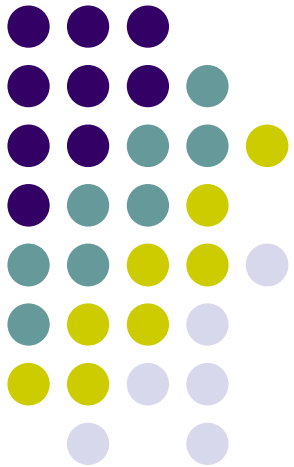
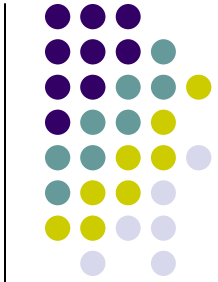
Skills for the Agile Designer

Presented at Software Development 2004

Rebecca Wirfs-Brock
Wirfs-Brock Associates
rebecca@wirfs-brock.com



Agenda



- What is agility?
- Tools for seeing
- Tools for shaping solutions
- Agility and Design Rhythms

Agility Means Finding a Balance

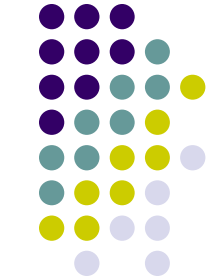
Those who pursue agile development practices, “seek to restore credibility to the concept of methodology. We want to restore a balance. We accept modeling, but not in order to file some diagram in a dusty corporate repository. We accept documentation, but not hundreds of pages of never-maintained and rarely used tomes. We plan, but recognize the limits of planning in a turbulent environment.” —Jim Highsmith

Disturbing Beliefs and Trends

Some mistakenly equate agility with following these practices:

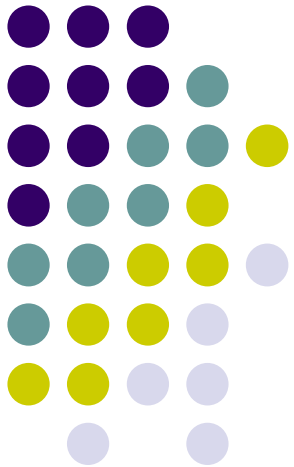
- Problem solving must occur in two week increments.
- Only talking informally about code “smells” instead of having deeper design discussions.
- Equating coding with designing and thinking the only way to improve a design is to refactor the code.
- Doing whatever works without regard to process.
- Equating Formal = Bad & Informal = Good.
- Believing that upfront design and design documentation add no value.

What Fueled these Trends



- Analysis paralysis.
- Endless high-level discussions with little regard for implementation concerns.
- Not appreciating the value of code. Code restructuring *does* add value. Design doesn't stop when coding starts. Cleaning up code helps preserve design integrity.
- Not acknowledging that being formal or precise takes time and isn't always needed.
- Piles of models. No useful results. Many design artifacts that are never read.
- Hope that if extra stuff is omitted, we'll get to market sooner.

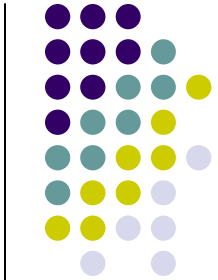
Tools for Seeing



How *Do* You Learn to Be An Agile Designer?

- Practice seeing the nature of the design problem.
- Learn fundamental strategies for producing acceptable solutions.
- Use patterns and good design techniques. Explore and study good designs.
- Examine your results. Maintain a healthy curiosity.

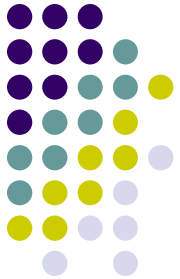
How Responsibility-Driven Designers See



An object supports one or more roles. A class implements one or more roles. Roles can be stereotyped. Stereotypes are purposeful simplifications:

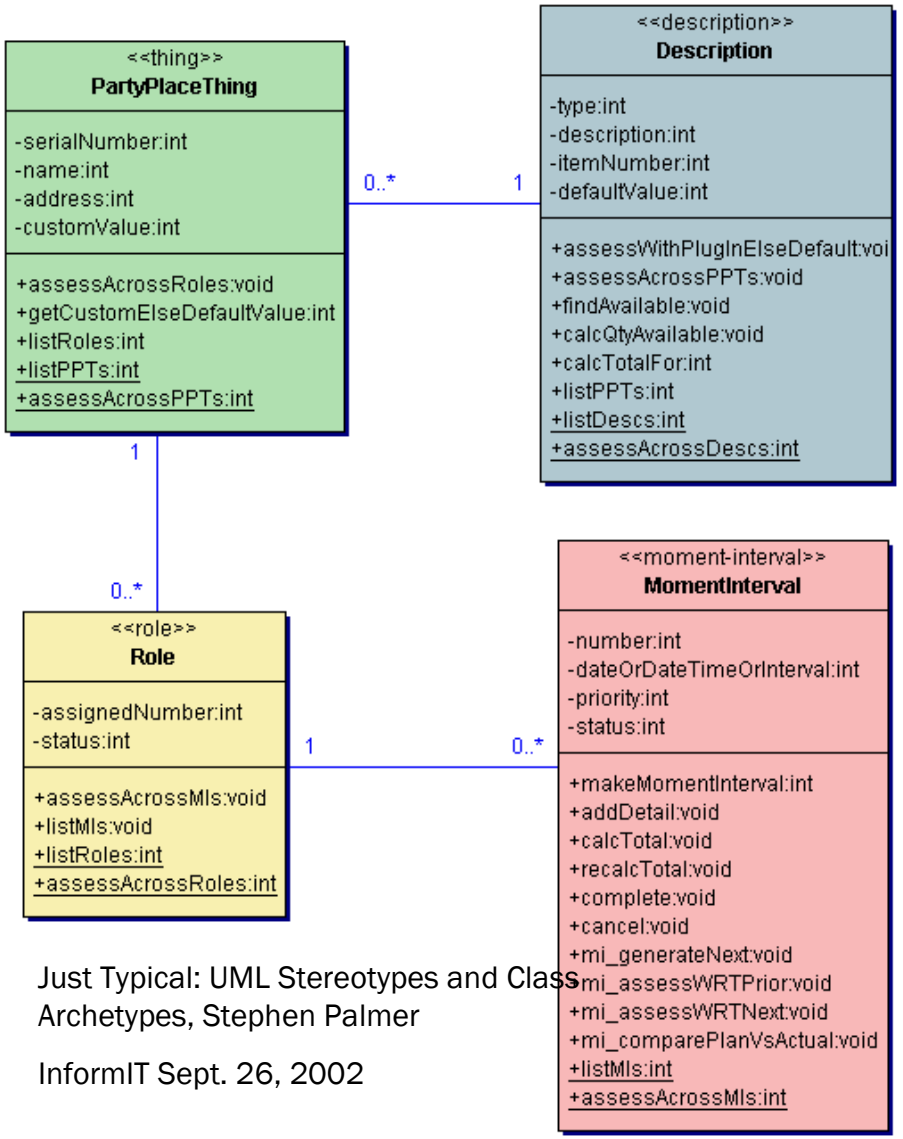
- Information holder - knows and provides information
- Structurer - maintains relationships between objects and information about those relationships
- Service provider - performs work and, in general, offers services
- Coordinator - reacts to events by delegating tasks to others
- Controller - makes decisions and closely directs others' actions
- Interfacer - transforms information and requests between parts of a system

The Generative Power of Role Stereotypes



- Pushing on an object's stereotype(s) leads to initial responsibilities and collaborators.
  - Ask of a service provider, "what requests should it handle?" Turn around and state these as responsibilities for "doing" or "performing" specific services.
  - What duties does an interfacer have for translating information and requests from one part of the system to another (and translating between different parts of an application)?
  - What important events does a controller handle and what other objects does it direct?
- Blending stereotypes makes objects smarter.

How Peter Coad Sees…



Just Typical: UML Stereotypes and Class Archetypes, Stephen Palmer

InformIT Sept. 26, 2002

Pink <<moment-interval>> represents events or activities that we need to keep track of for business or legal reasons

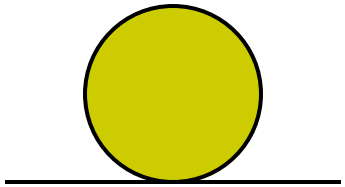
Yellow <<role>> classes represent a way of participating in an event or activity by a green entity

Green entity classes are categorized into <<party>>, <<place>>,and <<thing>>

Blue <<description>> represents a description of some physical thing

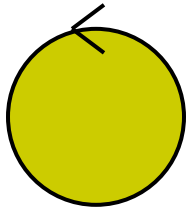
How Ivar Jacobson Classifies

Early analysis stereotypes from Objectory and now part of RUP



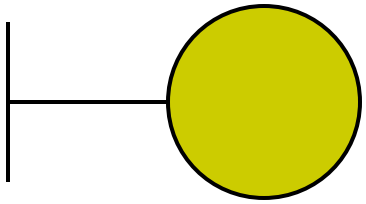
<<entity>>

An entity is passive. It does not initiate any action.



<<control>>

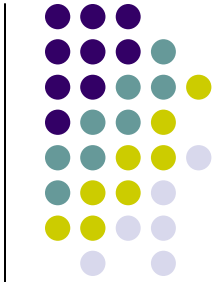
A control object controls interactions between a collection of objects. It usually corresponds to a use case.



<<boundary>>

A boundary object lies on the periphery of the system. It interacts with all 3 kinds of objects.

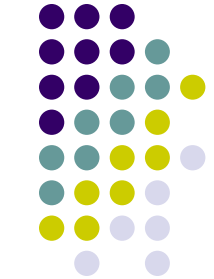
Seeing Should Generate Design Ideas



Classification is useful but not an end unto itself…role stereotypes, “seeing in color”, and finding a role in a pattern help you generate design ideas:

- "Black-and-white conveys basic information. Color reaches out and grabs you.” –Peter Coad
- “…archetype[s] have more or less the same sort of attributes and operations and tend to interact with other archetypes in generally predictable ways. These patterns of characteristics and behavior can help us very … quickly identify attributes and operations… and give us increased confidence in the structure of our code.” –Stephen Palmer

Seeing at Different Abstraction Levels



View objects and behavior at different levels:

- At the *conceptual* level, an object is a set of responsibilities
- At the *specification* level, an object is a set of methods that can be invoked by other objects or itself
- At the *implementation* level, an object is code and data

Pull Up a Level

Reverse engineer a class into responsibilities…

The Java Calendar class

Internally, Calendar keeps track of a point in time in two ways. First, a "raw" value is maintained, which is simply a count of milliseconds since midnight, January 1, 1970 GMT, or, in other words, a Date object. Second, the calendar keeps track of a number of fields, which are the values that are specific to the Calendar type. These are values such as day of the week, day of the month, and month. The raw millisecond value can be calculated from the field values, or vice versa.

