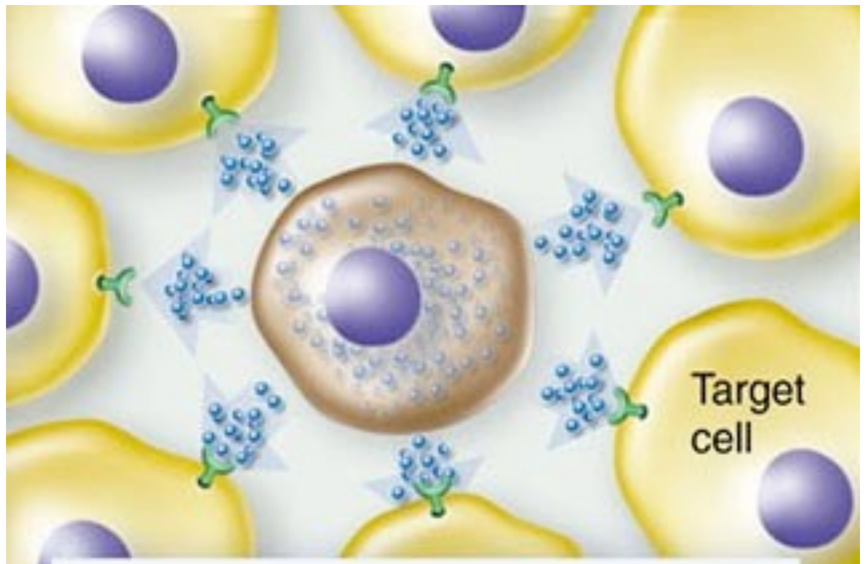
“Don't let anyone else play with your toys”.

(Pelrine)

Why not?

If you let others play with your data, then you suddenly cannot change since they rely on things that are not in your public interface.

Assigning Responsibilities: Be socialist



Paracrine signaling: Cells release signals that affect nearby target cells.

***Evenly distribute* system intelligence**

Avoid procedural centralization of responsibilities

One example: it is well known that in Smalltalk, any method that is longer than 10 lines of code is too long and should be split. Why? Because like this the code is self-explaining with method names explaining what the code does.

Assigning Responsibilities: Be general

> **State responsibilities as *generally* as possible**

— "draw yourself" vs. "draw a line/rectangle etc."

— leads to sharing, reuse, and extensibility

Assigning Responsibilities: Be organized

> **Keep *behaviour* together with any *related information***
>
> — principle of encapsulation

> **Keep *information* about one thing in *one place***
>
> — if multiple objects need access to the same information
>
> - *a new object may be introduced to manage the information, or*
> - *one object may be an obvious candidate, or*
> - *the multiple objects may need to be collapsed into a single one*

> ***Share* responsibilities among related objects**
>
> — break down complex responsibilities

Roadmap

- Responsibility-Driven Design
  - Finding Classes
  - Class Selection Rationale
  - CRC sessions
  - Identifying Responsibilities
  - **Finding Collaborations**
  - Structuring Inheritance Hierarchies
- SOLID object-oriented design principles

Relationships Between Classes

> Drawing Element ***has-knowledge-of*** Control Points

> Drawing element ***is-part-of*** Drawing

> Rectangle Tool ***is-kind-of*** Creation Tool

These are the three key kinds of relationships between classes (more on this in the static UML lecture).

One class can have a *reference* to another. Instances of these classes exist independently. The reference relation can change over time. A `Person` can be the `Chair` of different `Committees` over time.

A class can be *part of* another. The containment relationship is generally fixed, and the part cannot exist without the whole. A meeting `Schedule` contains consists of several possible meeting `Dates`.

A class can be a *specialization* of (kind-of) another class. A `CommitteeMember` is a `Person`.

These relationships have implications for assigning responsibilities …

Relationships Between Classes

> The “Is-Part-Of” Relationship:
    —*distinguish* (don’t share) responsibilities of *part* and of *whole*

Very often the whole will delegate responsibilities to its parts. A `TestSuite` delegates to its individual `Test` instances. A `Committee` may delegate certain tasks to its `Chair`.

Relationships Between Classes

> The “Is-Kind-Of” Relationship:
- —classes sharing a *common attribute* often share a *common superclass*
- —common superclasses suggest *common responsibilities*

e.g., to create a new **Drawing Element**, a Creation Tool must:

1. *accept user input — implemented in subclass*
2. *determine location to place it — generic*
3. *instantiate the element – implemented in subclass*

A specialization shares (inherits) responsibilities of its parent. Subclasses typically specialize or add to existing responsibilities of their parents.

Relationships Between Classes ...

> The “Is-Analogous-To” Relationship:
  —*similarities* between classes suggest as-yet-undiscovered superclasses

Difficulties in assigning responsibilities suggest:

—*missing classes* in design, or — e.g., Group Element

—*free choice* between multiple classes

Collaborations

What are collaborations?

