SWE 621

FALL 2018

DESIGN FOR CHANGE

LOGISTICS

- Midterms graded
  - Will return in class today
- HW3 due next week

IN CLASS EXERCISE

- What's an example of a decision you've hidden behind an interface?

DESIGN FOR CHANGE

- Design consists of making decisions.
- What happens when these decisions change?
- Some decisions may be more likely to change than others.
- How can we design software in ways that make likely to change decisions easier to change?

CHOOSING ELEMENTS

- We've looked at three ways so far to divide systems into elements
  - Design as domain modeling: choose elements that correspond to domain elements
  - Design for abstraction: choose elements that hide irrelevant details and make writing code easy
  - Architectural styles: choose elements which respect constraints which enable quality attributes to be satisfied
- Will look at a fourth today: dividing systems into elements to support change

KEY WORDS IN CONTEXT (KWIC) PROBLEM

- Accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters.
- Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line.
- Outputs a listing of all circular shifts of all lines in alphabetical order.

"CLASSIC" FLOW CHART DECOMPOSITION

- Each module (except master control) corresponds to step in flow chart
- Input: reads data from input and stores into data structures
- Circular shift: prepares data structure shifting words
- Alphabetizer: alphabetizes words
- Output: creates output listing
- Master control: invokes other modules

MODULARIZATION 2

- Line storage: functions and subroutines which give access to line data structures
- Input: reads input, calls line storage to store lines
- Circular shifter: offers interface for accessing circularly shifted lines as index on same underlying data structure
- Alphabetizer: alphabetizes words
- Output: renders data to console
- Master control: invokes other modules

WHAT ARE SOME DESIGN DECISIONS WHICH MIGHT CHANGE?

1. Input format: how is data entered into system
2. In memory: reading and storing data in memory rather than externally on disk
3. Representation: the data structure used to store data efficiently in memory
4. Index: generating output as an index into original data rather than as a copy of original data
5. Eager sort: make search faster by sorting list rather than doing a search on demand

DIFFERENCES BETWEEN MODULARIZATIONS

- Changing (2) in memory decision and (3) data structure decisions would require making edits to **all** modules in first decomposition
  - Input: reads data from input and stores into data structures
  - Circular shift: prepares data structure shifting words
  - Alphabetizer: alphabetizes words
  - Output: creates output listing
  - Master control: invokes other modules

DIFFERENCES BETWEEN MODULARIZATIONS

- Changing (2) in memory decision and (3) data structure decisions would require making edits to **one** module in second decomposition
  - Input: reads data from input and stores into data structures
  - Circular shift: prepares data structure shifting words
  - Alphabetizer: alphabetizes words
  - Output: creates output listing
  - Master control: invokes other modules

WHY?

- Knowledge of the exact way that the lines are stored is entirely hidden
  - Decisions (2) and (3) can be changed, and only the Line Storage module would ever know

INFORMATION HIDING

- Can change a decision locally in a module **without** change rippling to cause change in other module
- Modules characterized by knowledge of a design decision and what it hides from others
- Usually expressed as inverse: here's what decisions are exposed to clients through interface

INFORMATION HIDING VS. ABSTRACTION

- Isn't this abstraction all over again?
- Goal is different
  - Abstraction: offer operations that make writing client code compact and easy
  - Information hiding: enable design decisions in module to change
- Are there examples where a design increases abstraction but decreases information hiding?

ASIDE: GOOD ABSTRACTIONS REALLY MATTER

- Parnas originally estimated that KWIC could be built in a week or two in 1972
  - Assumed C style language with few collection abstractions
- Can implement in a few dozen lines with modern collection abstractions

EXAMPLE: UNIVERSAL RESOURCE IDENTIFIER (URI) DESIGN

- Uniquely describes a resource
  - https://mail.google.com/mail/u/0/#inbox/157d5fb795159ac0
  - https://www.amazon.com/gp/yourstore/home/ref=nav_cs_ys
  - http://gotocon.com/dl/goto-amsterdam-2014/slides/StefanTilkov_RESTIDontThinkItMeansWhatYouThinkItDoes.pdf
  - Which is a file, external web service request, or stored in a database?
    - It does not matter
- As client, only matters what actions we can *do* with resource, not how resource is represented on server

PRIVATE MEMBERS

- Information hiding offered important motivation for inclusion of access control in modern OO languages
  - Can specify **private** or **protected** access to limit access to "implementation details" (a.k.a. hidden design decisions) that clients should not know about
- But principle applies much more broadly to design decisions
  - Not necessarily about computing / caching data or how data is stored
  - Design decisions may not be closely associated with a data structure or method