Calendar also defines a number of symbolic constants. They represent either fields or values. For example, MONTH is a field constant. It can be passed to get() and set() to retrieve and adjust the month. AUGUST, on the other hand, represents a particular month value. Calling get(Calendar.MONTH) could return Calendar.AUGUST.

Calendar Methods

public int getFirstDayOfWeek()

This method returns the day that is considered the beginning of the week for this Calendar. This value is determined by the Locale of this Calendar. For example, the first day of the week in the United States is Sunday, while in France it is Monday.

public abstract int getGreatestMinimum(int field)

This method returns the highest minimum value for the given time field, if the field has a range of minimum values. If the field does not have a range of minimum values, this method is equivalent to getMinimum().

public abstract int getLeastMaximum(int field)

This method returns the lowest maximum value for the given time field, if the field has a range of maximum values. If the field does not have a range of maximum values, this method is equivalent to getMaximum(). For example, for a GregorianCalendar, the lowest maximum value of DATE_OF_MONTH is 28.

public abstract int getMaximum(int field)

This method returns the maximum value for the given time field. For example, for a GregorianCalendar, the maximum value of DATE_OF_MONTH is 31.

public final void set(int year, int month, int date)

This method sets the values of the year, month, and day-of-the-month fields of this Calendar.

public final void set(int year, int month, int date, int hour, int minute) This method sets the values of the year, month, day-of-the-month, hour, and minute fields of this Calendar.

public final void set(int year, int month, int date, int hour, int minute, int second)

This method sets the values of the year, month, day-of-the-month, hour, minute, and second fields of this Calendar.

public void setFirstDayofWeek(int value)

This method sets the day that is considered the beginning of the week for this Calendar. This value should be determined by the Locale of this Calendar. For example, the first day of the week in the United States is Sunday; in France it's Monday.

public void setLenient(boolean lenient)

This method sets the leniency of this Calendar. A value of false specifies that the Calendar throws exceptions when questionable data is passed to it, while a value of true indicates that the Calendar makes its best guess to interpret questionable data. For example, if the Calendar is being lenient, a date such as March 135, 1997 is interpreted as the 135th day after March 1, 1997.

public void setMinimalDaysInFirstWeek(int value)

This method sets the minimum number of days in the first week of the year. For example, a value of 7 indicates the first week of the year must be a full week, while a value of 1 indicates the first week of the year can contain a single day. This value should be determined by the Locale of this Calendar.

public final void setTime(Date date)

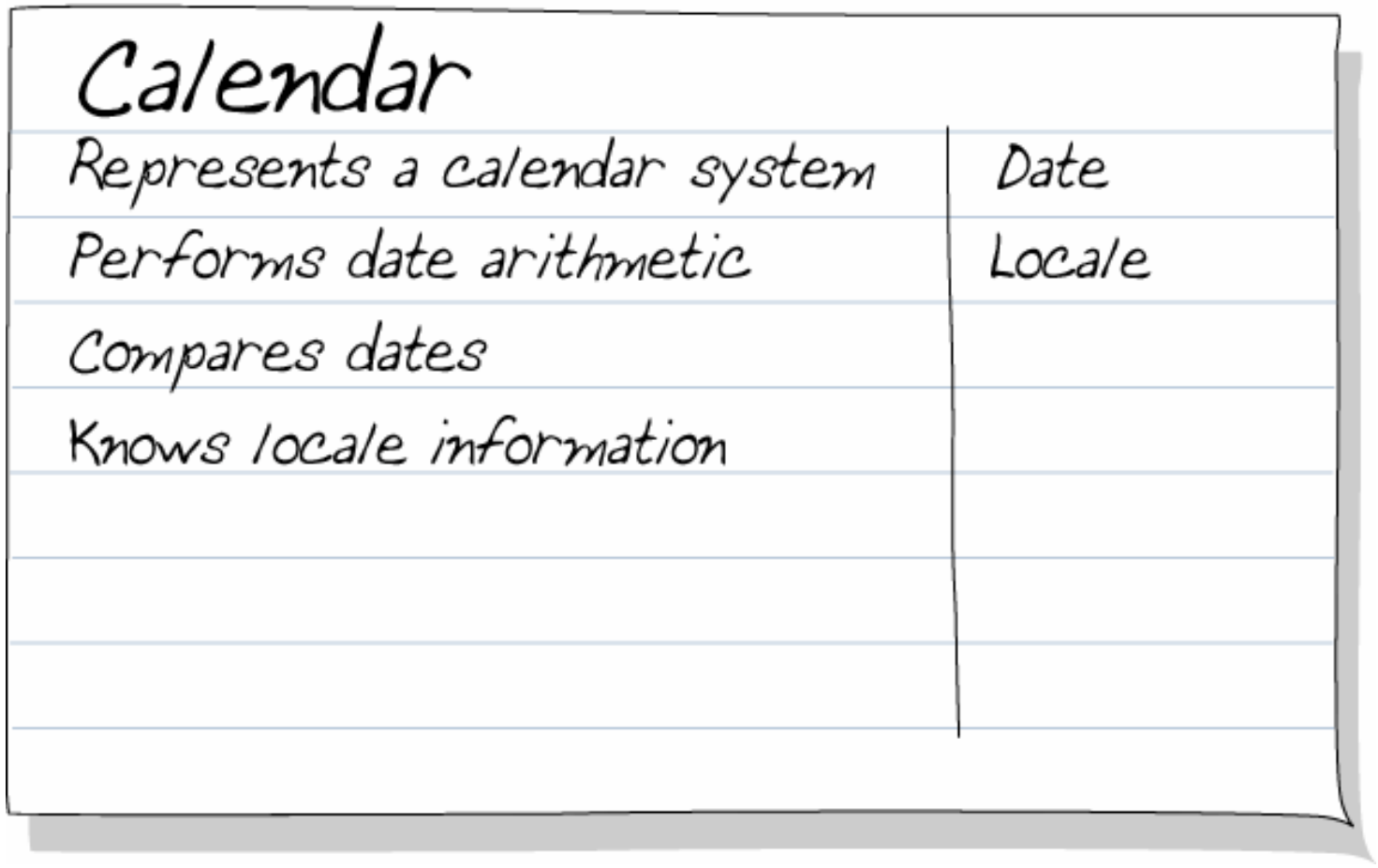
This method sets the point in time that is represented by this

Calendar.

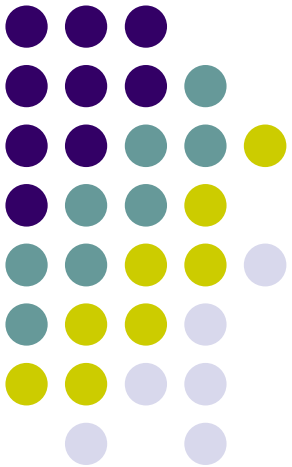
public void setTimeZone(TimeZone value)

This method is used to set the time zone of this Calendar.

To Get The General Picture: Calendar Revealed



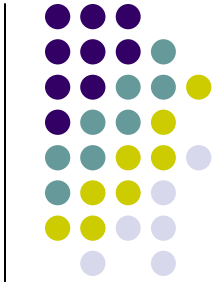
Tools for Shaping Solutions



Traits of An Agile Designer

- Understands the nature of the problem and reflects it in the solution
- Doesn’t fudge on complexity
- Doesn’t blindly apply design patterns as “best solutions”
- Uses patterns as archetypes
- Explores alternatives
- Develops a consistent control style
- Has a sense of aesthetics and compromises them when time gets tight

Frame Your Design Problems



Software systems can be thought of a set of related and interconnected sub-problems—and as a consequence may be comprised of several different problem frames. Each different class of problem has different concerns and design issues.

"When you turn on a light, you probably think of your movement of the control button and the illumination of the light as a single event. In fact, of course, something more complex is going on."

— Michael Jackson

5 Problem Frames

- **Control Problems** - controlling state changes of external devices or machinery.
- **Connection Problems** - receiving or transmitting information indirectly through a connection.
- **Information Display Problems** - presenting information in response to queries about things and events known by your software.
- **Workpiece Problems** - allows users to create and manipulate computer-processable objects or “workpieces.” Just like a lathe is a tool for woodworking, software help users create documents, compile programs, compose music, perform calculations, manipulate images….
- **Transformation Problems** - converting input to one or more output formats.

How a Connection Problem Affects Your Design

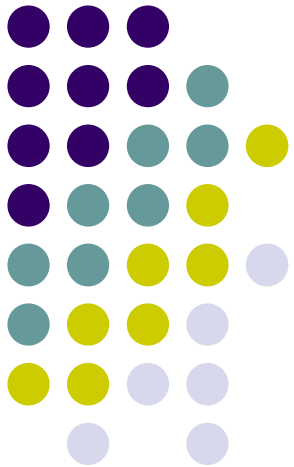
Basic strategies for dealing with connection issues:

- Consider that your software is really interacting with “something in the middle” that is connected to “something out there” that doesn’t always work.
- Design your software to react in the face of potential time-delays, conflicting states between “connected” system as well as faulty connections.

Problem Frames and Design Focus

- The ideal: Jackson advocates fully understanding the nature of the problems your software is trying to solve before you start design.
- The agile reality: The world is full of imperfect knowledge and time constraints. Quickly characterize what problem frames your design must address… realize that more may crop up as you work on your design.

