Ingeniería de Software II

Ingeniería de Software II

Segundo Cuatrimestre de 2011

Clase 17: Arquitecturas: Repaso, más tácticas para atributos de calidad

Buenos Aires, 24 de Octubre de 2011

Repaso de Definiciones y Conceptos

- Arquitectura
  - La arquitectura de software de un sistema de computación es la estructura o estructuras del sistema, que abarcan elementos de software, las propiedades externamente visibles de estos elementos y las relaciones entre ellos.
  - La arquitectura de un sistema de software es el conjunto de "decisiones principales de diseño".
- Arquitectura vs. diseño
  - No hay visión única. La más aceptada: la arquitectura es una parte del diseño
  - Diseño (Oxford Dictionary): "To form a plan or scheme of, to arrange or conceive in the mind… for later execution"
  - Hay decisiones de diseño "no arquitectónicas". Y además, lo que no es arquitectónico ahora, puede ser lo más tarde (depende de la visibilidad de los módulos)

Más definiciones

- Estilo arquitectónico
  - Una descripción (“de muy alto nivel y sin hablar de funcionalidad específica”) de tipos de relaciones y elementos, junto con restricciones sobre cómo deben usarse (ej. “client server”, “pipes and filters”).
- Patrón arquitectónico:
  - Para algunos, lo mismo que un estilo
  - Para otros, similar a un estilo pero con más detalle. Ejemplo, “Layered” vs. “Three tiered”.
- Arquitectura de referencia
  - Una división común de funcionalidad mapeada a elementos que cooperativamente implementan esa funcionalidad y flujos de datos entre ellos.
  - Ejemplo: uso de sensors y actuators en aplicaciones de robótica
  - DSSA: un paso más en ese camino

Más definiciones

- Viewtype
  - Tipo de descripción de una arquitectura orientada a una estructura en particular

- Viewtypes existentes
  - Módulos
  - Componentes y conectores
  - Alocación (Deployment)

- La palabra módulo se refiere a unidades en tiempo de diseño
- La palabra componente se refiere a unidades en tiempo de ejecución
- La palabra "elemento" de arquitectura agrupa a las demás
- Concern es un requerimiento que afecta la arquitectura. Un término muy general que aplica a los subtipos de atributos de calidad. Ejemplos: tiempo de respuesta, latencia, facilidad para testear un cambio.

Algo más sobre tácticas – Disponibilidad y tolerancia a fallas

- Todo sistema de hardware o software tiene fallas
- Se dice que un sistema es tolerante a fallas si puede seguir operando en presencia de fallas
- Uno de los puntos a lograr es que no tenga SPOF (Single Point of Failure), punto de falla que hace que todo el sistema deje de funcionar
- La mayoría de los SPOF se pueden solucionar con redundancia

Clasificación de fallas

Tipo de Falla	Descripción
Crash failure	El sistema funciona perfectamente hasta que ocurre un error inesperado. (Kernel panic, pantalla azul, etc)
Omission failure *Receive or Send*	El sistema falla recibiendo requerimientos, enviando o recibiendo mensajes.
Timing failure	El sistema responde fuera de los tiempos esperados
Response failure *Value failure* *State transition failure*	Respuesta del sistema incorrecta o se desvía del flujo de control correcto.

Disponibilidad

- Disponibilidad: Es el porcentaje del tiempo en que un sistema está disponible para realizar las funciones para las que fue diseñado.
- Disponibilidad = Tiempo Operativo / Tiempo Operativo + tiempo no operativo forzado o no forzado

- Se suele contar en cantidad de “nueves” y el periodo de tiempo suele ser un año. Por ejemplo un proveedor de enlace punto a punto nos puede decir que su SLA (Service Level Agreement) en disponibilidad es de 99,99% anual (4 nueves)

¿ Esto es mucho o es poco ?

Veamos la siguiente tabla

Ejemplo de Disponibilidad

SLA Disponibilidad	7x24 8760 horas al año	7x8 2920 horas al año
90%	876 horas (36,5 días)	292 horas (12,16 días)
95%	438 horas (18,25 días)	146 horas (6,07 días)
99%	87,6 horas (3,65 días)	29,2 horas (1,21 días)
99,9%	8,76 horas	2,92 horas
99,99%	52,56 minutos	17,47 minutos
99,999%	5,256 minutos	1,747 minutos
99,9999%	31,536 segundos	10,483 segundos

Tolerancia a fallos – confiabilidad en Serie

La Confiabilidad de un conjunto de equipos en serie es el producto de la confiabilidad de cada uno de los equipos.

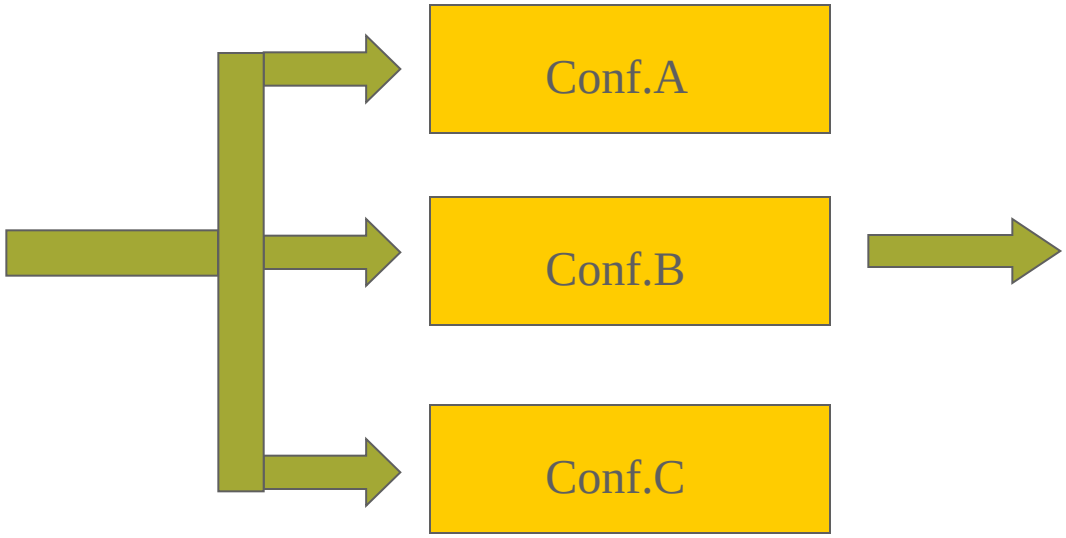
$$R(t) = \prod_{i=1}^{N} R_i(t)$$

Por ejemplo: R = 0,95 * 0,94 * 0, 99 = 0,884

La confiabilidad total de un sistema en serie siempre será menor que la confiabilidad de cualquiera de sus componentes.

