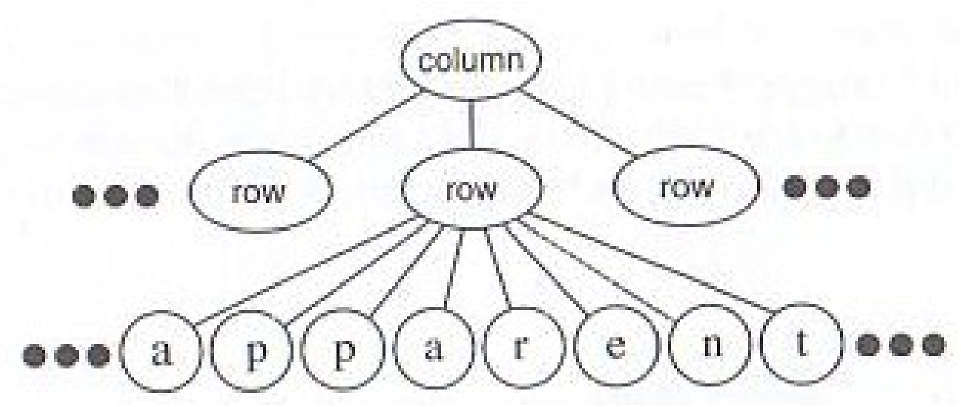
Plan for Today

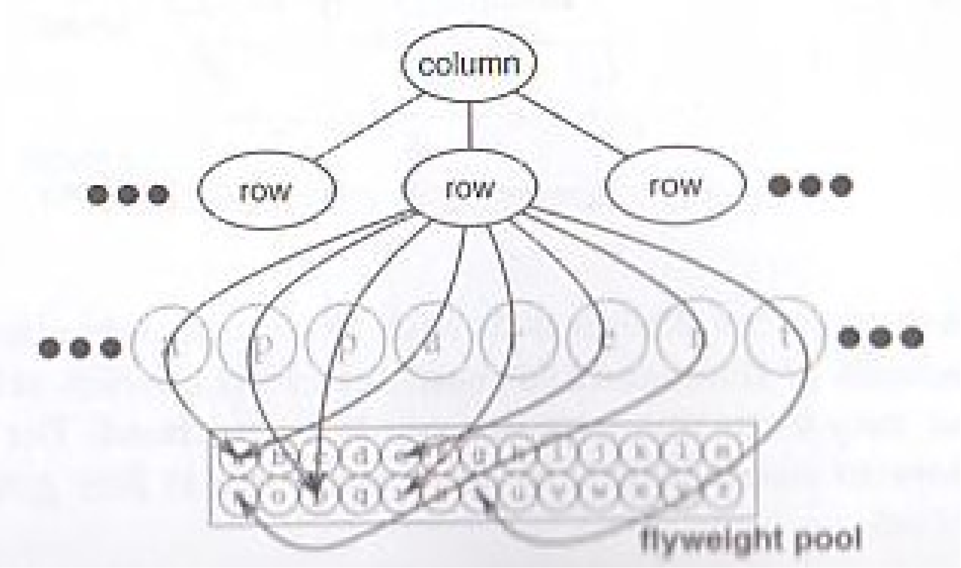
- Observer pattern
- Antipatterns
- Code smell

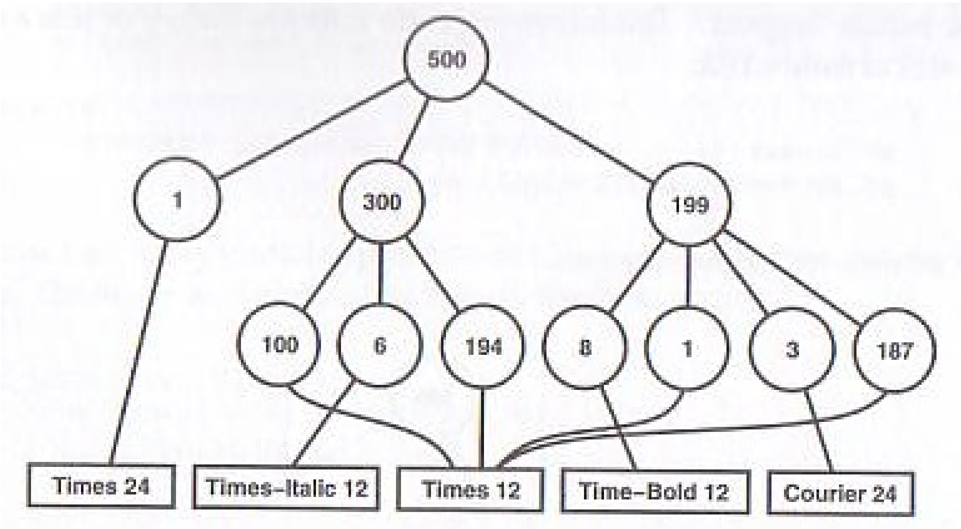
- Final

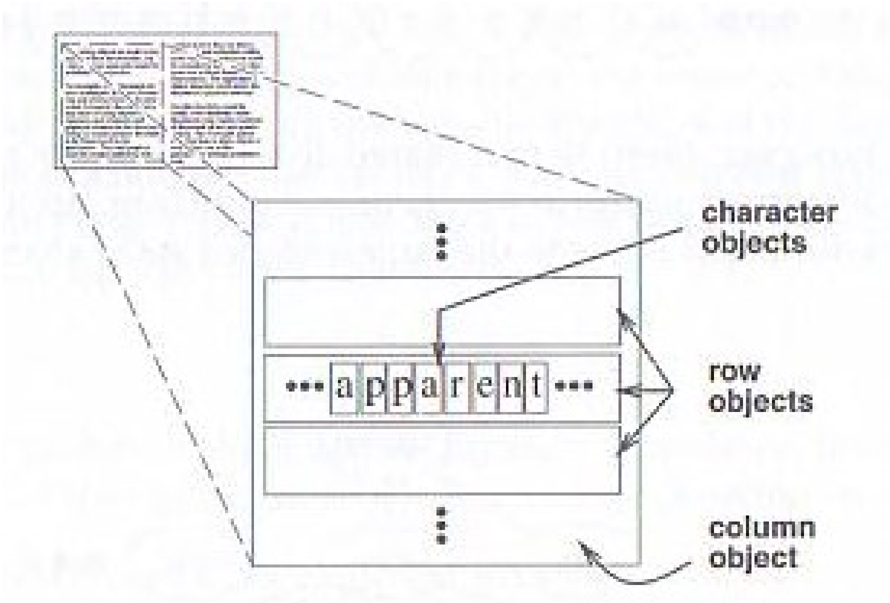
Lecture 53 – 5 December

Task 7 Questions?



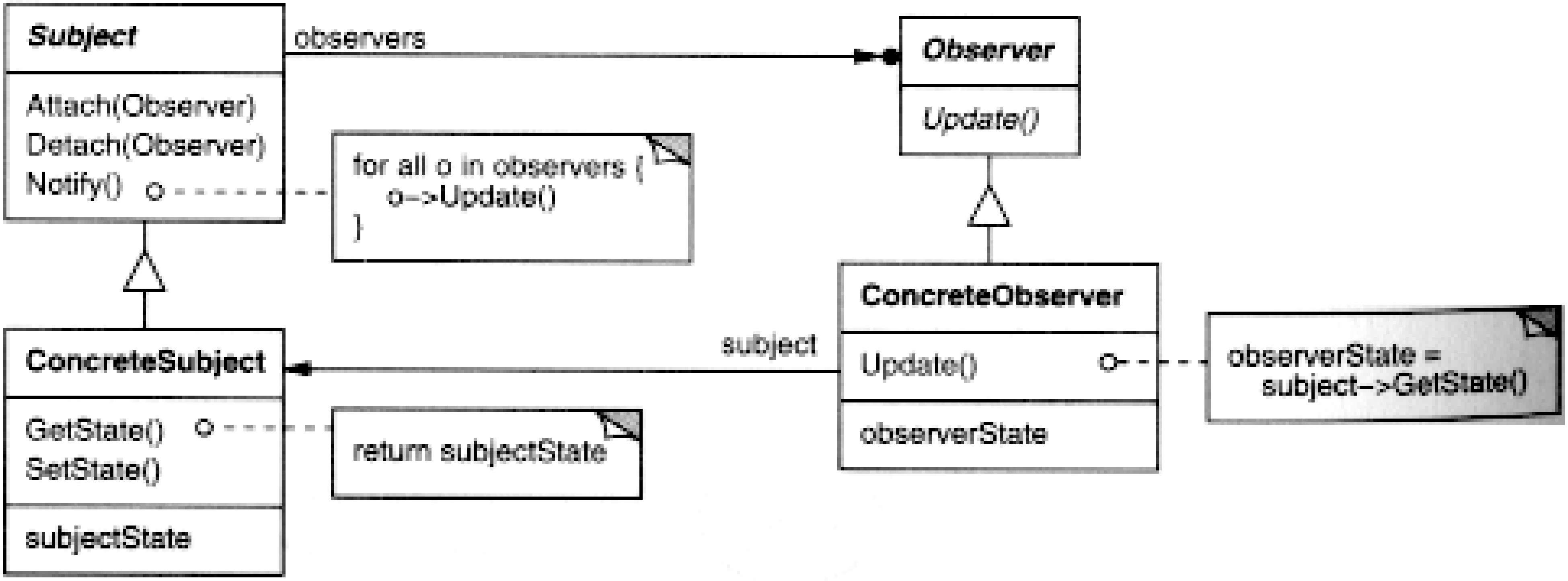






Observer Pattern

- Defines one-to-many dependency between objects so when one object changes state, all its dependents are notified and updated automatically



Observer Pattern

- Think "wireless network"
- One-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
  - notification responsibility belongs to initiator
  - update responsibility delegated to dependents, which may not care
- Application
  - when a change to one object requires changing others, and you do not know how many objects need to be changed
  - when an object should be able to notify other objects without making assumptions about who these objects are – very loose coupling
  - when an abstraction has two aspects, one dependent on the other
    - encapsulating these aspects in separate objects lets you vary and reuse them independently

Observer Pattern

- Consider
  - visualize interconnection network
  - compare strategies
    - direct coupling (wired network)
      - connect everything that needs to know
        - one-to-one mapping
        - one-to-many mapping
      - we know how to do this
        - “push” model
        - “pull” model
    - indirect coupling (wireless network)
      - allow everything that needs to know to know
        - one-to-one mapping
        - one-to-many mapping
      - how do we do this?
        - recall model-view-controller pattern from CS282

Observer Pattern

- Considerations and consequences
  - abstract coupling
  - networked communication
    - unicast
    - broadcast
  - behavioral considerations
    - cascading updates: causing a nuclear explosion (fission)
    - feedback updates: causing you to update yourself…
    - thrashing updates: causing you and other object(s) to fight each other
  - notification grouping
    - when to announce something?
      - non-atomic: after each operation … better be self-consistent!
      - atomic: after each operation group
  - notification administration
    - dangling recipients: “ghosts”

Observer Pattern

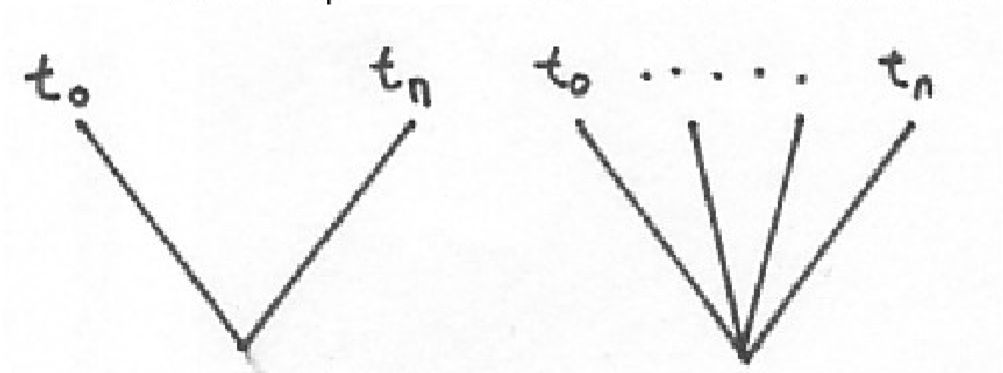
- Implementation
  - plug-and-play organization
  - centralized authority
    - possibly singleton!
  - register observers
    - listeners

```
public interface I_Observable {
  public void _register  (I_Observer observer);
  public void _unregister(I_Observer observer);
}
public interface I_Observer {
  public void _notify(???);
}
public class Logger implements I_Observable {
  private List<I_Observer> _observers = new ArrayList<I_Observer>();
  public _register(I_Observer observer) {
    _observers.add(observer);       // should error check
  }
  public _unregister(I_Observer observer) {
    _observers.remove(observer);    // should error check
  }
  public void log(String entry) {
    ...
    for (I_Observer observer : _observers) {
      observer._notify();
    }
  }
  ...
}
public class SomeClass implements I_Observer {
  public SomeClass() {
    Logger.getInstance()._register(this);
  }
  public void _notify(???) {
    System.out.println(“Yeah, I got notified!!”);
  }
  ...
```