A Case Study: Design for "Build A Message" Use Case

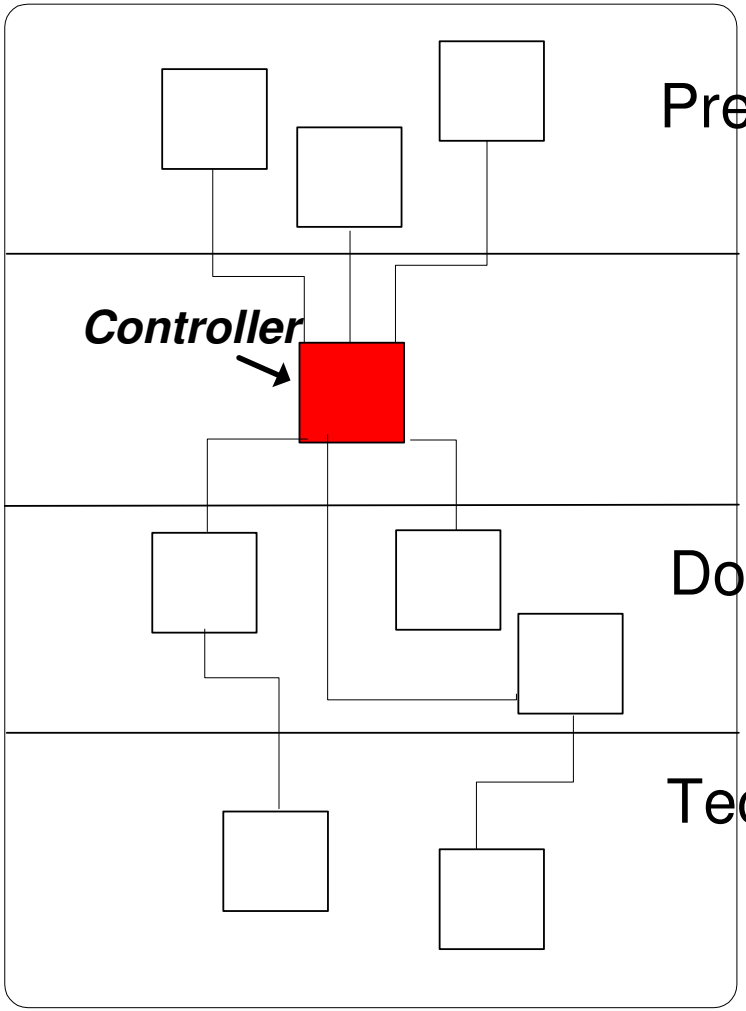


"Build A Message" Use Case

Speak for Me enables a severely disabled user to communicate

Actor Actions	System Responsibilities
"Click" to start software speaking	Start building a message
Repeat until . . .	
Optionally, "click" to select letter	Determine what to speak (letter, word, sentence, or space) Speak letter Add letter to word
Optionally, "click" to select word	Speak space Add word to end of sentence Start new word
Optionally, "click" to select sentence	Speak sentence Add sentence to end of message Start new sentence
... a command is issued	
	Process command (separate use cases)