Confiabilidad en Paralelo

La Confiabilidad de un conjunto de equipos en paralelo

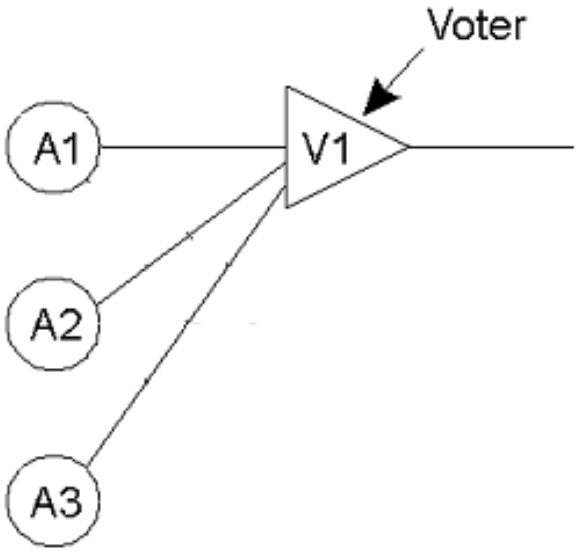


$$R(t) = 1 - \prod_{i=1}^{N} (1 - R_i(t))$$

Por ejemplo: R = 1 - (0,05 * 0,06 * 0, 01) = 0,99997

Redundancia – Uso de voters

Un “Voter” recibe entradas y replica en la salida el valor que tenga la mayoría de sus entradas

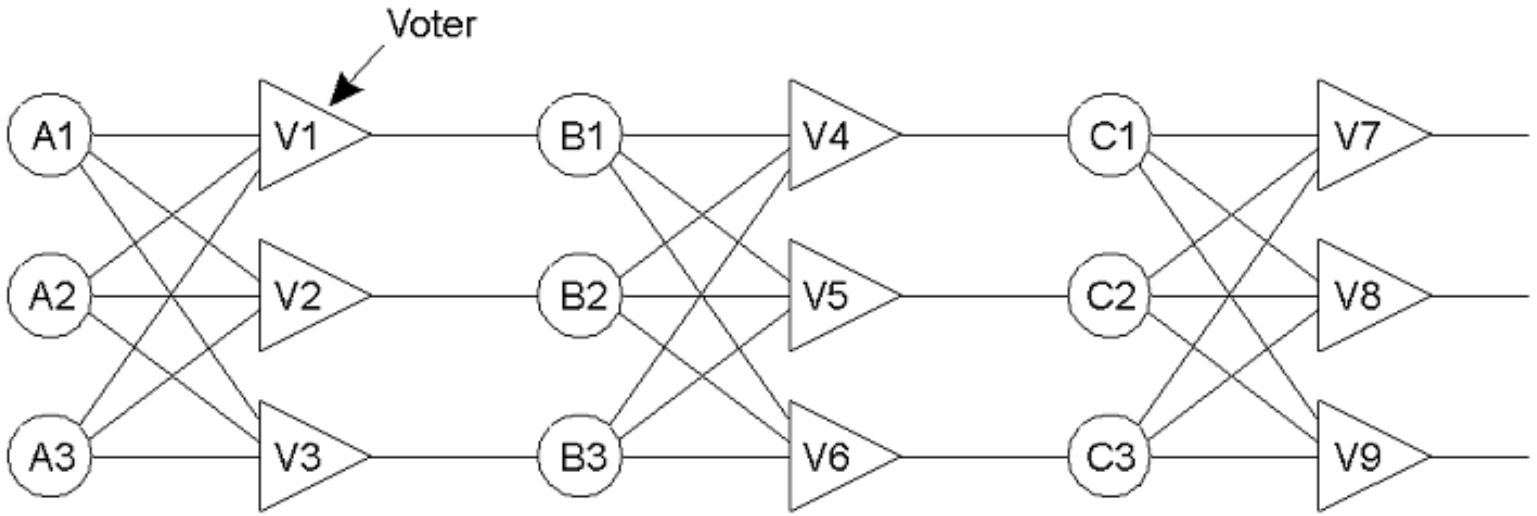


Por lo tanto, como las unidades A1, A2 y A3 son exactamente iguales, deberían dar los mismos valores a la entrada del Voter.

¿ Y si falla el Voter ? [SPOF]

Redundancia de voters

Sistema de 3 unidades (con 3 módulos cada uno) y replicación de "Voters"



Ejemplo 1: Falla la entrada A2. V1, V2 y V3 tienen 2 entradas iguales y una diferente. Por lo tanto la salida de los mismos será correcta y llegará bien a B1, B2 y B3.

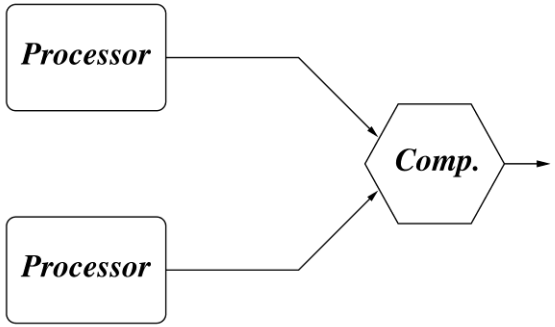
Ejemplo 2: Falla V1. La entrada de B1 será incorrecta, pero B2 y B3 producirán la misma salida, por lo tanto V4, V5 y V6 tendrán la misma entrada.