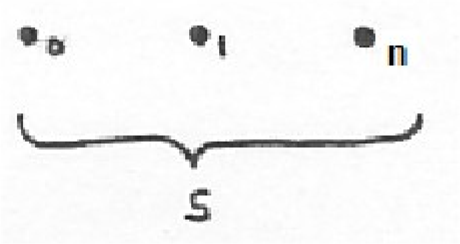
Case Study : Observer + Interpreter Pattern

wait + snap

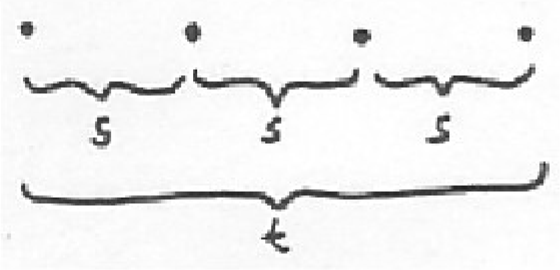
wait + move + wait + move + wait +move



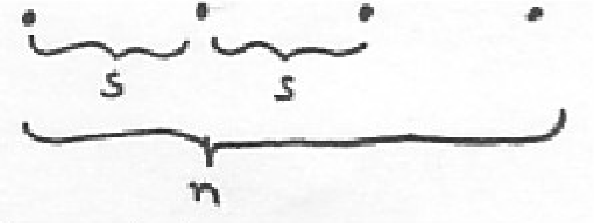
- *n* times over *s* seconds



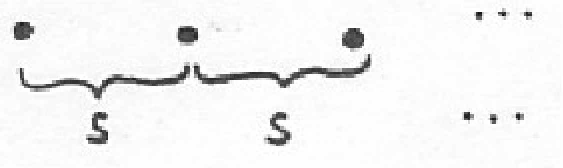
- every *s* seconds for *t* seconds



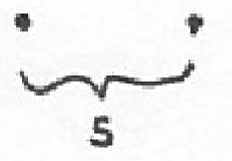
- every *s* seconds for *n* times



- continuous; every *s* seconds until canceled



- oneshot; one time in *s* seconds then self-cancels



4.3.1.3. Blocking

Executes asynchronous handoff vector:

- A calls B
- A blocks
    - B proceeds
    - B wakes A
    - B dies
- A proceeds

4.3.2. Response

The receptor will always respond immediate with acknowledgement of receipt and its intentions:

- ACCEPTED_PROCEED -- done immediately
- ACCEPTED_WAIT -- will do and call back when done
- REJECTED -- processable but unable
- IGNORED -- not applicable receptor (common in broadcast semantics)
- PENDING -- not seen by any receptor (i.e., not sent yet)
- ACKNOWLEDGED -- already seen by another receptor (i.e., chain letter in broadcast semantics)

4.3.2.1. Acceptance

Message receptor will service

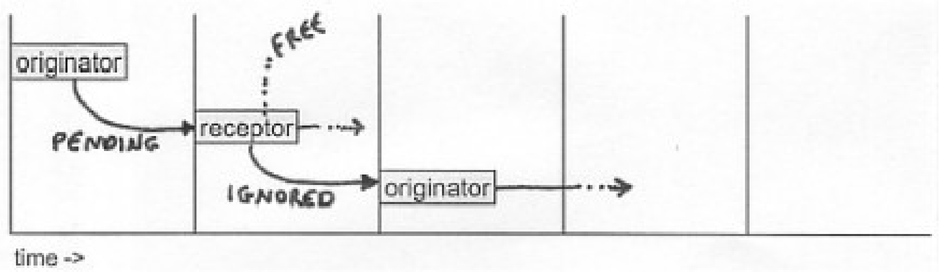
4.3.2.2. Rejection

- message receptor could service but is already servicing another message
    - message gate
- message originator may assign automatic rejection-followup policy:
    - DISCARD_INFORM -- abandons message but later informs originator when manual resubmit is possible
    - DISCARD_QUIET -- abandons message and does not inform originator
    - RESUBMIT_INFORM -- resubmits message to itself when possible and informs originator of such
    - RESUBMIT_QUIET -- resubmits message to itself when possible and does not inform originator

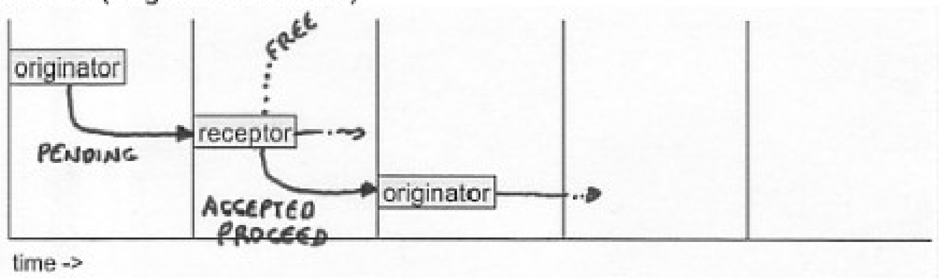
4.3.2.2.1. Finalizer

- completed -- receptor serviced message
- resubmitted -- receptor resubmitted deferred message to itself
- discarded -- receptor rejected message but originator may resubmit manually now

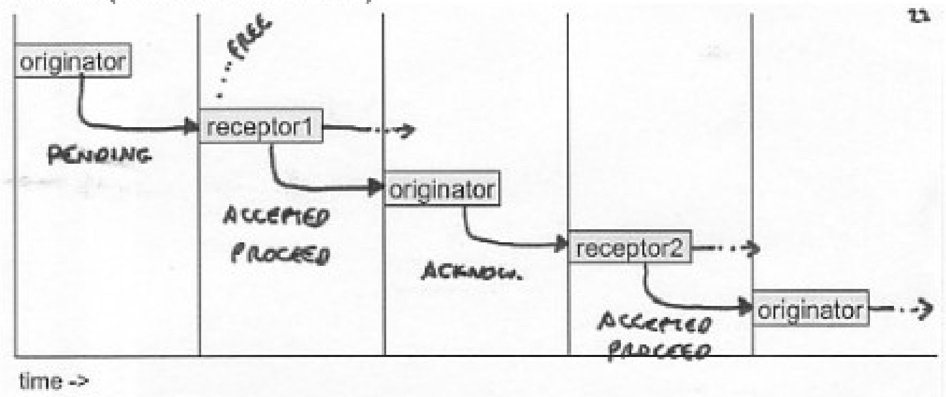
IGNORED



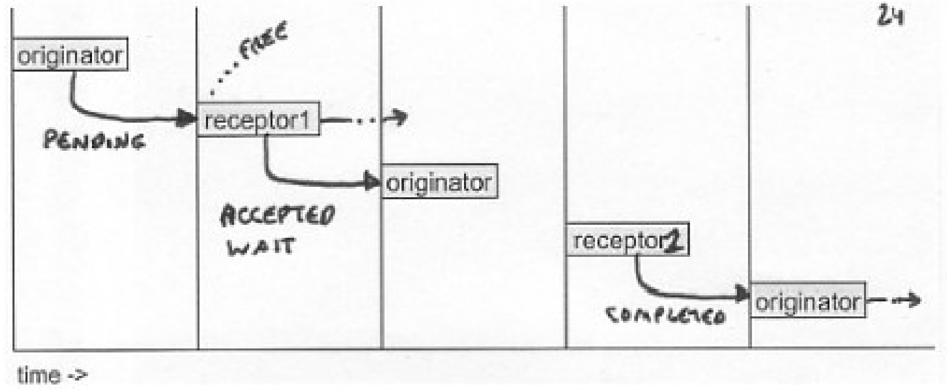
ACCEPTED_PROCEED (singleton semantics)



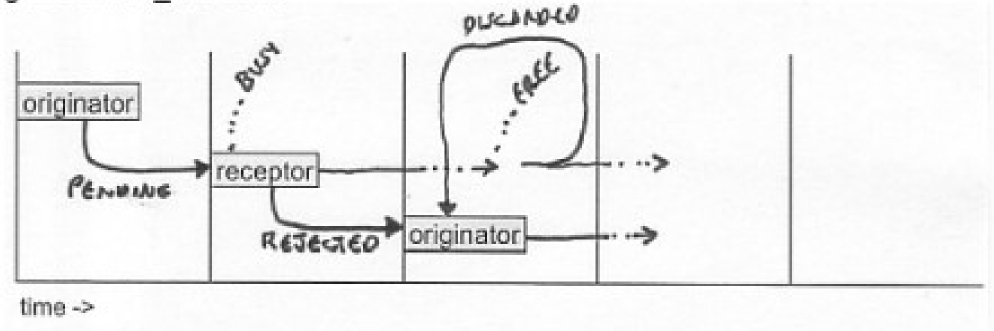
ACCEPTED_PROCEED (broadcast semantics)