Centralized Use Case Control



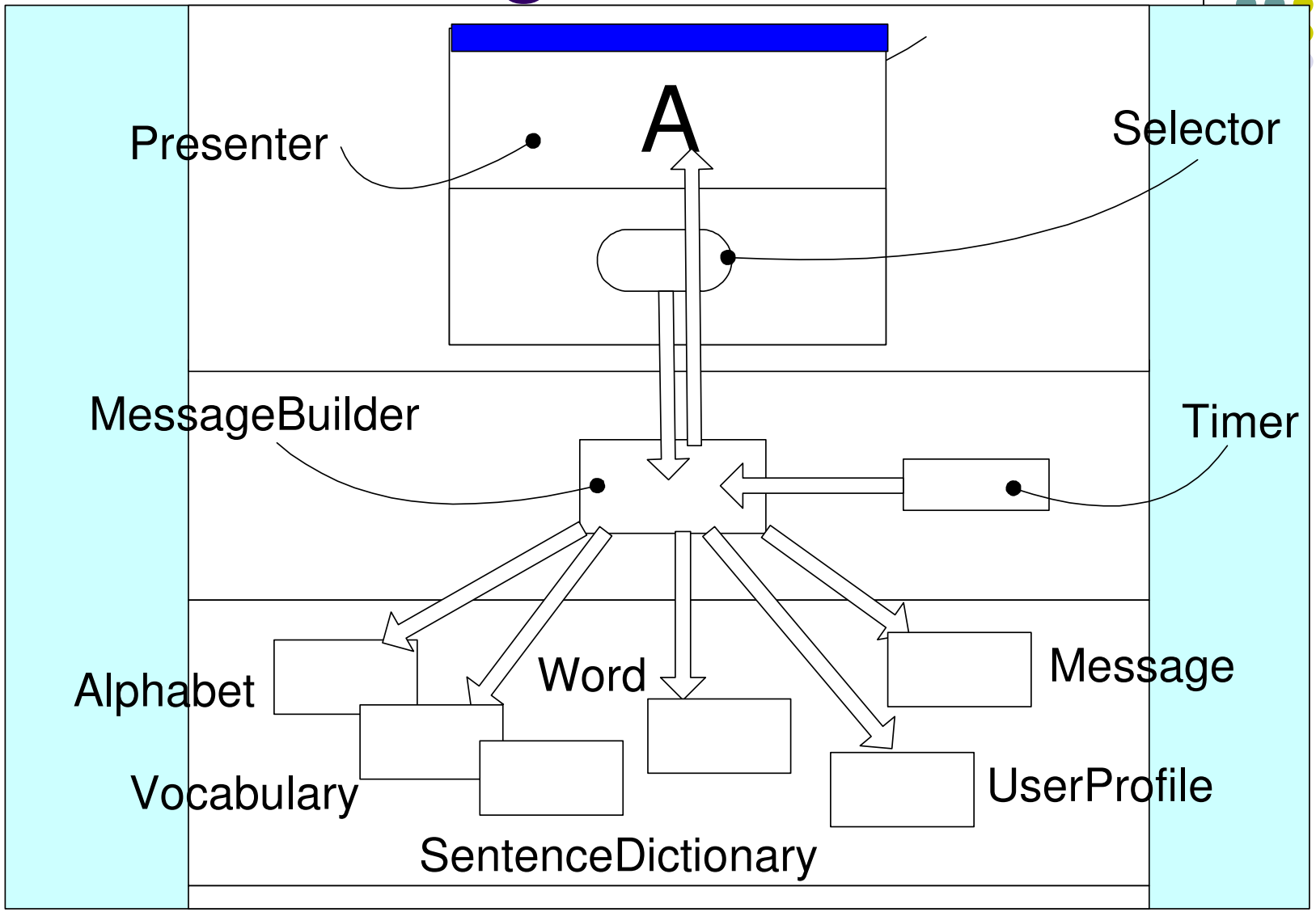
Presentation Layer

Application Layer

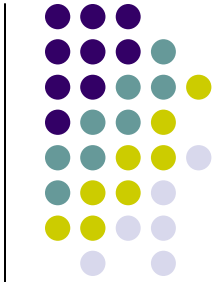
Domain Services Layer

Technical Services Layer

Build a Message

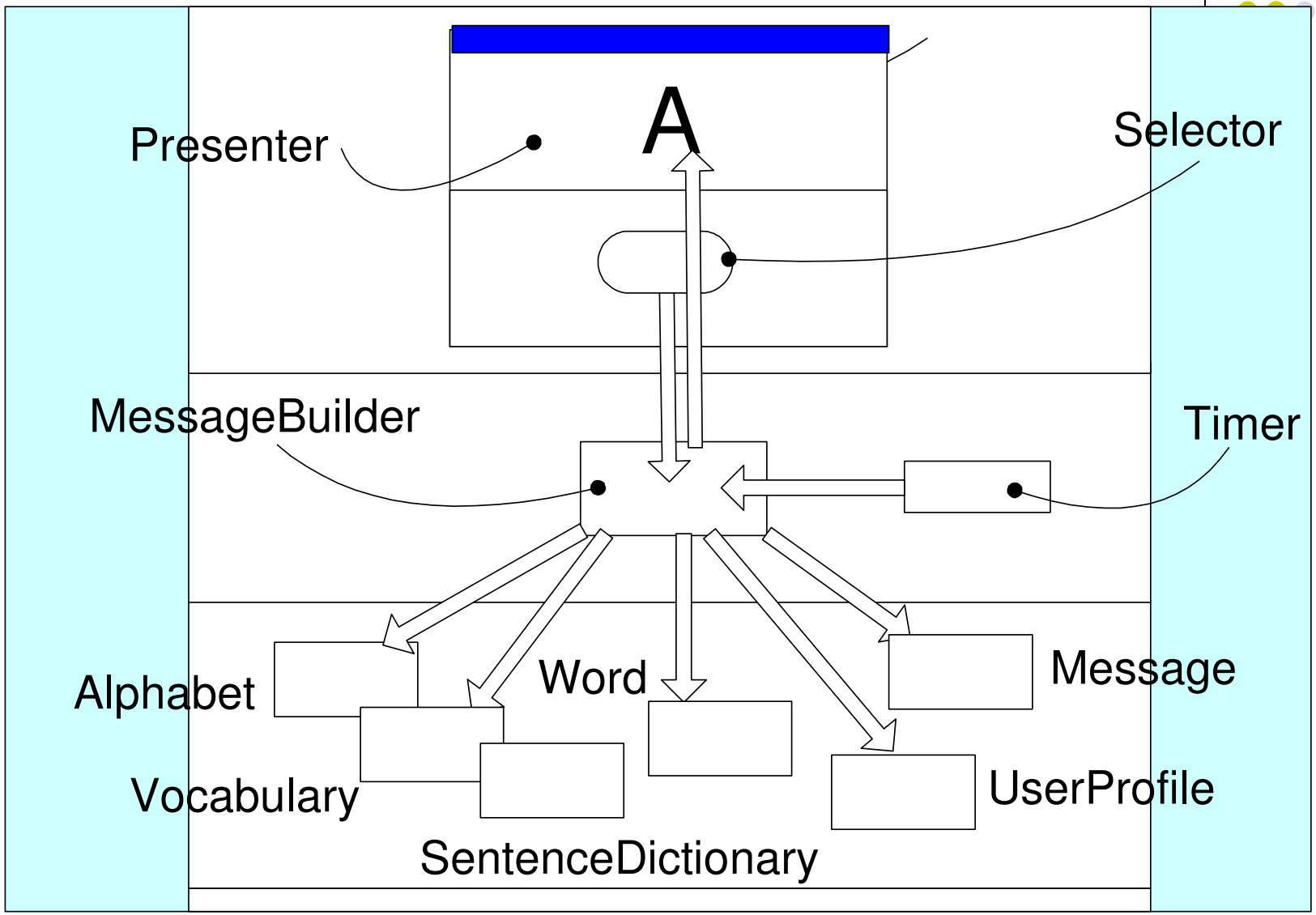


Centralizing Control

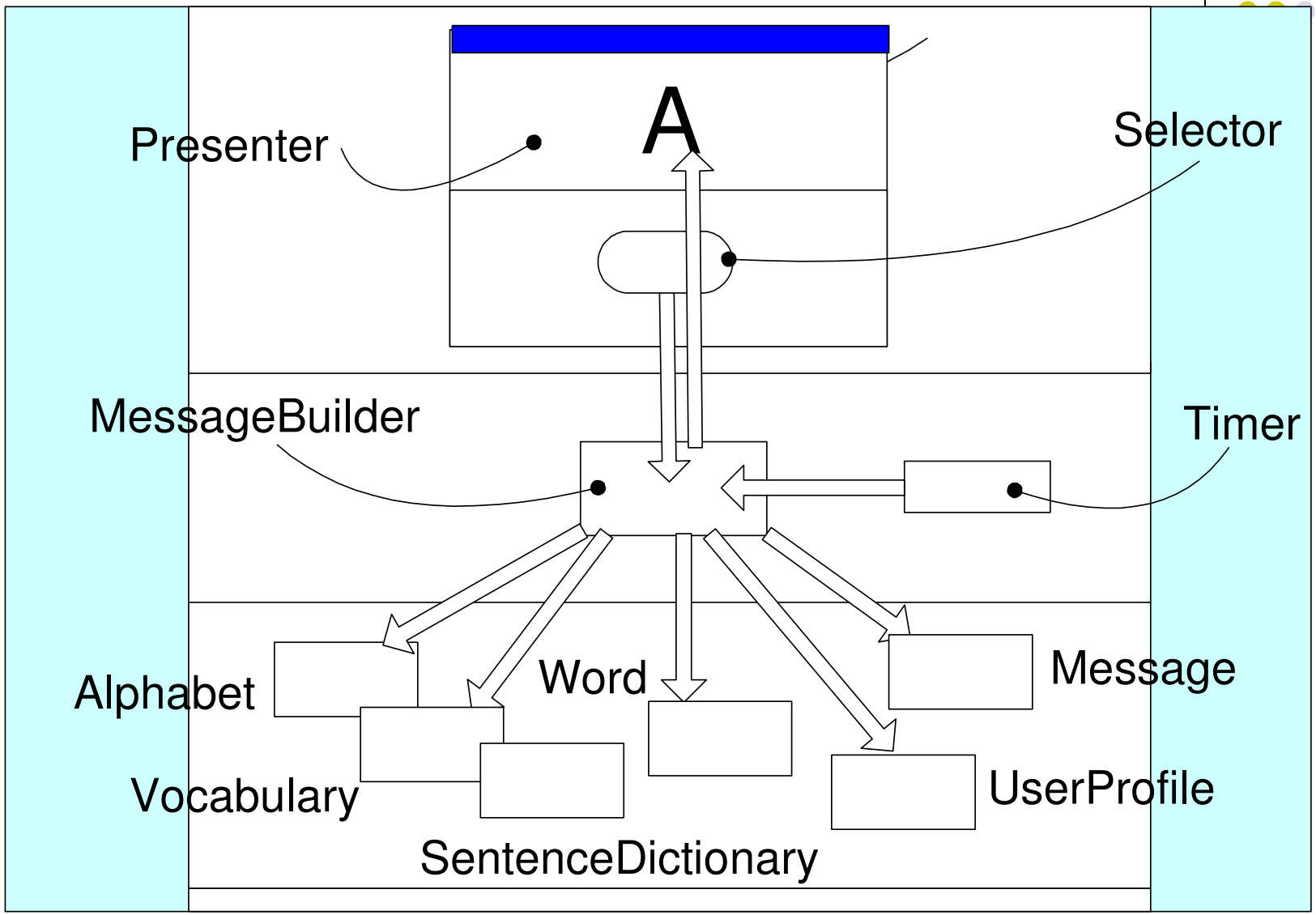


- Many decisions make the controller difficult to manage …
- What does it do when a user selects something? It could be a letter, a word, a space, a sentence, a command, a destination.
- When does it present each of the above to the user? It depends on the state of the message, what the user did last, and on the state of the software.

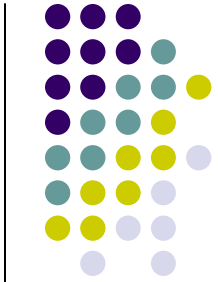
Controlling the Guessing



Handling the Selections

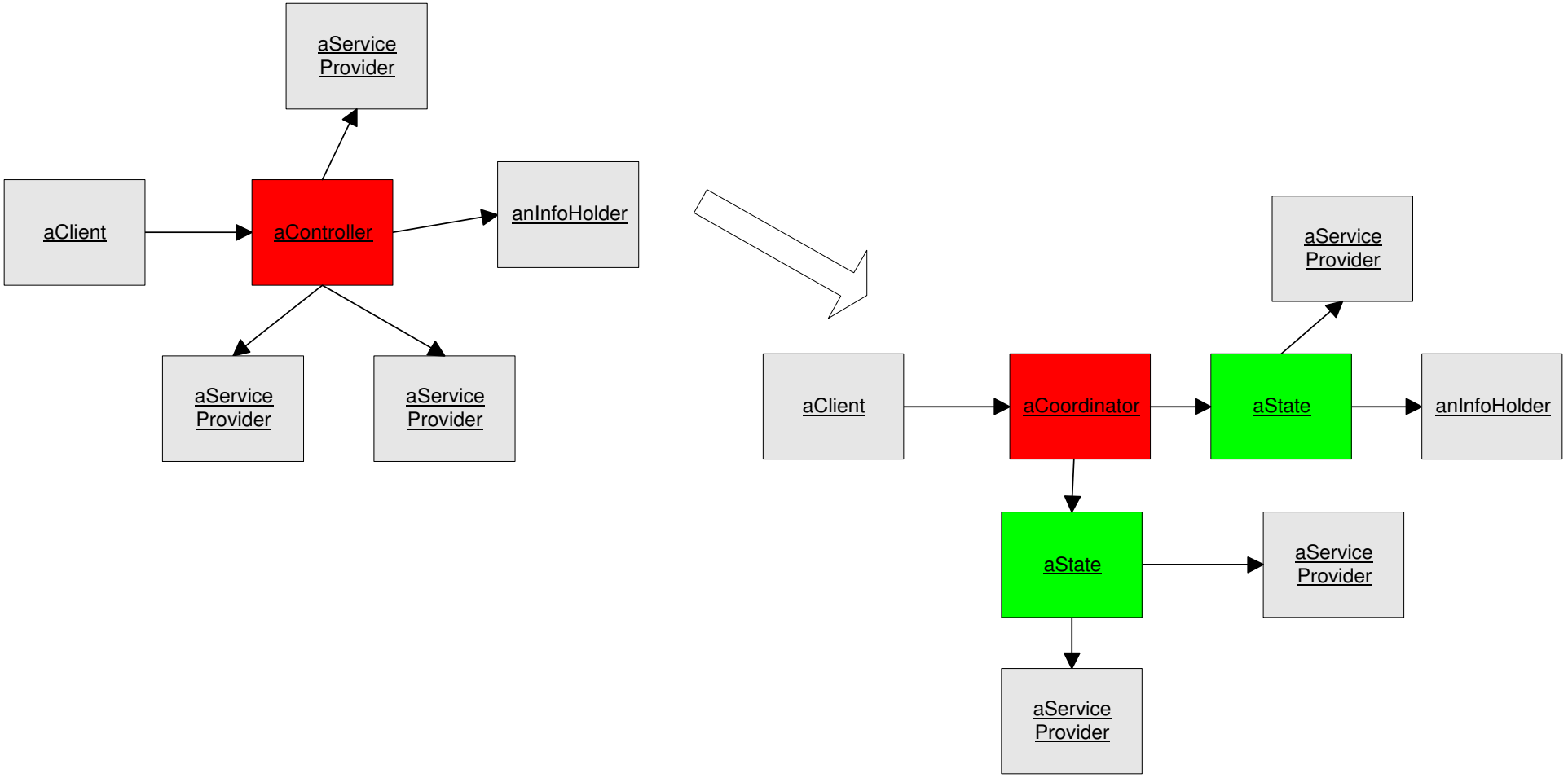


Delegating Control

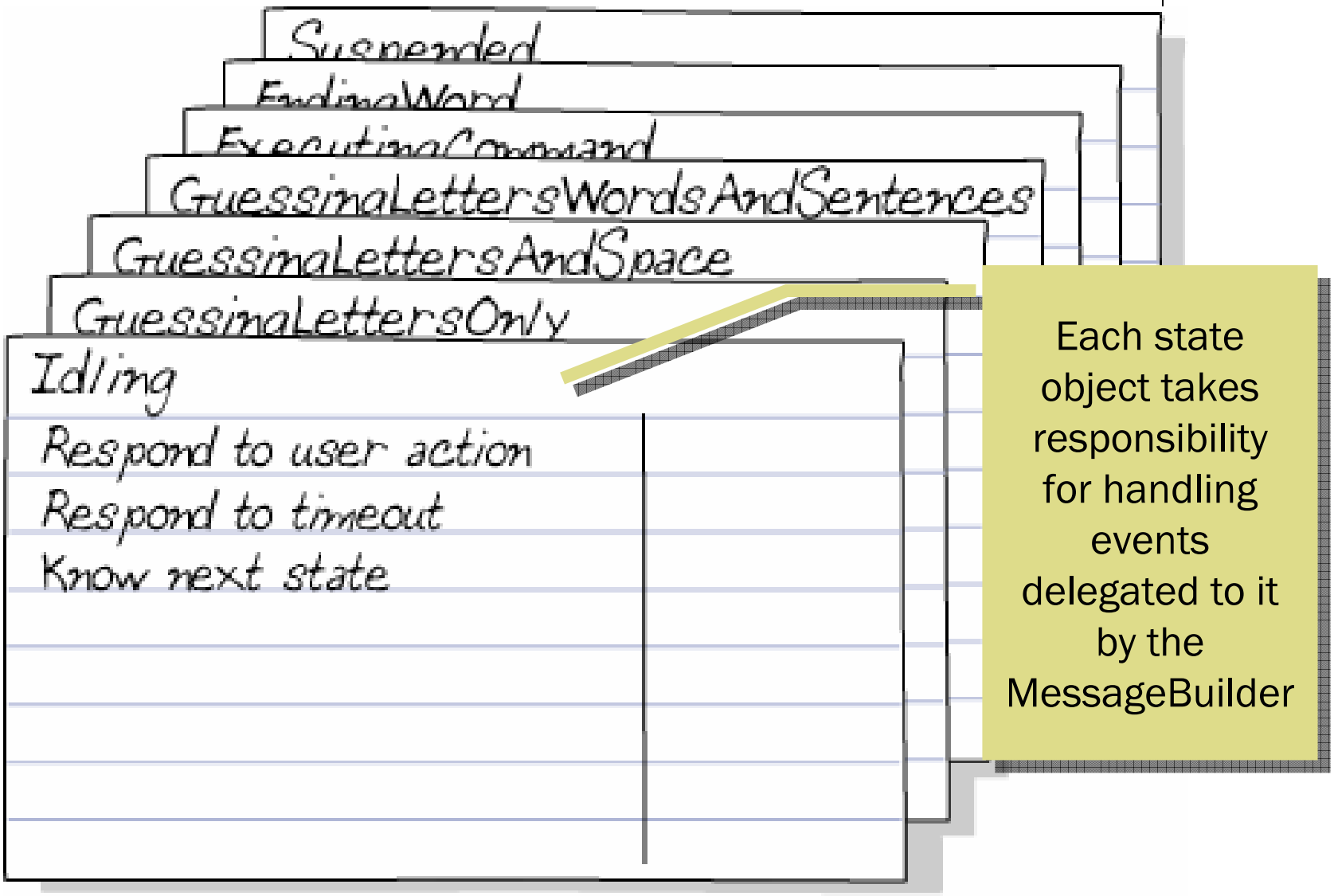


- Factors decision-making into helper objects.
- Replaces complex control with simpler coordination and delegation.
- Distributes focused logic into classes that implement singular, smaller roles.
- More classes and objects.