Redundancia – Sistemas Duplex

Los dos procesadores ejecutan lo mismo.

Si las salidas coinciden se asume que el resultado es correcto.

Si hay diferencia: ¿ Cuál falló ?

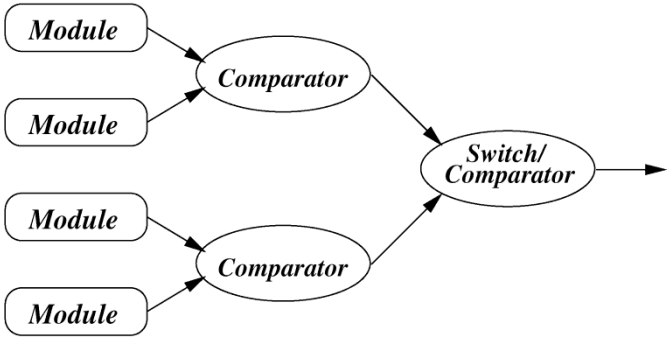


Soluciones posibles:

1 - Test de ambos procesadores.

2 – Utilizar un tercer procesador para determinar el resultado correcto

3 – Implementar un Sistema Duplex de backup.



Arquitecturas y Confiabilidad

- Controlar cuidadosamente inter-dependencias externas del componente
  - Los cambios de comportamiento de un componente en particular, incluyendo anomalías, deberían tener mínimo impacto en el comportamiento del sistema en general
  - Restringir todas las dependencias inter-componente a ser explicitas a través de sus interfaces públicas, minimizando “efectos colaterales ocultos”

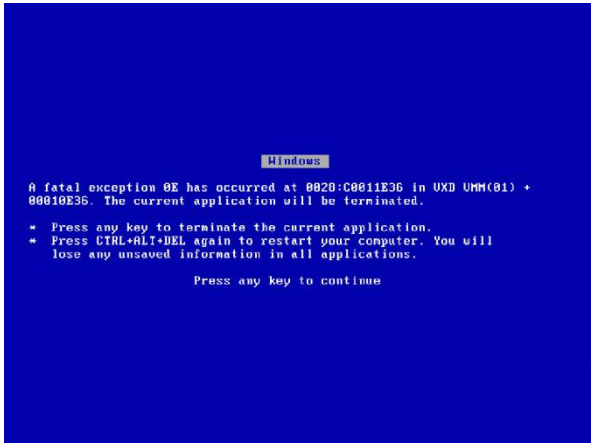
Más tácticas

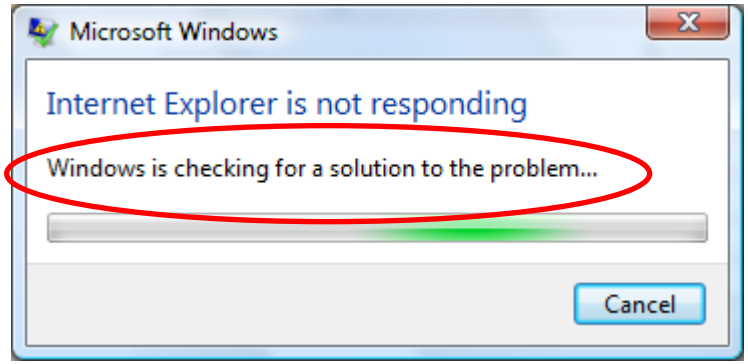
- Proveer capacidades de *reflection a los componentes*
  - En caso que un componente reduzca o interrumpa su servicio, debe ser posible monitorear su estado interno de modo de comprender la salud del mismo.
  - Proveer mecanismos públicos de manejo de excepciones
  - Especificar invariantes de estado de los componentes
- Emplear conectores que explícitamente controlen inter-dependencias entre componentes
  - Evitar memoria compartida si es posible, cuidado con mecanismos de invocación implícita

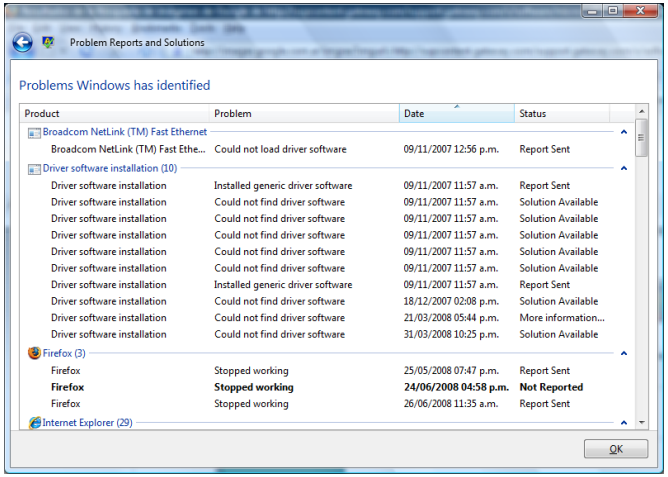
Más tácticas - Sistemas Self-Healing – Desafíos

- Los sistemas, cada vez más:
  - Integran funcionalidades heterogéneas
  - Deben correr continuamente
  - Soportan cambios en recursos
  - Son usados por usuarios móviles
- Algunas de las prácticas actuales dejarán de ser suficientes:
  - Verificación demasiado compleja
  - Mejoras “off-line” pueden no ser una opción
  - Reconfiguración manual puede ser inaceptable