> *collaborations* are *client requests* to servers needed to fulfill responsibilities

> collaborations reveal *control and information flow* and, ultimately, subsystems

> collaborations can uncover *missing responsibilities*

> analysis of communication patterns can reveal *misassigned* responsibilities

Finding Collaborations

For each responsibility:

1. Can the class *fulfill* the responsibility *by itself?*
2. If not, *what does it need*, and from what other class can it obtain what it needs?

For each class:

1. What does this class *know*?
2. What *other classes* need its information or results? Check for collaborations.
3. Classes that *do not interact* with others should be *discarded*. (Check carefully!)

Listing Collaborations

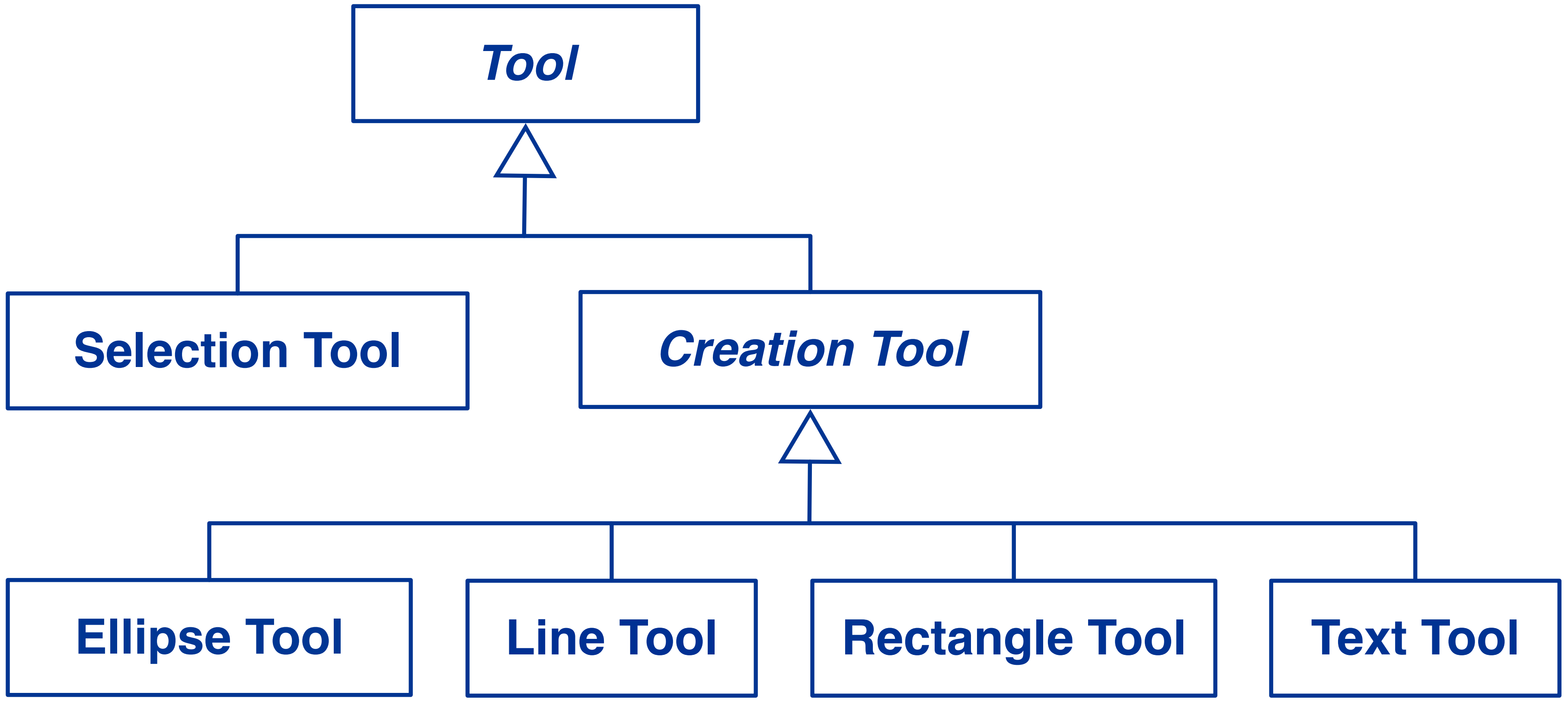
Drawing	
Knows which elements it contains	
Maintains order of elements	Drawing Element

Roadmap

- > Responsibility-Driven Design
  - —Finding Classes
  - —Class Selection Rationale
  - —CRC sessions
  - —Identifying Responsibilities
  - —Finding Collaborations
  - —**Structuring Inheritance Hierarchies**
- > SOLID object-oriented design principles

Finding Abstract Classes

Abstract classes factor out common behaviour shared by other classes



> group related classes with common attributes

> introduce abstract superclasses to represent the group

> “categories” are good candidates for abstract classes

Warning: beware of premature classification; your hierarchy will evolve!

An abstract class is a class that has no instances of its own. It serves only as a template for its subclasses by defining shared responsibilities.

You can easily identify abstract classes by looking for groups of related classes with similar responsibilities. Introduce an abstract class to represent the group.

You may find you need multiple abstract classes if there are subgroups with similar responsibilities, as in the example.