Controller = Coordinator + State

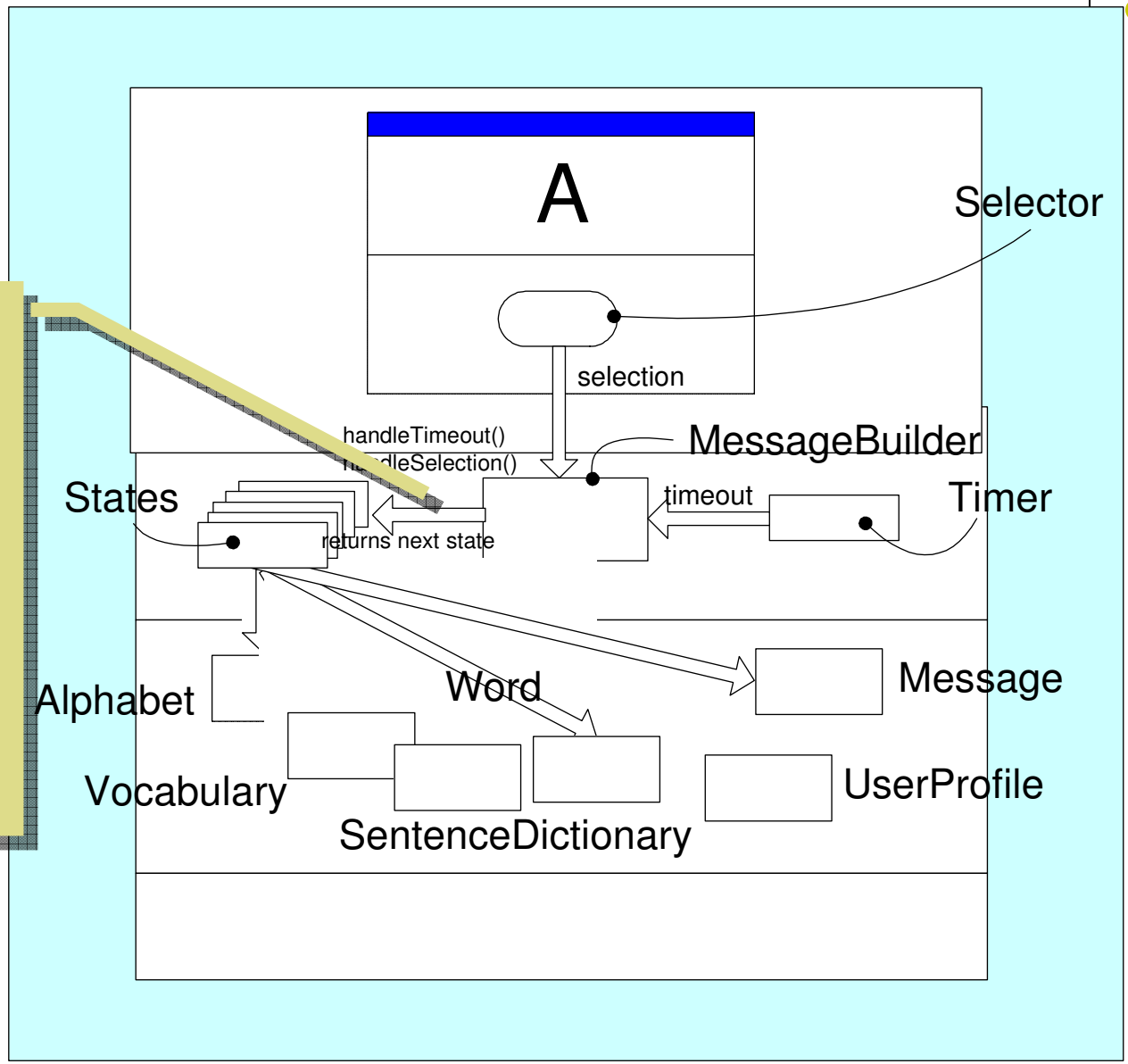


Applying the State Pattern to Simplify the Controller

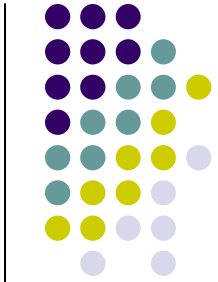


Applying the State Pattern

Complexity is still located in a “control center”. Can we simplify the control center by giving other objects more responsibility?