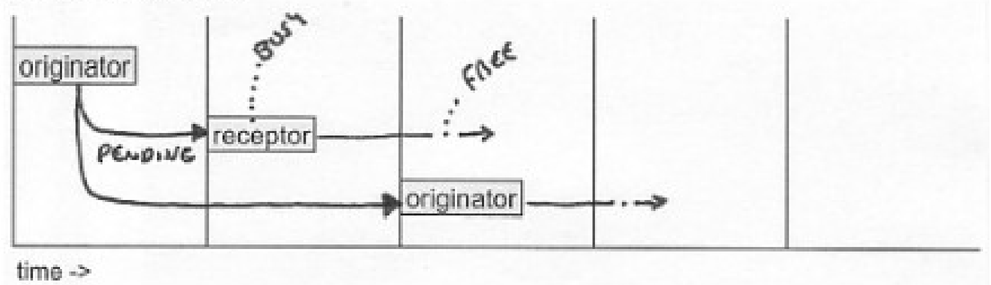
ACCEPTED_WAIT



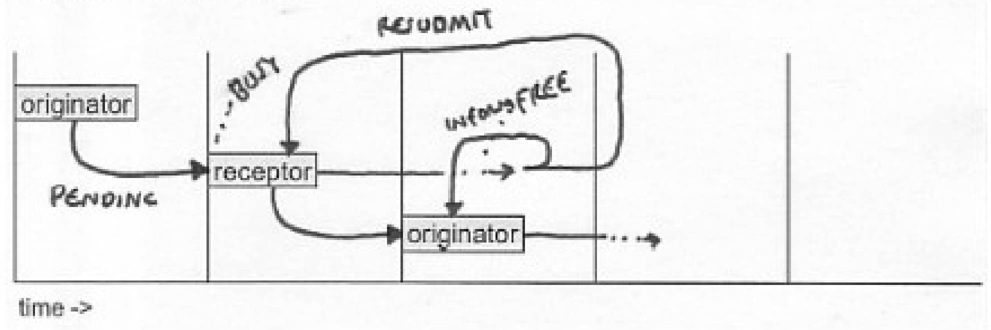
REJECTED using DISCARD_INFORM



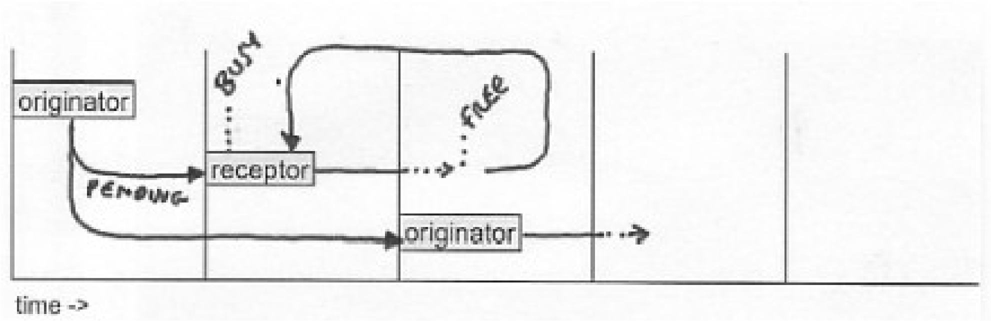
REJECTED using DISCARD_QUIET

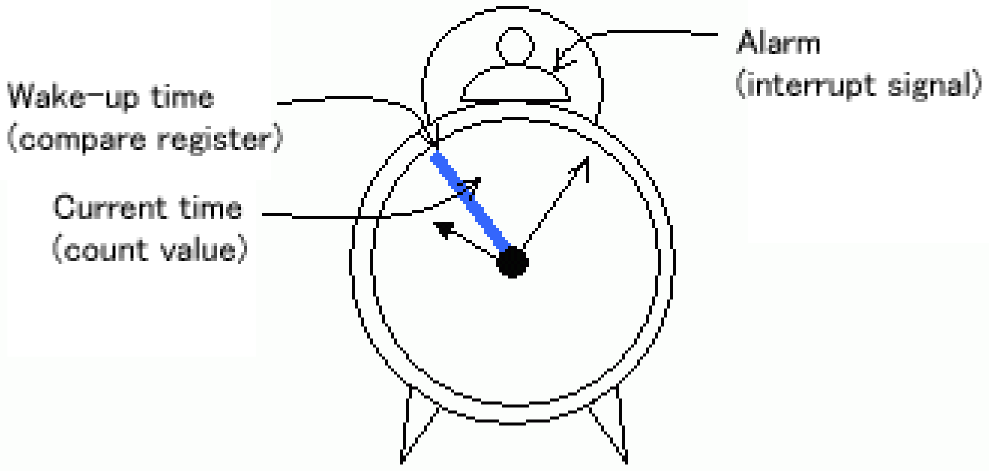


REJECTED using RESUBMIT_INFORM

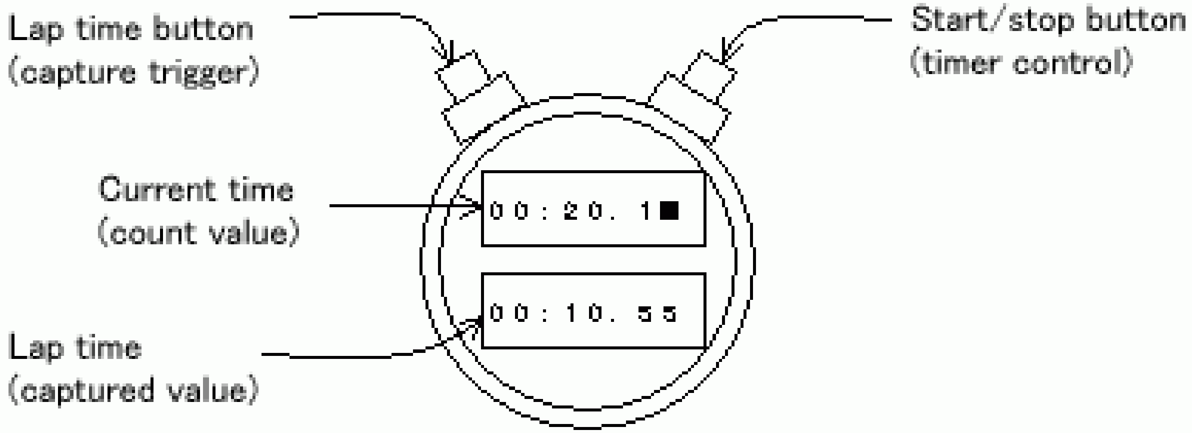


REJECTED using RESUBMIT_QUIET

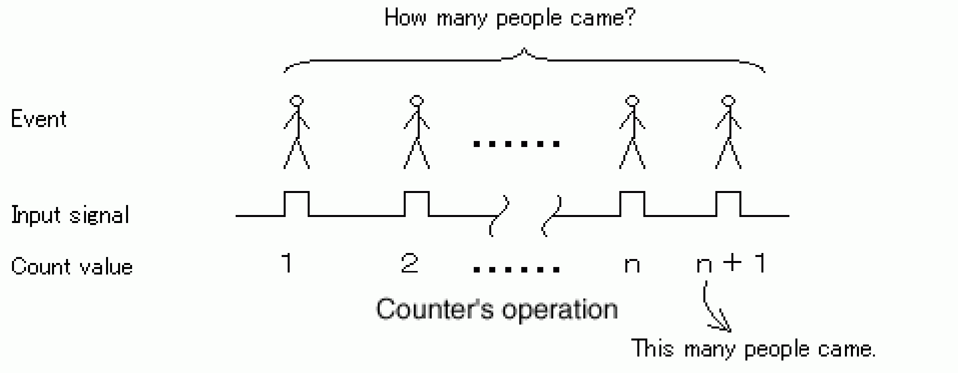




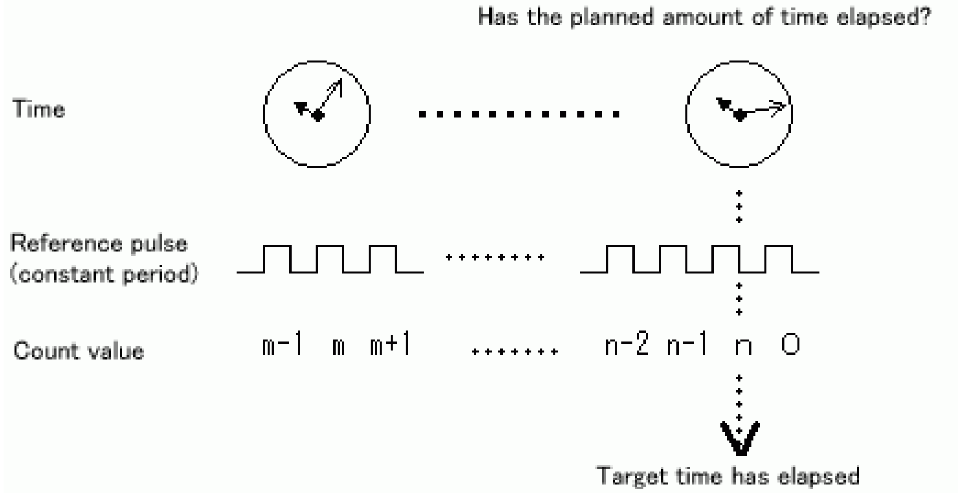
Alarm clock analogy for compare operation