Tendencia: Computación Autónoma. Veamos esta evolución







Sistemas Self-Healing - Definición

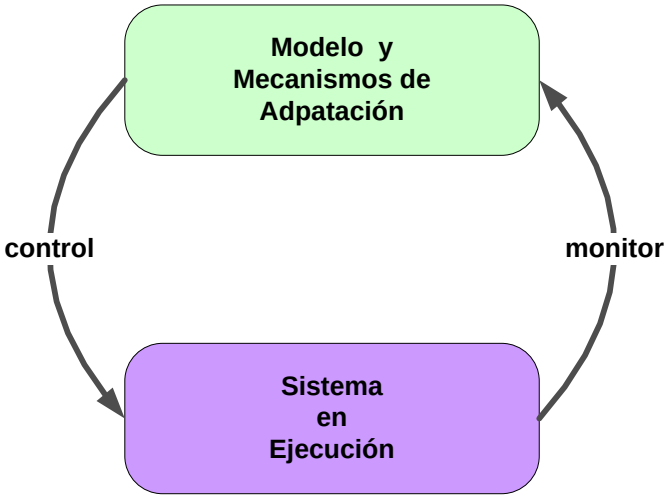
- Tener la posibilidad o propiedad de "curarse" a si mismo
- Self-healing describe cualquier dispositivo o sistema que tiene la habilidad de percibir que no está funcionando correctamente y, sin intervención humana, hacer los ajustes necesarios para volver a operación normal
- "Self healing" es el término más usado en los ámbitos académicos. Otros términos que expresan el mismo concepto o similares son "Autonomic Computing" (IBM, self-configuring, self-healing, self-optimizing y self-protecting), "Dynamic Systems", y "Conscientious Software"

Sistemas Self-Healing - Objetivos

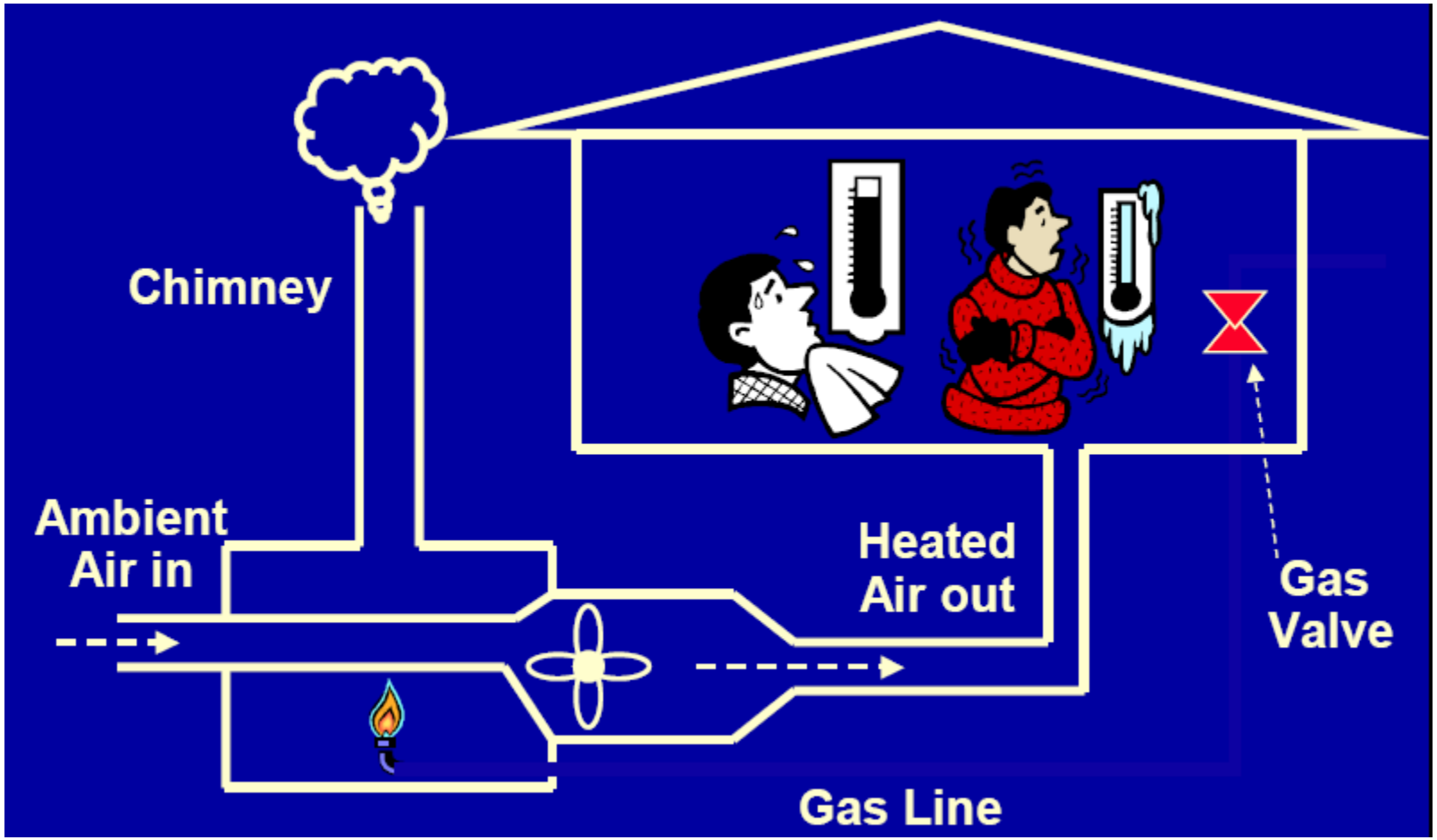
- Sistemas que automáticamente se adapten para manejar:
  - Cambios en los requerimientos (Atributos de calidad)
  - Recursos que cambian (Entorno Operativo)
  - Falla
  - Temas relacionados con Movilidad
  - Ubicuidad
- Cómo? Pasando de sistemas Closed Loop a Sistemas Open Loop