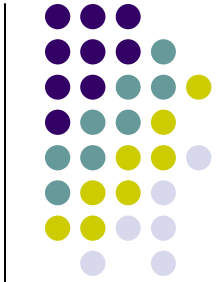


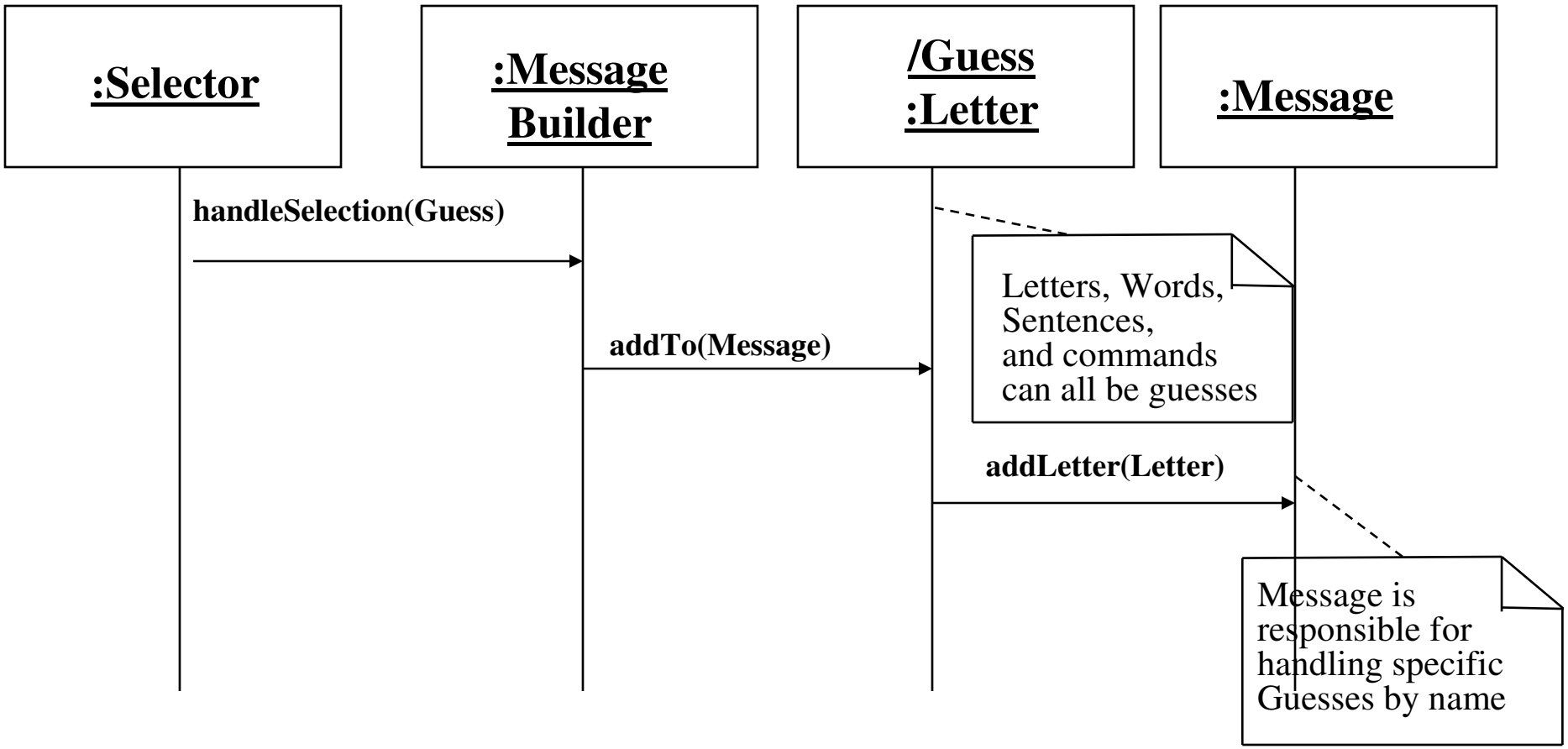
Inventing a common role—a Guess



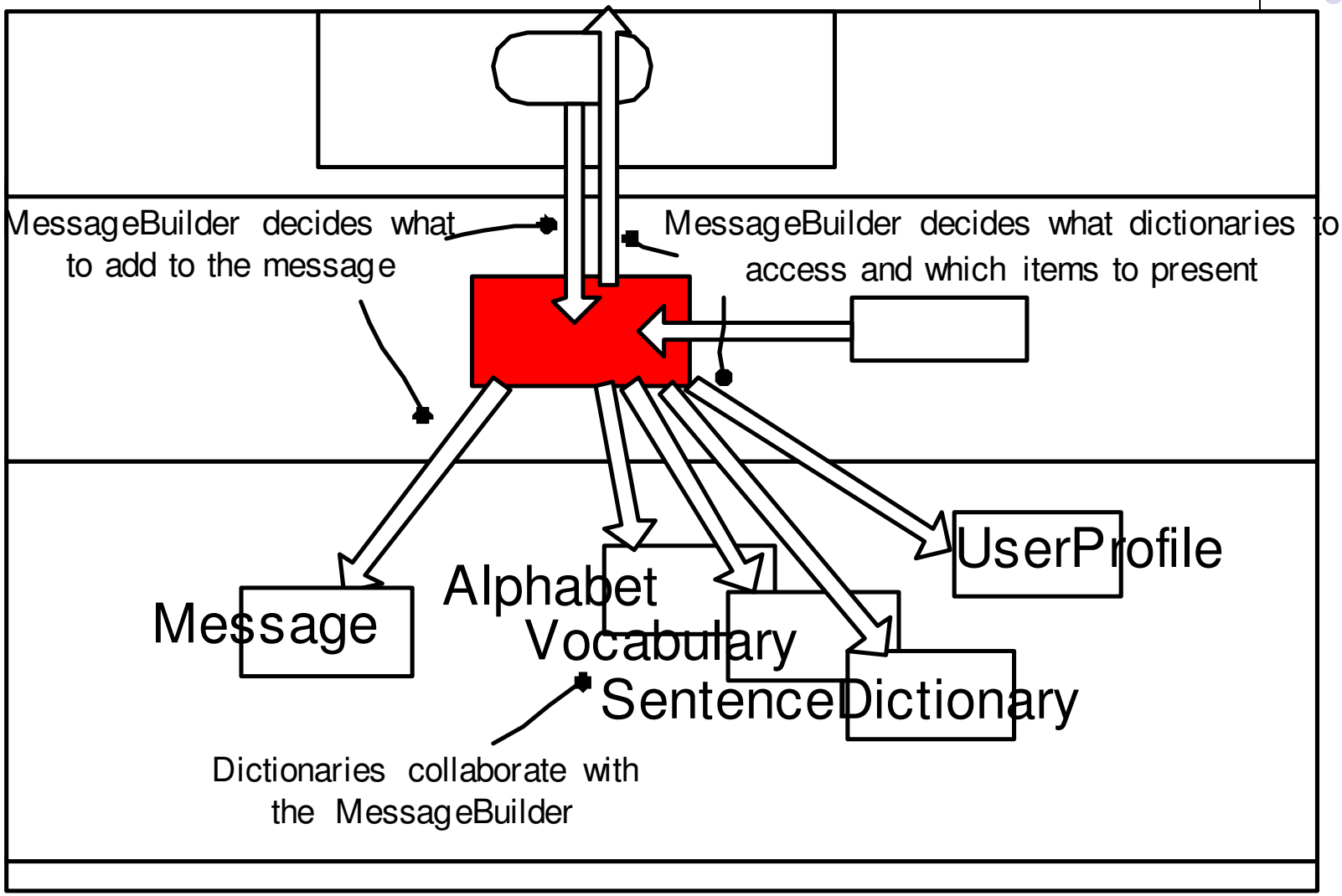
- In our first attempt, Letters, Words and Sentences didn't have many responsibilities. They held the item that the user could select and knew their spoken representation.
- But…if each of these objects (sharing the common role of a Guess) were responsible for directly adding themselves to a Message, type-based decisions could be eliminated!
- Message still knows about different kinds of Guesses, but collaboration between the MessageBuilder and Guesses are simplified.