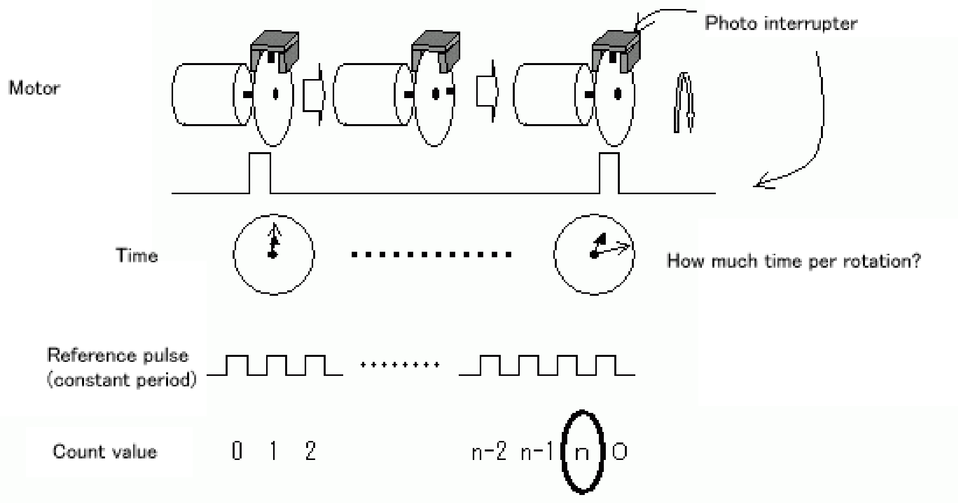
source: necel.com



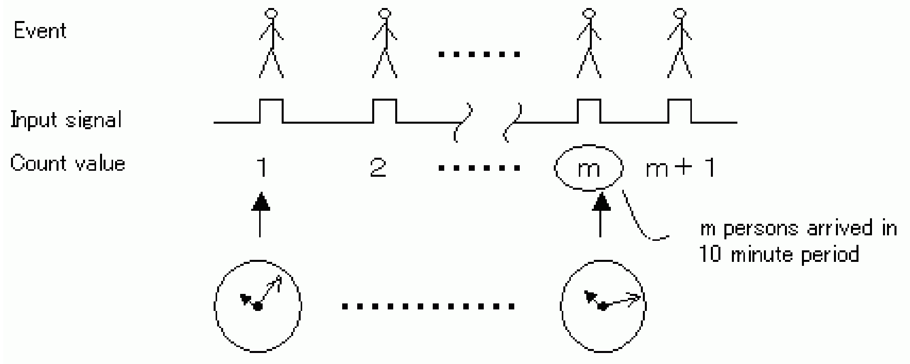
Counter's operation



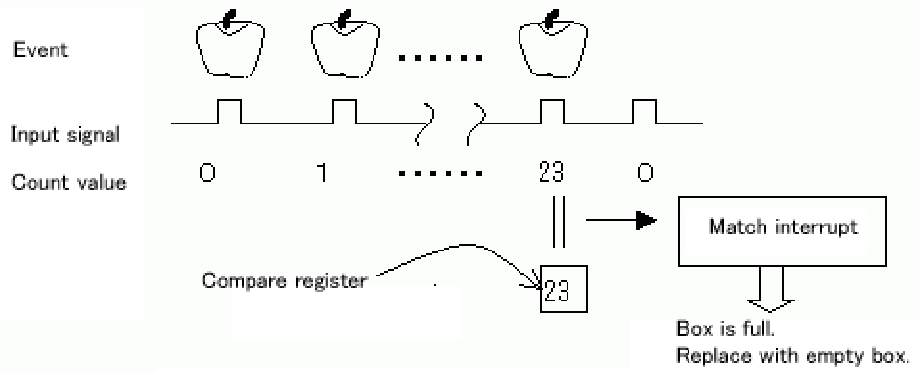
Timer's operation



Time interval measurement example

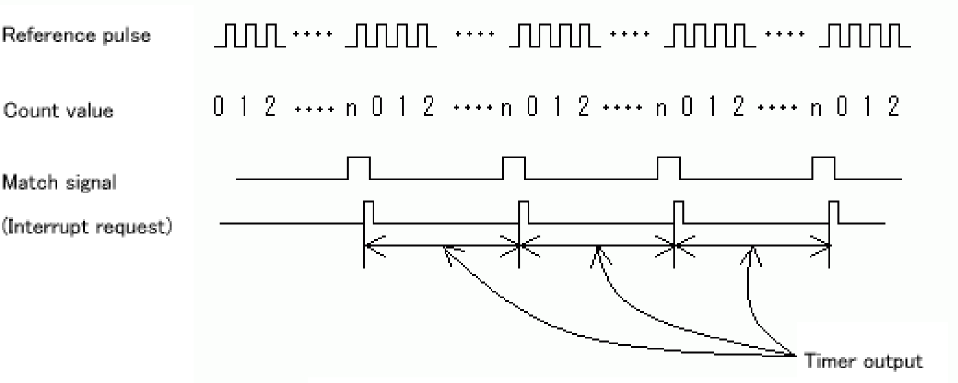


External event counter use example 1

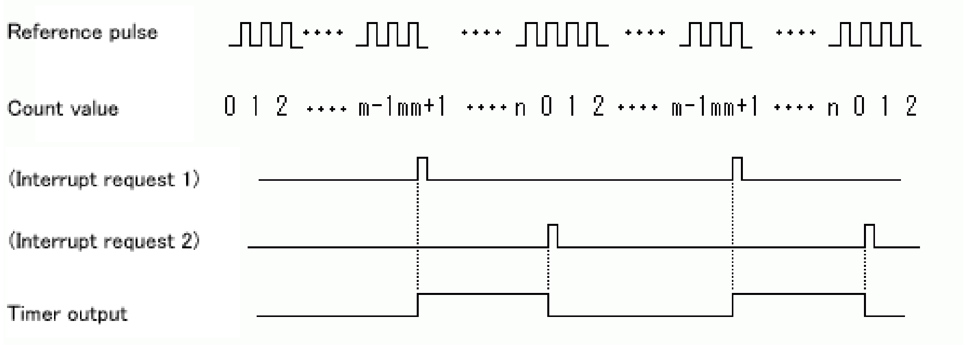


External event counter use example 2

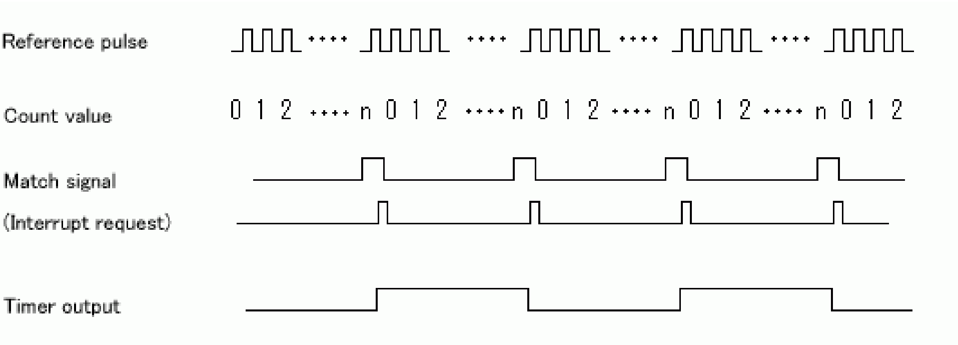
source: necel.com



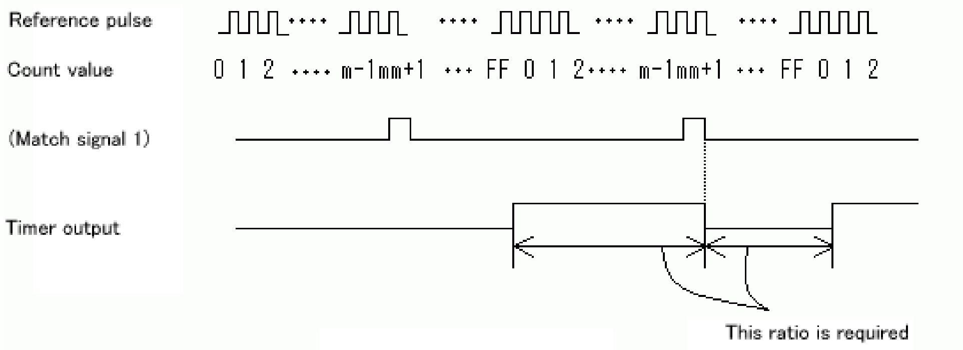
Pulse output operation example



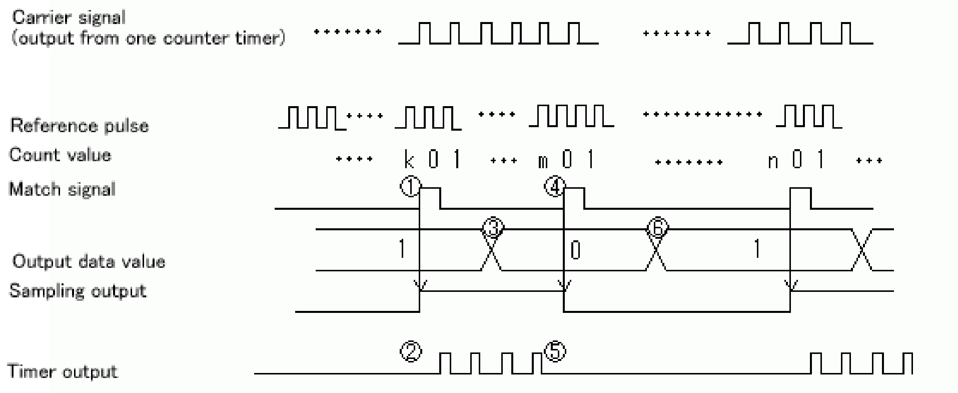
Pulse output operation example



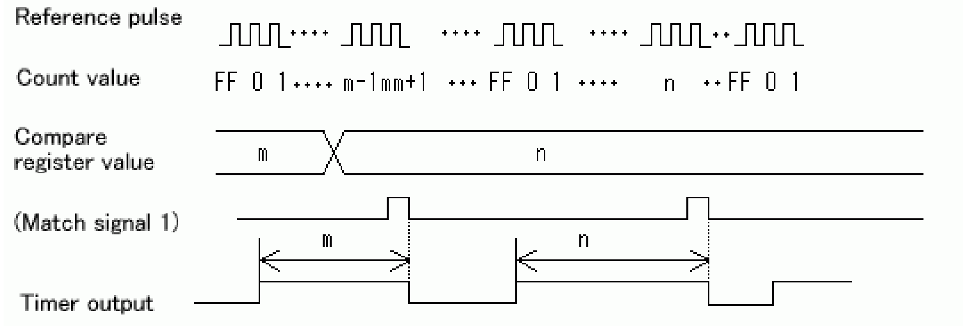
Square wave output operation example



PWM output operation example 1



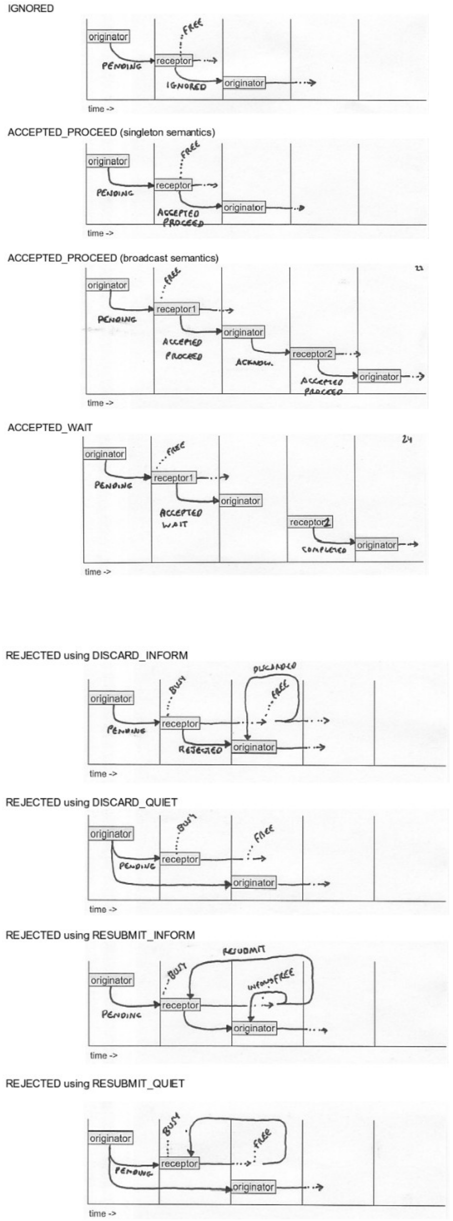