Sistema de control Closed Loop

- El sistema tiene modelos que funcionan dentro de él
- Estos modelos sirven para monitoreo, detección de problemas, y solución de problemas
- Esos modelos generan acciones que pueden cambiar el comportamiento del sistema en “run time”
- Se debe modelar explícitamente el contexto y lo que el usuario quiere hacer:
  - Adaptar la detección del problema a esa situación

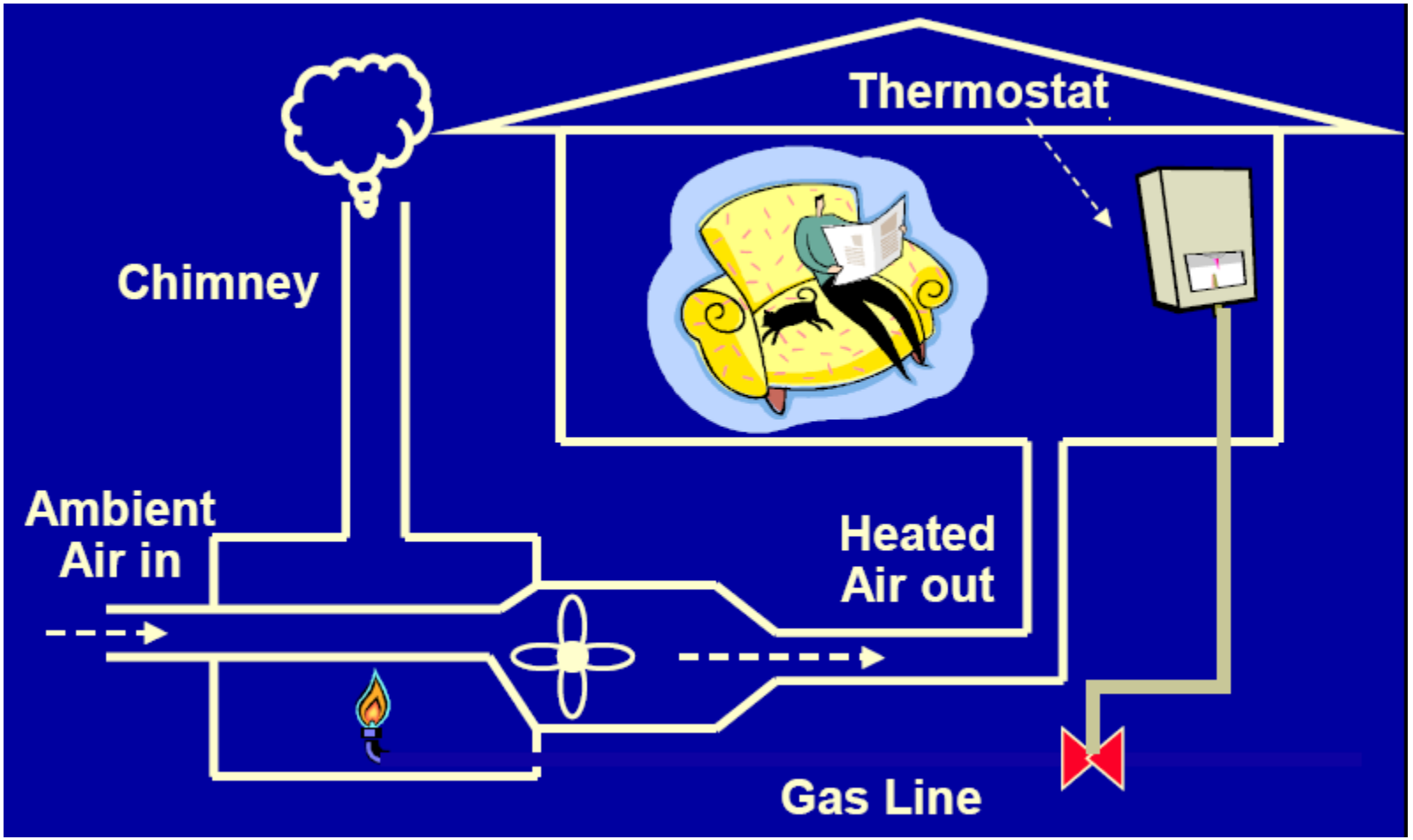


Ejemplo sistema de control Open Loop



Fuente: David Garlan. *Software – Heal Thyself! <<UML>> 2002*

Ejemplo sistema de control Closed Loop

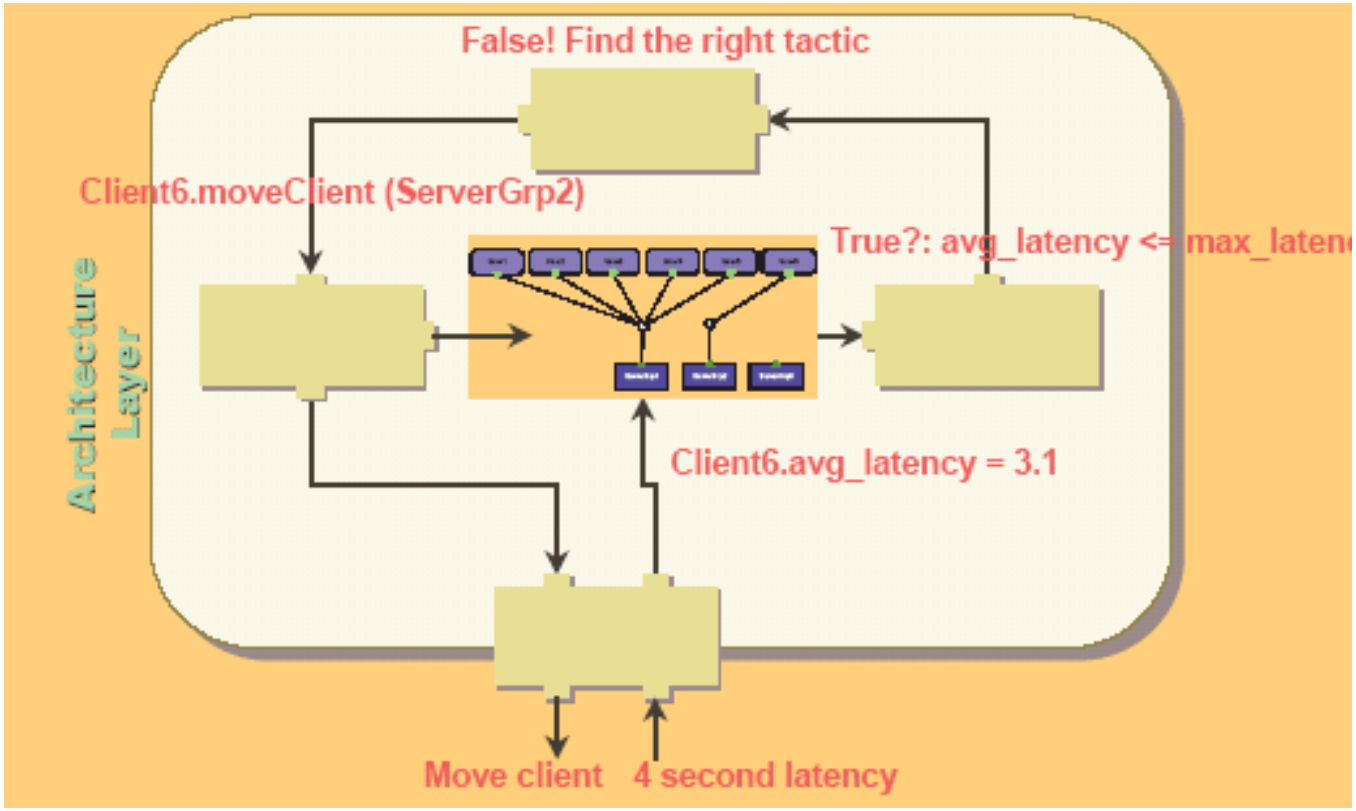


