



SE 4352

Software Architecture and Design

Fall 2018

Module 8

The Process of Design

- Definition:

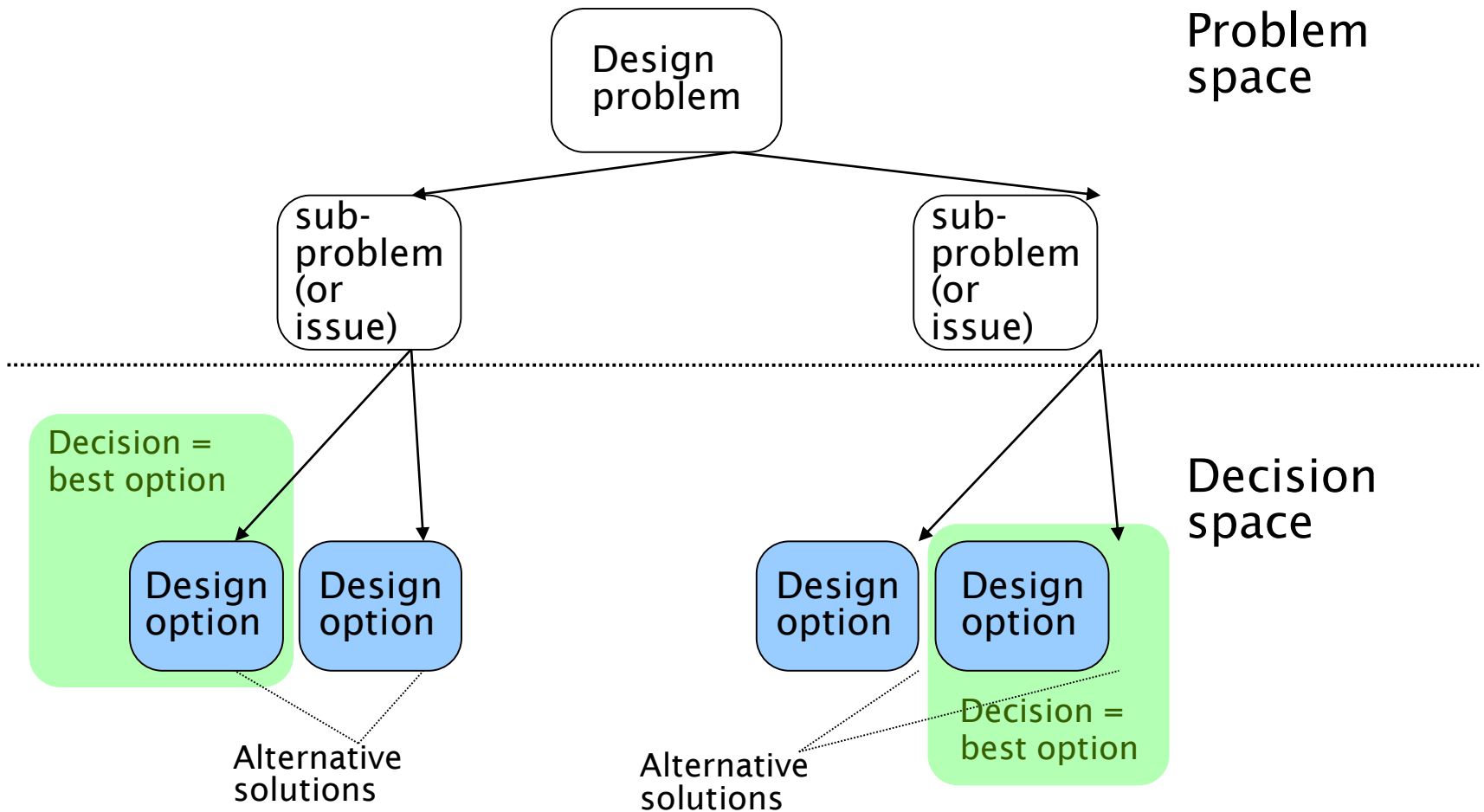
- *Design* is a problem-solving process whose objective is to find and describe a way:

- To implement the system's *requirements*...
 - Within the *architectural structure*....
 - While *respecting the constraints imposed*
 - including the budget
 - And while adhering to general principles of *good quality*

Design as a series of decisions

- A designer is faced with a series of *design issues*
 - These are sub-problems of the overall design problem.
 - Each issue normally has several alternative solutions:
 - *design options*.
 - The designer makes a *design decision* to resolve each issue.
 - This process involves choosing the best option from among the alternatives.

Making decisions





Making decisions

- To make each design decision, the software engineer uses:
 - Knowledge of
 - The architecture
 - the requirements
 - the design as created so far
 - the technology available
 - software design principles and ‘best practices’
 - what has worked well in the past

Different aspects of design

- *Architecture design:*
 - The division into subsystems and components,
 - How these will be connected.
 - How they will interact.
 - Their interfaces.
- *Class design:*
 - The various features of classes.
- *User interface design*
- *Algorithm design:*
 - The design of computational mechanisms.
- *Protocol design:*
 - The design of communications protocol.