Carrier generation operation example

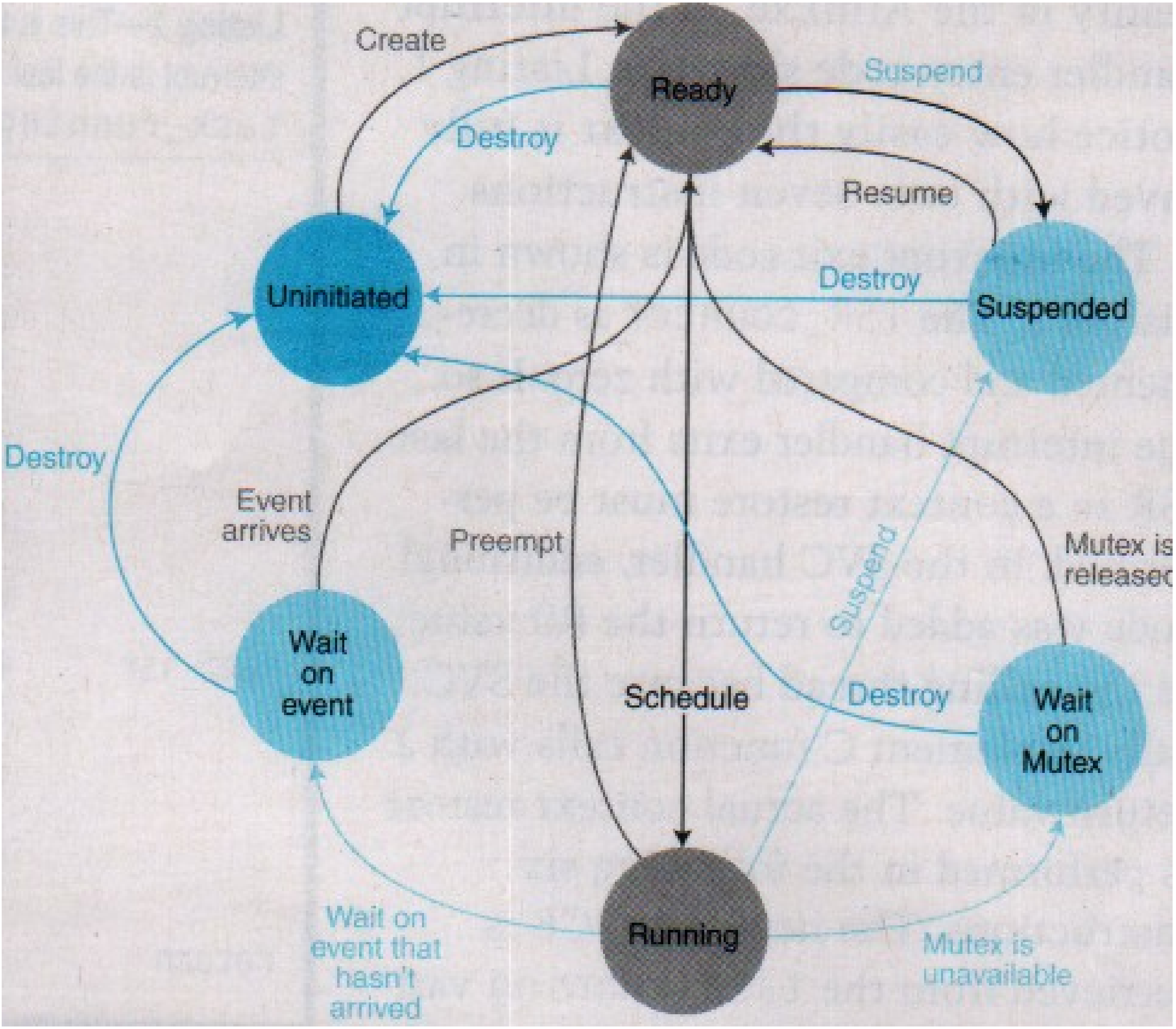


PWM output operation example 2

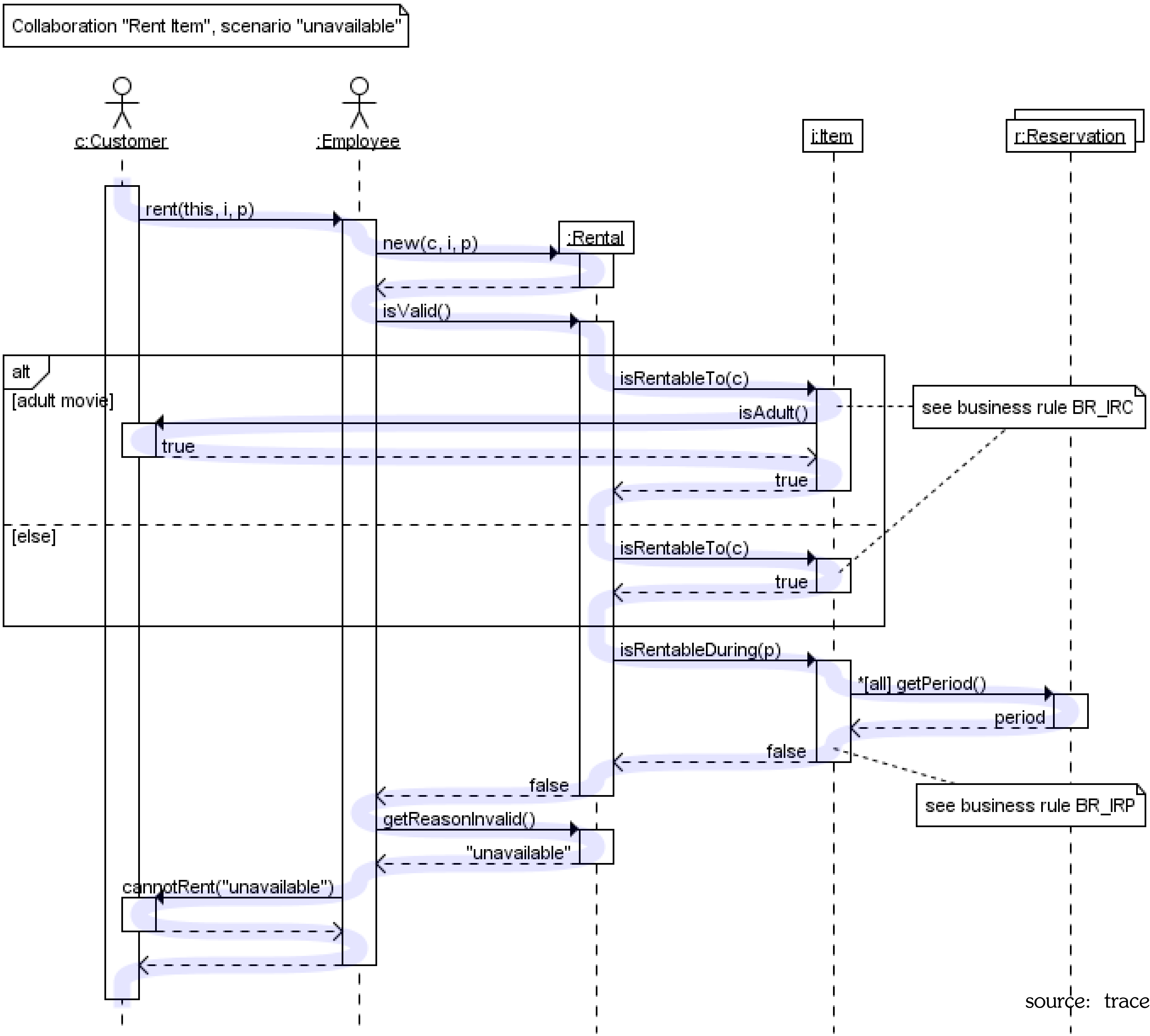
source: necel.com

Actual Example





source: Circuit Cellar, Sept. 08



source: tracemodeler.com

Observer Pattern

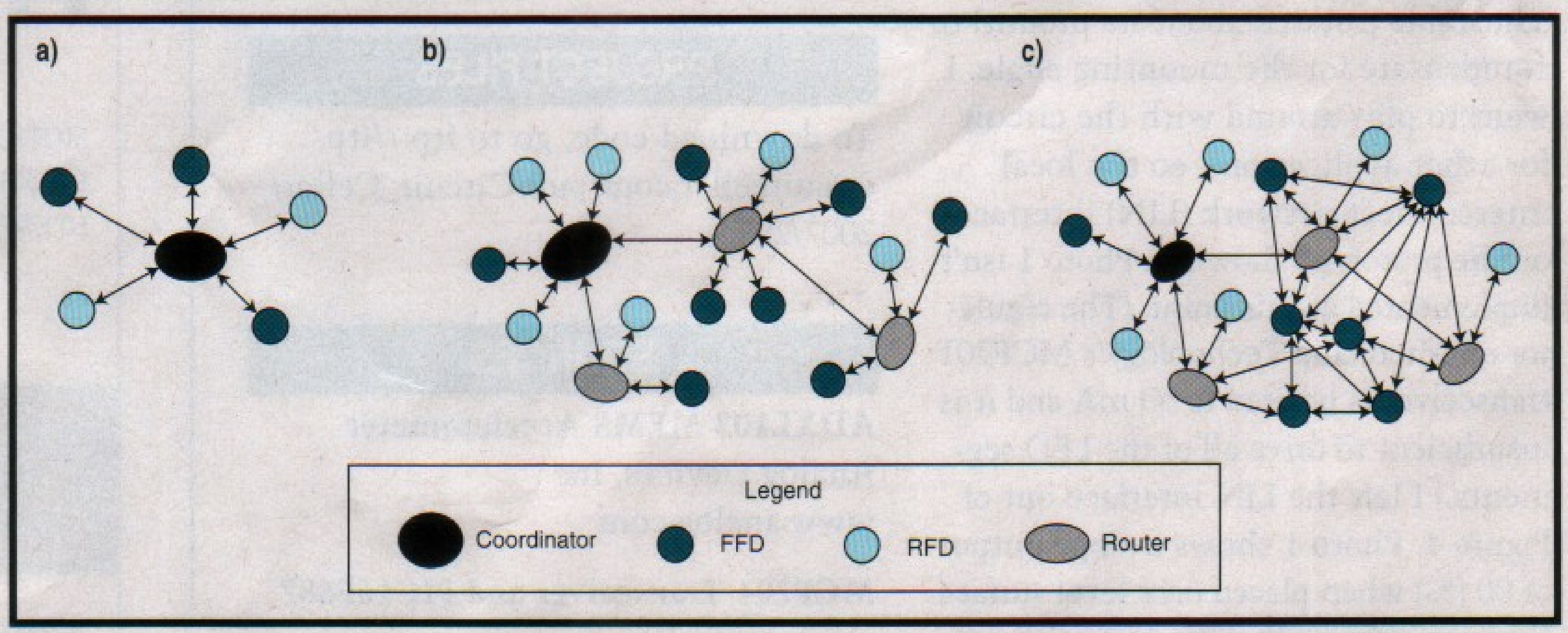
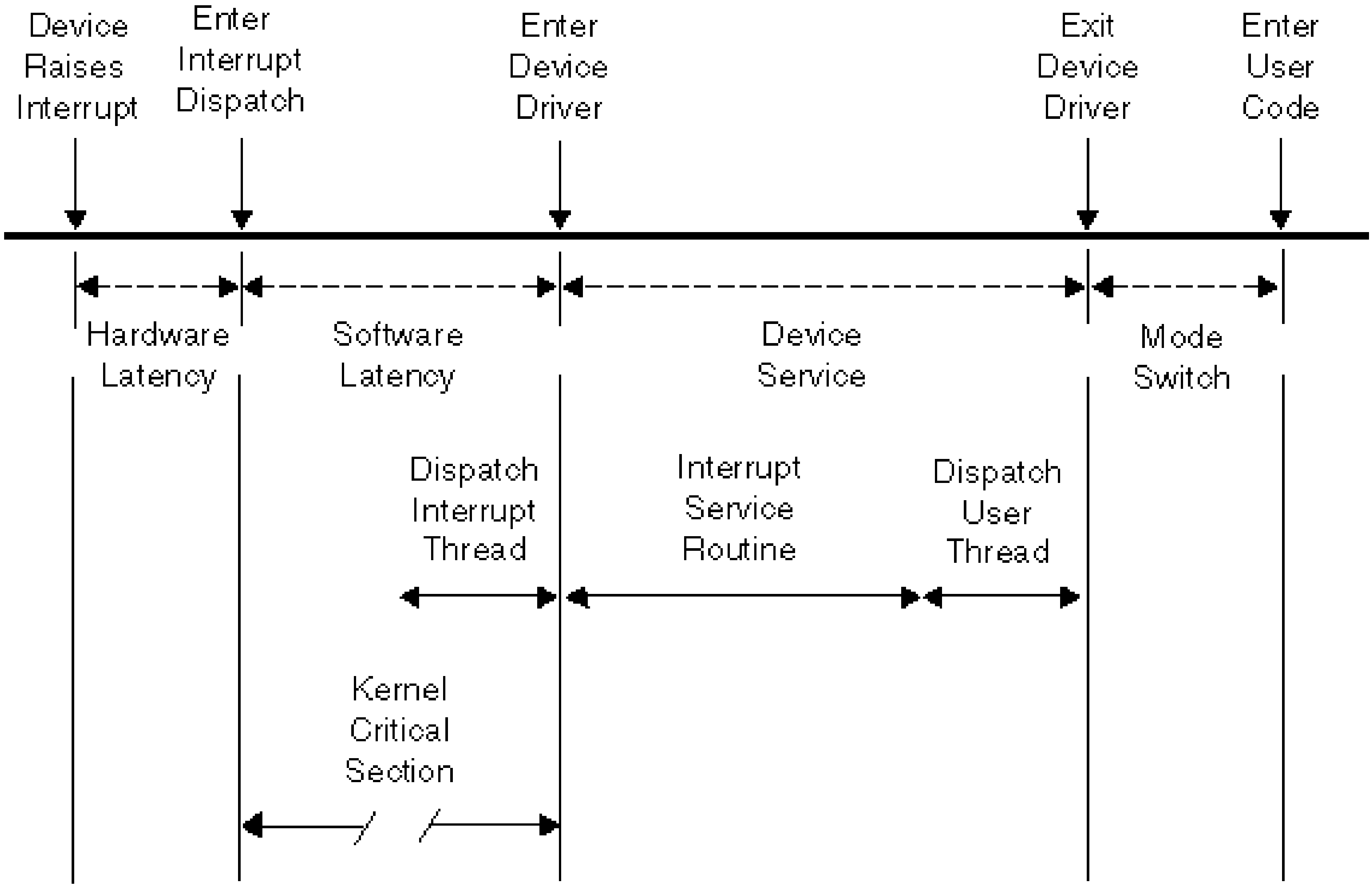


Figure 1—ZigBee has grown to support up to 64K nodes in a variety of topologies. The simplest is a star configuration **(a)** in which messages between devices are always passed via the coordinator. Star networks can be aggregated into a multihop cluster tree **(b)**, with the optional addition of routers to the mix. That brings us to the final option, mesh networks **(c)**, which support direct device-to-device links.