Delegating Message Construction to Guess—our preferred design

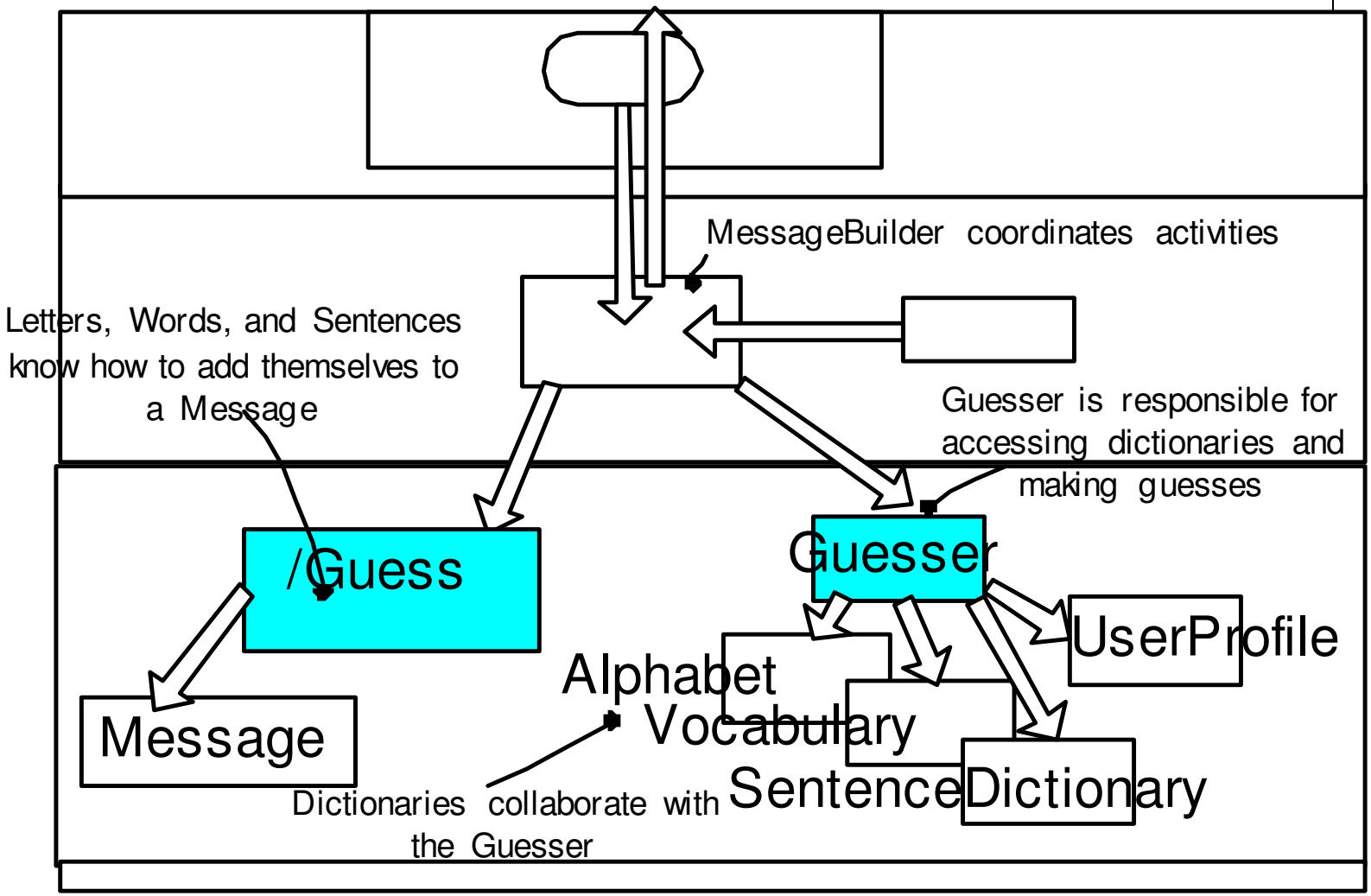




From Controlling Everything …



To Delegating Responsibilities

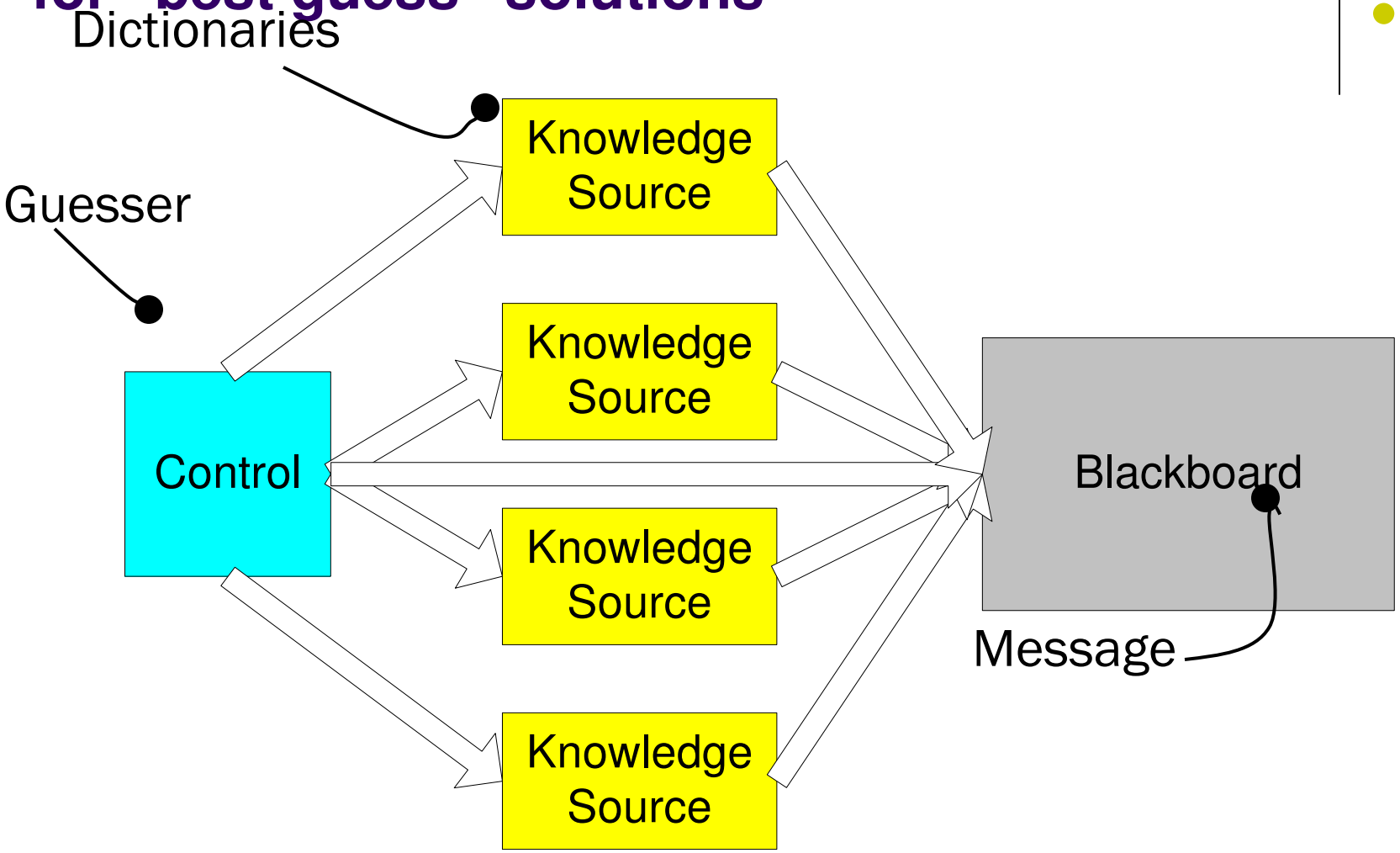


Further Delegation
a new invention

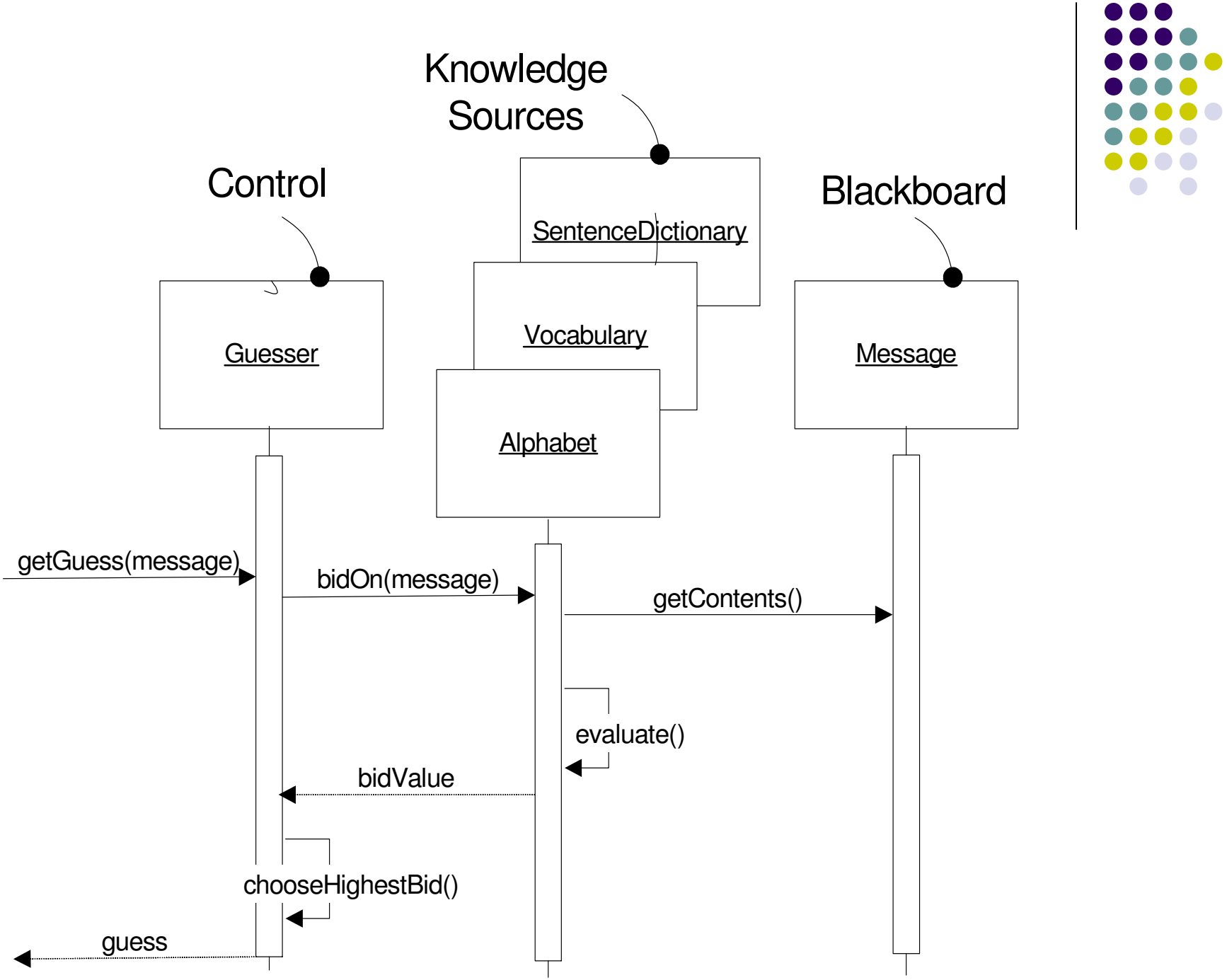
- A Guesser assumes responsibilities for guessing that were previously performed by an all controlling MessageBuilder.
- The Guesser hides the mechanisms of guessing, providing a black box for developing the guessing machinery.
- Since guessing is a complex task, maybe the Guesser can delegate some of its responsibilities, too.

The Blackboard Architecture

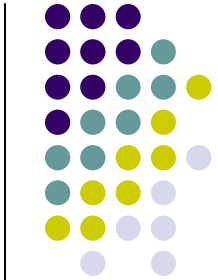
for "best guess" solutions



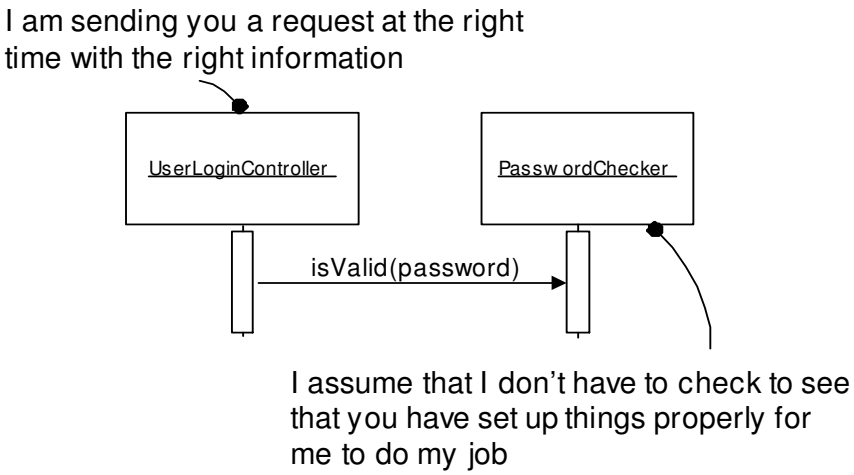
Modification: We can't write on the blackboard directly. Only after the user chooses a "guess" can we add it to the message.


Knowledge Sources
Control
Blackboard
SentenceDictionary
Guesser
Vocabulary
Message
Alphabet
getGuess(message)
bidOn(message)
getContents()
evaluate()
bidValue
chooseHighestBid()
guess

Trust Regions



- Carve your software into regions where “trusted communications” occur
- Objects in the same trust region communicate collegially, although they may still encounter exceptions and errors



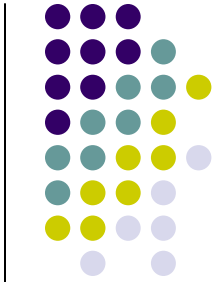
Collaboration Cases To Consider

- Collaborations between objects…
  - that interface to the user and the rest of the system
  - inside your software and objects that interface to external systems
  - in different layers or subsystems
  - you design and objects designed by someone else

Using An Untrusted Collaborator

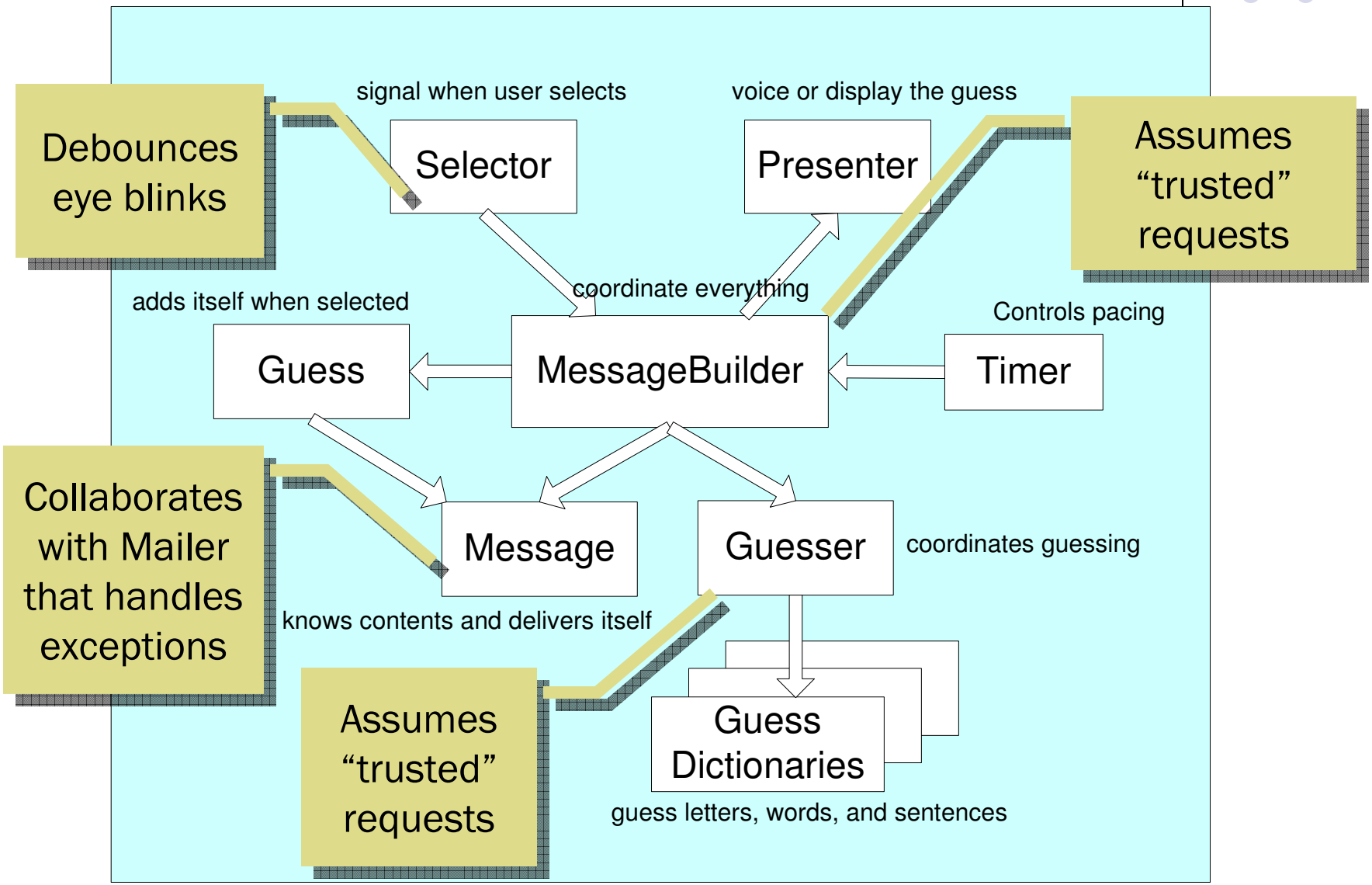
- Extra precautions may need to be taken. Especially if the client is responsible for making collaborations more reliable
  - Pass along a copy instead of sharing data
  - Check on conditions after the request completes
  - Employ alternate strategies when a request fails