Principles Leading to Good Design

- Overall *goals* of good design:
 - Increasing profit by reducing cost and increasing revenue
 - Ensuring that we actually conform with the requirements
 - Accelerating development
 - Increasing qualities such as
 - Usability
 - Efficiency
 - Reliability
 - Maintainability
 - Reusability



Design Principle 1: Divide and conquer

- Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things
 - Separate people can work on each part.
 - An individual software engineer can specialize.
 - Each individual component is smaller, and therefore easier to understand.
 - Parts can be replaced or changed without having to replace or extensively change other parts.



Design Principle 2:

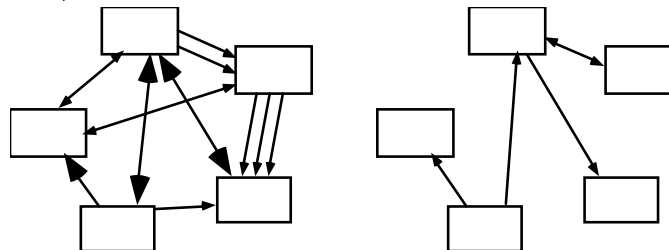
Increase cohesion where possible

- A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things
 - This makes the system as a whole easier to understand and change
 - Type of cohesion:
 - Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

Design Principle 3:


Reduce coupling where possible

- *Coupling* occurs when there are *interdependencies* between one module and another
 - When interdependencies exist, changes in one place will require changes somewhere else.
 - A network of interdependencies makes it hard to see at a glance how some component works.
 - Type of coupling:
 - Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External



Design Principle 4: Keep the level of abstraction as high as possible


- Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity
 - A good abstraction is said to provide *information hiding*
 - Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details



Design Principle 5:

Increase reusability where possible

- Design the various aspects of your system so that they can be used again in other contexts
 - Generalize your design as much as possible
 - Follow the preceding three design principles
 - Design your system to contain hooks
 - Simplify your design as much as possible



Design Principle 6:


Reuse existing designs and code where possible

- Design with reuse is complementary to design for reusability
 - Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
 - *Cloning* should not be seen as a form of reuse



Design Principle 7: Design for flexibility

- Actively anticipate changes that a design may have to undergo in the future, and prepare for them
 - Reduce coupling and increase cohesion
 - Create abstractions
 - Do not hard-code anything
 - Leave all options open
 - Do not restrict the options of people who have to modify the system later
 - Use reusable code and make code reusable



Design Principle 8:

Anticipate obsolescence

- Plan for changes in the technology or environment so the software will continue to run or can be easily changed
 - Avoid using early releases of technology
 - Avoid using software libraries that are specific to particular environments
 - Avoid using undocumented features or little-used features of software libraries
 - Avoid using software or special hardware from companies that are less likely to provide long-term support
 - Use standard languages and technologies that are supported by multiple vendors



Design Principle 9: Design for Portability

- Have the software run on as many platforms as possible
 - Avoid the use of facilities that are specific to one particular environment
 - E.g. a library only available in Microsoft Windows



Design Principle 10: Design for Testability

- Take steps to make testing easier
 - Design a program to automatically test the software
 - Ensure that all the functionality of the code can be driven by an external program, bypassing a graphical user interface
 - In Java, you can create a main() method in each class in order to exercise the other methods



Design Principle 11:

Design defensively

- Never trust how others will try to use a component you are designing
 - Handle all cases where other code might attempt to use your component inappropriately
 - Check that all of the inputs to your component are valid: the *preconditions*
 - Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking



The MVC architecture and design principles

1. *Divide and conquer.*
2. *Increase cohesion:*
3. *Reduce coupling:*
4. *Increase abstraction:*
5. *Increase reusability:*
6. *Increase reuse:*
7. *Increase flexibility:*
8. *Anticipate obsolescence:*
9. *Design for portability.*
10. *Design for testability.*
11. *Design defensively.*

Techniques for making good design decisions

- Using priorities and objectives to decide among alternatives
 - Step 1: List and describe the alternatives for the design decision.
 - Step 2: List the advantages and disadvantages of each alternative with respect to your objectives and priorities.
 - Step 3: Determine whether any of the alternatives prevents you from meeting one or more of the objectives.
 - Step 4: Choose the alternative that helps you to best meet your objectives.
 - Step 5: Adjust priorities for subsequent decision making.