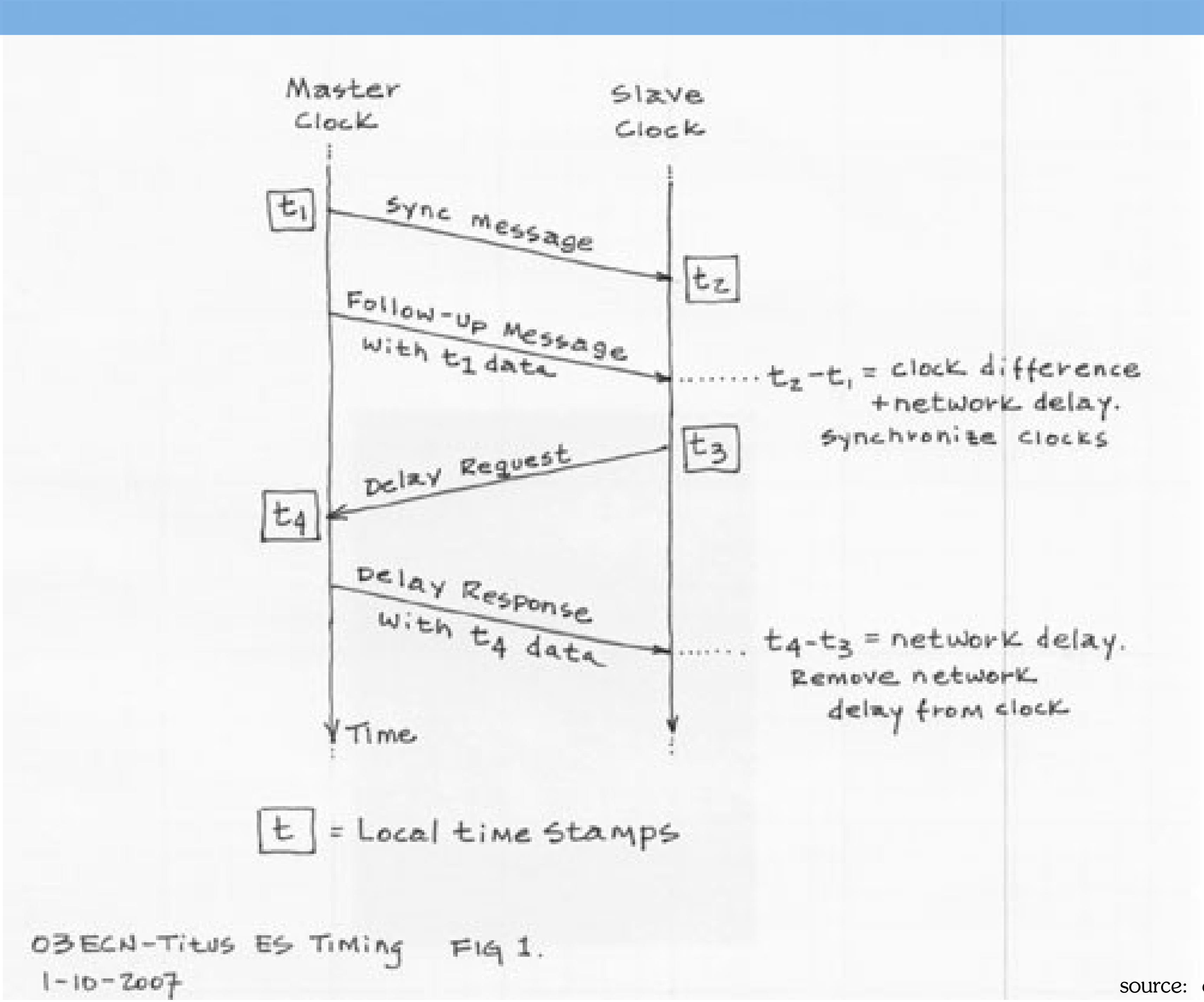
CIRCUIT CELLAR® www.circuitcellar.com

Observer Pattern



source: sgi.com

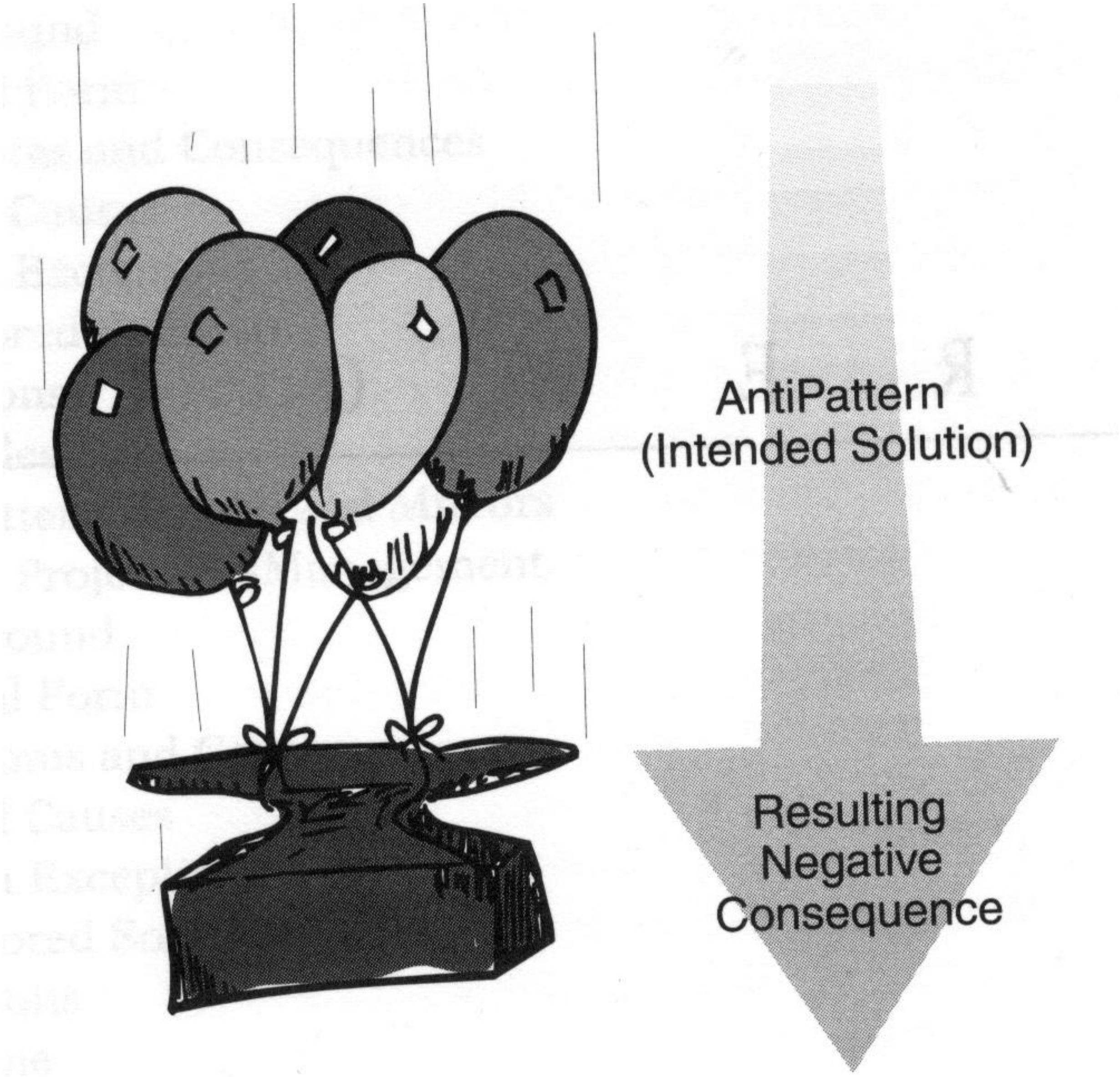
Observer Pattern


Master Clock
Slave Clock
t1
Sync Message
t2
Follow-Up Message with t1 data
t2-t1 = clock difference + network delay. synchronize clocks
t3
Delay Request
t4
Delay Response with t4 data
t4-t3 = network delay. Remove network delay from clock
Time
t = Local time Stamps
03ECN-Titus ES Timing Fig 1.
1-10-2007

source: ecnmag.com

SUPPLEMENTAL SLIDES…

Anti-Patterns



source: cs.ualberta.ca

Anti-Pattern : Organizational

- Accidental ownership: Employee is given a system that was tangentially related to their system and left to poorly maintain it without proper training, growth or focus (common among phone->network administrators in the late 90's)
- Analysis paralysis: Devoting disproportionate effort to the analysis phase of a project
- Captain in the engine room: The leader spends his time and attention on technical details, and nobody's running the ship
- Cash cow: A profitable legacy product that often leads to complacency about new products
- Continuous obsolescence: Devoting disproportionate effort to porting a system to new environments
- Cost migration: Transfer of project expenses to a vulnerable department or business partner
- Crisis mode (a.k.a firefighting mode): Dealing with things only when they become a crisis, with the result that everything becomes a crisis
- Design by committee: The result of having many contributors to a design, but no unifying vision
- Escalation of commitment: Failing to revoke a decision when it proves wrong
- Hero-mode: A policy of continuously relying on the heroic efforts of staff in order to meet impossible deadlines, whilst ignoring the long term cost of failing to build in software quality from the outset.
- I told you so: When the ignored warning of an expert proves justified, and this becomes the focus of attention

source: wikipedia.org

Anti-Pattern : Organizational

- Management by hope: Assuming that silence means everything is going well
- Management by neglect: Too much delegation
- Management by numbers: Paying excessive attention to quantitative management criteria, when these are non-essential or cost too much to acquire
- Management by perkele: Authoritarian style of management with no tolerance for dissent
- Management by wondering: Expecting a team to define their own objectives, and then wondering what they're doing
- Moral hazard: Insulating a decision-maker from the consequences of his or her decision.
- Mushroom management: Keeping employees uninformed and misinformed (kept in the dark and fed manure)
- Not invented here: Ignoring an idea or implementation originated outside the organization
- Polishing the polish: Giving a subordinate or team a finished project to work on, prohibiting them from doing anything else, and then complaining about their productivity
- Scope creep (along with the closely related terms complexity trap and featuritis): Allowing the scope of a project to grow without proper control
- Stovepipe: A structure that supports mostly up-down flow of data but inhibits cross organizational communication
- Vendor lock-in: Making a system excessively dependent on an externally supplied component

source: wikipedia.org

Anti-Pattern : Project Management

- Death march: Everyone knows that the project is going to be a disaster - except the CEO. However, the truth remains hidden and the project is artificially kept alive until the Day Zero finally comes ("Big Bang")
- Heel-dragging blindness: Project manager's blindness to heel dragging. Project workers (resources) tending to slow down their efforts to stretch out the project schedule because they have incentive to do so, for example if they get paid by time (not results) and there's no following project for them to seamlessly transition to
- Smoke and mirrors: Demonstrating how unimplemented functions will appear
- Software bloat: Allowing successive versions of a system to demand ever more resources
- Violin string organization: A highly tuned and trimmed organization with no flexibility

source: wikipedia.org

Anti-Pattern : Team Management

- Absentee manager: Any situation in which the manager is invisible for long periods of time.
- Cage match negotiator: When a manager uses a "victory at any cost" approach to management.
- Doppelganger: A manager or colleague who can be nice and easy to work with one moment, and then vicious and unreasonable the next.
- Fruitless hoops: The manager who requires endless (often meaningless) data before making a decision.
- Golden child: When special responsibility, opportunity, recognition, or reward is given to a team member based on personal relationships or contrary to the person's actual performance.
- Headless chicken: The manager who is always in a panic-stricken, fire-fighting mode.
- Leader not manager: The manager who is a good leader, but lacks in their administrative and managerial ability.
- Managerial cloning: The hiring and mentoring of managers to all act and work the same: identically to their bosses.
- Manager not leader: The manager who is proficient at their administrative and managerial duties, but lacks leadership ability.
- Metric abuse: The malicious or incompetent use of metrics and measurement.
- Mr. nice guy: The manager that strives to be everyone's friend.

source: wikipedia.org

Anti-Pattern : Team Management

- Ostrich: The worker who fills time doing inane work and ignoring work which needs to be done to meet deadlines assuming it will be ok. Often more concerned with looking productive but often ultimately ends up wasting, and hence running out of, time.
- Proletariat hero: The "everyman" worker who is held up as the ideal, but is really just a prop for management's increasing demands and lengthening production targets.
- Rising upstart: The potential stars who can't wait their time and want to forgo the requisite time to learn, mature and find their place.
- Seagull management: Flies in, makes a lot of noise, craps all over everything, then flies away.
- Spineless executive: The manager who does not have the courage to confront situations, take the heat for a failure, or protect their subordinates.
- Three-headed knight: The indecisive manager.
- Ultimate weapon: Phenomena that are relied upon so much by their peers or organization that they become the conduit for all things.
- Warm bodies: The worker who barely meets the minimum expectations of the job and is thusly shunted from project to project, or team to team.
- Yes man: The manager who will agree with everything the CEO says even though he has stated differently away from his presence.

source: wikipedia.org

Anti-Pattern : Analysis

- Napkin specification: The Functional/Technical specification is given to the Development team on a napkin (i.e., informally, and with insufficient detail) which is fundamentally equivalent to having no specification at all.
- Phony requirements: All requirements are communicated to the development teams in a rapid succession of netmeeting sessions or phone calls with no Functional/Technical specification or other supporting documentation.
- Retro-specification: To write Technical/Functional specification after project has already gone live.

source: wikipedia.org

Anti-Pattern : General Design

- Abstraction inversion: Not exposing implemented functionality required by users, so that they re-implement it using higher level functions
- Ambiguous viewpoint: Presenting a model (usually OOAD) without specifying its viewpoint
- Big ball of mud: A system with no recognizable structure
- Blob: see God object
- Gas factory: An unnecessarily complex design
- Input kludge: Failing to specify and implement handling of possibly invalid input
- Interface bloat: Making an interface so powerful that it is extremely difficult to implement
- Magic pushbutton: Coding implementation logic directly within interface code, without using abstraction.
- Race hazard: Failing to see the consequence of different orders of events
- Railroaded solution: A proposed solution that while poor, is the only one available due to poor foresight and inflexibility in other areas of the design
- Re-coupling: Introducing unnecessary object dependency
- Stovepipe system: A barely maintainable assemblage of ill-related components
- Staralised schema: A database schema containing dual purpose tables for normalised and datamart use

source: wikipedia.org

Anti-Pattern : Object-Oriented Design

- Anemic Domain Model: The use of domain model without any business logic which is not OOP because each object should have both attributes and behaviors
- BaseBean: Inheriting functionality from a utility class rather than delegating to it
- Call super: Requiring subclasses to call a superclass's overridden method
- Circle-ellipse problem: Subtyping variable-types on the basis of value-subtypes
- Empty subclass failure: Creating a class that fails the "Empty Subclass Test" by behaving differently from a class derived from it without modifications
- God object: Concentrating too many functions in a single part of the design (class)
- Object cesspool: Reusing objects whose state does not conform to the (possibly implicit) contract for re-use
- Object orgy: Failing to properly encapsulate objects permitting unrestricted access to their internals
- Poltergeists: Objects whose sole purpose is to pass information to another object
- Sequential coupling: A class that requires its methods to be called in a particular order
- Singletonitis: The overuse of the singleton pattern
- Yet Another F*cking Layer: Adding unnecessary layers to a program, library or framework. This became popular after the first book on programming patterns.
- Yo-yo problem: A structure (e.g., of inheritance) that is hard to understand due to excessive fragmentation