Sharing Responsibilities

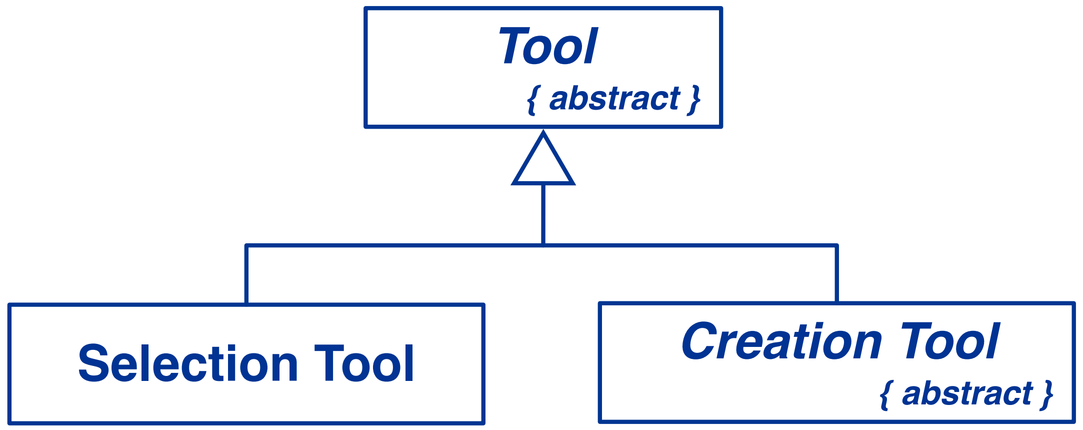
Concrete classes may be both instantiated and inherited from.

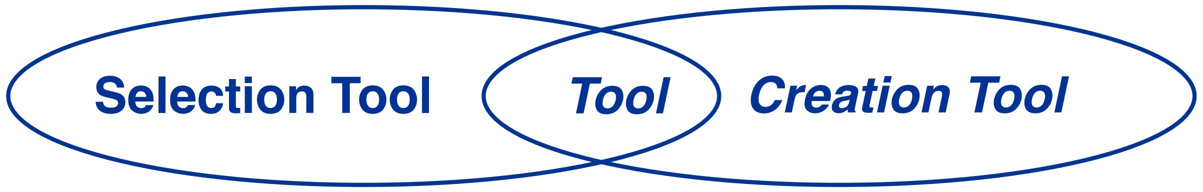
Abstract classes may only be inherited from.

Note on class cards and on class diagram.

Venn Diagrams can be used to visualize shared responsibilities.

(Warning: not part of UML!)