EXAMPLE: URI DESIGN

- Which is better?
    - http://myservice.com/cities
    - http://myservice.com/cities.cfm
    - http://myservice.com/cities.aspx

TYRANNY OF THE DOMINANT DECOMPOSITION

- Many design decisions
  - Can you hide all of them?
- No
  - Inevitably, will make some design decisions easier to change than others

COSTS OF INFORMATION HIDING

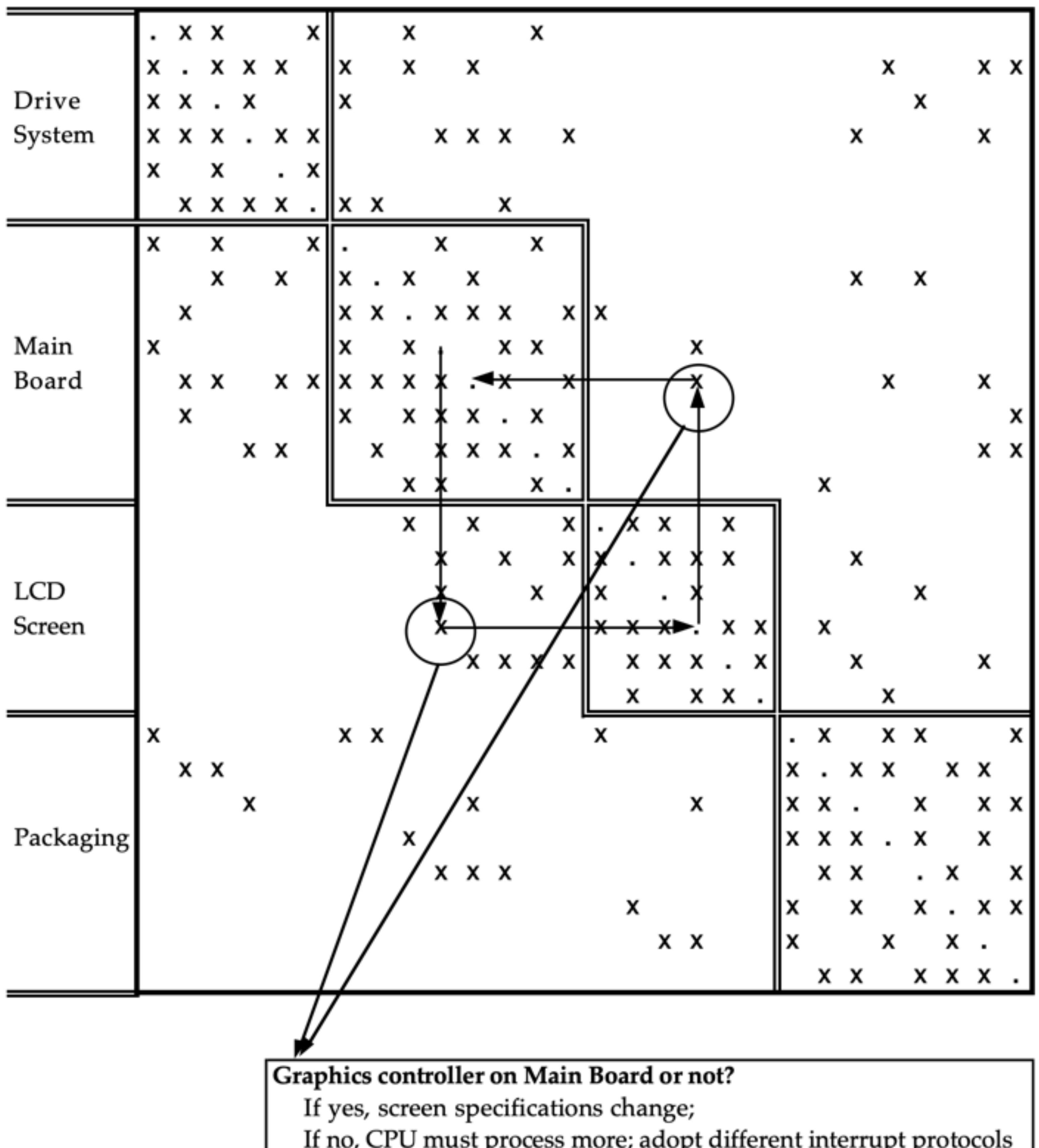
- Can't hide everything
  - Will inevitably make some design decisions easy to change, others harder to change
- What should you hide?
  - Decisions that are most likely to change
  - Get best payoff by reducing cost of making these expected changes easier