source: wikipedia.org

Anti-Pattern : Programming

- Accidental complexity: Introducing unnecessary complexity into a solution
- Accumulate and fire: Setting parameters for subroutines in a collection of global variables
- Action at a distance: Unexpected interaction between widely separated parts of a system
- Blind faith: Lack of checking of (a) the correctness of a bug fix or (b) the result of a subroutine
- Boat anchor: Retaining a part of a system that no longer has any use
- Bug magnet: A block of code so infrequently invoked/tested that it will most likely fail, or any error-prone construct or practice.
- Busy spin: Consuming CPU while waiting for something to happen, usually by repeated checking instead of proper messaging
- Caching failure: Forgetting to reset an error flag when an error has been corrected
- Cargo cult programming: Using patterns and methods without understanding why
- Checking type instead of interface: Checking that an object has a specific type when only a certain contract is required. May cause Empty subclass failure.
- Code momentum: Over-constraining part of a system by repeatedly assuming things about it in other parts
- Coding by exception: Adding new code to handle each special case as it is recognized
- Error hiding: Catching an error message before it can be shown to the user and either showing nothing or showing a meaningless message

source: wikipedia.org

Anti-Pattern : Programming

- Expection handling: (From Exception + Expect) Using a language's error handling system to implement normal program logic
- Hard code: Embedding assumptions about the environment of a system at many points in its implementation
- Lava flow: Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences
- Loop-switch sequence: Encoding a set of sequential steps using a loop over a switch statement
- Magic numbers: Including unexplained numbers in algorithms
- Magic strings: Including literal strings in code, for comparisons, as event types etc.
- Monkey work: Term for any general repeated support code required within projects which suffer from poor code reuse and design. Is often avoided and rushed, hence open to errors and can quickly become a Bug magnet.
- Packratting: Consuming excessive memory by keeping dynamically allocated objects alive for longer than they are needed
- Parallel protectionism: When code becomes so complex and fragile that it is easier to clone a parallel infrastructure than to add a trivial attribute to existing infrastructure
- Programming by accident: Trying to solve problems by trial and error, sometimes because of poor documentation or not thinking things out first.
- Ravioli code: Systems with lots of objects that are loosely connected

source: wikipedia.org

Anti-Pattern : Programming

- Soft code: Storing business logic in configuration files rather than source code
- Spaghetti code: Systems whose structure is barely comprehensible, especially because of misuse of code structures
- Wrapping wool in cotton: Commonly observed when framework methods are contained in single line methods in wrapper classes which provide no useful abstraction

source: wikipedia.org

Anti-Pattern : Methodological

- Copy and paste programming: Copying (and modifying) existing code rather than creating generic solutions
- De-factoring: The process of removing functionality and replacing it with documentation
- Golden hammer: Assuming that a favorite solution is universally applicable (See: Silver Bullet)
- Improbability factor: Assuming that it is improbable that a known error will occur
- Low hanging fruit: Dealing with easier issues first while ignoring larger more complex issues. It can be thought of as to the relative ease with which scientific, philosophical and technological discoveries are made at first compared with the difficulties once the subject has already been explored.[clarify]
- Not built here : See: Reinventing the wheel, Not invented here.
- Premature optimization: Coding early-on for perceived efficiency, sacrificing good design, maintainability, and sometimes even real-world efficiency
- Programming by permutation: Trying to approach a solution by successively modifying the code to see if it works
- Reinventing the square wheel: Creating a poor solution when a good one exists
- Reinventing the wheel: Failing to adopt an existing, adequate solution
- Silver bullet: Assuming that a favorite technical solution can solve a larger process or problem
- Tester driven development: Software projects where new requirements are specified in bug reports

source: wikipedia.org

Anti-Pattern : Testing

- Hostile testing: Antagonizing practical development solutions and workflow with over-testing procedures or over-scheduling test runs
- Meta-testing: Overdesigning testing procedures until it is necessary to test them, also known as "watchmen's watchmen"
- Moving target: Continuously change design and/or implementation in order to escape established testing procedures
- Slave testing: Conditioning testers by means of black-mail or corruption to satisfy the stakeholders urge to flag positive compliance

source: wikipedia.org

Anti-Pattern : Configuration Management

- Dependency hell: Problems with versions of required products
  - Classpath hell: Problems related to specifying libraries, their dependencies and their required version to run the application
  - Extension conflict: Problems with different extensions to pre-Mac OS X versions of the Mac OS attempting to patch the same parts of the operating system
  - DLL hell: Problems with versions, availability and multiplication of DLLs, specifically on Microsoft Windows
  - JAR hell: Problems with different versions or locations of JAR files, usually caused by a lack of understanding of the class loading model

source: wikipedia.org

Code Smell

How to Write Unmaintainable Code

- General
  - use different names for same thing
  - use same name for different things
  - use vague semantic abstractions: `doX`, `performX`, `handleX`, `processX`
  - misuse thesaurus
  - use multilingual variable names, including Latinate plurals and accents
  - use extended ASCII
  - use single-letter variable names
  - use paragraph variable names
  - use long variable names with similar spellings
  - make creative missspellings
  - go crazy with A.C.R.O.N.Y.M.S.
  - use sick abbreviations
  - misuse capitalization

How to Write Unmaintainable Code

- General, continued
  - reuse class names as member names and functions
  - violate language conventions
  - misapply `l` and `1`, `O` and `0`, etc.
  - recycle variables across multiple contexts
  - demonstrate lack of consistency and attention to detail (ATD)
  - use misleading names
  - inject side effects
  - have fun with Hungarian notation
  - make commented-out code look active
  - mismatch component variable names with their GUI labels
  - use global variables

How to Write Unmaintainable Code

- Documentation
  - include incorrect or stale comments
  - comment/document the obvious
  - be promiscuous with documentation templates
  - go overboard with *n*-level design documents, where *n*>>1
  - fail to mention gotchas
  - do not comment variables with range, units, assumptions, etc.

How to Write Unmaintainable Code

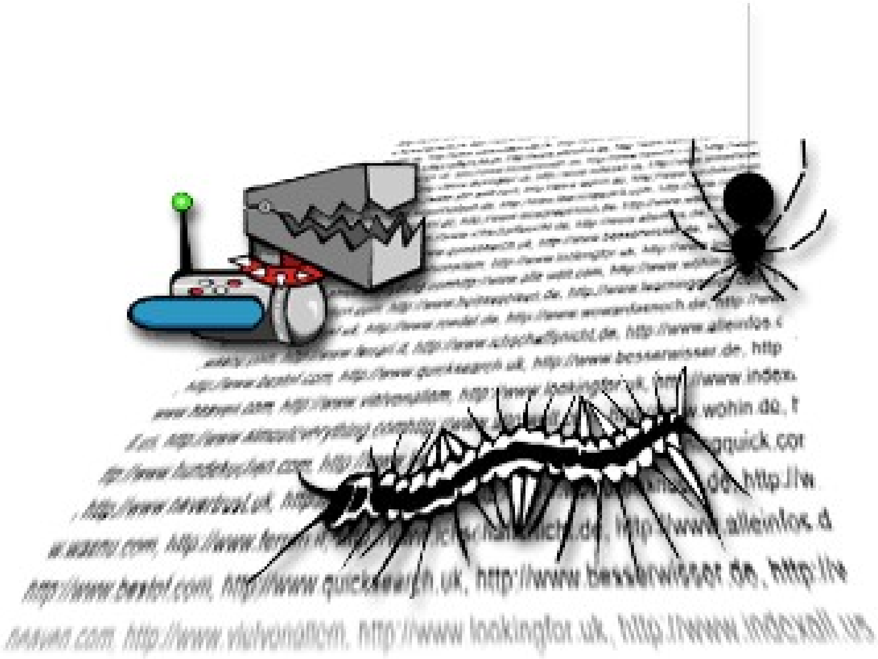
- Program design
  - specify each fact in multiple places
  - never validate
  - never assert
  - use copy/paste instead of reusable code
  - screw Ockam's Razor
  - use static arrays to waste space
  - declare everything public because someone might want to use it
  - permute order of formal parameters: `draw(x,y)` then `draw(y,x)`
  - pack rat obsolete code forever "just in case"
  - bloat classes beyond cohesion and encapsulation limits
  - put useless info in About box; omit version info
  - carry only a hammer because that way, everything looks like a nail
  - avoid libraries

How to Write Unmaintainable Code

- Program design, continued
  - flaunt standards
  - combine bug fixes with upgrades
  - change file formats with each release
  - ignore backwards compatibility
  - avoid KISS principle

How to Write Unmaintainable Code

- Testing
  - never test
  - never do performance analysis
  - test only in debug mode



How to Write Unmaintainable Code

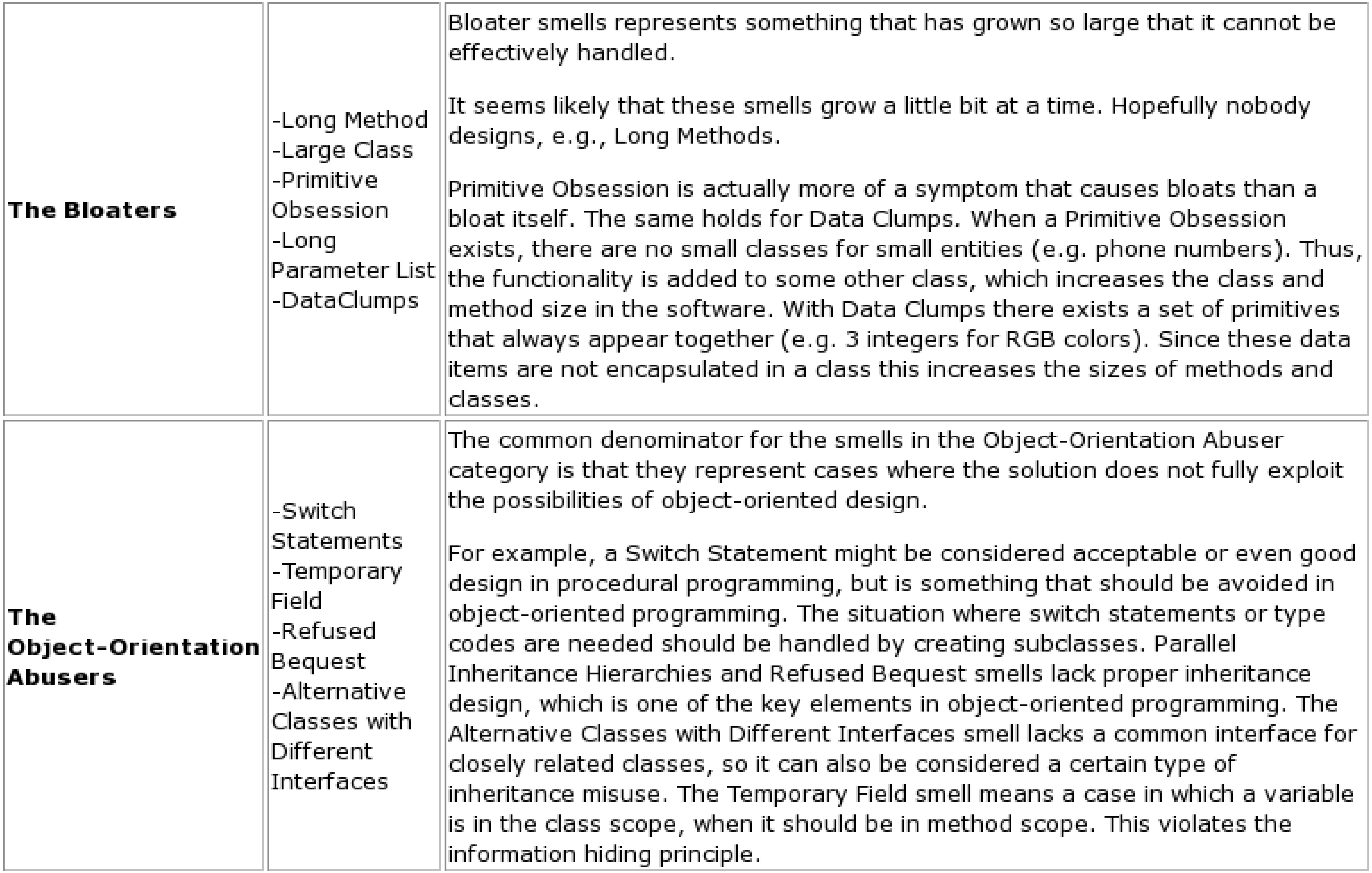
- Teamwork
  - appease thy boss with old-school (read: obsolete) techniques
  - subvert help desk by blaming someone else
  - fail to mention eventual doom from shortsighted decisions: Y2K