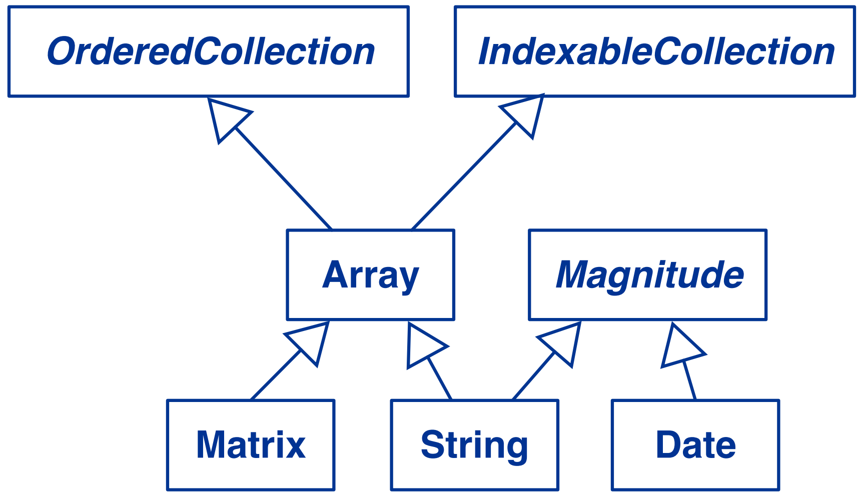


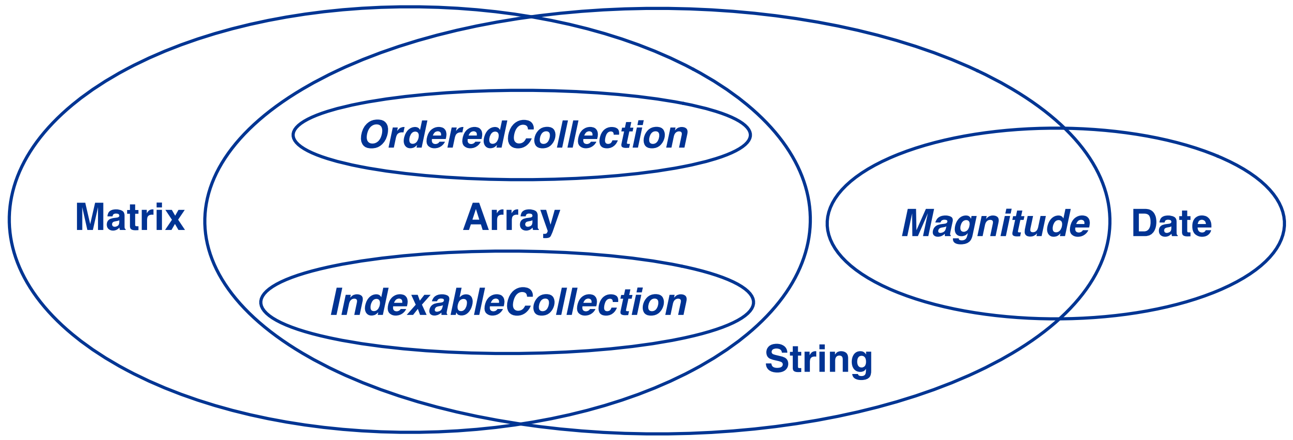
Multiple Inheritance

Decide whether a class will be *instantiated* to determine if it is *abstract or concrete*.

Responsibilities of subclasses are *larger* than those of *superclasses*.

Intersections represent *common superclasses.*





Some programming languages (like C++ and Python) support multiple inheritance and others do not (like Java and Smalltalk). In any case multiple inheritance may be useful in your design, although you may have to translate it in your implementation. In Java, for example, you can use interfaces to partly compensate for the lack of multiple inheritance.

Building Good Hierarchies

Model a “kind-of” hierarchy:

> Subclasses should *support all inherited responsibilities*, and possibly more

Factor common responsibilities as high as possible:

> Classes that *share common responsibilities* should *inherit from a common abstract superclass*; introduce any that are missing

Building Good Hierarchies ...

Abstract classes do not inherit from concrete classes:

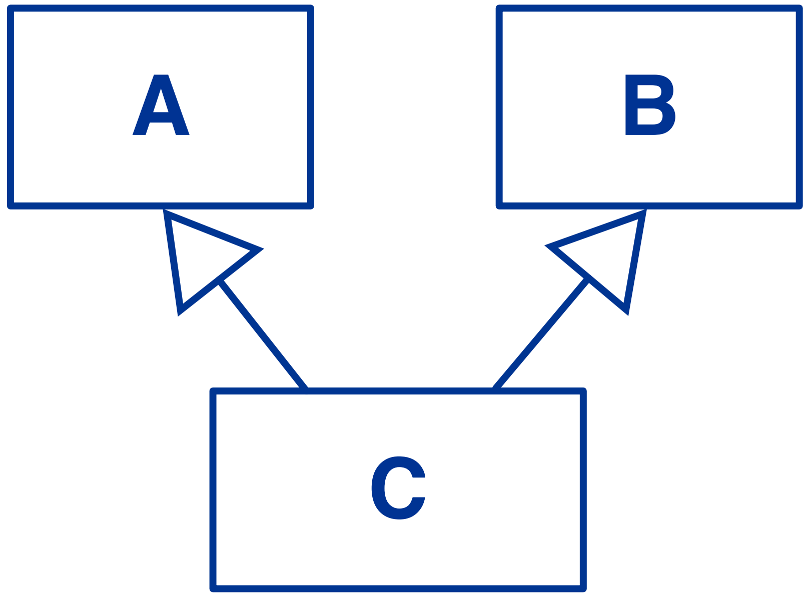
> Eliminate by introducing *common abstract superclass*: abstract classes should support responsibilities in an implementation-independent way

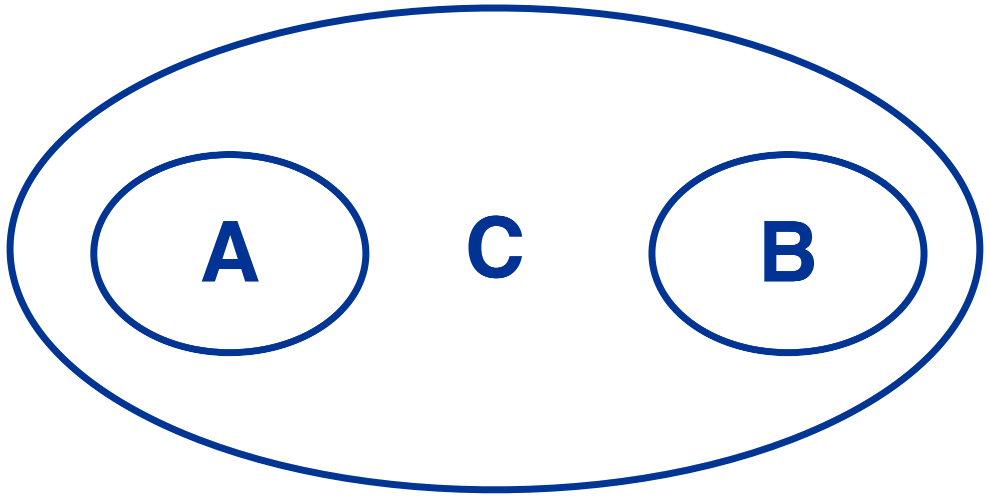
Eliminate classes that do not add functionality:

> Classes should either add new responsibilities, or a particular way of implementing inherited ones

Building Kind-Of Hierarchies

Correctly Formed Subclass Responsibilities:



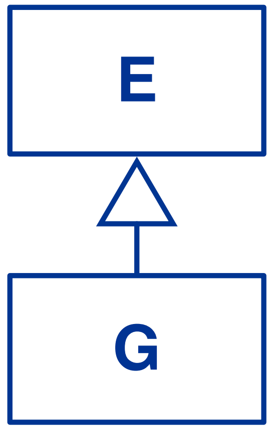


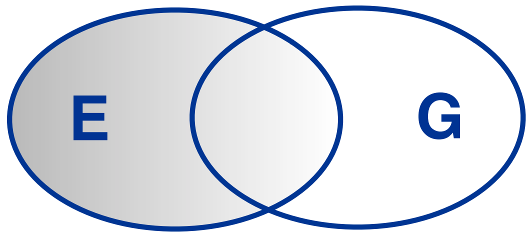
C assumes *all* the responsibilities of both A and B

Building Kind-Of Hierarchies ...

Incorrect Subclass/Superclass Relationships

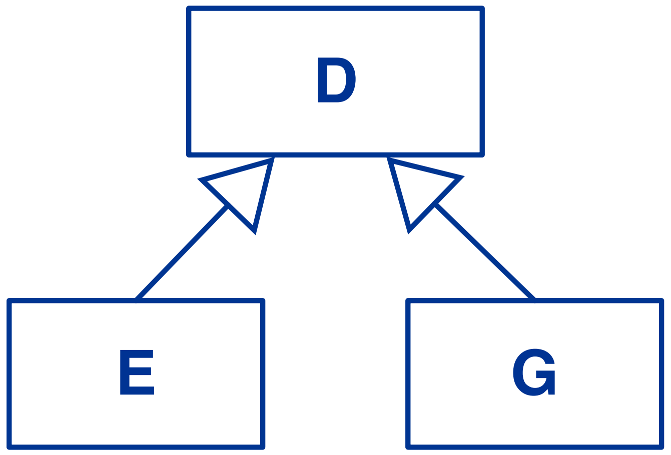
> G assumes only *some* of the responsibilities inherited from E