REAL OPTIONS

- Having a design decision that you can change at low cost creates **options**
  - Second modularization offers the **option** to consider whether data should be stored or handled online at low cost
- Option provides the right to make a change without the obligation
- Important mechanism for risk mitigation
  - If decision is wrong, project might fail
  - Mitigate risk by making it easy to change decision after made, by reducing dependencies on decision

EXAMPLE: OPTIONS IN A COMPUTER

Design Structure Matrix Map of a Laptop Computer



MAKING DEPENDENCY STRUCTURE EXPLICIT

- How do you know what options you have?
- Build a design structure matrix (DSM)
- Design decisions (or "design parameters") are rows
- What they depend on (every other design decision) are columns
- What might happen if design decision A changed?

	A	B	C
A	.		
B	X	.	X
C		X	.

best choice of design decision B **depends on** choice of design decision A

IN CLASS ACTIVITY: BUILD A DSM FOR A LAYERED ARCHITECTURE

- In groups of 2 or 3, build DSM that depicts dependency structure of a system in the layered architectural style

CREATING MODULARITY

- How do you break dependencies between modules that you'd like to be independent?

- Organize dependency structure so that there are shared decisions that others depend on and assert that they won't change.

	A	B	C
A	.		
B	X	.	X
C		X	.

	I	A	B	C
I	.			
A	X	.		
B	X		.	X
C			X	.

- B and C are now independent of A, because they depend on I

DSM FOR MODULARIZATION 1

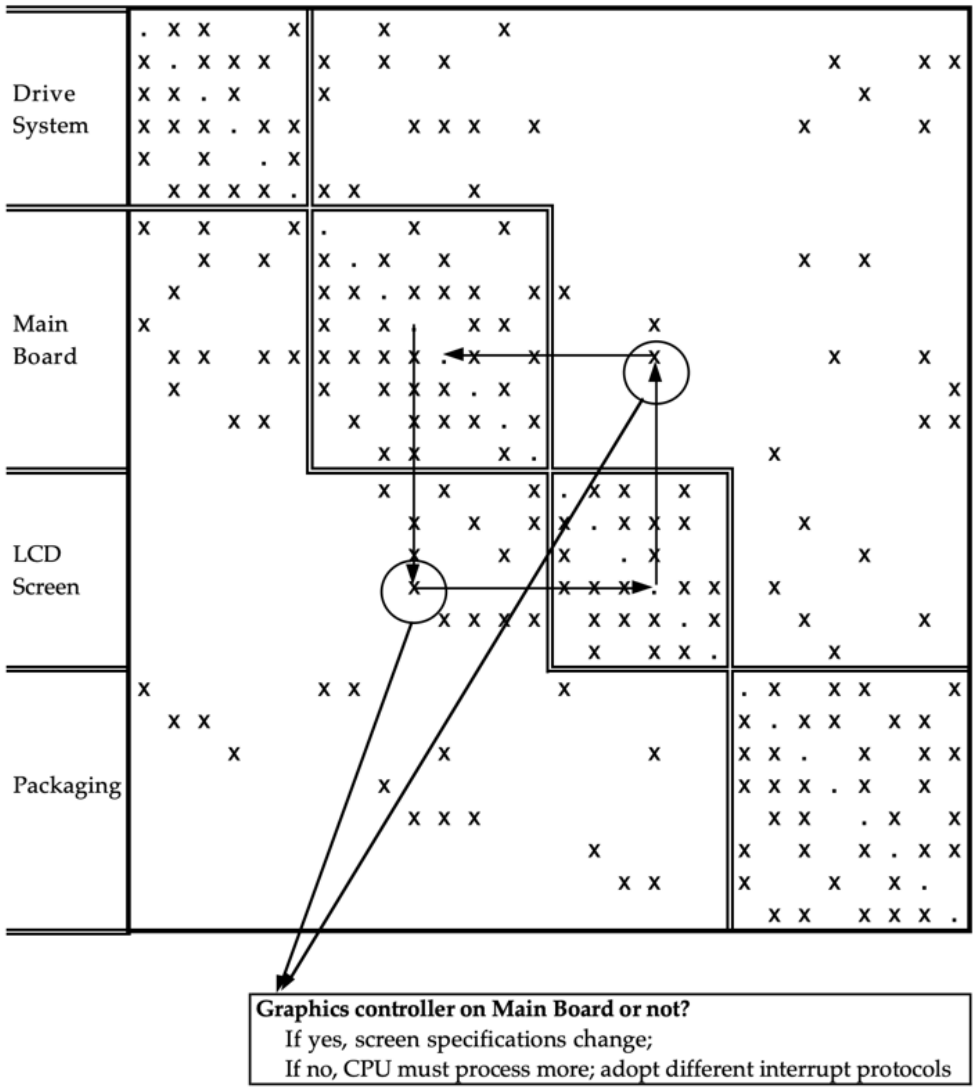
	A	B	C	D	E	F	G	H	I	J	K	L	M
A - In Type	.												X
B - In Data	X	.	X		X	X		X	X			X	
C - In Alg	X	X	.										
D - Circ Type				.									X
E - Circ Data		X		X	.	X		X	X				
F - Circ Alg		X		X	X	.							
G - Alph Type							.						X
H - Alph Data		X			X		X	.	X			X	
I - Alph Alg		X			X		X	X	.				
J - Out Type										.			X
K - Out Data										X	.	X	
L - Out Alg		X						X		X	X	.	
M - Master	X			X			X			X			.