Code Smell

- Large method - a method, function, or procedure that has grown too large.
- Large class - a class that has grown too large, see God object.
- Feature envy - a class that uses methods of another class excessively.
- Inappropriate intimacy - a class that has dependencies on implementation details of another class.
- Refused bequest - a class that overrides a method of a base class in such a way that the contract of the base class is not honored by derived class. See Liskov substitution principle.
- Lazy class - a class that does too little.
- Duplicated method - a method, function, or procedure that is very similar to another.
- Contrived Complexity - forced usage of overly complicated design patterns where simpler design would suffice.

source: wikipedia.org

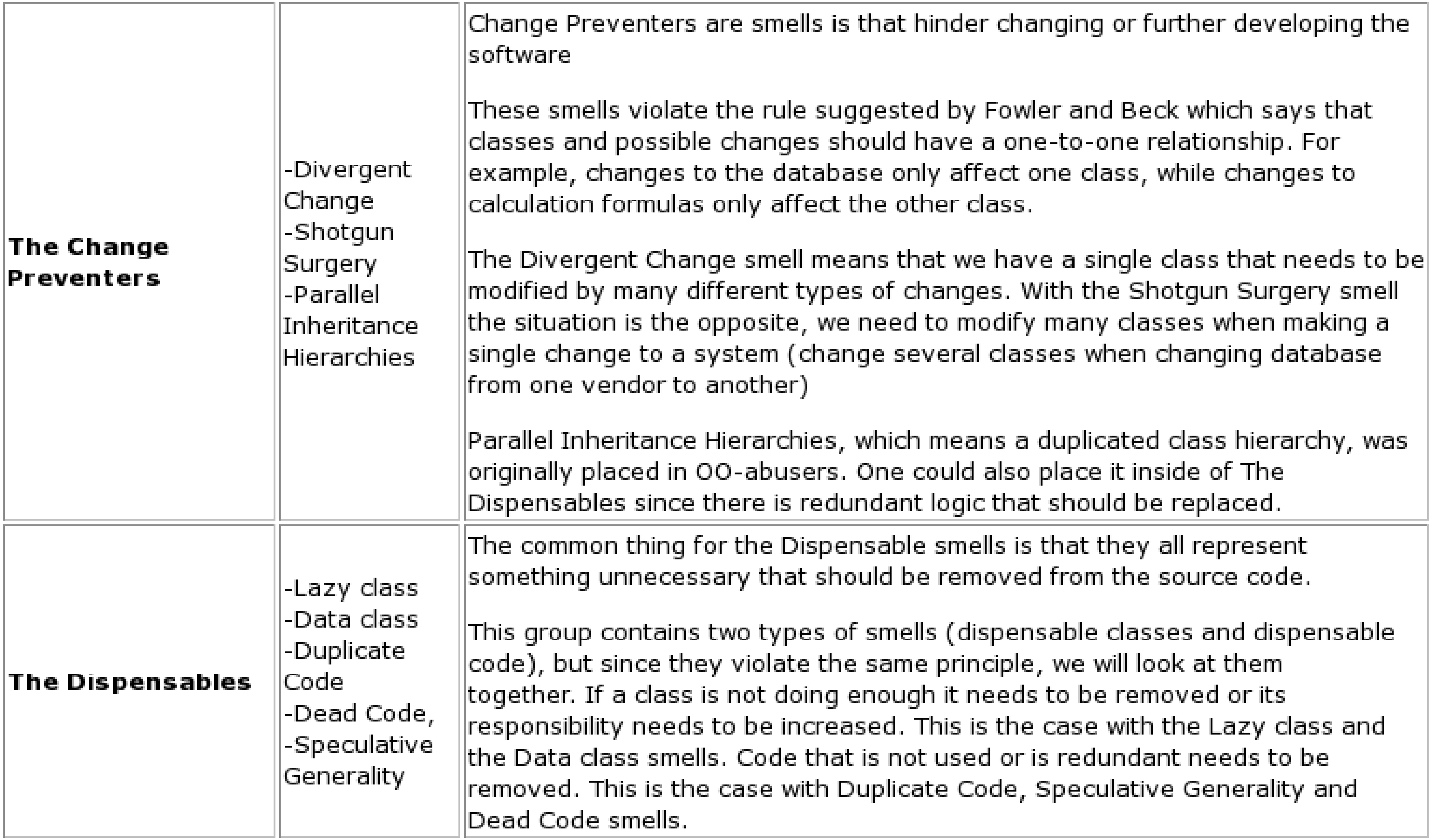
Code Smell : Taxonomy



The Bloaters	-Long Method -Large Class -Primitive Obsession -Long Parameter List -DataClumps	Bloater smells represents something that has grown so large that it cannot be effectively handled. It seems likely that these smells grow a little bit at a time. Hopefully nobody designs, e.g., Long Methods. Primitive Obsession is actually more of a symptom that causes bloats than a bloat itself. The same holds for Data Clumps. When a Primitive Obsession exists, there are no small classes for small entities (e.g. phone numbers). Thus, the functionality is added to some other class, which increases the class and method size in the software. With Data Clumps there exists a set of primitives that always appear together (e.g. 3 integers for RGB colors). Since these data items are not encapsulated in a class this increases the sizes of methods and classes.
The Object-Orientation Abusers	-Switch Statements -Temporary Field -Refused Bequest -Alternative Classes with Different Interfaces	The common denominator for the smells in the Object-Orientation Abuser category is that they represent cases where the solution does not fully exploit the possibilities of object-oriented design. For example, a Switch Statement might be considered acceptable or even good design in procedural programming, but is something that should be avoided in object-oriented programming. The situation where switch statements or type codes are needed should be handled by creating subclasses. Parallel Inheritance Hierarchies and Refused Bequest smells lack proper inheritance design, which is one of the key elements in object-oriented programming. The Alternative Classes with Different Interfaces smell lacks a common interface for closely related classes, so it can also be considered a certain type of inheritance misuse. The Temporary Field smell means a case in which a variable is in the class scope, when it should be in method scope. This violates the information hiding principle.

source: wikipedia.org

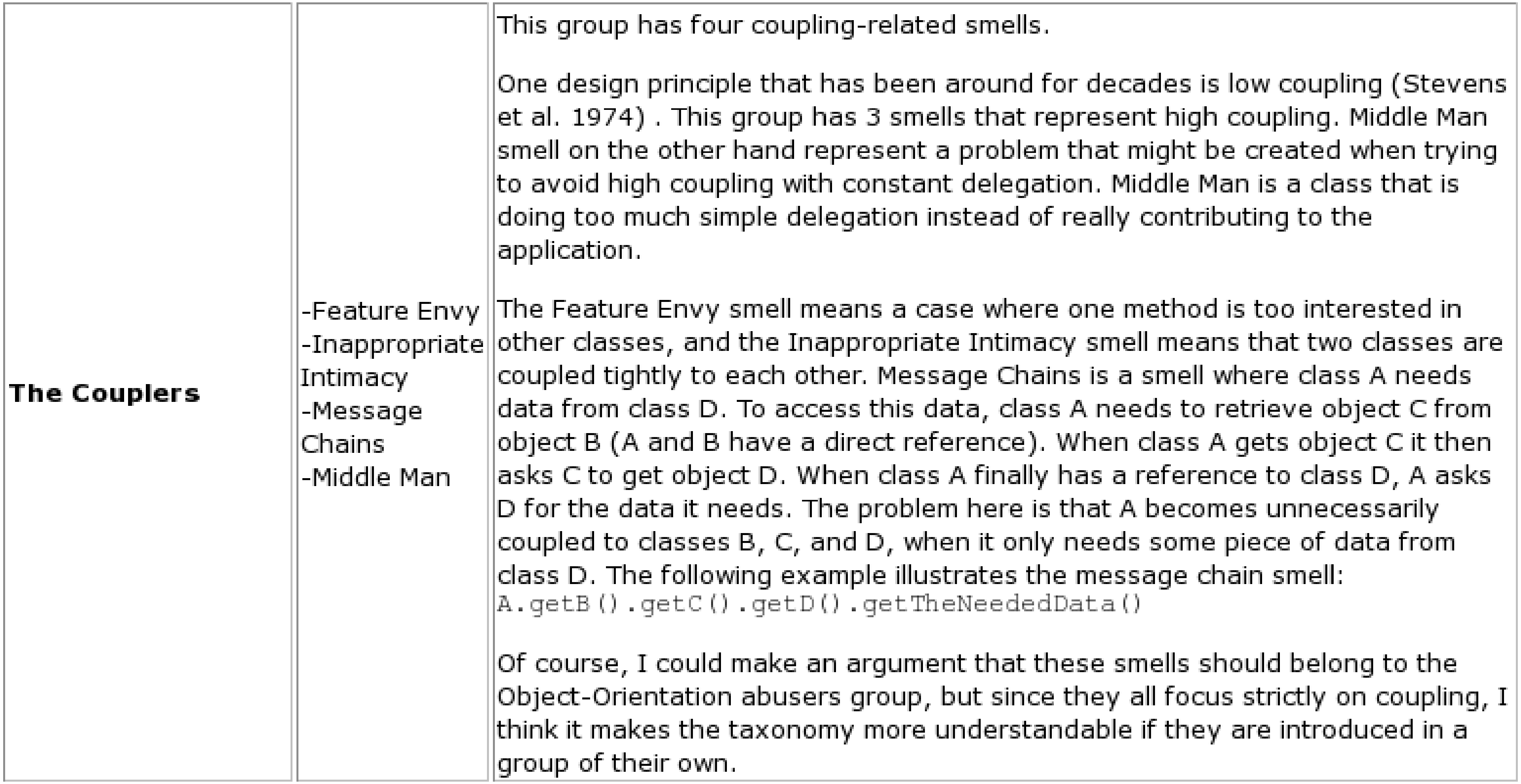
Code Smell : Taxonomy



The Change Preventers	-Divergent Change -Shotgun Surgery -Parallel Inheritance Hierarchies	Change Preventers are smells is that hinder changing or further developing the software These smells violate the rule suggested by Fowler and Beck which says that classes and possible changes should have a one-to-one relationship. For example, changes to the database only affect one class, while changes to calculation formulas only affect the other class. The Divergent Change smell means that we have a single class that needs to be modified by many different types of changes. With the Shotgun Surgery smell the situation is the opposite, we need to modify many classes when making a single change to a system (change several classes when changing database from one vendor to another) Parallel Inheritance Hierarchies, which means a duplicated class hierarchy, was originally placed in OO-abusers. One could also place it inside of The Dispensables since there is redundant logic that should be replaced.
The Dispensables	-Lazy class -Data class -Duplicate Code -Dead Code, -Speculative Generality	The common thing for the Dispensable smells is that they all represent something unnecessary that should be removed from the source code. This group contains two types of smells (dispensable classes and dispensable code), but since they violate the same principle, we will look at them together. If a class is not doing enough it needs to be removed or its responsibility needs to be increased. This is the case with the Lazy class and the Data class smells. Code that is not used or is redundant needs to be removed. This is the case with Duplicate Code, Speculative Generality and Dead Code smells.

source: wikipedia.org

Code Smell : Taxonomy



The Couplers	-Feature Envy -Inappropriate Intimacy -Message Chains -Middle Man	This group has four coupling-related smells. One design principle that has been around for decades is low coupling (Stevens et al. 1974) . This group has 3 smells that represent high coupling. Middle Man smell on the other hand represent a problem that might be created when trying to avoid high coupling with constant delegation. Middle Man is a class that is doing too much simple delegation instead of really contributing to the application. The Feature Envy smell means a case where one method is too interested in other classes, and the Inappropriate Intimacy smell means that two classes are coupled tightly to each other. Message Chains is a smell where class A needs data from class D. To access this data, class A needs to retrieve object C from object B (A and B have a direct reference). When class A gets object C it then asks C to get object D. When class A finally has a reference to class D, A asks D for the data it needs. The problem here is that A becomes unnecessarily coupled to classes B, C, and D, when it only needs some piece of data from class D. The following example illustrates the message chain smell: `A.getB().getC().getD().getTheNeededData()` Of course, I could make an argument that these smells should belong to the Object-Orientation abusers group, but since they all focus strictly on coupling, I think it makes the taxonomy more understandable if they are introduced in a group of their own.

source: wikipedia.org