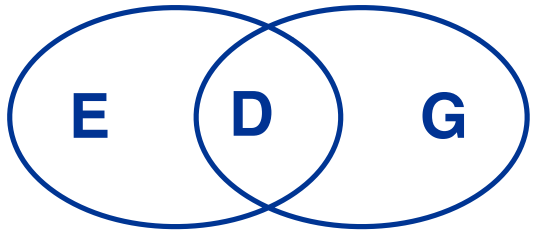




Revised Inheritance Relationships

> Introduce *abstract superclasses* to encapsulate common responsibilities

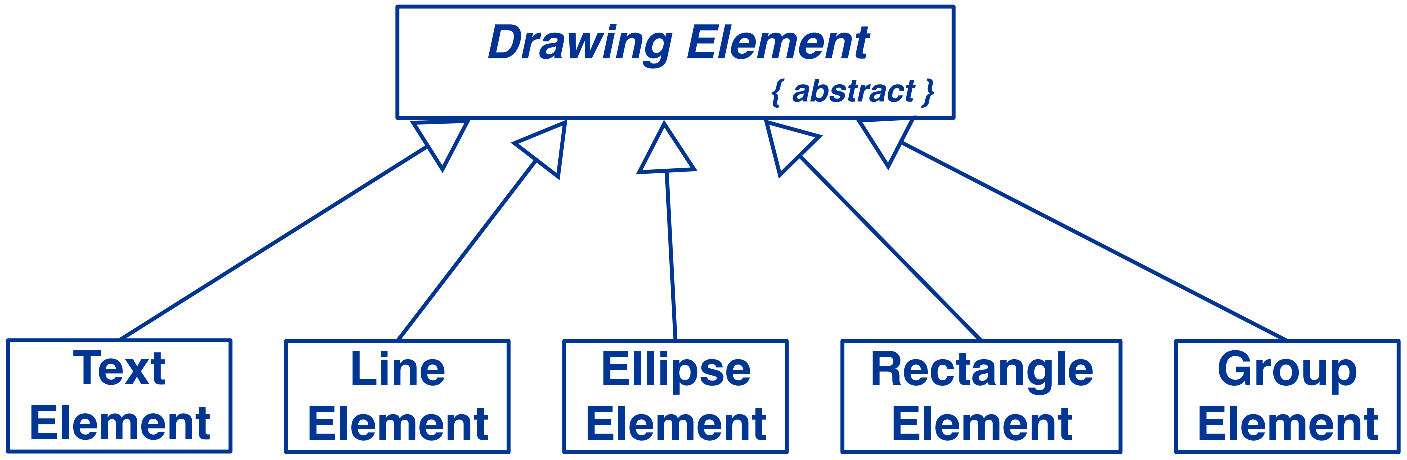


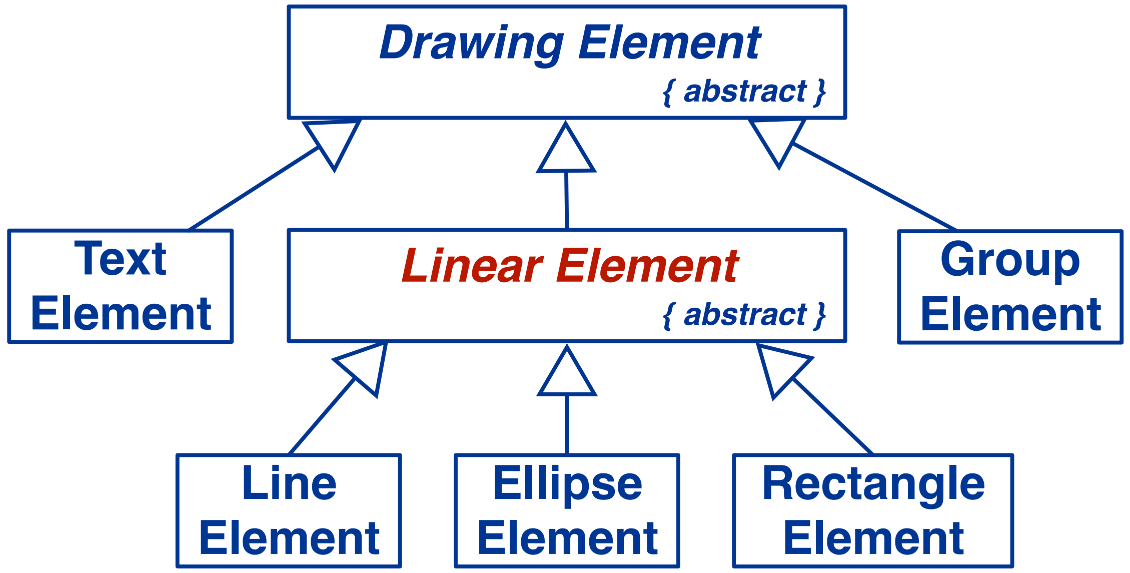


Refactoring Responsibilities

Lines, Ellipses and Rectangles are responsible for keeping track of the width and colour of the lines they are drawn with.

This suggests a *common superclass.*





Roadmap

- Responsibility-Driven Design
  - —Finding Classes
  - —Class Selection Rationale
  - —CRC sessions
  - —Identifying Responsibilities
  - —Finding Collaborations
  - —Structuring Inheritance Hierarchies
- **SOLID object-oriented design principles**

SOLID (object-oriented design principles)

- **S**ingle responsibility
- **O**pen-closed
- **L**iskov substitution
- **I**nterface segregation
- **D**ependency inversion

Concerns: rigidity, fragility, immobility, viscosity (!)

Robert C. Martin. *Design Principles and Design Patterns*. 2000.

"Uncle Bob" (Robert) Martin has written extensively on OO design principles. The principles here did not originate with him, but he has done a good job of popularizing them in terms of the following concerns:

- "rigidity": hard to change
- "fragility": breaks when changed
- "immobility": impossible to reuse (too many dependencies)
- "viscosity": changes break design

Robert C. Martin. *Design Principles and Design Patterns*. 2000.

https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

http://scgresources.unibe.ch/Literature/ESE/Mart00b-principles_and_patterns.pdf

Single responsibility principle

Every class should have a *single responsibility*

There should never be more than one reason for a class to change

If a class has multiple responsibilities, then they become coupled. A change to one responsibility will then impact another.

Difficulty: what is the granularity of a “responsibility”?

Martin equates this principle with “cohesion” — a cohesive class is one that has a single responsibility.

- http://scgresources.unibe.ch/Literature/ESE/SRP.pdf
- https://en.wikipedia.org/wiki/Single_responsibility_principle

Open/closed principle

Software entities should be *open for extension*, but *closed for modification*.

"In other words, we want to be able to change what the modules do, without changing the source code of the modules."

Bertrand Meyer, *Object-Oriented Software Construction*, 1988.

This principle was first formulated by Bertrand Meyer. The idea is that classes should be closed for instances (do not violate encapsulation), but open for subclasses (inherit, reuse and extend).

http://scgresources.unibe.ch/Literature/ESE/OCP.pdf

https://en.wikipedia.org/wiki/Open/closed_principle

Liskov substitution principle

(Instances of) subclasses should be *substitutable* for (instances of) their base classes.

Restated in terms of *contracts*, a derived class is substitutable for its base class if:

- *Its preconditions are no stronger than the base class method.*
- *Its postconditions are no weaker than the base class method.*

The basic idea of "LSP" is straightforward and appealing. If your code expects an object of a given type (or class), then it should still work if you substitute an instance of a subtype (subclass).

Liskov explains substitutability in terms of "contracts", as shown in the slide. Example: Is a `Circle` an `Ellipse`? Depends on the contract clients expect! (Ditto for Square and Rectangle.)

Barbara Liskov and Jeannette Wing. *A behavioral notion of subtyping*. ACM TOPLAS, 1994.

http://scgresources.unibe.ch/Literature/ESE/Lisk94a-TOPLAS94.pdf

https://en.wikipedia.org/wiki/Liskov_substitution_principle

Peter Wegner, Stanley Zdonik. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like*. ECOOP 1988.

http://scgresources.unibe.ch/Literature/ESE/Wegn88a-ECOOP88.pdf

Interface segregation principle

Many *client-specific interfaces* are better than one general purpose interface.