Type: procedure interfaces
Data: data structures decisions
Alg: algorithm decisions

DSM FOR MODULARIZATION 2

	N	A	D	G	J	O	P	B	C	E	F	H	I	K	L	M
N - Line Type	.															
A - In Type		.														
D - Circ Type			.													
G - Alph Type				.												
J - Out Type					.											
O - Line Data	X					.	X									
P - Line Alg	X					X	.									
B - In Data		X						.	X							
C - In Alg	X	X						X	.							
E - Circ Data	X		X							.	X					
F - Circ Alg	X		X							X	.					
H - Alph Data	X			X								.	X			
I - Alph Alg	X			X								X	.			
K - Out Data					X									.	X	
L - Out Alg	X				X									X	.	
M - Master	X	X	X	X	X											.

Design Structure Matrix Map of a Laptop Computer



CONWAY'S LAW

- The structure of a designed system is isomorphic to the organizational structure of those who built it
- If a design decision depends on a design decision made by another (e.g., developer, team, company), there must be coordination when this decision changes (e.g., email, face to face meeting) to stay consistent

SOCIO-TECHNICAL CONGRUENCE

- What happens when the required coordination does not happen?
  - e.g., the infrastructure team that owns the datastore just changed the query engine
- Poor design (if system still works, but less well)
- Defects (if system no longer works)
  - Can observe empirically by comparing decision dependencies (module references) against coordination (emails sent) to find divergences, which correlate with defects

Cataldo, M., & Herbsleb, J. D. (2013). Coordination Breakdowns and Their Impact on Development Productivity and Software Failures. IEEE Transactions on Software Engineering 39(3), 343-360.

INFORMATION HIDING AND COORDINATION

- Want to have clear idea of what the external interface of your team constitutes
  - What design decisions which might change would others care about?
  - Need to manage coordination around these decisions

HYRUM'S LAW: A PESSIMISTIC VIEW

- With a sufficient number of users of an API,
- it does not matter what you promise in the contract:
- all observable behaviors of your system
- will be depended on by somebody.

- Interfaces evaporate with additional clients, as every observable behavior eventually is depended on by someone

http://www.hyrumslaw.com/

SUMMARY

- Different organizations of functionality into elements leads to different design decisions being modularized and hidden behind interfaces
- What is hidden in a module is a design decision, not just a variable or method
- Hidden decisions offer real options, making it cheaper to explore alternative designs
- Technical dependencies require coordination between people, or defects may result

IN CLASS ACTIVITY

DESIGN ACTIVITY: DESIGN TODO APPLICATION FOR CHANGE

- Form group of 2 or 3
- Consider again Todo application requirements
  - User interactions with todos: add, delete, rename, complete, copy
  - Display todos to user
  - Persist todos
- Design a todo application for change, hiding decisions likely to change behind interfaces
- Deliverables:
  - component and connector model showing elements in your system
  - list of functionality for each element
  - list of important design decisions
  - DSM which shows dependencies between these design decisions
  - short description of how your design supports changes to a subset of these decisions

DESIGN ACTIVITY: STEP 2: DISCUSSION