Fuente: David Garlan. *Software – Heal Thyself! <<UML>> 2002*

Requerimientos de Self Healing Systems

- **Monitoreo**: observar el sistema funcionando y hacer una abstracción del comportamiento observado.
- **Detección**: continuamente revisar restricciones de diseño a través de modelos de run time explícitos
- **Resolución**: determinar la causa de la violación de una restricción y elegir una estrategia de reparación.
- **Adaptación**: adaptar el sistema utilizando estrategias de cambio verificadas.

Un Ejemplo de ciclo de reparaciones



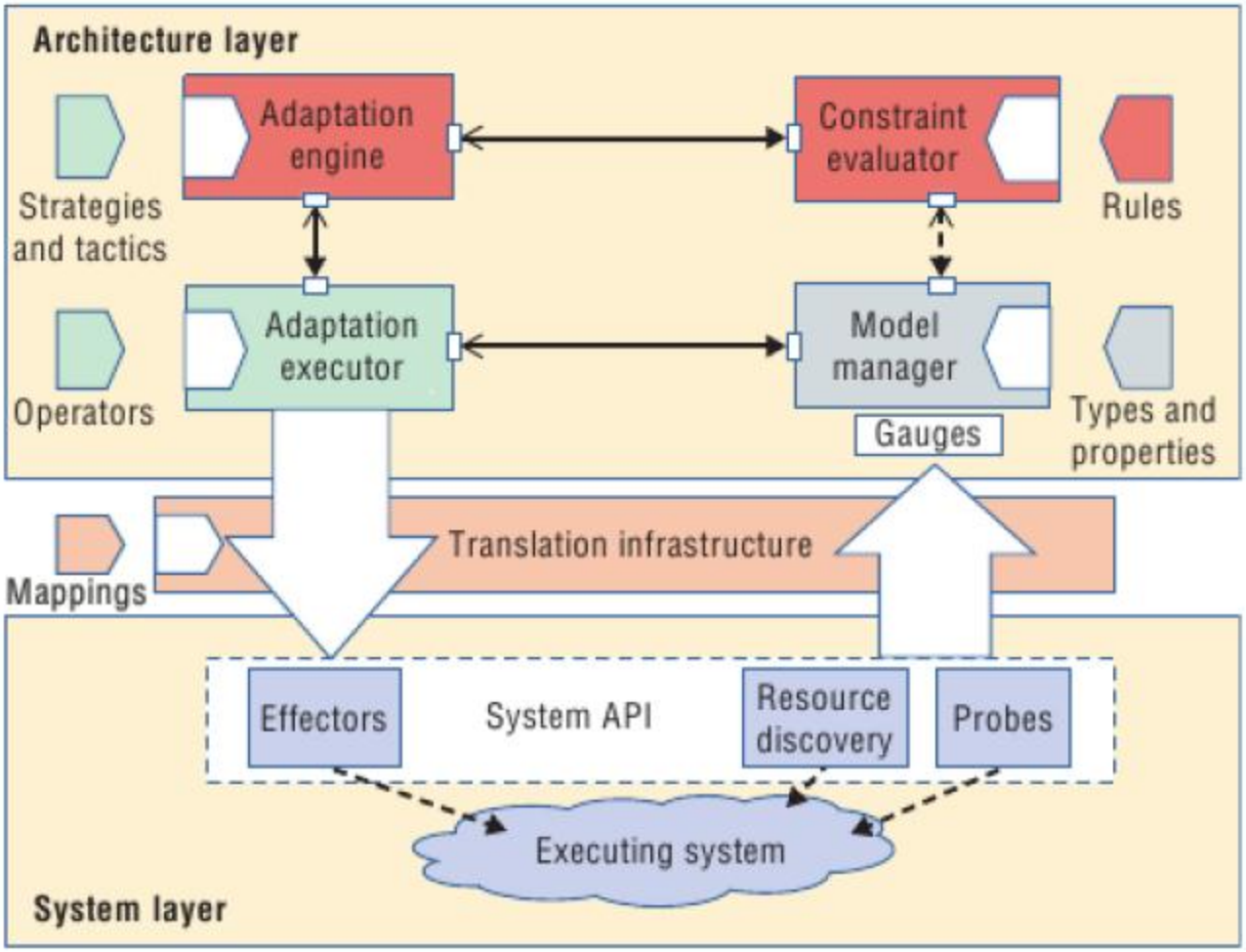
Fuente: David Garlan. *Software – Heal Thyself! <<UML>> 2002*