Clients should not be forced to depend upon interfaces that they don't use.

The point is to reduce coupling, and thus to avoid rippling changes to many clients when interfaces inevitably change.

The problem can be solved either by allowing classes to implement multiple interfaces (as in Java), or by delegation (interposing an Adapter).

Robert C. Martin, *The Interface Segregation Principle*, C++ Report, June 1996.

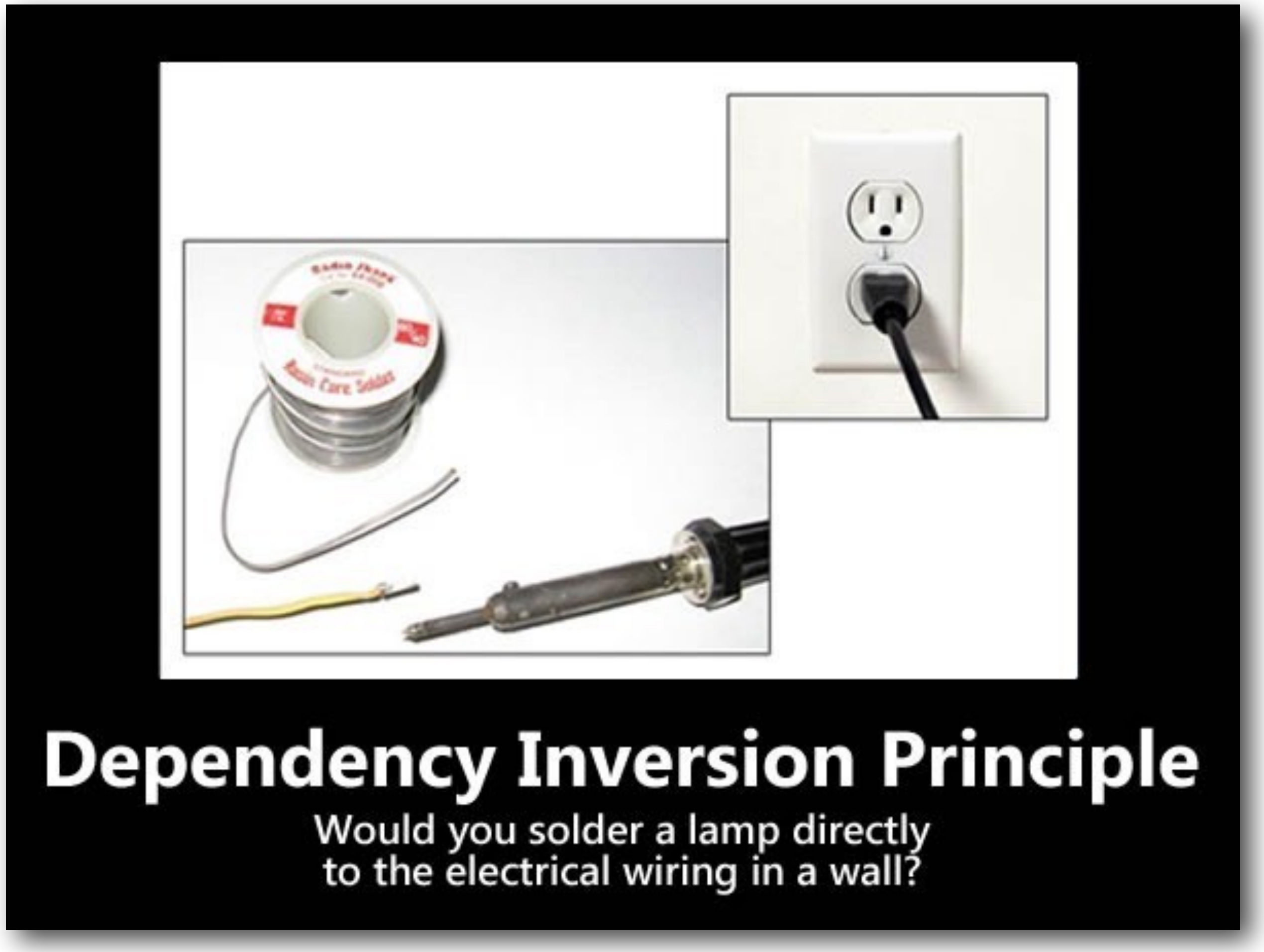
http://scgresources.unibe.ch/Literature/ESE/ISP.pdf

https://en.wikipedia.org/wiki/Interface_segregation_principle

Dependency inversion principle

Depend upon abstractions.
Do not depend upon concretions.

High-level modules should not depend on low-level modules. Both should depend on abstractions (i.e., *interfaces*). Abstractions should not depend upon details. Details should depend upon abstractions.



Dependency Inversion Principle
Would you solder a lamp directly to the electrical wiring in a wall?

Avoid referring to concrete classes in your code. Decouple high-level code from low-level code so details can change!

This can be solved by interfaces, subclassing, plugins, code generation, dependency injection ...