Back to Architectural Patterns

- ...for one of the patterns we skipped
- ...from Chapter 13



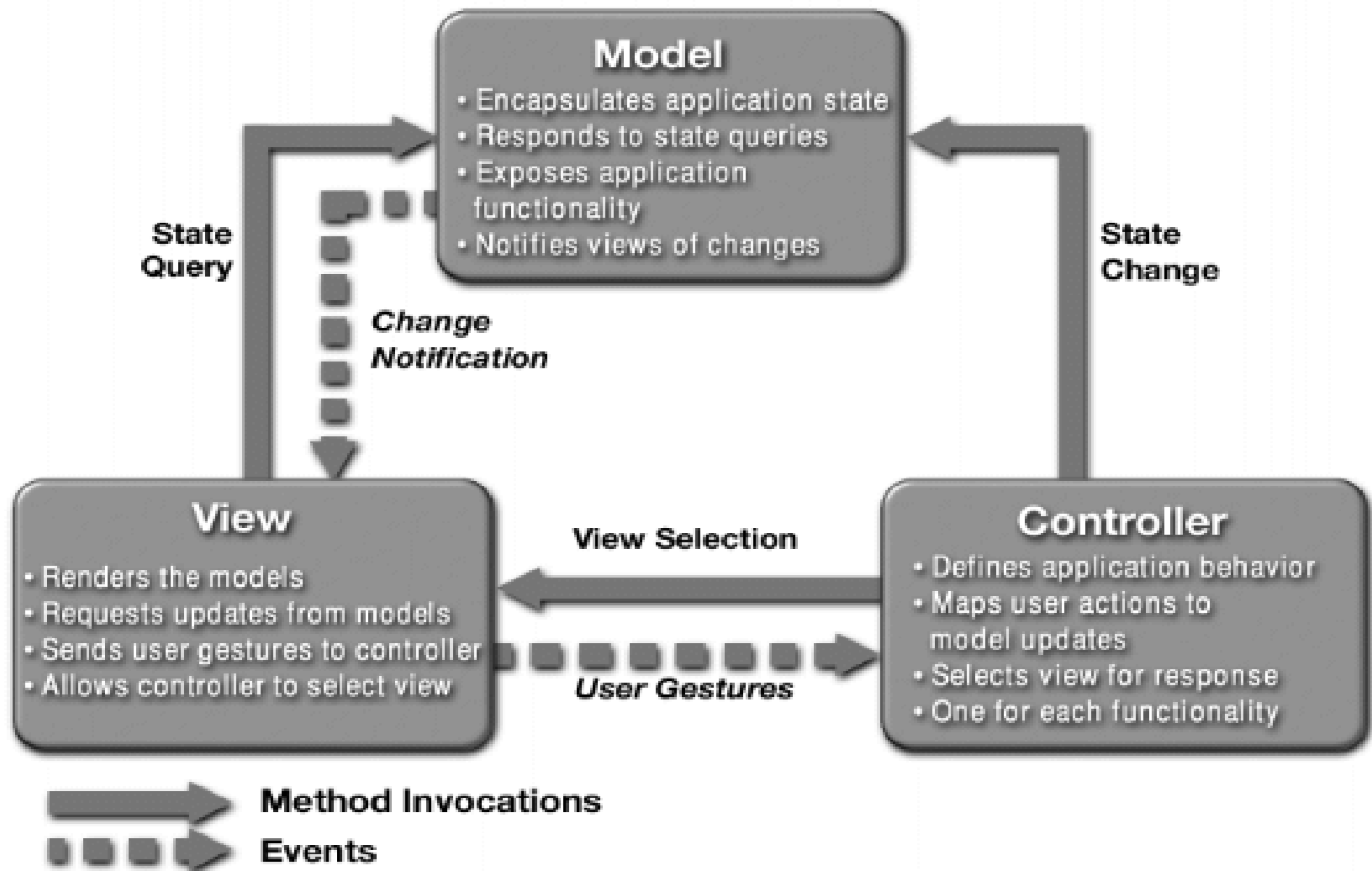
Model-View-Controller Pattern

- **Context:** User interface software is typically the most frequently modified portion of an interactive application. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.
- **Problem:** How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?

Model-View-Controller Pattern

- **Context:** User interface software is typically the most frequently modified portion of an interactive application. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.
- **Problem:** How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?
- **Solution:** The model-view-controller (MVC) pattern separates application functionality into three kinds of components:
 - A model, which contains the application's data
 - A view, which displays some portion of the underlying data and interacts with the user
 - A controller, which mediates between the model and the view and manages the notifications of state changes

MVC Example





MVC Solution - 1

- Overview: The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.
- Elements:
 - The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic.
 - The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both.
 - The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.

MVC Solution - 2

- Relations: The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes.
- Constraints:
 - There must be at least one instance each of model, view, and controller.
 - The model component should not interact directly with the controller.
- Weaknesses:
 - The complexity may not be worth it for simple user interfaces.
 - The model, view, and controller abstractions may not be good fits for some user interface toolkits.



Example of MVC in Web architecture

- The *View* component generates the HTML code to be displayed by the browser.
- The *Controller* is the component that interprets 'HTTP post' transmissions coming back from the browser.
- The *Model* is the underlying system that manages the information.