Nuevas Capacidades del Software

- "Reflection"
  - Capacidad de entender estado actual
- "Self Adaptation"
  - Capacidad para adaptarse con recursos que cambian, necesidades del usuario, fallas
- Context Awareness
  - Capacidad para reconocer cambios en el contexto para elegir estrategias
- Orientación a tareas
  - Sensibilidad a la intención del usuario

Relación con Arquitecturas: Raibow

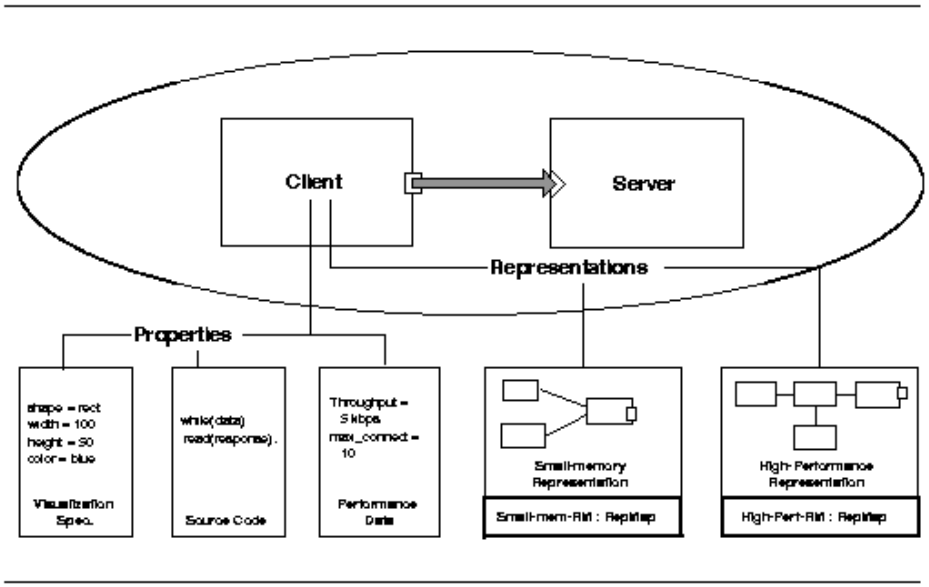
Una arquitectura de referencia para sistemas tipo "Self Healing"



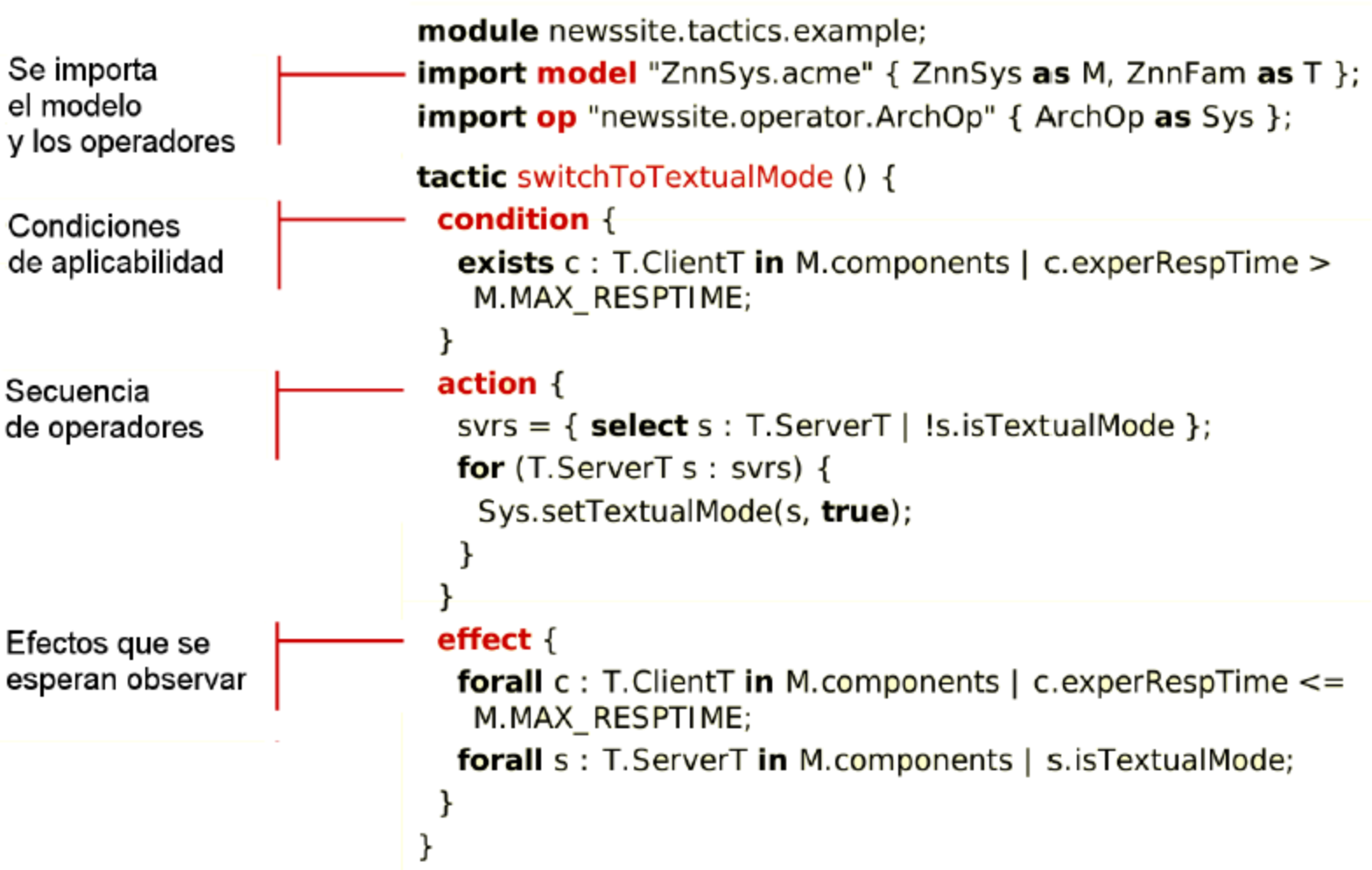
Fuente: David Garlan. *Software – Heal Thyself! <<UML>> 2002*