Problem: you need a concrete class when you create instances.

Solution: use an Abstract Factory!

Robert C. Martin, The Dependency Inversion Principle. C++ Report, May 1996.

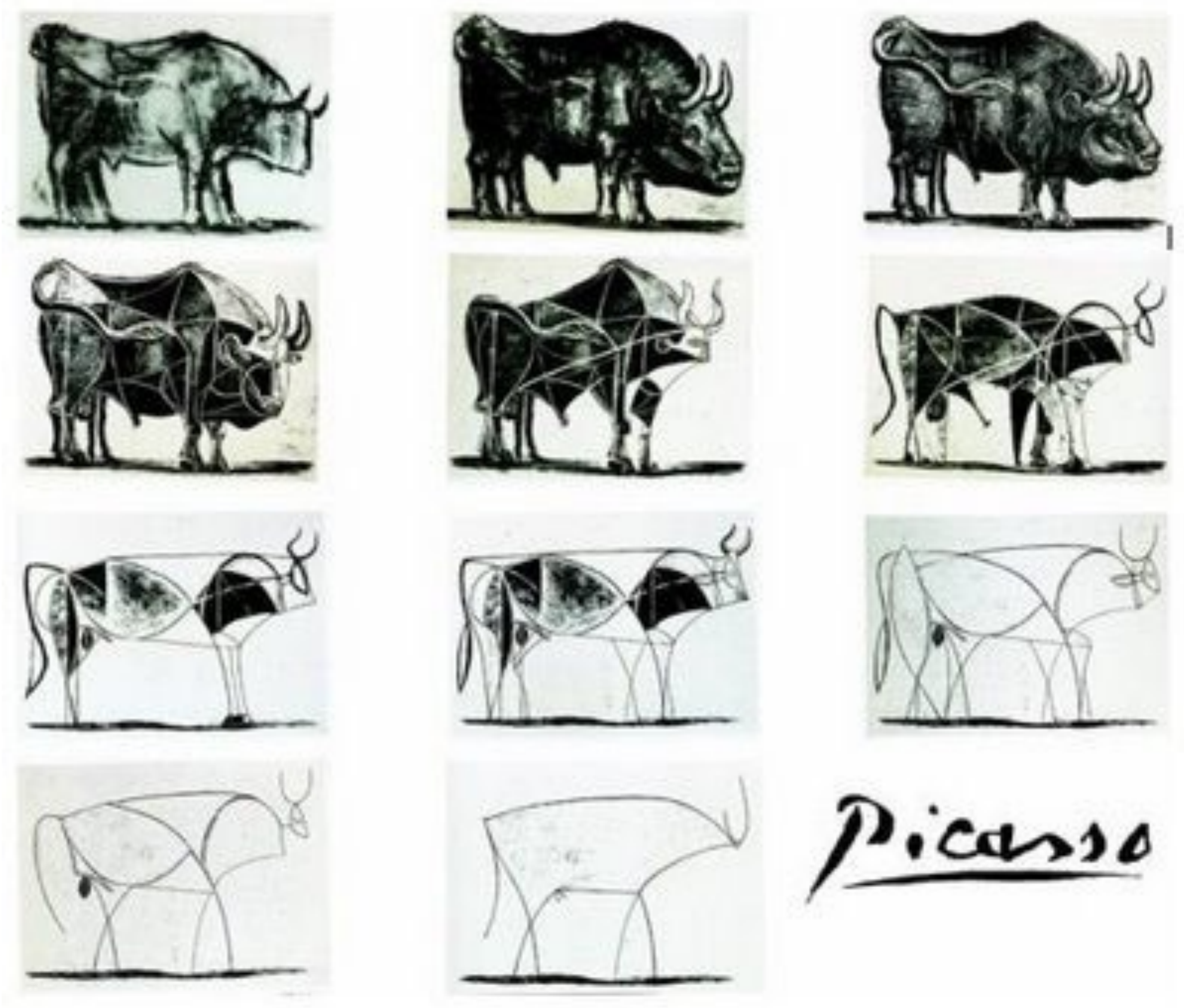
http://scgresources.unibe.ch/Literature/ESE/DIP.pdf

https://en.wikipedia.org/wiki/Dependency_inversion_principle

“Meme” from this blog post:

https://blogs.msdn.microsoft.com/cdndevs/2009/07/15/the-solid-principles-explained-with-motivational-posters/

Design is iterative



“There is no great writing, only great rewriting.”
— *Louis Brandeis*

What you should know!

- What criteria can you use to identify potential classes?
- How can CRC cards help during analysis and design?
- How can you identify abstract classes?
- What are class responsibilities, and how can you identify them?
- How can identification of responsibilities help in identifying classes?
- What are collaborations, and how do they relate to responsibilities?
- How can you identify abstract classes?
- What criteria can you use to design a good class hierarchy?
- How can refactoring responsibilities help to improve a class hierarchy?

Can you answer the following questions?

- When should an attribute be promoted to a class?
- Why is it useful to organize classes into a hierarchy?
- How can you tell if you have captured all the responsibilities and collaborations?
- What use is multiple inheritance during design if your programming language does not support it?