Implications of Trust



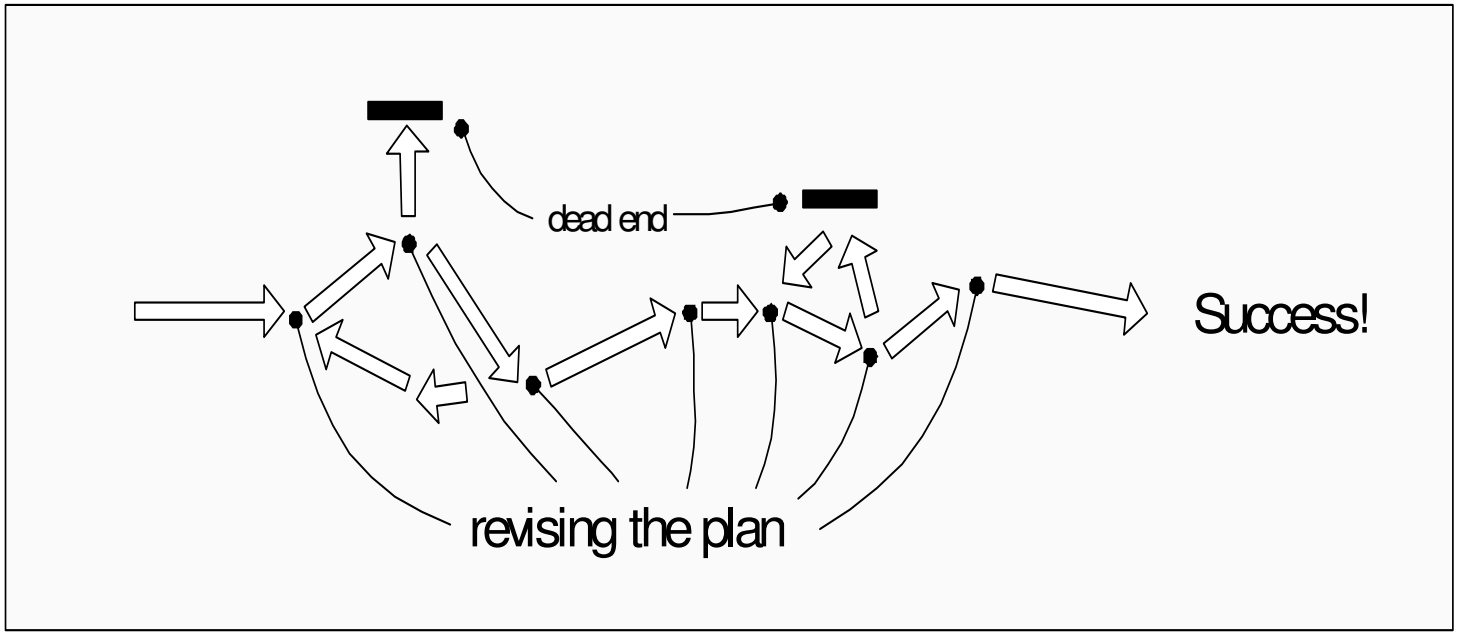
- In Speak for Me, all objects in the application “core” are within the same trust region
- Objects in the application control and domain layers assume trusted communications between each other
- Objects at the “edges”—within the user interface and in the technical services layers—make sure outgoing requests are honored and incoming requests are valid

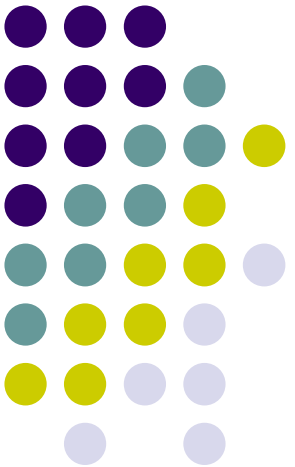
“Edge” Objects Take On Added Responsibilities



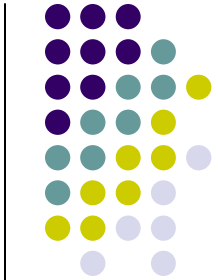
Agility and Design Rhythms

A design journey is filled with curves, switchbacks, and side excursions



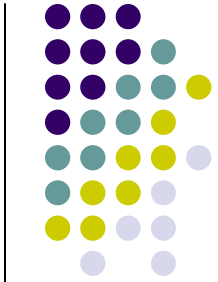


Design Isn't All Alike



- Software design problems vary:
  - ***Core design problems*** include those fundamental aspects of your design that are essential to your design's success (no not every part can be fundamental).
  - ***Revealing design problems*** when pursued, lead to a fundamentally new, deeper understanding.
  - ***The rest.*** While not trivial, the rest requires hard work, but far less creativity or inspiration.
- Each type of design problem warrants a different approach and has different rhythms to solving it.

Core Design Problems



Depending on your design requirements, you might nominate for the core:

- Mechanisms that increase reliability. These could include the design of exception handling and recovery, or connecting and synchronizing with other systems
- Mechanisms that increase performance
- Key objects in a domain model
- The design of important control centers
- Algorithms
- Mechanisms that enable specific areas of your software to adapt and flex
  - To changing environmental conditions
  - To evolving requirements

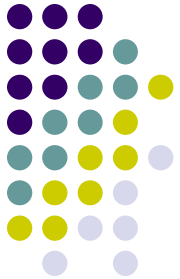
How Do You Decide What's Core?

- What are the consequences of fudging on that part of the design?
  - Would the project fail or other parts of your design be severely impacted? Then it's core.
  - When people have fundamentally different expectations, dig deeper. Someone may know something that others have ignored.
- Whether you classify something as part of the core or not, you'll still have to deal with it—it's a matter of emphasis.
  - Give design tasks the attention they deserve and be clear on your priorities.

Sorting Out The Rest From The Core

- It's easy to get caught up in a debate of what's core and what's in the rest. Don't.
- If you know that something is just basic design work that has to be there, nothing special, nothing fancy, it's probably part of the rest.
  - What about exception handling? Why isn't the 90% of your design work that supports the unhappy path scenarios a core design task? Well, depending on your project, they might be.
- Core problems should be given more attention. That doesn't mean the rest gets slighted. The rest just isn't at the top of your list.

Revealing Design Problems



- Sometimes you discover a core problem to be revealing, too.
- What distinguishes revealing problems is their degree of difficulty and the element of surprise, discovery and invention.
- Revealing design problems are always hard. They may be hard because…
  - Coming up with a solution is difficult—even though implementing it may be straightforward.
  - They may not have a simple, elegant solution.
  - They may not be solvable in a general fashion—each maddening detail may have to be tamed, one at a time.
  - They may require you to invent things that you have never before imagined.

Example: Redefining the Problem

- Often, redefining a problem doesn't simplify; it just opens up new possibilities.
  - In Java, C#, or Smalltalk, memory is automatically recovered from objects that are no longer used. Early implementations used reference counting to manage memory. This technique is simple, but very expensive. To speed up garbage collection algorithms, implementers of the languages redefined the problem—and now use sophisticated scavenging algorithms.

Some Revealing Problems Are *Really* Hard

Wicked problem characteristics:

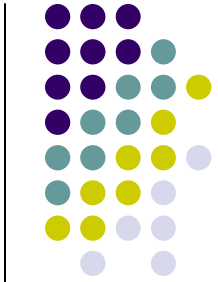
- They are hard to state concisely
- They can be symptoms of other problems
- Solutions are open to value judgments
- Solutions can be fuzzy or hard to describe
- There is no obvious way to verify that a proposed solution fixes the problem
- Their solutions have unforeseen consequences

. . . they are “tamed” not “solved”.

Observations On Solving Wicked Problems

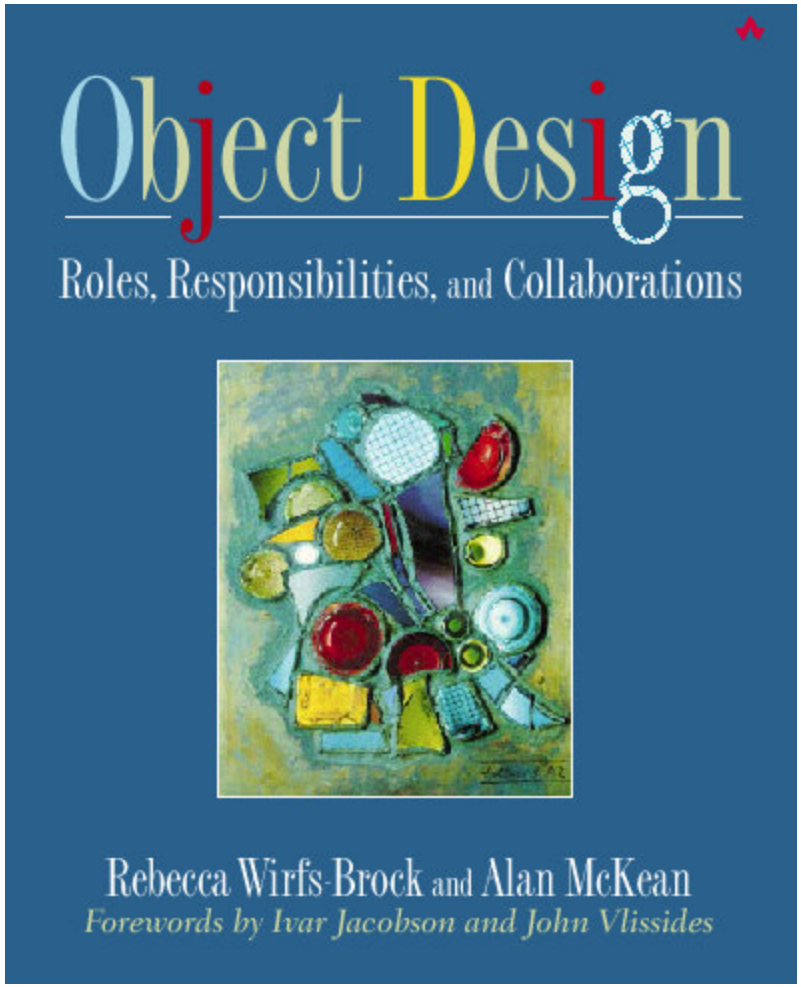
- To solve, you need to make connections between the problem and your past experiences and then experiment, extrapolate, combine partial solutions, and think.
- They either demand your undivided attention or lurk in your background thoughts.
- You may not get it right the first, second, or third try. They are hard to schedule.
- They cannot be solved by committee, although a proposed solution can be tuned by a group.

Agility Recognizes Design Values



- Agile designers acknowledge that:
  - Not all problems are alike. No single solution or approach fits all situations.
  - Agility isn't a single event; it's an ongoing process. Learn as you do.
  - Modeling should have a specific purpose. Documentation should add value.
- Articulate your design values and then focus on what's important.
- Sharpen your seeing, shaping, and problem solving skills.

Resources



- Read more about seeing and thinking, wicked problems, and object design strategies in our new book

  Object Design: Roles, Responsibilities and Collaborations, Rebecca Wirfs-Brock and Alan McKean, Addison-Wesley, 2003
- www.wirfs-brock.com for articles & resources