Ejemplo de ADL – ACME

```
System simple_cs = {
  Component client = {
    Port send-request;
    Property Aesop-style : style-id = client-server;
    Property UniCon-style : style-id = client-server;
    Property source-code : external = "CODE-LIB/client.c";
  }
  Component server = {
    Port receive-request;
    Property idempotence : boolean = true;
    Property max-concurrent-clients : integer = 1
    source-code : external = "CODE-LIB/server.c";
  }
  Connector rpc = {
    Role caller;
    Role callee;
    Property asynchronous : boolean = true;
    max-roles : integer = 2;
    protocol : Wright = " ... ";
  }
  Attachment client.send-request to rpc.caller;
  Attachment server.receive-request to rpc.callee;
}
```



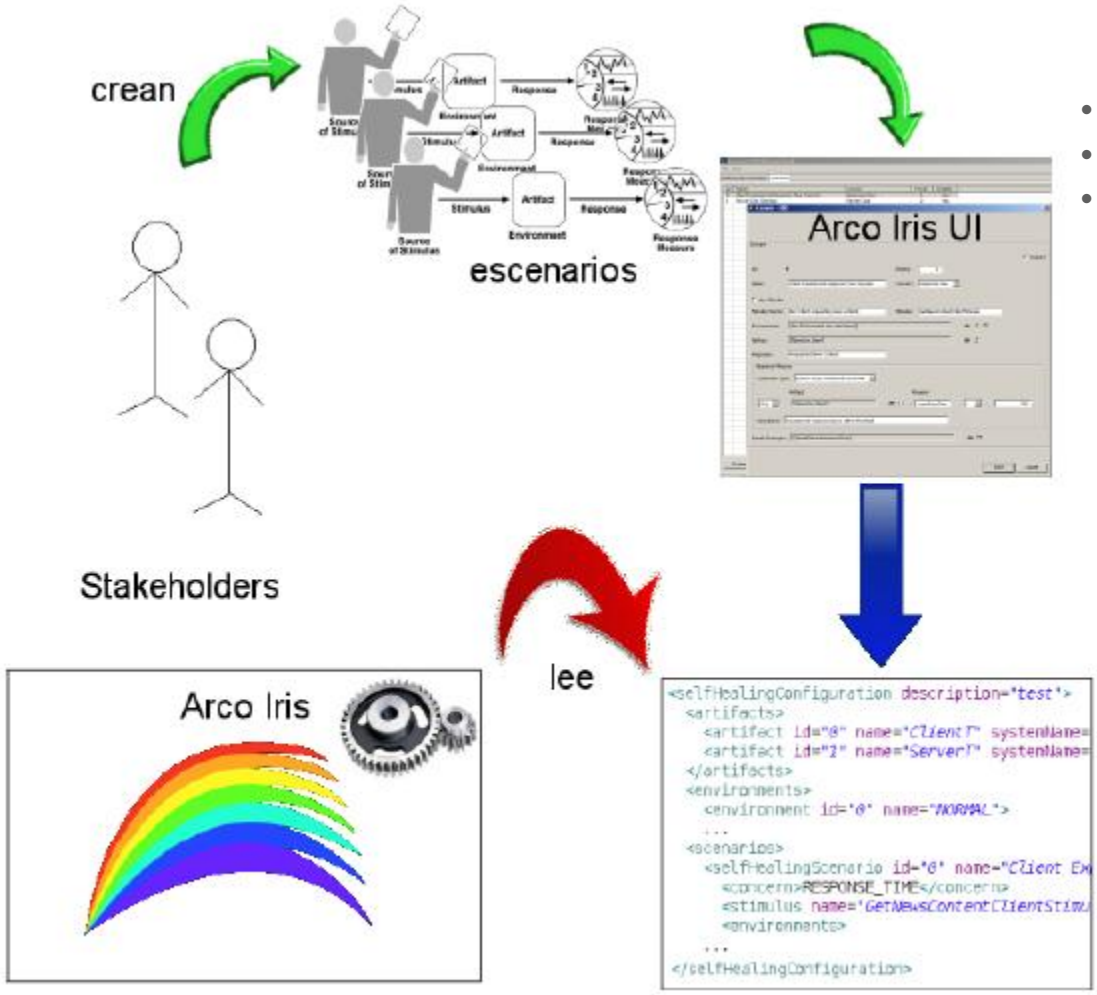
Ejemplo Stitch




```
module newssite.tactics.example;
import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
import op "newssite.operator.ArchOp" { ArchOp as Sys };

tactic switchToTextualMode () {
  condition {
    exists c : T.ClientT in M.components | c.experRespTime >
      M.MAX_RESPTIME;
  }
  action {
    svrs = { select s : T.ServerT | !s.isTextualMode };
    for (T.ServerT s : svrs) {
      Sys.setTextualMode(s, true);
    }
  }
  effect {
    forall c : T.ClientT in M.components | c.experRespTime <=
      M.MAX_RESPTIME;
    forall s : T.ServerT in M.components | s.isTextualMode;
  }
}
```

Una arquitectura con escenarios



- Escenarios con prioridades
- Utility Function
- Relación escenario – estrategia